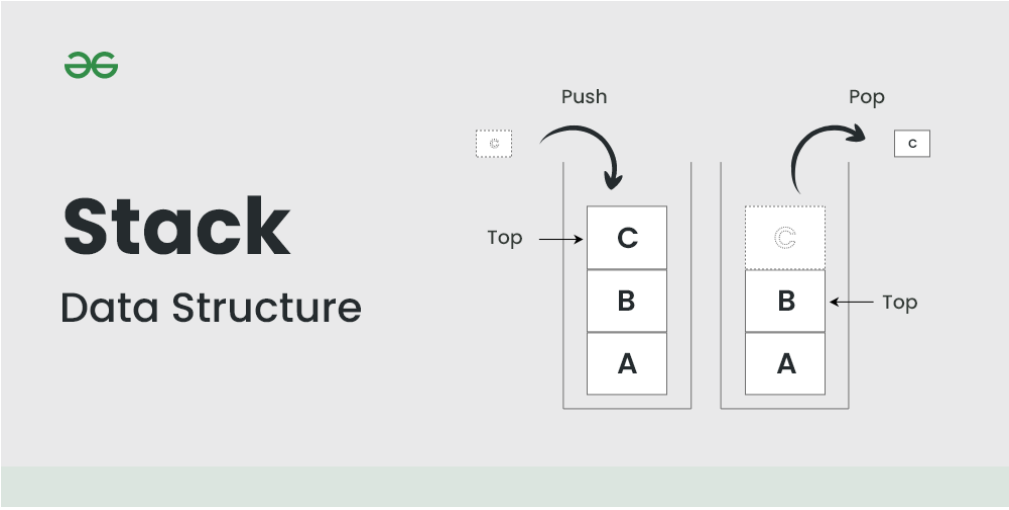
DSA

Table Of Content🍒🌮👅🍑🍆💦

Stack

Stack is a linear data structure that follows **LIFO (Last In First Out) Principle**, the last element inserted is the first to be popped out. It means both insertion and deletion operations happen at one end only.



Basic Operations on Stack:

In order to make manipulations in a stack, there are certain operations provided to us.

- **push()** to insert an element into the stack

  Algorithm for Push Operation:

  - Before pushing the element to the stack, we check if the stack is **full** .
  - If the stack is full **(top == capacity-1)** , then **Stack Overflows** and we cannot insert the element to the stack.
  - Otherwise, we increment the value of top by 1 **(top = top + 1)** and the new value is inserted at **top position** .
  - The elements can be pushed into the stack till we reach the **capacity** of the stack.

- **pop()** to remove an element from the stack

  Algorithm for Pop Operation:

  - Before popping the element from the stack, we check if the stack is **empty** .
  - If the stack is empty (top == -1), then **Stack Underflows** and we cannot remove any element from the stack.
  - Otherwise, we store the value at top, decrement the value of top by 1 **(top = top – 1)** and return the stored top value.

- **top()** Returns the top element of the stack.

  Algorithm for Top Operation:

  - Before returning the top element from the stack, we check if the stack is empty.
  - If the stack is empty (top == -1), we simply print “Stack is empty”.
  - Otherwise, we return the element stored at **index = top** .

- **isEmpty()** returns true if stack is empty else false.

  Algorithm for isEmpty Operation:

- Check for the value of **top** in stack.
- If **(top == -1)**, then the stack is **empty** so return **true** .
- Otherwise, the stack is not empty so return **false** .

- **isFull()** returns true if the stack is full else false.

  Algorithm for isFull Operation:

  - Check for the value of **top** in stack.
  - If **(top == capacity-1),** then the stack is **full** so return **true**.
  - Otherwise, the stack is not full so return **false**.

Implement Stack using Array:

Step-by-step approach:

1. **Initialize an array** to represent the stack.
2. Use the **end of the array** to represent the **top of the stack**.
3. Implement **push** (add to end), **pop** (remove from the end), and **peek** (check end) operations, ensuring to handle empty and full stack conditions.

```
#include <stdio.h>
#define MAX 5 // Maximum size of the stack
int stack[MAX]; // Array to store stack elements
int top = -1; // Global variable to keep track of the top elemer
// Function to check if the stack is full
int isFull() {
    return top == MAX - 1;
}
// Function to check if the stack is empty
int isEmpty() {
    return top == -1;
}
// Function to push an element onto the stack
void push(int item) {
    if (isFull()) {
```

```
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = item;
    printf("%d pushed onto stack\n", item);
}
// Function to pop an element from the stack
int pop() {
    if (isEmpty()) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}
// Function to get the top element of the stack
int peek() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack[top];
}
// Function to display the elements of the stack
void display() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return;
    }
    printf("Stack elements are:\n");
    for (int i = top; i >= 0; i--) {
        printf("%d\n", stack[i]);
    }
}
// Main function to test the stack
int main() {
    push(10);
```

```
    push(20);
    push(30);
    push(40);
    display(); // Display the current stack elements
    printf("%d popped from stack\n", pop());
    display(); // Display the stack elements after popping
    printf("Top element is %d\n", peek());
    return 0;
}
```

Implement a stack using singly linked list:

```
#include <stdio.h>
#include <stdlib.h>
// Node structure for the stack
typedef struct Node {
    int data;
    struct Node* next;
} Node;
Node* top = NULL; // Initialize top of stack
// Push an element onto the stack
void push(int value) {
Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Stack overflow\n");
        return;
    }
    newNode->data = value;
    newNode->next = top;
    top = newNode;
}
// Pop an element from the stack
int pop() {
    if (top == NULL) {
        printf("Stack underflow\n");
```

```
        return -1;
    }
    Node* temp = top;
    int value = temp->data;
    top = top->next;
    free(temp);
    return value;
}
// Peek the top element of the stack
int peek() {
    if (top == NULL) {
        printf("Stack is empty\n");
        return -1;
    }
    return top->data;
}
// Check if the stack is empty
int isEmpty() {
    return top == NULL;
}
// Display the stack elements
void display() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return;
    }
    Node* temp = top;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    push(10);
    push(20);
```

```
    push(30);
    display();
    printf("Popped element: %d\n", pop());
    display();
    printf("Top element: %d\n", peek());
    return 0;
}
```

Converting and evaluating Algebraic expressions:

1. Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
   a) If the scanned symbol is an operand, then place directly in the postfix expression (output).
   b) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
   c) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

2. Evaluation of postfix expression:

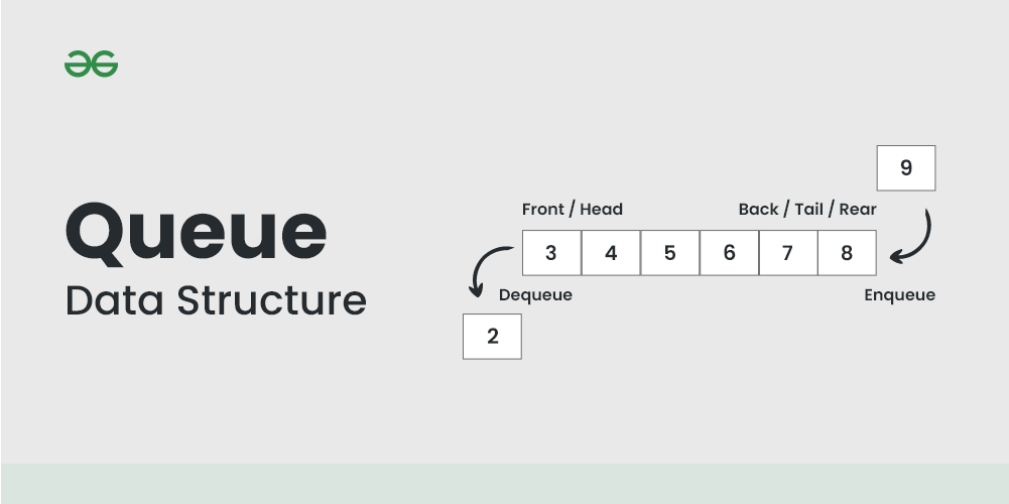
The postfix expression is evaluated easily by the use of a stack.

1. When a number is seen, it is pushed onto the stack.
2. When an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

3. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Queue

Queue is a linear data structure that follows **FIFO (First In First Out) Principle**, so the first element inserted is the first to be popped out.



Operations on Queue

1. **Enqueue:**

   Enqueue operation **adds (or stores) an element to the end of the queue**.

   Steps:

   1. Check if the **queue is full**. If so, return an **overflow** error and exit.
   2. If the queue is **not full**, increment the **rear** pointer to the next available position.
   3. Insert the element at the rear.

2. **Dequeue:**

Dequeue operation removes the element at the front of the queue. The following steps are taken to perform the dequeue operation:

1. Check if the **queue is empty**. If so, return an **underflow** error.
2. Remove the element at the **front**.
3. **Increment** the **front** pointer to the next element.

3. **Peek or Front Operation:**

   This operation returns the element at the front end without removing it.

4. **isEmpty Operation:**

   This operation returns a boolean value that indicates whether the queue is empty or not.

5. **isFull Operation:**

   This operation returns a boolean value that indicates whether the queue is full or not.

Linear Queue Implementation Using an Array in C:

```
#include <stdio.h>
#define MAX 5 // Maximum size of the queue
int queue[MAX]; // Array to store queue elements
int front = -1; // Index of the front element
int rear = -1; // Index of the rear element
// Function to check if the queue is empty
int isEmpty() {
return front == -1;
}
// Function to check if the queue is full
int isFull() {
return rear == MAX - 1;
}
// Function to add an element to the queue (Enqueue)
void enqueue(int item) {
if (isFull()) {
```

```
printf("Queue Overflow\n");
return;
}
if (isEmpty()) {
front = 0; // Initialize front to 0 when the first element is en
}
queue[++rear] = item;
printf("%d enqueued to queue\n", item);
}
// Function to remove an element from the queue (Dequeue)
int dequeue() {
if (isEmpty()) {
printf("Queue Underflow\n");
return -1;
}
int item = queue[front];
if (front == rear) {
front = rear = -1; // Reset the queue if the last element is deq
} else {
front++;
}
return item;
}
// Function to get the front element of the queue
int peek() {
if (isEmpty()) {
printf("Queue is empty\n");
return -1;
}
return queue[front];
}
// Function to display the elements of the queue
void display() {
if (isEmpty()) {
printf("Queue is empty\n");
return;
```

```
}
printf("Queue elements are:\n");
for (int i = front; i <= rear; i++) {
printf("%d ", queue[i]);
}
printf("\n");
}
// Main function to test the queue
int main() {
enqueue(10);
enqueue(20);
enqueue(30);
enqueue(40);
enqueue(50);
display(); // Display the current queue elements
printf("%d dequeued from queue\n", dequeue());
display(); // Display the queue elements after dequeuing
printf("Front element is %d\n", peek());
return 0;
}
```

Queue – Linked List Implementation:

```
#include <stdio.h>
#include <stdlib.h>
// Node structure for the queue
typedef struct Node {
int data;
struct Node* next;
} Node;
Node* front = NULL; // Initialize front of queue
Node* rear = NULL; // Initialize rear of queue
// Enqueue an element into the queue
```

```
void enqueue(int value) {
Node* newNode = (Node*)malloc(sizeof(Node));
if (newNode == NULL) {
printf("Queue overflow\n");
return;
}
newNode->data = value;
newNode->next = NULL;
if (rear == NULL) {
front = rear = newNode; // First element in the queue
} else {
rear->next = newNode;
rear = newNode;
}
}
// Dequeue an element from the queue
int dequeue() {
if (front == NULL) {
printf("Queue underflow\n");
return -1;
}
Node* temp = front;
int value = temp->data;
front = front->next;
if (front == NULL) {
rear = NULL; // Queue is empty now
}
free(temp);
return value;
}
// Peek the front element of the queue
int peek() {
if (front == NULL) {
printf("Queue is empty\n");
return -1;
}
```

```
return front->data;
}
// Check if the queue is empty
int isEmpty() {
return front == NULL;
}
// Display the queue elements
void display() {
if (isEmpty()) {
printf("Queue is empty\n");
return;
}
Node* temp = front;
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
printf("\n");
}
int main() {
enqueue(10);
enqueue(20);
enqueue(30);
display();
printf("Dequeued element: %d\n", dequeue());
display();
printf("Front element: %d\n", peek());
return 0;
}
```

Circular Queue

A Circular Queue is another way of implementing a **normal queue** where the **last element** of the queue is **connected** to the **first element** of the queue forming a circle.

Operations on Circular Queue

- **getFront:** Get the front item from the queue.
- **getRear:** Get the last item from the queue.
- **enqueue(value):** To **insert** an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.
- **dequeue()**: To **delete** an element from the circular queue. In a circular queue, the element is always deleted from the front position.

Implement Circular Queue using Array

1. Initialize an **array** of **size n**, where **n is the maximum number of element**s that the queue can hold.
2. Initialize **three variables (size, capacity, and front.)**
3. **Enqueue:** To enqueue an **element x** into the queue, do the following:
    a. Check if **size == capacity** (queue is full), display **“Queue is full”.**
    b. If not full: calculate **rear = (front + size) % capacity** and **Insert** value at the rear index. **Increment** size by 1.
4. **Dequeue:** To dequeue an element from the queue, do the following:
    a. Check if **size == 0 (**queue is empty), display **“Queue is empty”.**
    b. If not empty: **retrieve** the element at the **front index** and **move front = (front + 1) % capacity**. Also, **decrement** size by 1 and **return the removed element.**

```
#include<stdio.h>
#define size 10
int queue[size];
int front = -1;
int rear=-1;
int isFull(){
return front == (rear+1)%size;
}
```

```
int isEmpty(){
return front == -1;
}
void enque(int n){
if(isFull()){
printf("queue is full");
return;
}
if(front ==-1){
front =0;
}
rear = (rear+1)%size;
queue[rear]=n;
printf("%d inserted \n",n);
}
int deque(){
int data;
if(isEmpty()){
printf("queue is empty");
return -1;
}
data = queue[front];
if(front == rear){
front =rear=-1;
}
else{
front = (front +1)%size;
}
return data;
}
int peek(){
if(isEmpty()){
printf("queue is empty");
return -1;
}
return queue[front];
```

```
}
void display(){
int i;
if(isEmpty()){
printf("Queue is empty");
return ;
}
printf("Elements are:\n");
i= front;
while(1){
printf("%d\n",queue[i]);
if(i==rear){
break;
} i=(i+1)%size;
}}
int main(){
enque(10);
enque(20);
enque(30);
enque(40);
display();
printf("element dequeued is %d\n",deque());
printf("element dequeued is %d\n",deque());
display();
printf("front element is %d\n",peek());
return 0;
}
```

Advantages of Circular Queue Over Linear Queue:

- Efficient Use of Space: Circular queues make use of the unused spaces left behind by dequeued elements, avoiding the situation where the queue appears full in a linear queue even when there is space at the beginning of the array.

- Memory Management: Better utilization of available memory compared to a linear queue.

Priority Queue

A Priority Queue is a special type of queue in which each element is associated with a priority, and elements are dequeued based on their priority rather than their order of insertion. The element with the highest priority is served before the elements with lower priorities. If two elements have the same priority, they are served according to their order in the queue.

Priority Queue is used in algorithms such as **Dijkstra's algorithm**, **Prim's algorithm**, **Kruskal's algorithm** and **Huffman Coding**.

Data Structures for Implementation:

1. Array/List: Simple but inefficient for larger datasets, as insertion and deletion require scanning through the array to find the correct position.
2. Binary Heap: A complete binary tree where each parent node has a higher (or lower) priority than its children, allowing for efficient insertion and deletion.
3. Balanced Binary Search Tree: Allows for efficient operations but is more complex to implement than a heap.

Operations:

1. Insertion: Adds a new element to the queue with its associated priority.
2. Deletion: Removes the element with the highest (or lowest) priority.
3. Peek: Retrieves the element with the highest (or lowest) priority without removing it.

Implementation of Priority queue using array:

```
#include <stdio.h>
#define MAX 5 // Maximum size of the priority queue
// Priority Queue structure
int priorityQueue[MAX];
```

```
int priority[MAX];
int size = 0; // Keeps track of the current number of elements i
// Function to check if the queue is empty
int isEmpty() {
return size == 0;
}
// Function to check if the queue is full
int isFull() {
return size == MAX;
}
// Function to insert an element into the priority queue
void enqueue(int value, int prio) {
if (isFull()) {
printf("Priority Queue is Full\n");
return;
}
// Insert the new element at the end
priorityQueue[size] = value;
priority[size] = prio;
size++;
printf("Inserted %d with priority %d\n", value, prio);
}
// Function to find the index of the highest priority element
int findHighestPriority() {
int highestPriority = -1;
int index = -1;
for (int i = 0; i < size; i++) {
// Higher priority is denoted by lower numerical value
if (highestPriority == -1 || priority[i] < highestPriority) {
highestPriority = priority[i];
index = i;
}
}
return index;
}
// Function to remove an element from the priority queue (dequeu
```

```
int dequeue() {
if (isEmpty()) {
printf("Priority Queue is Empty\n");
return -1;
}
// Find the index of the highest priority element
int index = findHighestPriority();
int value = priorityQueue[index];
// Shift elements to the left to fill the gap created by dequeu
for (int i = index; i < size - 1; i++) {
priorityQueue[i] = priorityQueue[i + 1];
priority[i] = priority[i + 1];
}
size--; // Decrease the size of the queue
printf("Removed %d with priority %d\n", value, priority[index]);
return value;
}
// Function to display the elements of the priority queue
void display() {
if (isEmpty()) {
printf("Priority Queue is Empty\n");
return;
}
printf("Priority Queue elements:\n");
for (int i = 0; i < size; i++) {
printf("Value: %d, Priority: %d\n", priorityQueue[i], priority[i
}
}
int main() {
enqueue(10, 2);
enqueue(20, 1);
enqueue(30, 3);
enqueue(40, 0);
enqueue(50, 4);
display();
dequeue();
```

```
display();
dequeue();
display();
return 0;
}
```

Double-Ended Queue (Deque)

A Double-Ended Queue (Deque) is a data structure that allows you to insert and delete elements from both ends, i.e., the front and the rear. This flexibility makes it more versatile than a regular queue, where insertion happens only at the rear and deletion happens only at the front.

Operations on Deque:

Operation	Description	Time Complexity
push_front()	Inserts the element at the beginning.	**O(1)**
push_back()	Adds element at the end.	**O(1)**
pop_front()	Removes the first element from the deque.	**O(1)**
pop_back()	Removes the last element from the deque.	**O(1)**
front()	Gets the front element from the deque.	**O(1)**
back()	Gets the last element from the deque.	**O(1)**
empty()	Checks whether the deque is empty or not.	**O(1)**
size()	Determines the number of elements in the deque.	**O(1)**

Implementation of Double Ended Queue using Array:

```
#include <stdio.h>
#define SIZE 5
int deque[SIZE];
int front = -1, rear = -1;
// Check if deque is full
int isFull() {
```

```
return (front == (rear + 1) % SIZE);
}
// Check if deque is empty
int isEmpty() {
return (front == -1);
}
// Insert at front
void insertFront(int value) {
if (isFull()) {
printf("Deque is full\n");
return;
}
if (isEmpty()) {
front = rear = 0;
} else {
front = (front - 1 + SIZE) % SIZE;
}
deque[front] = value;
printf("Inserted %d at front\n", value);
}
// Insert at rear
void insertRear(int value) {
if (isFull()) {
printf("Deque is full\n");
return;
}
if (isEmpty()) {
front = rear = 0;
} else {
rear = (rear + 1) % SIZE;
}
deque[rear] = value;
printf("Inserted %d at rear\n", value);
}
// Delete from front
void deleteFront() {
```

```
if (isEmpty()) {
printf("Deque is empty\n");
return;
}
printf("Deleted %d from front\n", deque[front]);
if (front == rear) {
front = rear = -1; // Deque becomes empty
} else {
front = (front + 1) % SIZE;
}
}
// Delete from rear
void deleteRear() {
if (isEmpty()) {
printf("Deque is empty\n");
return;
}
printf("Deleted %d from rear\n", deque[rear]);
if (front == rear) {
front = rear = -1; // Deque becomes empty
} else {
rear = (rear - 1 + SIZE) % SIZE;
}
}
// Peek the front element
int peekFront() {
if (isEmpty()) {
printf("Deque is empty\n");
return -1;
}
return deque[front];
}
// Peek the rear element
int peekRear() {
if (isEmpty()) {
printf("Deque is empty\n");
```

```
return -1;
}
return deque[rear];
}
// Display the elements of deque
void display() {
if (isEmpty()) {
printf("Deque is empty\n");
return;
}
printf("Deque elements: ");
for(int i=front;i!=rear;i=(i+1)%SIZE)
printf("%d ", deque[i]);
printf("%d\n", deque[rear]); // Print the last element
}
// Main function
int main() {
insertRear(10);
insertRear(20);
display();
insertFront(5);
display();
printf("Front element: %d\n", peekFront());
printf("Rear element: %d\n", peekRear());
deleteFront();
display();
deleteRear();
display();
return 0;
}
```

Linked List

A **linked list** is a fundamental data structure in computer science. It mainly allows efficient **insertion** and **deletion** operations compared to **arrays**. Like arrays, it is

also used to implement other data structures like stack, queue and deque. Here's the comparison of Linked List vs Arrays

Linked List:

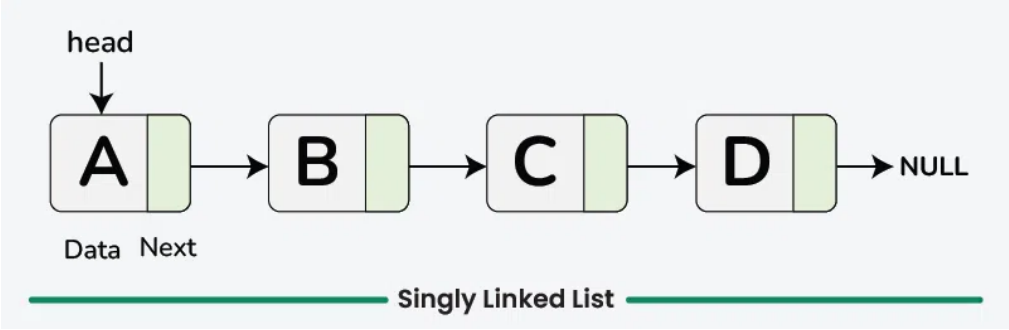
- **Data Structure:** Non-contiguous
- **Memory Allocation:** Typically allocated one by one to individual elements
- **Insertion/Deletion:** Efficient
- **Access:** Sequential

Array:

- **Data Structure:** Contiguous
- **Memory Allocation:** Typically allocated to the whole array
- **Insertion/Deletion:** Inefficient
- **Access:** Random

Singly Linked List

A **singly linked list** is a fundamental data structure, it consists of **nodes** where each node contains a **data** field and a **reference** to the next node in the linked list.



Understanding Node Structure

In a singly linked list, each node consists of two parts: data and a pointer to the next node. This structure allows nodes to be dynamically linked together, forming a chain-like sequence.

```
// Definition of a Node in a singly linked list
struct Node {
int data;

struct Node* next;
};

// Function to create a new Node
struct Node* newNode(int data) {
struct Node* temp =
(struct Node*)malloc(sizeof(struct Node));
temp->data = data;
temp->next = NULL;
return temp;
}
```

Operations on Singly Linked List

- **Traversal**

  Step-by-step approach:

  - Initialize a pointer current to the head of the list.
  - Use a while loop to iterate through the list until the current pointer reaches NULL.
  - Inside the loop, print the data of the current node and move the current pointer to the next node.

```
// Function to traverse and print the elements
// of the linked list
void traverseLinkedList(struct Node* head)
{
    // Start from the head of the linked list
    struct Node* current = head;
```

```
    // Traverse the linked list until reaching the end (NULL)
    while (current != NULL) {

        // Print the data of the current node
        printf("%d ", current->data);

        // Move to the next node
        current = current->next;
    }

    printf("\n");
}
```

- **Searching**

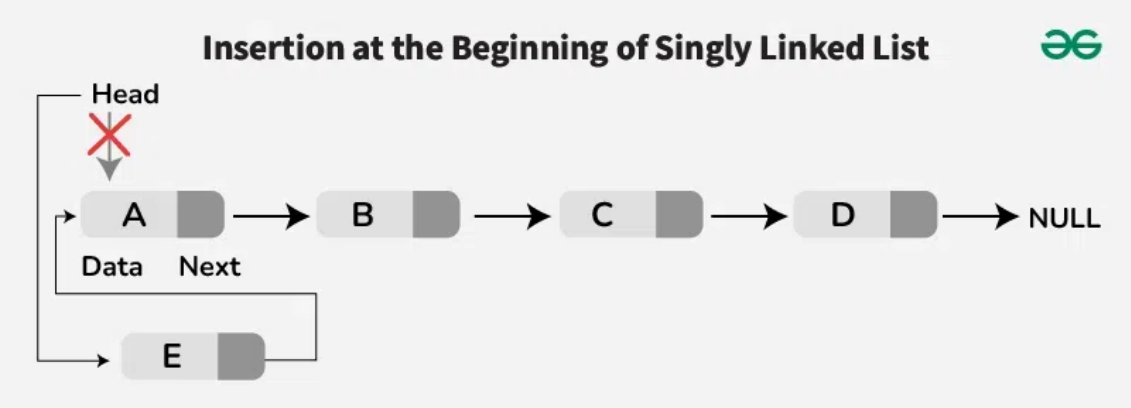
  Step-by-step approach:

  1. Traverse the Linked List starting from the head.
  2. Check if the current node's data matches the target value.
     - If a match is found, return **true**.
  3. Otherwise, Move to the next node and repeat steps 2.
  4. If the end of the list is reached without finding a match, return **false**.

```
// Function to search for a value in the Linked List
bool searchLinkedList(struct Node* head, int target)
{
    // Traverse the Linked List
    while (head != NULL) {

        // Check if the current node's
        // data matches the target value
        if (head->data == target) {
            return true; // Value found
        }
```

```
        // Move to the next node
        head = head->next;
    }

    return false; // Value not found
}
```

- **Insertion:**
  - Insert at the beginning



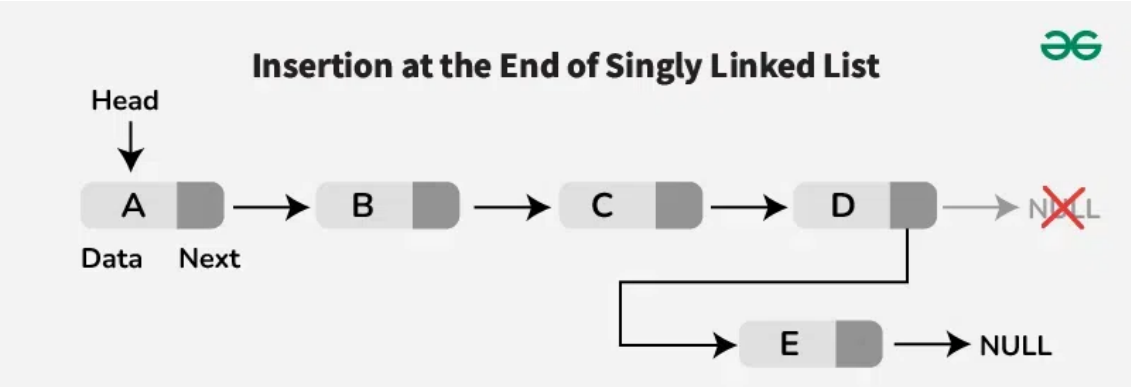
Step-by-step approach:

- Create a new node with the given value.
- Set the **next** pointer of the new node to the current head.
- Move the head to point to the new node.
- Return the new head of the linked list.

```
// Function to insert a new node at the beginning of the l
struct Node* insertAtBeginning(struct Node* head, int valu
{
    // Create a new node with the given value
    struct Node* new_node = newNode(value);

    // Set the next pointer of the new node to the current
    new_node->next = head;
```

```
    // Move the head to point to the new node
    head = new_node;

    // Return the new head of the linked list
    return head;
}
```

- Insert at the end



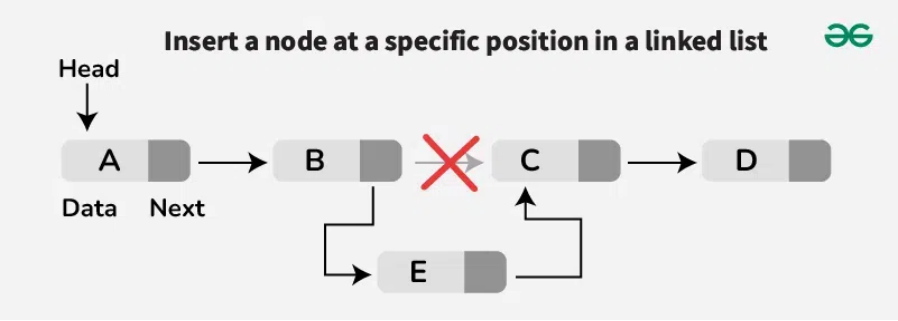
Step-by-step approach:

- Create a new node with the given value.
- Check if the list is empty:
  - If it is, make the new node the head and return.
- Traverse the list until the last node is reached.
- Link the new node to the current last node by setting the last node's next pointer to the new node.

```
// Function to insert a node at the end of the linked list
struct Node* insertAtEnd(struct Node* head, int value)
{
    // Create a new node with the given value
    struct Node* new_node = newNode(value);
```

```
    // If the list is empty, make the new node the head
    if (head == NULL)
        return new_node;

    // Traverse the list until the last node is reached
    struct Node* curr = head;
    while (curr->next != NULL) {
        curr = curr->next;
    }

    // Link the new node to the current last node
    curr->next = new_node;

    return head;
}
```

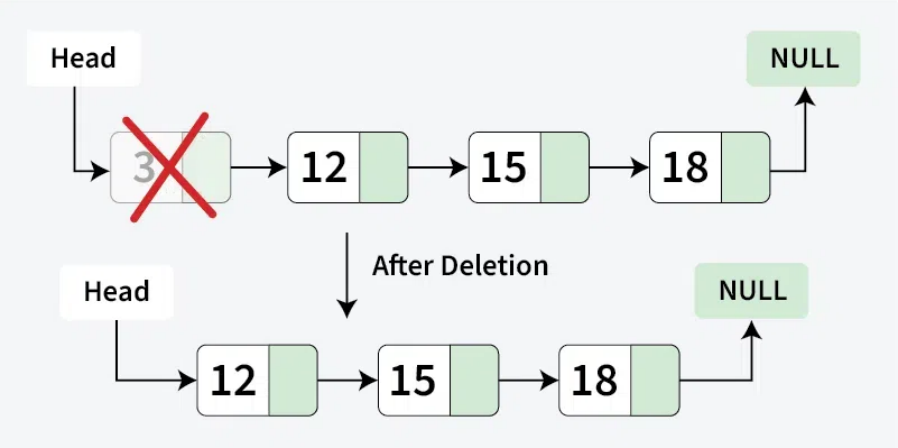
- Insert at a specific position




```
// Function to insert a node at a specified position
struct Node* insertPos(struct Node* head, int pos, int dat
    if (pos < 1) {
        printf("Invalid position!\n");
        return head;
    }

    // Special case for inserting at the head
    if (pos == 1) {
```

```
        struct Node* temp = getNode(data);
        temp->next = head;
        return temp;
    }

    // Traverse the list to find the node
    // before the insertion point
    struct Node* prev = head;
    int count = 1;
    while (count < pos - 1 && prev != NULL) {
        prev = prev->next;
        count++;
    }

    // If position is greater than the number of nodes
    if (prev == NULL) {
        printf("Invalid position!\n");
        return head;
    }

    // Insert the new node at the specified position
    struct Node* temp = getNode(data);
    temp->next = prev->next;
    prev->next = temp;

    return head;
}
```

- **Deletion:**
  - Delete from the beginning



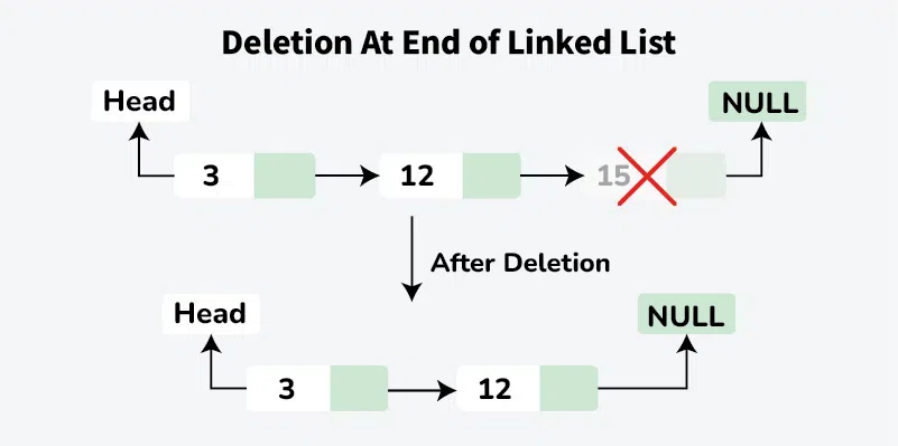
Steps-by-step approach:

- Check if the head is **NULL**.
  - If it is, return **NULL** (the list is empty).
- Store the current head node in a temporary variable **temp**.
- Move the head pointer to the next node.
- Delete the temporary node.
- Return the new head of the linked list.

```
// Function to remove the first node of the linked list
struct Node* removeFirstNode(struct Node* head)
{
    if (head == NULL)
        return NULL;

    // Move the head pointer to the next node
    struct Node* temp = head;
    head = head->next;

    // Free the memory of the old head
    free(temp);
```

```
    return head;
}
```

- Delete from the end



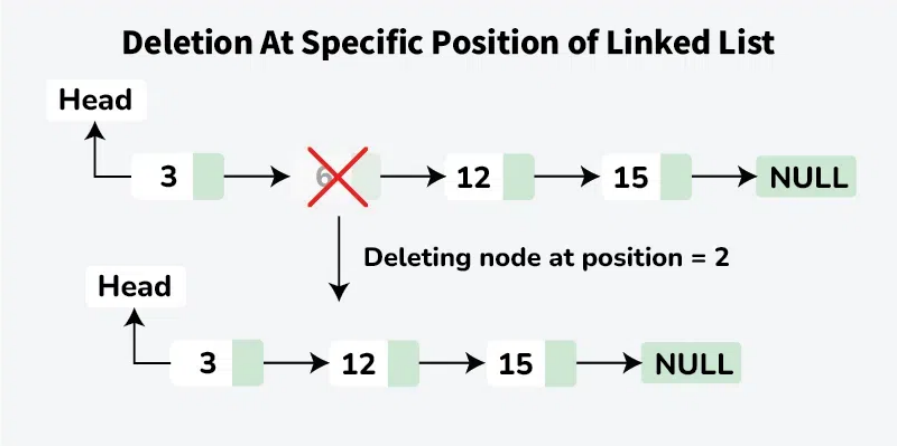
Step-by-step approach:

- Check if the head is **NULL**.
  - If it is, return NULL (the list is empty).
- Check if the head's **next** is **NULL** (only one node in the list).
  - If true, delete the head and return **NULL**.
- Traverse the list to find the second last node (**second_last**).
- Delete the last node (the node after **second_last**).
- Set the **next** pointer of the second last node to **NULL**.
- Return the head of the linked list.

```
// Function to remove the last node of the linked list
struct Node* removeLastNode(struct Node* head)
{
    if (head == NULL)
        return NULL;

    if (head->next == NULL) {
```

```
        free(head);
        return NULL;
    }

    // Find the second last node
    struct Node* second_last = head;
    while (second_last->next->next != NULL)
        second_last = second_last->next;

    // Delete last node
    free(second_last->next);

    // Change next of second last
    second_last->next = NULL;

    return head;
}
```

- Delete a specific node



Step-by-step approach:

- Check if the list is empty or the position is invalid, return if so.
- If the head needs to be deleted, update the head and delete the node.
- Traverse to the node before the position to be deleted.

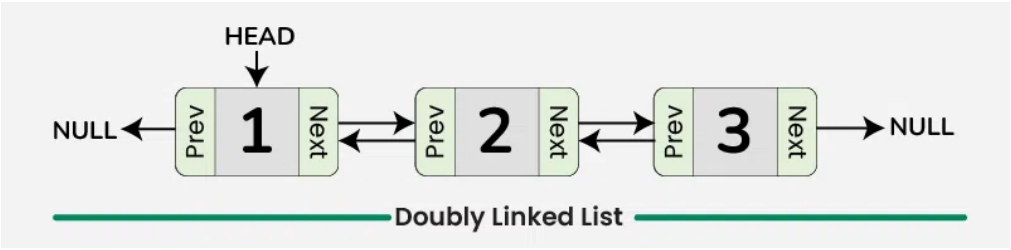
- If the position is out of range, return.
- Store the node to be deleted.
- Update the links to bypass the node.
- Delete the stored node.

```
// Function to delete a node at a specific position
struct Node* deleteAtPosition(struct Node* head, int posit
{
    // If the list is empty or the position is invalid
    if (head == NULL || position < 1) {
        return head;
    }

    // If the head needs to be deleted
    if (position == 1) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
        return head;
    }

    // Traverse to the node before the position to be dele
    struct Node* curr = head;
    for (int i = 1; i < position - 1 && curr != NULL; i++)
        curr = curr->next;
    }

    // If the position is out of range
    if (curr == NULL || curr->next == NULL) {
        return head;
    }

    // Store the node to be deleted
    struct Node* temp = curr->next;
```

```
    // Update the links to bypass the node to be deleted
    curr->next = curr->next->next;

    // Delete the node
    free(temp);

    return head;
}
```

Doubly Linked List

A **doubly linked list** is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node.



a doubly linked list is represented using nodes that have three fields:

1. Data
2. A pointer to the next node (**next**)
3. A pointer to the previous node (**prev**)

```
struct Node {

    // To store the Value or data.
    int data;
```

```
    // Pointer to point the Previous Element
    Node* prev;

    // Pointer to point the Next Element
    Node* next;
};

// Function to create a new node
struct Node *createNode(int new_data) {
    struct Node *new_node = (struct Node *)
    malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = NULL;
    new_node->prev = NULL;
    return new_node;
}
```

Operations on Doubly Linked List

- **Traversal**

  a. Forward Traversal:

  - Initialize a pointer to the head of the linked list.
  - While the pointer is not null:
    - Visit the data at the current node.
    - Move the pointer to the next node.

  b. Backward Traversal:

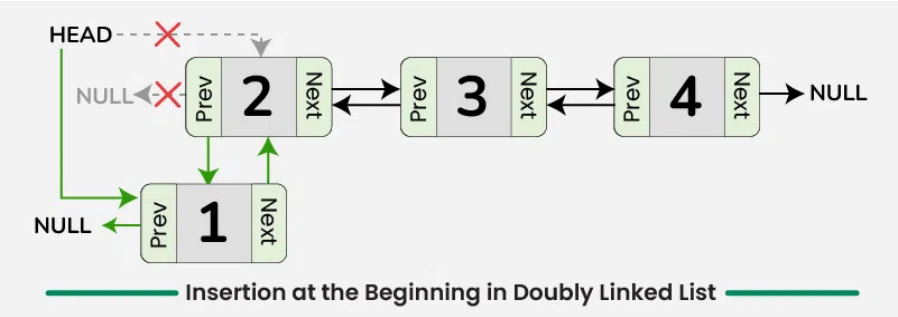
  - Initialize a pointer to the tail of the linked list.
  - While the pointer is not null:
    - Visit the data at the current node.
    - Move the pointer to the previous node.

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

// Define the Node structure
struct Node {
    int data; // Data stored in the node
    struct Node* next; // Pointer to the next node
    struct Node* prev; // Pointer to the previous node
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode =
      (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}

// Function to traverse the doubly linked list
// in forward direction
void forwardTraversal(struct Node* head) {

    // Start traversal from the head of the list
    struct Node* curr = head;

    // Continue until the current node is not
    // null (end of list)
    while (curr != NULL) {

        // Output data of the current node
        printf("%d ", curr->data);

        // Move to the next node
```

```
        curr = curr->next;
    }

    // Print newline after traversal
    printf("\n");
}

// Function to traverse the doubly linked list
// in backward direction
void backwardTraversal(struct Node* tail) {

    // Start traversal from the tail of the list
    struct Node* curr = tail;

    // Continue until the current node is not
    // null (end of list)
    while (curr != NULL) {

        // Output data of the current node
        printf("%d ", curr->data);

        // Move to the previous node
        curr = curr->prev;
    }

    // Print newline after traversal
    printf("\n");
}

int main() {

    // Sample usage of the doubly linked list and
    // traversal functions
    struct Node* head = createNode(1);
    struct Node* second = createNode(2);
    struct Node* third = createNode(3);
```

```
    head->next = second;
    second->prev = head;
    second->next = third;
    third->prev = second;

    printf("Forward Traversal:\n");
    forwardTraversal(head);

    printf("Backward Traversal:\n");
    backwardTraversal(third);

    // Free memory allocated for nodes
    free(head);
    free(second);
    free(third);

    return 0;
}
```

- **Searching**
- **Insertion:**
  - Insertion at the beginning



To insert a new node at the beginning of the doubly list, we can use the following steps:

- Create a new node, say **new_node** with the given data and set its previous pointer to null, **new_node→prev = NULL**.
- Set the next pointer of new_node to current head, **new_node→next = head.**
- If the linked list is not empty, update the previous pointer of the current head to new_node, **head→prev = new_node**.
- Return new_node as the head of the updated linked list.

```
// C Program to insert a node at the beginning
//of doubly linked list


#include <stdio.h>

// Node structure for the doubly linked list
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Create a new node
struct Node* createNode(int data) {
    struct Node* new_node =
      (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->prev = NULL;
    new_node->next = NULL;
    return new_node;
}

// Insert a node at the beginning
struct Node* insertBegin(struct Node* head, int data) {
```

```
    // Create a new node
    struct Node* new_node = createNode(data);

    // Make next of it as head
    new_node->next = head;

    // Set previous of head as new node
    if (head != NULL) {
        head->prev = new_node;
    }

    // Return new node as new head
    return new_node;
}

// Print the doubly linked list
void printList(struct Node* head) {
    struct Node* curr = head;
    while (curr != NULL) {
        printf("%d ", curr->data);
        curr = curr->next;
    }
    printf("\n");
}

int main() {

    // Create a hardcoded doubly linked list:
    // 2 <-> 3 <-> 4
    struct Node *head = createNode(2);
    head->next = createNode(3);
    head->next->prev = head;
    head->next->next = createNode(4);
    head->next->next->prev = head->next;

    // Print the original list
```

```
    printf("Original Linked List: ");
    printList(head);

    // Insert a new node at the front of the list
    head = insertBegin(head, 1);

    // Print the updated list
    printf("After inserting Node 1 at the front: ");
    printList(head);

    return 0;
}
```

- Insertion at the end
- Insertion at a specific position

- **Deletion:**
  - Deletion at the beginning
  - Deletion at the end
  - Deletion at a specific position

Circular Linked List

A **circular linked list** is a data structure where the last node connects back to the first, forming a loop. This structure allows for continuous traversal without any interruptions. Circular linked lists are especially helpful for tasks like **scheduling** and **managing playlists,** this allowing for smooth navigation.

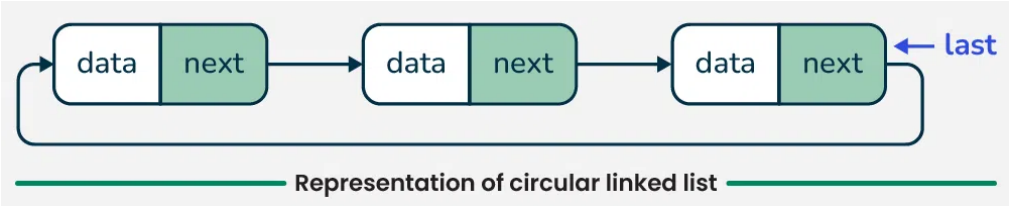
Types of Circular Linked Lists

We can create a circular linked list from both **singly linked lists** and **doubly linked lists**. So, circular linked list are basically of two types:

1. Circular Singly Linked List

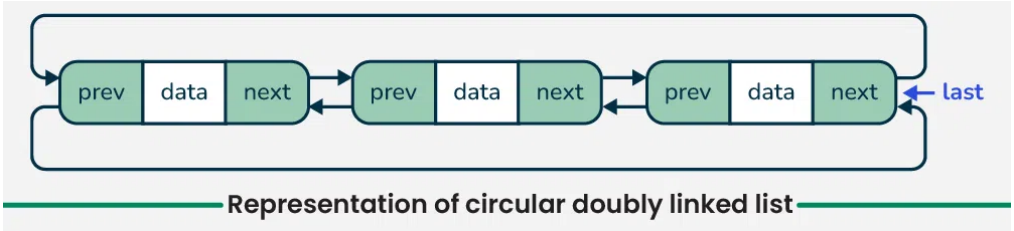
In **Circular Singly Linked List**, each node has just one pointer called the "**next**" pointer. The next pointer of **last node** points back to the **first node** and this results

in forming a circle. In this type of Linked list we can only move through the list in one direction.



2. Circular Doubly Linked List:

In **circular doubly linked list,** each node has two pointers **prev** and **next,** similar to doubly linked list. The **prev** pointer points to the previous node and the **next** points to the next node. Here, in addition to the **last** node storing the address of the first node, the **first node** will also store the address of the **last node**.




```
// Node structure
struct Node
{
    int data;
    struct Node *next;
};

// Function to create a new node
struct Node *createNode(int value){

    // Allocate memory
    struct Node *newNode =
      (struct Node *)malloc(sizeof(struct Node));
```

```
    // Set the data
    newNode->data = value;

    // Initialize next to NULL
    newNode->next = NULL;

    // Return the new node
    return newNode;
}
```

Operations on the Circular Linked list:

We can do some operations on the circular linked list similar to the singly and doubly linked list which are:

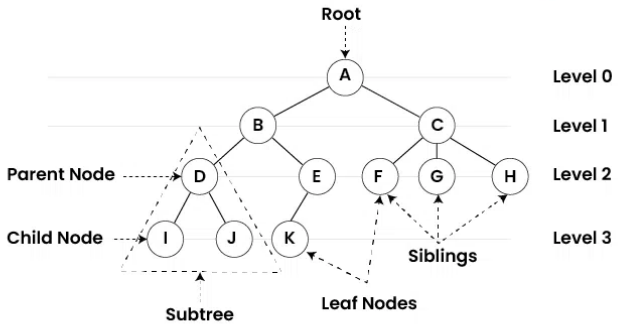
- **Insertion**
  - Insertion at the beginning
  - Insertion at the end
  - Insertion at the given position
- **Deletion**
  - Delete the first node
  - Delete the last node
  - Delete the node from any position

Trees

Tree Data Structure is a non-linear data structure in which a collection of elements known as nodes are connected to each other via edges such that there exists exactly one path between any two nodes.

Tree

Data Structure



Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{I, J, K, F, G, H}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.

- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

Types of Trees

1. Binary Tree:
   o Each node has at most two children, called the left child and the right child.
   o Example: Expression trees, Huffman coding trees.
2. Binary Search Tree (BST):
   o A binary tree in which the left subtree of a node contains only nodes with values
   less than the node's value, and the right subtree only contains nodes with values
   greater than the node's value.
   o It allows fast lookup, addition, and removal of items.
3. AVL Tree:
   o A self-balancing binary search tree. For each node, the difference between the
   heights of the left and right subtrees cannot be more than one.
   o Balancing ensures O(log n) time complexity for search, insertion, and deletion.
4. B-Tree:
   o A self-balancing tree data structure that maintains sorted data and allows searches,
   sequential access, insertions, and deletions in logarithmic time.
   o Widely used in database indexing.
5. B+ Tree:
   o An extension of B-Tree, in which all values are present in the leaf nodes and internal nodes are used for navigation.
   o Used in file systems and database systems.
6. Red-Black Tree:
   o A type of self-balancing binary search tree, where each node has an extra bit for

denoting the color of the node, either red or black.
o It helps in balancing the tree after insertions and deletions.

7. Heap:
   o A special tree-based data structure that satisfies the heap property (either Max-Heap
   or Min-Heap), where the parent node is always greater (in Max-Heap) or smaller (in
   Min-Heap) than the child nodes.
   o Used in priority queues.

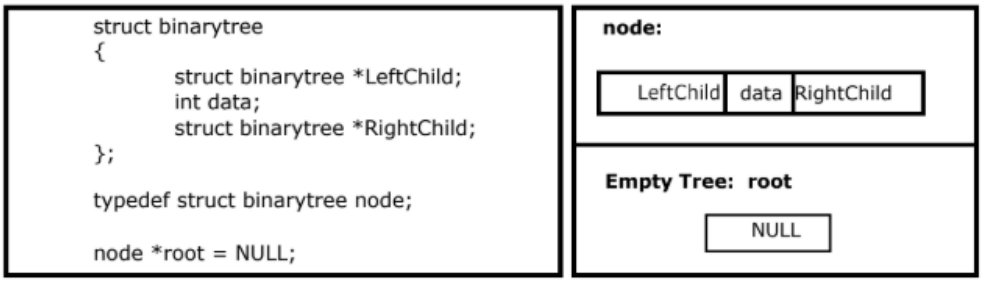


Figure 5.2.6. Structure definition, node representation and empty tree

Recursive Traversal Algorithms:

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
    if(root != NULL)
    {
        inorder(root->lchild);
        print root -> data;
        inorder(root->rchild);
    }
}
```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking.

The algorithm for inorder traversal is as follows:

```
void preorder(node *root)
{
    if( root != NULL )
    {
        print root -> data;
        preorder (root -> lchild);
        preorder (root -> rchild);
    }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed.

```
void postorder(node *root)
{
    if( root != NULL )
    {
        postorder (root -> lchild);
        postorder (root -> rchild);
        print (root -> data);
    }
}
```

Building Binary Tree from Traversal Pairs:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given, then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root

node can be identified using inorder. Same technique can be applied repeatedly to form sub-trees.

Expression Trees:

Expression tree is a binary tree, because all of the operations are binary. It is also possible for a node to have only one child, as is the case with the unary minus operator. The leaves of an expression tree are operands, such as constants or variable names, and the other (non-leaf) nodes contain operators.

An expression tree can be generated for the infix and postfix expressions. First convert the infix expression into postfix notation.
An algorithm to convert a postfix expression into an expression tree is as follows:
i. Read the expression one symbol at a time.
ii. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack.
iii. If the symbol is an operator, we pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively. A pointer to this new tree is then pushed onto the stack.

Converting expressions

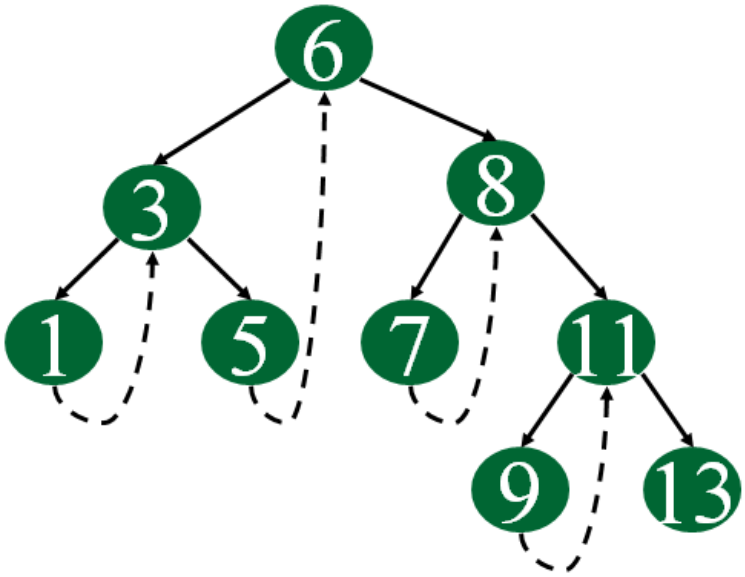
1. Postfix to Infix:
   The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.
   A. Create the expression tree from the postfix expression
   B. Run inorder traversal on the tree.
2. Postfix to Prefix:
   The following algorithm works for the expressions to convert postfix to prefix:
   A. Create the expression tree from the postfix expression
   B. Run preorder traversal on the tree.
3. Prefix to Infix:
   The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

A. Create the expression tree from the prefix expression
B. Run inorder traversal on the tree.

4. Prefix to postfix:
   The following algorithm works for the expressions to convert postfix to prefix:
   A. Create the expression tree from the prefix expression
   B. Run postorder traversal on the tree.

Threaded Binary Tree:

The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. 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.



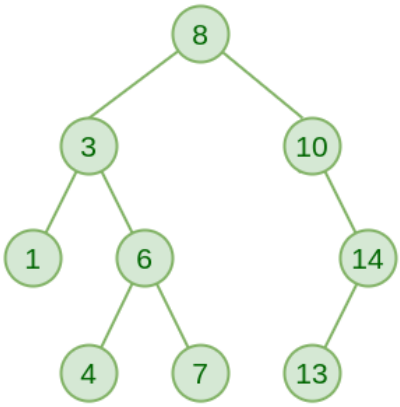
C and C++ representation of a Threaded Node

```
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    bool rightThread;
```

```
} Node;

// Constructor function
Node* createNode(int val) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode != NULL) {
        newNode->data = val;
        newNode->left = NULL;
        newNode->right = NULL;
        newNode->rightThread = false;
    }
    return newNode;
}
```

Binary Search Tree

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.



Binary Search Tree

Properties of Binary Search Tree:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

Insertion in Binary Search Tree

- Initilize the current node (say, **currNode or node**) with root node
- Compare the **key** with the current node.
- **Move left** if the **key** is less than or equal to the current node value.
- **Move right** if the **key** is greater than current node value.
- Repeat steps 2 and 3 until you reach a leaf node.
- Attach the **new key** as a left or right child based on the comparison with the leaf node's value.

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

```
// Define the structure for a BST node
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

// Function to create a new BST node
struct Node* newNode(int item) {
    struct Node* temp =
       (struct Node*)malloc(sizeof(struct Node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}


// Function to insert a new node with the given key
struct Node* insert(struct Node* node, int key) {

    // If the tree is empty, return a new node
    if (node == NULL)
        return newNode(key);

    // If the key is already present in the tree,
    // return the node
    if (node->key == key)
        return node;

    // Otherwise, recur down the tree. If the key
    // to be inserted is greater than the node's key,
    // insert it in the right subtree
    if (node->key < key)
        node->right = insert(node->right, key);
```

```
    // If the key to be inserted is smaller than
    // the node's key,insert it in the left subtree
    else
        node->left = insert(node->left, key);

    // Return the (unchanged) node pointer
    return node;
}

// Function to perform inorder tree traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// Driver program to test the above functions
int main() {
    // Creating the following BST
    //      50
    //     /  \
    //    30   70
    //   / \   / \
    //  20 40 60 80

    struct Node* root = newNode(50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Print inorder traversal of the BST
```

```
    inorder(root);

    return 0;
}
```

Output

```
20 30 40 50 60 70 80
```

Time Complexity:

- The worst-case time complexity of insert operations is **O(h)** where **h** is the height of the Binary Search Tree.
- In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become **n** and the time complexity of insertion operation may become **O(n).**

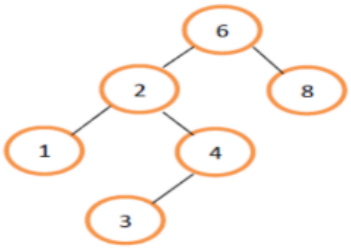
Auxiliary Space: The auxiliary space complexity of insertion into a binary search tree is **O(1)**

Deletion in Binary Search Tree

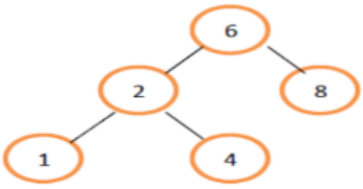
Case 1. Delete a Leaf Node in BST

If the node is a leaf, it can be deleted immediately.
Steps are

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



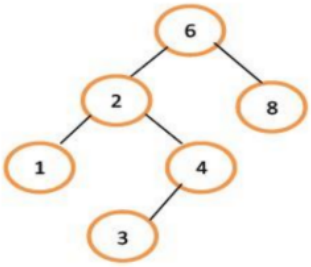
Before Deletion



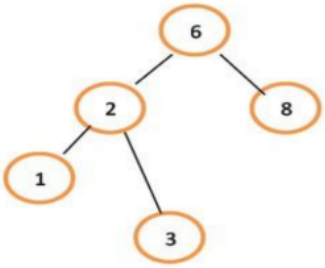
After Deletion of 3

Case 2. Delete a Node with Single Child in BST

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



Before deletion of 4

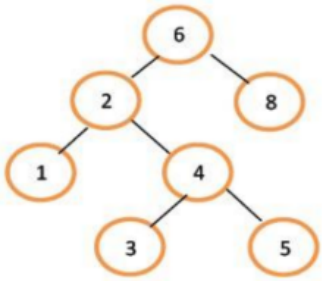


After deletion of 4

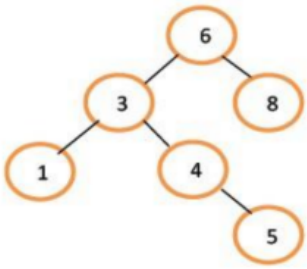
Case 3. Delete a Node with Both Children in BST

It is difficult to delete a node which has two children. So, a general strategy has to be followed.

1. Replace the data of the node to be deleted with either the smallest element from the right
   subtree or the largest element from the left subtree.



Before deletion of 2



After deletion

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

struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

// Note that it is not a generic inorder successor
// function. It mainly works when the right child
// is not empty, which is  the case we need in
// BST delete.
struct Node* getSuccessor(struct Node* curr) {
    curr = curr->right;
    while (curr != NULL && curr->left != NULL)
        curr = curr->left;
    return curr;
}

// This function deletes a given key x from the
// given BST  and returns the modified root of
// the BST (if it is modified)
struct Node* delNode(struct Node* root, int x) {
```

```
    // Base case
    if (root == NULL)
        return root;

    // If key to be searched is in a subtree
    if (root->key > x)
        root->left = delNode(root->left, x);
    else if (root->key < x)
        root->right = delNode(root->right, x);
    else {
        // If root matches with the given key

        // Cases when root has 0 children or
        // only right child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }

        // When root has only left child
        if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // When both children are present
        struct Node* succ = getSuccessor(root);
        root->key = succ->key;
        root->right = delNode(root->right, succ->key);
    }
    return root;
}

struct Node* createNode(int key) {
```

```
    struct Node* newNode =
       (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Utility function to do inorder traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// Driver code
int main() {
    struct Node* root = createNode(10);
    root->left = createNode(5);
    root->right = createNode(15);
    root->right->left = createNode(12);
    root->right->right = createNode(18);
    int x = 15;

    root = delNode(root, x);
    inorder(root);
    return 0;
}
```

Output

```
5 10 12 18
```

Time Complexity: O(h), where h is the height of the BST.

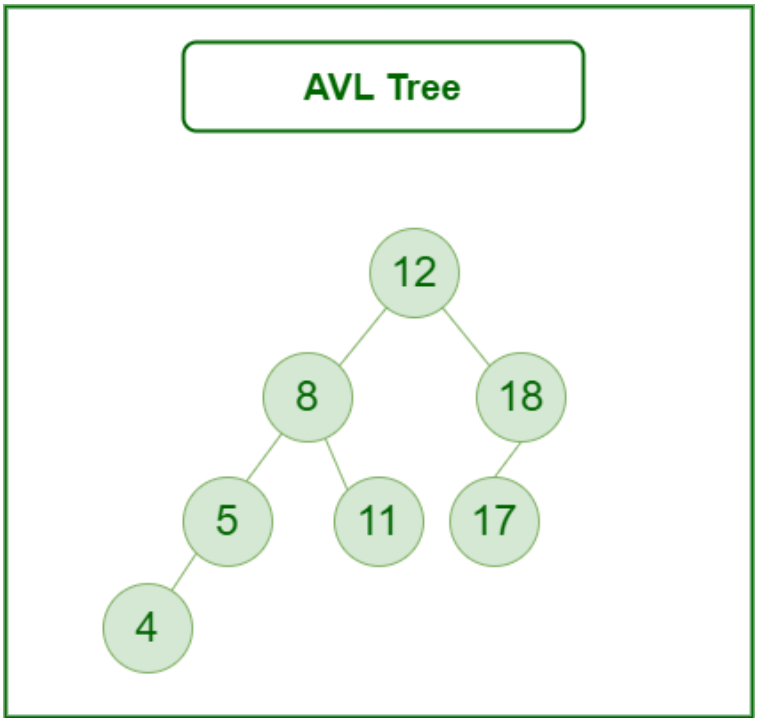
Auxiliary Space: O(h).

AVL Tree

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor = height Of LeftSubtree height Of RightSubtree



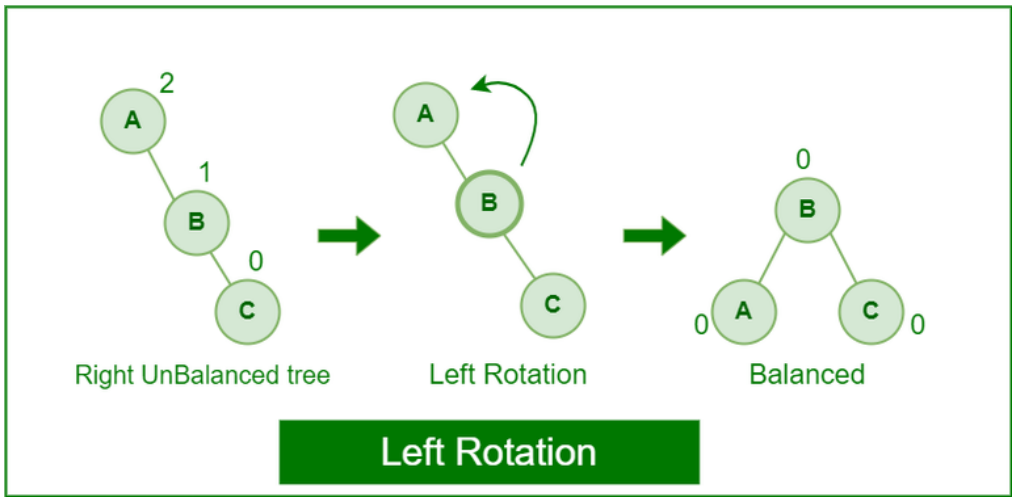
Rotating the subtrees in an AVL Tree:

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

There are four rotations

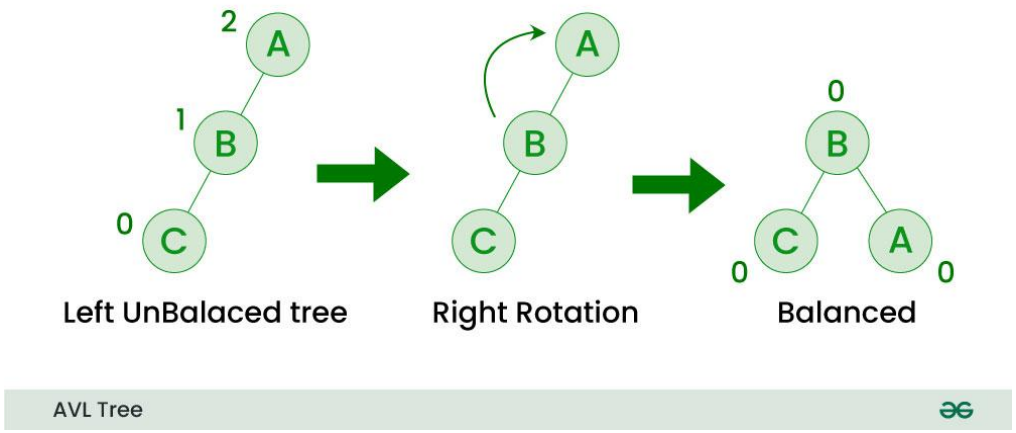
Left Rotation (LL Rotation):

When a node is added into the right subtree of the right subtree.



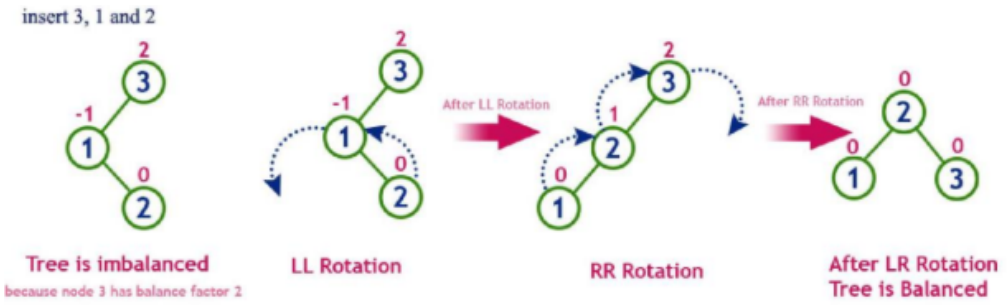
Right Rotation (RR Rotation):

If a node is added to the left subtree of the left subtree.



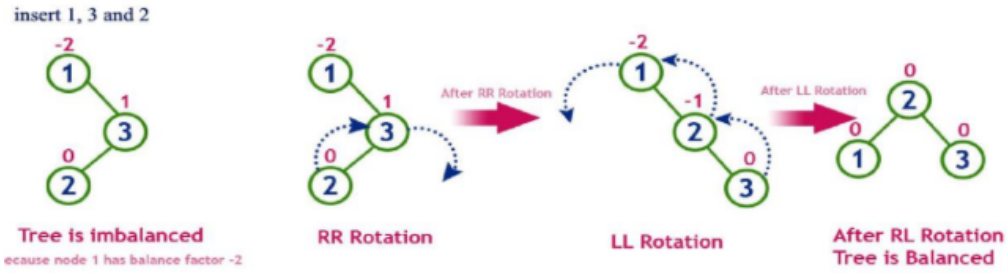
Left-Right Rotation:

A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



Insertion

Steps to follow for insertion:

Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.**

Advantages of AVL Tree:

1. AVL trees can self-balance themselves and therefore provides time complexity as O(Log n) for search, insert and delete.
2. It is a BST only (with balancing), so items can be traversed in sorted order.

Disadvantages of AVL Tree:

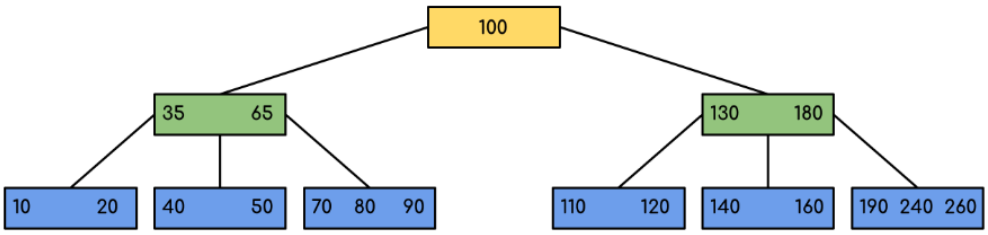
1. It is difficult to implement compared to normal BST.
2. AVL trees provide complicated insertion and removal operations as more rotations are performed.

B Tree

A B-tree is a balanced m- ordered multiway search tree, where m > 2. B-Tree of order m can have at most m-1 keys and m children.

Properties of B-Tree:

- All leaves are at the same level.
- B-Tree is defined by the term minimum degree '**t**'. The value of '**t**' depends upon disk block size.
- Every node except the root must contain at least t-1 keys. The root may contain a minimum of **1** key.
- All nodes (including root) may contain at most (**2*t – 1**) keys.
- Number of children of a node is equal to the number of keys in it plus **1**.
- All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, the time complexity to search, insert, and delete is O (log n).
- Insertion of a Node in B-Tree happens only at Leaf Node.



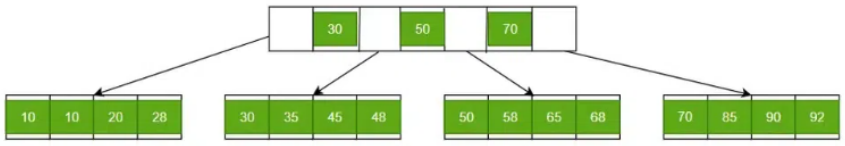
Insertions

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contains less than m-1 keys, then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   - Insert the new element in the increasing order of elements.
   - Split the node into the two nodes at the median.
   - Push the median element upto its parent node.
   - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

B+ Tree

B + Tree is a variation of the B-tree data structure. In a B + tree, data pointers are stored only at the leaf nodes of the tree. The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient



Difference Between B+ Tree and B Tree

Some differences between **B+ Tree and B Tree** are stated below.

Parameters	B+ Tree	B Tree
Structure	Separate leaf nodes for data storage and internal nodes for indexing	Nodes store both keys and data values

Leaf Nodes	Leaf nodes form a linked list for efficient range-based queries	Leaf nodes do not form a linked list
Order	Higher order (more keys)	Lower order (fewer keys)
Key Duplication	Typically allows key duplication in leaf nodes	Usually does not allow key duplication
Disk Access	Better disk access due to sequential reads in a linked list structure	More disk I/O due to non-sequential reads in internal nodes
Applications	Database systems, file systems, where range queries are common	In-memory data structures, databases, general-purpose use
Performance	Better performance for range queries and bulk data retrieval	Balanced performance for search, insert, and delete operations
Memory Usage	Requires more memory for internal nodes	Requires less memory as keys and values are stored in the same node

Graph

A **Graph** is a non-linear data structure consisting of vertices and edges. 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

Graph can be represented in the following ways:

a) Adjacency Matrix

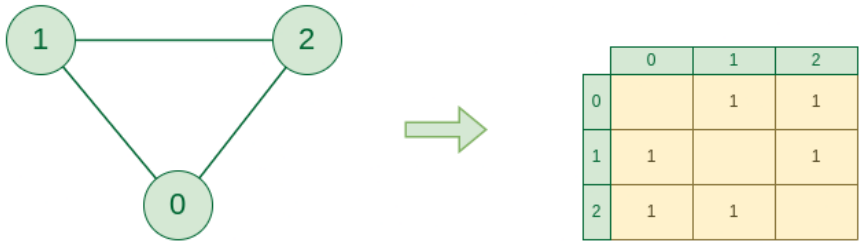
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)

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**.

Representation of Undirected Graph as Adjacency Matrix:

If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat[destination])** because we can go either way.



Undirected Graph

Adjacency Matrix

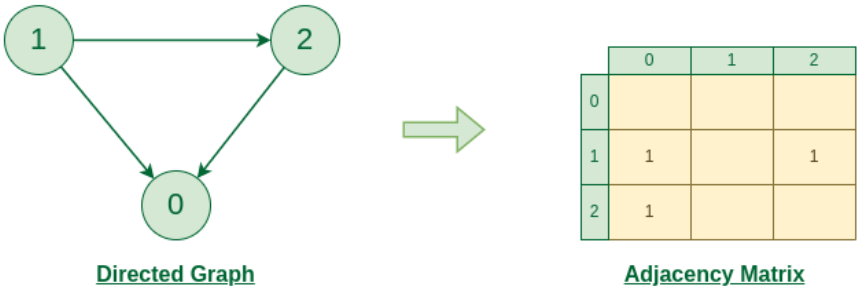
Graph Representation of Undirected graph to Adjacency Matrix

```
#include<stdio.h>

#define V 4

void addEdge(int mat[V][V], int i, int j) {
    mat[i][j] = 1;
    mat[j][i] = 1; // Since the graph is undirected
}

void displayMatrix(int mat[V][V]) {
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}

int main() {
    // Create a graph with 4 vertices and no edges
    // Note that all values are initialized as 0
    int mat[V][V] = {0};
```

```
    // Now add edges one by one
    addEdge(mat, 0, 1);
    addEdge(mat, 0, 2);
    addEdge(mat, 1, 2);
    addEdge(mat, 2, 3);

    /* Alternatively, we can also create using the below
       code if we know all edges in advance

    int mat[V][V] = {
        {0, 1, 0, 0},
        {1, 0, 1, 0},
        {0, 1, 0, 1},
        {0, 0, 1, 0}
    }; */

    printf("Adjacency Matrix Representation\n");
    displayMatrix(mat);

    return 0;
}
```

```
Output:
    Adjacency Matrix Representation
    0 1 1 0
    1 0 1 0
    1 1 0 1
    0 0 1 0
```

Representation of Directed Graph as Adjacency Matrix:



Graph Representation of Directed graph to Adjacency Matrix

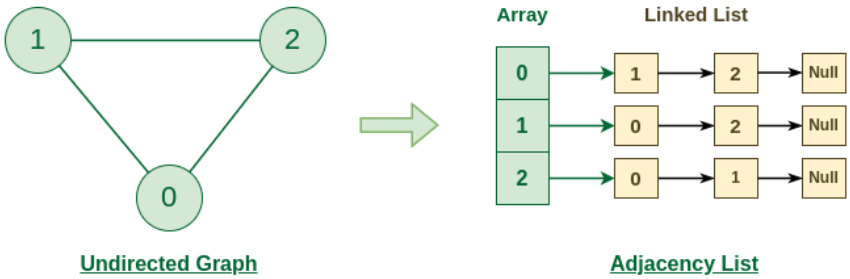
b) Adjacency Lists

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

Representation of Undirected Graph as Adjacency list:



Graph Representation of Undirected graph to Adjacency List

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

// Structure for a node in the adjacency list
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode =
      (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to add an edge between two vertices
void addEdge(struct Node* adj[], int i, int j) {
    struct Node* newNode = createNode(j);
    newNode->next = adj[i];
    adj[i] = newNode;

    newNode = createNode(i); // For undirected graph
    newNode->next = adj[j];
    adj[j] = newNode;
}

// Function to display the adjacency list
void displayAdjList(struct Node* adj[], int V) {
    for (int i = 0; i < V; i++) {
        printf("%d: ", i); // Print the vertex
        struct Node* temp = adj[i];
        while (temp != NULL) {
```

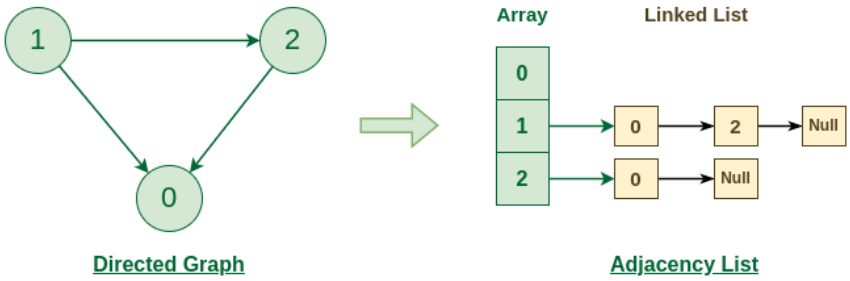
```
            printf("%d ", temp->data); // Print its adjacent
            temp = temp->next;
        }
        printf("\n");
    }
}

// Main function
int main() {

    // Create a graph with 4 vertices and no edges
    int V = 4;
    struct Node* adj[V];
    for (int i = 0; i < V; i++) {
        adj[i] = NULL; // Initialize adjacency list
    }

    // Now add edges one by one
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 1, 2);
    addEdge(adj, 2, 3);

    printf("Adjacency List Representation:\n");
    displayAdjList(adj, V);

    return 0;
}
```

Output:

```
Adjacency List Representation:
0: 1 2
1: 0 2
```

```
2: 0 1 3
3: 2
```

Representation of Directed Graph as Adjacency list:



Directed Graph

Adjacency List

Graph Representation of Directed graph to Adjacency List

BFS

Breadth First Search (BFS) is a fundamental **graph traversal algorithm.** It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed. This is different from **DFS** in a way that closest vertices are visited before others.

Initialization: Enqueue the given source vertex into a queue and mark it as visited.

1. **Exploration:** While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbor of the dequeued node:
        - Enqueue the neighbor into the queue.
        - Mark the neighbor as visited.
2. **Termination:** Repeat step 2 until the queue is empty.

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

```
#define MAX 100

// BFS from given source s
void bfs(int adj[MAX][MAX], int V, int s) {

    // Create a queue for BFS
    int q[MAX], front = 0, rear = 0;

    // Initially mark all the vertices as not visited
    // When we push a vertex into the q, we mark it as
    // visited
    bool visited[MAX] = { false };

    // Mark the source node as visited and enqueue it
    visited[s] = true;
    q[rear++] = s;

    // Iterate over the queue
    while (front < rear) {

        // Dequeue a vertex and print it
        int curr = q[front++];
        printf("%d ", curr);

        // Get all adjacent vertices of the dequeued vertex
        // If an adjacent has not been visited, mark it visited
        for (int i = 0; i < V; i++) {
            if (adj[curr][i] == 1 && !visited[i]) {
                visited[i] = true;
                q[rear++] = i;
            }
        }
    }
}
```

```
// Function to add an edge to the graph
void addEdge(int adj[MAX][MAX], int u, int v) {
    adj[u][v] = 1;
    adj[v][u] = 1;  // Undirected graph
}

int main() {
    // Number of vertices in the graph
    int V = 5;

    // Adjacency matrix representation of the graph
    int adj[MAX][MAX] = {0};

    // Add edges to the graph
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 4);

    // Perform BFS traversal starting from vertex 0
    printf("BFS starting from 0:\n");
    bfs(adj, V, 0);

    return 0;
}
```

Output:

```
BFS starting from 0 :
0 1 2 3 4
```

Time Complexity: O(V+E), where V is the number of nodes and E is the number of edges.

• BFS explores all the vertices and edges in the graph. In the worst case, it visits

every vertex and edge once. Therefore, the time complexity of BFS is O (V + E), where V and E are the number of vertices and edges in the given graph.

Auxiliary Space: O(V)

Applications of BFS in Graphs:

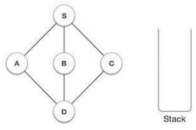
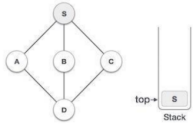
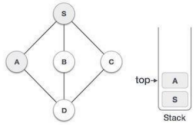
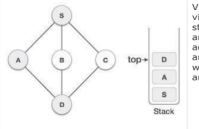
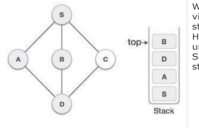
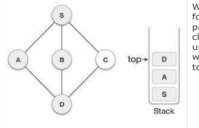
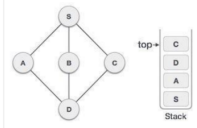
BFS has various applications in graph theory and computer science, including:

- **Shortest Path Finding**
- **Cycle Detection**
- **Connected Components**
- **Topological Sorting**
- **Level Order Traversal of Binary Trees**
- **Network Routing**

DFS

Depth First Search (or DFS) for a graph is similar to **Depth First Traversal of a tree**. Like trees, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. After we finish traversing one adjacent vertex and its reachable vertices, we move to the next adjacent vertex and repeat the process.

- Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1.		Initialize the stack.
2.		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4.		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7.		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

- As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

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

// Node structure for adjacency list
struct Node {
    int dest;
    struct Node* next;
};

// Structure to represent an adjacency list
struct AdjList {
    struct Node* head;
};

// Function to create a new adjacency list node
struct Node* createNode(int dest) {
    struct Node* newNode =
      (struct Node*)malloc(sizeof(struct Node));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}
```

```
// Recursive function for DFS traversal
void DFSRec(struct AdjList adj[], int visited[], int s) {
    // Mark the current vertex as visited
    visited[s] = 1;

    // Print the current vertex
    printf("%d ", s);

    // Traverse all adjacent vertices that are not visited yet
    struct Node* current = adj[s].head;
    while (current != NULL) {
        int dest = current->dest;
        if (!visited[dest]) {
            DFSRec(adj, visited, dest);
        }
        current = current->next;
    }
}

// Main DFS function that initializes the visited array
// and calls DFSRec
void DFS(struct AdjList adj[], int V, int s) {
    // Initialize visited array to false
    int visited[5] = {0};
    DFSRec(adj, visited, s);
}

// Function to add an edge to the adjacency list
void addEdge(struct AdjList adj[], int s, int t) {
    // Add edge from s to t
    struct Node* newNode = createNode(t);
    newNode->next = adj[s].head;
    adj[s].head = newNode;

    // Due to undirected Graph
    newNode = createNode(s);
```

```
    newNode->next = adj[t].head;
    adj[t].head = newNode;
}

int main() {
    int V = 5;

    // Create an array of adjacency lists
    struct AdjList adj[V];

    // Initialize each adjacency list as empty
    for (int i = 0; i < V; i++) {
        adj[i].head = NULL;
    }

    int E = 5;
    // Define the edges of the graph
    int edges[][2] = {{1, 2}, {1, 0}, {2, 0}, {2, 3}, {2, 4}};

    // Populate the adjacency list with edges
    for (int i = 0; i < E; i++) {
        addEdge(adj, edges[i][0], edges[i][1]);
    }

    int source = 1;
    printf("DFS from source: %d\n", source);
    DFS(adj, V, source);

    return 0;
}
```

Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Auxiliary Space: O(V + E), since an extra visited array of size V is required, And stack size for recursive calls to DFSRec function.

1. BFS and DFS using adjacency matrix represented graph

```
#include <stdio.h>

#define MAX_VERTICES 5

// Graph represented as an adjacency matrix
int adjMatrix[MAX_VERTICES][MAX_VERTICES];

// Visited array to keep track of visited vertices
int visited[MAX_VERTICES];

// Queue and Stack implemented with simple global variables
int queue[MAX_VERTICES];
int front = -1, rear = -1;

int stack[MAX_VERTICES];
int top = -1;

// Queue operations for BFS
int isQueueEmpty() {
    return front == -1;
}

void enqueue(int value) {
    if (rear == MAX_VERTICES - 1) {
        printf("Queue is full\n");
    } else {
        if (front == -1) front = 0;
        queue[++rear] = value;
    }
}

int dequeue() {
    if (isQueueEmpty()) {
        printf("Queue is empty\n");
```

```
        return -1;
    } else {
        int item = queue[front];
        if (front >= rear) {
            front = rear = -1;  // Reset queue when empty
        } else {
            front++;
        }
        return item;
    }
}

// Stack operations for DFS
int isStackEmpty() {
    return top == -1;
}

void push(int value) {
    if (top == MAX_VERTICES - 1) {
        printf("Stack is full\n");
    } else {
        stack[++top] = value;
    }
}

int pop() {
    if (isStackEmpty()) {
        printf("Stack is empty\n");
        return -1;
    } else {
        return stack[top--];
    }
}

// Function to add an edge to the graph (using adjacency matrix)
void addEdge(int src, int dest) {
```

```
    adjMatrix[src][dest] = 1;
    adjMatrix[dest][src] = 1;  // Since it's an undirected graph
}

// BFS traversal using global queue
void bfs(int startVertex) {
    // Reset visited array and queue for BFS
    for (int i = 0; i < MAX_VERTICES; i++) visited[i] = 0;
    front = rear = -1;

    visited[startVertex] = 1;
    enqueue(startVertex);

    printf("BFS starting from vertex %d: ", startVertex);

    while (!isQueueEmpty()) {
        int currentVertex = dequeue();
        printf("%d ", currentVertex);

        for (int i = 0; i < MAX_VERTICES; i++) {
            if (adjMatrix[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = 1;
                enqueue(i);
            }
        }
    }
    printf("\n");
}

// DFS traversal using global stack
void dfs(int startVertex) {
    // Reset visited array and stack for DFS
    for (int i = 0; i < MAX_VERTICES; i++) visited[i] = 0;
    top = -1;

    push(startVertex);
```

```
    printf("DFS starting from vertex %d: ", startVertex);

    while (!isStackEmpty()) {
        int currentVertex = pop();

        if (!visited[currentVertex]) {
            printf("%d ", currentVertex);
            visited[currentVertex] = 1;
        }

        for (int i = MAX_VERTICES - 1; i >= 0; i--) {
            if (adjMatrix[currentVertex][i] == 1 && !visited[i])
                push(i);
            }
        }
    }
    printf("\n");
}

int main() {
    // Initialize adjacency matrix
    for (int i = 0; i < MAX_VERTICES; i++) {
        for (int j = 0; j < MAX_VERTICES; j++) {
            adjMatrix[i][j] = 0;
        }
    }

    // Adding edges to the graph
    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(2, 3);
    addEdge(3, 4);

    // Perform BFS and DFS traversals
```

```
    bfs(0);
    dfs(0);
    return 0;
}
```

2. BFS and DFS with Linked list representation of the graph

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES 5
// Node structure for the adjacency list
struct Node
{
    int vertex;
    struct Node *next;
};

// Array of pointers for adjacency list representation
struct Node *adjList[MAX_VERTICES];
// Visited array to keep track of visited vertices
int visited[MAX_VERTICES];
// Queue and Stack implemented with simple global variables
int queue[MAX_VERTICES];
int front = -1, rear = -1;
int stack[MAX_VERTICES];
int top = -1;
// Queue operations for BFS
int isQueueEmpty()
{
    return front == -1;
}
void enqueue(int value)
{
    if (rear == MAX_VERTICES - 1)
    {
```

```
        printf("Queue is full\n");
    }
    else
    {
        if (front == -1)
            front = 0;
        queue[++rear] = value;
    }
}
int dequeue()
{
    if (isQueueEmpty())
    {
        printf("Queue is empty\n");
        return -1;
    }
    else
    {
        int item = queue[front];
        if (front >= rear)
        {
            front = rear = -1; // Reset queue when empty
        }
        else
        {
            front++;
        }
        return item;
    }
}
// Stack operations for DFS
int isStackEmpty()
{
    return top == -1;
}
void push(int value)
```

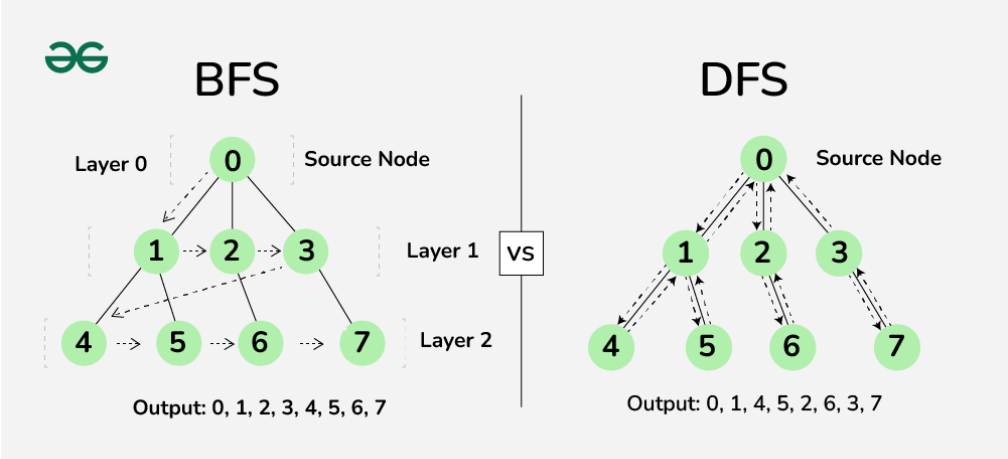
```
{
    if (top == MAX_VERTICES - 1)
    {
        printf("Stack is full\n");
    }
    else
    {

        stack[++top] = value;
    }
}
int pop()
{
    if (isStackEmpty())
    {
        printf("Stack is empty\n");
        return -1;
    }
    else
    {
        return stack[top--];
    }
}
// Function to add an edge to the graph (using adjacency list)
void addEdge(int src, int dest)
{
    struct Node *newNode = (struct Node *)malloc(sizeof(struct N
    newNode->vertex = dest;
    newNode->next = adjList[src];
    adjList[src] = newNode;
    // Since the graph is undirected, add an edge from dest to s
    newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->vertex = src;
    newNode->next = adjList[dest];
    adjList[dest] = newNode;
}
```

```
// BFS traversal using global queue
void bfs(int startVertex)
{
    // Reset visited array and queue for BFS
    for (int i = 0; i < MAX_VERTICES; i++)
        visited[i] = 0;
    front = rear = -1;
    visited[startVertex] = 1;
    enqueue(startVertex);
    printf("BFS starting from vertex %d: ", startVertex);
    while (!isQueueEmpty())
    {
        int currentVertex = dequeue();
        printf("%d ", currentVertex);
        struct Node *temp = adjList[currentVertex];
        while (temp)
        {
            int adjVertex = temp->vertex;
            if (!visited[adjVertex])
            {
                visited[adjVertex] = 1;
                enqueue(adjVertex);
            }
            temp = temp->next;
        }
    }
    printf("\n");
}
// DFS traversal using global stack
void dfs(int startVertex)
{
    // Reset visited array and stack for DFS
    for (int i = 0; i < MAX_VERTICES; i++)
        visited[i] = 0;
    top = -1;
    push(startVertex);
```

```
    printf("DFS starting from vertex %d: ", startVertex);
    while (!isStackEmpty())
    {
        int currentVertex = pop();
        if (!visited[currentVertex])
        {
            printf("%d ", currentVertex);
            visited[currentVertex] = 1;
        }
        struct Node *temp = adjList[currentVertex];
        while (temp)
        {
            int adjVertex = temp->vertex;
            if (!visited[adjVertex])
            {
                push(adjVertex);
            }
            temp = temp->next;
        }
    }
    printf("\n");
}
int main()
{
    // Initialize adjacency list and visited array
    for (int i = 0; i < MAX_VERTICES; i++)
    {
        adjList[i] = NULL;
        visited[i] = 0;
    }
    // Adding edges to the graph
    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(2, 3);
    addEdge(3, 4);
```

```
    // Perform BFS and DFS traversals
    bfs(0);
    dfs(0);

    return 0;
}
```

BFS vs DFS



Difference Between BFS and DFS

Parameters	BFS	DFS
Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.

Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
Applications	BFS is used in various applications such as bipartite graphs, shortest paths, etc. If weight of every edge is same, then BFS gives shortest pat from source to every other vertex.	DFS is used in various applications such as acyclic graphs and finding strongly connected components etc. There are many applications where both BFS and DFS can be used like Topological Sorting, Cycle Detection, etc.

Dijkstra’s Algorithm

Step 1: Initialization

1. Assign a tentative distance value to every node:
   o Source node: Distance = 0.
   o All other nodes: Distance = ∞ (unreachable initially).
2. Set the previous node for every node as "undefined."
3. Create a visited set to keep track of nodes that are processed (initially empty).

Step 2: Select the Node with the Smallest Tentative Distance

1. Pick the node with the smallest tentative distance from the unvisited nodes.
2. If all unvisited nodes have a distance of ∞, the algorithm stops (the remaining nodes are
   unreachable).

Step 3: Update Distances of Neighbors

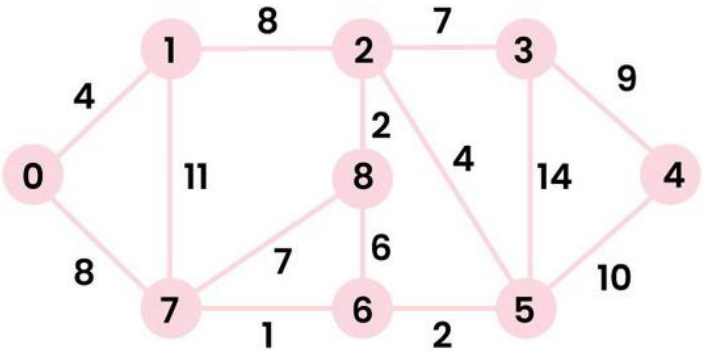
1. For each unvisited neighbor of the current node:
   o Calculate the new distance: New Distance=Current Node Distance+Edge Weight
   o If the new distance is smaller than the currently assigned distance for the neighbor,
   update it.

2. Update the previous node of the neighbor to the current node.

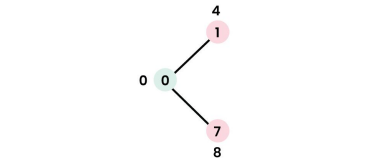
Step 4: Mark the Current Node as Visited

- Add the current node to the visited set. Once a node is visited, its distance will not change.

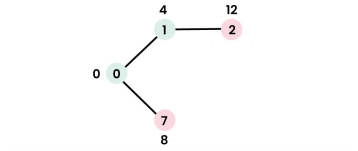
Step 5: Repeat



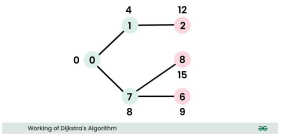
Working of Dijkstra's Algorithm



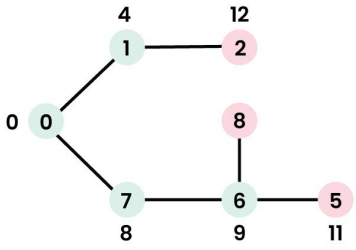
Working of Dijkstra's Algorithm



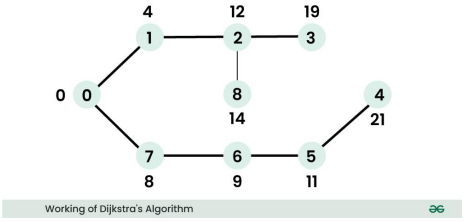
Working of Dijkstra's Algorithm



Working of Dijkstra's Algorithm



Working of Dijkstra's Algorithm



Working of Dijkstra's Algorithm

```
// C program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
```

```
// representation of the graph

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t\t\t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
```

```
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the
                 // shortest
    // distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is
                    // included in shortest
    // path tree or shortest distance from src to i is
    // finalized

    // Initialize all distances as INFINITE and stpSet[] as
    // false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of
        // vertices not yet processed. u is always equal to
        // src in the first iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the
        // picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet,
            // there is an edge from u to v, and total
            // weight of path from src to  v through u is
```

```
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v]
                && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}

// driver's code
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    // Function call
    dijkstra(graph, 0);

    return 0;
}
```

MST

A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together without any cycles and with

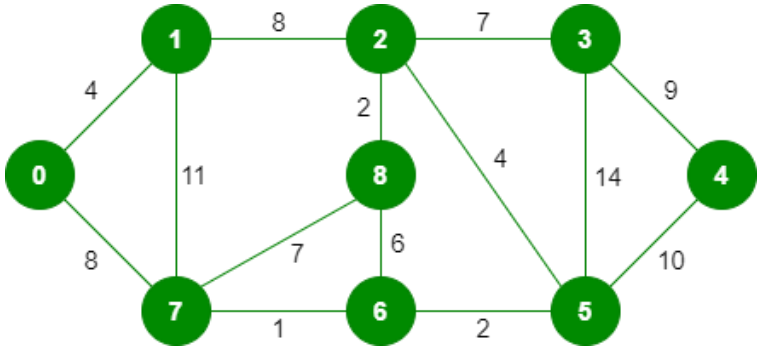
the minimum possible total edge weight.

Kruskal's Minimum Spanning Tree (MST) Algorithm

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last.

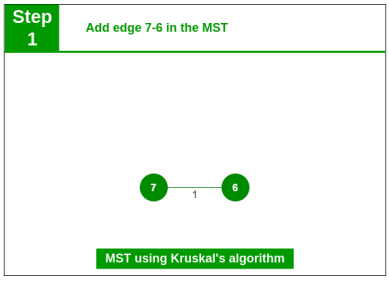
Below are the steps for finding MST using Kruskal's algorithm:

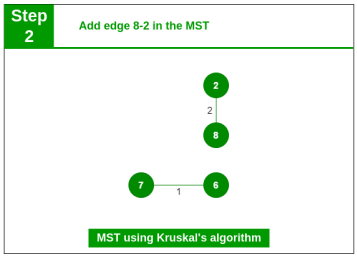
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

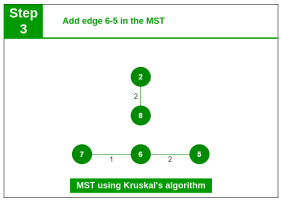


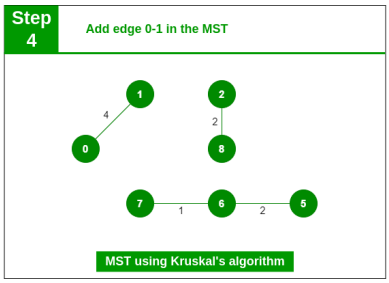
Weight	*Source*	*Destination*
1	*7*	*6*
2	*8*	*2*
2	*6*	*5*
4	*0*	*1*
4	*2*	*5*
6	*8*	*6*
7	*2*	*3*
7	*7*	*8*
8	*0*	*7*

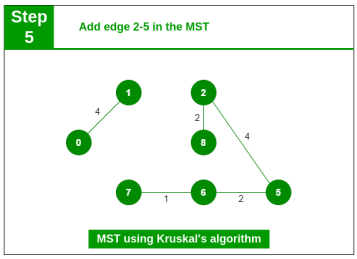
8	*1*	*2*
9	*3*	*4*
10	*5*	*4*
11	*1*	*7*
14	*3*	*5*

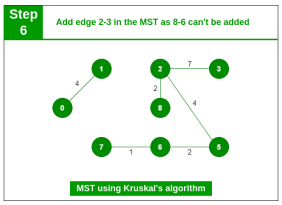


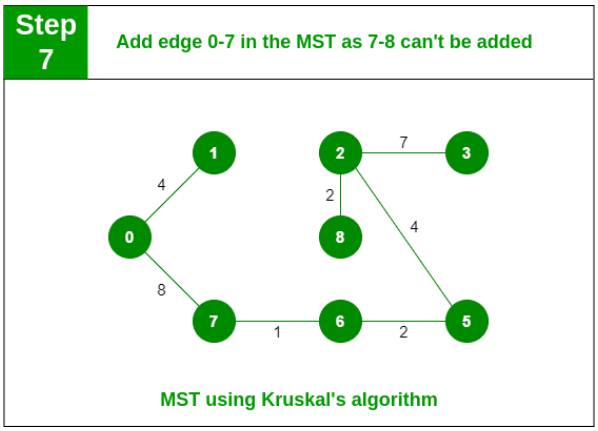


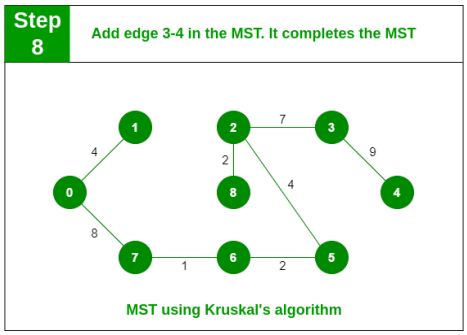













```
// C code to implement Kruskal's algorithm
```

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

// Comparator function to use in sorting
int comparator(const void* p1, const void* p2)
{
    const int(*x)[3] = p1;
    const int(*y)[3] = p2;

    return (*x)[2] - (*y)[2];
}

// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

// Function to find the parent of a node
int findParent(int parent[], int component)
{
    if (parent[component] == component)
        return component;

    return parent[component]
           = findParent(parent, parent[component]);
}

// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
    // Finding the parents
    u = findParent(parent, u);
```

```
    v = findParent(parent, v);

    if (rank[u] < rank[v]) {
        parent[u] = v;
    }
    else if (rank[u] > rank[v]) {
        parent[v] = u;
    }
    else {
        parent[v] = u;

        // Since the rank increases if
        // the ranks of two sets are same
        rank[u]++;
    }
}

// Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
    // First we sort the edge array in ascending order
    // so that we can access minimum distances/cost
    qsort(edge, n, sizeof(edge[0]), comparator);

    int parent[n];
    int rank[n];

    // Function to initialize parent[] and rank[]
    makeSet(parent, rank, n);

    // To store the minimun cost
    int minCost = 0;

    printf(
        "Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++) {
```

```
        int v1 = findParent(parent, edge[i][0]);
        int v2 = findParent(parent, edge[i][1]);
        int wt = edge[i][2];

        // If the parents are different that
        // means they are in different sets so
        // union them
        if (v1 != v2) {
            unionSet(v1, v2, parent, rank, n);
            minCost += wt;
            printf("%d -- %d == %d\n", edge[i][0],
                   edge[i][1], wt);
        }
    }

    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

// Driver code
int main()
{
    int edge[5][3] = { { 0, 1, 10 },
                       { 0, 2, 6 },
                       { 0, 3, 5 },
                       { 1, 3, 15 },
                       { 2, 3, 4 } };

    kruskalAlgo(5, edge);

    return 0;
}
```

Output

```
Following are the edges in the constructed MST
2 -- 3 == 4
```

```
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

Time Complexity: O(E * logE) or O(E * logV)

- Sorting of edges takes O(E * logE) time.
- After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(logV) time.
- So overall complexity is O(E * logE + E * logV) time.
- The value of E can be at most O(V^2), so O(logV) and O(logE) are the same. Therefore, the overall time complexity is O(E * logE) or O(E*logV)

Auxiliary Space: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Prim’s Algorithm for Minimum Spanning Tree (MST)

The working of Prim’s algorithm can be described by using the following steps:

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.

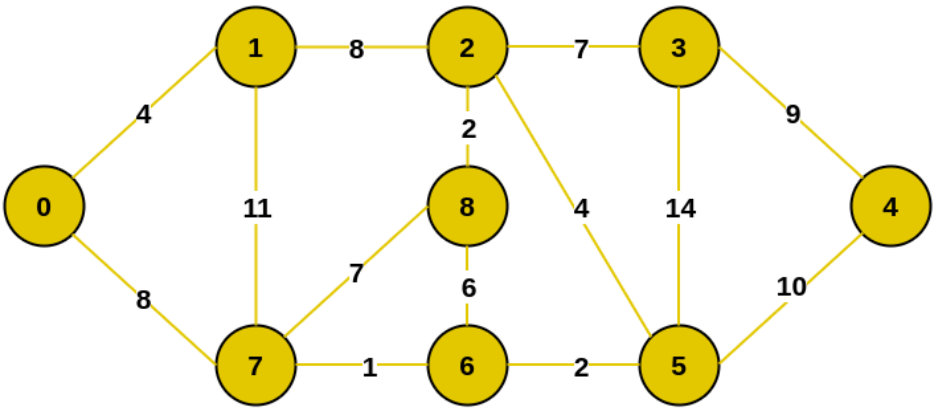
Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).

Step 3: Find edges connecting any tree vertex with the fringe vertices.

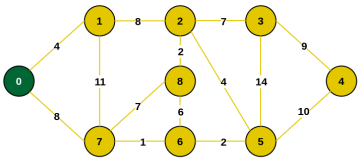
Step 4: Find the minimum among these edges.

Step 5: Add the chosen edge to the MST if it does not form any cycle.

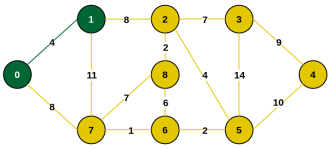
Step 6: Return the MST and exit



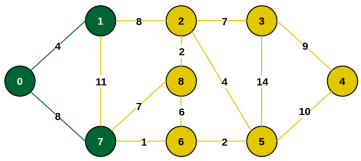
Example of a Graph



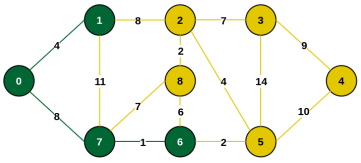
Select an arbitrary starting vertex. Here we have selected 0



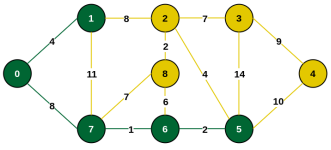
Minimum weighted edge from MST to other vertices is 0-1 with weight 4



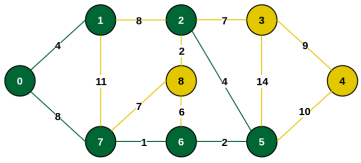
Minimum weighted edge from MST to other vertices is 0-7 with weight 8



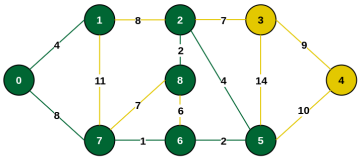
Minimum weighted edge from MST to other vertices is 7-6 with weight 1



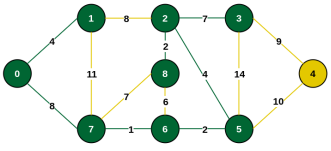
Minimum weighted edge from MST to other vertices is 6-5 with weight 2



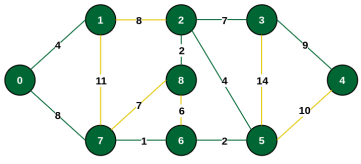
Minimum weighted edge from MST to other vertices is 5-2 with weight 4



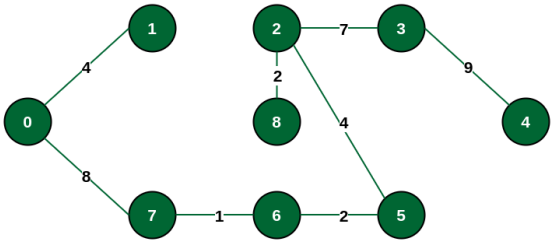
Minimum weighted edge from MST to other vertices is 2-8 with weight 2



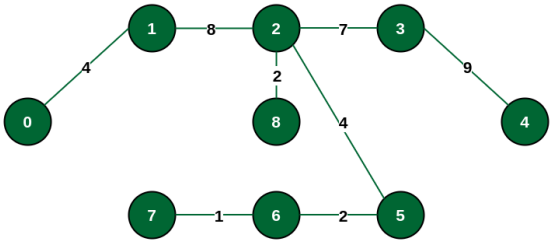
Minimum weighted edge from MST to other vertices is 2-3 with weight 7



Minimum weighted edge from MST to other vertices is 3-4 with weight 9



The final structure of MST



Alternative MST structure

```
// A C program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}
```

```
// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i,
               graph[i][parent[i]]);
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];
    // Key values used to pick minimum weight edge in cut
    int key[V];
    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first
    // vertex.
    key[0] = 0;

    // First node is always root of MST
    parent[0] = -1;

    // The MST will have V vertices
```

```
    for (int count = 0; count < V - 1; count++) {

        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
            // vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only
            // if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false
                && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

// Driver's code
int main()
{
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };
```

```
    // Print the solution
    primMST(graph);

    return 0;
}
```

Output

```
Edge     Weight
0 - 1     2
1 - 2     3
0 - 3     6
1 - 4     5
```

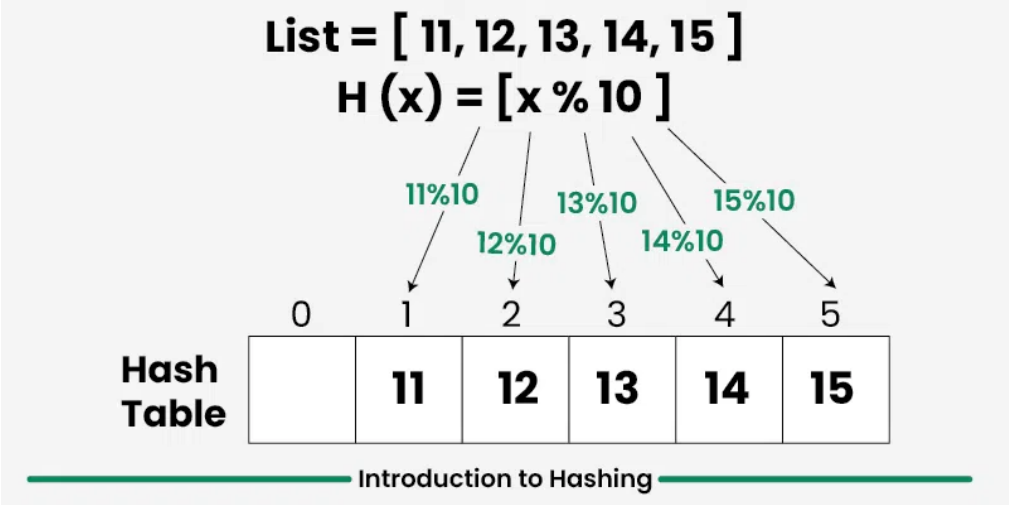
Time Complexity: O(V^2), If the input **graph is represented using an adjacency list**, then the time complexity of Prim's algorithm can be reduced to O(E * logV) with the help of a binary heap. In this implementation, we are always considering the spanning tree to start from the root of the graph

Auxiliary Space: O(V)

Hashing

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

- Hashing involves mapping data to a specific index in a hash table (an array of items) using a **hash function** that enables fast retrieval of information based on its key.
- The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average.

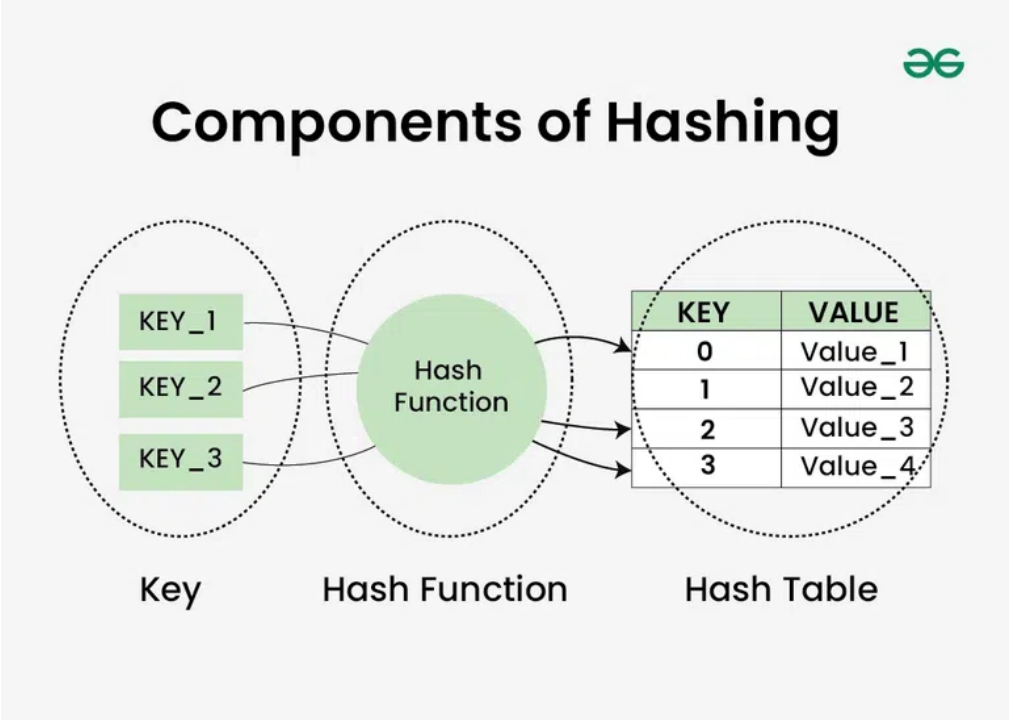


- There are mainly two forms of hash typically implemented in programming languages.
- **Hash Set** : Collection of unique keys (Implemented as **Set in Python**, **Set in JavaScript**, **unordered_set in C++** and **HashSet in Java**.
- **Hash Map** : Collection of key value pairs with keys being unique (Implemented as **dictionary in Python, Map in JavaScript**, **unordered_map in C++** and **HashMap in Java**)

Components of Hashing

There are majorly three components of hashing:

1. **Key:** A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** Receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index .
3. **Hash Table:** Hash table is typically an array of lists. It stores values corresponding to the keys. Hash stores the data in an associative manner in an array where each data value has its own unique index



What is a Hash function?

A **hash function** creates a mapping from an input key to an index in hash table, this is done through the use of mathematical formulas known as hash functions.

Types of Hash functions:

There are many hash functions that use numeric or alphanumeric keys.

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 m is a prime number.

   Advantages:

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

   Disadvantages:

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

2. **Multiplication Method**

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

   > h(k)=⌊m(kAmod1)⌋
   > 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.

4. **Folding Method.**

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

   Steps:

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

   Advantages:

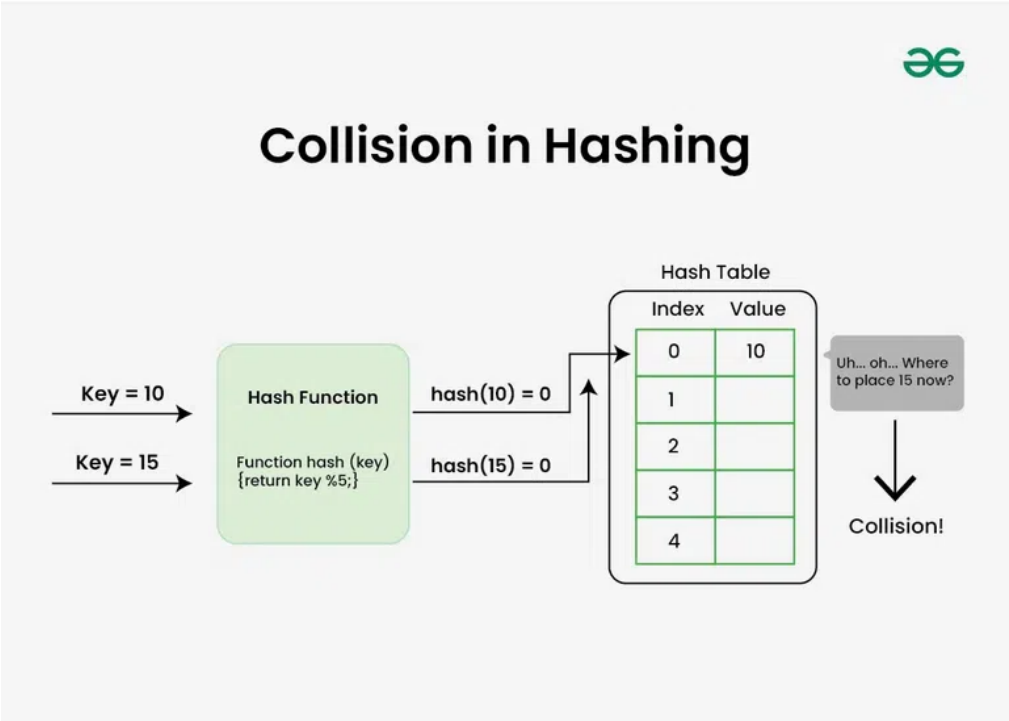
   - Simple and easy to implement.

Disadvantages:

- Depends on the choice of partitioning scheme.

Collision:

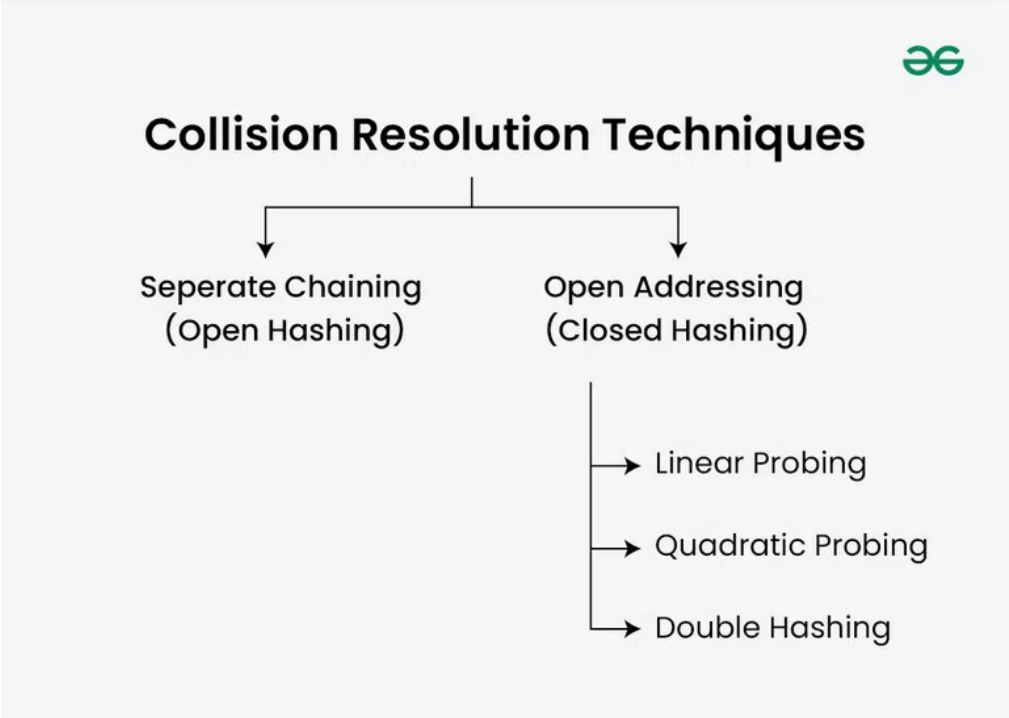
A collision occurs when two different keys hash to the same index in the hash table. Efficient handling of collisions is crucial for maintaining the performance of a hash table.



Collision Handling techniques

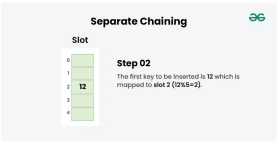
There are mainly two methods to handle collision:

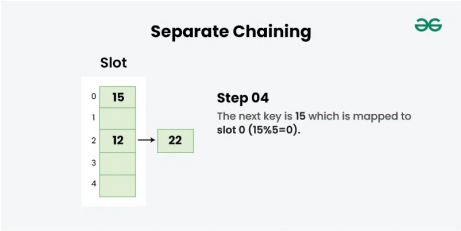
1. Separate Chaining
2. Open Addressing

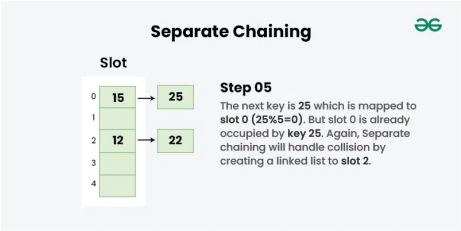


Separate Chaining

The idea behind separate chaining is to implement the array as a linked list called a chain. Uses a linked list to store multiple elements that hash to the same index. Each slot in the hash table points to a linked list of entries that map to the same index.







Open Addressing

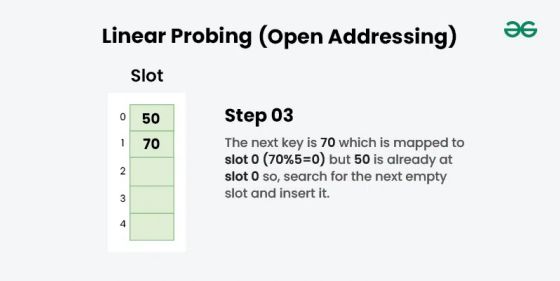
Open Addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. When a collision occurs, open addressing finds another slot using a probing sequence.

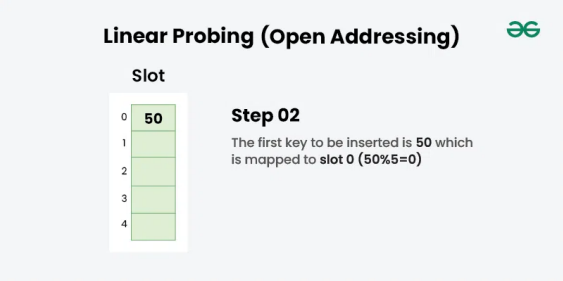
Different ways of Open Addressing:

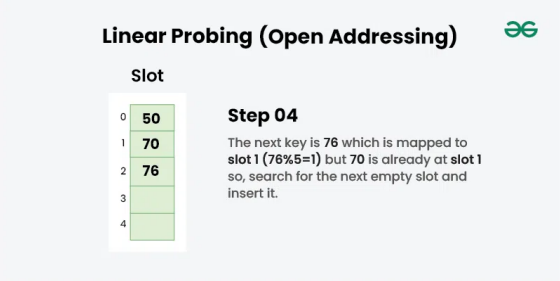
1. **Linear Probing:**

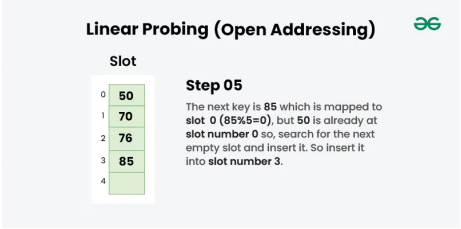
   In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

   > The function used for rehashing is as follows:
   > rehash(key) = (n+1)%table-size.









2. **Quadratic Probing**

   Uses a quadratic function to compute the next index.

   > *let hash(x) be the slot index computed using hash function.*

   > *If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*

> If (hash(x) + 1∗1) % S is also full, then we try (hash(x) + 2∗2) % S

> If (hash(x) + 2∗2) % S is also full, then we try (hash(x) + 3∗3) % S

3. Double Hashing

   Uses a secondary hash function to compute the next index.

> *let hash(x) be the slot index computed using hash function.*

> *If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S*

> If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

> If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S