Semana 6

Principais estruturas de dados: notação e complexidade

Estruturas de dados são formas de organizar os dados em memória. Um **tipo abstrato de dados** é um conjunto de operações que podem ser realizadas nos dados. Um mesmo tipo abstrato de dados pode ter **várias implementações possíveis**, e estas implementações **afetam o tempo de execução** das operações. Portanto, ao utilizar tipos abstratos de dados no algoritmo que estamos projetando, precisamos decidir quais implementações destes tipos abstratos de dados devem ser utilizadas de modo a deixar nosso algoritmo mais eficiente.

Em cada seção a seguir temos um **tipo abstrato, e indicado entre parênteses a implementação** deste tipo abstrato que será discutida. Nos pseudocódigos ao longo da disciplina vou assumir que esta implementação indicada entre parênteses é a **implementação adotada para o tipo abstrato, a menos que outra implementação seja informada** na descrição do algoritmo.

Para cada tipo abstrato é fornecida uma tabela contendo o **nome da operação**, um exemplo da **sintaxe** adotada para uso da operação, e a **complexidade da operação** na implementação discutida. Para simplificar a implementação, a sintaxe adotada é idêntica à utilizada na linguagem python. Podemos utilizar outras sintaxes nos pseudocódigos, mas neste caso é necessário deixar claro qual é a operação e o tipo abstrato. Além disso, se não utilizar a convenção indicada aqui, durante a determinação da complexidade é necessário informar qual implementação do tipo abstrato será considerada na análise.

Obs.: Para o **comando de atribuição** podemos usar os símbolos $=$ ou $\leftarrow$, mas se utilizar o símbolo $=$ é melhor fazer o teste de igualdade utilizando $==$. Os **índices de uma lista** podem ser indicados entre colchetes, como em $L[1..n]$, mas se não for informado vamos assumir que elas são indexadas começando de zero, como nas linguagens de programação.

6.1 Lista (com array dinâmico)

Uma lista é uma **sequência de elementos que podem ser acessados pela sua posição** na sequência. O valor n representa o tamanho (número de elementos) da lista. A menos que seja indicado o contrário, as posições começam em 0 e vão até $n-1$.

Operação	Exemplo	Complexidade
Cria vazia	`l = []`	$O(1)$
Cria inicializada	`l = [v1,..,vn]`	$O(n)$
Insere na posição	`l[a:b] = [v]`	$O(n)$
Insere no final	`l.append(v)`	$O(1)$
Remove da posição	`v = l.pop(i)`	$O(n)$
Remove do final	`v = l.pop()`	$O(1)$
Remove valor	`l.remove(v)`	$O(n)$
Acessa posição	`v = l[i]`	$O(1)$
Altera posição	`l[i] = v`	$O(1)$
Contido (`in, not in`)	`v in L`	$O(n)$
Mínimo	`v = min(l)`	$O(n)$
Máximo	`v = max(l)`	$O(n)$
Inverte ordem	`l.reverse()`	$O(n)$
Ordena	`l.sort()`	$O(n \log n)$
Posição do valor	`i = l.index(v)`	$O(n)$
Compara (`==, !=`)	`l1 == l2`	$O(n)$
Concatena	`l = l1 + l2`	$O(n)$
Tamanho	`len(l)`	$O(1)$
Iteração	`for v in l`	$O(n)$
Copia	`l1 = l.copy()`	$O(n)$
Fatia (slice)	`l1 = l[a:b]`	$O(b-a)$
Repete	`l1 = k*l`	$O(kn)$

- "Insere na posição": acrescenta elementos com valor v da posição a até a posição b, mantendo os elementos que já estavam na lista. Portanto, se quiser acrescentar o valor v na posição i, basta fazer `l[i:i] <- [v]`. Note que esta operação acrescenta elementos no meio da lista, deslocando os elementos originais. Se quiser apenas alterar o valor em uma determinada posição, use a operação "Altera posição".
- "Remove valor": remove a primeira ocorrência do valor v na lista, ou lança uma exceção se v não for encontrado. No pseudocódigo basta testar se a operação retorna "Não encontrado".
- "Posição do valor": retorna a primeira posição que contém o valor v, ou lança uma exceção se v não for encontrado. No pseudocódigo basta testar se o valor retornado é igual a "Não encontrado".
- "Repete": consiste em concatenar k cópias da lista.
- "Fatia" e "Repete" criam cópias da lista.
- As complexidades decorrem do fato das listas em python serem implementadas com arrays dinâmicos.
- "Mínimo" e "Máximo" podem ser realizadas em $O(1)$ se não permitirmos remoções. Para isso, basta atualizar duas variáveis que guardam os valores mínimo e o máximo sempre que uma inserção for feita. Note que a manutenção destas variáveis deixa de ter tempo constante se permitirmos remoções.

Tuplas

Tuplas são **listas imutáveis**, ou seja, cujo tamanho e referências dos elementos não podem ser modificados. A sintaxe das tuplas é similar à sintaxe das listas, exceto que usamos parênteses ao invés de colchetes. As tuplas **suportam todas as operações de lista que não provocam modificação**. As tuplas são muito utilizadas como **retorno de funções** (permite retornar vários valores), como **elementos de estruturas de dados** (vários valores armazenados em um mesmo elemento), e para **inicializar várias variáveis** com um único comando de atribuição.

6.2 Pilha, fila, deque (com lista encadeada)

Uma **pilha** é uma lista em que os elementos são **inseridos e removidos em apenas um dos extremos da lista**. Vimos que podemos inserir e remover no final de um lista em $O(1)$, o que nos permite concluir que as operações do tipo pilha podem ser implementadas em **tempo constante utilizando as operações de lista** `l.append(v)` e `v <- l.pop()`.

Uma **fila** é uma lista em que os elementos são **inseridos em um dos extremos e são removidos no outro extremo**. Como a inserção e a remoção no início da lista são operações com tempo $O(n)$, concluímos que estas operações são **ineficientes se implementadas com listas**.

É possível **inserir e remover no início e no final de uma lista em tempo constante** utilizando uma **lista duplamente encadeadas**. O preço que pagamos por utilizar uma lista encadeada é que o acesso a uma posição no meio da lista passa a ter tempo linear (ao invés de constante), mas em uma fila acessamos apenas elementos que estão nos extremos. Na linguagem python, uma lista duplamente encadeada é implementada no módulo `collections.deque`, cujas operações são apresentadas abaixo, com destaque para as complexidades que diferem do tipo lista.

Deque

Operação	Exemplo	Complexidade
Cria vazio	`q = deque([])`	$O(1)$
Cria inicializado	`q = deque([v1,..,vn])`	$O(n)$
Insere no início	`q.appendleft(v)`	$O(1)$
Insere no final	`q.append(v)`	$O(1)$
Remove do início	`v = q.popleft()`	$O(1)$
Remove do final	`v = q.pop()`	$O(1)$
Remove valor	`q.remove(v)`	$O(n)$
Acessa posição	`v = q[i]`	$O(n)$
Altera posição	`q[i] = v`	$O(n)$
Contido (`in, not in`)	`q in L`	$O(n)$
Mínimo	`v = min(q)`	$O(n)$
Máximo	`v = max(q)`	$O(n)$
Inverte ordem	`q.reverse()`	$O(n)$
Posição do valor	`i = q.index(v)`	$O(n)$
Compara (`==, !=`)	`q1 == q2`	$O(n)$
Concatena	`q = q1 + q2`	$O(n)$
Tamanho	`len(q)`	$O(1)$
Iteração	`for v in q`	$O(n)$
Copia	`q1 = q2.copy()`	$O(n)$
Repete	`k*q`	$O(kn)$

6.3 Dicionário (com tabela hash)

Um dicionário permite **localizar valores através de chaves**, ou seja, para cada valor v armazenado associamos uma chave exclusiva k. Desta forma, é possível localizar o valor v utilizando a chave k. Por exemplo, em um dicionário da língua portuguesa, a chave é a palavra k que desejamos localizar, e o valor v é o texto que descreve o significado da palavra.

Ao contrário das listas, que só permitem localizar um valor indicando a posição dele na lista (ou seja, a chave é um inteiro entre 0 e o tamanho da lista menos 1), os dicionários **permitem utilizar qualquer objeto imutável como chave**. As operações permitidas são apresentadas na tabela abaixo, e as complexidades fornecidas são de caso médio, e elas decorrem do fato do dicionário ser implementado com tabela hash.

Operação	Exemplo	Complexidade
Cria vazio	`d = {}`	$O(1)$
Cria inicializado	`d = {k1:v1,..,kn:vn}`	$O(n)$
Insere	`d[k] = v`	$O(1)$
Remove	`v = d.pop(k)`	$O(1)$
Acessa	`v = d[k]`	$O(1)$
Contido (`in, not in`)	`k in d`	$O(1)$
Menor chave	`k = min(d)`	$O(n)$
Maior chave	`k = max(d)`	$O(n)$
Compara (`==, !=`)	`d1 == d2`	$O(n)$
Tamanho	`len(d)`	$O(1)$
Iteração	`for k in d`	$O(n)$
Copia	`d1 = d.copy()`	$O(n)$
Lista de chaves (cópia)	`l = list(d.keys())`	$O(n)$
Lista de valores (cópia)	`l = list(d.values())`	$O(n)$
Lista de tuplas (cópia)	`l = list(d.items())`	$O(n)$

- "Acessa": lança uma exceção se a chave k não está no dicionário. No pseudocódigo basta testar se o valor retornado é "Não encontrado".
- Para obter um valor default u quando a chave k não existe, utilize `v = d.get(k,u)`. Para armazenar o valor v na chave k apenas se a chave k ainda não existir no dicionário d, utilize `d.setdefault(k,v)`.
- "Contido": verifica se o dicionário possui a chave k.
- "Iteração": realiza uma iteração do for para cada chave k.
- "Compara": retorna verdadeiro se e somente se os dois dicionários possuem as mesmas chaves, os mesmos valores e mesma associação entre chaves e valores.
- "Lista de tuplas": retorna lista de tuplas com os elementos do dicionário, onde na 1a posição da tupla temos a chave e 2a posição o valor.

6.4 Conjunto (com tabela hash)

O tipo conjunto é similar a um conjunto na matemática: **elementos não podem ser repetidos, e não existe uma ordem entre os elementos**. Nas operações de dicionário, observamos que a operação "Contido" é realizada em tempo constante pela tabela hash. Desta forma, os conjuntos em python são implementados como tabelas hash, onde os valores são ignorados (usamos apenas as chaves). **Nos pseudocódigos, as operações com conjuntos também podem ser representadas em notação matemática.**

Operação	Exemplo	Complexidade
Cria vazio	`A = set()`	$O(1)$
Cria inicializado	`A = {v1,..,vn}`	$O(n)$
Insere	`A.add(v)`	$O(1)$
Remove	`A.remove(v)`	$O(1)$
Contido (`in, not in`)	`v in A`	$O(1)$
União	`C = A | B`	$O(\|A\| + \|B\|)$
Interseção	`C = A & B`	$O(\min(\|A\|, \|B\|))$
Diferença	`C = A - B`	$O(\|A\|)$
Compara (`==, !=`)	`A == B`	$O(\|A\|)$
Subconjunto	`A <= B`	$O(\|A\|)$
Tamanho	`len(A)`	$O(1)$
Iteração	`for v in A`	$O(n)$
Copia	`C = A.copy()`	$O(n)$
Copia para lista	`l = list(A)`	$O(n)$

- "Compara": executa em $O(1)$ se os tamanhos de A e B são distintos.
- "Subconjunto": retorna verdadeiro se e somente se A está contido ou é igual a B.

6.5 Multiconjunto (com tabela hash)

Da mesma forma que em um conjunto, em um multiconjunto **não temos uma ordem entre os elementos**. Porém, ao contrário de um conjunto, em um multiconjunto **podemos ter elementos repetidos**. Uma forma eficiente de implementar o tipo multiconjunto é utilizar uma tabela hash onde os elementos armazenados são as chaves, e o valor associado a cada chave k é o número de ocorrências do elemento k no multiconjunto. Esta estrutura está implementada na classe `Multiset` do módulo python `multiset` (`https://pypi.org/project/multiset`). A sintaxe e a complexidade das operações na tabela estão de acordo com esta implementação.

Operação	Exemplo	Complexidade
Cria vazio	`A = Multiset()`	$O(1)$
Cria inicializado	`A = Multiset({v1:c1,..,vn:cn})`	$O(n)$
Acessa # cópias	`c = A[v]`	$O(1)$
Define # cópias	`A[v] = c`	$O(1)$
Contido (`in, not in`)	`v in A`	$O(1)$
União	`C = A | B`	$O(\|A\| + \|B\|)$
Interseção	`C = A & B`	$O(\min(\|A\|, \|B\|))$
Diferença	`C = A - B`	$O(\|A\|)$
Compara (`==, !=`)	`A == B`	$O(\|A\|)$
Subconjunto	`A <= B`	$O(\|A\|)$
Tamanho	`len(A)`	$O(1)$
Iteração	`for v in A`	$O(n)$
Copia	`C = A.copy()`	$O(n)$
Copia para lista	`l = list(A)`	$O(n)$

- "Acessa # cópias": retorna o número de cópias de v se v está no multiconjunto, ou zero se o elemento v não está no conjunto.
- "Subconjunto": retorna verdadeiro se e somente se o número de cópias de cada elemento v em A é menor ou igual ao número de cópias de v em B.
- "Tamanho": retorna o total de elementos considerando as repetições.

6.6 Conjuntos disjuntos (com union-find)

No tipo conjunto é possível fazer a união em tempo $O(\min(|A|, |B|))$ copiando os elementos do menor conjunto para o maior conjunto, com a desvantagem de perder o conjunto original. Ainda assim, o tempo de execução da união depende do número de elementos nos conjuntos.

Existe uma situação em que a **união pode ser feita em tempo "praticamente" constante: quando os conjuntos são mantidos disjuntos, ou seja, sem interseção.** Neste caso dispomos da estrutura de dados union-find. Uma implementação em python está disponível em `https://github.com/deehzee/unionfind/blob/master/unionfind.py` A tabela abaixo fornece a notação e a complexidade de cada operação nesta implementação.

Operação	Exemplo	Complexidade
Cria vazio	`U = UnionFind()`	$O(1)$
Cria inicializado	`U = UnionFind([e1,..,en])`	$O(n)$
Insere	`U.add(e)`	$O(1)$
União	`U.union(e1,e2)`	$O(1)$
Find	`U.find(e)`	$O(1)$
Conectados	`U.connected(e1,e2)`	$O(1)$
Componente	`U.component(e)`	$O(n)$
Componentes	`U.components(e)`	$O(mn)$
Contido (`in, not in`)	`e in U`	$O(1)$
Tamanho	`len(U)`	$O(1)$
Iteração	`for e in U`	$O(n)$
Copia	`U1 = UnionFind(list(U))`	$O(n)$
Copia para lista	`l = list(U)`	$O(n)$

- "Cria inicializado" cria um conjunto para cada elemento.
- "Tamanho": número de elementos (não de conjuntos).
- "Insere": acrescenta um novo elemento e, e cria um novo conjunto contendo apenas este novo elemento.
- "Find": retorna o identificador do conjunto que contém o elemento e. Este identificador é utilizado para verificar se dois elementos estão no mesmo conjunto.
- "Conectados": retorna verdadeiro apenas se os elementos e_1 e e_2 estão no mesmo conjunto.
- "União": realiza a união do conjunto que contém e_1 com o conjunto que contém e_2.
- "Componente": retorna o conjunto que contém o elemento e.
- "Componentes": retorna uma lista com todos os conjuntos. A complexidade é $O(mn)$, onde m é o número de conjuntos e n é o total de elementos.

6.7 Conjunto ordenado (com árvore balanceada)

Suponha que seu algoritmo precisa realizar **várias operações de inserção/remoção e também determinar várias vezes qual é o i-ésimo menor elemento**. Para comparar a complexidade das implementações abaixo, considere o cenário onde vamos inserir n elementos, e em seguida fazer n operações de inserção/remoção e depois n operação de busca pelo i-ésimo menor elemento.

Array não ordenado Gastamos $O(1)$ por inserção, $O(n)$ por remoção, e $O(n)$ por busca do i-ésimo menor, totalizando $O(n^2)$.

Array ordenado Gastamos $O(n \log n)$ para inserir e ordenar os n primeiros elementos, $O(n)$ por inserção/remoção, e $O(1)$ por busca do i-ésimo menor, totalizando $O(n^2)$.

Árvore balanceada Gastamos $O(\log n)$ em todas as operações, totalizando $O(n \log n)$.

Concluímos que neste caso a implementação com **árvore balanceada produz a menor complexidade total**. A sintaxe das operações apresentadas na tabela corresponde à classe `SortedSet` do módulo python `sortedcontainers` (`https://pypi.org/project/sortedcontainers/`). As complexidades listadas são de implementações de árvore balanceada, embora a classe `SortedSet` esteja implementada com árvore-B, resultando em complexidade amortizadas $O(\sqrt[3]{n})$ para inserção/remoção/busca. Apesar disso, como as constantes da complexidade da árvore-B são menores que de árvores balanceadas, os desenvolvedores afirmam que na prática o desempenho da árvore-B é melhor (`http://www.grantjenks.com/docs/sortedcontainers/performance-scale.html`).

Operação	Exemplo	Complexidade
Cria vazio	`S = SortedSet()`	$O(1)$
Cria inicializado	`S = SortedSet([v1,..,vn])`	$O(n \log n)$
Insere	`S.add(v)`	$O(\log n)$
Remove	`S.remove(v)`	$O(\log n)$
Contido (`in, not in`)	`v in S`	$O(1)$
i-ésimo menor	`S[i]`	$O(\log n)$
Mínimo	`v = S[0]`	$O(\log n)$
Máximo	`v = S[-1]`	$O(\log n)$
União	`C = A | B`	$O(\|A\| + \|B\|)$
Interseção	`C = A & B`	$O(\min(\|A\|, \|B\|))$
Diferença	`C = A - B`	$O(\|A\|)$
Compara (`==, !=`)	`A == B`	$O(\|A\|)$
Subconjunto	`A <= B`	$O(\|A\|)$
Tamanho	`len(S)`	$O(1)$
Iteração	`for v in S`	$O(n)$
Copia	`C = S.copy()`	$O(n)$
Copia para lista	`l = list(S)`	$O(n)$

- "i-ésimo menor": retorna o i-ésimo menor elemento começando da posição 0, ou seja, o menor elemento está na posição 0, o segundo menor na posição 1, assim sucessivamente. Portanto, o maior elemento está na última posição, que pode ser acessada com `S[-1]`.

6.8 Fila de prioridade (com heap)

Na fila de prioridade estamos interessados **apenas na operação de inserção de elemento, e na operação de remoção do menor elemento.** Por ser **mais restrita** que o conjunto ordenado, a fila de prioridade consegue realizar esta operações de forma **mais eficiente**. Note que o conjunto ordenado permite adicionalmente remover qualquer elemento e localizar o i-ésimo menor. A tabela mostra as complexidades da implementação das operações de fila de prioridade utilizando heap. Adotamos como notação as funções disponíveis no módulo python `heapq` (`https://docs.python.org/3/library/heapq.html`).

Operação	Exemplo	Complexidade
Cria vazia	`h = []`	$O(1)$
Cria inicializada	`h = [v1,..,vn]`	$O(n)$
	`heapify(h)`	$O(n)$
Insere v na heap h	`heappush(h,v)`	$O(\log n)$
Remove menor	`v = heappop(h)`	$O(\log n)$
Mínimo	`v = h[0]`	$O(1)$
Aumenta chave	`increase_key(h,v,k)`	$O(\log n)$
Reduz chave	`decrease_key(h,v,k)`	$O(\log n)$

- A heap é mantida em uma lista, sem dados adicionais. Portanto, para criar uma heap vazia simplesmente declaramos uma lista vazia, e para criar uma heap inicializada chamamos `heapfy` para transformar a lista em uma heap (esta operação apenas reposiciona elementos da lista).
- Todas as operações de lista também são permitidas na heap, pois a heap é uma lista. Porém, modificações na lista podem fazê-la deixar de ser uma heap.
- Se os elementos são números, para ter uma heap que acessa e remove o maior elemento (ao invés do menor elemento) inverta o sinal dos números armazenados. Esta estratégia também é válida quando os elementos são tuplas, e o primeiro elemento é um número que determina a ordem das tuplas (veja a seção "Comparação de tuplas"). Quando usamos tuplas, a primeira posição é chamada de **chave**.
- A operação `increase_key` aumenta para k o valor da chave de v na heap h. Da mesma forma, a operação `decrease_key` reduz para k o valor da chave de v na heap h. As duas operações exigem tempo $O(\log n)$, pois a mudança na chave k provoca um reposicionamento da tupla (k, v) na heap ao longo da altura da árvore. Estas operações estão implementadas no módulo `heapq` nos métodos "privados" `_siftup(h,p)` (correspondente ao `increase_key`) e `_siftdown(h,s,p)` (correspondente ao `decrease_key`), onde `p` é a posição na heap `h` do elemento que teve a chave modificada, e `s` é a posição inicial (que podemos fixar em zero) (veja o código-fonte: `https://hg.python.org/cpython/file/2.7/Lib/heapq.py`). Podemos usar um dicionário auxiliar para obter em $O(1)$ a posição de uma dada tupla na heap `h`. A desvantagem de usar métodos "privados" (métodos iniciando com underscore) é que versões futuras da biblioteca podem remover estes métodos.
- Vantagens da heap em relação ao conjunto ordenado:
  - Menor complexidade na criação inicializada e na consulta ao menor (sem remoção dele).
  - Embora tenha a mesma complexidade na inserção e remoção, a heap é mais rápida na prática, pois tem implementação mais simples e constantes menores na complexidade. Veja esta comparação de desempenho: `https://medium.com/@ssbothwell/comparing-sorted-containers-in-python-a2c41624bc84`

Comparação de tuplas

Podemos armazenar tuplas nas estruturas, obtendo assim mais flexibilidade, pois as tuplas permitem armazenar várias informações em cada elemento. Por exemplo, podemos colocar em uma heap tuplas contendo o nome, a matrícula e a nota de cada aluno.

Se tuplas são armazenadas nas estruturas, precisamos saber como elas são comparadas. Isto é particularmente importante nas estruturas "conjunto ordenado" e "fila de prioridade". Tuplas são comparadas **da menor posição para a maior posição**. Ex.: `(4,'def') < (5,'abc')` é verdadeiro, pois o elemento da 1a posição é menor na tupla da esquerda. Esta forma de comparar tuplas é chamada de **ordem lexicográfica**.

6.9 Grafo (com lista de adjacências)

Para grafos representados por **lista de adjacências**, vamos utilizar a notação e as complexidades indicadas na tabela. A notação é extraída da classe `Graph` do módulo python `graph_tool.all` (`https://graph-tool.skewed.de`).

Operação	Exemplo	Complexidade
Cria vazio	`g = Graph()`	$O(1)$
Cria vazio (não direcionado)	`g = Graph(directed=False)`	$O(1)$
Insere c vértices	`g.add_vertex(c)`	$O(c)$
Insere aresta	`g.add_edge(v1,v2)`	$O(1)$
Remove vértice	`g.remove_vertex(v,True)`	$O(d_v + d_{\text{last}})$
Remove aresta	`g.remove_edge(e)`	$O(1)$
Acessa vértice	`v = g.vertex(i)`	$O(1)$
Acessa aresta	`e = g.edge(i,j)`	$O(\min(d_i^{\text{out}}, d_j^{\text{in}}))$
Índice de vértice	`int(v)`	$O(1)$
Origem da aresta	`v = e.source()`	$O(1)$
Destino da aresta	`v = e.target()`	$O(1)$
Copia	`g2 = Graph(g1)`	$O(n + m)$
Iteração nos vértices	`for v in g.vertices()`	$O(n)$
Iteração nas arestas	`for e in g.edges()`	$O(m)$
Iteração incidência	`for u in v.all_neighbors()`	$O(d_v^{\text{in}} + d_v^{\text{out}})$
	`for u in v.in_neighbors()`	$O(d_v^{\text{in}})$
	`for u in v.out_neighbors()`	$O(d_v^{\text{out}})$
	`for e in v.all_edges()`	$O(d_v^{\text{in}} + d_v^{\text{out}})$
	`for e in v.out_edges()`	$O(d_v^{\text{out}})$
	`for e in v.in_edges()`	$O(d_v^{\text{in}})$

- Nas complexidades, n indica o número de vértices e m o número de arestas. O grau de entrada do vértice v é denotado por d_v^{in}, e o grau de saída é denotado por d_v^{out}. Se temos o índice i do vértice, denotamos o grau de entrada por d_i^{in}, e o grau de saída por d_i^{out}.
- "Acessa vértice": os vértices são indexados com inteiros de 0 até $n-1$. Retorna um objeto (descritor) que representa o vértice.
- "Acessa aresta": as arestas são indexadas através dos índices dos vértices que incidem nela, sendo o 1o o vértice de origem e o 2o o vértice de destino. Retorna um objeto (descritor) que representa a aresta.

- "Índice de vértice": dado o descritor v do vértice, retorna seu índice.
- A complexidade da remoção de vértice é da ordem do grau do vértice sendo removido mais o grau do último vértice (maior índice). Esta operação faz o último vértice (maior índice) passar a ter o índice do vértice sendo removido, invalidando assim os descritores que apontam para este último vértice.
- Nas operações onde a complexidade depende do grau do vértice, no pior caso podemos assumir que uma execução da operação é $O(m)$. Porém, observe que a soma dos graus de todos os vértices também é $O(m)$.
- Se for implementar usando o módulo `graph_tool`, observe que para a remoção de aresta ter tempo constante, é necessário executar antes `g.set_fast_edge_removal()`, o que exige consumo $O(m)$ de memória adicional. A operação provoca alteração nos índices das arestas, mas não invalida os descritores das arestas.

6.10 Exemplos

Os exemplos a seguir estão explicados no vídeo "tipos abstratos de dados (exemplos)", mas vou fornecer os pseudocódigos utilizando a notação e complexidades que foram apresentadas nestas notas de aula.

Algoritmo: Parsing($s[1..n]$)

Entrada: String $s[1..n]$ cujos caracteres são indexados de 1 até n. A string s contém apenas os símbolos (,), [,], { e }.
Saída : Retorna lista com inteiros que associam cada abertura com o fechamento correspondente em s, ou falso se s é inválida.

```
1  (P, prox_id, ids) ← ([ ], 0, [ ])                                  // O(1)
2  for i de 1 até n do                                                // O(n)
3      if s[i] = '(' or s[i] = '[' or s[i] = '{' then                 // O(n)
4          ids.append(prox_id)                          // O(n)·O(1) = O(n)
5          P.append((s[i], prox_id))                    // O(n)·O(1) = O(n)
6          prox_id ← prox_id + 1                                      // O(n)
7      else                                                           // O(n)
8          if len(P) = 0 then return false                            // O(n)
9          (topo_char, topo_id) ← P.pop()               // O(n)·O(1) = O(n)
10         if s[i] e topo_char não são do mesmo tipo then             // O(n)
11             return false                                           // O(1)
12         ids.append(topo_id)                          // O(n)·O(1) = O(n)
13 return ids                                                         // O(1)
```

Algoritmo: Frequência_palavras($s[1..n]$)

Entrada: Lista s$[1..n]$ contendo n palavras, indexadas de 1 até n.
Saída : Dicionário D, onde D[w] é o número de vezes que a palavra w aparece em s.

```
1  D ← {}                                                             // O(1)
2  for w in s do                                                      // O(n)
3      D[w] ← D.get(w,0) + 1                        // O(n)·O(1) = O(n)
4  return D                                                           // O(1)
```

Leitura complementar

Capítulos 3 e 4 do Skiena, e capítulos 6, 10, 11, 12, 13 e 21 do Cormen.

Livro online *"Estruturas de Dados Abertos"*:
`https://github.com/jaraujouerj/Estruturas-de-Dados-Abertos`
(tradução de `http://opendatastructures.org`)

Complexidades das operações em python:

- `https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt`
- `https://wiki.python.org/moin/TimeComplexity`

Algoritmo: Prim(G,w)

Entrada: Grafo não direcionado G. w[u, v] é o peso da aresta {u, v}.
Saída : Dicionário π, onde π[u] é o nó predecessor do nó u em uma árvore geradora mínima de G.

```
   // inicialização das estruturas
1  s ← nó qualquer de G                                               // O(1)
2  (π, key, Q) ← ({}, {}, [ ])                                        // O(1)
3  for v in G.vertices() do                                           // O(n)
4      π[v] ← null                                                    // O(n)
5      if v ≠ s then                                                  // O(n)
6          key[v] ← ∞                                                 // O(n)
7          Q.append((key[v], v))                                      // O(n)
8  key[s] ← 0                                                         // O(1)
9  Q.append((key[s], s))                                              // O(1)
10 heapify(Q)                                                         // O(n)

   // seleção das arestas da árvore geradora mínima
11 while len(Q) > 0 do                                                // O(n)
12     (k,u) ← heappop(Q)                  // O(n)·O(log n) = O(n log n)
13     key.pop(u)                                  // O(n)·O(1) = O(n)
14     for v in u.all_neighbors() do               // O(Σ_v d_v) = O(m)
15         if v in key and w[u, v] < key[v] then                      // O(m)
16             decrease_key(Q, v, w[u, v])  // O(m)O(log n) = O(m log n)
17             π[v] ← u                                               // O(m)
18 return π                                                           // O(1)
                                  // Complexidade total: O((n + m) log n)
```

Algoritmo: Kruskal(G,w)

Entrada: Grafo não direcionado G. w[e] é o peso da aresta e.
Saída : Lista E de arestas de uma árvore geradora mínima de G.

```
   // inicialização das estruturas
1  U ← UnionFind(G.vertices())                                        // O(n)
2  E ← set()                                                          // O(1)
3  Q ← [ ]                                                            // O(1)
4  for e in G.edges() do                                              // O(m)
5      Q.append((w[e], (e.source(), e.target())))                     // O(m)
6  heapify(Q)                                                         // O(m)

   // seleção das arestas da árvore geradora mínima
7  while len(Q) > 0 do                                                // O(m)
8      (k,(u,v)) ← heappop(Q)              // O(m)·O(log m) = O(m log m)
9      if U.find(u) = U.find(v) then                                  // O(m)
10         U.union(u,v)      // no max n − 1 uniões: O(n)·O(1) = O(n)
11         E.append((u,v))                                // O(n)·O(1) = O(n)
12 return E                                                           // O(1)
                                  // Complexidade total: O(n + m log m)
```