Árvores de busca binária

Prof. Mauro Henrique Mulati

UNICENTRO

Embasado no capítulo 12 do livro do Cormen, 3.a ed.

O estudo utilizando apenas este material **não é suficiente** para o entendimento do conteúdo. Recomendamos a leitura das referências no final deste material e a resolução (por parte do aluno) de todos os exercícios indicados.

Conteúdo

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:
  - `Search`

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:
  - `Search`
  - `Minimum`, `Maximum`

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:
  - `Search`
  - `Minimum`, `Maximum`
  - `Predecessor`, `Successor`

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:
  - `Search`
  - `Minimum`, `Maximum`
  - `Predecessor`, `Successor`
  - `Insert`, `Delete`

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:
  - `Search`
  - `Minimum`, `Maximum`
  - `Predecessor`, `Successor`
  - `Insert`, `Delete`
- Pode ser usada como **dicionário** e como **fila de prioridades**

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:
  - `Search`
  - `Minimum`, `Maximum`
  - `Predecessor`, `Successor`
  - `Insert`, `Delete`
- Pode ser usada como **dicionário** e como **fila de prioridades**
- Operações básicas levam tempo proporcional à altura da árvore
  - Árvore binária completa com n nós: Pior caso é $\Theta(\lg n)$
  - Cadeia linear de n nós: Pior caso é $\Theta(n)$

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:
  - `Search`
  - `Minimum`, `Maximum`
  - `Predecessor`, `Successor`
  - `Insert`, `Delete`
- Pode ser usada como **dicionário** e como **fila de prioridades**
- Operações básicas levam tempo proporcional à altura da árvore
  - Árvore binária completa com n nós: Pior caso é $\Theta(\lg n)$
  - Cadeia linear de n nós: Pior caso é $\Theta(n)$
- Há diferentes tipos de árvores de busca incluem:
  - Árvores de busca binária
  - Árvores vermelho-preto
  - Árvores B

Introdução: **Árvores de busca**

- Estruturas de dados que suportam muitas operações de conjuntos dinâmicos incluindo:
  - `Search`
  - `Minimum`, `Maximum`
  - `Predecessor`, `Successor`
  - `Insert`, `Delete`
- Pode ser usada como **dicionário** e como **fila de prioridades**
- Operações básicas levam tempo proporcional à altura da árvore
  - Árvore binária completa com n nós: Pior caso é $\Theta(\lg n)$
  - Cadeia linear de n nós: Pior caso é $\Theta(n)$
- Há diferentes tipos de árvores de busca incluem:
  - Árvores de busca binária
  - Árvores vermelho-preto
  - Árvores B
- Agora: Árvores de busca binária, percurso e operações

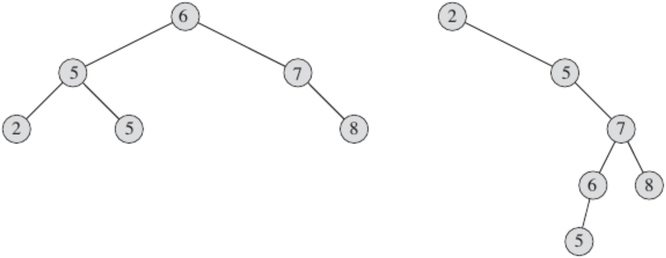
Árvores de busca binária I

Árvore de busca binária é uma estrutura de dados importante para conjuntos dinâmicos

- Realizam muitas operações de conjunto dinâmico em tempo $O(h)$, onde h é a altura da árvore
- Representamos uma árvore binárias por uma estrutura de dados ligada
- `T.raiz`/`T.root` é um ponteiro para a raiz da árvore `T`
- Cada nó contém os campos:
  - `chave`/`key` (e possivelmente outros dados satélites)
  - `esquerda`/`left`: ponteiro para o filho da esquerda
  - `direita`/`right`: ponteiro para o filho da direita
  - `p`: ponteiro para o pai. `T.raiz.p = NIL`

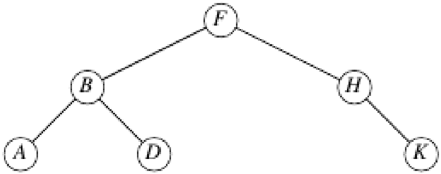
Árvores de busca binária II

- Chaves armazenadas devem satisfazer a **propriedade de árvore de busca binária:**
  - Seja x um nó em uma árvore de busca binária
  - Se y é um nó na subárvore esquerda de x, então `y.chave` $\leq$ `x.chave`
  - Se y é um nó na subárvore direita de x, então `y.chave` $\geq$ `x.chave`



(a) (b)

Árvores de busca binária III



A propriedade de árvore de busca binária nos permite imprimir todas as chaves em uma árvore de busca binária em sequência ordenada, recursivamente, usando o algoritmo de **percurso de árvore em-ordem**. Elementos são impressos em ordem monotonicamente crescente.

Árvores de busca binária IV

Como funciona:

- Verifica se x não é `NIL`
- Recursivamente, imprime as chaves dos nós na subárvore esquerda de x
- Imprime a chave de x
- Recursivamente, imprime as chaves dos nós na subárvore direita de x

Chamada: `Inorder-Tree-Walk(T.raiz)`

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

Exemplo (...)

Árvores de busca binária V

Tempo: Intuitivamente, realizar o percurso toma tempo $\Theta(n)$ para uma árvore com n nós, pois visita e imprime cada nó uma vez. Após a chamada inicial, o procedimento chama a si mesmo recursivamente duas vezes para cada nó na árvore (uma vez para o filho da esquerda e uma vez para o filho da direita).
Consulte prova formal no livro.
E ainda:

- Percurso de árvore em **pré-ordem**: Imprime a raiz antes dos valores das subárvores
- Percurso de árvore em **pós-ordem**: Imprime a raiz depois dos valores em suas subárvores

Consultas em uma árvore de busca binária: Busca I

Busca

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

- Chamada inicial: `Tree-Search(T.raiz, k)`
- Exemplo: Buscar pelos valores D e C no exemplo (...).
- **Tempo**: Os nós encontrados durante a recursão formam um caminho simples descendente partindo da raiz da árvore e, portanto, o tempo de execução de `Tree-Search` é $O(h)$ onde h é a altura da árvore.

Consultas em uma árvore de busca binária: Busca II

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Consultas em uma árvore de busca binária: Mínimo e máximo I

A propriedade de árvore de busca binária garante que:

- A chave mínima de uma árvore de busca binária está localizada no nó mais a esquerda; e
- A chave máxima de uma árvore de busca binária está localizada no nó mais a direita

Percorrer os ponteiros apropriados (`esquerda` ou `direita`) até que NIL seja atingido.

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

Consultas em uma árvore de busca binária: Mínimo e máximo II

Tempo: Ambos os procedimentos visitam nós que formam um caminho para baixo desde a raiz até uma folha. Ambos os procedimento executam em tempo $O(h)$, onde h é a altura da árvore.

Consultas em uma árvore de busca binária: Sucessor e predecessor I

- Assumindo todas as chaves distintas,
- O sucessor de um nó `x` é o nó `y` de modo que `y.key` é a menor chave $>$ `x.key`
- Se `x` tem a maior chave em uma árvore de busca binária, o sucessor de `x` é `NIL`

Dois casos:

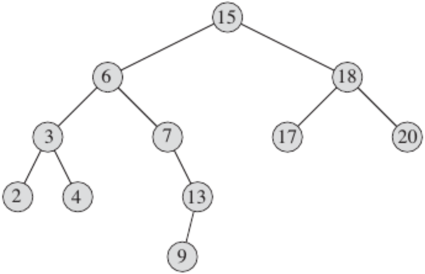
1. Se o nó `x` tem uma subárvore direita não-vazia, então o sucessor de `x` é o mínimo na subárvore direita de `x`
2. Se o nó `x` tem uma subárvore direita vazia, note que:
    - Enquanto movemos para esquerda e para cima (através de filhos da direita), estamos visitando chaves menores
    - O sucessor de `x`, denotado por `y`, é o nó do qual `x` é o predecessor (`x` é o máximo na subárvore esquerda de `y`)

Consultas em uma árvore de busca binária: Sucessor e predecessor II

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

Consultas em uma árvore de busca binária: Sucessor e predecessor III

- Exemplo:



- Encontrar o sucessor do nó com valor 15 (...)
- Encontrar o sucessor do nó com valor 6 (...)
- Encontrar o sucessor do nó com valor 4 (...)

Consultas em uma árvore de busca binária: Sucessor e predecessor IV

Encontrar o **predecessor** funciona de maneira simétrica
Exemplo:

- Encontrar o predecessor do nó com valor 6 (...)

Tempo: Para ambos os algoritmos, em ambos os casos, visitamos os nós em caminho para baixo ou para cima na árvore. Assim, o tempo de execução é $O(h)$.

Inserção I

- Inserção e eliminação provocam mudanças no conjunto dinâmico representado por uma árvore de busca binária
- A estrutura de dados deve ser modificada para refletir esta mudança
- A propriedade de árvore de busca binária deve continuar válida

Inserção II

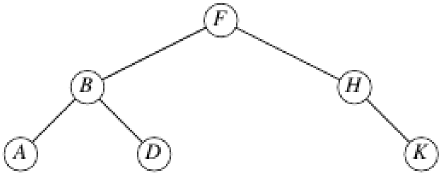
```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Inserção III

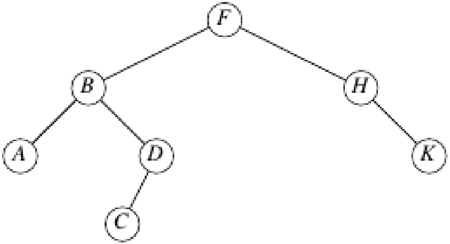
- Para inserir o valor `v`: é dado nó `z` com `z.key = v`, `z.left = NIL` e `z.right = NIL`
- Começando na raiz da árvore, percorrer um caminho para baixo mantendo dois ponteiros:
  - Ponteiro `x`: percorre o caminho para baixo
  - Ponteiro `y`: mantém ponteiro para o pai de `x`
- Percorre a árvore para baixo comparando o valor do nó em `x` com `v`, e move para o filho da esquerda ou da direita
- Quando `x` é `NIL`, está na posição correta para o nó `z`
- Comparar o valor de `z` com o valor de `y` e inserir `z` à esquerda ou à direita de `y`, apropriadamente

Inserção IV

Exemplo: Inserir `C`



Inserção V



Inserção VI

Tempo da inserção: Mesmo que `Tree-Search`. Em uma árvore de altura h, o procedimento leva tempo $O(h)$.

Eliminação I

- Conceitualmente, eliminar nó z de uma árvore binária de busca T tem três casos:
  - Se z não tem filhos, remova-o
  - Se z tem apenas um filho, faça este filho tomar a posição de z na árvore, trazendo junto a subárvore deste filho
  - Se z tem dois filhos, então encontre o sucessor de z denotado por y e substitua z por y na árvore
    - y deve estar na subárvore direita de z e y não tem filho à esquerda
    - O resto da subárvore original (à direita) de z se torna a nova subárvore direita de y e
      a subárvore esquerda de z se torna a nova subárvore esquerda de y

  Este caso é um pouco complicado, pois o fato de y ser ou não o filho à direita de z influi na sequência de passos

Eliminação II

- O pseudocódigo organiza os casos um pouco diferente
- Dado que moveremos subárvores na árvore de busca binária
- Usamos a subrotina `Transplant`, para substituir uma subárvore como um filho de seu pai por outra subárvore

TRANSPLANT(T, u, v)

if $u.p$ == NIL
    $T.root = v$
elseif u == $u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$

Eliminação III

`Transplant(T, u, v)` substitui a subárvore enraizada em `u` pela subárvore enraizada em `v`:

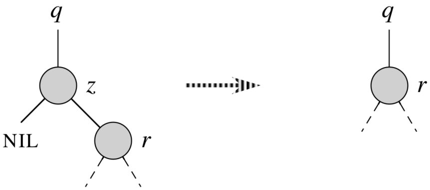
- Faz o pai de `u` se tornar o pai de `v`
  (a menos que `u` é a raiz, caso em que faz `v` ser a raiz)
- Pai de `u` toma `v` como filho à esquerda ou filho à direita, dependendo se `u` era filho à esquerda ou à direita
- Não atualiza `v.left` nem `v.right`, deixando para o chamador de `Transplant`

Eliminação IV

`Tree-Delete(T,z)` tem 4 casos na eliminação de `z` de uma árvore de busca binária `T`:

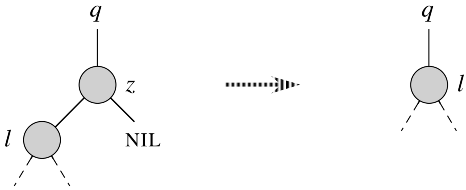
- Se `z` não tem filho à esquerda, substitua `z` por seu filho da direita
  O filho da direita pode ou não ser `NIL`
  Se o filho da direita de `z` for `NIL`, este caso trata a situação em que `z` não tem filhos

(a)



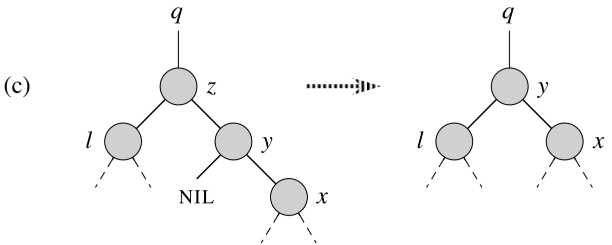
Eliminação V

- Se z tem somente um filho, e este filho é à esquerda, então substituia z por seu filho à esquerda

(b) 

Eliminação VI

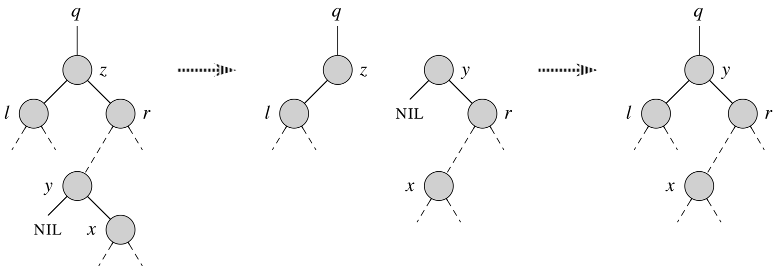
- **Caso contrário**, z tem dois filhos.
  Encontre o sucessor de z denotado por y.
  y deve situar-se na subárvore à direita de z e não tem filho à esquerda.
  O objetivo é substituir z por y, recortando y de sua localização atual.
  - Se y é o filho da direita de z, substitua z por y e deixe o filho da direita de y em paz



Eliminação VII

- Caso contrário, y situa-se na subárvore direita de z, mas não na raiz desse subárvore.
  Substitua y pelo seu próprio filho da direita.
  Então substitua z por y.

(d)



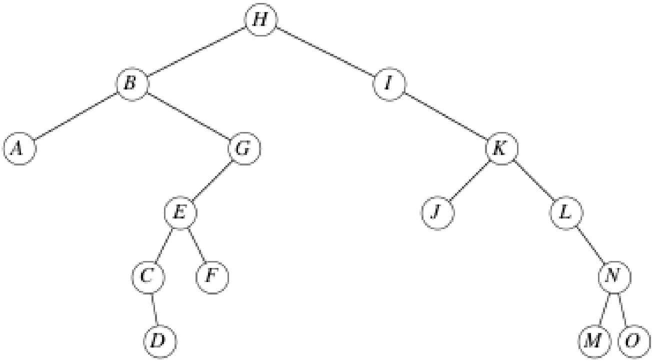
Eliminação VIII

```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)         // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)          // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)         // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this subtree.
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

Note que as últimas 3 linhas são executadas quando z tem 2 filhos, independentemente do fato de y ser ou não filho à direita de z

Eliminação IX

Exemplo:



Eliminação X

- `Tree-Delete(T, I)`: o nó eliminado não tem filho à esquerda
- `Tree-Delete(T, G)`: o nó eliminado tem filho à esquerda mas não à direita
- `Tree-Delete(T, K)`: o nó eliminado tem os dois filhos e seu sucessor é o filho à direita
- `Tree-Delete(T, B)`: o nó eliminado tem os dois filhos e seu sucessor não é o filho à direita

Eliminação XI

Tempo da eliminação: $O(h)$, em uma árvore de altura h. Todas as operações tomam tempo $O(1)$, exceto a chamada a `Tree-Minimum`, que gasta $O(h)$.

Minimizando o tempo de execução I

Analisamos os tempos de execução em termos de h (a altura de uma árvore binária de busca), ao invés de n (o número de nós em uma árvore)

- Problema: Pior caso para árvore de busca binária é $\Theta(n)$ – não é melhor que uma lista ligada
- Solução: Garantir altura pequena (árvore balanceada) – $h = O(\lg n)$

Variando propriedades de árvores de busca binárias, pode-se analisar o tempo de execução em termos de n

- Método: Restruturar a árvore se necessário. Nada especial é requerido para consultas, mas há trabalho extra na alteração da estrutura da árvore (inserção e remoção)

Árvores vermelho-preto são uma classe especial de árvores binárias que evitam o comportamento no pior caso de $O(n)$

Referências

- Thomas H. Cormen et al. Introdução a Algoritmos. 3ª edição. Capítulo 12.