Tarea Programada 2 - Estructuras de Datos

Jahir Valverde

Universidad de Costa Rica

Jahir Valverde (C28038) - Anàlisis de Algirtmos

Arturo Camacho

Mayo 30, 2024

Resumen

En este trabajo se implementan estructuras de datos vistas en clase (listas enlazadas con nodo centinela, árboles de búsqueda binaria, árboles rojinegros y tablas de dispersión) y se comprueba que su eficiencia teórica se corresponda con la práctica. Esta comparación entre teoría y realidad se hace creando las estructuras de datos, y sometiéndolas a pruebas de estrés cronometradas. Los tiempos de ejecución pueden variar según el equipo, en este caso se va a realizar en una computadora de 24GB de RAM, 8 núcleos.

Introduciòn

Se realizaron las diferentes pruebas planteadas por el enunciado de la tera, para las diferentes estructuras de datos. Estas pruebas consisten en ingresar a las estructuras de datos 2 tipos de elementos, en una nodos aleatorios, y en otra nodos secuenciales. Dependiendo del comportamiento de cada estructura, una se puede comportar mejor con aleatorios que secuenciales y viceversa, o incluso comportarse parecido. Esto se pretende discernir en este trabajo. En cada caso se harán 3 corridas.

Una vez implementados los algoritmos en los archivos .h proporcionados y archivos .c para aplicar las pruebas dichas anteriormente, se apuntan los resultados de las pruebas en una tabla y se analizan los comportamientos en los distintos casos.

Metodología

Lo primero es implementar las distintas estructuras en los archivos .h. Teniendo esto podemos hacer las pruebas del enunciado, que consiste en insertar elementos aleatorios y en orden, y buscar numeros aleatorios en 1 segundo, reportando el número total de búsquedas. Estos resultados se apuntan en la tabla de la **Figura 1**, y esto permite dar una visualización resumida de los distintos escenarios y sus comportamientos.

Vamos a comparar la versión aleatoria y secuencial en la estructura y también entre estructuras. Por ejemplo, la corrida aleatoria de la lista enlazada con la corrida aleatoria de la secuencial. Y también la corrida aleatoria de la lista enlazada con la corrida aleatoria del árbol binario. Y así sucesivamente.

Algoritmo	Tipo de Nodos		Variación
	Aleatorios	Secuenciales	
Lista Enlazada con Nodo Centinela	284	246	Similar
Árbol de Búsqueda Binario	879418	150	Más del doble

Figura 1

Resultados

Lista Enlazada con Nodo Centinela

Se puede apreciar en la **Figura 1** que para la corrida aleatoria logró hacer 284 búsquedas, mientras que para la corrida secuencial 246. Ambas corridas son muy similares, y es lo esperado desde la teoría, porque tanto para la aleatoria como secuencial se tiene una complejidad asintótica de $\theta(n)$, es decir, lineal.

Esto se debe a que como siempre se recorre de manera lineal, de la cabeza al final, en este caso la distribución de los elementos no afecta al algoritmos porque la búsqueda sigue siendo aleatoria. Si aleatoriamente sale un 5, va a encontrarlo más rápido en la secuencial, pero si sale un 980000 probablemente lo encuentre más rápido en la aleatoria. Estos dos casos se van balanceando y al final los tiempos son muy parecidos.

Árbol de Búsqueda Binaria

Para este caso, la diferencia sí es mucho más significativa. La versión aleatoria logró realizar 879418 búsquedas, mientras que la secuencial apenas 150. Por lo que la versión aleatoria es demasiadas veces más eficiente que la secuencial, y de igual forma esto se corresponde con la teoría, porque para la versión aleatoria se tiene una complejidad asintótica de $\theta(logn)$, mientras que la versión secuencial una complejidad de $\theta(n)$.

La lentitud de la versión secuencial se debe a que al insertar los datos siempre mayores que el anterior, se termina generando algo parecido a la lista enlazada de la **Figura 2**, donde

todos los elementos están a la derecha y no se logra efectuar un balanceo. Incluso es más lento que la búsqueda de la lista enlazada con nodo centinela porque el algoritmo de búsqueda es más complejo, teniendo más líneas de código, haciendo más comparaciones y gastando más memoria.

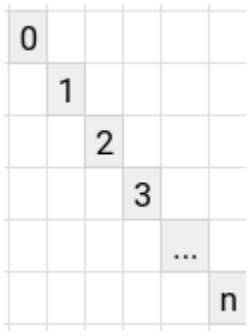

Figura 2

En cambio en la inserción aleatoria, logramos generar un árbol más balanceado, porque van a haber elementos a la izquierda y a la derecha, por lo que el número de iteraciones para encontrar un elemento es muchas veces menor.

Lista Enlazada vs Árbol Binario (Corrida Aleatoria)

Se puede apreciar que la búsqueda con árbol binario es muchas veces mejor que la búsqueda con lista enlazada cuando la inserción se hace de manera aleatoria.

Lista Enlazada vs Árbol Binario (Corrida Secuencial)

Se puede apreciar que la búsqueda con árbol binario es menos eficiente que la búsqueda con lista enlazada cuando la inserción se hace de manera secuencial.

Conclusión

Se concluye que la eficiencia teórica se corresponde con la práctica. Los distintos casos de prueba son fieles a la expectativa que nos da la teoría. Por ejemplo, es esperado que la versión aleatoria sea más eficiente para el árbol binario que para la lista enlazada, o que la versión aleatoria del árbol es más eficiente que la secuencial también del árbol.

Apèndice A

Lista Enlazada

```
//
//  llist.h
//

#ifndef llist_h
#define llist_h

// Nodos de la lista:
template <typename T>
class llnode
{
public:
 // Esta clase es para ser usada por otras clases. Por eficiencia los
atributos se dejan publicos.
 T key;
 llnode<T> *prev, *next;
  // Constructor por omision.
 llnode() {
 };
  // Inicializacion de los datos miembro.
 llnode (const T& k, llnode<T> *w = nullptr, llnode<T> *y =
nullptr):key(k), prev(w), next(y)  {};
  ~llnode() {
 };
};

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

 llist() {
   // Constructor (crea una lista vacia)
   nil = new llnode<T>();
```

```
   nil->next = nil; // el nodo siguiente apunta a si mismo
   nil->prev = nil; // el nodo anterior apunta a si mismo
 };
  ~llist() {
   // Destructor (borra la lista)
   llnode<T>* current = nil->next; // obtener la cabeza
   while (current != nil) { // si es distinto al centinela
     llnode<T>* to_delete = current; // guardar el puntero a borrar
     current = current->next; // guardar el siguiente
     delete to_delete; // borrar el actual
   }
   delete nil; // borrar el centinela
 };
  void Insert(llnode<T>* x) {
   // Inserta el nodo x en la lista.
   x->next = nil->next; // asignar el siguiente del nuevo nodo
   x->prev = nil; // asignar el anterior del nuevo nodo
   nil->next->prev = x; // hacer que el prev de la cabeza anterior ya no
apunte a nil sino al nodo que estamos insertando
   nil->next = x; // ahora la caebza es nuestro nuevo nodo
 };

 llnode<T>* Search(const T& k) {
   // Busca la llave iterativamente. Si la encuentra, devuelve un
apuntador al nodo que la contiene; sino devuelve el nodo nil (el
centinela).
   llnode<T>* current = nil->next; // obtener la cabeza
   while (current != nil && current->key != k) { // buscar hasta que el
current sea el centinela o se encuentre la llave
       current = current->next;
   }
   return current; // y retornar el valor encontrado
 };

 void Delete(llnode<T>* x) {
   // Saca de la lista la llave contenida en el nodo apuntado por x.
   if (x != nil) { // si no se esta pidiendo eliminar el centinela
     // se va a explicar con un ejemplo en vez de palabras con una lista
de
```

```
    // ejemplo en el que cada nodo se representa por (direccion, k=a) con
a
    // el valor de la llave del nodo. -> representa el siguiente nodo.
    // <- representa el antarior
    // Nodo centinela, k vacio: (0x2, k=9) <- (0x0, k) -> (0x5, k=2)
    // Lista: (0x5, k=13) <--> (0x7, k=12) <--> (0x3, k=3) <--> (0x9,
k=7)
    // Supongamos que se quiere eliminar 0x7
    x->prev->next = x->next; // x->prev: (0x5, k=13)
                             // x->next: (0x3, k=3)
                             // x->prev->next = x->next significa
                             // (0x5, k=13) -> (0x3, k=3)
    x->next->prev = x->prev; // x->next: (0x3, k=3)
                             // x->prev: (0x5, k=13)
                             // x->next->prev = x->prev significa
                             // (0x5, k=13) <- (0x3, k=3)
    // Con esto conseguimos la conexion (0x5, k=13) <--> (0x3, k=3)
    delete x; // como ya reubicamos las conexiones podemos borrar el nodo
en cuestion
  }
 };
};

#endif /* llist_h */
```

Árbol de Búsqueda Binaria

```
//
//  bstree.h
//

#ifndef bstree_h
#define bstree_h

#include <iostream>

// Nodos del arbol:
template <typename T>
```

```
class bstnode
{
public:
 // Esta clase es usada por otras clases. Para mayor eficiencia, los
atributos se hacen publicos.
 T key;
 bstnode<T> *p, *left, *right;
  // Constructor por omision
 bstnode() {
 };
  // Inicializacion de datos miembro
 bstnode(const T& k, bstnode<T> *w = nullptr, bstnode<T> *y = nullptr,
bstnode<T> *z = nullptr):key(k), p(w), left(y), right(z) {};
  ~bstnode() {
 };
};

// Arbol de busqueda binario:
template <typename T>
class bstree
{
public:
 bstnode<T> *root;    // raiz del arbol

 bstree() : root(nullptr) {}; // settear la raiz en nulo
  ~bstree() {
   clear(root); // borrado recursivo
 };

 void clear(bstnode<T> *node) {
   if (node != nullptr) { // si el nodo no es nulo
     clear(node->left); // borre el izquierdo
     clear(node->right); // y el derecho
     delete node; // cuando se terminen los llamados recursivos va a ir
eliminando cada 1
   }
 }
  void Insert(bstnode<T>* z) {
   bstnode<T> *y = nullptr; // auxiliar
```

```
  bstnode<T> *x = root;

  while (x != nullptr) {
    y = x;
    if (z->key < x->key) { // menores a la izquierda
      x = x->left;
    } else { // mayores o iguales a la derecha
      x = x->right;
    }
  }

  z->p = y;
  if (y == nullptr) {
    root = z; // el árbol estaba vacío
  } else if (z->key < y->key) { // menores a la izquierda
    y->left = z;
  } else {
    y->right = z; // mayores a la derecha
  }
};
 void InorderWalk(bstnode<T> *x) {
  if (x != nullptr) {
    InorderWalk(x->left);
    std::cout << x->key << std::endl;
    InorderWalk(x->right);
  }
};
 bstnode<T>* Search(bstnode<T> *x, const T& k) {
  if (x == nullptr || k == x->key) { // ya no hay mas elementos por
buscar o ya se encontro el deseado
    return x;
  }

  if (k < x->key) { // si es menor busque a la izquierda
    return Search(x->left, k);
  } else { // si es mayor o igual busque a la derecha
    return Search(x->right, k);
  }
};
```

```
 bstnode<T>* IterativeSearch(bstnode<T> *x, const T& k) {
  while (x != nullptr && k != x->key) { // deja de buscar cuando se le
acaben los elemtnos por buscar o ya haya encontrado el deseado
    if (k < x->key) {
      x = x->left; // buscar la izquierda los menores
    } else {
      x = x->right; // buscar a la derecha los mayores o iguales
    }
  }
  return x;
};
 bstnode<T>* Minimum(bstnode<T> *x) {
  while (x->left != nullptr) {
    x = x->left; // los menores siempre van a estar a la izquierda
                 // entonces busque hasta el ultimo
  }
  return x;
};
 bstnode<T>* Maximum(bstnode<T> *x) {
  while (x->right != nullptr) {
      x = x->right; // Los mayores o iguales siempre van a estar a la
                    // derecha entonces busque hasta el ultimo
  }
  return x;
};
 bstnode<T>* Successor(bstnode<T> *x) {
  if (x->right != nullptr) { // si el derecho no esta vacio, el sucesor
va a
                             // estar en su lado derecho, que es como un
subarbol
    return Minimum(x->right); // entonces basta con buscar el minimo de
ese subarbol
  }

  // si el elemento que se le busca el sucesor no tiene nodo derecho,
significa que
  // no tiene sucesor o esta arriba de el

  bstnode<T> *y = x->p; // guardar en y el padre de x
```

```
   while (y != nullptr && x == y->right) { // cuando esta condicion ya no
se cumpla, va a retornar nullptr por que no encontro sucesor o el sucesor
en y
     x = y; // ir recorriendo hacia arriba
     y = y->p; // ir recorriendo hacia arriba
   }
   return y;
 };
  void Delete(bstnode<T>* z) {
   bstnode<T> *y; // auxiliar
   bstnode<T> *x;

   if (z->left == nullptr) { // subarbol izquierdo vacio
     exchange(z, z->right);
   } else if (z->right == nullptr) { // subarbol derecho vacio
     exchange(z, z->left);
   } else { // tiene ambos subarboles
     y = Minimum(z->right); // hallar el minimo del arbol derecho de z
     if (y->p != z) { // si el papa del minimo del arbol derecho de z no
es z
       exchange(y, y->right); // remplaza y con su subarbol derecho
       y->right = z->right; // conecta el subarbol derecho de z con y
       y->right->p = y; // padre del subarbo derecho de z a y
     }
     exchange(z, y); // remplazar z con y
     y->left = z->left; // conectar el subarbol izquierdo con z y y
     y->left->p = y; // actualizar el apdre del subarbol izquierdo de z a
y
   }

   delete z; // liberar el elemento que se queria eliminar
 };

 void exchange(bstnode<T>* u, bstnode<T>* v) {
   if (u->p == nullptr) { // si u es la raiz
     root = v; // remplaza la raiz por v
   } else if (u == u->p->left) { // si u es el hijo izquierdo de su padre
     u->p->left = v; // remplaza el izquierdo con el v
   } else { // si u es el hijo derechod e su padre
```

```
      u->p->right = v; // lo remplaza con el derecho
    }
    if (v != nullptr) {
      v->p = u->p; // el padre de v ahora es el padre de u
    }
  }
};

#endif /* bstree_h */
```

Archivo con cuadro de valores y gràficos

Tarea Programada 2 - C28038 - CI-0116

Referencias

Cormen, T., Leiserson, C., Rivest, R., Stein, C. Introduction to Algorithms: The MIT Press

Material visto en clase