AULA 4: ALGORITMOS RECURSIVOS (HEAPSORT)

Fonte: https://pixabay.com/pt/vectors/computador-port%c3%a1til-infogr%c3%a1fico-6087062/

Caro(a) aluno(a). Bem-vindo(a) ao nosso capítulo sobre algoritmos recursivos. Neste capítulo estudaremos o algoritmo *heap-sort*. Traremos exemplos que permitam identificar como realizar implementação nas linguagens de programação JAVA e PYTHON. Vamos lá?

4.1 O que são algoritmos recursivos?

O que é recursão? Não é um algoritmo real, mas um método usado em alguns algoritmos! Na ciência da computação e em todas as áreas que envolve codificação, a recursão é um método no qual as etapas são executadas em repetição para resolver um problema. Você pode estar pensando em como isso é diferente da iteração. Pense na iteração como algo que leva a uma solução e a recursão como a divisão de um

problema em instâncias menores de si mesmo para chegar à solução. Em outras palavras, pense na recursão como um método que simplifica o problema em subproblemas menores. Vejamos um exemplo de algoritmo de recursividade abaixo (Algoritmo 4.1):

```
#include <iostream>
using namespace std;

int fact(int n)
{
  if (n <= 1)
    return 1;
  else
    return n*fact(n-1);
}

int main() {
  cout << fact(5) << endl;
}
```

Algoritmo 4.1: Fatorial (exemplificação)

Fonte: Elaborado pelo autor, 2021.

No exemplo acima temos o cálculo de um fatorial a programação C. O caso base para esta função fatorial seria **n = 0.** Retornamos 1 quando **n = 0**. Como resultado teremos 120, uma vez que desejamos o fatorial de 5 ***(fact(5))***. Este é um algoritmo de recursividade, uma vez que será executado até a condição de parada for satisfeita.

Por outro lado, podemos utilizar algoritmos de recursividade em outras linguagens de programação, como é o caso do JAVA e do PYTHON. A lógica é a mesma, modificando, neste caso, a construção do algoritmo.

Vejamos primeiramente a construção desse algoritmo em Java (Algortimo 4.2).

```
class Factorial {
  public static int fact(int n)
  {
    if (n <= 1)
      return 1;
    else
      return n * fact(n-1);
  }

  public static void main( String args[] ) {
```

```
      System.out.println(fact(5));
  }
}
```

Algoritmo 4.2: Fatorial (exemplificação)

Fonte: Elaborado pelo autor, 2021.

E agora vejamos a construção deste mesmo algoritmo em PYTHON. Um fator interessante é que na linguagem de programação PYTHON há uma estrutura mais básica e simples para a construção destes e de outros algoritmos (Algoritmo 4.3).

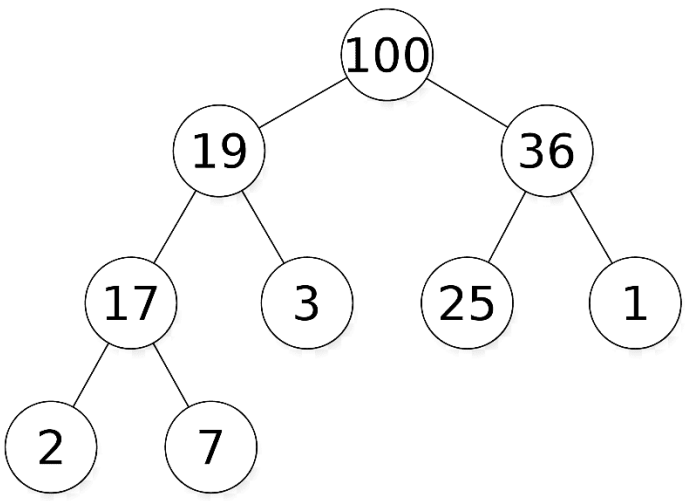
```
def fact(n):
  if n <= 1: # base case
    return 1;
  else :
    return n * fact(n-1);

print(fact(5))
```

Algoritmo 4.3: Fatorial (exemplificação)

Fonte: Elaborado pelo autor, 2021.

Agora que compreendemos o que são algoritmos recursivos e suas construções nas mais diferentes linguagens de programação, vamos entender como funciona o algoritmo de *heapsort.*

4.2 *HeapSort*



Fonte: https://www.pngwing.com/pt/free-png-ymmmn

O algoritmo *heapsort* pode ser entendido como uma versão aprimorada da árvore de pesquisa binária. Ele não cria um nó como no caso da árvore de pesquisa binária, em vez disso, ele constrói o *heap* ajustando a posição dos elementos dentro do próprio *array [vetor].*

Nesse método, uma estrutura em árvore chamada *heap* é usada, em que *heap* é um tipo de árvore binária. Uma árvore binária balanceada ordenada é chamada de ***heapMín***, onde o valor na raiz de qualquer subárvore é menor ou igual ao valor de qualquer um de seus filhos.

Uma árvore binária balanceada ordenada é chamada de ***heapMáximo***, quando o valor na raiz de qualquer subárvore é maior ou igual ao valor de qualquer um de seus filhos.

Um ***heap*** é uma estrutura de dados em árvore que satisfaz as seguintes propriedades:

Propriedade da forma: *Heap* é sempre uma árvore binária completa, o que significa que todos os níveis de uma árvore são totalmente preenchidos. Não deve haver um nó com apenas um filho. Cada nó, exceto as folhas, deve ter dois filhos, então apenas um ***heap*** é chamado como uma árvore binária completa.

Propriedade *heap*: todos os nós são maiores ou iguais ou menores ou iguais a cada um de seus filhos. Isso significa que se o nó pai for maior do que o nó filho, ele é chamado de *heap*Máximo (como vimos anteriormente). Considerando que, se o nó pai for menor do que o nó filho, ele é chamado de *heapM*ínimo (como vimos anteriormente).

Legal, né? Então, para resumirmos, um algoritmo *heapsort* utiliza uma estrutura de dados chamado de *heap* binário que ordena os elementos a medida que cada um vai sendo inserido na estrutura.

Agora que entendemos o que é um algoritmo *heapsort*, vamos compreender como acontece sua construção.

4.2.1 Estruturação de um algoritmo *HeapSort*

Iniciaremos esse subcapitulo com um exemplo que irá retratar o funcionamento de um algoritmo ***HeapSort***.

Vamos considerar árvore abaixo. Percebemos que essa árvore é completa e, com isso, atende a propriedade base do ***HeapSort***:

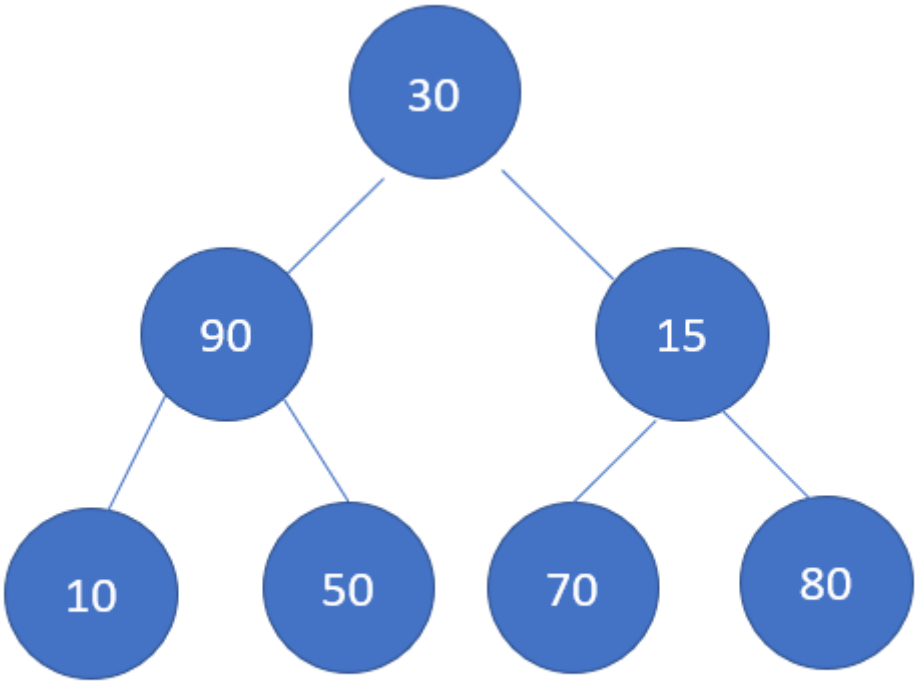


Figura 4.1: HeapSort

Fonte: Elaborado pelo autor, 2021

Lista = [30, 90, 15, 10, 50, 70, 80]

Neste caso sempre precisamos garantir que o valor da chave do pai é maior que a dos filhos. Com isso, se por acaso obtivermos um filho com valor maior que o pai, devemos trocar o filho com o pai.

As subárvores são analisadas e, sempre o maior elemento da subárvore deve ser colocado na raiz. Vamos analisar a primeira parte.

Na figura 4.2 percebe-se que o nó 80 está abaixo do nó 15, logo faremos a inversão, uma vez que sempre precisamos garantir que o nó pai seja maior que os filhos.

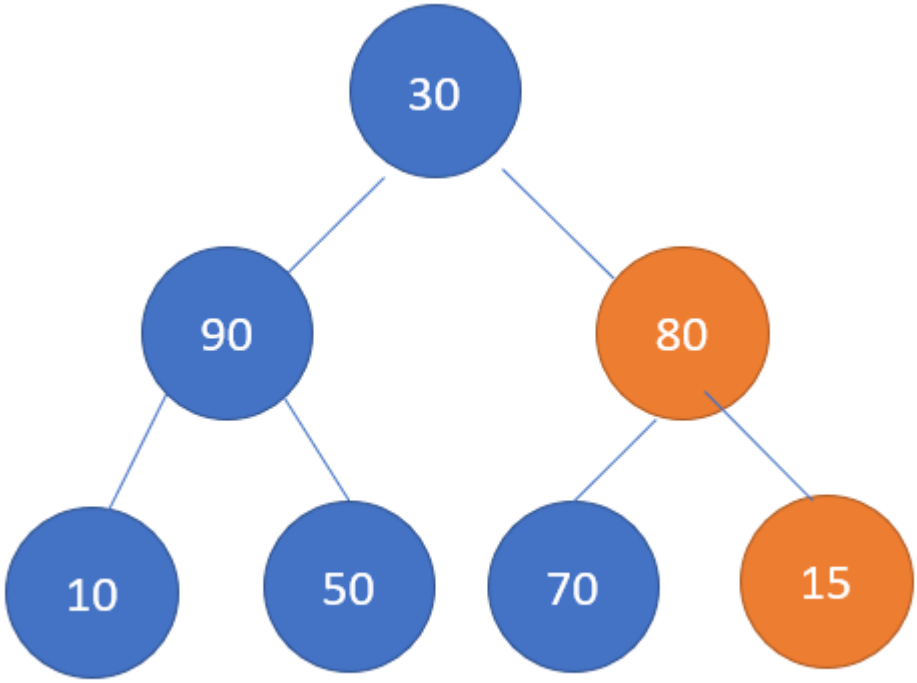


Figura 4.2: HeapSort

Fonte: Elaborado pelo autor, 2021

Com isso temos agora uma nova lista, onde:

Lista = [30, 90, 80, 10, 50, 70, 15]

Neste caso percebemos que na nossa lista houve a modificação de posições do número 15 e do número 80. Vamos continuar o processo analisando o próximo lado.

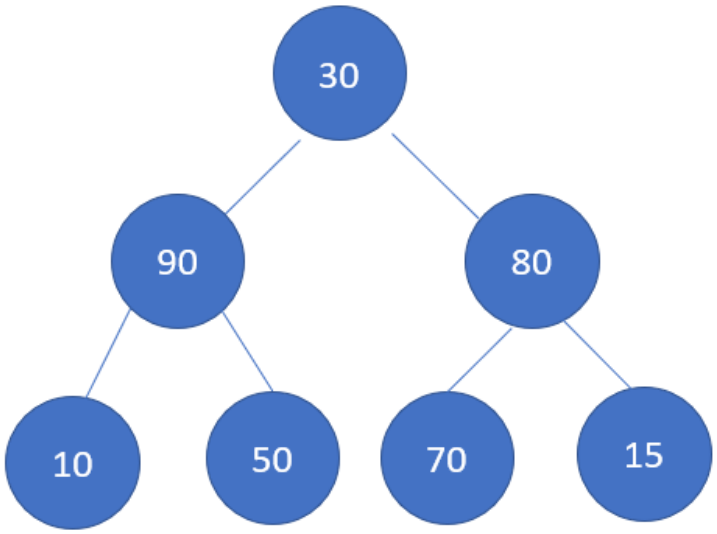


Figura 4.3: HeapSort

Fonte: Elaborado pelo autor, 2021

Percebemos na árvore acima que podemos trocar o nó 90 (maior) pelo nó 30 (que é o menor) (Figura 4.3).

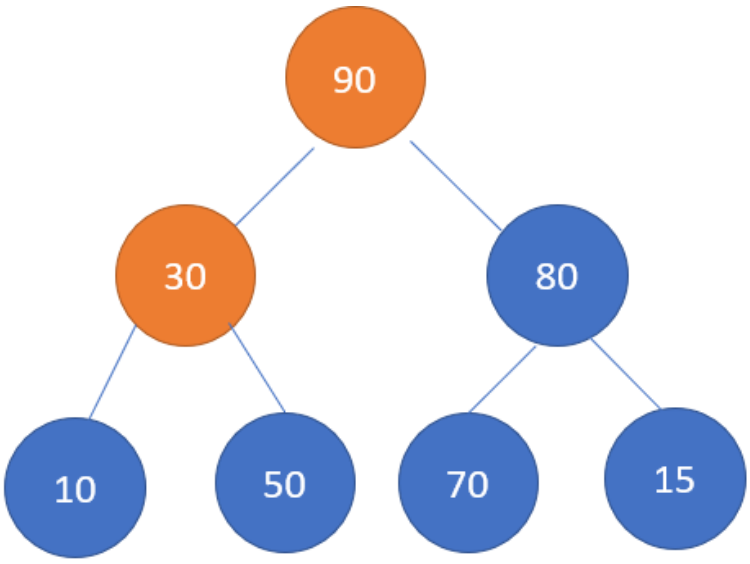


Figura 4.3: HeapSort

Fonte: Elaborado pelo autor, 2021

Com isso temos agora uma nova lista, onde:

Lista = [90, 30, 80, 10, 50, 70, 15]

Quando trocamos um nó pai, as subárvores precisam passar por um novo processo de análise, logo:

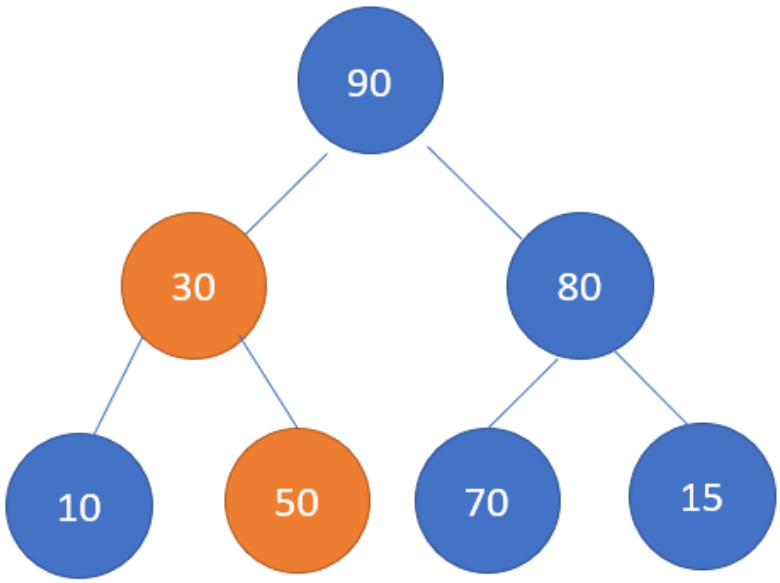


Figura 4.4: HeapSort

Fonte: Elaborado pelo autor, 2021

Como resultado, temos a troca de posições do nó 40 para o nó 50 e com isso temos uma nova lista, onde:

Lista = [90, 50, 80, 10, 30, 70, 15]

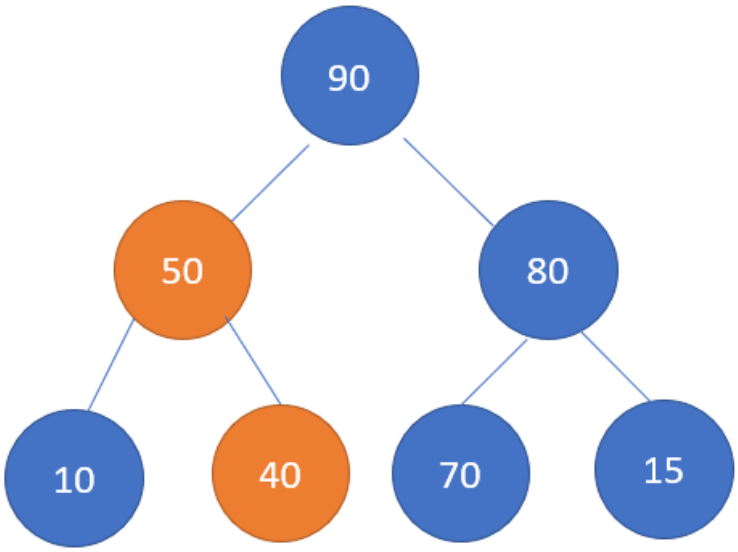


Figura 4.5: HeapSort

Fonte: Elaborado pelo autor, 2021

Observamos, assim, que cada uma das raízes possuem o maior número (elemento) nas suas arvores, e consequentemente, podemos trocar raiz pelo último elemento (Figura 4.6):

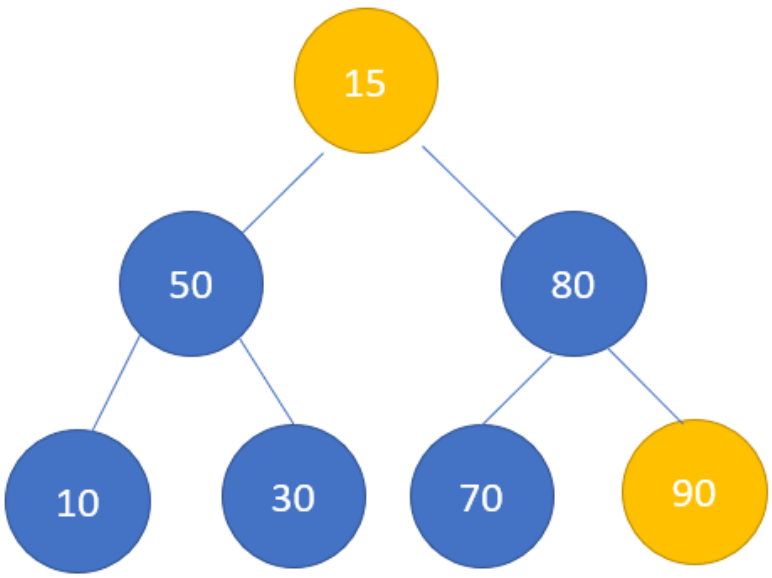


Figura 4.6: HeapSort

Fonte: Elaborado pelo autor, 2021

Onde nossa lista será:

Lista = [15, 50, 80, 10, 30, 70, 90]

Precisamos executar esta mesma etapa até que a lista esteja completamente ordenada.

Então aluno(a), neste capítulo estudamos o algoritmo de ordenação ***heapsort. Compreendemos a lógica envolvida e com isto estamos aptos*** a desenvolver esses algoritmos em linguagens de programação, como é caso de Java e Python. Nos capítulos 10 e 11 do nosso livro verificaremos como se realiza estas construções.

#ANOTA AÍ#

Entender a lógica do algoritmo é um dos fatores primordiais para o desenvolvimento e construção dos algoritmos. Por isso é importante estudar lógica de programação. A lógica é o primeiro passo para elaboração de diversos algoritmos em qualquer linguagem de programação.

#ANOTA AÍ#