# Proyecto de base de datos 2

José Leonidas García Gonzaes

15 de mayo del 2020

## 1. Introducción

### 1.1. Objetivo

El objetivo de este proyecto es implementar un B+ tree en disco que permita guardar data genérica y analizar el desempeño de la inserción y búsqueda del B+ tree para la organización de archivos en memoria secundaria.

### 1.2. Definiciones previas

- B+ tree: Es una estructura de datos eficiente para la organización de archivos en disco, especialmente para las consulta en rango.
- Binary Search Tree: Un binary search tree (BST) es una estructura de datos tipo árbol donde cada nodo tiene un hijo derecho e izquierdo.
- Forward list: Es una estructura de datos donde mantenemos un puntero al primer elemento y donde cada elemento apunta al siguiente elemento de la lista.
- Hoja de un B+ tree: Es un nodo del árbol que no tiene ningún hijo.

### 1.3. Resultados esperados

Buscamos una implementación genérica de un B+ tree en disco, esto es, implementaremos un B+ tree usando templates y traits para que el código pueda ser lo más genérico posible y se pueda adaptar a guardar cualquier tipo de dato. Además, esperamos obtener gráficas que representen como varía el tiempo requerido para construir el B+ tree de acuerdo a la cantidad de elementos y de manera similar para la cantidad de lecturas-escritas en disco. Por otro lado, para analizar la búsqueda en el B+ tree obtendremos gráficas de tiempo de ejecución y de cantidad de lecturas en disco.

### 1.4. Marco teórico

### 1.5. B+ tree

Usaremos un B+ tree para la organización de archivos en memoria secundaria. La idea de esta técnica si no consideramos la eliminación de datos no es tan complicada.

En un B+ tree tenemos lo que se conoce como el orden del árbol, la idea es mantener nodos que guarden entre $[BTREE\_ORDER/2, BTREE\_ORDER]$ de elementos en todos los nodos, aunque no necesariamente en la raíz. Así, en cada nodo tenemos $BTREE\_ORDER + 1$ hijos, uno a la izquierda y derecha de cada dato. De manera práctica, podemos ver un B+ tree como un BST generalizado, de este modo la búsqueda en un B+ tree es similar a la de un BST, en cada momento a partir del dato en el que nos encontramos decidimos si ir al hijo de la izquierda o derecha, solo que ahora en cada nodo buscaremos a partir de que dato conviene hacer esa acción. Además, una característica importante del B+ tree es que toda los datos se encuentran en las hojas (posiblemente replicadas en nodos internos) y los nodos donde se encuentran están enlazadas como un forward list, para optimizar las consultas en rango, por eso en la búsqueda siempre buscaremos el elemento en cuestión hasta llegar a una hoja.

Luego, para la inserción de un dato en el B+ tree simplemente tenemos que hacer una búsqueda del elemento en el árbol para obtener la hoja donde se deber+ia inserta el elemento en cuestión, lo agregamos en el nodo y en caso de que el nodo actual ahora tenga más elementos de los permitidos comenzar a hacer un split (dividimos el

árbol en 2 mitades) y subimos un elemento al parent del nodo actual y así de manera recursiva seguimos haciendo el split mientras sea necesario.

## 1.6. Estructura del código

- BPlusNode: Nodo del B+ Tree
- BPlusTree: B+ tree
- Iterator: Iterador sobre los datos del B+ tree
- Macros: Definiciones que entiende la implementación del B+ tree
- Manager: Encargado de la lecturas y escritura de datos en disco

La dependencia de las clases podemos verla en el diagrama 1.

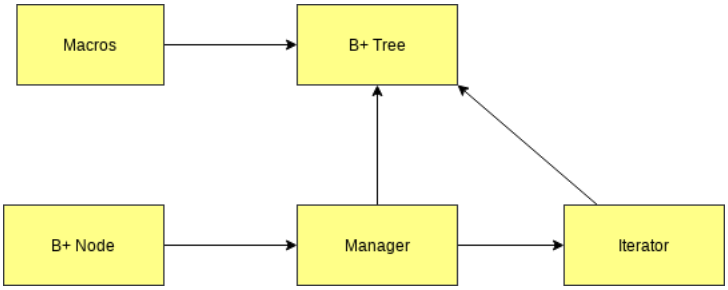


Figura 1: Diagrama de dependencias de la implementación B+Tree

Seguimos esta estructura porque permite abstraer toda la lógica detrás del B+ tree de manera que si alguien quiere usar esta implementación solo debe instancias a la clase BPlusTree y así puede hacer eso de cualquiera de estas funciones:

- Constructor (path del archivo header, path del archivo donde se guarda la data del B+ tree, trunc): Con el constructor podemos instanciar un B+ tree indicando en que archivos se guardaran (o en donde están guardados) la data del B+ tree y el header (la metadata del árbol), además acepta un tercer parámetro para poder indicar si queremos limpiar toda la data
- void insert(value): Inserta un valor en el B+ tree
- bool find(value): Retorna si un valor está en el B+ tree
- Iterator gteq(value): Retorna un iterator al primer elemento que es mayor o igual a un valor
- print: Imprime el árbol
- begin(): Retorna un iterador al elemento de menor valor en el B+ tree
- end(): Retorna un iterator al elemento que le sigue del elemento de mayor valor dentro del B+ tree

Además, el constructor del árbol requiere que se le pase un Trait, en él definimos lo siguiente:

- Tipo de dato que se guardará
- Tipo de dato de la key
- Tipo de dato que se usará para guardar las direcciones de memoria
- Funtor para indicar el tipo de impresión del árbol (en Macros podemos obtener las opciones)
- Funtor para indicar como obtener el key de un dato

Si bien las definiciones que requiere nuestro trait no son pocos, este nos permite tener mayor flexibilidad sobre el código y permite una mayor generalización.

### 1.7. Posibles mejoras

Toda esta estructura facilita el camino para en un futuro adaptar esta implementación para que pueda usar acceso concurrente. En el archivo BPlusTree está diferenciado los métodos que cambian y los que no cambian la estructura del árbol, de esta manera podemos usar por ejemplo un mutex para lograr una implementación al estilo readers and writers, es decir, una implementación que permita siempre lecturas en el árbol pero que solo permita escrituras cuando ningun hilo está leyendo-escribiendo en el árbol. Sin embargo, debido a la complejidad que tomó realizar todo lo anterior se optó por no realizar esta parte del proyecto.

## 2. Resultados experimentales

Para la experimentación se generó data aleatoria siguiendo esta estructura:

```
struct TaxiDriver {
  int dni;
  int edad;
  char nombre[20];
  char empresa[20];
  char modelo_carro[10];
  char placa[10];
  float ingresos;
};
```

Se usó el mismo orden para todos los árboles (1024) y se realizó 3 iteraciones cada vez que se generaba data. Para evitar mucho trabajo computacional ibamos avanzando en escala logatítmica desde 1 hasta $10^5$ elementos.

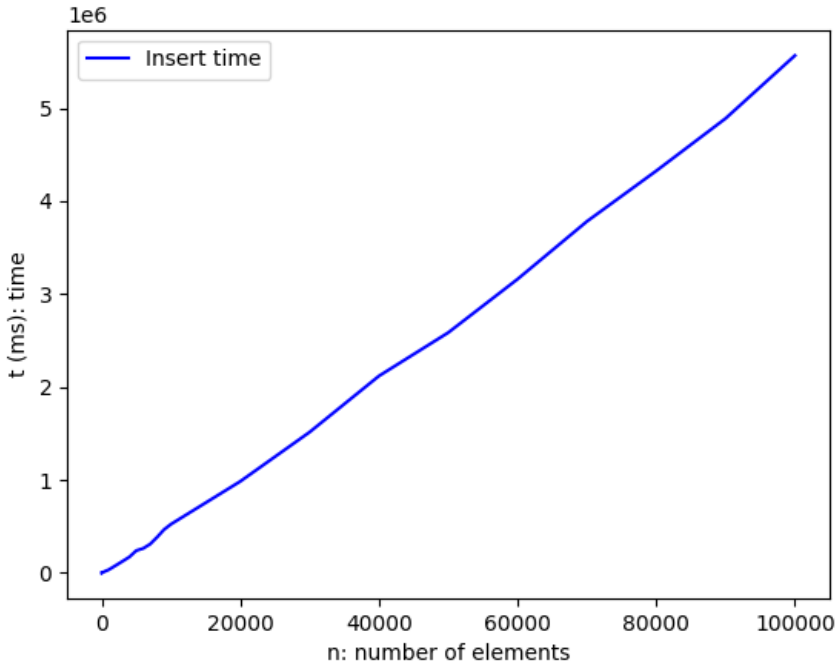


Figura 2: Tiempo de ejecución de la inserción en el B+ tree

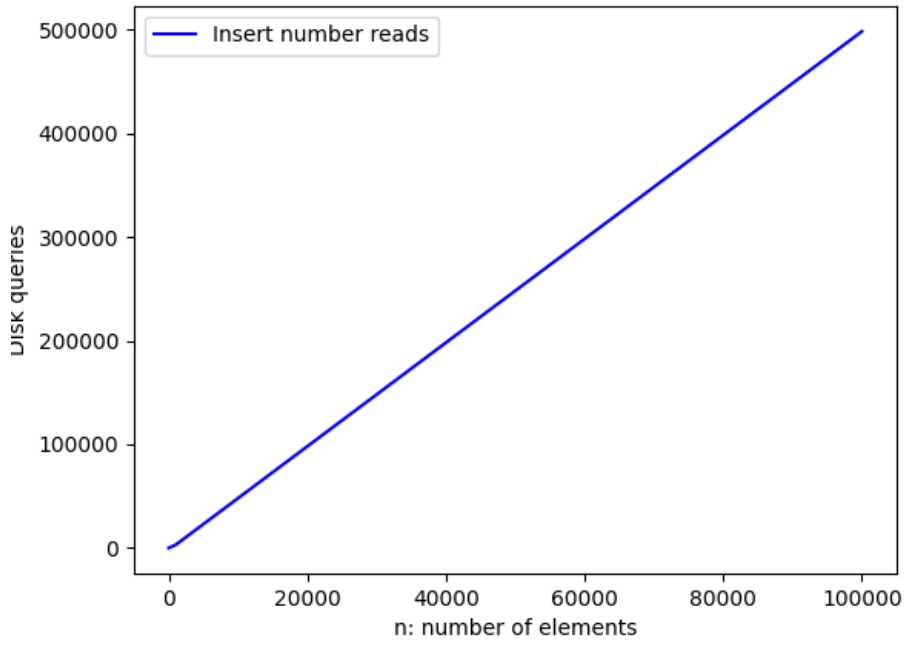


Figura 3: Número de lecturas en el B+ tree durante la inserción

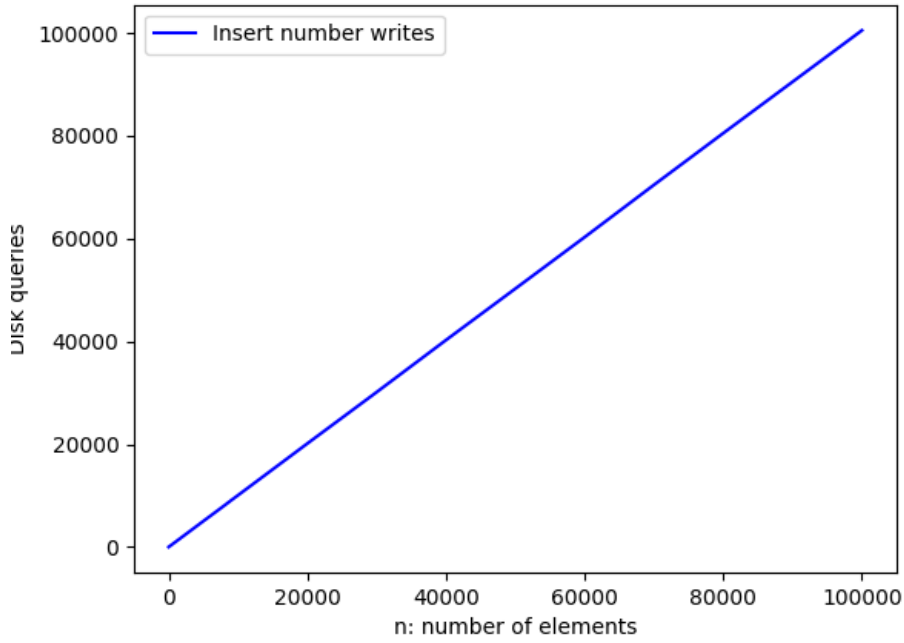


Figura 4: Núemro de escrituras en el B+ tree durante la inserción

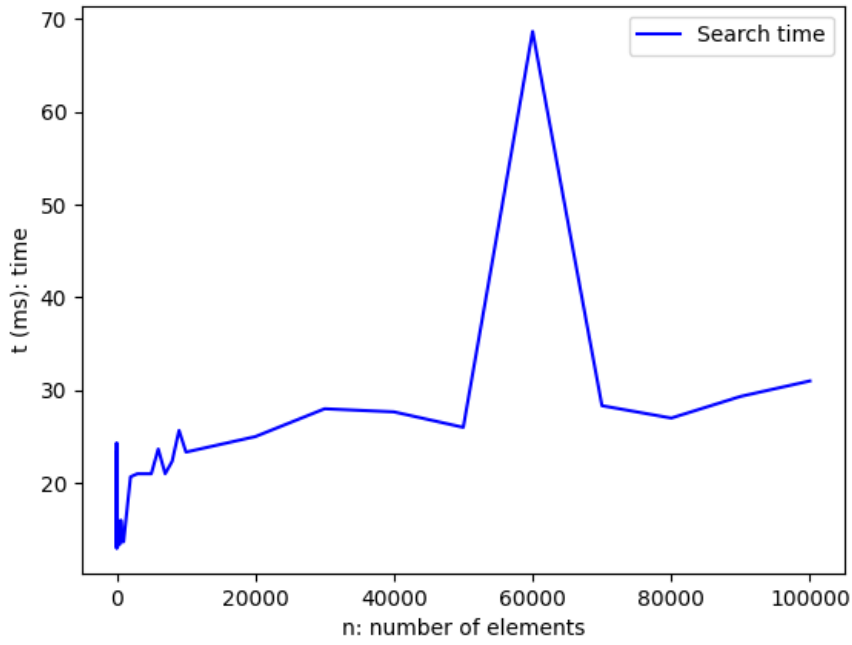


Figura 5: Tiempo de ejecución de la búsqueda en el B+ tree

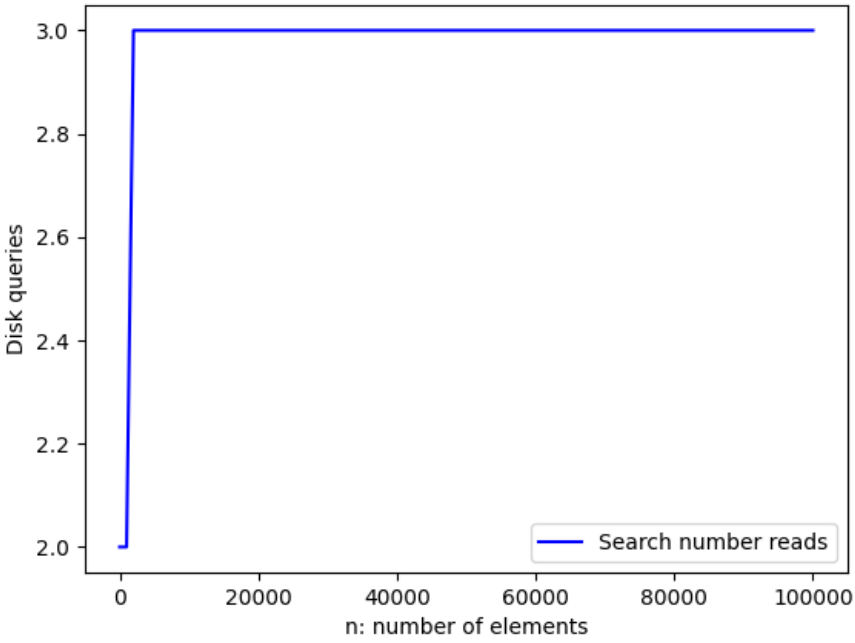


Figura 6: Número de lecturas en el B+ tree durante la búsqueda

# 3. Pruebas de uso

Para probar el programa se usó la librería GTest para poder realizar pruebas a los distintos métodos empleados. La idea es sencilla, probamos 2 configuraciones distintas de traits para distintos tipos de datos, generamos data aleatorias, la insertamos y luego verificamos que nuestra función find logra identificar que el elemento insertado se encuentra el el B+ tree. También hicimos las pruebas con tipos de datos más complejos (structs) y se incluyó un test para verificar la persistencia de los datos. En el README del repositorio (ver Anexos) está especificado como ejecutar los test, con ellos deberíamos obtener un resultado como el siguiente:

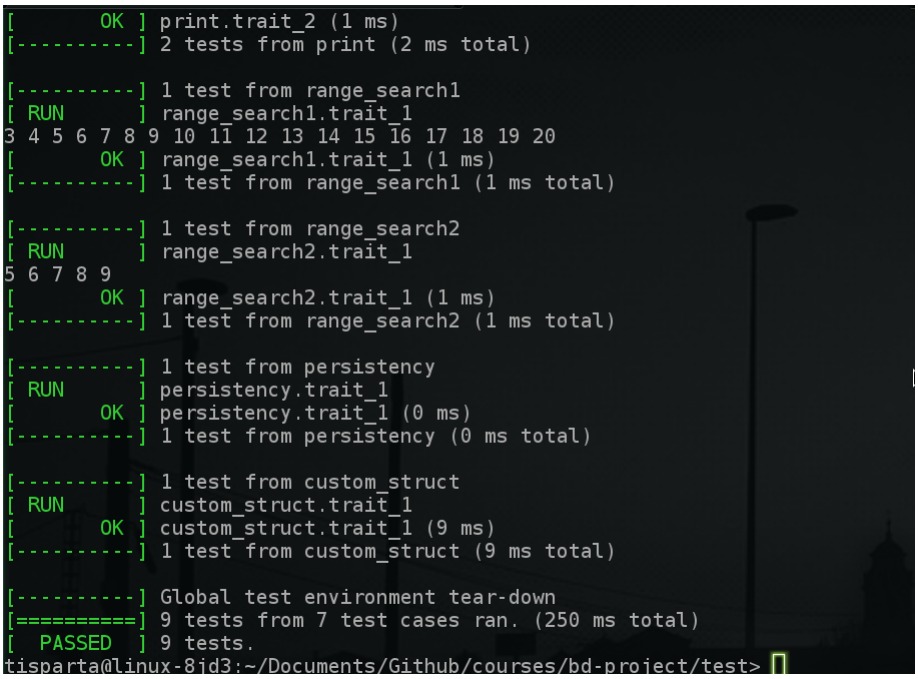


Figura 7: Ejecución de los tests del programa

También, realizamos un programa para mostrar el concepto de como se puede hacer el join de dos B+ trees con el código implementado.

# 4. Conclusión

El B+ tree es una buena estructura de datos para organización de archivos en memoria secuencia ya que puede obtener archivos con muy pocas lecturas a disco aún cuando hay una gran cantidad de elementos en el árbol, además permite buscar en rango eficientes. Por otro lado, a partir de las gráficas de performance de tiempo el incorcoporar una política de cache se vuelve una necesidad al trabajar con una gran cantidad de valores para reducir el tiempo de las consultas.

# 5. Anexos

Link al código fuente del proyecto en Github. En anterior link es al repositorio donde está el código fuente del pryecto, en el README están las indicaciones para poder ejecutarlo y las dependencias necesarias.