César Alán Silva Ramos - A01252916

Act 3.2 - Árbol Heap: Implementando una fila priorizada

Repositorio en Github
Casos Prueba

- **Push y Pop**

```
// Agregamos elementos
    pq.push(299);
    pq.push(792);
    pq.push(420);
    pq.push(2);
// Sacamos el elemento de mayor prioridad
    pq.pop();
```

Se agregaron 4 elementos a la fila y se utilizó pop al final, lo cual saca al elemento de mayor prioridad, en este caso al 792 por lo tanto se obtuvo lo siguiente:

```
Elementos en la fila priorizada: 420 299 2
```

- **Top**

```
// Se devuelve el elemento con mayor prioridad
    std::cout << "El elemento con mayor prioridad es: " << pq.top() << std::endl;
```

Top devuelve el dato que tiene mayor prioridad dentro de la fila, como anteriormente se había quitado el 792 se obtuvo lo siguiente:

```
El elemento con mayor prioridad es: 420
```

- **Empty**
  Se aplicó la función isEmpty en dos casos distintos: uno en el cual la fila no estaba vacía y el otro en el que la fila sí estaba vacía.
  Siguiendo con el caso prueba que se planteó desde el inicio se obtuvo:

```
La fila no esta vacía
```

  Después se utilizó la función pop() para dejar vacía la lista

```
// Agregamos elementos
    pq.push(299);
    pq.push(792);
    pq.push(420);
    pq.push(2);

    pq.pop();
    pq.pop();
    pq.pop();
    pq.pop();
```

  Se obtuvo el siguiente output:

```
La lista esta vacía
```

- Size

```
Existen 3 datos en la fila priorizada
```

<u>Complejidades computacionales</u>

Push

```
oid push(int value) {
heap.push_back(value);// se mete el nuevo valor
int index = heap.size() - 1; // se calcula el index
while (index > 0) {
```

```
int parentIndex = (index - 1) / 2; // se calcual el index mayor
if (heap[index] > heap[parentIndex]) { // si el index actual es mayor que el index
padre entonces swap
std::swap(heap[index], heap[parentIndex]);
index = parentIndex; // y el index actual se vuelve el nuevo padre
} else {
break;
}
}
}
```

La complejidad temporal de la operación push es O(log n), donde "n" representa la cantidad de elementos presentes en el heap. Aunque la inserción de un elemento en sí misma solo requiere una operación constante de O(1) en las estructuras de filas priorizadas, es fundamental recordar que la prioridad de los elementos debe ser mantenida, lo que involucra comparaciones y, en ocasiones, intercambios con el elemento padre. La cantidad de iteraciones en el bucle está limitada por la cantidad de elementos en el heap y sigue una complejidad logarítmica en función de estos. Conforme el tamaño del heap crece, la naturaleza logarítmica de la operación garantiza que siga siendo eficiente en términos de tiempo de ejecución.

Pop

```
int pop() {
if (isEmpty()) {
throw std::runtime_error("La lista priorizada esta vacía");
}

int top = heap[0];
heap[0] = heap.back();
heap.pop_back();

// Se checan los elementos para reordenar

int index = 0;
while (true) {
int leftChild = 2 * index + 1;
int rightChild = 2 * index + 2;
```

```
int largest = index;

if (leftChild < heap.size() && heap[leftChild] > heap[largest]) {
largest = leftChild;
}

if (rightChild < heap.size() && heap[rightChild] > heap[largest]) {
largest = rightChild;
}

if (largest != index) {
std::swap(heap[index], heap[largest]);
index = largest;
} else {
break;
}
}

return top;
}
```

Esta función es similar a la anterior, con la diferencia de que aquí en lugar de insertar un elemento, se extrae el elemento con la mayor prioridad. A pesar de que la eliminación de un elemento es una operación constante, se requieren comparaciones y reordenamiento a través del bucle while para mantener la jerarquía de prioridad. Este bucle se ejecuta hasta que el elemento extraído ocupa su posición correcta en relación a los demás elementos, lo que resulta en una complejidad temporal de O(log n) en el peor de los casos.

Top

```
int top() const {
if (isEmpty()) {
throw std::runtime_error("La lista priorizada esta vacía");
}
return heap[0]; // se regresa el primer elemento
}
```

La función top presenta una complejidad temporal constante de O(1). Esto se debe a que la función simplemente accede al elemento con la mayor prioridad, el cual ya está ordenado en el heap. No se realizan operaciones adicionales en esta función. La razón detrás de esta complejidad constante es que el acceso a un elemento en un vector dinámico tiene un tiempo de ejecución que siempre es el mismo, sin importar cuántos elementos haya en el vector. En otras palabras, esta es una operación constante en términos de tiempo, lo que la hace eficiente independientemente del número de elementos presentes en el heap.

Empty

```
bool isEmpty() const {
return heap.empty(); // se checa si esta vacía con función
}
```

La función isEmpty tiene una complejidad temporal constante de O(1). Esto se debe a que la función verifica de manera directa si el heap está vacío utilizando la función .empty(). La función .empty() actualiza automáticamente un valor que refleja el estado de la estructura de datos, y este valor se obtiene de manera inmediata sin necesidad de cálculos adicionales. En otras palabras, el rendimiento de esta función es constante y no depende de la cantidad de elementos presentes en el heap, lo que la hace eficiente independientemente del tamaño del heap.

Size

```
size_t size() const {
return heap.size(); // se checa el tamaño con función
}
```

La función size posee una complejidad temporal constante de O(1). En esta función, se obtiene el tamaño del heap utilizando la función .size() . La razón de esta complejidad constante está en que el contenedor mantiene un registro actualizado en tiempo real del número de elementos presentes, y al llamar a esta función, se accede directamente a dicho registro. En consecuencia, el tiempo requerido para determinar el tamaño del heap es constante y no varía en función de la cantidad de elementos contenidos en la estructura de datos.