

SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur – 603 203

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

**M.E CSE
I SEMESTER**



CP3165 – ADAVANCED DATASTRUCTURES AND ALGORITHMS LABORATORY

Regulation – 2023

Academic Year 2026 – 2027 (ODD SEMESTER)

Prepared by

Ms. S. PRAMILA /AP- SEL.G

EX. NO: 1 IMPLEMENTATION OF A BINARY SEARCH TREE (BST)
DATE:

AIM:

To implement a Binary Search Tree (BST) in Python with insertion, traversal, and searching operations.

ALGORITHM:

- Start the process.
- Initialize and declare variables.
- Construct the Tree
- Data values are given which we call a key and a binary search tree
- To search for the key in the given binary search tree, start with the root node and compare the key with the data value of the root node. If they match, return the root pointer.
- If the key is less than the data value of the root node, repeat the process by using the left subtree.
- Otherwise, repeat the same process with the right subtree until either a match is found or the subtree under consideration becomes an empty tree.
- Terminate

PROGRAM:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)
        return root

    def inorder(self, root):
```

```

        if root:
            self.inorder(root.left)
            print(root.key, end=" ")
            self.inorder(root.right)

    def search(self, root, key):
        if root is None or root.key == key:
            return root
        if key < root.key:
            return self.search(root.left, key)
        return self.search(root.right, key)

bst = BST()
root = None
data = [50, 30, 20, 40, 70, 60, 80]
for val in data:
    root = bst.insert(root, val)

print("Inorder Traversal:", end=" ")
bst.inorder(root)

search_key = 40
print("\nSearching for", search_key)
if bst.search(root, search_key):
    print("Key Found")
else:
    print("Key Not Found")

```

OUTPUT:

```

Inorder Traversal: 20 30 40 50 60 70 80
Searching for 40
Key Found

```

RESULT:

The Binary Search Tree was successfully implemented with insertion, traversal, and search operations.

EX. NO: 2
DATE:

IMPLEMENTATION OF AVL TREE

AIM:

To implement an AVL Tree with rotations for balancing.

ALGORITHM:

1. Start the process.
2. Initialize an AVL tree with a Node class containing key, left, right, and height attributes.
3. Insert keys into the AVL tree by recursively comparing the key with the root's key.
4. Update the height of each node after insertion and calculate the balance factor.
5. 5: Perform rotations (left, right, left-right, or right-left) if the balance factor exceeds |1| to maintain AVL tree properties.
6. Traverse the tree in-order to print keys in ascending order.
7. Terminate.

PROGRAM:

```
class Node:
```

```
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
```

```
class AVLTree:
```

```
    def height(self, root):
        return root.height if root else 0

    def getBalance(self, root):
        return self.height(root.left) - self.height(root.right) if root else 0

    def rightRotate(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self.height(y.left), self.height(y.right))
        x.height = 1 + max(self.height(x.left), self.height(x.right))
        return x

    def leftRotate(self, x):
        y = x.right
```

```

    T2 = y.left
    y.left = x
    x.right = T2
    x.height = 1 + max(self.height(x.left), self.height(x.right))
    y.height = 1 + max(self.height(y.left), self.height(y.right))
    return y
def insert(self, root, key):
    if not root:
        return Node(key)
    elif key < root.key:
        root.left = self.insert(root.left, key)
    else:
        root.right = self.insert(root.right, key)

    root.height = 1 + max(self.height(root.left), self.height(root.right))
    balance = self.getBalance(root)

    if balance > 1 and key < root.left.key:
        return self.rightRotate(root)
    if balance < -1 and key > root.right.key:
        return self.leftRotate(root)
    if balance > 1 and key > root.left.key:
        root.left = self.leftRotate(root.left)
        return self.rightRotate(root)
    if balance < -1 and key < root.right.key:
        root.right = self.rightRotate(root.right)
        return self.leftRotate(root)
    return root

def inorder(self, root):
    if root:
        self.inorder(root.left)
        print(root.key, end=" ")
        self.inorder(root.right)

avl = AVLTree()
root = None
for val in [10, 20, 30, 40, 50, 25]:
    root = avl.insert(root, val)
print("Inorder Traversal:", end=" ")
avl.inorder(root)

```

OUTPUT:

Inorder Traversal: 10 20 25 30 40 50

RESULT:

The AVL Tree was successfully implemented with insertion and balancing using rotations.

EX. NO: 3
DATE:

IMPLEMENTATION OF B TREE

AIM:

To implement a B-Tree of order t with insertion.

ALGORITHM:

- Start the process.
- Initialize a B-Tree with minimum degree t (here $t=3$), creating an empty root node that is a leaf.
- For each key in the insertion sequence [10, 20, 5, 6, 12, 30, 7, 17], insert the key into the B-Tree.
- If the root is full (contains $2t-1$ keys), create a new root, move the old root as its first child, split the full child, and insert into a non-full node.
- To insert into a non-full node, find the appropriate child by comparing the key with existing keys; if the child is full, split it first, then recurse into the correct (now non-full) child.
- When a full child node y is split at index i in parent node x , create a new node z , move the upper half of keys and children from y to z , insert the median key into x , and update child pointers.
- Perform an in-order traversal of the final B-Tree, visiting children before printing keys and the last child after the keys, to display all keys in sorted order.
- Terminate.

PROGRAM:

```
class BTreeNode:
    def __init__(self, t, leaf=False):
        self.t = t
        self.keys = []
        self.children = []
        self.leaf = leaf

class BTree:
    def __init__(self, t):
        self.root = BTreeNode(t, True)
        self.t = t

    def traverse(self, node=None):
        if node is None:
            node = self.root
        for i in range(len(node.keys)):
            if not node.leaf:
```

```

        self.traverse(node.children[i])
    print(node.keys[i], end=" ")
if not node.leaf:
    self.traverse(node.children[-1])

def insert(self, k):
    root = self.root
    if len(root.keys) == (2*self.t - 1):
        temp = BTreeNode(self.t, False)
        temp.children.insert(0, root)
        self.splitChild(temp, 0)
        i = 0
        if temp.keys[0] < k:
            i += 1
        self.insertNonFull(temp.children[i], k)
        self.root = temp
    else:
        self.insertNonFull(root, k)

def insertNonFull(self, node, k):
    i = len(node.keys) - 1
    if node.leaf:
        node.keys.append(0)
        while i >= 0 and k < node.keys[i]:
            node.keys[i+1] = node.keys[i]
            i -= 1
        node.keys[i+1] = k
    else:
        while i >= 0 and k < node.keys[i]:
            i -= 1
        i += 1
        if len(node.children[i].keys) == (2*self.t - 1):
            self.splitChild(node, i)
            if k > node.keys[i]:
                i += 1
        self.insertNonFull(node.children[i], k)

def splitChild(self, node, i):
    t = self.t
    y = node.children[i]
    z = BTreeNode(t, y.leaf)
    node.children.insert(i+1, z)
    node.keys.insert(i, y.keys[t-1])

```

```
z.keys = y.keys[t:(2*t-1)]
y.keys = y.keys[0:t-1]
if not y.leaf:
    z.children = y.children[t:(2*t)]
    y.children = y.children[0:t]

b = BTree(3)
for val in [10, 20, 5, 6, 12, 30, 7, 17]:
    b.insert(val)

print("B-Tree traversal:", end=" ")
b.traverse()
```

OUTPUT:

B-Tree traversal: 5 6 7 10 12 17 20 30

RESULT:

The B-Tree was successfully implemented with insertion and traversal operations.

EX. NO: 4
DATE:

RED-BLACK TREE IMPLEMENTATION

AIM:

To implement a Red-Black Tree in Python with insertion and balancing.

ALGORITHM:

- Start the process.
- Initialize a Red-Black Tree with a sentinel node TNULL (black) and set the root to TNULL.
- For each key in the insertion sequence [10, 20, 30, 15, 25, 5], insert the key using standard BST insertion, marking the new node as RED.
- If the inserted node is the root, color it BLACK and terminate fixing for that node.
- Otherwise, call fix_insert to restore Red-Black properties whenever a RED parent–RED child violation occurs.
- In fix_insert, while the parent is RED: identify the uncle node; if uncle is RED, recolor parent, uncle (to BLACK), and grandparent (to RED), then move up to grandparent; if uncle is BLACK, perform appropriate rotations (left/right or double left-right/right-left) and recolor to resolve the violation.
- After fixing, ensure the root is always BLACK; perform in-order traversal printing each key with its color to display the final balanced Red-Black Tree in sorted order.
- Terminate.

PROGRAM:

```
import sys

class Node:
    def __init__(self, key):
        self.key = key
        self.color = "RED"
        self.parent = None
        self.left = None
        self.right = None

class RedBlackTree:
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = "BLACK"
        self.root = self.TNULL

    def inorder(self, node):
        if node != self.TNULL:
```

```

        self.inorder(node.left)
        print(f"{{node.key}}({{node.color}})", end=" ")
        self.inorder(node.right)

def insert(self, key):
    node = Node(key)
    node.left = self.TNULL
    node.right = self.TNULL
    parent = None
    temp = self.root

    while temp != self.TNULL:
        parent = temp
        if node.key < temp.key:
            temp = temp.left
        else:
            temp = temp.right

    node.parent = parent
    if parent is None:
        self.root = node
    elif node.key < parent.key:
        parent.left = node
    else:
        parent.right = node

    if node.parent is None:
        node.color = "BLACK"
        return

    if node.parent.parent is None:
        return

    self.fix_insert(node)

def left_rotate(self, x):
    y = x.right
    x.right = y.left
    if y.left != self.TNULL:
        y.left.parent = x
    y.parent = x.parent
    if x.parent is None:
        self.root = y

```

```
elif x == x.parent.left:
    x.parent.left = y
else:
    x.parent.right = y
y.left = x
x.parent = y
```

```
def right_rotate(self, x):
    y = x.left
    x.left = y.right
    if y.right != self.TNULL:
        y.right.parent = x
    y.parent = x.parent
    if x.parent is None:
        self.root = y
    elif x == x.parent.right:
        x.parent.right = y
    else:
        x.parent.left = y
    y.right = x
    x.parent = y
```

```
def fix_insert(self, k):
    while k.parent.color == "RED":
        if k.parent == k.parent.parent.right:
            u = k.parent.parent.left
            if u.color == "RED":
                u.color = "BLACK"
                k.parent.color = "BLACK"
                k.parent.parent.color = "RED"
                k = k.parent.parent
            else:
                if k == k.parent.left:
                    k = k.parent
                    self.right_rotate(k)
                k.parent.color = "BLACK"
                k.parent.parent.color = "RED"
                self.left_rotate(k.parent.parent)
        else:
            u = k.parent.parent.right
            if u.color == "RED":
                u.color = "BLACK"
                k.parent.color = "BLACK"
```

```

        k.parent.parent.color = "RED"
        k = k.parent.parent
    else:
        if k == k.parent.right:
            k = k.parent
            self.left_rotate(k)
            k.parent.color = "BLACK"
            k.parent.parent.color = "RED"
            self.right_rotate(k.parent.parent)
        if k == self.root:
            break
    self.root.color = "BLACK"

rbt = RedBlackTree()
for val in [10, 20, 30, 15, 25, 5]:
    rbtree.insert(val)

print("Inorder Traversal:", end=" ")
rbtree.inorder(rbt.root)

```

OUTPUT:

Inorder Traversal: 5(RED) 10(BLACK) 15(RED) 20(BLACK) 25(RED) 30(BLACK)

RESULT:

The Red-Black Tree was successfully implemented with insertion and balancing operations.

EX. NO: 5
DATE:

MIN MAX HEAP IMPLEMENTATION

AIM:

To implement a Min-Max Heap in Python.

ALGORITHM:

- Start the process.
- Initialize a MinMaxHeap using a Python list data managed as a min-heap with the heapq module.
- For each value in the insertion sequence [10, 40, 5, 30, 25], insert the value into the heap using heapq.heappush to maintain the min-heap property.
- To retrieve the minimum element, return the first element of the list (data[0]), which is always the smallest due to the min-heap structure.
- To retrieve the maximum element, scan the entire list and return the largest value using max(data).
- To delete and return the minimum element, use heapq.heappop which removes and returns the smallest element while preserving the min-heap property.
- To delete and return the maximum element, find the max value, remove it from the list using list.remove, then call heapq.heapify to restore the min-heap property on the modified list.
- Print the current minimum, current maximum, the deleted minimum, and the deleted maximum after performing the operations.
- Terminate.

PROGRAM:

```
import heapq

class MinMaxHeap:
    def __init__(self):
        self.data = []

    def insert(self, val):
        heapq.heappush(self.data, val)

    def get_min(self):
        return self.data[0]

    def get_max(self):
        return max(self.data)

    def delete_min(self):
```

```
        return heapq.heappop(self.data)

    def delete_max(self):
        m = max(self.data)
        self.data.remove(m)
        heapq.heapify(self.data)
        return m

h = MinMaxHeap()
for val in [10, 40, 5, 30, 25]:
    h.insert(val)

print("Min:", h.get_min())
print("Max:", h.get_max())
print("Delete Min:", h.delete_min())
print("Delete Max:", h.delete_max())
```

OUTPUT:

```
Min: 5
Max: 40
Delete Min: 5
Delete Max: 40
```

RESULT:

The Min-Max Heap was successfully implemented with insertion, retrieval, and deletion of minimum and maximum elements.

EX. NO: 6
DATE:

LEFTIST HEAP IMPLEMENTATION

AIM:

To implement a Leftist Heap with merge operation.

ALGORITHM:

- Start the process.
- Define a Node with key, left, right children, and a distance (dist) field to track the right-spine length.
- For each value in the insertion sequence [10, 5, 30, 2, 8], insert the value by creating a new single-node heap and merging it with the current heap h.
- In the merge function, recursively combine two leftist heaps h1 and h2: if one is null, return the other; otherwise, ensure the smaller key becomes the root (swap if needed).
- Recursively merge the right subtree of the new root with the other heap, then attach the result to the root's right child.
- Maintain the leftist property: swap left and right children if the distance of the left child is less than that of the right child; update the root's dist as 1 plus the distance of its right child (0 if no right child).
- After all insertions, perform an in-order traversal of the resulting leftist heap (left subtree → root → right subtree) to print keys in sorted order, demonstrating the heap's BST property.
- Terminate.

PROGRAM:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.dist = 0

def merge(h1, h2):
    if h1 is None: return h2
    if h2 is None: return h1
    if h1.key > h2.key:
        h1, h2 = h2, h1
    h1.right = merge(h1.right, h2)
    if h1.left is None or (h1.left.dist < h1.right.dist):
        h1.left, h1.right = h1.right, h1.left
    h1.dist = 1 + (h1.right.dist if h1.right else 0)
    return h1
```

```
def insert(h, key):
    return merge(h, Node(key))

def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

h = None
for val in [10, 5, 30, 2, 8]:
    h = insert(h, val)

print("Inorder traversal:", end=" ")
inorder(h)
```

OUTPUT:

Inorder traversal: 2 5 8 10 30

RESULT:

The Leftist Heap was successfully implemented with insertion and merge operations.

EX. NO: 7
DATE:

FIBONACCI HEAP IMPLEMENTATION

AIM:

To implement a Fibonacci Heap in Python with insert and extract-min operations.

ALGORITHM:

- Start the process.
- Define a Fibonacci heap node (FibNode) with key, degree, parent, child, mark, and left/right pointers for a circular doubly-linked list.
- Initialize an empty Fibonacci heap (FibHeap) with min pointer set to None.
- For each value in the sequence [10, 3, 15, 6], create a new FibNode with the given value and insert it into the heap.
- During insertion, if the heap is empty, set the new node as the min pointer; otherwise, insert the node into the root list by updating circular doubly-linked list pointers (connect to min.right).
- After inserting into the root list, compare the new node's key with the current min key; if smaller, update the min pointer to the new node.
- After all insertions, retrieve the minimum element by returning the key of the node pointed to by min.
- Print the minimum value (which is 3 in this case).
- Terminate.

PROGRAM:

```
class FibNode:
    def __init__(self, key):
        self.key = key
        self.degree = 0
        self.parent = None
        self.child = None
        self.mark = False
        self.left = self
        self.right = self

class FibHeap:
    def __init__(self):
        self.min = None

    def insert(self, node):
        if self.min is None:
            self.min = node
        else:
```

```
node.left = self.min
node.right = self.min.right
self.min.right.left = node
self.min.right = node
if node.key < self.min.key:
    self.min = node
```

```
def get_min(self):
    return self.min.key if self.min else None
```

```
H = FibHeap()
for val in [10, 3, 15, 6]:
    H.insert(FibNode(val))

print("Minimum:", H.get_min())
```

OUTPUT:

Minimum: 3

RESULT:

The Fibonacci Heap was successfully implemented with insertion and minimum element retrieval operations.

EX. NO: 8
DATE:

SPANNING TREE IMPLEMENTATION

AIM:

To implement Prim's Algorithm for finding Minimum Spanning Tree (MST).

ALGORITHM:

- Start the process.
- Initialize an empty set visited to track vertices included in the MST, an empty list mst to store the resulting edges, and a min-heap edges containing the starting edge (weight 0, start vertex 0, parent None).
- While the min-heap edges is not empty, extract the edge with the smallest weight using heapq.heappop.
- If the extracted vertex v has not been visited, mark it as visited by adding it to the visited set.
- If the edge has a valid parent (not the starting vertex), add the edge (parent, v, weight) to the mst list.
- For each neighbor (nxt, w) of the newly visited vertex v in the graph, if nxt is not yet visited, push the edge (weight w, vertex nxt, parent v) into the min-heap edges.
- Repeat steps 3–6 until all reachable vertices are visited and the min-heap is empty, ensuring the minimum spanning tree is constructed greedily by always selecting the lowest-weight safe edge.
- Return and print the list mst containing the edges of the Minimum Spanning Tree (in this case: connections forming a tree with total minimum weight).
- Terminate.

PROGRAM:

```
import heapq

def prim(graph, start):
    visited = set()
    mst = []
    edges = [(0, start, None)]
    while edges:
        weight, v, parent = heapq.heappop(edges)
        if v not in visited:
            visited.add(v)
            if parent is not None:
                mst.append((parent, v, weight))
            for nxt, w in graph[v]:
                if nxt not in visited:
                    heapq.heappush(edges, (w, nxt, v))
```

```
    return mst
graph = {
    0: [(1, 2), (3, 6)],
    1: [(0, 2), (2, 3), (3, 8), (4, 5)],
    2: [(1, 3), (4, 7)],
    3: [(0, 6), (1, 8), (4, 9)],
    4: [(1, 5), (2, 7), (3, 9)]
}
mst = prim(graph, 0)
print("MST:", mst)
```

OUTPUT:

MST: [(0, 1, 2), (1, 2, 3), (1, 4, 5), (0, 3, 6)]

RESULT:

The Minimum Spanning Tree was successfully implemented using Prim's Algorithm.

EX. NO: 9
DATE:

SHORTEST PATH ALGORITHMS

AIM:

To implement Dijkstra's Algorithm to find shortest path in a graph.

ALGORITHM:

- Start the process.
- Initialize a dictionary dist with infinite distance for all vertices in the graph, except the start vertex (set to 0).
- Create a min-heap pq and push the tuple (distance 0, start vertex) into it.
- While the priority queue pq is not empty, extract the vertex v with the smallest distance d using heapq.heappop.
- If the extracted distance d is greater than the current known distance dist[v], skip this entry (it is outdated due to previous relaxation).
- For each neighbor (nxt, w) of vertex v, check if a shorter path exists via v: if dist[nxt] > dist[v] + w, update dist[nxt] to dist[v] + w and push the new (updated distance, nxt) into the priority queue.
- Repeat steps 4–6 until the priority queue is empty, ensuring each vertex is processed with its final minimum distance from the start.
- Return and print the dist dictionary containing the shortest path distances from the start vertex (0) to all reachable vertices in the graph.
- Terminate.

PROGRAM:

```
import heapq
```

```
def dijkstra(graph, start):
    dist = {v: float("inf") for v in graph}
    dist[start] = 0
    pq = [(0, start)]
    while pq:
        d, v = heapq.heappop(pq)
        if d > dist[v]:
            continue
        for nxt, w in graph[v]:
            if dist[nxt] > d + w:
                dist[nxt] = d + w
                heapq.heappush(pq, (dist[nxt], nxt))
    return dist
graph = {
    0: [(1, 4), (2, 1)],
```

```
1: [(3, 1)],  
2: [(1, 2), (3, 5)],  
3: []  
}  
print("Shortest Paths:", dijkstra(graph, 0))
```

OUTPUT:

Shortest Paths: {0: 0, 1: 3, 2: 1, 3: 4}

RESULT:

The shortest path in a weighted graph was successfully implemented using Dijkstra's Algorithm.

EX. NO: 10
DATE:

**IMPLEMENTATION OF TRAVELLING SALESMAN
PROBLEM SEGMENT TREES**

AIM:

To implement the Travelling Salesman Problem using Dynamic Programming (Held-Karp algorithm).

ALGORITHM:

- Start the process.
- Define a function tsp that takes a complete weighted graph (represented as an adjacency matrix) and computes the minimum cost of the Traveling Salesman Path starting and ending at city 0.
- Determine the number of cities n from the size of the graph.
- Use memoization with @lru_cache on a recursive helper function visit(mask, pos) where mask represents the set of visited cities (bitmask) and pos is the current city.
- Base case: If mask == (1 << n) - 1 (all cities visited), return the cost of returning from the current city pos to the starting city 0 (graph[pos][0]).
- For the current state (mask, pos), initialize answer ans to infinity; then for every unvisited city nxt (where (mask & (1 << nxt)) == 0), compute the cost of moving to nxt plus the minimum cost from nxt with updated mask:
 ans = min(ans, graph[pos][nxt] + visit(mask | (1 << nxt), nxt)).
- Cache and return the computed minimum cost for the current state (mask, pos).
- Initiate the recursion with mask = 1 (only city 0 visited) and pos = 0 (starting at city 0), which computes the minimum cost of visiting all cities exactly once and returning to city 0.
- Print the resulting minimum TSP tour cost (for the given graph: 80).
- Terminate.

PROGRAM:

```
from functools import lru_cache
```

```
def tsp(graph):
```

```
    n = len(graph)
```

```
    @lru_cache(None)
```

```
    def visit(mask, pos):
```

```
        if mask == (1 << n) - 1:
```

```
            return graph[pos][0]
```

```
        ans = float("inf")
```

```
        for nxt in range(n):
```

```
            if mask & (1 << nxt) == 0:
```

```
                ans = min(ans, graph[pos][nxt] + visit(mask | (1 << nxt), nxt))
```

```
        return ans
```

```
    return visit(1, 0)
graph = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
print("TSP Minimum Cost:", tsp(graph))
```

OUTPUT:

TSP Minimum Cost: 80

RESULT:

The Travelling Salesman Problem was successfully implemented using the Dynamic Programming (Held-Karp) approach.