Árvore Trie

A árvore trie serve para a busca rápida de padrões. Cada nó vai ter uma tabela (hash) com 255 posições (vou exemplificar uma árvore trie de strings). De cada posição do hash sai um ponteiro, que inicialmente aponta para “null”. Para inserir o “a” o ponteiro da posição 0 vai apontar para um nó cujo caracter é “m” que por sua vez terá o ponteiro da posição “o” apontando para um próximo nó. Assim formamos a palavra “amo”. Quando chegamos na última letra da palavra temos que sinalizar que é a última. Nenhuma palavra que possui o mesmo prefixo pode ser inserida na árvore como, por exemplo, “amor” e “amoroso” porém “amar” pode pois ela não usa o último caracter da outra palavra.

Vamos entender a classe nó:

```
class No
{
	public char elemento;
	public int tamanho = 255;
	public No[] prox;
	public boolean folha;

	public No()
	{
		this (‘ ‘);
	}

	public No (char elemento)
	{
		this.elemento = elemento;
		prox = new No[tamanho];
		for (int i = 0; i < tamanho; i++) prox[i] = null;
		folha = false;
	}

}
```

O nó tem o elemento que é o char que guarda, o tamanho de seu array e um array do tipo nó, chamado prox. Esse array é um hash que possui função de transformação (no caso transforma todas as letras em números) e cada posição tem um ponteiro que aponta para outro nó, por isso “prox”.
A variável booleana “folha” serve para indicar se aquela é a última letra da palavra e todos os ponteiros devem ser inicializados como “null”.

```
public void inserir (String s) throws Exception
{
```

```
        inserir(s, raiz, 0)
}
```

Esse método recebeu a string que deve ser inserida vai chamar um método recursivo.

OBS: se colocarmos array[s.charAt(índice)] a posição é transformada para o seu respectivo valor da tabelas ascii. Se quisermos, podemos fazer uma função de transformação. “return (int)x”

```
private void inserir (String s, No no, int i)
{
        if(no.prox[s.charAt(i)]) == null)
        {
                no.prox[s.charAt(i)] = new No(s.charAt(i));

                if(i == s.length() - 1) // última posição
                {
                        no.prox[s.charAt(i)].folha = true;
                }
                else
                {
                        inserir (s, no.prox[s.charAt(i), i+1);

        }else if (no.prox[charAt(i)].folha == false && i < s.length() - 1)
        {
                inserir (s, no.prox[s.charAt(i)], ++i);
        }else
        {
                throw new Exception (“Erro ao inserir”);
        }
}
```

primeiro passo: verificar se o nó é nulo, se sim, inserimos um elemento. Após isso devemos verificar se é a última letra da palavra (palavra.length() - 1) se for é preciso sinalizar que temos uma folha aí! Se for uma folha, o processo de inserir se encerra. Caso não seja uma folha, chamamos o processo recursivamente até que seja e acrescemos o i em uma unidade para verificarmos o próximo caracter.

passo 2: se o nó não for nulo, significa que já existem um ponteiro naquela posição do hash que aponta para esse valor. Só podemos inserir se o próximo nó não for

folha e i for menor que palavra.length() -1, por que? Por que se o próximo nó for folha significa que uma palavra com o mesmo prefixo e menor do que essa, já foi inserida (como se tentássemos inserir “amoroso” após “amor”). E tem essa condição para a variável i, pois se “i” for maior ou igual (no caso só vai rolar igual) que a última posição da palavra significa que aquela é sua última letra e ali deveria ser inserido uma folha mas se o nó não é null, já tem uma palavra com o mesmo prefixo maior do que a palavra em questão (isso foi muito inteligente, eu gostei), é como se tentássemos inserir “amor” em “amoroso”.

passo 3: se não for nenhuma das alternativas acima, algo está incorreto e um erro deve ser gerado.

O pesquisar é bem intuitivo

```
  public boolean pesquisar(String s) throws Exception {
      return pesquisar(s, raiz, 0);
   }

   public boolean pesquisar(String s, No no, int i) throws Exception {
      boolean resp;
      if(no.prox[s.charAt(i)] == null){
         resp = false;
      } else if(i == s.length() - 1){
         resp = (no.prox[s.charAt(i)].folha == true);
      } else if(i < s.length() - 1 ){
         resp = pesquisar(s, no.prox[s.charAt(i)], i + 1);
      } else {
         throw new Exception("Erro ao pesquisar!");
      }
      return resp;
   }
```

O método mostrar é muito bacana e a partir dele podemos fazer outros como merge, contar letras, pesquisar caracteres....

```
public void mostrar()
{
        mostrar (“”, raiz);
}
public void mostrar(String s, No no)
```

```
{
        if (no.folha == true)
        {
                MyIO.println (s + no.elemento);
                for(int i = 0; i < no.prox.length; i++)
        } else if(no.prox[i] != null)
        {
                                mostrar(s + no.elemento, no.prox[i]);
        }
                }
        }
}
```

```
  public int contarAs(){
     int resp = 0;
     if(raiz != null){
        resp = contarAs(raiz);
     }
     return resp;
  }

   public int contarAs(No no) {
     int resp = (no.elemento == 'A') ? 1 : 0; // amei isso

     if(no.folha == false){
        for(int i = 0; i < no.prox.length; i++){
           if(no.prox[i] != null){
              resp += contarAs(no.prox[i]);
           }
        }
     }
     return resp;
  }
```