Universidade Federal
de Ouro Preto

ENYA LUÍSA GOMES DOS SANTOS
19.4.4201

RELATÓRIO DO SEGUNDO TRABALHO PRÁTICO DE ESTRUTURA DE DADOS I

Relatório apresentado por exigência da disciplina BCC202 - Estrutura de Dados I, da Universidade Federal de Ouro Preto.
Professor: Pedro Henrique Lopes Silva

1. Implementação

1.1. QuickSort combinado com InsertionSort

A entrada do algoritmo pode ser recebida através de *m* arrays de inteiros do mesmo tamanho *n*, ou através de um arquivo de texto. Esse arquivo deve conter na primeira linha , obrigatoriamente, a quantidade de arrays e, separado por um espaço, o tamanho desses arrays - pois em cada arquivo só é possível ter arrays do mesmo tamanho - nas próximas linhas os arrays, além disso, o arquivo deve estar, obrigatoriamente, em uma pasta, na raiz, chamada “arquivos”. Como mostra um exemplo abaixo. Já a saída será mostrada via terminal. O objetivo é a organização de vetores em ordem crescente.

```
2 10
4 2 5 8 9 3 2 4 8 5
0 2 4 8 9 6 3 5 7 8
```

Através da implementação do *quicksort* e *insetionsort*, para combiná-los, o insetion é chamado toda vez que o tamanho do array após ser dividido pelo quicksort é um valor menor que k, onde k é definido no algoritmo. Como mostra o código abaixo, o mesmo está presente no arquivo hybridsort.c.

```
/* Item é um valor do tipo inteiro */
void hybridsort(Item *v, int l, int r, int op)
{
  if (l < r)
  { /*Se o tamanho do vetor for maior que K o quicksort é
usado,
    caso contrario, o insertionsort é usado */
    if ((r - l) > K)
    {
      Item q = partition(v, l, r, op);
      hybridsort(v, l, q - 1, op);
      hybridsort(v, q + 1, r, op);
    }
    else
      insertionsort(v, l, r + 1);
  }
}
```

Para visualizar e analisar os resultados, há uma contagem do número de movimentações e comparações, tanto no *quicksort* quanto no *insetionsort.* Para isso, há uma estrutura global que armazena essa contagem, através de ponteiros/referência, esses valores são “passados” para o *main* para serem impressos e analisados. A seguir a função que é chamada no *main* e está presente também no arquivo hybridsort.c.

```
void hybridsortInicia(Item *v, int n, Contador *c, int op)
{
  count = c;
  zeraContador();
  hybridsort(v, 0, n, op);
}
```

Além disso, conforme pedido, há duas formas para a escolha do pivô, sendo uma aleatória e a outra através da mediana de três.

O modo aleatório sempre escolhe o elemento da última posição do array. Já o outro modo escolhe o pivô através da mediana de três, onde três elementos do array são escolhidos, sendo eles o primeiro, o do meio e o último, logo em seguida é calculado a mediana entre eles, depois de obter esse resultado, essa posição é trocada com a última posição do array. A função abaixo também está presente no arquivo hybridsort.c.

```
void pivoMedianaDeTres(Item *v, int l, int r)
{
  int metade = (l + r) / 2;
  Item ini = v[l];
  Item meio = v[metade];
  Item fim = v[r];
  int medianaIndice = -1;

  if (ini < meio)
  {
    if (meio < fim)
    {
      medianaIndice = metade;
    }
    else
    {
      if (ini < fim)
      {
        medianaIndice = r;
      }
```

```
            else
            {
              medianaIndice = l;
            }
          }
        }
        else
        {
          if (fim < meio)
          {
            medianaIndice = metade;
          }
          else
          {
            if (fim < ini)
            {
              medianaIndice = r;
            }
            else
            {
              medianaIndice = l;
            }
          }
        }
        swap(v, medianaIndice, r);
      }
```

E é dessa forma que o *quicksort* é combinado com o *insetionsort* a fim de obter uma melhor performance na ordenação.

Também, com a finalidade de facilitar a geração de arrays para serem testados, foi implementado um gerador de arrays, em um arquivo chamado geradorArray.c, onde esse algoritmos recebe como entrada:

1. A opção da ordem do array, sendo 1, 2 ou 3 que representa arrays de ordem aleatória, crescente ou decrescente, necessariamente nessa ordem.
2. A quantidade m de arrays a serem gerados.
3. O tamanho n que esses arrays irão possuir.

Por fim, a saída será impressa via terminal.

1.2. Árvore binária de expressão

A entrada do algoritmo pode ser através de *n* expressões matemáticas em forma de string, ou de um arquivo de texto, necessariamente nomeado como *expressoes.txt,* além disso, o arquivo deve estar, obrigatoriamente, em uma pasta, na raiz, chamada “arquivos”, cada expressão será armazenado nas devidas posições de um vetor de strings. Esse arquivo deve conter, obrigatoriamente, em sua primeira linha a quantidade de expressões e nas linhas subsequentes, a expressão a ser calculada. Como mostra abaixo.

```
3
3 + 5 * 8
(3 + 5) * 8
5*5+(6-9)
```

Com o intuito de evitar problemas, cada string de entrada é tratada, de forma que todos os espaços sejam removidos.

Nos próximos passos do algoritmo será construído a árvore binária, cuja essa árvore tem a seguinte estrutura:

```
/* TChave é do tipo int, TElemento é do tipo char */
typedef struct
{
  TChave chave;
  TElemento elemento;
} TItem;

struct arvoreNo
{
  TItem item;
  ArvoreNo *esq, *dir;
};
```

A **chave** recebe a posição do elemento na string da expressão, e o **elemento** recebe o carácter naquela posição, desse modo, é possível construir a árvore ordenando os nós pelo valor desta chave para serem adicionados à esquerda ou à direita do seu nó pai.

Seguindo a lógica do algoritmo, o passo seguinte percorre a string à procura do operador principal, respeitando a ordem de precedência dos operadores e dos parênteses, a partir dele a string será dividida em duas partes.

Considerando uma expressão matemática representada por uma string s[*0..r*] com o operador principal na posição *q* , esse será dividida em duas partes a[*0..q-1*] e b[q+1..r].

Além disso, um nó raiz da árvore será criado recebendo o operador principal como elemento e sua posição na string como valor da chave.

O código abaixo está presente no arquivo arvore_expressao.c.

```
int buscaOperadorPrincipal(char *expressao, int ini, int fim)
{
  int index = -1;
  int temp = -1;
  int parentesesOpen = 0;

  for (int i = fim - 1; i >= ini; i--)
  {
    if (expressao[i] == ')')
      parentesesOpen += 1;
    else if (expressao[i] == '(')
      parentesesOpen -= 1;
    else if (ehOperador(expressao[i]) && !parentesesOpen)
    {
      if ((expressao[i] == '*' || expressao[i] == '/'))
      {
        if (temp == -1)
          temp = i;
      }
      else
      {
        index = i;
        break;
      }
    }
  }

  if (index == -1)
  {
    if (temp != -1)
      index = temp;
    else
       index = buscaOperadorPrincipal(expressao, ini + 1, fim -
1); /* caso a expressão esteja toda dentro de parênteses */
  }

  return index;
```

```
}
```

Exemplificação da execução do algoritmo de construção da árvore

Considerando a expressão “3 + (5 * 8) - 2 + 1” como entrada, obtém-se o seguinte arranjo de caracteres (string) após o tratamento:

0	1	2	3	4	5	6	7	8	9	10
3	**+**	**(**	**5**	*****	**8**	**)**	**-**	**2**	**+**	**1**

O operador principal será o [+] que está na posição nove do arranjo, pois é a última operação a ser resolvida, através da obtenção do operador principal será criado a raiz da árvore com esse (+). Logo será dividida a expressão em duas partes, a partir desse índice do operador principal, a fim de construir as subárvores, tanto a esquerda como a direita.

A seguir a demonstração das duas partes divididas.

0	2	3	4	5	6	7	8
3	**(**	**5**	*****	**8**	**)**	**-**	**2**

10
1

Funcionamento do algoritmo que constrói as subárvores

A partir da “divisão” da expressão, em duas partes, será criada duas subárvores, uma para a esquerda e a outra para a direita do nó raiz.

O código abaixo está presente no arquivo arvore_expressao.c.

```
ArvoreNo *geraArvoreExpressao(char *expressao)
{
  int n = strlen(expressao);
  trataString(expressao, n);
  n = strlen(expressao);

  int pOperador = buscaOperadorPrincipal(expressao, 0, n);
  if (pOperador == -1)
  {
    printf("Erro ao encontrar o operador principal na
expressão '%s'.\n", expressao);
    exit(1);
  }
  ArvoreNo *raiz = arvoreCriaNovoNo(expressao[pOperador],
pOperador);

  int atual = 0;
```

```
  ArvoreNo *noEsq = geraSubarvoreExpressao(expressao, 0,
pOperador - 1, &atual);
  raiz = arvoreAdicionaNo(raiz, noEsq);

  ArvoreNo *noDir = geraSubarvoreExpressao(expressao,
pOperador + 1, n, &atual);
  raiz = arvoreAdicionaNo(raiz, noDir);

  return raiz;
}
```

Esse algoritmo utiliza como principal ferramenta uma pilha que tem o item como sendo um ponteiro de nós de árvores binárias. A pilha utilizada para essa função é uma dinâmica.

A string é percorrida, e de acordo com um conjunto de condições, que ajudam a manter a precedência dos operadores e parênteses mantendo a expressão coerente. As principais regras as seguintes, analisando o carácter, caso seja:

- **Um operador:**
    - Se o tamanho da pilha for maior ou igual a dois e o operador for um operador de adição “+” ou subtração “-”: nessas circunstâncias ocorre o desempilhamento da pilha duas vezes, na primeira de um nó filho, e na segunda de um nó pai.
    - Se o operador for um operador de multiplicação “*” ou de divisão “/”: uma variável nomeada de *preferenciaOperador,* na qual se inicia com 0, será incrementada.

    Por fim, independente das condições acima, um novo nó pai é criado com esse operador, a pilha é desempilhada um vez, esse sendo o nó filho, logo, o nó pai recebe o nó filho e é empilhado.

    Exemplo, temos a expressão 3+5*8-3, a posição atual da string a ser analisada é a posição seis e tem como caractere o operador de subtração.

As primeiras condições citadas anteriormente são satisfeitas		Pilha após o final da execução da primeira condição	Pilha ao final da condição de ser um operador
Pilha		Pilha	Pilha
TOPO *(5()8())	+(3())	TOPO +(3()*(5()8()))	TOPO -(+(3()*(5()8())))

Nessa representação de árvore, o que está fora dos parênteses são os nós, suas partes internas são os filhos. *

Tabela 1 - Exemplificação da execução com um operador.

- **Um operando:** um nó filho será criado com chave igual ao índice e elemento igual ao próprio carácter. Logo temos as seguintes condições
    - Se a variável *preferenciaOperador,* tiver valor maior ou igual a um: a pilha é desempilhada uma vez e esse chamado de nó pai, logo esse nó recebe o nó filho com o operando atual criado anteriormente e é empilhado. E por fim o valor da variável *preferenciaOperador* é decrementado.
    - Caso a primeira condição não seja verdadeira: o nó filho, criado anteriormente, simplesmente é empilhado.

Exemplo, temos a expressão 3+5*8-3, considerando a posição atual da string a ser analisada, é a posição quatro e tem como caractere o operando oito.

<table>
<tr><td colspan="2">As primeiras condições citadas anteriormente são satisfeitas, pois o último operador é de multiplicação, então a variável preferenciaOperador tem valor um.</td><td colspan="2">Pilha após o final da execução da condição. Um novo nó é criado com o 8, e é chamado de filho, após isso a pilha é desempilhada, e é retornado o nó [*(5())], e é nele onde o filho 8() é adicionado, mantendo assim a precedência dos operadores.</td></tr>
<tr><td colspan="2">Pilha</td><td colspan="2">Pilha</td></tr>
<tr><td>TOPO
*(5())</td><td>+(3())</td><td>TOPO
*(5()8())</td><td>+(3())</td></tr>
</table>

Nessa representação de árvore, o que está fora dos parênteses são os nós, suas partes internas são os filhos. *

Tabela 2 - Exemplificação da execução com um operando.

- **Um parênteses:**
    - Se for um parênteses aberto “(“: há uma chamada recursiva à função atual, e ao final, com o retorno de um nó, será empilhado e o valor de de *i* do loop é atualizado para a posição do fechamento do parênteses mais um.
    - Caso seja um parênteses fechado “)”: o loop é parado, já que aqui, considerando uma expressão bem formatada, a chamada recursiva criada ao abrir o parênteses deve parar para continuar o resto da expressão.

Ao fim do loop que percorre a string, se o tamanho da pilha for maior ou igual a dois, será desempilhado de 2 a 2 elementos até que esse tenha tamanho um. O primeiro se torna um filho e o segundo um nó pai que receberá esse filho, logo o nó pai é empilhado novamente.

Ao final, com apenas um elemento na pilha, a subárvore, finalmente, construída será desempilhada e retornada pela função.

O código abaixo está presente no arquivo arvore_expressao.c.

```
      ArvoreNo *geraSubarvoreExpressao(char *exp, int ini, int fim,
int *atual)
      {
        PilhaArvore *pSubarvores = pilhaArvoreInicia();
        int preferenciaOperador = 0;

        for (int i = ini; i <= fim; i++)
        {
           *atual = i; /* caso o fechamento do parênteses esteja na
outra extremidade da expressão */
          if (exp[i] == '(')
          {
              ArvoreNo *noPai = geraSubarvoreExpressao(exp, i + 1,
fim, atual);
            pilhaArvorePush(pSubarvores, noPai);
            i = *atual;
          }
          else if (exp[i] == ')')
          {
            break;
          }
          else if (ehOperador(exp[i]))
          {
             if (pilhaArvoreTamanho(pSubarvores) >= 2 && (exp[i] ==
'+' || exp[i] == '-'))
            {
              ArvoreNo *noFilho = pilhaArvorePop(pSubarvores);
              ArvoreNo *noPai = pilhaArvorePop(pSubarvores);
              noPai = arvoreAdicionaNo(noPai, noFilho);
              pilhaArvorePush(pSubarvores, noPai);
            }
            else if (exp[i] == '*' || exp[i] == '/')
              preferenciaOperador += 1;

            ArvoreNo *noPai = arvoreCriaNovoNo(exp[i], i);
            ArvoreNo *noFilho = pilhaArvorePop(pSubarvores);
            noPai = arvoreAdicionaNo(noPai, noFilho);
```

```
      pilhaArvorePush(pSubarvores, noPai);
    }
    else if (ehOperando(exp[i]))
    {
      ArvoreNo *noFilho = arvoreCriaNovoNo(exp[i], i);
      if (preferenciaOperador >= 1)
      {
        ArvoreNo *noPai = pilhaArvorePop(pSubarvores);
        noPai = arvoreAdicionaNo(noPai, noFilho);
        pilhaArvorePush(pSubarvores, noPai);
        preferenciaOperador -= 1;
      }
      else
        pilhaArvorePush(pSubarvores, noFilho);
    }
  }

  int i = pilhaArvoreTamanho(pSubarvores);
  while (i >= 2)
  {
    ArvoreNo *noFilho = pilhaArvorePop(pSubarvores);
    ArvoreNo *noPai = pilhaArvorePop(pSubarvores);
    noPai = arvoreAdicionaNo(noPai, noFilho);
    pilhaArvorePush(pSubarvores, noPai);
    i = pilhaArvoreTamanho(pSubarvores);
  }

  ArvoreNo *aux = pilhaArvorePop(pSubarvores);

  pilhaArvoreUnstack(&pSubarvores);

  return aux;
}
```

Cálculo da árvore binária de expressão

Inicialmente a árvore é caminhada com o método de Pós-Ordem, assim, é possível obter uma expressão na notação pós-fixada.

Para resolver a expressão, é utilizado uma pilha que tem como item um float. Como na prática 05, realizada na disciplina BCC202 - Estrutura de dados I, já havia implementado uma pilha capaz de realizar cálculo de expressões pós-fixada, essa também foi utilizada para realizar essa tarefa de cálculo da árvore.

Em resumo, esse algoritmo empilha os operandos, e quando recebe um operados, a pilha e desempilhada duas vezes, e esses valores são armazenados em variáveis diferentes, em seguida a operação é realizada entre os valores de acordo com o operador e o resultado é empilhado.

O código abaixo está presente no arquivo pilha.c.

```
float calculadoraPosOrdem(char op, Pilha *pilha)
{
  float valor1 = pilhaPop(pilha);
  float valor2 = pilhaPop(pilha);

  switch (op)
  {
  case '+':
    return valor1 + valor2;
    break;
  case '-':
    return valor1 - valor2;
    break;
  case '*':
    return valor1 * valor2;
    break;
  case '/':
    return valor1 / valor2;
    break;
  }

  return 0;
}

void constroiPilhaPosFixada(Pilha *pilha, char elemento)
{
  if (elemento == '+' ||
      elemento == '-' ||
      elemento == '*' ||
      elemento == '/')
    pilhaPush(pilha, calculadoraPosOrdem(pilha, elemento));
```

```
  else
  {
    pilhaPush(pilha, ((float)elemento - 48));
  }
}
```

As próximas linhas código, abaixo, estão presentes no arquivo árvore.c

```
void arvorePosOrdem(ArvoreNo *raiz, Pilha *pilha)
{
  if (raiz == NULL)
    return;
  arvorePosOrdem(raiz->dir, pilha);
  arvorePosOrdem(raiz->esq, pilha);

  constroiPilhaPosFixada(pilha, raiz->item.elemento);
}
```

Ao final, é possível obter o saída do resultado da expressão de três formas: o resultado pode ser impresso no terminal em conjunto com a representação gráfica da árvore, o resultado pode ser armazenado em um arquivo de texto chamado de *resultados.txt* também em conjunto com a representação gráfica da árvore, ou, por fim, de ambas as formas. A opção é escolhida no início da execução.

Com a saída obtida, o programa é encerrado e todas as memórias alocadas são liberadas.

Limitações:

1. Não há verificação da validade da entrada.
2. A expressão pode ter no máximo 120 caracteres, é possível mudar para maior ou menor essa quantidade no código, mas ainda será uma quantidade estática.

2. Impressões gerais

2.1. QuickSort combinado com InsertionSort

A implantação foi de certa forma simples, devido ao fato de já ter implementado o algoritmo de quicksort e insertionsort anteriormente em outros exercícios da disciplina.

Com essa implementação foi possível entender melhor tanto o funcionamento do quicksort quanto do insertionsort. Foi possível, também, ver como um método de ordenação

não será perfeito em todos os casos e como combinados, suprindo a desvantagem de um com a vantagem de outro, pode-se ter uma performance maior.

2.2. Árvore binária de expressão

Particularmente, optei por resolver esse problema antes da primeira, pois já havia estudado sobre a estrutura de árvore binária a algumas semanas, através de vídeos no youtube do canal UNIVESP, por esse motivo o algoritmo foi implementado com um pouco mais de tempo, entretanto, sem dúvidas, a questão foi mais complicada de resolver em relação à primeira.

A implementação foi feita através de ciclos, após finalizar uma base do algoritmo testes eram realizados, e de acordo com os resultados ajustes eram feitos assim com soluções melhores eram encontradas.

O principal aprendizado é saber planejar bem, antes de colocar a mão no código, sem essa perspectiva obtive bastante trabalho e retrabalho que poderia ser facilmente evitado. Da mesma forma foi possível aprender como as estruturas podem trabalhar em conjunto, e também entender cada vez mais cada uma delas, suas limitações, vantagens e pensar nas possíveis possibilidade de uso, e também aprender a utilizar a ferramentas que facilitaram a encontrar erro, como o debug com extensões do vscode e o valgrind para verificar possíveis *memory leaks,* ferramenta na qual não tinha conhecimento nem existência.

Por mais complicado que tenha sido, foi um desafio bastante interessante de ser feito e trouxe novos conhecimentos.

3. Análise

3.1. QuickSort combinado com InsertionSort

A seguir é analisado diferentes valores para k em vetores de tamanho igual a 100, e diferentes ordens, aleatórios, crescentes e decrescentes. Os valores de k escolhidos foram 10, 20, 30 e 40. Esses valores foram escolhidos a partir do tamanho do vetor 100, tendo em vista que k deveria ser menor de 50 para o quicksort ser utilizado e não somente o insertionsort. O objetivo é a organização de vetores em ordem crescente.

Array de tamanho **100**, valor de **k** igual a **10**				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Aleatório	469	793	365	591
	344	654	315	623
Crescente	1352	1471	298	586
	1698	1771	254	514
Decrescente	1008	1855	368	610
	834	1629	358	631

Tabela 3 - Resultados obtidos com vetores de tamanho 100 e a variável k igual a 10.

Array de tamanho **100**, valor de **k** igual a **20**				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Aleatório	612	881	537	719
	470	708	431	698
Crescente	1374	1482	340	581
	1594	1677	318	557
Decrescente	1019	1859	566	794
	872	1656	548	793

Tabela 4 - Resultados obtidos com vetores de tamanho 100 e a variável k igual a 20.

Array de tamanho **100**, valor de **k** igual a **30**				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Aleatório	707	945	637	813
	568	791	613	834
Crescente	1296	1409	340	581
	1499	1587	426	617
Decrescente	1272	2057	717	908

	1076	1795	776	992

Tabela 5 - Resultados obtidos com vetores de tamanho 100 e a variável k igual a 30.

Array de tamanho **100**, valor de **k** igual a **40**				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Aleatório	859	1086	947	1097
	841	1037	841	1037
Crescente	1166	1282	319	551
	1329	1427	426	617
Decrescente	1323	2048	717	908
	1085	1085	943	1146

Tabela 6 - Resultados obtidos com vetores de tamanho 100 e a variável k igual a 40.

Analisando os valores das tabelas acima, é possível afirmar que, quando temos k igual a 20 ou 10 os valores de movimentações e comparações são mais satisfatórios, principalmente quando o pivô é escolhido com a mediana de três. Também, como o insertionsort é adequado para vetores com no máximo 10 elementos, mas para esse caso o 20 apresentou melhor eficiência em relação ao 10.

Agora com um k escolhido, sendo ele o 20, abaixo há uma sequência de tabelas com os resultados dos números de movimentações e comparações totais, testados com vetores aleatórios, crescentes e decrescentes. Além disso, também é possível comparar os resultados com o pivô escolhido de forma aleatória e o pivô escolhido a partir da mediana de três.

Os vetores testados possuem tamanhos 100, 300 e 500. Dois aleatórios, dois crescentes e dois decrescentes.

Array de tamanho **100**, valor de **k** igual a **20**				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Aleatório	612	881	537	719
	470	708	431	698
Crescente	1374	1482	340	581
	1594	1677	318	557

Decrescente	1019	1859	566	794
	872	1656	548	793

Tabela 7 - Resultados obtidos com vetores de tamanho 100

Array de tamanho **300**, valor de **k** igual a **20**				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Aleatório	1735	2781	1602	2536
	1569	2674	1423	2536
Crescente	44919	44661	598	1443
	44919	44661	600	1443
Decrescente	22063	43582	2788	3934
	21807	43096	2829	4002

Tabela 8 - Resultados obtidos com vetores de tamanho 300

Array de tamanho **500**, valor de **k** igual a **20**				
	Pivô aleatório		**Pivô com mediana de três**	
Ordem do vetor	**Movimentações**	**Comparações**	**Movimentações**	**Comparações**
Aleatório	3463	5342	3116	4673
	3157	5289	2957	4907
Crescente	125019	124560	1243	2880
	125019	124560	1244	2880
Decrescente	60058	119294	4551	6835
	59004	117204	4624	6981

Tabela 9 - Resultados obtidos com vetores de tamanho 500

Por meio dos resultados mostrados acima, pode-se concluir que, de modo geral, como o pivô escolhido por meio da mediana de três os resultados são muito melhores comparados com o pivô escolhido aleatoriamente. Também pode-se perceber que o pivô aleatório com vetores de ordem crescente tem piores resultados do que o decrescente, já com o pivô escolhido pela mediana de três o contrário ocorre, e o crescente apresenta melhores resultados também comparado ao aleatório.

Com auxílio do valgrind foi verificado que não há *memory leaks* na execução do programa, durante os testes.

3.2. Árvore binária de expressão

O resultado obtido com o trabalho foi satisfatório, visto que, testando uma entrada com 55 expressões em arquivo, todas apresentaram o resultado correto. Além das 55 outras eventuais expressões testadas também apresentaram resultados corretos.

Abaixo há uma tabela com algumas das expressões testadas é o resultado obtido, e logo abaixo o resultado de algumas expressões com a árvore “desenhada”.

3 + 5 * 8 = 43.00	((2+5)-2)+((5*9)/4) = 16.25
(3+5)*8 = 64.00	1 + 2 * 8 - (5 + 6) = 6.00
((3+5)-2)+((5*9)/4) = 17.25	7 * (2 + 2) / 7 = 4.00
(((3+5)-2)+((5*9)/4)) = 17.25	3*(6+3)-8/(2+2) = 25.00
(3+5) - 2 = 6.00	(9 + 8) + (6 - 9) = 14.00
1+(2*(4/2-1)) = 3.00	5 + (2 - 2) - 8 = -3.00
5 * (6 - 4 / 2) = 20.00	1 - (((5+5)-3) + 2) = -8.00
5 - (6 - 4 / 2) = 1.00	4 + 3 * 2 - 1 * 3 = 7.00
5+(6-4/2) = 9.00	9 * (6 + 8) - 9 / (5 + 4) = 125.00
(5+2)-3*(2-1) = 4.00	(1 + 9) + (9 / (6 - 3)) = 13.00
7*7*5/2*9-6+3*9-1 = 1122.50	5 * 5 = 25.00

Tabela 10 - Expressões matemáticas testadas e seu resultado

3+5*8 = 43.00 <+> `--<3> `--<*> `--<5> `--<8> (3+5)*8 = 64.00 <*> `--<+> |  `--<3> | `--<5>	(5-7-9)+5-2-9*(4-3+2) = -35.00 <-> `--<-> |  `--<+> | | `--<-> |  |  |  `--<-> | | | | `--<5> |  |  |  |  `--<7> | | | `--<9> |  |  `--<5> | `--<2> `--<*>

```
 `--<8>

((3+5)-2)+((5*9)/4) = 17.25
<+>
 `--<->
 |    `--<+>
 |    |    `--<3>
 |    |    `--<5>
 |    `--<2>
 `--</>
      `--<*>
      |    `--<5>
      |    `--<9>
      `--<4>
```

```
      `--<9>
      `--<+>
           `--<->
           |    `--<4>
           |    `--<3>
           `--<2>

9-6*8*6/2 = -135.00
<->
 `--<9>
 `--</>
      `--<*>
      |    `--<*>
      |    |    `--<6>
      |    |    `--<8>
      |    `--<6>
      `--<2>
```

Tabela 11 - Expressões matemáticas testadas, seu resultado e representação com árvore binária

Com auxílio do valgrind foi verificado que não há *memory leaks* na execução do programa, durante os testes.

4. Conclusão

A maior dificuldade encontrada durante a implementação foi o tempo. Mas comparando as duas questões, a segunda foi mais complicada de se desenvolver e demandou mais tempo para obter resultados satisfatórios.

Acredito que havia melhores soluções e mais simples para resolver o problema dois em relação a minha solução implementada, mas essa funciona de forma bem satisfatória.

De modo geral, foi bem interessante desenvolver as soluções dos problemas propostos no trabalho prático, foi possível ver melhor como a estrutura de árvore funciona e como os métodos de ordenação podem ser combinados para obter uma maior performance. Também, o melhor entendimento de como funcionam as memórias durante a execução do algoritmo foi obtido.

5. Referências

GAUL, Randy. Printing Pretty Ascii Trees. 16 de junho de 2015. Disponível em: <https://laptrinhx.com/printing-pretty-ascii-trees-259794796/> Acesso em 10 de abril de 2021.

SANTIADO, David. Árvores: Estrutura de dados. 03 de setembro de 2019. Disponível em: <https://algol.dev/arvores-estrutura-de-dados/> Acesso em 14 de abril de 2021.

UNIVESP. Estrutura de Dados - Aula 15 - Árvores - Conceitos básicos. Disponível em: <https://youtu.be/eiMMtyRBYCE> Acesso em 04 de abril de 2021.

UNIVESP. Estrutura de Dados - Aula 16 - Árvores binárias de pesquisa - Parte 1. Disponível em: <https://youtu.be/7IKXYhqipK8> Acesso em 04 de abril de 2021.

UNIVESP. Estrutura de Dados - Aula 17 - Árvores binárias de pesquisa - Parte 2. Disponível em: <https://youtu.be/O4AqgoO42pc> Acesso em 05 de abril de 2021.

UNIVESP. Estrutura de Dados - Aula 18 - Árvores binárias de pesquisa - Parte 3. Disponível em: <https://youtu.be/3koM42vL6js> Acesso em 05 de abril de 2021.

JAIN, Arpit. Hybrid Quick Sort in C++. Disponível em: <https://www.codespeedy.com/hybrid-quick-sort-in-c/> Acesso em 15 de abril de 2021.