Análisis de algoritmos de ordenamiento. Parte I

Fabián Orozco Chaves - B95690

Resumen—En este trabajo se realiza la implementación de dos estructuras de datos utilizando C++ como lenguaje de programación, para comparar su eficiencia se mide el tiempo que tarda cada uno en realizar búsquedas de elementos mediante operaciones estudiadas en el curso. El resultado de su comparación es aproximado a lo que se puede inferir con lo estudiado en clase: con una notable diferencia **en algunos casos** en las búsquedas entre la lista y el árbol binario.

I. Introducción

En este trabajo se buscan implementar dos estructuras de datos: lista enlazada con nodo centinela y árbol de búsqueda binario. además de comprobar sus diferencias de eficiencia (complejidad de tiempo) en la realidad respecto a funciones de búsqueda de elementos.

En teoría se espera que la lista enlazada tenga un tiempo de inserción y borrado de $T(n) = \Theta(1)$, y uno de $T(n) = \Theta(n)$ para la búsqueda tanto en promedio como en el peor caso; por otro lado el árbol debería de comportarse con una complejidad $T(n) = \Theta(n)$ para el peor caso pero con un caso promedio de $T(n) = \Theta(\log(n))$ para todas las funciones anteriores.

Metodología

Para realizar la comparación entre las estructuras de datos se implementa en código aportado, las operaciones estudiadas en el curso para cada estructura de datos. Además se realiza en un archivo aparte (main) las pruebas de cada estructura para comprobar la correctitud de las operaciones y además compararlas mediante las siguientes pruebas:

Primera prueba [llist]
1. Crear lista con 1millón (1M) de nodos con llaves como enteros aleatorios [0,2M].
2. Seleccionar elementos al azar [0,2M] y buscarlos en la lista (estén o no); tomando la cantidad de búsquedas realizadas en diez segundos.

Segunda prueba [llist]
1. Crear lista con 1M de nodos secuenciales (consecutivos [0,2M[ ).
2. Seleccionar elementos al azar [0,2M]; tomando la cantidad de búsquedas realizadas en 10 segundos.

Primera prueba [bstree]
1. Crear un árbol binario con 1M de nodos con llaves como enteros aleatorios [0,2M].
2. Seleccionar elementos al azar [0,2M] y buscarlos en el árbol (estén o no); tomando la cantidad de búsquedas realizadas en diez segundos.

Segunda prueba [bstree]

1. Crear árbol binario con 1M de nodos secuenciales (consecutivos [0,2M[ ).
2. Seleccionar elementos al azar [0,2M]; tomando la cantidad de búsquedas realizadas en 10 segundos.
   Ver nota en **resultados.**

El código se muestra en el apéndice A y está basado en el pseudocódigo del libro de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein titulado Introduction to Algorithms. 3ra edición.

Cuadro I
CANTIDAD DE BÚSQUEDAS REALIZADAS EN 10 SEGUNDOS.

ESTRUCTURA / BUSQUEDA	Tamaño(M)	Aleatoria	Secuencial
LLIST	1	2607	2644
BSTREE	1		
RECURSIVA		7236477	3056
ITERATIVA		7488920	2985

III. Resultados

2.1

1. Para la lista enlazada, las operaciones de construcción, destrucción, inserción, búsqueda y borrado fueron implementadas y probadas correctamente.

2. Para la primera prueba de la lista, se lograron realizar 2607 búsquedas**.**

3. Para la segunda prueba de la lista se lograron realizar 2644 búsquedas.

4. Para ambos casos de inserción (aleatorios y secuenciales) no se realizó una cantidad de búsqueda más sustancial que en la otra, esto corresponde a lo esperado ya que en una lista la búsqueda de un elemento toma el mismo tiempo sin importar si está o no ordenada. Complejidad esperada en teoría: $T(n) = \Theta(n)$.

```
$ g++ -Wall -Wextra -O3 mainList.cpp -o mainList; ./mainList

-inicia

Qty de busquedas en 10 segundos [llist]
Aleatoria:  2607
Secuencial: 2644

-termina
```

Figura 1. Salida en terminal con los resultados de la lista enlazada.

2.2

1. Para el árbol de búsqueda binario, las **operaciones** de construcción, destrucción, inserción, borrado, búsqueda de llave recursiva e iterativa, búsqueda del máximo, mínimo, sucesor de un nodo y recorrido del árbol en orden fueron implementadas y probadas correctamente.

2. Para la primera prueba del árbol se realizaron de ambas maneras: con la búsqueda **recursiva e iterativa**; se lograron realizar **7236477** búsquedas con la búsqueda recursiva, además de **7488920** con la búsqueda iterativa.

3. Insertar n llaves ordenadas en un árbol de búsqueda binario puede tomar mucho tiempo si n es grande ya que la complejidad de la inserción en el **peor caso** (línea recta) es $T(n) = \Theta(n)$ por lo que en cada inserción se debe recorrer hasta la última hoja. **Nota**: para evitar esperar el tiempo que tarda el método de inserción se **creó otro método** que utiliza la lógica de árbol binario donde recorre por los hijos derechos (mayores que el padre) e inserta el valor correspondiente **(algoritmo 10).**

4. Teniendo en cuenta que las búsquedas realizadas por la lista (3700) conforman apenas el 5% aproximado de las realizadas por el árbol (7236477) se puede inferir que el árbol binario es sustancialmente mejor que la lista ya que permitió hacer mucho más del doble de búsquedas **(caso promedio teórico esperado).**

5. Contrario al punto anterior, la estructura de datos más eficiente **no es tan clara**, lo cual corresponde a lo esperado ya que para el árbol los datos se encuentran consecutivos (todos hacia la derecha) por lo cual se comporta casi como una lista al realizar las búsquedas, teniendo el **peor caso posible** con un tiempo para ambas estructuras de $T(n) = \Theta(n)$.

```
$ g++ -Wall -Wextra -O3 mainTree.cpp -o mainTree; ./mainTree

-inicia

Qty de busquedas en 10 segundos [tree]
Aleatoria[rec]:  7236477
Aleatoria[it]:   7488920
Secuencial[rec]: 3056
Secuencial[it]:  2985

-termina
```

Figura 2. Salida en terminal con los resultados del árbol de búsqueda binario.

IV. Conclusiones

El orden de inserción de los datos define la complejidad temporal para el árbol binario, pero no es de este modo para la lista donde el peor caso es igual que el promedio.

Queda comprobado que la estructura de datos más eficiente fue la del árbol binario lo cual corresponde a lo esperado en teoría para los casos promedio, con una complejidad temporal de $T(n) = \Theta(\log(n))$ para la búsqueda la cual es la operación más común que se realiza en dicha estructura por lo cual es muy importante tener la estructura más eficiente en cuanto a este aspecto. Por otro lado, en el peor caso su complejidad equivale a la de la lista por lo que por eficiencia es preferible tener un árbol de búsqueda binario como estructura de datos.

APÉNDICE A
CÓDIGO DE LOS ALGORITMOS

El código se muestra en los algoritmos 1, 2, 3, 4, 5 y 6 además el archivo adjunto a este documento contiene el código de los algoritmos.

2.1 – Algoritmos de lista enlazada

Algoritmo 1 Constructor de nodo

```
llnode () {
    key = 0;
    prev = this;
    next = this;
};
```

Algoritmo 2 Constructor de la lista. Crea lista vacía.

```
llist(){
   nil = new llnode<T>();
};
```

Algoritmo 3 Destructor de la lista. Borra la lista.

```
~llist() {
    llnode<T> * actual = nil->next;
    llnode<T> * victim = nullptr;
    while (actual && actual != nil) {
        victim = actual;
        actual = actual->next;
        delete victim;
    }
    delete nil;
};
```

Algoritmo 4 Operación de búsqueda. Busca la llave iterativamente. Si la encuentra, devuelve un apuntador al nodo que la contiene; sino devuelve NULL.

```
llnode<T>* listSearch(const T& k){
    llnode<T> *actual = nil->next;
    while (actual->key != k && actual != nil){
        actual = actual->next;
    }
    return actual;
};
```

Algoritmo 5 Operación de inserción. Inserta el nodo x en la lista.

```
void listInsert(llnode<T>* x) {
    x->next = nil->next;
    nil->next->prev = x;
    nil->next = x;
    x->prev = nil;
};
```

Algoritmo 6 Operación de borrado. Saca de la lista la llave contenida en el nodo apuntado por x. Devuelve la dirección del nodo eliminado para que se pueda disponer de él.

```
llnode<T>* listDelete(llnode<T>* x){
    x->prev->next = x->next;
    x->next->prev = x->prev;
    return x;
};
```

2.2 – Algoritmos del árbol de búsqueda binario

Algoritmo 1 Constructor del árbol binario. Crea un árbol vacío.

```
tree(){
    root = new node<T>();
};
```

Algoritmo 2 Destructor del árbol binario. Borra el árbol.

```
~tree(){
    node<T>* victim = root;
    while(victim != nullptr) {
        // cout << "victim: " << victim->key << "\n";
        victim = treeDelete(victim);
        delete victim;
        victim = root;
    }
};
```

Algoritmo 3 Operación de recorrido en orden. Efectúa dicho recorrido del árbol cuya raíz es apuntada por x, imprimiendo en cada visita la llave de cada nodo (menos la de la raíz).

```
void inorderTreeWalk(node<T>* x){
    if (x != nullptr) {
        inorderTreeWalk(x->left);
        if (x != root){
            cout << x->key << " ";
        }
        inorderTreeWalk(x->right);
```

```
    }
};
```

Algoritmo 4 Operación de búsqueda recursiva. Busca la llave recursivamente; si la encuentra, devuelve un apuntador al nodo que la contiene, sino devuelve NULL

```
node<T>* f_treeSearch(node<T>* x, const T& k) {
    if (x == nullptr || k == x->key) {
        return x == root ? nullptr : x;
    }
    if (k < x->key) {
        return f_treeSearch(x->left,k);
    } else {
        return f_treeSearch(x->right,k);
    }
}

node<T>* treeSearch(const T& k){
    return f_treeSearch(root, k);
};
```

Algoritmo 5 Operación de búsqueda iterativa. Igual que el algoritmo 4 pero iterativo.

```
node<T>* iterativeTreeSearch(const T& k){
    node<T>* x = root;
    while(x != nullptr && k != x->key){
        if (k < x->key) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    return x  == root ? nullptr : x;
};
```

Algoritmo 6 Operación de mínimo. Devuelve el nodo que tiene la llave menor. Si el árbol está vacío devuelve NULL.

```
node<T>* f_treeMinimum(node<T>* x) {
    while (x->left != nullptr) {
            x = x->left;
    }
    return root != nullptr ? x : NULL;
}

node<T>* treeMinimum(){
    return f_treeMinimum(root);
};
```

Algoritmo 7 Operación de máximo. Devuelve el nodo que tiene la llave mayor. Si el árbol está vacío devuelve NULL.

```
node<T>* f_treeMaximum(node<T>* x) {
    while (x->right != nullptr && x->right != root) {
        x = x->right;
    }

    return x != nullptr ? x : NULL;
}

node<T>* treeMaximum() {
    return f_treeMaximum(root);
};
```

Algoritmo 8 Operación sucesor. Devuelve el nodo cuya llave es la que le sigue a la del nodo x. Si no existe tal nodo devuelve NULL.

```
node<T>* treeSuccessor(const node<T>* x){
    if (x->right != nullptr) {
        return f_treeMinimum(x->right);
    }
    node<T>* y = x->p;
    while (y != nullptr && x == y->right){
        x = y;
        y = y->p;
    }
    return y;
};
```

Algoritmo 8 Operación de inserción. Inserta el nodo z en la posicion que le corresponde en el arbol.

```
void treeInsert(node<T>* z){
    node<T>* y = nullptr;
    node<T>* x = root;
    while (x != nullptr){
        y = x;
        if (z->key < x->key){
            x = x->left;
        } else {
            x = x->right;
        }
    }
    z->p = y;
    if (y == nullptr){
        root = z; // tree was empty
    }
    else if (z->key < y->key) {
        y->left = z;
```

```
    }
    else {
        y->right = z;
    }
};
```

Algoritmo 9 Operación de borrado. Saca del árbol la llave contenida en el nodo apuntado por z. Devuelve la dirección del nodo eliminado para que se pueda disponer de ella

```
node<T>* treeDelete(node<T>* z){
    node<T> * victim;
    if (z->left == nullptr) {
        victim = f_transplant(z,z->right);
    }
    else if (z->right == nullptr) {
        victim = f_transplant(z, z->left);
    }
    else {
        node<T>* y = f_treeMinimum(z->right);
        if (y->p != z){
            victim = f_transplant(y,y->right);
            y->right = z->right;
            y->right->p = y;
        }
        victim = f_transplant(z,y);
        y->left = z->left;
        y->left->p = y;
    }
    return victim;
};

// mueve un subarbol dentro del árbol:
// el padre del nodo u se convierte en el padre del nodo v.
node<T>* f_transplant(node<T>* u, node<T>* v) {
    if (u->p == nullptr) { // cambia la raiz por v (u es la raiz)
        root = v;
    }
    else if (u == u->p->left) { // cambia a u por v (izq de padre)
        u->p->left = v;
    }
    else { // cambia a u por v (der de padre)
        u->p->right = v;
    }
    if (v != nullptr) { // v no es la raíz
        v->p = u->p; // padre de v es padre de u
    }
    return u;
}
```

Algoritmo 10 Creación de árbol consecutivo. Llena el árbol con 1M de números consecutivos / secuenciales [0,2M[.
2.1.2) método del paso 3.

```
void f_fillTree_sec(int qtyNodos){
    // tome el hijo derecho del nodo maximo (derecha)
    // asignele el nuevo valor
    node<int>* current = root; // inicia con la raiz
    for (int i = 0; i < qtyNodos; ++i) { // llena con 1M de valores
consecutivos
        // if (i % 100000 == 0) {cout << i << " ";} // para ver avance
(prueba)
        current->right = new node<int>(i);
        current = current->right;
    }
}
```

REFERENCIAS

Cormen, T. H., Leiserson, C. E., Rivest, R. L.,, Stein, C. (2001). Introduction to Algorithms. The MIT Press. ISBN: 0262032937