Ampliación de Estructuras de Datos

Amalia Duch

Barcelona, marzo de 2007

Índice

1. Diccionarios implementados con árboles binarios de búsqueda

Para comenzar esta sección recordaremos dos definiciones que ya se han estudiado en asignaturas anteriores. Éstas son la definición de tipo abstracto de datos y la definición del tipo abstracto de datos diccionario.

Definición 1.1 *Un **tipo abstracto de datos** (TAD) es un conjunto de elementos junto con las operaciones definidas en él, independientemente de su implementación.*

Definición 1.2 *El **TAD diccionario** es un conjunto S de elementos con una clave y una información asociadas. Las claves deben ser comparables entre ellas (es decir, el conjunto de claves está ordenado bajo un orden lineal). Las operaciones definidas para el TAD diccionario son: crear un diccionario vacío, determinar (buscando su clave) si un elemento dado está o no en el diccionario, insertar un elemento dado en el diccionario y borrar un elemento dado del diccionario.*

En notación funcional las operaciones del diccionario las denotaríamos como sigue:

Crear: `dicc_vacio:` $\rightarrow$ `dicc`

Buscar: `dicc_pertenece: dicc, elem` $\rightarrow \{0, 1\}$

Insertar: `dicc_insertar: dicc, elem` $\rightarrow$ `dicc`

Borrar: `dicc_borrar: dicc, elem` $\rightarrow$ `dicc`

El TAD diccionario puede implementarse eficientemente utilizando diferentes estructuras de datos como pueden ser las tablas de dispersión (o tablas de *hash*) con encadenamiento separado, las tablas de dispersión con encadenamiento lineal, los árboles binarios de búsqueda o los árboles AVL, entre otras.

Las tablas de dispersión son especialmente eficientes en aplicaciones en las que sólo se realizarán las operaciones elementales del TAD diccionario (crear, buscar, insertar y borrar) pues en éstos casos su coste es proporcional al factor de carga de la tabla, valor que tiende a una constante cuando la tabla está bien diseñada (ver Cuadro 1). Sin embargo en aplicaciones en las que además de las operaciones elementales es necesario realizar búsquedas por los elementos con la menor (o la mayor) clave, las tablas de dispersión dejan de ser eficientes pues éstas búsquedas tienen coste $\Theta(n)$ para tablas de dispersión de n elementos.

Una alternativa en éstos casos es implementar el TAD diccionario utilizando árboles binarios de búsqueda.

Definición 1.3 *Un* **árbol binario de búsqueda** *(ABB) es un árbol binario en el que los nodos están etiquetados por las claves de los elementos que se almacenan en él y en el que se cumple la propiedad de que para todo nodo etiquetado con la clave x, todos los nodos de su subárbol izquierdo están etiquetados con claves menores que x y todos los nodos de su subárbol derecho están etiquetados con claves mayores que x.*

Un ABB se representa computacionalmente con un puntero a un *nodo* que contiene la clave del elemento que guarda, la información asociada a ese elemento, un puntero a su subárbol izquierdo y un puntero a su subárbol derecho. Los ABBs vacíos se codifican con un puntero nulo. El coste de crear un ABB vacío es $\Theta(1)$ y el coste en espacio (coste espacial) de un ABB de n elementos es en todos los casos $\Theta(n)$.

El coste en el caso peor de la mayoría de las operaciones en un ABB de n elementos es $\Theta(n)$ ya que una entrada ordenada puede convertir al ABB en una lista. En el caso de diccionarios semi-estáticos (sin la operación de borrado) con inserciones aleatorias, el coste en el caso medio de las operaciones de búsqueda e inserción es $\Theta(\log n)$ para diccionarios de n elementos implementados con ABBs. El coste en caso promedio de todas las operaciones de un ABB de n nodos se calcula suponiendo que el ABB ha sido construido a partir de una permutación aleatoria de las claves de sus elementos.

Implementación. La implementación del TAD diccionario utilizando ABBs puede encontrarse en las transparencias de la asignatura.

Ejemplos y ejercicios de ABBs. Dados en clase.

Cabe hacer notar que los ABBs tienen la propiedad de que si se listan las claves de los nodos en *in–orden* éstas quedan ordenadas ascendentemente.

Búsquedas. Intuitivamente para buscar un elemento con clave x en un ABB no vacío cuya raíz está etiquetada con clave y se han de comparar x e y.

- Si son iguales la búsqueda ha terminado y la raíz contiene al elemento buscado.
- Si en cambio x es menor que y la búsqueda ha de continuar recursivamente en el subárbol izquierdo hasta encontrar la clave o bien un subárbol vacío.
- Finalmente en caso de que x sea mayor que y la búsqueda continua recursivamente en el subárbol derecho hasta encontrar la clave o un subárbol vacío.

En el caso peor el coste de esta operación es $\Theta(h)$ donde h es la altura del ABB. Para ABBs de n nodos h puede tomar valores entre $\Theta(\log n)$ (si el árbol está balanceado) y $\Theta(n)$ (si el árbol es una lista).

Código, ejemplos y ejercicios. Dados en clase.

Inserciones. La inserción de un elemento con clave x en un ABB vacío consiste en crear un nuevo nodo que contenga al elemento y toda su información asociada, éste nodo será la nueva raíz del ABB. Para insertar un elemento con clave x en un ABB no vacío hemos de comparar la clave x con la clave de la raíz del ABB y continuar la inserción recursivamente en el subárbol izquierdo si x es menor que la clave de la raíz y en el derecho si es mayor.

En el caso peor el coste de esta operación es $\Theta(h)$ donde h es la altura del ABB. Para ABBs de n nodos h puede tomar valores entre $\Theta(\log n)$ (si el árbol está balanceado) y $\Theta(n)$ (si el árbol es una lista).

Código, ejemplos y ejercicios. Dados en clase.

Es importante observar que los ABBs son sensibles al orden en el que se insertan las claves. Es decir que para el mismo conjunto de claves hay varios ABBs que las contienen y la forma (o estructura) del ABB dependerá del orden en que las claves hayan sido insertadas. Ver ejemplos dados en clase.

Borrados. Para borrar un nodo con clave x primero hemos de localizar este nodo en el árbol (búsqueda).

- Si se trata de un nodo con ambos subárboles vacíos (una hoja o nodo externo) simplemente lo eliminamos.
- En cambio si se trata de un nodo con al menos uno de sus subárboles no vacío (nodo interno) y simplemente lo borráramos desconectaríamos el ABB. Por tanto si el nodo que se ha de eliminar tiene un único subárbol hemos de reemplazarlo por la raíz de este subárbol.

- Si en cambio el nodo que hemos de eliminar tiene dos subárboles no vacíos este nodo se reemplaza por el nodo del subárbol derecho con menor clave.

En el caso peor el coste de esta operación es $\Theta(h)$ donde h es la altura del ABB. Para ABBs de n nodos h puede tomar valores entre $\Theta(\log n)$ (si el árbol está balanceado) y $\Theta(n)$ (si el árbol es una lista).

Código, ejemplos y ejercicios. Dados en clase.

Otras operaciones interesantes son las de buscar al nodo con clave menor (o mayor). Los algoritmos en este caso son fáciles pues el nodo con clave mínima (respectivamente máxima) tiene subárbol izquierdo (respectivamente derecho) nulo. Por tanto, comenzando desde la raíz del árbol debemos seguir hacia la izquierda (respectivamente derecha) hasta encontrar el primer subárbol nulo.

Código, ejemplos y ejercicios. Dados en clase.

El Cuadro 1 resume los costes de las operaciones del TAD diccionario implementado con diferentes estructuras de datos.

2. TAD Cola de Prioridad

Definición 2.1 *Un TAD* **cola de prioridad** *es un conjunto S de elementos en el que cada elemento x tiene una prioridad $p(x)$ asociada (donde se supone que las prioridades son miembros de un conjunto con un orden lineal definido). Las operaciones asociadas al conjunto de elementos son: construir una cola de prioridad vacía, insertar un elemento, buscar el elemento con prioridad mínima (o máxima pero no ambas) y borrar el elemento con prioridad mínima (o máxima pero no ambas).*

En notación funcional las operaciones del TAD cola de prioridad se escribirían como sigue:

Crear: `cua_prio_vacia:` $\rightarrow$ `cua_prio`

Buscar mínimo: `cua_prio_buscar: cua_prio` $\rightarrow$ `elem`

Insertar: `cua_prio_insertar: cua_prio, elem` $\rightarrow$ `cua_prio`

Borrar mínimo: `cua_prio_borrar: cua_prio` $\rightarrow$ `cua_prio`

Para implementar el TAD cola de prioridad podríamos utilizar listas ordenadas. En este caso las operaciones de crear la cola de prioridad vacía, buscar al elemento con prioridad mínima y borrar al elemento con prioridad mínima tendrían un coste constante mientras que el coste de insertar un nuevo elemento

Operaciones	ABB	AVL	TH-ES	TH-EL
Construir (a partir de un dicc. vacío)				
Caso Mejor	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(n)$
Caso Promedio	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(\alpha n)$	$\Theta(\alpha n)$
Caso Peor	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertar un elemento				
Caso Mejor	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
Caso Promedio	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\alpha)$	$\Theta\left(\frac{1}{1-\alpha}\right)$
Caso Peor	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Borrar un elemento				
Caso Mejor	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
Caso Promedio	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\alpha)$	$\Theta\left(-\frac{1}{\alpha}\log(1-\alpha)\right)$
Caso Peor	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Buscar un elemento que no está en el dicc.				
Caso Mejor	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
Caso Promedio	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\alpha)$	$\Theta\left(\frac{1}{1-\alpha}\right)$
Caso Peor	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Buscar un elemento que si está en el dicc.				
Caso Mejor	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Caso Promedio	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\alpha)$	$\Theta\left(-\frac{1}{\alpha}\log(1-\alpha)\right)$
Caso Peor	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Buscar elemento con clave mínima (máxima)				
Caso Mejor	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Caso Promedio	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Caso Peor	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

Cuadro 1: Resumen del coste de las operaciones del TAD Diccionario de n elementos implementado con árboles binarios de búsqueda (ABB)s, árboles AVL, tablas de dispersión con encadenamiento separado (TH-ES) y tablas de dispersión con encadenamiento abierto lineal (TH-EL), α es el factor de carga de la tabla de dispersión y el caso promedio del coste de todas las operaciones de un ABB de n nodos se calcula suponiendo que el ABB ha sido construido a partir de una permutación aleatoria de las claves de sus elementos.

sería lineal en el caso peor. Utilizando tablas de dispersión los costes empeorarían pues buscar y borrar el mínimo serían operaciones con coste lineal. Si se utilizaran árboles binarios de búsqueda en el caso peor el coste de insertar, buscar el mínimo y borrar el mínimo también tendrían un coste lineal, aunque en el caso promedio el coste sería logarítmico. Un coste logarítmico podría obtenerse en el caso peor para las operaciones de buscar el mínimo, borrar el mínimo e insertar un elemento utilizando árboles AVL. La mejor implementación que podemos obtener es una que combine el coste constante de acceder al elemento mínimo (como en las listas ordenadas) y el coste logarítmico de borrar el elemento mínimo e insertar un nuevo elemento en los árboles AVL, ambos en el caso peor. La estructura de datos que nos permite implementar una cola de prioridad con estos costes es el **heap**.

Definición 2.2 *Decimos que un árbol binario de tamaño n es* **completo** *si:*

1. *su altura es $\Theta(\log n)$,*
2. *todos sus niveles están llenos (i.e. tienen el máximo número de nodos posible), excepto quizás, el último y*
3. *todos sus nodos externos (hojas) están lo más a la izquierda posible*

Ejemplo. Dado en clase.

Definición 2.3 *Un* **heap** *(o árbol binario parcialmente ordenado) es un árbol binario completo en el que la prioridad (clave) asociada a cada nodo es menor o igual a las claves asociadas a los nodos de sus subárboles.*

Ejemplos. Dados en clase.

Observación 1. En algunas traducciones al castellano la palabra heap se traduce como montículo.

Observación 2. Un heap puede representarse vectorialmente mediante su recorrido por niveles. Esta propiedad permite el que su implementación no requiera punteros. Para implementar un heap mediante su representación vectorial, la primera posición del vector se deja libre y se comienzan a almacenar elementos en la segunda posición. De esta manera, un elemento situado en la i-ésima posición de la tabla tendrá como padre al elemento de la posición $\lfloor i/2 \rfloor$-ésima (con $i > 1$), el hijo izquierdo estará en la posición $2i$ (con $2i \leq n$) y el hijo derecho en la posición $2i + 1$ (con $2i + 1 \leq n$).

Operaciones

Buscar el elemento mínimo. Dada la definición de un heap el elemento con prioridad mínima será el elemento asociado a la raíz del árbol, por tanto el algoritmo consiste simplemente en acceder a la raíz y esto puede hacerse en tiempo constante. Coste: $\Theta(1)$.

Borrar el elemento mínimo. La idea del algoritmo es la siguiente.

- Para preservar la propiedad de tener un árbol binario completo, remplazamos la raíz del heap por la hoja que se encuentre más a la derecha (en la representación vectorial se ha de reemplazar la prioridad del elemento en la segunda posición por la prioridad del elemento de la última posición).
- A continuación hay que *hundir* el elemento que hemos puesto en la raíz para preservar la condición de las prioridades de un heap. Para ello intercambiaremos la prioridad de la nueva raíz por la prioridad de sus hijos que sea más pequeña y procederemos recursivamente hasta encontrar el sitio que le corresponde, es decir, el criterio de paro es que la clave sea menor que la clave de sus hijos o bien sea una hoja. En la representación vectorial hay que intercambiar la clave de la posición i con la menor de las claves en las posiciones $2i$ y $2i+1$, en caso de que sean posiciones válidas del vector.

El coste de este algoritmo en el caso peor es $\Theta(\log n)$ porque como máximo hemos de recorrer un camino completo del árbol que es completo. En el mejor de los casos la clave que se pone en la raíz ya es la más pequeña de todas lo que implicaría un coste de $\Theta(1)$. En promedio el coste es $\Theta(\log n)$.

Ejemplos. Dados en clase.

Código. En las transparencias de la asignatura.

Insertar un elemento. La idea del algoritmo es la siguiente.

- Para preservar la propiedad de tener un árbol binario completo, colocaremos al nuevo elemento como la hoja que se encuentre más a la derecha posible (en la representación vectorial se ha de insertar la prioridad del elemento en la última posición).
- A continuación hay que *subir* el nuevo elemento para preservar la condición de las prioridades de un heap. Para ello, si la prioridad de la nueva hoja es más pequeña que la prioridad de su padre, intercambiaremos las prioridades respectivas y procederemos de la misma manera hasta encontrar el sitio que le corresponde a la nueva prioridad, es decir, el criterio de paro es que la clave sea mayor o igual que la clave de su padre o bien sea la raíz del árbol. En la representación vectorial hay que intercambiar la clave de la posición i con la clave de la posición $\lfloor i/2 \rfloor$, siempre que i sea mayor que uno y la clave de i sea menor que la del padre.

El coste de este algoritmo en el caso peor es $\Theta(\log n)$ porque como máximo hemos de recorrer un camino completo del árbol que es completo. En el mejor de los casos la clave que se inserta ya es mayor o igual que la clave de su padre lo que implicaría un coste de $\Theta(1)$. En promedio el coste es $\Theta(\log n)$.

Ejemplos. Dados en clase.

Código. En las transparencias de la asignatura.

3. Heapsort

Como su nombre lo indica el algoritmo de heapsort utiliza un heap y sus operaciones asociadas para ordenar un vector de elementos. De forma general el algoritmo procede como sigue:

- Insertando en el heap todos los elementos (de hecho para no hacer uso de memoria adicional transforma el vector inicial en un heap) con un coste en el caso peor de $\Theta(n \log n)$ para un vector de n elementos, ya que corresponde al coste de hacer n inserciones en un heap.
- Mientras el heap no es vacío, el algoritmo procede recursivamente consultando al elemento mínimo y luego borrándolo.

El coste en el caso peor de estas operaciones es $\Theta(n)$ para las n consultas del elemento menor y $\Theta(n \log n)$ para los n borrados del elemento mínimo. En total el coste en el caso peor del heapsort es $\Theta(n \log n)$.

Es importante observar que, a diferencia de mergesort, este algoritmo no require de memoria adicional, pues se puede simular en el mismo vector una división entre el heap y la tabla ordenada (en este caso en orden decreciente), ya que cuando el heap incluye a todos los elementos de la tabla desordenada, la tabla ordenada está vacía. Y cada vez que se va borrando un elemento del heap este se va agregando a la tabla ordenada (ver código en las transparencias de la asignatura). Es fácil observar que el algoritmo de heapsort no es estable.