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 três integrantes** e no **máximo cinco 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).
7. Todos os exercícios devem ser feitos em linguagem C. Só podem ser usadas as seguintes bibliotecas básicas: <stdio.h>, <stdlib.h> e <string.h>. Caso você veja necessidade de usar alguma outra biblioteca, consulte o professor para verificar se é permitido.

8. Todos os exercícios devem aceitar execução com passagens de parâmetros. Caso nenhum parâmetro seja passado, o exercício deve ser executado com pelo menos um teste padrão feito por você ou obedecendo instruções específicas informadas no exercício. Nesses testes padrões, não insira campos que necessitem de entrada do usuário para serem executados.

9. Imprima apenas o resultado que for pedido pelo exercício, sem poluir a tela com impressões complementares.

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 ou mais arquivos .h**. Nesses arquivos devem ser adicionadas definições de tipo, constantes, e protótipos das funções que fazem parte da sua interface pública.
    - **um arquivo .c para cada arquivo .h** com o mesmo nome. 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.
    - **um arquivo que compile e execute** o exercício, podendo ser .bat (Windows) ou .sh (Linux)
- A violação de qualquer uma das regras acima, poderá acarretar **nota 0 no exercício**.
- No processo de avaliação, os exercícios podem receber pesos diferentes. Os pesos não serão informados com antecedência. Portanto, tome cuidado para não dar menor importância para nenhum exercício.

Exerc. 1. Crie um algoritmo recursivo que determine se é possível formar uma expressão numérica com todos os elementos de uma lista de inteiros positivos, utilizando apenas os operadores `+`, `-`, `*` e `/` (divisão inteira), de forma que o resultado final seja igual a um valor alvo.

A expressão deve ser construída combinando, a cada passo, dois valores ainda não utilizados, aplicando entre eles um dos operadores permitidos. O resultado da operação substitui os dois operandos na próxima chamada recursiva. Esse processo continua até restar apenas um valor, que será comparado ao valor alvo.

A implementação deve ser puramente recursiva, sem uso de laços (`for`, `while`) ou alocação dinâmica de memória. Utilize vetores de tamanho fixo.

A função deve imprimir a **primeira expressão encontrada** (com parênteses) que resulte no valor alvo, se ela existir, e também exibir o número total de chamadas recursivas realizadas.

Regras adicionais:

- Cada número da lista deve ser usado exatamente uma vez.
- A divisão só deve ser realizada se o segundo operando for diferente de zero e a divisão for exata (sem resto).
- A função deve lidar com até 6 valores inteiros.

Exemplo:

- Lista: `3,4,2`
- Valor alvo: `14`
- Expressão esperada:

```
((3 + 4) * 2) = 14
```

Teste do algoritmo

O algoritmo deve aceitar chamadas com parâmetros no seguinte formato:
`main <lista> <alvo>`

- `<lista>` é uma sequência de inteiros separados por vírgula.
- `<alvo>` é um número inteiro positivo indicando o resultado desejado da expressão.

Se nenhum parâmetro for informado, o algoritmo deve executar automaticamente dois testes:

1. Lista: `3,4,2`, alvo: `14`
2. Lista: `5,2,1,8`, alvo: `16`

Sugestões

- Use um vetor auxiliar para armazenar os valores intermediários.
- Utilize outro vetor de strings para representar a expressão associada a cada valor.
- Crie uma função auxiliar recursiva para gerar as combinações e avaliar as expressões.

Saída

Para cada teste (automático ou via parâmetros), o programa deve imprimir:

- A expressão encontrada (com parênteses), caso exista.
- A contagem de chamadas recursivas realizadas.
- Caso não exista solução, deve ser exibida a mensagem `Não foi possível formar o valor alvo.`

Exerc. 2. Crie um algoritmo que execute uma busca binária modificada capaz de localizar múltiplos valores em uma única lista ordenada, retornando a primeira e a última ocorrência de cada valor sem usar iteração ou busca linear adicional. A busca para encontrar as ocorrências repetidas deve também seguir a estratégia de divisão da lista. A lista pode estar ordenada de forma crescente ou decrescente, e o algoritmo deve lidar corretamente com ambas as situações. Além disso, a implementação deve ser puramente recursiva, sem utilizar laços ou controle explícito de índices fora das chamadas recursivas.

A função deve retornar um vetor com os pares de índices correspondentes a cada valor buscado. Se o valor não estiver presente na lista, o par deve ser `-1,-1`.

Exemplo:

- Lista: `[8,8,6,5,5,5,3,2,1]` (ordenada decrescentemente)
- Valores buscados: `5,8,7`
- Resultado esperado: `(3,5), (0,1), (-1,-1)`

Teste do algoritmo

O algoritmo deve tratar chamadas com parâmetros no seguinte formato:
`main <lista> <ordem> <valores>`

- `<lista>` é uma sequência de inteiros separados por vírgula.
- `<ordem>` deve ser `asc` para crescente ou `desc` para decrescente.
- `<valores>` é uma sequência de inteiros separados por vírgula indicando os valores a serem buscados.

Se nenhum parâmetro for informado, o algoritmo deve executar automaticamente dois testes:

1. Lista: `1,2,3,3,3,4,5,6`, valores: `3 e 7`
2. Lista: `8,8,6,5,5,5,3,2,1`, valores: `5,8,7`

Saída

Para cada teste (automático ou por parâmetro), o algoritmo deve imprimir para cada valor buscado, o índice de início e de fim.

Exerc. 3. Implemente uma tabela hash para armazenar os dados de alunos, em que cada aluno é representado por uma estrutura com os seguintes campos: número USP, nome e curso.

A chave utilizada pela função hash deve ser baseada no número USP, que é um identificador numérico único. Esse valor deve ser armazenado em uma variável do tipo `unsigned int`, localizada dentro da estrutura do aluno.

A função hash deve seguir os seguintes passos:

1. Para cada dígito do número USP, some 1 ao valor do dígito e multiplique pelo i-ésimo número primo, onde i representa a posição do dígito (iniciando em 1). Por exemplo: o primeiro dígito é multiplicado por 2, o segundo por 3, o terceiro por 5, e assim por diante.

2. Calcule o valor binário de 32 bits correspondente ao número resultante da etapa anterior. Em seguida, inverta a primeira metade (16 bits mais à esquerda) com a segunda metade (16 bits mais à direita), mantendo os 32 bits totais da representação.

3. Aplique uma operação **XOR entre bits de diferentes posições** da seguinte forma: o primeiro bit do primeiro dígito (após a transformação anterior) deve ser feito `XOR` com o último bit do segundo dígito; o resultado deve ser feito `XOR` com o primeiro bit do terceiro dígito; em seguida, com o último bit do quarto dígito; e assim sucessivamente, alternando entre o primeiro e o último bit dos próximos dígitos, até o final do número.

4. O valor final da função hash deve ser o número obtido ao final dessas operações, reduzido por `mod` m, onde m é o tamanho da tabela.

As operações com os bits devem ser realizadas diretamente sobre a variável do tipo `unsigned int`, utilizando operadores bit a bit, sem a necessidade de converter o número para vetores ou strings.

O sistema que implementa a tabela hash deve receber um parâmetro informando o tamanho desejado e criar uma tabela hash com um tamanho igual ao primeiro número primo mais próximo ao valor desejado.

Seu sistema precisa suportar duas operações principais:

- **inserir(nusp, nome, curso)**: insere a estrutura contendo num. USP, nome e curso na tabela, sendo que o núm. USP é a chave.

- **buscar(nusp)**: retorna os dados do aluno que contém o núm. USP igual a *nusp*.

1. A função hash é a descrita anteriormente.

2. A estratégia de colisão inicial deve ser **o reespalhamento quadrático**.

3. Caso a taxa de ocupação ultrapasse 70%, o sistema deve **automaticamente substituir o reespalhamento quadrático pelo reespalhamento duplo**, utilizando uma segunda função hash definida como:

$$h_2(x, i) = (h_1(x) + i \cdot h_3(x)) \bmod m$$

onde $h_1(x)$ é a função hash original solicitada anteriormente, e $h_3(x) = 1 + (x \bmod (m-1))$, sendo m o tamanho da tabela hash. .

4. Quando a estratégia for trocada, todos os elementos já inseridos devem ser **re-espalhados** usando a nova estratégia, sem perda de dados.

5. Caso o fator de carga for maior que 0.9, a tabela deve ser **redimensionada** para o próximo número primo maior que o dobro do tamanho atual, adotando reespalhamento quadrático se o fator de carga for menor que 0.7, e reespalhamento duplo caso contrário..

Teste do algoritmo

O algoritmo deve tratar chamadas com parâmetros nos seguintes formatos:

- `main criar <tamanho> <path>`
  - Cria uma tabela hash com o tamanho especificado (conforme descrito anteriormente) e a salva em um arquivo `.txt` no *path* informado. A primeira linha desse arquivo informa o tamanho da tabela hash.
- `main inserir <path> <nusp`$_0$`>,<nome`$_0$`>,<curso`$_0$`>,<nusp`$_1$`>,<nome`$_1$`>,<curso`$_1$`>, ...`
  - Considera que a tabela hash já existe no *path* especificado;
  - Carrega a tabela na memória;
  - Insere os dados dos alunos passados como parâmetro;
  - Salva a tabela hash atualizada no *path*. Cada aluno deve ser registrado em uma linha, com os campos separados por ponto e vírgula.
- `main buscar <path> <nusp`$_0$`>,<nusp`$_1$`>, ...`
  - Considera que a tabela hash já existe no *path* especificado;
  - Carrega a tabela na memória;
  - Imprime os dados dos alunos cujos N° USP correspondam às chaves informadas como parâmetro.
- `main remover <path> <nusp`$_0$`>,<nusp`$_1$`>,...`
  - Considera que a tabela hash já existe no *path* especificado;
  - Carrega a tabela na memória;
  - Remove os dados de cada aluno passado como parâmetro que for encontrado na tabela. Não se esqueça de criar uma lápide nos registros removidos.
  - Salva a tabela hash atualizada no *path*.

Se nenhum parâmetro for informado, o algoritmo deve executar os seguintes testes automaticamente:

1. Criar uma tabela hash com tamanho 11;
2. Inserir dados de cinco alunos;
3. Remover os dados de dois alunos;
4. Salvar a tabela no diretório raiz do projeto com nome `hash.txt`.
5. Carregar a tabela *hash* armazenada no arquivo `hash.txt` e fazer a busca de dois alunos cadastrados e de 1 que não esteja cadastrado.

Saída

Na criação da tabela *hash*, inserção de valores ou remoção de valores, deve ser impresso apenas uma mensagem indicando sucesso ou fracasso na operação.

Na operação de busca, devem ser impressos os dados do aluno que contém o núm. USP buscado ou mostrando uma mensagem que indique que a chave não foi encontrada.

Exerc. 4. Assim como no exercício anterior, crie uma tabela hash para armazenar dados de alunos, contendo três informações: número USP, nome e curso. No entanto, nesta versão, o nome do aluno é utilizado como chave da tabela hash.

A função hash deve seguir os passos abaixo:

1. Para cada caractere da string original (sem compressão), calcule:

$$h_1 = \sum_{i=0}^{n-1} (\text{ascii}[i] \ll (i \bmod 8))$$

   onde `ascii[i]` é o valor do caractere na posição `i` e << representa deslocamento à esquerda.

2. Aplique uma rotação à esquerda de 13 bits sobre o valor total, ou seja, mova os bits do valor para a esquerda em 13 posições, fazendo com que os bits que saem pela esquerda retornem pela direita.

   Por exemplo, suponha que o valor total seja representado por 16 bits apenas para facilitar a visualização:

$$\texttt{valor} = \texttt{1011001110001111}$$

   Após a rotação à esquerda de 3 bits (como exemplo simplificado), o resultado seria:

$$\texttt{rotl(valor, 3)} = \texttt{1001110001111101}$$

   Note que os três bits mais à esquerda (`101`) foram movidos para o final.

3. Retorne $h_2 = h1 \mod m$, onde m é o tamanho atual da tabela hash.

Para o tratamento de colisões, use a técnica de encadeamento:

- Cada posição da tabela deve conter uma lista ligada.
- Cada nó da lista deve conter:
  - O nome do aluno, núm. USP e curso;
  - Um ponteiro para o próximo nó.

Teste do algoritmo

O sistema deve tratar chamadas com parâmetros nos seguintes formatos:

- `main criar <tamanho> <path>`
  - Cria uma tabela hash com o tamanho informado, ajustando para o primo mais próximo.
  - Salva a tabela no caminho especificado, em formato texto.
- `main inserir <path> <nome`$_0$`>:<nusp`$_0$`>:<curso`$_0$`>,<nome`$_1$`>:<nusp`$_1$`>:<curso`$_1$`>, ...`
  - Carrega a tabela a partir do caminho especificado.
  - Insere os dados dos alunos informados, utilizando o nome como chave. Cada linha do arquivo resultante representa um índice da tabela hash. Em caso de colisões, os dados dos alunos que compartilham o mesmo índice devem ser encadeados na mesma linha, seguindo a ordem da lista encadeada.
  - Salva a tabela atualizada no mesmo arquivo.
- `main buscar <path> <nome`$_0$`>,<nome`$_1$`>, ...`
  - Carrega a tabela.
  - Busca os alunos a partir do nome (chave), descomprimindo as strings para comparação.
  - Imprime os dados completos (nome, número USP e curso) se encontrados.
- `main remover <path> <nome`$_0$`>,<nome`$_1$`>, ...`
  - Carrega a tabela.

- Remove os alunos cujos nomes correspondem às chaves informadas.
- Atualiza o arquivo no caminho especificado.

Se nenhum parâmetro for informado, o sistema deve executar as seguintes operações:

1. Criar uma tabela hash com tamanho 11;
2. Inserir cinco alunos com nomes distintos;
3. Remover dois deles;
4. Salvar a tabela no diretório raiz com o nome `hash_nome.txt`;
5. Carregar a tabela salva e buscar dois nomes presentes e um ausente;

Saída

As saídas devem seguir o seguinte padrão:

- Nas operações de criação, inserção ou remoção: exibir mensagem de sucesso ou falha.
- Na busca: exibir os dados completos do aluno se encontrado, ou uma mensagem indicando ausência.

Exerc. 5. O algoritmo Insertion Sort divide lista de números em uma porção ordenada e outra desordenada. A cada iteração, ele faz o processo de inserção, onde um número da porção desordenada é colocado na porção ordenada. Se você analisar essa parte do algoritmo Insertion Sort, você irá perceber que esse processo de inserção usa um algoritmo de busca linear. Implemente uma versão melhorada do Insertion Sort usando o algoritmo de busca binária para encontrar a posição p onde a nova inserção deve ser colocada.

Crie um algoritmo que receba duas palavras e verifique se elas são anagramas, usando o algoritmo **Insertion Sort com busca binária** para ordenar os caracteres de cada palavra. O algoritmo não deve ser sensível a maiúsculas e minúsculas.

Teste do algoritmo

O algoritmo deve tratar chamada com parâmetros no seguinte formato:
`main <param1> <param2>`

- `<param1>` é a primeira palavra e `<param2>` é a segunda palavra

Se nenhum parâmetro for informado, o algoritmo deve mostrar dois testes automáticos:

1. Palavras: `Pedro` e `poder`
2. Palavras: `Brasil` e `brasileiro`

Saída

Para cada teste (automático ou com parâmetros), o algoritmo deve imprimir exatamente a seguinte mensagem:

- Palavra 1: `<param1>` – Palavra 2: `<param2>` – Anagrama? `<resposta>`

Onde:

- `<resposta>` deve ser `sim` ou `não`

Exerc. 6. Crie um algoritmo recursivo que resolva uma versão estendida do problema das **Torres de Hanói**, incorporando ordenação, rastreamento de peso e exibição detalhada dos movimentos. A implementação deve respeitar os princípios de **divisão e conquista**, **recursividade** e **ordenação**.

Antes de iniciar a movimentação dos discos, o algoritmo deve ordenar os pesos fornecidos pelo usuário utilizando o algoritmo `quicksort` ou `heapsort`, de acordo com a escolha do usuário. Em seguida, o algoritmo deve realizar os movimentos das torres com base na versão tradicional do problema, respeitando as seguintes regras:

- Apenas um disco pode ser movido por vez.
- Um disco maior nunca pode ser colocado sobre um disco menor.
- A cada movimento, o sistema deve exibir:
  - Qual disco foi movido (índice no vetor ordenado).
  - De qual torre para qual torre o movimento foi feito.
  - O peso do disco movido.
- A carga total movimentada (soma dos pesos de todos os movimentos) deve ser acumulada e exibida ao final.

Toda a lógica central do algoritmo (quicksort, heapsort e movimentação) deve ser feita com recursão.

O código não deve usar estruturas de dados avançadas (como listas encadeadas, filas ou árvores). Apenas vetores.

Teste do algoritmo

O algoritmo deve ser executado por linha de comando com os seguintes parâmetros:
`main <algoritmo> <peso`$_0$`> <peso`$_1$`> ...`

- `<algoritmo>` deve ser `quick` ou `heap`, indicando o algoritmo de ordenação a ser utilizado.
- `<peso`$_i$`>` representa o peso de cada disco.

Se nenhum parâmetro for informado, o algoritmo deve executar automaticamente dois testes com ambos os algoritmos de ordenação:

1. Pesos: `8,1,5` com `quicksort`
2. Pesos: `3,9,2,4` com `heapsort`

Saída

O algoritmo deve imprimir:

- O algoritmo de ordenação utilizado.
- A sequência completa de movimentos realizados.
- O número total de movimentos.
- A carga total movimentada (soma dos pesos de cada movimento).

Exerc. 7. O Problema das N Rainhas consiste em posicionar N rainhas em um tabuleiro $N \times N$ de forma que nenhuma rainha ataque outra, ou seja, que não haja mais de uma rainha na mesma linha, coluna ou diagonal.

Desenvolva uma solução que encontre **todas as soluções distintas** possíveis para o problema das N rainhas, considerando equivalência por simetria (rotações e reflexões do tabuleiro).

A abordagem deve utilizar a técnica de **divisão e conquista** e **busca por retrocesso** (backtracking), implementada de forma recursiva.

Objetivos

- Encontrar todas as soluções possíveis para o problema das N rainhas.
- Eliminar soluções simétricas, considerando rotações (90°, 180°, 270°) e reflexões horizontais, verticais e diagonais.
- Exibir todas as soluções distintas (uma por linha), no formato especificado abaixo.
- Exibir o número total de soluções distintas.

Estruturas de Dados

Utilize as seguintes estruturas de dados auxiliares para a implementação:

- Um vetor de tamanho N para representar a posição das rainhas, onde o índice representa a coluna e o valor em cada índice representa a linha correspondente.
- Vetores auxiliares para verificar rapidamente se uma linha ou diagonal já está ocupada.
- Um conjunto para armazenar e comparar as soluções únicas, levando em conta simetrias.

Restrições de Implementação

- A solução deve ser implementada de forma recursiva, utilizando backtracking.
- Soluções simetricamente equivalentes devem ser consideradas como uma única solução distinta.

Teste do algoritmo

O algoritmo deve aceitar chamadas com parâmetros no seguinte formato:
`main <N>`

- `<N>` é um número inteiro positivo indicando o número de rainhas e o tamanho do tabuleiro.

Se nenhum parâmetro for informado, o algoritmo deve executar automaticamente dois testes:

1. Número de rainhas: `4`
2. Número de rainhas: `8`

Saída

Para cada teste (automático ou via parâmetros), o programa deve imprimir:

- Cada solução distinta encontrada, como um vetor com as posições das rainhas.
- O número total de chamadas recursivas realizadas.
- A quantidade total de soluções distintas encontradas.
- Caso não exista solução, deve ser exibida a mensagem `não há solução para <N> rainhas.`