Algoritmos e Estruturas de Dados

Paulo Netto

28 de Junho de 2020

Sumário

1 Algoritmos

1.1 Definição

Algoritmo é um conjunto de instruções o qual realiza uma tarefa ou mais tarefas. Tais instruções seguem uma sequência lógica de raciocínios ou operações para alcançar um objetivo. Por exemplo, ao seguir uma receita, os ingredientes (dados de entrada) devem passar por procedimentos e modulações (processamento ou instruções lógicas) para chegar ao prato final (tarefa concluída).

1.2 Validação

Para um algoritmo também faz-se necessário, para além de produzir uma resposta, dados parâmetros de entrada, que a resposta seja **correta**. O algoritmo é correto (resolve o problema) se, para toda instância de entrada, a resposta, ou saída, satisfaça todas as propriedades do problema e, também, não produza uma resposta de looping infinito.

Para prova de correção de um algoritmo, há algumas técnicas, são elas:

- **Técnica dos invariantes do laço**: Para algoritmos interativos, que usam laços, a técnica dos invariantes do laço é utilizada para garantir a execução verdadeira antes, durante e depois da execução do algoritmo.
- **Técnica da indução matemática**: Para algoritmos recursivos, é utilizada a técnica de indução matemática no tamanho das instâncias. Esse é um raciocínio que supõe a validade de um enunciado para o primeiro valor de um conjunto e para qualquer elemento "n" do mesmo conjunto.

1.3 Consumo de tempo

Para o cálculo de consumo de tempo, normalmente, a análise é feita em ordem de grandeza. Isso nos dá uma análise aproximada do consumo de tempo, porém, é mais efetiva em termos de refatoração de um algoritmo para melhoria de performance considerável, principalmente se tratando dos piores casos e casos médios.

1.3.1 Notação Big O

O tempo de execução de um algoritmo, se tratando de recursos computacionais, é dado, principalmente, em notação "Big O".

Sejam T(n) e f(n) funções dos inteiros nos reais, diz-se que T(n) é O(f(n)) se existem constantes positivas "c" e "n_0" tais que:

$$T(n) \leq cf(n) para\ todo\ "n"\ \geq n_0 \tag{1}$$

Na prática, considerando "n" o número de operações que um algoritmo faz, a ordem de grandeza, representada pela notação "Big O", desconsidera as "operações constantes", como as operações de inicialização, por exemplo, e

apenas considera o número de operações que o algoritmo deve fazer caso tenha muitos dados para processar. Ao decorrer das definições de estruturas de dados, a notação "Big O" ficará mais exemplificada e dará parâmetros para dizer se algo é ou não eficiente.

A notação "Big O" considera sempre o o pior caso. Isto quer dizer, dado uma lista de valores, implementando um algoritmo de busca de elemento, sendo esse algoritmo de pesquisa simples, a ordem "Big O" é dada como O(n), sendo "n" o número de elementos da lista. Explicando com mais detalhes, pode ser que, ao ser passada uma chave de busca por um elemento, o algoritmo o encontre "de primeira", porém, considera-se o pior caso, ou seja, como se o elemento fosse encontrado só na última operação.

1.4 Divisão e Conquista

A técnica de divisão e conquista é utilizada por muitos algoritmos que possuem uma estrutura recursiva. Recursão é o termo utilizado para funções que "chamam" elas mesmas. Quando se escreve uma função recursiva, deve-se informar quando a recursão deve parar, para que não ocorra casos de looping infinito. Por isso, toda função recursiva tem duas partes: o **caso base** e o **caso recursivo**. O caso base é quando a função não chama a sí mesma novamente, de forma que o programa não se torna um loop infinito. Já o caso recursivo é quando a função chama a sí mesma.

A técnica de divisão e conquista consiste, basicamente, em dividir o problema recursivamente e, obtendo soluções para os problemas menores, encontra-se a solução para o problema original. Ou seja, a técnica é uma maneira de pensar sobre um problema, e não um algoritmo fechado e genérico que possa ser aplicados em muitos casos.

Há alguns passos para a técnica, e são estes, apresentados abaixo, para cada nível de recursão:

- **Divisão**: O problema é dividido em subproblemas menores, tendo como objetivo, minimizar o tempo de execução.
- **Conquista**: Os subproblemas são resolvidos recursivamente e subproblemas pequenos são resolvidos diretamente.
- **Combinação**: As soluções dos subproblemas são combinadas, também tendo como objetivo, diminuir o tempo de execução.

Em geral, o consumo de tempo para um algoritmo que segue a técnica de divisão e conquista se dá pela equação abaixo, dado "n" como tamanho de entrada:

$$T(n) = Dividir(n) + Conquistar(n) + Combinar(n) \tag{2}$$

Alguns algoritmos de destaque, muito utilizados na prática, são o **Mergesort**, **Quicksort** e o **Heapsort**. Para entender tais algoritmos, faz-se necessário entender as estruturas de dados para os quais são utilizados, algumas dessas estruturas são explicadas com mais detalhes na seção 2.

1.4.1 Mergesort

A ideia sobre o mergesort é a de unir (merge) e ordenar (sort). Assim, o algoritmo tem por objetivo, primeiramente, separar os elementos até o caso-base e, após isso, os unir novamente. Exemplificando: dado uma lista desordenada de 8 elementos, o mergesort divide, por recursão, a lista original em até 8 sublistas. Assim, no processo contrário, o algoritmo retorna à sublista anterior uma nova sublista já com os elementos ordenados. Como em um processo de ida e volta, no qual, na ida, os elementos são divididos em sublistas e, na volta, os elementos vão retornando as sublistas acima, só que agora, ordenados.

Nota-se que, ao retornar uma sublista ordenada, ambas as sublistas anteriores à ela devem ser verificados elemento por elemento, pois, a ordenação anterior havia sido feita apenas considerando os elementos pertencentes as sublistas precedentes à elas. Tal algoritmo pode ser "visualizado" com o mesmo escopo de uma árvore, vista na sessão de estrutura de dados.

1.4.2 Quicksort

Quicksort é um algortimo de ordenação rápido e eficiente. O método se baseia em escolher, dentro de uma lista, um elemento pivô e, com isso, ordenar os elementos menores que ele à esquerda e, também, os maiores elementos à direita. A escolha do elemento pivô pode variar, mas, como exemplo, pode-se utilizar do último elemento da lista como pivô. Assim, percorre-se os elementos da lista, trocando as posições recursivamente dos elementos de maior valor e inserindo o último valor, valor pivô, entre o espaço determinado para que os elementos à esquerda sejam menores que o pivô e os elementos à direita sejam maiores que o pivô.

Nota-se que, fazendo esse procedimento, a lista ainda não fica ordenada em sua plenitude e sim, apenas em relação ao pivô. Assim, todos os elementos à esquerda do pivô formam uma sublista, assim como os da direita. Dessa sublista à esquerda, repete-se o procedimento, seleciona o pivô como último elemento e verifica dentre os outros elementos da mesma sublista qual são os maiores e menores elementos que ele. Faz-se o mesmo para a sublista à direita até que não haja mais elementos não ordenados dentro da lista original.

Em resumo:

- O quicksort também utiliza a estratégia de divisão e conquista, assim como o mergesort;
- Em média, o quicksort é muito mais rápido que outros algoritmos, mas é lento no pior caso;
- O passo da divisão é a parte crítica do algoritmo, nele, o vetor A[p...r] é particionado devolvendo um índice q tal que: $A[p...q-1] \leq A[q] \leq A[q+1...r]$;
- O elemento x=A[q] é chamado de pivô.

No pior caso, o algoritmo quicksort é quadrático, ou seja, caso ele deixe zero elementos do lado esquerdo e n-1 elementos do lado direito, ou o contrário, tendo tempo de execução de O(n^2). No caso médio e melhor caso, o algoritmo tem tempo de execução de $O(n \log n)$.

1.4.3 Comparação Mergesort e Quicksort

Em notação "Big O", como explicado, o tempo de execução, ou desempenho, é estimado pela ordem de operações que o algoritmo produz, levando em consideração "n" elementos na estrutura, pior ou melhor caso, além do caso médio. Assim, dentro dessa lógica, os valores de constantes, entendendo as constantes como o número de operações fixos para qualquer tamanho de estrutura e entradas, são desconsiderados. Porém, em alguns casos, as constantes são significativas para o desempenho final do algoritmo e, comparando o mergesort com o quicksort, o quicksort tem desempenho superior por ter uma constante menor. Em geral, o quicksort opera, na maior parte dos casos, no caso médio, com tempo de execução de $O(n \log n)$, que é o mesmo tempo de execução do mergesort, porém, este último, possui a constante maior e pode ter maior recorrência sobre os piores casos.

1.4.4 Heapsort

O heapsort ilustra o uso de estrutura de dados no projeto de algoritmos eficientes. A estrutura do vetor é parecida com a da árvore binária, estrutura conceituada na sessão de estrutura de dados, na qual, resumidamente, o nó pai, posição "i", possui dois filhos, um na posição $2i + 1$ e outro na posição $2i + 2$. A ideia de um heapsort para ordenar um vetor A[1,...,n], usando um heap, é:

- Construir um heap máximo;
- Trocar a raíz com o elemento da última posição do vetor;
- Diminuir o tamanho do heap em 1;
- Rearranjar o heap máximo, caso seja necessário.
- Repetir os três últimos passos -1 vezes.

1.5 Limite inferior para o problema de ordenação

1.5.1 Introdução

A notação "Big O" é utilizada para indicar o limite superior, ou cota superior, de problemas. Esta, por sua vez, denota o tempo de melhor algoritmo conhecido para o problema. Assim, a cota superior se torna mutável caso alguém descubra um algoritmo melhor. Já a cota inferior é o número de operações mínimas que qualquer algoritmo requer para resolver um problema. Se um algoritmo atinge a cota inferior, diz-se que o algoritmo é assintoticamente ótimo.

Existem algoritmos que podem ordenar "n" números no tempo $O(n \log n)$, são estes, o mergesort e heapsort, que alcançam esse limite superior no pior caso, e o quicksort, que alcança no caso médio. Todos estes são considerados algoritmos baseados em comparações.

Algoritmos baseiam-se em comparações se o fluxo de controle do algoritmo para um vetor depende apenas do resultado das comparações entre os elementos do vetor. Qualquer algoritmo que seja baseado em comparações deve efetuar, no pior caso, $O(n \log n)$. Isso implica em dizer que os algoritmos Mergesort e heapsort são assintoticamente ótimos.

Para melhor representação da cota inferior, um exemplo utilizando a árvore binária de decisão irá ser utilizado.

1.5.2 Árvore de decisão

Seja "A" um algoritmo de ordenação arbitrário e $a_1[...]a_n$ uma sequência arbitrária de números que são uma entrada de "A". Os nós internos da árvore são rotulados por pares de elementos da sequência, $a_i : a_j$, e representam uma comparação feita pelo algoritmo. Cada uma das arestas do nó é rotulada por $\leq$, menor ou igual à esquerda, e $>$, maior à direita. Cada folha é rotulada por uma permutação de 1 a "n" que é a permutação produzida pelo algoritmo como saída, quando essa folha é atingida.

A árvore indica a ordem em que os elementos da sequência são comparados durante a execução do algoritmo "A". O rótulo da raíz da árvore corresponde à primeira comparação efetuada pelo algoritmo. A subárvore esquerda da raíz descreve as comparações subsequentes supondo que $a_i \leq a_j$.

Suponha que a árvore tenha três elementos, 55, 75 e 31. A primeira comparação feita pelo algoritmo de ordenação é se 55 é menor ou igual a 75. Assim, a próxima comparação feita, no nó à esquerda ao nó raíz, será se 75 é maior ou igual a 31. Por fim, o algoritmo compara se 55 é maior ou igual a 31. Terminada as comparações, o vetor fica $31(a_3)$, $55(a_1)$ e $75(a_2)$. Isso exemplifica a ordem de comparações feitas por um algoritmo de ordenação por inserção.

Assim, para um "n" arbitrário, espera-se, para o algoritmo em questão, n! permutações. Se uma permutação não aparece, então o algoritmo não consegue ordenar a correspondente sequência de entrada. O número de comparações que o algoritmo faz, no pior dos casos, é exatamente a altura da árvore de decisão. Se for calculado um limitante inferior para a altura da árvore de decisão do algoritmo, tem-se um limitante inferior para a complexidade do pior caos do algoritmo.

1.5.3 Cota inferior para ordenação

Utilizando o argumento da prova da conta inferior, tem-se que o número de folhas de uma árvore binária de decisão de altura "h" é menor ou igual a 2^h. Além disso, toda árvore que representa um algoritmo de ordenação deve ter, pelo menos, n! folhas, seguindo a sequinte equação:

$$n! \leq numero\ de\ folhas \leq 2^h \tag{3}$$

Assim, aplicando o logaritmo na equação e utilizando suas propriedades, tem-se que, resumidamente, para o pior caso, o algoritmo de ordenação baseado em comparações faz $O(n \log n)$ comparações.

1.6 Ordenação Linear

1.6.1 Introdução

Qualquer algoritmo baseado em comparações deve efetuar $O(n \log n)$ comparações no pior caso. Isso implica que o mergesort e heapsort são assintoticamente ótimos. Já os algoritmos de **ordenação por contagem**, Radixsort e Bucketsort são executados e tempo linear. Esses algoritmos utilizam operações diferentes da operação de comparação para determinar a sequência ordenada. Logo, o limite inferior $O(n \log n)$ não se aplica à eles.

1.6.2 Ordenação por contagem

Supondo que é sabido que os elementos aos quais deseja-se ordenar são inteiros, dispostos no intervalo de [0, k]. O conceito da ordenação por contagem incide sobre a ideia básica de determinar, para cada elemento "x" do vetor, o número de elementos menores que "x". Assim, essa informação pode ser utilizada para inserir o elemento "x" diretamente em sua posição de saída.

Seja a entrada de um vetor A[1, ..., n] e a saída é um vetor B[1, ..., n] com os elementos de A[1, ..., n] em ordem crescente. O algoritmo utiliza um vetor auxiliar "C" para contar quantos números tem de cada um para determinar a posição em que os elementos devem ser colocados. Após isso, acumula-se os números no vetor "C", assim, C[i] terá o número de elementos da entrada que são menores ou iguais a "i". Por fim, o algoritmo faz a leitura do vetor inicial "A" e, com o auxílio do vetor "C", que determina quantos elementos de cada valor possui dentro do vetor "A", inserindo-os em ordem dentro de um vetor "B" inicialmente vazio.

O algoritmo de ordenação por contagem é estável, números com o mesmo valor aparecem no vetor de saída na mesma ordem em que aparecem no vetor de entrada. Seu consumo de tempo é, em geral, O(n).

1.6.3 Radixsort

Supondo que é sabido que os elementos que deseja-se ordenar tem "d" digitos. A ideia básica é ordenar em função dos digitos, um de cada vez, começando pelo digito menos significativo. Desse modo, apenas "d" passagens pela lista são necessárias para fazer a ordenação. Assim, tendo um vetor "A" com elementos com três dígitos significativos, o algoritmo inicializa a ordenação pelo primeiro dígito de cada elemento, o dígito da casa dos algarismos. Após isso, faz-se o mesmo procedimento para o dígito da casa das dezenas e, por fim, da casa das

centenas. O algoritmo de radixsort é dependente de um algoritmo intermediário, assim, seu tempo é dependente deste algoritmo auxiliar. Se cada dígito está no intervalo de 0 a k-1 e "k" não é muito grande, o algoritmo de ordenação por contagem é viável.

1.6.4 Bucketsort

Suponha que os elementos que deseja-se ordenar foram gerados aleatoriamente de forma uniforme no intervalo de [0,1). A ideia básica de ordenação desse algoritmo é:

- Dividir o intervalo em "n" buckets;
- Distribuir os "n" números entre os buckets;
- Ordenar os números em cada bucket usando ordenação por inserção

Assim, a ordenação em "buckets", ou baldes, ordena para cada balde uma lista ligada que contém uma faixa de valores. Isso é feito usando um algoritmo auxiliar de ordenção, ou ele mesmo, e, assim, percorre-se os baldes e colocam-se os valores de cada balde de volta no array ordenado.

Para o melhor caso, o bucketsort compõe-se de $O(K + N)$, onde "K" é o número de baldes e "N" os elementos. Para o pior caso, o algoritmo possui o tempo de $O(N^2)$, em que todos os elementos estão dispostos em um mesmo balde.

1.7 Grafos

Os conceitos sobre grafos estão dispostos na seção de estrutura de dados para fins organizacionais desse conteúdo. Cabe, nessa seção, algumas discussões sobre alguns algoritmos e operações que podem ser feitos nesse tipo de estrutura.

1.7.1 Grafos - Operações básicas

Para as operações abaixo demonstradas, a estrutura utilizada será a de lista de adjacências.

Criação de um grafo sem arestas: Para a criação de um grafo sem arestas, com número fixos de vértices, deve-se alocar espaço para a estrutura e inicializar o total de vértices e arestas. Dentro de um grafo, do tipo lista de adjacências, há apenas um ponteiro para o arranjo de vértices, não o arranjo em sí. Assim, deve-se alocar o arranjo com o número de vértices desejado. Uma vez alocado, inicializa-se cada elemento no arranjo com seu adjacente com o valor null. Por fim, retorna-se o grafo.

Inclusão de arestas (adjacência) no grafo: Uma vez criado o grafo, adiciona-se arestas a ele, criando, para isso, adjacências. Assim, os parâmetros serão o vértice final da aresta, além do seu peso. Aloca-se espaço para um nó na lista de adjacências, associa-se a ele o vértice final da aresta e seu peso.

Uma aresta só é criada quando adiciona uma adjacência à lista de um vértice, definindo seu ponto inicial e final.

Cria-se uma ajdacência na memória, insere-se no início da lista de adjacências e incrementa-se o número de arestas.

Visualização do grafo: Para visualizar o grafo, cria-se um cabeçalho geral, para cada vértice no grafo mostra-se a sua lista de ajdacências.

1.8 Grafos - Busca em Profundidade

Fazer buscas em um grafo significa seguir suas arestas sistematicamente, de modo a visitar seus vértices.

1.8.1 Grafos - Depth-first search (DFS)

A estrategia seguida pela busca em profundidade (Depth-first search ou DFS) é buscar mais fundo no grafo sempre que possível. A busca só para quando encontrar o que foi requisitado ou após todos os vértices terem sido varridos.

Na busca em profundidade, as arestas são exploradas a partir do vértice "v" mais recentemente descoberto que ainda tem arestas não exploradas saindo dele. Quando todas as arestas de "v" são exploradas, a busca volta ao vértice anterior a "v" (backtracking), para seguir as arestas ainda não exploradas. Este processo continua até que tenhamos descoberto todos os vértices que são atingíveis a partir do vértice inicial. Um vértice "v" é atingível a partir de um vértice "u" se houver um caminho de "u" e "v" no grafo. Se restarem ainda vértices não visitados, um é selecionado e reinicia-se a busca a partir dele.

O funcionamento de um DFS (Depth-first search) funciona definindo um nó inicial, escolhendo um dos seus adjacentes não visitados, então percorrendo-o. Assim, repete-se o processo até atingir o nó objetivo ou um nó cuja adjacência já tenha sido toda visitada (nó final). Caso o objetivo não tenha sido atingido, volta-se ao nó pai e continua-se a busca de um nó irmão ainda não visitado.

1.8.2 Grafos - Breadth-first search (BFS)

A busca em largura (Breadth-first search (BFS)) é um método de busca que se aplica aos grafos. Diferentemente da busca em profundidade, a busca em largura "visita" todos os adjacentes de um nó, para então visitar os adjacentes dos adjacentes, e assim por diante.

Dado um nó inicial "s", a busca determina a distância (em número de arestas) de cada vértice atingível do grafo "s". Vértices a uma distância de k + 1 arestas de "s" só são visitados após todos os vértices a uma distância de "k" arestas terem sido visitados.

Os nós a serem visitados são colocados em uma fila, inicialmente contendo apenas o nó inicial. Na medida em que se visita um nó, coloca-se os vizinhos, que não estiverem lá, na fila. Assim, continua-se o processo até achar o nó alvo da busca ou a fila ficar vazia. Se a fila estiver vazia e ainda restarem nós não visitados, reinicia-se o procedimento a partir de um destes.

Como procedimento de algoritmo, defina inicialmente um nó, marcando-o como explorado. Enfileire-o e, enquanto a fila não estiver vazia, remova o primeiro nó da fila, "u". Para cada vizinho "v" de "u", caso "v" não tenha sido explorado, marque "v" como explorado e o coloque no fim da fila. Caso haja outros nó não fortemente ligados, repita o procedimento de inicialização e varredura para os outros. Os grafos desacoplados, em forma de uma lista ligada, assemelham-se a uma árvore de busca.

Uma característica da BFS é que sua árvore de busca corresponde ao caminho mais curto do nó inicial a qualquer vértice do grafo, no caso, o menor número de arestas.

1.8.3 Grafos - Algoritmo de Dijkstra

Para o caso anterior, a busca em largura dava o menor caminho entre um vértice inicial e todos os demais no grafo. O caminho era medido em número de arestas, ignorando quaisquer pesos que estas possam ter. Todavia, para vários problemas, o peso nas arestas é crucial. Assim, o **Algoritmo de Dijkstra** calcula o caminho mais curto, em termos do peso total de arestas, entre um nó inicial e todos os demais nós no grafo. O peso total das arestas é a soma dos pesos das arestas que compõem o caminho.

Para cada vértice "v" do grafo, mantém-se um atributo d[v], que é um limite superior para o peso do caminho mais curto do nó inicial "s" a "v". Diz-se que d[v] é uma estimativa de caminho mais curto, inicialmente feito ∞. Armazena-se também o vértice que precede "v" (p[v] - precedente de "v") no caminho mais curto de "s" a "v".

Para exemplificar o algoritmo:

Faça a estimativa de distância de "s" a qualquer vértice ser infinita, exceto a distância de "s" a "s", que é zero. Ou seja, d[s] = 0 e d[v] = ∞ para todo "v" $\neq$ "s".

Assim, faça também que os precedentes de um nó sejam um valor qualquer. Na prática, pode-se fazer p[v] = -1, isso evita possíveis ambiguidades de valores nos vértices, visto que não há vértice de índice -1 na estrutura **grafo**. Além disso, marque todos os vértices como "abertos", ou seja, o valor de "d" é uma estimativa.

Enquanto houver vértices abertos, escolha o vértice aberto "u", cuja estimativa seja a menor dentre os demais abertos e "feche-o". Para o caso inicial, considerando que as estimativas dos vértices não iniciais foram de ∞, o nó inicial será o menor entre todos os abertos. Assim, para todo nó aberto "v" na adjacência de "u", some d[u] ao peso da aresta [u, v]. Caso a soma seja menor que d[v], atualize d[v] e faça p[v] = u. Tal procedimento é chamado de **relaxamento da aresta** (u,v). Sinteticamente, o processo de relaxar uma aresta consiste em testar se é possível melhorar o caminho mais curto para "v" a cada processamento de comparação de um caminho.

Em resumo:

- Inicialize o grafo com d[s] = 0, d[v] = ∞, para todo "v" $\neq$ s, e p[v] = -1, para todo v.

- Faça aberto[v] = true para todo "v" do grafo.

Enquanto houver vértice aberto:

- Escolha "u" cuja estimativa seja a menor dentre os vértices abertos.
- Feche "u".

Para todo nó aberto "v" na adjacência de "u":

- Relaxe a aresta (u, v)

Algoritmo de Dijkstra - Observações: Apenas fecha-se um nó "u" se já é de conhecimento a menor distância deste ao nó inicial "s". Então, faz-se o relaxamento de seus "vizinhos". Sabendo a menor distância de "s" a "u", tem-se a estimativa de "s" a "v", sendo "v" vizinho de "u". E, assim, busca-se saber se a distância de "s" a "u" mais a distâcia (u, v) é melhor que a estimativa atual de "s" a "v". Se for, atualiza-se essa estimativa.

Para esta aplicação exemplificada, como foi, por decisão de projeto, armazenado os predecessores dos nós, basta "olhar" para estes elementos precedentes para obter-se o caminho.

O algoritmo de Dijkstra tem um limite importante, os pesos das arestas devem ser não-negativos. Além disso, a implementação de um algoritmo auxiliar para verificação da menor distância e se existe nós abertos ainda no grafo são de fundamental importância para a velocidade do algortimo. Para uma melhor implementação, recomenda-se o uso com **filas de prioridade**.

1.9 Algoritmos Gulosos

A técnica de algoritmos gulosos consiste em resolver problemas de otimização em que há uma escolha mais "apetitosa", que parece ser a melhor em um primeiro momento, para que se chegue em uma possível melhor solução global do problema. A cada iteração, o algoritmo guloso busca a melhor opção para aquele momento, assim, cada parte, sugerindo sua melhor resolução, pode-se chegar A uma conquista final suficiente.

Características gerais dos algoritmos gulosos:

- Durante o processamento do algoritmo, dois conjuntos são criados: O conjunto do elementos que já foram avaliados e rejeitados, e o dos elementos que foram avaliados e escolhidos;
- Uma função pode ser necessária para verificar se o conjunto avaliado e escolhido é uma solução para o problema;
- Uma outra função pode ser necessária para verificar se há mais itens que otimizariam a solução e não excederiam a mesma;
- Outra função percorre os elementos ainda não processados a fim de obter o melhor elemento para o problema;

- Por fim, existe uma função objetivo, a qual retorna o valor da solução encontrada.

A estratágia gulosa pode não ser efetiva para alguns problemas. Duas propriedades para saber se pode ser utilizado um algoritmo guloso para resolver o problema são a propriedade de escolha gulosa e a subestrutura ótima.

1.9.1 Escolha gulosa

O conceito da propriedade de escolha gulosa é a de achar uma solução localmente ótima, aquela que parece ser melhor naquele momento, para assim achar uma solução globalmente ótima, desconsiderando os resultados de subproblemas.

É importante salientar que há a necessidade de provar que realmente uma escolha gulosa em cada passo resulta em uma solução global. Assim, normalmente se prova isso achando uma solução global para um subproblema e depois modificá-la para uma solução gulosa.

1.9.2 Subestrutura ótima

Quando um problema possui uma solução ótima que nela contém soluções ótimas para subproblemas, diz-se que essa subestrutura é ótima. Para demonstrar que a combinação de uma escolha gulosa, já realizada, com uma solução ótima, para o subproblema, resulta em uma solução ótima para o problema original, induz-se sobre os subproblemas para provar que fazer a escolha gulosa em cada etapa produz uma solução ótima.

1.10 Programação Dinâmica

A programação dinâmica e os algoritmos gulosos trabalham, em geral, com problemas de otimização e com sequência de passos e escolhas. Entretanto, eles se diferem quanto à forma.

A programação dinâmica pode depender das soluções para subproblemas, enquanto, para os algoritmos gulosos, a escolha é feita como a melhor do momento, dependendo apenas das escolhas anteoriormente feitas. Em resumo, a programação dinâmica pode ser entendida como subproblemas resolvidos debaixo para cima, de subrproblemas menores para maiores. Os algoritmos gulosos, por sua vez, partem de cima para baixo, de um problema maior e, a cada iteração, subproblemas menores vão surgindo.

Como em um algoritmo recursivo, na programação dinâmica, cada instância do problema é resolvida a partir da solução de instâncias menores, ou melhor, de subinstâncias da instância original. A característica distintiva da programação dinâmica é a tabela que armazena as soluções das várias subinstâncias. O consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela.

Como exemplo de aplicação de programação dinâmica, irá-se demonstrar, passo a passo, a resolução do **problema da mochila**.

1.10.1 Problema da Mochila

Dado uma mochila que possui capacidade máxima de até 4kgs e dado um conjunto de objetos com um certo peso e valor, qual é o maior valor que se pode carregar na mochila?

Considere três produtos diferentes, cada um possuindo os valores de 1500, 2000 e 3000, respectivamente. Além disso, considere os seguintes pesos, 1kg, 3kgs e 4kgs, respectivamente. A tabela 1 mostra o início do algoritmo:

	1kg	2kgs	3kgs	4kgs
Produto 1				
Produto 2				
Produto 3				

Tabela 1: Escopo inicial do problema da mochila

O algoritmo de programação dinâmica começa por resolver subproblemas, ou seja, considerando a primeira "submochila" com capacidade máxima de 1kg.

A linha do "Produto 1" indica que será testado se esse item cabe na mochila, cada célula terá a seguinte decisão: carregar ou não o produto avaliado na mochila, afim de maximizar o valor dentro dela.

A primeira célula indica a decisão que, para uma submochila de 1kg, o Produto 1, que pesa 1kg, caberá nela? Sendo a resposta sim, na primeira célula é armazenado o valor do produto que já cabe na mochila, como demonstra a tabela 2.

	1kg	2kgs	3kgs	4kgs
Produto 1	1500			
Produto 2				
Produto 3				

Tabela 2: Primeira iteração na primeira célula

Assim, aplicando a mesma decisão as outras submochilas de 2kgs, 3kgs e 4kgs, obtém-se a mesma resposta de que o Produto 1 cabe nelas.

	1kg	2kgs	3kgs	4kgs
Produto 1	1500	1500	1500	1500
Produto 2				
Produto 3				

Tabela 3: Decisões para as submochilas restantes em relação ao Produto 1

Repete-se o mesmo princípio para o Produto 2 e Produto 3. Em cada linha, será possível carregar o item relativo àquela linha e todos os itens das linhas anteriores. Assim, analisando o Produto 2, que pesa 4kgs, e possui o valor de 3000, entende-se que para as submochilas de 1kg, 2kgs e 3kgs, não cabe carregar

esse produto e, o máximo de valor encontrado até agora é ainda o do Produto 1, então repete-se o valor, como visto na tabela 4.

	1kg	2kgs	3kgs	4kgs
Produto 1	1500	1500	1500	1500
Produto 2	1500	1500	1500	
Produto 3				

Tabela 4: Decisões para o Produto 2

Sabendo que, o Produto 2 possui maior valor que o Produto 1 e, este último, era o maior valor anterior para uma submochila com 4kgs, atualiza-se a célula desta submochila, mas agora com o valor do Produto 2.

	1kg	2kgs	3kgs	4kgs
Produto 1	1500	1500	1500	1500
Produto 2	1500	1500	1500	3000
Produto 3				

Tabela 5: Atualização de valor para a submochila de 4kgs

Para o último produto, Produto 3, sabendo que ele pesa 3kgs e tem valor de 2000, a subochila de 3kgs deverá ser atualizada, mantendo as anteriores, como mostra a tabela 6.

	1kg	2kgs	3kgs	4kgs
Produto 1	1500	1500	1500	1500
Produto 2	1500	1500	1500	3000
Produto 3	1500	1500	2000	

Tabela 6: Decisões para o Produto 3

Para a última célula da tabela, sabendo que o Produto 3 tem menor valor que o Produto 2, seria de se esperar que a tabela mantivesse o último valor mais alto. Entretanto, sabendo também que, adicionando o Produto 3 em uma submochila de 4kgs, ainda sobra 1kg a ser ocupado. Assim, tal espaço pode ser ocupado pelo Produto 1. Ou seja, a maximização do valor a ser carregado por uma mochila de 4kgs é o somatório do Produto 1 com o Produto 3, resultando no valor de 3500, como mostra a tabela 7;

Em termos matemáticos, sendo "i", o índice da linha, e "j", o índice da coluna, tem-se que:

	1kg	2kgs	3kgs	4kgs
Produto 1	1500	1500	1500	1500
Produto 2	1500	1500	1500	3000
Produto 3	1500	1500	2000	3500

Tabela 7: Resultado do algoritmo da mochila

$$Celula[i][j] = \begin{cases} 1 - O\ maximo\ anterior(valor\ da\ celula[i-1][j] \\ vs \\ 2 - Valor\ do\ item\ atual\ + \\ valor\ do\ espaco\ restante\ da\ celula[i-1][j - Peso\ do\ item] \end{cases} \tag{4}$$

Este exemplo mostra a diferença entre a programação dinâmica e os algoritmos gulosos. Como é sabido, a programação dinâmica, para o caso da mochila, apresentou a melhor solução dentro das possíveis. Caso tivesse sido utilizado a estratégia gulosa, o item a ser carregado seria o melhor para a ocasião, ou seja, seria tido como melhor benefício o carregamento apenas do Produto 3, que possui valor de 3000 e cabe na última submochila de 4kgs.

1.11 Classes de problemas: problemas P, NP e NP-completos

1.11.1 Introdução

A seção trata de forma simplificada sobre alguns parâmetros de complexidade computacionais envolvendo a ideia sobre problemas simples, complexos, de otimização e de decisão.

1.11.2 Problema de otimização x problema de decisão

Problema de otimização é um problema em que cada solução possível tem um valor associado e deseja-se encontrar a melhor solução com relação a esse valor. Assim, dentre as possíveis soluções, acha-se qual é a melhor ou a mais viável.

Os problemas de decisão tratam de "dilemas" sobre haver ou não uma solução para um dado problema.

É possível formular um problema de otimização como um problema de decisão impondo um limite sobre o valor a ser otimizado. Exemplificando, dado um grafo de dois vértices (u, v), encontrar um caminho com o menor número de arestas possíveis (problema de otimização) ou existe um caminho com menos de dez arestas? (problema de decisão).

1.11.3 Classes P e NP

A teoria da complexidade não se aplica diretamente a problemas de otmização, mas sim, a problemas de decisão.

A classe P é formada por problemas de decisão tratáveis, isto é, que podem ser resolvidos por um algoritmo polinomial. Por exemplo, o problema de um grafo euleriano, estrutura a qual existe um trilho único do começo ao fim e que todos os vértices possuem grau par, pertence à classe P.

A classe NP é formada por problemas de decisão que possuem um verificador polinomial para a resposta "sim". NP denota um conjunto de linguagens que podem ser aceitas em tempo polinomial por uma máquina de Turing não-determinística. Um problema de decisão está em NP se existe um algoritmo A, tal que:

- Para qualquer instância "X" do problema com resposta "sim", existe um certificado "Y", tal que A(X, Y) devolve "sim".
- "A" consome tempo polinomial em —X—.

Como exemplo, um grafo hamiltoniano, grafo ao qual se busca saber se há um circuito simples que passa por todo o grafo, é um problema do tipo NP.

1.11.4 Classe NP-completo

Um problema do tipo NP-completo é um problema para o qual é improvável que exista um algoritmo eficiente que o resolva. Assim, deve-se buscar heurísticas ou aproximações para resolvê-los. Pode ser difícil a compreensão sobre o fato de um problema ser ou não NP-completo, pois a diferença é pequena. O livro "Entendendo Algoritmos", de Aditya Bhargava, exemplifica alguns indicativos para entender esse conceito, são esses:

- O algoritmo possui bom processamento para alguns itens, mas fica muito lento quando a escala de itens aumenta;
- Se o problema trata de todas as possíveis combinações de uma entrada, por exemplo, tratando-se de um problema de escala fatorial;
- O problema não é resolvido ao tentar dividí-lo em suproblemas;
- Se o problema envolve uma sequência (como o problema do caixeiro-viajante, problema tratado no livro);
- Se o problema envolve um conjunto, semelhante a um problema combinacional;

1.12 K-Vizinhos mais próximos

O algoritmo K-Vizinhos mais próximos, do inglês, K-Nearest Neighbor (K-NN), é um um método baseado em distâncias, uma técnica que considera a proximidade entre dados na realizações de predições. Por hipótese, dados similares tendem a estar concentrados na mesma região no espaço de dispersão dos mesmos. Resumidamente, o modelo busca classificar um novo dado que, dentro de um espectro de dados, baseado na menor distância, sendo a mais utilizada a **distância euclidiana**, entre os outros dados já classificados. Assim, se o novo dado possuir maior proximidade com um conjunto de dados já classificados, assume-se que, o novo dado entra nesse conjunto de classificação.

Baseados em métricas, o algoritmo verifica quais são as classes dos K-elementos vizinhos e a classe mais frequente será atribuída à classe do elemento desconhecido. Algumas métricas utilizadas para o cálculo de distâncias entre dois pontos, são apresentadas abaixo:

Distância Euclidiana: A distância euclidiana para "n" espaços é dada pela fórmula:

$$d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 ... (x_n - y_n)^2} \tag{5}$$

Distância Manhattan: A distância de Manhattan para "n" espaços é dada pela fórmula:

$$d(x,y) = |x_1 - y_1| + |x_2 - y_2| ... |x_n - y_n| \tag{6}$$

Distância Minkowski: A distância de Minkowaski para "n" espaços é dada pela fórmula:

$$d(x,y) = (|x_1 - y_1|^q + |x_2 - y_2|^q ... |x_n - y_n|^q)^(1/q) \tag{7}$$

A distância de Minkowski é a generalização das outras duas anteriormente apresentadas, ou seja, para o valor de q = 1, a distância representada é a de Manhattan, já para q = 2, a distância é euclidiana.

Se cada variável possui um peso relativo, pode-se escrever a distância euclidiana ponderada como:

$$d(x,y) = \sqrt{w_1(x_1 - y_1)^2 + w_2(x_2 - y_2)^2 ... w_n(x_n - y_n)^2} \tag{8}$$

O algoritmo de k-vizinhos próximos possui um classificador livre, ou seja, o usuário deve escolher quantos k-vizinhos o novo dado irá ser comparado. Para alguns casos, é de boa prática escolher k-vizinhos ímpares, para que a classificação não "dê empate" em relação as distâncias. Para outros casos também, se o conjunto de k-vizinhos é muito grande, a estratégia do algoritmo pode ser inviável computacionalmente.

1.13 Regressão Linear

Para dois conjuntos de dados correlacionados, a regressão linear é a estimativa calculada de uma reta, que, de certa forma, mostra o comportamento dos dados analisados. Assim, em termos gerais, dentro de um conjunto de dados que possuem correlação, a regressão linear busca uma função que melhor "descreve" este conjunto, assim, é possível utilizar a função, variando os valores da variável independente, para prever resultados aproximados para a variável dependente.

Com base nos dados, $(x_1, y_1), (x_2, y_2)...(x_n, y_n)$, constroi-se o digrama de dispersão, que exibe a tendência linear para que se possa usar a regressão linear. O diagrama permite decidir:

- Se um relacionamento linear entre as variáveis X e Y deve ser assumido
- Se o grau de relacionamento linear entre as variáveis é forte ou fraco, conforme o modo como se situam os pontos em redor de uma reta imaginária que passa através do conjunto de pontos.

O **coeficiente de correlação de Pearson** é o parâmetro que informa quão fortemente duas variáveis estão relacionadas. Essa variação pode ser em um intervalo de -1 a +1, sendo que, quanto mais perto de -1, os dados são mais fortemente relacionados com uma reta com coeficiente angular negativo e, quanto mais perto de +1, mais fortemente ligada com a reta positiva.

A forma matemática a qual descreve a regressão linear é dada pela fórmula:

$$y_i = \alpha + \beta X_i + \varepsilon_i \tag{9}$$

Onde:

- y_i é a variável dependente;
- α é o coeficiente linear;
- β é o coeficiente angular;
- X_i é a variável independente;
- ε_i que representa os fatores residuais. Estes podem ser advindos de erros e pontos onde a correlação da variável dependente e independente não condizem com fidelidade a tendência linear da função. Para que a função seja ainda fortemente correlacionada, os erros devem satisfazer condições de distribuição normal, variância, entre outros fatores.

2 Estrutura de Dados

Conceitua-se como estrutura de dados um campo onde se armazenam dados e, a partir da organização dos mesmos, é possível instanciá-los, acessá-los e também modificá-los.

2.1 Listas

2.1.1 Listas Lineares

Uma lista é um arranjo de elementos ordenados em que cada elemento possui apenas um antecessor, exceto o primeiro número, e, também, só possui um sucessor, com exceção do último.

2.1.2 Listas Lineares Sequenciais

É a lista cuja ordem lógica dos elementos coincide com a posição física (em memória).

A forma mais comum de implementação de listas é com um vetor do tipo REGISTRO, tendo o tamanho máximo especificado. Registro é toda a informação que o usuário está armazenando, pode ser um nome, um CPF, um endereço, etc.

Assim, o vetor é conjugado a um contador de números de posições, efetivamente ocupadas, ou seja, dado um vetor com "nroElement", nroElement-1 indica o último elemento existente a estrutura.

Vantagens:

- Acesso direto a qualquer elemento com base no seu índice, caso seja conhecido.
- Caso seja ordenada, a busca por uma chave pode ser do tipo binária ($\log N$).

Busca Binária: A busca binária é um eficiente algoritmo para encontrar um item em uma lista ordenada de itens. Ela funciona dividindo repetidamente pela metade a porção da lista que deve conter o item, até reduzir as localizações possíveis a apenas uma.

Desvantagens:

- Mesmo em uma estrutura ordenada, o pior caso de inserção e exclusão (na frente da lista) exige movimentação de todos os "n" elementos da lista, assim, para casos onde há muitas atualizações, tal estrutura pode ser inadequada.
- Implementação estática, exigindo que o tamanho do vetor seja previamente estabelecido.

Funções de Gerenciamento

Algumas funções de gerenciamento são importantes destacar, não só para listas, como também para outras estruturas de dados, tais como, a inicialização da estrutura, o retorno da quantidade de elementos válidos, a exibição dos elementos, a busca por um elemento, inserir elementos, excluir elementos e reinicializar a estrutura.

Inicialização: Para inicializar uma estrutura qualquer, faz-se necessário pensar nos valores adequados para cada campo. Para o caso da lista sequencial,

caso já tenha sido criada pelo usuário, basta apenas colocar o valor 0 no número de elementos válidos da estrutura.

Retornar o número de elementos: Para o caso abordado, em que contém um campo que consta o número de elementos, basta apenas retornar este dado.

Exibição dos elementos: Para a exibição de todos os elementos, basta criar um laço que itera sobre a sequência e, assim, usar um comando de impressão para sua exibição.

Busca por um elemento: O contexto de busca por um elemento deverá constar a chave do elemento, enviado pelo usuário e, caso seja encontrado, deverá ser retornado à posição do mesmo ou retornar algum valor inválido, no caso de não se ter esse registro na estrutura. Para esse caso também, um laço de iteração deverá ser passado na lista, comparando se a chave mandada pelo usuário coincide com a chave que consta nos elementos. Em resumo, a busca por um elemento, desse modo, passa por duas verificações, uma em relação à validade do índice passado pelo usuário e outra se a chave buscada corresponde ao elemento da lista no laço de iteração.

Para uma busca mais otimizada, utiliza-se o recurso de inserção da chave que o usuário passou para um novo campo, podendo ser chamado de campo, ou unidade, de sentinela. A criação desse novo registro, inserido no final da lista, garante que a chave que o usuário passou estará na lista e, se a iteração passar por todos os elementos e só achar o elemento buscado no campo sentinela, então sabemos que não há o elemento no arranjo. Se a lista já estiver cheia, não haverá espaço para a criação do sentinela, assim, cria-se uma lista com uma posição extra (um registro a mais) para garantir que haverá espaço para o sentinela. É válido lembrar também, que essa posição extra nunca terá um registro válido.

Um outro modo de busca mais eficiente, seria a busca binária. Porém, essa busca só é válida para os casos em que a lista é ordenada, ou seja, o modo de inserção dos valores na lista deverá iterar sobre os valores a fim de descobrir qual é a posição exata de inserção (comparando os valores e, ao achar o primeiro valor maior que o do registro inserido, inserir o elemento logo atrás dele).

Inserção dos elementos: O usuário passa um registro a ser inserido na estrutura e, para o caso de listas, há diferentes posições de inserção, no início, no fim, ordenado por uma chave ou em uma posição indicada pelo usuário. Além disso, antes da inserção, deve-se validar se a lista está cheia e se o índice, passado pelo usuário, é válido. Após a validação, desloca-se todos os elementos daquela posição para a direita e, assim, insere-se o do elemento na posição e um número deve ser acrescido no campo "nroElem", mantendo atualizado o número de elementos da lista.

Exclusão de um elemento: Para a exclusão de um elemento, o usuário passa a chave do elemento que ele quer excluir, assim, valida-se se há o elemento com esta chave na lista, exclui-se esse elemento, desloca-se todos os elementos posteriores uma posição para a esquerda e, por fim, diminui-se um elemento do "nroElem".

Reinicialização da lista: Reinicializar uma estrutura é deixá-la pronta para o usuário reutilizar, sem que haja mais nenhum elemento nela. Para o caso, como o arranjo já está previamente alocado, basta inserir o valor 0 no

campo "nroElement".

2.2 Listas Lineares Ligadas (ou Encadeadas)

Para listas sequenciais ordenadas, como visto anteriormente, há uma grande movimentação de dados com operações de inserção e exclusão de elementos, visto que, para se manterem ordenadas, necessitam manter a ordem lógica com a ordem física dos dados.

Diferentemente da lista sequencial, a lista ligada, ou encadeada, permite que os elementos fiquem dispostos em qualquer posição disponível, não necessariamente na posição física ordenada, e, para se manter a ordem lógica dos mesmos, é utilizado um vetor (para forma estática) ou, em linguagens de programação que oferecem suporte à alocação dinâmica, é utilizado os ponteiros.

2.2.1 Listas Ligadas de Implementação Estática

Uma lista ligada com implementação estática, deve ser formada por um vetor de registros (A), um indicador de início da estrutura (início) e um indicador de início da lista de nós disponíveis (dispo). Na prática, início e dispo são as entradas de duas listas que compartilham o mesmo vetor, sendo uma para os elementos efetivos da lista, e a outra para armazenar as posições livres. Além disso, cada registro contém, além dos campos exigidos pela aplicação, um campo prox que contém um índice para o próximo elemento na série.

Cada registro possui, além dos campos exigidos pela aplicação, um campo, do tipo índice, que aponta para o próximo elemento da série (sendo este sucessor vazio ou ocupado). Um índice com valor -1 é indicado para designar que o elemento não possui sucessor.

Em resumo, uma lista, com um valor inicial de -1, que indica ser uma lista vazia, e com um índice, de valor inicial 0, que indica que a primeira posição do vetor está vazia, e com o índice que gerencia os apontamentos da ordem lógica dos dados, é considerada ligada com implementação estática. A lista é considerada cheia quando o índice de nós disponíveis (entendendo nós como o conjunto unitário do dado mais o seu índice "apontador") é igual a -1 e, quando vazia, o índice de início é -1, indicando não haver um elemento inicial.

O gerenciamento de nós livres e ocupados exige a criação de rotinas específicas para "alocar" e "desalocar" um nó da lista apontada por dispo. A alocação envolve descobrir o índice de uma posição válida no vetor na qual novos dados possam ser inseridos, além de retirar esta posição da lista de disponíveis. A desalocação envolve a devolução de uma posição à lista de disponíveis para que possa ser reutilizada. Assim, rotinas de alocação/desalocação não devem ser chamadas sem que sejam seguidas da correspondente inserção/exclusão, pois haverá perda de dados e a estrutura se tornará inconsistente.

Inicialização: Para iniciar a lista ligada, faz-se necessário colocar todos os elementos em uma "lista" de disponíveis, acertar a variável que indica o primeiro item disponível e acertar a variável que indica que não há nenhum item válido.

Retornar o número de elementos: Para retornar o número de elementos ou inserimos uma variável a mais, como a variável nroElem, e a cada inserção, ou exclusão, aumentamos, ou diminuímos, o valor dela. Ou, ainda, podemos desconsiderá-la, mas a cada requisição do tamanho da lista devemos iterar sobre todos os elementos válidos para contar quantos são.

Exibição dos elementos: Para a exibição dos elementos, como na lista sequencial não ligada, devemos iterar sobre cada elemento disponível e imprimir suas chaves.

Busca por um elemento: A busca deverá receber uma chave do usuário, retornar a posição em que este elemento se encontra no arranjo, caso seja encontrado, ou retornar inválido, caso não haja um registro válido com essa chave na lista.

Inserção dos elementos: Para os casos de uma inserção ordenada pelo valor de chave do registro passado, deve-se achar os elementos, sucessor e antecessor, do elemento passado pelo usuário e, assim, insere-se esse elemento no "meio" dos dois, acertando os ponteiros, dentro de uma perspectiva lógica, não física. Em casos de listas ordenadas, o programador deverá prestar atenção em duas possibilidades, no caso elementos repetidos poderem ou não serem inseridos na lista, isso altera algumas estruturas em termos de código.

Exclusão de um elemento: Para excluir um elemento, o usuário passa a chave de registro e, se houver um elemento com esta chave na lista, exclui-se o elemento e o insere-o na lista de disponíveis, acertando os ponteiros da lista ligada.

Reinicialização da lista: Para reinicializar a lista, basta chamar a função de inicialização da mesma.

2.2.2 Listas Ligadas de Implementação Dinâmica

Para linguagens de programação que possuem recursos de alocação dinâmica de memória, em que o gerenciamento de nós livres/ocupados fica à cargo do ambiente de programação, não se tem a necessidade da definição prévia do tamanho de uma lista, como no caso da implementação estática.

Em uma lista ligada de implementação dinâmica, não há mais uso de vetores. Cada elemento da lista é uma estrutura do tipo NO, que contém os dados de cada elemento (inclusive a chave) e um ponteiro prox para o próximo nó da lista. Um nome auxiliar (estrutura) é usado para permitir a auto-referência ao tipo NO que está sendo definido.

O tipo LISTA propriamente dito é simplesmente um ponteiro "início" apontando para o primeiro nó da estrutura (ou para null no caso da lista vazia). O último elemento da lista possui seu ponteiro prox também apontando para null.

A alocação e desalocação de nós é feita dinamicamente pelo compilador da linguagem de programação utilizada, através de primitivas que criam um novo nó em memória e também liberam um espaço de memória, usando como recurso um apontador.

A única diferença significativa entre as implementações estática e dinâmica de listas ligadas está no fato de que a implementação estática "simula" uma

lista ligada em vetor, e nos obriga a gerenciar as posições livres e ocupadas. Isso deixa de ser necessário no caso da alocação dinâmica "real".

Inicialização: Para a inicialização, como não existe um arranjo pré definido, basta inicializar a lista ligada com um ponteiro para um registro igual a null, indicando que não há, ainda, nenhum registro válido na lista.

Retornar o número de elementos: Da mesma forma que a lista anterior, não havendo um campo com o número de elementos na lista, faz-se necessário percorrer todos os elementos para contar quantos são.

Exibição dos elementos: Para a exibição dos elementos, como na lista sequencial ligada estática, deve-se iterar sobre cada elemento disponível e imprimir suas chaves.

Busca por um elemento: A busca deverá receber uma chave do usuário, retornar a posição em que este elemento se encontra no arranjo, caso seja encontrado, ou retornar null, caso não haja um registro válido com essa chave na lista.

Inserção dos elementos: Na inserção de elementos, o usuário passa como parâmetro um registro a ser incluído. Assim, insere-se ordenadamente o valor da chave do registro passado, verifica-se sobre quais elementos o registro deverá ficar apontado, entre o seu sucessor e antecessor, acerta-se os ponteiros e um novo espaço de memória é alocado. Lembrando que, para as regras de inclusão de registros únicos, deve-se fazer a validação se há ou não o elemento já registrado na lista.

Exclusão de um elemento: Na exclusão, o usuário para a chave do elemento que quer excluir, se houver um elemento com esta chave na lista, exclui-se este elemento e acerta-se os ponteiros. Caso contrário, retorna-se falso. Vale lembrar que, para esta função, é necessário saber quem é o predecessor do elemento a ser excluído, a fim de ordenar os ponteiros.

Reinicialização da lista: Para reinicializar a lista, é necessário excluir todos os seus elementos e atualizar o campo de início para null.

2.2.3 Listas dinâmicas com nó sentinela

O nó sentinela, criado ao final de uma lista, é usado para armazenar a chave de busca e assim acelerar o respectivo algoritmo, uma vez que reduz o número de comparações necessárias pela metade.

2.2.4 Listas dinâmicas com nó cabeça e circularidade

O nó cabeça, criado no início de uma lista, é usado para simplificar o projeto dos algoritmos de inserção e exclusão. Pode também armazenar a chave de busca (funcionando como nó sentinela) quando a lista for circular. Para os casos de listas duplamente encadeadas, o que muda é que se tem tanto um apontador para o próximo elemento, quanto um apontador para um elemento antecessor.

A circularidade é em geral exigência da aplicação (que precisa percorrer continuamente a estrutura) mas pode também facilitar inserções e exclusões quando combinada com uso de um nó cabeça.

Em resumo, o nó "cabeça" sempre estará como apontador do ínicio da lista, e, pelo conceito da circularidade, sempre receberá o apontamento do último elemento.

Inicialização: Para a inicialização da lista ligada circular, com nó cabeça, é necessário criar o nó cabeça, a variável cabeça deve apontar para ele e o nó cabeça apontará para ele mesmo como o próximo.

Retornar o número de elementos: Para retornar o número de elementos válidos na lista, faz-se necessário a iteração dos elementos da lista, nos caso em que não é utilizado um campo "nroElem" como nos casos anteriores. Vale ressaltar que o nó cabeça não é um dos elementos válidos da lista, assim, nas estrutura da programação, deve-se levar em conta tal condição.

Exibição dos elementos: Para a exibição dos elementos, faz-se necessário, também, a iteração sobre os elementos, seguida do comando de impressão de suas chaves. E, como no caso anterior, deve-se também ressaltar que o nó cabeça não faz parte dos elementos válidos da estrutura.

Busca por um elemento: Para a busca por um elemento, o usuário passa a chave a ser buscada e, caso encontrada, retorna-se o endereço em que o elemento se encontra, ou retorna null caso não seja encontrada. Para estes casos, o nó cabeça pode funcionar como sentinela.

Inserção dos elementos: Na inserção de elementos, o usuário passa como parâmetro um registro a ser incluído. Assim, insere-se ordenadamente o valor da chave do registro passado, verifica-se sobre quais elementos o registro deverá ficar apontado, entre o seu sucessor e antecessor, acerta-se os ponteiros e um novo espaço de memória é alocado. Lembrando que, para as regras de inclusão de registros únicos, deve-se fazer a validação se há ou não o elemento já registrado na lista.

Exclusão de um elemento: Para a exclusão do elemento, o usuário passa a chave do elemento que deseja excluir, se houver um elemento com esse valor de chave na lista, exclui-se o elemento, acerta-se os ponteiros envolvidos e retorna true. Caso contrário, retorna-se false. Para esta função, devemos também saber quem é o predecessor do elemento a ser excluído.

Reinicialização da lista: Para reinicializar a estrutura, faz-se necessário excluir todos os elementos válidos e atualizar ponteiro do sucessor, apontando para o nó cabeça.

2.3 Pilha

Pilhas são estruturas lineares em que as inserções e exclusões ocorrem no topo da pilha, uma lógica semelhante à pilha de papeis, onde retira-se e insere-se papeis apenas no topo.

2.3.1 Pilhas - Implementação Estática

Para implementações estáticas é definido um arranjo de elementos de tamanho predefinido e faz-se necessário o controle da posição do elemento que está no topo da pilha.

Inicialização: Para inicialização de uma pilha, já criada pelo usuário, é necessário apenas registrar um valor no "topo" dela. O valor registrado, tendo em vista que a pilha está vazia, o valor do topo, que indicará a posição no arranjo do elemento, pode ser usado como -1.

Retornar o número de elementos: Partindo do princípio que o campo "topo" contém a posição no arranjo do elemento no topo da pilha, o número de elementos é igual ao "topo + 1".

Exibição dos elementos: Para a exibição dos elementos da estrutura, é necessário iterar sobre os elementos válidos e imprimir suas chaves.

Inserção dos elementos (push): Para inserção de um elemento, o usuário para como parâmetro um registro a ser inserido na pilha e, senão estiver cheia (lembrando que essa é uma aplicação estática), o elemento será inserido no topo da pilha, acima do último elemento que estava no topo.

Exclusão de um elemento (pop): Para o caso de pilhas, a exclusão é feita do elemento do topo da pilha. Assim, se a pilha não estiver vazia, além de excluir o elemento, pode-se condicionar o elemento excluído para um local indicado pelo usuário.

Reinicialização da pilha: Para reinicialização da pilha, é necessário apenas registrar novamente o valor do topo, como, por exemplo, o valor - 1.

2.3.2 Pilhas - Implementação dinâmica

Para implementações dinâmicas não há a necessidade da predefinição de um tamanho fixo do arranjo, ou seja, a alocação de memória é feita sob demanda. Como no caso anterior, faz-se necessário o controle do endereço do elemento que está no topo da pilha e, além disso, agora é necessário um apontador para o elemento sucessor, ou seja, o elemento que está imediamente "abaixo" da pilha. O valor do apontador para o último elemento da pilha tem valor null.

Inicialização: Para inicialização de uma pilha, já criada pelo usuário, é necessário apenas registrar um valor no "topo" dela. O valor registrado, tendo em vista que a pilha está vazia, pode ser atribuído como null.

Retornar o número de elementos: Para retornar o número de elementos, caso não possua um campo "nroElem", faz-se necessário a iteração sobre todos os elementos, fazendo um laço sobre os elementos válidos, navegando do elemento atual até o próximo dele.

Verificar se a pilha está vazia: O retorno do número de elementos pode ser utilizado para retornar se a pilha está vazia ou não, porém, para o caso onde há uma grande quantidade de elementos constados na estrutura, a iteração sobre os elementos pode demorar muito tempo. Assim, para o caso das pilhas, pode-se escrever uma estrutura de código para retornar apenas o valor do topo da pilha e, caso o valor seja null, a pilha está vazia.

Exibição dos elementos: Para a exibição dos elementos da estrutura, é necessário iterar sobre os elementos válidos e imprimir suas chaves.

Inserção dos elementos (push): O usuário passa como parâmetro um registro a ser inserido na pilha. O elemento será inserido no topo da pilha,

"acima" do elemento que anteriormente estava no topo da pilha, acertando assim os ponteiros.

Exclusão de um elemento: Para o caso de pilhas, a exclusão é feita do elemento do topo da pilha. Assim, se a pilha não estiver vazia, além de excluir o elemento, pode-se condicionar o elemento excluído para um local indicado pelo usuário.

Reinicialização da pilha: Para reinicializar a pilha, é necessário excluir todos os elementos e colocar como null o campo topo, como feito na inicialização.

2.4 Deque

A estrutura do tipo DEQUE é, por conceito, uma a qual podem ser inseridos e excluídos os elementos de suas extremidades, início ou fim. Para fins de implementação, faz-se necessário o uso do conceito de listas duplamente ligadas com um nó cabeça. Este, por sua vez, possui a funcionalidade de gerenciamento da estrutura, em que, o último elemento da estrutura tem o nó cabeça como sucessor e o nó cabeça tem o último elemento como antecessor.

Inicialização: Para a inicialização, é necessário criar o nó cabeça, e ele apontará para sí mesmo como elemento anterior e posterior do deque.

Retornar o número de elementos: Pra retornar o número de elementos, é necessário iterar pelos elementos da estrutura. Como a estrutura é ligada com nó cabeça, a iteração começa do elemento sucessor ao nó cabeça até ele mesmo.

Exibição dos elementos: Para exibir os elementos da estrutura, é necessário percorrer os elementos válidos e imprimir suas chaves. Tanto do início para o fim, quanto do fim para o ínício, a estrutura, por ser ligada e com nó cabeça, pode ser iterada e exibida, porém, deve-se atentar que o nó cabeça é apenas um elemento auxiliar, não sendo válido como elemento da estrutura.

Inserção dos elementos: O usuário escolhe a função que insere no início ou no fim e passa como parâmetro o registro a ser inserido. A função aloca a memória para o novo elemento. Assim, além do alocamento e registro do novo elemento, é necessário ajustar quatro ponteiros, dois do elemento novo e o campo anterior do elemento que será posterior do novo e o campo posterior do elemento que será o anterior do novo.

Exclusão de um elemento: O usuário passa a chave do elemento que quer excluir, se houver um elemento com essa chave, exclui-se o elemento, acerta-se os ponteiros envolvidos e retorna true. Caso contrário, retorna-se false. Para o caso da exclusão, é necessário saber quem era o predecessor, para que este, após a exclusão, passe a apontar para o elemento seguinte e, o elemento elemento do elemento excluído, passe a apontar o antecessor como o antecessor do elemento excluído.

Reinicialização do deque: Para reinicialização do deque, itera-se sobre os elementos da estrutura excluindo-os e, após o deque estar vazio, ajusta-se os ponteiros para o nó cabeça (não excluído do deque) como ele mesmo sendo o antecessor e sucessor, como na inicialização.

2.5 Fila

Fila é uma estrutura linear na qual as inserções ocorrem no final e as exclusões no ínicio. Utiliza a mesma lógica de uma fila de pessoas, em que, as pessoas que chegam na fila se dispõem na última posição e as da frente vão saindo, conforme são atendidas.

O arranjo possui um campo indicando a posição do primeiro elemento e um indicando o número de elementos. O sucessor de cada elemento está na próxima posição do arranjo (exceto o sucessor do último que estará na posição 0).

2.5.1 Fila - Implementação estática

Como as outras implementações estáticas, a estrutura possui espaço de memória pré definido, o controle da posição do elemento está no início da fila e faz-se necessário o controle do tamanho da fila.

Inicialização: Para inicializar a fila, acerta-se o valor de campo "nroElem", para indicar que não há nenhum elemento válido e o campo início, índice do primeiro valor válido.

Retornar o número de elementos: Para retornar o número de elementos, pode-se utilizar a variável "nroElem", configurada no início da fila, para retornar a quantidade de registros válidos dentro da estrutura.

Exibição dos elementos: Sabendo que há "nroElem" válidos, e o primeiro está na posição início do arranjo, percorre-se todos os elementos da estrutura, até o elemento de posição igual ao número de elementos ("nroElem") e exibe-os

Inserção dos elementos: Caso a fila não esteja cheia e com o parâmetro passado como registro pelo usuário a ser inserido no final da fila, identifica-se a posição no arranjo, na qual o elemento dever ser inserido, e insere-o. Além disso, soma-se 1 no campo "nroElem".

Exclusão de um elemento: Para a regra da fila, a exclusão é feita no início dela. Então, se a fila não estiver vazia, copia-se o elemento a ser excluído para o local indicado pelo usuário, exclui-o e acerta-se os valores dos campos "nroElem" e início.

Reinicialização da fila: Para a reinicialização da fila, chama-se a função de inicialização ou executa os mesmo comandos de inicialização da fila.

2.5.2 Fila - Implementação dinâmica

Para a implementação dinâmica, o alocamento e desalocamento de memória são feitos sob demanda, cada elemento indicará que é seu sucessor e o controle se dá pelos endereços dos elementos que estão no fim e no início da fila.

Inicialização: Para a inicialização, o usuário cria uma estrutura do tipo fila, indica em qual posição está e então indica-se os valores dos campos início e fim como sem nenhum valor válido (null).

Retornar o número de elementos: Como não há a implementação de um campo "nroElem", deve-se percorrer todos os elementos da fila para contar quantos são.

Exibição dos elementos: Do início da fila até o final, percorre-se os elementos e exibe-os.

Inserção dos elementos: Para inserção do elementos em ua fila, deve-se lembrar que a inserção é feita no final. Para implementação dinâmica, aloca-se um espaço de memória e guarda-se o registro. A estrutura, não sendo ligada e nem circular, também segue a lógica de acertar os ponteiros, ou seja, o elemento anterior ao elemento inserido agora aponta pra ele, com o campo próximo apontando pra null, e o campo fim aponta também para o novo elemento. Lembrando que também é necessário testar se a fila não está vazia, caso esteja, não há elemento anterior, então os acertos de ponteiros se dão apenas em cima do elemento inserido.

Exclusão de um elemento: Para a regra da fila, a exclusão é feita no início dela. Então, se a fila não estiver vazia, copia-se o elemento a ser excluído para o local indicado pelo usuário, acerta-se os valores dos campos início e fim e libera-se a memória. Além disso, deve-se lembrar que se o elemento excluído fosse o único da fila, ele também seria o último, então, para este caso, após a exclusão, o campo início e fim apontam para null, como no início de uma fila,

Reinicialização da fila: Para reinicializar a fila, é necessário excluir todos os elementos e colocar o valor null nos campos início e fim.

2.5.3 Duas pilhas - Implementação Estática

Como visto nos tópicos anteriores, a pilha é uma estrutura linear na qual, por regra, as inserções e exclusões ocorrem no topo. Assemelha-se, por lógica, à uma pilha de papéis.

Para o caso de uso de suas pilhas agrupadas no mesmo arranjo, a ideia central é tratar um mesmo objeto mas com dois atributos diferentes. Por exemplo, uma pilha de provas divididas em duas, as provas em que os alunos foram aprovados e as que não foram aprovados. Para este exemplo, poderia ser criado dois conjuntos de pilhas separados, porém, no caso da implementação estática, isto poderia ser um problema de desperdício de recursos.

Utilizando um único arranjo de elementos de tamanho predefinido, cada pilha será colocada em uma extremidade do arranjo, assim, conforme vão sendo acrescidos os elementos de cada pilha, mais elas vão se aproximando. Para este caso, é necessário predefinir duas estruturas, o tamanho do arranjo e o campo de posição de cada pilha.

Inicialização: Para inicialização da pilha, já criada pelo usuário, é necessário apenas acertar os valores dos dois topos das duas pilhas. Visto que o topo indicará a posição no arranjo do elemento que está no topo das pilhas e as pilhas estão vazias, pode-se, por exemplo, iniciar o topo da primeira pilha com o valor -1 e o topo da segunda pilha como valor máximo. Valor máximo é referente ao máximo valor da pilha, que foi previamente criada pelo usuário, dentro da implementação estática, ou seja, a pilha tendo um valor MAX-1, a posição do topo é uma "após", valendo MAX.

Retornar o número de elementos: Para este caso, pode-se utilizar os valores de topo das duas pilhas. Para a primeira pilha, sabendo que ela começa

do início para o fim do arranjo, o valor topo 1 mais 1 demonstra o valor total de elementos válidos. Para a segunda pilha, sabendo que ela começa do final para o início do arranjo, o valor MAX - topo 2 demonstra o valor total de elementos válidos.

Exibição dos elementos: Para exibição dos elementos, é necessário iterar sobre os elementos válidos da estrutura e imprimir suas chaves. Para o caso das duas pilhas, uma função auxiliar pode ser implementada para dizer quais das pilhas deve ser impresso os elementos.

Inserção dos elementos (push): Para a inserção, se as pilhas não estiverem cheias, o usuário passa um parâmetro de registro a ser inserido e indica em qual pilha deseja inserir. Assim, o elemento será inserido no topo da respectiva pilha e acertar os ponteiros do topo.

Exclusão de um elemento (pop): Para a exclusão, se a pilha não estiver vazia, o usuário solicita em qual pilha deverá ser feita a exclusão. Além disso, o elemento excluído será copiado para um local indicado pelo usuário.

Reinicialização da pilha: Para a reinicialização da pilha, chama-se a função de inicialização ou executa os mesmo comandos de inicialização da pilha, ou seja, atualizar os campos do topo 1, da primeira pilha, e do topo 2, da segunda pilha.

2.5.4 Matriz esparsa

Uma matriz bidimensional é um conjunto de elementos (ou tabela) composta por "m" linhas e "n" colunas.

Matriz esparsa é uma matriz na qual a grande maioria de seus elementos possui um valor padrão, como o zero, ou são nulos ou faltantes. Para os casos onde há poucos valores diferentes de zero este tipo de estrutura se aplica, evitando o desperdício de memória e processamento. Sendo assim, a estrutura definida só permitirá a alocação de elementos com valores diferentes de zero e alguma estrutura adicional de controle. Para isso, será implementado um arranjo de listas ligadas, nas quais, cada lista da matriz vai ser uma lista ligada e cada lista só irá conter elementos diferentes de zero ou poderá conter uma lista vazia, caso não haja elementos na lista determinada.

Como forma de exemplificar, considere uma matriz esparsa com três listas ligadas, em que a primeira linha contenha apenas o valor 5, a segunda linha nenhum valor, e a terceira linha com valores 3 e 4. A estrutura da matriz é do tipo ponteiro, que indica em qual posição está o arranjo de ponteiros das listas ligadas, o número de linhas e o número de colunas. No arranjo de listas, a primeira linha indica um ponteiro para o valor de memória da primeira linha, no caso, o valor 5. Consequentemente, a segunda linha aponta para o valor nulo, pois não há valores inseridos nessa linha. Por fim, a terceira, e última linha, aponta para o primeiro valor válido daquela linha, no caso, o valor 3, e este, por sua vez, aponta para o próximo da mesma linha, o valor 4. Como o valor 4 é o último da linha, seu ponteiro aponta para o valor null, indicando não haver mais nenhum valor válido naquela linha.

Inicialização da matriz: Para inicializar a matriz esparsa, faz-se necessário acertar o valor dos campos linhas e colunas, ou seja, a ordem da matriz passada pelo usuário. Além disso, é necessário criar o arranjo de listas ligadas e iniciar cada posição do arranjo com o valor null, indicando que cada lista está vazia.

Atribuição de valor da matriz: Para a atribuição de valor, o usuário deve passar o endereço da matriz, a linha, a coluna e o valor a ser colocado na respectiva posição da matriz. Assim, para critério de validação, se não houver nenhum nó na posição e o valor for diferente de zero, insere-se um novo nó na respectiva lista ligada. Se já existir um nó na posição e o valor for diferente de zero, substitui-se o valor do nó. Por fim, se já existir um nó na posição e o valor for igual a zero, exclui-se o nó da lista.

Acesso ao valor: Para acessar o valor, o usuário passa o endereço da matriz, a linha e a coluna e a função deve retornar o valor da respectiva posição. Se não houver um nó na posição, então deve retornar zero. Caso contrário, retornar o valor do nó.

2.5.5 Árvores:

A estrutura do tipo árvore é uma das possíveis soluções para um problema de busca binária. A busca binária, como visto anteriormente, facilita a busca por elementos dentro de uma estrutura, gastando menos processamento do que a busca iterada sobre todos os elementos. Porém, este método apenas é consistente em arranjos ordenados. Assim, a árvore assimila, à grosso modo, uma possível busca binária em arranjos ordenados e ligados (de implementação dinâmica).

Conceito: Uma árvore é um conjunto de nós consistindo de um nó, chamado de raíz, abaixo do qual estão as subárvores que compõem essa árvore. O número de subárvores de cada nó é chamado de grau desse nó. Os nós de grau zero, últimos nós das últimas subárvores, são chamados de nós externos ou folhas. Os demais nós são chamados de nós internos. Nós abaixo de um determinado nó são chamados de descendentes. A altura de uma árvore pode ser dada pela distância de seu último nó, ou nó de nível "n", até a raíz, nó primeiro ou de nível 0. A profundidade é dada pelo caminho inverso da altura. Assumindo a possibilidade de uma árvore não balanceada, número de nós não simétricos em relação à raiz, a altura é dada sempre pelo caminho mais longo de uma folha até a raíz. O endereço de uma árvore na memória é dado pelo endereço do seu nó raíz.

2.5.6 Árvore binária:

Uma árvore binária é uma árvore em que, abaixo de cada nó, existem, no máximo, duas subárvores. Para a construção de um algoritmo de árvores binárias, é necessário a criação de uma estrutura que contenha nós interligados, cada um desses nós deve conter uma chave e dois ponteiros, um para cada subárvore.

2.5.7 Árvores binárias de pesquisa:

Árvore binária de pesquisa é uma árvore binária em que, a cada nó, todos os registros com chaves menores que a deste nó estão na subárvore da esquerda, enquanto que os registros com chaves maiores estão na subárvore da direita.

Inicialização: Para inicialização de uma árvore, é necessário o endereço do nó raíz. Caso seja permitido a duplicação de chaves, basta definir uma estrutura, do tipo chave, menor ou igual, a um determinado nó ficam na subárvore da esquerda. Mas, por via de regra, é aplicado estruturas para impedir a duplicação de nós.

Inserção do elemento: Para inserção de um elemento em uma árvore, deve-se verificar se a raíz tem como chave o valor null, se sim, adiciona-se o elemento lá. Senão, verifica-se se a chave do elemento é menor que a raíz, se sim, insere o elemento na subárvore à esquerda, senão, à direita.

Busca de um elemento: Para busca de um elemento na árvore binária, a lógica se faz de forma parecida com a busca binária em uma lista ligada, ou seja, inicia-se a busca comparando o valor da raíz com o valor da chave passada pelo usuário, assim, caso o valor da raíz seja maior que o da chave, continua-se a busca pela subárvore da esquerda, caso seja menor, busca-se pela chave da direita e, assim, sucessivamente.

Contagem do número de elementos: Para a contagem dos elementos, há diversas maneiras de percorrer os elementos da árvore, por exemplo, raíz-subárvore da esquerda-subárvore da direita, subárvore da esquerda-raíz-subárvore da direita, entre outras. Uma ordem bastante utilizada é a da subárvore da esquerda-raíz-subárvore da direita, também chamada de inorder transversal, varredura infixa ou varredura central. Nessa varredura, os nós são visitados na ordem crescente das chaves de busca.

A contagem, em termos de código, funcionaria da seguinte forma, se não houver raíz, não há uma árvore, ou seja, a contagem é nula. Caso não seja nula, a subárvore à esquerda é percorrida e contabilizada, depois soma-se uma elemento, o elemento raíz, e percorre-se a subárvore à direita contabilizando-a também.

Leitura da árvore: Para leitura e exibição da árvore, utiliza-se, em geral, a ordem raíz-subárvore da esquerda-subárvore da direita. Assim, como na contagem do número de elementos, percorre-se os elementos, mas agora, imprimindo suas chaves.

Remoção de um elemento: Para a remoção de um elemento, ou de um nó, há algumas considerações a serem feitas. Após ser removido um nó, as subárvores desse nó devem manter a lógica estrutural de uma árvore binária de pesquisa, ou seja, os nós, à esquerda, devem ainda possuir suas chaves menores do que a do nó raíz e, os da direita, maiores.

Assim, se o nó a ser retirado possui no máximo um descendente, o descendente passa para para a posíção de memória do nó retirado. Caso, o nó possua dois descendentes, o nó, a ser retirado, é substituído pelo nó descendente mais à direita da subárvore da esquerda. Alternativamente, substitui-se o nó, a ser retirado, pelo nó mais à esquerda da subárvore da direita.

Note que, para qualquer procedimento, é viável usar qualquer nó da árvore como raíz e contar nós ou imprimir qualquer subárvore. Não é indicado incluir e excluir nós sem ter começado pela raíz pois, do contrário, pode-se perder a ordem dos nós.

2.5.8 Heap

Heap é uma estrutura de dados que pode ser visualizada como uma árvore binária quase completa. Cada nó da árvore é ocupado por um elemento e tem-se as seguintes propriedades:

- A árvore é completa até o penúltimo nível;
- No último nível as folhas estão mais à esquerda possível;
- O conteúdo de um nó é maior ou igual ao conteúdo dos nós na subárvore enraizada nele (max-heap);
- O conteúdo de um nó é menor ou igual ao conteúdo dos nós na subárvore enraizada nele (min-heap)

Tais condições garantem que a estrutura possa ser armazenada em um vetor A[1....m].

2.5.9 Árvores N-árias

Para conseguir boa performance com grandes quantidades de dados, a árvore binária de pesquisa pode não ser a melhor escolha. Isso porque, com grandes quantidades de dados, sua profundidade cresce e, dependendo da modelagem do sistema, cada nó precisará ter um número variado de sub nós, variando de 0 a n, com "n" > 2.

Para árvores que possuem "n" sub nós, ou "n filhos", tem-se duas definições: Uma árvore n-ária em que cada nó possa ter até "n" filhos, e uma árvore n-ária em que cada nó possa ter um número arbitrário de filhos.

Árvores n-árias tratam-se então de uma generalização das árvores binárias.

Árvores N-árias com "n" filhos: Considerando uma árvore n-ária desordenada, uma possibilidade para sua representação é dada por um nó chave que possui dois ponteiros, um para o nó "primogênito", e outro para o próximo nó "irmão".

Inicialização: Para a inicialização, aloca-se a memória para um nó, coloca-se um valor na chave e insere-se dois ponteiros de valor null que, como dito anteriormente, representam o "primogênito" e o "irmão".

Inserção de elementos: Para inserção de elementos, é necessário o usuário passar os valores da raíz, da chave e a chave do nó "Pai". Assim, busca-se a chave do nó "Pai", caso haja o nó "Pai", cria-se a chave ligada e verifica-se se há um "primogênito", caso não haja, será o "primogênito", caso haja, percorre-se sobre os todos os nós "irmãos" e o novo nó é apontado pelo último "irmão".

Exibição de elementos: Se a raíz for null, não há árvores a serem exibidas, encontra-se o nó da primeira subárvore e exibe seu último "filho". Assim, dentro de um laço, percorre-se por todos os elementos e os imprime. Para este tipo de estrutura, o uso de chamadas recursivas é muito grande, ou seja, há um consumo de memória de processamento muito alta.

Busca por um elemento: Para a busca de um elemento, o usuário passa um valor de chave e o ponteiro da raíz da árvore, assim, percorre-se os elementos, até encontrar a chave correta, não a encontrando, retorna-se null. Como não há ordem para os acoplamentos de subárvores em relação a raíz, como na árvore binária de pesquisa, a busca por elementos em árvores n-árias se torna uma tarefa mais complicada.

Exclusão: Como não há ordenação dos nós e subárvores, a exclusão se torna uma decisão de projeto, se um nó é excluído, qual será ligado, se sua "família" de nós é excluída junto, etc.

2.5.10 Árvore n-ária Tries

Uma trie (de retrieval), é uma árvore n-ária projetada para recuperação rápida de chaves de busca. A estrutura de uma Trie, usando o exemplo de uma estrutura de busca por palavras, como em um dicionário, o caminho de busca armazenaria uma chave inicial, como a primeira letra de uma palavra e, assim, o caminho para suas subárvores armazenariam as outras letras que formariam as palavras que comecem com essa letra. Os valores de retorno, nesse exemplo, funcionariam para validar, ou invalidar, o conjunto de palavras formado, ou seja, só retornariam um valor válido, caso a construção da palavra seja válida.

Características:

- A estrutura não armazena chave explicitamente. A chave está codificada no caminho.
- As chaves são codificadas nos caminhos a partir da raíz
- Todo nó tem um campo valor, retornado caos a chave termine nesse nó

O grande ganho da trie vem do fato de mapear diretamente um símbolo da chave a uma posição do arranjo de filhos no nó.

Inserção: Para inserção, percorre-se o caminho mapeado e insere como chave de retorno válida no nó ao qual se quer codificar. Caso não haja o caminho mapeado, mapeia-se inseriando novos "filhos" nas subárvores do caminho.

Busca: Percorre-se o caminho ao qual a chave está inserida, se o caminho e chave do último nó da busca estiver com o valor True, retorna-se True para a chave de busca. É semelhante à inserção, porém não há inclusão de nenhum nó ou caminho.

2.5.11 Árvores AVL

A árvore AVL, ou algoritmo de Adelson Velsky e Landis, é uma árvore de busca binária balanceada com relação à altura de suas subárvores. Uma árvore AVL

verifica a altura de suas subárvores, da esquerda e da direita, garantindo que essa diferença não seja maior que ± 1. A esta diferença é dado o nome de **Fator de Balanceamento**.

O fator de balanceamento é calculado a cada nó e, para cada nó, a diferença de altura entre a subárvore da esquerda e da direita não pode passa de ± 1. A árvore vazia possui uma altura igual a -1. Assim, o fator de balanceamento ou, alternativamente, a altura do nó, deve ser armazenada no próprio nó.

Inserção: Uma inserção pode fazer com que o fator de balanceamento de um nó vire ± 2. Porém, somente nós no caminho do ponto de inserção até a raíz podem ter mudado sua altura. Assim, após a inserção de um nó, volta-se até a raíz, nó por nó, e atualizam-se suas respectivas raízes. Se o fator de balanceamento para um determinado nó for ± 2, ajusta-se a árvore rotacionando em torno desse nó.

Rotação: Existem dois tipos de rotação para uma estrutura do tipo árvore AVL, a rotação para a esquerda e a rotação para a direita. Além disso, a depender de qual tipo de desbalanceamento ocorreu na árvore, há, também, os tipos de rotação simples e duplas.

Nas **rotações simples**, existe um nó "u" que é filho de "p", na direção "x". Então, "u" passa a ser a nova raíz da subárvore afetada.

Já nas **rotações duplas**, o nó "u" possui ainda um filho esquerdo, ou direito, denominado "v", também localizado em direção ao nó "x" que passa a ser a nova raíz da subárvore em questão (ou seja, substituindo "p").

Para as inserções, que geram um fator de desbalanceamento ± 2, na parte mais externa da árvore, pode-se utilizar a rotação simples para resolver o problema. Caso as inserções sejam na parte mais interna, a resolução se dá utilizando as rotações duplas. Para o caso de inserção de nó na subárvore direita, do "filho" à direita, faz-se uma rotação à esquerda para balancear a estrutura e, para o caso da inserção na subárvore à esquerda, no filho à esquerda, faz-se uma rotação à direita.

2.5.12 Árvores B

A estrutura de dados do tipo árvore B é semelhante à árvore binária, porém, diferentemente dessa, a árvore B, em cada um dos seus nós, pode ter mais do que dois filhos.

Definição: Uma árvore B de ordem "t" é vazia ou satisfaz as seguintes condições:

- A raíz é uma folha ou tem, no mínimo, dois filhos;
- Cada nó, diferente do nó raíz e das folhas possui, no mínimo, "t" filhos;
- Cada nó possui, no máximo, "2t" filhos;
- Todas as folhas estão no mesmo nível

Sabendo que, cada nó, ou página, pode ter mais do que dois filhos, assim, para acesso aos filhos, o nó "raíz" da subárvore deve possuir um conjunto de

chaves para ligação com seus filhos. Para uma árvore ser eficiente, cada página dela deve ocupar metade da ordem, exceto a raíz. Por exemplo, uma árvore de ordem dois, deve possui, no mínimo duas chaves e no máximo quatro.

Busca: A busca é feita comparando os valores das chaves dentro de cada página, começando pela raíz. De forma parecida com a busca binária, a busca em árvores B é facilitada pela ordenação da estrutura.

Inserção: Para inserção de um elemento, faz-se uma busca para encontra em qual página "cabe" o valor de chave a ser inserido. Se couber na pãgina, mantendo as propriedades discutidas anteriormente, insere-se a chave. Caso não caiba, a página está com o limite de 2t chaves, divide-se a página em duas, insere-se a chave na posição ordenada e promove-se o maior elemento da página à esquerda, caso a chave nova seja inserida à esquerda, ou o menor elemento da página à direita, caso o elemento seja introduzido na página à direita. Promover o elemento significa inseri-lo na página pai da subárvore de inserção para que essa página a mais criada não fique "solta" e a árvore mantenha sua estrutura.

Remoção: Para a remoção, há quatro caso possíveis e mais recorrentes. São eles:

- Caso 1: Se o elemento estiver em uma folha e a folha mantiver 50% de ocupação, basta removê-lo;
- Caso 2: Se o elemento não estiver em uma folha, troca-se pelo antecessor;
- Caso 3: Se a folha ficar com menos de 50% de ocupação, mas a página irmã pode ceder uma chave. Assim, promove-se o elemento da página irmã e recoloca o elemento da página pai para a página que foi removida a chave;
- Caso 4: Se a folha ficar com menos de 50% de ocupação e as páginas irmãs não puderem ceder uma chave. Para esse caso, aglutina-se duas folhas vizinhas e ajustam se os ponteiros dos elementos, mantendo a ordenação e propriedade da árvore.

2.5.13 Árvores rubro-negra

Uma árvore rubro-negra é um tipo de árvore binária balanceada em que, cada nó, é constituído de uma "cor" (um bit extra), sendo vermelha ou preta, uma chave, ponteiros para subárvores da esquerda e da direita e um ponteiro que aponta para o seu pai, ou para null, caso seja o nó raíz. Considera-se que quando um nó não possui um nó filho, então supõe-se que ele aponta para um nó fictício, null, que será a folha da árvore.

A propriedades de uma árvore rubro-negra são:

- Todo nó da árvore é vermelho ou preto;
- A raíz é preta;
- Toda folha (null) é preta;

- Se um nó é vermelho, então ambos os filhos são pretos, ou seja, não existe nós vermelhos consecutivos;
- Para todo nó, todos os caminhos até as folhas descendentes contêm o mesmo número de nós pretos.

Árvore AVL x Árvore Rubro-Negra Na teoria, ambas possuem a mesma complexidade computacional (inserção, remoção e busca), em notação Big O, $O(\log N)$. Na prática, a estrutura AVL é mais rápida em operações de busca, devido ao seu maior balanceamento, e mais lenta em operações de inserção e remoção. Nos piores casos, a operação de remoção e inserção pode custar "$O(\log N)$" rotações para árvore AVL e apenas três rotações para a árvore rubro-negra.

A árvore rubro-negra é inserida dentro de uma classe de estrutura de dados chamada árvores balanceadas de busca. Dentro deste conceito, mantendo a propriedade da classe, as árvores rubro negras tendem a manter a sua altura, mesmo após inserções e exclusões de nós. Isso garante o tempo de processamento $O(\log n)$ para operações de busca, inserção e remoção,

Busca: A busca funciona com a mesma lógica das outras árvores balanceadas, como a AVL.

Inserção: Um nó é inserido sempre na cor vermelha, assim, não altera a altura negra da árvore, garantindo a última propriedade da lista vista anteriormente.

A inserção começa com a busca da posição onde o novo nó deve ser inserido, partindo-se da raíz para os nós que possuam o valor mais próximo do elemento a ser inserido na estrutura.

Caso a inserção seja feita em uma árvore vazia, basta alterar a cor do nó inserindo, mantendo a propriedade da raíz preta.

Ao se inserir um nó, cuja posição equivale a estar ligado a uma família de nós, cujo o "tio" seja vermelho, é necessário trocar as cores do pai e do tio para preto e do avô para vermelho.

Caso o tio do nó inserido seja preto, faz-se necessário utilizar do procedimento de rotação, como visto na seção de árvores AVL.

Há quatro subcasos os quais faz-se necessário o uso da rotação para manter as propriedades da árvore rubro-negra. São eles:

Inserção à esquerda de um nó vermelho à esquerda:

Inserido o elemento, faz-se uma rotação à direita, descendo o avô para a direita com a cor vermelha, promovendo o pai como "raíz" da subárvore, com a cor preta, deixando a estrutura com pai preto e filhos vermelhos, a esquerda o nó inserido e, a direita, o antigo avô.

Inserção à direita de um nó vermelho à direita:

Para esse caso, faz-se um operação similar ao caso anterior, porém, a rotação é à esquerda, nesse caso. Assim, mantem-se uma estrutura com pai preto, antigo avô à esquerda e o elemento inserido à direita, ambos os filhos vermelhos.

Inserção de um nó vermelho à esquerda de um nó vermelho à direita:

Esse caso pode ser observado como uma combinação dos outros dois anteriormente citados. Primeiro, faz-se uma rotação simples à direita e, depois, uma rotação simples à esquerda, fazendo os ajustes de cor, mantendo o elemento inserido como preto, e novo pai, e o antigo avô e pai, como filhos vermelhos, à esquerda e a direita, respectivamente.

Inserção de um nó vermelho à direita de um nó vermelho à esquerda:

Da mesma forma, esse é mais um caso com combinação de rotações. Assim, para este caso, deve-se fazer uma rotação simples à esquerda e, após isso, uma rotação simples à direita, também fazendo os ajustes de cores.

Remoção: Para a remoção, caso o nó seja vermelho, não há alteração no balanceamento da altura negra da estrutura, apenas sendo necessária a substituição do apontamento do nó pai do elemento removido, para o seu sucessor vermelho. Porém, quando há a remoção de um nó preto, há algumas situações a serem consideradas, para que a remoção não viole as propriedades da árvore rubro-negra.

Antes de abordar os casos de remoção de nós pretos, considere os itens abaixo:

- Seja "z" o nó a ser removido;
- "y" o sucessor de "z";
- Seja "x" o filho de "y" antes da remoção de "z";
- Seja "w" o tio de "x" antes da remoção de "z".

Considere as seguintes condições a seguir:

	Nó a ser removido	Sucessor
Caso 1	Vermelho	Vermelho
Caso 2	Preto	Vermelho
Caso 3	Preto	Preto
Caso 4	Preto	Vermelho

Tabela 8: Casos para a remoção de um nó em uma árvore rubro-negra

Caso 1 Para o Caso 1, quando se tem sucessores como nós vazios (null) pretos, a remoção de um nó vermelho não altera as propriedades da árvores rubro-negra, assim, o nó pode ser removido diretamente.

Caso o filho seja não nulo e o sucessor seja vermelho, altera-se o valor da chave entre os nós vermelhos, sem alterar a cor dos mesmos, e remove-se o elemento, agora disposto como "folha" da árvore.

Caso 2 Para esse caso, troca-se de posição com o elemento a ser removido com o elemento sucessor, e, também, troca-se a cor do elemento sucessor.

Caso 3 Para o caso 3, há quatro subcasos a serem abordados.

- Caso 3.1 - O irmão de "w" é vermelho;

- Caso 3.2 - O irmão "w" de "x" é preto, e ambos os filhos de "w" são pretos;
- Caso 3.3 - O irmão "w" de "x" é preto, o filho esquerdo de "w" é vermelho e o filho da direita é preto;
- Caso 3.4 - O irmão "w" de "x" é preto, o filho direito de "w" é vermelho.

Caso 3.1 Como "w" é vermelho, ambos os filhos são pretos, assim, troca-se as cores de "w" e o seu pai e efetua-se a rotação à esquerda, com o pai de "w" como pivô. Essas alterações não produzem erros de propriedades à arvore, porém, transforma o caso 3.1 no caso 3.2, 3.3 ou 3.4.

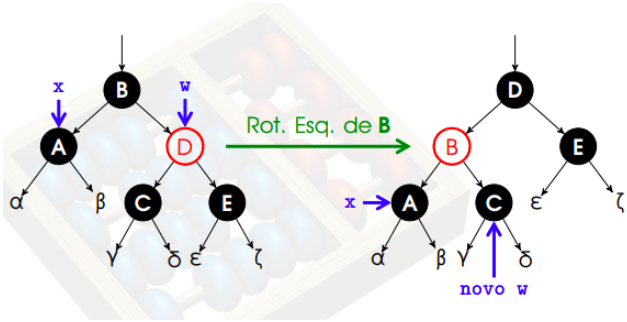

Figura 1: Remoção para o caso 3.1
Fonte: https://www.ic.unicamp.br/ francesquini/mc202/files/aula16-18.pdf

Caso 3.2 Para esse caso, retira-se um nó preto de "x" e "w" e redefine o "w" como vermelho. Para compensar a redução de um nó preto de "x" e "w", adiciona-se um nó preto extra como pai de "x" e "cresce-se" a árvore para cima, tratando o pai de "x" como o novo "x".

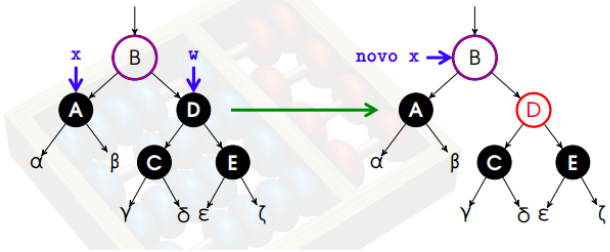

Figura 2: Remoção para o caso 3.2
Fonte: https://www.ic.unicamp.br/ francesquini/mc202/files/aula16-18.pdf

Caso 3.3 Para esse caso, troca-se as cores de "w" e de seu filho esquerdo, rotaciona-se a árvore à direita, usando como pivô "w". Assim, recai-se no caso 3.4.

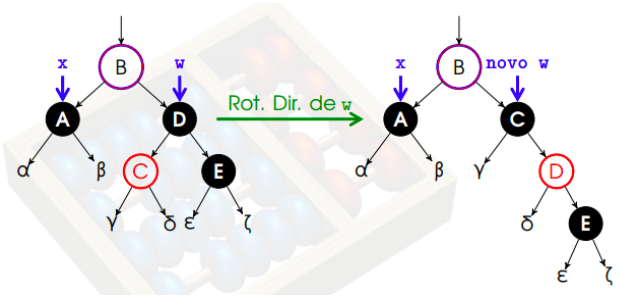

Figura 3: Remoção para o caso 3.3
Fonte: https://www.ic.unicamp.br/ francesquini/mc202/files/aula16-18.pdf

Caso 3.4 Para esse caso, rotaciona-se a árvore à esquerda usando como pivô o pai de "x", redefine "w" com a cor do pai de "x", redefine o pai de "x" como preto e, por último, redefine o filho direito de "w" como preto.

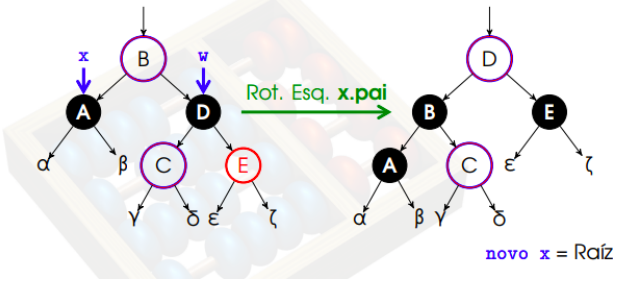

Figura 4: Remoção para o caso 3.4
Fonte: https://www.ic.unicamp.br/ francesquini/mc202/files/aula16-18.pdf

Caso 4 Para o último caso, redefine-se o "x" como vermelho e resolve o restante da solução como um caso 3.

2.6 Grafos

2.6.1 Grafos - Conceitos

Grafos são estruturas matemáticas que permitem codificar relacionamentos entre pares de objetos. Os objetos são os vértices (ou nós) do grafo. Os relacionamentos entre objetos são as arestas.

Alguns grafos são **dirigidos** (ou direcionados), estes, por sua vez, possuem arestas com sentidos definidos. Em grafos dirigidos, as arestas são pares ordenados de vértices, saindo um em direção ao outro, mesmo que ambos sejam o mesmo vértice (self-loop).

Outros são **não dirigidos** (ou não direcionados). As relações representadas

pelas arestas não têm sentido definido, ou seja, podem ser seguidas em qualquer direção. Entende-se também como um grafo dirigido com arestas de sentido duplo e estas, como pares não ordenados de vértices. Para estes casos, o self-loop não é permitido.

A adjacência, ou vizinhança, de dois vértices pode ser representada por (u, v), onde "u" é um vértice que tem aresta no sentido de "v". Para grafos não direcionados, a ordem dos fatores não importa, isso porque as relações não tem sentido definido.

Em grafos não dirigidos, o grau de um vértice é o número de arestas que incidem nele. Em grafos dirigidos, o grau de um vértice é o número de arestas que saem do vértice, chamado de grau de saída, mais o número de arestas que chegam nele, chamado de grau de entrada.

Um caminho de um vértice "x" a um vértice "y" é uma sequência de vértices em que, para cada vértice, do primeiro ao penúltimo, há uma aresta ligando esse vértice ao próximo da sequência. O comprimento de um caminho é dito pela soma de arestas que estão nele. Já um ciclo acontece quando, a partir de um determinado vértice, pudermos percorrer algum caminho que nos leve a esse mesmo vértice. Em grafos dirigidos, o caminho deve conter pelo menos uma aresta. Em grafos não dirigidos, um ciclo deve conter pelo menos três arestas, que formem, no mínimo, um triângulo, visto que não existe, para estes casos, o self-loop. Um grafo que possui, ao menos, um ciclo, é chamado de cíclico, assim também, o que não contém, é chamado de acíclico.

Um grafo não direcionado é conexo (ou conectado) se cada par de vértices nele estiver conectado por um caminho.

Um grafo dirigido é fortemente conexo se existir um caminho entre qualquer par de vértices no grafo. É conexo se possuir no mínimo um caminho de ligação entre os vértices, podendo ter uma "ida" e não uma "volta". Já os grafos direcionados fracamente conexos são aqueles que, se substituir todas suas arestas por arestas não-direcionadas produz nele um grafo conexo.

Grafos também podem ser ponderados, isso quer dizer que suas arestas possuem um "peso". Tal valor pode ser atribuído a um custo, uma distância, etc.

Uma árvore pode ser dita como um grafo acíclico, conexo e não dirigido.

2.6.2 Grafos - Representação

A representação de um grafo pode-se dar pelo mapeamento de cada nó à lista de nós aos quais ele está conectado. Há duas maneiras usuais de representação dos grafos, como uma matriz de adjacências ou como uma lista de adjacências.

Matriz de Adjacências: Uma matriz de adjacências "A" de um grafo com "n" vértices é uma matriz "n" x "n" de bits, em que:

- A[i, j] = 1 se houver uma aresta indo do vértice i para o vértice j no grafo.
- A[i, j] = 0 se não houver aresta indo de i para j.

No caso de um grafo ponderado pelas arestas, estas, representadas em uma matriz, terão o valor do peso das ligações dos vértices como inteiro, float, etc.

Exemplificando, no caso de um grafo não direcionado, uma aresta com peso 6, entre dois vértices,"u" e "v", são representadas, dentro da matriz, tanto no campo "u" para "v", quanto no campo "v" para "u", com o valor 6. Para o caso do peso zero, vale lembrar que, dentro do algoritmo, isto pode se tornar ambíguo, pois o valor zero pode representar a ausência de aresta entre os vértices também.

A matriz de adjacências deve ser utilizada para grafos densos, em que a quantidade de arestas é próxima ao quadrado dos vértices. O tempo necessário para acessar um elemento é independente das arestas e dos vértices. Com isso, a matriz de adjacência se torna muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices. Entretando, a estrutura possui a desvantagem de ocupar muito espaço de memória.

Listas de Adjacências: Uma lista de adjacências de um grafo com "n" vértices consiste em um arranjo de "n" listas ligadas, uma para cada vértice no grafo. Ou seja, para cada vértice "u", a lista contém todos os vizinhos de "u" e todos os vértices "v", para os quais existe uma aresta (u, v). Para os casos onde há ponderação, como nos casos das matrizes de adjacência, a ponderação é colocada na lista ligada com os vértices.

Um grafo então possui, dentro de um algoritmo, o número de vértices, o número de arestas e o arranjo de vértices, em que cada vértice contém uma lista de suas adjacências. Cada adjacência possui, o vértice de destino, o peso associado à aresta que leva ao vértice de destino e o próximo elemento na lista de adjacências.

A decisão para o uso da representação, como lista ou matriz, depende da densidade do grafo. Se o grafo for muito denso, quando possui muitas arestas em relação ao número de vértices, a matriz pode ser uma melhor alternativa, visto que na lista, os ponteiros sobracarregariam mais ainda a estrutura. Caso o grafo seja esparso, com poucas arestas, a lista pode ser uma melhor alternativa, visto que a busca é mais eficiente, pois, dado um nó, o ponteiro já indica o endereço da sua lista de adjacências. Porém, para testar a existência de uma aresta, a matriz é uma melhor opção, visto que, o valor da aresta já se encontra na matriz.

Em resumo, a escolha de uma melhor estrutura depende do tipo de aplicação e projeto. Assim, para se tomar a decisão, vale ressaltar que a matriz de adjacência tem uma melhor performance para grafos densos, operações como testes de aresta, identificação de predecessores, etc. Já a lista de adjacências possui melhor performance que a matriz quando a estrutura possui grafo esparsos, operações que tenham como base um caminho de um vértice a outro, como a busca, etc.

2.7 Tabela de Hash

Muitos métodos de busca funcionam com a base na comparação de chaves, ou seja, algum valor que o compõe. Porém, muitos dos algoritmos só conseguem ser realmente eficientes se a estrutura estiver ordenada. Para esses casos, a "busca ideal" seria a de acesso direto, ou instantâneo, sem nenhuma etapa

de comparação de chaves. Em notação O, seria o tempo de execução O(1). A estrutura do tipo array faz o retorno de uma chave de forma instantânea, devido ao regime de indexação, porém, ela não calcula a posição de uma chave, operação de busca, sem o efeito comparativo de chaves. Assim nasce a tabela hash.

Assim, a tabela hash, também conhecida como tabela de indexação ou espalhamento, nasce da generalização da ideia de um array. A tabela é criada a partir de uma função hash, que, de forma não ordenada, espalha os elementos pelo array. O espalhamento dos elementos dentro de um array é mais eficiente, visto dentro da perspectiva de hash, pois ele associa uma chave, parte da informação que compõe o elemento a ser inderido, a um índice, que é a posição do elemento dentro do array. Assim, a partir da chave é possível acessar, de forma rápida, uma determinada posição no array.

Vantagens:

- Alta eficiência na operação de busca;
- Tempo de busca é praticamente independente do número de chaves armazenadas na tabela;
- Implementação simples.

Desvantagens:

- Alto custo para recuperar os elementos da tabela ordenados pela chave. Nesse caso, convém-se ordenar a tabela;
- Para o pior caso, a tabela de hash possui desempenho de O(n), sendo "n" o tamanho da tabela. Para esse caso, a tabela sofre com muitas colisões, ou seja, situações as quais dois elementos "tentam" ocupar a mesma posição.

A tabela hash tem muita aplicabilidade na prática. Algumas de suas aplicações são:

- Busca de elementos em base de dados;
- Criptografia: MD5 e família SHA (Secure Hash Algoritm);
- Implementação da tabela de símbolos dos compiladores;
- Para fins de segurança, senhas dentro de servidores são armazenadas em funções hash;
- Verificação de integridade de dados e autenticação de mensagens.

O mapeamento entre a chave e o índice é conhecido como função de espalhamento. Tal função irá receber um item qualquer da coleção e irá retornar um inteiro dentro do intervalo dos índices, isto é, entre 0 a n-1. Um dos métodos utilizados para calcular o índice é o método do resto, o qual tem, por princípio, pegar um item e dividí-lo pelo tamanho da tabela e, o resto desta divisão, é o

valor de índice do item. Assim, quando for necessária a busca de um elemento dentro da estrutura, calcula-se o índice correspondente da chave e acessa-se a tabela de hash para verificar se a chave está lá. Visto que é necessário apenas um tempo constante para esse tipo de operação matemática, em notação "Big O", o tempo de busca em uma tabela hash é de O(1) em geral. Porém, há valores de chaves possíveis que, ao calcular, pelo exemplo da função de hash visto anteriormente, que podem resultar em um mesmo índice, ou seja, uma mesma posição dentro da estrutura. Por exemplo, em uma tabela de tamanho 11, dados dois valores de chave, 22 e 44, ambos possuem, como resto da divisão por 11, o valor zero. Assim, em teoria, ambos tentariam ocupar o mesmo espaço de memória. Sobre **colisões**, haverá mais uma sessão para debate das possíveis resoluções para esse problema.

2.7.1 Funções de Espalhamento

A **função de espalhamento perfeita** é a qual proporciona um valor de índice para cada chave valor dentro de uma estrutura de dados. Porém, isso seria possível caso a coleção de itens fosse estática e que nunca irão mudar. Felizmente, também não é necessário, para um bom desempenho, que a função de espalhamento seja perfeita.

Uma das maneiras de se obter a função de espalhamento perfeita é aumentando o tamanho da tabela de dispersão, de modo que cada valor possível no intervalo de itens possa ser acomodado com um único índice. Entretanto, tal método tem um custo de memória muito alto. O ideal é obter um método que concilie o baixo número de colisões, que distribua os elementos uniformemente na tabela de dispersão e que seja de fácil processamento. Abaixo há algumas definições para alguns métodos de espalhamento.

2.7.2 Método de Folding

O método de folding consiste em dividir o valor do item em pedaços de tamanhos iguais, caso necessário, deixando apenas o último pedaço com tamanho diferente. Assim, os pedaços são somados e seu resultado é dividido por 11, sendo o resto desta divisão o índice do item. Por exemplo, considere um item como um número telefônico, 436-555-4601, a divisão dos pedaços, a somatória dos termos e o resto da divisão por 11 seria algo como:

$$43 + 65 + 55 + 46 + 01 = 210 \tag{10}$$

$$210\%11 = 1 \tag{11}$$

Assim, o índice do item é 1. Em alguns casos do método, é feita a troca de algarismos de um dos pedaços, como, 65 vira 56.

2.7.3 Método do quadrado do meio

Outra técnica numérica para construir a função de espalhamento é o método do quadrado do meio. A técnica consiste em pegar o valor de um item, elevá-lo à segunda potência, retirar do resultado alguns dígitos, por exemplo, os dois algarismos do começo, e transformar o resto da divisão por 11 no índice do item.

Alguns outros métodos podem utilizar strings como parâmetros da função, onde vincula-se um número a cada letra e, com esse conjunto de números, a função de espalhamento é construída.

É importante lembrar que a função de hash precisa ser eficiente, de modo que ela não se torne parte dominante do processo de armazenamento e busca e há outras maneiras, fora as quais já foram apresentadas, para indexar os elementos da tabela. Assim, se a função se tornar complexa demais, em termos de processamento e espaço, outras estruturas, como uma lista ou árvore binária, serão mais eficientes do que a tabela hash.

2.7.4 Resolução de Colisões

A colisão acontece quando, dentro de uma estrutura, um item adicionado a ela tenta ocupar o espaço determinado para um item anteriormente posto. Em funções de hash perfeitas, a colisão nunca ocorre, porém, isso não condiz com a realidade em vários casos.

Um dos métodos de resolução de conflitos é conhecido como **endereçamento aberto** que consiste em, ao identificar uma colisão, é feita uma sondagem linear, método que verifica de forma sequencial uma posição por vez, pela estrutura, começando pela função original de espalhamento. Da mesma forma, ao se fazer a busca por um elemento, deve-se, primeiramente, calcular a sua posição em hash e, caso o valor de chave esteja ocupado por outro valor, faz-se necessário buscar sequencialmente a estrutura inteira. Assim, a tabela de hash, para alguns casos, não possui processamento de O(1).

Além do tempo de processamento de uma busca sequencial, uma outra desvantagem da tabela de hash que possui muitas colisões é a **aglutinação**, tendência de que os itens fiquem agrupados na tabela. Isso pode ser problemático para a estrutura inteira e, a fim de resolver esse problema, pode-se estabelecer algumas distâncias de inserção entre os elementos. Essa técnica de procurar por slots vazios dentro de uma estrutura de hash é chamada de **rehashing**. É importante entender também que a "distância", número de slots vazios entre os elementos da estrutura,, deve assumir um valor tal que todas as entradas da tabela sejam visitadas em algum momento, ou parte da tabela pode ser inutilizada. Assim, por garantia, recomenda-se que o tamanho da tabela seja um número primo.

Uma variação da sondagem linear é a **sondagem quadrática**, que consiste em, identificada a colisão, o rehash é feito na ordem 1, 3, 5, 7, por exemplo. Ou seja, na sondagem linear, os "pulos" dos slots eram fixos, na sondagem quadrática é dado em termos uma progressão.

O **Encadeamento** também é uma técnica para evitar colisões. O método consiste em, ao ser identificada uma colisão, o item seja adicionado à uma lista encadeada que tem como ponteiro inicial a posição de colisão. Assim, o item, ao colidir, é colocado na última posição encadeada apontada pelo sua posição calculada em hash.

Uma estrutura baseada em funções de hash, dentro da linguagem de Python, é o **dicionário**. O dicionário é um tipo de dado associativo onde é possível armazenar pares de chave-valor.

2.8 Extra

Esta seção trata de conceitos extras, sem uma "ordem lógica ou temporal", para ampliar ainda mais o entendimento de algoritmos e estruturas de dados.

2.8.1 Filas de Prioridade

Alguns algoritmos têm a necessidade de processar um conjunto em ordem, porém, não todos de uma vez só. Assim, diferentemente de uma fila comum, a fila de prioridade é uma estrutura de dados que atribui uma prioridade (ou chave) a cada elemento do seu conjunto. Assim, dentro de um conjunto "Q", durante o processamento do algoritmo, a estrutura da fila de prioridade garante que elementos que possuem maior prioridade sejam retirados da fila e reorganizados de forma decrescente sobre o nível de prioridade determinados, chaves de maior ou menor valor, por exemplo.

Um fila de prioridade de máximo deve admitir as seguintes operações:

- Inserção (Q, x): Insere o elemento "x" no conjunto "Q";
- Máximo (Q): Retorna o elemento com o maior valor da chave (maior prioridade da fila);
- ExtraiMáximo (Q): Remove e retorna o elemento com o maior valor da chave (maior prioridade da fila);
- IncrementaChave (Q, x, k): Altera a prioridade do elemento "x" para o valor de "k" a fim de aumentar sua prioridade na fila;

Alternativamente, uma fila de prioridade de mínimo suporta as operações de inserção, mínimo, ExtraiMínimo e DecrementaChave.

2.8.2 Algoritmo de Bellman-Ford

O Algoritmo de Bellman-Ford serve para solucionar um problema que o Algoritmo de Dijkstra não conseguia resolver, a solução de um caminho mais curto dentro de uma estrutura ponderada com pesos negativos. Porém, apesar de solucionar o problema do peso negativo, o algoritmo de Bellman-Ford possui um tempo de execução maior do que o Algoritmo de Dijkstra.

Como dito anteriormente, o algoritmo de Dijkstra faz a iteração partindo do nó inicial e determinando, ou "fixando", a não possibilidade de um caminho mais curto do que o do inicial, ou nó "pai", para seus respectivos "filhos" diretos, e assim por diante. Porém, nos casos em que há algum peso negativo nas arestas, o algoritmo é ineficaz. Já o algoritmo de Bellman-Ford, analisa, dentro de sua iteração, para além das possíbilidades de um vértice e seus caminhos diretos, as possibilidades dos níveis de familiariade simultaneamente. Em resumo, o algoritmo processa as informações sem prioridades de subcaminhos mais curtos entre os vértices e sim, com subcaminhos mais curtos entre os níveis do grafo. Com isso, processando essa maior quantidade de iterações, buscando sempre o caminho mais curto sem um fila de prioridades, adiciona, consideralvemente, volume de recerssões ao algoritmo, deixando-o mais lento, porém, resolvendo o problema de pesos negativos.