Árvore Binária Heap

Prof. Flavio B. Gonzaga
flavio.gonzaga@unifal-mg.edu.br
Universidade Federal de Alfenas
UNIFAL-MG

Sumário

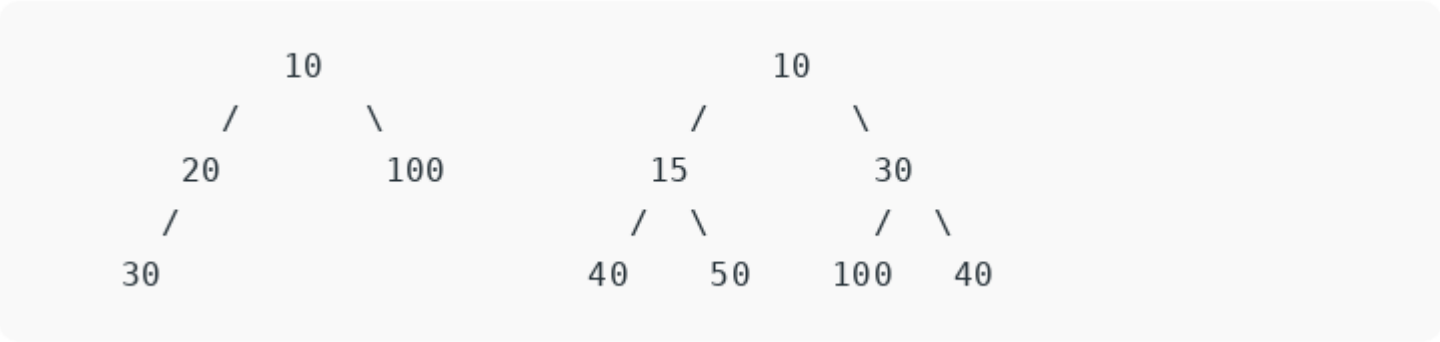
- Árvore Heap;
  - Motivação;
  - Funcionamento;
  - Inserindo nós...;
  - Removendo nós...;

Motivação

- A árvore Heap é uma árvore binária, mas **não uma árvore binária de busca**;
  - Portanto, não existe a garantia de que todos os elementos na subárvore esquerda serão menores que o nó raiz dessa subárvore;
  - O mesmo vale para a subárvore direita;
- A característica garantida pela árvore Heap no entanto é que o menor (ou o maior) elemento da árvore estará sempre na raiz da mesma;
  - A propriedade acima também se aplica recursivamente aos nós internos da árvore.
- A árvore Heap será portanto uma **Min Heap** ou uma **Max Heap**, de acordo com a natureza do valor oferecido na raiz (mínimo ou máximo dentre os valores presentes na árvore);

Motivação

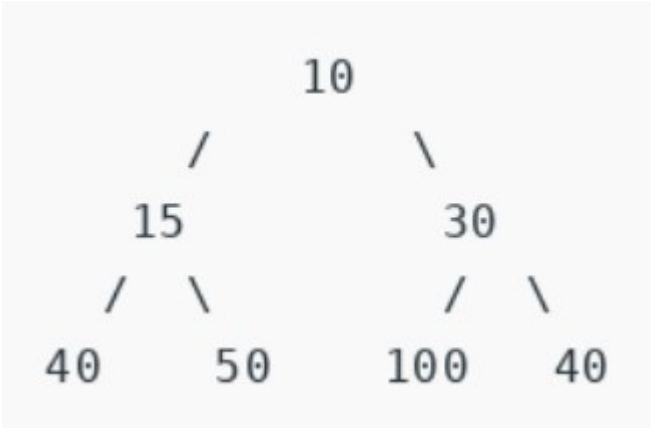
- Exemplos de Min Heap:



Fonte: https://www.geeksforgeeks.org/binary-heap/. Acesso em 08/03/2021.

Funcionamento

- Geralmente implementada como um vetor;
- É uma árvore completa, ou seja, possui todos os níveis preenchidos, exceto possivelmente o último nível, que terá os nós sempre o mais a esquerda possível;



	0	1	2	3	4	5	6
vet	10	15	30	40	50	100	40

Para o i-ésimo nó:

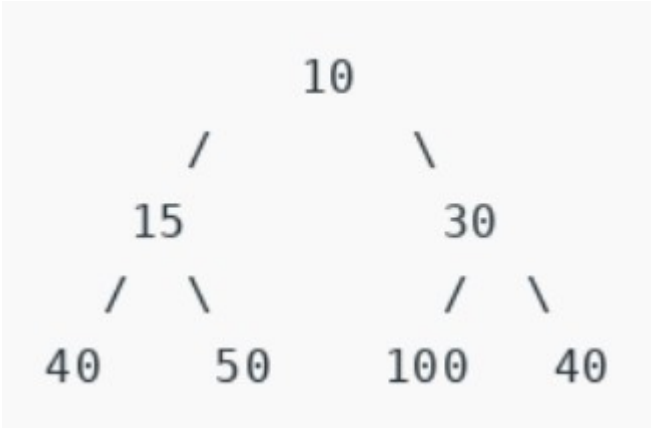
- vet[(i-1)/2] – retornará o nó pai

- vet[(2*i)+1] – retornará o filho a esquerda

- vet[(2*i)+2] – retornará o filho a direita

Funcionamento

- **<u>Inserções</u>** são sempre feitas no final da árvore O(log n);
- Se o novo nó for menor do que seu pai, troca-se a posição de um com o outro;
  - Essa operação é repetida subindo-se na árvore, até que o nó seja maior do que seu pai, ou se torne a raiz;



	0	1	2	3	4	5	6
vet	10	15	30	40	50	100	40

Para o i-ésimo nó:

- vet[(i-1)/2] – retornará o nó pai

- vet[(2*i)+1] – retornará o filho a esquerda

- vet[(2*i)+2] – retornará o filho a direita

Funcionamento

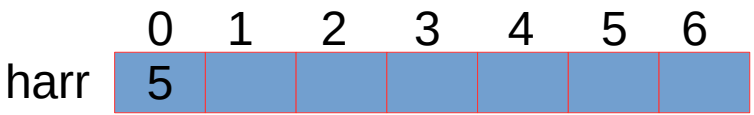
- **Remoções** são sempre feitas na raiz, que possui o menor ou o maior valor, dependendo do tipo da árvore O(log n);
- O último elemento inserido na árvore assume o lugar na raiz temporariamente;
  - Se a nova raiz for maior do que algum dos seus filhos, troca-se de lugar com o menor deles;
  - Realiza-se essa operação recursivamente até que o nó se torne menor ou igual do que ambos os filhos, ou até que o mesmo se torne um nó folha;

Inserindo nós...

5 – 2 – 4 – 15 – 7 – 3 – 1

Inserindo nós...

~~5~~ – 2 – 4 – 15 – 7 – 3 – 1



Para o i-ésimo nó:

- vet[(i-1)/2] – retornará o nó pai

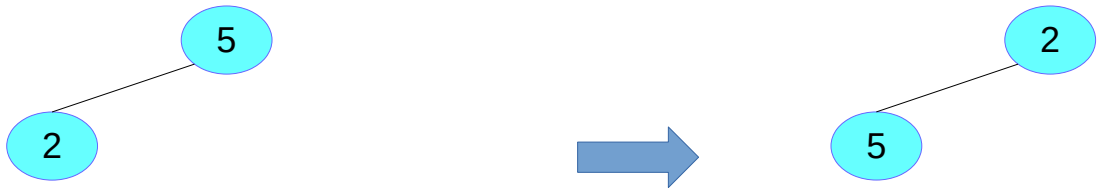
- vet[(2*i)+1] – retornará o filho a esquerda

- vet[(2*i)+2] – retornará o filho a direita

```
// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}
```

Inserindo nós...

5 – 2 – 4 – 15 – 7 – 3 – 1




```
// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}
```

Para o i-ésimo nó:

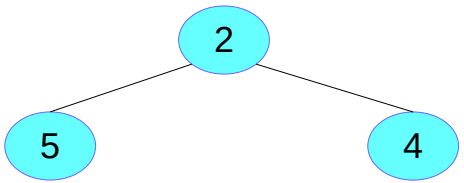
- vet[(i-1)/2] – retornará o nó pai

- vet[(2*i)+1] – retornará o filho a esquerda

- vet[(2*i)+2] – retornará o filho a direita

Inserindo nós...

~~5~~ – ~~2~~ – ~~4~~ – 15 – 7 – 3 – 1



	0	1	2	3	4	5	6
harr	2	5	4				

Para o i-ésimo nó:

- vet[(i-1)/2] – retornará o nó pai

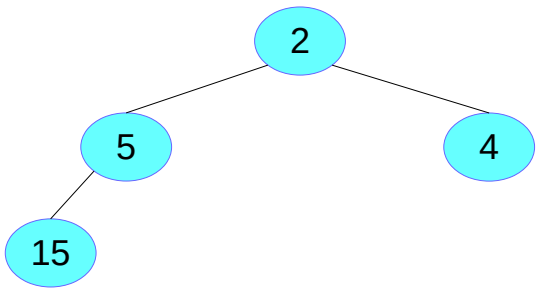
- vet[(2*i)+1] – retornará o filho a esquerda

- vet[(2*i)+2] – retornará o filho a direita

```
// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}
```

Inserindo nós...

~~5~~ – ~~2~~ – ~~4~~ – ~~15~~ – 7 – 3 – 1



	0	1	2	3	4	5	6
harr	2	5	4	15			

Para o i-ésimo nó:

- vet[(i-1)/2] – retornará o nó pai

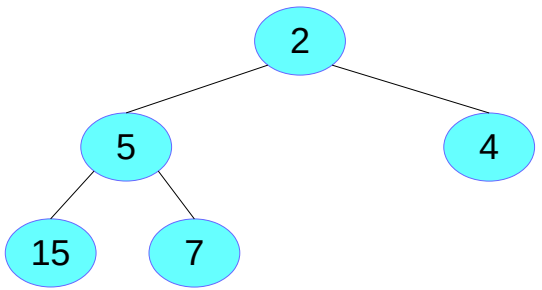
- vet[(2*i)+1] – retornará o filho a esquerda

- vet[(2*i)+2] – retornará o filho a direita

```
// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}
```

Inserindo nós...

~~5 – 2 – 4 – 15 – 7~~ – 3 – 1



	0	1	2	3	4	5	6
harr	2	5	4	15	7		

Para o i-ésimo nó:

- vet[(i-1)/2] – retornará o nó pai

- vet[(2*i)+1] – retornará o filho a esquerda

- vet[(2*i)+2] – retornará o filho a direita

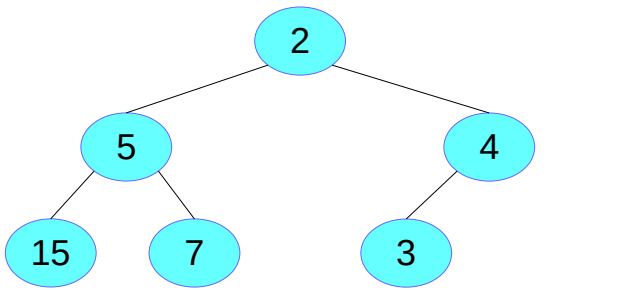
```
// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}
```

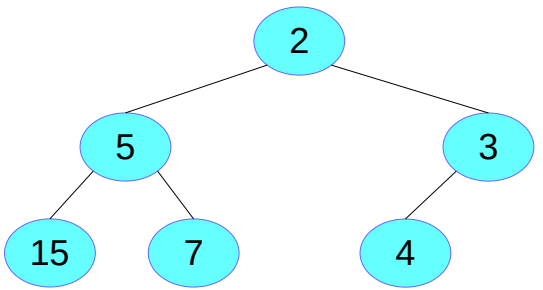
Inserindo nós...

Para o i-ésimo nó:

- vet[(i-1)/2] – retornará o nó pai
- vet[(2*i)+1] – retornará o filho a esquerda
- vet[(2*i)+2] – retornará o filho a direita

~~5 – 2 – 4 – 15 – 7 – 3 – 1~~





	0	1	2	3	4	5	6
harr	2	5	4	15	7	3	

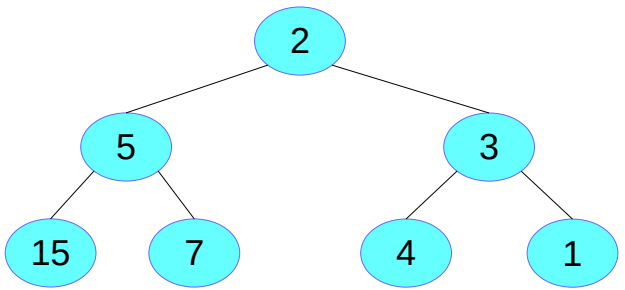
	0	1	2	3	4	5	6
harr	2	5	3	15	7	4	

Inserindo nós...

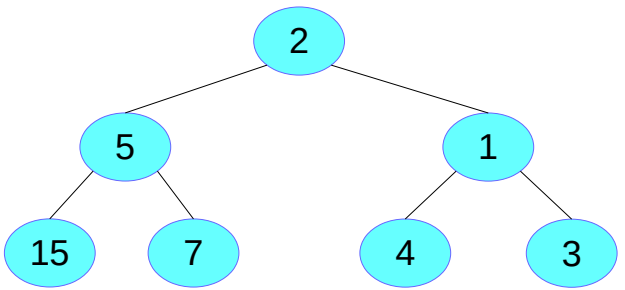
Para o i-ésimo nó:

- vet[(i-1)/2] – retornará o nó pai
- vet[(2*i)+1] – retornará o filho a esquerda
- vet[(2*i)+2] – retornará o filho a direita

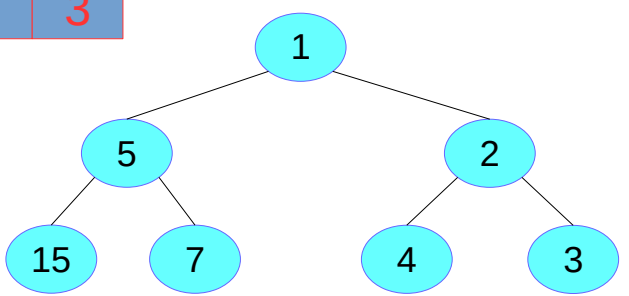
~~5 – 2 – 4 – 15 – 7 – 3 – 1~~



	0	1	2	3	4	5	6
harr	2	5	3	15	7	4	1



	0	1	2	3	4	5	6
harr	2	5	1	15	7	4	3

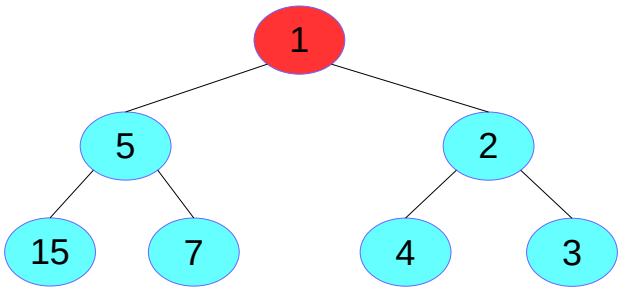


	0	1	2	3	4	5	6
harr	1	5	2	15	7	4	3

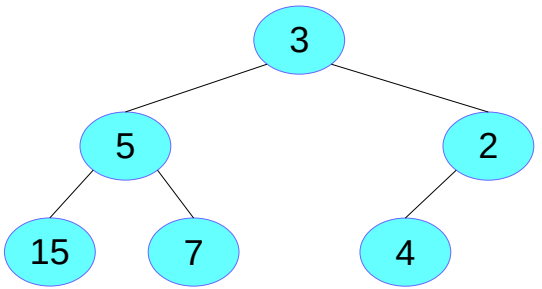
Removendo nós...

Para o i-ésimo nó:

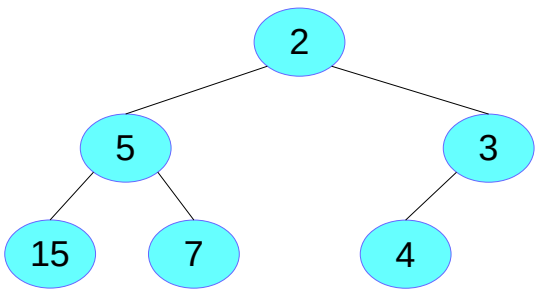
- vet[(i-1)/2] – retornará o nó pai
- vet[(2*i)+1] – retornará o filho a esquerda
- vet[(2*i)+2] – retornará o filho a direita



	0	1	2	3	4	5	6
harr	1	5	2	15	7	4	3



	0	1	2	3	4	5	6
harr	3	5	2	15	7	4	

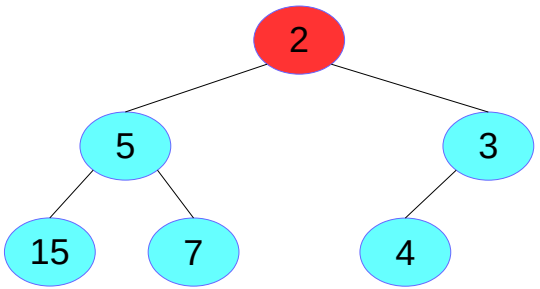


	0	1	2	3	4	5	6
harr	2	5	3	15	7	4	

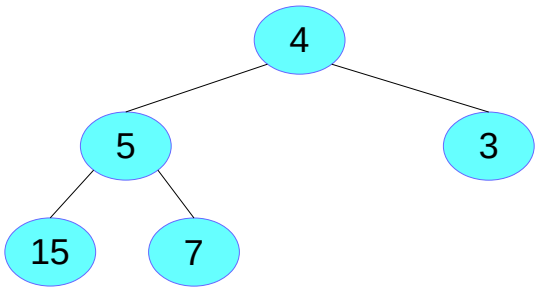
Removendo nós...

Para o i-ésimo nó:

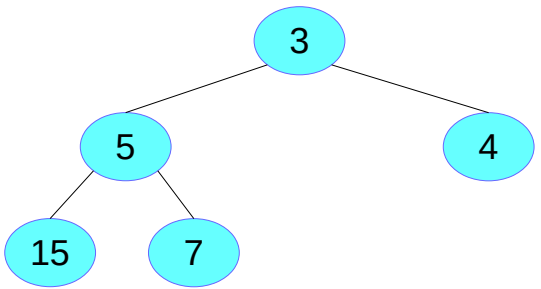
- vet[(i-1)/2] – retornará o nó pai
- vet[(2*i)+1] – retornará o filho a esquerda
- vet[(2*i)+2] – retornará o filho a direita



	0	1	2	3	4	5	6
harr	2	5	3	15	7	4	



	0	1	2	3	4	5	6
harr	4	5	3	15	7		



	0	1	2	3	4	5	6
harr	3	5	4	15	7		

Removendo nós...

```
// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    MinHeapify(0);

    return root;
}
```

```
// A recursive method to heapify a subtree with the root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}
```

Referências Bibliográficas

- Estruturas de Dados e Seus Algoritmos. Szwarcfiter J. L.; Markenzon L.. 3a Edição. Editora LTC. 2010.
- Estruturas De Dados Usando C. Tenenbaum A. M.; Langsam Y.; Augenstein M. J.. 1a Edição. Editora Pearson. 1995.
- Introdução a Estruturas de Dados: Com Técnicas de Programação em C. Celes W.; Cerqueira R.; Rangel J.. 2a Edição. Editora Elsevier. 2017.
- https://www.geeksforgeeks.org/binary-heap/, acesso em 18/03/2021.