Montículo de Fibonacci

Casos de prueba y mediciones de tiempos

Índice

Recursos utilizados

Detalles de implementación

El montículo de Fibonacci, cuyo código se divide en “montFib.h” y “montFib.cpp”, está implementado de forma genérica para un tipo y criterio de comparación dados. No obstante, para las pruebas posteriores lo trataremos como un montículo de mínimos para números enteros. De este modo, la operación ‘Top’ equivaldrá a ‘Min’, y ‘borraTop’ a ‘borraMin’.

Por otro lado, además de las operaciones requeridas para la práctica y funciones auxiliares de éstas, se han añadido dos operaciones principales (públicas): ‘clear’, libera toda la memoria dinámica utilizada para almacenar los nodos, dejando como resultado el montículo vacío; y ‘show’, que devuelve un string con la representación del estado actual del montículo.

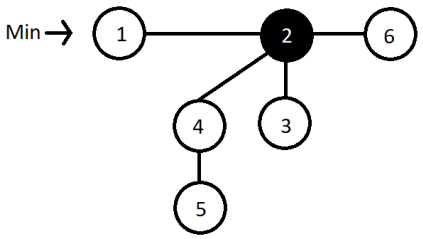
En el archivo “main.cpp” se encuentra la implementación de la interfaz y las funciones encargadas de generar los casos más grandes.

Representación del montículo

Para visualizar el montículo en un momento dado, se representa como una lista de nodos (raíces) encabezada por el mínimo, indicando entre paréntesis la clave, marca (*) y lista de hijos (en caso de tenerlos), representados recursivamente del mismo modo.

Ej: { -> (1) (6) (2* [(3) (4 [(5)])]) } =

(El orden dentro de una lista no es relevante)



Interfaz

Para realizar más fácilmente los casos de prueba, disponemos de una interfaz sencilla que da acceso a dos montículos, cada uno con todas las operaciones disponibles, de modo que se puedan manipular, unir y medir tiempos introduciendo el número de cada opción.

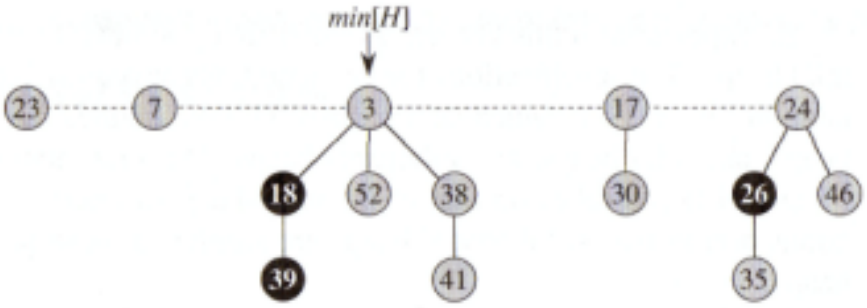
```
        MONTÍCULO DE FIBONACCI

Elija un montículo a utilizar (ambos son montículos de mínimos para números enteros):
        1.- Montículo_1
        2.- Montículo_2
        0.- Salir
```

```
Operaciones:
        1.- Mostrar top
        2.- Insertar
        3.- Unión con el otro montículo
        4.- Borrar top
        5.- Decrecer clave
        6.- Mostrar montículo
        7.- Tomar medidas de tiempos
        8.- Guardar estado
        0.- Volver
Elija una opción:
```

```
Elija una opción:7
Operaciones:
        1.- Top
        2.- Inserción
        3.- Unión
        4.- BorraTop
        5.- DecreceClave
        0.- Volver
Elija una operación a medir:
```

Caso de prueba sencillo

Para comprobar el correcto funcionamiento de todas las operaciones, podemos intentar construir con ellas un montículo conocido. Un buen ejemplo puede ser el utilizado en las diapositivas de la asignatura, que inicialmente tiene esta forma:



En el archivo “ejemploUso.txt” está la secuencia de números que, introduciéndose en la línea de comandos de la interfaz (basta con copiar y pegar), genera el montículo anterior y guarda varios estados intermedios en “estados.txt”.

El procedimiento en este caso se resume en generar dos árboles de grado 3 con las claves deseadas (mediante consecutivos insert y borraTop forzamos el orden en que los nodos se combinan), cada uno en un montículo, para posteriormente decrecer las claves necesarias para marcar los nodos indicados y aislar los árboles pequeños, eliminar nodos sobrantes y finalmente unir ambos montículos.

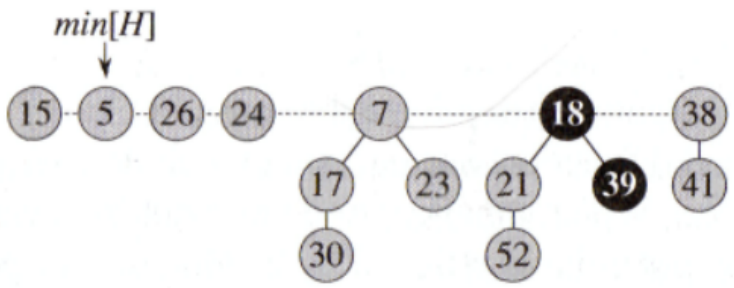
Resultado:

{-> (3 [(52) (38 [(41)]) (18* [(39*)])]) (24 [(46) (26* [(35)])]) (17 [(30)]) (23) (7) }

Como última comprobación, se puede comparar el resultado de aplicar sobre este montículo las mismas operaciones que en los apuntes: insertar 21, borrar min, y decrecer claves 46 y 35.
Secuencia, tras la anterior: 2 21 8 4 8 5 46 15 8 5 35 5 8

Resultado:

{-> (5) (24) (26) (15) (38 [(41)]) (18* [(39*) (21 [(52)])]) (7 [(23) (17 [(30)])]) }



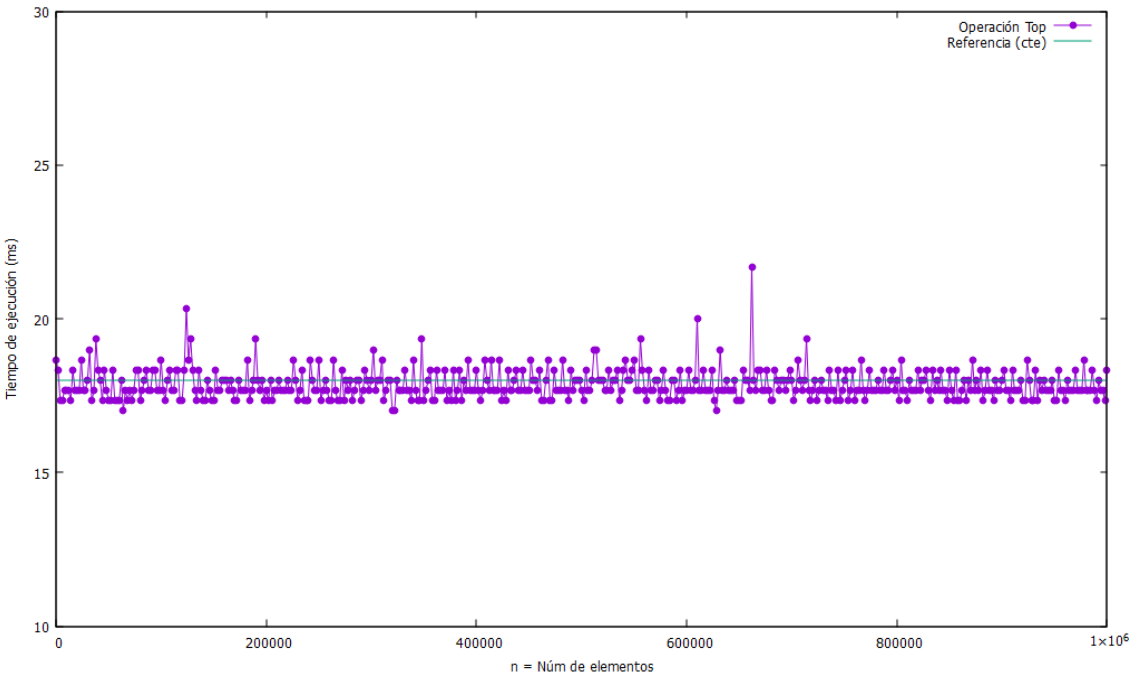
Mediciones y gráficas de tiempos de ejecución

A pesar de la recomendación de utilizar el reloj del sistema, las siguientes medidas han sido tomadas mediante la función clock() de la librería “ctime”, que cuenta con una precisión de 1 ms (he comprobado en distintas pruebas que es su precisión real, con un margen de error de 1 o 2 ms) y sólo cuenta el tiempo de CPU que ocupa el proceso del programa, reduciendo así el impacto de otros procesos en la medida final.

Al pie de cada imagen se encuentra el comando de gnuplot correspondiente para replicar el resultado.

Top

Para medir la operación Top, se realizan 1M de llamadas por cada 1000 elementos insertados, llegando hasta el millón de elementos (1000 muestras).

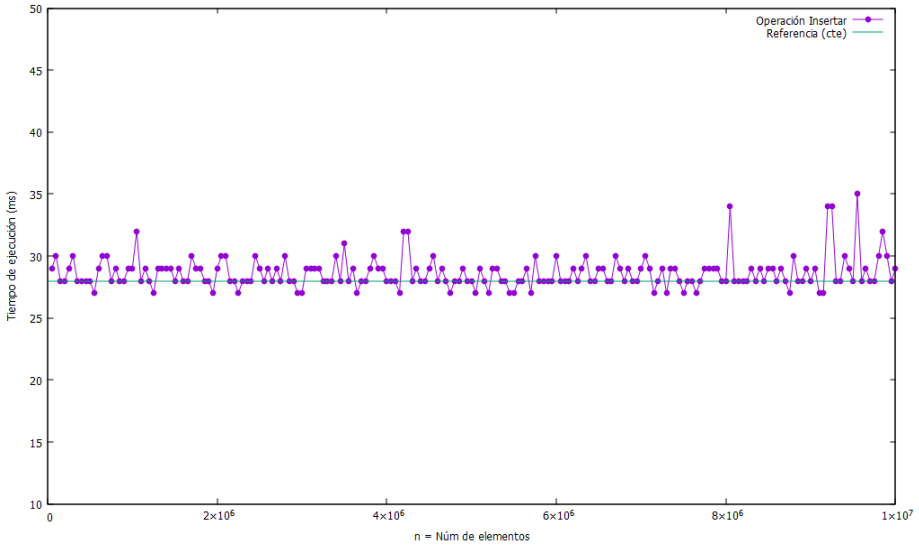


set yrange [10:30]; set xrange [0:1000000]; plot "top.dat" title "Operación Top" lt 7 lc 1 with linespoints, 18 title "Referencia (cte)" lt 1 lc 2

Como era de esperar por la trivialidad de la operación, el tiempo de ejecución se mantiene constante.

Insert

Cada intervalo de tiempo medido en este caso abarca 100 mil inserciones, repitiéndolo 200 veces hasta los 20M de elementos.

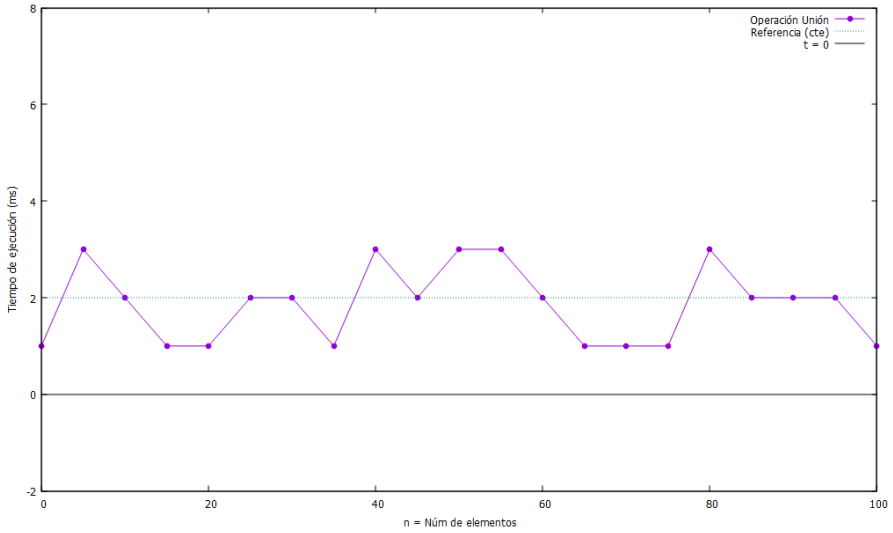


set yrange [10:50]; set xrange [0:10000000]; plot "insert.dat" title "Operación Insertar" lt 7 lc 1 with linespoints, 28 title "Referencia (cte)" lt 1 lc 2

Al igual que la anterior, esta operación mantiene un coste constante. Dado que la forma del montículo no influye en el resultado, no es necesario agrupar nodos en el proceso.

Unión

Debido a la necesidad de un gran número de ejecuciones consecutivas (y cada vez con más elementos) para asegurar la fiabilidad de la muestra, se han requerido de 5000 montículos auxiliares (x3 para hacer la media), lo que ha limitado el número de muestras y elementos por cuestiones de memoria.

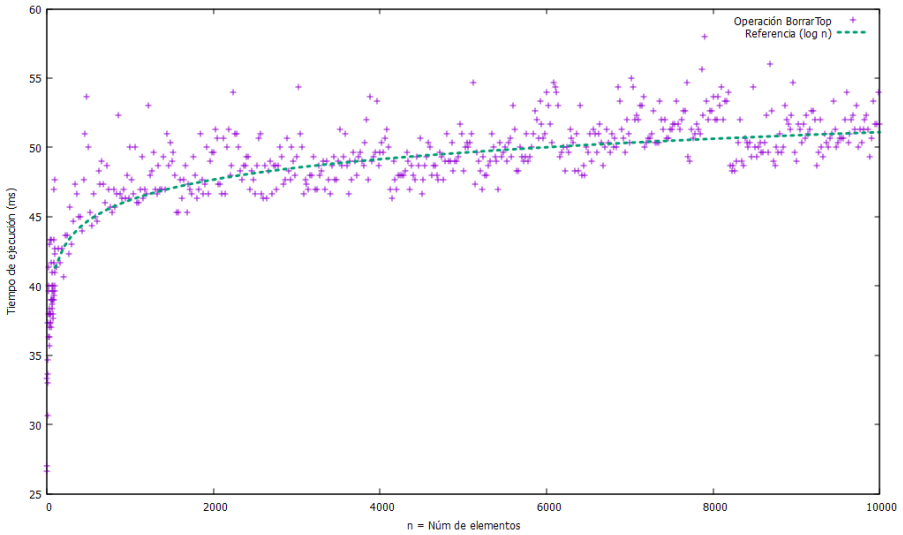


set yrange [-2:8]; set xrange [0:100]; plot "union.dat" title "Operación Unión" lt 7 lc 1 with linespoints, 2 title "Referencia (cte)" lt 0 lc 2, 0 lc -1 title "t = 0"

Igualmente se aprecia el coste constante de la operación, puesto que la concatenación de dos listas de raíces cualesquiera se reduce a la reasignación de 4 punteros.

BorraTop

Cada muestra de esta operación se toma repitiendo 10 mil veces inserción-borrado (el elemento insertado debe ser mayor que el mínimo, para forzar un mayor tiempo de consolidar), contando con que el coste constante de insertar no afectará a la gráfica. Como es la primera operación con coste amortizado, es interesante tomar más muestras.

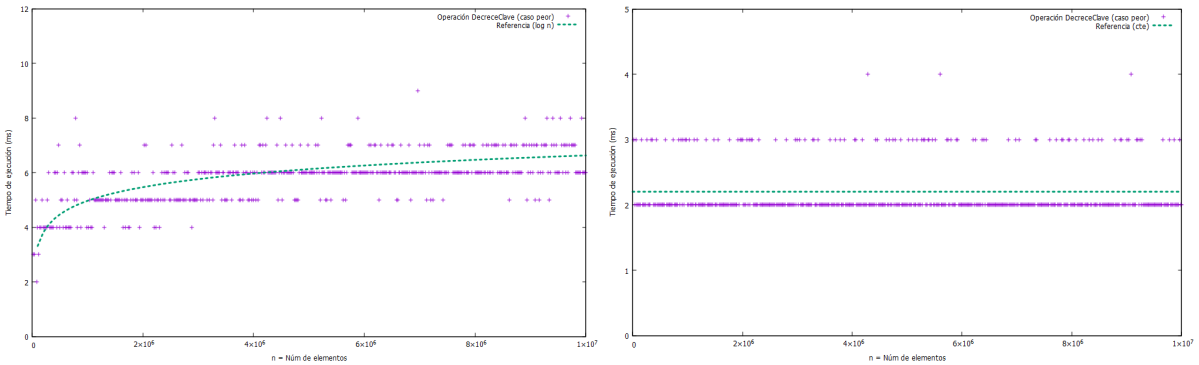


set yrange [25:60]; set xrange [0:10000]; plot "borraTop.dat" title "Operación BorrarTop" lt 1 lc 1 with points, (log(x)/log(1.6)+31.5) title "Referencia (log n)" lt 0 lc 2 lw 3

A pesar de que la operación en el caso peor tiene coste real lineal en el número de elementos, al estar manipulando constantemente el mismo montículo (de modo que muchos nodos están ya combinados en árboles), el tiempo de consolidar, y por tanto de borrar, muestra un ascenso logarítmico.

DecreceClave

Para esta medición se llama a decreceClave 10 mil veces consecutivas por cada 20 mil elementos insertados. En este caso, a diferencia de los primeros, es importante para maximizar el tiempo llamar a borraTop una vez, de modo que al consolidar existan ramas mayores y se produzcan más cortes. Es importante, a su vez, decrecer claves altas, que se encuentren más abajo en los árboles, y darles valores bajos que provoquen su corte.
En este caso podemos ver la comparación entre el coste real de la operación (se pueden generar esos datos descomentando una línea del código) y el coste amortizado.



set yrange [0:15]; set xrange [0:10000000]; plot "decreceClave.dat" title "Operación DecreceClave (caso peor)" lt 1 lc 1 with points, (log(x)/log(4) - 5) title "Referencia (log n)" lt 0 lc 2 lw 3
set yrange [0:5]; set xrange [0:10000000]; plot "decreceClave.dat" title "Operación DecreceClave (caso peor)" lt 1 lc 1 with points, (2.2) title "Referencia (cte)" lt 0 lc 2 lw 3

A diferencia de la anterior operación, el coste de ésta se amortiza con las propagaciones de cortes de llamadas anteriores, logrando pasar de un coste logarítmico a uno constante.