Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Conjuntos Indexados como Alternativa al Array Implementación y Evaluación

Autor: Carlos Gaspar Pozo Serrano
Tutor(a): Santiago Tapia Fernández

Madrid, Junio - 2023

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Conjuntos Indexados como Alternativa al Array Implementación y Evaluación

Junio - 2023

Autor: Carlos Gaspar Pozo Serrano
Tutor: Santiago Tapia Fernández
Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Aquí va el resumen del TFG. Extensión máxima 2 páginas.

Abstract

<<Abstract of the Final Degree Project. Maximum length: 2 pages.>>

End

Agradecimientos

Gracias

Índice general

1. Introducción

En el mundo de la programación, las estructuras de datos son elementos esenciales para la manipulación de información. Entre las estructuras de datos más utilizadas se encuentran las listas ordenadas, que permiten almacenar datos en un orden específico, lo que facilita la búsqueda y recuperación de información de manera eficiente y ordenada.

Una de las implementaciones de lista ordenada más utilizadas es la ofrecida por el estándar C++ a través de la clase std::set. En este trabajo se propone la implementación de una serie de interfaces de lista ordenada con el objetivo de plantear alternativas frente al std::set.

1.1. Motivación del proyecto

La necesidad de esta implementación surge de la demanda creciente de sistemas y aplicaciones que requieren la manipulación eficiente de grandes cantidades de información. En estos casos, es esencial contar con una estructura de datos que permita la recuperación rápida y precisa de los datos. Por lo tanto, a pesar de que la implementación std::set este optimizada, el desarrollo de una estructura alternativa más eficiente supondría una mejora en una gran cantidad de aplicaciones y sistemas. Además, aun en el caso de que se fracase en la implementación de una estructura más eficiente, la exploración de otras alternativas puede llevar a la concepción de mejoras a los algoritmos ya existentes.

1.2. Objetivos

El objetivo principal de este proyecto es la implementación de una interfaz de lista ordenada y compararla con el rendimiento de la implementación std::set del estándar de C++, el cual se puede dividir en los siguientes subobjetivos:

- Analizar las interfaces del std::set y el std::vector.
- Diseñar e implementar alternativas al array que no usen almacenamiento contiguo de todos sus elementos con interfaz tipo conjunto ordenado.
- Diseñar e implementar un conjunto de pruebas automáticas para evaluar la validez y rendimiento de las estructuras de datos diseñadas.
- Analizar el rendimiento de estructuras de datos alternativas a los arrays cuya interfaz se corresponde con un conjunto indexado y compararlos con otras implementaciones.

2. Entorno de desarrollo y especificaciones

Para la realización de este trabajo se ha decidido usar las siguientes herramientas:

- **Visual Studio Code:** se ha decidido usar este IDE debido a la gran gama de herramientas y extensiones de las que dispone para trabajar con distintos lenguajes y tipos de fichero.
- **CMake** [CMAa]**:** CMake es una herramienta de gestión de proyectos software que permite compilar código y generar tests independientemente del sistema operativo que se use. Además esta herramienta permite generar de forma sencilla binarios de un mismo fichero con distintos parámetros de compilación, lo cual facilita la realización de pruebas para binarios con distintos parámetros.
- **Python:** por último se ha decidido usar Pyhton para graficar los resultados obtenidos de las ejecuciones de las distintas implementaciones sobre los distintos casos de prueba. Se ha decidido usar este lenguaje por la facilidad que presenta a la hora de manejar ficheros csv, formato de salida de los casos de prueba, y filtrar según la implementación u otros criterios para poder visualizar los resultados.
- **Github:** para el control de versiones se ha decidido usar github debido al conocimiento que se tiene de esta herramienta, ya que es el sistema que se ha utilizado para la realización de otros proyectos.

2.1. Especificaciones del sistema

- Procesador: AMD Ryzen 5 5500U with Radeon Graphics 2.10 GHz
- RAM: 8 GB
- Sockets: 1
- Nucleos: 6
- Procesadores lógicos: 12
- Cache L1: 384 kB
- Cache L2 3 MB
- Cache L3: 8 MB
- Compilador: GNU 9.3.0 con opción -O3

3. Desarrollo de la propuesta

3.1. Implementaciones a comparar

3.1.1. std::set

La implementación del set en C++ es una variante de los Binary Search Trees llamada Red-Black Tree, esta es una estructura en árbol que se rige por las siguientes normas:

- Cada nodo tiene un color, negro o rojo.
- La raíz del árbol y todos los nodos hoja, se consideran de color negro.
- Un nodo rojo no puede tener nodos hijo de color rojo.
- Todos los caminos desde un nodo determinado hasta cualquier nodo hoja pasa a través del mismo número de nodos de color negro.

Además de seguir las restricciones que imponen los BST:

- Cada nodo tiene un máximo de 2 hijos.
- Los nodos a la izquierda de un determinado nodo siempre serán menores que el propio nodo.
- Los nodos a la derecha de un determinado nodo siempre serán mayores que el propio nodo.

Este conjunto de condiciones garantizan un árbol auto balanceado, lo cual resulta en los siguientes costes para cada operación:

- Insertar: $O(\log(n))$
- Borrar: $O(\log(n))$
- Búsqueda: $O(\log(n))$

3.1.2. HollowList y ptrHollowList

La HollowList es una estructura de datos que se basa en un std::vector que contiene otros std::vector. En el caso de HollowList, cada elemento del vector principal es un vector secundario que almacena los elementos de la lista. Por otro lado, en la variante ptrHollowList, en lugar de almacenar directamente los vectores secundarios, se utilizan punteros para hacer referencia a ellos.

3.1.2.1. Operaciones

Insertar

Al realizar una inserción sobre una HollowList se realizan dos búsquedas. Primero se realiza una búsqueda para determinar en qué vector se ha de insertar el nuevo elemento, esto se realiza a través de una búsqueda binaria sobre el primer elemento de cada vector. A continuación se realiza una segunda búsqueda para determinar la posición del elemento dentro del propio vector. Para esto se utiliza la función upper_bound, que devuelve la posición de un elemento a partir de un iterador de inicio, uno de final y el propio elemento, usando esa posición se inserta el elemento en el vector determinado. Por último se comprueba si el valor del vector supera un cierto valor límite, si esto sucede se extrae la mitad superior del vector y se inserta a continuación en la lista de vectores.

Teniendo esto en cuenta se puede calcular el coste de insertar un elemento de la siguiente forma, donde n es el número total de elementos y m es el tamaño de las sublistas:

$$\log_2\left(\frac{n}{m}\right) + \log_2(m) + m = \log_2\left(\frac{n}{m} \cdot m\right) + m \approx \log_2(n) \rightarrow O(\log(n))$$

Consulta del último elemento

Para consultar el primer valor de la lista se recupera el último valor del último vector de la lista.

Debido a que la operación de consulta accede de forma directa al último elemento de la lista se puede concluir que el coste de esta operación es $O(1)$.

Borrado del elemento mayor

Para eliminar el elemento con mayor valor de la lista se elimina el último elemento del último vector de la lista y, en el caso de que sea el último elemento restante de ese vector, este se elimina.

De forma análoga a la consulta del último elemento de la estructura, se puede concluir que el coste de esta operación es $O(1)$.

Operaciones adicionales

Aunque estas operaciones no se han implementado, por no ser necesarias o por no ser tan óptimas como las que ya se han implementado, se puede calcular el coste y explicar como se podrían implementar algunas operaciones adicionales.

Búsqueda

La operación de búsqueda realizaría la búsqueda del vector donde se debe insertar el elemento y lo busca en el vector.

Al igual que en la operación de insertar un nuevo elemento, se puede concluir que el coste de la búsqueda de un elemento en esta estructura es: $O(\log(n))$.

Borrado del primer elemento

Esta funcionalidad no se ha implementado ya que es posible realizar esta funcionalidad invirtiendo los criterios de ordenación del objeto que se desea ordenar. Además esta operación tiene un coste adicional en comparación con el borrado del último elemento.

La implementación de este método se realizaría borrando el primer elemento del primer vector de la lista. Esta operación tendría un sobrecoste, ya que al borrar el primer elemento el resto del vector tendría que adelantarse una posición en memoria.

El coste de esta operación se puede concluir que sería aproximadamente $O(1)$, con el matiz de que al borrar el primer elemento hay que recolocar el resto del vector con lo que el coste real es $O(m)$, donde m es el tamaño de las sublistas.

Consultar primer elemento

La consulta del primer elemento de la estructura, al contrario que el borrado no conlleva un sobrecoste con respecto a la consulta del último. Para realizar esta operación se consultar el primer elemento de la primera lista, lo que hace que la operación tenga un coste $O(1)$.

3.1.3. BinaryHeap

BinaryHeap es una implementación de un montículo sobre un array en lugar de sobre un árbol. Esto implica que en lugar a los nodos hijo o padre a través de los atributos del nodo actual, se accede aplicando un cálculo al índice actual. Dependiendo de a que nodo se desee acceder hay que realizar las siguientes operaciones:

- Nodo padre: $\frac{(indiceActual-1)}{2}$
- Nodo hijo izquierdo: $indiceActual \cdot 2 + 1$
- Nodo hijo derecho: $indiceActual \cdot 2 + 2$

De esta forma se puede acceder al nodo que se desee en tiempo constante. Esta forma de acceder puede presentar varias ventajas frente a la opción de la implementación en un árbol. La primera ventaja es que, debido a no tener que almacenar cada nodo la dirección de memoria de los nodos con los que esté emparentado, se reduce el consumo de memoria, variando desde 3·4 bytes en el caso de una máquina de 32 bits a 3·8 bytes en una máquina de 64 bits. La segunda ventaja viene por la adyacencia de los elementos en memoria. Al recuperar un elemento de memoria principal cuando se dé un fallo de caché el sistema operativo recuperará además regiones extra de memoria. Esto en el caso del vector significa recuperar más valores adyacentes al recuperado los cuales pueden necesitarse en el futuro y al haberse recuperado de está forma no se necesitará recuperarlo de memoria principal.

3.1.3.1. Operaciones

Insertar

Para insertar un nuevo elemento en la estructura se inserta el elemento al final de la lista y se compara con el valor que está almacenado en la posición del nodo padre. En caso de que sea menor se intercambian las posiciones y se repite la comparación con el elemento que esté almacenado en la posición del nuevo padre, esta operación se repite mientras se intercambien los elementos o hasta que el índice del nodo actual sea 0.

Esta operación se divide en dos fases, insertar el elemento y colocarlo en su posición. La primera operación tiene coste constante debido a que se inserta el elemento en la última posición de la lista. La segunda operación, dado que el máximo número de comparaciones que se van a hacer es $\log_2(n)$ donde n es el numero de elementos almacenados en la estructura, se puede concluir que tiene coste logarítmico, por tanto se puede concluir que el coste global de la operación es $O(\log(n))$.

Borrar el primer elemento de la estructura

El borrado del primer elemento se puede dividir en dos fases, la primera es el borrado del elemento y la segunda es el ajuste de la estructura para mantener la coherencia. Para la primera operación, en lugar de borrar el primer elemento de la estructura lo cual puede complicar el reajuste de la estructura, se intercambia el primer elemento con el último y a continuación se elimina el último elemento. Lo que se consigue con estas operaciones es conservar la estructura del array a la vez que se elimina el primer elemento de la lista. Una vez hecho esto se puede ordenar la estructura comparando el nodo raíz con sus dos hijos, y en el caso de que al menos uno de ellos sea menor que él se intercambiarán de posición con el menor de los dos y se repite la operación mientras uno de los dos hijos sea menor o los dos hijos sean nulos (sean indices mayores que el tamaño del vector).

Búsqueda

En el caso de esta estructura no se puede implementar la operación de búsqueda, esto se debe a que lo único que asegura la estructura en cuanto al orden de los elementos es que los hijos de un determinado nodo siempre tendrán un valor mayor o igual que el del propio nodo. Por lo tanto no hay forma de comprobar si un elemento está en la estructura que no pase por comprobar todos los elementos. Esto se puede optimizar ya que una vez encuentras un nodo con valor mayor al que se busca se puede descartar esa rama por completo, pero aun en ese caso sería una operación muy costosa y que en el peor caso tenga coste $O(\log(n))$.

3.1.4. Resultados esperados

Una vez exploradas las operaciones de insertado y borrado se puede teorizar que tanto en el caso de insertar elementos, como en el caso de buscar elementos, en las estructuras que lo permiten, los tiempos de ejecución deberían resultar similares, ya que estas operaciones tienen el mismo coste en todas las estructuras, $O(\log(n))$. Sin embargo el borrado de elementos debería mostrar tiempos de eje-

cución mucho menores en el caso de las estructuras HollowList, ya que el coste de la operación de borrado en esta estructura es constante, mientras que en el resto de estructuras es logarítmico. Por otra parte, en cuanto al uso de la memoria la estructura std::set debería ser la que más consumo de memoria tenga, ya que por cada nodo necesita 3 punteros adicionales. El segundo lugar sería esperable que lo ocupase la estructura HollowList, debido a que esta estructura usa una cantidad de std::vector que crece de forma linear, cada uno con tres punteros, uno al inicio del vector, otro al final y otro al final de la memoria reservada. Por último, la estructura con un menor uso de memoria se espera sea el BinaryHeap, ya que la memoria que usa es la de un único std::vector más la de los elementos que almacena por lo tanto no tiene un coste adicional por cada elemento insertado.

3.2. Tests realizados

3.2.1. Estructura de los tests realizados

Los tests que se han realizado sobre estas estructuras han consistido en insertar elementos, desde 500.000 elementos hasta 4.000.000, para medir los tiempos al insertar. A continuación se realiza la eliminación e insertado de 1.000.000 de elementos para probar el rendimiento de cada estructura con una cantidad de elementos considerable. Por último se borran todos los elementos que sigan almacenados para medir el rendimiento en el borrado. Además se han realizado estos tests sobre cada estructura con varios objetos para almacenar en la estructura y con varios casos de entrada de los datos en la estructura. Los objetos a insertar son los siguientes:

- Un objeto que contiene un double, por lo que ocupa 8 bytes,
- Un objeto con un double y un entero, por lo que ocupa 16 bytes
- Y por último una estructura con un double, un entero y un array de chars que contiene los valores «puntuacion:» concatenado con el valor del double, un guion y el valor del entero, ocupando un total de 48 bytes.

Los casos de prueba que se usan en los tests son los siguientes:

- Datos ordenados en orden,
- Datos ordenados en orden inverso
- Y datos ordenados aleatoriamente

Ademas, de cada uno de los tests realizados sobre cada una de las estructuras se extraen los siguientes datos:

- Tiempo de ejecución en insertado, uso y borrado
- Tiempo total
- Uso de memoria
- Fallos de página

3.2.2. Generación de los tests

La generación de los tests se puede dividir en dos partes, la compilación de los binarios de los distintos tests y la creación de los mismos.

En el código utilizado para la generación de los binarios (Algoritmo 3.1) se puede observar que se ha utilizado una función para generalizar la compilación y posteriormente se define un bucle para iterar sobre los distintos objetos y definir las compilaciones para todas las estructuras. En el ejemplo de función se define primero el ejecutable junto con los distintos ficheros necesarios para generar el binario y posteriormente se añaden las definiciones necesarias para cada test. En el caso de ejemplo del BinaryHeap solo es necesaria la definición del objeto que se va a utilizar en las pruebas. En otros casos como la estructura HollowList también es necesario definir cuál de las dos estructuras usar, si ptrHollowList o HollowList, ya que en ambos casos se usa el mismo código para realizar las pruebas, lo único que varía es la estructura que se usa para ejecutarlas.

Algoritmo 3.1 Compilación de los tests

```
function(addBinaryHeapExecutable type)
    add_executable(TestBinaryHeap${type} ./Prototipo/TestBinaryHeap.cpp
     ./InsercionEnListas/puntuacionLarge.hpp
      ./InsercionEnListas/puntuacionMedium.hpp
      InsercionEnListas/puntuacionSmall.hpp)
    target_compile_definitions(TestBinaryHeap${type} PUBLIC testType=${type})
endfunction(addBinaryHeapExecutable type)

# Distintos tipos de listas ordenadas a probar
set(types
    0
    1
    2)

foreach(type ${types})
    addHollowListExecutable(${type})
    addptrHollowListExecutable(${type})
    addTestVectorExecutable(${type})
    addTestSetExecutable(${type})
    addBinaryHeapExecutable(${type})
endforeach()
```

Además de la compilación de los tests, también se han de compilar los ejecutables que serán la entrada de las pruebas (Algoritmo 3.2), en este caso son generadores de números que se usarán para construir los objetos que se van a insertar en las estructuras.

Algoritmo 3.2 Compilación de los generadores

```
1 # Definiciones de compilacion
2 add_executable(GenRandom ./genNumbers/genNumbers.cpp)
3 add_executable(GenOrdered ./genNumbers/genNumbers.cpp)
4 add_executable(GenInverse ./genNumbers/genNumbers.cpp)
5
6 target_compile_definitions(GenRandom PUBLIC GNType=0)
7 target_compile_definitions(GenOrdered PUBLIC GNType=1)
8 target_compile_definitions(GenInverse PUBLIC GNType=2)
```

Una vez se ha definido la compilación de los binarios se pueden definir las pruebas que se van a realizar Algoritmo 3.3. Para ello se utilizan dos funciones, una para definir las pruebas de las estructuras HollowList, ya que necesitan un parámetro extra para definir el tamaño de las sublistas, y otra para definir el resto de casos de prueba. En ambas se realiza la misma operación, se usa un add_custom_command[CMAb]en el que se definen dos commands, uno para crear un fichero y escribir en él la entrada de la prueba, lineas 8 y 22, y un segundo en el que se define la ejecución de la prueba, lineas 9-12 y 23-26.

La definición de la ejecuciones divide en dos partes, la primera es la definición de los 4 parámetros del mandato /usr/bin/time, el primero es definir la salida de este mandato, el segundo es la opción -a, que se usa para indicar al mandato que concatene los resultados al final del fichero en lugar de borrar el contenido y escribir los resultados, el tercer parámetro indica el formato de la salida, que indica los siguientes campos:

- Estructura utilizada
- Caso de prueba
- Objeto utilizado
- Numero de elementos
- Tamaño máximo del conjunto de páginas residentes del proceso durante su tiempo de vida, en kilobytes.
- Número total de segundos de CPU que el proceso pasó en modo kernel.
- Número total de segundos de CPU que el proceso pasó en modo usuario.
- Número de fallos de página menores, o recuperables.

Y el ultimo parámetro indica la ejecución sobre la cual se van a recopilar estos datos, el test que se va a ejecutar con parámetro el numero de elementos a insertar en la prueba, entrada el fichero creado anteriormente y salida la la variable output.

Una vez definida esta función se puede utilizar en el bucle donde se crean todas las pruebas, el cuál itera desde el valor 500.000 hasta el 4.000.000 sumando 500.000 en cada iteración. Dentro de este bucle se itera sobre los distintos

generadores de casos de prueba y sobre los distintos objetos, se define como variable output un fichero por cada uno de los generadores y se define los casos de prueba llamando a la función ya mencionada con cada una de las estructuras.

Algoritmo 3.3 Generación de las pruebas

```
function(defineCaseHollowList program gen i type)
    MATH(EXPR j 2*${i}+2000000)
    set(size 128)
    MATH(EXPR cntr 512)
    while(size LESS_EQUAL cntr)
        add_custom_command(
            TARGET ${program}${type} POST_BUILD
            COMMAND ${gen} ${j} >tmp
            COMMAND /usr/bin/time -o ${CMAKE_SOURCE_DIR}/genGraphs/CSV/sys.csv
                -a
                -f "${program}-${size},${gen},${type},${i},%M,%S,%e,%R"
                ${CMAKE_BINARY_DIR}/${program}${type} ${i} ${size}  < tmp >> ${
                    output}
            WORKING_DIRECTORY ${CMAKE_BINARY_DIR})
        MATH(EXPR size 2*${size})
    endwhile()
endfunction(defineCaseHollowList program gen i)

function(defineCase program gen i type)
    MATH(EXPR j 2*${i}+2000000)
    add_custom_command(
        TARGET ${program}${type} POST_BUILD
        COMMAND ${gen} ${j} >tmp
        COMMAND /usr/bin/time -o ${CMAKE_SOURCE_DIR}/genGraphs/CSV/sys.csv
        -a
        -f "${program},${gen},${type},${i},%M,%S,%e,%R"
        ${CMAKE_BINARY_DIR}/${program}${type} ${i}  < tmp >> ${output}
        WORKING_DIRECTORY ${CMAKE_BINARY_DIR})
endfunction(defineCase program gen i)

set(i 500000)
MATH(EXPR cntr 4000000)
set(repeat 2)
while(i LESS_EQUAL cntr)
    foreach(gen ${gens})
        foreach(type ${types})
            set(output ${CMAKE_SOURCE_DIR}/genGraphs/CSV/${gen}.csv)
            defineCase(TestSet ${gen} ${i} ${type})
            foreach(rep RANGE ${repeat})
                defineCaseHollowList(ptrHollowList ${gen} ${i} ${type})
                defineCaseHollowList(HollowList ${gen} ${i} ${type})
                defineCase(TestBinaryHeap ${gen} ${i} ${type})
            endforeach()
        endforeach()
```

4. Resultados

Para la visualizaci los resultados se han utilizado las librer pandas y matplotlib de python. La primera librere emplea para leer los ficheros csv donde se han escrito los resultados y poder filtrarlos del modo deseado, la segunda librere emplea para obtener las graficas.

4.1. Exposicion de resultados

4.2. Analisis

5. Analisis Impacto / Conclusiones / Path fordward

5.1. Overview

5.2. The next section

A. Title of the first appendix chapter

A.1. Overview

bla bla bla bla bla bla bla bla bla bla bla bla bla bla b la bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla roughness parameter R_a bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla, see [**?**].

A.2. The next section

Bibliografía

[CMAa] Cmake reference documentation. https://cmake.org/cmake/help/latest/.

[CMAb] Cmake reference documentation,cmake-commands(7),add$_{c}ustom_{c}ommand.https : //cmake.org/cmake/help/latest/command/add$_{c}ustom_{c}ommand.html?highlight = addcommand : add$_{c}ustom_{c}ommand.$

Nomenclatura

R_a	arithmetic average roughness