Relatório - Árvore de Ordenação
Estrutura de Dados

Equipe : Henricky de Lima, Eliabe Soares
Matriculas: 475075, 470820

Professor: Atílio Gomes

Departamento de Ciência da Computação
Universidade Federal do Ceará (UFC) – Quixadá, CE – Brasil

Novembro 2019

1 Introdução

O problema consiste em realizar a ordenação de um ou mais vetores utilizando a Estrutura de dados da Árvore Binária, onde, dados, um vetor **V** de **n** elementos arbitrários devem ser obtidos de um arquivo (.txt).

Em resumo, o vetor deve ser utilizado criar um arvore binária cheia, ou seja, todas as suas folhas estão na mesma altura, onde todos os valores de suas folhas devem ser preenchidos com os elementos do vetor. Vale lembrar, que uma arvore cheia tem 2^h folhas na sua altura $h - 1$. Em seguida, "popular" os nós pais de cada par de folhas com o menor elemento entre o par de filhos, e assim até a raiz, vale lembrar, que se o tamanho **n** for menor que a quantidade de folhas faz necessário preencher essas folhas de alguma forma, logo foi escolhido usar o maior elemento incrementado em 1, o que chamamos de coringa(ϵ ou E), para não interferir na busca dos menores. E desta forma obter uma arvore que pode ser utilizada na ordenação deste vetor.

No processo de reolução do problema foram divididas tarefas para cada integrante de modo que os dois pudessem praticar os conceitos de arvore, percurso e o método de ordenação proposto. Desta forma, foram porposta motodologias de resolução e aplicá-las na resolução do problema. E através destas soluções definir o grau de complexidade pela análise assintotica da mesma.

2 Divisão de Tarefas

A divisão de tarefas na resolução e implementação do trabalho foram, relativamente, balanceadas, onde, os alunos se reuníam e discutiam o melhor método de resolução do problema. Quanto a implementação, foram feitas, na sua maioria, em conjunto, onde quem deu a ideia a implementava no papel ou computador, testava e faziam-se os devidos ajustes nos prototipos.

3 Metodologia

O problema de ordenação de um vetor utilizando uma arvore é um grande problema que pode ser quebrado em problemas menores e através da solução destes resolver o todo. Foi proposto a divisão do problema em três partes, **criar as folhas** utilizando os elementos do vetor, **popular a arvore** de baixo para cima através da comparação dos valores das mesmas, e fazer o **Percurso e armazenagem** dos valores da arvore de modo que retorne o vetor ordenado. E através destes métodos foram implementadas as função auxiliares **CreateLeaf**, Sessão 3.2, **OrdenaCria**, Sessão 3.3, e **OrdenaVec**, Sessão 3.4.

3.1 Interface QX_Tree

Este trabalho foi escrito em cima dos códigos de Árvore Binária estudados em sala, onde a implementação da árvore consiste em uma estrutura, **noArv**, que carrega um inteiro **int value**,e dois ponteiro de nó da mesma estrutura para o filho esquerdo, **noArv* esq**,e direito, **noArv* dir**. Além disto, uma variável global, **int Eglobal**, foi criada para salvar o valor do coringa(E) e algumas funções bás icas para trabalhar com estes nós de arvore, como criar um nó vazio, criar um nó com valor e filhos como parâmetro, liberar a memória alocada para os nós da arvore, dentre outras.

3.2 Função Auxiliar arv_CreateLeaf

Uma função que recebe um vetor **v** e o seu tamanho **n** e tem como retorno uma fila(Queue) de ponteiros e nós. Foi pensada para receber o vetor e seu tamanho e retornar uma fila de ponteiros de nós, equivalente as folhas da arvore que vai ser criada, já organizada para ser do tamanho $2^h >= n$ e substituindo os valores faltantes pelo coringa (E).

Esta função utiliza uma função auxiliar, **maiorElem(v,n)**, para obter o maior elemento de um vetor, uma constante k com valor 2 que irá incrementar por 2 em uma repetição com a condição do tamanho maior que k, ou seja, k recebe o valor 2^h mais próximo de n.

Por fim, cria uma fila e adiciona os elementos do vetor a fila até chegar no tamanho do mesmo, depois aficiona o valor do coringa a todos os nós acima de n e menores que k, e retorna a fila.

3.3 Função Auxiliar arv_OrdenaCria

Uma função iterativa que tem como entrada uma fila de ponteiros de nós e como saída um ponteiro de nó. Ela tem por objetivo criar a arvore de modo que os pais das folhas tenham o valor igual ao menor valor entre seus filhos e isso vale para todos os nós até a raiz.

Esta função utiliza de quatro ponteiros de nósm sendo eles, dois ponteiros que apontam para cada par de folhas(aux1,aux2), um que aponta para o menor deste par (menor) e um ponteiro de nó que vai ser retornado com a raiz da árvore (raiz).

Utiliza-se uma repetição com a condição da fila ter um tamanho diferente de um, ou seja ter somente um elemento. Nesta repetição os ponteiros aux1 e aux2 recebem os dois primeiros elementos da fila de nós e compara eles para saber qual dos dois tem menor valor, depois cria um nó com a função **arv_Cria**, que recebe o menor valor e os dois nós que comporem o par, onde o ponteiro raiz o recebe, e por fim adiciona este ponteiro ao fim da fila. Esta repetição ocorre até sobrar somente o ponteiro raiz da arvore com os valores adicionados sobre a condição imposta.

3.4 Função Auxiliar arv_OrdenaVec

Uma função interativa que tem como entrada um ponteiro de nó para a raíz da árvore já criada e como retorno uma fila de números inteiros. Esta função tem como objetivo utilizar da arvore criada pela função **arv_OrdenaCria** para retornar uma fila com os elementos do vetor original ordenado, através do método pedido, ou seja, enfileirar e substituindo o menor elemento, o valor da raiz, das folhas pelo coringa e retrabalhar a arvore de modo que, comoo menor saíu, o segundo menor tome seu lugar no valor da raiz.

A função utiliza de duas repetições, do tipo while, aninhadas de modo que a primeira tem como condição o valor da raíz ser diferente do coringa, enquanto que a segunda tem como condição os nós esquedo ou direitos serem nulos, ou seja chegar nas folhas. A segunda repetição tem como objetivo fazer um ponteiro auxiliar (aux) percorrer o caminho do valor da raiz (menor elemento) até a folha com o mesmo valor. Após percorrer o caminho até a folha altera-se o seu valor para o coringa e chama uam função auxiliar, **propaga(raiz)**, que recebe a raíz e retorna a raíz da arvore apósatualiza o menor valor das folhas para a raíz de forma recursiva. Deste modo, sobe a condição da primeira repetição o menor valor vai ser sempre adicionado a fila de inteiros e a arvore é atualizada até a raíz ter o valor do coringa, ou seja, todos os elementos menor que ele foram adicionados a fila.

3.5 Função arv_Ordena

Esta é a função que utiliza todas as funções auxiliares, de modo, a seguir o passo a passo de resolução do problema. Tem como entrada o vetor v e seu tamanho n, e como saída uma fila de inteiros com os elementos ordenados.

Esta função tem a implementação como no Código 3.5.

```
1  std::queue<int> arv_Ordena(int *v,int n){
2
3          std::queue<NoArv*> qLeafs = createLeaf(v,n);
4          NoArv* raiz = arv_OrdenaCria(qLeafs);
5          std::queue<int> ordenado = arv_OrdenaVec(raiz);
6          // desaloca es
7          arv_libera(raiz);
8
9
10         return ordenado;
11 }
```

Formato Final da Função de Ordenação

4 Complexidade da funções

Entrando na área de análise assintótica, pode-se analisar a complexidade de um algoritmo em termos do seu número de operações, notação **O(f(n))**. Deste, modo podemos concluir que o algoritmo criado tem complexidade maior que **O**(**n**), pela soma dos resultados obtidos nas Sessões 4.1 a 4.4.

4.1 createLeaf

O inicio é de seguido três instruções de atribuição, uma delas tem uma chamada de metódo o mesmo tendo complexidade $O(n)$.Em seguida à um laço que é díficil ser preciso na sua complexidade, pois será executado enquanto n seja maior que 2^k, k=[2,+∞], para pode criar árvores binárias cheia. Aproximando a complexidade ficaria entre $0 < O(x) < n$, sendo x o valor de complexidade. Na parte final do metódo, há um laço que se repete k vezes, sendo esse k a variável que esta salvo o valor da potencia de 2 maior que n. De modo geral a complixidade em si do metódo fica comprometida ao valor de k, n pode ser igual a k como pode tender ao $+\infty$. Dividindo em dois casos, se $k = n$ então complexidade $O(n)$, caso $k > n$ complexidade $O(k)$

4.2 arv_OrdenaCria

Pior caso é a fila passada não é vazia, pois vai entrar em um laço que será executado $\left(\frac{n}{2}\right)$ vezes sendo n o número de elementos e a primeira execução é a maior pois na segunda execução serão ligados nós dois a dois e eliminará os dois primeiros nós e será adicionado o nó que liga os dois retirados da fila, isso será repetido até que a fila de nós esteje completamente vazia. Portando, a complexidade do laço é $O(n)$. De modo geral, a complexidade do metódo é de ordem $O(n)$

4.3 propaga

O pior caso e o melhor nesse metodo se assemelham, pois ele irá percorrer necessariamente todos os n elementos que estão na arvore repetindo a função recursivamente n-vezes, tendo como ordem de complexidade $O(n)$, sendo n os número de elementos da árvore.

4.4 arv_OrdenaVec

Esse metódo é o típico que necessáriamente vai fazer o pior caso, pois a ideia é ir até o final e trocar os valores por um determinado valor. Na primeira parte há um laço que tem uma condição de parada passível de ser alcançada se todos os n elementos da arvore forem alterado, aninhado há outro laço que vair percorrendo os nós que estão à frente, tendo necessidade de executar $\approx (n-1)$ vezes e quando chega ao fim chama o Metódo propaga explicado acima. Nesse caso a complexidade de maneira geral do metodo é da ordem de $O(n^2)$ sendo n número de elementos pelo fato de um laço necessita executar n vezes e outro aninhado executado n vezes ele mesmo que necssita de (n-1) para ser concluído.Assim ficammos com a equação $n(n-1) = n^2 - n = O(n^2)$.

Logo da forma que foi implementado o algoritomo de Ordenação é da ordem de $O(n^2)$.