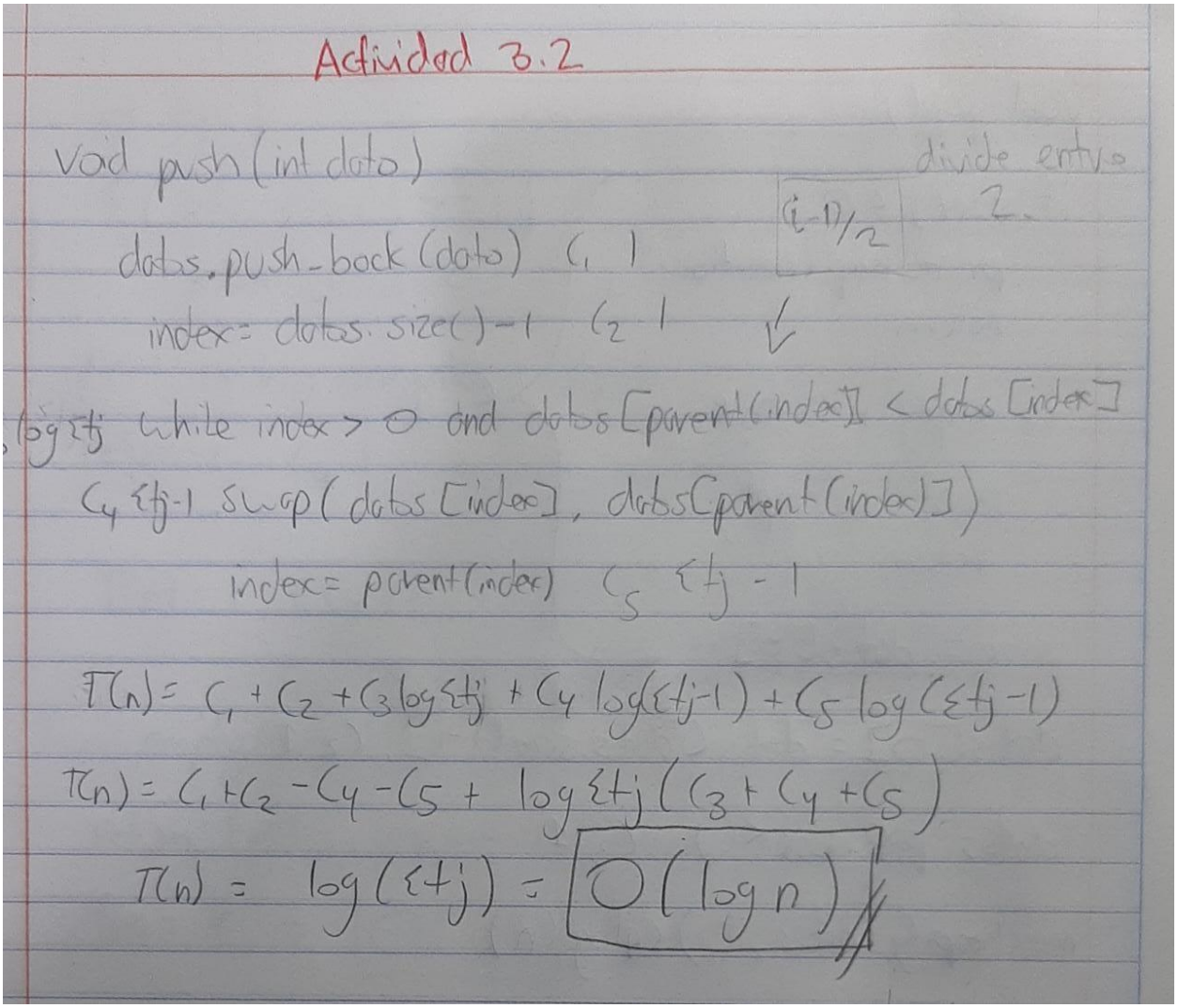
Jorge Del Barco Garza
A01284234

20 Oct 2021

Actividad 3.2 – Árbol Heap: Implementando una fila priorizada

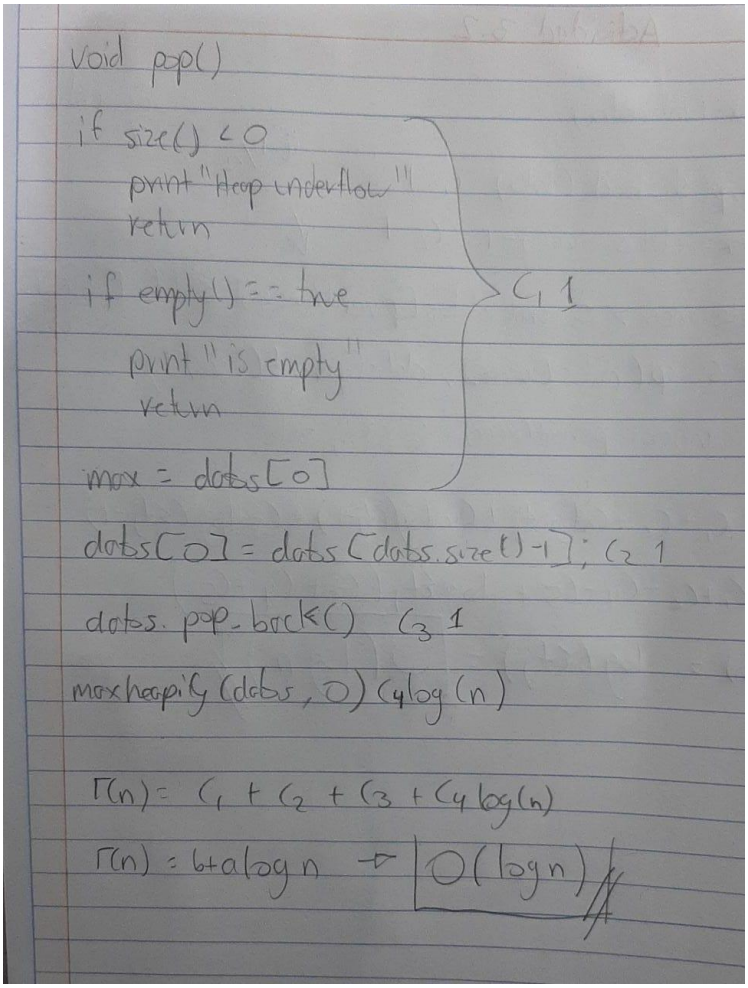
Push

```
int parent(int i){ return (i - 1) / 2; }

/*Agregue un dato a la fila priorizada*/
void priority_queue::push(int dato){

    datos.push_back(dato); // Se hace un append a la lista de datos
    int index = datos.size() - 1;

    while(index > 0 && datos[parent(index)] < datos[index]){
        // Cambiamos los elementos de posición.
        swap(datos[index], datos[parent(index)]);

        index = parent(index); // la POSICIÓN del padre pasa a ser nodo hijo. Es su posicion correcta
    }
}
```



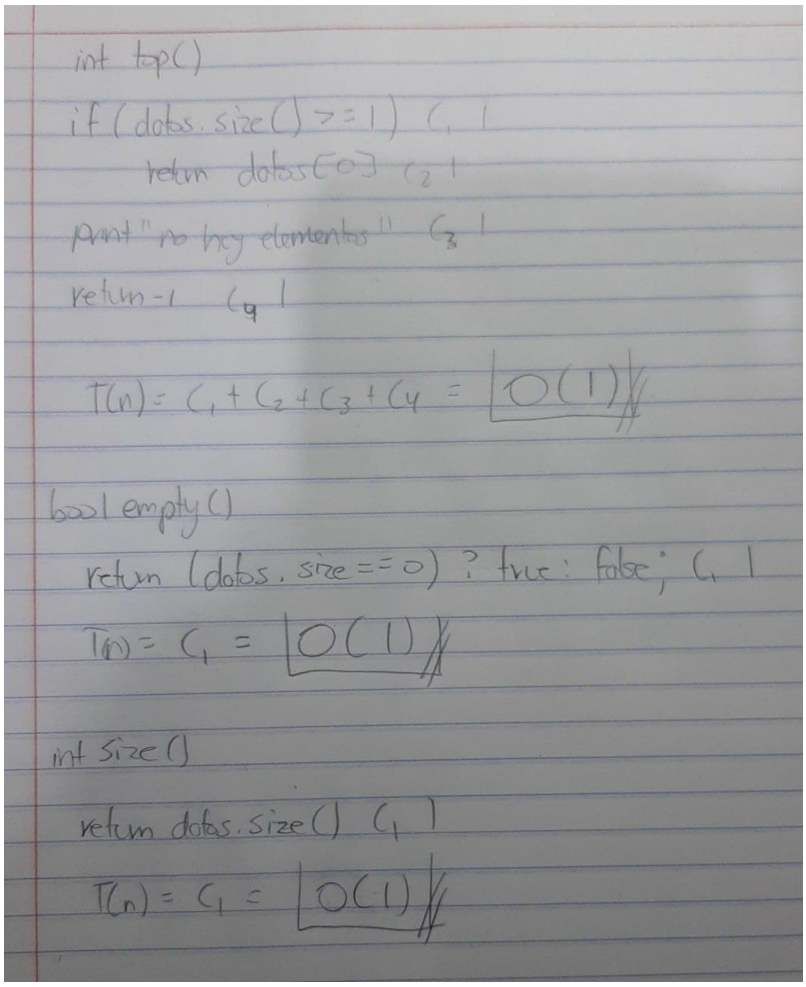
Pop

```cpp
void priority_queue::pop(){

    //cout << heapExtractMax(datos) << endl;

    if (size() < 0){
        cout << "Heap underflow" << endl;
        return;
    }

    if (empty() == true){
        cout << "La lista ya esta vacia" << endl;
        return;
    }

    // El más grande está siempre al inicio.
    int max = datos[0];

    // el último (menor valor) pasalo al frente para reorganizar todo el árbol y conservar propiedad.
    datos[0] = datos[datos.size() - 1];

    datos.pop_back(); // ya pasaste el más pequeño al frente. Borra el que está al final que es el mismo valor

    maxHeapify(datos, 0); // reorganizas el árbol
}
```



Top, Empty & Size

```cpp
/*Regresa el valor del dato que esta con mayor prioridad en la fila priorizada.*/
int priority_queue::top(){

    if (datos.size() >= 1){
        return datos[0];
    }

    cout << "No hay elementos en la lista" << endl;
    return -1;
}

/*Regresa un valor boleando diciendo si la fila priorizada esta vacía o tiene datos.*/
bool priority_queue::empty(){

    return (datos.size() == 0) ? true : false;
}

/*Regresa la cantidad de datos que tiene la fila priorizada*/
int priority_queue::size(){

    return datos.size();
}
```



maxHeapify

```
void maxHeapify(vector<int> &datos, int i){

    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int largest;

    if(left <= datos.size() && datos[left] > datos[right]){
        largest = left;
    } else {
        largest = i;
    }

    if (right <= datos.size() && datos[right] > datos[largest]){
        largest = right;
    }

    if (largest != i){
        swap(datos[i], datos[largest]);
        maxHeapify(datos, largest);
    }
}
```

```
Void maxHeapify (vector<int> & datos, int i)
    left = 2*i+1
    right = 2*i+2
    largest =
    if left <= datos.size() and datos[left] > datos[right]
        largest = left
    else
        largest = i

    if (right <= datos.size() and datos[right] > datos[largest]
        largest = right;

    if (largest != i)
        swap(datos[i], datos[largest])
        maxHeapify (datos, largest);
```

$$T(n) \begin{cases} 1 & largest == i \\ T(\frac{n}{2}) + k & largest != i \end{cases}$$

de 3 nodos, siempre interactuos solo con 2. De n nodos

$$T\left(\frac{n}{2^k}\right) + k \qquad n = 2^k \qquad \log n = k$$

$$2^{\log_2 n}$$

$$T\left(\frac{n}{n}\right) + \log n$$

$$T(n) = 1 + \log n = O(\log n)$$