Montículos n–arios

UNIVERSIDAD
COMPLUTENSE
MADRID

Inmaculada Pérez Garbín

Doble Grado en Ingeniería Informática y Matemáticas

Curso 2018–2019

Índice general

Prefacio

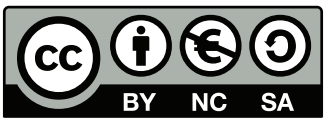
Este documento es un trabajo de investigación para la asignatura "*Métodos algorítmicos en resolución de problemas*", impartida por Narciso Martí Oliet y Clara María Segura Díaz en el curso 2018–2019 al curso de tercero del doble grado de Ingeniería Informática – Matemáticas en la Facultad de Informática de la Universidad Complutense de Madrid (UCM).

Contenido

Este texto contiene una implementación detallada de los montículos n–arios y establece un estudio comparativo entre los diferentes costes de dichas estructuras.

Las palabras resaltadas con este color contienen hipervínculos.

Licencia



El código fuente de este documento, así como el código de la implementación de los montículos n–arios, es de libre acceso y se encuentra alojado en https://github.com/Inmapg/d-ary-heaps paro uso y disfrute de todo el que quiera, siempre que se respeten los términos de la licencia.

Capítulo 1

Fundamentos teóricos

Para comprender la implementación y estructura de los *montículos n–arios* es necesario conocer todo en lo que se apoya. Es por eso que a lo largo de este capítulo daremos las nociones necesarias para adquirir una visión general de lo que más adelante implementaremos y analizaremos.

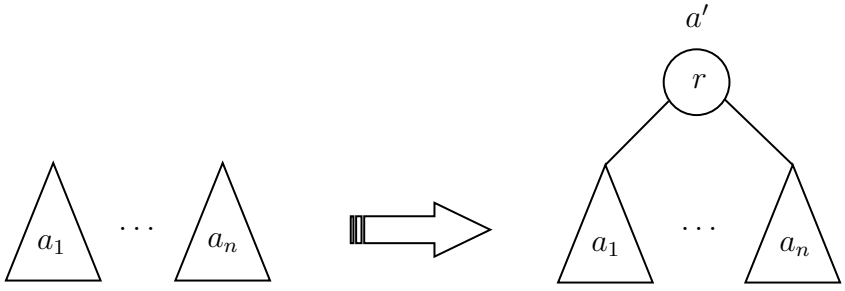
1.1. Árboles generales

Los Tipos Abstractos de Datos arborescentes se utilizan para representar datos organizados en jerarquías, como podría ser un árbol genealógicoo una estructura de directorios y archivos de un sistema operativo.

Entonces, es importante comprender cómo se constituye un árbol. Para ello, veamos algunos conceptos básicos.

Definición 1.1.1 (Nodo)**.** Un *nodo* es un punto de intersección, conexión o unón de varios elementos. En el caso que nos ocupa, será un registro que contendrá un dato de interés y, al menos, un puntero para referenciar a otro nodo.

Observación 1.1.1. Un solo nodo forma un árbol a. Se dirá que ese nodo es la *raíz* del árbol. Luego, dados n árboles $a_1, \ldots, a_n$, podemos construir un nuevo árbol a' añadiendo un nuevo nodo r como raíz y conectándolo con las raíces de los árboles $a_i, 1 \leq i \leq n$, los cuales se llamarán *subárboles* o *hijos de* a'. A r se le llamará *padre* de las raíces de estos subárboles. $\diamondsuit$



Definición 1.1.2 (Hojas)**.** Se llaman *hojas* a los nodos del árbol que no tienen hijos. Al resto de nodos se les llamará *nodos internos*.

Definición 1.1.3 (Nodos hermanos)**.** Dos nodos que comparten padre se dice que son *hermanos*.

Definición 1.1.4 (Camino)**.** Un *camino* es una sucesión de nodos en la que cada nodo es padre del siguiente. Al número de nodos en un camino se le llama *longitud*.

Definición 1.1.5 (Rama)**.** Una *rama* es cualquier camino que empieza en la raíz y acaba en una hoja.

Definición 1.1.6 (Nivel)**.** El *nivel* o *profundidad* de un nodo es la longitud del camino que va desde la raíz hasta él. En particular, el nivel de la raíz es el 1.

Definición 1.1.7 (Altura)**.** La *altura* o *talla* de un árbol es el máximo de los niveles de todos los nodos del árbol.

Definición 1.1.8 (Grado)**.** El *grado* o *aridad* de un nodo es su número de hijos.

Definición 1.1.9 (Antepasado/Descendiente)**.** Decimos que un nodo α es *antepasado* de β (respectivamente, β es *descendiente* de α) si existe un camino desde α hasta β.

Distinguimos distintos tipos de árboles en función de sus características:

- Árboles ordenados o no ordenados: un árbol es *ordenado* si el orden de los hijos de cada nodo es relevante. No debe confundirse este tipo de árbol ordenado, basado en la estructura de árbol, con los árboles binarios de búsqueda, en los que el orden está basado en el valor de los nodos del árbol.
- Árboles n–arios: un árbol es *n–ario* si el máximo número de hijos de cualquier nodo es n. Los *árboles generales* son la unión de todos los n–arios.
- Árboles binarios: un *árbol binario* es un árbol ordenado cuyos nodos siempre tienen dos hijos (el izquierdo y el derecho), aunque estos pueden ser vacíos.

Como se podrá intuir, existen formas de recorrer esta estructura. Veamos, para un árbol binario, los más importantes:

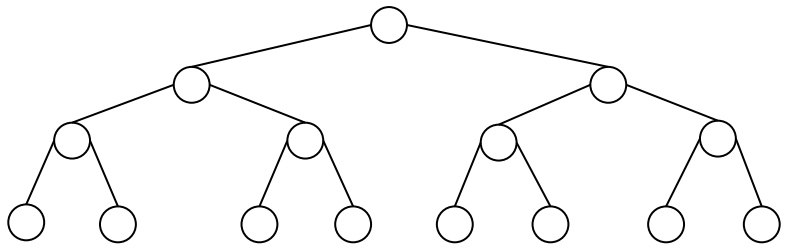
- Recorrido en profundidad (*DFS*):
  - Preorden: se visita en primer lugar la raíz del árbol y, a continuación, se recorren en preorden el hijo izquierdo y el hijo derecho.
  - Inorden: se recorre el hijo izquierdo, después se visita la raíz, y, por último, se recorre el hijo derecho.
  - Postorden: primero se recorren los hijos izquierdo y derecho (en ese orden), y después se visita la raíz.
- Recorrido en anchura (*BFS*): intuitivamente, se comienza en la raíz y se exploran todos los vecinos de este nodo. A continuación, para cada uno de los vecinos se exploran sus correspondientes vecinos adyacentes, y así hasta que se recorra todo el árbol.

Definición 1.1.10 (Árbol binario de búsqueda)**.** Los *árboles binarios de búsqueda* son árboles binarios cuyos nodos guardan elementos sobre los cuales hay definido un orden total estricto y que satisfacen la siguiente propiedad adicional: el elemento en cada nodo es mayor que todos sus descendientes izquierdos y menor que todos sus descendientes derechos.

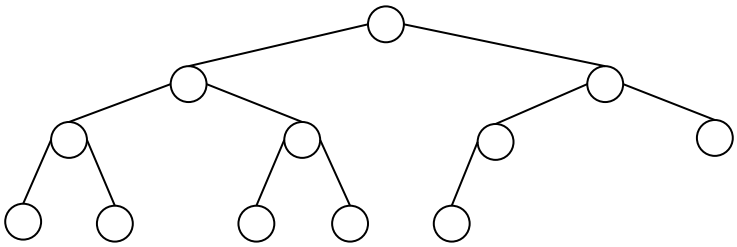
Observación 1.1.2. Equivalentemente, o bien el árbol es vacío, o bien el elemento en la raíz es el mayor que los elementos del hijo izquierdo y menor que los elementos del hijo derecho, y recursivamente los dos hijos son a su vez árboles binarios de búsqueda. ♢

Observación 1.1.3. Las operaciones de búsqueda, inserción y borrado de un elemento tienen un coste lineal respecto a la altura del árbol. En el caso peor, la altura es lineal respecto al número de nodos; aunque, en promedio, es logarítmica respecto al número de nodos. ♢

Definición 1.1.11 (Árbol completo)**.** Un árbol de altura h es *completo* cuando todos sus nodos internos tienen el máximo número de hijos no vacíos y todas sus hojas están en el nivel h.



Definición 1.1.12 (Árbol semicompleto)**.** Un árbol de altura h es *semicompleto* si bien es completo o tiene vacantes una serie de posiciones consecutivas del nivel h empezando por la derecha, de tal manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo.



1.2. Colas

El tipo *cola* es una estructura de datos lineal cuya característica principal es que el acceso a los elementos se realiza en el mismo orden que fueron almacenados, siguiendo el criterio *FIFO*[1]. Este criterio es muy utilizado en el diseño de algoritmos para diversas aplicaciones, sobre todo en simulación, debido a la ubicuidad de las colas en toda clase de sistemas.

El comportamiento de las colas es totalmente independiente del tipo de los datos almacenados en ellas, por lo que se trata de un tipo de datos genérico.

Las operaciones con las que cuenta esta estructura son crear una cola vacía, añadir un elemento al final de la cola, consultar o eliminar el primer elemento (si existe), y determinar si la cola es vacía.

1.2.1. Colas con prioridad

Como acabamos de ver, en las colas ordinarias se atiende por riguroso orden de llegada. Sin embargo, en la vida cotidiana también existe otra clase de colas donde uno tiene la impresión de que siempre son atendidos los demás aunque hayan llegado después. Se trata de colas como las de los servicios de urgencias, en los cuales se atiende según el nivel de urgencia y no según el orden de llegada.

La estructura de datos lineal que representa este concepto se conoce como *cola de prioridad*, cuyas principales operaciones son añadir un elemento, saber quién es el primero y atender al primero, es decir, eliminarlo de la cola.

[1] El primero en entrar es el primero en salir

Cada elemento tiene una prioridad que determina quién va a ser el primero en ser atendido; para poder hacer esto, hace falta tener un *orden total* sobre las prioridades. El primero en ser atendido puede ser el elemento con menor prioridad (por ejemplo, el cliente que necesita menos tiempo para su atención) o el elemento con mayor prioridad (por ejemplo, el cliente que esté dispuesto a pagar más por su servicio) según se trate de *colas con prioridad de mínimos* o *de máximos*, respectivamente.

Luego lo que diferencia una cola de prioridad de una cola FIFO es que el primer elemento en salir es el de mayor prioridad. Frecuentemente, para facilitar la presentación de las propiedades de la estructura, tomaremos como prioridad el valor del elemento almacenado.

Las colas de prioridad pueden implementarse de diferentes maneras. Si se utilizan listas (sin ordenar), la operación de añadir puede hacerse en tiempo constante, pero consultar y eliminar el mínimo tienen coste lineal por la búsqueda. Si la lista está ordenada de forma creciente, estas dos operaciones tienen coste constante, pues el mínimo aparece al principio de la lista, pero añadir un elemento pasa a tener un coste lineal, debido al coste de localizar el punto de inserción.

La implementación eficiente de una cola de prioridad requiere un tipo de datos denominado *montículo*, que es una particularización de los árboles binarios, no obstante lo cual, la visión que un usuario tiene de un montículo es la de una cola de prioridad; es decir, la de un TAD[2] que permite añadir elementos desordenadamente y recuperarlos con un orden específico.

1.3. Montículos

Es fácil convencerse de que su modelo inicial corresponde a la idea matemática de *multiconjunto*[3], con la propiedad adicional de que existe una relación de orden que permite preguntar por el mínimo o el máximo de ellos, dependiendo del tipo de comparación que se haya establecido previamente.

Definición 1.3.1 (Montículo)**.** Un *montículo* es un árbol binario que cumple dos propiedades adicionales:

- Son árboles casi completos.
- Orden especial.

Cuando aseguramos que cumplen con un orden especial, hacemos referencia a los siguientes tipos de montículos.

Definición 1.3.2 (Montículo de mínimos)**.** Un *montículo de mínimos* es un árbol binario semicompleto donde el elemento situado en la raíz es menor que todos los elementos en el hijo izquierdo y en el derecho, y ambos hijos son, a su vez, montículos de mínimos.

Observación 1.3.1. Equivalentemente, el elemento allmacenado en cada nodo es menor que los elementos en las raíces de sus hijos y, por tanto, que todos sus descendientes; así, la raíz contiene el mínimo de todos los elementos en el árbol. ♢

Definición 1.3.3 (Montículo de máximos)**.** Un *montículo de máximos* se define de manera completamente análoga, exigiendo que cada nodo sea mayor que sus hijos, de forma que la raíz del árbol contiene el máximo de todos los elementos.

[2]Tipo Abstracto de Datos.

[3]Conjuntos en los que pueden existir elementos repetidos

Por supuesto, según sea de mínimos o máximos, el montículo es apropiado para implementar una cola con prioridad de mínimos o de máximos, respectivamente.

Visto y entendido todo lo anterior, estudiemos una serie de propiedades interesantes sobre los árboles binarios.

Proposición 1.3.1.
Un árbol binario completo de altura h cumple que

1. *Tiene 2^{i-1} nodos en el nivel i, con $1 \leq i \leq h$.*
2. *Tiene 2^{h-1}, siendo $h \geq 1$.*
3. *Tiene $2^h - 1$ nodos, siendo $h \geq 0$.*

Demostración.

1. Veámoslo por inducción sobre el número de nivel i.

   Cuando $i = 1$, en el primer nivel solamente hay un nodo (la raíz) y, por tanto, se tiene que $2^{i-1} = 2^{1-1} = 2^0 = 1$.

   Supuesto cierto el resultado para $i < h$, como cada nodo en el nivel i–ésimo tiene exactamente dos hijos no vacíos, el número de nodos en el nivel $i+1$ será $2 \cdot 2^{i-1} = 2^i = 2^{(i+1)-1}$.

2. Es trivial, pues las hojas son los nodos en el último nivel h.

3. Si $h = 0$, el árbol es vacío y el número de nodos es igual a $0 = 2^0 - 1$.

   Si $h > 0$, el número total de nodos es $\displaystyle\sum_{i=1}^{h} 2^{i-1} = \sum_{j=0}^{h-1} 2^j = 2^h - 1$.

■

Proposición 1.3.2.
Un árbol binario semicompleto formado por n nodos tiene altura $\lfloor \log n \rfloor + 1$.

Demostración.

Supongamos que tenemos un árbol binario semicompleto con n nodos y altura h.

En el caso en que faltan mas nodos en el último nivel, el árbol es un árbol binario completo de $h-1$ niveles más un nodo de nivel h, por lo que hay en total $2^{h-1} - 1 + 1 = 2^{h-1}$ nodos.

Resumiendo, tenemos con respecto a n la siguiente desigualdad:

$$2^{h-1} \leq n \leq 2^h - 1$$

Tomando logaritmos en base dos

$$\log(2^{h-1}) \leq \log n \leq \log(2^h - 1) < \log(2^h)$$

equivalentemente,

$$h - 1 \leq \log n < h$$

es decir, $h - 1 = \lfloor \log n \rfloor$ y de aquí, despejando, sacamos $h = \lfloor \log n \rfloor + 1$ ■

1.3.1. Montículos n–arios

Definición 1.3.4 (Montículo n–ario)**.** Un *montículo n–ario*[4] es una estructura de datos que generaliza la de los montículos binarios con la diferencia de que, en ella, cada nodo tiene n hijos en lugar de dos. Quitando esa diferencia, cumplen las mismas propiedades.

Entonces, como hemos comentado antes, podemos utilizar este TAD para realizar una implementación eficiente de una cola con prioridad. Luego una cola de este tipo consta de un vector de n elementos, cada uno de ellos con una prioridad asociada. Estos elementos se pueden ver como nodos en un árbol n–ario completo y se encuentran listados de forma que se *recorra en anchura* (BFS). El padre de un nodo en la i–ésima posición, $i \geq 0$, es el elemento colocado en la posición $\dfrac{i-1}{n}$ y sus hijos se encuentran en las posiciones $n \cdot i + 1$ hasta $n \cdot i + n$.

La altura de este tipo de estructura es $\Theta(\log_n k)$, donde, haciendo un pequeño cambio en la notación, k es el número de nodos del montículo. En efecto,

$$\begin{array}{ccccc} 1 + n + n^2 + \ldots + n^{h-1} & < & k & \leq & 1 + n + n^2 + \ldots + n^h \\ \dfrac{n^h - 1}{n-1} & < & k & \leq & \dfrac{n^{h+1} - 1}{n-1} \\ n^h & < & k \cdot (n-1) + 1 & \leq & n^{h+1} \\ h & < & \log_n(k \cdot (n-1) + 1) & \leq & h + 1 \end{array}$$

lo que resulta en $h = \lceil (\log_n(k \cdot (n-1) + 1)) - 1 \rceil \in \Theta(\log_n k)$.

Los posibles ámbitos de aplicación de este TAD son múltiples. Por ejemplo, cuando se opera con un grafo de A aristas y V vértices, tanto el algoritmo de Dijkstra como el de Prim utilizan un montículo de mínimos donde hay V operaciones de borrado del mínimo y A operaciones de disminuir la prioridad de los elementos. Usando un montículo n–ario con $n = \dfrac{A}{V}$, se podría optimizar el algoritmo para que su coste fuera $O(A \cdot \log_{A/V}(V))$, mejorando el ya conocido coste $O(A \cdot \log(V))$ de hacerlo con un montículo binario.

Otra opción sería utilizar los *montículos de Fibonacci* que otorga un mejor tiempo, $O((A + V) \cdot \log(V))$, aunque los montículos que ahora nos conciernen son casi tan rápidos como los de Fibonacci para los problemas mencionados.

[4] Este tipo de estructura fue inventada en 1975 de la mano de *Donald B. Johnson*

Capítulo 2

Implementación de los montículos n–arios

Para comprender el código adjunto a este documento, es necesario saber que el elemento con menor prioridad en un montículo de mínimos (o el de mayor prioridad, de tratarse de un montículo de máximos) **siempre** se hallará en la posición 0 del vector.

El código es genérico; es decir, es válido para los dos tipos de montículos definidos anteriormente.

A continuación, se procederá a realizar una descripción de las funciones menos triviales del código implementado.

2.1. Creación del montículo

En este caso, existen tres formas diferentes de construirlo: especificando el número de hijos por nodo que se desean tener (y un *objeto comparador* si se desea), pasando como parámetro otro montículo existente, o, también, con un vector.

2.2. Inserción de un elemento

El elemento se añade al final del vector y se *flota*[1] en el árbol hasta que las propiedades de este tipo de montículo se satisfagan.

[1] Estudiaremos esta función en la siguiente página.

2.3. Borrado de un elemento

Se intercambia dicho elemento con el último del vector (llamémoslo x) y se hace decrecer el tamaño de éste. Entonces, mientras que x no satisfaga las restricciones del montículo, se cambia con uno de sus hijos (el que tenga menor prioridad, si el montículo es de mínimos; el que tenga mayor prioridad, si es de máximos), hundiéndolo en el árbol y, después, en el vector hasta que se cumplan las propiedades del TAD. El mismo procedimiento de hundir se aplica al incrementar la prioridad de un elemento en un montículo de mínimos o al disminuirla de uno de máximos.

Algoritmo 1 Borrar un elemento del montículo dada su posición y devolver dicho elemento.

```
función BORRAR(ind : ent)
    si vacio() ∨ ind ≥ tam() entonces
        devolver error;
    si no
        elem ← vector[ind];
        vector[ind] ← vector[tam() − 1];
        vector.eliminaUltimoElemento();
        si ¬vacio() ∧ ind < tam() entonces
            hundir(ind);
        fin si
    fin si
    devolver elem;
fin función
```

2.4. Hundir un elemento

Corresponde a bajar de nivel el elemento hasta situarlo en el lugar correspondiente de tal forma que se satisfagan las propiedades de este tipo de montículo.

Algoritmo 2 Hundir un elemento del montículo dada su posición.

```
procedimiento HUNDIR(i : ent)
    lugar ← i;
    elem ← vector[i];
    salir ← falso;
    mientras ¬salir ∧ nesimoHijo(lugar, 1) ≤ tam() hacer
        hijo ← posPrioritaria(lugar);
        si antes(vector[padre(lugar)], elem) entonces      ▷ Objeto comparador
            vector[lugar] ← vector[hijo];
            lugar ← hijo;
        si no
            salir ← cierto;
        fin si
    fin mientras
    vector[lugar] ← elem;
fin procedimiento
```

2.5. Flotar un elemento

En esta operación el elemento en cuestión va a subir de nivel del árbol hasta colocarse en la posición que le corresponda para cumplir las restricciones de la cola.

Algoritmo 3 Flotar un elemento del montículo dada su posición.

```
procedimiento FLOTAR(i : ent)
    lugar ← i;
    elem ← vector[i];
    mientras lugar > 0  ∧  antes(elem, vector[padre(lugar)]) hacer
        vector[i] ← vector[padre(lugar)];
        lugar ← padre(lugar);
    fin mientras
    vector[lugar] ← elem;
fin procedimiento
```

Capítulo 3

Comparación de los montículos n–arios

Esta estructura de datos permite cambiar la prioridad de las operaciones a realizar de una forma más rápida que los montículos binarios, con el inconveniente del aumento del coste de las operaciones de borrado de elementos.

La inserción de un elemento en un montículo n–ario tiene el coste de la operación *flotar*, ya que lo único que añade es una inserción al final del vector, lo cual tiene coste $O(1)$. Así, tendrá coste $O(\log_n(k))$[1]. Sin embargo, para un n grande, la operación de *borrar mínimo/máximo* es más costosa, pues aunque el árbol tenga menos profundidad, se deberá encontrar el mínimo de los n hijos, lo que conlleva $n-1$ comparaciones usando el algoritmo estándar. Esto nos conduce a tener un coste del orden $O(n \cdot \log_n(k))$. Si n es constante, tanto la inserción como el borrado serán $O(\log k)$.

Además, tiene un mejor comportamiento con la memoria caché, permitiendo ejecutar los programas, en la práctica, de una manera más rápida aunque, teóricamente, estos montículos tengan un tiempo de ejecución en el peor de los casos mayor en operaciones como la anterior mente mencionada.

Como los montículos binarios, los n–arios son un tipo de estructura de gran utilidad desde el punto de vista del ahorro de memoria, pues únicamente ocupan el almacenamiento de datos del vector.

Sería conveniente usar este tipo de estructura cuando se requiera hacer más inserciones que borrados o extracciones del mínimo, si el montículo es de mínimos, o del máximo, si el montículo es de máximos. Esto se debe a que la inserción tiene un coste menor que las otras operaciones mencionadas.

[1] Llamaremos k al número de elementos

Bibliografía

[1] N. Martí Oliet, Y. Ortega Mallén y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*, capítulo **8**, segunda edición, Pearson/Prentice Hall, 2013.

[2] Ricardo Peña Marí, *Diseño de programas: formalismo y abstracción*, capítulo **7.5.2**, tercera edición, Pearson/Prentice Hall, 2004.

[3] University of Waterloo, *Comparison of d–ary heaps*.

[4] Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++*, cuarta edición, Pearson/Prentice Hall, 2014.

[5] F. M. Carrano y J. J. Prichard, *Data Abstraction and Problem Solving with C++*, tercera edición, Addison-Wesley, 2012.