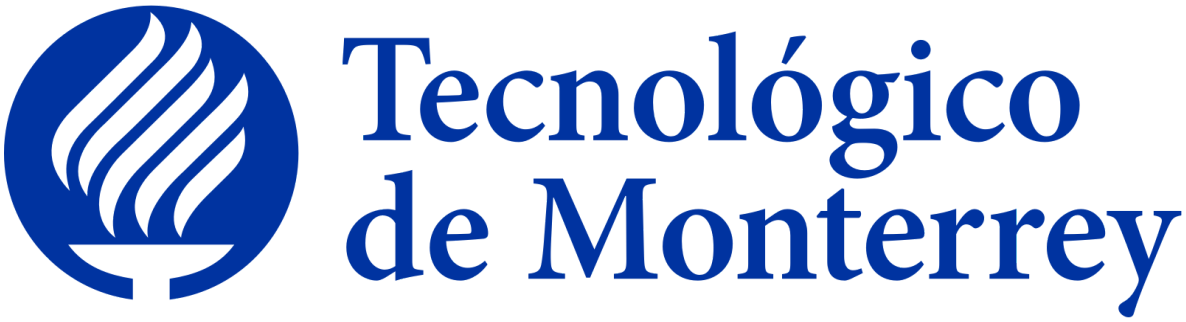
Instituto Tecnológico y de Estudios Superiores de Monterrey



Programación de Estructuras de Datos y Algoritmos Fundamentales

Ivan Reyez Amezcua

Act 3.2 - Árbol Heap: Implementando una fila priorizada

Santos Alejandro Arellano Olarte // A01643742
Carlos Iván Armenta Naranjo // A01643070
Arturo Ramos Martínez //A01643269
Adair Virgilio Figueroa Medina //A00572826
Leonardo Mario Alberto Guillén Soria // A00574110
Daniela Rocha Muñoz //A00573664

03 de Noviembre de 2023

Realiza en equipos una clase en C++ llamada "priority_queue" que tenga las siguientes operaciones utilizando un heap y simulando una fila priorizada de enteros con prioridad de valor mayor.

push	Descripción	Agregue un dato a la fila priorizada
	Entrada	Un entero, el cual se desea agregar a la fila priorizada
	Salida	Nada
	Precondición	n es un entero
	Postcondición	La fila priorizada contiene un nuevo dato.

pop	Descripción	Saca de la fila priorizada el dato que tiene mayor prioridad
	Entrada	Nada
	Salida	Nada

	Precondición	Que la fila priorizada contenga al menos 1 dato.
	Postcondición	La fila priorizada queda sin el dato con mayor prioridad

top	Descripción	Regresa el valor del dato que esta con mayor prioridad en la fila priorizada.(i*2+1)
	Entrada	Nada
	Salida	El dato que tiene mayor prioridad dentro de la Fila priorizada
	Precondición	Que la fila priorizada contenga al menos 1 dato.
	Postcondición	Nada

empty	Descripción	Regresa un valor boleando, diciendo si la fila priorizada está vacía o tiene datos.
	Entrada	Nada
	Salida	Un valor booleano que diga si la fila priorizada está vacía o tiene datos.
	Precondición	Nada.
	Postcondición	Nada

size	Descripción	Regresa la cantidad de datos que tiene la fila priorizada
	Entrada	Nada

	Salida	Un valor entero que representa la cantidad de datos de la fila priorizada
	Precondición	Nada.
	Postcondición	Nada

A continuación, la implementación y explicación de cada método desarrollado para el heap.

Cabe resaltar que el árbol binario está representado en un array; entonces, para acceder ya sea los nodos hijos o padres, usamos las siguientes fórmulas:

Para los nodos hijos: (i*2+1) y (i*2+2)

Para los nodos padres: (i-1)/2

Otra cosa importante, es que implementamos un max heap, el cual también puede ser conocido como priority queue, dependiendo de lo que se busque; además, esta manera de crear una priority queue presenta grandes ventajas de optimización en las complejidades e incluso simplifica la implementación.

Métodos u operaciones del max heap:

push	Descripción	Inserta un valor al max heap, y re ordena el heap si es necesario, para que en cada inserción se siga respetando que cada nodo sea mayor a sus nodos hijos.

	Entrada	un valor entero (int)
	Salida	nada
	Precondición	que el valor sea entero y que se haya creado el heap anteriormente
	Postcondición	Un valor nuevo en el heap
	Complejidad	Mejor de los casos O(1), ya que el elemento no tendría que hacer ningún swap. Peor de los casos O(log n), esto pasará cuando el nuevo elemento sea el más grande, por lo tanto tendrá que hacer swap con todos sus ancestros, o en otras palabras tendrá que hacer un número de swaps igual a la altura de un árbol, la cual en un árbol completo es log n.

pop	Descripción	Elimina el nodo raíz, o en este caso, el primer elemento del array. En pocas palabras, primeramente se verifica si efectivamente existe algún valor para eliminar, y si es así, primero le hace un swap con el último valor del heap, después se decrementa el size del heap y por último se implementa un algoritmo para ordenar los elementos y que así se respete que cada nodo sea mayor a sus nodos hijos.
	Entrada	nada

	Salida	nada
	Precondición	Que exista el heap
	Postcondición	El nodo raíz es eliminado, el heap ahora es un elemento más pequeño, y existe una nueva raíz o valor máximo.
	Complejidad	En el peor de los casos sería de O(log n), ya que si el último nodo tuviera que regresar a donde estaba antes de swap con el nodo raíz, tendría que hacer swaps con cada nodo hijo, y eso le tomaría la misma cantidad de swaps, que su altura misma, la cual es (log n)

top	Descripción	Retorna el elemento 0 del heap o del array, osea el elemento con mayor prioridad (en este caso el elemento más grande).
	Entrada	nada
	Salida	valor entero
	Precondición	Que exista el heap
	Postcondición	nada
	Complejidad	Constante O(1)

empty	Descripción	Verifica si el heap está o no vacío.
	Entrada	nada
	Salida	valor booleano “true” o “false”
	Precondición	Que exista el heap
	Postcondición	nada
	Complejidad	O(1)

size	Descripción	Nos indica el tamaño del heap, cabe resaltar que no toma en cuenta que las posiciones en el heap comienzan en 0, así que nos regresa el número de elementos en el heap, comenzando a contar desde el 1.
	Entrada	nada
	Salida	valor entero
	Precondición	Que exista el heap
	Postcondición	nada

	Complejidad	Complejidad constante O(1), ya que solo accede y devuelve la variable que lleva el conteo de elementos en el heap.

heapify	Descripción	Decidimos agregar una forma extra de crear el heap, y esta se hace a partir de un array ya creado.
	Entrada	nada
	Salida	nada
	Precondición	Que se haya creado una instancia u objeto de la clase Maxheap
	Postcondición	El array que se le pasó al constructor de la clase, ahora es un max heap.
	Complejidad	O(n)

Código:

```
#include <iostream>

using namespace std;


#define SIZE 12


// Clase Maxheap que define un heap máximo

class Maxheap{
```

```
    public:
        int* array; // Array para almacenar los elementos del heap
        int capacity; // Capacidad máxima del heap
        int heap_size; // Tamaño actual del heap
        Maxheap(); // Constructor por defecto
        Maxheap(int*, int); // Constructor que toma un array y su
tamaño
        bool empty(); // Método para verificar si el heap está vacío
        void push(int); // Método para agregar un elemento al heap
        void heapify(); // Método para convertir un array en un heap
        void pop(); // Método para eliminar el elemento máximo del heap
        int top(); // Método para obtener el elemento máximo del heap
        int size(); // Método para obtener el tamaño del heap
        void display(); // Método para mostrar los elementos del heap
};


// Este es el constructor por defecto, con una complejidad de 0(1)
Maxheap::Maxheap(){
    capacity = SIZE;
}


// Constructor que toma un array y su tamaño, la complejidad es 0(1)
Maxheap::Maxheap(int* array, int size){
    this->array = array;
    capacity = SIZE;
    heap_size = size;
}


/* Método para verificar si el heap está vacío, su complejidad es 0(1)
ya que solo revisa un valor a la variable*/
bool Maxheap::empty(){
    return heap_size == 0;
}
```

```
/* Método para agregar un elemento al heap con una complejidad de 0(log
n) */

void Maxheap::push(int new_value){

    if(heap_size == capacity){

        cout << "No hay espacio para más elementos" << endl;

        return;

    }

    heap_size ++;

    int i  = heap_size-1;

    array[i] = new_value;

    while(i != 0 && array[i]>array[(i-1)/2]){

        swap(array[i], array[(i-1)/2]);

        i = (i-1)/2;

    }

}



// Método para eliminar el elemento máximo del heap, su complejidad es
0(log n)

void Maxheap::pop(){

    if(heap_size == 0){

        cout << "Heap vacío" << endl;

        return;

    }

    int deleted = array[0];

    swap(array[0], array[heap_size-1]);

    heap_size--;

    int i = 0;

    int max_index;

    while((i*2+2 <= heap_size-1) && (array[i] < array[i*2+1] ||
array[i] < array[i*2+2])){

        max_index = i*2+2;

        if(array[i*2+1]>array[i*2+2]){

            max_index = i*2+1;

        }
```

```
        swap(array[i], array[max_index]);

        i = max_index;

    }

    cout << "Se eliminó el: " <<deleted<<endl;

}



// Método para obtener el elemento máximo del heap, con su complejidad
de O(1) ya que solo devuelve el valor de la raíz del arbol

int Maxheap::top(){

    if(empty()){

        cout << "Heap vacío" << endl;

        return -1;

    }

    return array[0];

}



// Método para obtener el tamaño del heap, con un valor O(1)

int Maxheap::size(){

    return heap_size;

}



/*Este método convierte un array en un heap. La complejidad de tiempo
es O(n) porque en el peor de los casos, puede tener que "hundir" cada
elemento del array.*/

void Maxheap::heapify(){

    if(empty()){

        cout << "Array vacío" << endl;

        return;

    }

    int num_iteraciones = heap_size/2;

    int max_index;

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

    while((i*2+2 <= heap_size-1) && (array[i] < array[i*2+1] ||
array[i] < array[i*2+2])){

        max_index = i*2+2;
```

```
        if(array[i*2+1]>array[i*2+2]){
            max_index = i*2+1;
        }
        swap(array[i], array[max_index]);
        i = max_index;
    }
    }
}


// Método para mostrar los elementos del heap, con una complejidad de
O(n).
void Maxheap::display(){
    cout << "[";
    for (int i = 0; i < heap_size; i++){
        cout << array[i] << ", " ;
    }
    cout << "]" << endl;
}


// Función principal
int main() {
    int eleccion;
    int eleccion2;
    int emArray[SIZE];
    int Array[SIZE] = {30,3,14,55,5,7,11,1,20,4,88,2};
    Maxheap* heap;
    cout << "Max heap" << endl;
    cout << "Elige entre crear un max heap desde cero, agregando
elementos uno a uno o \ntransforma un array con valores ya establecidos
en un max heap" << endl;
    cout << "Crear max heap uno a uno. teclea 1" << endl;
    cout << "Crear max heap a partir de un array. teclea 2" << endl;
    cin >> eleccion;
```

```
// Crear un heap a partir de la elección del usuario
if(eleccion == 1){
    heap = new Maxheap(emArray,0);
}
else if(eleccion == 2){
    heap = new Maxheap(Array,SIZE);
    heap->heapify();
}

// Menú para realizar operaciones en el heap
do{
cout << "------------------" << endl;
cout << "Max heap métodos" << endl;
cout << "push. 1" << endl;
cout << "pop. 2" << endl;
cout << "top. 3" << endl;
cout << "size. 4" << endl;
cout << "display. 5" << endl;
cout << "salir. 6" << endl;
cout << "------------------" << endl;
cin >> eleccion2;
switch (eleccion2)
{
case 1:
    int value;
    cout << "Valor (int): " << endl;
    cin >> value;
    heap->push(value);
    break;
case 2:
    heap->pop();
    break;
```

```
    case 3:
        cout <<"Max: "<< heap->top() << endl;
    break;
    case 4:
        cout <<"Size: "<<heap->size() << endl;
    break;
    case 5:
        heap->display();
    break;
    default:
        break;
    }
    }while(eleccion2 != 6);
    return 0;
}
```