Heaps

Estrutura de Dados

Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação

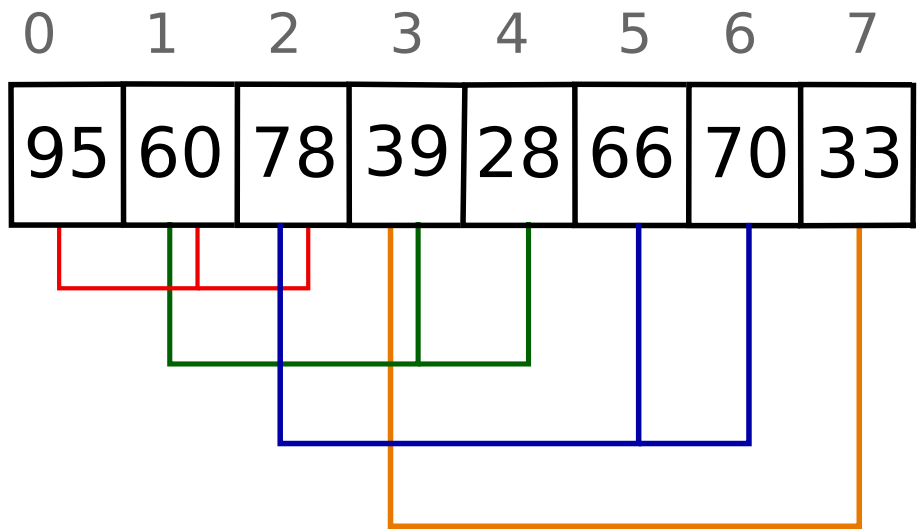
Conteúdo

Introdução

- A *heap* é uma estrutura de dados importante que pode surgir em diversas aplicações.
- Pode ser usada para a implementação de listas de prioridades
  - Casos especiais de listas em que os elementos possuem relações de prioridade entre si.
  - Exemplos de prioridades simples: pilhas e filas.
- Pode ser usada em algoritmos de ordenação:
  - Algoritmo Heapsort.

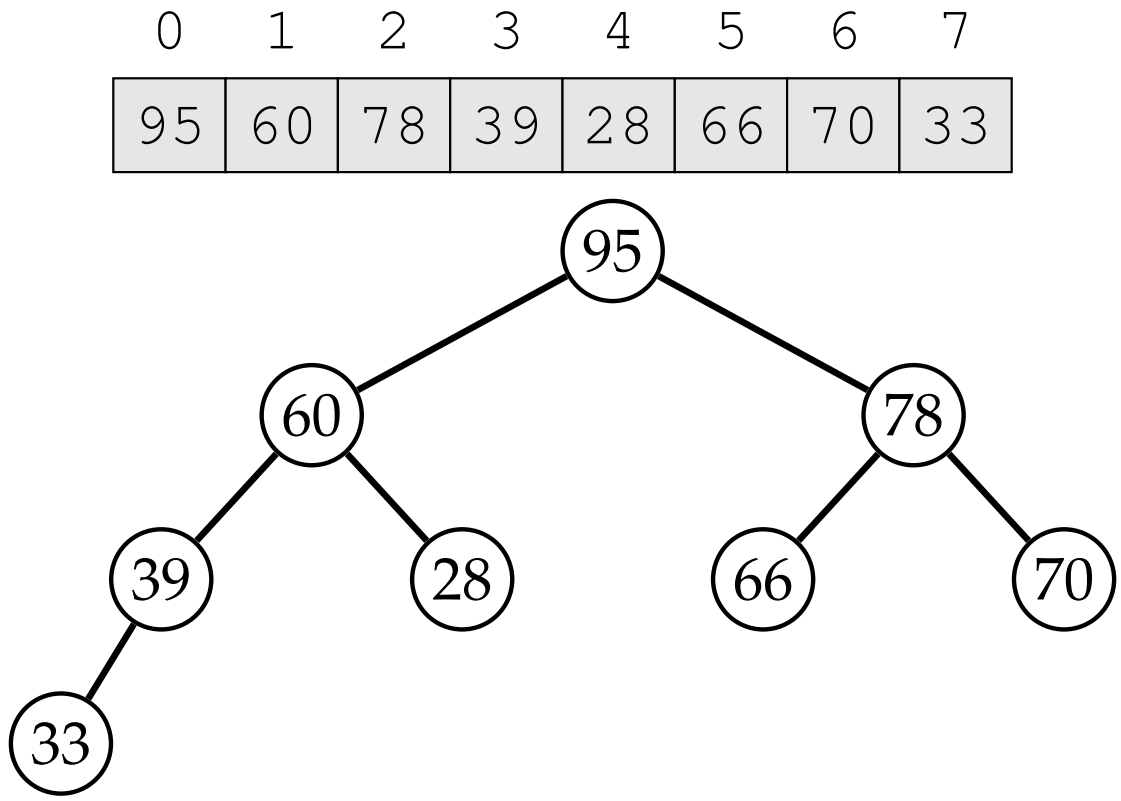
Definição

- Heap é uma lista composta pelos elementos $s_0, \ldots, s_{n-1}$ em que o elemento s_i está ordenado (segundo algum critério de ordenação) em relação ao elemento $s_{(i-1)/d}$.
- Heap binária: $d = 2$.



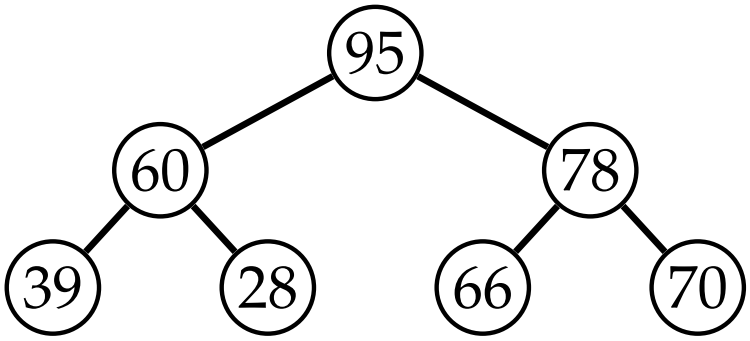
Heap Binária

- Pode-se visualizar uma heap binária como uma **árvore binária completa**, em que cada pai está **ordenado** em relação a seus filhos.
- Árvore binária completa: é uma árvore balanceada na qual a distância da raiz para qualquer folha é h ou $h - 1$.



Heap Binária

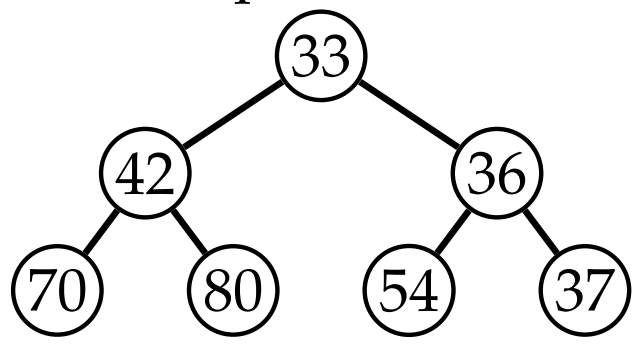
- Max-heap



- Critério de ordenação

$$s_i \leq s_{\lfloor (i-1)/2 \rfloor}$$

- Min-heap



- Critério de ordenação

$$s_i \geq s_{\lfloor (i-1)/2 \rfloor}$$

Heap Binária

Representação

- Encadeada
- **Vetorial**
  - Representação pouco flexível
  - Percurso da árvore é mais simplificado
  - Pode-se acessar qualquer pai ou filho da árvore com apenas um cálculo de índices.
  - Pai: $(i-1)/2$
  - Filho à esquerda: $2i+1$
  - Filho à direita: $2i+2$

Heap Binária

Operações

- Construtor e destrutor
- Inserção
- Remoção
- Consulta raiz
- Os algoritmos de **inserção** e **remoção** consistem em:
  - fazer uma modificação simples e;
  - em seguida, percorrer a **heap** e modificá-la para garantir que o critério de ordenação seja satisfeito em toda a estrutura.

Heap Binária

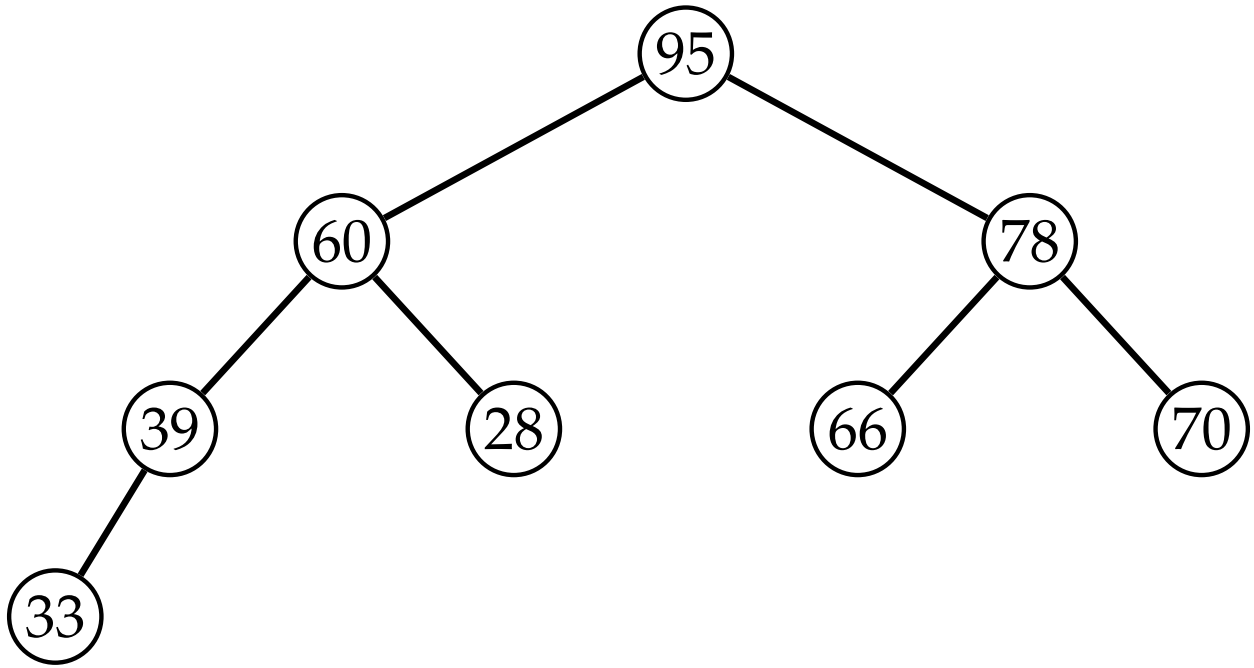
Inserção

1. Inserir um novo nó no final do vetor
2. Incrementar o total de nós da **heap**
3. Corrigir o posicionamento do nó inserido
   a. Verificar se o valor do novo nó deveria vir antes do pai
   b. Caso positivo: troque o nó com o pai
   c. Repita os passos (a) e (b) enquanto a verificação do passo (a) for verdadeira ou enquanto não atingir o início da **heap**

Heap Binária

Exemplo de Inserção

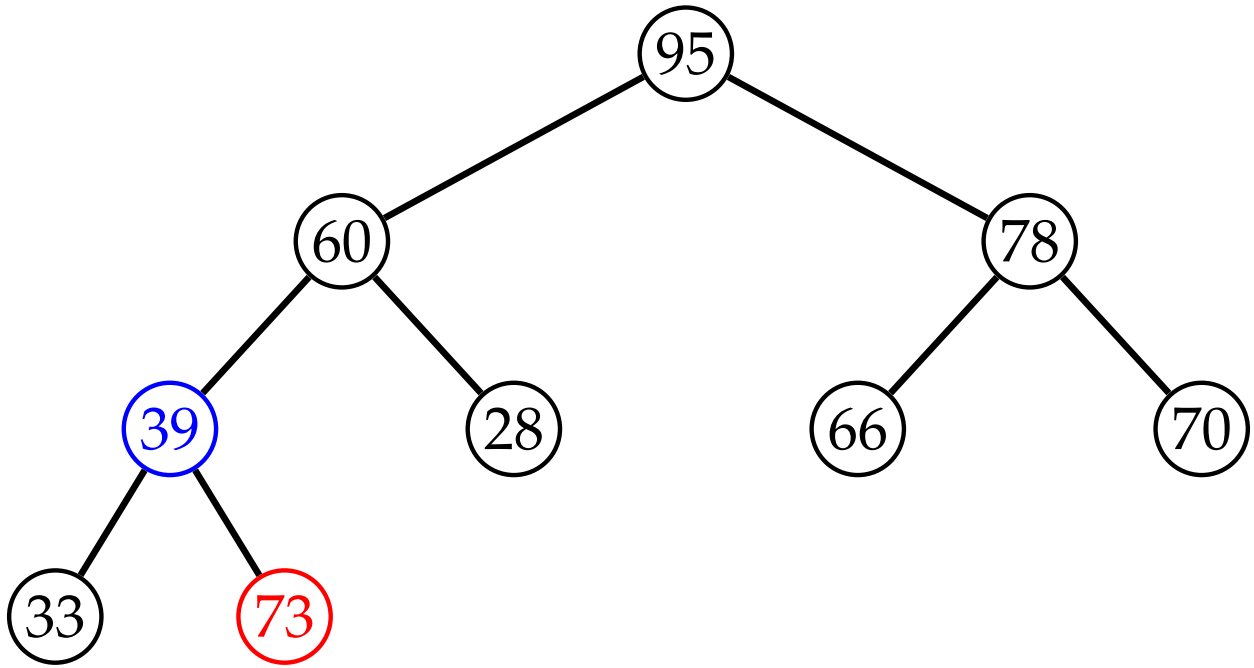
- Inserção do nó 73



Heap Binária

Exemplo de Inserção

- Inserção do nó 73

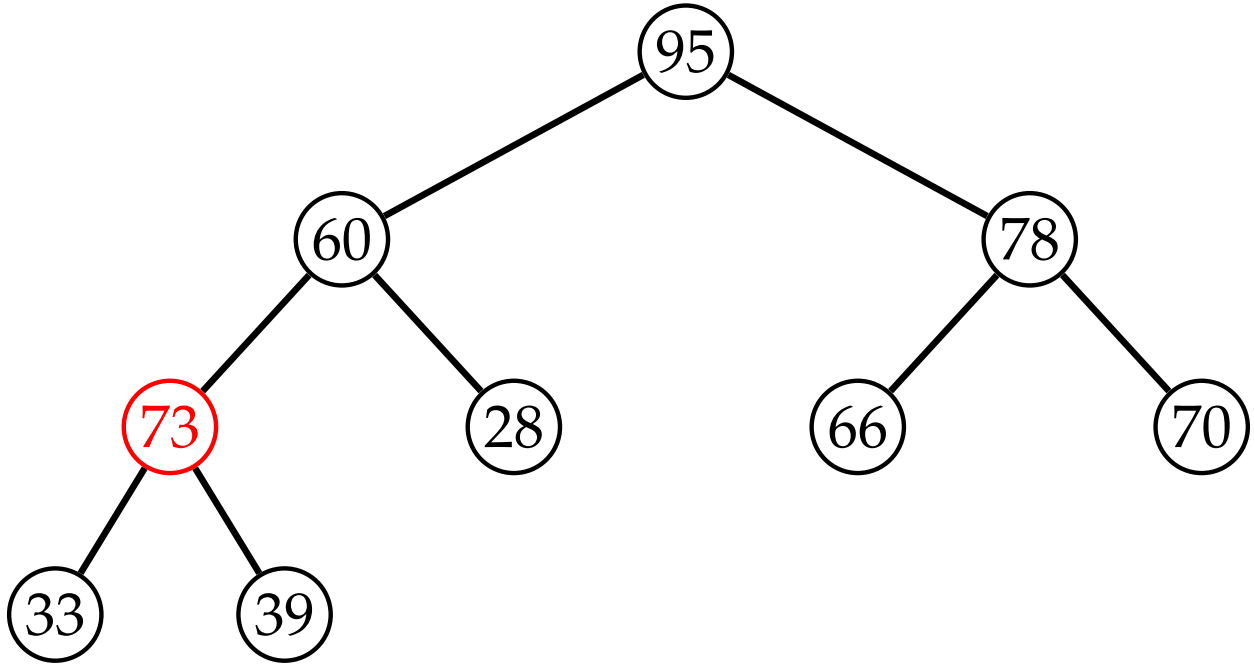


- $73 < 39$?
  - Se for menor, termina;
  - Senão, troca os valores.

Heap Binária

Exemplo de Inserção

- Inserção do nó 73

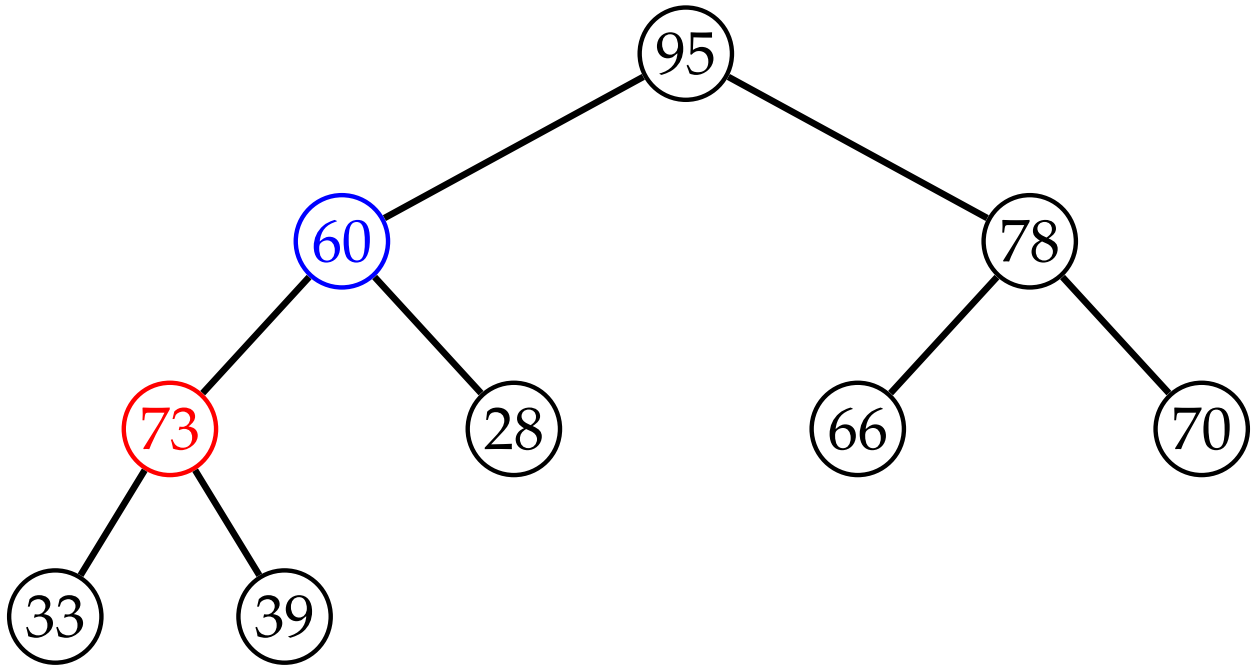


- $73 \leftrightarrow 39$

Heap Binária

Exemplo de Inserção

- Inserção do nó 73

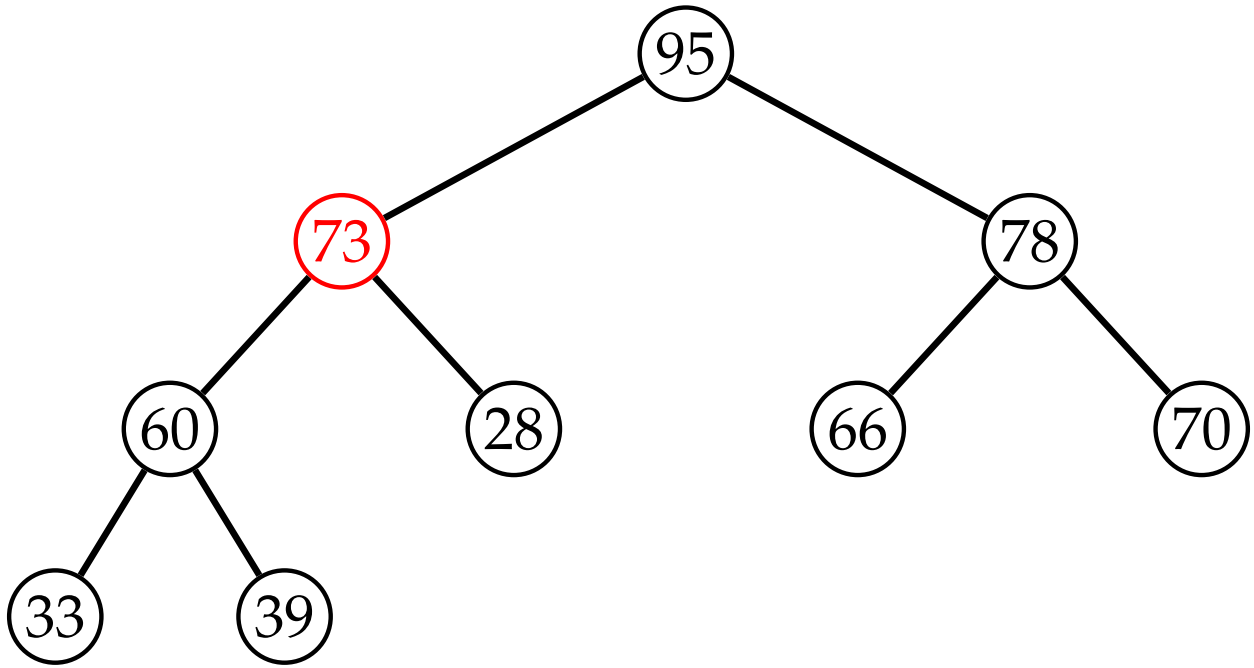


- $73 < 60$?
  - Se for menor, termina;
  - Senão, troca os valores.

Heap Binária

Exemplo de Inserção

- Inserção do nó 73

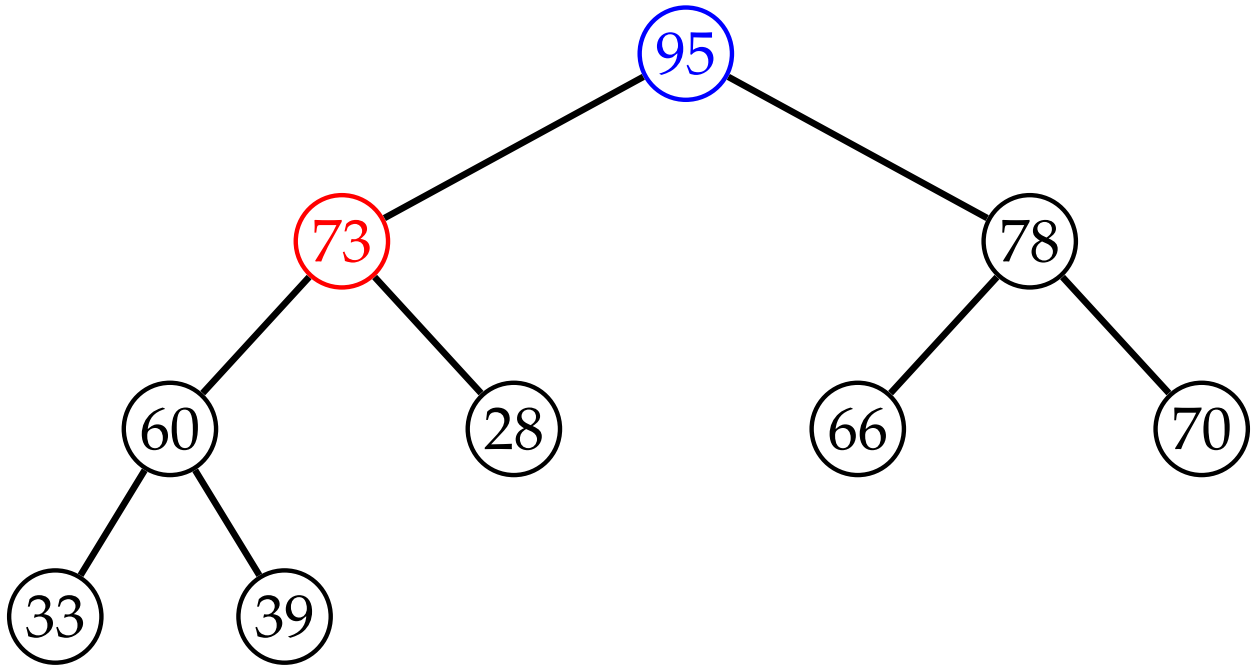


- $73 \leftrightarrow 60$

Heap Binária

Exemplo de Inserção

- Inserção do nó 73

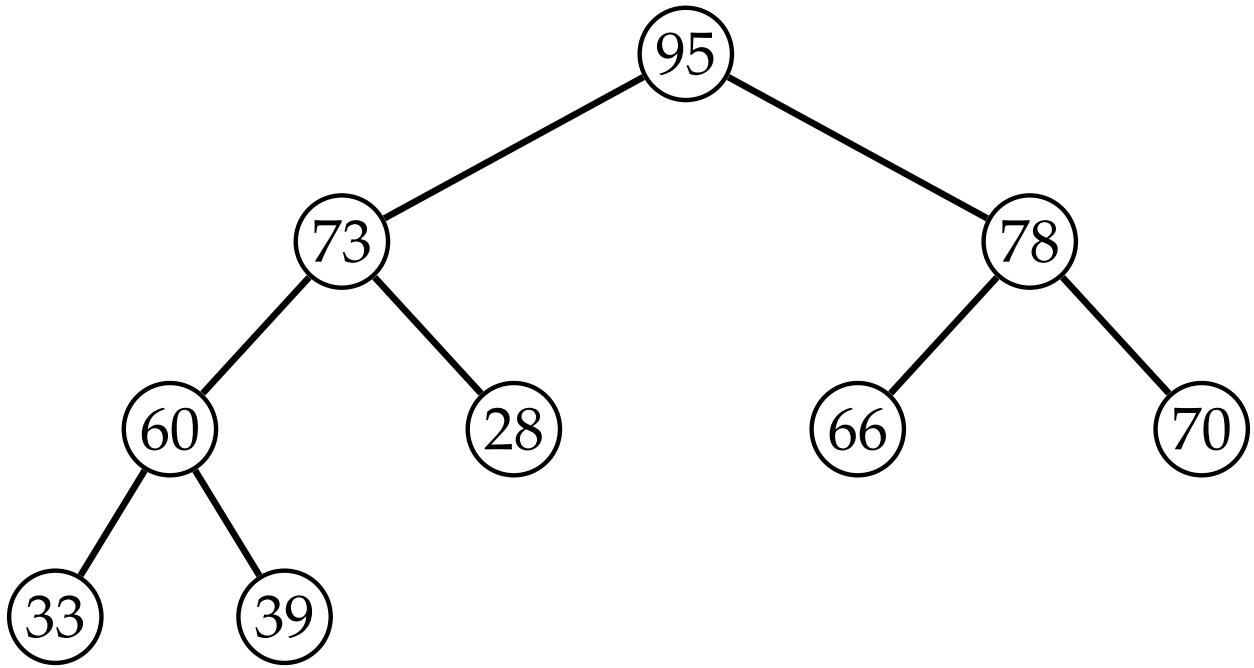


- $73 < 95$?
  - Se for menor, **termina**;
  - Senão, troca os valores.

Heap Binária

Exemplo de Inserção

- Situação final após a inserção do nó 73



Heap Binária

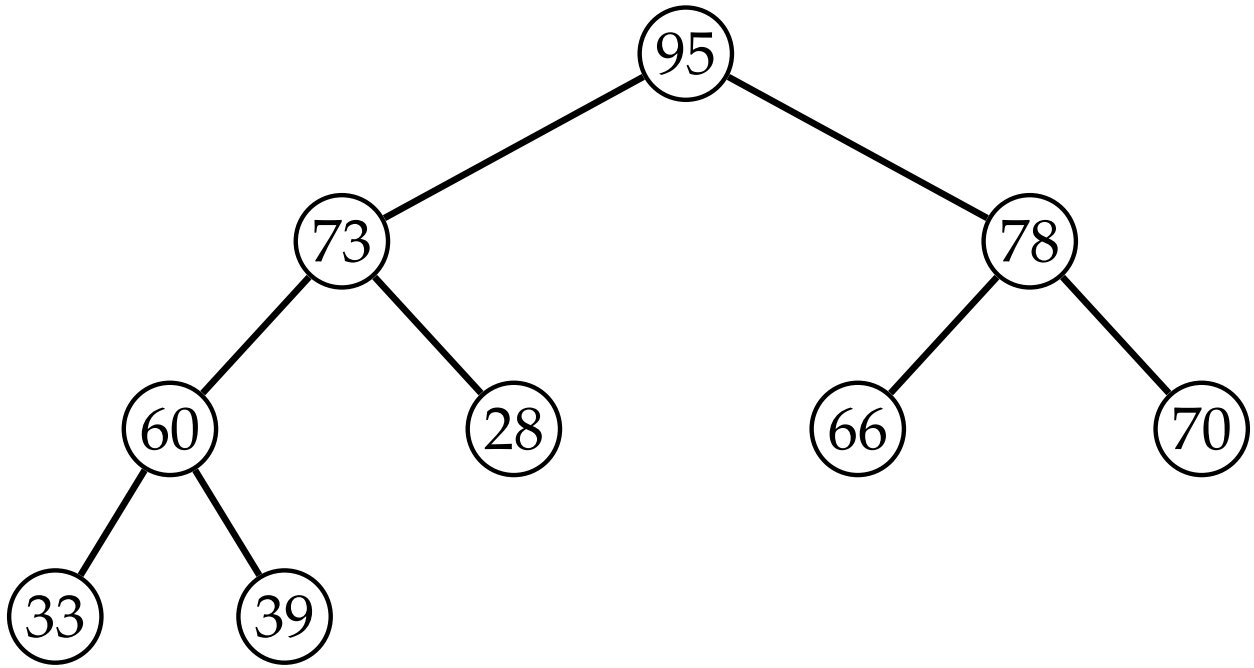
Remoção

1. Colocar o último nó na primeira posição do vetor
2. Decrementar o total de nós da **heap**
3. Corrigir o posicionamento do nó alterado
   a. Verificar por qual caminho descer na árvore: escolher o filho que tiver maior valor (max-heap) ou menor valor (min-heap)
   b. Verificar se o filho deveria vir antes do nó
   c. Caso positivo: troque o nó com o filho
   d. Repita os passos (a),(b) e (c) e enquanto a verificação do passo (b) for verdadeira ou enquanto não atingir o fim da **heap**

Heap Binária

Exemplo de Remoção

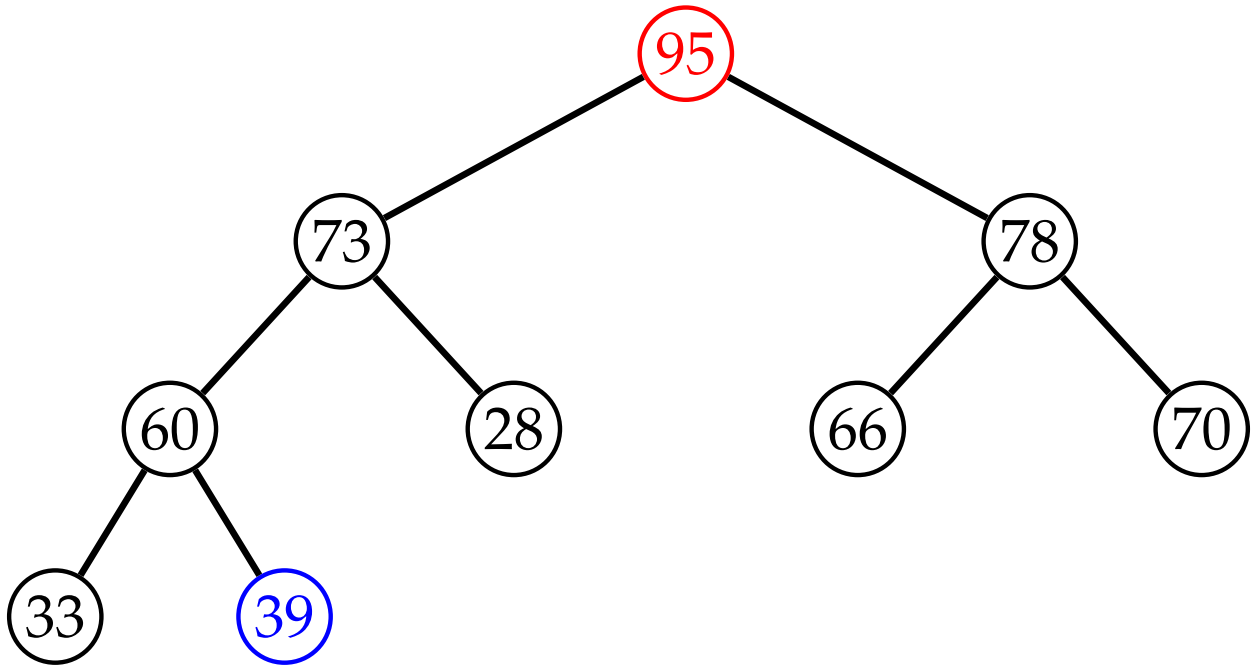
- Remoção do nó 95



Heap Binária

Exemplo de Remoção

- Remoção do nó 95

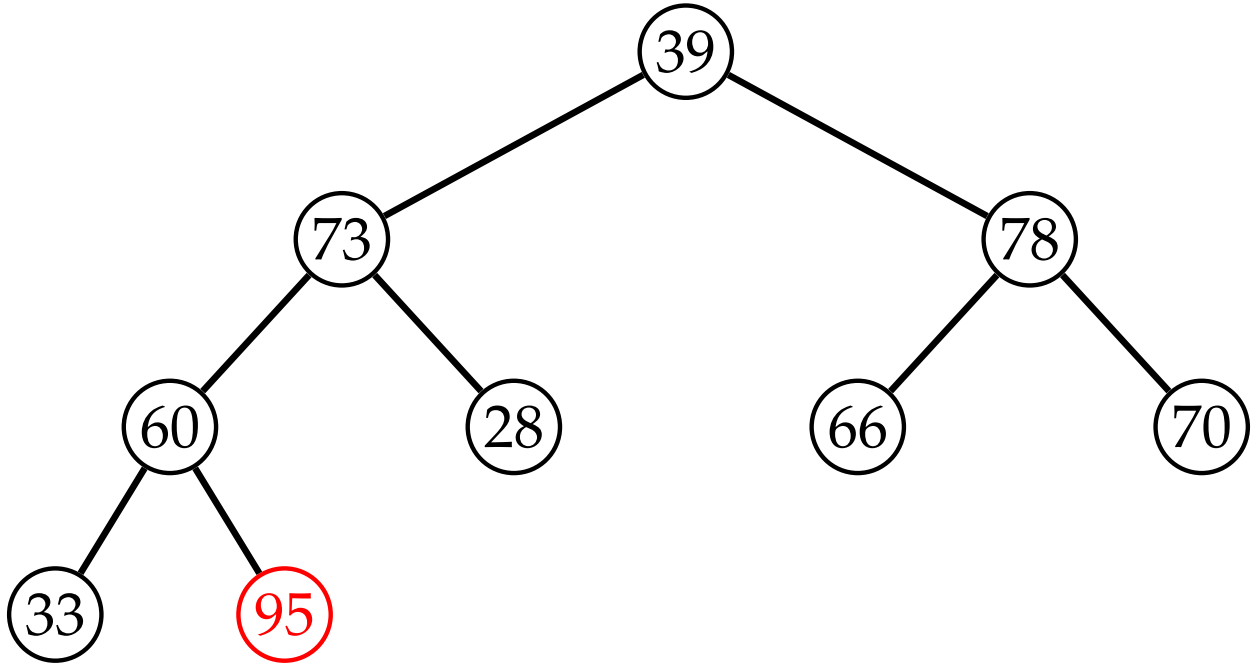


- Troca o nó 39 de posição com o nó 95 (posição 0)

Heap Binária

Exemplo de Remoção

- Remoção do nó 95

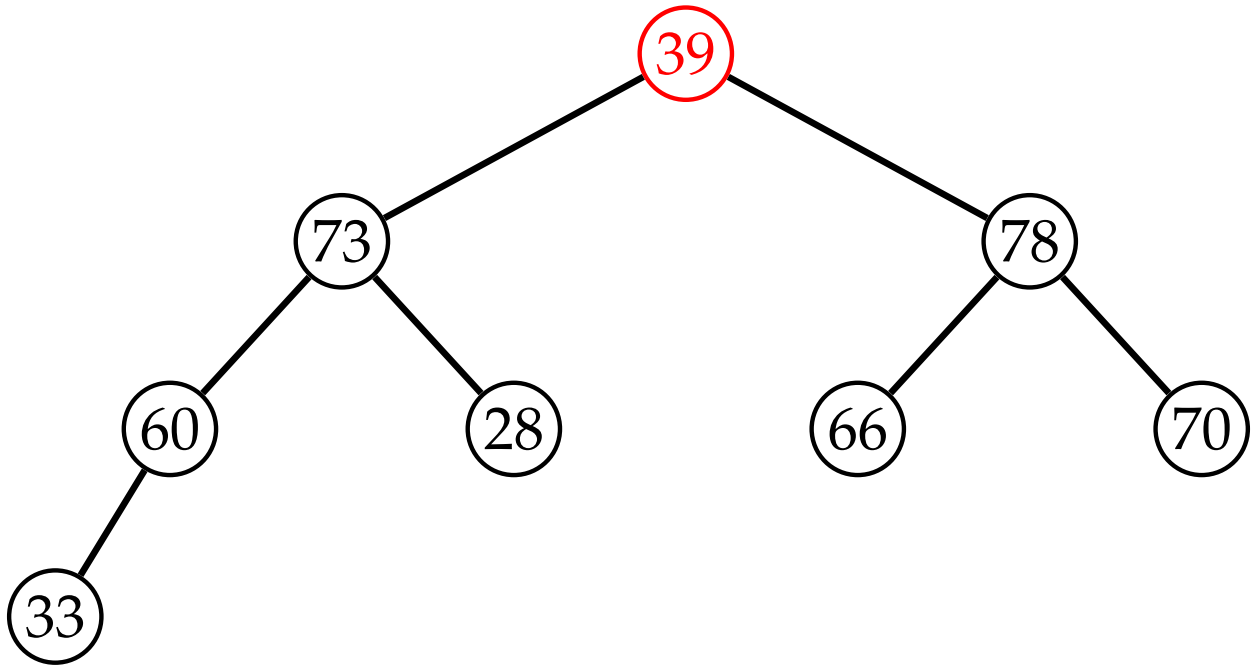


- Diminui o total de nós

Heap Binária

Exemplo de Remoção

- Remoção do nó 95

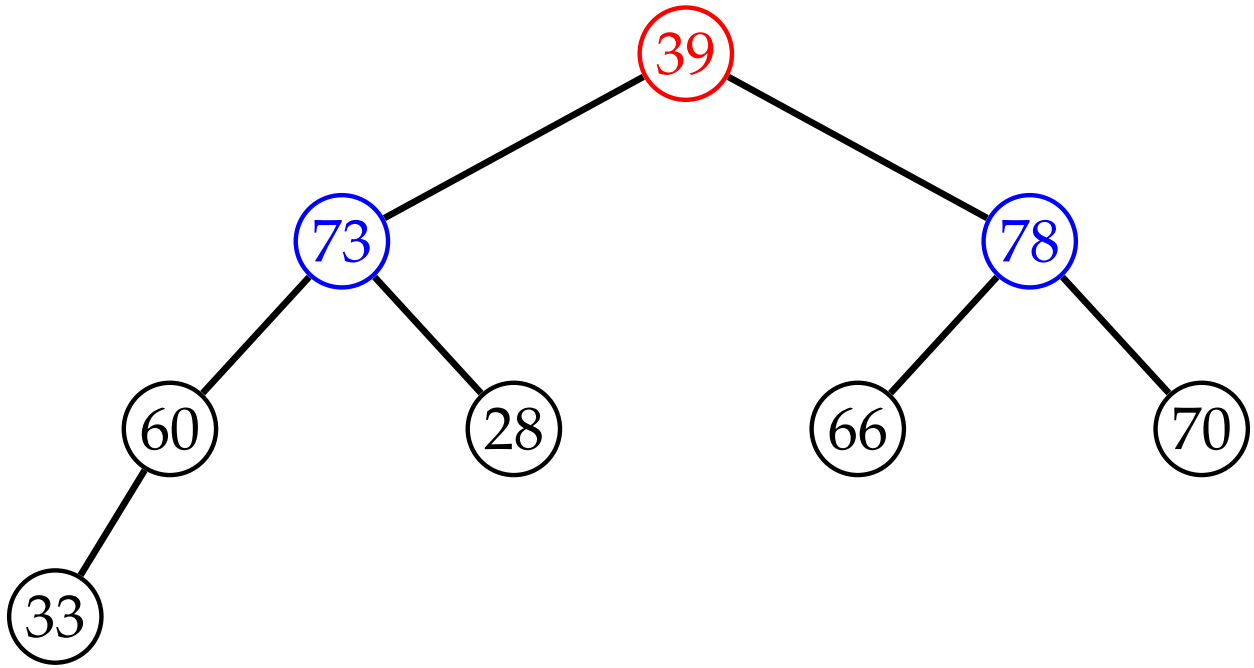


- Desce com o nó 39 para a posição correta.

Heap Binária

Exemplo de Remoção

- Remoção do nó 95

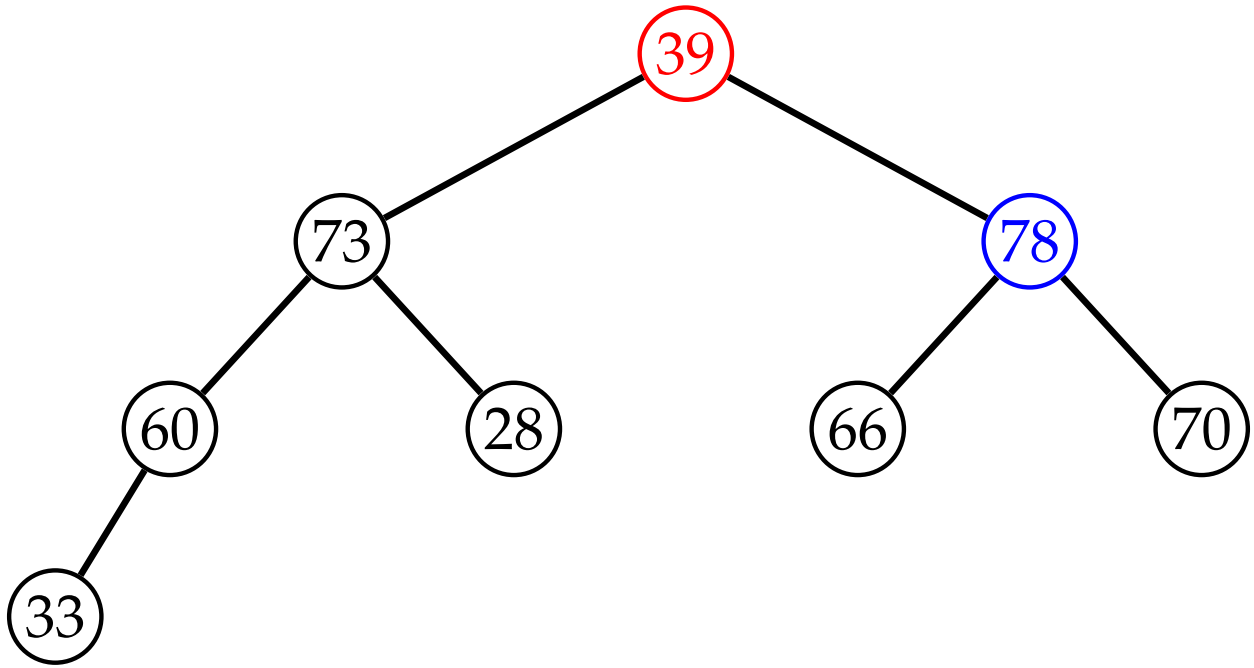


- Seleciona o maior entre os filhos.

Heap Binária

Exemplo de Remoção

- Remoção do nó 95

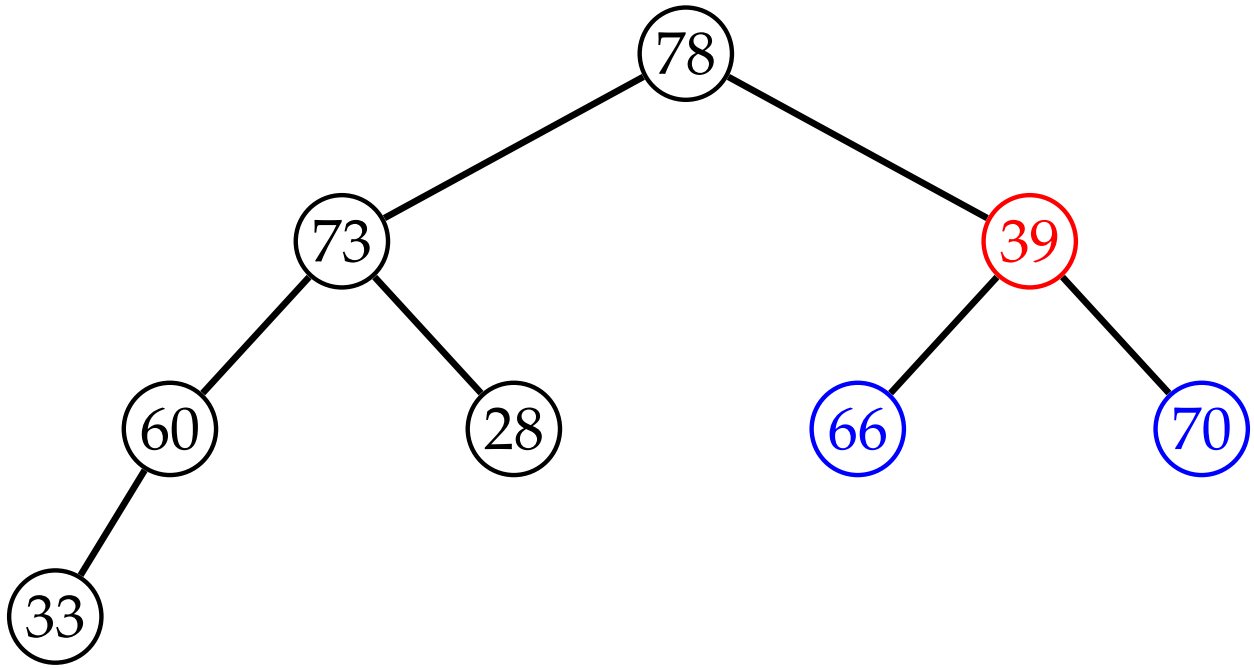


- $39 < 78$?
  - Se sim, desce com 39;
  - Senão, termina.

Heap Binária

Exemplo de Remoção

- Remoção do nó 95

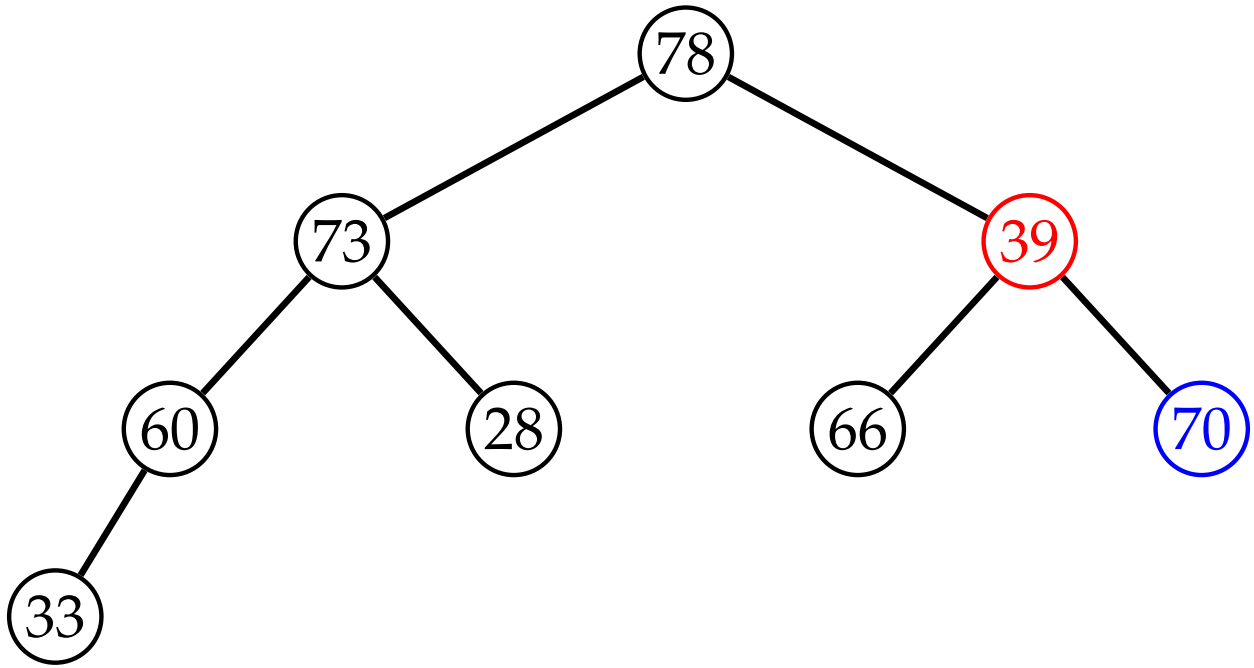


- Seleciona o maior entre os filhos.

Heap Binária

Exemplo de Remoção

- Remoção do nó 95

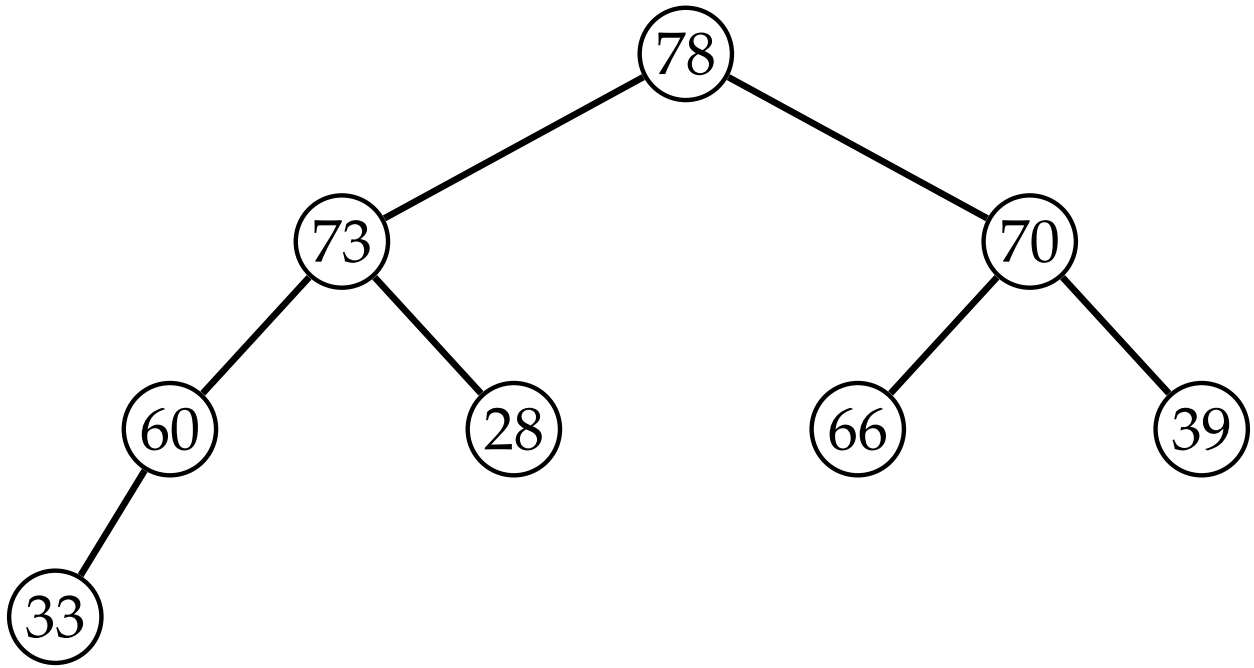


- $39 < 70$?
  - Se sim, desce com 39;
  - Senão, termina.

Heap Binária

Exemplo de Remoção

- Situação final após a remoção do nó 95



TAD HeapMax

- Definição da classe do TAD `HeapMax` (HeapMax.h)

```
class HeapMax
{
  private:
    int m;     // capacidade maxima de elementos
    int n;     // total de elementos corrente
    float *x; // vetor que armazena a heap
    void sobe(int i);
    void desce(int i);

  public:
    HeapMax(int tam);
    ~HeapMax();
    float getRaiz();
    void insere(float val);
    void remove();
};
```

TAD HeapMax

```
HeapMax::HeapMax(int tam)
{
  m = tam;
  n = 0;
  x = new float[tam];
}

HeapMax::~HeapMax()
{
  delete [] x;
}

float HeapMax::getRaiz()
{
  if(n > 0) return X[0];
  else {
    cout << "Heap vazia!" << endl; exit(1);
  }
}
```

TAD HeapMax

```
void HeapMax::insere(float val)
{
  if(n < m)
  {
    x[n] = val;
    n++;
    sobe(n-1);
  }
  else
  {
    cout << "Heap cheia!" << endl;
    exit(1);
  }
}
```

TAD HeapMax

- Implementação **recursiva** da operação `sobe()`

```
void HeapMax::sobe(int filho)
{
  int pai = (filho - 1)/2;
  if(x[filho] > x[pai])
  {
    // troca valores das posicoes pai com filho
    float aux = x[filho];
    x[filho] = x[pai];
    x[pai] = aux;

    sobe(pai);
  }
}
```

TAD HeapMax

```
void HeapMax::remove()
{
  if(n > 0)
  {
    x[0] = x[n-1];
    n--;
    desce(0);
  }
  else
  {
    cout << "Heap vazia!" << endl;
    exit(1);
  }
}
```

TAD HeapMax

- Implementação **recursiva** da operação `desce()`

```
void HeapMax::desce(int pai)
{
  int maxFilho = 2*pai + 1; //supoe filho esq maior
  if(maxFilho < n)
  {
    // determina o indice do maior filho
    if(maxFilho+1 < n)
      if(x[maxFilho+1] > x[maxFilho])
        maxFilho++; // filho dir maior que da esq

    if(x[pai] < x[maxFilho]) {
      float aux = x[pai];
      x[pai] = x[maxFilho];
      x[maxFilho] = aux;
      desce(maxFilho);
    }
  }
}
```

Lista de Prioridade

- Podemos usar uma *heap* para implementar uma lista de prioridades.
- Uma lista de prioridades é uma lista em que cada nó possui, além do valor armazenado, uma informação de prioridade.
- Em alguns casos, o próprio valor pode ser usado como a prioridade.
- Podemos definir o TAD `ListaPrioridade` exatamente como definimos o TAD `HeapMax`, mudando apenas o tipo do nó.

```
typedef struct
{
    float val;
    int prioridade;
} Dupla;
```

Lista de Prioridade

- TAD `ListaPrioridade`

```
class ListaPrioridade
{
  private:
    int m;     // capacidade maxima de elementos
    int n;     // total de elementos corrente
    Dupla *x; // vetor com os dados
    void sobe(int i);
    void desce(int i);
  public:
    ListaPrioridade(int tam);
    ~ListaPrioridade();
    float getRaiz();
    void insere(float val);
    void remove();
};
```

Exercícios

1. Implementar o TAD `ListaPrioridade` com as seguintes operações: construtor, destrutor; operações para inserir, remover, consultar a raiz, subir e descer um nó.
2. Implementar versões `iterativas` dos procedimentos `sobe()` e `desce()` do TAD `HeapMax`.
3. Modificar o TAD `HeapMax` para representar uma *Min-heap*.
4. Implementar uma operação no TAD para construir a heap dado um vetor de elementos.
5. Escrever um programa que usa uma *heap* para ordenar um vetor de 50 elementos lidos pelo usuário. Imprimir o vetor ordenado.