FILAS DE PRIORIDADE e HEAPS

Já estudamos a estrutura fila que fornece os elementos segundo um critério FIFO, da fila remove-se o elemento mais antigo. Acontece em algumas aplicações que necessitamos de remover da fila o elemento com maior prioridade , uma nova versão de fila, a que chamamos **fila de prioridade**. As operações de acesso e manipulação desta nova estrutura de informação são duas : **inserir** um elemento na fila e **extrair mínimo** (ou seja o valor com máxima prioridade).

Na maioria das aplicações, os elementos numa fila de prioridade são um par, chave-valor, em que a chave especifica o nível de prioridade. Por exemplo, num sistema operativo, cada tarefa tem um descritor de tarefa e um nível de prioridade. São ainda muito utilizadas na área de Simulação e em algoritmos de ordenação.

Há diferentes formas de fazer a implementação de filas de prioridade , por exemplo:

- Vector ordenado segundo a prioridade.

Neste caso, o extrair mínimo seria de ordem 1 (o primeiro elemento do vector ou o último de acordo com a ordenação do vector).
Mas o inserir obrigava a uma pesquisa (poder-se-á fazer uma pesquisa binária sendo portanto de ordem log n) e possivelmente a deslocamento dos elementos no vector (poderá ser no máximo de ordem n).

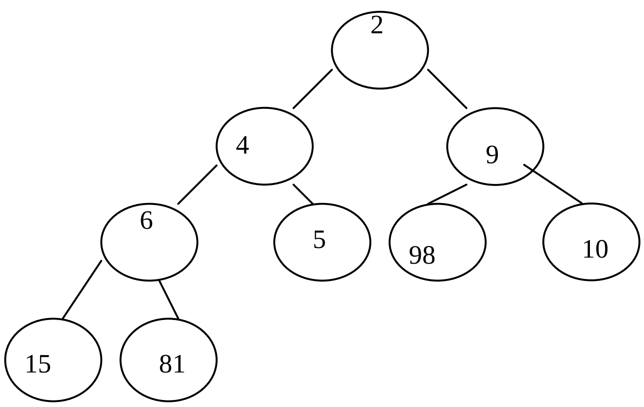
- Lista ligada ordenada

O extrair mínimo seria de ordem 1, também, deveria ser o primeiro da lista , mas o inserir um elemento embora não obrigasse ao deslocamento de elementos, simplesmente ao ajuste de apontadores, obrigaria no entanto a uma pesquisa que seria de ordem n (não é possível fazer-se uma pesquisa binária).

- Heap

Heap é uma **árvore binária completa e de prioridade.**
Uma árvore binária diz-se **completa** quando os seus níveis estão cheios, com possível excepção do último, o qual está preenchido da esquerda para a direita até um certo ponto.
Uma árvore binária diz-se de **prioridade** quando o valor de cada nó é menor ou igual que os dos seus filhos. Assim a raiz apresentará sempre o menor valor (prioridade máxima).

Exemplo de heap :



A árvore acima indicada (heap) tem uma representação muito simples e interessante através de vector. Esta representação deve-se a J. Williams.

Assim a raiz ocupará o primeiro elemento do vector (no exemplo anterior v[1]=2) e os seus filhos ocupam os indices ;

2* o índice do pai (índice 2) ---- v[2]=4

e

2 *índice do pai + 1 (índice 3) ---- v[3]=9

e por sua vez, os filhos de v[2] ocuparão as posições 4 e 5

v[4]=6

v[5]=5

e os de v[3] ocuparão as posições 6 e 7 . No exemplo só existe o filho esquerdo

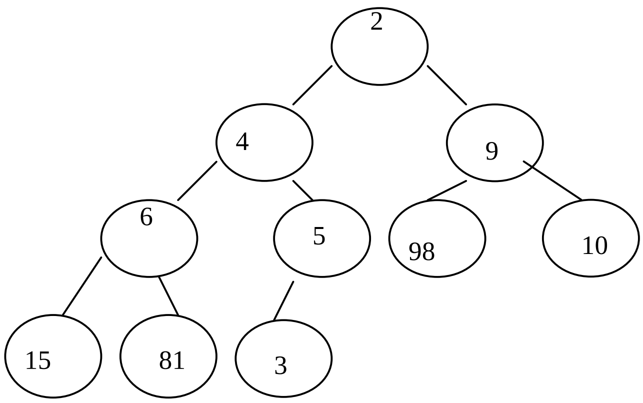
v[6]=98.

O vector teria então a seguinte configuração:

V[ ]	2	4	9	6	5	98	10	15	81		
	1	2	3	4	5	6	7	8	9	....	....

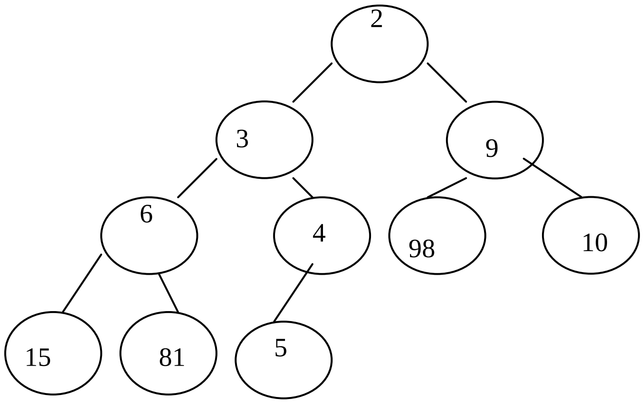
Com esta configuração é fácil obter o pai e os filhos desse nó. Por exemplo, o valor 6 tem índice 4, logo o seu pai tem índice 2 (4 / 2 = 2) e os filhos índices 8 (2 * 4) e 9 (2 * 4 + 1) respectivamente.

Vejamos agora o que acontece quando inserimos um elemento. Seja o elemento com prioridade 3. Para a árvore ser completa terá que ser colocado à esquerda do 5, ficando a árvore com o aspecto abaixo indicado.



Mas assim não é de prioridade, há que torná-la novamente de prioridade. Isto consegue-se comparando o valor com o nó pai e caso o pai seja maior do que ele trocam os valores. As comparações prosseguem até encontrar o pai com valor inferior ou igual.

No exemplo apresentado compara-se 3 com 5, como o pai é maior troca e depois compara-se 3 com 4 (pai) e troca e finalmente compara-se 3 com 2 , como o pai é menor terminam as comparações ficando a árvore final com o seguinte aspecto:



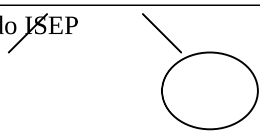
Como se vê fazemos subir o valor inserido até à posição certa. Nesta progressão que quando muito termina na raiz da árvore o número de comparações feita foi o mesmo que a altura da árvore ou seja da ordem de log n.

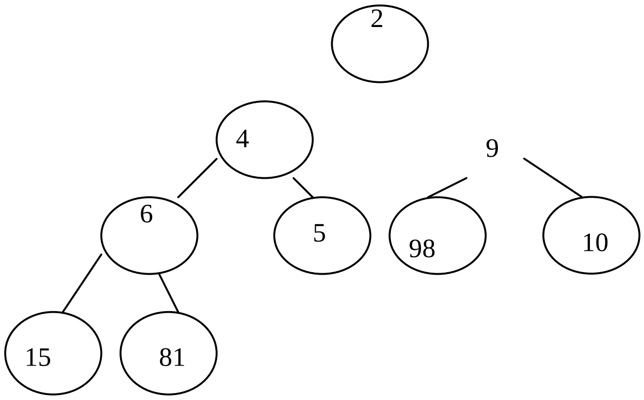
Na representação através de vector traduz-se em colocar o novo valor(3) no fim do vector, seja no índice i, e depois comparar com o elemento de índice i / 2, que é o índice do elemento pai e trocar no caso deste ser maior e assim sucessivamente, até encontrar um pai menor ou atingir a raiz.
Abaixo vamos indicar o algoritmo, supondo que temos um vector v[ ], que até ao momento tem n elementos e vamos inserir um valor x.

Algoritmo inserir (x)

```
Inicio
n=n+1
v[n]=x
i=n
Enquanto i>1            // raiz  i=1
          j = i / 2
          Se x < v[j]
                 Então  t=v[i]
                        v[i]=v[j]
                        v[j]=t
                        i=j
                 Senão  i=1            // terminar o ciclo
          Fse
Fenquanto
Fim inserir
```

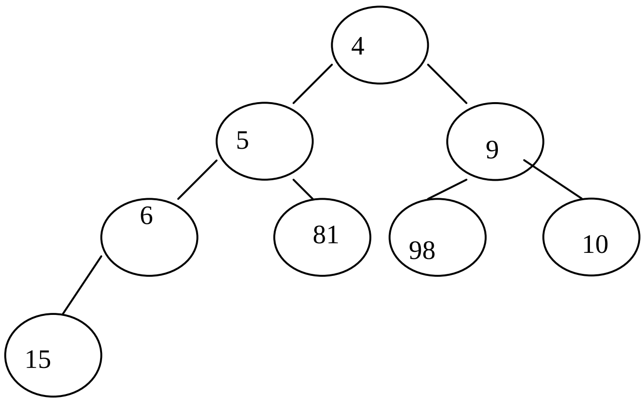
Vejamos agora, o que se passa na operação de extrair mínimo.
Consideremos a primeira heap, o extrair mínimo significa devolver o valor 2, que é a raiz da árvore. No entanto para o eliminar recorro à seguinte técnica : Substituo o conteúdo da raiz pelo valor do último nível que se encontra mais à direita.





Neste nosso exemplo, a raiz ficará com o valor 81, elimino o nó folha 81, e seguidamente vou afundar o valor que se encontra na raiz até à posição correcta. Assim compara-se o nó raiz com os filhos. Determina-se o menor dos filhos e se o pai for maior trocam-se os valores e assim sucessivamente até não Ter mais elementos ou atingir uma posição em que o pai é menor do que qualquer dos filhos.

No final da operação a árvore do exemplo terá o seguinte aspecto:



Tem portanto menos um elemento e agora a raiz é o valor mínimo 4.

Algoritmo extrair_mínimo()

```
Inicio
raiz=v[1]      //guarda-se o valor da raiz para o devolver no final
v[1]=v[n]      //o último elemento coloca-se na raiz
i=1
Enquanto 2*i <= n
        j1=2 * i
        j2=2 + i + 1 // j1 e j2 são os filhos do nó i
        k=j1
        Se  i< n
              Então  Se v[j2] < v[j1]
                              Então k= j2     // k é o índice do menor dos filhos
                     Fse
```

```
        Fse
        Se v[k] < v[i]
              Então  t=v[k]
                     v[k]=v[i]
                     v[i]=t
                     i=k
              Senão  i=n     // para terminar o ciclo
        Fse
Fenqunto
Devolve raiz
Fim extrair_mínimo
```

Tal como acontece no algoritmo de inserir, o de extrair-mínimo, tem também uma complexidade da ordem de log n, uma vez que no máximo, o número de comparações realizadas, são as mesmas que a altura da árvore, traduz o afundamento do elemento que foi substituir o valor na raiz.

Baseados nestes dois algoritmos surge um método de ordenação de vectores conhecido por HeapSort. É simples e extremamente eficiente. Consiste em construir um vector como um heap, através do método inserir e seguidamente fazer sucessivas extracções de mínimo, assim obteremos os valores por ordem crescente.

Algoritmo HeapSort

```
Inicio
Para i=1 até n
        ler(x)
        inserir(x)      //inserir no heap
FimPara

Para i=1 até n
        escreve (extrair_minimo())
FimPara
Fim HeapSort
```

Quanto à complexidade temporal deste algoritmo será a soma de duas parcelas, uma em que executa n vezes o inserir no heap, logo é da ordem n*log n.
A outra parcela , também é da mesma ordem de complexidade, uma vez que executa n vezes ao operação de extrair_mínimo, logo é da ordem de n*log n.
Assim o total é da ordem de complexidade 2*n*log n ou seja, ordem n*log n.