# Binary Search Tree

Node Structure

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

    Node(int val) {
        key = val;
        left = right = nullptr;
    }
};
```

Insertion

```
Node* insert(Node* root, int key) {
    if (root == nullptr)
        return new Node(key);

    if (key < root->key)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    return root;
}
```

## Search

```
Node* search(Node* root, int key) {
    if (root == nullptr || root->key == key)
        return root;

    if (key < root->key)
        return search(root->left, key);
    else
        return search(root->right, key);
}
```

## Deletion

```
Node* findMin(Node* node) {
    while (node->left != nullptr)
        node = node->left;
    return node;
}

Node* deleteNode(Node* root, int key) {
    if (root == nullptr)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
```

```cpp
    else {
        // Node with only one child or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // Node with two children
        Node* temp = findMin(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

| Operation | Best/Average | Worst (Unbalanced Tree) |
| --- | --- | --- |
| Insert | O(log n) | O(n) |
| Search | O(log n) | O(n) |
| Delete | O(log n) | O(n) |

## Minimum in BST

```
Node* findMin(Node* root) {
    if (root == nullptr)
        return nullptr;

    while (root->left != nullptr)
        root = root->left;

    return root;
}
```

## Maximum in BST

```
Node* findMax(Node* root) {
    if (root == nullptr)
        return nullptr;

    while (root->right != nullptr)
        root = root->right;

    return root;
}
```

**Time Complexity**

- **Best Case**: O(1) (if root is the minimum/maximum)
- **Worst Case**: O(h)
  where h = height of the tree
  O(log n) for balanced BST
  O(n) for skewed BST

## In-order Successor

```cpp
Node* findMin(Node* root) {
    while (root && root->left != nullptr)
        root = root->left;
    return root;
}

Node* findSuccessor(Node* root, Node* target) {
    if (target->right != nullptr)
        return findMin(target->right);

    Node* successor = nullptr;
    while (root != nullptr) {
        if (target->key < root->key) {
            successor = root;
            root = root->left;
        } else if (target->key > root->key) {
            root = root->right;
        } else {
            break;
        }
    }
    return successor;
}
```

**Best / Average case**: O(log n) ← (Balanced BST)

**Worst case**: O(n) ← (Skewed BST)

## Predecessor

```
Node* findMax(Node* root) {
    while (root && root->right != nullptr)
        root = root->right;
    return root;
}

Node* findPredecessor(Node* root, Node* target) {
    if (target->left != nullptr)
        return findMax(target->left);

    Node* predecessor = nullptr;
    while (root != nullptr) {
        if (target->key > root->key) {
            predecessor = root;
            root = root->right;
        } else if (target->key < root->key) {
            root = root->left;
        } else {
            break;
        }
    }
    return predecessor;
}
```

**Best/Average Case**: O(log n) (Balanced BST)

**Worst Case**: O(n) (Skewed tree)

Search Target in B-Tree Node

```
FUNCTION contains(target)
    SET i = 0
    WHILE (i < dataCount AND data[i] < target) DO
        i = i + 1
    END WHILE

    IF (i < dataCount AND data[i] == target) THEN
        RETURN true
    END IF

    IF (node is a leaf) THEN
        RETURN false
    END IF

    RETURN subset[i].contains(target)
 END FUNCTION
```

## Left Rotation Algorithm (Red-Black Tree)

```
SET y = x.right;

SET x.right = y.left;

IF y.left != NIL THEN
    SET y.left.p = x;
END IF;

SET y.p = x.p;

IF x.p == NIL THEN
    SET root = y;
ELSE IF x == x.p.left THEN
    SET x.p.left = y;
 ELSE
    SET x.p.right = y;
 END IF;

 SET y.left = x;

 SET x.p = y;
```

## Right Rotation Algorithm (Red-Black Tree)

```
SET x = y.left;

SET y.left = x.right;

IF x.right != NIL THEN
    SET x.right.p = y;
END IF;

SET x.p = y.p;

IF y.p == NIL THEN
    SET root = x;
ELSE IF y == y.p.right THEN
    SET y.p.right = x;
ELSE
    SET y.p.left = x;
END IF;

SET x.right = y;

SET y.p = x;
```

# Graph

## BFS

```
BFS(G, s)
BEGIN
  FOR EACH vertex u IN G.V - {s} LOOP
    u.color = WHITE;
    u.d = ∞;
    u.π = NIL;
  END LOOP;

  s.color = GRAY;
  s.d = 0;
  s.π = NIL;

  Q = ∅;
  ENQUEUE(Q, s);

  WHILE Q ≠ ∅ LOOP
    u = DEQUEUE(Q);

    FOR EACH v IN G.Adj[u] LOOP
      IF v.color = WHITE THEN
        v.color = GRAY;
        v.d = u.d + 1;
        v.π = u;
        ENQUEUE(Q, v);
      END IF;
    END LOOP;

    u.color = BLACK;
  END LOOP;
END;
```

## Kruskal’s Algorithm for Minimum Spanning Tree (MST)

```
Kruskal(G)
    MST = Ø
    for each vertex v in G.V
        MAKE-SET(v)
    sort the edges of G.E in non-decreasing order by weight
    for each edge (u, v) in sorted edges
        if FIND-SET(u) ≠ FIND-SET(v)
            MST = MST ∪ {(u, v)}
            UNION(u, v)
    return MST
```

## Prim’s Algorithm for Minimum Spanning Tree (MST)

```
Prim(G, start)
    for each vertex v in G.V
        v.key = ∞
        v.π = NIL
    start.key = 0
    Q = all vertices in G.V

    while Q is not empty
        u = EXTRACT-MIN(Q)
        for each vertex v adjacent to u
            if v ∈ Q and weight(u,v) < v.key
                v.π = u
                v.key = weight(u,v)
```

Dijkstra’s Algorithm

```
DIJKSTRA(G, w, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
Q = G.V         // Initialize priority queue with all vertices

while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each vertex v ∈ Adj[u]
        if v.d > u.d + w(u, v)
            v.d = u.d + w(u, v)
            v.π = u
```

## Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)

for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL

s.d = 0

for i = 1 to |V| - 1
    for each edge (u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            v.d = u.d + w(u, v)
            v.π = u

for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return "Negative weight cycle detected"

return "Shortest paths found"
```