

# 1. Write program to implement basic operations of Binary Search Tree.

## Algorithm:

### 1. Insertion:

**Step 1:** If the tree is empty, create a new node as the root.

**Step 2:** If the value  $<$  current node's data, recursively insert into the left subtree.

**Step 3:** If the value  $>$  current node's data, recursively insert into the right subtree.

### 2. Traversal:

**Inorder:** Traverse left  $\rightarrow$  visit root  $\rightarrow$  traverse right.

**Preorder:** Visit root  $\rightarrow$  traverse left  $\rightarrow$  traverse right.

**Postorder:** Traverse left  $\rightarrow$  traverse right  $\rightarrow$  visit root.

### 3. Deletion:

**Case 1:** Node has no children  $\rightarrow$  Delete directly.

**Case 2:** Node has one child  $\rightarrow$  Replace with its child.

**Case 3:** Node has two children  $\rightarrow$  Replace with its in order successor, then delete the successor.

## Example:

1. Insert: 50, 30, 70, 20, 40, 60, 80.
2. Inorder Traversal: 20 30 40 50 60 70 80.
3. Delete 20 (leaf node).
4. Delete 30 (node with one child).
5. Delete 50 (root with two children).

Inorder after deletions: 40 60 70 80

## Program:

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

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

```
// BST Node structure

typedef struct Node {
    int data;
    struct Node * left, * right;
}
Node;
```

```
// Create a new node

Node * createNode(int value) {
    Node * newNode = (Node * ) malloc(sizeof(Node));
    newNode -> data = value;
    newNode -> left = newNode -> right = NULL;
    return newNode;
}
```

```
// Insert a node into BST

Node * insert(Node * root, int value) {
    if (root == NULL) return createNode(value);
    if (value < root -> data) root -> left = insert(root -> left, value);
    else if (value > root -> data) root -> right = insert(root -> right, value);
    return root;
}
```

```
// Find inorder successor (smallest in right subtree)

Node * minValueNode(Node * node) {
    Node * current = node;
    while (current && current -> left != NULL) current = current -> left;
    return current;
}
```

// Delete a node from BST

```
Node * deleteNode(Node * root, int value) {  
    if (root == NULL) return root;  
  
    if (value < root -> data) root -> left = deleteNode(root -> left, value);  
    else if (value > root -> data) root -> right = deleteNode(root -> right, value);  
    else {  
        // Node with one or no child  
        if (root -> left == NULL) {  
            Node * temp = root -> right;  
            free(root);  
            return temp;  
        } else if (root -> right == NULL) {  
            Node * temp = root -> left;  
            free(root);  
            return temp;  
        }  
        // Node with two children: replace with inorder successor  
        Node * temp = minValueNode(root -> right);  
        root -> data = temp -> data;  
        root -> right = deleteNode(root -> right, temp -> data);  
    }  
    return root;  
}
```

// Inorder traversal

```
void inorder(Node * root) {  
    if (root != NULL) {  
        inorder(root -> left);  
        printf("%d ", root -> data);  
    }  
}
```

```

        inorder(root -> right);
    }
}

int main() {
    Node * root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder after insertion: ");
    inorder(root);
    printf("\n");

    root = deleteNode(root, 20);
    root = deleteNode(root, 30);
    root = deleteNode(root, 50);
    printf("Inorder after deletions: ");
    inorder(root);

    return 0;
}

```

### **Output:**

```

C:\Users\Suresh Dahal\dsa>a.exe
Inorder after insertion: 20 30 40 50 60 70 80
Inorder after deletions: 40 60 70 80
C:\Users\Suresh Dahal\dsa>

```

## 2. Write programs to implement sorting algorithms: Bubble, Insertion, Selection, Merge and Quick

### Bubble Sort

#### Algorithm:

1. **Step 1:** Start with the first element. Compare it with the next element.
2. **Step 2:** If the current element  $>$  next element, swap them.
3. **Step 3:** Move to the next pair and repeat Step 2 for all adjacent pairs in the array.
4. **Step 4:** Repeat Steps 1-3 for  $n-1$  passes (where  $n$  is the array size).
5. **Step 5:** If no swaps occur in a pass, the array is sorted.

#### Example:

**Input Array:** [4, 2, 0, 1, 3]

#### **Pass 1:**

- Compare 4 & 2  $\rightarrow$  swap  $\rightarrow$  [2, 4, 0, 1, 3]
- Compare 4 & 0  $\rightarrow$  swap  $\rightarrow$  [2, 0, 4, 1, 3]
- Compare 4 & 1  $\rightarrow$  swap  $\rightarrow$  [2, 0, 1, 4, 3]
- Compare 4 & 3  $\rightarrow$  swap  $\rightarrow$  [2, 0, 1, 3, 4]

#### **Pass 2:**

- Compare 2 & 0  $\rightarrow$  swap  $\rightarrow$  [0, 2, 1, 3, 4]
- Compare 2 & 1  $\rightarrow$  swap  $\rightarrow$  [0, 1, 2, 3, 4]
- Compare 2 & 3  $\rightarrow$  no swap
- No further swaps

#### **Pass 3:**

- Compare 0 & 1  $\rightarrow$  no swap
- Compare 1 & 2  $\rightarrow$  no swap
- No swaps  $\rightarrow$  Sorting stops early.

**Output:** [0, 1, 2, 3, 4]

**Program:**

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    int swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = 0;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        // Early termination if no swaps
        if (swapped == 0) break;
    }
}

int main() {
    int arr[] = { 4,2,0,1,3 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    bubbleSort(arr, n);
    printf("\nSorted array: ");
```

```

for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
}

return 0;
}

```

### **Output:**

```

C:\Users\Suresh Dahal\dsa>a.exe
Original array: 4 2 0 1 3
Sorted array: 0 1 2 3 4

```

## **Insertion Sort**

### **Algorithm:**

**Step 1:** Start with the second element in the array. Compare it with the first element.

**Step 2:** If the current element is smaller than the previous element, shift the previous elements to the right to make space for the current element.

**Step 3:** Insert the current element into the correct position.

**Step 4:** Move to the next element and repeat steps 2-3 for all elements in the array.

**Step 5:** Continue until all elements have been inserted into their correct positions.

### **Example:**

**Input Array:** [4, 2, 0, 1, 3]

#### **Pass 1:**

Compare 2 with 4 → 2 is smaller → shift 4 → [4, 4, 0, 1, 3]  
 Insert 2 → [2, 4, 0, 1, 3]

#### **Pass 2:**

Compare 0 with 4 → 0 is smaller → shift 4 → [2, 4, 4, 1, 3]  
 Compare 0 with 2 → 0 is smaller → shift 2 → [2, 2, 4, 1, 3]  
 Insert 0 → [0, 2, 4, 1, 3]

#### **Pass 3:**

Compare 1 with 4 → 1 is smaller → shift 4 → [0, 2, 4, 4, 3]  
 Compare 1 with 2 → 1 is smaller → shift 2 → [0, 2, 2, 4, 3]  
 Insert 1 → [0, 1, 2, 4, 3]

**Pass 4:**

Compare 3 with 4 → 3 is smaller → shift 4 → [0, 1, 2, 4, 4]

Insert 3 → [0, 1, 2, 3, 4]

**Output:** [0, 1, 2, 3, 4]

**Program:**

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

```
// Recursive function to perform insertion sort
```

```
void insertionSort(int arr[], int n) {
```

```
    // Base case: if there is only one element or no elements, return
```

```
    if (n <= 1) {
```

```
        return;
```

```
    }
```

```
    // Sort first n-1 elements
```

```
    insertionSort(arr, n - 1);
```

```
    // Insert the nth element in the sorted part of the array
```

```
    int last = arr[n - 1];
```

```
    int j = n - 2;
```

```
    while (j >= 0 && arr[j] > last) {
```

```
        arr[j + 1] = arr[j];
```

```
        j--;
```

```
    }
```

```
    arr[j + 1] = last;
```

```
}
```

```
// Function to print the array
```

```
void printArray(int arr[], int n) {
```



```

    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {4,2,0,1,3};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

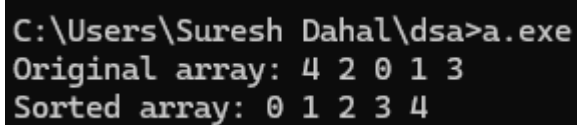
    insertionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}

```

### **Output:**



```

C:\Users\Suresh Dahal\dsa>a.exe
Original array: 4 2 0 1 3
Sorted array: 0 1 2 3 4

```

## **Selection Sort**

### **Algorithm:**

**Step 1:** Start with the first element. Find the smallest element in the remaining unsorted part of the array.

**Step 2:** Swap the smallest element with the first element.

**Step 3:** Move to the next element and repeat steps 1-2 for all elements except the last one (as it will be automatically sorted).

**Step 4:** Continue until the entire array is sorted.

**Example:**

**Input Array:** [4, 2, 0, 1, 3]

**Pass 1:**

Find the smallest element in the array [4, 2, 0, 1, 3] → smallest is 0.

Swap 4 and 0 → [0, 2, 4, 1, 3]

**Pass 2:**

Find the smallest element in the remaining array [2, 4, 1, 3] → smallest is 1.

Swap 2 and 1 → [0, 1, 4, 2, 3]

**Pass 3:**

Find the smallest element in the remaining array [4, 2, 3] → smallest is 2.

Swap 4 and 2 → [0, 1, 2, 4, 3]

**Pass 4:**

Find the smallest element in the remaining array [4, 3] → smallest is 3.

Swap 4 and 3 → [0, 1, 2, 3, 4]

**Output:** [0, 1, 2, 3, 4]

**Program:**

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

```
void selectionSort(int arr[], int n) {
```

```
    int minIndex, temp;
```

```
    // Move through the entire array
```

```
    for (int i = 0; i < n - 1; i++) {
```

```
        // Find the smallest element in the unsorted part of the array
```

```
        minIndex = i;
```

```
        for (int j = i + 1; j < n; j++) {
```

```
            if (arr[j] < arr[minIndex]) {
```

```
                minIndex = j;
```

```
            }
```

```
        }
```

```

// Swap the found smallest element with the first element of the unsorted part
if (minIndex != i) {
    temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
}
}
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = { 4,2,0,1,3 };
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    selectionSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}

```

### Output:

```
C:\Users\Suresh Dahal\dsa>a.exe
Original array: 4 2 0 1 3
Sorted array: 0 1 2 3 4
```

## Merge Sort

### Algorithm:

**Step 1:** Divide the array into two halves.

**Step 2:** Recursively apply the merge sort to both halves until you reach subarrays of size 1.

**Step 3:** Merge the two sorted halves back together into a single sorted array.

**Step 4:** Repeat steps 1-3 until the entire array is sorted.

### Example:

**Input Array:** [4, 2, 0, 1, 3]

**Step 1: Divide the array into two halves:**

- Left half: [4, 2, 0]
- Right half: [1, 3]

**Step 2: Recursively sort both halves:**

**Sorting [4, 2, 0]:**

- Divide into [4] and [2, 0]
- Sort [2, 0]:
  - Divide into [2] and [0]
  - Merge [2] and [0] → [0, 2]
  - Merge [4] and [0, 2] → [0, 2, 4]

**Sorting [1, 3]:**

- [1] and [3] are already sorted.

**Step 3: Merge [0, 2, 4] and [1, 3]:**

- Merging gives: [0, 1, 2, 3, 4]

**Output:** [0, 1, 2, 3, 4]

**Program:**

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

```
void merge(int arr[], int left, int mid, int right) {
```

```
    int n1 = mid - left + 1;
```

```
    int n2 = right - mid;
```

```
    // Create temporary arrays
```

```
    int leftArr[n1], rightArr[n2];
```

```
    // Copy data to temporary arrays
```

```
    for (int i = 0; i < n1; i++)
```

```
        leftArr[i] = arr[left + i];
```

```
    for (int j = 0; j < n2; j++)
```

```
        rightArr[j] = arr[mid + 1 + j];
```

```
    int i = 0, j = 0, k = left;
```

```
    // Merge the temporary arrays back into the original array
```

```
    while (i < n1 && j < n2) {
```

```
        if (leftArr[i] <= rightArr[j]) {
```

```
            arr[k] = leftArr[i];
```

```
            i++;
```

```
        } else {
```

```
            arr[k] = rightArr[j];
```

```
            j++;
```

```
        }
```

```
        k++;
```

```
    }
```

```
// Copy the remaining elements of leftArr[] if any
```

```
while (i < n1) {
```

```
    arr[k] = leftArr[i];
```

```
    i++;
```

```
    k++;
```

```
}
```

```
// Copy the remaining elements of rightArr[] if any
```

```
while (j < n2) {
```

```
    arr[k] = rightArr[j];
```

```
    j++;
```

```
    k++;
```

```
}
```

```
}
```

```
// Function to perform merge sort
```

```
void mergeSort(int arr[], int left, int right) {
```

```
    if (left < right) {
```

```
        int mid = left + (right - left) / 2;
```

```
        // Recursively sort the first and second halves
```

```
        mergeSort(arr, left, mid);
```

```
        mergeSort(arr, mid + 1, right);
```

```
        // Merge the sorted halves
```

```
        merge(arr, left, mid, right);
```

```
    }
```

```
}
```

```

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {4,2,0,1,3};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}

```

### **Output:**

```

C:\Users\Suresh Dahal\dsa>a.exe
Original array: 4 2 0 1 3
Sorted array: 0 1 2 3 4

```

## **Quick Sort**

### **Algorithm:**

**Step 1:** Choose a "pivot" element from the array. Different strategies can be used to choose the pivot (e.g., first element, last element, or middle element).

**Step 2:** Partition the array into two subarrays:

- Left subarray contains elements less than the pivot.
- Right subarray contains elements greater than the pivot.

**Step 3:** Recursively apply the quick sort to the left and right subarrays.

**Step 4:** Repeat steps 1-3 until the entire array is sorted.

**Example:**

**Input Array:** [4, 2, 0, 1, 3]

**Step 1: Choose a pivot.**

Let's choose the last element (3) as the pivot.

**Step 2: Partition the array into:**

- Left subarray: [2, 0, 1] (all elements less than 3)
- Right subarray: [4] (all elements greater than 3)

**Step 3: Apply Quick Sort to [2, 0, 1]:**

- Choose 1 as the pivot.
- Partition into: [0] (less than 1) and [2] (greater than 1).

**Step 4: Combine sorted subarrays:**

- Merge [0, 1, 2] with the pivot 3, followed by [4] → [0, 1, 2, 3, 4].

**Output:** [0, 1, 2, 3, 4]

**Program:**

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

```
// Function to partition the array
```

```
int partition(int arr[], int low, int high) {
```

```
    int pivot = arr[high]; // Choose the last element as the pivot
```

```
    int i = (low - 1); // Index of smaller element
```

```
    for (int j = low; j < high; j++) {
```

```
        if (arr[j] < pivot) { // If current element is smaller than the pivot
```

```
            i++;
```

```
            // Swap arr[i] and arr[j]
```

```
            int temp = arr[i];
```



```

        arr[i] = arr[j];
        arr[j] = temp;
    }
}

// Swap the pivot element with arr[i + 1] to place it in the correct position
int temp = arr[i + 1];
arr[i + 1] = arr[high];
arr[high] = temp;

return (i + 1); // Return the pivot index
}

// Function to perform quick sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Partitioning index

        // Recursively sort the two subarrays
        quickSort(arr, low, pi - 1); // Left of pivot
        quickSort(arr, pi + 1, high); // Right of pivot
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

```

```

int main() {
    int arr[] = {4,2,0,1,3};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

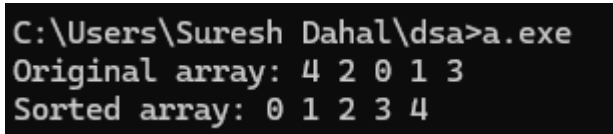
    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}

```

### **Output:**



```

C:\Users\Suresh Dahal\dsa>a.exe
Original array: 4 2 0 1 3
Sorted array: 0 1 2 3 4

```

## **7. Write programs to implement Binary and Sequential Search.**

### **Binary Search**

#### **Algorithm:**

**Step 1:** Start with the middle element of the array.

**Step 2:** Compare the middle element with the target value.

- If the target is equal to the middle element, the search is complete, and the index of the middle element is returned.
- If the target is less than the middle element, search the left half of the array.
- If the target is greater than the middle element, search the right half of the array.

**Step 3:** Repeat the above steps until the element is found or the search interval is empty.

**Example:**

**Input Array:** [0, 1, 2, 3, 4]

**Target:** 3

- Start by comparing the middle element (2) with the target (3).
  - The target is greater than 2, so search the right half of the array: [3, 4].
- Next, compare the middle element (3) with the target (3).
  - The target is found at index 3.

**Output:** Index 3

**Program:**

```
#include <stdio.h>

// Function to implement binary search
int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        // Check if the target is present at mid
        if (arr[mid] == target) {
            return mid; // Target found, return the index
        }

        // If target is smaller than mid, it is in the left subarray
        if (arr[mid] > target) {
            right = mid - 1;
        }
    }
}
```

```
// If target is larger than mid, it is in the right subarray
else {
    left = mid + 1;
}
}

return -1; // Target not found
}
```

```
// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
int main() {
    int arr[] = {0,1,2,3,4};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target;

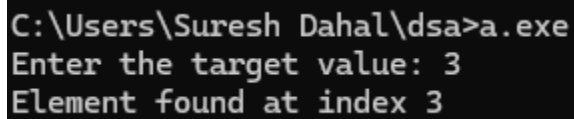
    printf("Enter the target value: ");
    scanf("%d", & target);

    int result = binarySearch(arr, size, target);

    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
```

```
printf("Element not found in the array\n");  
}  
return 0;  
}
```

### Output:



```
C:\Users\Suresh Dahal\dsa>a.exe  
Enter the target value: 3  
Element found at index 3
```

## Sequential Searching

### Algorithm:

**Step 1:** Start from the first element of the array.

**Step 2:** Compare the current element with the target value.

- If the current element matches the target, return the index of that element.
- If the current element does not match, move to the next element.

**Step 3:** Repeat steps 1-2 for all elements in the array until the target is found or the end of the array is reached.

**Step 4:** If the target is not found by the end of the array, return -1.

### Example:

**Input Array:** [4, 2, 0, 1, 3]

**Target:** 3

- Start by comparing the first element (4) with the target (3).
  - The target is not 4, move to the next element (2).
  - The target is not 2, move to the next element (0).
  - The target is not 0, move to the next element (1).
  - The target is not 1, move to the next element (3).
  - The target is 3, found at index 4.

**Output:** Index 4

**Program:**

```
#include <stdio.h>

// Function to implement sequential search
int sequentialSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index where target is found
        }
    }
    return -1; // Target not found
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {4,2,0,1,3};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target;

    printf("Enter the target value: ");
    scanf("%d", & target);

    int result = sequentialSearch(arr, size, target);
```

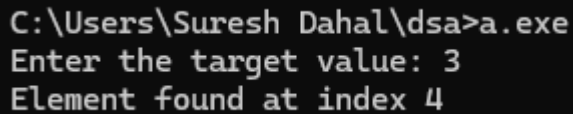
```

if (result != -1) {
    printf("Element found at index %d\n", result);
} else {
    printf("Element not found in the array\n");
}

return 0;
}

```

**Output:**



```

C:\Users\Suresh Dahan\dsa>a.exe
Enter the target value: 3
Element found at index 4

```

## 8. Write programs to implement search, spanning tree and shortest path algorithm in graph

### Searching: Depth First Search (DFS) and Breadth First Search (BFS)

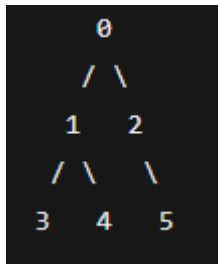
**BFS Algorithm:**

1. **Step 1:** Initialize a queue and enqueue the root node.
2. **Step 2:** While the queue is not empty:
  - **Step 2a:** Dequeue a node and visit it.
  - **Step 2b:** Enqueue its left child (if exists).
  - **Step 2c:** Enqueue its right child (if exists).

**DFS Algorithm:**

1. **Step 1:** Visit the current node.
2. **Step 2:** Recursively traverse the left subtree.
3. **Step 3:** Recursively traverse the right subtree.

### Example:



**BFS Output:** [0, 1, 2, 3, 4, 5]

**DFS Output:** [0, 1, 3, 4, 2, 5]

### Program:

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

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

```
// Tree Node structure
```

```
typedef struct Node {  
    int data;  
    struct Node *left, *right;  
} Node;
```

```
// Queue structure for BFS
```

```
typedef struct Queue {  
    Node **array;  
    int front, rear, size;  
} Queue;
```

```
// Create a new node
```

```
Node* createNode(int data) {  
    Node* newNode = (Node*)malloc(sizeof(Node));  
    newNode->data = data;  
    newNode->left = newNode->right = NULL;  
    return newNode;  
}
```



// Initialize queue

```
Queue* createQueue(int size) {  
    Queue* q = (Queue*)malloc(sizeof(Queue));  
    q->array = (Node**)malloc(size * sizeof(Node*));  
    q->front = q->rear = -1;  
    q->size = size;  
    return q;  
}
```

// Enqueue a node

```
void enqueue(Queue* q, Node* node) {  
    if (q->rear == q->size - 1) return;  
    q->array[++q->rear] = node;  
    if (q->front == -1) q->front = 0;  
}
```

// Dequeue a node

```
Node* dequeue(Queue* q) {  
    if (q->front == -1) return NULL;  
    Node* temp = q->array[q->front];  
    if (q->front == q->rear) q->front = q->rear = -1;  
    else q->front++;  
    return temp;  
}
```

// BFS Traversal

```
void BFS(Node* root) {  
    if (root == NULL) return;  
    Queue* q = createQueue(15); // Assuming max 15 nodes  
    enqueue(q, root);
```

```

while (q->front != -1) {
    Node* current = dequeue(q);
    printf("%d ", current->data);

    if (current->left != NULL) enqueue(q, current->left);
    if (current->right != NULL) enqueue(q, current->right);
}
free(q->array);
free(q);
}

```

// DFS Traversal (Recursive)

```

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

```

// Build the tree (as per the example)

```

Node* buildTree() {
    Node* root = createNode(0);
    root->left = createNode(1);
    root->right = createNode(2);
    root->left->left = createNode(3);
    root->left->right = createNode(4);
    root->right->right = createNode(5);
    return root;
}

```

```
int main() {  
    Node* root = buildTree();  
    printf("BFS Traversal: ");  
    BFS(root);  
    printf("\nDFS Traversal: ");  
    DFS(root);  
    return 0;  
}
```

**Output:**

```
C:\Users\Suresh Dahal\dsa>a.exe  
BFS Traversal: 0 1 2 3 4 5  
DFS Traversal: 0 1 3 4 2 5
```

## **Minimum Spanning Tree: Prim's and Kruskal Algorithms**

### **A. Prim's Algorithm:**

**Step 1:** Initialize a key array to store minimum weights and a boolean array to track included vertices.

**Step 2:** Start from an arbitrary vertex (e.g., vertex 0). Set its key to 0.

**Step 3:** For V-1 iterations:

Pick the vertex with the minimum key not yet included in the MST.

Update keys of adjacent vertices if a smaller edge weight is found.

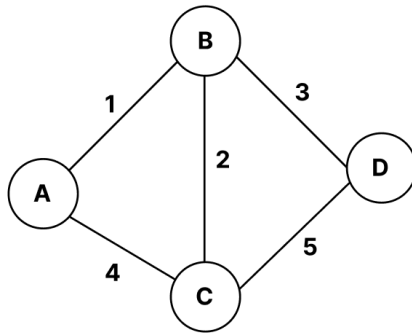
### **B. Kruskal's Algorithm:**

**Step 1:** Sort all edges in ascending order of weight.

**Step 2:** Use Union-Find to add edges to the MST, ensuring no cycles.

**Step 3:** Stop when V-1 edges are added.

**Example:**



**Kruskal's MST:** A-B (1), B-C (2), B-D (3) with total weight 6

**Prim's MST:** A-B (1), B-C (2), B-D (3) with total weight 6

**Program:**

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

#define V 4 // Number of vertices
#define INF INT_MAX

// ----- Prim's Algorithm -----

void primMST(int graph[V][V]) {
    int parent[V], key[V], mstSet[V];

    // Initialize all key values to INF and mstSet[] to false
    for (int i = 0; i < V; i++) {
        key[i] = INF;
        mstSet[i] = 0;
    }

    key[0] = 0; // Start from vertex A (0th index)
```

```
parent[0] = -1;
```

```
// Find the MST (V - 1) edges
```

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

```
    int u, min = INF;
```

```
    // Find the vertex with the minimum key value that is not yet included in MST
```

```
    for (int v = 0; v < V; v++)
```

```
        if (!mstSet[v] && key[v] < min)
```

```
            min = key[v], u = v;
```

```
    mstSet[u] = 1; // Include this vertex in MST
```

```
    // Update key values of the adjacent vertices of the selected vertex
```

```
    for (int v = 0; v < V; v++)
```

```
        if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])
```

```
            parent[v] = u, key[v] = graph[u][v];
```

```
}
```

```
// Print the MST edges
```

```
printf("Prim's MST Edges:\n");
```

```
for (int i = 1; i < V; i++)
```

```
    printf("%c - %c (Weight %d)\n", parent[i] + 'A', i + 'A', graph[i][parent[i]]);
```

```
}
```

```
// ----- Kruskal's Algorithm -----
```

```
struct Edge {
```

```
    int src, dest, weight;
```

```
};
```

```
struct subset {
```

```

    int parent, rank;
};

int find(struct subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else {
        subsets[yroot].parent = xroot;
        if (subsets[xroot].rank == subsets[yroot].rank)
            subsets[xroot].rank++;
    }
}

int compare(const void* a, const void* b) {
    return ((struct Edge*)a)->weight - ((struct Edge*)b)->weight;
}

void kruskalMST(struct Edge edges[], int E) {
    qsort(edges, E, sizeof(edges[0]), compare);

    struct subset* subsets = (struct subset*)malloc(V * sizeof(struct subset));
    for (int v = 0; v < V; v++) {

```

```

    subsets[v].parent = v;
    subsets[v].rank = 0;
}

struct Edge result[V];
int e = 0, i = 0;

while (e < V - 1 && i < E) {
    struct Edge next = edges[i++];
    int x = find(subsets, next.src);
    int y = find(subsets, next.dest);

    if (x != y) {
        result[e++] = next;
        Union(subsets, x, y);
    }
}

// Print the MST edges
printf("\nKruskal's MST Edges:\n");
for (i = 0; i < e; i++)
    printf("%c - %c (Weight %d)\n", result[i].src + 'A', result[i].dest + 'A', result[i].weight);
}

int main() {
    // Adjacency matrix for Prim's Algorithm
    int graph[V][V] = {
        {0, 1, 4, INF}, // A
        {1, 0, 2, 3},   // B
        {4, 2, 0, 5},   // C
        {INF, 3, 5, 0}  // D
    }

```

```

};

// Edge list for Kruskal's Algorithm
struct Edge edges[] = {
    {0, 1, 1}, {0, 2, 4}, {1, 2, 2}, {2, 3, 5}, {1, 3, 3}
};

int E = sizeof(edges) / sizeof(edges[0]);

// Calling Prim's and Kruskal's MST functions
primMST(graph);
kruskalMST(edges, E);

return 0;
}

```

### Output:

```

C:\Users\Suresh Dahal\dsa>gcc kruskal-and-prim.cpp

C:\Users\Suresh Dahal\dsa>a.exe
Prim's MST Edges:
A - B (Weight 1)
B - C (Weight 2)
B - D (Weight 3)

Kruskal's MST Edges:
A - B (Weight 1)
B - C (Weight 2)
B - D (Weight 3)

```

## Shortest Path Algorithm: Dijkstra's Algorithm

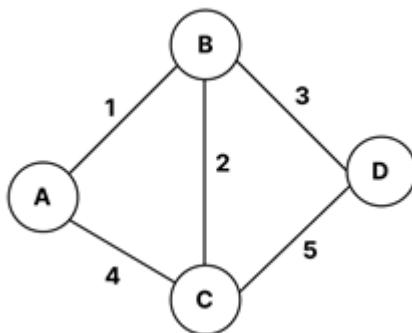
### Algorithm:

1. **Step 1:** Initialize a distance array `dist[]` where `dist[i]` represents the shortest distance from the source to vertex `i`. Set `dist[source] = 0` and all others to infinity (INF).
2. **Step 2:** Use a priority queue (min-heap) to track unvisited nodes, starting with the source.



3. **Step 3:** While the queue is not empty:
  - Extract the node  $u$  with the smallest tentative distance.
  - For each adjacent node  $v$ , if  $\text{dist}[u] + \text{weight}(u, v) < \text{dist}[v]$ , update  $\text{dist}[v]$  and add  $v$  to the queue.
4. **Step 4:** Repeat until all nodes are visited.

**Example:**



**Shortest Paths from Source A (vertex 0):**

- A to A: 0
- A to B:  $A \rightarrow B$  (Distance = 1)
- A to C:  $A \rightarrow B \rightarrow C$  (Distance = 3)
- A to D:  $A \rightarrow B \rightarrow D$  (Distance = 4)

**Program:**

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

```
#include <limits.h>
```

```
#define V 4
```

```
// Function to find the vertex with the minimum distance value
```

```
int minDistance(int dist[], int sptSet[]) {
```

```
    int min = INT_MAX, min_index;
```

```

for (int v = 0; v < V; v++) {
    if (sptSet[v] == 0 && dist[v] <= min) {
        min = dist[v];
        min_index = v;
    }
}
return min_index;
}

```

// Function to implement Dijkstra's algorithm

```

void dijkstra(int graph[V][V], int src) {
    int dist[V]; // The output array dist[i] holds the shortest distance from src to i
    int sptSet[V]; // sptSet[i] will be 1 if vertex i is included in the shortest path tree

```

// Initialize all distances as INFINITE and sptSet[] as 0

```

for (int i = 0; i < V; i++) {
    dist[i] = INT_MAX;
    sptSet[i] = 0;
}

```

// Distance from source to itself is always 0

```

dist[src] = 0;

```

// Find the 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
    int u = minDistance(dist, sptSet);
    sptSet[u] = 1; // Mark the picked vertex as processed

```

// Update dist[] values for the adjacent vertices of the picked vertex

```

    for (int v = 0; v < V; v++) {
        // Update dist[v] if and only if the current vertex is not in sptSet,
        // there is an edge from u to v, and the total distance through u is smaller than the
        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 calculated shortest distances
printf("Shortest Paths from Source A (vertex 0):\n");
for (int i = 0; i < V; i++) {
    printf("• A to ");
    if (i == 0) printf("A: 0\n");
    else if (i == 1) printf("B: A → B (Distance = %d)\n", dist[i]);
    else if (i == 2) printf("C: A → B → C (Distance = %d)\n", dist[i]);
    else if (i == 3) printf("D: A → B → D (Distance = %d)\n", dist[i]);
}

}

int main() {
    // Graph representation (adjacency matrix)
    int graph[V][V] = {
        {0, 1, 4, 0},
        {1, 0, 2, 3},
        {4, 2, 0, 5},
        {0, 3, 5, 0}
    };

```

```
// Run Dijkstra's algorithm for source vertex 0 (A)
dijkstra(graph, 0);

return 0;
}
```

**Output:**

```
C:\Users\Suresh Dahal\dsa>a.exe
Shortest Paths from Source A (vertex 0):
A to A: 0
A to B: A -> B (Distance = 1)
A to C: A -> B -> C (Distance = 3)
A to D: A -> B -> D (Distance = 4)
```