Universidade Federal de Ouro Preto

Cursos: Ciência da Computação

Disciplina: BCC202

Professor Pedro Henrique Lopes Silva

Aluno: Nathann Zini dos Reis

Matrícula: 19.2.4007

1. Implementação

Documentação referente à questão 1, sobre ordenação de vetores com QuickSort e InsertionSort:

Para esse código, eu criei um struct para armazenar todos os dados relativos aos vetores, que, por sua vez, serão lidos a partir de arquivos informados na execução do programa no terminal.

O arquivo contém a quantidade de vetores, o tamanho dos vetores e os valores dos vetores, nessa ordem (todos os vetores do mesmo do mesmo arquivo possuem o mesmo tamanho).

Em seguida, passo um vetor e chamo a função para ordernar com o QuickSort. Nessa função, a condição de parada dela é o valor k pré definido com 20. Quando um sub vetor proveniente do QuickSort tem um tamanho menor ou igual a 20, esse sub vetor é então ordenado pelo InsertionSort.

```
void quickSort(int* vetor, int l, int r){

     if( l < r){
         //para o pivô aleatorio, eu pego qualquer valor que esteja na
ultima posicao; Para pivô não aleatorio, eu pego o mediano
           //do primeiro, do ultimo e do valor do meio do vetor não
ordenado;



         //-----FAVOR COMENTAR UMA DAS LINHAS 49 ou 50 E DESCOMENTAR A
OUTRA para que a outra rode;
        //int q = partitionAleatorio(vetor, l, r);
        int q = partitionMediana(vetor, l, r);
        if(l + q - 1 <= k)
            insertionSort(vetor, l, q);
        else
            quickSort(vetor, l, q - 1);

        if( r - q - 1 <= k)
            insertionSort(vetor, q, r + 1);
        else
            quickSort(vetor, q + 1, r);
    }
}
```

Há dois modos de funcionamento do QuickSort que, no final, obtém-se o mesmo resultado. Eles se diferem na escolha do pivô de comparação que por um lado é aleatório (pegando sempre o valor que estiver na última posição) e por outro lado é escolhido como o valor mediano entre 3 valores (o primeiro, o do meio e o último).

O mesmo processo é feito para todos os outros vetores. Após isso, os vetores ordenados são impressos na tela do terminal.

Documentação referente à questão 2, sobre a manipulação de expressões matemáticas por meio de árvores binárias:

Para esse código, eu criei um struct que representa um nó da árvore binária e ele contém a informação dele (o item) e dois outros nós, um à esquerda e outro à direita. Eu trabalhei com a expressão salva em um vetor de caracteres.

Então eu chamo a função para montar a árvore binária e passo como parâmetro o nó raíz da árvore que eu instanciei anteriormente e o vetor com a expressão, juntamente com o range que será avaliado o vetor.

A minha lógica funciona em achar um operador (respeitando as regras de precedência, encontro sempre a última operação a ser feita com a expressão analisada no range passado como parâmetro do vetor).

```
    int acharRaiz(char* expressao, int l, int r){
        int temp = l;
        for(int i = l; i < r; i++){
            if (expressao[i] == '+' || expressao[i] == '-'){
                temp = i;
                /* code */
            }
        }
        if(temp == l){
            for(int i = l; i < r; i++){
            if (expressao[i] == '*' || expressao[i] == '/'){
                temp = i;
                /* code */
            }
        }
        }
        return temp;
    }
```

Esse operador então fica sendo o valor/item daquele nó e a parte à esquerda dele é passada para o nó à esquerda e a parte à direita é passada ao nó à direita. E o mesmo é feito, recursivamente, para as outras metades dos nós filhos, sempre fazendo primeiro com os filhos à esquerda e depois os às direitas.

```
    //recebe o vetor com a expressao e o inicio e o final do range a
ser avaliado e monta, a partir dai a arvore;
```

```
      //sempre vai dividindo a expressão ao meio até restar apenas um
valor que vai sendo as folhas e os operadores sempre os nós internos;
      void montarArvore(char* expressao, int l, int r, TNo** pRaiz){
        if(l < r){
          int root = acharRaiz(expressao, l, r);
          //insere o nó interno
          TArvore_Insere(pRaiz, expressao[root]);
          //passa a metade esquerda da expressao
          montarArvore(expressao, l, root - 1 , pRaiz);
          //passa a metade direita da expressao
          montarArvore(expressao, root +1, r , pRaiz);
        }else if( l == r){
          //insere a folha
          int temp = TArvore_Insere(pRaiz, expressao[l]);
        }

      }
```

Após isso é impresso no terminal a árvore em ordem pré-fixa, central e pós-fixa, respectivamente.

Para a segunda parte dessa mesma questão, em que é recebido uma árvore binária e, à partir dela, é resolvida a expressão matemática, eu aproveitei o código da primeira parte, que cria uma árvore binária com a expressão matemática, e resolvo a expressão.

Para isso, eu utilizei de pilha e lista (a lista foi por preferência de uso, não sendo necessária). Em uma pilha eu armazeno a expressão matemática da árvore e para isso eu busquei os valores nela de forma Ordem Pré-fixa. Fiz isso pois, para o funcionamento da minha calculadora, eu preciso de uma ordem específica para a informação dos operandos e operadores, que, em pilha, é a ordem pré-fixa da árvore binária.

```
      void arvoreEncaminhamentoPreOrdem(TNo* raiz, Pilha* pilha)
      {
        if (raiz != NULL)
        {
          Pilha_Push(pilha, raiz->item);

          arvoreEncaminhamentoPreOrdem(raiz->pEsq, pilha);
```

```
            arvoreEncaminhamentoPreOrdem(raiz->pDir, pilha);
        }
    }
```

Então criei uma outra pilha que será utilizada para o cálculo da expressão. Nela, será sempre salvo primeiro dois valores e depois feito a operação entre eles cabível e serão retirados da pilha e salvo o resultado da expressão, que será usado como um operando de uma nova operação e assim sucessivamente até encontrar o "=" que eu coloco no início da pilha manualmente.

```
    do{
                //Pega o primeiro dado da pilha com a expressão e
transforma para char para a comparação
            Pilha_Pop(pilhaExpressao, &x);
            entrada = (char) x;
               //Verifica se é um número. Se for, salva o numero na
pilha do calculo. Caso contrário, se for um operador,
            //verifica se há digitos suficientes para o cálculo
            if(entrada >= '0' && entrada <= '9'){
                operando = (int) entrada - 48;
                Pilha_Push(pilhaCalculo, operando);
            }else if(entrada == '-' ||
                     entrada == '+' ||
                     entrada == '*' ||
                     entrada == '/' &&
                     Pilha_Tamanho(pilhaCalculo) >= 2){
                        Pilha_Push(pilhaCalculo,  calculadora(entrada,
pilhaCalculo));
            }
        }while (entrada != '=');
```

Na calculadora será sempre retirado os dois últimos valores da pilha, feito a operação com eles e retorna o resultado dessa operação.

2. Impressões Gerais

O processo de implementação deste trabalho em específico foi trabalhoso. Passei 2 dias seguidos para conseguir completar a codificação deste trabalho. Foi muito bom obter novos conhecimentos a respeito da árvore binária e como ela pode auxiliar em outros códigos que já trabalhamos anteriormente e com certeza em futuros códigos. Particularmente, não havia ainda feito ou sabia como fazer para resolver uma expressão matemática digitada pelo usuário em sua forma mais simples respeitando as ordens de precedência. Agora, com a árvore binária, isso foi possível.

3. Análise

Segue a análise do código sobre ordenação do quickSort juntamente com o insertionSort:

Array de tamanho 100, valor de k igual a 20				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem inicial do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Crescente	888	700	801	578
	699	532	668	447
Aleatório	1488	1451	1488	1451
	1691	1666	1691	1666
Decrescente	1873	1081	1731	987
	1667	943	1695	985

Array de tamanho 300, valor de k igual a 20				
	Pivô aleatório		Pivô com mediana de três	
Ordem inicial do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Crescente	44660	44939	44660	44939
	44660	44939	44660	44939
Aleatório	2675	1918	2663	1783
	2565	1692	2550	1613
Decrescente	44068	22334	44068	22334
	42278	21431	42278	21431

Array de tamanho 400, valor de k igual a 20				
	Pivô aleatório		Pivô com mediana de três	
Ordem inicial do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Crescente	79610	79989	79610	79989
	79610	79989	79610	79989

Aleatório	4028	2870	4044	2738
	3706	2190	3956	2282
Decrescente	76697	38870	76397	38566
	72312	36511	72312	36511

Array de tamanho 100, valor de k igual a 10				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem inicial do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Crescente	796	522	801	578
	661	408	668	447
Aleatório	1474	1407	1488	1451
	1791	1794	1691	1666
Decrescente	1874	1084	1731	987
	1633	884	1695	985

Array de tamanho 100, valor de k igual a 30				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem inicial do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Crescente	930	759	1007	846
	781	631	875	716
Aleatório	1414	1377	1414	1377
	1598	1575	1598	1575
Decrescente	2039	180	1790	1109
	1776	1095	1804	1137

Array de tamanho 100, valor de k igual a 40				
	Pivô aleatório		Pivô com mediana de três	
Ordem inicial do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Crescente	930	759	1117	984
	956	829	940	816
Aleatório	1286	1251	1286	1251
	1438	1410	1438	1410
Decrescente	2054	1400	1927	1316
	1813	1162	1841	1204

Com a análise feita acima, verifiquei, através do número de movimentações e comparações realizadas pelo programa, que, a fim de melhorar a execução do quickSort, o InsertionSort deveria ser executado quando os subproblemas tiverem tamanho k = 10 ou k = 20. De preferência k = 10.

4. Conclusão

O processo de implementação deste trabalho em específico foi um tanto quanto complicado para mim. Eu, apesar de estar apresentando ele sozinho, tive auxílio de colegas de turma para a realização do mesmo. Particularmente não tive dificuldade em entender a teoria do assunto do trabalho, porém, aplicar esse entendimento na codificação foi complicado. Principalmente a parte de entender como é a criação e o desmembramento da árvore binária. Foi a parte em que encontrei a maior dificuldade.

Porém, essa dificuldade manteve ainda mais meu interesse em aprender para poder superá-la.