Estructuras de Datos

Elizabeth Huang Wu C23913

***Resumen*—En este trabajo se realiza un análisis de varias estructuras de datos para hacer una comparación de complejidades temporales teóricas con los resultados prácticos. Con el desarrollo de varios experimentos se recolectan los datos necesarios y se llega a la conclusión que las complejidades teóricas son respaldadas por los tiempos de ejecución obtenidos.**

***Palabras clave*—estructura, datos, lista, árbol, tiempos**

I. Introducción

En este trabajo se refiere al tema de las estructuras de datos, que se define como una manera de guardar y organizar datos para facilitar las modificaciones y su acceso. A la hora de implementar el diseño de un algoritmo, es importante utilizar una estructura de datos apropiada, ya que no se puede utilizar solo una para todos los casos existentes. Dado a eso, es significativo conocer las limitaciones y ventajas de cada una.

Para poder realizar este análisis se realizara un experimento donde se puedan ver los tiempos de ejecución del algoritmo de búsqueda en diferentes estructuras de datos, entre ellas: lista enlazada y árboles de búsqueda binaria. Se busca comparar la eficiencia teórica con la eficiencia practica de las estructuras con los datos obtenidos. Para este experimento se utilizaran dos estructuras de datos:

1. Lista Enlazada: Una lista enlazada es una estructura donde los objetos se acomodan de una forma lineal. Es diferente a un arreglo ya que utiliza punteros en los objetos para navegar en vez de índices. En este caso se utilizara una lista simplemente enlazada, entonces solo tiene de atributo un puntero al siguiente.
2. Árbol de Búsqueda Binaria: Se organiza de en un árbol binario. Cada nodo contiene atributos como un hijo izquierdo, un hijo derecho y un puntero al padre. Si alguno de los atributos no existe, se le pone un valor NIL.

II. Metodología

Para lograr lo propuesto se piensa crear un programa en el lenguaje programador c++ que nos permitirá hacer las pruebas necesarias para el análisis y la comparación de estas estructuras. El experimento se realizara en la IDE 'Visual Studio Code', en una computadora con sistema operativo macOS con un procesador 6-Core Intel Core i5, GPU AMD Radeon Pro 5300 4 GB y RAM 8 GB 2667 MHz DDR4. Es importante tener en cuenta el hardware en el cual se realizan estas pruebas ya que los resultados pueden variar del dispositivo.

Tanto el código, se creara un archivo .h para cada estructura de datos y se implementaran los operadores básicos, entre estos: crear, destruir, insertar, eliminar y buscar. El código a utilizar para cada estructura se muestra en los apéndices, son basados en el pseudocódigo del libro de Cormen y colaboradores [1]. En esta investigación, se utilizaran las estructuras de lista enlazada y árbol de búsqueda binaria para desarrollar los experimentos prácticos. Para ambas estructuras, se insertaran 1 000 000 elementos y se buscaran 10 000 al menos 4 veces y se guardaran los tiempos que dura en buscar dicha claves. Habrán dos tipos de inserciones, aleatoria y ordenada, esto es para poder diferenciar la eficiencia del operador en ambos casos. El rango de los datos aleatorio sera de 0 a 2*n, n siendo el numero de elementos por ser insertados. La búsqueda también sera aleatoria con números entre 0 y 2*n.

Para poder capturar los tiempos de ejecución de las búsquedas, se utilizara la librería <chrono>. Antes de la operación y después de la operación se guarda el tiempo actual, luego se resta el final con el principio para obtener el tiempo de duración. En este trabajo se estará utilizando milisegundos para poder hacer un análisis con datos mas precisos y detallados.

Cuadro I
Tiempo de ejecución del algoritmo de búsqueda en una lista enlazada.

	Tiempo (ms)				
	Corrida				
Inserción	1	2	3	4	Prom.
Aleatorio	34946.8	34217.9	34620.8	34053.9	34459.9
Ordenado	32037.7	32510.0	32601.9	32525.4	33168.8

Cuadro II
Tiempo de ejecución del algoritmo de búsqueda en un árbol de búsqueda binaria.

	Tiempo (ms)				
	Corrida				
Inserción	1	2	3	4	Prom.
Aleatorio	8.8	7.8	7.7	7.4	7.9
Ordenado	3.8	2.9	3.4	3.6	3.4

III. Resultados

Los tiempos de ejecución de las 4 corridas de los del algoritmo de búsqueda en una lista enlazada se muestra en el cuadro I. Los tiempos del árbol de búsqueda binaria, se encuentran en el cuadro II

Lista Enlazada

1. Lista Aleatoria: Entre los datos recolectados, se ve una desviación estándar de aproximadamente 348.70. De esto se puede deducir que los datos son relativamente consistentes, no muy dispersos pero si tiene variabilidad.

Esta consistencia se puede ver ya que los valores se acercan a la media 34459.9.

2. Lista Ordenada: De los datos obtenidos, hay una desviación estándar alrededor de 222.73. Similar a la lista aleatoria, los tiempos de búsqueda son relativamente consistentes con una media de 33168.8, pero si hay variabilidad.

Árbol de Búsqueda Binaria

1. Árbol de Búsqueda Binaria Aleatorio: Los tiempos recolectados en las corridas son parecidas con una desviación de 0.5262. Con esto se puede decir que hay una consistencia alta y poca variabilidad. También se puede asumir dado a que todos los datos se acercan a la media 7.9.
2. Árbol de Búsqueda Binaria Ordenado: Igual al anterior, hay una consistencia alta y poca variabilidad con una dispersión de 0.3345. Además los datos se acercan a la media 3.4.

Comparación entre Tipos de Inserción

Los resultados se pueden ver gráficamente en la figura 1 y 2. Se puede notar que los tiempos de ejecución de la búsqueda tienden a ser más rápidos cuando se insertan datos de una forma ordenada. La diferencia es más que el doble en el caso del árbol de búsqueda binaria, mientras que en la lista simplemente enlazada solo se ve una mejora pequeña.

También es importante denotar que los tiempos entre ambas estructuras son significativamente diferentes. La lista enlazada dura mas que el doble ejecutando la búsqueda de los 10 000 datos que el árbol de búsqueda binaria en ambos tipos de inserción. Con estas pruebas se nota una diferencia relevante entre las estructuras de datos y los tipos de inserciones.

IV. Conclusiones

A partir de los resultados se puede concluir que para la lista enlazada entre los dos casos de inserción, los promedios no tienen una gran diferencia, sin embargo si se nota que los tiempos de la lista ordenada son más bajos. Los resultados son los esperados considerando la complejidad temporal teórica. La lista aleatoria con cada búsqueda requiere un escaneo completo de la lista en el peor caso. Mientras que en la lista ordenada, tiene el beneficio de saber si un elemento no existe dependiendo de los valores que se van recorriendo que potencialmente reduzca el tiempo de búsqueda. En ambas listas, la complejidad en el peor caso es de *O(n)*.

Mientras que para el árbol de búsqueda binaria, entre los dos tipos de inserciones, se ve una diferencia significativa de tiempos de ejecución ya que el promedio del ordenado es menos que la mitad del tiempo promedio del aleatorio. Estos resultados son esperados dado a que el árbol aleatorio puede ser desequilibrado lo cual significa que la altura del árbol se hace mas larga y aumenta el tiempo de búsqueda. Mientras que un árbol ordenado es mas equilibrado ya que la entrada es de manera descendiente o ascendiente y asegura una estructura de tamaño mínimo. El peor caso de una búsqueda en un árbol de inserción aleatoria y ordenado es de *O(logn)*, que se ve reflejado en el corto tiempo de ejecución.

Entre las dos estructuras de datos se ve una diferencia significativa entre los promedios de tiempos de ejecución, ya sea con inserción aleatoria o inserción ordenada. La lista simplemente enlazada tiende a durar más del doble de tiempo buscando 10 000 datos que un árbol de búsqueda binaria. Sin embargo es relevante considerar que en ambas estructuras, el tiempo de duración en la inserción ordenada fue más rápida.

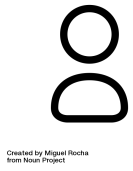
Este resultado fue el esperado ya un árbol binario de búsqueda usa la estructura del árbol que permite reducir el espacio de búsqueda a la mitad en cada paso, logrando una complejidad de tiempo promedio de *O(logn)* en árboles balanceados. Por otro lado, en una lista enlazada, cada nodo debe ser examinado secuencialmente hasta encontrar el nodo buscado, resultando en una complejidad de tiempo de *O(n)*. Esta diferencia significativa se da por la capacidad del árbol binario de búsqueda de dividir y conquistar, mientras que una lista enlazada requiere recorrer todos los nodos uno por uno y no tiene manera de acelerar este proceso.

El experimento y análisis que se llevo a cabo nos ayuda a decidir cual es la mejor estructura de datos dependiendo del tipo de problema y algoritmo en situaciones reales. Nos deja un conocimiento mas profundo y valioso que nos ayuda reconocer el rendimiento y eficiencia de cada uno. Los resultados nos deja concluir que las complejidades temporales teóricas son respaldadas por los tiempos prácticos.

Referencias

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

Elizabeth Huang Estudiante de la Universidad de Costa Rica.



Created by Miguel Rocha
from Noun Project

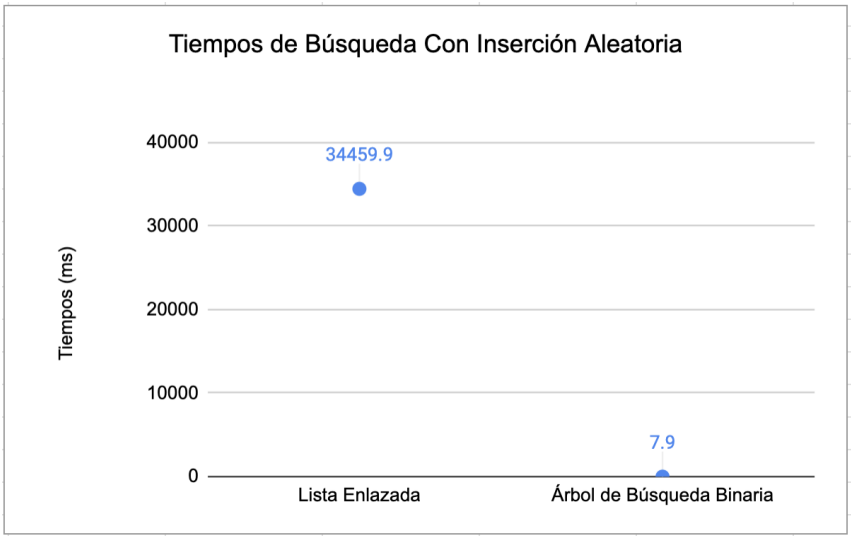


Figura 1. Gráfico comparativo de los tiempos promedio de búsqueda aleatoria en las estructuras de datos con inserción aleatoria.

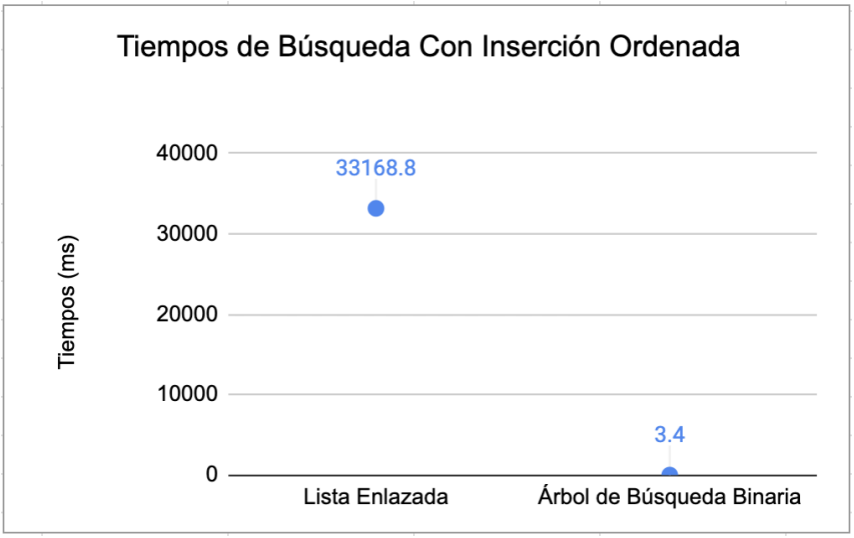


Figura 2. Gráfico comparativo de los tiempos promedio de búsqueda aleatoria en las estructuras de datos con inserción ordenada.

APÉNDICE A
CÓDIGO DE LOS ALGORITMOS

El código se muestra en los algoritmos 1, 2, 3, 4, 5 y 6.

Algoritmo 1 Algoritmo del nodo de una lista simplemente enlazada.

```
// Nodos de la lista:
template <typename T>
class llnode
{
private:
    T key;
    llnode<T> *next;

public:
    llnode() {
        this->next = nullptr;
    };

    llnode (const T& k, llnode<T> *y = nullptr):key(k), next(y)  {};

    ~llnode() {};

    llnode<T>* GetNext() {
        return next;
    }

    void SetNext(llnode<T>* next){
        this->next = next;
    }

    int GetKey(){
        return key;
    }

};
```

Algoritmo 2 Algoritmo de la lista simplemente enlazada.

```
// Lista enlazada con nodo centinela:
template <typename T>
class llist
{
private:
    llnode<T> *nil;

public:
    llist() {
        nil = new llnode<T>();
        nil->SetNext(nil);
    };

    ~llist() {
        while (nil->GetNext() != nil){
            Delete(nil->GetNext());
        }
        delete nil;
    };

    void Insert(llnode<T>* x) {
        x->SetNext(nil->GetNext());
        nil->SetNext(x);
    };

    llnode<T>* Search(const T& k) {
        llnode<T>* x = nil->GetNext();
        while (x != nil) {
            if (x->GetKey() == k) {
                return x;
            }
            x = x->GetNext();
        }
        return this->nil;
    };

    void Delete(llnode<T>* x) {
        llnode<T>* prev = nil;
        llnode<T>* curr = nil->GetNext();

        while (curr != nil && curr != x) {
            prev = curr;
            curr = curr->GetNext();
        }

        if (curr == x) {
            prev->SetNext(curr->GetNext());
            delete curr;
        }
    };
};
```

Algoritmo 3 Algoritmo del nodo de un árbol de búsqueda binaria.

```
template <typename T>
class bstnode
{
private:
    T key;
    bstnode<T> *p, *left, *right;
public:
    bstnode() {};

    bstnode(const T& k, bstnode<T> *w = nullptr, bstnode<T> *y = nullptr, bstnode<T> *z = null
        key(k), p(w), left(y), right(z) {};

    ~bstnode() {
    };

    int GetKey(){
        return key;
    }

    bstnode<T>* GetParent(){
        return p;
    }

    bstnode<T>* GetLeft(){
        return left;
    }

    bstnode<T>* GetRight(){
        return right;
    }

    void SetParent(bstnode<T>* p){
        this->p = p;
    }

    void SetLeft(bstnode<T>* left){
        this->left = left;
    }

    void SetRight(bstnode<T>* right){
        this->right = right;
    }
};
```

Algoritmo 4 Algoritmo de un árbol de búsqueda binaria.

```
template <typename T>
class bstree {
private:
    bstnode<T> *root;
public:
    bstree() {
        this->root = nullptr;
    };

    ~bstree() {
        clearTree(root);
    };

    void clearTree(bstnode<T>* x){
        if (x) {
            clearTree(x->GetLeft());
            clearTree(x->GetRight());
            delete x;
        }
    }

    bstnode<T>* GetRoot(){
        return root;
    }

    void buildBalancedBST(int n) {
        root = buildBalancedBSTHelper(0, n - 1);
    }

    bstnode<T>* buildBalancedBSTHelper(int start, int end) {
        if (start > end) {
            return nullptr;
        }
        int mid = start + (end - start) / 2;
        bstnode<T>* node = new bstnode<T>(mid);
        node->SetLeft(buildBalancedBSTHelper(start, mid - 1));
        if (node->GetLeft() != nullptr) {
            node->GetLeft()->SetParent(node);
        }
        node->SetRight(buildBalancedBSTHelper(mid + 1, end));
        if (node->GetRight() != nullptr) {
            node->GetRight()->SetParent(node);
        }
        return node;
    }
```

Algoritmo 5 Algoritmo de un árbol de búsqueda binaria.

```
void Insert(bstnode<T>* z) {
    bstnode<T>* x = root;
    bstnode<T>* y = nullptr;
    while (x != nullptr) {
        y = x;
        if (z->GetKey() < x->GetKey()){
            x = x->GetLeft();
        } else {
            x = x->GetRight();
        }
    }
    z->SetParent(y);
    if (y == nullptr) {
        root = z;
    } else if (z->GetKey() < y->GetKey()) {
        y->SetLeft(z);
    } else {
        y->SetRight(z);
    }
};
void InorderWalk(bstnode<T> *x) {
    if (x != nullptr) {
        InorderWalk(x->GetLeft());
        std::cout << x->GetKey() << std::endl;
        InorderWalk(x->GetRight());
    }
};

bstnode<T>* Search(bstnode<T> *x, const T& k) {
    if (x == nullptr || k == x->GetKey()) {
        return x;
    }
    if (k < x->GetKey()) {
        return Search(x->GetLeft(), k);
    } else {
        return Search(x->GetRight(), k);
    }
};

bstnode<T>* IterativeSearch(bstnode<T> *x, const T& k) {
    // Mientras x no sea puntero nulo y la clave de x no sea igual a k
    while (x != nullptr && k != x->GetKey()) {
        // Si la clave es menor a k
        if (k < x->GetKey()) {
            // Se mueve al hijo izquierdo de x
            x = x->GetLeft();
        } else {
            // Si no, se mueve al hijo derecho de x
            x = x->GetRight();
        }
    }
    // Retornar el nodo con la clave buscada
    return x;
};
```

Algoritmo 6 Algoritmo de un árbol de búsqueda binaria.

```
    bstnode<T>* Minimum(bstnode<T> *x) {
        while (x->GetLeft() != nullptr) {
            x = x->GetLeft();
        }
        return x;
    };

    bstnode<T>* Maximum(bstnode<T> *x) {
        while (x->GetRight() != nullptr) {
            x = x->GetRight();
        }
        return x;
    };

    bstnode<T>* Successor(bstnode<T> *x) {
        if (x->GetRight() != nullptr) {
            return Minimum(x->GetRight());
        } else {
            bstnode<T>* y = x->GetParent();
            while (y != nullptr && x == y->GetRight()) {
                x = y;
                y = y->GetParent();
            }
            return y;
        }
    };

    void Delete(bstnode<T>* z) {
        if (z->left == nullptr) {
            Transplant(z, z->GetRight());
        } else if (z->GetRight() == nullptr) {
            Transplant(z, z->GetLeft());
        } else {
            bstnode<T>* y = Minimum(z->GetRight());
            if (y->GetParent() != z) {
                Transplant(y, y->GetRight());
                y->SetRight(z->GetRight());
                y->GetRight()->SetParent(y);
            }
            Transplant(z, y);
            y->SetLeft(z->GetLeft());
            y->GetLeft()->SetParent(y);
        }
        delete z;
    };
    void Transplant(bstnode<T>* u, bstnode<T>* v) {
        if (u->GetParent() == nullptr) {
            root = v;
        } else if (u == u->GetParent()->GetLeft()) {
            u->GetParent()->SetLeft(v);
        // Si x es hijo derecho
        } else {
            u->GetParent()->SetRight(v);
        }
        if (v != nullptr) {
            v->SetParent(u->GetParent());
        }
    }
};
```