Pontifícia Universidade Católica do Rio de Janeiro
INF1010 - Estruturas de Dados Avançadas

Lucas Demarco Cambraia Lemos - 2110013
Jayme Augusto Avelino de Paiva - 2210289

Tarefa 2
Árvores Binárias e Árvores Binárias de Busca (ABB)

Rio de Janeiro
2023.1

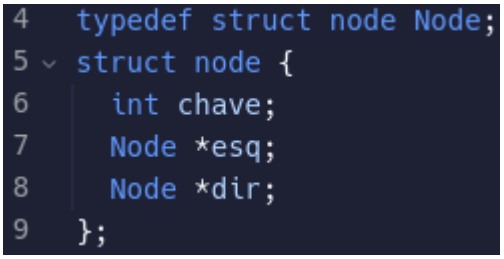
1. Objetivo do projeto

O projeto tem o objetivo de automatizar a criação de Árvores Binárias e Árvores Binárias de Busca, por meio de funções que realizam as operações para ambos os tipos de árvores. Para fazer com que o código funcione para casos diferentes, o enunciado pediu para gerar números aleatórios entre 1 e 20, exibindo as árvores formadas no percurso de pré-ordem e verificando a altura de cada árvore criada.

2. Estrutura do programa

O programa possui um arquivo contendo a *main*, que inclui os testes requisitados no enunciado da tarefa, além de possuir dois módulos *header*, que agrupa funções referentes a inicialização e operações com as árvores binárias geradas.

Todo o programa utiliza de uma estrutura pré-definida, que caracteriza cada nó da árvore, sendo possível visualizar isso pela imagem abaixo:



Inicialmente, o arquivo *t2-inicializacao.h* reúne todas as funções necessárias para inicializar e montar as duas árvores pedidas no enunciado. O primeiro passo é definir quais serão as chaves usadas, processo realizado pela função *geraChaves*, que modifica um vetor de inteiros passado como parâmetro, adicionando todos os valores dos nós das árvores. Essa função utiliza a *verificaChaves* como auxiliar para descartar todos os número que já foram gerados. A função *inicializa* apenas inicia a raiz como *NULL*, enquanto ambas as funções de inserir adicionam os nós na árvore, tendo seu espaço alocado pela *mkNode.*

Após as árvores estarem devidamente prontas, quando desejamos realizar qualquer operação com elas, buscamos o arquivo *t2-operacoes.h*, que reúne todas as funções com esse intuito. A funcão *exibeArvore* imprime a árvore binária pelo percurso de pré-ordem. A *libera* apenas irá liberar o espaço alocado para cada nó na memória. Ambas as funções *verificaAltura* e *verificaABB* irão fazer validações nas árvores para,

respectivamente, checar a altura da árvore e se é uma árvore binária de busca. A função *inverteArvore* irá inverter as sub-árvores da esquerda e direita da árvore toda.

3. Solução

Para este tópico do relatório, será mostrado como foi feita a resolução de cada item proposto pelo enunciado da tarefa.

3.1) Gerando inteiros aleatórios

O primeiro objetivo do trabalho é gerar os valores que ocuparão os nós das árvores que serão formadas. Para isso, a função geraChaves receberá um vetor de inteiros e irá alterá-lo, adicionando todos os números que forem gerados. Foi utilizada uma função *srand* da biblioteca *time.h* e *stdlib.h,* que fornece números aleatórios. Na medida que esses inteiros vão sendo gerados, são incluídos no vetor passado como parâmetro.

```
12  void geraChaves(int* valores){
13    srand(time(NULL));
14    int i = 0;
15    int chave;
16    while(i < 10){
17      chave = (rand() % 20) + 1;
18      if (verificaChaves(valores, chave) == 1){
19        printf("%d ", chave);
20        valores[i] = chave;
21        i++;
22      }
23    }
24    return;
25  }
```

Entretanto, um problema encontrado baseia-se no fato de que as árvores não podem ter diferentes chaves com o mesmo valor. Portanto, foi necessário criar a função *verificaChaves*, que recebe o mesmo vetor, além do inteiro gerado pela função *srand*, e percorre o *array* para descobrir se esta chave já foi inserida ou não. No caso positivo, a função retorna 0 e gera-se um novo inteiro para substituí-lo, se for confirmado como um novo valor, retorna 1 e continua gerando as chaves.

```c
int verificaChaves(int* valores, int numero){
  for (int i = 0; i<10; i++){
    if (valores[i] == numero){
      return 0;
    }
  }
  return 1;
}
```

3.2) Inserção nas árvores

Quando todas as chaves são definidas, torna-se possível montar as árvores de acordo com as suas características, ou seja, uma ABB ou uma Árvore Binária por Nível. Antes da inserção, existe a parte de inicialização de ambas as árvores, feita por meio da função *inicializa*, que retorna NULL, caracterizando uma árvore vazia.

```c
Node *inicializa(void) {
  return NULL;
}
```

Após esse passo, é possível começar a inserir as chaves nas árvores, tendo uma função para cada tipo de árvore. Entretanto, ambos os casos necessitam da função *mkNode*, que aloca o espaço na memória para cada nó que será formado, atribuindo os valores de NULL para os ponteiros da esquerda e direita, além de dar ao nó a sua chave específica.

```c
Node* mkNode(int chave){
  Node* node = (Node*)malloc(sizeof(Node));
  if(node == NULL){
    printf("Error: Falta de memória");
    exit(1);
  };
  node->esq = NULL; node->dir = NULL; node->chave = chave;
  return node;
};
```

3.2.1) Inserção em ABB (Árvore Binária de Busca)

Para incluir as chaves do jeito correto na ABB, foi codificado uma função recursiva que recebe a raiz da árvore e a chave a ser incluída. Inicialmente, verifica-se a posição exata para inserir o valor, fazendo recursivas comparações entre e chave e o nó atual, seguindo a regra das ABBs. Quando for identificado o local correto para cada valor, a função *mkNode* é chamada novamente, alocando seu espaço e, em sequência, retornando o endereço da raiz da árvore.

```
106 Node *insereABB(Node *raiz, int chave) {
107   // Verifica se a raiz é NULL, para adicionar o elemento. Caso base da
    recursão.
108   if (raiz == NULL) {
109     Node* auxiliar = mkNode(chave);
110     return auxiliar;
111   }
112   // Compara o valor passado como parâmetro com as chaves já existentes
    árvore. Caso seja maior ou igual, segue para a direita; caso contrário,
    segue para a esquerda
113   else{
114     if(chave >= raiz->chave){
115       raiz->dir = insereABB(raiz->dir, chave);
116     }
117     else if(chave < raiz->chave){
118       raiz->esq = insereABB(raiz->esq, chave);
119     }
120   }
121   return raiz;
122 }
```

3.2.2) Inserção em Árvore Binária por Nível

A função *insereAbNivel* é uma função recursiva para inserir os elementos na árvore binária de forma a sempre preencher um nível completamente, da esquerda para direita, antes de partir para o nível abaixo. A forma que ela faz isso é com chamadas recursivas, primeiramente ela é chamada com os parâmetros altura e nível zerados, então, durante a função ela observa que a altura ainda não foi definida (vale 0) e calcula o valor da altura, nesse ponto vale ressaltar que essa altura não é necessariamente igual a "altura oficial" da árvore (seguindo o conceito de altura de árvores binárias), ele apenas revela o nível que o próximo elemento deve ser inserido, a forma como faz isso é percorrendo a árvore seguindo o ramo na extrema direita até encontrar um NULL, dessa forma esse é o nível que o próximo elemento deve ser inserido, pois esse é o primeiro nível que possui pelo menos um nó NULL, após achar essa altura, a função se chama recursivamente, mas desta vez com a altura de inserção correta (como nas futuras chamadas a altura seguirá inalterada, nenhuma chamada irá recalcular de forma inútil a altura novamente). Nas próximas chamadas a função usa o valor "check" para saber se conseguiu ou não inserir o elemento, de forma em que ela tenta primeiro inserir o elemento nos ramos da esquerda, não conseguindo inserir lá (por só ter conseguido achar um NULL em um nível maior que a altura) ela retorna NULL, durante a recursão esse valor NULL vai ser atribuído ao check, então há uma verificação para saber se o check é NULL, se for a função é chamada recursivamente para os ramos da direita, isso é feito para prevenir que a função insira o valor em mais de um nó e também para que a função insira sempre os valores da esquerda para direita, sem deixar buracos no meio e, além disso, esse processo também previne chamadas recursivas desnecessárias, por fim, quando o valor é atribuído com sucesso, a função retorna o nó que ele foi atribuído, dessa forma o "check" vai assumir esse valor e não será NULL, o que impedirá chamadas recursivas subsequentes e irá retornar recursivamente até chegar a primeira chamada que irá retornar a árvore, agora com o novo elemento inserido.

```c
Node* insereAbNivel(Node* ab, int chave, int altura, int level){
    //check para saber se a chave conseguiu ser inserida nessa rodada da recursão
    int check = 1;
    /*...*/
    if(ab == NULL && !altura){
      return mkNode(chave);
    }else if(altura && level >= altura){
      return NULL;
    };
    //if para checar se a altura ainda não foi calculada, caso não tenha sido ele calcula
    //aqui uma única vez durante a recursão
    if(!altura){
      int alturaAtual = altura;
      Node* aux = ab;
      while(aux != NULL){
        aux = aux->dir;
        alturaAtual++;
      };
      //chamada recursiva com a altura calculada dessa vez
      insereAbNivel(ab,chave,alturaAtual,level);
    }else{
      /*...*/
      if(ab->esq == NULL){
        ab->esq = mkNode(chave);
        return ab;
      }else if(ab->dir == NULL){
        ab->dir = mkNode(chave);
        return ab;
      }else if(insereAbNivel(ab->esq,chave,altura,level+1) == NULL){
        check = (insereAbNivel(ab->dir,chave,altura,level+1) != NULL);
      };
    };
    // se o check continuar valendo 1 quer dizer que foi possível inserir o node, então ele
    // retorna a arvore, caso contrário retorna NULL
    if(check){
      return ab;
    }else{
      return NULL;
    };
};
```

3.3) Verificação de ABB

Para validar que a árvore passada como parâmetro é uma Árvore Binária de Busca, é necessário recorrer a essa função recursiva, que tem como caso base o parâmetro passado ser NULL. Caso contrário, verifica-se que seus ponteiros para a esquerda ou direita não apontam para NULL e, em seguida, confirma-se que a regra de uma ABB foi realmente seguida. Ou seja, a chave deve ser maior que a chave à sua esquerda e menor que a chave à sua direita.

Esta função retorna 1, caso percorra a árvore inteira e chegue até o caso base, e retorna 0 quando encontra uma chave que não esteja adequada com as regras determinadas para ser uma ABB.

```c
int verificaABB (Node* raiz){
  if (raiz == NULL){
    return 1;
  }
  if (raiz->esq != NULL){
    if (raiz->chave <= raiz->esq->chave){
      return 0;
    }
  }
  if (raiz->dir != NULL){
    if (raiz->chave > raiz->dir->chave){
      return 0;
    }
  }
  return verificaABB(raiz->esq) && verificaABB(raiz->dir);
}
```

3.4) Verificação de Altura

Para verificar a altura de cada árvore gerada, a função *verificaAltura* recebe o nó raiz da árvore, tendo como caso base se esse parâmetro seja NULL. Esse caso tem o valor de retorno como -1, já que a altura começa do 0. Se esse valor de retorno fosse 1, a altura encontrada sempre seria 1 a mais do que a altura real.

Em seguida, é possível buscar a maior altura, percorrendo as sub-árvores da esquerda e direita separadamente. Adicionando 1 a cada chamada, as variáveis *alturaEsq* e *alturaDir* têm seus valores totais, sendo possível compará-los e definindo qual a maior altura dentro da árvore, no final de todo o processo recursivo.

```c
int verificaAltura (Node* raiz){
  if (raiz == NULL){
    //retorna -1 pois os niveis devem começar valendo 0
    return -1;
  }
  int alturaEsq = verificaAltura(raiz->esq);
  int alturaDir = verificaAltura(raiz->dir);
  //verifica qual nivel, da direita ou esquerda, é maior e retorna a altura dele + 1 para
  //na recursão o retorno ser incrementado em uma unidade a cada chamada
  if (alturaDir > alturaEsq){
    return (alturaDir + 1);
  }
  else{
    return (alturaEsq + 1);
  }
}
```

3.5) Inversão das sub-árvores

Primeiramente a função checa se a árvore recebida é NULL (vazia) e retorna a msm caso sim, após isso a função inverte os ramos guardando o ramo esquerdo original numa variável auxiliar, fazendo o novo ramo esquerdo ser o ramo direito (de forma recursiva) e o novo ramo direito ser o antigo ramo esquerdo guardado pela variável auxiliar (de forma recursiva), devido às chamadas recursivas no final todos os nós da árvore estarão invertidos.

```
67  Node* inverteArvore(Node* ab){
68      if(ab == NULL)
69          return ab;
70      Node* aux = ab->esq;
71      // invertendo os nodes recursivamente
72      ab->esq = inverteArvore(ab->dir);
73      ab->dir = inverteArvore(aux);
74      return ab;
75  };
```

3.6) Liberação da memória

Lembrando que todos os nós foram alocados dinamicamente na memória, torna-se fundamental liberar esses espaços no final do código. Para isso, foi usada a função recursiva *libera,* que percorre ambas as sub-árvores, liberando todos os nós e, por final, o nó raiz da árvore.

```
24  void libera(Node* ab){
25      libera(ab->esq);
26      libera(ab->dir);
27      free(ab);
28  };
```

3.7) Exibição da árvore

Para exibir a saída do programa, a função *exibeArvore* recebe um ponteiro para o nó raiz da árvore e irá percorrê-la recursivamente, para imprimir cada nó da árvore, até que se chegue ao final de todas as sub-árvores. Nesse exibição, mostra-se o endereço do

nó, o valor da chave, e o endereço dos nós que são apontados para a esquerda e direita na árvore.

O percurso definido para a exibição foi de pré-ordem, ou seja, imprime primeiro o nó, em seguida toda a sub-árvore da esquerda e, por último, toda a sub-árvore da direita.

```
10  void exibeArvore(Node* ab){
11      if(ab == NULL){
12          printf("A árvore está vazia\n");
13          return;
14      };
15      printf("Chave: %d; Node pointer: %p; Left pointer: %p; Right pointer: %p\n", ab->chave, ab, ab->esq, ab->dir);
16      //chamada recursiva para printar os ramos a esquerda primeiro dos da direita
17      if(ab->esq != NULL)
18          exibeArvore(ab->esq);
19      if(ab->dir != NULL)
20          exibeArvore(ab->dir);
21  };
```

4. Saída e testes

De acordo com o que foi solicitado no enunciado do projeto, os testes de todos os itens foram feitos com as árvores geradas nos itens *b* e *c*. A imagem abaixo (primeira da próxima página) demonstra a saída para todos os itens, com base na Árvore Binária de Busca que foi gerada pela função *insereABB.*

No topo da imagem, estão listadas todas as chaves que foram geradas pela função *geraChaves*. Na sequência, a função *exibeArvore* mostra a ABB, de acordo com o percurso de pré-ordem, no modelo estabelecido pelo enunciado. Foi feita a verificação da altura da árvore e caso seja uma ABB ou não. Por último, a árvore foi invertida e exibida também no percurso de pré-ordem.

```
Chaves a serem inseridas: 10 16 13 11 12 14 7 17 8 4
----------------------------------------------------------------------
Percurso em pré-ordem da ABB:

Chave: 10; Node pointer: 0x1d546d0; Left pointer: 0x1d54850; Right pointer: 0x1d54710
Chave: 7; Node pointer: 0x1d54850; Left pointer: 0x1d54910; Right pointer: 0x1d548d0
Chave: 4; Node pointer: 0x1d54910; Left pointer: (nil); Right pointer: (nil)
Chave: 8; Node pointer: 0x1d548d0; Left pointer: (nil); Right pointer: (nil)
Chave: 16; Node pointer: 0x1d54710; Left pointer: 0x1d54750; Right pointer: 0x1d54890
Chave: 13; Node pointer: 0x1d54750; Left pointer: 0x1d54790; Right pointer: 0x1d54810
Chave: 11; Node pointer: 0x1d54790; Left pointer: (nil); Right pointer: 0x1d547d0
Chave: 12; Node pointer: 0x1d547d0; Left pointer: (nil); Right pointer: (nil)
Chave: 14; Node pointer: 0x1d54810; Left pointer: (nil); Right pointer: (nil)
Chave: 17; Node pointer: 0x1d54890; Left pointer: (nil); Right pointer: (nil)

Altura desta árvore: 4

É uma árvore ABB

Percurso em pré-ordem desta árvore invertida:

Chave: 10; Node pointer: 0x1d546d0; Left pointer: 0x1d54710; Right pointer: 0x1d54850
Chave: 16; Node pointer: 0x1d54710; Left pointer: 0x1d54890; Right pointer: 0x1d54750
Chave: 17; Node pointer: 0x1d54890; Left pointer: (nil); Right pointer: (nil)
Chave: 13; Node pointer: 0x1d54750; Left pointer: 0x1d54810; Right pointer: 0x1d54790
Chave: 14; Node pointer: 0x1d54810; Left pointer: (nil); Right pointer: (nil)
Chave: 11; Node pointer: 0x1d54790; Left pointer: 0x1d547d0; Right pointer: (nil)
Chave: 12; Node pointer: 0x1d547d0; Left pointer: (nil); Right pointer: (nil)
Chave: 7; Node pointer: 0x1d54850; Left pointer: 0x1d548d0; Right pointer: 0x1d54910
Chave: 8; Node pointer: 0x1d548d0; Left pointer: (nil); Right pointer: (nil)
Chave: 4; Node pointer: 0x1d54910; Left pointer: (nil); Right pointer: (nil)
```

Para a imagem abaixo, foram verificados os mesmos casos de teste, mas para a árvore gerada por nível, de acordo com o item *b.*

```
Percurso em pré-ordem da Árvore Binária por Nível:

Chave: 10; Node pointer: 0x1d546b0; Left pointer: 0x1d546f0; Right pointer: 0x1d54730
Chave: 16; Node pointer: 0x1d546f0; Left pointer: 0x1d54770; Right pointer: 0x1d547b0
Chave: 11; Node pointer: 0x1d54770; Left pointer: 0x1d54870; Right pointer: 0x1d548b0
Chave: 17; Node pointer: 0x1d54870; Left pointer: (nil); Right pointer: (nil)
Chave: 8; Node pointer: 0x1d548b0; Left pointer: (nil); Right pointer: (nil)
Chave: 12; Node pointer: 0x1d547b0; Left pointer: 0x1d548f0; Right pointer: (nil)
Chave: 4; Node pointer: 0x1d548f0; Left pointer: (nil); Right pointer: (nil)
Chave: 13; Node pointer: 0x1d54730; Left pointer: 0x1d547f0; Right pointer: 0x1d54830
Chave: 14; Node pointer: 0x1d547f0; Left pointer: (nil); Right pointer: (nil)
Chave: 7; Node pointer: 0x1d54830; Left pointer: (nil); Right pointer: (nil)

Altura desta árvore: 3

Não é uma árvore ABB

Percurso em pré-ordem desta árvore invertida:

Chave: 10; Node pointer: 0x1d546b0; Left pointer: 0x1d54730; Right pointer: 0x1d546f0
Chave: 13; Node pointer: 0x1d54730; Left pointer: 0x1d54830; Right pointer: 0x1d547f0
Chave: 7; Node pointer: 0x1d54830; Left pointer: (nil); Right pointer: (nil)
Chave: 14; Node pointer: 0x1d547f0; Left pointer: (nil); Right pointer: (nil)
Chave: 16; Node pointer: 0x1d546f0; Left pointer: 0x1d547b0; Right pointer: 0x1d54770
Chave: 12; Node pointer: 0x1d547b0; Left pointer: (nil); Right pointer: 0x1d548f0
Chave: 4; Node pointer: 0x1d548f0; Left pointer: (nil); Right pointer: (nil)
Chave: 11; Node pointer: 0x1d54770; Left pointer: 0x1d548b0; Right pointer: 0x1d54870
Chave: 8; Node pointer: 0x1d548b0; Left pointer: (nil); Right pointer: (nil)
Chave: 17; Node pointer: 0x1d54870; Left pointer: (nil); Right pointer: (nil)
```