Universidad de Chile
Departamento de Ciencias de la Computación
CC4102 - Diseño y Análisis de Algoritmos

Auxiliar 8 - "Hollow Heaps y Ordenando Strings"

Profesor: Gonzalo Navarro
Auxiliar: ~~Manuel~~ Ariel Cáceres Reyes

14 de Mayo del 2018

P1. Hollow Heaps

Los *hollow heaps* implementan las operaciones vistas para colas de prioridad, además de la operación *decrease-key*, que cambia la clave de un elemento a un valor menor.
Para realizarlo, guarda los elementos en un conjunto de árboles que cumplen la *propiedad de heap* (la llave del padre es menor a la de todos sus hijos) y que pueden tener nodos que solo tienen claves, pero no elementos, los *hollow nodes*.
La operación básica para unir dos árboles con raíces u y v se denomina $link(u, v)$ y, suponiendo que u tiene una clave menor a v, cuelga a v como hijo de u y aumenta el $rank$ de u en 1 (el $rank$ es un atributo de los nodos que se inicia en 0 y cambia con el procedimiento antes descrito).

- Las operaciones de *insertar* y *meld* se realizan de manera análoga a los *Heaps de Fibonacci*.
- *extraer_minimo* saca el elemento del nodo que tiene el mínimo (al igual que en los *Heaps de Fibonacci* este puntero se recomputa con las operaciones), luego elimina todas las raíces que son *hollow nodes* y finalmente hace *links* entre nodos de mismo $rank$, mientras sea posible.
- *decrase-key*, saca el elemento del nodo correspondiente (creando un *hollow node*) y lo inserta con su nueva llave. Además, si el nodo en el que estaba tenía $rank$ r, se mueven sus $r-2$ hijos (con sus subárboles) de menor $rank$, al nuevo nodo insertado (dándole un $rank$ de $r-2$ si es no negativo).

Muestre que un conjunto de operaciones válidas sobre un *hollow heap* inicialmente vacío alcanza $\mathcal{O}(1)$ amortizado para todas las operaciones, excepto *extraer_min* para la cual tiene un costo amortizado de $\mathcal{O}(\log N)$

P2. Ordenando Strings

Tenemos los `Strings` $w_1, w_2, \ldots w_n \in \Sigma$, con $\sigma = |\Sigma| \in \mathcal{O}(n)$ y queremos ordenarlos alfabéticamente. Además, el largo total de los `Strings` es $N = \sum |w_i|$.
Diseñe algoritmos de tiempo $\mathcal{O}(N)$ en los casos:

a) Todas las cadenas del mismo largo.

b) Cadenas de largo variable.

"The world is continuous but the mind is discrete"
David Mumford

Soluciones

P1. Hollow Heaps

Veamos primero que todas las operaciones, excepto *extraer_minimo* tienen costo $\mathcal{O}(1)$ en el peor caso.

Veamos que se cumple que los nodos con item de *rank* r tienen r hijos con *ranks* $0, 1, \ldots, r-1$, por otro lado, los *hollow nodes* de *rank* r tienen 2 hijos de *ranks* $r-1$ y $r-2$.

Veamos ahora que un nodo de *rank* r tiene al menos $F_{r+3} - 1$ descendientes (donde F_n es el n-ésimo número de fibonacci). Esto es simple de ver por inducción en r pues en los casos $r = 0 \vee 1$ se cumple y en el caso $r \geq 2$ el nodo tiene al menos 2 hijos que cumplen la hipótesis inductiva, luego tienen al menos $1 + (F_{r+2} - 1) + (F_{r+1} - 1) \geq F_{r+3} - 1 \geq F_{r+2} \geq \phi^r$. Y por lo tanto, el *rank* de un nodo en un *hollow heap* de N nodos es a lo más $\log_\phi N$.

Definimos ahora el potencial de un nodo v como 0 si es *hollow node* o si es un nodo con item hijo de un nodo con item y 1 en otro caso, además definimos el potencial de la estructura como la suma de los potenciales de sus nodos. Podemos comprobar que en las operaciones distintas a *extraer_minimo* los aumentos de potencial son constantes por lo que el costo amortizado de estas operaciones sigue siendo constante. Por otro lado, para *extraer_minimo*, la eliminación del elemento de la raíz produce un aumento de potencial de a lo más su *rank* (todos sus hijos tenían un item) que es $\mathcal{O}(\log_\phi n)$, luego, la destrucción de los *hollow nodes* se las cobraremos a su correspondiente creación (inserción o decreace-key), finalmente, veamos que cada *link* que se realice tiene costo amortizado 0, pues genera un decremento de potencial de 1. Así el costo amortizado de *extraer_minimo* es $\mathcal{O}(\log n)$.

P2. Ordenando Strings

a) Si adaptamos CountingSort para ordenar `Strings` de una letra su complejidad será $\mathcal{O}(n + \sigma)$, y por lo tanto hacer un RadixSort sobre estos *Strings* tiene costo $\mathcal{O}\left(\frac{N}{n}(n+\sigma)\right) = \mathcal{O}(N)$.

b) Si los `Strings` tienen diferente largo no podemos usar el RadixSort de la parte anterior pues este ordena desde el dígito menos significativo, a si es que por ejemplo *ba* quedaría antes de *a* al ordenar ascendentemente.

Otra opción es usar `padding` en las cadenas de largo menor a la más larga, de modo que tengamos n cadenas de largo $\max |w_i|$. Sin embargo, este procedimiento podría llegar a ser $\mathcal{O}(N^2)$ (si por ejemplo tenemos 1 cadena larga y las demás cortas). 😭

"The world is continuous but the mind is discrete"
David Mumford

La solución viene dada por procesar los `Strings` desde el dígito más significativo y hacer $\leq \sigma$ llamados recursivos para ordenar por el siguiente dígito. Este algoritmo es llamado MSD-RadixSort y se presenta a continuación:

```
//W es el arreglo de Strings y d el dígito por el que se está ordenando
//Asumimos también que CountingSort además retorna el arreglo que contiene las
  posiciones iniciales de cada letra
Funcion sort (W, d)
    if |W| ≤ 1 then
        return
    end
    i ← 0
    count ← CountingSort(W, d)
    while W[i + 1][d] = −1 do
        i ← i + 1
    end
    for r ← 1 . . . σ do
        sort(W[i+count[r]: count[r+1]-1], d+1)
    end
```

Por razones prácticas de implementación le agregaremos un -1 al final de las cadenas, lo que agrega solo n al tamaño del input.
Finalmente considerando que el costo amortizado por letra para CountingSort es de $\mathcal{O}(1)$ y notando que en el `peor` caso MSD-RadixSort procesa todas las letras del input con CountingSort, el costo total es $\mathcal{O}(N)$.