Heapsort

Algoritmos e Estruturas de Dados
Prof. Dr. Luciano Demétrio Santos Pacífico
{luciano.pacifico@ufrpe.br}

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Conteúdo

- **Introdução**
- ***Heaps***
- ***Heapsort***
- **Listas de Prioridade**

Introdução

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Introdução

- *Hearpsort* também é um algoritmo de ordenação.

- O *Heapsort* possui complexidade de execução na ordem de O(n log n).

- No algoritmo *Heapsort*, uma estrutura auxiliar é usada para o gerenciamento da informação: os ***Heaps***.

Introdução

- A estrutura de dados *Heap* não é usada apenas pelo algoritmo de *Heapsort*, mas também em outras aplicações, como ***listas de prioridade***.

- Veremos aplicações de listas de prioridades em alguns algoritmos na disciplina:

  - Algoritmo Prim;
  - Algoritmo Dijkistra.

Introdução

- Observação: O termo *Heap* tem sido adotado por linguagens de programação, como Java e Lisp, para se referir aos elementos de armazenamento do “lixo” coletado pelos compiladores.

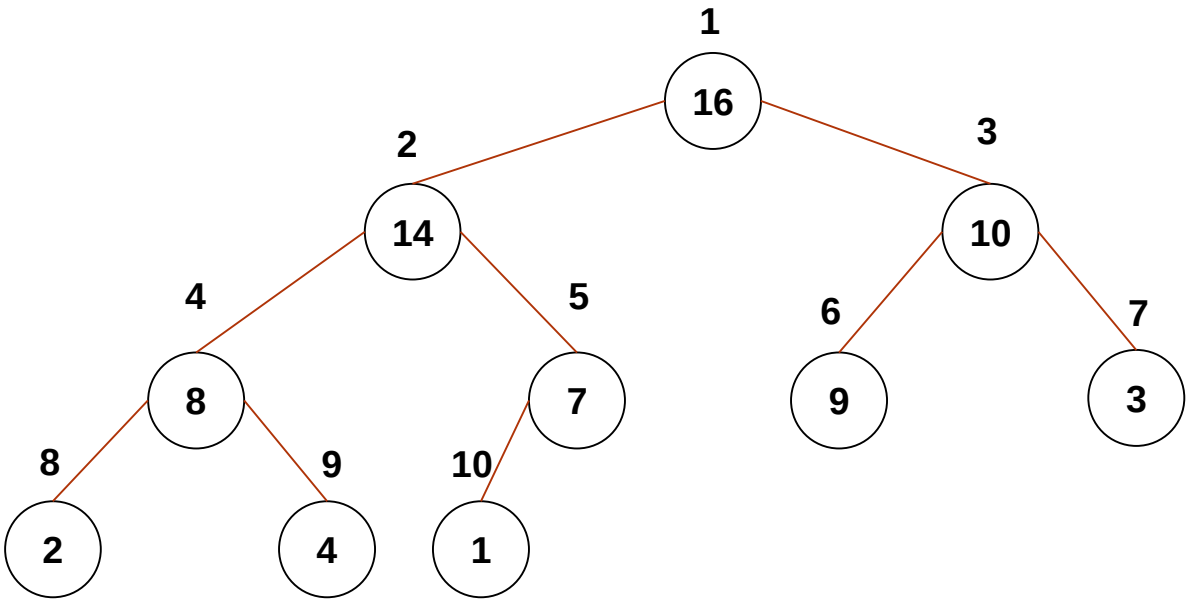
- Quando nos referirmos à palavra *Heap*, estaremos tratando da **estrutura de dados**, e não do mecanismo apresentado acima.

Heaps

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

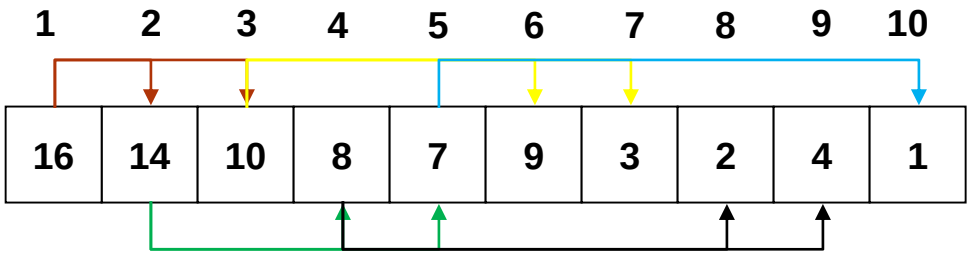
Heaps

- *Heap* é uma estrutura de prioridades, que pode ser representada na forma de árvore binária semi-completa, indicando uma ordem parcial entre seus elementos.



Heaps

- Os *Heaps* são geralmente implementados como vetores unidimensionais (**`Arrays`**), onde a raiz ocupa a posição 1, e os elementos obedecem à relação:
  - A[*i*].*esquerda* == A[2*i*];
  - A[*i*].*direita* == A[2*i* + 1].



Heaps

```
1. procedimento retornarIndicePai(i)
2.     retorne floor(i / 2)

1. procedimento retornarIndiceFilhoEsquerda(i)
2.     retorne i * 2

1. procedimento retornarIndiceFilhoDireita(i)
2.     retorne (i * 2) + 1
```

- **`floor`** é a função piso (arredondamento para baixo).

Heaps

- Existem dois tipos principais de *Heaps*.
  - *Max-Heaps*;
  - *Min-Heaps*.
- Ambos os casos, os elementos devem satisfazer a uma *propriedade do Heap*.

Heaps

- Em um *Max-Heap*, cada elemento *i* deve satisfazer a propriedade abaixo (maior elemento na raiz):

$$A[i].pai \geq A[i]$$

- Em um *Min-Heap*, cada elemento *i* deve satisfazer a propriedade abaixo (menor elemento na raiz):

$$A[i].pai \leq A[i]$$

Heaps

- Considerando um *Heap* como uma **árvore**, definimos a ***altura*** de um nó como o número de arestas no maior caminho entre o nó e uma de suas **folhas** (folhas são nós que não possuem filhos).

- A altura de um *Heap* é dada como a altura de sua **raiz** (raiz é o primeiro elemento em uma árvore).

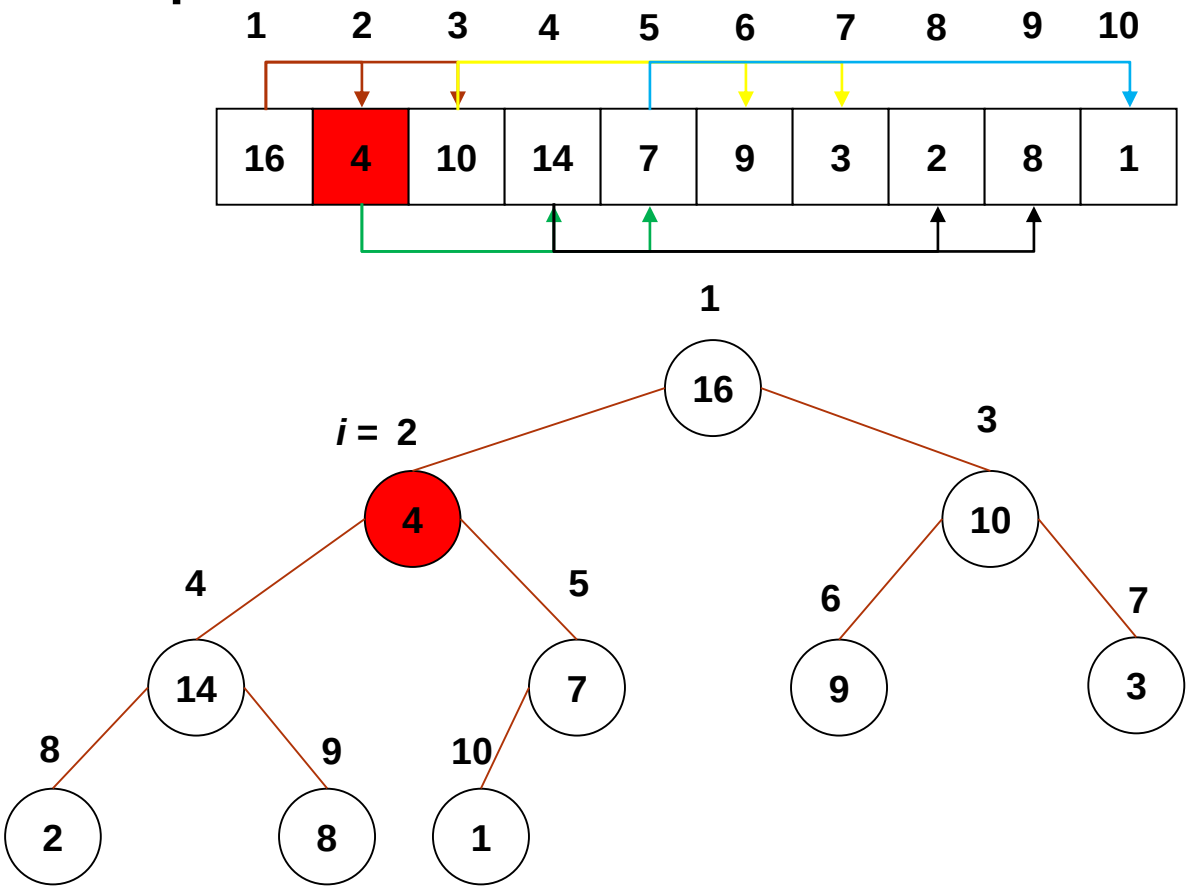
- Como um *Heap* é baseado em uma **árvore binária**, a altura do mesmo é proporcional a $\theta(\log n)$.

Heaps – Max-Heapfy

```
1.   //A -> Array<inteiro> que contém os dados
2.   //i -> índice do nó em análise
3.   //tamanhoHeap -> tamanho do Heap considerado
4.   procedimento maxHeapfy(A, i, tamanhoHeap)
5.       l = retornarIndiceFilhoEsquerda(i)
6.       r = retornarIndiceFilhoDireita(i)
7.       se (l <= tamanhoHeap) e (A[l] > A[i])
8.           maior = l
9.       senão
10.          maior = i
11.      se (r <= tamanhoHeap) e (A[r] > A[maior])
12.          maior = r
13.      se maior != i
14.          trocar(A[i], A[maior]) //intercambia o conteúdo das posições do vetor
15.          maxHeapfy(A, maior, tamanhoHeap)
```

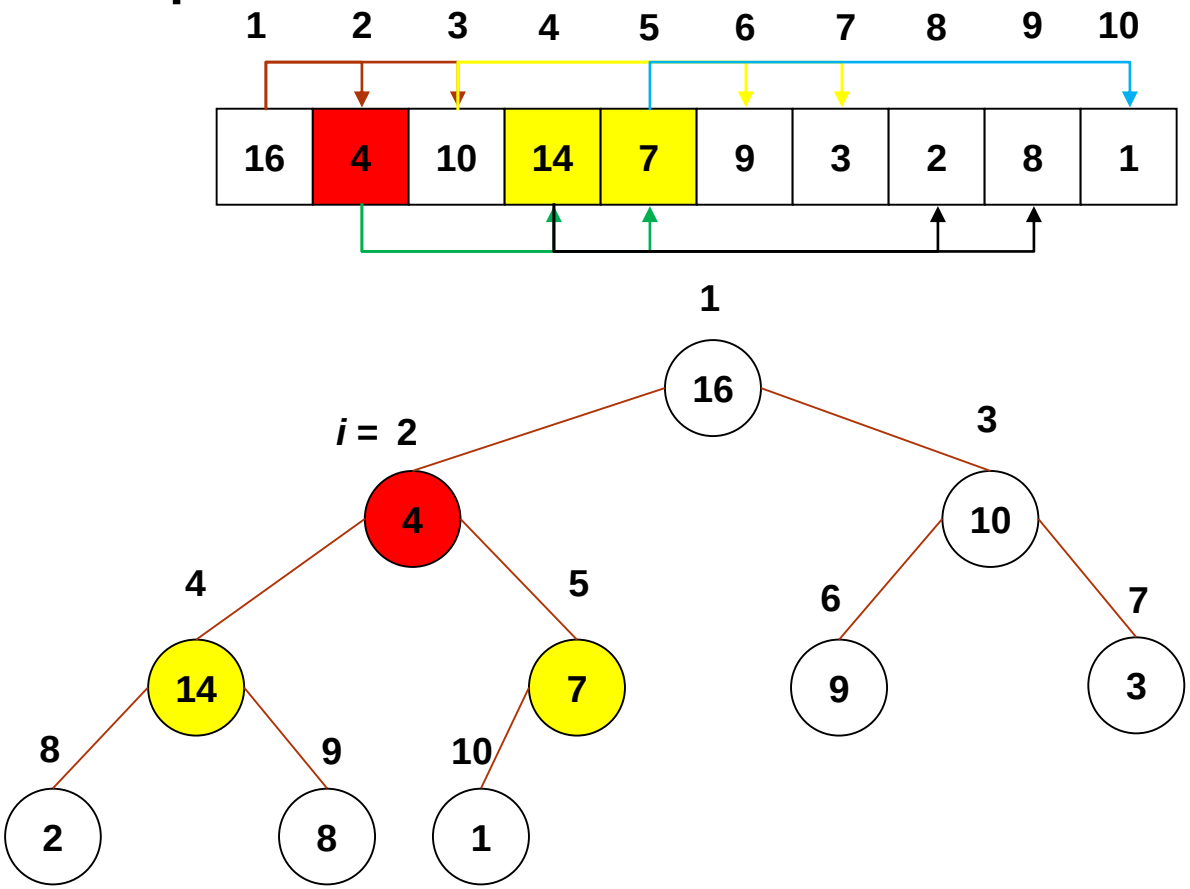
Heaps

- Exemplo:



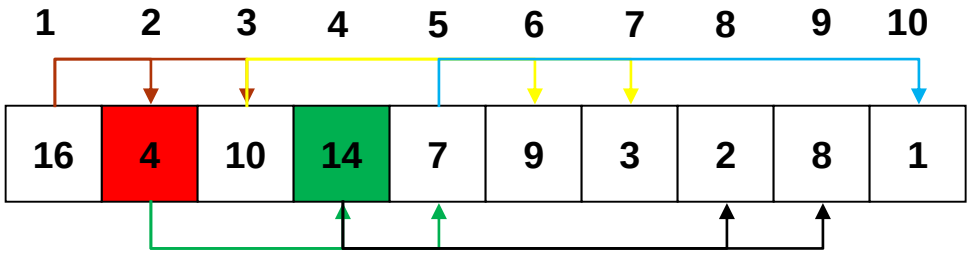
Heaps

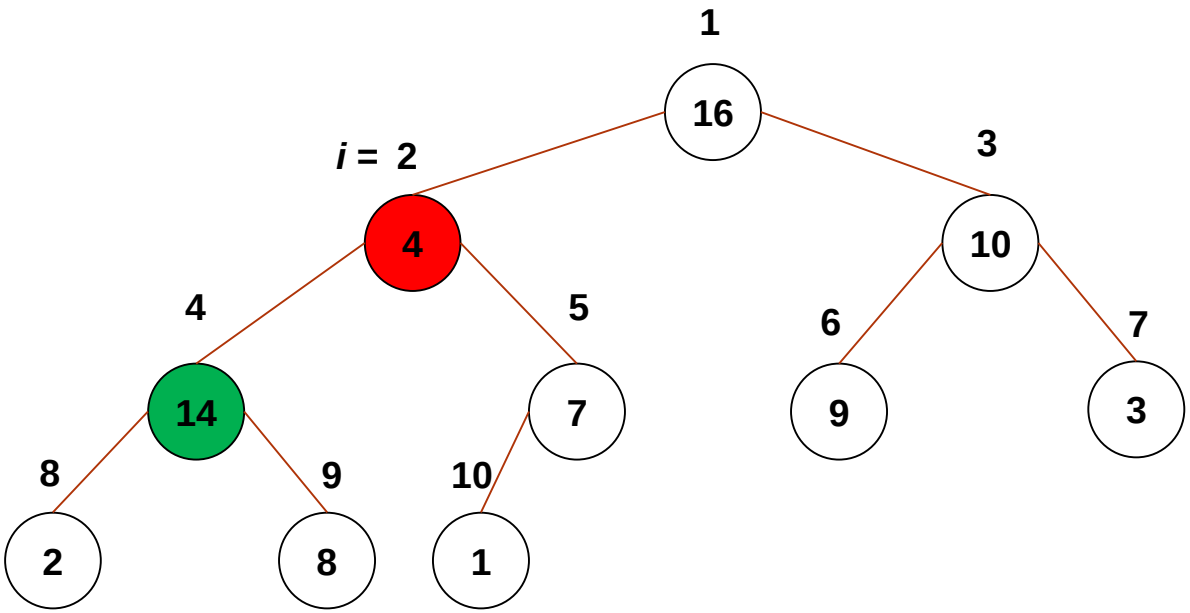
- Exemplo:



Heaps

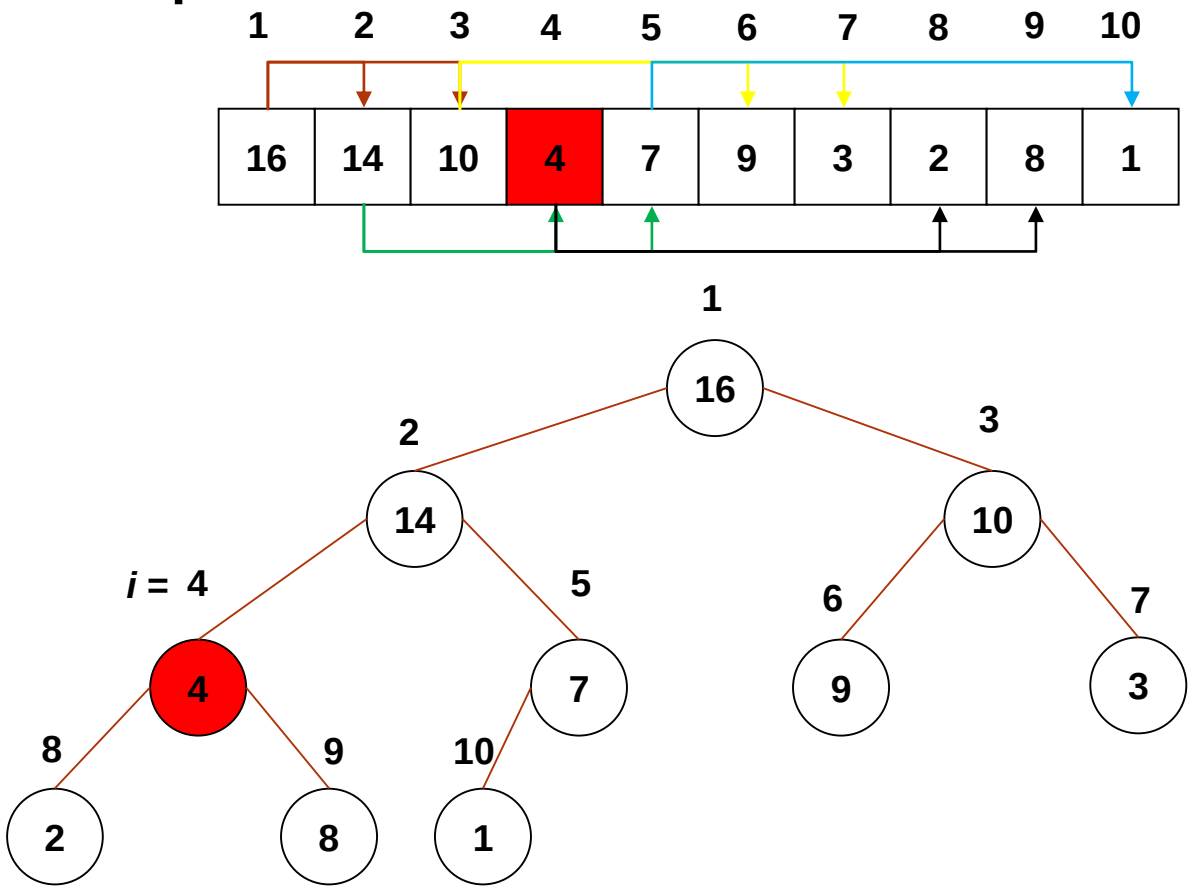
- Exemplo:





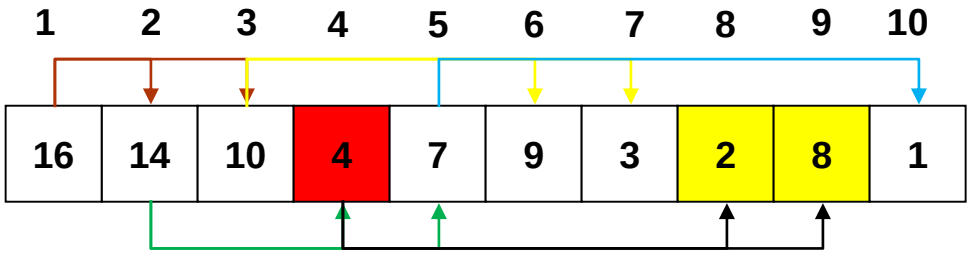
Heaps

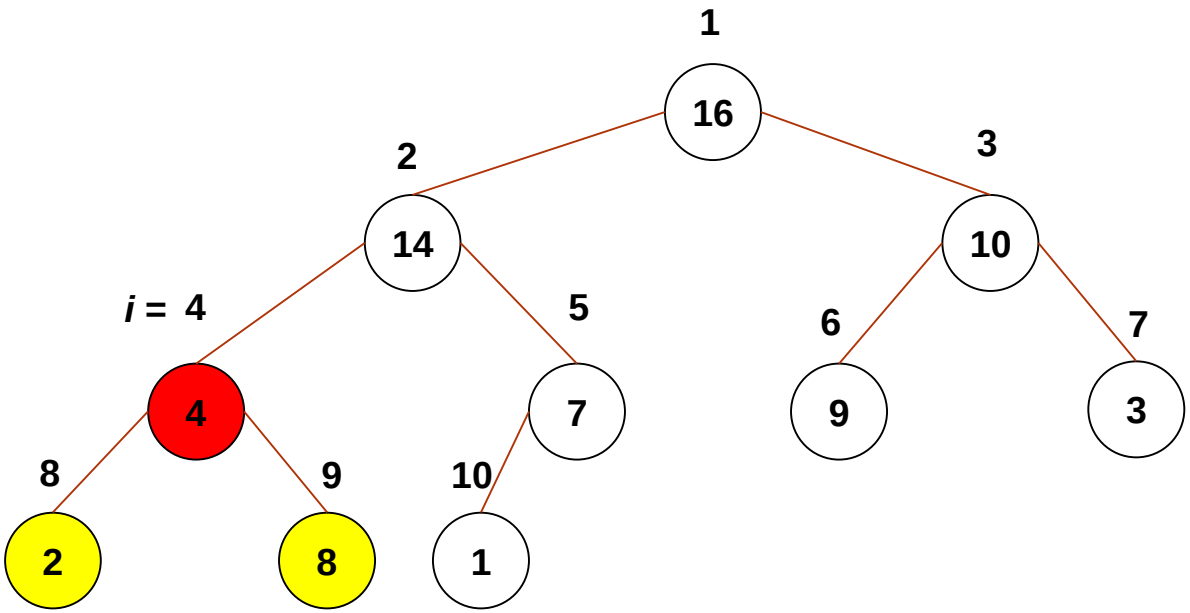
- Exemplo:



Heaps

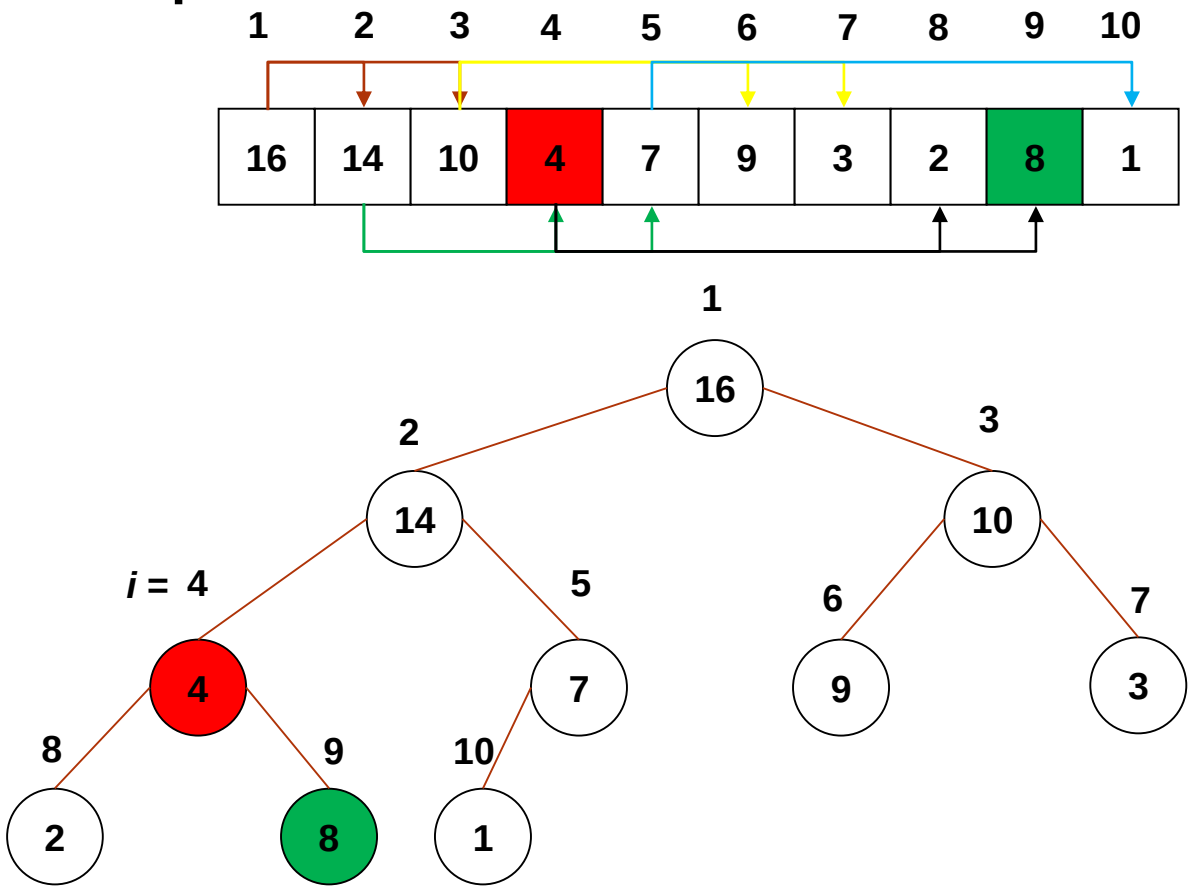
- Exemplo:





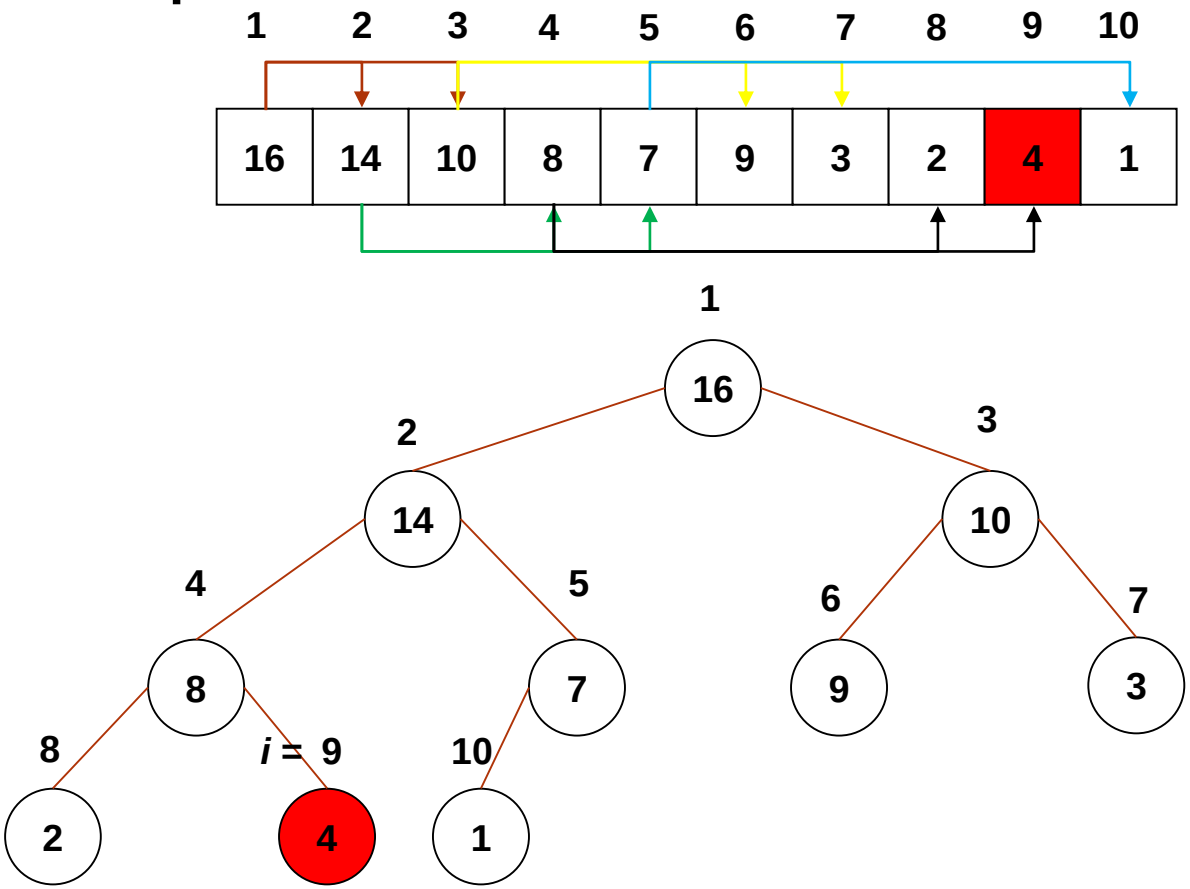
Heaps

- Exemplo:



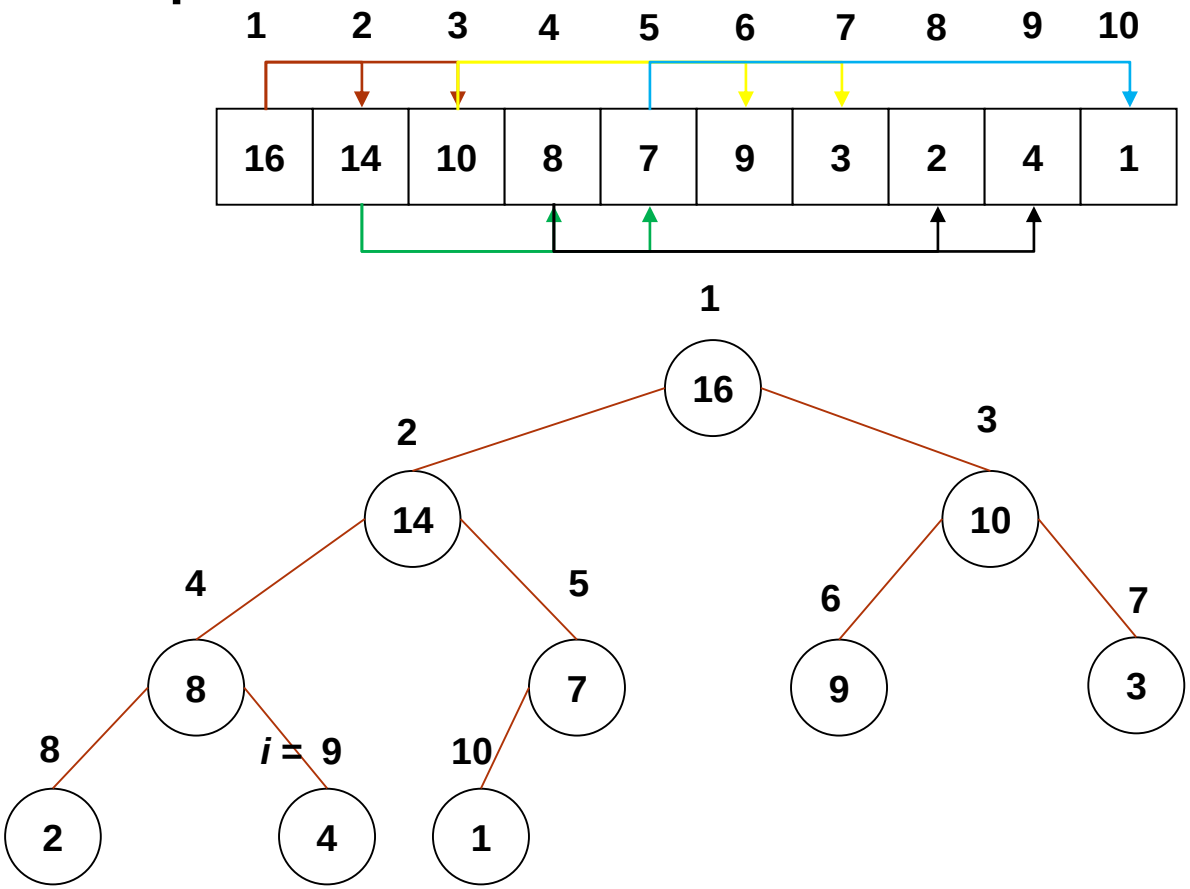
Heaps

- Exemplo:



Heaps

- Exemplo:



Heaps

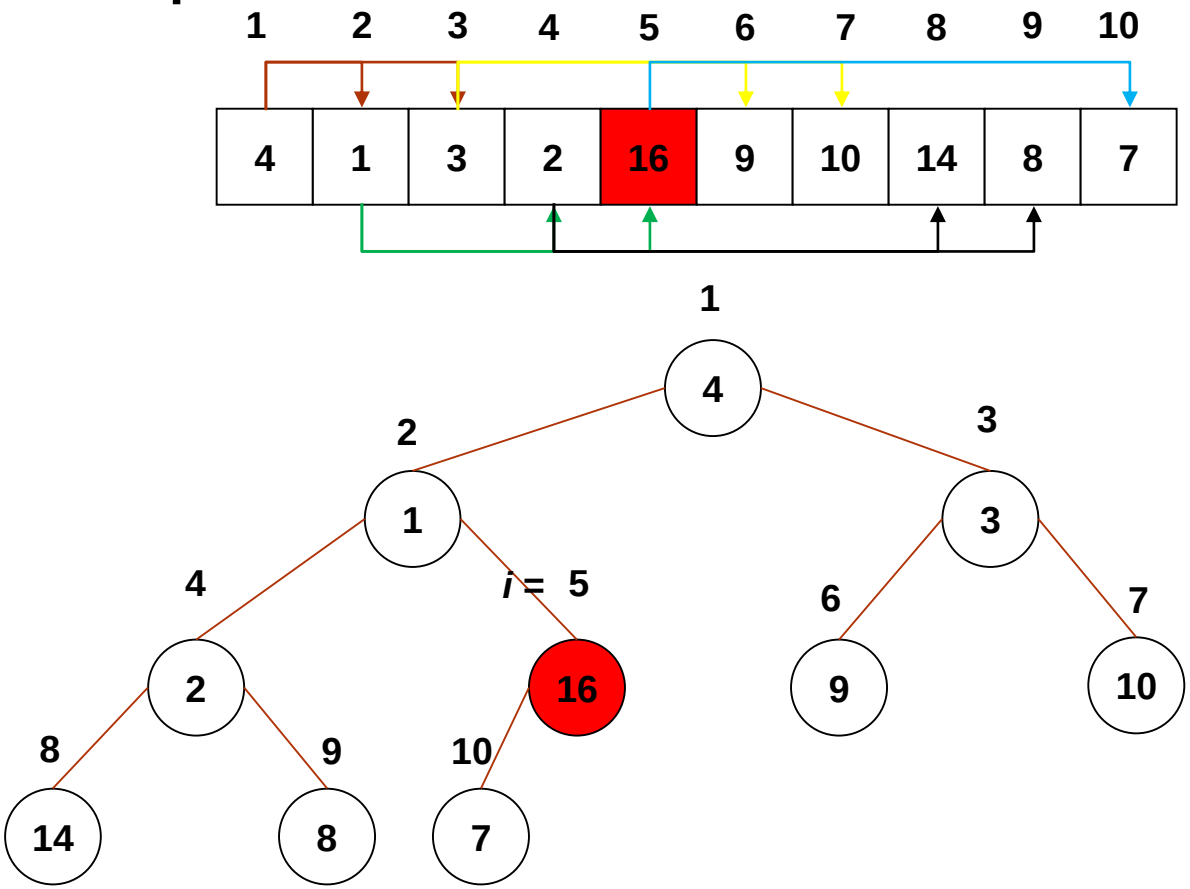
- A construção de um *Heap* pode ser feita de baixo-para-cima através do processo abaixo.

- No *Heapsort*, o tamanho do *Heap* considerado não necessariamente será igual ao do vetor de dados, como veremos em breve.

```
1.  //A -> Array<inteiro> que contém os dados
2.  //tamanhoHeap -> tamanho do Heap considerado
3.  procedimento construirMaxHeap(A, tamanhoHeap)
4.      para i = floor(tamanhoHeap / 2) até 1 //decremento unitário omitido
5.          maxHeapfy(A, i, tamanhoHeap)
```

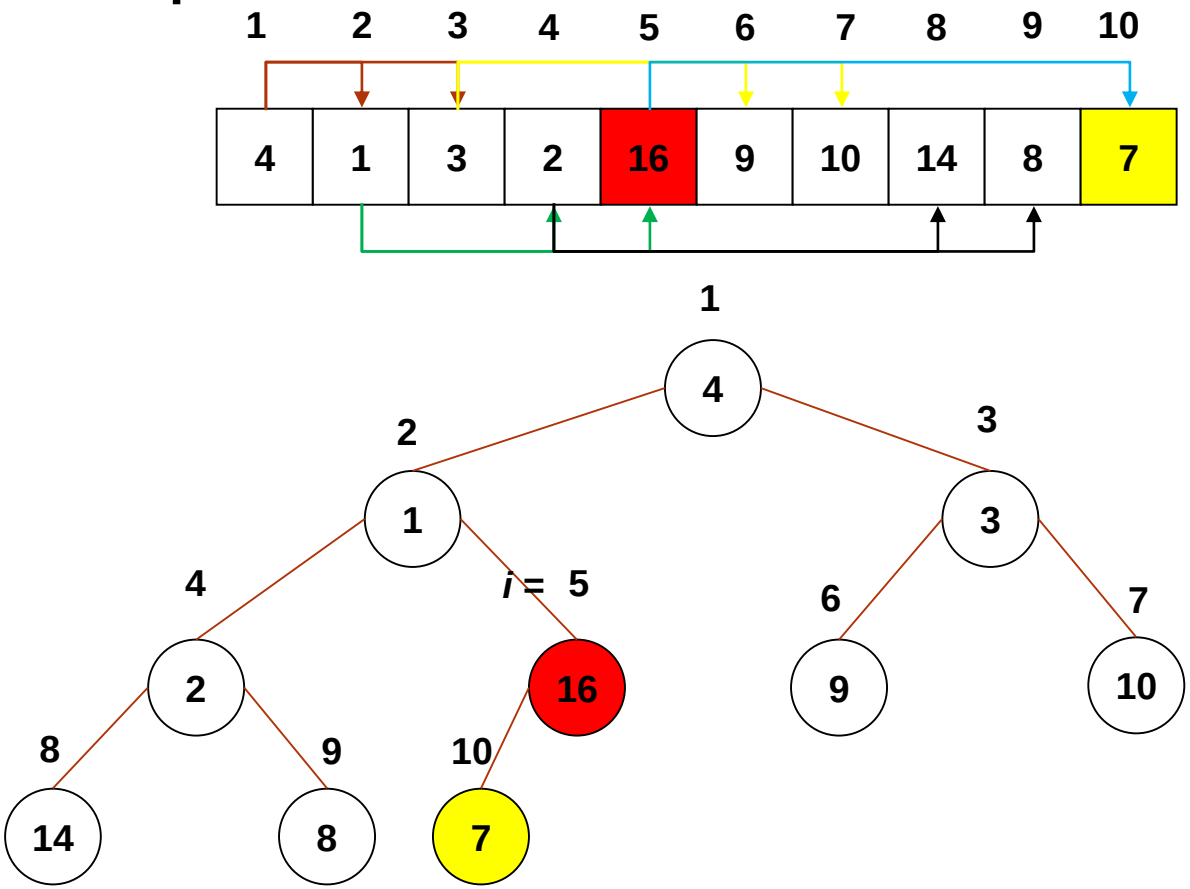
Heaps

- Exemplo:



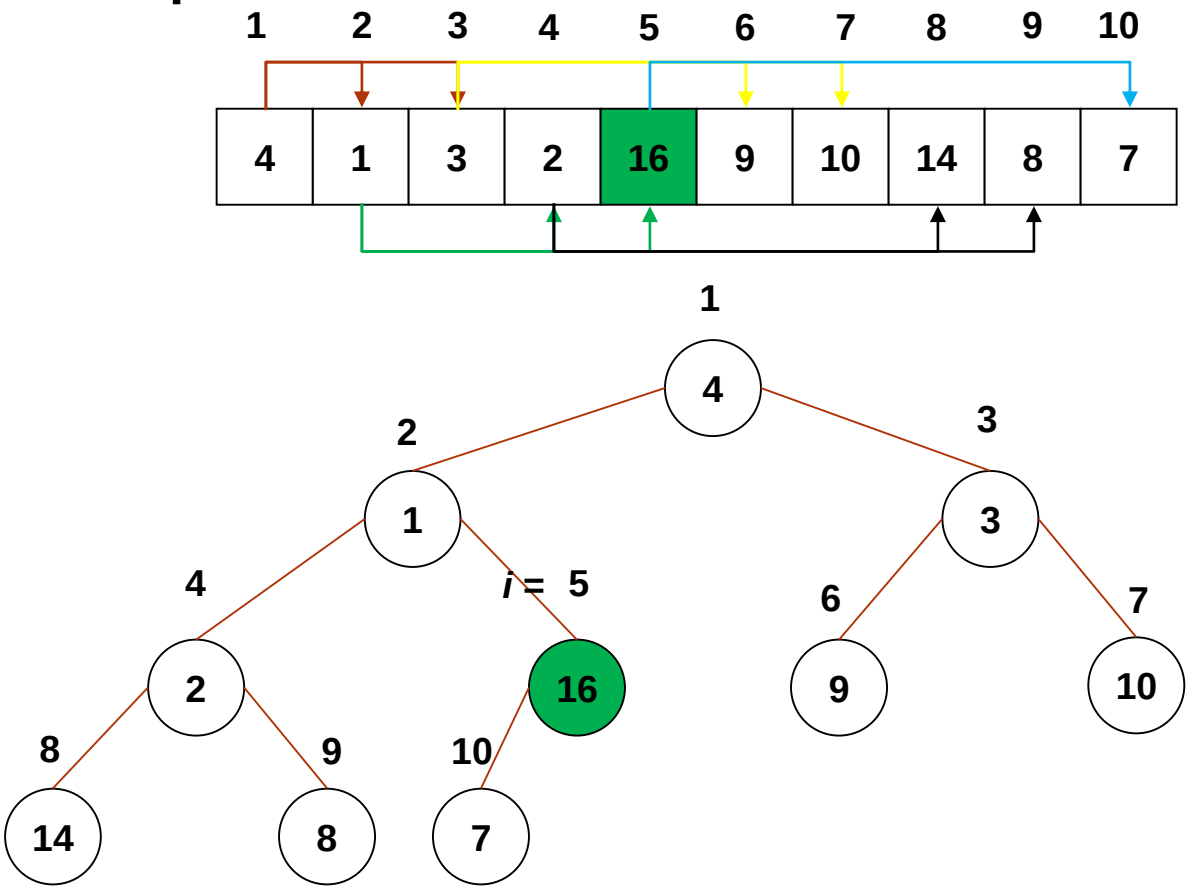
Heaps

- Exemplo:



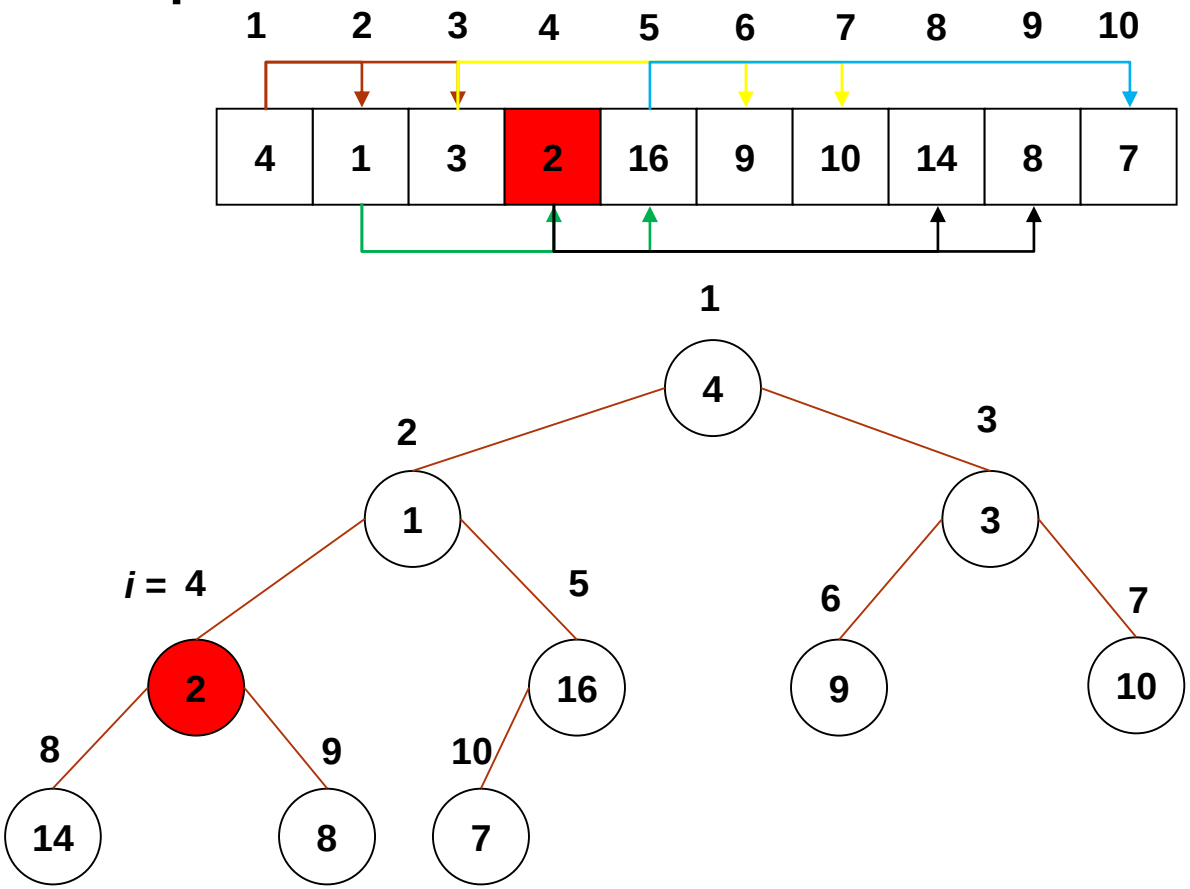
Heaps

- Exemplo:



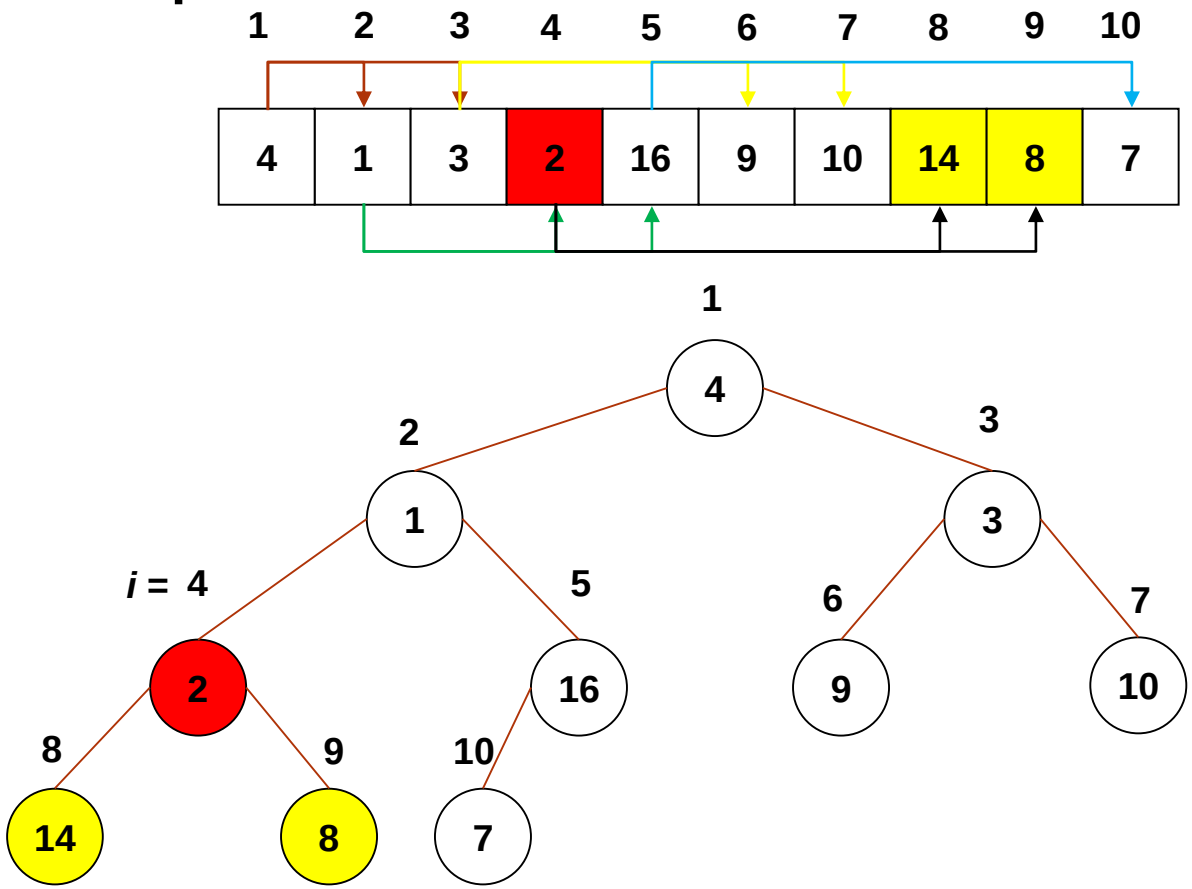
Heaps

- Exemplo:



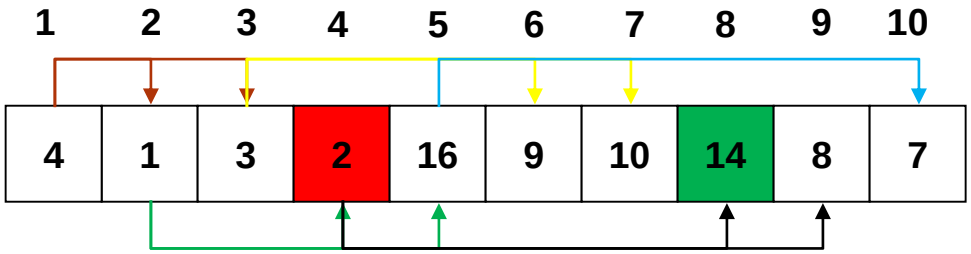
Heaps

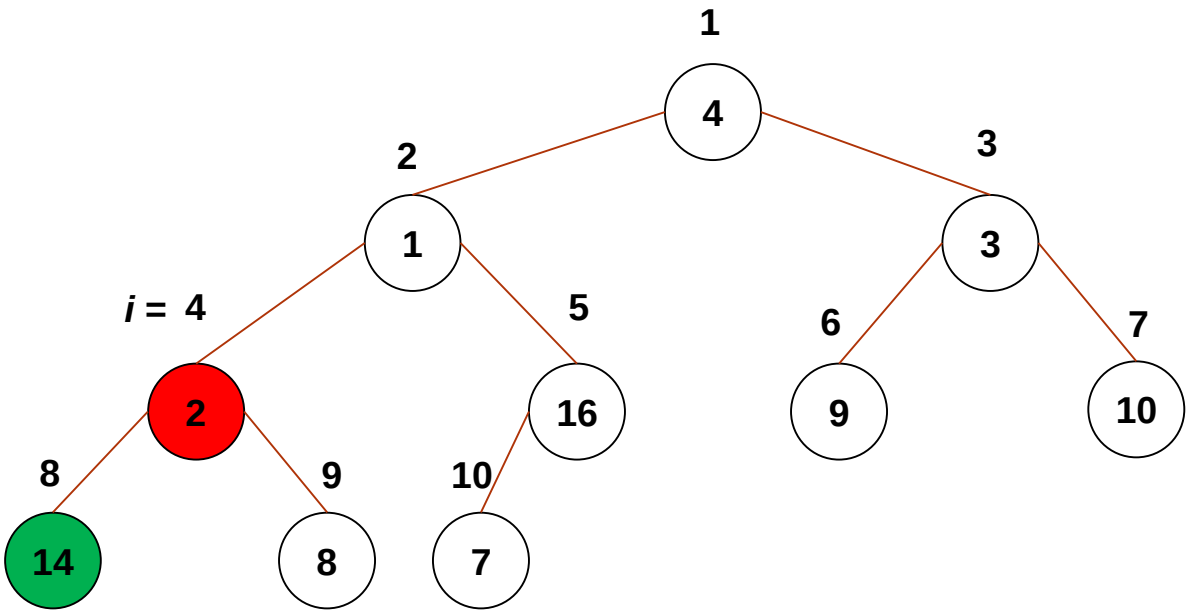
- Exemplo:



Heaps

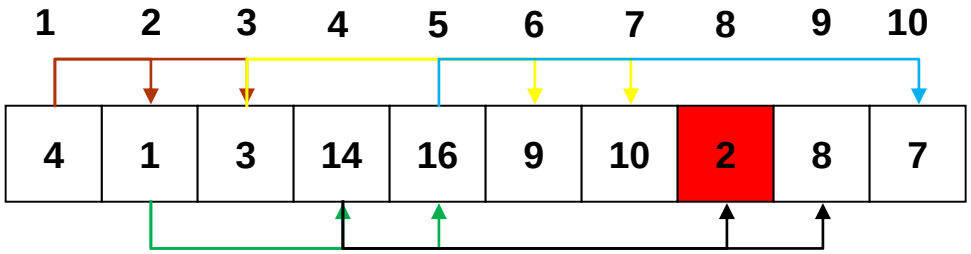
- Exemplo:

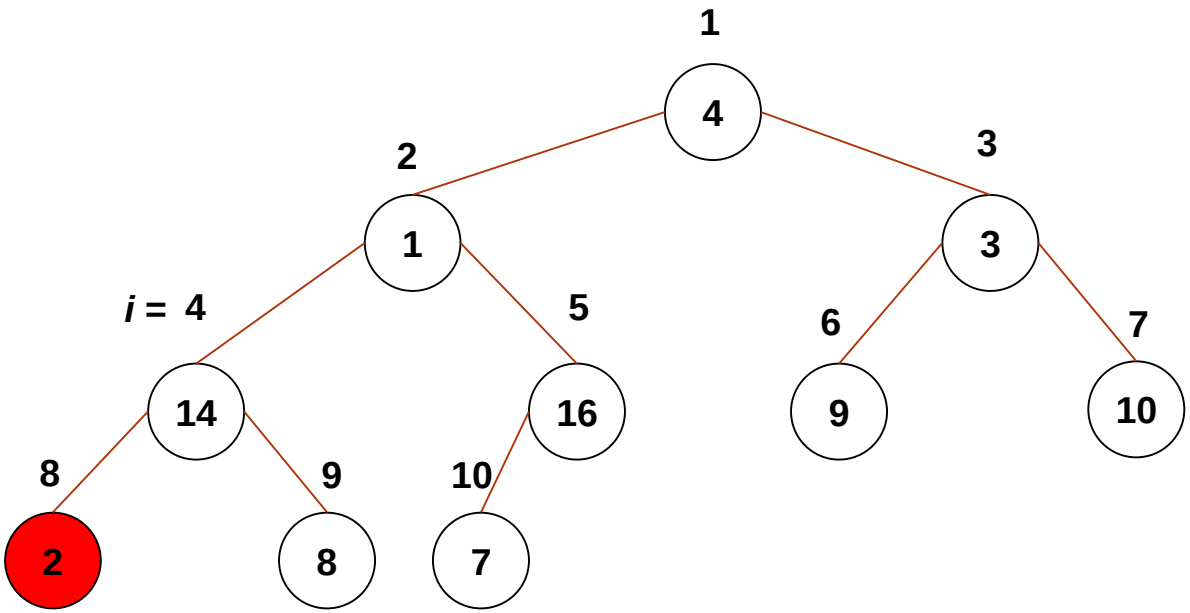




Heaps

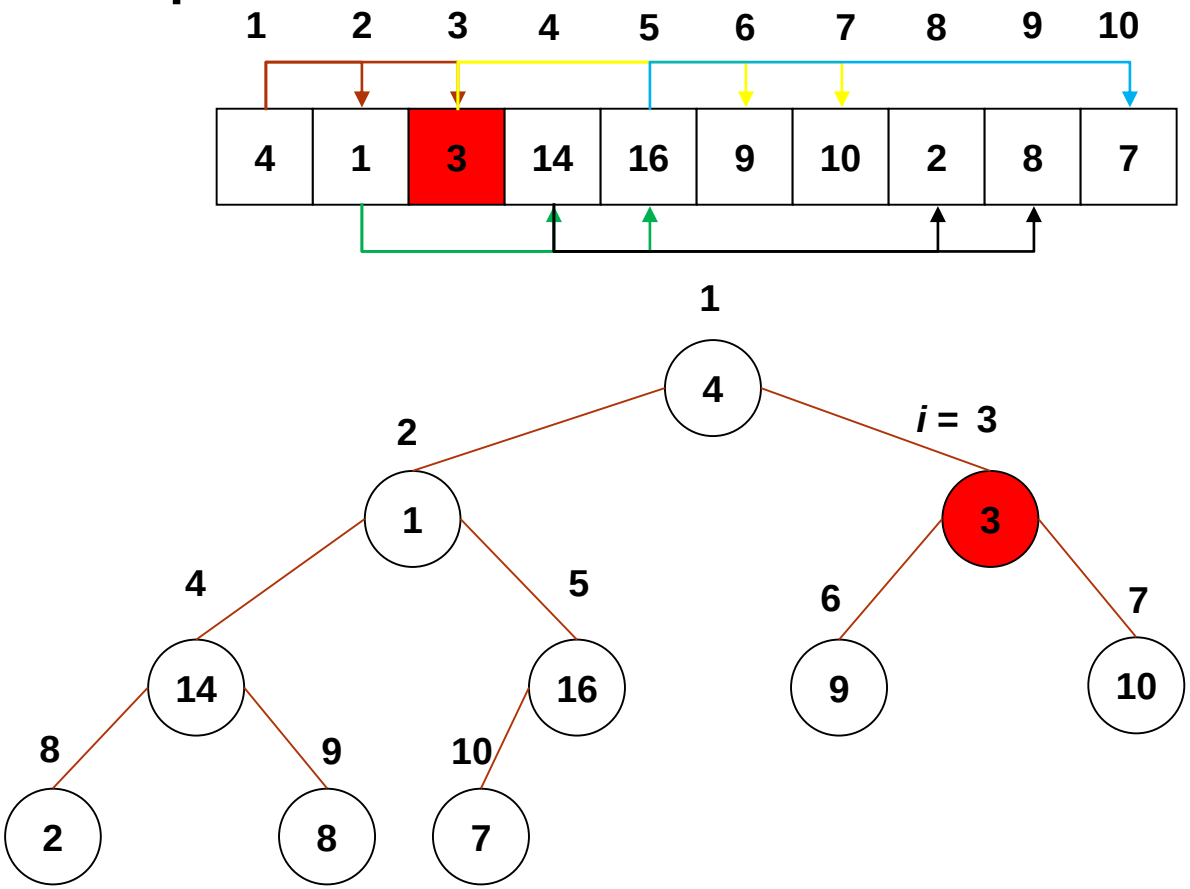
- Exemplo:





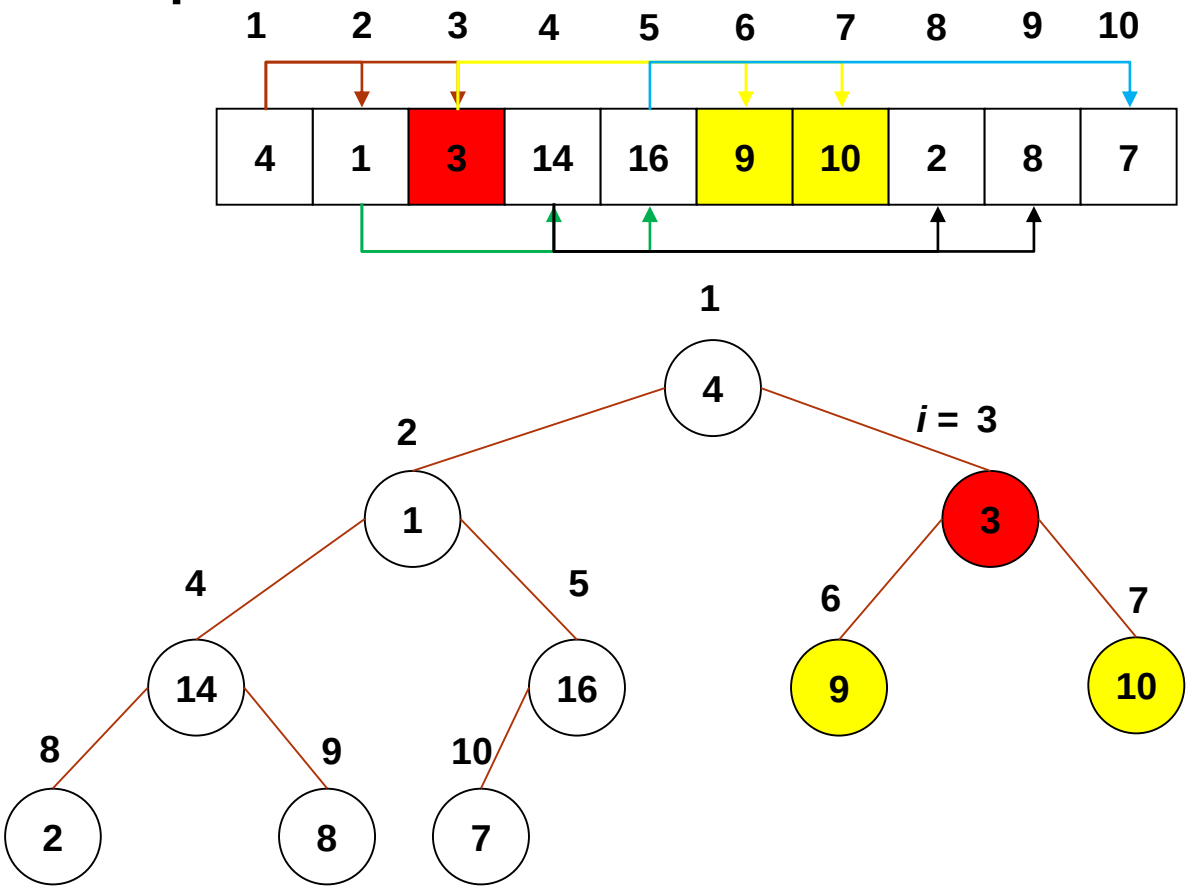
Heaps

- Exemplo:



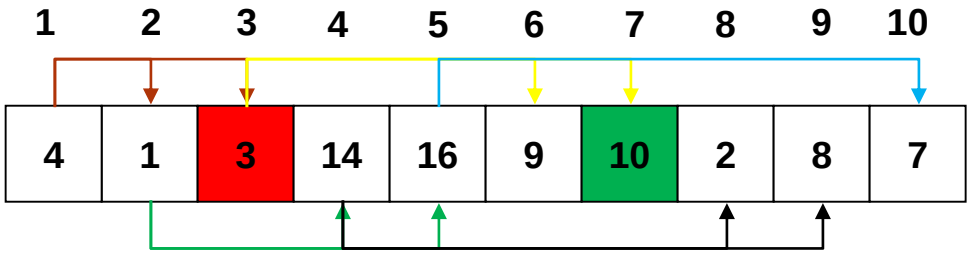
Heaps

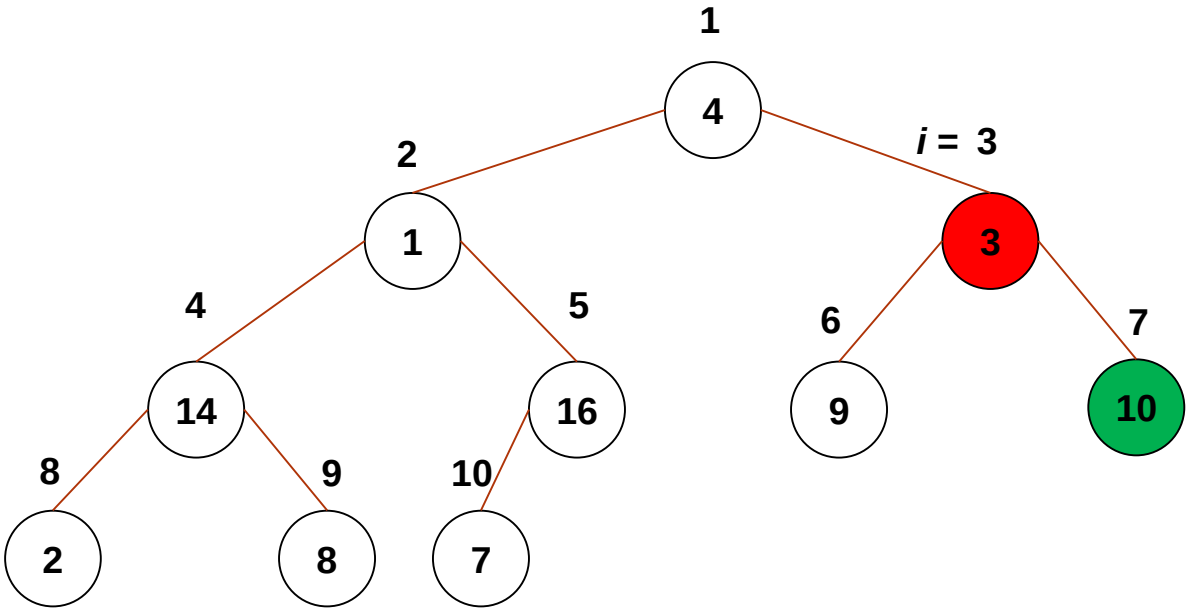
- Exemplo:



Heaps

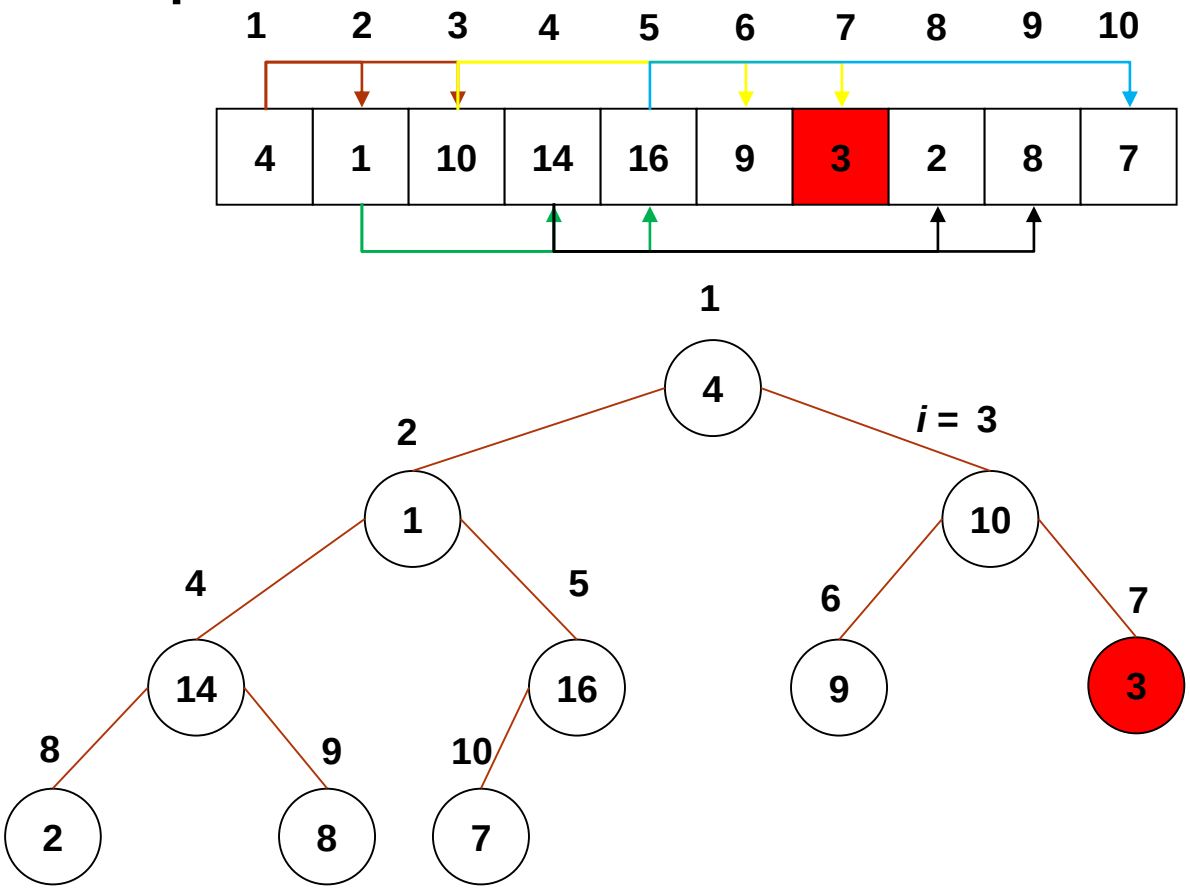
- Exemplo:





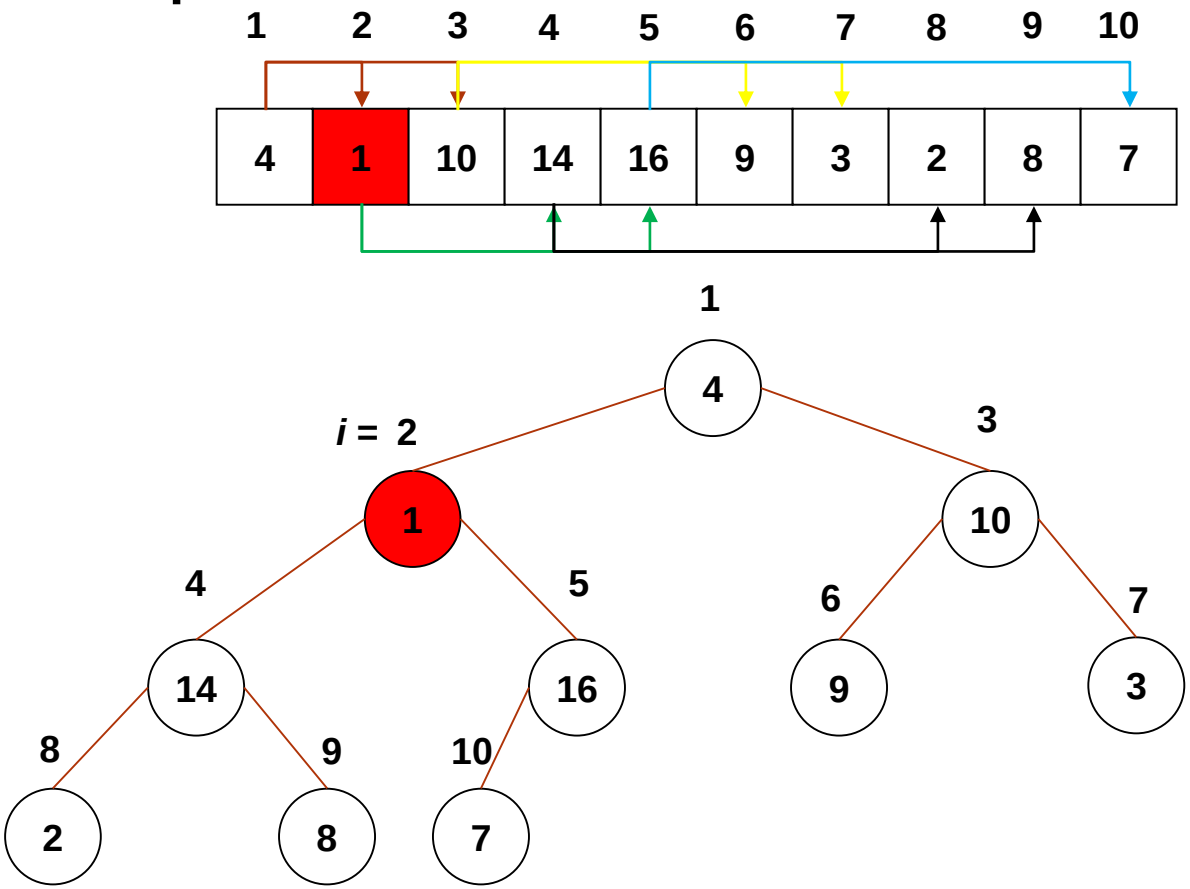
Heaps

- Exemplo:



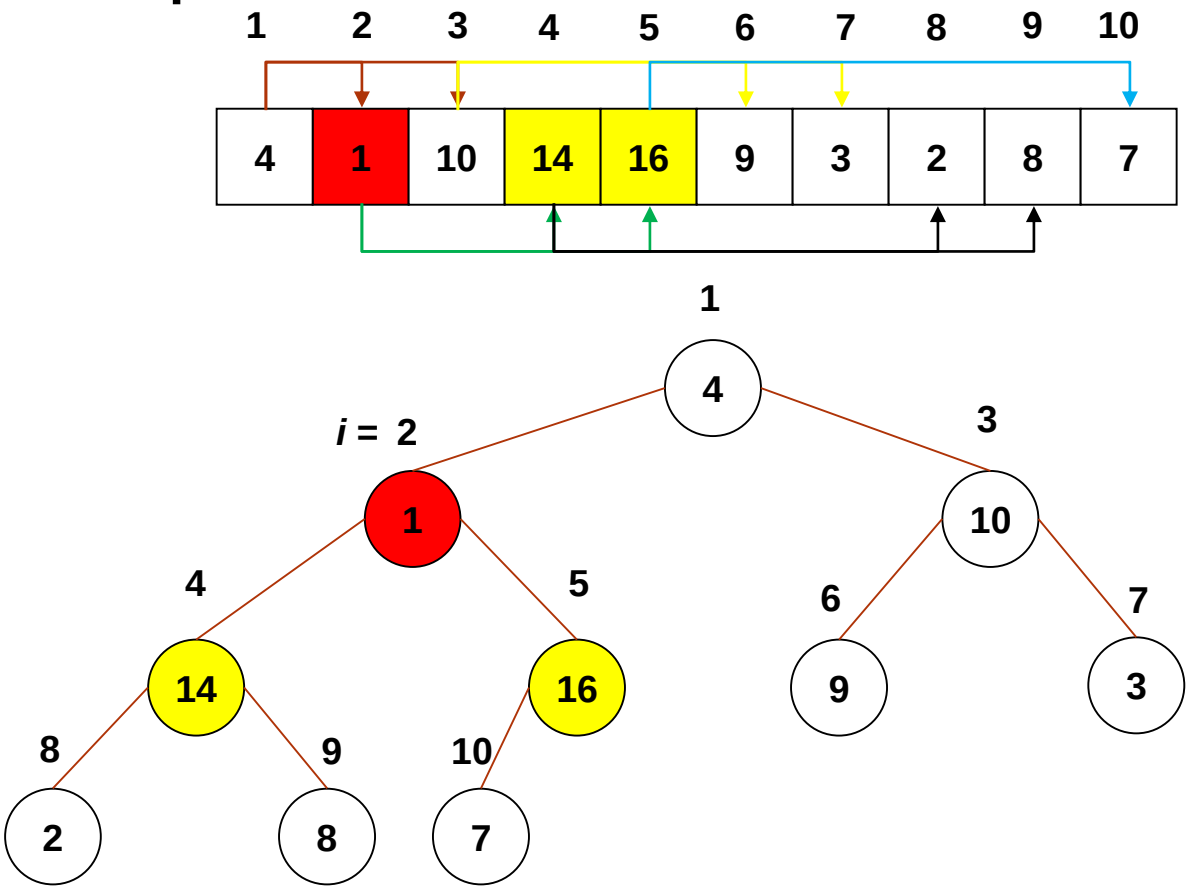
Heaps

- Exemplo:



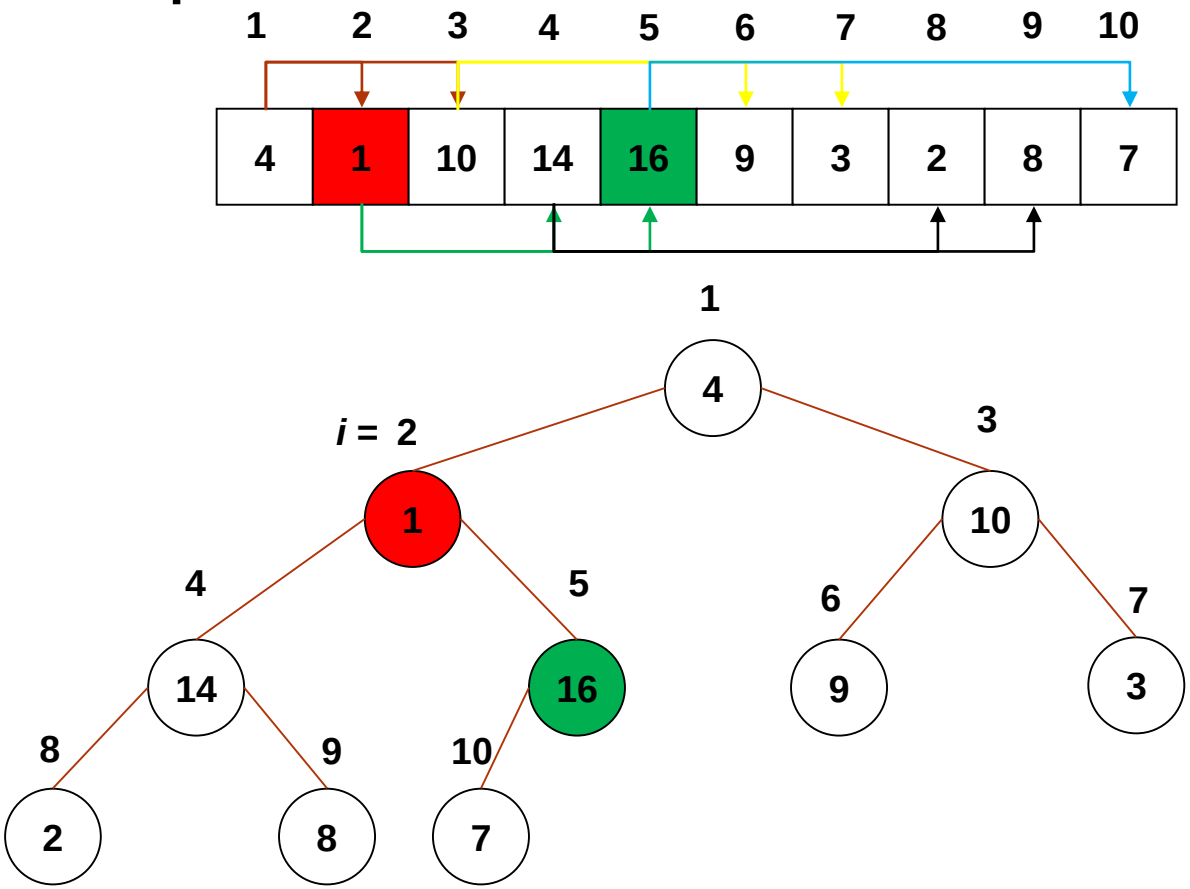
Heaps

- Exemplo:



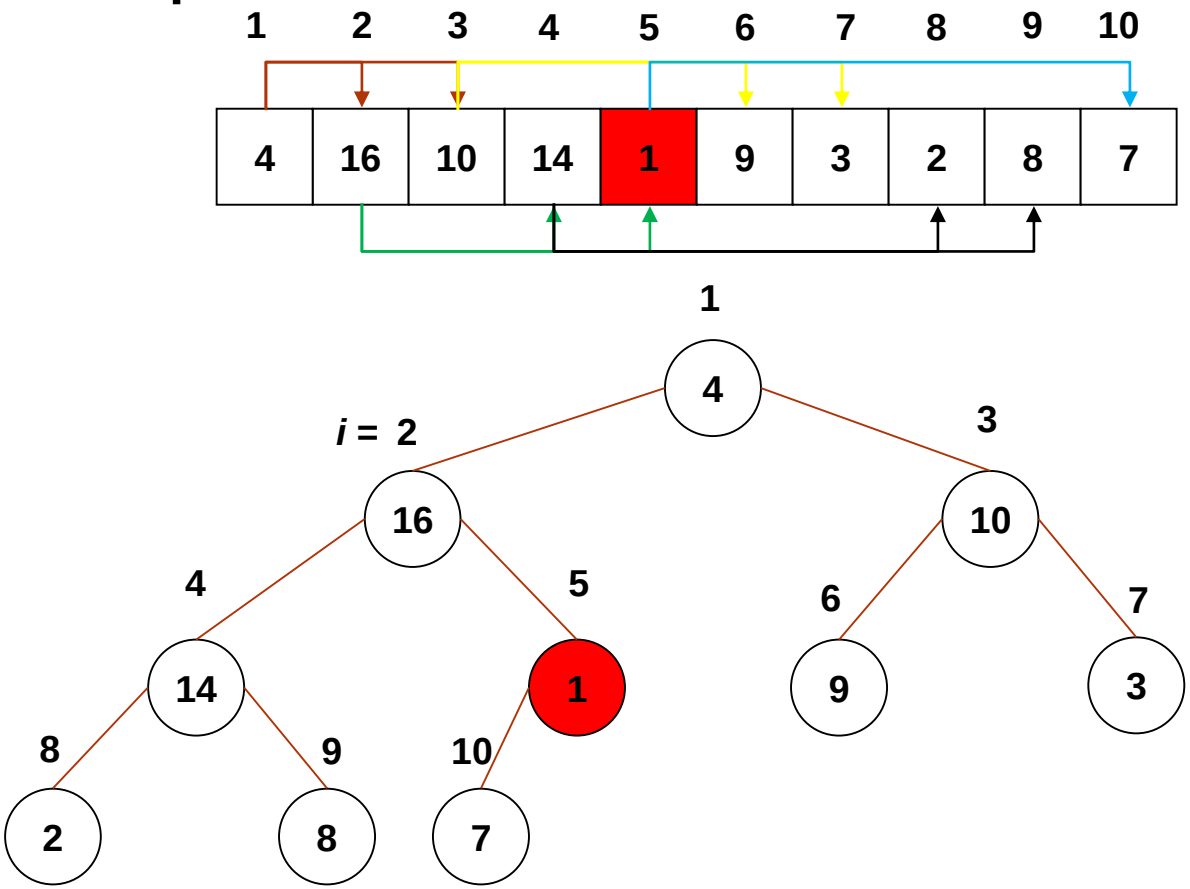
Heaps

- Exemplo:



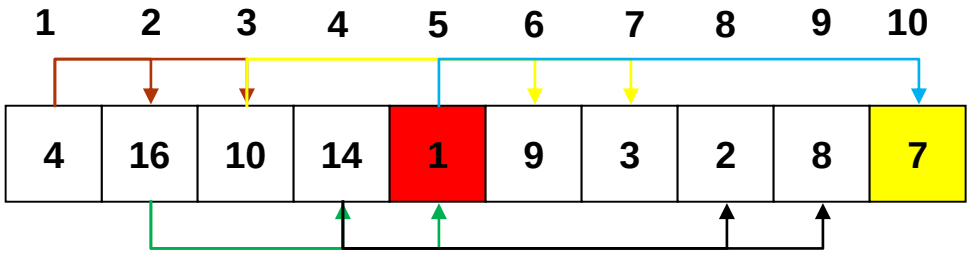
Heaps

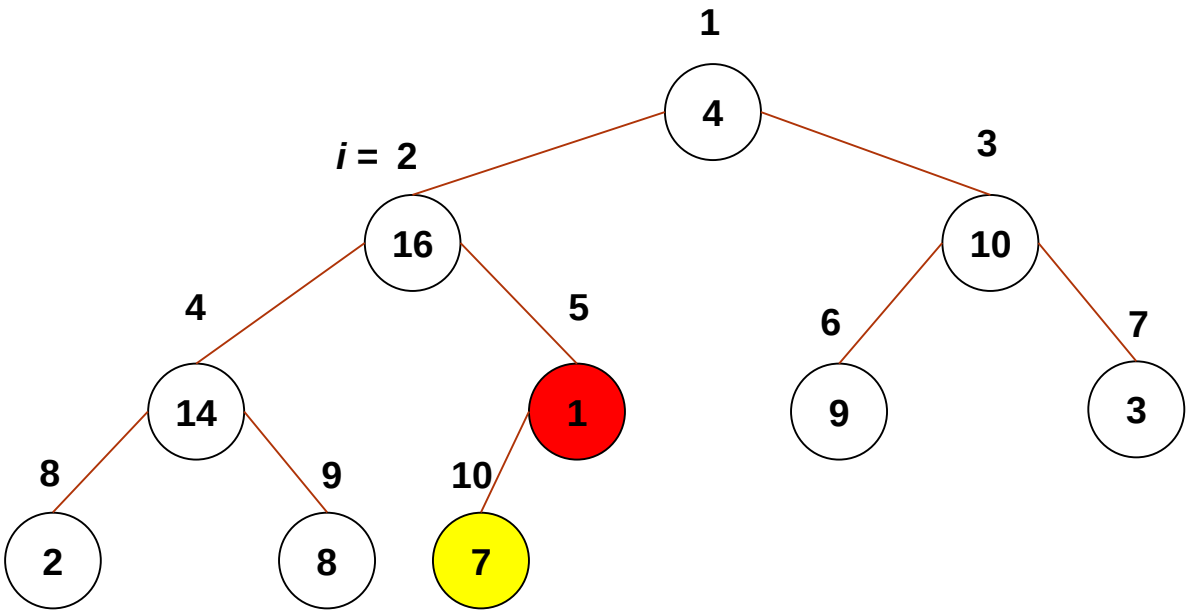
- Exemplo:



Heaps

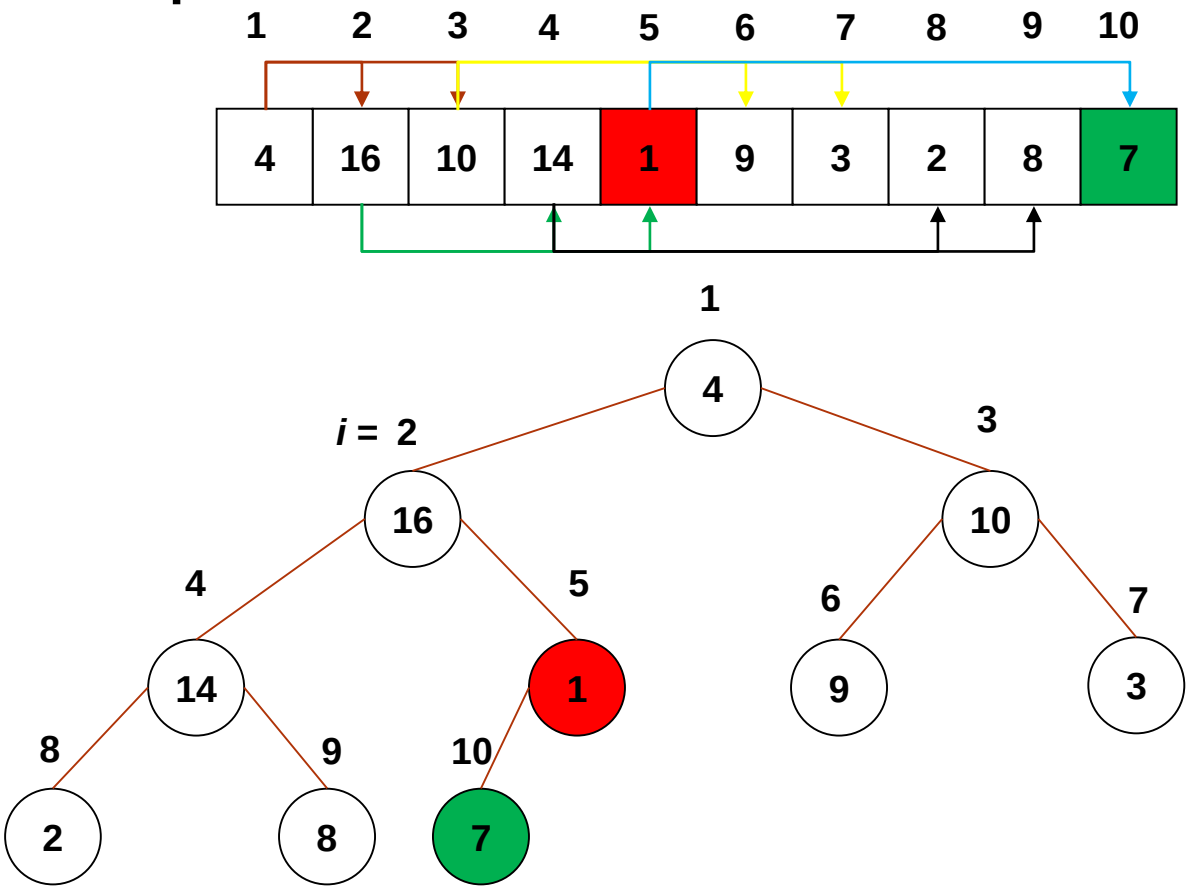
- Exemplo:





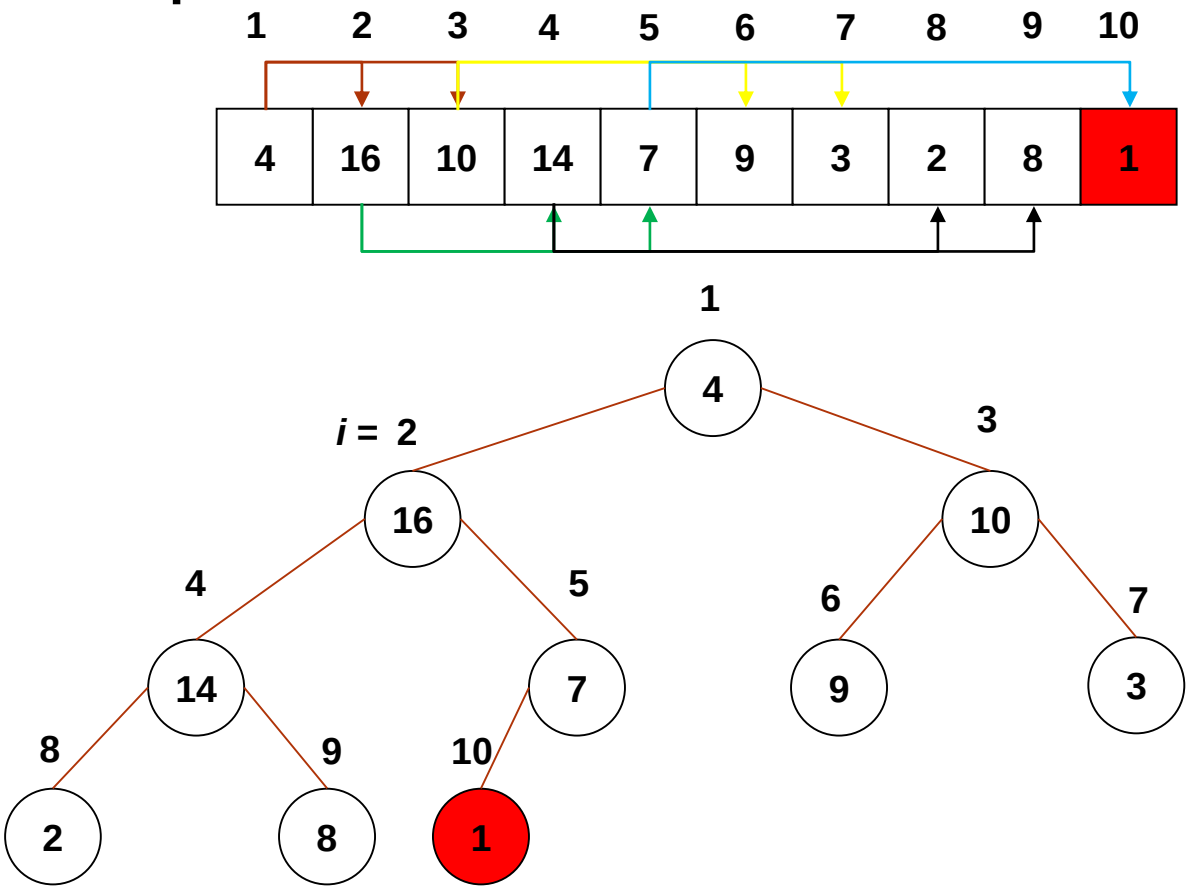
Heaps

- Exemplo:



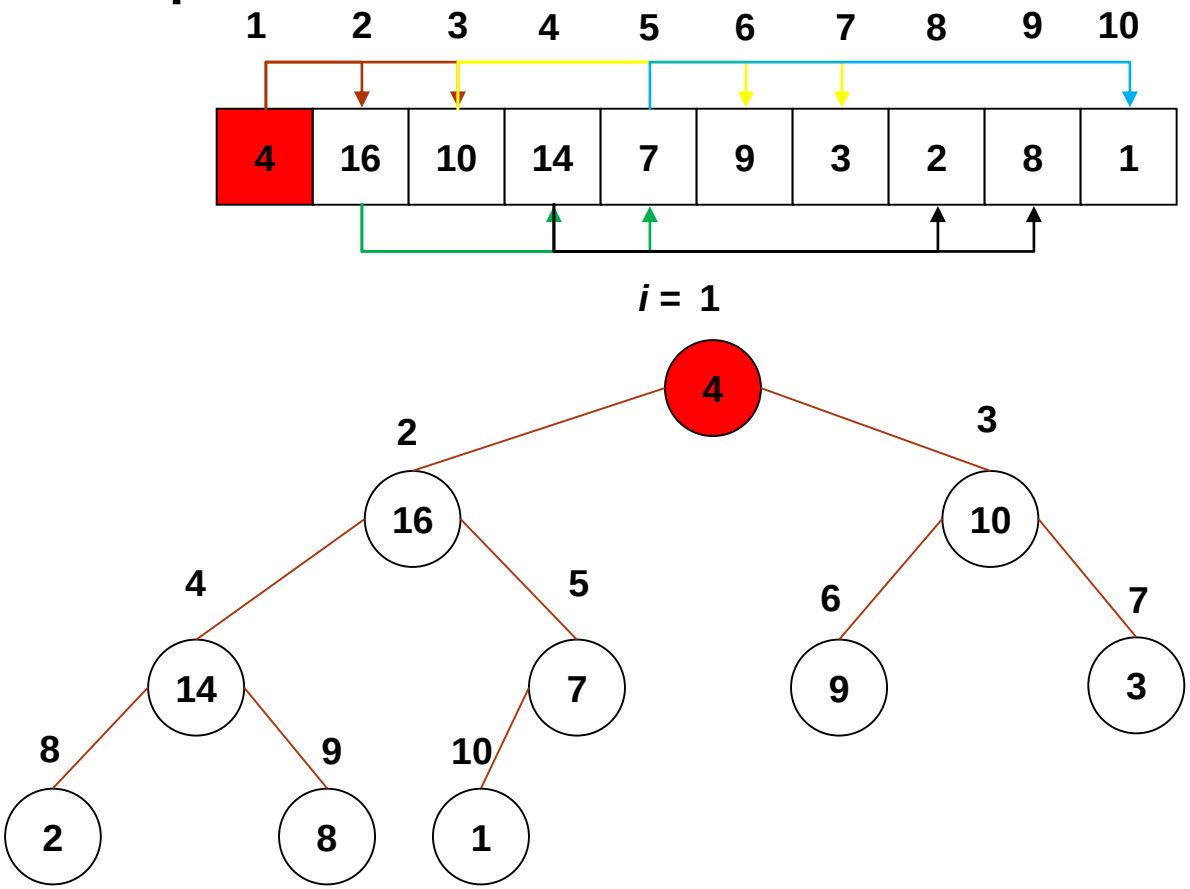
Heaps

- Exemplo:



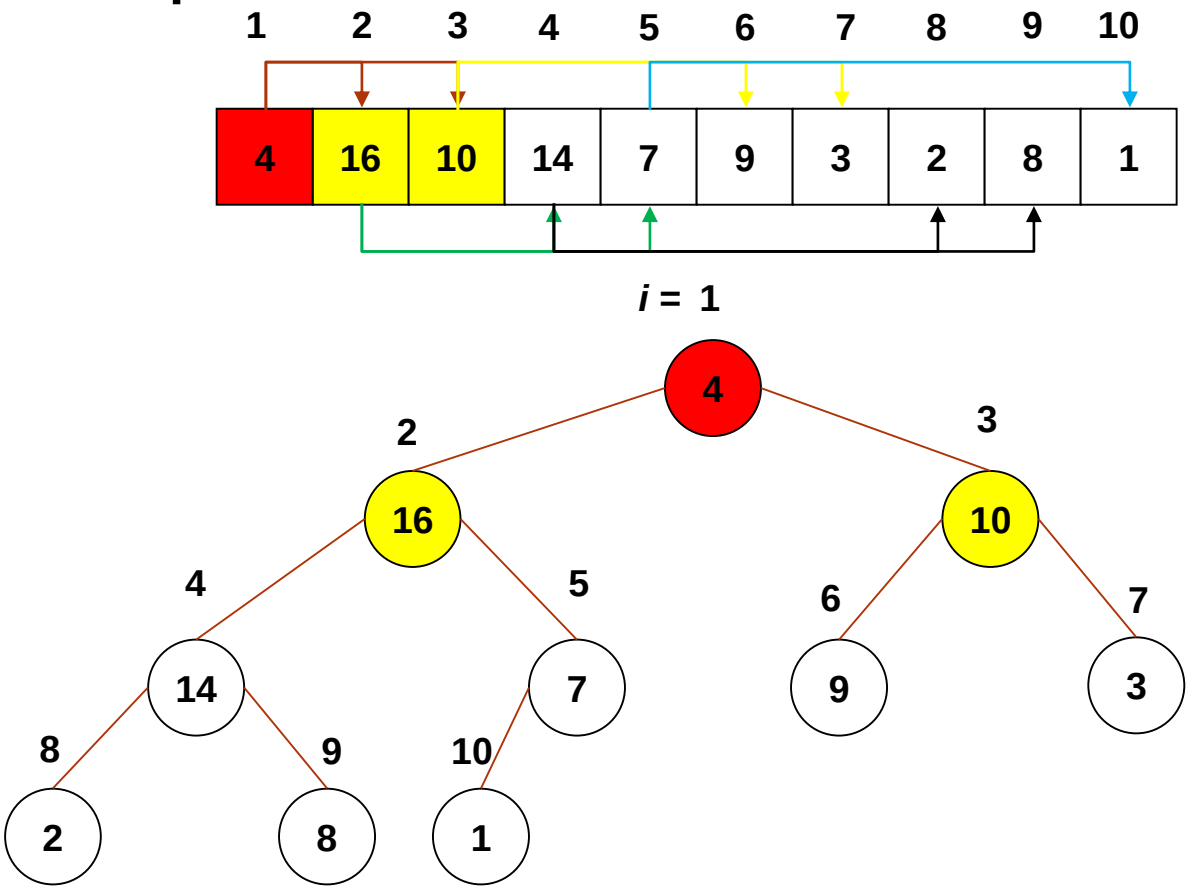
Heaps

- Exemplo:



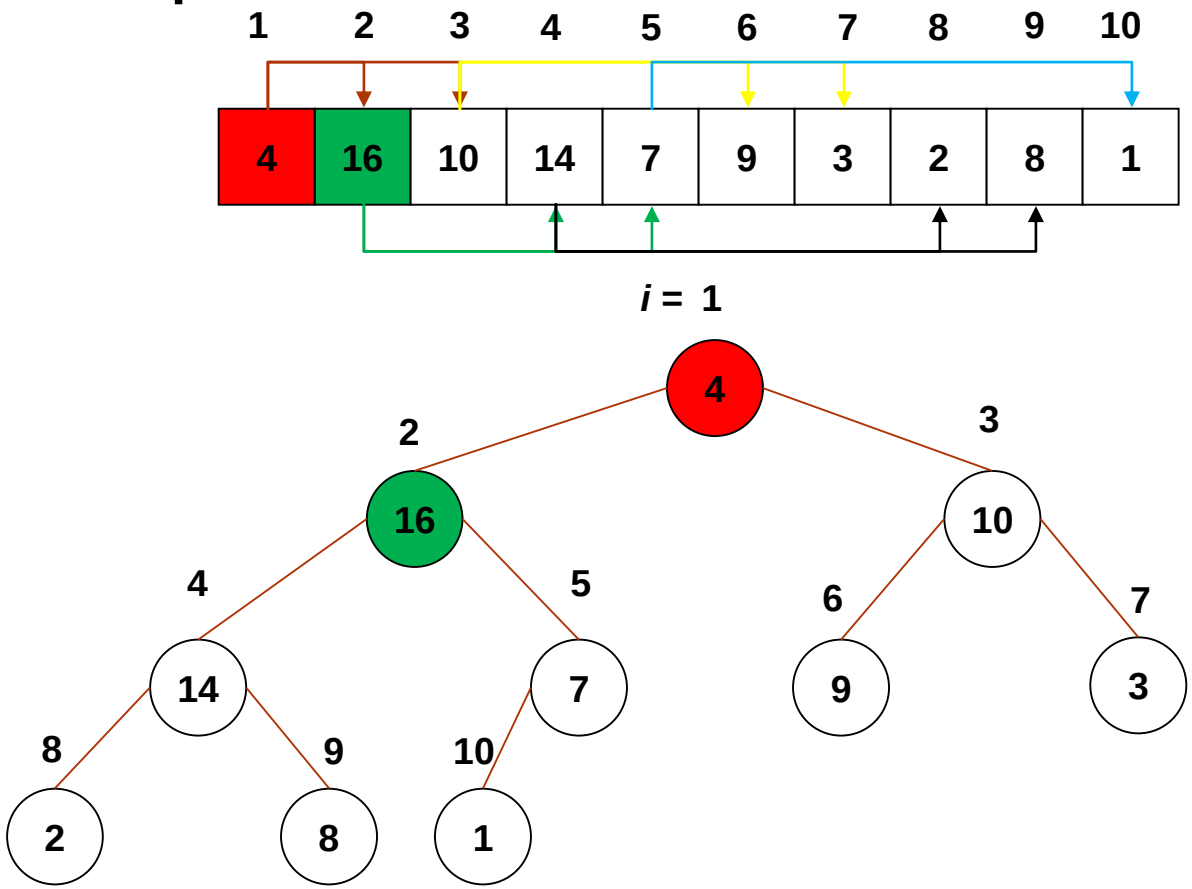
Heaps

- Exemplo:



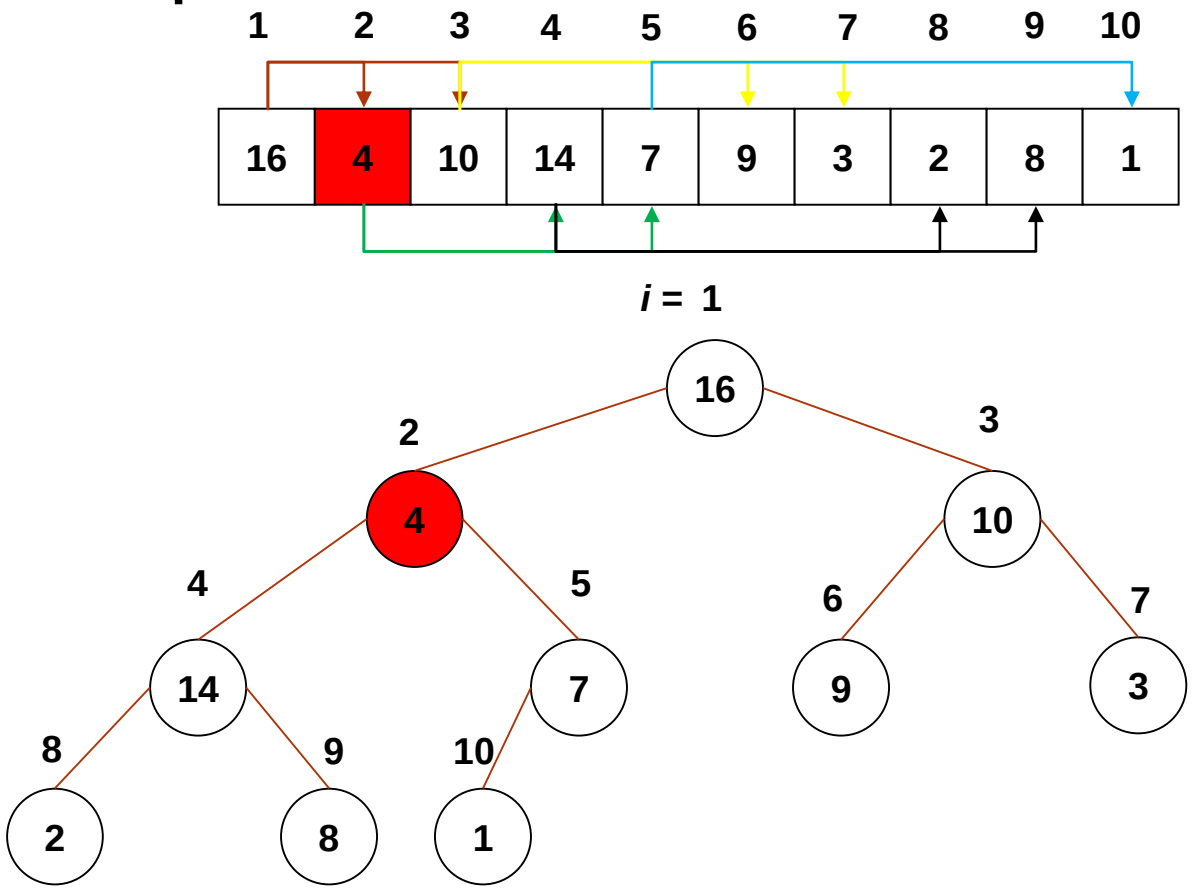
Heaps

- Exemplo:



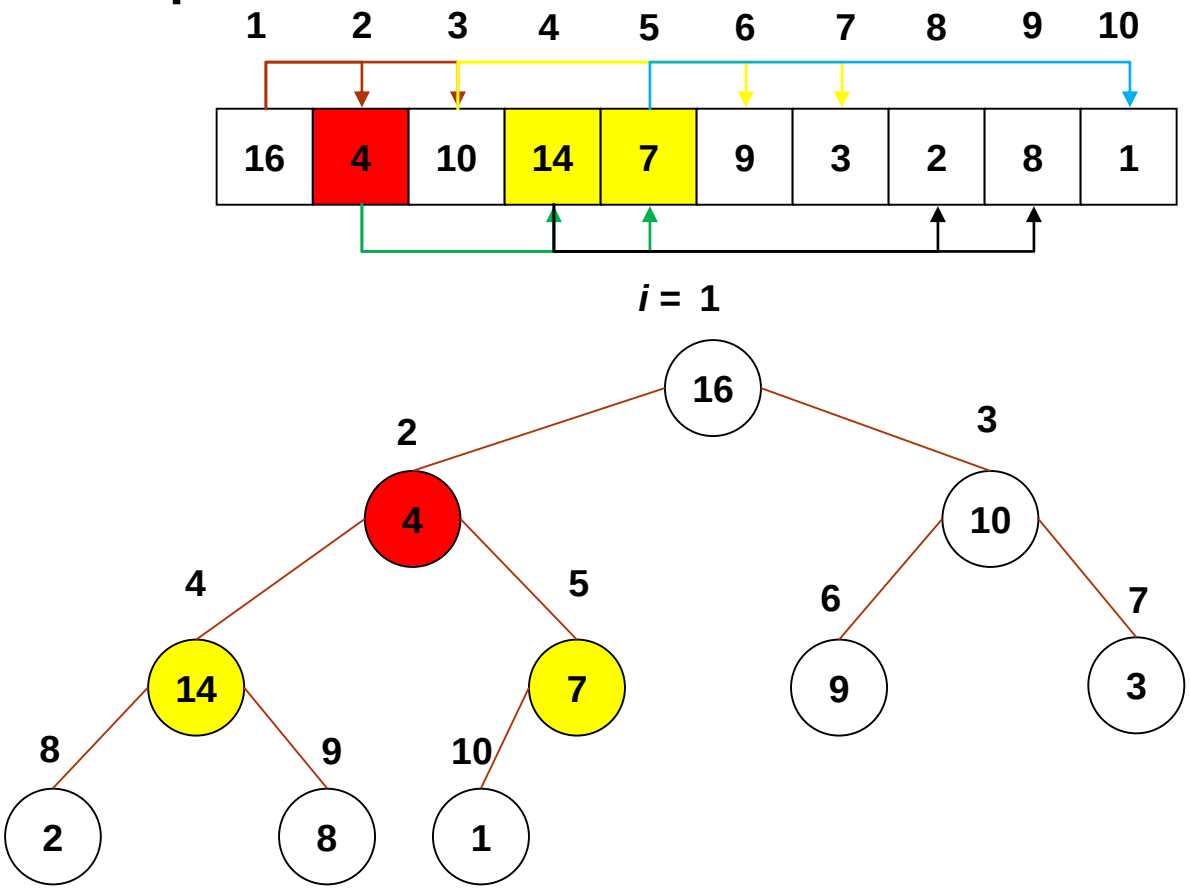
Heaps

- Exemplo:



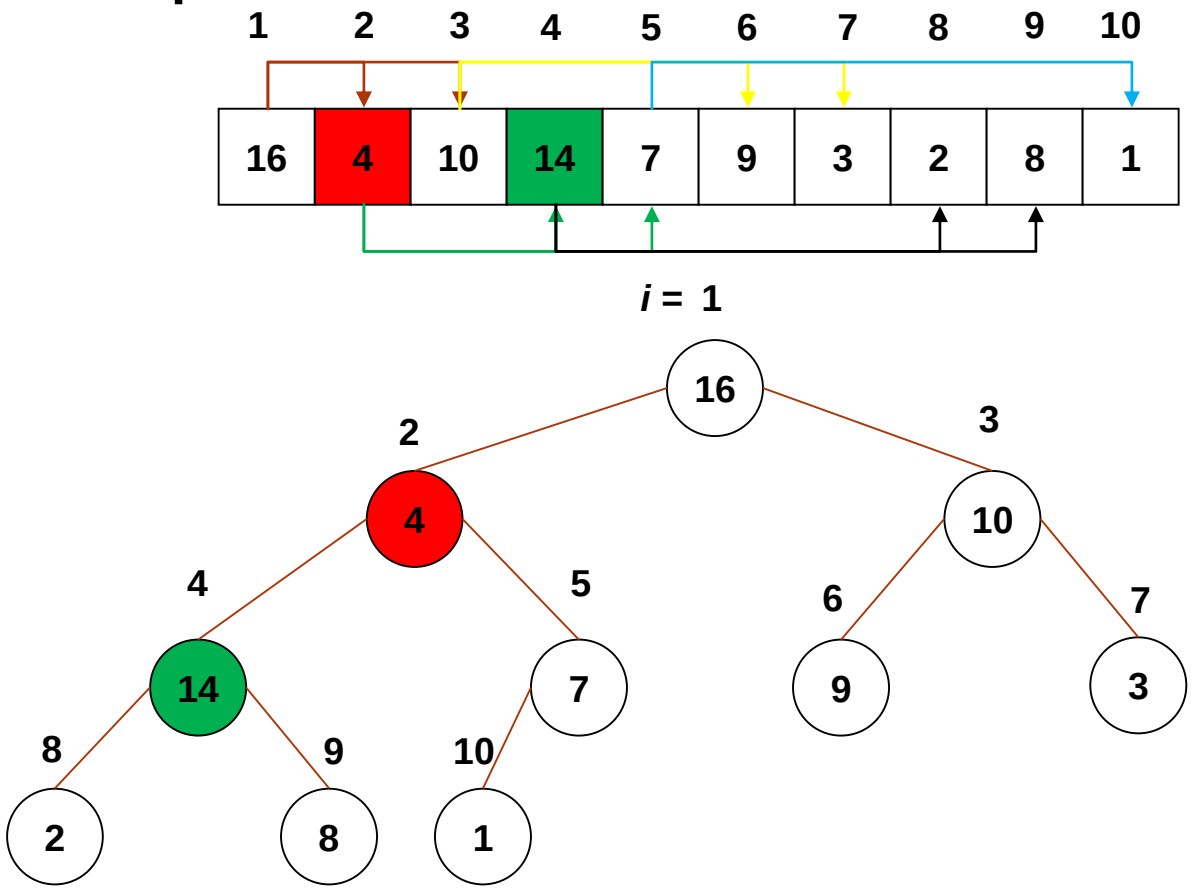
Heaps

- Exemplo:



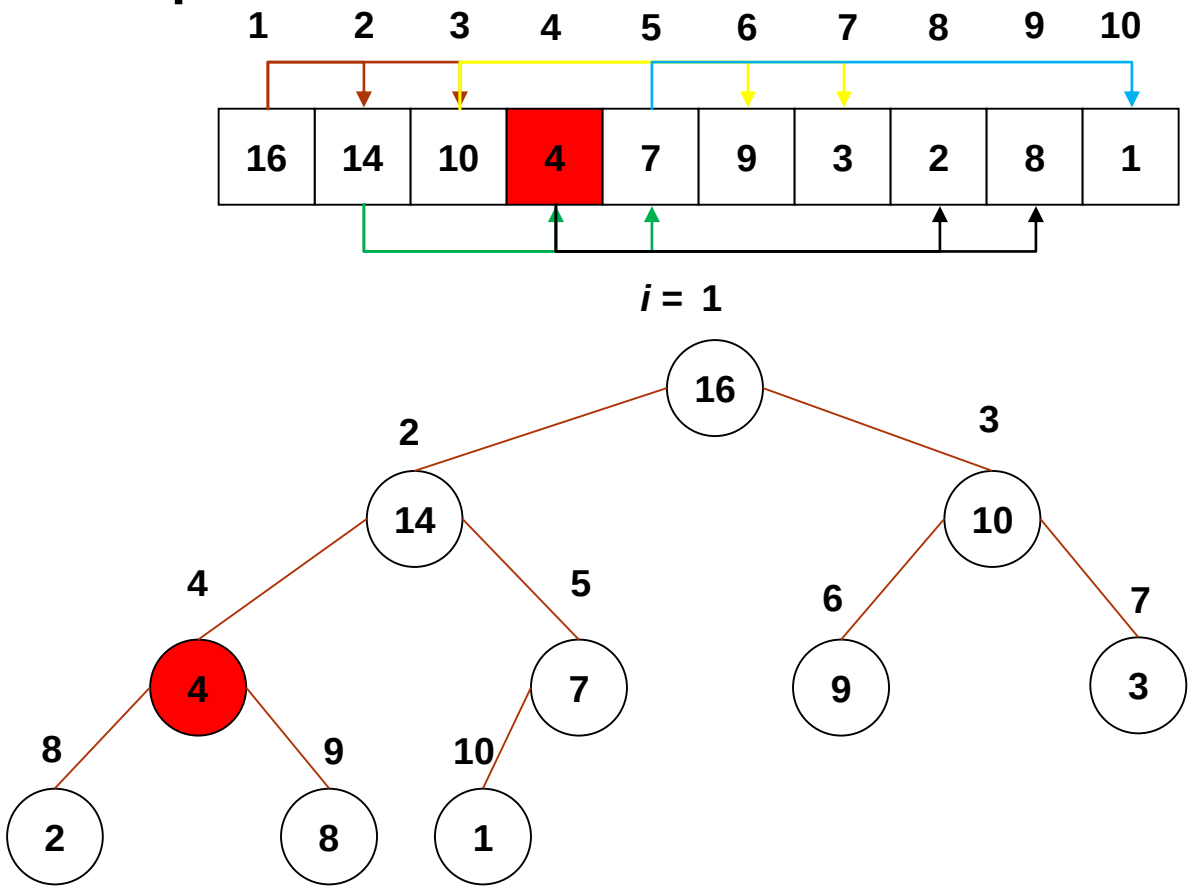
Heaps

- Exemplo:



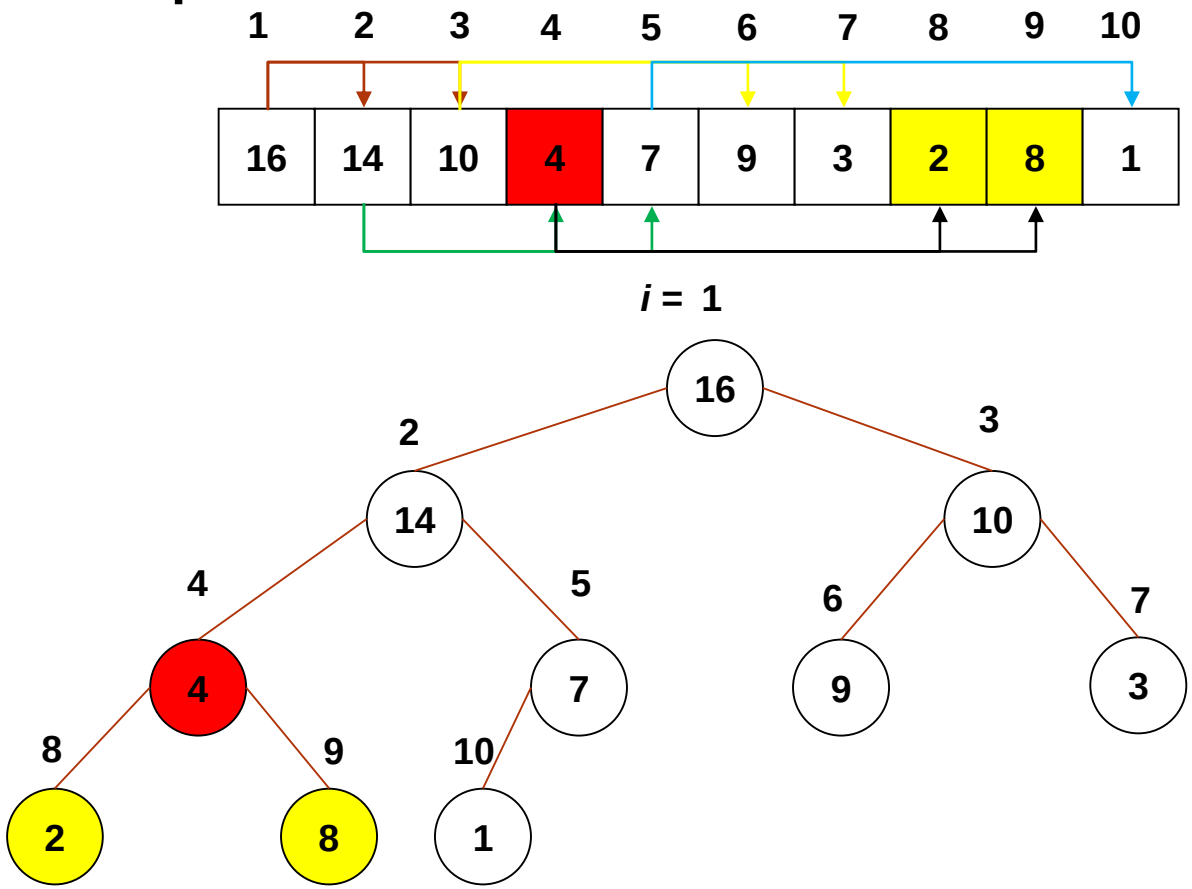
Heaps

- Exemplo:



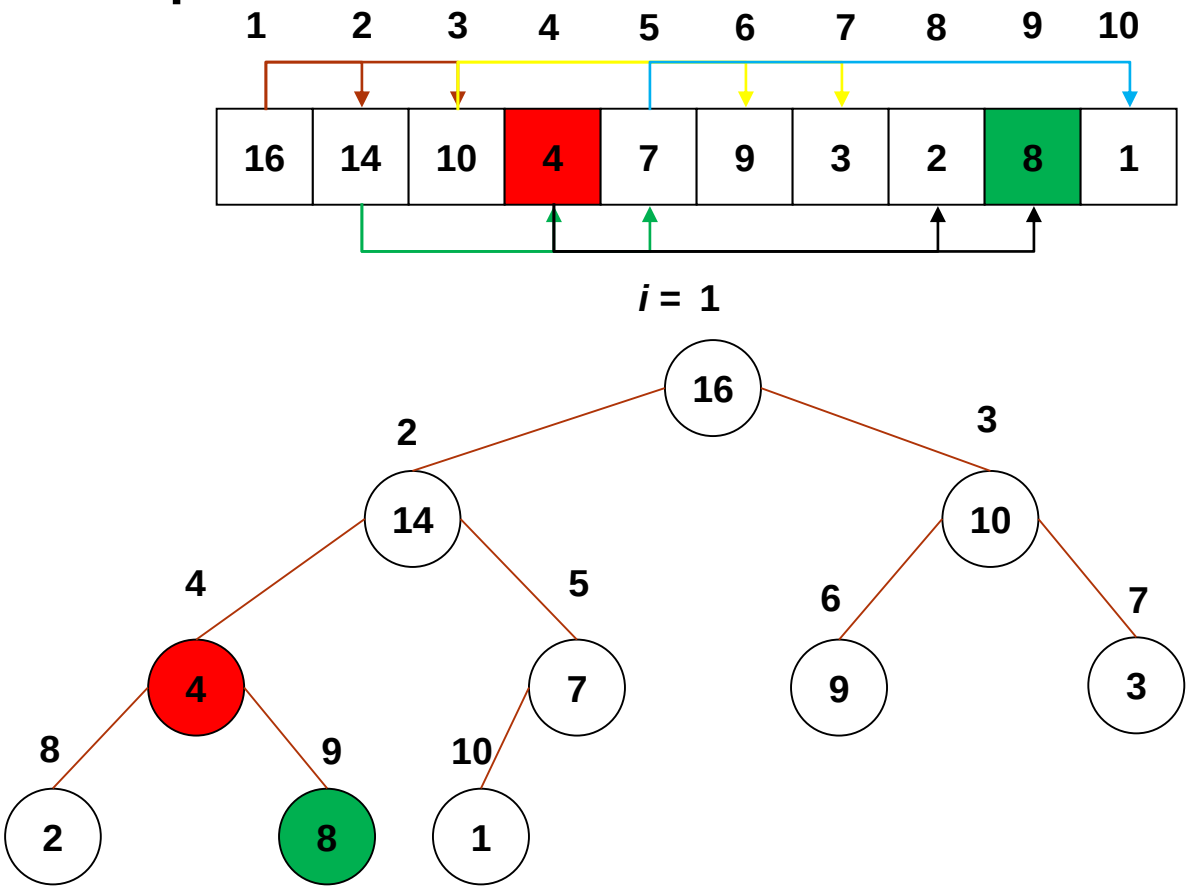
Heaps

- Exemplo:



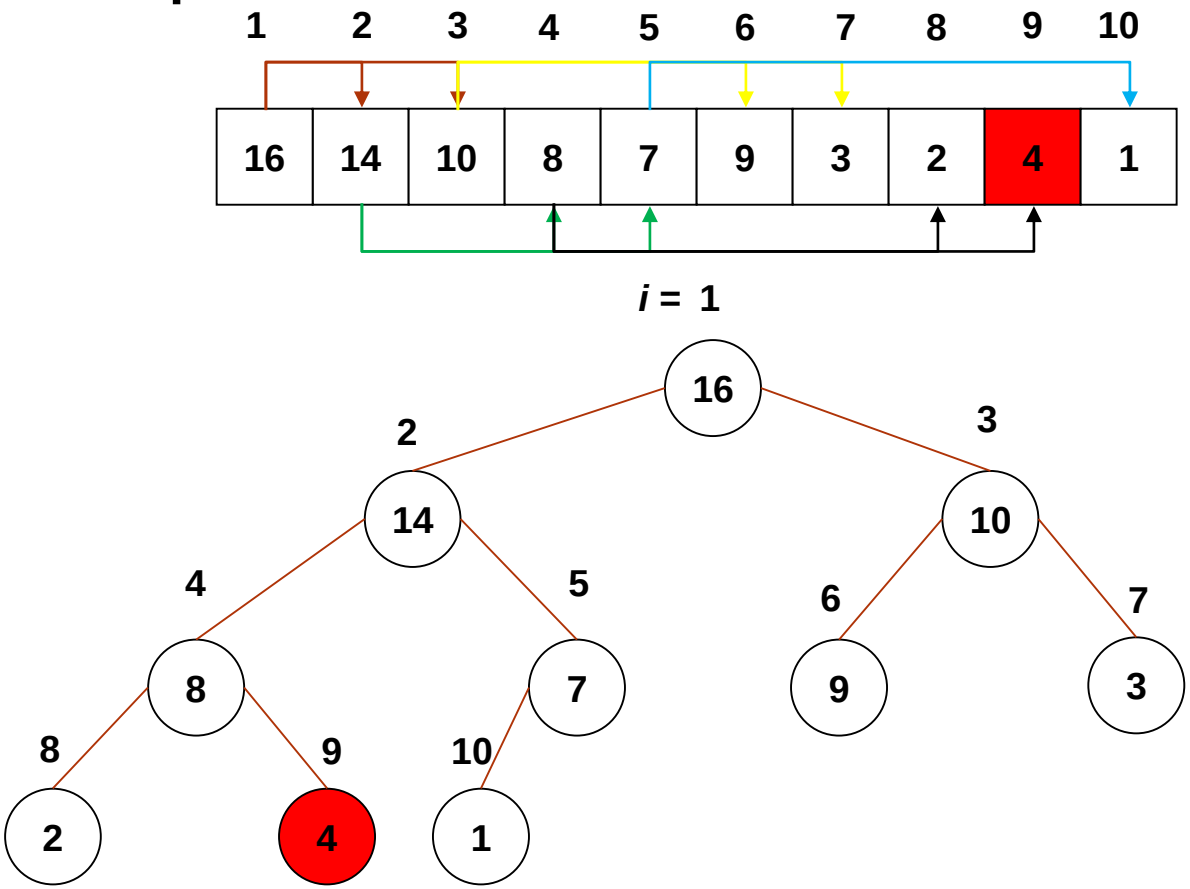
Heaps

- Exemplo:



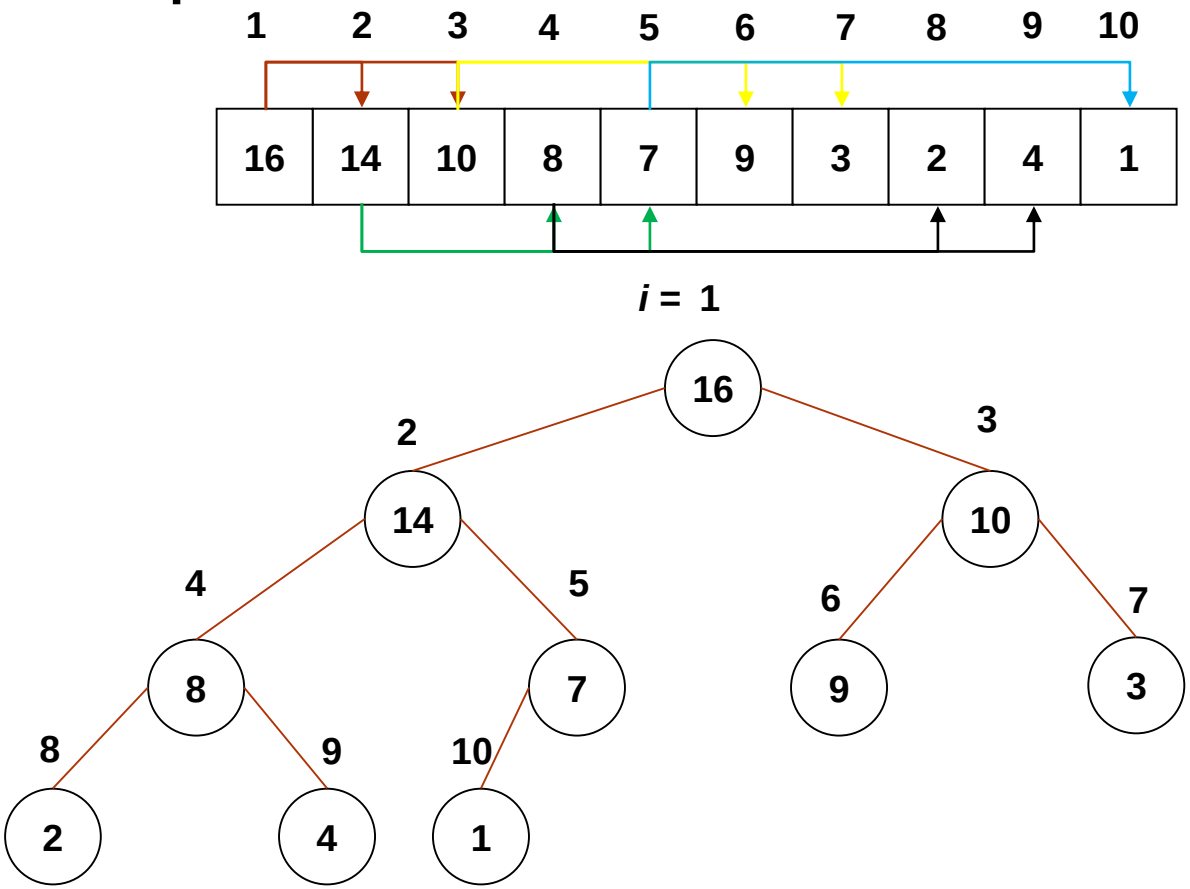
Heaps

- Exemplo:



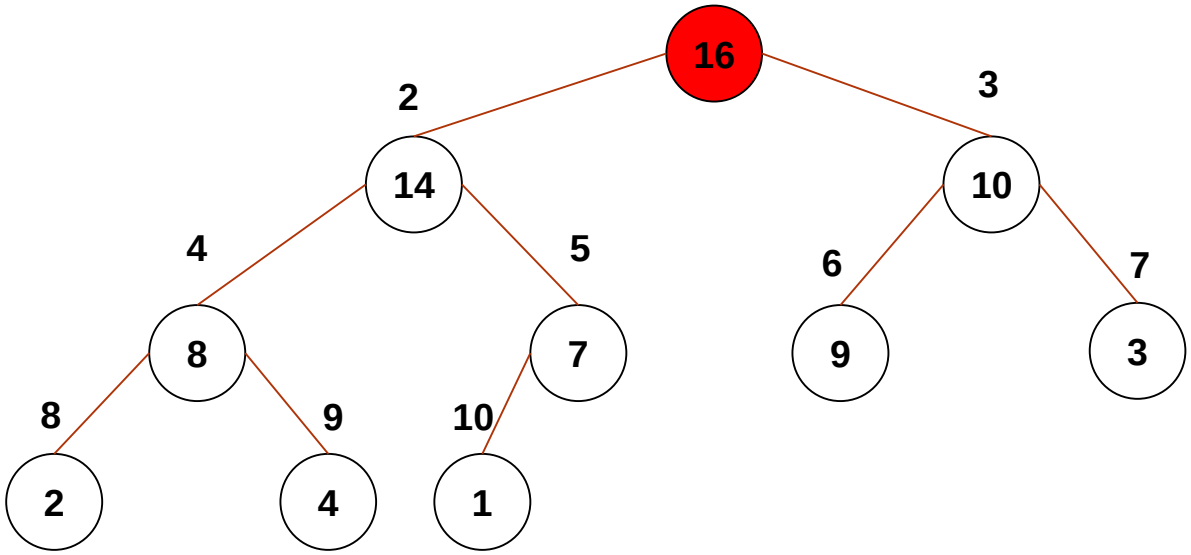
Heaps

- Exemplo:



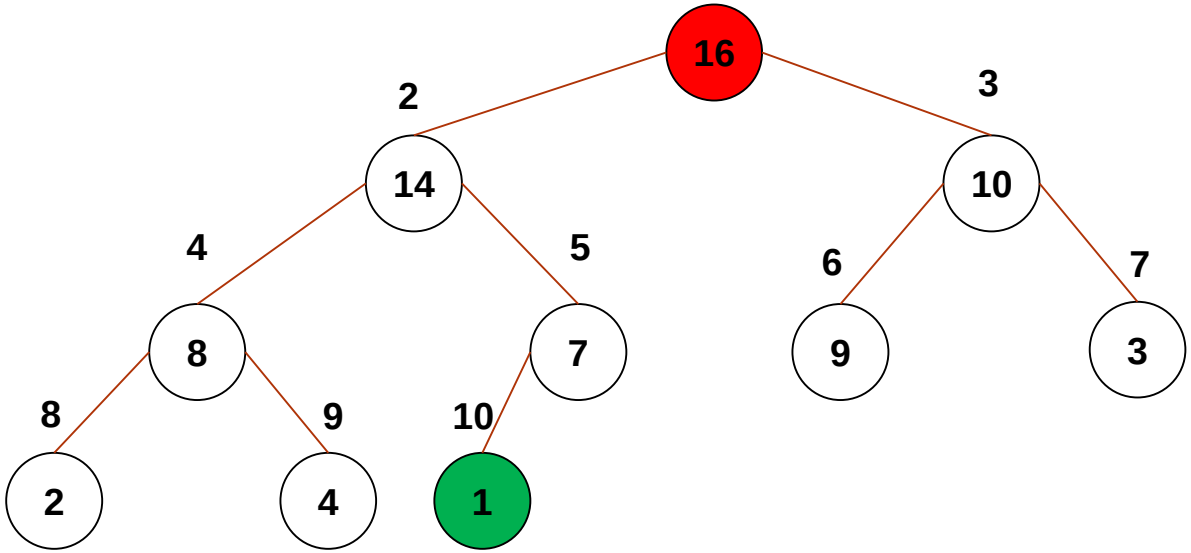
Remoção

- Como os *Heaps* são **estruturas com prioridade**, remoções ocorrem apenas na raiz.



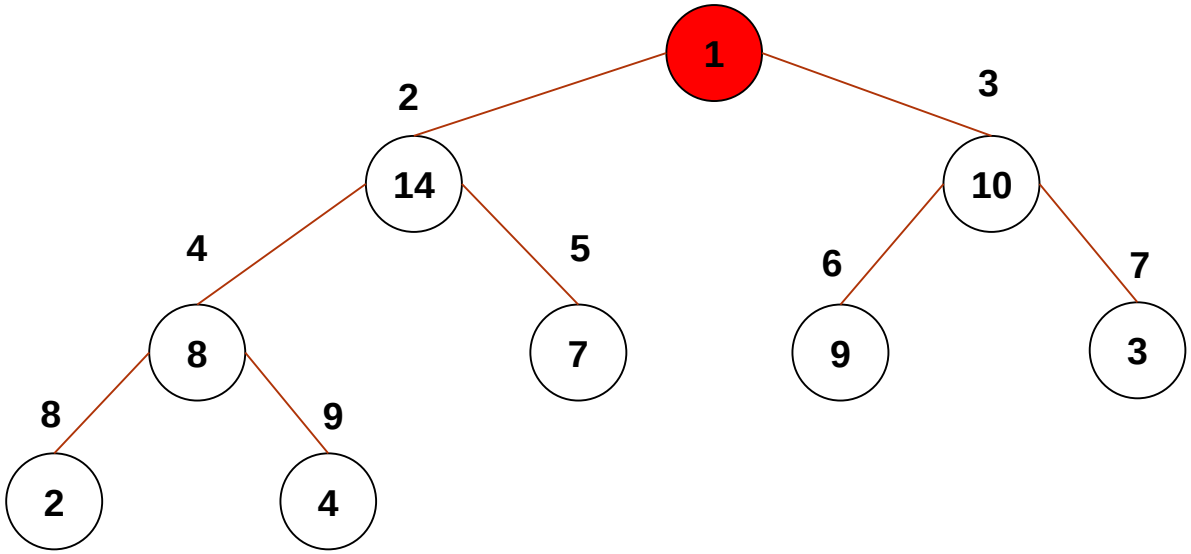
Remoção

- Porém, o ideal seria se as remoções ocorressem no último elemento.



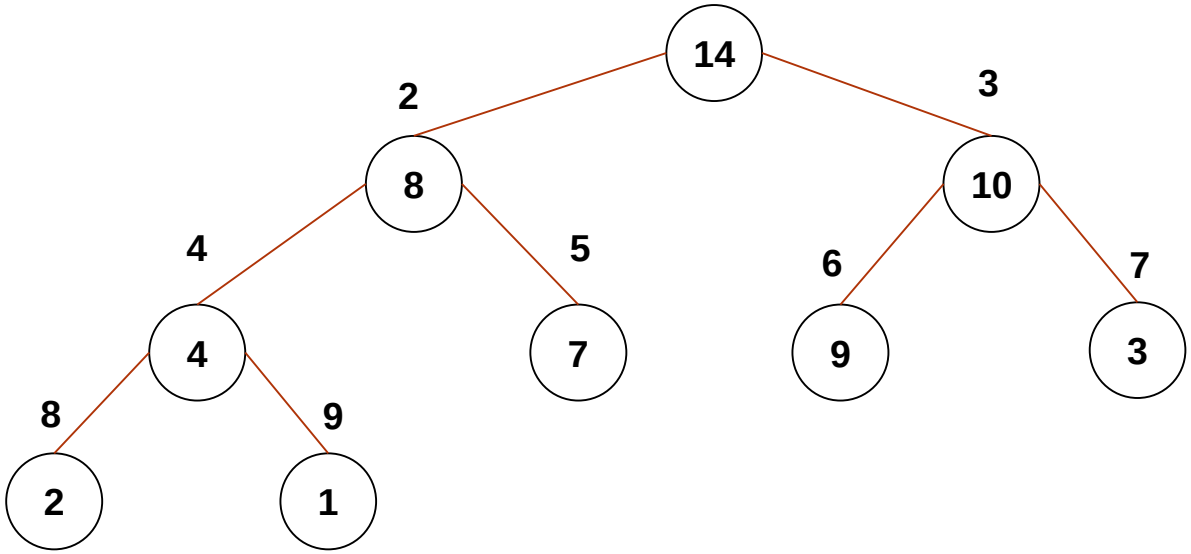
Remoção

- Para isso, basta copiar a informação do último elemento na primeira posição e usar o procedimento `maxHeapfy` para corrigir sua posição.



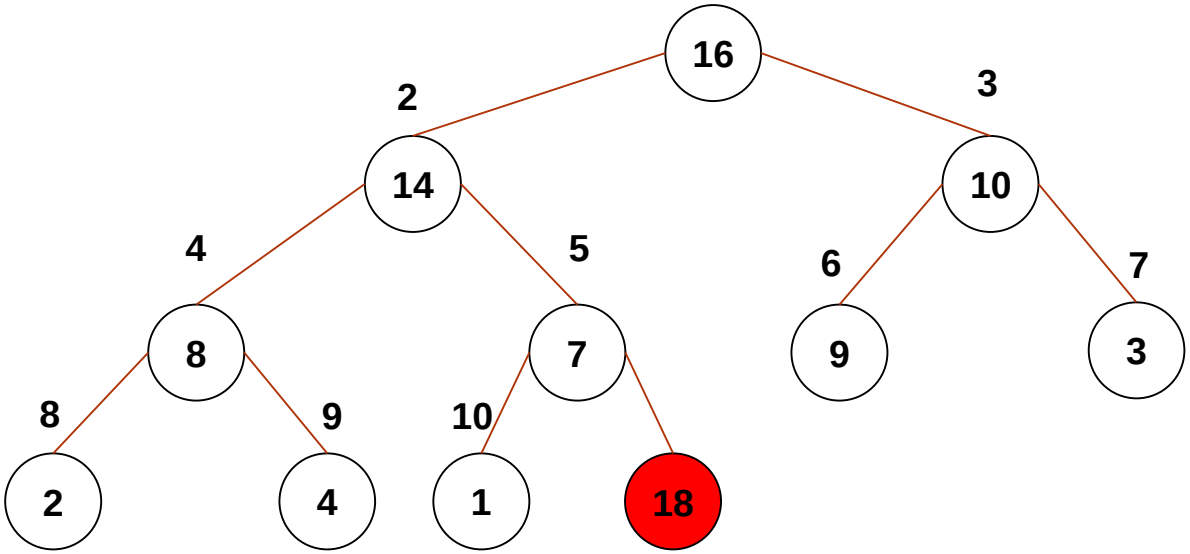
Remoção

- Após o `maxHeapfy`:



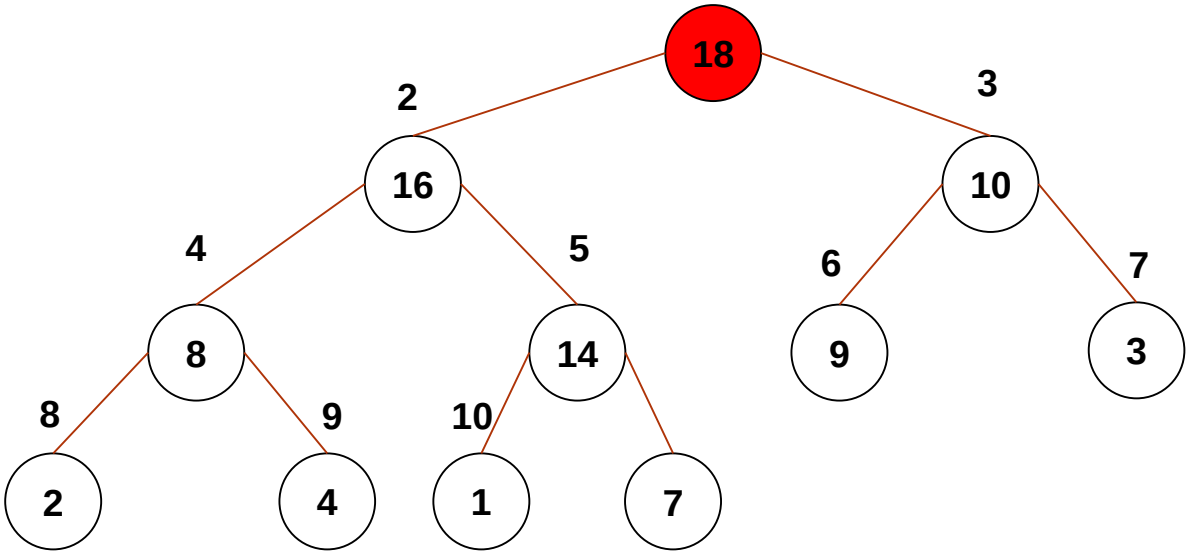
Inserção

- O novo elemento deve ser inserido após o último elemento do *Heap*.



Inserção

- Após a inserção, deve-se corrigir a posição do mesmo em relação aos seus ancestrais.



Heapsort

Heapsort

- O algoritmo *Heapsort* tem início com a construção de um *Max-Heap* para um vetor de dados *A* qualquer.

- Tendo em vista que o maior elemento estará localizado na raiz, podemos colocá-lo em sua posição correta pela troca do mesmo com o elemento da posição A[*n*].

Heapsort

- Após a modificação anterior, passamos a ignorar a última posição do vetor original.
- Isso é feito através do decremento do tamanho do *Heap*.
- Como a nova raiz pode estar violando à propriedade de um *Max-Heap*, realizamos um `maxHeapfy` na raiz.
- Esse processo é repetido no *Heap* de tamanho *n*-1 até que o *Heap* tenha tamanho igual a 1.

Heapsort

- Para os exemplos a seguir, não haverá necessidade da definição de novas estruturas básicas para a execução do *Heapsort*, pois precisamos apenas de um vetor numérico, que será representado por um **`Array`**<**`inteiro`**>.

- Para os problemas de ordenação numérica, será permitida a inserção de valores iguais no vetor (**`Array`**<**`inteiro`**>) a ser ordenado.

- No exemplo resolvido:

  - Todas as posições do vetor a ser ordenado já estarão preenchidas.

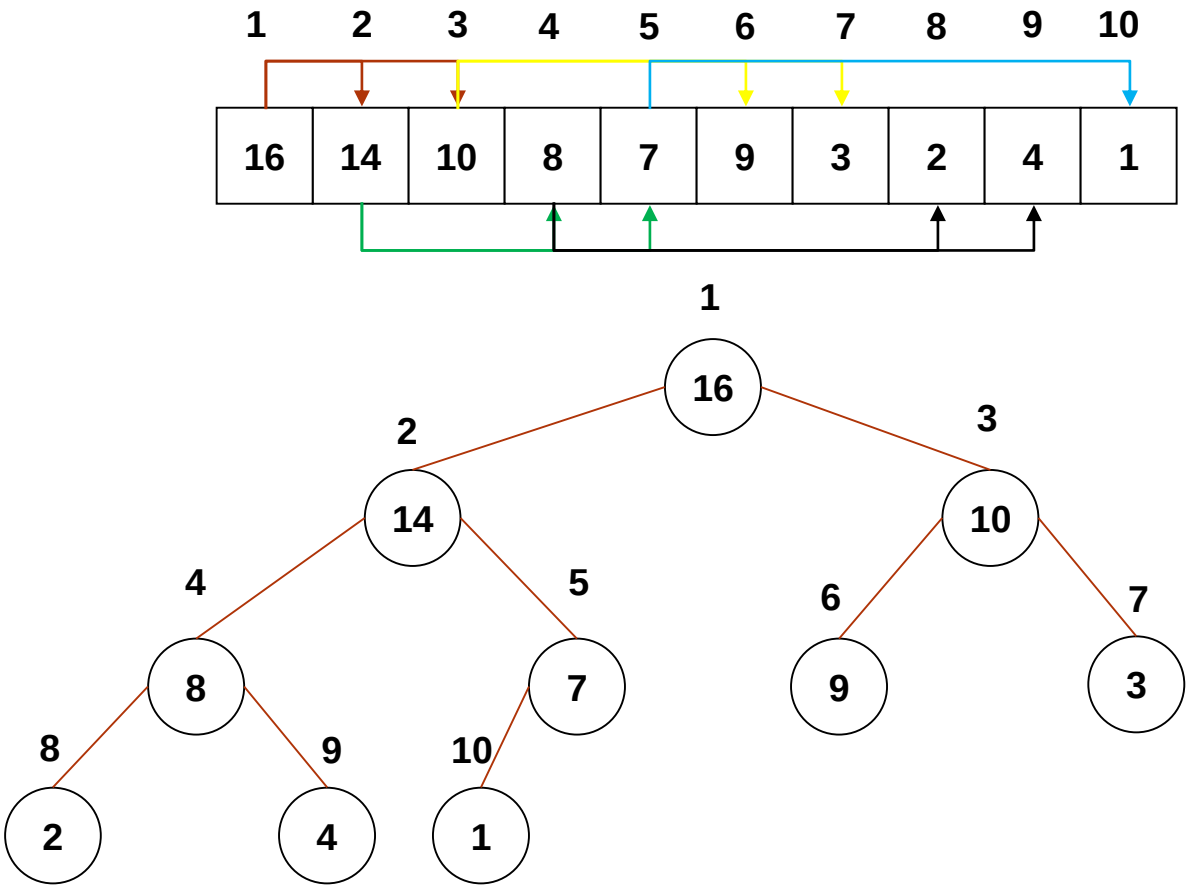
  - Usaremos duas variáveis para controle do tamanho do vetor de dados original (`tamanhoA`) e do tamanho do *Heap* (`tamanhoHeap`).

Heapsort

```
1.  //A -> Array<inteiro> que contém os dados
2.  //tamanhoA -> tamanho do vetor de dados original
3.  procedimento heapsort(A, tamanhoA)
4.      tamanhoHeap = tamanhoA
5.      construirMaxHeap(A, tamanhoHeap)
6.      para i = tamanhoA até 2
7.          trocar(A[1], A[i])
8.          tamanhoHeap = tamanhoHeap - 1
9.          maxHeapfy(A, 1, tamanhoHeap)
```

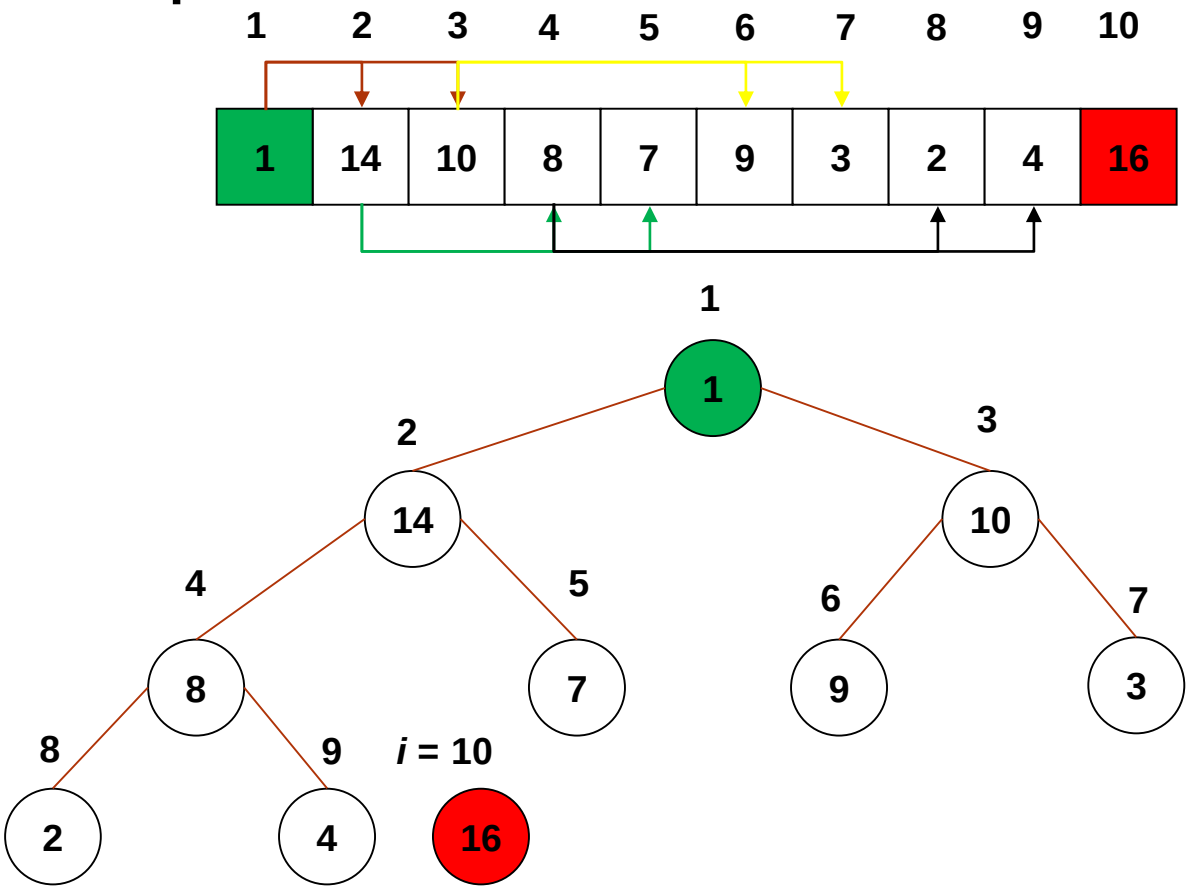
Heapsort

- Exemplo:



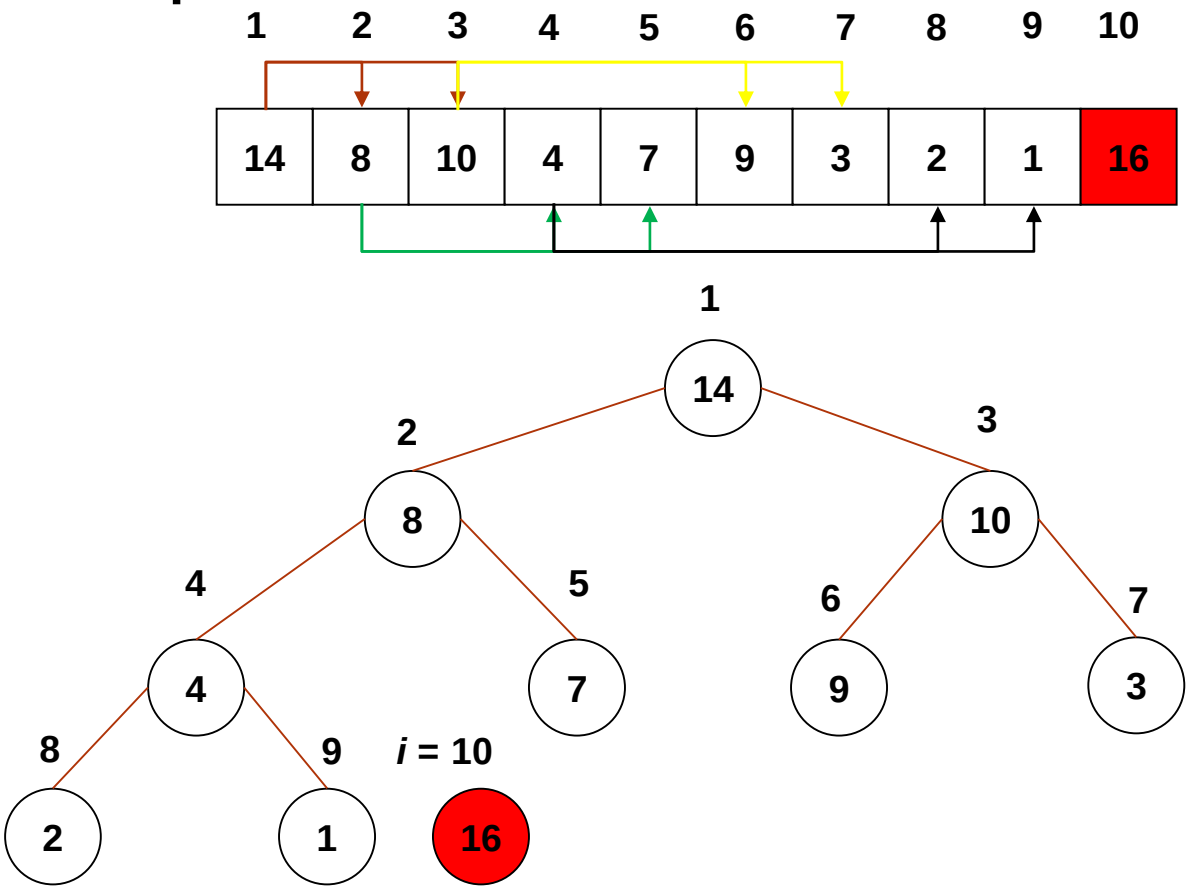
Heaps

- Exemplo:



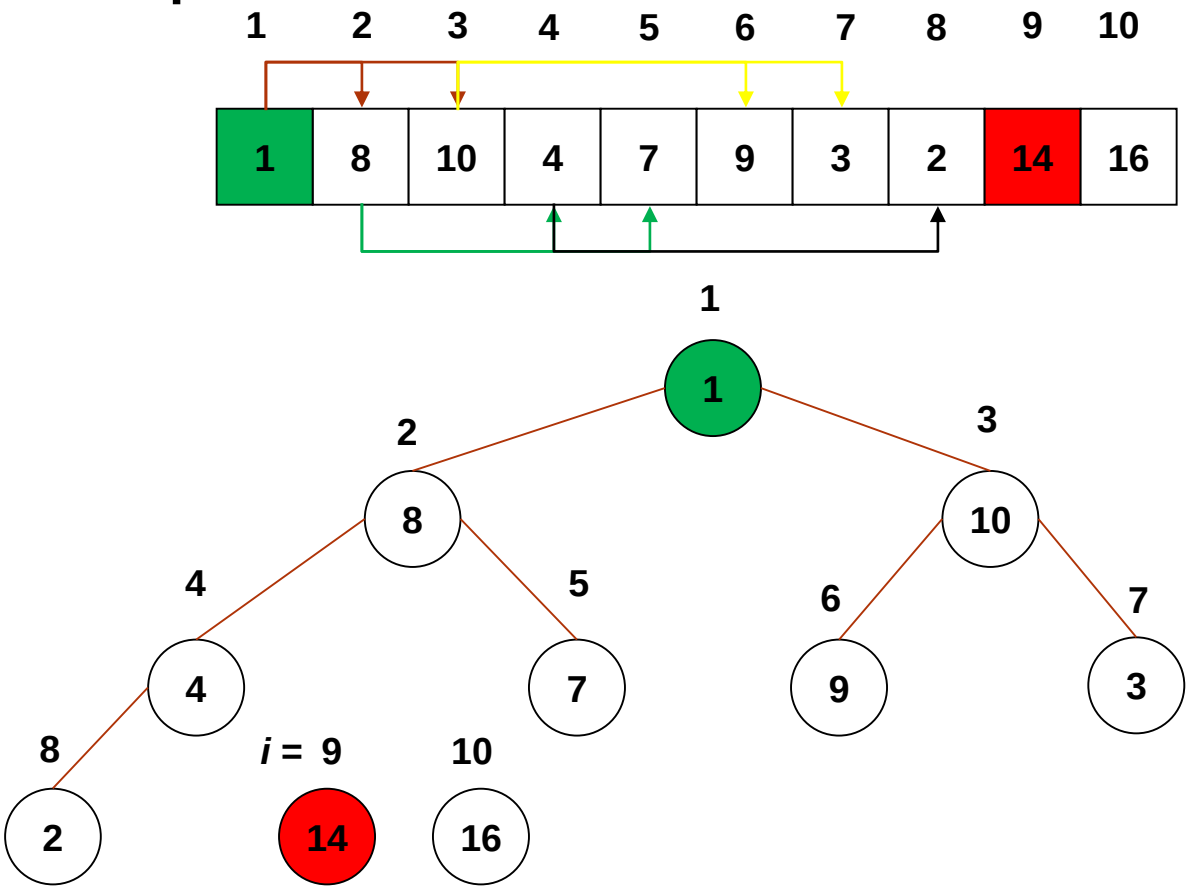
Heaps

- Exemplo:



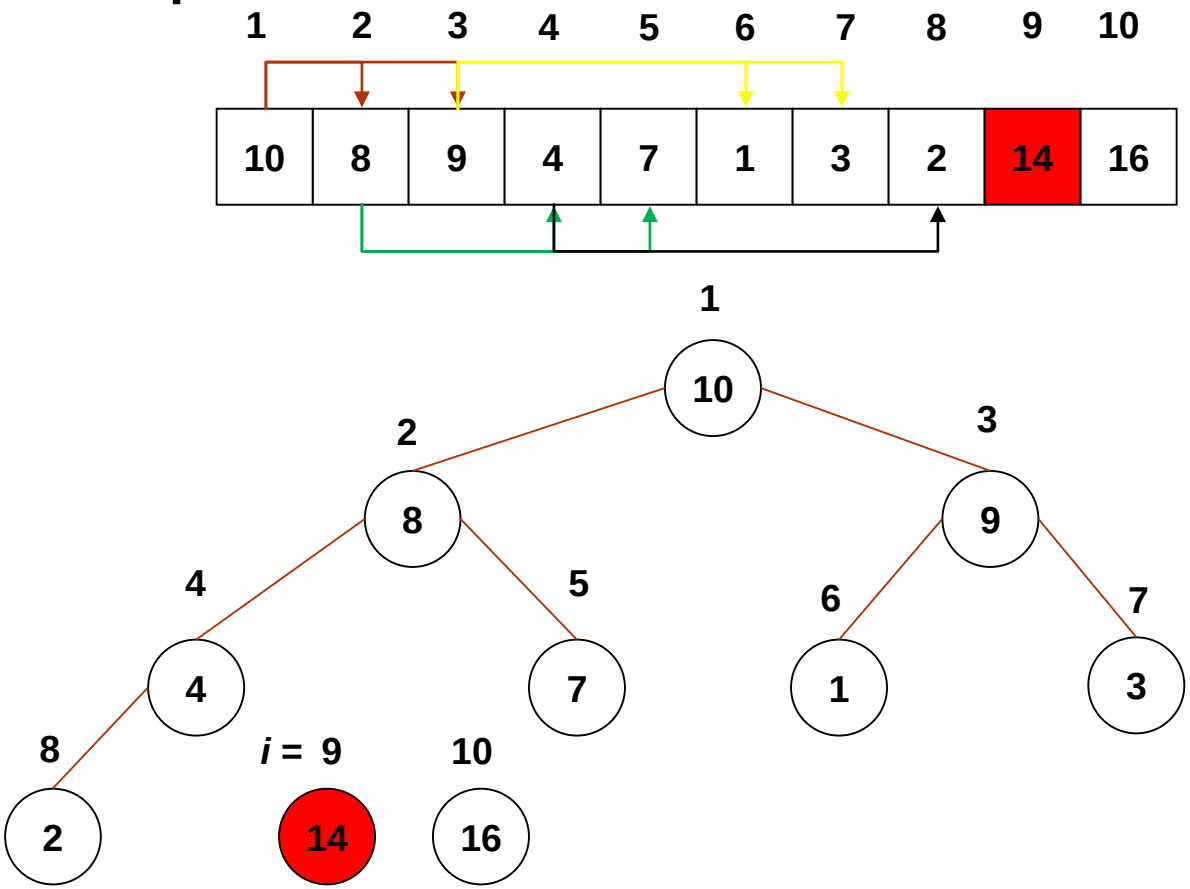
Heaps

- Exemplo:



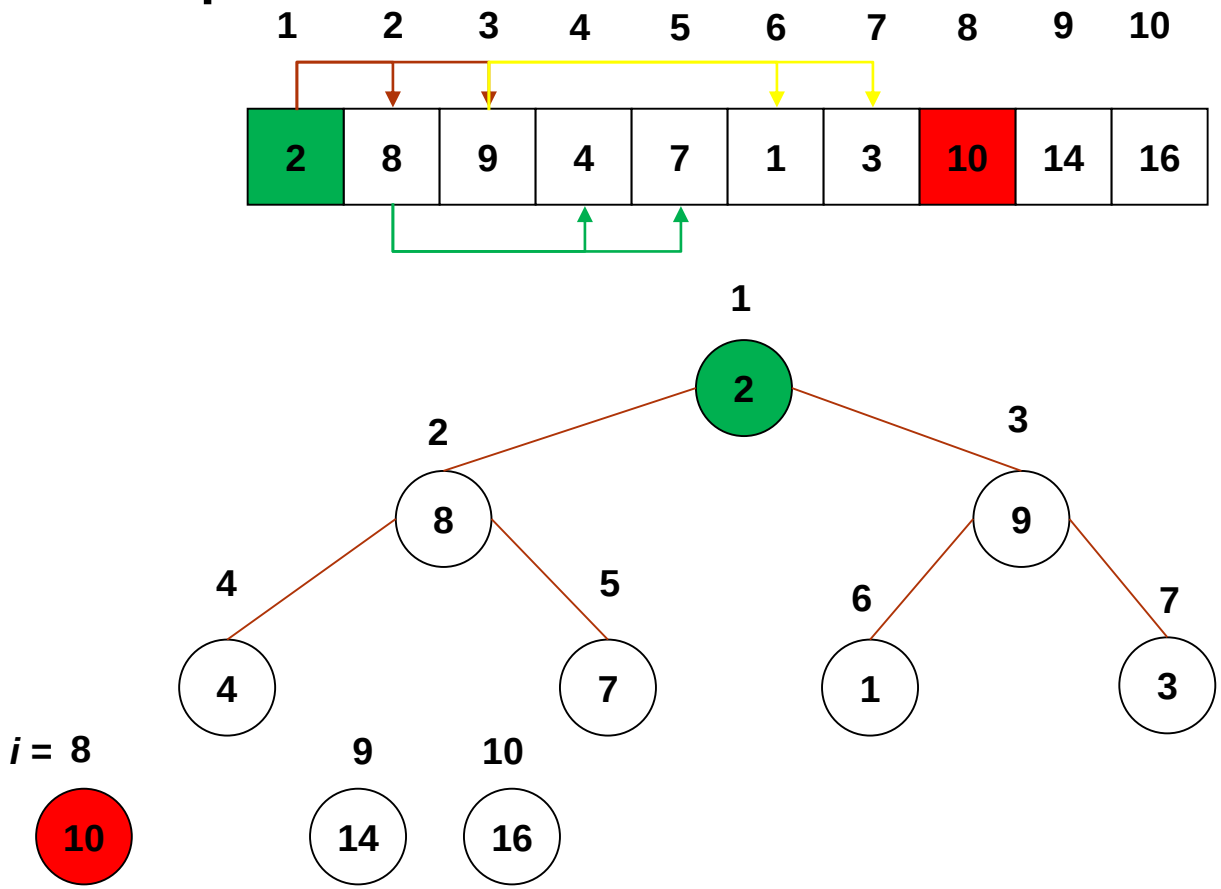
Heaps

- Exemplo:



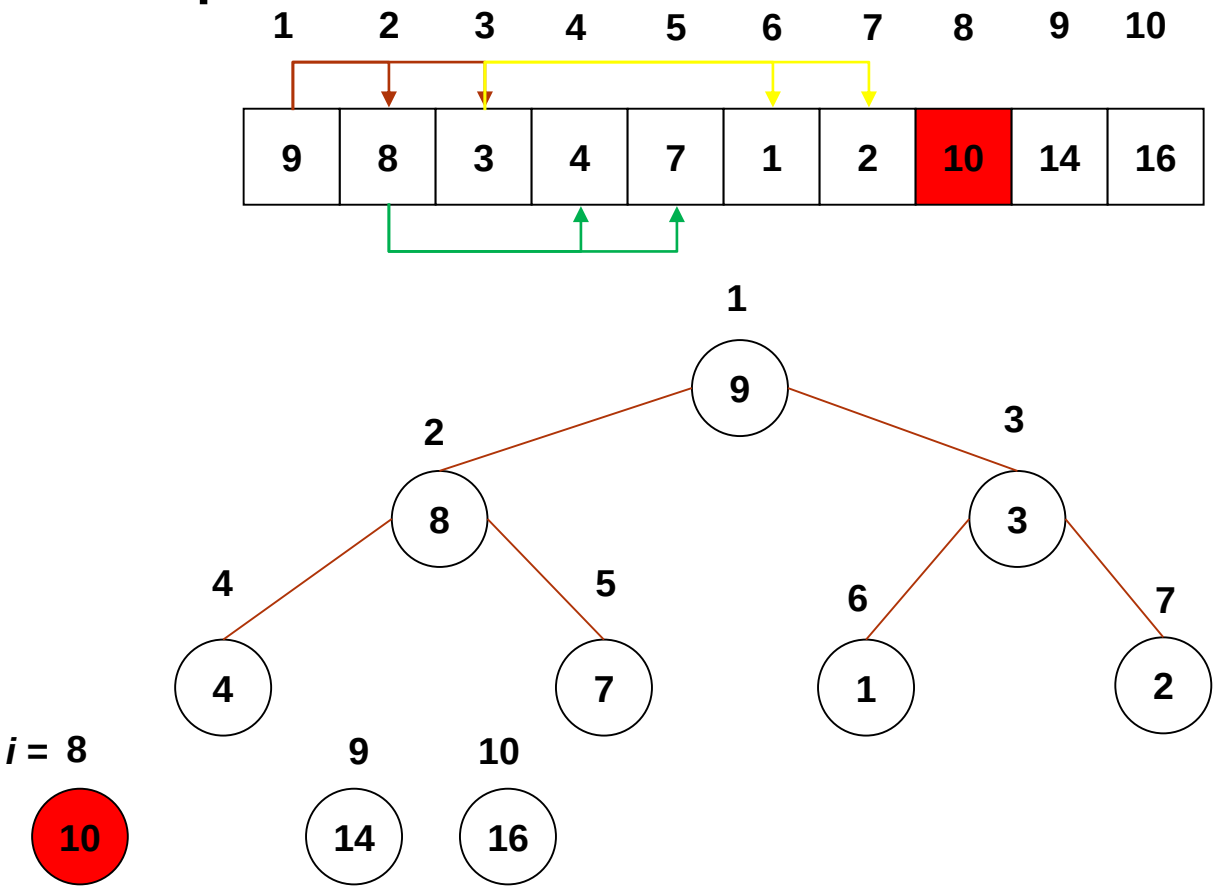
Heaps

- Exemplo:



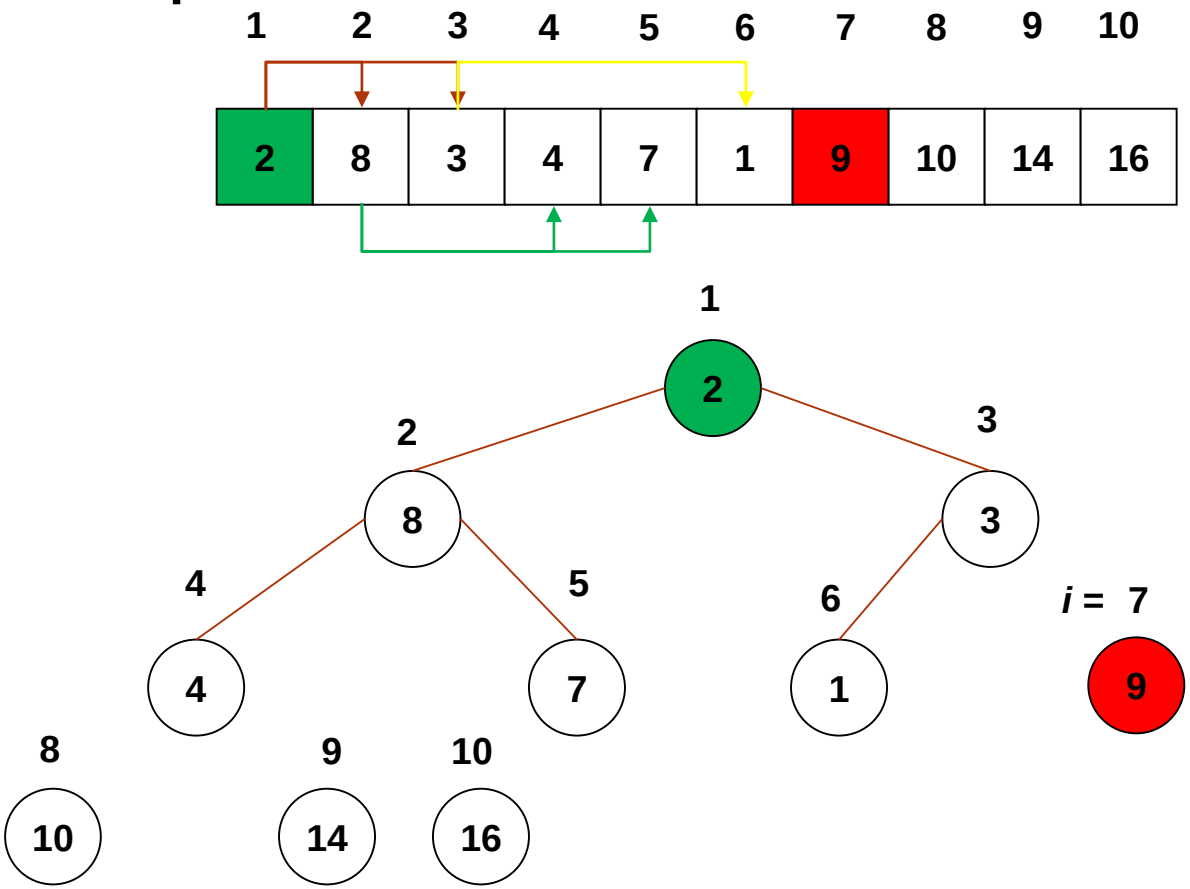
Heaps

- Exemplo:



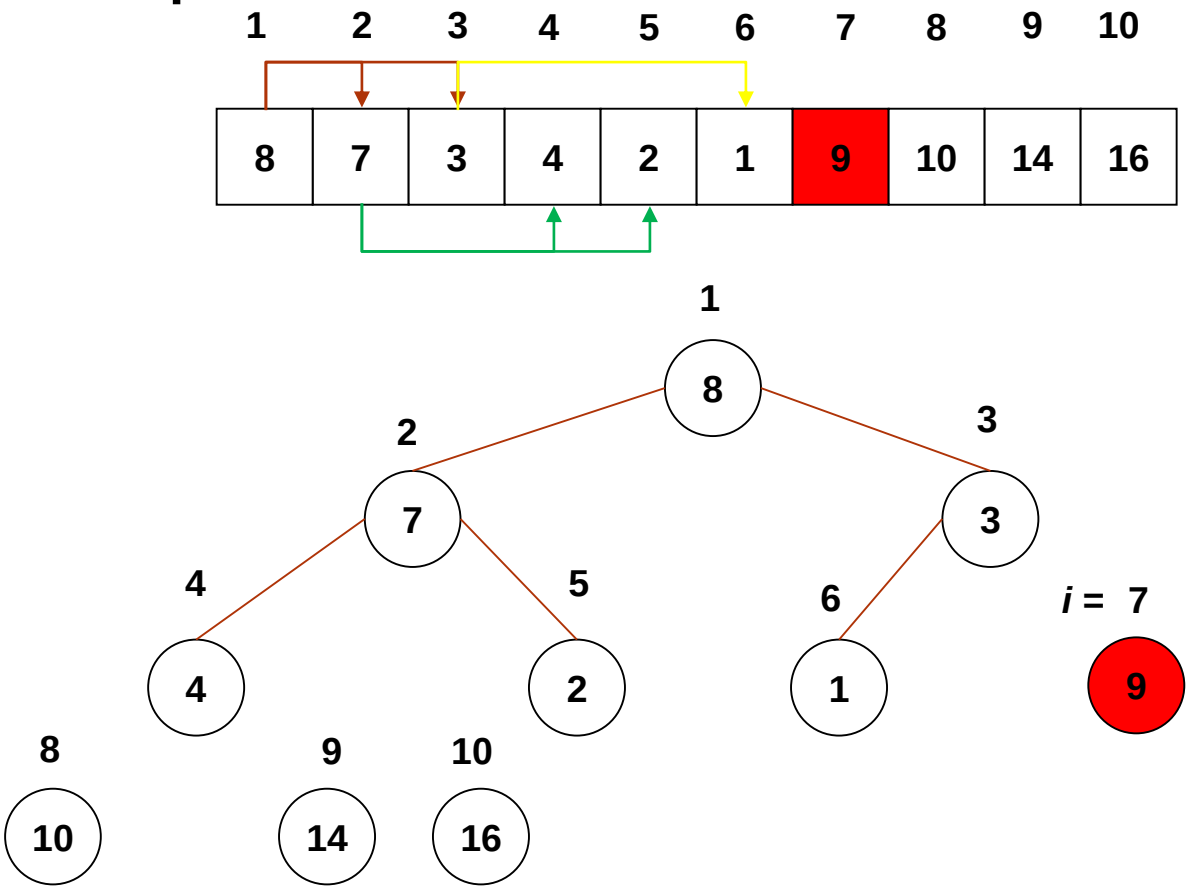
Heaps

- Exemplo:



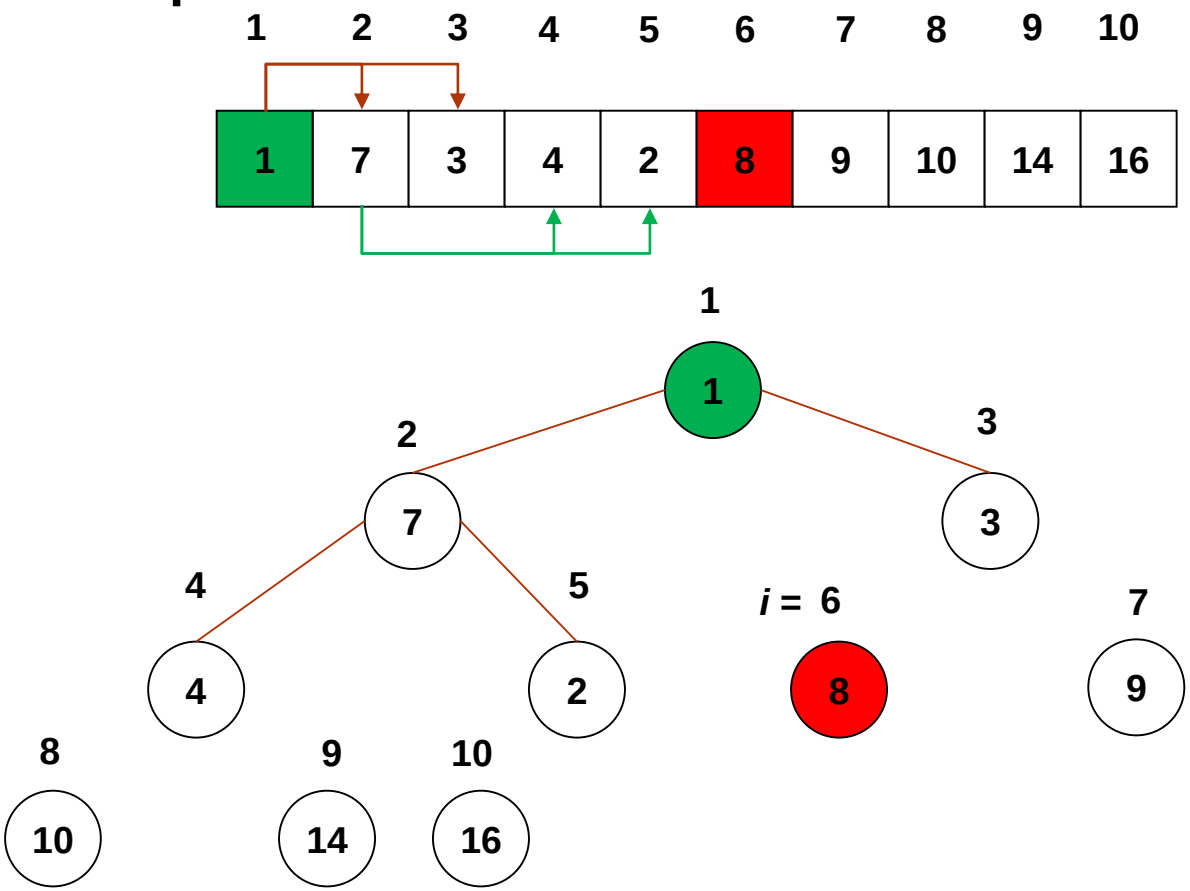
Heaps

- Exemplo:



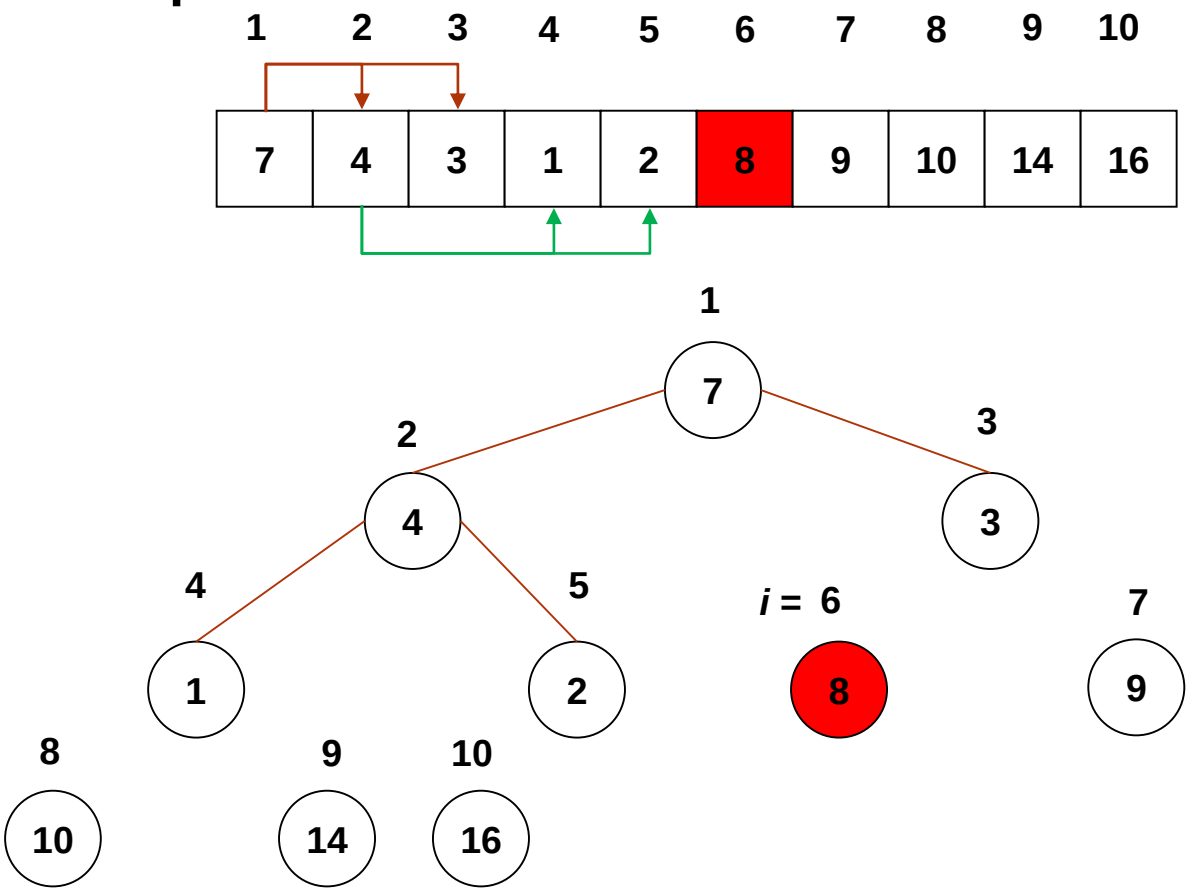
Heaps

- Exemplo:



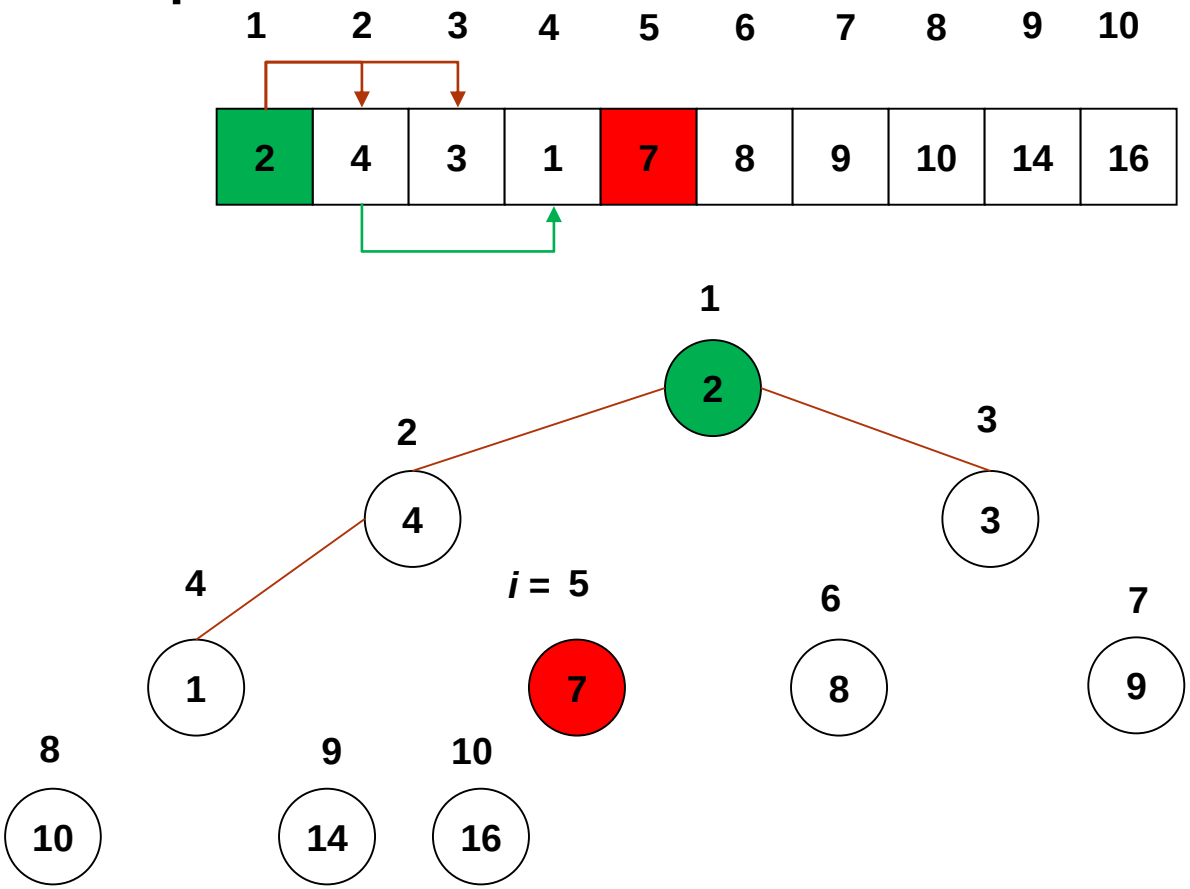
Heaps

- Exemplo:



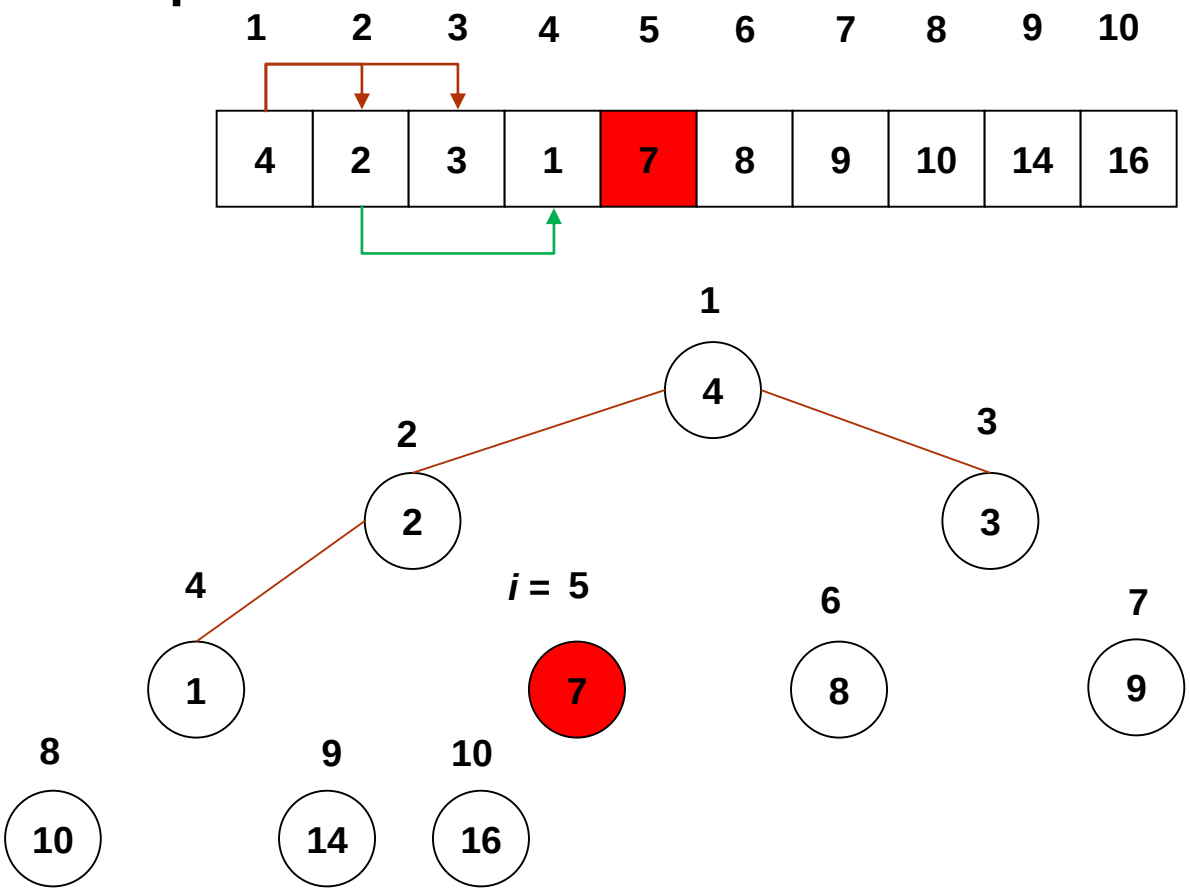
Heaps

- Exemplo:



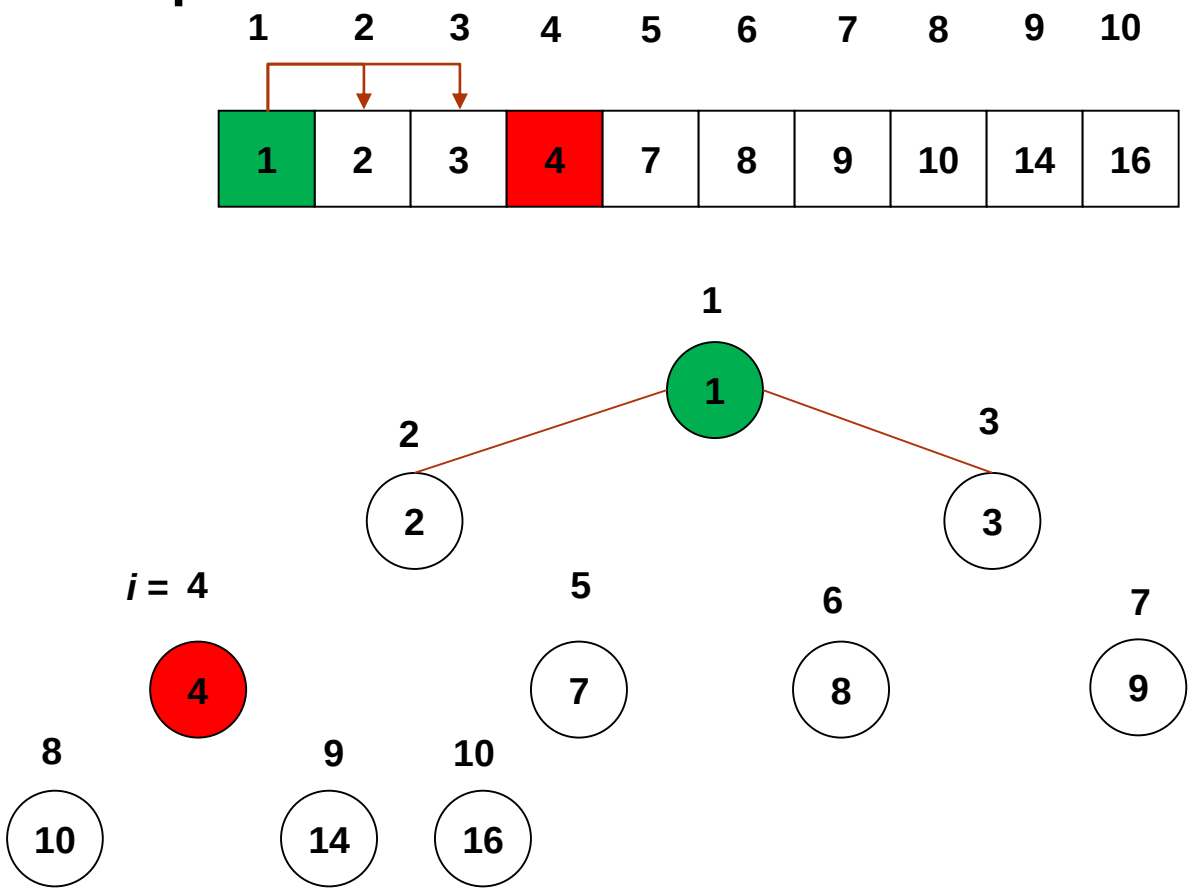
Heaps

- Exemplo:



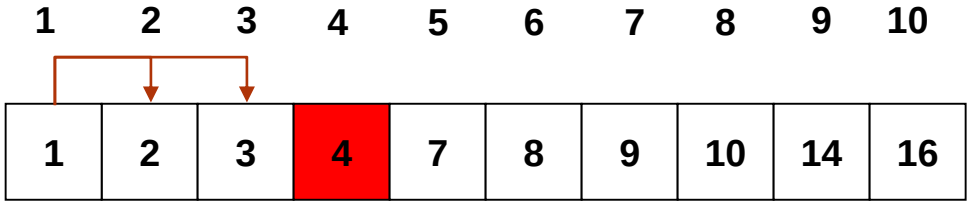
Heaps

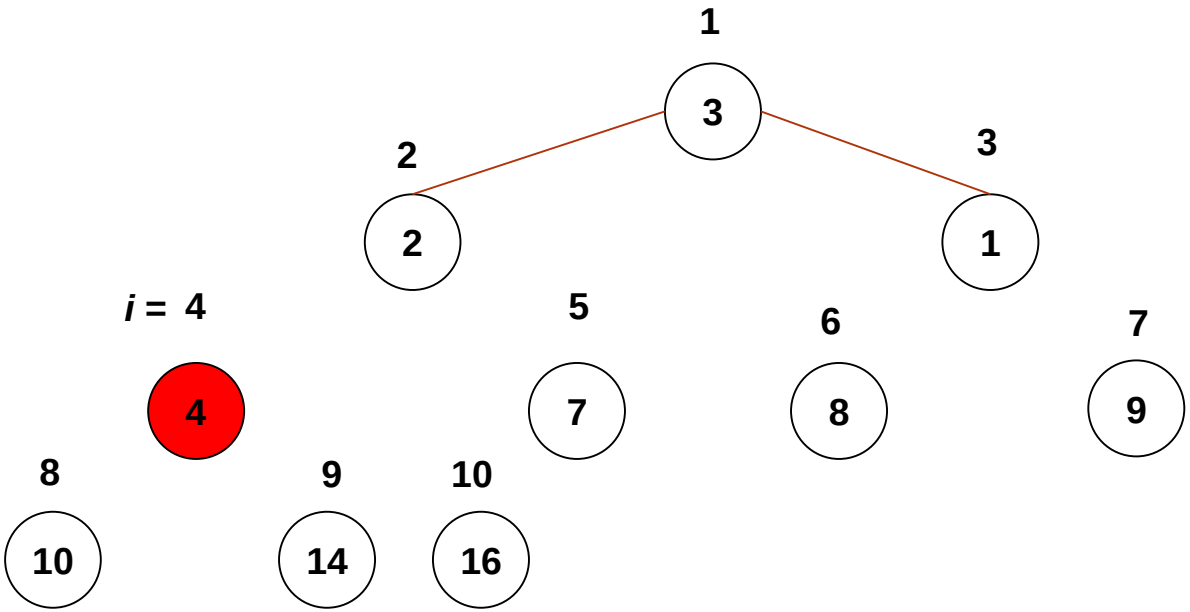
- Exemplo:



Heaps

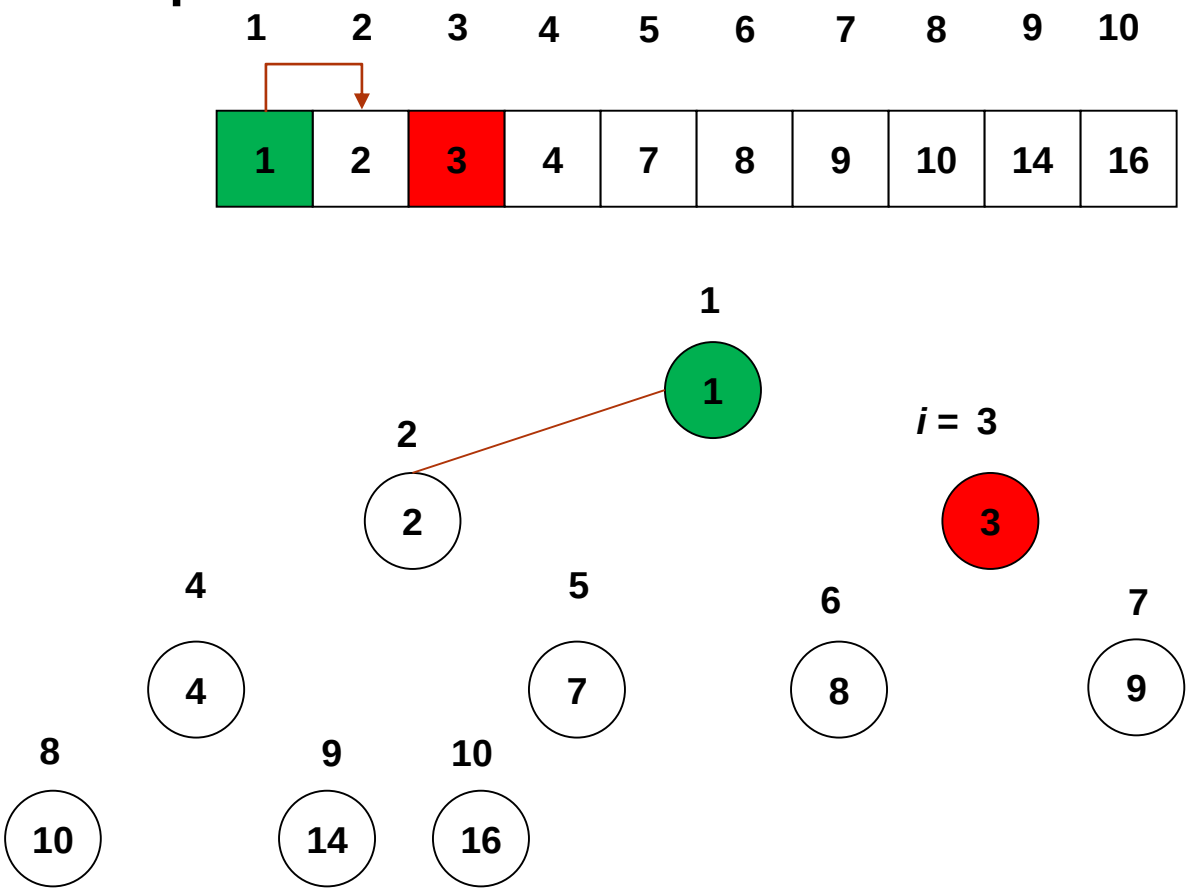
- Exemplo:





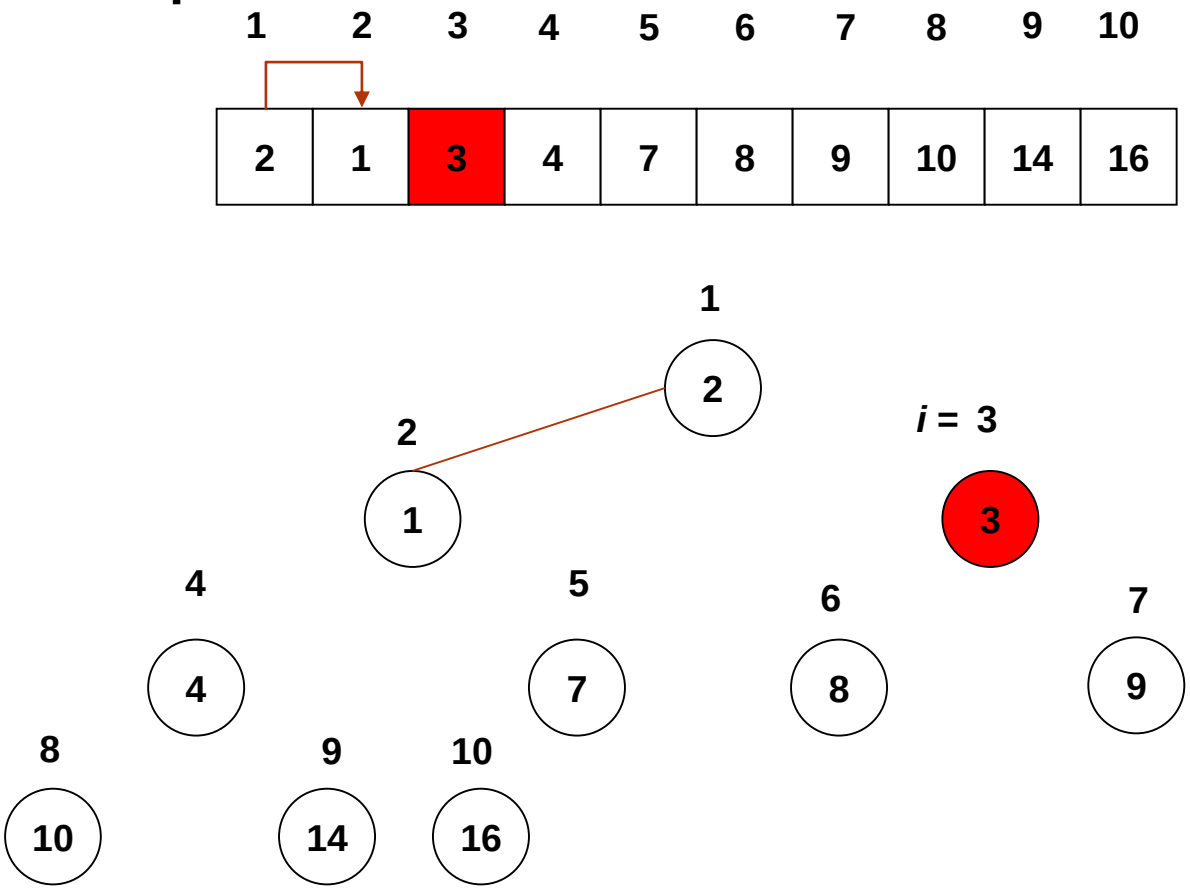
Heaps

- Exemplo:



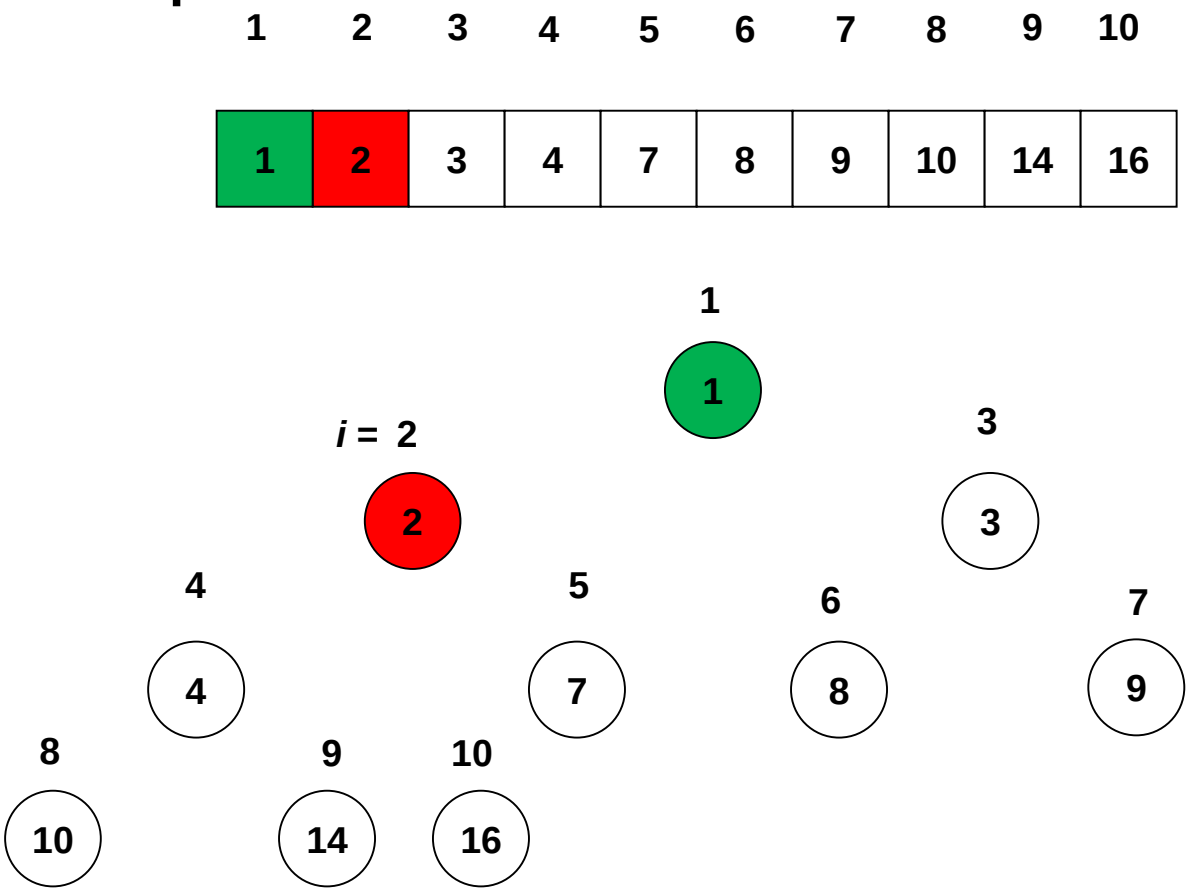
Heaps

- Exemplo:



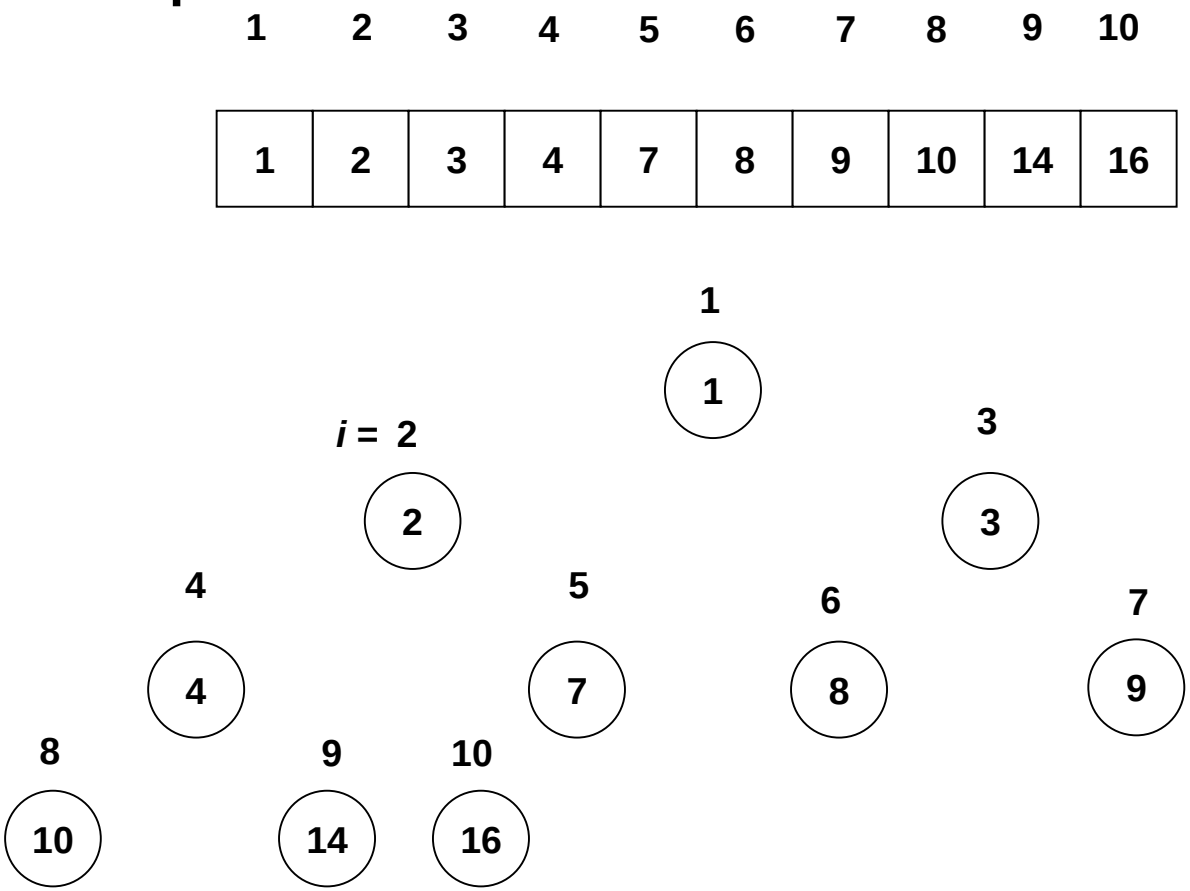
Heaps

- Exemplo:



Heaps

- Exemplo:



Listas de Prioridade

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Listas de Prioridade

- Uma lista de prioridades pode ser facilmente implementada e mantida através da estrutura de *Heaps*.

- Em uma lista de prioridades, um conjunto de *n* elementos é mantido, cada um com um valor de chave *k* **distinto**.

- Da mesma forma que nos *Heaps*, podemos ter listas de prioridade ordenadas de acordo com o valor máximo das chaves ou com o valor mínimo.

Listas de Prioridade

- Supondo que deseja-se implementar uma lista de prioridade máxima.

- Tal lista deve suportar as seguintes operações:

  - inserir(*S*, *x*): insere um novo elemento à lista;

  - obterMaximo(*S*): retorna o elemento com a maior chave em *S*;

  - extrairMaximo(*S*): remove e retorna o elemento com maior chave em *S*;

  - aumentarChave(*S*, *x*, *k*): aumenta o valor de uma chave *x* para *k*.

Listas de Prioridade

```
1. procedimento obterMaximo(A, tamanhoHeap)
2.     se tamanhoHeap > 0
3.         retorne A[1]
```

```
1. procedimento extrairMaximo(A, tamanhoHeap)
2.     maximo = NIL
3.     se tamanhoHeap < 1 então
4.         imprimir("Underflow")
5.     senão
6.         tamanhoHeap = tamanhoHeap - 1
7.         maxHeapfy(A, 1, tamanhoHeap)
8.     retorne maximo
```

Listas de Prioridade

```
1.  //i -> posição da chave a ser aumentada
2.  //k -> valor de chave aumentado para a posição i
3.  procedimento aumentarChave(A, i, k)
4.      se k < A[i] então
5.          imprimir("chave menor que a atual")
6.      senão
7.          A[i] = k
8.          pai = retornarIndicePai(i)
9.          enquanto (i > 1) e (A[pai] < A[i])
10.             trocar(A[i], A[pai])
11.             i = pai
12.             pai = retornarIndicePai(i)
```

Listas de Prioridade

```
1.  //tamanhoHeap -> tamanho do Heap
2.  //tamamhoA -> numéro máximo de posições no vetor de dados
3.  //k -> valor da chave a ser inserida
4.  procedimento inserir(A, k, tamanhoHeap, tamamnhoA)
5.      se tamanhoHeap == tamanhoA
6.          imprimir("Overflow")
7.      senão
8.          tamanhoHeap = tamanhoHeap + 1
9.          A[tamanhoHeap] = -Inf          //Inf é o valor infinito
10.         aumentarChave(A, tamamnhoHeap, k)
11.     retorne tamanhoHeap
```

Referências

- CORMEN, H. T.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms, 3rd ed., *Boston: MIT Press*, 2009.

- FEOFILOFF, Paulo. Algoritmos em Linguagem C. Editora Campus/Elsevier, 2009.

Heapsort

Algoritmos e Estruturas de Dados
Prof. Dr. Luciano Demétrio Santos Pacífico
{luciano.pacifico@ufrpe.br}

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO