Capítulo 8

Colas de prioridad.

Se desea disponer de una estructura de datos y encontrar los algoritmos asociados que sean eficientes para **seleccionar** un elemento de un grupo.

Uno de los valores de la información agrupada en la estructura se denomina **prioridad**. Es un valor entero, y por tradición, el menor valor entero está asociado a la estructura que tiene mayor prioridad. Prioridad se entiende como sinónimo de lo más importante. Puede haber varias estructuras con igual prioridad; en este sentido no son conjuntos.

8.1. Operaciones.

Se definen dos operaciones: Insertar un elemento en el conjunto, y otra que busca y descarta (selecciona) el elemento con valor menor del campo prioridad.

Es necesario encontrar una nueva estructura, ya que las vistas anteriormente tienen limitaciones. La tabla de hash permite insertar con costo constante, pero encontrar y descartar el mínimo tiene alta complejidad.

Una lista ordenada en forma ascendente por la prioridad, permite seleccionar el mínimo con costo $O(1)$. Pero la inserción, manteniendo en orden tiene costo promedio $O(n)$; si encuentra el lugar para insertar a la primera es costo 1; pero si debe recorrer toda la lista, debe efectuar n comparaciones; en promedio debe efectuar un recorrido de $n/2$ pasos.

Una lista no ordenada, tiene costo de inserción $O(1)$, y búsqueda $O(n)$.
En ambos casos la repetición de n operaciones sobre la estructura da origen a complejidad n^2.

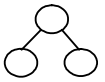
Estudiaremos usar la estructura de un árbol binario, ya que ésta garantiza que las operaciones de inserción y descarte sean de complejidad $O(log_2 (n))$, pero deberemos efectuar modificaciones ya que en una cola de prioridad se permiten claves duplicadas.

8.2. Relaciones entre el número de nodos y la altura en árboles binarios.

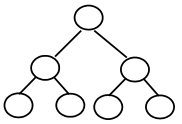
La altura es el número de nodos de una trayectoria desde la raíz hasta las hojas, incluyendo ambos.

Los siguientes diagramas ilustran árboles binarios balaceados completos, con todas las hojas en el nivel más bajo.

En un primer caso se tiene que el número de nodos *n* es tres; un nivel (*m* elementos entre los nodos en una trayectoria desde la raíz a las hojas) y altura *h* igual dos.
Con un nivel:
n=3 m=1 h= 2



Con dos niveles:
$n=7=2^3-1$ m=2 h=3



Con tres niveles:
$n=15=2^4-1$ m=3 h=4

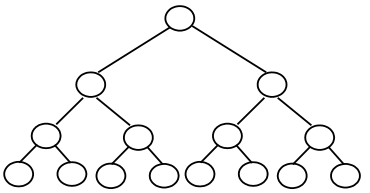


Figura 8.1. Árboles binarios completos.

En un caso general:
$n = 2^h - 1$, $h = m + 1$ y $h=\log_2 (n+1)$, despejando h de la primera relación.

La **altura**, es el concepto importante para la complejidad, ya que define el número de nodos a revisar en una trayectoria desde la raíz hasta las hojas.

Se estudian a continuación las relaciones para árboles incompletos, pero tal que las hojas que faltan se encuentran en el último nivel y siempre a la derecha de las presentes.

Se tienen para árboles de altura tres, los casos con n=4, 5, y 6.
Es decir para $n >= 2^{3-1}$ y $n < 2^3-1$

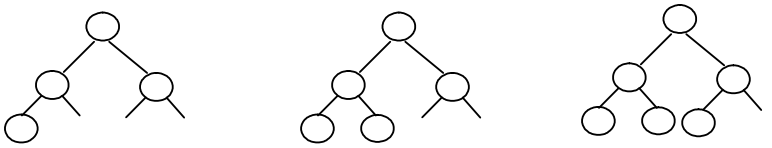


Figura 8.2. Árboles binarios incompletos.

Para árboles incompletos de altura cuatro se tienen árboles con nodos entre 8 y 9. Es decir para $n >= 2^{4-1}$ y $n < 2^4-1$.

Para árboles incompletos de altura *h* se tienen árboles con nodos en el intervalo:

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

Lo que implica: $\log_2 (n+1) < h <= \log_2 (n) + 1$

Se puede encontrar un n_0 y un c tal que $c*\log_2 (n+1)$ sea una cota superior para h.

Entonces:

$$h = \text{Parte entera}(\log_2 (n+1)) = O(\log_2 (n))$$

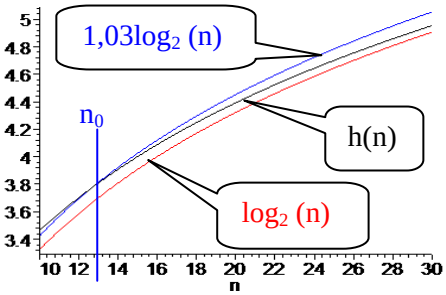


Figura 8.3. Altura árbol binario incompleto.

8.3. Características de una posible estructura.

Si escogemos un *árbol binario parcialmente completo*, lo más balanceado posible; que podemos definir como un árbol que en su nivel más bajo (cerca de las hojas) cumple con la condición de que las hojas que le faltan están a la derecha de las presentes.

Además debemos definirlo como *parcialmente ordenado*, ya que es posible tener elementos repetidos. Esta condición la logramos estableciendo, que los nodos queden ordenados según:

valor (hijo) >= valor (padre)

Estas definiciones, determinan que en la raíz se encuentra el nodo con valor mínimo; y que la posición para insertar queda definida como la hoja de menor nivel que falta.

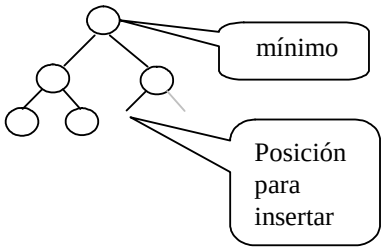


Figura 8.4. Árbol binario parcialmente ordenado.

Si el elemento a insertar tiene un valor menor que su padre, debe ascender por intercambio.
Si se descarta la raíz, para mantener la estructura, se reemplaza la posición vacante en la raíz por la hoja ubicada más a la derecha, y si ésta es mayor que los hijos se la hace descender por intercambio. Ambas operaciones tienen complejidad O(*a*), donde *a* es la altura del árbol.

Esta estructura se denomina heap. Que equivale, en español, a grupo de cosas unas al lado de otras (montón, colección).

8.4. Descripción de la estructura heap.

Se visualiza como un árbol binario, pero el valor de cualquier nodo es menor o igual a los valores de los nodos hijos. La Figura 8.5, muestra 12 valores almacenados en un heap, todos los nodos cumplen la propiedad.
Se agregan nodos primero en el sub-árbol izquierdo, y luego en el derecho, manteniendo un árbol binario lo más balanceado posible.

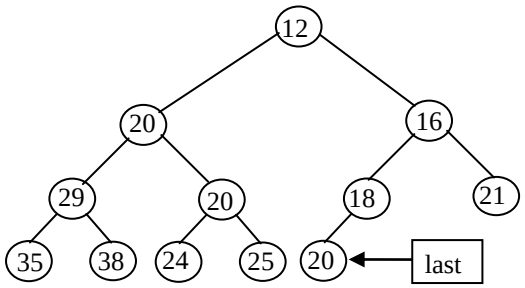


Figura 8.5. Árbol binario con relación de orden tipo heap.

De esta forma el nodo de mayor prioridad (el de menor valor) se encuentra ubicado en la raíz.

La siguiente idea, además de ser una genialidad, es el concepto fundamental del heap.

La estructura anterior puede almacenarse en un arreglo.

La clave de la raíz se almacena en el primer elemento del arreglo, con índice 1 (no se usa el 0); luego se almacenan los nodos de primer nivel (de izquierda a derecha); luego los de segundo nivel (siempre de izquierda a derecha) y así sucesivamente; el último presente debe marcarse con el cursor last:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
12	20	16	29	20	18	21	35	38	24	25	20				

last max

Figura 8.6. Arreglo con relación de orden tipo heap.

Se denomina cursor a una variable que almacena un índice de un arreglo, para diferenciarla de un puntero que almacena una dirección.

Al almacenar en un arreglo se fija el número máximo de elementos que esperan en la cola de prioridad, este valor es max en la figura 8.6. Esto es común a toda estructura estática.

De este modo:

- el hijo izquierdo del nodo *i* tiene el índice 2**i*.
- el hijo derecho del nodo *i*, si existe, tiene el índice: 2**i* + 1
- el padre del nodo *i*, tiene el índice: *i*/2
- el nodo no existe si: (*i* < 1) o (*i* > last)

La estructura anterior se denomina heap.

Si la raíz tiene índice 1, sus hijos tienen índices 2 y 3 respectivamente; si se hubiera escogido la raíz con índice 0, tendríamos que tratar en forma excepcional a los hijos de la raíz; del mismo modo esta elección permite determinar que el nodo raíz no tiene padre, ya que 1/2, en división entera resulta con valor cero.

Existen algoritmos eficientes para agregar un nodo, manteniendo la propiedad del heap; y para extraer la raíz, y reordenar los elementos de tal modo que se mantenga la propiedad de ser un heap.

8.5. Complejidad.

En el caso del heap, como veremos a continuación, la inserción tiene complejidad $\log_2(n)$; y la extracción del mínimo, manteniendo el heap, también es $\log_2(n)$.

Si se realizan *n* operaciones en un heap éstas tienen un costo: $n\log_2(n)$. Como veremos más adelante se requieren realizar *n* operaciones para formar un heap a partir de un arreglo desordenado, mostrando de este modo que el heap puede emplearse para desarrollar un eficiente algoritmo de ordenamiento.

Cuando se busca: se tiene el valor a buscar; la operación de seleccionar es escoger el nodo que tiene determinada propiedad, en este caso el de menor valor de prioridad. La selección retorna el valor buscado.

8.5.1. La operación de inserción.

Se ilustra la inserción de un nodo con valor 13, respecto a la Figura 8.5.

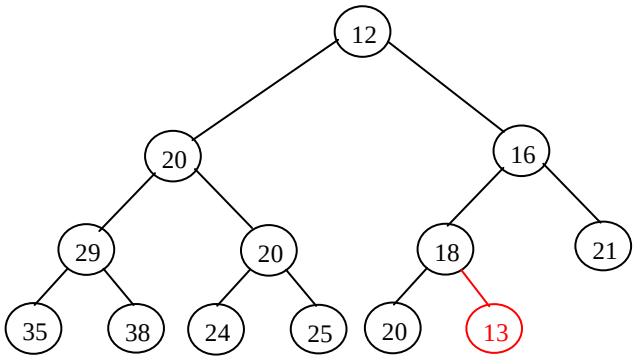


Figura 8.7. Inserción de nodo con valor 13 en un heap que tenía 12 elementos.

Se agrega al final, y luego se lo hace ascender, manteniendo la propiedad del heap: El valor de cualquier nodo es menor o igual a los valores de los nodos hijos.

El algoritmo se denomina **sift-up** (filtrar, separar, examinar).
El número de comparaciones es el del número de nodos desde una hoja hasta la raíz en un árbol binario, lo más balanceado posible (este número es log(*n*)).

Solo es necesario comparar el valor del nodo con el valor del padre del nodo, ya que el hermano tiene un valor mayor o igual que el del padre. Se intercambia el nodo con valor 18 y el recién insertado. Aún no se cumple la propiedad de ser un heap.

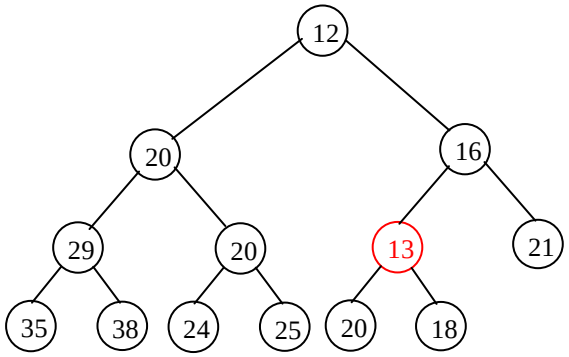


Figura 8.8. Ascenso del nodo con valor 13 por intercambio con el padre.

Debe continuarse la revisión, en forma ascendente hacia la raíz, para que se mantenga la propiedad del heap. Y debe intercambiarse el nodo de valor 13, con su padre, el nodo 16.

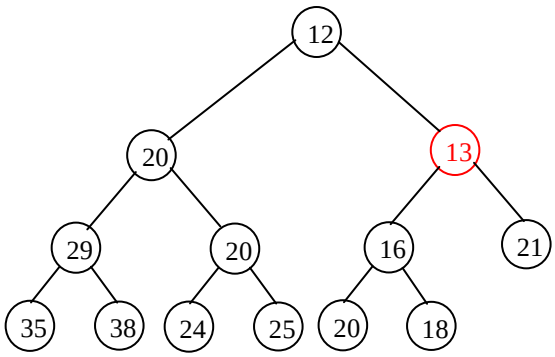


Figura 8.9. Sigue ascendiendo el nodo con valor 13. Queda un heap de 13 elementos.

Comparando con la raíz, se advierte que el recién insertado no debe intercambiarse. Luego de estas operaciones, se mantiene la propiedad del heap.

8.5.2. Declaraciones de estructuras de datos y variables.

Para describir el algoritmo en lenguaje C, es preciso efectuar algunas definiciones. Definiremos un tipo de datos denominado registro, que contenga el valor de prioridad y agregaremos otro valor (ntarea) para ejemplificar que dicho registro contiene la información asociada a una tarea. También se define el tipo puntero a registro con el nombre: preg.

```
typedef struct nn1{
      int prioridad;
      int ntarea;
} registro, *preg;
```

Luego definiremos un tipo de datos denominado heap, como un arreglo de registros de 10 elementos (el elemento r[0] puede emplearse como registro temporal, ya que en C, los arreglos parten desde cero, y en este caso nos interesa usar desde el 1 en adelante), mediante:

```
#define Ntareas  10
typedef registro heap[Ntareas+1];

heap r;          /*  r es arreglo de registros  */
int last=0;      /*  Apunta al último           */
```

Definiremos un registro, al cual se apunta con nuevo, para mostrar la inserción de un nuevo elemento al heap.

```
registro Tarea;          /*se asume un registro iniciado con valores  */
preg nuevo=&Tarea;
```

8.5.3. Inserción.

La función insertar se implementa en términos de la función siftup, descrita en 8.5.1.

```
//inserta en posición n-ésima y lo hace ascender
void siftup(int n)
{   int i, j;
    registro temp;
      for (j=n; j>1; j=i){          /*Al inicio j apunta al último ( n ) */
        i=(j>>1) ;                  /* i= j/2  Con i el cursor del padre de j*/
        if ( r[i].prioridad <= r[j].prioridad ) break;  // Sale del for si es un heap.
        /* invariante: heap(1, n) excepto entre j y su padre i */
        temp=r[j], r[j]= r[i], r[i]=temp; /* intercambio sólo si el hijo es menor que el padre*/
         /* La condición de reinicio hace que j apunte a su padre. j=i; */
      }
}
```

Se lo inserta en la última posición y se lo hace ascender en log(*last*) comparaciones.

```
void insert(preg nuevo)
{         last++;
          r[last]=*nuevo;
          /*  Antes se tenía un heap(1, last-1) */
          siftup(last);
          /* Luego de sift-up se tiene un heap(1, last) */
}
```

8.5.4. Extracción del mínimo.

Para obtener el elemento de mayor prioridad, basta tomar la raíz.
Una vez extraída la raíz la precondición es: heap(2, *n*) && *n* >= 0. Es decir los elementos desde el 2 hasta el *n*, están en la relación de orden de un heap.

El algoritmo consiste en colocar el elemento apuntado por last en el lugar de la raíz, y hacer descender este nodo, de tal modo de mantener la propiedad de un heap. Al final la postcondición es: heap(1, *n*).

A partir de la Figura 8.5., se tiene, después de sacar el nodo con valor 12, y mover el último al lugar de la raíz y decrementar last:

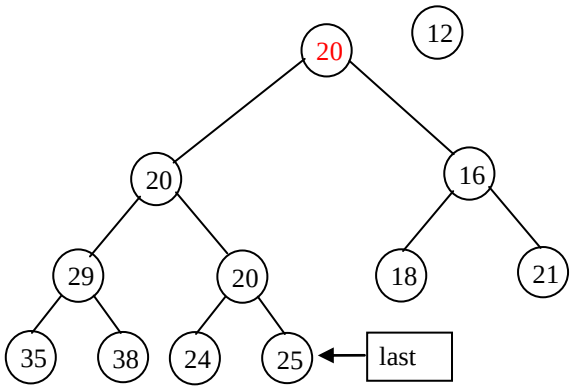


Figura 8.10. Se extrajo la raíz, se movió el último a la raíz, se cambia last.

Se busca el hijo menor de la raíz y se lo intercambia con el padre, si el hijo tiene un valor menor que el padre.

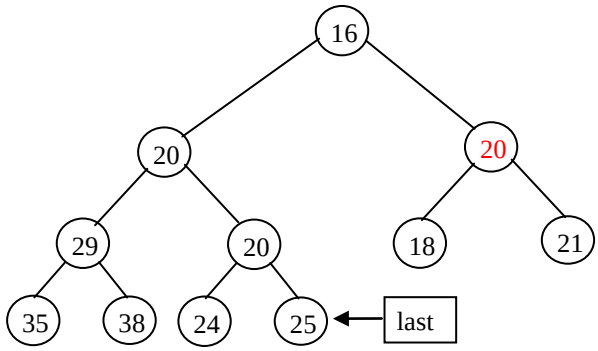


Figura 8.11. Descenso del más pesado. Sift-down.

Es necesario comparar la raíz con los dos hijos, esto en el caso de que ambos existan, y efectuar el intercambio con el hijo con clave menor.
Aún es necesario seguir intercambiando, para mantener propiedad de heap, en la Figura 8.11, se intercambia ahora el nodo con valor 20, con su hijo izquierdo, que tiene valor menor. El recorrido se efectúa en log (*n*) pasos.

8.5.5. Descenso en el heap.

Basada en while.

```
void siftdown(int n)
{   int i=1, j;
    registro temp;          /* al inicio i apunta a la raíz; j al hijo izquierdo  */
    while (( j=(i<<1)) <= n )                   /*   j = 2*i   */
      {    if ( ( (j+1)<=n ) && ( r[j+1].prioridad < r[j].prioridad)) j++;
           /*Si existe hijo derecho y es menor que el hijo izquierdo, j apunta al derecho */
```

```
            if (r[i].prioridad <= r[j].prioridad) break;
            temp=r[j], r[j]= r[i], r[i]=temp;   /*Intercambia si el hijo es menor que el padre  */
            i=j;                /* Nueva raíz en el descenso  */
        }
}
```

Basada en for:

```
void siftdown(int n)
{   int i, j;
    registro temp;          /* al inicio i apunta a la raíz; j al hijo izquierdo  */
    for(i=1, j=i<<1; j<=n; i=j, j=i<<1)  /*   j = 2*i   */
      {     if ( ( (j+1)<=n ) && ( r[j+1].prioridad < r[j].prioridad)) j++;
            /*Si existe hijo derecho y es menor que el hijo izquierdo, j apunta al derecho */
            if (r[i].prioridad <= r[j].prioridad) break;
            temp=r[j], r[j]= r[i], r[i]=temp; /*Intercambia si el hijo es menor que el padre  */
            /* La condición de reinicio apunta a nueva raíz en el descenso: i=j; */
      }
}
```

8.5.6. Extracción del mínimo.

Si asumimos que la raíz se copia en el registro Tarea, la extracción del mínimo se puede escribir:

```
int extraemin(void)
{
   if (last>=1)
     {  Tarea = r[1] ; /*salvar raiz*/
        printf(" %d ", Tarea.ntarea);  //Hace algo con la tarea
        r[1]=r[last--];   //Mueve el ultimo a la posición de la raíz. Decrementa last.
        siftdown(last);  //Lo hace descender, manteniendo propiedad de heap.
        return(0);
     }
  else    return(1);   /*error intento extracción en heap vacio*/
}
```

8.6. Prueba de las funciones.

El printf, en extraemin, se ha colocado para revisar el algoritmo. Va mostrando el orden en que se extraen las tareas del heap.
Las siguientes rutinas ilustran un test para las funciones. Llenecola le da valores al heap, mediante insert.

```
void llenecola(void)
{ int i;
  for(i=1; i< Ntareas+1; i++)
```

```
   {    /*se prueban con igual prioridad. Deben salir todas una vez. */
        Tarea.prioridad=10;

        /*Si se insertan con prioridad creciente. Las tareas deben salir en orden creciente*/
       //Tarea.prioridad=i;

       /*Si se prueban con prioridad decreciente. Deben salir tareas en orden decreciente*/
       //Tarea.prioridad=Ntareas+1-i;

      Tarea.ntarea=i;   /*se escribe en registro Tarea, que es apuntado por nuevo*/
       insert(nuevo);
   }
}
```

La función Sch es un itinerador, que extrae uno a uno los elementos; cuando el heap queda vacío se lo vuelve a llenar, y se avanza a la siguiente línea.

```
void Sch(void)
{
   if ( extraemin()!= 0)    { putchar('\n'); llenecola();}
}
```

El itinerador siempre está corriendo.

```
void main(void)
{
    for(;;)  { Sch(); }
 }
```

Probar las funciones requiere un diseño cuidadoso, para asegurar que éstas cumplan las especificaciones. El ejemplo anterior, sirve de inicio para diferentes pruebas a realizar. Por ejemplo, qué pasa si se intenta agregar más tareas de las que puede soportar el tamaño definido para el heap. Verificar que efectivamente siempre seleccionará al menor, requiere iniciar las prioridades con valores diferentes.

8.7. Resumen:

La operación de descartar el mínimo consiste en:
Sacar la raíz, lo cual corta el árbol.
La hoja más derechista, de menor nivel, se coloca en el lugar de la raíz.
Se la hace descender, por intercambio con los hijos, desde la raíz, para mantener la propiedad del heap.

La complejidad queda dada por altura del árbol, que es la trayectoria más larga desde la raíz hasta las hojas, y es: $O(\log_2(n))$

En inserción:

Se coloca el nuevo elemento en el menor nivel, lo más a la izquierda posible.
Se la hace ascender, por intercambio con el padre, manteniendo la propiedad de heap.
La complejidad queda dada por altura del árbol, que es la trayectoria más larga desde la hoja, recién insertada, hasta la raíz, y es: $O(\log_2(n))$

Se puede cambiar la relación de orden, de tal modo que el valor numérico del padre sea mayor que el de los hijos. En este caso se hacen descender a los más livianos, y ascender a los más pesados.

Referencias.

Robert W. Floyd. "Algorithm 113: Treesort," *Communications of the ACM* , Volume 5 pág. 434. 1962.
J.W.J. Williams. Algorithm 232 (Heapsort). *Communications of the ACM*, Volume 7, pp.347-348, 1964.

Índice general.

Índice de figuras.