Filas de Prioridade (Heaps)

Instituto de Computação - UNICAMP

Concurso Público de Provas e Títulos
Cargo de Professor Doutor MS-3.1

Ulisses Martins Dias

Roteiro

Roteiro

Utilização

- Métodos iterativos baseados na seleção sistemática de itens de maior (menor) valor.

Utilização

- Métodos iterativos baseados na seleção sistemática de itens de maior (menor) valor.
- Sistemas operacionais usam filas de prioridade em que as chaves representam o tempo em que eventos devem ocorrer.

Utilização

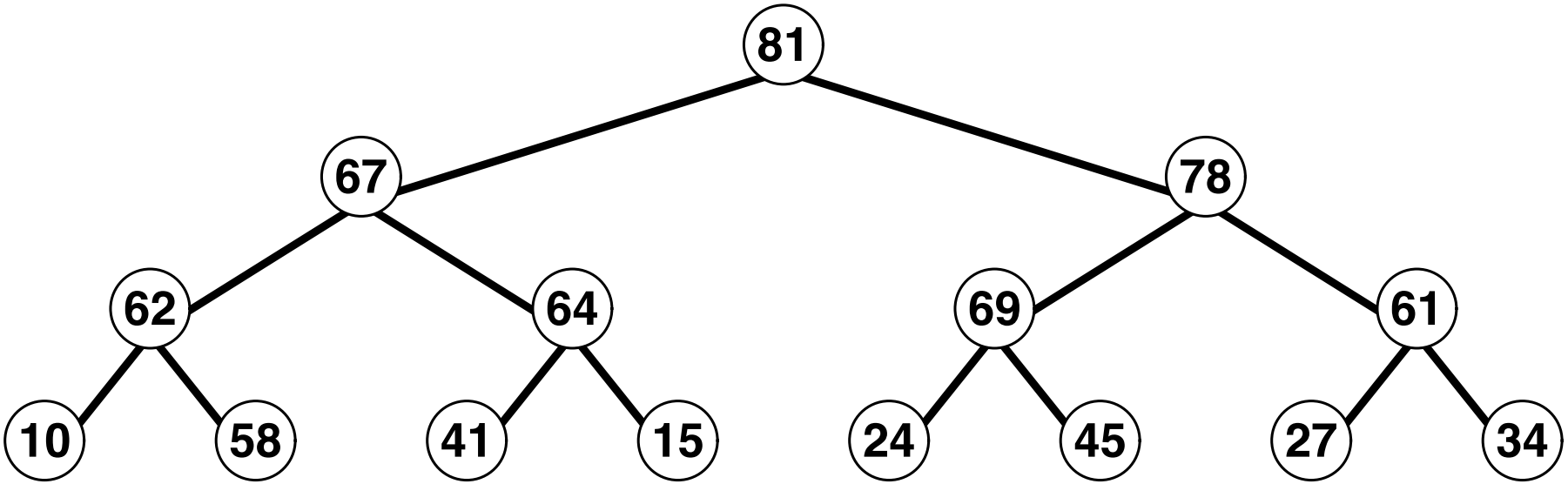
- Métodos iterativos baseados na seleção sistemática de itens de maior (menor) valor.
- Sistemas operacionais usam filas de prioridade em que as chaves representam o tempo em que eventos devem ocorrer.
- Sistemas de gerenciamento de memória substituem páginas menos utilizadas por uma nova página.

Roteiro

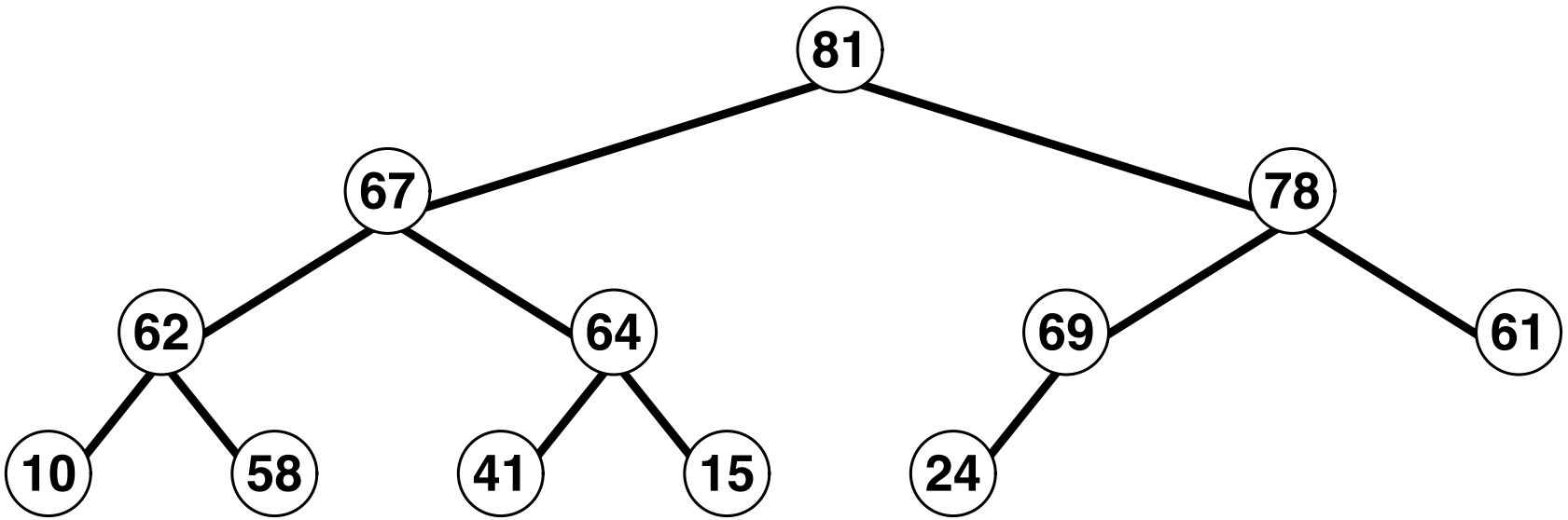
Definição e Propriedades

- A árvore é Completa ou quase-completa.

Árvore Completa ($h = \lfloor \lg n \rfloor$)



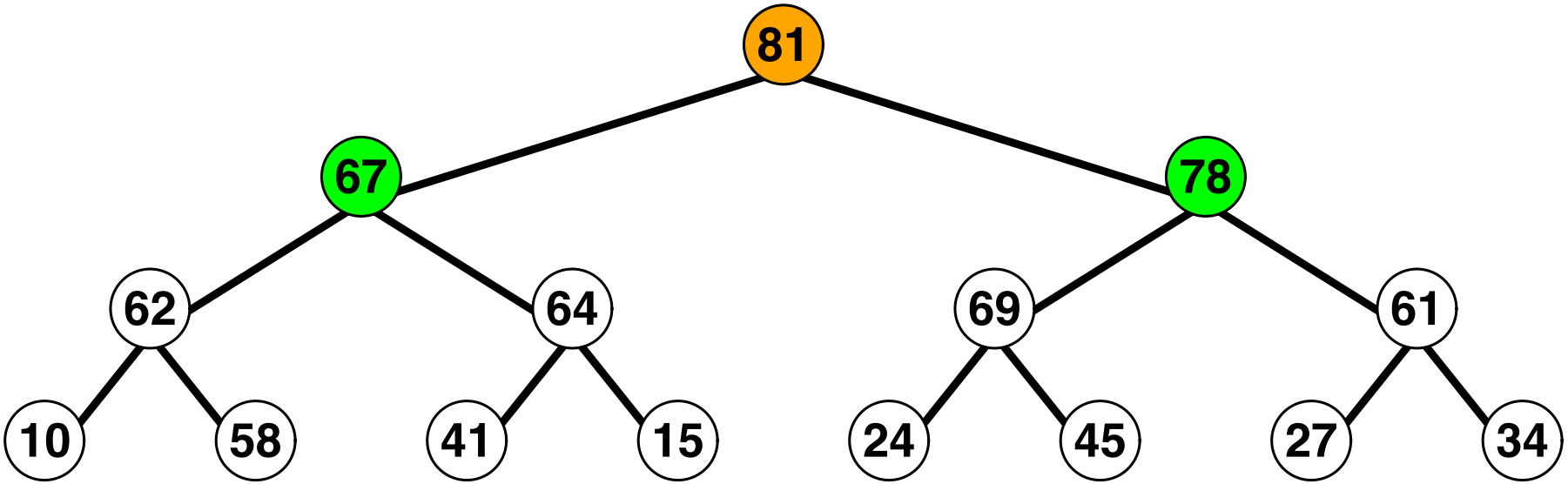
Árvore Quase-Completa ($h = \lfloor \lg n \rfloor$)



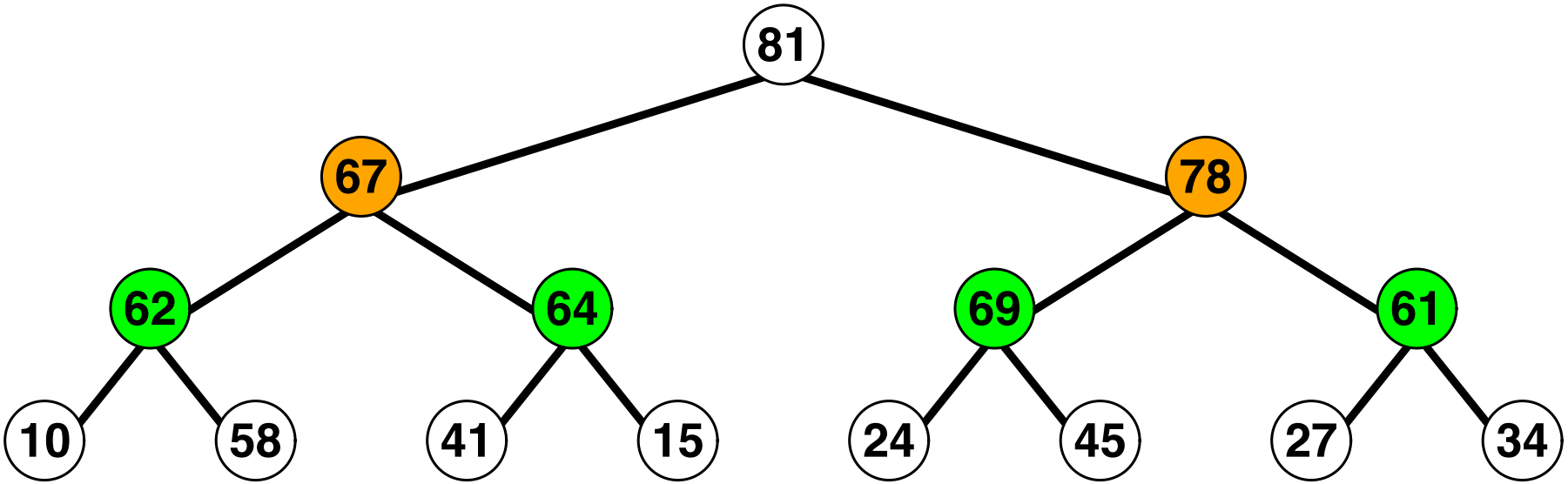
Definição e Propriedades

- A árvore é Completa ou quase-completa.
- Em cada nó da árvore, o valor da chave é maior ou igual aos valores das chaves dos filhos.

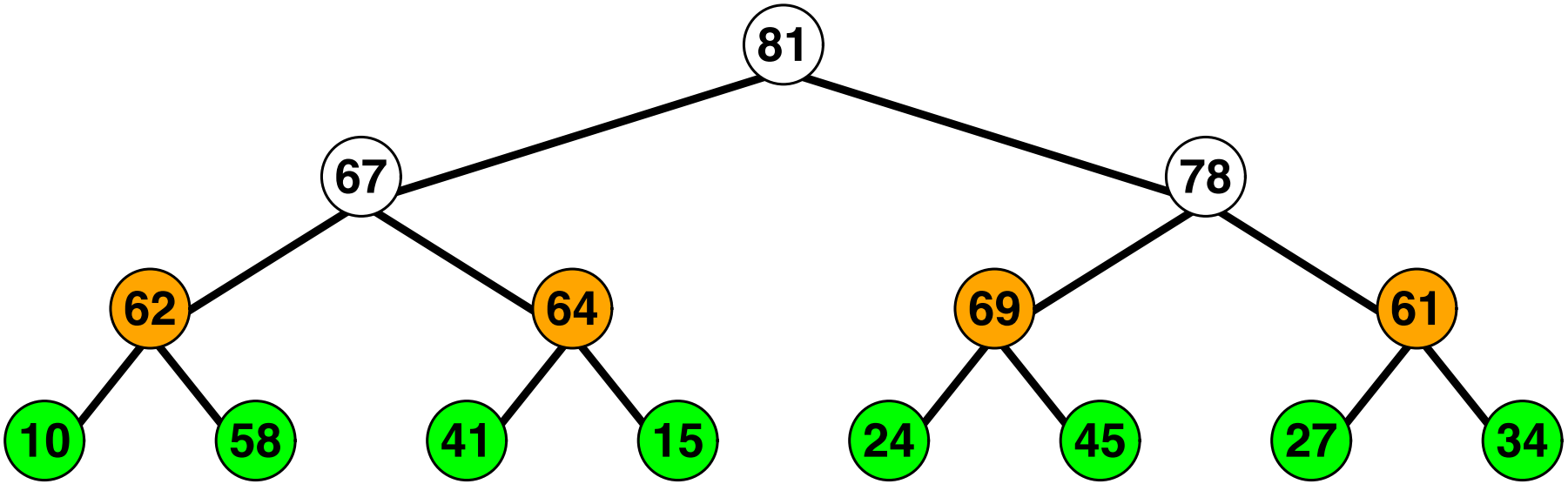
$chave(x) \geq chave(esq(x)), chave(dir(x))$



$chave(x) \geq chave(esq(x)), chave(dir(x))$



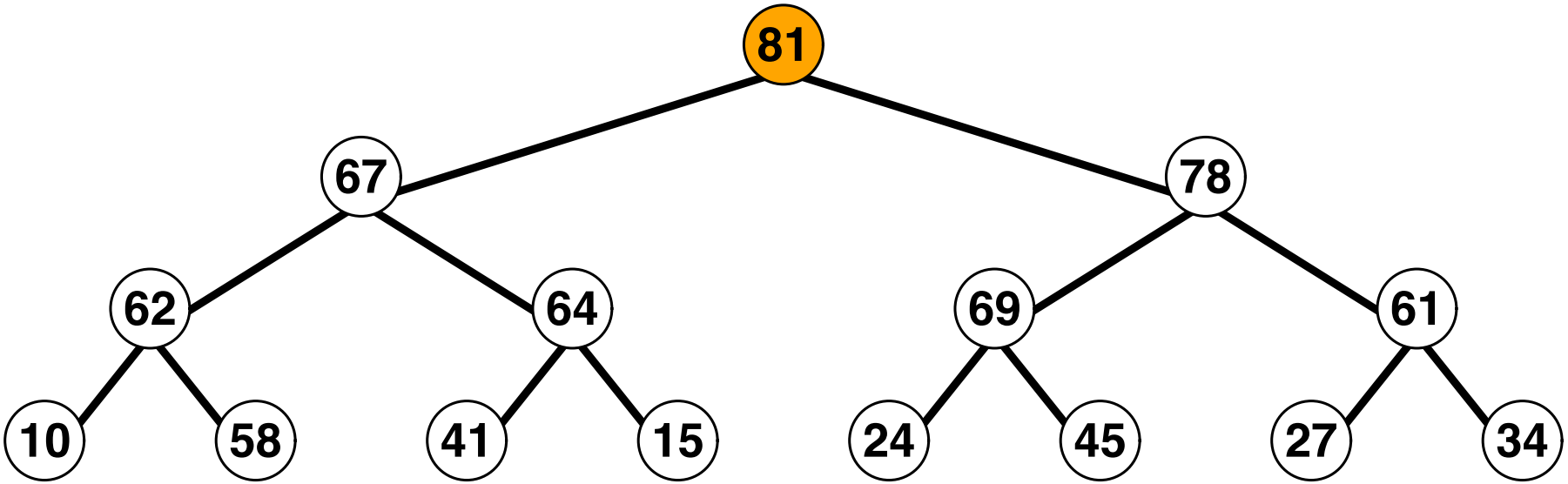
$chave(x) \geq chave(esq(x)), chave(dir(x))$



Definição e Propriedades

- A árvore é Completa ou quase-completa.
- Em cada nó da árvore, o valor da chave é maior ou igual aos valores das chaves dos filhos.
- O elemento de maior prioridade pode ser encontrado em tempo constante.

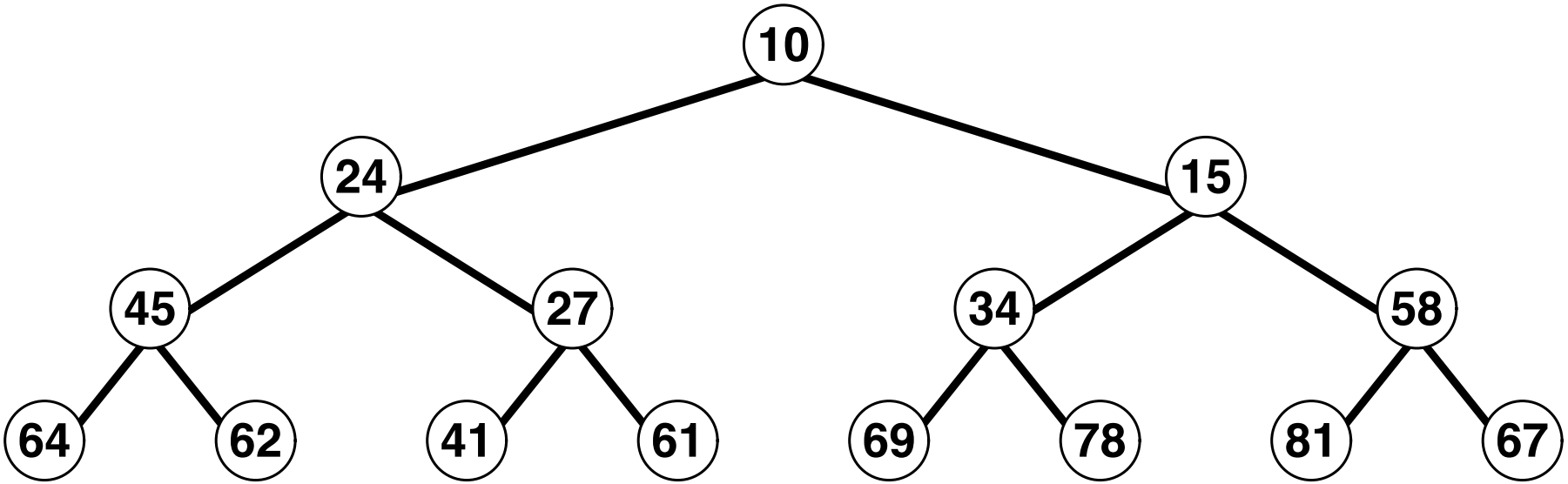
Elemento de Maior Prioridade



Definição e Propriedades

- A árvore é Completa ou quase-completa.
- Em cada nó da árvore, o valor da chave é maior ou igual aos valores das chaves dos filhos.
- O elemento de maior prioridade pode ser encontrado em tempo constante.
- Existe uma versão de fila de prioridade em que o valor da chave de cada elemento é menor ou igual aos valores das chaves dos filhos.

Fila de Prioridade Mínima



Definição e Propriedades

- A árvore é Completa ou quase-completa.
- Em cada nó da árvore, o valor da chave é maior ou igual aos valores das chaves dos filhos.
- O elemento de maior prioridade pode ser encontrado em tempo constante.
- Existe uma versão de fila de prioridade em que o valor da chave de cada elemento é menor ou igual aos valores das chaves dos filhos.
- A inserção de um elemento pode ser feita em tempo proporcional à altura da árvore ($O(\log n)$).

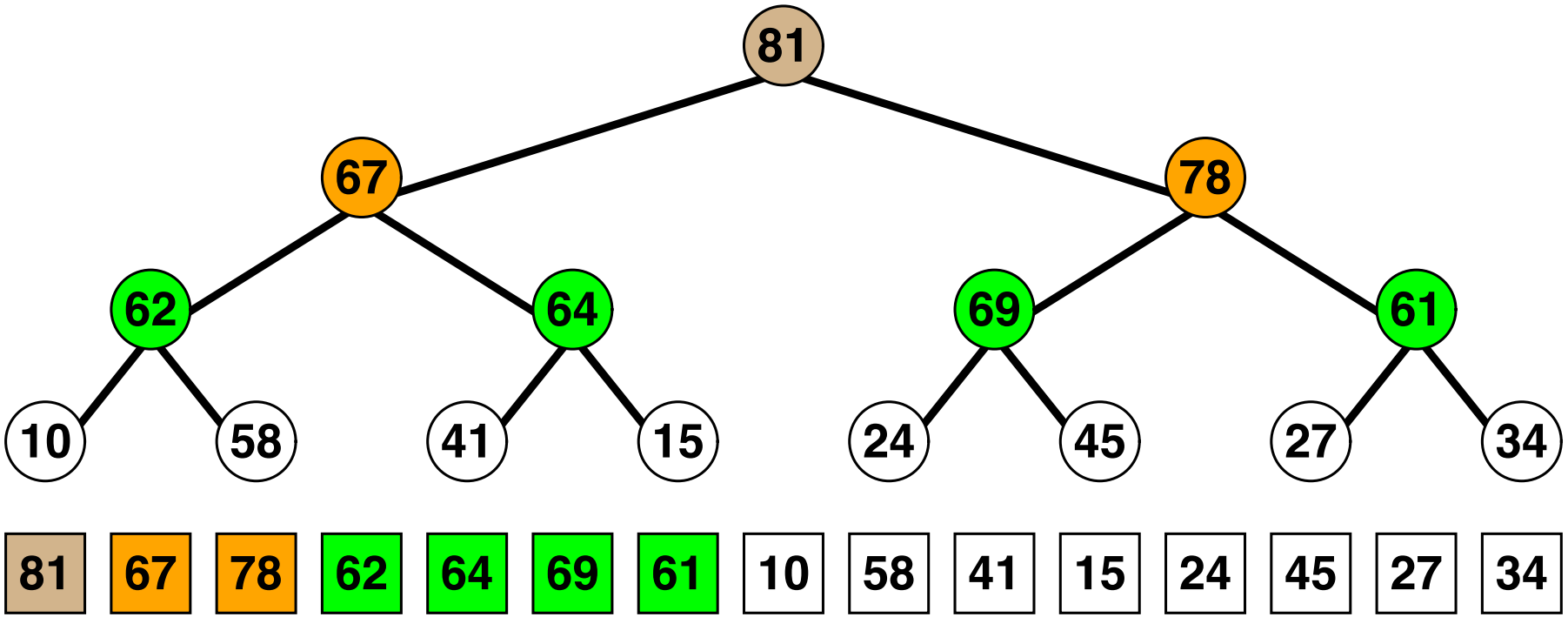
Definição e Propriedades

- A árvore é Completa ou quase-completa.
- Em cada nó da árvore, o valor da chave é maior ou igual aos valores das chaves dos filhos.
- O elemento de maior prioridade pode ser encontrado em tempo constante.
- Existe uma versão de fila de prioridade em que o valor da chave de cada elemento é menor ou igual aos valores das chaves dos filhos.
- A inserção de um elemento pode ser feita em tempo proporcional à altura da árvore ($O(\log n)$).
- A remoção do elemento máximo pode ser feita em tempo proporcional à altura da árvore ($O(\log n)$)

Definição e Propriedades

- A árvore é Completa ou quase-completa.
- Em cada nó da árvore, o valor da chave é maior ou igual aos valores das chaves dos filhos.
- O elemento de maior prioridade pode ser encontrado em tempo constante.
- Existe uma versão de fila de prioridade em que o valor da chave de cada elemento é menor ou igual aos valores das chaves dos filhos.
- A inserção de um elemento pode ser feita em tempo proporcional à altura da árvore ($O(\log n)$).
- A remoção do elemento máximo pode ser feita em tempo proporcional à altura da árvore ($O(\log n)$)
- Filas de prioridade podem ser implementadas de maneira sequencial.

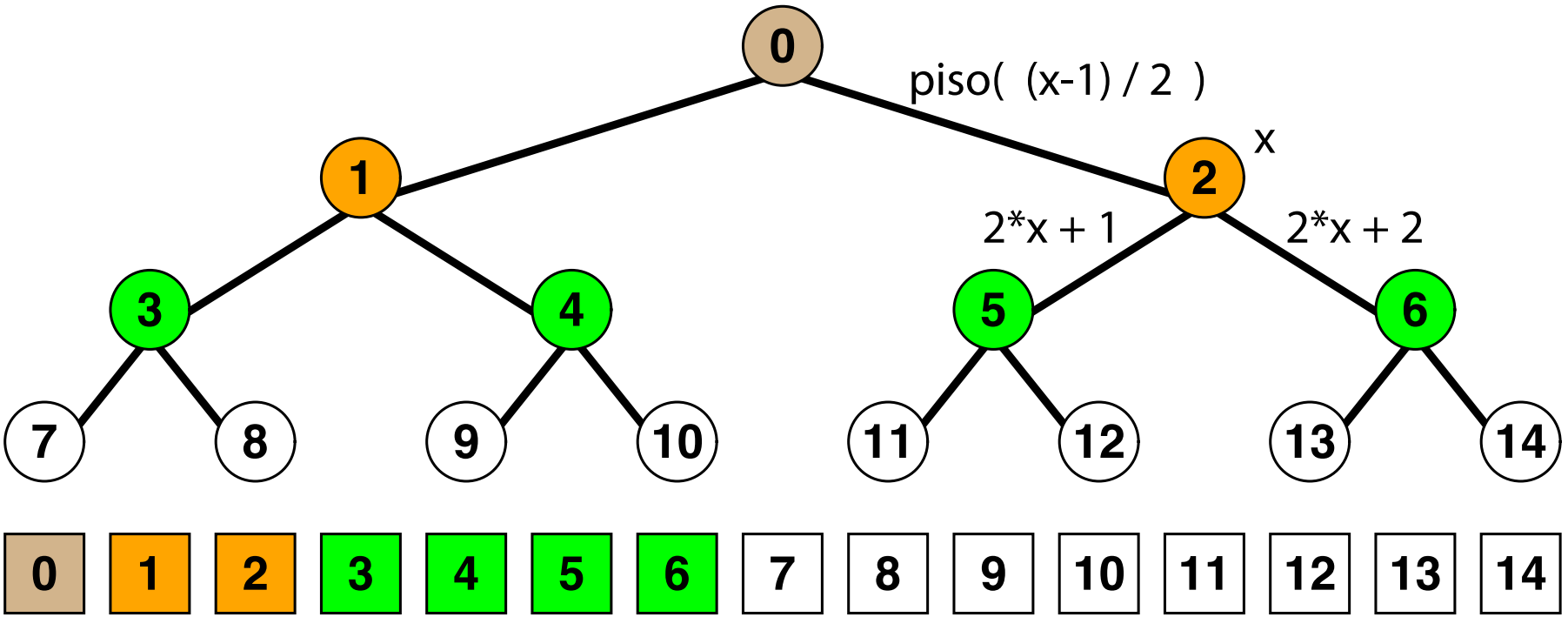
Implementação Sequencial



Definição e Propriedades

- A árvore é Completa ou quase-completa.
- Em cada nó da árvore, o valor da chave é maior ou igual aos valores das chaves dos filhos.
- O elemento de maior prioridade pode ser encontrado em tempo constante.
- Existe uma versão de fila de prioridade em que o valor da chave de cada elemento é menor ou igual aos valores das chaves dos filhos.
- A inserção de um elemento pode ser feita em tempo proporcional à altura da árvore ($O(\log n)$).
- A remoção do elemento máximo pode ser feita em tempo proporcional à altura da árvore ($O(\log n)$)
- Filas de prioridade podem ser implementadas de maneira sequencial.
  - $pai(x) = \lfloor \frac{x-1}{2} \rfloor$; $esq(x) = 2 * x + 1$; $dir(x) = 2 * x + 2$.

Implementação Sequencial - Índices



Roteiro

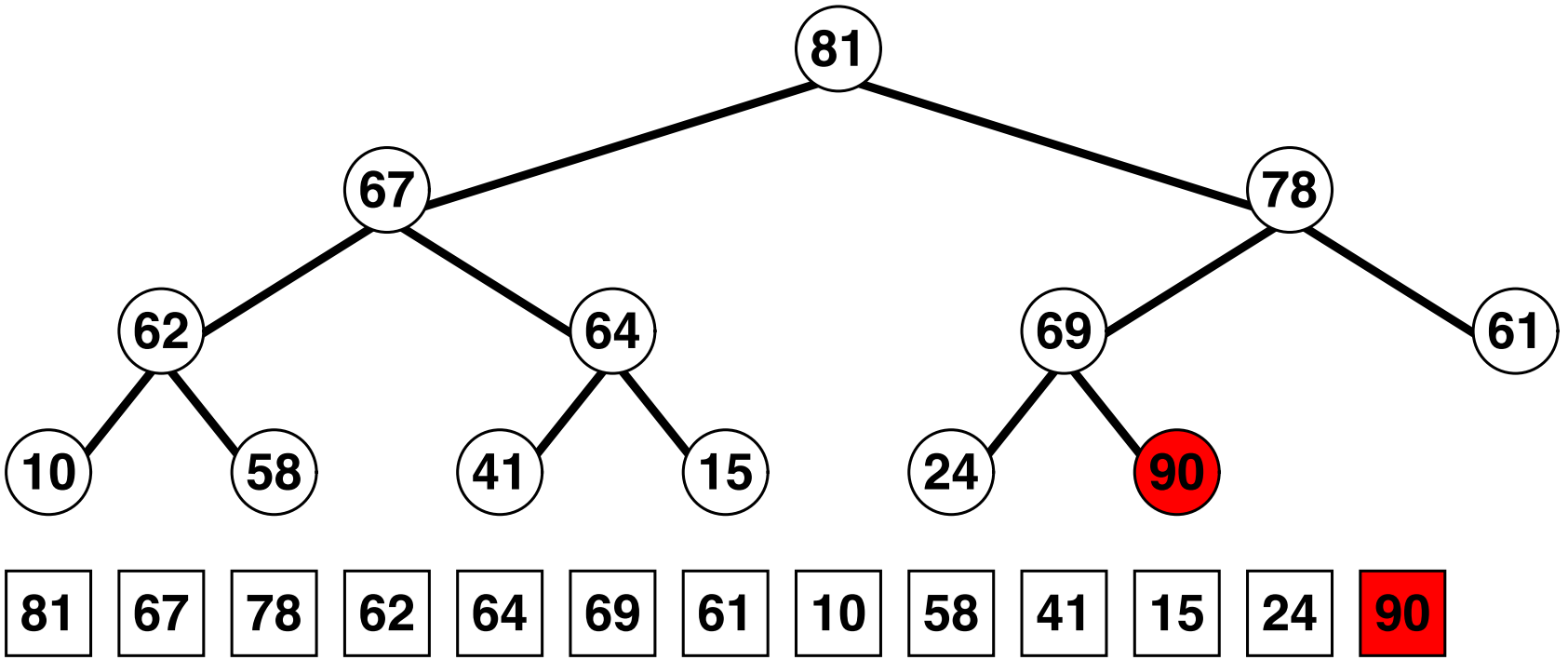
Definição e Propriedades

- Vamos assumir que temos uma estrutura de dados em que apenas um dos nós viola as propriedades de Fila de Prioridade.

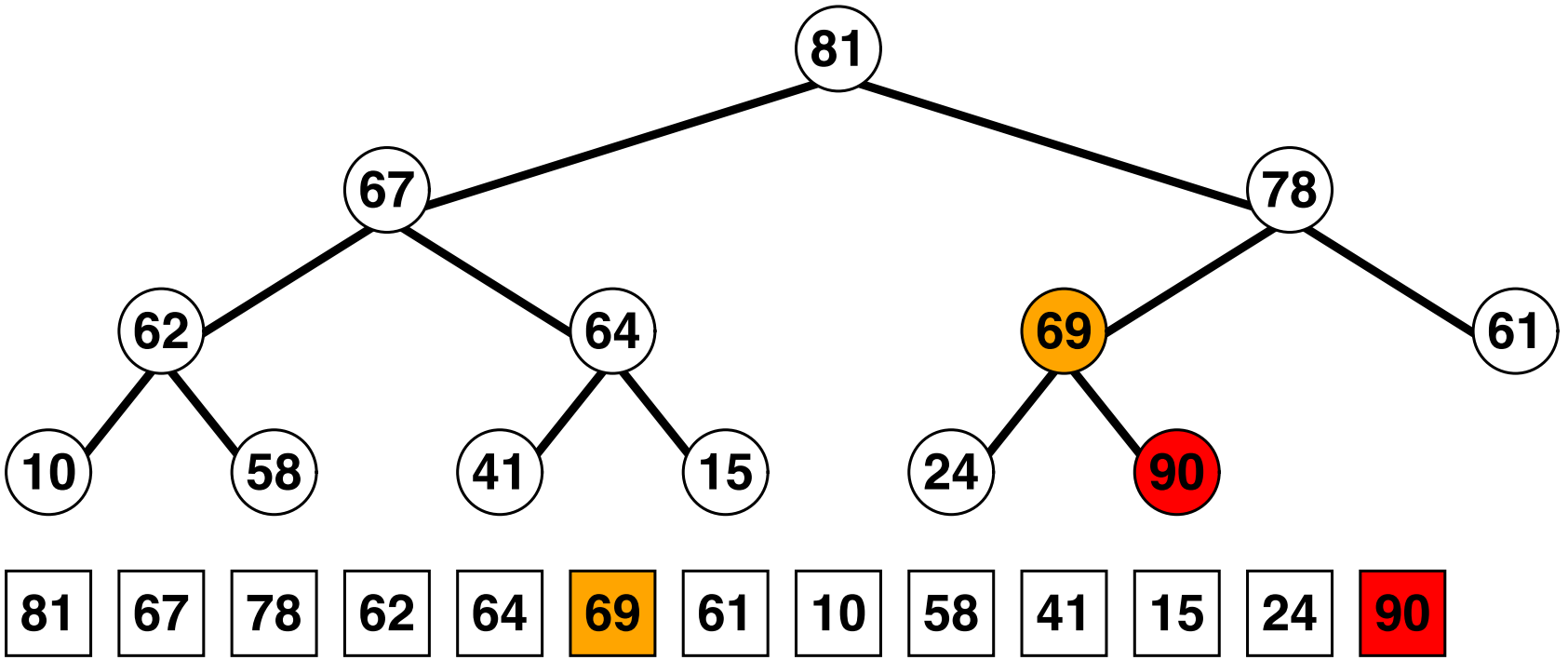
Definição e Propriedades

- Vamos assumir que temos uma estrutura de dados em que apenas um dos nós viola as propriedades de Fila de Prioridade.
- Usamos a operação de subida quando a chave desse nó é maior do que a chave do seu nó pai.

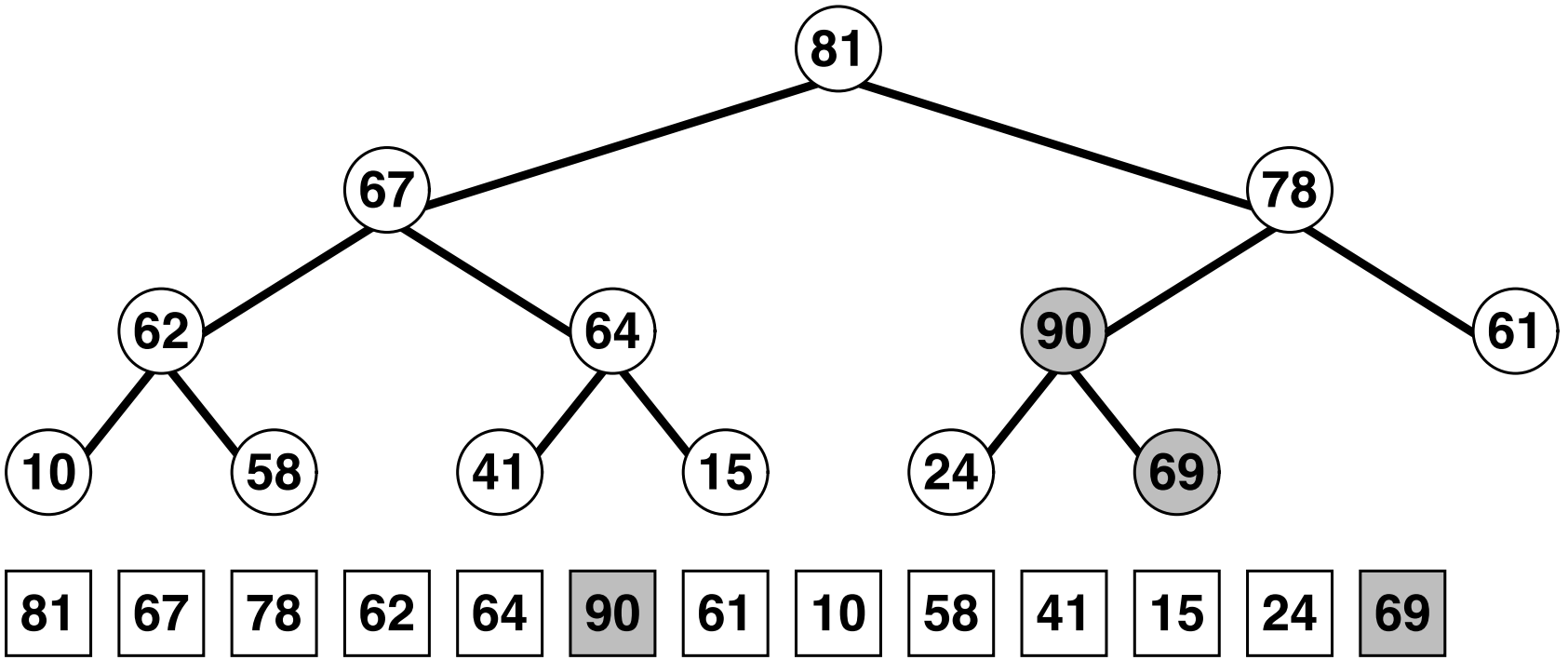
Operação de Subida



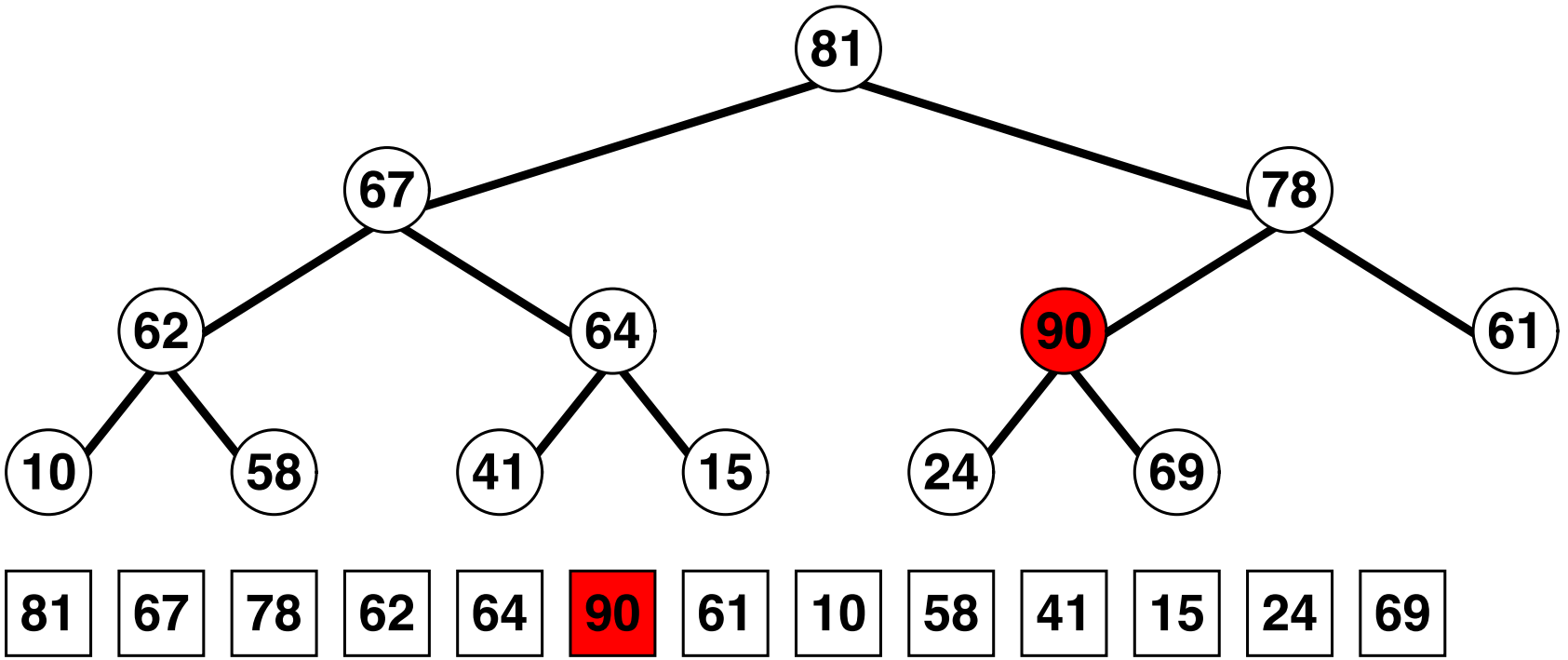
Operação de Subida



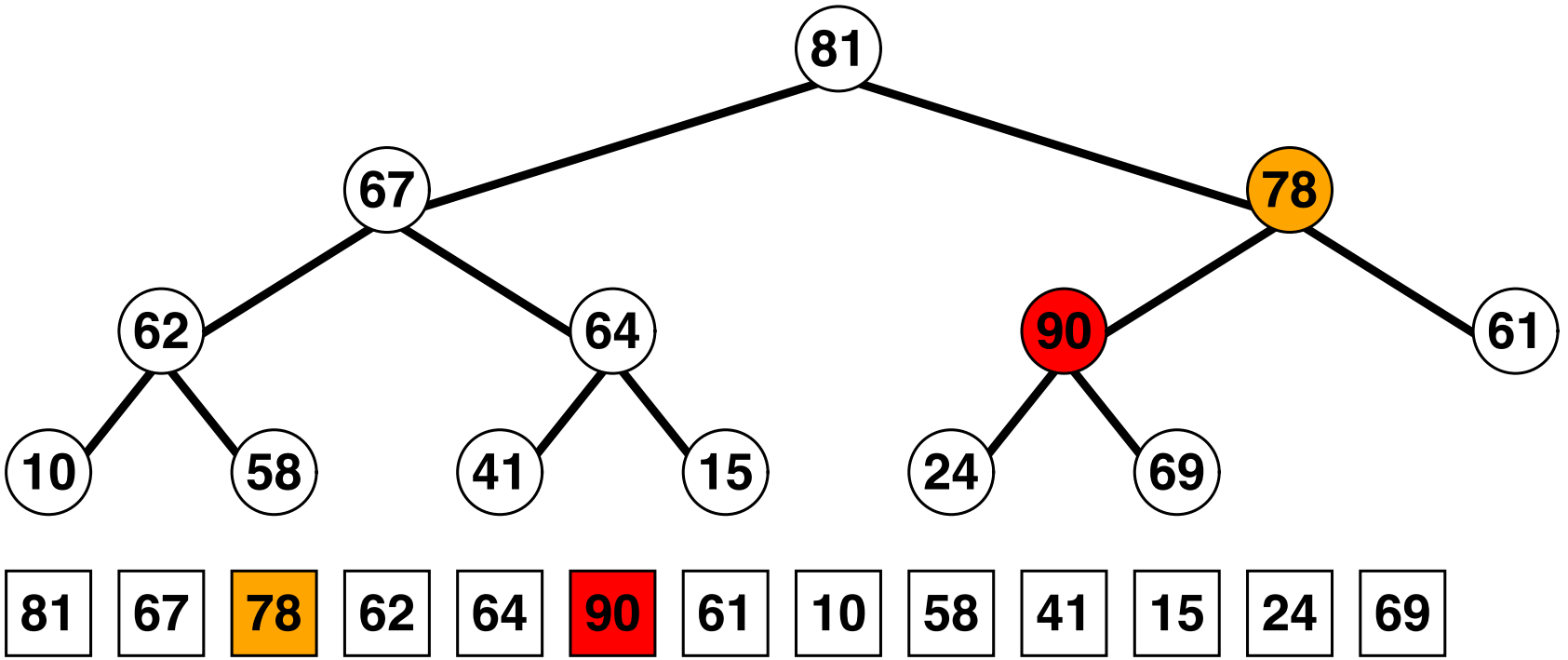
Operação de Subida



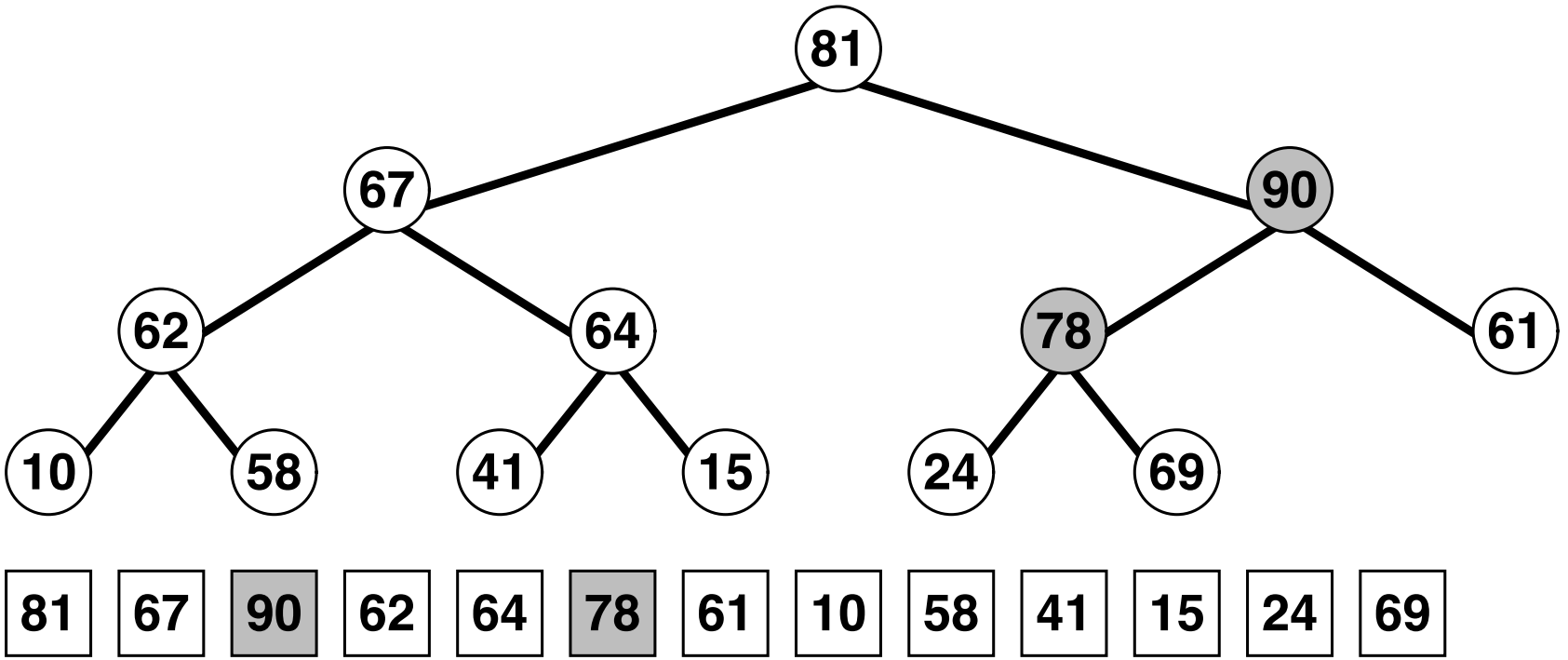
Operação de Subida



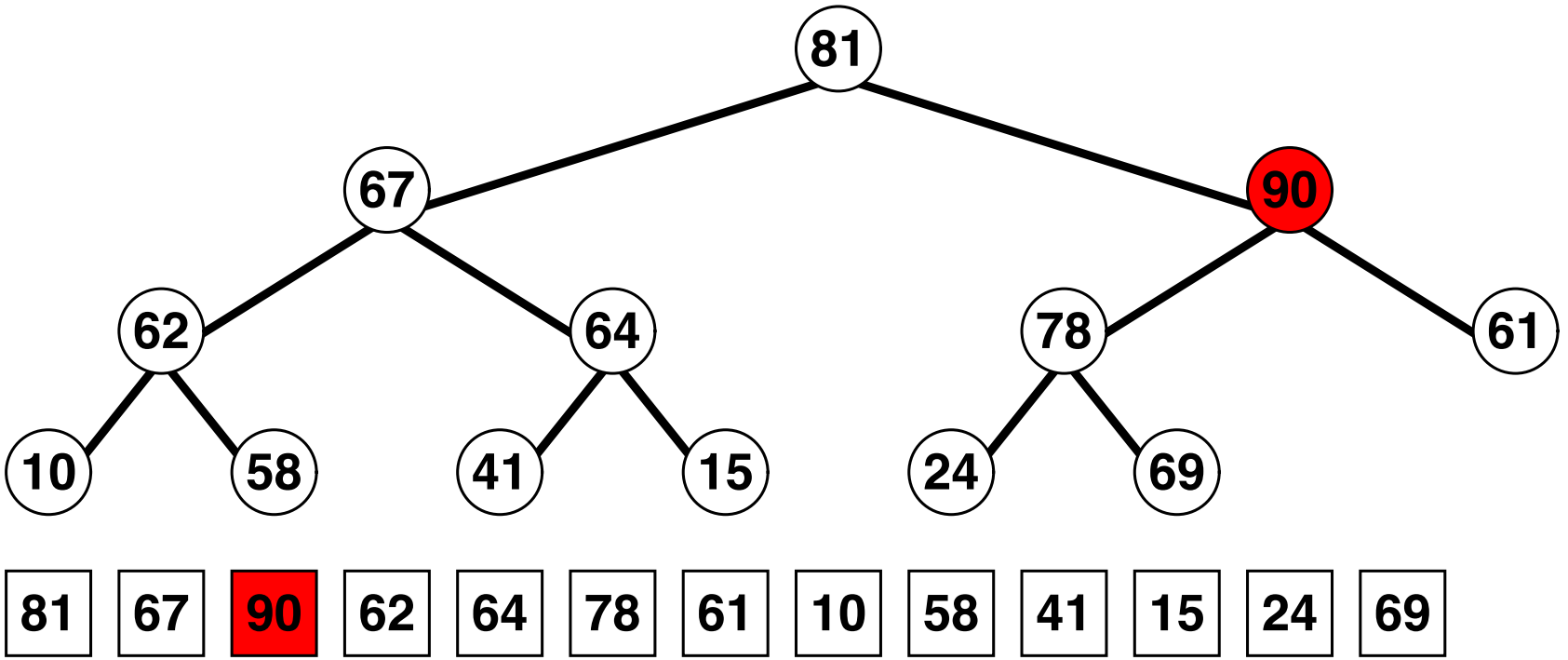
Operação de Subida



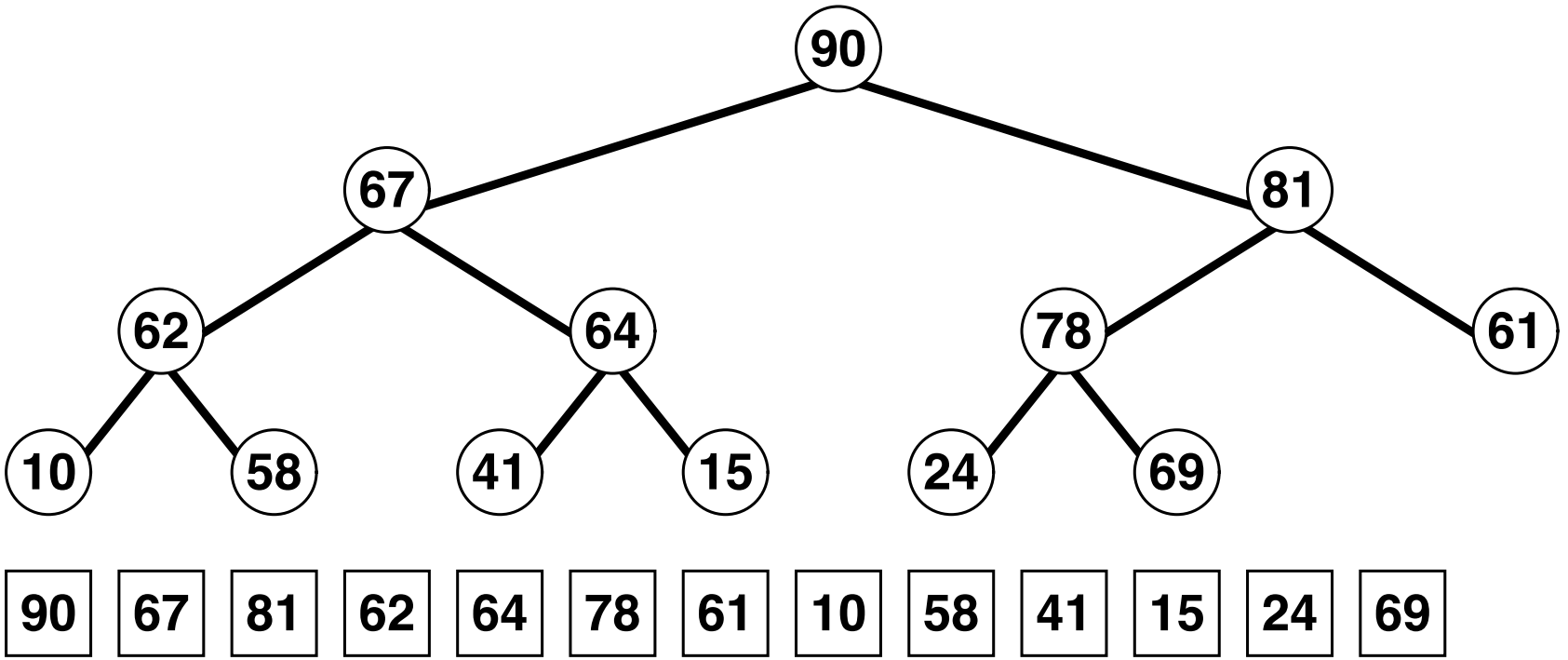
Operação de Subida



Operação de Subida



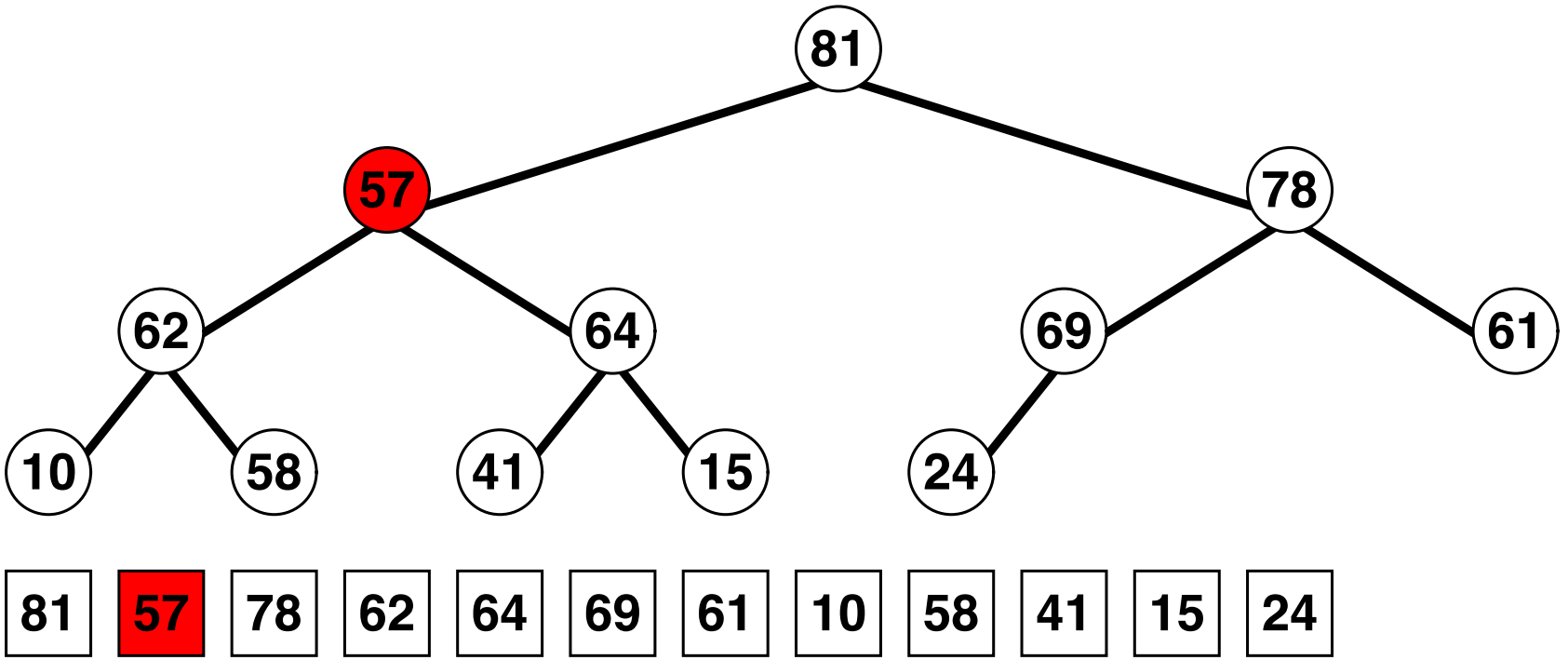
Operação de Subida



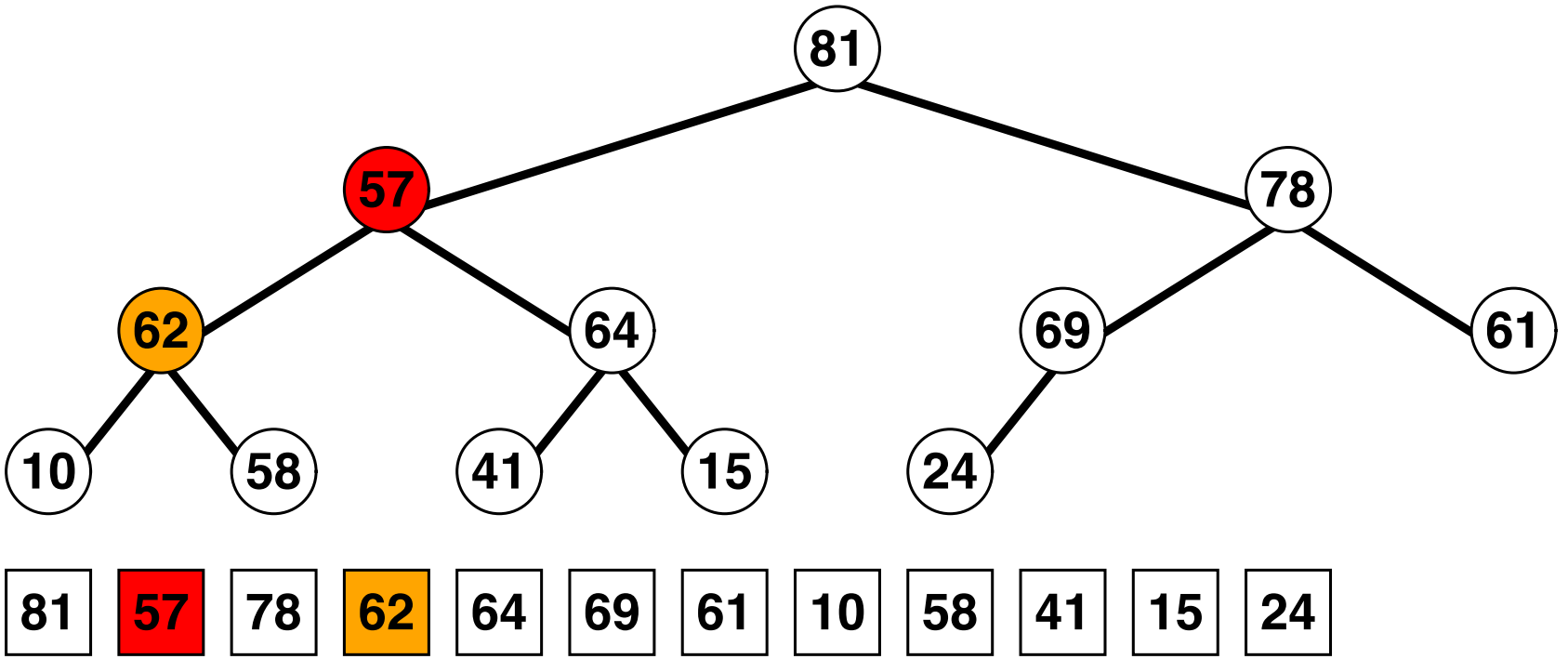
Definição e Propriedades

- Vamos assumir que temos uma estrutura de dados em que apenas um dos nós viola as propriedades de Fila de Prioridade.
- Usamos a operação de subida quando a chave desse nó é maior do que a chave do seu nó pai.
- Usamos a operação de descida quando a chave desse nó é menor do que a chave de um de seus filhos.

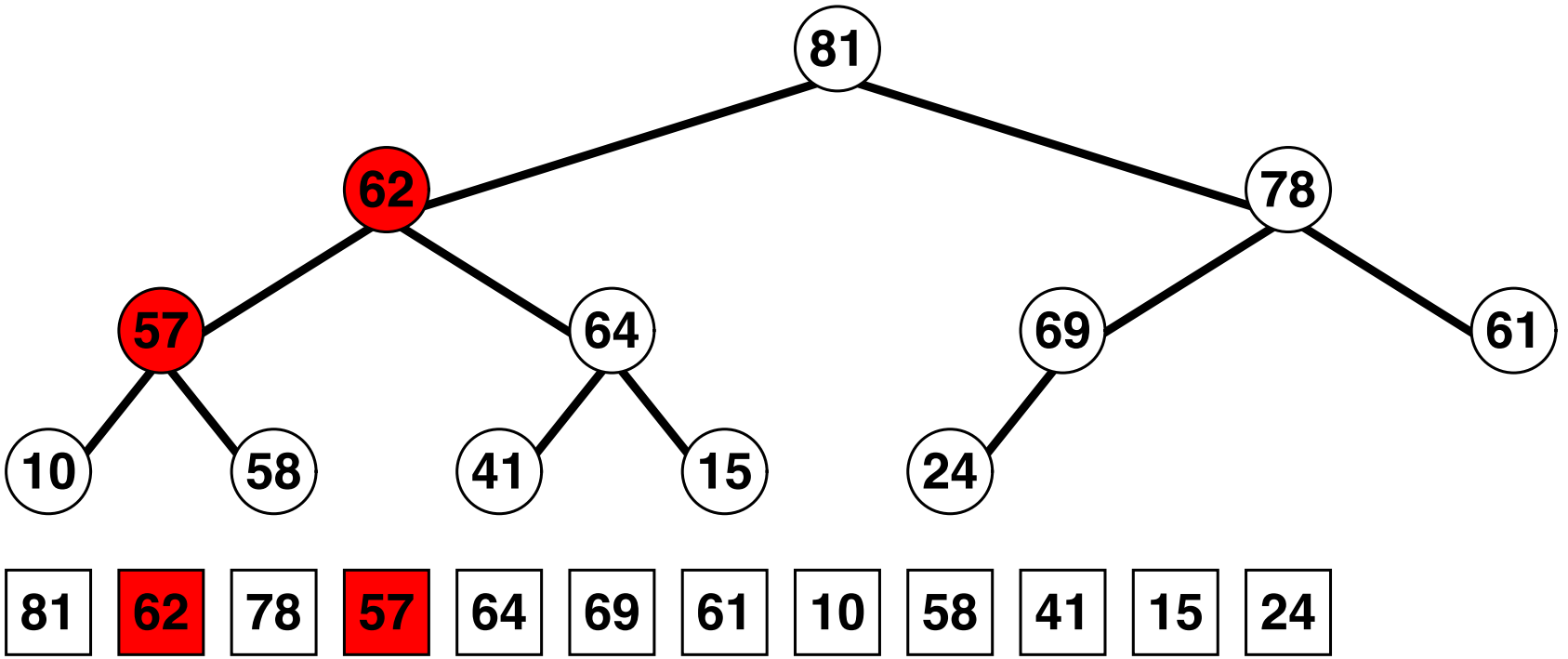
Operação de Descida



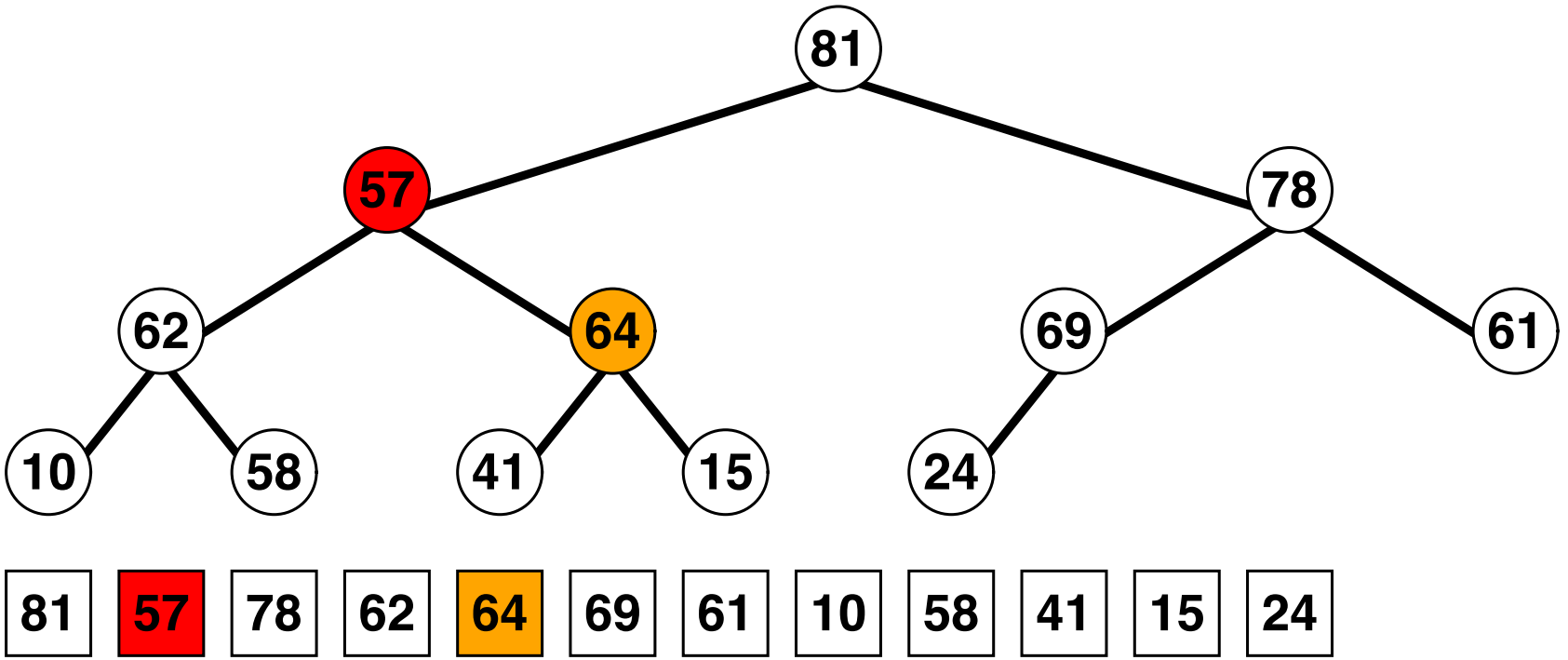
Operação de Descida - Supondo um caminho errado



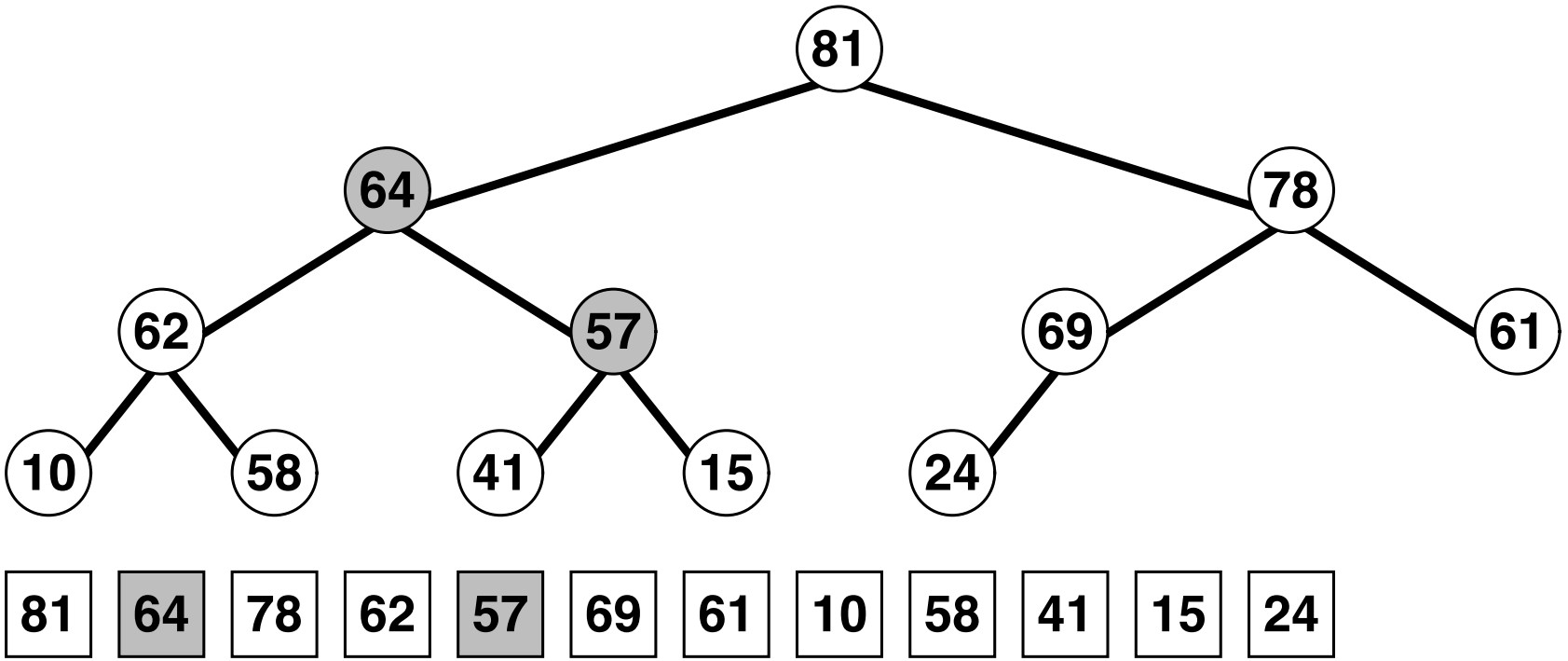
Operação de Descida - Supondo um caminho errado



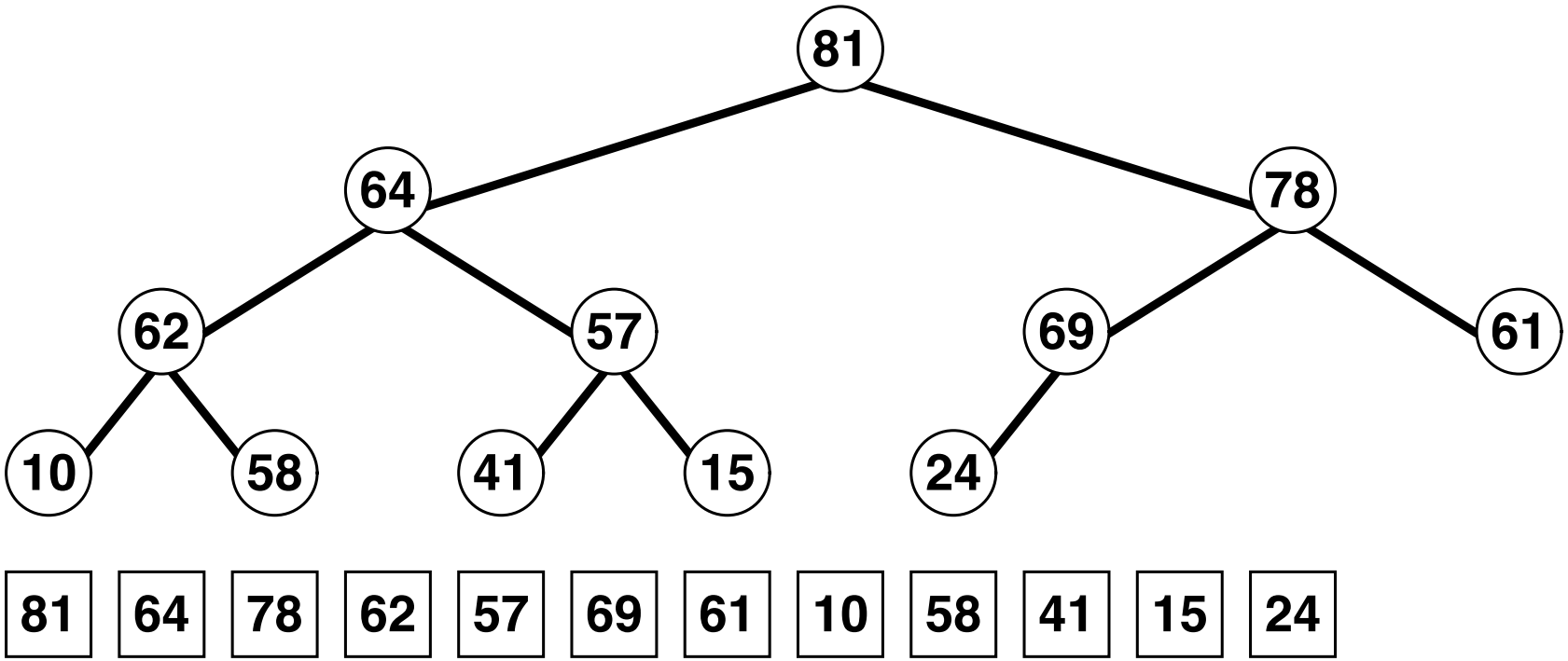
Operação de Descida - Seguindo o caminho correto



Operação de Descida - Seguindo o caminho correto



Operação de Descida - Seguindo o caminho correto



Roteiro

Heap - Registro

Armazenando as Informações da Heap

```
#define MAX 15

typedef struct {
  int chaves[MAX];
  int tam;
} Heap;
```

Subida

Algoritmo em C

```
void subida(Heap *h, int m) {
  int j = (m-1)/2
  int x = h->chaves[m];

  while ((m>0) && h->chaves[j] < x) {
    h->chaves[m] = h->chaves[j];
    m = j;
    j = (j-1)/2;
  }
  h->chaves[m] = x;
}
```

Descida

Algoritmo em C

```
void descida(Heap *h, int m) {
  int j = 2*m+1;
  int x = h->chaves[m];
  while (j< h->tam) {
    if ((j< h->tam -1) && h->chaves[j]< h->chaves[j+1])
      j++;
    if (x < h->chaves[j]) {
      h->vetor[m] = h->vetor[j];
      m = j;
      j = 2*m + 1;
    } else break;
  }
  h->chaves[m] = x;
}
```

Roteiro

Inserção

- Vamos assumir que temos uma Heap em que queremos inserir um novo elemento.

Inserção

- Vamos assumir que temos uma Heap em que queremos inserir um novo elemento.
- Inserimos o elemento na última posição.

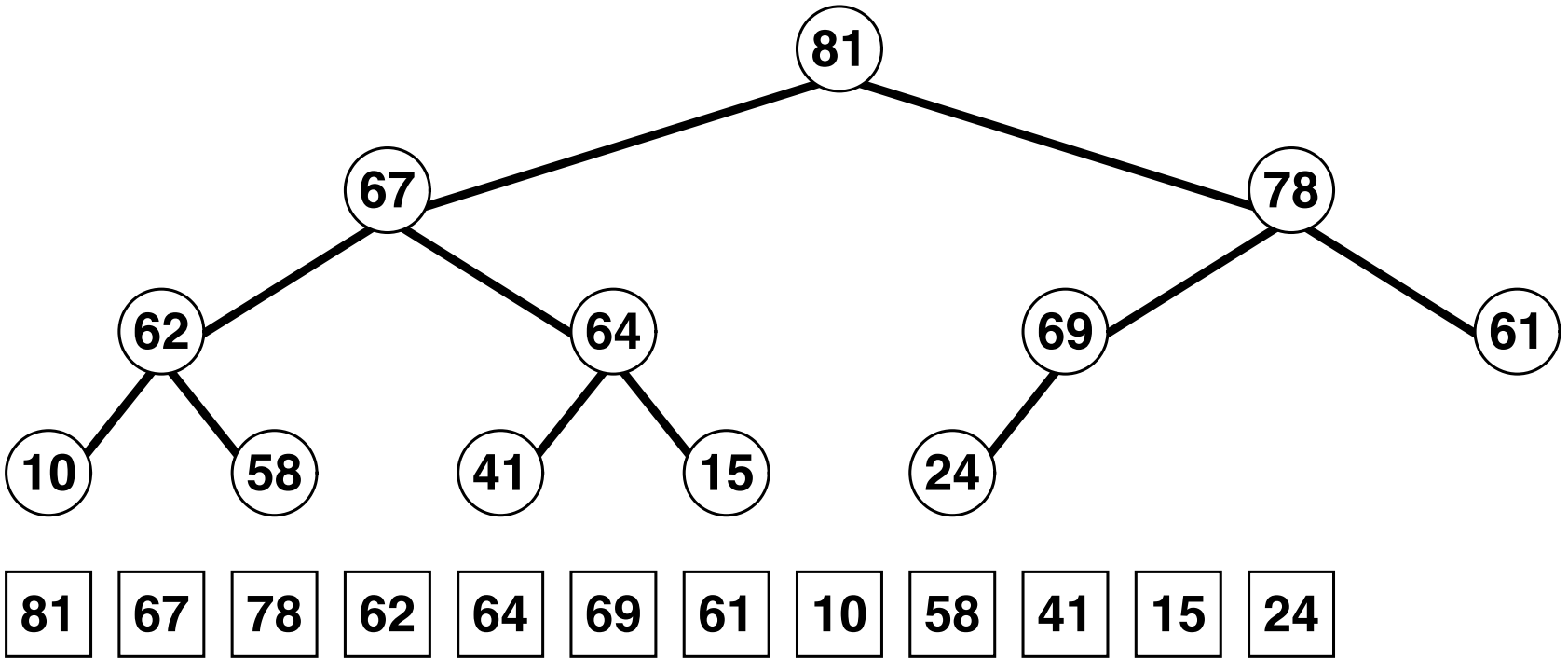
Inserção

- Vamos assumir que temos uma Heap em que queremos inserir um novo elemento.
- Inserimos o elemento na última posição.
- A nova chave é a única que pode violar as propriedades da heap, caso seja maior que os seus pais.

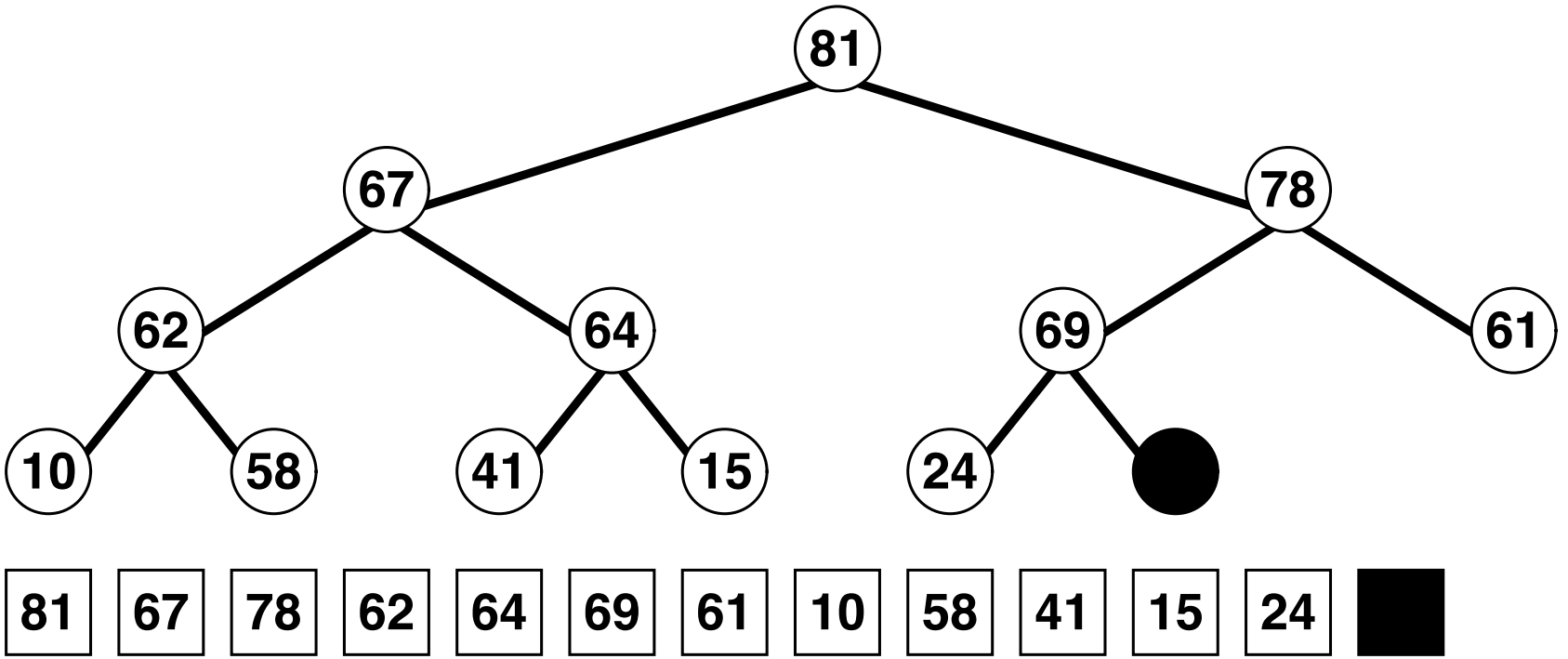
Inserção

- Vamos assumir que temos uma Heap em que queremos inserir um novo elemento.
- Inserimos o elemento na última posição.
- A nova chave é a única que pode violar as propriedades da heap, caso seja maior que os seus pais.
- Usamos o algoritmo de subida para refazer a Heap.

Inserção



Inserção



Inserção

Algoritmo em C

```
void insercao(Heap *h, int x) {
  chaves[h->tam] = x;
  h->tam++;
  subida(h, h->tam-1);
}
```

Remoção

- O elemento de maior prioridade precisa ser removido da Heap. Será necessário reconstruir a Heap.

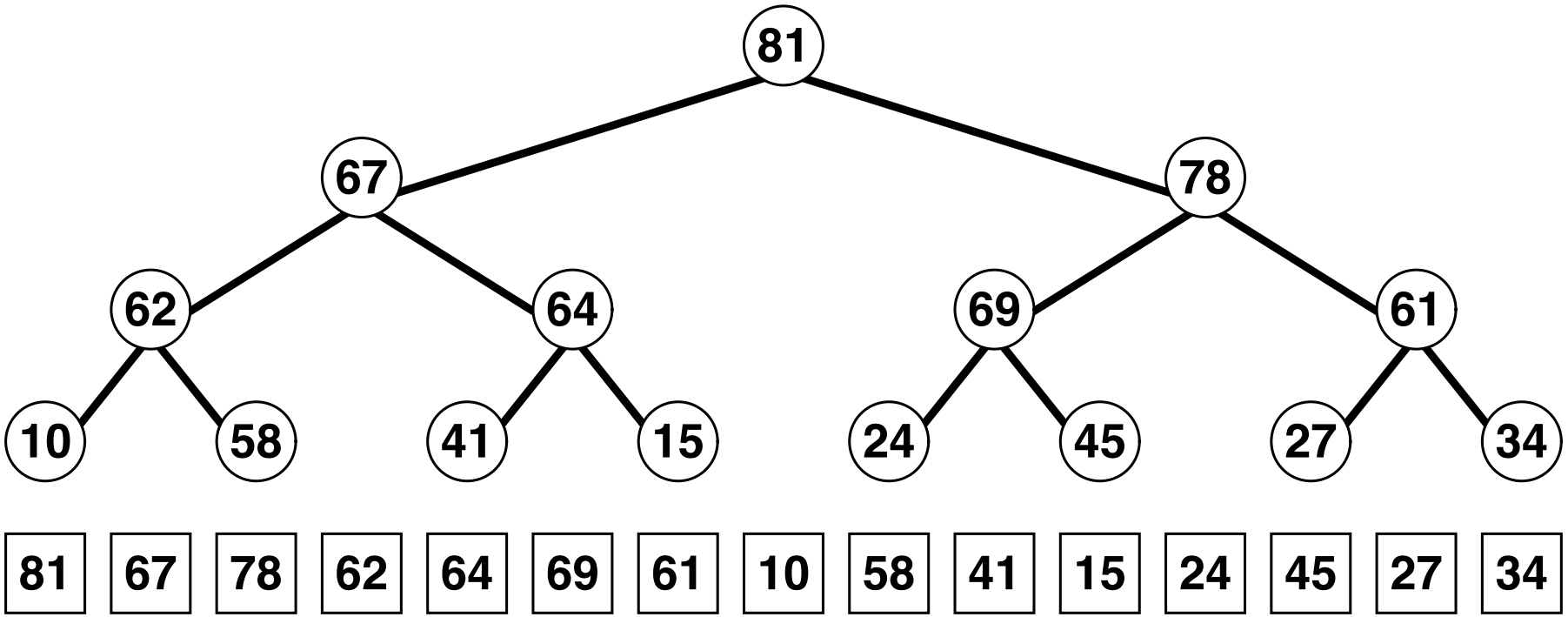
Remoção

- O elemento de maior prioridade precisa ser removido da Heap. Será necessário reconstruir a Heap.
- Colocamos o último elemento na primeira posição. Isso viola propriedades da Heap.

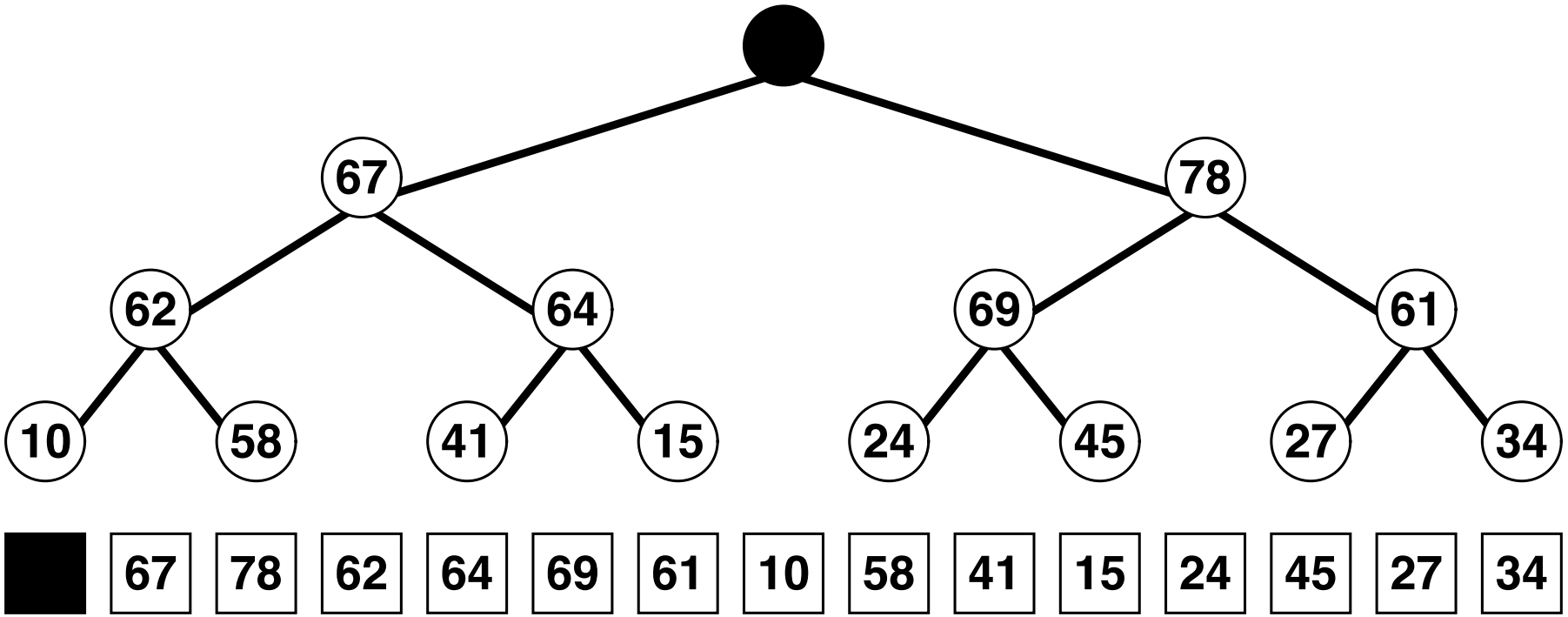
Remoção

- O elemento de maior prioridade precisa ser removido da Heap. Será necessário reconstruir a Heap.
- Colocamos o último elemento na primeira posição. Isso viola propriedades da Heap.
- Usamos o algoritmo de descida para refazer a Heap.

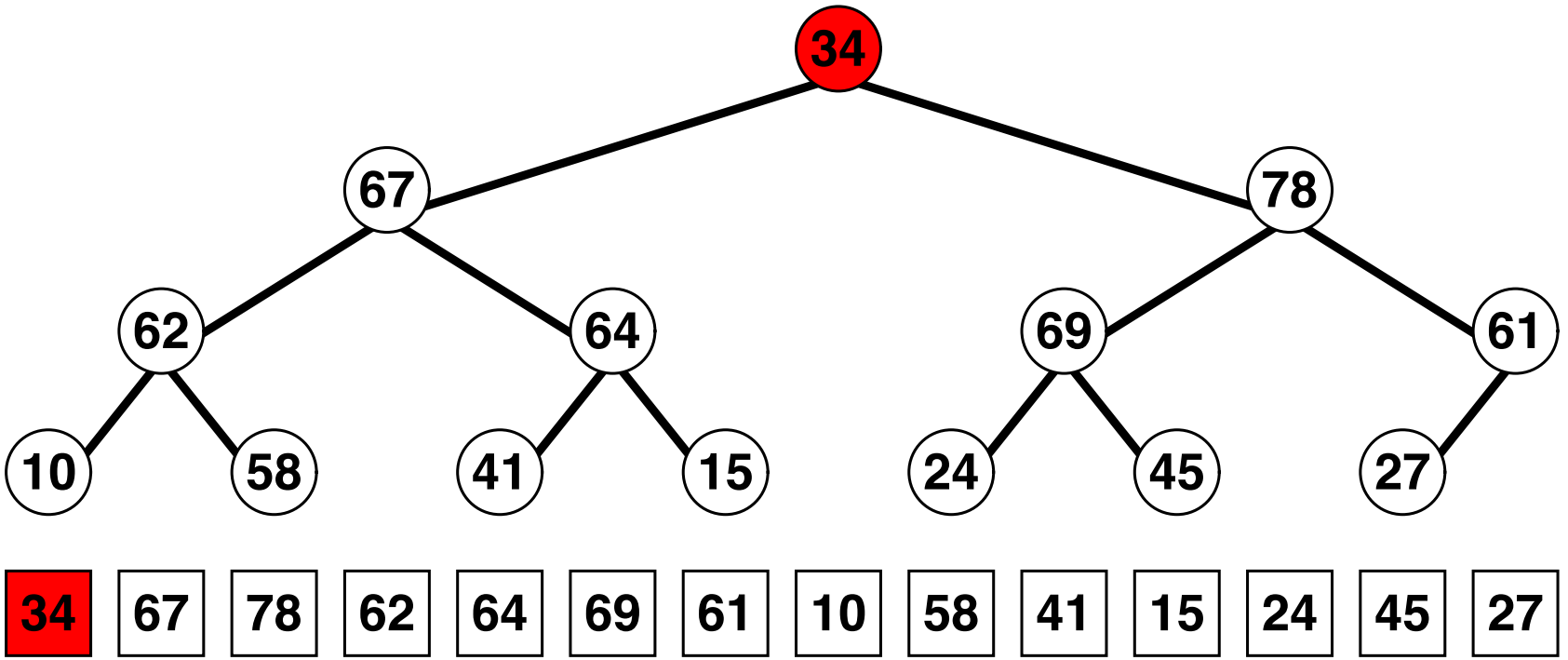
Remoção


81
67
78
62
64
69
61
10
58
41
15
24
45
27
34
81
67
78
62
64
69
61
10
58
41
15
24
45
27
34

Remoção


67
78
62
64
69
61
10
58
41
15
24
45
27
34
67
78
62
64
69
61
10
58
41
15
24
45
27
34

Remoção


34
67
78
62
64
69
61
10
58
41
15
24
45
27
34
67
78
62
64
69
61
10
58
41
15
24
45
27

Remoção

Algoritmo em C

```
void remocao(Heap *h, int *x) {
  *x = h->chaves[0];
  h->tam--;
  h->chaves[0] =  h->chaves[h->tam];
  descida(h, 0);
}
```

Roteiro

Construção Inicial da Heap

- A construção inicial de uma heap pode ser efetuada usando a seguinte estratégia.

Construção Inicial da Heap

- A construção inicial de uma heap pode ser efetuada usando a seguinte estratégia.

Código em C

```
void constroi_heap(Heap *h) {
  int i;
  for (i=h->tam/2 -1; i >= 0; i--)
     descida(h,i);
}
```

Construção Inicial da Heap

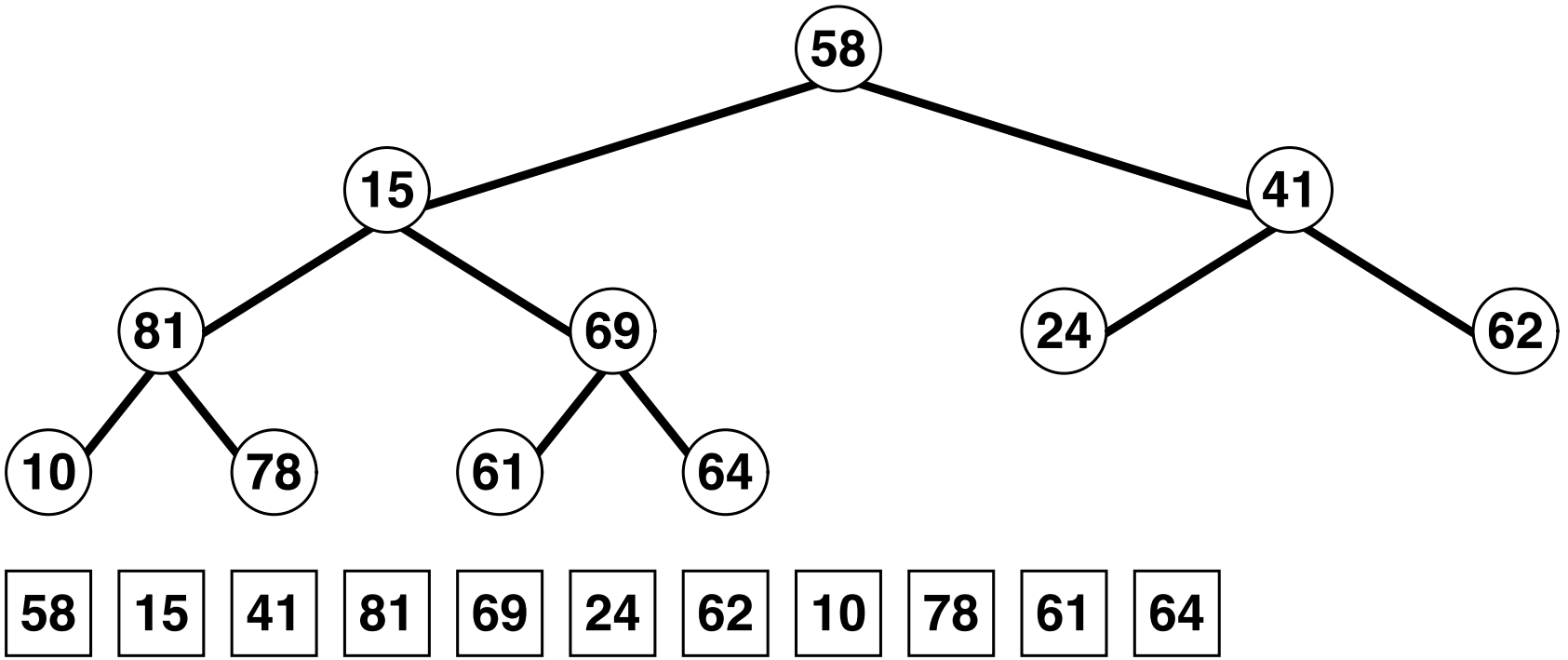
- A construção inicial de uma heap pode ser efetuada usando a seguinte estratégia.

Código em C

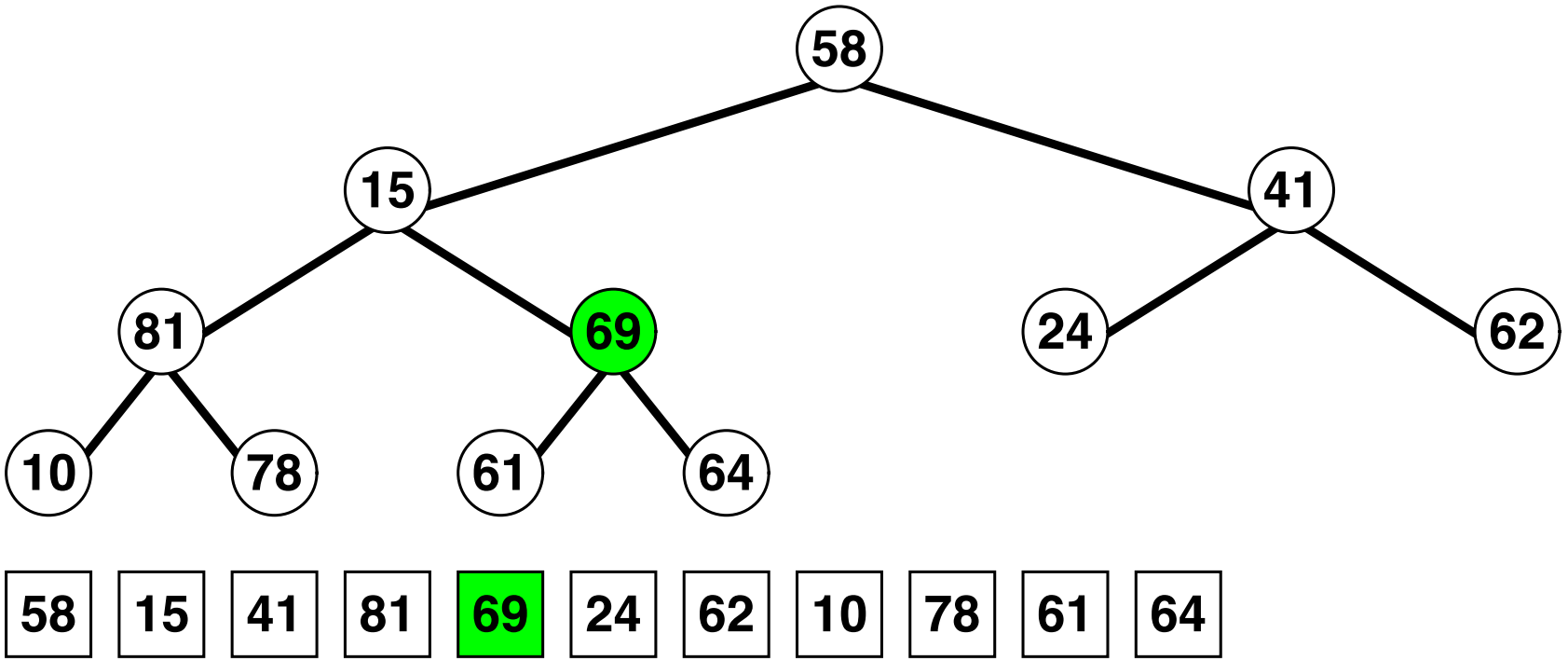
```
void constroi_heap(Heap *h) {
  int i;
  for (i=h->tam/2 -1; i >= 0; i--)
     descida(h,i);
}
```

- É possível provar que esse código executa em $O(n)$.

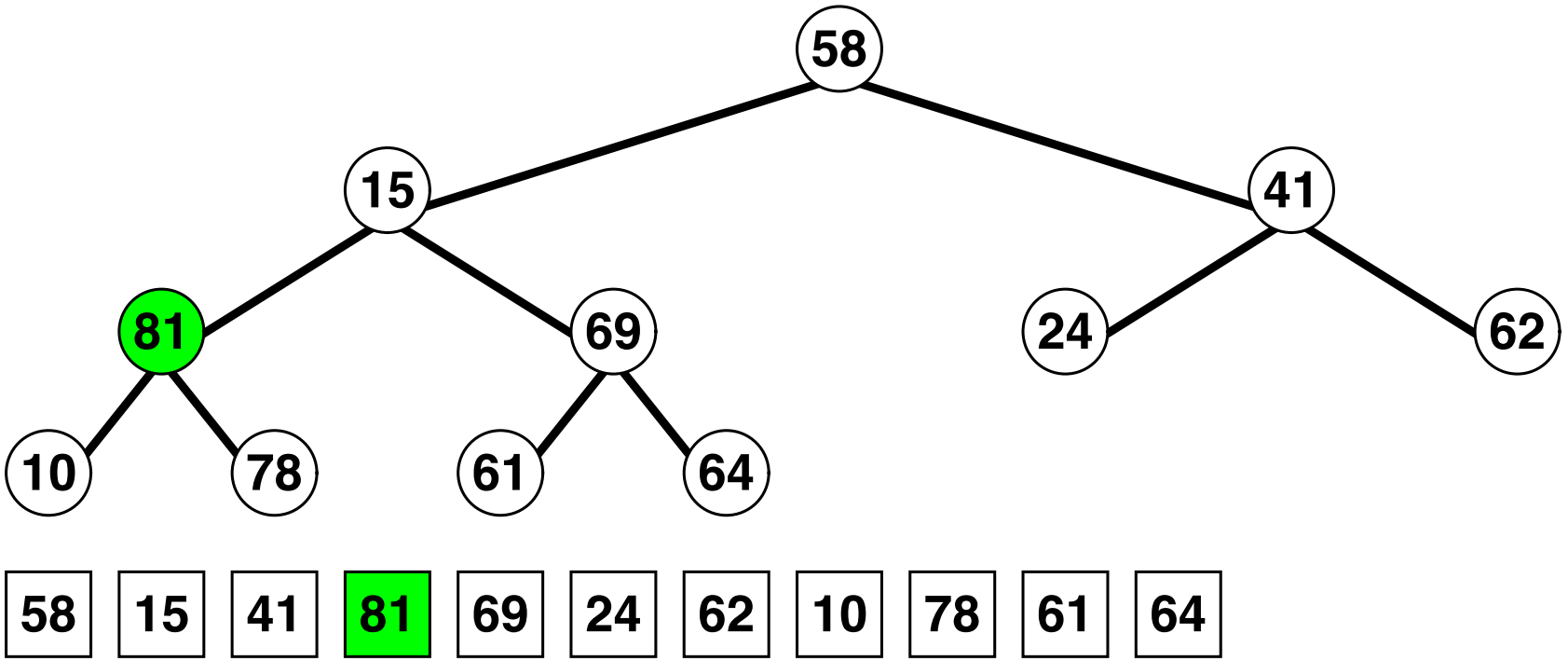
Construção Inicial da Heap



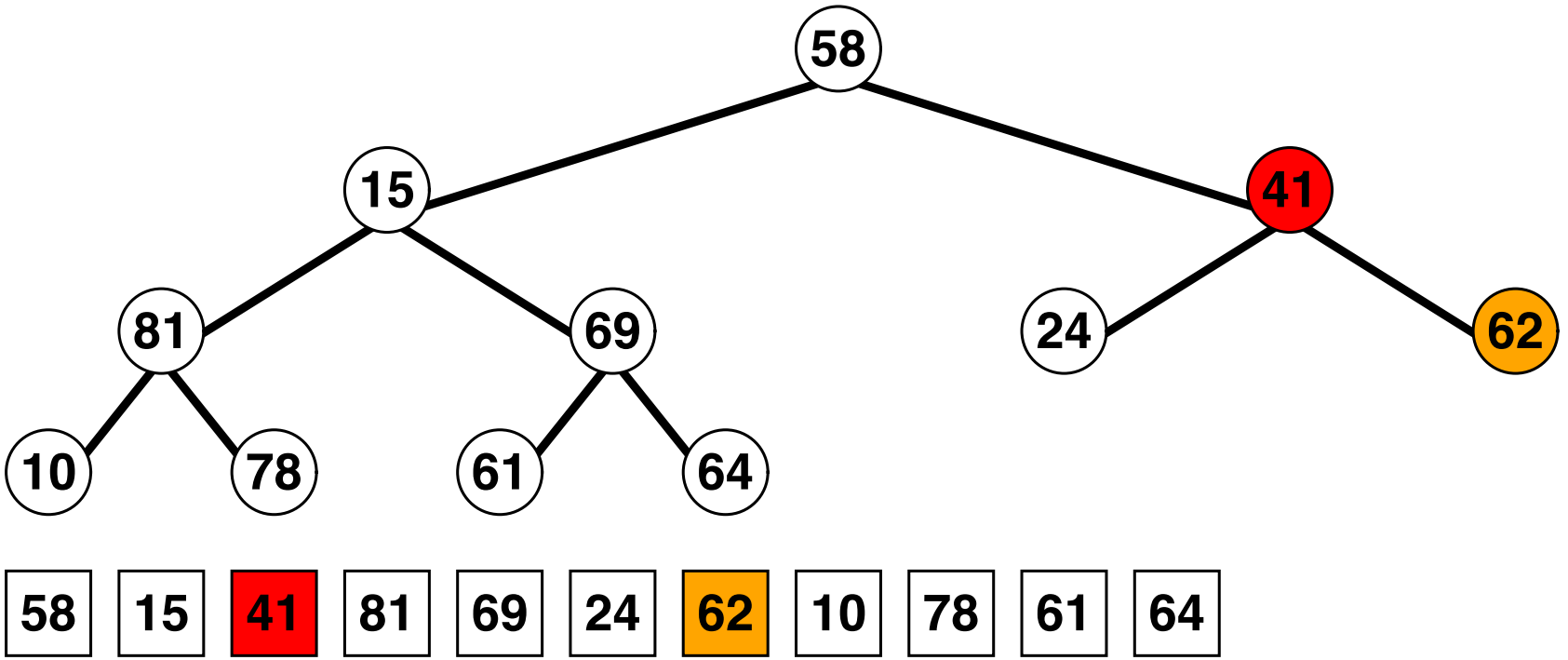
Construção Inicial da Heap



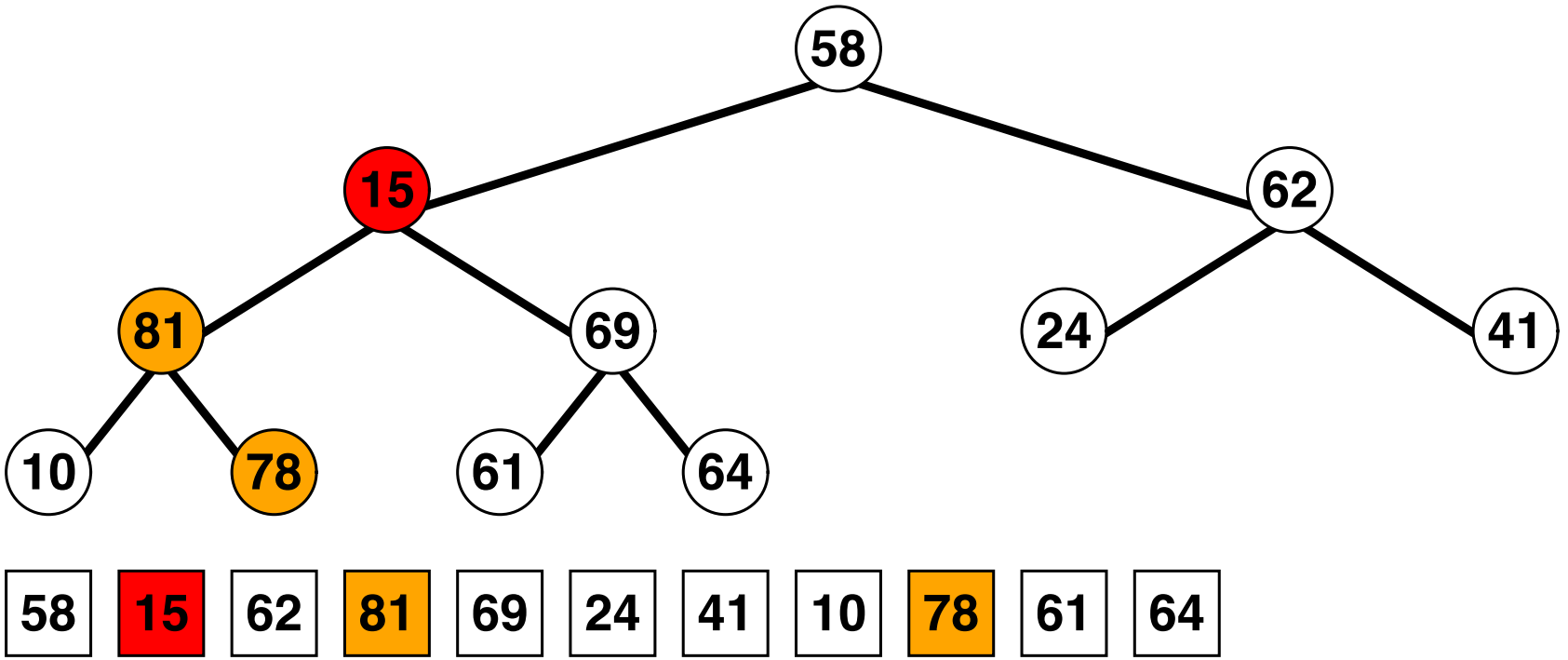
Construção Inicial da Heap



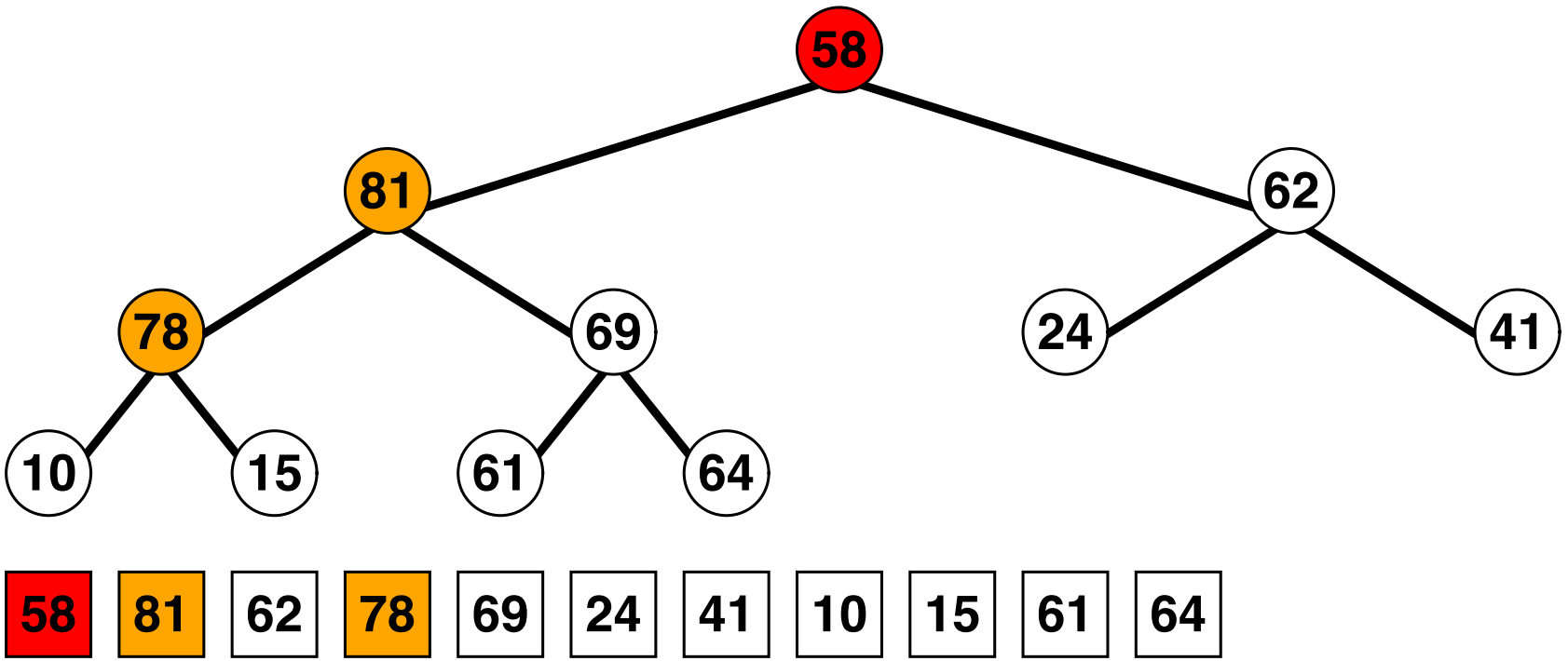
Construção Inicial da Heap



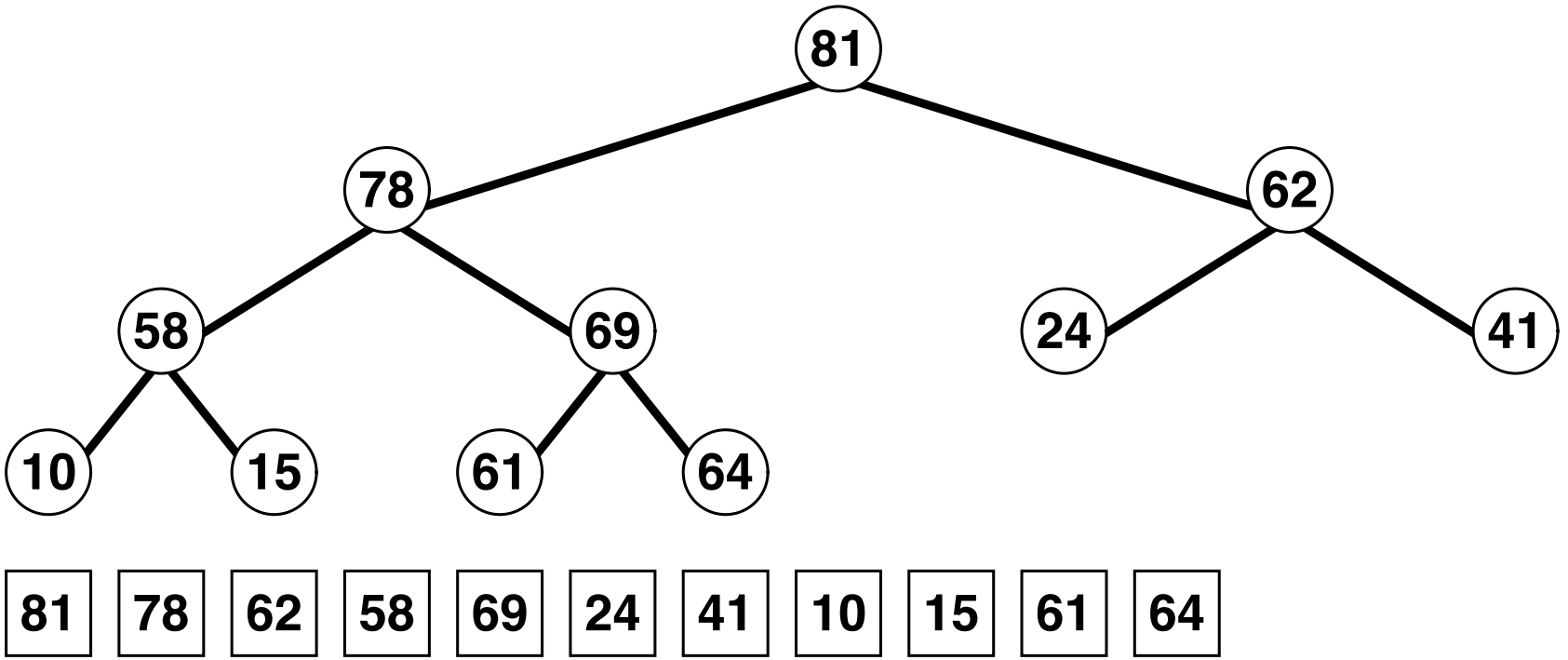
Construção Inicial da Heap



Construção Inicial da Heap



Construção Inicial da Heap



Construção Inicial do Heap

- Cuidado deve ser tomado para não usar a estratégia a seguir, que resultaria em um tempo de execução de $O(n \log n)$.

Construção Inicial do Heap

- Cuidado deve ser tomado para não usar a estratégia a seguir, que resultaria em um tempo de execução de $O(n \log n)$.

Código em C

```
void constroi_heap2(Heap *h) {
  int i;
  for (i=1; i < h->tam; i++)
     subida(h,i)
}
```

Roteiro

Algoritmo de Ordenação - Recapitulando

- Temos como entrada um vetor de chaves.

Algoritmo de Ordenação - Recapitulando

- Temos como entrada um vetor de chaves.
- Ao final da ordenação o vetor deve estar ordenado do menor para o maior.

Algoritmo de Ordenação - Recapitulando

- Temos como entrada um vetor de chaves.
- Ao final da ordenação o vetor deve estar ordenado do menor para o maior.
- Algoritmo simples para resolver a tarefa: SelectionSort.

Algoritmo de Ordenação - Recapitulando

- Temos como entrada um vetor de chaves.
- Ao final da ordenação o vetor deve estar ordenado do menor para o maior.
- Algoritmo simples para resolver a tarefa: SelectionSort.
  - Executa em tempo $O(n^2)$.

Algoritmo de Ordenação - Recapitulando

- Temos como entrada um vetor de chaves.
- Ao final da ordenação o vetor deve estar ordenado do menor para o maior.
- Algoritmo simples para resolver a tarefa: SelectionSort.
  - Executa em tempo $O(n^2)$.
  - Computa a cada iteração o mínimo (máximo) da parte não ordenada do vetor e adiciona no início (final).

Algoritmo de Ordenação - Recapitulando

- Temos como entrada um vetor de chaves.
- Ao final da ordenação o vetor deve estar ordenado do menor para o maior.
- Algoritmo simples para resolver a tarefa: SelectionSort.
  - Executa em tempo $O(n^2)$.
  - Computa a cada iteração o mínimo (máximo) da parte não ordenada do vetor e adiciona no início (final).
  - Algoritmo ingênuo e pouco sofisticado.

Algoritmo de Ordenação - Recapitulando

- Temos como entrada um vetor de chaves.
- Ao final da ordenação o vetor deve estar ordenado do menor para o maior.
- Algoritmo simples para resolver a tarefa: SelectionSort.
  - Executa em tempo $O(n^2)$.
  - Computa a cada iteração o mínimo (máximo) da parte não ordenada do vetor e adiciona no início (final).
  - Algoritmo ingênuo e pouco sofisticado.
- Se adicionarmos uma Heap ao algoritmo para cálcular o mínimo (máximo)?

SelectionSort e HeapSort (Conceitualmente)

SelectionSort - Pseudocódigo

```
SelectionSort(A = [A_1, A_1, ... , A_n-1])
  para (j = n-1; j>0; j--)
    - Procurar 'maior' no intervalo [A_1 ... A_j].
    - Trocar maior e A[j] de posição.
```

SelectionSort e HeapSort (Conceitualmente)

SelectionSort - Pseudocódigo

```
SelectionSort(A = [A_1, A_1, ... , A_n-1])
  para (j = n-1; j>0; j--)
    - Procurar 'maior' no intervalo [A_1 ... A_j].
    - Trocar maior e A[j] de posição.
```

HeapSort - Pseudocódigo

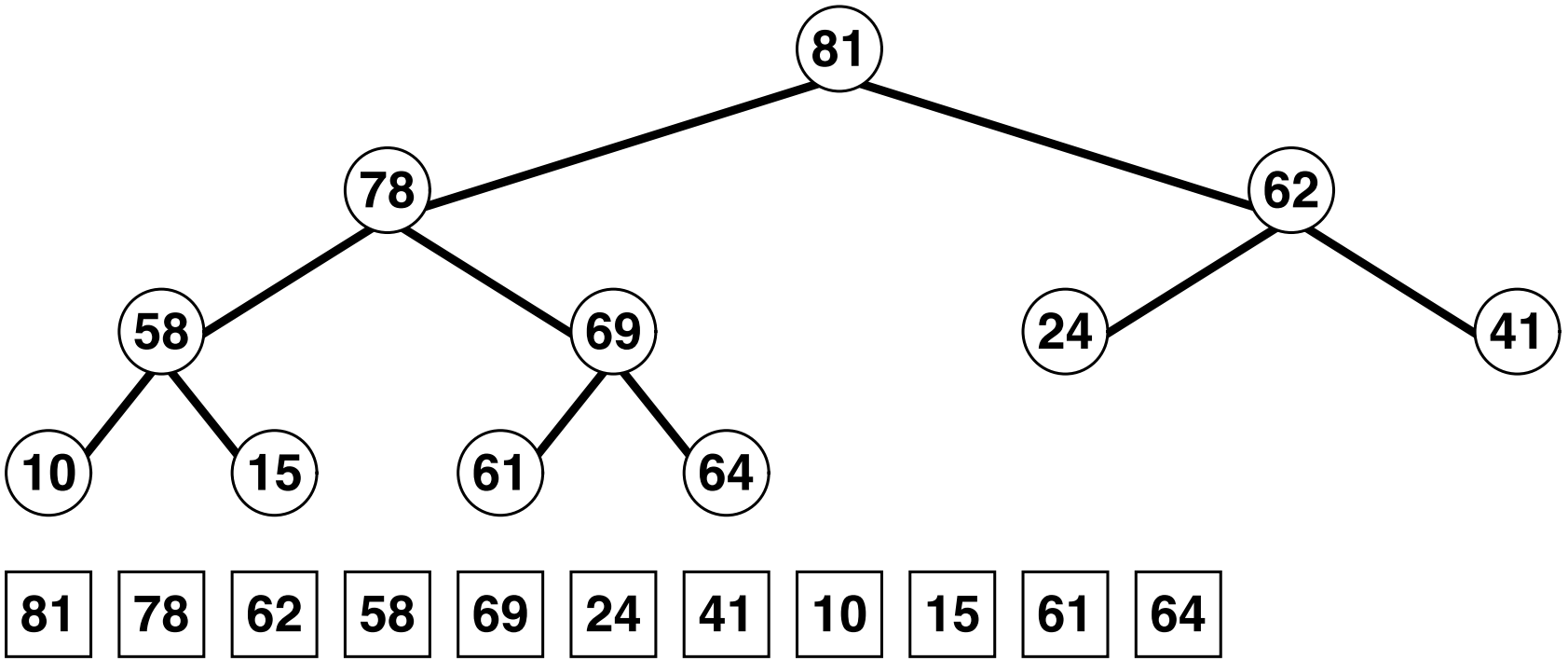
```
HeapSort(A = [A_1, A_2, ... , A_n-1])
  - Inicializar Heap h.
  para (j = n-1; j>0; j--)
    - Selecionar e remover 'maior' da raiz de h.
    - Reajustar h.
    - Trocar 'maior' e A[j] de posição.
```

SelectionSort e HeapSort (Conceitualmente)

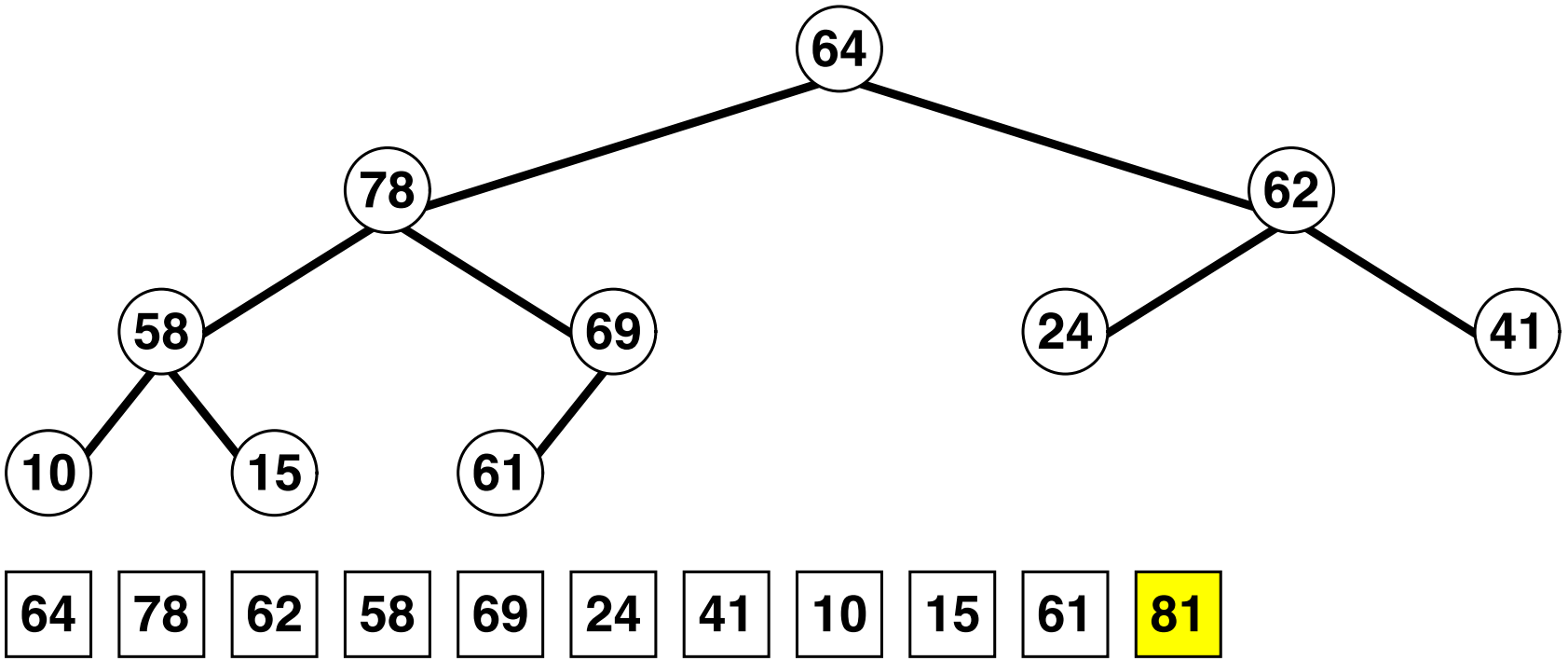
HeapSort - Pseudocódigo

```
void heapsort(Heap *h) {
  int i;
  for (i=h->tam/2 -1; i >= 0; i--)
     descida(h,i);
  for (i=h->tam-1; i>0; i--) {
    int chave = h->chaves[0];
    h->chaves[0] = h->chaves[i]
    h->chaves[i] = chave
    h->tam--;
    descida(h,0)
  }
  h->tam = n;
}
```

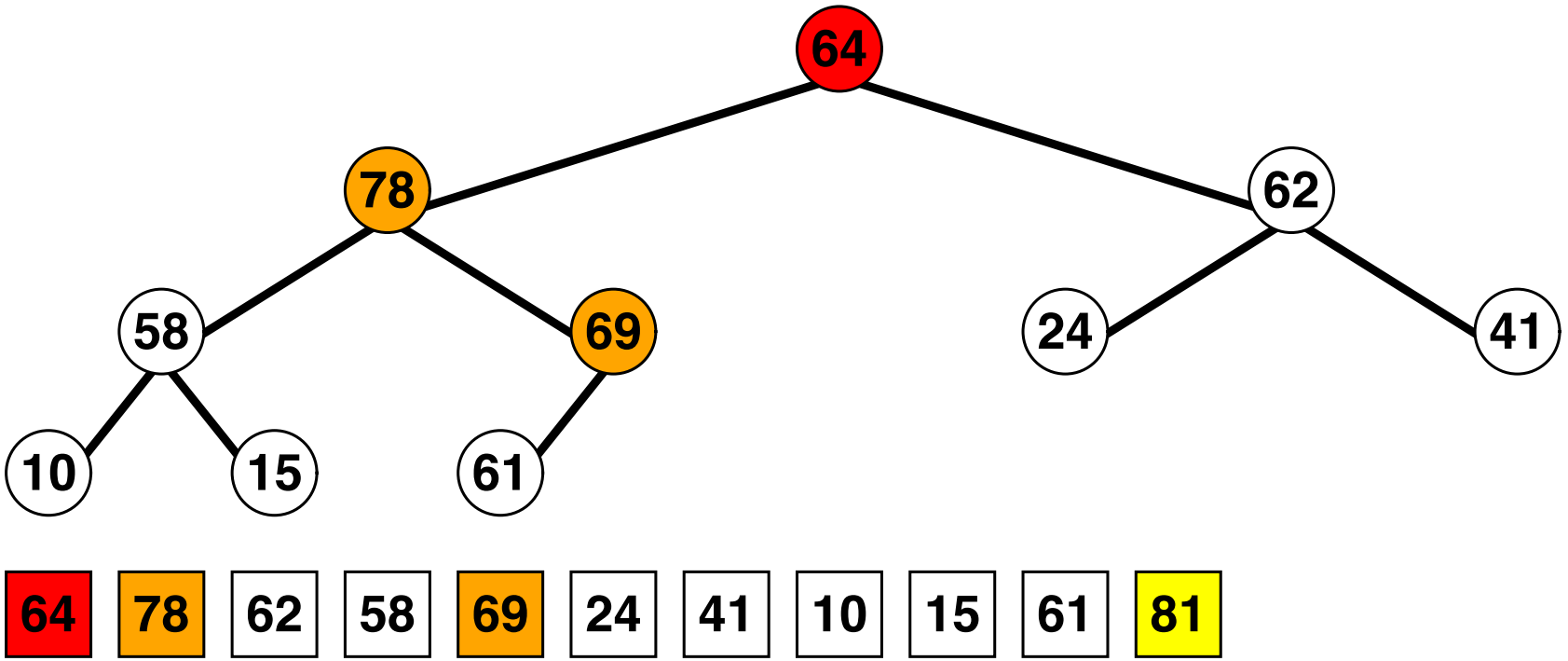
HeapSort



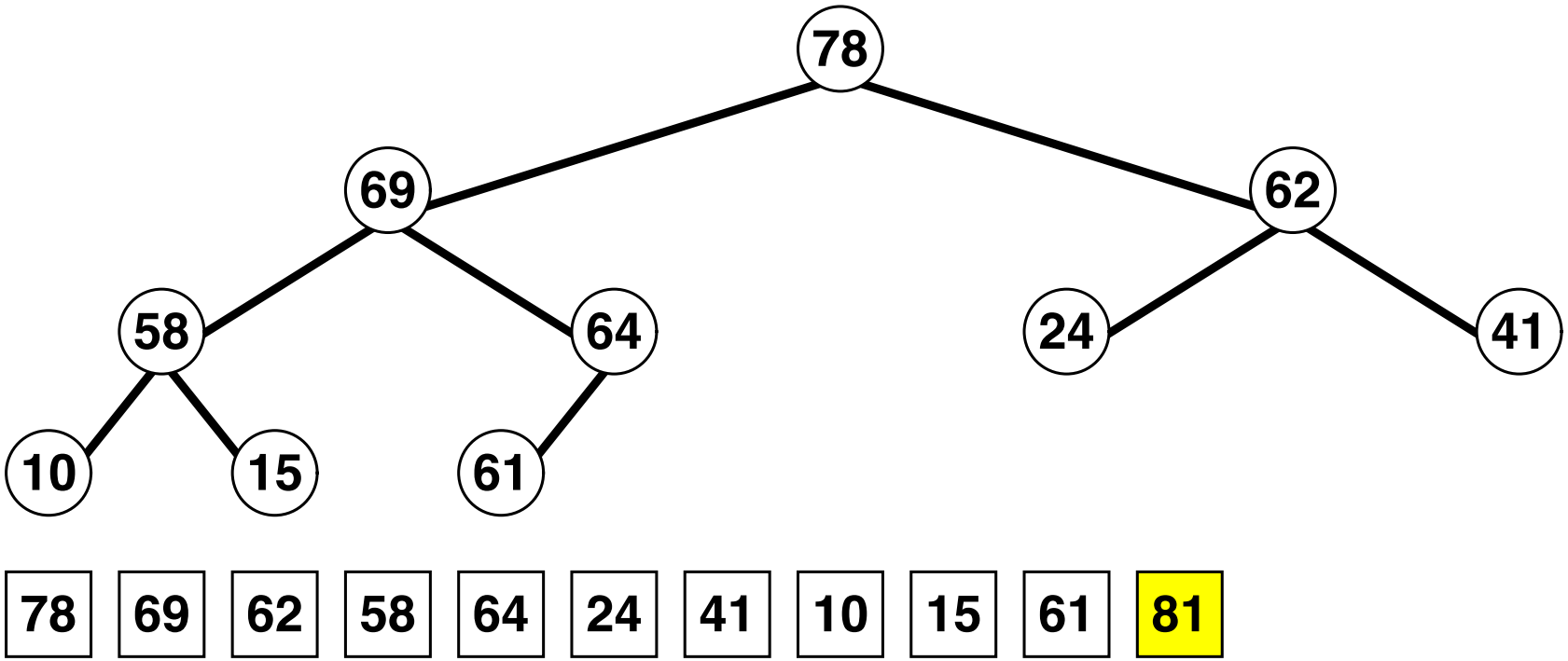
HeapSort



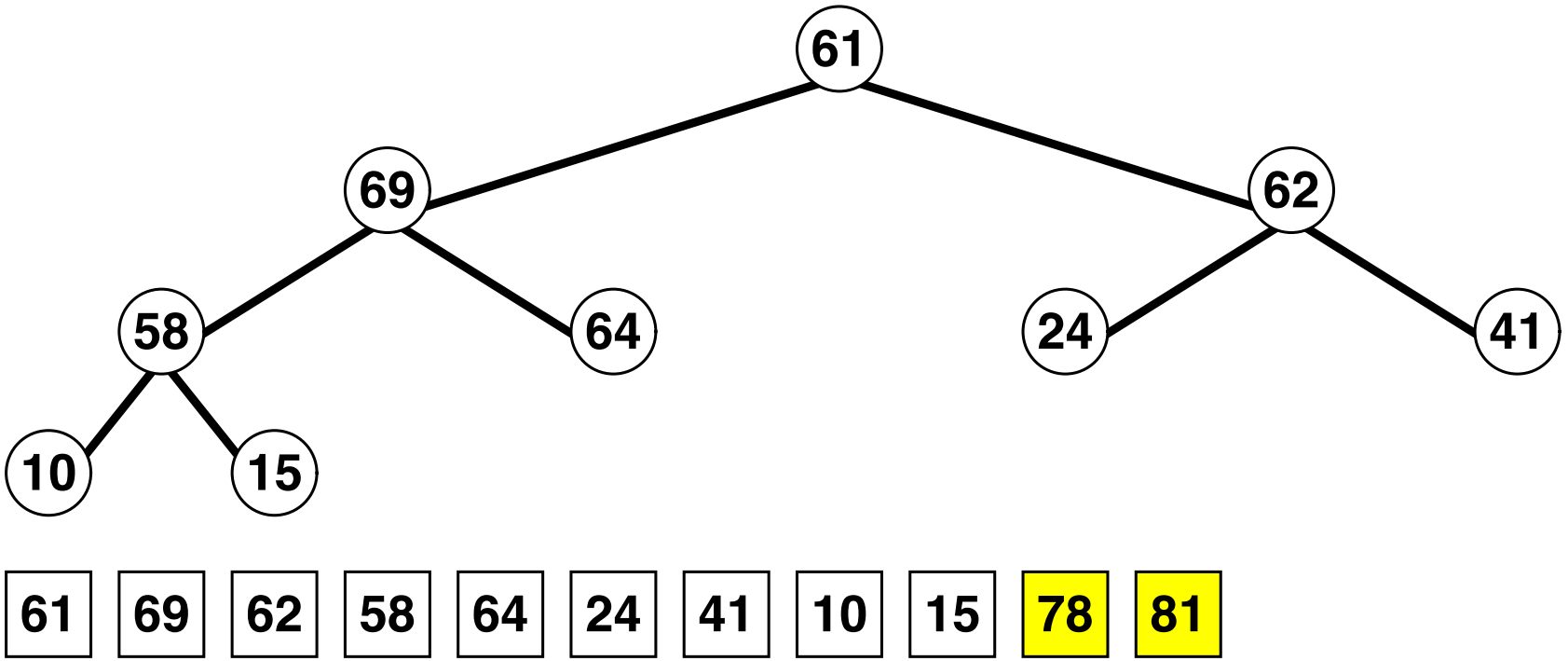
HeapSort



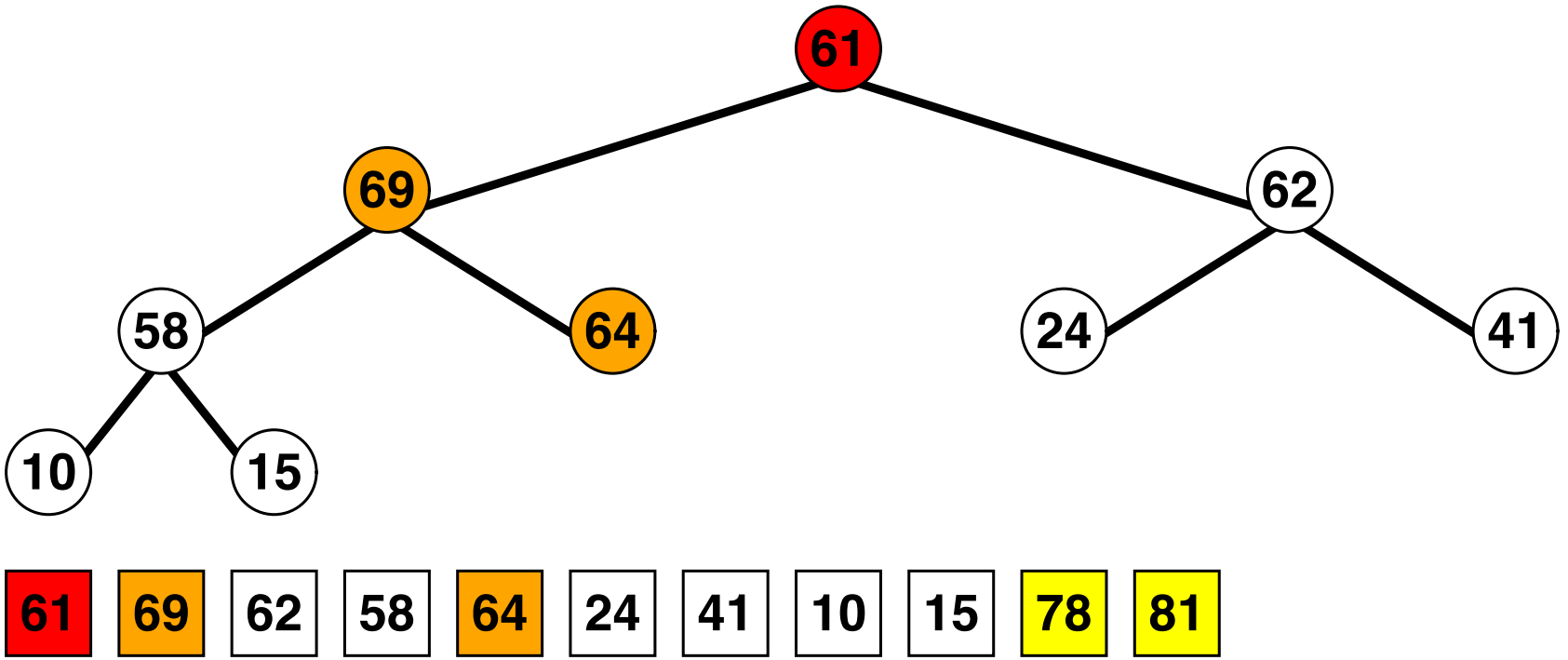
HeapSort



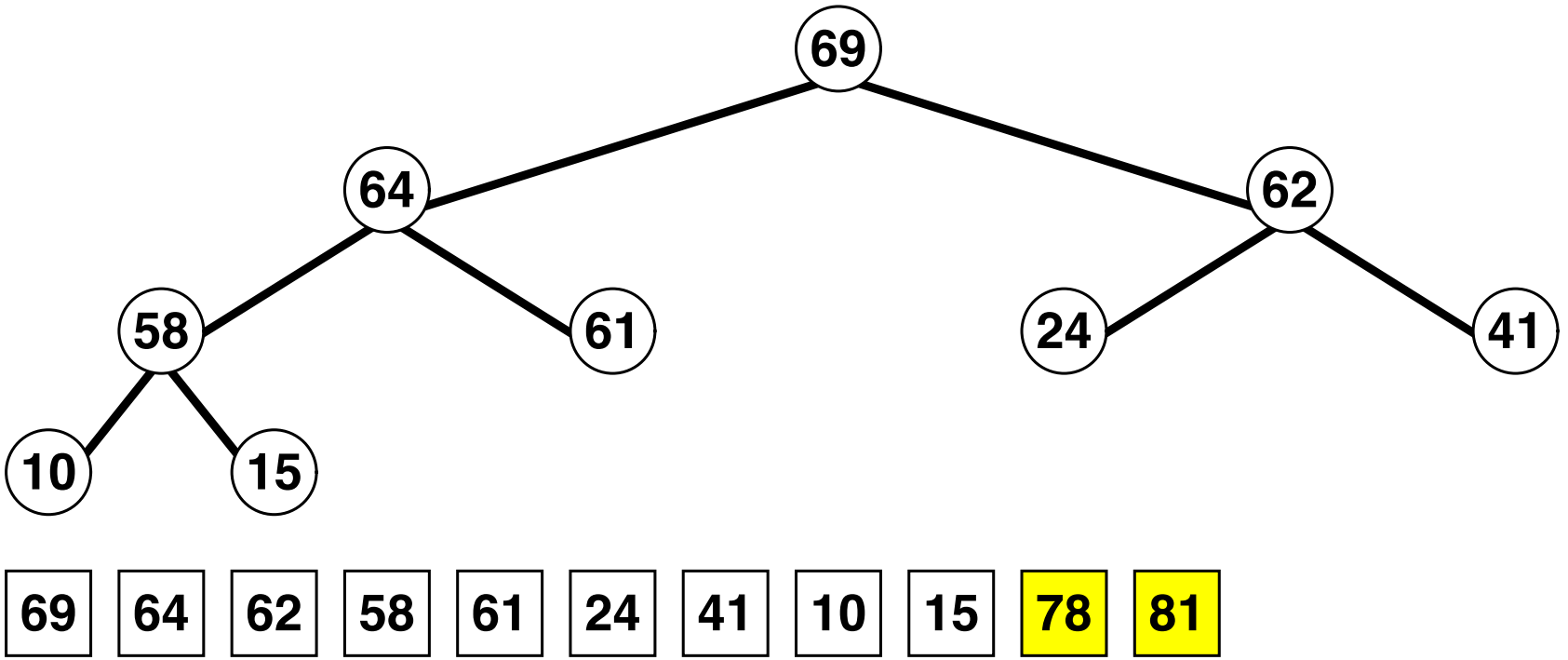
HeapSort



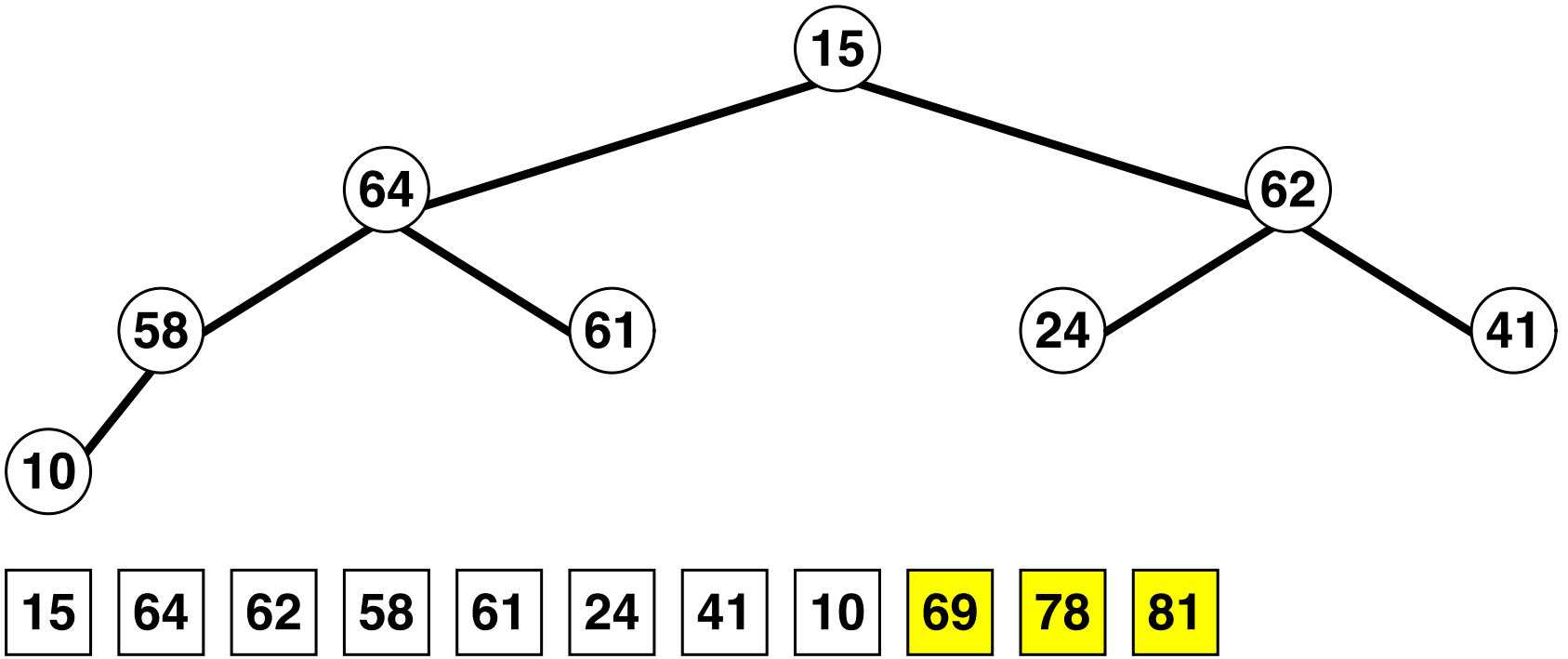
HeapSort



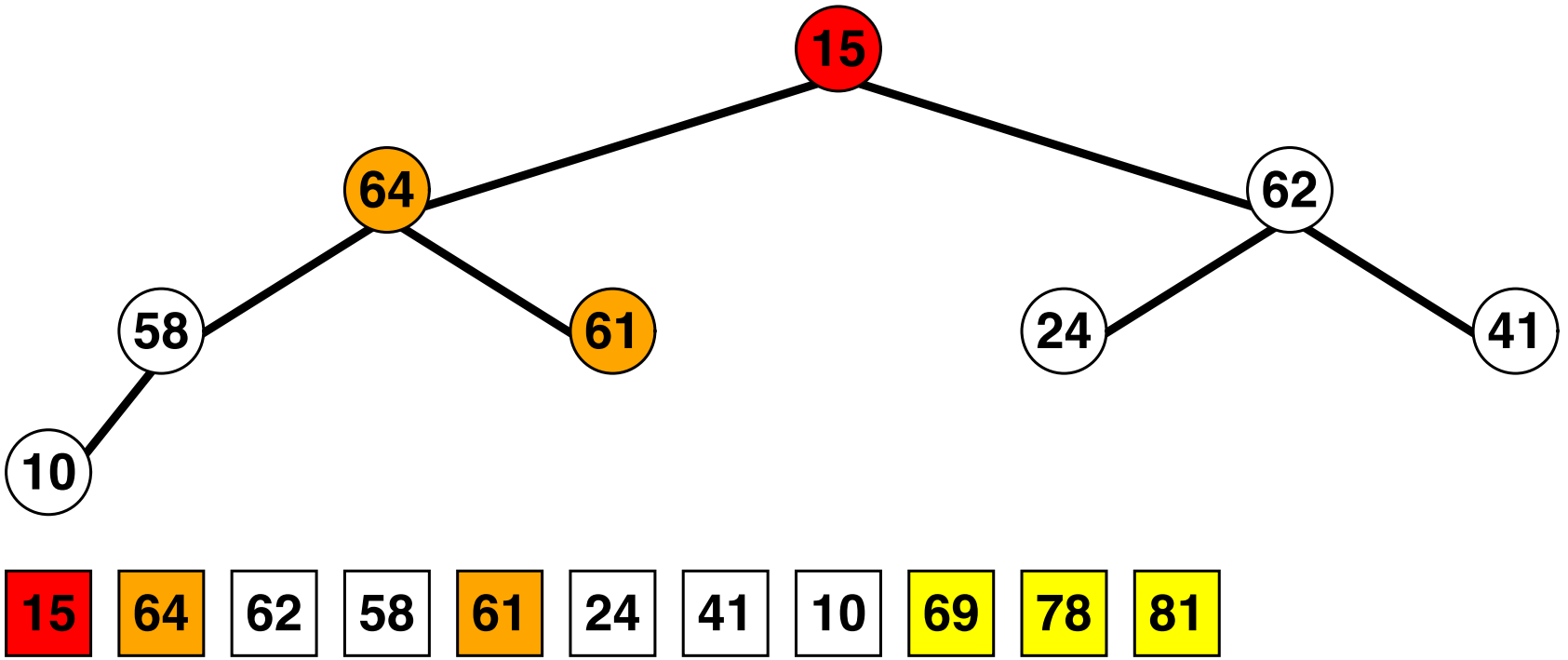
HeapSort



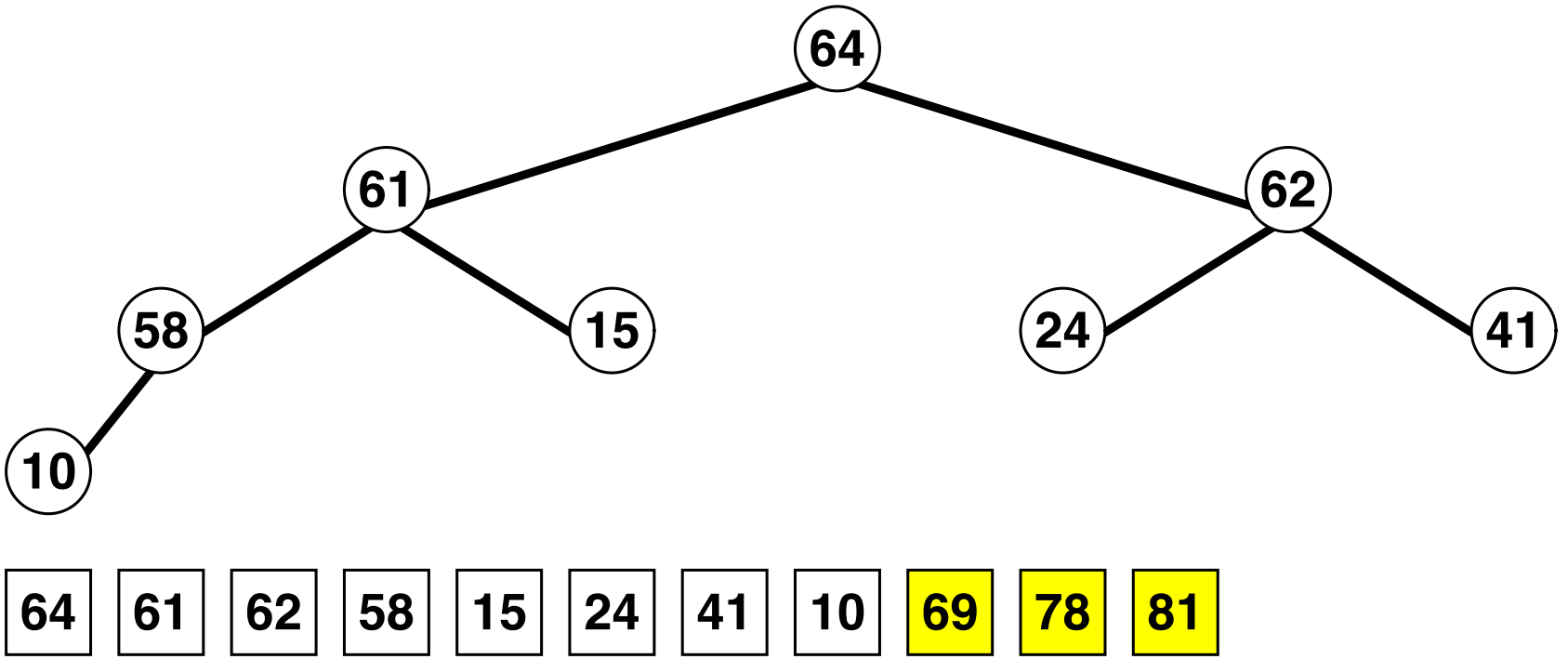
HeapSort



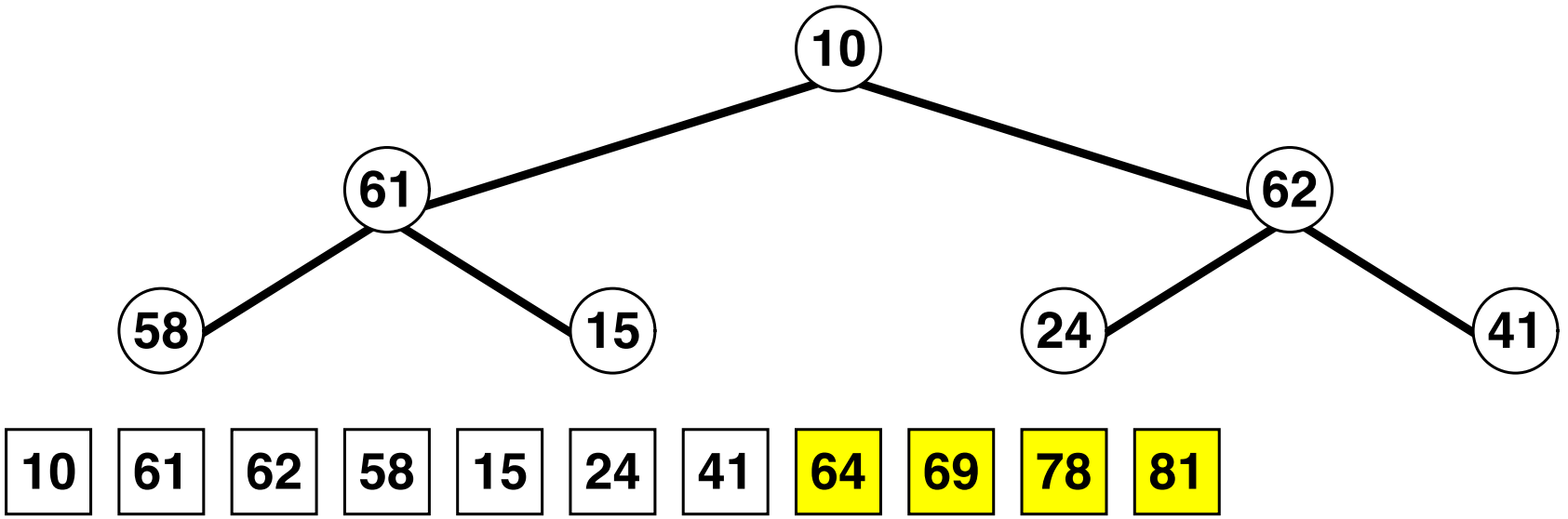
HeapSort



HeapSort



HeapSort



HeapSort

Roteiro

Referências

- N. Ziviani, Projeto de Algoritmos, Thomson, 2004.
- T. Cormem, C. E. Leiserson, R. L. Rivest, C. Stein, Algoritmos - Teoria e Prática, Campus, 2002.
- R. Sedgewick, Algorithms in C, Parts 1-4, Addison-Wesley, 2009.