Universidad de San Carlos de Guatemala

Centro Universitario de Occidente

División de Ciencias de la Ingeniería

Manual Tecnico

Sistema de contactos en consola con C++

Estructura de datos

02/04/2024

202031953 - Hania Mirleth Mazariegos Alonzo

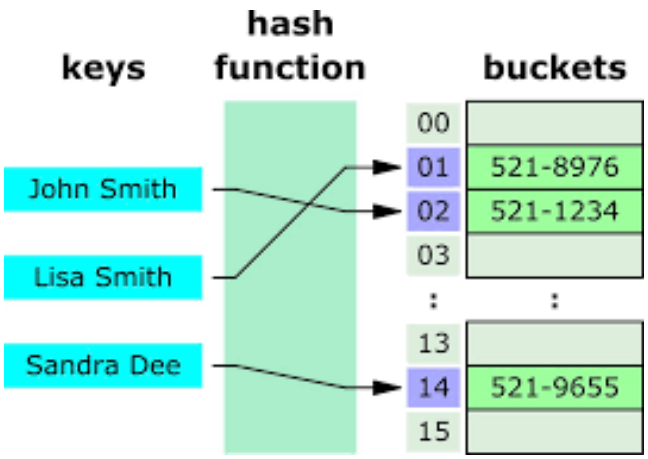
Introducción

En este manual técnico, se describirá la implementación del sistema de almacenamiento y gestión de contactos en consola utilizando C++. Se utilizaron listas, tablas hash y árboles AVL como estructuras de datos, para ello fue indispensable el uso de punteros y clases genéricas. Además, se usa un archivo Makefile para compilar el proyecto.

Según los requerimientos del problema, el sistema debía almacenar los grupos en una tabla hash, siendo las llaves los nombres de los grupos y los valores apuntarían cada uno a una tabla hash con los campos de dicho grupo, estas segundas tablas hash de campos tienen como llave el nombre del campo y como valor un árbol AVL en el que se almacenan los datos de cada contacto para el campo al que pertenecen.

Tablas Hash

Las tablas hash son una estructura de datos que permite el almacenamiento y recuperación eficiente de información. Funcionan mediante una técnica llamada "hashing", que asigna valores a claves a través de una función de hash. Esta función toma una clave como entrada y devuelve una posición en la tabla.



En una tabla hash, los datos se almacenan en "buckets" o "casillas" que están indexadas por los valores generados por la función de hash. Cuando se necesita acceder a un dato, la función de hash se utiliza para calcular la ubicación del bucket donde se espera encontrar el dato. Esto permite un acceso rápido a los datos, ya que el tiempo de búsqueda es constante en promedio, O(1), independientemente del tamaño de la tabla.

Sin embargo, las colisiones pueden ocurrir cuando dos claves diferentes tienen el mismo valor hash y deben ser asignadas al mismo bucket. Hay varias técnicas para manejar colisiones, como la resolución por encadenamiento (donde se mantiene una lista en cada bucket) o la resolución por sondaje (donde se busca otro bucket disponible).

Las tablas hash son utilizadas en una amplia variedad de aplicaciones, incluyendo bases de datos, algoritmos de búsqueda, y la implementación de estructuras de datos como conjuntos y diccionarios en muchos lenguajes de programación. Ofrecen un equilibrio entre el tiempo de acceso rápido y el uso eficiente de la memoria.

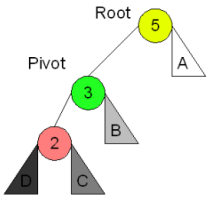
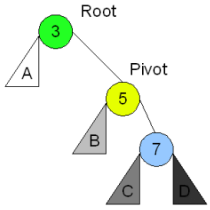
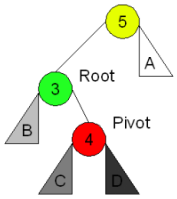
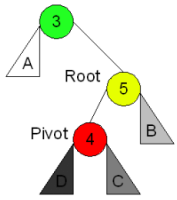
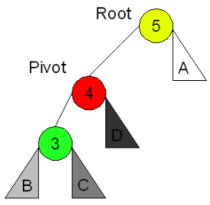
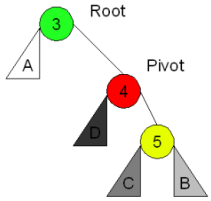
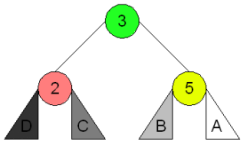
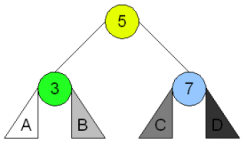
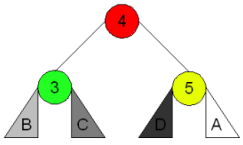
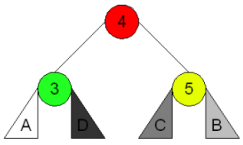
Arboles AVL

Los árboles AVL son una forma de árbol binario de búsqueda que se caracteriza por mantenerse balanceado automáticamente después de cada inserción o eliminación de nodos. Fueron inventados por Adelson-Velsky y Landis en 1962, de ahí su nombre. El balanceo se logra manteniendo una propiedad llamada "altura balanceada" o "balanceo AVL".

La altura balanceada se refiere a que la diferencia de alturas entre los subárboles izquierdo y derecho de cualquier nodo en el árbol no puede ser mayor que 1. Esto significa que para cualquier nodo, la diferencia de alturas entre su subárbol izquierdo y su subárbol derecho debe ser 0, 1 o -1. Si esta propiedad se mantiene para todos los nodos del árbol, se garantiza que la altura del árbol esté en el orden de logarítmico en el peor caso, lo que garantiza un rendimiento eficiente en operaciones como búsqueda, inserción y eliminación.

Cuando se realiza una inserción o eliminación que puede desequilibrar el árbol, se ejecutan rotaciones simples o dobles para restaurar el balance. Estas rotaciones reorganizan los nodos del árbol de manera que la propiedad de altura balanceada se mantenga.

Los árboles AVL son útiles en situaciones donde se requiere un rendimiento eficiente en operaciones de búsqueda, inserción y eliminación, ya que garantizan un tiempo de ejecución de estas operaciones en el orden de O(log n) en el peor de los casos. Sin embargo, el mantenimiento del balance puede requerir un costo adicional en términos de tiempo y espacio, especialmente durante las inserciones y eliminaciones, en comparación con árboles binarios de búsqueda no balanceados.

Left Left Case	Right Right Case	Left Right Case	Right Left Case
 **Right Rotation**	 **Left Rotation**	 **Left Rotation**	 **Right Rotation**
		 **Right Rotation**	 **Left Rotation**
			

Definición de clases:

HashTable<K, V>

Parámetros

K: Tipo de datos para la clave.

V: Tipo de datos para el valor.

Atributos

int **size**: Tamaño del arreglo de Pares de la tabla.

Pair<K, V> ***table**: Arreglo de pares clave-valor, la tabla en sí.

Métodos:

int **hashFunction**(K key):

Parámetro K key: clave sobre la que se aplica la función hash.

Descripción: Función de hash que utiliza el método de multiplicación. Multiplica el valor hash de la llave por una constante real entre 0 y 1 (la inversa de la razón áurea en este caso) luego determina la parte decimal de esta multiplicación para luego multiplicar esta parte decimal por el tamaño del arreglo para obtener y retornar un índice disperso dentro del rango del arreglo.

Retorna: índice generado en base a la llave dada.

bool **hasKey**(K key):

Parámetro K key: clave a verificar

Descripción: Función que verifica si una clave ya existe en la tabla.

Retorna: **true** Si la clave existe en la tabla, **false** Si la clave no existe en la tabla.

void **rehash**():

Descripción: Realiza el re-hashing de la tabla cuando se excede el factor de carga del 60% o en caso de colisiones. Genera un nuevo arreglo con un tamaño 40% mayor que el del arreglo anterior y vuelve a insertar todos los pares clave-valor del arreglo previo en el nuevo.

void **insertP**() noexcept(false):

Parámetro K key: clave a insertar

Parámetro V value: valor asociado a la clave

Throw std::invalid_argument: Si la clave ya existe o está vacía.

Descripción: Inserta un par clave-valor en la tabla hash. Primero verifica el factor de carga de la tabla y en caso de colisiones hará un re-hashing hasta que no haya más colisiones

V **get**(K key) noexcept(false):

Parámetro K key: clave de la que se quiere obtener el valor asociado.
Retorna V value: valor asociado a la clave.
Throw std::invalid_argument: Si la clave no se encuentra en la tabla.
Descripción: Obtiene el valor asociado a una clave.

Node<T>

Atributos:

Node<T> ***left**:Puntero al hijo izquierdo del nodo.

Node<T> ***right**: Puntero al hijo derecho del nodo.

Node<T> ***sup**: Puntero al nodo padre del nodo.

int **bf**: Factor de balance del nodo.

int **hl**: Altura del subárbol izquierdo.

int **hr**: Altura del subárbol derecho.

int **id**: Número identificador del nodo.

T **value**: Valor almacenado en el nodo.

List<std::string> ***list**: Puntero a una lista de cadenas asociadas al nodo.

Tree<T>

Parámetros

T: Tipo de datos para el valor de cada nodo

Atributos

Node<T> *root: Nodo raíz del árbol.

int **nodes**: Contador del número de nodos en el árbol.

Métodos:

int **calcHeightsAndBf**(Node<T> *node);

Parámetro Node<T> *node: nodo sobre el que se calcula

Descripción: Función recursiva que calcula la altura de un nodo, para ello calcula la altura de sus nodos hijos, derecho e izquierdo y actualiza el factor de balanceo de cada nodo.

Retorna: altura general del nodo

void **balance**(Node<T> *node);

Parámetro Node<T> *node: nodo a balancear

Descripción: Función recursiva que balancea todos sub árboles a partir del nodo indicado, calcula las alturas y el factor de balanceo de cada nodo y evalúa si es necesario aplicar una rotación.

void **rotateRR**(Node<T> *node);

Parámetro Node<T> *node: nodo sobre el que se rota

Descripción: Realiza una rotación hacia la derecha-derecha sobre el nodo indicado.

void **rotateLL**(Node<T> *node);

Parámetro Node<T> *node: nodo sobre el que se rota

Descripción: Realiza una rotación hacia la izquierda-izquierda sobre el nodo indicado.

void **rotateLR**(Node<T> *node);

Parámetro Node<T> *node: nodo sobre el que se rota

Descripción: Realiza una rotación hacia la izquierda-derecha sobre el nodo indicado.

void **rotateRL**(Node<T> *node);

Parámetro Node<T> *node: nodo sobre el que se rota

Descripción: Realiza una rotación hacia la derecha-izquierda sobre el nodo indicado.

void **insertNode**(Node<T> *, Node<T>*);

Parámetro Node<T> *newNode: nodo a insertar

Parámetro Node<T> *tmp: nodo sobre el cual se insertará

Descripción: Función recursiva que compara el valor hash del nuevo nodo con el valor de tmp para saber si se convierte en su hijo derecho o izquierdo, de ya haber un nodo hijo en la posición indicada se vuelve a evaluar tomando a ese hijo como el nuevo tmp, hasta que la posición esté vacía.

void **insert**(T, List<string>*);

Parámetro T newValue: valor del nuevo nodo

Parámetro List<string> *newList: puntero a la lista del nuevo nodo

Descripción: Registra el nuevo nodo y llama a la función insertNode(Node<T> *, Node<T>*) para ubicar al nodo donde corresponde desde la raíz y luego al metodo balance(Node<T> *).

List<List<string>*>* **get**(T);

Parámetro T value: valor a comparar

Descripción: Una lista de las listas asociadas con los nodos que coinciden con el valor dado, para ello usa la función search(Node<T> *, T, List<List<string>*>*).

Retorna List<List<string>*>*: Listado de listas de cadenas

List<List<string>*>* **get**();

Parámetro T value: valor a comparar

Descripción: Una lista de las listas asociadas con todos los nodos del árbol

Retorna List<List<string>*>*: Listado de listas de cadenas

void **search**(Node<T> *, T, List<List<string>*>*)

Parámetro Node<T> *node: nodo actual

Parámetro T value: valor a comparar con el del nodo

Parámetro List<List<string>*>*listL: lista sobre la que insertar

Descripción: Función recursiva que recorre el árbol y cuando el valor dado coincide con el valor del nodo agrega la lista de cadenas a la lista pasada como parámetro.

Node<T> * **getRoot**();

Descripción: Devuelve el puntero a la raíz del árbol.

Retorna Node<T> *: Puntero a la raíz del árbol.