

UNIT I -- LISTS

Abstract Data Types (ADTs) – List ADT – Array-based implementation – Linked list implementation – Singly linked lists – Circularly linked lists – Doubly-linked lists – Applications of lists – Polynomial ADT

1. INTRODUCTION

- Computer science is a field of study that deals with solving a variety of problems by using computers.
- To solve a given problem by using computers, you need to design an algorithm for it.
- Multiple algorithms can be designed to solve a particular problem.
- An algorithm that provides the maximum efficiency should be used for solving the problem.
- The efficiency of an algorithm can be improved by using an appropriate data structure.

Role of Algorithms and Data Structures in Problem Solving

- Problem solving is an essential part of every scientific discipline.
- Computers are widely being used to solve problems pertaining to various domains, such as, banking, commerce, medicine, manufacturing, and transport..
- To solve a given problem by using a computer, you need to write a program for it.
- A program consists of two components, algorithm and data structure.
Program = Data Structures + Algorithms
- Data structures help in creating programs that are simple, reusable, and easy to maintain.

Role of Algorithms

- An algorithm can be defined as a step-by-step procedure for solving a problem.
- An algorithm helps the user arrive at the correct result in a finite number of steps.
- An algorithm has five important properties:
 - ✓ **Finiteness** – An algorithm terminates after a finite number of steps.
 - ✓ **Definiteness** – Each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted in multiple ways and can be performed without any confusion.
 - ✓ **Input** – An algorithm accepts zero or more inputs.
 - ✓ **Output** – An algorithm produces atleast one output.
 - ✓ **Effectiveness** – An algorithm consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

2. DATA STRUCTURE

- Data → Values or set of values of different data types such as int, float, etc.,
- Structure → Way of organizing and storing information so that it is easy to use
- **Data structure is defined as a way of organizing the various data elements in memory with respect to each other.**

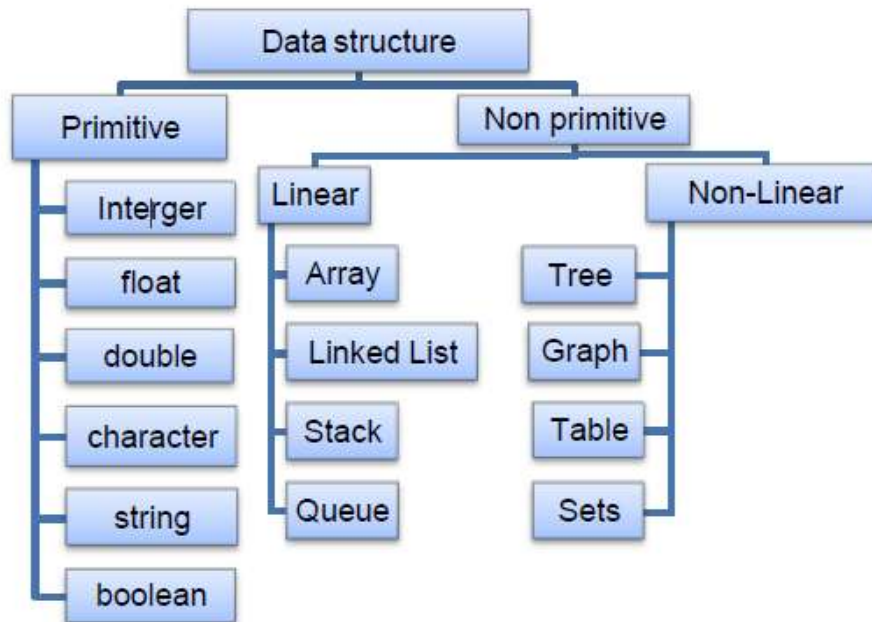
- **Classification of Data Structures**

- ✓ **Primitive Data Structure:**

- A Primitive Data Structure used to represent the standard data types of any one of the computer languages (integer, Character, float etc.).

✓ **Non - Primitive Data Structure :**

- A Primitive Data Structure can be constructed with the help of any one of the primitive data structure.
- It can be structure and it is having a specific functionality.
- It can be designed by user.
- It can be classified as Linear and Non-Linear Data Structure.



✓ **Linear Data Structures:**

- A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached.
- Ex: Arrays, Linked Lists, Stacks, Queues

✓ **Non-Linear Data Structures:**

- Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure.
- Ex: Trees, Graphs, Heaps

▪ **Operations on Data Structures**

The basic operations that are performed on data structures are as follows:

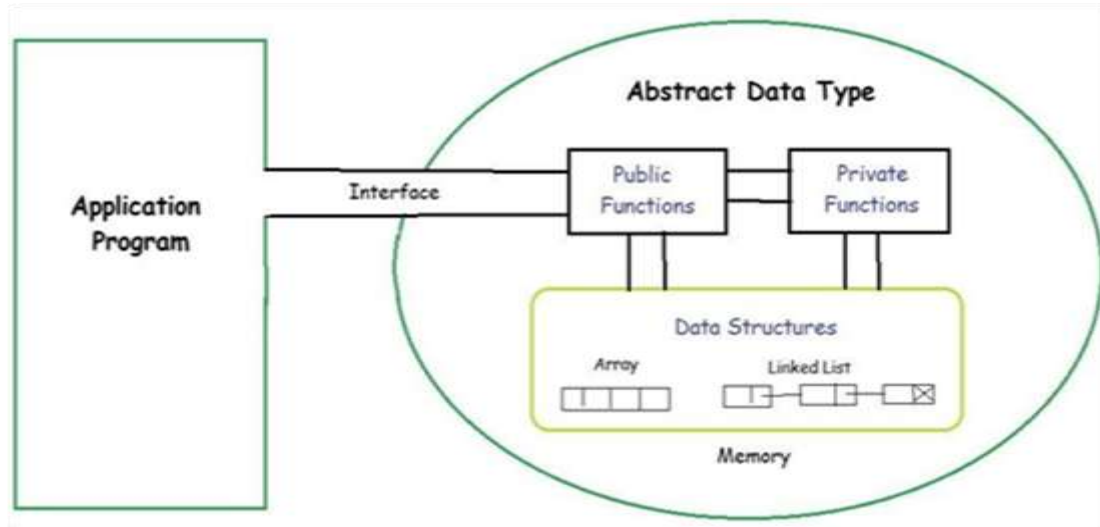
- ✓ **Traversal:** Traversal of a data structure means processing all the data elements present in it exactly once.
- ✓ **Insertion:** Insertion means addition of a new data element in a data structure.
- ✓ **Deletion:** Deletion means removal of a data element from a data structure if it is found.
- ✓ **Searching:** Searching involves searching for the specified data element in a data structure.
- ✓ **Sorting:** Arranging data elements of a data structure in a specified order is called sorting.
- ✓ **Merging:** Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

3. ABSTRACT DATA TYPE

- In programming each program is breakdown into modules, so that no routine should ever exceed a page. Each module is a logical unit and does a specific job modules which inturn will call another module.

Modularity has several advantages

1. Modules can be compiled separately which makes debugging process easier.
 2. Several modules can be implemented and executed simultaneously.
 3. Modules can be easily enhanced.
- Abstract Data type is an extension of modular design.
 - An abstract data type is a set of operations such as Union, Intersection, Complement, Search etc.,
 - The basic idea of implementing ADT is that the operations are written once in program and can be called by any part of the program.
 - ADT is a type for objects whose behavior is defined by set of values and set of operations from user point of view.
 - Mentions only what operations are to be performed but not how they will be implemented.
 - Thus, ADT is the specification of a data type within some language, independent of implementation.



- Examples – List ADT, Stack ADT, Queue ADT, etc.,
- Some operations on these ADT are:
 - ✓ List: insert(x), delete(x), find(x)
 - ✓ Stack: Push (x), Pop(), isFull(), isEmpty()
 - ✓ Queue: enqueue(), dequeue(), isFull(), isEmpty()

4. THE LIST ADT

- List is an ordered set of elements.
- The general form of the list is $A_1, A_2, A_3, \dots, A_N$
 - A_1 - First element of the list
 - A_N - Last element of the list
 - N - Size of the list
- If the element at position i is A_i then its successor is A_{i+1} and its predecessor is A_{i-1} .
- **Various operations performed on List**
 - ✓ Insert (X, 5) - Insert the element X after the position 5.
 - ✓ Delete (X) - The element X is deleted
 - ✓ Find (X) - Returns the position of X.

- ✓ Next (i) - Returns the position of its successor element i+1.
- ✓ Previous (i) - Returns the position of its predecessor i-1.
- ✓ Print list - Contents of the list is displayed.
- ✓ Makeempty - Makes the list empty.

▪ **Implementation of List ADT**

1. Array Implementation
2. Linked List Implementation
3. Cursor Implementation.

5. ARRAY IMPLEMENTATION OF LIST

- Array is a collection of specific number of data stored in a consecutive memory locations.
- Array is a Collection of elements of same data type.
- Elements in an array is stored in adjacent memory location.
- Syntax: datatype arrayname [size]
- Eg: int A [6];

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

▪ **Advantages of array implementation:**

- ✓ The elements are faster to access using random access.
- ✓ Searching an element is easier.

▪ **Limitations of array implementation:**

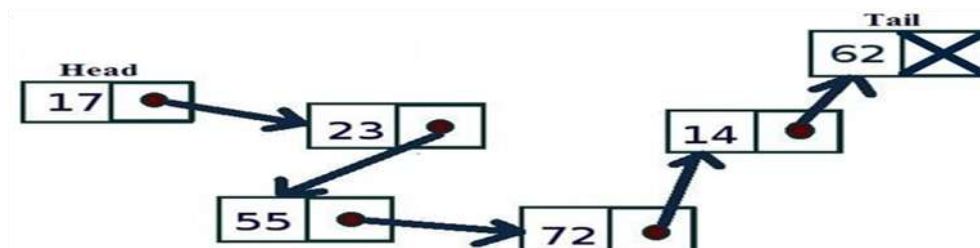
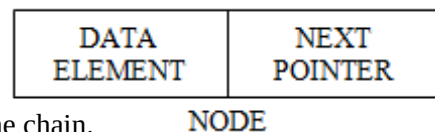
- ✓ An array store its nodes in consecutive memory locations.
- ✓ The number of elements in the array is fixed and it is not possible to change the number of elements.
- ✓ Insertion and deletion operation in array are expensive. Since insertion is performed by pushing the entire array one position down and deletion is performed by shifting the entire array one position up.

▪ **Applications of arrays:**

- ✓ Arrays are particularly used in programs that require storing large collection of similar type data elements.

6. LINKED LIST IMPLEMENTATION

- Linked list is a linear data structure.
- Made up of chain of nodes.
- Each node contains a data and a pointer to the next node in the chain.
- Last node of the list is pointed to NULL.
- Each node contains two fields:
 - ✓ Data field: Contains data element to be stored.
 - ✓ Pointer field: Contains the address of next node.

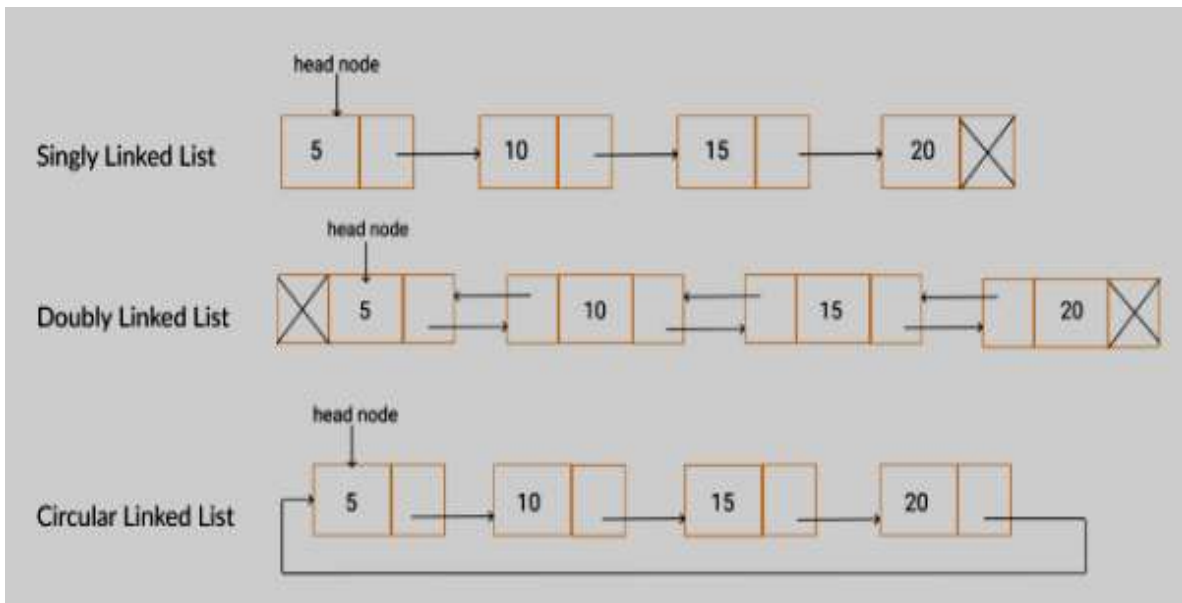


TYPES OF LINKED LIST

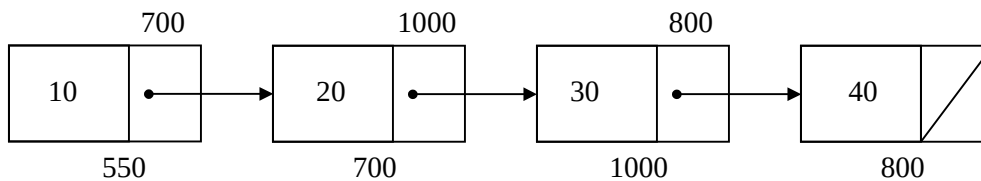
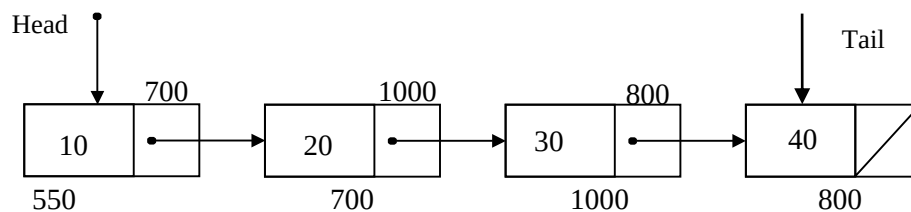
1. Singly Linked List

2. Doubly Linked List

3. Circular Linked List.

**6.1 SINGLY LINKED LIST**

- A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.

**Fig. 1.5.1 (a) Linked List****Fig. 1.5.1 (a) Linked List with a Head and Tail Pointer****Operations on Singly Linked List**

Major operations are:

- Insertion
 - At the beginning
 - At the middle
 - At the end
- Deletion
 - At the beginning
 - At the middle
 - At the end
- Search

Declaration for Linked List

```

Struct node ;
typedef struct Node *List ;
typedef struct Node *Position ;
int IsLast (List L) ;
int IsEmpty (List L) ;
position Find(int X, List L) ;
void Delete(int X, List L) ;
position FindPrevious(int X, List L) ;
position FindNext(int X, List L) ;
void Insert(int X, List L, Position P) ;
void DeleteList(List L) ;
Struct Node
{
    int element ;
    position Next ;
};

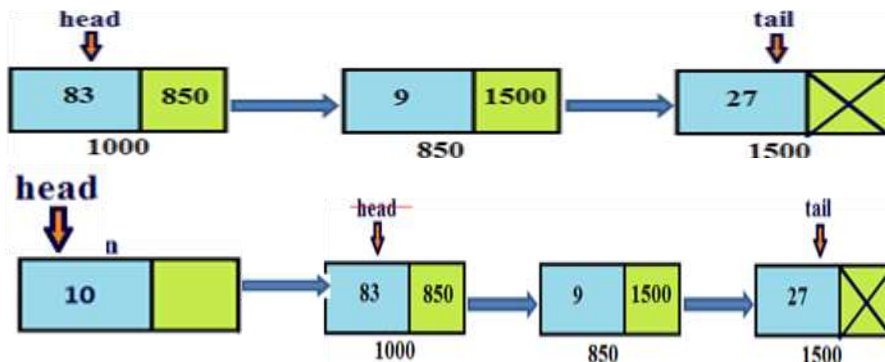
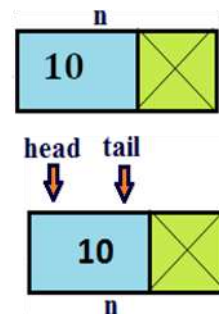
```

Routine for Insertion at the Beginning

```

void ins_beg (int num) //Let num=10
{
    n=(struct node *) malloc (size of(struct node));
    n->next =NULL;
    n-> data=num;
    if (head==NULL)
    {
        head=n;
        tail=n;
    }
    else
    {
        n->next=head;
        head=n;
    }
}

```

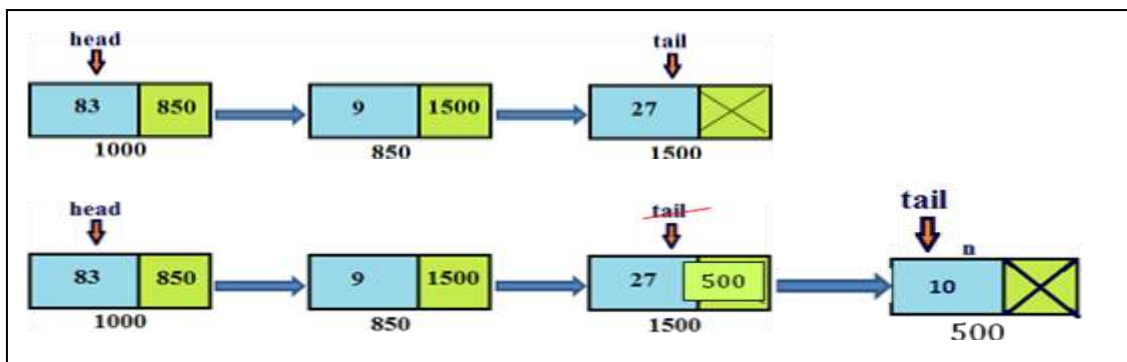
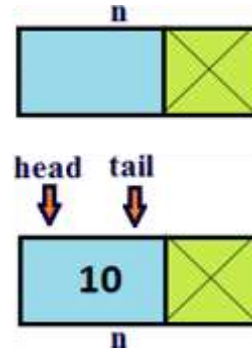


Routine for Insertion at the End

```

void ins_end (int num) //Let num=10
{
    n=(struct node *) malloc (size of(struct node));
    n->next=NULL;
    n->data=num;
    if (head==NULL)
    {
        head=n;
        tail=n;
    }
    else
    {
        tail->next=n
        tail=n;
    }
}

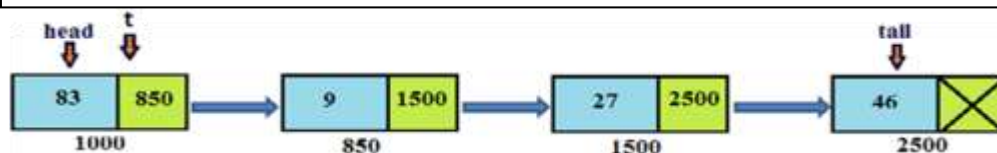
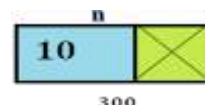
```

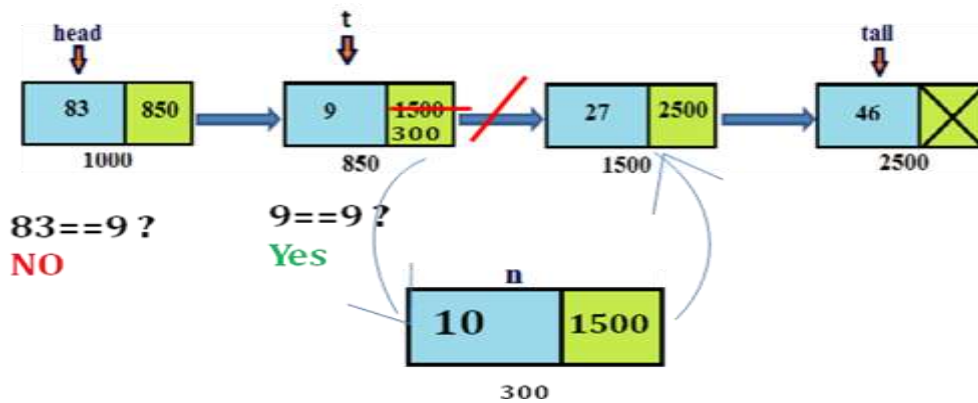
**Routine for Insertion at the Middle**

```

void ins_mid (int num, int mid_data) //Let num=10, mid_data=9
{
    n=(struct node *) malloc (size of(struct node));
    n->next=NULL;
    n->data=num;
    for(t=head; t!=NULL; t=t->next)
    {
        if(t->data==mid_data)
            break;
    }
    n->next=t->next;
    t->next=n;
}

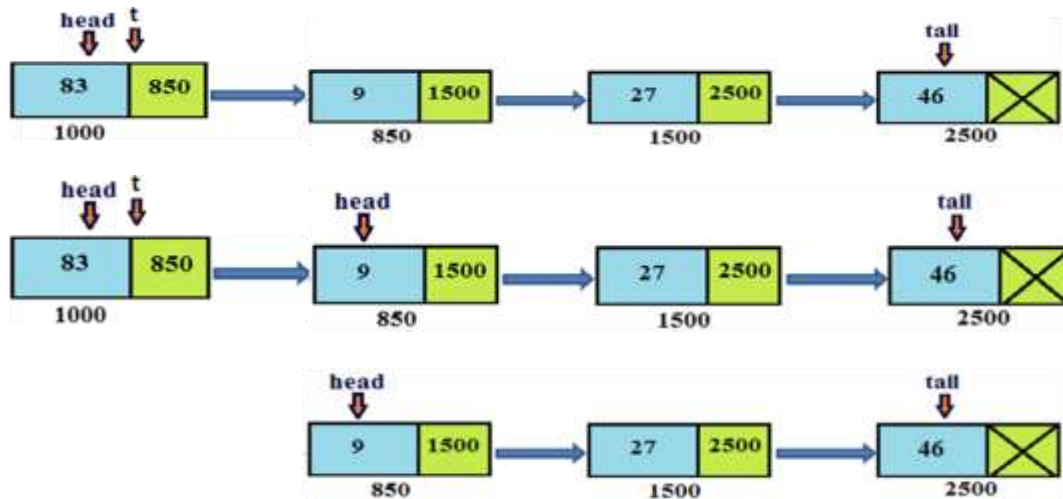
```





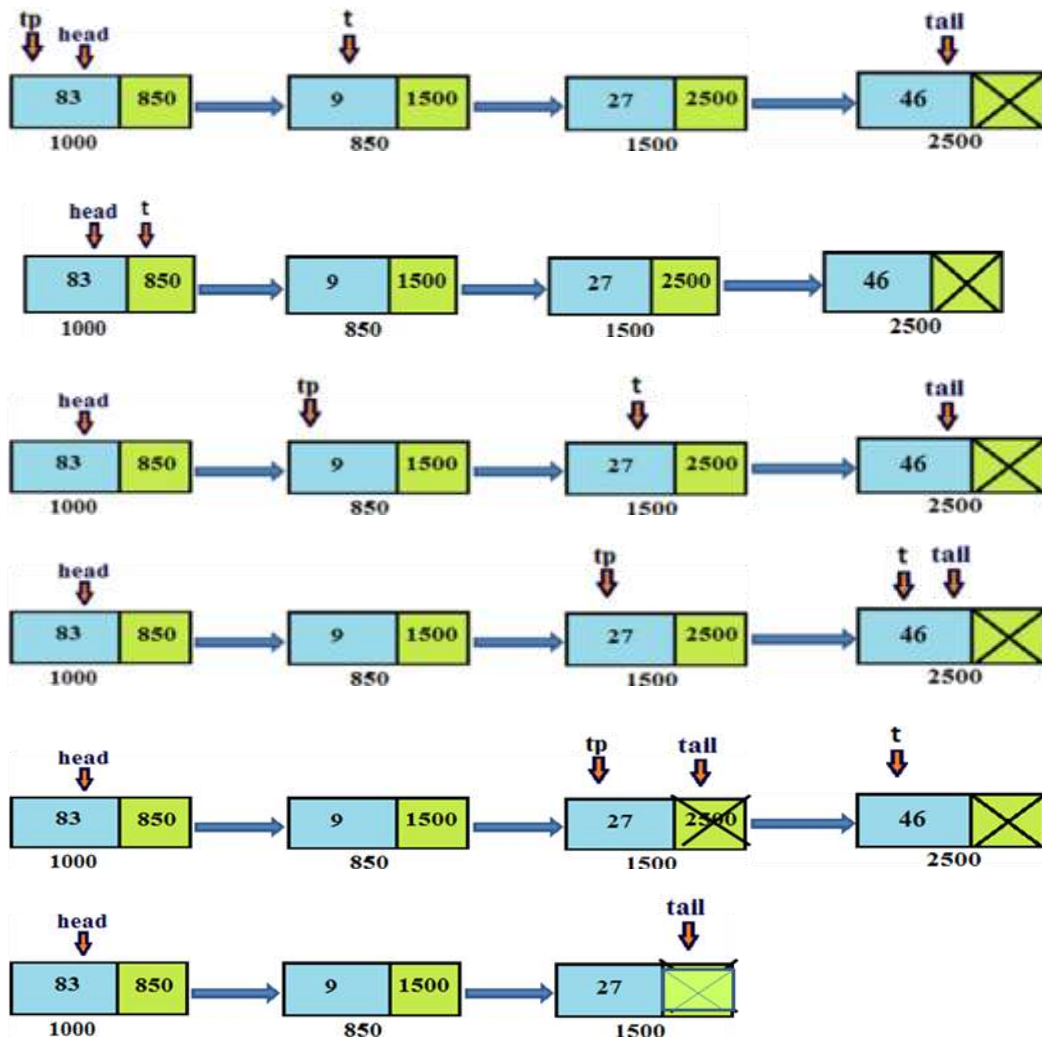
Routine for Deletion at the Beginning

```
void del_beg ()
{
    t=head;
    head=t->next;
    free(t);
}
```



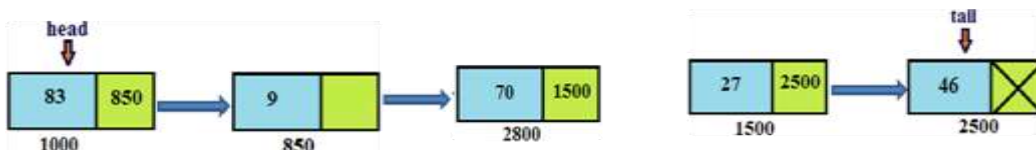
Routine for Deletion at the End

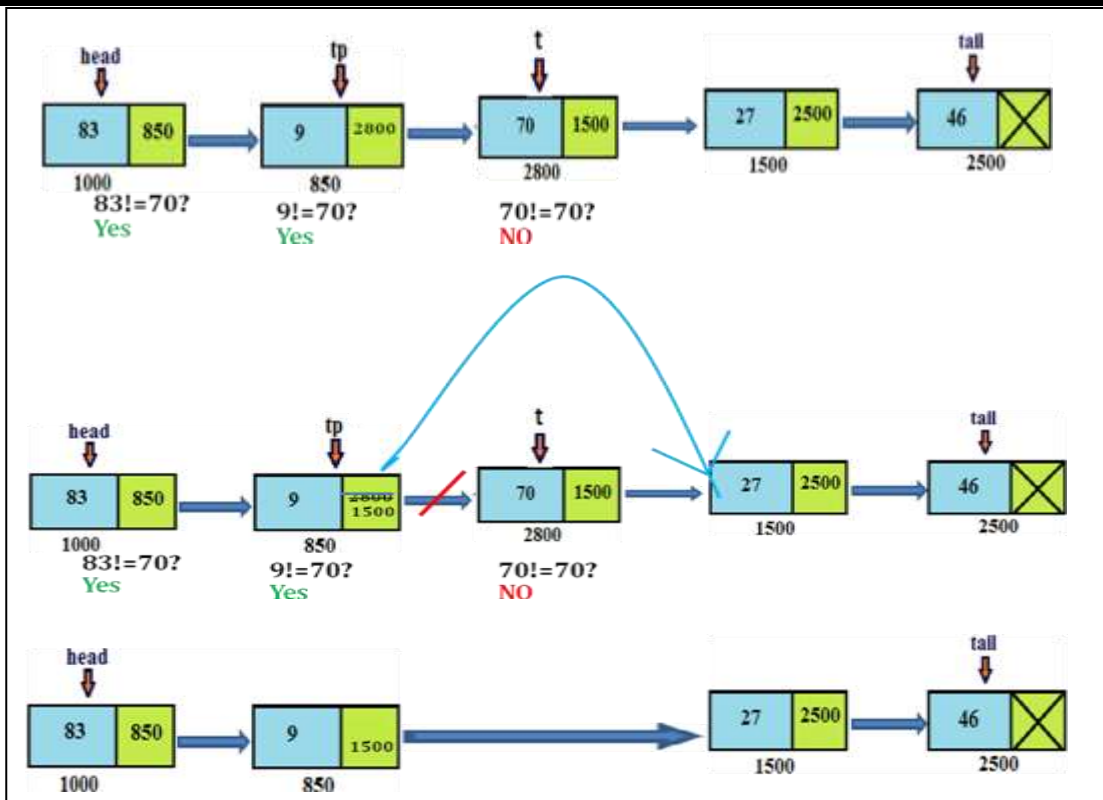
```
void del_end ()
{
    struct node *tp;
    t=head;
    while(t->next!=NULL)
    {
        tp=t;
        t=t->next;
    }
    tail=tp;
    tail->next=NULL;
    free(t);
}
```



Routine for Deletion at the Middle

```
void del_mid (int num) //Let num=70
{
    struct node *tp;
    t=head;
    while(t->data!=num)
    {
        tp=t;
        t=t->next;
    }
    tp->next=t->next;
    free(t);
}
```



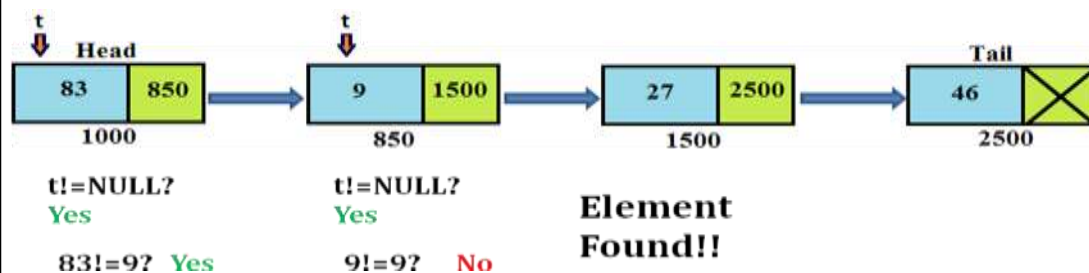


Routine for Search an Element

```

struct node* search (int num) //Let num=9
{
    t=head;
    while(t!=NULL && t->data!=num)
    {
        t=t->next;
    }
    return t;
}

```



Linked List – Advantages & Disadvantages

Advantages:

- No wastage of memory
 - Memory allocated and deallocated as per requirement
- Efficient insertion and deletion operations

Disadvantages:

- Occupies more space than array to store a data
 - Due to the need of a pointer to the next node
- Does not support random or direct access

6.2 DOUBLY LINKED LIST (DLL)

- Each node contains two pointer fields that contains the address of the next node and that of previous node in the list



Representation of Node in DLL

- Each node contains three fields:
 - Data: Contains data element to be stored
 - next: Contains the address of next node
 - prev: Contains address of previous node

Declaration of a node:

```
struct node
{
  int data;
  struct node *next, *prev;
};
```

A Node in DLL



Operations on Doubly Linked List

- Major operations are:
 - Insertion
 - At the beginning
 - At the middle
 - At the end
 - Deletion
 - At the beginning
 - At the middle
 - At the end
 - Search

Global declaration of a Node:

```
struct node
{
  int data;
  struct node *next, *prev;
} *head, *tail, *n, *t;
```

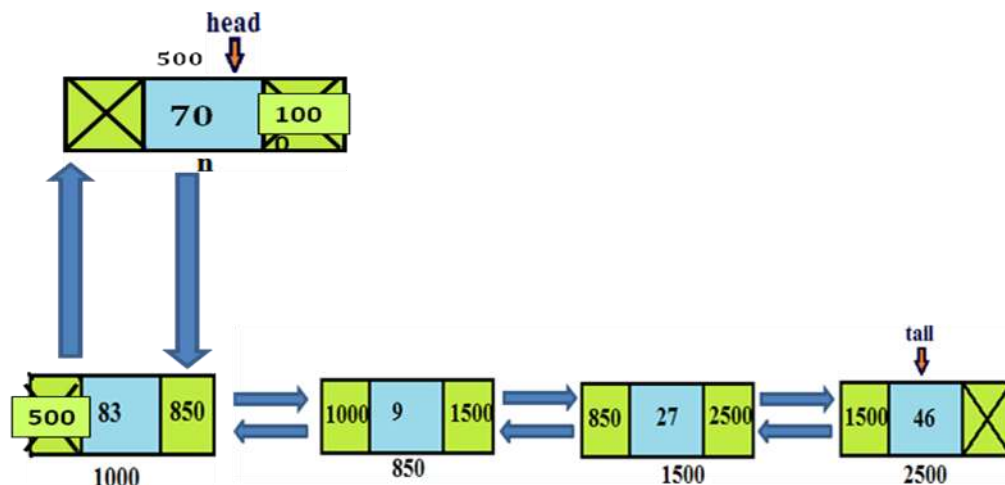
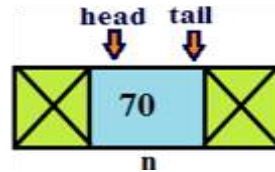
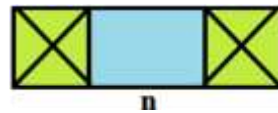


Routine for Insertion at the Beginning

```

void ins_beg_dll (int num) //Let num=70
{
n=(struct node *) malloc(size of(struct node));
n->next=NULL;
n->prev=NULL;
n->data=num;
if(head==NULL)
{
head=n; tail=n;
}
else
{
n->next=head;
head->prev=n;
head=n;
}
}

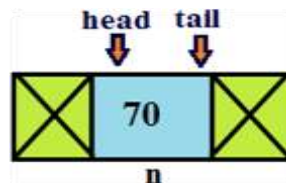
```

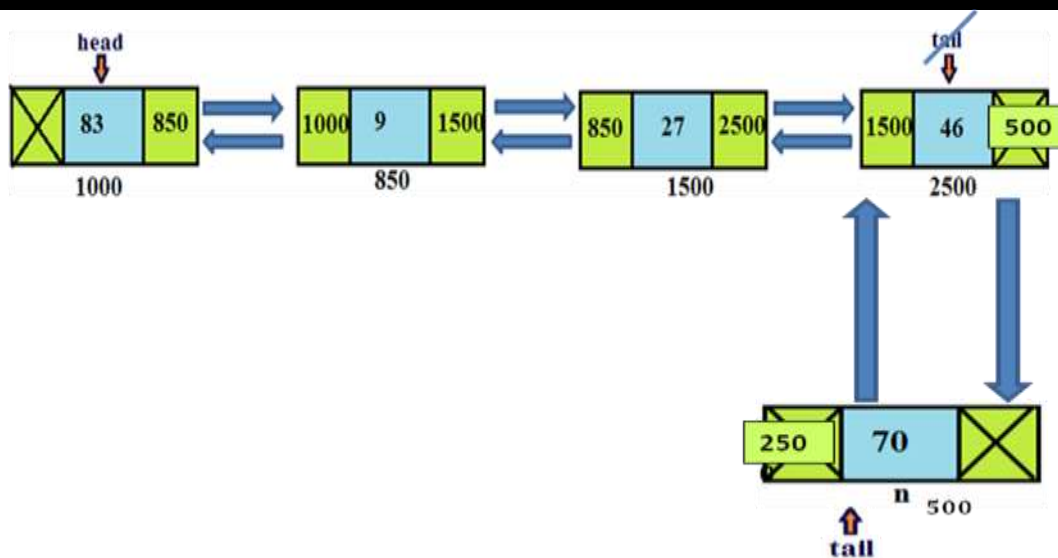
**Routine for Insertion at the End**

```

void ins_end_dll (int num) //Let num=70
{
n=(struct node *) malloc (size of(struct node));
n->next=NULL; n->prev=NULL; n->data=num;
if (head==NULL)
{
head=n; tail=n;
}
else
{
tail->next=n; n->prev=tail;
tail=n;
}
}

```

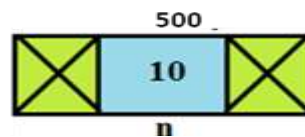




Routine for Insertion at the Middle

void ins_mid_dll (int num, int mid_data) //Let num=10, mid_data=9

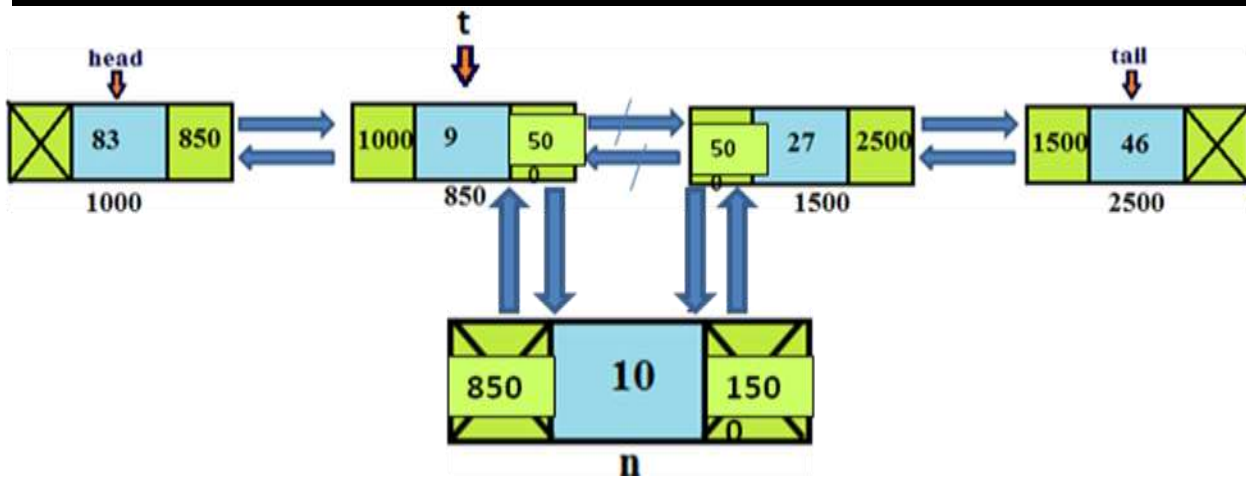
```
{
n=(struct node *) malloc (size of(struct node));
n->next=NULL;
n->prev=NULL;
n->data=num;
for(t=head; t!=NULL; t=t->next)
{
if(t->data==mid_data)
break;
}
n->next=t->next;
n->prev=t;
t->next->prev=n;
t->next=n;
}
```



83==9, No

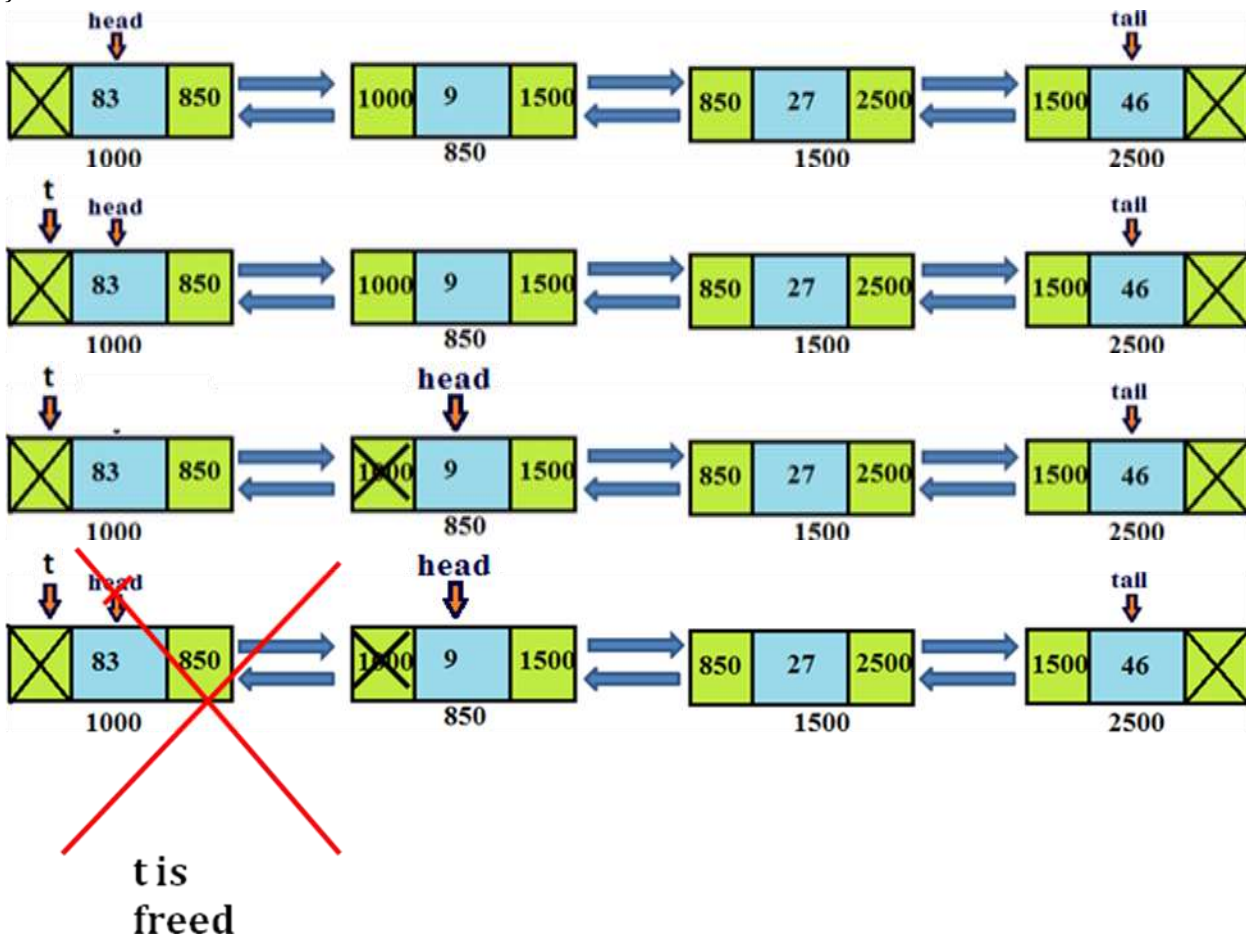


9==9, Yes



Routine for Deletion at the Beginning

```
void del_beg_dll ()
{
    t=head;
    head=head->next; head->prev=NULL;
    free(t);
}
```

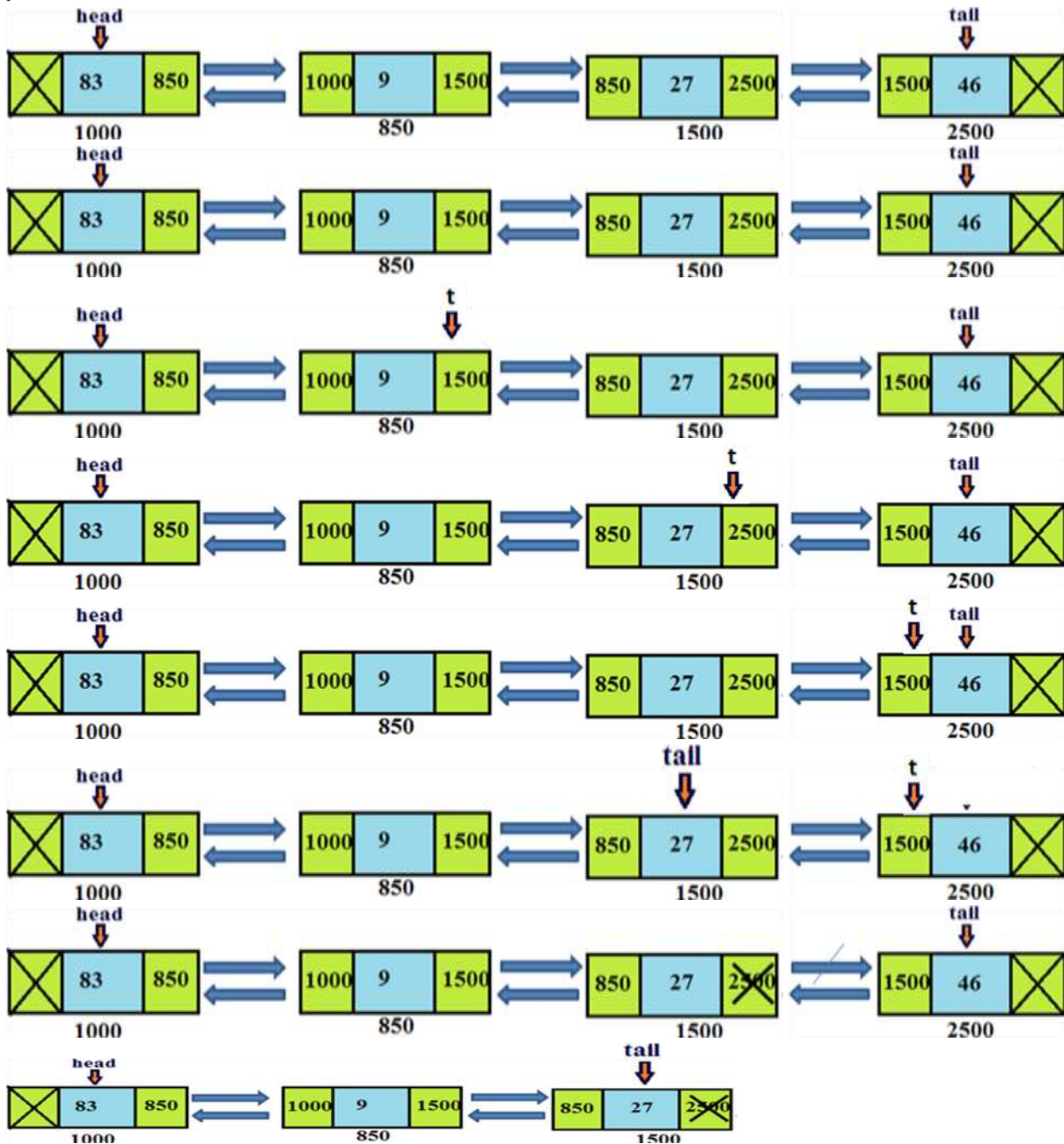


Routine for Deletion at the End

```

void del_end_dll ()
{
    t=head;
    while(t->next!=NULL)
    {
        t=t->next;
    }
    tail=tail->prev;//tail=t->prev;
    tail->next=NULL;
    free(t);
}

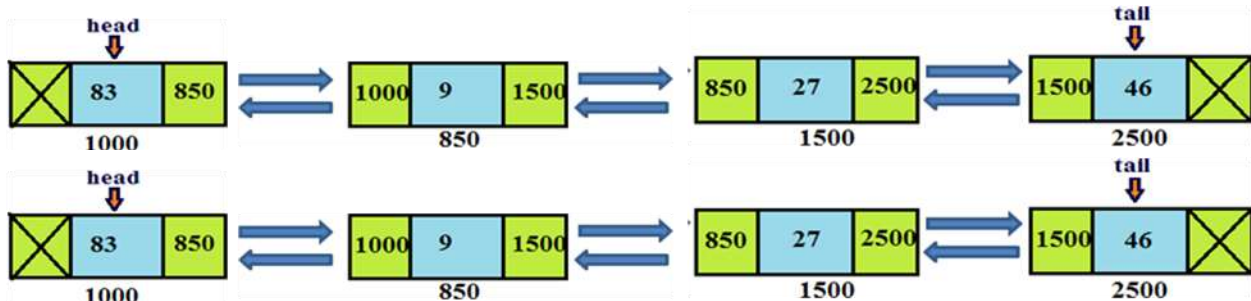
```



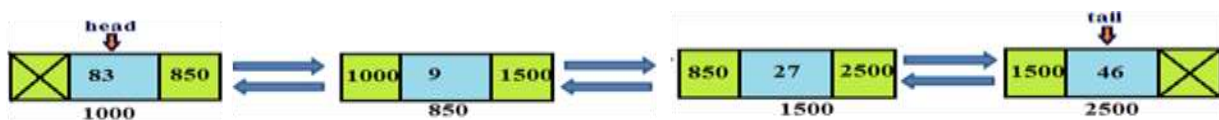
Routine for Deletion at the Middle

```
void del_mid_dll (int num) //Let num=9
```

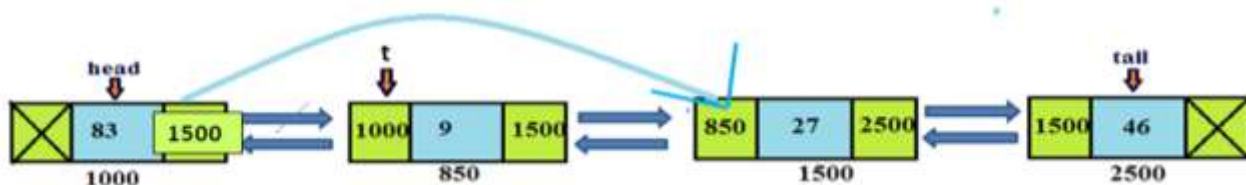
```
{
t=head;
while(t->data!=num)
{
t=t->next;
}
t->prev->next=t->next;
t->next->prev=t->prev;
free(t);
}
```



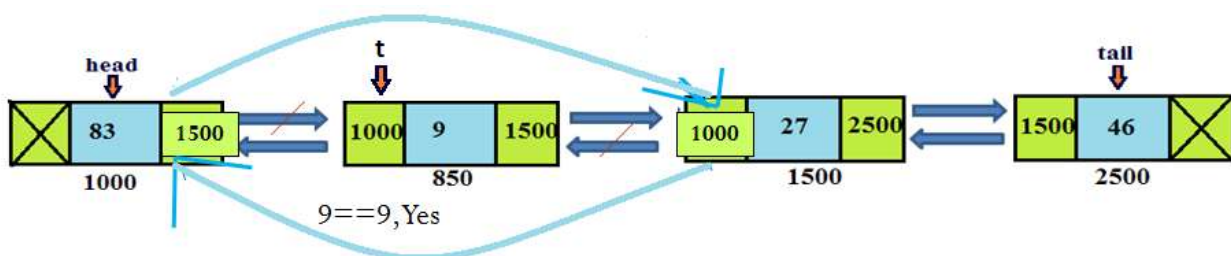
83!=9, Yes



9!=9, No



9==9, Yes



9==9, Yes

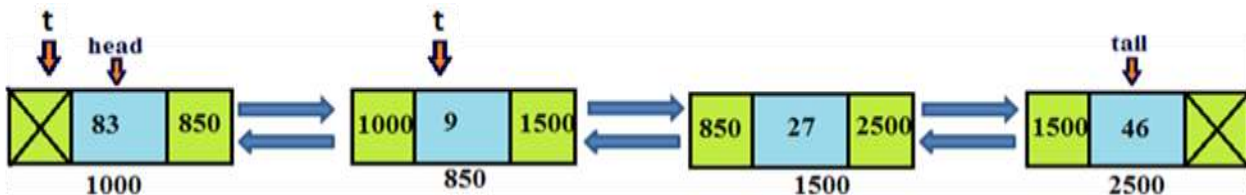


Routine for Search an Element

```

struct node* search_dll (int num) //Let num=9
{
t=head;
while(t!=NULL && t->data!=num)
{
t=t->next;
}
return t;
}

```



**Element
Found!! t is
returned**

DLL – Advantages & Disadvantages**Advantages:**

- Can traverse in both directions
- Operations in the middle of the list can be done efficiently
 - No need of extra pointer (tp) to track the previous node
- Reversing the list is easy

Disadvantage:

- Space required to store a data is even more higher than SLL
 - Due to the use of two pointers

6.3 CIRCULAR LINKED LIST (CLL)

- In circular linked list the pointer of the last node points to the first node.
- Circular linked list can be implemented as Singly linked list and Doubly linked list with or without headers.



Global declaration of a node:

```

struct node
{
int data;
struct node *next, *prev;
}*head,*n,*t;

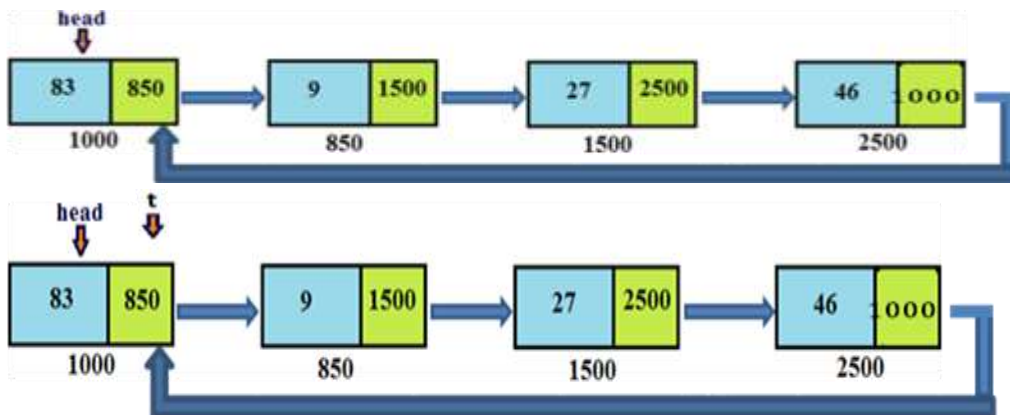
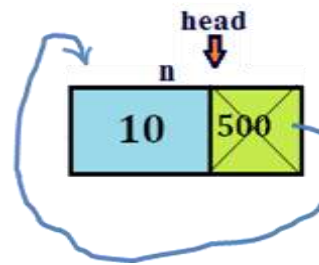
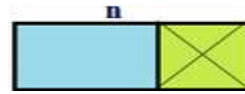
```

Routine for Insertion at the Beginning

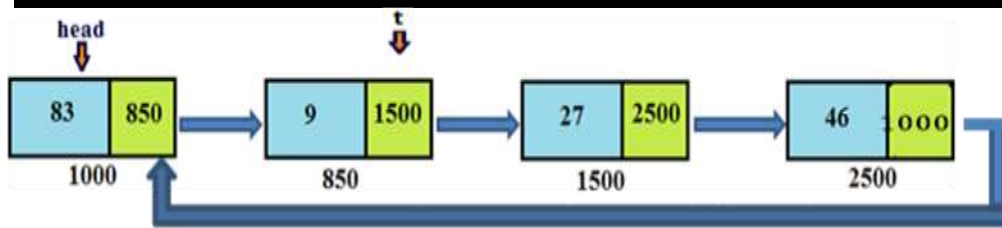
```

void ins_beg_cll (int num) //Let num=10
{
n=(struct node *) malloc (sizeof(struct node));
n->next=NULL;
n->data=num;
if (head==NULL) //case 1
{
    head=n;
    n->next=head;
}
else //case 2
{
    t=head;
    While(t->next!=head)
        t=t->next;
    t->next=n;
    n->next=head;
    head=n;
}
}

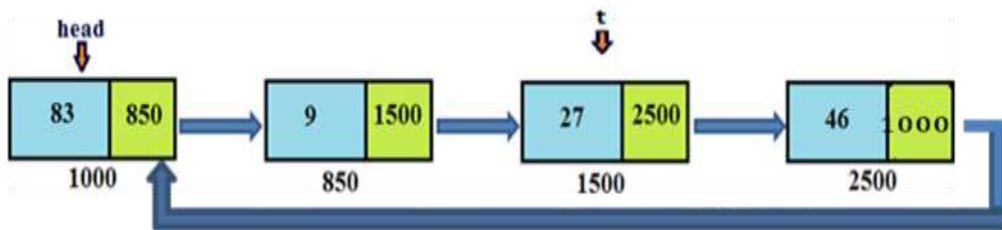
```



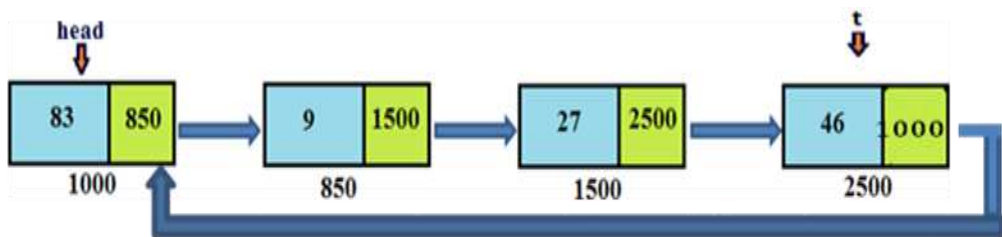
850 != 1000,
Yes



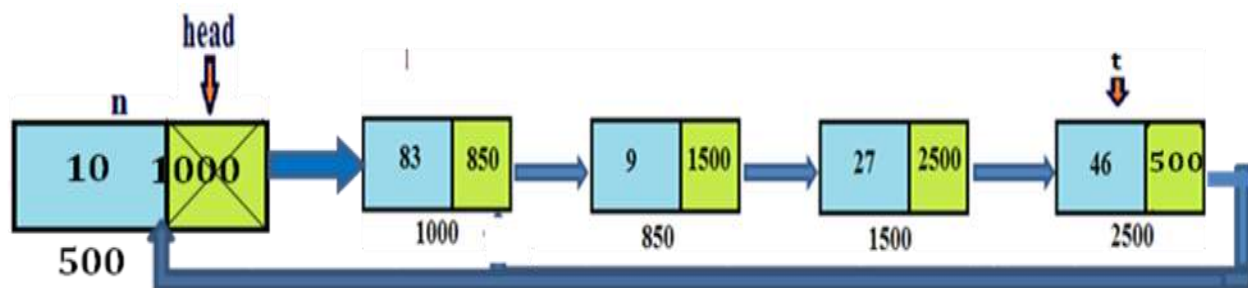
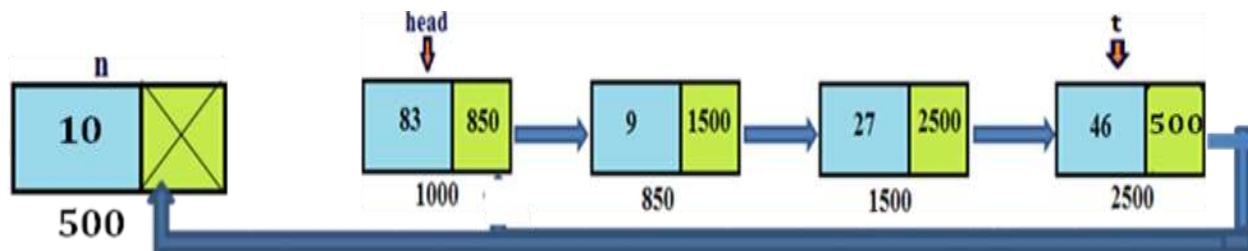
$1500 \neq 1000$,
Yes



$2500 \neq 1000$,
Yes



$1000 \neq 1000$,
No

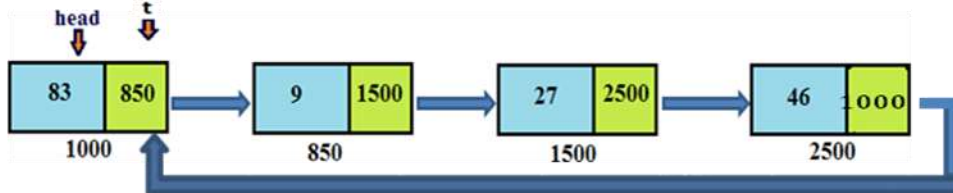
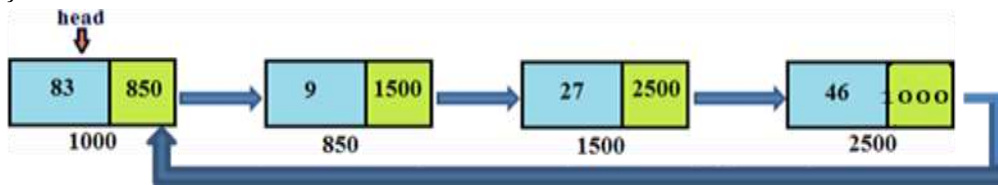


Routine for Insertion at the End

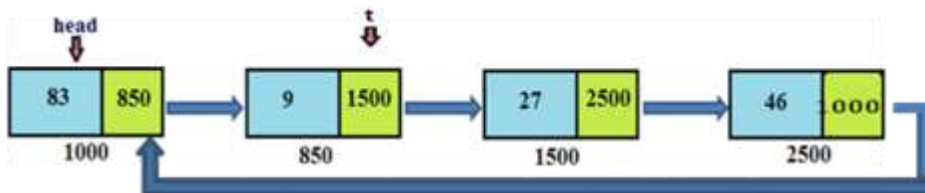
```

void ins_end_cll (int num) //Let num=10
{
n=(struct node *) malloc (size of(struct node));
n->next=NULL;
n->data=num;
t=head;
while(t->next!=head)
    t=t->next;
t->next=n;
n->next=head;
}

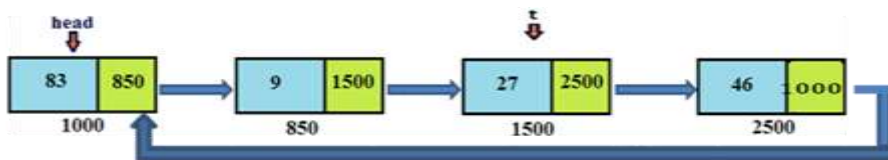
```



850 != 1000,
Yes



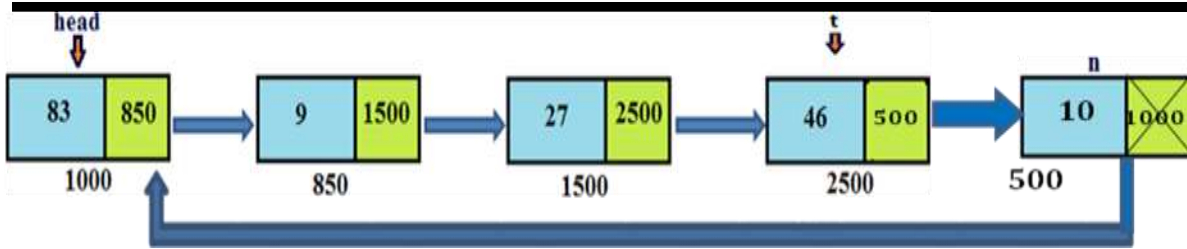
1500 != 1000,
Yes



2500 != 1000,
Yes



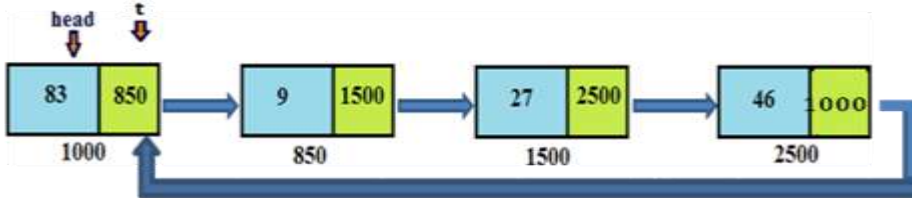
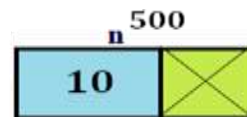
1000 != 1000,
No



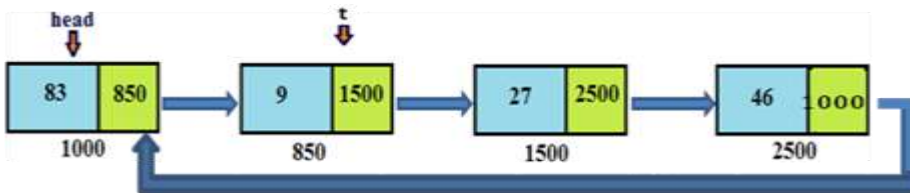
Routine for Insertion at the Middle

void ins_end_cll (int num, int mid_data) //Let num=10, mid_data=9

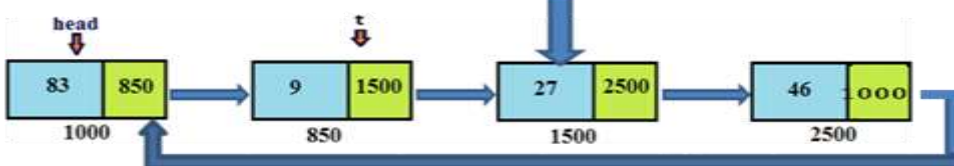
```
{
n=(struct node *) malloc (size of(struct node));
n->next=NULL;
n->data=num;
for(t=head; t->next!=head; t=t->next)
{
if(t->data==mid_data)
break;
}
n->next=t->next;
t->next=n;
}
```



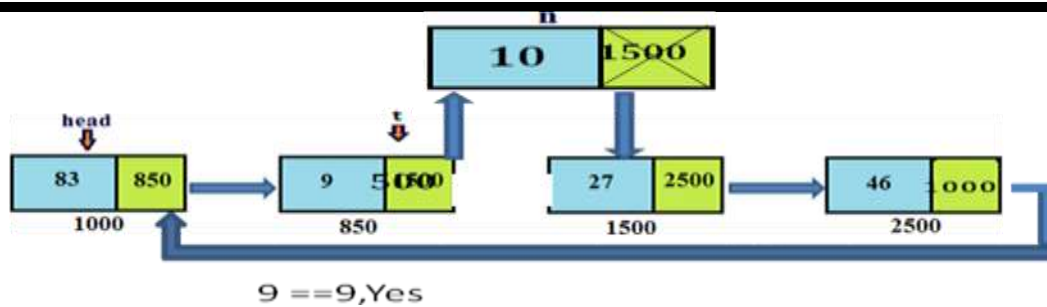
83 == 9, No



9 == 9, Yes



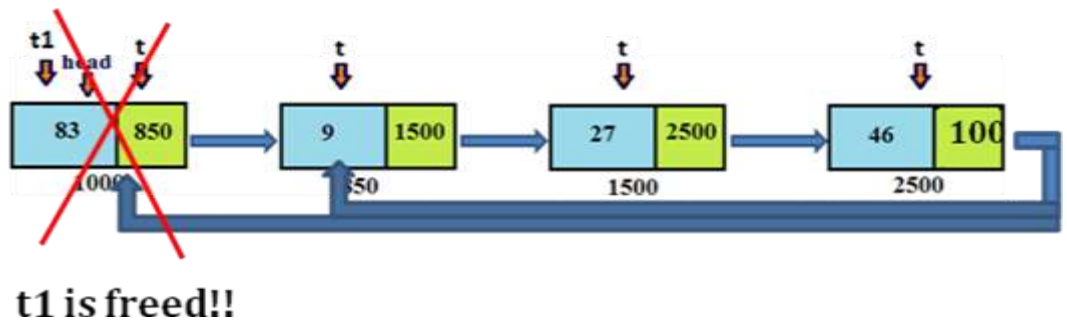
9 == 9, Yes



Routine for Deletion at the Beginning

```
void del_beg_cll ()
```

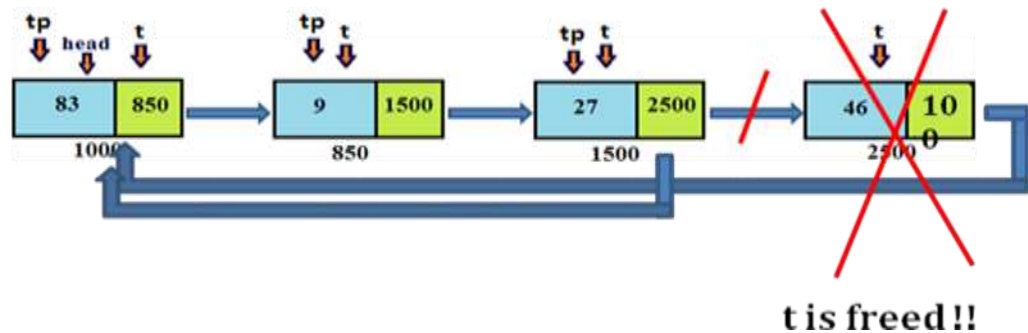
```
{
struct node *t1;
t=head;
t1=head;
while(t->next!=head)
    t=t->next;
t->next=head->next;
head=t->next;
free(t1);
}
```



Routine for Deletion at the End

```
void del_end_cll ()
```

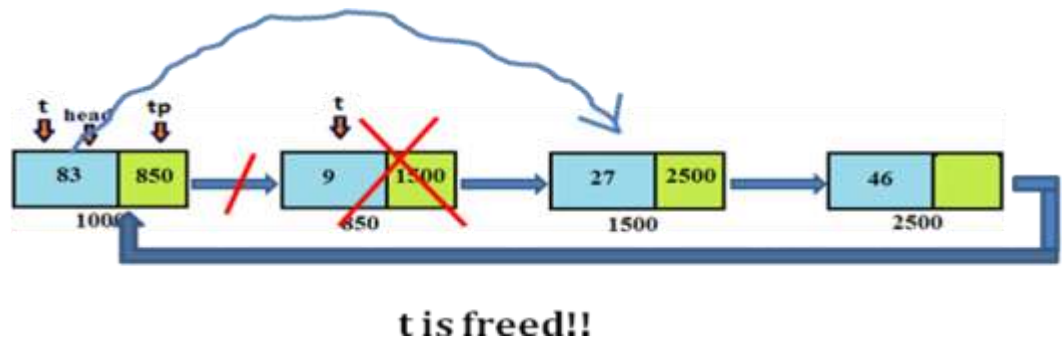
```
{
struct node *tp;
t=head;
while(t->next!=head)
{
tp=t;
t=t->next;
}
tp->next=head;
free(t);
}
```



Routine for Deletion at the Middle

```
void del_mid_cll (int num) //Let num=9
```

```
{
struct node *tp;
t=head;
while(t->data!=num)
{
tp=t;
t=t->next;
}
tp->next=t->next;
free(t);
}
```

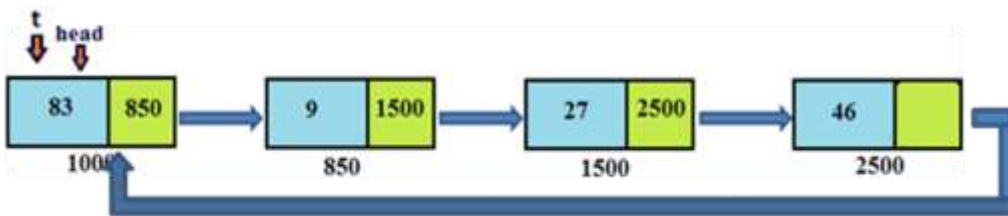


Routine for Search

```

struct node* search_cll (int num) //Let num=9
{
t=head;
while(t->next!=head && t->data!=num)
{
t=t->next;
}
return t;
}

```

**CLL – Advantages & Disadvantages****Advantage:**

- Comparing to SLL, moving to any node from a node is possible

Disadvantages:

- Reversing a list is complex compared to linear linked list
- If proper care is not taken in managing the pointers, it leads to infinite loop
- Moving to previous node is difficult, as it is needed to complete an entire circle

6.6 APPLICATIONS OF LIST

Lists can be used to

- Sort elements
- Implement stack and queue
- Represent graph (adjacent list representation)
- Implement hash table
- Implement multiprocessing of applications
- Manipulate polynomial equations

Polynomial ADT

- Examples of polynomial equations:
 - $-9x^5 + 7x^3 - 4x^2 + 8$
 - $7x^4 - 17x^3 - 8x^2$
- To store the data of polynomial equations in the linked list, each node contains two data fields, namely coefficient and power and one next pointer field

```

struct node

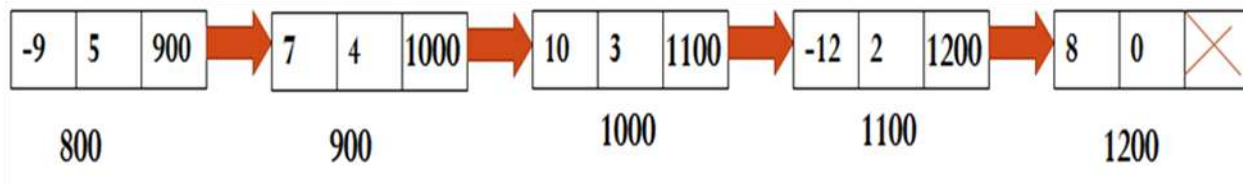
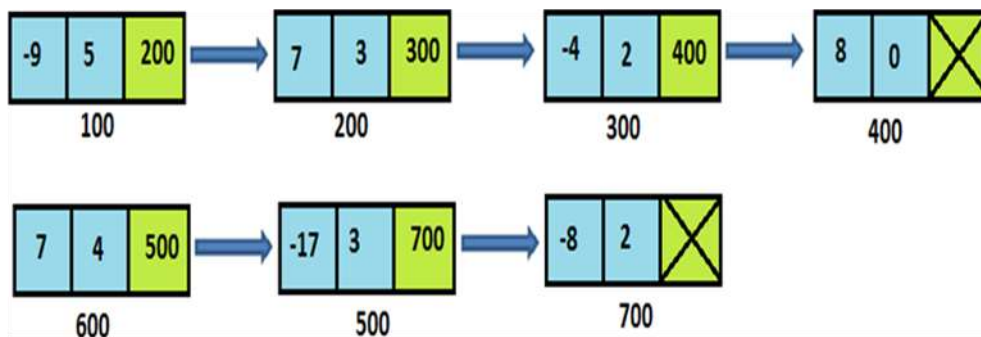
```

```

{
int coeff;
int pow;
struct node *next;
}*n;

```





Creating a Polynomial List

```

struct node* createpoly (int c, int p, struct node *t)
{
    struct node *head, *tail;
    head=t;
    n=(struct node*) malloc(size of(struct node));
    n->coeff=c;
    n->pow=p;
    n->next=NULL;
    if (head==NULL)
    {
        head=n;
        tail=n;
    }
    else
    {
        tail->next=n;
        tail=n;
    }
    return head;
}

```

Polynomial Addition

```

Void addpoly (struct node *poly1, struct node *poly2, struct node *poly)
{
    while(poly1!=NULL && poly2!=NULL)
    {
        if(poly1->pow>poly2->pow)
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff;
            poly1=poly1->next;
        }
    }
}

```

```

else if(poly1→pow < poly2→pow)
{
    poly→pow=poly2→pow;
    poly→coeff=poly2→coeff;
    poly2=poly2→next;
}
else
{
    poly→pow=poly1→pow;
    poly→coeff=poly1→coeff + poly2→coeff;
    poly1=poly1→next;
    poly2=poly2→next;
}
}
while(poly1 != NULL || poly2 != NULL)
{
    if (poly1 != NULL)
    {
        poly→pow=poly1→pow;
        poly→coeff=poly1→coeff;
        poly1=poly1→next;
    }
    if (poly2 != NULL)
    {
        poly→pow=poly2→pow;
        poly→coeff=poly2→coeff;
        poly2=poly2→next;
    }
}

```

RADIX SORT:

- Radix Sort is performed using Buckets from 0 to 9. The digit position of a number is used to perform sorting. The number of passes in a Radix sort depends on number of digits in the number given.
- Example: Consider the numbers to be sorted are 42,24,11,36,32,54,10,19,27,45.
- The number of digits in the numbers is 2. Therefore the number of passes is 2.

PASS 1: 42,24,11,36,32,54,10,19,27,45

Consider the first number, 42 its unit digit is 2. So place the 42 in the Bucket 2.

- Next number is 24; its unit digit is 4. So place the 24 in the Bucket 4.
- 11 is stored in Bucket 1,
- 36 in Bucket 6,
- 32 in Bucket 2,
- 54 in Bucket 4,
- 10 in Bucket 0,
- 19 in Bucket 9,
- 27 in Bucket 7,
- 45 in Bucket 5.

10	11	32		54					
0	1	2	3	4	5	6	7	8	9

Collect back all the numbers from the Buckets 0 to 9 as 10, 11, 42, 32, 24, 54, 45, 36, 27 and 19.

PASS 2: 10, 11, 42, 32, 24, 54, 45, 36, 27 and 19.

- Consider the first number which is obtained from PASS 1 is 10; its tenth digit is 1 so place the number 10 in Bucket 0.
- Next number is 11; its tenth digit is 1, so place the number 11 in Bucket 1. Likewise, 42 is stored in Bucket 4,
- 32 is stored in Bucket 3,
- 24 is stored in Bucket 2,
- 54 is stored in Bucket 5,
- 45 is stored in Bucket 4,
- 36 is stored in Bucket 3,
- 27 is stored in Bucket 2 and 19 is stored in Bucket 1.

	19								
	11	27	36	45					
	10	24	32	42	54				
0	1	2	3	4	5	6	7	8	9

Collect all the numbers from the Bucket 0 to 9 as 10, 11, 19, 24, 27, 32, 36, 42, 45 and 54.

Applications of linked list in real world-

- Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.
- Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- Music Player* – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

PART-A

1. Define Data Structure.

Data Structure is a way of representing data that contains both the data item & their relationship with each other.

2. List the types of Data Structure. (or) Outline a linear and a non-linear data structure with an example. (Nov / Dec 2022)

- Linear Data Structures: All the elements are arranged in a sequence or maintained in a linear ordering. Example: Arrays, Stacks, Queues and Linked Lists.
- Non- Linear Data Structures: No such sequence is maintained for the elements and the elements are distributed over a plane. Example: Trees, Graphs, Tables and Sets.

3. State the advantages of modularity in programming. (Apr / May 2024)

- Improves readability and debugging
- Enhances code reuse
- Supports parallel development
- Easier to maintain and test

4. Define ADT. (Nov / Dec 2023) (or) What are ADT? Give an example. (Apr / May 2024)

An Abstract Data Type (ADT) is a keyword used in a programming language to specify the amount of memory needed to store the data and the type of data that will be stored in that memory location.

5. State the uses of Abstract Data Type (ADT). (Apr / May 2023)

- Provides data abstraction and encapsulation
- Helps in modular programming
- Enhances code reusability and maintenance
- Separates implementation from usage

6. What is an array?

An array is collection of variables of the same data type that share a common name.

7. Identify how multidimensional array can be represented in memory. (Nov / Dec 2024)

- Row-major order: Rows stored one after the other.
- Column-major order: Columns stored sequentially.
- Memory location is calculated using formulas depending on dimensions and order.

8. List the disadvantages of an Array.

- Fixed size of memory allocation
- Slow speed in searching, insertion and deletion

9. List the advantages and disadvantages of a Linked List.

Advantages:

- Efficient utilization of memory
- Easy and efficient insertion and deletion of the data items

Disadvantages:

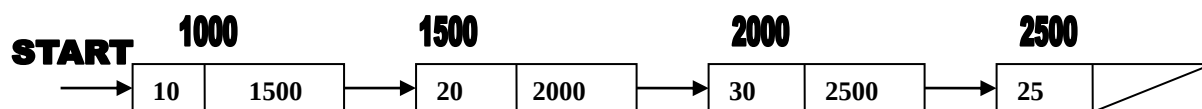
- Increased time of search
- Additional overhead on space
- Additional overhead on the management of memory
- Difficult to implement

10. Define Linked List.

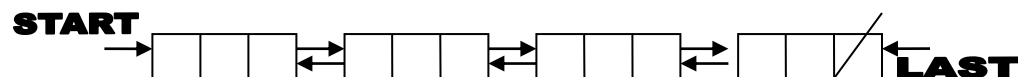
A linked list is a chain of elements in which each element consists of data as well as link to the next element.

11. Define Singly Linked List.

It is the simplest type of linked list, where each node points to the next node. The last node does not point to any other node in the list. Therefore, it points to NULL. This means that a node pointing to NULL signals the end of the list.

**12. Define Doubly Linked List.**

In doubly linked list, each node contains a reference or address of the next node as well as its previous node. Therefore, it is possible to traverse in the reverse direction also, which is not possible in singly linked list.



13. List the advantages of doubly linked list. (Apr / May 2023)

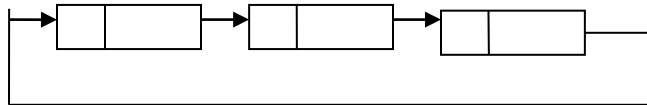
- Can traverse in both directions
- Insertion/deletion is easier at both ends
- Backtracking is possible
- More flexible than singly linked list

14. Infer the usage of doubly linked list. (Nov / Dec 2023)

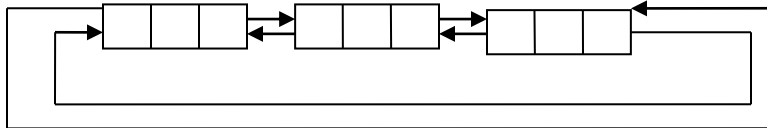
Useful in navigation systems, undo-redo functionality, managing playlists, browser history, etc., where bidirectional traversal is needed.

15. Define Circular Singly Linked List. (or) Outline a circular linked list with a diagram. (Nov / Dec 2022)

It is similar to a singly linked list where each node points to the next node in sequence. Also the last node points to the first node instead of pointing to NULL.

**16. Define Circular Doubly Linked List. (or) Outline a circular linked list with a diagram. (Nov / Dec 2022)**

It is similar to a doubly linked list where each node points to the next node in sequence. Also the last node points to the first node instead of pointing to NULL.

**17. List the Applications of Linked List.**

- Gaming Applications
- File system in Operating System
- Polynomial manipulation

18. List the basic operations performed on a Linked Lists.

Insertion, Deletion, Traversal, Search and Merge

19. Define Polynomial ADT. (Nov / Dec 2024)

- Polynomial ADT represents polynomials using arrays or linked lists.
- Each term has a coefficient and exponent.
- Supports operations like addition, evaluation, and multiplication.

20. Give the difference between array and linked list.

Arrays	Linked Lists
An array is a collection of elements of a similar data type .	Linked List is an ordered collection of elements of the same type in which each element is connected to the next using pointers.
Array elements can be accessed randomly using the array index.	Random accessing is not possible in linked lists. The elements will have to be accessed sequentially .
Data elements are stored in contiguous locations in memory.	New elements can be stored anywhere and a reference is created for the new element using pointers.

Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	Insertion and Deletion operations are fast and easy in a linked list.
Memory is allocated during the compile time (Static memory allocation).	Memory is allocated during the run-time (Dynamic memory allocation).
Size of the array must be specified at the time of array declaration/initialization.	Size of a Linked list grows/shrinks as and when new elements are inserted/deleted.

21. Distinguish between singly, doubly and circular linked list with an example.

Feature	Singly Linked List	Doubly Linked List	Circular Linked List
Structure	Each node contains data and a pointer to the next node.	Each node contains data, a pointer to the next node, and a pointer to the previous node.	Can be singly or doubly linked, but the last node points back to the first node.
Traversal / Direction	Only in one direction (from head to end).	In both directions.	Can continue indefinitely.
Last node	Points to NULL.	Points to NULL in next; first node's prev is also NULL.	No node points to NULL.
Node Links	One (next)	Two (prev, next)	One or two (depends on type)
Memory Usage	Less (one pointer/node)	More (two pointers/node)	Varies (similar to singly/doubly)

PART- B

ABSTRACT DATA TYPES (ADTS) – LIST ADT – ARRAY-BASED IMPLEMENTATION

1. Explain how list operations can be implemented using arrays. (Apr / May 2023)
2. What are the different ways the list can be implemented? State and explain list ADT. (6) (Apr / May 2024)
3. Explain the operation of traversing linked list. Write the algorithm and give an example. (Nov / Dec 2024)

SINGLY LINKED LISTS

1. What is a singly linked list? Write an algorithm for inserting an element from singly linked list with illustration.
2. What is a singly linked list? Write an algorithm for deleting an element from singly linked list with illustration.
3. Write a C Program to sort and reverse a linked list.
4. Outline the steps to search a linked list with an example and relevant diagrams. (7) (Nov / Dec 2022)
5. Outline the steps to delete from a linked list with an example and relevant diagrams. (6) (Nov / Dec 2022)
6. Infer how lists and linked list are stored in memory with an example and write a function to check current position is last in a linked list. (7) (Nov / Dec 2023)
7. Elaborate the steps to implement the operations of singly linked list: Traverse, Insert at front, Insert at any and Insert at end. (Nov / Dec 2024)

DOUBLY-LINKED LISTS

1. What is a doubly linked list? Write an algorithm for inserting an element from doubly linked list with illustration.
2. What is a doubly linked list? Write an algorithm for inserting an element from doubly linked list with illustration.
3. Write the functions for the following operations on doubly linked list. (Apr / May 2024)
 - Sum up the values stored in the nodes of a list. (5)
 - Count the even numbers in the list. (5)
 - Delete the node with an element x. (5)

CIRCULAR LINKED LISTS

1. Distinguish circular linked list and doubly linked list during insertion and deletion operation. (6) (Nov / Dec 2023)
2. Distinguish between singly, doubly and circular linked list with an example. (6) (Apr / May 2024)
3. Write a C function to insert a node in the middle of the linked list and count the number of nodes in the circular linked list. (7) (Apr / May 2024)

APPLICATIONS OF LISTS – POLYNOMIAL ADT

1. Explain in detail with illustration to add the given two polynomials:
Polynomial 1: $4x^5 + 5x^4 + 2x^3 + 3x^2 + 7x$ Polynomial 2: $9x^6 + 6x^4 + 3x^2$
2. Explain in detail the Applications of Linked List.
3. How can a polynomial be represented as a linked list? Outline the algorithm for addition of two polynomials using linked lists with an example. (13) (Nov / Dec 2022)
4. Explain how polynomial expression can be represented using linked list. (Apr / May 2023)
5. Express an algorithm to add two polynomials having m and n terms respectively using linked list with an example. (7) (Nov / Dec 2023)
6. Write the function to add two polynomial given as a linked list. Input: P1: $13x^6 + 7x^5 + 32x^2 + 54$ P2: $3x^{12} + 17x^5 + 3x^3 + 98$ (7) (Apr / May 2024)
7. Explain how linked list are used for sorting the following set of numbers using radix sort.
 $A = \{432, 8, 530, 90, 88, 231, 1, 45, 677, 199\}$ (6) (Nov / Dec 2023)

UNIT II -- STACKS & QUEUES

Stack ADT – Operations – Applications – Balancing Symbols – Evaluating arithmetic expressions- Infix to Postfix conversion – Function Calls – Queue ADT – Operations – Circular Queue – DeQueue – Applications of Queues.

Defining a Stack

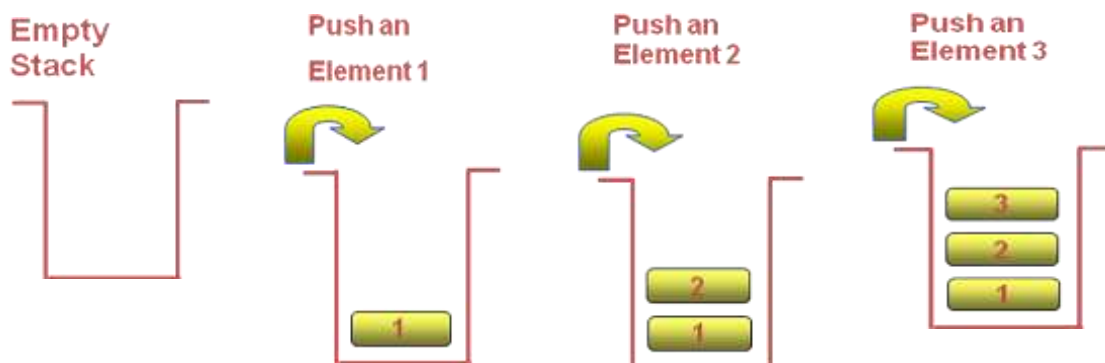
- ◆ **A stack is a collection of data items that can be accessed at only one end, called top.**
- ◆ Items can be inserted and deleted in a stack only at the top.
- ◆ The last item inserted in a stack is the first one to be deleted.
- ◆ Therefore, a stack is called a **Last-In-First-Out (LIFO)** data structure.
- ◆ There are two basic operations that are performed on stacks:

- ◆ PUSH
- ◆ POP

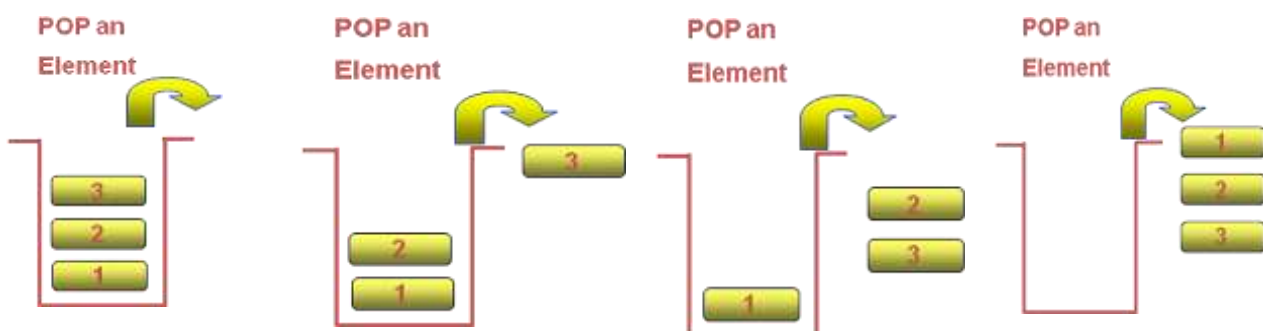
◆ Example

- ◆ A stack is like an empty box containing books, which is just wide enough to hold the books in one pile.
- ◆ The books can be placed as well as removed only from the top of the box.
- ◆ The book most recently put in the box is the first one to be taken out.
- ◆ The book at the bottom is the first one to be put inside the box and the last one to be taken out.

- ◆ **PUSH:** It is the process of inserting a new element on the top of a stack.



- ◆ **POP:** It is the process of deleting an element from the top of a stack.



- ◆ List down some real life examples that work on the LIFO principle.
 - ✓ **Pile of books:** Suppose a set of books are placed one over the other in a pile. When you remove books from the pile, the topmost book will be removed first. Similarly, when you have to add a book to the pile, the book will be placed at the top of the pile.

- ✓ **Pile of plates:** The first plate begins the pile. The second plate is placed on the top of the first plate and the third plate is placed on the top of the second plate, and so on. In general, if you want to add a plate to the pile, you can keep it on the top of the pile. Similarly, if you want to remove a plate, you can remove the plate from the top of the pile.
- ✓ **Bangles in a hand:** When a person wears bangles, the last bangle worn is the first one to be removed.

Implementing a Stack

- ◆ A stack is similar to a list in which insertion and deletion is allowed only at one end.
- ◆ Therefore, similar to a list, stack can be implemented using both arrays and linked lists.

Implementing a Stack Using an Array

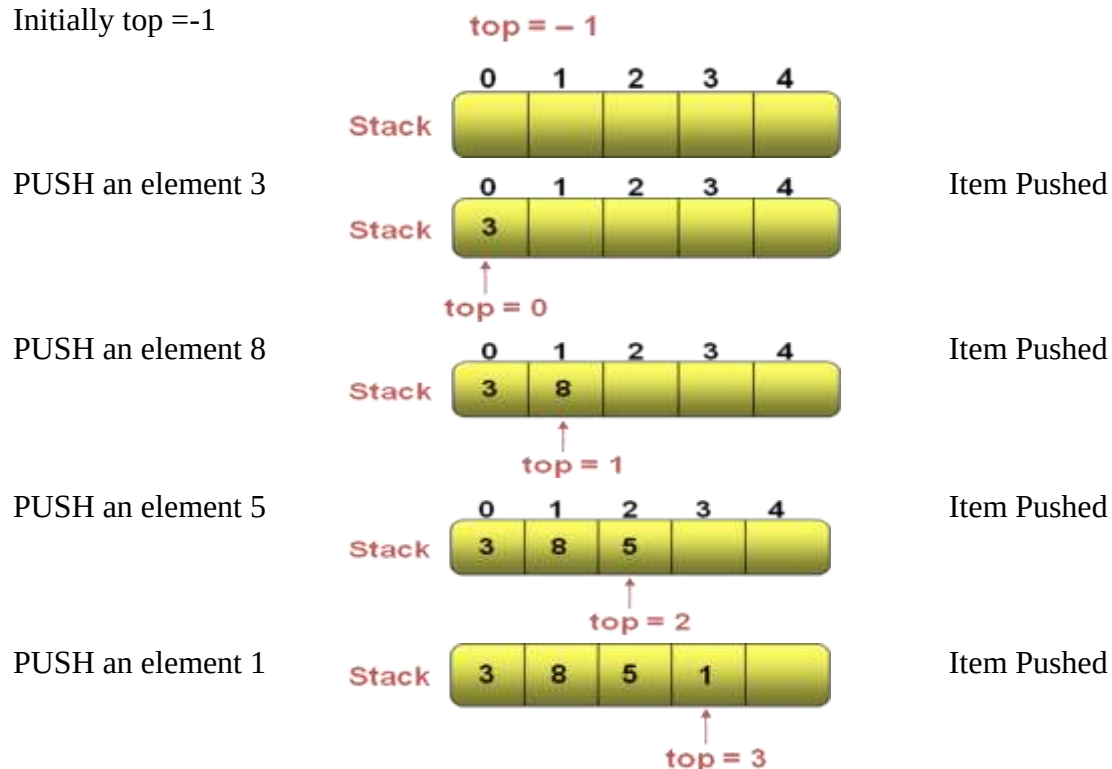
PUSH Operation:

```
int Stack[5]; // Maximum size needs to be specified in advance
int top; // Declare a variable, top to hold the index of the topmost element in the stacks
top = -1 // Initially, when the stack is empty, set top value
If top = MAX - 1:
    Display "Stack Overflow / Full"
    Exit
Increment top by 1
Store the value to be pushed at index top in the array
```

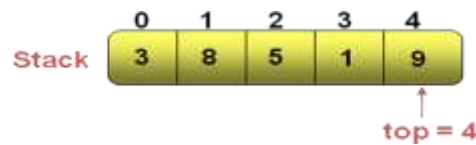
Explanation

- ◆ The stack has been implemented in an array of size 5.
- ◆ Therefore, you cannot store more than 5 elements in the stack.
- ◆ To avoid the stack overflow, you need to check for the stack full condition before pushing an element into the stack.

Initially top = -1

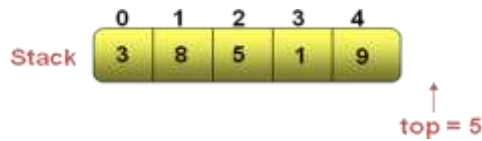


PUSH an element 9



Item Pushed

PUSH an element 2



Stack overflow / Full

POP Operation:If $\text{top} = -1$:

Display "Stack Underflow / Empty"

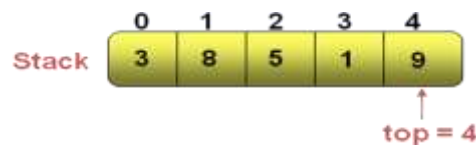
Exit

Retrieve the value stored at index top

Decrement top by 1

Explanation

POP an element



Item Popped is 9

POP an element



Item Popped is 1

POP an element



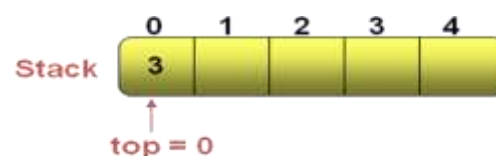
Item Popped is 5

POP an element



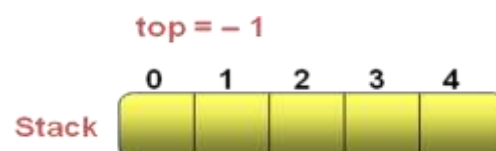
Item Popped is 8

POP an element



Item Popped is 3

POP an element

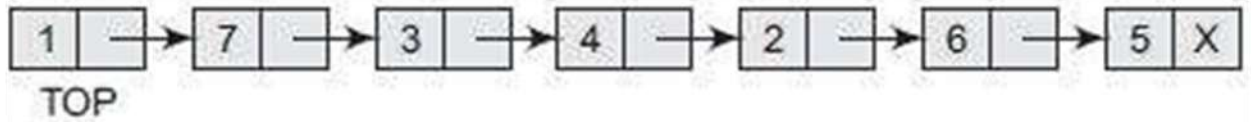


Stack Underflow / Empty

Implementing a Stack Using Linked List

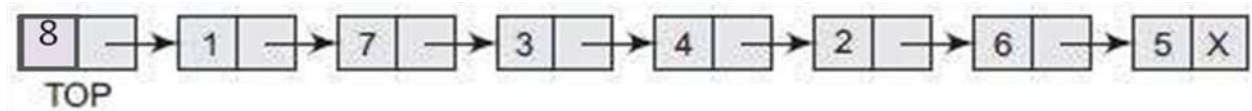
- **The drawback of using** array method is fixed size, but in this method it's removed.
- The storage requirement of linked representation of the stack with n elements is $O(n)$ and the typical time requirement for the operation is $O(1)$.
- In a linked stack, every node has two parts : one that stores data and another that stores the address of the next node.

- The START pointer of the linked list is used as TOP.



Push Operation:

- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack.
- To insert an element with value 8, we first check if TOP=NULL.
- If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part.
- The new node will then be called TOP.
- If TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP.



Linked stack after inserting a new node

Push Operation

Step 1: Allocate memory for the new node and name it as NEW_NODE

Step 2: SET NEW_NODE -> DATA = VAL

Step 3: IF TOP = NULL

SET NEW_NODE -> NEXT = NULL

SET TOP = NEW_NODE

ELSE

SET NEW_NODE -> NEXT = TOP

SET TOP = NEW_NODE

Step 4: END

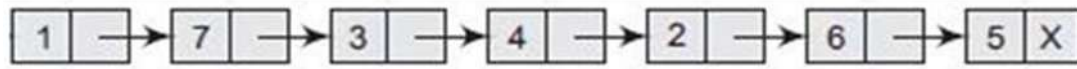
Explanation:

- In Step 1, memory is allocated for the new node.
- In Step 2, the DATA part of the new node is initialized with the value to be stored in the node.
- In Step 3, we check if the new node is the first node of the linked list.
- This is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

Pop Operation:

- The pop operation is used to delete an element from the stack.
- The element is deleted at the topmost position of the stack.
- However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done.

- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.
- In case $TOP \neq NULL$, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes like this.



Linked stack after deleting a node

Pop Operation

```

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

Explanation:

- In Step 1, we first check for the UNDERFLOW condition: $TOP \neq NULL$
- In Step 2, we use a pointer PTR that points to TOP.
- In Step 3, TOP is made to point to the next node in sequence.
- In Step 4, the memory occupied by PTR is given back to the free pool.

Peek Operation:

- The Peek operation is used to see the value of the topmost element of the stack without deleting it from the stack.
- Peek operation first checks if the stack is empty, i.e., if $TOP = NULL$, then an appropriate message is printed, else the value is returned.

Peek Operation

```

Step 1: IF TOP = NULL
        PRINT "UNDERFLOW"
        ELSE
        RETURN TOP->DATA
Step 2: END
  
```

Explanation

- In step 1, if $TOP = NULL$ means that the stack is empty.
- Else it will return the data on TOP, which will be our output or Peeped out Value.

Applications of Stack

- Implementing function calls
- Maintaining the UNDO list for an application
- Checking the nesting of parentheses in an expression
- Evaluating expressions

◆ Implementing function calls:

- ◆ Consider an example. There are three functions, F1, F2, and F3. Function F1 invokes F2 and function F2 invokes F3, as shown.

```

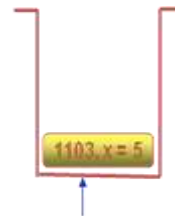
void F1 ()
{
1100      int x;
1101      x = 5;
1102      F2 ();
1103      print(x);
}
void F2 (int x)
{
1120      x = x + 5;
1121      F3 (x);
1122      print(x);
}
void F3 (int x)
{
1140      x = x * 2;
1141      print x;
}

```

Assuming these instructions at the given locations in the memory.

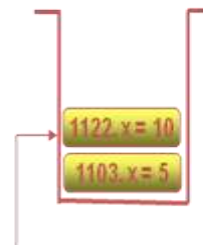
The execution starts from function F1.

x = 5



Address and the local variable of F1

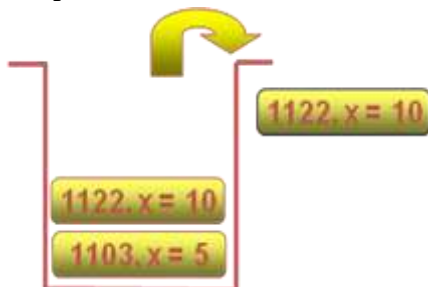
x = 10



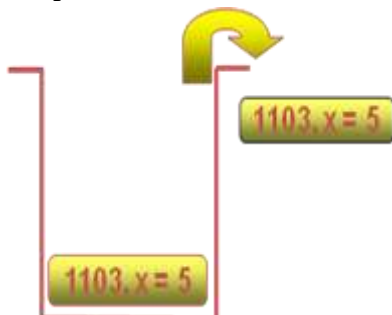
Address and the local variable of F2

X=20

Output: 20



Output: 20 10



Output: 20 10 5

- ◆ Maintaining the UNDO list for an application:
 - ◆ Consider that you made some changes in a Word document. Now, you want to revert back those changes. You can revert those changes with the help of an UNDO feature.
 - ◆ The UNDO feature reverts the changes in a LIFO manner. This means that the change that was made last is the first one to be reverted.
 - ◆ You can implement the UNDO list by using a stack.

◆ Conversion of number bases:

- ◆ Conversion of a decimal number (base 10) to an equivalent number in any other base (say x).
- ◆ This is accomplished by dividing the number by x and the repeatedly dividing the quotient of division by x until a zero quotient is obtained.
- ◆ The remainder after each division must be stored and at the end, the reminders must be written out in the reverse order.
- ◆ Again since the remainders are required in the reverse order, a LIFO list, i.e., a stack can be used for storing the reminders after each division.

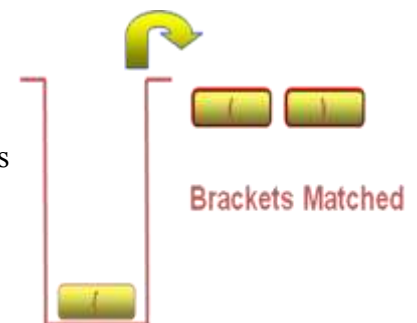
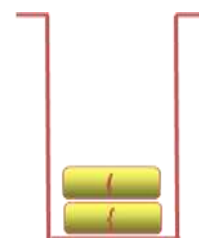
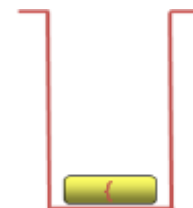
2	24	
2	12	- 0
2	6	- 0
2	3	- 0
	1	- 1

↑

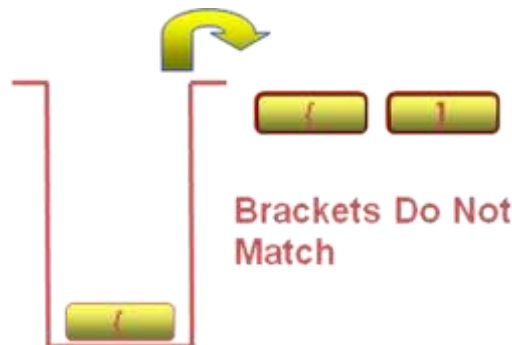
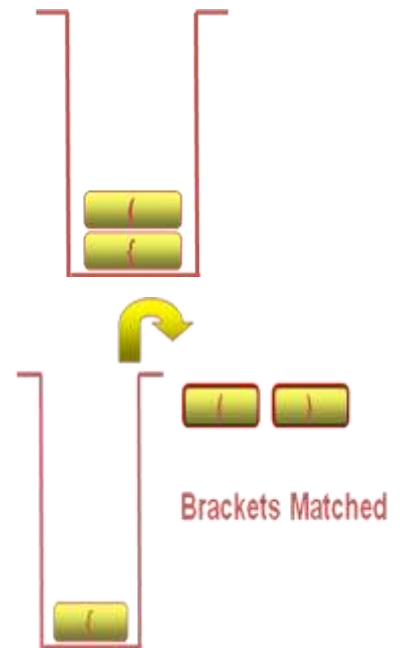
11000

◆ Checking the **nesting of parentheses** in an expression:

- ◆ You can do this by checking the following two conditions:
 - ◆ The number of left parenthesis should be equal to the number of right parenthesis.
 - ◆ Each right parenthesis is preceded by a matching left parenthesis.
- ◆ **Example: $\{(a + b) \times (c + d) + (c \times d)\}$**
- ◆ Suppose the expression is: $\{(a + b) \times (c + d) + (c \times d)\}$
- ◆ Scan the expression from left to right.
- ◆ The first entry to be scanned is '{', which is a left parenthesis.
- ◆ Push it into the stack.
- ◆ The next entry to be scanned is '(', which is a left parenthesis.
- ◆ Push it into the stack.
- ◆ The next entry is 'a', which is an operand. Therefore, it is discarded.
- ◆ The next entry is '+', which is an operator. Therefore, it is discarded.
- ◆ The next entry is 'b', which is an operand. Therefore, it is discarded.
- ◆ The next entry to be scanned is ')', which is a right parenthesis
- ◆ POP the topmost entry from the stack.
- ◆ Match the two brackets.
- ◆ The next entry to be scanned is '×', which is an operator. Therefore, it is discarded.



- ◆ The next entry to be scanned is '(', which is a left parenthesis
- ◆ Push it into the stack
- ◆ The next entry to be scanned is 'c', which is an operand.
Therefore it is discarded.
- ◆ The next entry to be scanned is '+', which is an operator.
Therefore it is discarded
- ◆ The next entry to be scanned is 'd', which is an operand.
Therefore it is discarded.
- ◆ The next entry to be scanned is ')', which is a right parenthesis.
- ◆ POP the topmost element from the stack.
- ◆ Match the two brackets.
- ◆ The next entry to be scanned is ']', which is a right parenthesis.
- ◆ POP the topmost element from the stack.
- ◆ Match the two brackets. Brackets do not match. The Expression is INVALID.



INFIX, PREFIX AND POSTFIX EXPRESSIONS:

There are three ways to represent an arithmetic expression. They are

1. Infix Expressions
2. Prefix Expressions and
3. Postfix Expressions

INFIX: An expression is said to be infix expression if the operator is placed in between the operands
i.e., Operand 1 Operator Operand 2.

Example : $A + B$

PREFIX: An expression is said to be prefix expression if the operator precedes the operands as
Operator Operand 1 Operand 2.

Example : $+ A B$

POSTFIX: An expression is said to be postfix expression if the operator follows the operands as
Operand 1 Operand 2 Operator.

Example : $A B +$

Example : The steps in conversion of $A + (B - C / D) - E$ to prefix form of expression is shown below:

Sub expression	Operator	Operand 1	Operand 2	Prefix form
C / D	$/$	C	D	$/ C D$
$(B - C / D)$	$-$	B	$/ C D$	$- B / C D$
$A + (B - C / D)$	$+$	A	$- B / C D$	$+ A - B / C D$
$A + (B - C / D) - E$	$-$	$+ A - B / C D$	E	$- + A - B / C D E$

Example : The steps in conversion of $A + (B - C / D) - E$ to postfix form of expression is shown below:

Sub expression	Operand 1	Operand 2	Operator	Postfix form
C / D	C	D	$/$	$C D /$
$(B - C / D)$	B	$C D /$	$-$	$B C D / -$
$A + (B - C / D)$	A	$B C D / -$	$+$	$A B C D / - +$
$A + (B - C / D) - E$	$A B C D / - +$	E	$-$	$A B C D / - + E -$

Evaluation of Prefix expression:

The steps to evaluate the Prefix expression are:

1. Find the right-most operator in the expression.
2. Select the two operands immediately to the right of the operator found.
3. Perform the indicated operation.
4. Replace the operator & the operand with the result.

Expression	Operator	Operand 1	Operand 2	Infix form	
$- + A - B / C D E$	$/$	C	D	C / D	\$1
$- + A - B \$1 E$	$-$	B	$\$1 \rightarrow C / D$	$(B - C / D)$	\$2
$- + A \$2 E$	$+$	A	$\$2 \rightarrow (B - C / D)$	$A + (B - C / D)$	\$3
$- \$3 E$	$-$	$\$3 \rightarrow A + (B - C / D)$	E	$A + (B - C / D) - E$	

Evaluation of Postfix Expression:

The steps to evaluate the Postfix expression are:

1. Find the left-most operator in the expression.
2. Select the two operands immediately to the left of the operator found.
3. Perform the indicated operation.
4. Replace the operator & the operand with the result

Expression	Operator	Operand 1	Operand 2	Infix form	
$A B C D / - + E -$	$/$	C	D	C / D	\$1
$A B \$1 - + E -$	$-$	B	$\$1 \rightarrow C / D$	$(B - C / D)$	\$2
$A \$2 + E -$	$+$	A	$\$2 \rightarrow (B - C / D)$	$A + (B - C / D)$	\$3
$\$3 E -$	$-$	$\$3 \rightarrow A + (B - C / D)$	E	$A + (B - C / D) - E$	

Conversion of infix to postfix

Step 1: Scan the infix expression from left to right.

Step 2 : If the character is an operand, place it on to the output.

Step 3 : If the character is an operator, push it onto the stack. If the stack operator has a higher or equal priority than input operator then pop that operator from the stack and place it onto the output.

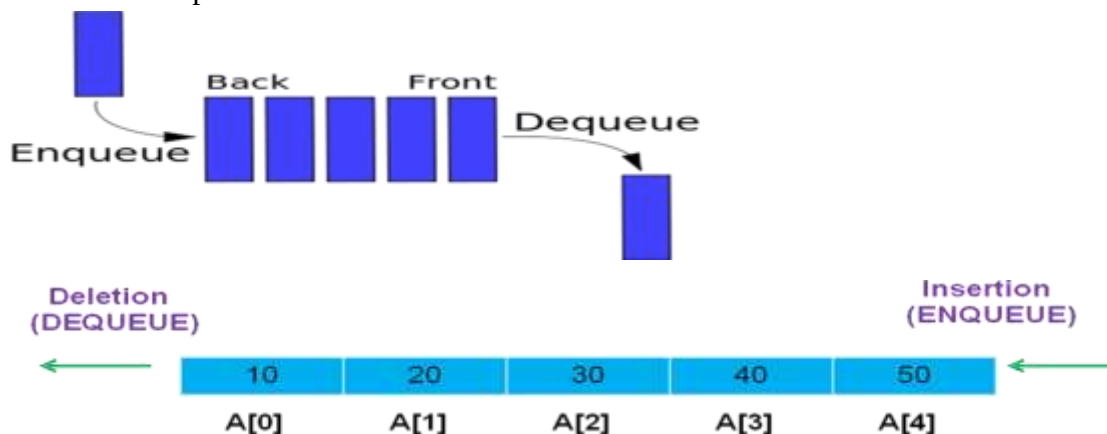
Step 4 : If the character is a left parenthesis, push it onto the stack.

Step 5 : If the character is a right parenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

Infix Expression : A+B*(C^D-E)				
Token	Action	Result	Stack	Notes
A	Add A to the result	A		
+	Push + to stack	A	+	
B	Add B to the result	AB	+	
*	Push * to stack	AB	* +	* has higher precedence than +
(	Push (to stack	AB	(* +	
C	Add C to the result	ABC	(* +	
^	Push ^ to stack	ABC	^ (* +	
D	Add D to the result	ABCD	^ (* +	
-	Pop ^ from stack and add to result	ABCD^	(* +	- has lower precedence than ^
	Push - to stack	ABCD^	- (* +	
E	Add E to the result	ABCD^E	- (* +	
)	Pop - from stack and add to result	ABCD^E-	(* +	Do process until (is popped from stack
	Pop (from stack	ABCD^E-	* +	
	Pop * from stack and add to result	ABCD^E-*	+	Given expression is iterated, do Process till stack is not Empty, It will give the final result
	Pop + from stack and add to result	ABCD^E-*+		
Postfix Expression : ABCD^E-*+				

QUEUE

- Queue is Linear Data Structure
- It follows First In First Out(FIFO) principle
- It has two pointers front and rear.



Operations on Queue

The fundamental operations performed on queue are

- **Enqueue** : The process of inserting an element in the queue.
- **Dequeue** : The process of deleting an element from the queue.

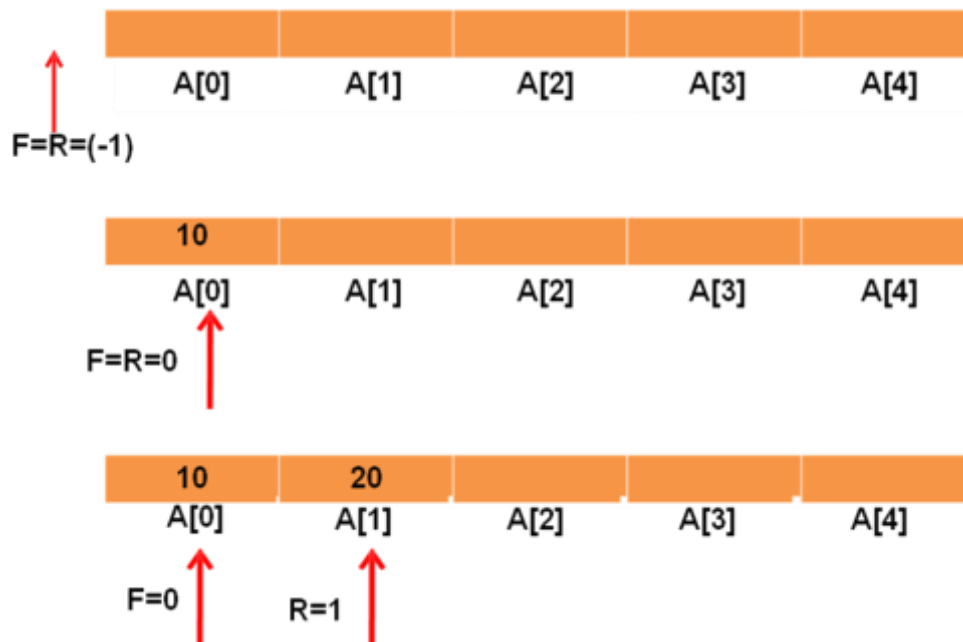
Exception Conditions

- **Overflow** : Attempt to insert an element, when the queue is full is said to be overflow condition.
- **Underflow** : Attempt to delete an element from the queue, when the queue is empty is said to be underflow.

Array Implementation of Queue

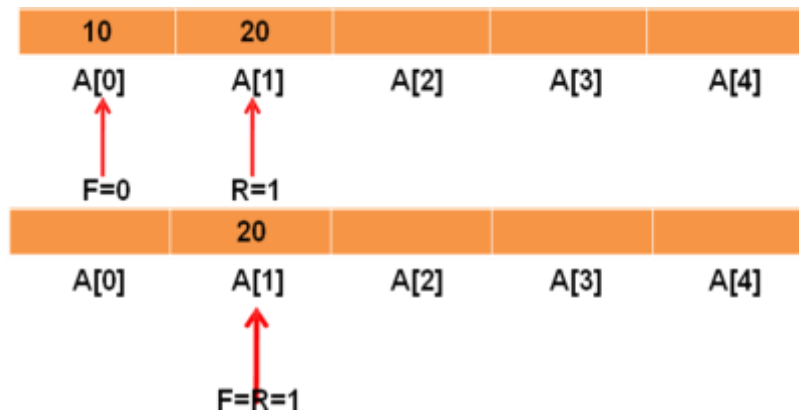
Routine for Enqueue / Insertion:

Step 1: If $REAR = MAX - 1$ then
Write "Queue is Overflow"
Goto step 4
Step 2: IF $FRONT = -1$ and $REAR = -1$
SET $FRONT = REAR = 0$
ELSE
SET $REAR = REAR + 1$
Step 3: SET $QUEUE[REAR] = NUM$
Step 4: EXIT



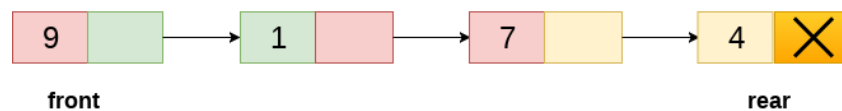
Routine for Dequeue / Deletion:

Step 1: IF $FRONT = -1$ OR $FRONT > REAR$
Write "Queue is Underflow"
ELSE
SET $VAL = QUEUE[FRONT]$
SET $FRONT = FRONT + 1$
Step 2: EXIT



Linked List Implementation of Queue

- Insertion and deletions are performed at rear and front end respectively.
- If front and rear both are NULL, it indicates that the queue is empty.



Routine for Enqueue / Insertion:

Step 1: Allocate the space for the new node PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
 SET FRONT = REAR = PTR
 SET FRONT -> NEXT = REAR -> NEXT = NULL
 ELSE
 SET REAR -> NEXT = PTR
 SET REAR = PTR
 SET REAR -> NEXT = NULL
Step 4: END

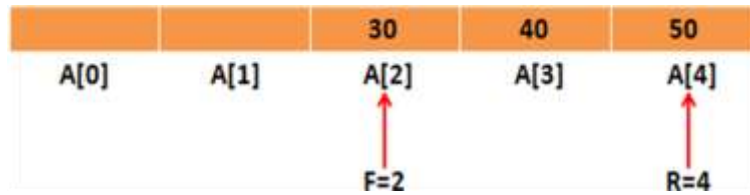
Routine for Dequeue / Deletion:

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

CIRCULAR QUEUE

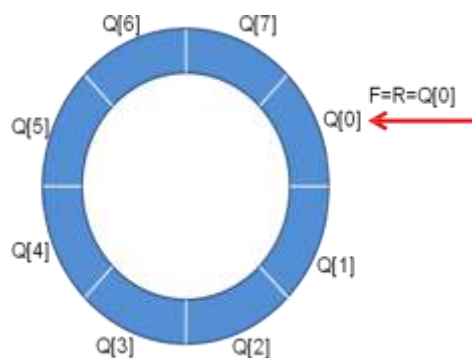
Why Circular Queue is needed?

Problem: Wastage of memory in standard queue in operation DEQUEUE



What is Circular Queue?

- The Arrangement of the elements $Q[0], Q[1], \dots, Q[n]$ in a circular fashion with $Q[n]$ following $Q[0]$ is called Circular Queue.
- The last node is connected to first node to make a circle.
- Initially, Both Front and Rear pointers points to the beginning of the array.
- It is also known as “Ring Buffer”.



Insert Operations on Circular Queue

Step 1: IF FRONT=0 and REAR=MAX-1 Then

Write “Overflow”

Goto Step 4

Step 2: IF FRONT = REAR=-1 then

SET FRONT=REAR=0

ELSE IF REAR=MAX-1 and FRONT!=0

SET REAR=0

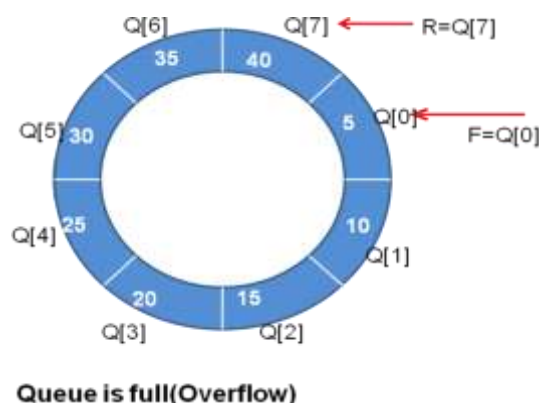
ELSE

SET REAR=REAR+1

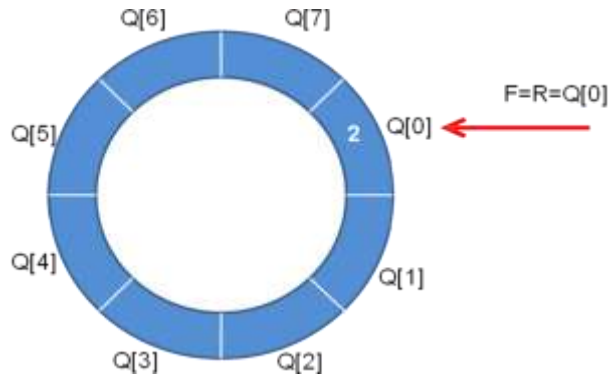
Step 3: SET QUEUE[REAR]=VAL

Step 4: EXIT

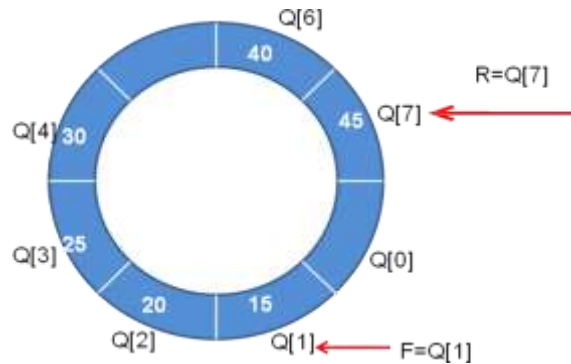
IF FRONT=0 and REAR=MAX-1 Then Write “Overflow”



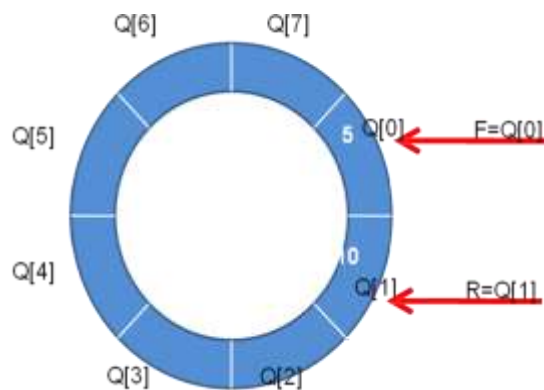
If Front = Rear = -1 then Front = Rear = 0



If REAR=MAX-1 and FRONT!=0



Rear=Rear+1



Deletion Operations on Circular Queue

Step 1: If FRONT = -1 then
Write ("Circular Queue Underflow")
GOTO Step 4

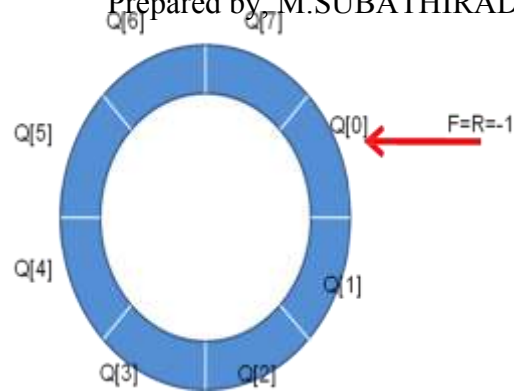
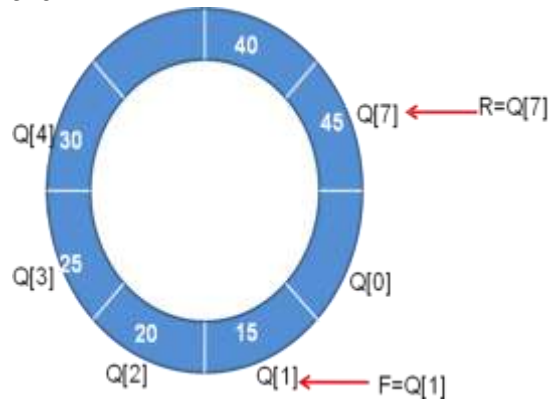
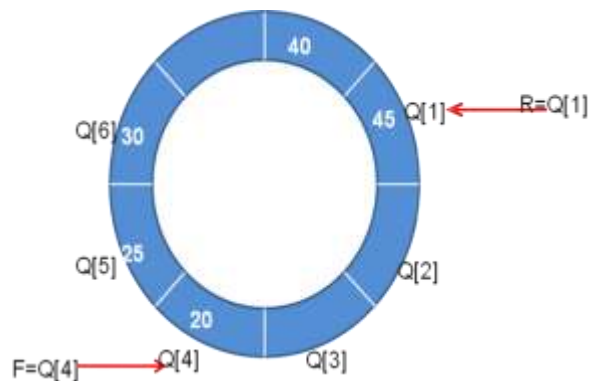
Step 2: SET VAL=QUEUE[FRONT]

Step 3: If FRONT = REAR then
SET FRONT=REAR=-1
ELSE
IF FRONT=MAX-1
SET FRONT=0
ELSE
SET FRONT=FRONT+1

Step 4: EXIT

Queue is Empty(Underflow)

Prepared by, M.SUBATHIRADEVI, AP/CSE, SANCET

**If Front = Rear = 0 then Front = Rear = -1****If Front=MAX-1 then Front=0****Front=Front+1**

PRIORITY QUEUE

- In priority queue, each element is assigned a priority.
- Priority of an element determines the order in which the elements will be processed.
- Rules:
 1. An element with higher priority will be processed before an element with a lower priority.
 2. Two elements with the same priority are processed on a First Come First Serve basis.

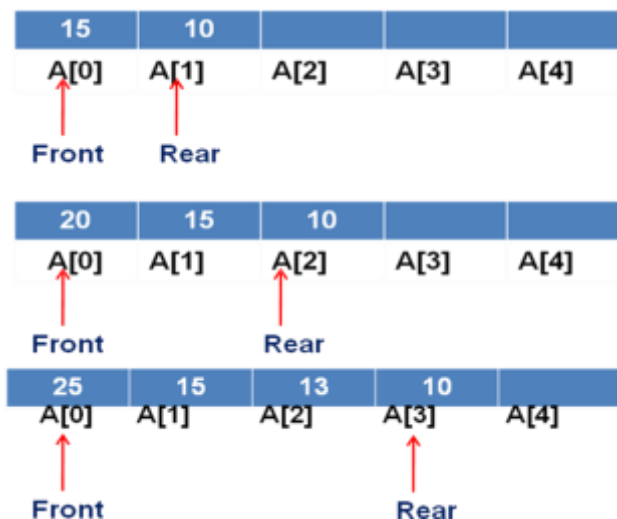
Types of Priority Queue

- Ascending Priority Queue : In this type of priority queue, elements can be inserted into any order but only the smallest element can be removed.
- Descending Priority Queue : In this type of priority queue, elements can be inserted into any order but only the largest element can be removed.

Array Representation of Priority Queue

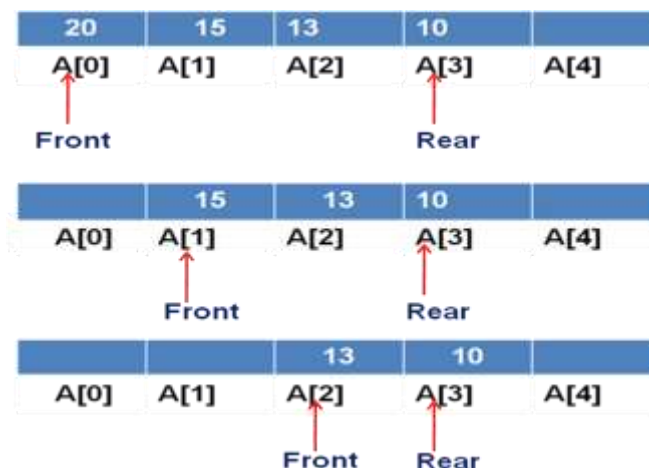
Insertion Operation:

- While inserting elements in priority queue we will add it at the appropriate position depending on its priority
- It is inserted in such a way that the elements are always ordered either in Ascending or descending sequence



Deletion Operation:

- While deletion, the element at the front is always deleted.

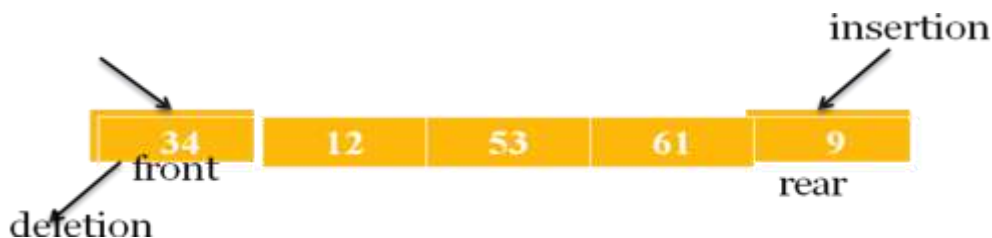


DEQUE / DOUBLE ENDED QUEUE

- A list in which elements can be inserted or deleted either end
- It is also known as “Head-Tail Linked List”
- It has two pointers LEFT and RIGHT, which point to either end of the deque.
- Dequeue can be implemented using Circular Array or a Circular doubly linked list where Dequeue[n-1] is followed by Dequeue[0]
- Input Restricted Deque: In this deque, insertion can be done only at one of the end, while deletion can be done from both ends.



- Output Restricted Deque: In this deque, deletion can be done only at one of the ends, while insertions can be done on both ends

**Applications of queues with examples:****1. CPU Scheduling in Operating Systems**

- Use: Queues manage processes waiting to be executed by the CPU.
- Example: In First Come, First Served (FCFS) scheduling, processes are added to the ready queue as they arrive and executed in order.
- Why Queue?: Because it follows the FIFO principle — the first process that arrives is executed first.

2. Print Spooling

- Use: In a printing system, print jobs from multiple users are stored in a print queue.
- Example: If three users send documents to a printer, the documents are printed in the order they were submitted.
- Why Queue?: Ensures fair and orderly processing of print jobs.

3. Disk Scheduling

- Use: Disk controllers maintain a queue of requests for reading/writing data blocks.
- Example: A database server may have multiple read/write requests for different files — these are queued for disk access.

4. Network Data Transmission

- Use: Routers and switches use queues to store packets before forwarding them.
- Example: In a busy network, packets wait in a queue until the network link is free.
- Why Queue?: Prevents packet loss by storing them temporarily in buffers.

5. Call Center Systems

- Use: Customer calls are queued when all operators are busy.

- Example: The call waiting system stores callers in a queue until a representative becomes available.

6. Breadth-First Search (BFS) in Graphs

- Use: BFS uses a queue to explore nodes level by level.
- Example: Finding the shortest path in an unweighted graph requires BFS, which queues nodes to be visited.

7. Order Processing in E-commerce

- Use: Orders are processed in the order they are placed.
- Example: Amazon's warehouse processes orders in the same sequence they are received to ensure fairness.

8. Banking and Ticket Counters

- Use: Customers are served in order of arrival.
- Example: ATM queues or railway ticket booking lines follow FIFO rules.

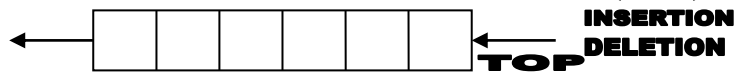
9. Simulation of Real-life Scenarios

- Use: Queues simulate systems like traffic management, supermarket checkout lines, and airport boarding.
- Example: Traffic lights use queues to control cars waiting to move.

PART-A

1. Define Stack. Or Define LIFO.

A stack is a collection of data items that can be accessed at only one end, called top. This means that the items are inserted and deleted at the TOP. The last item that is inserted in a stack is the first one to be deleted. Therefore, a stack is called a Last-In-First-Out (LIFO) data structure.



2. What are the various operations involved in Stack? (or)

List the operations of the stack. (Nov / Dec 2024)

Push: When we insert an item into a stack, we say that we have pushed the items on to the stack.

Pop: when we delete an item from a stack, we say that we have popped the item from the stack.

3. What are the necessary conditions for insertion and deletion operations on static stack?

(Nov / Dec 2021)

Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

4. What are the various methods to implement Stack?

The various methods to implement Stack are Array and Linked List.

5. List the Applications of Stack.

- Implementing Function calls
- Checking the validity of an expression
- Maintaining the undo list for an application
- Evaluating an expression

6. Define Infix Expression.

An expression is said to be infix expression, if the operator is placed between the operands.

Operand1 Operator Operand2
A + B

7. Define Prefix Expression.

An expression is said to be prefix expression, if the operator precedes the operands.

Operator Operand1 Operand2
+ A B

8. Define Postfix Expression.

An expression is said to be postfix expression, if the operator follows the operands.

Operand1	Operand2	Operator
A	B	+

9. Convert the infix expression $a + b * c + (d + e + f) / g$ to postfix expression using stack.

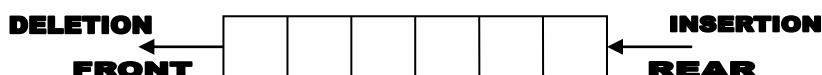
$a bc* + de + f + g /$

10. Convert the infix expression $(a + b) * (c - d)$ to prefix expression. (Nov / Dec 2022)

Prefix expression : $*+ab-cd$

11. Define Queue. Or Define FIFO. (Nov / Dec 2022)

A queue is a list of elements in which items are inserted at one end of the queue (REAR) and deleted from the other end of the queue (FRONT). The first element to be inserted in the Queue is the first one to be deleted. Therefore, a queue is said to be First-In-First-Out (FIFO) data structure.

**12. What are the various operations involved in Queue?**

Enqueue: An element is always inserted at the rear end of the queue.

Dequeue: An element is always deleted at the front end of the queue.

13. What are the various methods to implement Queue?

The various methods to implement Queue are Array and Linked List.

14. List the Applications of Queue. (Nov / Dec 2023)

- Printer Spooling
- Mail service
- CPU scheduling
- Keyboard buffering

15. List the types of Queue.

- Circular Queue
- Deque
- Priority Queue

16. Define Circular Queue.

In Circular Queue, if we reach the end of the queue while inserting elements into the queue, it is possible to insert new elements into queue if array slots at the beginning of the circular queue are empty.

17. Give the limitations of linear queue. (Apr / May 2023)

- Wastage of memory as freed spaces at the front cannot be reused.
- Fixed size, so it can overflow even if space is available at the front.

18. What condition is required for a queue to behave as a circular queue? (Nov / Dec 2021)

Circular Queue is also a linear data structure, which follows the principle of FIFO (First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

19. Differentiate between Stack and Queue.

Stack	Queue
A stack is a collection of data items that can be accessed at only one end, called TOP.	A queue is a list of elements in which items are inserted at one end of the queue (REAR) and deleted from the other end of the queue (FRONT).
The last item that is inserted in a stack is the first one to be deleted. Therefore, a stack is called a Last-In-First-Out (LIFO) data structure.	The first element to be inserted in the Queue is the first one to be deleted. Therefore, A queue is said to be First-In-First-Out (FIFO) data structure

20. What do you mean by Overflow and Underflow conditions in Stack? (Apr / May 2023) (or) Recall when an overflow and underflow condition occur during stack operations. (Nov / Dec 2023)

When a new element is inserted into the stack, where there is no space to insert in the array (maximum size of array is reached) and this situation is said to be Stack Overflow.

When all the elements are deleted, top points to bottom of the stack is empty, it is not possible to delete an element and this situation is called Stack underflow.

21. A circular queue has a size of 5 and has 3 elements 11, 30, 41 where F = 1 and R = 3. After inserting 50 and 60, what is the value of F and R? Trying to insert 33 at this stage what happens. Delete 2 elements from the queue and insert 71, 98. Show the sequence of steps for the above operations. (Apr / May 2024)

Process	Front & Rear	Array
Initial state	F = 1, R = 3	[11, 30, 41, -, -]
Insert 50 → put at 4	F = 1, R = 4	[11, 30, 41, 50, -]
Insert 60 → put at 5	F = 1, R = 5 (queue now full)	[11, 30, 41, 50, 60]
Try to insert 33 → Overflow	F = 1, R = 5	No change.
Delete → remove 11	F = 2, R = 5	[-, 30, 41, 50, 60]
Delete → remove 30	F = 3, R = 5	[-, -, 41, 50, 60]
Insert 71 → wrap R to 1	F = 3, R = 1	[71, -, 41, 50, 60]
Insert 98 → R = 2	F = 3, R = 2	[71, 98, 41, 50, 60]

Final state:

- F = 3, R = 2
- Queue sequence from front: 41, 50, 60, 71, 98.
- Overflow occurred when trying to insert 33 while full.

22. A letter means push and an asterisk means pop in the following sequence. Give the sequence of values returned by the pop operations, when this sequence of operations is performed on an initially empty LIFO stack. `U S G \* O \* O L D \* \* \* E V \* \* \* Y \* O \* \*` (Apr / May 2024)

Rule:

- Letter → Push onto stack
- Asterisk (*) → Pop from stack and record the popped value

Start: Stack = `[]`, Output = `[]`

1. U → Push → `[U]`
2. S → Push → `[U, S]`
3. G → Push → `[U, S, G]`
4. * → Pop → G → `[U, S]`, Output = `[G]`
5. O → Push → `[U, S, O]`
6. * → Pop → O → `[U, S]`, Output = `[G, O]`
7. O → Push → `[U, S, O]`
8. L → Push → `[U, S, O, L]`
9. D → Push → `[U, S, O, L, D]`
10. * → Pop → D → `[U, S, O, L]`, Output = `[G, O, D]`
11. * → Pop → L → `[U, S, O]`, Output = `[G, O, D, L]`
12. * → Pop → O → `[U, S]`, Output = `[G, O, D, L, O]`
13. E → Push → `[U, S, E]`
14. V → Push → `[U, S, E, V]`
15. * → Pop → V → `[U, S, E]`, Output = `[G, O, D, L, O, V]`

16. $\backslash * \rightarrow \text{Pop} \rightarrow E \rightarrow '[U, S]'$, Output = $'[G, O, D, L, O, V, E]'$
 17. $\backslash * \rightarrow \text{Pop} \rightarrow S \rightarrow '[U]'$, Output = $'[G, O, D, L, O, V, E, S]'$
 18. $Y \rightarrow \text{Push} \rightarrow '[U, Y]'$
 19. $\backslash * \rightarrow \text{Pop} \rightarrow Y \rightarrow '[U]'$, Output = $'[G, O, D, L, O, V, E, S, Y]'$
 20. $O \rightarrow \text{Push} \rightarrow '[U, O]'$
 21. $\backslash * \rightarrow \text{Pop} \rightarrow O \rightarrow '[U]'$, Output = $'[G, O, D, L, O, V, E, S, Y, O]'$
 22. $\backslash * \rightarrow \text{Pop} \rightarrow U \rightarrow '[]'$, Output = $'[G, O, D, L, O, V, E, S, Y, O, U]'$

Final Output Sequence (values popped in order): G, O, D, L, O, V, E, S, Y, O, U

23. Distinguish between static data structure and dynamic data structures. (Nov / Dec 2024)

Static Data Structure	Dynamic Data Structure
Memory size is fixed at compile time.	Memory size can change at runtime.
Easier to implement, as memory is allocated in advance.	More complex to implement, as memory must be allocated and freed during execution.
May lead to memory wastage if allocated size is larger than needed.	Efficient memory usage as memory is allocated as required
Examples: Arrays, Static stacks, Static queues.	Examples: Linked lists, Dynamic stacks, Dynamic queues.
Faster access as elements are stored contiguously in memory.	Slower access as elements may be stored at non-contiguous memory locations

24. Give the difference between circular queue and deque. (Nov / Dec 2023)

Circular Queue	Deque (Double-Ended Queue)
A linear data structure where the last position is connected back to the first position to form a circle.	A linear data structure where insertion and deletion can be done from both ends.
Insertion is allowed only at the rear end, and deletion is allowed only at the front end.	Insertion and deletion are allowed at both front and rear ends.
Used mainly to utilize memory efficiently and avoid wastage of space in a simple queue.	Used where flexibility in adding/removing elements from both ends is needed.
Example operations: Enqueue (rear), Dequeue (front).	Example operations: Input-restricted deque, Output-restricted deque.
More restrictive — cannot directly insert at the front or delete at the rear.	More flexible — supports insertion/deletion from both ends.

PART- B & C

STACK

1. Explain the concept of Stack using array in detail.
2. Outline the pseudo-code for performing the following operations in stacks using array implementation. (1) Create (2) Push (3) Pop (4) Stack Top (Nov / Dec 2023)
3. Define stack? Explain the steps to implement:
 (i) Stack using arrays (6) (ii) Stack using linked list (7) (Nov / Dec 2024)
4. Explain the concept of Stack using linked list in detail.
5. Write an algorithm for Push and Pop operations on stack using linked list. (Nov / Dec 2019) (Apr / May 2023)
6. Write procedures for significant operations on a Stack data structure. Apply stack, to convert the following infix expression to the corresponding postfix expression: $a + b / (d - e) - f$. (Nov / Dec 2021)

7. Outline the algorithm for evaluating a postfix expression using stack data structure with an example. (Nov / Dec 2022)
8. Give the illustration, to convert the given infix expression into postfix expression:
 $(A - B) - C + D * E + F$
9. Explain the applications of Stacks in detail.
10. Give the illustration, to check whether the give expression has parenthesis matched:
 $\{(A - B) - C + D * E + F\}$
11. Convert the following infix expression to postfix expression with a neat sketch using stacks.
(i) $((A - (B + C) * D) * (E + F))$ (ii.) $(A / (B - C + D)) * (E - A) * C$ (Nov / Dec 2023)
 - Devise an algorithm for evaluating the above postfix expression using stacks and analyze the complexity. (Nov / Dec 2023)
12. Write a C function for the following conversions. (Apr / May 2024)
 - (i) Infix to postfix expression
 - (ii) Evaluate the postfix expression
13. Write algorithms to check if the given parenthesized expression contains balanced parenthesis and to convert such expression to postfix form and evaluate it. Illustrate with example. (Apr / May 2019)
14. Identify how stacks are used for checking whether an expression is balanced or not for every right and left brace, bracket and parenthesis. (Nov / Dec 2023)

QUEUE

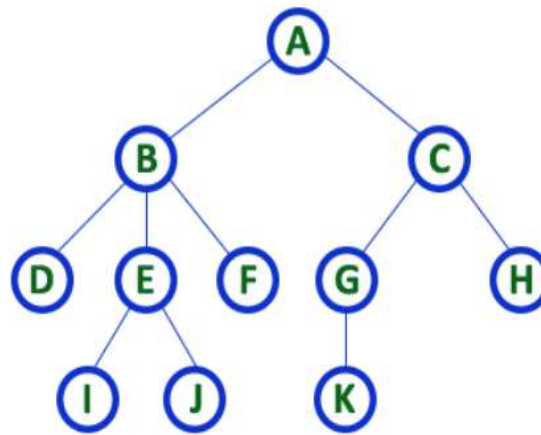
15. Explain the concept of Queue using array in detail.
16. State the procedure for performing enqueue, dequeue, IsEmpty, and IsFull operations in a queue data structure with example data. (Nov / Dec 2023)
17. (i) Write an algorithm to insert and delete an element from a simple queue. (8)
(ii) Explain FIFO approach. (5) (Nov / Dec 2024)
18. Explain the concept of Queue using linked list in detail.
19. Write procedures for significant operations on a Queue data structure. List any four applications of queue data structure. (Nov / Dec 2021)
20. What is circular queue? Explain with suitable example. (Apr / May 2024)
21. State the application of Queue and explain any one application with an example. (Apr / May 2024)
22. Explain the concept of Circular Queue in detail.
23. Explain the addition and deletion operations performed on a circular queue with necessary algorithms. (Apr / May 2023)
24. State the advantage of circular queue over linear queue. Write the functions for insertion in a circular queue. (Apr / May 2019)
25. Explain the implementation of circular queue using array. How an "empty queue" is distinguished from a "full queue"? Write necessary functions to perform all valid operations on circular queue. (Nov / Dec 2024)
26. Recognize the differences between circular queue and deque. (Nov / Dec 2023)
27. Explain the applications of Queue in detail.
28. Explain the concept of Priority Queue in detail.
29. What is a deQueue? Explain its operation with example. (Nov / Dec 2019)
30. What is a deque? Outline the operations that can be performed on a deque with an algorithm, example and relevant diagrams. (Nov / Dec 2022)

UNIT III – TREES

Tree ADT – Tree Traversals - Binary Tree ADT – Expression trees – Binary Search Tree ADT – AVL Trees – Priority Queue (Heaps) – Binary Heap.

TREE:

- ✓ A tree is a non-linear data structure, which represents hierarchical relationships between individual data items.
- ✓ A tree is a connected graph without any circuits.
- ✓ If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.

**TREE TERMINOLOGY**

NODES: Each data item in a tree

EDGE/LINK: It is a connection between two nodes

PATH:

- ✓ It is a sequence of distinct nodes in which successive nodes are connected by edges in the tree.
- ✓ An important property of a tree is that there should be precisely one path connecting any two nodes.
- ✓ If there is no path between some pair of nodes, or if there is more than one path, between some pair of nodes then we call the structure as a graph and not a tree.

ROOT/ROOT NODE:

- ✓ The node at the top hierarchy of the tree is said to be Root / Root node.
- ✓ The highest level or the first level in a tree is said to be Root / Root node.
- ✓ A tree has only one root.

PARENT/PARENT NODE: A node at one level up in the hierarchy of the tree with which it is connected is said to be Parent Node. It is the immediate predecessor of a node.

CHILD/CHILD NODE: Each link to the root node are said to be Child Node. If the immediate predecessor of a node is the parent of the node then all the immediate successors of a node are known as child.

LEFT CHILD: Child which is at left side is said to be Left Child.

RIGHT CHILD: Child which is at right side is said to be Right Child.

SIBLINGS: The nodes which have same parent are called siblings.

LEAF NODES/TERMINAL NODES: Nodes that do not have any children are said to be Leaf Nodes.

INTERIOR NODES/NON-TERMINAL NODES: All intermediate nodes that traversing the given tree from its root to the terminal nodes except root node & leaf nodes are said to be Interior Nodes.

LENGTH OF A PATH: Number of edges in a path is said to be Length.

LEVEL OF A NODE: Root is at level 0. The level of any node is one plus the level of its parent.

DEPTH/HEIGHT OF A TREE: Maximum level of any leaf in the tree is said to be Height of a Tree.

DEGREE: Maximum number of children that it's possible for a node.

SUBTREE: A subset of tree is itself a tree.

EXAMPLE:

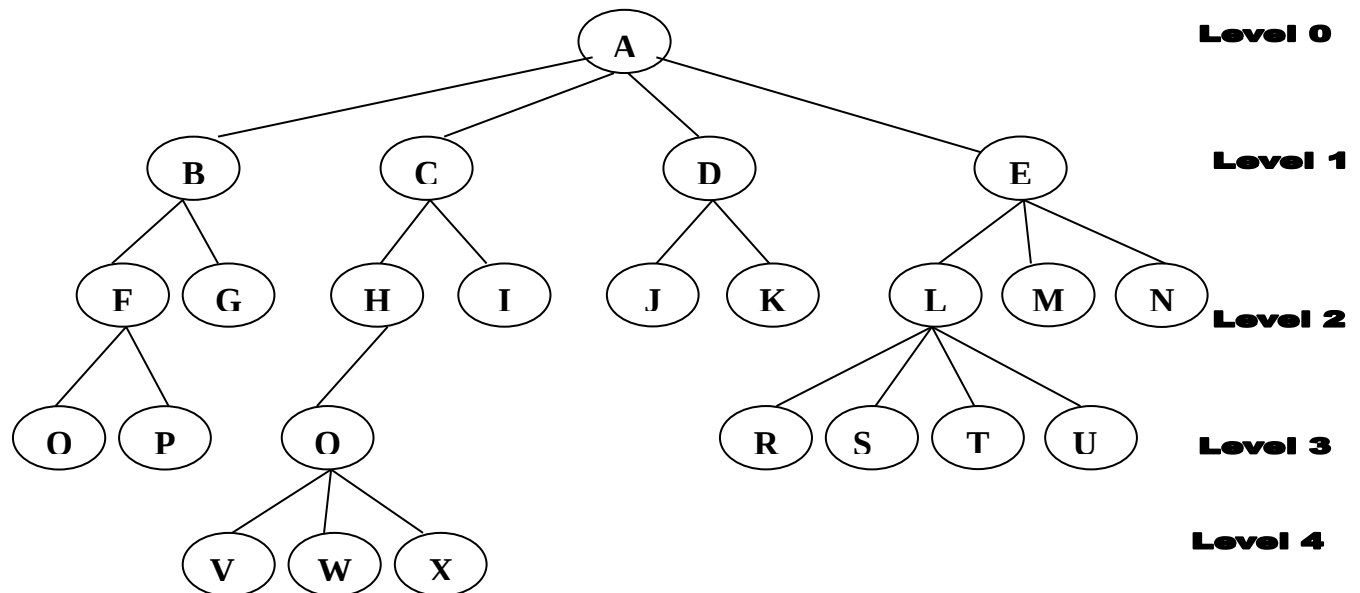


FIGURE: Example for a Tree

ROOT: A

PARENT: B, C, D, E $\rightarrow$ A; F, G $\rightarrow$ B; H, I $\rightarrow$ C; J, K $\rightarrow$ D.

CHILD: A $\rightarrow$ B, C, D, E; B $\rightarrow$ F, G; C $\rightarrow$ H, I; D $\rightarrow$ J, K; E $\rightarrow$ L, M, N; F $\rightarrow$ O, P; H $\rightarrow$ Q; L $\rightarrow$ R, S, T, U; Q $\rightarrow$ V, W, X

LEFT CHILD: B $\rightarrow$ F; C $\rightarrow$ H; D $\rightarrow$ J; F $\rightarrow$ O; H $\rightarrow$ Q;

RIGHT CHILD: B $\rightarrow$ G; C $\rightarrow$ I; D $\rightarrow$ K; F $\rightarrow$ P;

SIBLINGS: B, C, D, E; F, G; H, I; J, K; L, M, N; O, P; R, S, T, U; V, W, X

LEAF NODES: O, P, G, V, W, X, I, J, K, R, S, T, U, M, N

INTERIOR NODES: B, C, D, E, F, H, L, Q

LENGTH OF NODE W IS 4 whereas for S is 3.

HEIGHT: for R the height is 3, whereas for C is 2.

BINARY TREEDefinition:

1. A binary tree is a two array tree i.e., each node in the tree can have either 0, or 1 or 2 sub trees.
2. A binary tree is a finite ordered collection of elements in which one element is designated as root & remaining elements are partitioned into 2-disjoint sets, called left & right sub trees.
3. A binary tree is a tree that has nodes either empty or not more than 2 child nodes, each of which may be a leaf node.

TYPES OF BINARY TREE:**1) STRICTLY BINARY TREE:**

In a binary tree where all the nodes will have either 0 or 2 children. Nodes cannot have only one child.

2) COMPLETE BINARY TREE:

It is strictly binary tree in which all leaves are at the last level.

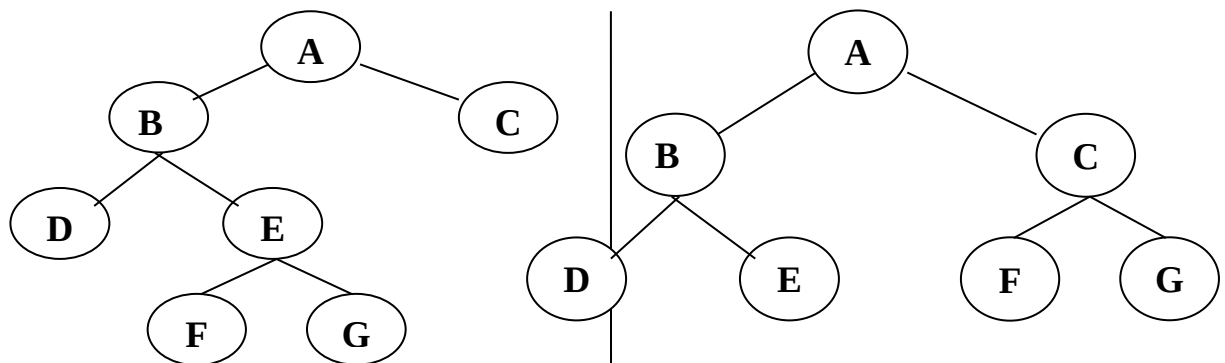
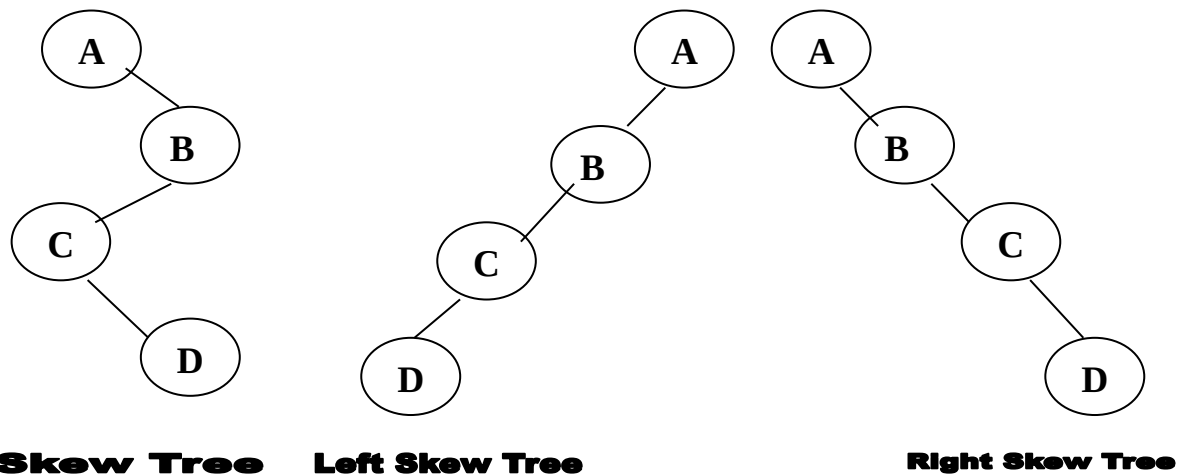


FIGURE: Strictly Binary Tree, Complete Binary Tree

3) SKEW TREES:

Except the leaf nodes, the nodes which have the only one child node i.e.; there is only one node at any level.

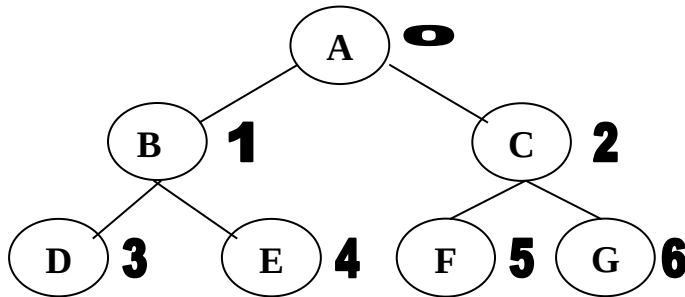
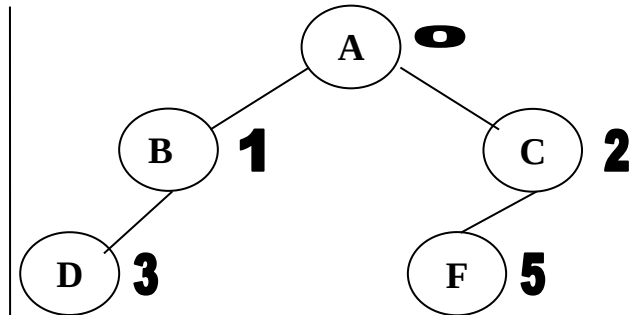


REPRESENTATION OF BINARY TREES:

- 1) Linear Representation (using arrays)
- 2) Linked Representation (using pointers)

LINEAR REPRESENTATION:

Nodes of a tree are numbered from 0, level by level.

**Figure – (a)****Figure – (b)**

An array can be used to store the nodes of a binary tree. The nodes are accessed sequentially. Linear representation of binary trees uses a single dimensional array of size $2^{h+1} - 1$ where h is the height of the tree.

The height h is 2, therefore, $2^{h+1} - 1 = 2^{2+1} - 1 = 2^3 - 1 = 8 - 1 = 7$ i.e., 0 to 6

For figure – (a), therefore, the array representation is

0	1	2	3	4	5	6
A	B	C	D	E	F	G

For figure – (b), the array representation is

0	1	2	3	4	5	6
A	B	C	D	-	F	-

After determining the size of the array, the following method is used to represent binary tree in arrays.

- 1) Store the value of the root node in the first location of the array (i.e., at index 0).
- 2) The left child of the n^{th} node is stored at location $(2n+1)$

The right child of the n^{th} node is stored at location $(2n+2)$

- 3) If there is no left or right child, the array location is left empty.
- 4) If the left child is at lc , its sibling is at $lc+1$
- 5) If the right child is at rc , its siblings is at $rc-1$
- 6) Parent of a node C is at $(\text{Position of } C - 1) / 2$.

ADVANTAGE:

- 1) Representing using array is simple & easy to implement.

DISADVANTAGES:

1. A lot of unutilized space.
2. If the binary tree is Full binary tree, there is No wastage of space, whereas, for Complete/skew trees there is a Wastage of space.
3. Insertion & deletions in a node, takes an excessive amount of processing time, due to data movement up & down in an array.

LINKED REPRESENTATION:

Linked representation uses linked lists to represent binary trees. Here empty sub trees are not separately stored as in linear representation. Hence, it improves utilization of memory in non complete binary trees. Each node in the binary tree is represented as node in linked list. An external pointer to the root node refers to the trees. Null (/) value represents that there is no child.

Lchild	Info	Rchild
--------	------	--------

Lchild - Pointer to the Left Child

Rchild - Pointer to the Right Child

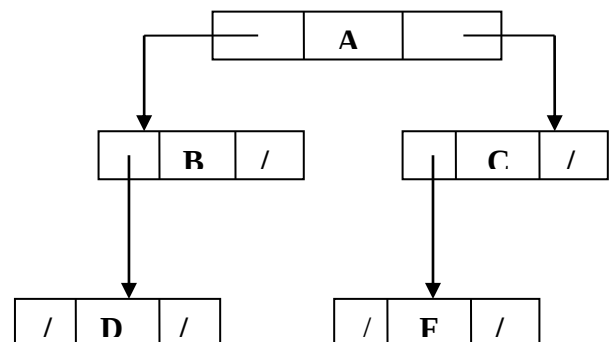
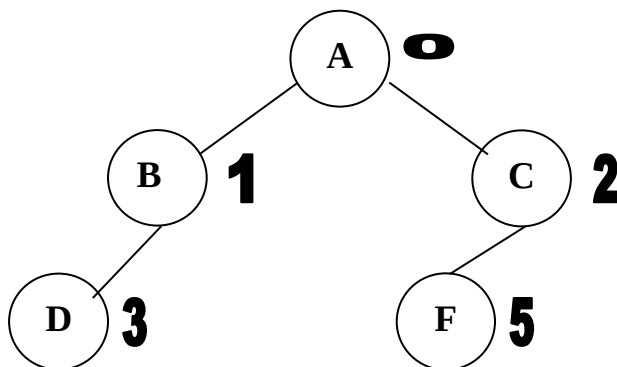
Info - The information at the node.

Advantage:

Insertion & deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

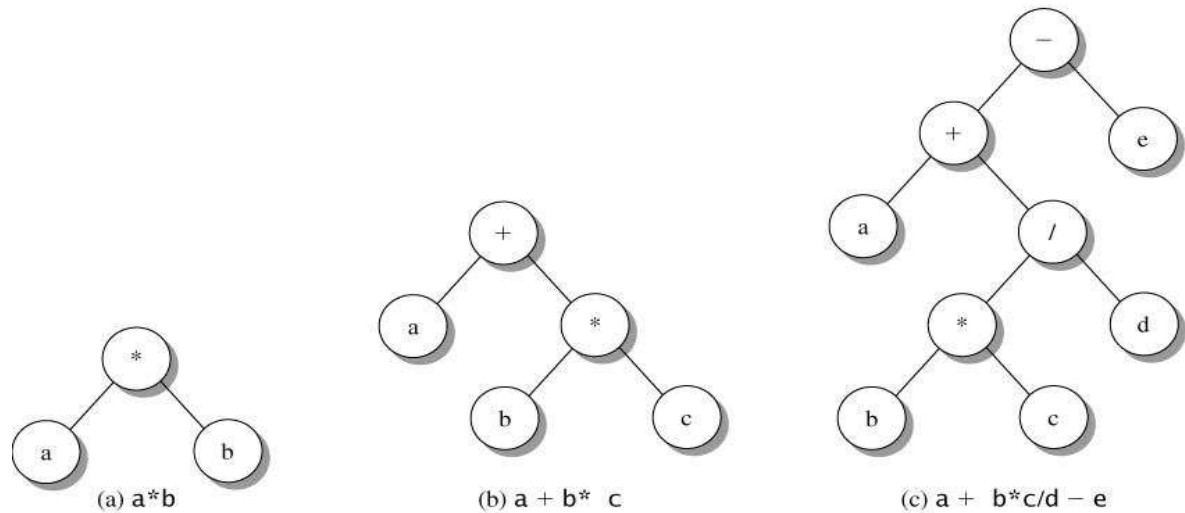
Disadvantage:

- 1) Given a node structure, it is difficult to determine its parent node.
- 2) Memory spaces are wasted for storing null pointers for the nodes, which have one or no sub trees.

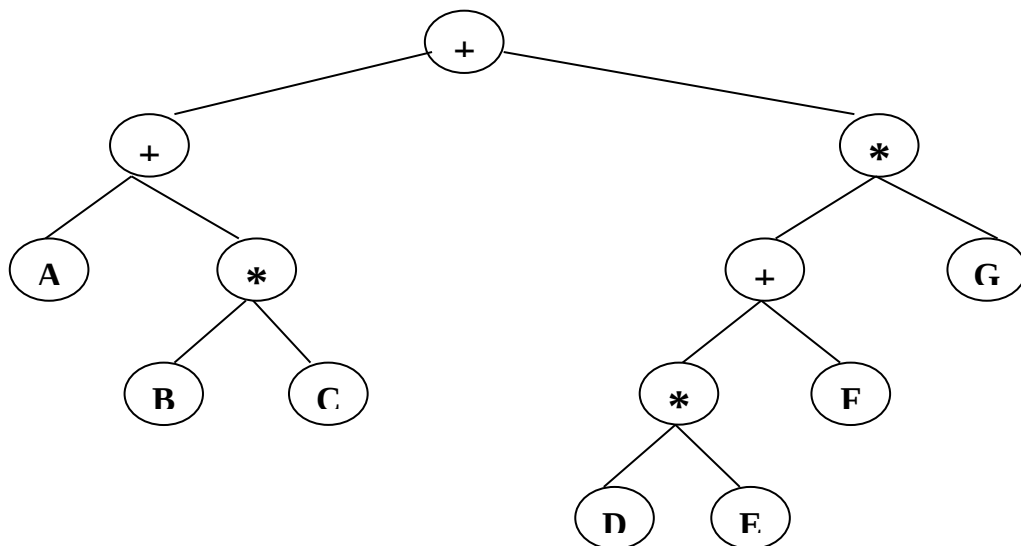


EXPRESSION TREES:

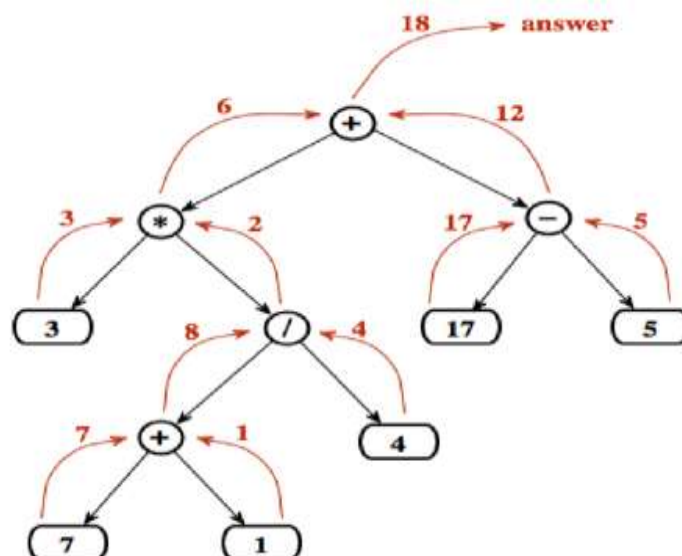
An expression can be represented using binary trees. Such trees are referred to as expression trees. The operands of the expression form the leaves of the tree & the operators form the interior nodes.



Expression Tree for the expression $(A + B * C) + ((D * E + F) * G)$



Example: $3 * ((7 + 1)/4) + (17-5)$



TREE TRAVERSALS:

1. Traversal refers to processing every node of the tree once & only once.
2. Traversing a binary tree means moving through all the nodes in the binary tree, visiting each node in the tree exactly once.

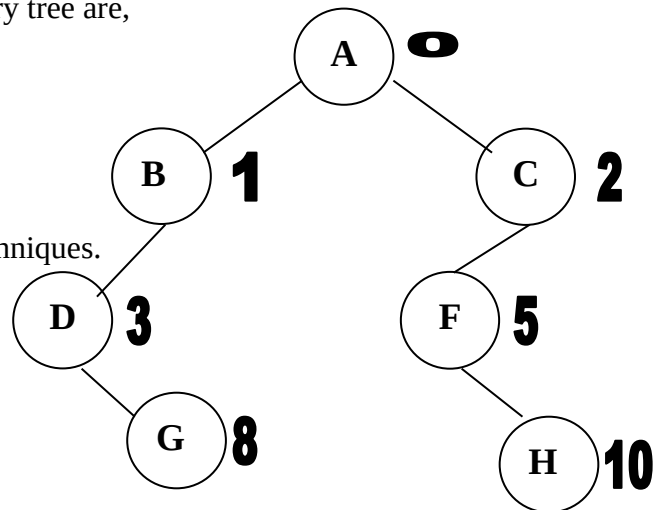
For a linked list, the nodes are in natural order from first to last and hence traversal follows the order. However, for the trees the nodes are not in order, hence we have to follow different procedure for traversing all the nodes. One thing to be kept in mind while traversing a binary tree is that treats each node & its sub trees in the same fashion.

The tasks performed while traversing a binary tree are,

1. Visiting a node.
2. Traversing the left sub tree.
3. Traversing the right sub tree.

There are 3 types of binary tree traversal techniques.

1. In order traversal (L P R)
2. Pre order traversal (P L R)
3. Post order traversal (L R P)

**INORDER TRAVERSAL:**

The steps for traversing a binary tree in inorder traversal are

- 1) Traverse the left sub tree.
- 2) Process the root node/parent node
- 3) Traverse the right sub tree.

In-Order Traversal for the given Tree manipulated as

For the root node: $B \rightarrow A \rightarrow C$

For the node B: $D \rightarrow B$

For the node C: $F \rightarrow C$

For the node D: $D \rightarrow G$

For the node F: $F \rightarrow H$

Replace the nodes B & C in the root node traversal $D \rightarrow B \rightarrow A \rightarrow F \rightarrow C$.

Then replace the nodes D & F from the above traversal as $D \rightarrow G \rightarrow B \rightarrow A \rightarrow F \rightarrow H \rightarrow C$

Therefore the in-order traversal for the given tree will be as

$D \rightarrow G \rightarrow B \rightarrow A \rightarrow F \rightarrow H \rightarrow C$

PREORDER TRAVERSAL:

The steps for traversing a binary tree in preorder traversal are

- 1) Process the root node.
- 2) Traverse the left sub tree.
- 3) Traverse the right sub tree.

Pre-Order Traversal for the given Tree manipulated as

For the root node: $A \rightarrow B \rightarrow C$

For the node B: $B \rightarrow D$

For the node C: $C \rightarrow F$

For the node D: $D \rightarrow G$

For the node F: $F \rightarrow H$

Replace the nodes B & C in the root node traversal $A \rightarrow B \rightarrow D \rightarrow C \rightarrow F$

Then replace the nodes D & F from the above traversal as $A \rightarrow B \rightarrow D \rightarrow G \rightarrow C \rightarrow F \rightarrow H$

Therefore the in-order traversal for the given tree will be as

$A \rightarrow B \rightarrow D \rightarrow G \rightarrow C \rightarrow F \rightarrow H$

POSTORDER TRAVERSAL:

The steps for traversing a binary tree in postorder traversal are

- 1) Traverse the left sub tree.
- 2) Traverse the right sub tree.
- 3) Process the root node/ parent node.

Post-Order Traversal for the given Tree manipulated as

For the root node: $B \rightarrow C \rightarrow A$

For the node B: $D \rightarrow B$

For the node C: $F \rightarrow C$

For the node D: $G \rightarrow D$

For the node F: $H \rightarrow F$

Replace the nodes B & C in the root node traversal $D \rightarrow B \rightarrow F \rightarrow C \rightarrow A$

Then replace the nodes D & F from the above traversal as $G \rightarrow D \rightarrow B \rightarrow H \rightarrow F \rightarrow C \rightarrow A$

Therefore the in-order traversal for the given tree will be as

$G \rightarrow D \rightarrow B \rightarrow H \rightarrow F \rightarrow C \rightarrow A$

LEVEL - ORDER TRAVERSAL/ BREADTH FIRST TRAVERSAL:

In level-order traversal, we traverse the nodes according to their levels. The steps for traversing a binary tree in level order traversal are:

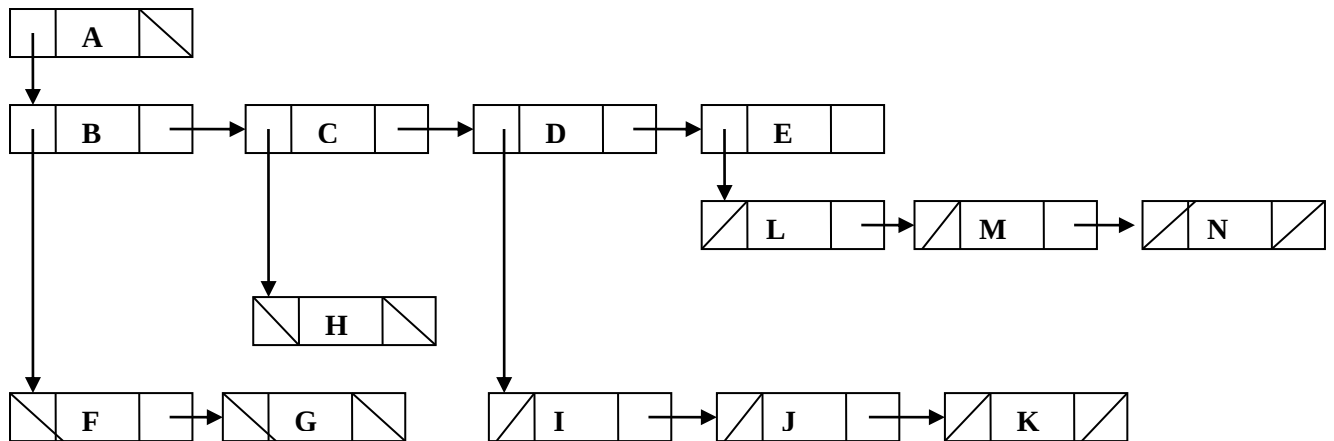
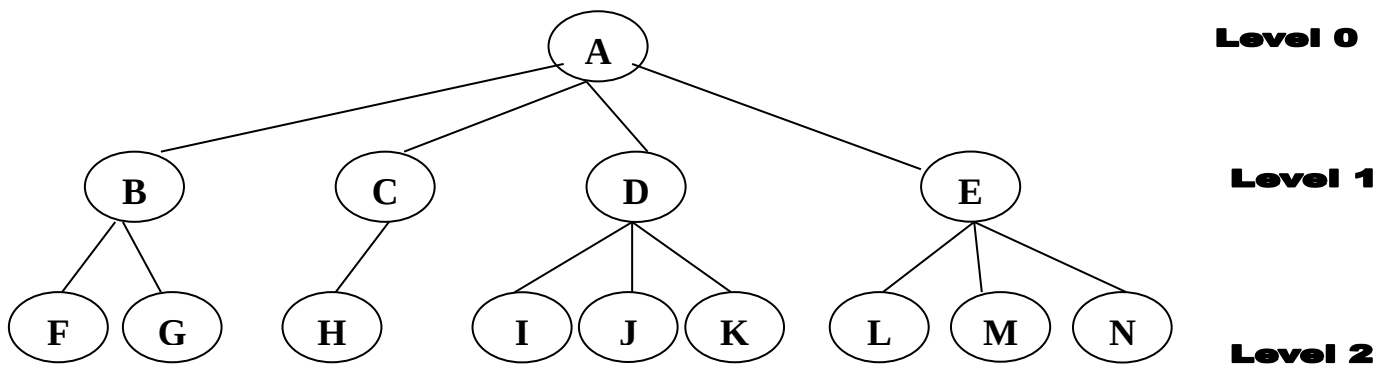
1. Process the root node at level 0.
2. Traverse the next level (i.e., level 1) below the root node.
3. Process the nodes from left to right in that level.
4. Similarly traverse the next level and process the nodes from left to right and continue till the end of levels.

Level – Order Traversal for the given tree manipulated as

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H$

CONVERSION OF GENERAL TREES TO BINARY TREES:

A general tree is a tree with nodes having any number of children. A general tree can be converted into an equivalent binary tree using the **leftmost child right siblings representation**. To convert a general tree to a binary tree using the leftmost child- right siblings relationship first connect the parent to its leftmost child & connect the leftmost child to all the right siblings at the same level within the same tree.



BINARY SEARCH TREES:

A binary tree where the value at root is greater than those in left sub tree and less than those in its right sub tree is called a **Binary search tree (BST)**. Both left & right sub trees also have this property of BST.

In BST, they should not have any duplicate values. If duplicate values are stored, they are referred as **BST with duplicates**. An **indexed BST** is one where every node has an index associated with it. Index represents the number of elements in its left sub tree. This could be used to identify its position in the sequence of elements in its sub tree.

PROPERTIES OF BST:

- 1) The elements in the Left Sub tree are less than the element in the root.
- 2) The elements in the Right Sub tree are greater than the element in the root.
- 3) Both left & right sub trees are in turn binary search trees.
- 4) The elements in BST should be unique i.e., they should not have any same value.

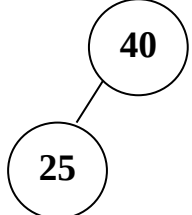
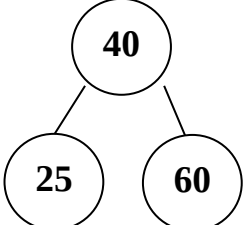
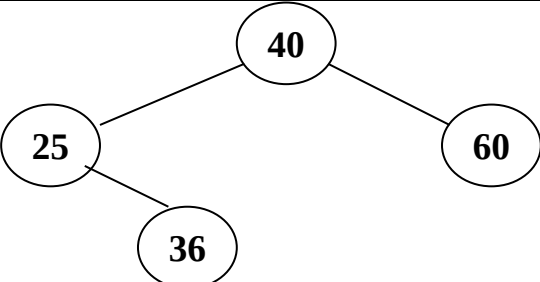
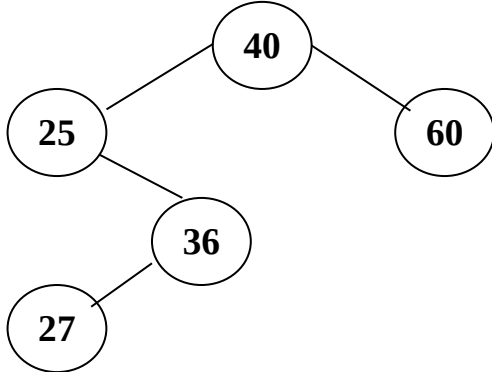
OPERATIONS ON BST:

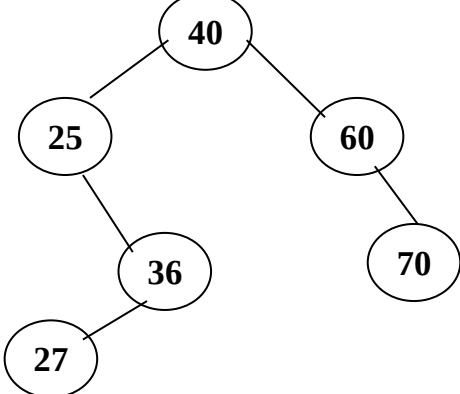
- 1) Creating a BST.
- 2) Inserting an element into BST.
- 3) Deleting an element from BST.
- 4) Searching for an element &
- 5) Printing elements of BST in ascending order.

Inserting An Element Into BST:

Inserting an element into a BST should not affect its ordering properties. The element key value should be compared with that of root. If it is less, then it should be added in the left sub tree, otherwise in the right sub tree. Similar comparison should be made in left or right sub tree. The new element added will always be at the leaf level.

Inserting elements 40, 25, 60, 36, 27, 70 one after the other.

Element Added	B S T	Comparisons
Insert 40		As 40 is the first element, it becomes the root.
Insert 25		Compare 40 and 25 As $25 < 40$, add it to the left sub tree. Since left sub tree was originally empty, 25 becomes the left child of 40.
Insert 60		Compare 40 and 60. As $60 > 40$, add it as right child(as its right subtree is empty).
Insert 36		Compare 40 and 36. As $36 < 40$, move to left sub tree and compare with its root 25. As $36 > 25$, add it to its right.
Insert 27		Compare 40 and 27. As $27 < 40$, move to left sub tree. Compare 27 and 25. As $27 > 25$, move to right. Compare 36 & 27. As $27 < 36$, add it to its right.

Insert 70	 <pre> graph TD 40((40)) --> 25((25)) 40 --> 60((60)) 25 --> 36((36)) 36 --> 27((27)) 60 --> 70((70)) </pre>	Compare 40 and 70. As $70 > 40$, move to right sub tree. Compare 60 and 70. As $70 > 60$, add it to right.
-----------	--	--

Insertion Procedure:

```
Void main ( )
```

```
{
```

```
int i, data [6] = {40, 25, 60, 36, 27, 70};
```

```
NODE *tree=NULL;
```

```
for (i=0; i<6; i++)
```

```
tree = insertBST (data[i], tree);
```

```
return;
```

```
}
```

```
Struct node
```

```
{
```

```
int info;
```

```
Struct node *leftchild;
```

```
Struct node *rightchild;
```

```
}
```

```
typedef struct node NODE;
```

```
NODE * insertBST (int elt, node *T)
```

```
{
```

```
if (T== NULL)
```

```
{
```

```
T= (node*) malloc (sizeof (node));
```

```
T→ info =elt;
```

```
T → leftchild= T→ rightchild = NULL;
```

```
}
```

```
else if (elt < T→ info)
```

```
T→ leftchild=insert BST (elt, T→leftchild);
```

```
else if (elt > T→ info)
```

```
T→ rightchild=insert BST (elt, T→rightchild);
```

```
return T;
```

```
}
```

Deleting an element into BST:

The node to be deleted can fall into any one of the following categories:

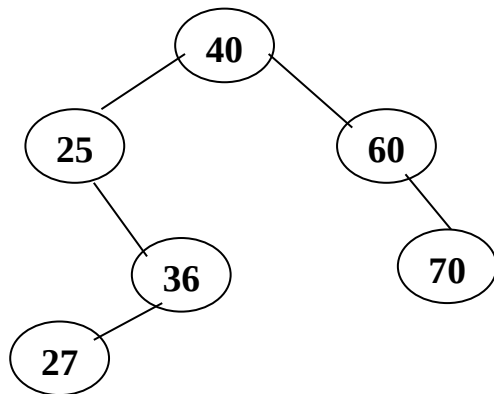
- 1) Node may not have any children (i.e., it is a leaf node)
- 2) Node may have only one child (either left/right child)
- 3) Node may have two children(both left right)

Deleting the Leaf node:

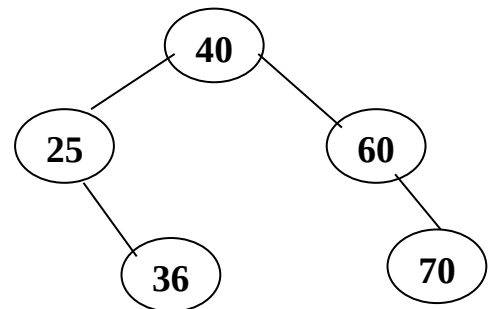
Steps followed to delete a leaf node from the binary search tree:

1. Search the parent of the leaf node and make the link to the leaf node as NULL.
2. Release the memory for the deleted mode.

For example if we want to delete the node 27, we can simply delete the node 27 by releasing the memory space allotted for that node



Before deleting the leaf node 27



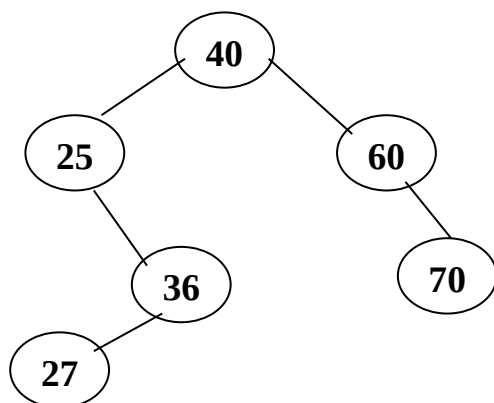
After deleting the leaf node 27

Deleting the node with only one child:

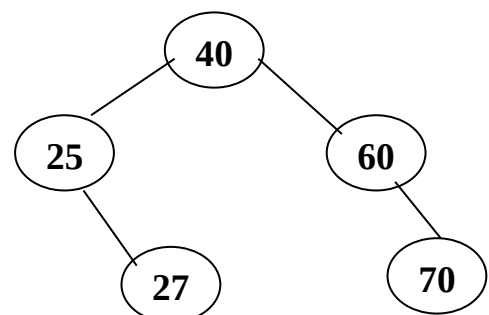
Steps followed to delete the node with only one child from the binary search tree:

1. Search the parent of the node to be deleted with only one child.
2. Assign the link of the parent node to the child of the node to be deleted.
3. Release the memory for the deleted mode.

For example if we want to delete the node 36 which has the child node as 27, we can delete the node 36 by replacing the node with its child 27 and then release the memory space allotted for that node



Before deleting the node 36



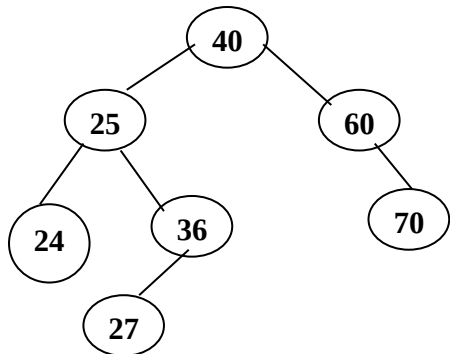
After deleting the node 36

Deleting the node with only two children:

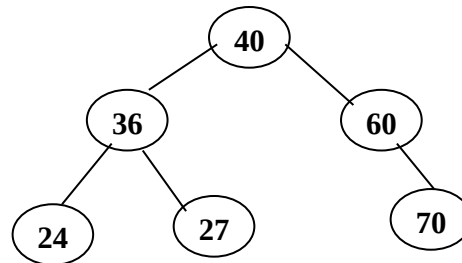
Steps followed to delete the node with only two children from the binary search tree:

1. Search the parent of the node to be deleted with two children.
2. Assign the link of the parent node to the child of the node to be deleted with its smallest value in the left sub tree or with the largest value in the right sub tree.
3. Release the memory for the deleted mode.

For example if we want to delete the node 25 which has the two children, we can delete the node 25 by replacing the node with its child 36 (which is greater in the right sub tree) and then release the memory space allotted for that node



Before deleting the node 25



After deleting the node 25

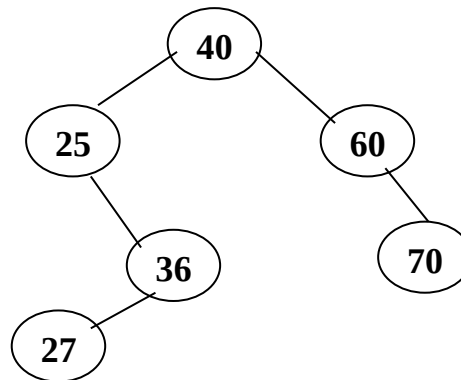
Procedure for Deleting a node from BST:

```

NODE *BSTDelete (int elt, NODE *tree)
{
    NODE *minelt;
    if (tree==NULL)
        printf ("element not found")
    else if (elt < tree->info)
        tree->leftchild = BSTdelete (elt, tree->leftchild)
    else if (elt > tree->info)
        tree->rightchild = BSTdelete (elt, tree->rightchild)
    else if (tree->leftchild && tree->rightchild != NULL)
    {
        minlet = Findmin (tree->rightchild);
        tree->info = minelt->info
        tree->rightchild = BSTdelete (tree->info, tree->rightchild);
    }
    else
    {
        minelt=tree;
        if (tree->leftchild == NULL)
            tree = tree->rightchild;
        else if (tree->rightchild == NULL)
            tree=tree->leftchild;
        free (minelt);
    }
    return tree;
}
  
```

Searching An Element In BST:

Searching starts from root of the tree. If the search key value is less than that in root, then the possible position of the element is in the left sub tree. If the search key value is greater than that in root, then it should be in right sub tree. This searching should continue till the node with the search key value or a null pointer is reached. In case a null pointer is reached, it is an indication of the absence of the node. One comparison is done at each level of the tree and at the most it can go up to the leaf level (the element can also be found in intermediate levels at which point the program will terminate).



To search 36

- 1) Compare 40 & 36
- 2) As $36 < 40$, move left & compare 25 & 36
- 3) As $36 > 25$, move right & compare 36 & 36
- 4) Match occurs. Hence stop.

To search 65,

- 1) Compare 40&65
- 2) As $65 > 40$, move right& compare 60&65
- 3) As $65 > 60$, move right& compare 65&70
- 4) As $65 > 70$, move left. Here left child is null, hence declare the element is not found in the tree.

Procedure for Searching an element:

```
NODE *SearchBST (int elt, NODE *tree)
```

```
{
    if (tree == NULL)
        printf ("Tree is empty")
    if (elt < tree->info)
        return searchBST (elt, tree → leftchild);
    else if (elt>tree->info)
        return searchBST (elt, tree → rightchild);
    else
        return tree;
}
```

Printing Elements of BST (Ordered Output):

The leftmost element of the leftmost branch is the minimum element and the rightmost element of the rightmost branch is the maximum. As the inorder traversal of a BST would give the elements of the BST in ascending order. Hence, ordered output algorithm is same as the inorder traversal algorithm.

```
Void inorder(NODE * Tree)
{
    if(Tree != NULL)
    {
        inorder(Tree → leftchild);
        printf("Tree → info);
        inorder(Tree → rightchild);
    }
    return;
}
```

Finding Minimum Element in A BST:

The minimum number lies at the leftmost node in the leftmost branch starting from the root. To reach the node with minimum value, we need to traverse the tree from the root along the left branch till we get a node with a null/ empty left sub tree.

```
NODE *Findmin(NODE * Tree)
{
    if(Tree == NULL)
        return NULL;
    else if(Tree → leftchild ==NULL)
        return Tree;
    else
        return Findmin(tree → leftchild);
}
```

Finding Maximum Element In A BST:

The maximum number lies at the rightmost node in the rightmost branch starting from the root. To reach the node with maximum value, we need to traverse the tree from the root along the right branch till we get a node with a null/ empty right sub tree.

```
NODE *Findmax(NODE * Tree)
{
    if(Tree == NULL)
        return NULL;
    else if(Tree → rightchild ==NULL)
        return Tree;
    else
        return Findmax(tree → rightchild);
}
```

HUFFMAN CODING ALGORITHM

- Every information in computer science is **encoded** as strings of **1s and 0s**.
- The objective of information theory is to usually transmit information using fewest number of bits in such a way that every encoding is unambiguous.
- Encoding, in computers, can be defined as the process of transmitting or storing sequence of characters efficiently.
- Fixed-length and variable length are two types of encoding schemes, explained as follows-
- ✓ **Fixed-Length encoding** - Every character is assigned a binary code using same number of bits. Thus, a string like “aabacdad” can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.
- ✓ **Variable- Length encoding** - As opposed to Fixed-length encoding, this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text. Thus, for a given string like “aabacdad”, frequency of characters ‘a’, ‘b’, ‘c’ and ‘d’ is 4,1,1 and 2 respectively. Since ‘a’ occurs more frequently than ‘b’, ‘c’ and ‘d’, it uses least number of bits, followed by ‘d’, ‘b’ and ‘c’. Suppose we randomly assign binary codes to each character as follows-

a 0**b 011****c 111****d 11**

Thus, the string “aabacdad” gets encoded to

00011011111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11),

using fewer number of bits compared to fixed-length encoding scheme.

But the real problem lies with the decoding phase. If we try and decode the string 00011011111011, it will be quite ambiguous since, it can be decoded to the multiple strings, few of which are-

aaadacdad (0 | 0 | 0 | 11 | 0 | 111 | 11 | 0 | 11)**aaadbcbad (0 | 0 | 0 | 11 | 011 | 111 | 0 | 11)****aabbcb (0 | 0 | 011 | 011 | 111 | 011)**

... and so on

To prevent such ambiguities during decoding, the encoding phase should satisfy the “**prefix rule**” which states that no binary code should be a prefix of another code. This will produce uniquely **decodable codes**. The above codes for ‘a’, ‘b’, ‘c’ and ‘d’ do not follow prefix rule since the binary code for a, i.e. 0, is a prefix of binary code for b i.e 011, resulting in ambiguous **decodable codes**.

Lets reconsider assigning the binary codes to characters 'a', 'b', 'c' and 'd'.

a 0

b 11

c 101

d 100

Using the above codes, string “**aabacdad**” gets encoded to
001101011000100 (0 | 0 | 11 | 0 | 101 | 100 | 0 | 100).

Now, we can decode it back to string “**aabacdad**”.

Problem Statement-

Input: Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights

Output: Prefix-free and variable-length binary codes with minimum expected codeword length. Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

Huffman Encoding-

Huffman Encoding can be used for finding solution to the given problem statement.

- Developed by **David Huffman** in 1951, this technique is the basis for all data compression and encoding schemes
- It is a famous algorithm used for lossless data encoding
- It follows a Greedy approach, since it deals with generating minimum length prefix-free binary codes
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code

The major steps involved in Huffman coding are-

Step I - Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol

- A Huffman tree, similar to a binary tree data structure, needs to be created having **n** leaf nodes and **n-1** internal nodes
- Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.
- Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character

- Internal nodes, on the other hand, contain weight and links to two child nodes

Step II - Assigning the binary codes to each symbol by traversing Huffman tree

- Generally, bit '0' represents the left child and bit '1' represents the right child

Algorithm for creating the Huffman Tree-

Step 1- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Step 2- Repeat Steps 3 to 5 while heap has more than one node

Step 3- Extract two nodes, say x and y, with minimum frequency from the heap

Step 4- Create a new internal node z with x as its left child and y as its right child.

Also $\text{frequency}(z) = \text{frequency}(x) + \text{frequency}(y)$

Step 5- Add z to min heap

Step 6- Last node in the heap is the root of Huffman tree

Applications of Huffman Tree:

1. File Compression algorithm
2. Transmission of data in an encode form

Procedure:

1. Take the characters and their frequencies, and sort this list by **increasing** frequency
2. All the characters are vertices of the tree
3. Take the first 2 vertices from the list and make them children of a vertex having the sum of their frequencies
4. Insert the new vertex into the sorted list of vertices waiting to be put into the tree
5. If there are at least 2 vertices in the list, go to step 3.
6. Read the Huffman code from the tree

Algorithm:

```

algorithm Huffman( $W[0..n-1]$ )
    // Computes optimal prefix code.
    // Input: Array W of character probabilities
    // Output: The Huffman tree.
    for  $i \leftarrow 0$  to  $n-1$  do
         $T \leftarrow$  node labeled with character  $i$ 
         $T.\text{weight} \leftarrow W[i]$ ; add  $T$  to set  $S$ 
    for  $i \leftarrow 0$  to  $n-2$  do
         $L, R \leftarrow$  remove two min. weight trees from  $S$ 
         $T \leftarrow$  node with children  $L$  and  $R$ 
         $T.\text{weight} \leftarrow L.\text{weight} + R.\text{weight}$ 
        add  $T$  to set  $S$ 
    return  $T$ 
  
```

Time complexity of Huffman Tree is $O(n \log n)$.

Example:

Consider the five alphabets with following occurrence probabilities.

Character	A	B	C	D	E
Probability	0.40	0.1	0.25	0.2	0.15

Solution:

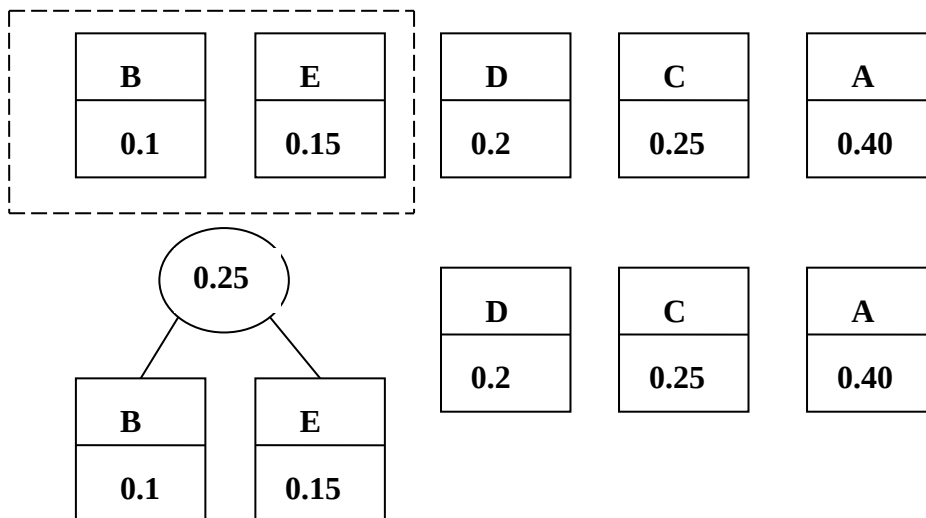
1. Take the characters and their frequencies, and *sort* this list by **increasing** frequency

Character	B	E	D	C	A
Probability	0.1	0.15	0.2	0.25	0.40

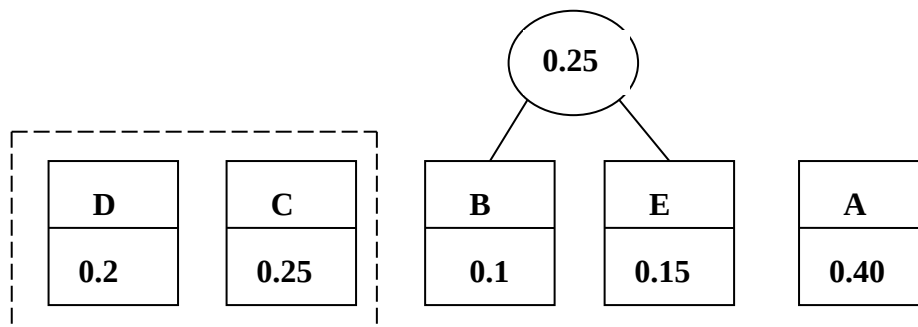
2. All the characters are vertices of the tree

B	E	D	C	A
0.1	0.15	0.2	0.25	0.40

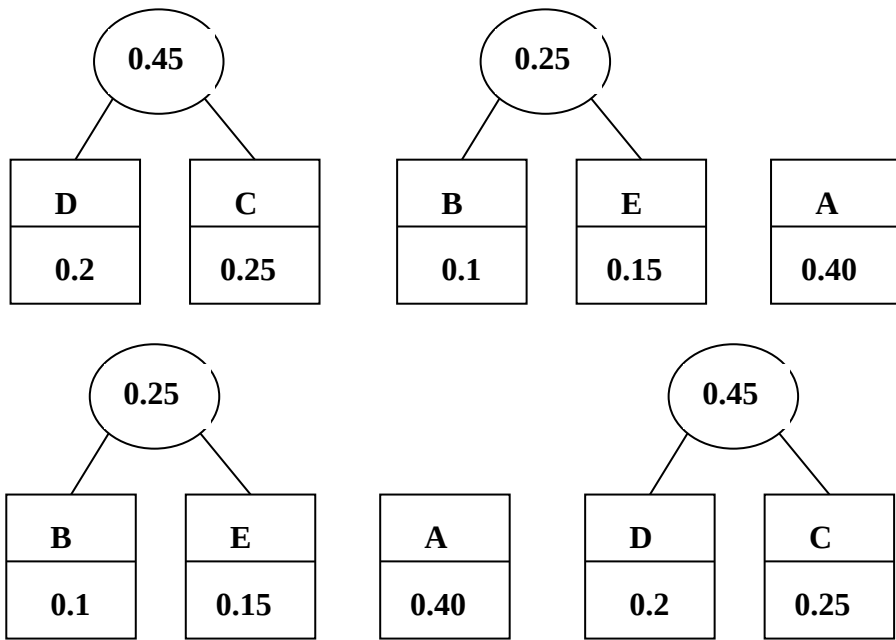
3. Take the first 2 vertices from the list and make them children of a vertex having the sum of their frequencies



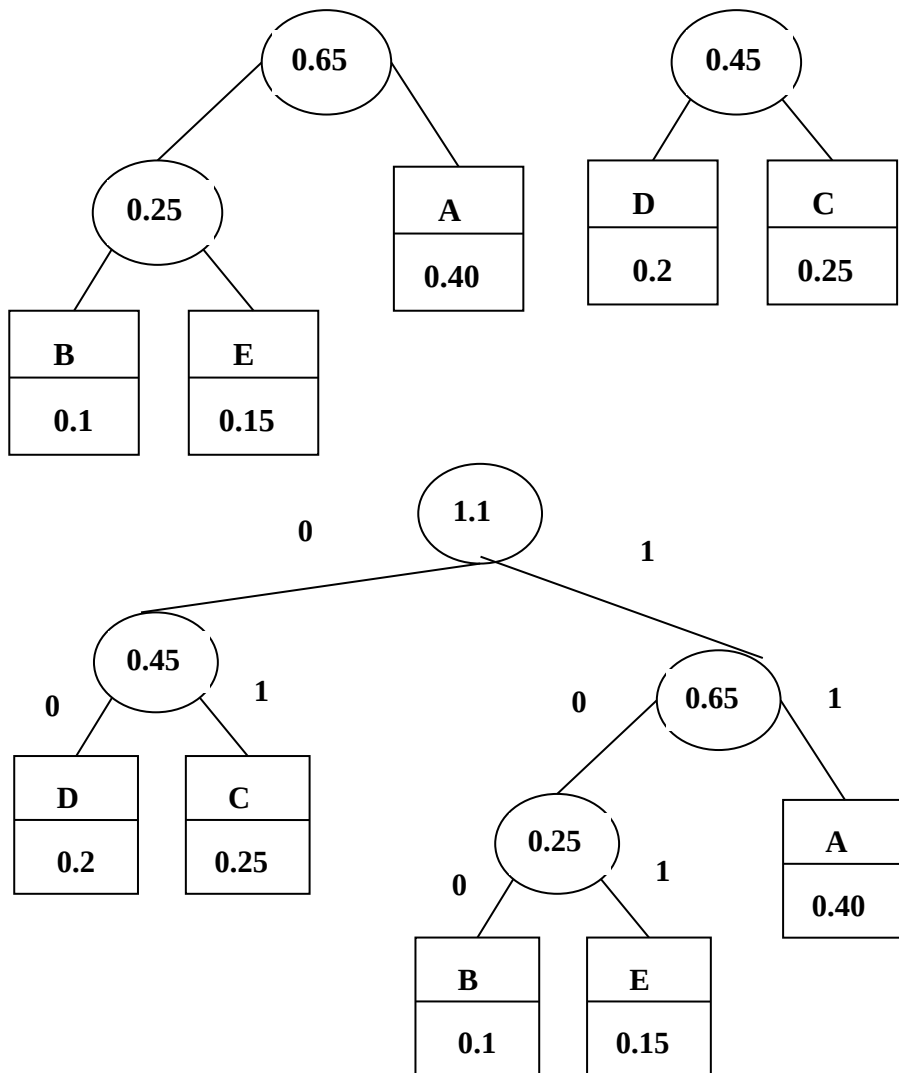
4. Insert the new vertex into the sorted list of vertices waiting to be put into the tree.



5. If there are at least 2 vertices in the list and make them children of a vertex having the sum of their frequencies. Sort the list of vertices.



6.



7. Read the Huffman code from the tree

Thus the encoding can be

A	B	C	D	E
11	100	01	00	101

EXAMPLE

Let's try and create Huffman Tree for the following characters along with their frequencies using the above algorithm-

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Step A- Create leaf nodes for all the characters and add them to the min heap.

- **Step 1-** Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

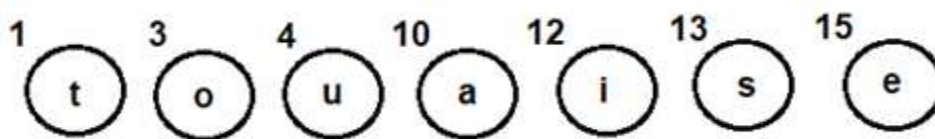


Fig 1: Leaf nodes for each

Step B- Repeat the following steps till heap has more than one nodes

- **Step 3-** Extract two nodes, say x and y, with minimum frequency from the heap
 - **Step 4-** Create a new internal node z with x as its left child and y as its right child. Also $\text{frequency}(z) = \text{frequency}(x) + \text{frequency}(y)$
 - **Step 5-** Add z to min heap
- Extract and Combine node u with an internal node having 4 as the frequency
 - Add the new internal node to priority queue-

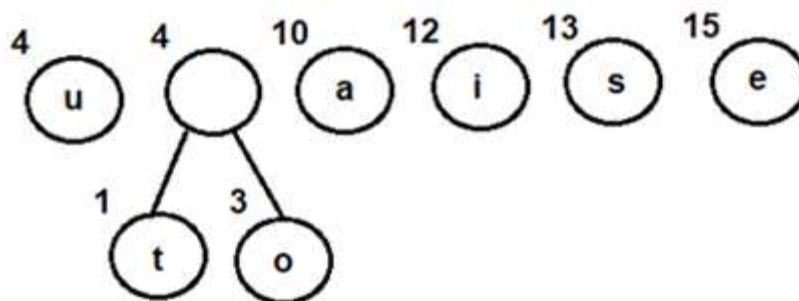


Fig 2: Combining nodes o and t

- Extract and Combine node a with an internal node having 8 as the frequency
- Add the new internal node to priority queue-

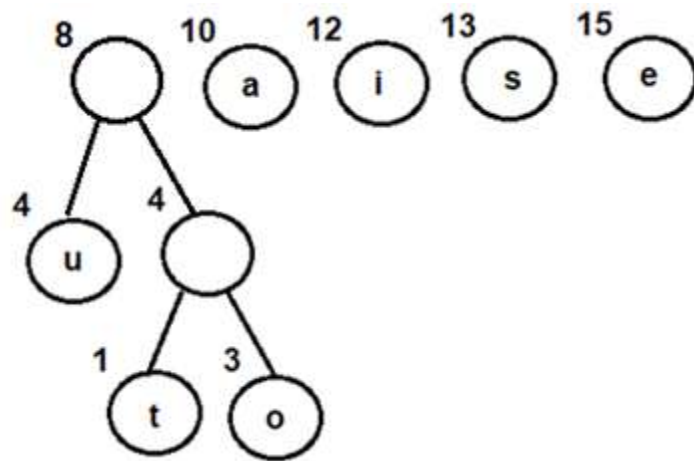


Fig 3: Combining node u with an internal node having 4 as frequency

- i. Extract and Combine nodes i and s
- ii. Add the new internal node to priority queue-

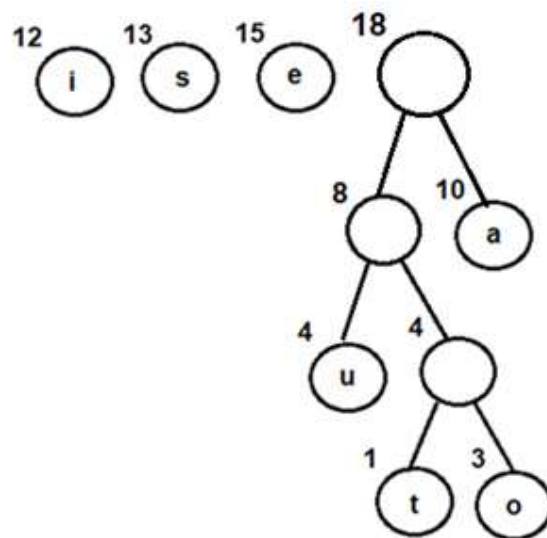


Fig 4: Combining node u with an internal node having 4 as frequency

- i. Extract and Combine nodes i and s
- ii. Add the new internal node to priority queue-

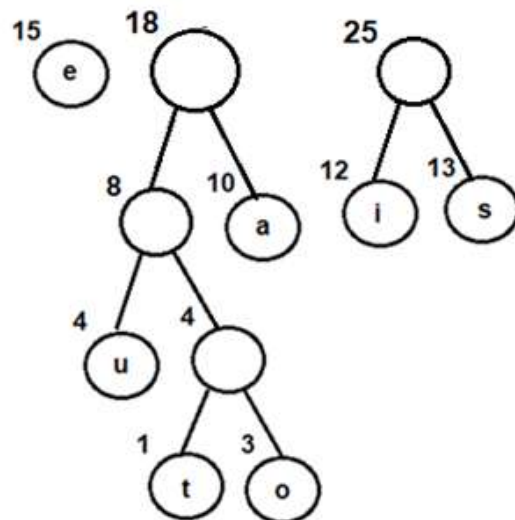


Fig 5: Combining nodes i and s

- i. Extract and Combine node with an internal node having 18 as the frequency
- ii. Add the new internal node to priority queue-

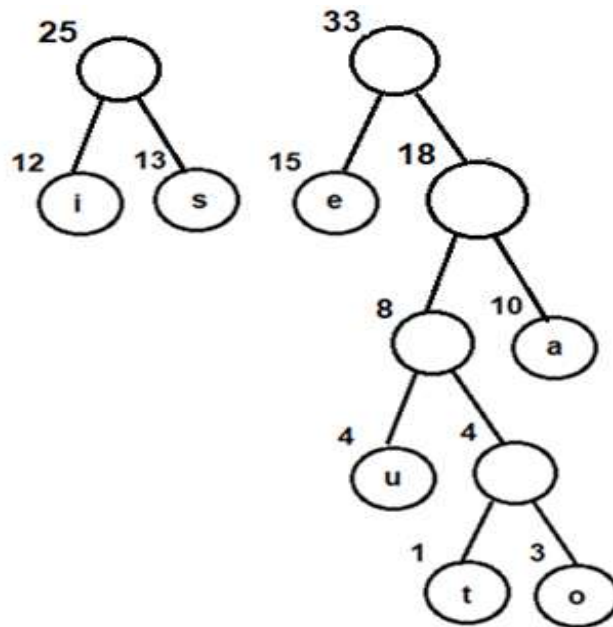


Fig 6: Combining node e with an internal node having 18 as frequency

- i. Finally, Extract and Combine internal nodes having 25 and 33 as the frequency
- ii. Add the new internal node to priority queue-

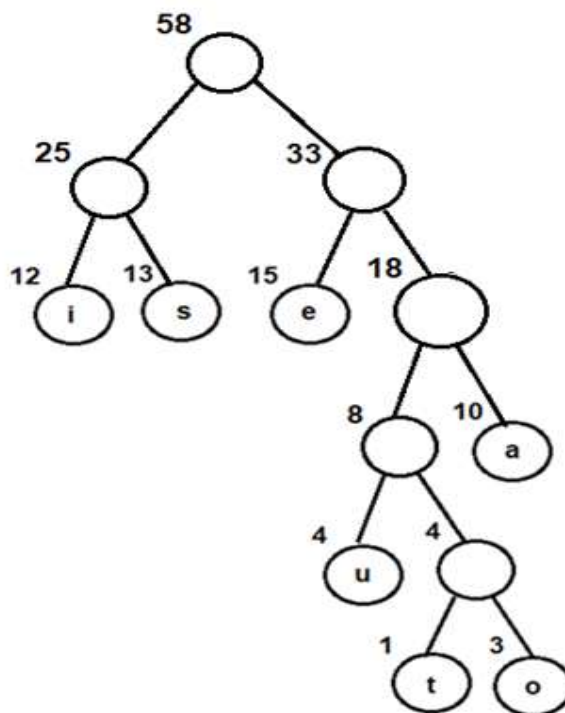


Fig 7: Final Huffman tree obtained by combining internal nodes having 25 and 33 as frequency

Now, since we have only one node in the queue, the control will exit out of the loop

Step C- Since internal node with frequency 58 is the only node in the queue, it becomes the root of **Huffman tree**.

Step 6- Last node in the heap is the root of Huffman tree

Steps for traversing the Huffman Tree

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Add 0 to array while traversing the left child and add 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

Following the above steps for Huffman Tree generated above, we get prefix-free and variable-length binary codes with minimum expected codeword length-

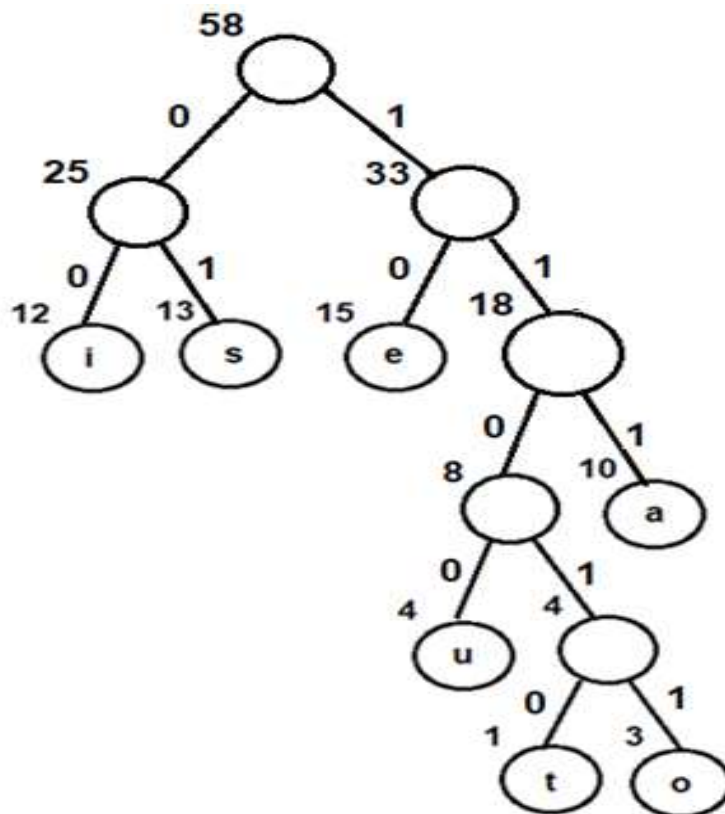


Fig 8: Assigning binary codes to Huffman tree

Characters	Binary Codes
i	00
s	01
e	10
u	1100
t	11010
o	11011
a	111

Using the above binary codes-

Suppose the string “staeiou” needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding, the string gets encoded to “0111010111100011011110011010”

(01 | 11010 | 111 | 10 | 00 | 11011 | 1100 | 11010) at the sender side.

Once received at the receiver's side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the string. A '1' or '0' in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node.

Thus for the above bit stream



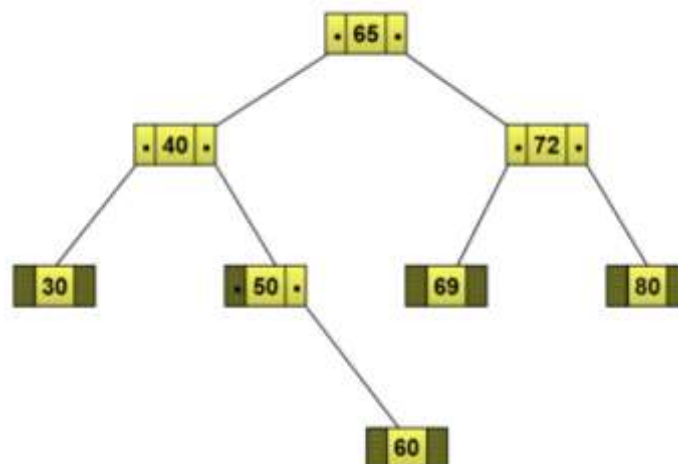
Fig 9: Decoding the bit stream

On similar lines-

- 111 gets decoded to 'a'
- 10 gets decoded to 'e'
- 00 gets decoded to 'i'
- 11011 gets decoded to 'o'
- 1100 gets decoded to 'u'
- And finally, 11010 gets decoded to 't', thus returning the string "staeiout" back

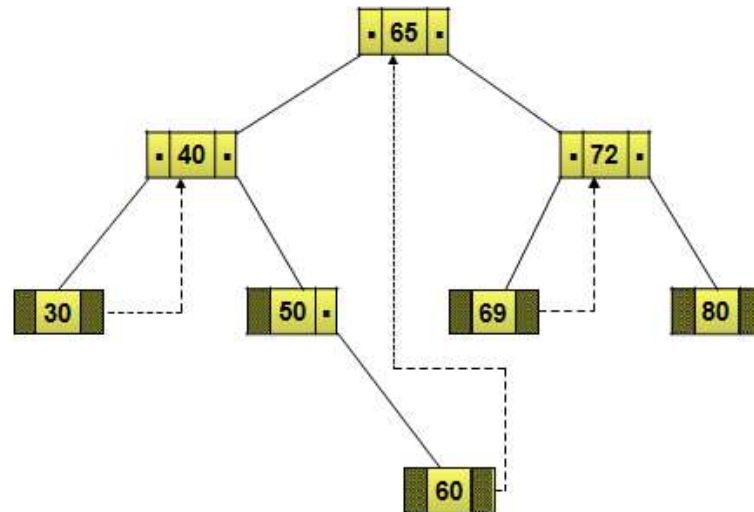
THREADED BINARY TREES:

- In a binary search tree, there are many nodes that have an empty left child or empty right child or both.

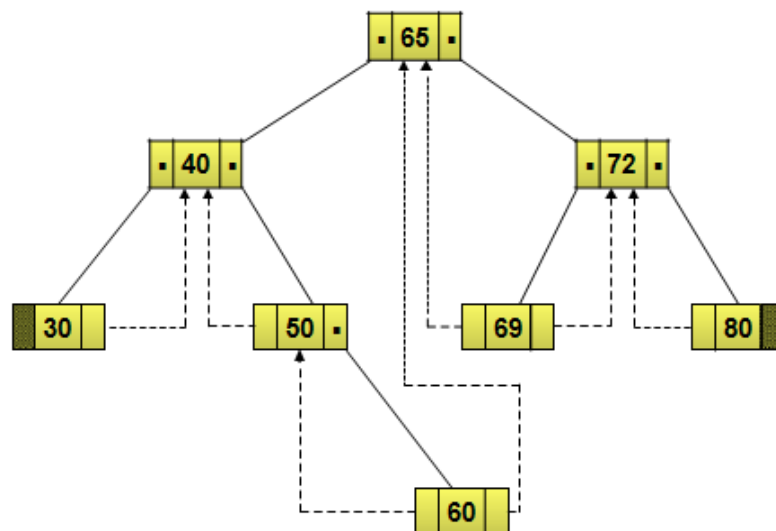


- We can utilize these fields in such a way so that the empty left child of a node points to its inorder predecessor and empty right child of the node points to its inorder successor.
- Types of Threaded Binary Tree:
 - ✓ **One way threading:** A thread will appear in a right field of a node and will point to the next node in the inorder traversal.

- ✓ **Two way threading:** A thread will also appear in the left field of a node and will point to the preceding node in the inorder traversal.
- **One Way Threaded Binary Trees**
 - ✓ The empty left child field of a node can be used to point to its inorder predecessor.
 - ✓ Similarly, the empty right child field of a node can be used to point to its in-order successor.
 - ✓ Such a type of binary tree is known as a one way threaded binary tree.
 - ✓ A field that holds the address of its in-order successor is known as thread.
 - ✓ In-order :- $30 \rightarrow 40 \rightarrow 50 \rightarrow 60 \rightarrow 65 \rightarrow 69 \rightarrow 72 \rightarrow 80$

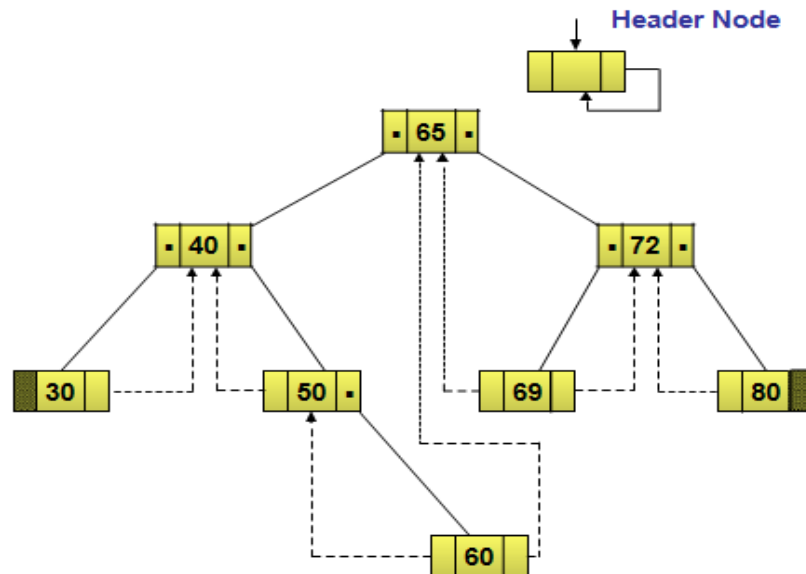


- **Two way Threaded Binary Trees**
 - ✓ Such a type of binary tree is known as a threaded binary tree.
 - ✓ A field that holds the address of its inorder successor or predecessor is known as thread.
 - ✓ The empty left child field of a node can be used to point to its inorder predecessor.
 - ✓ Similarly, the empty right child field of a node can be used to point to its inorder successor.
 - ✓ Inorder :- $30 \rightarrow 40 \rightarrow 50 \rightarrow 60 \rightarrow 65 \rightarrow 69 \rightarrow 72 \rightarrow 80$

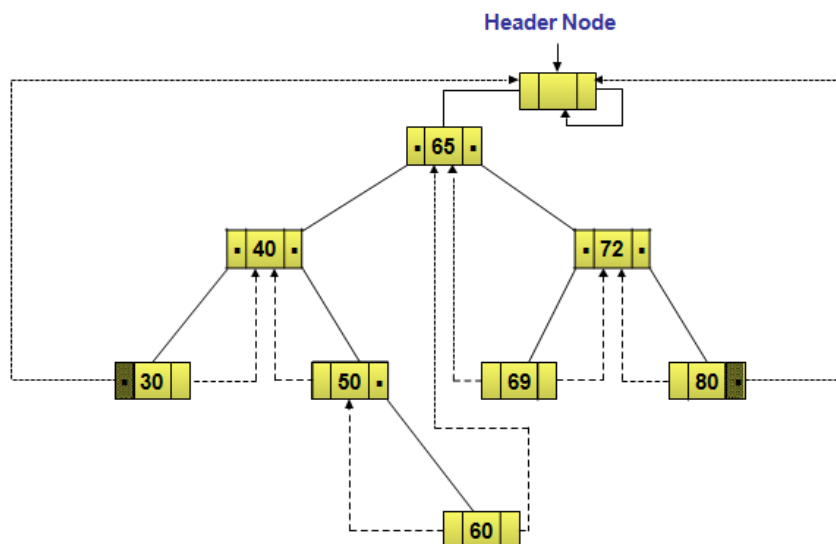


- ✓ Node 30 does not have an inorder predecessor because it is the first node to be traversed in inorder sequence.
- ✓ Similarly, node 80 does not have an inorder successor.

- ✓ Two-way threading has left pointer of the first node and right pointer of the last node will contain the null value. This is known as Dangling.
- **Two way Threaded Binary Trees with header Node**
- ✓ The right child of the header node always points to itself.
- ✓ Therefore, you take a dummy node called the header node.

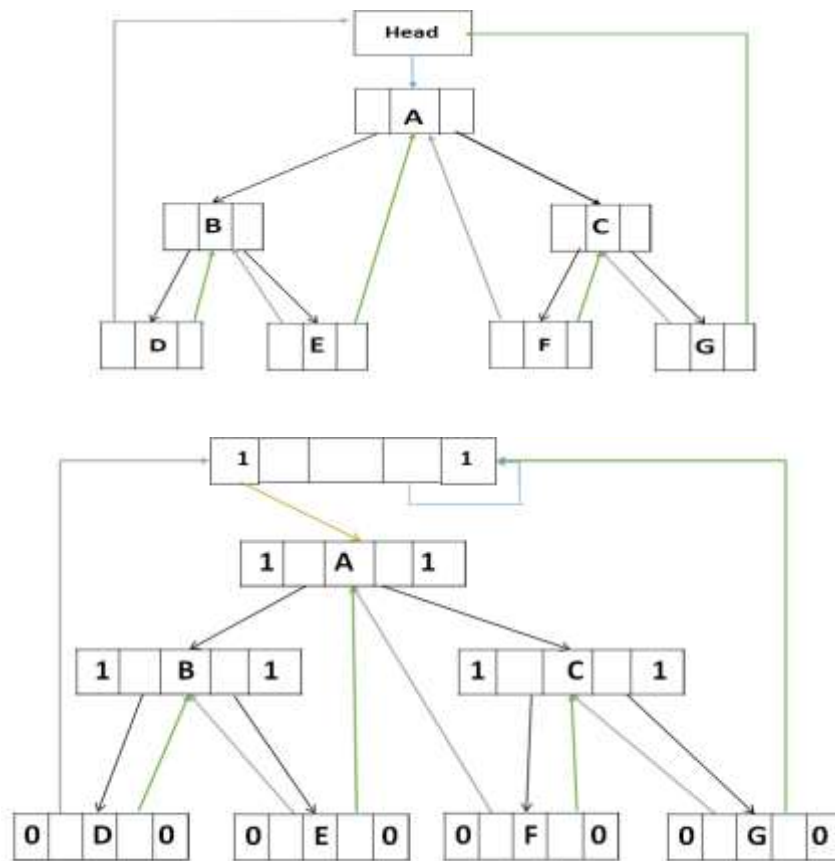


- ✓ The threaded binary tree is represented as the left child of the header node.
- ✓ The left thread of node 30 and the right thread of node 80 point to the header node.



Representing a Threaded Binary Tree

- ✓ The structure of a node in a threaded binary tree is a bit different from that of a normal binary tree.
- ✓ Unlike a normal binary tree, each node of a threaded binary tree contains two extra pieces of information, namely left thread and right thread.
- ✓ The left and right thread fields of a node can have two values:
- ✓ **1:** Indicates a normal link to the child node
- ✓ **0:** Indicates a thread pointing to the inorder predecessor or inorder successor



AVL TREES:

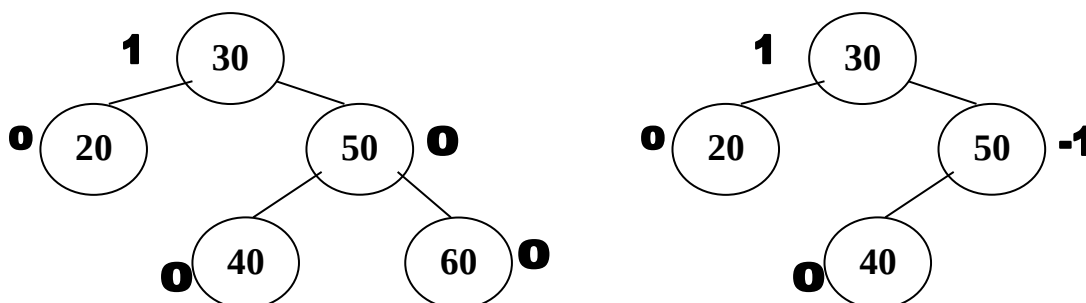
AVL trees are named after the inventor's **Adelson, Velskii & Landis**. An AVL tree is a self adjusting, height balanced BST where the difference in heights of the Left & Right sub trees is a maximum of 1, i.e., every node in a tree is height balanced (difference in heights of the left & right sub trees is at most 1). These trees are also referred to as **admissible trees**.

The **balance factor** of a node is defined as the height of the R sub tree minus the height of the L sub tree. Hence in an AVL tree, the balance factor of each node is +1, 0, or -1.

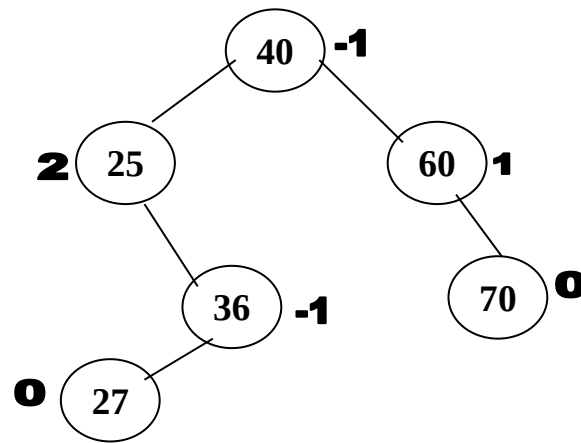
$$\text{Balance Factor} = h_r - h_l$$

Where $h_r - h_l$ denote the height of the Left & Right sub trees respectively.

A balance factor of 0 indicates that both sub trees are of the same height. A balance factor of +1 indicates the height of the right sub tree is 1 more than the height of the left sub tree. Such a sub tree is said to be **right heavy**. Similarly, a sub tree is **left heavy** if its root has a balance factor of -1. The balance factor of each node is given adjacent to the node.



EXAMPLES OF AVL TREES



EXAMPLE OF NON-AVL TREES

Operations:

In an AVL tree, during insertion or deletion of a node, the nodes in the path from the insertion / deletion point to the root may become unbalanced. In such cases, the structure of the tree should be slightly modified to restore the balance of the nodes. Such modification of the structure is called rotation. A node becomes unbalanced if the left / right sub tree grows longer than the left / right sub tree by 2 levels i.e., $h_r - h_l > 1$. Rotation tries to shorten the sub tree that has grown longer to make $h_r - h_l$ either 0 or 1.

Rotation:

In an AVL Tree, during insertion or deletion of a node, the nodes in the path from the insertion / deletion point to the root may become unbalanced. In such cases, the structure of the tree should be slightly modified to restore the balance of the nodes. Such modification of the structure is called as **Rotation**. Inserting a node on the `outside` of the tree, i.e., in the Left of the Left sub tree or in the Right of the Right sub tree. The unbalance caused by such insertions can be fixed by a **Single Rotation** of the tree. Inserting a node on the `inside` of the tree, in the Left of the Right sub tree or in the Right of the Left sub tree. Unbalance of nodes by such insertions can be handled by a **Double Rotation** of the tree.

There are four types of rotations used in rebalancing an AVL Tree:

1. Single Left Rotation (SLR)
2. Single Right Rotation (SRR)
3. Double Left Rotation (DLR)
4. Double Right Rotation (DRR)

Single Right Rotation (SRR):

This rotation is performed when the tree is unbalanced to the left i.e., when the left sub tree is taller (Left heavy). The SRR moves a node from the left to the right sub tree, making all the nodes balanced.

Single Left Rotation (SLR):

This rotation is performed when the tree is unbalanced to the right i.e., when the right sub tree is taller (Right heavy). The SLR moves a node from the right to the left sub tree, making all the nodes balanced.

Double Left Rotation (DLR):

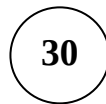
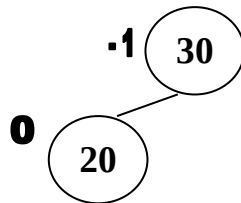
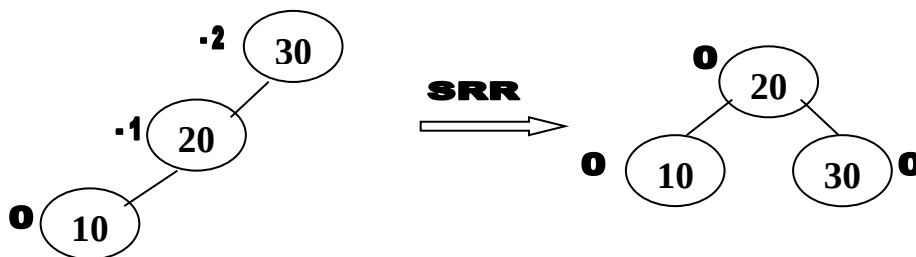
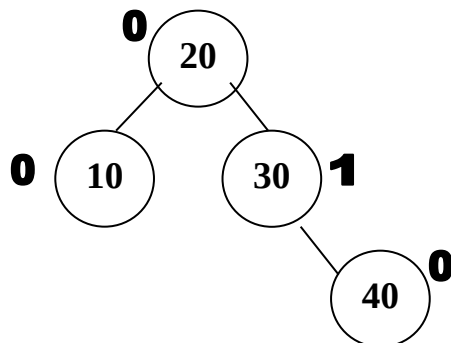
This rotation is performed when a node is added to the left of the right sub tree of a node. This is equivalent to an SLR followed by an SRR.

Double Right Rotation (DRR):

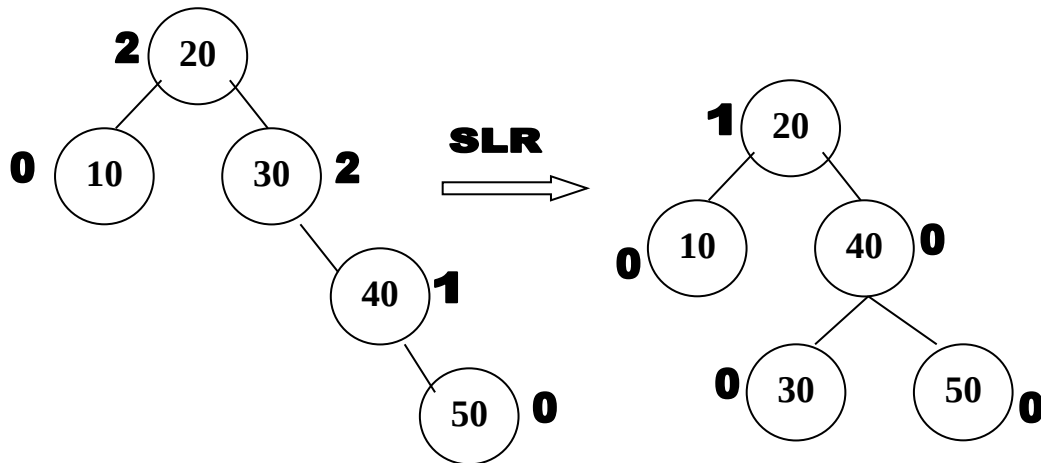
This rotation is performed when a node is added to the right of the left sub tree of a node. This is equivalent to an SRR followed by an SLR.

Illustration of balancing of nodes in an AVL Tree:

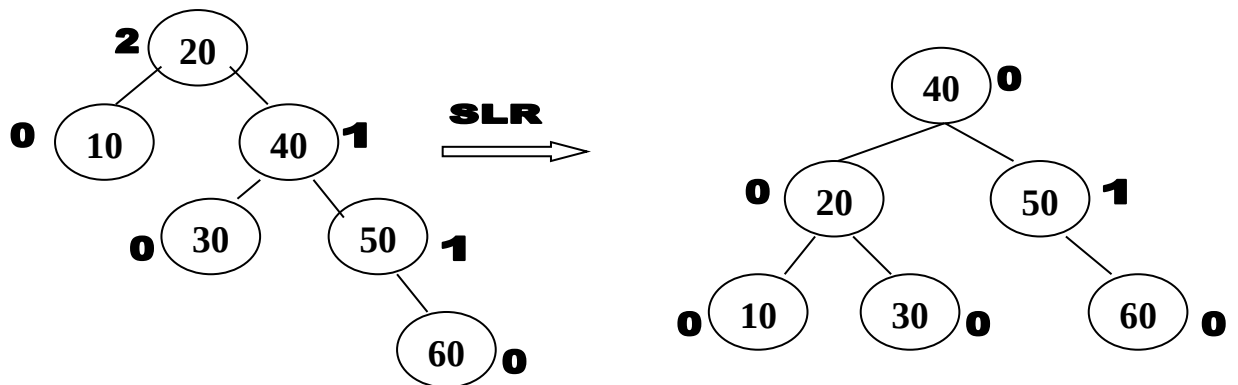
Consider creating an AVL tree, starting with an initially empty tree with elements 30, 20, 10, 40, 50, 60 and 25 inorder.

Step 1: Insert 30**Step 2: Insert 20****Step 3: Insert 10.** While inserting 10, node 30 unbalanced.**Step 4: Insert 40.**

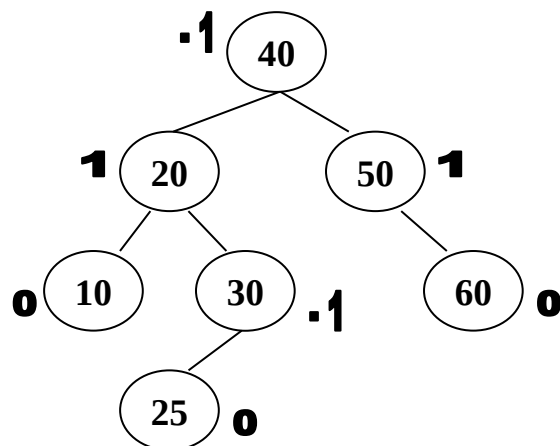
Step 5: Insert 50. While inserting 50, node 30 unbalanced.



Step 6: Insert 60. While inserting 60, node 20 unbalanced.



Step 7: Insert 25.



Deletion of a Node:

Deletion of a node from an AVL Tree involves 2 steps:

1. Search the tree and delete the node, similar to binary search tree.
2. Check whether all nodes are balanced. If not, balance them using appropriate rotations.

BINARY HEAP:

A binary heap is an array that is viewed as a complete binary tree. A complete binary tree is completely filled on all levels except possibly the lowest, and the lowest level is filled from the left. The root of the tree is stored in array element 0. The parent of the node n is stored at node $(n / 2)$. The right child of node is stored at node $2n$ and the left child at node is stored at $2n + 1$.

Properties of Heap:Structure Property:

A heap is an almost complete binary tree with last level filled from the left, i.e., all leaves are as far as possible to the left.

Order Property:

Max-Heap Property: If the value of the node is greater than its children, it is referred as Max-Heap.

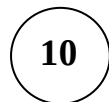
Min-Heap Property: If the value of the node is smaller than its children, it is referred as Min-Heap.

Representation of Heap:

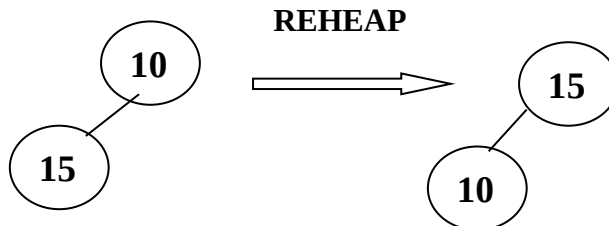
Since heaps are almost complete binary trees, array representation will be efficient (as there will be no empty sub trees in the intermediate levels). In array representation, the right child of node is stored at node $2n$ and the left child at node is stored at $2n + 1$.

Heap tree for the following numbers are illustrated below: 10, 15, 12, 25, 31.

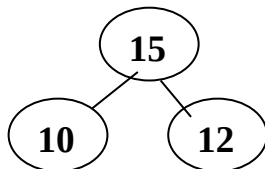
a) Insert 10



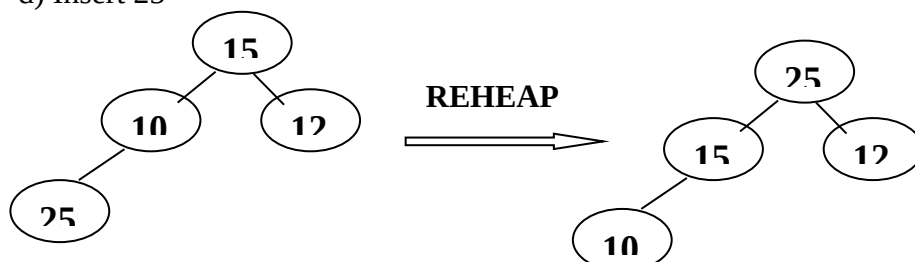
b) Insert 15



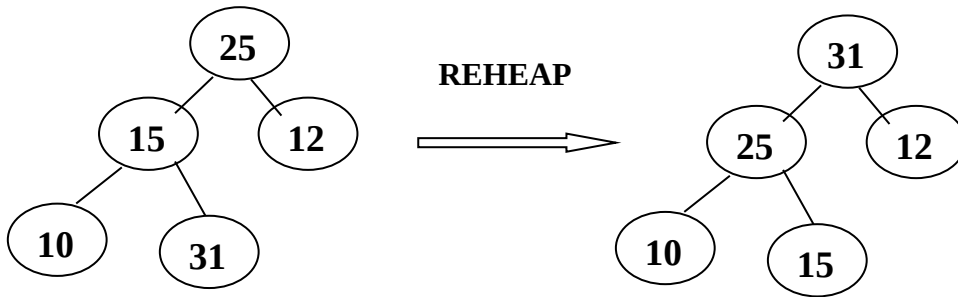
c) Insert 12



d) Insert 25



e) Insert 31



The array representation of the Heap will be as

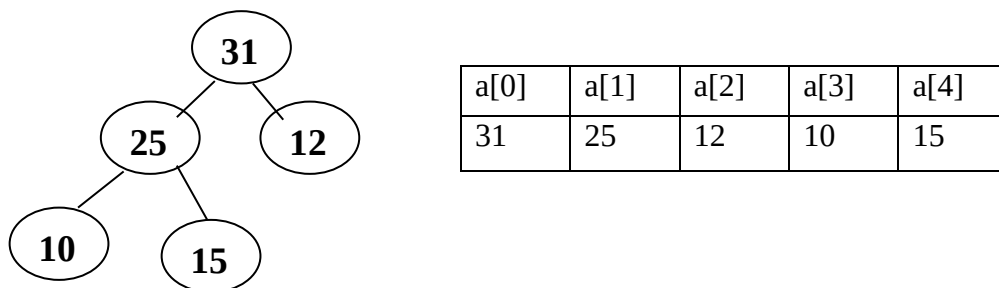
a[0]	a[1]	a[2]	a[3]	a[4]
31	25	12	10	15

HEAP SORT:

The order property of heap i.e., max heap is used in sorting. Given n numbers arranged as heap, the largest element is moved to the sorted sublist. The remaining elements are adjusted to again form a heap. This forms one pass of heap sort. If these steps are repeated N times, then the list with N elements will be sorted. In every pass, the sorted list grows by one element and the heap size reduces by one. At the N^{th} pass the sorted list will have all the N elements. As heaps are almost complete binary trees, array representation is efficient and is used in implementation of Heap sort.

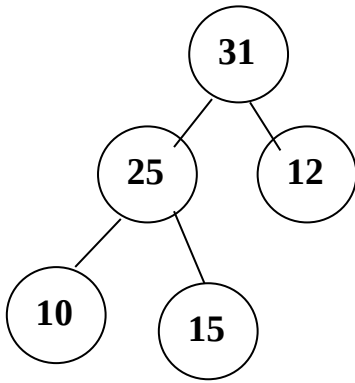
Heap sort for the following numbers are illustrated below: 10, 15, 12, 25, 31.

Before sorting the elements, heap tree should be obtained and then the array representation.

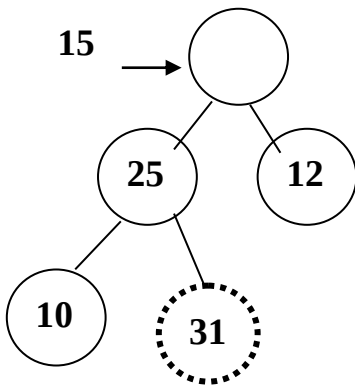


The array is unsorted and the maximum element is at the root (and hence in the first position of the array).

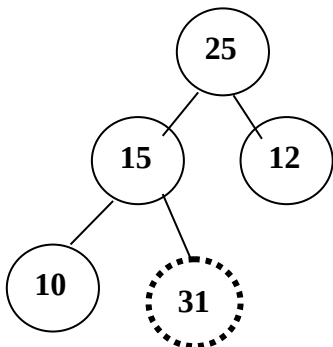
For sorting the elements in the heap, move the value at the root to the last leaf and place the value in the last leaf in the proper position, i.e., the root will be vacant when it is moved down but at the same time, the value at the last leaf cannot be placed there because it may violate the order property of the heap. Hence the heap should be adjusted properly to maintain the order. Repeat this for all elements till the sorted array grows to the size of N .

PASS 0

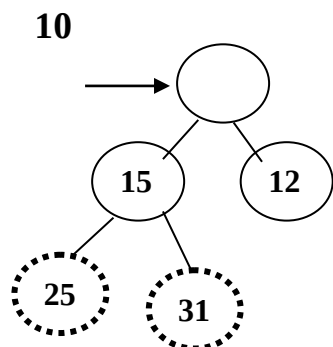
a[0]	a[1]	a[2]	a[3]	a[4]
31	25	12	10	15

PASS 1

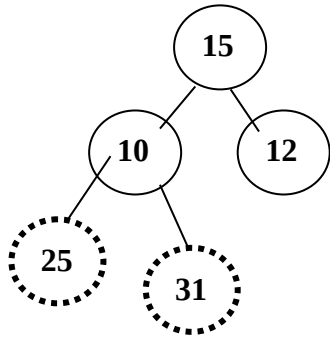
a[0]	a[1]	a[2]	a[3]	a[4]
	25	12	10	31



a[0]	a[1]	a[2]	a[3]	a[4]
25	15	12	10	31

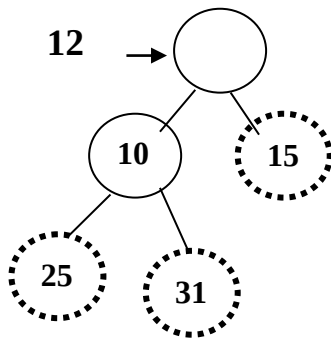
PASS 2

a[0]	a[1]	a[2]	a[3]	a[4]
	15	12	25	31

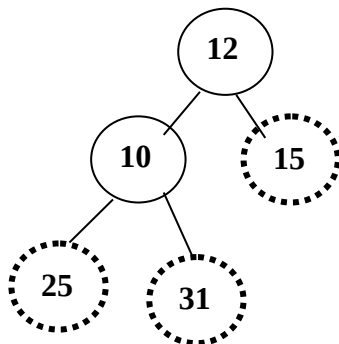


a[0]	a[1]	a[2]	a[3]	a[4]
15	10	12	25	31

PASS 3

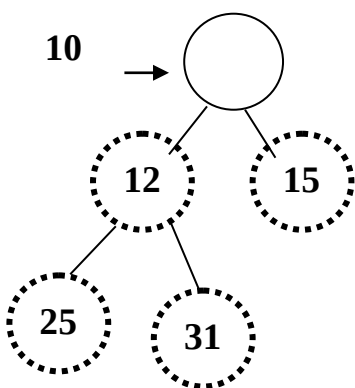


a[0]	a[1]	a[2]	a[3]	a[4]
	10	15	25	31

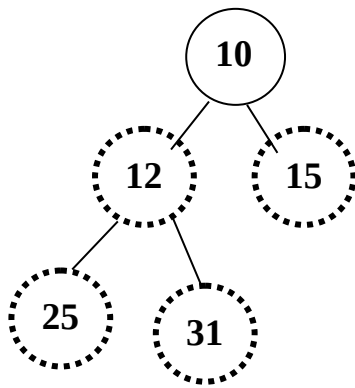


a[0]	a[1]	a[2]	a[3]	a[4]
12	10	15	25	31

PASS 4



a[0]	a[1]	a[2]	a[3]	a[4]
	12	15	25	31



a[0]	a[1]	a[2]	a[3]	a[4]
10	12	15	25	31

Thus the sorted elements are 10, 12, 15, 25, 31.

Algorithm for Heap Sort:

1. Construct a MaxHeap for the given input sequence
2. Remove the root element from the heap and store it in an array
3. Fill the vacancy of the heap with the last element (at the deepest level right corner) of the heap
4. Again check the heap order property for this new heap
5. Repeat steps 2 to 4 until all the elements removed from the heap

PART – A

1. Define Tree

A tree is a data structure, which represents hierarchical relationship between individual data item.

2. Define Non-linear data structure. (OR) Give two examples of non-linear data structures, which are widely used.

Data structure, which is capable of expressing more complex relationship than that of physical adjacent, is called non-linear data structure.

Example: Trees, graph, lists

3. What is meant by Binary tree?

- ✓ A binary tree is a two array tree i.e., each node in the tree can have either 0, or 1 or 2 sub trees.
- ✓ A binary tree is a finite ordered collection of elements in which one element is designated as root & remaining elements are partitioned into 2-disjoint sets, called left & right sub trees.
- ✓ A binary tree is a tree that has nodes either empty or not more than 2 child nodes, each of which may be a leaf node.

4. Indicate the properties of binary tree. (Nov / Dec 2023)

- ✓ Each node can have at most two children.
- ✓ The maximum number of nodes increases with height, and the tree's height is smallest when it is complete.

5. What are the various operations performed on binary tree using linked representation?

- ✓ Insertion – Insert a given item into a tree.
- ✓ Traversal – Processing the nodes of the tree one by one.
- ✓ Search – Search for the specified item.
- ✓ Copying – To obtain the exact copy of the given tree.

- ✓ Deletion – Delete an item from the tree.

6. What are the two methods of binary tree implementation?

- ✓ Linear representation.
- ✓ Linked representation.

7. What are the different types of tree traversal technique?

The different types of tree traversal technique are,

- ✓ In-order traversal
- ✓ Pre-order traversal
- ✓ Post- order traversal

8. Define Pre-order traversal.

Pre-order traversal contains the following steps:

- ✓ Process the root node.
- ✓ Process the left sub tree.
- ✓ Process the right sub tree.

9. Define Post- order traversal.

Post- order traversal contains the following steps:

- ✓ Process the left sub tree.
- ✓ Process the right sub tree.
- ✓ Process the root node.

10. Define In-order traversal.

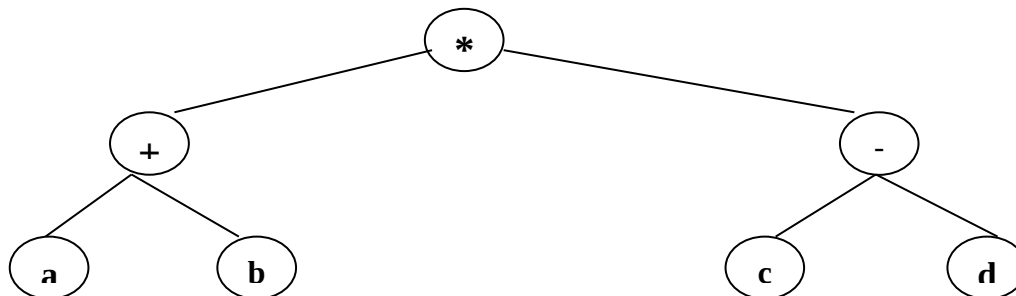
In-order traversal contains the following steps:

- ✓ Process the left sub tree.
- ✓ Process the root node.
- ✓ Process the right sub tree.

11. Define Strictly and complete binary tree.

- ✓ STRICTLY BINARY TREE: In a binary tree where all the nodes will have either 0 or 2 children. Nodes cannot have only one child.
- ✓ COMPLETE BINARY TREE: It is strictly binary tree in which all leaves are at the last level.

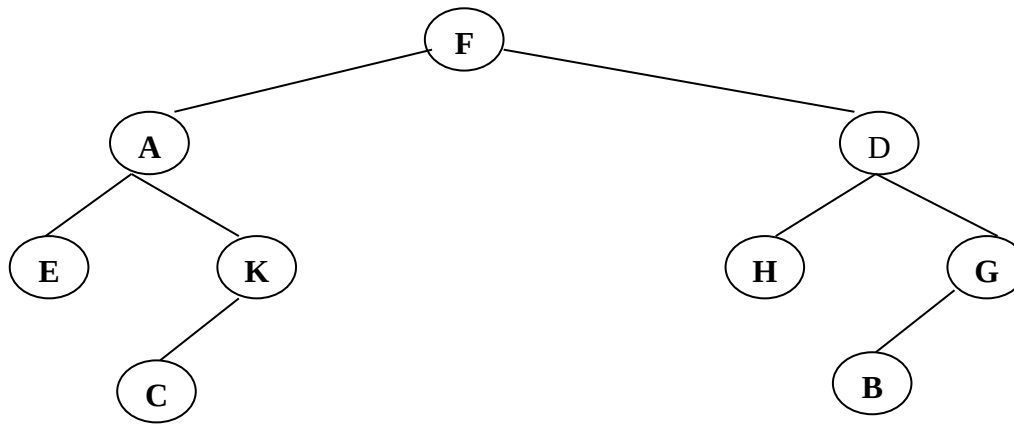
12. Construct a binary tree for the expression $(a + b) * (c - d)$. (Nov / Dec 2022)



13. A binary tree T has 9 nodes. The inorder and postorder traversals of T yield the following:

Inorder traversal (I): E A C K F H D B G, Postorder traversal (Po): E C K A H B G D F.

Draw the binary tree. (Apr / May 2024)



14. Define Binary Search tree.

A binary tree where the value at root is greater than those in left sub tree and less than those in its right sub tree is called a **Binary search tree (BST)**. Both left & right sub trees also have this property of BST.

15. State the properties of binary search tree. (Apr / May 2023)

- ✓ Left subtree values are smaller, right subtree values are larger than the root.
- ✓ Inorder traversal of a BST gives elements in ascending order.

16. Define threaded binary tree. (Nov / Dec 2024)

A threaded binary tree is a binary tree in which the NULL pointers (where a child does not exist) are replaced with special pointers called threads that point to the node's inorder predecessor or successor.

This helps in performing inorder traversal without using recursion or a stack.

17. What is meant by height balanced trees or AVL trees?

AVL trees are named after the inventor's **Adelson, Velskii & Landis**. An AVL tree is a self adjusting, height balanced BST where the difference in heights of the Left & Right sub trees is a maximum of 1, i.e., every node in a tree is height balanced (difference in heights of the left & right sub trees is at most 1). These trees are also referred to as **admissible trees**.

18. What is a balance factor in AVL trees?

The **balance factor** of a node is defined as the height of the R sub tree minus the height of the L sub tree. Hence in an AVL tree, the balance factor of each node is +1, 0, or -1.

$$\text{Balance Factor} = h_R - h_L$$

Where $h_R - h_L$ denote the height of the Left & Right sub trees respectively.

19. Define AVL rotation.

In an AVL Tree, during insertion or deletion of a node, the nodes in the path from the insertion / deletion point to the root may become unbalanced. In such cases, the structure of the tree should be slightly modified to restore the balance of the nodes. Such modification of the structure is called as **Rotation**.

20. What are the properties of AVL trees? (Apr / May 2024)

- ✓ Height-balanced – For every node, the height difference between the left and right subtrees (balance factor) is -1, 0, or +1.
- ✓ Self-balancing – Rotations are performed automatically after insertions/deletions to maintain balance.

21. Mention the types of rotations performed on AVL tree. (Apr / May 2023)

- ✓ Left Rotation (LL)
- ✓ Left-Right Rotation (LR)
- ✓ Right Rotation (RR)
- ✓ Right-Left Rotation (RL)

22. Distinguish binary search tree and AVL tree. (Nov / Dec 2023)

Binary Search Tree (BST)	AVL Tree
Follows the BST property: left subtree < root < right subtree.	Follows BST property and height-balance property.
May become unbalanced after multiple insertions/deletions.	Always balanced; height difference between subtrees ≤ 1 .
No rotations are performed.	Performs rotations (LL, RR, LR, RL) to maintain balance.

23. What is a heap? Outline the properties of a heap. (Nov / Dec 2022) (Nov / Dec 2024)

- ✓ A binary heap is a complete binary tree in which elements are stored to satisfy the heap property:
 - Max-Heap: Every parent node's value is greater than or equal to its children's values.
 - Min-Heap: Every parent node's value is less than or equal to its children's values.
- ✓ It is commonly used to implement priority queues*.

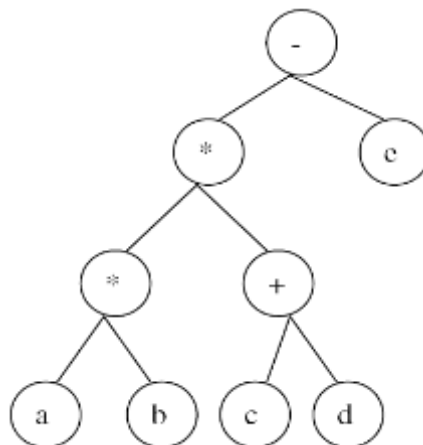
24. What are the applications of trees?

Trees are used in the representation of sets.

- ✓ Decision-making. E.g.: Eight coins problem.
- ✓ Computer-games such as chess, tic-tac-toe, checkers etc., are created and executed using trees.
- ✓ Manipulation of arithmetic expression.
- ✓ Symbol table construction.
- ✓ Syntax-analysis.

PART B**TREE ADT – TREE TRAVERSALS - BINARY TREE ADT – EXPRESSION TREES**

1. Give the prefix, infix and postfix expressions corresponding to the tree given in figure.



2. Explain how a postfix expression is converted into an expression tree with an example. (6)
(Nov / Dec 2023)
3. Write an algorithm for preorder, inorder and postorder traversal of a binary tree. (Nov / Dec 2024)
4. Outline preorder, inorder and postorder traversal on a binary tree with an algorithm and an example.
(Nov / Dec 2022)

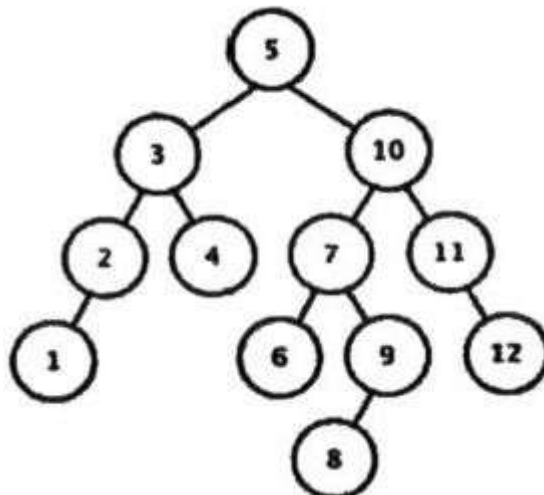
5. Classify the three types of binary tree traversal and write the pseudo-code for in-order traversal with an example. (7) (Nov / Dec 2023)

BINARY SEARCH TREE ADT

6. State the binary search tree property and outline the algorithm to search a binary search tree with an example. (Nov / Dec 2022)
7. (a) How do you insert an element in a binary search tree? (8)
(b) Show that for the perfect binary tree of height h containing $2^{h+1}-1$ nodes, the sum of the heights of the nodes is $2^{h+1}-1-h(h+1)$. (8)
8. Explain how to find a maximum element and minimum element in BST? Explain detail about Deletion in Binary Search Tree?
9. Write the insertion algorithm for binary search tree and analyze its complexity. (7) (Nov / Dec 2023)
10. How to insert and delete an element into a binary search tree and write down the pseudo code with an example. (Nov / Dec 2024)
11. Write a C function for the following in the Binary search tree: (5+5+3) (Apr / May 2024)
(i) To find the height of a tree.
(ii) To find minimum and maximum.
(iii) Preorder traversals.
12. Distinguish between Binary tree, general tree and binary search tree and also give an example. (6) (Apr / May 2024)

AVL TREES

13. Given the AVL Tree in Figure. Draw the resulting balanced tree step by step after 5 is removed. Label each node with balance factor. (7) (Apr / May 2024)



14. Summarize how rebalancing occurs in AVL trees during double rotation with an example. (6) (Nov / Dec 2023)
15. Explain in detail (i) Single rotation (ii) double rotation of an AVL tree.

PRIORITY QUEUE (HEAPS) - BINARY HEAP

16. Explain the efficient implementation of the priority queue ADT.
17. Identify the types of Priority Queue. Explain about min heap insertion and deletion operations. (Apr / May 2023)
18. Explain the operation and implementation of Binary Heap.
19. Explain Heap tree ADT in detail. (Apr / May 2023)