MAC0121/0122 - 2025
Exercício de programação 1

31 de agosto de 2025

1 Introdução

A sua tarefa nesse EP será a de utilizar o conhecimento adquirido em sala de aula para implementar as estruturas de dados **lista ligada**, **fila**, **pilha** e **grafos**, de forma a serem manipuladas a partir de um menu interativo, que já estará implementado no `main.c`.

2 Requisitos

O seu programa deve ser implementado em `C` e você não deve utilizar (`# include`) bibliotecas que não tenham sido implementadas por você. Nos arquivos do tipo header (`*.h`) você deve apenas modificar os espaços marcados com `/* PREENCHA */`. Note que também foram fornecidos, além deste enunciado, diversos arquivos `.h` com as declarações das assinaturas das funções que deverão ser implementadas. Você não deve trocar o nome de nenhuma função obrigatória, pois tais nomes serão utilizados pelos arquivos de teste. Funções obrigatórias com nomes diferentes não serão consideradas.

2.1 Estruturas a serem implementadas

As estruturas que devem ser implementadas estão descritas abaixo. Você pode escrever funções adicionais para ajudar na implementação das que foram solicitadas e tem liberdade para implementar da maneira que achar melhor (naturalmente, questões relacionadas à eficiência do programa serão avaliadas e influenciarão na nota). O mais importante é que as assinaturas das funções solicitas sejam **exatamente** como estão descritas.

Para ajudar na implementação, vamos considerar que os elementos inseridos nas estruturas são **inteiros não-negativos**, i.e., não vamos testar as estruturas com números negativos (isso permite que algumas funções retornem `-1` para indicar que uma operação foi mal-sucedida, por exemplo).

1. **Lista ligada:** a lista ligada deve ser implementada da forma como você preferir, mas deve possibilitar o seu uso nas outras estruturas de dados que serão implementadas.

2. **Fila:** a sua fila **deve** ser implementada utilizando a lista ligada implementada no item 1. Lembre-se que, em uma fila, o elemento que está no início (frente) da fila é o elemento que está na fila há mais tempo. Sua fila deve suportar as operações a seguir.

   - `Queue* create_queue()`: cria uma fila vazia;
   - `void enqueue(Queue* Q, int novo_valor)`: insere o elemento `novo_valor` na fila `Q`;
   - `int dequeue(Queue* Q)`: remove e retorna o elemento que está no início da fila `Q`, se a fila não estiver vazia; caso contrário retorna `-1`.
   - `int front(Queue* Q)`: retorna o elemento que está no início da fila `Q`, se a fila não estiver vazia; caso contrário, retorna `-1`.
   - `int is_empty(Queue* Q)`: informa se a fila `Q` está vazia ou não; se estiver, retorna `1`; caso contrário, retorna `0`.

3. **Pilha:** a sua pilha deve ser implementada utilizando a lista ligada implementada no item 1. Lembre-se que, em uma pilha, o elemento que está no topo da pilha é o elemento que está na pilha há menos tempo. Sua pilha deve suportar as operações a seguir.

   - `Stack* create_stack()`: cria uma pilha vazia;
   - `void push(Stack* S, int novo_valor)`: insere o elemento `novo_valor` na pilha;
   - `int pop(Stack* S)`: remove e retorna o elemento que está no topo da pilha `S`, se a pilha não estiver vazia; caso contrário, retorna `-1`;
   - `int top(Stack* S)`: informa qual é o elemento que está no topo da pilha `S`, se a pilha não estiver vazia; caso contrário, retorna `-1`;
   - `int is_empty(Stack* S)`: informa se a pilha `S` está vazia ou não; se estiver, retorna `1`; caso contrário, retorna `0`.

4. **Grafos:** um grafo é uma estrutura matemática utilizada para abstrair redes. Formalmente, um grafo G é um par (V, E) em que V é um conjunto finito de objetos, e E é um conjunto de pares não-ordenados de elementos de V. Os elementos de V são chamados de *vértices*, e os elementos de E são chamados de *arestas* (do inglês *edges*). Se $\{u, v\} \in E$, dizemos que u e v são *vizinhos* ou *adjacentes*. Para simplificar, considere que em um grafo com n vértices temos $V = \{0, \ldots, n-1\}$.

   Há duas maneiras usuais de se implementar o armazenamento das arestas de um grafo: *listas de adjacências* e *matriz de adjacência*. Nas listas de adjacência, **a cada vértice** associamos uma lista ligada contendo precisamente os seus vizinhos.

   Na matriz de adjacência, associamos **ao grafo** uma matriz $n \times n$ `AM` tal que `AM[u][v] = AM[v][u]` $\in \{0, 1\}$ e `AM[u][v]` vale `1` se e somente se u e v são adjacentes. Tais modelos não devem ser utilizados ao mesmo tempo. Dessa forma, a estrutura grafo implementada neste EP deve possuir um atributo inteiro chamado `type` que assume os valores `0` ou `1`. Dizemos que este atributo é o *tipo* do grafo. Quando o tipo de um grafo G é `0`, o armazenamento das arestas deve ser feito utilizando listas de adjacência, enquanto quando o tipo de G é `1`, o armazenamento deve ser feito utilizando a matriz de adjacência.

   O seu grafo deve suportar as operações a seguir (note que a implementação das operações **deve** levar em conta o tipo do grafo).

   - `Graph* create_graph(int n, int tipo)`: cria um grafo do tipo `tipo` com `n` vértices;
   - `int add_edge(Graph *G, int u, int v)`: insere uma aresta entre os vértices `u` e `v`, caso não exista aresta entre `u` e `v`. Se a inserção for bem sucedida, retorna `1`; caso contrário, retorna `0`;
   - `int remove_edge(Graph *G, int u, int v)`: remove a aresta entre os vértices `u` e `v`, caso exista. Se a remoção for bem sucedida, retorna `1`; caso contrário, retorna `0`.

   Você deverá implementar também as seguintes operações tradicionais de busca em grafos.

   (a) Dado um vértice inicial u, a *Busca em Largura* (do inglês *BFS - Breadth-First Search*) explora o grafo em "camadas", i.e., de acordo com a distância de u. Primeiramente ela visita todos os vizinhos imediatos de u. Depois, visita todos os vizinhos desses vizinhos, e assim por diante, sem visitar duas vezes um mesmo vértice, até que não seja possível visitar mais nenhum vértice (veja Cap 22.2 do Introduction to Algorithms de Cormen–Leiserson–Rivest–Stein). Então a *distância* de u para um vértice v é a camada em que v se encontra. Por exemplo, se $v = u$, então a distância de u para v é 0, e se v é um vizinho de u, então a distância de u para v é 1. Se v não foi visitado, então a distância de u para v é -1.

   Dica: utilize uma fila para armazenar os vértices que foram descobertos;

   (b) Dado um vértice inicial u, a *Busca em Profundidade* (do inglês *DFS - Depth-First Search*) explora o grafo em "profundidade" a partir de u. Na busca em profundidade as arestas são exploradas a partir do vértice visitado mais recentemente que ainda tem vizinhos não visitados. Ao verificar que todos os vizinhos de um vértice v já foram visitados, voltamos ao único vizinho w de v que foi visitado antes de v. Neste caso, dizemos que w é o *predecessor* de

v a partir (ou com relação a) u. Fazemos isso até que não seja possível visitar mais nenhum vértice (veja Cap 22.3 do Introduction to Algorithms de Cormen–Leiserson–Rivest–Stein). Se w não foi visitado a partir de u, então o predecessor de w a partir de u é -1.

Dica: utilize uma pilha para armazenar os vértices que foram descobertos.

O seu grafo deve então suportar as duas operações a seguinte correspondente às buscas apresentadas acima.

- `int* bfs(Graph *G, int u)`: Realiza uma BFS no grafo `G` a partir do vértice `u`, e retorna o vetor `d` de distâncias para `u`, i.e., tal que `d[v]` é a distância de `u` para `v`. **Você deve utilizar a fila implementada no item 2**;
- `int* dfs(Graph *G, int u)`: Realiza uma DFS no grafo `G` a partir do vértice `u`, e retorna o vetor `p` de predecessores, i.e., tal que `p[v]` é o predecessor de `v`. **Você deve utilizar a pilha implementada no item 3 (a função `dfs` não deve ser implementada de maneira recursiva neste EP)**.

Finalmente, dizemos que um grafo é *conexo*, se é possível chegar de qualquer vértice em qualquer outro vértice. Utilize os algoritmos acima para decidir se um grafo é conexo, de modo que a seguinte operação seja suportada.

- `int is_connected(Graph *G)`: Retorna `1` se `G` é conexo, e `0` caso contrário.

2.2 Relatório

Após a sua implementação, deverá ser escrito um relatório contendo explicações sobre os raciocínios e decisões que foram tomadas durante o desenvolvimento do seu EP.

Além disso, deverão ser executados testes de desempenho comparando a implementação de grafos com listas de adjacência e matriz de adjacência. Os experimentos deverão considerar, para ambos os tipos, três grafos distintos:

- Com uma quantidade **pequena** de vértices;
- Com uma quantidade **média** de vértices;
- Com uma quantidade **grande** de vértices.

→ Todos os grafos deverão conter uma quantidade razoável de arestas. Especifique no relatório os valores escolhidos para as quantidades de vértices e arestas e o porquê de tais escolhas.

Considerando os seis grafos distintos que foram construídos, deverão ser medidos três tempos distintos:

1. O tempo de **construção** de cada grafo;
2. O tempo que cada grafo consumiu para executar uma **BFS**;
3. O tempo que cada grafo consumiu para executar uma **DFS**;

→ Note que os tempos de execução nunca são constantes. Portanto, para cada medição, realize 15 experimentos e use como valor final a média dos resultados obtidos.

→ Além disso, para obter resultados mais concretos na medição do tempo, realize os experimentos com a menor quantidade possível de impressões em tela, uma vez que elas podem comprometer a performance e, por conseguinte, os resultados da análise.

Por fim, deverá ser realizada uma **análise comparativa** das medições, explicando se os resultados obtidos estiveram de acordo com o esperado, levando em consideração a complexidade teórica de cada operação e de cada implementação do grafo – com listas de adjacência e matriz de adjacência.

3 Entrega

Você deverá entregar, por meio do e-Disciplinas, um **único** arquivo `.tar.gz`. A descompactação do arquivo `.tar.gz` deve gerar uma pasta cujo nome deve ser o seu número USP. Arquivos `.tar.gz` vazios ou corrompidos acarretarão um desconto de 1 ponto na nota. Similarmente, se o nome do diretório criado após a descompactação do arquivo `.tar.gz` não for o seu NUSP, haverá desconto de 1 ponto na nota. O diretório deverá conter os seguintes arquivos.

- um arquivo `list.c` e um arquivo `list.h`, referentes à implementação da lista ligada;
- um arquivo `queue.c` e um arquivo `queue.h` referentes à implementação da fila;
- um arquivo `stack.c` e um arquivo `stack.h` referentes à implementação da pilha;
- um arquivo `graph.c` e um arquivo `graph.h` referentes à implementacão do grafo (tipos `0` e `1`); e
- um arquivo `relatorio.pdf` descrevendo as funções implementadas, explicando suas decisões de implementação e relatando os resultados dos testes feitos.

Caso haja mais arquivos do que os solicitados, haverá desconto na nota.

3.1 Observações

1. Certifique-se de alocar e liberar os espaços de memória de maneira consistente e segura. Se você tiver dúvidas com relação a isto, escreva no fórum de discussões.
2. Os arquivos `.c` enviados **não devem conter função** `main`. Evidentemente, você pode incluir a função `main` durante o desenvolvimento do trabalho, mas certifique-se de apagá-la antes do envio;
3. Como mencionado em 2, você não deve modificar os arquivos `.h`, a menos dos espaços marcados com `/* PREENCHA */`;
4. Programas que utilizem bibliotecas externas recebrão nota **zero** (com exceção, obviamente, de `stdlib` e `stdio`);
5. É **vetado** o uso de IA para elaboração do trabalho. Se for constatado o uso de IA, o trabalho receberá nota **zero** e o caso será encaminhado para a coordenação.