UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

Ordenação de Vetores utilizando árvores

Eliton Lima Rosendo Filho - 493508

Guilherme Girão Alves - 494336

Resumo

Neste documento estão sendo apresentados os métodos para a elaboração e desenvolvimento do sistema proposto para o trabalho da disciplina de Estrutura de Dados: Ordenação de Vetores utilizando árvores.

Desenvolvimento

O desenvolvimento do projeto foi auxiliado pelas seguintes ferramentas: Discord, onde os integrantes da equipe conversavam entre si ao decorrer dos dias em que eles se encontravam para debater sobre possíveis erros e as dificuldades que surgiram no decorrer do trabalho. Visual Studio Code, para programar o código, juntamente com a extensão Live Share para poderem editar o código fonte do trabalho simultaneamente.

Descrição do problema

O problema a ser resolvido no trabalho é desenvolver, com base nas etapas apresentadas no documento do projeto, um método de ordenação de vetores utilizando árvores binárias. Onde, recebemos o tamanho do vetor e seus respectivos elementos em um arquivo de entrada, ordenamos com nosso algoritmo e retornamos para um arquivo de saída, o vetor ordenado.

Descrição do algoritmo

Ao algoritmo do projeto demos o nome de Tree Sort, ou ordenação em árvore. O algoritmo consiste em criar uma árvore binária de altura $\lceil \log_2 n \rceil + 1$ (onde n é o número de elementos do vetor) a partir dos dados de um vetor e populá-la de baixo para cima, ou seja, os elementos do vetor se tornarão folhas (as folhas restantes serão preenchidas por um valor que é o máximo elemento do vetor + 1, valor utilizado como auxiliar para verificar se um elemento do vetor já foi ordenado, chamaremos no relatório de **maxValue**). A execução e decisões tomadas para realizar o funcionamento deste algoritmo estão no próximo tópico.

Descrição das estruturas de dados e decisões tomadas

Utilizamos duas estruturas de dados: **Árvores Binárias (Binary Trees) e Filas (Queues).**

As árvores binárias são estruturas que armazenam informações em forma de nós, que, ao serem interligados, formam uma árvore, ou seja, temos um nó principal, onde o chamamos de raiz e seus [0.. 2] nós filhos que são interligados ao nó raiz, cada filho também pode conter até outros dois nós associados à eles, nós que não tem filhos associados são chamados de nós folha. Útil para quando queremos um sistema de hierarquia ou prioridade para nossos dados, no caso do

projeto, é utilizada para priorizarmos o menor elemento do vetor de estados (abordaremos sobre mais a frente), através de sucessivas transformações dos nós pais e descendentes para o valor do menor filho entre eles até chegar na raiz.

As filas são estruturas de dados onde o IO (input e output) de dados funcionam de maneira FIFO (First-In, First-Out), ou primeiro a entrar, primeiro a sair (em tradução literal). Os novos dados vão para o final, e os dados mais antigos na fila são os primeiros a serem executados. São úteis para quando queremos organizar nossos dados de forma sequencial a partir da ordem de chegada. Utilizamos no projeto para armazenar as folhas a partir do vetor de estados na função **generateLeaves()**, e gerar as árvores com base nessas folhas (**buildTree()**). Dessa maneira, geramos as árvores de baixo para cima, onde o elemento pai tem o valor do mínimo valor entre os seus dois filhos (os quais são removidos da fila), e ele é jogado para o final da fila, a fim de processarmos todos os elementos filhos que estão aguardando, até sobrar um elemento, o qual será nosso nó raiz, onde, o mesmo, é o menor elemento do vetor de estados.

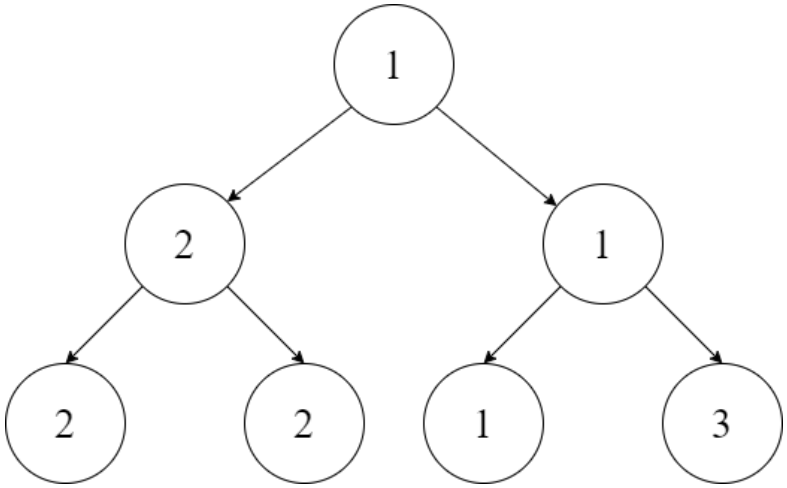
O **vetor de estados (stateVector[])** supracitado é um vetor de tamanho e de valores iniciais iguais ao vetor que desejamos ordenar, onde, a cada menor valor ordenado pela árvore (e inserido em **sortedVector[],** nossa variável utilizada para armazenar o vetor ordenado), a primeira menor ocorrência do valor é substituída pelo **maxValue**, a fim do algoritmo conseguir identificar quem já foi ordenado. Essa foi uma decisão tomada por dois motivos:

1. Havia um problema ao pesquisar o menor valor pela árvore, o qual poderia remover mais de um valor repetido na iteração, funcionamento esse, o qual não é desejado para nosso algoritmo (i.e.: V = [2, 2, 1, 1], logo, V = [1, 2]).
2. Para solucionar o problema acima, teríamos que criar outro método gerador de folhas e de árvores, além dos métodos acima citados (**generateLeaves() e buildTree()**), contudo, estaríamos repetindo código, o que não seria uma boa prática, dado que essas funções estavam em uso apenas para os casos iniciais, então, pensamos em uma solução compatível com o guia do algoritmo:
    a. Toda a árvore é gerada a partir dos menores valores dos pares de folhas;
    b. Quando um valor se torna raíz, ele será ordenado e substituído por **maxValue** em todas as suas ocorrências na árvore;

c. Rodará uma nova iteração para procurar e ordenar o novo menor elemento até que a árvore seja toda preenchida por **maxValue** ou que o total de iterações seja igual ao tamanho do vetor;

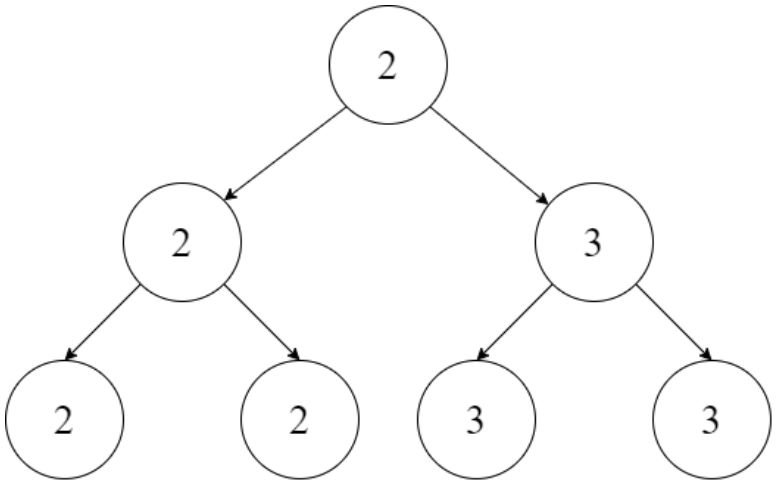
d. Logo, se o valor ordenado é substituído por **maxValue** em todas as ocorrências na árvore, então, podemos substituí-lo apenas em um local, nesse caso, no **vetor de estados** e geramos novas folhas a partir dele, e, consequentemente, uma nova árvore, dessa maneira, o algoritmo tem controle de qual valor já foi ordenado, e a nova árvore já recebe a substituição de valor em suas ocorrências. i.e.: Considere: V = [2, 2, 1] (constante), stateVector = [2, 2, 1], sortedVector = [], maxValue = 3 (constante), altura da árvore: 3, número de folhas: 4, logo:

   i. Na primeira iteração:



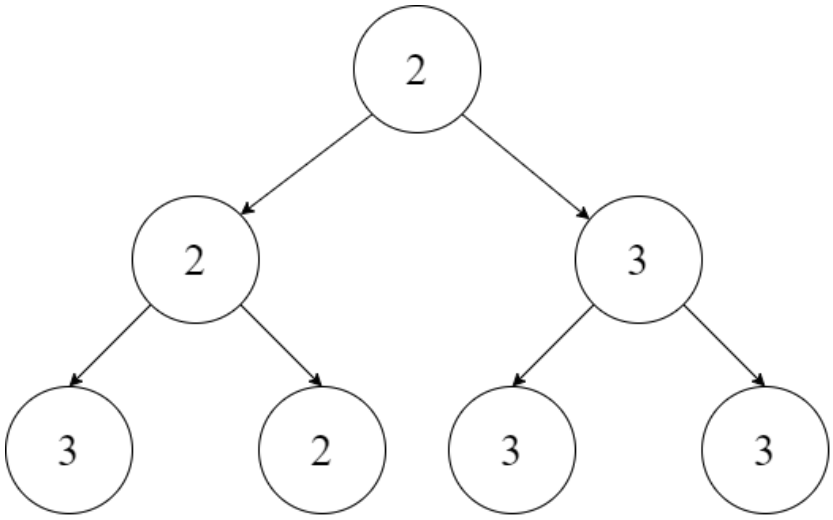
stateVector = [2, 2, 3], sortedVector = [1]

ii. Na segunda iteração:



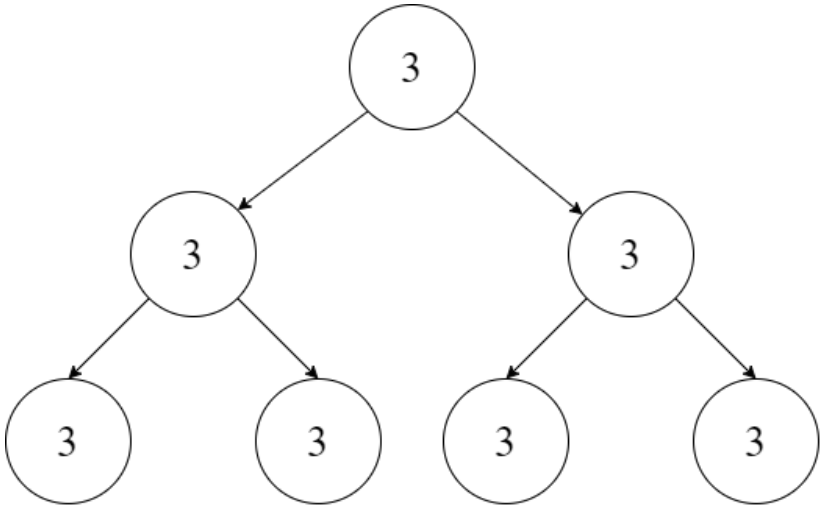
stateVector = [3, 2, 3], sortedVector = [1, 2]

iii. Na terceira e última iteração:



stateVector = [3, 3, 3], sortedVector = [1, 2, 2]

Caso ouvesse mais uma iteração, a árvore ficaria assim:



Porém, dado o funcionamento do nosso algoritmo, o valor 3 seria inserido no vetor ordenado, o que não pode acontecer, visto que 3 é o **maxValue**, e não um elemento do vetor.

Fizemos também, algumas funções extras relativas às árvores, como um **print** simples e a função **empty**, para caso necessário uso. O resto das funções e decisões foram tomadas a partir do documento de orientação do projeto e/ou das atividades relacionadas às Trees e da Queue feita em sala de aula (i.e.: na função **getHeight()**, o cálculo utilizado foi o mesmo relatado no documento ($\lceil \log_2 n \rceil + 1$), etc).

Métodos e estudo da complexibilidade

Observação: Algumas descrições mais detalhadas dos métodos utilizados estão nos comentários do código;

Queue.h contém as funções:

- **QueueNode(Node *key, QueueNode *next = nullptr)**: Construtor para inicialização com key e sem obrigatoriedade de ter o próximo nó.

Complexibilidade: **O(1)**, visto que a função apenas insere os valores passados por parâmetro;

- **Queue()**: Inicializa os nó cabeça e cauda com nullptr.

  Complexibilidade: **O(1)**, visto que o construtor apenas insere os valores com nullptr;

- **~Queue()** : Que percorre toda fila removendo os nós relacionados (next).

  Complexibilidade: **O(n)**, visto que o destrutor percorre todos nós relacionados ao nó cabeça a fim de removê-los;

- **bool empty()**: e retorna se a fila está vazia.

  Complexibilidade: **O(1)**, visto que a função apenas faz uma comparação de igualdade;

- **void push(Node *key)**: Nesta função é inserido um elemento no final da fila. Se a fila estiver vazia, insere o elemento como head e tail, caso contrário, será referenciado como o próximo nó, indo para o final da fila.

  Complexibilidade: **O(1)**, visto que a função apenas atribui um valor para head ou tail;

- **void pop**(): Esta função remove o primeiro elemento da fila, passando o próximo elemento para a primeira posição.

  Complexibilidade: **O(1)**, visto que a função apenas atribui um valor para head ou tail;

- **Node *front**(): Retorna o elemento que está no início da fila.

  Complexibilidade: **O(1)**, visto que a função apenas retorna o elemento que está no início;

- **Node *back**(): Retorna o elemento que está no final da fila.

  Complexibilidade: **O(1)**, visto que a função apenas retorna o elemento que está no final;

- **int size():** Retorna o tamanho da fila, a partir da contagem de quantas iterações são feitas até chegar ao final da fila.

Complexibilidade: **O(n)**, visto que a função percorre por todos os nós fazendo contagem até chegar ao final da final.

Node.h classe referente ao nó da árvore contém as funções:

- **Node(int key, Node *left = nullptr, Node *right = nullptr):** Construtor para inicialização com key e sem obrigatoriedade de ter os nós esquerda e/ou direita.

  Complexibilidade: **O(1)**, visto que o construtor apenas insere os valores passados por parâmetro.

TreeSort.h contém as seguintes funções:

- **bool empty():** Retorna se a árvore está vazia ou não.

  Complexibilidade: **O(1)**, visto que a função apenas faz uma comparação de igualdade;

- **Node *buildTree(Queue leavesQueue):** Método para construção da árvore a partir de uma fila de folhas gerada na função **generateLeaves.**

  Complexibilidade: **O(n)**, visto que a função executa uma iteração de “tamanho da fila” - 1 vezes, no pior dos casos;

- **Queue generateLeaves():** Método para gerar as folhas com base no vetor de estado.

  Complexibilidade: **O(n)**, visto que a função executa uma iteração pelo “número de folhas da árvore (getLeavesNumber())”;

- **int getIndexByRootKey():** Retorna a posição no vetor de estado que contém a chave da raíz.

  Complexibilidade: **O(n)**, visto que a função executa uma iteração pelo “tamanho do vetor a ser ordenado”, no pior dos casos;

- **Node *clear(Node *node):** Função recursiva utilizada para limpar todos os nós da árvore.

  Complexibilidade: **O(2n)**, **logo**, **O(n)**, visto que a função executa, no pior caso, *n* vezes iterações para os nós da esquerda e *n* vezes iterações para os nós da direita;

- **void _print(Node *node):** Função recursiva que printa os nós-filhos (esquerda e direita) a partir do nó passado por parâmetro.

  Complexibilidade: **O(2n)**, **logo**, **O(n)**, visto que a função executa, no pior caso, *n* vezes iterações para os nós da esquerda e *n* vezes iterações para os nós da direita;

- **int _getTreeSize(Node *node)**: Função recursiva que retorna o número de nós existentes a partir do nó passado por parâmetro.

  Complexibilidade: **O(2n)**, **logo**, **O(n)**, visto que a função executa *n* vezes iterações para os nós da esquerda e *n* vezes iterações para os nós da direita, no pior caso;

- **TreeSort(int _vectorSize, int *_vector):** Construtor que recebe o tamanho do vetor e o vetor, constrói e inicia a árvore e executa o método de ordenação, salvando o novo vetor ordenado no atributo *sortedVector.

  Complexibilidade: **O(2n² + n)**, **logo**, **O(n²)**, visto que o construtor executa um loop pelo tamanho do vetor para encontrar o maior valor (+n) e que as funções buildTree() e getIndexByRootKey() que ambas são O(n) estão sendo executada dentro de outro loop, logo, $n^2 + n^2 = 2n^2$, então temos complexibilidade, no pior caso, de $2n^2 + n$, a qual é a mesma coisa que **O(n²)**;

- **~TreeSort():** Destrutor (desaloca da memória todos os atributos do vetor, inclusive os nós da árvore).

Complexibilidade: **O(2n)**, **logo**, **O(n)**, no pior caso, visto que o destrutor executa a função clear() que é O(n);

- **void print():** Printa todos os nós-filhos a partir do nó-raiz.

  Complexibilidade: **O(2n)**, **logo**, **O(n)**, no pior caso, visto que a função executa a função _print(), que é O(n);

- **int *getSortedVector():** Retorna o vetor ordenado.

  Complexibilidade: **O(1)**, visto que a função apenas retorna uma variável, que é o vetor ordenado;

- **int getVectorSize():** Retorna o tamanho do vetor.

  Complexibilidade: **O(1)**, visto que a função apenas retorna uma variável, que é tamanho do vetor;

- **int getTreeSize():** Retorna o número de nós da árvore.

  Complexibilidade: **O(2n)**, **logo**, **O(n)**, no pior caso, visto que a função executa a função _getTreeSize(), que é O(n);

- **int getHeight():** Retorna a altura da árvore.

  Complexibilidade: **O(1)**, visto que a função apenas retorna apenas um cálculo;

- **int getLeavesNumber()**: Retorna o número de folhas da árvore.

  Complexibilidade: **O(1)**, visto que a função apenas retorna apenas um cálculo.

Considerações finais

1. Os arquivos de input.txt e output.txt (quando gerado após a execução do código de ordenação na main) estão no diretório Arquivos;

2. Arquivos para serem compilados: main.cpp TreeSort/TreeSort.cpp, visto que os outros do projeto estão em .h;
3. A complexibilidade do programa como um todo, ou seja, para executar o seu propósito de receber o vetor de um arquivo, ordená-lo e salvar o vetor ordenado em um novo arquivo é de **$O(n^3)$**, pois, na main temos um while que percorre o arquivo linha-por-linha até o fim das mesmas (O(n)), e, dentro dele, instanciamos a nossa classe TreeSort, para fazer a ordenação do vetor recebido do arquivo, a qual é $O(n^2)$, logo, O(n) * $O(n^2)$ = **$O(n^3)$**;
4. Repositório para o Github do projeto: https://github.com/guilhermegirao/estrutura-de-dados/blob/main/projeto-2