Tipos Abstratos de Dados (TADs)

Tipos abstratos de dados (TADs)

- Facilitam a comunicação sobre as operações em EDs complexas
- Definido pelo tipo de objeto que pode armazenar e operações permitidas
- São como objetos em linguagens orientadas a objetos
  - São sensíveis ao histórico de operações realizadas, ao contrário de funções
  - Possui uma interface pública de métodos que podem ser chamado por outros objetos
  - As EDs internas são ocultas
    - Oculta detalhes desnecessários para quem usa
    - Garante que quem usa não danifica as EDs
  - Possuem invariantes mantidos durante toda a existência do objeto
- Podemos usar TADs em nossos algoritmos
  - A corretude de nosso algoritmo independe da implementação dos TADs
  - A complexidade geralmente depende da implementação: avalie as possibilidades
- Conjunto restrito de operações pode melhorar a eficiência

Tipos simples (fornecidos pelas LPs)

- Números inteiros
- Números ponto flutuante
- Strings
- Arrays
- Registros

TAD Lista

- Sequência ordenada de elementos
  - Ao contrário de arrays, não possui posições vazias
- Elementos podem ser:
  - Inseridos
  - Removidos
  - Lidos (acessados)
  - Modificados
  - Buscados
- Implementação com array
  - i-ésimo elemento pode ser acesso em θ(1)
  - Inserção e remoção em θ(n), devido ao deslocamento de elementos
  - Deixando espaços em branco
    - Inserção e remoção em θ(1)
    - Acesso do i-ésimo elemento em θ(n), devido ao salto de espaços em branco

TAD Lista

- Implementação com lista encadeada
  - Arrays ocupam regiões contíguas na memória
    - Aumento do tamanho pode exigir copiar todos os elementos para outra região
    - Listas encadeadas mudam de tamanho de forma mais eficiente
  - Desvantagens:
    - Gasto de memória com ponteiros
    - Acesso do i-ésimo elemento em $\theta(n)$
- Implementação com árvores
  - Heaps, árvores AVL e árvores rubro-negras realizam todas as operações em $\theta(\log n)$

TAD Pilha

- Similar a uma lista, mas ocorrem apenas em um dos extremos
  - push: insere elemento no topo da pilha
  - pop: remove elemento do topo da pilha
- Modo LIFO: last in, first out
- Operações são θ(1)
- Implementação com array
  - Base da pilha é armazenada no início do array
  - Uma variável “topo” indica a posição onde está o topo da pilha
  - Operações push e pop movem o ponteiro do topo da pilha
- Implementação com lista encadeada

TAD Fila

- Similar a uma lista, mas operações ocorrem apenas nos extremos
  - Insere no final da fila
  - Remove do início da fila
- Modo FIFO: first in, first out
- Operações são θ(1)
- Implementação com array
  - Estrutura circular: depois do último temos o primeiro elemento
  - Duas variáveis indicam o final e o início da fila
- Implementação com lista encadeada

TAD Fila de Prioridade

- Elementos com maior prioridade movem para o início da fila
  - Na inserção informamos a prioridade do elemento
  - Podemos alterar depois a prioridade de um elemento
  - Removemos e retornamos o elemento com maior prioridade
    - Empates na prioridade são resolvidos de forma arbitrária
- Implementação com árvore
  - Heaps, árvores AVL e árvores rubro-negras realizam cada operação em $\theta(\log n)$

TAD Conjunto

- Similar a lista, mas os elementos não podem ser repetidos ou ordenados
- Implementação com array
  - Quando universo de possíveis elementos é pequeno
  - Cada elemento do array é um valor booleano indicando se o elemento está no conjunto
  - Todas as operações são θ(1)
- Implementação com tabelas hash
  - Permite universo infinito de possíveis elementos
  - Todas as operações são θ(1)

TAD Sistema de Conjuntos

- Conjunto ou lista onde os elementos são conjuntos
  - Criação de conjunto
  - União, interseção, complemento e subtração
  - Find (conjunto que contém um determinado elemento)
- Implementação
  - Array ou tabela hash, onde cada elemento é a implementação do TAD Conjunto
  - Geralmente as operações são $\theta(n)$
- Conjuntos disjuntos
  - ED union-find
  - Operações de união e find possuem tempo $\theta(1)$ (na prática)

TAD Dicionário

- Associa cada elemento a uma chave
  - Buscar elemento com determinada chave
  - Inserir elemento (informando a chave)
  - Remover elemento com determinada chave
- Implementação com tabelas hash
  - Operações são θ(1)

Exemplos - Parsing

- Dada uma sequência de chaves, parênteses e colchetes
  - Numere cada par com um mesmo valor inteiro, distinto dos outros pares

```
Entrada: ( [ { } ( ) ] ( ) { ( ) } )
Saída:   1 2 3 3 4 4 2 5 5 6 7 7 6 1
```

- Passos básico:
  - Algoritmo do tipo mais da entrada
  - Se for um símbolo de abertura ([ou {
    - Empilhe símbolo e o id
    - Imprima e incremente o id
  - Senão (símbolo de fechamento)
    - Se pilha vazia ou o símbolo no topo não for do mesmo tipo, retorne “Inválida”
    - Imprima o id do topo
    - Remova o elemento do topo

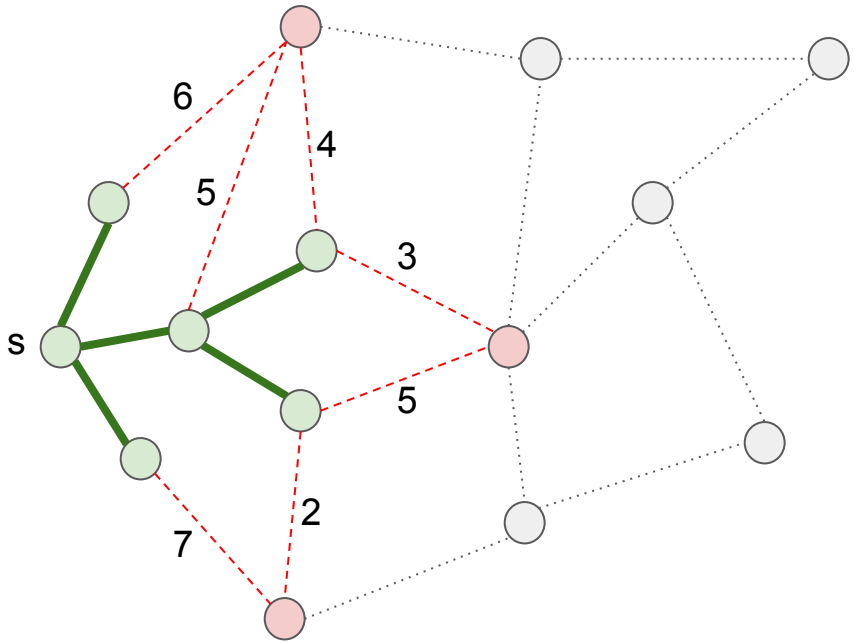
Exemplos - Parsing

```
algoritmo Parsing(s)
    <pre-cond>: s[1..n] é uma string contendo apenas ()[]{}.
    <pos-cond>: imprima inteiros que associem abertura c/ fechamento do mesmo tipo.
    P = pilha vazia
    id = 1
    para i = 1 até n
        se s[i] é igual a '(' ou '[' ou '{'
            imprima id
            P.push( (s[i], id) )
            id = id + 1
        senão
            se P é vazia, retorne "Inválida"
            (topo_s, topo_id) = P.pop()
            se s[i] e topo_s não são do mesmo tipo, retorne "Inválida"
            imprima topo_id
```

Exemplos - Frequência de palavras

```
algoritmo frequência_palavras(s)
    <pre-cond>: s é uma string.
    <pos-cond>: imprime cada palavra encontrada em s, e sua frequência.

    D = dicionário vazio
    para cada palavra w em s
        se w é uma chave de D
            D[w] = D[w] + 1
        senão
            D[w] = 1

    para cada chave w em D
        imprima w e D[w]
```

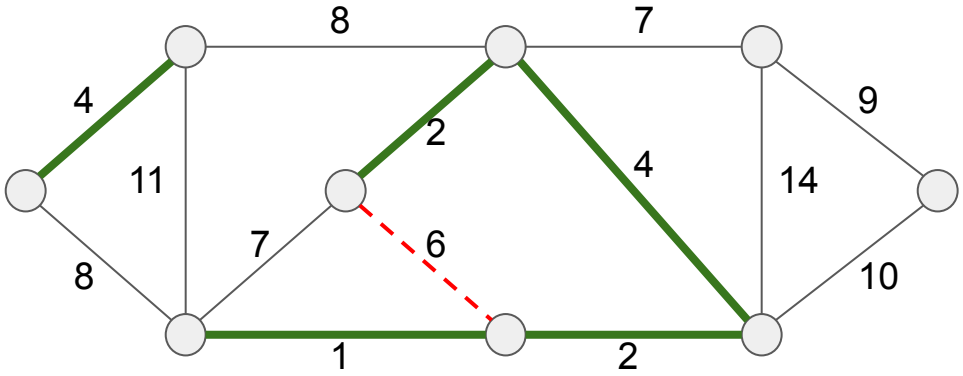
Exemplos - Prim



Exemplos - Prim

```
algoritmo Prim(G,s,w)
     <pre-cond>: G é grafo não direcionado, s é o nó inicial, w[] é a matriz de pesos.
     <pos-cond>: retorna uma árvore geradora mínima (predecessor de cada nó).

     para cada nó u de G
          Pred[u] = Nulo;  P[u] = -∞

     Q = fila de prioridade vazia
     Insira todo nó v de G em Q com prioridade -∞
     Atualize prioridade de s em Q para 0, e faça P[s] = 0

     enquanto Q não é vazia
          u = extraia (remova e retorne) o nó com maior prioridade em Q
          para cada v adjacente a u em G
               se v está em Q e -w[u,v] > P[v]
                    Atualize a prioridade de v em Q para -w[u,v], e faça P[v] = -w[u,v]
                    Pred[v] = u

     retorne Pred
```

Exemplos - Kruskal



Exemplos - Kruskal

```
algoritmo Kruskal(G,w)
     <pre-cond>: G é grafo não direcionado, w[] é a matriz de pesos.
     <pos-cond>: retorna uma árvore geradora mínima (arestas escolhidas).

     A = conjunto vazio // arestas escolhidas

     S = union-find vazia
     para cada nó u em G
          Insira o conjunto {u} em S

     Ordene as aresta de G por peso (não decrescente)
     para cada aresta (u,v) em ordem não decrescente de peso
          se S.find(u) != S.find(v) // não estão no mesmo conjunto
               S.union(S.find(u), S.find(v))
               Insira (u,v) no conjunto A

     retorne A
```