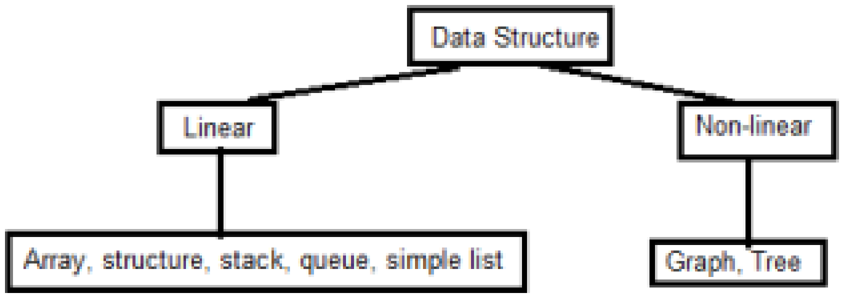
1.Define Data Structures. Explain the classification of data structures with a neat diagram.

Ans:

A structure represents name, type, size and format. A data structure is a represents to organize and store data in terms of name, type, size and & format in computers memory. Data may contain a single element or sometimes it may be a set of elements. Whether it is a single element or **multiple** elements but it must be organized in a particular way in the computer memory system.

General data structure types include the array, lists, vectors, the file, the record, the table, the tree, graphand so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and operated with appropriate ways.

Types of data structure:



Data Structure operations:

· Addition/Insertion: To add data at beginning or end or to insert data between two data · Sorting: To sort data either in ascending or descending order to make the search operation faster

· Search: To search data within given collection by different ways

· Modification: To modify existing data with new data

· Deletion: To delete existing data

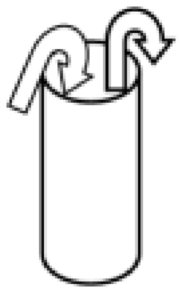
b. Write a C Functions to implement pop, push and display operations for

stacks using arrays.

Ans:

A stack is a LIFO(Last in first out) data structure having one end open only to insert or delete items. The item inserted at last will be deleted at first.

A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.



A stack is generally implemented with only two basic principle operations

· Push : adds an item to a stack on “top”.

· Pop: extracts the most recently pushed item from the stack from “top”.

Procedure/Functions:

```
/*MAX is the size of stack & item is to be pushed incrementing  the top variable */

void push(int stack[], int item)

{

 if (top == (MAX - 1))

 printf(“Stack is full/Overflow”);

 else

 {

 top=top+1;

 stack[top]=item;

 }

}

/* function to pop the elements off the stack */

int pop(int stack[])
```

```
{
 int ret;
 if(top == -1)
 {
 printf("The stack is empty/underflow" );
 return (NULL);
 }
 else
 {
 ret = stack[top];
 top = top-1;
 return (ret);
 }
 }
/* function to display stack elements */
void display(int stack[])
{
 int x;
 printf("The Stack:\n");
 for(x=top;x>=0;x--)
 printf(" %d\n", stack[x]);
}
```

C. Differentiate structures and unions.

Points	Structure	Union
1	Structure allocates storage space for all its members separately. Keyword: **struct**	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space. Keyword: **union**
2	Structure occupies larger memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.

4	Structure example:
	struct student
	{
	int mark;
	double average;
	};
		Union example:
		union student
		{
		int mark;
		double average;
		};

5	For above structure, memory allocation will be like below. int mark – 2 Bytes double average – 8B Total memory allocation = 2+8 = 10 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

2 a.Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression.

62/3-4 2*+.

Algorithm to evaluate postfix expression:

i. Read postfix expression from left to right (Taking care of precedence & Parentheses) a. If operand is encountered push it onto stack

b. If operator is encountered, pop two operands (a= top element; b= next to top element)

c. Evaluate b operator a and push onto stack

ii. Set final result to pop

Variables	**Stack**	**Popped Elsement**
6	**6**	
2	**2**	
/		6/2=3
3	**3 3**	
-		3-3=0
4	**0 4**	
2	**0 4 2**	
*****	**0**	4*2=8
+		0+8=8

Final Item	8

b. Explain the dynamic memory allocation function in detail.

Ans.

Dynamic Memory allocation and de-allocation:

Memory can be allocated/ de-allocated at run-time as and when required. C has four in-built functions for the same with header file stdlib.h

malloc(), calloc() and realloc() to allocate and free() to de-allocate

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size in bytes and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

ptr=(cast-type*)malloc(byte-size);

example:

ptr=(int*)malloc(100*sizeof(int));

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax for calloc():

ptr=(cast-type*)calloc(n,element-size);

example:

ptr=(int*)calloc(25,sizeof(int));

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax/example:

ptr=realloc(ptr, newsize);

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

Syntax/example:

free(ptr);

C.What is Sparse matrix? Give the triplet form of a given matrix and find its transpose

0 0 3 0 4

0 0 5 7 0

A =

0 0 0 0 0

0 2 6 0 0

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

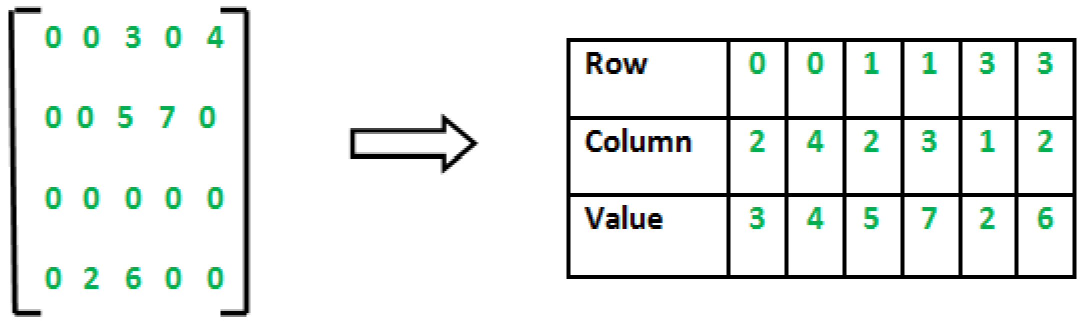
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

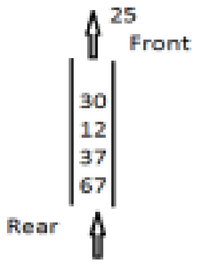
.

Module - 2

3.a. Define Queue. Discuss how to represent a queue using dynamic arrays.

Ans.

Queue is FIFO (first in first out data structure). Items are processed and deleted according to their sequence of arrival in the queue. “Items are inserted from rear and deleted from front end.



A linear queue of size 5

Disadvantages of linear queue:

· Items cannot be inserted from front end even the space is available

· Items cannot be deleted from rear or middle

b. Write a C Function to implement insertion (), deletion () and display ()

operations on circular queue.

//insert

```
void insertq ( int arr[], int item )
{
if (( rear == MAX - 1 && front == 0)||( rear + 1 == front ))
 {
printf("\nCircular Queue is full" );
return ;
}
rear=(rear+1)%MAX;
arr[rear] = item ;
if ( front == -1 )
front = 0 ;
}
```

//delete

```
int deleteq(int arr[] )
{
int data ;
if ( front == -1 )
 {
 puts("\nQueue is empty") ;
 return(NULL);
 }
data = arr[front] ;
arr[front] = 0 ;
if ( front == rear )
 front = rear = -1; /* when all elements are deleted */
else
 front=(front+1)%MAX;
return (data) ;
}
```

//display

```
void display(int cq[])
{
 int x;
 printf(“The items from circular queue:\n”);
 for(x=0;x<MAX;x++)
 printf(“%5d”,cq[x]);
 printf(“\n”);
}
```

C.Write a note on Multiple stacks and queues with suitable diagram.

MULTIPLE STACKS AND QUEUES

• In multiple stacks, we examine only sequential mappings of stacks into an array. The array is one dimensional which is memory[MEMORY_SIZE]. Assume n stacks are needed, and then divide the available memory into n segments. The array is divided in proportion if the expected sizes of the various stacks are known. Otherwise, divide the memory into equal segments. • Assume that i refers to the stack number of one of the n stacks. To establish this stack, create indices for both the bottom and top positions of this stack. boundary[i] points to the position immediately to the left of the bottom element of stack i, top[i] points to the top element. Stack i is empty iff boundary[i]=top[i]. The declarations are:

```
 #define MEMORY_SIZE 100

/* size of memory

*/ #define MAX_STACKS 10

/* max number of stacks plus 1

*/ element memory[MEMORY_SIZE];

 /* global memory declaration

*/ int top [MAX_STACKS]; int boundary [MAX_STACKS] ; int n; /*number of stacks entered by
the user */ To divide the array into roughly equal segments top[0] = boundary[0] = -1;

for (j= 1;j< MAX_STACKS, and m =MEMORY_SIZE.
```

 Stack i grow from boundary[i] + 1 to boundary [i + 1] before it is full. A boundary for the last stack is needed, so set boundary [n] to MEMORY_SIZE-1

Implementation of the add operation

```
void push(int i, element item)

 { /* add an item to the ith stack */

 if (top[i] == boundary[i+l])

stackFull(i);

 memory[++top[i]] = item; }
```

Program: Add an item to the ith stack Implementation of the delete operation element pop(int i) {

/* remove top element from the ith stack */

if (top[i] == boundary[i]) return

stackEmpty(i);

return memory[top[i]--]; }

Program: Delete an item from the ith stack The top[i] == boundary[i+1] condition in push implies only that a particular stack ran out of memory, not that the entire memory is full. But still there may be a lot of unused space between other stacks in array memory as determines if there is any free space in memory. If there is space available, it should shift the stacks so that space is allocated to the full stack.

OR

Q.4 a. What is a linked list? Explain the different types of linked list with neat diagram.

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers. In simple words, a linked list consists of nodes where each node contains a data field and a **reference(link)** to the next node in the list.

b. Write a C function for the following on singly linked list with example :

i) Insert a node of the beginning

ii) Delete a node at the front

iii) Display.

C.Write the C function to add two polynomials.

Ans.

NODE **poly_add(NODE head1, NODE head2, NODE head3)**

{

NODE a,b;

int coeff;

```
a = head1->link;
b = head2->link;
while(a != head1 && b != head2)
{
if(a->expon == b->expon)
{
coeff = a->coeff + b->coeff;
if(coeff != 0)
head3 = attach(coeff, a->expon, head3);  a = a->link;
b = b->link;
}
else if(a->expon > b->expon)
{
head3 = attach(a->coeff, a->expon, head3);  a = a->link;
}
else
{
head3 = attach(b->coeff, b->expon, head3);  b = b->link;
}
}
while(a != head1)
{
head3 = attach(a->coeff, a->expon, head3);  a = a->link;
}
while(b != head2)
{
 head3 = attach(b->coeff, b->expon, head3);  b = b->link;
 }
 return head3;
}
```

Module - 3

Q.5 a. Discuss how binary trees are represented using: i) Array ii) Linked list.

Binary tree is a **tree data structure (non-linear)** in which each node can have at most two children which are referred to as the **left** child and the **right** child. The topmost node in a binary tree is called the **root**, and the bottom-most nodes are called **leaves**. A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom.

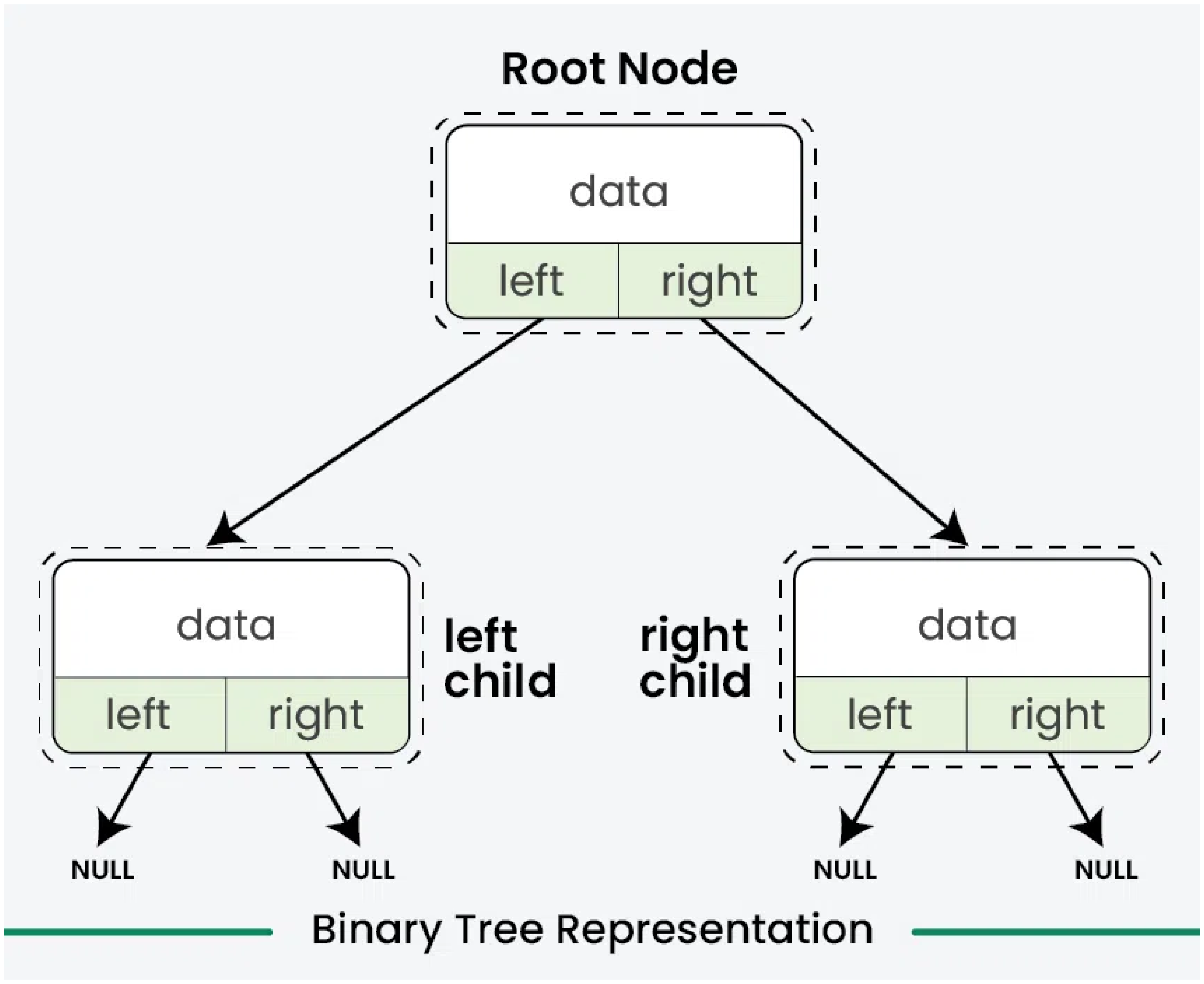
Representation of Binary Trees

There are two primary ways to represent binary trees:

1. Linked Node Representation
2. Array Representation

1. Linked Node Representation

This is the simplest way to represent a binary tree. Each node contains **data** and pointers to its **left** and **right** children.



This representation is mostly used to represent binary tree with multiple advantages. The most common advantages are given below.

Advantages:

- It can easily grow or shrink as needed, so it uses only the memory it needs.
- Adding or removing nodes is straightforward and requires only pointer adjustments.
- Only uses memory for the nodes that exist, making it efficient for sparse trees.

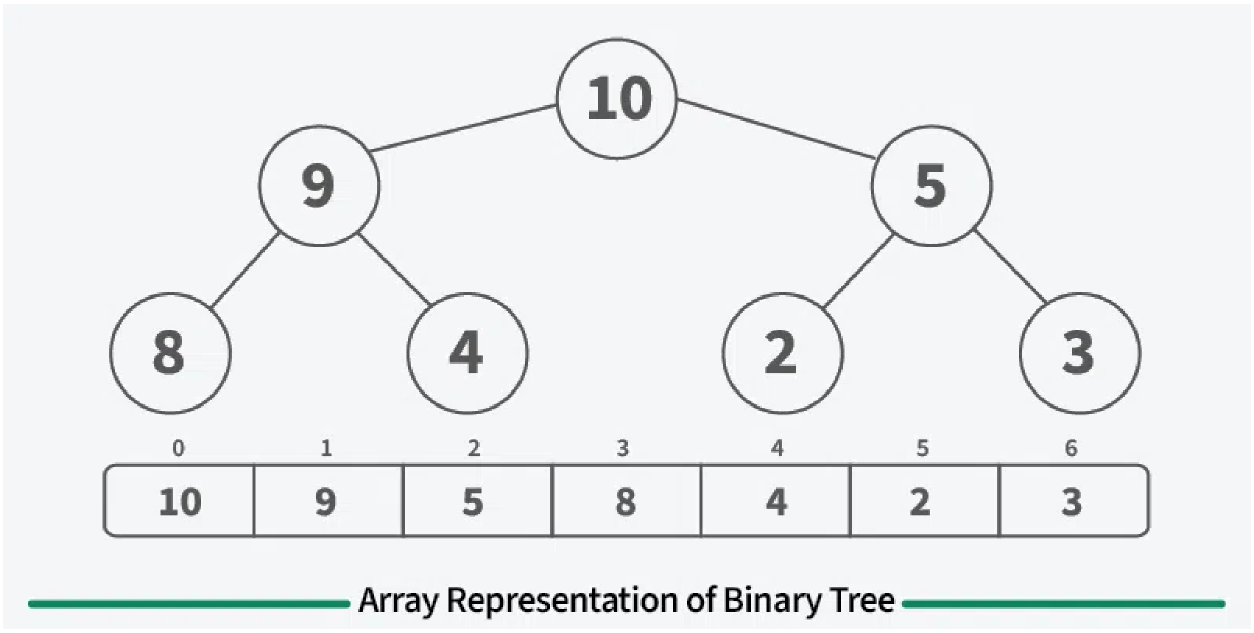
Disadvantages:

- Needs extra memory for pointers.
- Finding a node can take longer because you have to start from the root and follow pointers.

2. Array Representation

Array Representation is another way to represent binary trees, especially useful when the tree is complete (all levels are fully filled except possibly the last, which is filled from left to right). In this method:

- The tree is stored in an array.
- For any node at index **i**:
  - Left Child: Located at **2 * i + 1**
  - Right Child: Located at **2 * i + 2**
- Root Node: Stored at index **0**



b. Define Threaded binary tree. Discuss In- threaded binary tree.

A threaded binary tree is a type of binary tree that uses threads to link nodes instead of null pointers. This allows for faster traversal of the tree without the need for a stack or recursion.

In a threaded binary tree, null child pointers are replaced with pointers to in-order predecessors or successors. These pointers are called "threads".

How it works

- In a standard binary tree, many child pointers are null, especially in leaf nodes.
- In a threaded binary tree, the null pointers are replaced with pointers to in-order predecessors or successors.

- The leftmost and rightmost child pointers of a tree always point to null.

- The higher node to which the thread points in the tree is determined according to the tree traversal technique used.

C.

Write the C function for the following additional list operation

i) Inverting Singly linked list

```
struct Node* reverseList(struct Node* head) {
        struct Node *curr = head,
         *prev = NULL, *next;
          while (curr != NULL) {
          next = curr->next;
           curr->next = prev;
           prev = curr;
           curr = next;
  }
  return prev;
}
```

ii) Concatenating Singly linked list.

```
struct Node *concat(struct Node *head1,  struct Node *head2) {

  if (head1 == NULL)
    return head2;

  // Find the last node of the first list
  struct Node *curr = head1;
  while (curr->next != NULL){
    curr = curr->next;
  }
  curr->next = head2;
  return head1;
}
```

OR

Q.6 a. Discuss Inorder, Preorder, Postorder and Level order traversal with suitable function for each.

Ans:

Inorder Traversal

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

So, in the inorder traversal, each node is visited in between of its subtrees.

The applications of Inorder traversal includes -

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

```
void inorderTraversal(struct Node* root) {
   if (root == NULL)
      return;
   inorderTraversal(root->left);
     printf("%d ", root->data);
    inorderTraversal(root->right);
}
```

Preorder Traversal

This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

So, in a preorder traversal, each node is visited before both of its subtrees.

```
void preorderTraversal(struct Node* root) {
   if (root == NULL)
      return;
  printf("%d ", root->data);
   preorderTraversal(root->left);
```

```
    preorderTraversal(root->right);
}
```

Postorder Traversal

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

So, in a postorder traversal, each node is visited after both of its subtrees.

```
void postorderTraversal(struct Node* node) {
    if (node == NULL)
        Return;
    postorderTraversal(node->left);
    postorderTraversal(node->right);
    printf("%d ", node->data);
}
```

b. Define the threaded binary tree. following element: A, B, C, D, E, F, G, H, I.

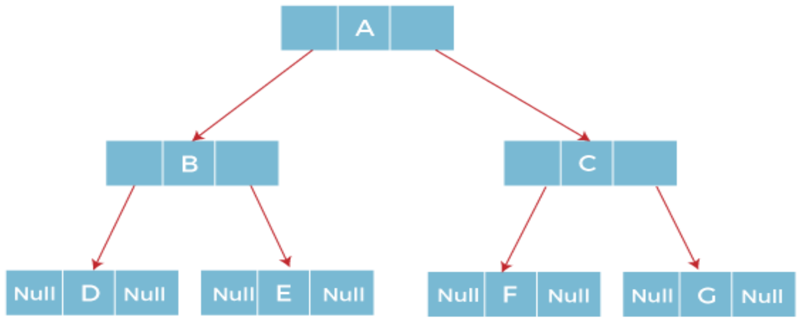
Construct threaded binary tree .

The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

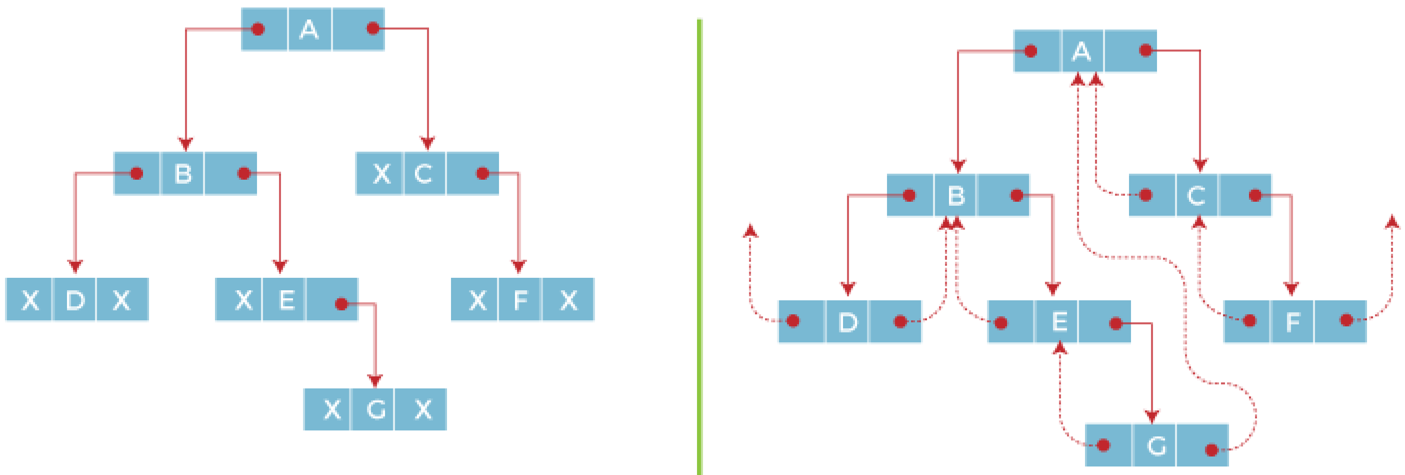
A threaded binary tree is a type of binary tree data structure where the empty left and right child pointers in a binary tree are replaced with threads that link nodes directly to their in-order predecessor or successor, thereby providing a way to traverse the tree without using recursion or a stack.

Threaded binary trees can be useful when space is a concern, as they can eliminate the need for a stack during traversal. However, they can be more complex to implement than standard binary trees.

There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.
The threads are also useful for fast accessing ancestors of a node.



Threaded binary tree



A binary tree (Inorder traversal - D, B, E, G, A, C, F)

A two - way threaded binary tree

C. Write a C function for the following:

i) Deleting a node at the end of the doubly linked list.

```
struct Node* delLast(struct Node *head) {
    if (head == NULL)
        return NULL;
    if (head->next == NULL) {
        free(head);
        return NULL;
    }

    // Traverse to the last node
    struct Node *curr = head;
    while (curr->next != NULL)
        curr = curr->next;

    // Update the previous node's next pointer
    curr->prev->next = NULL;

    // Delete the last node
    free(curr);

    // Return the updated head
    return head;
}
```

ii)Insert a node at the beginning of doubly linked list.

```
struct Node *insertAtFront(struct Node *head, int new_data) {
    // Create a new node
    struct Node *new_node = createNode(new_data);


    // Make next of new node as head
    new_node->next = head;


    // Change prev of head node to new node
    if (head != NULL) {
        head->prev = new_node;
    }


    // Return the new node as the head of the doubly linked list
    return new_node;
}
```

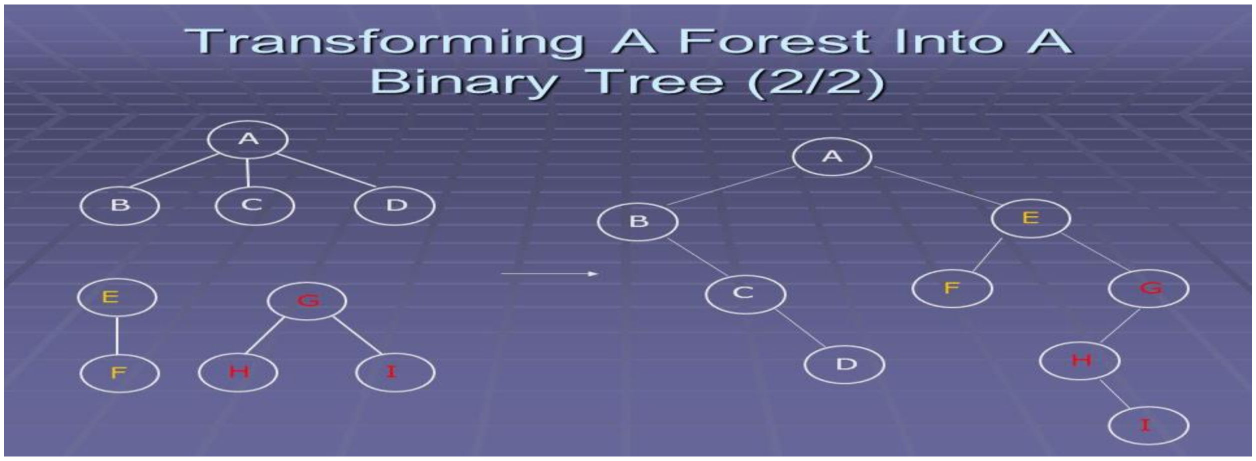
Module - 4

Q.7 a. Define Forest, Transform the forest into a binary tree and traverse using inorder, preorder and postorder traversal with an example.

A forest is generally defined as a collection of trees. To transform a forest into a binary tree, we typically use a method that links the roots of the trees in the forest.

Consider a forest with trees T1, T2, T3:

- **Binary tree construction:**
  - T1 becomes the root node.
  - T2 becomes the right child of T1's left child (which is initially empty).
  - T3 becomes the right child of T2's right child (which is also initially empty).



b. Define Binary search tree. Construct a binary search tree for the following 6

elements: 100, 85, 45, 55, 120,20,70, 90, 115, 65, 130, 145.

A **Binary Search Tree (or BST)** is a data structure used in computer science for organizing and storing data in a sorted manner. Each node in a **Binary Search Tree** has at most two children, a **left** child and a **right** child, with the **left** child containing values less than the parent node and the **right** child containing values greater than the parent node. This hierarchical structure allows for efficient **searching**, **insertion**, and **deletion** operations on the data stored in the tree.

```
                    ↙100↘

            ↙  85            ↙  120↘

       ↙  45   ↘                115        130↘

  20              55↘                            145

                    ↙ 70↘

                   65        90
```

C. Discuss Selection tree with an example.

The Tournament tree is a complete binary tree with n external nodes and n – 1 internal nodes. The external nodes represent the players, and the internal nodes are representing the winner of the match between the two players. This tree is also known as Selection tree.

There are some properties of Tournament trees. These are like below –

- This tree is rooted. So the link in the tree and directed path from parent to children, and there is a unique element with no parents
- The parent value is less or equal to that node to general any comparison operators, can be used as long as the relative value of the parent and children are invariant throughout the tree
- Trees with a number of nodes not a power of 2, contain holes. Holes can be present at any place in the tree.
- This tree is a proper generalization of binary heaps
- The root will represent overall winner of the tournament.

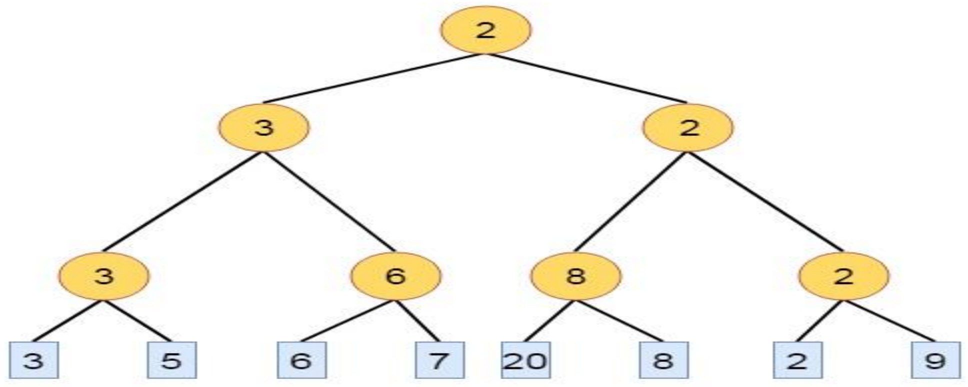
There are two types of Tournament Trees –

- Winner Tree
- Looser Tree

Winner Tree

Winner tree is a complete binary tree, in which each node is representing the smaller or greater of its two children, is called winner tree. The root is holding the smallest or greatest node of the tree. The winner of the tournament tree is the smallest or greatest n key in all the sequences. It is easy to see that winner tree can be formed in O(log n) time.

Example – Suppose there are some keys, 3, 5, 6, 7, 20, 8, 2, 9

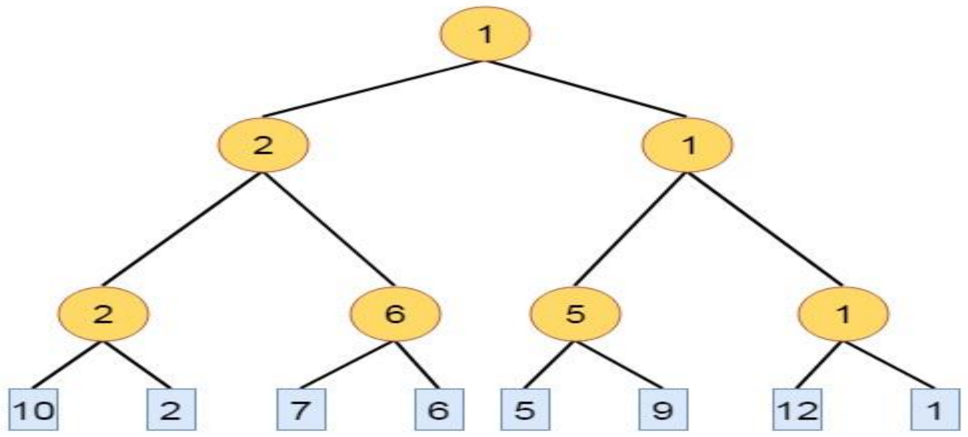


Looser Tree

Looser Trees are complete binary tree for n players where n external nodes and n – 1 internal nodes are present. The looser of the match is stored in the internal nodes. But in this overall winner is stored at tree[0]. The looser is an alternative representation, that stores the looser of a match at the corresponding node. An advantage of the looser is that, to restructure the tree after winner tree been output, it is sufficient to examine node on the path from leaf to root rather than the sibling of the nodes on this path.

Example – To form a looser tree, we have to create winner tree at first.

Suppose there are some keys, 10, 2, 7, 6, 5, 9, 12, 1. So we will create minimum winner tree at first.



OR

Q.8 a. Define Graph. Explain adjacency matrix and adjacency list representation 8

with an example

A **Graph** is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E)**.

Representations of Graph

Here are the two most common ways to represent a graph

1. Adjacency Matrix

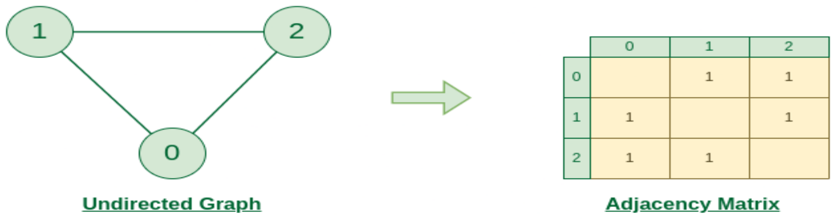
2. Adjacency List

Adjacency Matrix

Adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)

Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- *If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.*
- *If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.*



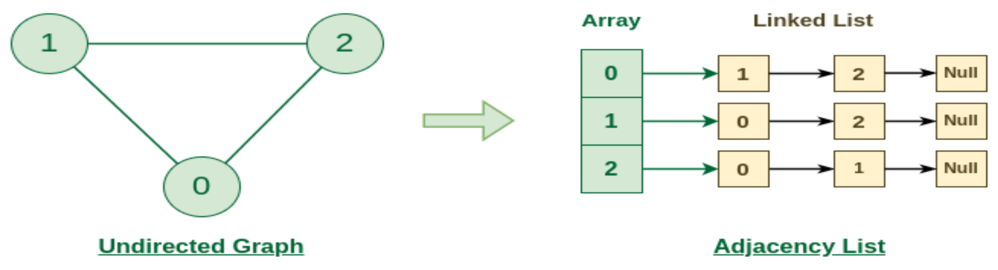
Graph Representation of Undirected graph to Adjacency Matrix

Adjacency List

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.

Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- *adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.*
- *adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.*

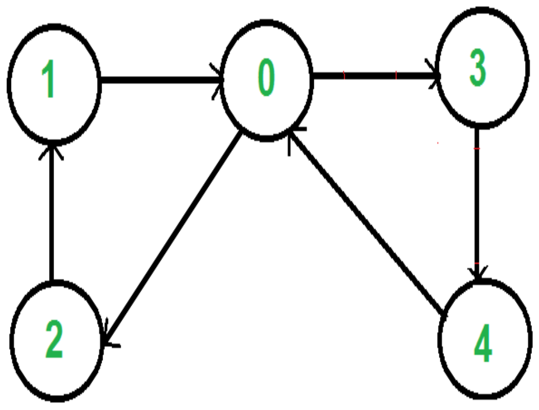


Graph Representation of Undirected graph to Adjacency List

b. Define the following terminology with example:

i) Digraph ii) Weighted graph iii) Self loop iv) Connected graph. |

- *A **directed graph** is defined as a type of graph where the edges have a direction associated with them.*



A weighted graph or a network is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand. Such graphs arise in many contexts, for example in shortest path problems such as the traveling salesman problem.

A self-loop, also known as a loop or self-edge, is an edge in a graph that connects a vertex to itself.

A connected graph is a graph where every pair of vertices is connected by a path. A path is a sequence of edges that joins two vertices.

c. Briefly explain about Elementary graph operations.

Operations on Graph in Data Structure

Following are the basic graph operations in data structure:

- Add/Remove Vertex – Add or remove a vertex in a graph.
- Add/Remove Edge – Add or remove an edge between two vertices.
- Check if the graph contains a given value.
- Find the path from one vertex to another vertex.

Graph Traversal in Data Structure

Graph traversal is visiting or updating each vertex in a graph. The order in which they visit the vertices classifies the traversals. There are two ways to implement a graph traversal:

- **Breadth-First Search (BFS) –** It is a traversal operation that horizontally traverses the graph. It traverses all the nodes at a single level before moving to the next level. It begins at the graph's root and traverses all the nodes at a single depth level before moving on to the next level.
- **Depth-First Search (DFS):** This is another traversal operation that traverses the graph vertically. It starts with the root node of the graph and investigates each branch as far as feasible before backtracking.

Module - 5

9 a.Explain in detail about Static and Dynamic Hashing.

Ans.

Hashing is an efficient technique to directly search the location of desired data on the disk without using index structure. Data is stored at the data blocks whose address is generated by using hash function. The memory location where these records are stored is called as data block or data bucket.

Static hashing:

· Single hash function h(k) on key k

· Desirable properties of a hash function

o Uniform: Total domain of keys is distributed uniformly over the range

o Random: Hash values should be distributed uniformly irrespective of distribution of keys O(1) search
· Example of hash functions: h(k) = k MOD m (size of hash table)
· Collision resolution
· Chaining
o Load factor
o Primary pages and overflow pages (or buckets)
o Search time more for overflow buckets
· Open addressing
o Linear probing
o Quadratic probing
o Double hashing
Problems of static hashing
· Fixed size of hash table due to fixed hash function
· **Primary/secondary Clustering (We should keep size of hash table a prime number to reduce clustering)**
· May require rehashing of all keys when chains or overflow buckets are full

Dynamic hashing:

In this hashing scheme the set of keys can be varied & the address space is allocated dynamically. If a file F is collection of record a record R is key+data stored in pages (buckets) then space utilization:

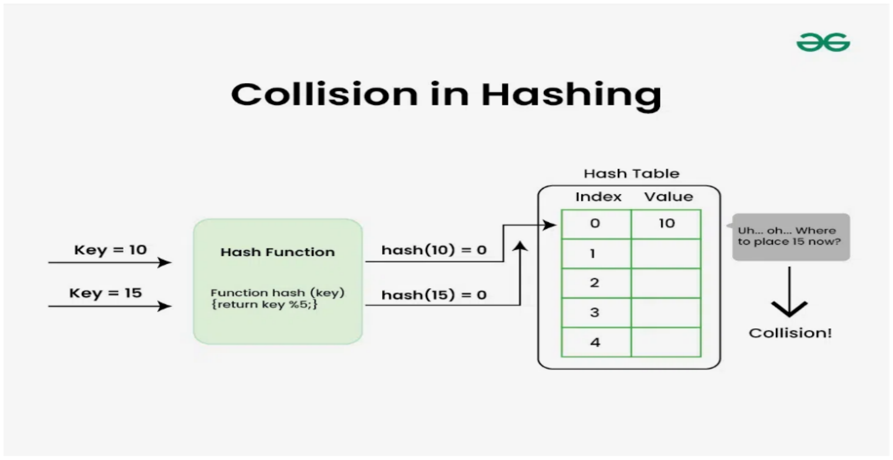
- **Number of records/(Number of pages*Page capacity)**

· Hash function modified dynamically as number of records grow
· Needs to maintain determinism
· Extendible hashing
· Linear hashing
Again: Dynamic hashing Hash function modified dynamically as number of records grow Needs to maintain determinism Extendible hashing and Linear hashing
· Organize overflow buckets as binary trees
· m binary trees for m primary pages
· $h_0(k)$ produces index of primary page
· Particular access structure for binary trees
· Family of functions : $g(k) = \{ h_1(k),....h_i(k),....\}$
· Each $h_i(k)$ produces a bit
· At level i, $h_i(k)$ is 0, take left branch, otherwise right branch Example: bit representation

b. What is Collision? What are the methods to resolve collision?

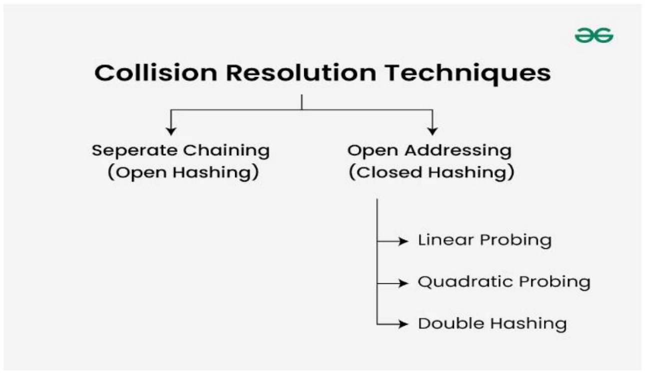
Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access.

In **Hashing**, hash functions were used to generate hash values. The hash value is used to create an index for the keys in the hash table. The hash function may return the same hash value for two or more keys. When two or more keys have the same hash value, a **collision** happens. To handle this collision, we use **Collision Resolution Techniques**.



There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing


Collision Resolution Techniques
Seperate Chaining (Open Hashing)
Open Addressing (Closed Hashing)
Linear Probing
Quadratic Probing
Double Hashing

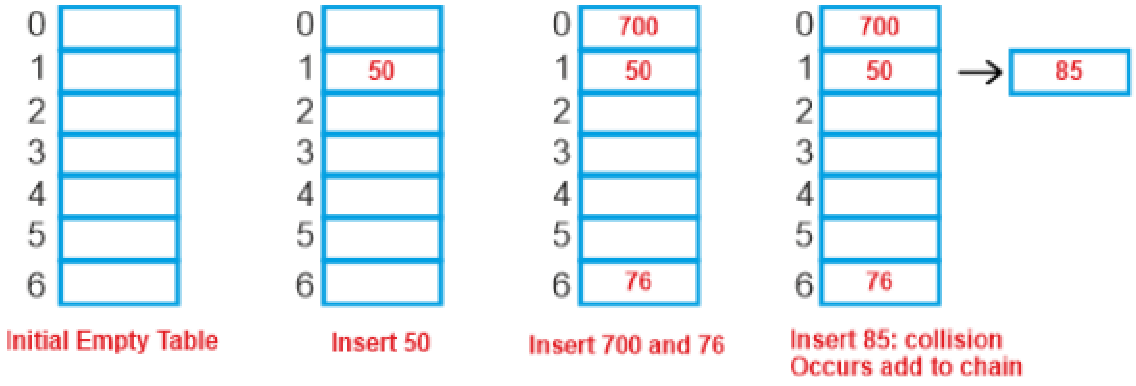
) Separate Chaining

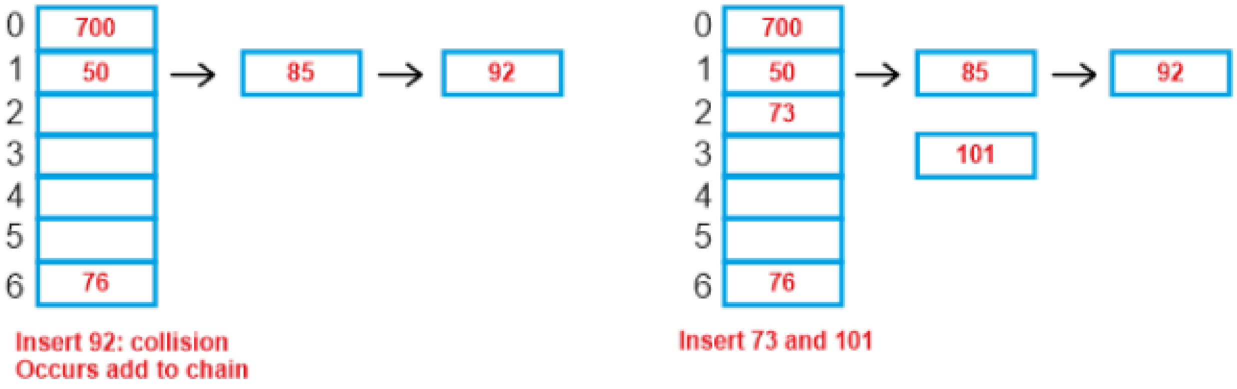
The idea behind Separate Chaining is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

With separate chaining, the array is implemented as a chain, which is a linked list. One of the most popular and often employed methods for handling accidents is separate chaining.

This method is implemented using the linked list data structure. As a result, when numerous elements are hashed into the same slot index, those elements are added to a chain, which is a singly-linked list. Here, a linked list is created out of all the entries that hash into the same slot index. Now, using merely linear traversal, we can search the linked list with a key K. If the intrinsic key for any entry equals K, then we have identified our entry. The entry does not exist if we have searched all the way to the end of the linked list and still cannot find it. In separate chaining, we therefore get to the conclusion that if two different entries have the same hash value, we store them both in the same linked list one after the other.

Let's use "key mod 7" as our simple hash function with the following key values: 50, 700, 76, 85, 92, 73, 101.





2) Open Addressing

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

2.a) Linear Probing

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

Algorithm:

1. *Calculate the hash key. i.e.* ***key = data % size***
2. *Check, if* ***hashTable[key]*** *is empty*
    - *store the value directly by* ***hashTable[key] = data***
3. *If the hash index already has some value then*
    - *check for next index using* ***key = (key+1) % size***
4. *Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.*
5. *Do the above process till we find the space.*

.

c. Explain Priority queue with the help of an examples.

A **priority queue** is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved or removed before elements with lower priority values. Each element has a priority value associated with it. When we add an item, it is inserted in a position based on its priority value.

There are several ways to implement a priority queue, including using an array, linked list, heap, or binary search tree, binary heap being the most common method to implement. The reason for using Binary Heap is simple, in binary heaps, we have easy access to the min (in min heap) or max (in max heap) and binary heap being a complete binary tree are easily implemented using arrays. Since we use arrays, we have cache friendliness advantage also.

Priority queues are often used in real-time systems, where the order in which elements are processed is not simply based on the fact who came first (or inserted first), but based on priority. Priority Queue is used in algorithms such as **Dijkstra's algorithm**, **Prim's algorithm**, **Kruskal's algorithm** and **Huffman Coding**.

Properties of Priority Queue

So, a priority Queue is an extension of the queue with the following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.

- If two elements have the same priority, they are served according to their order in the queue.

  Examples:

- **Hospital emergency room**

  Patients are treated in order of their medical condition, with the most critical patients treated first.
- **Task scheduling**

  Tasks with higher priority are executed first, such as system tasks, input/output operations, and critical applications.
- **Event-driven simulation**

  Events are placed in a priority queue and executed in order of their scheduled times.
- **Airline luggage**
  Baggage with higher priority, such as business or first-class, is loaded onto the conveyor belt first.

OR

b. Write short note on

i) Leftist trees

A leftist tree, also known as a leftist heap, is a type of binary heap data structure used for implementing priority queues. Like other heap data structures, it is a complete binary tree, meaning that all levels are fully filled except possibly the last level, which is filled from left to right.

1. In a leftist tree, the priority of the node is determined by its key value, and the node with the smallest key value is designated as the root node. The left subtree of a node in a leftist tree is always larger than the right subtree, based on the number of nodes in each subtree. This is known as the “leftist property.”
2. One of the key features of a leftist tree is the calculation and maintenance of the “null path length” of each node, which is defined as the distance from the node to the nearest null (empty) child. The root node of a leftist tree has the shortest null path length of any node in the tree.

3. The main operations performed on a leftist tree include insert, extract-min and merge. The insert operation simply adds a new node to the tree, while the extract-min operation removes the root node and updates the tree structure to maintain the leftist property. The merge operation combines two leftist trees into a single leftist tree by linking the root nodes and maintaining the leftist property.

In summary, a leftist tree is a type of binary heap data structure used for implementing priority queues. Its key features include the leftist property, which ensures that the left subtree of a node is always larger than the right subtree, and the calculation and maintenance of the null path length, which is used to maintain the efficiency of operations such as extract-min and merge.

ii) Optimal binary search tree.

A binary search tree is a tree in which the nodes in the left subtree are smaller than the root node, while the nodes in the right subtree are greater than the root node. When we talk about binary search trees, the searching concept comes first. The cost of searching for a node in the tree plays a very important role. Our purpose is to reduce the cost of searching, and that can only be done by an optimal binary search tree.

The optimal binary search tree (Optimal BST) is also known as a **weight-balanced search tree**. It is a binary search tree that provides the shortest possible search time or expected search time. An optimal binary search tree is helpful in dictionary search.