Department of Decision and Computing Sciences

17MDCEL3 – Advanced Data Structures Laboratory

Record Work

Submitted by

GENGA K

2033012

Submitted to

Dr. A. Kannammal

Dr. N. Yamuna Devi

Dr. S. Chandia

Department of Computing

A picture containing text

Description automatically generated

COIMBATORE INSTITUTE OF TECHNOLOGY

(Government Aided Autonomous Institution)

Coimbatore – 641014.

**INDEX**

|  |  |  |
| --- | --- | --- |
| **Ex. No.** | **Date** | **Title** |
|  |  | **Binary Tree** |
| 1.1 | 11.12.2023 | **Binary Search Tree Operations**  I. Insert  II. Delete  III. Search  IV. Traversal |
|  |  | **Binary Tree Application** |
| 1.2 | 12.12.2023 | **Application in Binary Search Tree**  I. Insert  II. Delete  III. Search |
|  |  | **Binary Tree Special Operations** |
| 1.3 | 18.12.2023 | **Special Operations in BST Application – 1**  I. Minimum and maximum value node and its level  II. Find the third minimum node and its level  III. Find the sum of all numerical value on the left and right sub tree  IV. Find the distance between any two given nodes in terms of levels. |
| 1.4 | 26.12.2023 | **Special Operations in BST Application – 2**  I. Delete all the leaf nodes  II. Count the total no.of. Nodes  III. Fing the height of the tree  IV. Create and display the right threads of leaf node  V. Create and display the left threads of also leaf node |
|  |  | **AVL Tree** |
| 2 | 08.01.2024 | **AVL Tree Operations**  I. Insertion  II. Deletion  III. Inorder Traversal  IV. Counting total nodes  V. Searching for a key  VI. Checking if the key is empty |
|  |  | **Heap Operations** |
| 3.1 | 22.01.2024 | **Performing Heap operations**  I. Insertion  II. Deletion  III. Display the elements in array format  IV. Print the children of given node  V. Print the index of 1st leaf and last non-leaf nodes |
|  |  | **Application of Heap** |
| 3.2 | 24.01.2024 | **Application of Heap Tree**  I. Insertion  II. Deletion  III. Display the elements in array format  IV. Print the children of given node  V. Print the index of 1st leaf and last non-leaf nodes |

|  |  |
| --- | --- |
| **EXERCISE 1** | **Binary Search Tree Operations** |
| 11.12.2023 |

A Binary Search Tree (BST) is a binary tree data structure where each node has at most two child nodes, typically referred to as the left child and the right child. The key property of a BST is that the value of each node in the left subtree is less than or equal to the value of the node, and the value of each node in the right subtree is greater than or equal to the value of the node.

**AIM**

To implement insertion, deletion, searching and traversal operations in Binary Search Tree.

**CONCEPT IMPLEMENTED:**

* Insertion:
  + Adding a new node with a given key value into the BST while maintaining the BST property.
  + Compare the key value with the current node and traverse left or right until finding an appropriate spot to insert the new node.
* Deletion:
  + Removing a node with a specific key value from the BST. It handle different scenarios: a node has no children, a node has one child or a node has two children.
  + Reorganize the tree while maintaining the BST property.
* Search:
  + Finding a specific key value within the BST.
  + Start from the root node and compare the target key with the current node’s key. Move left or right in the tree based on the comparison until finding a match or reaching a leaf node.
* Traversal:
  + Visiting and processing all the nodes in the BST in a specific order.
  + Common traversal methods include Inorder (Left-Parent-Right), Preorder (Parent-Left-Right) and Postorder (Left-Right-Parent).

**APPLICATION**

1. Searching
2. Insertion and Deletion
3. Inorder Traversal for Sorted Output
4. Symbol Tables
5. File Systems

**ADVANTAGES**

1. Efficient Search Operations
2. Dynamic Data Sets
3. Sorted Data Output
4. Memory Efficiency
5. Simple Implementation

**CODE**

class Node:

def \_\_init\_\_(self, key):

self.key = key

self.left = None

self.right = None

def insert(root, key):

if root is None:

return Node(key)

else:

if key < root.key:

root.left = insert(root.left, key)

else:

root.right = insert(root.right, key)

return root

def inorder\_traversal(root):

if root:

inorder\_traversal(root.left)

print(root.key, end=" ")

inorder\_traversal(root.right)

def find\_min(node):

while node.left:

node = node.left

return node

def deletion(root, key):

if not root:

return None

elif key < root.key:

root.left = deletion(root.left, key)

elif key > root.key:

root.right = deletion(root.right, key)

return root

root = None

inputs = input("Enter a list of integers separated by spaces: ")

keys = list(map(int, inputs.split()))

for key in keys:

root = insert(root, key)

def is\_sorted(root):

if root is None:

return True

if root.left and root.left.key > root.key:

return False

if root.right and root.right.key < root.key:

return False

return is\_sorted(root.left) and is\_sorted(root.right)

print("Inorder traversal:")

inorder\_traversal(root)

choice = input('Enter choice: ')

print('\n1:Insertion \n2:Deletion')

if choice == '1':

new\_val = int(input('Enter the value to be inserted: '))

insert(root, new\_val)

print("Insertion successful!")

elif choice == '2':

del\_val = int(input('Enter the value to be deleted: '))

deletion(root, del\_val)

print("Deletion successful!")

else:

print("Invalid choice!")

sorted\_status = is\_sorted(root)

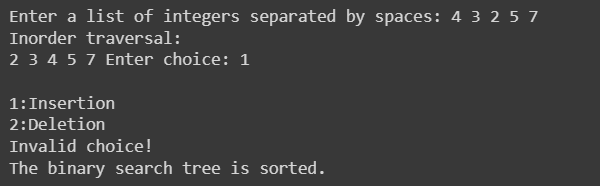
if sorted\_status:

print("The binary search tree is sorted.")

else:

print("The binary search tree is not sorted.")

**RESULT**

****

|  |  |
| --- | --- |
| **EXERCISE 2** | **Application in Binary Search Tree** |
| 12.12.2023 |

**AIM**

To implement insertion, deletion, searching and traversal operations in Binary Search Tree in BST Application.

**APPLICATION CHOSEN**

Inventory Date Sorting application to identify the expiring products

**CONCEPT**

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node’s value and the right child has a value greater than the node’s value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree. Binary Search Tree (BST) quickly allows us to maintain a sorted list of numbers. The key operations associated are:

* Insertion:
  + Adding a new node with a given key value into the BST while maintaining the BST property.
  + Compare the key value with the current node and traverse left or right until finding an appropriate spot to insert the new node.
* Deletion:
  + Removing a node with a specific key value from the BST. It handle different scenarios: a node has no children, a node has one child or a node has two children.
  + Reorganize the tree while maintaining the BST property.
* Search:
  + Finding a specific key value within the BST.
  + Start from the root node and compare the target key with the current node’s key. Move left or right in the tree based on the comparison until finding a match or reaching a leaf node.
* Traversal:
  + Visiting and processing all the nodes in the BST in a specific order.
  + Common traversal methods include Inorder (Left-Parent-Right), Preorder (Parent-Left-Right) and Postorder (Left-Right-Parent).

**APPLICATION**

* BSTs are used for indexing in databases.
* It is used to implement searching algorithms.
* BSTs are used to implement Huffman coding algorithm.
* It is also used to implement dictionaries.
* Used for data caching.

**ADVANTAGES**

* BST is fast in insertion and deletion when balanced. It is fast with a time complexity of O(log n).
* BST is also for fast searching, with a time complexity of O(log n) for most operations.
* We can also do range queries – find keys between N and M (N <= M).
* BST can automatically sort elements as they are inserted, so the elements are always stored in a sorted order.
* BST can be easily modified to store additional data or to support other operations. This makes it flexible.

**CODE**

import datetime

class Node:

def \_\_init\_\_(self, product\_id, name, expiration\_date):

self.product\_id = product\_id

self.name = name

self.expiration\_date = expiration\_date

self.left = None

self.right = None

def insert(root, product):

if root is None:

return Node(product.product\_id, product.name, product.expiration\_date)

if product.expiration\_date < root.expiration\_date:

root.left = insert(root.left, product)

else:

root.right = insert(root.right, product)

return root

def inorder\_traversal(root):

if root:

inorder\_traversal(root.left)

print(f"Product ID: {root.product\_id}, Name: {root.name}, Expiration Date: {root.expiration\_date}")

inorder\_traversal(root.right)

def find\_expiring\_products(root, days):

expiring\_products = []

def traverse(node, days):

if node:

traverse(node.left, days)

if (node.expiration\_date - datetime.date.today()).days <= days:

expiring\_products.append(node)

traverse(node.right, days)

traverse(root, days)

return expiring\_products

def get\_product\_details():

product\_id = int(input("Enter product ID: "))

name = input("Enter product name: ")

expiration\_date = input("Enter expiration date (YYYY-MM-DD): ")

return {"product\_id": product\_id, "name": name, "expiration\_date": datetime.datetime.strptime(expiration\_date, "%Y-%m-%d").date()}

products = []

while True:

choice = input("Add a product? (y/n): ")

if choice.lower() != "y":

break

product\_details = get\_product\_details()

products.append(product\_details)

root = None

for product in products:

product = Node(product["product\_id"], product["name"], product["expiration\_date"])

root = insert(root, product)

print("Inventory:")

inorder\_traversal(root)

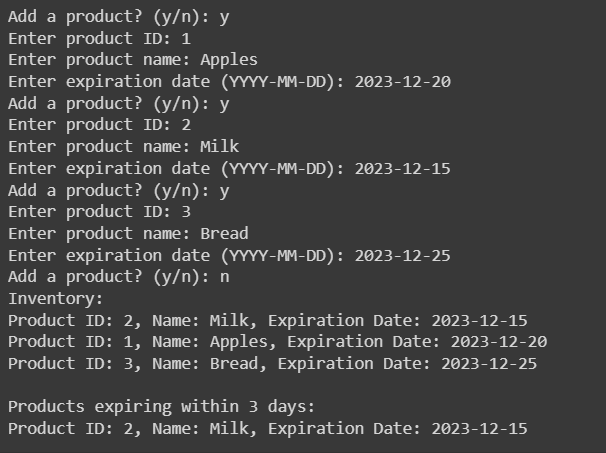
expiring\_products = find\_expiring\_products(root, 3)

print(f"\nProducts expiring within 3 days:")

for product in expiring\_products:

print(f"Product ID: {product.product\_id}, Name: {product.name}, Expiration Date: {product.expiration\_date}")

**RESULT**

****

|  |  |
| --- | --- |
| **EXERCISE 3** | **Special Operations in BST Application - 1** |
| 18.12.2023 |

**AIM**

To implement special operations like finding the minimum and maximum value node and its level, finding the third minimum and maximum value node and its level, finding the sum of all numerical value on the left and right subtree separately, finding the distance between any two given nodes in terms of levels in a BST Application.

**CONCEPT**

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node’s value and the right child has a value greater than the node’s value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree. Binary Search Tree (BST) quickly allows us to maintain a sorted list of numbers. Various Special operations associated with BSTs:

* Minimum and maximum value node and its level:
  + Implement an in-order traversal and store it in an array. Visit array[0] to obtain the minimum value node and array[-1] for the maximum value node.
  + Initialize the level variable to 0. Starting from the root node, compare the keys of the minimum and maximum value nodes with the current node's key. Move left or right in the tree based on the comparison until finding a match or reaching a leaf node.
  + While moving left or right, increment the level variable by 1.
* Third minimum and maximum value node and its level:
  + Implement an in-order traversal and store it in an array. Visit array[2] to obtain the third minimum value node and array[-3] for the third maximum value node.
  + Initialize the level variable to 0. Starting from the root node, compare the keys of the minimum and maximum value nodes with the current node's key. Move left or right in the tree based on the comparison until finding a match or reaching a leaf node.
  + While moving left or right, increment the level variable by 1.
* Sum of all numerical value on the left and right subtree separately:
  + Implement an in-order traversal and store it in an array. Find the index of the root node in the array. Sum the values below the index of the root node to determine the sum of the left subtree and sum the values above the index of the root node to determine the sum of the right subtree.
* Distance between any two given nodes in terms of levels:
  + Initialize the level variable to 0. Starting from the root node, compare the keys of the minimum and maximum value nodes with the current node's key. Move left or right in the tree based on the comparison until finding a match or reaching a leaf node.
  + While moving left or right, increment the level variable by 1.
  + Do the above for both node and subtract it to find the distance between any two given nodes in terms of node.

**APPLICATION**

* BSTs are used for indexing in databases.
* It is used to implement searching algorithms.
* Used for data caching.
* Used in Priority queues.
* Used in spell checkers.

**ADVANTAGES**

* BST is fast in insertion and deletion when balanced. It is fast with a time complexity of O(log n).
* BST is also for fast searching, with a time complexity of O(log n) for most operations.
* We can also do range queries – find keys between N and M (N <= M).
* BST can automatically sort elements as they are inserted, so the elements are always stored in a sorted order.
* BST can be easily modified to store additional data or to support other operations. This makes it flexible.

**CODE**

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

def insert(root, key):

    if root is None:

        return Node(key)

    else:

      if key < root.key:

            root.left = insert(root.left, key)

      else:

          root.right = insert(root.right, key)

      return root

def inorder\_traversal(root):

    if root:

      inorder\_traversal(root.left)

      inorder\_list.append(root.key)

      inorder\_traversal(root.right)

def min(node):

  while node.left:

    node=node.left

    return node.key

def max(node,):

  while node.right:

    node=node.right

    return node.key

def sum\_left\_tree(node):

  sum=0

  while node.left:

    node=node.left

    sum+=node.key

  return sum

def sum\_right\_tree(node):

  sum=0

  while node.right:

    node=node.right

    sum+=node.key

  return sum

def find\_level(root, key, level):

    if root is None:

        return 0

    if root.key == key:

        return level

    left = find\_level(root.left, key, level+1)

    if left != 0:

        return left

    return find\_level(root.right, key, level+1)

root = None

inputs = input("Enter the product id separated by spaces: ")

keys=list(map(int, inputs.split()))

for key in keys:

    root = insert(root, key)

def lca(root, n1, n2):

    # Base Case

    if root is None:

        return None

    if(root.key > n1 and root.key > n2):

        return lca(root.left, n1, n2)

    if(root.key < n1 and root.key < n2):

        return lca(root.right, n1, n2)

    return root

inorder\_list=[]

inorder\_traversal(root)

print("Inorder traversal:",inorder\_list)

for val in range(len(inorder\_list)):

  if val==2:

    third\_min =inorder\_list[val]

level = find\_level(root, third\_min, 1)

print(f"The 3rd minimum element is {third\_min} and its level is {level}")

minval=min(root)

level\_l = find\_level(root, minval, 1)

print("Minimum value is:",minval,"Its level is:",level\_l)

maxval=max(root)

level\_r=find\_level(root, minval, 1)

print("Maximun value is:",maxval,"Its level is:",level\_r)

total\_l=sum\_left\_tree(root)

print("Sum of left subtree",total\_l)

total\_r=sum\_right\_tree(root)

print("Sum of left subtree",total\_r)

node1 = int(input("Enter first node: "))

node2 = int(input("Enter second node: "))

level\_n1=find\_level(root, node1, 1)

level\_n2=find\_level(root, node2, 1)

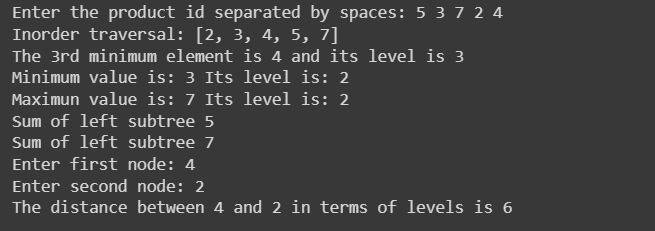
lca\_val=lca(root, node1, node2)

lca\_l=find\_level(root, lca\_val, 1)

distance=(level\_n1-lca\_l)+(level\_n2-lca\_l)

print(f"The distance between {node1} and {node2} in terms of levels is {distance}")

**RESULT**

****

|  |  |
| --- | --- |
| **EXERCISE 4** | **Special Operations in BST Application - 2** |
| 26.12.2023 |

**AIM**

To implement special operations like deleting all the leaf nodes, counting the total number of nodes, finding the height of the tree, creating and displaying the right threads of a leaf node and creating and displaying the left threads of a leaf node in a BST Application.

**CONCEPT**

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node’s value and the right child has a value greater than the node’s value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree. Binary Search Tree (BST) quickly allows us to maintain a sorted list of numbers. Various Special operations associated with BSTs:

* Delete all the leaf nodes:
  + If left child and right child is none in a node while implementing an in-order traversal, then append it in the array.
  + Traverse the array and delete each node in the tree and handle different deletion scenarios
* Total number of nodes:
  + Implement an in-order traversal and store it in an array. Size of the array determines the total number of nodes
* Height of the tree:
  + The maximum of left and right subtrees can be used to find the height of the tree by adding one.
* Right threads of a leaf node:
  + If right child is none in a node while implementing an in-order traversal, then the right thread is successor of that node.
* Left threads of a leaf node:
  + If left child is none in a node while implementing an in-order traversal, then the left thread of node is predecessor of that node.

**APPLICATION**

* BSTs are used for indexing in databases.
* It is used to implement searching algorithms.
* BSTs are used to implement Huffman coding algorithm.
* It is also used to implement dictionaries.
* Used for data caching.
* Used in Priority queues.
* Used in spell checkers.

**ADVANTAGES**

* BST is fast in insertion and deletion when balanced. It is fast with a time complexity of O(log n).
* BST is also for fast searching, with a time complexity of O(log n) for most operations.
* We can also do range queries – find keys between N and M (N <= M).
* BST can automatically sort elements as they are inserted, so the elements are always stored in a sorted order.
* BST can be easily modified to store additional data or to support other operations. This makes it flexible.

**CODE**

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.left\_thread = None

        self.right\_thread = None

def insert(root, key):

    if root is None:

        return Node(key)

    else:

      if key < root.key:

            root.left = insert(root.left, key)

      else:

          root.right = insert(root.right, key)

      return root

def deleteLeaves(root):

    if root is None:

        return None

    # Check if the node is a leaf node

    if root.left is None and root.right is None:

        return None  # Deleting the leaf node by returning None

    # Recursively delete leaf nodes in left and right subtrees

    root.left = deleteLeaves(root.left)

    root.right = deleteLeaves(root.right)

    return root

def tree\_height(root):

    if root is None:

        return -1  # Height of an empty tree is -1

    left\_height = tree\_height(root.left)

    right\_height = tree\_height(root.right)

    # Return the maximum height between left and right subytrees, plus 1 for the current node

    return max(left\_height, right\_height) + 1

def create\_threads(root):

    current = root

    prev = None

    while current:

        if not current.left:

            if prev:

                prev.right\_thread = current

            prev = current

            current = current.right

        else:

            predecessor = current.left

            while predecessor.right and predecessor.right != current:

                predecessor = predecessor.right

            if not predecessor.right:

                predecessor.right = current

                current.left\_thread = predecessor  # Assigning left thread

                current = current.left

            else:

                predecessor.right = None

                if prev:

                    prev.right\_thread = current

                prev = current

                current = current.right

def inorder\_traversal(root):

    if root:

        inorder\_traversal(root.left)

        inorder\_list.append(root.key)

        inorder\_traversal(root.right)

root = None

inputs = input("Enter a list of integers separated by spaces: ")

keys=list(map(int, inputs.split()))

for key in keys:

    root= insert(root, key)

inorder\_list=[]

inorder\_traversal(root)

print("Inorder traversal:",inorder\_list)

print('The number of nodes present in the Tree',len(inorder\_list))

create\_threads(root)

def print\_threads(node):

    if not node:

        return

    print(f"Node {node.key}: Left thread - {node.left\_thread.key if node.left\_thread else None}, Right thread - {node.right\_thread.key if node.right\_thread else None}")

    print\_threads(node.left)

    print\_threads(node.right)

print\_threads(root)

height = tree\_height(root)

print("\nHeight of the tree is:", height)

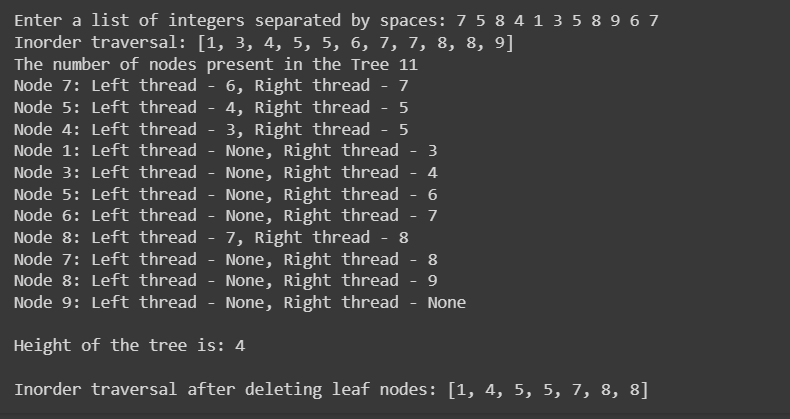
root = deleteLeaves(root)

inorder\_list=[]

inorder\_traversal(root)

print("\nInorder traversal after deleting leaf nodes:", inorder\_list)

**RESULT**

****

|  |  |
| --- | --- |
| **EXERCISE 5** | **AVL Tree Operations** |
| 08.01.2024 |

**AIM**

To implement insertion, deletion, in-order traversal, count the total number of nodes, search for a key and check if the key is empty in AVL Tree.

**CONCEPT**

An **AVL tree** defined as a self-balancing [**Binary Search Tree**](https://www.geeksforgeeks.org/binary-search-tree-data-structure/)(BST) where the difference between heights of left and right subtrees for any node cannot be more than one. The key operations associated with AVL Tree:

* Insertion
  + Insertion in an AVL tree is similar to insertion in a binary search tree. But after inserting an element, check the balance factor. If a node doesn’t satisfy the balance factor condition, then we need to fix the AVL properties using left or right rotations:
    - If there is an imbalance in the left child's right sub-tree, perform a left-right rotation
    - If there is an imbalance in the left child's left sub-tree, perform a right rotation
    - If there is an imbalance in the right child's right sub-tree, perform a left rotation
    - If there is an imbalance in the right child's left sub-tree, perform a right-left rotation
* Deletion
  + Deletion in an AVL tree is similar to deletion in a binary search tree. But after deleting an element, check the balance factor. If any node doesn’t satisfy the balance factor condition, then we need to fix the AVL properties using left or right rotations:
    - If there is an imbalance in the left child's right sub-tree, perform a left-right rotation
    - If there is an imbalance in the left child's left sub-tree, perform a right rotation
    - If there is an imbalance in the right child's right sub-tree, perform a left rotation
    - If there is an imbalance in the right child's left sub-tree, perform a right-left rotation
* In-order Traversal
  + Visiting and processing all the nodes in the AVL tree which follows Left-Parent-Right pattern.
* Total number of nodes
  + Implement an in-order traversal and store it in an array. Size of the array determines the total number of nodes
* Search for a key
  + Finding a specific key value within the BST.
  + Start from the root node and compare the target key with the current node’s key. Move left or right in the tree based on the comparison until finding a match or reaching a leaf node.
* Checking if the tree is empty
  + To check if an AVL tree is empty, examine whether the root of the tree is null or not.

**APPLICATION**

* It is used to index huge records in a database and also to efficiently search in that.
* For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
* Database applications, where insertions and deletions are less common but frequent data lookups are necessary
* Software that needs optimized search.
* It is applied in corporate areas and storyline games.

**ADVANTAGES**

* AVL trees can self-balance themselves.
* It is surely not skewed.
* It provides faster lookups than Red-Black Trees
* Better searching time complexity compared to other trees like binary tree.
* Height cannot exceed log(N), where, N is the total number of nodes in the tree.

**CODE**

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1

class AVLTree:

    def \_\_init\_\_(self):

        self.root = None

    def count\_nodes(self, root):

        if root is None:

            return 0

        return 1 + self.count\_nodes(root.left) + self.count\_nodes(root.right)

    def \_height(self, node):

        if node is None:

            return 0

        return node.height

    def \_balance(self, node):

        if node is None:

            return 0

        return self.\_height(node.left) - self.\_height(node.right)

    def \_right\_rotate(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 \_left\_rotate(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 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)

        root.height = 1 + max(self.\_height(root.left), self.\_height(root.right))

        balance = self.\_balance(root)

        if balance > 1 and key < root.left.key:

            return self.\_right\_rotate(root)

        # Right Right Case

        if balance < -1 and key > root.right.key:

            return self.\_left\_rotate(root)

        if balance > 1 and key > root.left.key:

            root.left = self.\_left\_rotate(root.left)

            return self.\_right\_rotate(root)

        # Right Left Case

        if balance < -1 and key < root.right.key:

            root.right = self.\_right\_rotate(root.right)

            return self.\_left\_rotate(root)

        return root

    def delete(self, root, key):

        if root is None:

            return root

        if key < root.key:

            root.left = self.delete(root.left, key)

        elif key > root.key:

            root.right = self.delete(root.right, key)

        else:

            if root.left is None:

                temp = root.right

                root = None

                return temp

            elif root.right is None:

                temp = root.left

                root = None

                return temp

            temp = self.\_min\_value\_node(root.right)

            root.key = temp.key

            root.right = self.delete(root.right, temp.key)

        if root is None:

            return root

        root.height = 1 + max(self.\_height(root.left), self.\_height(root.right))

        balance = self.\_balance(root)

        if balance > 1 and self.\_balance(root.left) >= 0:

            return self.\_right\_rotate(root)

        if balance > 1 and self.\_balance(root.left) < 0:

            root.left = self.\_left\_rotate(root.left)

            return self.\_right\_rotate(root)

        if balance < -1 and self.\_balance(root.right) <= 0:

            return self.\_left\_rotate(root)

        if balance < -1 and self.\_balance(root.right) > 0:

            root.right = self.\_right\_rotate(root.right)

            return self.\_left\_rotate(root)

        return root

    def \_min\_value\_node(self, node):

        current = node

        while current.left is not None:

            current = current.left

        return current

    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)

    # Inorder traversal of the AVL tree

    def inorder\_traversal(self, root):

        if root is not None:

            self.inorder\_traversal(root.left)

            print(root.key, end=" ")

            self.inorder\_traversal(root.right)

    def is\_empty(self):

        return self.root is None

avl\_tree = AVLTree()

print("Is the AVL tree empty?", avl\_tree.is\_empty())

root = None

inputs = input("Enter a list of integers separated by spaces: ")

keys = list(map(int, inputs.split()))

for key in keys:

    root = avl\_tree.insert(root, key)

print("Inorder traversal before deletion:", end=" ")

avl\_tree.inorder\_traversal(root)

print("\nNumber of nodes in the AVL tree:", avl\_tree.count\_nodes(root))

print("Is the AVL tree empty?", avl\_tree.is\_empty())

#print("Is the AVL tree empty?", 'False')

print("Search for key 10:", avl\_tree.search(root, 10))

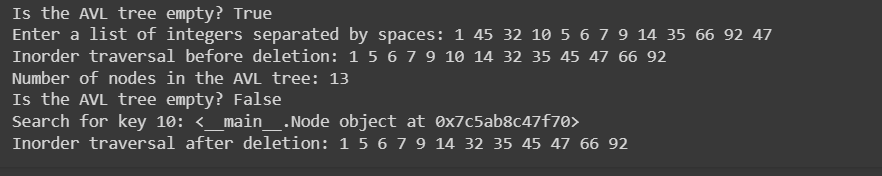
root = avl\_tree.delete(root, 10)

print("Inorder traversal after deletion:", end=" ")

avl\_tree.inorder\_traversal(root)

print()

**RESULT**

****

|  |  |
| --- | --- |
| **EXERCISE 6** | **Heap Tree Operations** |
| 22.01.2024 |

**AIM**

To implement insertion, deletion, display the elements in array format, print the children of given node and print the index of 1st leaf and last non-leaf nodes in Heap Tree.

**CONCEPT**

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

* always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
* always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

The key operations associated with Heap Tree:

* Insertion
  + A new element is always inserted at the last child of the original heap. The new element may now violate the **heap property** that a heap must satisfy.
  + Therefore, an operation known as **reheapify upward** is performed on the heap.
  + If the value is **greater** (in a max heap) or **smaller** (in a min heap) it is swapped with its parent. The process is then continued from the parent node recursively until the heap property is satisfied or the root node is hit. And also store it in an array.
* Deletion
  + An element is always deleted from the root of the heap. After deleting, swap the last node in te heap to the root. This causes the heap to violate the heap property.
  + Similar to inserting an element, an operation known as **reheapify downward** is performed on the heap. The value of the root node is first replaced with the largest (or smallest) value amongst its children. Then the down heap property is repeated from this child again recursively until we hit the leaf node. And also store it in an array.
* Elements in array format
  + Just print the array where max heap or min heap is performed.
* Children of given node
  + Given the node, first find the index of the node and store it in variable ‘I’. The left child is array[2i+1] and the right child is array[2i+2].
* Index of 1st leaf and last non-leaf node
  + The last non-leaf node is array[(n/2 – 1)-1].
  + If n is even, then the first leaf node is array[n/2] and if n is odd, then the first leaf node is array[n/2 – 1].

**APPLICATION**

* **Heapsort**: This is one of the best in-place sorting methods with no quadratic worst-case scenarios. This is because the minimum or maximum element is always the root of the heap.
* **Implementing priority queues**: As the highest (or lowest) priority element is always stored at the root of the heap, they could be accessed quickly.
* **Selection algorithms**: A heap allows access to the min or max element in constant time, and other selections (such as median or kth-element) can be done in sub-linear time on data that is in a heap.
* **Graph algorithms**: By using heaps as internal traversal data structures, run times can be reduced by polynomial order.

**ADVANTAGES**

* **Efficient insertion and deletion**: The heap data structure allows efficient insertion and deletion of elements. When a new element is added to the heap, it is placed at the bottom of the heap and moved up to its correct position using the reheapify upward operation. Similarly, when an element is removed from the heap, it is replaced by the bottom element, and the heap is restructured using the reheapify downward operation.
* **Efficient priority queue**: The heap data structure is commonly used to implement a priority queue, where the highest priority element is always at the top of the heap. The heap allows constant-time access to the highest priority element, making it an efficient data structure for implementing priority queues.
* **Guaranteed access to the maximum or minimum element**: In a max-heap, the top element is always the maximum element, and in a min-heap, the top element is always the minimum element. This provides guaranteed access to the maximum or minimum element in the heap, making it useful in algorithms that require access to the extreme values.
* **Space efficiency**: The heap data structure requires less memory compared to other data structures, such as linked lists or arrays, as it stores elements in a complete binary tree structure.
* **Heap-sort algorithm**: The heap data structure forms the basis for the heap-sort algorithm, which is an efficient sorting algorithm that has a worst-case time complexity of O(n log n).

**CODE**

class MaxHeap:

    def \_\_init\_\_(self):

        self.heap = []

    def insert(self, value):

        self.heap.append(value)

        self.\_heapify\_up()

    def delete(self):

        if len(self.heap) == 0:

            return None

        if len(self.heap) == 1:

            return self.heap.pop()

        root = self.heap[0]

        self.heap[0] = self.heap.pop(-1)

        self.\_heapify\_down()

        return root

    def display(self):

        print("Heap:", self.heap)

    def print\_children(self, index):

        left\_child\_index = 2 \* index + 1

        right\_child\_index = 2 \* index + 2

        print(f"Children of node at index {index}:")

        if left\_child\_index < len(self.heap):

            print(f"Left Child: {self.heap[left\_child\_index]}")

        if right\_child\_index < len(self.heap):

            print(f"Right Child: {self.heap[right\_child\_index]}")

    def first\_leaf\_index(self):

        return len(self.heap) // 2

    def last\_non\_leaf\_index(self):

        return (len(self.heap) - 1) // 2

    def \_heapify\_up(self):

        index = len(self.heap) - 1

        while index > 0:

            parent\_index = (index - 1) // 2

            if self.heap[index] > self.heap[parent\_index]:

                self.heap[index], self.heap[parent\_index] = self.heap[parent\_index], self.heap[index]

                index = parent\_index

            else:

                break

    def \_heapify\_down(self):

        index = 0

        while True:

            left\_child\_index = 2 \* index + 1

            right\_child\_index = 2 \* index + 2

            largest = index

            if left\_child\_index < len(self.heap) and self.heap[left\_child\_index] > self.heap[largest]:

                largest = left\_child\_index

            if right\_child\_index < len(self.heap) and self.heap[right\_child\_index] > self.heap[largest]:

                largest = right\_child\_index

            if largest != index:

                self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]

                index = largest

            else:

                break

heap = MaxHeap()

while True:

    print("\n1. Insert")

    print("2. Delete")

    print("3. Display")

    print("4. Print Children")

    print("5. First Leaf Index")

    print("6. Last Non-Leaf Index")

    print("7. Quit")

    choice = int(input("Enter your choice: "))

    if choice == 1:

        value = int(input("Enter the value to insert: "))

        heap.insert(value)

    elif choice == 2:

        deleted\_element = heap.delete()

        if deleted\_element is not None:

            print(f"Deleted Element: {deleted\_element}")

        else:

            print("Heap is empty.")

    elif choice == 3:

        heap.display()

    elif choice == 4:

        index = int(input("Enter the index to print children: "))

        heap.print\_children(index)

    elif choice == 5:

        print(f"First Leaf Index: {heap.first\_leaf\_index()}")

    elif choice == 6:

        print(f"Last Non-Leaf Index: {heap.last\_non\_leaf\_index()}")

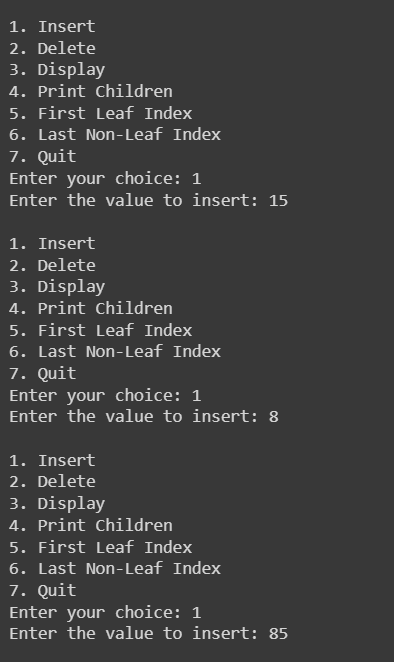
    elif choice == 7:

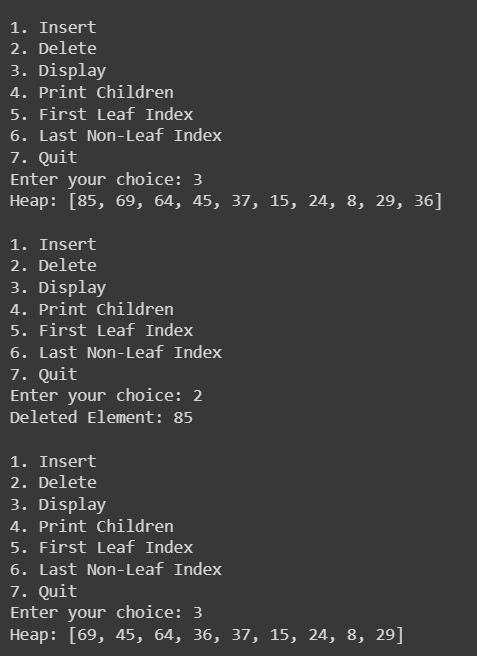
        break

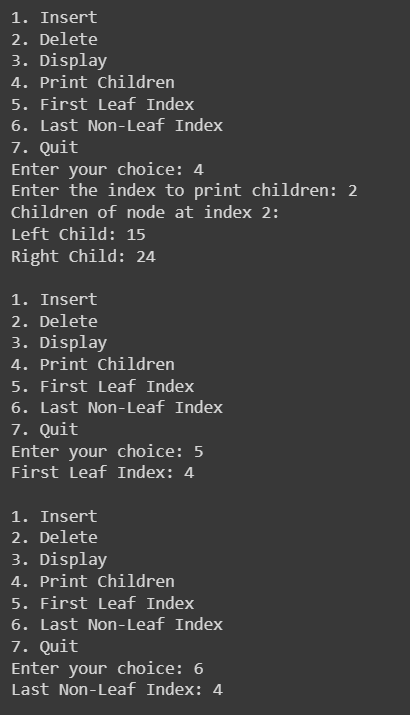
    else:

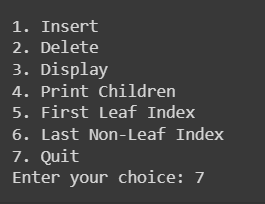
        print("Invalid choice. Please enter a valid option.")

**RESULT**

****

****

****

****

|  |  |
| --- | --- |
| **EXERCISE 7** | **Application of Heap Tree** |
| 24.01.2024 |

**AIM**

To implement insertion, deletion, display the elements in array format in Heap Application.

**APPLICATION CHOSEN**

To do list application.

**CONCEPT**

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

* always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
* always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

The key operations associated with Heap Tree:

* Insertion
  + A new element is always inserted at the last child of the original heap. The new element may now violate the **heap property** that a heap must satisfy.
  + Therefore, an operation known as **reheapify upward** is performed on the heap.
  + If the value is **greater** (in a max heap) or **smaller** (in a min heap) it is swapped with its parent. The process is then continued from the parent node recursively until the heap property is satisfied or the root node is hit. And also store it in an array.
* Deletion
  + An element is always deleted from the root of the heap. After deleting, swap the last node in te heap to the root. This causes the heap to violate the heap property.
  + Similar to inserting an element, an operation known as **reheapify downward** is performed on the heap. The value of the root node is first replaced with the largest (or smallest) value amongst its children. Then the down heap property is repeated from this child again recursively until we hit the leaf node. And also store it in an array.
* Elements in array format
  + Just print the array where max heap or min heap is performed.
* Children of given node
  + Given the node, first find the index of the node and store it in variable ‘I’. The left child is array[2i+1] and the right child is array[2i+2].
* Index of 1st leaf and last non-leaf node
  + The last non-leaf node is array[(n/2 – 1)-1].
  + If n is even, then the first leaf node is array[n/2] and if n is odd, then the first leaf node is array[n/2 – 1].

**APPLICATION**

* **Heapsort**: This is one of the best in-place sorting methods with no quadratic worst-case scenarios. This is because the minimum or maximum element is always the root of the heap.
* **Implementing priority queues**: As the highest (or lowest) priority element is always stored at the root of the heap, they could be accessed quickly.
* **Selection algorithms**: A heap allows access to the min or max element in constant time, and other selections (such as median or kth-element) can be done in sub-linear time on data that is in a heap.
* **Graph algorithms**: By using heaps as internal traversal data structures, run times can be reduced by polynomial order.

**ADVANTAGES**

* **Efficient insertion and deletion**: The heap data structure allows efficient insertion and deletion of elements. When a new element is added to the heap, it is placed at the bottom of the heap and moved up to its correct position using the reheapify upward operation. Similarly, when an element is removed from the heap, it is replaced by the bottom element, and the heap is restructured using the reheapify downward operation.
* **Efficient priority queue**: The heap data structure is commonly used to implement a priority queue, where the highest priority element is always at the top of the heap. The heap allows constant-time access to the highest priority element, making it an efficient data structure for implementing priority queues.
* **Guaranteed access to the maximum or minimum element**: In a max-heap, the top element is always the maximum element, and in a min-heap, the top element is always the minimum element. This provides guaranteed access to the maximum or minimum element in the heap, making it useful in algorithms that require access to the extreme values.
* **Space efficiency**: The heap data structure requires less memory compared to other data structures, such as linked lists or arrays, as it stores elements in a complete binary tree structure.
* **Heap-sort algorithm**: The heap data structure forms the basis for the heap-sort algorithm, which is an efficient sorting algorithm that has a worst-case time complexity of O(n log n).

**CODE**

import heapq

class ToDoList:

    def \_\_init\_\_(self):

        self.tasks = []

    def add\_task(self, task, priority):

        # Each task is represented as a tuple (priority, task)

        heapq.heappush(self.tasks, (priority, task))

    def get\_next\_task(self):

        if self.tasks:

            return self.tasks[0][1]  # Return the task with the highest priority

        else:

            return None

    def complete\_task(self):

        if self.tasks:

            priority, task = heapq.heappop(self.tasks)

            print(f"Task '{task}' completed with priority {priority}")

        else:

            print("No tasks to complete.")

todo\_list = ToDoList()

while True:

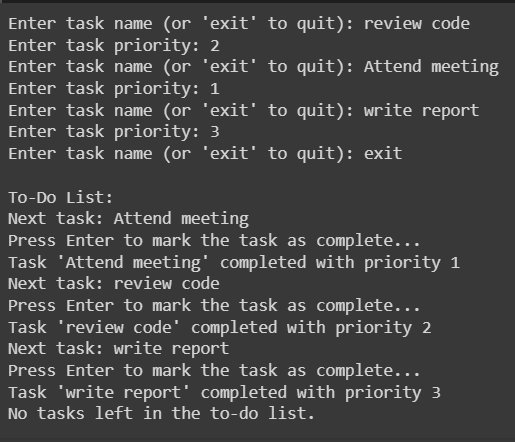
    task\_name = input("Enter task name (or 'exit' to quit): ")

    if task\_name.lower() == 'exit':

        break

print("\nTo-Do List:")

**RESULT**

****