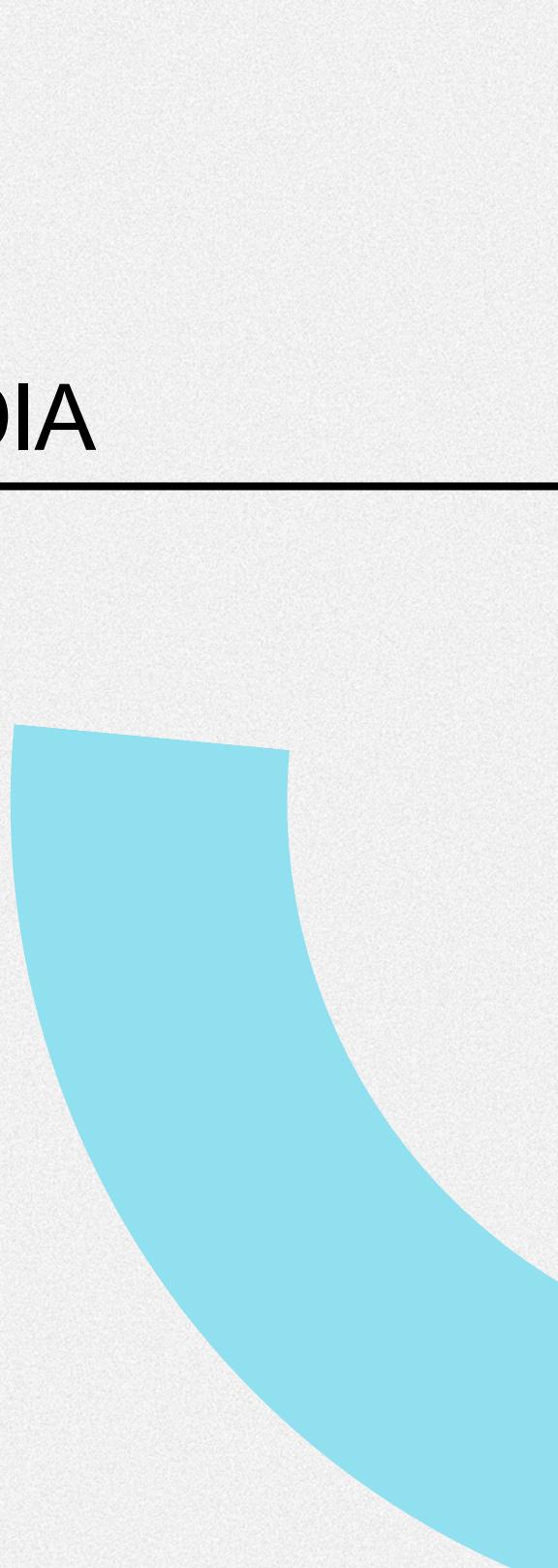




UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Estruturas Intermediárias



SET, MAP, PRIORITY QUEUE, UNION-FIND

Aluno(s): Geovanna David Gonzaga - 12021BSI232
Joao Paulo Marques Ribeiro - 12021BSI208
Sara Batista Pereira - 12211bsi233

índice

04 SET

06 MAP

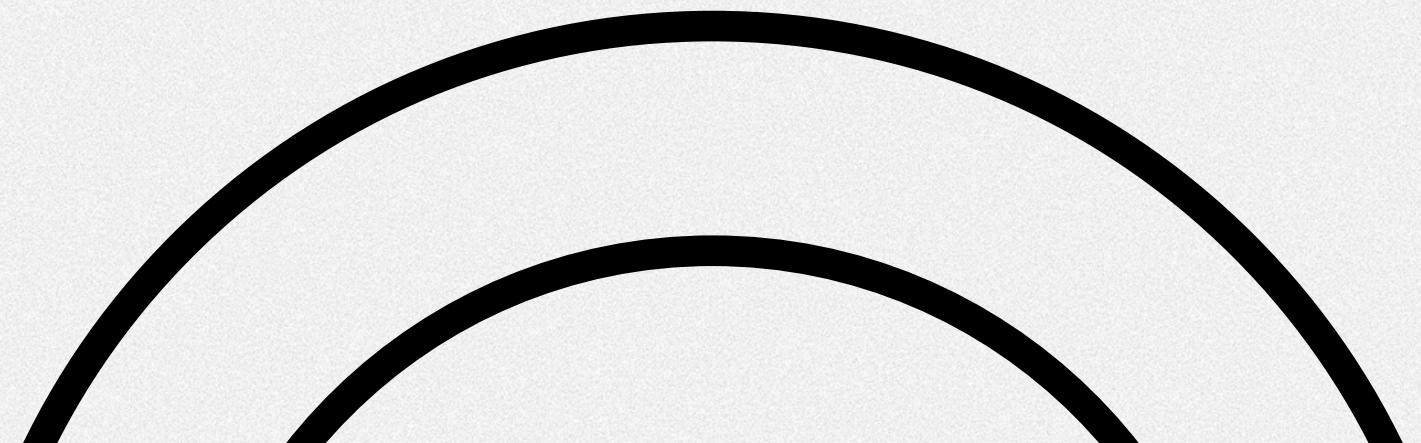
08 PRIORITY QUEUE

16 UNION-FIND



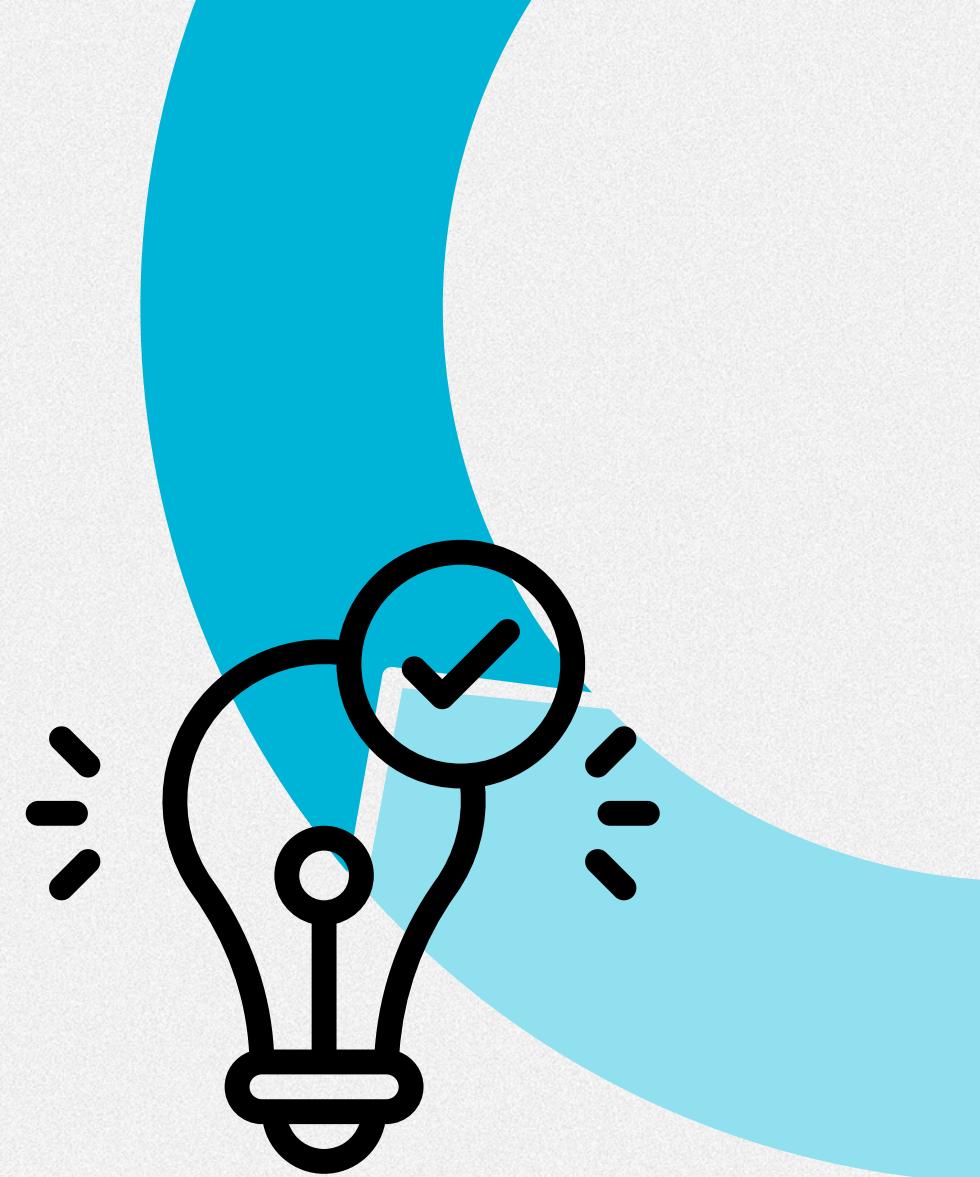


Set e Map



SET (Conjunto)

- O Set é uma estrutura de dados que não permite elementos duplicados e geralmente oferece operações eficientes para inserção, remoção e busca.
- **Características:**
 - Armazena valores únicos (não há elementos repetidos).
 - Normalmente implementado como uma tabela hash ou árvore balanceada.
 - Operações rápidas de inserção, remoção e verificação de existência.
 - Permite operações de conjuntos matemáticos, como união, interseção e diferença.



Introdução

SET

PYTHON (SET)

```
# Criando um conjunto
meu_set = {1, 2, 3, 4, 5}

# Adicionando um elemento
meu_set.add(6)

# Removendo um elemento
meu_set.remove(3) # Erro se não existir
meu_set.discard(10) # Não gera erro se o elemento não existir

# Verificando a existência
print(4 in meu_set) # True

# Operações de conjunto
outro_set = {4, 5, 6, 7}
print(meu_set.union(outro_set)) # União
print(meu_set.intersection(outro_set)) # Interseção
print(meu_set.difference(outro_set)) # Diferença
```



MAP

(DICIONÁRIO OU TABELA HASH)

- O Map é uma estrutura de dados que armazena pares chave-valor, permitindo acesso eficiente aos valores por meio de suas chaves. remoção e busca.
- **Características:**
 - Chaves únicas (cada chave aparece no máximo uma vez).
 - Acesso rápido a valores através da chave.
- **Implementado como:**
 - Tabela Hash (unordered_map em C++, dict em Python) → $O(1)$ no melhor caso.
 - Árvore balanceada (std::map em C++) → $O(\log n)$.



Introdução

MAP

PYTHON (DICT)

```
# Criando um dicionário (map)
meu_map = {"nome": "Ana", "idade": 30, "cidade": "São Paulo"}

# Acessando valores
print(meu_map["nome"]) # Ana

# Adicionando e modificando elementos
meu_map["idade"] = 31
meu_map["profissão"] = "Engenheira"

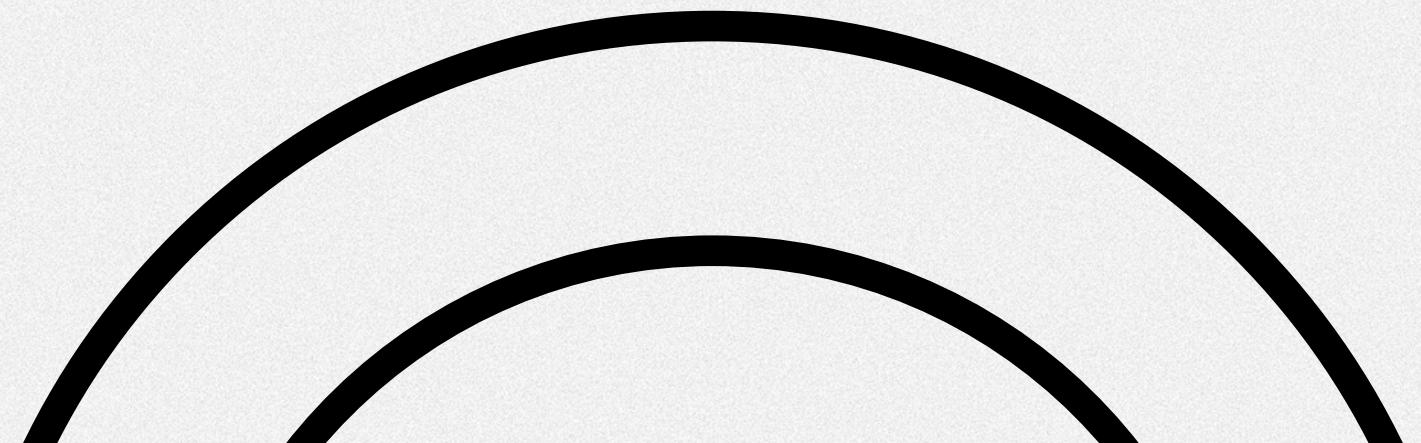
# Removendo elementos
del meu_map["cidade"]

# Verificando a existência de uma chave
print("idade" in meu_map) # True

# Iterando sobre um dicionário
for chave, valor in meu_map.items():
    print(f'{chave}: {valor}')
```

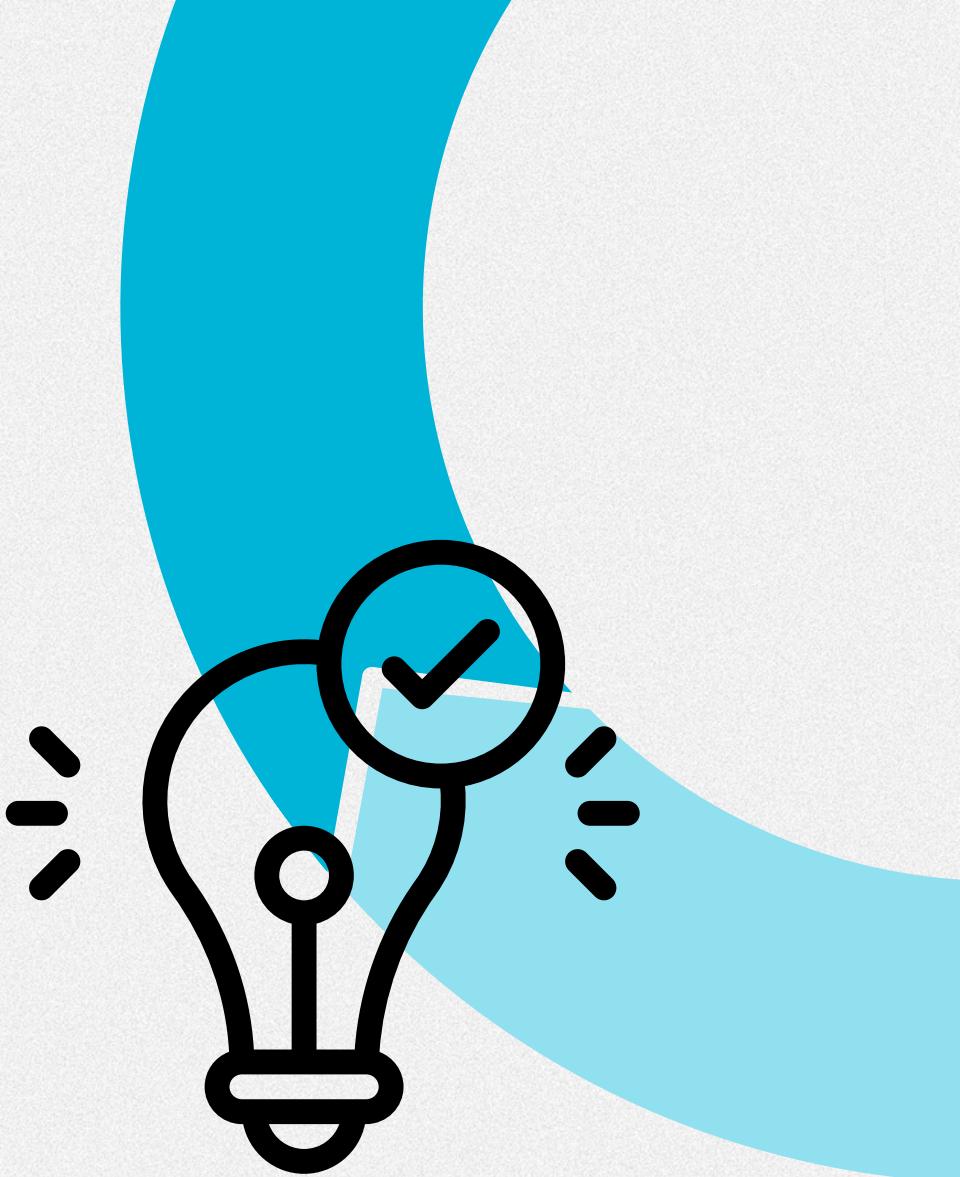


Priority Queue



Introdução

- A Fila de Prioridade (do inglês Priority Queue) é um Tipo Abstrato de Dado (TAD) que opera de forma similar a uma Fila.
- O TAD Fila tem comportamento FIFO (First-In, First-Out), onde o elemento de maior prioridade para sair da fila é o elemento que entrou primeiro na fila.
- O conceito de prioridade é explicitado nas Filas de Prioridade através de um valor numérico.
- Nesse caso, a lógica de prioridade pode operar pelo menor ou pelo maior valor, dependendo da aplicação.
- Diferente de uma fila convencional, a fila de prioridade segue a regra de maior prioridade primeiro.

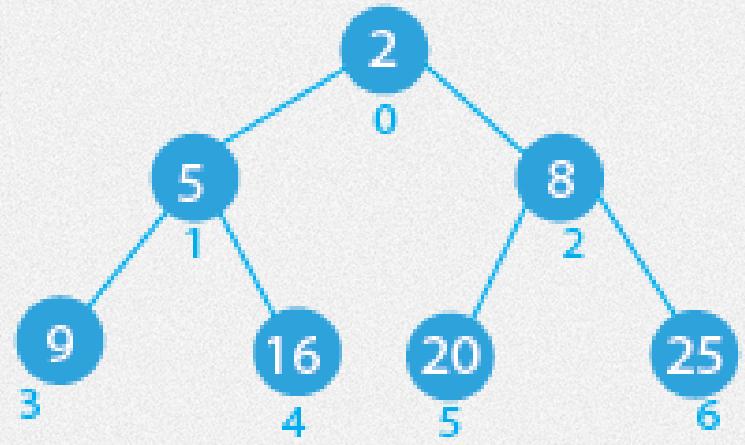


Conceito

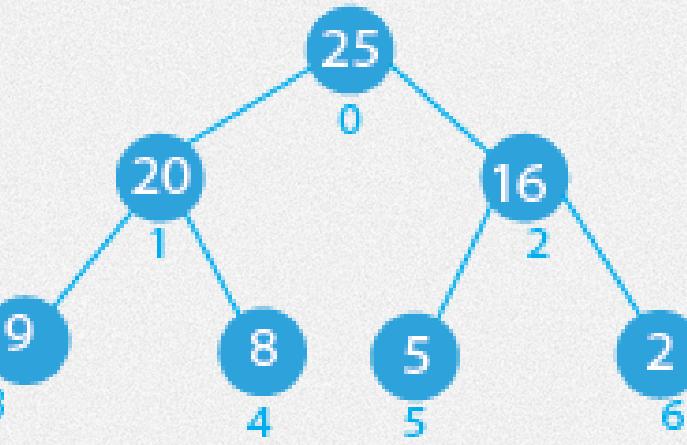
IMPLEMENTAÇÃO DE UMA FILA DE PRIORIDADE

- Exemplo cotidiano: em um supermercado, a fila do caixa preferencial segue o princípio da fila de prioridade, onde idosos e gestantes têm prioridade sobre outros clientes.
- Filas de Prioridade podem ser implementadas de diferentes formas, como Heap Binário, arrays, listas ligadas.
- Estruturas como Heap Máximo e Heap Mínimo são utilizadas para otimizar as operações de inserção e remoção.





Min Heap



Max Heap

Heap Binário

- O Heap Binário é uma árvore binária completa onde cada nó segue a propriedade de heap:
 - Max-Heap: o valor do nó pai é sempre maior ou igual ao valor dos filhos.
 - Min-Heap: o valor do nó pai é sempre menor ou igual ao valor dos filhos.
- As operações principais no heap são:
 - Inserção: um novo elemento é adicionado no final e ajustado para manter a propriedade do heap.
 - Remoção: o elemento de maior (ou menor) prioridade é removido e a estrutura é reorganizada.
- A complexidade das operações de inserção e remoção é $O(\log n)$, garantindo eficiência em aplicações que exigem ordenação rápida de prioridades.

Array

ESTRUTURA DE DADOS COM ARRAY

```
#define MAX 100

struct PriorityQueue {
    int arr[MAX]; // Array que armazena os elementos
    int size;     // Número de elementos presentes na fila de prioridade
};
```

A estrutura PriorityQueue contém:

- arr[MAX]: O array para armazenar os elementos da fila de prioridade.
- size: A variável que mantém o controle do número de elementos na fila.

Inserção

INSERÇÃO NA FILA DE PRIORIDADE (ENQUEUE, PUSH OU INSERT)

```
void insert(struct PriorityQueue *pq, int value) {
    if (pq->size >= MAX) return; // Se a fila estiver cheia, retorna
    pq->arr[pq->size++] = value; // Adiciona o elemento no final da fila
    int i = pq->size - 1;
    while (i > 0 && pq->arr[(i - 1) / 2] < pq->arr[i]) { // Verifica a ordem do heap
        int temp = pq->arr[i];
        pq->arr[i] = pq->arr[(i - 1) / 2];
        pq->arr[(i - 1) / 2] = temp;
        i = (i - 1) / 2; // Sobe o nó na árvore
    }
}
```

Função insert: Insere um novo elemento na fila de prioridade.

- Adiciona o valor ao final do array.
- O while organiza os elementos para garantir que a estrutura de heap seja mantida, movendo o novo valor para a posição correta.
- Em um Máx-Heap, a maior prioridade está no topo (início do array).

Remoção

REMOÇÃO DO ELEMENTO DE MAIOR PRIORIDADE (DEQUEUE MIN, POP MIN OU EXTRACT MIN)

```
int extractMax(struct PriorityQueue *pq) {
    if (pq->size <= 0) return -1; // Se não houver elementos, retorna -1
    int max = pq->arr[0]; // O primeiro elemento é o de maior prioridade
    pq->arr[0] = pq->arr[--pq->size]; // Substitui o topo pelo último elemento
    heapify(pq, 0); // Reorganiza a fila para manter a estrutura de heap
    return max; // Retorna o valor removido
}
```

- Função extractMax:
 - Remove o elemento de maior prioridade (no caso de um Máx-Heap, é o topo do heap).
 - Substitui o topo pela última posição do array.
 - A função heapify é chamada para reorganizar a fila e garantir que a propriedade do heap seja restaurada.

Reapify

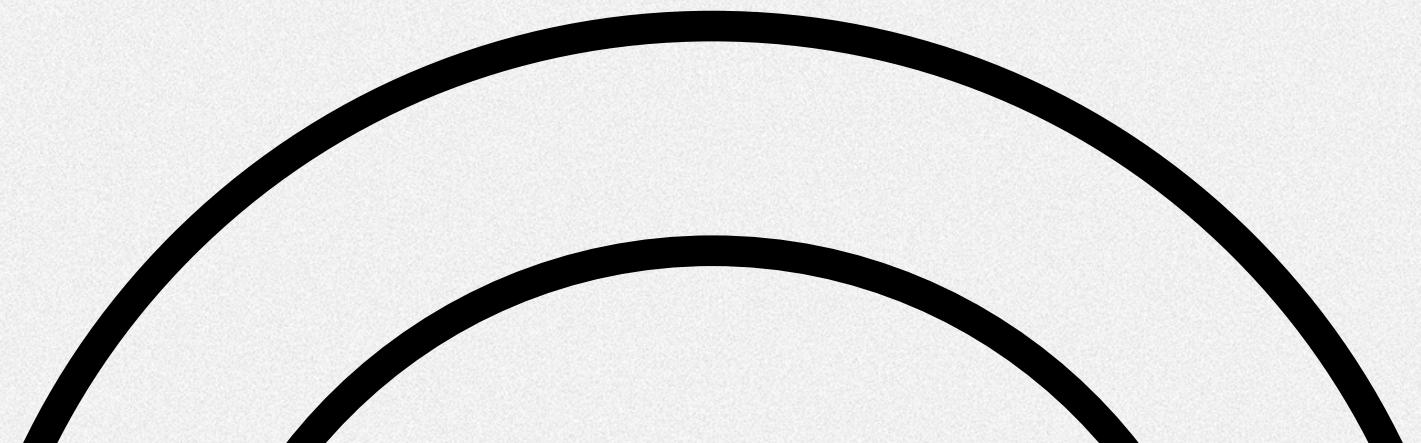
HEAPIFY (REORGANIZANDO A FILA)

```
void heapify(struct PriorityQueue *pq, int i) {  
    int largest = i; // Inicializa o maior como o nó atual  
    int left = 2 * i + 1; // Índice do filho à esquerda  
    int right = 2 * i + 2; // Índice do filho à direita  
  
    // Verifica se o filho à esquerda é maior que o nó atual  
    if (left < pq->size && pq->arr[left] > pq->arr[largest])  
        largest = left;  
  
    // Verifica se o filho à direita é maior que o nó atual  
    if (right < pq->size && pq->arr[right] > pq->arr[largest])  
        largest = right;  
  
    // Se o maior valor não for o nó atual, troca-os  
    if (largest != i) {  
        int temp = pq->arr[i];  
        pq->arr[i] = pq->arr[largest];  
        pq->arr[largest] = temp;  
        heapify(pq, largest); // Recursivamente reorganiza a árvore  
    }  
}
```

- Função heapify:
 - Garante que a estrutura do heap seja mantida após a remoção de um elemento.
 - A função compara o nó atual com seus filhos à esquerda e direita.
 - Se algum filho tiver maior prioridade, ele é trocado com o nó atual.
 - A função é chamada recursivamente até que a estrutura do heap seja restaurada.

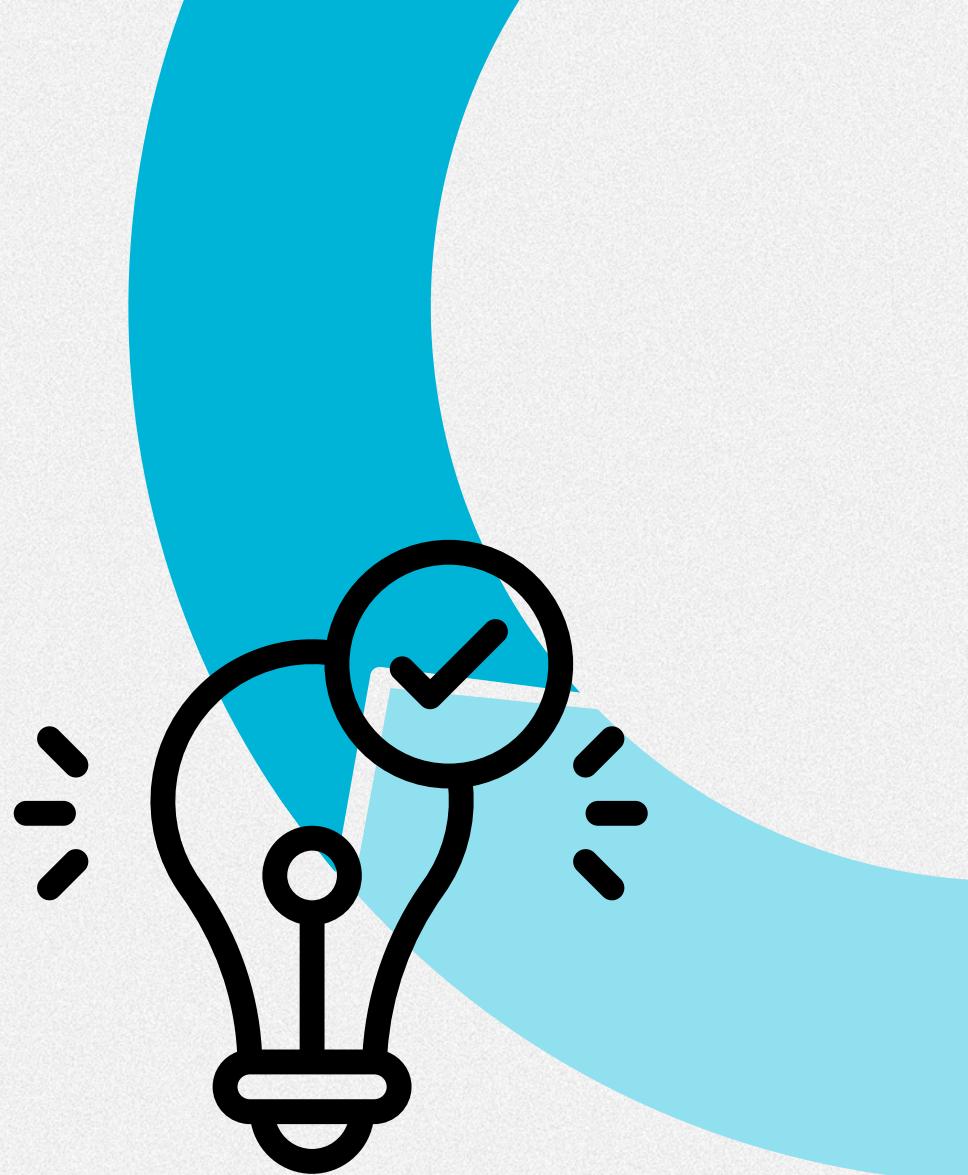


Union-Find



Introdução

- **Definição:** O Union-Find é uma estrutura de dados que gerencia conjuntos disjuntos, permitindo unir e buscar elementos eficientemente.
- **Principais operações:**
 - $\text{find}(x)$: Encontra o representante do conjunto de x .
 - $\text{union}(x, y)$: Une dois conjuntos distintos.
- **Aplicabilidade:**
 - Algoritmo de Kruskal (para árvores geradoras mínimas)
 - Componentes conectados em grafos
 - Problemas de conectividade em redes



Conceito

Funcionamento

- Cada elemento pertence a um conjunto identificado por um líder (ou raiz).
- Representação comum: um vetor `parent[]`, onde `parent[i]` aponta para o pai do elemento `i`.
- Inicialmente, cada elemento é seu próprio pai (`parent[i] = i`).

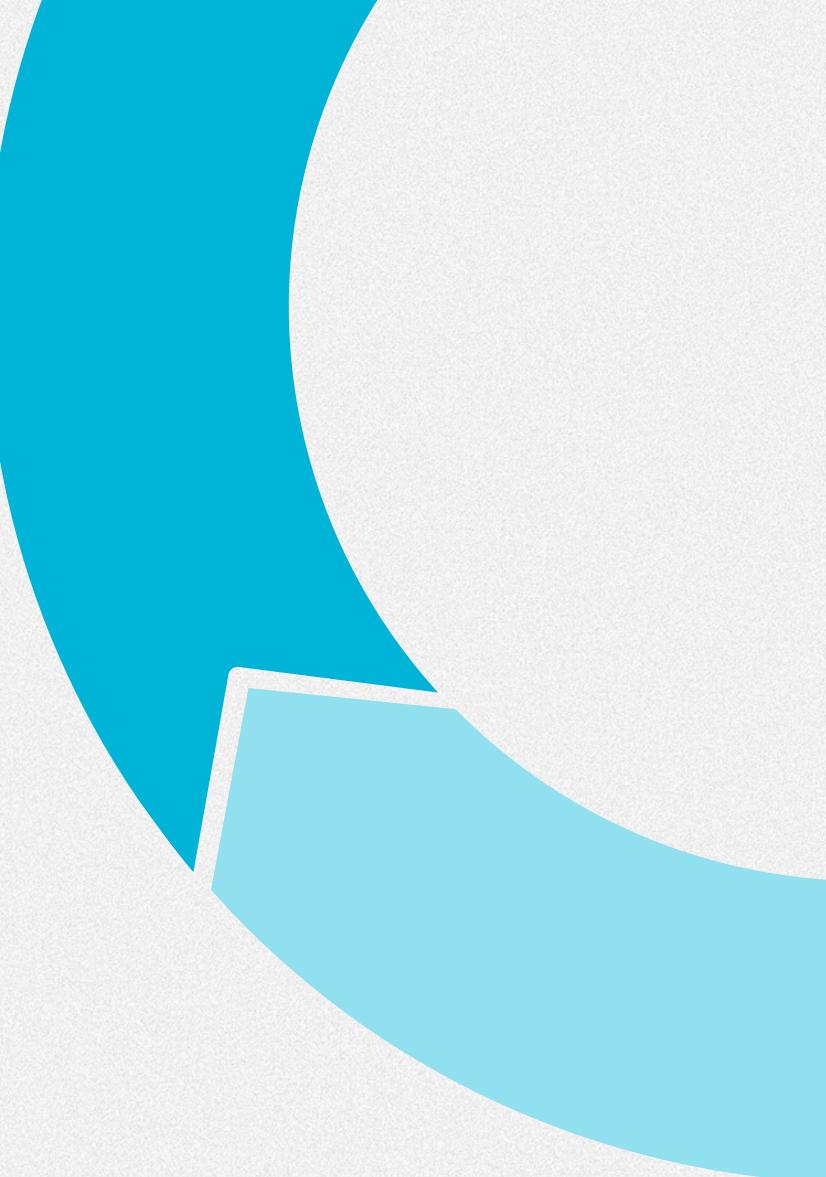


Implementação

- A estrutura básica pode ser implementada com vetores e duas otimizações importantes:
 - Compressão de Caminho (Path Compression): Otimiza o Find reduzindo a profundidade da árvore.
 - União por Ranking (Union by Rank): Faz com que o conjunto com menos elementos seja anexado ao maior.

Path-Compression

- O Path Compression melhora a eficiência da operação $\text{find}(x)$.
- Quando buscamos o representante de um conjunto, fazemos com que todos os nós no caminho apontem diretamente para a raiz.
- Isso reduz a profundidade da árvore, tornando futuras buscas mais rápidas.





SEM PATH-COMPRESSION

`union(1, 2)`

`union(2, 3)`

`union(3, 4)`

`1 → 2 → 3 → 4`

- Toda vez que chamamos `find(1)`, precisamos percorrer toda a cadeia.



COM PATH-COMPRESSION

- Toda vez que chamamos `find(1)`, todos os nós apontam diretamente para a raiz:

`1 → 4`

`2 → 4`

`3 → 4`

`4 → 4`

Union by Rank

- O Union by Rank otimiza a operação $\text{union}(x, y)$.
- Ele mantém as árvores o mais rasas possível.
- Sempre anexando a árvore menor à árvore maior.





SEM UNION BY RANK

Se fizermos:

$\text{union}(1, 2)$

$\text{union}(3, 4)$

$\text{union}(2, 3)$

podemos acabar com uma árvore
mais profunda:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4$



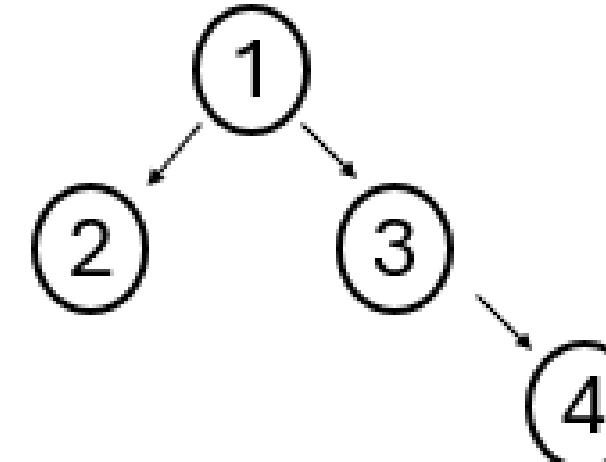
COM UNION BY RANK

Se fizermos:

$\text{union}(1, 2)$



$\text{union}(1, 3)$



$\text{union}(3, 4)$



Agora a árvore ficou mais rasa, o que torna as
buscas ($\text{find}(x)$) mais rápidas!

Exercício

- Temos 5 pessoas (1, 2, 3, 4 e 5) e queremos verificar se estão no mesmo grupo. Realize as operações:

`union(1, 2)`

`union(3, 4)`

`union(2, 4)`

Após realizar as operações, responda:

1 e 3 estão no mesmo grupo?

1 e 5 estão no mesmo grupo?

```
#include <stdio.h>

// Vetores para armazenar os pais e os ranks dos conjuntos
int parent[6], rank[6];

// Inicializa os conjuntos: cada elemento é seu próprio pai (representante do conjunto)
void makeSet() {
    for (int i = 1; i <= 5; i++) {
        parent[i] = i; // No início, cada elemento é sua própria raiz
        rank[i] = 0;   // Inicialmente, todos os ranks são 0
    }
}
```

```
// Função para encontrar a raiz de um conjunto com Path Compression
int find(int x) {
    if (parent[x] != x) // Se x não é sua própria raiz
        parent[x] = find(parent[x]); // Faz o caminho apontar diretamente para a raiz
    return parent[x]; // Retorna a raiz do conjunto
}
```

```
// Função para unir dois conjuntos usando Union by Rank
void unionSet(int x, int y) {
    int rootX = find(x); // Encontrar a raiz do conjunto de x
    int rootY = find(y); // Encontrar a raiz do conjunto de y

    if (rootX != rootY) {
        if (rank[rootX] > rank[rootY])
            parent[rootY] = rootX;
        else if (rank[rootX] < rank[rootY])
            parent[rootX] = rootY;
        else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
}
```

// Se não pertencem ao mesmo conjunto

// Se a árvore de rootX for maior

// Fazemos rootY apontar para rootX

// Se a árvore de rootY for maior

// Fazemos rootX apontar para rootY

// Se as alturas forem iguais

// Fazemos rootY apontar para rootX

// Aumentamos o rank da nova raiz

```
int main() {  
    makeSet(); // Inicializa os conjuntos  
  
    // Realizamos algumas uniões  
  
    unionSet(1, 2); // Une 1 e 2  
    unionSet(3, 4); // Une 3 e 4  
    unionSet(2, 4); // Une os conjuntos {1,2} e {3,4}, criando um grupo maior  
  
    // Testamos se elementos estão no mesmo grupo  
  
    printf("1 e 3 estão no mesmo grupo? %s\n", find(1) == find(3) ? "Sim" : "Não");  
    printf("1 e 5 estão no mesmo grupo? %s\n", find(1) == find(5) ? "Sim" : "Não");  
  
    return 0;  
}
```

Obrigado

