Universidad de Concepción

Facultad de ingeniería
Departamento de ingeniería informática y Ciencias de la computación

Tarea 2 de Análisis de Algoritmos

Análisis amortizado

Lisette Morales.

Pablo Venegas.

Profesor : Andrea Rodríguez.

Índice

1. Implementación.

1.1. Códigos.

1.1.1. Clase nodo.

```
class Node{
private:
    int key;
    Node *prev;
    Node *next;
public:
    Node(){}
    void set_key(int x){ key=x; }
    void set_prev(Node *x){ prev=x;}
    void set_next(Node *x){ next=x;}
    int get_key(){return this->key;}
    Node *get_prev(){return this->prev;}
    Node *get_next(){return this->next;}
};
```

Esta es la clase que ocupamos para trabajar con punteros o "nodos"dentro de la pila (implementada como una lista doblemente ligada) y que posee las funciones básicas para poder trabajar la pila.

1.1.2. Clase que representa la pila.

```
class Doubly_Linked_List{
private:
    Node *top = new Node();
public:
    Doubly_Linked_List(){
        top = NULL;
    }
}
```

Esta es la forma en que nuestra implementación maneja la inicialización de la pila, y el nodo "top" es el puntero que es el encargado de estar apuntando siempre a nuestro tope de la pila.

1.1.3. Función insert.

```
void insert(int x){

    Node *newTop = new Node();
    newTop->set_key(x);
    newTop->set_next(NULL);
    newTop->set_prev(top);
    top=newTop;
    while(top->get_prev()!=NULL && x < (top->get_prev())->get_key()){
     pop();
        pop();
        newTop->set_key(x);
        newTop->set_next(NULL);
        newTop->set_prev(top);
        top=newTop;
}
}
```

En primer lugar esta función tiene que insertar en un pila ordenada un elemento x, si este nuevo elemento ingresado en el tope de la pila no es el mayor, restablece el orden de la pila hasta que x sea el mayor elemento en la pila eliminando todos los números entre el tope y la posición correcta donde debe estar nuestro nuevo elemento.

Principalmente esto se logra mediante un estilo de Insertion sort que debe ir aplicando un estilo multipop para ir eliminando los elementos en la pila, mediante el while hace que se recorra nuestra pila "hacia abajo" y en cada iteración vaya comparando nuestro elemento nuevo que ingresamos en el tope de la pila con el que inmediatamente debajo de este, de esta forma recorremos la pila para encontrar el lugar correcto donde debería ubicarse nuestro nuevo elemento y restablecer así el orden de la pila.

De esta forma cuando analizamos el peor caso para esta función nos encontramos con que si tenemos una pila ya ordenada de tamaño "n" y queremos agregar un nuevo elemento que debe estar en la primera posición en nuestra pila, esto tendría un costo O(n) dado que debe hacer un "multipop", que es aplicar la función pop() "n" veces en este caso, para poder así ubicar nuestro nuevo elemento en el lugar correspondiente de la pila, de esta forma debe "recorrer" toda la pila para poder insertar el nuevo elemento.

1.1.4. Función pop.

```
int pop(){

if (top == NULL){
    cout<<"UPS, no hay nada en la pila"<<endl;
     }
        else{
            Node *antiguo= new Node();
            antiguo = top;
            int eliminado = antiguo->get_key();

            if(top->get_prev()==NULL){
             top=NULL;
             delete antiguo;
             return eliminado;
            }
            else{
                top = top->get_prev();
                top->set_next(NULL);
                top->set_prev(top->get_prev());
                delete antiguo;
                return eliminado;
            }
        }
    }
```

Esta función es la que nos permite realizar el pop en la pila, la implementación de esta función es la típica para este tipo de implementación de pila, busca simplemente en borrar el tope de la pila y que el elemento bajo el tope se convierta en el nuevo tope de la pila, esta función tiene un costo constante O(1) dado que solo elimina un elemento de la pila, y este es el que está en el tope, por ende no hay que hacer búsquedas ni ordenamientos, solo borrar el primer elemento y referenciar el nuevo tope.

1.1.5. Función destroy.

```
void destroy(){
        if(top==NULL){cout<<"nada que destruir"<<endl;return;}
        else{
            Node *y = top;
            Node *prev = new Node();

            while(y!=NULL){
             prev=y->get_prev();
             prev->set_next(NULL);
             if (prev->get_prev()==NULL){delete y; top=NULL; return;}
             else{
                 prev->set_prev(prev->get_prev());
                 delete y;
                 y = prev;
             }
            }
            top=NULL;
        }
    }
```

Esta función es la que se encarga de eliminar toda la pila, esta función se asemeja mucho a la función multipop, con la única particularidad que en este caso la cantidad de pop que se hacen es la cantidad de elementos en la pila, de esta manera ocupamos un algoritmo muy parecido al de la función pop previamente mostrada, con la diferencia que el while presente nos permite llegar hasta el último elemento de la pila, eliminando cada elemento de esta hasta que ya no queden mas, el costo en peor caso para esta función se basa en que si se tiene una pila de tamaño "n" ordenada, para poder destruirla completamente tomará "n" iteraciones, por ende destroy termina teniendo un costo O(n) dado que principalmente consta de ejecutar "n veces pop".

2. Análisis de las funciones.

2.1. Análisis de costo amortizado ocupando potencial.

Al realizar el análisis amortizado para estas funciones se puede aplicar el método del potencial, de esta manera el análisis queda dado de la siguiente manera:

Definimos la función potencial para la pila como el número de objetos en ella, teniendo inicialmente una pila con N elementos ordenados podemos decir que $\Phi(D_0)$ inicialmente sera $\Phi(D_0) = N$. Dada la implementación que tenemos puede ocurrir que $\Phi(D_0) >= \Phi(D_i)$, por ende el costo total amortizado de n operaciones con respecto a nuestra función puede representar una cota inferior en el costo real. La formula del costo amortizado viene dada por:

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

En el caso de la función destroy, que debe hacer pop una cantidad de veces igual al tamaño de nuestra pila, tenemos que:

$$\Phi(D_i) - \Phi(D_{i-1}) = N - (N-1) = 1.$$

Por ende el costo amortizado de la función destroy es:

$$\hat{c_i} = 1 + 1 = 2.$$

En el caso de la función pop lo único que se hace es una nueva referencia al puntero que está en el tope de la pila, tenemos que:

$$\Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0.$$

Por ende el costo amortizado de la función pop es:

$$\hat{c_i} = 1 + 0 = 1.$$

Para el análisis de la función insert hay que tomar ciertas cosas en cuenta, primero que si el número a ingresar a la pila es mayor que el numero que esta en el tope de esta no hay ningún problema a la hora de insertar dado que se mantiene ordenada la pila, por ende el costo amortizado en esta situación será el siguiente:

$$\hat{c_i} = 1 + N - (N-1) = 2.$$

En cambio si el número a ingresar no es mayor, se deberá hacer una cantidad k de pop's igual a la cantidad de números que se tienen que eliminar para dejar el numero a insertar en el tope de la pila y mantener la pila ordenada, por ende su costo amortizado será el siguiente:

$$\hat{c_i} = 1 + N - k - (N - k + 1) = 0.$$

En consecuencia, el costo amortizado total de las n operaciones O(n), dado que todos los costos son çonstantes", y esto corresponde también a el costo que este algoritmo tiene cuando se analiza el peor caso.

2.2. Análisis de costo amortizado ocupando contabilidad

En primer lugar sabemos que los costos reales de estas funciones son O(1) para la función pop, O(N) para la destroy y para la función insert puede variar entre O(1) en el mejor caso, O(K) o O(N) en el peor caso, con N representando el número de elementos en la pila y K la cantidad de elementos que hay que eliminar para insertar el elemento donde corresponde. Partiendo de esta base ocupando el método de contabilidad podemos decir lo siguiente.

La función insert se le asignará un crédito de 3 por cada llamada, dado que esta función ocupa dos pop dentro de ella si es que no estamos en el mejor caso (numero a insertar es menor que el tope de la pila), de esta manera para las funciones pop y destroy les asignamos un crédito de 0, esto se define así dado que en primer lugar para poder hacer pop o destroy, primero tienen que haber elementos en la pila, por ende se puede decir que siempre habrá crédito suficiente para poder ejecutar cualquiera de estas dos operaciones, De esta manera cada vez que se produzca un insert, en cualquier caso, siempre habrá un mínimo de 1 crédito para que sea usado por cualquiera de las otras dos funciones. De esta forma si nos enfrentamos cada vez al peor caso con n operaciones el costo de esto será a lo más O(n).

2.3. Análisis de costo amortizado ocupando análisis agregado.

Para el análisis agregado de las funciones, en primer lugar como lo planteamos previamente, la única función que tiene un peor caso es la función insert, dado que esta es la que debe analizar el correcto ordenamiento de la pila, por ello el análisis agregado del algoritmo queda dado por lo siguiente.

Dado que para que las funciones pop y destroy puedan ejecutarse se necesita que haya elementos en la pila, se toma en cuenta una pila con "n" elementos ordenados, si analizamos el peor caso de estas funciones, insert es la única que genera conflicto en el peor caso dada su implementación, por ende como sabemos que para el peor caso tenemos que esta tiene un costo de O(n), si aplicamos n veces la operación insert siempre en peor caso nos terminará entregando un costo de O(n cuadrado), en el caso de pop esta tiene un costo constante y la función destroy tiene un costo igual a la cantidad total de elementos en la pila, O(n) en nuestro caso, por ello dado que solo la función insert es la que al aplicar n operaciones de esta nos entrega un costo de O(n cuadrado) tenemos que aplicando el análisis agregado tenemos lo siguiente.

$$\frac{T(n)}{n} = \frac{O(n^2)}{n} = O(n)$$

.

Por ende la pila tiene un costo amortizado de O(n).

2.4. Análisis experimental

Para realizar el análisis experimental de los datos se trabajó con cantidades fijas de elementos en la pila, probando múltiples casos que varían entre 10^2 y 10^8 aumentando la cantidad a la siguiente potencia de 10, se analizaron solo 2 funciones, insert y destroy, para el caso del insert se analizó primero insertando los N elementos a la pila de forma ordenada, luego se analizó qué pasa si se ingresa un nuevo elemento y este es el que cumple el peor caso para este algoritmo, por ende deberá destruir casi completamente la pila para poder ingresar el nuevo elemento donde corresponde, el inicio de la pila. Finalmente se analizó la función destroy, que principalmente se considera como un multipop de la cantidad total de datos en la pila, obteniendo los siguientes resultados:

Cantidad de nodos	Insert	Insert Worst Case	Destroy
100	0.000117054	0.000101677	0.000022509
1000	0.000372548	0.00090363	0.000192181
10000	0.0037394	0.00681667	0.000947308
100000	0.0200345	0.0477888	0.0113305
1000000	0.190301	0.456727	0.0995602
10000000	1.8893	4.4613	0.997171
100000000	26.0367	81.1894	16.2123

Tabla 1: Tiempo en segundos que toma realizar cada una de las operaciones.

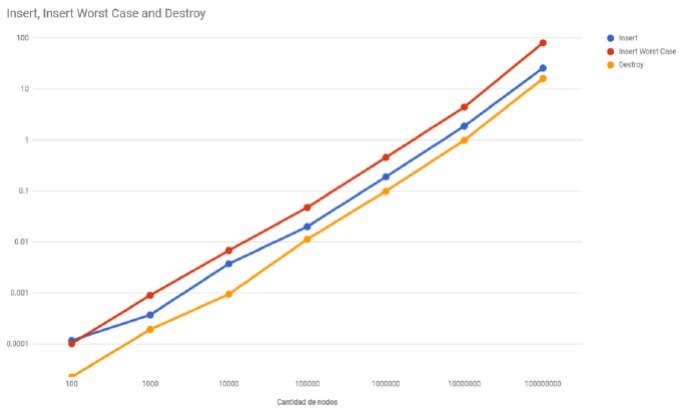

Figura 1: Gráfico que muestra como aumenta el tiempo a medida que avanza la cantidad de elementos en la pila