



Tecnológico de Monterrey

Documentación de código Heap y Priority Queues

Programming of data structures

Inés Alejandro Garcia Mosqueda A00834571

Profesores: Luis Ricardo Peña Llamas
Jorge Gonzalez

17/10/2022

En la implementación de esta estructura de datos se crea la clase nodo, la cual contiene 4 punteros para el enlace con otros nodos y formar la estructura de datos, se esta pidiendo una implementación de priority queues con heaps tres, por lo cual para gozar de los privilegios de estas estructuras, se crean punteros para los hijos de un nodo (left, right) para la estructura de un árbol heap, los punteros a los nodos adyacentes en la queue y aparte contiene variables para guardar los datos de cada nodo.

El nodo contiene métodos que pueden modificar los datos del nodo, así como obtenerlos.

```
class Node{
public:
    //Data Nodo
    int value;
    int priority;
    //Accesos
    Node* left;
    Node* right;
    Node* next;
    Node* previous;
    Node(){
        left=NULL;
        right=NULL;
        next=NULL;
        previous=NULL;
        priority=0;
        value=0;
    }
    Node(int value, int priority){
        left=NULL;
        right=NULL;
        next=NULL;
        previous=NULL;
        this -> value = value;
        this -> priority = priority;
    }
};
```

La estructura de la estructura principalmente solo contiene un puntero a un nodo cabeza y un nodo cola, los cuales nos sirven para obtener los datos ya sea mas altos o mas bajos en la estructura

```
class PriorityQueue
{
    int len;
    Node *head;
    Node *tail;
private:
```

Para la implementación de los métodos se crearon dos clases extras, los cuales son una estructura en forma de Stack, el cual va guardando rutas de un elemento para realizar las búsquedas necesarias y a su vez se crearon nodos, los cuales son el elemento básico para el funcionamiento de nuestro stack de direcciones.

```
//Nodo Direccion
class NodeDir{
public:
    //Ruta
    int index;
    int direction;
    //Enlace
    NodeDir* next = NULL;
    NodeDir(int index, int direction){
        this->index = index;
        this->direction = direction;
        next=NULL;
    }
    ~NodeDir(){
        //delete next;
    }
};
```

```
//Stack directorio
class Directory{
public:
    NodeDir* head;
    int size;
    Directory(){
        head = NULL;
        size = -1;
    };
};
```

Estas estructuras de datos fueran elegidas debido a que la forma de su implementación será de complejidad constante, ya que queremos un sistema de first in fist out, lo cual es lo más óptimo para nosotros debido a que empezaremos a deducir la ruta de un nodo en el Heap mediante el índice de este en la priority queue

```
NodeDir pop(){
    if (size >= 0)
    {
        NodeDir *tmp = head;
        head = head->next;
        size--;
        return *tmp;
    }
    return NodeDir(0,0);
};

NodeDir top(){
    if (size >= 0)
    {
        return *head;
    }
    return NodeDir(0,0);
};

bool empty(){
    if (size <= -1)
        return true;
    else
        return false;
};
};
```

En la practica se piden las funciones, push, pop, top, empty y size

Para la función Pop se realizaron funciones aparte de esta para poder dividir la función en pedazos más chicos y poder reutilizar algunos procesos que este conlleva

```
void push(int index, int direction){
    size++;
    NodeDir* newNode = new NodeDir(index,direction);
    NodeDir* tmp = head;
    head = newNode;
    head->next = tmp;
};
```

En esta función se tiene que realizar un directorio de rutas para identificar a que nodos se le va a agregar como rama, esto lo realiza la función searchChildFather, la cual agrega las rutas necesarias desde el header, para llegar al padre del nodo recién insertado.

```
void searchChildFather(int index, Directory &directory){
    int fatherIndex, fatherDirection;
    if (index == 0){
        fatherIndex = 0;
        fatherDirection = 0;
        return;
    }
    else
    {
        double resDouble = (index-1)/2.0;
        int resInt = (index-1)/2;
        if (resDouble > resInt)
            fatherDirection = 1;
        else
            fatherDirection = -1;

        fatherIndex = resInt;
        directory.push(fatherIndex, fatherDirection);
    }
}
```

Consecutivo a esto se realiza la búsqueda del nodo a partir de las rutas del nodo padre y agregarle el puntero hijo (nodo recién agregado)

```

Node* accesNode(Node* node, Directory &directory){
    if (directory.size < 1 || node == NULL)
    {
        return node;
    }
    else
    {
        Node* tmp = NULL;
        NodeDir NodeDirection = directory.pop();
        if (NodeDirection.direction == -1)
            tmp = node->left;
        else if(NodeDirection.direction == 1)
            tmp = node->right;
        return accesNode(tmp, directory);
    }
}

```

Finalmente se ejecuta la función bubbleUp, la cual eleva el nodo en cuestión de los punteros dentro del Heap a su posición correspondiente, haciendo un switch de nodos entre los nodos que se van recorriendo

```

void swap(Node* &node1, Node* &node2){
    int valueAux = node1->value;
    int priorityAux = node1->priority;
    node1->value = node2->value;
    node1->priority = node2->priority;
    node2->value = valueAux;
    node2->priority = priorityAux;
}

```

```

void bubbleUp(int index, Node* &node){
    Node* tmp;
    Directory fathersDir;
    searchChildFather(index, fathersDir);
    tmp = maxAncestor(head, fathersDir, node);
    swap(node, tmp);
    bubbleDown(node);
}

```

Este método (bubble up) a partir de la ruta calculada recorre los nodos por altura comparándolos con el valor de prioridad del nodo que se insertó y de manera recursiva va intercambiando los nodos mayores con los nodos menores, de manera que la estructura Heap mantenga sus propiedades

```

void elevate(Directory directory, Node* &node1, Node* &node2){
    if (directory.size < 0 || node1 == NULL)
    {
        return;
    }
    else
    {
        if (node1->value > node2->value)
        {
            swap(node1, node2);
        }
        Node* tmp = NULL;
        NodeDir NodeDirection = directory.pop();
        if (NodeDirection.direction == -1)
            tmp = node1->left;
        else if (NodeDirection.direction == 1)
            tmp = node1->right;
        return elevate(directory, tmp, node2);
    }
}

```

Este algoritmo al no tener que recorrer el árbol para la inserción de un nodo se calcula una **complejidad $O(\log n)$** , ya que se conoce la ruta de inserción del nodo y se recorre el Heap de forma vertical

Por otro lado, la creación del árbol implicaría una **complejidad $O(n \log)$** (función fillQueue para cargar el HeapPriorityQueue)

```
void fillQueue(string fileName){
    ifstream file;
    file.open(fileName);
    int value, priority;
    while (file >> value >> priority)
    {
        push(value, priority);
    }
    file.close();
};
```

La función pop se encarga de borrar un nodo en un índice dado, se calcula la ruta de dicho nodo en dicho índice, se accede al nodo y se intercambia por la cola de la Priority Queue, de esta forma se pueden recorrer sus hijos hasta encontrar el espacio indicado para mantener la propiedad del Heap

```
void pop(int index){
    Node* tmp;
    Directory fathersDir;
    searchChildFather(index, fathersDir);
    tmp = accesNode(head, fathersDir);
    swap(tmp, tail);
    bubbleDown(tmp);
    tmp = tail->previous;
    delete tail;
    tail = tmp;
    tail->next=NULL;
    len--;
};
```

La función que se encarga de mantener la propiedad del Heap es la función bubble down


```

void bubbleDown(Node* node){
    if (node->left == NULL)
    {
        return;
    }
    else if (node->left->value > node->value && node->right == NULL)
    {
        swap(node, node->left);
    }
    else if (node->left->value < node->value && node->right == NULL){
        return;
    }
    else if (node->left->value > node->value && node->right->value > node->value)
    {
        return;
    }
    else
    {
        if (node->left->value < node->right->value)
        {
            swap(node, node->left);
            bubbleDown(node->left);
        }
        else if (node->left->value > node->right->value)
        {
            swap(node, node->right);
            bubbleDown(node->right);
        }
        bubbleDown(node->left);
        return;
    }
}

```

Esta última es una función recursiva que recorre verticalmente el Heap para ordenar los datos debido a la eliminación de un nodo de un subtree

Debido a que para la eliminación del nodo se calcula la ruta para acceder mediante las rutas del Heap, la **complejidad** para la función pop es **$O(\log n)$** , ya que no tiene que recorrer el árbol para encontrar ordenar y encontrar el nodo

Por otro lado, la función Top, Size y Empty se pueden considerar de **complejidad $O(1)$** ya que por las propiedades del Queue el acceder a la cola o a la cabeza de la estructura es constante, además que para el funcionamiento de la estructura se va registrando por cada acción los elementos contenidos en la estructura

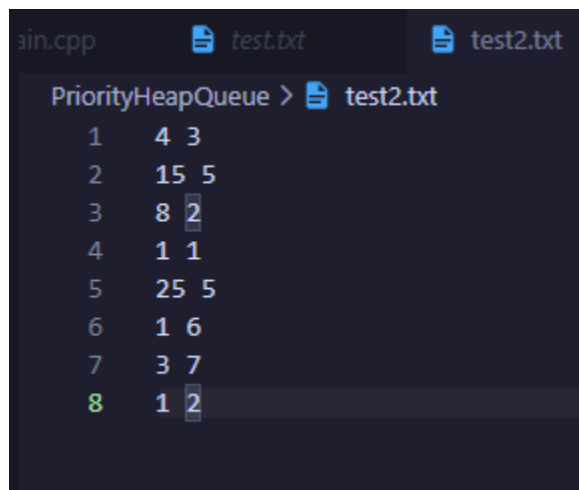
```
Node top(){
    if(isEmpty())
    {
        throw invalid_argument("heap vacio");
    }
    return *head;
};
```

```
bool isEmpty(){
    if (len == -1)
        return true;
    else
        return false;
};

int size(){
    return len+1;
};
```

```
int size(){
    return len+1;
};
```

Finalmente se tiene un caso de prueba con la implementación de esta estructura



```
main.cpp test.txt test2.txt
PriorityHeapQueue > test2.txt
1 4 3
2 15 5
3 8 2
4 1 1
5 25 5
6 1 6
7 3 7
8 1 2
```

```

int main(){
    PriorityQueue pq;
    cout<<"\nLlenado de estructura "<<endl;
    pq.fillQueue("test2.txt");
    pq.print();
    cout<<"\nEstructura con tamaño de "<< pq.size() << " elementos\n" <<endl;
    cout<<"\npop de dato en índice 5 "<<endl;
    pq.pop(5);
    cout<<endl;
    pq.print();
    cout<<"\npush de dato con prioridad 7 y data 5 "<<endl;
    pq.push(7, 5);
    cout<<endl;
    pq.print();
    cout<<"\nPrioridad del nodo en cabeza "<<pq.top().value;
    return 0;
}

```

Obteniendo como resultado lo siguiente

```

Llenado de estructura
( 1, 1 )      ( 1, 2 )      ( 1, 6 )      ( 4, 3 )      ( 25, 5 )      ( 8, 2 )      ( 3, 7 )      ( 15, 5 )
Estructura con tamaño de 8 elementos

pop de dato en índice 5

( 1, 1 )      ( 1, 2 )      ( 3, 7 )      ( 4, 3 )      ( 25, 5 )      ( 8, 2 )      ( 15, 5 )
push de dato con prioridad 7 y data 5

( 1, 1 )      ( 1, 2 )      ( 3, 7 )      ( 4, 3 )      ( 25, 5 )      ( 8, 2 )      ( 15, 5 )      ( 7, 5 )
Prioridad del nodo en cabeza 1

```

```

  ✓ pq: {...}
    len: 7
    ✓ head: 0x20cec983f70
      value: 1
      priority: 1
    ✓ left: 0x20cecba3820
      value: 1
      priority: 2
    ✓ left: 0x20cecba38e0
      value: 4
      priority: 3
    ✓ left: 0x20cecba3b60
      value: 7
      priority: 5
      > left: 0x0
      > right: 0x0
      > next: 0x0
      > previous: 0x20cecba3ac0
      > right: 0x20cecba3b60
      > next: 0x20cecba3980
      > previous: 0x20cecba3880
      > right: 0x20cecba3980
      > next: 0x20cecba3880
      > previous: 0x20cec983f70
      > right: 0x20cecba3880
      > next: 0x20cecba3820
      > previous: 0x0

```

Y si notamos el dato esta insertado correctamente, ya que se crea el siguiente nivel y se inserta debajo del nodo con prioridad 4

```

( 1, 1 )    ( 1, 2 )    ( 3, 7 )    ( 4, 3 )    ( 25, 5 )    ( 8, 2 )    ( 15, 5 )
( 1, 1 )    ( 1, 2 )    ( 3, 7 )    ( 4, 3 )    ( 25, 5 )    ( 8, 2 )    ( 15, 5 )    ( 7, 5 )

```