Tarea 1
Árboles B y B+ en disco

Profesores: Benjamín Bustos y Gonzalo Navarro
Auxiliares: Diego Arias y Claudio Gaete

1. Indicaciones administrativas

1. Fecha de entrega: jueves 2 de octubre a las 23:59.
2. Grupos de a lo más 3 personas **de la misma sección**.
3. Lenguaje a utilizar: C, C++ o Java.

2. Contexto

El objetivo de esta tarea es implementar un árbol B y un árbol B+ en memoria externa, que permitan la inserción de pares llave-valor y búsquedas según un rango de llaves.

Se compararán los tiempos de creación y búsqueda de ambos árboles usando datos reales de temperatura de la Estación Quinta Normal, en Santiago, entre enero de 2019 y julio de 2025. Los datos se pueden descargar desde el siguiente enlace. Las mediciones de temperatura están guardadas desordenadamente en un archivo binario, como pares llave-valor: un `int` (4 bytes) indicando el tiempo Unix en el cual se tomó la medición, y un `float` (4 bytes) indicando la temperatura medida en ese instante.

3. Árboles B y B+

Para efectos de esta tarea, ambos tipos de árboles utilizarán la misma estructura de nodo, que permite un máximo de b elementos en su interior. Los campos de cada nodo serán los siguientes:[1]

- `es_interno`: un `int` (4 bytes) que indica si el nodo es interno o un nodo externo (hoja).
- `k`: un `int` (4 bytes) que guarda la cantidad de pares llave-valor actualmente contenidas en el nodo. En todo momento, $\frac{b}{2} - 1 \leq k \leq b$.[2]
- `llaves_valores`: un arreglo de b pares llave-valor, con llaves de tipo `int` (4 bytes), y valores de tipo `float` (4 bytes).
  - Noten que el arreglo es de tamaño **fijo** b, aunque el nodo tenga menos elementos (simplemente no nos fijaremos en los elementos después del k-ésimo).

[1] Esto es una simplificación; en la realidad, los nodos internos de un árbol B+ solo guardan llaves, no pares llave-valor, pues los pares completos están replicados en las hojas. También se puede aumentar la eficiencia eliminando el campo de `hijos` para los nodos externos, permitiendo guardar una mayor cantidad de pares llave-valor. Sin embargo, no se pedirá implementar esto en la tarea, pues se prefirió que todos los nodos tuvieran la misma estructura.

[2] La cantidad mínima de pares permitidos al interior de cada nodo depende de la implementación del algoritmo; en este caso, la implementación divide nodos antes de que sea estrictamente necesario, por lo cual la cantidad mínima es $\frac{b}{2} - 1$ y no $\frac{b}{2}$. Para más información, se puede ver la pág. 493 de Introduction to Algorithms.

  - Los k pares llave-valor guardados en `llaves_valores` deben estar siempre ordenados de manera ascendiente.
- `hijos`: un arreglo de $b+1$ enteros de tipo `int` (4 bytes), que guarda las posiciones en disco de los $k+1$ hijos del nodo.
  - Nuevamente, el arreglo es de tamaño fijo; los espacios posteriores al $(k+1)$-ésimo serán ignorados.
  - El nodo `hijos[i]` guarda únicamente valores cuya llave es mayor a la llave de `llaves_valores[i-1]`, y menor o igual a la llave de `llaves_valores[i]`. Esta es la invariante que permite buscar de manera sencilla en este tipo de árboles.
  - Si el nodo es externo, este arreglo será ignorado completamente, aunque **debe seguir siendo parte de la estructura**.
- `siguiente`: un entero de tipo `int` (4 bytes), que solo se utiliza en las **hojas de los árboles B+** (para cualquier otro nodo, se fija este valor como -1). Guarda la posición en disco de la hoja siguiente a la actual, lo cual se utiliza en las búsquedas por rango del árbol B+.

Para esta tarea, se utilizará $b = 340$ para que cada nodo tenga un tamaño exacto de 4096 bytes (el tamaño típico de un bloque en disco).

3.1. Utilización de memoria externa

Los árboles B y B+ se guardan de manera **serializada** en un archivo binario: cada bloque del disco contiene exactamente un nodo, y cada nodo padre guarda las posiciones en disco de sus nodos hijos (para poder acceder a ellos cuando necesite). Sin embargo, tener el árbol completamente en disco implica que cada inserción requiere múltiples lecturas; realizar las millones de inserciones que se requieren en la tarea toma un tiempo excesivo. Por ende, para la **creación** del árbol B/B+, se simulará la serialización de la siguiente forma:

- El árbol será representado como un arreglo (o vector) de nodos en RAM; el primer elemento del arreglo siempre será el nodo raíz.
- Cada nodo interno guardará **el índice** en el cual está ubicado cada uno de sus hijos.
- Cada vez que se acceda al arreglo de nodos para leer un elemento, se considerará como **una lectura**.
- Cada vez que se escriba un elemento en el arreglo (ya sea modificando una posición existente, o agregando un nuevo elemento al final), se considerará como **una escritura**.

Un diagrama de esta representación se puede apreciar en la Figura 1. Una vez se hayan insertado todos los pares, este árbol se guardará en disco escribiendo cada uno de los nodos de forma secuencial a un archivo binario. Las búsquedas por rango se deben hacer **leyendo desde el archivo binario**, que en términos de código es muy similar a leer desde el arreglo en RAM, solo que en vez de acceder al elemento i de un arreglo, tenemos que leer desde la posición $i \cdot$ `(tamaño de un nodo en bytes)` del archivo binario.

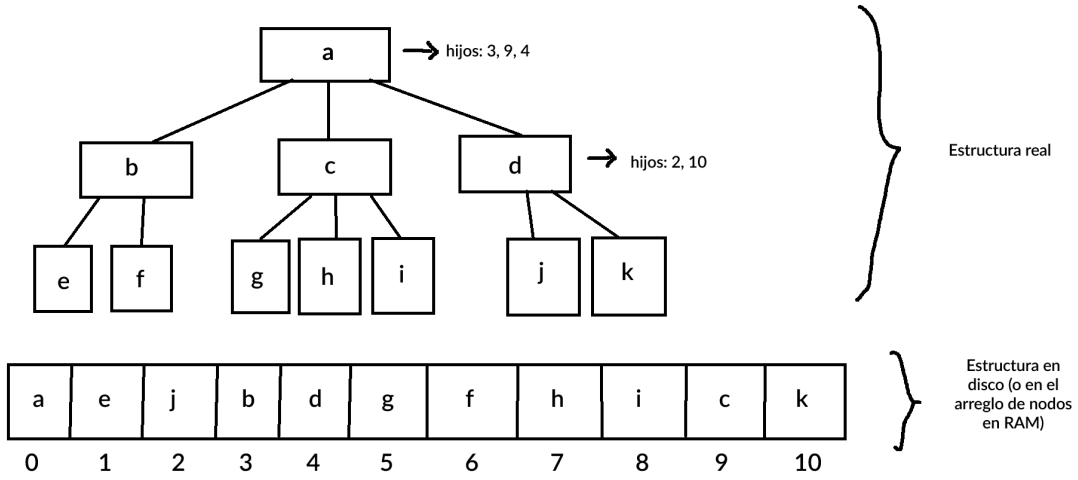

Figura 1: Representación de un árbol B en disco, o como un arreglo de nodos en RAM

3.2. Implementación

Se espera que puedan implementar, tanto para el árbol B como para el árbol B+, mecanismos para **insertar** un par (t, v) y **buscar** todos los pares con llaves en un rango $[\ell, u]$. **No se debe implementar el borrado de elementos del árbol**. Recuerden que para insertar nodos, se guarda el árbol en un arreglo de nodos en RAM, mientras que para la búsqueda, los nodos se deben leer desde un archivo en disco.

A continuación se presenta un esqueleto de implementación, el cual pueden utilizar como guía.

3.2.1. *Split* de un nodo (árbol B)

Este método recibe un nodo que está lleno (es decir, posee b pares llave-valor, y $b+1$ hijos si es interno), y lo divide en dos nuevos nodos.

1. Seleccionar el par llave-valor en la posición $\frac{b}{2}$ (contando desde 1). Esta será nuestro **par mediano**.
2. Guardar los $\frac{b}{2} - 1$ pares a la izquierda del mediano en un nuevo nodo izquierdo, y los $\frac{b}{2}$ pares a la derecha del mediano en un nuevo nodo derecho.
3. Si el nodo es interno, guardar los $\frac{b}{2}$ primeros hijos en el nuevo nodo izquierdo, y los otros $\frac{b}{2} + 1$ en el nuevo nodo derecho.
4. Retornar los nuevos nodos y el par mediano.

3.2.2. *Split* de un nodo (árbol B+)

En el árbol B+, el *split* es idéntico al del árbol B, solamente que si se está dividiendo una hoja, el par mediano también se guarda en el nuevo nodo izquierdo (como el elemento número $\frac{b}{2}$, contando desde 1).

3.2.3. Inserción de un par en un nodo (árbol B)

- Si el nodo actual H es una hoja, y no es la raíz:

1. Insertar el par en H, de tal forma que los pares queden ordenados según sus llaves.
2. Actualizar la cantidad de elementos en H.
3. Escribir la hoja actualizada en su posición correspondiente.

- Si el nodo actual V es interno, pero no la raíz:
  1. Usando las llaves del nodo, encontrar el hijo U en el cual insertar el nuevo par (recordar que el nodo `hijos[i]` guarda únicamente valores cuya llave es mayor a la llave de `llaves_valores[i-1]`, y menor o igual a la llave de `llaves_valores[i]`).
  2. Si U no está lleno, insertar el nuevo par en U.
  3. Si U está lleno (tiene b pares llave-valor):
     1. Realizar un *split* de U, obteniendo los nuevos nodos U_{izq} y U_{der}, junto con el par mediano (k_u, v_u).
     2. Insertar (k_u, v_u) en la lista de pares llave-valor de V (recordando que estos pares deben quedar ordenados ascendientemente por llave).
     3. Escribir los nodos de los nuevos hijos:
        - U_{der} al final del arreglo.
        - U_{izq} en el índice antiguo de U (como este se dividió, se puede reemplazar).
     4. Agregar el índice del U_{der} a la lista de hijos del nodo (inmediatamente después del índice antiguo de U, ahora ocupado por U_{izq}).
     5. Actualizar la cantidad de elementos en V.
     6. Escribir el nodo V, ahora actualizado.
     7. Insertar el nuevo par en U_{izq} si la llave del par es menor o igual a k_u, o U_{der} si la llave es mayor a k_u.
- Si el nodo actual es la raíz R:
  1. Si la raíz no está llena, insertar el par en R.
  2. Si la raíz está llena:
     1. Hacer un *split*, recibiendo nodos R_{izq}, R_{der} junto con el par mediano (k_r, v_r).
     2. Escribir **ambos** nuevos nodos al final del arreglo.
     3. Crear una nueva raíz R' (vacía).
     4. Agregar el par mediano a la lista de pares llave-valor de R'.
     5. Insertar los índices de los nuevos hijos en R'.
     6. Actualizar la cantidad de elementos en R'.
     7. Escribir R' en la primera posición del arreglo, reemplazando la raíz antigua.
     8. Insertar el nuevo par en R_{izq} si la llave del par es menor o igual a k_r, o R_{der} si la llave es mayor a k_r.

3.2.4. Inserción en un nodo (árbol B+)

En el árbol B+, la inserción es idéntica a la del árbol B, con una diferencia si el hijo U es dividido, y además U es una hoja. Después de guardar U_{der} pero antes de guardar U_{izq}, se debe modificar U_{izq} para que en su campo `siguiente` esté el índice/posición en la cual se guardó U_{der}.

Esta misma asignación debe realizarse cuando la raíz se divide por primera vez (ya que inicialmente es una hoja): después de guardar R_{der} pero antes de guardar R_{izq}, se modifica R_{izq} para que en su campo `siguiente` esté el índice/posición en la cual se guardó R_{der}.

Si estas modificaciones no se realizan, va a fallar la búsqueda por rango en el árbol.

3.2.5. Búsqueda por rango (árbol B)

Esta búsqueda se inicia en la raíz (el primer nodo del archivo binario correspondiente al árbol), y se continúa recursivamente por los nodos internos hasta llegar a las hojas.

- Si el nodo actual es una hoja:
  1. Revisar todos los pares y retornar aquellos cuya llave esté dentro del rango $[\ell, u]$.
- Si el nodo actual es interno:
  1. Usando las llaves de los pares, buscar recursivamente en todos los hijos que podrían tener elementos en el rango $[\ell, u]$.
  2. Además, revisar todos los pares y guardar aquellos cuya llave esté dentro del rango $[\ell, u]$.
  3. Retornar los elementos encontrados en ambas búsquedas anteriores.

3.2.6. Búsqueda por rango (árbol B+)

Esta búsqueda tiene dos partes:

1. Recursiva o iterativamente (partiendo desde la raíz), bajar por los nodos hasta encontrar la hoja en la cual debería ser insertada la llave ℓ.
2. Una vez en esa hoja, ir recorriendo todas las hojas (saltando de una a la siguiente mediante el campo `hoja_siguiente`), guardando todos los pares cuya llave esté en el rango $[\ell, u]$. Cuando encontremos una llave con valor mayor a u, detener la búsqueda y retornar los pares encontrados.

4. Experimentación

Para cada $N \in \{2^{15}, 2^{16}, ..., 2^{26}\}$, realizar lo siguiente:

- Uno a uno, leer los primeros N pares del archivo binario `datos.bin` e insertarlos en un árbol B y en un árbol B+. Esta operación se realiza en RAM.
- Escribir los árboles a disco (escribiendo cada nodo de manera secuencial).
- En ambos árboles, realizar 50 búsquedas en rangos aleatorios $[\ell, u]$ de tamaño 604.800 (cantidad de segundos equivalentes a una semana), generados uniformemente. Para esto, deben obtener ℓ uniformemente dentro del rango $[1546300800, 1754006400]$ (valor mínimo y máximo de las llaves), y sumarle 604.800 para obtener u.
- Para cada árbol, reportar:
  - El tiempo de creación del árbol (inserción de los N pares).
  - La cantidad de I/Os realizadas al insertar los N pares. Estas I/Os son simuladas, pues la inserción ocurre en RAM, pero deben ser contabilizadas según lo mencionado en la Sección 3.1.
  - El tamaño del árbol (ya sea tamaño en disco o cantidad de nodos usados).
  - El tiempo promedio de ejecución de cada búsqueda.

– La cantidad promedio de I/Os durante cada búsqueda.

Deberán graficar todas estas métricas en función de N, y compararlas entre el árbol B y el árbol B+.

Además, deberán graficar el **resultado** (es decir, los tiempos y las temperaturas) de una misma búsqueda por rango para dos valores de N distintos, y comparar lo obtenido en cada caso. Para esto, es irrelevante si utilizan los resultados entregados por el árbol B o el árbol B+, ya que debieran ser iguales.

5. Entregables

Se deberá entregar el código y un informe donde se explique el experimento en estudio. Con esto se obtendrá una nota de código ($NCod$) y una nota de informe ($NInf$). La nota final de la tarea será el promedio simple entre ambas notas ($NT_1 = 0.5 \cdot NCod + 0.5 \cdot NInf$).

5.1. Código

La entrega de código tiene que contener:

- **(0.3 pts)** README: Archivo con las instrucciones para ejecutar el código, debe ser lo suficientemente explicativo para que cualquier persona solo leyendo el README pueda ejecutar la totalidad de su código (incluyendo librerías no entregadas por el equipo docente que potencialmente se deban instalar).
- **(0.2 pts)** Firmas: Cada estructura de datos y función debe tener una descripción de lo que hace y una descripción de sus parámetros de entrada y salida.
- **(1.0 pts)** Uso de disco: El uso de disco debe ser correcto.
  - Se deben realizar lecturas y escrituras por bloque (recuerden que un bloque es equivalente a un nodo para efectos de esta tarea).
  - Al crear el árbol B/B+, se deben guardar los nodos en un arreglo secuencial simulando el disco.
  - Una vez completo el árbol, se debe guardar en un archivo binario.
  - Se debe poder realizar la interpretación del binario en memoria principal para realizar las búsquedas y cargar los datos de temperatura.
- **(1.5 pts)** Implementación de inserción: La implementación del *split* y el algoritmo de inserción es correcto para ambos árboles.
- **(1.5 pts)** Implementación de búsqueda: La implementación de búsquedas por rango es correcta para ambos árboles.
- **(0.5 pts)** Experimento: Se realiza la experimentación para los valores de N pedidos.
- **(0.5 pts)** Obtención de resultados: La forma en el que se obtienen los resultados (tiempos de ejecución, I/Os y tamaño de los árboles) es correcta.
- **(0.5 pts)** Main: Un archivo o parte del código (función main) que permita ejecutar la construcción y búsquedas.

5.2. Informe

El informe debe ser claro y conciso. Se recomienda hacerlo en LaTeX o Typst. Debe contener:

- **(0.8 pts)** Introducción: Presentar el tema en estudio, resumir lo que se dirá en el informe y presentar una hipótesis.
- **(0.8 pts)** Desarrollo: Presentación de algoritmos, estructuras de datos, cómo funcionan y por qué. Recordar que los métodos ya son conocidos por el equipo docente, lo que importa son sus propias implementaciones (qué decisiones tomaron que no están mencionadas en el enunciado).
- **(2.4 pts)** Resultados: Especificación de los datos que se utilizaron para los experimentos, la cantidad de veces que se realizaron los tests, con qué inputs, qué tamaño, etc. Se debe mencionar el sistema operativo y los tamaños de sus cachés y RAM con los que se ejecutaron los experimentos. Se deben mostrar gráficos/tablas con la información solicitada en la Sección 4 y mencionar solo lo que se puede observar de estos.
- **(1.2 pts)** Análisis: Comentar y concluir sus resultados. Se hacen las inferencias de sus resultados, aplicando conocimientos del curso.
- **(0.8 pts)** Conclusión: Recapitulación de lo que se hizo, se concluye lo que se puede decir con respecto a sus resultados. También ven si su hipótesis se cumplió o no y analizan la razón. Por último, se menciona qué se podría mejorar en su desarrollo en una versión futura, qué falta en su documento, qué no se ha resuelto y cómo se podrían extender.

6. Informaciones útiles

- Si hacen búsquedas en rangos de una semana, debieran obtener cerca de un 0.29% de los datos en cada búsqueda. Si están obteniendo muchos menos datos que eso, significa que su búsqueda no está funcionando adecuadamente.
- Les recomendamos que, al escribir las estructuras árbol B y B+, hagan una implementación genérica (si su lenguaje les permite) para abstraer el hecho de estar trabajando en RAM versus trabajando en disco. Esto significa que solo deben realizar una implementación para cada tipo de árbol, y es más fácil contabilizar correctamente el número de accesos al disco.
- Si están usando C++, les puede ser útil el siguiente repositorio creado por Pablo Skewes (auxiliar del semestre pasado). El proyecto consiste en serializar un árbol de búsqueda binaria, por lo cual comparte similitudes con lo que se les pide en esta tarea (p.ej. al leer y escribir estructuras en disco). También les puede servir como referencia en cuanto a calidad del README y las firmas.
- Si desean saber a qué momento en el tiempo corresponde una búsqueda por rango, pueden utilizar la siguiente herramienta para convertir de tiempo Unix a UTC o tiempo local. Si quieren hacerlo de forma programática, podría ser útil la función `strftime`.