National University of Computer and Emerging Sciences



Name: Muhammad Suleman

Roll #: 22F-3350

Section: BCS-4B

Lab # 12

# Problem 01

#include <iostream>

using namespace std;

class MaxHeap

{

private:

int\* heap;

int capacity;

int size;

int parent(int i)

{

return i / 2;

}

int left(int i)

{

return 2 \* i;

}

int right(int i)

{

return 2 \* i + 1;

}

void heapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l <= size && heap[l] < heap[i])

{

smallest = l;

}

if (r <= size && heap[r] < heap[smallest])

{

smallest = r;

}

if (smallest != i)

{

swap(heap[i], heap[smallest]);

heapify(smallest);

}

}

public:

MaxHeap(int cap)

{

capacity = cap;

size = 0;

heap = new int[cap + 1];

}

~MaxHeap() {

delete[] heap;

}

int getMin()

{

if (size == 0)

{

return numeric\_limits<int>::max();

}

else

{

return heap[1];

}

}

int extractMin()

{

if (size == 0)

{

return -1;

}

int min = heap[1];

heap[1] = heap[size];

size--;

heapify(1);

return min;

}

void decreaseKey(int i, int new\_val)

{

heap[i] = new\_val;

while (i > 1 && heap[parent(i)] > heap[i])

{

swap(heap[i], heap[parent(i)]);

i = parent(i);

}

}

void insert(int key)

{

if (size == capacity)

{

return;

}

size++;

heap[size] = numeric\_limits<int>::max();

decreaseKey(size, key);

}

void deleteKey(int i)

{

decreaseKey(i, numeric\_limits<int>::min());

extractMin();

}

};

int main() {

int arr[] = { 10, 14, 2, 5, 33, 9, 3, 23, 73, 7 };

MaxHeap heap(10);

for (int i = 0; i < 10; i++)

{

heap.insert(arr[i]);

}

cout << "Minimum element: " << heap.getMin() << endl;

cout << "Extracted minimum: " << heap.extractMin() << endl;

heap.decreaseKey(5, 1);

cout << "Decreased key: " << heap.getMin() << endl;

heap.insert(0);

cout << "New minimum: " << heap.getMin() << endl;

heap.deleteKey(3);

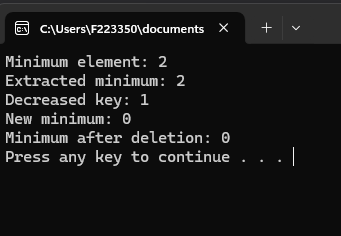
cout << "Minimum after deletion: " << heap.getMin() << endl;

system("pause");

return 0;

}

## Output



# Problem 02

#include <iostream>

using namespace std;

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

{

int parent = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[parent])

{

parent = left;

}

if (right < n && arr[right] > arr[parent])

{

parent = right;

}

if (parent != i)

{

swap(arr[i], arr[parent]);

heapify(arr, n, parent);

}

}

void heapSort(int arr[], int n)

{

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

heapify(arr, n, i);

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

{

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

heapify(arr, i, 0);

}

}

void printArray(int arr[], int n)

{

for (int i = 0; i < n; ++i)

{

cout << arr[i] << " ";

}

cout << endl;

}

int main()

{

int arr[] = { 90,80,70,60,50,40,30,20,10 };

int n = 9;

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

heapify(arr, n, i);

}

cout << "After heapifying array: ";

printArray(arr, n);

heapSort(arr, n);

cout << "Sorted array: ";

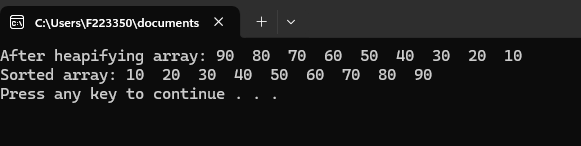
printArray(arr, n);

system("pause");

return 0;

}

## Output



# Problem 03

#include <iostream>

using namespace std;

void kLargest(int \*arr, int n, int k)

{

int \*kLargest = new int[k];

for (int i = 0; i < k; i++)

{

kLargest[i] = arr[i];

}

for (int i = 0; i < k; i++)

{

for (int j = i + 1; j < k; j++)

{

if (kLargest[i] > kLargest[j])

{

swap(kLargest[i], kLargest[j]);

}

}

}

for (int i = k; i < n; i++)

{

if (arr[i] > kLargest[0])

{

kLargest[0] = arr[i];

for (int j = 0; j < k - 1; j++)

{

if (kLargest[j] > kLargest[j + 1])

{

swap(kLargest[j], kLargest[j + 1]);

}

}

}

}

cout << k << " largest elements are: ";

for (int i = k - 1; i >= 0; i--) {

cout << kLargest[i] << " ";

}

cout << endl;

}

int main()

{

int arr[] = { 10,9,8,7,6,5,4,3,2,1 };

int n = 10;

cout << "Give array: ";

for (int i = 0; i < 10; i++)

{

cout << arr[i] << " ";

}

cout << endl;

int k = 3;

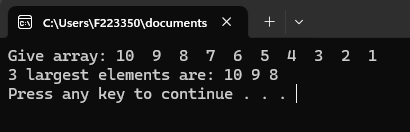
kLargest(arr, n, k);

system("pause");

return 0;

}

## Output



# Problem 04

#include <iostream>

using namespace std;

template<typename T>

struct Node

{

T data;

Node\* left;

Node\* right;

Node(T value)

: data(value), left(nullptr), right(nullptr) {}

};

template<typename T>

class BinarySearchTree

{

private:

Node<T>\* root;

void destroyTree(Node<T>\* node)

{

if (node != nullptr)

{

destroyTree(node->left);

destroyTree(node->right);

delete node;

}

}

Node<T>\* insert(Node<T>\* root, T data)

{

if (root == nullptr)

{

Node<T>\* newNode = new Node<T>(data);

return newNode;

}

if (data < root->data)

{

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

}

else if (data > root->data)

{

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

}

return root;

}

void inorderTraversal(Node<T>\* root, T arr[], int& index)

{

if (root == nullptr)

{

return;

}

inorderTraversal(root->left, arr, index);

arr[index++] = root->data;

inorderTraversal(root->right, arr, index);

}

void preorderTraversal(Node<T>\* root, T arr[], int& index)

{

if (root == nullptr)

{

return;

}

root->data = arr[index++];

preorderTraversal(root->left, arr, index);

preorderTraversal(root->right, arr, index);

}

void prvtPreOrder(Node<T>\* root)

{

if (!root)

{

return;

}

cout << root->data << " ";

prvtPreOrder(root->left);

prvtPreOrder(root->right);

}

public:

BinarySearchTree()

: root(nullptr) {}

~BinarySearchTree()

{

destroyTree(root);

}

void insert(T data)

{

root = insert(root, data);

}

void convertToMinHeap()

{

int size = getSize(root);

T\* arr = new T[size];

int index = 0;

inorderTraversal(root, arr, index);

index = 0;

preorderTraversal(root, arr, index);

delete[] arr;

}

int getSize(Node<T>\* node)

{

if (node == nullptr)

return 0;

return 1 + getSize(node->left) + getSize(node->right);

}

void preOrder()

{

prvtPreOrder(root);

}

};

int main()

{

cout << "Convertion from BST to MinHeap" << endl;

BinarySearchTree<int> tree;

tree.insert(50);

tree.insert(30);

tree.insert(70);

tree.insert(20);

tree.insert(40);

tree.insert(60);

tree.insert(80);

cout << "Before conversion preOrder: ";

tree.preOrder();

cout << endl;

tree.convertToMinHeap();

cout << "After conversion preOrder: ";

tree.preOrder();

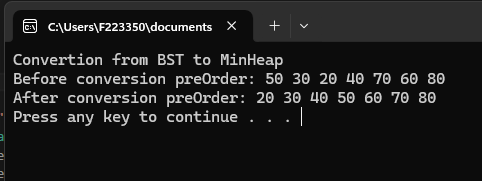
cout << endl;

system("pause");

return 0;

}

## Output



# Problem 05

#include <iostream>

using namespace std;

void MaxHeapify(int arr[], int i, int size)

{

int l = 2 \* i + 1;

int r = 2 \* i + 2;

int largest = i;

if (l < size && arr[l] > arr[i])

{

largest = l;

}

if (r < size && arr[r] > arr[largest])

{

largest = r;

}

if (largest != i)

{

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

MaxHeapify(arr, largest, size);

}

}

void convertMaxHeap(int arr[], int size)

{

for (int i = (size - 2) / 2; i >= 0; --i)

{

MaxHeapify(arr, i, size);

}

}

void printArray(int\* arr, int size)

{

for (int i = 0; i < size; ++i)

{

cout << arr[i] << " ";

}

}

int main()

{

int arr[] = { 1,2,3,4,5,6,7,8,9,10 };

int n = 10;

cout << "Min Heap array : ";

printArray(arr, n);

convertMaxHeap(arr, n);

cout << endl;

cout << "Max Heap array : ";

printArray(arr, n);

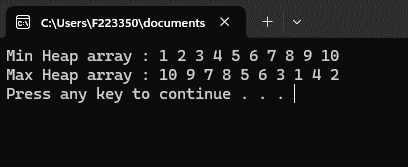
cout << endl;

system("pause");

return 0;

}

## Output



# Problem 06

#include <iostream>

using namespace std;

class Task

{

public:

int id;

int priority;

int executionTime;

Task \*next;

Task(int id, int priority, int executionTime)

: id(id), priority(priority), executionTime(executionTime), next(nullptr) {}

void printTaskInfo();

};

class PriorityQueue

{

private:

Task \*head;

void insertTask(Task \*newTask);

public:

PriorityQueue()

: head(nullptr) {}

void insert(Task task);

Task extractMin();

Task peek();

};

int main()

{

PriorityQueue priorityQ;

priorityQ.insert(Task(1, 3, 5));

priorityQ.insert(Task(2, 1, 2));

priorityQ.insert(Task(3, 2, 3));

priorityQ.insert(Task(4, 1, 4));

priorityQ.insert(Task(5, 3, 1));

cout << "Highest priority task before extract min call" << endl;

priorityQ.peek().printTaskInfo();

priorityQ.extractMin();

cout << "\nHighest priority task after extract min call" << endl;

priorityQ.peek().printTaskInfo();

return 0;

}

void Task::printTaskInfo()

{

cout << "\nTask ID: " << id << endl;

cout << "Execution time: " << executionTime << endl;

cout << "Task priority: " << priority << endl;

}

/\*Question: How would you ensure that tasks with the highest

priority are executed first while maintaining the

integrity of the min-heap?\*/

/\*Ans: Task with highest priority will be exceted first

because we will be inserting the task will highest priority

at head.

\*/

void PriorityQueue::insertTask(Task \*newTask)

{

// if priority is high make it head;

if (head == nullptr || newTask->priority < head->priority)

{

newTask->next = head;

head = newTask;

return;

}

// else place according to its priority;

Task \*current = head;

while (current->next != nullptr && current->next->priority <= newTask->priority)

{

current = current->next;

}

newTask->next = current->next;

current->next = newTask;

}

void PriorityQueue::insert(Task task)

{

Task \*newTask = new Task(task.id, task.priority, task.executionTime);

insertTask(newTask);

}

/\*Question: How would you handle cases where

multiple tasks have the same priority level?\*/

/\*Ans: Task at head will be the one which will be excuted 1st

because while inserting if priority is same first come, first served

be applied\*/

Task PriorityQueue::extractMin()

{

if (head == nullptr)

{

cout << "Priority queue is empty" << endl;

}

Task temp = \*head;

Task \*newHead = head->next;

delete head;

head = newHead;

return temp;

}

Task PriorityQueue::peek()

{

if (head == nullptr)

{

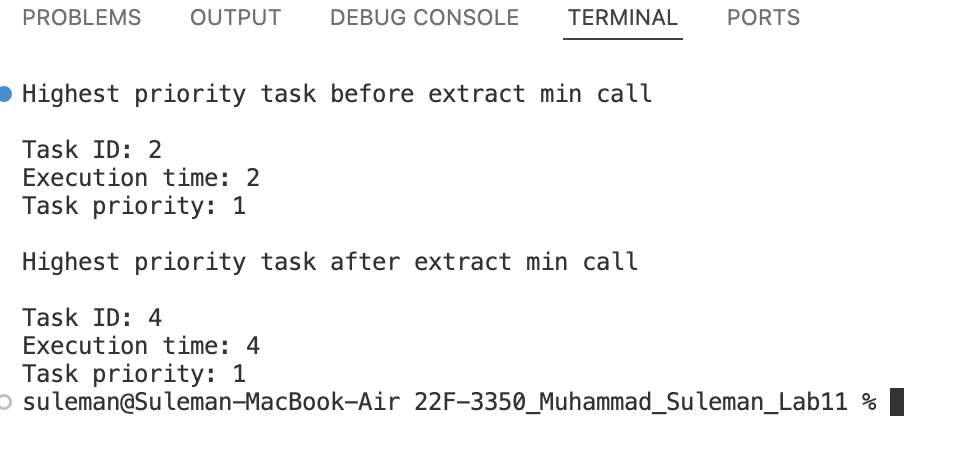
cout << "Priority queue is empty." << endl;

}

return \*head;

}

## Output



1. How would you ensure that tasks with the highest priority are executed first while maintaining the integrity of the min-heap?

Ans: Task with highest priority will be exceted first

because we will be inserting the task will highest priority

at head.

1. How would you handle cases where multiple tasks have the same priority level?

Ans: Task at head will be the one which will be excuted 1st

because while inserting if priority is same first come, first served

be applied