MallaReddyEngineeringCollege
AnUGCAutonomousInstitution,ApprovedbyAICTE,NewDelhi&Affiliated to JNTUH, Hyderabad, Accredited by NAAC with ‘A++’ Grade (3rd Cycle), Maisammaguda
(H),Medchal-Malkajgiri,SecunderabadTelangana–500100 www.mrec.ac.in

Department of Information Technology

II B.TECH I SEM (A.Y.2024-25)

LectureNotes

On

C0510-DataStructures

2022-23 Onwards (MR22)	MALLAREDDYENGINEERINGCOLLEGE (Autonomous)	B.Tech. IIISemester		
Code: C0510	DataStructures	L	T	P
Credits:3		3	-	-

Prerequisites: Acourseon“ProgrammingforProblemSolving“

CourseObjectives:

- Exploringbasicdatastructuressuchaslinkedlist,stacksandqueues.
- Introducesavarietyofdatastructuressuchasdictionariesandhashtables
- Tolearnnonlineardatastructuresi.e.Binarysearchtreesandheightbalancedtrees.
- Tounderstandthegraphtraversalalgorithmsandheapsort.
- Introducesthepatternmatchingandtriesalgorithms

Module-I: [10Periods]

Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation,insertion,deletionandsearching operationsonlinearlist,Stacks-Operations,array and linked representations of stacks, stack applications, Queues-operations, array and linked representations.

Module-II: [09Periods]

Dictionaries:linearlistrepresentation,skiplistrepresentation,operations-insertion,deletionand searching.

HashTableRepresentation:hashfunctions,collisionresolution-separatechaining,open addressing-linearprobing,quadraticprobing,doublehashing,rehashing,extendiblehashing.

Module-III: [10Periods]

SearchTrees: BinarySearchTrees,Definition,Implementation,Operations-Searching, InsertionandDeletion,AVLTrees,Definition,HeightofanAVLTree,Operations–Insertion, Deletion and Searching, Definition and example of Red–Black, Splay Trees.

Module-IV: [10Periods]

Graphs: GraphImplementationMethods.GraphTraversalMethods.

Sorting: MaxHeap,MinHeap,HeapSort.ExternalSorting:Modelforexternalsorting,Mergesort.

Module-V: [09Periods]

PatternMatchingandTries: Patternmatching algorithms-Bruteforce,theBoyer–Moorealgorithm,theKnuth-Morris-Prattalgorithm,StandardTries,CompressedTries,Suffix tries.

Text Books:

1. JeanPaulTremblay,PaulGSorenson,“AnIntroductiontoDataStructureswithApplications”, Tata McGraw Hills, 2nd Edition, 1984.
2. RichardF.Gilberg,BehrouzA.Forouzan,“DataStructures:A
3. PseudocodeapproachwithC",Thomson(India),2ndEdition,2004.

References:

1. Horowitz,Ellis,Sahni,Sartaj,Anderson-Freed,Susan,“FundamentalsofDataStructurein C”, University Press (India), 2nd Edition, 2008.
2. A.K.Sharma,“DatastructuresusingC”,Pearson,2ndEdition,June,2013.
3. R.Thareja,“DataStructuresusingC”,OxfordUniversityPress,2ndEdition,2014.

E-Resources:

1. http://gvpcse.azurewebsites.net/pdf/data.pdf
2. http://www.sncwgs.ac.in/wp-content/uploads/2015/11/Fundamental-Data-Structures.pdf
3. http://www.learnerstv.com/Free-Computer-Science-Video-lectures-ltv247-Page1.htm
4. http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qw
   H9xY7-
   3lcmoMApVUMmjlExpIb1zste4YXX1pSpX8a2mLgD
   zZ-E41CJ6PVmY4S0MqVbxsFQ
5. http://nptel.ac.in/courses/10610

2064/1**CourseOutcomes:**

Attheendofthecourse,studentswillb
eableto

COs	CourseOutcome	Bloom's Taxonomy Level
CO1	**Implement**thelineardatastructuressuchaslinkedlist,stacksandqueues	Understand
CO2	**Understand** theDictionariesandHashtablerepresentation	Understand
CO3	**Analyze**thevariousnonlineardatastructureswithitsoperations	Analyze
CO4	**Develop**theprogramsbyusingGraphTraversalandheapsort	Understand
CO5	**Apply**datastructureconceptsforthe implementationofpattern matching andtries	Apply

CO-PO,PSOMapping (3/2/1indicatesstrengthofcorrelation)3-Strong,2-Medium,1-Weak															
COs	ProgrammeOutcomes(POs)												PSOs		
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**
O1	2	3	2										2	3	
CO2	2	2	3										3	2	
CO3		2	2											2	1
CO4		2	3										2	3	
CO5	2	3	3										2	3	

MODULE-1
DATA STRUCTURES

MODULE-I:

Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation, insertion, deletion and searching operations on linear list, Stacks-Operations, array and linked representations of stacks, stack applications, Queues-operations, array and linked representations.

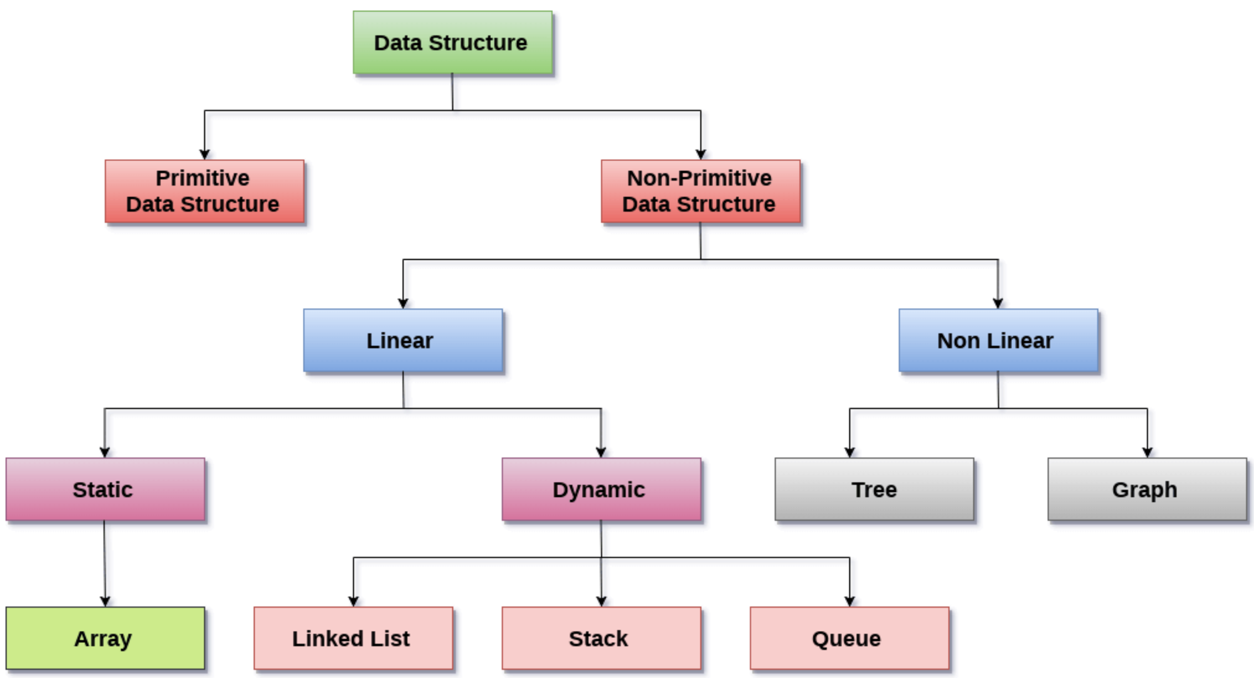
Introduction:

.Datastructureisacollectionoforganizeddatainthememorylocations. Data structure can be classified as

.LinearDatastructure:oneelementis connectedtoanotherelement inlinearform .

.Inthelineardatastructurevaluesarearrangedinalinearfashion.Anarray,linkedlist,stacks and queues are Examples of linear data structure.

TypesofDataStructures:



Therearetwotypesofdatastructureavailablefortheprogramming purpose:

- **Primitivedatastructure**
- **Non-primitivedatastructure**

Primitivedatastructure

Primitive data structure is a data structure that can hold a single value in a specific location whereas the non-linear data structure can hold multiple values either in a contiguous location or random locations

The examples of primitive data structure are float, character, integer and pointer. The value tothe primitive data structure is provided by the programmer. The following are the four primitive data structures:

- **Integer:** The integer data type contains the numeric values. It contains the whole numbers that can be either negative or positive. When the range of integer data type isnot large enough then in that case, we can use long.
- **Float:** The float is a data type that can hold decimal values. When the precision of decimalvalue increases then the Double data type is used.
- **Boolean:** It is a data type that can hold either a True or a False value. It is mainly usedfor checking the condition.
- **Character:** It is a data type that can hold a single character value both uppercase and lowercase such as 'A' or 'a'.

Non-primitivedatastructure

The non-primitive data structure is a kind of data structure that can hold multiple values either in a contiguous or random location. The non-primitive data types are defined by the programmer. The non-primitive data structure is further classified into two categories, i.e., linear and non-linear data structure.

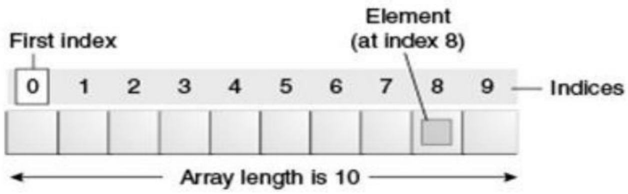
LinearDataStructures:

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each elementhas the successors and predecessors except the first and last element.

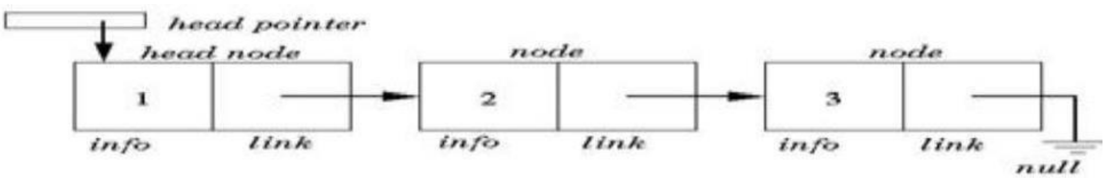
TypesofLinearDataStructuresaregivenbelow:

Arrays: An array is a collection of similar type of data items and each data item is called an elementofthearray.Thedatatypeoftheelementmaybeanyvaliddatatypelikechar,int, floator double.

- Theelementsofarraysharethesamevariablenamebuteachonecarriesadifferentindex number known as subscript. The array can be one dimensional, two dimensional or multidimensional.
- Theindividualelements ofthearrayageare:
- age[0],age[1],age[2],age[3],age[98],age[99].

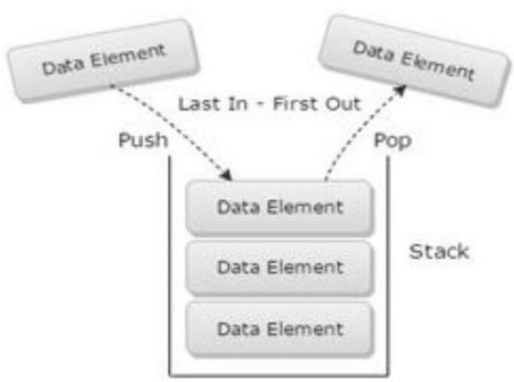


Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memorylocations. Each node of the list contains a pointer to its adjacent node.



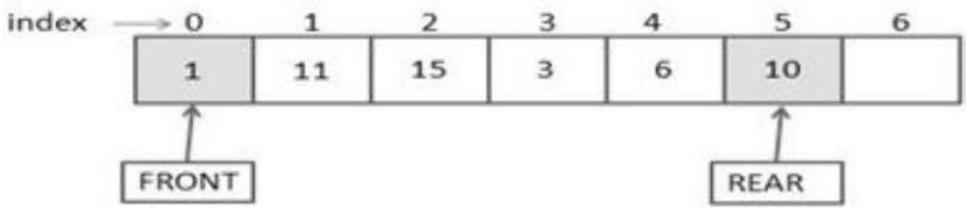
Stack: Stack is a linear list in whichinsertion and deletions are allowed only at one end, called **top**.

- A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.



Queue: Queueisalinearlistinwhichelementscanbeinsertedonlyatoneend called **rear** and deleted only at the other end called **front**.

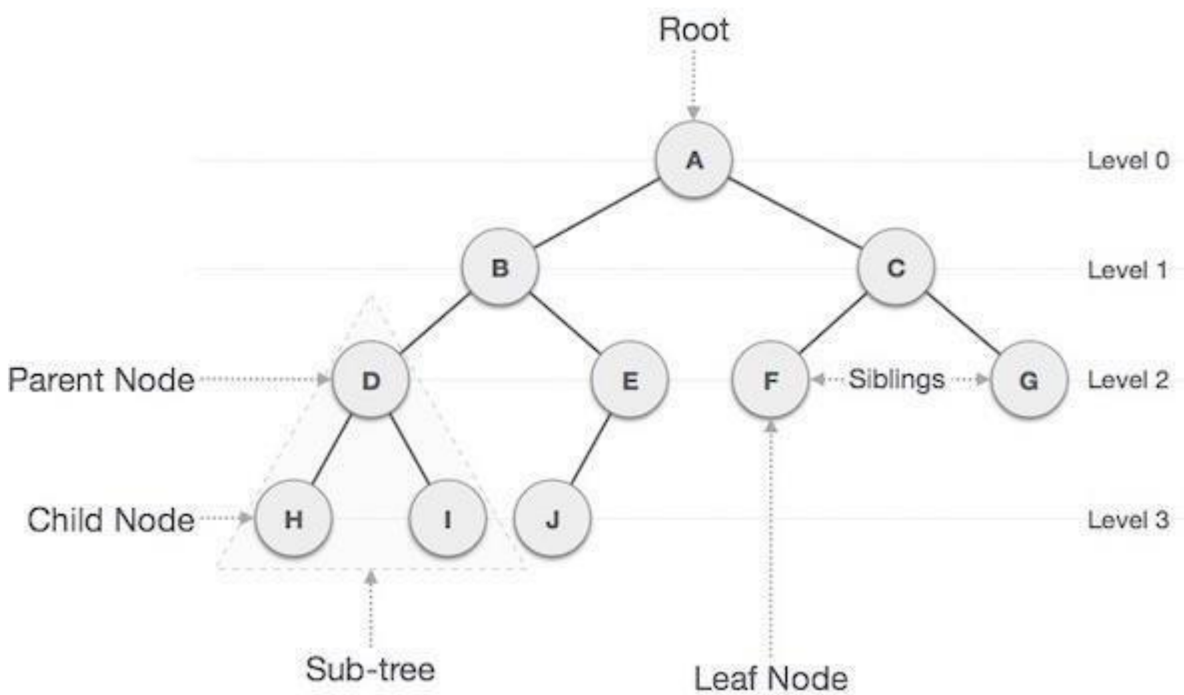
- Itisanabstractdatastructure,similartostack.Queueisopenedatbothendthereforeit followsFirst-In-First-Out(FIFO)methodologyfor storingthedataitems.



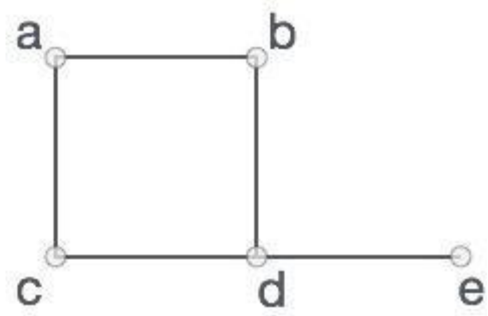
Non Linear Data Structures: Anon-lineardatastructureisanotherimportanttypeinwhich data elements are not arranged sequentially .mainly data elements arranged in random order. **.**nonlineardatastructuresoneelementisconnectedmultipleelements. Types

of Non Linear Data Structures are given below:

Trees: Atreeisanon-linearabstractdatatypewithahierarchy-basedstructure.Itconsistsofnodesthatare connected via links. The tree data structure stems from a single node called a root node and has subtrees connected to the root.



Graphs: Agraphisa non-linear kindofdatastructure madeup of nodes or vertices and edges.The edges connect any two nodes in the graph, and the nodes are also known as vertices.



Intheabovegraph, V

= {a, b, c, d, e}

E={ab,ac,bd,cd, de}

.ifthereisnoedgebetweentwonodesthepresencevalue0

.ifthereisanedgebetweentwonodesthepresencevalue1

.AbstractDataTypes:

An ADT is a theoretical construct that consist of data as well as the operations to be performed on data to implementation.

. data

.operations–insertion,deletionand list

. implementation

. Errors

ADTexamplesaresack,queuesandlinkedlist

.Abstractdatatypescan beclassified as

1. **List ADT:**Listsarelineardatastructuresstoredinanon-continuousmannerthelistismade up of a series of connected nodes that are randomly stored in the memory.

   .Hereeachnodeconsists oftwopartsthefirstpartisthedataandthesecondpartcontainsthe pointer to the address of the next node.

   TheListADTFunctionsisgivenbelow:

   - get()–Returnanelementfrom thelistatanygivenposition.
   - insert()–Insertanelementatanypositionofthelist.
   - remove()–Removethefirstoccurrenceofanyelementfromanon-emptylist.
   - removeAt()–Removetheelementataspecifiedlocationfromanon-emptylist.
   - replace()–Replaceanelementatanypositionbyanotherelement.
   - size()–Returnthenumberofelementsinthelist.

- isEmpty()–Returntrueifthelistisempty,otherwisereturnfalse.
- isFull()–Return trueifthelistisfull,otherwisereturnfalse.

2. Stack ADT:Astackisan orderedlistandwhichelementsareinsertedanddeletedatonlyone end called TOP of the stack.
   .stackisaLIFO Technique
   .Thestack ADToperationsas follows

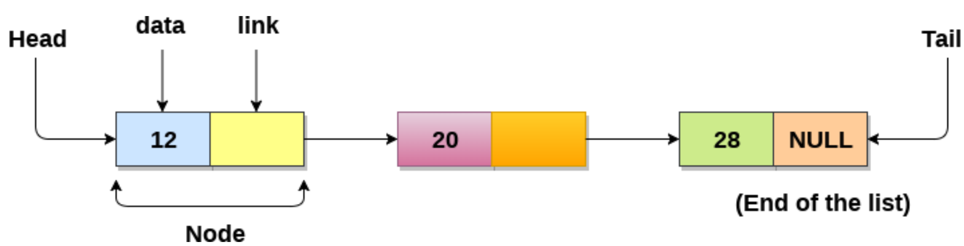
- push()–Insertanelementatoneendofthestackcalledtop.
- pop()–Removeandreturntheelementatthetopofthestack,ifitisnotempty.
- peek() – Returnthe element at the top ofthe stack without removing it, ifthe stack is notempty.
- size()–Returnthenumberofelementsinthestack.
- isEmpty()–Return trueifthestackisempty,otherwisereturnfalse.
- isFull()–Return trueif thestackisfull,otherwisereturnfalse.

3. Queue ADT:Aqueueisanorderedlistinwhichinsertionisdoneatoneendcalled REARand deletion at another end called FRONT.
   . The first inserted element is availablefirst for theoperations to be performed and is the first one to be deleted.
   .Henceitisknownasfirstinfirstout(FIFO) The queue ADT operations as follows

- enqueue()–Insertanelementattheendofthequeue.
- dequeue()–Removeandreturnthefirstelementofthequeue,ifthequeueisnotempty.
- peek()–Returntheelementofthequeuewithoutremovingit,if thequeueisnotempty.
- size()–Returnthenumberofelementsinthequeue.
- isEmpty()–Returntrueifthequeueisempty,otherwisereturnfalse.
- isFull()–Returntrueifthequeueisfull,otherwisereturnfalse.

Linked List: Anelementinalinkedlistisaspeciallytermednode.Anodeconsistoftwo fields data and link(address).

.Alinkedlistisanorderedcollectionoffinite,homogeneousdataelementscallednodes.where the linear order is maintained by means of links or pointers.

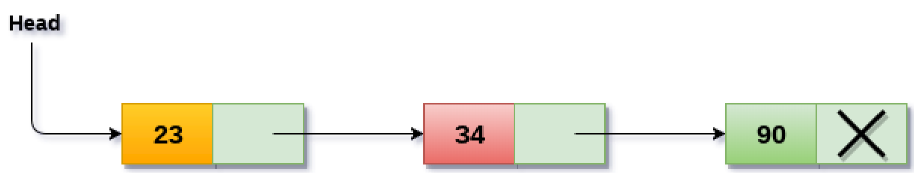


. **singlelinkedlist**:inasinglelinkedlisteach nodecontainsonlyonelinkwhichpointstothe subsequent node in the list.

.onewaychain or singlylinked listcan betraversedonly one direction.

UsesofLinkedList

- Thelistisnotrequiredto becontiguouslypresentinthememory.
- listsizeislimitedtothe memorysizeanddoesn'tneedtobedeclaredinadvance.
- Emptynodecannotbepresentinthelinkedlist.
- Wecanstorevaluesofprimitive typesorobjects inthesingly linkedlist.



NodeCreation:

```
structnode

{
   int data;
   structnode*next;
};
```

Advantages:Theyaredynamicinnaturewhichallocatesthememorywhen required.
.insertionand deletioncan beeasily implemented.
.stacksandqueuescanbeeasily executed.
.itreducesthe accesstime.

Disadvantages:Thememoryiswastedaspointersrequiredextramemoryfor storage.
.Eachnodehastoaccess sequentially.
.Reversetraversingis difficult.
Applicationsoflinkedlist :Linkedlistareusedtoimplementstack,queue,graphetc.
.linked listlet youinsert element atthebeginningand endofthe list.
.in linked list wedo not need to knowthe sizein advance.

SinglyLinkedList operations:

Therearevariousoperationswhichcanbeperformedonsinglylinkedlist.Alistofallsuch operations is given below.

Insertion

.Theinsertionintoasinglylinkedlistcanbeperformedatdifferentpositions.Basedon theposition ofthe new node being inserted, the insertion is categorized into the following categories

1. insertionat beginning
2.insertion at end of the list
3.insertionafterspecifiednode

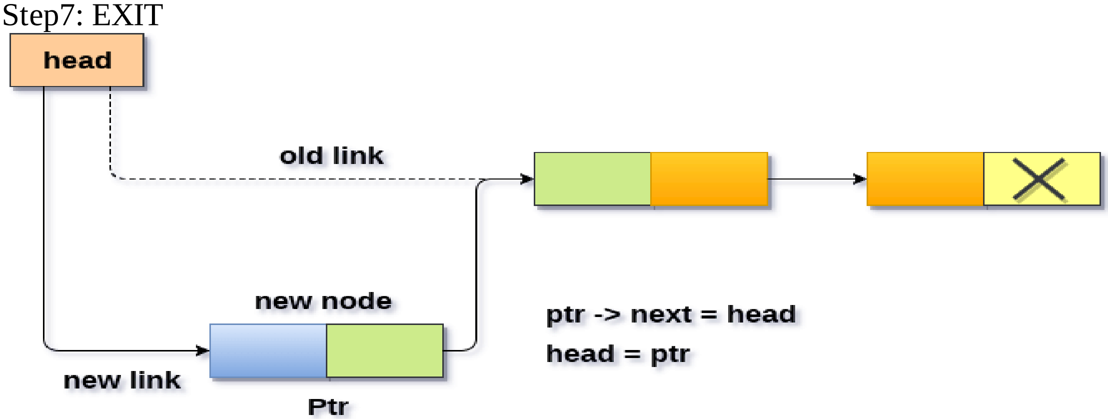
.insertionatbeginningoftheList:Toinsertanewelementatbeginningpositionandptrnode connected to next node and last node it indicates NULL.

Algorithm:

- Step1: IFPTR =NULL

  WriteOVERFLOW
  GotoStep7[END
  OF IF]

- Step2: SETNEW_NODE =PTR
- Step3: SETPTR=PTR→NEXT
- Step4:SETNEW_NODE→DATA= VAL
- Step5:SETNEW_NODE →NEXT= HEAD
- Step6:SETHEAD=NEW_NODE
- Step7: EXIT



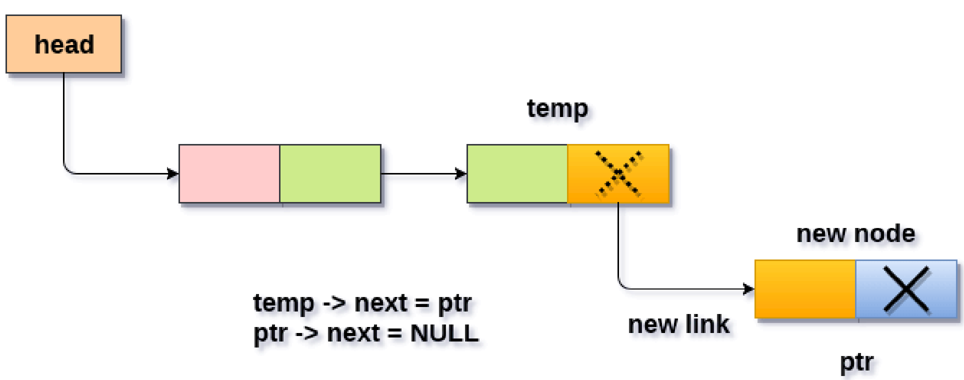
InsertingAtEndoftheList:Toinsertelementinlastnodeofthelisttheheadpreviouslastnodeslinkfield which was NULL.

Algorithm:

- **Step1:**IFPTR=NULL

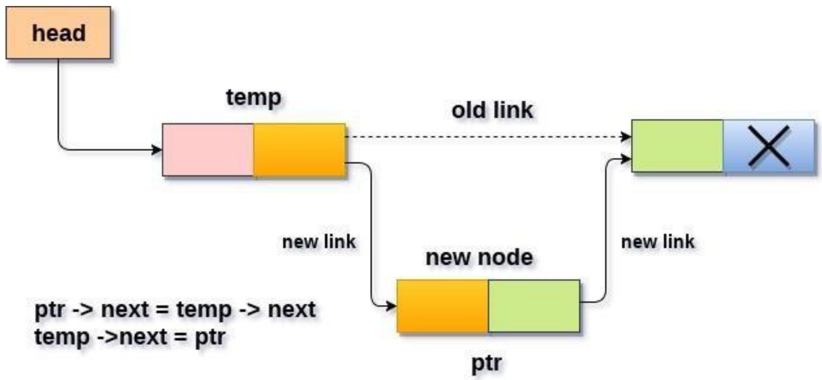
  WriteOVERFLO
  WGotoStep10

- **Step2:**SETNEW_NODE=PTR
- **Step 3:**SETPTR=PTR->NEXT
- **Step4:**SETNEW_NODE->DATA=VAL
- **Step5:**SETNEW_NODE- >NEXT=NULL
- **Step6:**SETPTR=HEAD
- **Step 7:**RepeatStep 8whilePTR ->NEXT !=NULL
- **Step8:**SETPTR=PTR- >NEXT[END OFLOOP]
- **Step 9:**SETPTR->NEXT=NEW_NODE
- **Step10:**EXIT



Inserting node at the last into a non-empty list

InsertingAfterspecifiednode:Inordertoinsertanelementafterthespecifiednumberofnodes in to the linked list we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted .



Algorithm:

- **STEP1:**IFPTR=NULL
  WRITEOVERFLO
  WGOTOSTEP12
  END OF IF
- **STEP2:** SETNEW_NODE=PTR
- **STEP3:**NEW_NODE → DATA=VAL
- **STEP4:**SETTEMP= HEAD
- **STEP5:**SETI=0
- **STEP6:**REPEAT STEP5AND6UNTIL I
- **STEP7:**TEMP=TEMP→ NEXT
- **STEP8:** IFTEMP=NULL

  WRITE"DESIREDNODENOT
  PRESENT"GOTO STEP 12
  END OF IF
  ENDOF
  LOOP
- **STEP9:**PTR→NEXT=TEMP →NEXT
- **STEP10:**TEMP→ NEXT=PTR
- **STEP11:**SETPTR=NEW_NODE
- **STEP12:**EXIT

Deletion:Thedeletionoperationisusedtodeleteanodefromthelistanditcanbeperformedat three different locations

. deletion of thefirst node

.deletion of thelast node
.deletionatthespecifiedposition

.**Deletionofthefirstnode**:Deletinganodefromthebeginningofthelististhesimplestoperationof all.since the first node of the list is to be deleted therefore we just need to make the head ,point to the next of the head. Now free the pointer ptr which was pointing to the head node of the list.

Algorithm:

Step1:IFHEAD=NULL

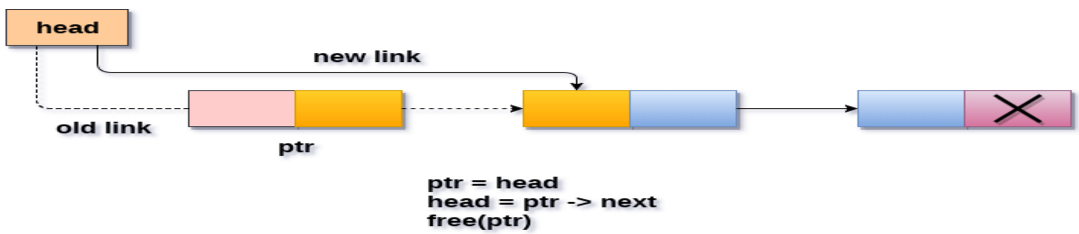
WriteUNDERFLOWGotoStep 5
[ENDOFIF]

Step2:SETPTR=HEAD

Step3:SETHEAD=HEAD->NEXT

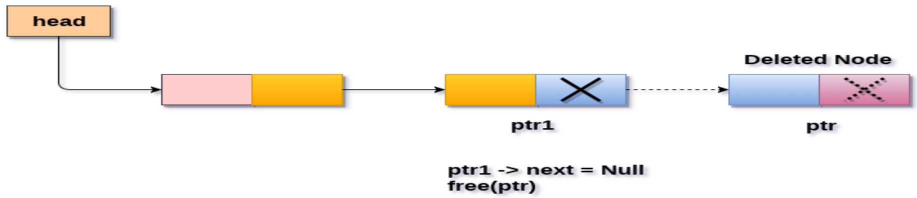
Step4:FREEPTR

Step5:EXIT



Deleting a node from the beginning

2. DeletionoftheLastnode:Therearetwoscenariosinwhichanodeisdeletedfromtheendofthe linked list
   .Thereis only onenodein the list andthat needs to bedeleted.
   .There aremorethan onenodein thelist and thelast nodeof thelist will be deleted.



Deleting a node from the last

Algorithm:

- **Step1:**IFHEAD=NULL

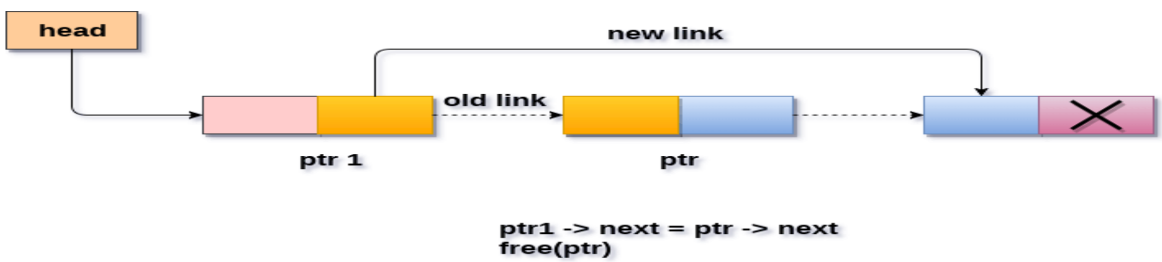
WriteUNDERFLOW
GotoStep8
[ENDOFIF]

- **Step2:**SETPTR=HEAD
- **Step3:**RepeatSteps4and5whilePTR->NEXT!=NULL
- **Step4:**SETPREPTR=PTR
- **Step5:**SETPTR=PTR-

  >NEXT[END OF LOOP]

- **Step6:**SETPREPTR->NEXT=NULL
- **Step7:**FREEPTR
- **Step8:**EXIT

3. **Deletionatthespecifiedposition:**To delete a nodefromthe singlylinkedlistbefore the specified position in a singly linked list.



Deletion a node from specified position

Algorithm:

STEP1:IFHEAD=NULLWRITE

UNDERFLOW
GOTOSTEP10
END OF IF

STEP2:SETTEMP=HEAD

STEP3:SETI=0

STEP4:REPEATSTEP5TO8UNTILI<loc<li=""></loc<>

STEP5:TEMP1=TEMP

STEP6:TEMP=TEMP→NEXT

STEP7:IFTEMP=NULL

WRITE"DESIREDNODENOTPRESENT"
GOTOSTEP11
ENDOFIF

STEP8:

I=I+1ENDOFLO

OP

STEP9:TEMP1→NEXT=TEMP→NEXT

STEP10:FREETEMP

STEP11:Exit

Searching in singly linked list : Searching is performed in order to find the location of a particular element in the list .searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element.

. if the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm:

- **Step1:**SETPTR =HEAD
- **Step2:**Set I=0
- **STEP3:**IFPTR= NULL

  WRITE"EMPTYLIST"
  GOTOSTEP8
  ENDOFIF

- **STEP4:**REPEATSTEP5TO7UNTIL PTR!= NULL
- **STEP5:**if ptr→ data = item

  Writei+1
  EndofIF

- **STEP 6:**I=I+1
- **STEP7:**PTR=PTR→NEXT

  [END OF LOOP]

- **STEP8:**EXIT

Traversing in singly linked list : Traversing is the most common operation that is performed in almost every scenario of singly linked list .Traversing means visiting each node of the list once in order to perform some operation on that .

Algorithm:

STEP1:SETPTR=HEAD

STEP2:IFPTR=NULL

WRITE"EMPTYLIST"
GOTOSTEP6
STEP3:REPEATSTEP5AND6UNTILPTR!=NULL

STEP 4: PRINT PTR→

DATA**STEP5:**PTR = PTR →

NEXT**STEP6:**EXIT

Writeacprogram tosinglylinkedlistimplementation andoperations

a)creation b)insertion c)deletiond)Traversal

```
#include<stdio.h>
```

```
#include<stdlib.h>

structnode{
int data;
structnode*next;
}*head=NULL;

intcount()
{
structnode*temp; int
i=1;
    temp=head;
while(temp->next!=NULL)
{
        temp=temp->next; i++;
}
return(i);
}

structnode*create(intvalue)
{
structnode*temp;
    temp=(structnode*)malloc(sizeof(structnode));
    temp->data=value;
    temp->next=NULL;
```

```
{
structnode*newnode;
     newnode=create(value);
if(head==NULL)
{
          head=newnode;
}
else
{
          newnode->next=head; head=newnode;

}
}

voidinsert_end(intvalue)
{
structnode*newnode,*temp;
     newnode=create(value);
if(head==NULL)
{
          head=newnode;
}
else
{
          temp=head;
while(temp->next!=NULL)
{
               temp=temp->next;
}
          temp->next=newnode;
}
}

voidinsert_pos(intvalue,intpos)
{
structnode*newnode,*temp1,*temp2; int
i,c=1;
     newnode=create(value);
     i=count();
if(pos==1)
insert_begin(value);
elseif(pos>i+1)
{
printf("insertion is not posible"); return;
}
else
{
          temp1=head;
while(c<=pos-1&&temp1!=NULL)
```

```
{
        temp2=temp1;
        temp1=temp1->next;
        c++;
}
    newnode->next=temp2->next;
    temp2->next=newnode;
```

```
}
}

voiddelete_begin()
{
structnode*temp;
if(head==NULL)
{
printf("deletionisnotpossible");
}
else
{
        temp=head;
        head=head->next;
free(temp);
}
}
voiddelete_end()
{
structnode*temp1,*temp2;
if(head==NULL)
{
printf("deletionisnotpossible");
}
else
{
        temp1=head;
while(temp1->next!=NULL)
{
             temp2=temp1;
             temp1=temp1->next;
}
        temp2->next=NULL;
free(temp1);
}
}
voiddelete_pos(intpos)
{
structnode*temp1,*temp2;
int i,c=1;
    i=count();
if(pos==1)
delete_begin();
elseif(pos>i)
{
printf("Deletion is not posible"); return;
}
else
{
```

```
        temp1=head;
while(c<=pos&&temp1->next!=NULL)
{
            temp2=temp1;
            temp1=temp1->next;
            c++;
}
```

```
		temp2->next=temp1->next;
free(temp1);
}
}
voiddisplay()
{
structnode*temp;
if(head==NULL)
{
printf("listisempty");
}
else
{
		temp=head;
while(temp->next!=NULL)
{
printf("%d->",temp->data);
			temp=temp->next;
}
printf("%d",temp->data);
}
}

voidmain()
{
intch,pos,value; do
{
printf("¥n1.InsertBegin¥n2.InsertEnd¥n3.InsertPosition¥n4.DeleteBegin¥n5.DeleteEnd¥n6
.Delete Position¥n7.Display¥n8.Exit¥n");
printf("enter your choice:");
scanf("%d",&ch);
switch(ch)
{
case1:printf("enter the value:");
scanf("%d",&value);
insert_begin(value);
break;
case2:printf("enter value:");
scanf("%d",&value);
insert_end(value);
break;
case3:printf("enter value:");
scanf("%d",&value);
printf("enter position you want to insert: ");
scanf("%d",&pos);
insert_pos(value,pos);
break;
case4:delete_begin();
break;
```

```
case5:delete_end();
break;
case6:printf("enter position you want to delete: ");
scanf("%d",&pos);
delete_pos(pos);
break;
case7:display();
```

```
break;
case8:break;
default:printf("¥nyourchoiceiswrong!..");
}
}while(ch!=8);
}
```

DifferencebetweenArrayandLinkedList

S.No.	ARRAY	LINKED LIST
1.	An array is a grouping of dataelementsofequivalent data type.	A linked list is a group of entitiescalledanode.The node includes two segments: data and address.
2.	Itstoresthedataelements in a contiguous memory zone.	It stores elements randomly, orwe can say anywhereinthememory zone.
3.	In the case of an array, memory size is fixed, and itisnotpossibletochange it during the run time.	In the linked list, the placementofelementsis allocated during the run time.
4.	The elements are not dependentoneachother.	The data elements are dependentoneachother.
5.	Thememoryisassignedat compile time.	Thememoryisassignedat run time.
6.	It is easier and faster to accesstheelementinan array.	Inalinkedlist,theprocess of accessing elements takes more time.
7.	Inthecaseofanarray, memory utilization is ineffective.	In the case of the linked list,memoryutilizationis effective.
8	When it comes to executinganyoperation like insertion, deletion, array takes more time.	When it comes to executing any operation likeinsertion,deletion,the linked list takes less time.

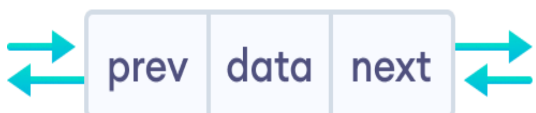
DOUBLY LINKED LIST:A doubly linked list is a more complex data structure than a singly linked list the main advantage of a doubly linked list is that it allows for efficient traversal of the listinbothdirections.Thisisbecauseeachnodeinthelistcontainsapointertothepreviousnode and a pointer to the next node.

.Thisallowsforquickandeasyinsertionanddeletionofnodesfromthelistaswellas efficient traversal of the list in both directions,

.Adoublylinkedlistisadatastructurethatconsistsofasetofnodeseach ofwhichcontainsa value and two pointers .onepointing to the previous nodein thelist and onepointing to the next node in the list.

Representationofdoublylinkedlistindatastructure:Inadatastructureadoublylinkedlistis represented using nodes that have three fields

1. Data
2. A pointerto thenext node(next)
3. A pointerto theprevious node(prev)

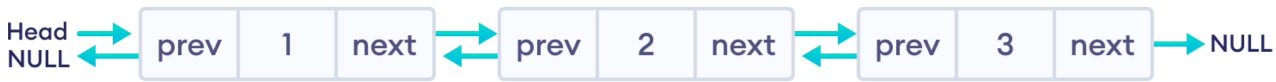


InC,thestructureofadoublylinkedlist canbegivenas, struct node
{
structnode
*prev;int data;
structnode*next;
};

Doublylinkedlistoperations:

1.insertion
2.deletion
3.searching
4.Traversing

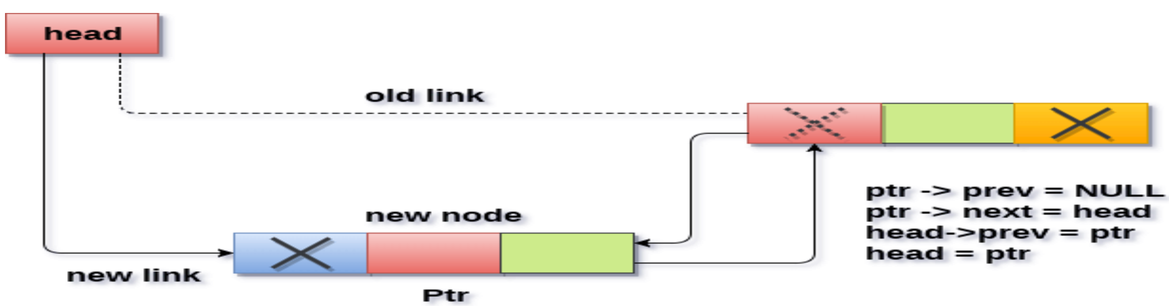
Example:



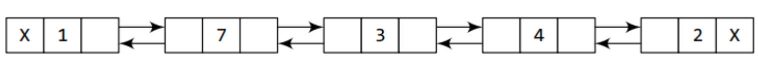
Insertion: To inserta newelementinthe listthisoperationcan beperformedinthreeways

1.insertionatbeginningofthelist
2.insertion at end of the list
3.insertion after specified node

1. **insertion at beginningof the list :The elements is inserted at beginning the newnode connectedtonextnode.Indoublylinkedlistfirstnodeandlastnodeitindicates NULL value.**

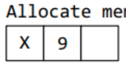


Insertion into doubly linked list at beginning

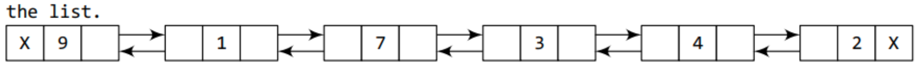


START

Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.



Add the new node before the START node. Now the new node becomes the first node of the list.



START

Algorithm:

Step1:IFptr=NULL

WriteOVERFLOW
GotoStep9
[ENDOFIF]

Step2:SETNEW_NODE=ptr

Step3:SET ptr=ptr->NEXT

Step4:SETNEW_NODE->DATA=VAL

Step5:SETNEW_NODE-

>PREV=NULL**Step6:**SETNEW_NODE-

>NEXT=START**Step7:**SEThead-

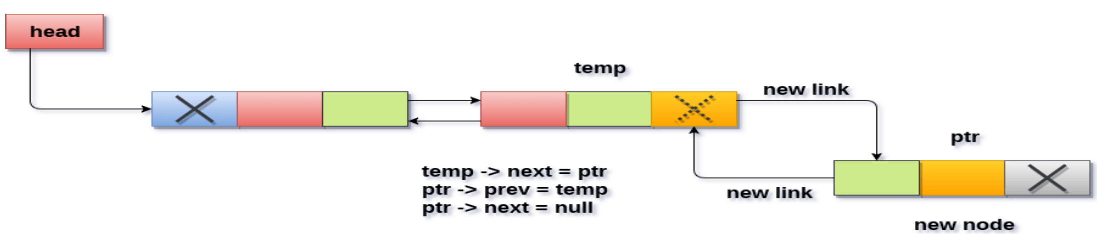
>PREV=NEW_NODE

Step8:SEThead=NEW_NODE

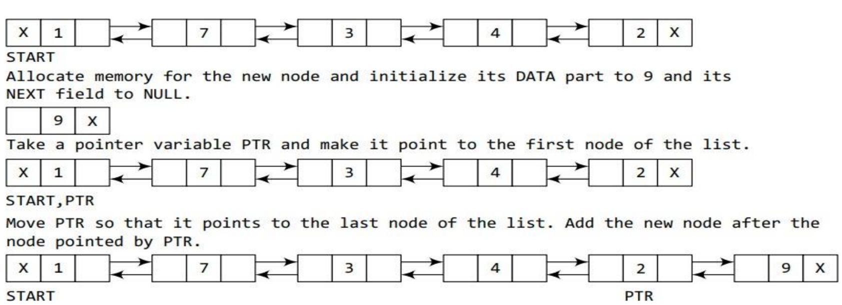
Step9:EXIT

2. insertion at end oftheList :The element is inserted at end thenew node inserted into the list.

.Allocate the memory for the new node make the pointer ptr point to the newnode being inserted .



Insertion into doubly linked list at the end



Algorithm :

Step1:IFPTR=NULL

WriteOVERFLOW
GotoStep11
[END OF IF]

Step2:SETNEW_NODE=PTR

Step3:SETPTR=PTR->NEXT

Step4:SET NEW_NODE-

>DATA=VAL**Step5:**SET NEW_NODE ->

NEXT = NULL**Step6:**SETTEMP=START

Step7: RepeatStep8whileTEMP->NEXT!=NULL

Step8:SETTEMP=TEMP->NEXT[END

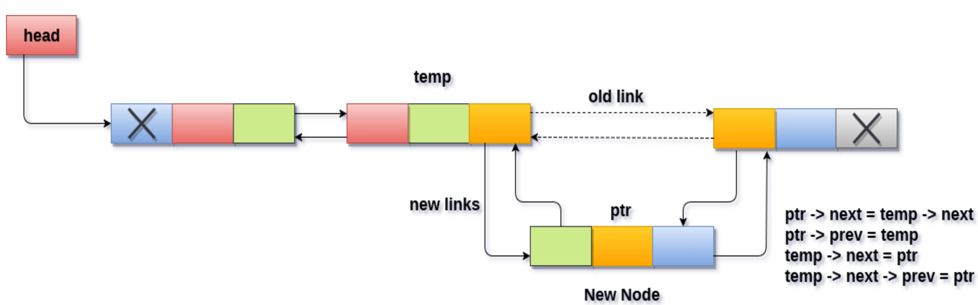
OF LOOP]

Step9:SETTEMP->NEXT=NEW_NODE

Step10C:SETNEW_NODE-

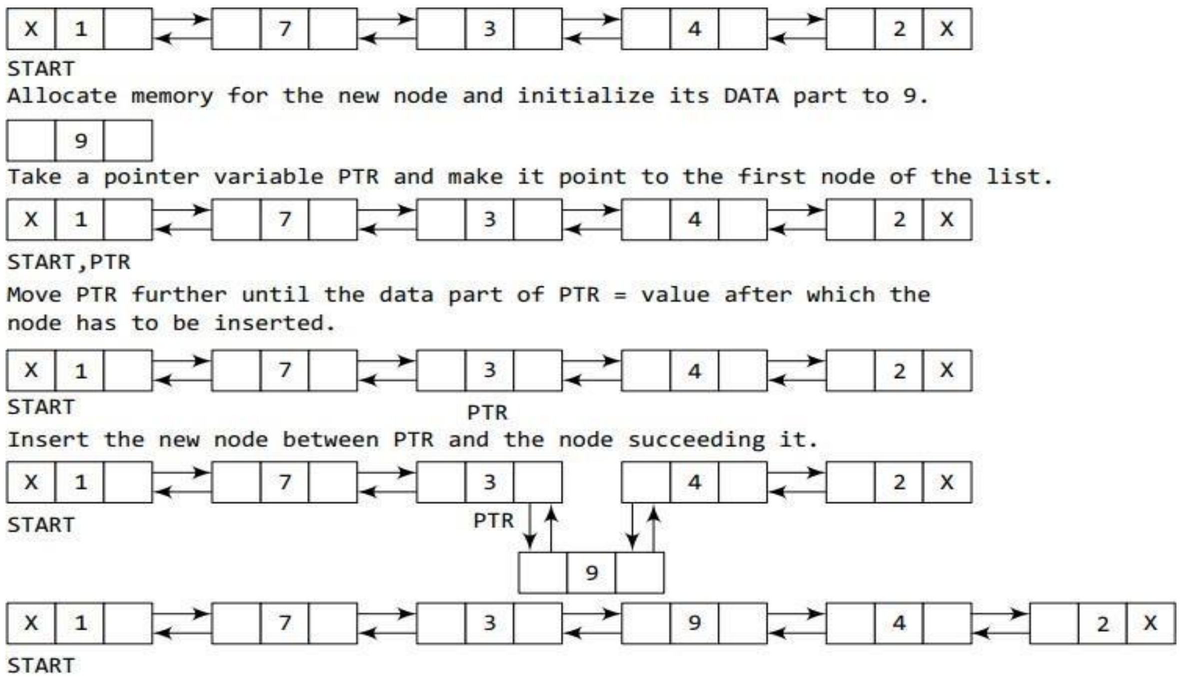
>PREV=TEMP**Step11:**EXIT

3. insertionafter specifiednode:To insertanode after thespecified positionin thelist.



Insertion into doubly linked list after specified node

Algorithm:



Step 1:IFPTR= NULL

WriteOVERFLOWG
otoStep15
[ENDOFIF]

Step2:SETNEW_NODE=PTR

Step 3:SETPTR=PTR->NEXT

Step4:SETNEW_NODE->DATA=VAL

Step5:SETTEMP=START

Step6:SETI=0

Step7:REPEAT8to10untilI

Step8:SETTEMP=TEMP ->NEXT

STEP9:IFTEMP=NULL

STEP10:WRITE"LESSTHANDESIREDNO.OFELEMENTS"

GOTOSTEP15[
ENDOFIF][END
OFLOOP]

Step11:SETNEW_NODE->NEXT=TEMP->NEXT

Step12:SETNEW_NODE ->PREV=TEMP

Step13:SETTEMP->NEXT=NEW_NODE

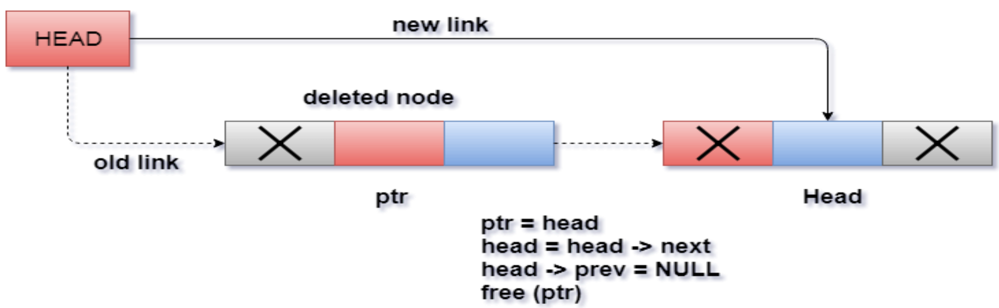
Step14:SETTEMP ->NEXT ->PREV=NEW_NODE

Step15:EXIT

2. Deletion:Deleteanodefromthelistthedeletionofanodeinadoublylinkedlistcanbedividein to three categories

1. Deletionat beginning of thelist
2. Deletionat end of thelist
3. deletionat aspecified node

1. Deletion at beginning of the list : Deletion in doubly linked list at the beginning is the simplest operationtodeleteanodeatfirstnodeandheadpointingto nextnodeofthelistnowfreethepointer ptr by using free function.



Deletion in doubly linked list from beginning

Algorithm:

STEP1:IFHEAD=NULL

WRITEUNDERFLOW
GOTO STEP 6

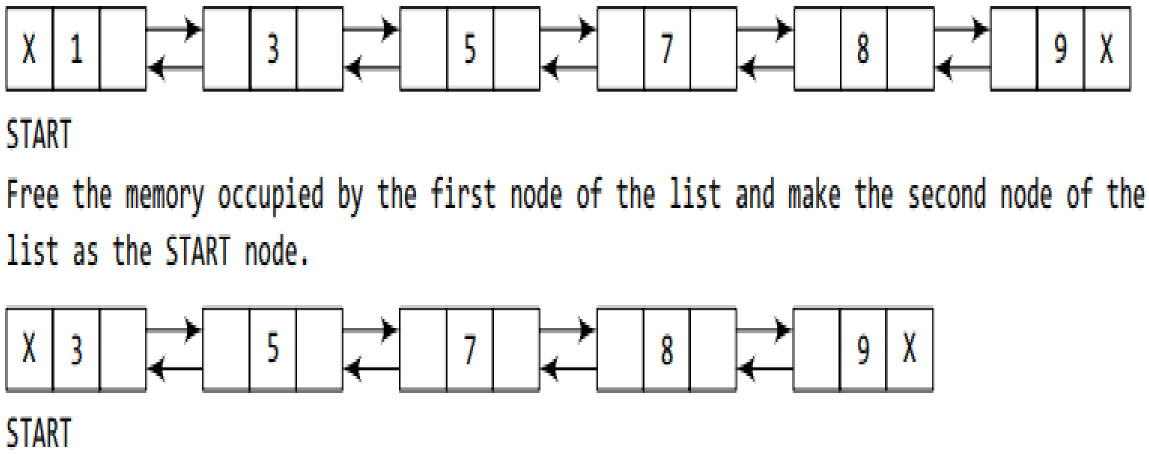
STEP2:SETPTR=HEAD

STEP3:SETHEAD=HEAD→NEXT

STEP 4:SET HEAD→PREV =NULL

STEP 5: FREE PTR

STEP6:EXIT

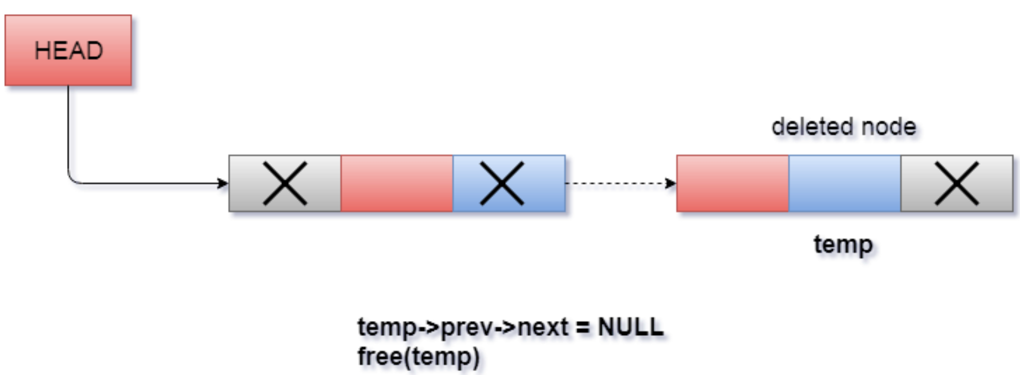


2. deletionatendofthelist:Deletionofthelastnodeinadoublylinkedlist needstraversingthelistin order to reach the last node of the list and then make pointer.

1. ifthelistisalreadyemptythentheconditionhead==NULLwillbecometrueandthereforethe operation cannot be carried on.

2. ifthereis onlyone nodein thelist thenthe conditionhead->next==NULLbecome true.

3. otherwisejust traverse the listtoreach thelastofthe list.



Deletion in doubly linked list at the end

Algorithm:

Step1: IFHEAD= NULL

WriteUNDERFLOW
GotoStep7 7
[ENDOF IF]

Step2:SET TEMP=HEAD

Step3:REPEATSTEP4WHILETEMP->NEXT!=NULL

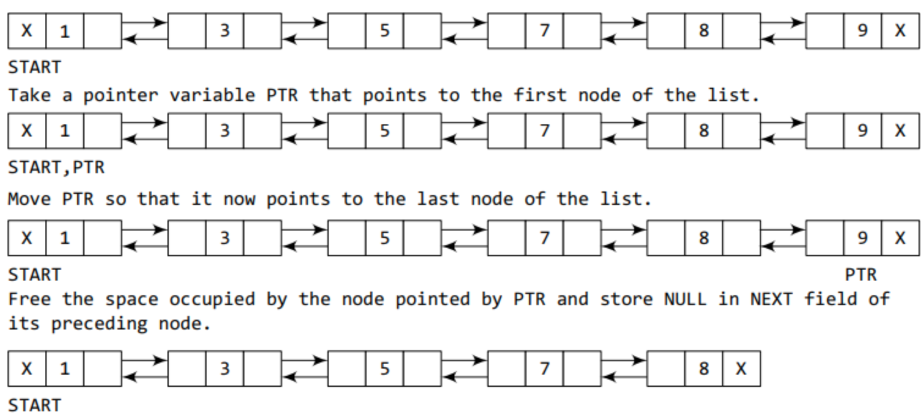
Step4:SETTEMP=TEMP->NEXT
[END OF LOOP]

Step5:SETTEMP ->PREV->NEXT=NULL

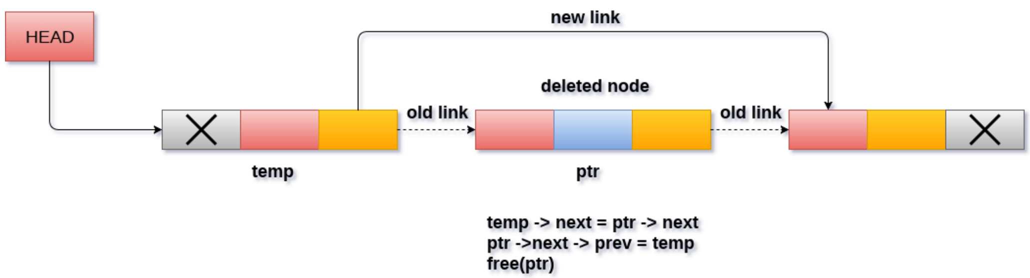
Step6:FREETEMP

Step7:EXIT

DeletingtheLast Nodefroma DoublyLinked List:



3. Deletion at a specified node : in order to delete the node after the specified data we need to perform the following steps
   .copytheheadpointerintoatemporarypointertemp.
   .Traversethelistuntilwefindthedesireddatavalue.
   .checkifthisis thelastnodeofthelist.
   .checkifthenodewhichistobedeletedisthelastnodeofthelistifitsothenwe have to make the next pointer of this node point to NULL so that it can be the new last node of the list.



Deletion of a specified node in doubly linked list

Algorithm:

Step1: IFHEAD= NULL

WriteDERFLOW
GotoStep9
[ENDOF IF]

Step2:SET TEMP=HEAD

3:Repeat Step 4whileTEMP->DATA!=ITEM

Step4:SETTEMP=TEMP->NEXT
[END OF LOOP]

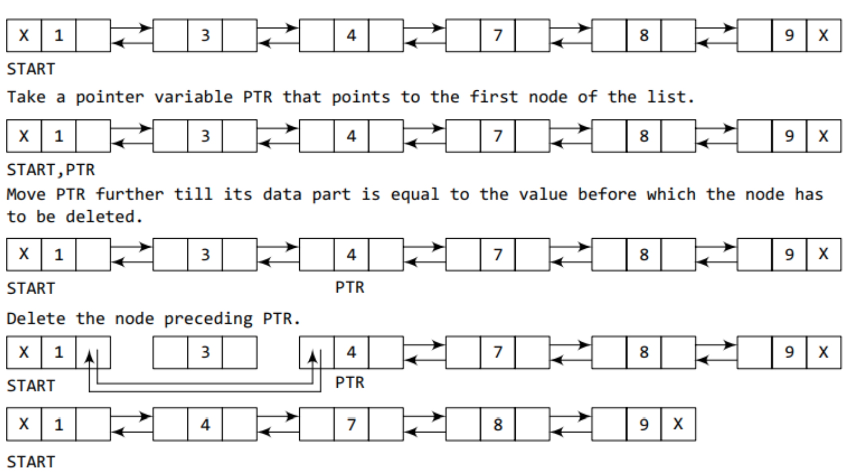
Step5:SETPTR =TEMP->NEXT

Step6:SETTEMP->NEXT=PTR->NEXT

Step7:SETPTR ->NEXT->PREV=TEMP

Step 8: FREE PTR

Step9:EXIT



3. searching:we justneedtraversethelistinordertosearchfor aspecificelementinthe list perform following operations in order to search a specific operation

1. copyheadpointerintotemporarypointervariableptr.

2. declarea local variableI and assign it to 0.

3. Traverse the list until the pointer ptr becomes NULL keep shifting pointer to its next and increasing i by plus one.

4. compareeachelement ofthe listwith theitemwhichis tobesearched.

5. if the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm :

Step1:IFHEAD==NULL

WRITE"UNDERFLOW"GOTO
STEP8
[ENDOFIF]

Step 2: SetPTR= HEAD

Step 3: Seti =0

Step4: Repeatstep5to7whilePTR!=NULL

Step5:IFPTR→ data=item

return i
[ENDOFIF]

Step6:i=i +1

Step7:PTR=PTR→next

Step8: Exit

4.Traversing :Although traversing means visiting each node of the list once to perform some specificoperation .Here we are printing the data associated with each node of the list.

Algorithm :

Step1:IFHEAD==NULL

WRITE"UNDERFLOW"
GOTOSTEP6
[ENDOF IF]

Step2: SetPTR = HEAD

Step3: Repeatstep 4and 5whilePTR !=NULL

Step4: WritePTR→ data

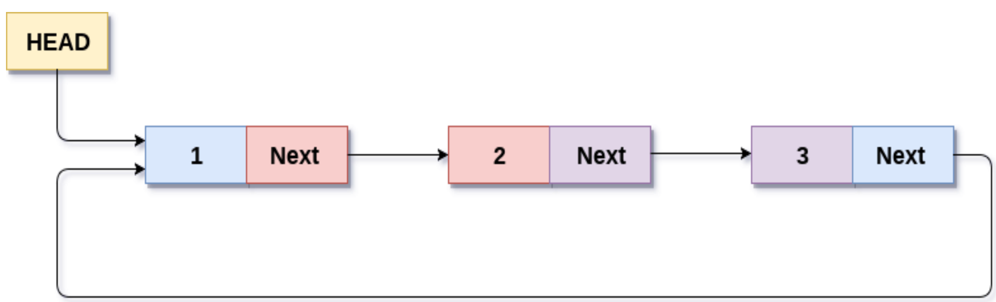
Step5:PTR =PTR → next

Step6: Exit

. Circular linkedlist : in a circular singly linked list the last node of the list contains a pointer to the first node of the list. we can have circular singly linked list as well as circular doubly linked list.

. we traverse a circular singly linked list until we reach the same node where we started .The circular singly linked list has no beginning and no ending there is no null value present in the next part of any of the nodes.

.circularlinkedlistaremostly usedin taskmaintenancein operating system.



Circular Singly Linked List

Operations:circularlinkedlistoperationsare

1.insertition

2.deletion

3.Traversing

4.Searching

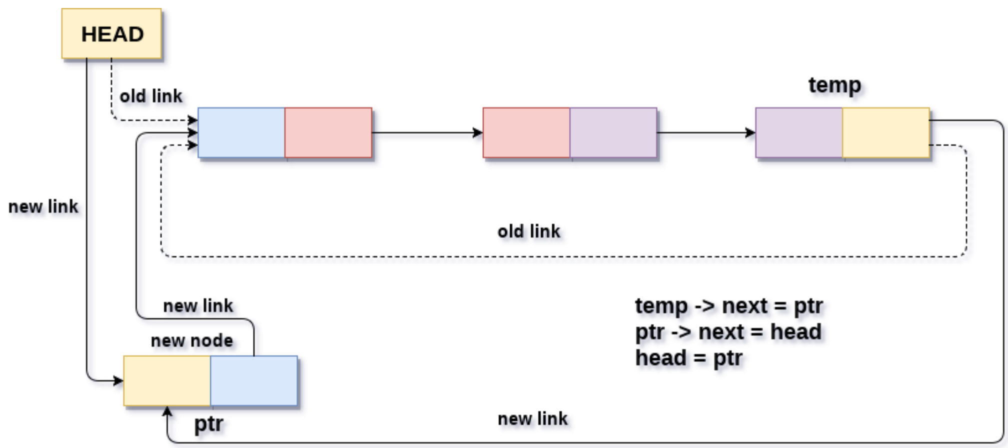
1. **insertion:Toinsertanewnodein thelist.insertioncanbeclassified as**

.insertionatbeginning:Therearetwoscenarioinwhichanodecanbeinsertedin circular singly linked list at beginning .

.Eitherthenodewillbeinsertedinanemptylistorthenodeistobeinsertedinanalready filled list.

. The condition head==NULL will be truesince the list in which we are inserting the node is a circular singly linked list there fore the only node of the list.

. The condition head==NULL will become false which means that the list contains at least one node.



Insertion into circular singly linked list at beginning

Algorithm :

Step 1:IFPTR= NULL

WriteOVERFLOWG
otoStep11
[ENDOFIF]

Step2:SETNEW_NODE=PTR

Step 3:SETPTR=PTR->NEXT

Step4:SETNEW_NODE->DATA=VAL

Step5:SETTEMP=HEAD

Step6: RepeatStep8whileTEMP ->NEXT!=HEAD

Step7:SETTEMP=TEMP ->NEXT

[ENDOFLOOP]

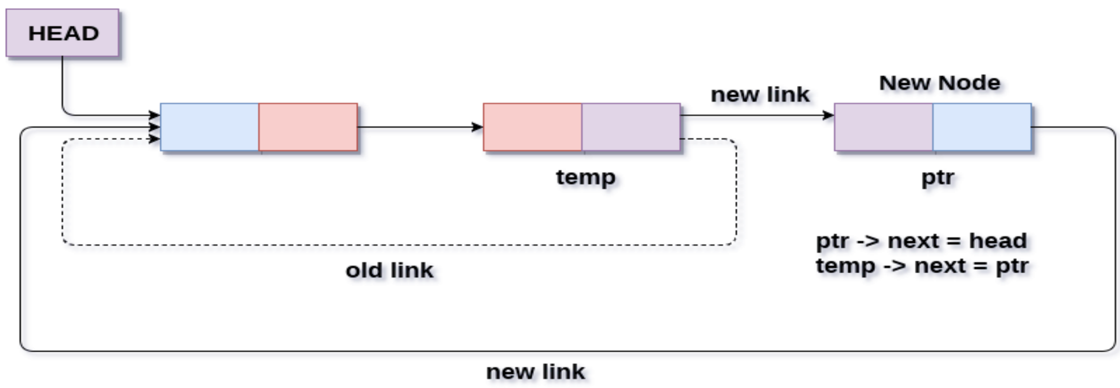
Step8:SETNEW_NODE->NEXT=HEAD

Step9:SETTEMP→NEXT=NEW_NODE

Step 10: SET HEAD = NEW_NODE

Step11:EXIT

2. **insertingnodeatendofthelist:To inserta newnodein end ofthelist**



Insertion into circular singly linked list at end

Algorithm :

Step 1:IFPTR= NULL

WriteOVERFLOWG
otoStep1

[ENDOFIF]

Step2:SETNEW_NODE=PTR

Step 3:SETPTR=PTR->NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step5:SETNEW_NODE->NEXT=HEAD **Step**

6: SET TEMP = HEAD

Step7: RepeatStep8whileTEMP ->NEXT!=HEAD

Step8:SETTEMP=TEMP ->NEXT
[ENDOFLOOP]

Step9:SETTEMP ->NEXT=NEW_NODE

Step10:EXIT

2. Deletion:Todeletethe elementintothelistitcanbeclassified as

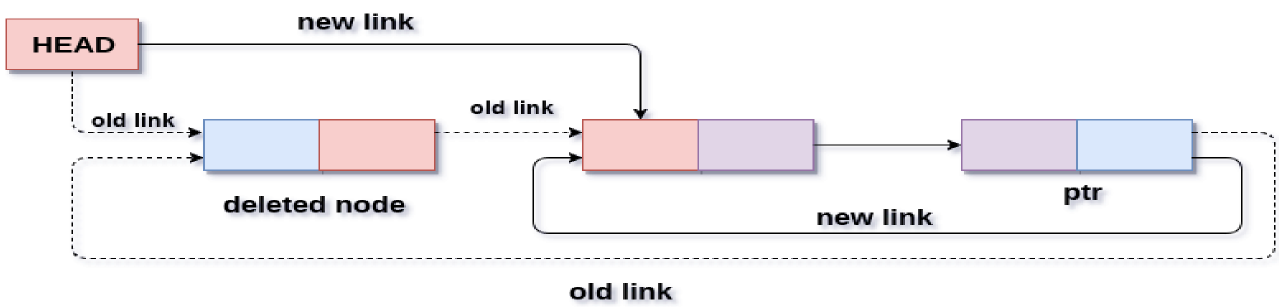
.Deletionatbeginning:Removethenodefromcircularsinglylinkedlistatthebeginningofthe node in the list .

.There arethreescenarios ofdeleting anodefrom circularsinglylinked list at beginning

1. Thelist is empty

2. Thelist contains singlenode.

3. Thelist contains morethan onenode.



Deletion in circular singly linked list at beginning

Algorithm:

Step1:IFHEAD=NULL
WriteUNDERFLOW
GotoStep8
[ENDOFIF]

Step2:SETPTR=HEAD

Step3: RepeatStep4whilePTR→NEXT!=HEAD

Step4:SETPTR=PTR→next
[ENDOFLOOP]

Step5:SETPTR→NEXT=HEAD→ NEXT

Step6:FREEHEAD

Step7:SET HEAD=PTR→NEXT

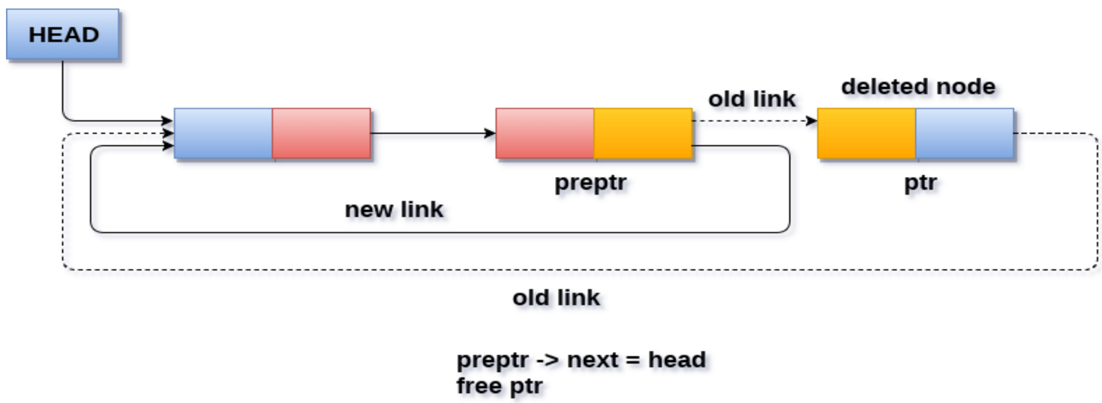
Step8:EXIT

.Deletion at end of the List :There are three scenarios of deleting a nodein circular singly linked list at the end

. Thelist is empty

.Thelistcontainssingleelement.

.Thelistcontainsmorethanoneelement.



Deletion in circular singly linked list at end

Algorithm :

Step1:IFHEAD=NULL
WriteUNDERFLOW
GotoStep8
[ENDOFIF]

Step2:SETPTR =HEAD

Step3: RepeatSteps 4and 5while PTR->NEXT!=HEAD

Step4:SETPREPTR =PTR

Step5:SETPTR=PTR->NEXT [END
OF LOOP]

Step6:SET PREPTR ->NEXT=HEAD

Step7:FREEPTR

Step8:EXIT

3. **Traversing:Althoughtraversingmeansvisitingeachnodeofthelistoncetoperformsome specific operation Here we are printing the data associated with each node of the list.**

Algorithm:

1. **Setptr=Head**

2. **Ifptr=NULLwriteemptylistgoto step7**

3. **Repeatstep4and5until Ptr->next!=head**

4. **printptr->data=value**

5.ptr=ptr->next

6.printptr->data

7.Exit

4. **Searching : searching in circular singly linked list needs traversing across the list the item which is to be searched in the list is matched with each node data of the list once and if the match found then the location of that item is returned other wise -1 returned.**

Algorithm :

1. **setPtr=head**
2. **setI=0**
3. **ifptr=NULLwrite“emptylist”gotostep8**
4. **ifhead->data=item**

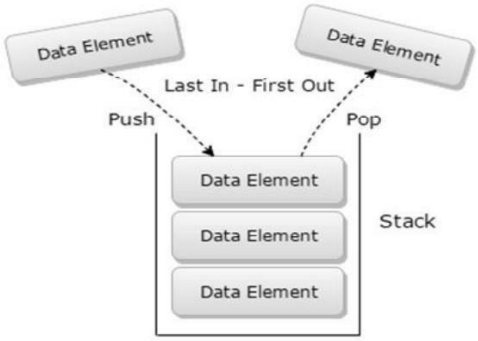
   Write i+1

5. **Repeatstep5to7untilptr->next!=head**
6. **if ptr->data=item**
7. **i=i+1**
8. **ptr=ptr->next**
9. **Exit**

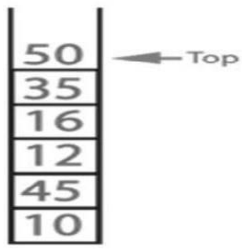
STACKS:

Stack is a linear data structure in which the insertion and deletion operationsare performed at only one end. In a stack adding and removingof elements are performed at single position which is known as"top".Thatmeans,newelementisaddedattopofthestack andan elementisremovedfrom thetopof the stack only. In stack, the insertion and deletion operations are performed based on LIFO (Last In First Out) principle.The firstelementwhichisinsertedintostack isdeleted last thelast element whichis insertedintostackisdeletedfirst.

In a stack, the insertion operation is performed using a function called "push" and deletion operation is performed using a function called "pop". In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end i.e., at Top.



Example: If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image below.



OPERATIONSONASTACK:

Thefollowingaresomecommonoperationsimplementedonthe stack

.push():whenweinsert anelementinastackthentheoperationisknownasapush.ifthestackis full then the overflow condition occurs.

.pop():whenwedeleteanelementfromthestacktheoperationisknownasapop.Ifthestackis empty means that no element exists in the stack this state is known as an underflow state.

.isEmpty(): it determineswhetherthe stackis empty or not.

.isfull():itdetermines whetherthe stackis fullor not.

.peek():itreturns theelementat thegiven position.

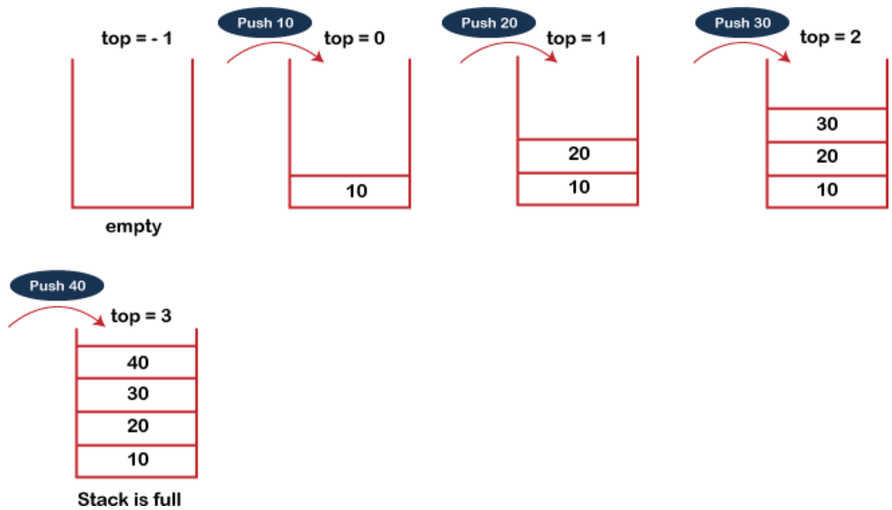
.count():it returnsthe totalnumber ofelements availablein astack.

.display():itprintsalltheelementsavailableinthestack.

.Pushoperation:Beforeinsertinganelementinastackwecheck whetherthestackisfull.ifwe try to insert the element in a stack and the stack is full then the overflow condition occurs.

.wheneinitializeastackwesetthevalueoftopas -1tocheckthatthestackisempty.when the new element is pushed in a stack first the value of the top gets incremented i.e top=top+1 And the element will be placed at the new position of the top.

.Theelementswill beinserted untilwereachthemaxsizeof the stack.

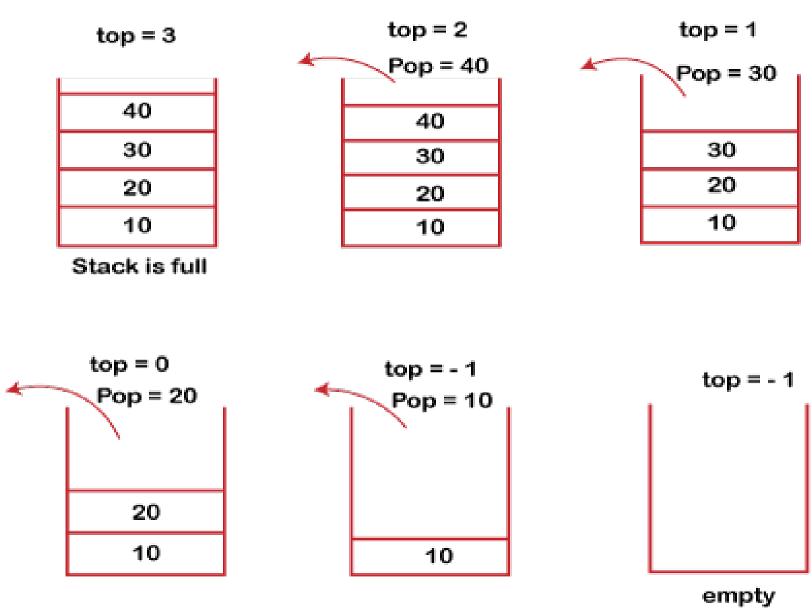


.POPoperation:**Beforedeletingtheelementfromthestack,wecheckwhetherthestackis empty.**

.Ifwetryto deletetheelement from theempty stack, thenthe***underflow***condition occurs.

.Ifthe stack is notempty, wefirst access theelementwhich is pointed by the***top***

.Oncethe popoperationisperformed, thetopis decrementedby 1,i.e.,**top=top-1**.



Stack datastructure canbe implement intwo ways.

They are as follows. 1. Stack Using Arrays

2.stackUsingLinkedList

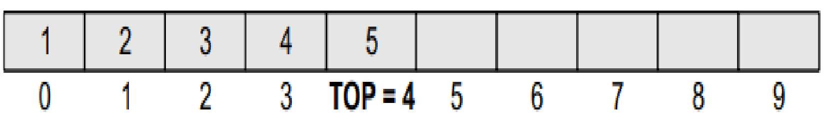
When stackisimplemented usingarray,thatstack can organize only limited numberof elements.

When stack is implemented using linked list, that stack canorganize unlimited number of elements.

ARRAYREPRESENTATIONOF STACKS:

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost elementof the stack. It is this position where the element will be added to or deleted from. Thereisanother variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full. (You must be wondering why we have written MAX–1. It is because array indices start from 0.).

TheabovestackshowsthatTOP=4,soinsertionsanddeletionswillbedoneatthis



position.Intheabovestack,fivemoreelements canstillbestored.

Stackimplementationusingarray:

Beforeimplementingactualoperations,firstfollowthebelowstepstocreateanempty stack.

Step1: Declareallthe functionsusedinstack(push,pop,display)implementation.

Step2: Createaonedimensionalarraywith fixedsize.

Step3: Definea integervariable'top'andinitializewith'-1'.(inttop=-1).

Step 4: In main method displaya menu with list ofoperations and make suitable functioncallsto perform operation selected by the user on the stack.

PushOperation:

The push operation is used to insert an element into the stack. The new element is addedatthetopmostpositionofthestack.However,beforeinsertingthevalue,wemustfirstcheckif

TOP=MAX–1,becauseifthatisthecase,thenthestackisfullandnomoreinsertionscanbe done.IfanattemptismadetoinsertavalueinastackthatisalreadyfulI,anOVERFLOWmessageisprinted.

1	2	3	4	5					
0	1	2	3	**TOP = 4**	5	6	7	8	9

Stack

Toinsertanelement withvalue6, wefirstcheckifTOP=MAX–1. Iftheconditionisfalse, then we increment the value of TOP and store the new element at the position given by stack[TOP]. Thus, the updated stack becomes as shown

1	2	3	4	5	6				
0	1	2	3	4	**TOP = 5**	6	7	8	9

Stack after insertion

AlgoritmtoInsertanElementinaStack:

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

The algorithm to insert an element in a stack. In Step 1, we first check for the OVERFLOW condition. In Step 2, TOP is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed byTOP.

PopOperation:

The pop operation is used to delete the topmost element from the stack. However, before deletingthevalue,wemust firstcheckifTOP=NULLbecauseifthat isthecase,thenit meansthe stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

1	2	3	4	5					
0	1	2	3	**TOP = 4**	5	6	7	8	9

Stack

Todeletethetopmostelement,wefirstcheckifTOP=NULL. Iftheconditionisfalse,then wedecrement thevaluepointedbyTOP.Thus,theupdatedstackbecomes as

1	2	3	4						
0	1	2	**TOP = 3**	4	5	6	7	8	9

Stack after deletion

AlgoritmtoDeleteanElementfromStack:

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

The algorithm to delete an element from a stack. In Step 1, we first check for the UNDERFLOW condition. In Step 2, the value of thelocation in the stack pointed by TOP is stored in VAL. In Step 3, TOP is decremented.

PeekOperation:

Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.

Here,thePeekoperationwillreturn5,asitisthevalue ofthetopmostelementofthe stack.

1	2	3	4	5					
0	1	2	3	**TOP = 4**	5	6	7	8	9

Stack

AlgorithmforPeekoperation:

```
Step 1: IF TOP = NULL
             PRINT "STACK IS EMPTY"
             Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Displayoperation:

Displays the elements of a Stack. We can use the following steps to display the elements of a stack.

Step 1: Check whetherstackisEMPTY. (top==-1)

Step2: IfitisEMPTY,thendisplay"StackisEMPTY!!!"andterminatethefunction.

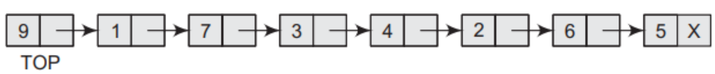
Step3: IfitisNOTEMPTY,thendefineavariable'i'andinitializewithtop.Displaystack[i] value and decrement i value by one (i--).
Step 4: Repeatabovestepuntilivaluebecomes '0'.

StackusingLinkedList:

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself(size of the stack). Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. Thestack implementedusing linked list can work forunlimited numberofvalues.Thatmeans,stackimplementedusinglinkedlistworksforvariablesizeofdata. So,thereisnoneedtofixthesizeatthebeginningoftheimplementation.TheStackimplemented using linked list can organize as many data values as we want.

In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

Example:



Linked stack

Inaboveexample,thelastinsertednodeis9andthefirstinsertednodeis5.Theorderofelements inserted is 5,6,2,4,3,7,1 and 9.

StackimplementationusingLinkedlist:

To implement stack using linked list, we need to set the following things beforeimplementing actual operations.

Step1:Declareallthefunctionsusedinstack(push,pop,disply)implementation Step 2: Define a 'Node' structure with two fields data and link.

Step3: DefineaNodepointer 'top'andsetittoNULL.

Step4: Implement the main method bydisplaying Menu withlist ofoperationsand makesuitable function calls in the main method.

OPERATIONSON ALINKED STACK:

Alinked stacksupportsallthe threestackoperations,thatis,push, pop,and peek.

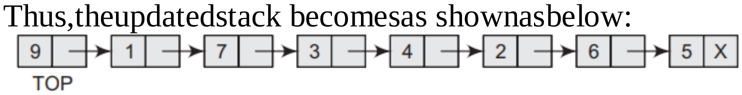
Push Operation:

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.

Linked stack

To insert an element with value 9, we first check if TOP=NULL. If this is the case, thenwe allocate memory for a new node, store the value in its DATA part and NULL inits NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP.

Thus,theupdatedstack becomesas shownasbelow:



Linked stack after inserting a new node

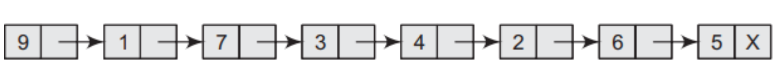
Algorithmtopush anelementintoalinked stack:

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

In Step 1, memory is allocated for the new node. In Step 2,the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is thefirst nodeofthelinked list. This isdoneby checking ifTOP =NULL.In casethe IFstatement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then itis added before the first node of the list (that is, the TOP node) and termed as TOP.

Pop Operation:

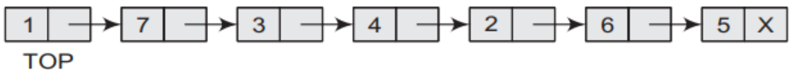
The pop operation is used to delete the topmost element from a stack. However, before deletingthevalue,wemustfirstcheckifTOP=NULL,becauseifthisisthecase,thenitmeans



Linked stack

that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown below



Linked stack after deletion of the topmost element

Algorithmtodeleteanelementfromastack:

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

InStep1,wefirstcheck fortheUNDERFLOWcondition.InStep2,weuseapointerPTR that points to TOP. In Step 3, TOP is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

Display operation:

Displaying stack of elements We can use the following steps to display the elements (nodes) of a stack.

Step1:Checkwhether stackisEmpty(top==NULL).

Step 2: If it is Empty, then display 'Stack is Empty!!!' and terminate the function.

Step3:Ifit isNotEmpty,thendefineaNodepointer'temp'andinitializewithtop.

Step4:Display'temp->data->andmoveittothenextnode.Repeatthesameuntiltempreachesto the first node in the stack (temp-> link != NULL).

Step5:Finally!Display'temp->data->NULL'.

APPLICATIONSOF STACKS:

Stacks can be easily applied for a simple and efficient solution. Different Application of Stacks:

- Reversingalist

- Arithmetic Expression

  a) infixnotation

  b) prefixnotation

  c) postfixnotation

  d) Evaluationpostfixexpression

- Conversionofaninfix expressionintoapostfix expression
- Conversionofan infixexpression into aprefixexpression
- Evaluationofaprefix expression
- Recursion

  TowerofHanoi

.Balancingsymbols

1. ReversingaList:Toreverseastringor elementsinreverseorder.

.Itis a efficient

.writeaprogramtoimplementstackoperationsusingareversingalistofelements.

```
#include<stdio.h>
#include<conio.h>
#definemaxsize10
‘    intstack[size],top=-1;
 Void push(int);
   intpop();
    intisempty();
    void main()
     {
    inta[size],i;
    clrscr();
     printf("enterarrayelements\n");
       for(i=0;i<size;i++)
          scanf("%d",&a[i]);
```

```
            for(i=0;i<size;i++)
              push(a[i]);
               printf("Listisreverseorder\n");
               for(i=0;!isempty();i++)
              {
             int ele;ele=pop();
             printf("%d",ele);
            }
          getch();
}
Voidpush(intele)
   {
if(top==size-1)
  {
   Printf("stackisoverflow");
else
top=top+1;
stack[top]=ele;
}
}
intpop()
{
int ele;
if(top==-1)
printf("stackisunderflow");
else
{
ele=stack[top];
 top=top-1;
}
returnele;
```

```
}
intisempty()
{
if(top==-1)
return 1;
else
return 0;
}
```

2.Expressions:itisasetofoperandsinbetweentheoperatoriscalledasexpression

Example : A+B

a) **Infix:whentheoperatoriswritteninbetweentheoperandsthenitisknownasinfix notation.**

Example:a+b, a/b,a*b

b) **Prefix:Theoperatorcomesfirstfollowedbytheoperands.**

Example:++a,+ab,--ab

c) **Postfix:Theoperands comesfirst followedby theoperator.**

Example: ab+,ab*,ab/

d.Evaluationofpostfix Expression:itisaoperand stack

.only onestackisused .

.if thecharacterisan operandthenpushinto stack.

.ifthecharacterisanoperatorthenpoptop,twooperandsfromthestackandpushthe result back into stack.

.After reading all the characters from the postfix expression stack will be having only thewhich is result.

Example: 562*+

character	stack
5	5
5	56
2	5 6 2
*	Pop2,pop6 6*2=12 5 12

+	**Pop12,pop5** **5+12=17**

562*+=17

3. Conversionofinfixtopostfixexpression:

1) **Ifthecharacterisleftparenthesispushtothestack.**
2) **Ifthecharacterisoperand,addtothepostfix expression**
3) **Ifthecharacterisoperator,checkwhetherstackis empty**
   1) **Ifthestackisempty,pushoperatorintostack**
   2) **Ifthestackisnot empty ,checkthepriorityofthe operator**
      i) **ifthepriorityoperator>operatorpresentattopofstackthenpushoperatorinto stack.**
      ii) **ifthepriorityoftheoperatoris<=operator presentattopofthestack,thenpop theoperatorfrom stackandADDtopostfixexpressionandgoto step(i).**

4) **ifthecharacterisrightparenthesisthenpopalltheoperationsfromthestackuntilitmatches left parenthesis and ADD to postfix expression.**

5) **Afterreadingallthecharacters,ifstackisnotemptythenpopandADDtopostfixexpression.**

Example : Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

InputExpression	Stack	PostfixExpression
K		K
+	+	
L	+	KL
-	-	KL+
M	-	KL+M
*	-*	KL+M

N	-*	KL+MN
+	+	K L+M N* KL+MN*-
(	+(	KL+MN*-
O	+(	KL+MN*-O
^	+(^	KL+MN*-O
P	+(^	KL+MN*-OP
)	+	KL+MN*-OP^
*	+*	KL+MN*-OP^
W	+*	KL+MN*-OP^W
/	+/	KL+MN*-OP^W*
U	+/	KL+MN*-OP^W*U
/	+/	KL+MN*-OP^W*U/
V	+/	KL+MN*-OP^W*U/V
*	+*	KL+MN*-OP^W*U/V/
T	+*	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*- OP^W*U/V/T*+Q+

Thefinalpostfixexpressionofinfixexpression(K+L-M*N+(O^P)*W/U/V*T+Q)isKL+MN*-

OP^W*U/V/T*+Q+.

4. Conversionofinfixtoprefixexpression:

.reversethe expression

.Apply thepostfix notation

.leftparenthesis (to)

.rightparenthesis)to(

.reversethepostfixexpression

Example : (A+B)*C-D+F

ReversetheexpressionF+D-C*(B+A)

character	stack	postfix
F		F
+	+	F
D	+	FD
-	+-(POP +)	FD+
C	-	FD+
*	-*	FD+C
(	- * (	FD+C
B	- * (	FD+CB
+	- *(+ POP(*)	FD+CB FD+CB*
A	-(+	FD+CB*A
)	-(+) POP(+) POP(-)	FD+CB*A+-

ReversetheExpression=-+A*BC+FD

5. EvaluationofaPrefixExpression:startrightto left

-+3*45/16^23=21

symbol	stack
3	3
2	32
^	8
16	816
/	2
5	25
4	254
*	220
3	2203
+	223
-	21

6. **Recursion:Recursionisafunctionthefunctioncallitselfiscalledasrecursion**

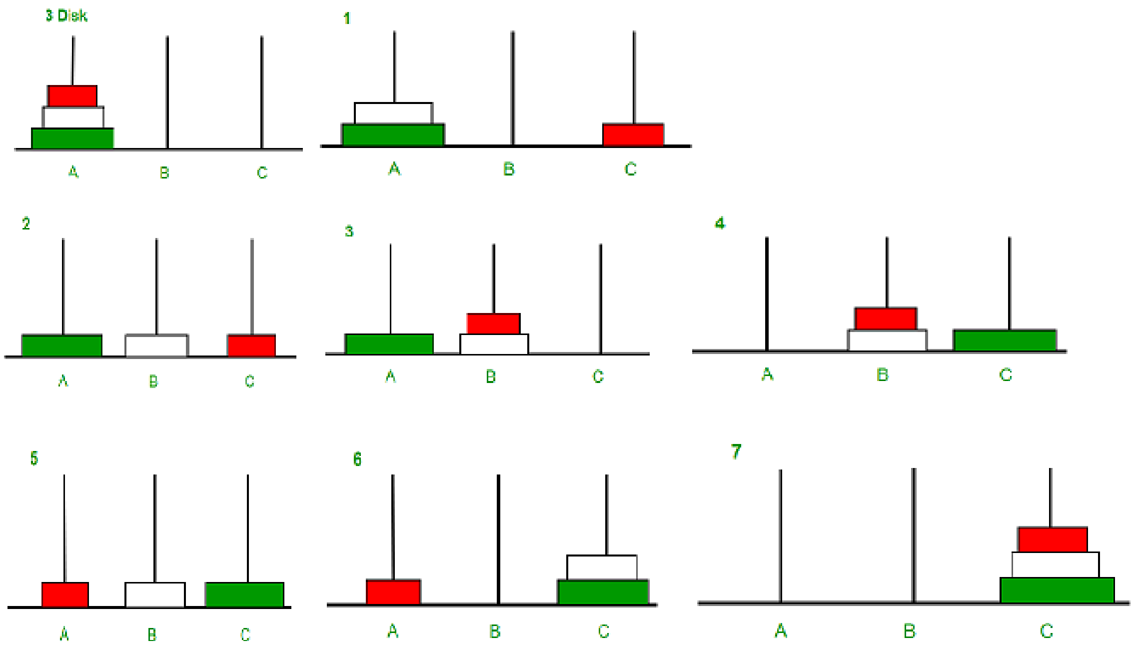
Recursion example Towers of Hanoi problem

.TowersofHanoiproblem:Itisapuzzle game

. HereThreepegsordisksthedataismovedtoonedisktoanotherdisk Disks are A,B,C DISK=3 N=3

1. **Tower(N-1,Begin,End,Aux)**
2. **Tower(1,Begin,Aux,End)**
3. **Tower(N-1,Aux,Begin,End)**

S.NO	disk
1	A->C
2	A->B
3	C->B
4	A->C
5	B->A
6	B->C
7	A->C



AlgorithmforTowerofHanoi:

Step1: Start

Step2:Letthethreetowersbethesource,dest,aux.Step 3:

Read the number of disks, n from the user. Step 4: Move

n-1 disks from source to aux.

Step5:Moventhdisk fromsourcetodest.Step 6:

Move n-1 disks from aux to dest.

Step7:Repeat Steps3to5,bydecrementingnby1.

Step8:Stop

7. BalancingSymbols:symbolstackisused

(	**Expressions**
[	**Expressions**
{}	**Blockofstatements**
()	**Balancedsymbols**
[ ]	**Balancedsymbols**

.countingopensymbolsandcountingclosedsymbols.

Algorithm : read character from Expression

.ifcharacterisopensymbol‘(‘,’[‘,’{‘pushsymbolintothestack.

.ifcharacterisclosedsymbol‘)’,’]’,’}’

a) checkifstackisemptyifthusexpressionisunbalanced.

b) ifthestackisnotemptythenpopthesymbolfromthestackandcomparewiththe symbol whichisread,ifdoesn’tmatchesexpressionisunbalanced.elserepeat the process.

. After Reading all character of expression still stack is not empty that implies unbalanced expression.

Ex: [(a+b)(a-b)]it is a balanced

expression . [(a+b)(a-

b]itisaunbalancedexpression

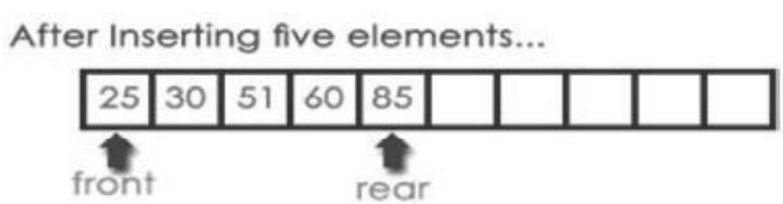
QUEUES: Queue isa linear datastructurewhich elementsareinserted at one endcalledrearand whichelementsaredeleted otherendcalledfront.frontisfrontsideandrearisbackside.Queue is a FIFOTechniquewhich element is insertedfirst thatelement isdeletedfirst.

Inaqueuedatastructure,theinsertionoperationisperformedusingafunctioncalled "enQueue()" and deletion operation is performed using a function called "deQueue()".

.Queueusestwopointersfrontandrear

The front pointer accesses the datafrom the front end.while the rear pointer accesses datafrom the rear end.

Example:Queueafterinserting 25,30,51,60and 85.



OperationsonaQueue:

Thefollowing operationsareperformed onaqueuedatastructure.

a. enQueue(value)-(Toinsertanelementintothequeue)

b. deQueue() -(To deleteanelementfromthequeue)

c. display()-(To display theelements of the queue)

1. Enqueue():Enqueue is a function used to insert a new element into the queue.in a queue the new element is always inserted at rear position .The Enqueue() function takesoneintegervalueasparameterandinsertsthatvalue into the queue.

Ex: consider the list of elements 12,9,7,18,14,36,45

Initiallyfront=0andrear=-1thenrear=rear+1=-1+1=0

Front=0 and rear=0 12is insertedo th positionFront=0

and rear=19 is inserted1 st position

Front=0rear=2 7isinserted2ndposition

Front=0 rear=3 18 is inserted 3rd position

Front=0 rear=4 14isinserted4thposition

Front=0 rear=536 is inserted 5 th position

Front=0rear=645 is inserted 6 th position

The elements are1297 18143645 Algorithm

:

Step1:ifRear=Max-1writeoverflowgotostep4

Step2: if front=-1 and rear=-1

Setfront=rear=0

Else

SetRear=Rear+1

Step3:SetQueue[Rear]=num Step

4:Exit

2. Dequeueoperation:Deletingavaluefromthequeue.in a queue data structure dequeue() is a function used to delete an element from the queue.in a queue the element is always deleted from front position .The dequeue() function doesnot take any values as parameter.

Initiallyfront=-1andrear=6thenfront=front+1and front=0

Front=0 andrear=612isdeleted

Front=1andrear=69isdeletedfromthequeue.

Front=2 rear=67 is deleted from the queue

Front=3 rear=618 is deleted from the queue

Front=4 rear=6 14 is deleted from the queue

Front=5rear=636 is deleted from the queue

Front=6rear=645isdeletedfromthequeuethenqueueis empty .

Algorithm:

Step1:iffront=-1orfront>Rear

Write under flow

Else

Setval=queue[front]

Set front=front+1

Step2:Exit

Display():

DisplaystheelementsofaQueue:Wecanusethefollowingstepstodisplaytheelementsofa queue.

Step1:CheckwhetherqueueisEMPTY.(front==rear)

Step2:IfitisEMPTY,thendisplay"Queue isEMPTY!!!"andterminatethefunction.

Step3:IfitisNOTEMPTY,thendefineanintegervariable'i'andset'i=front+1'.

Step4:Display'queue[i]'valueandincrement'i'valuebyone(i++).Repeatthesameuntil'i'value is equal to rear (i <= rear)

Queuedatastructurecanbeimplementedintwoways.

Theyareasfollows...

1. Using Array
2. UsingLinkedList

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number ofelements.

QueueimplementationbyUsingArray:

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue datastructureusing array isverysimple,justdefineaonedimensional arrayof specific sizeandinsert or delete the values into that array by using FIFO (First In FirstOut) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position.

.Two conditions

Overflow–insertionintoqueuewhichisfull

Underflow –deletion from empty queue

.TwoEnds

Front–it'spointtostarting element

Rear–it'spoint tolastelement .

Step1:Declarealltheuserdefinedfunctionswhichareused inqueueimplementation.

Step2:CreateaonedimensionalarraywithabovedefinedSIZE(intqueue[SIZE])

Step3:Definetwointegervariables'front'and'rear'andinitializebothwith'-1'.(intfront=-1,rear= -1)

Step 4: Then implement main method by displaying menu of operations list and make suitablefunction calls to perform operation selected by the user on queue.

EnQueueOperation:

Insertingvalueintothequeue:

In a queue data structure, enQueue() is a function used to insert a new element into the queue. Ina queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following stepsto insert an element into the queue.

12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Newelement=45

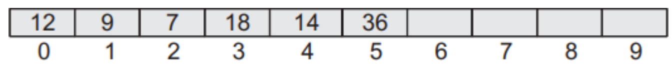
Queue

12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Queue after insertion of a new element

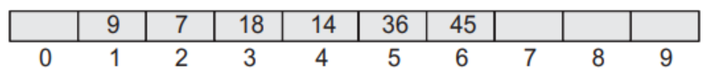
DeQueueOpeartion:

Deleting a value from the Queue In a queue data structure, deQueue() is a function usedto delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the followingsteps to delete an element from the queue.



Deletedelement=12

Queue



Queue after deletion of an element

LINKEDREPRESENTATIONOFQUEUES:

The major problem with the queue implemented using array is, It will work for only fixed numberofdata.Thatmeans,the amountofdatamustbespecifiedinthebeginningitself. Queueusing array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue usinglinked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation ofa queue,the last insertednode is always pointed by'rear'and the first node is always pointed by 'front'.

Example:

In above example, the last inserted node is 50 and it ispointed by 'rear' and the firstinsertednode is 10 and itis pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

To implement queue using linked list, we need to set the following things beforeimplementing actual operations.

Step1:Includeallthe header fileswhichareused intheprogram. Anddeclarealltheuserdefined functions.

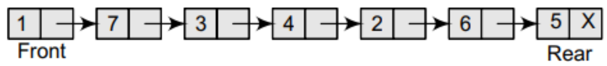
Step2:Definea 'Node'structurewithtwomembersdataand next.

Step3:DefinetwoNode pointers'front'and'rear'andsetbothto NULL.

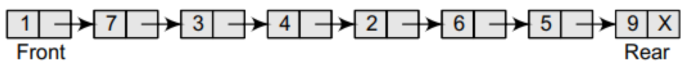
Step4:ImplementthemainmethodbydisplayingMenuoflistofoperationsandmakesuitable functioncallsinthemainmethodtoperformuserselected operation.

EnQueue(value):

Inserting an element into the Queue We can use the following steps to insert a new nodeinto the queue.



Linked queue



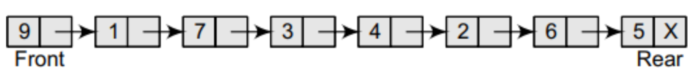
Linked queue after inserting a new node

AlgorithmtoInsertanElementintoQueueusingLinkedList:

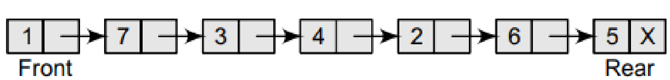
```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

DeQueue():

DeletinganElementfromQueueWecanusethefollowingstepstodeleteanodefromthe queue.



Linked queue



Linked queue after deletion of an element

AlgorithmtoDeleteanElementfromQueueusingLinked List:

```
Step 1: IF FRONT = NULL
             Write "Underflow"
             Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

Display():

DisplayingtheelementsofQueue Wecanusethefollowingstepstodisplaytheelements(nodes)ofa queue...

Step1:CheckwhetherqueueisEmpty(front==NULL).

Step2:IfitisEmptythen,display'Queue isEmpty!!!'andterminatethefunction.

Step3:IfitisNotEmptythen,defineaNodepointer'temp'andinitializewith front.

Step 4:Display 'temp data -> and move it tothenext node. Repeatthe same until'temp'reachesto 'rear' (temp -> next != NULL).

Step5: FinallyDisplay'temp->data->NULL'.

TYPESOFQUEUES:

Aqueuedatastructurecanbeclassifiedinto thefollowingtypes:

1. CircularQueue
2. Deque
3. PriorityQueue

Applicationsof Queues:
.Printingjobmanagement

.clientservermodel

.CPUscheduling-inwhichprocessisexecuted first

.Batchprocessing–manageincomingjobsandprocesstheminorder.

.ResourceAllocation

.Simulation-lineofcustomerswaitingforbank

.Inprocesscommunication–processandmultithreadsystem

DICTIONARIES:

Dictionaryisacollectionofpairsofkeyandvaluewhereeveryvalueisassociated with thecorresponding key.

Basicoperationsthatcanbeperformedondictionaryare:

1. Insertionofvalueinthedictionary
2. Deletionofparticularvaluefromdictionary
3. Searchingofaspecificvalue withthehelp ofkey

LinearListRepresentation

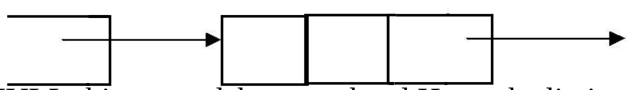
Thedictionarycanberepresentedasalinearlist.Thelinearlistisacollectionofpairand value.There are two method of representing linear list.

1. SortedArray-Anarraydatastructureisusedtoimplementthedictionary.
2. SortedChain-Alinkedlistdatastructureisusedtoimplementthedictionary

Insertionofnew nodein thedictionary:

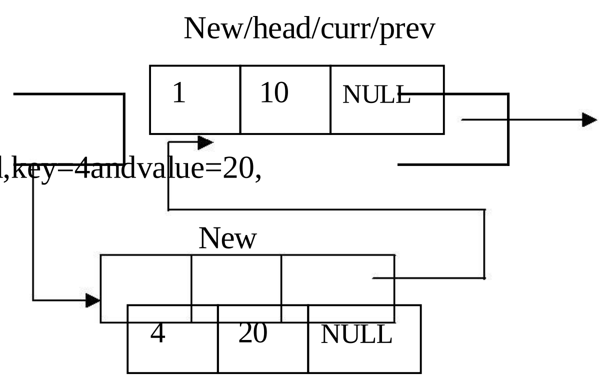
Considerthatinitiallydictionaryisempty thenhead = NULL

Wewillcreateanewnodewithsomekeyandvalue containedinit.

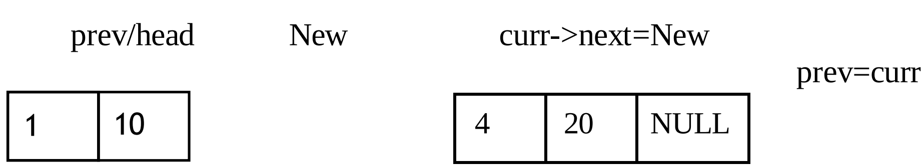


NowasheadisNULL,thisnewnodebecomeshead.Hencethedictionarycontainsonlyone record. this node will be _curr‘ and _prev‘ as well. The _cuur‘ node will always point to current visiting node and _prev‘ will always point to the node previous to _curr‘ node. As now there is only one node in the list mark as _curr‘ node as _prev‘ node.

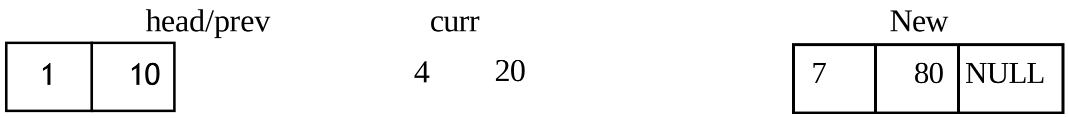
Insertarecord,key=4andvalue=20,



Comparethekeyvalueof_curr‘and_New‘node.IfNew->key>Curr->keythenattach New node to _curr‘ node.

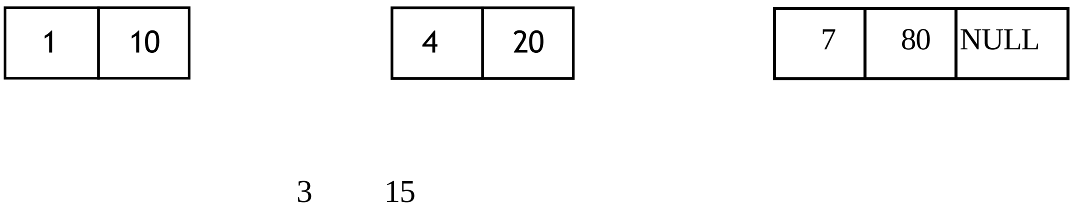


Addanewnode<7,80>then



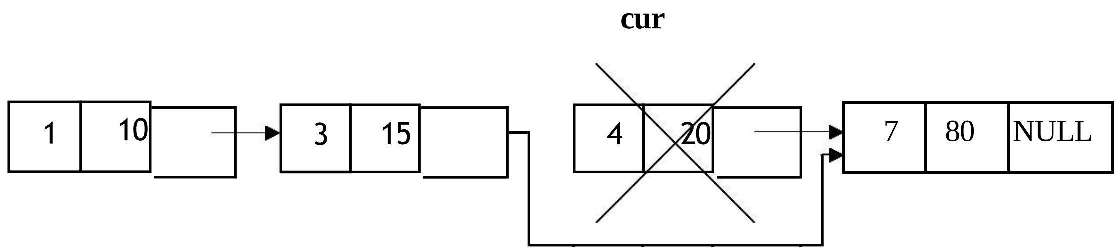
Ifweinsert<3,15>thenwehavetosearchforitproperpositionbycomparingkey

value.(curr->key < New->key) is false. Hence else part will get executed.



Thedelete operation:

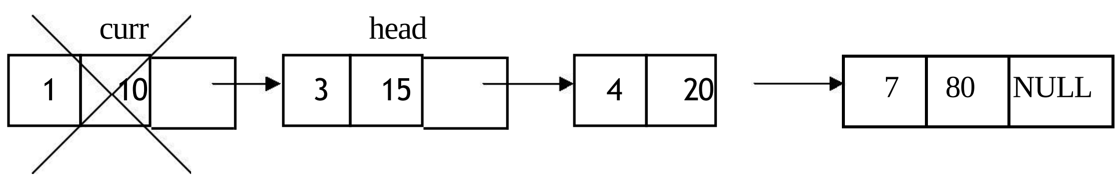
Case 1: Initially assign _head‘ node as _curr‘ node.Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node‘s key value. We will get node which is to be deleted in variable _curr‘. The node given by variable _prev‘ keeps track of previous node of _cuu‘ node. For eg, delete node with key value 4 then



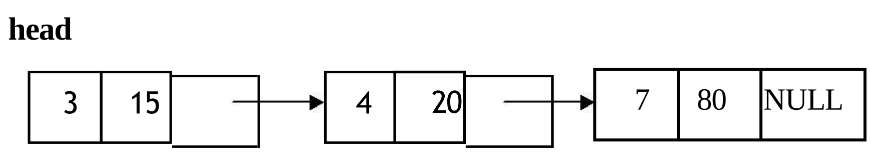
se2:

Ifthenodetobedeletedishead
nodei.e.. if(curr==head)

Then,simplymake_head‘nodeasnextnodeanddelete_curr‘

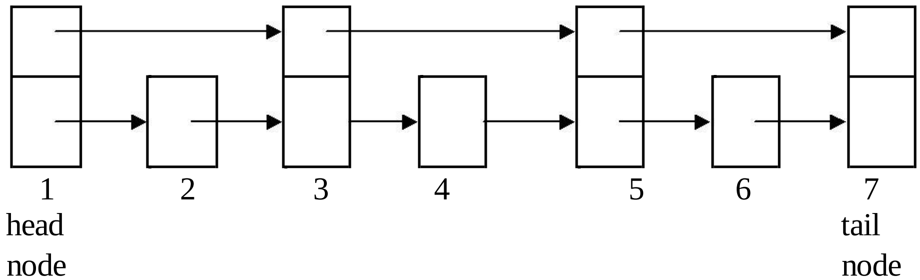


Hencethelist becomes



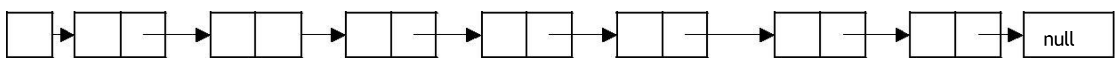
SKIPLISTREPRESENTATION

Skiplistisavariantlistforthelinkedlist.Skiplistsaremadeupof a series of nodes connected one after the other. Each node contains a key and value pair as wellasoneormorereferences,orpointers,tonodesfurtheralonginthelist.Thenumberof references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level. There are two special nodes in the skip list one is head node which is the starting node of thelist and tail node is the last node of the list

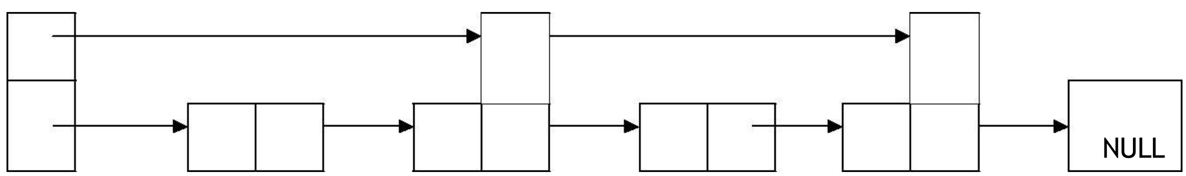


The skip list is an efficient implementation of dictionary using sorted chain. This is becauseinskiplisteachnodeconsistsofforwardreferencesofmorethanonenodeata time.

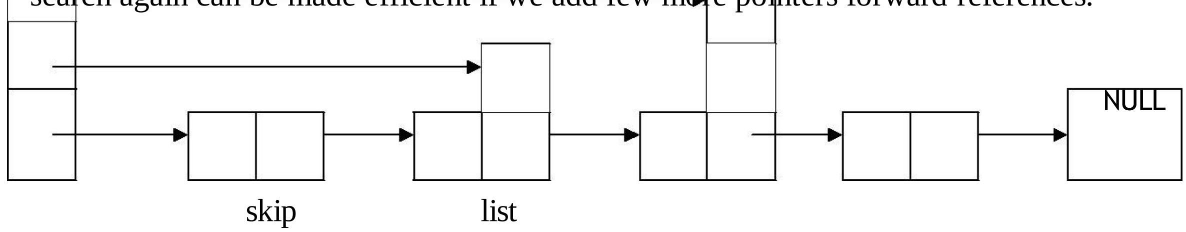
Eg:



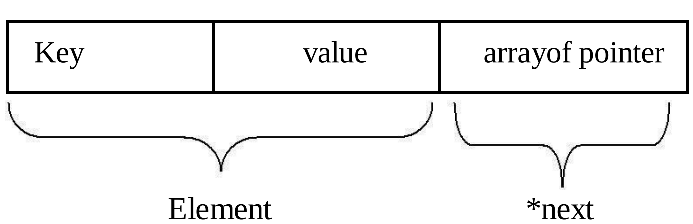
Now to search any node from above given sorted chain we have to search the sortedchain from head nodeby visiting each node. But this searching timecan be reduced if we add one level in every alternate node. This extra level contains the forward pointer of some node.That means in sorted chain come nodes can holds pointers to more than one node.



Ifwewanttosearchnode40fromabovechaintherewewillrequirecomparativelyless time.This search again can be made efficient if we add few more pointers forward references.



Theindividualnodelookslikethis:



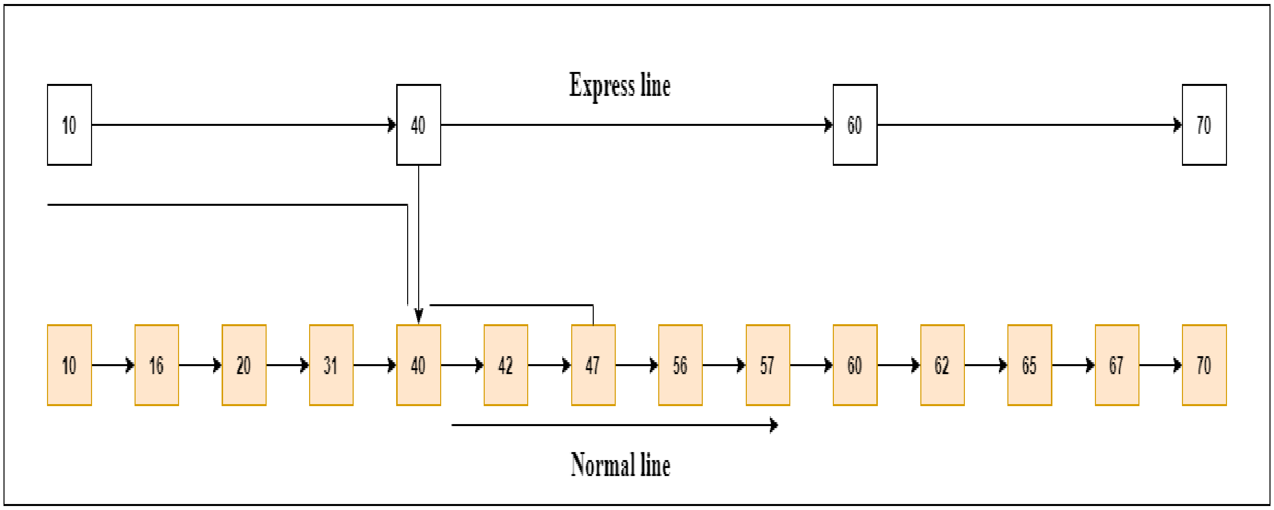
WorkingoftheSkiplist

Let's take an example to understand the working of the skip list. In this example, we have 14 nodes, such that these nodes are divided into two layers, as shown in the diagram.

Thelowerlayerisacommonlinethatlinksallnodes,andthetoplayerisanexpresslinethatlinks onlythemainnodes,asyoucanseeinthe diagram.

Suppose you want tofind 47in this example. You willstart the search from the first nodeof the express line and continue running ontheexpress line until you find a node that is equal a 47 or more than 47.

You can see in the example that 47 doesnot exist in the expressline, so you search fora node of lessthan 47, which is40.Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.



Note: Onceyoufind anode likethis on the "express line", yougofromthis nodetoa "normallane"usinga pointer, andwhenyousearchforthenode in the normal line.

SkipListBasicOperations

Therearethefollowingtypesofoperationsintheskiplist.

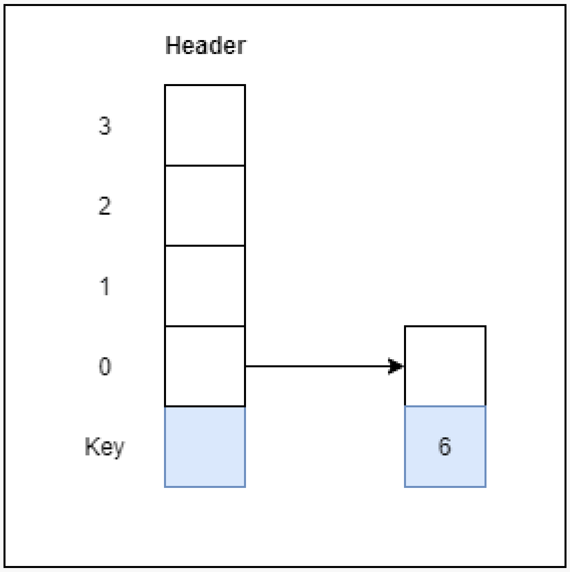
- **Insertionoperation:**Itisusedtoaddanewnodetoaparticularlocationinaspecificsituation.
- **Deletionoperation:**Itisusedtodeleteanodeinaspecificsituation.
- **SearchOperation:**Thesearchoperationisusedtosearch aparticularnodeinaskiplist.

Example1: Createaskiplist,wewanttoinsert thesefollowingkeysin theemptyskiplist.

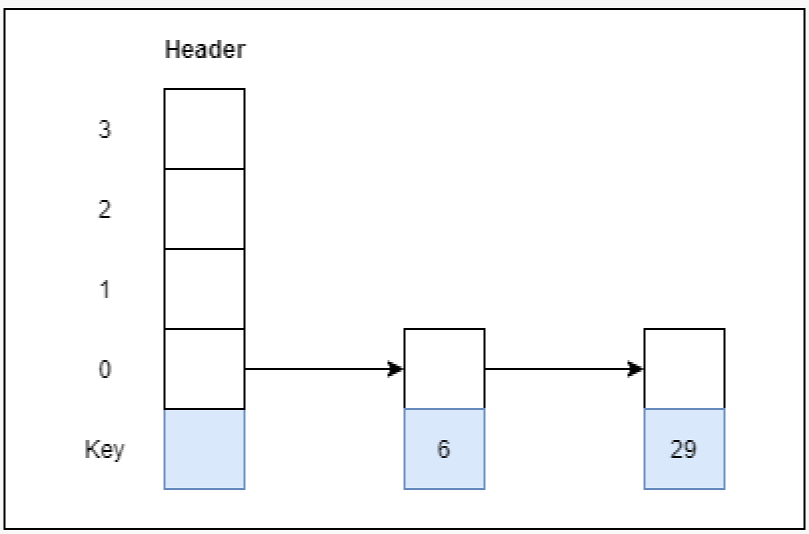
1. 6withlevel1.
2. 29withlevel1.
3. 22withlevel4.
4. 9withlevel3.
5. 17withlevel1.
6. 4withlevel2.

Ans:

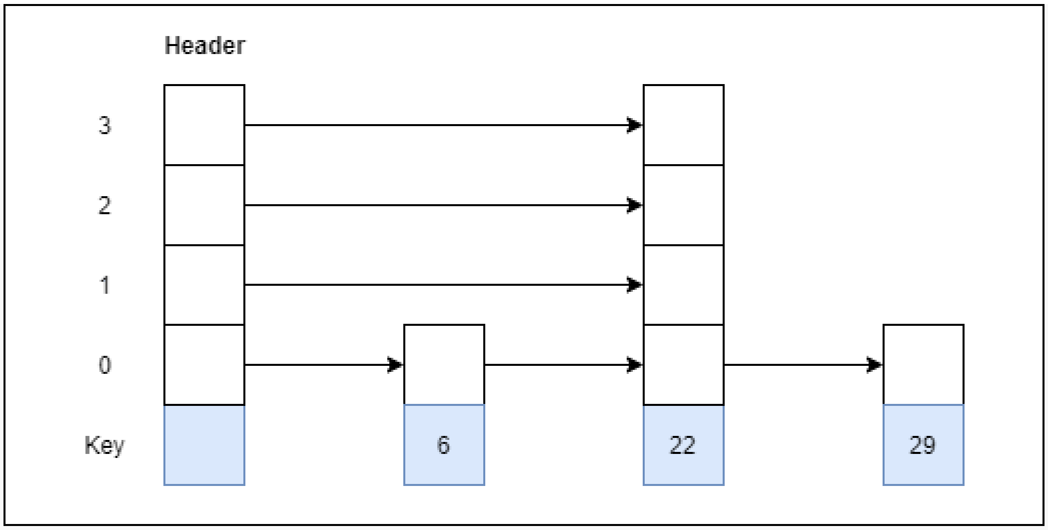
Step1: Insert6withlevel1



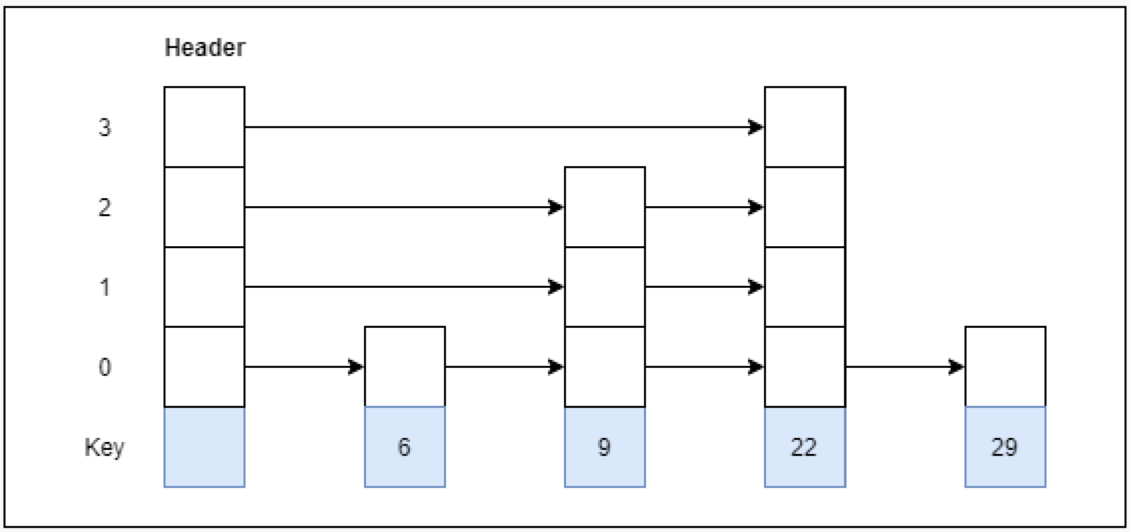
Step2: Insert29with level1



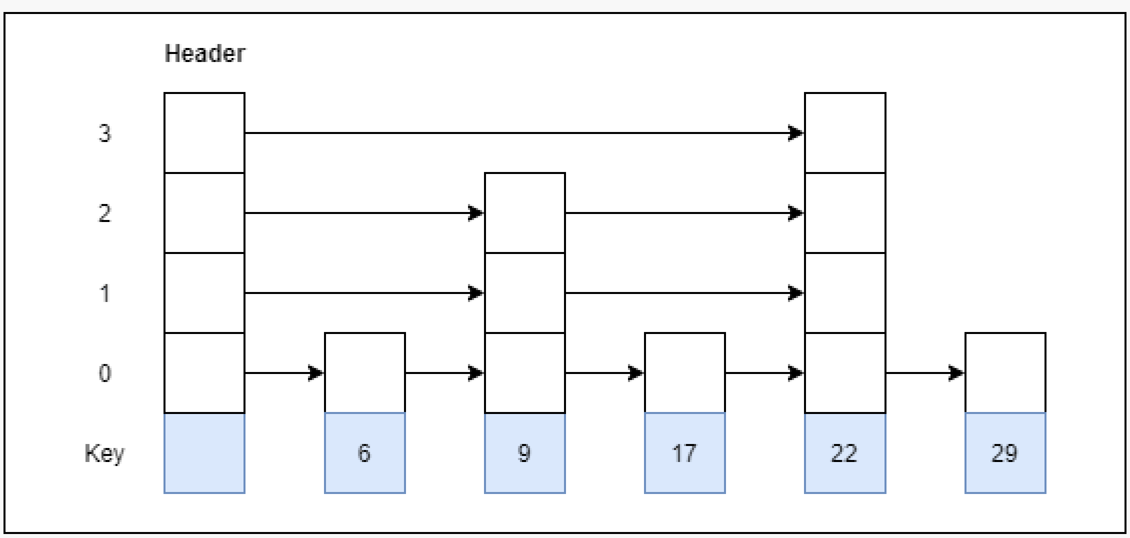
Step3: Insert22withlevel4



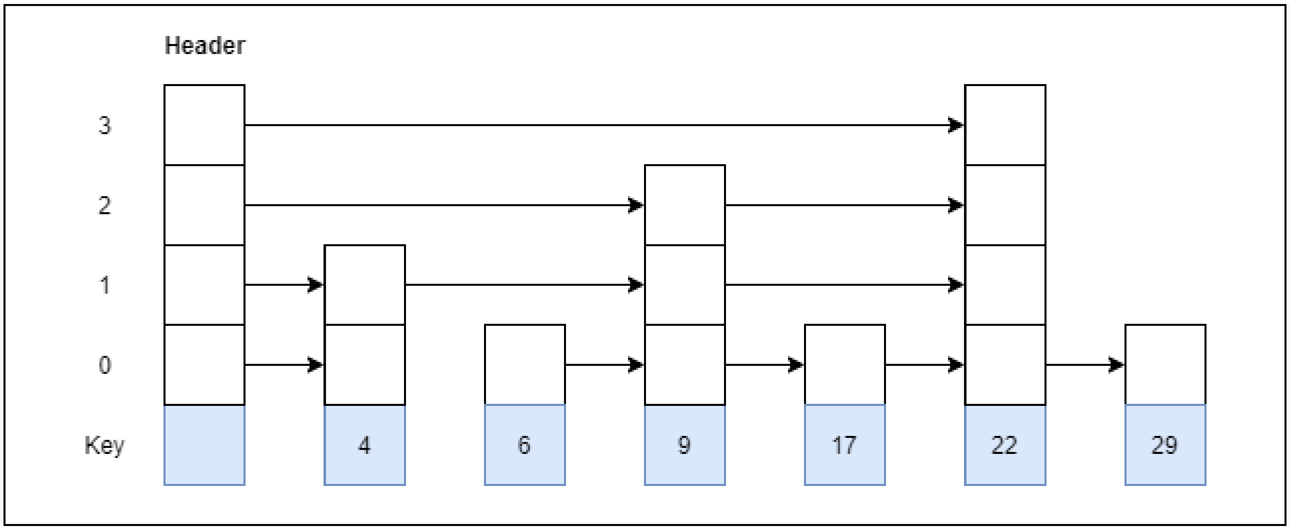
Step4: Insert9withlevel3



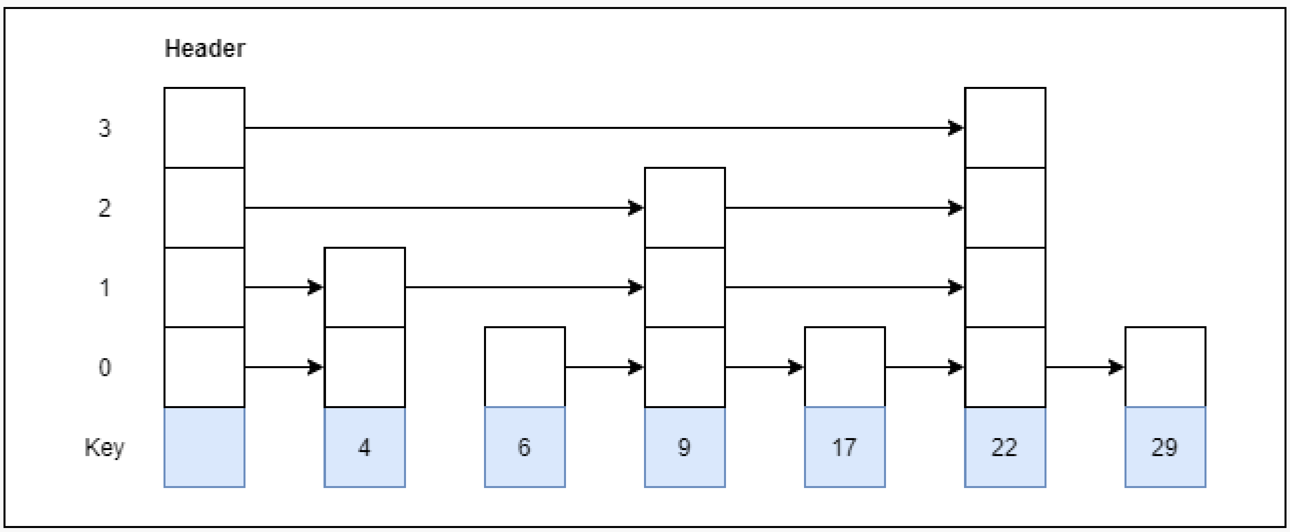
Step5: Insert17withlevel1



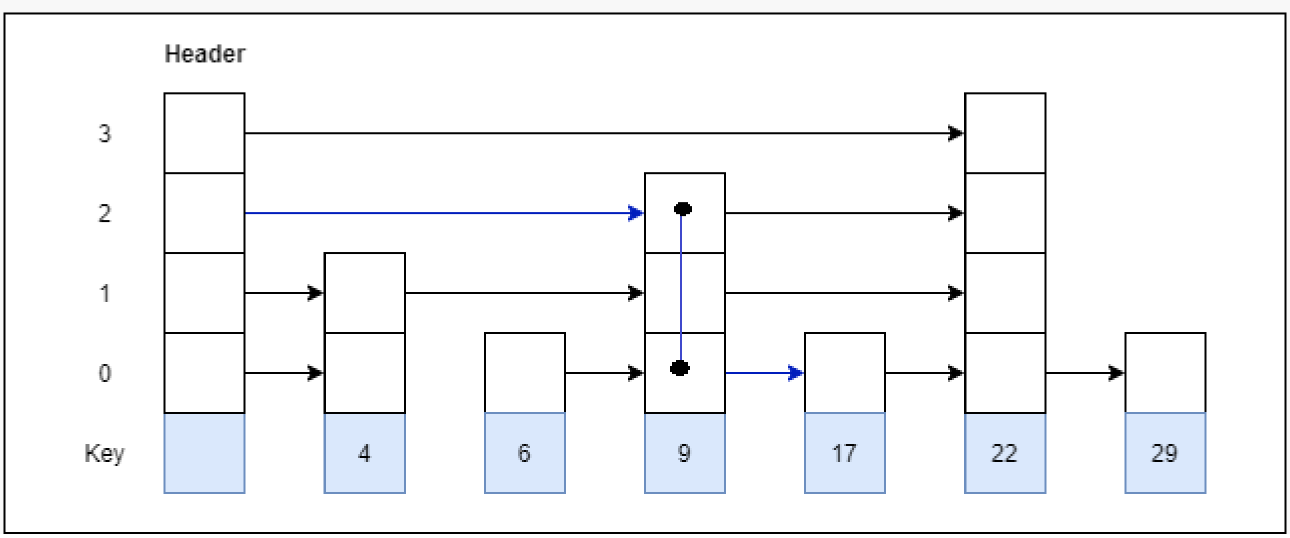
Step6: Insert4withlevel2



Example2: Considerthisexamplewherewewanttosearchforkey17.



Ans:



AdvantagesoftheSkiplist

1. Ifyouwanttoinsertanewnodeintheskiplist,thenitwillinsertthenodeveryfastbecausethereareno rotationsintheskiplist.
2. Theskiplistissimpletoimplementascomparedtothehashtableandthebinarysearchtree.
3. Itisverysimpletofindanodeinthelistbecauseitstoresthenodesinsortedform.
4. Theskiplistalgorithmcanbemodifiedveryeasilyinamorespecificstructure,suchasindexableskiplists,trees,orpriority queues.
5. Theskiplistisarobustandreliablelist.

DisadvantagesoftheSkiplist

1. Itrequiresmorememorythanthebalancedtree.
2. Reversesearchingisnotallowed.
3. Theskiplistsearchesthenodemuchslowerthanthelinkedlist.

Searching:

Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the key_val. If the node key is less than thekey_val,thesearchcontinuesmovingforwardatthesamelevel.Ifothe other hand, thenode key is equal to or greater than the key_val, the search drops one level and continues forward. This process continues until the desired key_val has been found if it is present in the skip list. If it is not, the search will either continue at the end of the list or until the first key witha value greater than the search key is found.

Insertion:

Therearetwotasksthatshould bedonebeforeinsertion operation:

1. Before insertion of any node the place for this new node in the skip list is searched. Hence before any insertion to take place the search routine executes. The last[] array in the search routine is used to keep track of the references to the nodes where the search, drops down one level.

2. Thelevelforthenewnodeisretrievedbytheroutinerandomelevel()

HASHTABLEREPRESENTATION:

- Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key.
- Using the hash key the required piece of data can be searched in the hash table by few or morekeycomparisons. The searching time isthen dependent uponthesize ofthe hashtable.
- The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.

HASH FUNCTION:

A **Hash Function** is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function **maps** a significantnumber or string to a small integer that can be used as the **index** in the hash table.

The pair is of the form **(key, value)**, where for a given key, one can find a value using some kind of a "function" that maps keys to values. The key for a given object can be calculated using a function called a hash function. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i].

TypesofHashfunctions:

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

1. **DivisionMethod.**
2. **MidSquareMethod.**
3. **FoldingMethod.**
4. **MultiplicationMethod.**

1. DivisionMethod:

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:

h(K)=kmodM

Here,

***k**isthekeyvalue,and*

***M**isthesizeofthehashtable.*

Itisbestsuitedthat **M** isaprimenumberasthatcanmakesurethekeysaremoreuniformly distributed. The hash function is dependent upon the remainder of a division.

Example:

Iftherecord54,72,89,37isplaced inthe hashtableand ifthe tablesize is10 then

h(key) = record % table size

54%10=4	

72%10=2	
89%10=9	
37%10=7	

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

2. <u>MidSquareMethod:</u>

Themid-squaremethodisaverygoodhashingmethod.Itinvolvestwostepstocomputethe hash value-

1. Squarethevalueofthekey ki.e.k^2
2. Extractthemiddle**r**digitsasthehashvalue.

Formula:

h(K)= h(kxk)

Here,

k*isthekeyvalue.*

Thevalueof**r**canbedecidedbasedonthesizeofthetable.

Example:

Considerthatifwewanttoplace arecord3111

then$\mathbf{3111}^2$=9678321forthehashtableofsize1000

H(3111) = 783 (the middle 3 digits)

3. <u>DigitFoldingMethod:</u>

Thismethodinvolvestwosteps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has thesame number of digits exceptfor the last part that can have lesser digits than the other parts.
2. Addtheindividualparts.Thehashvalueisobtainedbyignoringthelastcarryifany.

Formula:

k=k1,k2,k3,k4,…..,kn

s=k1+k2+k3+k4+….+knh(K)=s

Here,

s*isobtainedbyaddingthepartsofthekey**k***

Example:

k=12345

k1=12,k2=34,k3 =5

s =k1+k2+k3

=12+34+5

=51

h(K) =51

Note:

The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

4. MultiplicationMethod

Thismethodinvolvesthefollowingsteps:

1. ChooseaconstantvalueAsuchthat0<A<1.
2. MultiplythekeyvaluewithA.
3. ExtractthefractionalpartofkA.
4. Multiplytheresultoftheabovestepbythesizeofthehashtablei.e.M.
5. The resultinghashvalueisobtainedby takingthefflooroftheresultobtainedinstep4.

Formula:

h(K)=floor(M(kAmod1))

Here,

***M**isthesizeofthehashtable.*

***k**isthekeyvalue.*

***A**isaconstantvalue.*

Example:

k=12345

A=0.357840

M=100

*h(12345)=floor[100(12345*0.357840mod1)]*

=floor[100(4417.5348mod1)]

=floor[100(0.5348)]

=floor[53.48]

=53

(OR)

Theformulaforcomputingthehashkeyis:

H(key)=floor(p*(fractionalpartofkey*A))wherepisintegerconstant andAisconstant real number.

DonaldKnuthsuggestedtouseconstantA=0.61803398987

Ifkey107andp=50then

H(key)=floor(50*(107*0.61803398987))

=floor(3306.4818458045)

=3306

At3306locationinthehashtabletherecord 107willbeplaced.

COLLISION:

The hash function is a function that returns the key value using which the record can be placed in the hash table. Thus this functionhelps us inplacing the record in the hash table at appropriate position and due to this we can retrieve the record directly from that location. This function need to be designed very carefully and it should not return the samehash key address for two different records. This is an undesirable situation in hashing.

Definition:

The situation in which the hash function returns the same hash key (home bucket) for more than one record is called **collision** and two same hash keys returned for different records is called **synonym.**

Similarly when there is no room for a new pair in the hash table then such a situation is called **overflow.** Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions.

Forexample:

Considerahashfunction.

H(key)=recordkey%10 having thehashtablesize of10

Therecordkeystobeplacedare

67 131,44, 43, 78, 19, 36, 57and77

131%10=1
44%10=4
43%10=3
78%10=8
19%10=9
36%10=6
57%10=7
77%10=7

0	
1	131
2	
3	43
4	44
5	
6	36
7	57
8	78
9	19

Nowif wetrytoplace 77inthehashtablethen weget thehashkeytobe 7and at index7 already the recordkey 57 is placed. This situation is called collision. From theindex7 ifwelook for next vacant position at subsequent indices 8,9 then we find that there is no room to place 77in the hash table. This situation is called overflow.

COLLISIONRESOLUTIONTECHNIQUES:

If collision occurs then itshouldbe handledby applying some techniques.Such a techniqueis called collision handling technique.

1. Chaining

2. Openaddressing(linearprobing)

3.Quadratic probing

4. Doublehashing

5. Doublehashing

6.Rehashing

CHAINING:

Incollisionhandlingmethodchainingisaconceptwhichintroducesanadditional fieldwith data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then alinked list(chain) is maintained at the home bucket.

ForExample:

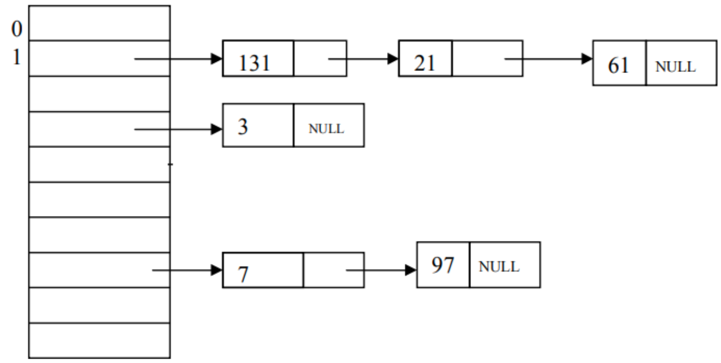
Considerthekeystobeplacedintheirhomebucketsare 131,

3, 4, 21, 61, 7, 97, 8, 9

thenwewillapplya hash functionasH(key)=key%D

Where D is the size of table. The hash table will be-

Here D = 10



Achainismaintainedforcollidingelements.forinstance131hasahomebucket(key)1. similarlykey 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

OPENADDRESSING–LINEARPROBING:

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

Forexample:

Considerthatfollowingkeysaretobeinsertedinthehashtable 131, 4,

8, 7, 21, 5, 31, 61, 9, 29

Initially,wewillputthefollowing keysinthehashtable.

WewilluseDivisionhashfunction.That meansthekeysareplacedusingtheformula H(key) =

key % tablesize

H(key)=key% 10

Forinstancetheelement131canbeplaced at

H(key)= 131%10

=1

Index1willbethehome bucket for131.Continuing inthis fashionwewillplace4,8,7. Now the next key to be inserted is 21. According to the hash function

H(key)=21%10

H(key)=1

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element. Therefore 21 will be placed at the index 2. If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is emptyso we will put the element 5 at index 5.

Index	Key	Key	Key
0	NULL	NULL	NULL
1	131	131	131
2	NULL	21	21
3	NULL	NULL	31
4	4	4	4
5	NULL	5	5
6	NULL	NULL	61
7	7	7	7
8	8	8	8
9	NULL	NULL	NULL

after placing keys 31, 61

The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0th index.

QUADRATICPROBING:

Quadratic probing operates by taking the original hash value and addingsuccessive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H(key) = (Hash(key) + i^2) \% m)$$

where mcan betablesizeoranyprimenumber.

Example:

wehavealist ofsize20(m=20).Wewant toputsomeelementsinlinearprobing fashion. The

elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}

x	$h(x, i) = (h'(x) + i^2) \bmod 20$
96	i = 0, h(x, 0) = 16
48	i = 0, h(x, 0) = 8
63	i = 0, h(x, 0) = 3
29	i = 0, h(x, 0) = 9
87	i = 0, h(x, 0) = 7
77	i = 0, h(x, 0) = 17
48	i = 0, h(x, 0) = 8 i = 1, h(x, 1) = 9 i = 2, h(x, 2) = 12
65	i = 0, h(x, 0) = 5
69	i = 0, h(x, 0) = 9 i = 1, h(x, 1) = 10
94	i = 0, h(x, 0) = 14
61	i = 0, h(x, 0) = 1

Hash Table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	61		63		65		87	48	29	69		48		94		96	77		

DOUBLE HASHING:

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert.

Therearetwoimportantrulestobefollowedforthesecondfunction:

- Itmustneverevaluatetozero.
- Mustmakesurethatallcellscanbeprobed. The

formula to be used for double hashing is:

$$H_1(\text{key}) = \text{key mod tablesize}$$
$$H_2(\text{key}) = M - (\text{key mod } M)$$

whereMisaprimenumbersmallerthanthesizeofthetable.

Considerthefollowingelementstobeplacedinthehashtableofsize10 37, 90,

45, 22, 17, 49, 55

InitiallyinserttheelementsusingtheformulaforH1(key). Insert

37, 90, 45, 22,49

<table>
<tr><td>H1(37)=37%10=
7</td><td rowspan="5"><table><tr><td>Key</td></tr><tr><td>90</td></tr><tr><td></td></tr><tr><td>22</td></tr><tr><td></td></tr><tr><td></td></tr><tr><td>45</td></tr><tr><td></td></tr><tr><td>37</td></tr><tr><td></td></tr><tr><td>49</td></tr></table></td></tr>
<tr><td>H1(90)=90%10=
0</td></tr>
<tr><td>H1(45)=45%10=
5</td></tr>
<tr><td>H1(22)=22%10=
2</td></tr>
<tr><td>H1(49)=49%10=
9</td></tr>
</table>

Nowif17tobeinsertedthen

H1(17) = 17 % 10 = 7

H2(key) =M–(key%M)

HereMisprimenumbersmallerthanthesizeofthetable.Prime numbersmallerthantablesize 10 is 7

Hence M =
7H2(17)=7-(17%7)

=7–3=4

Thatmeanswehavetoinserttheelement17at4placesfrom37.Inshortwehajumps. Therefore the 17 will be placed at index 1.

Key
90
17
22
45
37
49

Nowtoinsertnumber55

H1(55) =55%10=5 →Collision

H2(55)= 7-(55%7)

=7–6=1

That meanswehavetotakeonejumpfromindex5toplace55. Finally

the hash table will be –

Key
90
17
22
45
55
37
49

REHASHING:

When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

Alltheentries intheoriginalhashtablewillthen havetobemovedtothe newhash table. This is done bytaking each entry, computing its new hash value, and then inserting it inthe new hash table.

Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently. Consider the hash table of size 5 given below. The hash function used is h(x) = x % 5. Rehash the entries into to a new hash table.

0	1	2	3	4
	26	31	43	17

Note that the new hash table is of 10 locations, double the size of the original table.

0	1	2	3	4	5	6	7	8	9

Now, rehash the key values from the old hash table into the new one using hash function—h(x) = x % 10.

0	1	2	3	4	5	6	7	8	9
	31		43			26	17		

Advantages:

1. Thistechniqueprovidestheprogrammeraflexibilityto enlargethetablesizeifrequired.

2. Onlythespacegetsdoubled withsimplehashfunctionwhichavoidsoccurrenceofcollisions.

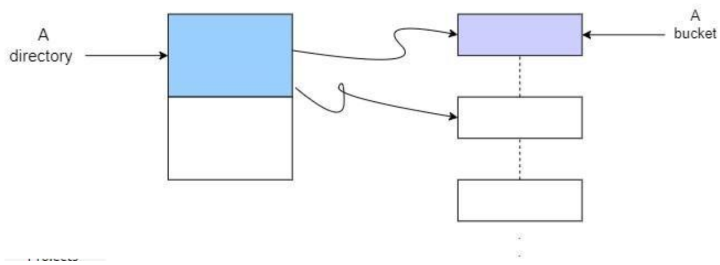
ExtendibleHashing:

Extendible hashing is a dynamic approach to managing data. In this hashing method, flexibility is acrucial factor. This method catersto flexibility so that even the hashing function dynamically changes according to the situation and data type.

Algorithm

Thefollowingillustrationrepresentstheinitialphasesofour hashtable:

The following illustration represents the initial phases of our hashtable:



Directories and **buckets** are two key terms in this algorithm. *Buckets* are the holders of hashed data, while *directories* are the holders of pointers pointing towards these buckets. Each directory has a unique ID.

The following pointsexplainhowthealgorithmwork:

1. Initializethebucketdepthsandtheglobaldepthofthedirectories.
2. Convertdataintoabinaryrepresentation.
3. Considerthe"globaldepth"numberoftheleastsignificantbits(LSBs)ofdata.
4. Mapthedataaccording totheIDofadirectory.
5. Checkforthefollowingconditionsifabucket overflows(ifthenumberofelementsina bucket exceeds the set limit):
   I. **Global depth == bucket depth**: Split the bucket into two and increment the globaldepthandthebuckets'depth.Re-hashtheelementsthat werepresent inthe split bucket.
   II. **Globaldepth>bucketdepth**:Splitthebucketintotwoandincrementthebucket depth only. Re-hash the elements that were present in the split bucket.
6. Repeatthestepsaboveforeachelement.

Example

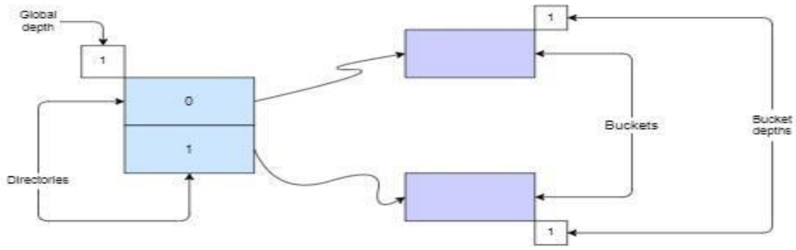
Let'stakethefollowingexampleto seehowthis hashingmethod workswhere:

- Data= {28,4,19,1,22,16,12,0,5,7}
- Bucket limit=3

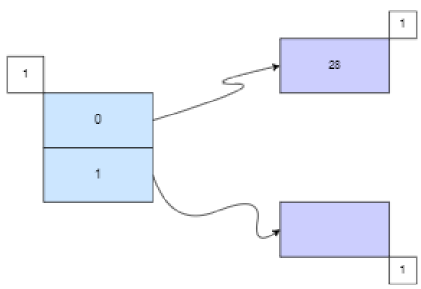
Convertthedataintobinaryrepresentation:

- 28=11100
- 4=00100
- 19=10011
- 1=00001
- 22=10110
- 16=10000
- 12=01100
- 0=00000
- 5=00101
- 7=00111

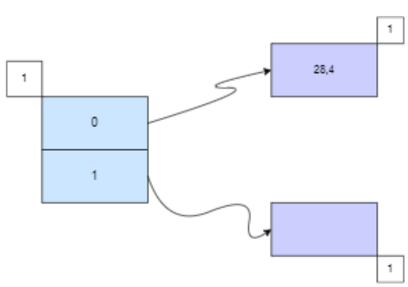
Thefollowingslideshowrepresentstheremaining steps:



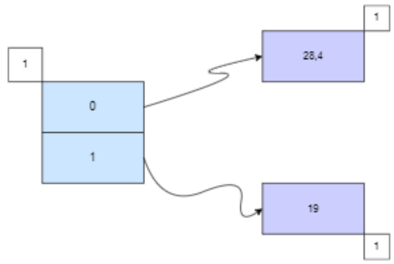
Initialize the hash table with two initial directories and buckets. Set the global depth and bucket depth to 1



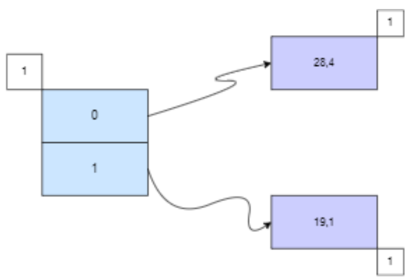
Consider 28 = 11100. Take the global depth number of LSBs, which is currently one. We consider 0 as it is the rightmost LSB in 11100, and 28 is placed in the "0" ID bucket



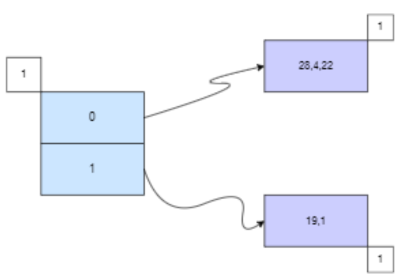
Consider 4 = 00100. Add it to the "0" ID bucket as one LSBs will be considered



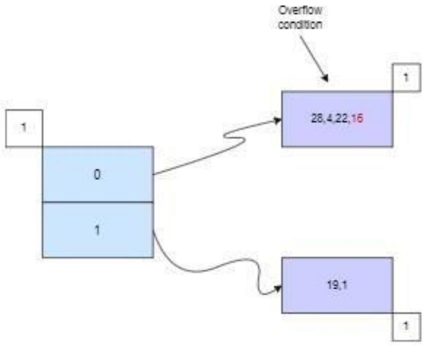
Consider 19 = 10011. Add it to the "1" ID bucket as one LSBs will be considered



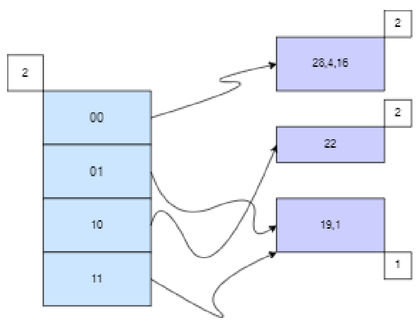
Consider 1 = 00001. Add it to the "1" ID bucket as one LSBs will be considered



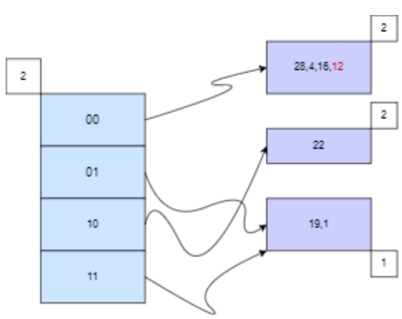
Consider 22 = 10110. Add it to the "0" ID bucket as one LSBs will be considered



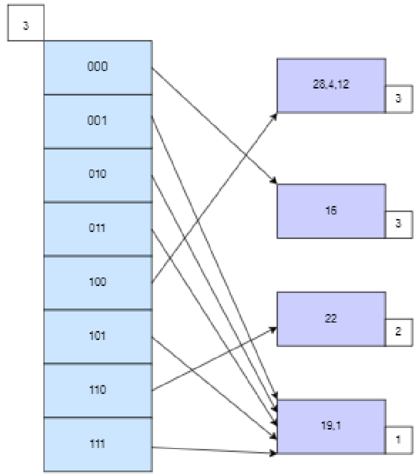
Consider 16 = 10000. Adding it to the "0" ID bucket makes it overflow, and the first condition is satisfied where global depth == bucket depth



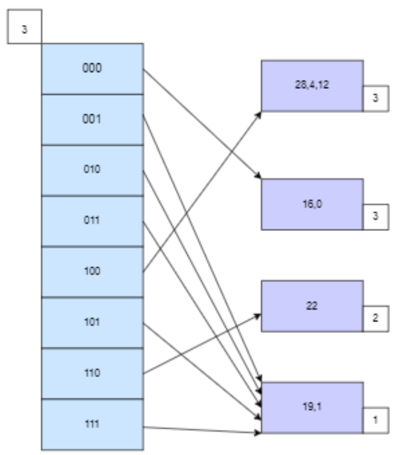
Split the bucket, grow directories, increment the global and bucket depths, and re-hash the bucket elements according to two LSBs



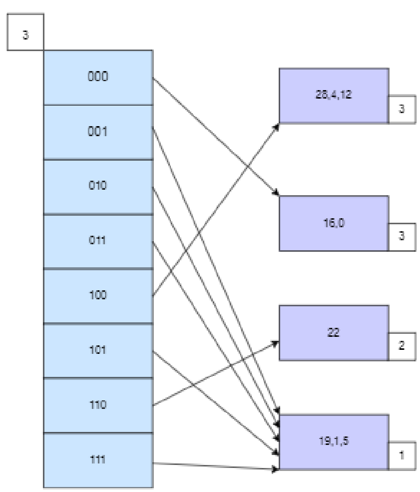
Consider 12 = 01100. Adding it to the "00" ID bucket according to two LSBs makes it overflow, and satisfies the first condition



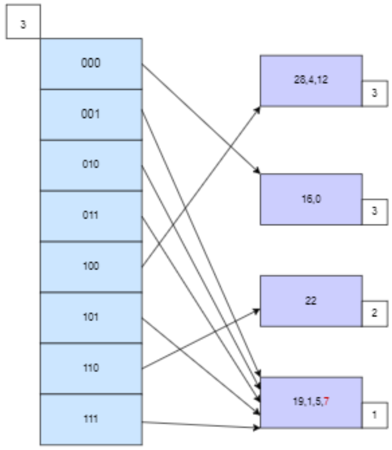
Repeat the previous step and re-hash the elements of the bucket according to three LSBs



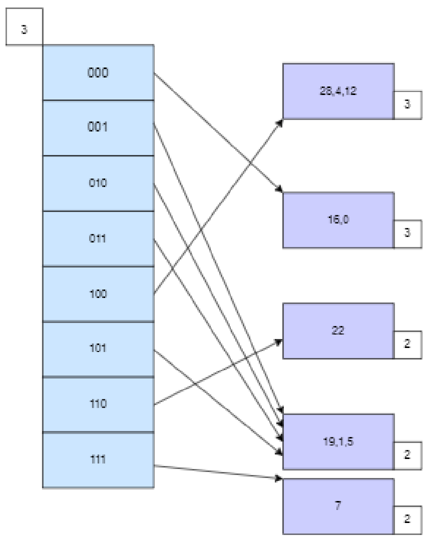
Consider 0 = 00000. Add it to the "000" ID bucket as three LSBs are considered



Consider 5 = 00101. Add it to the "101" bucket as three LSBs are considered



Consider 7 = 00111. Adding it to the "111" bucket according to three LSBs makes it overflow



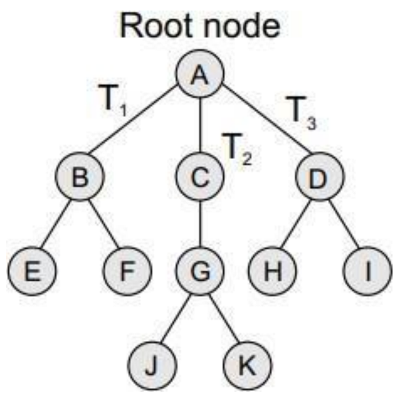
The bucket depth is now less than global depth. So, the second condition is fulfilled, and now only the bucket split step is executed. Elements are re-hashed according to the global depth, which is still 3.

MODULE-3

TREES

Tree is non-linear data structure that consists of root node and potentially many levels of additional nodes that form a hierarchy.

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.



Tree

Where nodeA isthe rootnode;nodesB,C,andDarechildrenoftherootnode.

BasicTerminology:

Rootnode: TherootnodeRisthetopmostnodeinthetree.IfR=NULL,thenitmeansthetreeis empty.

Sub-trees: Iftheroot nodeRisnot NULL,thenthetreesT1,T2,andT3arecalled thesub-treesofR.

Leafnode: Anodethathasnochildren is calledtheleafnodeortheterminalnode.

Path: Asequenceofconsecutiveedgesiscalled apath. Forexample, in abovediagram, the pathfromthe root node A to node I is given as: A, D, and I.

Ancestornode: Anancestor ofanodeis anypredecessornodeonthepathfromroot tothat node.

The root node does not have any ancestors. In the tree given in above diagram, nodes A, C, and G are the ancestors of node K.

Descendant node: A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in the above diagram, nodes C, G, J, and K are the descendants of node A.

Level number: Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given byparent’s level number + 1.

Degree: Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.

In-degree: In-degree of anodeis the numberofedges arriving at that node.

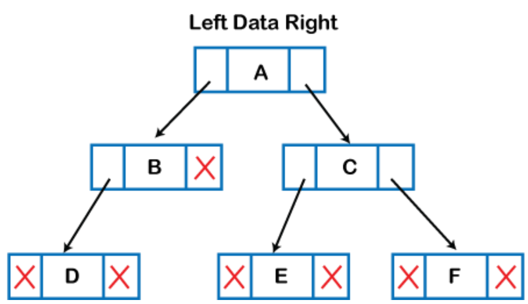
Out-degree: Out-degreeofanodeisthenumberofedgesleavingthat node.

Siblings: Nodes with the same parent.

Height: Numberofnodes whichmustbetraversedfromtherootto thereachaleafofatree.

ImplementationofTree:

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

Inprogramming,thestructureofanodecanbedefined as:

```
structnode
{
 intdata;
struct node *left;
structnode*right;
}
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure ofthe node for generic trees would be different as compared to the binary tree.

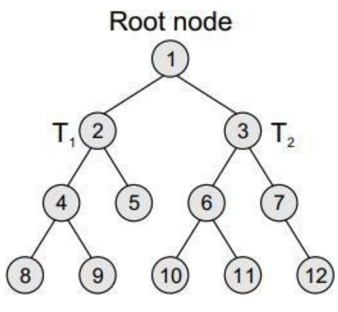
Applicationsoftrees

Thefollowing aretheapplicationsoftrees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are inthe formofthenaturallyhierarchicaldataandstored intheformoftrees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** Itisaspecialkindoftreethatisusedtostorethedictionary.Itisafastandefficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priorityqueues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used tostore the data in routing tables in the routers.

BinaryTrees:

Abinarytreeisadatastructurethatisdefinedas acollectionofelements called nodes. In abinary tree,thetopmostelementis called therootnode,and eachnodehas0,1,orat themost2 children. A node that has zero children is called a leaf node or a terminal node. Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.



Binary tree

In the above diagram, R is the root node and the two trees T1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R. Note that the left sub-tree of the root node consists of thenodes: 2, 4, 5, 8, and 9. Similarly, the right sub-tree ofthe root node consists ofnodes: 3, 6, 7, 10, 11, and 12.

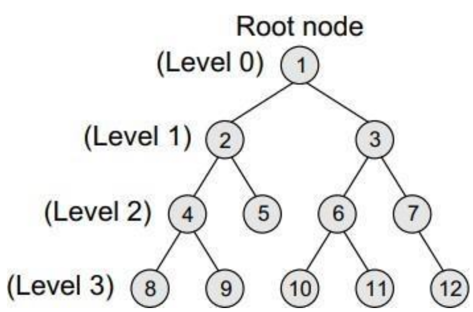
In the tree, root node 1 has two successors: 2 and 3. Node 2 has two successor nodes: 4 and 5. Node 4 has two successors: 8 and 9. Node 5 has no successor. Node 3 has two successor nodes:6and7.Node6hastwosuccessors:10and11.Finally,node7hasonlyonesuccessor: 12.

A binary tree is recursive by definition as every node in the tree contains a left sub-tree and a right sub-tree. Even the terminal nodes contain an empty left sub-tree and an empty right sub-tree. In the above diagram, nodes 5, 8, 9, 10, 11, and 12 have no successors and thus said to have empty sub-trees.

Terminology:

Parent: If N is any node in T that has left successor S1 and right successor S2, then N is called the parent of S1 and S2. Correspondingly, S1 and S2 are called the left child and the right childof N. Every node other than the root node has a parent.

Level number: Every node in the binary tree is assigned a level number . The root node is defined to be at level 0. The left and the right child of the root node have a level number 1. Similarly,everynodeisatonelevelhigherthanitsparents.Soallchildnodesaredefinedtohave level number as parent's level number + 1.



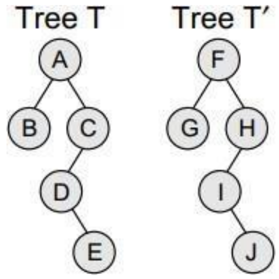
Levels in binary tree

Degree of a node It is equal to the number of children that a node has. The degree of a leaf node is zero. For example, in the tree, degree of node 4 is 2, degree of node 5 is zero and degree of node 7 is 1.

Sibling: All nodes that are at the same level and share the same parent are called siblings (brothers).Forexample,nodes2and3;nodes4and5;nodes6and7;nodes8and9;andnodes 10and11are siblings.

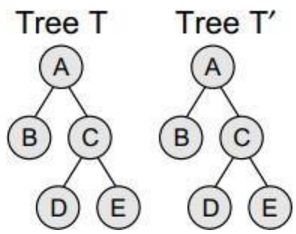
Leaf node: Anodethat hasnochildreniscalled aleafnodeoraterminal node.Theleafnodesin the tree are: 8, 9, 5, 10, 11, and 12.

Similar binary trees: Two binary trees T and T¢ are said to be similar if both these trees have the same structure.



Similar binary trees

Copies: Two binary trees T and T¢ are said to be copies if they have similar structure and if they have same content at the corresponding nodes. Below diagram shows that T¢ is a copyof T.



T′ is a copy of T

Edge: It is the line connecting a node N to any of its successors. A binary tree of n nodes has exactly n – 1 edges because every node except the root node is connected to its parent via an edge.

Path: A sequence of consecutive edges. For example, in the above diagram, the path from the root node to the node 8 is given as: 1, 2, 4, and 8.

Depth: The depth of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.

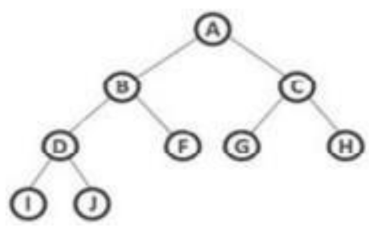
Height of a tree: It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1. A binary tree of height h has at leasthnodesand atmost2^h–1nodes.Thisisbecauseeverylevel willhaveatleastonenodeand can have at most 2 nodes. So, if every level has two nodes then a tree with height h will have at the most 2^h – 1 nodes as at level 0, there is only one element called the root. The height of a binarytree with n nodes is at least $\log_2(n+1)$ and at most n.

In-degree/out-degree of a node: It is the number of edges arriving at a node. The root node is the only node thathas an in-degree equal to zero. Similarly, out-degree of a node is the number of edges leaving that node.

Binary trees are commonly used to implement binary search trees, expression trees, tournament trees, and binary heaps.

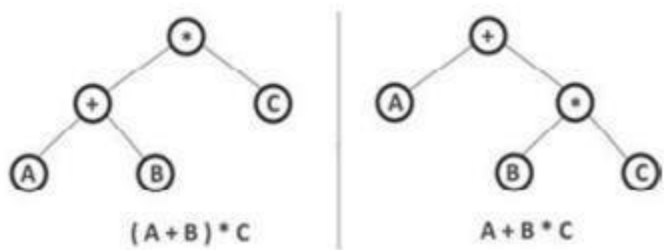
Therearedifferent typesofbinarytreesandtheyare...

1.Strictly Binary Tree: In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows... Abinarytreeinwhicheverynodehaseithertwoorzero numberof children is called Strictly Binary Tree Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.

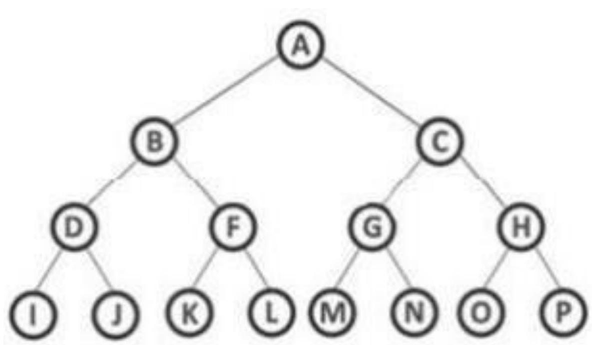


Strictlybinarytreedatastructureisusedtorepresentmathematicalexpressions.

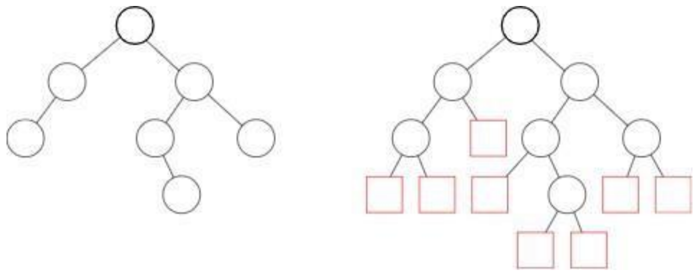
Example



Complete Binary Tree: In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2level number of nodes. For example at level2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes. A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete BinaryTree. Complete binarytree is also called as Perfect BinaryTree



Extended Binary Tree: A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required. The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

BinaryTreeRepresentations

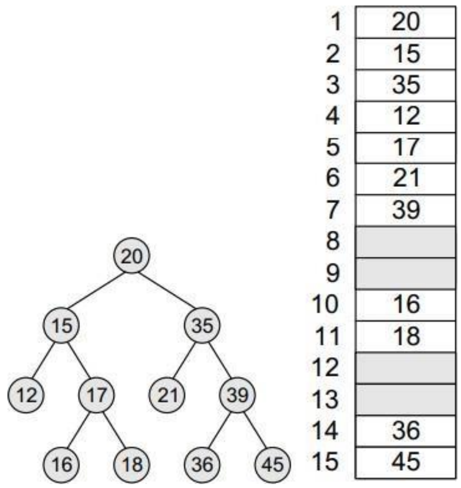
Abinarytreedatastructureisrepresentedusingtwomethods. Those methods are as follows:

1. ArrayRepresentation2. LinkedListRepresentation

Sequentialrepresentation ofbinarytrees:

Sequential representation of trees is done using single or one-dimensional arrays. Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space. A sequential binary tree follows the following rules:

1. Aone-dimensionalarray,calledTREE,isusedtostoretheelementsoftree.
2. The root of the tree will be stored in the firstlocation. That is, TREE will store the data of the root element.
3. Themaximumsizeof thearrayTREEisgivenas(2^h–1),wherehistheheightof the tree.
4. An empty tree orsub-treeisspecifiedusingNULL.If TREE=NULL,thenthetreeis empty.



Binary tree and its sequential representation

Theabovediagramshowsthebinarytreeanditscorrespondingsequentialrepresentation.Thetreehas11 nodes and its height is 4.

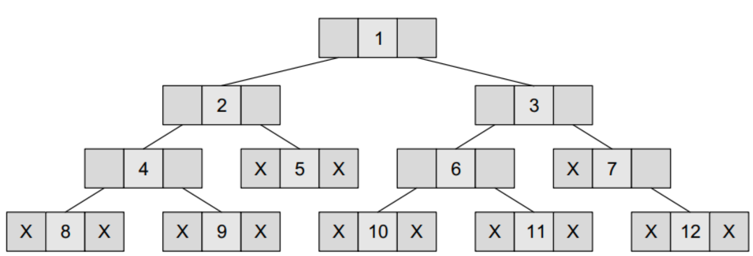
Linked representationofbinarytrees:

Inthelinkedrepresentationofabinarytree,everynodewillhavethreeparts:thedataelement,a pointer to the left node, and a pointer to the right node.

SoinC,thebinarytreeisbuilt withanodetypegivenbelow. struct node

```
{
structnode*left;
int data;
structnode*right;
};
```

Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree. If ROOT = NULL, then the tree is empty. The schematic diagram of the linked representation of the binary tree is shown below. In the below diagram, the left position is used to point to the left child of the node or to store the address of the left child of the node. The middle position is used to store the data. Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node. Empty sub-trees are represented using X (meaning NULL).



Linked representation of a binary tree

TreeTraversals(Inorder,PreorderandPostorder):

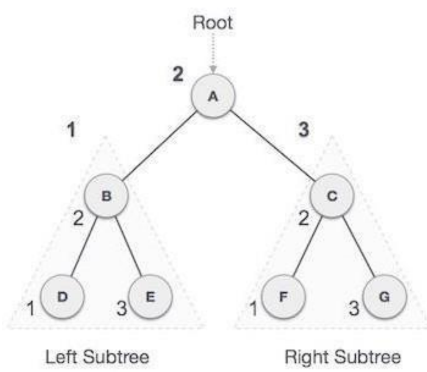
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-orderTraversal
- Pre-orderTraversal
- Post-orderTraversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-orderTraversal:

- In this traversal method,theleftsubtree isvisitedfirst, then therootandlater theright sub-tree. We should always remember that every node may represent a subtree itself.
- Ifabinarytreeistraversed **in-order**,theoutputwillproducesortedkeyvaluesinan ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

D→B→E→A→F→C→G

Algorithm

Untilallnodesaretraversed−

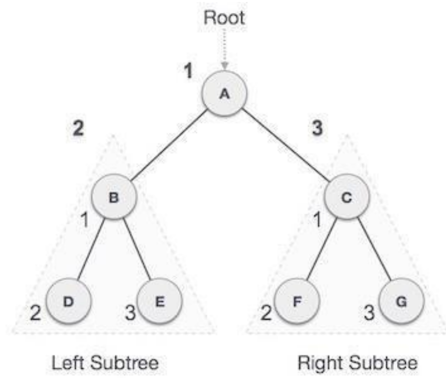
Step 1−Recursivelytraverse leftsubtree.

Step2−Visitrootnode.

Step3−Recursivelytraverserightsubtree.

Pre-orderTraversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



Westartfrom **A**, andfollowingpre-ordertraversal,wefirstvisit **A** itselfandthenmovetoitsleft subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A→B→D→E→C→F→G

Algorithm

Untilallnodesaretraversed−

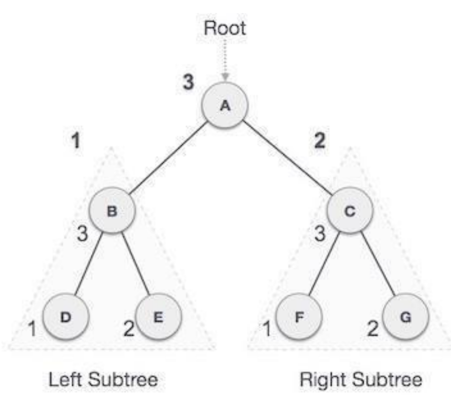
Step1−Visitrootnode.

Step2−Recursivelytraverseleftsubtree.

Step3−Recursivelytraverserightsubtree.

Post-orderTraversal

In this traversal method, the rootnode is visitedlast, hence the name.Firstwe traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversedpost-order.Theprocessgoesonuntilallthenodesarevisited.Theoutputofpost-order traversalofthistreewill be−

D→E→B→F→G→C→A

Algorithm

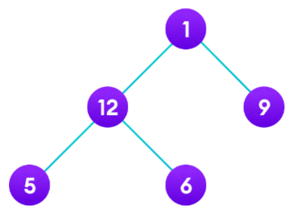
Untilallnodesaretraversed−

Step 1 − Recursively traverse left subtree.

Step2−Recursivelytraverserightsubtree.

Step 3 − Visit root node.

ImplementationinC:



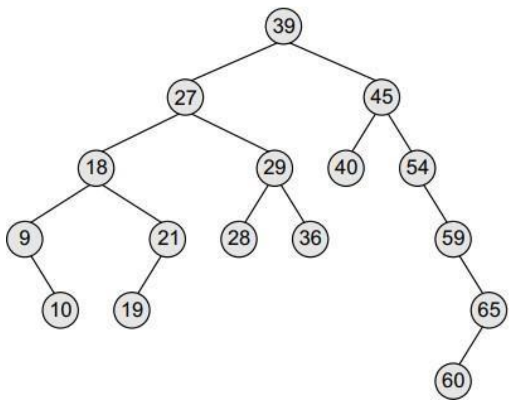

```
//TreetraversalinC
#include <stdio.h>
#include <stdlib.h>
struct node {
  intitem;
  struct node* left;
  structnode*right;
};
//Inordertraversal
voidinorderTraversal(structnode*root){ if
  (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}
//preorderTraversaltraversal
voidpreorderTraversal(structnode*root){ if
  (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
```

```
  preorderTraversal(root->right);
}
//postorderTraversaltraversal
voidpostorderTraversal(structnode*root){ if
  (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);printf("%d
  ->", root->item);
}
//CreateanewNode
structnode*createNode(value){
  structnode*newNode=malloc(sizeof(structnode));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right=NULL;
  return newNode;
}
//Insertontheleftofthenode
structnode*insertLeft(structnode*root,intvalue){
  root->left = createNode(value);
  returnroot->left;
}
//Insertonthe rightofthenode
structnode*insertRight(structnode*root,intvalue){
  root->right = createNode(value);
  returnroot->right;
}
intmain(){
  structnode*root=createNode(1);
  insertLeft(root, 12);
  insertRight(root,9);
  insertLeft(root->left,5);
  insertRight(root->left, 6);
  printf("Inorder traversal \n");
  inorderTraversal(root);
  printf("\nPreordertraversal\n");
```

```
preorderTraversal(root);
printf("\nPostordertraversal
\n"postorderTraversal(root);
}
```

BINARYSEARCHTREES:

A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to everysub-tree in the tree.



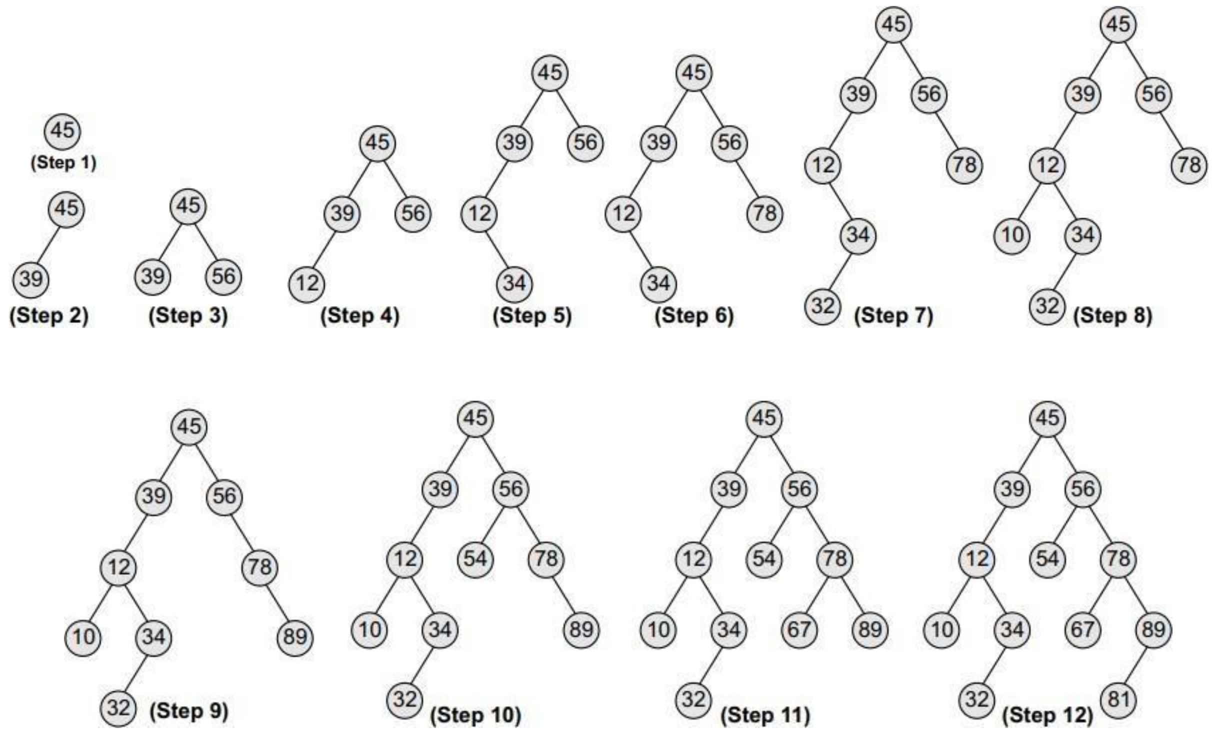
The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the root node. The rightsub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65. Recursively, each of the sub-trees also obeys the binary search tree constraint. For example, in the left sub-tree of the root node, 27 is the root and all elements in its left sub-tree (9, 10, 18, 19, 21) aresmaller than 27, while all nodes in its right sub-tree (28, 29, and 36) are greater than the root node's value.

Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced. Whenever we search for an element, we do not need to traverse the entire tree. At every node, we get a hint regarding which sub-tree to search in. For example, in the given tree, if we have to search for 29, then we know that we have to scan only the leftsub-tree.If the valueis presentin the tree, it will only be in the left sub-tree, as 29 is smaller than 39 (the root node's value). The left sub-tree has a root node with the value 27. Since29 is greater than 27, wewill move to the right sub-tree, where wewill find the element. Thus, the average running time of a search operation is O(log2 n), as at every step, we eliminate half of the sub-tree from the search process. Due to its efficiency in searching elements, binary search trees are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

Createabinarysearchtreeusingthefollowing dataelements:

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

Solution:



ThebasicoperationsofaBinarySearchTree/Implementation

A. Search- Searchesanelementinatree.

B. Insert-Insertsanelementina tree.

C. Traversing-Processingofelements

Insertionoperation:

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node.

Addingavalueto BSTcanbedividedinto two stages:

• Searchforaplacetoputanewelement;

• Insertthenewelementtothisplace.

Theinsertionoperationisperformedasfollows...

Step1: CreateanewNodewithgivenvalueandsetitsleftandright toNULL.

Step2: CheckwhethertreeisEmpty.

Step 3: IfthetreeisEmpty,thensetset rootto newNode.

Step 4: Ifthe tree is Not Empty, thencheck whether value ofnewNode is smaller or larger thanthe node (here it is root node).

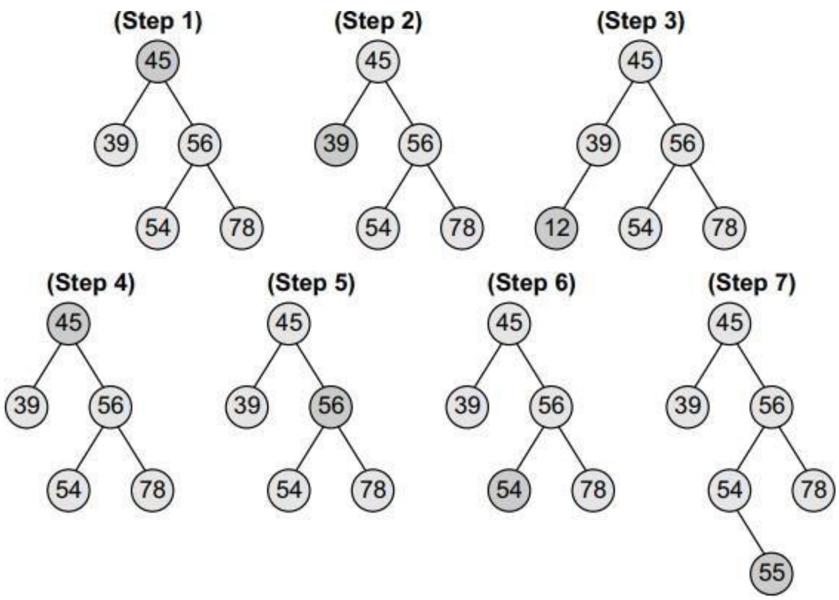
Step 5: Ifnew Node is smaller thanor equaltothe node, then move to its left child. IfnewNode is larger than the node, then move to its right child.

Step6: Repeattheabove stepuntilwereachtoaleafnode(e.i.,reachto NULL).

Step 7: After reaching a leaf node, then insert the new Node asleft child if new Nodeis smaller or equal to that leaf else insert it as right child.

ThenewnodewillalwaysreplaceaNULLreference.

Example:



Inserting nodes with values 12 and 55 in the given binary search tree

AlgorithmtoInsertagivenvalueinBinarySearchTree:

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

DeleteOperation:

Thedeletefunction deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process. We will take up three cases in this section and discuss how a node is deleted from a binarysearch tree.

Basically, it canbedividedintotwo stages:

1. Searchforanodeto remove.
2. If the node is found, remove that element. For that we will use simple recursion to find the node and delete it from the tree.

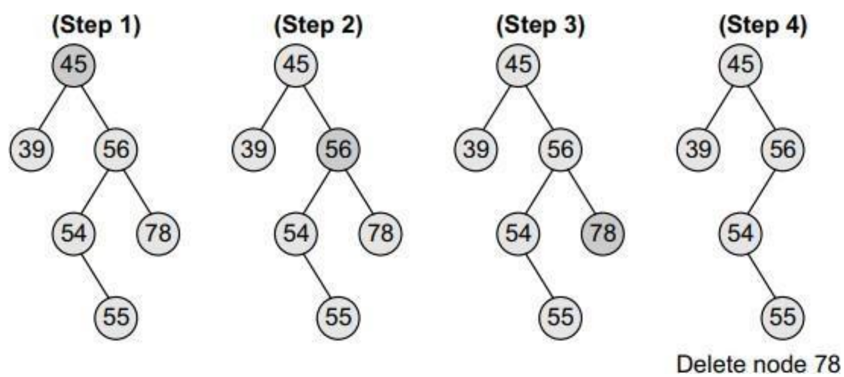
Thealgorithmhas3caseswhiledeletingnode:

1. Nodetobedeleted isaleaf node(nochildren).
2. Nodetobedeletedhasonechild.
3. Nodetobedeletedhastwochildren(leftand rightchildnodes).

Case1:

DeletingaNodethat hasNoChildren:

Look at the binary search tree. If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.

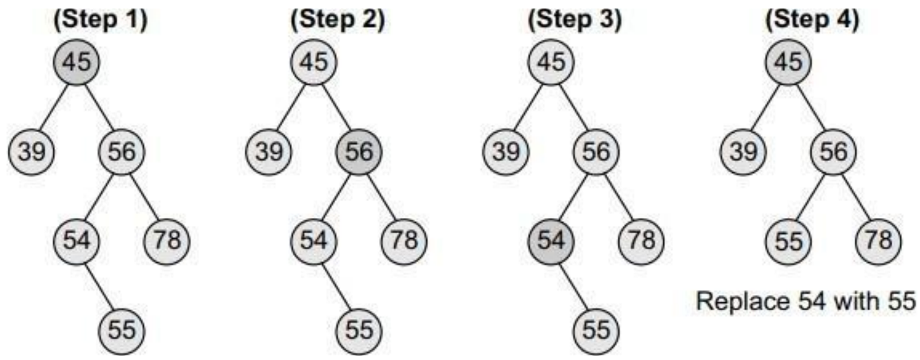


Deleting node 78 from the given binary search tree

Case2:

DeletingaNode withOneChild:

Tohandlethiscase,thenode’schildisset asthechildofthenode’sparent. Inother words,replacethe node with its child. Now, if the node is the left child of its parent, the node’s child becomes the left child ofthenode’sparent. Correspondingly, ifthenodeis theright child ofits parent, thenode’schild becomes the right child of the node’s parent. Look at the binary search tree shown below and see how deletion of node 54 is handled.

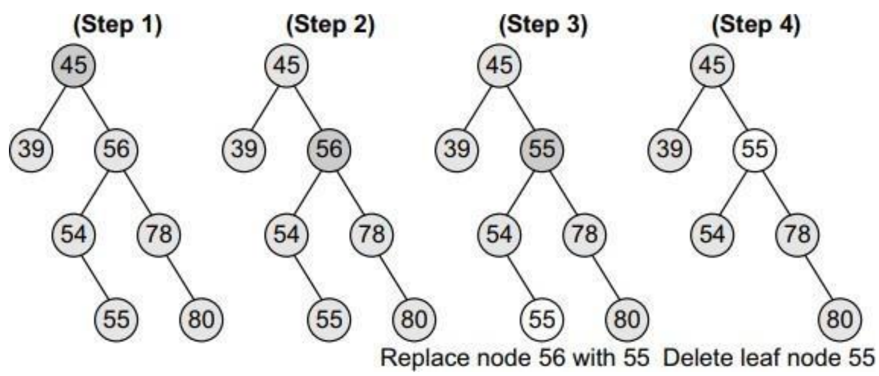


Deleting node 54 from the given binary search tree

Case3:

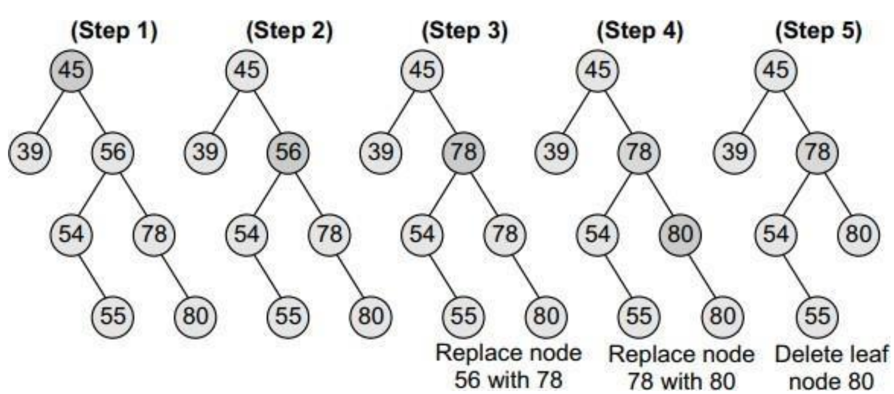
DeletingaNodewithTwoChildren:

Tohandlethis case, replacethenode'svaluewith its in-orderpredecessor (largest valuein theleft sub- tree) or in-order successor (smallest value in the right sub-tree). The in-order predecessor or the successor can then be deleted using any of the above cases. Look at the binary search tree given in below figure and see how deletion of node with value 56 is handled.



Deleting node 56 from the given binary search tree

This deletion could also be handled by replacing node 56 with itsin-order successor, as shown in below diagram.



Deleting node 56 from the given binary search tree

AlgorithmtoDeleteaNodefromBinarySearchTree:

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Thealgorithmtodeletea nodefromabinarysearch tree:

In Step 1 of the algorithm, we first check if TREE=NULL, because if it is true, then the node to be deleted is not present in the tree. However, if that is not the case, then we check if the value to be deleted is less than the current node's data. In case the value is less, we call the algorithm recursively on the node's left sub-tree, otherwise the algorithm is called recursively on the node's right sub-tree. Note that if we have found the node whose value is equal to VAL, then we check which case of deletion it is. If the node to be deleted has both left and right children, then we find the in-order predecessor of the node by calling findLargestNode(TREE -> LEFT) and replace the current node's value with that of its in-order predecessor. Then, we call Delete(TREE -> LEFT, TEMP -> DATA) to delete the initial node of the in-order predecessor. Thus, we reduce the case 3 of deletion into either case 1 or case 2 of deletion. If the nodetobedeleteddoesnothaveany child,thenwesimplysetthenodeto NULL.Lastbutnottheleast,if thenodeto bedeleted has eitheraleft oraright child but notboth, thenthecurrent nodeis replaced by its child node and the initial child node is deleted from the tree.

Searchfora Nodein aBinarySearchTree:

The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node. The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displayinganappropriatemessage.However,iftherearenodesinthetree,thenthesearchfunction checks to see if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node.

Thesearchoperation isperformedasfollows:

Step1: Readthesearchelementfromtheuser

Step2: Compare, thesearchelementwiththevalueofrootnodeinthetree.

Step 3: Ifbotharematching,thendisplay"Givennodefound!!!"andterminatethe function

Step4: If both are notmatching, then checkwhether search elementis smalleror larger than thatnode value.

Step 5: If search element is smaller, then continue the search process in left subtree.

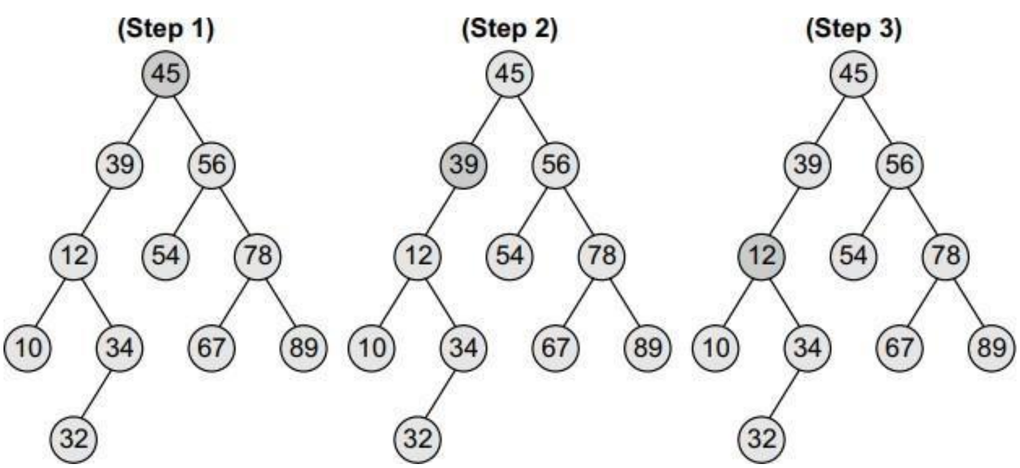
Step 6: If search element is larger, then continue the search process in right subtree.

Step7: Repeatthesameuntilwefoundexact elementorwecompletedwithaleafnode

Step8: Ifwereachtothenodewithsearchvalue,thendisplay"Elementisfound"andterminatethe function.

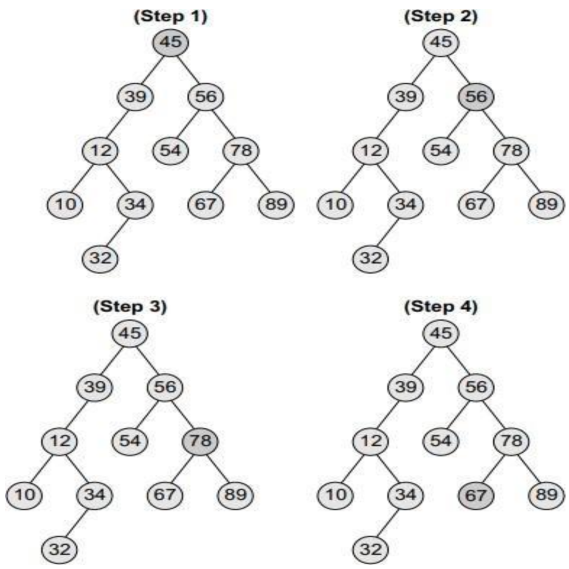
Step9: Ifwereachtoaleafnodeanditisalsonotmatching,thendisplay"Elementnotfound"and terminate the function.

Thebelowfigureshows howabinarytreeissearchedtofindaspecificelement.First,seehowthetree will be traversed to find the node with value 12.



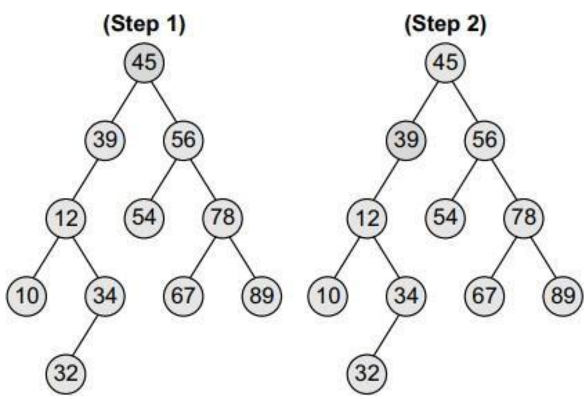
Searching a node with value 12 in the given binary search tree

Theproceduretofindthenodewithvalue67 is illustratedin belowfigure.



Searching a node with value 67 in the given binary search tree

The procedure tofind thenode with value 40isshown inbelowfigure.Thesearch would terminate after reaching node 39 as it does not have any right child.



Searching a node with the value 40 in the given binary search tree

AlgorithmtoSearchforanElementinBinarySearch Tree:

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

Thealgorithmtosearchforanelementinthebinarysearchtreeasshownbelow.InStep1,wecheckif the value stored at the current node of TREE is equal to VAL or if the current node is NULL, then we return the current node of TREE. Otherwise, if the value stored at the current node is less than VAL, then the algorithm is recursively called on its right sub-tree, else the algorithm is called on its left sub-tree.

AVLTREES:

AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962. The tree is namedAVLin honor of its inventors. In an AVL tree, the heights of the two sub-treesof a node may differ by at most one. Due to this property, the AVL tree is also known as a height- balanced tree. The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to O(log n).

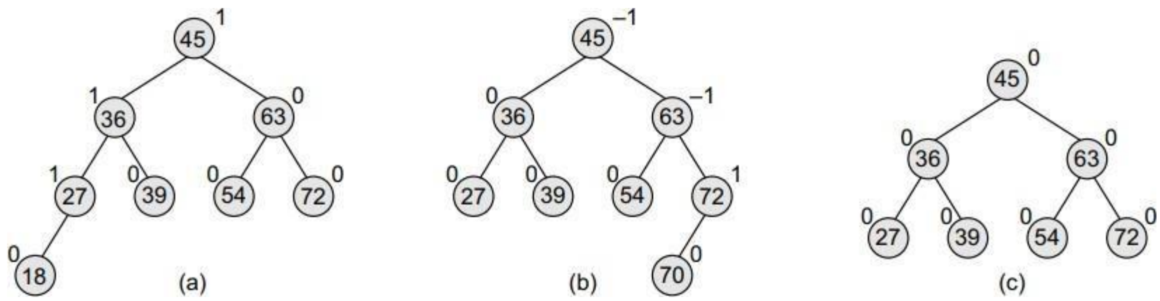
The structure of an AVL tree is the same as that of a binary search tree but with a little difference.In its structure, it stores an additional variable called the Balance Factor. Thus, every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

Balancefactor=Height(leftsub-tree)–Height(rightsub-tree).

- If balance factor ofany node is 1, itmeans that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0,itmeans that the left sub-tree and rightsub-tree contain equal height.
- If balance factor ofany node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

Look at the below figure. Note that the nodes 18, 39, 54, and 72 have no children, so their balance factor = 0. Node 27 has one left child and zero right child. So, the height of left sub-tree = 1, whereas the height ofright sub-tree=0.Thus, itsbalance factor =1. Lookat node36, it hasa left sub-treewithheight

= 2, whereas the heightof rightsub-tree = 1. Thus, its balancefactor = 2 – 1 = 1. Similarly, the balance factorofnode 45 = 3 – 2 =1; and node 63 has a balance factorof 0 (1 – 1).



(a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

The trees given in above figure are typical candidates of AVL trees because the balancing factor of every node is either 1, 0, or –1. However, insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, rebalancing of the tree may have to be done. The tree is rebalanced by performing rotation at the critical node.

There are four types of rotations: LL rotation, RR rotation, LR rotation, and RL rotation. The type of rotation that has to be done will vary depending on the particular situation.

OperationsonAVLtree:

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

1. Insertion 2.Deletion

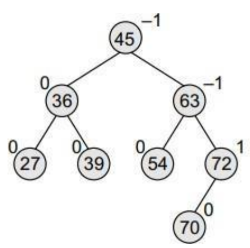
InsertingaNewNodeinanAVLTree:

Insertion in an AVL tree is also done in the same way as it is done in a binary search tree. In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree. However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still –1, 0, or 1, then rotations are not required. During insertion, the new nodeisinsertedastheleafnode,soitwillalwayshaveabalancefactorequaltozero.Theonly nodes whose balance factors willchange are those which lie in the pathbetweenthe rootofthe tree and the newly inserted node.

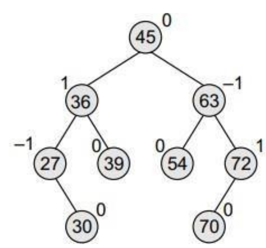
Thepossiblechangeswhich maytakeplaceinanynodeonthepathareas follows:

- Initially,thenodewaseitherleft-orright-heavyandafterinsertion,itbecomesbalanced.
- Initially,thenodewasbalancedandafterinsertion,itbecomeseitherleft-orright-heavy.
- Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a critical node.

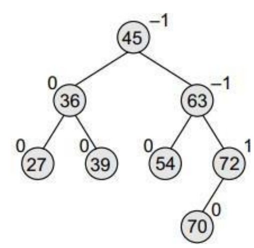
Consider the AVL tree given in below figure. If we insert a new node with the value 30, then the new tree will still be balanced and no rotations will be required in this case. Look at the tree which shows the tree after inserting node 30.



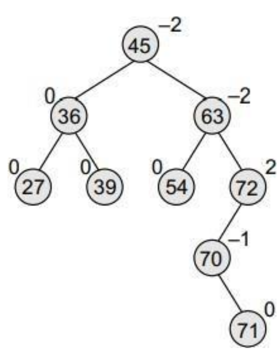
AVL tree



AVL tree after inserting a node with the value 30



AVL tree



AVL tree after inserting a node with the value 71

Let us take another example to see how insertion can disturb the balance factors of the nodes and how rotations are done to restore the AVL property of a tree. After inserting a new node with the value 71, the new tree will be as shown in the above figure. Note that there are threenodesinthetreethathavetheirbalancefactors2,–2,and–2,therebydisturbingthe

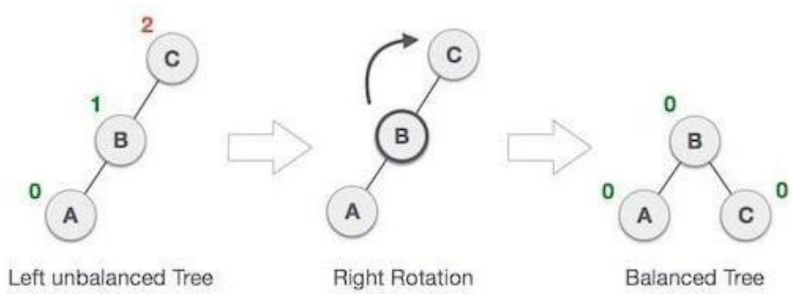
AVLness of the tree. So, here comes the need to perform rotation. To perform rotation, our first task is to find the critical node. Critical node is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither –1, 0, nor 1. In the tree given above, the critical node is 72. The second task in rebalancing the tree is to determine which type of rotation has to be done. There are four types of rebalancing rotations and application of these rotations depends on the position of the inserted node with reference to the critical node.

The fourcategoriesofrotationsare:

- **LLrotation:**Thenewnodeisinsertedintheleftsub-treeoftheleftsub-treeofthe critical node.
- **RRrotation:**Thenewnodeisinsertedin therightsub-tree of therightsub-tree of the critical node.
- **LRrotation:**Thenewnodeisinsertedintherightsub-treeoftheleftsub-treeofthe critical node.
- **RLrotation:**Thenewnodeisinsertedintheleftsub-treeoftherightsub-treeofthe critical node.

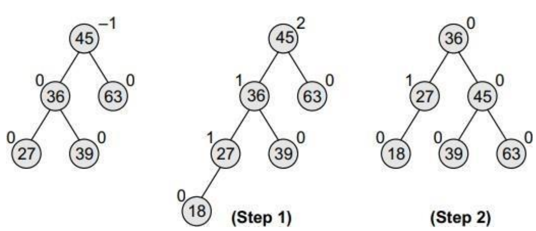
LL Rotation:

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



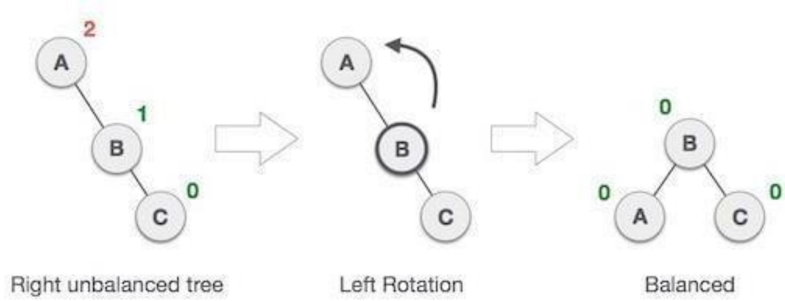
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

Example: ConsidertheAVLTreeandinsert 18 intoit.



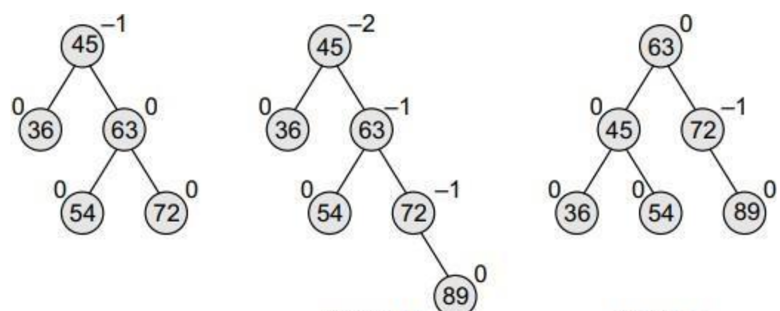
RR Rotation

When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2.



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

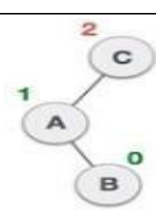
Example: ConsidertheAVLTreeandinsert 89 intoit.

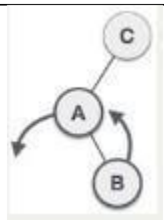
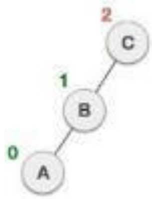
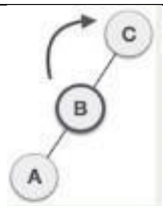
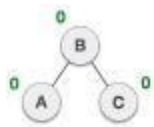


LRRotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LLrotationisperformedonfulltree,byfulltreewemeanthefirstnodefromthepathofinserted node whose balance factor is other than -1, 0, or 1.

Letusunderstandeachandeverystepveryclearly:

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node isin the right subtree of left subtree of C

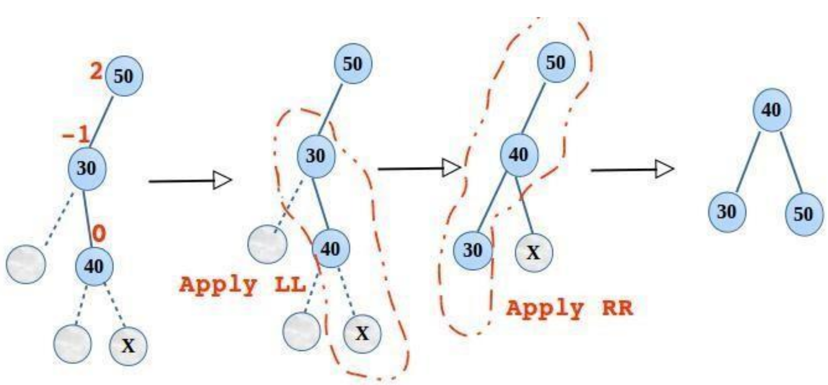
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first.By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**.
	Now we perform LL clockwise rotation on full tree, i.e. on node **C**. node C has now become the right subtree of node B, A is leftsubtree of B.
	Balancefactorofeachnodeisnoweither-1,0,or1,i.e.BSTis balanced now.

Example:

Shownbelow isthecase ofLR rotation, heretworotationsare performed. First RR andthen, LL as follows,

- Rightrotationisappliedat70,afterrestructuring,60takestheplaceof70and70asthe right child of 60.
- Now left rotation is required at the root 50, 60 becomes the root. 50 and 70 become the left and right child respectively.

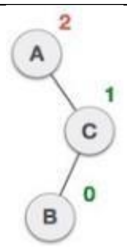
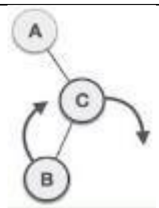
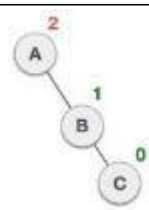
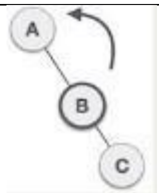
LRRotation

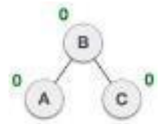


Wecouldalso thinkofthe shownwaytobalance quicklyrather thangoingwithtwo rotations.

RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree wemean the first node from the path of inserted node whose balance factor is other than -1, 0,or 1.

State	Action
	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doint RR rotation, node**C** has become the right subtree of **B**.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation(anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.

	Balancefactorofeachnodeisnoweither-1,0,or 1,i.e.,BSTis balanced now.

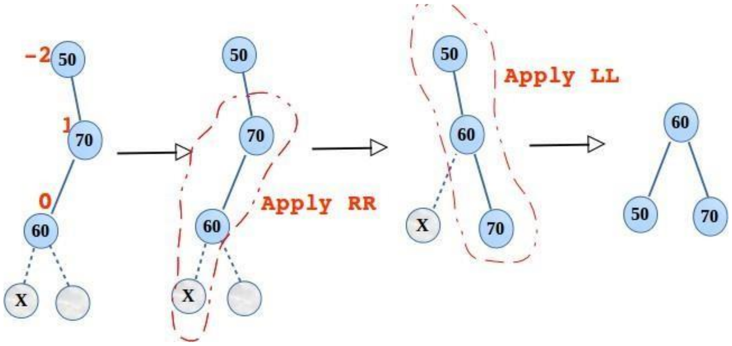
Example:

Shownbelow isthe case ofRLrotation, heretworotationsare performed. First LLand then, RR as follows,

- Leftrotationisappliedat30,afterrestructuring40takestheplaceof30and30asthe left child of 40.
- Now rightrotation is required at the root 50, 40 becomes root.30 and 50becomes the left and right child respectively.

RLRotation

Wecouldalso thinkofthe shownwaytobalance quicklyrather thangoingwithtwo rotations.



Example:

Construct AVLTreeforthefollowingsequenceofnumbers:

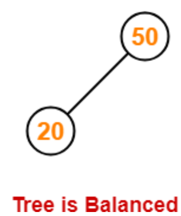
50,20, 60,10, 8,15,32, 46,11,48

Solution:

Step-01: Insert50

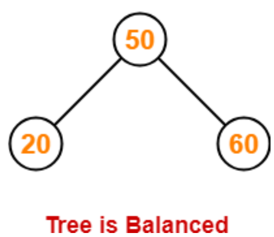
Step-02: Insert20

- As20 <50,so insert20in50'sleftsubtree.



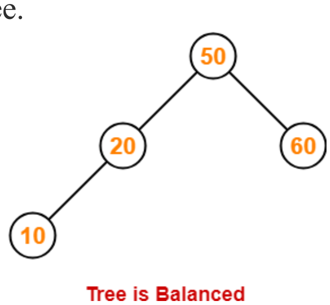
Step-03: Insert60

- As60>50,so insert60in50'srightsubtree.



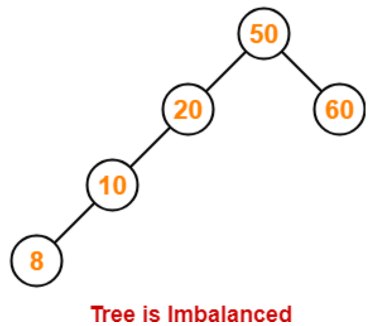
Step-04: Insert10

- As10<50,soinsert 10in50'sleft subtree.
- As10<20,soinsert10in20'sleftsubtree.



Step-05: Insert8

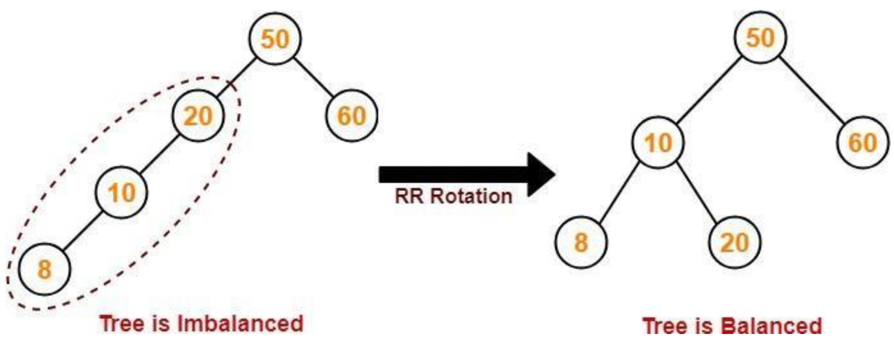
- As8<50,soinsert 8in50'sleft subtree.
- As8<20,soinsert 8in20'sleft subtree.
- As8<10,soinsert8in10'sleftsubtree.



Tobalancethetree,

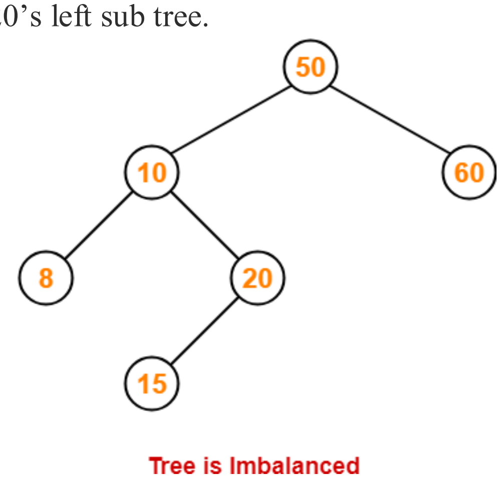
- Findthefirst imbalanced nodeonthepathfromthenewlyinsertednode(node8)totheroot node.
- Thefirst imbalancednode isnode20.
- Now,countthreenodesfromnode20inthedirectionofleafnode.
- Then, use AVL tree rotation to balance the tree.

Followingthis,wehave-



Step-06: Insert15

- As 15 < 50, so insert 15 in 50's left sub tree.
- As15>10,soinsert 15in10'sright subtree.
- As 15 < 20, so insert 15 in 20's left sub tree.

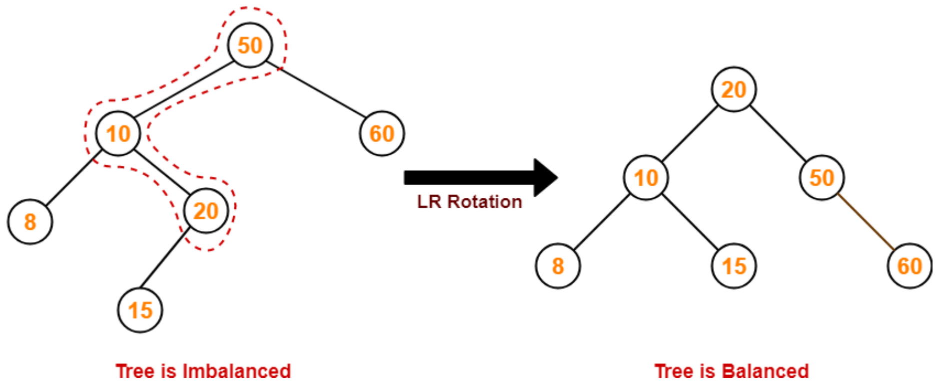


Tobalancethetree,

- Findthefirst imbalanced nodeonthepathfromthenewlyinsertednode(node15)totheroot node.
- Thefirst imbalancednode isnode50.
- Now,countthreenodesfromnode50inthedirectionofleafnode.

Then, use AVL tree rotation to balance the tree.

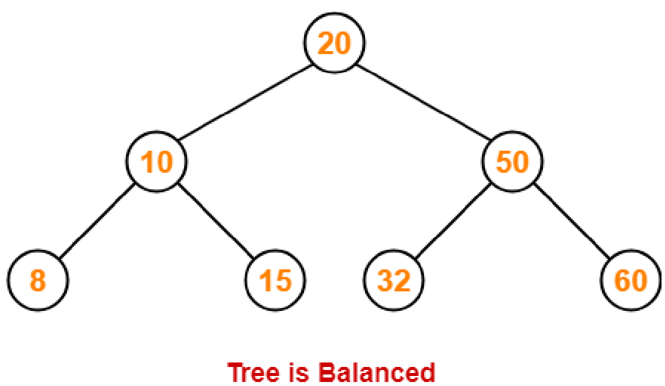
Followingthis,wehave-



Step-07: Insert32

- As32>20,soinsert 32in20'sright subtree.

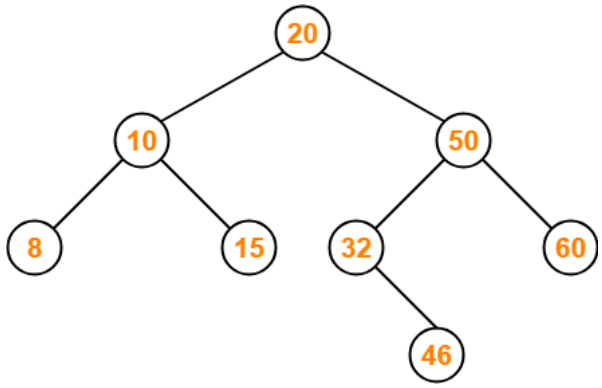
As 32 < 50, so insert 32 in 50's left sub tree.



Step-08: Insert46

- As46>20,soinsert 46in20'sright subtree.

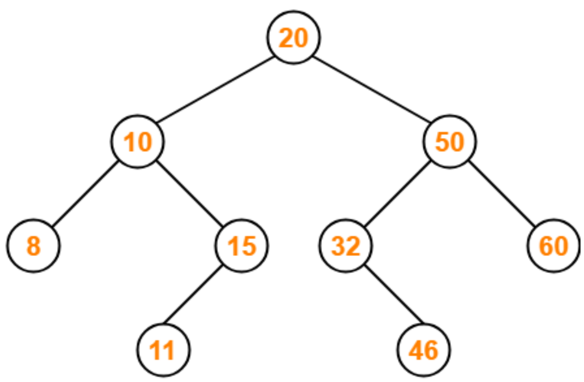
As 46 < 50, so insert 46 in 50's left sub tree.
As46>32,soinsert 46in32'sright subtree.



Tree is Balanced

Step-09: Insert11

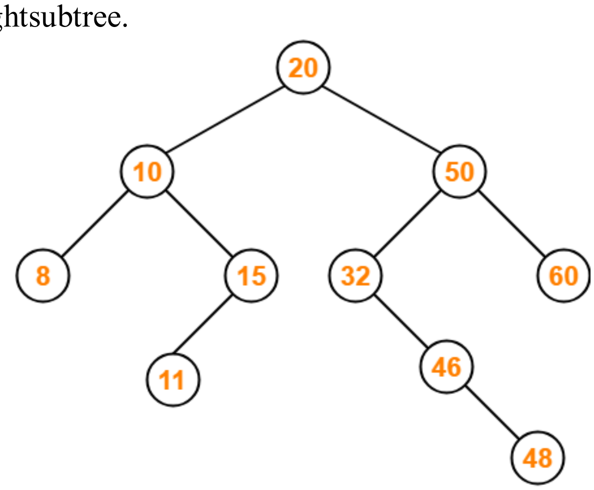
- As 11 < 20, so insert 11 in 20’s left sub tree.
- As11>10,soinsert 11in10’sright subtree.
As 11 < 15, so insert 11 in 15’s left sub tree.



Tree is Balanced

Step-10: Insert48

- As48>20,soinsert 48in20’sright subtree.
As 48 < 50, so insert 48 in 50’s left sub tree.
As48>32,soinsert 48in32’sright subtree. As48
>46,so insert48in46’srightsubtree.



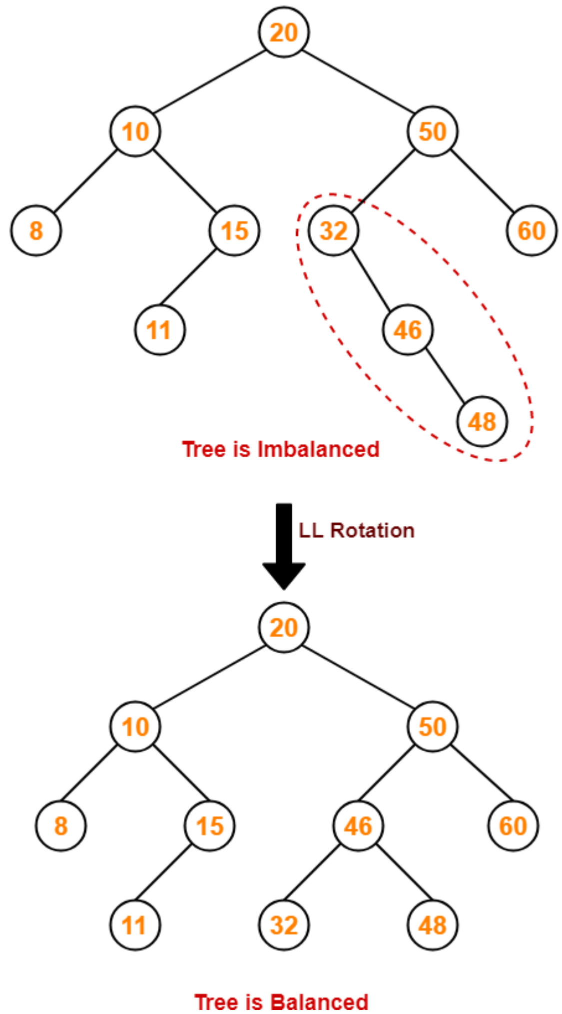
Tree is imbalanced

Tobalancethetree,

- Findthefirst imbalanced nodeonthepathfromthenewlyinsertednode(node48)totheroot node.
- Thefirst imbalancednode isnode32.
- Now,countthreenodesfromnode32inthedirectionofleafnode.

Then, use AVL tree rotation to balance the tree.

Followingthis,wehave-



DeletingaNodefromanAVL Tree:

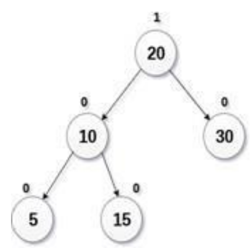
Deletion of a node in an AVL tree is similar to that of binary search trees. But it goes one step ahead. Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree, we need to perform rotations. There are two classes of rotations that can be performed on an AVL tree after deleting a given node. These rotations are R rotation and L rotation. On deletion of node X from the AVL tree, if node A becomes the critical node (closestancestor node on the path from X to the root node that does not have its balance factor as 1, 0, or –1), then the type of rotation depends on whether X is in the left sub-tree of A or in its right sub-tree.

If the node to be deleted is present in the left sub-tree of A, then L rotation is applied, else if X is in the right sub-tree, R rotation is performed. Further, there are three categories of L and R rotations. The variations of L rotation are L–1, L0, and L1 rotation. Correspondingly for R rotation, there are R0, R–1, and R1 rotations. In this section, we will discuss only R rotation. L rotations are the mirror images of R rotations.

R0rotation(NodeBhasbalancefactor0)**:**

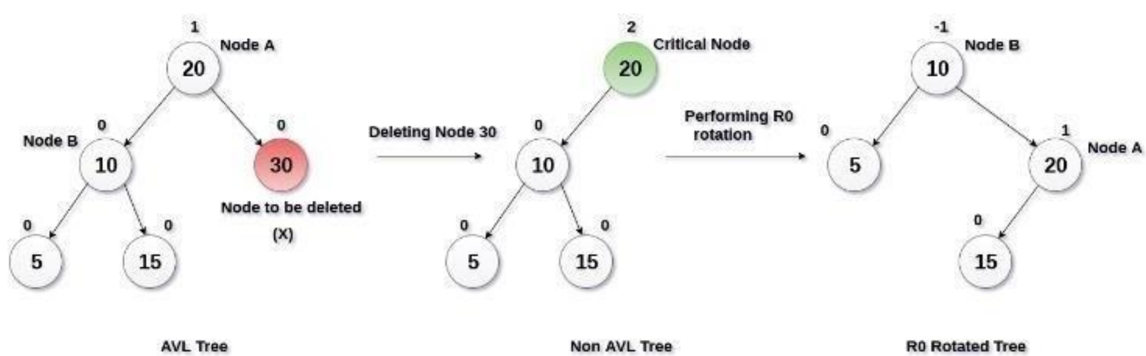
Example:

Deletethenode30fromtheAVLtreeshowninthefollowingimage.



Solution:

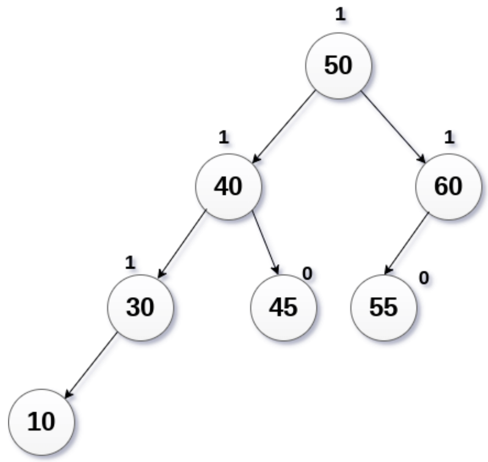
Inthis case,thenodeBhasbalancefactor0,thereforethetreewillberotatedbyusingR0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



R1Rotation(NodeBhasbalancefactor1):

Example:

DeleteNode55fromtheAVLtreeshowninthefollowingimage.

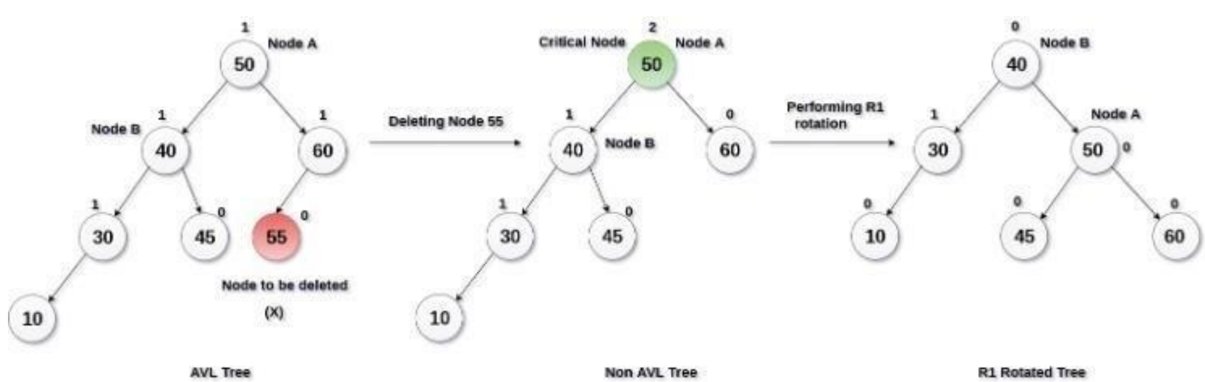


AVL Tree

Solution:

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45).

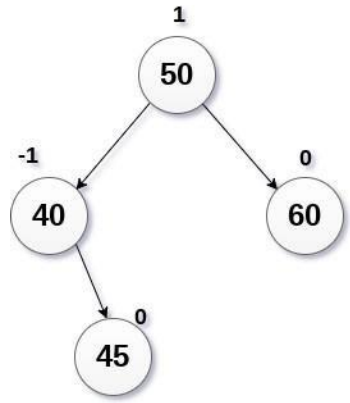
Theprocess involvedinthesolutionisshowninthefollowing image.



R-1Rotation(NodeBhasbalancefactor-1):

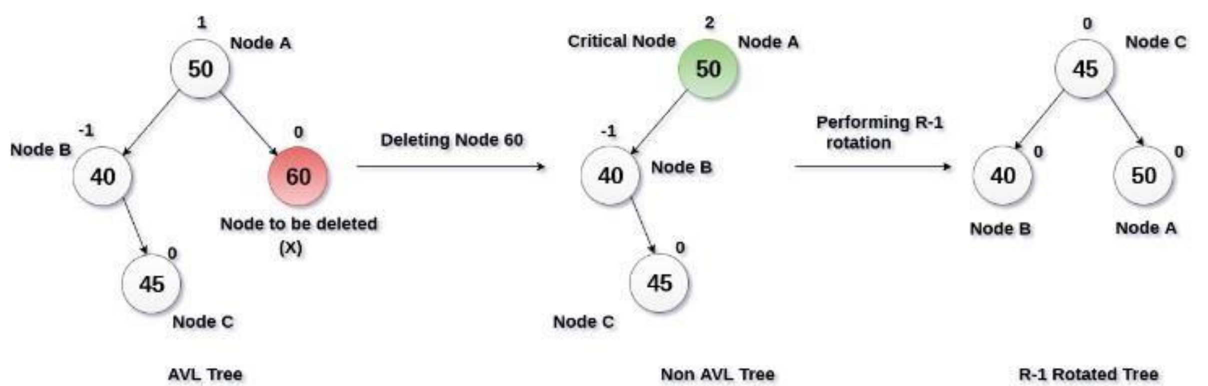
Example:

DeletethenodeỌ60fromtheAVLtreeshowninthefollowingimage.



Solution:

In this case, nodeB has balance factor -1. Deleting thenode60, disturbsthebalancefactorofthe node50 therefore, it needs to beR-1 rotated.ThenodeC i.e.45 becomestherootofthetreewith the node B(40) and A(50) as its left and right child.



BTREES:

A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access. A B tree of order m can have a maximum of m–1 keys andm pointers to its sub-trees. A B tree may contain a large number of key values and pointersto sub-trees. Storing a large number of keys in a single node keeps the heightof the tree relatively small.

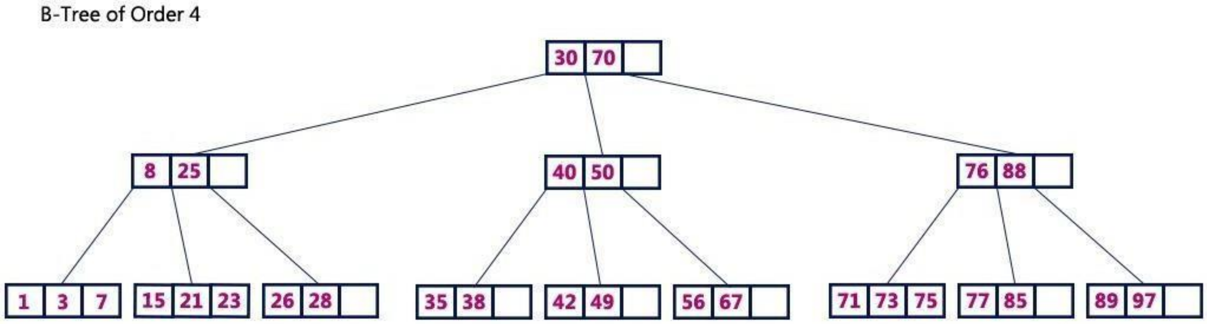
Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is keptlow byputting the maximumpossible keysina B-Treenode.Generally,the B-Treenode sizeis kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, etc.

PropertiesofBTrees:

1. EverynodeintheBtreehasatmost(maximum)mchildren.
2. Every node in the B tree except the root node and leaf nodes has at least (minimum) m/2 children. This condition helps to keep the tree bushy so that the path from the rootnode to the leaf is very short, even in a tree that stores a lot of data.
3. Therootnodehasatleasttwochildrenifitis notaterminal(leaf)node.
4. Allleafnodesareatthesamelevel.

Forexample:

B-TreeofOrder4contains a maximumof3 keyvalues ina nodeand maximumof4 childrenfor a node.



Whileperforminginsertionanddeletionoperationsina Btree,the number ofchildnodes may change. So, in order to maintain a minimum number of children, the internal nodes may be joined or split.

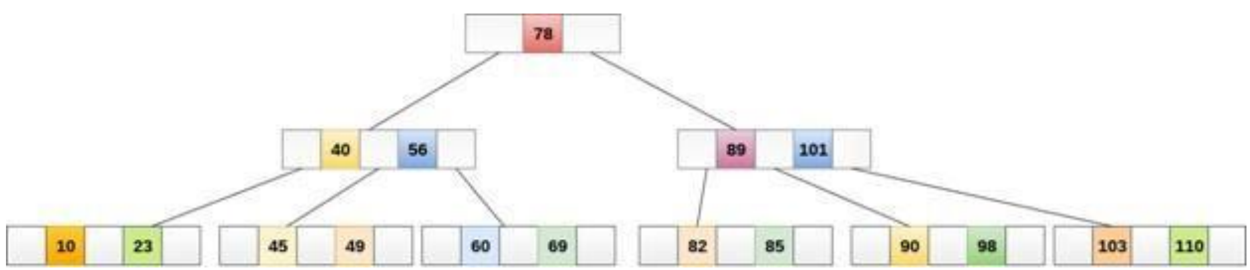
Wewilldiscusssearch,insertion,and deletionoperations:

Searching:

SearchinginBTreesissimilartothatinBinarysearchtree.Forexample,ifwesearchforan item 49 in the following B Tree. The process will something like following :

1. Compareitem49 withrootnode78. since49 <78hence, movetoitsleftsub-tree.
2. Since,40<49<56,traverserightsub-treeof40.
3. 49>45,movetoright.Compare49.
4. matchfound,return.

Searching inaBtreedependsupontheheight ofthetree.ThesearchalgorithmtakesO(logn) time to search any element in a B tree.



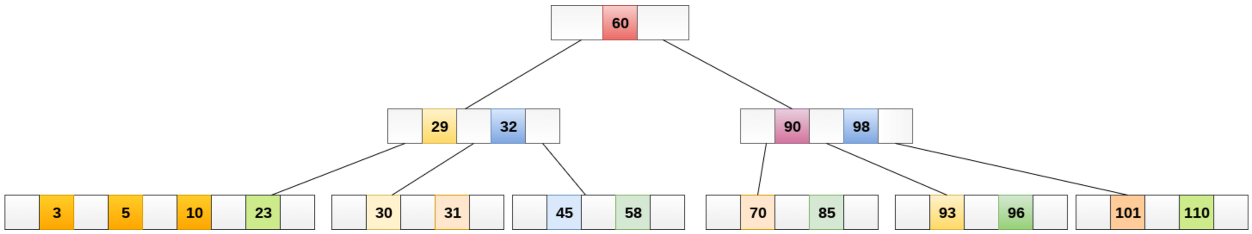
Inserting

Insertions are done atthe leaf node level. The following algorithm needs to be followed inorder to insert an item into B Tree.

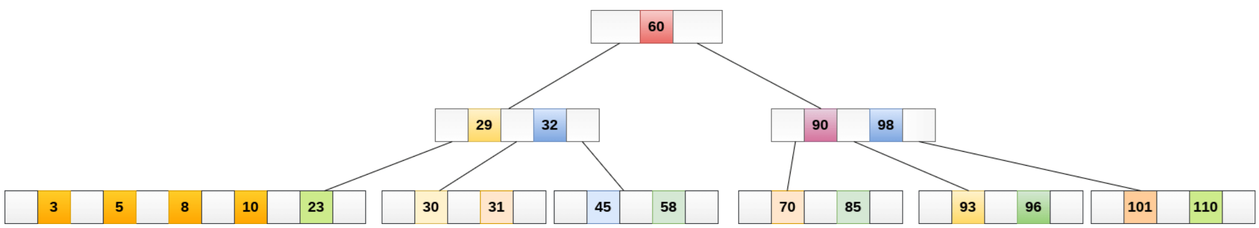
1. Traverse the B Treein order tofind the appropriate leafnode at which thenode can beinserted.
2. Iftheleafnodecontainlessthanm-1 keystheninserttheelementintheincreasingorder.
3. Else,iftheleafnodecontains m-1keys, thenfollowthefollowingsteps.
   - Insertthenewelementintheincreasingorderofelements.
   - Splitthenodeintothetwonodesatthemedian.
   - Pushthemedianelementuptoitsparentnode.
   - Ifthe parent node also contain m-1 number ofkeys, then split it too by following the same steps.

Example:

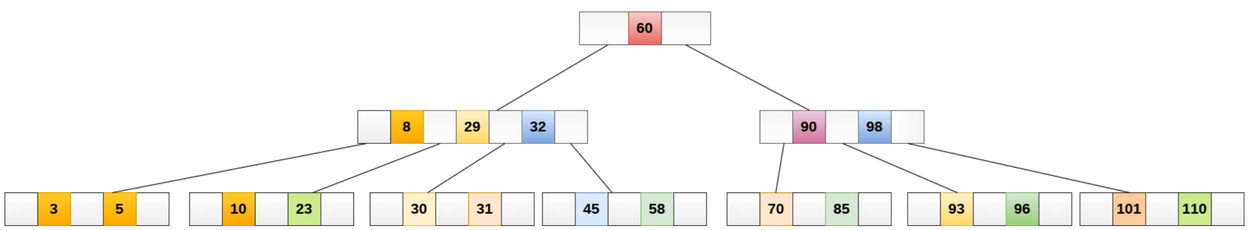
Insertthenode8 intotheBTreeoforder5showninthefollowingimage.



8willbeinsertedtotherightof5,thereforeinsert 8.



Thenode,nowcontain5keyswhichisgreaterthan(5 -1=4)keys.Thereforesplit thenode fromthe median i.e. 8 and push it up to its parent node shown as follows.



Deletion:

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followedin order to delete a node from a B tree.

1. Locatetheleafnode.
2. Ifthere aremorethanm/2 keysintheleafnodethendeletethedesiredkeyfromthenode.
3. If theleaf nodedoesn'tcontainm/2keys then complete thekeysby takingthe element from eight or left sibling.

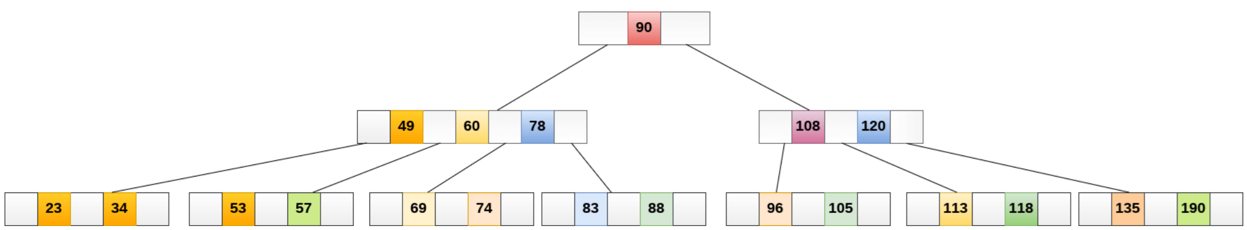
- Iftheleft siblingcontainsmorethanm/2elementsthenpushitslargest elementup to its parent and move the intervening element down to the node where the key is deleted.
- Iftherightsiblingcontainsmorethanm/2elementsthenpushitssmallestelement up to the parent and move intervening element down to the node where the key is deleted.

4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. Ifparentisleftwith lessthanm/2nodesthen, applytheaboveprocessontheparent too.

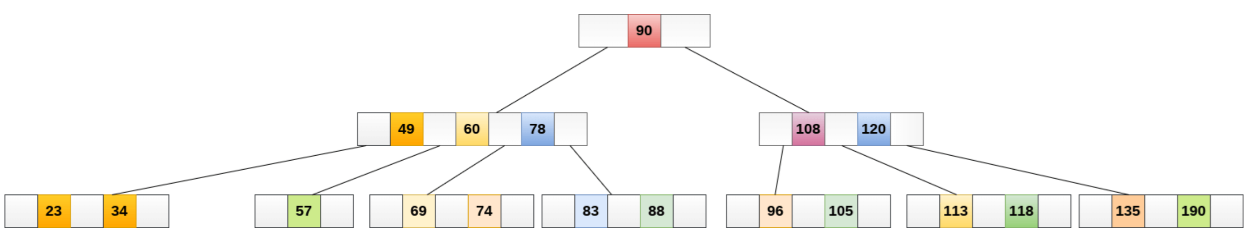
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example1

Deletethenode53fromtheBTreeoforder5showninthefollowingfigure.

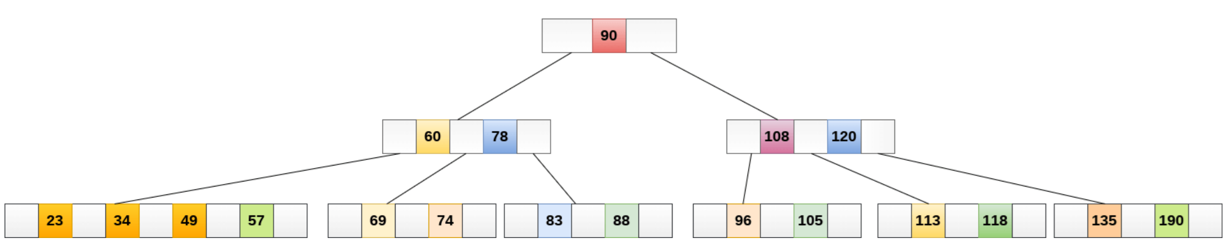


53ispresentintherightchildofelement49.Deleteit.



Now, 57 is the only element which is leftin the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree arealso not sufficient therefore, mergeit with theleft sibling and intervening element of parent i.e. 49.

ThefinalBtreeisshownas follows.



Example

Constructa**B-TreeofOrder3**byinserting numbersfrom1to 10.

insert(1)

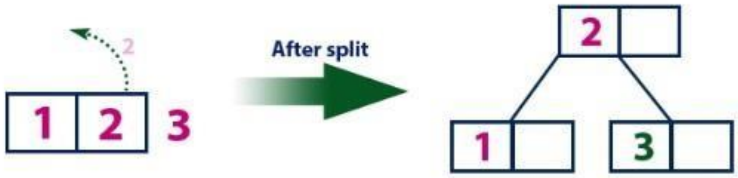
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

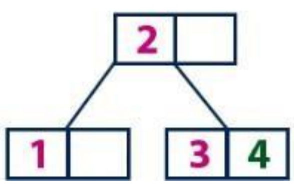
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

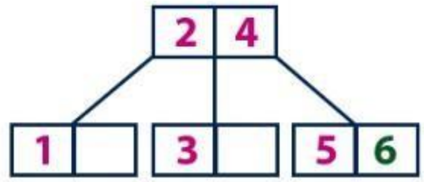


insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

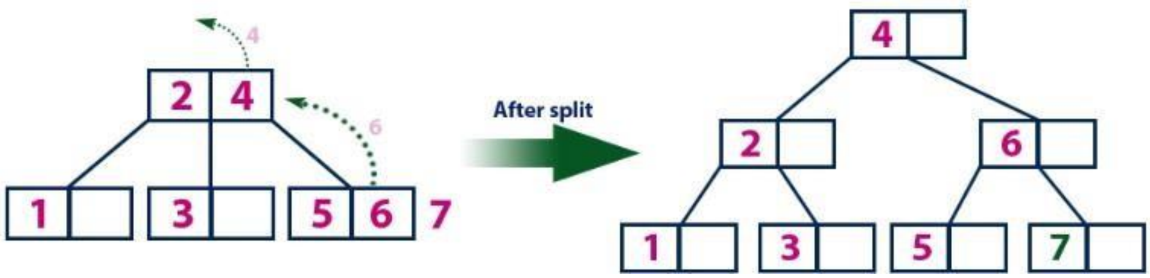
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



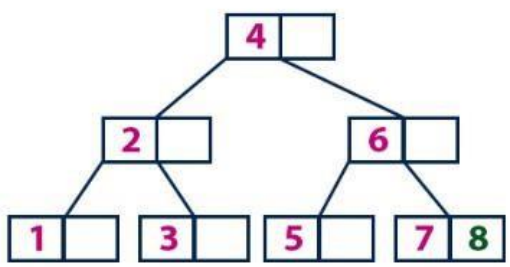
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



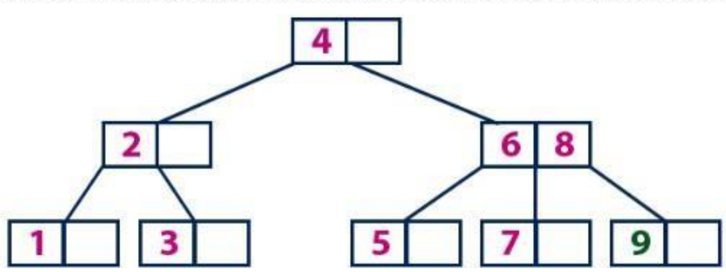
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



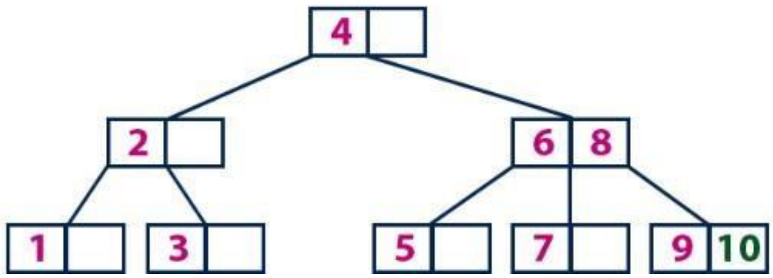
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



B+TREES

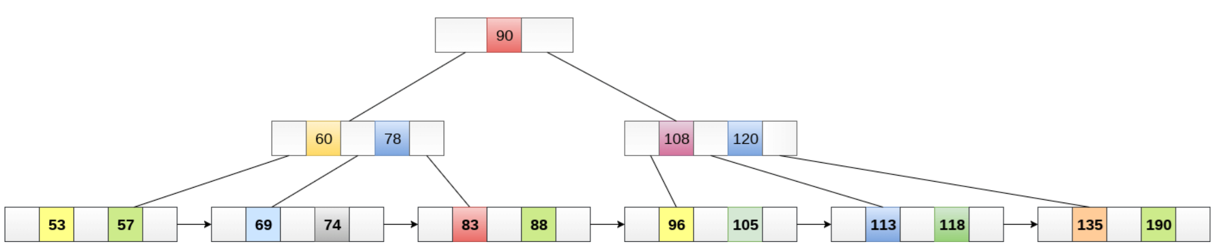
A B+ tree is a variant of a B tree which stores sorted data in a way thatallows for efficientinsertion,retrieval,andremovalofrecords,eachofwhichisidentifiedbyakey.Whilea B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; onlykeys are stored in the interior nodes.

The leaf nodes of a B+ tree are often linked to one another in a linked list. This has an added advantage of making the queries simpler and more efficient.

Typically, B+ trees are used to store large amounts of data that cannot be stored in the main memory. With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory.

B+ trees store data only in the leaf nodes. All other nodes (internal nodes)are called index nodes or i-nodes and store index values. This allows us to traverse the tree from the root down to the leaf node that stores the desired data item.

The internal nodes of B+ tree are often called index nodes. A B+ tree oforder 3 is shown in the following figure.



Many database systems are implemented using B+ tree structure because of its simplicity. Since all the data appear in the leaf nodes and are ordered, the tree is always balanced and makes searching for data efficient.

AdvantagesofB+Tree

1. Recordscanbefetchedinequalnumber ofdiskaccesses.
2. HeightofthetreeremainsbalancedandlessascomparetoBtree.
3. Wecanaccessthedatastored inaB+treesequentiallyaswellasdirectly.
4. Keysareused forindexing.
5. Fastersearchqueries asthedataisstoredonlyontheleafnodes.

ComparisonBetweenBTreesandB+Trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

InsertioninB+Tree:

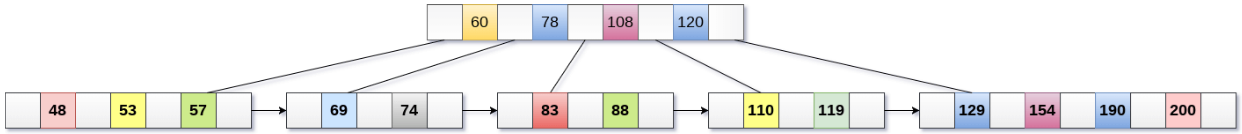
Step1: Insertthenewnodeasaleaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

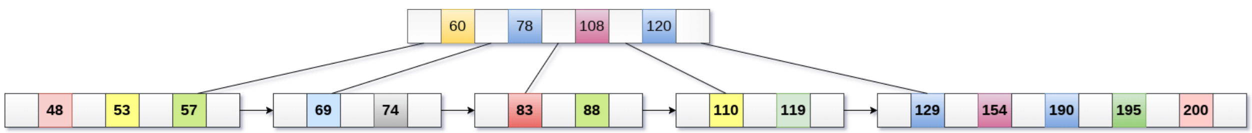
Step 3: Iftheindexnodedoesn'thaverequiredspace,splitthenodeandcopythemiddleelement to the next index page.

Example:

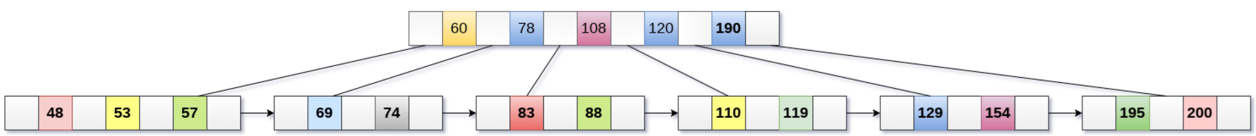
Insertthevalue195intotheB+treeoforder5showninthefollowingfigure.



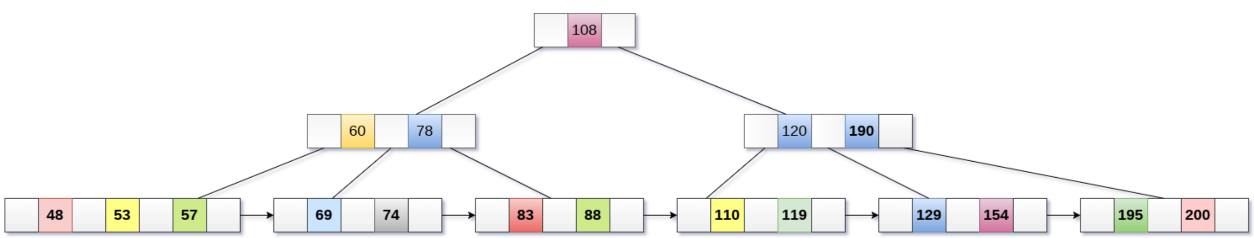
195will be insertedintherightsub-treeof120after 190.Insertitatthedesired position.



Thenodecontainsgreaterthanthemaximumnumberofelementsi.e.4,thereforesplititand place the median node up to the parent.



Now,theindexnodecontains6childrenand5keyswhichviolatestheB+treeproperties, therefore we need to split it, shown as follows.



DeletioninB+Tree:

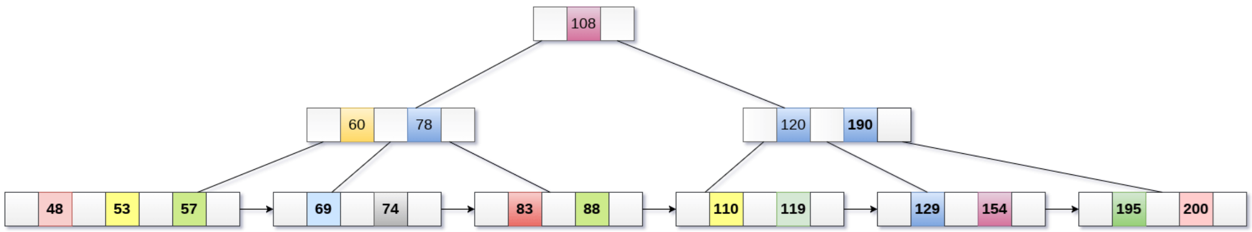
Step1: Deletethekeyanddatafromtheleaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

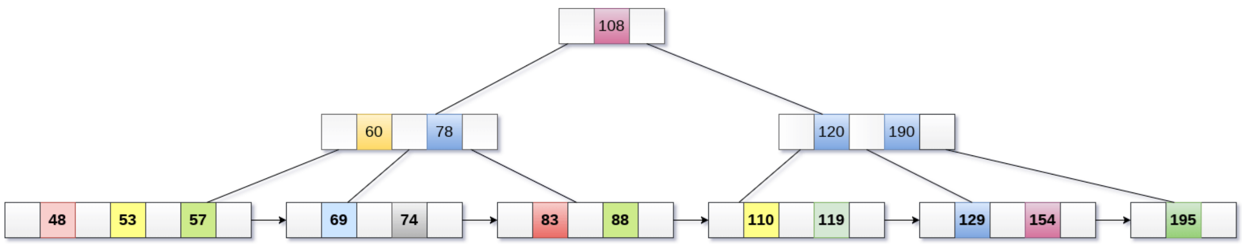
Step 3:if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

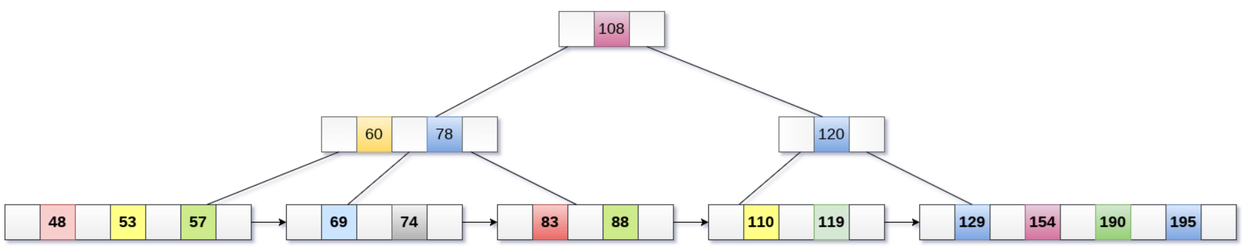
Deletethekey200 fromtheB+Treeshowninthefollowingfigure.



200ispresentintherightsub-treeof190,after195.deleteit.

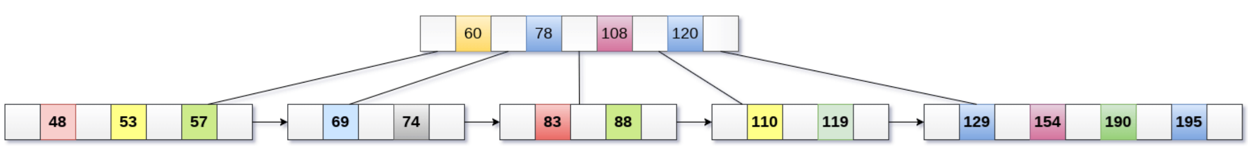


Mergethetwo nodes byusing 195,190,154 and129.



Now,element120isthesingleelementpresentinthenodewhichisviolatingtheB+Tree properties. Therefore, we need to merge it byusing 60, 78, 108 and 120.

Now,the height ofB+treewillbedecreasedby1.



RED–BLACK TREE

Red-Black Trees are another type of the Balanced Binary Search Trees with two colourednodes: Red and Black. It is a self-balancing binary search tree thatmakes use of these colours to maintain the balance factor during the insertion and deletion operations. Hence, during the Red-Black Tree operations, the memoryuses 1 bit of storage to accommodate the colour information of each node

InRed-Blacktrees,alsoknownasRBtrees,therearedifferentconditionstofollowwhileassigningthecolourstothenodes.

- Therootnodeisalwaysblackin colour.
- Notwoadjacentnodesmustberedincolour.
- Everypathinthetree(fromtherootnodetotheleafnode)musthavethesameamountofblackcolourednodes.

Even though AVL trees are more balanced than RB trees, with the balancing algorithm in AVL trees being stricter than that of RB trees, multiple and fasterinsertion and deletion operations are made more efficient through RB trees.

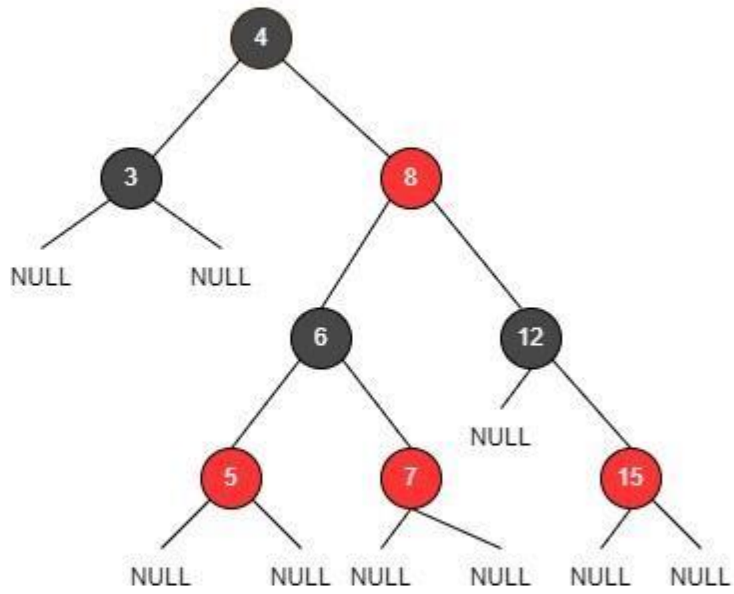


Fig:RBtrees

BasicOperationsofRed-BlackTrees

TheoperationsonRed-BlackTreesincludeallthebasicoperationsusuallyperformedonaBinarySearchTree.SomeofthebasicoperationsofanRBTreeinclude −

- Insertion
- Deletion
- Search

InsertionoperationofaRed-Blacktreefollowsthesameinsertionalgorithmofabinarysearchtree.Theelementsareinsertedfollowingthebinarysearchproperty and as an addition, the nodes are color coded as red and black to balance the tree according to the red-black tree properties.

Followtheproceduregivenbelowtoinsertanelementintoared-blacktreebymaintainingbothbinarysearchtreeandredblacktreeproperties.

Case1−Checkwhetherthetreeisempty;makethecurrentnodeastherootandcolorthenodeblackifitisempty.

Case2−Butifthetreeisnotempty,wecreateanewnodeandcoloritred.Herewefacetwodifferentcases−

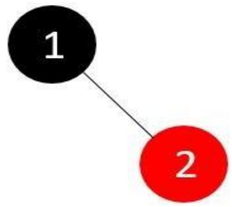
- Iftheparentofthenewnodeisablackcolorednode,weexittheoperationandtreeisleftasitis.
- Iftheparentofthisnewnodeisredandthecoloroftheparent'ssiblingiseitherblackorifitdoesnotexist,weapplyasuitablerotationandrecoloraccordingly.
- If the parent of thisnew node is red and color of the parent's sibling is red, recolor the parent, thesibling and grandparent nodes to black. The grandparentis recolored only if it is **not** the root node; if it is the root node recolor only the parent and the sibling.

InsertionExample

LetusconstructanRBTreeforthefirst7integernumberstounderstandtheinsertionoperationindetail−The tree is

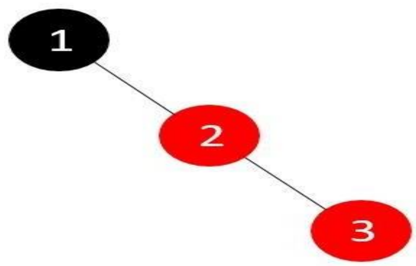
checked to be empty so the first node added is a root and is colored black.

Now,thetreeisnotemptysowecreateanewnodeandaddthenextintegerwithcolorred,

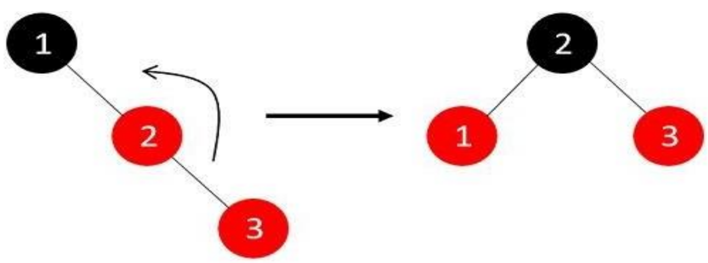


ThenodesdonotviolatethebinarysearchtreeandRBtreeproperties,hencewemoveaheadtoaddanothernode.

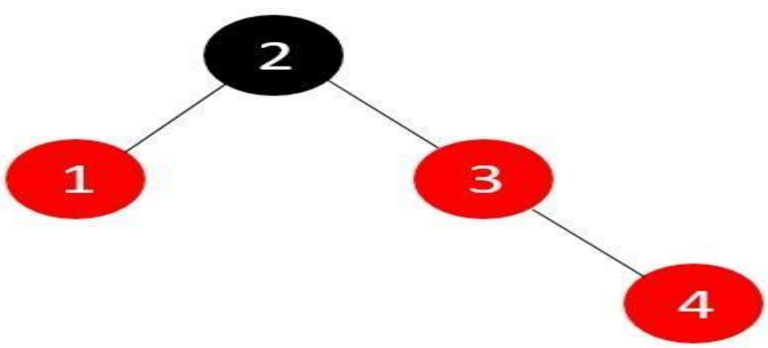
Thetreeisnotempty;wecreateanewrednodewiththenextintegertoit.Buttheparentofthenewnodeisnotablackcolorednode,



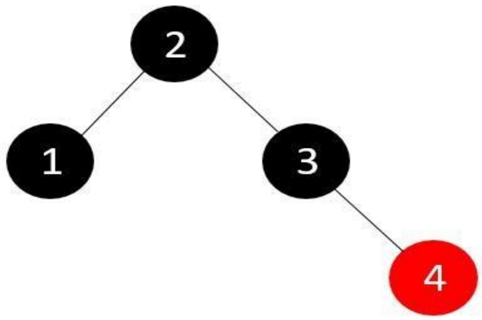
ThetreerightnowviolatesboththebinarysearchtreeandRBtreeproperties;sinceparent'ssiblingisNULL,weapplyasuitablerotationandrecolorthenodes.



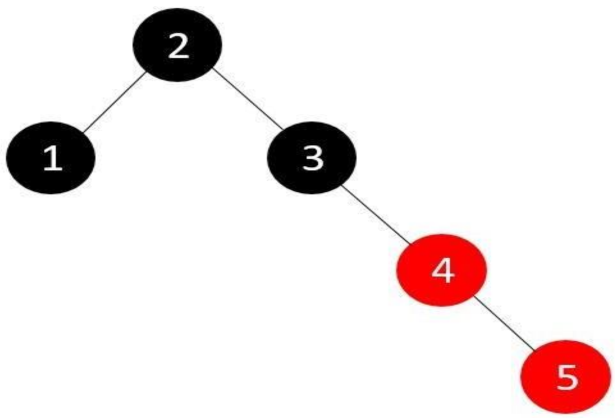
NowthattheRBTreepropertyisrestored,weaddanothernodetothetree−



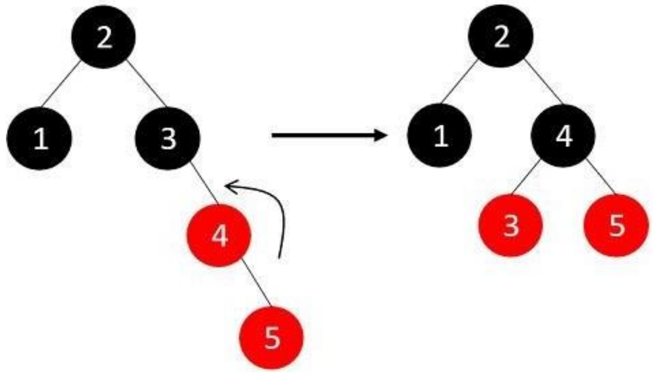
The tree once again violates the RB Tree balance property, so we check for the parent's sibling node color, red in this case,so we just recolor the parent and thesibling.



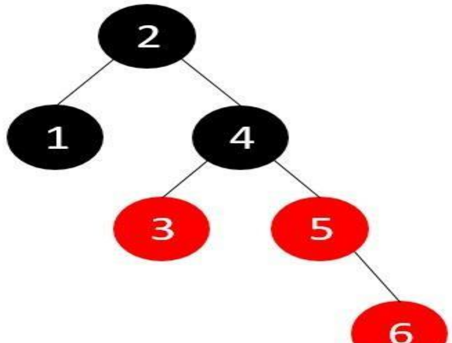
Wenextinserttheelement5,whichmakesthetreeviolatetheRBTreebalancepropertyonceagain.



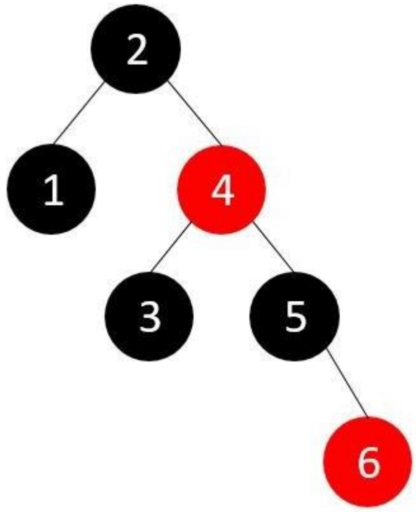
AndsincethesiblingisNULL,weapplysuitablerotationandrecolor.



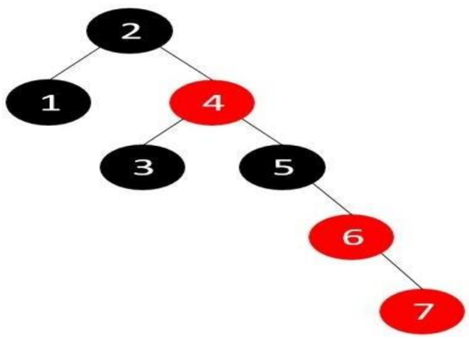
Now,weinsertelement6,buttheRBTreepropertyisviolatedandoneoftheinsertioncasesneedtobeapplied−



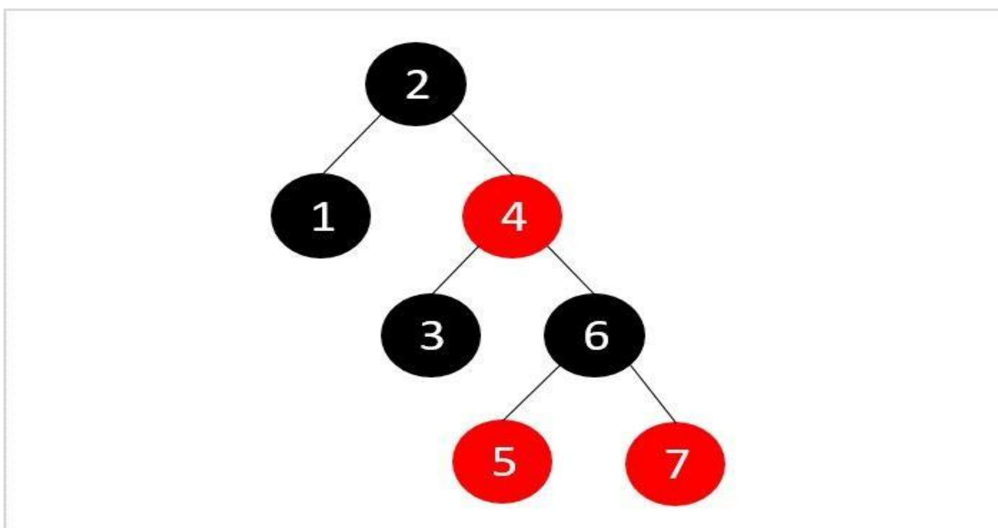
Theparent'ssiblingisred,sowerecolortheparent,parent'ssiblingandthegrandparentnodessincethegrandparentisnottherootnode.



Now,weaddthelastelement,7,buttheparentnodeofthisnewnodeis red.



Sincetheparent'ssiblingisNULL,weapplysuitablerotations(RRrotation)



ThefinalRBTreeisachieved.

DeletioninRedBack tree

Let's understandhowwecan delete the particular node fromthe Red-Black tree. The following are therulesusedto delete the particularnode from the tree:

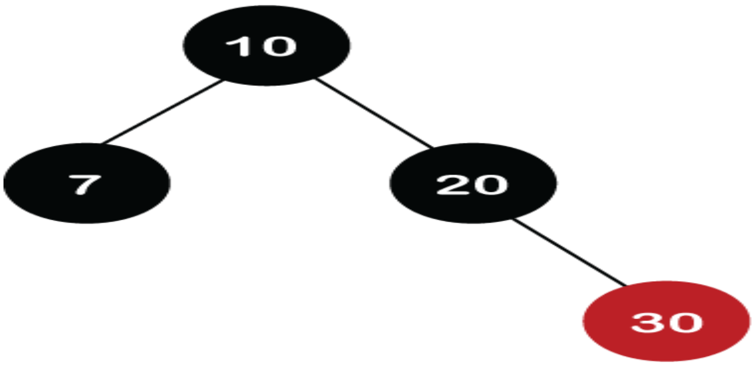
Step1: First,we performBSTrulesfor the deletion.

Step2:

Case1:ifthenodeisRed,whichistobedeleted,wesimplydeleteit. Let's

understand case 1 through an example.

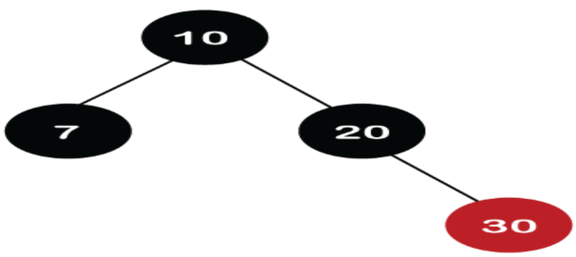
Supposewewanttodeletenode30fromthetree,whichisgivenbelow.



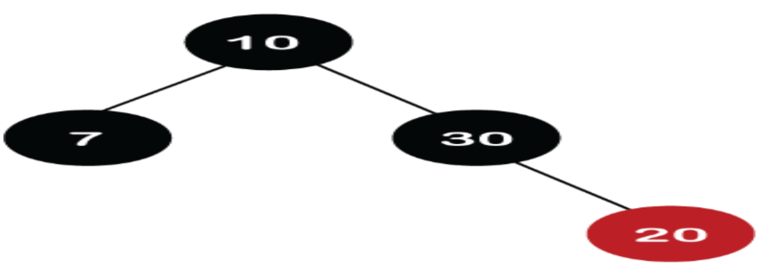
Initially, we are having the address of the root node. First, we will apply BST to search the node. Since 30 is greater than 10 and 20, which means that 30 is the right child of node 20. Node 30 is a leaf node and Red in color, so it is simply deleted from the tree.

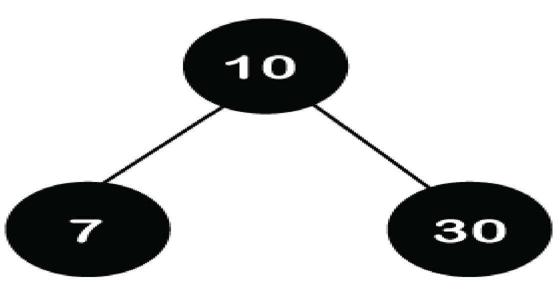
If we want to delete the internal node that has one child. First,replace the value of the internal node with the value of the child node and thensimply delete the child node.

Let'stakeanotherexampleinwhichwewanttodeletetheinternalnode,i.e.,node20.



We cannot delete the internal node; we can onlyreplace the value of that node with another value. Node 20 is at the right of the root node, andit is having only one child, node 30. So, node 20 is replaced with a value 30, but the color of the node would remain the same, i.e., Black. In the end, node 20 (leaf node) is deleted from the tree.

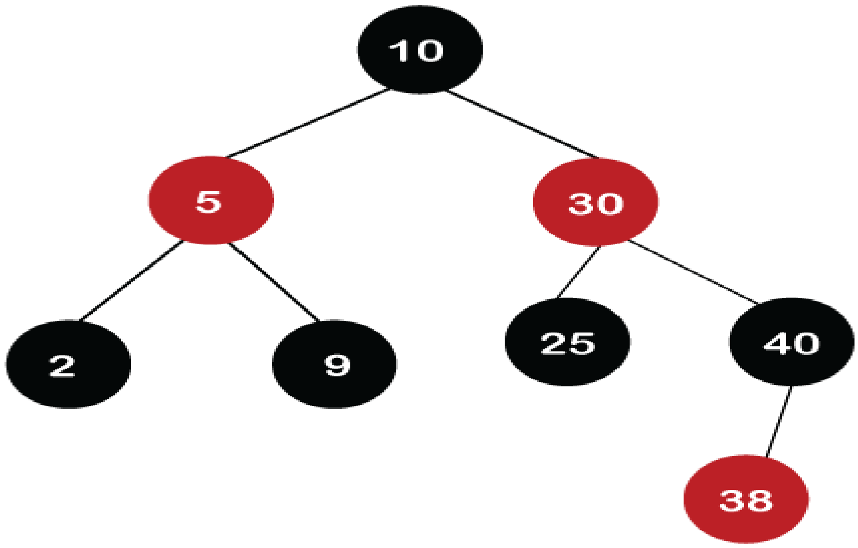




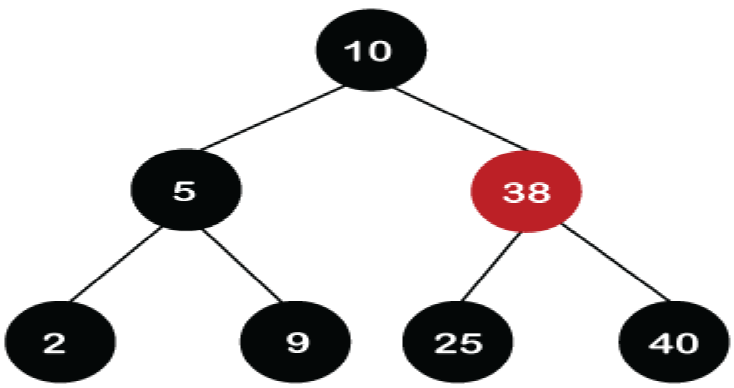
If we want to delete the internal node that has two child nodes. In this case, we have to decide from which we have to replace the value of the internal node (either left subtree or right subtree). We have two ways:

- **Inorderpredecessor:**Wewillreplacewiththe largestvaluethatexists intheleftsubtree.
- **Inordersuccessor:**Wewill replacewith the smallestvalue thatexists in the rightsubtree.

Supposewewanttodeletenode30fromthetree,whichisshownbelow:



Node30isattherightoftherootnode.Inthiscase,wewilluse**theinordersuccessor**.Thevalue38isthesmallestvalueintherightsubtree,sowe willreplacethevalue30with38,butthenodewouldremainthesame,i.e.,Red.Afterreplacement,theleafnode,i.e.,30, wouldbedeletedfromthe tree. Since node 30 is a leaf node and Red in color, we need to delete it (we do not have to perform any rotations or any recoloring).



Case2: Iftherootnodeisalsodoubleblack,thensimplyremovethedoubleblackandmakeitasingleblack.

Case3: Ifthe double black'ssiblingisblackandboth itschildren areblack.

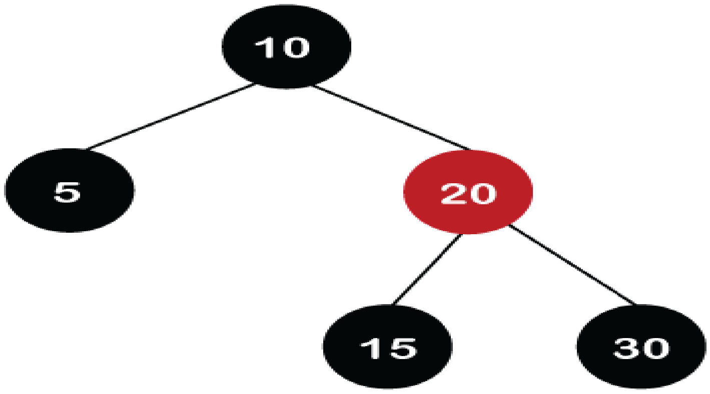
- Removethedoubleblacknode.
- Add thecolorofthenodetotheparent(P)node.

1. IfthecolorofPisredthenitbecomesblack.
2. IfthecolorofPisblack,thenitbecomesdoubleblack.

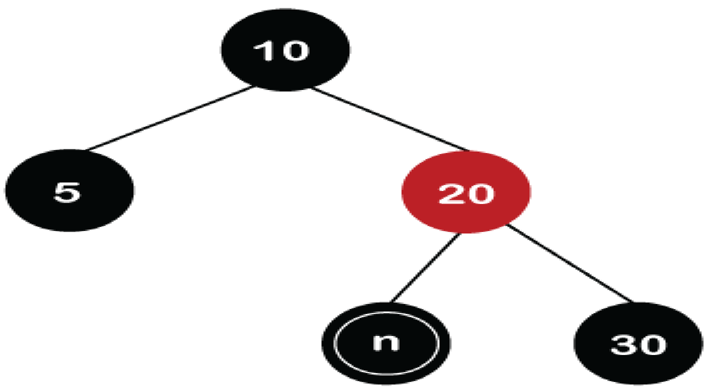
- Thecolorofdoubleblack'ssiblingchangestored.
- Ifstilldoubleblacksituationarises,thenwewillapplyothercases.

Let'sunderstandthiscasethroughanexample.

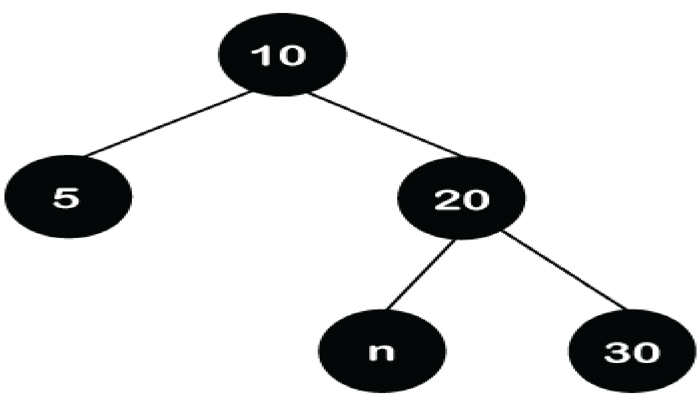
Supposewewanttodeletenode15inthebelowtree.



We cannot simply delete node 15 from the tree as node 15 is Black in color. Node 15 has two children, which are nil. So, we replace the 15 value with a nil value. As node 15 and nil node are black in color, the node becomes double black after replacement, as shown in the below figure.



Intheabovetree,wecanobservethatthedoubleblack'ssiblingisblackincoloranditschildrenarenil,whicharealso black. Asthedouble black's sibling and its children have black so it cannot give its black color to neither of these. Now, thedouble black's parent node is Red so double black's node add its black color to its parent node. The color of the node 20 changes to black while the color of the nil node changes to a single black as shown in the below figure.



Afteraddingthecolortoitsparent node, thecolorofthedoubleblack'ssibling, i.e., node30changestoredas shownin thebelowfigure. In the above

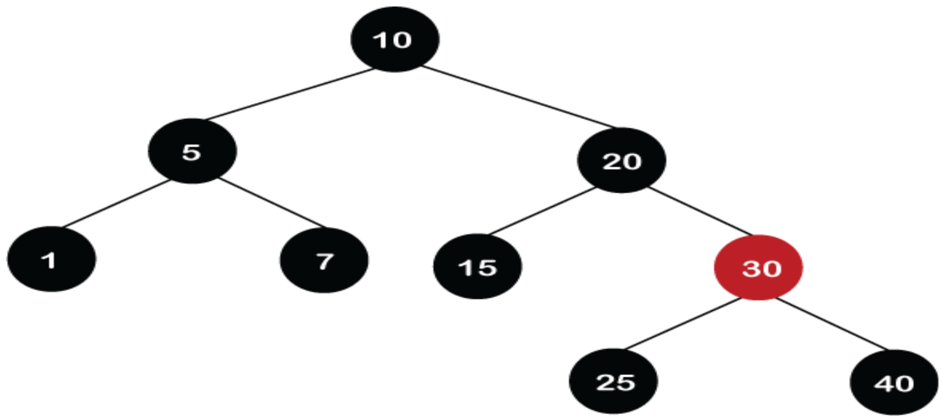
tree, we can observe that there isno longer double black'sproblem exists, andit isalso a Red-Blacktree.

Case4: Ifdoubleblack's siblingisRed.

- Swapthecolorofitsparentanditssibling.
- Rotatetheparentnodeinthedoubleblack'sdirection.
- Reapplycases.

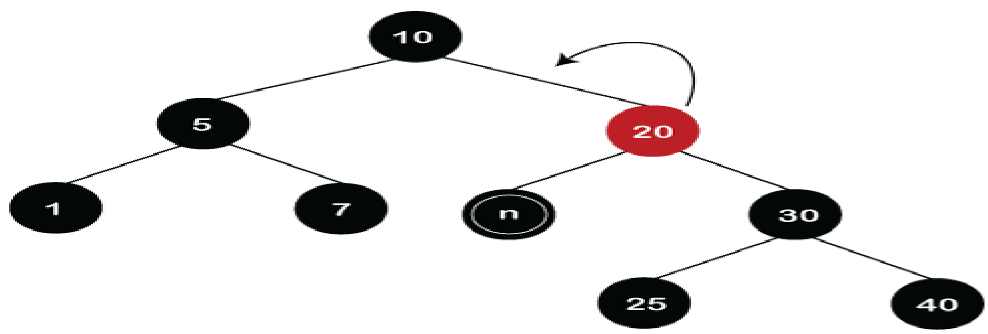
Let'sunderstandthiscasethroughanexample.

Supposewewanttodeletenode 15.

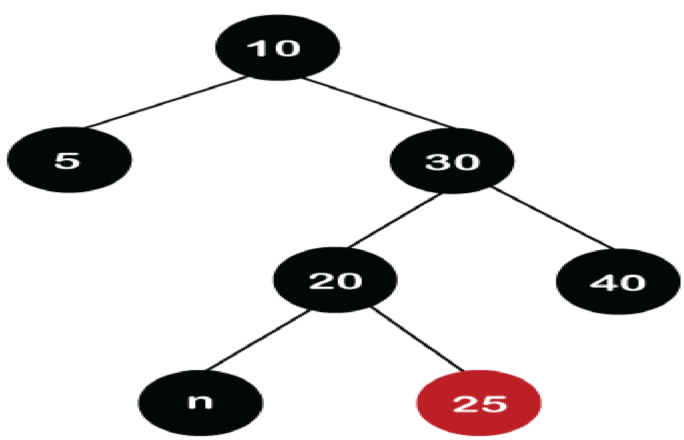


Initially, the 15 is replaced with a nil value. After replacement, the node becomes double black. Since double black's sibling is Red so color of the node 20 changes to Red and the color of the node 30 changes to Black.

Once the swapping of the color is completed, the rotation towards the double black would be performed. The node 30 will move upwards and the node 20 will move downwards as shown in the below figure.



In the above tree, we can observe that double black situation still exists in the tree. It satisfies the case 3 in which double black's sibling is black as well as both its children are black. First, we removethe double black from the node and add the black color to its parent node. At the end, the color of the double black's sibling, i.e., node 25 changes to Red as shown in the below figure.



Intheabovetree,wecanobservethatthedoubleblacksituationhasbeenresolved.ItalsosatisfiesthepropertiesoftheRedBlacktree.

Search

The searchoperationinred-blacktreefollows the same algorithm as that ofa binarysearchtree. The tree is traversedandeachnode is comparedwiththe keyelement to be searched; if found it returns a successful search. Otherwise, it returns an unsuccessful search.

SplayTree

Splaytrees arethealteredversions oftheBinarySearchTrees,sinceitcontains alltheoperations of BSTs,likeinsertion,deletionandsearching,followedbyanother extended operation called **splaying**.

Forinstance,avalue“A”issupposedtobeinsertedintothetree.Ifthetreeisempty,add“A”totherootofthetreeandexit;butifthetreeisnotempty,usebinary search insertion operation to insert the element and then perform splaying on the new node.

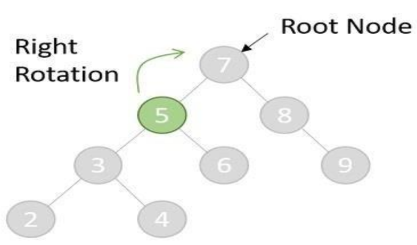
Similarly,aftersearchinganelementinthesplaytree,thenodeconsistingoftheelementmustbesplayedaswell.

Buthowdoweperformsplaying? Splaying,insimplerterms,isjustaprocesstobringanoperationalnodetotheroot.Therearesixtypesofrotationsforit.

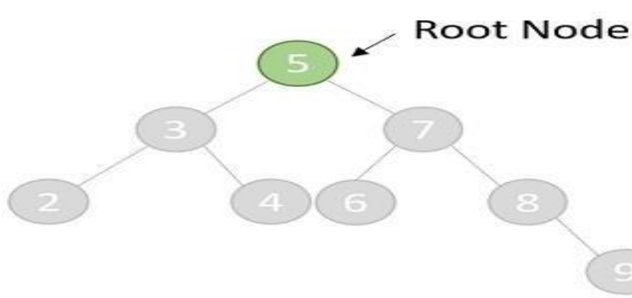
- Zigrotation
- Zagrotation
- Zig-Zigrotation
- Zag-Zagrotation
- Zig-Zagrotation
- Zag-Zigrotation

Zig rotation

Thezigrotationsareperformedwhentheoperationalnodeiseithertherootnodeortheleftchildnodeoftherootnode.Thenodeisrotatedtowardsitsright.

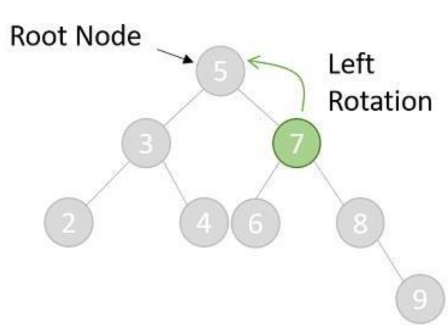


Aftertheshift,thetreewilllooklike−

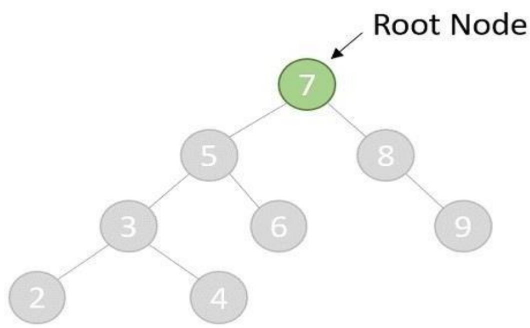


Zagrotation

Thezagrotationsarealsoperformedwhentheoperationalnodeiseithertherootnodeortherightchildnodoftherootnode.Thenodeisrotatedtowardsitsleft.

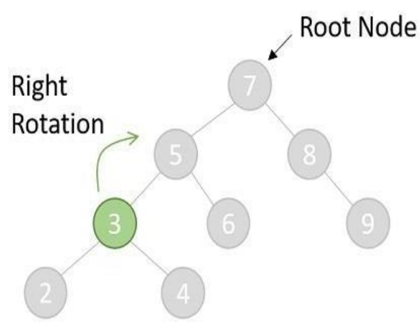


Theoperationalnodebecomestherootnodeaftertheshift−

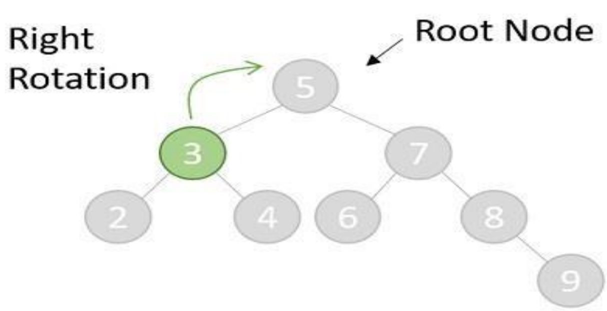


Zig-Zigrotation

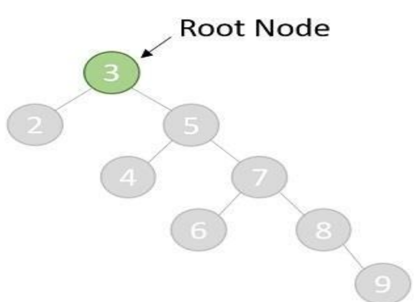
Thezig-zigrotationsareperformedwhentheoperationalnodehasbothparentandagrandparent.Thenodeisrotatedtwoplacestowardsitsright.



Thefirstrotationwillshiftthetreetoonepositionright−

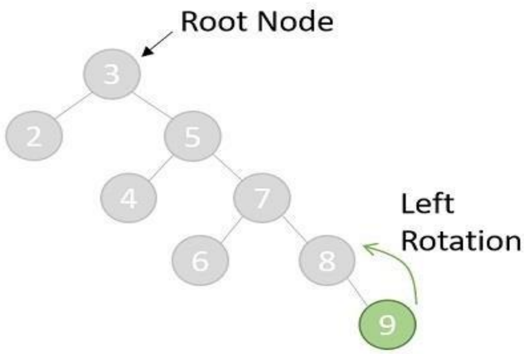


Thesecondrightrotationwillonceagainshiftthenodeforoneposition.Thefinaltreeaftertheshiftwilllooklikethis−

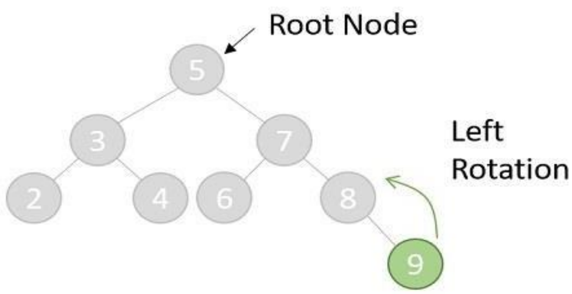


Zag-Zagrotation

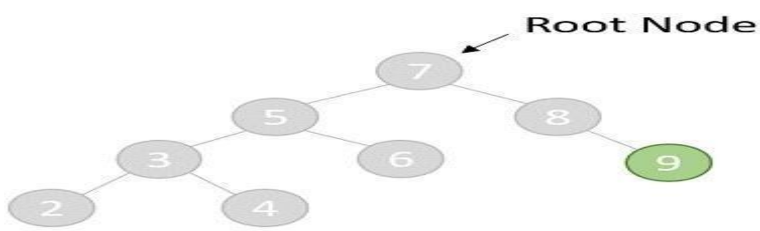
Thezag-zagrotationsarealsoperformedwhentheoperationalnodehasbothparentandagrandparent.Thenodeisrotatedtwoplacestowardsitsleft.



Afterthefirstrotation,thetreewilllooklike−

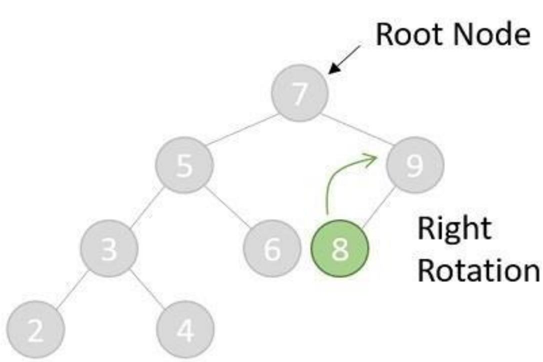


Thenthefinaltreeafterthesecondrotationisgivenasfollows.However,theoperationalnodeisstillnottherootsothe splayingisconsideredi ncomplete.Hence,other suitable rotations are again applied in this case until the node becomes the root.

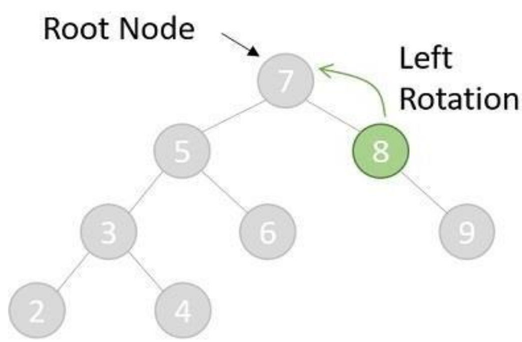


Zig-Zagrotation

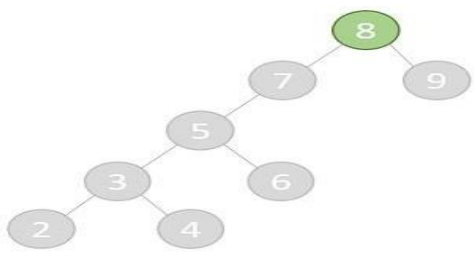
The zig-zag rotations are performed when the operational node has both a parent and a grandparent. But the difference is the grandpar ent, parent and child are inLRL format. The node is rotated first towards its right followed by left.



Afterthefirstrotation,thetreeis−

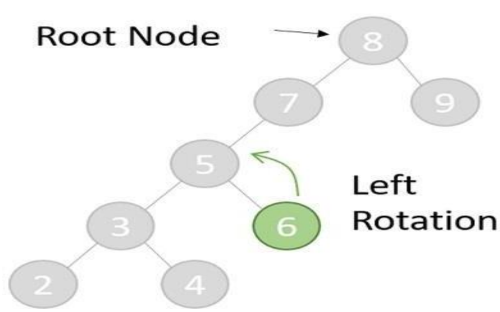


Thefinaltreeafterthesecondrotation–

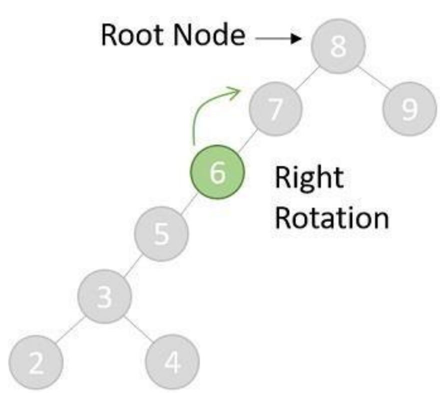


Zag-Zigrotation

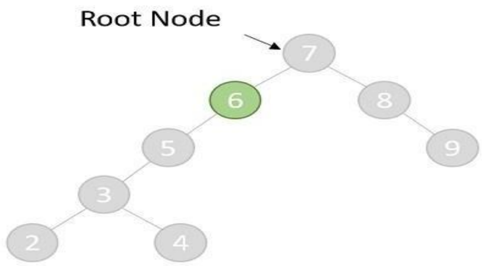
The zag-zig rotations are also performed when the operational node has both parent and grandparent. But the difference isthe grandparent, parent and child are inRLR format. The node is rotated first towards its left followed by right.



Firstrotationisperformed,thetreeisobtainedas–



After second rotation, the final tree is given as below. However, the operational node is not the root node yet so one more rotation needs to be performed to makethe said node as the root.



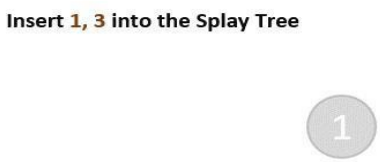
BasicOperationsofSplayTrees

A splay contains the same basic operations that a Binary Search Tree provides with: Insertion, Deletion, and Search. However, after every operation there is anadditional operation that differs them from Binary Search tree operations: Splaying. We have learned about Splaying already so let us understand the proceduresof the other operations.

Insertion

The insertion operationin a Splay tree isperformed in the exact same way insertion in abinary search tree isperformed. The procedure to performthe insertion ina splay tree is given as follows −

- Checkwhetherthetreeisempty;ifyes,addthenewnodeandexit



- Ifthetreeisnotempty,addthenewnodetotheexistingtreeusingthebinarysearchinsertion.

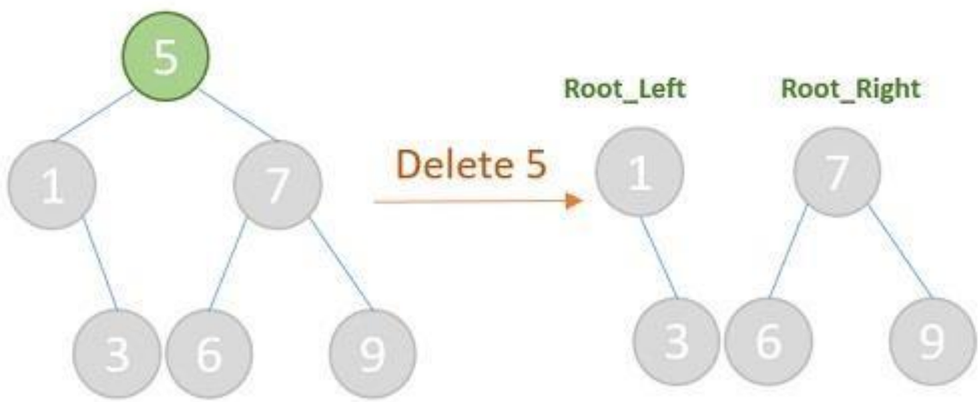
- Then,suitablesplayingischosenandappliedonthenewlyadded node.

Zag(Left)Rotationisappliedonthenew node

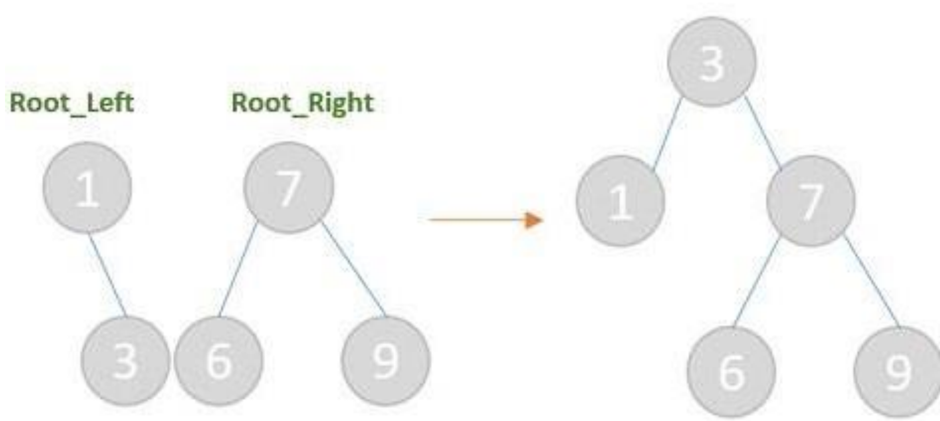
Deletion

Thedeletionoperationinasplaytreeisperformedasfollowing−

- Applysplayingoperationonthenodetobedeleted.
- Once,thenodeismadetheroot,deletethenode.
- Now,thetreeissplitintotwotrees,theleftsubtreeandtherightsubtree;withtheirrespectivefirstnodesastherootnodes:sayroot_leftand root_right.



- Ifroot_leftisaNULLvalue,thentheroot_right willbecometherootofthetree.Andviceversa.
- But if both root_left and root_right are not NULL values, then select the maximum value from the left subtree and make it thenew root by connecting thesubtrees.



Search

ThesearchoperationinaSplaytree followsthesameprocedureof theBinarySearchTreeoperation.However,afterthesearchingisdoneandtheelementisfound, splaying is applied on the node searched. If the element is not found, then unsuccessful search is prompted.

MODULE-4

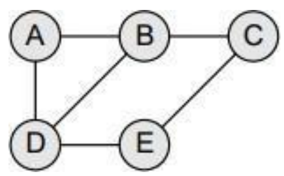
GRAPHS

INTRODUCTION

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

DEFINITION:

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.



UndirectedGraph

The above Figure shows a Graph with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}. Notethat there are five vertices or nodes and six edges in the graph.

A graph can be Directed or Undirected. In an Undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B aswell as from B to A.The above Figure shows an undirected graph because it does not give any information about the direction of the edges.

Look at the Below Figure which shows a Directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

GRAPHTERMINOLOGY:

Weusethefollowingtermsingraphdatastructure.

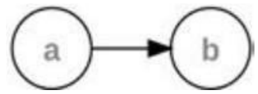
1. Vertex: An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

2. Edge: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex , ending Vertex). For example, the link between vertices A and B is represented as (A,B). In above graph, there are 7 edges (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E).

Edgesarethreetypes:

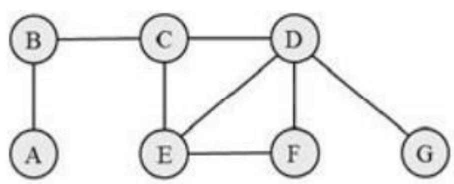
A. UndirectedEdge: Anundirectededgeisabidirectionaledge.Ifthereisaundirectededge between vertices A and B then edge (a , b) is equal to edge (b, a).

B. DirectedEdge: Adirectededgeisaunidirectionaledge.If thereisadirectededgebetween vertices A and B then edge (A , B) is not equal to edge (B , A).

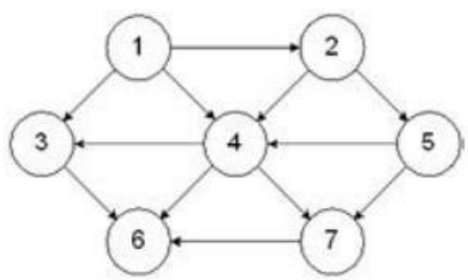


C. WeightedEdge:-Aweighted edgeis anedge with costonit.

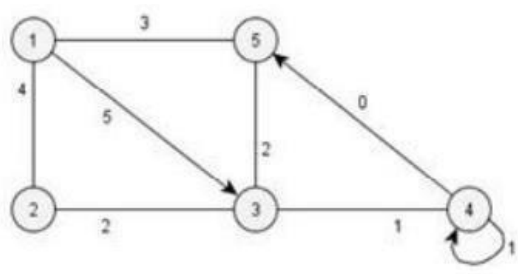
3. Undirected Graph: Agraphwithonlyundirected edges issaidtobeundirected graph.



4. Directed Graph: A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



5. MixedGraph: Agraphwithundirectedanddirectededgesissaidtobemixed graph.



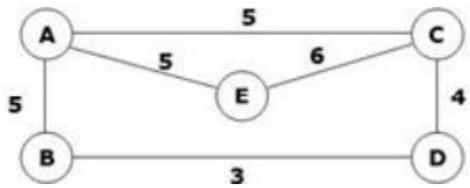
6. Weighted Graph: In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned toit. If you want to go from vertex 1 to vertex 3, you can take one ofthe following 3 paths:

a. 1->2->3

b. 1->3

c. 1->4->3

Thereforethe totalcost ofeachpathwillbeasfollows:-Thetotalcostof1->2->3willbe(1+2) i.e. 3units - Thetotalcost of1 ->3 will be 1 unit - Thetotalcostof1 -> 4-> 3will be (3 +2) i.e. 5 units



7. EndverticesorEndpoints: Thetwoverticesjoinedbyanedgearecalledtheendvertices(or endpoints) of the edge.

8. Origin: Ifanedgeisdirected,itsfirstendpointissaid tobeoriginofit.

9. Destination: Ifanedgeisdirected,itsfirstendpointissaidtobeoriginofitandtheother endpoint is said to be the destination of the edge.

10. Adjacent: IfthereisanedgebetweenverticesAandBthenbothAandBaresaidtobe adjacent.

11. Degree: Totalnumberofedges connectedtoavertexissaid tobedegreeofthatvertex.

12. Indegree: Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

13. Outdegree: Totalnumber ofoutgoing edgesconnectedto avertexissaid to beoutdegreeofthat vertex.

14. Path: Apath is a sequence ofalternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

GraphTraversals:(SearchesinGraphs):

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversalfindstheegdestobeusedinthesearchprocesswithoutcreatingloopsthatmeansusing graph traversal we visit all vertices of graph without getting into looping path.

Therearetwographtraversaltechniquesandtheyareas follows.

1. DFS(DepthFirstSearch)
2. BFS(BreadthFirst Search)

1. Depth-firstSearchAlgorithm:

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

WeusethefollowingstepstoimplementDFS traversal:

Step1: DefineaStackofsizetotalnumber ofverticesinthegraph.

Step2: Selectanyvertex as startingpointfor traversal.Visitthatvertex andpushiton to the Stack.

Step 3: Visitany oneof theadjacentvertex of thevertex whichis at top of thestack whichis not visited and push it on to the stack.

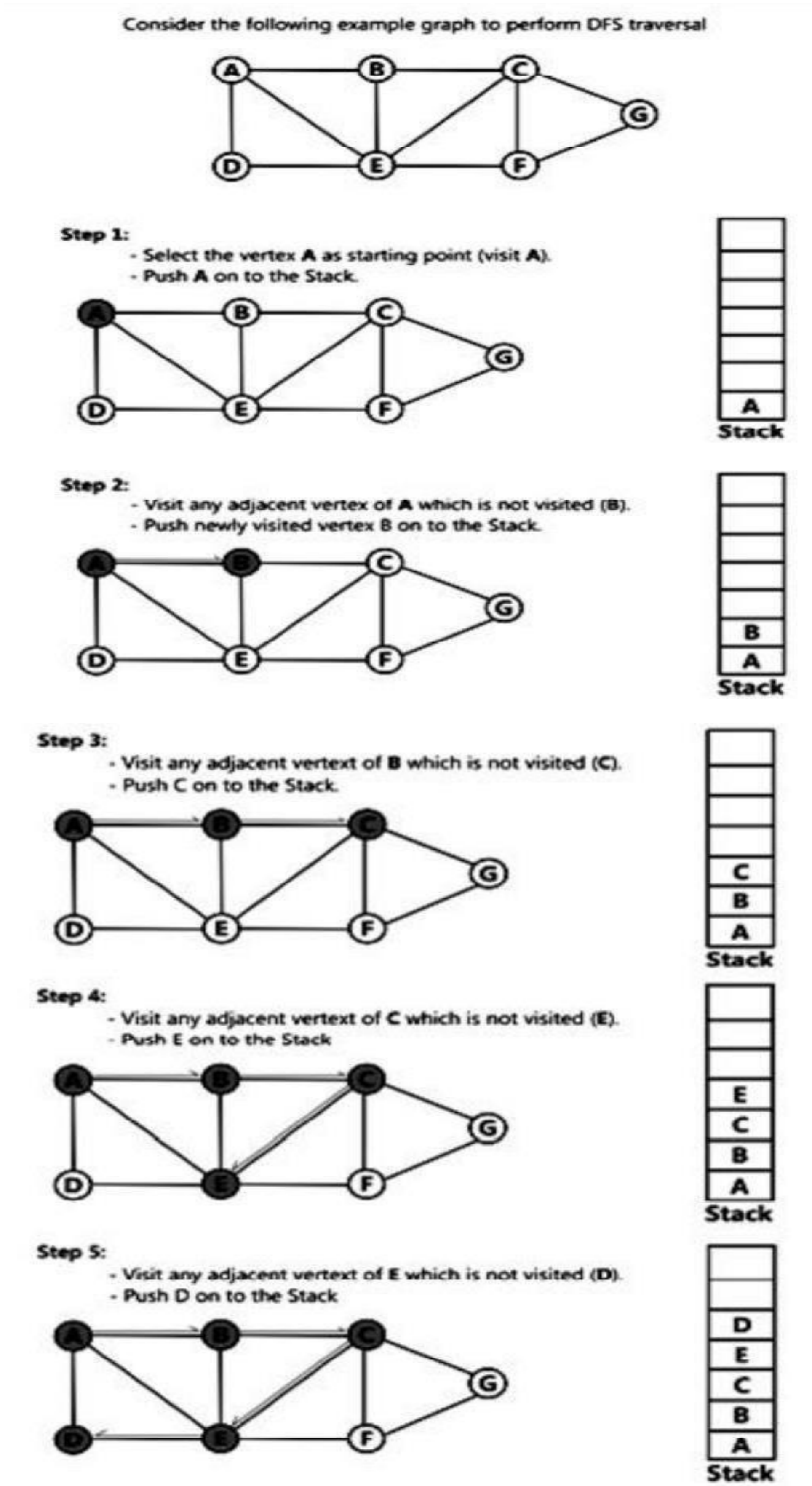
Step4: Repeatstep3untiltherearenonewvertextobevisitfromthevertexontopofthe stack.

Step 5: Whenthere is no new vertexto be visit thenuse back tracking and pop one vertex from the stack.

Step 6: Repeatsteps3,4and5untilstackbecomesEmpty.

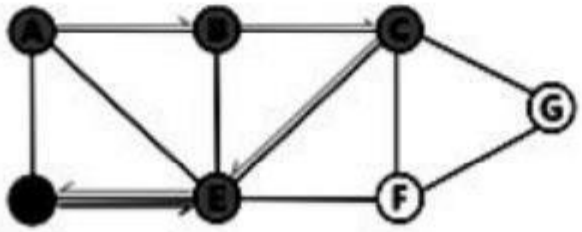
Step7: Whenstack becomesEmpty,thenproducefinalspanningtreebyremovingunused edges from the graph

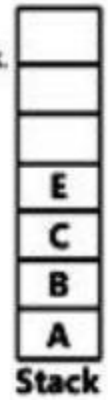
Example:


Consider the following example graph to perform DFS traversal
A
B
C
D
E
F
G
Step 1:
- Select the vertex A as starting point (visit A).
- Push A on to the Stack.
A
Stack
Step 2:
- Visit any adjacent vertex of A which is not visited (B).
- Push newly visited vertex B on to the Stack.
B
A
Stack
Step 3:
- Visit any adjacent vertext of B which is not visited (C).
- Push C on to the Stack.
C
B
A
Stack
Step 4:
- Visit any adjacent vertext of C which is not visited (E).
- Push E on to the Stack
E
C
B
A
Stack
Step 5:
- Visit any adjacent vertext of E which is not visited (D).
- Push D on to the Stack
D
E
C
B
A
Stack

Step 6:

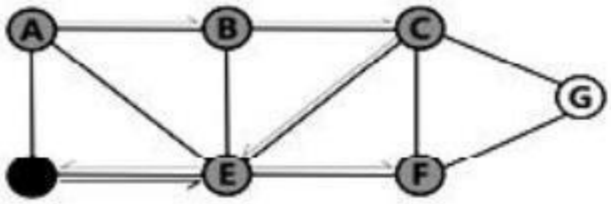
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

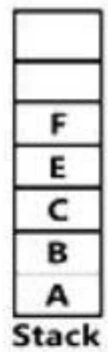




Step 7:

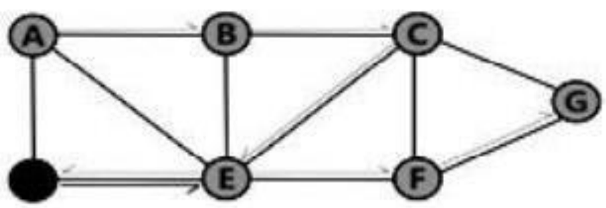
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

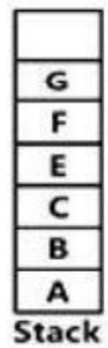




Step 8:

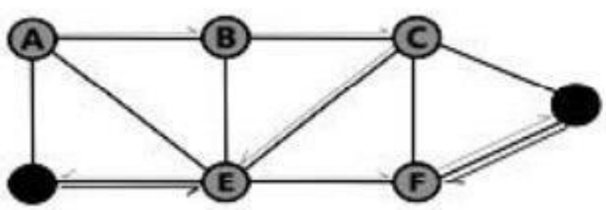
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

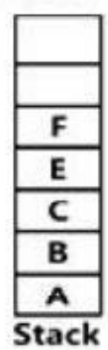




Step 9:

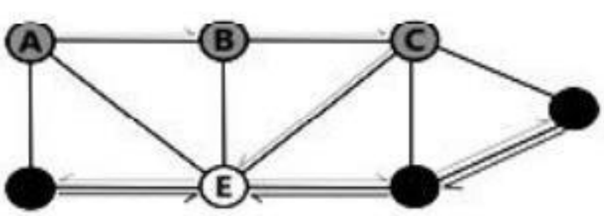
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

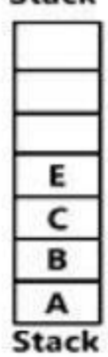




Step 10:

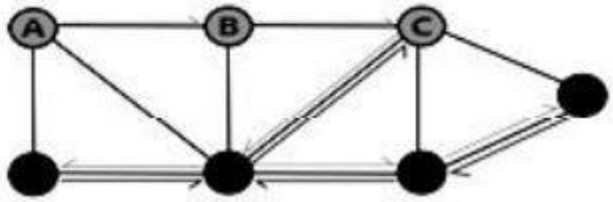
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

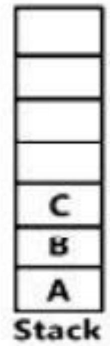




Step 11:

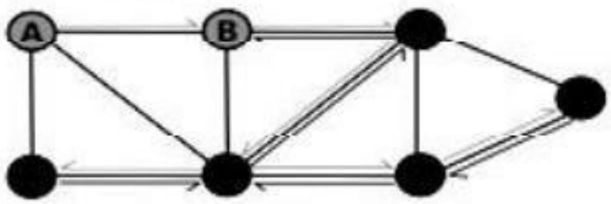
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.





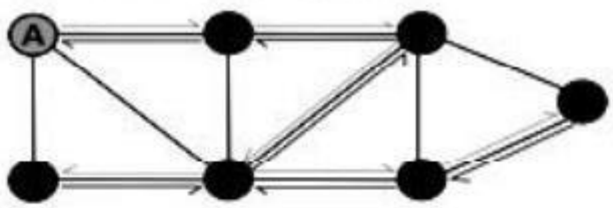
Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



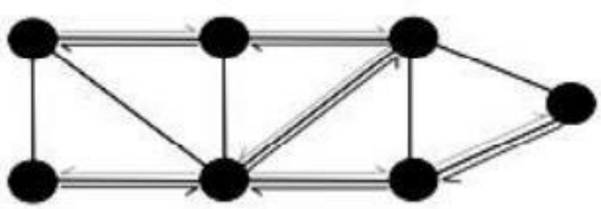
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

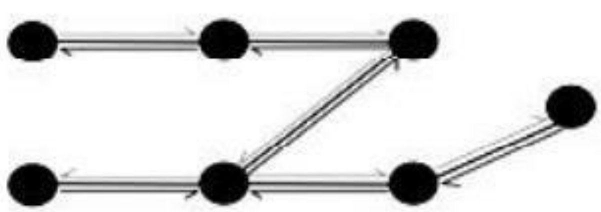


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



ApplicationsofDepth-FirstSearchAlgorithm :

Depth-firstsearchisusefulfor:

1. Findingapathbetweentwospecifiednodes,u andv,ofanunweighted graph.
2. Findingapathbetweentwospecifiednodes,u andv,ofaweighted graph.
3. Findingwhetheragraphisconnectedornot.
4. Computingthespanningtreeofaconnectedgraph.

Breadth-FirstSearchAlgorithm:

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes,thealgorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

WeusethefollowingstepstoimplementBFS traversal:

Step1: DefineaQueueofsizetotalnumberofverticesinthegraph.

Step2: Selectanyvertexasstartingpointfortraversal.Visitthatvertexandinsertitintothe Queue.

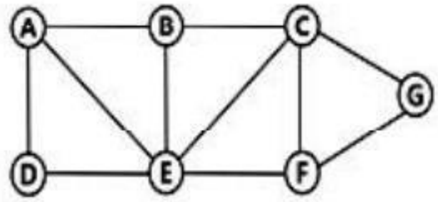
Step3: Visitall theadjacentvertices of thevertex whichis atfrontof theQueuewhichis not visited and insert them into the Queue.

Step 4: When there is no new vertexto be visit fromthe vertexat front ofthe Queue then delete that vertex from the Queue.

Step5: Repeatstep3and 4untilqueuebecomesempty.

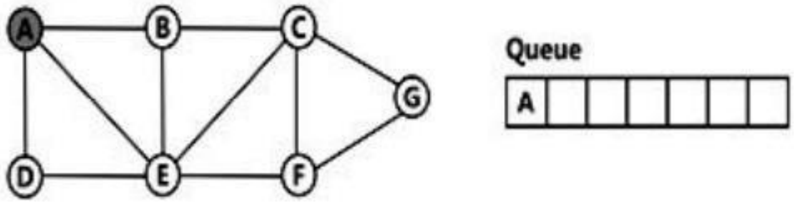
Step6: WhenqueuebecomesEmpty,thenproducefinalspanningtreebyremovingunused edges from the graph.

Consider the following example graph to perform BFS traversal



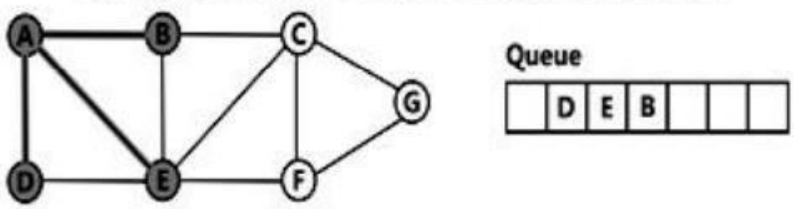
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



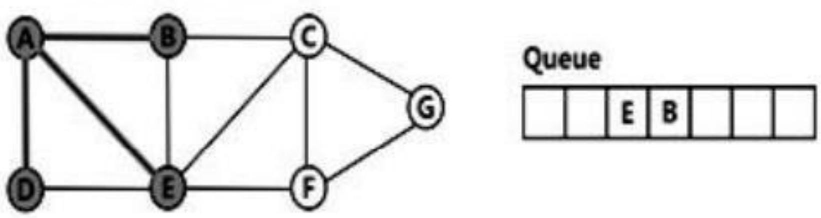
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



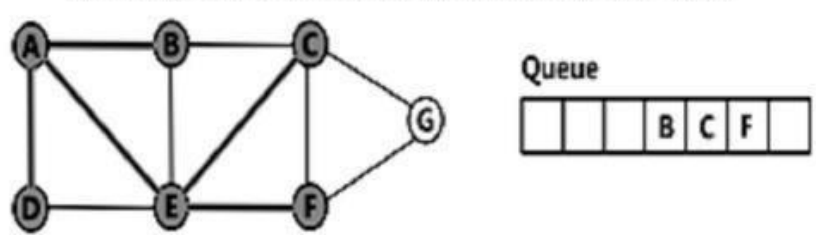
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



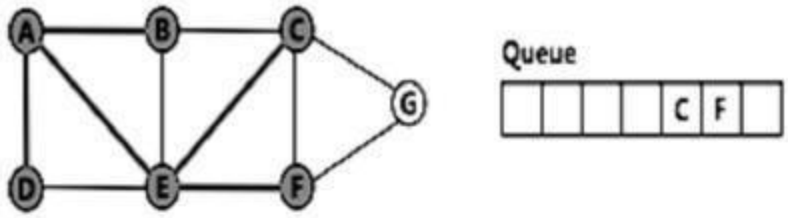
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



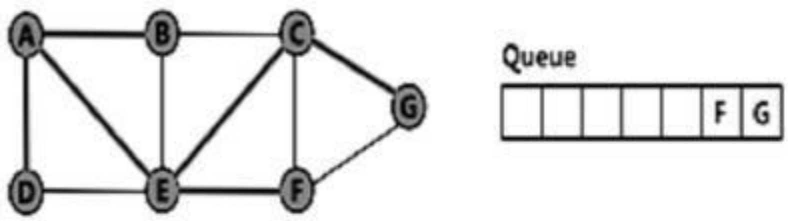
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



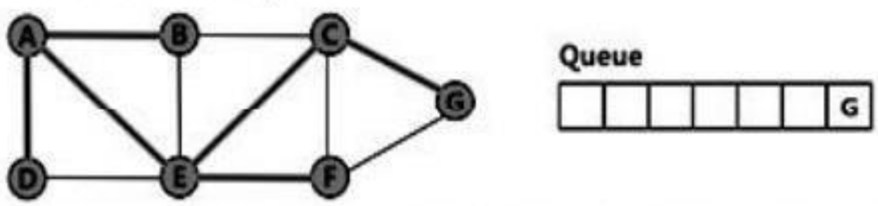
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



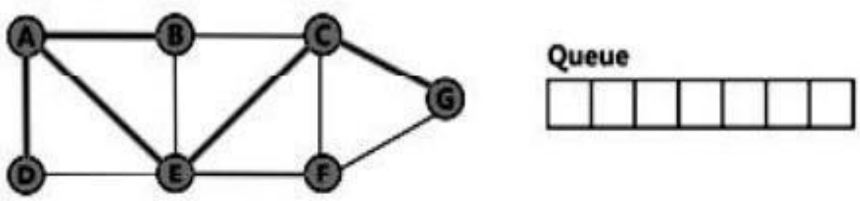
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

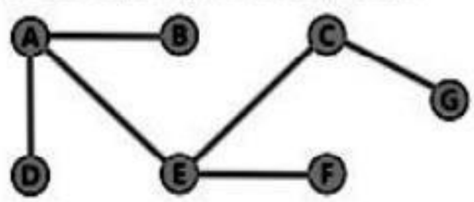


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Sorting

QuickSort Algorithm

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes**n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solvingthe subproblems, and combining the results back together to solve the original problem.

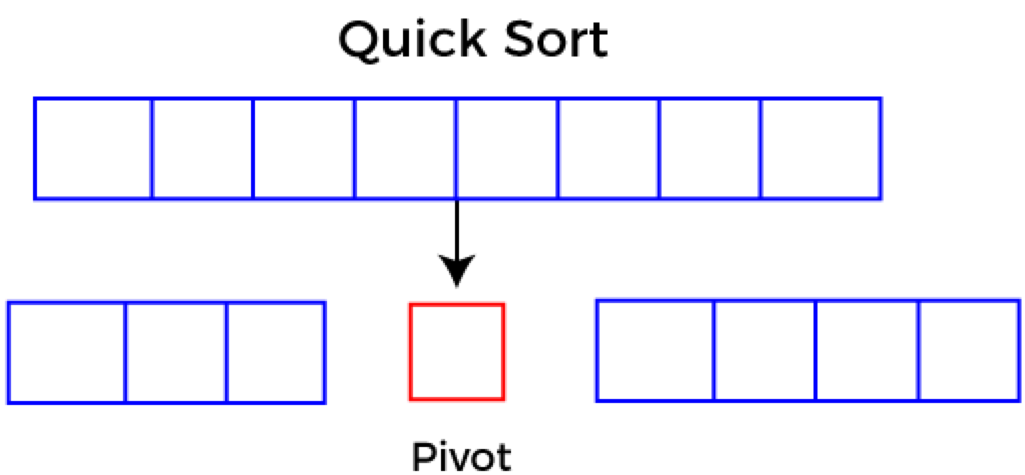
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively,sorttwosubarrayswithQuicksort.

Combine: Combinethealreadysorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivotcanberandom, i.e.selecttherandom pivotfromthegiven array.
- Pivotcaneither bethe rightmost elementofthe leftmostelementof thegiven array.
- Selectmedianasthepivot element.

Algorithm

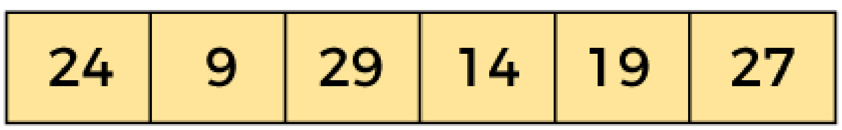
```
QUICKSORT(arrayA,start, end)
{
 1if(start<end)
 2 {
3p=partition(A,start, end)
4QUICKSORT(A,start,p-1)
5QUICKSORT(A,p+ 1, end)
6 }
}
```

WorkingofQuickSort Algorithm

Now,let's seethe working of theQuicksort Algorithm.

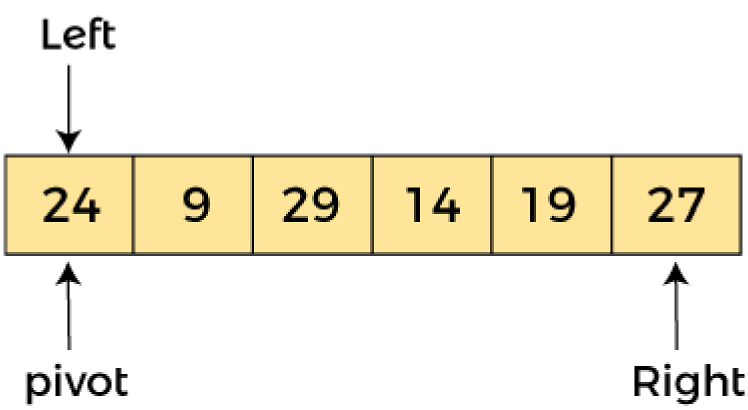
Tounderstandtheworkingofquicksort,let'stake anunsortedarray.Itwillmake theconceptmore clear and understandable.

Lettheelementsofarrayare-

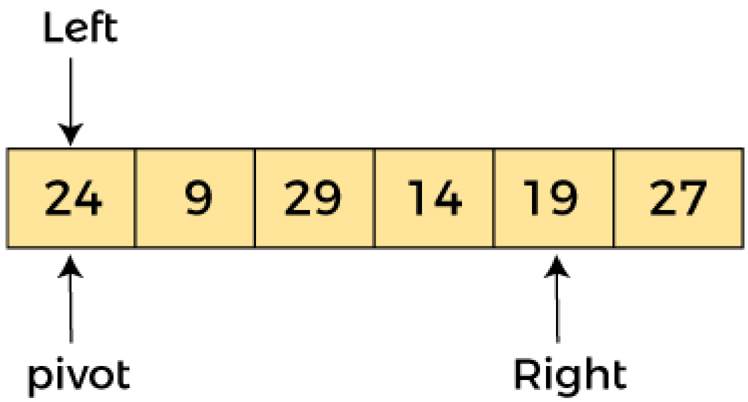


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] =27 and a[pivot] = 24.

Since,pivotisatleft, soalgorithmstartsfrom rightandmovetowards left.

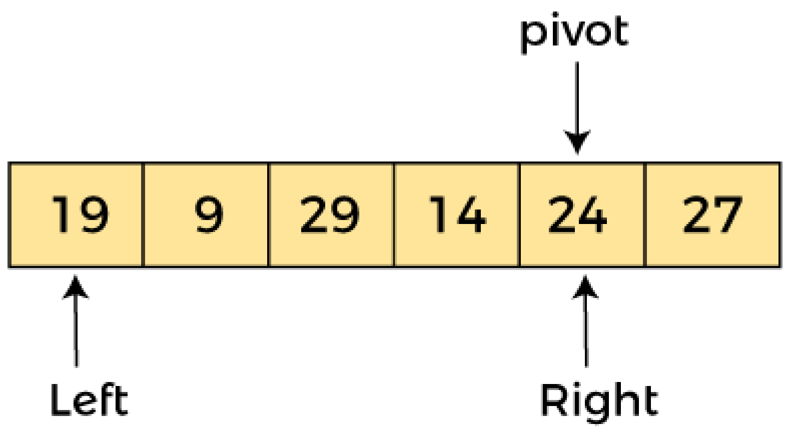


Now,a[pivot]<a[right], soalgorithmmovesforwardonepositiontowards left,i.e.-



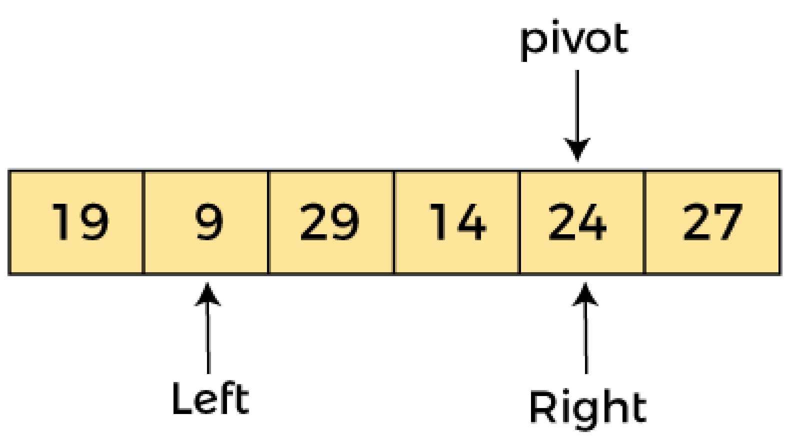
Now,a[left]=24,a[right] =19, anda[pivot] =24.

Because,a[pivot]>a[right], so,algorithmwill swapa[pivot]with a[right], andpivot movestoright, as -

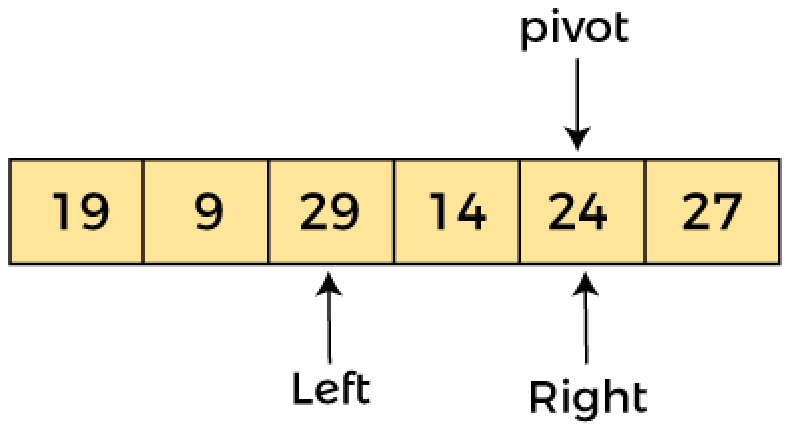


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot isat right, so algorithm starts from left and moves to right.

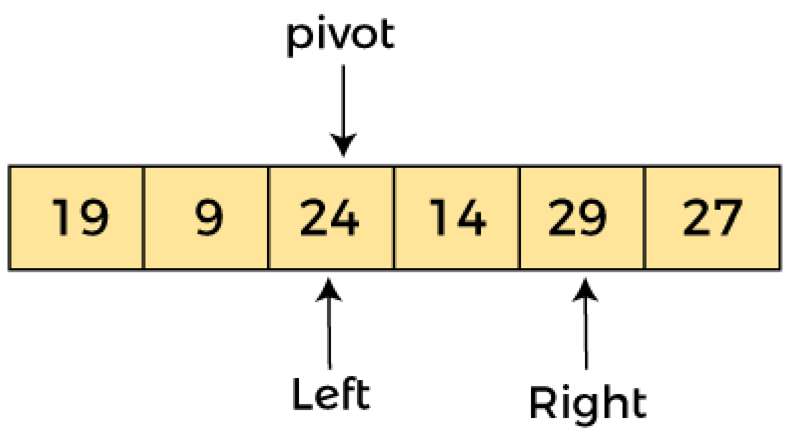
Asa[pivot]> a[left],soalgorithm movesoneposition torightas-



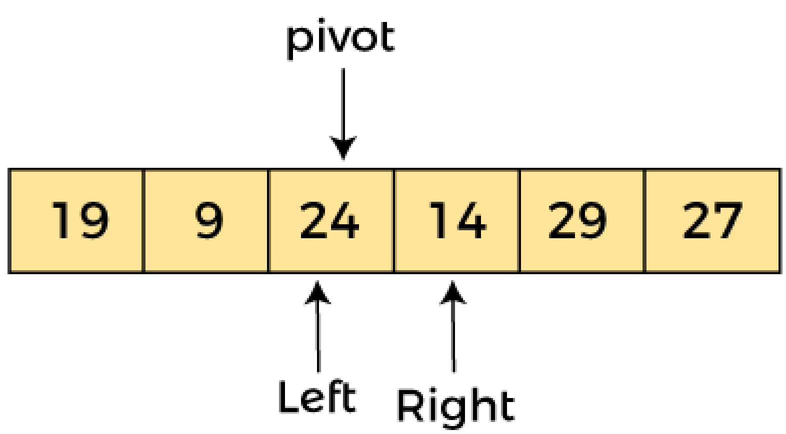
Now,a[left]=9,a[right]=24,anda[pivot]=24.Asa[pivot]>a[left],soalgorithmmovesone position to right as -



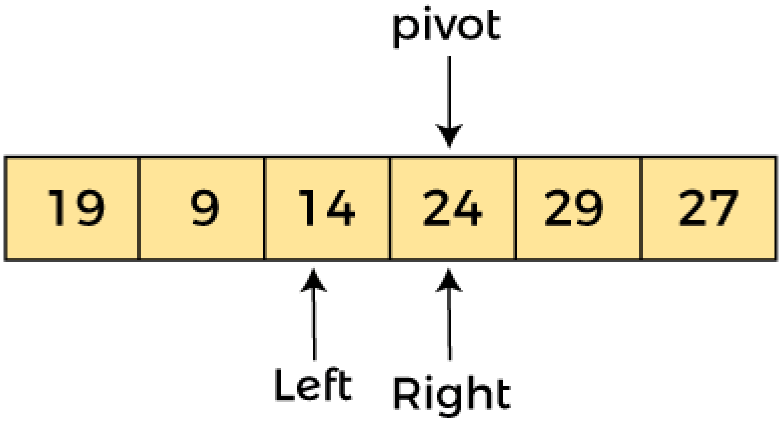
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



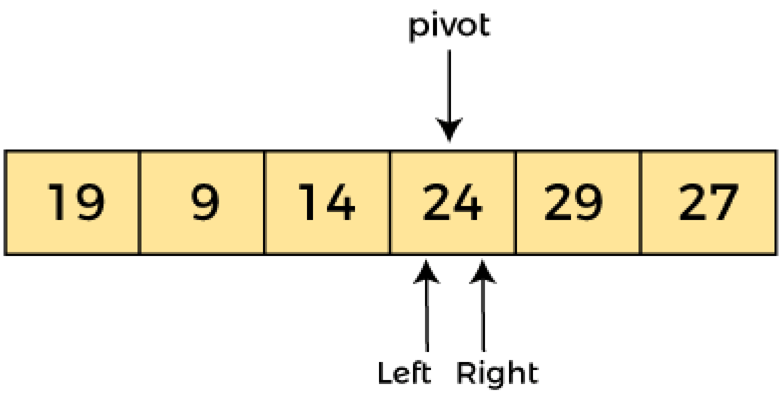
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now,a[pivot]=24,a[left]=24,anda[right]=14.Asa[pivot]>a[right],so,swapa[pivot]and a[right], now pivot is at right, i.e. -



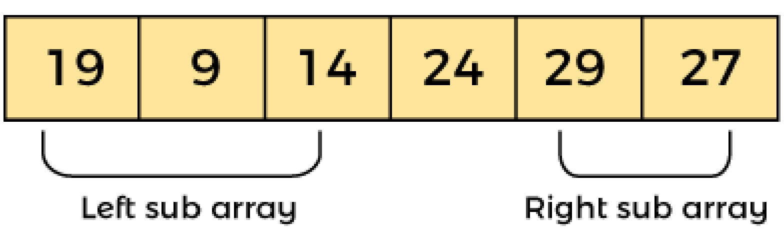
Now,a[pivot]=24,a[left]=14,anda[right]=24.Pivotisatright, sothealgorithmstartsfromleft and move to right.



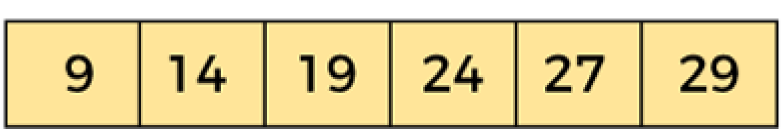
Now,a[pivot]=24,a[left]=24,anda[right]=24.So,pivot,leftandrightarepointingthesame element. It represents the termination of procedure.

Element24,whichisthe pivotelementisplacedatitsexact position.

Elementsthatarerightsideofelement24aregreaterthanit,andtheelementsthatareleftsideof element 24 are smaller than it.



Now,inasimilarmanner,quicksortalgorithmisseparatelyappliedtotheleftandrightsub-arrays. After sorting gets done, the array will be -



Quicksortcomplexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. TimeComplexity

Case	Time Complexity
BestCase	O(n*logn)
AverageCase	O(n*logn)
WorstCase	$O(n^2)$

.

2. SpaceComplexity

Space Complexity	O(n*logn)
Stable	NO

- Thespacecomplexityofquicksortis O(n*logn).

Implementationof quicksort

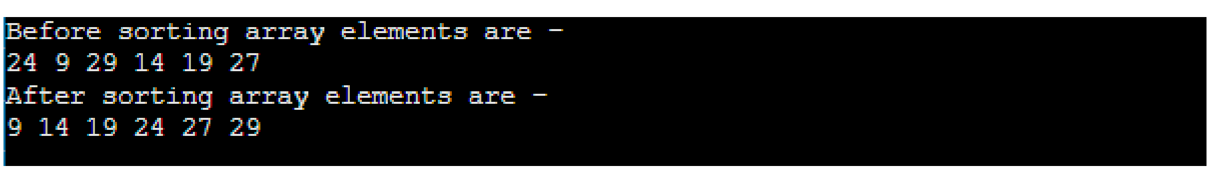
Now,let'sseetheprograms ofquicksortin differentprogramming languages.

Program: WriteaprogramtoimplementquicksortinClanguage.

```
#include <stdio.h>
/*functionthatconsiderlastelement aspivot,
placethepivotatitsexactposition,andplace smaller
elements to left of pivot and greater elements to
right of pivot.*/
intpartition (inta[],intstart, int end)
{
   intpivot=a[end];// pivotelement
   int i =(start-1);

   for(intj =start;j <=end -1;j++)
   {
      //If currentelementissmallerthanthepivot
      if(a[j] < pivot)
      {
         i++;//incrementindexofsmallerelement
         intt=a[i];
         a[i] = a[j];
         a[j] = t;
      }
```

```
    }
    int t = a[i+1];
    a[i+1]=a[end];
    a[end] = t;
    return (i + 1);
}

/*function toimplementquicksort */
voidquick(inta[],intstart, intend)/*a[]=arrayto besorted,start =Starting index,end=Ending index
*/
{
    if(start <end)
    {
        intp=partition(a,start,end);//pisthepartitioningindex
        quick(a, start, p - 1);
        quick(a,p+1,end);
    }
}

/*functiontoprintanarray */
void printArr(inta[],intn)
{
    inti;
    for(i=0;i<n;i++)
        printf("%d", a[i]);
}
int main()
{
    inta[]={ 24, 9, 29, 14, 19, 27};
    intn =sizeof(a)/ sizeof(a[0]);
    printf("Beforesortingarrayelementsare-\n");
    printArr(a, n);
    quick(a,0,n-1);
    printf("\nAftersortingarrayelementsare-\n");
    printArr(a, n);

    return0;
}
```

Output:



Heap Sort Algorithm

In this article, we will discuss the Heapsort Algorithm. Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heapsortbasicallyrecursivelyperformstwomainoperations-

- BuildaheapH,usingtheelementsofarray.
- Repeatedlydeletetherootelementoftheheapformedin1st phase.

Beforeknowingmoreabouttheheapsort,let'sfirstseeabriefdescriptionof**Heap.**

Whatisaheap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Whatisheapsort?

Heapsort isapopularand efficient sortingalgorithm.Theconceptofheapsortistoeliminatetheelementsonebyonefrom the heap part of the list, and then insert them into the sorted part of the list.

Heapsortisthein-placesortingalgorithm. Now,

let's see the algorithm of heap sort.

WorkingofHeapsortAlgorithm

Now,let'sseetheworkingoftheHeapsortAlgorithm.

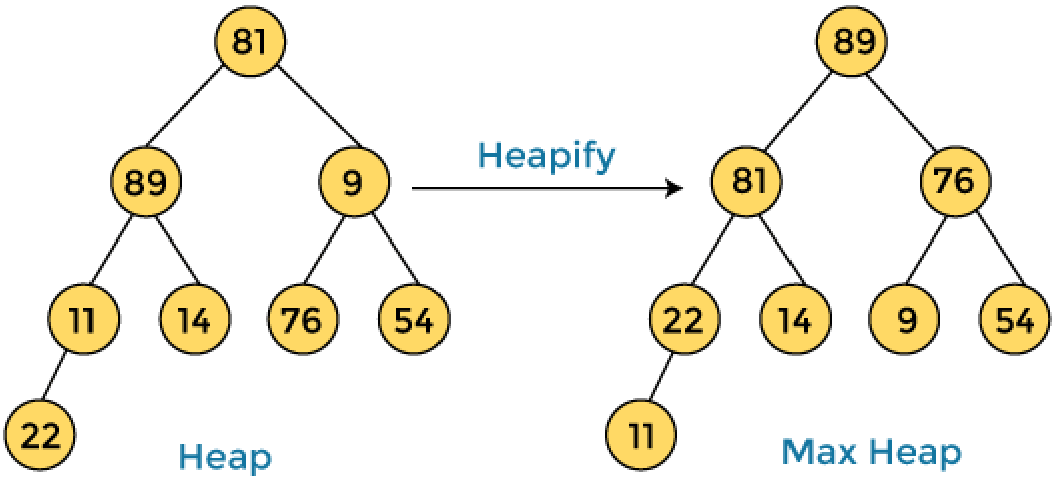
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- Thefirststepincludesthecreationofaheapbyadjustingtheelementsofthearray.
- Afterthecreationofheap,nowremovetherootelementoftheheaprepeatedlybyshiftingittotheendofthe array, and then store the heap structure with the remaining elements.

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

81	89	9	11	14	76	54	22

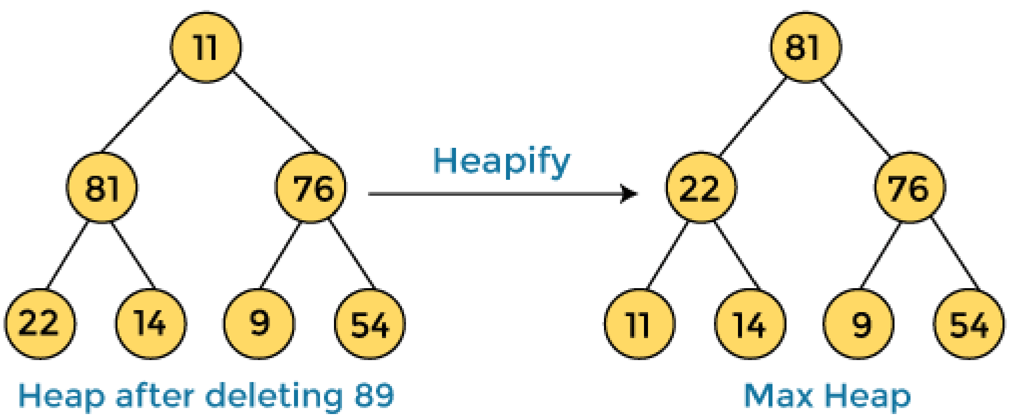
First,wehavetoconstructaheapfromthegivenarrayandconvertitintomaxheap.



Afterconvertingthegivenheapintomaxheap,thearrayelementsare-

89	81	76	22	14	9	54	11

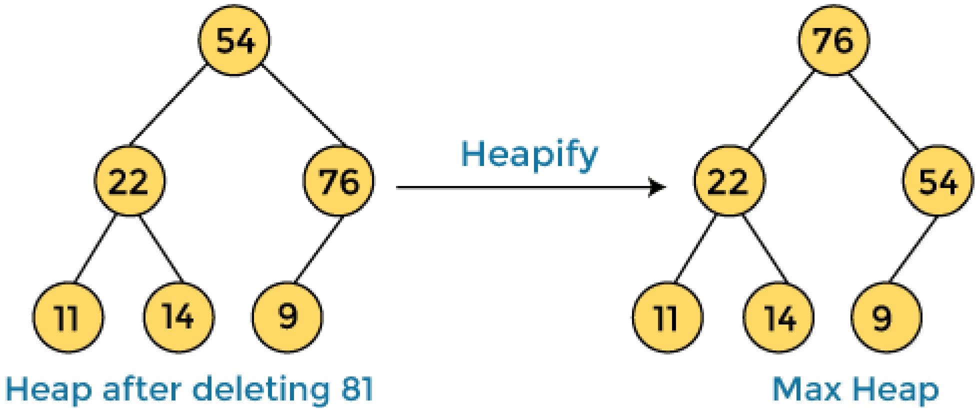
Next,wehavetodeletetherootelement **(89)**fromthemaxheap.Todeletethisnode,wehavetoswapitwiththelast node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



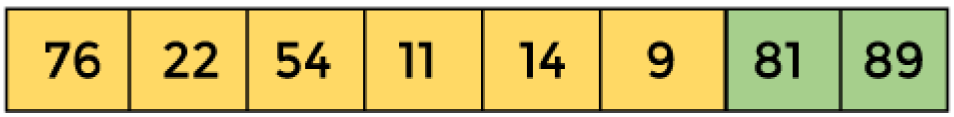
Afterswappingthearrayelement**89**with**11,**andconvertingtheheapintomax-heap,theelementsofarrayare-

81	22	76	11	14	9	54	89

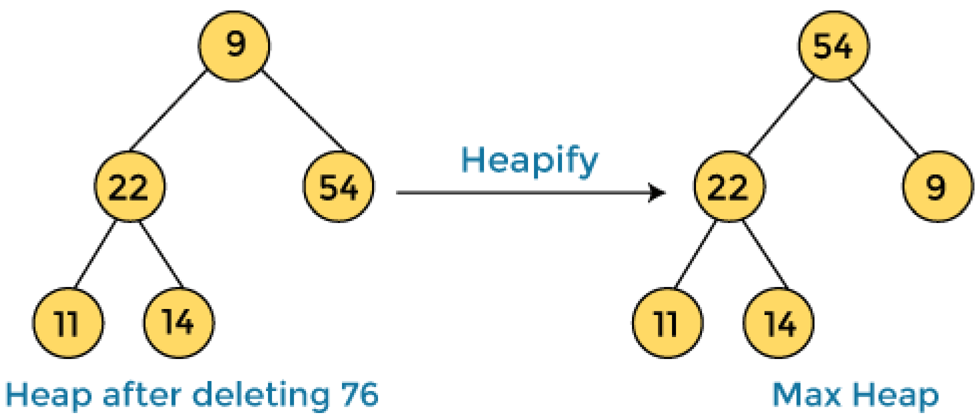
In the next step, again, we have to delete the root element**(81)**from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



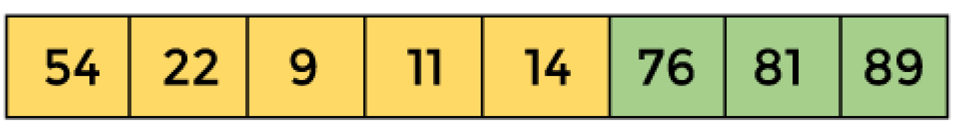
Afterswappingthearrayelement**81**with**54**andconvertingtheheapintomax-heap,theelementsofarrayare-



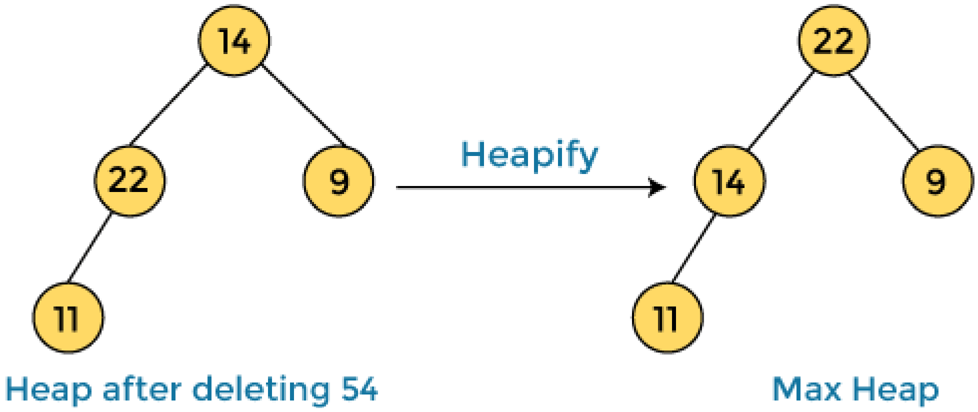
In the next step, we have to delete the root element **(76)**from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



Afterswappingthearrayelement**76**with**9**andconvertingtheheapintomax-heap,theelementsofarrayare-



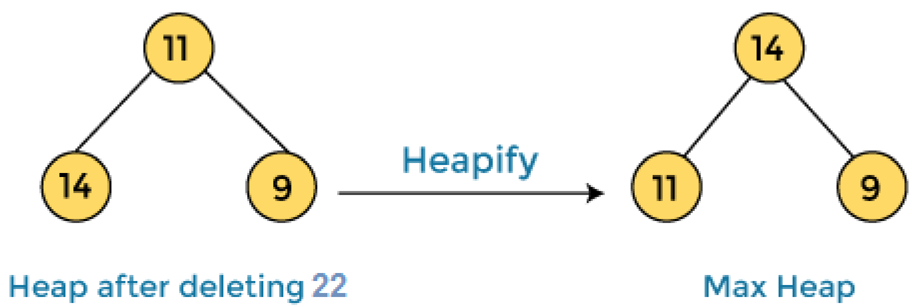
In the next step, again we have to delete the root element **(54)**from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



Afterswappingthearrayelement**54**with**14**andconvertingtheheapintomax-heap,theelementsofarrayare-

22	14	9	11	54	76	81	89

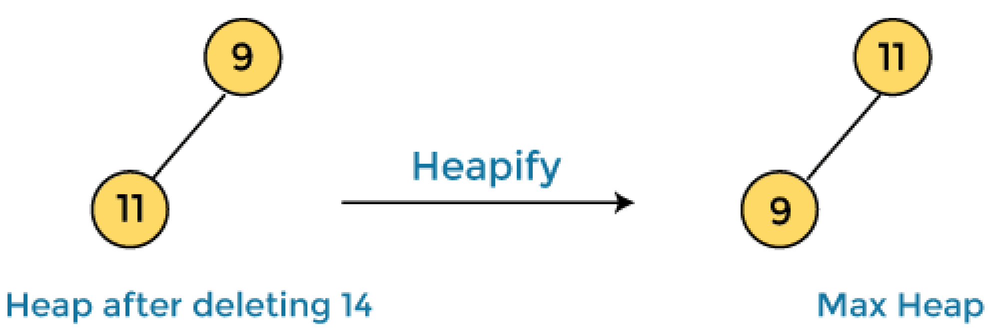
In the next step, again we have to delete the root element **(22)**from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



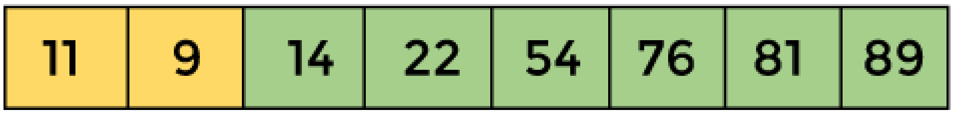
Afterswappingthearrayelement**22**with**11**andconvertingtheheapintomax-heap,theelementsofarrayare-

14	11	9	22	54	76	81	89

In the next step, again we have to delete the root element **(14)**from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

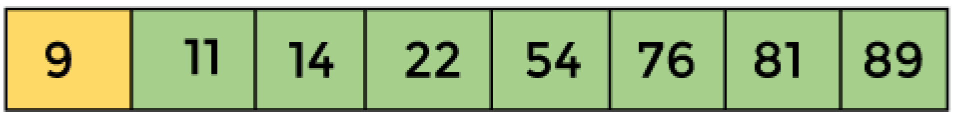


Afterswappingthearrayelement**14**with**9**andconvertingtheheapintomax-heap,theelementsofarrayare-



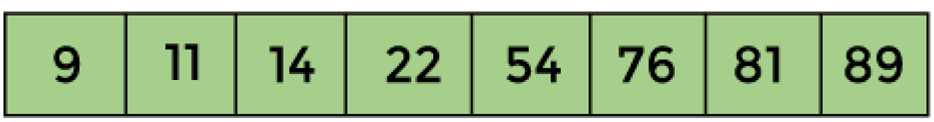
In the next step, again we have to delete the root element **(11)**from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

Afterswappingthearrayelement **11**with**9,**theelementsofarrayare-



Now,heaphasonlyoneelementleft.Afterdeletingit,heapwillbeempty.

Aftercompletionofsorting,thearrayelementsare-



Now,thearrayiscompletelysorted.

Program: WriteaprogramtoimplementheapsortinClanguage.

```
#include <stdio.h>
/*function toheapifyasubtree.Here'i'isthe
indexofrootnodeinarraya[],and'n'isthesizeofheap.*/ void
heapify(int a[], int n, int i)
{
   intlargest=i;//Initializelargestasroot int
   left = 2 * i + 1; // left child
   intright=2*i+2;//rightchild
   //Ifleftchildislargerthan root
   if(left<n&&a[left]>a[largest])
```

```
        largest=left;
    //Ifrightchildislargerthanroot
    if(right<n&&a[right]>a[largest]) largest =
        right;
    //Ifrootisnotlargest if
    (largest != i) {
        //swapa[i]witha[largest] int
        temp = a[i];
        a[i] = a[largest];
        a[largest]=temp;

        heapify(a,n,largest);
    }
}
/*Functiontoimplementtheheapsort*/ void
heapSort(int a[], int n)
{
    for(inti=n/2-1;i>=0;i--) heapify(a, n,
        i);
    //Onebyoneextractanelementfromheap for
    (int i = n - 1; i >= 0; i--) {
        /*Movecurrentrootelementtoend*/
        //swapa[0]witha[i]
        int temp = a[0];
        a[0] = a[i];
        a[i]=temp;

        heapify(a,i,0);
    }
}
/*functiontoprintthearrayelements*/ void
printArr(int arr[], int n)
{
    for(int i=0;i<n;++i)
    {
        printf("%d",arr[i]);
        printf("");
    }

}
intmain()
{
    inta[]={48,10,23,43,28,26, 1};
    intn =sizeof(a)/ sizeof(a[0]);
    printf("Beforesortingarrayelementsare-\n"); printArr(a,
    n);
```

```
    heapSort(a,n);
    printf("\nAftersortingarrayelementsare-\n"); printArr(a,
    n);
    return0;
}
```

Output

```
Before sorting array elements are -
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48
```

MergeSort Algorithm

In this article, we will discuss the merge sort Algorithm. Merge sort is the sorting technique that follows the divide and conquerapproach.Thisarticlewillbeveryhelpfulandinterestingtostudentsastheymightfacemergesortasaquestionin their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

Merge sort issimilar to the quick sort algorithm as it uses the divide andconquer approach to sort the elements. It isoneof the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the twohalves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

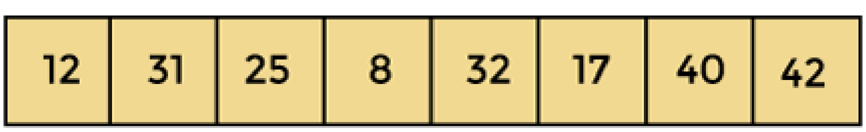
The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

WorkingofMergesortAlgorithm

Now,let'sseetheworkingofmergesortAlgorithm.

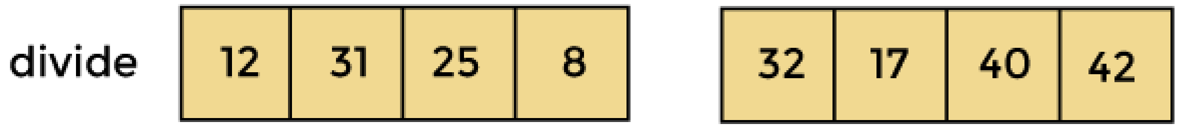
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Lettheelementsofarrayare-

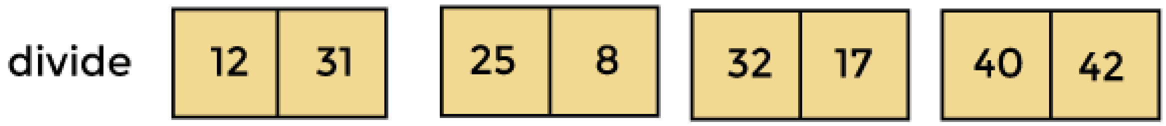


According to themergesort,firstdividethe given array intotwo equalhalves.Mergesortkeepsdividing thelistinto equal parts until it cannot be further divided.

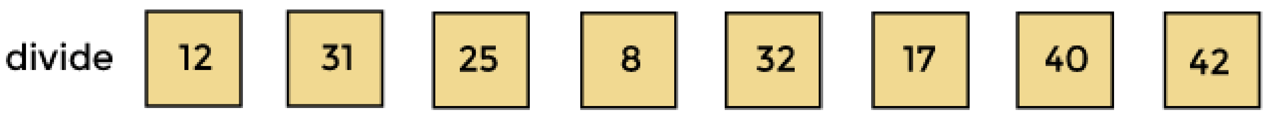
Astherearee ightelementsinthegivenarray,soitisdividedintotwoarraysofsize4.



Now,againdividethesetwoarraysintohalves.Astheyareofsize4,sodividethemintonewarraysofsize2.



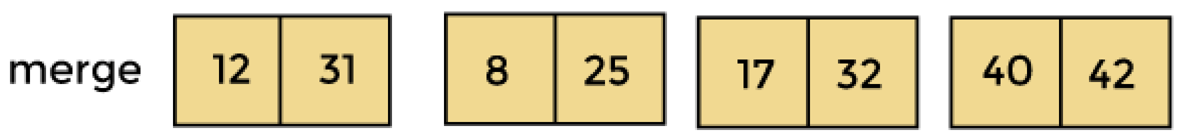
Now,againdividethesearraystogettheatomicvaluethatcannotbefurtherdivided.



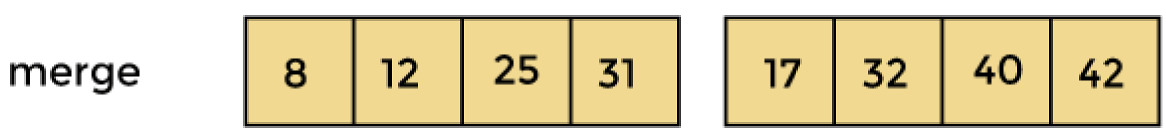
Now,combinetheminthesamemannertheywerebroken.

Incombining,firstcomparetheelementofeacharrayandthencombinethemintoanotherarrayinsortedorder.

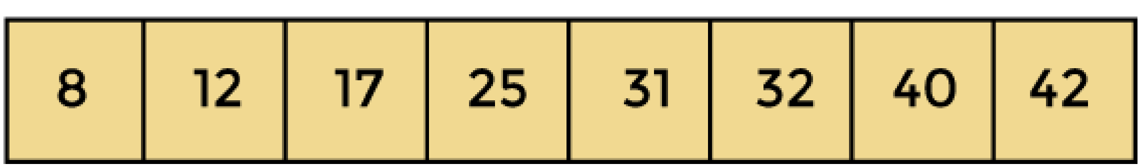
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now,thereisafinalmergingofthearrays.Afterthefinalmergingofabovearrays,thearraywilllooklike-



Now,thearrayiscompletelysorted.

```
#include<stdio.h>

/*Functiontomergethesubarraysofa[]*/
voidmerge(inta[],intbeg,intmid,intend)
{
    inti,j,k;
    intn1=mid-beg +1;
    intn2=end-mid;

    intLeftArray[n1],RightArray[n2];//temporaryarrays

    /*copydatatotemparrays*/ for
    (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j]=a[mid+1+j];

    i=0;/*initialindexoffirstsub-array*/
    j=0;/*initialindexofsecondsub-array*/
    k=beg;/*initialindexofmerged sub-array*/

    while(i<n1&&j<n2)
    {
        if(LeftArray[i]<=RightArray[j])
        {
            a[k]=LeftArray[i];
            i++;
        }
        else
        {
            a[k]=RightArray[j];
            j++;
        }
        k++;
    }
    while(i<n1)
    {
        a[k]=LeftArray[i];
        i++;
        k++;
    }

    while(j<n2)
    {
        a[k]=RightArray[j];
        j++;
```

```
            k++;
        }
    }

    voidmergeSort(inta[],intbeg,intend)
    {
        if(beg<end)
        {
            int mid = (beg + end) / 2;
            mergeSort(a, beg, mid);
            mergeSort(a,mid+1,end);
            merge(a, beg, mid, end);
        }
    }

    /*Functiontoprintthearray*/
    voidprintArray(inta[],intn)
    {
        inti;
        for(i=0;i<n;i++)
            printf("%d",a[i]);
        printf("\n");
    }

    intmain()
    {
        inta[]={12,31,25,8,32,17,40, 42};
        intn=sizeof(a)/sizeof(a[0]);
        printf("Beforesortingarrayelementsare-\n");
        printArray(a, n);
        mergeSort(a,0,n-1);
        printf("Aftersortingarrayelementsare-\n");
        printArray(a, n);
        return0;
    }
```

Output:

```
Before sorting array elements are -
12 31 25 8 32 17 40 42
After sorting array elements are -
8 12 17 25 31 32 40 42
```

MODULE-5

Patternmatchingalgorithms:

A pattern matching algorithm is used to determine the index positions where a given pattern string (P) is matchedin a text string (T). It returns `"pattern not found"`if the pattern does not match in the text string. For example, for the given string (s) = `"packt publisher"`, and the pattern (p)= `"publisher"`, the pattern matching algorithm returns the index position wherethe pattern is matched in the text string.

In this section, we will discuss two pattern matching algorithms, that is, the brute-force method, as well as Knuth-Morris-Pratt (KMP).

BruteForceApproach:

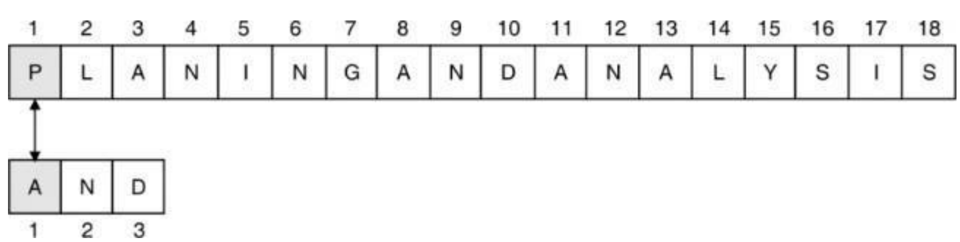
Bruteforceapproachcanalso becalledasexhaustivesearch. Basically bruteforcemeans you go through all the possible solutions.

It isone ofthe easiest wayto solve aproblem. But intermsoftimeand spacecomplexity will take a hit.

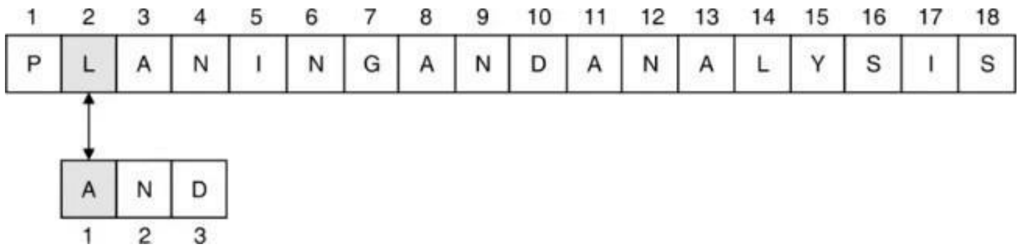
WorkingMechanism

- This is simple and efficient brute force approach. It compares the firstcharacter of pattern with searchable text. If a match is found, pointers in both strings are advanced. If a match is not found, the pointer to text is incremented and pointer of the pattern is reset. This process is repeated till the end of the text.
- The naïve approach does not require any pre-processing. Given text Tand pattern P, it directly starts comparing both strings character by character.
- Aftereachcomparison,itshiftspatternstring*oneposition*totheright.
- Followingexampleillustratestheworkingofnaïvestringmatchingalgorithm.Here,T = PLANINGANDANALYASIS and P = AND
- Here,$t_i$and$p_j$areindicesoftextandpatternrespectively.

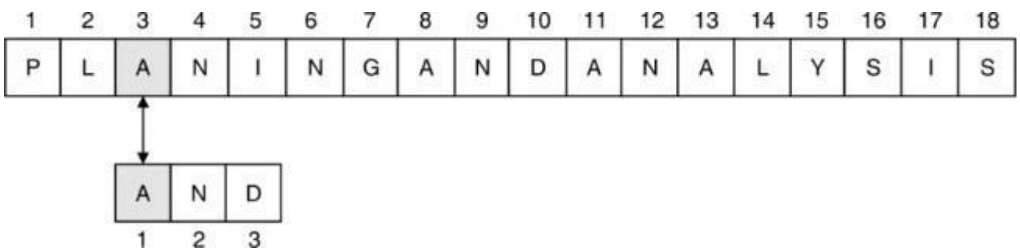
Step1:T[1]≠P[1],soadvancetextpointer,i.e.t_i++.



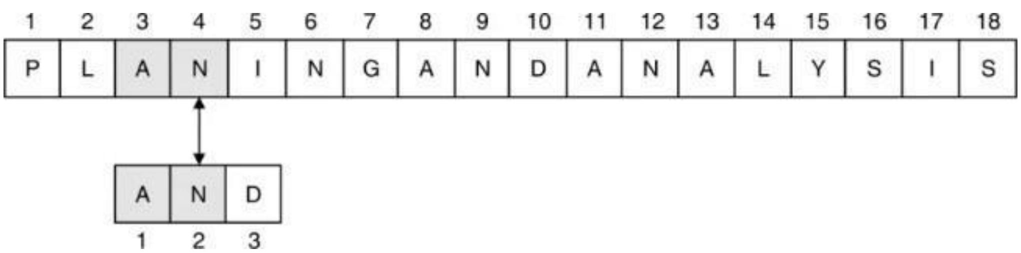
Step2:T[2]≠P[1], soadvancetextpointersi.e. t_i++



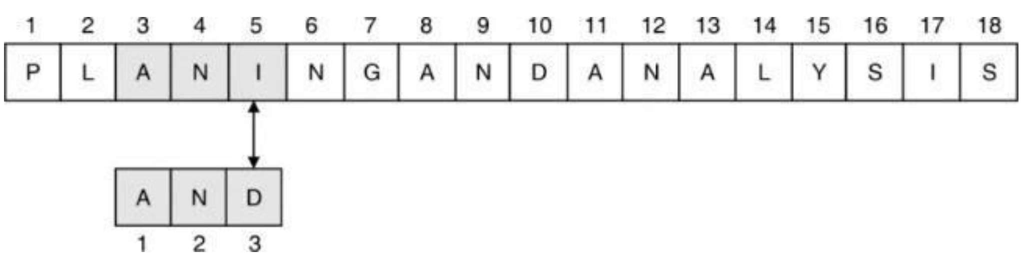
Step3:T[3] =P[1],soadvancebothpointersi.e.t_i++, p_j++



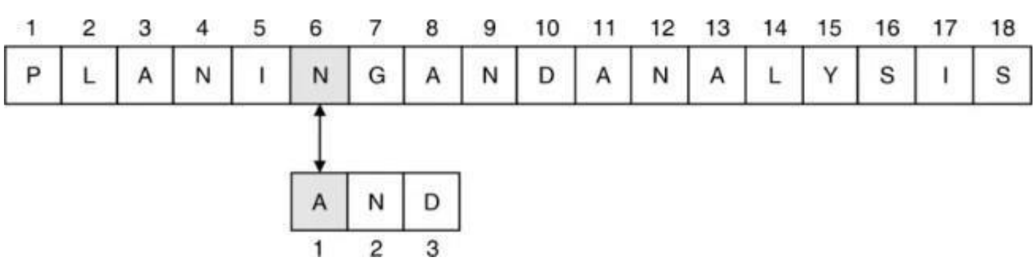
Step4:T[4] =P[2], soadvancebothpointers,i.e.t_i++, p_j++



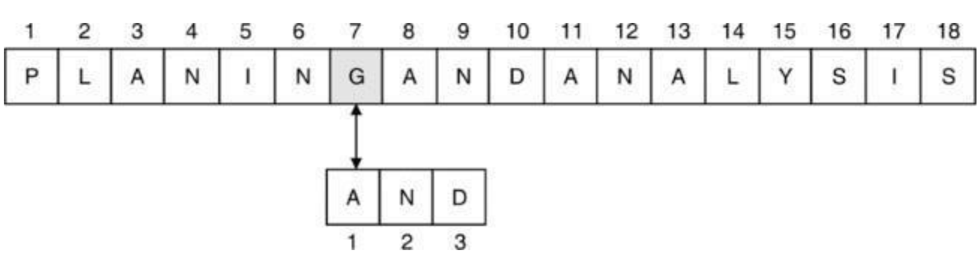
Step5:T[5]≠P[3],soadvancetextpointerand resetpatternpointer,i.e.t_i++,p_j=1



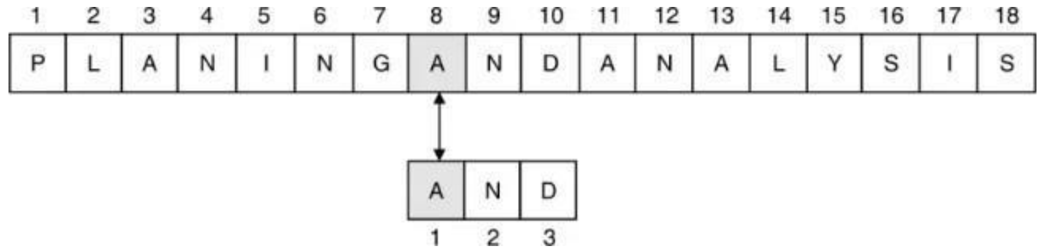
Step6:T[6] ≠P[1],soadvancetextpointer,i.e. t_i++



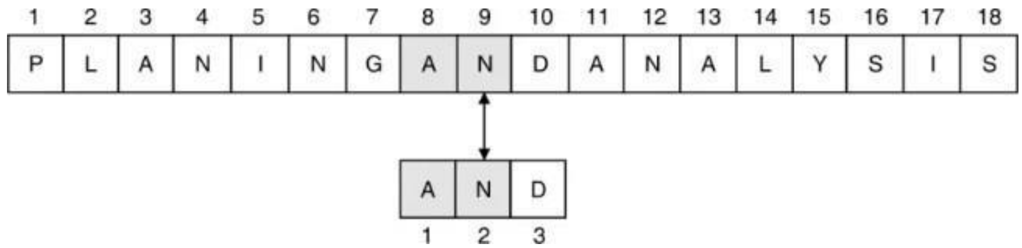
Step7:T[7]≠P[1],soadvancetextpointeri.e. t_i++



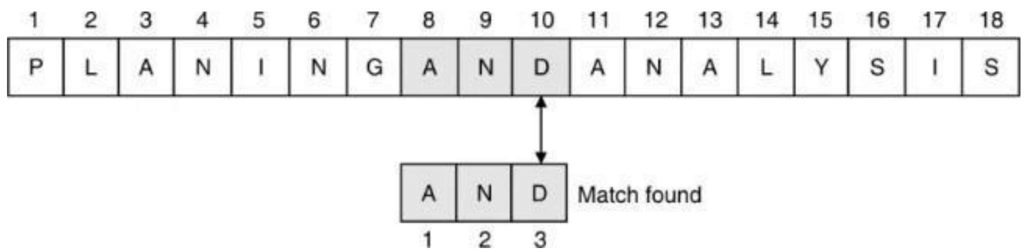
Step8:T[8] =P[1], soadvancebothpointers,i.e.t_i++, p_j++



Step 9:T[9]=P[2],soadvancebothpointers,i.e.t_i++, p_j++



Step 10:T[10] =P[3], soadvancebothpointers,i.e.t_i++,p_j++



Algorithm

Algorithmfornaïvestringmatchingapproachisdescribedbelow:

```
Algorithm  NAÏVE_STRING_MATCHING(T, P)
// T is the text string of length n
// P is the pattern of length m

for  i ← 0 to n – m  do
    if P[1… m] == T[i+1…i+m]  them
        print  "Match Found"
    end
end
```

Knuth-Morris-Pratt(KMP)Algorithm:

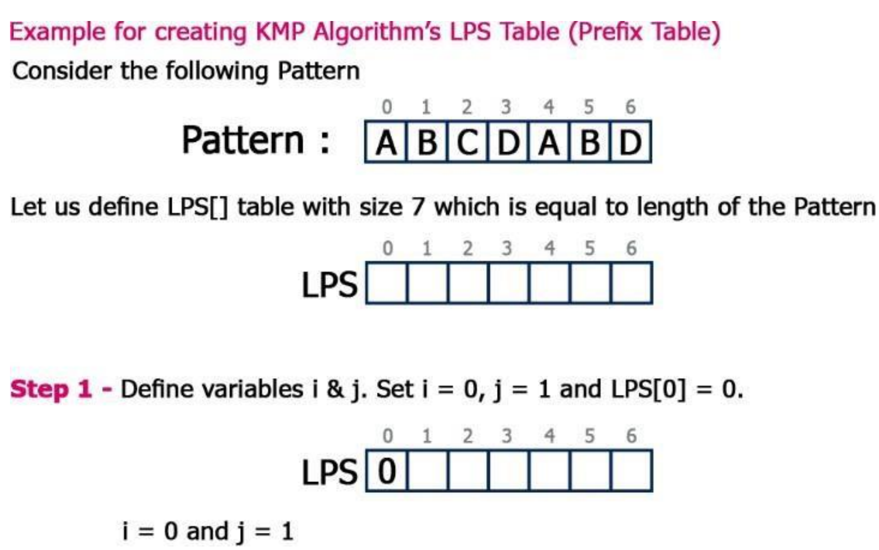
KMPAlgorithmisoneofthemost popularpatternsmatching algorithms.KMPstandsfor KnuthMorrisPratt.KMPalgorithmwasinventedby **DonaldKnuth** and **VaughanPratt** together and independently by **James H Morris** in the year 1970. In the year 1977, all the three jointly published KMP Algorithm.

KMP algorithm is used to find a **"Pattern"**in a **"Text"**. This algorithm campares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called **"Prefix Table"**to skip characters comparison while matching. Some times prefix table is also known as **LPS Table**. Here LPS stands for **"Longest proper Prefix which is also Suffix"**.

StepsforCreating LPSTable (PrefixTable):

- **Step 1 -** Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])
- **Step2-** Definevariables **i &j**.Seti=0,j=1 andLPS[0]=0.
- **Step3-**Comparethecharactersat**Pattern[i]**and**Pattern[j].**
- **Step 4 -** If both are matched then set **LPS[j] = i+1** and incrementboth i & j values by one. Goto to Step 3.
- **Step5-**Ifbotharenotmatchedthencheckthevalueofvariable'i'.Ifitis'0'then set **LPS[j] = 0** and increment 'j' value by one, if it is not '0' then set **i = LPS[i-1]**. Goto Step 3.
- **Step6-**Repeat abovestepsuntilallthevaluesofLPS[]arefilled.

Letususeabovestepstocreateprefixtablefor apattern:



Step 2 - Campare Pattern[i] with Pattern[j] ===> A with B.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0					

i = 0 and j = 2

Step 3 - Campare Pattern[i] with Pattern[j] ===> A with C.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0				

i = 0 and j = 3

Step 4 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0			

i = 0 and j = 4

Step 5 - Campare Pattern[i] with Pattern[j] ===> A with A.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1		

i = 1 and j = 5

Step 6 - Campare Pattern[i] with Pattern[j] ===> B with B.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 2 and j = 6

Step 7 - Campare Pattern[i] with Pattern[j] ===> C with D.
Since both are not matching and i !=0, we need to set i = LPS[i-1]
===> i = LPS[2-1] = LPS[1] = 0.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 0 and j = 6

Step 8 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...

	0	1	2	3	4	5	6
LPS	**0**	**0**	**0**	**0**	**1**	**2**	**0**

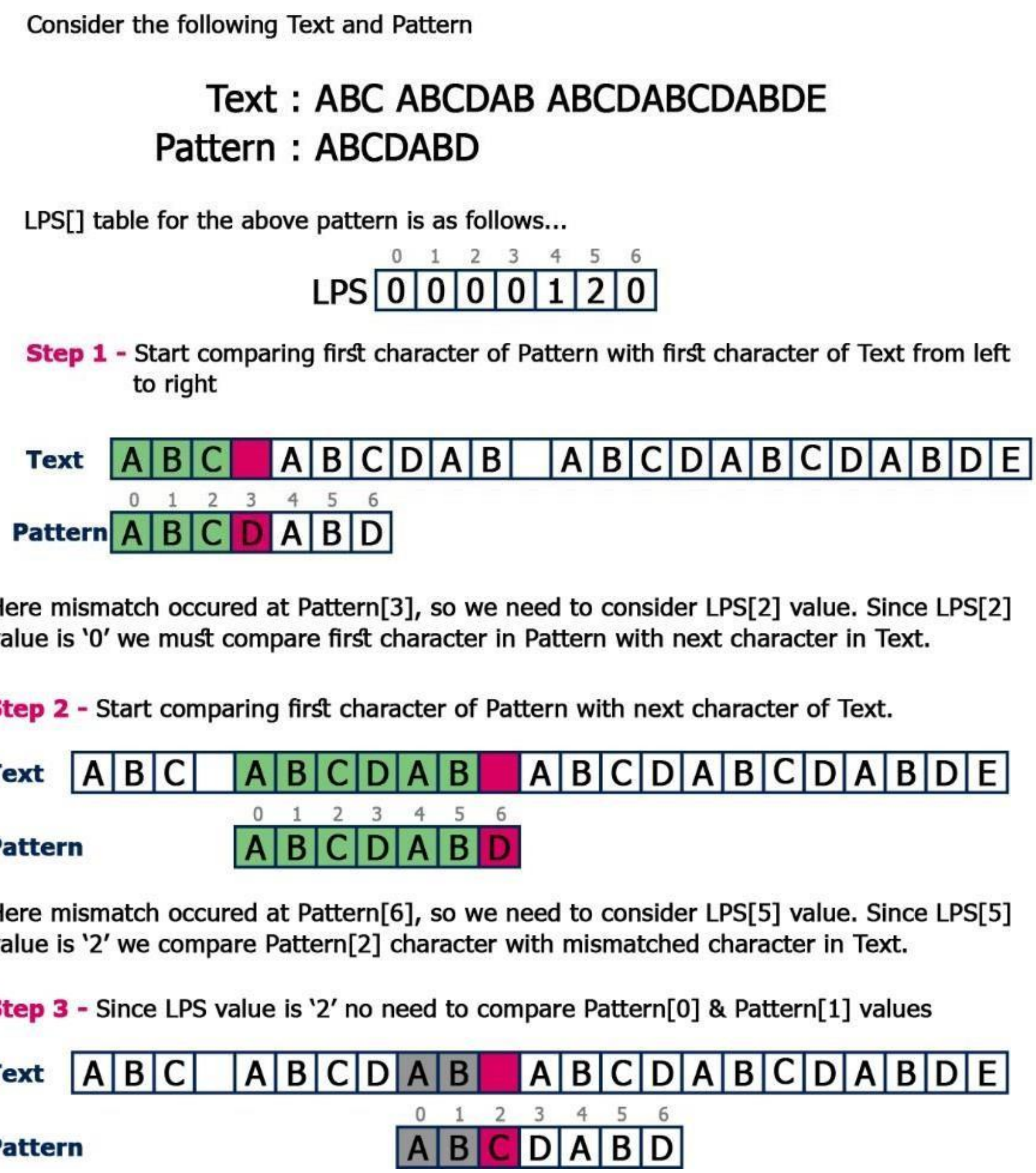
HowtouseLPS Table

WeusetheLPStabletodecidehow manycharactersareto beskipped forcomparisonwhena mismatch has occurred.

When a mismatch occurs, check the LPS value of the previous character of the mismatched character inthepattern. Ifit is'0'thenstartcomparingthefirst characterofthepatternwiththe next character to the mismatched character in the text. If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

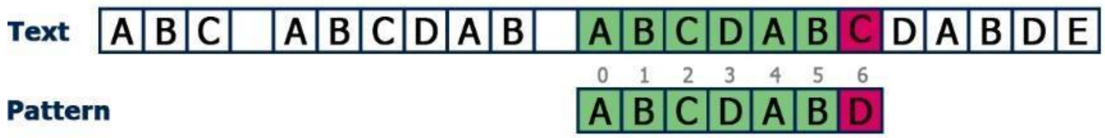
How theKMPAlgorithm Works

LetusseeaworkingexampleofKMPAlgorithmtofind aPatterninaText...



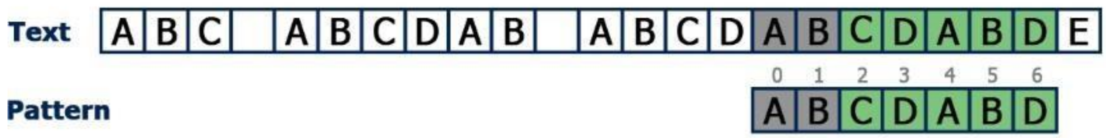
Here mismatch occured at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

TheBoyer-MooreAlgorithm

Robert Boyer and J Strother Moore established it in 1977. The B-M String search algorithm is a particularly efficient algorithm and has served as a standard benchmark for string search algorithmever since.

The B-M algorithm takes a 'backward' approach: the pattern string (P) is aligned with the start of the text string (T), and then compares the characters of a pattern from right to left, beginning with rightmost character.

If a character is compared that is not within the pattern, no match can be found by analyzing any further aspects at this position so the pattern can be changed entirely past the mismatching character.

For deciding the possible shifts, B-M algorithm uses two preprocessing strategies simultaneously. Whenever a mismatch occurs, the algorithm calculates a variation using both approaches and selects the more significant shift thus, if make use of the most effective strategy for each case.

Thetwostrategiesarecalledheuristics ofB-Mastheyareused toreducethesearch.Theyare:

1. BadCharacter Heuristics
2. GoodSuffix Heuristics

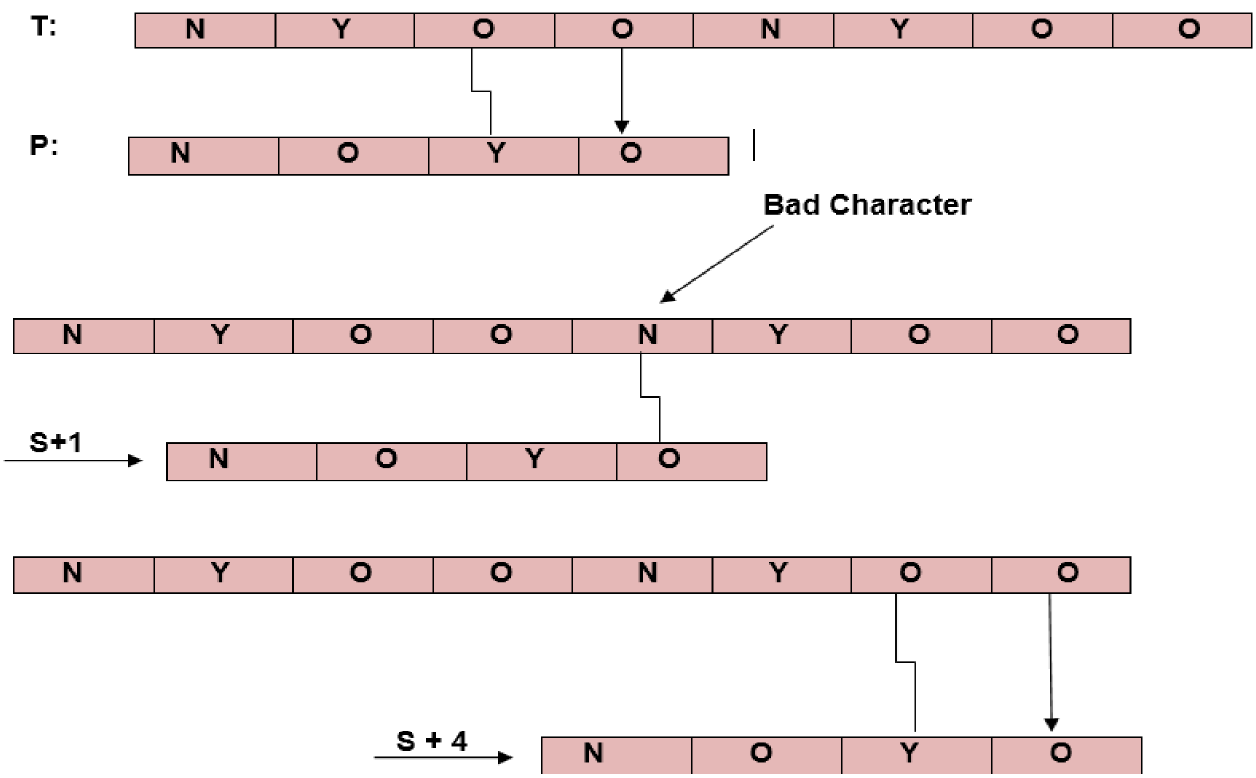
1. BadCharacter Heuristics

ThisHeuristicshastwo implications:

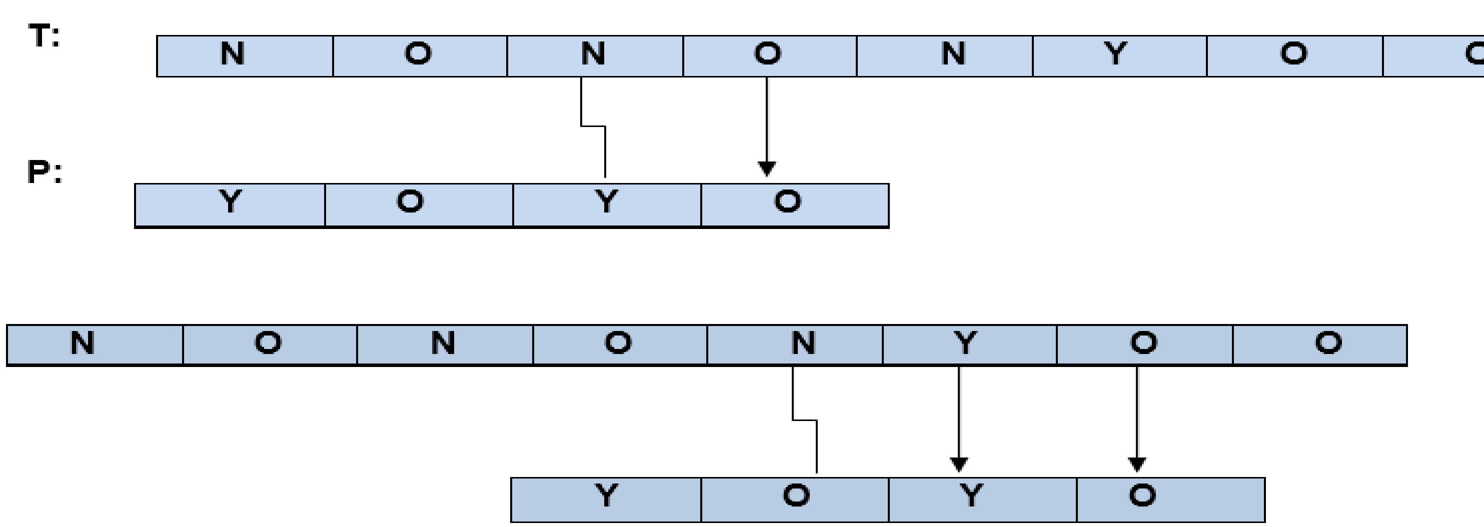
- Suppose there is a character in a text in which does not occur in a pattern at all. When a mismatch happens at thischaracter (called as bad character), the whole pattern can be changed, begin matching form substring next to this 'bad character.'
- On the other hand, it might be that a bad character is present in the pattern, in this case, align the nature of the pattern with a bad character in the text.

Thusin any caseshiftmay behigherthan one.

Example1: LetText T=<nyoonyoo>andpatternP=<noyo>



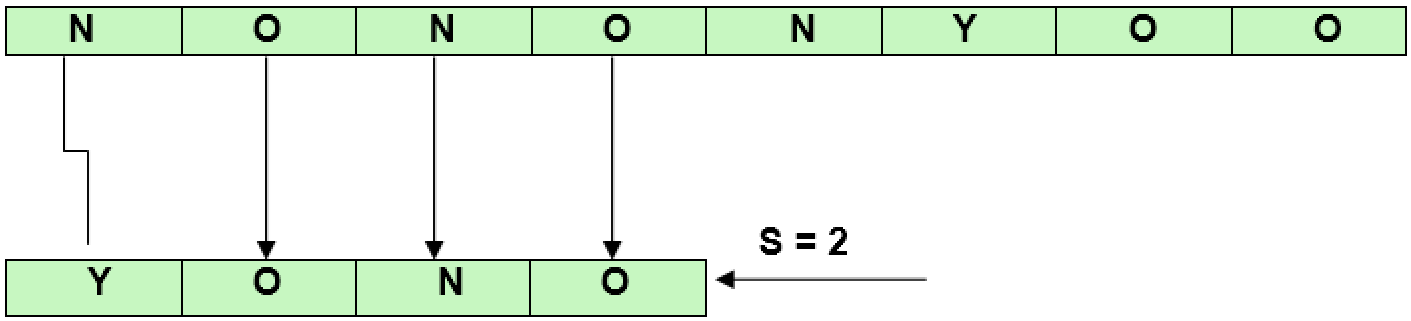
Example2: Ifabad characterdoesn't exist the pattern then.



ProbleminBad-Character Heuristics:

Insomecases,Bad-CharacterHeuristicsproducessomenegativeshifts. For

Example:



This means that we need some extra information to produce a shift on encountering a bad character. This information is about the last position of every aspect in the pattern and also the set of characters used in a pattern (often called the alphabet ∑of a pattern).

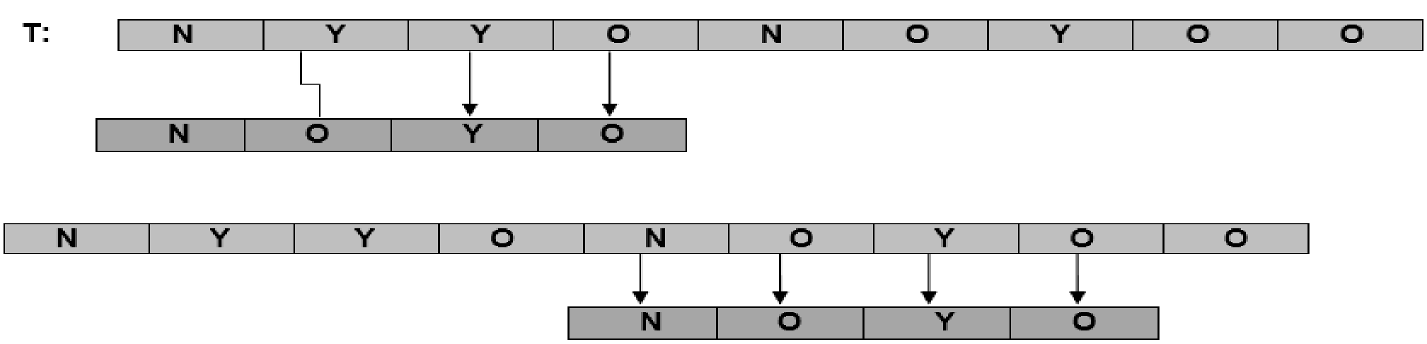
COMPUTE-LAST-OCCURRENCE-FUNCTION(P,m,∑)

1. foreachcharactera∈∑
2. doλ [a] =0
3. forj ← 1 to m
4. doλ[P[j]]← j
5. Returnλ

2. Good SuffixHeuristics:

A good suffix is a suffix that has matched successfully. After a mismatch which has a negative shift in bad characterheuristics, look ifasubstringofpatternmatched till bad character has agood suffixin it, if it is so then we have an onward jump equal to the length of suffix found.

Example:



TRIES

A trie is a tree-like information retrieval data structure whose nodes store the letters of an alphabet. It is also known as a digital tree or a radix tree or prefix tree. Tries are classified into three categories:

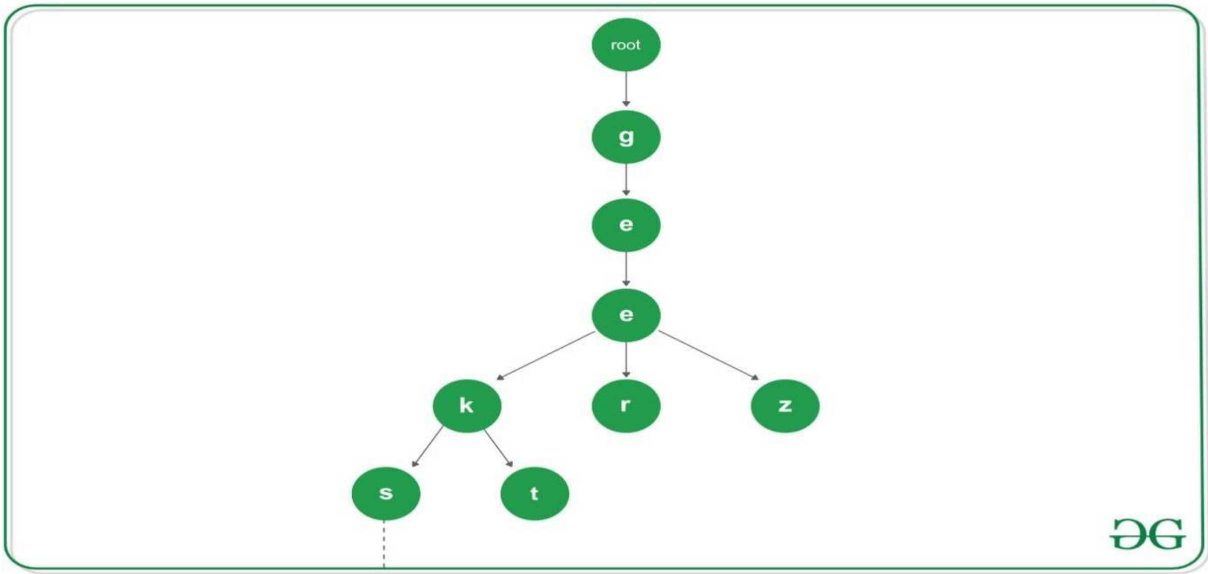
1. StandardTrie
2. CompressedTrie
3. SuffixTrie

Standard TrieA standard trie have the following properties:

A Standard Trie has the below structure:

```
classNode{


    // Array to store the nodes of a tree

    Node[] children = new Node[26];


    // To check for end of string

    boolean isWordEnd;

}
```

- Itisanorderedtreelikedatastructure.
- Eachnode(excepttheroot**root**node)inastandardtrieislabeledwithacharacter.
- Thechildrenofanodeareinalphabeticalorder.
- Eachnodeorbranchrepresentsapossiblecharacterofkeysorwords.
- Eachnodeorbranchmayhavemultiplebranches.
- Thelastnodeofeverykeyorwordisusedtomarktheendofwordornode.
- BelowistheillustrationoftheStandardTrie:



Compressed TrieA Compressed trie have the following properties:

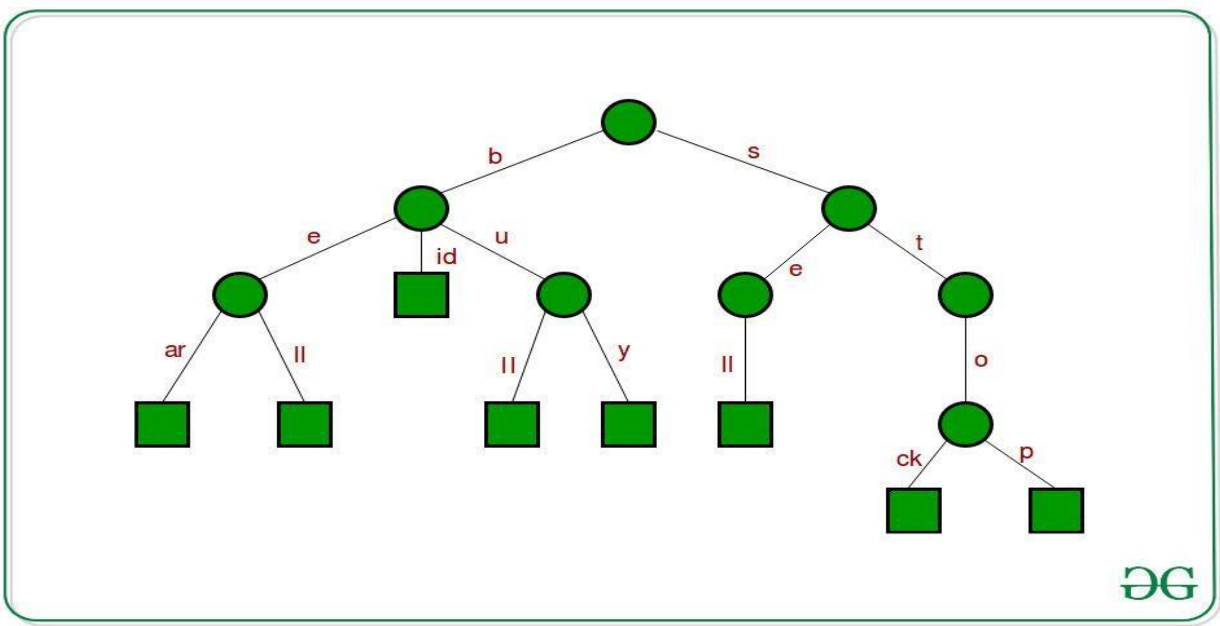
A Compressed Trie has the below structure:

```
classNode{


    // Array to store the nodes of tree
```

```
Node[] children = new Node[26];
```

```
//TostoretheedgeLabel

StringBuilder[]edgeLabel=newStringBuilder[26];


// To check for end of string boolean

isEnd;

}
```

- ACompressedTrieisanadvancedversionofthestandardtrie.
- Eachnodes(excepttheleafnodes)haveatleast2children.
- Itisusedtoachievespaceoptimization.
- To derive a Compressed Trie from a Standard Trie, compression of chains of redundant nodes is performed.
- Itconsistsofgrouping,re-groupingandun-groupingofkeysofcharacters.
- While performing the insertion operation, it may be required to un-group the already grouped characters.
- While performing the deletion operation, it may be required to re-group the already grouped characters.
- A compressed trie T storing s strings(**keys**) has s external nodes and O(s) total number of nodes.
- BelowistheillustrationoftheCompressedTrie:



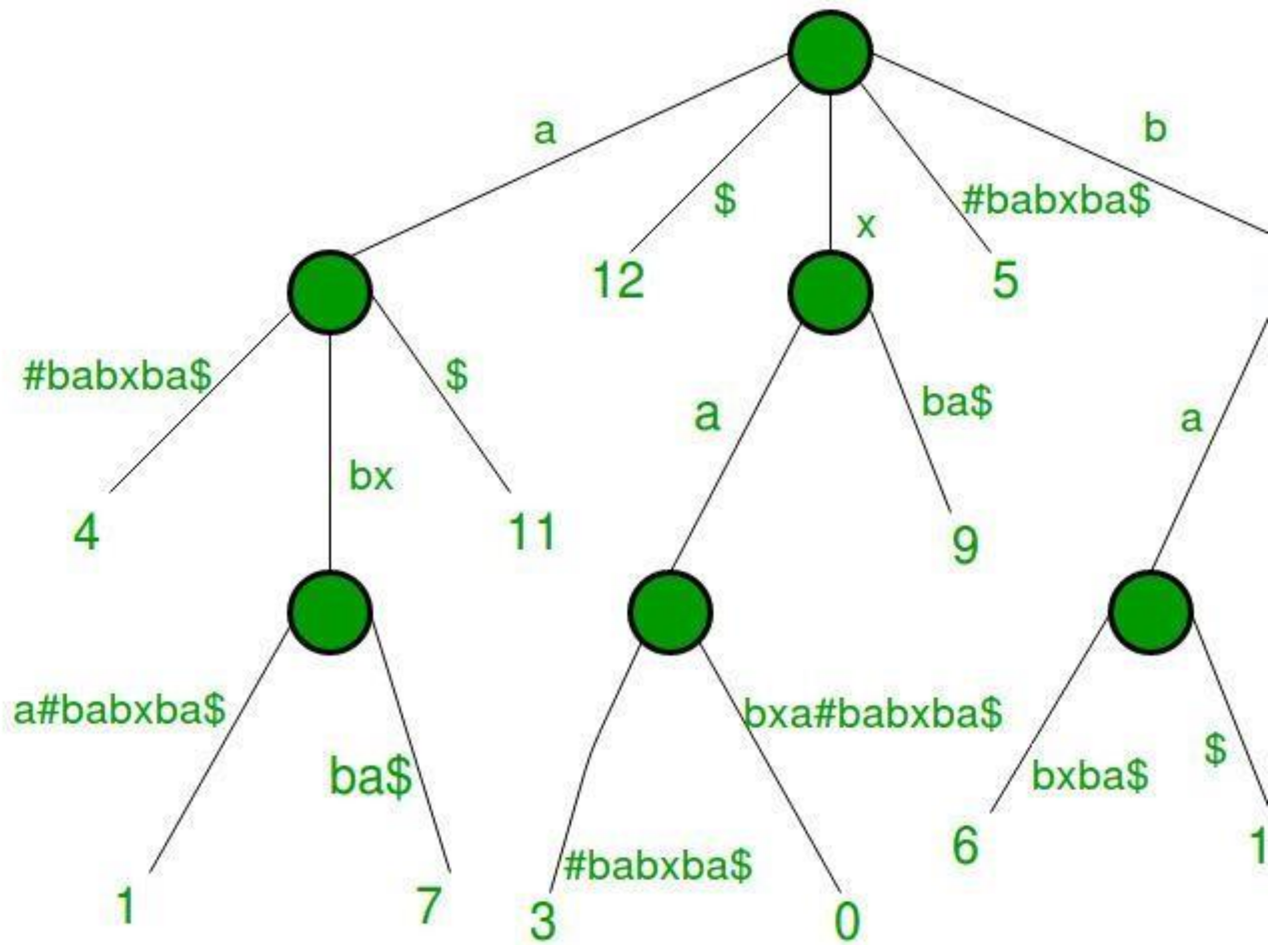
Suffix TrieA Suffix trie have the following properties:
A Compressed Trie has the below structure:

```
structSuffixTreeNode{


    //Arraytostorethenodes

    structSuffixTreeNode*children[256];


    //pointer to other node via suffix link

    struct SuffixTreeNode *suffixLink;
```

```
//(start,end)intervalspecifiestheedge,
```

```
    //bywhichthenodeisconnectedtoits
    // parent node
    int start;
    int*end;

    //Forleafnodes, itstorestheindexof
    // Suffix for the pathfrom root to leaf int
    suffixIndex;
}
```

- ASuffixTrieisanadvancedversionofthecompressedtrie.
- ThemostcommonapplicationofsuffixtrieisPatternMatching.
- Whileperformingtheinsertionoperation,boththewordanditssuffixesarestored.
- Asuffixtrieisalsousedinwordmatchingandprefixmatching.
- Togenerateasuffixtrie,allthesuffixesofgivenstringareconsideredasindividualwords.
- Usingthesuffixes,compressedtrieisbuilt.
- BelowistheillustrationoftheSuffixTrie:



Suffix tree for string xabxa#babxa$