TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENERÍA

DISEÑO E IMPLEMENTACIÓN DE UN ÁRBOL DE BÚSQUEDA CONCURRENTE PARA DISTRIBUCIONES DE ACCESO NO UNIFORMES

Tesis para optar por el Título de Ingeniero Informático, que presenta el bachiller:

Walter Alfredo ERQUÍNIGO PEZO

ASESOR: Doctor César BELTRÁN CASTAÑÓN

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

"Science is always discovering odd scraps of magical wisdom and making a tremendous fuss about its cleverness."

Aleister Crowley

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Índice general

Índice de figuras IV

Índice de cuadros V

Resumen VI

1. Generalidades 1
- 1.1. Definición de la problemática . . . 1
- 1.2. Objetivo general . . . 4
- 1.3. Objetivos específicos . . . 4
- 1.4. Resultados esperados . . . 4
- 1.5. Alcance y limitaciones . . . 5
- 1.6. Justificación y viabilidad . . . 6
- 1.7. Plan de proyecto . . . 8

2. Métodos y procedimientos 9
- 2.1. FindBugs TM . . . 9
- 2.2. Ingeniería de Algoritmos . . . 10
- 2.3. Distribución de Zipf como fuente de datos para comparaciones . . . 11

3. Estado del arte 12
- 3.1. Marco Conceptual . . . 12
  - 3.1.1. Árbol de búsqueda . . . 12
  - 3.1.2. Rotaciones en árboles binarios . . . 13
  - 3.1.3. Árboles autobalanceados . . . 14
  - 3.1.4. Árbol balanceado por peso . . . 15
  - 3.1.5. Árbol de Van Emde Boas . . . 15
  - 3.1.6. Diccionarios . . . 16
  - 3.1.7. Funciones Hash . . . 17
  - 3.1.8. Análisis amortizado . . . 19
  - 3.1.9. Heap . . . 19
  - 3.1.10. Hilo . . . 19
  - 3.1.11. Lock . . . 19
- 3.2. Árboles balanceados para distribuciones no uniformes . . . 20
  - 3.2.1. AVL Tree . . . 20
  - 3.2.2. Splay Tree . . . 20
  - 3.2.3. Lazy Splay Tree . . . 21

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

3.2.4. CBTree . . . 22
3.2.5. Treap . . . 23
3.2.6. Tango Tree y Zipper Tree . . . 24
3.3. Conclusiones del estado del arte . . . 25

4. Diseño de la estructura UHTree **26**
4.1. Diseño . . . 26
4.1.1. Definición . . . 26
4.1.1.1. Propiedades . . . 26
4.1.1.2. Definición de *UHElement* . . . 27
4.1.1.3. Definición de *UHList* . . . 28
4.1.1.4. Definición de *UHNode* . . . 28
4.1.1.5. Definición de *UHTree* . . . 30
4.1.2. Detalles de implementación . . . 31

5. Análisis de la estructura UHTree **32**
5.1. Análisis de la estrutura . . . 32
5.1.1. Análisis de UHElement . . . 32
5.1.2. Análisis de UHList . . . 32
5.1.3. Análisis de UHBitmaskHeap . . . 33
5.1.4. Análisis de UHNode . . . 33
5.1.4.1. Método isEmpty . . . 34
5.1.4.2. Métodos getMinHash y getMinElement . . . 34
5.1.4.3. Método updateBest . . . 34
5.1.4.4. Método add . . . 34
5.1.4.5. Método contains . . . 35
5.1.4.6. Método remove . . . 36
5.1.5. Conclusiones del análisis . . . 36

6. Comparación numérica **38**
6.1. Estadísticas . . . 39
6.1.1. Rendimiendo con datos generados aleatoreamente usando distribución de Zipf . . . 39
6.1.2. Rendimiento con datos del proyecto Gutenberg (sólo lecturas) . . . 39
6.1.3. Rendimiento con datos del proyecto Gutenberg (caso general) . . . 41
6.2. Conclusiones de la experimentación . . . 42

7. Observaciones, conclusiones y recomendaciones **43**
7.1. Observaciones . . . 43
7.2. Conclusiones . . . 44
7.3. Recomendaciones y trabajos futuros . . . 44

A. Código del UHTree **46**

Bibliografía **47**

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Índice de figuras

2.1. Diagrama de interacción de los componentes de la Ingeniería de Algoritmos. Fuente: [5]. 11

3.1. Rotaciones en un árbol binario. Fuente: `http://en.wikipedia.org/wiki/File:Tree_rotation.png`. 14
3.2. Ejemplo de un árbol balanceado por peso, donde a mayor prioridad de un elemento, éste se encuentra a menor produndidad. Fuente: `http://en.wikipedia.org/wiki/File:Weight_balanced_tree2.jpg`. 15
3.3. Ejemplo de un árbol de Van Emde Boas que contiene a los elementos 1, 2, 3, 5, 8 y 10, los cuales sólo necesitan 4 bits para representarse. El segundo nivel contiene a todos los pares con los dos bits más significativos iguales. El tercer nivel contiene a todos los elementos. Fuente: `http://upload.wikimedia.org/wikipedia/commons/thumb/6/6b/VebDiagram.svg/1000px-VebDiagram.svg.png`. 17

6.1. Resultados usando datos generados aleatoriamente. 40
6.2. Resultados con datos del proyecto Gutenberg realizando sólo lecturas. . 40
6.3. Resultados con datos del proyecto Gutenberg realizando lecturas y escrituras. 41

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Índice de cuadros

1.1. Emparejamiento entre los objetivos específicos, resultados esperados y sus método de verificación. 5

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Resumen del proyecto de fin de carrera

En muchas aplicaciones de búsqueda de información se necesita una estructura eficiente que pueda almacenar y leer datos concurrentemente bajo el supuesto de que la distribucion del acceso a dichos datos no es uniforme. Ejemplos son las memorias caché para aplicaciones Web o las mismas bases de datos. Existen soluciones para esto y cada una con un rendimiento diferente bajo ciertos entornos. Sin embargo, estas soluciones suelen estar basadas en un mismo tipo de árboles de búsqueda, los autobalanceados, por lo que tienen ciertas limitaciones en términos de concurrencia.

En el presente trabajo se desarrollará una nueva estructura concurrente y escalable, la cual se basará en el árbol no autobalanceado de Van Emde Boas [1], y se realizarán experimentos para determinar su rendimiento en comparación con el de otras estructuras comúnmente usadas. Finalmente, se determinará bajo qué circunstancias es útil. Esta estructura la llamaremos UHTree (Unicursal Hexagram Tree).

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Capítulo 1

Generalidades

En este capítulo, se aborda la problemática en la gestión eficiente de información digital. Asimismo, se explica una nueva solución para el caso de aplicaciones donde los elementos de la información son accedidos de manera no uniforme, el cual es un caso común en la realidad.

Con el fin de estructurar de manera adecuada el proyecto, se definen los objetivos del proyecto junto con sus resultados esperados. Adicionalmente, la justificativa, la viabilidad y los alcances y limitaciones son mostrados. Finalmente, se presenta la metodología a usar en el proyecto y el plan a seguir en su desarrollo.

1.1. Definición de la problemática

Una rama muy importante de las Ciencias de la Computación es la enfocada a desarrollar soluciones para problemas de búsqueda de información. A lo largo de la historia se inventaron los árboles binarios de búsqueda, Tries, técnicas de Hashing, listas dinámicas, etc. [2] como soluciones para muchos casos específicos. Un caso particular y de gran importancia se da cuando el acceso a la información es concurrente y no tiene distribución uniforme [3]; en otras palabras, cuando hay elementos que se acceden con mayor frecuencia que otros y, adicionalmente, no hay un patrón específico en el orden en que se acceden a estos.

Aplicaciones que requieren de este tipo de estructuras son las siguientes, entre otras:

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- Búsqueda lexicográfica de texto
- Compresión de datos
- Encriptación
- Bases de datos no persistentes (memorias caché)
- Búscadores en servidores DNS

Las soluciones más comunes para este problema en particular son basadas en árboles de búsqueda, tales como los árboles AVL, Splay Trees, Treaps, CBTrees [4], B-trees, entre otros. La mayoría de estos árboles, sean binarios o no, se basan en el *auto-balanceo*, que es una operación que mantiene la áltura del árbol bajo ciertos limites mediante el cambio de posición o rotación de sus elementos. Sin embargo, esta propiedad fundamental trae consigo nuevos problemas en el campo de la concurrencia. Si un elemento es cambiado de posición, todos los accesos a los elementos de su subárbol son paralizados cuando intenten atraversarlo mientras el cambio ocurre, lo que ocasiona demoras que pueden llegar a ser sustanciales. Este fenómeno es peor aún cuando la raíz del árbol es la que se rota, ya que detiene todo acceso al árbol momentáneamente.

No se han estudiado muchas alternativas a esto, ya que las estructuras sin rotación son mucho más complejas y suelen ser más lentas. No obstante, la posibilidad de eliminar esta desventaja puede traer consigo mejoras considerables e incluso de gran impacto. El problema en sí deriva de que las aplicaciones que usan este tipo de estructuras necesitan el mayor rendimiento posible, razón por la cual desde el inicio de la Computación no se ha dejado de buscar nuevas soluciones que sean más eficientes. Como ya mencionamos, casi todas estas estructuras tienen un cuello de botella dada por las rotaciones. Ante este conexto, hallar una nueva estructura eficiente sin esta desventaja podría ser de gran utilidad para todas estas aplicaciones.

Por esto, en el presente proyecto se presenta una nueva alternativa de solución que es una estructura basada en el *árbol de Van Emde Boas* [1], el cual no es autobalanceado, ya que mantiene una estructura semi-estática y presenta un tiempo de acceso en $\mathcal{O}(\log \log n)$ cuando sólo almacena numeros enteros, a diferencia de la mayoría de

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

árboles de búsqueda que suelen estar en $\mathcal{O}(\log n)$ por acceso, que es exponencialmente más lento. A esta nueva estructura se le otorgó el nombre de Unicursal Hexagram Tree o UHTree.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

1.2. Objetivo general

El objetivo del presente proyecto es diseñar y desarrollar un nuevo árbol de búsqueda concurrente sin rotaciones llamado UHTree para la manipulación de información con distribuciones de acceso no uniformes.

1.3. Objetivos específicos

Los objetivos específicos del presente proyecto son:

1. **OE1:** Diseñar el UHTree definiendo las entradas y tipos de datos que soportará la estructura.
2. **OE2:** Analizar la correctitud y complejidad teórica del UHTree como teoremas.
3. **OE3:** Desarrollar el UHTree como una aplicación.
4. **OE4:** Realizar una comparación numérica del UHTree con otros árboles como diagramas estadísticos según el rendimiento ante diversas entradas para determinar su eficiencia en la práctica.

1.4. Resultados esperados

Los resultados esperados son:

1. **RE1:** Diseño de del UHTree como definición abstracta, junto con las especificaciones de sus propiedades e interfaces.
2. **RE2:** Análisis de la complejidad y correctitud del UHTree como teoremas.
3. **RE3:** Una librería con una implementación correcta del árbol junto con sus pruebas de integración.
4. **RE4:** Diagramas estadísticos de comparación del UHTree con otros árboles para analizar su rendimiento.

Tesis publicada con autorización del autor
No olvide citar esta tesis

A continuación se muestra la relación entre los objetivos específicos y los resultados esperados, junto con su esquema de verificación.

Objetivo específico	Resultado esperado	Verificación
OE1	RE1	Verificar que la especificación del UHTree sea consistente luego de realizar las pruebas de código.
OE2	RE2	Verificar la correctitud de las demostraciones.
OE3	RE3	Verificar el correcto funcionamiento de la implementación a través de pruebas concurrentes aleatorias.
OE4	RE4	Verificar la integridad de los datos de entrada de los experimentos al ajustarlo a distribuciones de Zipf.

CUADRO 1.1: Emparejamiento entre los objetivos específicos, resultados esperados y sus método de verificación.

Nota: si bien verificar RE1, RE3 y RE4 no es una tarea muy compleja y es metódica, RE2 no presenta esta característica. Esto se debe a que para probar la correctitud de un algoritmo o estructura de datos concurrente no basta hacer pruebas; más bien, se necesita demostrar matemáticamente, línea por línea del pseudocódigo o del código, que no hay errores. Este trabajo es puramente lógico y no hay mayor método que la lógica misma.

1.5. Alcance y limitaciones

Para el presente proyecto se debe tener en consideración lo siguiente:

- La solución se basa en una modificación del árbol de Van Emde Boas.
- El proyecto se enfoca en desarrollar una nueva solución para el problema planteado, por lo que el desarrollo de la estructura se limita a producir una librería y no una herramienta visual.
- El fin del proyecto no es desarrollar una herramienta para usuario final, sino mostrar la solución y realizar las comparaciones pertinentes para analizar su grado de utilizabilidad en las aplicaciones de uso real. Sin embargo, se decidió producir

Tesis publicada con autorización del autor
No olvide citar esta tesis

una librería escrita en Java para pueda ser usada en cualquier sistema operativo. Además, dado que la mayoría de implementaciones de estructuras concurrentes está hecha en Java, se decidió usar Java.

- El proyecto incluye un análisis teórico exhaustivo de la correctitud y ventajas de la solución.

- En lo que respecta a las comparaciones con otras estructuras, se limita a realizar pruebas numéricas con datos de comparación especializados para este tipo de estructuras. Dichas pruebas son resumidas en gráficos estadísticos.

1.6. Justificación y viabilidad

El fin último de este proyecto es experimentar una nueva alternativa a los árboles con rotación de propósito común. Al no haber en la literatura muchas experimentaciones con árboles semi-estáticos, tal como el de Van Emde Boas, es de carácter relevante explorar nuevas alternativas. Sea cual fuere el rendimiento real de la presente solución, se pueden obtener dos conclusiones.

- En caso de que el rendimiento de la estructura sea pobre, se puede analizar cuánto mejor es esta modificación respecto de la versión original del árbol de Van Emde Boas, y así conocer qué mejoras son aprovechables para aplicaciones o estructuras similares.

- Si el rendimiento es similar al de las estructuras comunes o incluso mejor, se puede analizar bajo qué circunstancias la estructura es una buena opción, abriendo una nueva posibilidad al elegir una estructura de datos al diseñar algoritmos que necesiten un árbol para almecenar información. Más aún, se le da mayor importancia al árbol de Van Emde Boas que ha sido relegado poco a poco a lo largo de las décadas a pesar de su poca complejidad y simplicidad.

Acerca de la usabilidad, esta estructura se podría usar en cualquier aplicación de las mencionadas en la sección de Problemática. Yendo más allá, se podría usar en reemplazo de estructuras de Java como HashTable, SortedSet o ConcurrentHashMap.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

Finalmente, es pertinente mencionar que el proyecto es viable para un proyecto de fin de carrera, ya que se basa modificaciones de una estructura ya existente, junto con un marco de trabajo basado en investigaciones similares. Siendo un limitante común en estos proyectos, y más aún en éste, el tiempo, se procedederá a detener el estudio de las modificaciones en un tiempo dado, ya que no es sencillo determinar cuánto tiempo tomará analizar y experimentar cada tipo de alteración. Por lo tanto, en cuestiones de tiempo es viable.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

1.7. Plan de proyecto

Se decidió dividir el desarrollo del proyecto en la siguientes fases cuyo desarrollo es secuencial:

1. Diseño del UHTree: el cual incluye la estructura junto con su especificación y sus interfaces. Adicionalmente, se justifica cada componente y sus características.
2. Análisis del UHTree: se procede a demostrar la complejidad teórica de la estructura y se demuestra la correctitud de sus especificaciones.
3. Implementación del UHTree: se implementa el UHTree en Java 7. Adicionalmente, se crean pruebas concurrentes para su verificación usando asertos.
4. Experimentación: esta etapa incluye 3 fases:
   - Recopilar y/o implementar otros algoritmos comúnmente usados. Asimismo, crear una pequeña suite de comparación de algoritmos.
   - Preparar los juegos de datos con los que se realizará las comparaciones numéricas.
   - Finalmente, se procede a realizar la experimentación y a analizar los resultados.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Capítulo 2

Métodos y procedimientos

En esta sección se definen las metodologías empleadas en la elaboración de las partes específicas del proyecto.

2.1. FindBugs ™

Cubre el OE3: en la implementación de cualquier algoritmo o estructura de datos sin concurrencia, las pruebas es una parte fundamental para probar que la implementación es correcta . Sin embargo, para aplicaciones concurrentes las pruebas no garantizan la correctitud en lo más mínimo, por lo que existen herramientas de análisis estático de código para hallar errores latentes en concurrencia. FindBugs suple este requerimiento para tener más certeza que la implementación no tiene errores de sincronización.

Es una herramienta de análisis estático para buscar bugs en código Java. Es una herramienta de uso libre y fue desarrollado por la Universidad de Maryland. Dado que el código del presente proyecto es concurrente, esta herramienta permite hallar bugs y dar recomendaciones muy útiles para evitar algunos errores muy sutiles que en concurrencia a veces es dificil de encontrar simplemente probando. Esta herramienta tiene más de 50 bugs documentados para verificar la correctitud de la concurrencia. Adicionalmente, tiene mas de 200 bugs documentados para cualquier otro ámbito del lenguaje Java. Su página principal es `http://findbugs.sourceforge.net/`.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

2.2. Ingeniería de Algoritmos

Cubre los OE1 y OE2: en el diseño y en el análisis de esta nueva estructura es necesario llevar en consideración la teoría y los entornos prácticos de implementación. La Ingeniería de Algoritmos suple esta necesidad.

Es una disciplina encargada de unir la teoría y práctica de los algoritmos. Si bien el estudio teórico produce una basta candidad de algoritmos y estructuras de datos junto con su análisis teórico, esto no es suficiente para aplicaciones del mundo real, donde el hardware (jerarquía de memoria, cachés, multinúcleos, clusters, etc.) impacta en el rendimiento de las implementaciones de dichos algoritmos. Con el fin de suplir esta carencia, la Ingenería de Algoritmos se encarga del diseño y análisis sistemático de algoritmos bajo entornos reales.

La base del desarrollo de algoritmos en esta disciplina es un ciclo basado en los siguientes puntos [5]:

- Diseño
- Análisis
- Implementación
- Experimentación

Sin ir muy profundamente, el diseño se encarga de crear teóricamente el algoritmo, sin olvidarse del entorno real (hardware, usabilidad, capas, etc.) como parte del diseño mismo. El análisis se enfoca en demostrar teoremas acerca de la correctitud y de la eficiencia teórica y práctica. La implementación se basa en el desarrollo físico del diseño. Finalmente, la experimentación se encarga de probar si las hipótesis sobre el algoritmo son correctas y asimismo proveer de más resultados sobre su comportamiento para decidir qué cambios en el diseño realizar, lo que permite proseguir con el ciclo de mejora continua.

Tesis publicada con autorización del autor
No olvide citar esta tesis

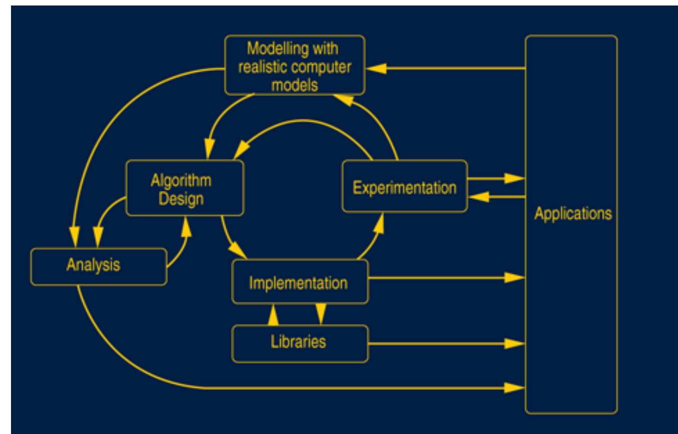


FIGURA 2.1: Diagrama de interacción de los componentes de la Ingeniería de Algoritmos. Fuente: [5].

2.3. Distribución de Zipf como fuente de datos para comparaciones

Cubre OE4: para poder realizar la comparación numérica, es necesario definir qué distribución de datos se usará como fuente de datos. Esta distribución cumple ese rol permitiendo generar datos de prueba.

La distribución de Zipf, también llamada distribución Zeta, es una distribución de probabilidad inventada por George Kingsley Zipf, lingüista de la universidad de Hardvard, al estudiar la probabilidad de aparición de cada letra o palabra en los lenguajes naturales. Luego, se descubrió que la versión discreta de esta distribución aproxima bastante bien la mayoría de datos estudiados en las ciencias físicas y sociales. Dado que esta distribución es la de mayor aparición en las aplicaciones de la realidad, se decide usarlo como fuente de datos aleatorios para las pruebas de comparación.

En general, se suele aproximar esta distribución de manera que el n-ésimo elemento más probable de la distribución discreta tiene probabilidad $1/n^a$, donde $a > 1$.

Cabe mencionar que ya existen librerías como Apache para generar datos aleatorios con dicha distribución, los cuales fueron usados en la experimentación. Adicionalmente, hay estudios en los que se detallan como realizar estas pruebas, tales como el realizado por Jim Bell y Gopal Gupta en su publicación 'An Evaluation of Self-adjusting Binary Search Tree Techniques' [3], la cual es la fuente principal usada en este proyecto.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Capítulo 3

Estado del arte

En este capítulo se presentan todos los conceptos necesarios para comprender cabalmente la estructura, su diseño, desarrollo y experimentación. Asimismo, se muestra una gama de estructuras similares usadas actualmente junto con los conceptos necesarios para comprenderlos.

3.1. Marco Conceptual

A continuación se hará una breve descripción de los conceptos teóricos referente a este tipo de estructuras.

3.1.1. Árbol de búsqueda

Es un árbol binario T que almacena elementos S' de un conjunto S que tiene definido un orden $\leq$, tal que $\forall x \in S'$:

$$y \in SubArbolIzquierdo(x) \Rightarrow y \leq x$$

$$y \in SubArbolDerecho(x) \Rightarrow x \leq y$$

A pesar de que en la literatura [6] se les define usualmente como árboles binarios, nos damos la libertad de no asumir que son binarios, para poder trabajar con un mayor

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

rango de árboles con el propósito común de almacenar datos de un conjunto S con un orden $\leq$. Adicionalmente, estos árboles puede admitir elementos repetidos, con lo que S' se torna un multiconjunto.

Las operaciones usuales en estos árboles, así como en otras estructuras similares, son las siguientes:

- Búsqueda: determinar si un elemento x está presente en T.
- Inserción: añadir un elemento x a T.
- Eliminación: eliminar un elemento x de T si es que se encuentra dentro.

Similarmente, las siguientes complejidades son de general importancia:

- Espacio: la complejidad del espacio que ocupa T para almacenar un subconjunto $S' \subset S$.
- Búsqueda: la complejidad del tiempo que tarda realizar una búsqueda cuando T contiene un subconjunto $S' \subset S$.
- Inserción: la complejidad del tiempo que tarda realizar una inserción cuando T contiene un subconjunto $S' \subset S$.
- Eliminación: la complejidad del tiempo que tarda realizar una eliminación cuando T contiene un subconjunto $S' \subset S$.

Comúnmente, no se asume ninguna propiedad sobre el subconjunto $S' \subset S$ presente en T y simplemente se describe las complejidades en función de la cardinalidad de S'. Sin embargo, el orden en el que los elementos de S' son insertados, eliminados y buscados puede dar más información relevante para el diseño de mejores algoritmos, el cual es el objetivo del presente proyecto.

3.1.2. Rotaciones en árboles binarios

Uno de los objetivos al realizar implementaciones eficientes de árboles binarios es garantizar que la profundidad de estos sea la menor posible. Asumiendo que un árbol

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

tiene n elementos, es posible disponer de los elementos de manera que la profundidad sea $\lfloor \log_2 n \rfloor$, lo cual es óptimo, ya que en esta configuración todos los niveles del árbol salvo tal vez el último están completos. Sin embargo, es muy complicado mantener un árbol con esta configuración luego de aplicarle ciertas operaciones, por lo que simplemente garantizar una cota superior pequeña a la altura del árbol ya es algo muy útil. Cuando un árbol tiene altura acotada según el número de sus elementos, se dice que es balanceado por altura.

Una subrutina común para garantizar esta cota superior es la de rotación, la cual es usada como parte importante es los AVL Trees, Treaps, Splay Trees, Red-Black Trees, etc.

Una rotación se puede realizar en dos sentidos: horario y antihorario. Ambos quedan resumidos en el siguiente gráfico

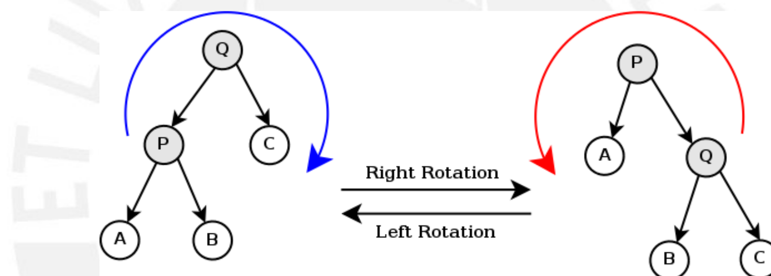


FIGURA 3.1: Rotaciones en un árbol binario. Fuente: http://en.wikipedia.org/wiki/File:Tree_rotation.png.

Es fácil observar que si un árbol binario de búsqueda es rotado en cualquiera de las dos direcciones, seguirá siendo un árbol binario de búsqueda ya que el orden relativo de los elementos según el orden $\leq$ no se altera. Además, una rotación aumenta la profundidad de un lado en 1 y disminuye el del lado opuesto en 1, lo que permite aplicar sucesivas rotaciones a un árbol de manera que quede balanceado por altura.

3.1.3. Árboles autobalanceados

Un árbol autobalanceado es simplemente un árbol que asume el invariante de que antes y después de cada operación esta balanceado por altura.

Tesis publicada con autorización del autor
No olvide citar esta tesis

3.1.4. Árbol balanceado por peso

Un árbol balanceado por peso, a diferencia de los autobalanceados, no se balancea según su altura, si no según la probabilidad del acceso a cada elemento, de manera que los elementos con mayor probabilidad de acceso estén más cerca de la raíz. Existen dos tipos, los offline y los online. Los offline son los que conocen a priori la distribución del acceso de los elementos, por lo que el árbol se puede organizar óptimamente, disminuyendo el número esperado de operaciones realizadas luego de n operaciones. En cambio, los online no conocen a priori esta distribución, más bien deben balancearse a medida que los elementos son accedidos (y así la distribución se va conociendo) de manera que se intente minimizar el número esperado de operaciones en las siguientes llamadas a la estructura.

Estos árboles son de gran importancia en la realidad, ya que suele suceder que los elementos con los que trabajará un árbol tiene una distribución de probabilidad no uniforme.

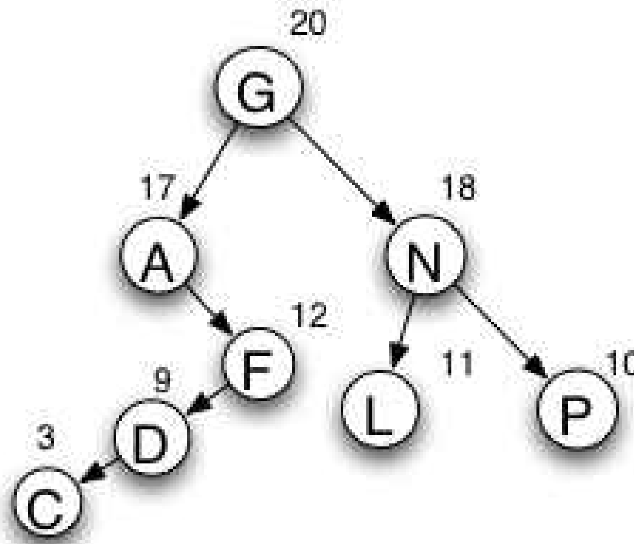


FIGURA 3.2: Ejemplo de un árbol balanceado por peso, donde a mayor prioridad de un elemento, éste se encuentra a menor produndidad. Fuente: http://en.wikipedia.org/wiki/File:Weight_balanced_tree2.jpg.

3.1.5. Árbol de Van Emde Boas

El árbol de Van Emde Boas, de ahora en adelante llamado vEB, es un árbol de búsqueda cuyo dominio es $\{1, 2, 3, \ldots, n\}$ [7]. Este simple hecho permite que se pueda tener

Tesis publicada con autorización del autor
No olvide citar esta tesis

la siguiente especificación para la estructura:

- Nombre: $vEB(n)$.
- Estado: $T \subset \{1, 2, 3, \ldots, n\}$.
- Operaciones:
  $insertar(x)$: Agregar x a T.
  $remover(x)$: Remover x de T.
  $sucesor(x)$: Retornar el menor elemento en $T \geq x$.

Existen más propiedades del vEB, pero estas son las más relevantes para este proyecto. Para una explicación profunda del vEB, se puede leer el trabajo seminal de Peter van Emde Boas sobre la estructura [1]. Por el momento nos limitaremos a hacer una breve definición.

El vEB se define como una estructura recursiva en la que $vEB(n)$ divide sus elementos en $\lceil\sqrt{n}\rceil$ instancias de $vEB(\lfloor\sqrt{n}\rfloor)$, lo que permite que la profundidad del $vEB(n)$ sea de $\mathcal{O}(\log\log n)$ ya que si n tiene $m = log_2(n)$ bits, cada recursión divide el número de bits por 2, lo que garantiza una profundidad de $\mathcal{O}(\log m)$. Adicionalmente, un elemento x en el $vEB(n)$, al ser un entero, puede ser representado como $x = a\lfloor\sqrt{n}\rfloor + b$, donde $0 < b < \lfloor\sqrt{n}\rfloor$, esto permite que x sea asignado al a-ésimo $vEB(\lfloor\sqrt{n}\rfloor)$ y que para dicho subárbol x sea tratado como b, que sólo necesita la mitad de bits de x para su representación. Dado que el UHTree no usa la implementación básica del vEB, no se la mostrará en el presente trabajo. Una implementación concisa es la que se encuentra en [7].

3.1.6. Diccionarios

Un diccionario o arreglo asociativo es una estructura abstracta que contiene un conjunto S de pares $(key, value)$ tal que no existen dos pares con el mismo key y $key \in A$ y $value \in B$. Las operaciones que la definen son las siguientes:

- Añadir un par $(key, value)$ a S.
- Remover un par $(key, value)$ de S.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

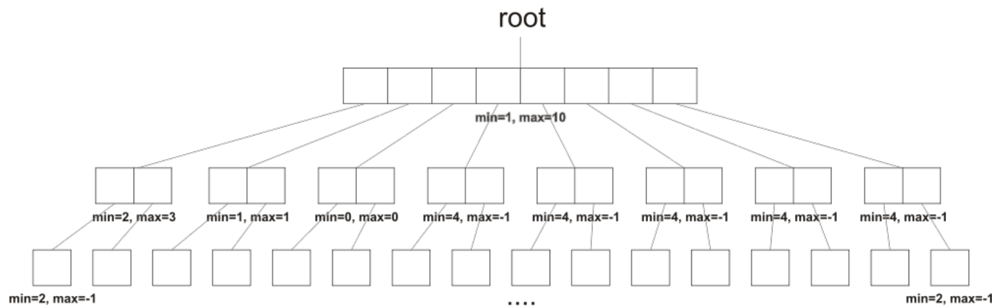


FIGURA 3.3: Ejemplo de un árbol de Van Emde Boas que contiene a los elementos 1, 2, 3, 5, 8 y 10, los cuales sólo necesitan 4 bits para representarse. El segundo nivel contiene a todos los pares con los dos bits más significativos iguales. El tercer nivel contiene a todos los elementos. Fuente: http://upload.wikimedia.org/wikipedia/commons/thumb/6/6b/VebDiagram.svg/1000px-VebDiagram.svg.png.

- Modificar un par $(key, value)$ en S.
- Hallar el $value$ asociado a un key determinado.

Por lo tanto, un diccionario no es más que la representación computacional de un mapeo dinámico

$$f : A \rightarrow B$$

Hay diversas maneras de implementar diccionarios. Dependiendo de los conjuntos A y B se pueden usar arreglos simples, árboles binarios de búsqueda, tablas hash [2], etc. En general, cualquier estructura E que permita almacenar elementos comparables (igualidad o desigualdad) puede ser usada para almacenar un diccionario, ya que E puede almacenar los $keys$ y a cada uno de estos elementos le puede asignar auxiliarmente una variable $value$. Por lo tanto, todo árbol de búsqueda puede ser modificado para que sea un diccionario.

3.1.7. Funciones Hash

Una función f es hash si puede ser definido de la siguiente manera [6]:

$$f : A \rightarrow B, B finito$$

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Generalmente, $|B| \leq |A|$, lo que permite trabajar con elementos de B en vez de hacerlo con los de A. En general, la siguiente expresión obvia es la que proporciona la ventaja de las funciones hash:

$$\text{Si } x, y \in A \text{ y } x = y \Rightarrow f(x) = f(y)$$

Por lo tanto, si el costo computacional de determinar si $x = y$ es mayor que el de determinar si $f(x) = f(y)$, entonces realizar lo último permite tener una condición necesaria para que $x = y$. Cabe mencionar que el hecho de que $f(x) = f(y)$ no implica que $x = y$, a menos que f sea una biyección, lo cual en general no es cierto.

Cuando el rango B es el conjunto de los enteros, realizar $f(x) = f(y)$ es $\mathcal{O}(f(x)) + \mathcal{O}(f(y))$ y si es que $\mathcal{O}(f(x)) + \mathcal{O}(f(y)) < \mathcal{O}(x = y)$ entonces es muy útil usar la función hash para determinar si dos elementos son diferentes, lo cual es absoluto, y además tener una condición necesaria cuando $x = y$. Si es que se va a realizar una cantidad considerable de comparaciones, de las cuales solo un pequeño porcentaje realmente serán de igualdad, verificar primero que $f(x) = f(y)$ y luego, en caso de ser cierto, $x = y$ conlleva a una disminución considerable en el tiempo de ejecución de esta tarea, lo cual es la característica más importante de las funciones hash. Como ejemplo, la siguiente función es hash:

$$f : S \to \mathbb{Z}_7, \text{ S es el conjunto de todas las cadenas finitas de dígitos}$$

$$f(s) \mapsto \text{suma de elementos de } s \pmod 7$$

Es claro que el rango es el conjunto de los enteros módulo 7, que es un conjunto mucho más pequeño que S, que es infinito. Adicionalmente, comparar si $s, t \in S$ son iguales necesitaría ver cada elemento de ambas cadenas, por lo que

$$\mathcal{O}(s = t) \in \mathcal{O}(min(longitud(s), longitud(t)))$$

Sin embargo, comparar $f(s) = f(t)$ es $\mathcal{O}(1)$ asumiendo que $f(s)$ y $f(t)$ ya han sido previamente calculado.

Tesis publicada con autorización del autor
No olvide citar esta tesis

3.1.8. Análisis amortizado

Es una manera de analizar la complejidad de un algoritmo en función de una secuencia de operaciones en vez de una sola operación. Se basa en el hecho de que algunas operaciones en un árbol pueden ser más costosas que otras, pero debido a su frecuencia, en conjunto todas las operaciones pueden incluso tener una complejidad pequeña, como en el caso de los arreglos dinámicos o la estructura Union-Find [2].

3.1.9. Heap

Un heap o montículo es una estructura abstracta basada en un árbol binario en el que cada nodo u tiene un valor $c(u)$. Así, la estructura debe cumplir que si v es un descendiente de u, entonces

$$c(u) > c(v).$$

3.1.10. Hilo

Un hilo de un proceso es una porción instrucciones que puede ser ejecutada por el procesador como si fuera un proceso independiente. A diferencia de un proceso independiente, un hilo comparte los mismos recursos que su proceso padre y por lo tanto necesita menos datos para su existencia, por lo que su manejo por parte del procesador es más eficiente. Son llamados procesos ligeros (light-weight process en inglés).

3.1.11. Lock

En concurrencia, un lock es un mecanismo de sincronización que permite que un sólo hilo acceda a un recurso (objeto, funciones, instrucciones, hardware, etc.). Específicamente en el lenguaje Java, un lock permite que sólo un hilo acceda a un recurso y todo otro hilo que quiera accederlo se queda paralizado hasta que el hilo poseedor libere el lock; luego, cualquiera de los hilos en espera obtiene el lock y la ejecución prosigue de la misma manera.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

En la terminología de concurrencia, el término lock-free es adjudicado a un recurso libre de cualquier tipo de locks, lo que permite que varios hilos lo accedan paralelamente.

3.2. Árboles balanceados para distribuciones no uniformes

En esta sección se darán a conocer las principales soluciones a este problema que se usan en la actualidad.

3.2.1. AVL Tree

Este árbol es el primer árbol de búsqueda autobalanceado en ser inventado. Se basa en el invariante de que dado un nodo, la profundidad de sus subárboles difieren a lo mucho en 1. Sin ir muy lejos en la explicación, si luego de una operación existe un nodo cuyos subárboles tienen alturas que difieren por más de 1, se aplican rotaciones sucesivas para garantizar el invariante.

Este árbol garantiza una profundidad menor a $1{,}44 \log_2(n+2) - 0{,}328$ [8], lo cual es $\mathcal{O}(\log n)$, y permite que las operaciones de inserción, eliminación y búsqueda sean realizadas en tiempo $\mathcal{O}(\log n)$.

Versiones concurrentes de este árbol existen y en la práctica es muy usando cuando las operaciones de búsqueda son las de mayor frecuencia, con lo que se realizan pocas rotaciones, y como la altura es bastante acotada, las búsquedas son muy rápidas [9].

3.2.2. Splay Tree

Este árbol es un árbol binario de búsqueda inventado por Robert Tarjan y Daniel Sleator [6] con la siguiente característica: cada vez que un nodo es accedido, por medio de rotaciones, él es llevado a la raíz del árbol. Esta operación es llamada splaying. Simplemente por esto, es posible demostrar lo siguiente a través de análisis amortizado [10]:

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- El costo de realizar m accesos en un árbol con n elementos es

$$\mathcal{O}(m(1 + \log n) + n \log n)$$

- Sea q_i el número de accesos del elemento i del árbol, entontes el costo de realizar m accesos es

$$\mathcal{O}(m + \sum_{i=1}^{n} q_i \log \frac{m}{q_i})$$

La primera propiedad establece que, en general, realizar m accesos tenga una complejidad similar al de un árbol binario óptimo, el cual es $\mathcal{O}(m \log n)$. Adicionalmente, tenemos por la segunda propiedad que cada elemento tiene contribucion por operación al costo total que se reduce a medida que el número de accesos a dicho elemento aumenta. Por lo tanto, los elementos accedidos más frecuentemente tienen un costo individual menor por operación que los menos frecuentes. Así, si hay un conjunto de elementos accedidos mucho más frecuentemente que el resto, el rendimiento de la estructura en conjunto puede ser menor que el de muchos otros árboles.

No obstante, esta estructura tiene puntos negativos en lo que corresponde a concurrencia, dado que, al acceder a un elemento u, todos los nodos en el camino desde la raíz hasta u van a ser parte de rotaciones hasta dejar a u como raíz, por lo tanto, cada uno de estos nodos detendrá a todo posible acceso a él o a sus descendientes en algún momento, empobreciendo su rendimiento ante entornos concurrentes.

3.2.3. Lazy Splay Tree

Este árbol es una modificación muy inteligente y simple del Splay Tree [11]. Se basa en que, en vez de que el un elemento accedido tenga que ser rotado hasta la raíz, sea rotado como máximo hasta el nivel superior inmediato.

En primer lugar, a cada nodo se le asigna un contador del número de veces que ha sido accedido. Luego, cuando un nodo es accedido, si su contador de accesos se vuelve mayor que el de su padre, entonces es rotado para ocupar el lugar de este último. Se ha demostrado [11] que este árbol tiene un rendimiento mayor al del Splay Tree común en la mayoría de casos y que en términos de concurrencia es una muy buena estructura, ya que permite que los elementos mas frecuentes estén cerca de la raíz y

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

al haber máximo una rotación por acceso, hay menos accesos que son detenidos por la concurrencia. Esta estructura es usada en muchos casos debido a su simplicidad, sin embargo, existen estructuras similares y más sotifisticadas como el CBTree, que tiene un rendimiento en la práctica mucho mayor.

3.2.4. CBTree

El Count Based Tree es un árbol recientemente inventado muy similar al Lazy Splay Tree pero con más sofisticaciones.

En primer lugar, tal como el Lazy Splay Tree tiene contadores que reflejan la cantidad de veces que un elemento ha sigo accedido, el CBTree lo tiene, pero en vez de garantizar que los contadores estén en una zona de exclusión mutua, simplemente obvia cualquier problema de integridad de los valores debido a la concurrencia y deja libre a estas variables. Esto aumenta el rendimiento por acceso, ya que cada acceso modifica todos los contadores en su camino de búsqueda. Adicionalmente, a pesar de que no hay integridad de los valores asegurada completamente, se ha demostrado [4] que se logra mejores resultados que teniendo integridad, ya que es muy poco probable que estos fenómenos ocurran y el valor exacto del contador tampoco es exigible.

Adicionalmente, las rotaciones se realizan usando las técnicas para árboles AVL concurrentes estudiadas en [9], las cuales permiten tener un muy buen rendimiento concurrente.

Como otra característica relevante, a diferencia del Lazy Splay Tree, que realiza máximo una rotación por acceso, el CBTree puede realizar varias rotaciones usando una técnica llamada Semi-splaying. En general, el CBTree intenta garantizar que los contadores decrezcan geométricamente a medida que se desciende en el árbol. Esto permite que el camino de la raíz a las hojas tenga altura logarítmica. Así, cada vez que se pueda garantizar esto realizando una rotación, esta última es efectuada.

La última característica relevante es que modifica el valor de los contadores según un función decaimiento que es aplicado a cada nodo cada cierto tiempo usando otro hilo. Esto resuelve el hecho de que elementos antiguos con contadores grandes demoran mucho en ser alcanzados por elementos nuevos y frecuentes, a pesar de que los antiguos no sean vistos nuevamente.

Tesis publicada con autorización del autor
No olvide citar esta tesis

Finalmente, se muestran dos propiedades similares a las del Splay Tree común:

- El costo de realizar una secuencia de m accesos en un árbol con n elementos cuyas frecuencias de acceso son $q_1, q_2, \ldots, q_n$ es

$$\mathcal{O}(m + \sum_{i=1}^{n} q_i \log \frac{m}{q_i})$$

- Por lo anterior, el costo de realizar un acceso al elemento i-ésimo puede ser representado como

$$\mathcal{O}(\log \frac{m}{q_i})$$

Así, a pesar de las ventajas en concurrencia, la complejidad teórica de los accesos es equivalente al del Splay Tree.

3.2.5. Treap

Tree + Heap. Es un árbol T donde cada nodo u tiene dos valores $c(u)$ y $k(u)$ tal que T es un árbol binario de búsqueda según $c(u)$ y es un heap según $k(u)$. Así, se cumple lo siguiente:

$$c(u) >= c(HijoIzquierdo(u))$$

$$c(u) <= c(HijoDerecho(u))$$

$$k(u) >= k(HijoIzquierdo(u))$$

$$k(u) >= k(HijoDerecho(u))$$

Esta estructura no tiene como propósito ser usado como un heap, sino simplemente como un árbol de búsqueda. El objetivo de mantener un heap es el de tener un árbol probabilísticamente balanceado. A grosso modo, cada vez que un nuevo elemento v con valor $c(v)$ es añadido a T, aleatoriamente se genera su $k(v)$, luego, se inserta v en el árbol de manera que termine siendo un árbol binario de búsqueda según $c(v)$ y un heap según $k(v)$. El aporte del k es que este define la forma que tendrá el árbol. Se sabe que un árbol binario de búsqueda, luego de aplicarse cualesquiera rotaciones,

Tesis publicada con autorización del autor
No olvide citar esta tesis

seguirá siendo de búsqueda. Por lo tanto, dados ciertos elementos, existen muchos árboles binarios de búsqueda que los contengan; sin embargo, el uso de k fija la forma que tendrá [12].

k es generado aleatoriamente con distribución uniforme en un intervalo definido. Así, se puede demostrar que la probabilidad de que un árbol T de n elementos tenga cierta forma es casi $1/g(n)$ [12] donde $g(n)$ es la cantidad de árboles binarios de n elementos no numerados. Adicionalmente, se sabe que la profundidad esperada de un árbol binario de n elementos está en $\mathcal{O}(\log n)$ [12], lo que implica que la profundidad esperada de T esté en $\mathcal{O}(\log n)$. Esta interesante propiedad implica que con rotaciones que no usan información sobre la altura de cada subárbol el árbol logre estar bien balanceado.

En [9] se realizó una implementación bastante buena de una versión concurrente del Treap. Dependiendo de la aplicación, el Treap puede funcionar tan bien o mejor que otras estructuras. Sin embargo, para propósitos generales estructuras como el CBTree tienen mejor rendimiento.

3.2.6. Tango Tree y Zipper Tree

La definición y estudio de estos dos árboles de búsqueda escapan los objetivos de este proyecto, pero por su importancia son mencionados [13] [14]. Ambos árboles binarios se basan en descomponer el árbol en muchos caminos, cada uno representado por un árbol. Son árboles muy complejos no ideados para entornos concurrentes, mas sí para distribuciones no uniformes. Sin embargo, para entornos no concurentes logran una complejidad de tiempo por acceso, cuando hay n elementos, de $\mathcal{O}(f(n) \log \log n)$, donde $f(n)$ es la complejidad del mejor algoritmo offline para dichos n elementos.

Los árboles Tango no tienen una complejidad por operación fácilmente definida, empero los Zipper sí están en $\mathcal{O}(\log n)$.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

3.3. Conclusiones del estado del arte

Luego de haber hecho una mención de las soluciones más usadas actualmente, así como de las técnicas que estas usan, se puede concluir en los siguientes puntos relevantes para el proyecto.

- La desventaja principal de todas estas soluciones es realmente la operación de rotación, que ralentiza la concurrencia, por lo que se debe evitar operaciones similares.
- Las mejores soluciones intentan realizar un número limitado de locks por operación, sin importar la profundidad de los nodos en cuestión.
- La técnica de ordenar los nodos en el árbol según su frecuencia de acceso es la principal ventaja de las mejores soluciones (CBTree y Lazy Splay Tree).
- Todas estas soluciones utilizan muy poca memoria por nodo, por lo que se debería intentar no usar mucha en el UHTree.
- Estructuras más complejas, como el Tango Tree y Zipper Tree, no soy eficientes en la práctica a pesar de su optimalidad teórica; así que evitar la complejidad en la implementación es un objetivo importante en el UHTree.

Ests observaciones fueron vitales en el diseño e implementación del UHTree.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Capítulo 4

Diseño de la estructura UHTree

En este capítulo se presenta el diseño de los componentes del UHTree.

4.1. Diseño

4.1.1. Definición

El UHTree es un árbol de búsqueda concurrente cuyo dominio es cualquier objeto representable en el computador, sea este dominio D, el cual es convertido en un número natural acotado por $N = 2^{2^M}, M \in \mathbb{N}$. Esta transformación se puede lograr usando una función hash sobre el objeto, o en el caso más simple, sobre su representación como cadena.

4.1.1.1. Propiedades

Primero se presentan las propiedades del UHTree, para entender con mayor profundidad sus ventajas, desventajas y usabilidad. Así también se facilita llegar a una mejor compresión del diseño específico.

- La estructura puede almacenar cualquier tipo de objeto, dado que un número entero hash de este objeto es provisto.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

- Tiene complejidad de memoria $\mathcal{O}(n)$, donde n es el número de elementos en el árbol.
- Tiene complejidad esperada de tiempo por operación $\mathcal{O}(\log \log m)$, donde m es un entero positivo arbitrario. Sin embargo, esta operación es amortizada, ya que internamente tablas hash son usadas para resolver colisiones, por lo que a menor m, las tablas hash son más lentas. Asimismo, a mayor m, las tablas hash son más rápidas. Experimentalmente, un valor de m de 4 o 5 es suficiente para un n máximo de 2^{32}.
- Por lo anterior, el UHTree tiene altura independiente del número de elementos.
- El nodo definido por $UHTree(x)$, tiene x hojas y $\mathcal{O}(\sqrt{x})$ hijos directos.
- Cada nodo contiene al elemento más frecuente en su subárbol. Esto quiere decir que en los primeros dos niveles del árbol se encuentran almacenados los $\sqrt{n}$ elementos más frecuentes. Considerando que la mayoría de búsquedas basadas por una distribución de Zipf son sobre menos de los $\sqrt{n}$ elementos más probables, todas estas búsquedas revisan a lo más 2 nodos, disminuyendo considerablemente el tiempo de acceso.
- No existen rotaciones.
- Las búsquedas no causan locks sobre los nodos, sólo sobre algunas variables de los nodos.
- Las inserciones y remociones causan locks muy severos sobre el nodo que modifican.
- La implementación es compleja y las operaciones individuales son costosas.

4.1.1.2. Definición de *UHElement*

- Nombre: $UHElement(o), o \in D$.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

- Estado:

  $\mathbf{Ob}$: $Ob = o$, que es un objeto cualquiera.

  $\mathbf{count}$: contador del número de accesos a Ob.

- Operaciones:

  $\mathbf{incr}$: incrementa $count$ en una unidad.

4.1.1.3. Definición de *UHList*

- Nombre: $UHList()$.

- Estado:

  $\mathbf{list}$: lista de $UHElement$ de ciertos objetos con el mismo hash.

- Operaciones:

  $\mathbf{addNewElement(o)}$: agrega un nuevo $UHElement$ o un $UHList$ a $list$.

  $\mathbf{contains(o)}$: determina si o está dentro de $list$

  $\mathbf{isEmpty}$: determina si $list$ está vacío.

  $\mathbf{remove(o)}$: elimina o de $list$.

4.1.1.4. Definición de *UHNode*

El UHNode se define de la siguiente manera recursiva:

- Nombre: $UHTree(n = 2^{2^m})$.

Tesis publicada con autorización del autor
No olvide citar esta tesis

- Estado:

  min: menor hash de los elementos contenidos en el árbol. Se debe cumplir adem'as que ningún hijo del árbol contenga a min, lo cual garantiza complejidad $\mathcal{O}(k)$ en memoria, donde k es el número de elementos distintos en el árbol.

  minElements: $UHList$ de los elementos con hash igual a min.

  max: $2^{2^m} - 1$.

  order: n.

  shift: 2^{m-1}.

  isEmpty: flag sobre si el árbol está vacío.

  best: $UHElement$ del objeto más frecuentemente accesado en el árbol.

  LQ: arreglo de hijos de este árbol. Hay $max+1$ hijos, cada uno es un $UHTree(\sqrt{n})$.

  HQ: $UHTree(\sqrt{n})$: que contiene los índices de los hijos no vacíos.

- Operaciones: ($o \in D$, $hash(o) = a \cdot \sqrt{n} + b$, $0 \leq b < \sqrt{n}$)

  isEmpty: determina si el árbol está vacío.

  getMinHash: retorna min.

  getMinElements: retorna $minElements$.

  $\mathbf{add(hash(o), o)}$: inserta $UHElement(o)$ al árbol. Si el árbol está vació o $min = hash(o)$, $UHElement(o)$ es insertado en $minElements$. En caso contrario, $UHElement(o)$ es insertado en $LQ[a]$ con hash igual a b.

  $\mathbf{updateBest(uhElement)}$: si $uhElement$ tiene mayor frecuencia de acceso que $best$, $best \leftarrow uhElement$.

  $\mathbf{contains(hash(o), o)}$: determina si el árbol contiene al elemento o. Si $best = o$, para la recursión. En caso contrario, busca en $LQ[a]$ con hash igual a b.

  $\mathbf{remove(hash(o), o)}$: remueve o del árbol. Si $hash(o) = min$, remueve o de $minElements$. Si esto ocaciona que $minElements$ esté vacío, $minElement$ y min obtienen sus nuevos valores de los nuevos elementos mínimos del árbol, consecuentemente elimina estos valores de su subárbol. Si $hash(o) \neq min$, remueve o de $LQ[a]$ con hash igual a b.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

4.1.1.5. Definición de *UHTree*

- Nombre: $UHTree(n = 2^{2^m})$.
- Estado:

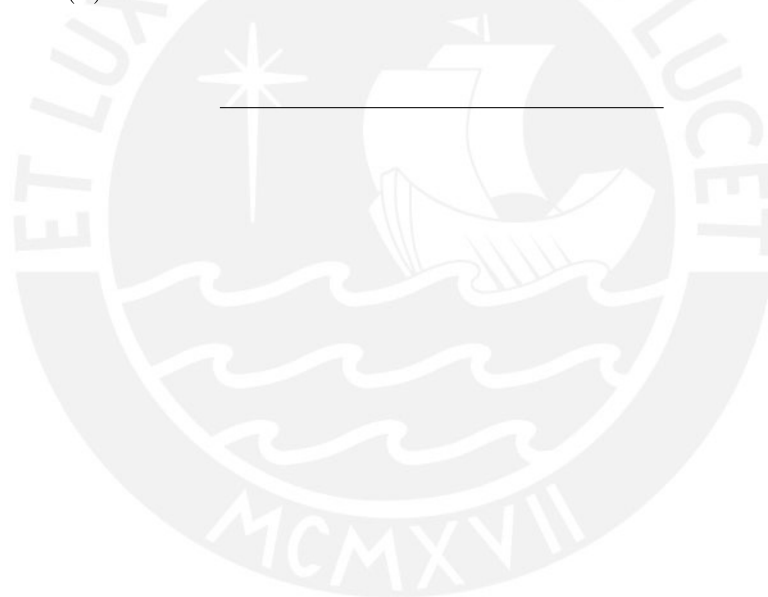
  $\mathbf{root}$: raíz del árbol. Es un $UHNode(n)$.
- Operaciones:

  $\mathbf{addNewElement(o)}$: agrega o a $root$.

  $\mathbf{contains(o)}$: determina si o está dentro de $root$

  $\mathbf{isEmpty}$: determina si $root$ está vacío.

  $\mathbf{remove(o)}$: elimina o de $root$.



Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

4.1.2. Detalles de implementación

- Las llamandas a $isEmpty$, $getMinHash$ y $getMinElements$ son lock-free, dado que son atómicas.
- La llamada a $updateBest$, a pesar de no ser atómica, ya que envuelve comparaciones de contadores, se asumirá que es atómica y por lo tanto lock-free, ya que a pesar de que existan casos en que el mejor elemento no sea el preferido, eventualmente lo será y es más lento sincronizar esto que permitir ciertas inconsistencias que no eliminan la correctitud de la estructura.
- Las llamadas a add y $remove$ bloquean todo acceso al nodo actual.
- Las llamadas a $contains$ son lock-free, salvo cuando modifica algún elemento. Esto permite que la estructura sea bastante rápida, ya que el número de llamadas a $contains$ excede por mucho a las de add y $remove$.
- Finalmente, $s.incr$ se ejecuta cada vez que el elemento s es buscado.

Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Capítulo 5

Análisis de la estructura UHTree

En este capítulo se procede a analizar la correctitud y complejidad de la estructura y de sus componentes.

5.1. Análisis de la estrutura

Se procede a analizar la estructura no bajo pseudocódigo, sino bajo el código mismo que se encuentra en el anexo A. Específicamente se trabaja sobre la versión https://github.com/a20012251/uhtree/commit/0d3f00f50453506418e0767401cc2cc7c0465d21.

5.1.1. Análisis de UHElement

Esta estructura no tiene nada relevante para comentar. Todas sus operaciones con $\mathcal{O}(1)$, salvo el método $Equals$ que llama al método del mismo nombre para el valor almacenado en $UHElement$. Esta función es definida por el usuario o tiene una complejidad desconocida que se la asume constante. Dado que sus variables son inmutables, no se necesita sincronizarlo.

5.1.2. Análisis de UHList

Esta estructura, que almacena una lista de $UHElements$, tambiés es simple. Es una composición de ConcurrentHashMap, que es implementado nativamente por Java y

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

tiene una complejidad esperada de $\mathcal{O}(1)$. Por lo tanto, las llamadas a $remove$, $isEmpty$ y $contains$ son $\mathcal{O}(1)$. El método $contains$, en caso de que el elemento buscado exista en la lista, llama al método $updateBest$ del objeto que lo llamó, el cual se verá que es $\mathcal{O}(1)$. Finalmente, el método $addNewElement$ puede almacenar o bien un único elemento, que es $\mathcal{O}(1)$, o bien una lista de elementos. En el segundo caso, que no es de tiempo constante, se cumple que cada elemento es ingresado en la lista una única vez en toda su vida, por lo tanto en tiempo amortizado es $\mathcal{O}(1)$ para cada elemento. Finalmente, se concluye que la complejidad amortizada para toda operación es $\mathcal{O}(1)$. Dado que el UHList hace uso de un ConcurrentHashMap para almacenar sus datos, esta sincronizado por defecto en las operaciones de lectura y escritura.

5.1.3. Análisis de UHBitmaskHeap

Esta estructura es un heap simple de enteros. A pesar de no haber mencionado a esta estructura en el diseño, en la práctica se la implementó por fines de rendimiento cuando el número máximo de elementos en el heap es menor a 32. Cabe mencionar que su existencia es prescindible en la correctitud del UHTree. El UHBitmaskHeap no es más que un heap de números enteros implementado usando un máscara de bits. Las operaciones de inserción, remoción, búsqueda y menor elemento son implementadas usando operaciones a nivel de bits, por lo que presenta complejidad constante en todas sus operaciones. Su código es muy simple y no merece mayores comentarios.

5.1.4. Análisis de UHNode

Esta estructura representa un nodo del UHTree, y particularmente es el caso de la raíz, por lo que el análisis del UHTree se centra más que todo en el del UHNode. Se debe cumplir el invariante (I) de que los elementos con menor hash en un UHNode sean almacenados en él y no en ningún descendiente, garantizando la linealidad de memoria. Adicionalmente, las variables no finales del UHNode son volatiles, lo que en Java significa que las operaciones de lectura y escritura de estas variables (no de sus datos internos) son atómicas.

Tesis publicada con autorización del autor
No olvide citar esta tesis

5.1.4.1. Método isEmpty

Este método determina si la estructura está vacía. Dicha información está almacenada en una variable $isEmpty$. Dado que este método sólo retorna esta variable, es $\mathcal{O}(1)$ y es muy simple, sin problemas de concurrencia.

5.1.4.2. Métodos getMinHash y getMinElement

Estos métodos son muy similares al anterior, ya que sólo retornan una variable. Son $\mathcal{O}(1)$ y sin problemas de sincronización.

5.1.4.3. Método updateBest

Este método actualiza el elemento más accedido del subárbol. Es constante y se asume que es lock-free para mejorar el rendimiento, ya que su exactitud es prescindible.

5.1.4.4. Método add

Este método recorre iterativamente los nodos usando el while de la línea 76. Desde la línea 78 hasta la 85, se realiza el caso en que el nuevo hash ingresado es un hash mínimo, por lo que sólo se actualiza min y $minList$. Además, se para la recursión. Esta parte es $\mathcal{O}(1)$.

La siguiente sección es más elaborada. Si el hash ingresado es mayor al hash mínimo actual (línea 106), se procede a dividir el hash en dos números, su mitad más significativa y su mitad menos significativa en binario. La primera es el índice del UHNode hijo a visitar, y la segunda es el hash que se usará en el nuevo nodo. En la línea 110 se crea si es necesario el nodo hijo, lo cual es $\mathcal{O}(1)$; finalmente, en las líneas 115 y 116 se actualiza el hash a insertar y el nodo siguiente.

La última condición (líneas 88-104), analizan el caso cuando el hash ingresado es menor que el mínimo del nodo actual. En esta sección se almacena el valor a ingresar y el nuevo hash mínimo en $minElements$ y en min, dejándolos en el nodo actual. Adicionalmente, el min y el $minElements$ anterior son tratados como elementos a insertar

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

en el nivel siguiente de nodo, para mantener la invariante (I). Esta nueva inserción se ejecuta del mismo modo que en el caso anterior, en que el hash era mayor al mínimo.

Por lo tanto, la complejidad de la tarea en cada nivel de nodo es $\mathcal{O}(1)$ y dado que hay $\log_2 \log_2 n$ niveles como máximo, la complejidad total es $\mathcal{O}(\log \log n)$.

Finalmente, todo el bloque de código dentro del while que itera sobre cada nodo está sincronizado, de manera que mientras se escribe en un nodo, es imposible modificarlo desde otro hilo, evitando así inconsistencias.

5.1.4.5. Método contains

Este método recorre el árbol de una manera similar al del método add, salvo que es más eficiente en concurrencia, lo que es bueno, ya que usualmente el número de búsquedas es mayor que el de modificaciones.

Este método, dentro del while que recorre los nodos necesarios hasta encontrar el elemento buscado, no sólo mantiene una variable con el nodo actual, sino, además, mantiene al nodo padre del actual, ya que si se encuentra el elemento buscado en el nodo actual, es posible que este se vuelva el más frecuente de su padre, y entonces se actualiza dicho valor. En las líneas 139-151, de manera sincronizada, se verifica si el elemento buscado es el más frecuente del nodo actual. En caso que sea esto cierto, se actualiza, cuando fuere necesario, al más frecuente del padre del nodo actual.

Si el nodo está vacío (líneas 152-153), simplemente se retorna falso. Y si el hash del elemento buscado es el mismo que el min del nodo actual (154-159), se revisa en la lista de mínimos si se encuentra. Esta verificación es sincronizada, ya que $minElements$ no debe cambiar en este proceso.

Por último, si el elemento buscado aún no fue encontrado, se desciende al nodo hijo correspondiente y se lo busca ahí, sólo en caso que dicho hijo exista. Esta parte no necesita sincronización, ya que no se altera ningun objeto importante.

Este método tiene complejidad $\mathcal{O}(\log \log n)$, ya que en el peor de los casos se desciende a un nodo hoja en $\log \log n$ pasos, y se lo busca ahí.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

5.1.4.6. Método remove

Este método, a diferencia de los anteriores, es recursivo y totalmente sincronizado en cada recursión, debido a que modifica muchas variables y requiere que no se realicen modificaciones en el nodo actual.

En la recursión, primero se verifica si el elemento más frecuente del nodo actual es el buscado (líneas 185-188), si es así, se lo elimina, pero no para la ejecución, puesto que dicho elemento más frecuente se encuentra también en un $UHList$.

Si el nodo actual está vacío, se debe returnar falso (líneas 190-191). Adicionalmente, si el hash buscado es igual a min, se debe explorar la lista de mínimos. Si no está dentro (206-207), se retorna falso; en caso contrario, se lo elimina de la lista de mínimos y esta lista queda vacía, se determina el nuevo mínimo del árbol (líneas 216-226) haciendo uso de HQ, que es un UHNode que almacena los nodos hijos que contienen al menos un elemento. Toda esta parte es $\mathcal{O}(1)$, debido a la búsqueda en HQ que es constante. Como último detalle, los nuevos elemento de $minList$ deben ser elminados del nodo inferior de donde fueron obtenidos, por lo que se asigna como objeto y hash a eliminar a este $minList$ y su hash correspondiente (líneas 220-225).

Finalmente, si hay algun elemento por eliminar (o bien el elemento con el que se inició la recursión o algún minList por eliminar), se calcula el nodo hijo al que se debe ir y el hash a elminar, y se prosigue la recursión (líneas 233-243).

Se observa que la complejidad de una recursión es $\mathcal{O}(1)$, por lo que en el peor caso toda la ejecución de remove es $\mathcal{O}(\log \log n)$.

5.1.5. Conclusiones del análisis

A modo explicativo del algoritmo, se demostró que los métodos principales de add, $remove$ y $contains$ son $\mathcal{O}(\log \log n)$ amortizado, ya que hacen como máximo $\log_2 \log_2 n$ visitas de nodos y usan una lista hash interna que es constante amortizado. Adicionalmente, los métodos $isEmpty$, $getMinHash$ y $getMinElement$ son constantes. En lo que respecta a concurrencia, los métodos add y $remove$ hacen lock del nodo actual la mayor parte del tiempo, mientras que el método $contains$, que es el más frecuente, lo hace muy poco, lo que es eficiente. El código fue inspirado en [15]. Teóricamente

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

es mucho más eficiente que cualquier árbol binario conocido hasta el momento. Sin embargo, el largo código, la gran cantidad de variables dentro de cada UHNode y los locks excesivos al insertar y remover compensan la ganancia de complejidad con una constante grande en las operaciones.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Capítulo 6

Comparación numérica

En este capítulo se presenta el análisis comparativo entre el UHTree y otras estructuras similares usando tres pruebas basadas en datos similares a la realidad:

1. Distribución de Zipf con valores de sesgo 1.4.
2. Concatenación de 5 libros en inglés, con palabras en minúsculas, del proyecto Gutenberg, tal como se realizó en [4]. Estos libros, elegidos por el autor, son:
   - The Mahabharata of Krishna-Dwaipayana Vyasa, Adi Parva
   - The Mahabharata of Krishna-Dwaipayana Vyasa, Sabha Parva
   - The Mahabharata of Krishna-Dwaipayana Vyasa, Vana Parva, Part 1
   - The Mahabharata of Krishna-Dwaipayana Vyasa, Vana Parva, Part 2
   - The Mahabharata of Krishna-Dwaipayana Vyasa, Virata Parva

   Su concatenación resulta en una lista de aproximadamente 710204 palabras.

Para el primer experimento, basado en la distribución de Zipf, aleatoriamente se realizan búsquedas, inserciones y remociones, siendo las búsquedas las operaciones más frecuente; las inserciones las siguientes en frencuencia; y las remociones las menos frecuentes.

Para el segundo experimento, sólo se realizan búsquedas sobre los libros, para probar la eficiencia ante un árbol muy denso en todo momento.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Para el tercer experimento, también sobre los libros, se realizan búsquedas, inserciones y remociones de la misma manera que en el primer experimento.

Las pruebas fueron realizadas en un computador UltraSPARC T3, de 16 núcleos de 1.65 GHz cada uno, con un máximo de 128 hilos por hardware. Además, el UHTree tenia un universo de 2^{2^4} valores enteros.

En todas las pruebas se realizan casos con 1, 2, 4, 6, 8, 16, 32 y 64 hilos.

6.1. Estadísticas

6.1.1. Rendimiendo con datos generados aleatoreamente usando distribución de Zipf

Entorno:

- Generación de datos aleatorios con distribución de Zipf usando un sesgo de 1.4, i.e. la probabilidad de aparición del elemento i-ésimo del dominio es $(1-(1/1{,}4))^i$.
- Dominio de datos: enteros $\in [0, 130000[$.
- Comparación entre AVL Tree [9], Lazy CBTree [4] y UHTree.
- El número de lecturas es el 80 % de todas las operaciones, siguiendo el esquema de [3].
- Probabilidad de realizar una lectura: 80 %.
- Probabilidad de realizar una escritura: 15 %.
- Probabilidad de realizar una remoción: 5 %.

6.1.2. Rendimiento con datos del proyecto Gutenberg (sólo lecturas)

Entorno:

- Dominio de datos: concatenación de libros arriba mencionados.

Tesis publicada con autorización del autor
No olvide citar esta tesis

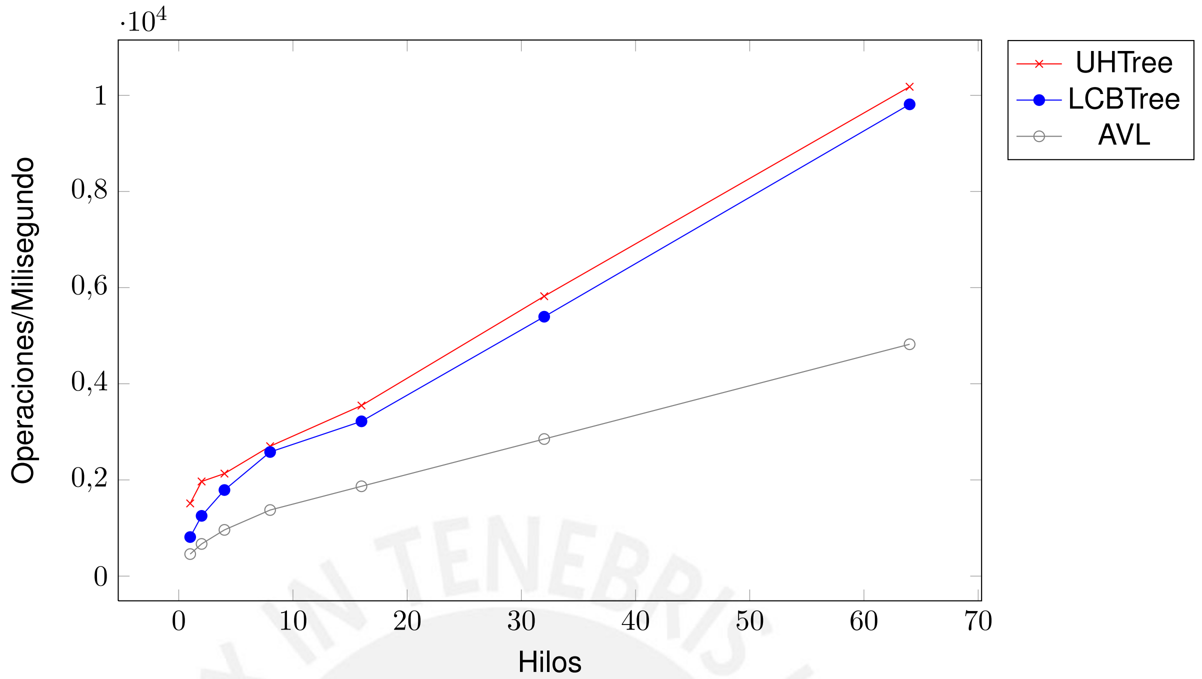


FIGURA 6.1: Resultados usando datos generados aleatoriamente.

- Comparación entre AVL Tree [9], Lazy CBTree [4] y UHTree.
- Sólo lecturas sobre las palabras del dominio seleccionándolas aleatoriamente según su probabilidad de aparición.

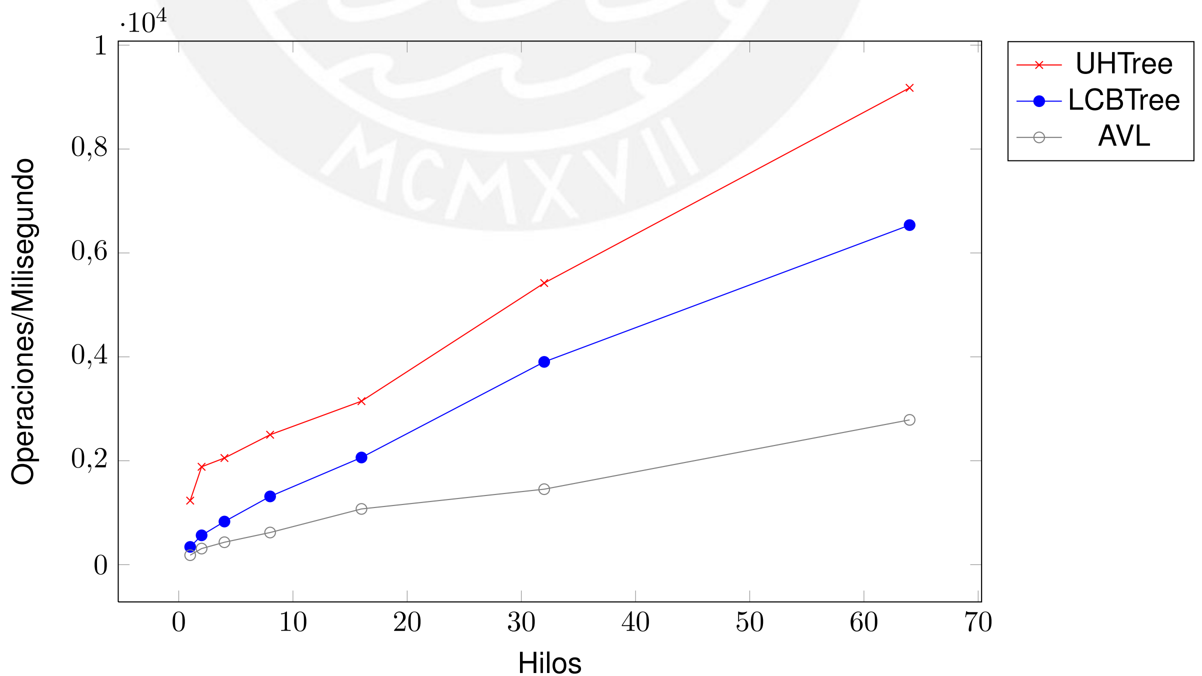


FIGURA 6.2: Resultados con datos del proyecto Gutenberg realizando sólo lecturas.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

6.1.3. Rendimiento con datos del proyecto Gutenberg (caso general)

Entorno:

- Dominio de datos: concatenación de libros arriba mencionados.
- Comparación entre AVL Tree [9], Lazy CBTree [4] y UHTree.
- El número de lecturas es el 80 % de todas las operaciones, siguiendo el esquema de [3].
- Las palabras del dominio son seleccionadas aleatoriamente según su probabilidad de aparición.
- Probabilidad de realizar una lectura: 80 %.
- Probabilidad de realizar una escritura: 15 %.
- Probabilidad de realizar una remoción: 5 %.

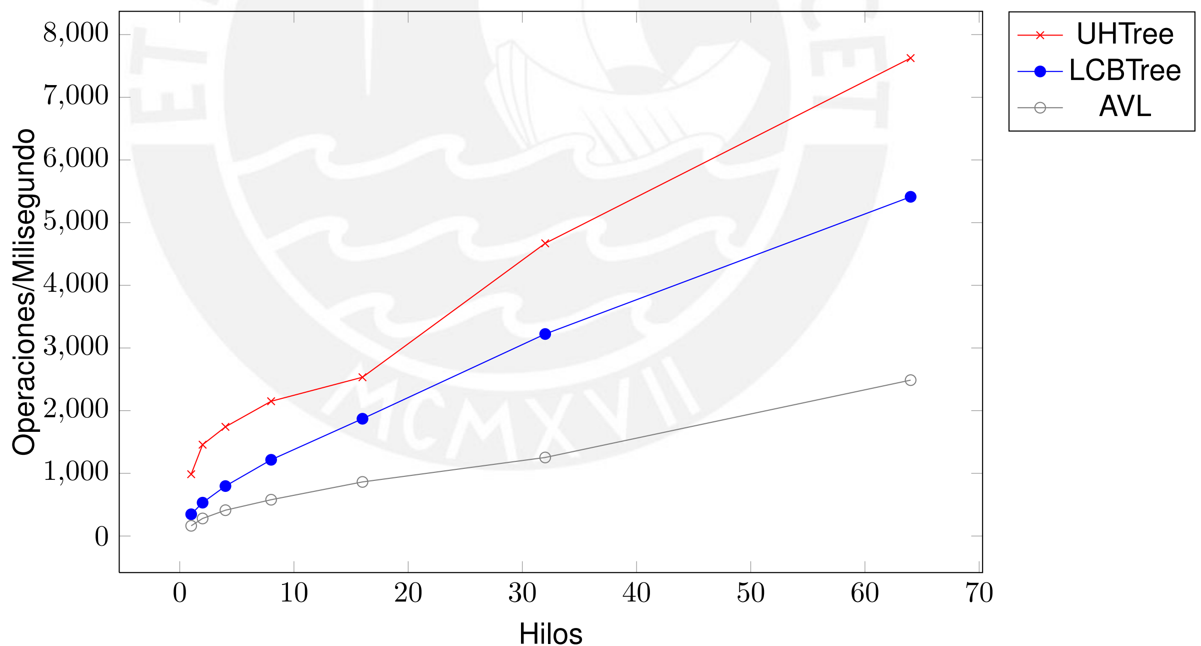


FIGURA 6.3: Resultados con datos del proyecto Gutenberg realizando lecturas y escrituras.

Nota: en estos experimentos no se han incluido otras estructuras como el Lazy Splay Tree, Treap, entre otros, dado que se sabe que el LCBTree los supera considerablemente [4]. El AVL Tree fue analizado también por ser una algoritmo pionero en concurrencia con autobalanceo [9].

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

6.2. Conclusiones de la experimentación

Se ha podido mostrar que el UHTree presenta un mejor rendimiento que otras estructuras en un entorno real (distribución de datos real y un número de elementos en el orden de los millones). En el caso en que sólo se realizan búsquedas, es por mucho mejor que el resto; y cuando existen modificaciones, es también más eficiente, pero con una diferencia ligeramente menor. Esto se puede explicar observando el número de operaciones realizadas por milisegundo. En el gráfico 6.1, desde 1 hilo hasta 32, el UHTree realiza aproximadamente 800 operaciones más por milisegundo que el LCB-Tree, y muchas más que el AVL. Cuando 64 hilos son procesados, la diferencia es más de 2000. En gráfico 6.2 la diferencia es casi similar, favoreciendo también al UHTree.

En experimento 6.1, que es bajo un entorno artificial, donde el número de elementos no es tan grande, el UHTree es prácticamente igual en eficiencia que el LCBTree; lo supera por menos de 200 operaciones por milisegundo en promedio en todas las configuraciones de hilos.

Ante esto, se podría concluir que el UHTree es mejor que otras estructuras como el LCBTree; sin embargo, sería una afirmación muy precipitada, ya que sólo se usó un procesador UltraSPARC. Sería necesario probar con un procesador Intel y con otros juegos de datos. No obstante, se puede afirmar que la estructura tiene un rendimiento muy superior a las estructuras dominadas por el LCBTree, las cuales son la mayoría [4].

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Capítulo 7

Observaciones, conclusiones y recomendaciones

En este capítulo se presentan las observaciones y conclusiones obtenidas en el proyecto. Además, se presentan las recomendaciones pertinentes para trabajos futuros.

7.1. Observaciones

La estructura desarrollada en el presente proyecto ha presentado versatilidad en el manejo de datos tal como las otras estructuras del estado del arte y permite ser usado en cualquier aplicación donde se necesite un árbol de búsqueda.

En primer lugar, se analizaron las estructuras más comunes y de propósito general y se detectaron sus debilidades en el ámbito concurrente, que principalmente era el balanceo. Esto permitió enfocar la nueva estructura de manera que intente evitar este problema, por lo que se decidió usar las ideas del árbol de van Emde Boas, lo cual presentó un rendimiento mejor de lo esperado.

Para el desarrollo de la implementación se usaron las ideas extraídas pricipalmente de las estructuras de [15] y [9], lo cual permitió una buena estructuración que facilitó esta etapa.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

En la sección de experimentación se comparó el UHTree con la mejor estructura mencionada en el estado del arte (CBTree [4]) y con otra que fue referencia para el manejo de hilos (AVL Tree [8]), que además fue un trabajo pionero en la materia.

7.2. Conclusiones

Las conclusiones a las que se llegaron al finalizar este trabajo de fin de carrera son:

- Se realizó el diseño de la estructura que fue satisfactoriamente implementada, cumpliendo con los propiedades establecidas y en el plazo establecido. Se usaron muchas de las ideas de las estructuras del estado del arte, lo cual permitió obtener un diseño más eficiente.
- Se realizó el análisis de la implementación y se demostró su correctitud. Cabe mencionar que la complejidad obtenida en tiempo es $\mathcal{O}(\log \log n)$, lo cual es más eficiente que la mayoría de estructuras usadas comúnmente.
- Se realizó una librería junto con sus pruebas de integración. Dicha librería se encuentra en los anexos del proyecto. Se logró ejecutarla en diversas computadoras y con una eficiencia mejor que la esperada.
- Se realizó la comparación numérica con otros algoritmos muy eficientes en la literatura y se demostró que el UHTree es tan bueno como el LCBTree (el mejor de la literatura) en un entorno artificial creado con un generador aleatorio. Además, se demostró que en entornos reales (experimentos 6.2 y 6.3) este algoritmo superó significativamente a los otros con los que se comparó, lo cual fue mejor de lo esperado y demostró que, en entornos concurrentes y con distribución de datos no uniformes, una estructura de por sí compleja como el árbol de van Embe Boas puede modificarse para superar a estructuras más simples pero con balanceo.

7.3. Recomendaciones y trabajos futuros

Debido a que no se ha explotado mucho la optimización de las estrategias de sincronización, se podría crear una mejor implementación que aumente el rendimiento.

Tesis publicada con autorización del autor
No olvide citar esta tesis

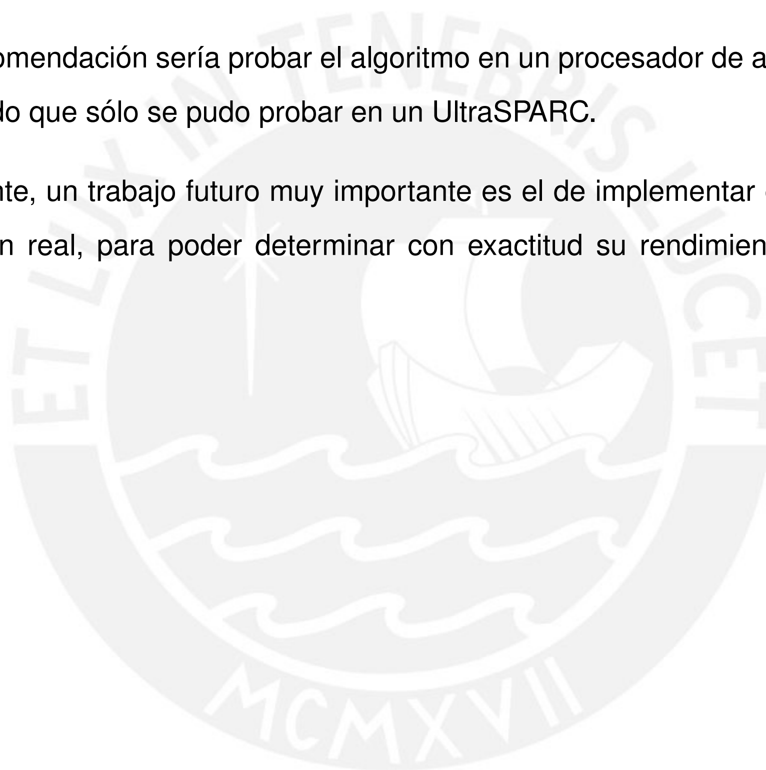
TESIS PUCP

Específicamente, los métodos de adición e inserción en el UHTree bloquean completa o parcialmente durante casi toda su ejecución el nodo en inspección, lo que bloquea a los hilos que quieran accederlos. Si se reduce el tiempo de bloqueo de estos nodos, el rendimiento en general se vería muy favorecido.

Otra característica que no se ha explotado es la de convertir la estructura en una distribuída. Esto significa que la estructura no sólo se almacene en una única computadora, sino en muchas. Esto permitiría que las peticiones se distribuyan entre muchas computadoras, disminuyendo el número de procesos bloqueados y aumentando el paralelismo por el número de procesadores en uso.

Otra recomendación sería probar el algoritmo en un procesador de alto paralelismo de Intel, dado que sólo se pudo probar en un UltraSPARC.

Finalmente, un trabajo futuro muy importante es el de implementar el UHTree en una aplicación real, para poder determinar con exactitud su rendimiento en un entorno práctico.



Tesis publicada con autorización del autor
No olvide citar esta tesis

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Apéndice A

Código del UHTree

El código del UHTree es de libre acceso y se encuentra en https://github.com/a20012251/uhtree.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Bibliografía

[1] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, (10):75–84, 1975.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.

[3] Jim Bell and Gopal Gupta. An evaluation of self-adjusting binary search tree technique. 1992.

[4] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. CBTree: A practical concurrent self-adjusting search tree. 2012.

[5] Matthias Müller-Hannemann and Stefan Schirra (Eds.). *Algorithm Engineering*, volume 5971. 2010.

[6] Donald Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1968.

[7] Gudmund Skovbjerg Frandsen. Dynamic algorithms: Course notes on van emde boas trees (pdf), 2004.

[8] Donald Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2 edition, 2000.

[9] Nathan G. Bronson, Jared Casper, Hassan Cha, and Kunle Olukotun. A practical concurrent binary search tree. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (10):257–268, 2010.

[10] Daniel Sleator and Robert Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, (32(3)):652–686, 1985.

Tesis publicada con autorización del autor
No olvide citar esta tesis

TESIS PUCP

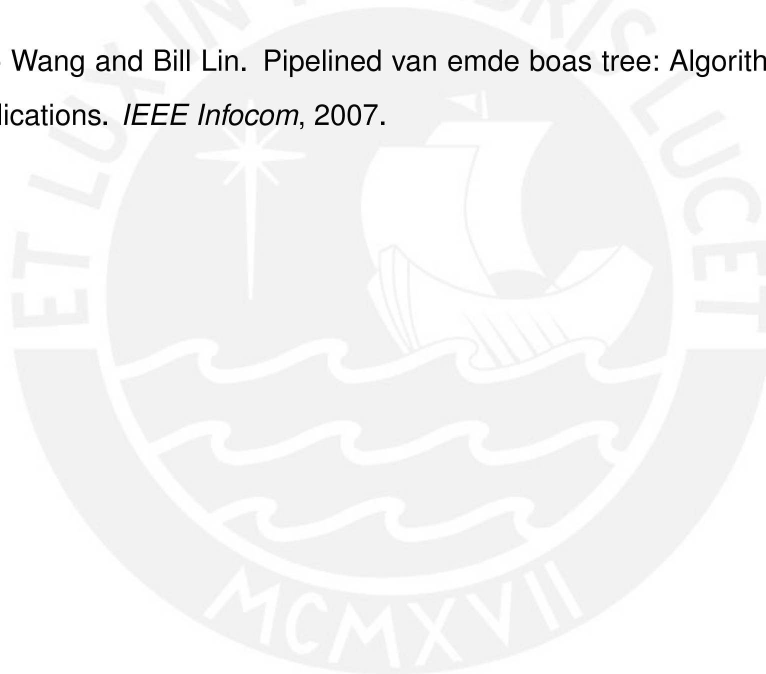
[11] Yehuda Afek, Boris Korenfeld, and Adam Morrison. Concurrent search tree by lazy splaying. 2011.

[12] Cecilia Aragon and Raimund Seidel. Randomized search trees. *Proceedings 30th Symp. Foundations of Computer Science*, page 540–545, 1989.

[13] E. Demaine, D. Harmon, J. Iacono, and Patrascu. Dynamic optimality—almost. *M. SIAM Journal on Computing*, (37:1):240–251, 2007.

[14] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and Rolf Fagerberg. An $\mathcal{O}(\log \log n)$-competitive binary search tree with optimal worst-case access times. *Proceedings of 12th Scandinavian Symposium and Workshops on Algorithm Theor*, pages 38–49, 2010.

[15] Hao Wang and Bill Lin. Pipelined van emde boas tree: Algorithms, analysis, and applications. *IEEE Infocom*, 2007.



Tesis publicada con autorización del autor
No olvide citar esta tesis