Tipos Especiais de Listas

Árvores Binárias de Busca

Sumário

- Conceitos Introdutórios
- Inserção em Árvores Binárias de Busca
- Pesquisa em Árvores Binárias de Busca
- Remoção em Árvores Binárias de Busca
- Conceitos Adicionais

Definições

- Uma Árvore Binária de Busca (ABB) possui as seguintes propriedades
  - É uma Árvore Binária (AB)
  - Seja S = {s1; : : : ; sn} o conjunto de chaves dos nós da árvore T
    - Esse conjunto satisfaz s1 < : : : < sn
    - A cada nó vj ∈ T está associada uma chave distinta sj ∈ S, que pode ser consultada por r(vj) = sj
  - Dado um nó v de T
    - Se vi pertence a sub-árvore esquerda de v, então r(vi) < r(v)
    - Se vi pertence a sub-árvore direita de v, então r(vi) > 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

Inserção em Árvores Binárias de Busca

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

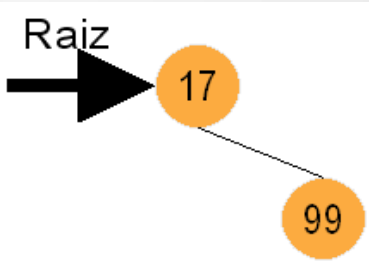
- Para entender o algoritmo considere a inserção do conjunto de números, na sequência

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

- No início a ABB está vazia!

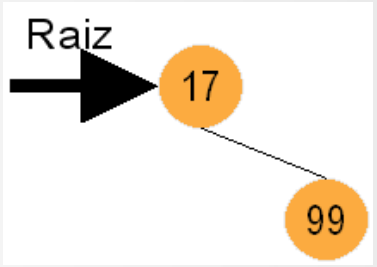
Inserção

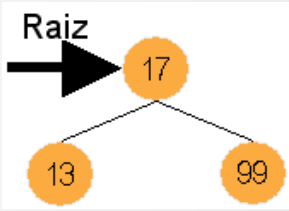
- O número 17 será inserido tornando-se o nó raiz.
- A inserção do 99 inicia-se na raiz.
- Compara-se 99 com 17 Como 99 > 17, 99 deve ser colocado na sub-árvore direita do nó contendo 17 (subárvore direita, inicialmente, nula)



Inserção

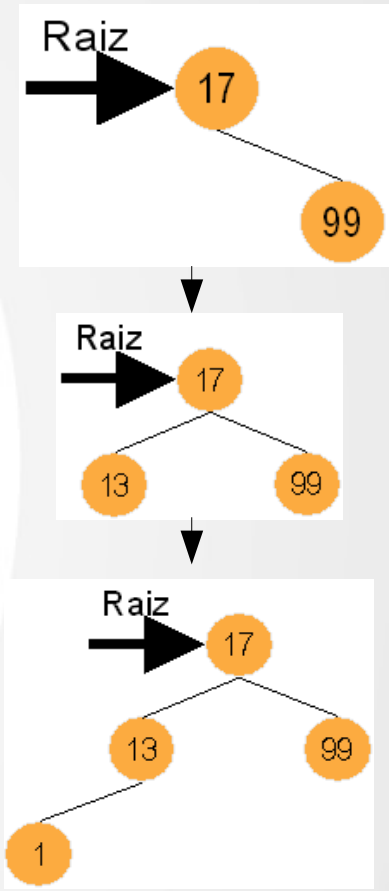
- A inserção do 13 inicia-se na raiz.
- Compara-se 13 com 17.
- Como 13 < 17, 13 deve ser colocado na sub-árvore esquerda do nó contendo 17
- Já que o nó 17 não possui descendente esquerdo, 13 é inserido na árvore nessa posição





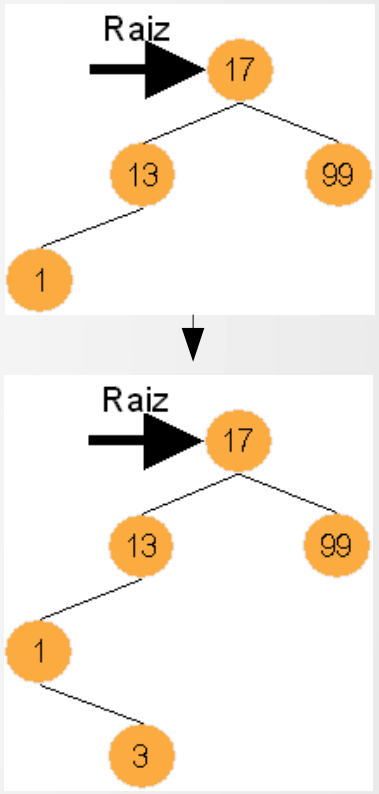
Inserção

- 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



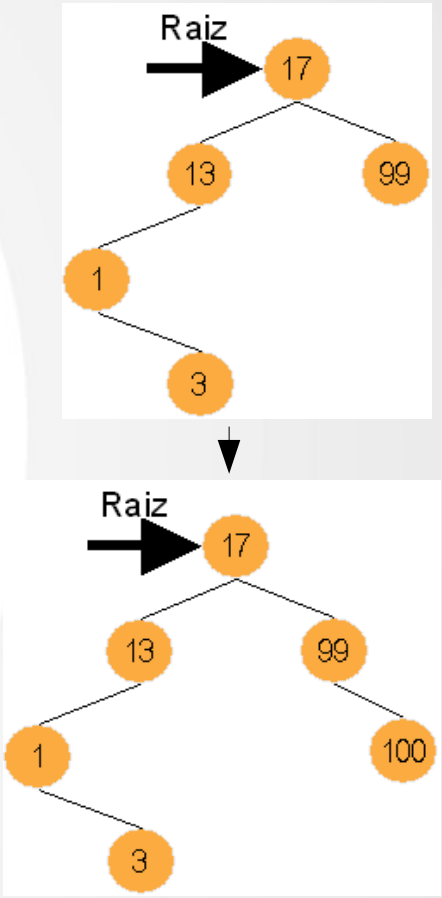
Inserção

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



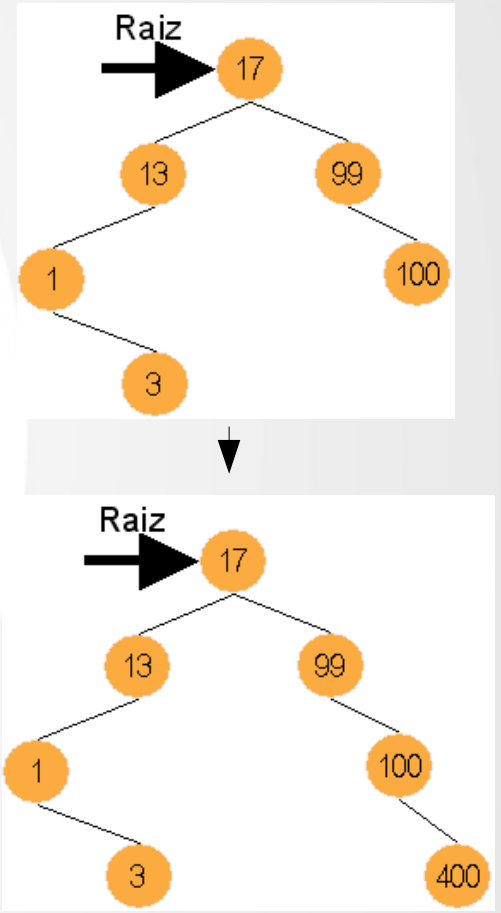
Inserção

- 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

```
typedef struct {
 int chave;
 int valor;
} INFO;
```

```
typedef struct NO {
   INFO info;
   struct NO *fesq;
   struct NO *fdir;
} tNO;
```

```
typedef struct {
   NO *raiz;
} ARVORE;
```

Relembrando

```
void criar(ARVORE *arv) {
      arv->raiz = NULL;
}
```

```
int criar_raiz(ARVORE *arv, INFO *info) {
  arv->raiz = (tNO*)malloc(sizeof(tNO));
  if (arv->raiz != NULL) {
      arv->raiz->fesq = NULL;
      arv->raiz->fdir = NULL;
      arv->raiz->info = *info;
      return 1;
  }
  return 0;
}
```

Relembrando

```
tNO *inserir_direita(tNO *no, INFO *info) {
   nO->fdir = (tNO*)malloc(sizeof(tNO));
   if (no->fdir != NULL) {
     no->fdir->fesq = NULL;
     no->fdir->fdir = NULL;
     no->fdir->info = *info;
   }
   return no->fdir;
}
```

Relembrando

```
tNO *inserir_esquerda(tNO *no, INFO *info) {
   no->fesq = (NO*)malloc(sizeof(NO));
   if (no->fesq != NULL) {
     no->fesq->fesq = NULL;
     no->fesq->fdir = NULL;
     no->fesq->info = *info;
   }
   return no->fesq;
}
```

```
int inserir_aux(tNO *no, INFO *info) {
   if (no->info.chave > info->chave) {
      if (no->fesq == NULL) {
         return (inserir_esquerda(no, info) != NULL);
      } else {
         return inserir_aux(no->fesq, info);
      }
   } else {
      if (no->fdir == NULL) {
          return (inserir_direita(no, info) != NULL);
      } else {
         return inserir_aux(no->fdir, info);
      }
   }
}
```

```
int inserir(ARVORE *arv, INFO *info) {
   if (arv->raiz == NULL) {
      return criar_raiz(arv, info);
   } else {
      return inserir_aux(arv->raiz, info);
   }
}
```

Exercícios

- Criar um método iterativo para inserção em ABB

Pesquisa em Árvores Binárias de Busca

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

```
int buscar_aux(NO *no, INFO *info) {
    if (no == NULL) {
      return 0;
    } else {
      if (no->info.chave > info->chave) {
         return buscar_aux(no->fesq, info);
      } else if (no->info.chave < info->chave) {
         return buscar aux(no->fdir, info);
      } else {
         info->valor = no->info.valor;
         return 1;
       }
    }
 }
```

```
int buscar(ARVORE *arv, INFO *info) {
    return buscar_aux(arv->raiz, info);
}
```

Exercícios

- Criar um método iterativo para pesquisa em ABB

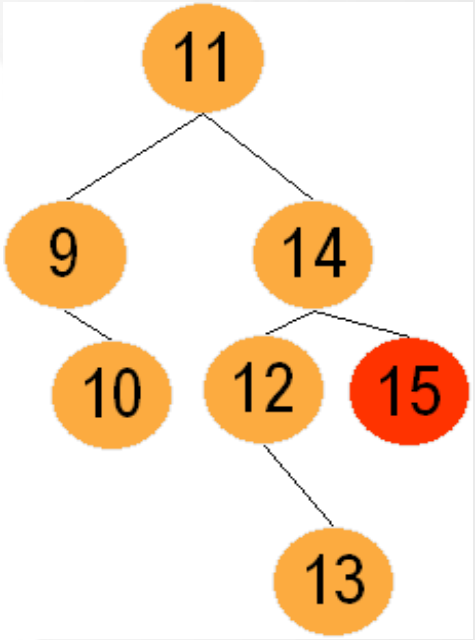
Remoção em Árvores Binárias de Busca

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

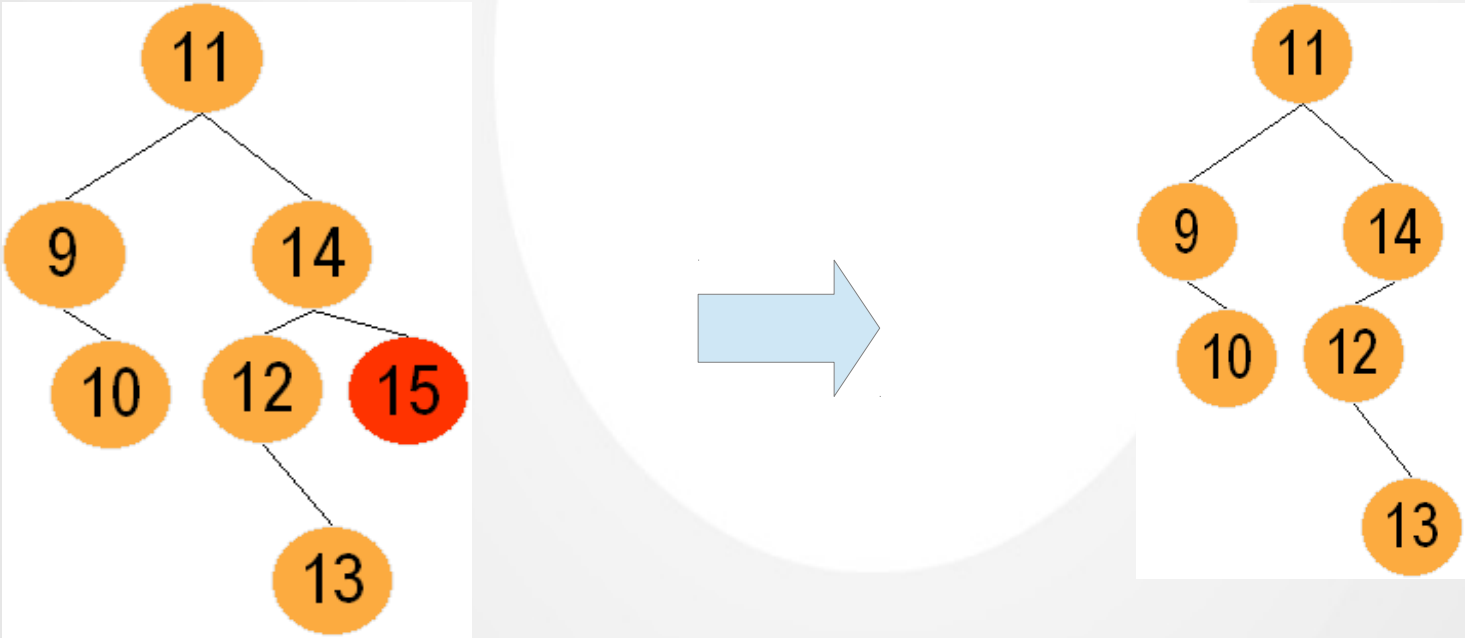
Remoção – Caso 1

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



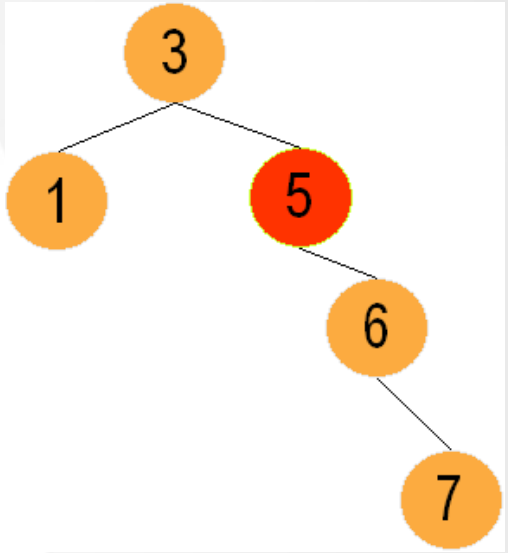
Remoção – Caso 1

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



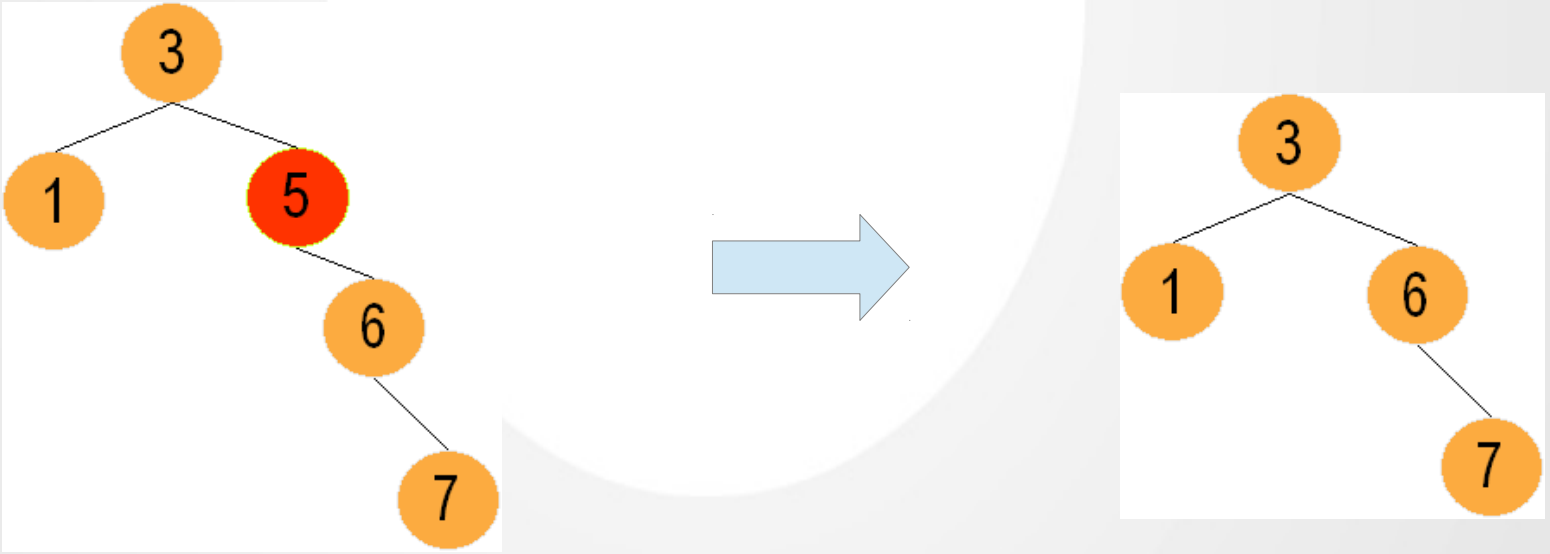
Remoção – Caso 2

- 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



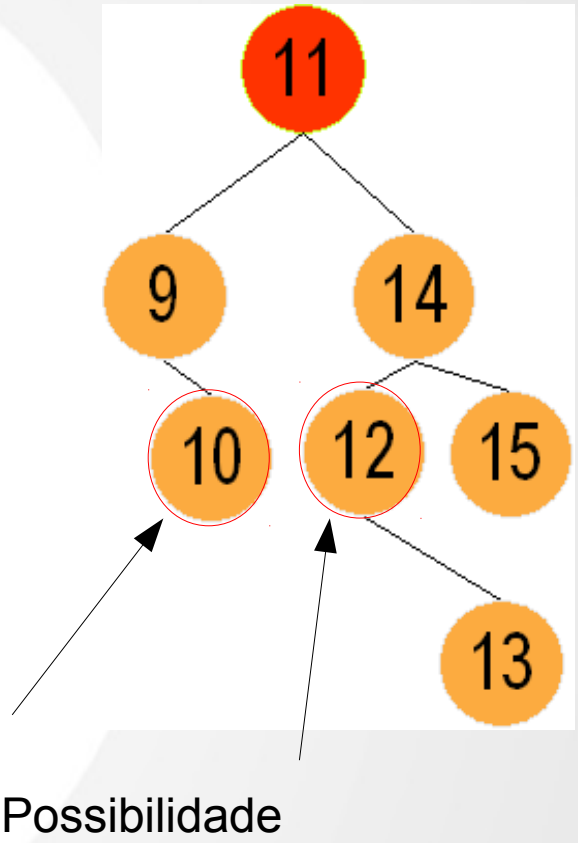
Remoção – Caso 2

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

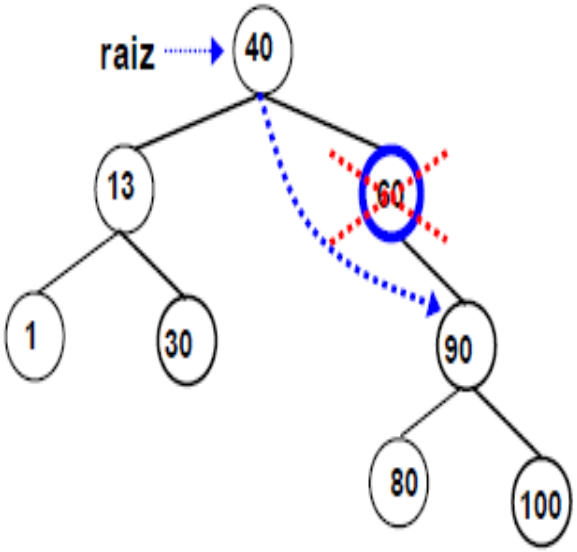


Remoção – Caso 3

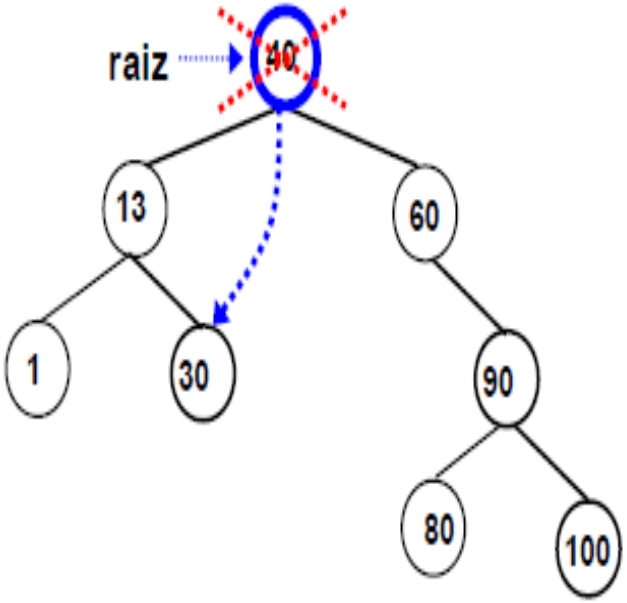
- Eliminando-se o nó de chave 11
- Neste caso, existem 2 opções
  - O nó com chave 10 pode "ocupar" o lugar do nó-raiz, ou
  - O nó com chave 12 pode "ocupar" o lugar do nó-raiz

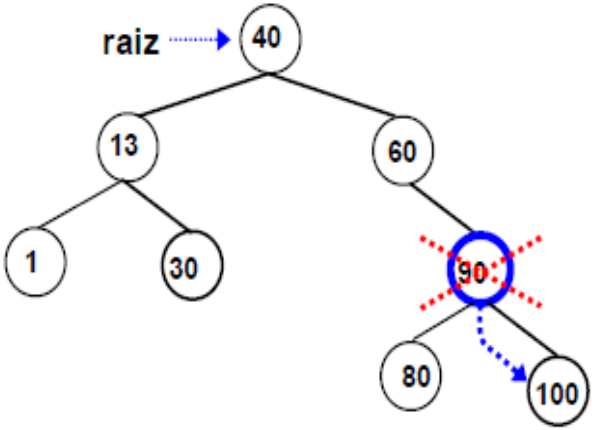


Remoção – Caso 3



Ou





```
int remover_aux(ARVORE *arv, NO *pant, NO *prem, int chave) {
   if (prem == NULL) { //chave não encontrada
      return 0;
   } else if (prem->info.chave > chave) {
     return remover_aux(arv, prem, prem->fesq, chave);
   } else if (prem->info.chave < chave) {
      return remover_aux(arv, prem, prem->fdir, chave);
   } else {
      //tem apenas um filho
       if (prem->fdir == NULL || prem->fesq == NULL) {
       //pega o único filho não nulo
 tNO *pprox = (prem->fdir == NULL) ? prem->fesq : prem->fdir;
          if (pant == NULL)  arv->raiz = pprox; //raiz
          else if (pant->fdir == prem) pant->fdir = pprox;
          else pant->fesq = pprox;
          free(prem);
       } else { //tem os dois filhos
         //troca com o maximo da subarvore esquerda
         trocar_max_esq(prem, prem, prem->fesq);
       }
    return 1;
 }
}
```

Pesquisa

Inicialmente iguais

Inicialmente esq

- Troca com o máximo elemento da sub-árvore esquerda

```
void trocar_max_esq(NO *prem, NO *pant, NO *pno) {
     if (pno->fdir != NULL) {
        trocar_max_esq(prem, pno, pno->fdir);
    } else {
    //raiz da arvore esquerda não tem filhos a direita
      if (pant != prem)
            pant->fdir = pno->fesq; //pno->dir = NULL
      else
          pant->fesq = pno->fesq; //pno->dir = NULL
      prem->info = pno->info; //realiza a copia
      free(pno);
    }
}
```

Remoção

```
int remover(ARVORE *arv, int chave) {
 if (arv->raiz != NULL) {
  return remover_aux(arv, NULL, arv->raiz, chave);
 }
 return 0;
}
```

Conceitos Adicionais

Exercícios

- 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 *log2(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

Exercícios

- Criar um método iterativo para pesquisa em ABB