A estrutura de *heap*

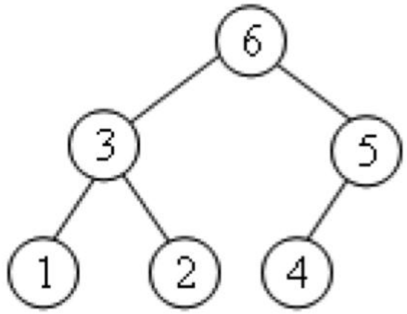
- O *heap* é um caso particular de árvore binária balanceada, que possui as seguintes propriedades:
  - O valor de cada nó não é menor (maior) do que os valores de seus filhos. Neste caso, o *heap* é conhecido como ***heap* máximo** (mínimo).
  - As folhas do último nível estão nas posições mais à esquerda.

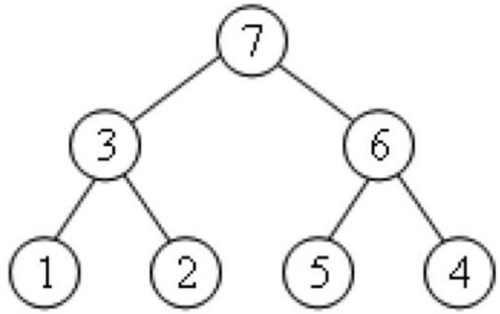
- Note que num *heap* máximo, a raiz contém o maior elemento e num *heap* mínimo, a raiz contém o menor elemento.

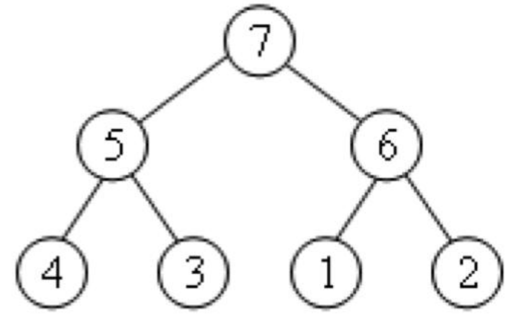
- Note também que a altura de um *heap* é $O(\log n)$, onde n é o número de nós da árvore.

A estrutura de *heap*

Exemplos:

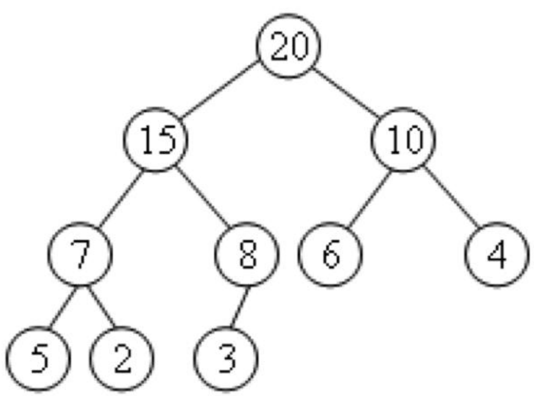






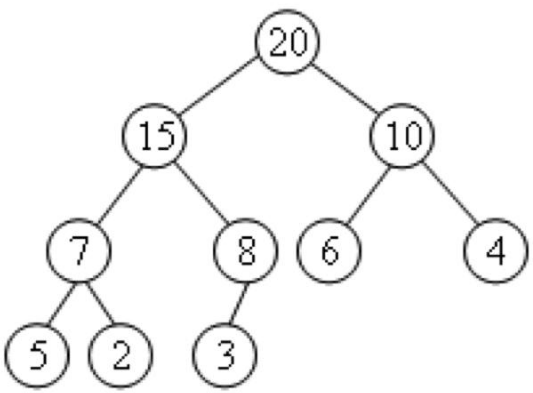
heaps diferentes, construídos com os mesmos elementos.

- Um heap pode ser representado por um vetor:



20	15	10	7	8	6	4	5	2	3
0	1	2	3	4	5	6	7	8	9

A estrutura de *heap*

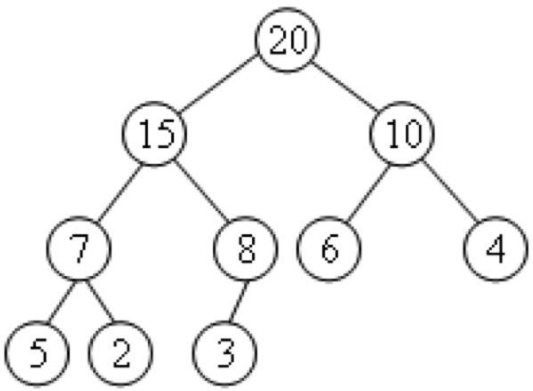
- Representado por um vetor, os índices correspondentes ao pai, filho esquerdo e filho direito de um nó que aparece na posição ***i*** do vetor podem ser calculados facilmente como:
  - pai(i) = piso[(i+1)/2] - 1
  - filhoEsquerdo(i) = 2i + 1
  - filhoDireito(i) = 2i + 2
- 

20	15	10	7	8	6	4	5	2	3
0	1	2	3	4	5	6	7	8	9

Considere, por exemplo, o **nó 8**, cujo índice na representação de vetor é ***i* = 4**.
pai(4) = 1 (nó 15); filhoEsquerdo(4) = 9 (nó 3); filhoDireito(4) = 10 (não existe).

A estrutura de *heap*

- Note também que se um heap contem n nós, na representação de vetor suas folhas terão índices: k, k+1, ..., n-1, onde k = piso[n / 2].



20	15	10	7	8	6	4	5	2	3
0	1	2	3	4	5	6	7	8	9

No exemplo: n = 10, k = 5. Logo, os índices das folhas são: 5, 6, 7, 8, 9.

- Vamos examinar como construir um ***heap* máximo** a partir de um vetor v de n elementos não ordenados. A ideia do algoritmo é construir o *heap* de baixo para cima, ou seja, das folhas para a raiz.

A estrutura de *heap*

- Note que qualquer folha é um *heap* máximo.
- O algoritmo de construção de *heap* máximo chama para os demais nós o procedimento **HeapMax(v, n, i)**, onde v é o vetor de entrada, n é o número de elementos de v e i é o índice do próximo nó a ser incluído no *heap*. Notar que i deve variar de (n/2)-1 até 0, pois as folhas têm índices de n/2 até n-1.
- Quando **HeapMax(v, n, i)** é chamado, os nós filhoEsquerdo(i) e filhoDireito(i) são raízes de *heaps* máximos, mas v[i] pode ser menor que seus filhos, violando a propriedade de *heap* máximo.
- O algoritmo faz com que v[i] "desça" no *heap* até que v[i] seja raiz de um *heap* máximo.

A estrutura de *heap*

Exemplo:

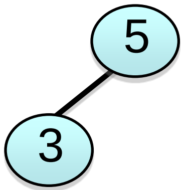
2	**7**	**5**	**8**	**9**	**3**

n = 6. Logo, as folhas deverão ficar nas posições: 3, 4, 5.

Seja i = 2:

v[2] = 5; filhoEsq(2) = 2*2+1 = 5; filhoDir(2) = 2*2+2 = 6 (vazio)

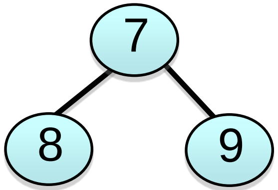
Logo, temos:



Seja i = 1:

v[1] = 7; filhoEsq(1) = 2*1+1 = 3; filhoDir(1) = 2*1+2 = 4

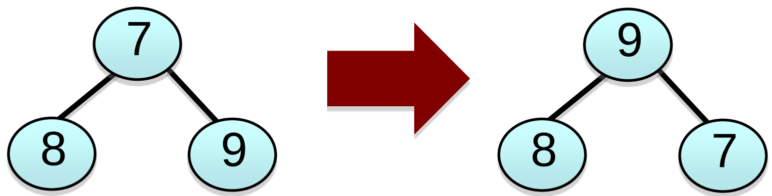
Logo, temos:



Neste caso, v[i] (raiz) viola a propriedade de *heap* máximo.

A estrutura de *heap*

Então:



Essa transformação é feita **recursivamente**.

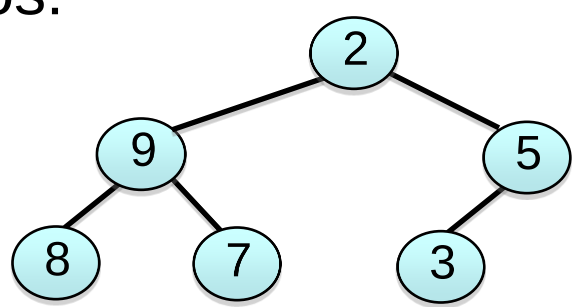
Neste ponto, existem 2 *heaps* máximos:

2	**9**	**5**	**8**	**7**	**3**

Seja i = 0:

v[0] = 2; filhoEsq(0) = 2*0+1 = 1; filhoDir(0) = 2*0+2 = 2

Logo, temos:



E o nó 2 deverá "descer" até que a condição de ***heap* máximo** seja satisfeita.

A estrutura de *heap*

Algoritmo:

```
void ConstruirHeapMaximo(int v[], int n)
{
  int i;

  // Construir heap
  for (i = (n/2)-1; i >= 0; i--)
    HeapMax(v,n,i);

  // Mostrar a representação de vetor do heap
  printf("\nHeap: ");
  for (i = 0; i < n; i++)
    printf("%d ",v[i]);
}
```

A estrutura de *heap*

```
void HeapMax(int v[], int n, int i)
{
  int e,d,maior,temp;

  e = filhoEsquerdo(i);
  d = filhoDireito(i);
  if ((e < n) && (v[e] > v[i]))
    maior = e;
  else
    maior = i;
  if ((d < n) && (v[d] > v[maior]))
    maior = d;
  if (maior != i)
  {
    temp = v[i];
    v[i] = v[maior];
    v[maior] = temp;
    HeapMax(v,n,maior);
  }
}
```

O algoritmo *HeapSort*

- Uma das aplicações importantes de *heaps* é na ordenação de vetores. O algoritmo *HeapSort* começa construindo um *heap* máximo para o vetor de entrada v, de n elementos.
- Como a raiz da árvore construída (ou seja, v[0]) é o maior elemento do vetor, este elemento pode ser colocado em sua posição correta no vetor ordenado por meio de uma operação de troca (ou seja, v[n-1] ⇔ v[0]).
- Com isso, podemos descartar um nó e transformar o vetor v[0 .. n-2] em um novo *heap*, o que pode ser feito facilmente observando que os filhos do nó descartado são raízes de *heaps* máximos. Assim, basta chamar **HeapMax** para inserir a nova raiz (v[0]) no novo *heap* (de n-1 nós).
- Repetindo este procedimento até que o novo *heap* tenha tamanho igual a 1 teremos, ao final, o vetor ordenado.

O algoritmo *HeapSort*

```
void HeapSort(int v[], int n)
{
  int i;

  ConstruirHeapMaximo(v,n);

  for (i = n-1; i > 0; i--)
  {
    Troca(&v[0],&v[i]);
    HeapMax(v,i,0);
  }
}
```

- A complexidade de tempo do algoritmo *HeapSort* é O(n log n), pois a construção do *heap* máximo inicial é feita em O(n) e cada uma das n-1 chamadas a **HeapMax** executa em tempo O(log n).
- Observe que o algoritmo *HeapSort* alcança este limite no pior caso, enquanto que o algoritmo *QuickSort* o alcança na média.

O algoritmo *HeapSort*

- Pode-se mostrar que qualquer algoritmo de ordenação que se baseia apenas em comparações entre os elementos de entrada para ordenar um vetor de n elementos deve efetuar O(n log n) comparações no pior caso.
- Portanto, o algoritmo *HeapSort* é ótimo e não existe um algoritmo de ordenação basedo em comparações que seja mais rápido, a menos de um fator constante.
- Algoritmos que usam outros recursos que não apenas a comparação podem ser mais rápidos, como o **algoritmo de ordenação por contagem**, que ordena um vetor v de n elementos em tempo O(n), pressupondo que cada um dos n elementos é um inteiro no intervalo [0, k] para algum k.

Ver o **Capítulo 8** do livro: CORMEN, T.H.; LEISERSON, C.E.; RIVEST, R.L.; STEIN, C. *Algoritmos: Teoria e Prática*, Rio de Janeiro: Elsevier, 2002.

Fila de prioridades eficiente

- Outra utilização importante de *heaps* é como uma fila de prioridades eficiente.
- Como vimos, uma fila de prioridades é uma estrutura de dados para armazenar um conjunto S de elementos, cada um dos quais está associado a um valor denominado **chave**, que estabelece a prioridade do elemento.
- Dependendo de como são selecionados os elementos, a fila de prioridades pode ser:
  - máxima (seleciona-se o elemento de maior chave), ou
  - mínima (seleciona-se o elemento de menor chave).

Fila de prioridades eficiente

- Para uma fila de prioridades máxima, podemos considerar as seguintes operações:
  - **Maximo**(S) - retorna o elemento de S com a maior chave;
  - **ExtrairMaximo**(S) - exclui e retorna o elemento de S com a maior chave;
  - **AumentarChave**(S, k, x) - altera o valor da chave do k-ésimo elemento de S (que é menor do que x) para o valor x;
  - **Inserir**(S, x) - insere o elemento x no conjunto S.

- Se a fila de prioridades for implementada como um *heap* máximo, essas operações podem ser feitas de forma eficiente.

Fila de prioridades eficiente

- Considerando S como um vetor v de n elementos inteiros:

```
int Maximo(int v[], int n)
{
  return v[0];
}
```

```
int ExtrairMaximo(int v[], int n)
{
  int max;

  if (n < 1)
    Erro("O heap está vazio");
  max = v[0];
  v[0] = v[n-1];
  HeapMax(v,n-1,0);
  return max;
}
```

Fila de prioridades eficiente

```
void AumentarChave(int v[], int n, int k, int x)
{
  int max;

  if (x < v[k])
    Erro("A nova chave é menor do que a atual");
  v[k] = x;
  while ((k > 0) && (v[pai(k)] < v[k]))
  {
    Troca(&v[k],&v[pai(k)]);
    k = pai(k);
  }
}
```

```
void Inserir(int v[], int n, int x)
{
  v[n] = -INFINITO;
  AumentarChave(v, n+1, n, x);
}
```

Fila de prioridades eficiente

Análise dos algoritmos:

- **Maximo**: será feita em O(1).
- **ExtrairMaximo**: será feita em O(log *n*), pois executa **HeapMax** apenas uma vez.
- **AumentarChave**: o aumento de v[*k*] pode afetar a condição de *heap* máximo. Assim, é preciso percorrer o caminho deste nó até a raiz para encontrar o lugar apropriado deste novo valor (como na ordenação por inserção). Portanto, esta operação será feita em O(log *n*), pois o maior caminho de um nó até a raiz tem comprimento O(log *n*).
- **Inserir**: será feita em O(log *n*), pois executa a operação **AumentarChave** exatamente uma vez.
- Portanto: todas as operações podem ser executadas em tempo, no máximo, O(log *n*).