Algoritmos e Estruturas de Dados

Estudo Dirigido - Semana 7

Professor: Tadeu Zubaran

Sumário

.

1 Introdução

Nesta semana estudaremos a estrutura de dados árvores.

2 Árvores

Uma árvore enraizada T (vamos omitir a palavra enraizada pelo resto deste estudo) é um conjunto de nodo (nós ou vértices), tal que

- cada nodo tem um subconjunto destes nodos como seus *filhos*.
- Um nodo é considerado *pai* dos seus nodos filhos.
- Cada nodo pode ser filho de apenas um nó.
- Somente um nodo não tem pai, e é chamado de *raiz*.
- Um nodo que não tem filhos é chamado de *folha*.
- Nodos que não são folhas ou a raiz são chamados de nodos *internos*.
- Não tem ciclos, e é conexa.

Note que uma árvore é um grafo direcionado conexo e acíclico, que tem um nodo especial, a raiz.

A altura de um nodo é zero se o nodo é a raiz, ou o comprimento do (único) caminho do nodo para a raiz caso contrário. A altura da árvore é a maior altura de seus nodos.

O grau de um nodo é seu número de filhos.
Vamos considerar a árvore T cujo conjunto de nodos é $\{A, B, C, D, E, F, G\}$. Onde

- A é a raiz da árvore,
- A tem filhos B, C e D
- B tem filho E,
- D tem filhos F e G, e
- C, E, F e G não tem filhos.

Existem diversas maneiras de se representar uma árvore, vamos mostrar maneiras alternativas de representar a árvore T.

2.1 Diagrama de Inclusão

O *diagrama de inclusão* mostra a árvore utilizando a representação gráfica para conjuntos mais comum, que é o diagrama de Venn. Cada nodo contém seus nodos filhos. A árvore T pode ser vista na Figura 1.

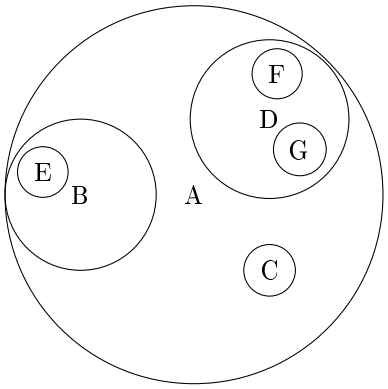

Figura 1: Representação da árvore T por diagrama de inclusão.

2.2 Parênteses Aninhados

Podemos representar uma árvore por parênteses aninhados onde iniciamos abrimos um parêntese, escrevemos o rótulo do nodo, e a seguir colocamos seus filhos, que por sua vez estão entre parênteses, finalmente fechando adequadamente o parênteses aberto.

A árvore T é representada da forma $(A(B(E))(C)(D(F)(G)))$

2.3 Representação Hierárquica

A maneira mais comum de representar uma árvore é a maneira que usamos para representar grafos. Nodos são círculos, conectados aos seus filhos. Por convenção não usamos direção nas conexões, mas desenhamos as árvores de cima para baixo, iniciando pela raiz, deixando implícita a direção das conexões no desenho.

Na Figura 2 temos a representação hierárquica de T

Uma vantagem desta representação é a visualização intuitiva da altura dos nodos. A altura está mostrada na Figura 2 como h.

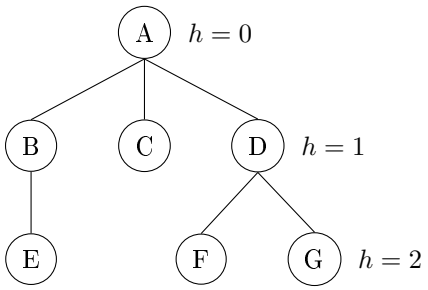

Figura 2: Representação hierárquica da árvore T.

2.4 Representação de Árvores em Memória

Vamos usar novamente a classe *Aluno* dos estudo anteriores, para testar nossa estrutura de dados.

Existem maneiras de se representar árvores de forma estática, mas elas são incomuns, a forma mais comum, e que mostraremos aqui, é a representação dinâmica. Para representar uma árvore utilizaremos o conceito de *Nodo*, assim como fizemos na lista encadeada:

```
1  class Nodo {
2  public:
3          Nodo() {  }
4          Nodo{const Aluno & novoAluno) {
5                  aluno = novoAluno;
6          }
7          ~ Nodo() { }
8
9          //operacoes a serem feitas
10
11         ListaNodoPtr filhos;
12         Aluno aluno;
13 };
```

Note que temos uma **Lista** de ponteiros para os nodos filhos. Podemos utilizar qualquer das listas aprendidas, com suas vantagens e desvantagens. Por exemplo para usarmos listas estáticas temos que garantir que o tamanho seja suficiente para conter o grau máximo da árvore.

A classe que representa a árvore precisa apenas de um ponto de entrada, a raiz da árvore. Deixamos as operações que complementam o TAD para a parte final deste estudo.

```
1  class Arvore {
2  public:
3          Arvore() {  }
4          ~ Nodo() {
5                  //a ser feito
```

```
6           }
7
8           //operacoes a serem feitas
9
10          Nodo * raiz;
11  };
```

3 Árvores Binárias

Em uma **árvore binária** cada nodo tem **no máximo** dois filhos (subárvores). Estes filhos recebem o nome de filho da esquerda e filho da direita.

A árvore T mostrada anteriormente não é binária pois o *Nodo A* tem 3 filhos. Por outro lado a *árvore* T_b na Figura 3 é uma *árvore binária*.

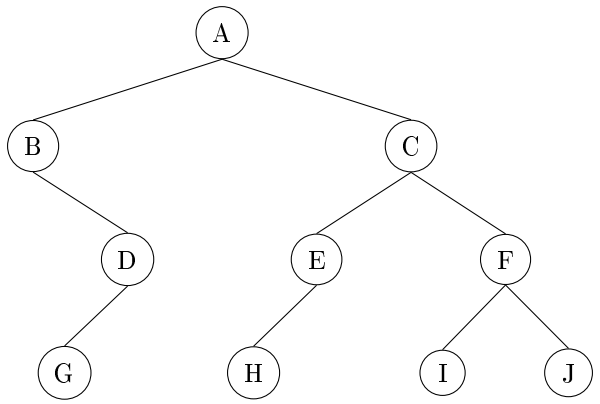

Figura 3: Árvore binária T_b.

3.1 Representação de Árvores Binárias em Memória

Como agora temos apenas dois filhos a representação dos nodos da árvore fica um pouco mais simples

```
1   class Nodo {
2   public:
3           Nodo() {  }
4           Nodo{const Aluno & novoAluno) {
5                   aluno = novoAluno;
6           }
7           ~ Nodo() { }
8
9           //operacoes a serem feitas
10
11          Nodo * filhoEsquerda;
12          Nodo * filhoDireita;
13          Aluno aluno;
```

```
14 };
```

Temos agora apenas um ponteiro para o filho da esquerda em um filho da direita. A classe árvore não recebe modificação, e continua apenas contendo o ponteiro para a raiz.

3.2 Percursos

Temos mais de uma forma de percorrer uma árvore binária.

3.2.1 Percurso em Pré-ordem

Vamos iniciar invocando o método pela classe *Arvore*,

```
1  class Arvore {
2          ...
3      void preOrdem() const {
4          if(raiz)
5              raiz->preOrdem();
6          cout << endl;
7      }
8      ...
9      Nodo * raiz;
10 };
```

que por sua vez invoca o método na classe *Nodo*.

```
1  class Nodo {
2      ...
3      void preOrdem() const {
4          aluno.imprime();
5          if(filhoEsq)
6              filhoEsq->preOrdem();
7          if(filhoDir)
8              filhoDir->preOrdem();
9      }
10     ...
11     Nodo * filhoEsq;
12     Nodo * filhoDir;
13     Aluno aluno;
14 };
```

Desta forma visitamos um nodo (representamos essa visita com o método *imprime()*), iniciando pela raiz, e após invocamos o método, de forma recursiva, para ambos os filhos.

No caso da árvore T_b o resultado do percorrimento em pré-ordem é:

$$A, B, D, G, CE, H, F, I, J$$

3.2.2 Percurso em Pós-Ordem

O percurso em pós-ordem é muito similar ao percurso em pré-ordem, com a modificação que a chamada recursiva é feita antes da visita ao nodo atual.

Vamos omitir a chamada na classe *Arvore*, pois é igual ao percurso em pré-ordem, e o código do percurso em pós-ordem em *Nodo* é

```
1  class Nodo {
2      ...
3      void posOrdem() const {
4          if(filhoEsq)
5              filhoEsq->posOrdem();
6          if(filhoDir)
7              filhoDir->posOrdem();
8          aluno.imprime();
9      }
10
11     ...
12     Nodo * filhoEsq;
13     Nodo * filhoDir;
14     Aluno aluno;
15 };
```

Desta forma o percurso em pós-ordem da árvore T_b é

$$G, D, B, H, E, I, J, F, C, A$$

O percurso em ordem simétrica também é comumente utilizado. Neste visitamos o nodo entre as duas chamadas recursivas.

4 Árvores Binárias de Pesquisa

Uma árvore binária de pesquisa (ABP) usa um a chave para localizar o nodo dentro da árvore. Sejam v e v' dois nodos em uma árvore binária de pesquisa, e sejam $c(v)$ e $c(v')$ suas chaves. Se

- v' pertence à subárvore **esquerda** de v, então $c(v') < c(v)$, e se
- v' pertence à subárvore **direita** de v, então $c(v') > c(v)$.

A árvore binária da FIgura 4, **não é** uma ABP, pois em dois lugares ela não obedece as condições acima. A chave 6 está na subárvore da direita da chave 7, e a chave 9 está na subárvore da esquerda de 6.

Por outro lado a árvore T_{abp} da Figura 5 é uma ABP, pois é fato que para todos os nodos, os nodos em sua subárvore da esquerda tem chave **menor** que ele, e os nodos em sua subárvore da direita tem chave **maior** que ele.

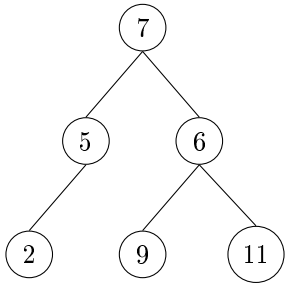

Figura 4: Não é uma árvore binária de pesquisa.

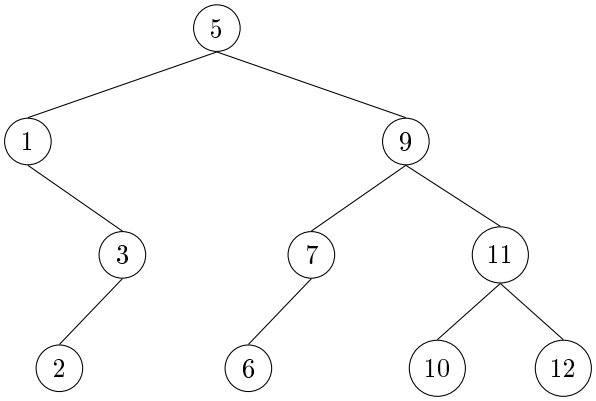

Figura 5: Árvore binária de pesquisa T_{abp}.

4.1 Operações em Árvores Binárias de Pesquisa

Uma ABP é um caso particular da árvore binária, mas para manter a condições de árvore binária de pesquisa temos que fazer a inserção de novos nodos de forma adequada. Isso vai nos ajudar depois quando quisermos pesquisar por uma chave dentro desta árvore.

4.1.1 Inserção

Na inserção iniciamos na raiz da árvore e seguimos sempre para a sub-árvore da esquerda ou da direita de acordo com a chave. Inserimos a nova chave sempre como uma folha.

```
1         void insere(const Aluno & aluno) {
2         Nodo * novoNodoPtr = new Nodo(aluno);
3         Nodo * ptrNodoAtual = raiz;
4
5         if( ! raiz) {
6             raiz = novoNodoPtr;
7             return;
8         }
```

```
9
10          while(true) {
11              if(ptrNodoAtual->aluno.chave < aluno.chave) {//direita do nodo atual
12                  if(ptrNodoAtual->filhoDir)//ainda nao chegamos
13                      ptrNodoAtual = ptrNodoAtual->filhoDir;
14                  else {//chegamos na folha
15                      ptrNodoAtual->filhoDir = novoNodoPtr;
16                      return;
17                  }
18              } else {//esquerda do nodo atual
19                  if(ptrNodoAtual->filhoEsq)//ainda nao chegamos
20                      ptrNodoAtual = ptrNodoAtual->filhoEsq;
21                  else {//chegamos na folha
22                      ptrNodoAtual->filhoEsq = novoNodoPtr;
23                      return;
24                  }
25              }
26          }
27      }
```

O *if* das linhas 5 à 8 testa o caso base, que ocorre quando a árvore está vazia. Neste caso o novo *Nodo* será sempre a raiz. Dentro do *while* das linhas 10 até 26 vamos precorrendo a árvore, armazenando um ponteiro para o nodo em que estamos na variável *ptrNodoAtual*. No momento que chegamos em um penteiro NULL inserimos o nodo *Nodo*. Os *if..else* das linhas 11 e 18 garante que vamos manter a propriedade de árvore binária de pesquisa.

Note que a ordem da inserção muda a ABP resultante. A inserção das chaves na ordem

$$5, 9, 7, 11, 6, 10, 12, 1, 3, 2$$

, por exemplo, gera a árvore T_{abp} da Figura 5.

Por outro lado a inserção

$$1, 9, 3, 2, 5, 7, 10, 11, 12, 8$$

gera árvore da Figura 6, apesar de serem exatamente as mesmas chaves.

4.1.2 Busca

Na busca usamos a propriedade de árvore binária de pesquisa. Começamos pela raiz e percorremos a árvore atrás chave.

```
1       Aluno busca(int chave) {
2           Nodo * ptrNodoAtual = raiz;
3
4           while(ptrNodoAtual) {
5               if(ptrNodoAtual->aluno.chave == chave)
6                   return ptrNodoAtual->aluno;
7               else if(ptrNodoAtual->aluno.chave < chave)
```

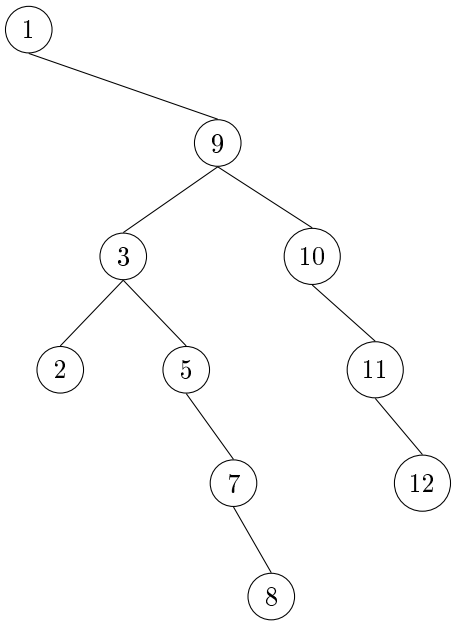

Figura 6: Outra árvore binária de pesquisa com os mesmos elementos de T_{abp}, mas inseridos em ordem diferentes.

```
8                   ptrNodoAtual = ptrNodoAtual->7filhoDir;
9               else
10                  ptrNodoAtual = ptrNodoAtual->filhoEsq;
11          }
12
13          return Aluno();
14      }
```

Armazenamos um ponteiro para o nodo atual em *ptrNodoAtual*. Dentro do *while* das linhas 4 até 11 vamos percorrendo a árvore. Caso o ponteiro *ptrNodoAtual* seja NULL, a chave não se encontra na árvore e retornamos um *Aluno* dummy. O *if...else...else* das linhas 5, 7 e 9 tratam se

- (*if* linha 5) encontramos a chave procurada, e podemos retornar o *Aluno*, se
- (*else* linha 7) a chave no nodo atual é **menor** que a chave procurada, e portanto vamos para a **subárvore da direita**, ou se
- (*else* linha 9) a chave no nodo atual é **maior** que a chave procurada, e portanto vamos para a **subárvore da esquerda**.

4.1.3 Gerência da Memória

Temos que desalocar adequadamente a memória alocada para a ABP. Para isso usamos o destrutor

```
1     ~ABP() {
2         if(raiz)
3             apaga(raiz);
4     }
```

que utiliza o método auxiliar

```
1     void apaga(Nodo * nodePtr) {
2         if(nodePtr->filhoEsq)
3             apaga(nodePtr->filhoEsq);
4         if(nodePtr->filhoDir)
5             apaga(nodePtr->filhoDir);
6         delete nodePtr;
7     }
```

Neste método fazemos o percorrimento de toda árvore, em ordem pós-fixada, apagando todos os nodos alocados. A escolha da ordem pós-fixada foi feita apenas para evitar o uso de variáveis auxiliares.