UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação

SCC0502 – Algoritmos e Estruturas de Dados I

Prof. Dr. Renato Moraes Silva

Avaliação prática

Instruções gerais

1. **Leia atentamente** a essas instruções e, também, às descrição dos exercícios para garantir que você está executando o que foi pedido.
   - O não atendimento de qualquer item descrito neste documento, implicará perda de nota.
   - O não atendimento ao que tenha sido pedido por qualquer exercício poderá implicar nota 0 (zero) naquele exercício.
2. O trabalho pode ser feito em grupo de no **mínimo dois integrantes** e no **máximo até quatro integrantes**.
   - Somente um integrante do grupo deve enviar a atividade.
3. Siga boas práticas de programação, pois, caso contrário, poderá causar perda de nota:
   - dar nomes intuitivos para as variáveis;
   - dar nomes intuitivos para as funções;
   - comentar o código.
4. **Tentativa de fraude**: cuidado com plágio ou outros tipos de fraude. Qualquer tipo de fraude resultará em **nota zero**.
   - Se for detectado plágio entre grupos, a punição será dada para todos os envolvidos: todos do grupo que copiou e todos do grupo que forneceu a cópia.
   - A detecção de plágio em qualquer um dos exercícios **implicará nota 0 (zero) em todo o trabalho**.
   - É **permitido** usar, com ou sem modificação, qualquer **trecho dos códigos feitos durante as aulas** de laboratório e disponibilizados pelo seu professor no sistema e-Disciplinas. Isso não será considerado fraude.
5. Alguns exercícios dessa lista pedem funcionalidades que nem sempre são vantajosas. Essas funcionalidades são pedidas apenas por motivos didáticos. O objetivo é avaliar sua compreensão em relação às estruturas de dados envolvidas ou para ajudar a desenvolver sua lógica de programação. Em caso de dúvida, consulte seu professor.
6. **Erros de compilação/execução:** antes de submeter o trabalho, certifique-se que não há erros de código. Exercício que não possam ser compilados receberão nota 0 (zero).

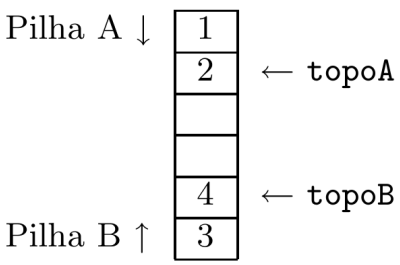
O que deve ser submetido?

- Cada grupo deverá submeter um arquivo .zip contendo:
  - 1 arquivo chamado **grupo.txt** informando o nome e N^o USP de todos os componentes do grupo;
  - 1 pasta zip para cada exercício nomeadas como **exerc1**, **exerc2**, ..., **exercN**
  - Na pasta de cada exercício, devem ser adicionados apenas os arquivos .h e .c utilizados para esse exercício. Não deve haver nenhum outro tipo de arquivo ou pasta dentro.
  - Espera-se que a pasta de cada exercício contenha:
    - **um arquivo .h para cada estrutura de dados** usada no exercício. Nesse arquivo devem ser adicionadas definições de tipo, constantes, e protótipos das funções que fazem parte da sua interface pública
    - **um arquivo .c para cada estrutura de dados** com o mesmo nome do .h. Esse arquivo deve implementar as funções declaradas no arquivo .h.
    - **um arquivo chamado main.c** que deve ser usado apenas para chamar e testar as funções que implementam o que foi pedido no exercício. Essa função deve chamar todos os arquivos .h necessários.
- A violação de qualquer uma das regras acima, poderá acarretar **nota 0 no exercício**.

Exerc. 1. Escreva uma função chamada 'verificaBalanceamento' que receba uma fila contendo uma sequência de parênteses '()', colchetes '[]' e chaves '{}'. A função deve usar uma pilha para verificar se a sequência de caracteres está corretamente balanceada. A sequência está balanceada se cada tipo de abertura de parêntese, colchete ou chave tem uma correspondente e os pares são corretamente fechados na ordem reversa.

- Você deve usar as estruturas de Pilha e Fila ensinadas na disciplina e disponibilizadas no e-Disciplinas. A única coisa que pode ser modificada nessas estruturas é o atributo data para permitir receber caracteres.
- As únicas operações permitidas para acessar os valores da pilha e fila são, respectivamente, as operações pop e dequeue.
- Você deve garantir que ao final da função, a pilha e fila passadas como parâmetro estejam intactas, isto é, como os mesmos valores passados como entrada.

Exerc. 2. Crie uma variação de uma pilha estática chamada de **PilhaDupla**. Nessa estrutura de dados, duas pilhas A e B devem compartilhar o mesmo vetor (list), conforme mostrado na seguinte figura:



A estrutura de dados deverá ter as seguintes funções:

- **pushA**: insere um valor no topo da pilha A

- **pushB**: insere um valor no topo da pilha B
- **popA**: remove o valor do topo da pilha A
- **popB**: remove o valor do topo da pilha B
- **clearA**: remove os valores da pilha A
- **clearB**: remove os valores da pilha B
- **imprimirA**: imprime a pilha A.
- **imprimirB**: imprime a pilha B.

Só deve ser emitida uma mensagem de pilha cheia se todas as posições do vetor estiverem ocupadas. Você **não** deve reservar metade do vetor para a PilhaA e metade para a PilhaB. Elas podem crescer enquanto houver espaço vazio no vetor e podem ter tamanhos diferentes ao longo da execução do algoritmo.

Exerc. 3. Implemente uma **lista não sequencial e estática** (LNSE). Essa lista deve ter o mesmo comportamento de uma **lista não sequencial e dinâmica**, mas deve ser guardada dentro de um vetor com capacidade limitada.

Essa classe deve possuir os seguintes métodos:

- **inserir(x, i)**: insere x na posição i da lista real. A lista real pode ser diferente da sequência de valores que está no vetor.
  - Para controlar as posições vazias do vetor sem ter que percorrer toda a lista, guarde elas dentro de uma fila estática. Use a lista estática ensinada na disciplina, limitando seu uso às operações enqueue e dequeue. Quando a LNSE for criada, essa fila deverá ser preenchida com todos os índices do vetor que guardará a LNSE. Quando um valor for inserido na LSNE, remova o índice na fila usando a operação dequeue. Quando um valor for removido na LSNE, adicione o índice na fila usando enqueue.
- **remover(i)**: remove o item que está na posição i da lista real e retorna seu valor.
- **buscar(x)**: busca o item x na lista e retorna sua posição na lista real.
- **size()**: retorna o número de elementos da lista real.
- **clear()**: remove todos os elementos da lista.
- **imprimir**: imprime a lista real, o índice do elemento **head**, o índice do elemento **tail** e, também, os valores que estão no vetor acompanhados do índice do próximo valor na lista. Nas posições vazias do vetor, imprima (-).

Exerc. 4. Modifique a função da lista duplamente ligada ensinada na disciplina.

- Acrescente um parâmetro na lista chamado **ordenado** que poderá assumir o valor 1 ou 0.
- Modifique as funções de inserção para que sempre que uma nova inserção seja feita, o parâmetro **ordenado** receba o valor 0.
- Adicione uma nova função chamada ordenar. Essa função deverá modificar os valores da lista duplamente ligada ordenando eles por ordem decrescente. Ao final dela, o parâmetro **ordenado** deve receber o valor 1.
- Adicione uma nova função chamada **buscaBinaria** que use a busca binaria para buscar um valor nessa lista.
  - Essa função deve ter o mesmo comportamento de uma busca binária: dividir a lista ao meio, procurando o valor na parte esquerda se for menor que o meio, ou à direita se for maior que o meio.

  - A função deve ser recursiva.
  - **Sugestão**: crie uma função auxiliar que retorne o ponteiro do elemento central entre as posições i e j.

- Modifique a função **buscar** para que ela só faça a busca linear se o parâmetro ordenado for **igual a 0**. Se o parâmetro ordenado for igual a 1, deve chamar a função **buscaBinaria** para ordenar.

Exerc. 5. Implemente uma matriz esparsa utilizando uma estrutura baseada em listas ligadas. Cada linha da matriz será representada por uma lista ligada.

Cada nó da lista deve armazenar:

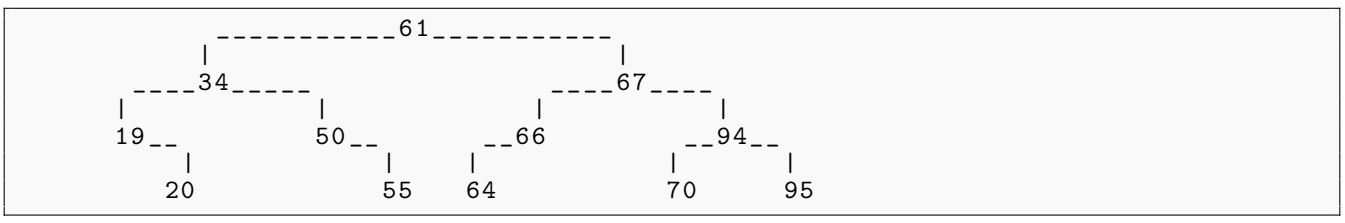
- O valor não-nulo da matriz.
- O índice da coluna correspondente ao valor.
- O índice da linha correspondente ao valor.
- Um ponteiro para o próximo elemento não-nulo na mesma linha.
- Um ponteiro para o próximo elemento não-nulo na mesma coluna.

A estrutura de dados deve conter as seguintes funções:

- **inserir(linha, coluna, valor)**: insere ou atualiza um valor não-nulo na posição especificada (linha, coluna).
  - Se o valor for zero, o elemento deve ser removido da lista (caso ele exista).
  - Se o valor for diferente de zero, ele deve inserir ou atualizar o valor.
- **remover(linha, coluna)**: Remove o elemento na posição especificada, se ele existir.
- **buscar(linha, coluna)**: Retorna o valor na posição especificada (linha, coluna). Se o valor não existir (ou for zero), retorna zero.
- **imprimir()**: Imprime a matriz completa, exibindo os valores em suas respectivas posições.
- **somar(MatrizEsparsa outraMatriz)**: Retorna uma nova matriz esparsa que é o resultado da soma da matriz atual com outra matriz esparsa. As duas matrizes devem ter o mesmo tamanho.
- **somaInterna()**: Soma todos os valores da matriz.
- **calculaEsparsidade()**: Retorna o nível de esparsidade da matriz, que é a proporção de elementos zero em relação ao total de elementos.

Exerc. 6. Modifique a árvore ensinada nessa disciplina criando uma função que receba um valor como entrada, procure esse valor na árvore e retorne a quantidade de subnós (nós abaixo dele, incluindo ele mesmo).

Exerc. 7. Modifique a árvore binária de busca ensinada nesta disciplina. Acrescente nessa árvore uma função chamada **caminhos**. Essa função deve imprimir a soma do valores dos nós que estão no caminho de cada um dos nós folhas até a raiz da árvore. Por exemplo, considere a árvore apresentada abaixo:



Se a árvore acima fosse apresentada como entrada, a função **caminhos** deveria imprimir os seguintes valores [134, 200, 258, 292, 317], que é formado pelas seguites somas: [61+34+19+20, 61+34+50+55, 61+67+66+64, 61+67+94+70, 61+67+94+95].