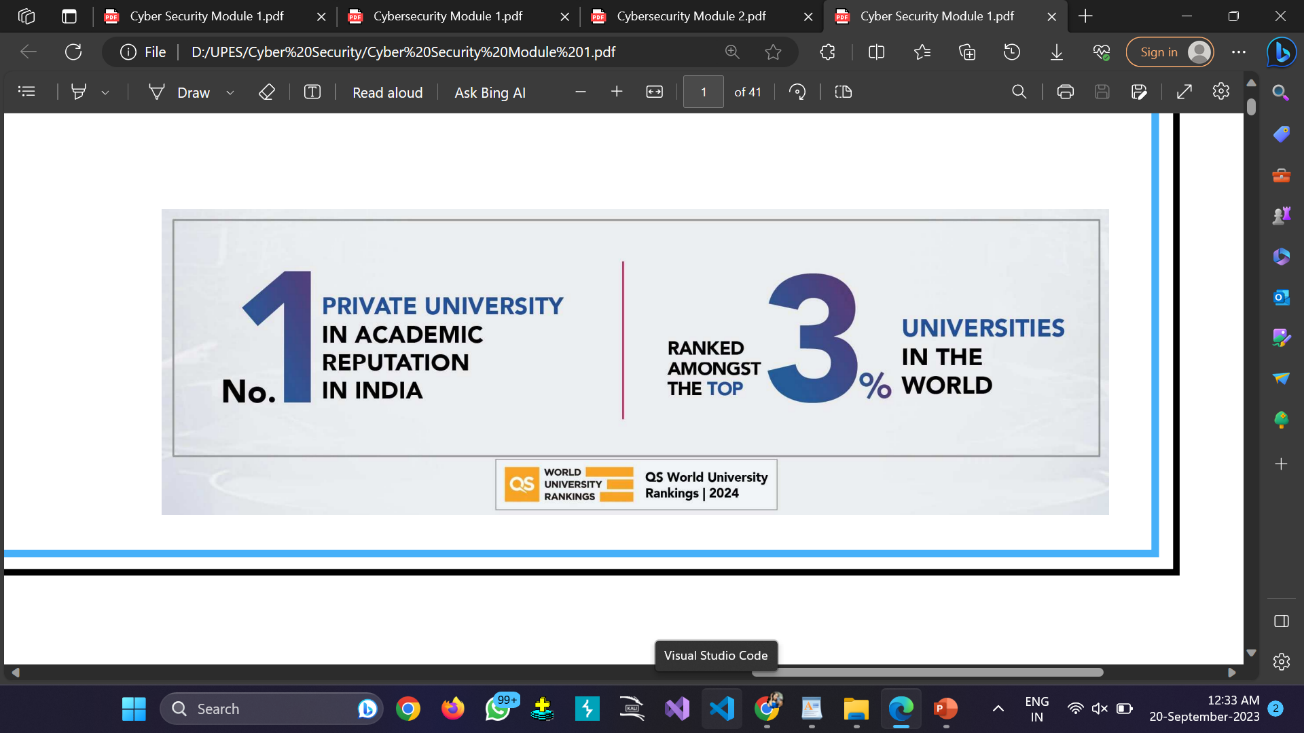
A picture containing text, clipart

Description automatically generated

1

**Lab Experiment: 10**

**Student Detail:**

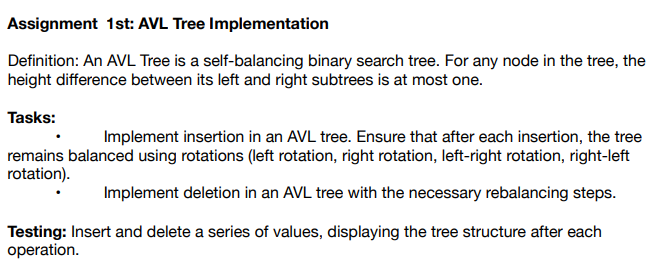
**• Name:** Prashant Joshi

**• Student ID:** 590010879

**• Branch:** MCA

**• Batch:** B1

**• Instructor:** Dr. Sourbh Kumar



Solution:

#include <stdio.h>

#include <stdlib.h>

// Definition of a node in the AVL tree

struct AVLNode {

int data;

struct AVLNode\* left;

struct AVLNode\* right;

int height; // Height of the node (used for balancing)

};

// Function to get the height of a node

int height(struct AVLNode\* node) {

if (node == NULL) return 0;

return node->height;

}

// Function to get the balance factor of a node

int getBalance(struct AVLNode\* node) {

if (node == NULL) return 0;

return height(node->left) - height(node->right);

}

// Function to perform a right rotation

struct AVLNode\* rightRotate(struct AVLNode\* y) {

struct AVLNode\* x = y->left;

struct AVLNode\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

// Return new root

return x;

}

// Function to perform a left rotation

struct AVLNode\* leftRotate(struct AVLNode\* x) {

struct AVLNode\* y = x->right;

struct AVLNode\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

// Return new root

return y;

}

// Function to insert a node in the AVL tree

struct AVLNode\* insert(struct AVLNode\* node, int data) {

// 1. Perform the normal BST insert

if (node == NULL) {

struct AVLNode\* newNode = (struct AVLNode\*)malloc(sizeof(struct AVLNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

newNode->height = 1;

return newNode;

}

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

else // Duplicate keys are not allowed

return node;

// 2. Update the height of this ancestor node

node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));

// 3. Get the balance factor to check whether this node became unbalanced

int balance = getBalance(node);

// 4. If the node becomes unbalanced, then there are 4 cases

// Left Left Case

if (balance > 1 && data < node->left->data)

return rightRotate(node);

// Right Right Case

if (balance < -1 && data > node->right->data)

return leftRotate(node);

// Left Right Case

if (balance > 1 && data > node->left->data) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && data < node->right->data) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

// Return the (unchanged) node pointer

return node;

}

// Function to find the node with the minimum value

struct AVLNode\* minNode(struct AVLNode\* node) {

struct AVLNode\* current = node;

while (current->left != NULL) {

current = current->left;

}

return current;

}

// Function to delete a node in the AVL tree

struct AVLNode\* delete(struct AVLNode\* root, int data) {

// Step 1: Perform normal BST delete

if (root == NULL) return root;

if (data < root->data)

root->left = delete(root->left, data);

else if (data > root->data)

root->right = delete(root->right, data);

else {

// Node to be deleted is found

// Node with only one child or no child

if (root->left == NULL) {

struct AVLNode\* temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL) {

struct AVLNode\* temp = root->left;

free(root);

return temp;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

struct AVLNode\* temp = minNode(root->right);

// Copy the inorder successor's content to this node

root->data = temp->data;

// Delete the inorder successor

root->right = delete(root->right, temp->data);

}

// Step 2: Update height of the current node

root->height = 1 + (height(root->left) > height(root->right) ? height(root->left) : height(root->right));

// Step 3: Get the balance factor to check whether this node became unbalanced

int balance = getBalance(root);

// Step 4: If the node becomes unbalanced, then there are 4 cases

// Left Left Case

if (balance > 1 && getBalance(root->left) >= 0)

return rightRotate(root);

// Right Right Case

if (balance < -1 && getBalance(root->right) <= 0)

return leftRotate(root);

// Left Right Case

if (balance > 1 && getBalance(root->left) < 0) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

// Right Left Case

if (balance < -1 && getBalance(root->right) > 0) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

// Function to print the tree (in-order traversal)

void inOrder(struct AVLNode\* root) {

if (root != NULL) {

inOrder(root->left);

printf("%d ", root->data);

inOrder(root->right);

}

}

// Driver code to test the AVL Tree implementation

int main() {

struct AVLNode\* root = NULL;

// Insertion in AVL Tree

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

printf("In-order traversal after insertions: ");

inOrder(root);

printf("\n");

// Deletion in AVL Tree

root = delete(root, 20);

root = delete(root, 25);

printf("In-order traversal after deletions: ");

inOrder(root);

printf("\n");

return 0;

}

Output:

Inserting values:



In-order traversal after insertions:

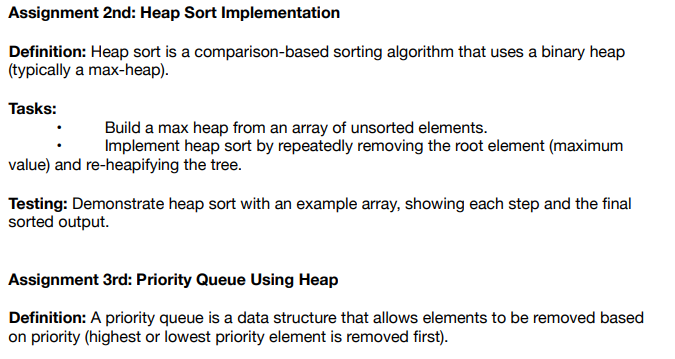


Deleting Values:



In-Order Trivarsal After deletion





Solution:

#include <stdio.h>

// Function to swap two elements in an array

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted at index i

void heapify(int arr[], int n, int i) {

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left child index

int right = 2 \* i + 2; // right child index

// If left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root, swap and continue heapifying

if (largest != i) {

swap(&arr[i], &arr[largest]);

heapify(arr, n, largest);

}

}

// Function to build a max-heap from an unsorted array

void buildMaxHeap(int arr[], int n) {

// Start from the last non-leaf node and heapify each node

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

}

// Function to implement heap sort

void heapSort(int arr[], int n) {

// Build a max-heap

buildMaxHeap(arr, n);

// One by one extract elements from the heap

for (int i = n - 1; i >= 1; i--) {

// Move current root to the end

swap(&arr[0], &arr[i]);

// Call heapify on the reduced heap

heapify(arr, i, 0);

}

}

// 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[] = { 12, 11, 13, 5, 6, 7 };

int n = sizeof(arr) / sizeof(arr[0]);

printf("Unsorted array: ");

printArray(arr, n);

// Perform heap sort

heapSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

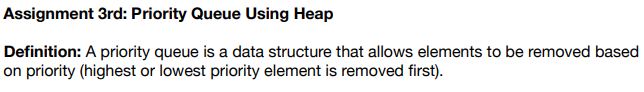
Output:

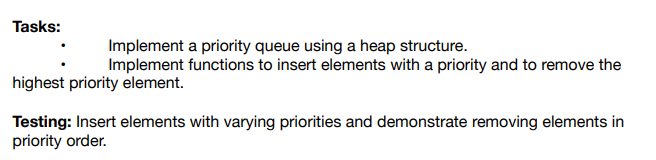


**Step-by-step Process:**

1. **Build Max-Heap**:
   * Start from index n/2 - 1 and move towards index 0.
   * For i = 2 (arr[2] = 13), no change is needed since it's already larger than its children.
   * For i = 1 (arr[1] = 11), swap with arr[3] = 5, resulting in {12, 5, 13, 11, 6, 7}.
   * For i = 0 (arr[0] = 12), swap with arr[2] = 13, resulting in {13, 12, 7, 11, 6, 5}.
2. **Heap Sort**:
   * Swap arr[0] = 13 with arr[5] = 5, resulting in {5, 12, 7, 11, 6, 13}.
   * Heapify the root: {12, 11, 7, 5, 6}.
   * Swap arr[0] = 12 with arr[4] = 6, resulting in {6, 11, 7, 5, 12, 13}.
   * Heapify the root: {11, 6, 7, 5}.
   * Continue this process until the array is fully sorted.







Solution:

#include <stdio.h>

#include <stdlib.h>

// Structure to represent the priority queue (max-heap)

struct PriorityQueue {

int\* arr; // Array to store the heap

int size; // Current size of the heap

int capacity; // Maximum capacity of the heap

};

// Function to create a priority queue with the given capacity

struct PriorityQueue\* createPriorityQueue(int capacity) {

struct PriorityQueue\* pq = (struct PriorityQueue\*)malloc(sizeof(struct PriorityQueue));

pq->capacity = capacity;

pq->size = 0;

pq->arr = (int\*)malloc(capacity \* sizeof(int));

return pq;

}

// Function to get the parent index of a given node

int parent(int i) {

return (i - 1) / 2;

}

// Function to get the left child index of a given node

int leftChild(int i) {

return 2 \* i + 1;

}

// Function to get the right child index of a given node

int rightChild(int i) {

return 2 \* i + 2;

}

// Function to swap two elements in the heap

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to maintain the max-heap property (heapify) starting from index i

void heapify(struct PriorityQueue\* pq, int i) {

int largest = i; // Assume the current node is the largest

int left = leftChild(i); // Left child index

int right = rightChild(i); // Right child index

// Check if the left child exists and is greater than the largest element

if (left < pq->size && pq->arr[left] > pq->arr[largest]) {

largest = left;

}

// Check if the right child exists and is greater than the largest element

if (right < pq->size && pq->arr[right] > pq->arr[largest]) {

largest = right;

}

// If the largest is not the current node, swap and heapify the affected subtree

if (largest != i) {

swap(&pq->arr[i], &pq->arr[largest]);

heapify(pq, largest);

}

}

// Function to insert a new element into the priority queue

void insert(struct PriorityQueue\* pq, int key) {

// Check if the priority queue is full

if (pq->size == pq->capacity) {

printf("Priority queue is full\n");

return;

}

// Insert the new key at the end of the heap

pq->size++;

int i = pq->size - 1;

pq->arr[i] = key;

// Fix the max-heap property if it's violated by the new insertion

while (i != 0 && pq->arr[parent(i)] < pq->arr[i]) {

swap(&pq->arr[i], &pq->arr[parent(i)]);

i = parent(i);

}

}

// Function to remove the highest priority element (root) from the priority queue

int removeMax(struct PriorityQueue\* pq) {

// Check if the heap is empty

if (pq->size <= 0) {

printf("Priority queue is empty\n");

return -1;

}

// If there's only one element, remove it

if (pq->size == 1) {

pq->size--;

return pq->arr[0];

}

// Otherwise, replace the root with the last element

int root = pq->arr[0];

pq->arr[0] = pq->arr[pq->size - 1];

pq->size--;

// Heapify the root to restore the max-heap property

heapify(pq, 0);

return root;

}

// Function to print the elements of the priority queue

void printPriorityQueue(struct PriorityQueue\* pq) {

for (int i = 0; i < pq->size; i++) {

printf("%d ", pq->arr[i]);

}

printf("\n");

}

int main() {

// Create a priority queue with a capacity of 10

struct PriorityQueue\* pq = createPriorityQueue(10);

// Insert elements with varying priorities

insert(pq, 10);

insert(pq, 20);

insert(pq, 5);

insert(pq, 30);

insert(pq, 15);

insert(pq, 25);

// Print the priority queue (max-heap)

printf("Priority Queue (Max-Heap): ");

printPriorityQueue(pq);

// Remove elements one by one in priority order (highest priority first)

printf("Removing elements based on priority:\n");

printf("Removed: %d\n", removeMax(pq)); // Should remove 30

printf("Removed: %d\n", removeMax(pq)); // Should remove 25

printf("Removed: %d\n", removeMax(pq)); // Should remove 20

printf("Removed: %d\n", removeMax(pq)); // Should remove 15

printf("Removed: %d\n", removeMax(pq)); // Should remove 10

printf("Removed: %d\n", removeMax(pq)); // Should remove 5

// Print the priority queue after all removals

printf("Priority Queue after all removals: ");

printPriorityQueue(pq);

return 0;

}

Output:

