MC202 - Estruturas de Dados

Guilherme P. Telles

IC

14 de Outubro de 2019

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides foram escritos usando português anterior à reforma ortográfica de 2009.

Parte I

Busca

90% of everything is CRUD[1]

- Create, retrieve, update, delete.
- Atualizar, remover e criar em geral envolvem recuperar primeiro.
- Recuperar é uma operação freqüente em computação e é uma parte importante de outros problemas mais complexos.

[1] fortune

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 usa expressões do tipo "a chave" como sinônimo de "registro que tem a chave".

Roteiro

- Chaves em um intervalo contínuo e 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 e denso de chaves

- Se as chaves são inteiros e formam um intervalo contínuo e denso (p.ex. entre 49 e 1051, entre k e ℓ ou entre 1 e n), então uma organização contínua na memória (como a de um vetor) permite resolver o problema da busca com apenas um cálculo de endereço e um acesso à memória.
- Tipicamente as chaves não formam um intervalo contínuo nem denso.

Busca seqüencial

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

Busca seqüencial (2cmps)

SEQUENTIAL-SEARCH$(A[1,n],k)$

```
i = 1
while (i <= n)
    if A[i] == k
        return i
return 0
```

Busca seqüencial (1cmp)

```
SEQUENTIAL-SEARCH(A[1, n], k)
A[n + 1] = k
i = 1
while (true)
    if A[i] == k
        if i < n + 1
            return i
        else
            return 0
    i = i + 1
```

Número de comparações de 1cmp

- Se a busca tiver sucesso, são realizadas i comparações. Caso contrário são realizadas $n + 1$ comparações.
- 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 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 suficiente.
- Normalmente tais probabilidades não são conhecidas ou mudam com o tempo.

Lista ou array auto-organizável

- Usa alguma estratégia de permutação dos registros para aproveitar a localidade das buscas.
    - Elas movem o registro que acabou de ser recuperado um certo número de posições em direção ao início, sem modificar a ordem relativa dos demais registros.

- As mais usadas incluem:
  - Move-to-front (MTF): quando um registro é recuperado ele é movido para o início.
  - Transpose: quando um registro é recuperado 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 recuperado o contador é incrementado e ele é movido para uma posição anterior a todos os registros com contador menor ou igual ao dele.
- Nem todas as estratégias são boas em vetores, por movimentarem muitas chaves.

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.
- A observação é que depois de comparar k contra k_i,
  ou $\mathbf{k} < \mathbf{k_i}$ e as chaves $k_i, \ldots, k_n$ não precisam mais ser consideradas,
  ou $\mathbf{k} = \mathbf{k_i}$,
  ou $\mathbf{k} > \mathbf{k_i}$ e as chaves $k_1, \ldots, k_i$ não precisam mais 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)

$\textsc{Binary-Search}(A, \ell, r, k)$

if $\ell > r$
    return 0
$m = \lfloor (\ell + r)/2 \rfloor$
if $A[m] == k$
    return m
elseif $k < A[m]$
    return $\textsc{Binary-Search}(A, \ell, m-1, k)$
else
    return $\textsc{Binary-Search}(A, m+1, r, k)$

Busca binária (2cmps)

Binary-Search(A, ℓ, r, k)

if $\ell == r$
    if $A[\ell] == k$
        return ℓ
    else
        return 0
else
    $m = \lceil (\ell + r)/2 \rceil$
    if $k < A[m]$
        return Binary-Search$(A, \ell, m - 1, k)$
    else
        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 $2(\lfloor \log_2 n \rfloor + 2)$.

Busca por interpolação

- A busca por interpolação tenta reduzir o espaço de busca mais rapidamente, estimando a posição de k supondo que as chaves estejam distribuídas de forma razoavelmente uniforme ao longo do intervalo $[k_\ell, k_r]$.
- Para um intervalo entre k_ℓ e k_r a busca por interpolação compara k contra

$$\ell + \lceil (k - k_\ell) \frac{r - \ell}{k_r - k_\ell} \rceil.$$

Busca por interpolação

$\textsc{Interpolation-Search}(A, \ell, r, k)$

if $A[\ell] == k$
    return ℓ
if $\ell == r$ or $A[\ell] == A[r]$
    return 0
$p = \ell + \lceil (k - A[\ell])(r - \ell)/(A[r] - A[\ell]) \rceil$
if $k < A[p]$
    return $\textsc{Interpolation-Search}(A, \ell, p - 1, k)$
else
    return $\textsc{Interpolation-Search}(A, p, r, k)$

Número de comparações

- Em média essa busca reduz o tamanho do vetor para $\sqrt{n}$ a cada passo e faz $\lg \lg n$ comparações de chaves.
- Na prática, mesmo quando as chaves têm uma distribuição adequada, o número de operações aritméticas maior faz com que a busca por interpolação seja melhor apenas para tamanhos de entrada muito grandes.

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.

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 são mais adequadas nessas situações porque implementam busca, inserção e remoção em tempo proporcional a $\log_2 n$.

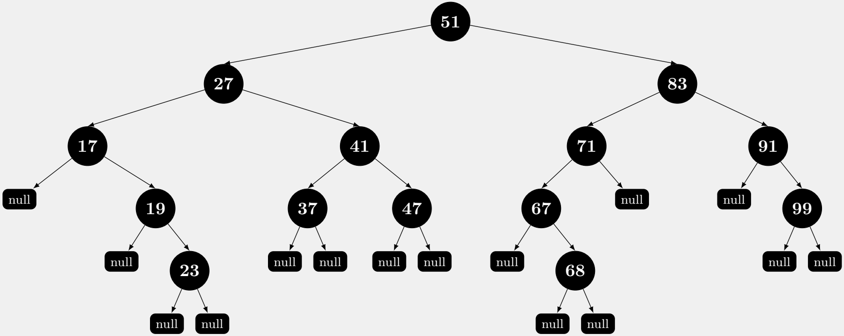
Árvore binária de busca

- Uma árvore binária de busca (BST) é uma árvore binária em que para todo nó u com chave $u.k$ e filhos $u.l$ e $u.r$ vale

  $u.k$ é maior que todas as chaves na subárvore enraizada em $u.l$ e

  $u.k$ é menor que todas as chaves na subárvore enraizada em $u.r$.

Árvore binária de busca



Operações

- A operações típicas na árvore são busca, mínimo, máximo, predecessor, sucessor, inserção e remoção.

Operações

- A operações típicas na árvore são busca, mínimo, máximo, predecessor, sucessor, inserção e remoção.
- Deste ponto em diante vamos supor que uma BST é implementada explicitamente, isto é, com apontadores.
- Cada nó u tem como atributos a chave $u.key$ e os apontadores $u.parent$, $u.left$ e $u.right$.
- Cada 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.

$\textsc{Search}(u, k)$

if $u == \text{NULL}$ or $u.key == k$
    return u
if $k < u.key$
    return $\textsc{Search}(u.left, k)$
else
    return $\textsc{Search}(u.right, k)$

Busca iterativa

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

$\textsc{Search}(u, k)$

while $u \neq \text{NULL}$ **and** $u.key \neq k$
    if $k < u.key$
        $u = u.left$
    else
        $u = u.right$
return u

Mínimo

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

```
MINIMUM(u)
if u == NULL
    return NULL
while u.left ≠ NULL
    u = u.left
return u
```

Máximo

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

$\textsc{Maximum}(u)$

if $u == \text{NULL}$
    return NULL
while $u.right \neq \text{NULL}$
    $u = u.right$
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)
if u.right ≠ NULL
    return MINIMUM(u.right)
p = u.parent
while p ≠ NULL and u == p.right
    u = p
    p = p.parent
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

$\textsc{Predecessor}(u)$

if $u.\mathit{left} \neq \textsc{null}$
    return $\textsc{Maximum}(u.\mathit{left})$
$p = u.\mathit{parent}$
while $p \neq \textsc{null}$ and $u == p.\mathit{left}$
    $u = p$
    $p = p.\mathit{parent}$
return p

Inserção

```
INSERT(T, z)
    // z is a new node with the key to be inserted
    u = T.root
    p = NULL
    while u ≠ NULL
        p = u
        if z.key < u.key
            u = u.left
        else
            u = u.right
    if p == NULL
        T.root = z
    elseif z.key < p.key
        p.left = z
    else
        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.

Remoção

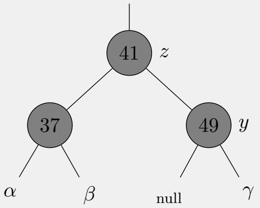
- Remover uma folha é trivial.
- Remover um nó que só tem um filho também é fácil, basta "trocá-lo" pelo filho.
- Para remover um nó z com dois filhos podemos "trocá-lo" com seu sucessor e remover o sucessor.
  - O sucessor é o mínimo na subárvore direita. Sendo mínimo tem zero ou um filho, e é fácil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

```
DELETE(T, z)
if z.left == NULL
    REPLACE(T, z, z.right)
elseif z.right == NULL
    REPLACE(T, z, z.left)
else
    y = MINIMUM(z.right)
    if y.parent ≠ z
        REPLACE(T, y, y.right)
        y.right = z.right
        y.right.parent = y
    REPLACE(T, z, y)
    y.left = z.left
    y.left.parent = y
```

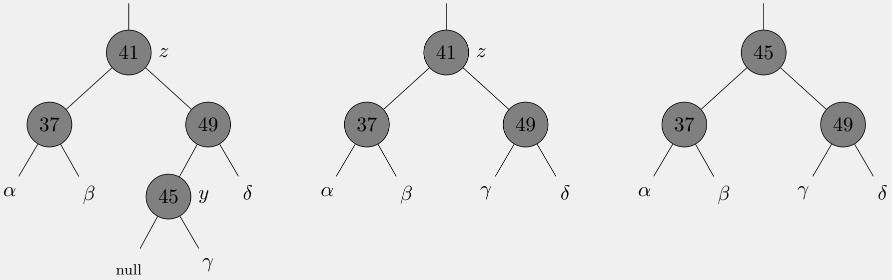
Remoção de nó com dois filhos, sucessor é o próprio filho

- 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.



Número de comparações

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

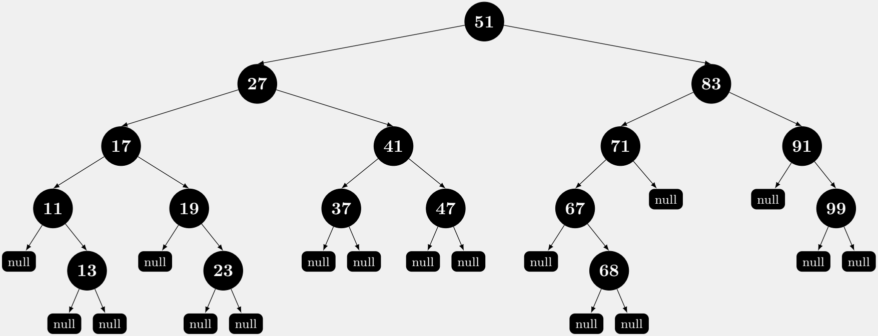
Chaves em ordem ou em ordem inversa

- 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 $log_2 n + k$ 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



Árvores balanceadas

Árvores balanceadas

- Árvores binárias de busca balanceadas fazem algum trabalho adicional para garantir que busca, inserção e remoção façam $\log_2 n$ comparações no pior caso.
- São balanceadas por altura, pelo número de nós em subárvores, pelo rank (estimativa do tamanho das subárvores) ou por algum outro critério.
- As mais usadas são AVL, vermelha-e-preta e splay.

Árvores AVL

Árvore AVL

- G.M. Adelson-Velskii e E.M. Landis, 1962.
- É uma árvore binária de busca com balanceamento de altura.
- Em uma árvore AVL a diferença das alturas das subárvores que são filhas de qualquer nó é no máximo 1.
- Foi a primeira estrutura-de-dados desse tipo.

Fator de balanceamento

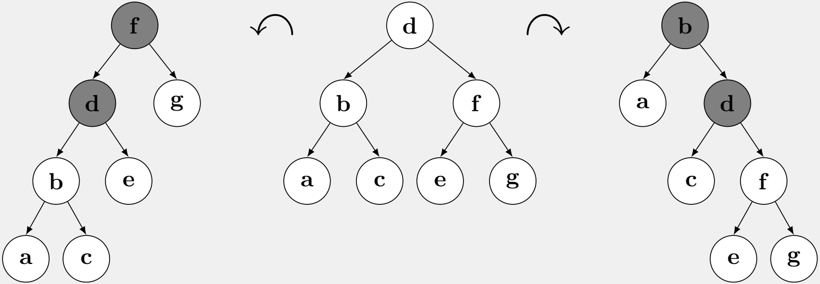
- O fator de balanceamento (fb) de um nó u de uma árvore binária é

$$altura(u.left) - altura(u.right).$$

- Em uma árvore AVL, o fb de todo nó está em $\{-1, 0, +1\}$.
- A árvore armazena o fb para cada nó.

Rotação

- Uma rotação é uma operação que altera a estrutura de uma árvore binária sem interferir na ordem das chaves.
- Rotação à esquerda em $(d, d.left)$ ($\curvearrowleft$) e rotação à direita em $(d, d.right)$ ($\curvearrowright$):



Inserção

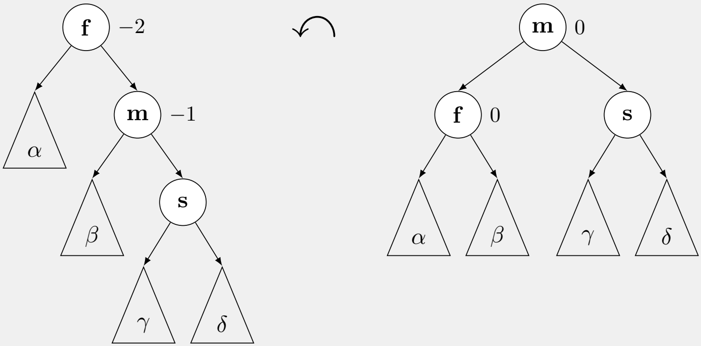
- A inserção avança para baixo, como na árvore binária de busca.
- Depois que o novo nó é adicionado, o mesmo caminho é percorrido em direção à raiz, atualizando os fbs com base na variação de altura das subárvores.
- Se um nó u com fb -2 ou $+2$ for encontrado então a subárvore enraizada em u é reorganizada.
- Só há necessidade de ajustar uma vez.
- Mas a atualização dos fbs deve avançar até a raiz.
- São 4 casos cuja solução são uma ou duas rotações.

Left-Rotate

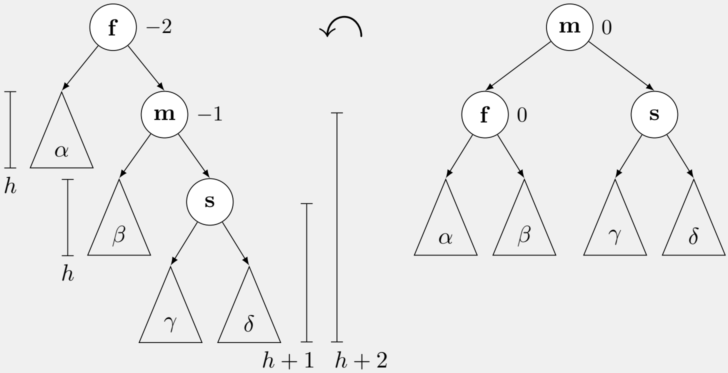
Left-Rotate(T, x)

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.parent = x
y.parent = x.parent
if x.parent == T.nil
    T.root = y
elseif x == x.parent.left
    x.parent.left = y
else
    x.parent.right = y
y.left = x
x.parent = y
```

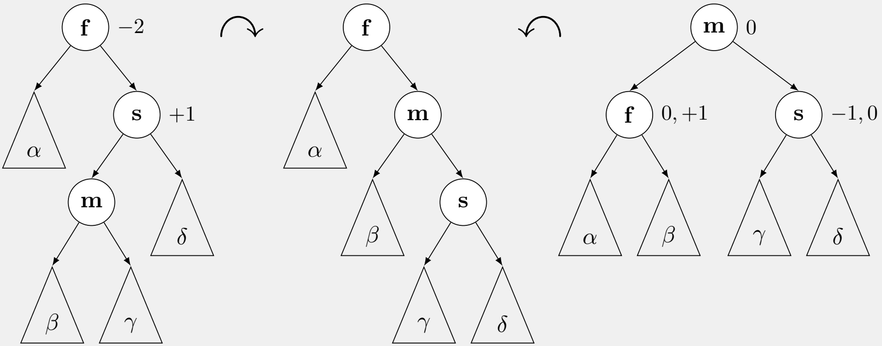
Right/Right



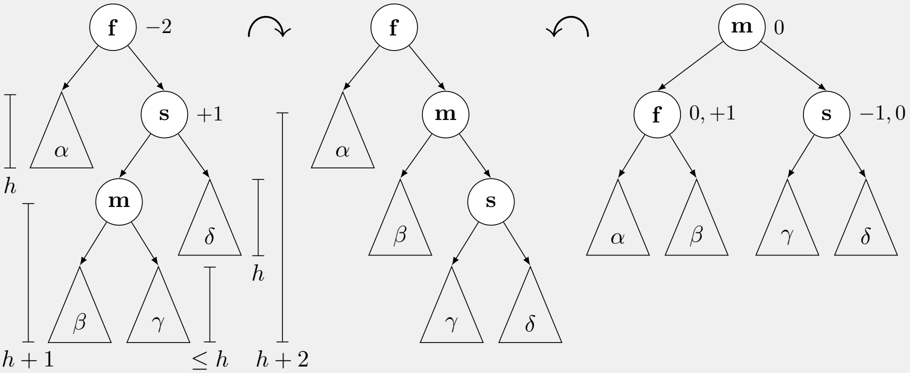
Right/Right



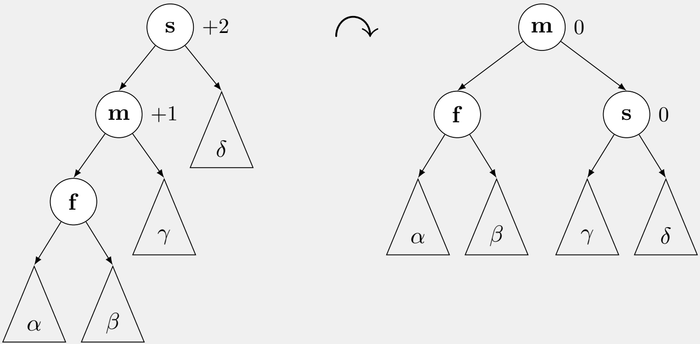
Right/Left



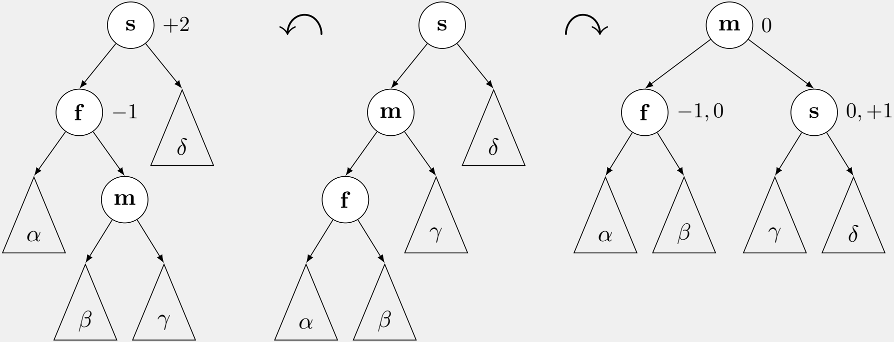
Right/Left



Left/Left



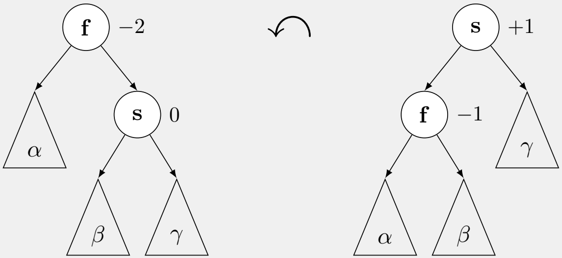
Left/Right



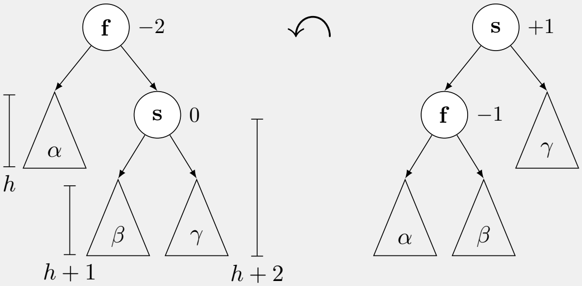
Remoção

- A remoção começa como na árvore de busca.
- Depois da remoção de um nó os ancestrais dele têm que ter seu fb ajustado.
- Diferentemente da inserção, o ajuste pode ter que ser feito mais de uma vez ao longo do caminho até a raiz.
- Além dos 4 casos anteriores, mais dois são possíveis.

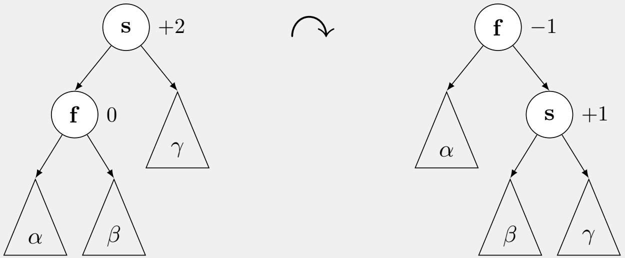
-2/0



-2/0

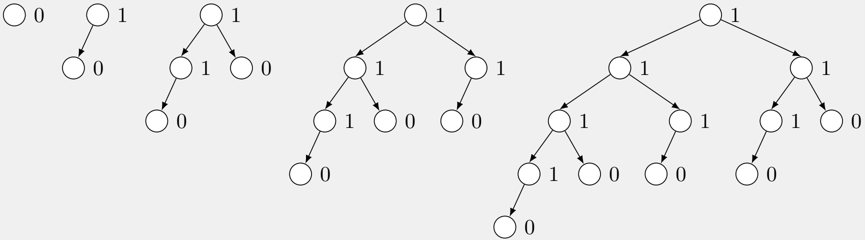


$+2/0$



Número mínimo de nós

- A altura de uma árvore AVL é no mínimo $\log_2 n$.
- Seja T_h uma árvore AVL com o menor número possível de nós.
- Não é difícil ver que $T_h = T_{h-1} + T_{h-2} + 1$ em geral.



Árvore de Fibonacci

- O número de nós em uma árvore AVL de altura h é no mínimo igual ao número de nós na árvore de Fibonacci de ordem $h + 1$.
- Uma árvore de Fibonacci de ordem k chamada F_k é construída assim:

1. F_0 é a árvore vazia,
2. F_1 tem um único nó e
3. se $k > 2$ então F_k tem uma raiz conectando F_{k-1} como subárvore esquerda e F_{k-2} como subárvore direita.
4. Pode-se mostrar por indução que o número de nós em F_h é $f_{h+2} - 1$.

Custo

- Usando a aproximação para f_i,

$$f_i \approx ((1+\sqrt{5})/2)^i/\sqrt{5},$$

obtém-se que a altura de uma árvore AVL é no máximo $1,4404\log_2(n+2)$.

- Logo, o número de comparações para uma busca é proporcional a $\log_2 n$.
- A inserção e a remoção são uma busca seguida de rotações ao longo de um caminho de tamanho proporcional $\log_2 n$. Cada rotação faz um número constante de operações, resultado em custo total proporcional a $\log_2 n$.

Árvore Splay

Árvore Splay

- D.M. Sleator e R.E. Tarjan, 1985.
- Uma splay tree é uma árvore binária de busca em que a chave mais recentemente acessada está na raiz. A idéia é favorecer a localidade de acessos aos registros.
- Não é exatamente uma árvore balanceada, é chamada de auto-ajustável.
- A reestruturação é feita de forma que, para uma seqüência suficientemente longa de buscas, cada busca faça um número de operações proporcional a $\log n$ em média.
- A reestruturação não garante que a altura da árvore seja limitada.

Splay tree

- Para seqüências longas de acesso, o pior desempenho possível não é muito pior que árvores balanceadas.
- Experimentos mostraram que o desempenho em seqüências reais é superior ao desempenho da AVL e da VP.
- Não registra nenhum dado adicional (cor, fb, número de filhos etc).
- É simples.
- Sofre mais ajustes na estrutura, inclusive durante a busca.
- Algumas operações sobre a árvore podem ser lentas individualmente, o que é uma limitação para aplicações que precisam de uma garantia de tempo.

Splaying

- A operação de reestruturação se chama splaying, e é feita para cada busca.
- Ela faz rotações ao longo da volta de um caminho de busca.
- A profundidade dos nós ao longo do caminho de busca é diminuída mais-ou-menos pela metade.
- As rotações são feitas em pares, em uma ordem que depende da estrutura do caminho da busca.
- Mover a chave acessada para a raiz com rotações individuais pode levar a uma seqüencia de acessos que faz um número de operações proporcional a n em média, por isso a Splay faz rotações em pares.

Splaying

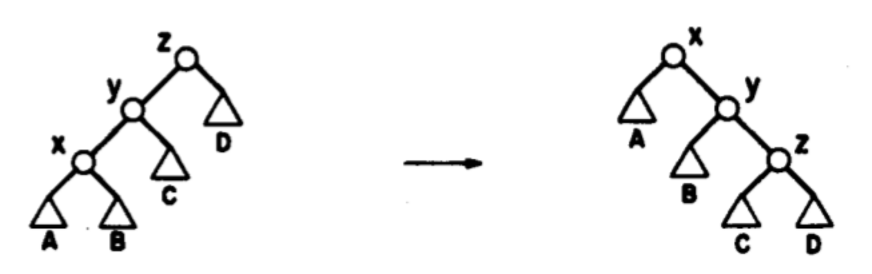
- Sejam x o nó com a chave buscada, $p(x)$ o pai de x e $g(x)$ o avó de x.
- Cada passo da splaying é repetido até que x se torne a raiz da árvore.
- São três casos.

Caso 1: zig

- Se $p(x)$ é a raiz da árvore então rotacione a aresta que liga x a $p(x)$. Esse passo é terminal.

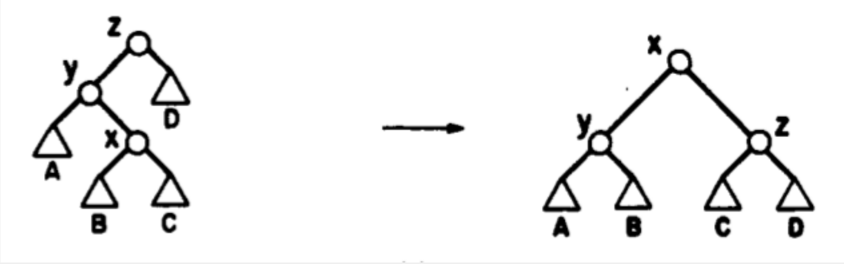
Caso 2: zig-zig

- Se $p(x)$ não é a raiz da árvore e ambos x e $p(x)$ são filhos da esquerda (ou da direita) então rotacione a aresta que liga $p(x)$ a $g(x)$ e depois rotacione a aresta que liga x a $p(x)$.

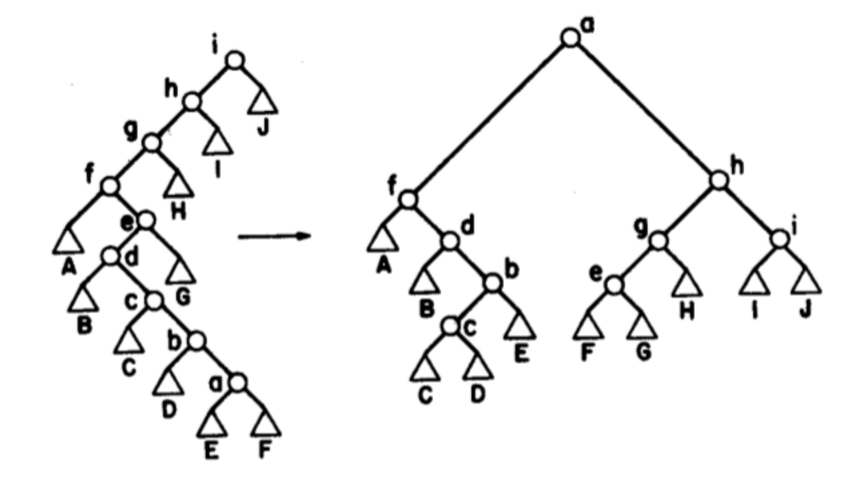


Caso 3: zig-zag

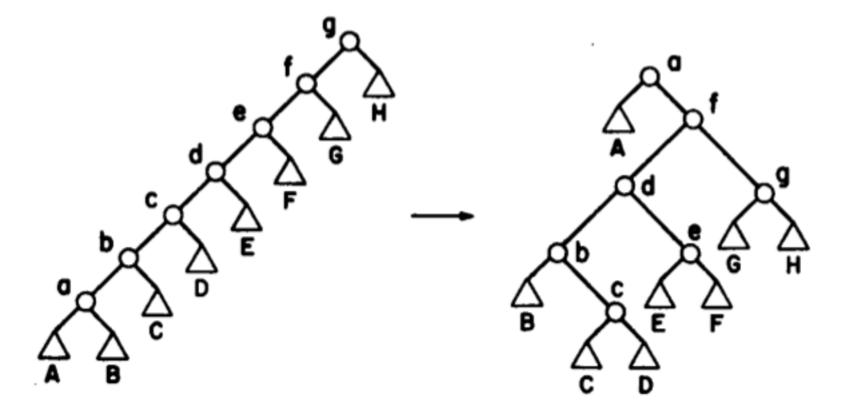
- Se $p(x)$ não é a raiz da árvore e x é filho da esquerda e $p(x)$ é filho da direita ou vice-versa, então rotacione a aresta que liga x a $p(x)$ e depois rotacione a aresta que liga x ao novo $p(x)$.



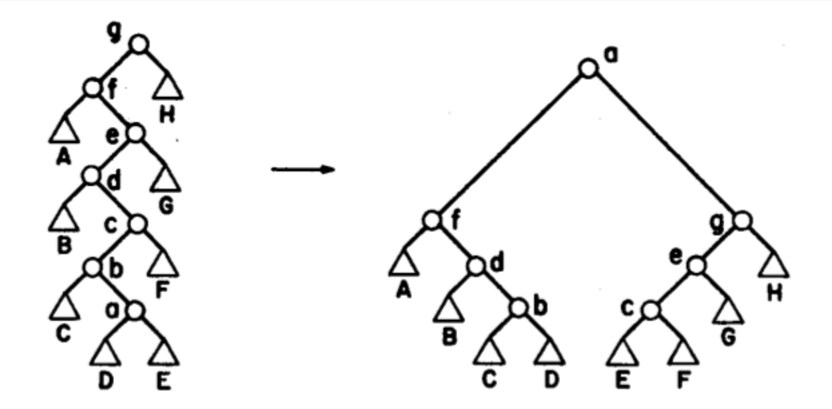
Exemplo



Exemplo



Exemplo



Busca

- A busca por x na splay começa na raiz.
- Se for bem sucedida então a busca é seguida por uma splaying de x.
- Se for mal sucedida então a busca é seguida por uma splaying do último nó não-nulo visitado pela busca.

Inserção e remoção

- A inserção de x é feita como na árvore binária de busca e é seguida de um splay em x.
- A remoção de x é feita como na árvore binária de busca e é seguida de um splay em $p(x)$.

Observações

- A operação splaying pode ser implementada na volta da busca (bottom-up) ou pode ser implementada na própria descida para a busca (top-down).
- As operações de acesso reestruturam a árvore, o que dificulta a execução de consultas em paralelo.

Árvores Vermelhas e Pretas

Árvore vermelha e preta

- R. Bayer, 1972.
- A árvore vermelha e preta (ou rubro-negra, ou VP) usa cores nos nós para garantir que cada caminho na árvore é no máximo duas vezes maior que qualquer outro.

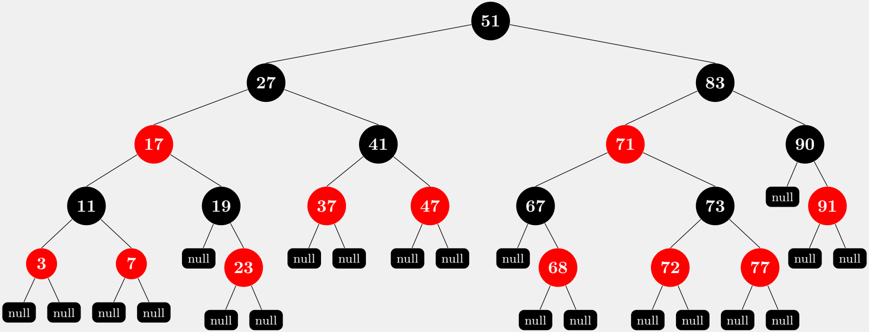
Árvore vermelha e preta

- Cada nó v contém os atributos $v.key$, $v.color$, $v.left$, $v.right$ e $v.parent$.
- Filhos ou pai inexistentes são representados por apontadores nulos.
- As folhas da árvore VP são todos os nós nulos. Os nós internos são todos aqueles que contém uma chave.

Árvore vermelha e preta

- Uma árvore VP é uma árvore binária de busca que satisfaz as seguintes propriedades:
  1. Todo nó é vermelho ou preto.
  2. A raiz é preta.
  3. Toda folha (nula) é preta.
  4. Os filhos de todo nó vermelho são pretos.
  5. Para todo nó v, todos os caminhos de v até uma folha que descende dele têm o mesmo número de nós pretos.

Exemplo


51
27
83
17
41
71
90
11
19
37
47
67
73
null
91
3
7
null
23
null
null
null
null
null
68
72
77
null
null
null
null
null
null
null
null
null
null
null
null
null
null
null
null

- Mais relaxada que a AVL. A altura de uma árvore VP com n nós é no máximo $2\log_2(n+1)$.
- As operações normalmente afetam subárvores menores mas são mais complicadas de implementar que nas árvores AVL.

Hashing

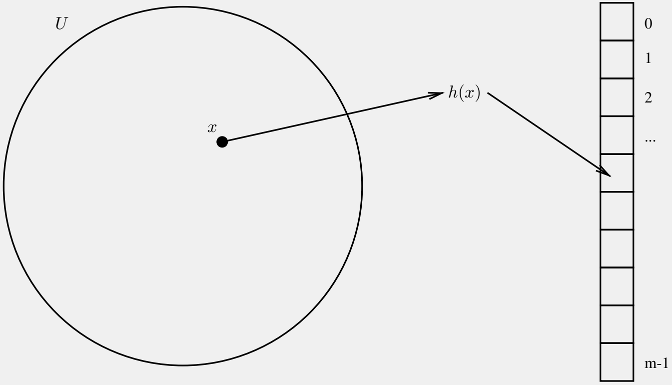
Hashing (scatter storage)

- Hashing é uma família de técnicas de busca baseadas na idéia de usar uma função para calcular posições para armazenar e recuperar as chaves.
- Gastando algum espaço adicional, hashing resolve a busca com um número constante de acessos em média.

Hashing

- Suponha que temos um universo de chaves U e que queremos armazenar n chaves, sendo que $n << |U|$.
- Por exemplo, usando o CPF como chave, queremos armazenar os registros dos alunos da U.
- A estratégia usa um vetor (tabela de hashing) de tamanho m ($m > n$) e uma função (função de hashing) que mapeia chaves de U em posições do vetor.

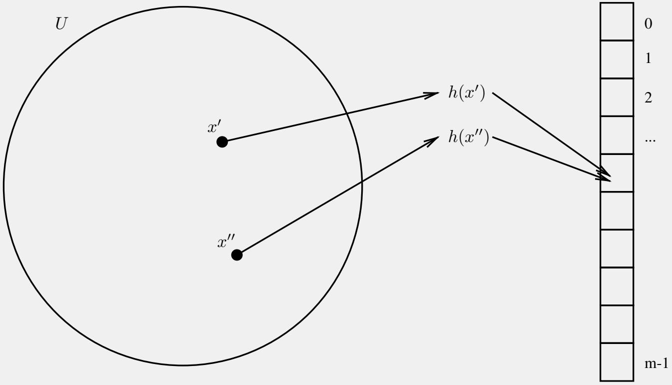
Hashing



Colisões

- Mesmo para uma tabela grande em relação ao número de chaves que serão armazenadas, a probabilidade de que duas chaves sejam mapeadas na mesma posição da tabela tem que ser considerada.
- Dizemos que acontece uma *colisão* quando duas chaves são mapeadas na mesma posição da tabela.
- Chamamos duas chaves que colidem de *sinônimas*.

Colisão



Hashing perfeito

- Se as chaves são conhecidas antecipadamente e não mudam, o ideal é usar uma função para mapear n chaves em exatamente n posições consecutivas e distintas na memória. Esse mapeamento é chamado de *hashing perfeito*.
- Existe um algoritmo que calcula uma função de hashing perfeita para n chaves em tempo proporcional a n, produzindo uma tabela de tamanho cn, onde c é uma constante inteira pequena.

Não tão perfeito

- Na maioria das aplicações as chaves não são conhecidas antecipadamente e mudam ao longo do tempo.
- Mas se tivermos uma função que espalha bem as n chaves em uma tabela de tamanho m maior que n, teremos um bom desempenho médio.

Função e colisões

- Então temos que escolher uma função uma técnica para resolver as colisões.
  - Funções: métodos da divisão e multiplicação.
  - Colisões: encadeamento e sondagem (e variações).
- Há outros métodos, mas esses são considerados os que valem a pena na prática.

Função de hashing

- Características de uma boa função de hashing:
  1. poder ser calculada rapidamente.
  2. espalhar bem as chaves, minimizando colisões.
  3. usar todos os bits da chave, já que elas raramente são aleatórias.

Funções: método da divisão

- $h(k) = k \bmod m$

Funções: método da divisão

- $h(k) = k \bmod m$
- Uma boa escolha prática de m é um número primo que não é próximo de uma potência de 2.
- Escolhas ruins são pares ($k \bmod m$ é par se k é par e é impar se k é impar e isso introduz um viés no espalhamento para várias seqüências de chaves), potências de 2 ($k \bmod m$ vai ter apenas parte dos bits de k) etc.

Funções: método da multiplicação

- $h(k) = \lfloor m(Ak \bmod 1) \rfloor$,
  onde A é uma constante no intervalo $(0, 1)$ e
  $Ak \bmod 1$ é a parte fracionária de Ak.

Funções: método da multiplicação

- $h(k) = \lfloor m(Ak \bmod 1) \rfloor$,
  onde A é uma constante no intervalo $(0, 1)$ e
  $Ak \bmod 1$ é a parte fracionária de Ak.
- A escolha de m não é crítica.
- A de A também não. Sugere-se usar
  $A \approx (\sqrt{5} - 1)/2 = 0.618033988\ldots$
- Espalha melhor chaves consecutivas e relacionadas por uma progressão geométrica.
- Normalmente é mais rápida de computar porque pode ser implementada sem usar divisão.

Método da multiplicação: implementação

- $h(k) = \lfloor m(Ak \bmod 1) \rfloor$
- Seja w o tamanho da palavra do computador.
- O tamanho da tabela m é escolhido como uma potência de 2, 2^z.
- Escolhe-se uma constante inteira S relativamente prima a 2^w.
- Fazendo $A = \frac{S}{2^w}$, A é igual a S com um ponto decimal à esquerda.
- Computar Sk é o mesmo que computar $A2^w k$. Esse número tem $2w$ bits. A parte fracionária de Ak está na palavra menos significativa.
- Multiplicar a parte fracionária de Ak por $m = 2^z$ e tomar o piso é equivalente a formar um número com os z bits iniciais da parte fracionária de Ak.
- $h(k)$ consiste então dos z bits iniciais da palavra menos significativa do produto Sk.

Método da multiplicação: implementação

$w = 16$
$m = 2^{10}$
$S = 40503 = 1001\ 1110\ 0011\ 0111$
$k = 249$
$Sk = 10085247 = 0000\ 0000\ 1001\ 1001\ 1110\ 0011\ 0111\ 1111$
$h(249) = 1110\ 0011\ 01 = 909$

Colisões: encadeamento

- No encadeamento, as chaves sinônimas são armazenadas em listas encadeadas fora da tabela.
- Usa memória para $m + n$ apontadores e n chaves.
- Admite overflow.
- As listas podem ser mantidas ordenadas ou podemos usar move-to-front.
- Registros podem ser removidos facilmente.

Custo

- Para uma tabela com fator de carga $\alpha = \frac{n}{m}$, o número de acessos esperado é $1 + \alpha$ para uma busca bem sucedida ou para uma busca mal sucedida.
- No pior caso o número de acessos pode ser n para uma busca.

Variação: encadeamento na própria tabela

- Cada posição da tabela armazena uma chave e um índice para outra posição da tabela (link).
- Chaves que colidem são colocadas em uma outra posição da tabela, encadeadas pelo link.
- A primeira chave de uma lista de sinônimos tem que estar na posição definida por $h()$.
  - Pode ser necessário reorganizar as chaves para garantir isso.
- Usa menos memória: para m chaves e m índices.
- Não admite overflow.

Variação: encadeamento na própria tabela com mistura de listas

- Quando uma chave colide, ela é colocada na lista que ocupa a posição dela, sem se preocupar se a a primeira chave de uma lista está na posição correta.
- As listas podem se misturar.
- Evita o trabalho de reorganização.
- A remoção não é mais direta. Mais sobre isso a seguir.

Colisões: sondagem ou endereçamento aberto

- As chaves são armazenadas na própria tabela.
- Usa memória para m chaves.
- As colisões são resolvidas procurando por outra posição na tabela.
- É preciso reservar algum valor de chave para marcar posições vazias na tabela.

Incrementos

- Vários incrementos são possíveis
- Vamos chamar de i o número de colisões de uma chave k contra posições já ocupadas na tabela. i assume os valores na seqüência $0, 1, \ldots, m-1$.

Incremento unitário

- $h(k, i) = (h(k) + i) \bmod m$.

Incremento unitário

- $h(k, i) = (h(k) + i) \bmod m$.
- Sofre o problema de clustering: chaves que colidem ocupam posições de outras chaves, o que aumenta a chance de novas colisões e a tendência de formação de blocos de chaves que colidiram.
- O problema se agrava com função baseada em divisão quando as chaves são consecutivas.
- Menos que 75% de ocupação ainda vai mais ou menos bem. Quando a tabela está cheia, com $n = m - 1$, o número médio de acessos é $(m + 1)/2$, equivalente a uma busca seqüencial.

Incremento não-unitário

- $h(k, i) = (h(k) + ic) \bmod m$.

Incremento não-unitário

- $h(k, i) = (h(k) + ic) \bmod m$.
- Não resolve muito, o clustering acontece c registros adiante.

Incremento quadrático

- $h(k, i) = (h(k) + ic_1 + i^2 c_2) \bmod m$.

Incremento quadrático

- $h(k,i) = (h(k) + ic_1 + i^2c_2) \bmod m$.
- Acontece um efeito chamado clustering secundário, já que duas chaves que colidem quando $i = 0$ irão colidir para os demais valores de i.

Incremento por hashing duplo

- $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$

Incremento por hashing duplo

- $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$
- Nessa função o incremento depende da chave. Isso reduz a probabilidade de colisões sucessivas para um par de chaves.
- h_1 e h_2 são tais que
  - $h_1(k)$ produz um número entre 0 e $m - 1$ e
  - $h_2(k)$ produz um número entre 1 e $m - 1$ que seja relativamente primo a m.

Hashing duplo: divisão

- Se m é primo, então
  $h_1(k) = k \bmod m$ e
  $h_2(k) = 1 + (k \bmod (m - 1))$
  são uma escolha razoável.
- Como $m - 1$ é par, então
  $h_1(k) = k \bmod m$ e
  $h_2(k) = 1 + (k \bmod (m - 2))$
  são escolhas melhores.

Hashing duplo: multiplicação

- Se $m = 2$ então h_2 pode ser calculada primeiramente pegando os k bits seguintes aos de h_1 e depois fazendo um ou bit-a-bit com 1.

Custo

- Em uma tabela com fator de carga $\alpha = \frac{n}{m} < 1$ e incremento unitário:
  - Número esperado de acessos para busca mal sucedida: $\frac{1}{1-\alpha}$.
  - Número esperado de acessos para busca bem sucedida: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.
- No pior caso o número de acessos pode ser n para uma busca.
- Na prática, mantendo α máximo em torno de 0.8 resulta em um bom desempenho, sem considerar remoções.

Remoções

- Um registro não pode ser removido, apenas marcado como removido.
  - É preciso reservar um valor de chave para marcar posições removidas na tabela.
  - Depois de uma seqüência longa de inserções e remoções as posições vazias vão desaparecendo e as seqüências de colisões se tornam mais longas, até o ponto em que a busca mal sucedida fará m acessos à tabela.
  - Pode-se tentar reorganizar as chaves sinônimas sempre que acontece uma remoção.
  - Quando há remoções será preciso reorganizar a tabela de tempos em tempos.

Variação de Brent

- Usa mais tempo na inserção para reduzir o número médio de acessos.
- Na prática dá resultados muito bons, mesmo com fatores de carga próximos a 1, mas ameniza pouco o impacto das remoções.
- Não há aumento do uso de memória na tabela.

Idéia

- Suponha que uma busca mal-sucedida pela chave k_t visitou as posições $p_0, p_1, \ldots, p_{t-1}, p_t$ e p_t está vazia.
- Sejam $k_0, k_1, \ldots, k_{t-1}$ as chaves com que k_t colidiu.
- Seja $c_i = h_2(T[p_i])$.

Idéia

- se $t = 0$ então k_t vai para p_0.
- se $t = 1$ então k_t vai para p_1.
- se $t >= 2$ e $p_0 + c_0$ está vazia então
  k_0 vai para $p_0 + c_0$ - aumenta 1
  k_t vai para p_0 - diminui pelo menos 2

Idéia

- senão se $t >= 3$ e $p_0 + 2c_0$ está vazia então
  k_0 vai para $p_0 + 2c_0$ - aumenta 2
  k_t vai para p_0 - diminui pelo menos 3
- senão se $t >= 3$ e $p_1 + c_1$ está vazia então
  k_1 vai para $p_1 + c_1$ - aumenta 1
  k_t vai para p_1 - diminui pelo menos 2

Idéia

- senão se $t >= 4$ e $p_0 + 3c_0$ está vazia então
  k_0 vai para $p_0 + 3c_0$ - aumenta 3
  k_t vai para p_0 - diminui pelo menos 4
- senão se $t >= 4$ e $p_1 + 2c_1$ está vazia então
  k_1 vai para $p_1 + 2c_1$ - aumenta 2
  k_t vai para p_1 - diminui pelo menos 3
- senão se $t >= 4$ e $p_2 + c_2$ está vazia então
  k_2 vai para $p_2 + c_2$ - aumenta 1
  k_t vai para p_2 - diminui pelo menos 2

Idéia

- senão se $t >= 5$ ...

Variação de Brent

- Em geral sejam $c_i = h_2(k_i)$ e $p_{i,j} = (p_i + jc_i) \bmod m$.

  Se $T[p_{i,j}]$ está ocupada para todos os índices i e j tais que $i + j < r$ e se $t \geq r + 1$ verificamos as posições

  $$T[p_{0,r}], T[p_{1,r-1}], \ldots, T[p_{r-1,1}].$$

  Se o primeiro espaço livre está em $p_{i,r-i}$ fazemos $T[p_{i,r-i}] = k_i$ e inserimos k_t na posição p_i.

Redimensionamento da tabela

- Se o conjunto de chaves ultrapassa o tamanho da tabela então é preciso redimensioná-la.
- Essa operação exige um volume razoável de processamento.
- O impacto sobre o redimensionamento para conjuntos de registros grandes que mudam com freqüência pode ser importante.

Busca em listas

Busca em listas

- Em uma lista, uma busca só pode avançar seqüencialmente.
- Supondo que as buscas em uma lista têm localidade, algumas estratégias de permutação foram propostas para melhorar o desempenho médio das buscas em listas.
- As estratégias de permutação mais usadas movimentam o registro que está sendo buscado, mas preservam a ordem relativa dos demais registros.

Permutações

- Move-to-front: quando um registro é recuperado ele é movido para o início da lista.
- Transpose: quando um registro é recuperado 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 recuperado 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: quando um registro é recuperado ele é movido k posições em direção ao início da lista. k pode ser um percentual da distância para o início da lista ou outra medida baseada na distância.