A complete note on,

Data Structure and Algorithms(CT552)

Along with the problems and solutions

By: Er. Bishwas Pokharel(Lecturer)
B.E- Pulchowk campus
MSc.Engg. -Pulchowk Campus
Published On: 2019/01/05

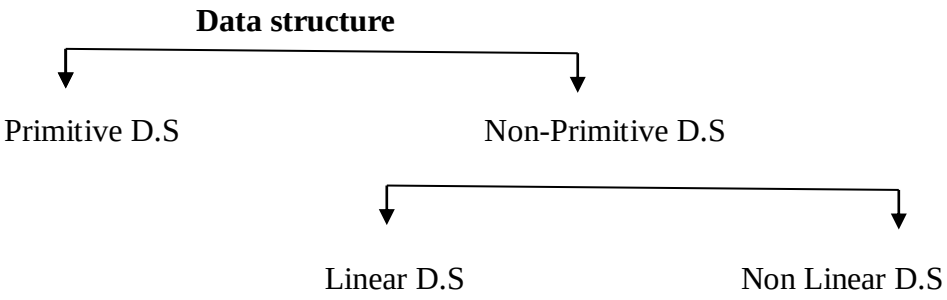
Chapter 1: Concept of Data Structure

(It is better to read chapter 1 after understanding stack ,queue, linked list)

Q. Define Data Structure and its types

Data structure is a particular way of organizing and storing data in a computer so that data it can be used efficiently, in term of time and space. Organization of data takes place either in main memory or in disk storage.

Algorithm+ Data Structure=Program



Primitive D.S: These data structures are defined by system with their operation like add, sub, multiply etc. Some primitive data structures used in general programming languages are char, int, float, double etc.

Non Primitive D.S: These data structures are formed by the collection of primitive data structures and for implementation it must be declared with their operation.

a. **Linear:** stack, queue, linked list etc.
b. **Non Linear**: tree, graph, hash map etc.

Q. Explain the basic data structure operations

a. Traversing: It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class

b. Searching: It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in the mathematics.

c. Inserting: It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

d. Deleting: It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course

e. Sorting: Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

f. Merging: Lists of two sorted data items can be combined to form a single list of sorted data items.

Q. Define ABSTRACT DATA TYPES:

Before defining the abstract data types, first break it into word "ABSTRACT" AND "DATA TYPE"

.

DATA TYPE: of the variable defines the set the values that the variable can take. For example integer is the data type which if signed 32 bit then value from -2,147,483,648 to 2,147,483,647 and can be operated with operators +,-,*,/ .

ABSTRACT: means hiding the implementation details and providing the function that can work. It is for the simplicity.

So, abstract data type can be a structure where user is not concerned with how the function process the task, how data structure are organized and implemented. Knowing the available operation is sufficient. To simplify the process of solving problems, these data structures are combined with their operation. An ADT consist of two parts:

1. Declaration of data 2.Declaration of operations

Commonly used ADT are: list, stack and queue.

Example: STACK

Declaration of data: Stack is Linear Data Structures with LIFO (Last-In-First-Out) mechanism where LIFO means the last inserted element is the first element to get deleted.

Declaration of operations: creating stack, pushing an element onto the stack, popping an element from the stack, find the current top of the stack etc.

In conclusion, They should just know that to work with stacks, they have push() and pop() functions available to them. Using these functions, they can manipulate the data (insertion or deletion) stored in the stack.

Q. Write any data structure as an ADT.

Stack as ADT

A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:

a. push() – Insert an element at one end of the stack called top.

b. pop() – Remove and return the element at the top of the stack, if it is not empty.

c. peek() – Return the element at the top of the stack without removing it, if the stack is not empty.

d. size() – Return the number of elements in the stack.

e. isEmpty() – Return true if the stack is empty, otherwise return false.

f. isFull() – Return true if the stack is full, otherwise return false.

Q. Mention the Applications of data structure:

1. In, Computer Network: Most of the cable network companies use the Disjoint Set Union data structure in Kruskal's algorithm to find the shortest path to lay cables across a city or group of cities.

2. In, Large Database Management System: creating your own database just to store the data based upon some key value, in Banks for combining two or more accounts for matching Social Security Numbers.

3. In, Gaming: Consider graphs for example, imagine you are using Google Maps to travel from your home to office and you want to reach your office in the shortest path possible, There comes in graphs to find the shortest path using an algorithm.

4. In, Operating System: to run various processes which enters in the FIFO manner (first in first out) i.e. whichever process enters first will move out first. To clarify it, assume you ask your system to do the following: play music, open browser, open paint then these 3 will be take as different process and will be done in FIFO manner.

5. In, Search Engines (Google, face book …): The web crawlers in a Google search that gives you the best results at the top using the breadth first search traversal of a graph etc.

Q. Define an Algorithm with its example.

Definition: Algorithm is a step by step instruction to solve the problems.

Example:

What are the steps you follow for preparing for omelet?
For preparing omelet the steps we follow are as follows:

1. Turn on the stove.

2. Get the frying pan.

3. Get the oil.

    a. Do you have oil?
        a. If yes, put it in a pan.

        b. If no, we can terminate.
4. Etc etc..

So, what we are doing is for the given problem (preparing an omelet), giving step by step procedure for solving it.

Q. Why analysis of an algorithms?

To go from say, city Kathmandu to Biratnagar, there can be many ways of accomplish this: by flight, by bus, by motorcycle, by cycle, by walking and the convenience we choose the one which suits us based upon time, money, interest, urgency etc.

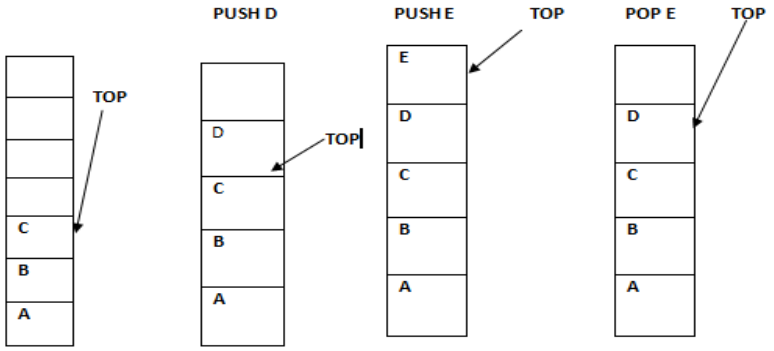
Similarly, in computer science to solve a particular problem there are many algorithms (like insertion, merge, radix etc). So algorithm analysis helps us to determining which of them is efficient in terms of time and space consumed.

So goal of analysis of an algorithm is to compare algorithms (or solution) mainly in terms of running times but also in terms of other factors (memory, developer effort etc).

STACK AND QUEUE

2.1 What is stack? Explain its operation in details.

Stack is the linear data structure which follows the **Last in First out Order (LIFO)**. The last element that inserted is the one which removed first. When an element is inserted in a stack, the concept is called as **push** and when an element is removed from the stack, the concept is called as **pop.** Trying to pop out an empty stack is called as **underflow** and trying to push an element in a full stack is called as **overflow.** The pointer which tracks the topmost element of stack is known as **top pointer.**



Imagine an example to understand the stack concept,

1. Consider an example of washing the plates in restaurant, the plate which is washed first is placed at the bottom and sequentially pushed one above other, the last one washed is placed at the top. But when plate has to be used, then the last one placed is pop out first and so on until there remains plate.

2. Create Stack:

*****Stack can be created by declaring with two members.
*One Member can store the actual data in the form of array.
*Another Member can store the position of the topmost element.

3. **Push Operations:** The process of putting a new data element onto stack is known as push operation. Push operation involves a series of steps:

Step 1-Checks if the stack is full.
Step 2-If the stack is full, produces stack overflow and exit.
Step 3-If the stack is not full, increments top to point next empty space.
Step 4-Add data element to the stack location, where top is pointing.
Step 5-Return success.

a. If top=max-1 then write "stack overflow and stop"
b. Read data from user
c. top-<top+1

d. **stack[top]<-data**

e. **stop**

4. **Pop Operations:** The process of extracting a data element from stack is known as pop operation. Pop operation involves a series of steps:

Step 1-Checks if the stack is empty.
Step 2-If the stack is empty, display stack underflow and exit.
Step 3-If the stack is not empty, access the elements at which the top is pointing.
Step 4-Decrease the value of top by 1
Step 5- Return success

a. **If top<0 then write "stack underflow and stop"**

b. **Return Stack[top]**

c. **top<-top-1**

d. **stop**

Note: you can use a single linked list for the implementation and then you can indeed reduce the size of the stack when doing a pop and not just "replace" the popped element with a dummy value or create a new smaller stack and copy the rest there.(Ref: stack overflow)

2.2 What are the applications of the stack?

Applications of stack:

1. Balancing of symbols

2. Infix to Postfix /Prefix conversion

3. Redo-undo features at many places like editors, Photoshop.

4. Forward and backward feature in web browsers

5. Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.

2.3 Infix, Postfix and Prefix notation

Infix:

An infix expression take, only one single letter or it has two letters with operators (+,-,*,/)in between them or complete two infix expression with operators (+, -,*,/) in between them.
A -> single letter

A+B -> two letters with one operator + in between them.
(A+B)+(C-D) -> Two infix expression with one operator + in between them.

Prefix:

An prefix expression take, only one single letter or it has two letters in sequence with (+,-,*,/)before them or complete two prefix expression with operators (+, -,*,/) in between them.
A -> single letter
++AB -> two letters with one operator ++ before them.
++AB-CD -> Two prefixes expression.

Postfix: An postfix expression take, only one single letter or it has two letters in sequence with (+,-,*,/)after them or complete two postfix expression with operators (+, -,*,/) in between them.
A -> single letter
AB+ -> two letters with one operator + before them.
AB+CD-+ -> Two postfixes expression.

Q. Write an algorithm to convert infix into postfix expression.

Algorithm:
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
    3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty), push it.

    3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.

4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop and output from the stack until an '(' is encountered.
6. Repeat steps 2-6 until infix expression is scanned.
7. Pop and output from the stack until it is not empty.

Q. Write an algorithm to evaluate the postfix expression.

Algorithm:

1. If an operand is encountered, push it on stack.
2. If an operator 'op' is encountered.
    a. Pop two elements of stack, where A is the top element and B is the next top element
    b. Evaluate B op A.
    c. Push the result on stack.
3. The evaluated value is equal to the value at the top of the stack.

Precedence plays very important role in this case. Check precedence table attached below

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^=	assignment	2	right-to-left
,	Comma	1	left-to-right

Q. Convert A+(B*C-(D/E^F)*G)*H into postfix expression and evaluate the expression if A=2,B=9,C=3,D=16,E=4,F=1,G=5,H=8 into tabular form.

INPUT CHARACTER	OPERATION ON STACK	STACK	POSTFIX EXPRESSION
A		**EMPTY**	**A**
+	**PUSH**	**+**	**A**
(	**CHECK AND PUSH**	**+(**	**A**
B		**+(**	**AB**
*****	**CHECK AND PUSH**	**+(***	**AB**
C		**+(***	**ABC**
-	**CHECK, – HAS LOWER PRECEDENCE THAN * SO POP ALL OPERATORS UNTILL STACK HAS OPERATOR**	**+(-**	**ABC***

	WITH LESS-PRECEDENCE THAN – (STACK ALWAYS HOLD EQUAL OR LESS OPERATOR ON THE TOP THAN SCAN OPERATOR)		
(	**CHECK AND PUSH**	**+(-(**	**ABC***
D		**+(-(**	**ABC*D**
/	**CHECK AND PUSH**	**+(-(/**	**ABC*D**
E		**+(-(/**	**ABC*DE**
^	**CHECK,^ HAS HIGHER PRECEDENCE**	**+(-(/^**	**ABC*DE**
F		**+(-(/^**	**ABC*DEF**
)	**POP ALL OPERATOR UNTILL YOU ENCOUNTER RESPECTIVE (**	**+(-**	**ABC*DEF^/**
*****	**CHECK, * HAS HIGHER PRECEDENCE**	**+(-***	**ABC*DEF^/**
G		**+(-***	**ABC*DEF^/G**
)	**POP ALL OPERATOR UNTILL YOU ENCOUNTER RESPECTIVE (**	**+**	**ABC*DEF^/G*-**
*****	**CHECK AND PUSH**	**+***	**ABC*DEF^/G*-**
H		**+***	**ABC*DEF^/G*-H**
END	**POP ALL REMAINING OPERATORS**		**ABC*DEF^/G*-H*+**

Now evaluating the expression ABC*DEF^/G*-H*+ 2 9 3* 16 4 1 ^/ 5 *-8*+

INPUT CHARACTER	STACK	OPERATION
2	**2**	
9	**2 9**	
3	**2 9 3**	
*****	**2 27**	*** OPERATOR , POP**

		TWO VALUES AND PERFORM 9*3=27 AND PUSH RESULT INTO STACK
16	**2 27 16**	
4	**2 27 16 4**	
1	**2 27 16 4 1**	
^	**2 27 16 4**	**4^1**
/	**2 27 4**	**16/4=4**
5	**2 27 4 5**	
*****	**2 27 20**	**4*5=20**
-	**2 7**	**27-20=7**
8	**2 7 8**	
*****	**2 56**	**7*8=56**
+	**58 (ANS)**	**2+56=58**

Note: to validate your answer check on the calculator A+(B*C-(D/E^F)*G)*H with given value as: 2+(9*3-(16/4^1)*5)*8 then you get 58.

Q. Write an algorithm to convert infix into prefix expression.

Algorithm:

First Reverse the input string and follows steps:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,

   If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty), push it.

   Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.

4. If the scanned character is an ')', push it to the stack.
5. If the scanned character is an '(', pop and output from the stack until an ')' is encountered.
6. Repeat steps 2-6 until infix expression is scanned.
7. Pop and output from the stack until it is not empty
8. Reverse the output string

Q. Convert A+(B*C-(D/E^F)*G)*H into prefix expression and evaluate with given values as above.

Reverse the given string as H*)G*)F^E/D(-C*B(+A

INPUT CHARACTER	OPERATION ON STACK	STACK	POSTFIX EXPRESSION
H		EMPTY	H
*	PUSH	*	H
)	CHECK AND PUSH	*)	H
G		*)	HG
*	CHECK AND PUSH	*)*	HG
)	CHECK AND PUSH	*)*)	
F		*)*)	HGF
^	CHECK AND PUSH	*)*)^	HGF
E		*)*)^	HGFE
/	CHECK AND POP ^	*)*)/	HGFE^
D		*)*)/	HGFE^D
(	POP	*)*	HGFE^D/
-	CHECK AND POP*	*)-	HGFE^D/*
C		*)-	HGFE^D/*C
*	CHECK, * HAS HIGHER PRECEDENCE	*)-*	HGFE^D/*C
B		*)-*	HGFE^D/*CB
(	POP ALL OPERATOR UNTILL YOU ENCOUNTER RESPECTIVE (	*	HGFE^D/*CB*-
+	CHECK AND POP *	+	HGFE^D/*CB*-*
A		+	HGFE^D/*CB*-*A
END	POP ALL REMAINING OPERATORS		HGFE^D/*CB*-*A+

Reverse **HGFE^D/*CB*-*A+** as **+A*-*BC*/D^EFGH** is the prefix form.

Q. Evaluate the +A*-*BC*/D^EFGH or +2*-*93*/16^4158 prefix expression.

First reverse the string as **HGFE^D/*CB*-*A+ or 8 5 1 4^16/*3 9*-*2+**

INPUT CHARACTER	STACK	OPERATION
8	8	
5	8 5	
1	8 5 1	
4	8 5 1 4	
^	8 5 4	4^1=4 (reverse than postfix)
16	8 5 4 16	
/	8 5 4	16/4=4
*	8 20	4*5=20
3	8 20 3	16/4=4
9	8 20 3 9	
*	8 20 27	9*3=27
-	8 7	27-20=7
*	56	7*8=56
2	56 2	
+	58 (ANS)	2+56=58

EXERCISE (vvi): Convert into postfix and prefix.

1. **(A+B*(C+D/E)^F*G)**
   Ans: A B C D E / + F ^ * G * + prefix: +A*B*^+C/DEFG

2. **A+[B+C/(D/E)*F]/G**
   Ans: A [B + C D E / / F] * G / + prefix: +A+ [B/C*/DE/F] G. if [ ] is used expressed in expression ELSE
   Ans: ABCDE//F*+G/+ prefix: +A/+B/C*/DEFG -> preferred this for exam

3. **A+[B+C/(D/E)*$*F]/G**
   Ans: A [B + C D E / / $ * F] * G / + prefix: +A+ [B/C*/DE*$/F] G. if [ ] is used expressed in expression ELSE
   Ans: ABCDE//$*F*+G/+ prefix: +A/+B/C*/DE*$FG -> preferred this for exam

1. Stack can be used for checking balancing of symbols.

Stacks can be used to check whether the given expression has balanced symbols or not. This algorithm is very much useful in compiler. Each time parser reads one character at a time. If the character is an opening delimiter like (,{,[- then it is written to stack. When a closing delimiter is encounter like),},]- is encountered the stack is popped. The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line.

Algorithm:

a) **Create a stack.**

b) **While(end of input is not reached){**

{

a) **If the character read is not a symbol to be balanced, ignore it.**

b) **If the character is an opening symbol like (,[,{, push it onto the stack.**

c) **If it is a closing symbol like),],} then if the stack is empty report an error. Otherwise pop the stack.**

d) **If the symbol popped is not the corresponding opening symbol, report an error.**

c) **At end of input, if the stack is not empty report an error.**

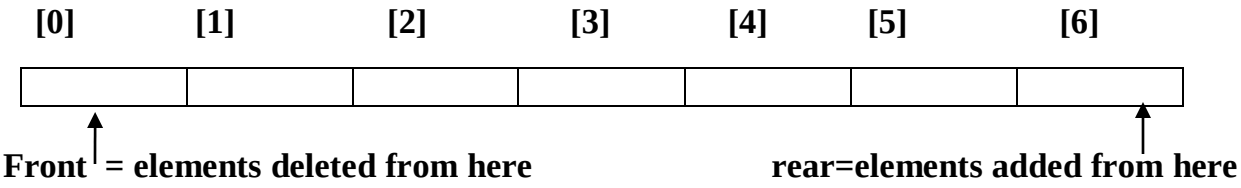
By using algorithm, check validity of ()(()[()]).

Example	Valid?	Description
(A+B)+(C-D)	Yes	The expression is having balanced symbol
((A+B)+(C-D)	No	One closing brace is missing
((A+B)+[C-D])	Yes	Opening and immediate closing braces correspond
((A+B)+[C-D]]	No	The last closing brace does not correspond with the first opening parenthesis

Input Symbol, A[i]	Operation	Stack	Output
(	Push (	(	
)	Pop (Test if (and A[i] match? YES		
(	Push (	(	
(	Push (	((	
)	Pop (Test if (and A[i] match? YES	(	
[	Push [	([	
(	Push (	([(	
)	Pop (Test if(and A[i] match? YES	([	
]	Pop [Test if [and A[i] match? YES	(	
)	Pop (Test if(and A[i] match? YES		
	Test if stack is Empty? YES		TRUE

Q. Define Queue:

A Queue is an ordered list in which insertions are done at one end (rear) and deletions are done at other end (front). The first element inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) list.



A good example of queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Terms used in queue are as follows:

1. **Enqueue**: Adds an item to the queue. If the queue is full, then it is said to be an **Overflow condition.**

2. **Dequeue**: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an **Underflow condition.**

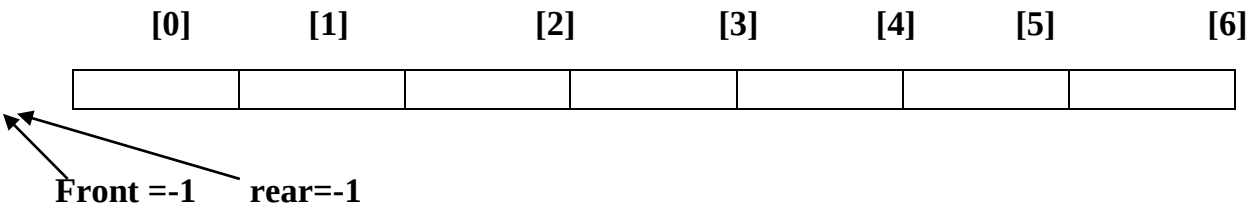
3. **Front(head):** Get the front item from queue.

4. **Rear(tail):** Get the last item from queue.

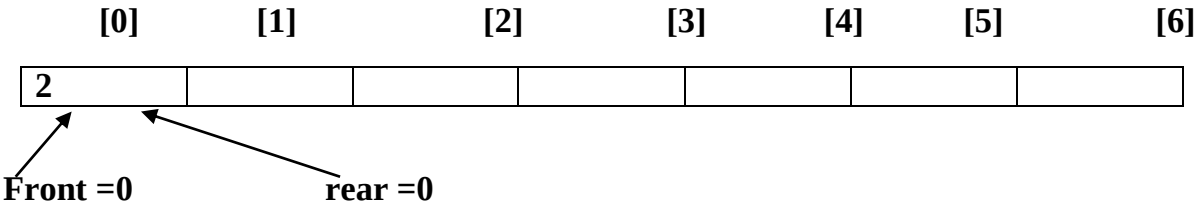
5. You can initialize **front =rear=0 or front =rear=-1**.

Q. Explain the operation of Queue.

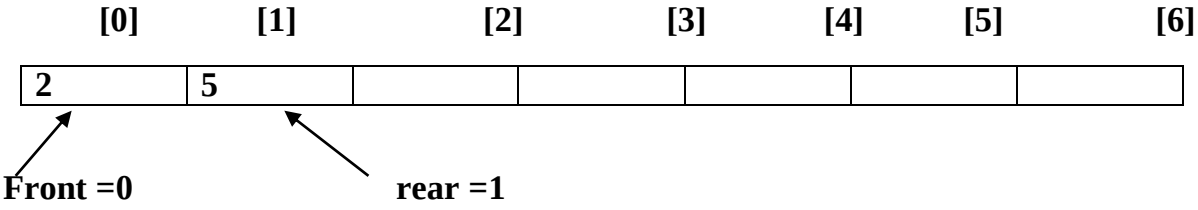
Operations in Queue graphically(initialize front =rear=-1)



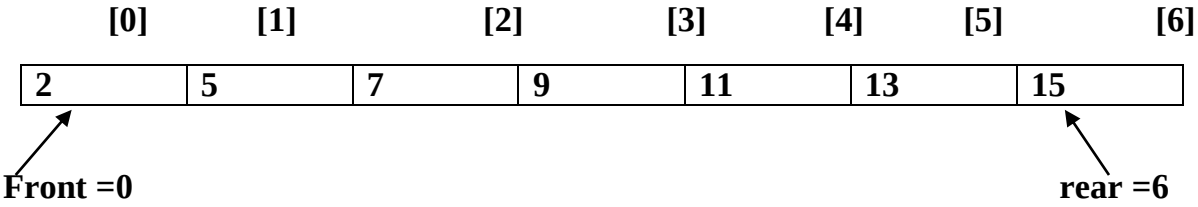
a. insert 2



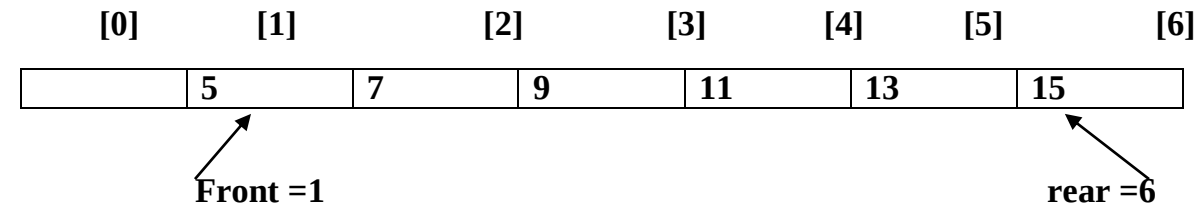
b.Insert 5



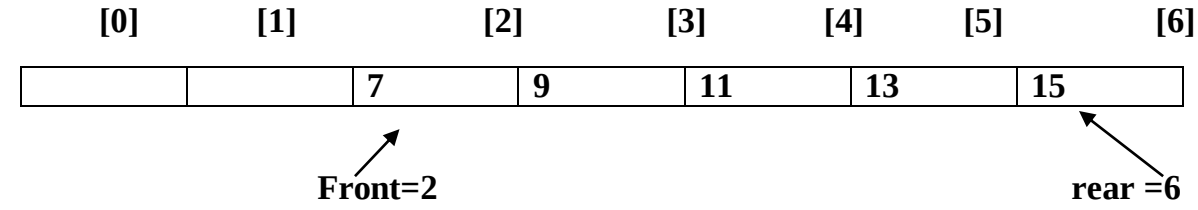
c.Insert 7,911,13,15



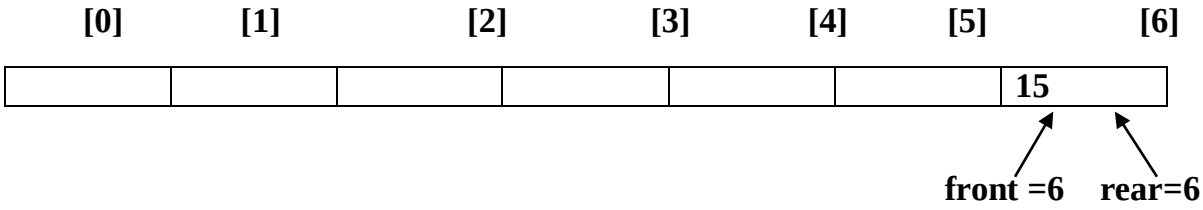
d.delete 2



e.delete 5



f.delete 7,9,11,13



Algorithm for enqueue and dequeue:

a. Enqueue:

Step 1: Initialize the front = rear=-1;
Step 2: Repeat step 3 to until rear<MAXSIZE-1
Step3: Read item
Step 4: if front==-1 then
Front=rear=0
else
rear=rear+1
step5: queue[rear]=item
step6: if condition of step 2 does not satisfy then print queue overflow.

b.Dequeue:

Step 1: Repeat step 2 to 4 untill front>=0

Step 2: Set item=queue[front]
Step3: If front==real
Set front=-1
Set rear=-1
else

front=front+1
step5: print deleted item
step6: print queue is empty

Note: Writing style of an algorithm may vary but concept should not be……

Q. Explain about the linear Queue.

Linear Queue based on FIFO concept and implemented in two ways:

a. **Contiguous linear queue**: The queue is implemented using an array as explained above. (Explain enqueue and dequeue operation as described as above).

b. **Linked linear queue:** The queue is implemented using pointer and dynamic memory allocation concept. We will discuss in next chapter- linked list.

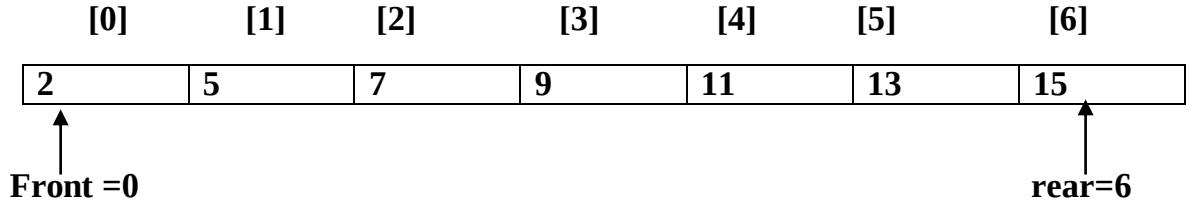
Q. Explain demerit of linear queue

Or,

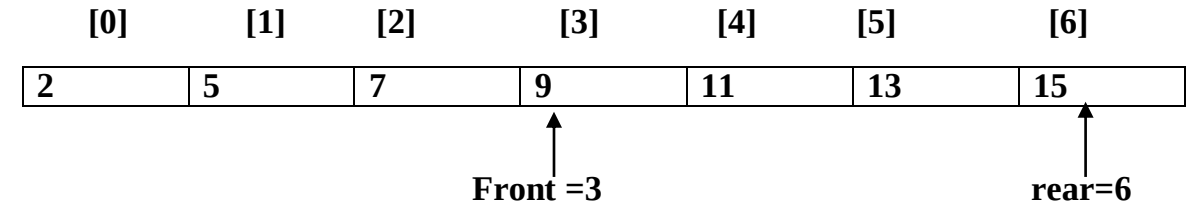
Explain why circular queue comes into existence.

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we cannot insert the next element until all the elements are deleted from the queue.

For example: consider the queue below after inserting all the elements into the queue.

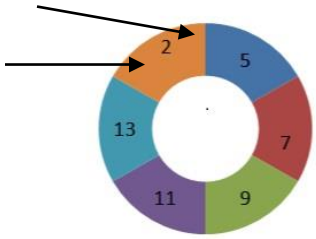


Now consider the following situation after deleting three elements from the queue.

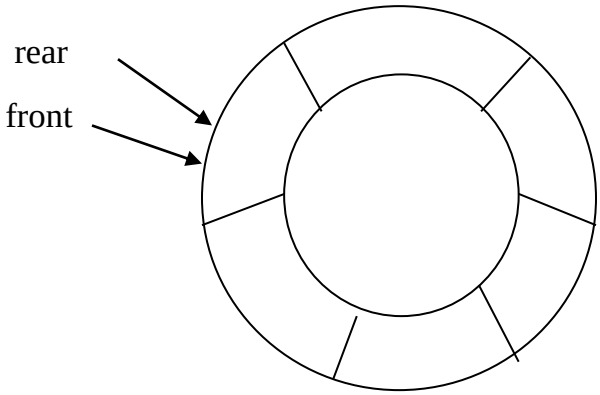


This situation also says that Queue is Full and we cannot insert the new element because, 'rear' is still at last position. In above situation, even though we have empty positions in the queue we cannot make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem we use circular queue data structure.

Q. Explain the operation enqueue and dequeue in circular queue.



Circular Queue: is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. Graphical representation of a circular queue is as follows...



Algorithm for Enqueue:

1. If (front ==(rear+1)%MAX) then
   print "Overflow"

   else:
   read the value

   if(front==-1)then

   set front=rear=0

   else,

   rear=(rear+1)%MAX;

   q[rear]=data;

Alternative algorithm for enqueue:

Step 1:If (front==0 && rear=max-1)|| (front == rear+1)

Then write "Queue overflow " and stop.

Step 2: Read data to insert in circular queue

Step 3: if (front = -1) then set front=rear=0

Step 4:if(rear=max-1) then rear=0

else rear=rear+1

Step 5: cqueue[rear]=data

Step 6: Stop

Algorithm for Dequeue:

```
If (front ==-1) then
        print "underflow"
  else:
      data=q[front]
      if(front==rear)then
            set  front=rear=-1
      else,
       front=(front+1)%MAX;
```

Q. Define priority queue with its application.

Applications of Priority Queue:

1)CPU Scheduling

2)Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc

3)All queue applications where priority is involved.

Assignment: Priority Queue

By: Er. Bishwas Pokharel, Lecturer

Q. Why there is the need of the linked list? Or Mention drawbacks of array

Arrays can be used to store linear data of similar types, but arrays have following limitations.

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

2) Inserting a new element in an array of elements is expensive; because room has to be created for the new elements and to create room existing elements have to shift.

For example, in a system if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Q. Define Linked List:

A linked list is a linear data structure where each element is a separate object. Each element (we will call it a node) of a list is comprising of two items –

- ➔ The data and a reference to the next node
- ➔ The last node has a reference to null.
- ➔ The entry point into a linked list is called the HEAD of the list

Data	Reference to next node

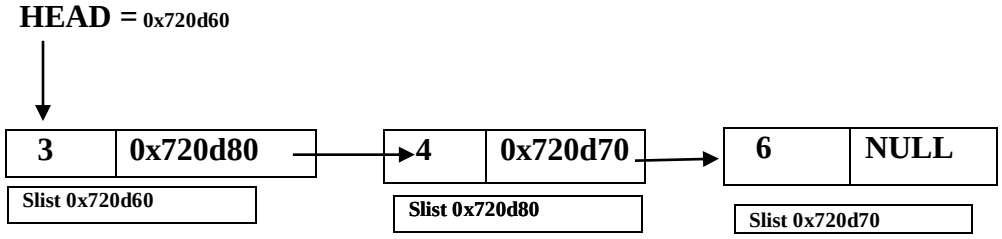
Note: It should be noted that HEAD is not a separate node, but the reference to the first node. If the list is empty then the HEAD is a null reference. A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

There are two type of linked list:

a. Singly linked list
b. Doubly linked list

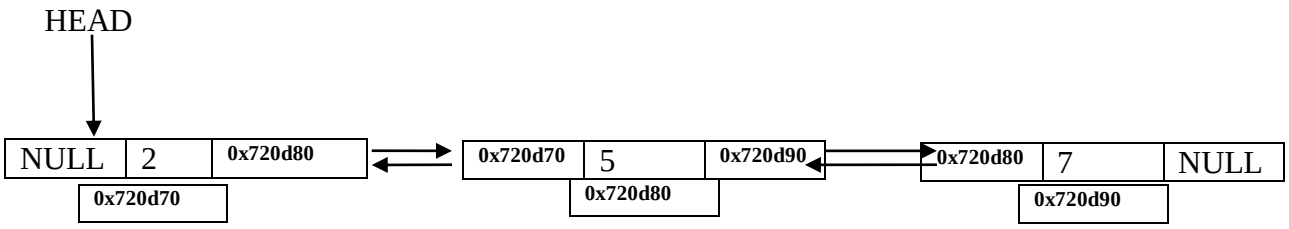
Graphical representation;

a. **Singly linked list**



Here, HEAD is the pointer which holds the address of node having data (3) and the next field of same node contain the address of the node having data(4) and so on until there is data in the node. Finally, the last node points to no other node so are mentioned null.

b. **Doubly linked list**



Here, a node contain one data and two address, previous (prev) and next. Prev is null and next points to **0x720d80**. In figure above, **0x720d80** contain a data 5 and it's previous contain the address of the node1 **0x720d70**.This means that there is dual linked up. Node 1 has information of Node 2 and Node 2 has information about the Node1.

Q. Mention advantage of linked list over array.

Advantages over arrays

1) Dynamic size

2) Ease of insertion/deletion

3) Memory utilization is inefficient in the array. Conversely, memory utilization is efficient in the linked list.

Q. Mention disadvantage of linked list over array.

Disadvantages over arrays

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.

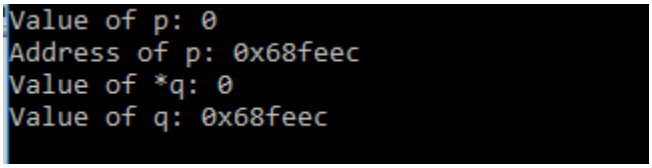
2) Extra memory space for a pointer is required with each element of the list.

3) Elements are stored consecutively in arrays whereas it is stored randomly in Linked lists.

Note: Not important for exam but **concept is necessary** for understanding the algorithm clearly. Before knowing singly lists implementation we have to understand some program related to pointer as well.

```
#include<iostream>

using namespace std;

void pointer(int **q)

{

   cout<<"Value of *q: "<<*q<<endl;   // same as p

   cout<<"Value of q: "<<q<<endl;    //give address of q

}

int main(void)

{

//here p is null pointer which doesnot point anywhere with value of p as 0

   int *p=NULL;

   cout<<"Value of p: " << p<<endl;

   cout<<"Address of p: "<< &p<<endl;

   pointer(&p);

return 0;

}
```

Ans:



Explanation:

//int **q is the pointer to pointer and initialize with the address of the pointer that why //&p, q=&p and int **(&p) = *p as *& inverse with each other // q=&p so *q = *(&p)=p //gives the address of the pointer p

Q. Implementation of singly linked list for insertion:

```
#include<iostream>

using namespace std;

class Singlylist

{

private:

   int data;

    Singlylist *next;

public:


   void add(Singlylist **HEAD_ref,int a)

   {

      //HEAD_ref=&HEAD

      //*HEAD_ref= *(&HEAD)= HEAD =0


      Singlylist *slist=new Singlylist();

      cout<<"slist address: "<<slist<<endl;
```

```
        slist->data=a;
        slist->next=(*HEAD_ref);
        *HEAD_ref=slist;

        cout<<"Slist->data: "<< slist->data<<"\t";
        cout<<"Slist->next: "<< slist->next<<"\t"<<endl;
        cout<<"*HEAD_ref: "<< *HEAD_ref<<endl;
        cout<<endl;
    }
    void printList(Singlylist *node)
    {
        cout<<"data in the linked list: "<<endl;
        while (node != NULL)
        {
            cout<<node->data<<"\t";
            node = node->next;
        }
    }
};
int main(void)
{
    Singlylist s;
    Singlylist *HEAD=NULL;
    cout<<"Value of HEAD: "<< HEAD<<endl;
```

```
    cout<<"Address of HEAD: "<< &HEAD<<endl<<endl;
    s.add(&HEAD,2);
    cout<<endl;
    s.add(&HEAD,3);
    s.printList(HEAD);
    return 0;
}
```

Explanation:

```
Value of head: 0
Address of head: 0x68fee4

slist address: 0x720d80
Slist->data: 2  Slist->next: 0
*head_ref: 0x720d80


slist address: 0x720d90
Slist->data: 3  Slist->next: 0x720d80
*head_ref: 0x720d90

data in the linked list:
3       2
```

Singly linked list

Q. Explain the Inserting a new node in a singly linked list:

We will take some case and see how insertion is done in each case

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

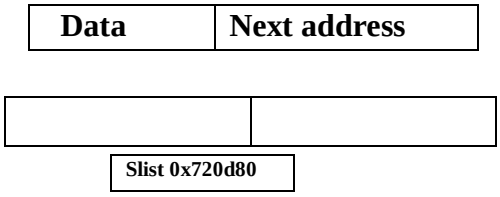
Case 4: The new node is inserted before a given node (Assignment)

Q. Explain the inserting of new node at the beginning.

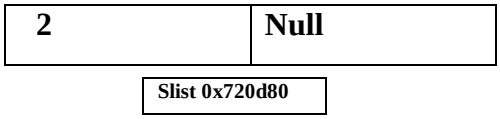
1. First, if there is no any node then the HEAD is pointing to the Null. Where, HEAD has its own address. (Say, 0x68fee4)

HEAD→NULL

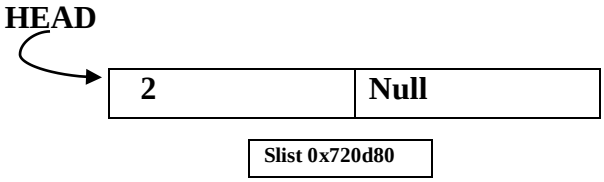
2. Secondly, A new node is created say (Slist, from above program) having its own address (Say, 0x720d80).



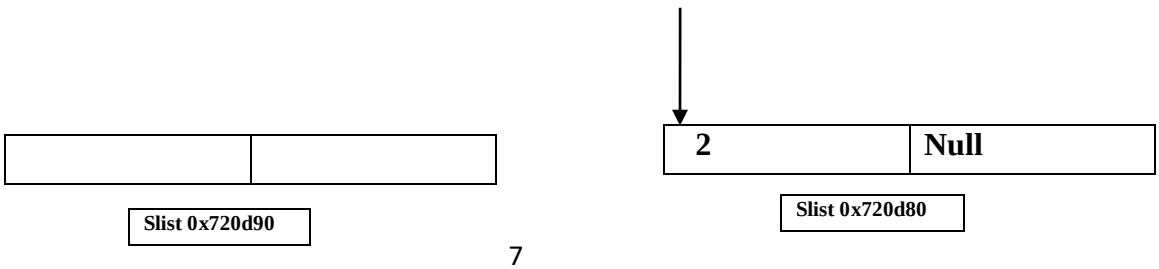
3. Insert data 2 in Slist and has the data 2, in the next address it is assigned with the HEAD ie Null.



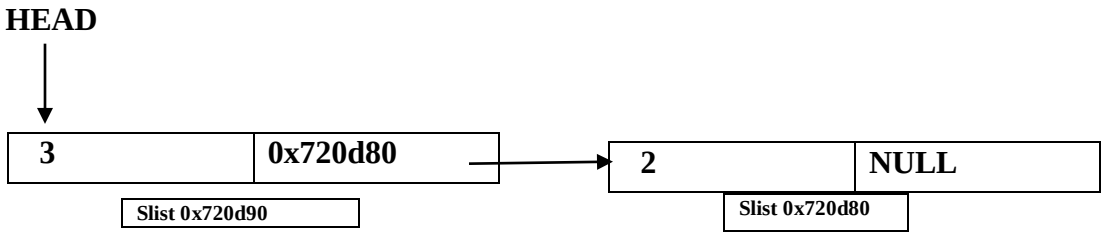
4. Now HEAD pointer points to the first node that means at the address of the Slist(Say, Slist 0x720d80)



5. Again insert new node at the beginning. So create a new node having its own address (say 0x720d890).



6. Insert data having value 3 in new node and in its next address we have to assign the address of the old Slist say,0x720d80 and HEAD with address of the new node 0x720d90.



Algorithm:

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 7

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL-> NEXT

Step 4: SET DATA = VAL

Step 5: SET NEW_NODE NEXT = HEAD

Step 6: SET HEAD = NEW_NODE

Step 7: EXIT

Explanation:

In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if a free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in

HEAD. Now, since the new node is added as the first node of the list, it will now be known as the HEAD node, that is, the HEAD pointer variable will now hold the address of the NEW_NODE. Note the following two steps:

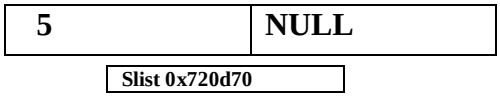
Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

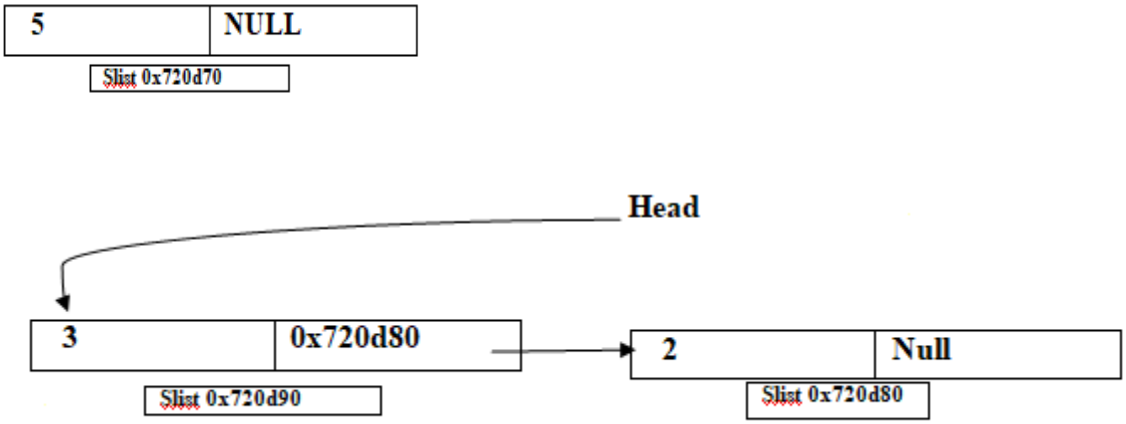
These steps allocate memory for the new node. In C, there are functions like malloc(), alloc, and calloc() which automatically do the memory allocation on behalf of the user.

Q. **Explain the inserting of new node at the end.**

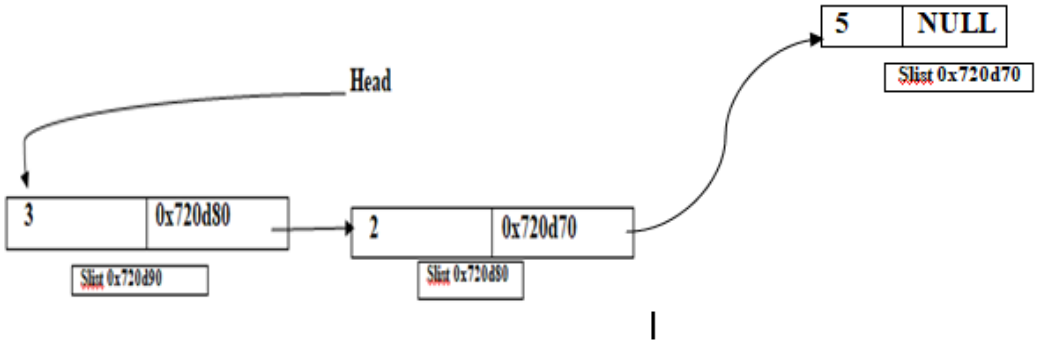
1. Create a new node. Insert the data =5 in the new node and new node have its own address say 0x720d70 and in next = NULL



2. Assign the new pointer with PTR→ HEAD and move it in the singly list until PTR->next =NULL.



3. Finally, set the value PTR-->next=New node when it finds PTR->next=Null.



5	NULL
Slist 0x720d70	

Algorithm:

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 1

[END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL-> NEXT

Step 4: SET DATA = VAL

Step 5: SET NEW_NODE = NULL

Step 6: SET PTR = HEAD

Step 7: Repeat Step 8 while PTR → NEXT != NULL

Step 8: SET PTR = PTR →NEXT

[END OF LOOP]

Step 9: SET PTR →NEXT = NEW NODE

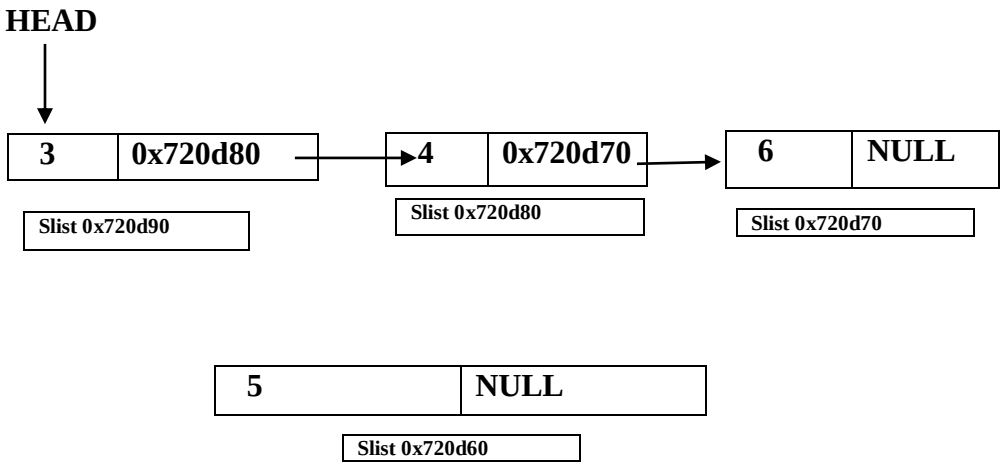
Step 10: EXIT

Explanation: The algorithm to insert a new node at the end of a linked list. In Step 6, we take a pointer variable PTR and initialize it with HEAD. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

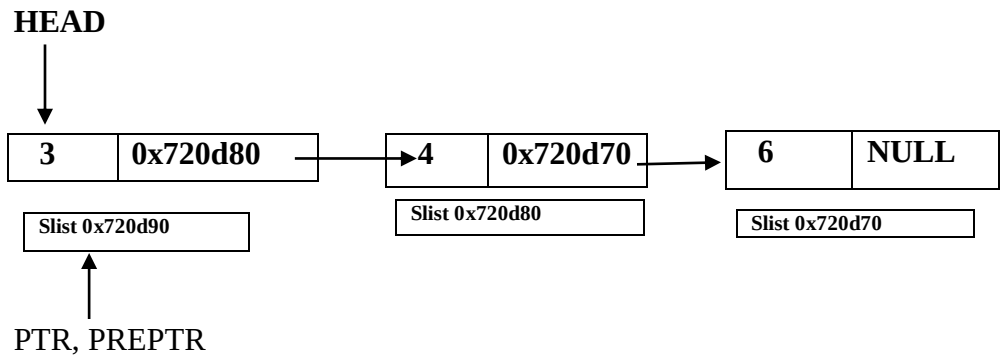
Q. **Explain the inserting of new node at any position (after any specific new node)**

Insert after node 2 having data 4,

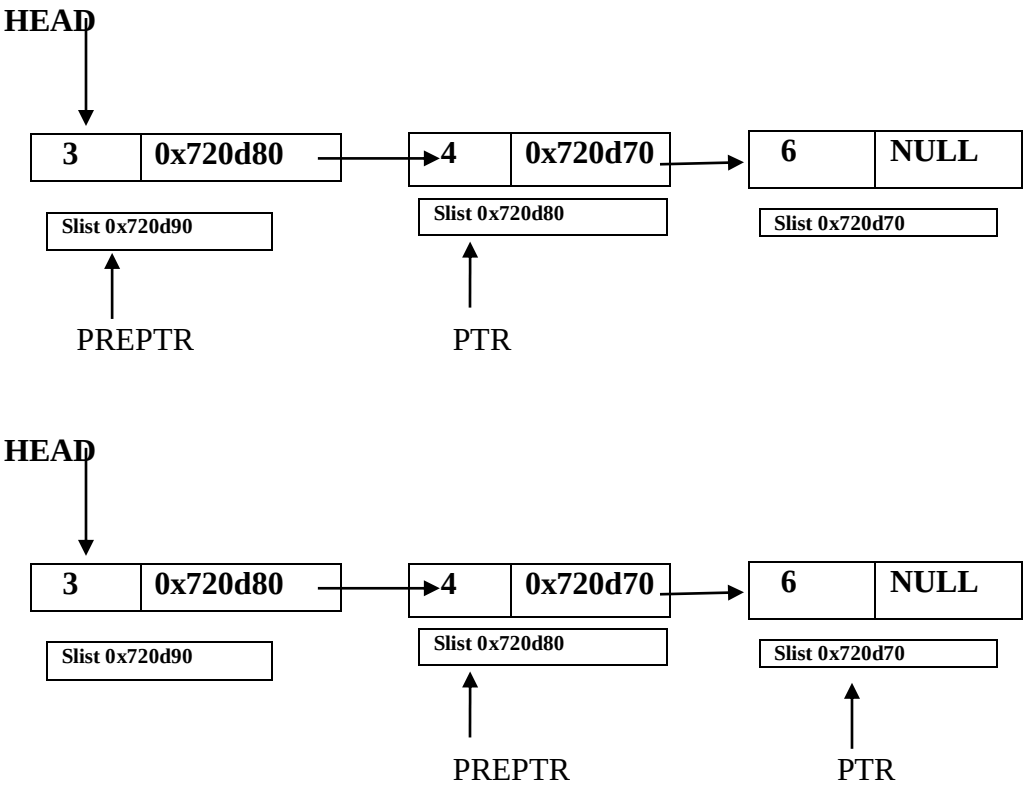
1. Allocate memory for the new node and initialize its DATA part to 5.



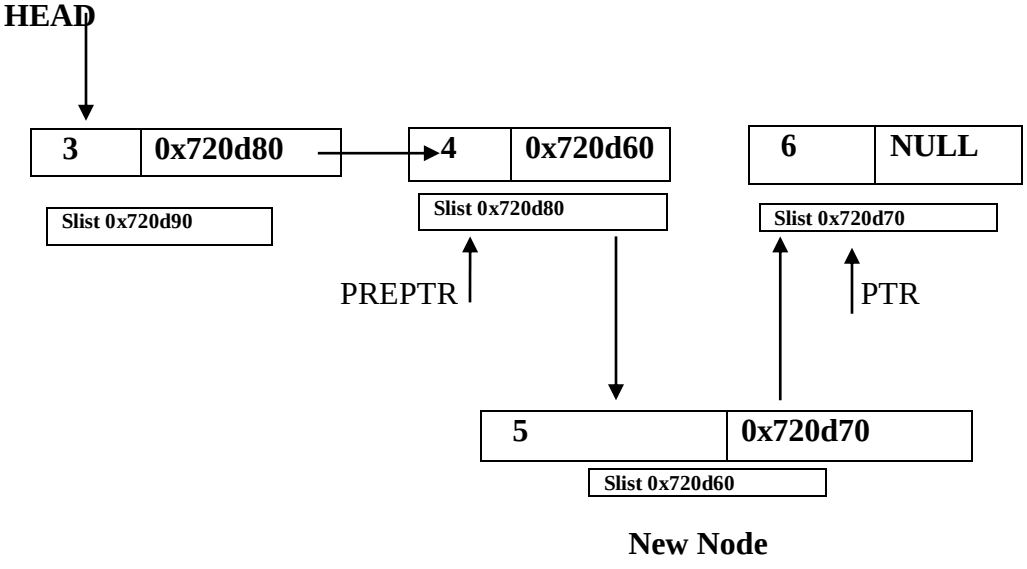
2. Take two pointer variables PTR and PREPTR and initialize them with **HEAD** so that HEAD, PTR, and PREPTR point to the first node of the list.



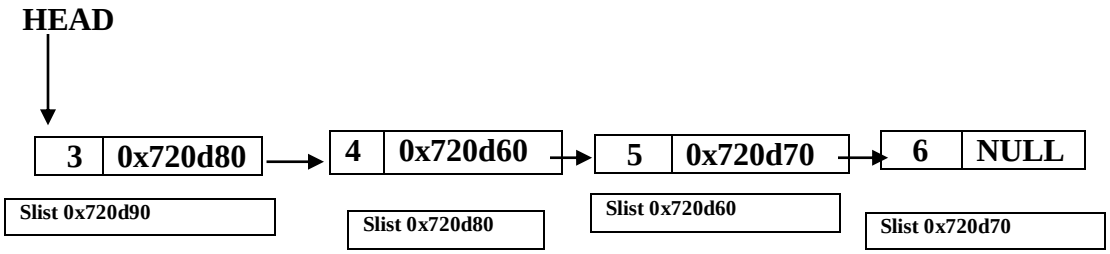
3. Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.



4. Add the new node in between the nodes pointed by PREPTR and PTR.



5.



Algorithm:

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 12

[END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL→ NEXT

Step 4: SET DATA = VAL

Step 5: SET PTR = HEAD

Step 6: SET PREPTR = PTR

Step 7: Repeat Steps 8 and 9 while PREPTR →DATA!= NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR →NEXT

[END OF LOOP]

Step 10: PREPTR→ NEXT = NEW_NODE

Step 11: SET NEW_NODE→NEXT=PTR

Step12: EXIT

Explanation: We take a pointer variable PTR and initialize it with HEAD. That is, PTR now points to the first node of the linked list. Then, we take another pointer variable PREPTR and initialize it with PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is inserted before the desired node.

Q. Explain the deleting a new node in a singly linked list:

In this section, we will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

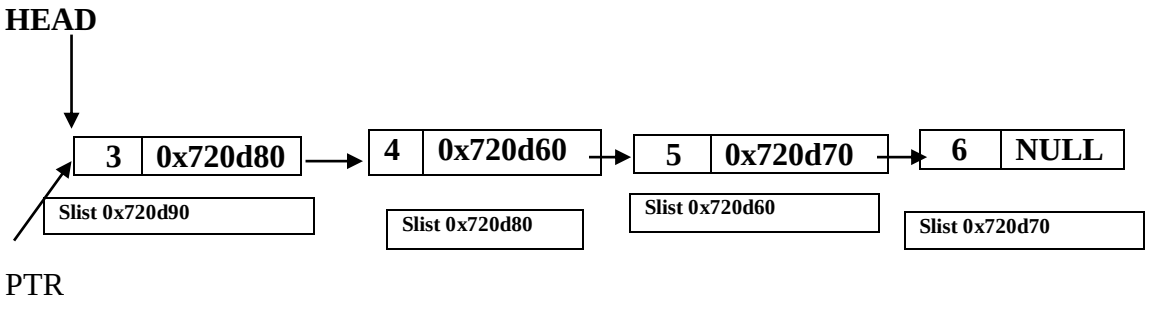
Case 1: The first node is deleted.

Case 2: The last node is deleted.

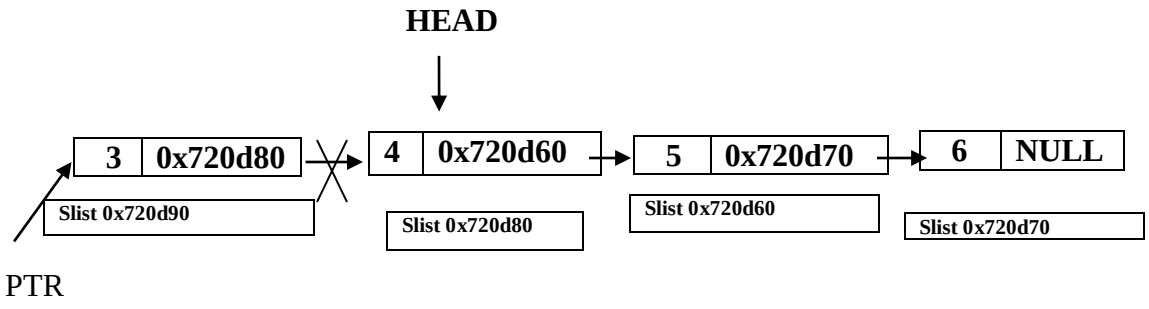
Case 3: The node after a given node is deleted

Q .Explain the deleting of first node in singly linked list.

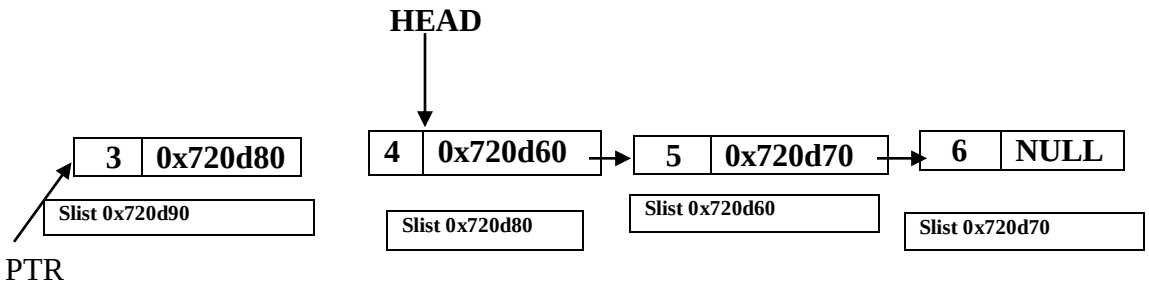
1. A given linked list where PTR is the pointer which points to the same address as pointed by the HEAD.



2. HEAD value is assigned with the HEAD→next and PTR pointer is made free.



3. Here is the final linked list after deletion from the beginning**.**



Algorithm:

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 5

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET HEAD = HEAD→ NEXT

Step 4: FREE PTR

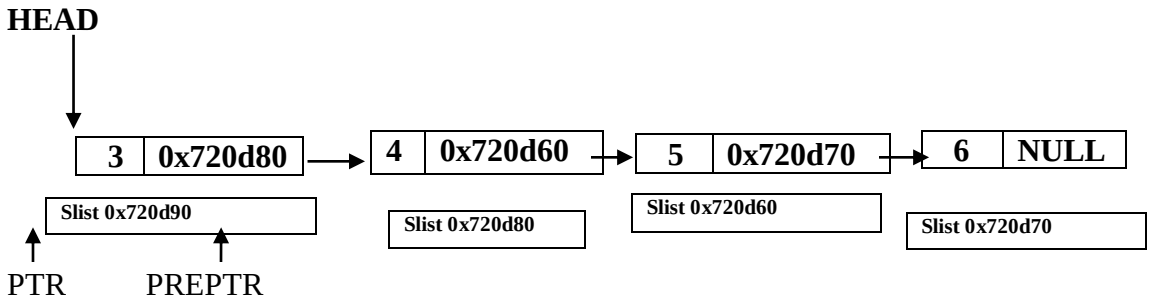
Step 5: EXIT

Explanation:

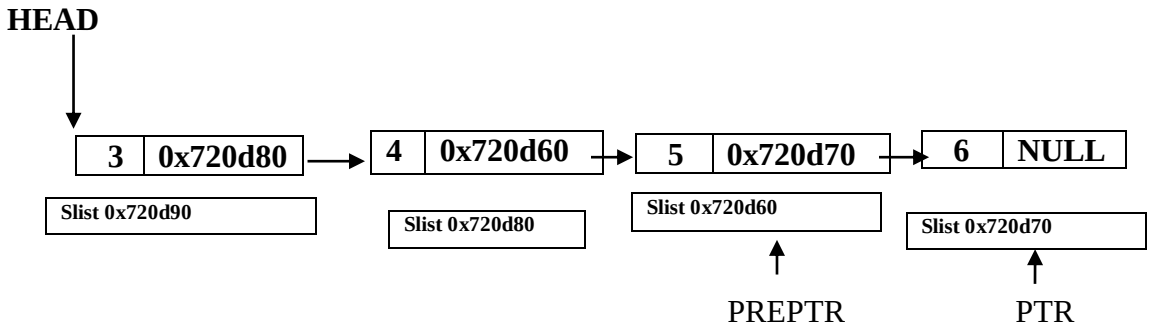
In Step 1, we check if the linked list exists or not. If HEAD = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with HEAD that stores the address of the first node of the list. In Step 3, HEAD is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

Q .Explain the deleting of last node in singly linked list.

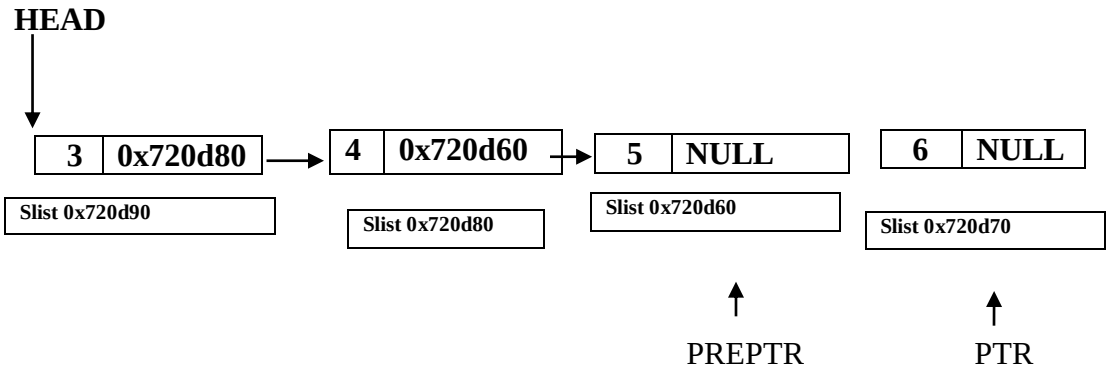
1. Take pointer variables PTR and PREPTR which initially point to HEAD



2. Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.



3. Set the NEXT part of PREPTR node to NULL.



Algorithms:

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR →NEXT != NULL

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR →NEXT

[END OF LOOP]

Step 6: SET PREPTR →NEXT = NULL

Step 7: FREE PTR

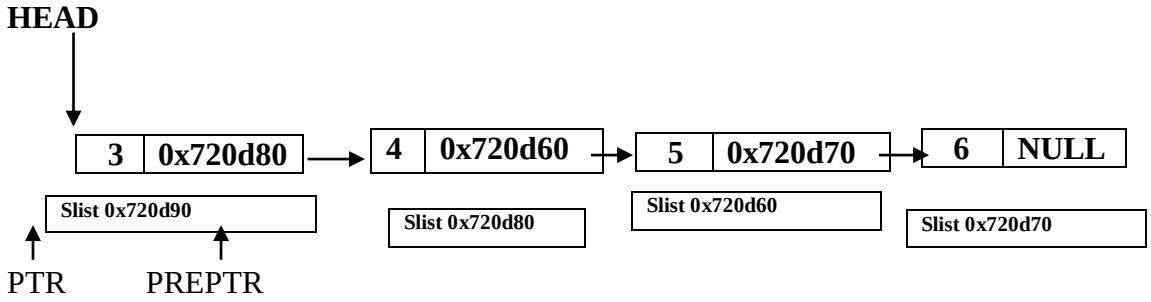
Step 8: EXIT

Explanation:

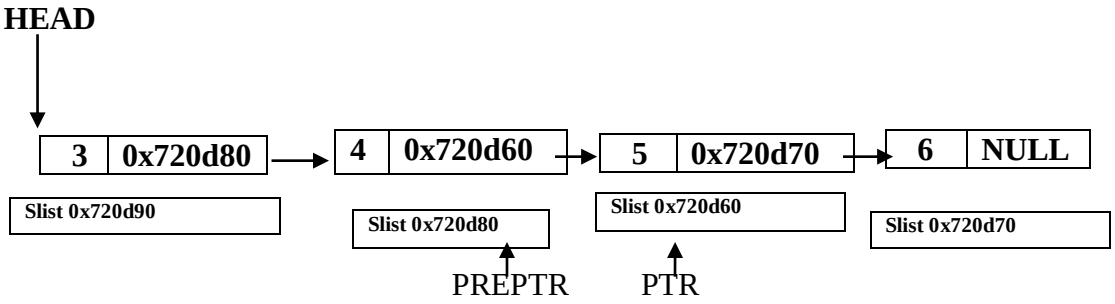
In Step 2, we take a pointer variable PTR and initialize it with HEAD. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned back to the free pool.

Q .Explain the deleting of node after a given node in singly linked list.

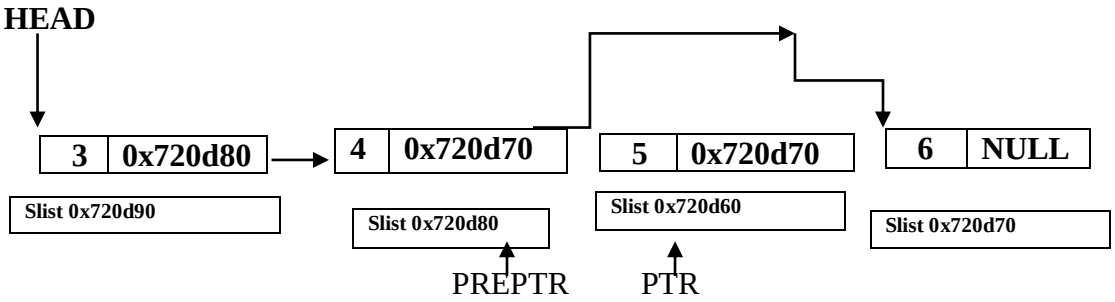
1. Take pointer variables PTR and PREPTR which initially point to HEAD.



2. Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.



3. Set the NEXT part of PREPTR to the NEXT part of PTR.



Algorithm:

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 1

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET PREPTR = PTR

Step 4: Repeat Steps 5 and 6 while PREPTR →DATA!= NUM

Step 5: SET PREPTR = PTR

Step 6: SET PTR = PTR→ NEXT

[END OF LOOP]

Step 7: SET TEMP = PTR

Step 8: SET PREPTR →NEXT = PTR→ NEXT

Step 9: FREE TEMP

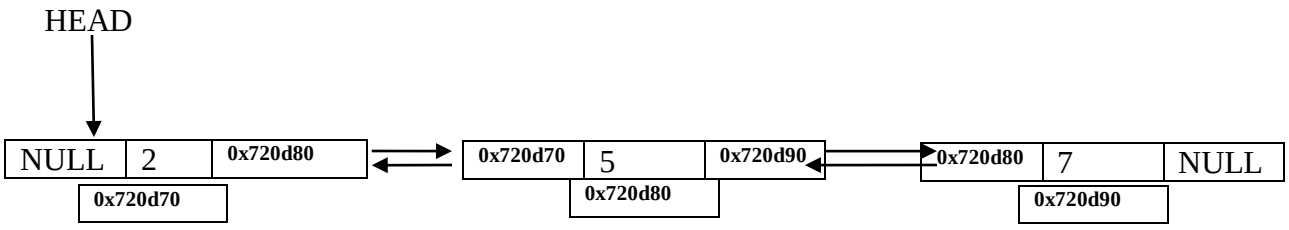
Step 10 : EXIT

Explanation:

In Step 2, we take a pointer variable PTR and initialize it with HEAD. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it. The memory of the node succeeding the given node is freed and returned back to the free pool.

Doubly linked list:

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node.



Implementation of insertion of node at the beginning of the doubly linked list

```
#include<iostream>
using namespace std;
class doublylist
{
private:
   int data;
    doublylist *next;
```

```
    doublylist *prev;
public:
   void add(doublylist **head_ref,int a)
   {
      //head_ref=&head
      //*head_ref= *(&head)= head =0

      doublylist *slist=new doublylist();
      doublylist *temp;
      cout<<"slist address: "<<slist<<endl;
      slist->data=a;
      slist->next=(*head_ref);

      if(*head_ref==NULL)
      {
         temp=slist;
         temp->prev=NULL;
      }
      else
      {
         slist->next=*head_ref;
         slist->prev=NULL;
         temp->prev=slist;
      }
      *head_ref=slist;
```

```
        cout<<"slist->prev: "<<slist->prev<<"\t";
        cout<<"Slist->data: "<< slist->data<<"\t";
        cout<<"Slist->next: "<< slist->next<<"\t"<<endl;
        cout<<"\t \t\t \t post_slist->prev: "<<temp->prev<<endl;
        cout<<"*head_ref: "<< *head_ref<<endl;
        cout<<endl;
    }
    void printList(doublylist *node)
    {
        cout<<"data in the linked list: "<<endl;
        while (node != NULL)
        {
            cout<<node->data<<"\t";
            node = node->next;
        }
    }
};
int main(void)
{
    doublylist s;
    doublylist *head=NULL;
    cout<<"Value of head: "<< head<<endl;
    cout<<"Address of head: "<< &head<<endl<<endl;
    s.add(&head,2);
    cout<<endl;
```

```
    s.add(&head,3);
    s.printList(head);
    s.add(&head,5);
    s.printList(head);
    return 0;
}
```

Output:

```
Value of head: 0
Address of head: 0x68fee0

slist address: 0xd30d80
slist->prev: 0  Slist->data: 2  Slist->next: 0
                                post_slist->prev: 0
*head_ref: 0xd30d80


slist address: 0xd30d98
slist->prev: 0  Slist->data: 3  Slist->next: 0xd30d80
                                post_slist->prev: 0xd30d98
*head_ref: 0xd30d98

data in the linked list:
3       2       slist address: 0xd30db0
slist->prev: 0  Slist->data: 5  Slist->next: 0xd30d98
                                post_slist->prev: 0xd30db0
*head_ref: 0xd30db0

data in the linked list:
5       3       2
```

Inserting a new node in a doubly Linked List

In this section, we will discuss how a new node is added into an already existing doubly linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.

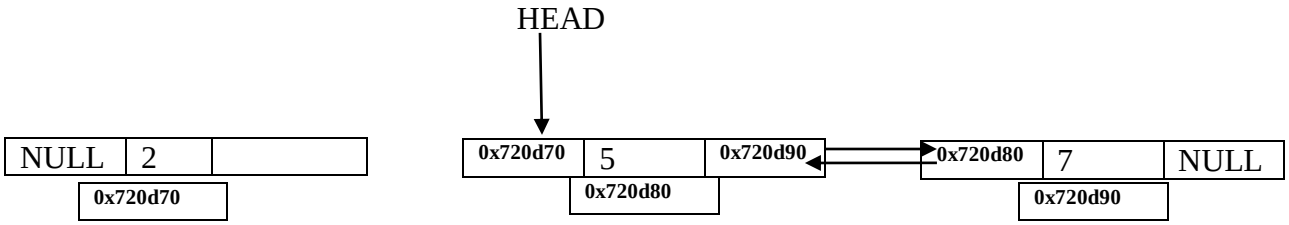
Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

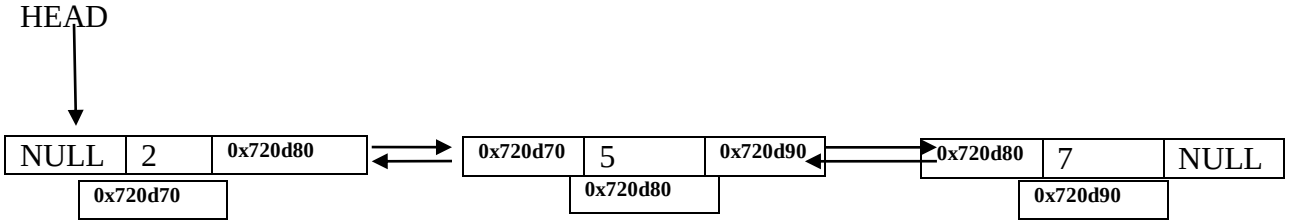
Case 4: The new node is inserted before a given node. (Assignment)

Q. Explain the insertion of the new node at the beginning of the doubly linked list.

1. Allocate memory for the new node and initialize its DATA part to 2 and PREV field to NULL.



2. Add the new node before the HEAD node. Now the new node becomes the first node of the list.



Algorithms:

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 9

[END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL→ NEXT

Step 4: SET NEW_NODE→ DATA = VAL

Step 5: SET NEW_NODE→PREV = NULL

Step 6: SET NEW_NODE→NEXT= HEAD

Step 7: SET HEAD→PREV= NEWNODE

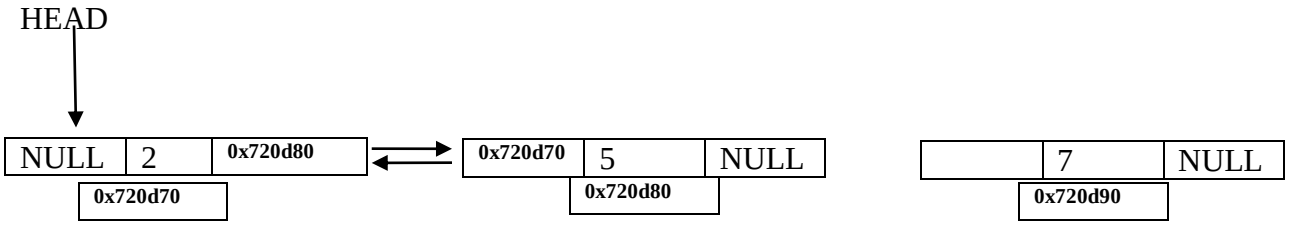
Step 8: HEAD=NEW_NODE Step 9: EXIT

Explanation:

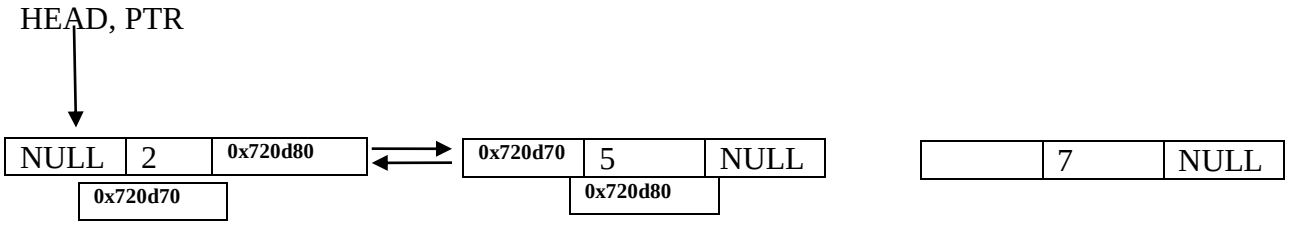
In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in HEAD. Now, since the new node is added as the first node of the list, it will now be known as the HEAD node, that is, the HEAD pointer variable will now hold the address of NEW_NODE

Q. Explain the insertion of the new node at the ending of the doubly linked list.

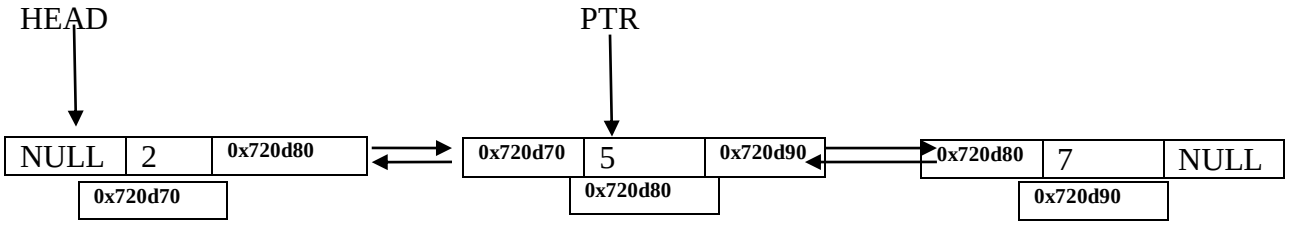
1. Allocate memory for the new node and initialize its DATA part to 7 and its NEXT field to NULL



2. Take a pointer variable PTR and make it point to the first node of the list



3. Move PTR so that it points to the last node of the list. Add the new node after the node pointed by PTR.



Algorithm

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL →NEXT

Step 4: SET NEW_NODE →DATA = VAL

Step 5: SET NEW_NODE→NEXT = NULL

Step 6: SET PTR = HEAD

Step 7: Repeat Step 8 while PTR→ NEXT! = NULL

Step 8: SET PTR = PTR →NEXT

[END OF LOOP]

Step 9: SET PTR →NEXT =NEW_NODE

Step 10: SET NEW_NODE→PREV = PTR

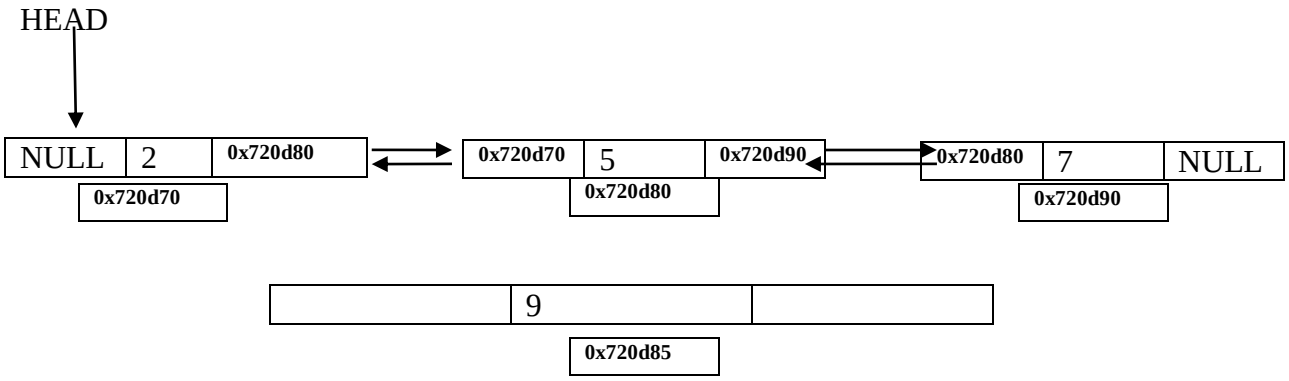
Step 11: EXIT

Explanation:

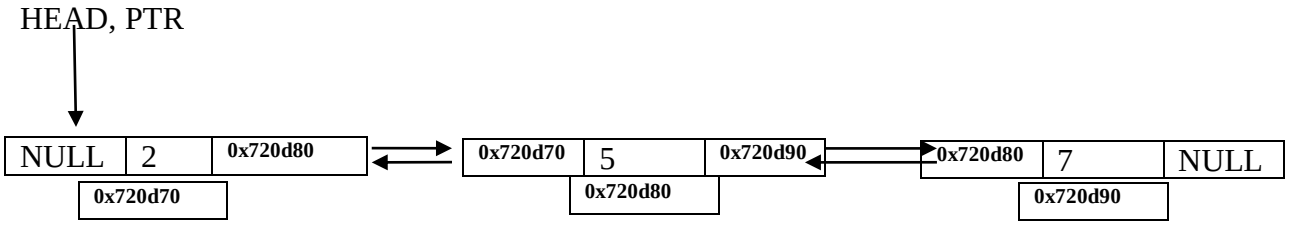
In Step 6, we take a pointer variable PTR and initialize it with HEAD. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list. The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

Q. Explain the insertion of the new node after a given node of the doubly linked list.

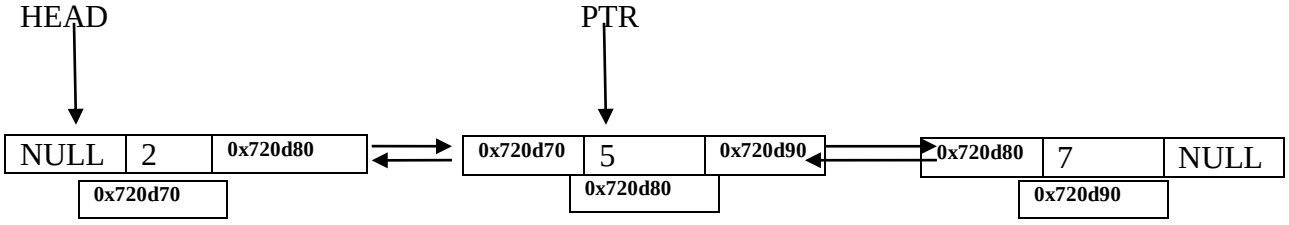
1. Allocate memory for the new node and initialize its DATA part to 9.



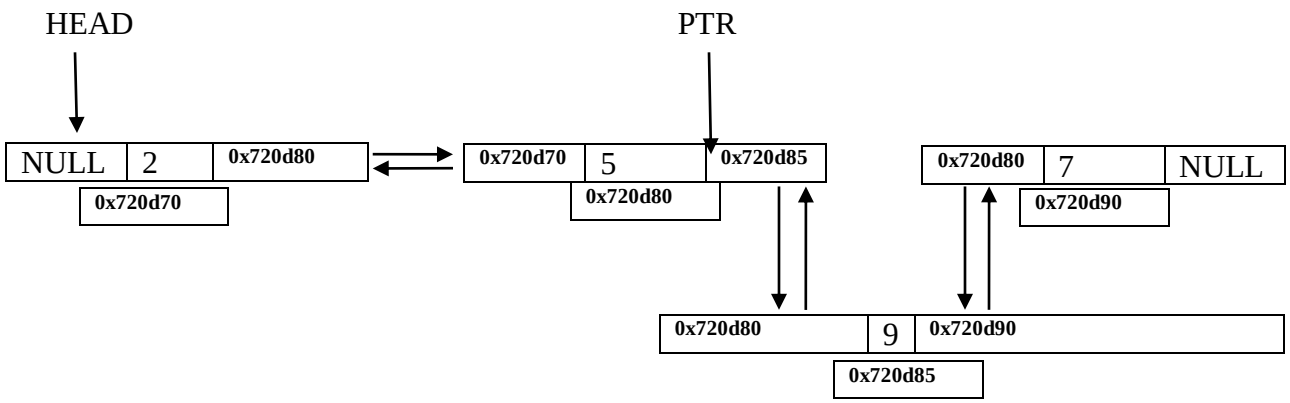
2. Take a pointer variable PTR and make it point to the first node of the list.



3. Move PTR further until the data part of PTR = value after which the node has to be inserted.



4. Insert the new node between PTR and the node succeeding it.



Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 12

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL→ NEXT

Step 4: SET NEW_NODE →DATA = VAL

Step 5: PTR=HEAD

Step 6: Repeat Step 7 while PTR →DATA! = NUM

Step 7: SET PTR = PTR→ NEXT

[END OF LOOP]

Step 8: SET NEW_NODE→NEXT = PTR→NEXT

Step 9: SET NEW_NODE→PREV=PTR

Step 10: SET PTR→ NEXT =NEW_NODE

Step 11: SET PTR→ NEXT→PREV=NEW_NODE

Step 12: EXIT

Explanation:

In Step 5, we take a pointer PTR and initialize it with HEAD. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

Assignment:

Q. Explain the insertion of the new node before a given node of the doubly linked list.

Q. Explain the deleting a node from a doubly Linked List

In this section, we will see how a node is deleted from an already existing doubly linked list. We will take four cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

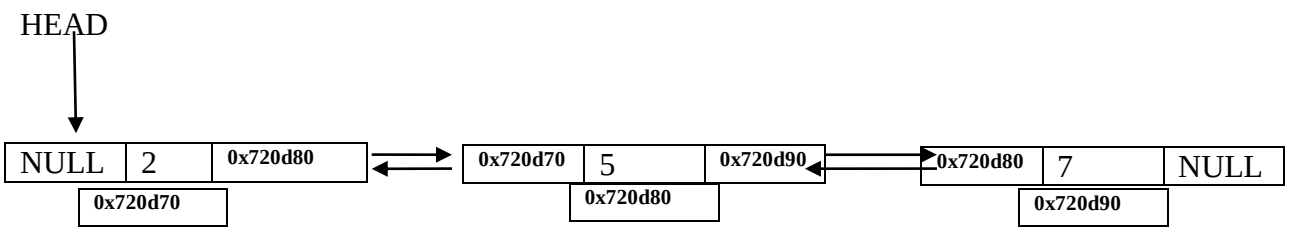
Case 2: The last node is deleted.

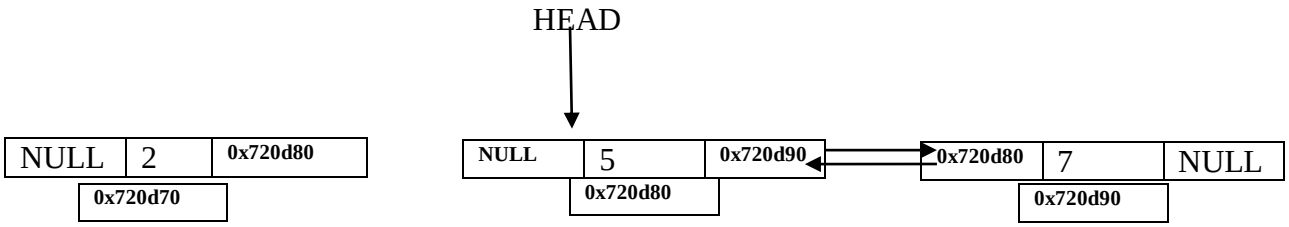
Case 3: The node after a given node is deleted.

Case 4: The node before a given node is deleted

Q1. Explain the deleting of the first node from a doubly linked list.

1. Free the memory occupied by the first node of the list and makes the second node of the list as the HEAD node.





Algorithm:

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 6

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET HEAD= HEAD→ NEXT

Step 4: SET HEAD→ PREV = NULL

Step 5: FREE PTR

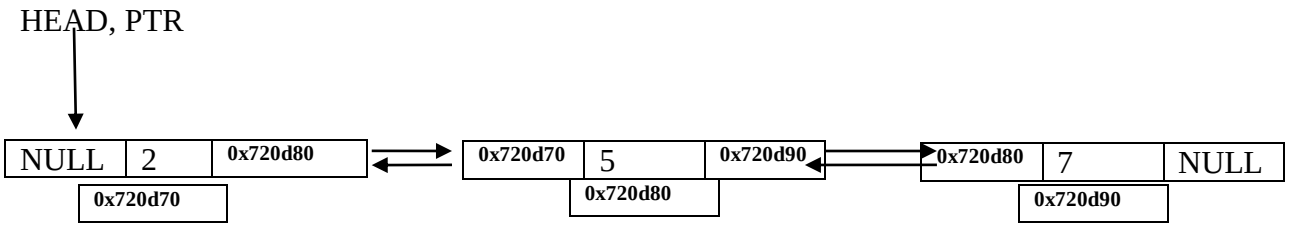
Step 6: EXIT

Explanation:

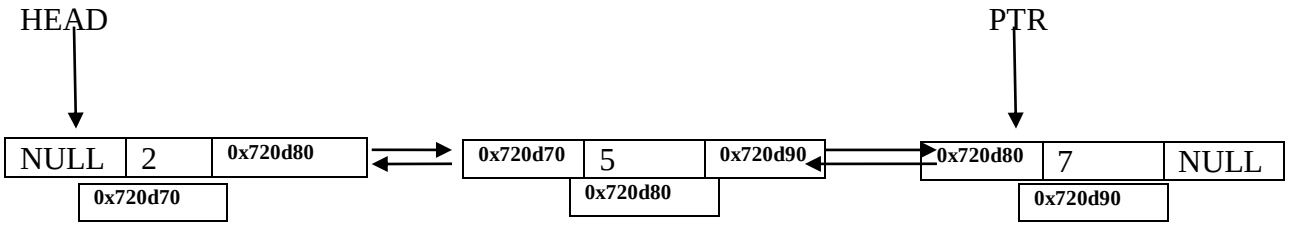
In Step 1 of the algorithm, we check if the linked list exists or not. If HEAD=NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with HEAD that stores the address of the first node of the list. In Step 3, HEAD is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

Q1. Explain the deleting of the last node from a doubly linked list.

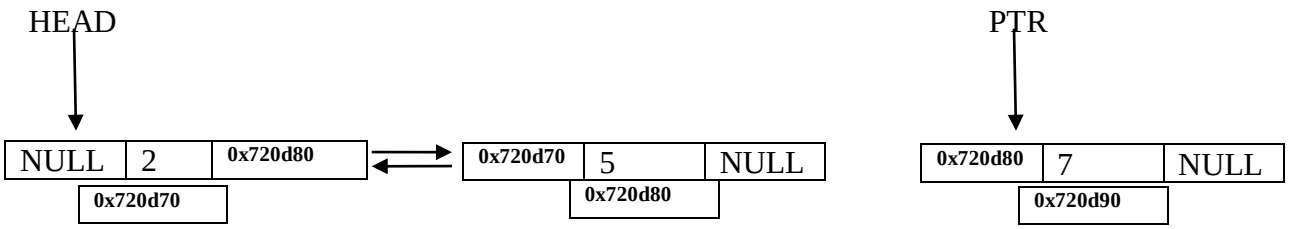
1. Take a pointer variable PTR that points to the first node of the list.



2. Move PTR so that it now points to the last node of the list.



3. Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node.



Algorithms:

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 7

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Step 4 while PTR →NEXT! = NULL

Step 4: SET PTR = PTR →NEXT

[END OF LOOP]

Step 5: SET PTR→ PREV→ NEXT = NULL

Step 6: FREE PTR

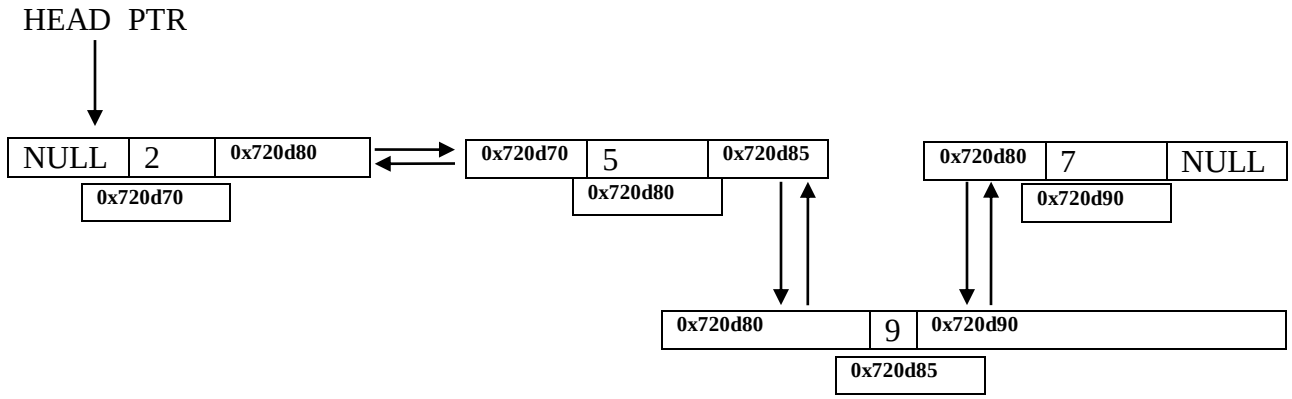
Step 7: EXIT

Explanation:

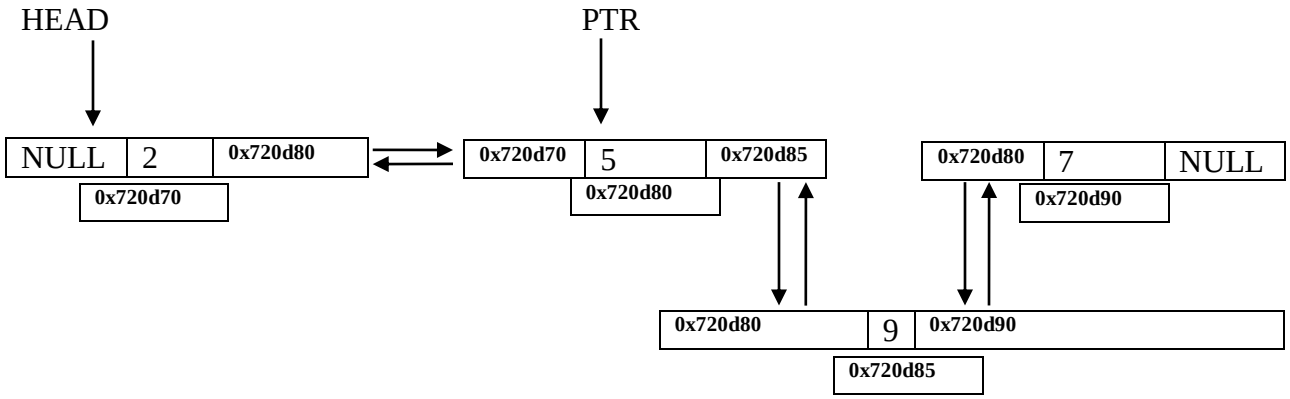
In Step 2, we take a pointer variable PTR and initialize it with HEAD. That is, PTR now points to the first node of the linked list. The while loop traverse through the list to reach the last node. Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node. To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

Q1. Explain the deleting of the node after a given node from a doubly linked list.

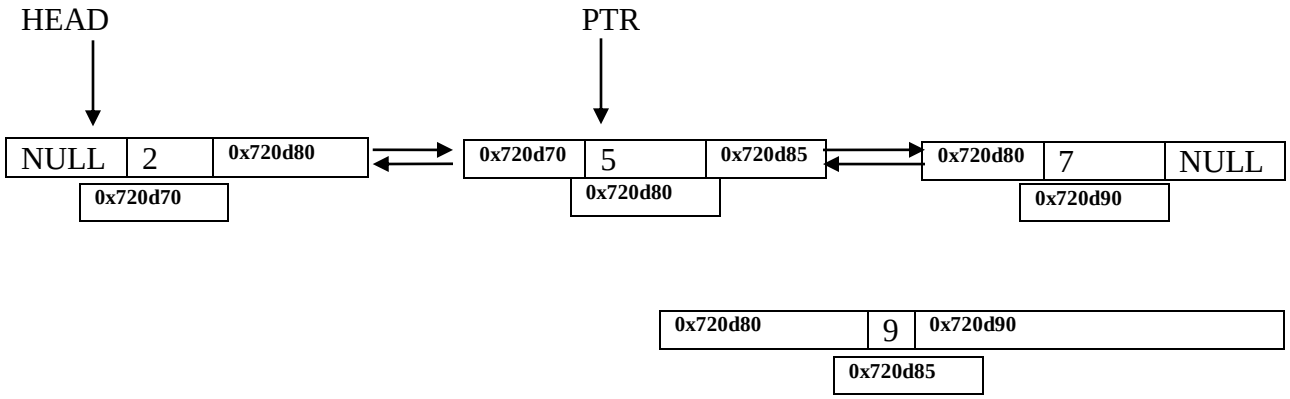
1. Take a pointer variable PTR and make it point to the first node of the list.



2. Move PTR further so that its data part is equal to the value after which the node has to be inserted.



3. Delete the node succeeding PTR.



Algorithm

Step 1: IF HEAD= NULL

Write UNDERFLOW

Go to Step 9

[END OF IF]

Step 2: SET PTR =HEAD

Step 3: Repeat Step 4 while PTR →DATA! = NUM

Step 4: SET PTR = PTR →NEXT

[END OF LOOP]

Step 5: SET TEMP = PTR→ NEXT

Step 6: SET PTR→ NEXT = TEMP →NEXT

Step 7: SET TEMP →NEXT→ PREV = PTR

Step 8: FREE TEMP

Step 9: EXIT

Explanation

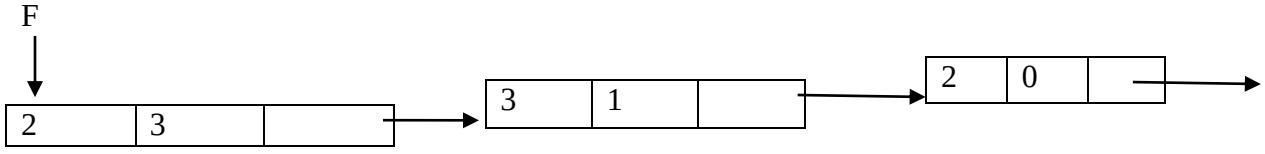
In Step 2, we take a pointer variable PTR and initialize it with HEAD. That is, PTR now points to the first node of the doubly linked list. The while loop traverse through the linked list to reach the given node. Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

APPLICATION: ADDITION OF TWO POLYNOMIALS

Consider a polynomial function f(x)= a(n)x^n+ a(n-1)x^n-1+a(n-2)x^n-2+……+a1x^1+a Here a(n) are the coefficient and x^n are the exponents so for linked representation of the polynomials we must have fields as:

Coeff	Expo	link

Suppose, F=2x^3+3x+2 then it is represented as a:



Algorithms:

Let P and Q are two polynomial then we have

1. Read the number of terms in the first polynomial, P.
2. Read the coefficient and exponents of the first polynomial.
3. Read the number of terms in the second polynomial Q.
4. Read the coefficient and exponents of the second polynomial.
5. Set the temporary pointers p and q to visit the two polynomial
6. Compare the exponents of two polynomial from first node
7.
    a. If exponent values are same then add the coefficient and store in the resultant linked list and move both pointers to next node.

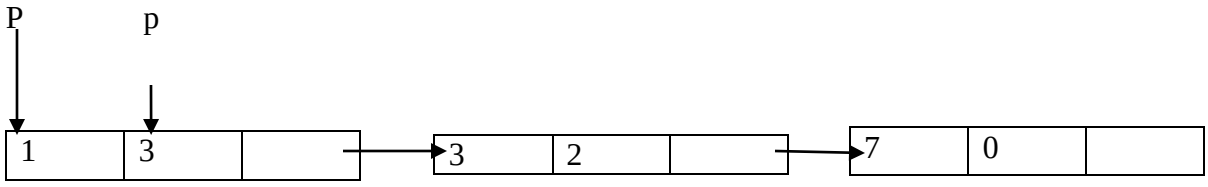
    b. If the exponent of the scanned term in the P polynomial is less than the exponent of current scanned term in Q then put the current exponents and coefficient of Q in resultant list and move the pointer q to the next node.

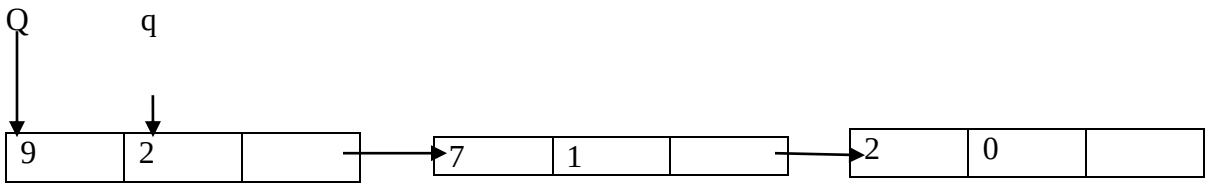
    c. If the exponent of the scanned term in the P polynomial is greater than the exponent of current scanned term in Q then put the current exponents and coefficient of P in resultant list and move the pointer p to the next node.

    d. Append the remaining nodes of either of the polynomials to get the resultant linked list.

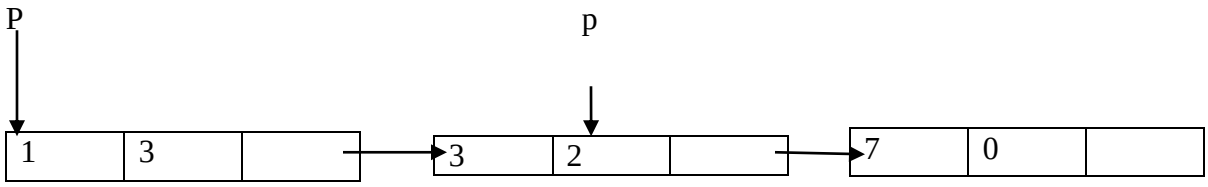
Q. Explain in details about the addition of two polynomials of degree 3 and degree 2.

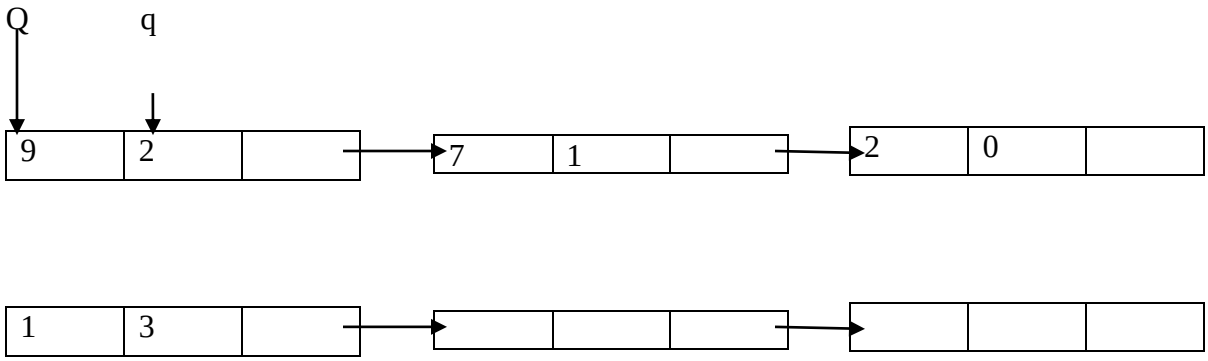
Let us consider two polynomial P=1x^3+3x^2+7 and Q=9x^2+7x+2 with pointer p and q and represented in linked list as:



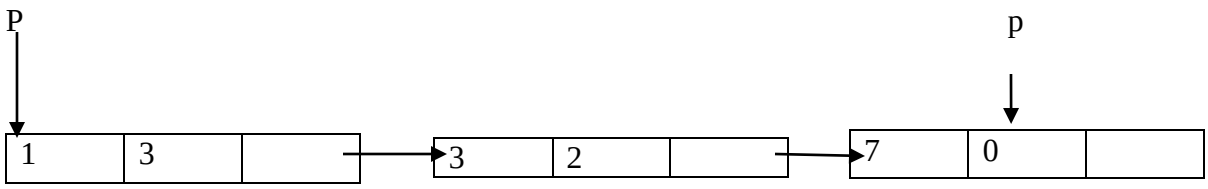


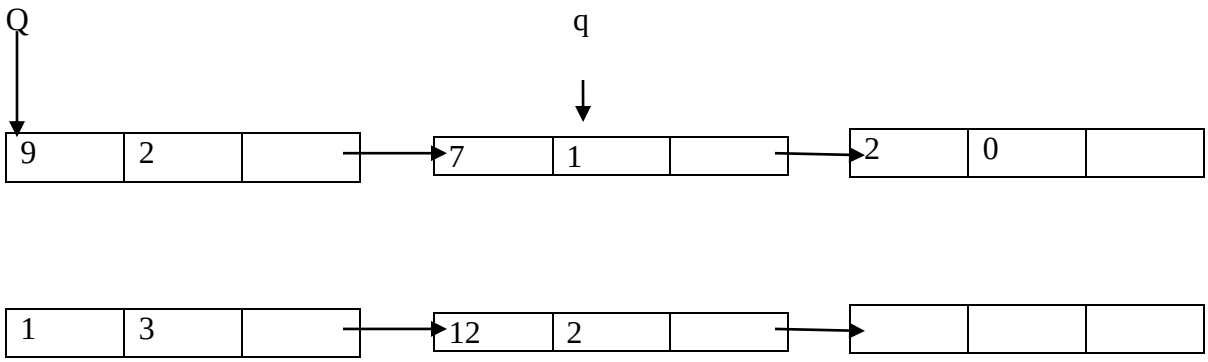
Step 1: Compare the exponent of p and corresponding exponent of q. Here, q has less value so adding the p value in resultant node and moving the p pointer to next node.



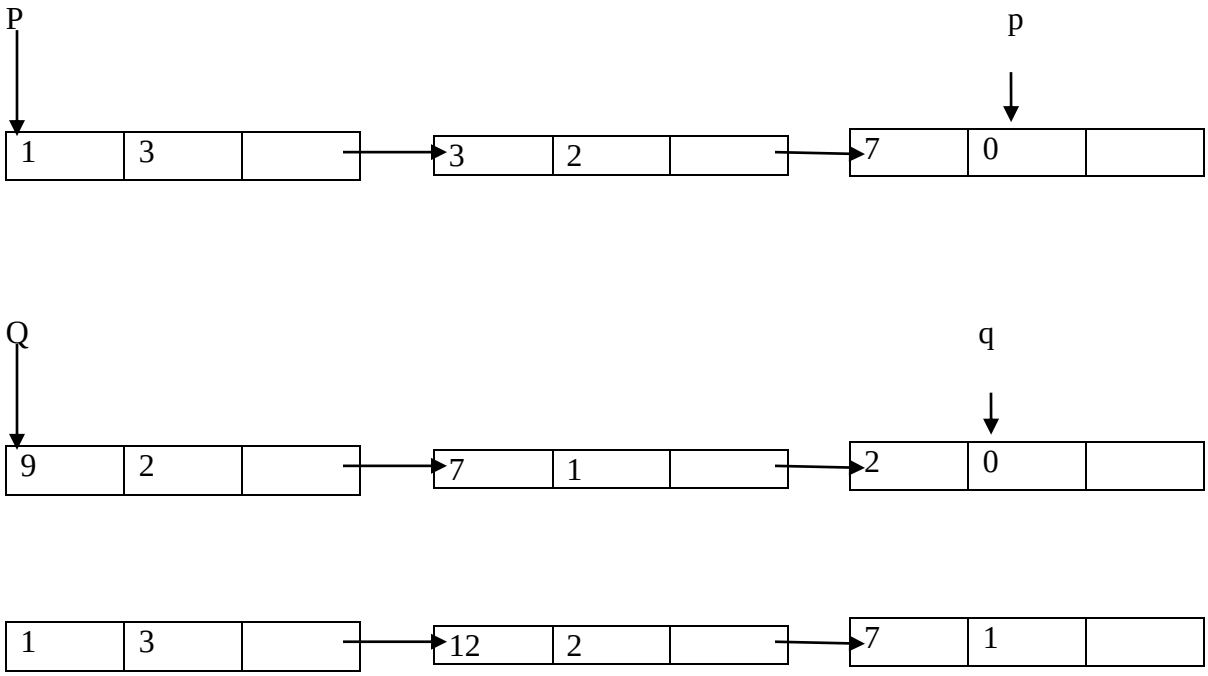


Step 2: Compare the exponent of p and corresponding exponent of q. Here, q has equal value as p so adding the both coefficient value in resultant node and moving the both pointer to next node.

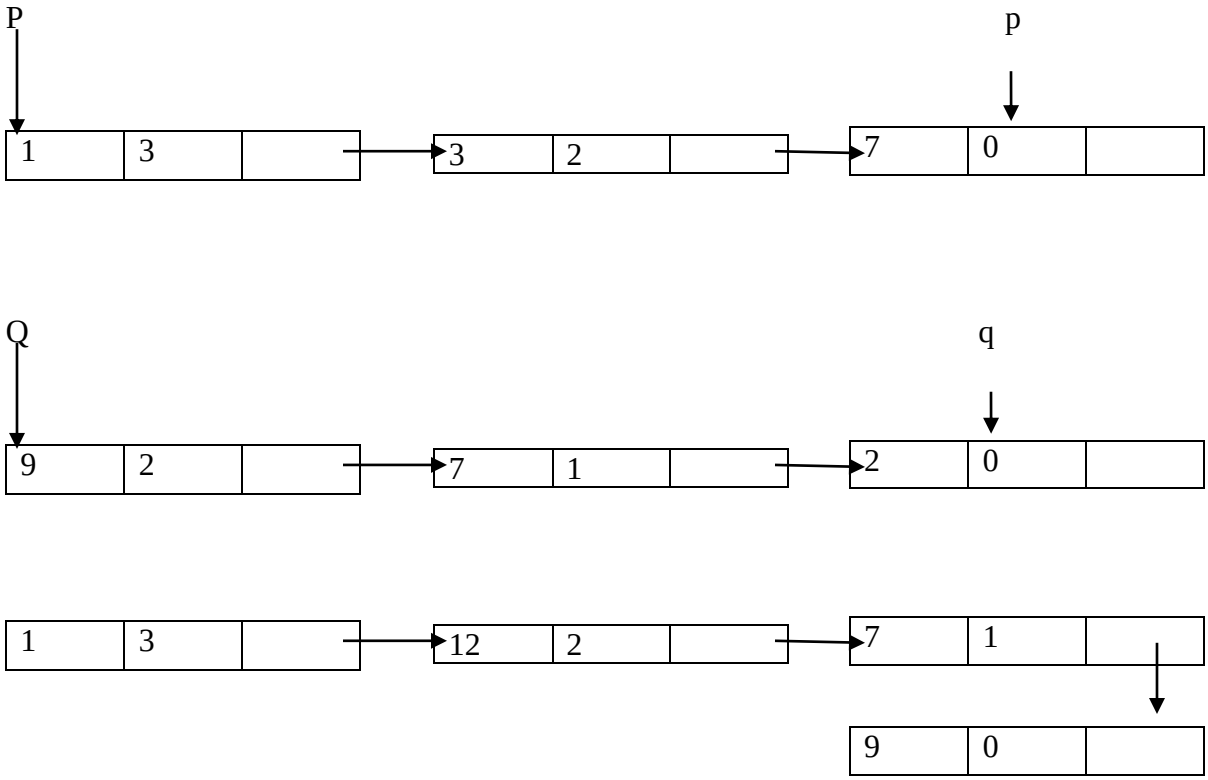




Step 3: Compare the exponent of p and corresponding exponent of q. Here, p has less value so adding the q value in resultant node and moving the q pointer to next node.



Step 4: Compare the exponent of p and corresponding exponent of q. Here, q has equal value as p so adding the both coefficient value in resultant node and moving the both pointer to next node.



So the resultant polynomial is: x^3+12x^2+7x+9

Linked stack and queue(Dyanamic implementation of stack and queue)

Q. Explain the dynamic implementation of the stack.

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. 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. The stack implemented using linked list can work for unlimited number of values. That mean, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation.

Algorithm for push and pop

1. Push (value) - Inserting an element into the Stack.

We can use the following steps to insert a new node into the stack.

Step 1: Create a new Node with given value.

Step 2: Check whether stack is Empty (top == NULL)

Step 3: If it is Empty, then set new Node → next = NULL.

Step 4: If it is Not Empty, then set new Node → next = top.

Step 5: Finally, set top = new Node.

2. pop() - Deleting an Element from a Stack
We can use the following steps to delete a node from the stack...

Step 1: Check whether stack is Empty (top == NULL).

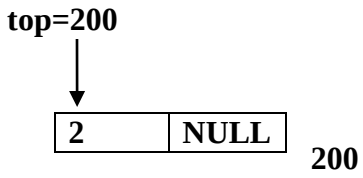
Step 2: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function

Step 3: If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.

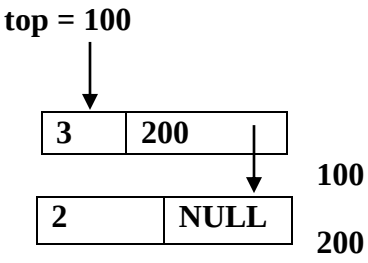
Step 4: Then set 'top = top → next'.

Step 5: Finally, delete 'temp' (free(temp)).

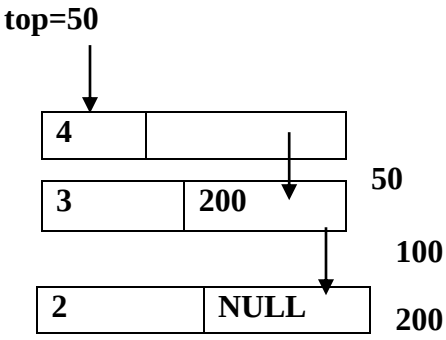
Graphically,



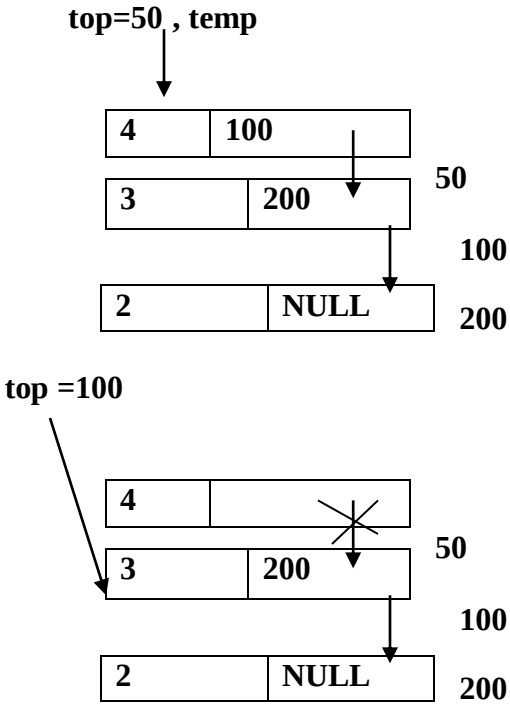
Push (3),



Push(4),



Pop(),



Q. Explain the dynamic implementation of the queue.

The major problem with the queue implemented using array is, it will work for only fixed number of data. That mean, the amount of data must be specified in the beginning itself. Queue using 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 using linked 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 of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.

Algorithm

1. **Enqueue- Inserting an Element into Queue**
   We can use the following steps to insert a new node into the queue...

   Step 1: Create a new Node with given value and set 'new Node → next' to NULL.

   Step 2: Check whether queue is Empty (rear == NULL)

   Step 3: If it is Empty then, set front = new Node and rear = new Node.

   Step 4: If it is Not Empty then, set rear → next = new Node and rear = new Node**.**

2. **Dequeue- Deleting an Element from Queue**

   We can use the following steps to delete a node from the queue.

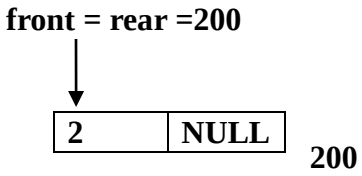
   Step 1: Check whether queue is Empty (front == NULL).

   Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function

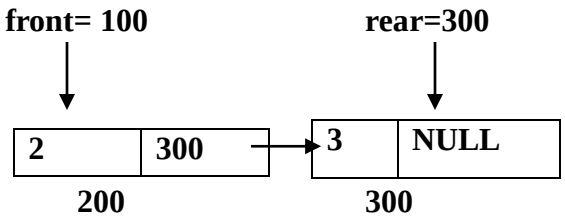
   Step 3: If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.

   Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).

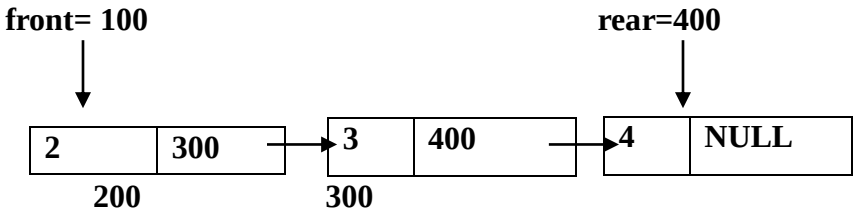
Graphically,



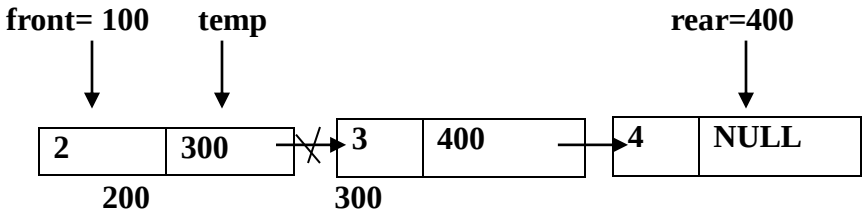
Enqueue(3),

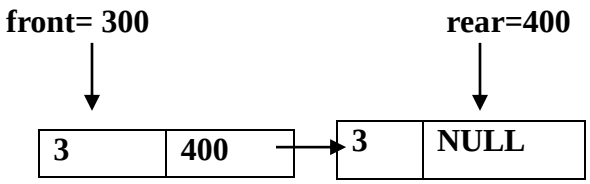


Enqueue(4),



Dequeue(),





Q. Define List: A list or sequence or contiguous list is a data structure that implements an ordered collection of values, where the same value may occur more than once. Each value in the list is called an item, or an entry or element of the list. A list can often be constructed by writing the items in sequence, separated by commas, semicolons, or spaces, within a pair of delimiters such as parenthesis'()', brackets'[]', braces'{}', or angle brackets'<>'.It differs from stack and queue in such a way that additions and removals can be made at any positions in the list.

Static list Structure:

1. Static data structures are of fixed size (eg: array) i.e. the memory allocated remains same throughout the program execution i.e the size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

2. The static implementation allows faster access to elements but is expensive for insertion/deletion operations. i.e Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

3. In the case of static data structures, the memory space is allocated to the actual operations on the list. Hence, the memory may go wasted or may be insufficient in cases. So, static implementation requires the knowledge of the exact amount of data in advance. If there is no certainty about the amount of data, dynamic implementation is to be used.

Dynamic list Structure:

1. Dynamic data structures, on the other side, have flexible (eg: linked list) size i.e. they can grow or shrink as needed to store data during program runtime.
    a. For example, suppose we maintain a sorted list of IDs in an array id[]. Id[] = {1000, 1010, 1050, 2000, 2040, ….} And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

2. In the case of dynamic implementation, the access to elements is slower but insertion/deletion operations are faster.

3. Memory is optimized.

Q. Differentiate static and dynamic implementation of the list with suitable example.

Static List	Dynamic List
Array Implementation	Linked list implementation
Static size of list	Dynamic size of list
Suitable if fewer nodes are used	Suitable if large no. of nodes are used
Programming implementation is easy	Programming implementation is difficult
Less computation time	More computation time

Assignment: Circular Linked list

1. **Insertion and deletion of node at the begin**
2. **Insertion and deletion of node at the end.**
3. **Insertion and deletion of node after/before given node.**

By: Er. Bishwas Pokharel, Lecturer

Assignment: Explain about the static linked list.

Static Linked list

Static Linked List is a data structure that stores data in static arrays. It has fixed no. of nodes and each nodes contains two section one is data and another is next index.

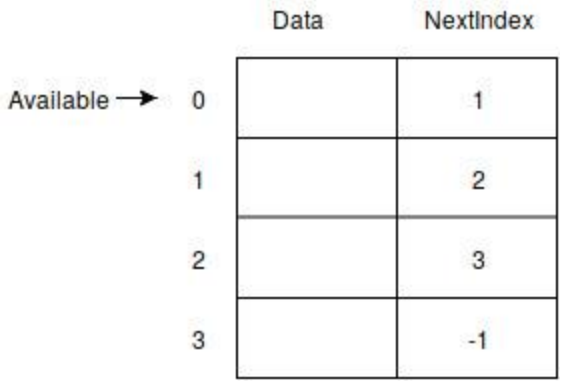
Assume that the data type for the linked list is character. What we can do is to implement such a linked list, we can declare a really large array. We will define some MaxSize and declare an array such as Nodes[MaxSize].

Suppose MaxSize = 4

initially,

Head= -1

Available = 0



Here, Head points starting node and Available points blank node. All blanked nodes are linked

and -1 in next index of Nodes[3] indicates that the Nodes[3] doesn't have any next node.

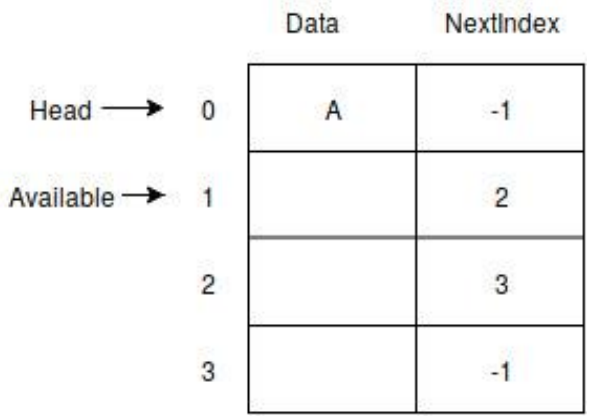
Insertion

For inserting A at begining,

1) Set Head= Available

2) Set Available = Nodes[Available].NextIndex

3) Set Nodes[Head].NextIndex = -1



Here, Nodes[Head].NextIndex = -1 because nodes[0] doesn't have any next node.

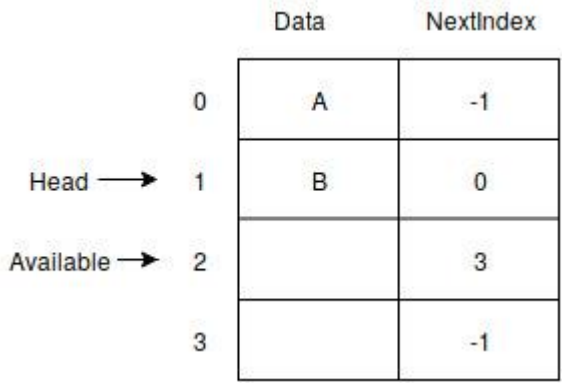
Similiarly, For inserting B at begining,

1) Set temp = Head

2) Set Head = Available

3) Set Available = Nodes[Available].NextIndex

4) Set Nodes[Head].NextIndex= temp



Here, Nodes[Head].NextIndex = 0 because we add B before A. So, node of B must points node of A.

Deletion

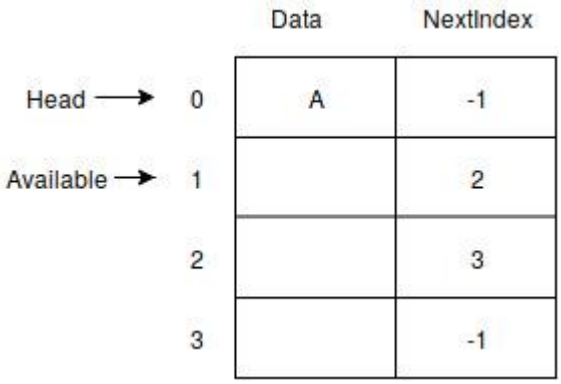
For removing B,

1) temp = Available

2) Available = Head

3) Head= Nodes[Head].NextIndex

4) Nodes[Available].NextNode = temp

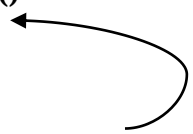


(Submitted by: Subas Shrestha, Student of Kathford College)

Define Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

a. Directly recursive: method that calls itself,

a.
```
int A()
{
…..
A();

}
```


b. Indirectly recursive: method that calls another method and eventually results in the original method call,

b.
```
int A()
 {
   B();
  }
int B()
{
   A();
}
```


Assignment: Tail and non tail recursion

THE PROS AND CONS OF RECURSION

Recursion can be very useful in computer programming, in game development, but there are a few pros and cons that all programmers should keep in mind when learning about recursion.

The pros are:

1. **It is conceptually easier to code.**
2. **It is easier to maintain and modify in some situations.**

The cons are:

1. **There is overhead with the calling of functions that can quickly add up with recursion.**
2. **When enough functions and their arguments are pushed onto the stack, it can cause a stack overflow when the recursive function gets to a point where it exceeds the system's capacity.**
3. **Using recursion can be less effective in performance than using loops.**

In cases where the efficiency loss is great, you should avoid using recursion if it becomes a serious source of a bottleneck in the application.

Q. Mention some application of recursion.

Application:

- **a.** Recursion is applied to problems which can be broken into smaller parts, each part looking similar to the original problem.
- **b.** Recursion is used to implement algorithms on tree.
- **c.** Recursion is used in parsers and compilers
- **d.** Recursion is used in networking
- **e.** Recursion is used to guarantee the correctness of an algorithm.

Q. Mention the Base case of Recursion.

VERY IMPORTANT

a. Stops the recursion (prevents infinite loops)

b. Solved directly to return a value without calling the same method again.

Fibonacci sequence:

As we know for any given number say N the nth term of Fibonacci series can be represented as, F(N) = Nth term in Fibonacci series

F(n) = F(n-1) + F(n-2)

Where, F(0) = 0 and F(1) = 1

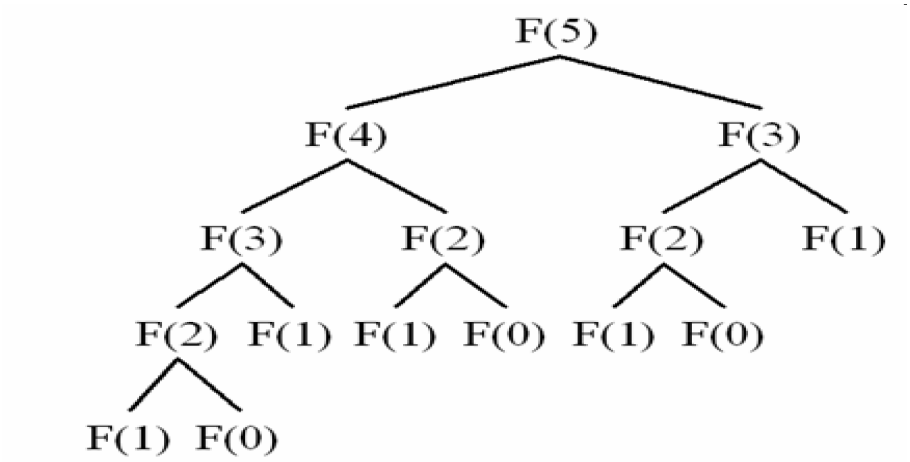
Hence, the recursive function will take one parameter that is number itself and the base criteria will be designed based on the number.

Algorithm

```
int Fibonacci (int number)
 {
        if (number== 0)
         {
                return 0;
        }
        else if (number== 1)
        {
                 return 1;
        }
         else
         {
                return fibonacci(number- 1)+ fibonacci(number- 2); }
```

From above algorithm it is clear that after each call to Fibonacci function we reduced the value of number by 1. Once it reached to base function, the function returns based on base criteria.

Recursion tree for Fibonacci F(5).



Towers of Hanoi: The legend of the temple of Brahma)

a. 64 golden disks on 3 pegs

b. The universe will end when the priest move all disks from the first peg to the last

c. Only move one disk at a time

d. A move is taking one disk from a peg and putting it on another peg (on top of any other disks)

e. Cannot put a larger disk on top of a smaller disk

f. With 64 disks, at 1 second per disk, this would take roughly 585 billion year.

The Rules

a. Only move one disk at a time.

b. A move is taking one disk from a peg and putting it on another peg (on top of any other disks).

c. Cannot put a larger disk on top of a smaller disk.

Recursive algorithm

To Move n disks from A to C using B

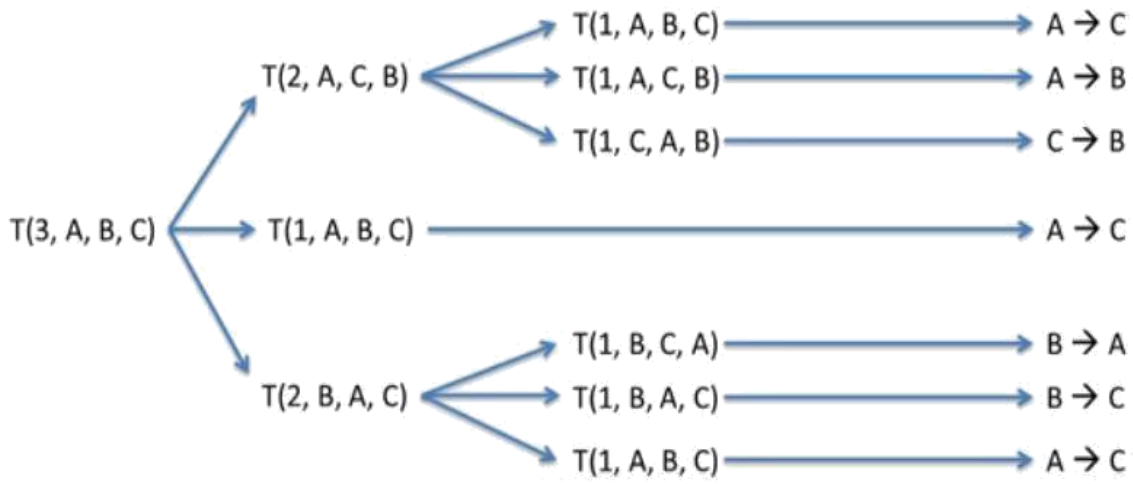
a. move top n-1 disks from A to B using C

b. move bottom disks from A to C

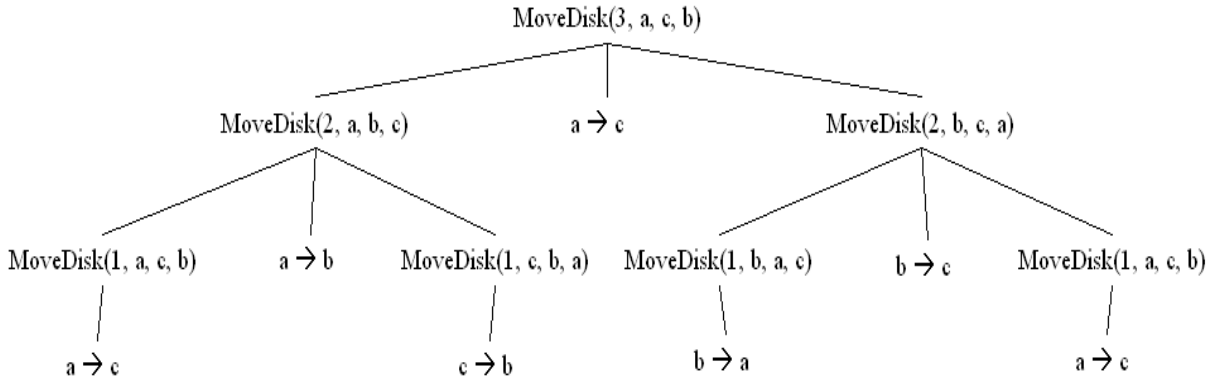
c. move n-1 disks from B to C using A

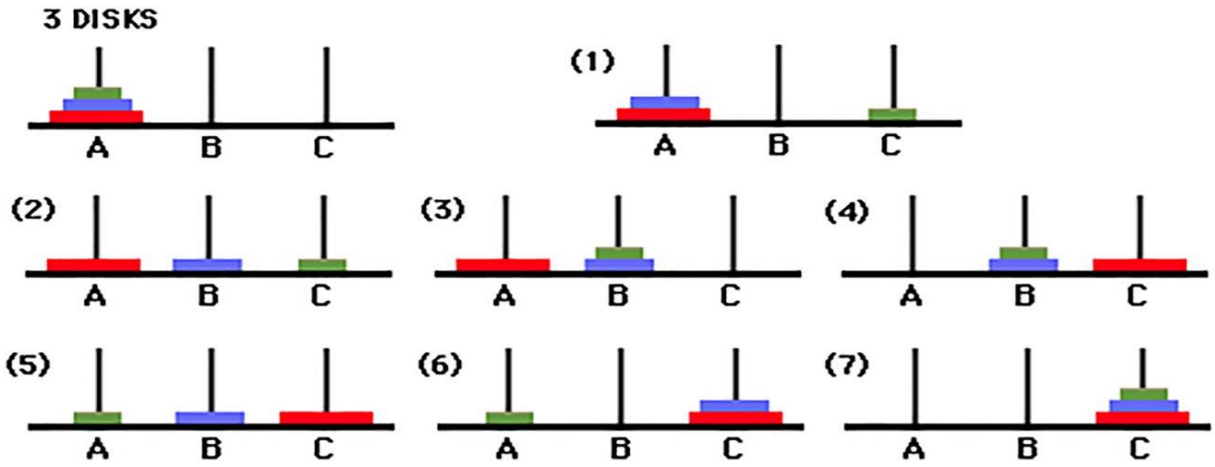
T(n) = 2^n -1, the running time is exponential in n

Recursion Tree for TOH



Or,





Assignment: Draw recursion tree for Tower of Hanoi for 4 disks.

Q. Compare recursive and non-recursive function(iterative).

Property	**Recursion**	**Iterative**
Definition	**Function call itself**	**A set of instructions repeatedly executed.**
Application	**For function**	**For loops**
Termination	**Through base case, where there will be no function call.**	**When the termination condition for the iterator ceases to be satisfied.**
Usage	**Used when code size needs to be small, and time complexity is not an issue.**	**Used when time complexity needs to be balanced against an expanded code size.**
Code Size	**Smaller code size**	**Larger Code Size.**
Time Complexity	**Very high(generally exponential) time complexity.**	**Relatively lower time complexity(generally polynomial-logarithmic)**

Assignment: Explain about the tail and non tail recursion.

Recursion

There are four types of recursion:

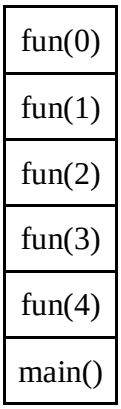
- Direct Recursion
- Indirect Recursion
- Tail Recursion
- Non-tail Recursion

Tail Recursion

A recursive function is said to be tail recursive if the recursive call is the last thing done by the function. There is no need to keep a record of previous state. For eg:

```
void fun(int n){
        if(n == 0)
                return;
        else
                cout<<n<<" ";

        return fun(n-1);

}
int main(){
        fun(4);
        return 0;
}
```

In the above example, the recursive call is the last thing done by the function. During the execution of this program the activation of each function is recorded as in the following diagram.



Fig(a): Activation Call of each function

In this program, the activation call reaches to fun(0) but there is nothing left to be evaluated after recursive call so, it directly returns to fun(1), fun(1) -> fun(2), fun(2) -> fun(3), fun(3) -> fun(4) , fun(4) -> main() without any evaluation and activation stack becomes empty.

Output:

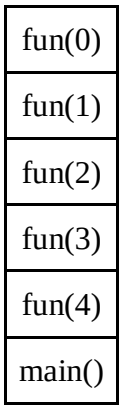
4 3 2 1

Non-tail Recursion

A recursive function is said to be non-tail recursive if there is some operation after recursive call i.e. recursive call is not the last thing done by the function. There is a need to keep a record of previous state. For eg:

```
void fun(int n){
        if(n == 0)
                return;

        fun(n-1);

        cout<<n<<" ";

}
int main(){
        fun(4);
        return 0;
}
```

In the above example, the recursive call is not the last thing done by the function. During the execution of this program the activation of each function is recorded as in the following diagram.



Fig(a): Activation Call of each function

But in this program, the activation call reaches to fun(0) and there is something left to be evaluated after recursive call i.e printing the value so, it returns to fun(1) and prints the value and similarly other recursive call and then finally to main() call and activation stack becomes empty.

Output:

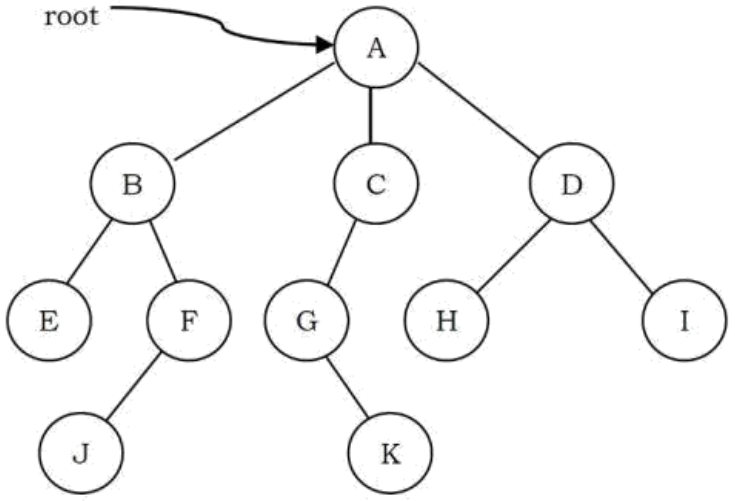
1 2 3 4

(Submitted by: Alish Dahal, Student of Kathford College)

Trees:

Tree is an example of a nonlinear data structure. A tree structure is a way of representing the hierarchical nature of a structure in a graphical form. In trees ADT (Abstract Data Type), the order of the elements is not important. If we need ordering information, linear data structures like linked lists, stacks, queues, etc. can be used.

Tree Vocabulary:



The root of a tree is the node with no parents. There can be at most one root node in a tree (node A in the above example).

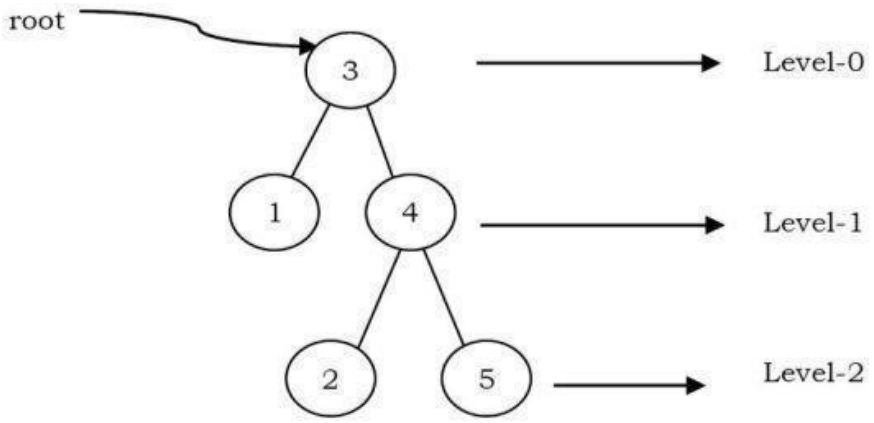
a. An edge refers to the link from parent to child (all links in the figure).

b. A node with no children is called leaf node (E,J,K,H and I).

c. Children of same parent are called siblings (B,C,D are siblings of A, and E,F are the siblings of B).

d. A node p is an ancestor of node q if there exists a path from root to q and p appears on the path. The node q is called a descendant of p. For example, A,C and G are the ancestors of k.

e. The set of all nodes at a given depth is called the level of the tree (B, C and D are the same level). The root node is at level zero.



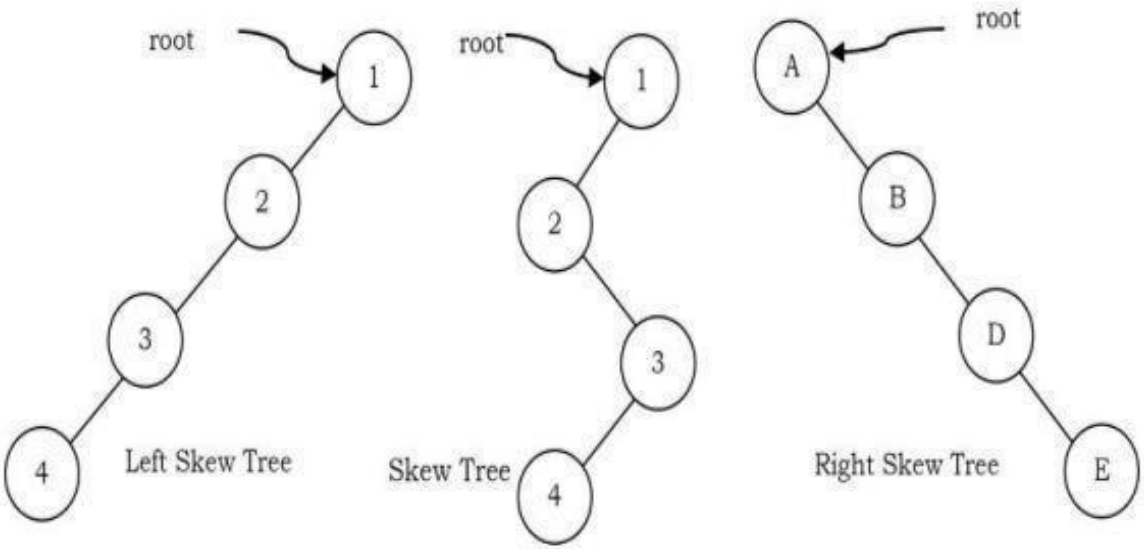
f. The depth of a node is the length of the path from the root to the node (depth of G is 2, A – C – G).

g. The height of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the height of B is 2 (B – F – J).

h. Height of the tree is the maximum height among all the nodes in the tree and depth of the tree is the maximum depth among all the nodes in the tree. For a given tree, depth and height returns the same value. But for individual nodes we may get different results.

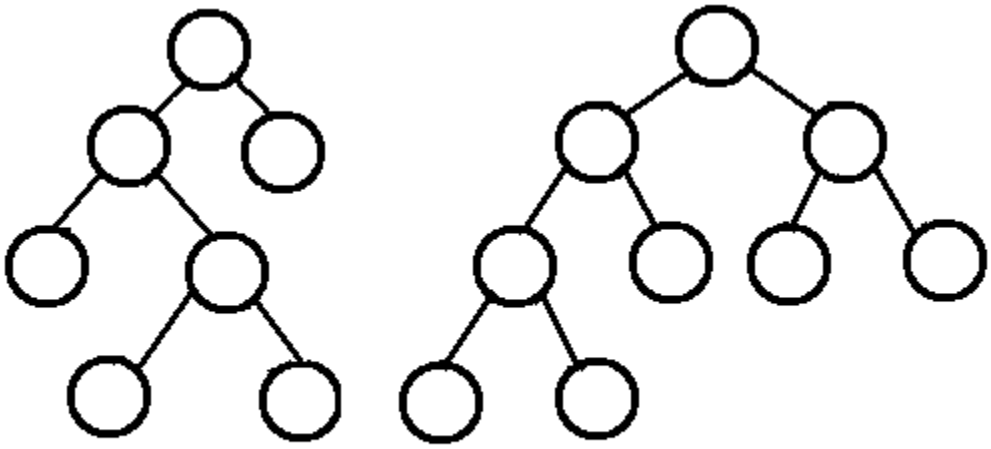
i. The size of a node is the number of descendants it has including itself (the size of the subtree C is 3).

j. If every node in a tree has only one child (except leaf nodes) then we call such trees skew trees. If every node has only left child then we call them left skew trees. Similarly, if every node has only right child then we call them right skew trees.



Binary Trees

A tree is called binary tree if each node has zero child, one child or two children. Empty tree is also a valid binary tree. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right sub trees of the root.



full tree complete tree

Types of Binary Trees

Strict Binary Tree: A binary tree is called strict binary tree if each node has exactly two children or no children.

Full Binary Tree: A binary tree is called full binary tree if each node has exactly two children and all leaf nodes are at the same level.

Complete Binary Tree: Before defining the complete binary tree, let us assume that the height of the binary tree is h. In complete binary trees, if we give numbering for the nodes by starting at the root (let us say the root node has 1) then we get a complete sequence from 1 to the number of nodes in the tree. While traversing we should give numbering for NULL pointers also. A binary tree is called complete binary tree if all leaf nodes are at height h or h – 1 and also without any missing number in the sequence.

Height, level and depth of a tree

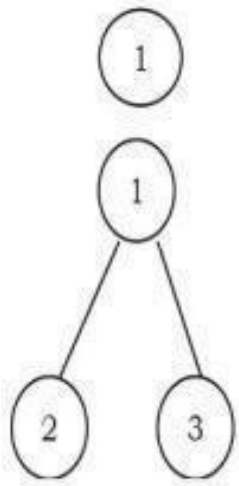
Properties of Binary Trees: For the following properties, let us assume that the height of the tree is

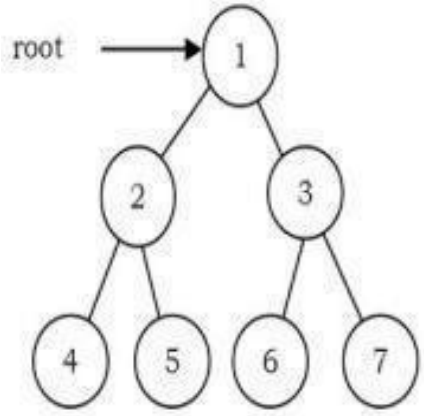
h. Also, assume that root node is at height zero.

a. The number of nodes n in a full binary tree is 2h+1 – 1. Since, there are h levels we need to add all nodes at each level [20 + 21+ 22 + ••• + 2h = 2h+1 – 1].

b. The number of leaf nodes in a full binary tree is 2h.

From the diagram we can infer the following properties:





Height	Number of nodes at level h
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$
$h = 2$	$2^2 = 4$

Operations on Binary Trees

Basic Operations: •Inserting an element into a tree •Deleting an element from a tree
•Searching for an element •Traversing the tree

Tree traversals (pre-order, post-order and in-order):

Binary Tree Traversals

In order to process trees, we need a mechanism for traversing them, and that forms the subject of this section. The process of visiting all nodes of a tree is called tree traversal. Each node is processed only once but it may be visited more than once. As we have already seen in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order. But, in tree structures there are many different ways. Tree traversal is like searching the tree, except that in traversal the goal is to move through the tree in a particular order. In addition, all nodes are processed in the traversal but searching stops when the required node is found. Traversal Possibilities Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (referred to as "visiting" the node and denoted with "D"), traversing to the left child node (denoted with "L"), and traversing to the right child node (denoted with "R"). This process can be easily described through recursion. Based on the above definition there are 6 possibilities:

1. LDR: Process left sub tree, process the current node data and then process right sub tree
2. LRD: Process left sub tree, process right sub tree and then process the current node data
3. DLR: Process the current node data, process left sub tree and then process right sub tree
4. DRL: Process the current node data, process right sub tree and then process left sub tree
5. RDL: Process right sub tree, process the current node data and then process left sub tree
6. RLD: Process right sub tree, process left sub tree and then process the current node data

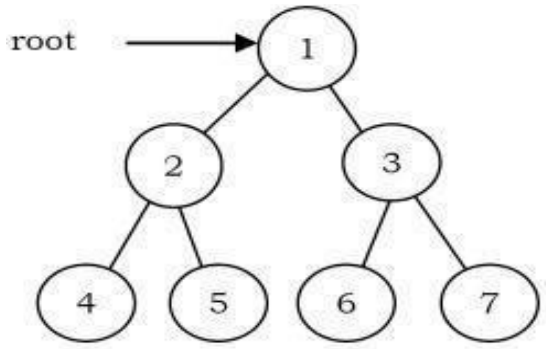
Classifying the Traversals

The sequence in which these entities (nodes) are processed defines a particular traversal method. The classification is based on the order in which current node is processed. That means, if we are classifying based on current node (D) and if D comes in the middle then it does not matter whether L is on left side of D or R is on left side of D. Similarly, it does not matter whether L is on right side of D or R is on right side of D. Due to this, the total 6 possibilities are reduced to 3 and these are:

- Preorder (DLR) Traversal
- Inorder (LDR) Traversal

- Postorder (LRD) Traversal

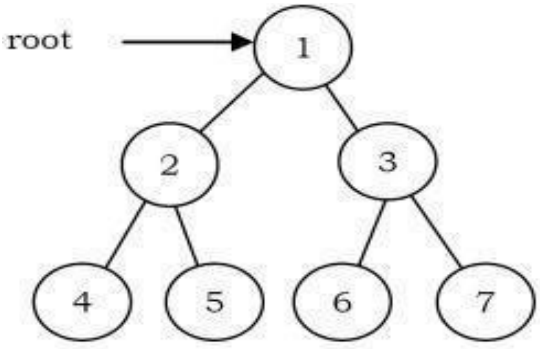
Q. Write the sequence of node in preoder, postorder and inorder traversal of given figure.



1. **PreOrder Traversal (root - leftChild - rightChild):** In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all sub trees in the tree.

   Preorder traversal is defined as follows:

   - Visit the root.
   - Traverse the left subtree in Preorder.
   - Traverse the right subtree in Preorder.

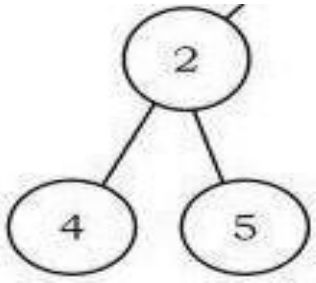


The nodes of tree would be visited in the order: **1 2 4 5 3 6 7**
Time Complexity: O(n). Space Complexity: O(n).

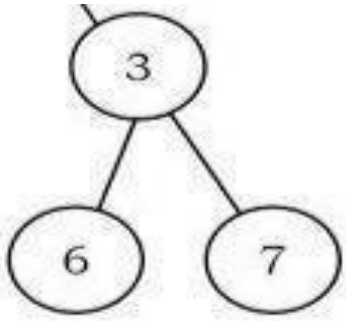
Explanation:

I. Visit root node 1.

II. Then, left sub tree as it first visits root node 2 and then it's left child 4 and then right child 5.



III. Then, right sub tree as it first visits root node 3 and then it's left child 6 and then right child 7.

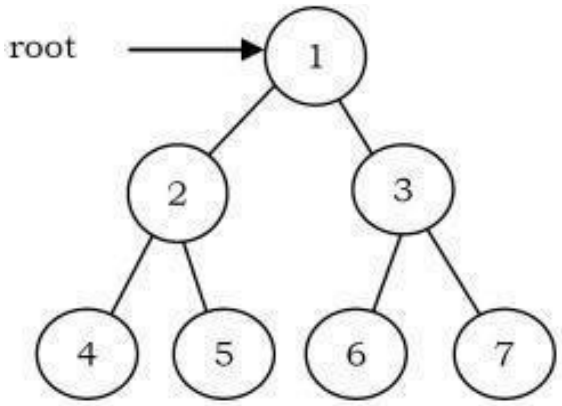


2. **In - Order Traversal (leftChild - root - rightChild):** In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

   Inorder traversal is defined as follows:

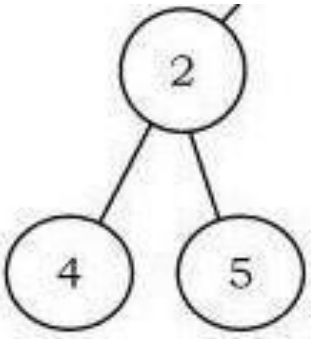
   - Traverse the left subtree in Inorder.
   - Visit the root.
   - Traverse the right subtree in Inorder.

The nodes of tree would be visited in the order: **4 2 5 1 6 3 7**
Time Complexity: O(n). Space Complexity: O(n).

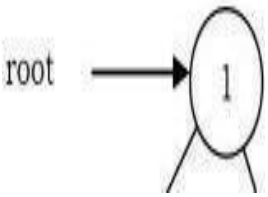


Explanation:

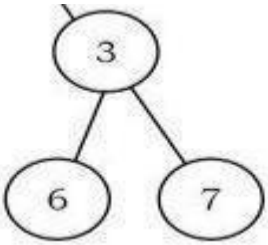
Left subtree: It visits 4 which is left child of subtree, then root node 2 and right child 5.



subtree: It visits then root node 1.



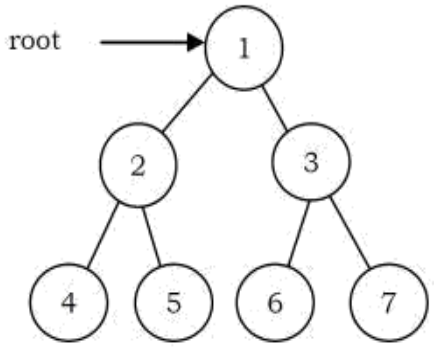
Right subtree: It visits 6 and root node 3 and then 7



3. Post - Order Traversal (leftChild - rightChild - root): In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

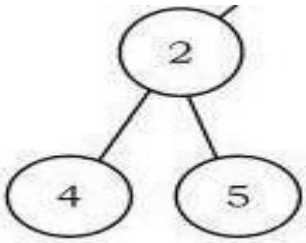
Postorder traversal is defined as follows:

- Traverse the left subtree in Postorder.
- Traverse the right subtree in Postorder.
- Visit the root.

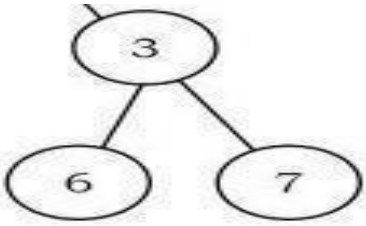


The nodes of the tree would be visited in the order: **4 5 2 6 7 3 1**
Time Complexity: O(n). Space Complexity: O(n)

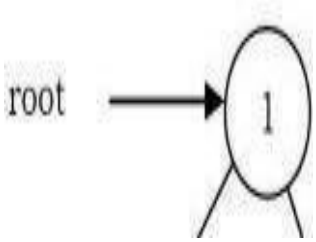
Left subtree: visit left child 4 , right child 5 and root node 2.



Right subtree: visit left child 6, right child 7 and root node 3. finally, visit root node 1.



Finally, visit root node 1.

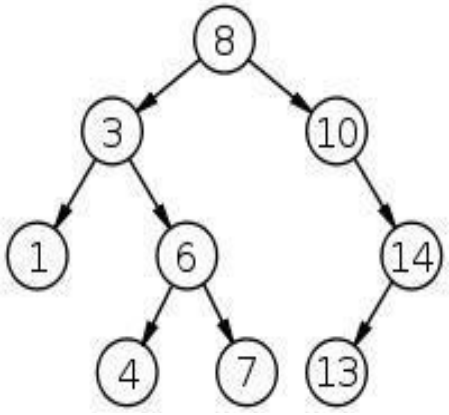


Binary Search Tree: is a node-based binary tree data structure which has the following properties:

I. The left sub tree of a node contains only nodes with keys less than the node is key.

II. The right sub tree of a node contains only nodes with keys greater than the node's key.

III. The left and right sub tree each must also be a binary search tree. There must be no duplicate nodes.



Q. Write an algorithm to search the record in Binary Search Tree.

Algorithm

1. Check, whether value in current node and searched value are equal. If so, value is found.

Otherwise,

2. If searched value is less, than the node's value:
    a. if current node has no left child, searched value doesn't exist in the BST;
    b. Otherwise, handle the left child from step 1.
3. If a new value is greater, than the node's value:
    a. if current node has no right child, searched value doesn't exist in the BST;
    b. Otherwise, handle the right child from step 1.

Q. Write an algorithm to insert the record in Binary Search Tree.

Algorithm
Starting from the root,

1. Check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,
2. If a new value is less, than the node's value:
    a. if a current node has no left child, place for insertion has been found;
    b. Otherwise, handle the left child again from step 1.
3. If a new value is greater, than the node's value:
    a. if a current node has no right child, place for insertion has been found;
    b. Otherwise, handle the right child again from step 1.

Q. Write an algorithm to delete the record in Binary Search Tree.

Remove operation on binary search tree is more complicated, than add and search. Basically, in can be divided into two stages:

a. search for a node to remove;

b. If the node is found, run remove algorithm.

Algorithm

Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is tricky. There are three cases, which are described below.

Case 1.Node to be removed has no children.

This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node. Example: Remove -4 from a BST.

Case 2.Node to be removed has one child.

It this case, node is cut from the tree and algorithm links single child (with its sub tree) directly to the parent of the removed node. Example: Remove 18 from a BST

Case 3.Node to be removed has two children.

This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example those BSTs: which has two children:

a. find a minimum value in the right sub tree;

b. Replace value of the node to be removed with found minimum. Now, right sub tree contains a duplicate!

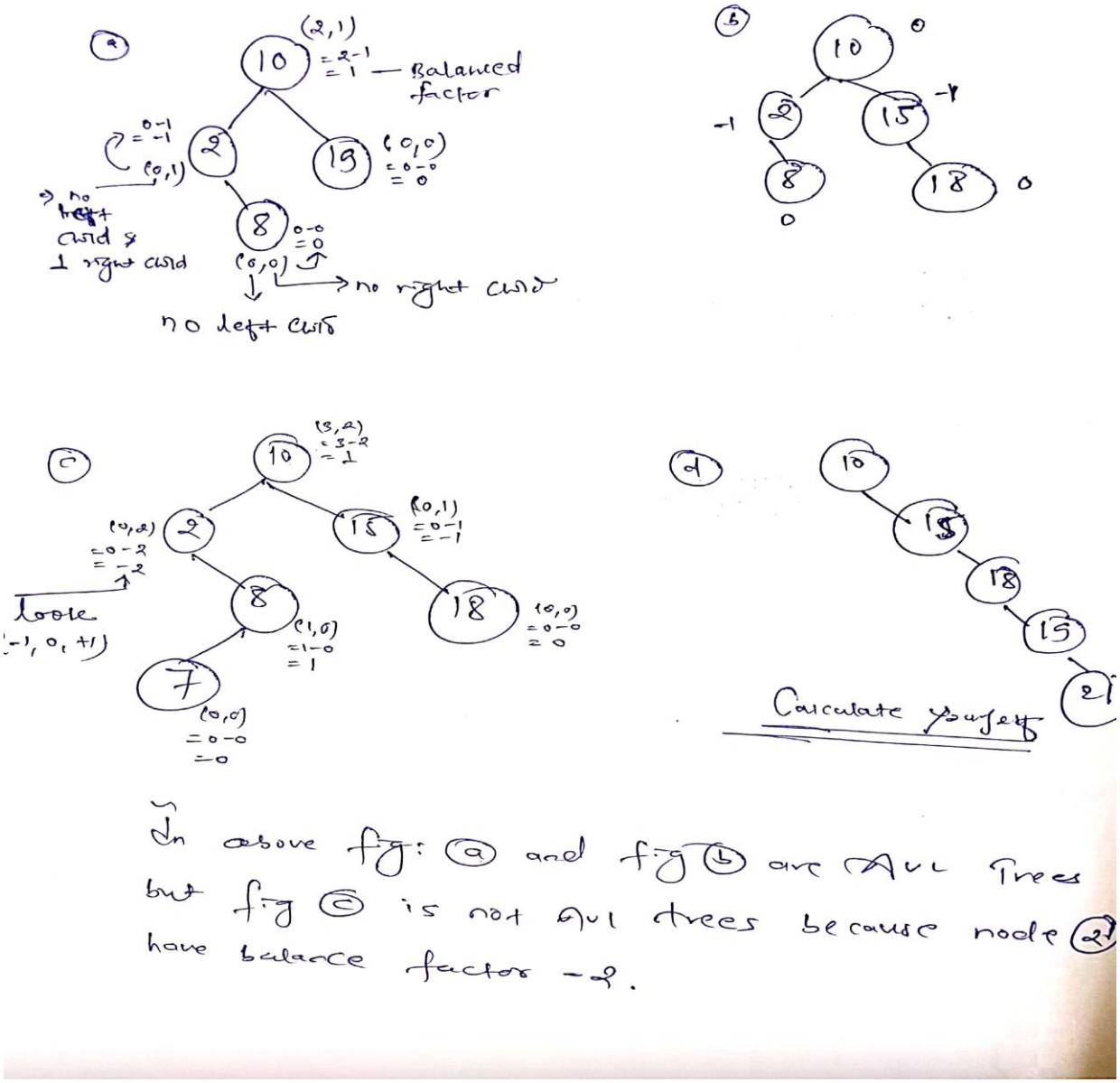
c. Apply remove to the right sub tree to remove a duplicate.

Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

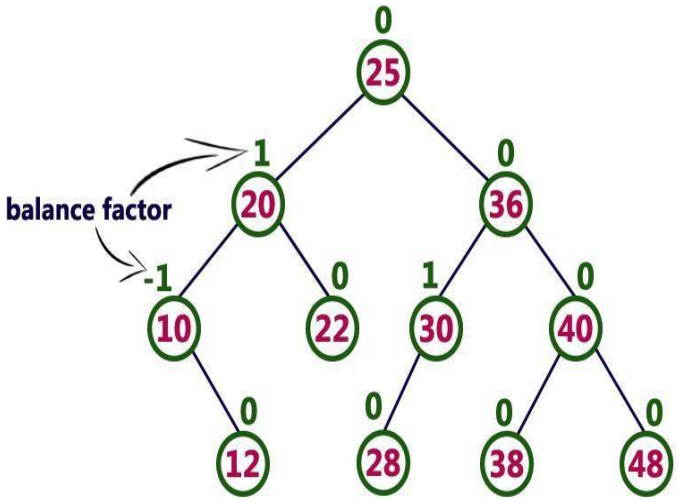
Q. Define AVL Trees:

The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis. **AVL tree is a self-balanced binary search tree**. That means, an AVL tree is also a binary search tree but it is a balanced tree. **A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1**. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as balance factor..The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree.

Balance factor = heightOfLeftSubtree – heightOfRightSubtree



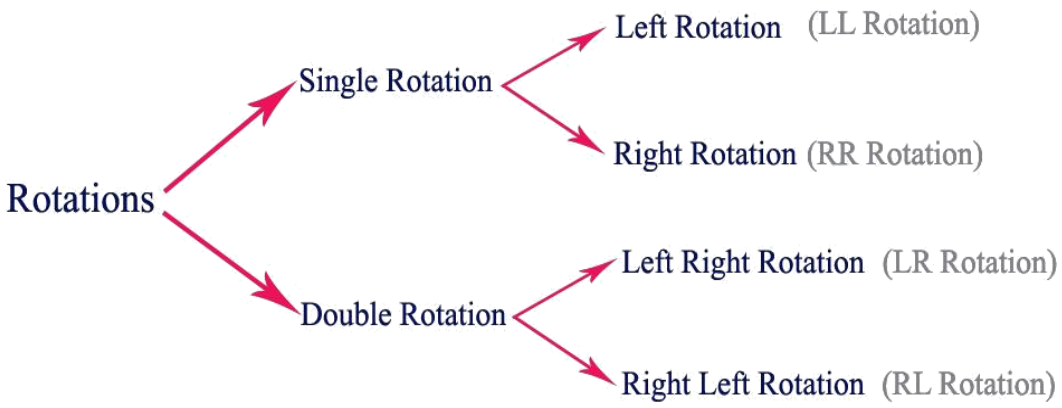
Q. Find balance factor for every node of below diagram and check result.



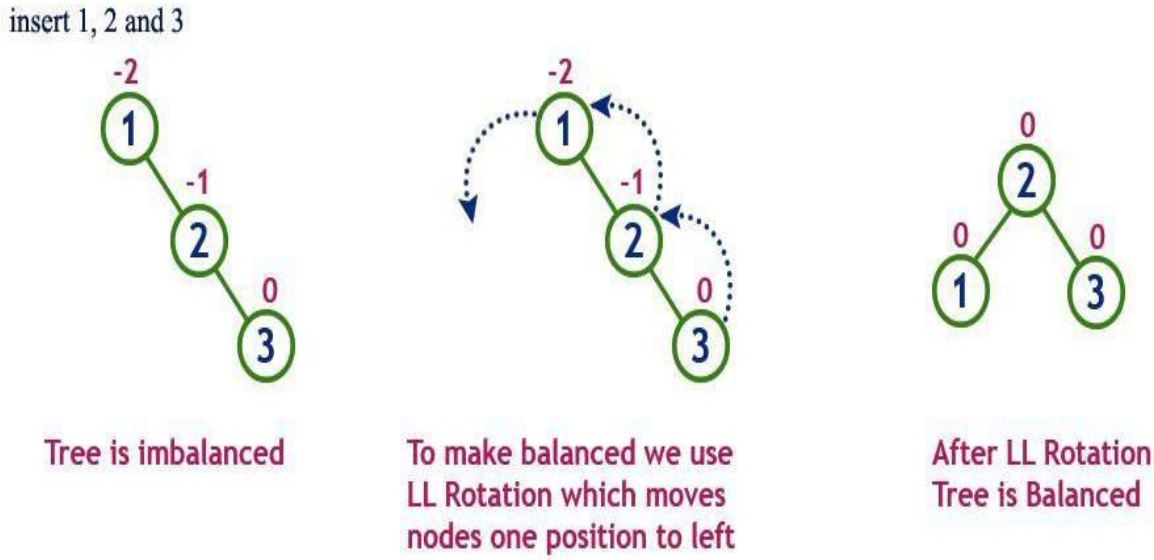
AVL Tree Rotations

In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation. i.e Rotation operations are used to make a tree balanced.

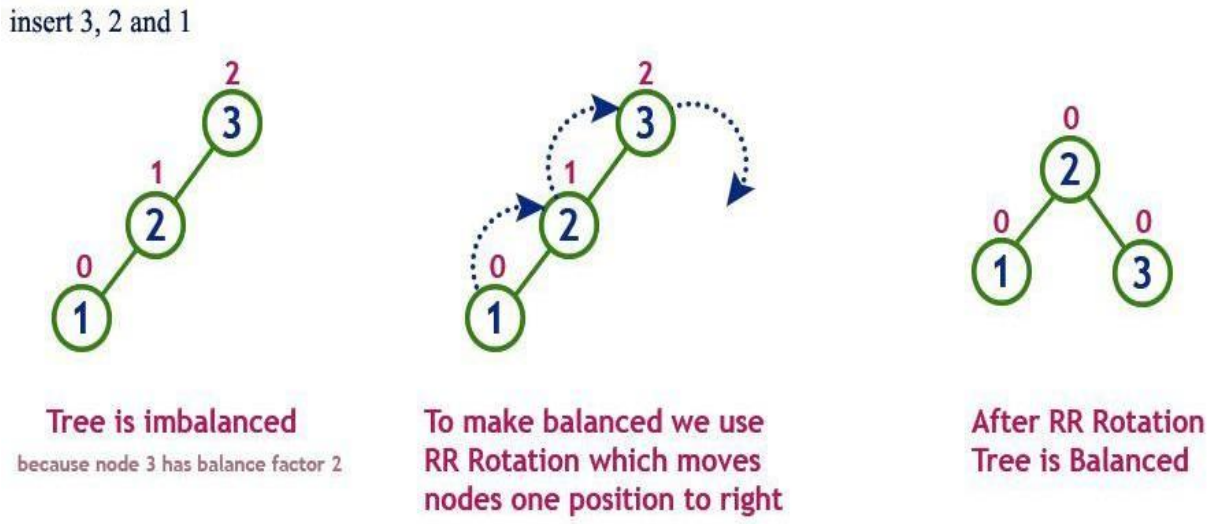
Rotation is the process of moving the nodes to either left or right to make tree balanced.



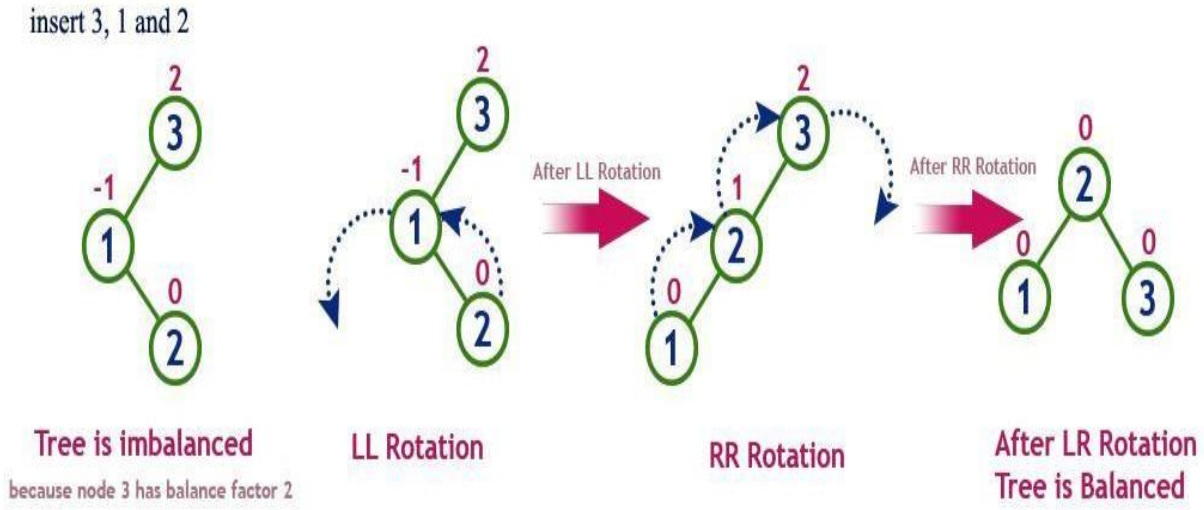
a. **Single Left Rotation (LL Rotation)**: In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree.



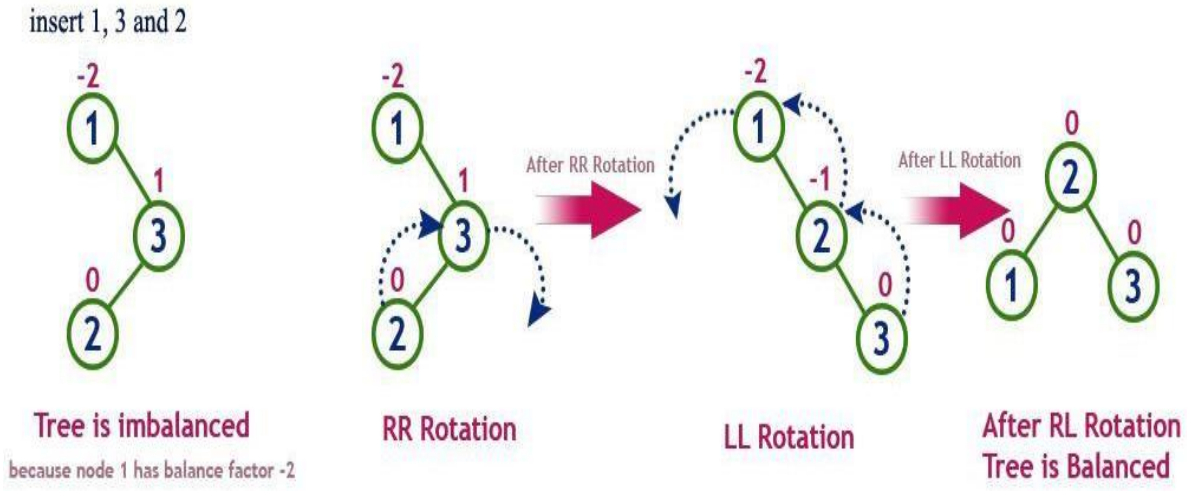
b. Single Right Rotation (RR Rotation): In RR Rotation every node moves one position to right from the current position. To understand RR Rotation, let us consider following insertion operations into an AVL Tree.



c. Left Right Rotation (LR Rotation): The LR Rotation is combination of single left rotation followed by single right rotation. In LR Rotation, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree.



d. Right Left Rotation (RL Rotation): The RL Rotation is combination of single right rotation followed by single left rotation. In RL Rotation, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree.



Operations on an AVL Tree

The following operations are performed on an AVL tree...

a. Search
b. Insertion
c. Deletion

a. Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node 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.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

b. Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

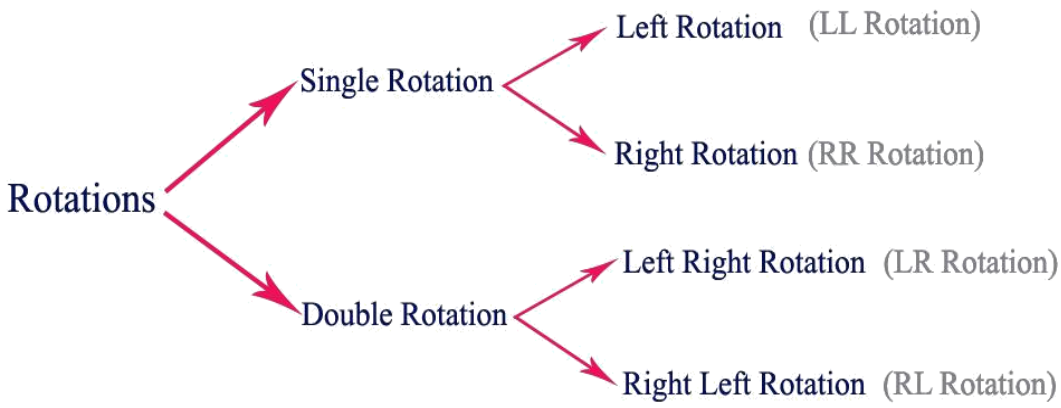
Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the Balance Factor of every node.

Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. **Then perform the suitable Rotation** to make it balanced. And go for next operation.

We already know the suitable rotations:


Rotations
Single Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Double Rotation
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

Huffman coding:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable- legth codes to input characters; lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de- compressed output may be “cccd” or “ccb” or “acd” or “ab”.

There are mainly two major parts in Huffman Coding

1) Build a Huffman Tree from input characters.

2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree: Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

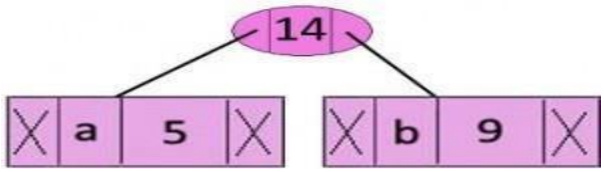
Let us understand the algorithm with an example:

character	**Frequency**	
A	05	

b	09	
c	12	
d	13	
e	16	
f	45	

Step 1: Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

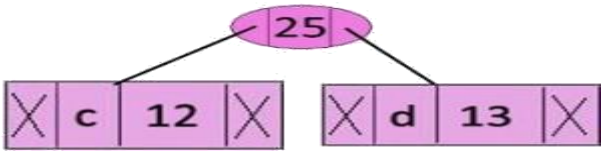
Step 2: Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements.

character		**Frequency**
c		12
d		13
Internal Node		14
e		16
f		45

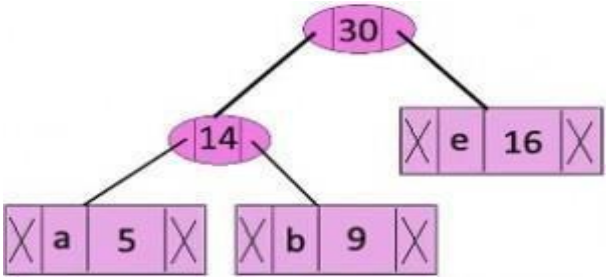
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

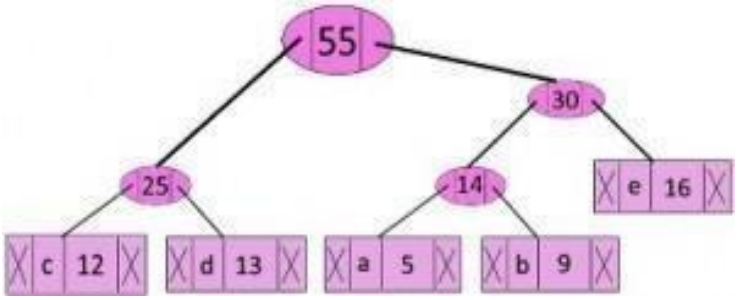
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

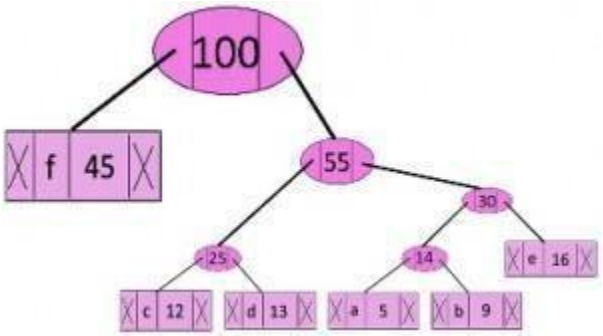
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes.

character	Frequency		
f	45		
Internal Node	55		

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



Now min heap contains only one node.

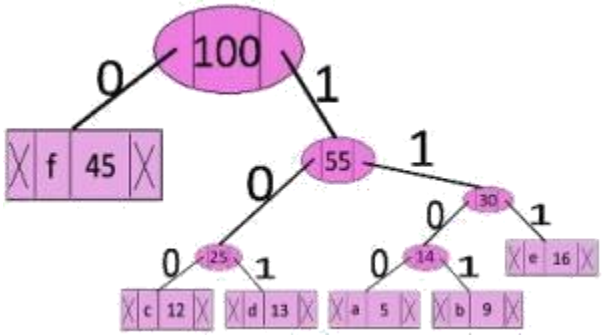
character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.

The codes are as follows



character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Red Black:

Tree is a Binary Search Tree in which every node is colored eigther RED or BLACK. In a Red Black Tree the color of a node is decided based on the Red Black Tree properties. Every Red Black Tree has the following properties.

Properties of Red Black Tree:

Property #1: Red - Black Tree must be a Binary Search Tree.

Property #2: The ROOT node must color BLACK.

Property #3: The children of Red colored node must colored BLACK. (There should not be two consecutive RED nodes).

Property #4: In all the paths of the tree there must be same number of BLACK colored nodes.
Property #5: Every new node must insert with RED color.

Property #6: Every leaf (i.e. NULL node) must color BLACK.

Insertion into RED BLACK Tree: In a Red Black Tree, every new node must be inserted with color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red Black Tree. If all the properties are satisfied then we go to next operation otherwise we need to perform following operation to make it Red Black Tree.

1. Recolor

3. Rotation followed by Recolor

The insertion operation in Red Black tree is performed using following steps...

Step 1: Check whether tree is Empty.

Step 2: If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.

Step 3: If tree is not Empty then insert the newNode as a leaf node with Red color.

Step 4: If the parent of newNode is Black then exit from the operation.

Step 5: If the parent of newNode is Red then check the color of parent node's sibling of new Node.

Step 6: If it is Black or NULL node then make a suitable Rotation and Recolor it.

Step 7: If it is Red colored node then perform Recolor and Recheck it. Repeat the same until tree becomes Red Black Tree

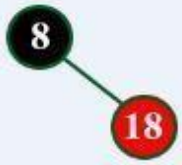
Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

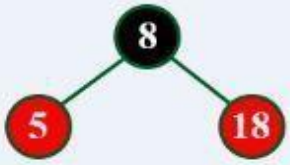
insert (18)

Tree is not Empty. So insert newNode with red color.



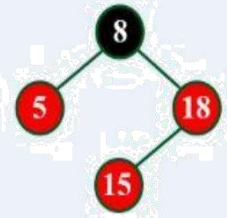
insert (5)

Tree is not Empty. So insert newNode with red color.



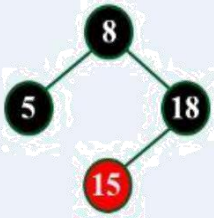
insert (15)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

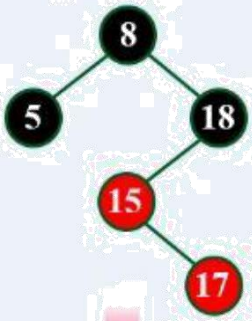
After RECOLOR



After Recolor operation, the tree is satisfying all Red Black Tree properties.

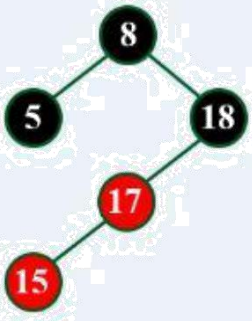
insert (17)

Tree is not Empty. So insert newNode with red color.

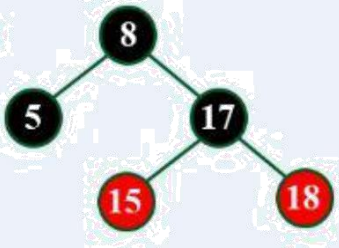


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

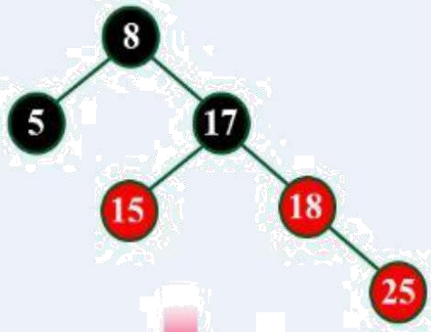


After Right Rotation & Recolor



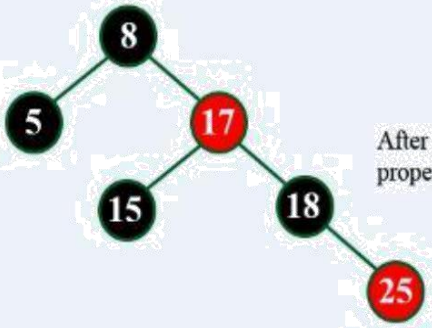
insert (25)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



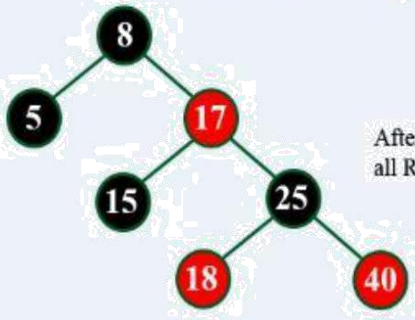
After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (40)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

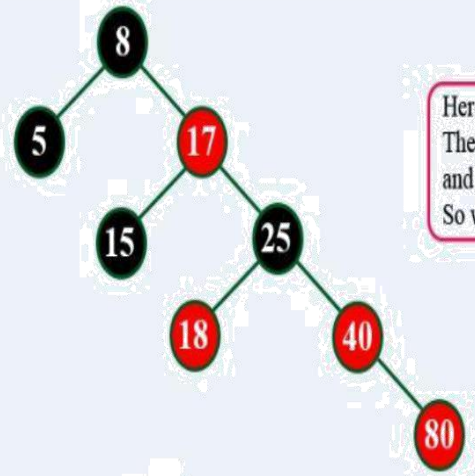
After LL Rotation
& Recolor



After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (80)

Tree is not Empty. So insert newNode with red color.

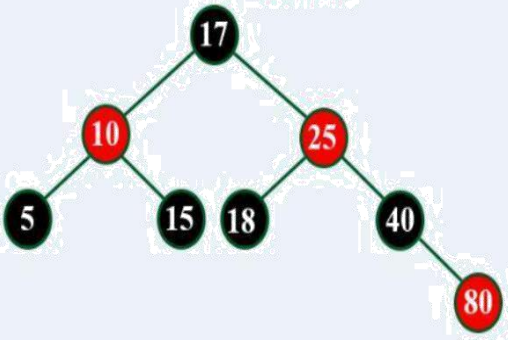


Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor

After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.

After Left Rotation
& Recolor



Deletion Operation in Red Black Tree:

In a Red Black Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Red Black Tree properties. If any of the property is violated then make suitable operation like Recolor or Rotation & Recolor.

B-Tree

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children. B-Tree was developed in the year of 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.

B-Tree can be defined as follows...

B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. Every B-Tree has order.

B-Tree of Order m has the following properties...

Property #1 - All the leaf nodes must be at same level.

Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.

Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.

Property #4 - If the root node is a non leaf node, then it must have at least 2 children.

Property #5 - A non leaf node with n-1 keys must have n number of children.

Property #6 - All the key values within a node must be in Ascending Order.

Assignment: Multi way search tree.

7. Sorting

Q. What is sorting?
Sorting is an algorithm that arranges the elements of a list in a certain order [either ascending or descending].

Q. Why is Sorting Necessary?
Sorting is one of the important categories of algorithms in computer science and a lot of research has gone into this category. Sorting can significantly reduce the complexity of a problem, and is often used for database algorithms and searches.

Method of classifying sorting algorithms is:
- Internal Sort
- External Sort

Internal Sort
Sort algorithms that use main memory exclusively during the sort are called internal sorting algorithms. This kind of algorithm assumes high-speed random access to all memory.

External Sort
Sorting algorithms that use external memory, such as tape or disk, during the sort come under this category.

1. Insertion Sorting

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

- It has one of the simplest implementation
- It is efficient for smaller data sets, but very inefficient for larger lists.
- Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
- It is better than Selection Sort and Bubble Sort algorithms.
- Its space complexity is less, like Bubble sorting, Insertion sort also requires a single additional memory space.
- It is Stable, as it does not change the relative order of elements with equal keys.

Idea of Algorithm: Insertion Sort

It works the way you might sort a hand of playing cards:

a) We start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].

b) Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].

c) To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.

Note that at all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Pseudocode:

We use a procedure INSERTION_SORT. It takes as parameters an array A[1.. n] and the length n of the array. The array A is sorted in place: the numbers are rearranged within the array, with at most a constant number outside the array at any time.

INSERTION_SORT (A)

```
INSERTION_SORT (A)

1.   FOR j ← 2 TO length[A]
2.        DO key ← A[j]
3.           {Put A[j] into the sorted sequence A[1 . . j – 1]}
4.           i ← j – 1
5.           WHILE i > 0 and A[i] > key
6.                 DO A[i +1] ← A[i]
7.                      i ← i – 1
8.           A[i + 1] ← key
```

Example:
12, 11, 13, 5, 6
Let us loop for i = 1 (second element of the array) to 5 (Size of input array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12, THEN 11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13 THEN 11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

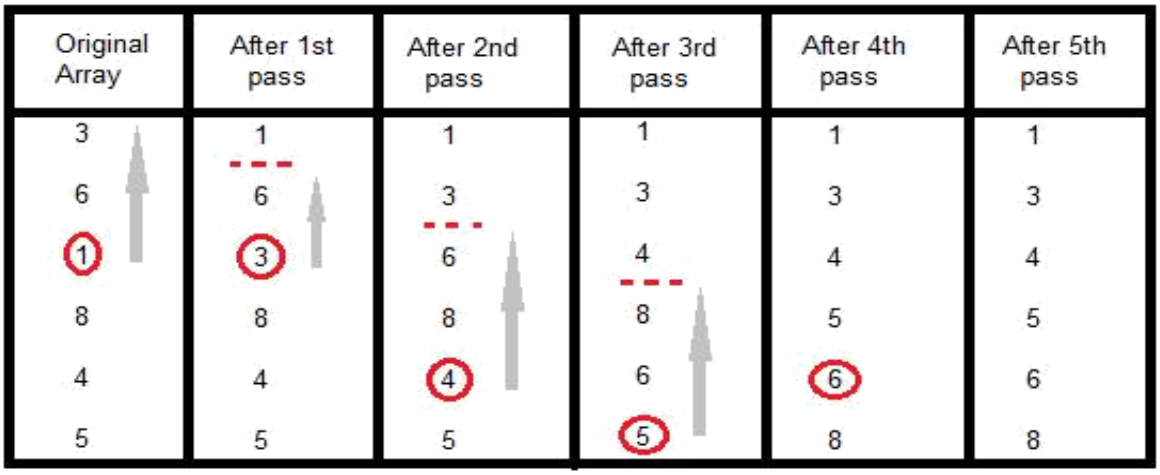
1. Selection Sorting

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

Pseudocode:

```
SELECTION_SORT (A)

for i ← 1 to n-1 do
   min j ← i;
   min x ← A[i]
   for j ← i + 1 to n do
      If A[j] < min x then
         min j ← j
         min x ← A[j]
   A[min j] ← A [i]
   A[i] ← min x
```

How Selection Sorting Works
Selection Sorting in Data Structures



In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.

2. Bubble Sorting

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares the entire element one by one and sort them based on their values. It is called Bubble sort, because with each iteration the largest element in the list bubbles up towards the last place, just like a water bubble rises up to the water surface. Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Bubble Sort Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:
First Pass:
(5 1 4 2 8) –> (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 4 5 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 2 5 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(1 4 2 5 8) –> (1 4 2 5 8)
(1 **4 2** 5 8) –> (1 2 4 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 4 5 8)
(1 2 **4** 5 **8**) –> (1 2 4 5 8)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

Classwork: 5 4 3 2 1 Sort in ascending order using bubble sort.

5 4 3 2 1 → check 5 and 4 → 5> 4 so

4 5 3 2 1 now check 5 and 3
4 3 5 2 1 now check 5 and 2
4 3 2 5 1 now check 5 and 1
4 3 2 1 5

Similarly,
4 3 2 1 5 → 3 4 2 1 5
3 2 4 1 5
3 2 1 4 5
3 2 1 4 5

3 2 1 4 5 → 2 3 1 4 5
2 1 3 4 5
2 1 3 4 5
2 1 3 4 5

2 1 3 4 5 → 1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5

1 2 3 4 5 → 1 2 3 4 5 and so on

Pseudocode:
SEQUENTIAL BUBBLESORT (A)

```
for i ← 1 to length [A] do
   for j ← length [A] downto i +1 do
      If A[A] < A[j-1] then
         Exchange A[j] ↔ A[j-1]
```

Complexity Analysis of Bubble Sorting
In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be (n-1)+(n-2)+(n-3)+.....+3+2+1

Sum = n(n-1)/2 i.e O(n2)

Hence the complexity of Bubble Sort is O(n2).

The main advantage of Bubble Sort is the simplicity of the algorithm. Space complexity for Bubble Sort is O(1), because only single additional memory space is required for temp variable. Best-case Time Complexity will be O(n), it is when the list is already sorted.

3. Merge Sort Algorithm

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

middle m = (l+r)/2

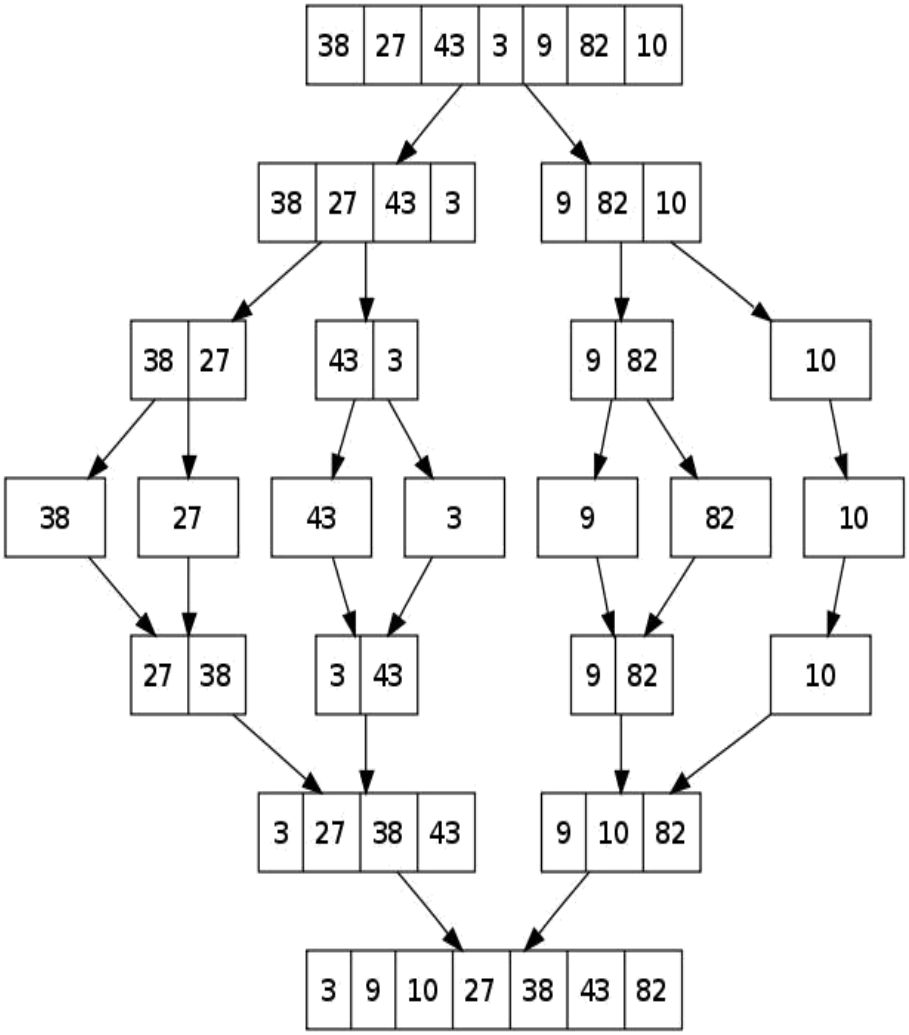
2. Call mergeSort for first half:

3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

Complexity Analysis of Merge Sort: Worst Case Time Complexity : O(n log n) Best Case Time Complexity : O(n log n) Average Time Complexity : O(n log n) Space Complexity : O(n)
Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

4. Radix sort:

is a small method that many people intuitively use when alphabetizing a large list of names. (Here Radix is 26, 26 letters of alphabet). Specifically, the list of names is first sorted according to the first letter of each names, that is, the names are arranged in 26 classes. Intuitively, one might want to sort numbers on

their most significant digit. But Radix sort do counter-intuitively by sorting on the least significant digits first. On the first pass entire numbers sort on the least significant digit and combine Then on the second pass, the entire numbers are sorted again on the second least-significant digits and
combine in a array and so on.

Following example shows how Radix sort operates on seven 3-digits number.

INPUT	**1st**	**2nd**	**3rd**
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the above example, the first column is the input. The remaining shows the list after successive sorts on increasingly significant digits position. The code for Radix sort assumes that each element in the n element array A has d digits, where digit 1 is the lowest-order digit and d is the highest-order digit.

Pseudocode:

```
RADIX_SORT (A, d)
 for i ← 1 to d do

use a stable sort to sort A on digit i
// counting sort will do the job
```

Complexity Analysis:

There are d passes, so the total time for Radix sort is (n+k) time. There are d passes, so the total time for Radix sort is (dn+kd). When d is constant and k = (n), the Radix sort runs in linear time.

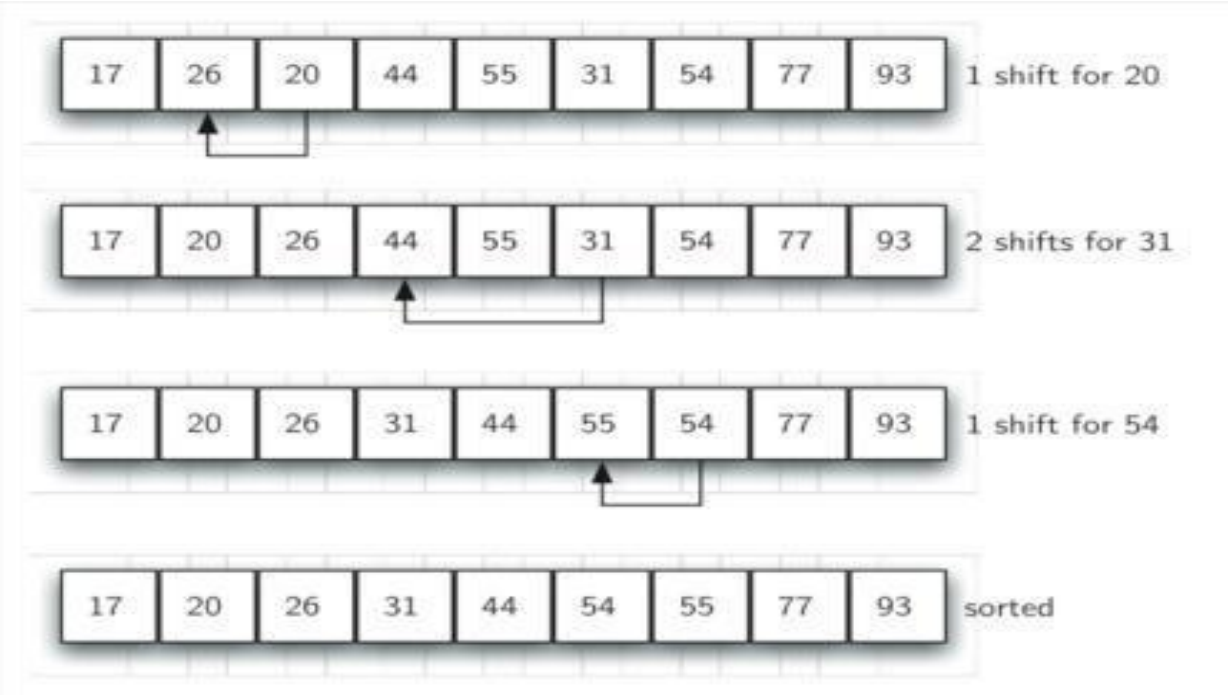
5. The Shell Sort

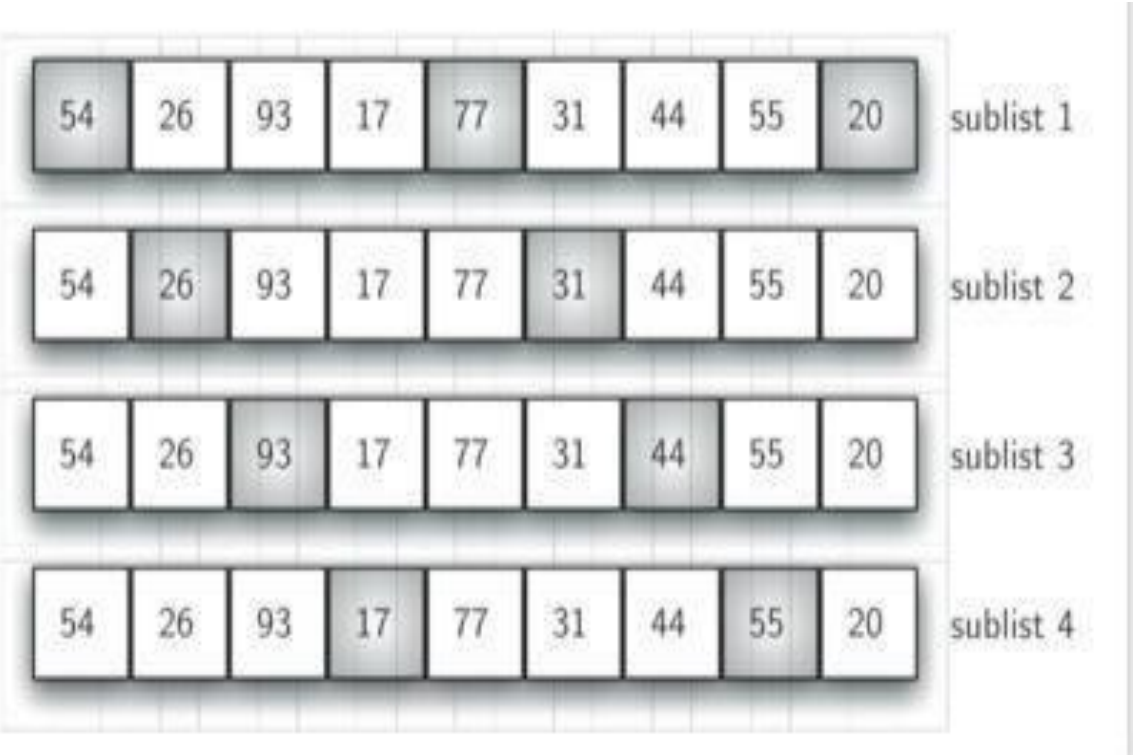
The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sublist by choosing all items that are i items apart.

This can be seen in Figure a. This list has nine items. If we use an increment of three, there are three sublists, each of which can be sorted by an insertion sort. After completing these sorts, we get the list shown in Figure b. Although this list is not completely sorted, something very interesting has

happened. By sorting the sublists, we have moved the items closer to where they actually belong.

Figure c shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.


17 26 20 44 55 31 54 77 93
1 shift for 20
17 20 26 44 55 31 54 77 93
2 shifts for 31
17 20 26 31 44 55 54 77 93
1 shift for 54
17 20 26 31 44 54 55 77 93
sorted


54 26 93 17 77 31 44 55 20
sublist 1
54 26 93 17 77 31 44 55 20
sublist 2
54 26 93 17 77 31 44 55 20
sublist 3
54 26 93 17 77 31 44 55 20
sublist 4

We said earlier that the way in which the increments are chosen is the unique feature of the shell sort. The function shown in ActiveCode 1 uses a different set of increments. In this case, we begin with n/2 sublists. On the next pass, n/4 sublists are sorted. Eventually, a single list is sorted with the basic insertion sort. Figure d shows the first sublists for our example using this increment.

This algorithm is a simple extension of Insertion sort. Its speed comes from the fact that it exchanges elements that are far apart (the insertion sort exchanges only adjacent elements).

The idea of the Shell sort is to rearrange the file to give it the property that taking every hth element (starting anywhere) yields a sorted file. Such a file is said to be h-sorted.

Pseudocode:

```
SHELL_SORT (A)
for h = 1 to h <=N/9 do
for (; h > 0; h != 3) do
for i = h +1 to i <= n do
v = A[i]
j = i
while (j > h AND A[j - h] > v
A[i] = A[j - h]
j = j - h
A[j] = v
i = i + 1
```

6. Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap.
3. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
4. Repeat above steps while size of heap is greater than 1.

Q. How to build the heap?

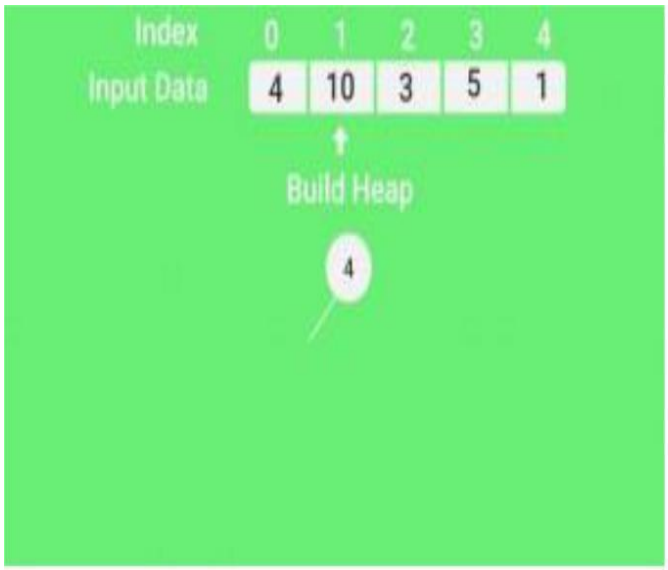
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

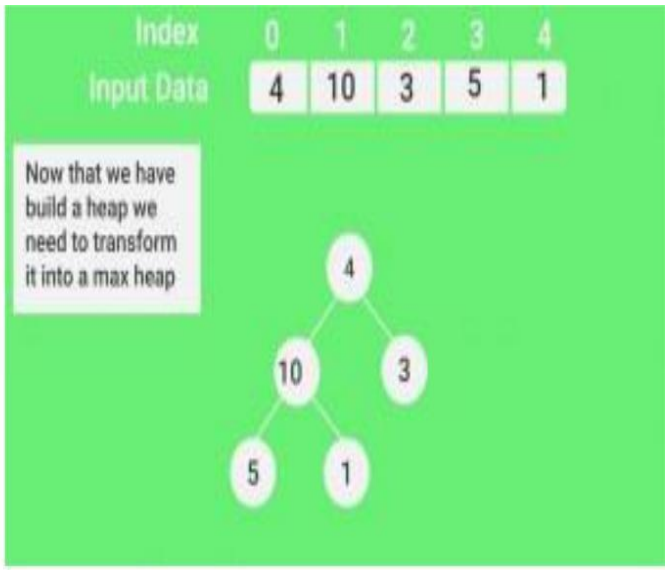
Lets understand with the help of an example:

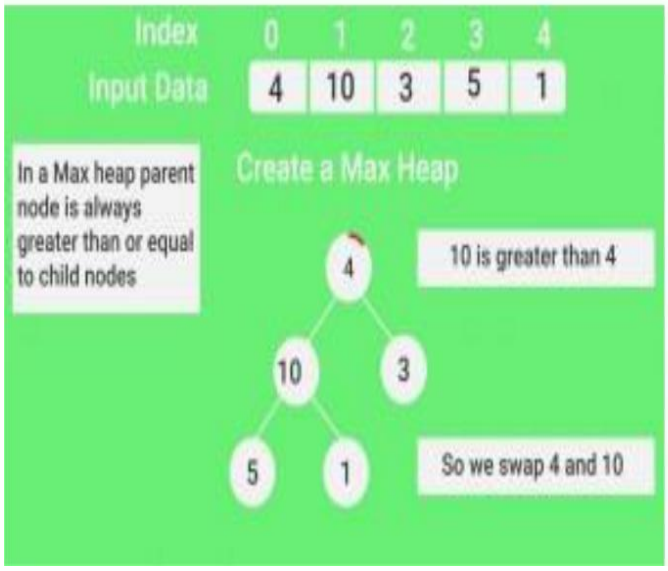
Input data: 4, 10, 3, 5, 1

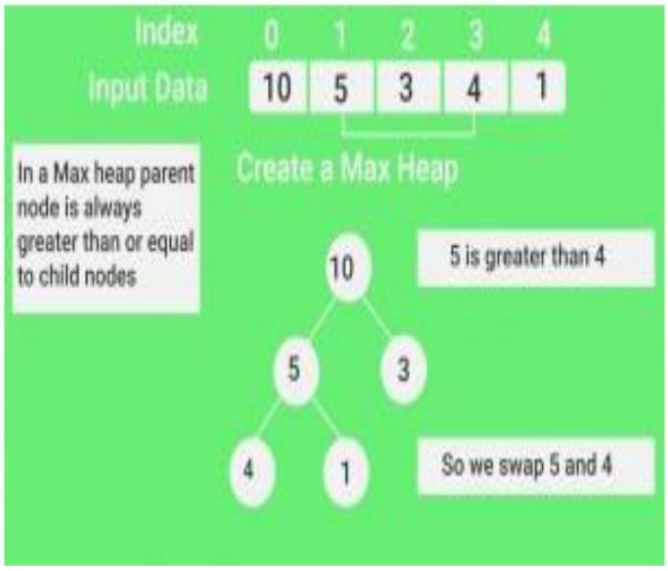
```
                4(0)
               / \
            10(1) 3(2)
             / \
          5(3)1(4)
```

The numbers in bracket represent the indices in the array representation of data.

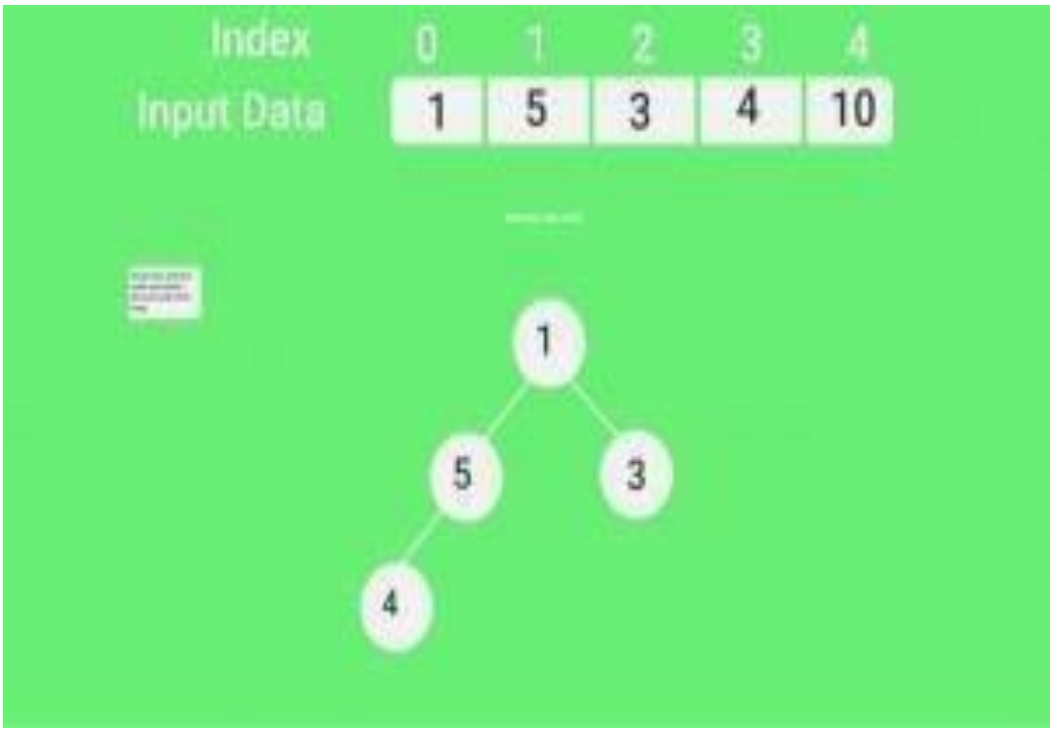




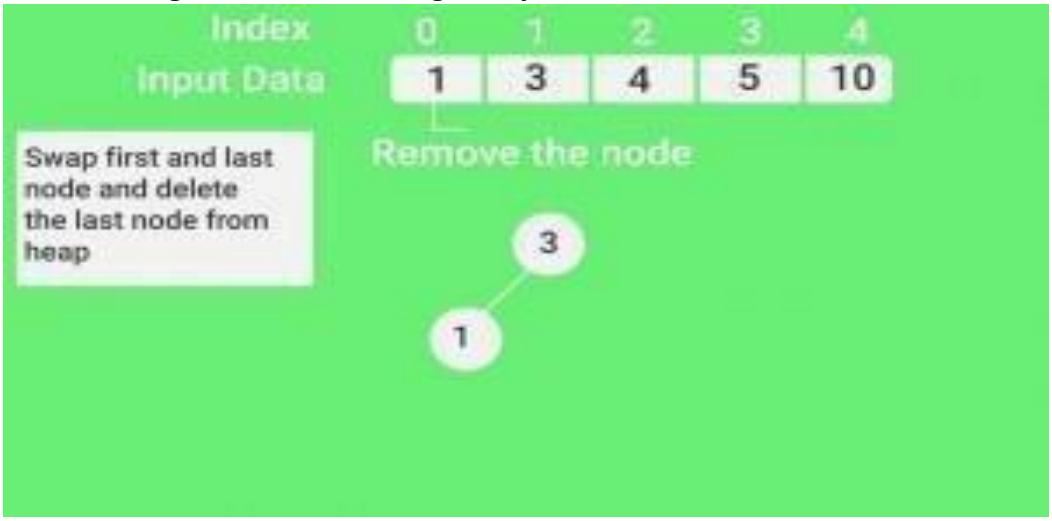




Swap first (10) and last node (1) and delete last node (10), Now from this tree again create max heap and delete last node.



Continuing this result below fig finally,



Algorithm HeapSort(A)
1: Build-Max-Heap(A)
2: heapsize ← heap-size[A]
3: while heapsize > 1 do
4: A*heapsize+ ← Heap-Extract-Max (A)
5: heapsize ← heapsize - 1
6: end while

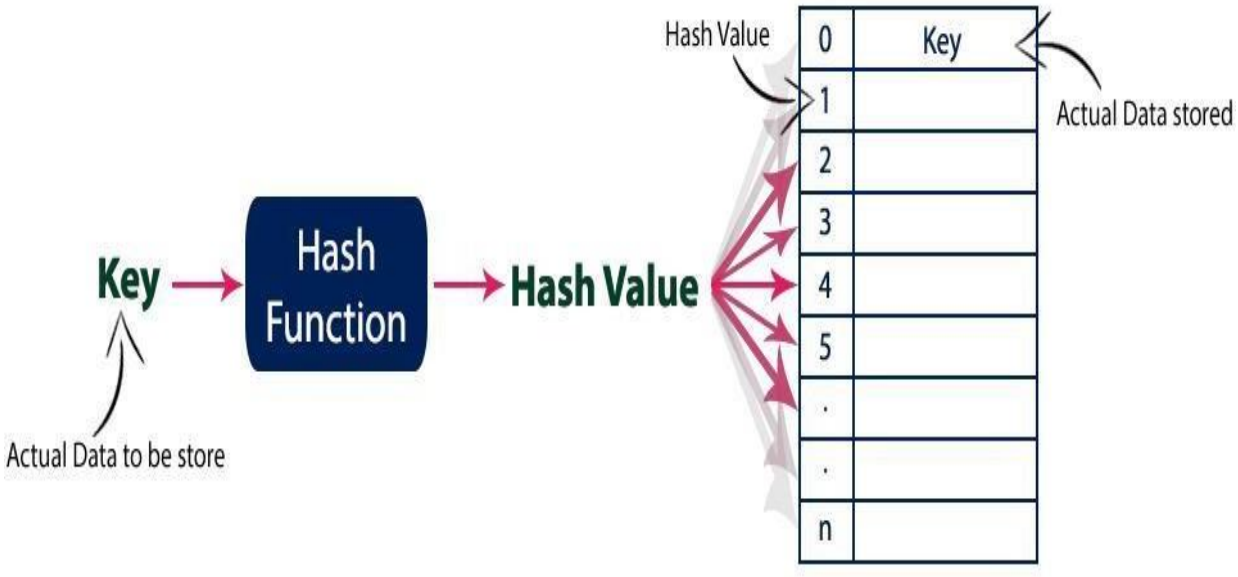
Assignment : Write an algorithm of quick sort with an example.

Hashing

In all search techniques like linear search, binary search and search trees, the time required to search an element is depends on the total number of element in that data structure. In all these search techniques, as the number of element are increased the time required to search an element also increased linearly.

Hashing is another approach in which time required to search an element doesn't depend on the number of element. Using hashing data structure, an element is searched with constant time complexity. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

Basic concept of hashing and hash table is shown in the following figure.



Hashing

Hashing is the process of indexing and retrieving element (data) in a data structure to provide faster way of finding the element using the hash key. Here, hash key is a value which provides the index value where the actual data is likely to store in the data structure.

In this data structure, we use a concept called Hash table to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a hash function. That means every entry in the hash table is based on the key value generated using a hash function.

A Hash function

Hash function is a function which takes a piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table.

Hash Table

Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1)).

Hash tables are used to perform the operations like insertion, deletion and search very quickly in a data structure. Using hash table concept insertion, deletion and search operations are accomplished in constant time. Generally, every hash table makes use of a function, which we'll call the hash function to map the data into the hash table.

1. **Direct Assign**
   If you have given elements represented by keys- 8,3..10 then define array of particular size having size greater than or equal to the maximum value of the key. Then assign each element of keys in an array having index equal to the value of key's element.
   i.e.
   value 8 is stored in an array of index 8.
   Value 3 is stored in an array of index 3 and so on.

   If you have to search for key value 10 then go an index 10 of an array and so on. It can be represented as: in fig 1.

   Drawbacks:

   a. If you have next element 50 after 10 then you have to define size of an array 50 to store just 1 element which is waste of lot of space in an array. So to overcome this technique we have to select hash function efficiently.

2. Using hash function as,

a. h(x)=x% size of an array, where x is an input element and h(x) gives an index.

See fig 2.

Collision Resolution Techniques:

The process of finding an alternate location is called collision resolution. Even though hash tables have collision problems, they are more efficient in many cases compared to all other data structures, like search trees. There are a number of collision resolution techniques, and the most popular are direct chaining and open addressing.

•**Direct Chaining**: An array of linked list application

- Separate chaining

•**Open Addressing**: Array-based implementation

- Linear probing (linear search)
- Quadratic probing (nonlinear search)
- Double hashing (use two hash functions)

1. Direct Chaining

a. Separate Chaining:

Collision resolution by chaining combines linked representation with hash table. When two or more records hash to the same location, these records are constituted into a singly-linked list called a chain. **(see fig 3)**

2. Open Addressing

a. Linear Probing:

The interval between probes is fixed at 1. In linear probing, we search the hash table sequentially, starting from the original hash location. If a location is occupied, we check the next location. We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following: rehash(key) = (n + 1)% table size.

For 4, index is 4 but is already full so it moved next location until it finds an empty slot in an array, in below fig it finds empty slot at an index 5. If key 5 is present in case than it finds index full is again and go for index 6 and so on until it finds empty space.

If key 4 is search than index 4 is looked first if finds ok, else move to next index and so on. For 24 it first search index 4 as 24%10=4. But not present there, move to next index 5 also not present there and continues until it finds empty slot.

(see fig 4)

Drawback:

One of the problems with linear probing is that table items tend to cluster together in the hash table. This means that the table contains groups of consecutively occupied locations that are called clustering. Clusters can get close to one another, and merge into a larger cluster. Thus, the one part of the table might be quite dense, even though another part has relatively few items. Clustering causes long probe searches and therefore decreases the overall efficiency. The next location to be probed is determined by the step-size, where other step-sizes (more than one) are possible. The step-size should be relatively prime to the table size, i.e. their greatest common divisor should be equal to 1. If we choose the table size to be a prime number, then any step-size is relatively prime to the table size. Clustering cannot be avoided by larger step-sizes.

b. Quadratic Probing:

The interval between probes increases proportionally to the hash value (the interval thus increasing linearly and the indices are described by a quadratic function). The problem of Clustering can be eliminated if we use the quadratic probing method. In Quadratic probing, we start from the original hash location i. If a location is occupied, we check the locations i + 12 , i +22, i + 32, i + 42... We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following: rehash (key) = (n + k^2)% table size.

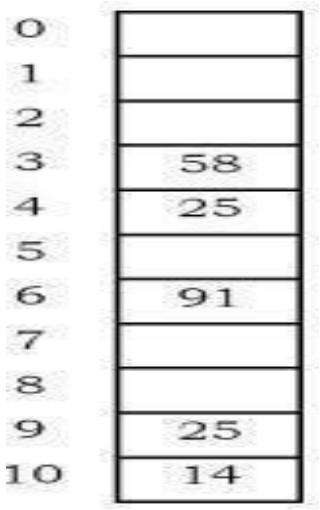
Even though clustering is avoided by quadratic probing, still there are chances of clustering. Clustering is caused by multiple search keys mapped to the same hash key. Thus, the probing sequence for such search keys is prolonged by repeated conflicts along the probing sequence. Both linear and quadratic probing use a probing sequence that is independent of the search key**. (See fig 5))**

c. Double Hashing

The interval between probes is computed by another hash function. Double hashing reduces clustering in a better way. The increments for the probing sequence are computed by using a second hash function. The second hash function h2 should be: h2(key) ≠ 0 and h2 ≠ h1.We first probe the location h1(key). If the location is occupied, we probe the location h1(key) + h2(key), h1(key) + 2 * h2(key), ...

Example:
Table size is 11 (0..10)
Hash Function: assume h1(key) = key mod 11 and h2(key) = 7- (key mod 7)
Insert keys: 58 mod 11 = 3
14 mod 11 = 3 → 3 + 7 = 10
91 mod 11 = 3 → 3+ 7,3+ 2* 7 mod 11 = 6

25 mod 11 = 3 → 3 + 3,3 + 2*3 = 9



Q. what is the running time analysis?

It is the process of determining how processing time increases as the size of problem (input size increases). In general we encounter following types of inputs: Size of an array, Number of elements in matrix, Vertices and edge in graph, Polynomial degree etc.

Q. How to compare algorithms?

To compare algorithms, let us define few objective measures:

Execution times? Not a good measures as execution times are specific to a particular computer.

Number of statements executed? Not a good measure, since the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal solution? We expressed running time of given algorithm as function of the input size n (f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming styles, etc.

Rate of growth:

The rate at which the running time increases as a function of input is called rate of growth. Let us assume you went to a shop for buy car and a cycle. If your friend sees you there and asks what are you buying then in general we say buying a car. This is because, cost of car is too big compared to cost of cycle.

Total Cost = cost_of_car + cost _of_cycle.
Total cost nearly equal to cost_of_car(approx..)

For the above example, we can represent the cost of car and cost of cycle in terms of function and for given function ignore the low order terms that are relatively insignificant. For Ex: if n^4 , 2n^2,110n, 500 are individual costs of some function and approximate it to n^4 is the highest rate of growth.

Running time in term of function:

```
1. for(i=1;i<=n;i++)
        m=m+1;

 //executes n times
// for 1 execution say, take constant time c
```

for 1 execution take c time, for n execution take cn time, so, total running time :f(n)=cn

```
2 .for(i=1;i<n;i++){         //outer loop executed n times

for(i=1;i<n;i++){   //inner loop executed n times

k=k+1;         // for 1 execution say, take constant time c
}
}
```

for 1 execution take cn time, for n execution take cn*n time,
so, total running time : f(n)= cn^2

Types of analysis:
To analyze given algorithm we need to know on what inputs the algorithm is taking less time (performing well) and on what inputs the algorithm is taking huge time. Best case:

Algorithm takes less time.
Input is one for which algorithm runs the faster.
Worst case:
Algorithm takes huge time.
Input is one for which algorithm runs the slower.
Average case:
Lower Bound<= Average Time <= Upper Bound

An algorithm may run faster on some inputs than it does on others of the same size. Thus we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size. However, an average case analysis is typically challenging.

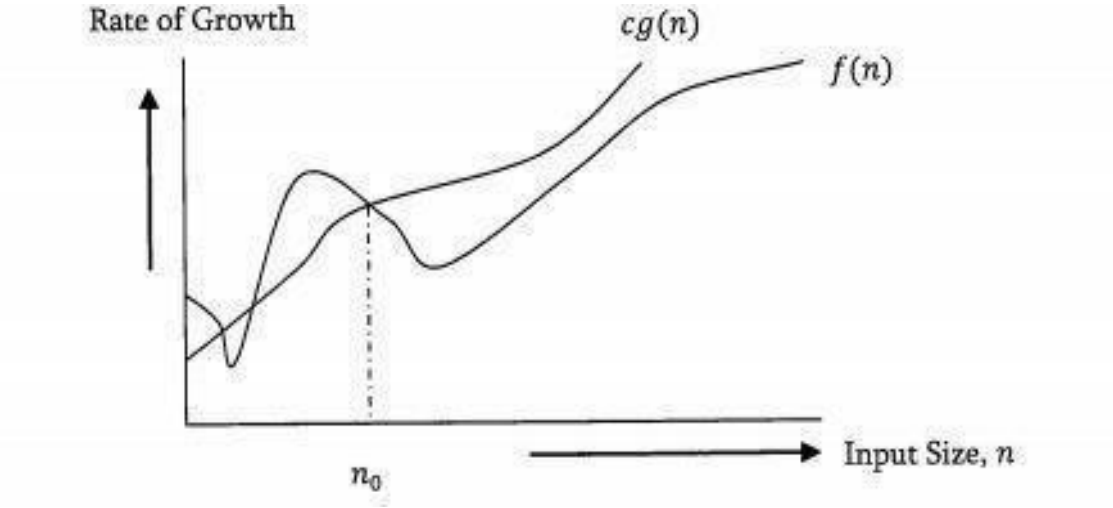
In real-time computing, the worst case analysis is often of particular concern since it is important to know how much time might be needed in the worst case to guarantee that the algorithm would always finish on time. The term best case performance is used to describe the way an algorithm behaves under optimal conditions.

A worst case analysis is much easier than an average case analysis, as it requires only the ability to identify the worst case input. This approach typically leads to better algorithms. Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it will do well on every input.

Asymptotic Notations:

1. Big-O notation:

Let f(n) and g(n) be functions mapping nonnegative integers to real numbers. We say that f(n) is O(g(n)) if there is a real constant c > 0 and an integer constant n0 >= 1 such that f(n) <= cg(n), for n >= n0.



Explanation:

The big-Oh notation gives an upper bound on the growth rate of a function. The statement "f(n) is O(g(n))" means that the growth rate of f(n) is no more than the growth rate of g(n).

Ex: f(n)=5n^4 + 3n^3 + 2n^2 + 4^n + 1 is O(n^4).

Solution: 5n^4 + 3n^3 + 2n^2 + 4n + 1 <= (5 + 3 + 2 + 4 + 1)n^4

<= 15 n^4

= cg(n), for c = 15 and n0 >= 1.

Ex: f(n)=3n+8 is O(n) Solution: 3n+8 <=(3n+n) <=4n

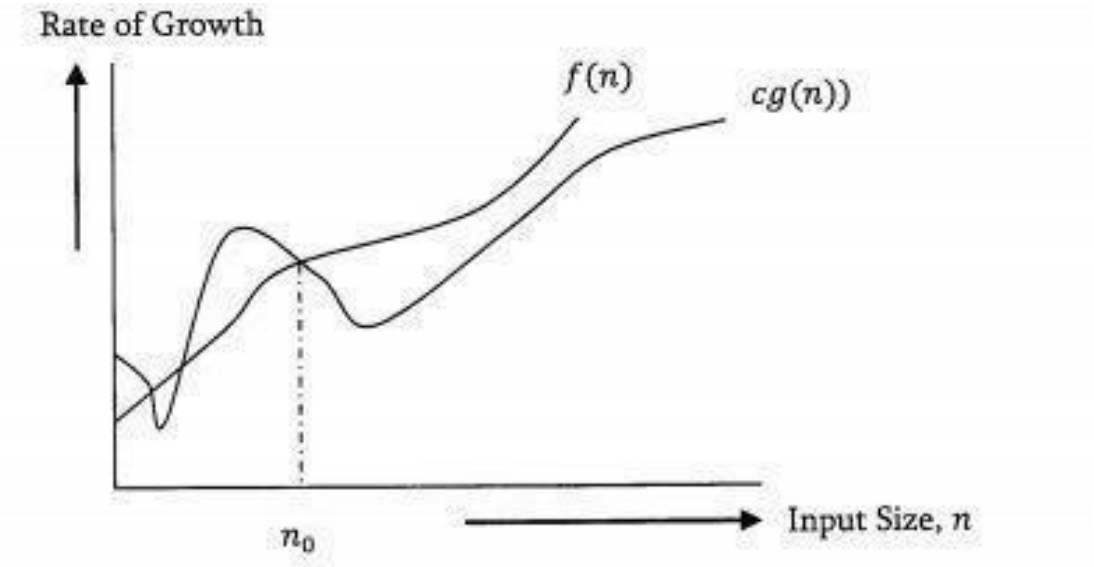
f(n)<=cg(n), for c =4 and n0>=8

2. Big Omega-Ω notation:

Let f(n) and g(n) be functions mapping nonnegative integers to real numbers. We say that f(n) is Ω(g(n)) (pronounced "f(n) is big-Omega of g(n)") if there is a real constant c > 0 and an integer constant n0 >= 1 such that f(n) >= cg(n), for n >= n0.

3. Big-Theta-Θ notation:

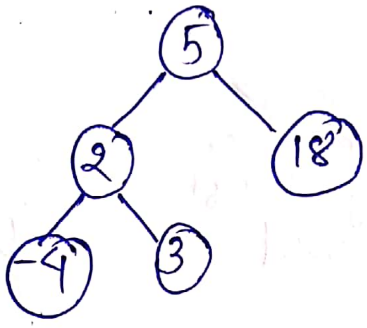
f(n) is Θ(g(n)) (pronounced "f(n) is big-Theta of g(n)") if f(n) is O(g(n)) and f(n) is Ω(g(n)), that is, there are real constants c1 > 0 and c2 > 0, and an integer constant n0 >= 1 such that c1g(n) <= f(n)
<= c2g(n), for n >= n0.



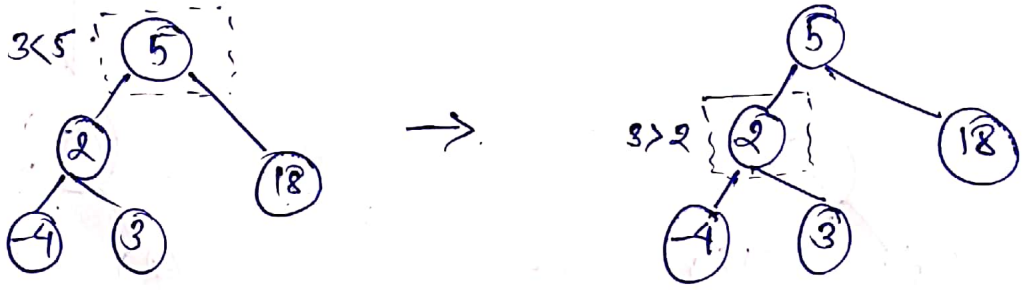
Search Algorithm of BST

Example:

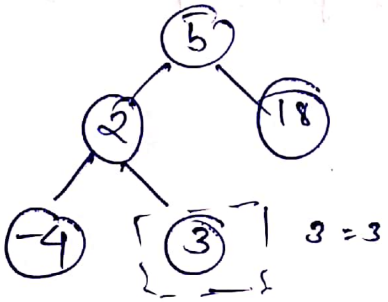
① Search for 3 in the tree, given below.



② 3 is less than 5 go to the left child.



③ 3 is more than 2 go to the right child.



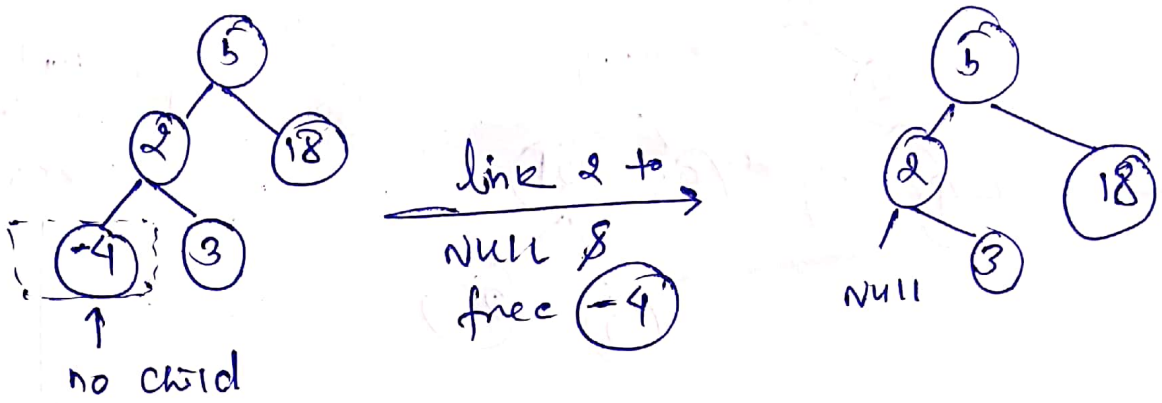
④ 3=3, search value is found.

Scanned by CamScanner

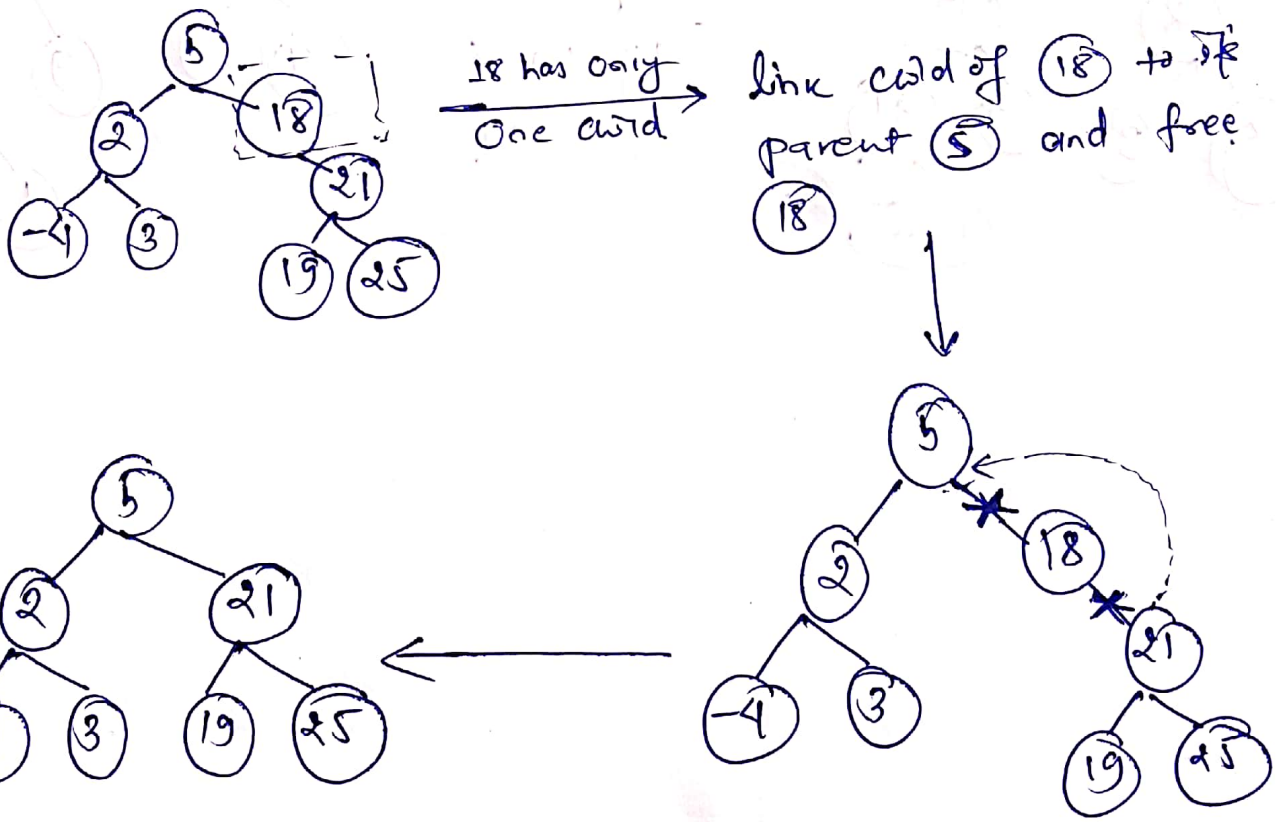
Remove Algorithm in BST

Example:

Case 1: Remove -4 from a BST.



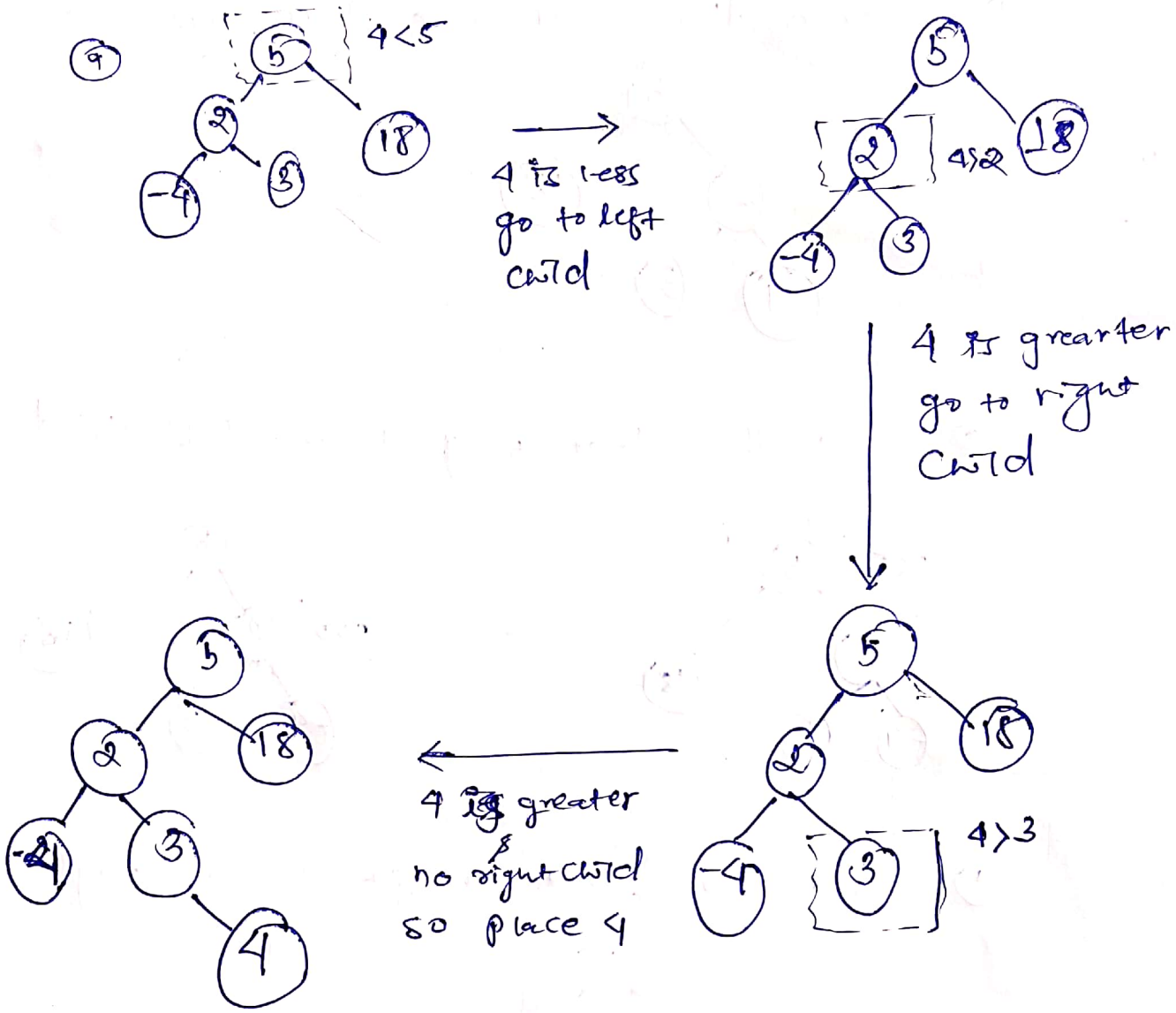
Case 2 Remove 18 from a BST.



Scanned by CamScanner

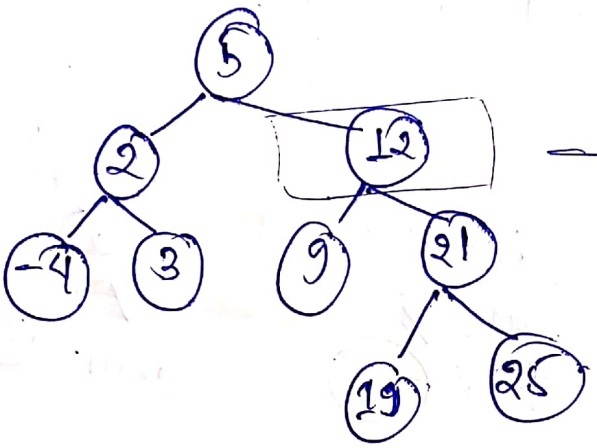
Insertion Algorithm of B.S.T

Example: Insert 4 to the tree.



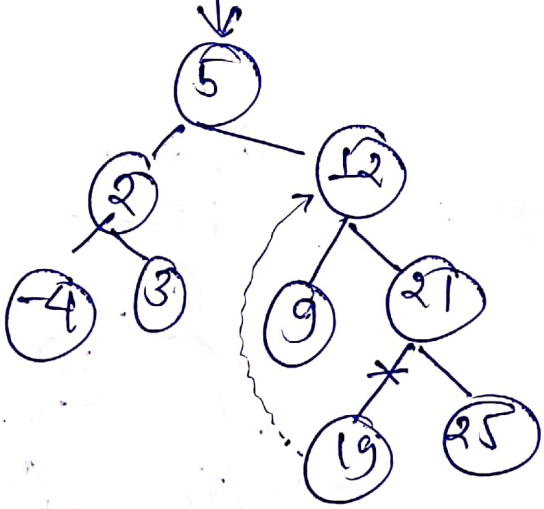
Scanned by CamScanner

Case 3 Remove 12 from a BST

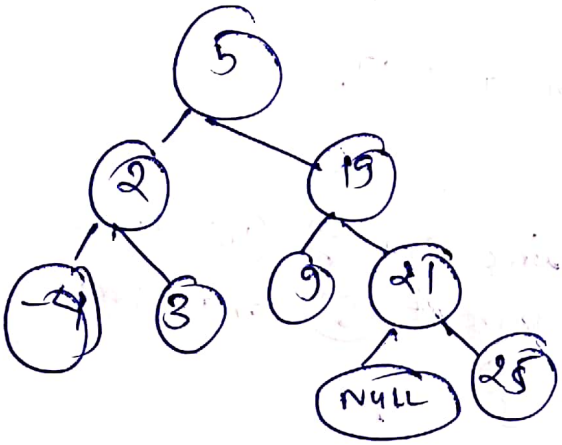


12 has two child

Find minimum node in right child ie. 19



Replace 12 by 19 & free 19.



Scanned by CamScanner

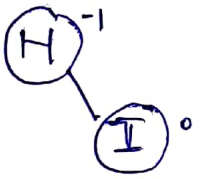
(1) Create an AVL search tree from the given set of values.

H, I, J, B, A, E, C, F, D, G, K, L

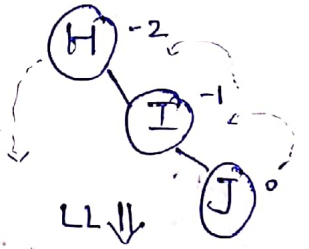
Sol^{n}: ↓ [A<B<C ... <Z]

Insert H

Insert I

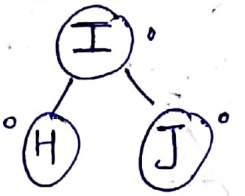


Insert J

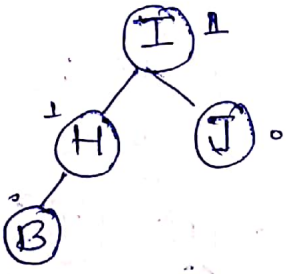


Rebalancing needed

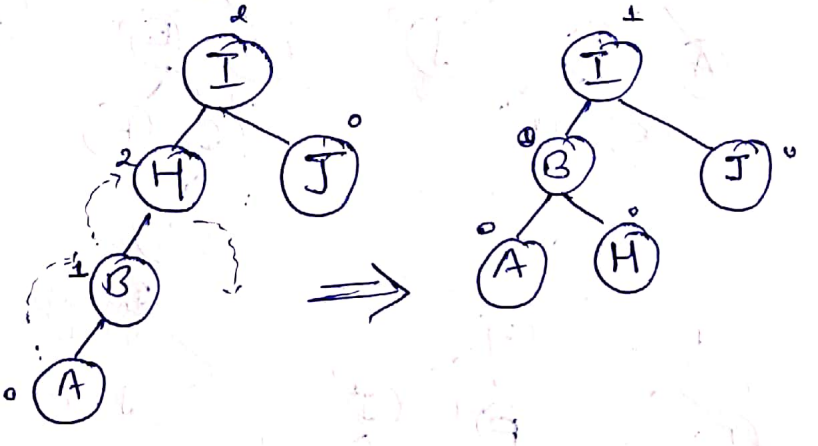
LL ⇓



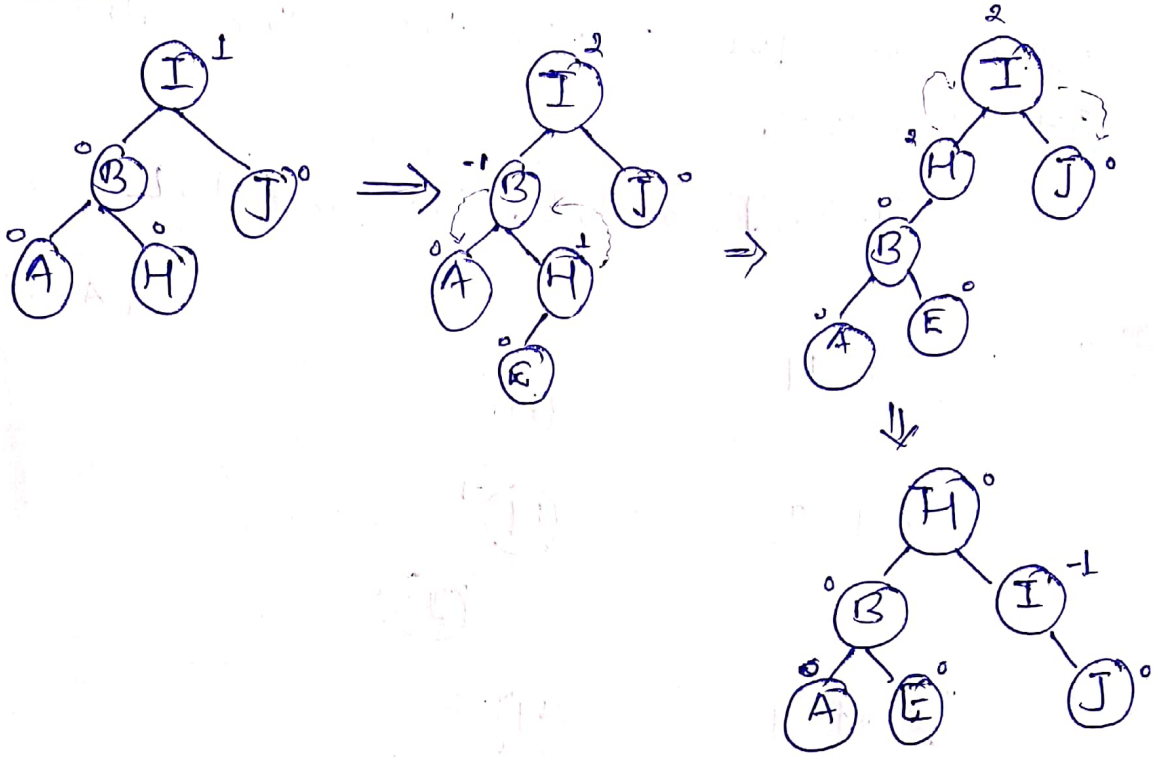
Insert B



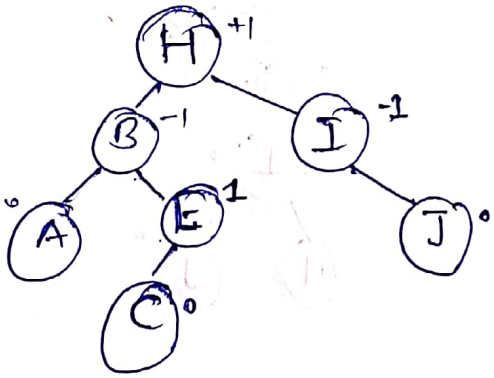
Insert A



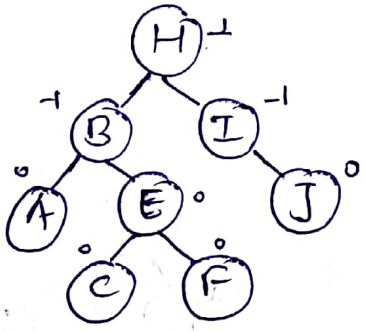
Scanned by CamScanner



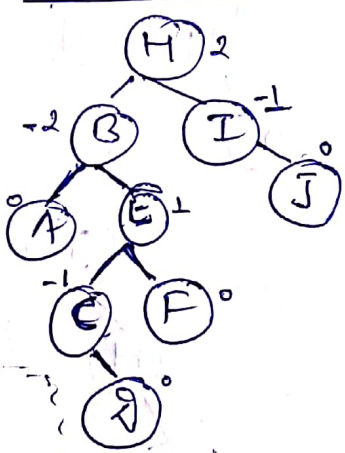
Insert C



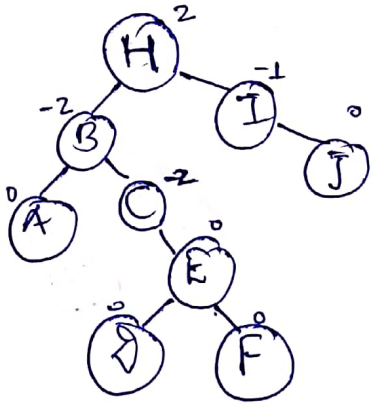
Insert F

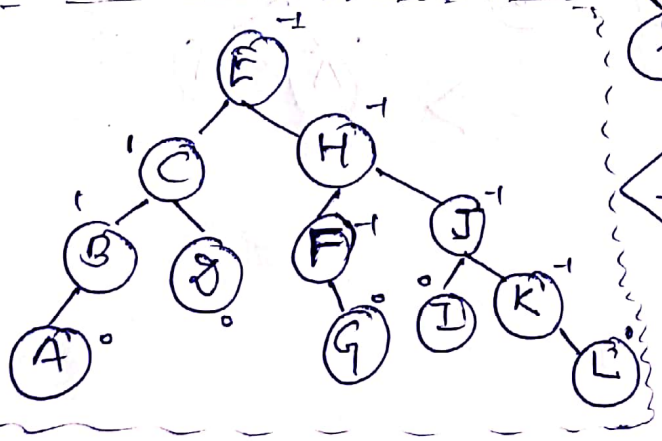


Insert D

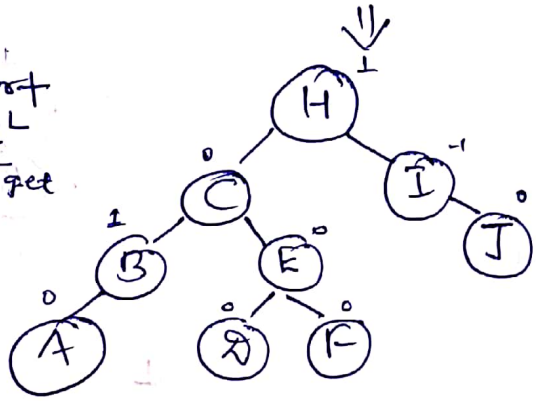


RL





Insert G, K, L you get



Scanned by CamScanner

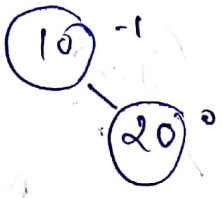
AVL

Q. Insert the following keys in order shown to Construct AVL trees.

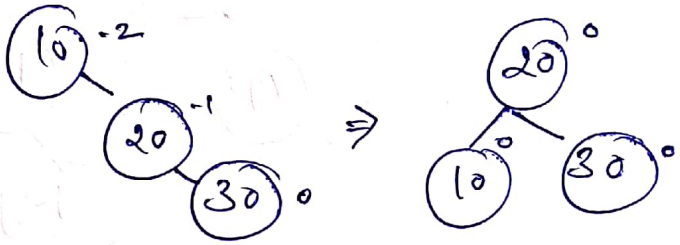
10, 20, 30, 40, 50

Ans: Insert 10:

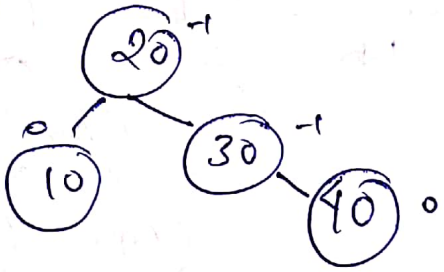
Insert 20:



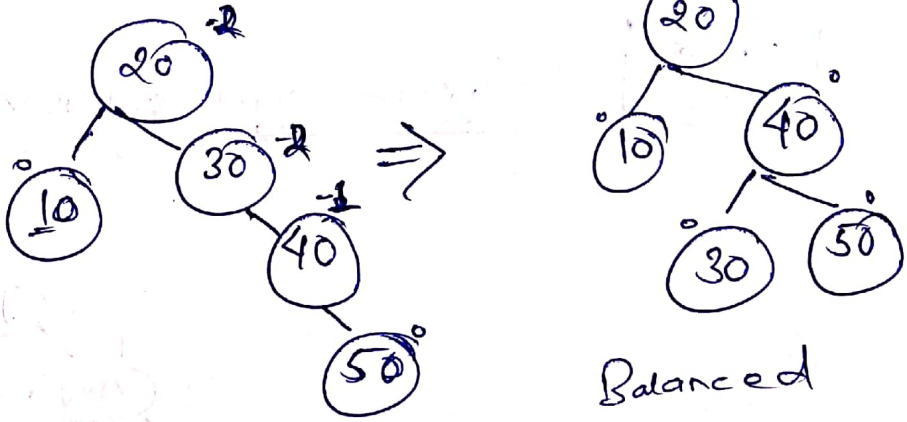
Insert 30:



Insert 40:



Insert 50:



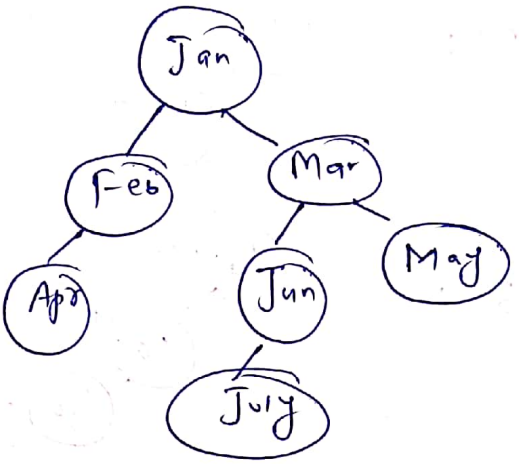
Balanced AVL Trees.

Scanned by CamScanner

Q3: Create AVL for : Jan, Feb, Mar, April, May, Jun, July, Aug, Sep, Oct, Nov, Dec.

Sol^{n}: Use Ascending order as like the dictionary Book.

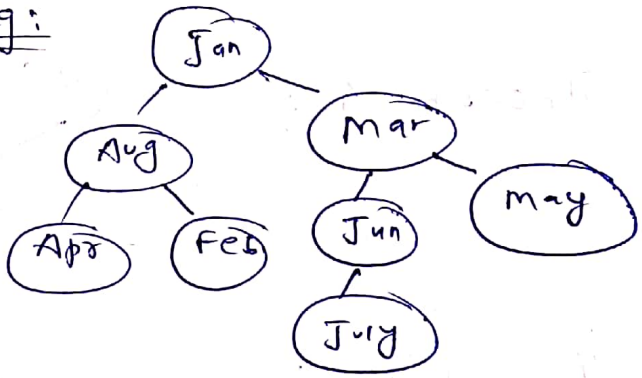
Hints: Insert Jan, Feb, Mar, Apr, may, Jun, July.



Assignment

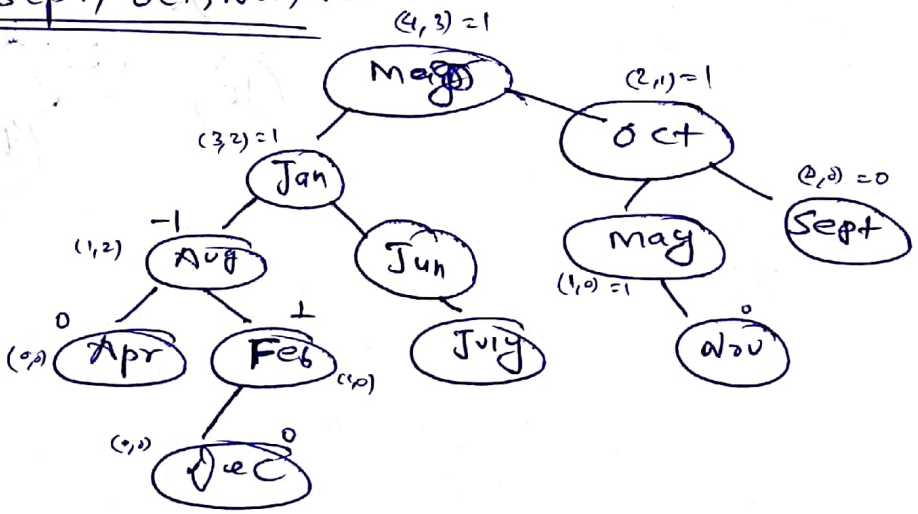
Insert Aug:

LR



Try:

Insert sept, oct, Nov, Dec



Balanced AVL Trees

Scanned by CamScanner

Q. Construct Binary Tree:

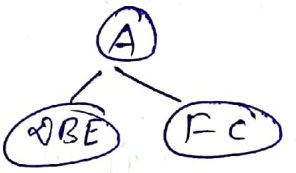
Inorder = DBEAFC

Preorder = ABDECF

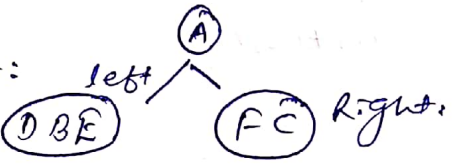
Soln:-

preorder ⇒ root - left - Right }
Inorder ⇒ left - root - Right }

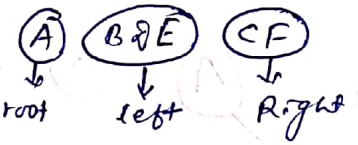
step 1: A is root ⇒ from preorder, so from Inorder DBE lies in left and FC lies on the Right.



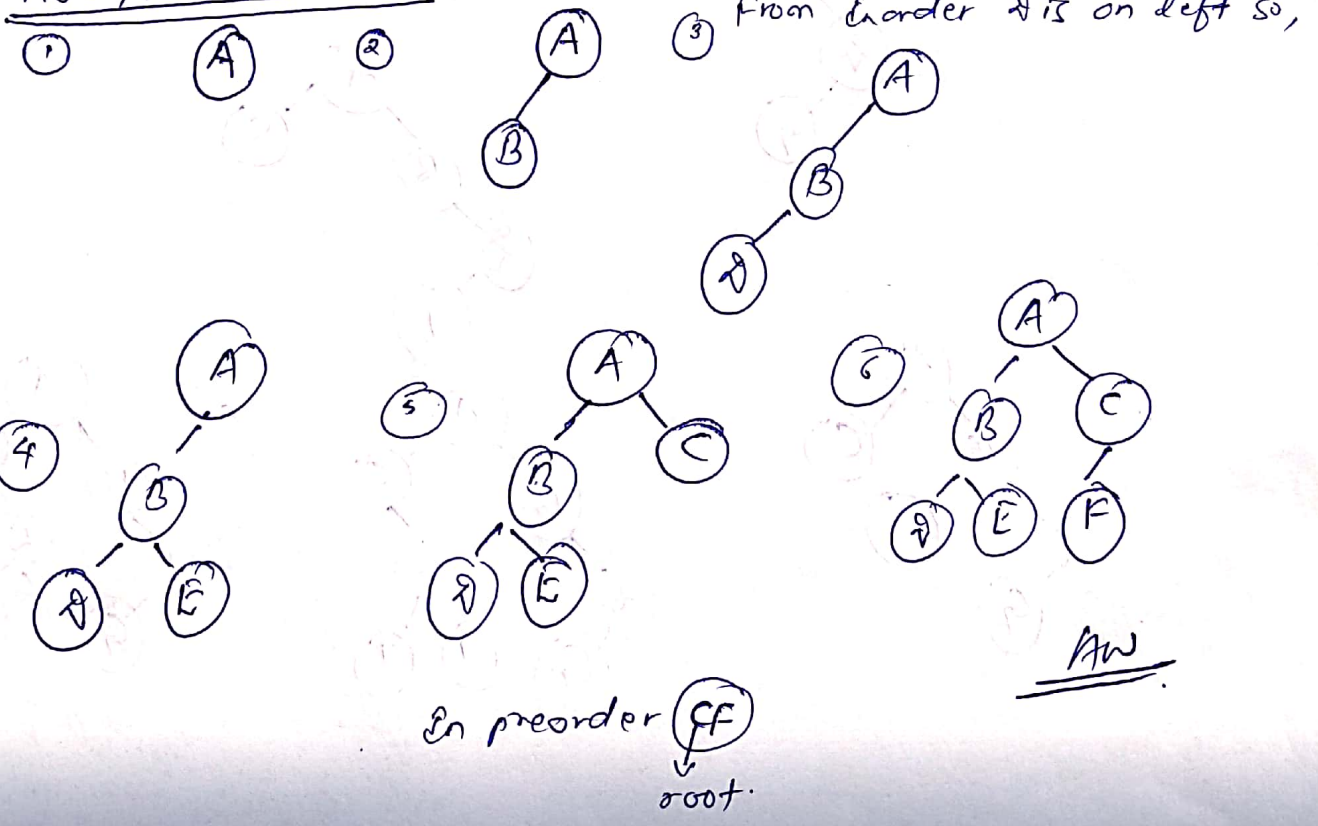
step 2: Inorder:



preorder:



Now, preorder ~~Scan~~



AW

In preorder CF

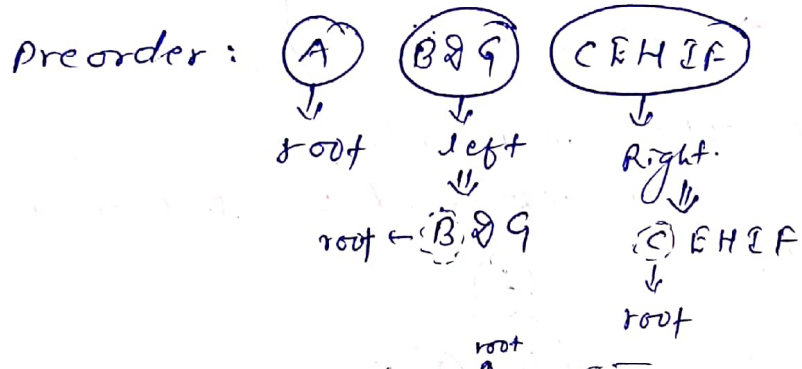
Scanned by CamScanner

Q2

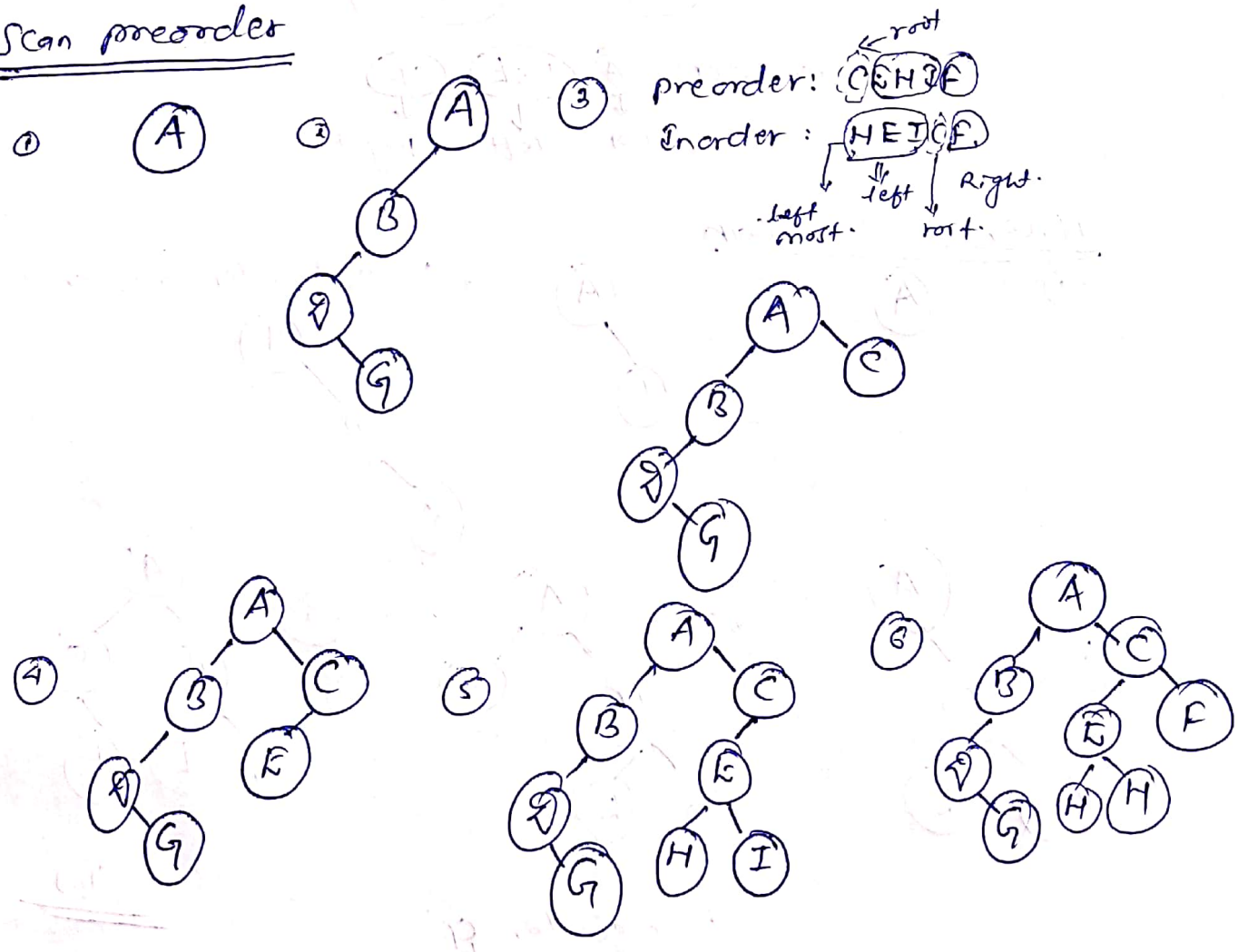
Preorder: A B D G C E H I F

Inorder = D G B A H E I C F

Sol^{n}: I, From preorder A is root so, D G B = left subtree
H E I C F = Right subtree.



Scan preorder



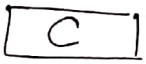
Scanned by CamScanner

① Create a B-tree of order 5.

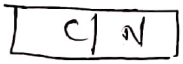
C N G A H E K Q M F W L T Z D P R X Y S.

Sol^{n}: order 5 means that a node can have maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys.

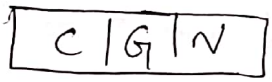
(a) Insert C.



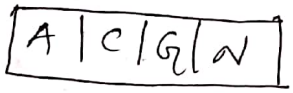
(b) Insert N.



(c) Insert G.

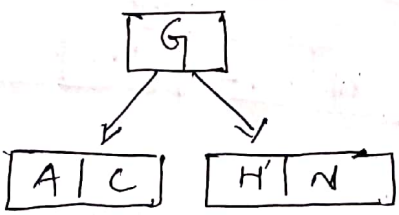


(d) Insert A.

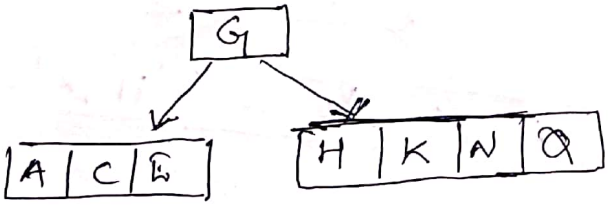


(e) Insert H.

order 5 so split it down.

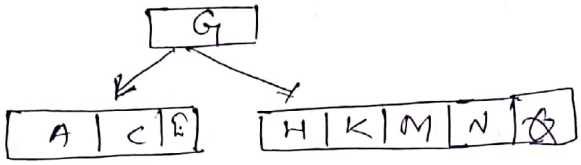


(f) Insert E, K, Q in B-tree.

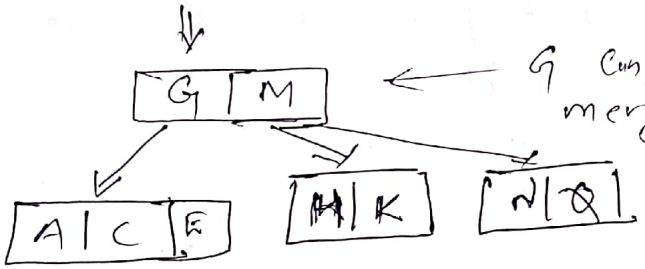


Scanned by CamScanner

(g) Insert M.

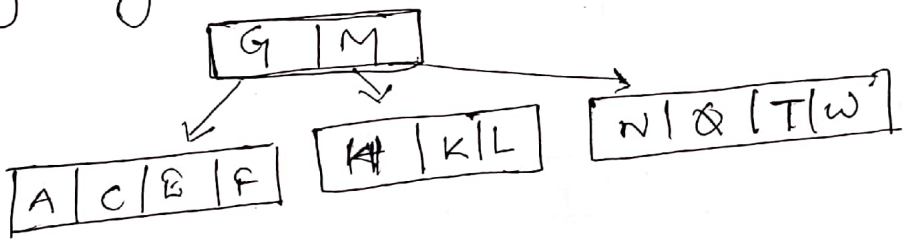


← order 5 split then,

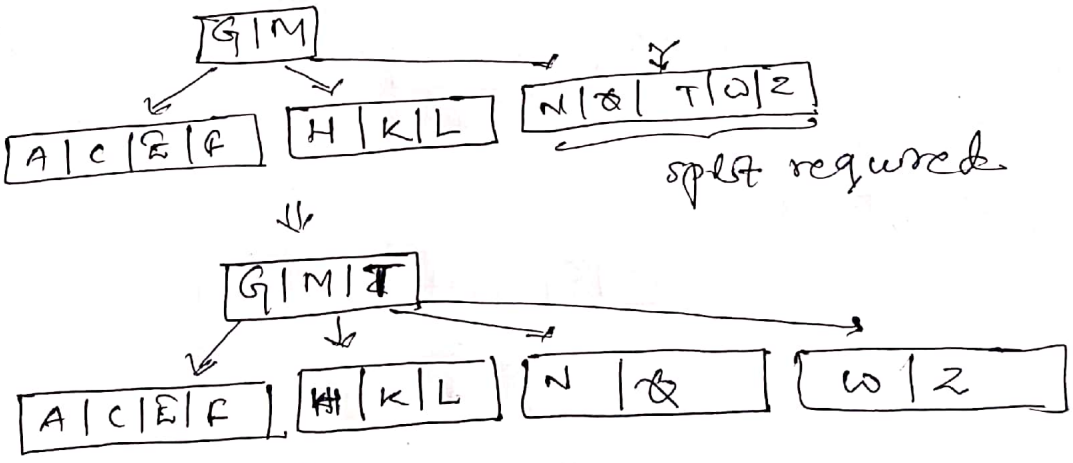


← G can have order upto 5 so, merging M

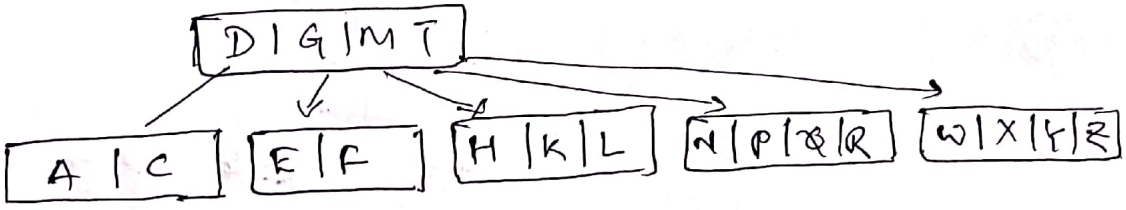
(h) Insert F, W, L and T are then added without needing any split.



(i) Insert Z.



(j) Insert D, P, R, X, Y into B-tree.



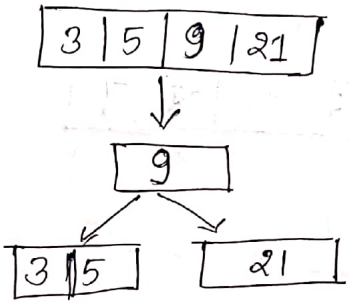
Scanned by CamScanner

Q: Create B-Tree of order 4

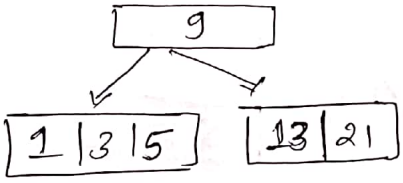
Insert ⇒ 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8.

Sol^n: ↓

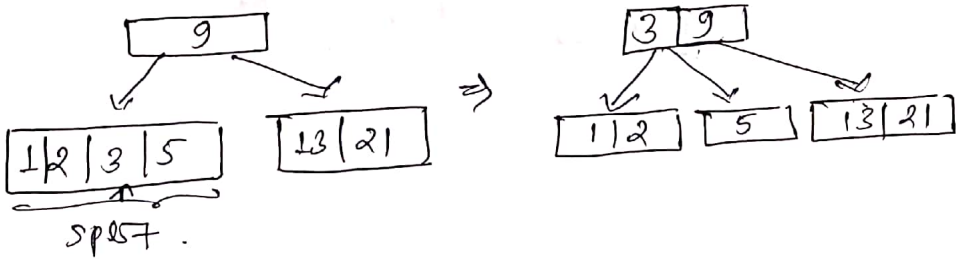
(a) Insert 5, 3, 21, 9



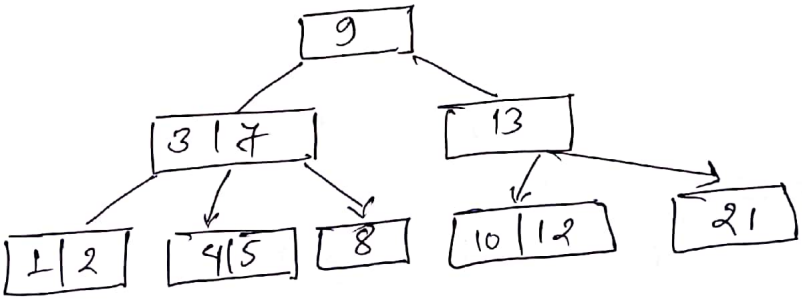
(b) Insert 1, 13,



(c) Insert 2;

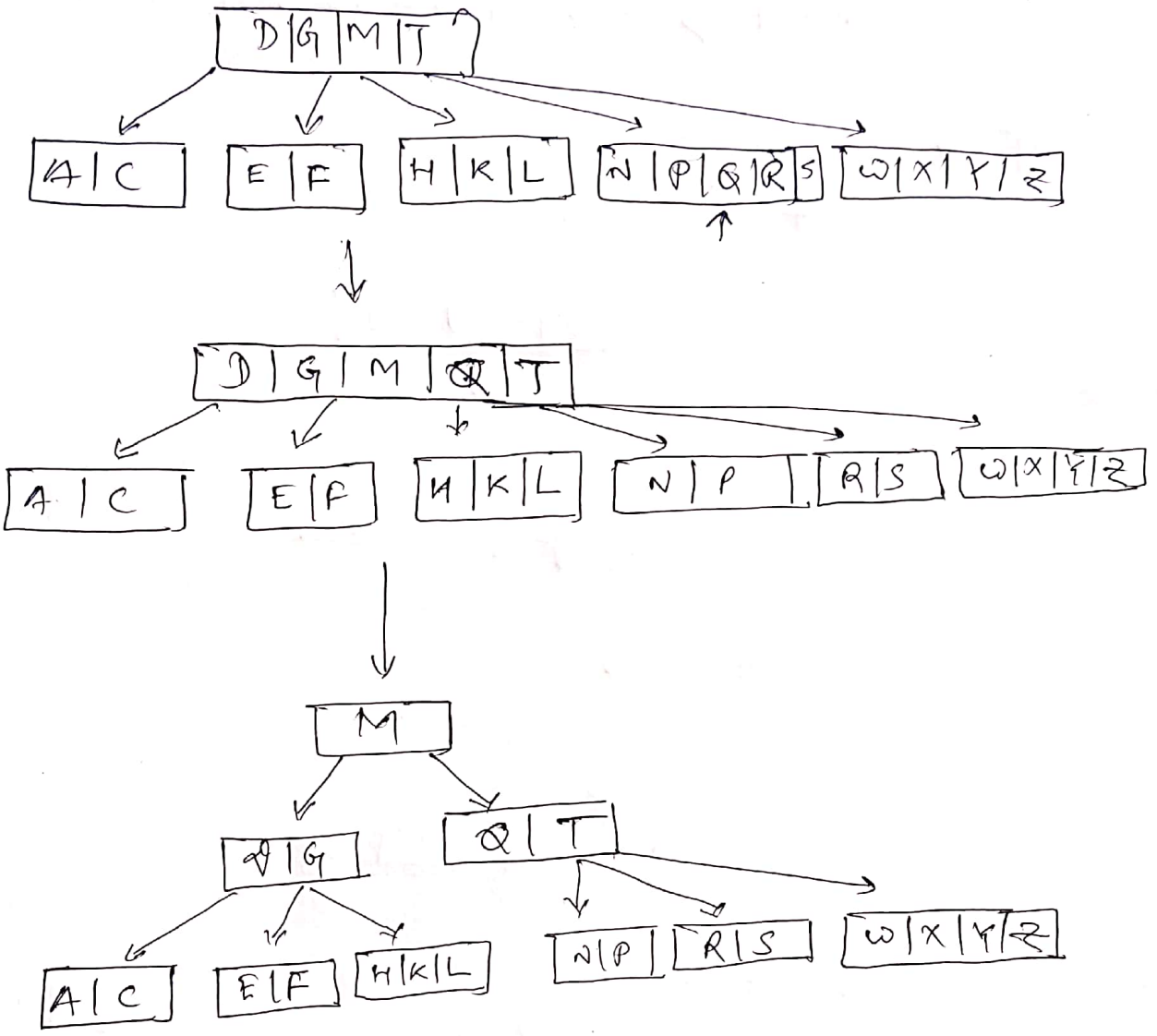


(d) Insert 7, 10, 12, 4, 8



Scanned by CamScanner

(K) finally, when S is added. then,

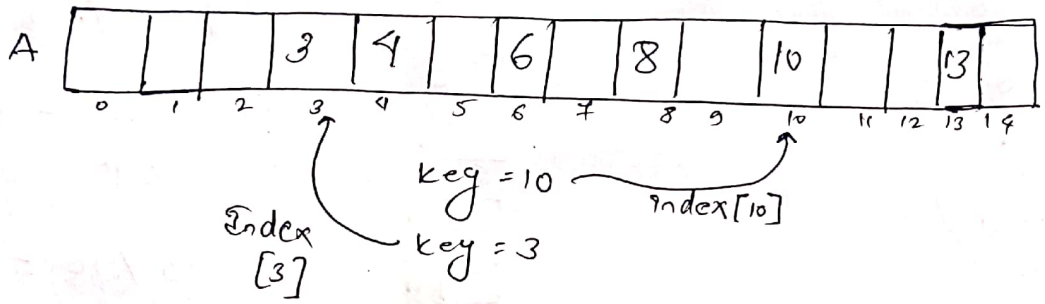


Scanned by CamScanner

① Direct Addressing.

Hashing.

keys: 8, 3, 13, 6, 4, 10, 50



It can be represented as:

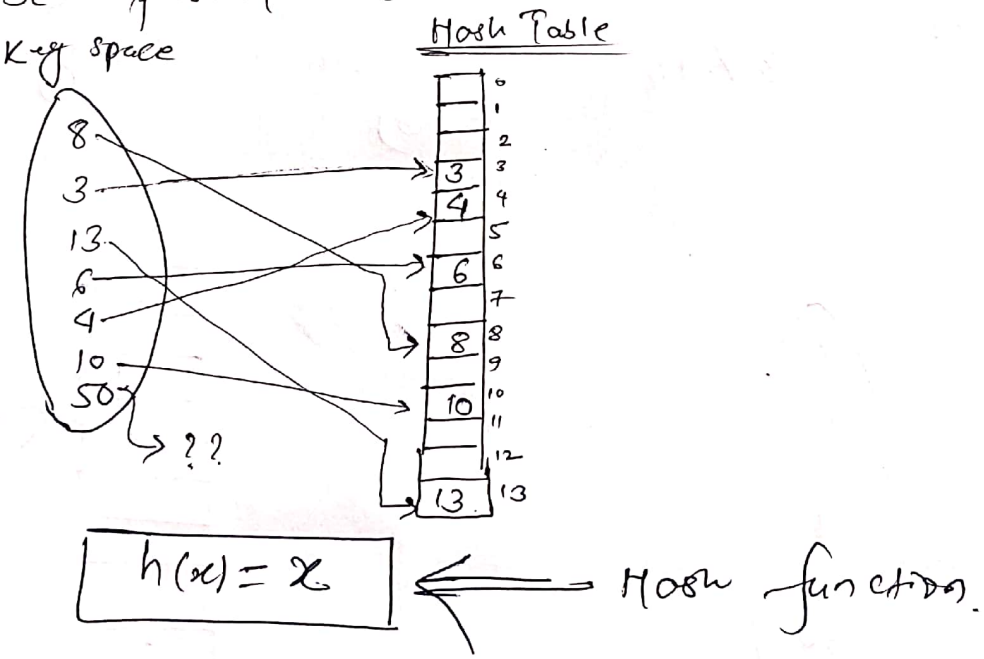
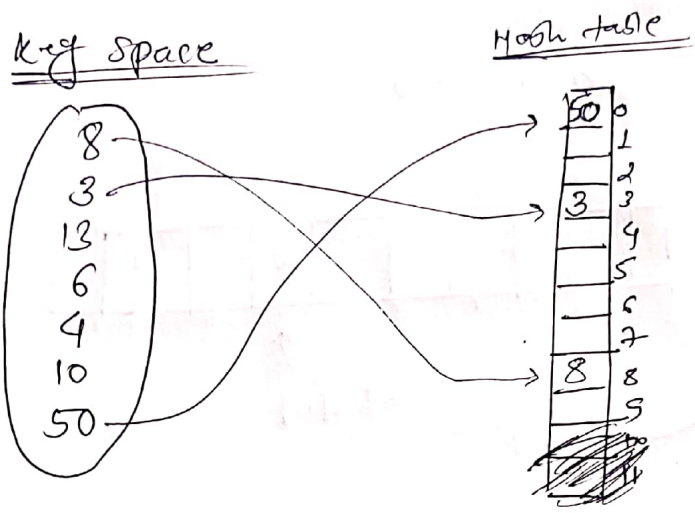


Fig 1.

Scanned by CamScanner

Using hash function as,

$h(x) = x \% \text{ size of an array}$



Size of an array = 10

$h(x) = x \% \text{ size}$

(a) $8 \% 10 = 8$

(b) $3 \% 10 = 3$.

(c) $50 \% 10 = 0$

Here space is saved but Collision may arises like: $13 \% 10 = 3$ (which is already occupied by 3)

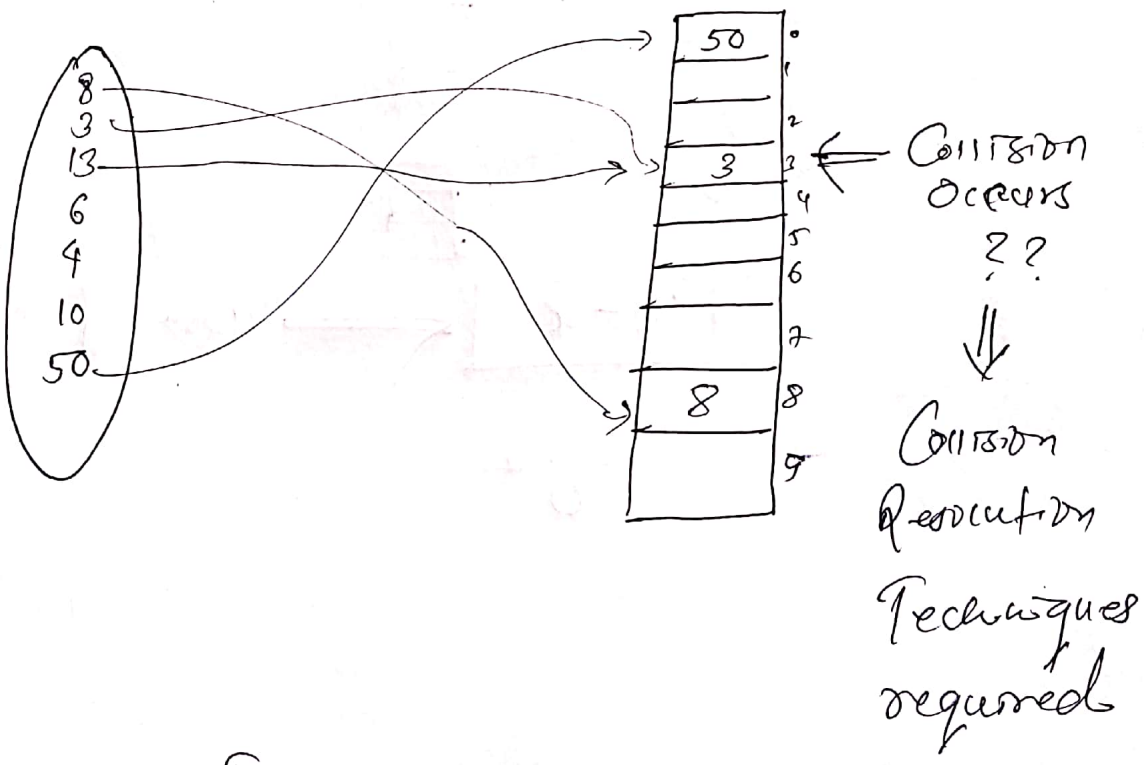


Fig: 2

Scanned by CamScanner

② Example of separate chaining.

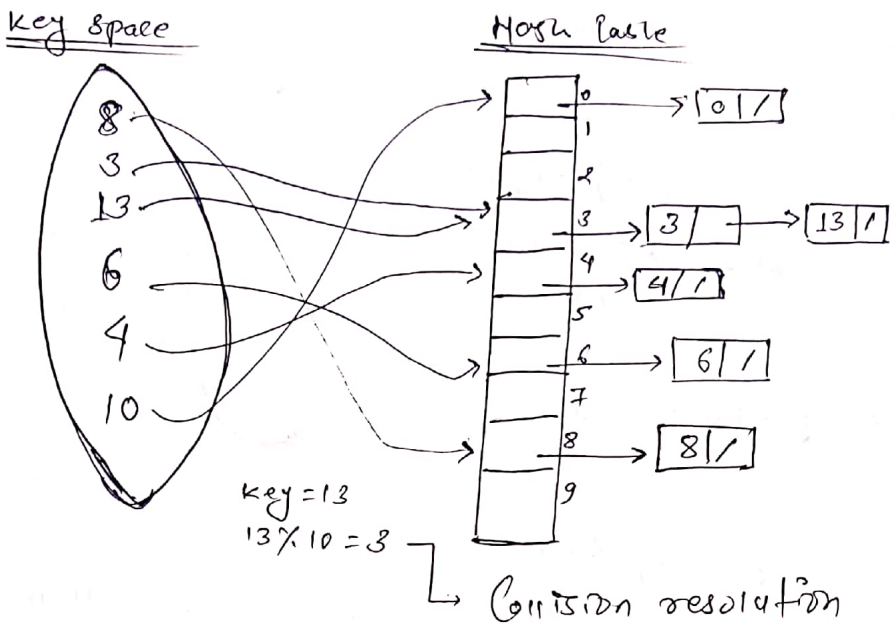


Fig. 3.

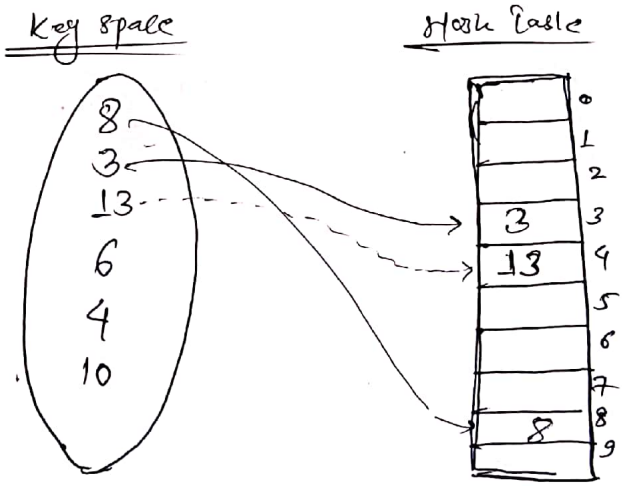
① Linear probing

Hash: $h(x) = x \% size$.

Rehash: $h'(x) = [h(x) + f(i)] \% size$

where,

$f(i) = i$, $i = 0, 1, 2 \ldots$



For 3

(a) $h(x) = [3 \% 10] = 3$

For 13

(b) $h(x) = [13 \% 10] = 3 \rightarrow$ Collision.

$\Rightarrow h'(x) = [3 + f(0)] \% 10$, $i = 0$, $f(i) = i$, $f(0) = 0$

$= [3 + 0] \% 10$

$= 3 \rightarrow$ again collision

$\Rightarrow h'(x) = [3 + 1] \% 10$, $i = 0$, $f(i) = i$, $f(1) = 1$

$= 4 \% 10$

$= 4$

empty so put 13 there

fig (4).

Scanned by CamScanner

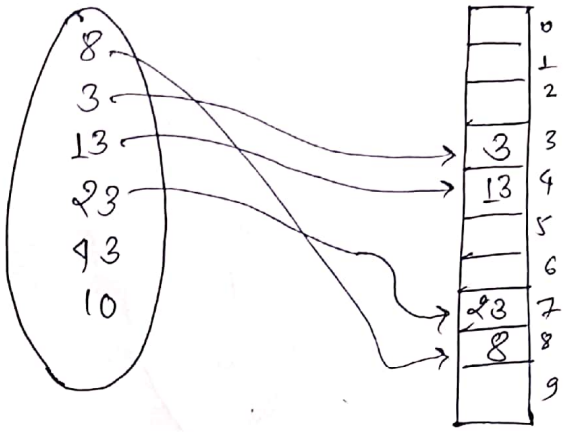
ⓒ Quadratic Probing.

Hash, $h(x) = x \,\%.\, size$

Rehash, $h'(x) = [h(x) + f(i)] \,\%.\, size$

$\therefore f(i) = i^2$

$i = 0, 1, 2, \ldots$



For 3

(i) $h'(x) = [3+0] \,\%.\, 10$

$= 3$

For 13

(ii) $h'(x) = [3+1] \,\%.\, 10$

$= 4$ ⟶ Collision so resolution using rehash.

For 23

$h'(x) = [3+2^2] \,\%.\, 10$

$= [3+4] \,\%.\, 10$

$= 7 \,\%.\, 10$

$= 7$ ⟶ Again collision so, using rehash.

Fig 5

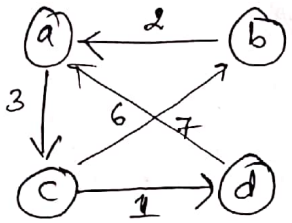
Scanned by CamScanner

Floyd's Algorithm (pseudocode and analysis).

Algorithm:

```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D.
```

Example:



Step 1:

$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

⇒ b → c
min [∞, 2+3]
= 5

⇒ c → c
min [0, ∞+3]
= 0

⇒ d → c
min [∞, 6+3] = 9

Step 2:

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

Step 3:

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

Step 4:

$D^{(3)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

Scanned by CamScanner

Step 5

$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Hence this is the required soln:

Scanned by CamScanner

Q1 Warshall's Algorithm: Transitive Closure:

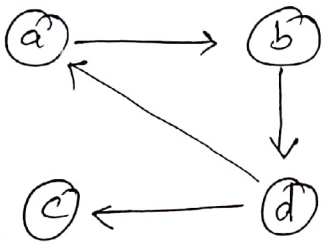
$R^{(K-1)}$ =

	J	K
K	1	
i	0 →	1

$\Rightarrow R^{(K)}$ =

	j	K
K	1	
i	1	1

Q.



Step 1 $R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

$\Rightarrow$

Step 2 $R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

Step 3 $R^{(2)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Step 4 $R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Scanned by CamScanner

$$R^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

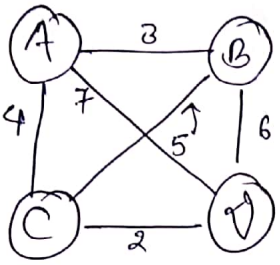
Hence, the transitive closure of given adjacency matrix is,

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

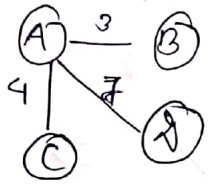
Scanned by CamScanner

Spanning Trees

① Consider a graph G, with defined weight as,

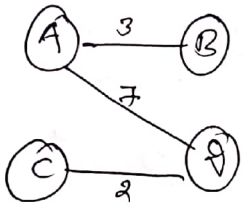


The possible spanning Trees are



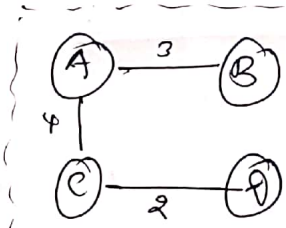
(i)

Total cost
= 4+7+3
= 14



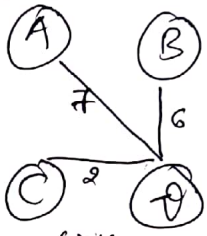
(ii)

Total cost
= 7+3+2
= 12



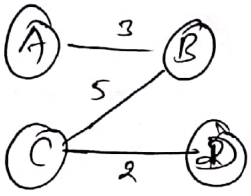
(iii)

Total cost
= 4+3+2
= 9



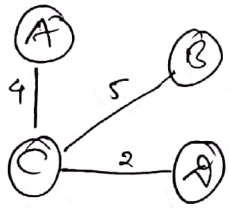
(iv)

Total cost
= 7+2+6
= 15

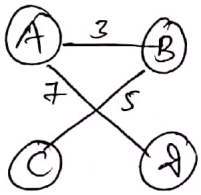


(v)

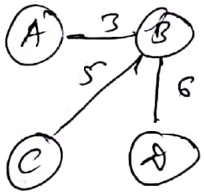
Total cost
= 3+5+2
= 10



(vi) Total cost
= 4+5+2
= 11



(vii) Total cost
= 7+3+5
= 15



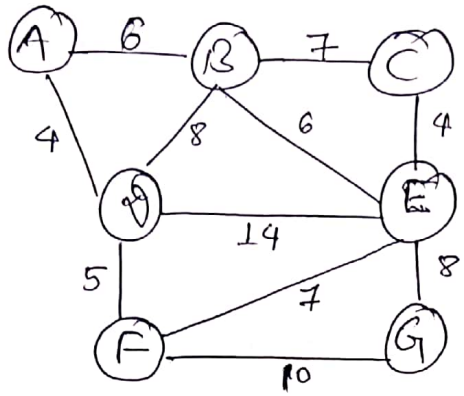
(viii) Total cost
= 3+5+6
= 14

Among the given spanning Trees the one highlighted have minimum weight cost and is considered as the minimum spanning Trees.

fig ⑨.

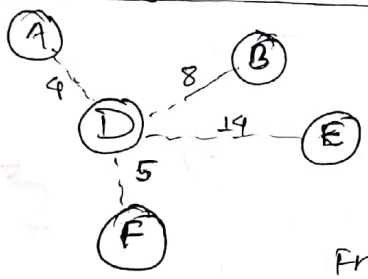
Scanned by CamScanner

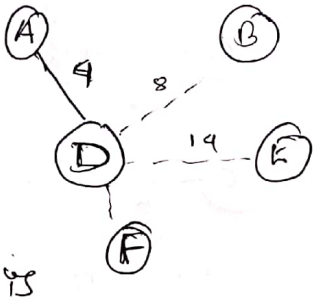
Prim's algorithm Example:



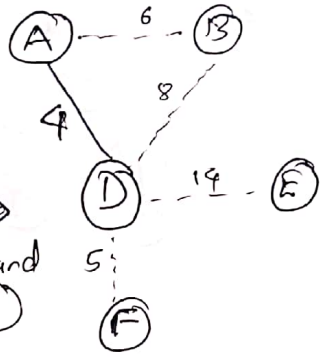
Starting Node D:

(1)

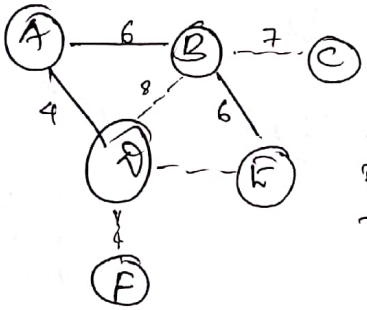




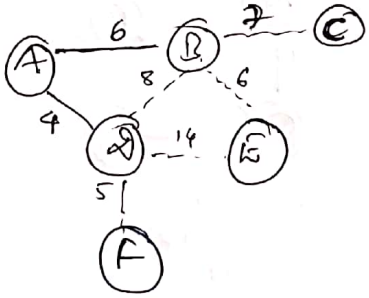
From A, 4 is minimum so Connect it



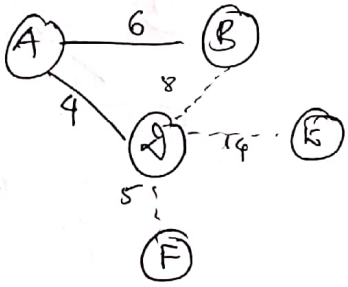
From A, B is 6



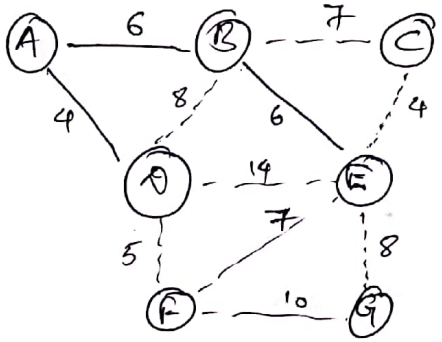
From B to E



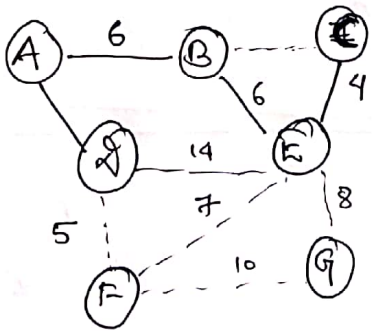
Expand B.

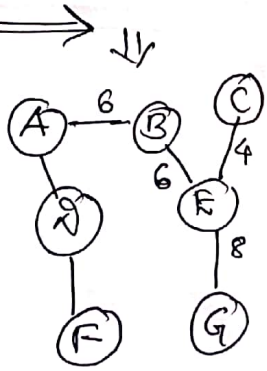


Expand E



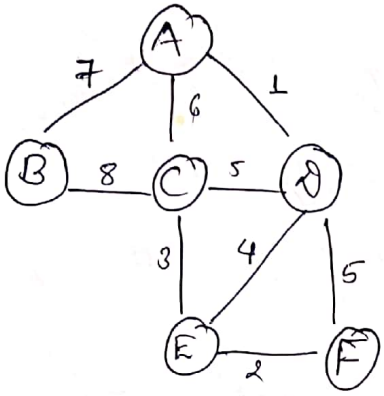
E to C





Scanned by CamScanner

Kruskal's algorithm examples:



Initially, we have,

$F = \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}\}$

$MST = \{\}$

$Q = \{\overset{1}{(A,D)}, \overset{2}{(E,F)}, \overset{3}{(C,E)}, (E,D), (C,D), (D,F), (A,C), (A,B), (B,C)\}$

where Q is priority queue in which the edges that have minimum weight takes a priority over any other edge in the graph.

Step 1 Remove the edge (A,D) from Q and make the following changes.

- $F = \{\{A,D\}, \{B\}, \{C\}, \{E\}, \{F\}\}$
- $MST = \{A, D\}$
- $Q = \{(E,F), (C,E), (E,D), (C,D), (D,F), (A,C), (A,B), (B,C)\}$

Step 2: Remove the edge (E,F) from Q and make the following changes.

- $F = \{\{A,D\}, \{B\}, \{C\}, \{E,F\}\}$
- $MST = \{(A,D), (E,F)\}$
- $Q = \{(C,E), (E,D), (C,D), (D,F), (A,C), (A,B), (B,C)\}$

Step 3: Remove the edge (C,E) from Q and make the following changes.

- $F = \{\{A,D\}, \{B\}, \{C,E,F\}\}$ $(\because (C,E)(E,F)\}$
- $MST = \{(A,D), (C,E), (E,F)\}$
- $Q = \{(E,D), (C,D), (D,F), (A,C), (A,B), (B,C)\}$

Scanned by CamScanner

Step4: Remove the edge (E,D) from Q and make the following changes.

F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}

Step5: Remove the edge (C, D) from Q. Note that when (C, D) is added it forms closed loop so, its discarded.

Next, (D, F), (A, C), are discarded.

Step 6: Remove the edge (A, B) from Q and make changes.

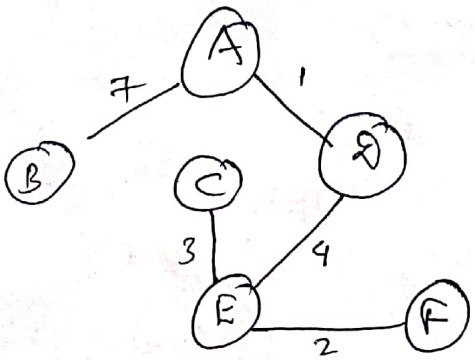
F = {{A, B, C, D, E, F}}

MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}

Q = {(B, C)}

Step 7: (B, C) again form closed loop so discarded it and form spanning Trees finally.

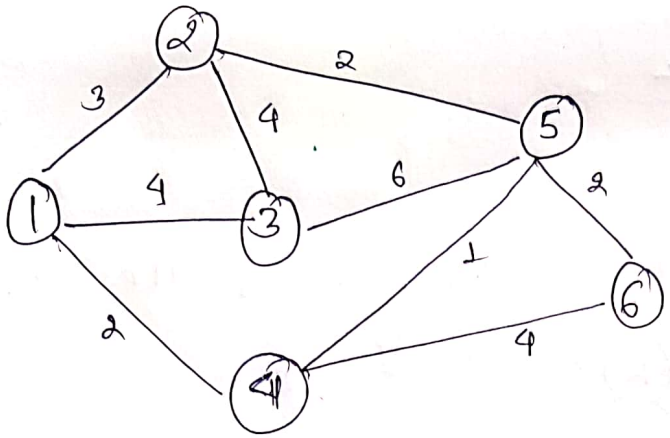
So, MST = {(A, D), (C, E), (E, F), (E, D) (A, B)}



2.

Scanned by CamScanner

(1)



from (1) to every node Calculate Shortest distance using Dijkstra's Algorithm

Step 1

1 → 2 = 3
1 → 3 = 4
1 → 4 = 2 (preferred)

Step 2

For 2

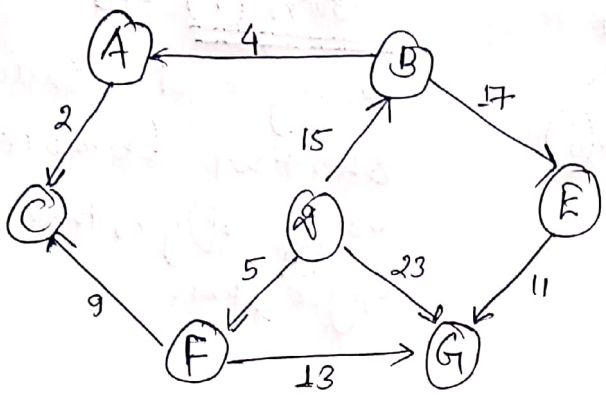
min (1→2, 1→3→2)
⇒ min (3, 4+4) = 3 (1→2 preferred)

For 3

min (1→3, 1→2→3)
⇒ min (4, 7) = 4 (1→3 preferred)

Scanned by CamScanner

Dijkstra's Algorithm:



Take D as the initial node.

Sol^{n}: ↓

Step 1:

Set the label of D = 0 and N = {D}

Step 2:

Label of D = 0, B = 15, G = 23, and F = 5. Therefore N = {D, F}

Step 3:

Label of D = 0, B = 15, D → F → G = 18. So, G has been re-labelled 18. because min (5+13, 23) = 18.

N = {D, F, G}.

Step 4:

C has been re-labelled as, (5+9) = 14

Therefore, N = {D, F, C} ∵ min(14, 15+4+2)
min(14, 21) = 14

Step 5:

Label of D = 0, B = 15, G = 18, C = 14.

A = 15+4 = 19

Therefore N = {D, F, G, B, C, A}

Step 6:

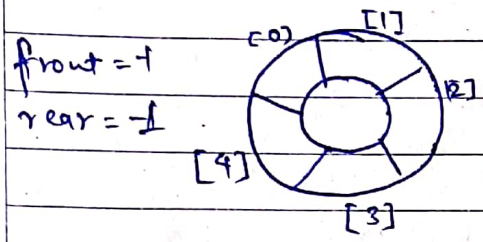
E is not reachable from D so,

N = {D, F, G, B, C, A, E} = {0, 5, 18, 15, 14, 19, ∞}

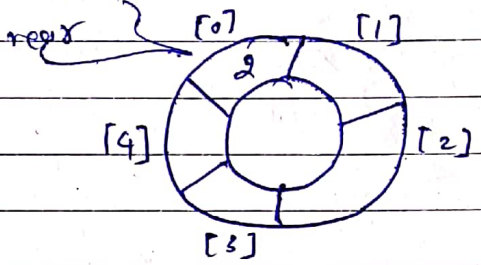
Scanned by CamScanner

Circular queue:

Max size = 5

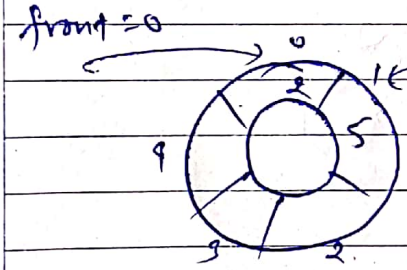


Insert 2

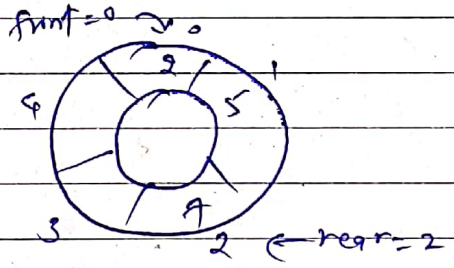


front = 0
rear = 0
rear ⇒ (1+rear) % maxsize
= (1-1) % 5
= 0

Insert 5

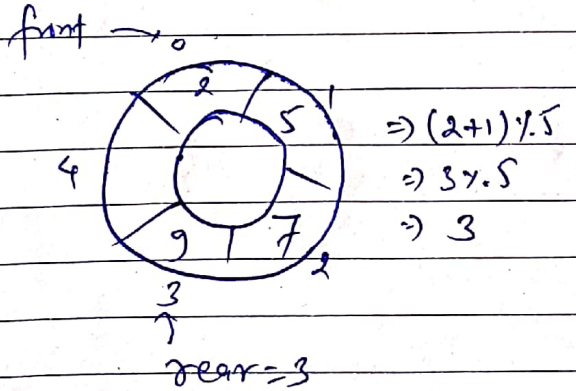


Insert 4

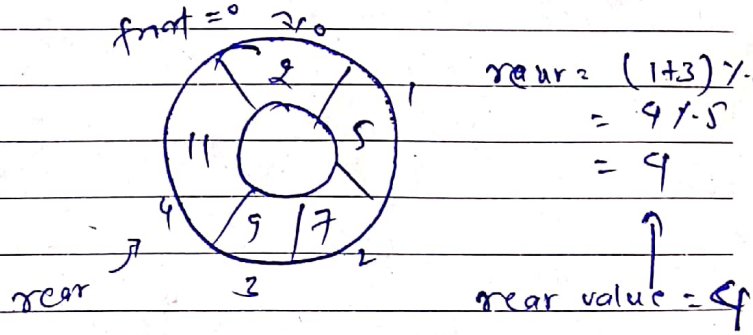


= (1+1) % 5
= 2 % 5
= 2

Insert 9



Insert

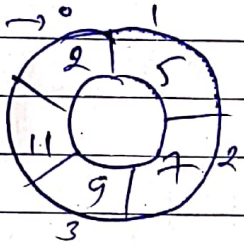


⇒ Now, rear = 4
rear = (4+1) % 5 = 0 Here, rear value = front value
ie, when,
front = (1+rear) % Maxsize then
Overflow occurs

Scanned by CamScanner

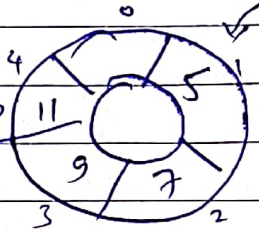
Now, Delete some values

front = 0 → 0



Delete 2

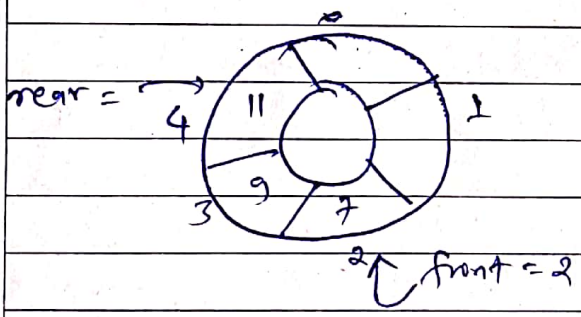
front = 1



= (1 + front) % max
= (1 + 0) % 5
= 1 % 5
= 1

Delete 9

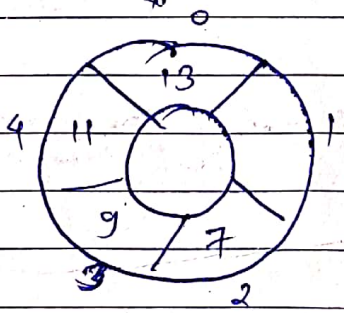
front = (1 + 1) % 5
= 2 % 5
= 2.



Now, 0 and 1 position are vacant so, this can be filled with values.

Enqueue (insert 13)

rear = 0

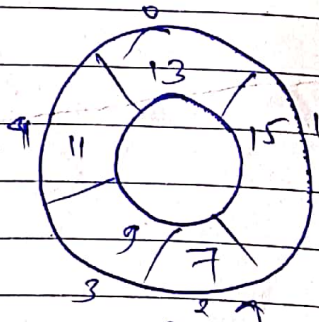


front = 2

rear = (4+1) % 5
= 5 % 5
= 0

Insert 15

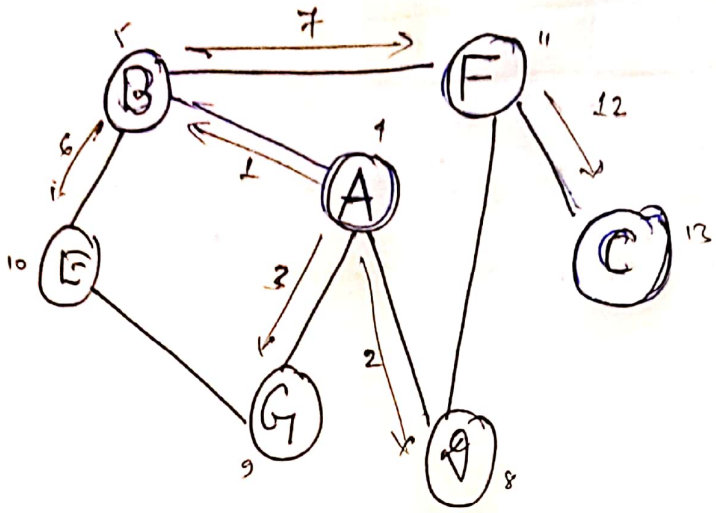
rear = (0+1) % 5
= 1 % 5
= 1



front = 2

Scanned by CamScanner

Breadth First Search: (Source: techdifferences.com)

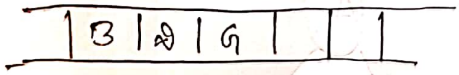


Use : Queue:

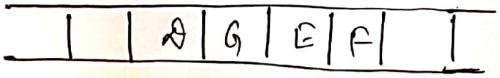
* Starting from A:

(a) Vertex A is expanded and stored in the Queue.

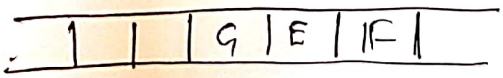
(b) Vertexes B, D and G successors of A, are expanded and stored in the queue, meanwhile vertex A removed.



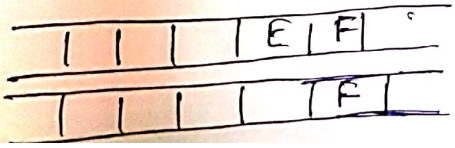
(c) Now, B at the front end of the queue is removed along with storing its successors vertices E and F.



(d) Vertex D is at the front end of the queue is removed, and its connected node F is already visited.

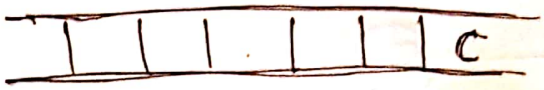


(e) G removed has its successor E which is already visited, now, E is removed.



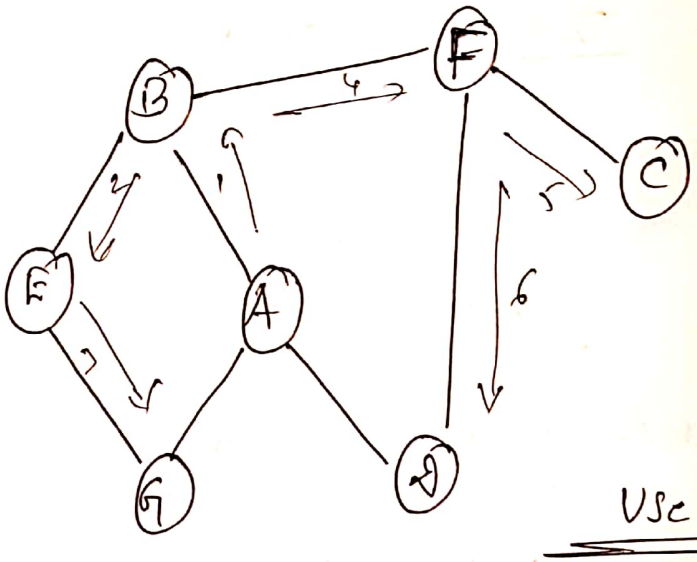
Scanned by CamScanner

(6) III'd, F is removed and its successor vertex C is traversed and stored in the queue.



(7) At last C is also removed and queue is empty which means we are done and generated output is A, B, D, G, E, F, C

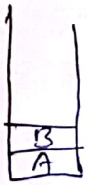
(2) Depth First Search:



Use: Stack:

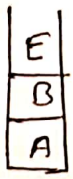
(a) A as the starting vertex which is explored and stored in stack

(b) B successor vertex of A is stored in stack



Scanned by CamScanner

(c) Vertex B have two successors E and F, among them alphabetically E is explored first and stored in the stack

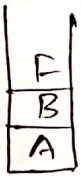


(d) successor of E, G is stored in stack,

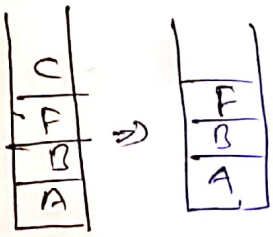
(e) G has two successors, Both are visited so pop out from stack.

(f) If D. E also removed.

(g) B at top; F is ~~unvis~~ not visited. and stored in stack.



(h) F has C, D So, C traversed first.



(i) C, removed,

(j) D is now visited of F and stored in stack.

(k) Now, all D, F, B, A are popped, stack is empty. Output: Serially visited nodes A B E G F C D.

Scanned by CamScanner