

HeapSort

Estrutura de Dados II

Jairo Francisco de Souza

HeapSort

- Algoritmo criado por John Williams (1964)
- Complexidade $O(N\log N)$ no pior e médio caso
- Mesmo tendo a mesma complexidade no caso médio que o QuickSort, o HeapSort acaba sendo mais lento que algumas boas implementações do QuickSort
- Porém, além de ser mais rápido no pior caso que o QuickSort, necessita de menos memória para executar
- QuickSort necessita de um vetor $O(\log N)$ para guardar as estruturas enquanto o HeapSort não necessita de um vetor auxiliar.

HeapSort

- Utiliza a abordagem proposta pelo SelectionSort
- O SelectionSort pesquisa entre os n elementos o que precede todos os outros $n-1$ elementos
- Para ordenar em ordem ascendente, o heapsort põe o maior elemento no final do array e o segundo maior antes dele, etc.
- O heapsort começa do final do array pesquisando os maiores elementos, enquanto o selectionsort começa do início do array pesquisando os menores.

HeapSort

Para ordenar, o heapsort usa um Heap

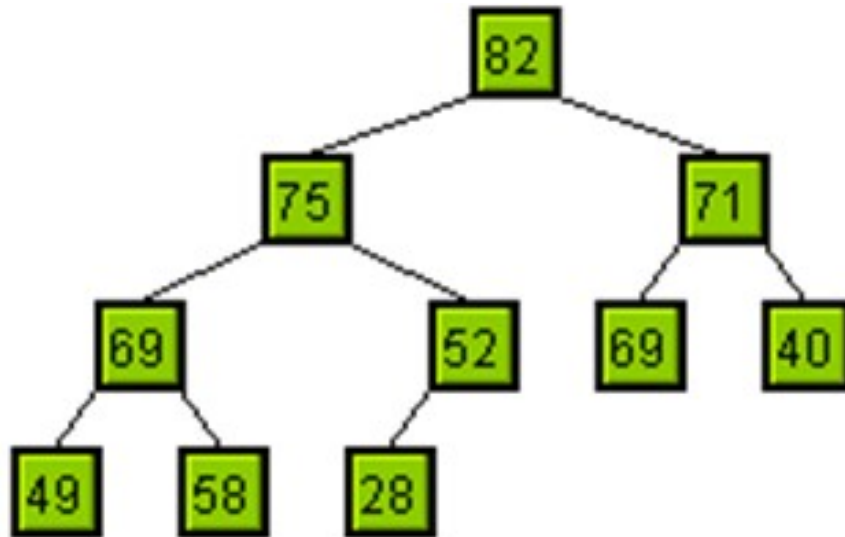
Heap é uma árvore binária com as seguintes propriedades:

O valor de cada nó não é menor que os valores armazenados em cada filho

A árvore é perfeitamente balanceada e as folhas no último nível estão todas nas posições mais a esquerda.

HeapSort

Exemplo de um heap:



HeapSort

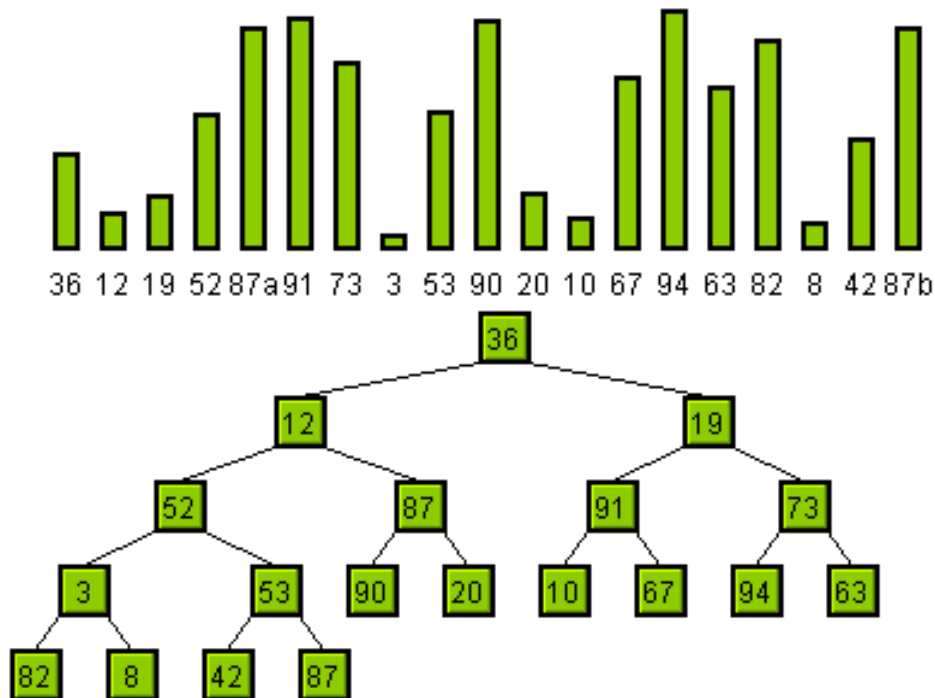
Elementos no heap não estão perfeitamente ordenados.

Apenas sabe-se que o maior elemento está no nó raiz e que, para cada nó, todos os seus descendentes não são maiores que os elementos na raiz.

HeapSort

Tenta-se evitar a utilização real de uma árvore.

A idéia é utilizar a abordagem de heap representando uma árvore como um array:



HeapSort

- Por que usar um heap é importante?
 - a pergunta "qual o maior elemento de vetor?" pode ser respondida instantaneamente: o maior elemento do vetor é $v[1]$;
 - se o valor de $v[1]$ for alterado, o heap pode ser restabelecido muito rapidamente: a operação de *heapfy* não demora mais que $\lg(n)$ para fazer o serviço;

HeapSort

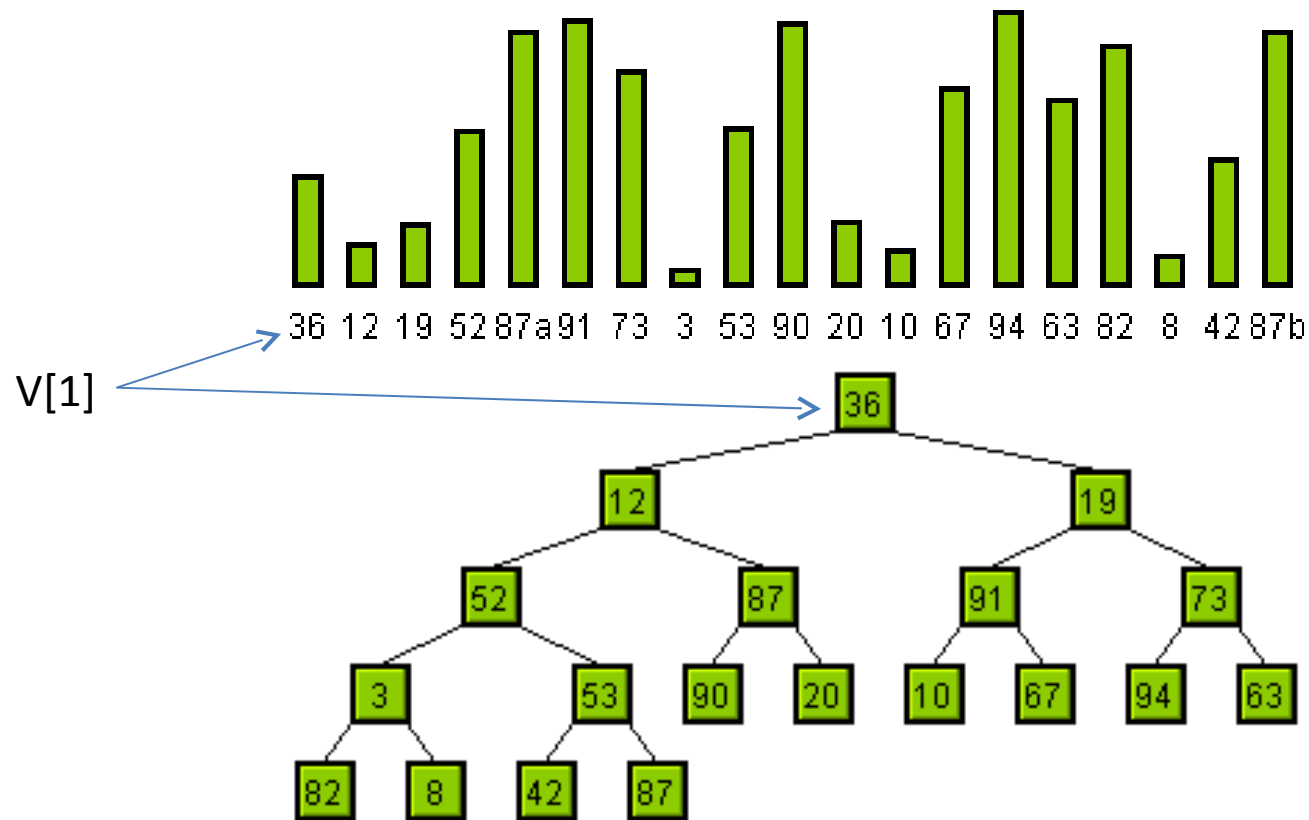
- Algoritmo
 - Dado um vetor V de n elementos, transformar o vetor em um heap
 - Pegar a posição $V[1]$ (ou seja, o maior elemento) e trocar de posição com $V[\max]$.
 - Repetir o processo com um array formado pelos elementos $V[1], \dots, V[n-1]$

HeapSort

- Primeiro passo:
 - Como transformar o array em um heap?
- Verifique que, caso o array represente uma árvore, então:
 - A raiz está em $V[1]$
 - O pai de um índice f é $f \div 2$
 - O filho esquerdo de um índice p é $2p$
 - O filho direito de um índice p é $2p+1$

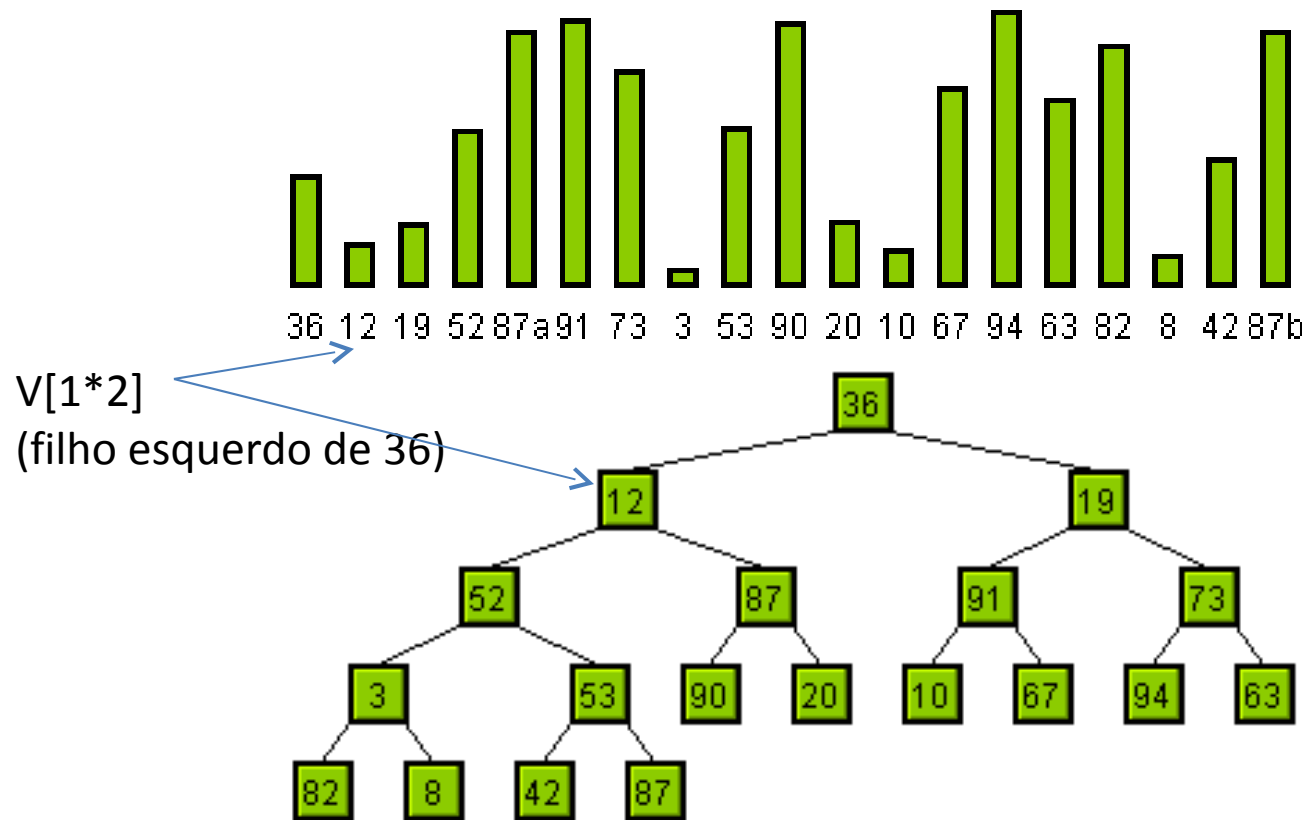
HeapSort

Exemplo:



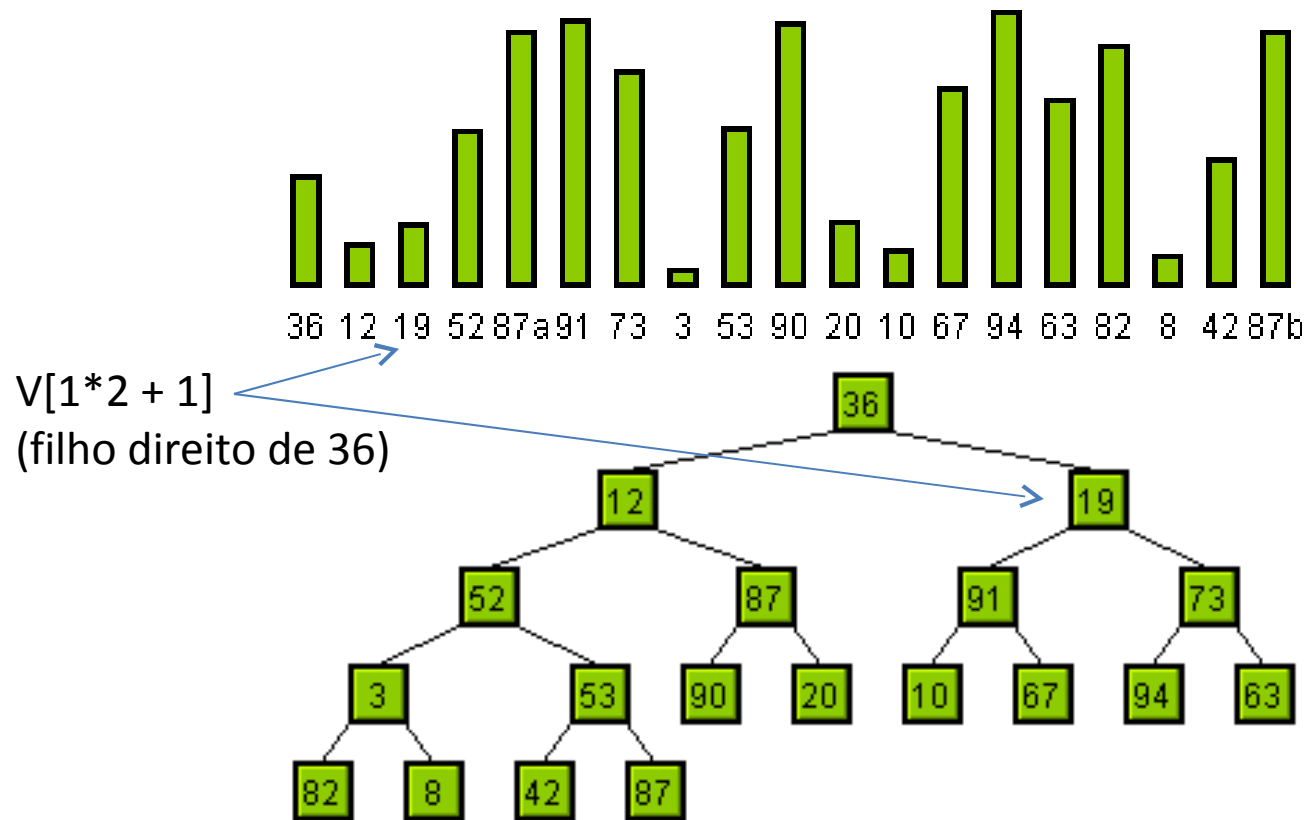
HeapSort

Exemplo:



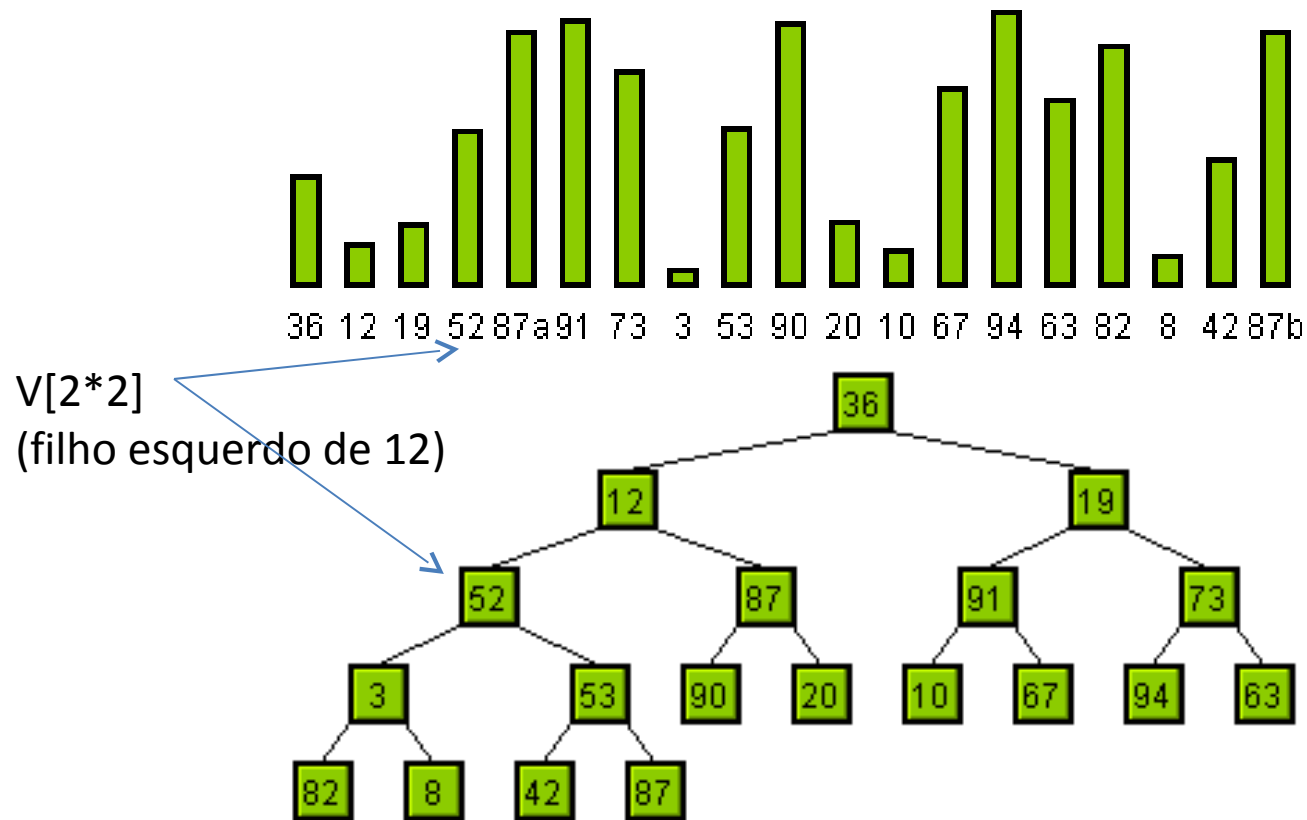
HeapSort

Exemplo:



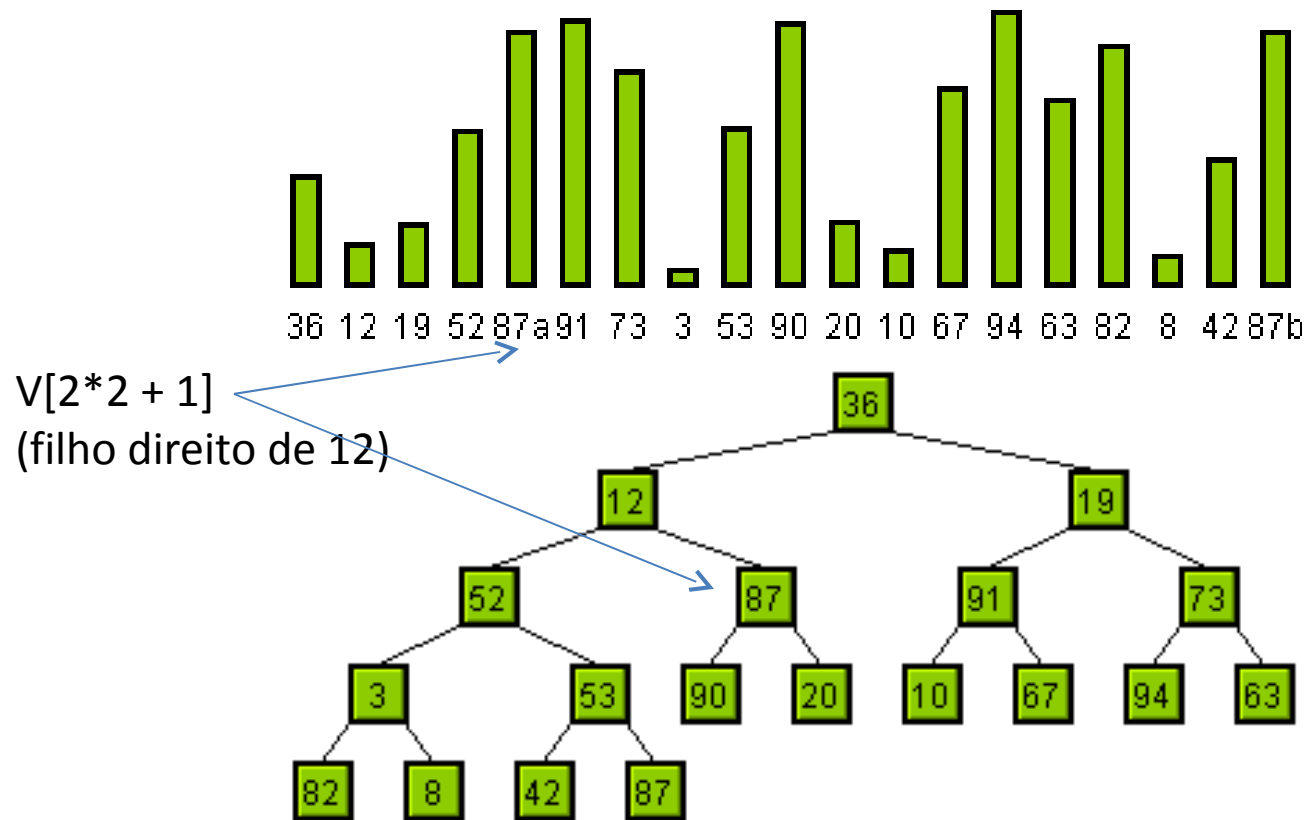
HeapSort

Exemplo:



HeapSort

Exemplo:



HeapSort

- Se as propriedades de um heap são:
 - O valor de cada nó não é menor que os valores armazenados em cada filho
- Então:
 - $v[p] \geq v[2p]$
 - $v[p] \geq v[2p+1]$
- Assim, para transformar o array em um heap, basta reposicionar as chaves utilizando a propriedade acima!

HeapSort (Ordenação por Árvore Binária)

- Para construir um **heap**, é necessário dividir o vetor em duas partes
- Na primeira parte do vetor devem ser considerados os elementos **max-heap**
- Para isso, o algoritmo identifica um índice raiz e suas ramificações da árvore

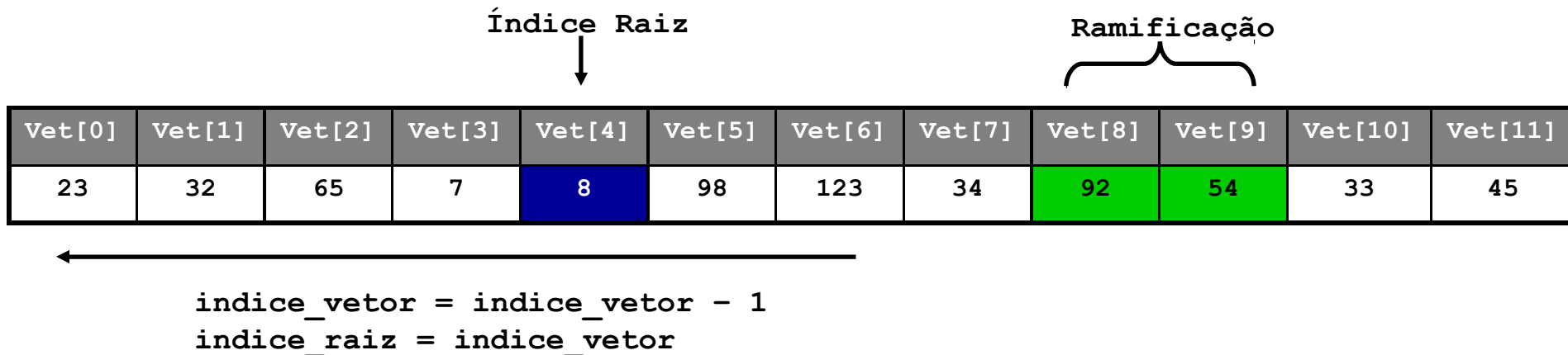
Índice Raiz
↓

Ramificação
┌──────────┐

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	7	8	98	123	34	92	54	33	45

HeapSort (Ordenação por Árvore Binária)

- O algoritmo localiza uma raiz no vetor, baseado no decremento da variável *índice*. Cada valor de índice corresponde a uma raiz (**índice_raiz**)
- O decremento acontece da metade do vetor para cima
- Para cada índice raiz as ramificações são calculadas da seguinte forma:
 - **ramificacao = 2 * índice_raiz**
- Por exemplo:



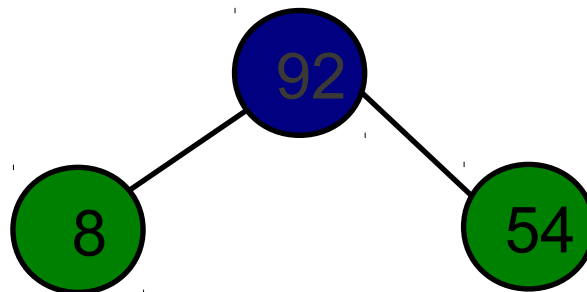
HeapSort (Ordenação por Árvore Binária)

- Se o primeiro elemento ramificado for menor que o segundo, então o algoritmo caminha para a próxima ramificação
- A intenção é localizar o maior elemento (raiz **max-heap**) entre as ramificações. Caso alguma ramificação for maior que o índice raiz, então ocorre a troca entre estes elementos
- Por exemplo:

Índice Raiz
↓

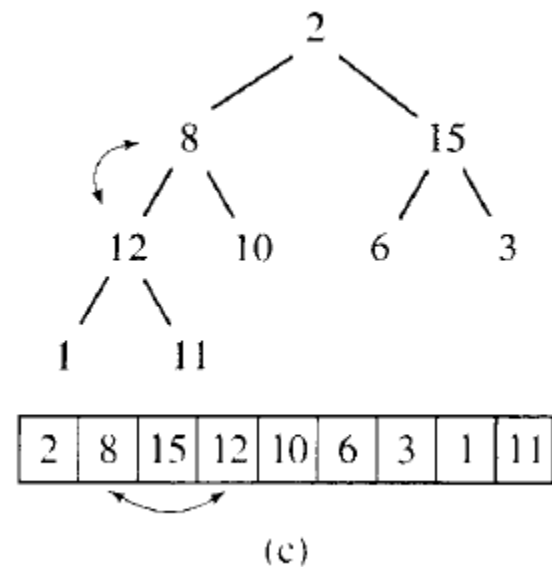
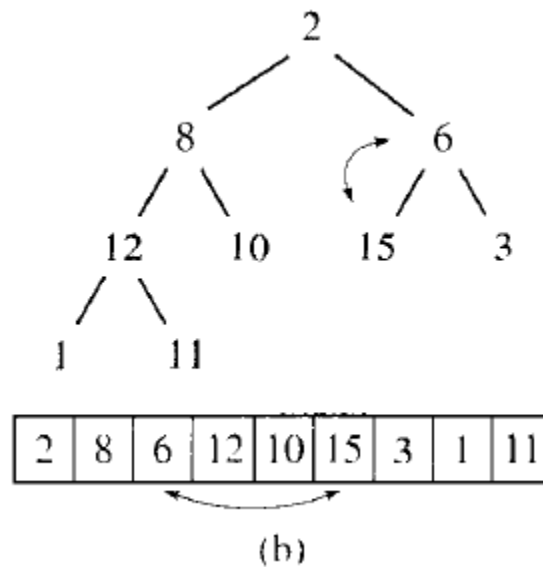
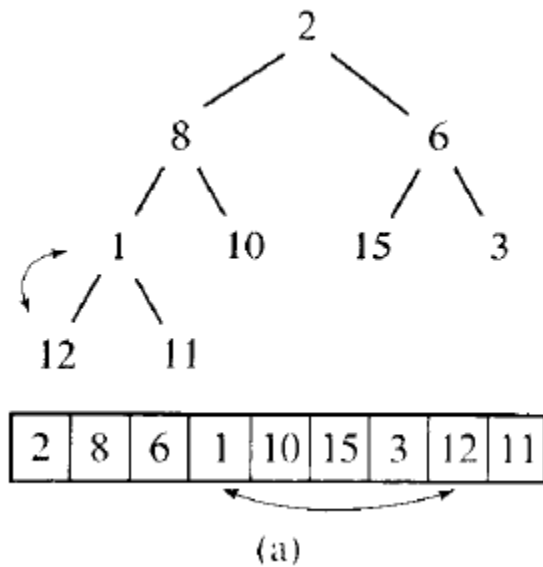
Ramificação
┌──────────┐

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	7	92	98	123	34	8	54	33	45



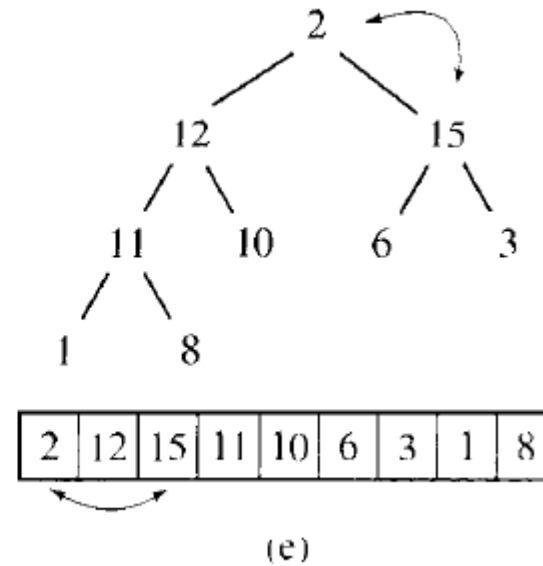
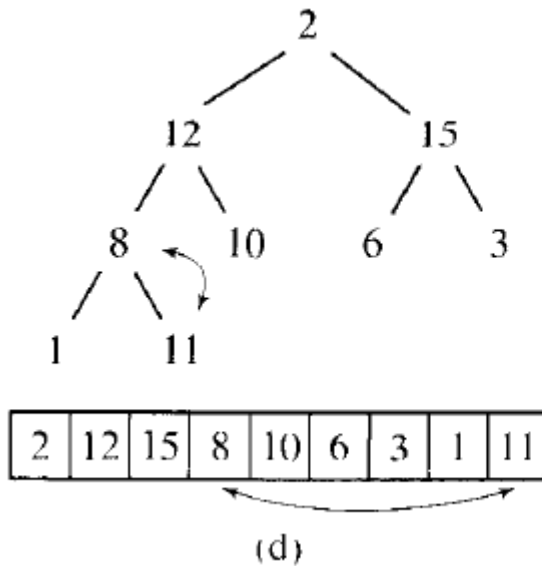
Algoritmo para criação do heap

[2 8 6 1 10 15 3 12 11]



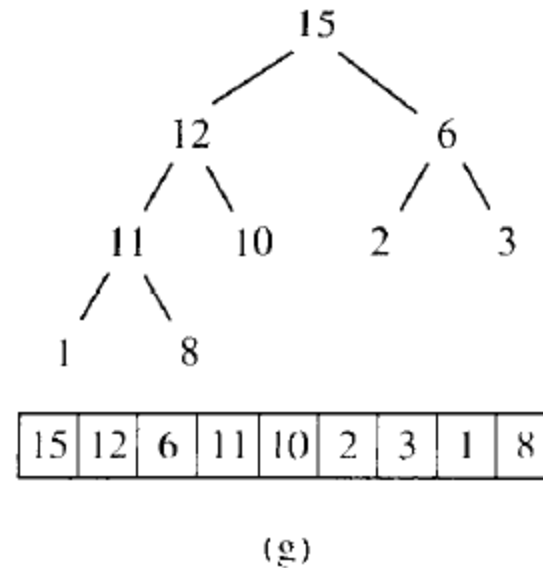
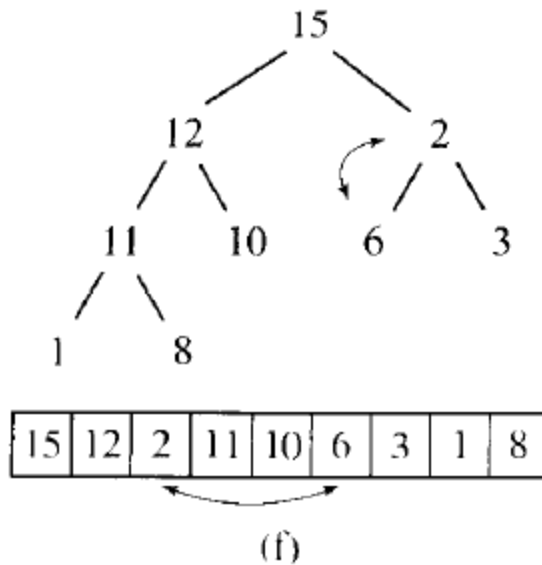
Algoritmo para criação do heap

[2 8 6 1 10 15 3 12 11]



Algoritmo para criação do heap

[2 8 6 1 10 15 3 12 11]



HeapSort (Ordenação por Árvore Binária)

- Código em C para seleção do *max-heap*
- Complexidade $O(\log N)$

```
void constroiHeap( int *p_vetor, int tamanho, int indice_raiz )
{
    int ramificacao, valor;
    valor = p_vetor[ indice_raiz ];

    while( indice_raiz <= tamanho/2 ){
        ramificacao = 2 * indice_raiz;

        if( ramificacao < tamanho && p_vetor[ ramificacao ] < p_vetor[ ramificacao + 1 ] )
            ramificacao++;

        if( valor >= p_vetor[ ramificacao ] )//Identifica o max-heap
            break;

        p_vetor[ indice_raiz ] = p_vetor[ ramificacao ];
        indice_raiz = ramificacao;
    }
    p_vetor[ indice_raiz ] = valor;
}
```

HeapSort (Ordenação por Árvore Binária)

- Aplicando o algoritmo no vetor ***vet***
- $\text{Indice_vetor} = 6$
 - tamanho = 12
 - **indice_raiz = indice_vetor**
 - $\text{ramificacao} = 2 * \text{indice_raiz}$
- Para o nó 123 não houve necessidade de calcular o ***heap-max***, pois não existe nenhum elemento para suas ramificações

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	7	8	98	123	34	92	54	33	45

HeapSort (Ordenação por Árvore Binária)

- $\text{Indice_vetor} = 5$
 - tamanho = 12
 - $\text{indice_raiz} = \text{indice_vetor}$
 - $\text{ramificacao} = 2 * \text{indice_raiz}$
- Para o nó 98 não houve necessidade de calcular o *heap-max*, os elementos de ramificação são menores que o indice_raiz

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	7	8	98	123	34	92	54	33	45

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	7	8	98	123	34	92	54	33	45

HeapSort (Ordenação por Árvore Binária)

- $\text{Indice_vetor} = 4$
 - tamanho = 12
 - $\text{indice_raiz} = \text{indice_vetor}$
 - $\text{ramificacao} = 2 * \text{indice_raiz}$
- Para o nó 8 houve necessidade de calcular o *heap-max*, onde o elemento 8 assume a posição do elemento 92, e vice-versa.

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	7	8	98	123	34	92	54	33	45

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	7	92	98	123	34	8	54	33	45

HeapSort (Ordenação por Árvore Binária)

- $\text{Indice_vetor} = 3$
 - tamanho = 12
 - $\text{indice_raiz} = \text{indice_vetor}$
 - $\text{ramificacao} = 2 * \text{indice_raiz}$
- Para o nó 7 houve necessidade de calcular o *heap-max*, onde o elemento 7 assume a posição do elemento 123 e vice-versa

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	7	92	98	123	34	8	54	33	45

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	123	92	98	7	34	8	54	33	45

HeapSort (Ordenação por Árvore Binária)

- $\text{Indice_vetor} = 2$
 - tamanho = 12
 - $\text{indice_raiz} = \text{indice_vetor}$
 - $\text{ramificacao} = 2 * \text{indice_raiz}$
- Para o nó 65 houve necessidade de calcular o *heap-max*, onde o elemento 65 assume a posição do elemento 98 e vice-versa

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	65	123	92	98	7	34	8	54	33	45

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	98	123	92	65	7	34	8	54	33	45

- O algoritmo continua verificando elementos (ramificações) que sejam maiores que 65, pois elemento também pode ser um *heap-max*

HeapSort (Ordenação por Árvore Binária)

- $\text{Indice_vetor} = 1$
 - tamanho = 12
 - $\text{indice_raiz} = \text{indice_vetor}$
 - $\text{ramificacao} = 2 * \text{indice_raiz}$
- Para o nó 32 houve necessidade de calcular o *heap-max*, onde o elemento 32 assume a posição do elemento 123 e vice-versa.

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	32	98	123	92	65	7	34	8	54	33	45

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	123	98	32	92	65	7	34	8	54	33	45

- O algoritmo continua verificando elementos maiores que a ramificação 32, onde o índice raiz continuou com o valor 3. Neste caso, a verificação de um *heap-max* acontecerá entre as ramificações 7 e 34 ($2 * \text{indice_raiz}$), e ocorre a troca entre os elementos 32 e 34.

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	123	98	34	92	65	7	32	8	54	33	45

HeapSort (Ordenação por Árvore Binária)

- $\text{Indice_vetor} = 0$
 - tamanho = 12
 - $\text{indice_raiz} = \text{indice_vetor}$
 - $\text{ramificacao} = 2 * \text{indice_raiz}$

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
23	123	98	34	92	65	7	32	8	54	33	45

- Para o nó 23 houve necessidade de calcular o heap-max, onde o elemento 123 assume a posição de índice 0. O elemento 23 fica armazenado em memória

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
123	123	98	34	92	65	7	32	8	54	33	45

- O algoritmo se posiciona no índice 1 após a troca, e compara o elemento do mesmo índice com o próximo (índice 2), e efetua a troca de posição

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
123	98	98	34	92	65	7	32	8	54	33	45

HeapSort (Ordenação por Árvore Binária)

- O algoritmo executa novamente a comparação
- Após a troca anterior o vetor se posiciona no elemento 2
- Aplica-se novamente a fórmula **ramificacao = 2 * indice_raiz**
- Novas ramificações são encontradas, e novamente ocorre a comparação para encontrar um *heap-max*

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
123	98	98	34	92	65	7	32	8	54	33	45

- O elemento 92 ocupa a posição 2.

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
123	98	92	34	92	65	7	32	8	54	33	45

HeapSort (Ordenação por Árvore Binária)

- O algoritmo executa novamente a comparação
- Após a troca anterior o vetor se posiciona no elemento 2
- Aplica-se novamente a fórmula **ramificacao = 2 * indice_raiz**
- Novas ramificações são encontradas, e novamente ocorre a comparação para encontrar um *heap-max*

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
123	98	92	34	92	65	7	32	8	54	33	45

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
123	98	92	34	92	65	7	32	8	54	33	45

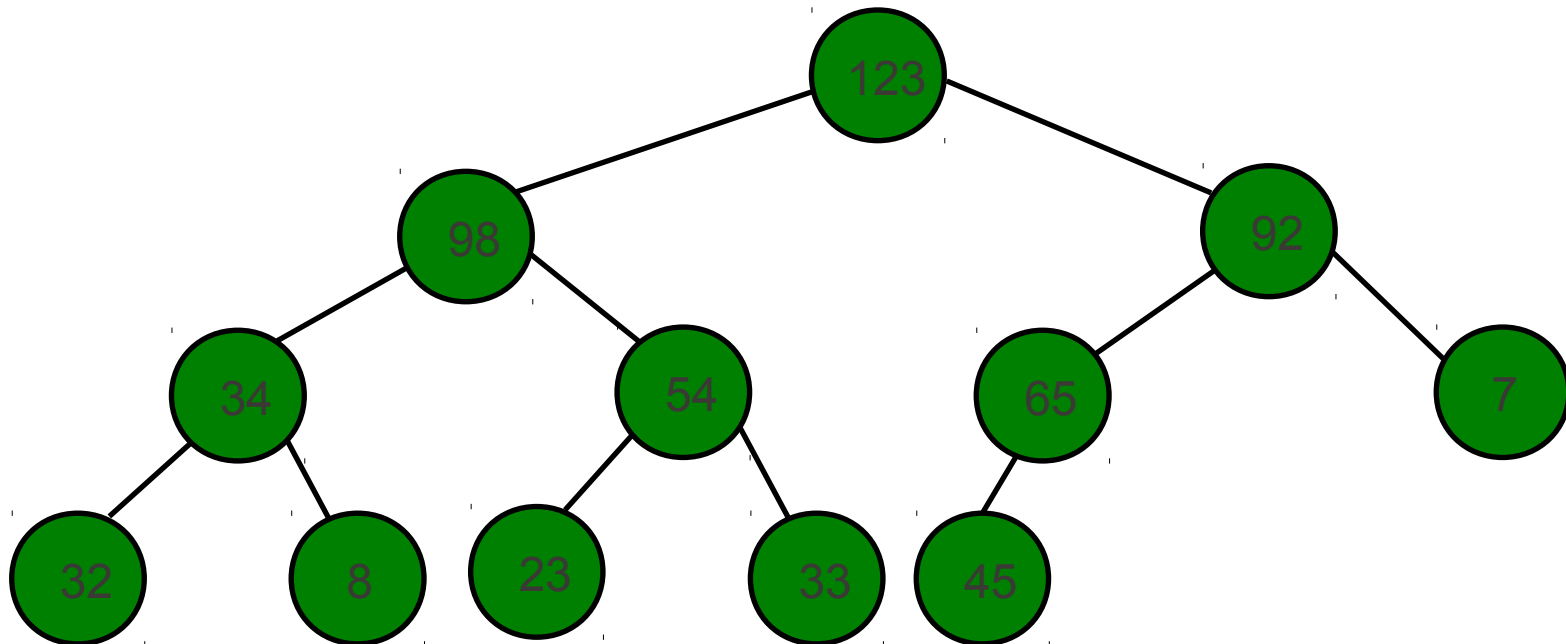
- O elemento 54 ocupa a posição 4, e finalmente o valor 23 ocupa a posição 9 do vetor

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
123	98	92	34	54	65	7	32	8	23	33	45

HeapSort (Ordenação por Árvore Binária)

- O vetor ***vet*** terá a seguinte sequência por ***max-heap***:

Vet[0]	Vet[1]	Vet[2]	Vet[3]	Vet[4]	Vet[5]	Vet[6]	Vet[7]	Vet[8]	Vet[9]	Vet[10]	Vet[11]
123	98	92	34	54	65	7	32	8	23	33	45



HeapSort (Ordenação por Árvore Binária)

- Pode-se então aplicar um algoritmo para inverter os elementos da árvore, do maior elemento da árvore para o menor
 - Ou seja: $V[0] \leftrightarrow V[n]$
- Após a troca, o heap precisa ser refeito
- Em seguida, os passos serão reaplicados nos $n-1$ elementos restantes

HeapSort (Ordenação por Árvore Binária)

- O código abaixo apresenta como esta inversão pode acontecer

```
void HeapSort( int *p_vetor, int tamanho )
{
    int indice, troca;
    for( indice = tamanho/2; indice >= 0; indice-- )
        constroiHeap( p_vetor, tamanho, indice );

    while( tamanho > 0 )
    {
        troca = p_vetor[ 0 ];
        p_vetor[ 0 ] = p_vetor[ tamanho ];
        p_vetor[ tamanho ] = troca;
        constroiHeap( p_vetor, --tamanho, 0 );
    }
}
```

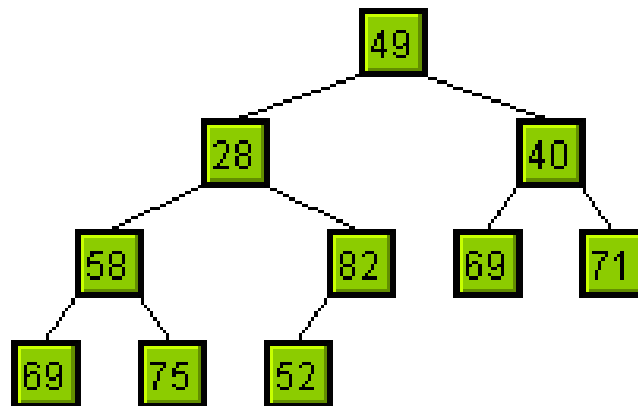
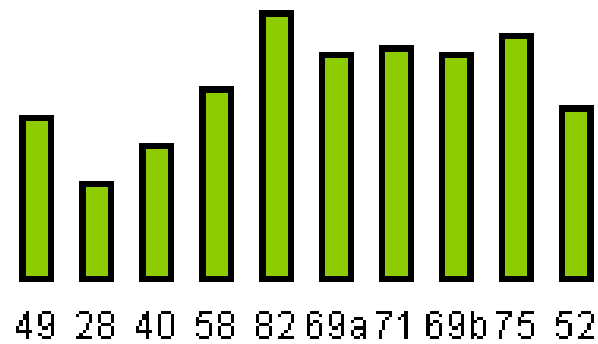
HeapSort (Ordenação por Árvore Binária)

■ O método *main* para execução do *HeapSort*

```
int main(int argc, char *argv[]){  
  
    int vetor[] = {23, 32, 65, 7, 8, 98, 123, 34, 92, 54, 33, 45};  
    int tamanho = 12;  
    int indice;  
  
    HeapSort(vetor, tamanho);  
  
    for (indice=0 ;indice <= tamanho-1 ;indice++){  
        printf("O valor do indice %d ordenado e: %d \n", indice, vetor[indice]);  
    }  
  
    system("PAUSE");  
    return 0;  
  
}
```

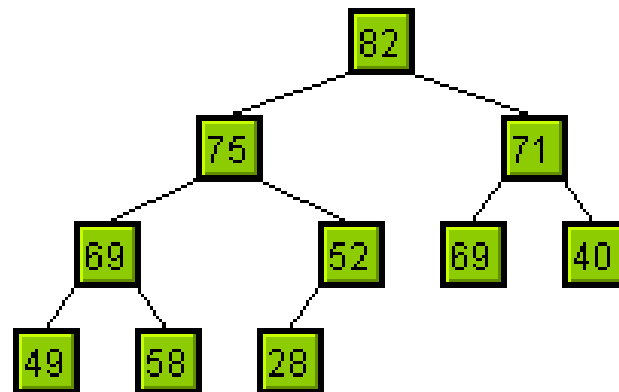
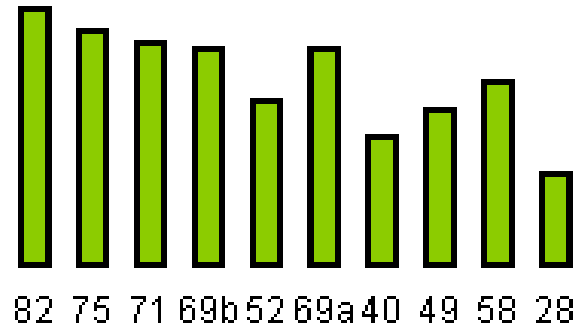
Exemplo

Vetor original



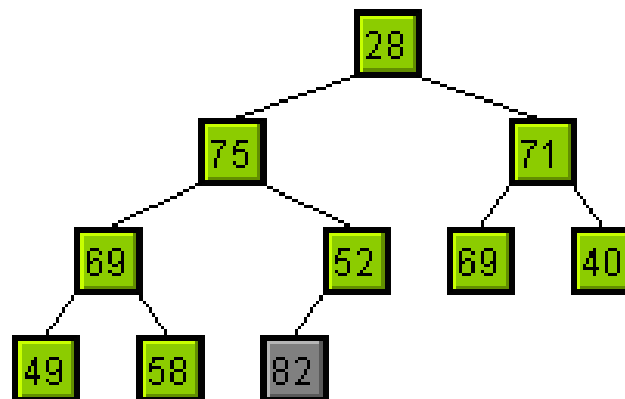
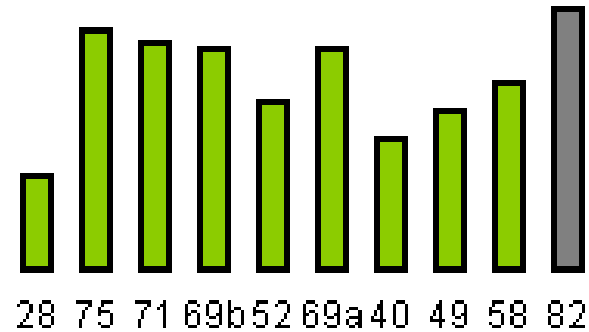
Exemplo

Transformando em
um heap



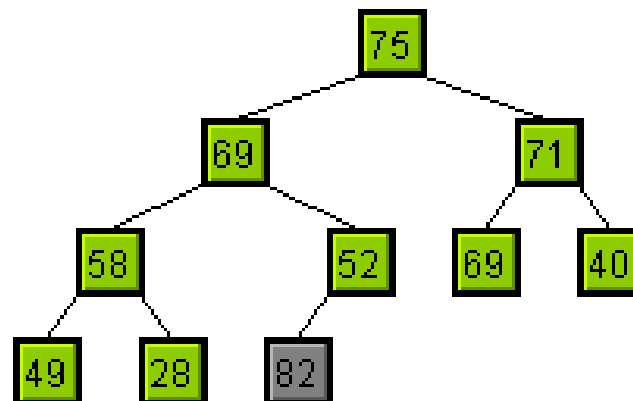
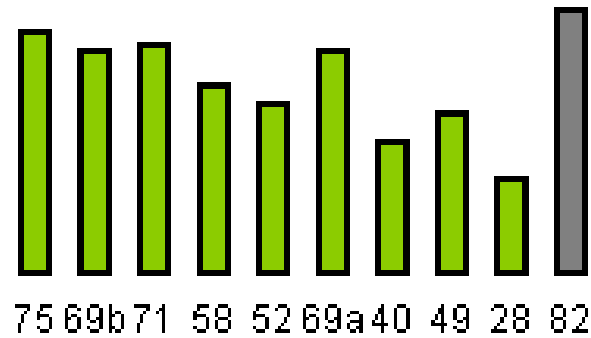
Exemplo

Colocando o maior valor
no final do vetor



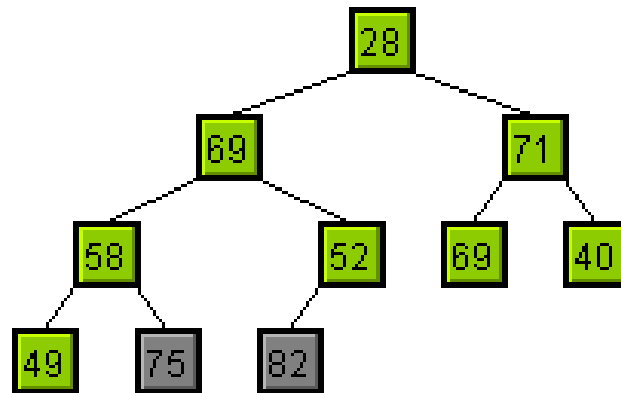
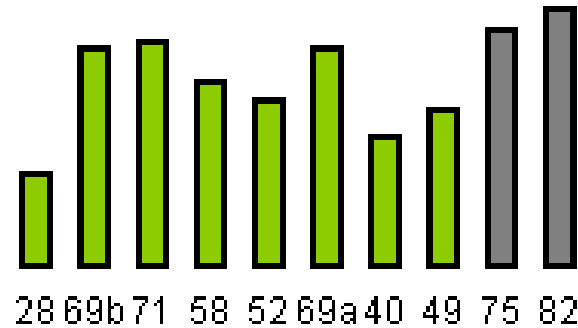
Exemplo

Refazendo o heap...



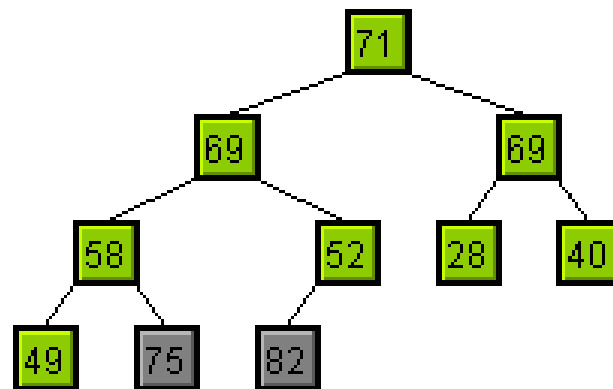
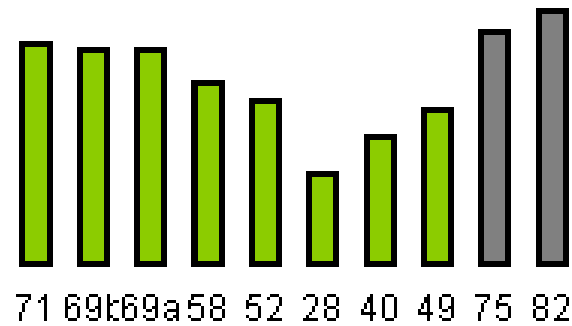
Exemplo

Colocando o maior valor
no final do subvetor



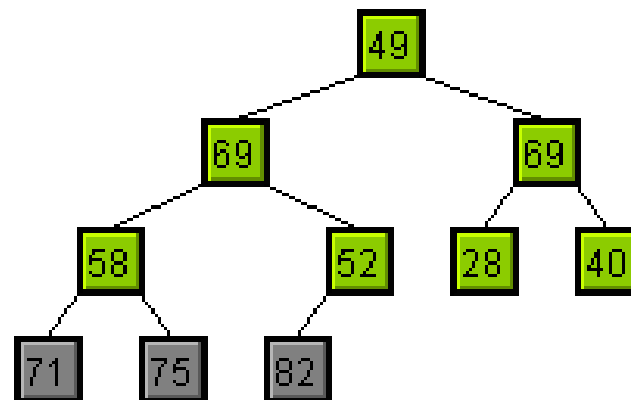
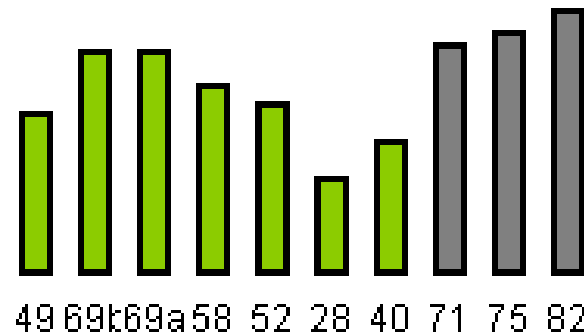
Exemplo

Refazendo o heap...



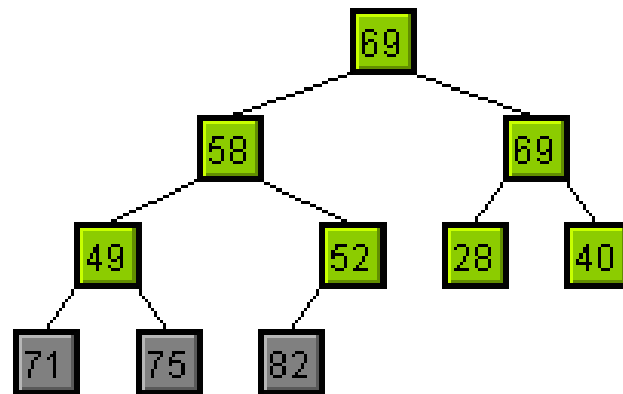
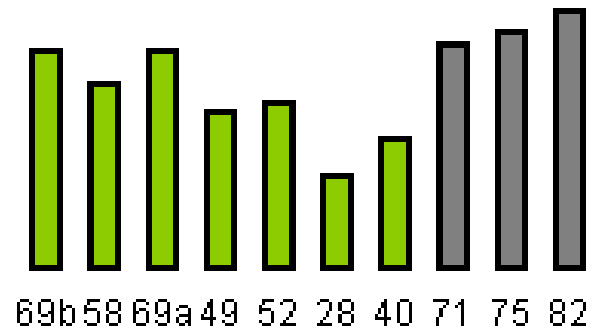
Exemplo

Colocando o maior valor
no final do subvetor



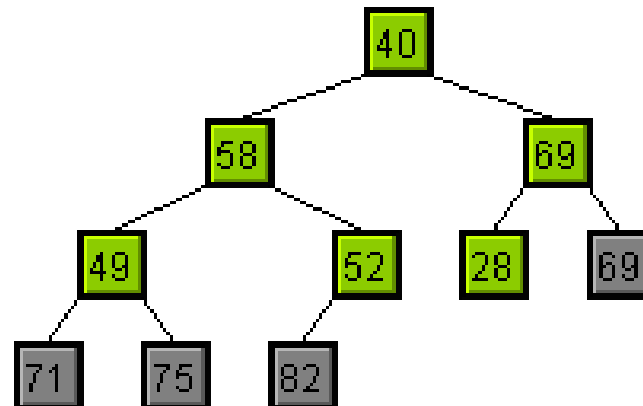
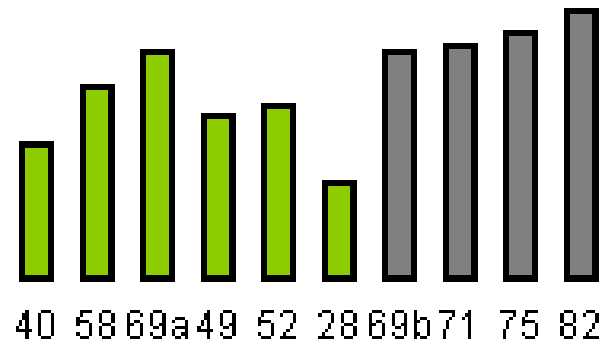
Exemplo

Refazendo o heap...



Exemplo

Colocando o maior valor
no final do subvetor



Estudo da estabilidade

O algoritmo é considerado instável, pois há a possibilidade de elementos com mesma chave mudar de posição no processo de ordenação

Suponha $v = \{4^1, 4^2, 4^3\}$

Heapfy: $v = \{4^1, 4^2, 4^3\}$

Primeira iteração: $v = \{4^3, 4^2, 4^1\}$

Heapfy: $v = \{4^3, 4^2, 4^1\}$

Segunda iteração: $v = \{4^2, 4^3, 4^1\}$