MC202 - Estruturas de Dados

Guilherme P. Telles

IC

18 de Maio de 2023

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides usam português anterior à reforma ortográfica de 2009.

Parte I

Busca

Busca

- Dado um conjunto de registros $R = \{R_1, R_2, \ldots, R_n\}$ com chaves distintas $k_1, k_2, \ldots, k_n$ e dada uma chave x, o *problema da busca* é encontrar o registro em R com chave igual a x.
- O resultado da busca pode ser o registro em R com chave x ou a conclusão de que nenhum registro em R tem chave igual a x.
- Vamos usar expressões da forma "a chave" como sinônimo de "registro que tem a chave", lembrando que cada registro tem outros dados de interesse da aplicação relacionados com a chave.

Roteiro (não necessariamente nessa ordem)

- Chaves em um intervalo contínuo ou denso: acesso direto.
- Poucas chaves ou poucas buscas: busca seqüencial.
- Muitas buscas ou muitas chaves:
  - Que não mudam ou mudam pouco: busca binária, hashing.
  - Que mudam: hashing, árvore de busca binária, árvore AVL.

Busca em um intervalo contínuo ou denso de chaves

- Vamos supor que as chaves são inteiros e formam um intervalo contínuo
  - p.ex. entre 1 e $1.217.458$, entre 491.875 e $10.519.005$, em geral entre k e ℓ

  ou formam um intervalo denso
  - p.ex. todos os pares entre 1 e $1.217.458$, todos os números entre k e ℓ em que 5% estão faltando, etc.

- Podemos colocar os registros consecutivamente na memória (em um vetor) e resolver o problema da busca com apenas um cálculo de endereço e um acesso à memória.

- Podemos colocar os registros consecutivamente na memória (em um vetor) e resolver o problema da busca com apenas um cálculo de endereço e um acesso à memória.
- O desempenho dessa solução é ótimo (não pode ser melhorado). O uso de memória extra é pequeno demais para ser importante.

- Podemos colocar os registros consecutivamente na memória (em um vetor) e resolver o problema da busca com apenas um cálculo de endereço e um acesso à memória.
- O desempenho dessa solução é ótimo (não pode ser melhorado). O uso de memória extra é pequeno demais para ser importante.
- Na vida real raramente as chaves formam um intervalo contínuo ou denso.

Busca seqüencial

- A busca seqüencial percorre o vetor ou lista encadeada a partir da primeira posição, até encontrar k ou chegar ao fim.
- Se são poucos dados ou se o número de buscas é pequeno então a busca seqüencial pode ser suficientemente eficiente.

Busca seqüencial (2cmps)

```
SEQUENTIAL-SEARCH(A[1, n], k)
1  i = 1
2  while (i <= n)
3        if A[i] == k
4              return i
5  return 0
```

Busca seqüencial (1cmp)

```
SEQUENTIAL-SEARCH(A[1, n], k)
1  A[n + 1] = k
2  i = 1
3  while (true)
4       if A[i] == k
5            if i < n + 1
6                 return i
7            else
8                 return 0
9       i = i + 1
```

Número de comparações de 1cmp

- Uma busca mal-sucedida faz $n + 1$ comparações.
- A busca bem-sucedida faz $i + 1$ comparações para encontrar a chave na posição i de A.
- O número médio de comparações de chaves para buscas bem-sucedidas quando as chaves têm a mesma probabilidade de serem buscadas é

$$\frac{2 + 3 + \ldots + n + 1}{n} = \frac{n + 3}{2}$$

Chaves com probabilidades conhecidas

- Se as probabilidades de cada chave ser buscada são conhecidas e não são iguais, as chaves podem ser organizadas em ordem decrescente de sua probabilidade.
- Isso reduz o número médio de comparações na busca seqüencial e pode ser suficientemente eficiente.
- Na vida real, raramente tais probabilidades são conhecidas e elas mudam com o tempo.

Estratégias de permutação

- A idéia dessas estratégias é aproveitar a localidade das buscas.
- P.ex. você chega na DAC para pedir uma declaração Z. Várias buscas pelo seu RA vão ser feitas em várias tabelas: aluno, disciplinas, catálogo etc. Algumas acontecem mais de uma vez na mesma tabela. Depois de pegar o histórico, não vai haver consulta pelo seu RA por muito tempo.

Localidade

- O princípio da localidade foi definido para páginas de memória, mas se aplica a registros da mesma forma:

1. Durante qualquer intervalo de tempo, um programa distribui os acessos não uniformemente sobre suas páginas,
2. vista como uma função do tempo, a freqüência com que uma dada página é acessada tende a mudar lentamente, ou seja, é quase estacionária, e
3. a correlação entre padrões de acesso imediatos tende a ser alta e a correlação entre padrões de acesso disjuntos tende a zero quando a distância entre eles tende ao infinito.

P.J. Denning e S.C. Schwartz. Properties of the working-set model. Comm. of ACM. 1972.

- Estratégias de permutação movem o registro que acabou de ser recuperado um certo número de posições em direção ao início do vetor ou lista encadeada, sem modificar a ordem relativa dos demais registros.
- As estruturas-de-dados são chamadas de vetor auto-organizável ou lista auto-organizável.

- As estratégias de permutação mais usadas incluem:
  - Move-to-front (MTF): quando um registro é buscado ele é movido para o início da seqüência.
  - Transpose: quando um registro é buscado ele é trocado de posição com o registro que o precede.
  - Count: cada registro tem um contador do número de acessos. Quando um registro é buscado o contador é incrementado e ele é movido para uma posição anterior a todos os registros com contador menor ou igual ao dele.
  - move-ahead-k: a chave buscada é movida k posições em direção ao início da seqüência.
- Nem todas as estratégias são boas em vetores, por exigirem muitas movimentações de registros.

- Para buscas em dados que exibem muita localidade temporal, busca seqüencial com uma estratégia de permutação pode ser suficientemente eficiente.

- Dicionário wamerican-2019.10.06-1: 102.774 termos colocadas em uma lista encadeada, inicialmente em ordem alfabética.
- Guerra e Paz, Leo Tolstoy, 577.058 palavras buscadas na lista, na mesma ordem em que aparecem no texo, sendo 24.843 buscas mal sucedidas.

	comparações de chave
Sequencial	37.385.210.305
MTF	1.761.400.314
Transpose	34.607.971.677
Count	1.716.974.879

Busca em chaves ordenadas

- Há métodos de busca eficientes quando as chaves estão ordenadas e em posições consecutivas na memória.
- A idéia é reduzir o espaço de busca a cada passo.
- Depois de comparar k contra k_i,
  ou $k < k_i$ e as chaves $k_i, \ldots, k_n$ podem deixar de ser consideradas,
  ou $k = k_i$ e a busca termina,
  ou $k > k_i$ e as chaves $k_1, \ldots, k_i$ podem deixar de ser consideradas.
- Se o número de buscas é grande, o custo da ordenação pode valer a pena.

Busca binária

- A busca binária é uma busca em um conjunto de chaves ordenadas em que a chave mediana é comparada contra k sucessivamente.

Busca binária (3cmps)

```
BINARY-SEARCH(A, ℓ, r, k)
1  if ℓ > r
2        return 0
3  m = ⌊(ℓ + r)/2⌋
4  if A[m] == k
5        return m
6  elseif k < A[m]
7        return BINARY-SEARCH(A, ℓ, m − 1, k)
8  else
9        return BINARY-SEARCH(A, m + 1, r, k)
```

Busca binária (2cmps)

```
BINARY-SEARCH(A, ℓ, r, k)
 1  if ℓ == r
 2        if A[ℓ] == k
 3              return ℓ
 4        else
 5              return 0
 6  else
 7        m = ⌈(ℓ + r)/2⌉
 8        if k < A[m]
 9              return BINARY-SEARCH(A, ℓ, m − 1, k)
10        else
11              return BINARY-SEARCH(A, m, r, k)
```

Número de comparações com 2cmps

- Na busca binária o espaço de busca é reduzido pela metade a cada passo.
- A cada passo do algoritmo a busca realiza trabalho constante e divide a entrada em partes de tamanho $\lceil \frac{n-1}{2} \rceil$ e $\lfloor \frac{n-1}{2} \rfloor$.
- Em uma busca o número de chamadas da função está entre $\lfloor \log_2 n \rfloor + 1$ e $\lfloor \log_2 n \rfloor + 2$. O número de comparações de chaves está entre $2(\lfloor \log_2 n \rfloor + 1)$ e $2(\lfloor \log_2 n \rfloor + 2)$.

Atualizações do conjunto de registros

- Quando a chave na posição i é removida, é necessário deslocar $n - i - 1$ regitros.
- Quando uma chave é inserida e deve ocupar a posição i, é necessário deslocar $n - i - 1$ chaves.
- Se as atualizações são freqüentes, manter o vetor ordenado vai custar muito caro.

Outras

- Existem outras buscas em chaves ordenadas como busca interpolada, busca Fibonacci etc.

Árvore Binária de Busca

Buscas em conjuntos de chaves que mudam

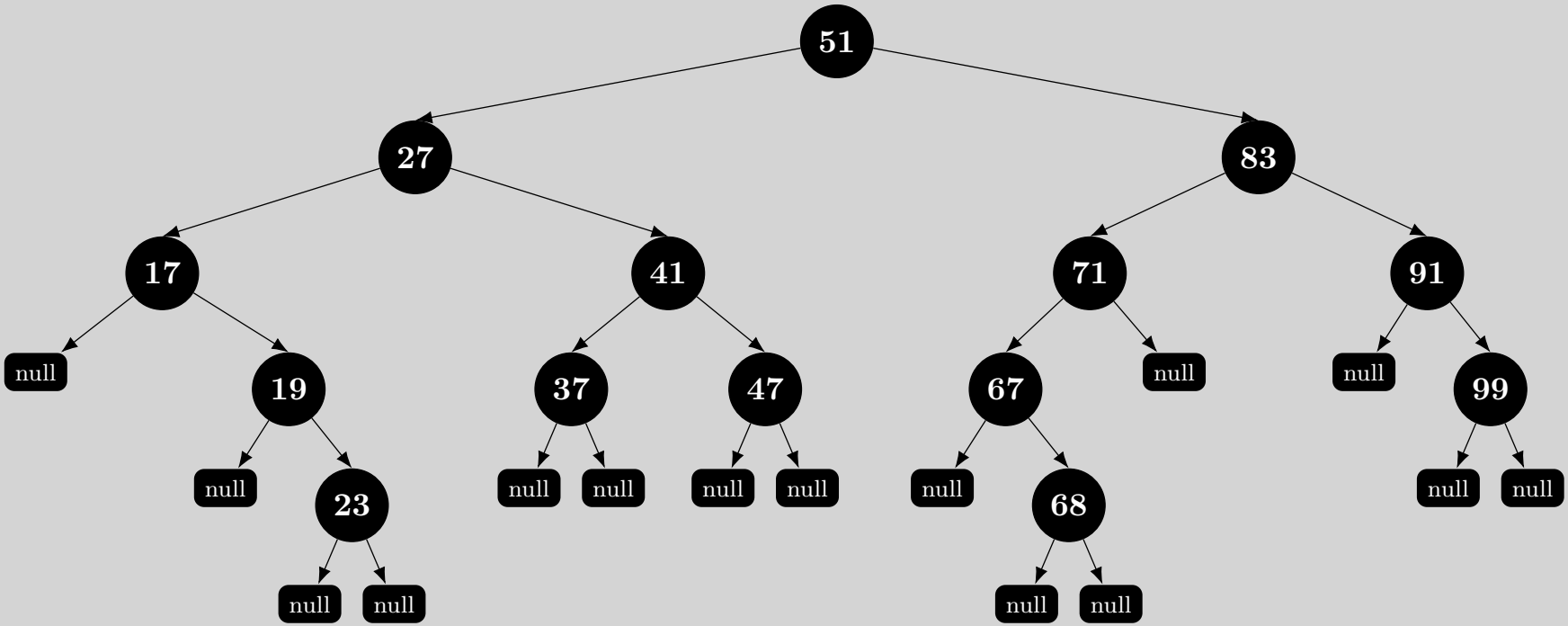
- Várias aplicações mantêm conjuntos de chaves que sofrem alterações.
- Para usar busca binária, cada inserção e remoção precisa comparar e mover até n chaves para manter um vetor ordenado.
- Árvores de buscas binárias e suas versões balanceadas podem ser mais adequadas nessas situações porque implementam busca, inserção e remoção em tempo proporcional a $\log_2 n$.

Árvore de busca binária

- Uma árvore de busca binária é uma árvore binária em que cada nó armazena um registro de dados e tal que para todo nó u

  – a chave em u é maior que todas as chaves na subárvore enraizada no filho da esquerda de u e

  – a chave em u é menor que todas as chaves na subárvore enraizada no filho da direita de u.


51
27
83
17
41
71
91
null
19
37
47
67
null
null
99
null
null
23
null
null
null
null
null
68
null
null
null
null
null
null

Operações

- A operações típicas na árvore são busca, inserção, remoção, mínimo, máximo, predecessor, sucessor.
- Vamos supor que cada nó u tem como atributos a chave $u.key$ e os apontadores $u.parent$, $u.left$ e $u.right$.
- Vamos supor que uma árvore T tem como atributo um apontador para sua raiz $T.root$.

Busca por uma chave k

- A busca começa na raiz de uma árvore e retorna um apontador para o nó u que contém k ou NULL.

```
SEARCH(u, k)
1  if u == NULL or u.key == k
2        return u
3  if k < u.key
4        return SEARCH(u.left, k)
5  else
6        return SEARCH(u.right, k)
```

Busca iterativa

- A busca pode ser convertida facilmente em um procedimento iterativo.

```
SEARCH(u, k)
1  while u ≠ NULL and u.key ≠ k
2        if k < u.key
3              u = u.left
4        else
5              u = u.right
6  return u
```

Mínimo

- O nó com a menor chave em uma árvore pode ser encontrado tomando sempre o apontador da esquerda.

```
MINIMUM(u)
1  if u == NULL
2        return NULL
3  while u.left ≠ NULL
4        u = u.left
5  return u
```

Máximo

- O nó com a maior chave em uma árvore pode ser encontrado tomando sempre o apontador da direita.

```
MAXIMUM(u)
1  if u == NULL
2        return NULL
3  while u.right ≠ NULL
4        u = u.right
5  return u
```

Sucessor de u

- Se u tem um filho da direita então o sucessor de u é o mínimo na subárvore enraizada em $u.right$.
- Se u não tem um filho da direita então u é o máximo de alguma subárvore. Seja r a raiz da subárvore de maior altura em que u é o máximo.
- Se r tem pai então o pai de r é o sucessor de u. Senão u não tem sucessor.

Sucessor

```
SUCCESSOR(u)
1  if u.right ≠ NULL
2        return MINIMUM(u.right)
3  p = u.parent
4  while p ≠ NULL and u == p.right
5        u = p
6        p = p.parent
7  return p
```

Predecessor de u

- Se u tem um filho da esquerda então o predecessor de u é o máximo na subárvore enraizada em $u.left$.
- Se u não tem um filho da esquerda então u é o mínimo de alguma subárvore. Seja r a raiz da subárvore de maior altura em que u é o mínimo.
- Se r tem pai então o pai de r é o predecessor de u. Senão u não tem predecessor.

Predecessor

```
PREDECESSOR(u)
1  if u.left ≠ NULL
2        return MAXIMUM(u.left)
3  p = u.parent
4  while p ≠ NULL and u == p.left
5        u = p
6        p = p.parent
7  return p
```

Inserção

```
INSERT(T, z)
    // z is a new node with the key to be inserted
 1  u = T.root
 2  p = NULL
 3  while u ≠ NULL
 4        p = u
 5        if z.key < u.key
 6              u = u.left
 7        else
 8              u = u.right
 9  if p == NULL
10        T.root = z
11  elseif z.key < p.key
12        p.left = z
13  else
14        p.right = z
```

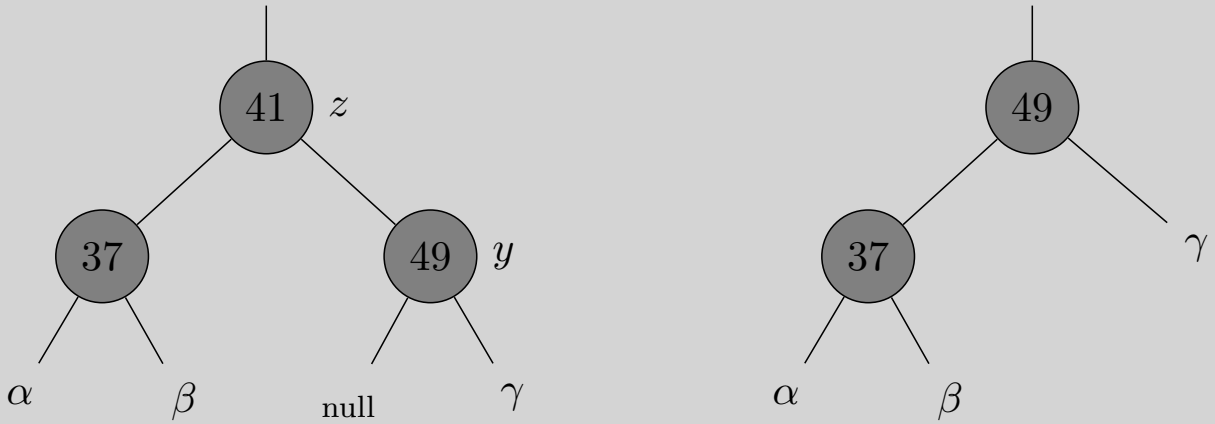
Remoção

- Remover uma folha é trivial.
- Remover um nó que só tem um filho também é fácil, basta "trocá-lo" pelo filho.
- Para remover um nó z com dois filhos podemos "trocá-lo" com seu sucessor e remover o sucessor.
  - O sucessor é o mínimo na subárvore direita. Sendo mínimo tem zero ou um filho, e é fácil removê-lo.
  - A troca tem dois casos: o sucessor é filho de z ou o sucessor não é filho de z.

- Se os nós são pequenos então podemos trocar o conteúdo dos nós e manter os apontadores. Senão, ajustamos os apontadores e mantemos o conteúdo nos mesmos nós.

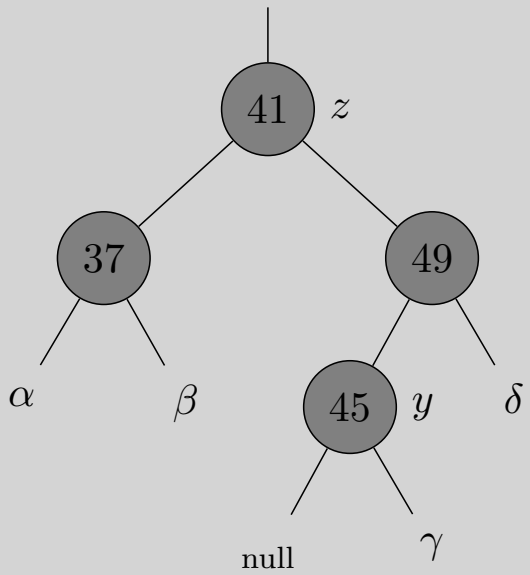
Remoção de nó com dois filhos, sucessor é o próprio filho

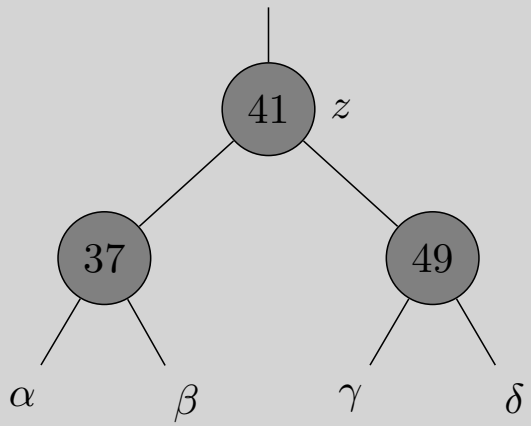
- z é substituído por y.

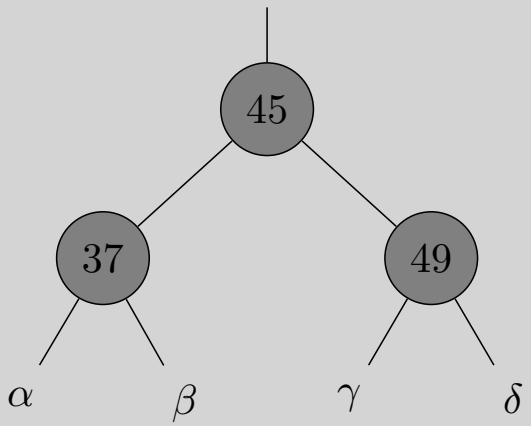


Remoção nó com dois filhos, sucessor não é o próprio filho

- y é substituído por $y.right$.
- z é substituído por y.







Remoção

```
DELETE(T, z)
 1  if z.left == NULL
 2       REPLACE(T, z, z.right)
 3  elseif z.right == NULL
 4       REPLACE(T, z, z.left)
 5  else
 6       y = MINIMUM(z.right)
 7       if y.parent ≠ z
 8            REPLACE(T, y, y.right)
 9            y.right = z.right
10            y.right.parent = y
11       REPLACE(T, z, y)
12       y.left = z.left
13       y.left.parent = y
```

Remoção

Replace substitui a subárvore enraizada em u pela subárvore enraizada em v.

```
REPLACE(T, u, v)
1  if u.p == NIL
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else
6        u.p.right = v
7  if v ≠ NIL
8        v.p = u.p
```

Número de comparações

- A busca percorre um caminho da raiz até um nó.
- Quando a árvore está degenerada em uma estrutura linear a função SEARCH faz até $3n - 1$ comparações.
- Quando a árvore é completa a função SEARCH faz no máximo $3(\lfloor(\log_2 n)\rfloor + 1) - 1$ comparações.
- As demais operações são similares e realizam um volume de trabalho parecido.

Chaves em ordem ou em ordem inversa

- É comum querermos recuperar as chaves em ordem ou em ordem reversa.
- Vimos que podemos usar a representação costurada com um pequeno overhead de memória (2 bits por nó) para isso de forma bastante eficiente.
- É fácil ver que manter a representação costurada não acrescenta muita dificuldade ou custo.

Range query

- Uma *range query* é uma busca por todas as chaves em um intervalo $[k_1, k_2]$, $k_1 < k_2$.
- Uma busca desse tipo pode ser resolvida em tempo proporcional a $\ell og_2 n + k$ onde k é o número de chaves no intervalo.
- A busca começa buscando por k_1 e k_2 simultaneamente até encontrar o primeiro nó em que k_1 vai para a esquerda e k_2 vai para a direita. Então durante o caminho de k_1, sempre que a aresta para a esquerda for tomada, todos os nós na subárvore direita são reportados. Para k_2 é simétrico.

Range query 12,45