

◀Linked Lists▶ e ▶Heap▶

Kaio Christaldo
Fabricio Matsunaga

◀Linked Lists▶

Apresentação Problema Motivador

beecrowd | 2479



Ordenando a Lista de Crianças do Papai Noel

Por Cristian J. Ambrosi, URI  Brazil

Timelimit: 1

Papai Noel está nos preparativos finais para a entrega dos presentes para as crianças do mundo todo pois o natal está chegando mais uma vez. Olhando suas novas listas de crianças que irão ganhar presentes neste ano ele percebeu que o duende estagiário (que havia ficado responsável por fazer as listas) não havia colocado os nomes em ordem alfabética.

Como o Papai Noel é um homem muito organizado ele deseja que cada lista de crianças possua, no seu final, o total de crianças que foram bem comportadas neste ano e um total das que não foram. Assim ele pode comparar a quantidade de crianças que se comportam este ano com as dos anos anteriores.

Para ajudar o bom velhinho, seu dever é criar um programa que leia todos os nomes da lista e imprima os mesmos nomes em ordem alfabética. No final da lista, você deve imprimir o total de crianças que foram e não foram comportadas neste ano.

Entrada

A entrada é composta por vários nomes. O primeiro valor **N** ($0 \leq N \leq 100$), indica quantos nomes tem na lista. As **N** linhas seguintes, contem um caracter especial correspondente ao comportamento da criança (+ indica que a criança foi bem comportada, - indica que a criança não foi bem comportada). Após o caracter especial, segue o nome da criança com no máximo 20 caracteres.

Saída

Para cada lista de crianças, você deve imprimir os nomes em ordem alfabética. Após imprimir os nomes das crianças, você deve mostrar o total de crianças que se comportaram bem ou mal durante o ano.

2479 – Ordenando a Lista de Crianças do Papai Noel

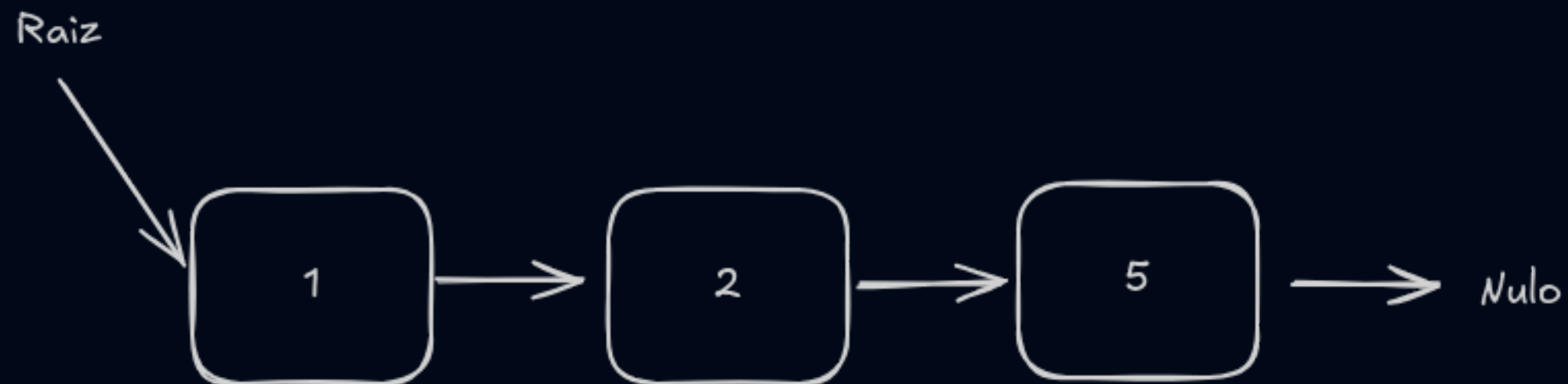
◀Linked Lists▶

- **Definição:** São estruturas de dados fundamentais, que consiste em uma sequência de elementos, chamados de **nós** que não seguem uma alocação sequencial na memória, diferente de um vetor.
- Também conhecidas como Listas Encadeadas, são elas:
 - Lista Simplesmente Encadeada
 - Lista Duplamente Encadeada
 - Lista Circular
 - Lista Duplamente Circular
 - Todas elas podem ser implementadas com **Nó** cabeça ou não.

◀Linked Lists▶

Lista Simplesmente Encadeada

Sem nó cabeça



Com nó cabeça



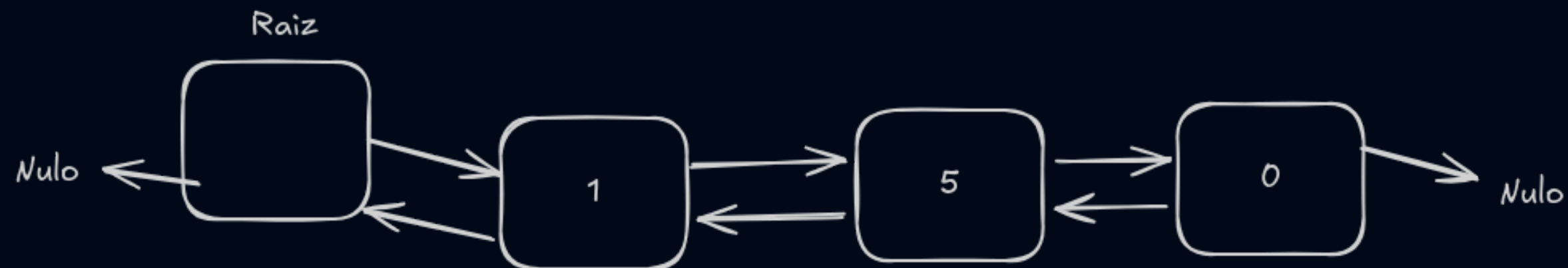
◀Linked Lists▶

Lista Duplamente Encadeada

Sem nó cabeça

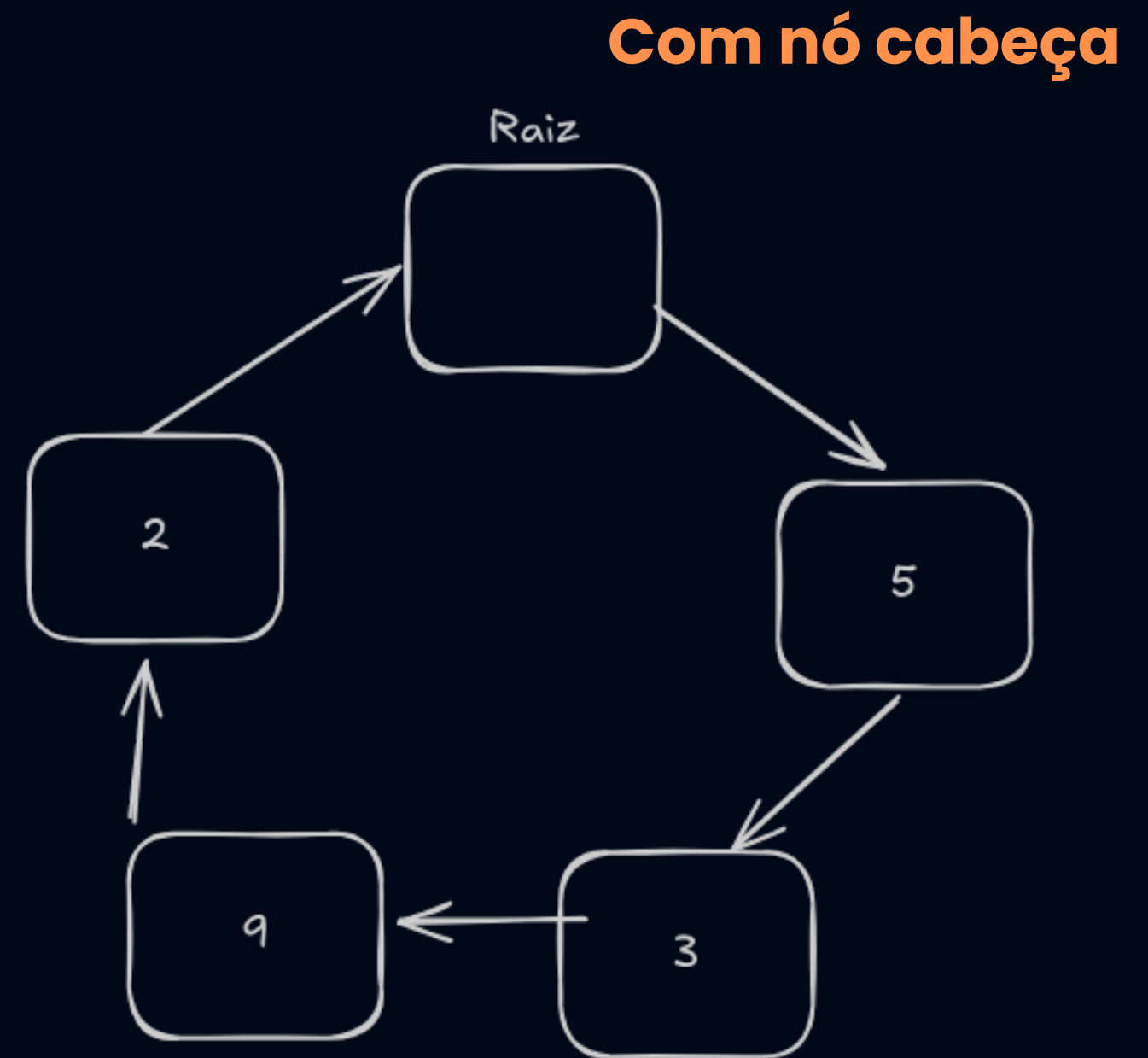
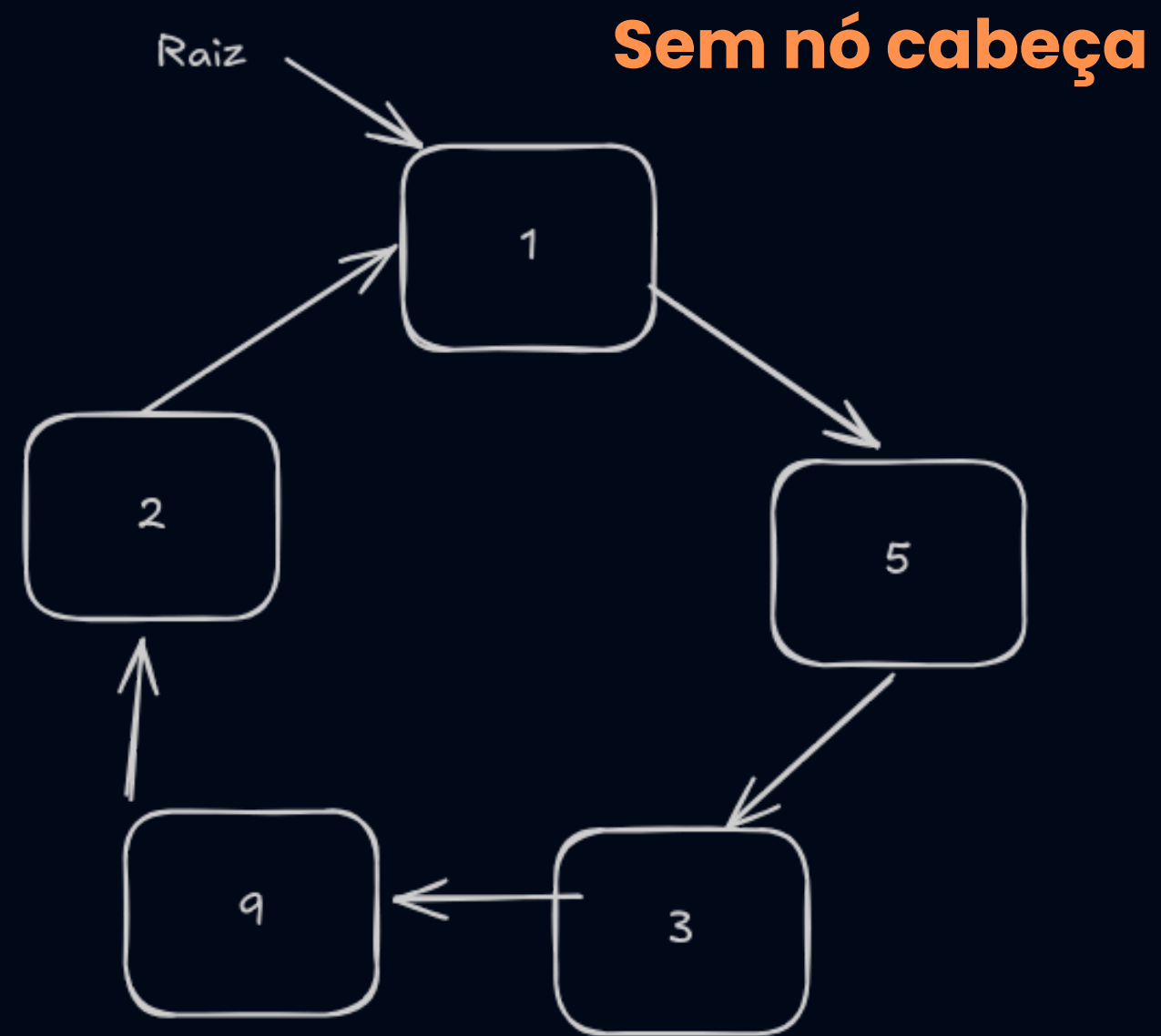


Com nó cabeça



◀Linked Lists▶

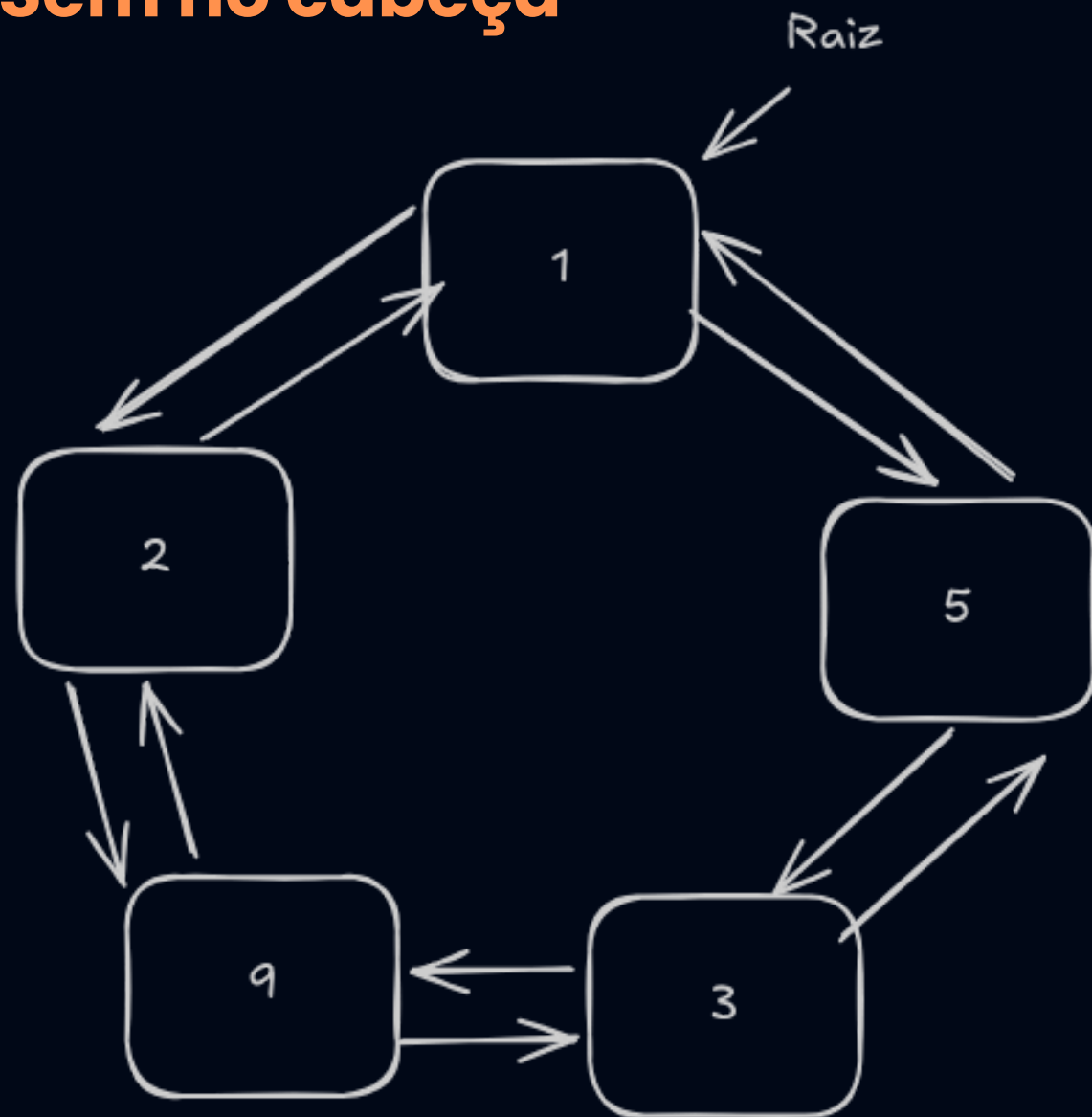
Lista Circular



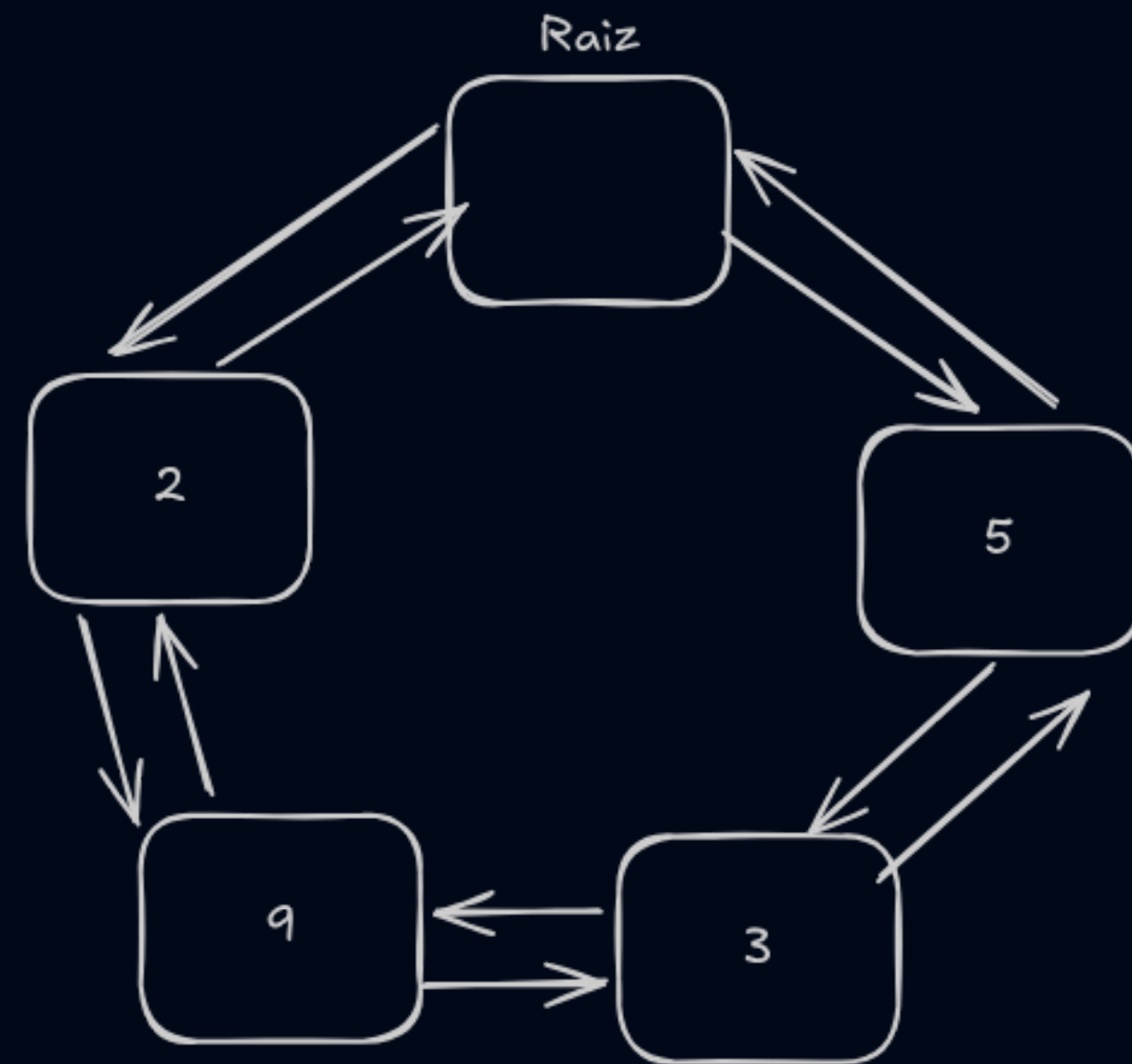
◀Linked Lists▶

Lista Duplamente Circular

Sem nó cabeça



Com nó cabeça



Template <List>

Template <list>

- **Definição:** Uma Lista (**list**) é uma estrutura de dados que oferece operações eficientes para inserção e remoção de elementos em qualquer posição da lista.
 - O template **list** no C++ é baseado em uma **Lista Duplamente Encadeada**
 - Inserção e Remoção no começo e no final: Tempo $O(1)$.
 - Inserção/remover em qualquer posição: Tempo $O(n)$
 - Acesso aos elementos: Acesso direto por índice não é eficiente, é necessário percorrer a lista até o elemento desejado (tempo $O(n)$).
- **Cabeçalho necessário:**



```
1  #include <iostream>
2  #include <list>
3
4  using namespace std;
```

Template <list>

Operações na List

Principais Operações Usadas:

- **`l.push_back(val)`** – Insere um elemento no final da lista – **$O(1)$**
- **`l.push_front(val)`** – Insere um elemento no início da lista – **$O(1)$**
- **`l.pop_back()`** – Remove o último elemento – **$O(1)$**
- **`l.pop_front()`** – Remove o primeiro elemento – **$O(1)$**
- **`l.insert(it, val)`** – Insere **`val`** antes do iterador **`it`** – **$O(1)$**
- **`l.erase(it)`** – Remove o elemento na posição **`it`** – **$O(1)$**
- **`l.begin()` / `l.end()`** – Retorna iterador para o **início/fim** da lista – **$O(1)$**
- **`l.rbegin()` / `l.rend()`** – Iteradores reversos – **$O(1)$**
- **`l.splice(it, L2)`** – move todos os elementos de **`L2`** antes do **`it`** – **$O(1)$**

Template <list>

Operações na List

Principais Operações Usadas:

- **l.size()** – Retorna o número de elementos na lista – **$O(1)$**
- **l.empty()** – Verifica se a lista está vazia – **$O(1)$**
- **l.clear()** – Remove todos os elementos da lista – **$O(n)$**
- **l.remove(val)** – Remove todos os elementos iguais a val – **$O(n)$**
- **l.sort()** – Ordena os elementos da lista (por padrão em ordem crescente) – **$O(n \log n)$**
- **l.reverse()** – Inverte a ordem dos elementos – **$O(n)$**
- **l.merge(l2)** – Junta duas listas ordenadas – **$O(n)$**
- **l.unique()** – Remove elementos consecutivos duplicados (lista deve estar ordenada) – **$O(n)$**
- **l.swap(l2)** – Troca os elementos com outra lista – **$O(1)$**

Template <list>

- `push_back(val)` - $O(1)$



```
1  list<int> lista; // Declara uma lista de inteiros
2
3  lista.push_back(10);
4  lista.push_back(20);
5  // 10 20
```

Template <list>

- `push_front(val)` – $O(1)$



```
1  list<int> lista; // Declara uma lista de inteiros
2
3  lista.push_front(10);
4  lista.push_front(20);
5  lista.push_front(30);
6  // 30 20 10
```

Template <list>

- `pop_back()` - $O(1)$



```
1  list<int> lista = { 10, 20, 30 }; // Declara uma lista com inteiros
2
3  lista.pop_back();
4  lista.pop_back();
5  // 10
```

Template <list>

- `pop_front()` - $O(1)$



```
1  list<int> lista = { 10, 20, 30 }; // Declara uma lista com inteiros
2
3  lista.pop_front();
4  lista.pop_front();
5  // 30
```


Template `<list>`

- `insert(it, val)` - $O(1)$



```
1  list<int> lista = { 10, 20, 30 }; // Declara uma lista com inteiros
2
3  auto it = --lista.end(); // Retorna o iterator para 30
4  lista.insert(it, 25); // insere 25 entre 20 e 30
5
6  // 10 20 25 30
```

Template <list>

- `begin() / end()` e `rbegin() / rend()`



```
1  list<int> lista = { 10, 20, 30 }; // Declara uma lista com inteiros
2
3  auto it1 = lista.begin(); // Retorna iterator para 10
4  auto it2 = lista.rbegin(); // Retorna iterator para 30 (reverso)
5  auto it3 = lista.end(); // Retorna iterator apos de 30 (inválido)
6  auto it4 = lista.rend(); // Retorna iterator antes de 10 (inválido)
```

Template `<list>`

- `splice(it, l2) - O(1)`



```
1  list<int> list1, list2;
2  list1 = {1, 2, 3}; list2 = {4, 5, 6};
3
4  auto it = list1.begin();
5  advance(it, 1); // aponta para o 2
6
7  for (int x : list1) cout << x << " ";
8  cout << endl;
9  list1.splice(it, list2); // move todos os elementos de list2 antes do 2
10
11 // list1: 1 4 5 6 2 3
12 // list2: vazia
13
14 for (int x : list1) cout << x << " ";
15 cout << endl;
```

Template <list>

- `size()`, `empty()`, `size()` - $O(1)$



```
1  list<int> lista = { 10, 20, 30 }; // Declara uma lista com inteiros
2
3  cout << "Lista tinha: " << lista.size() << " elementos"; // 3
4
5  if (!lista.empty()) // se a lista tiver elementos
6      lista.clear(); // esvaziar a lista
7
8  cout << ", agora está Vazia!\n";
```

Template <list>

- **remove(val) - O(n)**



```
1 list<int> lista = { 10, 20, 30 }; // Declara uma lista com inteiros
2
3 lista.remove(10);
4
5 for (auto& i : lista) cout << i << " "; // Imprime a lista
6 cout << "\n"; // 20 30
```

Template <list>

- `sort()` – $O(n \log n)$



```
1 list<int> lista = { 10, 4, 3, 1, 7, 13 }; // Declara uma lista com inteiros
2
3 lista.sort(); // Ordena em ordem crescente
4 for (auto& i : lista) cout << i << " "; // 1 3 4 7 10 13
5
6 lista.sort(descrescente); // passando função
7 for (auto& i : lista) cout << i << " "; // 13 10 7 4 3 1
```



```
1 bool descrescente(int a, int b) {
2     return a > b;
3 }
```

Template <list>


- `reverse()`, `merge(list2)` e `unique()` – $O(n)$



```
1  list<int> list1, list2; // Declara das listas
2
3  list1 = {7, 6, 5, 4}; // Ordem decrescente
4  list2 = {1, 2, 3, 4, 5}; // Ordem crescente
5
6  list1.reverse(); // transforma em ordem crescente
7
8  list1.merge(list2); // Juntas listas em list1 e mantem a ordem crescente
9  // list2 fica vazia.
10
11 list1.unique(); // Remove as duplicatas.
12
13 for (auto& i : list1) cout << i << " "; // 1 2 3 4 5 6 7
```

Template <list>

- `swap(list2) - O(1)`



```
1  list<int> list1, list2; // Declara as listas
2
3  list1 = {7, 6, 5, 4}; // Ordem decrescente
4  list2 = {1, 2, 3, 4, 5}; // Ordem crescente
5
6  list1.swap(list2);
7
8  for (auto& i : list1) cout << i << " "; // 1 2 3 4 5
9  cout << "\n";
10 for (auto& i : list2) cout << i << " "; // 7 6 5 4
```


Template `<list>`

Vantagens

- inserir ou remover elementos em qualquer posição (dado o iterador) é $O(1)$, sem realocação ou cópia.
- O `.sort()` mantém a ordem relativa de elementos iguais.
- Métodos como `.splice()`, `.merge()`, `.remove()`, `.unique()` e `.sort()` são otimizados para listas.

Desvantagens

- Sem acesso direto a elementos (sem `[]` ou `.at()`)
- Alto overhead de memória
 - Cada elemento ocupa mais espaço, pois guarda dois ponteiros (`prev` e `next`).
- Mais lenta que vector em muitos casos
 - Para percorrer, ordenar ou processar em lote, um vector é geralmente mais rápido por causa da localidade de memória.

Template **<list>**

Alternativas

- **vector** ou **deque**
- Lista simplesmente encadeada (**<forward_list>**)
 - Operações: push_front, pop_front, insert_after, erase_after, sort, reverse, remove, merge, clear, empty

Resolução do Problema Motivador

2479 – Ordenando a Lista de Crianças do Papai Noel

Dicas:

- Criar duas listas
- Ler como operador(char) e nome(string)
- Usar merge
- Usar Sort


A resolução estará disponível no Drive. Tente resolver por conta própria e, se precisar, compare com a solução! 😊

◀Heap▶

Apresentação Problema Motivador

beecrowd | 2065

Fila do Supermercado

Por Cristhian Bonilha, UTFPR  Brazil

Timelimit: 1

Hoje é a inauguração de um grande supermercado em sua cidade, e todos estão muito excitados com os baixos preços prometidos.

Este supermercado tem **N** funcionários que trabalham no caixa, identificados por números de 1 a **N**, onde cada funcionário leva um determinado tempo **v_i** para processar um item de um cliente. Ou seja, se um cliente tem **c_j** itens em sua cesta, um determinado funcionário levará **v_i*c_j** segundos para processar todos os itens deste cliente.

Quando um cliente entra na fila para ser atendido ele espera até que um funcionário esteja livre para o atendê-lo. Se mais de um funcionário estiverem livres ao mesmo tempo, o cliente será atendido pelo funcionário de menor número de identificação. Tal funcionário só estará livre novamente após processar todos os itens deste cliente.

Há **M** clientes na fila para serem atendidos, cada um com um determinado número de itens na sua cesta. Dadas as informações sobre os funcionários nos caixas e os clientes, o gerente pediu sua ajuda para descobrir quanto tempo levará para que todos os clientes sejam atendidos.

Entrada

A primeira linha conterá dois inteiros **N** e **M**, indicando o número de funcionários no caixa e o número de clientes, respectivamente ($1 \leq N \leq M \leq 10^4$).

Em seguida haverá **N** inteiros **v_i**, indicando quanto tempo leva para o **i**-ésimo funcionário processar um item ($1 \leq v_i \leq 100$, para todo $1 \leq i \leq N$).

Em seguida haverá **M** inteiros **c_j**, indicando quantos itens o **j**-ésimo cliente tem em sua cesta ($1 \leq c_j \leq 100$, para todo $1 \leq j \leq M$).

Saída

Imprima uma linha contendo um inteiro, indicando quanto tempo levará para que todos os clientes sejam atendidos.

Exemplos de Entrada	Exemplos de Saída
1 1 3 6	18
1 2 1 5 3	8
2 3 1 2 10 5 3	13

2065 – Fila do Supermercado

◀Heap▶ – Listas de prioridade

pode-se definir lista de prioridades como uma tabela na qual a cada um de seus dados está associada uma prioridade. Essa prioridade é, em geral, definida através de um valor numérico e armazenada em algum de seus campos.

- *seleção* do elemento de maior prioridade;
- *inserção* de um novo elemento;
- *remoção* do elemento de maior prioridade.

Implementação por lista não ordenada

Implementação por lista ordenada

Implementação por “heap”

◀Heap▶ – Implementação por lista não ordenada

- seleção: $O(n)$
- inserção: $O(1)$
- remoção: $O(n)$
- alteração: $O(n)$
- construção: $O(n)$

◀Heap▶ – Implementação por lista ordenada

- seleção: $O(1)$
- inserção: $O(n)$
- remoção: $O(1)$
- alteração: $O(n)$
- construção: $O(n \log n)$
-

◀Heap▶ – Implementação por “heap”

$$s_i \leq s_{\lfloor i/2 \rfloor} \quad \text{para } 1 < i \leq n.$$

95 60 78 39 28 66 70 33

FIGURA 6.1 Lista de prioridades.

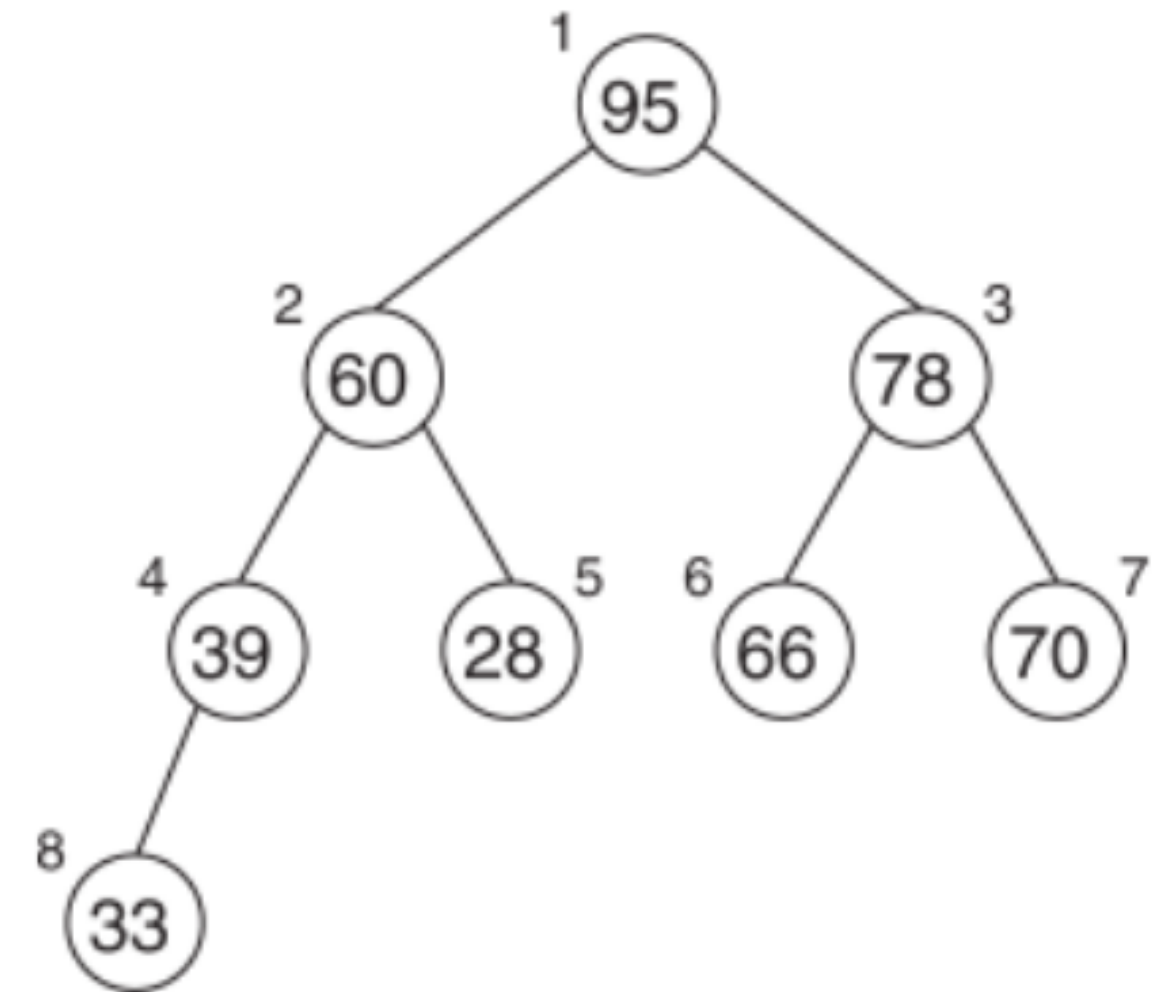


FIGURA 6.2 Representação gráfica de um heap.

◀Heap▶ – Implementação por “heap”

- seleção: $O(1)$
- inserção: $O(\log n)$
- remoção: $O(\log n)$
- alteração: $O(\log n)$
- construção: $O(n)$

◀Heap▶ – Alteração de Prioridades

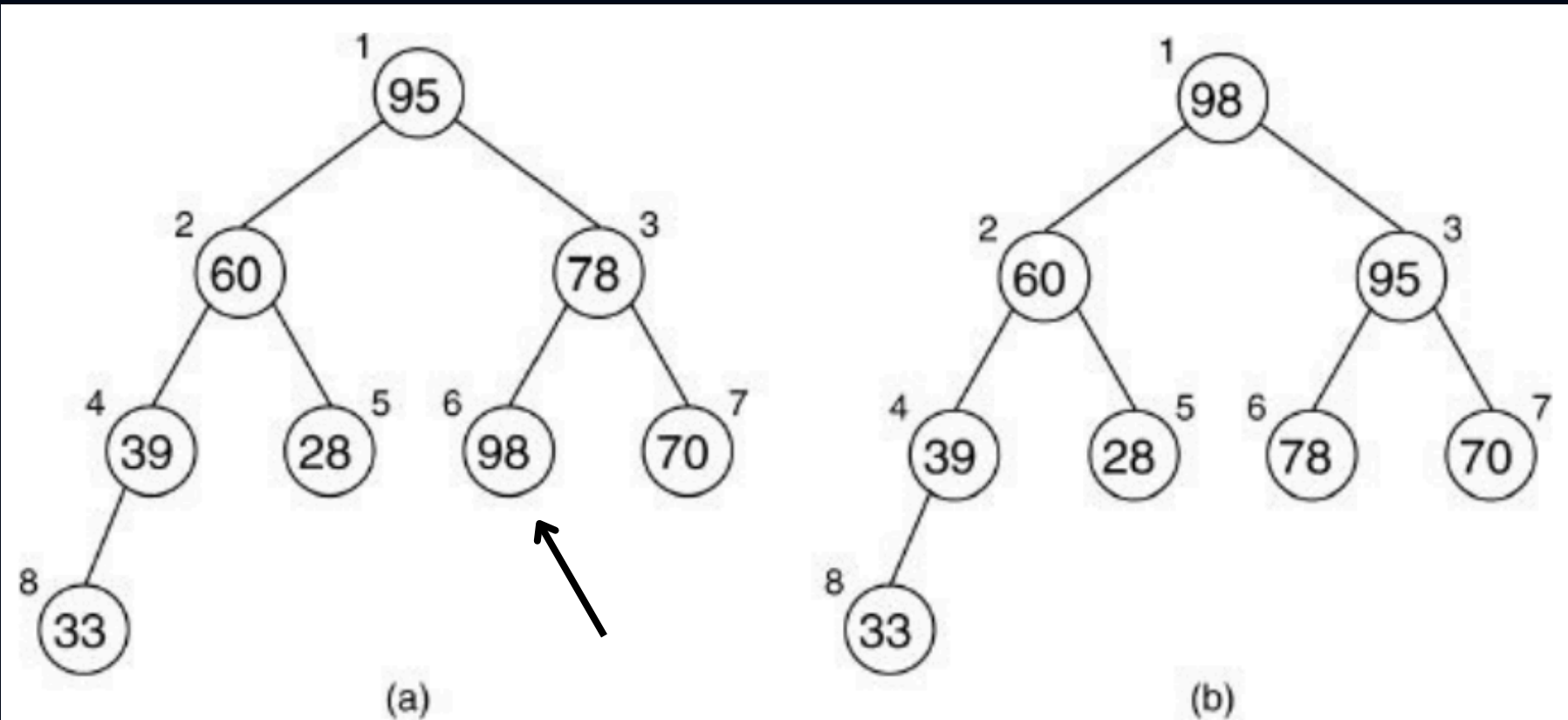


FIGURA 6.3 Aumento da prioridade de um nó.

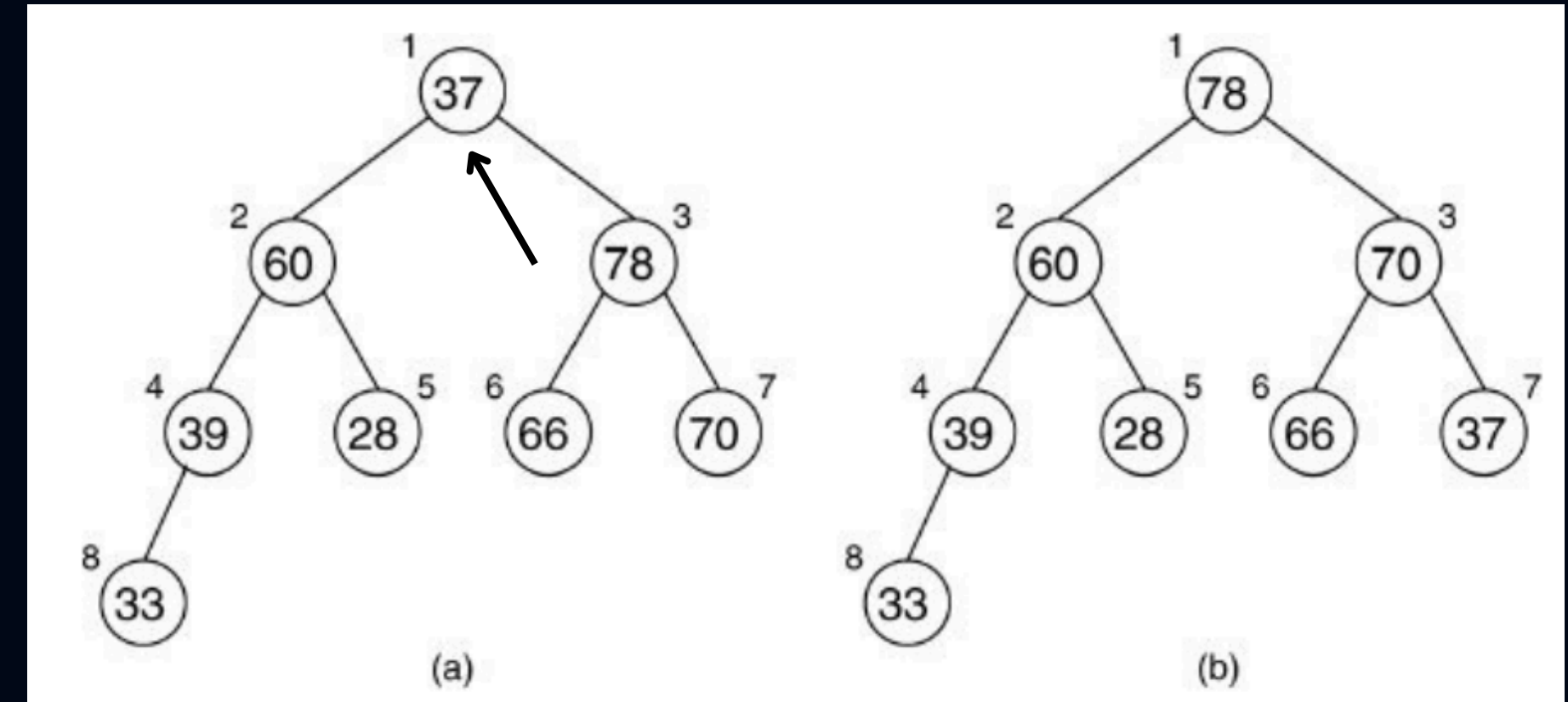


FIGURA 6.4 Diminuição da prioridade de um nó.

◀Heap▶ – Inserção e Remoção

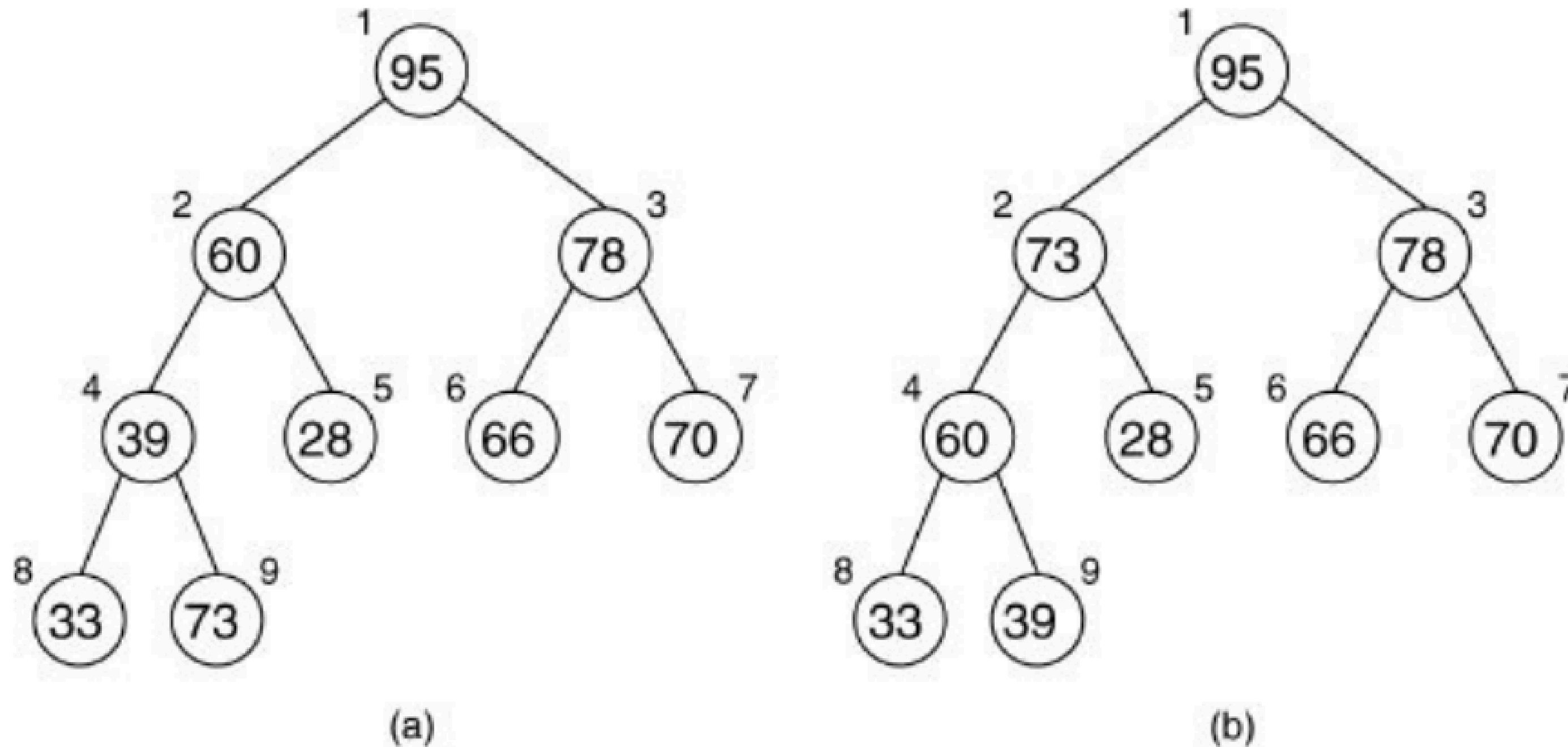


FIGURA 6.5 Inserção de um elemento.

◀Heap▶ – Inserção e Remoção

Estruturas de Dados e Seus Algoritmos, 3ª Edição

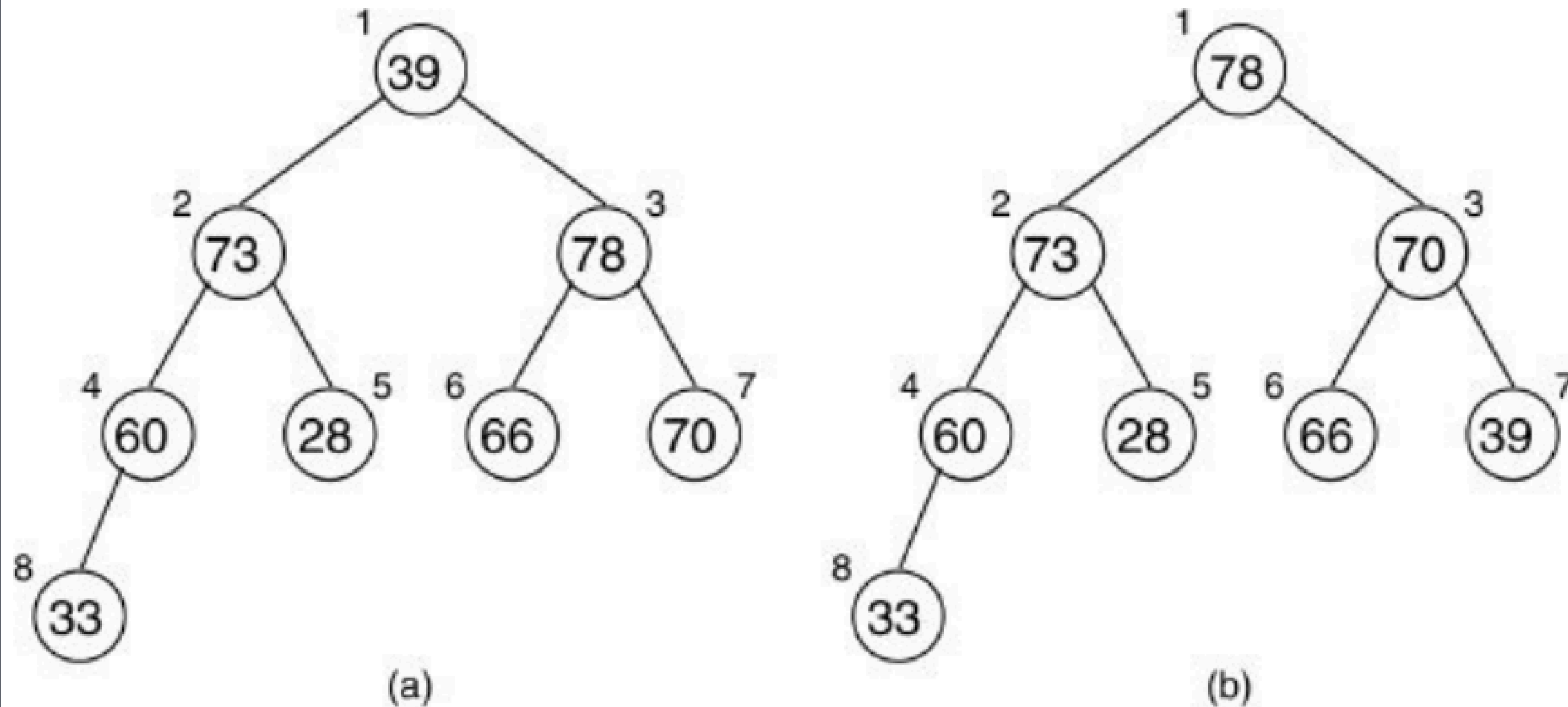


FIGURA 6.6 Remoção de um elemento.

◀Heap▶ – Heap no c++

1. **make_heap()**: Converte um intervalo em um heap.
2. **push_heap()**: Organiza o heap depois de uma inserção no final.
3. **pop_heap()**: Move o maior elemento no final para exclusão.
4. **sort_heap()**: Organiza os elementos do heap em ordem crescente.
5. **is_heap()**: Verifica se um dado intervalo é um heap.
6. **is_heap_until()**: Retorna o maior sub-intervalo que é um heap

◀Heap▶ – make_heap()



```
1  vector<int> v1 = {20,30,40,25,15};  
2  
3  make_heap(v1.begin(),v1.end());  
4  
5  cout << "The maximum element of heap is : ";  
6  cout << v1.front() << endl;  
7  
8  return 0;
```

◀Heap▶ – push_heap()



```
1  vector<int> vc{ 20, 30, 40, 10 };
2
3  make_heap(vc.begin(), vc.end());
4  cout << "Initial Heap: ";
5  print(vc);
6
7  vc.push_back(50);
8  cout << "Heap just after push_back(): ";
9  print(vc);
10 push_heap(vc.begin(), vc.end());
11
12 cout << "Heap after push_heap(): ";
13 print(vc);
14
```



```
1  void print(vector<int>& vc)
2  {
3      for (auto i : vc) {
4          cout << i << " ";
5      }
6      cout << endl;
7  }
```


◀Heap▶ – pop_heap()

```
1 // initial vector
2 vector<int> vc{ 40, 10, 20, 50, 30 };
3
4 // making heap
5 make_heap(vc.begin(), vc.end());
6 cout << "Initial Heap: ";
7 print(vc);
8
9 // using pop_heap() function to move the largest element
10 // to the end
11 pop_heap(vc.begin(), vc.end());
12 cout << "Heap after pop_heap(): ";
13 print(vc);
14
15 // actually removing the element from the heap using
16 // pop_back()
17 vc.pop_back();
18 cout << "Heap after pop_back(): ";
19 print(vc);
```

```
1 void print(vector<int>& vc)
2 {
3     for (auto i : vc) {
4         cout << i << " ";
5     }
6     cout << endl;
7 }
```

◀Heap▶ – sort_heap()

```
1 // Initializing a vector
2 vector<int> v1 = { 20, 30, 40, 25, 15 };
3
4 // Converting vector into a heap
5 // using make_heap()
6 make_heap(v1.begin(), v1.end());
7
8 // Displaying heap elements
9 cout << "The heap elements are: ";
10 for (int& x : v1)
11     cout << x << " ";
12 cout << endl;
13
14 // sorting heap using sort_heap()
15 sort_heap(v1.begin(), v1.end());
16
17 // Displaying heap elements
18 cout << "The heap elements after sorting are: ";
19 for (int& x : v1)
20     cout << x << " ";
```

◀Heap▶ – is_heap() e is_heap_until()



```
1 // Initializing a vector
2 vector<int> v1 = { 40, 30, 25, 35, 15 };
3
4 // Declaring heap iterator
5 vector<int>::iterator it1;
6
7 // Checking if container is heap
8 // using is_heap()
9 is_heap(v1.begin(), v1.end())
10     ? cout << "The container is heap "
11     : cout << "The container is not heap"; // ternary operator
12 cout << endl;
13
14 // using is_heap_until() to check position
15 // till which container is heap
16 auto it = is_heap_until(v1.begin(), v1.end());
17
18 // Displaying heap range elements
19 cout << "The heap elements in container are : ";
20 for (it1 = v1.begin(); it1 != it; it1++)
21     cout << *it1 << " ";
22
```

Resolução do Problema Motivador

A resolução estará disponível no Drive. Tente resolver por conta própria e, se precisar, compare com a solução! 😊

Lista de Exercícios

2479 – Ordenando a Lista de Crianças do Papai Noel



Se tiver alguma dúvida ou dificuldade na resolução de algum exercício, sinta-se à vontade para perguntar! 😊

Referências

[1] CPLUSPLUS.COM. C++ list. Disponível em: <https://cplusplus.com/reference/list/list/>. Acesso em: 08 mai. 2025.

[1] CPLUSPLUS.COM. C++ forward_list. Disponível em: https://cplusplus.com/reference/forward_list/forward_list/. Acesso em: 08 mai. 2025.

GEEKSFORGEEKS. Heap em C++ STL. GeeksforGeeks, [S. l.], [s. d.]. Disponível em: https://www-geeksforgeeks-org.translate.goog/cpp-stl-heap/?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em: 8 maio 2025.