**Heap**

**O que é um heap?**

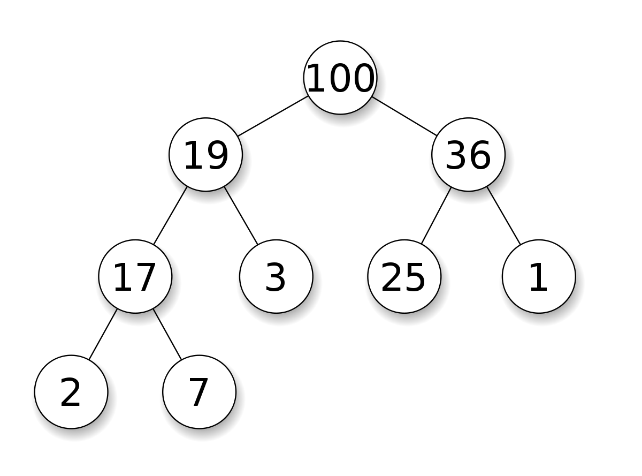
É uma estrutura de dados parecida com uma árvore binária, no entanto há alguns requisitos a serem preenchidos para que possamos considerar a estrutura um heap.

Definições básicas:

* Raiz: primeiro e maior elemento do heap.
* Nós: itens que possui filhos.
* Folhas: itens que não possui filho.
* Arestas: linhas que ligam os nós com seus filhos.

Estrutura de um heap

* O maior elemento deve estar na raiz da árvore.
* Cada **nó** tem o valor superior ao dos seus filhos.
* A estrutura deve estar o mais à esquerda possível.

  
(Na imagem acima podemos ver todos os requisitos sendo aplicados)

**Enxergando um heap em forma de vetor**

Para transformar um heap em um vetor, basta aplicar a seguinte regra:  
Definindo que o vetor comece no index 1, e sabendo que a raiz do heap é o primeiro elemento do vetor, basta aplicar o seguinte cálculo para encaixar os demais itens no vetor:

Posição do filho esquerdo = 2\*Posição do pai

Posição do filho direito = 2\*Posição do pai + 1

Transformando o heap acima em vetor:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 100 | 19 | 36 | 17 | 3 | 25 | 1 | 2 | 7 |

Também podemos inverter a operação para achar o pai de um elemento no vetor

Posição do pai = ⌊(Posição do filho/ 2)⌋

**Encontrando a altura**

É o número de aresta do nó que deseja saber a altura até uma folha levando em consideração o caminho mais longo. Para achar o valor exato basta realizar o cálculo abaixo:

m = número nós + folhas i = posição do nó no vetor

Fórmula: altura = ⌊lg(m/i)⌋

Exemplo: altura da árvore = ⌊lg(9/1)⌋ = ⌊lg(9)⌋ = ⌊3,2...⌋ = **3**

**Manutenção do Heap**

**MAX-HEAPFY**

Caso o vetor não esteja satisfazendo os requisitos para ser um heap em algum dos nós (exceto o primeiro), então podemos aplicar o algoritmo “**MAX-HEAPFY**” para deixá-lo na estrutura correta.

Assinatura: MAX-HEAPFY(v, n, i)

Parâmetros:

* v: vetor com todos os itens
* n: números de itens no vetor
* i: index do nó inicial em que iremos aplicar as regras do heap, para que seja um heap.

A função acima fará com que a estrutura a partir do index passado seja um heap.

Quanto custa este algoritmo?

Levando em consideração o pior caso, em que seja necessário realocar o item da raiz e descê-lo até a última folha, o custo seria O(altura), pois terá que percorrer todas as arestas. Como a altura não é um valor que obtemos via parâmetro, podemos substitui-la por ⌊ lg(m/i) ⌋ e como a raiz tem o index 1, então a altura é igual a ⌊ lg(m) ⌋ já que i será 1.   
Então, como limitante superior sabemos que o MAX-HEAPFY é **O(lg(m)).**

**BUILD-MAX-HEAP**

Dado um vetor qualquer, podemos construir um heap com a função a BUILD-MAX-HEAP.  
Assinatura: BUILD-MAX-HEAP(v, n)  
A função acima percorre de forma decrescente do nó **⌊n/2⌋** até 1, chamando a função MAX-HEAPFY para alocar os valores nos lugares corretos.

Algoritmo e análise:

BUILD-MAX-HEAP (A, n)

1. para i ← ⌊n/2⌋ decrescendo até 1 faça = Θ(n/2)

2. MAX-HEAPIFY (A, n, i) = O(n/2lgn)  
  
Análise grosseira

Realizando uma análise grosseira nosso resultado será: O(nlgn)

Análise detalhada:

Precisamos contar o número de nós em cada nível e multiplicar pelo custo do MAX-HEAPIFY para cada um (que depende da altura do nó). Se fizermos isto veremos que o consumo de tempo de BUILD-MAX-HEAP é **Θ(n).**

**Algoritmos de ordenação baseados em comparação**

Todo algoritmo de ordenação baseado em comparação irá ter como limitante inferior a análise assintótica de **Ω(nlgn)**, este é melhor resultado que um algoritmo baseado em comparação pode alcançar para seu pior caso.

**Algoritmos de ordenação lineares**

Existem algoritmos de ordenação que não são baseados em comparação, mas levam em consideração as características dos dados de entrada. Com eles podemos chegar em algoritmo que execute em **Θ(n).**

|  |  |  |
| --- | --- | --- |
| **Tabela de Algoritmos Estudados** | | |
| Nome | Aplicabilidade | T(n) |
| Insertion Sort | Todos os cenários. | O(n²) |
| Merge Sort | Todos os cenários. Algoritmo baseado em divisão e conquista, utiliza recursão para otimizar a execução. | Ω(nlgn) |
| Heap Sort | Todos os cenários. Algoritmo que estrutura o vetor em forma de heap para que possamos saber que sempre o maior elemento estará na raiz. | Ω(nlgn) |
| Quick Sort | Todos os cenários. Algoritmo que utiliza divisão e conquista baseado em um elemento pivô. | Pior caso: O(n²) caso médio: Ω(nlgn) |
| Counting Sort | Somente para o cenário em que o maior elemento do vetor (k) é menor que a quantidade de elementos do vetor (n), pois é criado um vetor auxiliar de tamanho K. | Θ(n) |
| Radix Sort | Somente para o cenário onde a quantidade dígitos dos elementos do vetor são iguais, caso necessário podemos aplicar um cálculo para que se tornem. | Θ(n) |
| Bucket Sort | Cenário onde sabemos o range dos elementos que estão no vetor. Para garantir o seu tempo caso médio cada bucket precisa ter apenas um elemento. | Θ(n) |

**Limitante Superior e Inferior**

**Limitante Inferior**

Dado um determinado problema P, chamamos de limite inferior (ou lower bound) a complexidade mínima necessária para resolvê-lo.

**Limitante Superior**

Dado um determinado problema P, chamamos de limite superior (ou upper bound) a complexidade do melhor algoritmo conhecido que o resolve.