M5L: Árvore

Murilo Dantas

1. **Faça um programa para executar as operações abaixo em uma árvore binária.**
   Menu
   1 – Inserir número
   2 – Mostrar todos os números
   3 – Mostrar o maior número
   4 – Mostrar o menor número
   5 – Mostrar quantos números têm na árvore
   6 – Sair

2. **Faça um programa para executar as operações abaixo em uma árvore binária.**
   Menu
   1 – Inserir número
   2 – Mostrar nós folha
   3 – Mostrar os nós ancestrais de um nó
   4 – Mostrar os descendentes de um nó
   5 – Mostrar o nó pai e os nós filhos de um nó
   6 – Sair

3. **Faça um programa para executar as operações abaixo em uma árvore binária.**
   Menu
   1 – Inserir número
   2 – Mostrar todos
   3 – Mostrar a sub-árvore direita de um nó
   4 – Mostrar a sub-árvore esquerda de um nó
   5 – Sair

4. **Apenas imprimindo linearmente os elementos da árvore não é possível reproduzir sua estrutura. Faça um método imprimeRelacoes que percorra a árvore imprimindo as relações entre os nós, de forma que se possa através dessa descrição reproduzir a estrutura de uma árvore. Exemplos de descrições impressas são:**
   - o nó de valor XXX é filho esquerdo de YYY
   - o nó de valor ZZZ é filho direito de YYY
   - o nó de valor XXX não tem filho esquerdo
   - o nó de valor ZZZ não tem filho direito

5. **Faça um programa para executar as operações abaixo em uma árvore binária.**
   Menu
   1 – Inserir número
   2 – Mostrar a altura da arvore
   3 – Mostrar o nível de um nó
   4 – Sair

6. **Faça um programa para executar as operações abaixo em uma árvore binária.**
   Menu
   1 – Inserir número
   2 – Mostrar se a arvore é estritamente binária
   3 – Mostrar se a arvore é completa
   4 – Mostrar se a arvore é cheia
   5 – Sair

7. **Faça um programa para executar as operações abaixo em uma árvore binária não ordenada.**
   Menu
   1 – Inserir número aleatório com geração e inserção aleatórias
   2 – Mostrar os números da árvore usando o imprimeRelacoes da questão 4
   3 – Imprimir emOrdem, preOrdem e posOrdem
   4 – Mostrar o maior número na árvore
   5 – Sair

8. **Faça um programa para executar as operações abaixo em uma árvore binária ordenada pelo código, onde o dado não seja um inteiro, mas uma estrutura contendo dados de um estudante: código, nome, idade, altura e média acadêmica.**
   Menu
   1 – Inserir estudante
   2 – Mostrar o estudante mais alto
   3 – Mostrar o estudante mais velho
   4 – Mostrar os estudantes maiores de idade
   5 – Mostrar o estudante com maior média acadêmica
   6 – Mostrar o estudante com menor média acadêmica
   7 – Sair

9. **Crie uma árvore para armazenar os elementos da seguinte expressão matemática: A + B * C – D / E. Cada letra ou símbolo matemático deverá ocupar a posição de um elemento. O arranjo dos elementos deve seguir uma ordem que gere uma saída representando a ordem em que a expressão deve ser executada, considerando a precedência dos operadores.**

10. **Crie uma árvore binária ordenada para implementar um dicionário de dados. Cada nó precisa ter a palavra e o seu significado. Implemente as funções básicas para inserção, remoção, pesquisa e impressão do dicionário.**

Se você quiser se aprofundar, segue uma árvore com Orientação a Objetos:

Código exemplo

No.h:

```
#include <iostream.h>

class No
{
	int info;
	No* esquerdo;
	No* direito;

public:
	No(int elem);//construtor
	~No();//Destrutor

	/* Métodos de Acesso
	 */
	int getInfo();
	void setInfo(int elem);
	No *getEsquerdo();
	void setEsquerdo(No *novoNo);
	No *getDireito();
	void setDireito(No* novoNo);
	//Fim dos métodos de acesso

	bool eFolha();
	/* Verifica se os filhos esquerdo e direito do
	 * nó são nulos
	 */
};
```

No.cpp:

```
#include "no.h"

No::No(int elem)
{
	info = elem;
	esquerdo = direito = NULL;
}

No::~No()
{
	cout << "No de valor " << info << " destruido!\n";
	//mensagem apenas para certificação de que o destrutor
	//esteja sendo chamado apropriadamente.
}

int No::getInfo()
{
	return info;
}
```

```
void No::setInfo(int elem)
{
	info = elem;
}

No* No::getEsquerdo()
{
	return esquerdo;
}

void No::setEsquerdo(No *novoNo)
{
	esquerdo = novoNo;
}

No* No::getDireito()
{
	return direito;
}

void No::setDireito(No* novoNo)
{
	direito = novoNo;
}

bool No::eFolha()
{
	if(esquerdo == NULL && direito ==   NULL)
		return true;
	return false;
}
```

Arvore.h:

```
#include "no.h"
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

const int ESQUERDA = -1;
const int DIREITA = 1;
const int ARVORE_VAZIA = -1;
const int LIMITE_PADRAO = 100;

class Arvore
{
	No* raiz;

public:
	Arvore();//construtor
	~Arvore();//destrutor

	bool eVazia();
```

```
        /* verifica se a árvore é uma árvore vazia
         */

        void insere(int elem);
        /* Insere um elemento na árvore, se ele nao existir,
         * de acordo com o seguinte algoritmo:
         * começando pela preferencia de inserir à esquerda, procura
         * o nó nulo com nível mais baixo possível para inserir o
         * novo nó. Ao inserir, a preferencia é invertida para a próxima
         * inserção.
         */

        void remove(int elem);
        /* Remove  da árvore o nó com o valor dado por "elem",
         * se ele existir
         */

        void imprimeCentral();
        /* Imprime os nós da árvore que é varrida de forma central:
         * esquerda - raiz - direita
         */

        bool existe(int elem);
        /* Verifica se o nó de valor "elem" existe na árvore
         */

        void geraArvore(int nElementos, int limite = LIMITE_PADRAO);
        /* Gera uma árvore com nElementos  aleatórios cujo valor vai de 0 até limite.
         * A inserção de cada elemento gerado é feita também de forma aletatória.
         * A preferência de inserção à esquerda ou à direita pode ser
         * invertida ou não a cada nível descido na busca de um nó vazio.
         */

private:
        No* procura(int elem);
        /* Procura pelo nó de valor "elem", retornando um ponteiro para
         * este nó. Retorna NULL caso esse nó não exista.
         */

        No* procuraPai(No* umNo);
        /* Procura o pai de "umNo" dentro da árvore. Se umNo não
         * existir, ou for a raiz da árvore, retorna NULL.
         */
};
```

Arvore.cpp:

```
#include "arvore.h"

Arvore::Arvore()
{
        raiz = NULL;
}
```

```
Arvore::~Arvore()
{

	Arvore *aux=new Arvore();
	if(!eVazia())
	{
		if(raiz->getEsquerdo() != NULL)
		{//se há uma subárvore à esquerda...
			aux->raiz = raiz->getEsquerdo();
			aux->~Arvore();//...remover essa árvore
		}
		if(raiz->getDireito() != NULL)
		{//se há uma subárvore à direita...
			aux->raiz = raiz->getDireito();
			aux->~Arvore();//...remover essa árvore também
		}
		delete raiz;//remover a raiz
	}

}

bool Arvore::eVazia()
{
	if(raiz == NULL)
		return true;
	return false;
}

bool Arvore::existe(int elem)
{
	if(eVazia())
		return false;

	if(raiz->getInfo() == elem)//procura o elemento na raiz,
		return true;  //se encontrar, encerra a busca

	Arvore *pS = new Arvore();
	pS->raiz = raiz->getEsquerdo();

	if(pS->existe(elem) == true)//procura o elemento na subárvore esquerda
		return true; //se encontrar, encerra a busca.

	pS->raiz = raiz->getDireito(); //se não encontrou...
	return pS->existe(elem); //procura na subárvore direita
}

void Arvore::insere(int elem)
{
	if(existe(elem))
		return; //insere não admite a inserção de elementos repetidos

	if(eVazia())
	{
		raiz = new No(elem);
		return;
```

```
	}

	static int preferencia = ESQUERDA;

	No* pAux=raiz;
	do{//procura repetidamente um nó vazio, com a
		//preferência dada
		if(preferencia == ESQUERDA)
		{
			if(pAux->getEsquerdo() != NULL)//o nó esquerdo não está vazio
			{
				pAux = pAux->getEsquerdo();//procura agora a partir dele
				continue;
			}
			else//o nó esquerdo está vazio
			{
				pAux->setEsquerdo(new No(elem));//insere o elemento
				preferencia*=-1;//muda a preferência na próxima inserção
				break; //encerra a procura por um noh vazio
			}
		}
		else//preferencia == direita, mesmo algoritmo
		{
			if(pAux->getDireito() != NULL)
			{
				pAux = pAux->getDireito();
				continue;
			}
			else
			{
				pAux->setDireito(new No(elem));
				preferencia*=-1;
				break; //encerra a procura por um noh vazio
			}
		}
	} while(true);
}//fim de insere

void Arvore::remove(int elem)
{
	if(eVazia() || !existe(elem))
		return;

	No* pNo = procura(elem); //procura o nó na árvore

	while(!pNo->eFolha())//enquanto o nó apontado por pNo não for uma folha...
	{
		if(pNo->getEsquerdo() != NULL)// o nó esquerdo não é vazio
		{
			pNo->setInfo((pNo->getEsquerdo())->getInfo()); //copia o valor do filho
esquerdo
			pNo = pNo->getEsquerdo(); //aponta para o filho esquerdo, para
removê-lo
		}
		else//o nó direito é que não é vazio...
```

```
                {
                        pNo->setInfo((pNo->getDireito())->getInfo()); //copia o valor do filho
direito
                        pNo = pNo->getDireito();//aponta para o filho direito, para removê-lo
                }

        }

                No* pPai = procuraPai(pNo);//procura o pai do nó a ser removido
                if(pPai->getEsquerdo() == pNo)
                        pPai->setEsquerdo(NULL);//anula a referência para o filho...
                else if(pPai->getDireito() == pNo)//...seja ele o esquerdo...
                        pPai->setDireito(NULL); // .. ou o direito.
                else
                                raiz = NULL; //se o não tem pai, o nó é a raiz.

                delete pNo;//remove o nó da memória

}
void Arvore::imprimeCentral()
{//esquerda-raiz-direita
        if(eVazia())
        {
                cout << "Arvore vazia!\n";
                return;
        }

        Arvore* aux = new Arvore;

        //impressao da subárvore esquerda
        if(raiz->getEsquerdo() != NULL)
        {
                aux->raiz=raiz->getEsquerdo();
                aux->imprimeCentral();//chamada recursiva a imprimeCentral
        }

        //impressao da raiz
        cout << " - " << raiz->getInfo();

        //impressao da subárvore direita
        if(raiz->getDireito() != NULL)
        {
                aux->raiz=raiz->getDireito();
                aux->imprimeCentral();//chamada recursiva a imprimeCentral
        }

}

void Arvore::geraArvore(int nElementos, int limite)
{
        int valor, preferencia=ESQUERDA, tentativas=0;

        if (limite <= 0)
                limite = LIMITE_PADRAO;
```

```
	srand((unsigned)time( NULL ));
	//semente para a sequência de números aleatórios

	for(int i=0; i<nElementos; i++)
	{
		do{
			valor = rand(); //geração de número aleatório
			valor %= limite+1; //o valor gerado é colocado dentro do limite
			if(tentativas++ > limite)//evita um loop infinito...
				return; //...caso todos os valores já tenham sido gerados
		}	while(existe(valor));//não permite inserção de valores repetidos

		if(eVazia())
				raiz = new No(valor);
		else
		{
			No* pAux=raiz;
			do{ //busca para um nó vazio
				if(rand() % 2 == 0) //troca ou nao a preferencia de busca
					preferencia*=-1; // essa troca é aleatória

				if(preferencia == ESQUERDA)
				{
					if(pAux->getEsquerdo() != NULL)// o nó
esquerdo não é vazio
					{
						pAux = pAux->getEsquerdo();
						continue;//continua a busca
					}
					else//nó esquerdo vazio
					{
						pAux->setEsquerdo(new
No(valor));//insere o novo nó
						break; //encerra a procura por um noh
vazio
					}
				}
				else//preferência == direita, mesmo algoritmo
				{
					if(pAux->getDireito() != NULL)
					{
						pAux = pAux->getDireito();
						continue;
					}
					else
					{
						pAux->setDireito(new No(valor));
						break; //encerra a procura por um noh
vazio
					}

				}
			} while(true);
		}//fim do for
	}//fim do if
```

```
}//fim de geraArvore

/* Métodos privados da árvore, auxiliares de remove( )
 */
No* Arvore::procura(int elem)
{
	if(eVazia())
		return NULL;

	if(raiz->getInfo() == elem)
		return raiz;//o elemento está na raiz.
	else
	{
			Arvore *pSub=new Arvore;
			No* pN;
			pSub->raiz = raiz->getEsquerdo();//procura na subárvore esquerda
			pN=pSub->procura(elem);
			if(pN != NULL)
				return pN;

			//nao foi encontrado à esquerda:
			pSub->raiz = raiz->getDireito();//procura na subárvore direita
			pN=pSub->procura(elem);

			return pN;
	}
}

No* Arvore::procuraPai(No* umNo)
{
	if(eVazia() || raiz == umNo)
		return NULL; //o nó não tem pai

	Arvore* pAux = new Arvore;
	pAux->raiz = raiz;

	if(pAux->raiz->getEsquerdo() == umNo || pAux->raiz->getDireito() == umNo)
		return pAux->raiz;// o pai é a raiz.

	pAux->raiz = raiz->getEsquerdo();
	No *pN;
	if((pN=pAux->procuraPai(umNo)) != NULL)
		return pN; //procura o pai na subárvore esquerda
	else
	{ //procura o pai na subárvore direita
		pAux->raiz = raiz->getDireito();
		return pAux->procuraPai(umNo);
	}
}
```

Teste.cpp:

```
#include "arvore.h"
```

```
//opções de menu
#define GERA_ARVORE 1
#define INSERE_NOH 2
#define REMOVE_NOH 3
#define IMPRIME_CENTRAL 4
#define EXISTE 5
#define SAI 0

int menu();

void main()
{
	Arvore arv;
	int opc, valor;

	while((opc=menu())!= SAI)
	{
		switch(opc)
		{
			case INSERE_NOH:
				cout << " Valor? ";
				cin >> valor;
				arv.insere(valor);
				break;

			case REMOVE_NOH:
				cout << "qual o valor? ";
				cin >> valor;
				arv.remove(valor);
				cout << "Remocao executada!\n";
				break;

			case IMPRIME_CENTRAL:
				arv.imprimeCentral();
				break;

			case EXISTE:
				cout << "Que valor?";
				cin >> valor;

				if(arv.existe(valor))
					cout << "Elemento encontrado!";
				else
					cout << "Elemento nao encontrado!\n";
				break;

			case GERA_ARVORE:
				cout << "Quantos elementos?";
				cin >> valor;
				arv.geraArvore(valor);
				cout << "Arvore gerada!\n";
				break;

			case SAI:
				break;
```

```
                        default:
                                cout << " Opcao nao implementada!\n";
                }
        }

        cout << " Ateh a proxima arvore!!!!\n";
}

int menu()
{
        int op;
        cout << endl << GERA_ARVORE << " - gerar uma arvore aleatoriamente\n";
        cout << INSERE_NOH << " - inserir um elemento\n";
        cout << REMOVE_NOH << " - remove um noh da arvore\n";
        cout << IMPRIME_CENTRAL << " - imprimir com varrimento central\n";
        cout << EXISTE << " - procura um elemento da arvore\n";
        cout << SAI << " - sair \n";

        cin >> op;
        return op;
}
```

Os exercícios abaixo são complementares à árvore de inteiros implementada no exemplo. Implemente-os incluindo o que é solicitado a cada exercício, incrementando o código do exemplo acima.

11. **Implemente um método chamado imprimePre, com varrimento pré-fixado e outro chamado imprimePos, com varrimento pós fixado.**

12. **Apenas imprimindo linearmente os elementos da árvore não é possível reproduzir sua estrutura. Faça um método imprimeRelacoes que percorra a árvore imprimindo as relações entre os nós, de forma que se possa através dessa descrição reproduzir a estrutura de uma árvore. Exemplos de descrições impressas são:**
    - o nó de valor XXX é filho esquerdo de YYY
    - o nó de valor ZZZ é filho direito de YYY
    - o nó de valor XXX não tem filho esquerdo
    - o nó de valor ZZZ não tem filho direito

13. **Implemente um método chamado altura que calcula e retorna a altura de uma árvore.**

14. **Implemente um método chamado nivelDe que calcula e retorna o nível em que se encontra um nó, dado o seu valor como parâmetro. Se o nó não existir na árvore, o valor retornado deve ser –1, definido como constante simbólica INEXISTENTE.**

15. **Faça uma nova opção de menu para gerar uma árvore com um limite diferente do limite padrão.**

    Crie uma classe ArvoreDeBusca que seja uma subclasse de Arvore para executar os próximos exercícios.

16. **Sobreponha o método insere para inserir os elementos da seguinte forma: todos os elementos na subárvore à esquerda de qualquer elemento têm que ser menores que ele e todos os elementos na subárvore da direita têm que ser maiores do que ele.**

17. **Sobreponha o método geraArvore para gerar árvores ordenadas de acordo com o método insere acima. Use o imprimeCentral e o imprimeRelacoes para mostrar que a árvore está sendo gerada corretamente.**

18. **Crie um método chamado procuraMenor que retorna um ponteiro para o nó de menor valor dentro da árvore binária de busca.**

19. **Crie um método semelhante chamado procuraMaior.**

20. **Verifique o que acontece ao remover um elemento de uma árvore ordenada de acordo com o método remove herdado.**

21. **Sobreponha o método remove( ) de forma que ele mantenha o ordenamento da árvore resultante. DICA: utilize os métodos procuraMenor e procuraMaior criados anteriormente.**

22. **Sobreponha os métodos existe e procura para encerrarem as buscas assim que os elementos restantes da árvore forem maiores do que aquele elemento que se procura.**