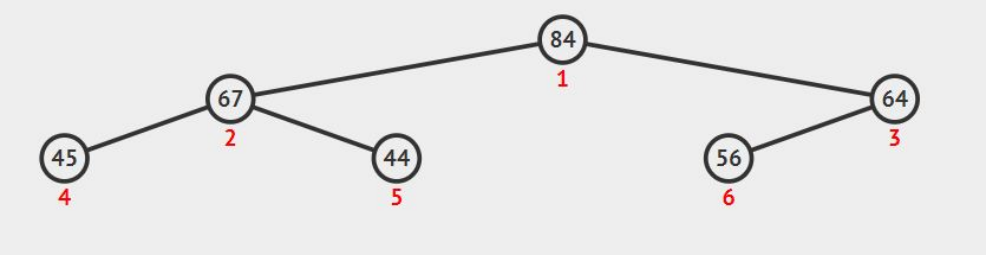
Para esta ocasión, hicimos uso del concepto de ‘Priority Queue’ que también es conocido como Heap. La manera en la que funciona, es que los datos introducidos a nuestra lista tendrán prioridad con base a su valor. Aunque en realidad estamos trabajando con una lista, se piensa como si fuese un árbol. Por ejemplo:



Este árbol es simplemente una representación de una lista que va de la siguiente manera: 84, 67, 64, 45, 44, 56. La manera en la que se piensa el árbol, es que se intenta mantenerlo balanceado, de tal manera que ningún lado tenga dos niveles más que el otro. Entonces la manera en la que “recorremos” este árbol es usando los índices. Cada índice será padre de su mismo índice dos veces y su mismo índice dos veces más uno. El índice 1 (84), es padre del índice 2 (67) y 3 (64). El índice 2 (67) es padre del índice 4 (45) y 5 (44), y así sucesivamente. Por tanto, la manera en la que insertamos y eliminaremos datos tendrá un orden específico.

FUNCIÓN DE INSERTAR:

La manera en la que se inserta un dato es simple. Se pone el dato al final de la fila, y este dato va subiendo de prioridad si es mayor que su padre. Por lo tanto, obtenemos el siguiente código:

```
template<class T>
void LinkedList<T> :: insert(T data) {
    if(isEmpty()) {
        head = new Node<T>(data, head);
        tail = head;
        size++;
    }
    else {
        addLast(data);
        int i = size;
        Node<T>* temp = tail;
        Node<T>* parent = getPos(i/2);
        while(parent->data < temp->data && i > 1) {
            swap(parent->data, temp->data);
            i = i/2;
            temp = parent;
            parent = getPos(i/2);
        }
    }
}
```

Primero tendremos que verificar que la lista no esté vacía. En caso de estar vacía, el dato es simplemente insertado y ambos ‘head’ y ‘tail’ (cabeza y cola) apuntan a dicho dato. En caso contrario, llevaremos a cabo un corto algoritmo. Primero agregamos el valor al final de la fila con la función “addLast”. Luego creamos un Nodo apuntador padre que será la división entera de su índice sobre 2. En caso de ser mayor el dato que el padre, cambiaremos sus valores con la función “swap”. En caso de no serlo, significa que el dato ha sido acomodado y que hemos terminado. Al simplemente ser un ciclo singular, este algoritmo tiene una complejidad de O(n).

FUNCIÓN DE ELIMINAR:

Para eliminar, guardaremos el valor del dato que está en la primera posición y lo eliminaremos de la lista. En su lugar pondremos el último valor de la lista. Por ejemplo, si mi lista es 84, 67, 64, 45, 44, 56, guardaremos el valor de 84 aparte y pondremos la última posición primero, dándonos la siguiente lista: 56, 67, 64, 45, 44. Después, tendremos que reacomodar la primera posición, básicamente hacer un insert pero desde arriba para abajo. Iremos comparando con los hijos y si uno de estos es mayor, cambiaremos de posición con él.

```
template<class T>
T LinkedList<T> :: remove() {
    if(size == 1) {
        T value = head->data;
        head = NULL;
        tail = NULL;
        return value;
    }
    else if(!isEmpty()) {
        T value = head->data;
        head->data = popBack();
        bool doing = true;
        int pos = 1;
        Node<T>* left;
        Node<T>* right;
        Node<T>* temp = head;
```

Esta es la primera parte del código. En caso de solo haber un valor en la lista, lo devolvemos y luego borramos la lista. En caso contrario, usaremos el siguiente algoritmo. Primero guardaremos el valor principal en nuestra variable “value”, la cual usaremos para retornar el valor más adelante. Luego haremos uso de la función “popBack” para eliminar el último valor y asignarle el valor al principio. Luego crearemos unos apuntadores con los que nos dirigiremos al hijo izquierdo y al derecho.

```
while(doing) {
    if(pos * 2 <= size) { // Existe hijo izquierdo
        left = getPos(pos*2);
        if(pos * 2 + 1 <= size) { // Existe hijo derecho
            right = getPos(pos*2+1);
            if(left->data >= right->data && left->data > temp->data) { // Hijo izquierdo mayor que derecho y mayor que dato
                swap(left->data, temp->data);
                temp = left;
                pos = pos * 2;
            }
            else if(right->data >= left->data && right->data > temp->data) { // Hijo derecho mayor que izquierdo y mayor que dato
                swap(right->data, temp->data);
                temp = right;
                pos = pos * 2 + 1;
            }
            else { // Ningun hijo es mayor que el dato
                doing = false;
            }
        }
        else {
            // Solo existe hijo izquierdo
            if(left->data > temp->data) { // Hijo izquierdo mayor que dato
                swap(left->data, temp->data);
                temp = left;
                pos = pos * 2;
            }
            else { // Solo existe un hijo y no es mayor
                doing = false;
            }
        }
    }
    else {
        // No tiene hijos
        doing = false;
    }
}
return value;
```

Como podemos ver, los casos terminan siendo más específicos. Los comentarios colocados nos ayudarán a entender que es cada caso. La manera en la que checamos si existe un hijo es la siguiente. Si la posición dos veces es menor o igual que el tamaño de la lista, entonces tiene un hijo a la izquierda. Si la posición dos veces más uno es menor o igual que el tamaño de la lista, entonces también tiene un hijo a la derecha. Si tiene ambos hijos y uno o ambos son mayores que nuestro dato, se hará una sustitución con el objetivo de pasar el dato con mayor prioridad hacia arriba. En caso de no tener ningún dato mayor, significa que ya hemos acabado de posicionar nuestro dato. Al final haremos un ‘return’ de nuestro dato mayor del principio, el cual nos servirá para ordenar la lista.

FUNCIÓN SORT:

Como mencionamos anteriormente, nuestra función de eliminar nos devuelve en todo caso el dato más grande de la lista. Haciendo uso de este conocimiento, podemos crear un vector en el cual tendremos todos los datos ordenados. Tendremos dos opciones entonces. En caso de querer ordenar la lista de mayor a menor, haremos un ciclo en el que se le está agregando datos al final de la cola. En caso de querer ordenar la lista de menor a mayor, haremos un ciclo en el que se le está agregando datos al principio de la cola.

```cpp
template<class T>
vector<T> LinkedList<T> :: sort() {
    vector<T> v;
    int n = size;
    for(int i = 0; i < n; i++) {
        v.insert(v.begin(), remove());
    }
    return v;
}
```

```cpp
template<class T>
vector<T> LinkedList<T> :: sortReverse() {
    vector<T> v;
    int n = size;
    for(int i = 0; i < n; i++) {
        v.push_back(remove());
    }
    return v;
}
```

REFLEXIÓN:

Al principio, tenía cierto miedo de no ser capaz de poder trabajar con este tipo de ordenamiento. Sin embargo, gracias a todos los ordenamientos que habíamos visto previamente, resolver esta problemática terminó siendo más simple de lo que yo pensaba. Termine teniendo unos problemas pero mi algoritmo funciona correctamente, simplemente me falto poner un ‘else’ para evitar que en ciertos casos termine de ordenar antes de tiempo. Una de las áreas que más puede llegar a interesar en el sector de Tecnologías Computacionales termina siendo el de ciberseguridad, por lo que esta actividad nos deja ver el potencial que tenemos como ITC para poder detectar si hubo un ataque DDoS, por ejemplo. Al final termine con con mucha satisfacción ya que resolver estos codigos ayuda a mejorar el pensamiento computacional de uno y lo ayuda a incrementar sus conocimientos, volviendolo entonces en un mejor programador.