

HEAP

Vanessa Braganholo
Estruturas de Dados e Seus
Algoritmos

PRIORIDADE

Algumas aplicações precisam recuperar rapidamente um dado de maior prioridade

Exemplo: lista de tarefas

- A cada momento, deve-se executar a tarefa que tem mais prioridade
- Selecionar a tarefa mais prioritária de uma lista e retirá-la da lista
- Prioridades **podem mudar**
- Novas tarefas podem chegar e precisam ser acomodadas

LISTA DE PRIORIDADES

Tabela onde cada registro está associado a uma prioridade

Prioridade: valor numérico armazenado em um dos campos do registro

OPERAÇÕES

Seleção do elemento de maior prioridade

Inserção de novo elemento

Remoção do elemento de maior prioridade

Alteração da prioridade de um determinado elemento

IMPLEMENTAÇÃO DE LISTAS DE PRIORIDADE

Lista não ordenada

Lista ordenada

Heap

IMPLEMENTAÇÃO DE LISTAS DE PRIORIDADE

Lista não ordenada

Lista ordenada

Heap

IMPLEMENTAÇÃO POR LISTA NÃO ORDENADA

Inserção e Construção: elementos (registros) podem ser colocados na tabela em qualquer ordem

Remoção: percorrer a tabela sequencialmente em busca do elemento de maior prioridade

Alteração: não implica em mudança na estrutura da tabela, mas exige busca do elemento a ser alterado

Seleção: idem à Remoção

COMPLEXIDADE

Para uma tabela com n elementos

- Seleção: $O(n)$
- Inserção: $O(1)$
- Remoção: $O(n)$
- Alteração: $O(n)$
- Construção: $O(n)$

IMPLEMENTAÇÃO DE LISTAS DE PRIORIDADE

Lista não ordenada

Lista ordenada

Heap

IMPLEMENTAÇÃO POR LISTA ORDENADA

Remoção e Seleção: imediata, pois elemento de maior prioridade é o primeiro da tabela

Inserção: exige percorrer a tabela para encontrar a posição correta de inserção

Alteração: semelhante a uma nova inserção

Construção: exige ordenação prévia da tabela

COMPLEXIDADE

Para uma tabela com n elementos

- Seleção: $O(1)$
- Inserção: $O(n)$
- Remoção: $O(1)$
- Alteração: $O(n)$
- Construção: $O(n \log n)$ (complexidade da ordenação)

IMPLEMENTAÇÃO DE LISTAS DE PRIORIDADE

Lista não ordenada

Lista ordenada

Heap

IMPLEMENTAÇÃO POR HEAP

Mais eficiente na atualização do que as alternativas anteriores, que eram $O(n)$

HEAP

Lista linear (vetor) composta de elementos com chaves s_1, \dots, s_n

Chaves representam as prioridades

Não existem dois elementos com a mesma prioridade

Heap máximo: chaves s_1, \dots, s_n , tal que $s_i \leq s_{i/2}$ para $1 < i \leq n$

Heap mínimo: chaves s_1, \dots, s_n , tal que $s_i \geq s_{i/2}$ para $1 < i \leq n$

Nessa aula, **focaremos em Heap Máximo**

HEAP MÁXIMO

Lista linear composta de elementos com chaves s_1, \dots, s_n , tal que $s_i \leq s_{i/2}$ para $1 < i \leq n$

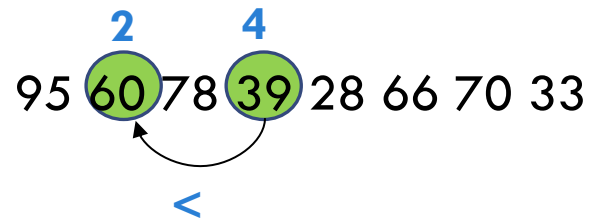
Exemplo

95 60 78 39 28 66 70 33

HEAP MÁXIMO

Lista linear composta de elementos com chaves s_1, \dots, s_n , tal que $s_i \leq s_{i/2}$ para $1 < i \leq n$

Exemplo



HEAP MÁXIMO

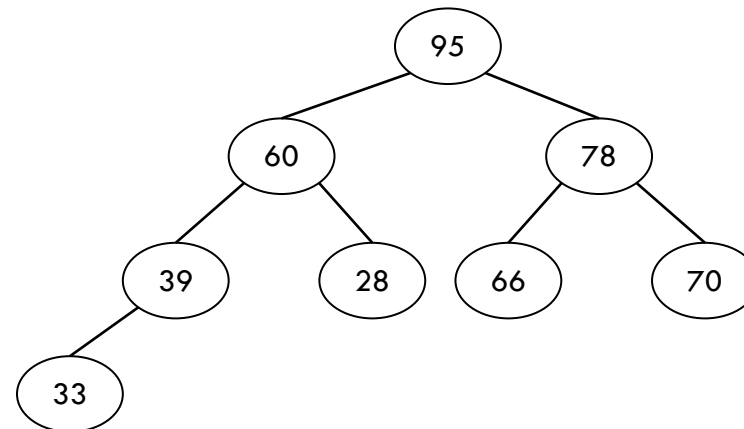
Lista linear composta de elementos com chaves s_1, \dots, s_n , tal que $s_i \leq s_{i/2}$ para $1 < i \leq n$

Exemplo

95 60 78 39 28 66 70 33

↓

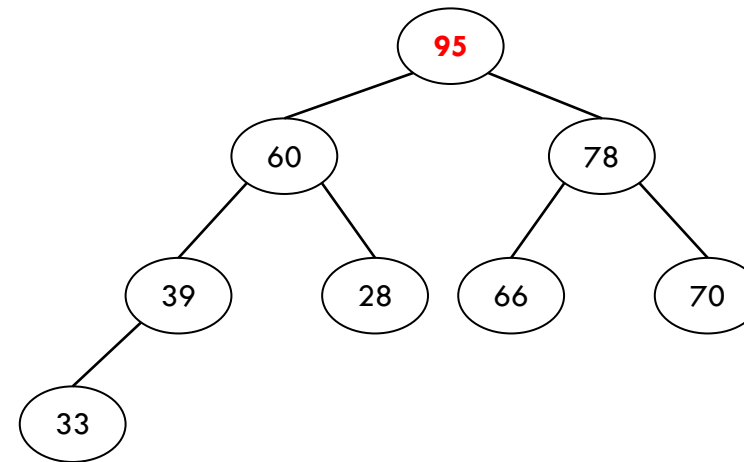
Lista representada como uma árvore binária completa (subárvores vazias apenas no último ou penúltimo nível) →



MONTAGEM DA ÁRVORE BINÁRIA

Nós da árvore são gerados sequencialmente, da raiz para os níveis mais baixos, da esquerda para a direita

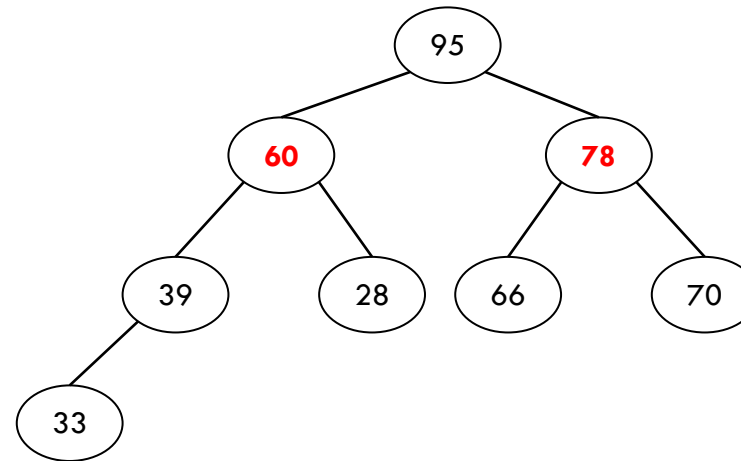
95 60 78 39 28 66 70 33



MONTAGEM DA ÁRVORE BINÁRIA

Nós da árvore são gerados sequencialmente, da raiz para os níveis mais baixos, da esquerda para a direita

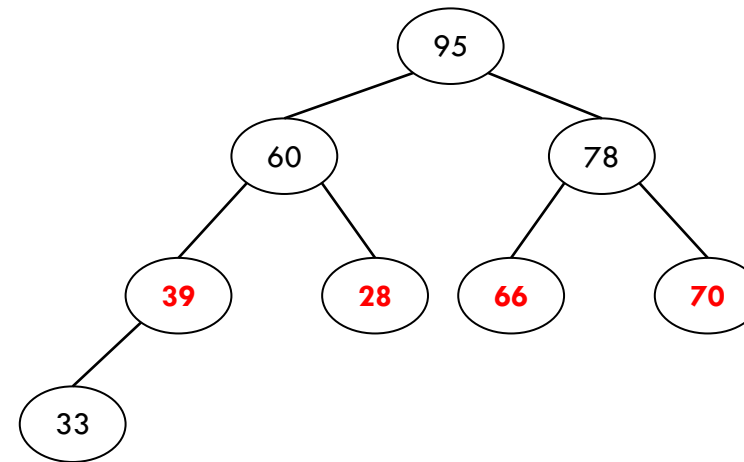
95 **60 78** 39 28 66 70 33



MONTAGEM DA ÁRVORE BINÁRIA

Nós da árvore são gerados sequencialmente, da raiz para os níveis mais baixos, da esquerda para a direita

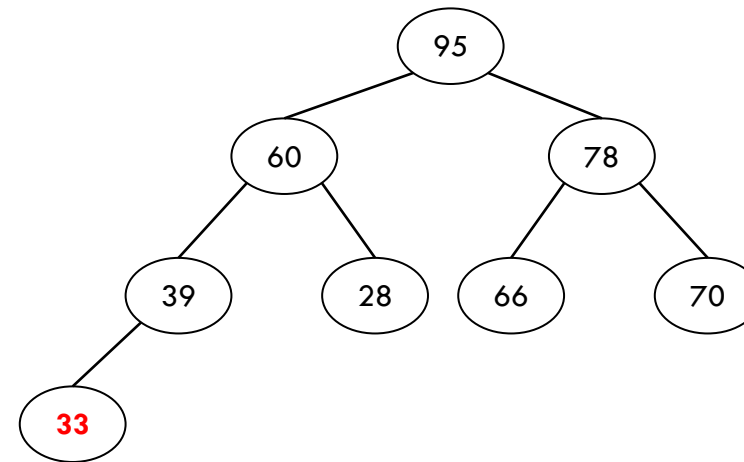
95 60 78 **39 28 66 70** 33



MONTAGEM DA ÁRVORE BINÁRIA

Nós da árvore são gerados sequencialmente, da raiz para os níveis mais baixos, da esquerda para a direita

95 60 78 39 28 66 70 **33**

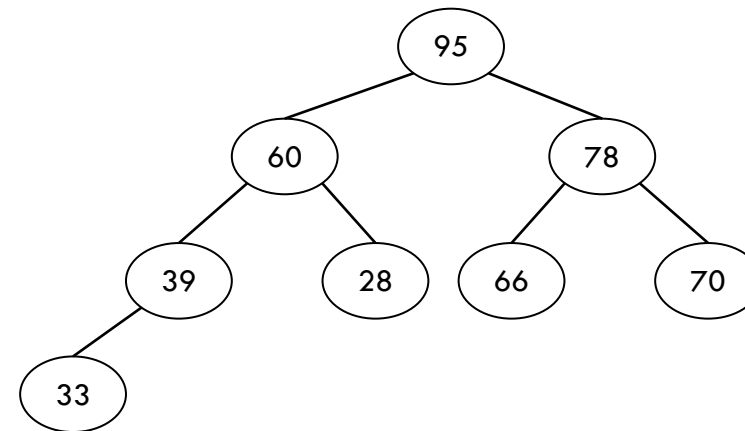
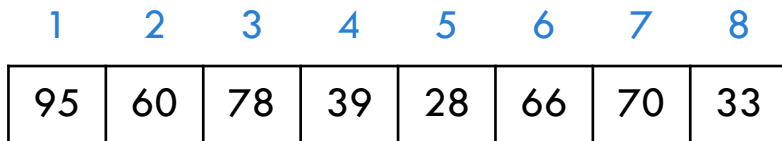


PROPRIEDADES

Cada nó possui **prioridade maior do que seus dois filhos**

O elemento de maior prioridade é sempre a **raiz da árvore**

A representação em memória pode ser feita usando um **vetor** (índice do primeiro elemento é **1**)



IMPLEMENTAÇÃO EM MEMÓRIA

```
int main(void) {
    int *heap;
    int n;

    printf("Digite o tamanho do vetor de elementos: ");
    scanf("%d", &n);
    if(n <= 0) {
        return 0;
    }

    //vetor começará em 1, por isso alocação de tamanho n+1
    heap = (int *) malloc(sizeof(int) * (n + 1));

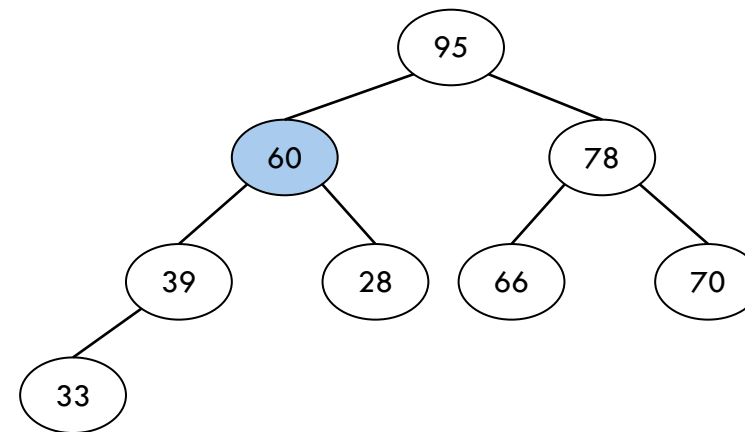
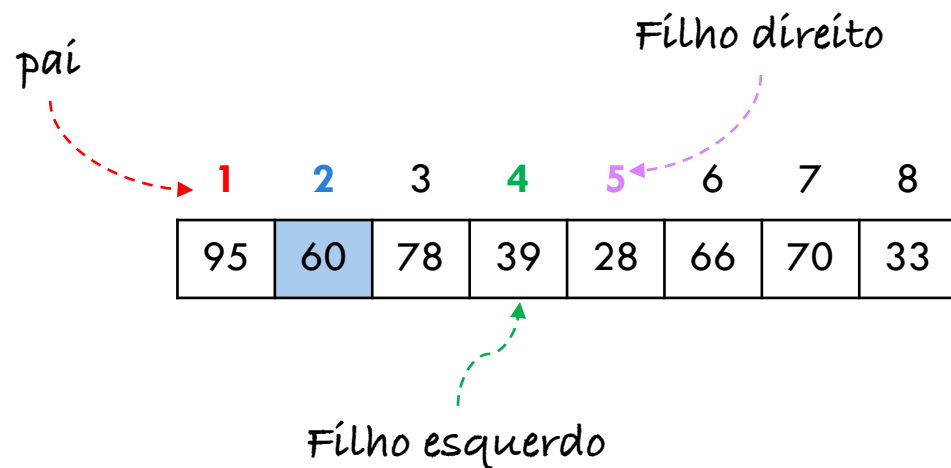
    ...
}
```

Implementação usa vetor de inteiros para simplificar, mas na prática seria um vetor de struct com os dados (da tarefa, por exemplo)

PROPRIEDADES

Para um determinado elemento i :

- pai de i é $i/2$
- filho esquerdo é $i * 2$
- filho direito é $i * 2 + 1$



IMPLEMENTAÇÃO EM MEMÓRIA

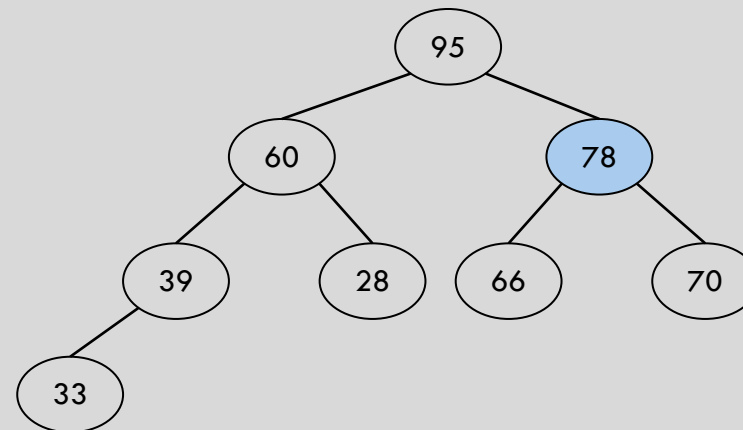
```
/* Lembrar que os índices assumem que o primeiro elemento está na  
posição 1 do vetor, e não na posição 0 */
```

```
int pai(int i){  
    return (i/2);  
}
```

```
int esq(int i){  
    return (i*2);  
}
```

```
int dir(int i){  
    return (i*2+1);  
}
```

1	2	3	4	5	6	7	8
95	60	78	39	28	66	70	33



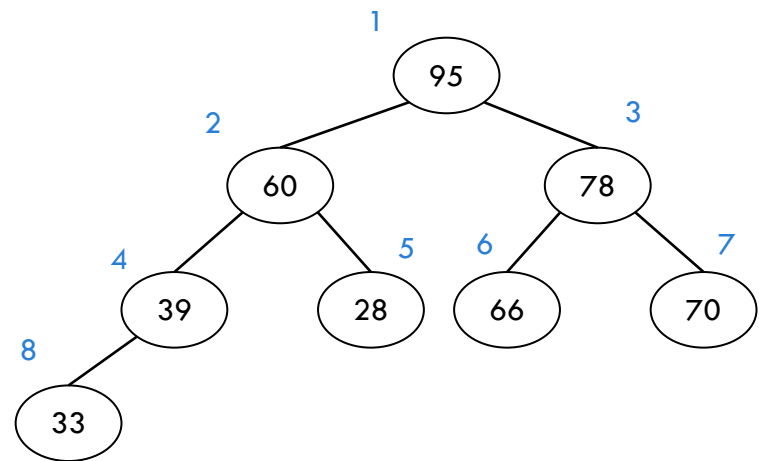
ALTERAÇÃO DE PRIORIDADE

Ao alterar a prioridade de um nó, é necessário re-arrumar a heap para que ela respeite as prioridades

- Um nó que tem a prioridade aumentada precisa “subir” na árvore
- Um nó que tem a prioridade diminuída precisa “descer” na árvore

EXEMPLO:

AUMENTAR A PRIORIDADE DO NÓ 6 DE 66 PARA 98

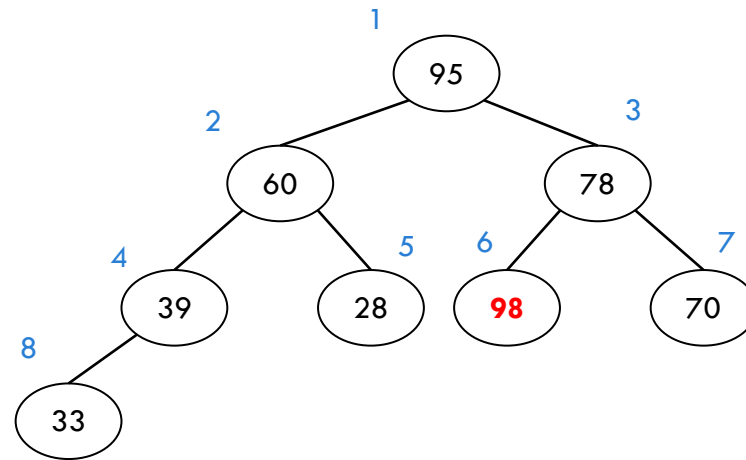


1	2	3	4	5	6	7	8
95	60	78	39	28	66	70	33

EXEMPLO:

AUMENTAR A PRIORIDADE DO NÓ 6 DE 66 PARA 98

Apenas o ramo que vai do nó atualizado até a raiz é afetado – o restante da árvore permanece inalterado

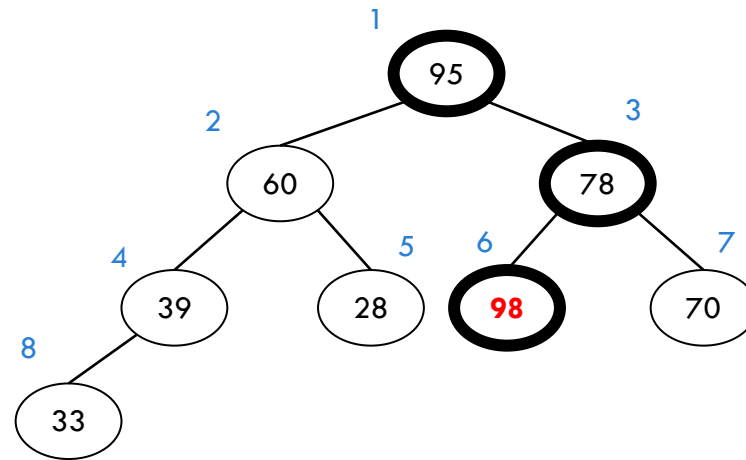


1	2	3	4	5	6	7	8
95	60	78	39	28	98	70	33

EXEMPLO:

AUMENTAR A PRIORIDADE DO NÓ 6 DE 66 PARA 98

Apenas o ramo que vai do nó atualizado até a raiz é afetado – o restante da árvore permanece inalterado

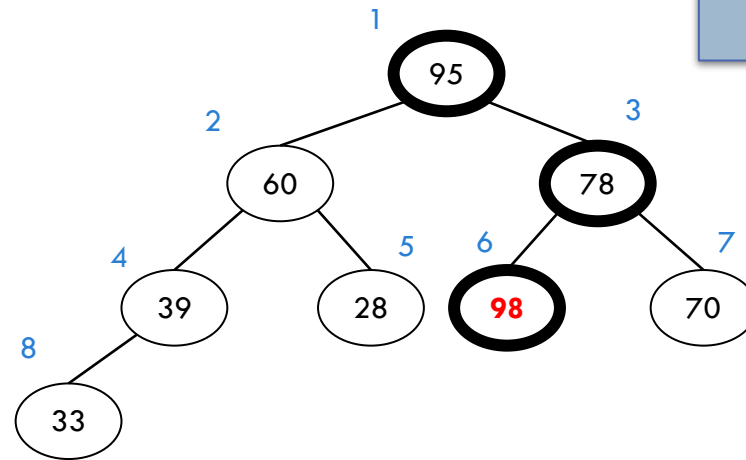


1	2	3	4	5	6	7	8
95	60	78	39	28	98	70	33

EXEMPLO:

AUMENTAR A PRIORIDADE DO NÓ 6 DE 66 PARA 98

“Subir” elemento alterado na árvore, fazendo trocas com o nó pai, até que a árvore volte a ficar correta (todo nó tem prioridade maior que seus filhos)

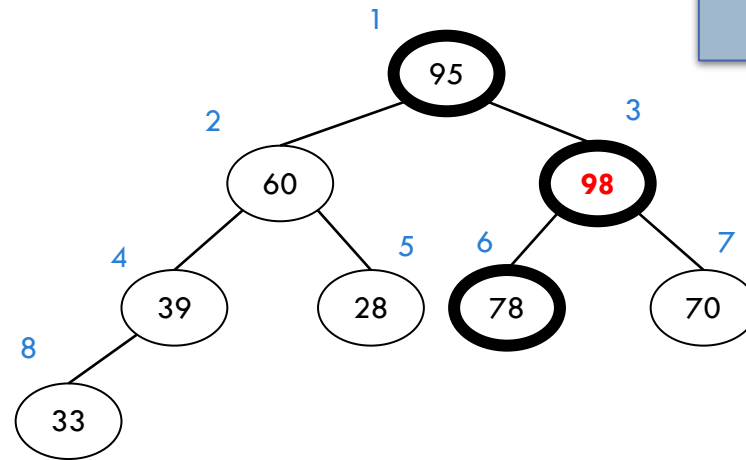


1	2	3	4	5	6	7	8
95	60	78	39	28	98	70	33

EXEMPLO:

AUMENTAR A PRIORIDADE DO NÓ 6 DE 66 PARA 98

“Subir” elemento alterado na árvore, fazendo trocas com o nó pai, até que a árvore volte a ficar correta (todo nó tem prioridade maior que seus filhos)

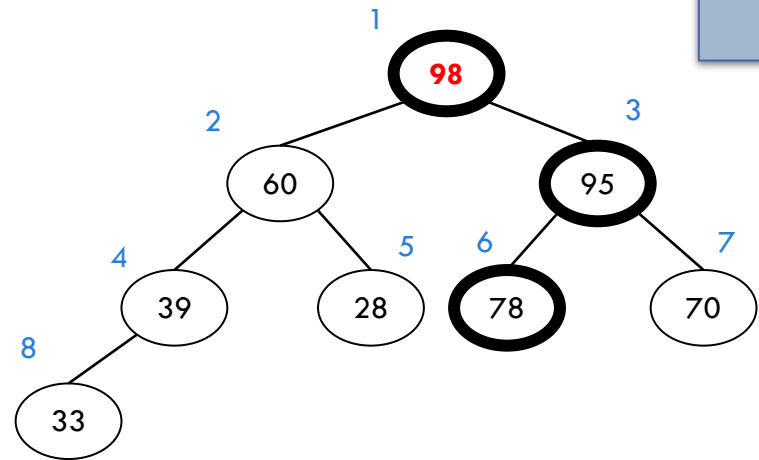


1	2	3	4	5	6	7	8
95	60	98	39	28	78	70	33

EXEMPLO:

AUMENTAR A PRIORIDADE DO NÓ 6 DE 66 PARA 98

“Subir” elemento alterado na árvore, fazendo trocas com o nó pai, até que a árvore volte a ficar correta (todo nó tem prioridade maior que seus filhos)



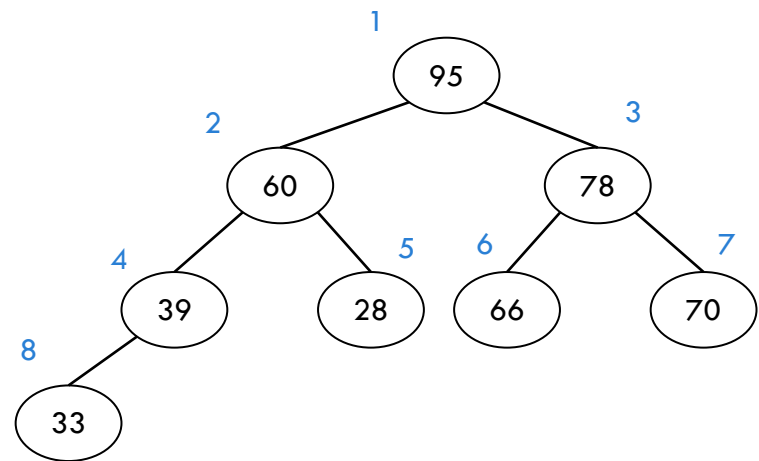
1	2	3	4	5	6	7	8
98	60	95	39	28	78	70	33

FUNÇÃO SUBIR

```
void subir(int *heap, int i) {  
    int j = pai(i);  
    if (j >= 1) {  
        if (heap[i] > heap[j]) {  
            //faz a subida  
            int temp = heap[i];  
            heap[i] = heap[j];  
            heap[j] = temp;  
            subir(heap, j);  
        }  
    }  
}
```

EXEMPLO:

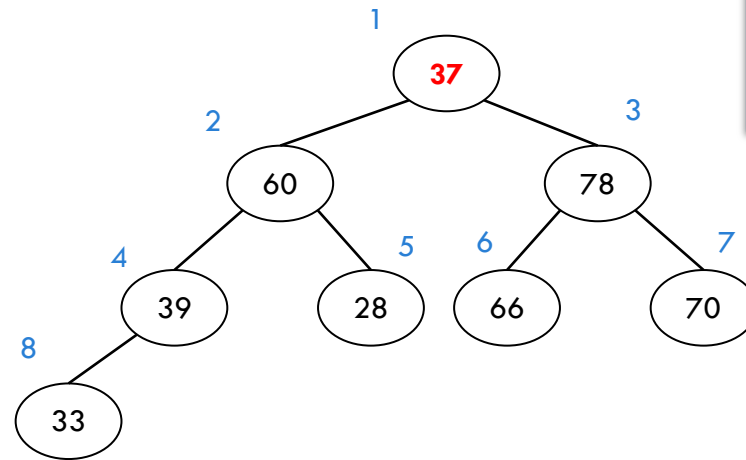
DIMINUIR A PRIORIDADE DO NÓ 1 PARA 37



1	2	3	4	5	6	7	8
95	60	78	39	28	66	70	33

EXEMPLO:

DIMINUIR A PRIORIDADE DO NÓ 1 PARA 37



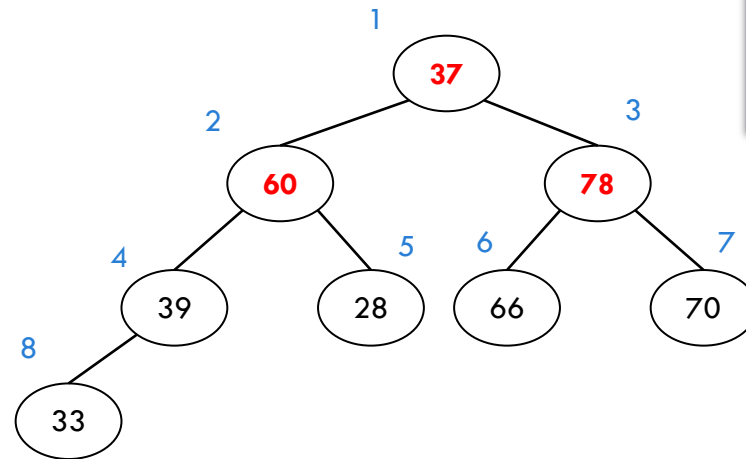
1	2	3	4	5	6	7	8
37	60	78	39	28	66	70	33

“Descer” elemento alterado na árvore, fazendo trocas **com o nó filho de maior prioridade**, até que a árvore volte a ficar correta

EXEMPLO:

DIMINUIR A PRIORIDADE DO NÓ 1 PARA 37

“Descer” elemento alterado na árvore, fazendo trocas **com o nó filho de maior prioridade**, até que a árvore volte a ficar correta



Os dois filhos de i nesse exemplo são 60 e 78:

$$i = 1$$

$$\text{esq}(i) = 2$$

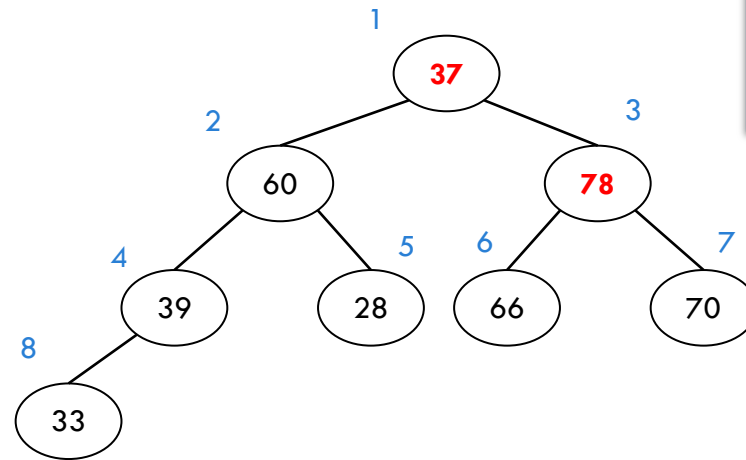
$$\text{dir}(i) = 3$$

1	2	3	4	5	6	7	8
37	60	78	39	28	66	70	33

EXEMPLO:

DIMINUIR A PRIORIDADE DO NÓ 1 PARA 37

“Descer” elemento alterado na árvore, fazendo trocas **com o nó filho de maior prioridade**, até que a árvore volte a ficar correta



Os dois filhos de i nesse exemplo são 60 e 78:

$$i = 1$$

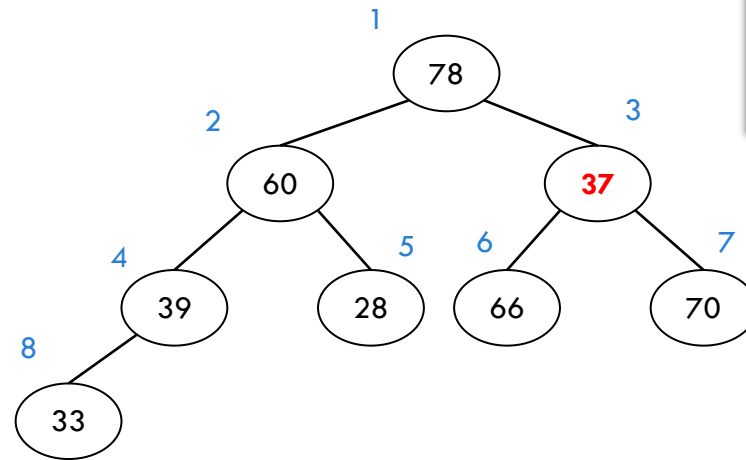
$$\text{esq}(i) = 2$$

$$\text{dir}(i) = 3$$

1	2	3	4	5	6	7	8
37	60	78	39	28	66	70	33

EXEMPLO:

DIMINUIR A PRIORIDADE DO NÓ 1 PARA 37

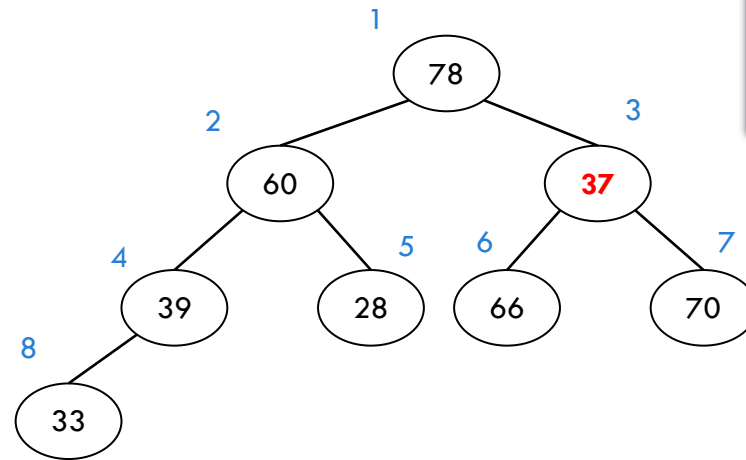


1	2	3	4	5	6	7	8
78	60	37	39	28	66	70	33

“Descer” elemento alterado na árvore, fazendo trocas **com o nó filho de maior prioridade**, até que a árvore volte a ficar correta

EXEMPLO:

DIMINUIR A PRIORIDADE DO NÓ 1 PARA 37

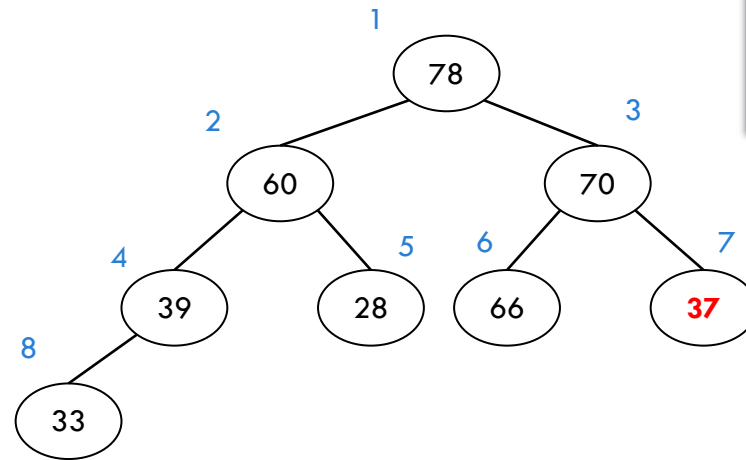


“Descer” elemento alterado na árvore, fazendo trocas **com o nó filho de maior prioridade**, até que a árvore volte a ficar correta

1	2	3	4	5	6	7	8
78	60	37	39	28	66	70	33

EXEMPLO:

DIMINUIR A PRIORIDADE DO NÓ 1 PARA 37



“Descer” elemento alterado na árvore, fazendo trocas **com o nó filho de maior prioridade**, até que a árvore volte a ficar correta

1	2	3	4	5	6	7	8
78	60	70	39	28	66	37	33

FUNÇÃO DESCER

```
void descer(int *heap, int i, int n){  
    //descobre quem é o maior filho de i  
    int e = esq(i);  
    int d = dir(i);  
    int maior = i;  
    if (e<=n && heap[e] > heap[i]) {  
        maior=e;  
    }  
    if (d<=n && heap[d] > heap[maior]) {  
        maior=d;  
    }  
    if (maior != i){  
        //faz a descida trocando com o maior filho  
        int temp=heap[i];  
        heap[i] = heap[maior];  
        heap[maior]=temp;  
        descer(heap, maior, n);  
    }  
}
```

INSERÇÃO

Tabela com n elementos

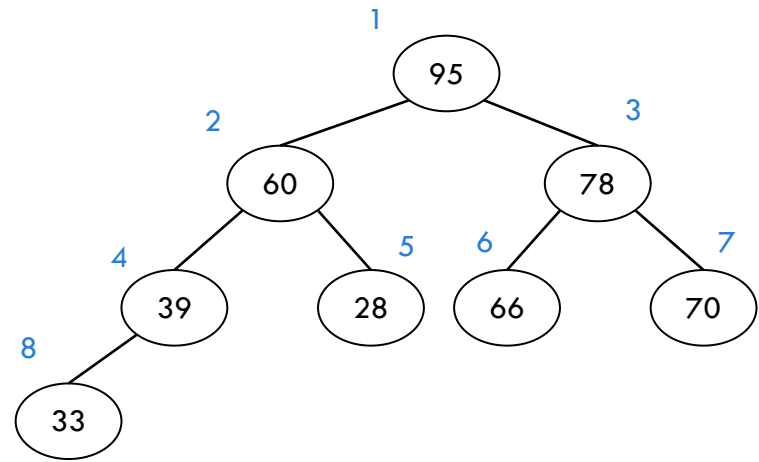
Inserir novo elemento na posição $n+1$ da tabela

Assumir que esse elemento já existia e teve sua prioridade aumentada

Executar algoritmo de **subida** na árvore para corrigir a prioridade e colocar o novo elemento na posição correta

EXEMPLO:

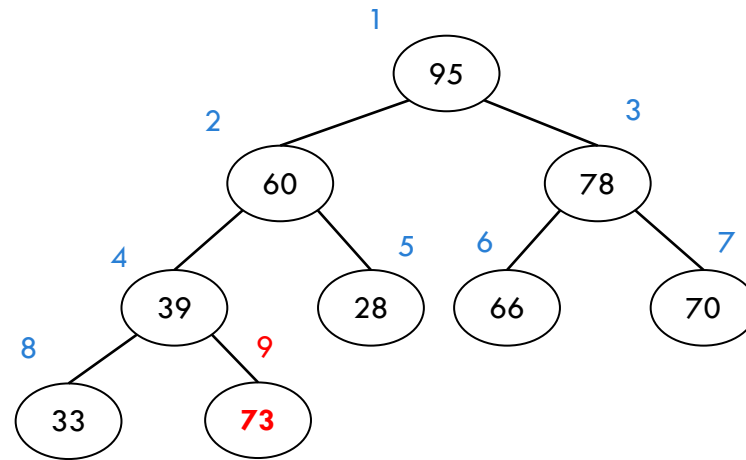
INSERIR ELEMENTO 73



1	2	3	4	5	6	7	8
95	60	78	39	28	66	70	33

EXEMPLO:

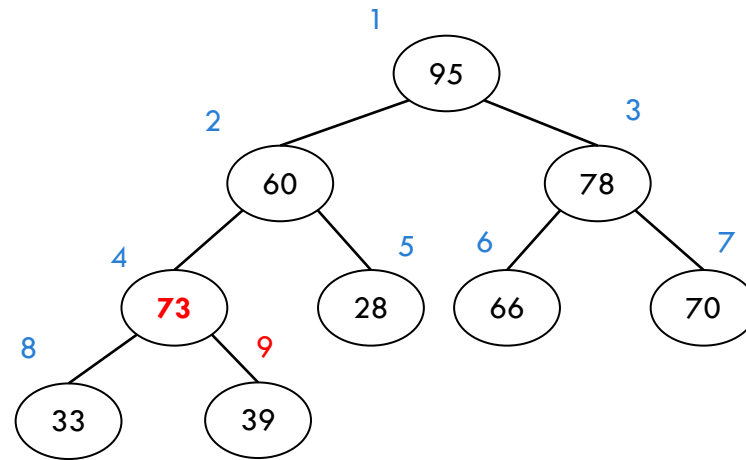
INSERIR ELEMENTO 73



1	2	3	4	5	6	7	8	9
95	60	78	39	28	66	70	33	73

EXEMPLO:

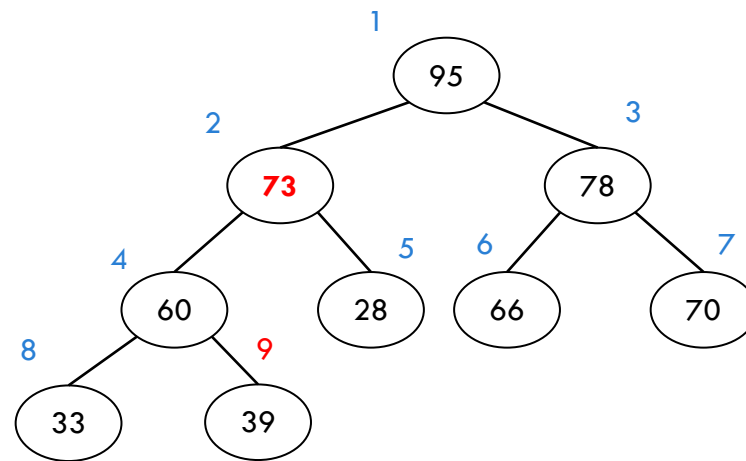
INSERIR ELEMENTO 73



1	2	3	4	5	6	7	8	9
95	60	78	73	28	66	70	33	39

EXEMPLO:

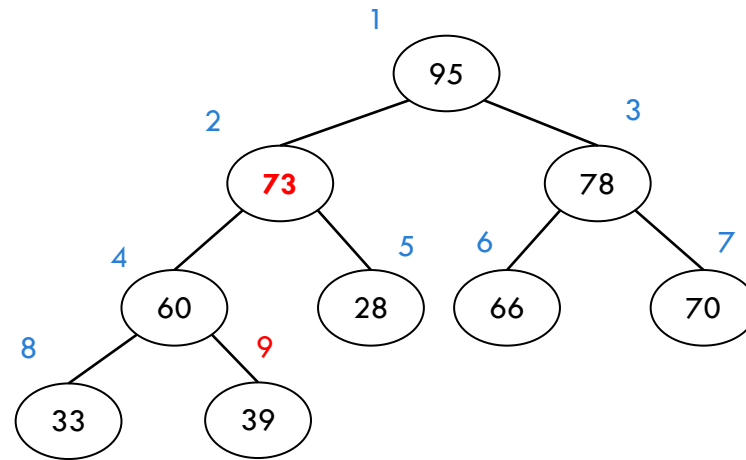
INSERIR ELEMENTO 73



1	2	3	4	5	6	7	8	9
95	73	78	60	28	66	70	33	39

EXEMPLO:

INSERIR ELEMENTO 73



1	2	3	4	5	6	7	8	9
95	73	78	60	28	66	70	33	39

INSERÇÃO

```
int insere(int *heap, int novo, int n) {  
    //aumenta o tamanho do vetor  
    heap = (int *) realloc(heap, sizeof(int) * (n + 2));  
    n = n + 1;  
    heap[n] = novo;  
  
    subir(heap, n);  
    //retorna o novo valor de n  
    return n;  
}
```


REMOÇÃO DO ELEMENTO MAIS PRIORITÁRIO

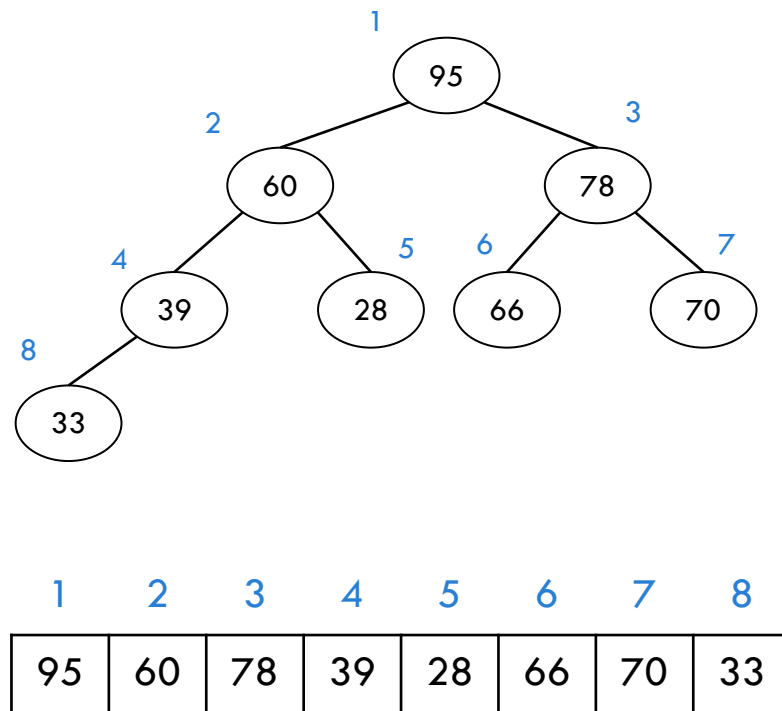
Remover o **primeiro** elemento da tabela

Preencher o espaço vazio deixado por ele com o **último elemento da tabela**

Executar o algoritmo de **descida** na árvore para corrigir a prioridade desse elemento

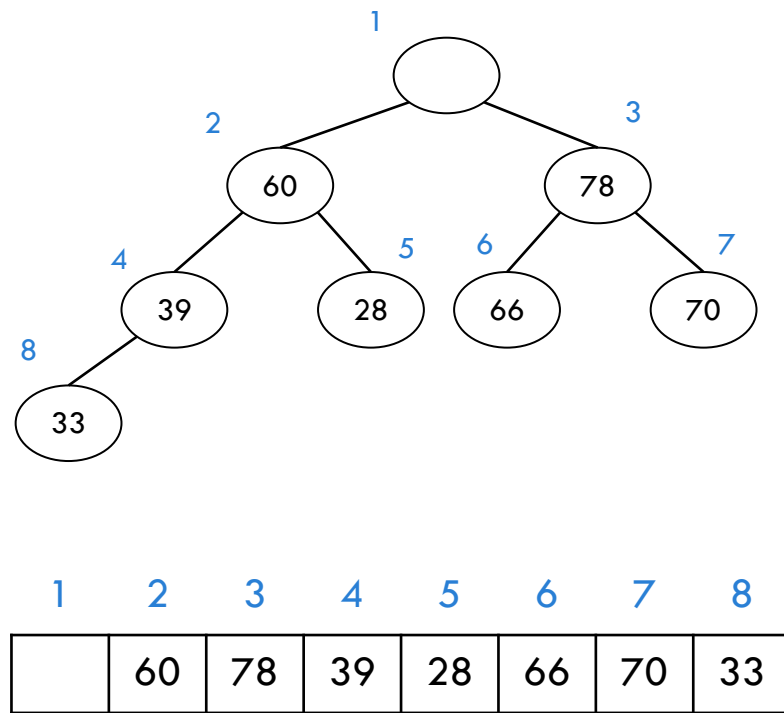
EXEMPLO

REMOVER ELEMENTO MAIS PRIORITÁRIO



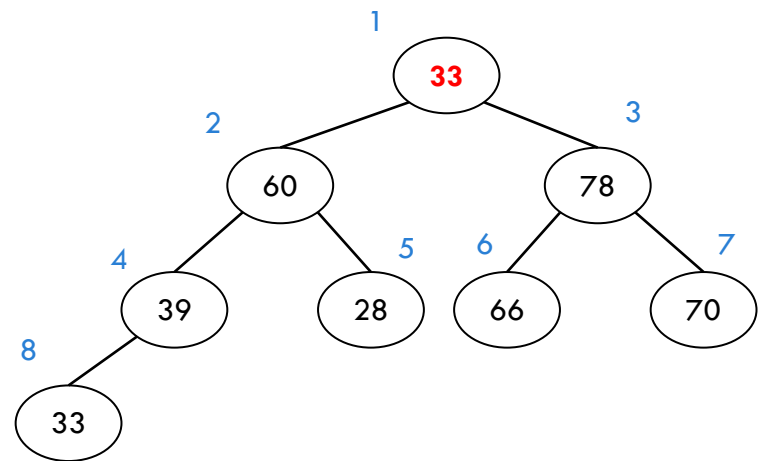
EXEMPLO

REMOVER ELEMENTO MAIS PRIORITÁRIO



EXEMPLO

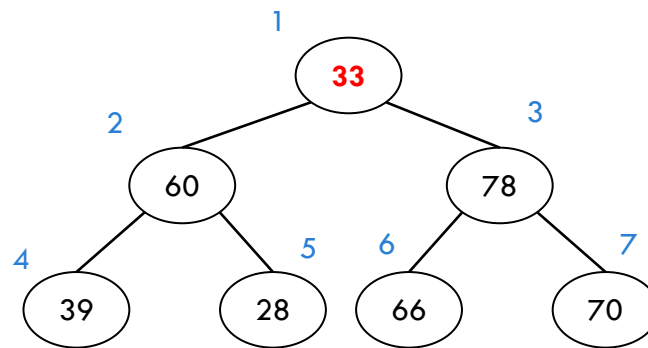
REMOVER ELEMENTO MAIS PRIORITÁRIO



1	2	3	4	5	6	7	8
33	60	78	39	28	66	70	

EXEMPLO

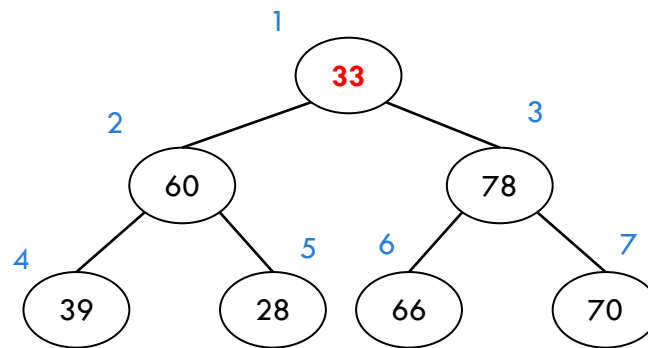
REMOVER ELEMENTO MAIS PRIORITÁRIO



1	2	3	4	5	6	7
33	60	78	39	28	66	70

EXEMPLO

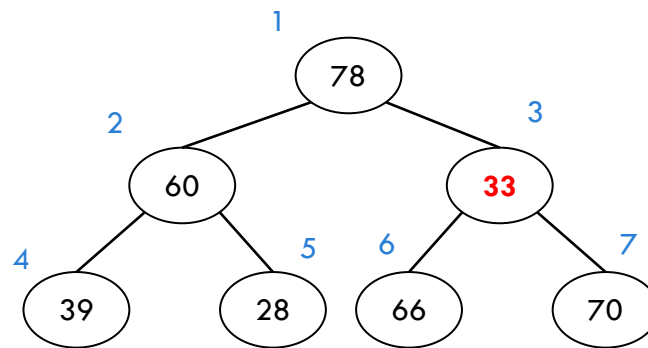
REMOVER ELEMENTO MAIS PRIORITÁRIO



1	2	3	4	5	6	7
33	60	78	39	28	66	70

EXEMPLO

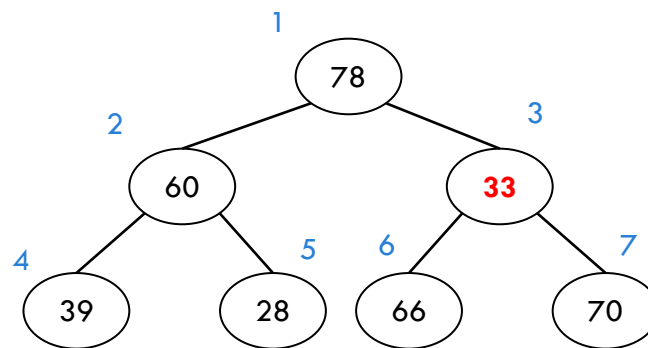
REMOVER ELEMENTO MAIS PRIORITÁRIO



1	2	3	4	5	6	7
78	60	33	39	28	66	70

EXEMPLO

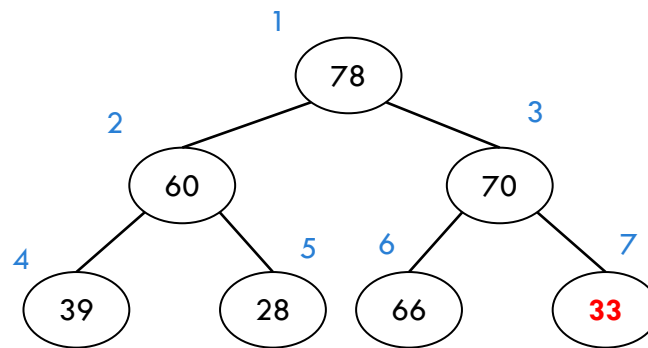
REMOVER ELEMENTO MAIS PRIORITÁRIO



1	2	3	4	5	6	7
78	60	33	39	28	66	70

EXEMPLO

REMOVER ELEMENTO MAIS PRIORITÁRIO



1	2	3	4	5	6	7
78	60	70	39	28	66	33

EXCLUSÃO

```
int exclui(int *heap, int n) {
    heap[1] = heap[n];
    n = n - 1;
    //diminui o tamanho do vetor
    heap = (int *) realloc(heap, sizeof(int) * (n + 1));

    descer(heap, 1, n);
    //retorna o novo valor de n
    return n;
}
```

CONSTRUÇÃO DE LISTA DE PRIORIDADES

Dada uma lista L de elementos para a qual se deseja construir uma heap H , há duas alternativas

- 1) Considerar uma heap vazia e ir inserindo os elementos de L um a um em H
- 2) Considerar que a lista L é uma heap, e corrigir as prioridades.
 - Assumir que as prioridades das folhas estão corretas (pois eles não têm filhos, então satisfazem à propriedade de terem prioridade maior que seus filhos)
 - Acertar as prioridades dos nós internos realizando descidas quando necessário

EXEMPLO

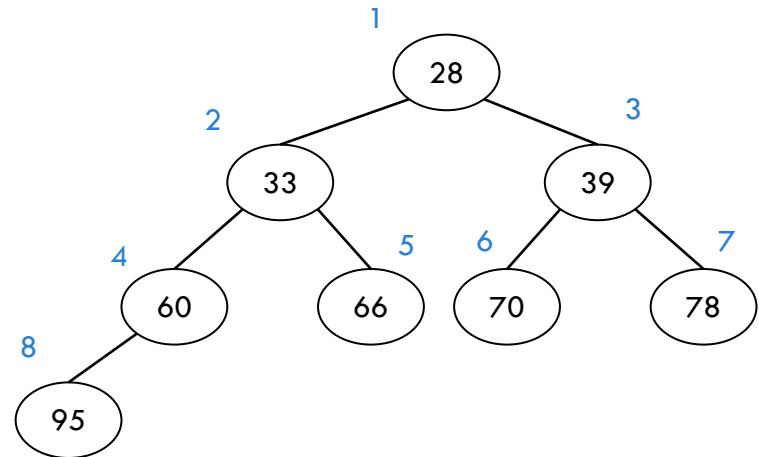
CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95

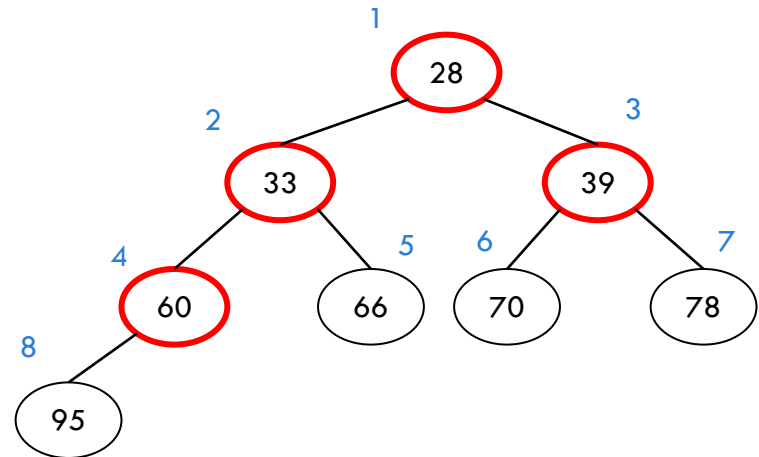


1	2	3	4	5	6	7	8
28	33	39	60	66	70	78	95

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



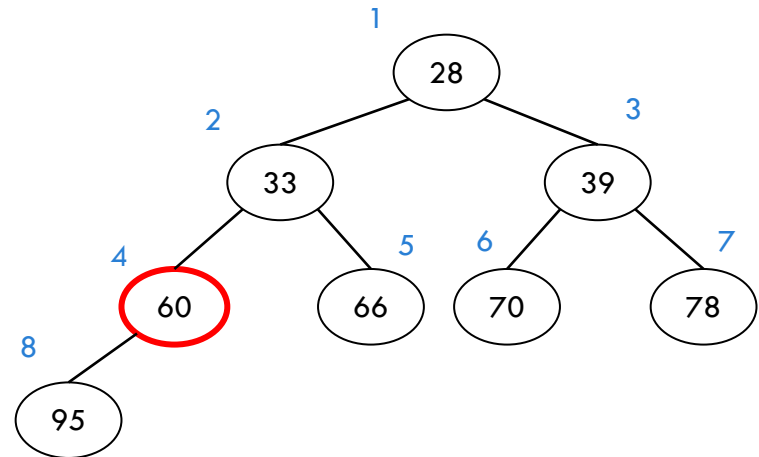
1	2	3	4	5	6	7	8
28	33	39	60	66	70	78	95

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando descidas quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



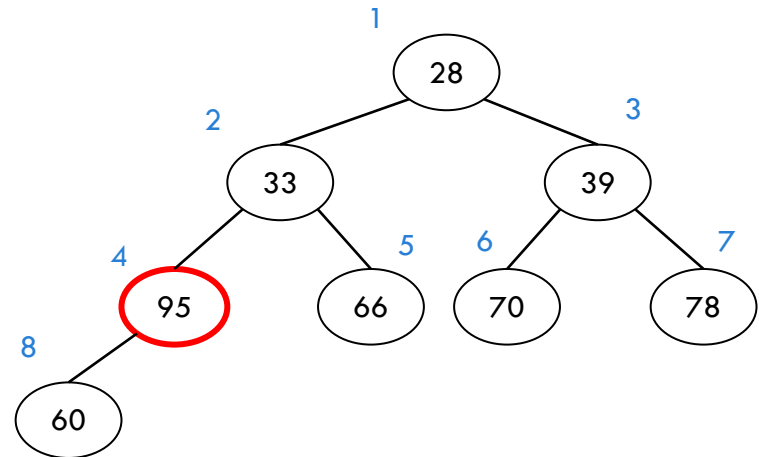
1	2	3	4	5	6	7	8
28	33	39	60	66	70	78	95

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando descidas quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



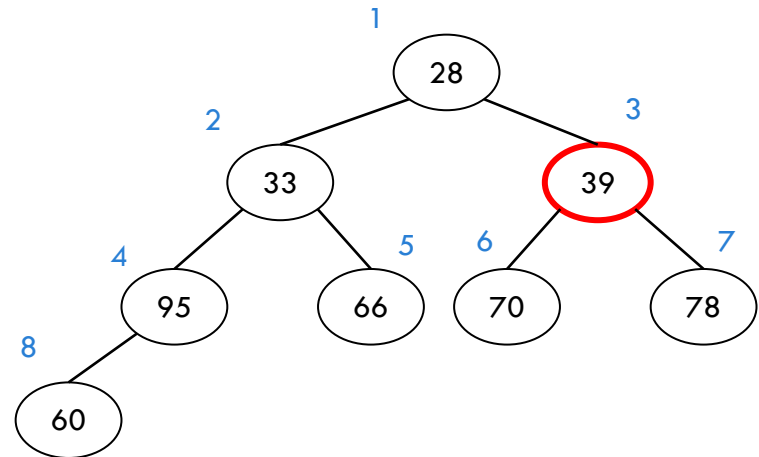
1	2	3	4	5	6	7	8
28	33	39	95	66	70	78	60

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



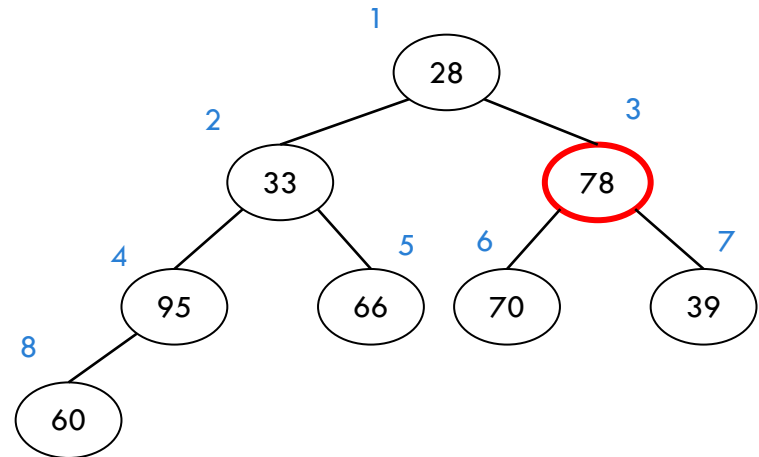
1	2	3	4	5	6	7	8
28	33	39	95	66	70	78	60

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



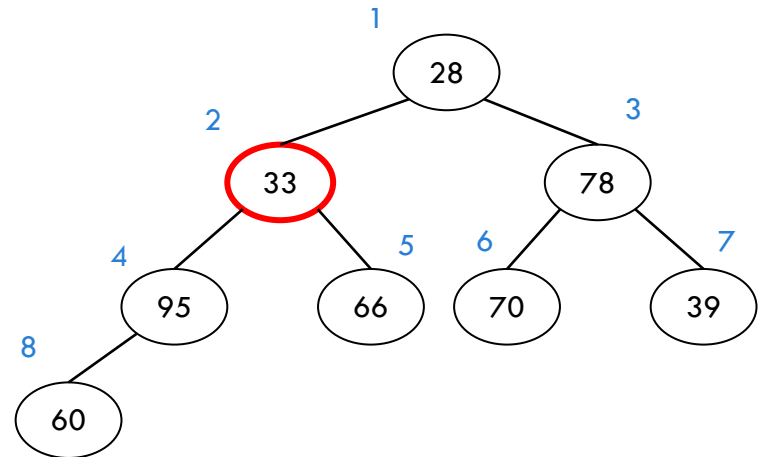
1	2	3	4	5	6	7	8
28	33	78	95	66	70	39	60

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



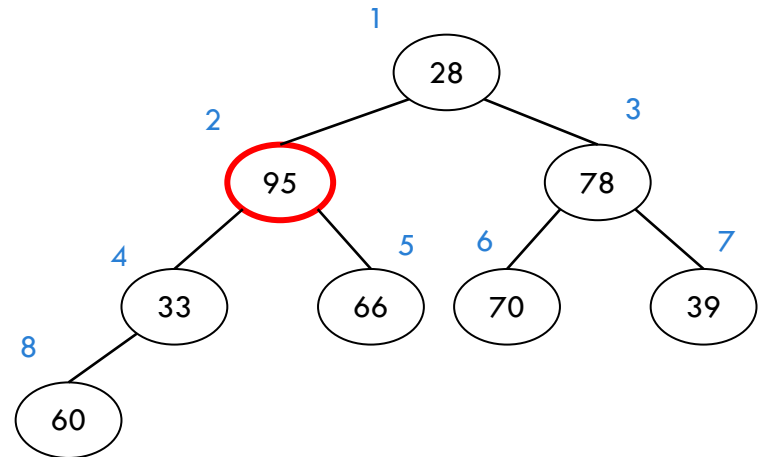
1	2	3	4	5	6	7	8
28	33	78	95	66	70	39	60

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



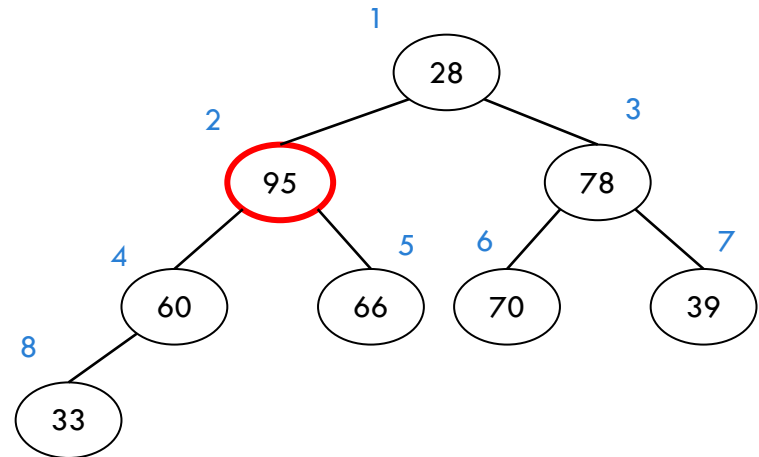
1	2	3	4	5	6	7	8
28	95	78	33	66	70	39	60

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



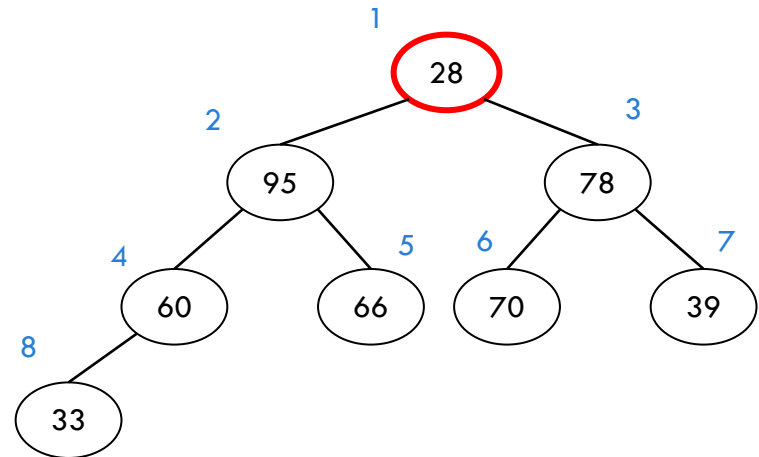
1	2	3	4	5	6	7	8
28	95	78	60	66	70	39	33

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



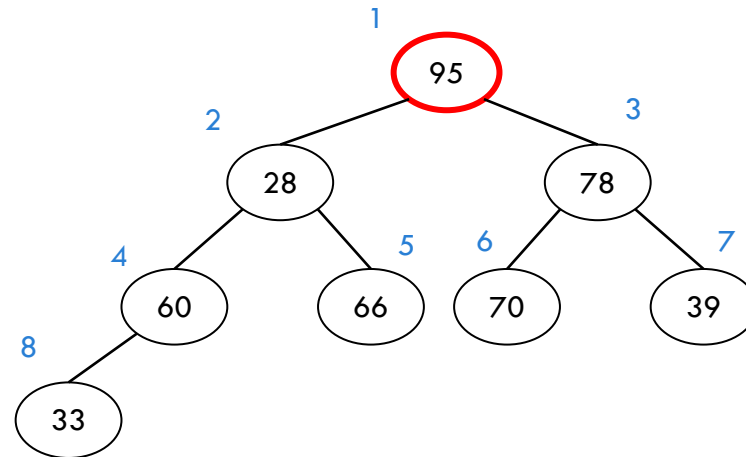
1	2	3	4	5	6	7	8
28	95	78	60	66	70	39	33

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



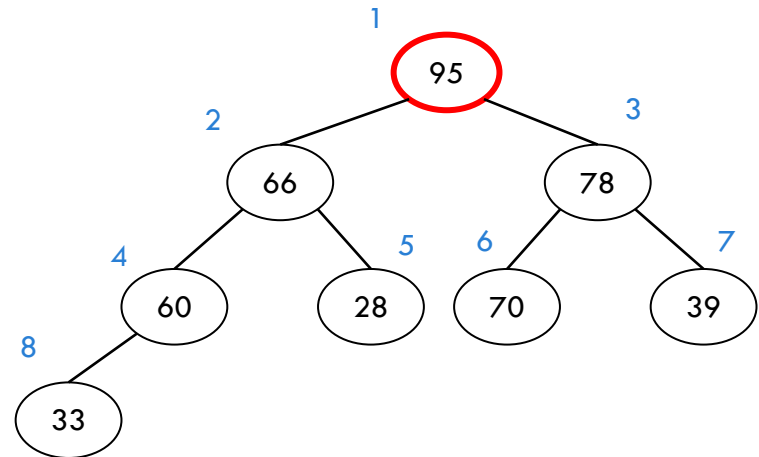
1	2	3	4	5	6	7	8
95	28	78	60	66	70	39	33

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

EXEMPLO

CONSTRUÇÃO DE HEAP

Construir uma Heap a partir da lista 28, 33, 39, 60, 66, 70, 78, 95



1	2	3	4	5	6	7	8
95	66	78	60	28	70	39	33

Acertar prioridade do elemento $n/2$ até o elemento 1, nessa ordem, realizando **descidas** quando necessário

CONSTRUÇÃO DA HEAP

```
void constroi_heap_maximo(int *heap, int n) {  
    int i;  
    int j = (n/2);  
    for (i=j; i>=1; i--)  
        descer(heap, i, n);  
}
```

COMPLEXIDADE

Seleção: imediata, pois elemento de maior prioridade é o primeiro da tabela
– $O(1)$

Inserção, Alteração e Remoção: $O(\log n)$

Construção: pode ser feita em $O(n)$ (melhor que no cenário anterior, que exigia ordenação total)

FUNCIONALIDADE ADICIONAL: ORDENAÇÃO

A partir da heap, é possível ordenar os dados, fazendo trocas:

- O maior elemento (raiz) é trocado com o último elemento do vetor
- Esse elemento então já está na posição correta (ordenação crescente)
- Considerar que vetor tem tamanho $n-1$ e descer a raiz para que o heap fique consistente
- Repetir esses passos $n-1$ vezes

A Complexidade do heap sort é $O(n \log n)$

HEAP SORT

```
void heap_sort(int *heap, int n){
    int i;
    int j=n;
    constroi_heap_maximo(heap, n);
    for(i=n; i>1; i--){
        //troca raiz com o ultimo elemento (posicao j)
        int temp=heap[i];
        heap[i]=heap[1];
        heap[1]=temp;
        //diminui o tamanho do vetor a ser considerado na heap
        j--;
        //desce com a raiz nessa nova heap de tamanho j-1
        descer(heap, 1, j);
    }
}
```

EXERCÍCIOS

1. Verificar se essas sequências correspondem ou não a um heap

(a) 33 32 28 31 26 29 25 30 27

(b) 36 32 28 31 29 26 25 30 27

(c) 33 32 28 30 29 26 25 31 27

(d) 35 31 28 33 29 26 25 30 27

EXERCÍCIOS

2. Seja o heap especificado a seguir: 92 85 90 47 71 34 20 40 46. Sobre esse heap, realizar as seguintes operações:

(a) Inserir os elementos 98, 75, 43

(b) Remover o elemento de maior prioridade (sobre o heap original)

(c) Remover o elemento de maior prioridade (sobre o heap resultante do exercício (b))

(d) Alterar a prioridade do 5º. nó de 71 para 93 (sobre o heap original)

(e) Alterar a prioridade do 5º. nó de 71 para 19 (sobre o heap original)

IMPLEMENTAÇÃO DE HEAP EM DISCO

Ver implementação no site da disciplina

REFERÊNCIA

Szwarcfiter, J.; Markezon, L. Estruturas de Dados e seus Algoritmos, 3a. ed. LTC. Cap. 6