array

// Heap Sort in C

#include <stdio.h>

// Function to swap the the position of two elements

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

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

// Find largest among root, left child and right child

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

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

largest = left;

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

largest = right;

// Swap and continue heapifying if root is not largest

if (largest != i) {

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

heapify(arr, n, largest);

}

}

// Main function to do heap sort

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

// Build max heap

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

heapify(arr, n, i);

// Heap sort

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

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

// Heapify root element to get highest element at root again

heapify(arr, i, 0);

}

}

// Print an array

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

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

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

printf("\n");

}

// Driver code

int main() {

int arr[] = {1, 12, 9, 5, 6, 10};

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

heapSort(arr, n);

printf("Sorted array is \n");

printArray(arr, n);

}

Heap

#include <iostream>

#include <vector>

#include <algorithm>

#include <stdexcept>

using namespace std;

// Data structure for Max Heap

struct PriorityQueue

{

private:

// vector to store heap elements

vector<int> A;

// return parent of A[i]

// don't call this function if it is already a root node

int PARENT(int i)

{

return (i - 1) / 2;

}

// return left child of A[i]

int LEFT(int i)

{

return (2 \* i + 1);

}

// return right child of A[i]

int RIGHT(int i)

{

return (2 \* i + 2);

}

// Recursive Heapify-down algorithm

// the node at index i and its two direct children

// violates the heap property

void heapify\_down(int i)

{

// get left and right child of node at index i

int left = LEFT(i);

int right = RIGHT(i);

int largest = i;

// compare A[i] with its left and right child

// and find largest value

if (left < size() && A[left] > A[i])

largest = left;

if (right < size() && A[right] > A[largest])

largest = right;

// swap with child having greater value and

// call heapify-down on the child

if (largest != i) {

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

heapify\_down(largest);

}

}

// Recursive Heapify-up algorithm

void heapify\_up(int i)

{

// check if node at index i and its parent violates

// the heap property

if (i && A[PARENT(i)] < A[i])

{

// swap the two if heap property is violated

swap(A[i], A[PARENT(i)]);

// call Heapify-up on the parent

heapify\_up(PARENT(i));

}

}

public:

// return size of the heap

unsigned int size()

{

return A.size();

}

// function to check if heap is empty or not

bool empty()

{

return size() == 0;

}

// insert key into the heap

void push(int key)

{

// insert the new element to the end of the vector

A.push\_back(key);

// get element index and call heapify-up procedure

int index = size() - 1;

heapify\_up(index);

}

// function to remove element with highest priority (present at root)

void pop()

{

try {

// if heap has no elements, throw an exception

if (size() == 0)

throw out\_of\_range("Vector<X>::at() : "

"index is out of range(Heap underflow)");

// replace the root of the heap with the last element

// of the vector

A[0] = A.back();

A.pop\_back();

// call heapify-down on root node

heapify\_down(0);

}

// catch and print the exception

catch (const out\_of\_range& oor) {

cout << "\n" << oor.what();

}

}

// function to return element with highest priority (present at root)

int top()

{

try {

// if heap has no elements, throw an exception

if (size() == 0)

throw out\_of\_range("Vector<X>::at() : "

"index is out of range(Heap underflow)");

// else return the top (first) element

return A.at(0); // or return A[0];

}

// catch and print the exception

catch (const out\_of\_range& oor) {

cout << "\n" << oor.what();

}

}

};

int main()

{

PriorityQueue pq;

// Note - Priority is decided by element's value

pq.push(3);

pq.push(2);

pq.push(15);

cout << "Size is " << pq.size() << endl;

cout << pq.top() << " ";

pq.pop();

cout << pq.top() << " ";

pq.pop();

pq.push(5);

pq.push(4);

pq.push(45);

cout << endl << "Size is " << pq.size() << endl;

cout << pq.top() << " ";

pq.pop();

cout << pq.top() << " ";

pq.pop();

cout << pq.top() << " ";

pq.pop();

cout << pq.top() << " ";

pq.pop();

cout << endl << std::boolalpha << pq.empty();

pq.top(); // top operation on an empty heap

pq.pop(); // pop operation on an empty heap

return 0;

}