Ponteiros e árvores binárias

Kenji Yamane

Abril de 2020

Tópicos da aula

Alocação dinâmica

Para trabalhar com árvores binárias, precisa-se primeiramente aprofundar-se um pouco mais no assunto de ponteiros. Comentou-se sobre como ele pode ser usado para apontar para variáveis normais; por exemplo, se for necessário fazer um ponteiro p apontar para uma variável inteira x, basta fazer:

Código 1

```
int x = 5;
int *p = &x;
```

Pois o operador & retorna o endereço da variável x. Agora, é possível também criar uma "variável", ou seja um espaço de memória somente a partir de ponteiros, porém ele deve ser criado e deletado manualmente, conforme mostra o código do próximo slide, utilizando comandos de C++.

Código 2

```
#include <iostream>

int main() {
    int *p = new int;
    *p = 5;
    delete p;

    return 0;
}
```

Operador seta

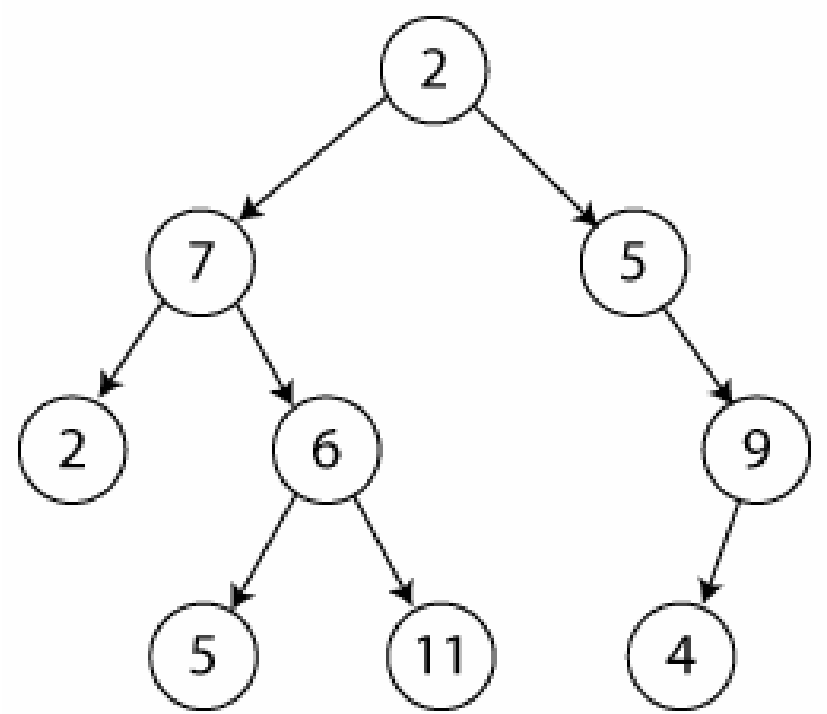
Além disso, também se trabalhará com ponteiros de struct. Devem se lembrar das aulas de treinamento de C que se acessa o que se tem dentro de uma variável de struct através do ponto (.). Se for um ponteiro para struct, as variáveis de dentro da struct são acessadas com seta (->), como mostra o código abaixo.

Código 3

```
#include <iostream>

struct par {
    int first;
    int second;
}

int main() {
    par *p = new par;
    p->first = p->second = 1;
    printf("%d %d\n", p->first, p->second);
    delete p;

    return 0;
}
```

O que é uma árvore binária

Agora pode-se iniciar os estudos de árvores binárias: compostas por um nó raiz, que aponta para dois filhos, um correspondente à uma "raiz" de uma sub-árvore esquerda, e um correspondente à uma "raiz" de uma sub-árvore direita. Cada nó pode ter no máximo dois filhos.



Em C++

Já é possível imaginar, dos termos usados no último slide, como se construiria uma árvore binária em C++: uma struct que possui três membros: uma informação, um ponteiro apontando para o filho esquerdo, e um ponteiro apontando para o filho direito. Para construir por exemplo uma árvore simples que tem uma raiz com o valor 2 e filho esquerdo com valor 1 e filho direito com valor 7, utilizaria-se do código 5.

Código 4

```
struct tree {
    int info;
    tree *left;
    tree *right;
}
```

Código 5


```
int main() {
    tree *rightSon = new tree;
    tree *leftSon = new tree;
    tree *root = new tree;

    rightSon->info = 7;
    rightSon->left = rightSon->right = NULL;
    leftSon->info = 1;
    leftSon->left = leftSon->right = NULL;
    root->info = 2;
    root->right = rightSon;
    root->left = leftSon;

    return 0;
}
```

Deletando a árvore

Perceba que não foi realizado o delete. Dado que foi criado com um ponteiro, o delete deve ser feito manualmente. Se ele não for, a memória continua sendo usada, o que se chama de memory leak. Admito que eu creio que a maioria dos juízes online não se importa com isso, mas é bom evitar e ajuda na didática da aula mostrar: como deletar uma árvore? Tem de ser uma ideia que funcione também para quando a árvore ficar bem complicada. Não se pode deletar a raiz antes de deletar os filhos senão se perde o acesso aos filhos. Isso se resolve recursivamente: caso se procure deletar uma árvore t, primeiro deleta-se a sub-árvore esquerda, depois a sub-árvore direita, e finalmente se deleta t. Não se deleta t se ele for NULL, que é o ponteiro que não aponta para nada, usado como ponto de parada nesse algoritmo.

Código 6


```
delete_tree (tree *t) {
    if (t != NULL) {
        delete_tree(t->left);
        delete_tree(t->right);
        delete t;
    }
}
```

Exploração de árvores

Serão expostas três formas de exploração de árvores aqui. Sabe-se que caso se queira explorar uma árvore t, é necessário explorar o nó t (usado para imprimir nesse caso), explorar a sub-árvore esquerda, e a sub-árvore direita. Há três formas possíveis de exploração(sub-árvore esquerda sempre antes da direita), a primeira sendo explorar o nó e depois os filhos, chamada de pré-ordem:

Código 7

```
void explore(tree *t) {
    if (t != NULL) {
        cout « t->info « endl;
        explore(t->left);
        explore(t->right);
    }
}
```

Exploração de árvores

O NULL é usado novamente como ponto de parada, e recursão é novamente usado. A outra forma se chama in-ordem, ou ordem simétrica. Nela, se explora primeiro a sub-árvore esquerda, depois a raiz, e então a sub-árvore direita.

Código 8

```
void explore(tree *t) {
    if (t != NULL) {
        explore(t->left);
        cout « t->info « endl;
        explore(t->right);
    }
}
```

Exploração de árvores

A terceira e última forma consiste em primeiro explorar os filhos e somente depois explorar a raiz, chamada pós-ordem. Novamente se utiliza recursão:

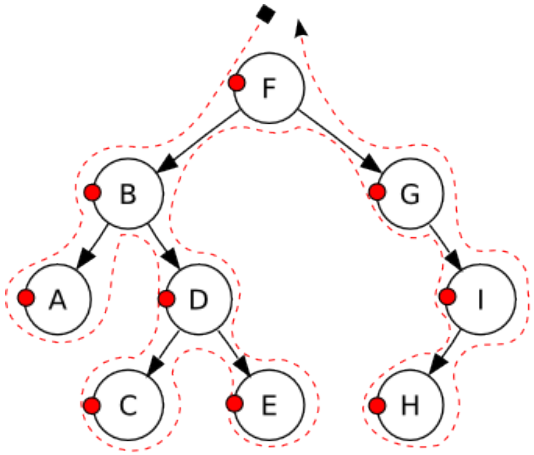
Código 9

```
void explore(tree *t) {
    if (t != NULL) {
        explore(t->left);
        explore(t->right);
        cout « t->info « endl;
    }
}
```

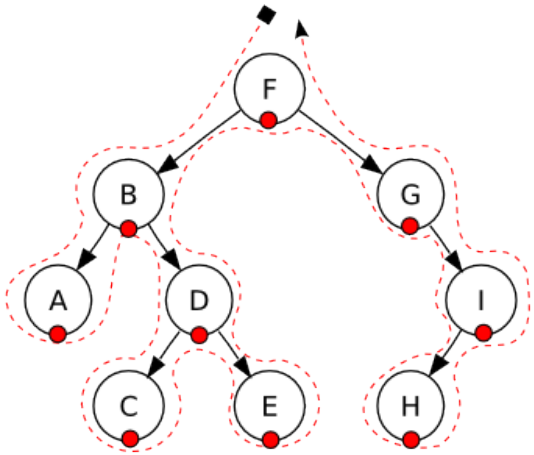
Perceba que deletar uma árvore é simplesmente um percurso pós-ordem, que se aproveita do fato de que na pós-ordem, primeiro os filhos são visitados.

Exploração de árvores

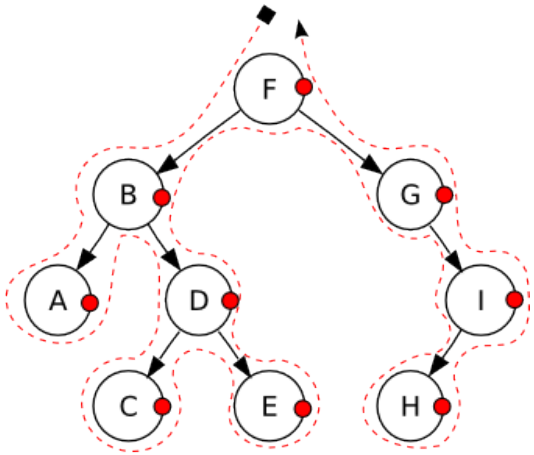
Uma representação gráfica dos três tipos de percurso estão mostradas a seguir (a linha tracejada é a exploração e ela explora o nó no ponto vermelho). Veja que as figuras correspondem aos algoritmos recursivos.



Pré-ordem: F, B, A, D, C, E, G, I, H



Ordem simétrica: A, B, C, D, E, F, G, H, I



Pós-ordem: A, C, E, D, B, H, I, G, F

O que são ABP's

Árvores Binárias de Pesquisa são árvores binárias especiais no sentido de que: todos os números que estão na sub-árvore esquerda de uma raiz são menores do que a raiz, e todos os números da sub-árvore direita da raiz são maiores que ela. Qualquer sequência de números pode gerar uma ABP, lendo número por número, e inserindo cada um na árvore. A forma de inserção leva em conta a característica da ABP e usa recursão também, como mostrado no próximo slide.

Código 10


```
tree *insert(tree *t, int value) {
    if (t == NULL) {
        t = new tree;
        t->info = value;
        t->left = t->right = NULL;
    }
    else if (value < t->info)
        t->left = insert(t->left, value);
    else
        t->right = insert(t->right, value);

    return t;
}
```

Propriedades de ABP's

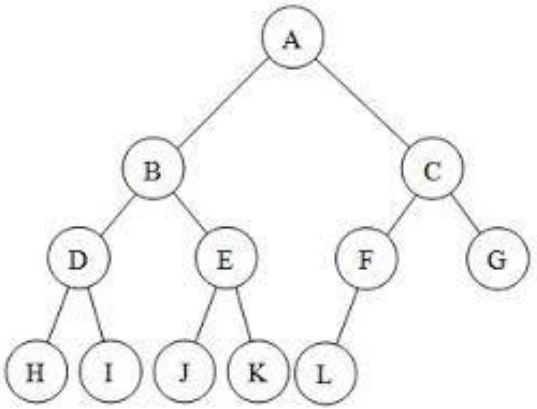
Os benefícios principais das ABP's são dois: o percurso infixo deles gera a sequência ordenada (se a sub-árvore esquerda é impressa antes da raiz e a direita depois, a raiz está na posição certa, e isso é válido recursivamente para qualquer nó). O outro, o mais importante, é que a busca nessa estrutura de dados é bem otimizada, em logN:

Código 11

```
bool search(tree *t, value) {
    if (t == NULL) return false;
    else if (value < t->value)
        return search(t->left, value);
    else if (value > t->value)
        return search(t->right, value);
    else return true;
}
```

Forma alternativa

Há ainda uma outra forma de se representar árvores binárias, mas um tipo especial de árvores binárias: completas. Elas são tais que todos os nós tem 2 ou 0 filhos e todos os níveis exceto possivelmente o último estão preenchidos, e no último nível, todos os nós estão para o mais à esquerda possível.



Forma alternativa

Arrays são uma forma cômoda de se representar esse tipo de árvore binária pois, utilizando a posição 1 como raiz, e considerando que 2*i e 2*i + 1 serão os filhos esquerdo e direito de 1, pode-se representar a árvore binária, sem buracos no array, se for completa. É possível acessar o pai também simplesmente dividindo a posição por 2. Esse tipo de representação está sendo comentado aqui pois na última aula serão estudadas dois tipos de árvores binárias e elas serão representadas dessa forma.

Problemas de árvores binárias

- Árvore Binária de Busca
- Operações em ABP I
- Operações em ABP II
- Recuperação da Árvore
- Percurso em Árvore por Nível