DCC012 – Estruturas de Dados II – Trabalho Parte 1
Prof. Bárbara Quintela

Avaliações de Jogos de Tabuleiro

O BoardGameGeek é um sistema de avaliações de jogos de tabuleiro para usuários cadastrados. Cada usuário pode atribuir uma nota para cada jogo de tabuleiro, e essa informação pode ser utilizada por outros usuários. Seu grupo ficou encarregado de implementar uma funcionalidade para determinar as categorias de jogos mais avaliados e os usuários mais ativos. Uma vez que o volume de avaliações postadas diariamente é alto, a sua solução deve ter um bom desempenho. Para que isso seja garantido, seu grupo deverá analisar o desempenho de algoritmos de ordenação e de tratamento de colisões em tabelas *hash* em diferentes cenários, descritos a seguir. Para os algoritmos de ordenação, esta análise consiste em comparar os algoritmos considerando três métricas de desempenho: número de comparações de chaves, o número de cópias de registros realizadas, e o tempo total gasto para ordenação (tempo de processamento e não o tempo de relógio). As métricas de desempenho para os algoritmos de tratamento de colisão são: número de comparações de chaves e gasto de memória. Em seguida, você deve escolher o melhor algoritmo de ordenação e utilizá-lo para ordenar a frequência dos usuários mais ativos. O melhor algoritmo de tratamento de colisões deve ser utilizado para armazenar os registros, conforme descrito abaixo.

Você deverá utilizar um conjunto real de registros para os experimentos. No link https://www.kaggle.com/jvanelteren/boardgamegeek-reviews#bgg-13m-reviews.csv, há em torno de 13 milhões de avaliações de jogos de tabuleiros, sendo que cada jogo possui mais de 30 avaliações até Maio de 2019, armazenadas em um arquivo CSV: “bgg-13m-reviews.csv”, contendo ID do usuário, ID do jogo, avaliação (variando de 0 a 10), comentário (se houver) e o nome do jogo. O arquivo bgg-13m-reviews.csv se relaciona com os demais arquivos da base de dados pelos campos identificadores. Por exemplo, o campo ID pode ser usado para obter as informações detalhadas sobre o jogo, como a categoria (boardgamecategory), no arquivo "games_detailed_info.csv" ou número de avaliações de cada jogo (Users rated) no arquivo "2019-05-02.csv".

1 – Análise dos Algoritmos

Cenário I: Impacto de diferentes estruturas de dados

Neste cenário, você deverá avaliar o desempenho do método de ordenação Quicksort (recursivo) considerando dois tipos de entradas:

1. Os elementos a serem ordenados são inteiros armazenados em um vetor de tamanho N
   - ~~`ID` (i.e., identificador do jogo – tipo inteiro - **chave para ordenação**)~~
   - # (identificador da avaliação, primeira coluna)
2. Os elementos a serem ordenados são registros/objetos armazenados em um vetor de tamanho N. Cada registro contém:
   - `USER` (i.e., identificador do usuário – tipo char)
   - `RATING` (i.e., avaliação - tipo inteiro)
   - ~~`ID` (i.e., identificador do jogo – tipo inteiro - **chave para ordenação**)~~
   - # (identificador da avaliação, primeira coluna)

Note que os arquivos estão no formato CSV (*Comma-separated values*). Isso significa que cada campo listado acima é separado do outro por uma vírgula. Obs: O arquivo original contém mais campos que podem ser ignorados nesse momento. Considere realizar um pré-processamento.

Para cada tipo de dado, você deverá implementar o Quicksort Recursivo (como apresentado em sala de aula) que recebe, como entrada, o conjunto de elementos a serem ordenados e o número de elementos a serem ordenados. Você deverá instrumentar os algoritmos para contabilizar o número de comparações de chaves, o número de cópias de registros e o tempo total gasto na ordenação. Estes números deverão ser impressos ao final de cada ordenação para posterior análise.

Você ainda deverá implementar funções/métodos para importar os conjuntos de elementos aleatórios. Estes métodos/funções devem ser chamados uma vez para cada um dos N elementos a serem ordenados.

Análise:

O algoritmo Quicksort deverá ser aplicado a entradas escolhidas aleatoriamente com diferentes tamanhos (parâmetro N). Para cada valor de N, você deve gerar 5 (cinco) conjuntos de elementos diferentes, utilizando sementes diferentes para o gerador de números aleatórios. Você pode selecionar um valor aleatório entre 1 e o número de bytes do arquivo e importar o registro na linha correspondente ao deslocamento em bytes dado pelo valor selecionado (ou na linha seguinte a esta). Experimente, no mínimo, com valores

de N = 1000, 5000, 10000, 50000 e 100000. Os algoritmos serão avaliados comparando os valores médios das 5 execuções para cada valor de N testado.
O seu programa principal deve receber um arquivo de entrada (`entrada.txt`) com o seguinte formato:

5 → número de valores de N que se seguem, um por linha
1000
5000
10000
50000
100000

Para cada valor de N, lido do arquivo de entrada:

- Gera cada um dos conjuntos de elementos, ordena, contabiliza estatísticas de desempenho
- Armazena estatísticas de desempenho em arquivo de saída (`saida.txt`).

Ao final, basta processar os arquivos de saída referentes a cada uma das sementes, calculando as médias de cada estatística, para cada valor de N e estrutura de dados considerados.

Resultados:

Apresente gráficos e tabelas para as três métricas pedidas, número de comparações, número de cópias e tempo de execução (tempo de processamento), comparando o desempenho médio do Quicksort para os dois tipos de estruturas de dados e diferentes valores de N. Discuta seus resultados. Quais são os compromissos de desempenho observados?

Cenário 2: Impacto de variações do Quicksort

Neste cenário, você deverá comparar o desempenho de diferentes variações do QuickSort para ordenar um conjunto de N inteiros armazenados em um vetor. Cada elemento do vetor deve ser constituído de `ID`, importados **aleatoriamente** da sua base de dados. As variações do Quicksort a serem implementadas e avaliadas são:

- Quicksort Recursivo: este é o Quicksort recursivo apresentado em sala de aula.
- Quicksort Mediana(k): esta variação do Quicksort recursivo escolhe o pivô para partição como sendo a mediana de k elementos do vetor, aleatoriamente escolhidos. Experimente com k = 3 e k = 5.
- Quicksort Insercao(m): esta variação modifica o Quicksort Recursivo para utilizar o algoritmo de Inserção para ordenar partições (isto é, pedaços do vetor) com tamanho menor ou igual a m. Experimente com m =10 e m = 100.

Realize experimentos com as três variações considerando vetores aleatoriamente gerados com tamanho N = 1000, 5000, 10000, 50000, 100000 e 500000, no mínimo. Para cada valor de N, realize experimentos com 5 sementes diferentes e avalie os valores médios do tempo de execução, do número de comparações de chaves e do número de cópias de registros. Estruture o seu programa principal como sugerido acima para facilitar a coleta e posterior análise das estatísticas de desempenho.

Apresente gráficos e tabelas com os resultados obtidos. Discuta os resultados e conclusões obtidos. Qual variação tem melhor desempenho, considerando as diferentes métricas. Por quê? Qual o impacto das variações nos valores de k e de m nas versões Quicksort Mediana(k) e Quicksort Insercao(m)?

Cenário 3: Quicksort X InsertionSort X Mergesort X Heapsort X Meusort

Neste cenário, você irá comparar a melhor variação do Quicksort (justificada pelos resultados do Cenário 2) com o InsertionSort, o Mergesort, o Heapsort, e um outro algoritmo de ordenação de sua escolha (não pode ser nenhum algoritmo dado em sala de aula) para ordenar um conjunto de N inteiros. Cada elemento do vetor deve ser constituído de `ID`, importados **aleatoriamente** da sua base de dados. Você deverá apresentar, no seu relatório, o quinto algoritmo escolhido, explicitando sua fonte.

Realize experimentos considerando vetores aleatoriamente gerados com tamanho N=1000, 5000, 10000, 50000, 100000 e 500000, no mínimo. Para cada valor de N, realize experimentos com 5 sementes diferentes. Para a comparação de Quicksort, InsertionSort, Mergesort, Heapsort e Meusort, avalie os valores médios do tempo de execução, do número de comparações de chaves e do número de cópias de registros. Apresente gráficos e/ou tabelas com os resultados obtidos. Discuta os resultados e conclusões obtidas. Qual algoritmo tem melhor desempenho, considerando as diferentes métricas? Por quê?

Cenário 4: Tratamento de Colisões: Endereçamento X Encadeamento

Neste cenário, você deverá comparar o desempenho de algoritmos para tratamento de colisão considerando duas métricas de desempenho: número de comparações de chaves e o gasto de memória. Os algoritmos de tratamento de colisão a serem implementados são:

- Endereçamento – Sondagem Linear
- Endereçamento – Sondagem Quadrática
- Endereçamento – Duplo Hash
- Encadeamento Separado
- Encadeamento Coalescido (sem porão)

Realize experimentos com os cinco algoritmos considerando vetores aleatoriamente gerados com tamanho N = 1000, 5000, 10000, 50000, 100000 e 500000, no mínimo. Cada

elemento do vetor deve ser constituído de ~~(USER, ID)~~ (USER, #), importados aleatoriamente da sua base de dados. **As avaliações não devem ser repetidas**. Para cada valor de N, realize experimentos com 5 sementes diferentes e avalie os valores médios do número de comparações de chaves e do gasto de memória. Estruture o seu programa principal como sugerido nos outros cenários para facilitar a coleta e posterior análise das estatísticas de desempenho. Observe que `USER` é um campo char e portanto deve ser tratado para representar um valor inteiro junto ao ~~ID~~ `# (identificador da avaliação)`.

Apresente gráficos e tabelas com os resultados obtidos. Discuta os resultados e conclusões obtidos. Qual algoritmo tem melhor desempenho, considerando as diferentes métricas? Por quê? Qual o impacto na escolha do tamanho da *hash table*?

2 – Implementação das Categorias Frequentes e dos Usuários Ativos

Você deverá implementar um programa que leia N jogos aleatórios no arquivo de informações sobre os jogos "games_detailed_info" e conte quantas vezes a mesma categoria se repete dentro desses N jogos de tabuleiro, ou seja, determinar sua frequência. O arquivo de games_detailed_info possui muitos campos, e para essa atividade pode ser pré-processado para conter apenas os ~~dois~~ três campos descritos a seguir:

- # (identificador da avaliação - tipo inteiro, primeira coluna)
- `ID` (i.e., identificador do jogo - tipo inteiro)
- `boardgamecategory` (i.e., categoria do jogo, que é uma string contendo uma ou mais categorias entre aspas simples e separadas por vírgula dentro de colchetes. Sintaxe: por exemplo contendo apenas uma categoria `['Trains']` ou contendo várias categorias `['Ancient', 'Card Game', 'City Building', 'Civilization']`

Você também deverá verificar quais são os usuários mais ativos no site. Para isso, você deverá determinar a frequência do campo `USER` no arquivo "bgg-13m-reviews.csv".

Você deverá utilizar a estrutura de dados *Hash Table* para armazenar as avaliações selecionadas e também para pesquisar e inserir as categorias dos filmes. **Lembre-se que não deve haver avaliações repetidas.** Você então deverá implementar uma *Hash Table* de avaliações e uma *Hash Table* para categorias. Para as duas, você deverá implementar uma função *hash* que suporte chaves que são sequências de caracteres (códigos ASCII). As funções *hash* devem apresentar duas propriedades básicas: seu cálculo deve ser rápido e deve gerar poucas colisões. Além disso, é desejável que ela leve a uma ocupação uniforme da tabela para conjuntos de chaves quaisquer.

Para resolver as colisões da *Hash Table* de avaliações, você deverá utilizar o melhor resultado obtido no Cenário 4. O tamanho da tabela também deve levar em conta os resultados obtidos nesse cenário.

Seu programa deve ler cada avaliação da *Hash Table* de avaliações. Cada categoria encontrada deve ser armazenada na *Hash Table* de categorias. No final da execução, seu programa deve imprimir as N (parâmetro definido pelo usuário) palavras-chave mais frequentes. A saída deve ser ordenada de forma decrescente de frequência. Para ordenação, escolha o algoritmo de ordenação segundo o Cenário 3. O grupo pode usar a sua criatividade para implementar a *Hash Table* de categorias, mas será avaliado de acordo com a qualidade da sua solução.

Considerações

1. Todo código fonte deve ser documentado. A documentação inclui, dentre outros, a documentação de procedimentos, de funções, de variáveis, de partes do código fonte que realizam tarefas específicas. Ou seja, o código fonte deve ser documentado tanto em nível de rotinas quanto em nível de variáveis e blocos funcionais.
2. A interface pode ser feita em modo texto (terminal) ou modo gráfico e deve ser funcional.
3. A implementação deve ser realizada usando a linguagem de programação C, C++ ou Java.

Entrega

O grupo deverá ser formado por no máximo 4 alunos e as responsabilidades de cada aluno deve ser documentada e registrada. O prazo final para entrega é dia **14/10**. Deverá ser agendada uma data para apresentação do trabalho para o(a) professor(a) no final do período.

Devem ser entregues os códigos implementados e um relatório com os seguintes itens:

1. Descrição das atividades realizadas por cada membro do grupo
2. Análises dos cenários da Parte 1
3. A explicação sobre as estruturas de dados implementadas na Parte 2

Critérios de avaliação

Você não fechará o trabalho só tendo um “sistema que funciona”. O sistema deve funcionar bem e o quão bem ele funcionar será refletido na sua nota. A nota poderá ser comparativa, então se esforce para ter uma solução melhor que a dos outros colegas. O objetivo do trabalho é testar a sua capacidade de fazer boas escolhas (e boas adaptações) de estruturas para resolver problemas. **Então usar classes prontas ou métodos prontos não**

são permitidos aqui. Você poderá, se quiser, comparar sua solução com outras prontas. Mas deve perseguir o seu melhor sem usar recursos de terceiros.

Os membros da equipe serão avaliados pelo produto final do trabalho e pelos resultados individuais alcançados. Assim, numa mesma equipe, um membro pode ficar com nota 90 e outro com nota 50, por exemplo. Dentre os pontos que serão avaliados, estão:

- Execução do programa (caso o programa não funcione,a nota será zero)
- Código documentado e boa prática de programação (o mínimo necessário de variáveis globais, variáveis e funções com nomes de fácil compreensão, soluções elegantes de programação, código bem modularizado etc.)
- Testes: procure fazer testes relevantes como, por exemplo, aqueles que verificam casos extremos e casos de exceções
- Relatório bem redigido

Note que o grande desafio deste trabalho está na avaliação dos vários algoritmos nos diferentes cenários, e não na implementação de código. Logo, na divisão de pontos, a documentação receberá, no mínimo, 50% dos pontos totais. Uma boa documentação deverá apresentar não somente resultados brutos mas também uma discussão dos mesmos, levando a conclusões sobre a superioridade de um ou outro algoritmo em cada cenário considerado, para cada métrica avaliada.