Universidade do Minho	Escola de Engenharia da Universidade do Minho **Mestrado Integrado em Eng. Electrónica Industrial e Computadores** **Complementos de Programação de Computadores**	**2014/2015** **MIEEIC** **(1º Ano)** **2º Sem**
Luís Paulo Reis		
Aula Prática 9: Árvores Binárias de Pesquisa		

Objectivos:

Esta Folha de Exercícios destina-se a:

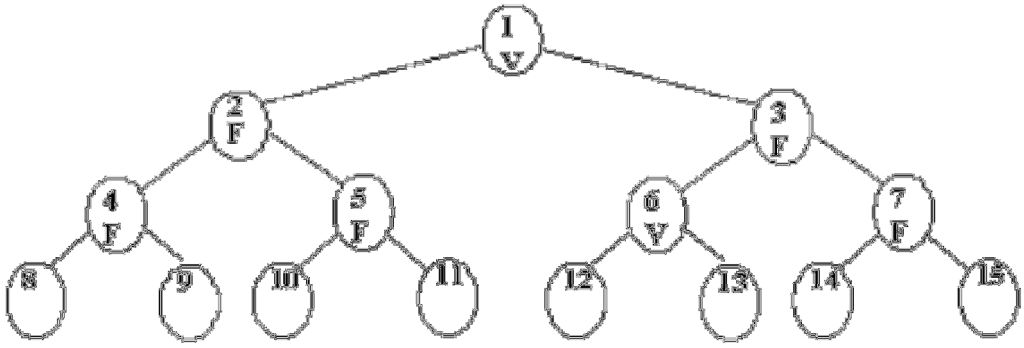
• Compreender a utilização de árvores binárias de pesquisa;

Os exercícios aqui propostos deverão ser realizados no mais simples ambiente de desenvolvimento possível para a linguagem C: editor de texto de programação ou editor DevC++ e ferramentas da GCC (GNU Compiler Collection) e afins.

Exercício 9

Faça download do ficheiro ***Prog2_Aula9.rar*** do blackboard e descomprima-o (contém os ficheiros ***Jogo.h, Jogo.cpp, Dicionario.h, Dicionario.cpp***, ***Test.cpp, BinaryTree.h, BST.h, xceptions.h e dic.txt***). Deverá realizar esta ficha respeitando a ordem das alíneas.

Uma bola é lançada sobre um conjunto de círculos dispostos sob a forma de uma árvore binária completa (figura). (*nota*: uma árvore binária completa tem todos os seus níveis completamente preenchidos)



Cada círculo tem uma pontuação, e um estado representado por um valor booleano que indica qual o caminho que a bola percorre quando chega a esse círculo: se o estado é igual a falso, a bola vai para a esquerda; se é igual a verdadeiro, a bola vai para a direita. Quando a bola passa por um qualquer círculo, este muda o seu estado: se era verdadeiro passa a falso; se era falso passa a verdadeiro. Sempre que a bola passa num círculo, deve ser incrementado o valor de visitas a esse círculo. Quando a bola atinge um círculo na base (nó da árvore), o jogador que lançou a bola ganha o número de pontos inscritos nesse círculo. Ganha o jogo o jogador que conseguir a maior soma de pontos em uma série de *n* lançamentos.

Utilize uma árvore binária (**BinaryTree**) para representar o conjunto de círculos que constituem o tabuleiro de jogo (classe **Jogo**). A informação contida em cada nó da árvore está representada na classe **Circulo**:

```
class Circulo
{
    int pontos;
    bool estado;
    int nVisitas;
  public:
    Circulo(int p=0, bool e=false): pontos(p), estado(e), nVisitas(0) {}
    int getPontos() const { return pontos; }
    bool getEstado() const { return estado; }
    void mudaEstado() {
        if (estado==false) estado=true; else estado=false;
    }
    int getNVisitas() const { return nVisitas; }
    int incNVisitas() { return nVisitas++; }
};

class Jogo
{
    BinaryTree <Circulo> jogo;
  public:
    BinaryTree <Circulo> &getJogo() { return jogo; }
};
```

a) Implemente o construtor da classe Jogo, que cria um tabuleiro de jogo:

```
Jogo::Jogo(int niv, vector<int> &pontos, vector<bool> &estados )
```

Esta função cria uma árvore binária completa, de altura *niv*. Os vetores *pontos* e *estados* representam a pontuação e o estado dos círculos (nós da árvore) quando se efetua uma visita por nível.

Nota: Se numerar a posição dos nós de uma árvore visitada por nível de 0 a n-1 (n= nº de nós da árvore), o nó na posição *p* possui o filho esquerdo e o filho direito nas posições *2*p+1* e *2*p+2*, respectivamente.

b) Implemente a função que descreve o estado do jogo em um determinado instante:

```
string Jogo::escreveJogo()
```

Esta função retorna a informação sobre todo o tabuleiro de jogo no formato *"pontuacao-estado-nVisitas\nl"* para cada um dos círculos, onde *pontuacao* é um inteiro, *estado* é "true" ou "false" e *nVisitas* é o nº de visitas já efetuadas a esse círculo, quando a árvore de jogo é percorrida por nível.

c) Implemente a função que realiza uma jogada:

```
int Jogo::jogada()
```

Esta função realiza uma jogada, segundo as regras já descritas, devendo alterar o estado e incrementar o número de visitas de todos os círculos por onde a bola passa. A função retorna a pontuação do círculo base (folha da árvore) onde a bola lançada termina o seu percurso. Sugestão: use um iterador por nível para percorrer a árvore.

d) Implemente a função que determina qual o círculo mais visitado:

```
int Jogo::maisVisitado()
```

Esta função retorna o número de visitas realizadas ao círculo mais visitado até ao momento, de todas as jogadas já realizadas (com exceção da raiz da árvore, claro).

2) Dicionários eletrónicos são ferramentas muito úteis. Mas se não forem implementados numa estrutura adequada, a sua manipulação pode ser bastante morosa. Pretende-se, portanto implementar um dicionário utilizado uma árvore de pesquisa binária (**BST**) onde as palavras se encontram ordenadas alfabeticamente. Considere que a árvore contém objetos da classe **PalavraSignificado:**

```
class PalavraSignificado
{
    string palavra; // a palavra
    string significado; // o significada da palavra (explicação)
  public:
    PalavraSignificado(string p, string s): palavra(p), significado(s){}
    string getPalavra() { return palavra; }
    string getSignificado() { return significado; }
    void setSignificado(string sig) { significado = sig; }
};
```

Pretende-se implementar a classe `Dicionario`, com a seguinte estrutura

```
class Dicionario
{
    BST<PalavraSignificado> palavras;
  public:
    BST<PalavraSignificado> getPalavras() const;
};
```

a) Complete a classe **Dicionario**, declarando convenientemente o membro-dado *palavras*. Implemente também o membro-função:

```
void Dicionario::lerDicionario(ifstream &fich)
```

que lê, a partir de um ficheiro, as palavras e o significado e guarda essa informação na árvore. No ficheiro, a primeira linha contém a palavra e a linha seguinte o seu significado, ...

```
gato
mamífero felino
banana
fruta
...
```

b) Implemente na classe **Dicionario** o membro-função:

```
void Dicionario::imprime() const
```

que imprime no monitor o conteúdo do dicionário ordenado alfabeticamente por palavras, no formato:

```
palavra1
significado da palavra1
palavra2
significado da palavra 2
....
```

c) Implemente na classe **Dicionario** o membro-função:

```
string Dicionario::consulta(string palavra) const
```

que retorna o significado da palavra indicada como parâmetro. Se a palavra não existir, lança a excepção **PalavraNaoExiste**.

Esta classe possui os seguintes métodos que deve implementar:

```
string PalavraNaoExiste::getPalavraAntes() const
      // retorna a palavra imediatamente antes
string PalavraNaoExiste::getSignificadoAntes() const
      // retorna o significado da palavra imediatamente antes
```

```
string PalavraNaoExiste::getPalavraApos() const
      // retorna a palavra imediatamente após
string PalavraNaoExiste::getSignificadoApos () const
      // retorna o significado da palavra imediatamente após
```

d) Implemente na classe **Dicionario** o membro-função:

```
bool Dicionario::corrigir(string palavra, string significado)
```

que modifica o significado da palavra para um novo significado indicado no argumento. Altere convenientemente a classe **PalavraSignificado** a fim de facilitar a implementação desta função. Se a palavra para a qual se pretende alterar o significado existir o método retorna true, senão esta nova palavra é adicionada ao dicionário e o método retorna false s).