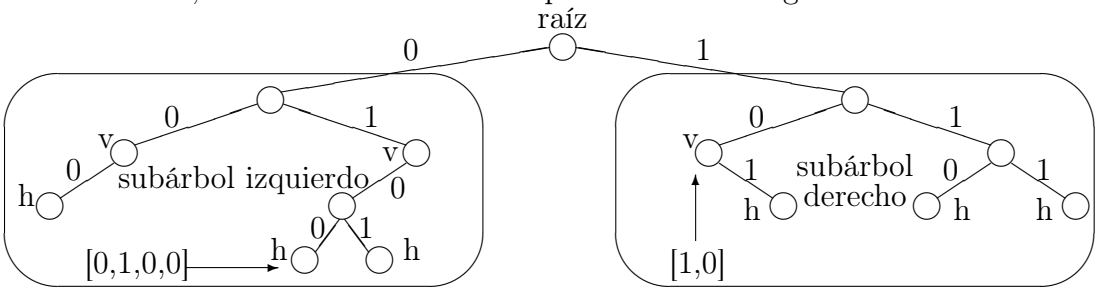
APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 2: ESTRUCTURAS DE DATOS

Árboles Binarios

TAD árbol binario. Ésta es una estructura muy usada en numerosas aplicaciones. Su nombre y demás terminología afín proviene de los árboles de la botánica, pero sobre todo de los árboles genealógicos. A pesar de que la definición puede generalizarse naturalmente, definiremos sólo los árboles binarios. Un árbol binario puede adoptar una de estas dos formas: o bien es un **árbol vacío** o bien es un **nodo** que, además de un elemento, tiene **dos subárboles** (de ahi el vocablo "binario"). Así, un árbol no vacío tiene una **raíz** (el elemento), un **subárbol izquierdo** y un **subárbol derecho**.

Gráficamente, los árboles binarios se representan de la siguiente forma:



En esta representación, los árboles vacíos no se dibujan, sólo se dibujan los nodos. Uno puede fácilmente deducir dónde se encuentran los árboles vacíos ya que cada nodo debe tener exactamente 2 subárboles. Así, los nodos marcados con una "v" tienen un subárbol vacío, y los marcados con una "h" tienen sus 2 subárboles vacíos.

TAD árbol_binario[elem]
constructores
<> : árbol_binario
< _, _, _ > : árbol_binario × elem × árbol_binario → árbol_binario
operaciones
raíz : árbol_binario → elem {se aplica sólo a un árbol no vacío}
izquierdo : árbol_binario → árbol_binario {se aplica sólo a un árbol no vacío}
derecho : árbol_binario → árbol_binario {se aplica sólo a un árbol no vacío}
es_vacío : árbol_binario → booleano
ecuaciones
raíz(<i,e,d>) = e
izquierdo(<i,e,d>) = i
derecho(<i,e,d>) = d
es_vacío(<>) = verdadero
es_vacío(<l,e,r>) = falso

Un nodo con sus dos subárboles vacíos (<<>,e, <>>) se llama **hoja** (en el ejemplo gráfico, son los marcados con una "h") y se denota más brevemente <e>. Dado un nodo n, sus 2 subárboles izquierdo y derecho se denominan usualmente **hijos**, respectivamente **hijo izquierdo** e **hijo derecho** de n. A su vez, n se dice **padre** de sus hijos. Dos árboles son **hermanos** cuando son hijo izquierdo y derecho del mismo padre. Un **camino** es una secuencia $A_1, A_2, \ldots, A_n$ de árboles tales que para cada i, A_{i+1} es hijo de A_i. La **longitud** de este camino es n-1. Además, A_1 se dice **ancestro** o **ascendiente** de A_n y A_n **descendiente** o **subárbol** de A_1. Decimos que siempre hay un camino de longitud 0 de todo árbol a sí mismo. La **altura** de un árbol es la longitud del camino que va desde él hasta su hoja más lejana. La **profundidad** de un subárbol es la longitud del camino que va desde el árbol hasta él. Se llama **nivel** a todo conjunto de subárboles de igual profundidad.

Intuitivamente, si seguimos la representación gráfica, un camino se puede identificar con un recorrido descendente del árbol. Toda esta terminología (salvo por los términos "izquierdo" y "derecho") puede ser extendido sencillamente a otras definiciones de árboles, como ternarios (cada nodo tiene 3 hijos), finitarios (cada nodo tiene una cantidad finita de hijos), etc.

Resulta conveniente hablar de las posiciones en que la información puede encontrarse en un árbol. En la representación gráfica dada, se decoró con un "0" la arista que une un nodo con su hijo izquierdo, y con un "1" la que lo une con su hijo derecho. Gracias a ello, una posición cualquiera dentro del árbol puede ser señalada dando una secuencia de 0's y 1's. Esta secuencia indica, si uno parte de la raíz, qué subárbol debe elegir (izquierdo o derecho) para llegar hasta la posición indicada. En la representación gráfica se ejemplifican las posiciones indicadas por [0,1,0,0] y por [1,0]. Se define el conjunto de posiciones pos = [{0,1}].

Dado un árbol t y una posición p, definimos $t \downarrow p$ para denotar el descendiente de t que se encuentra en la posición p de t.

$<>\downarrow p =<>$
$< i, e, d >\downarrow [] =< i, e, d >$
$< i, e, d >\downarrow (0 \rhd p) = i \downarrow p$
$< i, e, d >\downarrow (1 \rhd p) = d \downarrow p$

Nótese que si p no está entre las posiciones que aparecen en la representación gráfica de t, entonces $t \downarrow p =<>$. Definimos pos(t) como el conjunto de posiciones que sí aparecen en la representación gráfica: $\text{pos}(t) = \{p \in \text{pos} \mid t \downarrow p \neq <>\}$.

Ahora podemos definir la operación $t.p$ que devuelve el elemento alojado en la posición p de t. Esta operación sólo está definida para $p \in \text{pos}(t)$.

$< i, e, d > .[] = e$
$< i, e, d > .(0 \rhd p) = i.p$
$< i, e, d > .(1 \rhd p) = i.p$

o equivalentemente, se puede definir $t.p =$raíz$(t \downarrow p)$.

Utilizando punteros hay una forma inmediata de implementar árboles binarios, consistente en reemplazar cada arista de la representación gráfica justamente por un puntero:

```
type node = tuple
              lft: pointer to node
              value: elem
              rgt: pointer to node
            end
type bintree = pointer to node
fun empty() ret t:bintree
    t:= null
end
{Post: t ∼<> }
{Pre: l ∼ L ∧ e ∼ E ∧ r ∼ R}
fun node(l:bintree,e:elem,r:bintree) ret t:bintree
    alloc(t)
    t→lft:= l
    t→value:= e
    t→rgt:= r
{Post: t ∼ <L,E,R>} end
{Pre: t ∼ T ∧¬ is_empty(t)}
fun root(t:bintree) ret e:elem
    e:= t→value
end
{Post: e ∼ raíz(T)}
{Pre: t ∼ T ∧¬ is_empty(t)}
fun left(t:bintree) ret l:bintree
    l:= t→lft
end
{Post: l ∼ izquierdo(T)}
{Pre: t ∼ T ∧¬ is_empty(t)}
fun right(t:bintree) ret r:bintree
    r:= t→rgt
end
{Post: r ∼ derecho(T)}
{Pre: t ∼ T}
fun is_empty(t:bintree) ret b:bool
    b:= (t = null)
end
{Post: b ∼ es_vacío(T)}
```

Tratándose de estructuras en las que el programador debe administrar la memoria, es conveniente también disponer del destructor correspondiente:

```
{libera todo el espacio de memoria ocupado por t}
proc destroy(in/out t:bintree)
    if ¬ is_empty(t) then destroy(left(t))
                          destroy(right(t))
                          free(t)
                          t:= null
    fi
end
```

La implementación de árbol binario que se acaba de dar no es la única posible, se dió sólo a los fines de ilustrar una posible manera de hacerlo, y otro ejemplo del uso de punteros. De hecho, al presentar heaps, se dará una representación de los árboles binarios totalmente diferente (basada en la utilización de arreglos).

Existen numerosas aplicaciones de árboles binarios. Presentaremos dos de las más interesantes: árboles binarios de búsqueda, para implementar el TAD diccionario, y heaps, para implementar el TAD cola de prioridades.

Si T es un tipo, resulta conveniente denotar por $< T >$ al tipo de los árboles binarios cuyos elementos son los de T. Por simplicidad y mayor comprensión, abandonaremos por momentos el pseudo-código que veníamos usando y preferiremos un estilo de pogramación funcional, utilizando cuando resulte conveniente pattern-matching en vez de las operaciones raíz, izquierdo, derecho y es_vacío.

TAD diccionario y árboles binarios de búsqueda. Un diccionario es un tipo de dato que soporta las siguientes operaciones: creación del diccionario vacío, agregado de un elemento a un diccionario, consulta si el diccionario es vacío, búsqueda de un elemento en el diccionario y borrado de un elemento del mismo.

Se puede especificar como sigue:

TAD diccionario[elem]
constructores
    vacío : diccionario
    agregar : elem × diccionario → diccionario
operaciones
    es_vacío : diccionario → booleano
    está : elem × diccionario → booleano
    borrar : elem × diccionario → diccionario
ecuaciones
    es_vacío(vacío) = verdadero
    es_vacío(agregar(e,d)) = falso está(e,vacío) = falso
    está(e,agregar(e,d)) = verdadero
    e ≠ e' ⇒ está(e,agregar(e',d)) = está(e,d)
    borrar(e,vacío) = vacío
    borrar(e,agregar(e,d)) = borrar(e,d)
    e ≠ e' ⇒ borrar(e,agregar(e',d)) = agregar(e',borrar(e,d))

Para poder implementar eficientemente el TAD diccionario, es necesario contar con un orden entre los elementos. Esto permite utilizar árboles binarios de búsqueda (ABB).

Como su nombre lo indica son árboles binarios cuya información ha sido organizada de manera de facilitar la búsqueda de un elemento cualquiera que pueda estar almacenado en el árbol. Para que un árbol binario sea un ABB debe cumplirse, para todo nodo del árbol, que todos los elementos alojados en el hijo izquierdo sean menores que el alojado en el propio nodo, y que éste a su vez sea menor que todos los alojados en el hijo derecho. En símbolos,

$$\mathrm{ABB}(t) \stackrel{\text{def}}{=} \forall p \in \mathrm{pos}(t).\forall q \in \mathrm{pos} \begin{cases} (p \lhd 0) +\!\!+ q \in \mathrm{pos}(t) \Rightarrow t.((p \lhd 0) +\!\!+ q) < t.p \\ (p \lhd 1) +\!\!+ q \in \mathrm{pos}(t) \Rightarrow t.p < t.((p \lhd 1) +\!\!+ q) \end{cases}$$

En un ABB la operación principal es la de búsqueda (llamada "está" en la especificación del TAD diccionario), que en estilo funcional podría definirse:

```
search : T × <T>→Bool                                   {se aplica a un ABB}
search(e,<>) = false
search(e,<l,e',r>) = if e<e' → search(e,l)
                        e=e' → true
                        e>e' → search(e,r)
                     fi
```

Si el árbol está balanceado esta operación es del orden de $\log n$ donde n es el número de nodos del árbol. Para convencerse de ello, observe que la búsqueda se realiza a lo largo de un camino que parte desde la raíz. Un árbol perfectamente balanceado de altura $m-1$ tiene $n = 2^m - 1$ nodos. Sus caminos tienen a lo sumo longitud m (o sea, aproximadamente $\log n$).

Otra operación importante es la de insertar (llamada "agregar" en la especificación del TAD diccionario) un nuevo elemento en el lugar correspondiente de manera de que el resultado siga siendo un ABB.

```
insert : T × <T>→<T>                                    {se aplica a un ABB}
insert(e,<>) = <e>
insert(e,<l,e',r>) = if e<e' → <insert(e,l),e',r>
                        e=e' → <l,e',r>
                        e>e' → <l,e',insert(e,r)>
                     fi
```

Si el árbol está balanceado esta operación también es logarítmica.

Finalmente definimos la operación borrar que es la más difícil de definir, pues requiere la definición de una función auxiliar que calcule el máximo de un ABB, y otra que lo borre:

```
max : <T>→ T                                            {se aplica a un ABB no vacío}
max(<l,e,r>) = if r=<> → e
                  r≠<> → max(r)
               fi

delete_max : <T>→<T>                                    {se aplica a un ABB no vacío}
delete_max(<l,e,r>) = if r=<> → l
                         r≠<> → <l,e,delete_max(r)>
                      fi
```

delete : T × <T>→<T> {se aplica a un ABB}
delete(e,<>) = <>
delete(e,<l,e',r>) = **if** e<e' → <delete(e,l),e',r>
e=e' ∧ l =<> → r
e=e' ∧ l ≠<> → <delete_max(l),max(l),r>
e>e' → <l,e',delete(e,r)>
fi

Todas estas operaciones también son logarítmicas si el árbol está balanceado.

TAD cola de prioridades y heaps. Ya se ha presentado el TAD cola de prioridades: una variante del TAD cola en el que el criterio de atención se basa exclusivamente en la prioridad del elemento y no en el orden en que llega. Por simplicidad, se considera además que cada elemento "es" un valor, es decir, una prioridad.

La cola de prioridades se puede implementar de cualquiera de las formas vistas para cola, salvo que, o bien enqueue, o bien dequeue y first serán lineales en el tamaño de la cola. A continuación veremos que nuevamente los árboles nos proporcionan, una solución logarítimica.

Al igual que el ABB, el heap es un árbol binario con ciertas condiciones suplementarias respecto a la organización de la información. Las implementaciones del heap proporcionan implementaciones eficaces de las colas de prioridades.

Dado T equipado con un orden total $\leq$, un heap es un árbol binario tal que todos sus nodos alojan el máximo de todo el subárbol que comienza en ese nodo. En símbolos

$$\text{heap}(t) \stackrel{\text{def}}{=} \forall p \in \text{pos}(t).\forall q \in \text{pos}.\ p +\!\!+ q \in \text{pos}(t) \Rightarrow t.(p +\!\!+ q) \leq t.p$$

o equivalentemente, por transitividad de $\leq$,

$$\text{heap}(t) \stackrel{\text{def}}{=} \forall p \in \text{pos}(t).\ \begin{cases} p \lhd 0 \in \text{pos}(t) \Rightarrow t.(p \lhd 0) \leq t.p \\ p \lhd 1 \in \text{pos}(t) \Rightarrow t.(p \lhd 1) \leq t.p \end{cases}$$

Veremos que con un heap se puede implementar una cola de prioridades de manera de que first sea constante y enqueue y dequeue logarítmicas.

Una implementación muy eficiente de heaps es mediante el uso de arreglos. Dado un arreglo a: **array**[1..n] **of** T, el arreglo puede ser visto como un árbol binario en el que la celda 1 tiene como hijos a las celdas 2 y 3, la 2 tiene como hijos a 4 y 5, la 3, a 6 y 7, en general, la celda n tienen como hijos a las celdas 2n y 2n+1. Como el arreglo es finito, una celda puede alcanzar a tener 2 hijos, sólo uno, o ninguno.

type heap = **tuple**
elems: **array**[1..n] **of** elem
size: **nat**
end

Las dos funciones que siguen se utilizan para poder luego abstraernos de la operación aritmética que permite obtener las posiciones del hijo izquierdo y del hijo derecho a partir de la del padre.

```
fun left(i:nat) ret j:nat
    j:= 2*i
end

fun right(i:nat) ret j:nat
    j:= 2*i+1
end
```

Análogamente, la siguiente función determina la posición j donde se encuentra el padre de un nodo que está en posición i.

```
fun father(i:nat) ret j:nat
    j:= i ÷ 2
end
```

En un heap, un nodo puede no tener hijos (cuando es una hoja). La siguiente función dice si el nodo que se encuentra en la posición i del heap tiene hijos o no.

```
{Pre: 1 ≤i≤ h.size}
fun has_children(h:heap, i:nat) ret b:bool
    b:= (left(i) ≤ h.size)
end
{Post: b = i tiene hijos en h}
```

Similarmente, un nodo puede no tener padre (cuando es la raíz).

```
fun has_father(i:nat) ret b:bool
    b:= (i ≠ 1)
end
```

Si un nodo tiene hijos, es conveniente poder compararlo (e intercambiarlo) con el mayor de sus hijos. Para ello, la siguiente función devuelve en j la posición donde se encuentra el mayor de sus hijos. La función asume que el nodo que se encuentra en la posición i tiene hijos y distingue el caso en que tiene ambos hijos (right(i) ≤ h.size) o sólo uno.

```
{Pre: 1 ≤ i ≤ h.size ∧ has_children(h,i)}
fun max_child(h:heap, i:nat) ret j:nat
    if right(i) ≤ h.size ∧ h.elems[left(i)] ≤ h.elems[right(i)] then j:= right(i)
    else j:= left(i)
    fi
end
{Post: j = posición donde se encuentra el mayor de los hijos de i en h}
```

El procedimiento lift, asume que i es un nodo que tiene padre y modifica el arreglo intercambiando lo que se encuentra en la posición i con lo que se encuentra en el nodo padre de i.

```
{Pre: 1 ≤ i ≤ h.size ∧ has_father(i)}
proc lift(in/out h:heap,in i:nat)
      swap(h.elems,i,father(i))
end
```

A continuación, se define la función booleana must_lift que asume que i tiene padre y decide cuándo debe ejecutarse el procedimiento anterior.

```
{Pre: 1 ≤ i ≤ h.size ∧ has_father(i)}
fun must_lift(h:heap, i:nat) ret b:bool
    b:= (h.elems[i] >h.elems[father(i)])
end
{Post: b = i es mayor que su padre}
```

El procedimiento float es el que compara el dato alojado en el nodo i con los alojados en cada uno de los ancestros de i hasta encontrarle la posición correcta. Como a lo sumo recorre un camino en un árbol binario balanceado, es en el peor caso del orden de $\log n$.

```
{Pre: 1 ≤ i ≤ h.size ∧ h (= H) es heap excepto tal vez porque el elem en i es grande}
proc float(in/out h:heap,in i:nat)
    var c: nat
    c:= i
    while has_father(c) ∧ must_lift(h,c) do
        lift(h,c)
        c:= father(c)
    od
end
{Post: h es un heap con los mismos elementos que H}
```

El procedimiento sink es el que compara el dato alojado en el nodo i con los alojados en algunos de sus descendientes hasta encontrarle una posición correcta. En cada paso lo compara con el mayor de los hijos. De esta manera, a lo sumo recorre un camino en un árbol binario balanceado. Por ello, en el peor caso es del orden de $\log n$.

```
{Pre: 1 ≤ i ≤ h.size ∧ h (= H) es heap excepto tal vez porque el elem en i es chico}
proc sink(in/out h:heap, in i:nat)
    var f: nat
    f:= i
    while has_children(h,f) ∧ must_lift(h,max_child(h,f)) do
        f:= max_child(h,f)
        lift(h,f)
    od
end
{Post: h es un heap con los mismos elementos que H}
```

Estas primitivas facilitan enormemente la tarea de implementar una cola de prioridades utilizando un heap. A continuación se presentan los detalles finales para obtener dicha implementación. El tipo pqueue resulta simplemente un sinónimo del tipo heap.

```
type pqueue = heap
```

El procedimiento que inicializa la cola de prioridades no presenta novedades.

```
proc empty(out q:pqueue)
    q.size:= 0
end
{Post: q ∼ vacía}
```

El procedimiento enqueue, incrementa el campo size del heap (que al ser incrementado señala la primer celda libre del arreglo) aloja en la primer celda libre el nuevo elemento y finalmente, para restablecer la condición de heap, "hace flotar" el nuevo elemento. La única operación no constante es flotar, por ello enqueue resulta en el peor caso del orden de $\log n$.

```
{Pre: q ∼ Q ∧ q.size <n}
proc enqueue(in/out q:pqueue,in e:elem)
      q.size:= q.size+1
      q.elems[q.size]:= e
      float(q,q.size)
end
{Post: q ∼ encolar(Q,e)}
```

La función first es muy sencilla. En un heap no vacío el máximo se encuentra en la raíz, que está en la posición 1 del arreglo.

```
{Pre: q ∼ Q ∧¬is_empty(q)}
fun first(q:pqueue) ret e:elem
    e:= q.elems[1]
end
{Post: e ∼ primero(Q)}
```

El procedimiento dequeue elimina el máximo sobreescribiendo la posición 1 del arreglo con lo que se encuentra en la última posición del heap (la señalada por size), decrementa el campo size dado que ahora va a tener un elemento menos y finalmente, para restablecer la condición de heap, "hunde" el elemento que ahora está en la raíz. La única operación no constante es hundir, por ello dequeue resulta en el peor caso del orden de $\log n$.

```
{Pre: q ∼ Q ∧¬is_empty(q)}
proc dequeue(in/out q:pqueue)
      q.elems[1]:= q.elems[q.size]
      q.size:= q.size-1
      sink(q,1)
end
{Post: q ∼ dequeue(Q)}
```

Por último, la función is_empty no presenta ninguna dificultad.

```
{Pre: q ∼ Q}
fun is_empty(q:pqueue) ret b:bool
    b:= (q.size = 0)
end
{Post: b ∼ es_vacía(Q)}
```

Como se verá pronto, además de ser una representación ingeniosa de cola de prioridades el heap da lugar a un algoritmo de ordenación eficiente, llamado heapSort.