Tarea Programada

Diccionario/Map

Descripción:

Los arreglos asociativos, también llamados diccionarios o mapas, son una estructura de datos abstracta que se utiliza para enlazar parejas de valores llave->valor (key->value) en dónde el diccionario tiene para cada llave un único valor asociado en su interior. De esta manera es posible mapear y encontrar valores rápidamente utilizando una llave.

Al ser una estructura de datos abstracta, no existe una manera específica de implementarla, sino que cada programador puede elegir la implementación que más beneficie a su programa. Actualmente, una estructura diccionario/mapa es parte de la biblioteca estándar (#include <map>) dónde su implementación se encuentra hecha por medio de un árbol rojo y negro. Los árboles rojo y negro son uno de los dos tipos de árbol autobalanceados más populares, el motivo para ello es que poseen rápidos tiempos de inserción y al ser árboles autobalanceados mantienen relativamente la misma profundidad en todos sus nodos lo que conlleva tiempos de acceso cercanos a $O(log_2 n)$.

El otro tipo de árbol autobalanceado más popular es el árbol Adelson-Velskii y Landis, conocido bajo el nombre “árbol AVL”. Este árbol presenta peores tiempos al realizar inserciones en comparación al árbol rojo y negro (aunque ambos se encuentran en los tiempos de duración de $O(log_2 n)$), sin embargo, su nivel de balance es superior en comparación al árbol rojo y negro lo que conlleva que mayor cantidad de sus tiempos de acceso se acerquen al ideal $O(log_2 n)$. Esto significa que cuando se enfrenta un problema que requiere significativamente más accesos que inserciones a un árbol puede resultar una buena opción utilizar árboles AVL.

Los árboles AVL son árboles binarios similares a los que hemos estudiado hasta ahora: cada nodo posee un hijo derecho y un hijo izquierdo. Sin embargo, se diferencian en que cada nodo almacena adicionalmente la información relacionada a su altura. Además, al terminar de realizar una inserción en el árbol, todos los nodos utilizados en el recorrido de la inserción (empezando por el último en recorrerse, como una pila) deberán actualizar sus alturas y, de encontrarse una diferencia mayor a 1 entre las alturas de sus dos hijos, procederán a utilizar una de cuatro rotaciones posibles para rebalancearse.

Estas rotaciones que ejecuta el árbol AVL garantizan que, para cada nodo, la diferencia máxima entre la altura de sus hijos será de 1 y así el árbol se mantendrá relativamente balanceado:

- En el mejor de los casos un árbol de altura *h* puede almacenar un máximo de $2^h - 1$ datos.
- En el peor de los casos un árbol de altura *h* puede almacenar un mínimo de $1 + f(h-1) + f(h-2)$ datos tal que $f(0) = 0$ y $f(1) = 1$.
- De esta manera para un árbol de altura h=4 podría contar con un mínimo de 7 datos almacenados y un máximo de 15.
- La altura h se puede interpretar como la cantidad máxima de accesos necesarios para encontrar un dato o determinar que dicho dato no se encuentra en la estructura.

Estos tiempos de acceso se vuelven significativos con árboles más grandes: por ejemplo, con h = 30 nos encontraríamos con un árbol que posee entre 2 178 308 y 1 073 741 823 datos. Es decir, con solo 30 accesos podríamos acceder a un dato entre 2 millones (peor caso) y 1 millardo de datos (mejor caso).

¿Cómo funciona el árbol AVL?

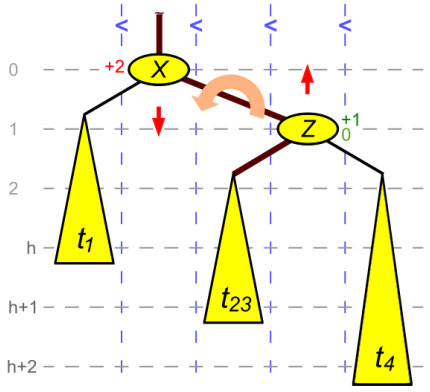
1. Al insertar un nuevo valor en el árbol se procede como se haría en un árbol binario regular: recorriendo desde la raíz del árbol hasta encontrar el nodo nulo dónde debería ir el valor a insertar y se genera el nuevo nodo.
2. Una vez que se inserta el nuevo elemento ocurre el rebalanceo del árbol.
3. Para cada uno de los elementos recorridos para llegar al punto de inserción debe ocurrir lo siguiente:
    a. Si la diferencia de altura entre ambos hijos es mayor a 1, debe aplicarse la rotación adecuada.
    b. Existen 4 tipos de rotaciones diferentes que se explicarán más adelante.
    c. Las rotaciones alteran el orden de los nodos, por lo que deben actualizarse los punteros necesarios para mantener el orden en el árbol.
    d. Después de aplicada la rotación y actualizar los punteros, para los nodos afectados por la rotación se recalculan sus alturas (máximo entre los dos hijos + 1).

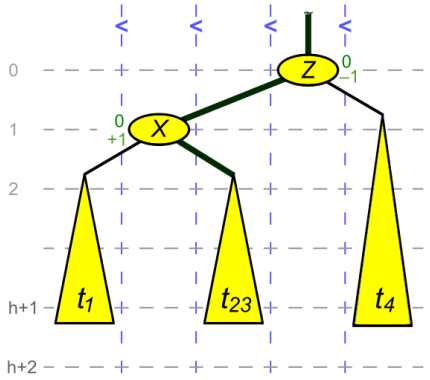
¿Qué tipos de rotaciones hay?

Rotación simple derecha:

Ocurre cuando el desbalance es ocasionado por el hijo derecho del hijo derecho. En el ejemplo, el hijo derecho (t4) del hijo derecho (Z) sufrió una inserción y ahora posee altura h+2. Z posee una diferencia de alturas de 1 y está balanceado. Sin embargo X posee una diferencia de alturas de 2. Por lo tanto X deberá realizar una rotación simple a la derecha:

- Z dejará de ser hijo izquierdo de X, el nuevo hijo izquierdo de X será el subárbol t23, ahora X posee una diferencia de alturas de 0: t1 y t23 tienen la misma altura.
- t23 dejará de ser hijo izquierdo de Z (ahora es hijo izquierdo de X), el nuevo hijo izquierdo de Z será X, ahora Z posee una diferencia de alturas de 0: el nuevo subárbol X y t4 miden lo mismo.
- El antiguo padre de X ahora deberá apuntar a Z, dado que Z se convirtió en su nuevo hijo derecho o izquierdo (dependiendo de si X era hijo derecho o izquierdo de su padre).





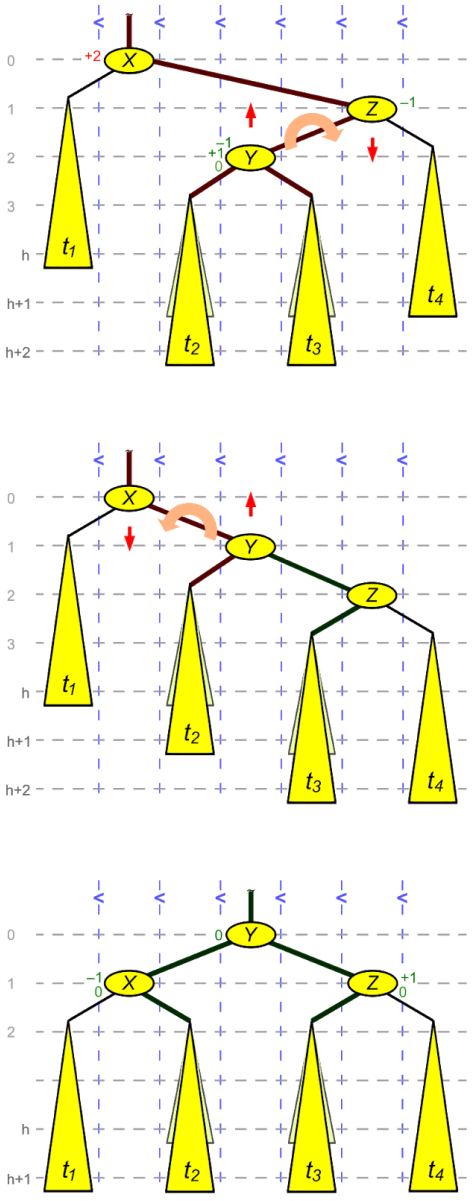
Rotación simple izquierda:

Ocurre cuando el desbalance es ocasionado por el hijo izquierdo del hijo izquierdo. Esta transformación es simétrica a la de la rotación simple derecha.

Rotación doble derecha:

Ocurre cuando el desbalance es ocasionado por el hijo izquierdo del hijo derecho. En el ejemplo el hijo t2 o el hijo t3 del nodo Y sufrió una inserción. El nodo Y sigue balanceado y el nodo Z también, sin embargo, el nodo X posee un desbalance de 2. Por lo tanto X debe realizar una rotación doble derecha:

- Z realiza una rotación simple izquierda: t3 pasa a ser hijo izquierdo de Z, y Z pasa a ser hijo derecho de Y. El nuevo hijo derecho de X es Y.
- X realiza una rotación simple derecha: t2 pasa a ser hijo derecho de X, y X pasa a ser hijo izquierdo de Y. El nuevo hijo izquierdo/derecho del padre de X es Y.



Rotación doble izquierda:

Ocurre cuando el desbalance es ocasionado por el hijo derecho del hijo izquierdo. Esta transformación es simétrica a la de la rotación doble derecha.

Después de realizadas las rotaciones y recalcular el balance para los nodos recorridos en la inserción (y aquellos rotados), se completa la inserción y se garantiza que el árbol se mantiene balanceado: la máxima diferencia de altura entre dos subárboles es de 1.

¿Cómo se pueden utilizar árboles para hacer diccionarios?

Para utilizar árboles como diccionarios lo que se necesita es que cada nodo del árbol tenga dos valores: un valor key y un valor value. El valor key es el que se utiliza para realizar la selección de valores: al insertar un nuevo valor o buscar un valor dentro del árbol, el valor key es el que se utiliza con fines de comparación. El valor value es el que se retorna al realizar una búsqueda dentro del árbol.

De esta manera la descripción de dicho árbol se puede ver como Arbol<K,V> donde K es el tipo de dato de las llaves y V el tipo de datos de los valores.

Tarea programada:

En esta tarea el objetivo consiste en implementar de manera adecuada una clase Map<K,V> que utilice en su interior una estructura de soporte de árbol AVL para manejar las inserciones y las búsquedas. De esta manera su estructura diccionario deberá contener los métodos:

- Map(): Constructor de la clase, inicia el diccionario vacío.
- ~Map(): Destructor de la clase, elimina el diccionario sin fugas de memoria.
- int getSize(): Retorna la cantidad de elementos en el diccionario.
- int count(): Cuenta la cantidad de nodos en el diccionario de manera recursiva.
- void insert(K key, V value): Inserta el elemento value con la llave key.
- V get(K key): Retorna el valor asociado a la llave key, de no encontrarse el elemento retorna 0.

Además será necesario crear una clase Iterator que deberá encontrarse anidada en la clase Map. Las clases Iterator anidadas son usuales en la implementación de estructuras de datos y permiten la iteración sobre la estructura de una manera ordenada. Esencialmente, las clases iteradoras lo que poseen es el mínimo de información necesaria para almacenar su ubicación actual dentro de la estructura de datos sobre la cual iteran y la capacidad para desplazarse en la misma de manera ordenada. Su clase Map debe poseer los siguientes métodos con respecto a la clase Iterator:

- Iterator begin(): Retorna un objeto de la clase Iterator ubicado en el primer elemento de la estructura.
- Iterator end(): Retorna un objeto de la clase Iterator ubicado al final de la estructura (no el último elemento, sino en un estado de “detenido” después del último elemento).

A su vez la clase Iterator debe contar con los siguientes métodos:

- Iterator &operator++(): Sobrecarga del operador ++ prefijo, avanza el iterador al siguiente elemento de la estructura.

- bool operator!=(const Iterator &it) const: Permite comparar dos objetos Iterator y saber si se encuentran ubicados en el mismo elemento.
- K key(): Retorna la llave del elemento en el que se encuentra el iterador.
- V value(): Retorna el valor del elemento en el que se encuentra el iterador.

Al contar con esta estructura iterador es posible realizar un ciclo como el siguiente:

```
for(Map::Iterator it = map->begin(); it != map->end(); ++it){
      cout << it.key() << “: “ << it.value() << endl;
}
```

Que imprimiría de manera ordenada todos los elementos del diccionario, aprovechando la estructura iterador. Es recomendable que la estructura iterador no posea atributos alocados en memoria dinámica, por riesgo a fugas en caso de mal manejo del mismo y para que los iteradores puedan ser utilizados como variables locales con facilidad.

Implemente el diccionario solicitado y haga un código main para probar lo siguiente:

Inicialice un objeto Map<unsigned int, const char* str> diccionario.

Inserte los siguientes objetos a su diccionario, en el orden mostrado:

26, “manzana”
4, “banco”
2, “abeja” // Prueba rotación simple izquierda
34, “quetzal”
40, “trampa” // Prueba rotación simple derecha
28, “norte” // Prueba rotación doble derecha
16, “guante”
8, “dado”
24, “lambda”
0, “NIL”
25, “lobo” // Prueba rotación doble izquierda

Luego utilice un iterador para recorrer el diccionario, imprima tanto las llaves como los elementos. Compare a su vez el resultado de getSize y count que deberían obtener el mismo resultado.

Pueden trabajar en parejas.