UNIVERSIDADE DO ESTADO DO AMAZONAS - UEA
ESCOLA SUPERIOR DE TECNOLOGIA - EST
ENGENHARIA DA COMPUTAÇÃO

Sarah Silveira
Nicolas Fernandes de Lima

Algoritimo e Estrutura de Dados - HeapSort

MANAUS, AM
2019

Sumário

1 Algoritmo de classificação de heap

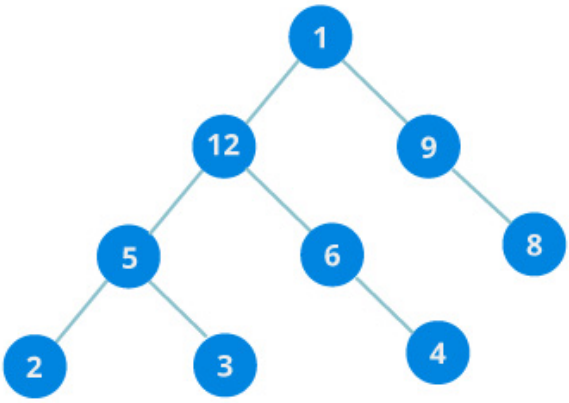
Heap Sort é um algoritmo de classificação popular e eficiente na programação de computadores. Aprender a escrever o algoritmo de classificação de heap requer conhecimento de dois tipos de estruturas de dados - matrizes e árvores.

O conjunto inicial de números que queremos classificar é armazenado em uma matriz, por exemplo, [10, 3, 76, 34, 23, 32]e após a classificação, obtemos uma matriz classificada[3,10,23,32,34,76]

A classificação de heap funciona visualizando os elementos da matriz como um tipo especial de árvore binária completa chamada heap.

1.1 Árvore binária

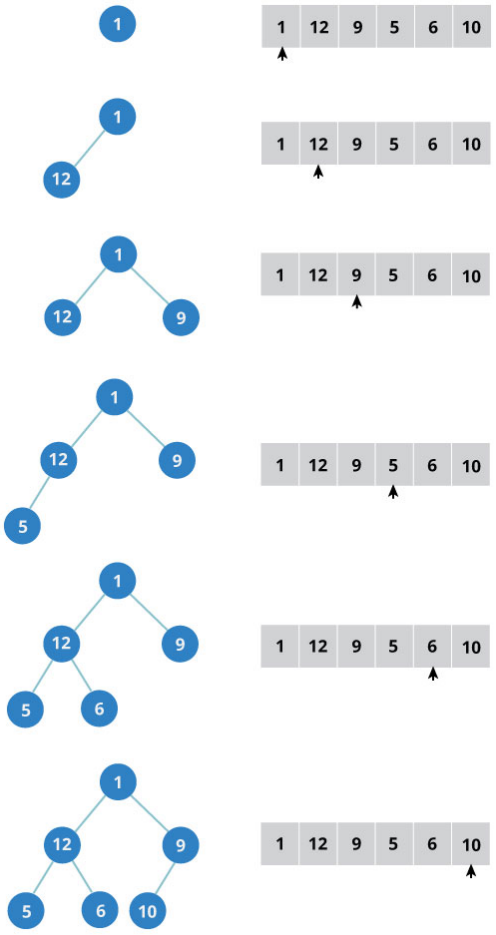
Uma árvore binária é uma estrutura de dados em árvore na qual cada nó pai pode ter no máximo dois filhos



Na imagem acima, cada elemento tem no máximo dois filhos.

1.2 Como criar uma árvore binária completa a partir de um vetor não organizado?

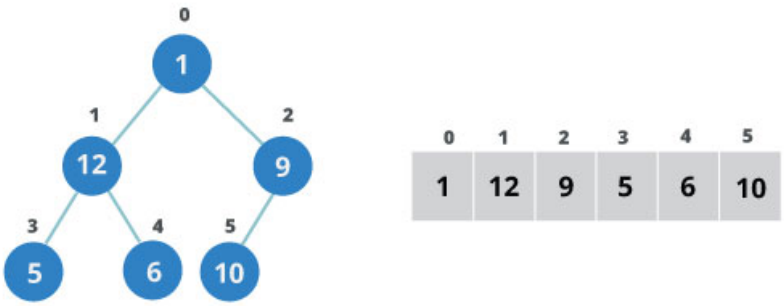
- Selecione o primeiro elemento da lista para ser o nó raiz. (Primeiro nível - 1 elemento)
- Coloque o segundo elemento como filho esquerdo do nó raiz e o terceiro elemento como filho direito. (Segundo nível - 2 elementos)
- Coloque os próximos dois elementos como filhos do nó esquerdo do segundo nível. Novamente, coloque os próximos dois elementos como filhos do nó direito do segundo nível (3^{o} nível - 4 elementos).
- Continue repetindo até chegar ao último elemento.



1.3 Relação entre índices do vetor e elementos de árvore

A árvore binária completa possui uma propriedade interessante que podemos usar para encontrar os filhos e os pais de qualquer nó.

Se o índice de qualquer elemento do vetor for i , o elemento no índice 2i+1se tornará o filho esquerdo e o elemento no 2i+2índice se tornará o filho direito. Além disso, o pai de qualquer elemento no índice i é fornecido pelo limite inferior de (i-1)/2.



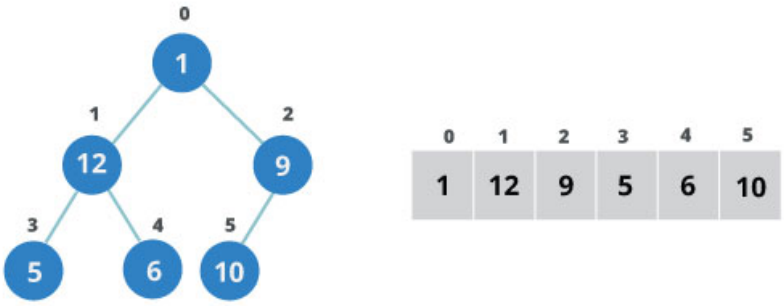
A compreensão desse mapeamento de índices do vetor para posições em árvore é fundamental para entender como a Estrutura de Dados de Heap funciona e como é usada para implementar a Classificação do Heap .

1.4 O que é estrutura de dados heap?

Heap é uma estrutura de dados baseada em uma a´rvore binária especial. Diz-se que uma árvore binária segue uma estrutura de dados de heap se:

- é uma árvore binária completa;
- Todos os nós na árvore seguem a propriedade de que eles são maiores que seus filhos, ou seja, o maior elemento está na raiz e seus filhos e menores que a raiz e assim por diante. Tal Heap é chamada de max-heap. Se, em vez disso, todos os nós forem menores que seus filhos, isso será chamado de min-heap;

O diagrama de exemplo a seguir mostra Max-Heap e Min-Heap.

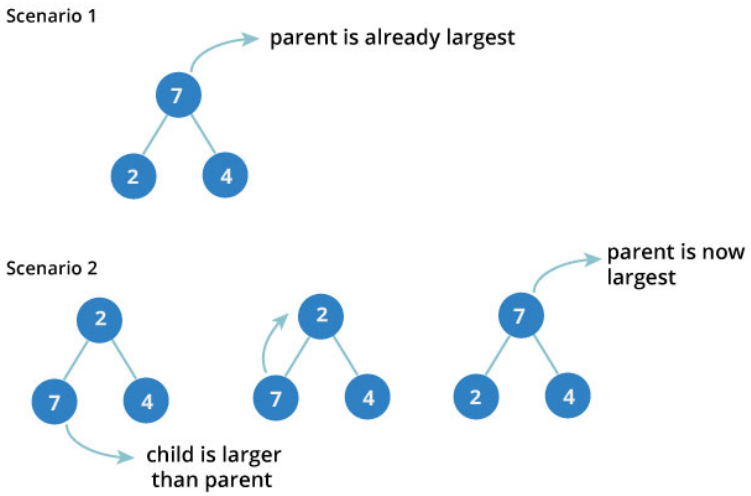


1.5 Como aplicar o Heapfy à um ávore?

A partir de uma árvore binária completa, podemos modificá-la para se tornar um Max-Heap executando uma função chamada heapify em todos os elementos não-folha da pilha.

Como o heapfiy usa recursão, pode ser difícil de entender. Então, vamos primeiro pensar em como você empilharia uma árvore com apenas três elementos.

```
heapify(array)
    Root = array[0]
    Largest = largest( array[0] , array [2*0 + 1]. array[2*0+2])
    if(Root != Largest)
          Swap(Root, Largest)
```

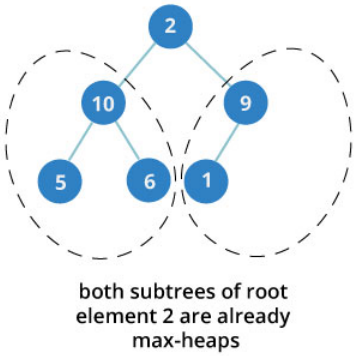


O exemplo acima mostra dois cenários - um no qual a raiz é o maior elemento e não precisamos fazer nada. E outra na qual o root tinha um elemento maior quando criança e precisávamos trocar para manter a propriedade max-heap.

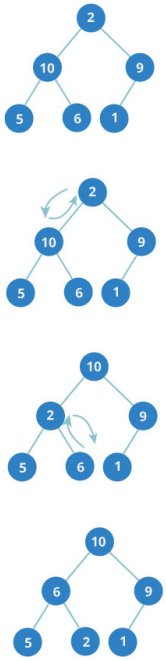
Se você já trabalhou com algoritmos recursivos antes, provavelmente já identificou que esse deve ser o caso base.

Agora vamos pensar em outro cenário em que existem mais de um nível.

O elemento top não é um heap máximo, mas todas as subárvores são heaps máximas.



Para manter a propriedade max-heap para toda a árvore, teremos que continuar pressionando 2 para baixo até que ela atinja sua posição correta.



Portanto, para manter a propriedade max-heap em uma árvore em que ambas as subárvores são max-heaps, precisamos executar heapify no elemento raiz repetidamente até que seja maior que seus filhos ou se torne um nó folha.

Podemos combinar essas duas condições em uma função heapify como:

```
void heapify(int arr[], int n, int i)
{
   int largest = i;
   int l = 2*i + 1;
   int r = 2*i + 2;
   if (l < n && arr[l] > arr[largest])
     largest = l;
   if (right < n && arr[r] > arr[largest])
     largest = r;
   if (largest != i)
   {
     swap(arr[i], arr[largest]);
     // Recursively heapify the affected sub-tree
     heapify(arr, n, largest);
   }
}
```

Esta função funciona para o caso base e para uma árvore de qualquer tamanho. Assim, podemos mover o elemento raiz para a posição correta para manter o status de heap máximo para qualquer tamanho de árvore, desde que as subárvores sejam heaps máximas.

1.6 Criar max-heap

Para construir um heap máximo a partir de qualquer árvore, podemos começar a empilhar cada subárvore de baixo para cima e terminar com um heap máximo depois que a função for aplicada a todos os elementos, incluindo o elemento raiz.

No caso da árvore completa, o primeiro índice do nó não-folha é dado por n/2 - 1. Todos os outros nós seguintes são nós de folha e, portanto, não precisam ser heapizados.

Assim, podemos construir uma pilha máxima como:

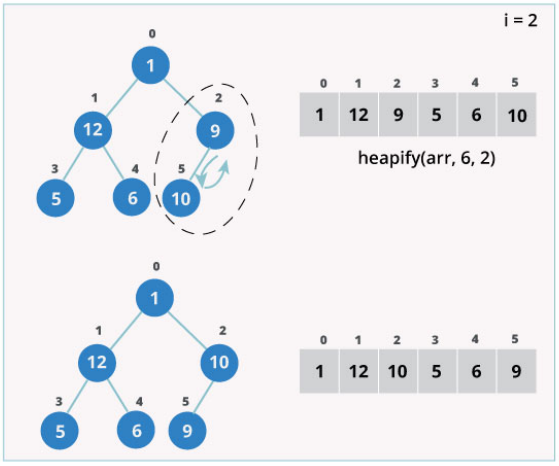
```
// Build heap (rearrange array)
   for (int i = n / 2 - 1; i >= 0; i--)
     heapify(arr, n, i);
```

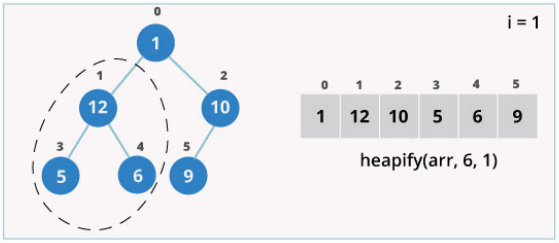
Como mostra o diagrama abaixo, começamos empilhando as árvores menores menores e subindo gradualmente até chegarmos ao elemento raiz.

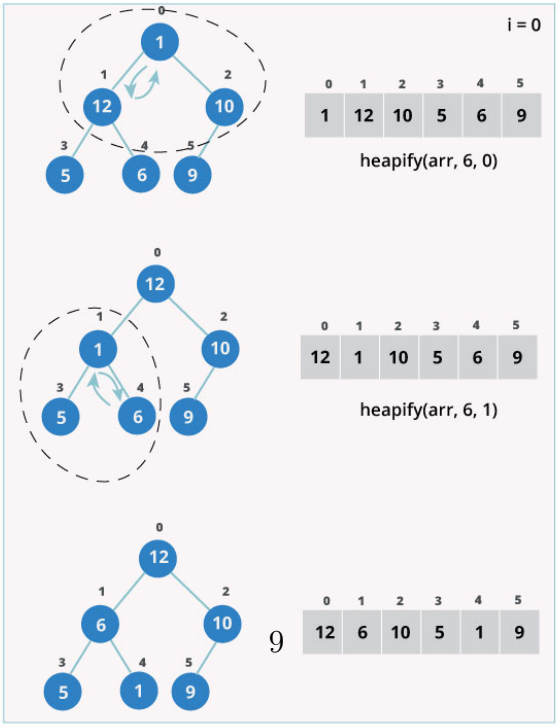
	0	1	2	3	4	5
arr	1	12	9	5	6	10

n 6

i = 6/2-1
= 2 -> 0

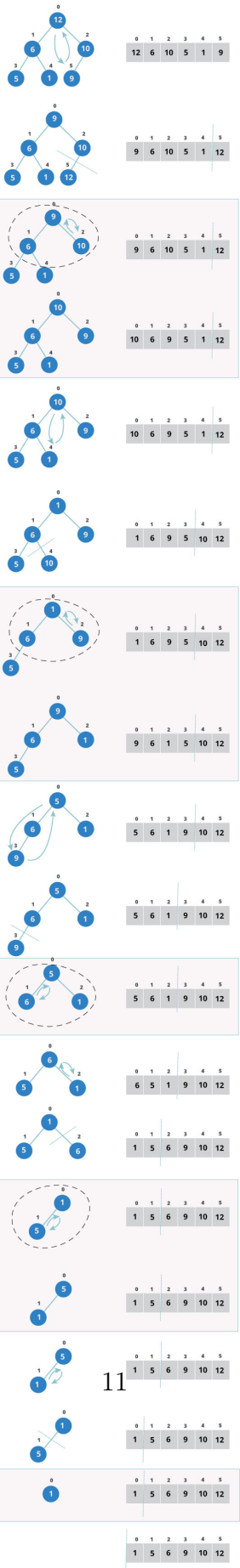






1.7 Procedimentos seguir com o Heapsort

- Como a árvore atende à propriedade Max-Heap, o item maior é armazenado no nó raiz.
- Remova o elemento raiz e coloque no final da matriz (enésima posição) Coloque o último item da árvore (pilha) no local vago.
- Reduza o tamanho do heap em 1 e monte novamente o elemento raiz para que tenhamos o elemento mais alto na raiz.
- O processo é repetido até que todos os itens da lista sejam classificados.


0 1 2 3 4 5
12 6 10 5 1 9
0 1 2 3 4 5
9 6 10 5 1 12
0 1 2 3 4 5
9 6 10 5 1 12
0 1 2 3 4 5
10 6 9 5 1 12
0 1 2 3 4 5
10 6 9 5 1 12
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
9 6 1 5 10 12
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5
6 5 1 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12

O código abaixo mostra a operação:

```
for (int i=n-1; i>=0; i--)
   {
     // Move current root to end
     swap(arr[0], arr[i]);
     // call max heapify on the reduced heap
     heapify(arr, i, 0);
   }
```

1.8 Comportamento

A classificação de heap possui O(nlogn)complexidades de tempo para todos os casos (melhor caso, caso médio e pior caso).

Vamos entender o motivo. A altura de uma árvore binária completa contendo n elementos élog(n)

Como vimos anteriormente, para empilhar completamente um elemento cujas subárvores já são max-heaps, precisamos continuar comparando o elemento com seus filhos esquerdo e direito e empurrando-o para baixo até atingir um ponto em que ambos os filhos sejam menores que ele.

Na pior das hipóteses, precisaremos mover um elemento da raiz para o nó folha, fazendo várias log(n)comparações e trocas.

Durante o estágio 'build max heap', fazemos isso para n/2elementos, portanto a pior complexidade da etapa 'build heap' é n/2*log(n) nlogn.

Durante a etapa de classificação, trocamos o elemento raiz com o último elemento e empilhamos o elemento raiz. Para cada elemento, isso novamente leva o lognpior tempo, porque talvez tenhamos que trazê-lo da raiz até a folha. Como repetimos isso n vezes, a etapa 'heap sort' também é nlogn.

Além disso, como as etapas 'build max heap' e 'heap sort' são executadas uma após a outra, a complexidade algorítmica não é multiplicada e permanece na ordem de nlogn.

Também realiza a classificação na O(1)complexidade do espaço. Comparando com o Quick Sort, o melhor é o pior caso (O(nlogn)). A Classificação Rápida possui complexidade O(n2) para o pior caso. Mas em outros casos, a Classificação Rápida é rápida. O Introsort é uma alternativa ao heapsort que combina quicksort e heapsort para reter vantagens de ambos:

velocidade do pior caso do heapsort e velocidade média do quicksort.

1.9 Aplicação de Heap Sort

Os sistemas preocupados com a segurança e o sistema incorporado, como o Linux Kernel, usam o Heap Sort por causa do O(n log n)limite superior no tempo de execução do Heapsort e O(1)do limite superior constante em seu armazenamento auxiliar.

Embora a classificação de pilha tenha O(n log n)complexidade de tempo, mesmo na pior das hipóteses, ela não possui mais aplicativos (em comparação com outros algoritmos de classificação como Classificação rápida, Classificação por mesclagem). No entanto, sua estrutura de dados subjacente, heap, pode ser usada com eficiência se quisermos extrair o menor (ou maior) da lista de itens sem a sobrecarga de manter os itens restantes na ordem classificada. Por exemplo, paginação.

2 Implementação de C ++

```
#include <iostream>
using namespace std;
void heapify(int arr[], int n, int i)
{
   // Find largest among root, left child and right child
   int largest = i;
   int l = 2*i + 1;
   int r = 2*i + 2;
   if (l < n && arr[l] > arr[largest])
     largest = l;
   if (right < n && arr[r] > arr[largest])
     largest = r;
   // Swap and continue heapifying if root is not largest
   if (largest != i)
   {
     swap(arr[i], arr[largest]);
     heapify(arr, n, largest);
   }
}
// main function to do heap sort
```

```
void heapSort(int arr[], int n)
{
   // Build max heap
   for (int i = n / 2 - 1; i >= 0; i--)
     heapify(arr, n, i);
   // Heap sort
   for (int i=n-1; i>=0; i--)
   {
     swap(arr[0], arr[i]);

     // Heapify root element to get highest element at root again
     heapify(arr, i, 0);
   }
}
void printArray(int arr[], int n)
{
   for (int i=0; i<n; ++i)
     cout << arr[i] << " ";
   cout << "\n";
}
int main()
{
   int arr[] = {1,12,9,5,6,10};
   int n = sizeof(arr)/sizeof(arr[0]);
   heapSort(arr, n);
   cout << "Sorted array is \n";
   printArray(arr, n);
}
```