Universidad Rafael Landívar

Facultad de Ingeniería

Estructura de Datos II

Sección 01 (matutina)

Ing. Fredy Bustamante

DOCUMENTACIÓN LABORAOTRIO NO.2

ALMACENAMIENTO Y RECUPERACIÓN DE ÁRBOL B+

Sergio Iván Cardona Polanco

Carné: 1222419

Guatemala, 28 de agosto de 2024

DESCRIPCIÓN

El objetivo de este laboratorio es la implementación de la estructura de datos tipo árbol B+, en donde se desarrolló dos programas, el primer programa es para la inserción de los datos (id y nombre) en el árbol donde se puede ver la estructura que este tiene en un archivo que se llama IDX_datos.TXT, además calculamos el tiempo en que se toma desde que lee el archivo de datos hasta que se insertan en la estructura.

El segundo programa se encarga de recuperar los datos almacenados en la estructura del árbol, primero lee el archivo donde se encuentra la estructura del árbol (IDX_datos.TXT) para luego cargarlos en memoria, también se toma el tiempo de lectura del archivo y carga de datos a memoria; luego nos pedirá el “id” que deseamos buscar para luego devolvernos el nombre.

PSEUDOCÓDIGO DE ESCRITURA

```
insert(k, name) {
    Si la raíz está llena (tiene 2t - 1 claves):
        Crear un nuevo nodo temporal vacío (temp)
        Asignar temp como la nueva raíz
        Hacer que temp tenga como primer hijo la antigua raíz
        Llamar a split_child(temp, 0) para dividir la raíz
        Llamar a insert_non_full(temp, k, name) para insertar la clave en el árbol

    Si la raíz no está llena:
        Llamar a insert_non_full(root, k, name) para insertar la clave en la raíz
}

insert_non_full(x, k, name) {
    i = número de claves en x - 1
    Si x es hoja:
        Añadir espacio vacío para la nueva clave en x
        Mientras i >= 0 y k < clave en posición i en x:
            Mover la clave en i a i + 1
            Decrementar i
        Insertar (k, name) en la posición correcta en x
    Si x no es hoja:
        Mientras i >= 0 y k < clave en posición i en x:
            Decrementar i
        Incrementar i para encontrar el hijo correcto
        Si el hijo en la posición i está lleno (tiene 2t - 1 claves):
            Llamar a split_child(x, i) para dividir el hijo
            Si k > clave en la nueva posición i, mover i hacia la derecha
```

```
        Llamar recursivamente a insert_non_full(hijo en posición i, k, name)
}

split_child(x, i) {
    t = grado mínimo del árbol B+
    y = hijo en la posición i de x
    Crear un nuevo nodo z que será la división de y (z es hoja si y es hoja)

    Si y es hoja:
        Mover las claves de la segunda mitad de y a z
        Insertar la primera clave de z en x en la posición i
        Conectar z como el siguiente nodo de y en la lista enlazada de hojas

    Si y no es hoja:
        Mover las claves de la segunda mitad de y a z
        Mover los hijos correspondientes de y a z
        Insertar la clave mediana de y en x en la posición i
}
save_to_file(filename){
    Abrir archivo para escritura
    Llamar a _write_node(raíz, archivo, nivel=0) para escribir cada nodo en el archivo
}
_write_node(nodo, archivo, nivel){
    Escribir nodo con indentación basada en el nivel
    Si el nodo no es hoja:
        Para cada hijo en nodo:
            Llamar a _write_node(hijo, archivo, nivel + 1)
}
```

```
process_file(filename, t){
    Crear un árbol B+ con grado t
    Leer todas las líneas del archivo de entrada
    Inicializar un patrón para detectar comandos de inserción (Insert:{id, nombre})

        Si la línea contiene un comando de inserción:
            Extraer id y nombre
            Medir el tiempo de inicio de la inserción
            Insertar (id, nombre) en el árbol B+
            Medir el tiempo de fin de la inserción
            Calcular el tiempo de la inserción y acumularlo en el total

    Guardar el árbol B+ en un archivo de índice
    Guardar los detalles de la escritura en el log
    Guardar los tiempos totales de inserción y escritura en un archivo de resultados
}
```

PSEUDOCÓDIGO CARGA DE ARCHIVO

load_from_file(filename) {

Iniciar temporizador para medir el tiempo de carga

Abrir archivo `filename` para lectura

Leer todas las líneas del archivo

Crear diccionario `level_nodes` para almacenar nodos según su nivel en el árbol

Inicializar `previous_level` en -1 y `previous_node` en None para ayudar con la vinculación de nodos hoja

Calcular el nivel del nodo basado en la cantidad de espacios en blanco al inicio de la línea

Extraer los datos del nodo eliminando texto extra y obteniendo solo las claves

Si la línea está vacía, continuar con la siguiente iteración

Si los datos del nodo contienen tuplas (lo que indica que es un nodo hoja):

Convertir los datos a una lista de tuplas (clave, nombre)

Si no contienen tuplas (nodo interno):

Convertir los datos a una lista de claves enteras

Crear un nuevo nodo `BPlusNode` (especificar si es hoja o no)

Si el nodo es hoja:

Asignar las tuplas (clave, nombre) a las `keys` del nodo

Si no es hoja:

Asignar las claves enteras a las `keys` del nodo

Si el nivel es mayor a 0:

Buscar el último nodo en el nivel superior

Añadir el nodo actual como hijo del nodo padre

Si el nivel no está en `level_nodes`:

Inicializar una nueva lista para este nivel en `level_nodes`

Añadir el nodo actual a `level_nodes` para su nivel correspondiente

Si el nodo es hoja y el nivel es igual a `previous_level`:

Enlazar el nodo anterior con el nodo actual usando `previous_node.next`

Actualizar `previous_level` y `previous_node` con los valores actuales

Asignar el primer nodo en `level_nodes[0]` como la raíz del árbol

Detener temporizador y calcular el tiempo total de carga

}

CONCLUSIONES

- Si es posible recuperar los datos desde un archivo y cargarlos en la memoria.
- La forma en que se escribe la estructura del árbol en el archivo IDX afecta en la búsqueda de los id.
- El árbol b+ es una de las mejores estructuras para manejar un gran volumen de datos