UNIVERSIDADE FEDERAL DE OURO PRETO
UFOP
CURSO BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DE ESTRUTURA DE DADOS 1

TRABALHO PRÁTICO 02

Antonio Alex Gomes Rodrigues - 19.1.4173
Arthur Mayan Faria de Almeida - 18.2.4149

.

Ouro Preto - MG
19 de Abril de 2021

IMPLEMENTAÇÃO

PROBLEMA 1 - QUICKSORT

O Quicksort é o algoritmo mais eficiente na ordenação por comparação e usa o método de dividir para conquistar. Não é um algoritmo estável, mas existe uma versão dele que consegue obter estabilidade com um custo extra

Foi implementado um algoritmo QuickSort utilizando duas formas de seleção do pivô. A primeira forma foi por meio da seleção dos valores aleatoriamente e a outra pela mediana de três valores.

Sabemos que o QuickSort executa no seu pior caso em tempo O(n log n), já o InsertionSort, O(n^2). O InsertionSort é mais eficiente que o QuickSort para um pequeno valor de n. Diante disso, buscando eficiência no código, foi implementado um algoritmo que executa o QuickSort enquanto n for maior que 10. Quando n for menor ou igual a 10, será executado o InsertionSort.

No nosso main, pedimos ao usuário para escolher o tamanho do vetor que deseja ser ordenado, se ele quer um vetor aleatório ou definir os valores. Após isso, temos duas opções para o pivô do quicksort ser definido, sendo ele de forma aleatória ou uma mediana de três valores.

Chamamos nossa função para realizar o QuickSort, passando por parâmetro o vetor, o primeiro índice do vetor(esquerda) e o último índice (direita).

```
void quickSort_aleatorio(int *v, int esquerda, int direita)
```

```
void quickSort_mediana(int *v, int esquerda, int direita)
```

* Temos duas funções do QuickSort, uma para o pivô de forma aleatória e outra para a mediana, pois dentro dessa função específicas, é feita a partição de acordo com a forma definida do pivô.

Quando o valor da direita menos a esquerda for menor que 20, executamos o insertionSort. Caso for maior, chamaremos uma função para realizar a partição do vetor de acordo com a escolha do pivô.

```
int particao_aleatorio(int *v, int esquerda, int direita)
```

```
int particao_mediana(int *v, int esquerda, int direita)
```

Dentro dela, nós definimos o pivô. Todos os valores menores que o pivô, irão para a partição da esquerda e todos os valores maiores, para o da direita.

```
for (j = esquerda; j <= direita - 1; j++)
{
        if (v[j] <= pivo)
        {
            i = i + 1;
            troca(v, i, j);
        }
}
```

E o pivô será colocado no meio das duas partições.

```
   troca(v, i + 1, direita);
```

Nessa função será retornado a posição do pivô. Após isso, faremos comparações com os valores dentro de cada partição. Caso o pivô menos a esquerda for menor que a direita menos o pivô, iremos chamar recursivamente a função QuickSort, mas dessa vez passando como parâmetro o vetor, o valor que o vetor começa, e ao invés do último valor, passaremos o pivô menos 1. Após isso, definimos a esquerda como pivô +1 para não serem comparados os mesmos valores, o que poderia ocasionar um loop infinito. Caso o pivô menos a esquerda for maior que direita menos o pivô, chamaremos novamente recursivamente a função QuickSort e passamos por parâmetro o vetor, no lugar do valor inicial do vetor, ele agora irá valer o valor do pivô mais 1, e passaremos o valor final do vetor. Essa função irá ordenar todos os valores do vetor pelo método de ordenação QuickSort por meio de chamadas recursivas.

```
int pivo = particao_aleatorio(v, esquerda, direita);

            if(pivo - esquerda < direita - pivo)
            {
                quickSort_aleatorio(v, esquerda, pivo - 1);
                esquerda = pivo + 1;
            }
            else{
                quickSort_aleatorio(v, pivo + 1, direita);
                direita = pivo - 1;
            }
```

ESCOLHA DO PIVÔ

Temos várias formas de escolha do pivô do QuickSort, podendo ser o primeiro termo, o último, etc. No nosso trabalho prático foi definido de forma aleatória e foi escolhida a mediana de três valores.

O algoritmo do Quicksort tem complexidade quadrática dependendo de como o pivô for definido. Fazendo a escolha do pivô em forma aleatória, diminuímos as chances do número escolhido ser o maior, caindo no pior caso. Para eliminar totalmente as chances de cair no pior caso, podemos fazer uma mediana de três elementos e definir o termo que está entre o maior e o menor como o pivô. Sendo assim, no melhor caso e no caso médio, requer O(n log n) comparações em média.

PIVÔ DEFINIDO DE FORMA ALEATÓRIA

Na função "int pivoAleatorio(int *v, int esquerda, int direita)", passamos por parâmetro o vetor, o índice do começo do vetor, e o do final. Dentro da função iremos sortear um índice aleatoriamente entre o início e o fim do vetor, e o valor que está nesse índice será definido como nosso pivô.

```
sortear = (double) rand () /
((double) RAND_MAX + 1);
   valor = sortear * (direita -
esquerda + 1);
   int aleatorio = esquerda + valor;
```

Após isso trocamos o valor que está na direita(final) para o valor do pivô aleatório

```
   troca(v, aleatorio, direita);
```

PIVÔ DEFINIDO PELA MEDIANA DE TRÊS VALORES

Na função "int pivo_mediana(int *v, int esquerda, int direita)" passamos por parâmetro o vetor, o índice do começo do vetor, e o do final. Dentro da função iremos fazer uma mediana de três valores.

Iremos definir "a" como o primeiro valor do vetor, "c" como o valor do meio (o tamanho do vetor divido por dois) e "c" como o último valor.

```
int meio = (esquerda + direita) /
2;
   int a = v[esquerda];
   int b = v[meio];
   int c = v[direita];
```

Fazemos comparações para verificar qual dos valores é o maior, qual o menor, e qual valor está entre eles. O valor médio será definido como o pivô. Após isso, chamamos a função troca para trocarmos o elemento que está na direita pela mediana do índice.

```
   troca(v, indice_do_meio,
direita);
```

INSERTIONSORT

O InsertionSort costuma ser uma escolha mais adequada para vetores com no máximo 10 elementos ou para vetores quase ordenados. A complexidade de tempo no seu pior caso é $O(n^2)$ e no melhor caso $O(n)$. É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear. O algoritmo de ordenação por inserção é estável. Na função do InsertionSort, comparamos um número com o seu sucessor, caso ele for menor, ele irá mudar de posição com o número comparado e realizará uma nova comparação com todos os números anteriores e realocando caso for necessário.

```
for(int i = esquerda + 1; i <= direita; i++)
   {
       aux = v[i];
       j = i;

       while(j > esquerda && v[j-1] > aux)
       {
           v[j] = v[j-1];
           j--;
       }

       v[j] = aux;
   }
```

COMPARACOES DOS METODOS DE ORDENACAO

PROBLEMA 2 - ÁRVORES BINÁRIAS

A principal característica de uma árvore binária é que cada um dos elementos pode ter no máximo dois filhos. O tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores, ou seja, se ela está balanceada ou não. O pior caso da árvore binária é O(n) tanto para pesquisa quanto para inserção e remoção. Para obter o pior caso basta que as chaves sejam inseridas em ordem crescente ou decrescente. E no melhor caso temos O (1).

IMPRESSÕES GERAIS

Foi bastante interessante a implementação desde algoritmo, aprendemos diversas formas de conversão em c, principalmente um char, array de char já tinha por definição funções da biblioteca padrões (string, stdio...), e principalmente descobrimos por fim a implementar uma arvore de formar correta em C.

ANÁLISE

O algoritmo foi dividido em 3 partes principais sendo eles:

1 – Ordem de precedência dos operadores
2 – Inserção na Árvore Binária
3 – Cálculo da expressão

PARTE 1

A ordem de precedência foi tratada no próprio main, em um vetor de char auxiliar, as movimentações e trocas foram feitas neste vetor em especifico. O intuito aqui é: Os operadores *, /, + e – são tratados nesta ordem, isto é, quando encontrado um deles o numero que antecede esse operador e o seu sucessor irão para uma string auxiliar (aqui a parte mais complicada foi tratar os índices desses vetores de char). O mais importante desta parte encontra-se abaixo.

```
for(int i = 0 ; i < tamanho_string; i++){
        if(string[i] == '*' || string[i] == '/')
            alteraPrecedenciaOperadores(&string, &string_aux, &indice_string_aux,
 string[i]);
    }
    for(int i = 0 ; i < tamanho_string; i++){
        if(string[i] == '+' || string[i] == '-')
            alteraPrecedenciaOperadores(&string, &string_aux, &indice_string_aux
, string[i]);
```

```
}
```

PARTE 2

Esta parte foca na inserção dos registros(elementos) que estão na string(expressão) na Arvore Binaria, para tal cada registro é atribuído uma chave, gerada pela função geraChaveItem, a parte mais importante nesta etapa é a função criaRegistros, exemplificado abaixo:

```
for(int i = 0; i < tamanho_string; i++){
    Item item_x;
    item_x.string = string[i];
    geraChaveItem(&item_x);
    arvoreInsere(ponteiro_raiz, item_x);
}
```

PARTE 3

Nesta parte o foco é centrado para o cálculo dos registros que pegamos da Árvore Binária, como “tecnicamente” estão ordenados, isto é, os operadores estão organizados por ordem de precedência jogamos todos os registros que estão na arvore em uma Lista, na TAD Lista que o calculo da expressão é feita, nesta TAD a função mais importante é a calculadora, no qual é responsável por fazer uma analise na lista e então calcular o resultado, exemplificado abaixo:

```
//Primeira verificao, descobrir se na lista esta posicao e um operador
    //Caso verdadeiro pegar as informacoes, fazer a operacao e guardar
o resultado em uma lista auxiliar
    if(operador_ou_numero >= 1 && operador_ou_numero <= 4){
        (...)
        //Segundo verificacao, descobrir se for um numero
        //caso verdadeiro ha duas posibilidades, quando for
mutiplicacao ou divisao a operacao, pegar os dois
        //numeros e "inutilizar" os registros pegos da lista atribuino
'A' a eles
        //Inutilizacao necessaria devido a ser um laco, porem so nos
casos de multiplicacao e divisao
        if(operador_ou_numero == 0){
            (...)
            if(operador_ou_numero == 0){
                (....)
            }
            else{

                //Nao ha mais 2 numeros e um operador para ser pego na
lista, portanto, pegar o operador
                //e o numero seguinte e fazer a operacao com os valores
que estao na lista auxiliar
                if(!lst_vazia(&lista_aux)){
                 (...)
                }
            }
        }
```

```
    }
    ponteiro_aux = ponteiro_aux->proximo;
}
```

CONCLUSÃO

A partir desse trabalho podemos analisar melhor como implementar os métodos de ordenação e Árvores Binárias. Podemos entender melhor também sobre a complexidade de cada método, analisar e comparar quais algoritmos são mais eficientes. Usamos a linguagem C para a implementação do código por ser uma linguagem que temos mais facilidade.