Facultad de Ingeniería - UDELAR

Herramientas de Programación para Procesamiento de Señales

Implementación de un heap binario

Estudiante	Jimena Arruti - 5127857-5
Profesor	Juan Cardelino

Montevideo, 31 de octubre de 2016

1. Introducción

En el presente informe se presentan los análisis realizados en el marco de la implementación de un heap binario mínimo. Se siguió la guía de la práctica a la hora de implementar las funciones requeridas, las cuales fueron programadas utilizando Gedit. Se debuggeó utilizando el programa ddd y se chequeo el uso de memoria mediante valgrind. En el texto se detalla el análisis de performance realizado teóricamente y comprobado con la herramienta Kcachegrind, en el cual se compara lo obtenido para el heap contra el desempeño de un arreglo simple.

2. Herramientas Utilizadas

- Gedit (editor de texto)
- GDB (debugger) en entorno ddd
- Valgrind (profiler)
- Aplicación Kcachegrind
- Función time

3. Análisis de performance

3.1. Comparación de desempeño de add_item y extractMin contra estimaciones teóricas

En esta sección se busca comparar el rendimiento las funciones desarrolladas contra sus rendimientos teóricos. Tanto para add_item como para extractMin se prevé un costo teórico de $O(\log(N))$.

Se intenta relevar el costo computacional de las funciones add_item y extractMin del heap. Para esto, se ejecuta el programa para diferentes tamaños de heap, agregando o quitando un item según la función que se esté relevando, y utilizando siempre el peor caso posible. Se leen los resultados mediante el uso de *Kcachegrind*, cuyos resultados se muestran en el cuadro 1. Es interesante observar que el programa colapsa por falta de memoria cuando se intenta utilizar un heap de 10000000 elementos, si se está reservando memoria de manera estática para los arreglos, excediendo el tamaño del stack; esto es corregido reservando memoria mediante el uso de la función malloc.

Observando el cuadro 1 y la figura 1, se puede concluir que efectivamente se tienen costos de relación logarítmica con la cantidad de nodos.

Nodos del heap	Costo (ciclos) add_item	Costo (ciclos) extractMin
100	1097	1029
1000	1689	1588
10000	2462	2196
100000	3047	2652

Cuadro 1: Relevamiento de costo computacional para add_item y extractMin

A partir del relevamiento descrito en el cuadro 1, se grafican los puntos obtenidos y se ajusta cada costo por una función logarítmica, teniendo como resultado la figura 1, en la cual se observa que el costo teórico puede considerarse acertado.

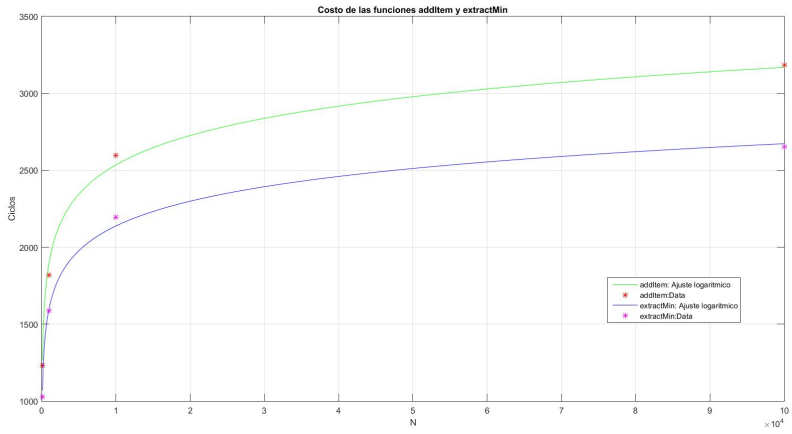

Figura 1: Resultados obtenidos al analizar la performance de add_item y extractMin, operando con un solo item, considerando el peor caso, y variando el tamaño del heap

3.2. Comparación de extracción y búsqueda del mínimo utilizando heap o array

Para esta sección se parte de arreglos ordenados de manera descendiente, siendo éste el peor caso para todas las operaciones. Primero se compara la búsqueda del mínimo utilizando tanto el heap como un arreglo. Los resultados experimentales se muestran en la tabla 2, en los cuales se evidencia que el costo de búsqueda en el heap es constante al variar la cantidad de datos, mientras que utilizando un array éste costo es lineal. Para el caso de la extracción, la operación con el array sigue siendo la misma, mientras que en el caso del heap se suman los costos (cada vez mayores) del reordenamiento del árbol. Los datos pueden verse en el cuadro 3.

Se puede concluir que el heap es mucho más eficiente que un array a la hora de buscar el mínimo de un conjunto de datos.

Nodos del heap	Búsqueda de mínimo/heap	Búsqueda de mínimo/array
100	7	1701
1000	7	17007
10000	7	170007
100000	7	1700001

Cuadro 2: Relevamiento de costo computacional para la búsqueda del mínimo. Se utiliza un heap y un array.

Nodos del heap	Extracción de mínimo/heap	Extracción de mínimo/array
100	1029	1709
1000	1588	17009
10000	2196	170009
100000	2652	1700009

Cuadro 3: Relevamiento de costo computacional para la extracción del mínimo. Se utiliza un heap y un array.

3.3. Comparación de construcción lineal de un heap contra construcción utilizando add_item y construcción de un array lineal

Finalmente, interesa contrastar el desempeño al crear el heap utilizando la función buildheap contra la creación del mismo utilizando sucesivas llamadas a add_item. En el cuadro 4 se muestran los costos obtenidos para los dos casos no considerando el costo de la inicialización (7200 ciclos), y tomando el peor caso (arreglo de entradas en orden descendiente al comienzo). La evidencia se condice con los resultados teóricos, mostrando la construcción

mediante buildheap un costo lineal con la cantidad de nodos, y la construcción mediante add_item una dependencia de la forma $\log(N!)$. Como puede verse en el gráfico 2, la construcción del heap para tamaños considerables de datos es mucho más eficiente al utilizar la construcción lineal.

Nodos del heap	buildheap	Llenar heap	add_item
100	19527	3310	100701
1000	191605	33010	1571266
10000	1918919	330010	22788846
100000	19189609	3300010	287745271

Cuadro 4: Relevamiento de costo computacional para la creación del heap mediante la función buildheap y mediante sucesivos usos de add_item. Para el costo de la función buildheap además se debe considerar el costo del llenado del heap sin ordenar. Peor caso.

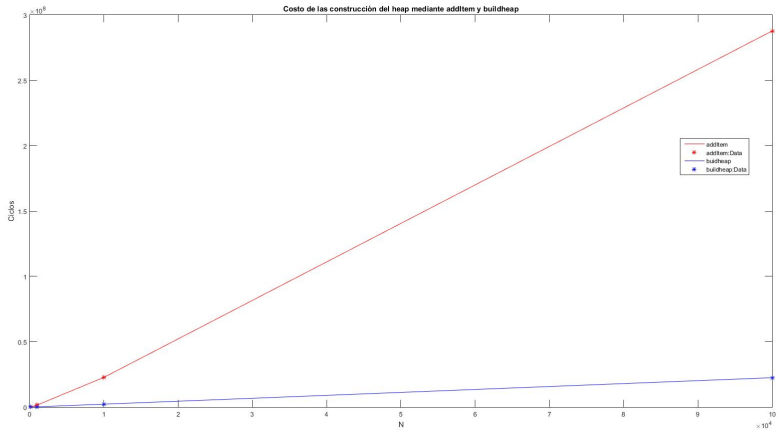

Figura 2: Resultados obtenidos al analizar la construcción del heap mediante add_item y buidheap, considerando el peor caso, y variando el tamaño del heap

4. Conclusiones

Se concluye que los costos teóricos son comprobados con suficiente exactitud mediante los datos obtenidos experimentalmente. Además, se observa cómo el almacenamiento en un heap binario es muy eficiente para la búsqueda de mínimos, comparado con la búsqueda en un arreglo lineal.