VTU – Jan 2025

# Solution

| Sub | Data Structures and Applications | | | | | Sub code | BCS304 |
|---|---|---|---|---|---|---|---|
| Date | 21/1/25 | Duration | **180 mins** | Max Marks | **50** | Sem /Sec | III A, B&C |

1 a)**Define data structures. explain the classification of data structures with a neat diagram.**

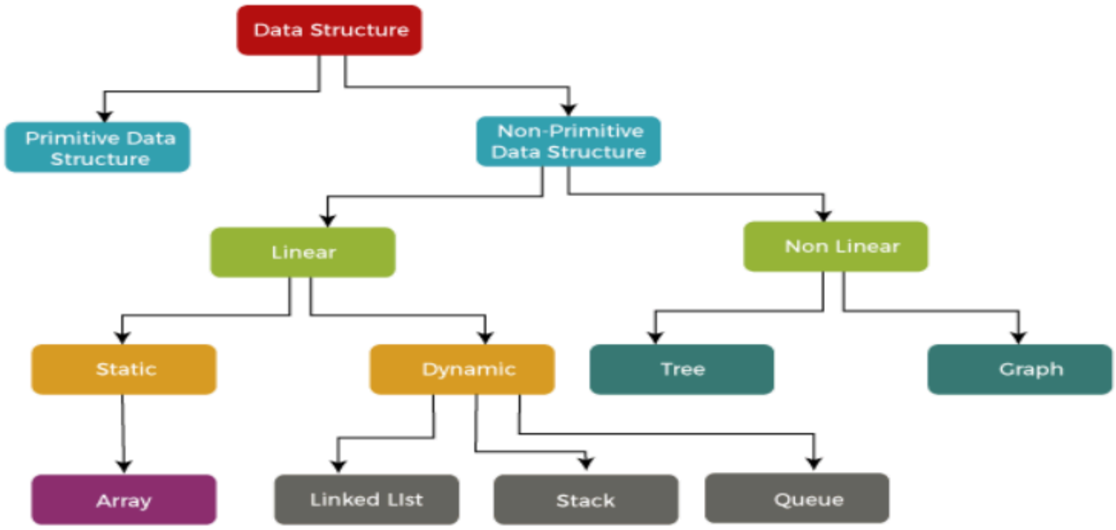
**Solution:**

A **data structure** is a way of organizing, managing, and storing data in a computer so it can be used efficiently. Data structures allow data to be arranged in a way that enables easy access, modification, and processing.

## Classification of Data Structures

Data structures can be broadly classified into two main types:

1. **Primitive Data Structures**
2. **Non-Primitive Data Structures**




**1. Primitive Data Structures**

Primitive data structures are the basic data types provided by programming languages, such as integers, floats, characters, and booleans. These types hold a single value and are usually built into the language.

**2. Non-Primitive Data Structures**

Non-primitive data structures are more complex and are used to store multiple values in a single structure. They are divided into two main categories: **linear** and **non-linear** data structures.

**A. Linear Data Structures**

- **Arrays:** A collection of elements, each identified by an index or key. Elements are stored in contiguous memory locations, and all elements are of the same type.
  - *Example:* `[10, 20, 30, 40]`
- **Linked Lists:** A sequence of elements called nodes, where each node contains a value and a reference to the next node. Unlike arrays, elements are not stored in contiguous memory locations.
  - *Example:* `10 -> 20 -> 30 -> 40`
- **Stacks:** A collection of elements that follows the Last-In-First-Out (LIFO) principle. Operations are performed at only one end of the structure (top of the stack).
  - *Example:* A stack of plates where only the top plate is accessible.
- **Queues:** A collection of elements that follows the First-In-First-Out (FIFO) principle. Elements are added at one end (rear) and removed from the other end (front).
  - *Example:* A line of people waiting to buy tickets, where the person at the front of the line is served first.

**B. Non-Linear Data Structures**

- **Trees:** A hierarchical data structure consisting of nodes, where each node has a value and references to child nodes. Trees are commonly used for data that has a natural hierarchy, like file directories.
  - *Example:* A binary tree representing a family tree, with each node representing a family member.
- **Graphs:** A collection of nodes (vertices) connected by edges. Graphs are used to represent networks and relationships, such as social networks or web page links.
  - *Example:* A social network graph where each person is a node, and an edge represents a friendship.

**b)Write a C function to implement pop, push and display operations for stacks using Arrays.**
**Solution:**

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

#define MAX 5  // Maximum size of the stack

int stack[MAX], top = -1;  // Stack and top pointer

// Function to push an element onto the stack
void push(int value) {
   if (top == MAX - 1) {
      printf("Stack Overflow! Cannot push %d\n", value);
      return;
   }
   stack[++top] = value;
   printf("%d pushed onto the stack\n", value);
}

// Function to pop an element from the stack
void pop() {
   if (top == -1) {
```

```
        printf("Stack Underflow! Cannot pop\n");
        return;
    }
    printf("%d popped from the stack\n", stack[top--]);
}

// Function to display the stack elements
void display() {
    if (top == -1) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = top; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

// Main function to test stack operations
int main() {
    int choice, value;
    while (1) {
        printf("\nStack Operations:\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    }
    return 0;
}
```

**c)Differentiate structures and union.**

**Solution:**

The below table lists the primary differences between the C structures and unions:

| Parameter | Structure | Union |
|---|---|---|
| Definition | A structure is a user-defined data type that groups different data types into a single entity. | A union is a user-defined data type that allows storing different data types at the same memory location. |
| Keyword | The keyword **struct** is used to define a structure | The keyword **union** is used to define a union |
| Size | The size is the sum of the sizes of all members, with padding if necessary. | The size is equal to the size of the largest member, with possible padding. |
| Memory Allocation | Each member within a structure is allocated unique storage area of location. | Memory allocated is shared by individual members of union. |
| Data Overlap | No data overlap as members are independent. | Full data overlap as members shares the same memory. |
| Accessing Members | Individual member can be accessed at a time. | Only one member can be accessed at a time. |

2 a)Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 62/3-4/2*+.

**Solution:**

A postfix expression (also called Reverse Polish Notation) is evaluated using a stack.

Algorithm:

1. Initialize an empty stack to store operands.
2. Scan the postfix expression from left to right.
   - If the symbol is an operand, push it onto the stack.
   - If the symbol is an operator (`+`, `-`, `*`, `/`):
     1. Pop the top two elements from the stack.
     2. **Apply the operator:**
        `result = operand1 operator operand2`
     3. Push the result back onto the stack.
3. After scanning the expression, the stack will contain a single element, which is the final result.

**Step-by-step Evaluation**

| Symbol | Stack Action | Stack Status |
|---|---|---|
| 6 | Push 6 | [6] |
| 2 | Push 2 | [6, 2] |
| `/` | Pop 6, 2 → `6 / 2 = 3` | [3] |
| 3 | Push 3 | [3, 3] |
| `-` | Pop 3, 3 → `3 - 3 = 0` | [0] |
| 4 | Push 4 | [0, 4] |
| `/` | Pop 0, 4 → `0 / 4 = 0` | [0] |
| 2 | Push 2 | [0, 2] |
| `*` | Pop 0, 2 → `0 * 2 = 0` | [0] |
| `+` | Pop 0, 0 → `0 + 0 = 0` | [0] |

✅ **Final Answer:** `0`

**b)Explain the dynamic memory allocation function in detail.**

**Solution**:

### What is Dynamic Memory Allocation?

Dynamic memory allocation refers to the process of allocating memory **at runtime** rather than at compile time. This is useful when the amount of memory required is not known in advance. It allows programs to efficiently use memory by allocating and deallocating it as needed.

### Functions Used for Dynamic Memory Allocation

C provides four standard library functions for dynamic memory allocation, all of which are defined in the **`stdlib.h`** header file:

1. **`malloc()`** → Allocates memory but does not initialize it.
2. **`calloc()`** → Allocates and initializes memory with zeros.
3. **`realloc()`** → Resizes previously allocated memory.
4. **`free()`** → Deallocates memory to prevent memory leaks.

### 1. `malloc()` (Memory Allocation)

- Allocates a **single block** of memory of the specified size (in bytes).
- Returns a **void pointer** (`void *`), which needs to be typecast to the desired data type.
- Memory **is not initialized**, so it may contain garbage values.

- If the allocation fails, it returns `NULL`.

Syntax:
void* malloc(size_t size);

### 2. `calloc()` (Contiguous Allocation)

- Allocates **multiple blocks** of memory and initializes them to **zero**.
- Returns a **void pointer** (`void *`).
- If the allocation fails, it returns `NULL`.

Syntax:
void* calloc(size_t num, size_t size);

### 3. `realloc()` (Reallocation)

- Resizes an already allocated memory block.
- Can expand or shrink the memory.
- If the new size is larger, extra memory **may contain garbage values**.
- If the reallocation fails, it returns `NULL` and preserves the original block.

Syntax:
void* realloc(void *ptr, size_t new_size);

### 4. `free()` (Deallocation)

- Releases dynamically allocated memory back to the system.
- Prevents **memory leaks** (memory that is allocated but never freed).
- After freeing, the pointer **becomes a dangling pointer** (points to invalid memory), so it's good practice to set it to `NULL`.

Syntax:
void free(void *ptr);

**c) What is a sparse matrix? Give the triplet form of the given matrix and find its transpose.**

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

**Solution:**

# Sparse Matrix

A sparse matrix is a matrix that has a large number of zero elements compared to non-zero elements. In other words, if the majority of elements in a matrix are zero, it is considered a sparse matrix.

### Step 1: Triplet Representation

In triplet representation, we store only the non-zero elements along with their row and column indices.

| Row | Col | Value |
|---|---|---|
| 0 | 2 | 3 |
| 0 | 4 | 4 |
| 1 | 2 | 5 |
| 1 | 3 | 7 |
| 3 | 1 | 2 |
| 3 | 2 | 6 |

So, the **triplet form** of the given matrix is:

$$\begin{bmatrix} Row & Col & Value \\ 0 & 2 & 3 \\ 0 & 4 & 4 \\ 1 & 2 & 5 \\ 1 & 3 & 7 \\ 3 & 1 & 2 \\ 3 & 2 & 6 \end{bmatrix}$$

### Transpose of the Matrix:

The transpose of a matrix is obtained by swapping rows and columns (i.e., `A[i][j]` becomes `A[j][i]`).So, in triplet form, we swap the Row and Col values:

| Col | Row | Value |
|---|---|---|
| 2 | 0 | 3 |
| 4 | 0 | 4 |
| 2 | 1 | 5 |
| 3 | 1 | 7 |
| 1 | 3 | 2 |
| 2 | 3 | 6 |

Sorting by **Row (Col in the original matrix order)** gives:

$$\begin{bmatrix} Row & Col & Value \\ 1 & 3 & 2 \\ 2 & 0 & 3 \\ 2 & 1 & 5 \\ 2 & 3 & 6 \\ 3 & 1 & 7 \\ 4 & 0 & 4 \end{bmatrix}$$

## Final Answer

### Triplet Form

$$\begin{bmatrix} Row & Col & Value \\ 0 & 2 & 3 \\ 0 & 4 & 4 \\ 1 & 2 & 5 \\ 1 & 3 & 7 \\ 3 & 1 & 2 \\ 3 & 2 & 6 \end{bmatrix}$$

### Transpose in Triplet Form

$$\begin{bmatrix} Row & Col & Value \\ 1 & 3 & 2 \\ 2 & 0 & 3 \\ 2 & 1 & 5 \\ 2 & 3 & 6 \\ 3 & 1 & 7 \\ 4 & 0 & 4 \end{bmatrix}$$

3 a)**Define queue. Discuss how to represent a queue using dynamic Array.**
**Solution:**

A **queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle. This means that elements are inserted at one end (**rear**) and removed from the other end (**front**).

### Basic Queue Operations

1. **Enqueue (Insertion)** – Adds an element at the rear.
2. **Dequeue (Deletion)** – Removes an element from the front.
3. **Front (Peek)** – Retrieves the front element without removing it.
4. **isEmpty** – Checks if the queue is empty.
5. **isFull** – Checks if the queue is full (in a fixed-size array representation).

**Representation of a Queue Using a Dynamic Array**

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

int *queue;   // Dynamic array for the queue
```

```c
int front = 0, rear = 0, capacity = 0;

// Function to create a queue with initial capacity
void createQueue(int size) {
    capacity = size;
    queue = (int*)malloc(capacity * sizeof(int));
}

// Function to resize the queue dynamically
void resizeQueue() {
    capacity *= 2;  // Double the capacity
    queue = (int*)realloc(queue, capacity * sizeof(int));
    printf("Queue resized, new capacity: %d\n", capacity);
}

// Function to add an element to the queue (enqueue)
void enqueue(int value) {
    if (rear == capacity) {
        resizeQueue();  // Resize if the queue is full
    }
    queue[rear++] = value;  // Add element and increment rear
    printf("Enqueued: %d\n", value);
}

// Function to remove an element from the queue (dequeue)
int dequeue() {
    if (front == rear) {
        printf("Queue is empty, cannot dequeue.\n");
        return -1;
    }
    int dequeuedValue = queue[front++];
    printf("Dequeued: %d\n", dequeuedValue);
    return dequeuedValue;
}

// Function to get the front element without removing it (peek)
int peek() {
    if (front == rear) {
        printf("Queue is empty, nothing to peek.\n");
        return -1;
    }
    return queue[front];
}
```

```
// Function to display the queue
void displayQueue() {
    if (front == rear) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i < rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

// Driver code
int main() {
    createQueue(5);  // Initial capacity of 5

    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    enqueue(50);
    enqueue(60);  // This will trigger resizing

    displayQueue();

    dequeue();
    displayQueue();

    printf("Front element: %d\n", peek());

    return 0;
}
```

**b)**Write a C function to implement insertion(), deletion() and display() operations on circular queue.

**Solution:**

```
#include <stdio.h>

#define SIZE 5  // Define the maximum size of the queue

int queue[SIZE];
int front = -1;
```

```
int rear = -1;

// Check if the queue is full
int isFull() {
   if ((front == 0 && rear == SIZE - 1) || (rear == (front - 1) % (SIZE - 1))) // rear just behind
the front
 {
      return 1;
   }
   return 0;
}

// Check if the queue is empty
int isEmpty() {
   if (front == -1) {
      return 1;
   }
   return 0;
}

// Add an element to the queue (enqueue)
void enqueue(int value) {
   if (isFull()) {
      printf("Queue is full. Cannot enqueue %d\n", value);
      return;
   }

   if (front == -1) {
      front = rear = 0;
   } else {
      rear = (rear + 1) % SIZE;
   }

   queue[rear] = value;
   printf("Enqueued %d\n", value);
}

// Remove an element from the queue (dequeue)
int dequeue() {
   if (isEmpty()) {
      printf("Queue is empty. Cannot dequeue.\n");
      return -1;
   }
```

```c
    int data = queue[front];

    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % SIZE;
    }

    printf("Dequeued %d\n", data);
    return data;
}

// Display the elements of the queue along with status
void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Queue elements: ");
    int i = front;
    int count = 0;

    // Traverse the queue to print the elements
    while (i != rear) {
        printf("%d ", queue[i]);
        i = (i + 1) % SIZE;    // size when it goes beyond the limit then comes back to 0
        count++;
    }
    printf("%d\n", queue[rear]);  // Print the last element
    count++;  // Including the rear element

    // Print the status
    printf("************Status is :***************\n");
    printf("Front index: %d\n", front);
    printf("Rear index: %d\n", rear);
    printf("Number of elements(status): %d\n", count);
}

int main() {
    int choice, value;

    while (1) {
```

```c
        printf("\nCircular Queue Operations:\n 1. Enqueue\n 2. Dequeue\n 3. Display\n 4.
Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                display();
                break;
            case 2:
                dequeue();
                display();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting...\n");
                return 0;
            default:
                printf("Invalid choice. Please enter again.\n");
        }
    }

    return 0;
}
```

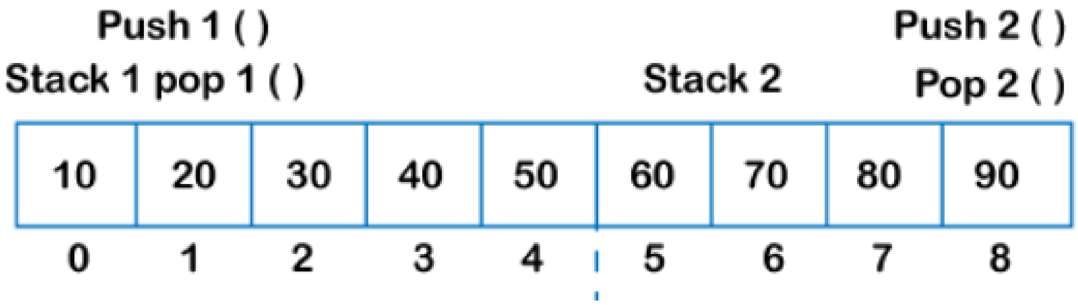
**c) Write a note on multiple stacks and queues with suitable diagrams.**
**Solution:**

In some applications, we may need to maintain multiple stacks or queues within a single array to optimize space and improve efficiency. This approach is particularly useful in memory-constrained environments, such as embedded systems or applications requiring multiple independent data structures.

# Multiple Stacks in a Single Array

Instead of creating separate arrays for multiple stacks, we can use a single array and divide it into multiple stack sections. This helps in reducing memory wastage.

example:

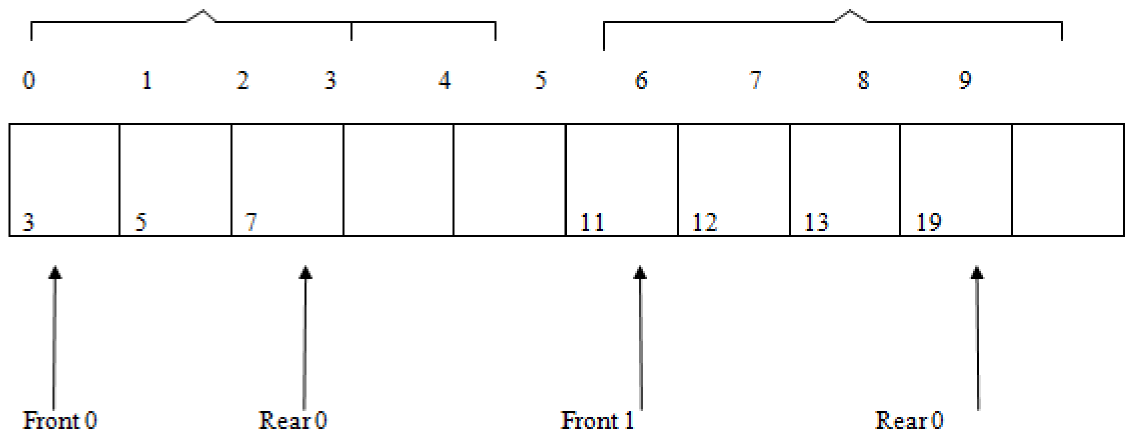



### Fixed Division Approach

- The array is **equally divided** into multiple stack segments.
- Each stack has its own **top pointer** to track the top element.
- This method is **easy to implement** but can lead to **wastage of space** if some stacks are underutilized.

## Multiple Queues in a Single Array

Like multiple stacks, we can implement **multiple queues** in a single array by using different approaches.




### Fixed Division Approach

- The array is **equally divided** among multiple queues.
- Each queue has its own **front** and **rear** pointers.
- Simple to implement, but **wasteful if one queue is underutilized**.

### Circular Queue Approach

- A **single array** is used for multiple queues with **wrap-around** indexing.
- Front and Rear **circulate within** the available space.
- Efficient memory utilization compared to fixed partitioning.

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

**Solution:**

**Linked List:**

A linked list is a linear data structure where elements (nodes) are connected using pointers instead of being stored in contiguous memory like arrays. Each node consists of two parts:

1. Data – Stores the actual information.
2. Pointer (Next) – Stores the memory address of the next node.

Unlike arrays, linked lists can dynamically allocate memory, making them efficient for insertions and deletions.

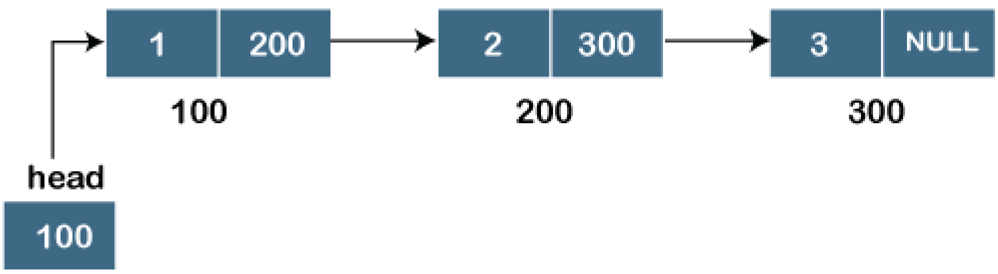
**Types:**

**The following are the types of linked list:**

- **Singly Linked list**
- **Doubly Linked list**
- **Circular Linked list**
- **Doubly Circular Linked list**

**Singly Linked list**

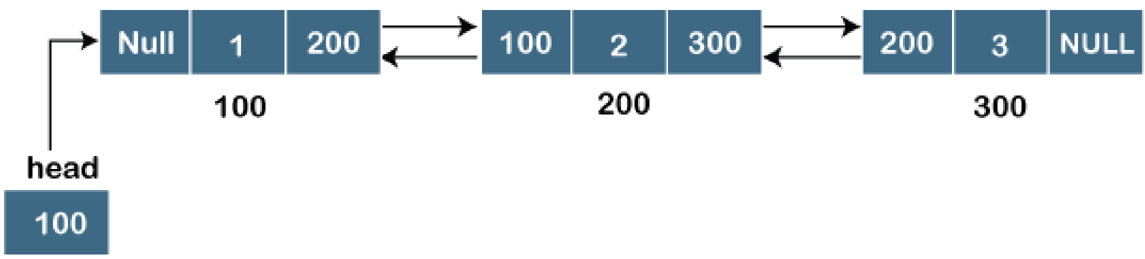
Each node points to the next node in the sequence. The last node points to `NULL`.




**Doubly Linked List:**

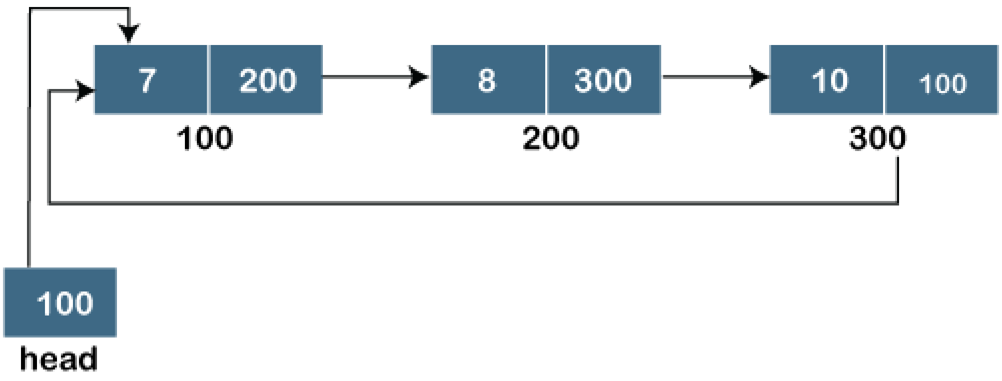
Each node has **two pointers**:

- One pointing to the **next node**
- One pointing to the **previous node**



**Circular Linked List:**

In a Singly Circular Linked List, the last node points back to the first node instead of NULL.
In a Doubly Circular Linked List, both the first and last nodes are connected to form a loop.




**Doubly Circular Linked List:**

A Doubly Circular Linked List (DCLL) is a type of linked list where:

1. Each node has two pointers – one pointing to the next node and another pointing to the previous node.
2. The last node connects back to the first node, forming a circular structure.

b) **Insert a node at the beginning of a singly linked list:**

```
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = newData;
    newNode->next = *head;
    *head = newNode;
}
```

2. **Delete a node at the front of a singly linked list:**

c
CopyEdit

```
void deleteAtFront(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;
```

```
    *head = (*head)->next;
    free(temp);
}
```

3. **Display the singly linked list (for testing purposes):**

c
CopyEdit
```c
void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

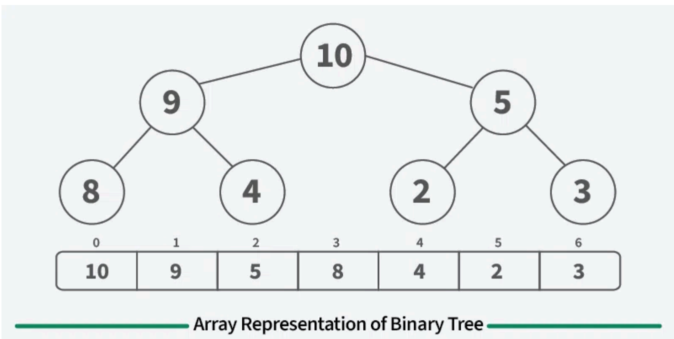
5

a) Discuss how binary trees are represented using i) array ii) Linked List

Solution:

i) Array Representation is 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**




**Advantages:**

- Easy to navigate parent and child nodes using index calculations, which is fast
- Easier to implement, especially for complete binary trees.

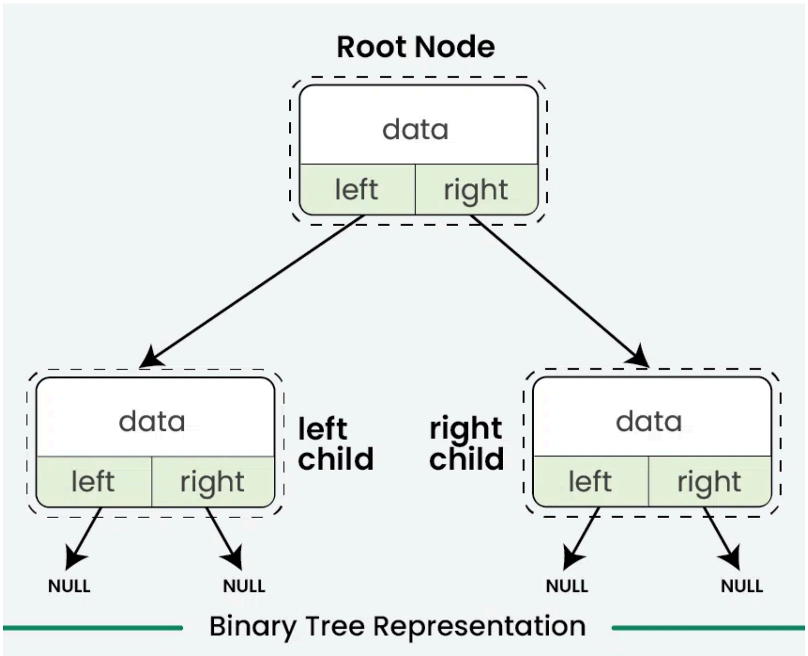
**Disadvantages:**

- You have to set a size in advance, which can lead to wasted space.
- If the tree is not complete binary tree then then many slots in the array might be empty, this will result in wasting memory
- Not as flexible as linked representations for dynamic trees.

ii) Linked List 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.

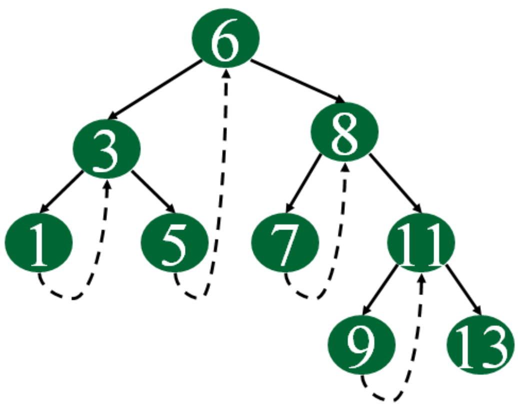
b) Define threaded binary tree. Discuss in threaded binary tree.

Solution:

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.




**Advantages of Threaded Binary Tree**

- In this Tree it enables linear traversal of elements.
- It eliminates the use of stack as it perform linear traversal, so save memory.
- Enables to find parent node without explicit use of parent pointer
- Threaded tree give forward and backward traversal of nodes by in-order fashion
- Nodes contain pointers to in-order predecessor and successor
- For a given node, we can easily find inorder predecessor and successor. So, searching is much more easier.
- In threaded binary tree there is no NULL pointer present. Hence memory wastage in occupying NULL links is avoided.

- The threads are pointing to successor and predecessor nodes. This makes us to obtain predecessor and successor node of any node quickly.
- There is no need of stack while traversing the tree, because using thread links we can reach to previously visited nodes.

**Disadvantages of Threaded Binary Tree**

- Every node in threaded binary tree need extra information(extra memory) to indicate whether its left or right node indicated its child nodes or its inorder predecessor or successor. So, the node consumes extra memory to implement.
- Insertion and deletion are way more complex and time consuming than the normal one since both threads and ordinary links need to be maintained.
- Implementing threads for every possible node is complicated.
- Increased complexity: Implementing a threaded binary tree requires more complex algorithms and data structures than a regular binary tree. This can make the code harder to read and debug.
- Extra memory usage: In some cases, the additional pointers used to thread the tree can use up more memory than a regular binary tree. This is especially true if the tree is not fully balanced, as threading a skewed tree can result in a large number of additional pointers.
- Limited flexibility: Threaded binary trees are specialized data structures that are optimized for specific types of traversal. While they can be more efficient than regular binary trees for these types of operations, they may not be as useful in other scenarios. For example, they cannot be easily modified (e.g. inserting or deleting nodes) without breaking the threading.

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

### 1. Inorder Traversal (Left → Root → Right)

- First, visit the left subtree.
- Then, visit the root node.
- Finally, visit the right subtree.
- Used in BSTs to get elements in sorted order.

**C Function:**

c

CopyEdit

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

---

**2. Preorder Traversal (Root → Left → Right)**

- First, visit the root node.
- Then, visit the left subtree.
- Finally, visit the right subtree.
- Used for tree cloning and expression evaluation.

**C Function:**

c

CopyEdit

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

---

### 3. Postorder Traversal (Left → Right → Root)

- First, visit the left subtree.
- Then, visit the right subtree.
- Finally, visit the root node.
- Used for deleting a tree or evaluating postfix expressions.

**C Function:**

c

CopyEdit

```
void postorderTraversal(struct Node* root) {

    if (root == NULL)

        return;

    postorderTraversal(root->left);

    postorderTraversal(root->right);

    printf("%d ", root->data);

}
```

---

### 4. Level Order Traversal (Breadth-First Search - BFS)

- Visit nodes level by level from left to right.
- Uses a queue to process nodes in FIFO order.
- Used in shortest path algorithms and tree breadth analysis.

**C Function:**

c

CopyEdit

```
#include <stdio.h>

#include <stdlib.h>



void levelOrderTraversal(struct Node* root) {
```

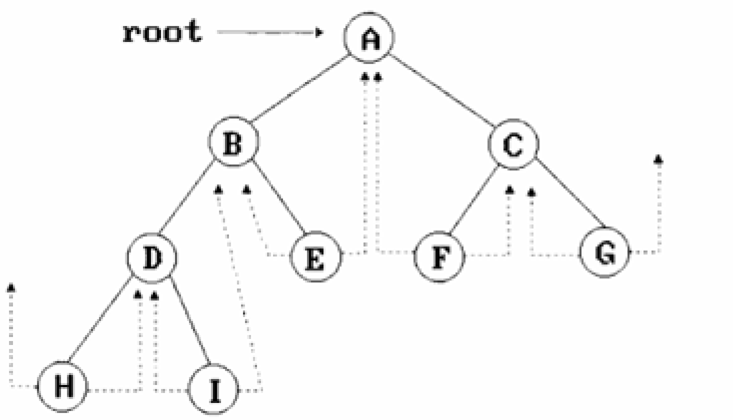
```
    if (root == NULL) return;

    struct Queue* q = createQueue();
    enqueue(q, root);

    while (!isEmpty(q)) {
        struct Node* temp = dequeue(q);
        printf("%d ", temp->data);
        if (temp->left) enqueue(q, temp->left);
        if (temp->right) enqueue(q, temp->right);
    }
}
```

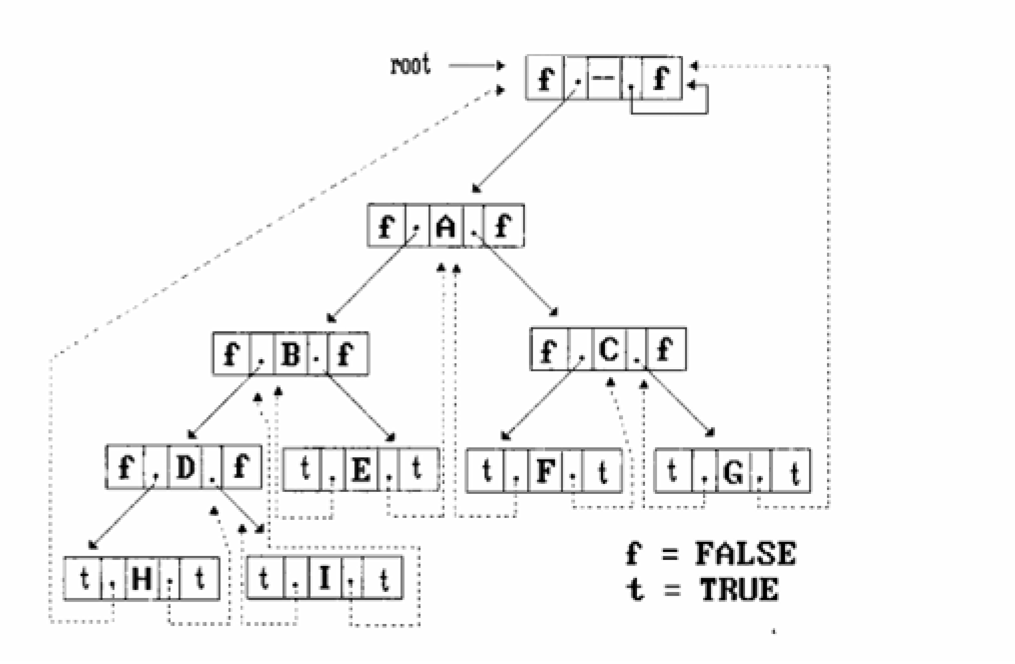
6.b

Threaded Binary Tree

(1) If *ptr -> left_child* is null, replace *ptr -> left_child* with a pointer to the node that would be visited before *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder predecessor* of *ptr*.

(2) If *ptr -> right_child* is null, replace *ptr -> right_child* with a pointer to the node that would be visited after *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder successor* of *ptr*.



|  |  |
| --- | --- |
| | <br>Memory Representation of Threaded Binary Tree |
| 6c | **i) Insert a node at the beginning of a doubly linked list**<br>c<br>CopyEdit |

```
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = *head;

    if (*head != NULL)
        (*head)->prev = newNode;

    *head = newNode;
}
```

**ii) Deleting a node at the end of the doubly linked list**

c

CopyEdit

```c
void deleteAtEnd(struct Node** head) {
    if (*head == NULL)
        return;

    struct Node* temp = *head;

    while (temp->next != NULL)
        temp = temp->next;

    if (temp->prev != NULL)
        temp->prev->next = NULL;
    else
        *head = NULL;

    free(temp);
}
```

**Q.7**

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

**b. Define Binary Search Tree. Construct a binary search tree for the following elements:**
**100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145.**

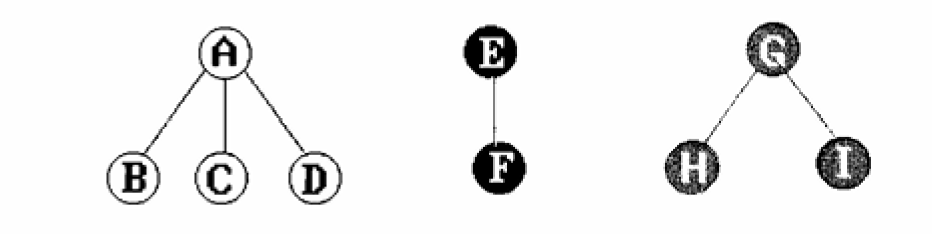
**c. Discuss Selection Tree with an example.**

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

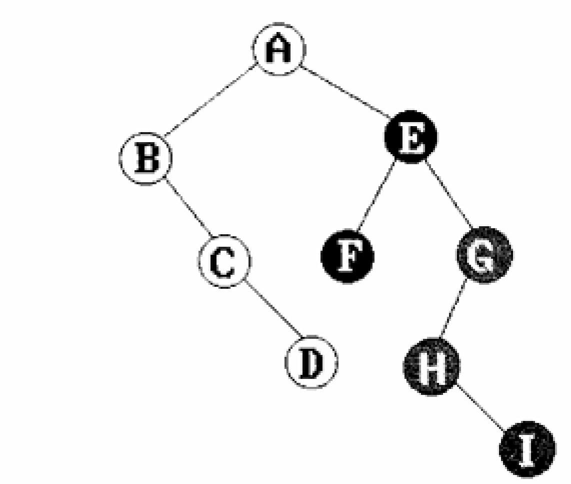
**Definition of Forest**
A forest is a collection of disjoint trees. In simpler terms, it is a set of multiple trees where each tree consists of a root and its descendants but is independent of other trees in the forest.

Transforming a Forest into a Binary Tree
To convert a forest into a binary tree, we use the left-child right-sibling (LCRS) representation:

- The leftmost child of a node is linked as its left child.
- The next sibling of the node is linked as its right child.
- The process is applied recursively for all nodes in the forest.




***Transforming a Forest into a Binary Tree:***

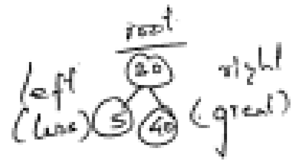


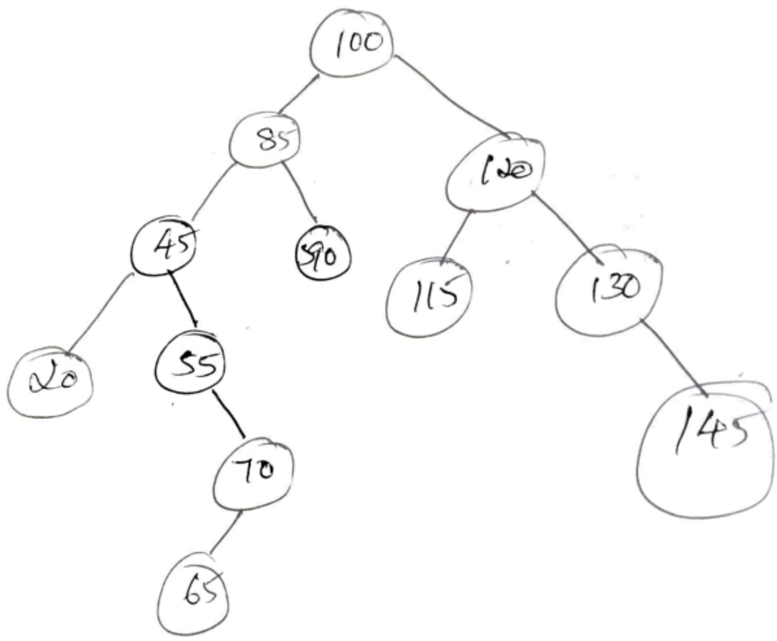
**b. Define Binary Search Tree. Construct a binary search tree for the following elements:**
**100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145.**

## 4.8 Binary Search Tree:

1) Definition: * It is a binary tree and it can be empty and if it is not empty, it satisfies the following properties.

* (i) Each node has exactly one key and the key in the trees are distinct.
* (ii) The key in the left subtree are smaller than the key in the root.
* (iii) The key in the right subtree are larger than the key in the root.
* (iv) The left and right subtree are also binary search trees.



**c. Discuss Selection Tree with an example.**

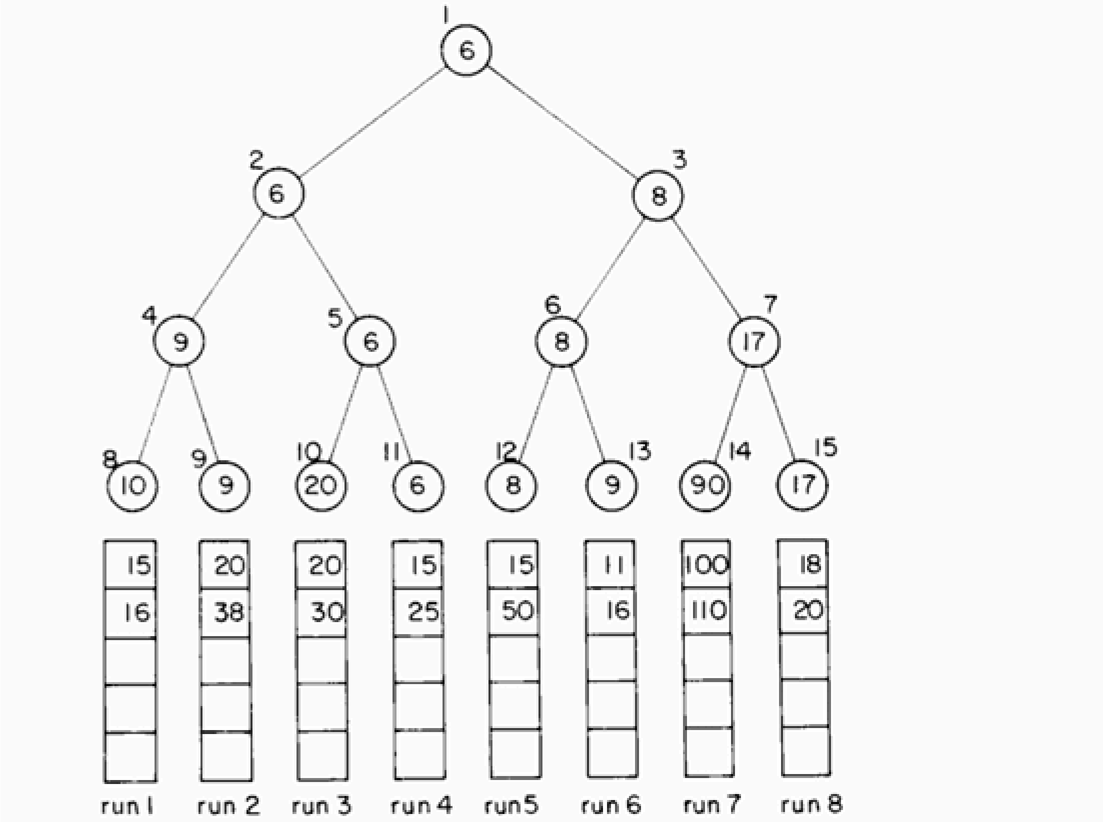
* Assume k ordered sequences called runs.
* Runs consists of records and the keys of the records are in non-decreasing order.
* Merging is done by repeatedly outputting records with the smallest key.
* This is achieved by using by a selection tree by reducing the number of comparisons needed to find the next smallest element.

Two kinds of selection Trees:

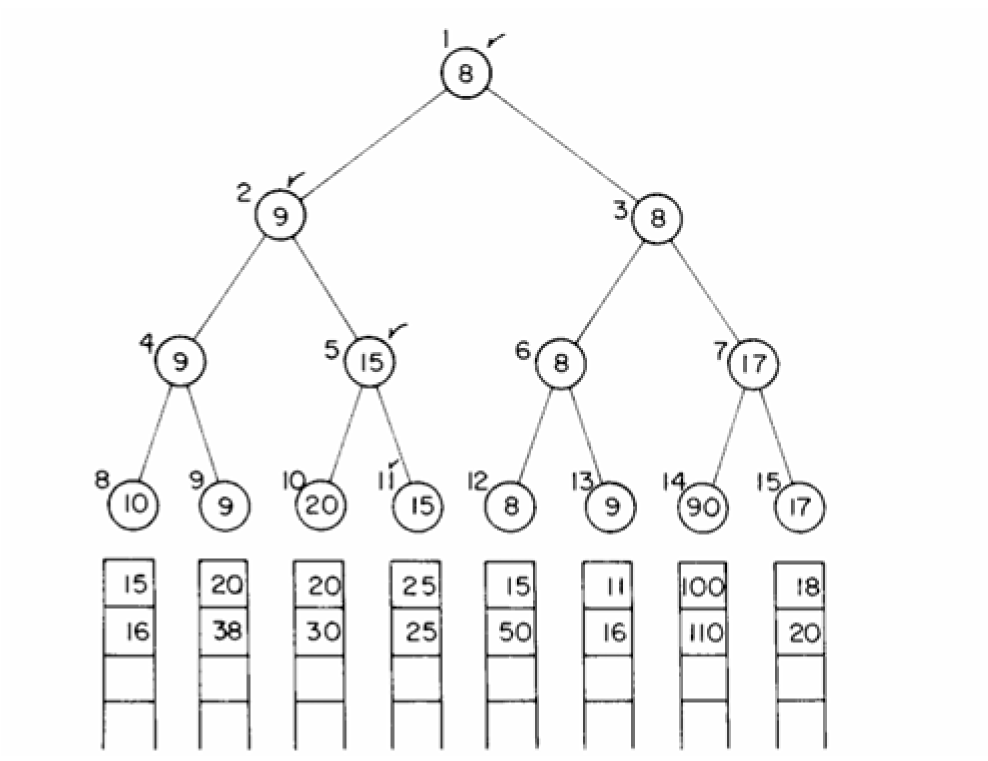
1. Winner Tree
2. Loser Tree.

Winner Tree:

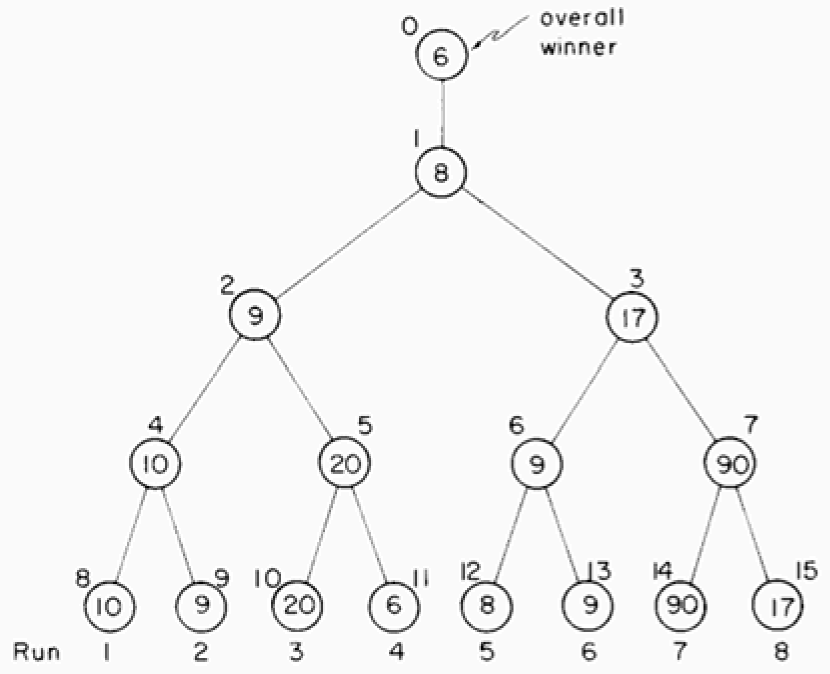
* A complete binary tree in which each node represent the smaller of its two children.
* The root represent the smallest node in the tree.



6
6
8
9
6
8
17
10
9
20
6
8
9
90
17
15 16
20 38
20 30
15 25
15 50
11 16
100 110
18 20
run 1
run 2
run 3
run 4
run5
run 6
run 7
run 8

8
9
8
9
15
8
17
10
9
20
15
8
9
90
17
15 16
20 38
20 30
25 25
15 50
11 16
100 110
18 20




'igure 5.36: Tree of losers corresponding to Figure 5.34

**Q8. a**

**Define Graph. Explain adjacency matrix and adjacency list representation with an example.**

A graph, $G$, consists of two sets: a finite, nonempty set of *vertices*, and a finite, possibly empty set of *edges*. $V(G)$ and $E(G)$ represent the sets of vertices and edges of $G$, respectively. Alternately, we may write $G = (V, E)$ to represent a graph.

An *undirected graph* is one in which the pair of vertices representing any edge is unordered. For example, the pairs $(v_0, v_1)$ and $(v_1, v_0)$ represent the same edge.

A *directed graph* is one in which we represent each edge as a directed pair of vertices. For example, the pair $<v_0, v_1>$ represents an edge in which $v_0$ is the *tail* and $v_1$ is the *head*. Therefore, $<v_0, v_1>$ and $<v_1, v_0>$ represent two different edges in a directed graph.

Figure 6.2 shows three sample graphs. We represent the vertices as circles numbered from 0 to $n - 1$, where $n$ is the number of vertices currently in use. For an undirected graph, we represent the edges as lines or curves. For a directed graph, we represent the edges as arrows, drawn from the tail to the head. Graphs $G_1$ and $G_2$ are undirected, while graph $G_3$ is a directed graph.

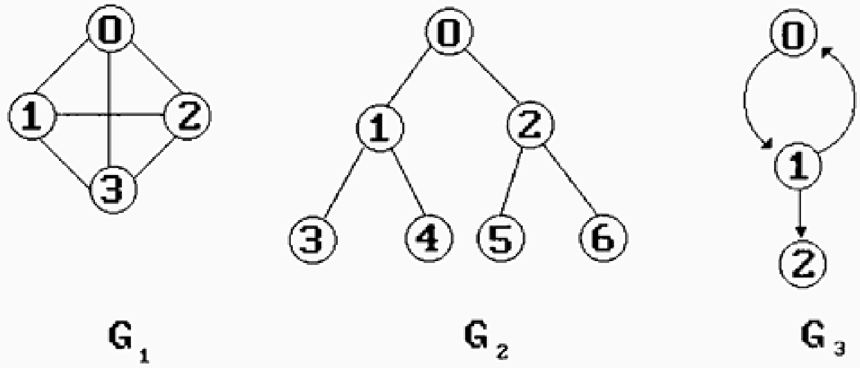



Figure 6.2: Three sample graphs

The set representation of each of these graphs is:

$V(G_1)$, = {0, 1, 2, 3}   $E(G_1)$, = {(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}
$V(G_2)$, = {0, 1, 2, 3, 4, 5, 6}   $E(G_2)$, = {(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)}
$V(G_3)$} = {0, 1, 2}   $E(G_3)$ = {<0, 1>, <1, 0>, <1, 2 >}

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

$G_1$

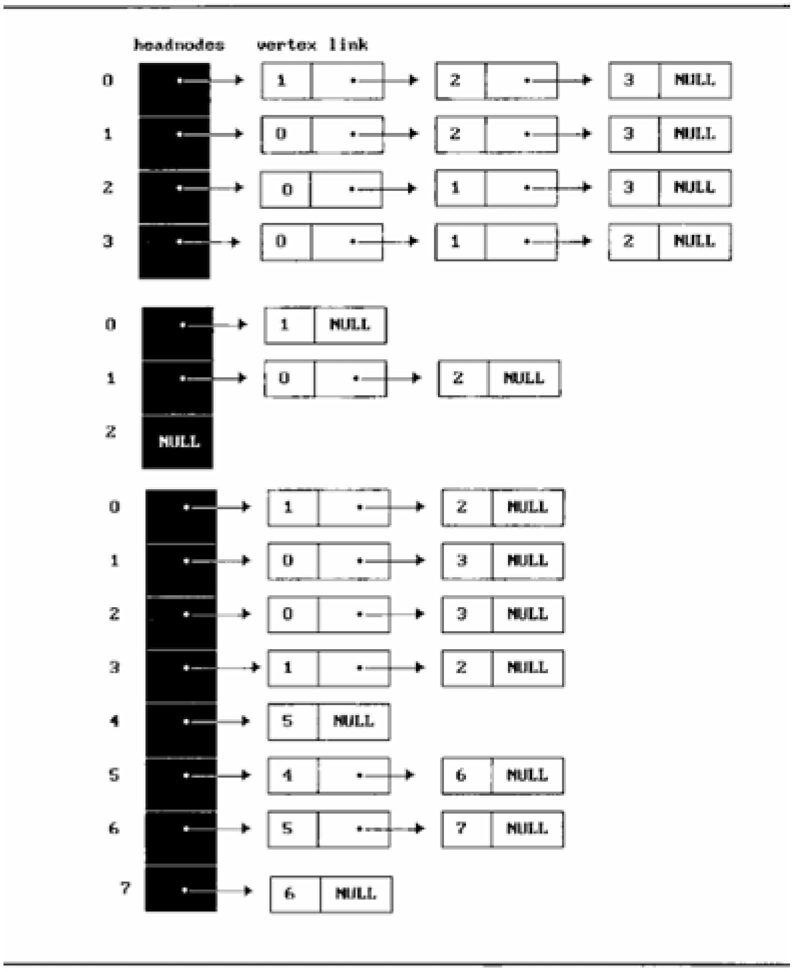
$$\begin{array}{c c} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

$G_3$

$$\begin{array}{c c} & \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$G_4$

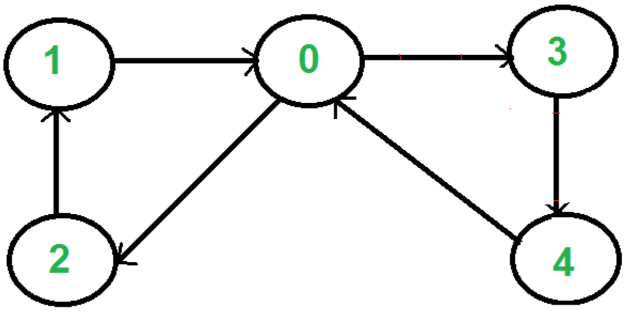
7: Adjacency matrices for $G_1$, $G_3$, and $G_4$



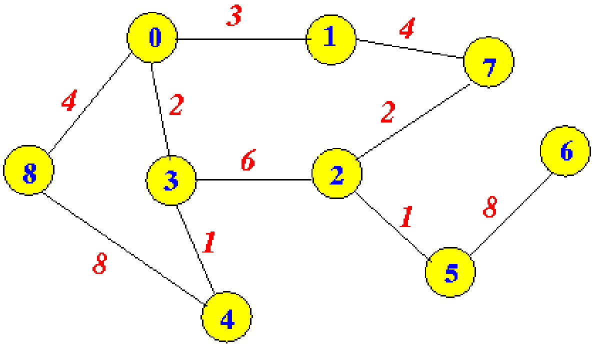

**6.8:** Adjacency lists for $G_1$, $G_3$, and $G_4$

**8.b** Define the following terminology with example: i) Digraph ii) Weighted graph iii) Self loop iv) Connected graph

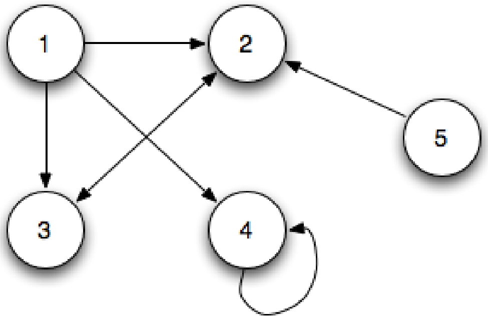
i) Digraph (Directed Graph):
A digraph (directed graph) is a graph in which the edges have a specific direction, meaning each edge points from one vertex to another.



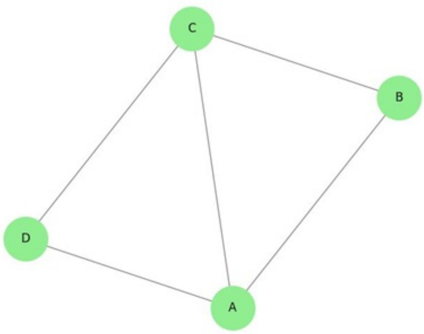

ii) Weighted Graph:
A weighted graph is a graph where each edge is assigned a numerical value (weight), which often represents cost, distance, or time.




iii) Self Loop:
A self loop is an edge that starts and ends at the same vertex, creating a direct connection of a node to itself.




iv) Connected Graph:
A connected graph is a graph where there exists at least one path between every pair of vertices, ensuring that no node is isolated.



| | |
|---|---|
| **8.c** | **c. Briefly explain about Elementary graph operations.** |

BFS and DFS

a. Breadth First Search

```
void bfs(int v)
{
/* breadth first traversal of a graph, starting with node v
the global array visited is initialized to 0, the queue
operations are similar to those described in
Chapter 4. */
   node_pointer w;
   queue_pointer front,rear;
   front = rear = NULL; /* initialize queue */
   printf("%5d",v);
   visited[v] = TRUE;
   addq(&front, &rear, v);
   while (front) {
      v = deleteq(&front);
      for (w = graph[v]; w; w = w->link)
         if (!visited[w->vertex]) {
            printf("%5d", w->vertex);
            addq(&front,&rear,w->vertex);
            visited[w->vertex] = TRUE;
         }
      }
}
```

b.

c. **Depth First Search**

```
void dfs(int v)
{
/* depth first search of a graph beginning with vertex v.*/
   node_pointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w->link)
      if (!visited[w->vertex])
         dfs(w->vertex);
}
```

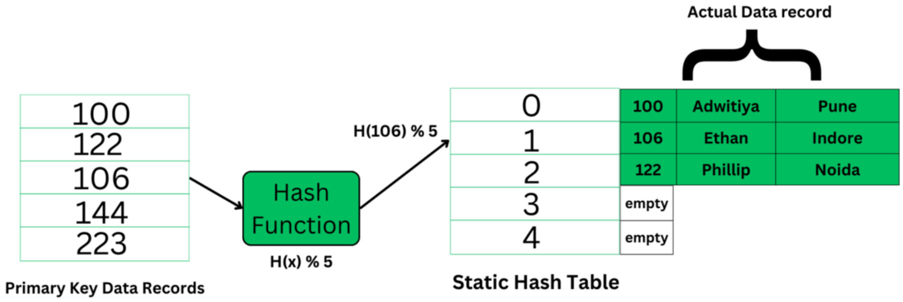
d.

**Q9** a. **Explain in detail about static and dynamic hashing.**

**Solution:**

1. **Static Hashing:**

In static hashing, the hash function always generates the same bucket's address. For example, if we have a data record for employee_id = 107, the hash function is mod-5 which is - H(x) % 5, where x = id. Then the operation will take place like this:
H(106) % 5 = 1.
This indicates that the data record should be placed or searched in the 1st bucket (or 1st hash index) in the hash table.




The primary key is used as the input to the hash function and the hash function generates the output as the hash index (bucket's address) which contains the address of the actual data record on the disk block.

Static Hashing has the following Properties

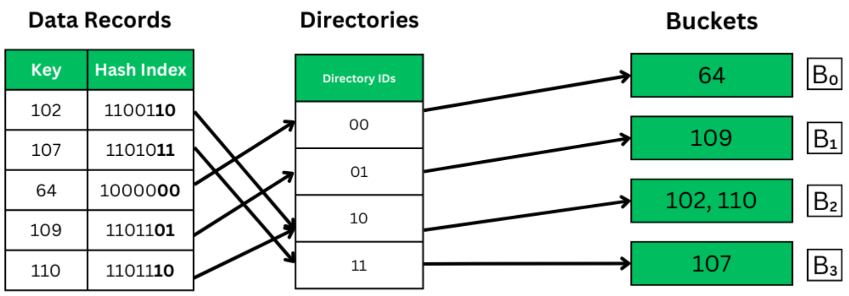
- Data Buckets: The number of buckets in memory remains constant. The size of the hash table is decided initially and it may also implement chaining that will allow handling some collision issues though, it's only a slight optimization and may not prove worthy if the database size keeps fluctuating.
- Hash function: It uses the simplest hash function to map the data records to its appropriate bucket. It is generally modulo-hash function
- Efficient for known data size: It's very efficient in terms when we know the data size and its distribution in the database.
- It is inefficient and inaccurate when the data size dynamically varies because we have limited space and the hash function always generates the same value for every specific input. When the data size fluctuates very often it's not at all useful because collision will keep happening and it will result in problems like - bucket skew, insufficient buckets etc.

**2. Dynamic Hashing**

Dynamic hashing is also known as extendible hashing, used to handle database that frequently changes data sets. This method offers us a way to add and remove data buckets on demand dynamically. This way as the number of data records varies, the buckets will also grow and shrink in size periodically whenever a change is made.

Properties of Dynamic Hashing

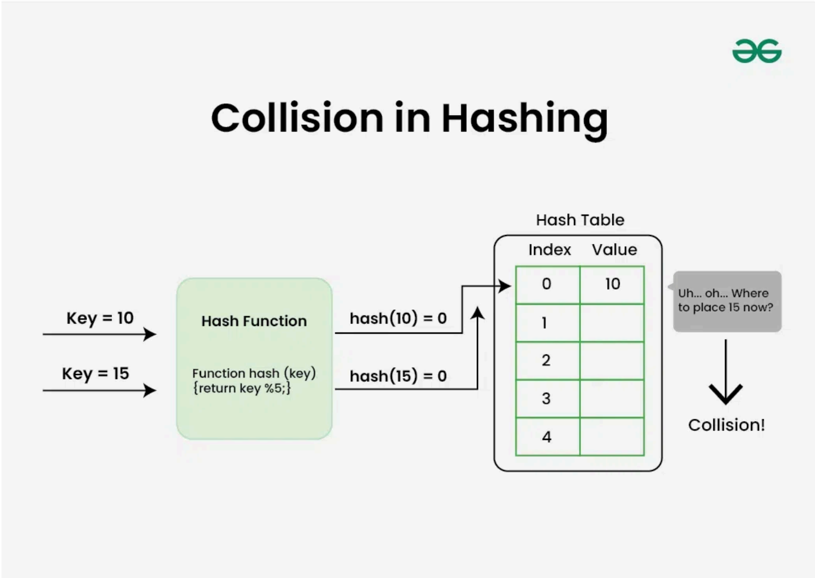
- The buckets will vary in size dynamically periodically as changes are made offering more flexibility in making any change.
- Dynamic Hashing aids in improving overall performance by minimizing or completely preventing collisions.
- It has the following major components: Data bucket, Flexible hash function, and directories
- A flexible hash function means that it will generate more dynamic values and will keep changing periodically asserting to the requirements of the database.
- Directories are containers that store the pointer to buckets. If bucket overflow or bucket skew-like problems happen to occur, then bucket splitting is done to maintain efficient retrieval time of data records. Each directory will have a directory id.
- Global Depth: It is defined as the number of bits in each directory id. The more the number of records, the more bits are there.




b. **What is collison? what are the methods to resolve collison?**

**Solution:**

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.

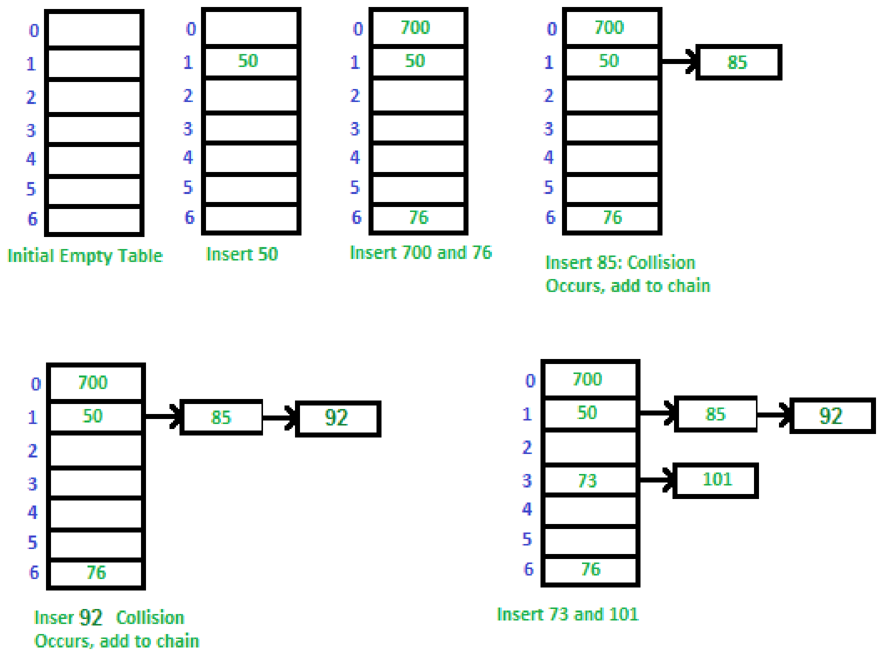



There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing

1) 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.




2) Open Addressing

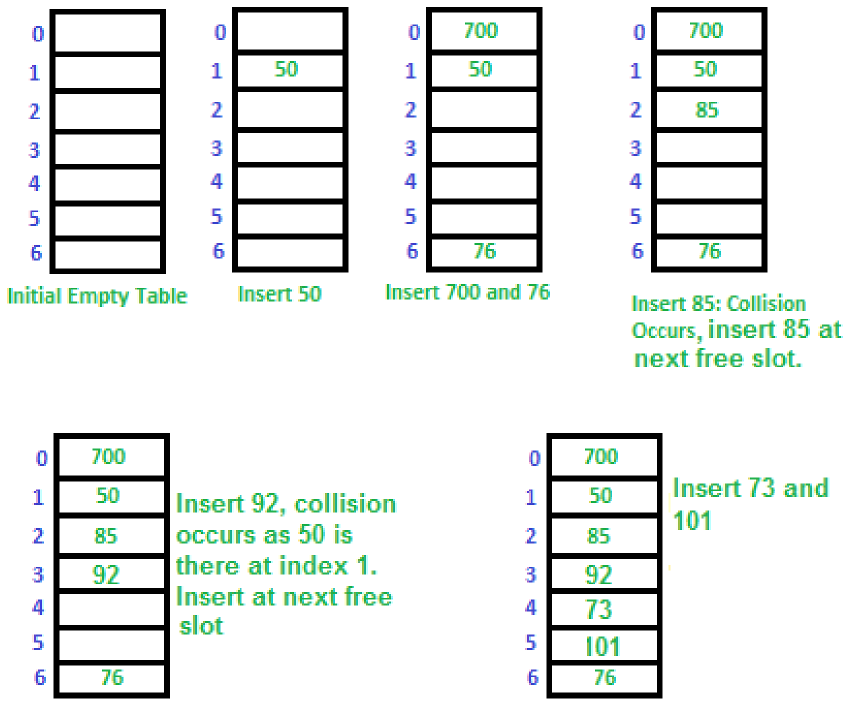
In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

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 examples.**

**Solution**:

A priority queue is a type of queue that arranges elements based on their priority values.

- Each element has a priority associated. When we add an item, it is inserted in a position based on its priority.
- Elements with higher priority are typically retrieved or removed before elements with lower priority.
- Binary heap is the most common method to implement a priority queue. 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 Queue is used in algorithms such as Dijkstra's algorithm, Prim's algorithm, and Huffman Coding.

For example, in the below priority queue, an element with a maximum ASCII value will have the highest priority. The elements with higher priority are served first.
Types of Priority Queue:
1) Ascending Order Priority Queue
As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in ascending order like 4,6,8,9,10. Here, 4 is the smallest number, therefore, it will get the highest priority in a priority queue and so when we dequeue from this type of priority queue, 4 will remove from the queue and dequeue returns 4.

2) Descending order Priority Queue
The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future. The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.
Operations of a Priority Queue:
A typical priority queue supports the following operations:
1) Insertion : If the newly inserted item is of the highest priority, then it is inserted at the top. Otherwise, it is inserted in such a way that it is accessible after all higher priority items are accessed.
2) Deletion in a Priority Queue : We typically remove the highest priority item which is typically available at the top. Once we remove this item, we need not move next priority item at the top.
3) Peek in a Priority Queue : This operation only returns the highest priority item (which is typically available at the top) and does not make any change to the priority queue.
There can be additional operations required like change priority and traverse all items.

Heaps are frequently used to implement *priority queues.* Unlike the queues we discussed in Chapter 3, a priority queue deletes the element with the highest (or the lowest) priority. At any time we can insert an element with arbitrary priority into a priority queue. If our application requires us to delete the element with the highest priority, we use a max heap.

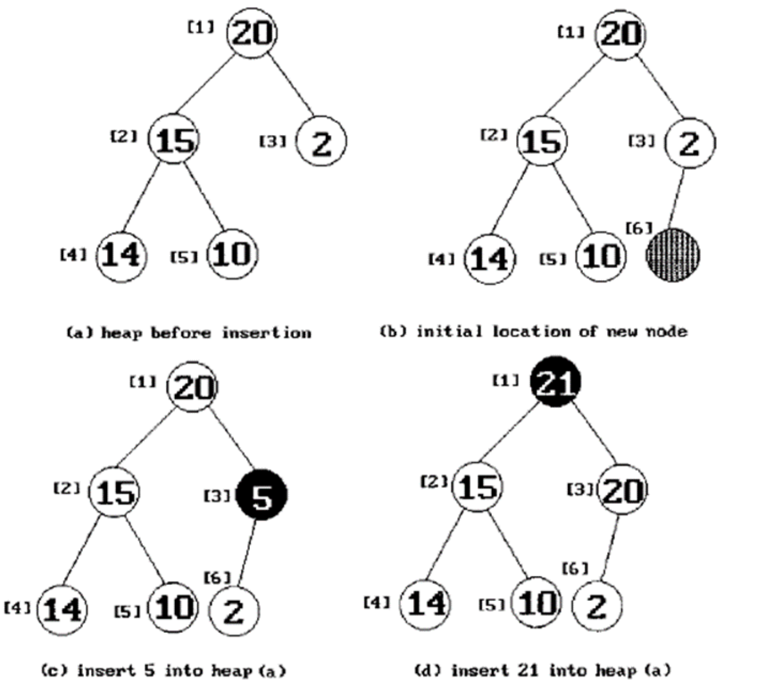



**Figure 5.28:** Insertion into a max heap

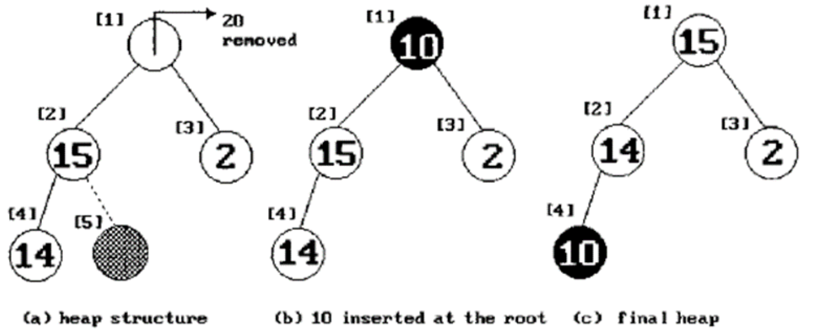



**Figure 5.29:** Deletion from a max heap

| | |
|---|---|
| **Q10.** | **a. Define hashing. Explain different hashing functions with suitable examples.** |

**Solution**:
**Hashing** refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.

# 1. Division Method

The division method involves dividing the key by a prime number and using the remainder as the hash value.

*h(k)=k mod m Where k is the key and mm is a prime number.*

**Advantages**:

- Simple to implement.
- Works well when *mm* is a prime number.

**Disadvantages**:

- Poor distribution if *mm* is not chosen wisely.

2. Multiplication Method

In the multiplication method, a constant *AA* ($0 < A < 1$) is used to multiply the key. The fractional part of the product is then multiplied by *mm* to get the hash value.

$h(k)=\lfloor m(kA \bmod 1)\rfloor$

*Where ⌊ ⌋ denotes the floor function.*

**Advantages**:

- Less sensitive to the choice of *mm*.

**Disadvantages**:

- More complex than the division method.

3. Mid-Square Method
In the mid-square method, the key is squared, and the middle digits of the result are taken as the hash value.

**Steps**:

1. Square the key.
2. Extract the middle digits of the squared value.

**Advantages**:

- Produces a good distribution of hash values.

**Disadvantages**:

- May require more computational effort.

4. Folding Method
The folding method involves dividing the key into equal parts, summing the parts, and then taking the modulo with respect to *mm*.

**Steps**:

1. Divide the key into parts.
2. Sum the parts.
3. Take the modulo *mm* of the sum.

**Advantages**:

- Simple and easy to implement.

**Disadvantages**:

- Depends on the choice of partitioning scheme.

5. Cryptographic Hash Functions
Cryptographic hash functions are designed to be secure and are used in cryptography. Examples include MD5, SHA-1, and SHA-256.

**Characteristics**:

- Pre-image resistance.
- Second pre-image resistance.
- Collision resistance.

**Advantages**:

- High security.

**Disadvantages**:

- Computationally intensive.

6. Universal Hashing

Universal hashing uses a family of hash functions to minimize the chance of collision for any given set of inputs.

$h(k)=((a \cdot k+b) \bmod p) \bmod m$

*Where a and b are randomly chosen constants, p is a prime number greater than m, and k is the key.*

**Advantages**:

- Reduces the probability of collisions.

**Disadvantages**:

- Requires more computation and storage.

7. Perfect Hashing

Perfect hashing aims to create a collision-free hash function for a static set of keys. It guarantees that no two keys will hash to the same value.

**Types**:

- Minimal Perfect Hashing: Ensures that the range of the hash function is equal to the number of keys.
- Non-minimal Perfect Hashing: The range may be larger than the number of keys.

**Advantages**:

- No collisions.

**Disadvantages**:

- Complex to construct.

**b. Write short notes on i. Leftist tree ii. Optimal Binary Search Tree**

**Solution:**

i. Leftist Tree:
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.

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an s-value (or rank or distance) which is the distance to the nearest leaf.A leftist tree is a binary tree with properties:

1. Normal Min Heap Property : key(i) >= key(parent(i))
2. Heavier on left side : dist(right(i)) <= dist(left(i)). Here, dist(i) is the number of edges on the shortest path from node i to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and dist( i ) = 1 + dist( right( i ) ).

ii. Optimal Binary Search Tree:

an **optimal binary search tree (Optimal BST)**, sometimes called a **weight-balanced binary tree**,[1] is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities). Optimal BSTs are generally divided into two types: static and dynamic.

In the **static optimality** problem, the tree cannot be modified after it has been constructed. In this case, there exists some particular layout of the nodes of the tree which provides the smallest expected search time for the given access probabilities. Various algorithms exist to construct or approximate the statically optimal tree given the information on the access probabilities of the elements.

In the **dynamic optimality** problem, the tree can be modified at any time, typically by permitting tree rotations. The tree is considered to have a cursor starting at the root which it can move or use to perform modifications. In this case, there exists some minimal-cost sequence of these operations which causes the cursor to visit every node in the target access sequence in order. The splay tree is conjectured to have a constant competitive ratio compared to the dynamically optimal tree in all cases, though this has not yet been proven.

- **Purpose**: To reduce the average time required to search for keys.
- **Input**:
  - A sorted set of n keys: K1, K2, ..., Kn.
  - Probabilities p1, p2, ..., pn for searching each key.
  - Probabilities q0, q1, ..., qn for unsuccessful searches (between keys).
- **Output**: A binary search tree with the **minimum expected search cost**.
- **Approach**:
  - Uses **Dynamic Programming** to compute the minimum cost.
  - Recursively finds the root that leads to minimal total cost for each subproblem.

Time Complexity:

- **$O(n^3)$** for dynamic programming approach.

Applications:

- Efficient searching in databases and compilers.
- Used where search frequencies of keys are known in advance.