INF1010 - Relatório Tarefa 2

João Gabriel Peixoto Drumond Lopes - 2210168

1 Introdução

O trabalho tem como objetivo a implementação de dois tipos de árvore binária, de busca e genérica (sem ordenação intrínseca). De agora em diante, será descrita a solução utilizada para resolver esse problema, conforme as instruções presentes no enunciado. De forma resumida, foram implementados dois tipos de árvore binária e um tipo de fila. Ademais, foi criado um módulo para realizar funções de execução do programa, não inerentes aos Tipos Abstratos de Dados criados.

2 Estrutura do programa

O programa como um todo possui cinco módulos. O principal, *main*, que contém a função de entrada da execução; o *execution*, que tem como foco a retirada de certas construções de código da main, para possibilitar maior clareza da solução; os *bintree* e *binstree*, que implementam respectivamente uma árvore binária genérica e uma árvore binária de busca; e o *queue*, que tem como objetivo o auxílio ao funcionamento do módulo *bintree*, e implementa uma fila.

A divisão entre os módulos de árvore binária genérica e árvore binária de busca foi motivada pela necessidade de tratamento diferente de ambos. As funções de inserção, por exemplo, possuem duas abordagens muito díspares. Além disso, uma hipotética função de busca ou de deleção requisitaria implementações diferentes para cada um dos dois tipos.

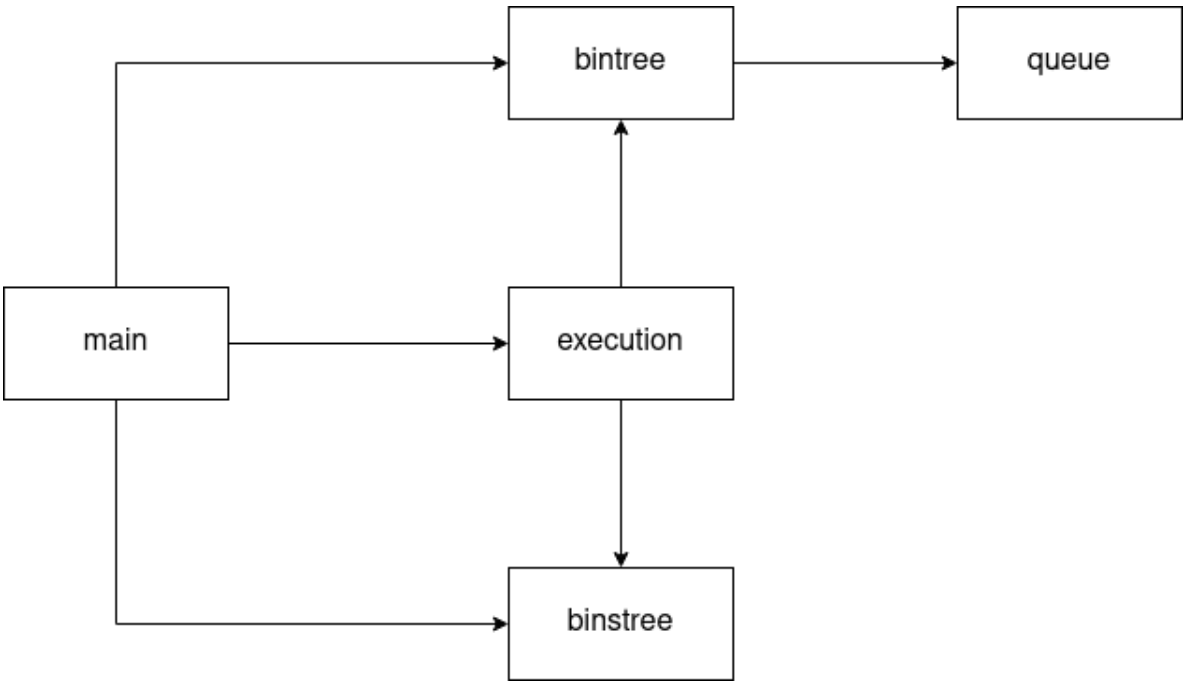

Figura 1: Diagrama da estrutura dos módulos do programa.

2.1 Módulo *main*

Nesse módulo, o ponto de entrada do programa, somente são realizadas as atividades requisitadas no enunciado, além da liberação de memória alocada ao longo da execução. Importa os módulos *bintreee*, *binstree* e *execution*.

2.2 Módulo *binstree*

Esse módulo implementa um TAD, mais especificamente, uma árvore binária de busca. O tipo estruturado tem os seguintes campos: ponteiro para o pai, ponteiro para a subárvore da direita, ponteiro para subárvore da esquerda e chave, um inteiro.

As funções implementadas foram: *cria_bst*(), *cria_no_bst*(), *insere_bst*(), *imprime_bst*(), *verifica_bst*(), *altura_bst*(), *libera_bst*() e *troca_bst*().

- *cria_bst*(): retorna NULL para representar a criação da árvore binária de busca, vazia.
- *cria_no_bst*(): cria um nó de árvore binária de busca.

  A função recebe dois parâmetros, um ponteiro para nó de árvore de busca (que pode ser NULL) e um inteiro que representa a chave do nó. São então atribuídos aos ponteiros esquerdo e direito do nó NULL, o ponteiro pai recebe o parâmetro de número 1, key recebe o parâmetro de número 2. Ao final, é retornado esse novo nó alocado, com verificação de alocação correta de memória previamente.
- *insere_bst*(): insere um nó na árvore, mantendo a ordenação de uma árvore binária de busca. Recebe a árvore e a chave a ser inserida, e retorna a árvore atualizada.

  A função inicialmente verifica se a árvore está vazia. Caso esteja, insere um novo nó, alocando memória e preenchendo os campos. Caso não esteja vazia, realiza uma comparação com a chave do nó atual para determinar se deve realizar uma chamada recursiva para inserção nas subárvores da direita ou da esquerda. Ao final da execução, retorna a árvore atualizada, passando diretamente para o fim da função, já que as comparações serão falsas, visto que a chave do nó atual é igual à chave a ser inserida (que já foi).
- *imprime_bst*(): imprime as chaves dos nós da árvore recebida em pré-ordem.

  Como é uma implementação em pré-ordem, ele trata o nó atual, e, então, percorre as subárvores da esquerda e da direita, nessa ordem. Quando chega a NULL, pula para o fim da execução. A impressão dos parênteses antes dos campos tem por objetivo representar nós vazios da árvore com (), para melhor representação da estrutura.
- *verifica_bst*(): verifica se a árvore é binária de busca ou não.

  A função executa até chegar ao fim da árvore, NULL, ou até achar uma instância em que é quebrada a lógica da árvore binária de busca (chave maior na esquerda ou chave menor na direita). Realizada chamadas recursivas até o fim do programa. Retorna 1 caso a árvore sejá ABB, 0 caso contrário.
- *altura_bst*(): retorna a altura da árvore binária recebida.

  A execução da função segue a lógica de incrementar o contador de altura, chamando a função para as subárvores da direita e da esquerda. Ao final, verifica qual é a maior, e a retorna, somando 1. Quando chega ao fim da árvore, retorna 0 (caso base da recursão).
- *libera_bst*(): libera a memória alocada da árvore binária de busca.

  Funciona com base na pós-ordem, percorrendo as subárvores esquerda e direita e, por fim, liberando o nó atual. Ao final da execução, retorna NULL, para indicar a liberação bem sucedida.
- *troca_bst*(): troca as subárvores da direita com as da esquerda.

  Primeiramente cria uma variável temporária para armazenar a subárvore da esquerda. Então, troca a esquerda com a direita e depois, a direita com a esquerda. Realizada chamadas recursivas para percorrer a árvore inteira. No fim, retorna a árvore atualizada. No caso da árvore binária de busca, ela deixa de o ser.

2.3 Módulo *queue*

O módulo queue é também um TAD, implementando uma fila. Possui um tipo estruturado que representa um nó da fila, com os dados sendo um ponteiro para nó de árvore binária, além de um ponteiro para o próximo nó da fila. Também implementa um tipo estruturado de fila propriamente dita, guardando em seus campos ponteiros para o início e para o fim da estrutura de dados. Seu objetivo principal é auxiliar na função de inserção da árvore binária genérica por nível.

Implementa as seguintes funções: $cria\_fila()$, $vazia\_fila()$, $insere\_fila()$, $retira\_fila()$ e $libera\_fila()$.

- $cria\_fila()$: aloca memória para uma fila, inicializando os campos de início e fim como NULL.
- $vazia\_fila()$: retorna 1 para fila vazia, 0 caso contrário.
- $insere\_fila()$: função de inserção em fila.

  Caso a fila esteja vazia, insere o nó na primeira posição e determina que tanto o início quanto o fim são iguais a ele. Caso contrário, insere o nó no final da fila, e o torna o novo final.
- $retira\_fila()$: função de remoção de fila.

  Remove o primeiro item da fila. Torna o próximo, então, o primeiro. Caso a fila então fique vazia, põe o final também como vazio. Retorna o campo de dados do nó removido.
- $libera\_fila()$: função de liberação de memória de fila.

  Percorre uma fila até o final, com o auxílio de dois cursores. Libera toda a memória alocada dos nós e, por fim, da fila em si. Retorna NULL para indicar sucesso.

2.4 Módulo *bintree*

Esse módulo implementa uma árvore binária genérica. A diferença para a árvore binária de busca no tipo estruturado é que não possui um ponteiro para o nó ancestral direto. Importa somente o módulo *queue*.

Implementa as funções: $cria\_vazia\_bt()$, $cria\_bt()$, $insere\_bt\_q()$, $verifica\_bt()$, $imprime\_bt()$, $altura\_bt()$, $libera\_bt()$ e $troca\_bt()$.

- $cria\_vazia\_bt()$: retorna NULL para representar a criação da árvore binária vazia.
- $cria\_bt()$: aloca memória para um nó da árvore, inicializando os campos com os parâmetros recebidos, chave, subárvore esquerda e subárvore direita.
- $insere\_bt\_q()$: insere um nó na árvore binária, em ordenação por nível. O q representa o uso de uma queue para a implementação, importada do módulo *queue*.

  A implementação segue: caso a árvore seja NULL, é inserido um novo nó e a árvore atualizada é retornada. Caso contrário, uma fila é criada, e o nó atual é nela inserido. Enquanto a fila não encontra-se vazia, o primeiro elemento é retirado. Caso um dos campos à direita ou à esquerda sejam vazios, a árvore insere lá o nó, com preferência para a esquerda. A fila é liberada após a inserção, e a árvore atualizada retornada. Caso tanto o nó direito quanto o esquerdo não sejam NULL, eles são inseridos na fila, e a execução continua até achar um nó vazio. Ao final, é liberada a memória alocada da fila e a nova árvore é retornada.
- $verifica\_bt()$: verifica se a árvore é binária de busca ou não.

  A função executa até chegar ao fim da árvore, NULL, ou até achar uma instância em que é quebrada a lógica da árvore binária de busca (chave maior na esquerda ou chave menor na direita). Realizada chamadas recursivas até o fim do programa. Retorna 1 caso a árvore sejá ABB, 0 caso contrário.
- $imprime\_bt()$: imprime as chaves dos nós da árvore recebida em pré-ordem.

  Como é uma implementação em pré-ordem, ele trata o nó atual, e, então, percorre as subárvores da esquerda e da direita, nessa ordem. Quando chega a NULL, pula para o fim da execução. A impressão dos parênteses antes dos campos tem por objetivo representar nós vazios da árvore com (), para melhor representação da estrutura.

- *altura_bt*(): retorna a altura da árvore binária recebida.

  A execução da função segue a lógica de incrementar o contador de altura, chamando a função para as subárvores da direita e da esquerda. Ao final, verifica qual é a maior, e a retorna, somando 1. Quando chega ao fim da árvore, retorna 0 (caso base da recursão).

- *libera_bt*(): libera a memória alocada da árvore binária de busca.

  Funciona com base na pós-ordem, percorrendo as subárvores esquerda e direita e, por fim, liberando o nó atual. Ao final da execução, retorna NULL, para indicar a liberação bem sucedida.

- *troca_bt*(): troca as subárvores da direita com as da esquerda.

  Primeiramente cria uma variável temporária para armazenar a subárvore da esquerda. Então, troca a esquerda com a direita e depois, a direita com a esquerda. Realizada chamadas recursivas para percorrer a árvore inteira. No fim, retorna a árvore atualizada.

2.5 Módulo *execution*

Esse módulo foi criado para auxiliar na clareza da execução do programa principal. Implementa três funções simples: *rand_nums*(), *insert_level*() e *insert_bst*().

- *rand_nums*(): retorna um array de n inteiros de valor máximo max, com os inteiros gerados aleatoriamente pelas funções *rand*() e *srand*() (determina a seed) da *stdlib.h*.

- *insert_level*(): cria uma nova árvore binária genérica e insere os valores presentes no array nums nela. Recebe também o número de elementos de nums.

- *insert_bst*(): função semelhante à anterior. Contudo, realiza a inserção em uma árvore binária de busca. Elementos repetidos são descartados.

3 Solução

A solução foi realizada, conforme os itens presentes no enunciado. Será ilustrada a realização de cada tarefa utilizando capturas que mostrem a parte do código que realiza cada funcionalidade requisitada.

a) Gere 10 números inteiros (chaves), aleatórios, pertencentes ao intervalo [1,20] e exiba as chaves na ordem gerada.

```
int main(void) {
    int* rand_numbers = rand_nums(20, 10);
```

Figura 2: Implementação de a).

b) Elabore e execute um algoritmo de inserção das chaves geradas em a) por nível, em uma árvore binária.

```
    BinTree* bt_tree = insert_level(rand_numbers, 10);
```

Figura 3: Implementação de b).

c) Elabore e execute um algoritmo de inserção das chaves geradas em a) em uma ABB

```
BinSTree* bst_tree = insert_bst(rand_numbers, 10);
```

Figura 4: Implementação de c).

d) Exiba as árvores criadas nos itens b) e c) usando o percurso em pré-ordem.

e) Implemente uma função para verificar se uma árvore cujo endereço da raiz é passado como parâmetro é ABB. Use as árvores criadas nos itens b) e c) como exemplos.

f) Implemente uma função para indicar a altura de uma árvore binária cujo endereço da raiz é passado como parâmetro. Use as árvores criadas nos itens b) e c) como exemplos.

g) Troque as sub-árvores esquerda e direita de todos os nós de uma árvore exibindo a árvore resultante. Use as árvores criadas nos itens b) e c) como exemplos.

```
imprime_bt(bt_tree);
printf("\n");
imprime_bst(bst_tree);
printf("\n");
```

Figura 5: Implementação de d).

```
printf("binaria de busca?\narvore 1 (bt): %d\narvore 2 (bst): %d\n",
                verifica_bt(bt_tree),
                verifica_bst(bst_tree));
```

Figura 6: Implementação de e).

```
printf("altura?\narvore 1 (bt): %d\narvore 2 (bst): %d\n",
                altura_bt(bt_tree),
                altura_bst(bst_tree));
```

Figura 7: Implementação de f).

```
bt_tree = troca_bt(bt_tree);
bst_tree = troca_bst(bst_tree);

imprime_bt(bt_tree);
printf("\n");
imprime_bst(bst_tree);
printf("\n");
```

Figura 8: Implementação de g).

3.1 Descrição da solução

Conforme observado no código acima, primeiramente foi utilizada a função *rand_nums*() para gerar um array de 10 números de forma aleatória. Então, duas árvores binárias, uma genérica e uma de busca, foram criadas, utlizando as funções implementadas no módulo *execution* de inserção por nível (que chama funções de *bintree*) e inserção por ordenação de ABB (que chama funções de *binstree*). A impressão de ambas as árvores é realizada, utilizando as respectivas funções. São impressos "contrabarra n" para auxiliar na visualização da saída do programa.

As verificações então, de se a árvore é ou não binária de busca são realizadas, com a primeira e a segunda árvores, e o resultado é impresso em seguida. Logo adiante, o mesmo ocorre com a função de altura. A troca ocorre logo abaixo, e a impressão da árvore resultante também. Depois disso, é chamada a função de verificação de ser binária de busca ou não. Por fim, embora não requisitado explicitamente, foram chamadas funções para liberar a memória alocada pelas árvores binárias e pelo array de números. O programa também foi analisado com o Valgrind, para certificar que não existem memory leaks. A linha de comando utilizada foi: "valgrind –tool=memcheck –leak-check=full –show-leak-kinds=all –track-origins=yes ./main".

Obs: a impressão da árvore binária de busca possui um campo extra, o de ponteiro para o pai. A ordenação então fica: ponteiro-pai — ponteiro-próprio — ponteiro-esquerdo — ponteiro-direito

```
==4553== Memcheck, a memory error detector
==4553== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4553== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4553== Command: ./main
==4553==
3 9 7 13 9 14 15 11 18 2
IMPRESSAO INICIAL ARVORE 1:
( 3  0x4a994f0 0x4a995f0 0x4a996f0 ( 9  0x4a995f0 0x4a99890 0x4a99a30 ( 13  0x4a99890 0x4a9a190 0x4a9a470 ( 11  0x4a9a190 (nil) (nil) ()())( 18  0x4a9a470 (nil)
 (nil) ()()))( 9  0x4a99a30 0x4a9a7f0 (nil) ( 2  0x4a9a7f0 (nil) (nil) ()())()))( 7  0x4a996f0 0x4a99c70 0x4a99eb0 ( 14  0x4a99c70 (nil) (nil) ()()))( 15  0x4a99
eb0 (nil) (nil) ()()))
IMPRESSAO INICIAL ARVORE 2:
( 3  (nil) 0x4a9a850 0x4a9ab50 0x4a9a8b0 ( 2  0x4a9a850 0x4a9ab50 (nil) (nil) ()())( 9  0x4a9a850 0x4a9a8b0 0x4a9a910 0x4a9a970 ( 7  0x4a9a8b0 0x4a9a910 (nil) (
nil) ()())( 13  0x4a9a8b0 0x4a9a970 0x4a9aa90 0x4a9a9d0 ( 11  0x4a9a970 0x4a9aa90 (nil) (nil) ()())( 14  0x4a9a970 0x4a9a9d0 (nil) 0x4a9aa30 ()( 15  0x4a9a9d0 0
x4a9aa30 (nil) 0x4a9aaf0 ()( 18  0x4a9aa30 0x4a9aaf0 (nil) (nil) ()()))))))
binaria de busca?
arvore 1 (bt): 0
arvore 2 (bst): 1
altura?
arvore 1 (bt): 4
arvore 2 (bst): 6
IMPRESSAO APOS TROCA ARVORE 1:
( 3  0x4a994f0 0x4a996f0 0x4a995f0 ( 7  0x4a996f0 0x4a99eb0 0x4a99c70 ( 15  0x4a99eb0 (nil) (nil) ()())( 14  0x4a99c70 (nil) (nil) ()()))( 9  0x4a995f0 0x4a99a3
0 0x4a99890 ( 9  0x4a99a30 (nil) 0x4a9a7f0 ()( 2  0x4a9a7f0 (nil) (nil) ()()))( 13  0x4a99890 0x4a9a470 0x4a9a190 ( 18  0x4a9a470 (nil) (nil) ()())( 11  0x4a9a1
90 (nil) (nil) ()()))))
IMPRESSAO APOS TROCA ARVORE 2:
( 3  (nil) 0x4a9a850 0x4a9a8b0 0x4a9ab50 ( 9  0x4a9a850 0x4a9a8b0 0x4a9a970 0x4a9a910 ( 13  0x4a9a8b0 0x4a9a970 0x4a9a9d0 0x4a9aa90 ( 14  0x4a9a970 0x4a9a9d0 0x
4a9aa30 (nil) ( 15  0x4a9a9d0 0x4a9aa30 0x4a9aaf0 (nil) ( 18  0x4a9aa30 0x4a9aaf0 (nil) (nil) ()())())())( 11  0x4a9a970 0x4a9aa90 (nil) (nil) ()()))( 7  0x4a9a
8b0 0x4a9a910 (nil) (nil) ()()))( 2  0x4a9a850 0x4a9ab50 (nil) (nil) ()()))
binaria de busca?
arvore 1 (bt): 0
arvore 2 (bst): 0
==4553==
==4553== HEAP SUMMARY:
==4553==     in use at exit: 0 bytes in 0 blocks
==4553==   total heap usage: 71 allocs, 71 frees, 2,392 bytes allocated
==4553==
==4553== All heap blocks were freed -- no leaks are possible
==4553==
==4553== For lists of detected and suppressed errors, rerun with: -s
==4553== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 9: Saída do programa e do Valgrind.

4 Observações e Conclusões

O programa parece atender ao que foi requisitado. Não foram encontrados memory leaks e, também, todos os testes parecem ter resultado em sucesso. É possível que exista algum caso que possa causar problemas, embora improvável (a princípio). A função *main* poderia ter ainda mais funções delegadas para o módulo *execution*, como uma função única de impressão que recebesse ponteiro para as duas árvores e chamasse as funções dos módulos respectivos no módulo *execution* em si. Contudo, não alteraria consideravelmente a execução do programa, apenas um detalhe menor. O mesmo se aplica a outras funcionalidades presentes no módulo *main*.

Quanto ao relatório, algumas imagens ficaram fora da ordenação desejada devido à formatação do LaTeX. Contudo, todas as imagens possuem captions que indicam o seu conteúdo e a que parte da solução pertence.