Tema: Montículos sesgados

Asignatura: Métodos Algorítmicos de Resolución de Problemas. **Autor:** Ricardo Peña

Curso 2014/15

1. Introducción

⋆ En 1972, C.A. Crane inventó los llamados **montículos zurdos** (*leftist heaps*), que fueron los primeros en ofrecer la operación *union* con un coste en tiempo $O(\log n)$ en el caso peor. Además, están implementados como árboles binarios. Por tratarse de una estructura dinámica, no están sujetos a la limitación de tamaño de los montículos de Williams, que están implementados sobre un vector. Para una explicación detallada de este tipo de montículos, consúltese [1, Sec. 7.5.4].

⋆ En 1986, D.D. Sleator y R.F. Tarjan inventaron los **montículos sesgados** (*skew heaps*) bajo el nombre *self adjusting heaps*. También están implementados como árboles binarios y ofrecen la operación *unión* con un coste $O(\log n)$, pero ahora se trata de un **coste amortizado**. En el caso peor, el coste de *unión* está en $O(n)$.

⋆ ¿Por qué es importante disponer de la operación *union*? Tres razones:

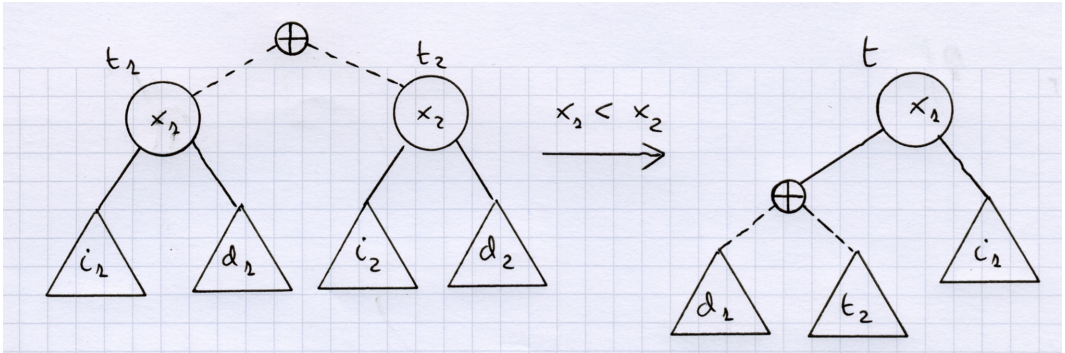
1. Útil en sí misma. Por ejemplo, en un sistema de reparto de carga, al caer un procesador, el otro debe hacerse cargo de las dos colas de procesos.
2. La inserción puede implementarse como un caso particular de la unión.
3. El borrado también: se suprime la raíz y se unen los dos montículos hijos.

2. Montículos sesgados y operación de unión

⋆ *Definición:* Un montículo sesgado es un arbol binario que satisface la propiedad de montículo: o bien es vacío, o bien en la raíz se encuentra el elemento mínimo, y adicionalmente sus dos hijos son montículos sesgados.

⋆ Cuando uno o ambos montículos son vacíos, la unión de dos montículos es trivial. La **unión** de dos montículos sesgados no vacíos consiste en comparar las dos raíces, escoger la menor, intercambiar sus dos hijos, y sustituir su hijo izquierdo, es decir su antiguo hijo derecho, por la unión recursiva de dicho hijo derecho, con todo el montículo de mayor raíz.

⋆ Gráficamente:



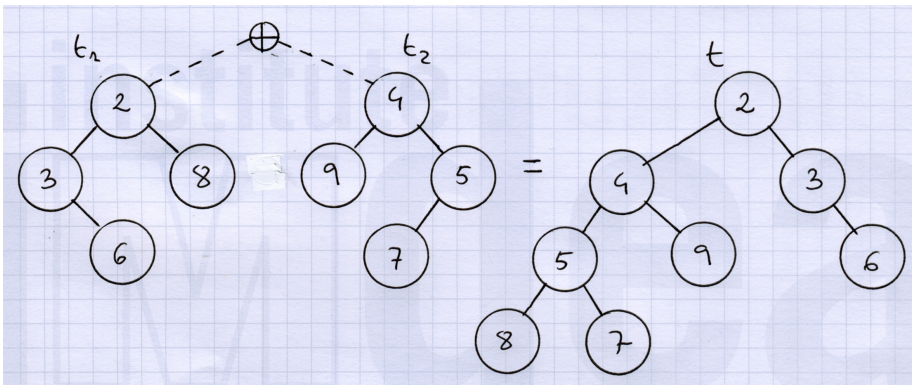
⋆ El código correspondiente es:

```
fun unir (t1 : arbin, t2 : arbin) dev (t : arbin) =
   caso vacio? (t1)                    → t2
     [] ¬vacio? (t1) ∧ vacio? (t2)     → t1
     [] ¬vacio? (t1) ∧ ¬vacio? (t2)    → (i1, x1, d1) = desc (t1);
                                         (i2, x2, d2) = desc (t2);
                                         caso x1 < x2  → crear (unir (d1, t2), x1, i1)
                                           [] x1 ≥ x2  → crear (unir (d2, t1), x2, i2)
                                         fcaso
   fcaso
ffun
```

⋆ Veamos que esta transformación garantiza la propiedad de montículo en el árbol binario resultante:

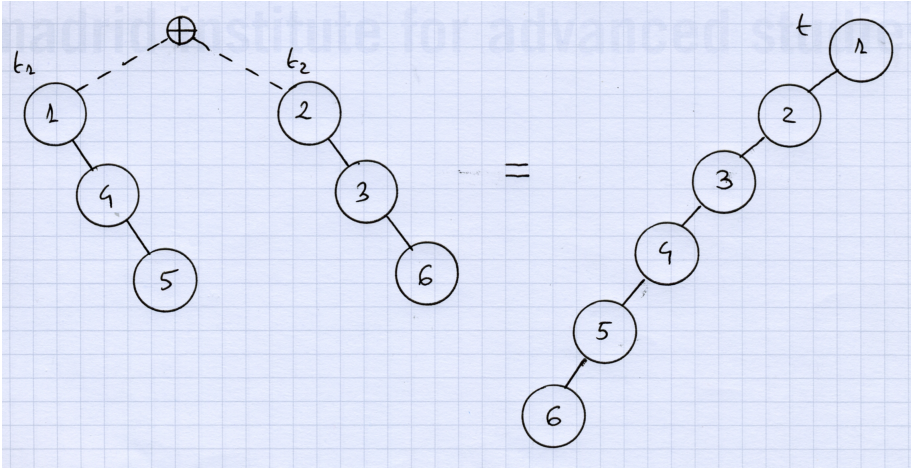
- La raíz es el mínimo de ambos montículos y preserva esta propiedad con respecto a su hijo derecho, porque este era previamente su hijo izquierdo en uno de los montículos de partida.
- Los elementos de su hijo izquierdo son mayores o iguales que dicha raíz porque todos ellos provienen, o bien de su previo hijo derecho, o bien de un montículo cuyos elementos son todos ellos mayores o iguales que dicha raíz.
- El hijo izquierdo es un montículo, por hipótesis de inducción sobre la propia operación *union*.

⋆ Ejemplo de aplicación del algoritmo:



3. Estudio del coste

⋆ Si los montículos de entrada fueran árboles binarios arbitrarios, el caso peor de la unión es aquél en el que todos los elementos están en las respectivas espinas derechas. En ese caso, el coste sería $O(n_1 + n_2)$.



- Sin embargo, veremos que si tales árboles fueran posibles, la mayor parte del coste real de la unión, se paga mediante el decrecimiento de la función de potencial, de forma que el coste amortizado de la misma está siempre en $O(\log n)$, siendo n el cardinal del montículo resultado de la unión.

- En definitiva, probaremos que una secuencia de m uniones, resultando en un montículo final de cardinal n, tiene un coste peor en $O(m \log n)$, y por tanto el coste amortizado de cada unión está en $O(\log n)$.

- *Definición:* Diremos que un nodo x es **pesado** si el cardinal de su subárbol derecho d es mayor que el de su subárbol izquierdo i, expresado $n(d) > n(i)$. En caso contrario, diremos que x es **ligero**.

- Dados dos montículos segados $t_1,\ t_2$, de cardinales $n_1,\ n_2$, número de nodos ligeros $l_1,\ l_2$, y número de nodos pesados $p_1,\ p_2$, definimos la **función de potencial** antes de unir los montículos como $\Phi(t_1, t_2) = p_1 + p_2$. La función de potencial del monticulo resultante t se define como el número de sus nodos pesados $\Phi(t) = p$.

- *Teorema:* Si n es el cardinal resultante de la unión de dos montículos sesgados, el coste amortizado de dicha operación está en $O(\log n)$.

  Demostración: Consta de los siguientes pasos:

  1. La primera observación es que los únicos nodos que pueden cambiar su carácter (de pesado a ligero, o al contrario) son los que están en las espinas derechas de ambos montículos. Esto es debido a que los subárboles izquierdos de cada nodo de dichas espinas son copiados íntegros al montículo final. Llamaremos $l'_1,\ l'_2$ y $p'_1,\ p'_2$ repectivamente a los nodos ligeros y pesados de las espinas de t_1 y t_2.
  2. La segunda observación es que los **todos** los nodos pesados de dichas espinas pasan a ser ligeros. Tomando por ejemplo el caso $x_1 < x_2$ del código (el otro es simétrico), si x_1 era pesado en t_1, entonces $n(d_1) > n(i_1)$. Por tanto $n(d_1) + n(t_2) > n(i_1)$ y x_1 será ligero en el montículo final t.
  3. Algunos nodos ligeros de las espinas pueden pasar a ser pesados. Si en el caso $x_1 < x_2$, x_1 era ligero, entonces $n(i_1) \geq n(d_1)$. Si se cumple $n(i_1) > n(d_1) + n(t_2)$, entonces x_1 será pesado en t.
  4. El número l' de nodos ligeros en la espina derecha de un árbol cualquiera de cardinal n es a lo sumo $l' \leq \lfloor \log n \rfloor + 1$. Esto es debido a que el hijo derecho de un nodo ligero tiene a lo sumo $n/2$ elementos. Se suma 1 porque una hoja tiene $n = 1$ y es ligera.
  5. El coste real c de la unión es la suma del número de nodos de las espinas derechas, es decir:

$$c = l'_1 + p'_1 + l'_2 + p'_2 \leq \lfloor \log n_1 \rfloor + 1 + p'_1 + \lfloor \log n_2 \rfloor + 1 + p'_2 \leq 2 \log n + p'_1 + p'_2$$

6. El incremento de potencial es la suma de los nodos que pasan de ligeros a pesados menos la suma de los que pasan de pesados a ligeros:

$$\Delta\Phi \leq l'_1 + l'_2 - (p'_1 + p'_2) \leq 2\log n - (p'_1 + p'_2)$$

7. Por tanto, el coste amortizado de la unión resulta ser:

$$\hat{c} = c + \Delta\Phi \leq 2\log n + p'_1 + p'_2 + 2\log n - (p'_1 + p'_2) = 4\log n \in O(\log n)$$

⋆ La operación *insert* (t, x) consiste en crear un montículo unitario $t' =$ *crear* $(vacio, x, vacio)$ y luego unir t y t'. Las hojas son ligeras, por tanto la función de potencial inicial $\Phi(t, t')$ coincide con $\Phi(t)$. El razonamiento dado para la unión sigue siendo válido para la inserción.

⋆ La operación *borraMin* comienza eliminando la raíz. Si esta era ligera, el potencial de los dos hijos coincide con el del montículo antes de borrar y el coste amortizado de unir los hijos sigue estando en $O(\log n)$. Si la raíz era pesada, al borrarla el potencial del conjunto decrece en 1 y entonces $\Delta\Phi$ es una unidad más pequeña, por lo que el coste $O(\log n)$ sigue siendo una cota superior de $\hat{c}$.

⋆ El montículo vacío tiene potencial $\Phi_0 = 0$, y cualquier montículo t tiene potencial no negativo $\Phi(t) \geq 0$.

⋆ Concluimos entonces que en cualquier secuencia de operaciones de inserción, unión o borrado, que comience en el montículo vacío, el coste amortizado de cada una de ellas está en $O(\log n)$.

Lecturas complementarias

Estas notas están basadas en el Cap. 22 de [2], donde hay más ilustraciones y explicaciones con las que el lector podrá completar lo resumido aquí.

Referencias

[1] R. Peña. *Diseño de Programas: Formalismo y Abstracción.* Tercera edición. Pearson Prentice-Hall, 2005.

[2] M. A. Weiss. *Estructuras de datos en Java.* Addison Wesley, 2000.