Algoritmos e Estrutura de Dados

Árvores Binárias de Busca

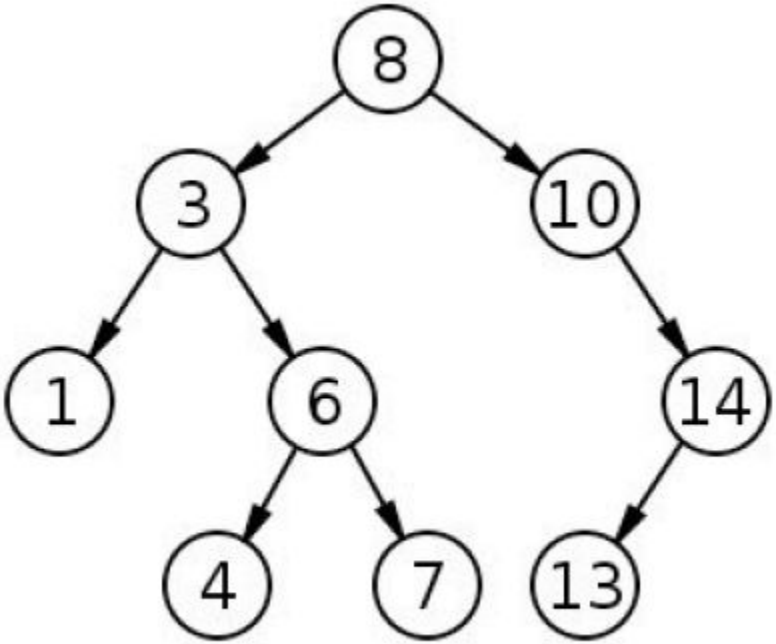
Baseado no material dos Prof:
Wesley Romão

Prof. Nilton Luiz Queiroz Jr.

Árvores binárias de busca

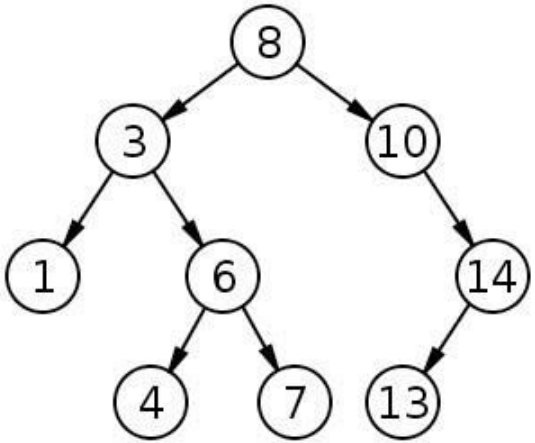
- Uma árvore binária de busca (ABB) obedece as seguintes propriedades:
  - Todo nó contém um registro com uma chave;
  - Todo nó que contém filhos obedece duas propriedades:
    - Nós com chaves menores estão na subárvore esquerda;
    - Nós com chaves maiores estão na subárvore direita;

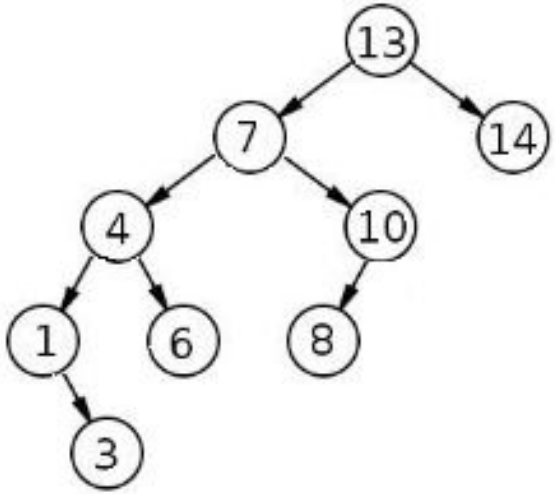
Árvores binárias de busca



Árvores binárias de busca

- Uma ABB não é única
  - A árvore final depende da **ordem** que os dados foram inseridos;





Árvores binárias de busca

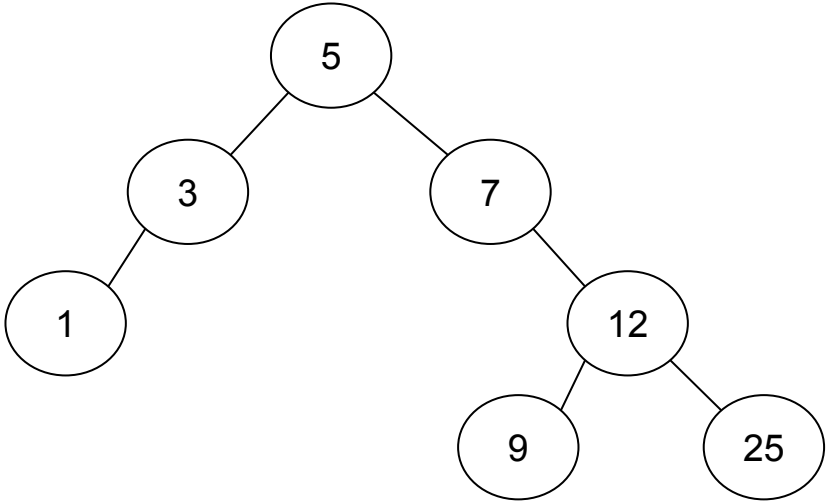
- Como construir uma ABB com a seguinte sequência de números:

  5, 3, 1, 7, 12, 25, 9

- Siga os passos:
  - Inicialmente coloque 5 na raiz;
  - O elemento seguinte, 3, será comparado com 5, como 3 é menor que 5 ele será inserido na subárvore esquerda de 5, que é uma árvore vazia;
  - Comparamos então 1 com a raiz, e como ele é menor, devemos inseri-lo na subárvore esquerda de 5, que é a árvore de raiz 3, dessa forma a mesma comparação deve ser feita com 3;
  - Em seguida é feita a inserção do elemento 7, e como ele é maior que 5, será inserido na subárvore direita de 5, que é uma subárvore vazia;
  - O mesmo deve ser repetido com os outros elementos;

Árvores binárias de busca

- O resultado seria a árvore:



Exercício

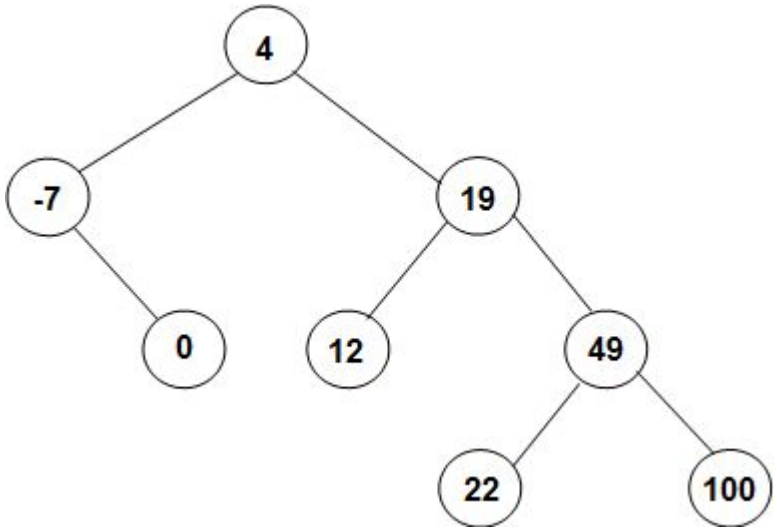
- Desenhe a árvore de busca para os seguintes números:

4 19 -7 49 100 0 22 12

Exercício

- Desenhe a árvore de busca para os seguintes números:

4 19 -7 49 100 0 22 12



Aplicação

- ABBs são úteis para casos onde se fazem consultas com frequência
  - Ocorrem diversas buscas;
- Em geral as ABBs são implementadas dinamicamente, e podem sofrer alterações;
  - Inserção de valores;
  - Remoção de valores;

Implementação dinâmica

- ABBs implementadas dinamicamente tem as seguintes vantagens:
  - Não há desperdício de memória;
  - Não há limite de nós a serem inseridos;
  - Facilmente gerenciável;
- E as seguintes desvantagens:
  - Utiliza estruturas de dados mais complexas;
  - Deve ser balanceada para ser otimizada;
    - Árvores AVL;

Implementação

- A implementação de uma árvore binária de busca requer o uso de uma estrutura para cada nó e a estrutura da árvore;

```
struct tipo_item{
    int chave;
};
struct tipo_no{
    struct tipo_item dado;
    struct tipo_no *esq;
    struct tipo_no *dir;
};
struct tipo_arvore{
    struct tipo_no *raiz;
};
```

Implementação

- Inicializar árvore:

```
void inicializa(struct tipo_arvore *a){
    a->raiz = NULL;
}
```

- Ver se a árvore é vazia;

```
int vazia(struct tipo_arvore *a){
    return a->raiz == NULL;
}
```

Inserção

- Inserir elemento:
  - Sempre que um nó é inserido ele é inserido como nó folha, ou seja no lugar de uma subárvore vazia;
    - Quando a árvore não é vazia procura-se uma subárvore vazia e se insere nela;
    - Nessa busca leva-se em consideração as seguintes questões:
      - Se a chave do valor inserido for menor que a chave do nó atual faz a inserção na subárvore esquerda;
      - Caso contrário faz a inserção na subárvore direita;
- A inserção numa árvore com n elementos tem complexidade:
  - O(n) no pior caso;
  - O($\log_2$ n) no melhor caso;

Inserção

- A implementação da inserção pode ser feita da seguinte maneira:
  - É interessante a utilização de uma função para auxiliar a implementação da inserção em uma ABB;
    - Uma função apenas para fazer a chamada da função recursiva para a raiz;
  - Além disso, para simplificar, iremos adotar a função para criar nós;

```
struct tipo_no *cria_no(struct tipo_item x){
    struct tipo_no *aux;
    aux=(struct tipo_no *)malloc(sizeof(struct tipo_no));
    aux->dir=NULL;
    aux->esq=NULL;
    aux->dado=x;
    return aux;
}
```

Inserção

- Inserção:

```
void insere(struct tipo_arvore *a, struct tipo_item x){
    a->raiz = inserir_no(a->raiz,x);
}
```

- Como implementar a função inserir nó?

Inserção

- Inserir nó

```
struct tipo_no *inserir_no(struct tipo_no *n, struct tipo_item x){
    if(n==NULL){
        n=cria_no(x);
    }else if(n->dado.chave < x.chave){
        n->dir=inserir_no(n->dir,x);
    }else if(n->dado.chave > x.chave){
        n->esq=inserir_no(n->esq,x);
    }
    return n;
}
```

Inserção

- Uma outra alternativa seria a inserção com retorno indicando se foi ou não possível inserir um item

```
int insere(struct tipo_arvore *a, struct tipo_item x){
    return insere_no(&(a->raiz),x);
}
int insere_no(struct tipo_no **n, struct tipo_item x){
    /*algoritmo*/
}
```

Inserção

- Uma outra alternativa seria a inserção com retorno indicando se foi ou não possível inserir um item

```
int insere(struct tipo_arvore *a, struct tipo_item x){
    return insere_no(&(a->raiz),x);
}
int insere_no(struct tipo_no **n, struct tipo_item x){
    if(*n==NULL){
        *n=cria_no(x);
        return 1;
    }else if( (*n)->dado.chave < x.chave ){
        return insere_no(&((*n)->dir),x);
    }else if( (*n)->dado.chave > x.chave){
        return insere_no(&((*n)->esq),x);
    }else{
        return 0;
    }
}
```

Busca

- Busca:
  - Se a árvore estiver vazia retorna-se um ponteiro nulo;
  - Se a chave buscada for encontrada retorna-se o ponteiro para o nó que contém aquele valor;
  - Se a chave do nó for maior que a chave buscada retorna-se o resultado da busca na subárvore esquerda;
  - Caso nenhuma condição anterior tenha sido satisfeita retorna-se o resultado da busca na subárvore direita;
- A busca numa árvore com n elementos tem complexidade;
  - O(n) no pior caso;
  - O($\log_2$ n) no melhor caso;

Busca

- A implementação da busca será feita de maneira semelhante a da inserção;
  - Uma função auxiliar para fazer a chamada da função de busca dos nós no nó raiz;
- Busca:

```
struct tipo_no *busca(struct tipo_arvore *a, int chave){
    return busca_no(a->raiz,chave);
}
```

- Como implementar a função busca nó?

Busca

- Busca nó

```
struct tipo_no *busca_no(struct tipo_no *n, int ch){
    /*algoritmo*/
}
```

Implementação

- Busca nó

```
struct tipo_no *busca_no(struct tipo_no *n, int ch){
    if(n->dado.chave == ch){
        return n;
    }else if(n->dado.chave > ch){
        return busca_no(n->esq,ch);
    }else if(n->dado.chave < ch){
        return busca_no(n->dir,ch);
    }else{
        return NULL;
    }
}
```

Exercícios

1. Faça um algoritmo para encontrar o maior elemento em uma ABB:
   a. Recursivo.
   b. Iterativo.
2. Faça um algoritmo para encontrar o menor elemento em uma ABB:
   a. Recursivo;
   b. Iterativo;
3. Implemente a função iterativa de inserção em uma ABB;
4. Implemente a função iterativa de busca em uma ABB.
5. Implemente uma função que receba uma ABB e retorne um vetor ordenado contendo as chaves da ABB;

Exercícios

6. Para o seguinte conjunto de elementos, e dada uma raiz fixa, desenhe a respectiva árvore binária de busca:

   {10, 22, 36, 5, 33, 27, 40, 4, 7, 20, 23, 35, 37,42}

   a. raiz: 24;
   b. raiz:12;

7. Faça a inserção dos elementos a seguir em uma árvore binária de busca e responda:

   {4, 8, 12, 13, 18, 22, 27}

   a. Qual a altura da árvore?
   b. Qual o problema dessa nessa árvore?

Referências

DROZDEK, A. Estrutura de Dados e Algoritmos em C++. São Paulo, SP, Brasil: Thomson, 2005. 579 p. ISBN 85-221-0259-3.