Árvores Binárias de Busca

SCC0202 - Algoritmos e Estruturas de Dados I

Prof. Fernando V. Paulovich
**Baseado no material do Prof. Gustavo Batista*
http://www.icmc.usp.br/~paulovic
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

7 de novembro de 2017

Sumário

Sumário

Definições

- Uma **Árvore Binária de Busca (ABB)** possui as seguintes propriedades

Definições

- Uma **Árvore Binária de Busca (ABB)** possui as seguintes propriedades
  - Seja $S = \{s_1, \ldots, s_n\}$ o conjunto de chaves dos nós da árvore T
    - Esse conjunto satisfaz $s_1 < \ldots < s_n$
    - A cada nó $v_j \in T$ está associada uma chave distinta $s_j \in S$, que pode ser consultada por $r(v_j) = s_j$

Definições

- Uma **Árvore Binária de Busca (ABB)** possui as seguintes propriedades
  - Seja $S = \{s_1, \ldots, s_n\}$ o conjunto de chaves dos nós da árvore T
    - Esse conjunto satisfaz $s_1 < \ldots < s_n$
    - A cada nó $v_j \in T$ está associada uma chave distinta $s_j \in S$, que pode ser consultada por $r(v_j) = s_j$
  - Dado um nó v de T
    - Se v_i pertence a sub-árvore esquerda de v, então $r(v_i) < r(v)$
    - Se v_i pertence a sub-árvore direita de v, então $r(v_i) > r(v)$

Definições

- Em outras palavras
  - Os nós pertencentes à sub-árvore esquerda possuem valores menores do que o valor associado ao nó-raiz r
  - Os nós pertencentes à sub-árvore direita possuem valores maiores do que o valor associado ao nó-raiz r

Definições

- Um **percurso em-ordem** em uma ABB resulta na sequência de valores em **ordem crescente**
- **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 em-ordem resultaria nos valores em **ordem decrescente**
- **Uma ABB** criada a partir de um conjunto de valores **não é única**: o resultado depende da sequência de inserção dos dados

Definições

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

- Uma árvore de binária de busca é dinâmica e pode sofrer alterações (inserções e remoções de nós) após ter sido criada

Operações em ABB's

- Inserção
- Pesquisa
- Remoção

Sumário

Inserção (operações em ABB's)

- Passos do algoritmo de inserção
  - Procure um "local" para inserir o novo nó, começando a procura a partir do nó-raiz
  - Para cada nó-raiz de uma sub-árvore, compare; se o novo nó possui um valor menor do que o valor no nó-raiz (vai para sub-árvore esquerda), ou se o valor é maior que o valor no nó-raiz (vai para sub-árvore direita)
  - Se um ponteiro (filho esquerdo/direito de um nó-raiz) nulo é atingido, coloque o novo nó como sendo filho do nó-raiz

Inserção

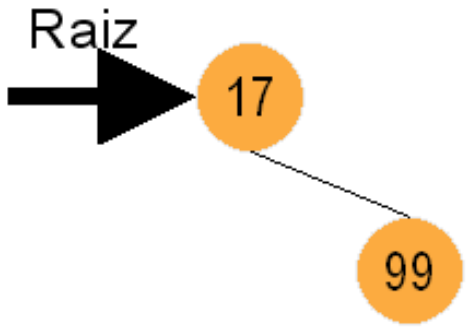
- Para entender o algoritmo considere a inserção do conjunto de números, na sequência

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

- No início a ABB está vazia!

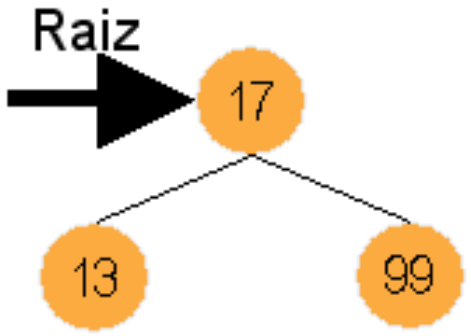
Inserção

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



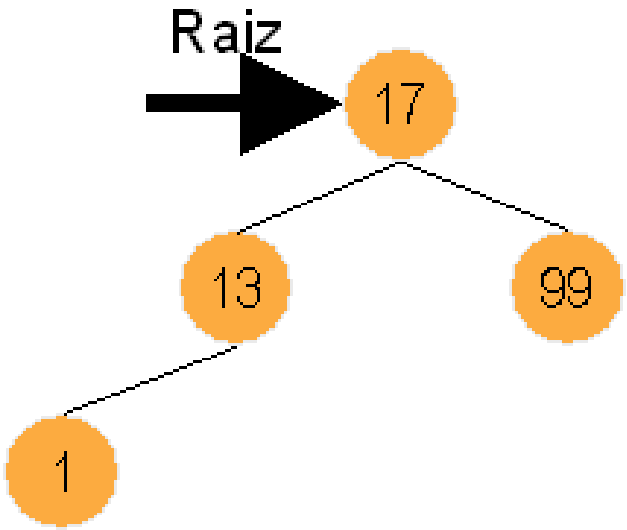
Inserção

- A inserção do 13 inicia-se na raiz
- Compara-se 13 com 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 na árvore nessa posição



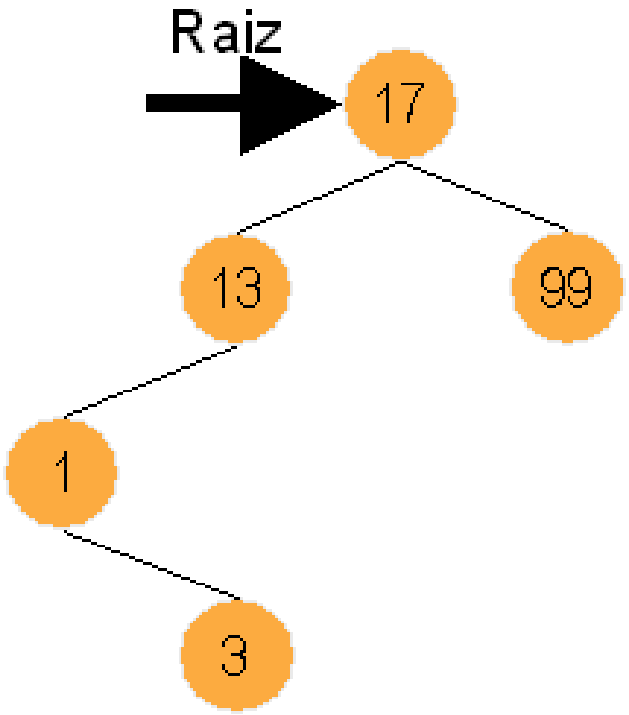
Inserção

- 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



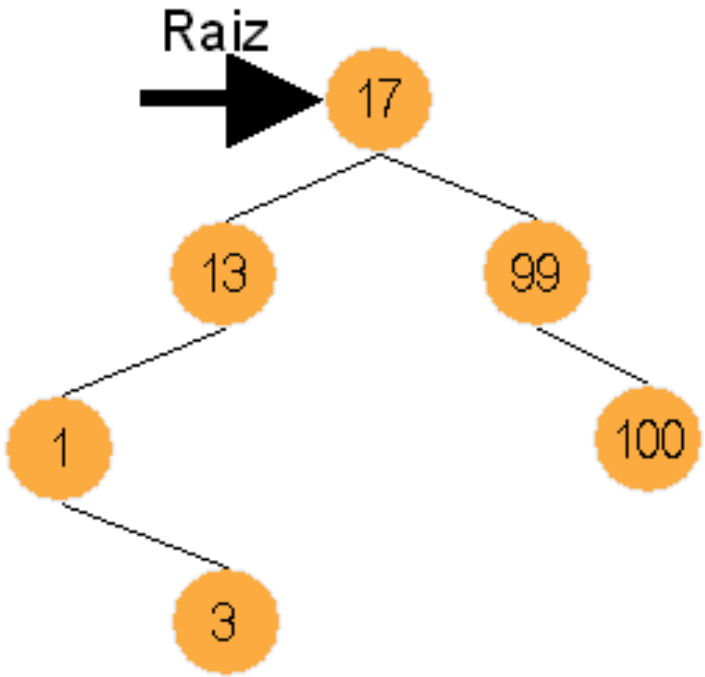
Inserção

- Repete-se o procedimento para inserir o elemento 3
  - $3 < 17$
  - $3 < 13$
  - $3 > 1$



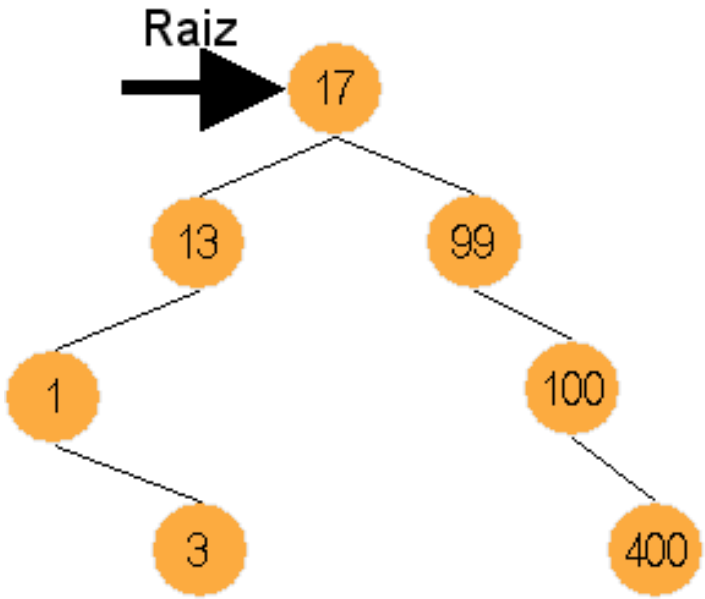
Inserção

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



Inserção

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



Relembrando

```
1  typedef struct arvore_binaria ARVORE_BINARIA;
2  typedef struct no NO;
3
4  struct no {
5    ITEM *item;
6    NO *filhoesq;
7    NO *filhodir;
8  };
9
10 struct arvore_binaria {
11   NO *raiz;
12 };
```

Relembrando

```
1  ARVORE_BINARIA *criar_arvore() {
2    ARVORE_BINARIA *arv = (ARVORE_BINARIA *)malloc(sizeof(ARVORE_BINARIA↩
         ));
3
4    if (arv != NULL) {
5      arv->raiz = NULL;
6    }
7
8    return arv;
9  }
10
11 NO *criar_raiz(ARVORE_BINARIA *arvore, ITEM *item) {
12   arvore->raiz = (NO *) malloc(sizeof (NO));
13
14   if (arvore->raiz != NULL) {
15     arvore->raiz->filhodir = NULL;
16     arvore->raiz->filhoesq = NULL;
17     arvore->raiz->item = item;
18   }
19
20   return arvore->raiz;
21 }
```

Relembrando

```
1  #define FILHO_ESQ 0
2  #define FILHO_DIR 1
3
4  NO *inserir_filho(int filho, NO *no, ITEM *item) {
5    NO *pnovo = (NO *) malloc(sizeof (NO));
6
7    if (pnovo != NULL) {
8      pnovo->filhodir = NULL;
9      pnovo->filhoesq = NULL;
10     pnovo->item = item;
11
12     if (filho == FILHO_ESQ) {
13       no->filhoesq = pnovo;
14     } else {
15       no->filhodir = pnovo;
16     }
17   }
18
19   return pnovo;
20 }
```

Código

```
1  int inserir_aux(NO *raiz, ITEM *item) {
2    if (raiz->item->chave > item->chave) {
3      if (raiz->filhoesq != NULL) {
4        return inserir_aux(raiz->filhoesq, item);
5      } else {
6        return (inserir_filho(FILHO_ESQ, raiz, item) != NULL);
7      }
8    } else if (raiz->item->chave < item->chave) {
9      if (raiz->filhodir != NULL) {
10       return inserir_aux(raiz->filhodir, item);
11     } else {
12       return (inserir_filho(FILHO_DIR, raiz, item) != NULL);
13     }
14   } else {
15     return 0;
16   }
17 }
18
19 int inserir(ARVORE_BINARIA *arvore, ITEM *item) {
20   if (vazia(arvore)) {
21     return (criar_raiz(arvore, item) != NULL);
22   } else {
23     return inserir_aux(arvore->raiz, item);
24   }
25 }
```

Exercícios

- Criar um método iterativo para inserção em ABB

Sumário

Pesquisa (operações em ABB's)

- Passos do algoritmo de busca
  - Comece a busca a partir do nó-raiz
  - Para cada nó-raiz de uma sub-árvore compare: se o valor procurado é menor que o valor no nó-raiz (continua pela sub-árvore esquerda), ou se o valor é maior que o valor no nó-raiz (sub-árvore direita)
  - Caso o nó contendo a chave pesquisada seja encontrado, retorne **true** e o nó pesquisado, caso contrário retorne **false**

Pesquisa

```
1  ITEM *busca_aux(NO *raiz, int chave) {
2    if (raiz == NULL) {
3      return NULL;
4    } else {
5      if (raiz->item->chave > chave) {
6        return busca_aux(raiz->filhoesq, chave);
7      } else if (raiz->item->chave < chave) {
8        return busca_aux(raiz->filhodir, chave);
9      } else {
10       return raiz->item;
11     }
12   }
13 }
14
15 ITEM *busca(ARVORE_BINARIA *arvore, int chave) {
16   return busca_aux(arvore->raiz, chave);
17 }
```

Exercícios

- Criar um método iterativo para busca em ABB

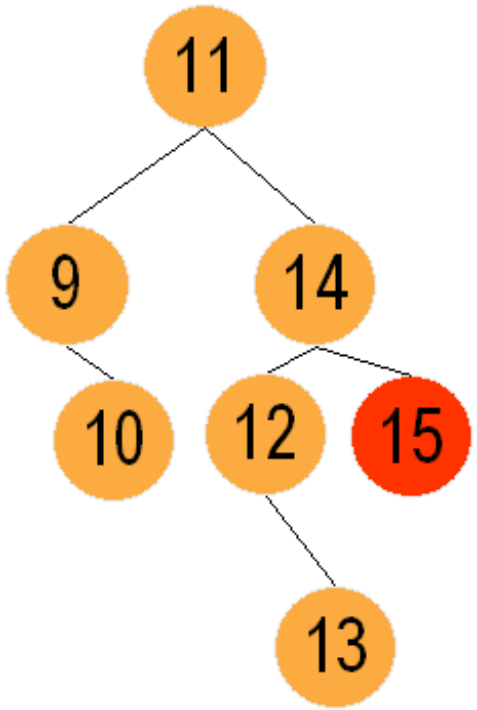
Sumário

Remoção (operações em ABB's)

- 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) "ocupa" o lugar do nó retirado
  - **Caso 3**: o nó possui duas sub-árvores
    - O nó contendo o menor valor da sub-árvore direita pode "ocupar" o lugar
    - Ou o maior valor da sub-árvore esquerda pode "ocupar" o lugar

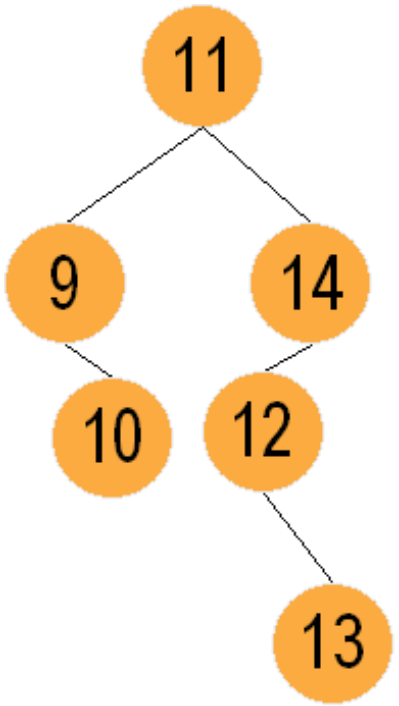
Remoção - Caso 1

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



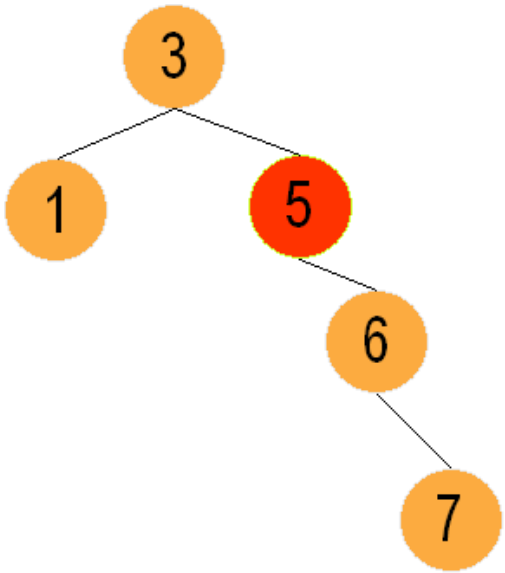
Remoção - Caso 1

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



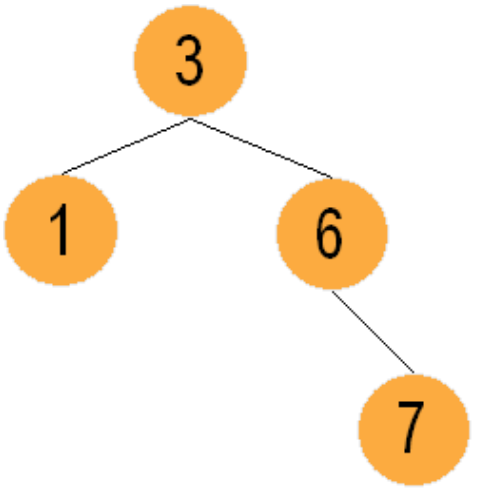
Remoção - Caso 2

- Removendo-se o nó com o valor 5
- Como ele possui somente a sub-árvore direita, o nó contendo o valor 6 pode "ocupar" o lugar do nó removido



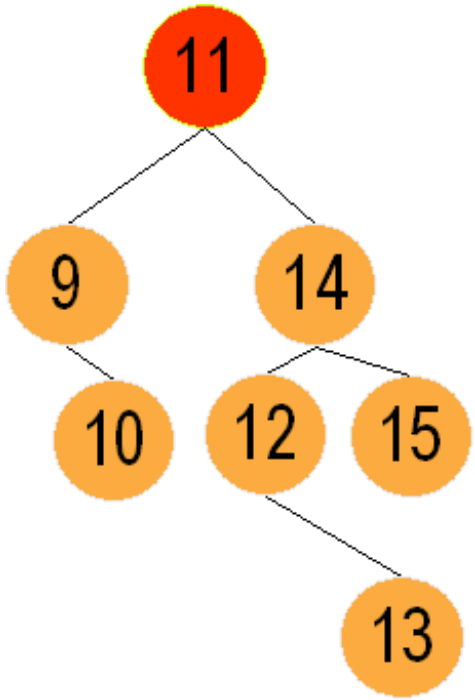
Remoção - Caso 2

- Esse segundo caso é análogo, caso existisse um nó com somente uma sub-árvore esquerda

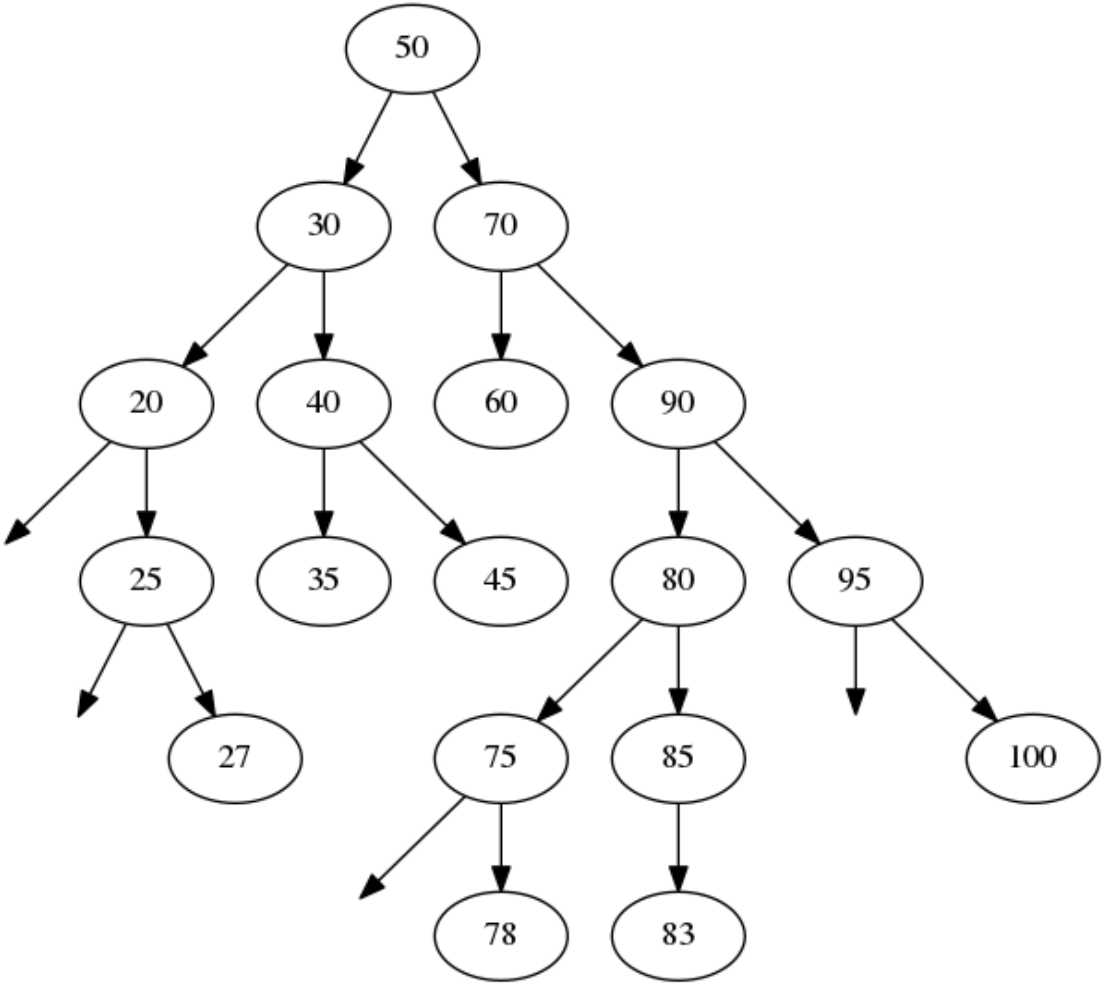


Remoção - Caso 3

- Eliminando-se o nó de chave 11
- Neste caso, existem 2 opções
  - O nó com chave 10 pode "ocupar" o lugar do nó-raiz, ou
  - O nó com chave 12 pode "ocupar" o lugar do nó-raiz



Remoção - Exemplos



Remoção

```
1  int remove_abb_aux(ARVORE_BINARIA * arv, NO *prem, NO *pant, int chave↩
       ) {
2    if (prem == NULL) {
3      return 0;
4    } else if (prem->item->chave > chave) {
5      return remove_abb_aux(arv, prem->filhoesq, prem, chave);
6    } else if (prem->item->chave < chave) {
7      return remove_abb_aux(arv, prem->filhodir, prem, chave);
8    } else {
9      if (prem->filhoesq == NULL || prem->filhodir == NULL) {
10       NO *pprox = (prem->filhoesq == NULL) ? prem->filhodir : prem->↩
             filhoesq;
11
12       if (pant == NULL) arv->raiz = pprox;
13       else if (prem == pant->filhoesq) pant->filhoesq = pprox;
14       else pant->filhodir = pprox;
15
16       apagar_item(&(prem->item));
17       free(prem);
18     } else {
19       troca_max_esq(prem->filhoesq, prem, prem);
20     }
21     return 1;
22   }
23 }
```

Remoção

- Troca com o máximo elemento da sub-árvore esquerda

```
1  void troca_max_esq(NO *ptroca, NO *prem, NO *pant) {
2    if (ptroca->filhodir != NULL) {
3      troca_max_esq(ptroca->filhodir, prem, ptroca);
4    }
5    else {
6
7       if (prem == pant)
8          pant->filhoesq = ptroca->filhoesq;
9       else
10         pant->filhodir = ptroca->filhoesq;
11
12      apagar_item(&(prem->item));
13      prem->item = ptroca->item;
14      free(ptroca);
15   }
16 }
```

Remoção

```
1  int remove_abb(ARVORE_BINARIA *arv, int chave) {
2    return remove_abb_aux(arv, arv->raiz, NULL, chave);
3  }
```

Sumário

Complexidade da busca em ABB

- Pior caso
  - Número de passos é determinado pela altura da árvore
  - Árvore degenerada possui altura igual a n
- Altura da árvore depende da sequência de inserção das chaves
  - O que acontece se uma sequência ordenada de chaves é inserida
- Busca eficiente se árvore razoavelmente balanceada

Árvores Binárias de Busca

- ABB "aleatória"
  - Nós externos: descendentes dos nós folha (não estão, de fato, na árvore)
  - Uma árvore A com n nós possui $n + 1$ nós externos
  - Uma inserção em A é considerada "aleatória" se ela tem probabilidade igual de acontecer em qualquer um dos $n + 1$ nós externos
  - Uma ABB aleatória com n nós é uma árvore resultante de n inserções aleatórias sucessivas em uma árvore inicialmente vazia

Árvores Binárias de Busca

- É possível demonstrar que para uma ABB "aleatória" o número esperado de comparações para recuperar um registro qualquer é cerca de $1,39 * log_2(n)$
  - 39% pior do que o custo do acesso em uma árvore balanceada
- Pode ser necessário garantir um melhor balanceamento da ABB para melhor desempenho na busca

Árvores Binárias de Busca

- As complexidade das operações de inserção e remoção também dependem da eficiência da busca
- O tempo necessário para realizar essas operações depende principalmente do tempo necessário para encontrar a posição do nó a ser inserido/removido
- A remoção ainda pode requerer encontrar o nó máximo da sub-árvore esquerda ($troca\_max\_esq(...)$), mas o número de operações realizadas é sempre menor ou igual do que a altura da árvore

Exercícios

- Quais sequências de inserções criam uma ABB degenerada? Quais sequências criam uma ABB balanceada?
- Implemente um TAD para árvores binárias de busca com as operações discutidas em aula
- Implemente uma versão iterativa do algoritmo de remoção em ABBs

Exercícios

- Escreva uma função que verifique se uma árvore binária está perfeitamente balanceada
  - O número de nós de suas sub-árvores esquerda e direita difere em, no máximo, 1