

Árvore Binária (AB)

1. Considerar os tipos abstratos de dados definidos a seguir.

Para o nó de uma árvore binária de números inteiros:

```
class NoArv {
private:
    NoArv *esq; // ponteiro para o filho à esquerda
    int info;    // informação (valor) do nó (inteiro)
    NoArv *dir;  // ponteiro para o filho à direita

public:
    NoArv()      { };
    ~NoArv()     { };
    void setEsq(NoArv *p) { esq = p; };
    void setInfo(int val) { info = val; };
    void setDir(NoArv *p) { dir = p; };
    NoArv* getEsq()      { return esq; };
    int getInfo()         { return info; };
    NoArv* getDir()       { return dir; };
};
```

Para a árvore binária (AB):

```
class ArvBin{
private:
    NoArv *raiz;
public:
    ArvBin();
    ~ArvBin();
};
```

Cada operação a seguir deve ser declarada no bloco privado da classe **ArvBin** e o seu chamador, com os parâmetros necessários - sem o ponteiro para **NoArv**- no bloco público.

Desenvolver as seguintes operações no MI para o TAD **ArvBin**:

- (a) **int ArvBin::impares()**. Calcular e retornar o número de nós que armazena valores ímpares fazendo um percurso pós-ordem.
- (b) **int ArvBin::soma()**. Calcular e retornar a soma dos valores armazenados nos nós da árvore.
- (c) **float ArvBin::media()**. Calcular e retornar a média dos valores armazenados nos nós da árvore.
- (d) **float ArvBin::mediaPares()**. Calcular e retornar a média dos valores pares armazenados nos nós da árvore.
- (e) **int ArvBin::maiorVal()**. Determinar e retornar o maior valor armazenado nos nós da árvore.
- (f) **int ArvBin::menorVal()**. Determinar e retornar o menor valor armazenado nos nós da árvore.

- (g) `int ArvBin::maiores(int val)`. Calcular e retornar a quantidade de nós que armazenam chaves com valores maiores que `val` fazendo um percurso pós-ordem.
- (h) `float ArvBin::mediaNivel(int nivel)`. Calcular e retornar a média dos valores armazenados num dado nível. Uma vez que o nó do nível é visitado não é necessário percorrer os níveis abaixo dele.
- (i) `void ArvBin::imprimirNivel(int nivel)`. Imprimir o valor (`info`) de todos os nós que têm níveis menores ou iguais a um dado nível, ou seja, imprimir os valores de todos os nós de nível `i` tal que $i \leq \text{nivel}$. Uma vez que o valor de um nível é impresso não é necessário percorrer os níveis abaixo dele. Fazer a operação `imprimirNivel()` realizando o percurso em:
- pré-ordem;
 - ordem;
 - pós-ordem.
- (j) `int ArvBin::sucessor(int val)`. Determinar e retornar o valor do nó da AB que representa o sucessor de `val`. O sucessor de `val` é o menor valor encontrado na AB maior que `val`. Se não existir sucessor, retornar `val`.
- (k) Nos exercícios acima, de que forma o tipo de percurso pode influenciar o resultado da operação sobre a árvore binária?
2. Implementar uma função (operação no MI) para determinar se uma árvore binária é ou não uma árvore cheia. Para uma árvore cheia tem-se: $n = 2^{h+1} - 1$, onde `n` é o número de nós e `h` a altura da AB T. Percorrer a AB apenas uma vez. Protótipo:

```
bool ArvBin::eCheia()
```

3. Implementar uma função (operação no MI) para determinar se uma árvore binária é ou não uma árvore completa. Para saber se uma árvore é completa, basta desconsiderar o nível `h` e verificar se a árvore resultante é cheia. Onde `h` é a altura da árvore. Protótipo:

```
bool ArvBin::eCompleta()
```

Nas questões 4 a 7, considerar o seguinte TAD `NoArv` usado para representar os nós de AB:

```
class NoArv {
private:
    NoArv *esq;    // ponteiro para o filho a esquerda
    int altura;    // armazena altura do nó da AB
    int info;      // informação do nó (int)
    NoArv *dir;    // ponteiro para o filho a direita
public:
    NoArv()        { };
    ~NoArv()       { };
    void setEsq(NoArv *p) { esq = p; };
    void setInfo(int val) { info = val; };
    void setAltura(int alt) { altura = alt; };
    void setDir(NoArv *d) { dir = d; };
    NoArv* getEsq() { return esq; };
    int getInfo() { return info; };
};
```

```

    int getAltura()                { return altura; };
    NoArv* getDir()                { return dir;    };
};

```

A esse novo TAD `NoArv` é adicionado o atributo privado `altura`. Essa variável membro do TAD `NoArv` armazena a altura em que se encontra o nó na AB. O novo TAD `ArvBinAlt` permanece com a mesma definição de `ArvBin` do exercício 1 (não foi definido aqui; basta trocar os nomes no TAD `ArvBin`).

4. `void ArvBinAlt::cria(int valRaiz, ArvBinAlt* sae, ArvBinAlt *sad)`. Dadas as árvores binárias `sae` e `sad`, criar uma nova AB com raiz em `raiz` e com subárvore à esquerda `sae` e à direita `sad`. Não se esquecer de calcular a altura da nova raiz.
5. `void ArvBinAlt::alturaNos()`. Calcular a altura de cada nó da AB e armazená-la no campo `altura` de cada nó. Essa operação será útil se a operação `cria()` acima não for usada.

Solution:

```

//a função altura(NoArv *p, int alt) é a mesma da questão anterior
void ArvBinAlt::alturaNos(){
    altura(raiz, 0);
    return;
}

```

6. `NoArv* ArvBinAlt::noAlt(int alt)`. Determinar e retornar um ponteiro para o primeiro nó cuja altura é igual à altura `alt` dada.

Solution:

```

NoArv* ArvBinAlt::noAlt(int alt) {
    return auxNoAlt(raiz, alt);
}

NoArv* ArvBinAlt::auxNoAlt(NoArv* T, int Alt) {
    if(T == NULL)
        return T;
    else {
        if(T->getAltura() < alt && T->getEsq() == NULL
            && T->getDir() == NULL)
            return NULL;
        else if(T->getAltura() == alt)
            return T;
        else {
            NoArv *p;

```

```

        p = auxNoAlt(T->getEsq(), alt);
        if(p != NULL && p->getAltura() == alt)
            return p;
        else {
            p = auxNoAlt(T->getDir(), alt);
            if(p != NULL && p->getAltura() == alt)
                return p;
            else
                return NULL;
        }
    }
}
}

```

7. Implementar uma função (operação no MI) para determinar se uma árvore binária segue as restrições de uma árvore AVL. Uma AB é AVL se ela é binária de busca (ver exercício para verificar se uma AB é uma ABB) e para todo nó *V* da árvore a diferença, em módulo, da altura da subárvore a esquerda do nó *V* pela altura da subárvore a direita de *V* é no máximo 1. Essa função deve obedecer ao protótipo:

```
bool ArvBinAlt::eAVL()
```

Solution:

```

int ArvBinAlt::auxeAVL(NoArv *p) {
    if(p == NULL)
        return 0;
    else {
        int alturaesq = auxeAVL(p->getEsq());
        if(alturaesq == -1)
            return -1;
        int alturadir = auxeAVL(p->getDir());
        if(alturadir == -1)
            return -1;

        if(alturadir > alturaesq) {
            if(alturadir - alturaesq > 1)
                return -1;
            else
                return (alturadir + 1);
        }
        else if(alturaesq > alturadir) {
            if(alturaesq - alturadir > 1)
                return -1;
            else

```

```

        return (alturaesq + 1);
    }
    else
        return (alturaesq + 1);
}
}

```

Árvores Binárias de Busca (ABB)

1. Implementar uma função (operação no MI) para verificar se uma árvore binária é de busca. Esta função deverá ter o protótipo

```
bool ArvBin::eABB()
```

Uma AB é uma árvore de busca se para todo nó não folha V da AB, a raiz da subárvore a esquerda, caso exista, tem valor menor que o de V e da subárvore a direita, caso exista, tem valor maior que o de V . Ou

```

(V -> getEsq()) -> getInfo() <= V -> getInfo() &&
(V -> getDir()) -> getInfo() > V -> getInfo()

```

Essa condição deve valer para qualquer nó V da árvore desde que existam uma ou as duas subárvores.

2. Considerar que numa ABB cada nó armazena as seguintes informações sobre um aluno: nome, número de matrícula, nome da disciplina, turma e média. A chave da ABB é o número de matrícula do aluno.
 - Desenvolver os TADs para o nó e para a ABB que armazena os dados de vários alunos. Considerar, além do construtor e destrutor, as operações para: inserir, remover e imprimir todos os dados de um aluno dado seu número de matrícula.
 - Desenvolver uma operação para imprimir o nome do aluno que tem a maior média (não há notas com valores iguais).
 - Desenvolver uma operação para imprimir o nome do aluno que tem a menor média (não há notas com valores iguais).
3. Refazer o exercício (2), considerando como chave da ABB a média do aluno. Qual a complexidade para o melhor caso nos itens (ii) e (iii) dos exercícios (2) e (3).
4. Considerando `ArvBinBusca`, o TAD que implementa uma ABB de inteiros e `NoArv` (ver exercício 1 sobre AB), o TAD dos nós da `ArvBinBusca`; desenvolver, usando a propriedade da ABB, as operações:
 - (a) `void ArvBinBusca::imprimeFilhos(int x)`. Dado um inteiro x , imprimir os valores dos filhos, se existir, do nó que tem valor x .
 - (b) `void ArvBinBusca::imprimeIntervalo(int x, int y)`. Dados dois números inteiros x e y , imprimir os valores dos nós que estão no intervalo $[x, y]$; considere que $x < y$.

- (c) `void ArvBinBusca::imprimeCrescente()`. Imprimir os valores dos nós da árvore em ordem crescente.
- (d) `void ArvBinBusca::imprimeDecrescente()`. Imprimir os valores dos nós da árvore em ordem decrescente.
- (e) `void ArvBinBusca::insereDoVetor(int n, int *vet)`. Dado o vetor de inteiros `vet` de tamanho `n`, inserir todos os valores de `vet` na árvore. Os valores no vetor `vet` estão em ordem crescente. Desenvolver a operação de forma que:
 - a ABB torne-se degenerada (há mais de uma forma de fazer essa operação);
 - a ABB torne-se completa.
- (f) `int* ArvBinBusca::insereVetorCrescente()`. Preencher e retornar um vetor (`int*`) com os valores armazenados na ABB. Considerar que há `n` nós na ABB. Os elementos do vetor a ser retornado devem ficar em ordem crescente.
- (g) `int* ArvBinBusca::insereNoVetDecrescente()`. Preencher e retornar um vetor (`int*`) com os valores armazenados na ABB. Considerar que há `n` nós na ABB. Os elementos do vetor a ser retornado devem ficar em ordem decrescente.
- (h) `NoArv* ArvBinBusca::buscaValor(int val)`. Dado um inteiro `val`, desenvolver a operação `buscaValor` não recursiva para achar e retornar o ponteiro para o nó da ABB cujo valor é `val`. Retornar `NULL` se `val` não for encontrado.
- (i) `int ArvBinBusca::classificaNo(int val)`. Dado um inteiro `val`. Seja `noVal`, o ponteiro para o nó que tem valor `val`; retornar 2 se `noVal` tem 2 filhos, retornar 1 se `noVal` tem 1 filho e retornar 0 se `noVal` tem 0 filhos (folha).
- (j) `void ArvBinBusca::insere(int val)`. Inserir um novo nó na ABB com valor inteiro `val`. A operação `insere` deve ser não recursiva.
- (k) `int ArvBinBusca::nos1Filho()`. Operação para calcular e retornar o número de nós com um único filho.
- (l) `int ArvBinBusca::nos2Filho()`. Operação para calcular e retornar o número de nós com 2 filhos.
- (m) `bool ArvBinBusca::estritamenteBin()`. Operação para determinar se uma árvore binária é (`true`) ou não (`false`) uma ABB estritamente binária.
- (n) `void ArvBinBusca::transfABemABB()`. Transformar a AB numa ABB completa.