Estrutura de Dados

Árvore Binária de Busca

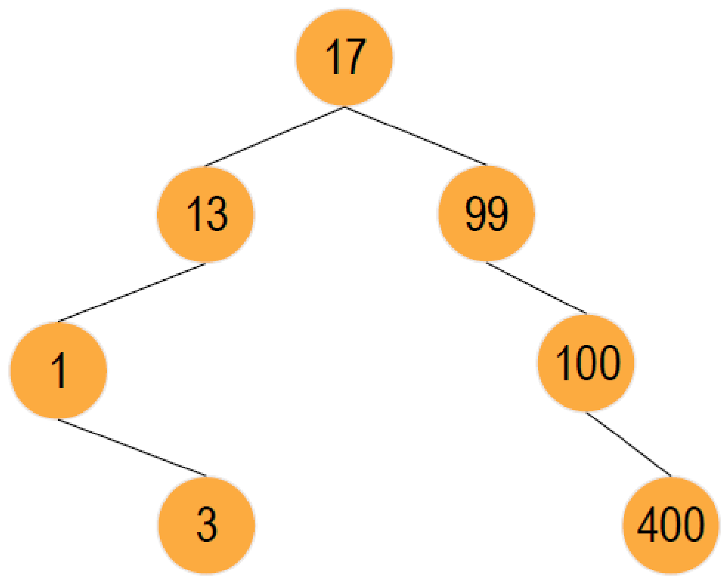
- O processo de busca de informação em uma árvore binária como vimos até agora é caro.
  - Se as chaves não estão em uma ordem pré-estabelecida, toda a estrutura precisa ser percorrida para encontrar uma determinada chave (no pior caso),o que não seria eficiente

- Veremos uma forma de melhorar o tempo de busca, utilizando Árvores Binárias de Busca

Árvore Binária de Busca

Uma Árvore Binária de Busca possui as mesmas propriedades de uma AB, acrescida da seguinte propriedade: Para todo nó da árvore, se seu valor é X, então:

- Os nós pertencentes a sua sub-árvore esquerda possuem valores menores do que X;
- Os nós pertencentes a sua sub-árvore direita possuem valores maiores do que X.
- Um percurso em in-ordem nessa árvore resulta na sequência de valores em ordem crescente



Características

- Se invertêssemos as propriedades descritas na definição anterior, de maneira que a sub-árvore esquerda de um nó contivesse valores maiores e a sub-árvore direita valores menores, o percurso in-ordem resultaria nos valores em ordem decrescente

- Uma árvore de busca criada a partir de um conjunto de valores não é única: o resultado depende da ordem de inserção dos dados

- A grande utilidade da árvore binária de busca é armazenar dados contra os quais outros dados são frequentemente verificados (busca!)

- Uma árvore binária de busca é armazenada dinamicamente e em geral sofre alterações (inserções e remoções de nós) após ter sido criada

Listas X ABB

- O tempo de busca é estimado pelo número de comparações entre chaves.

- Em listas de n elementos, temos:
  - Sequenciais (Array): O(n) se não ordenadas; ou $O(\log_2 n)$, se ordenadas
  - Encadeadas (Dinâmicas): O(n)

- As ABB constituem a alternativa que combina as vantagens de ambos: são dinâmicas e permitem a busca binária $O(\log_2 n)$

Operações em ABB

- Busca
- Inserção –*mantendo as propriedades de ABB*
- Remoção –*mantendo as propriedades de ABB*

Busca

Passos do algoritmo de busca:

- Se a árvore é vazia, fim e não achou. Se chave da raiz é igual à chave procurada, termina busca com sucesso.
- Senão: Repita o processo para a sub-árvore esquerda, se chave da raiz é maior que a chave procurada; se for menor, repita o processo para a sub-árvore direita.
- Caso o nó contendo o valor pesquisado seja encontrado, retorne um ponteiro para o nó; caso contrário, se se deparar com uma árvore vazia, retorne um ponteiro nulo.

Busca (recursiva)

```
No * busca(No *t, tipo_elem chave){
    if(t == NULL)
        return NULL;

    if(t->info == chave)
        return(t);

    if(t->info<chave)
        return(busca(t->esq, chave));

    else
        return(busca(t->dir, chave));
}
```

Exercício: fazer a versão não-recursiva

Recursiva versus Não-Recursiva

- Em relação a tempo (número de comparações): equivalentes
- Em relação a espaço (memória): a recursiva requer espaço extra para as chamadas recursivas. Esse espaço extra é diretamente proporcional (linear) à altura da árvore. (VERIFIQUE!)

Custo da busca em ABB

- **Pior caso**: como o número de passos é determinado pela altura da árvore, o pior caso é a árvore degenerada (altura = n).
- Altura da ABB depende da sequência de inserção das chaves...
  - Considere, p.ex., o que acontece se uma sequência ordenada de chaves é inserida...
  - Seria possível gerar uma árvore balanceada com essa mesma sequência, se ela fosse conhecida a priori. Como?
- **Busca ótima**: árvore de altura mínima (perfeitamente balanceada)
- **Busca eficiente**: árvore razoavelmente balanceada...(árvore balanceada)

Inserção em ABB

PASSOS DO ALGORITMO DE INSERÇÃO:

- Procure um “local” para inserir a nova chave, começando a procura a partir do nó-raiz:

- Para cada nó-raiz, compare:
  – se a nova chave for menor do que o valor no nó-raiz, repita o processo para sub-árvore esquerda; ou
  – se a nova chave for maior que o valor no nó-raiz, repita o processo para sub-árvore direita.

- Se um ponteiro (filho esquerdo/direito de um nó-raiz) nulo é atingido, coloque o novo nó como sendo raiz dessa sub-árvore vazia.

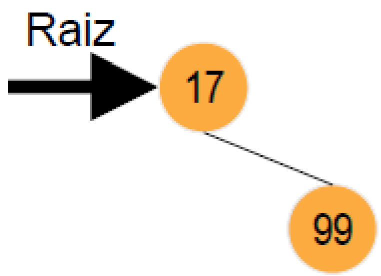
- **A inserção sempre se dá como nó folha: não exige deslocamentos!**

Inserção - Exemplo

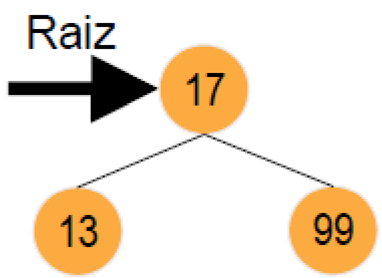
- Para entender o algoritmo, considere a inserção do conjunto de números, na sequência

{17, 99, 13, 1, 3, 100, 400}

No início, a ABB está vazia!

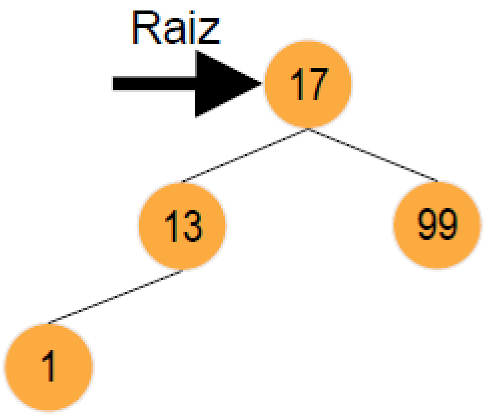


O número 17 será inserido tornando-se o nó raiz
- A inserção do 99 inicia-se na raiz. Compara-se 99 c/ 17.
- Como 99 > 17, 99 deve ser colocado na sub-árvore direita do nó contendo 17 (subárvore direita, inicialmente, nula)

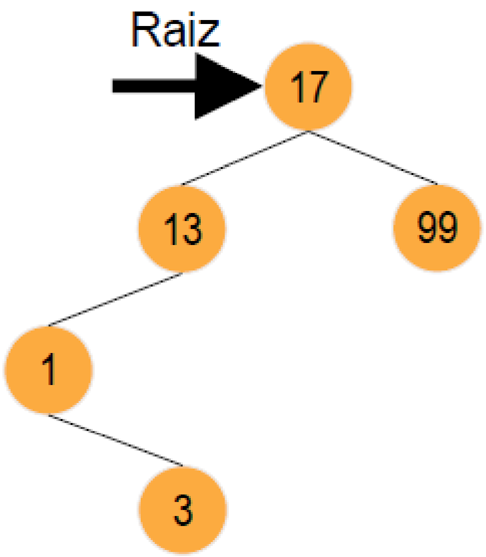


A inserção do 13 inicia-se na raiz
-Compara-se 13 c/ 17. Como 13 < 17, 13 deve ser colocado na sub-árvore esquerda do nó contendo 17
-Já que o nó 17 não possui descendente esquerdo, 13 é inserido como raiz dessa sub-árvore

Inserção - continuação



Repete-se o procedimento para inserir o valor 1
- 1<17, então será inserido na sub-árvore esquerda
- Chegando nela, encontra-se o nó 13, 1<13 então ele será inserido na sub-árvore esquerda de 13



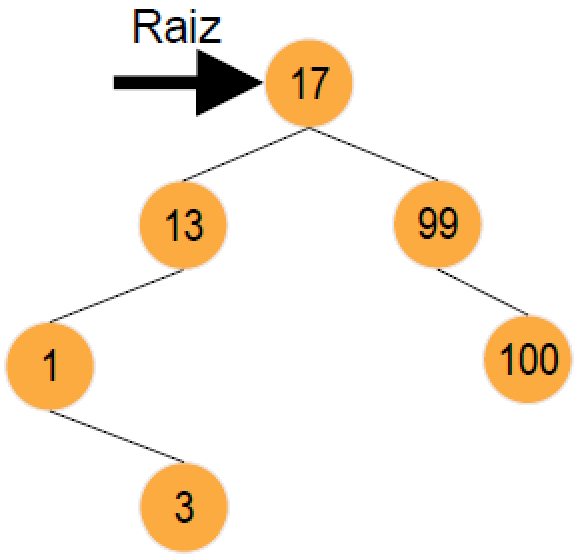
Repete-se o procedimento para inserir o elemento 3:

-3 < 17;

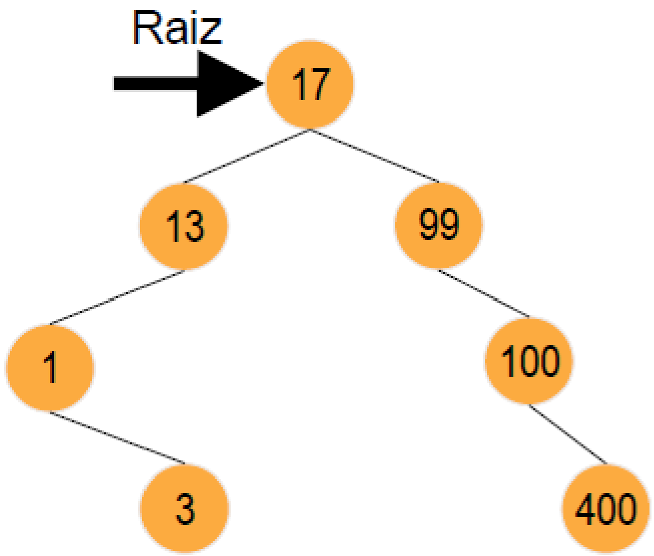
-3 < 13

-3 > 1

Inserção - continuação



Repete-se o procedimento para inserir o elemento 100:
- 100 > 17
- 100 > 99



Repete-se o procedimento para inserir o elemento 400:
- 400 > 17
- 400 > 99
- 400 > 100

Inserção

- O algoritmo de inserção não garante que a árvore resultante seja perfeitamente balanceada ou mesmo apenas balanceada.

- A árvore do exemplo anterior não é perfeitamente balanceada nem balanceada.

FUNÇÃO RECURSIVA DA INSERÇÃO

- Seja uma função recursiva que insere uma chave x numa ABB, se ela já não estiver lá. Retorna o ponteiro para o nó que contém x.

Inserção

```
int insere(Arvore *A, tipo_elem v){
   return(insere_abb(&A->raiz, v));
}

int insere_abb(No **t, tipo_elem v){
    if(*t == NULL){
        *t = (No *) malloc (sizeof(No));
        (*t)->info = v;
        (*t)->esq = (*t)->dir = NULL;
        return 1;
    }
    if((*t)->info > v)
      return(insere_abb(&(*t)->esq, v));
    if((*t)->info < v)
        return(insere_abb(&(*t)->dir, v));

    return 0;
}
```

Custo da operação de Inserção

- A inserção requer uma busca pelo lugar da chave, portanto, com custo de uma busca qualquer (tempo proporcional à altura da árvore).
- O custo da inserção, após a localização do lugar, é constante; não depende do número de nós.
- Logo, tem complexidade análoga à da busca.

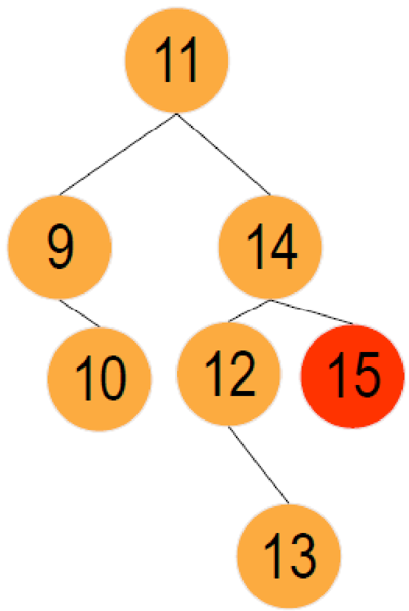
Remoção

Casos a serem considerados no algoritmo de remoção de nós de uma ABB:

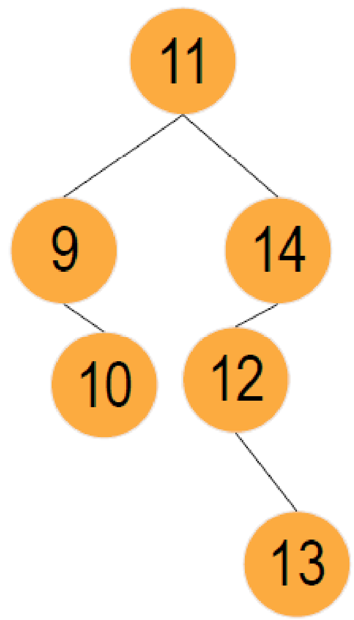
- **Caso 1**:o nó é folha
  - O nó pode ser retirado sem problema;
- **Caso 2**:o nó possui uma sub-árvore (esq./dir.)
  - O nó-raiz da sub-árvore (esq./dir.) pode substituir o nó eliminado;
- **Caso 3**:o nó possui duas sub-árvores
  - O nó cuja chave seja a menor da sub-árvore direita pode substituir o nó eliminado; ou, alternativamente, o de maior valor da sub-árvore esquerda pode substituí-lo.

Remoção – Caso 1
Nó folha

-Caso o valor a ser removido seja o 15
- pode ser removido sem problema, não requer ajustes posteriores

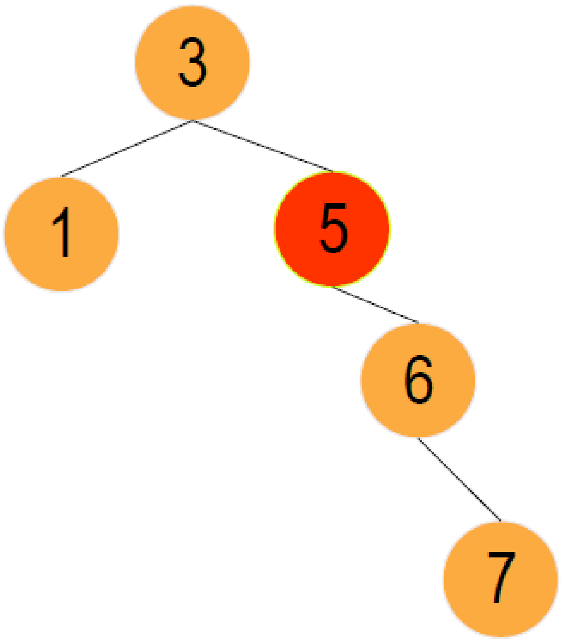


- Os nós com os valores 10 e 13 também podem ser removidos!

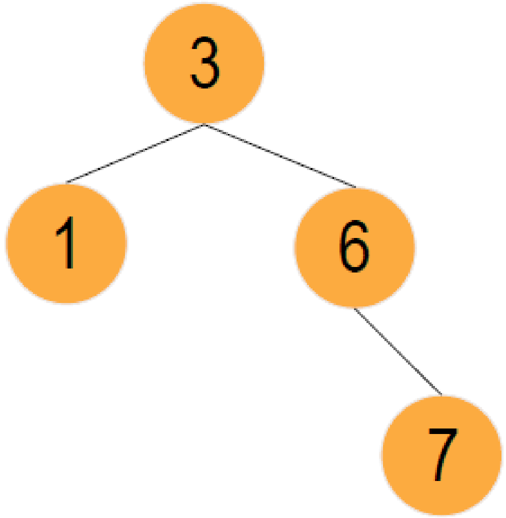


Remoção – Caso 2
Nó possui 1 sub-árvore

- Removendo-se o nó com o valor 5
- Como ele possui apenas uma sub-árvore direita, seu nó filho, contendo o valor 6, pode "ocupar" o lugar do nó removido

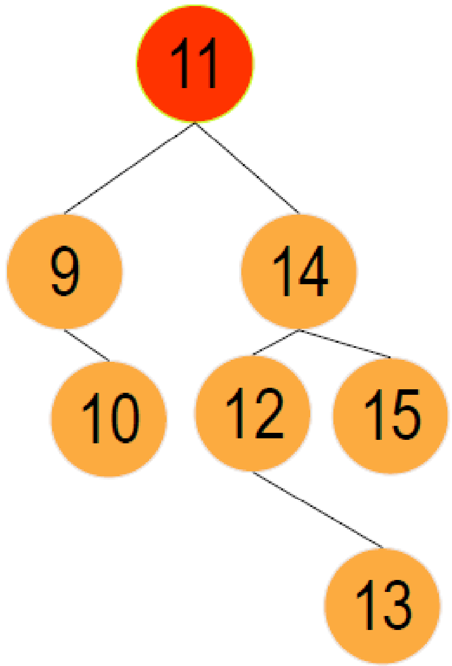


Caso existisse um nó com somente uma sub-árvore esquerda, seria análogo.



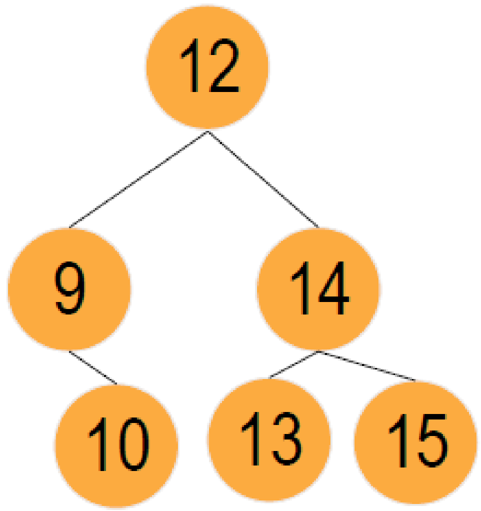
Remoção – Caso 3
Nó possui 2 sub-árvores

- Eliminando-se o nó de chave 11
- Neste caso, existem 2 opções:
  - A chave 10 pode "ocupar" o lugar do nó-raiz, ou
  - A chave 12 pode "ocupar" o lugar do nó-raiz
  - O nó cuja chave substituiu a raiz é removido



Esse terceiro caso, também se aplica ao nó com chave 14, caso seja retirado.

-Nessa configuração, os nós com chave 13 ou 15 poderiam ocupar seu lugar.



Remoção

```
int remove_abb(No **t, tipo_elem v){
    No *aux;

    if(*t==NULL) // não achou - não remove
        return 0;
    else if(v > (*t)->info)
        return(remove_abb(&(*t)->dir, v));
    else if(v < (*t)->info)
        return (remove_abb(&(*t)->esq, v));
    else{ // remove
        //Caso 1: Nó folha - não tem filhos
        if((*t)->esq==NULL && (*t)->dir==NULL){
            free(*t);
            *t = NULL;
            return 1;
        }
/* ... */
```

```c
 // Caso 2a: Só tem filho direito
        else if((*t)->esq == NULL){
            aux = *t;
            *t = (*t)->dir; // faz ligação com o filho a direita
            free(aux);
            return 1;
        }

        // Caso 2b: Só tem filho esquerdo
        else if((*t)->dir == NULL){
            aux = *t;
            *t = (*t)->esq; // faz ligação com o filho a esquerda
            free(aux);
            return 1;
        }

/* ... */
```

```
 // Caso 3: Tem 2 filhos
        else{ //substitui pelo maior da subárvore esquerda
            (*t)->info = busca_maior((*t)->esq);
            return(remove_abb(&(*t)->esq, (*t)->info));
        }
    }
}// fim remove_abb
```

```
tipo_elem busca_maior(No *p) {
    while(p->dir!=NULL)
        p=p->dir;
    return(p->info);
}
```

```
int remove(Arvore *A, tipo_elem v){
    return(remove_abb(&A->raiz, v));
}
```

Custo da Operação de Remoção

- A remoção requer uma busca pela chave do nó a ser removido, portanto, com custo de uma busca qualquer (tempo proporcional à altura da árvore).

- O custo da remoção, após a localização do nó dependerá de 2 fatores:
  - do caso em que se enquadra a remoção: se o nó tem 0, 1 ou 2 sub-árvores; se 0 ou 1 filho, custo é constante.
  - de sua posição na árvore, caso tenha 2 sub-árvores (quanto mais próximo do último nível, menor esse custo)

- Repare que um maior custo na busca implica num menor custo na remoção propriamente dita; e vice-versa.

- Logo, <u>tem complexidade dependente da altura da árvore</u>.

Consequências das operações de inserção e eliminação

- Uma ABB balanceada ou perfeitamente balanceada tem a organização ideal para buscas.

- Inserções e eliminações podem desbalancear uma ABB, tornando futuras buscas ineficientes.

- Possível solução:
  - Construir uma ABB inicialmente perfeitamente balanceada
  - após várias inserções/eliminações, aplicamos um processo de rebalanceamento

ABB: Resumo

- Boa opção como ED para aplicações de pesquisa (busca) de chaves, SE árvore balanceada: O(log2n)

- Inserções (como folhas) e Eliminações (mais complexas) causam desbalanceamento.

- Inserções: melhor se em ordem aleatória de chaves, para evitar linearização (se ordenadas)

- Para manter o balanceamento, veremos as Árvores AVL