Estructuras de Datos
Árboles

Cola de Prioridad

Una Cola de prioridad es una variante del tipo de datos Cola que mencionamos anteriormente. La característica que lo distingue es que los datos poseen una clave perteneciente a un conjunto ordenado.

Cola de Prioridad

Las funciones que posee permiten

- *insertar* nuevos elementos;
- *extraer el máximo* (o el mínimo, en caso de que la estructura se organice con un criterio de orden inverso).

Frecuentemente los valores de las claves se ven como prioridades, con lo cual la estructura permite insertar elementos de cualquier prioridad, y extraer el de mayor prioridad.

Cola de Prioridad

La *cola de prioridad* puede ser implementada con:

- Una lista ordenada:
  - Inserción: O(n) para poder insertar tenemos que recorrer la lista, en orden secuencial, nodo a nodo, hasta encontrar el lugar que le corresponda.
  - Extracción de máximo: O(1) estaría en el primer lugar de la lista al cual accedemos en forma directa.
- Una lista desordenada:
  - Inserción: O(1) insertamos el dato en cualquier lugar.
  - Extracción de máximo: O(n) debemos hallar el máximo en una lista desordenada, esto implica recorrer toda la lista.

Cola de Prioridad

¿Podemos usar arrays para implementar una Cola de prioridad? Analicemos un poco esto.
Supongamos que usamos un array ordenado.

Si queremos insertar un dato tenemos que recorrer el array, esto lo podemos hacer usando búsqueda binaria para posicionarnos en el lugar adecuado pero después, tendríamos que desplazar todos los datos siguientes para poder mantener el orden de prioridades. Por lo tanto, si bien, encontraríamos más rápido la posición donde insertar (O(log n)) tendríamos que desplazar todos los elementos restantes del arreglo.

Debido a esto es que no es una buena opción pensar en un array ordenado.

Cola de Prioridad

¿Podemos usar arrays para implementar una Cola de prioridad? Supongamos que usamos un array sin orden.

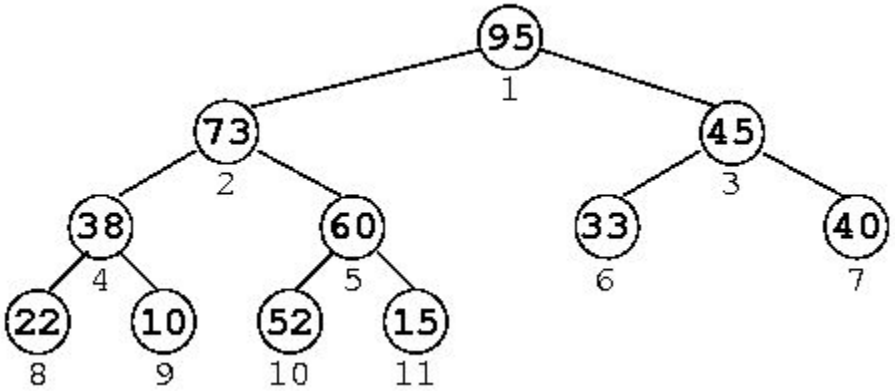
En este caso:

- Inserción: O(1) insertamos el dato en cualquier lugar.
- Extracción de máximo: O(n) debemos hallar el máximo en un array desordenado, esto implica recorrer todo el array.

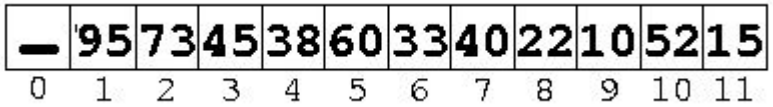
Heap

Un Heap es un árbol binario completo, que permite su almacenamiento en un arreglo sin usar punteros. ¿Cómo se puede hacer esto? Lo veremos a continuación pero, recordemos que un árbol binario completo tiene todos sus niveles llenos, excepto posiblemente el de más abajo, y en este último los nodos están lo más a la izquierda posible.

Supongamos que tenemos el siguiente árbol binario completo:



La numeración por niveles (indicada bajo cada nodo) son las posiciones en donde cada elemento sería almacenado en el arreglo. Dejaremos la primera posición (índice 0) sin usar, para simplificar algunos cálculos que veremos a continuación. En el caso del ejemplo dado, el arreglo sería:



Heap

La característica que permite que un Heap se pueda almacenar sin punteros es que, si se utiliza la numeración por niveles indicada, entonces la relación entre padres e hijos es:

$$\text{Hijos del nodo } j = \{2*j,\ 2*j+1\}$$
$$\text{Padre del nodo } k = \text{floor}(k/2)$$

Heap

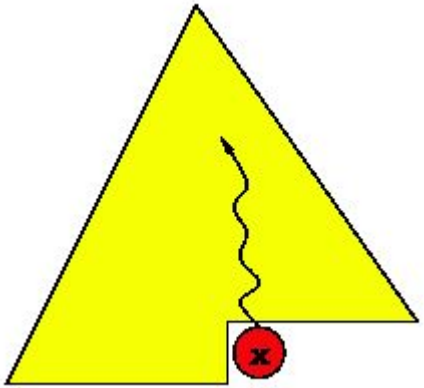
Un Heap puede utilizarse para implementar una Cola de prioridad almacenando los datos de modo que las claves estén siempre ordenadas de arriba a abajo (a diferencia de un árbol de búsqueda binaria, que ordena sus claves de izquierda a derecha). En otras palabras, el padre debe tener siempre mayor prioridad que sus hijos (ver el slide 8).

Inserción en un Heap

La inserción se realiza agregando el nuevo elemento en la primera posición libre del heap, esto es, el próximo nodo que debería aparecer en el recorrido por niveles o, equivalentemente, un casillero que se agrega al final del arreglo.

Inserción en un Heap

Después de agregar este elemento, la *estructura* del Heap se preserva, pero la restricción de orden no tiene por qué cumplirse. Para resolver este problema, si el nuevo elemento es mayor que su padre (recordemos que, por la propiedad del Heap , el padre es mayor que el otro hijo), se intercambia con él, y ese proceso se repite mientras sea necesario. Una forma de describir esto es diciendo que el nuevo elemento "trepa" en el árbol hasta alcanzar el nivel correcto según su prioridad.



Inserción en un Heap

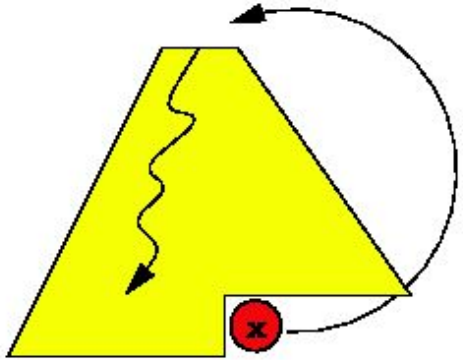
El siguiente bloque de código muestra el proceso de inserción de un nuevo elemento x:

```
//n representa la cantidad de elementos del heap
a[++n]=x;
for(j=n; j>1 && a[j]>a[j/2]; j/=2)
  { //intercambiamos con el padre
    t=a[j];
    a[j]=a[j/2];
    a[j/2]=t;
  }
```

El proceso de inserción, en el peor caso, toma un tiempo proporcional a la altura del árbol, esto es, O(log n).

Extracción del máximo en un Heap

El máximo evidentemente está en la raíz del árbol (casillero 1 del arreglo). Al sacarlo de ahí, podemos imaginar que ese lugar queda vacante. Para llenarlo, tomamos al *último* elemento del Heap y lo trasladamos al lugar vacante. En caso de que no esté bien ahí de acuerdo a su prioridad (¡que es lo más probable!), lo hacemos descender intercambiándolo siempre con el mayor de sus hijos. Decimos que este elemento "se hunde" hasta su nivel de prioridad.



Extracción del máximo en un Heap

El siguiente fragmento de programa implementa este algoritmo:

```
int esMayor =1;
m=a[1];  //La variable m lleva el máximo
a[1]=a[n--];  //Movemos el último a la raíz y achicamos el heap
j=1;
while(2*j<=n && esMayor) //mientras tenga algún hijo
{
    k=2*j; //el hijo izquierdo
    if(k+1<=n && a[k+1]>a[k])
        k=k+1;  //el hijo derecho es el mayor
    if(a[j]>a[k])
        esMayor = 0;  //es mayor que ambos hijos
    else {
        t=a[j];
        a[j]=a[k];
        a[k]=t;
        j=k;   //lo intercambiamos con el mayor hijo
    }
}
```

Este algoritmo también demora un tiempo proporcional a la altura del árbol en el peor caso, esto es, O(log n).