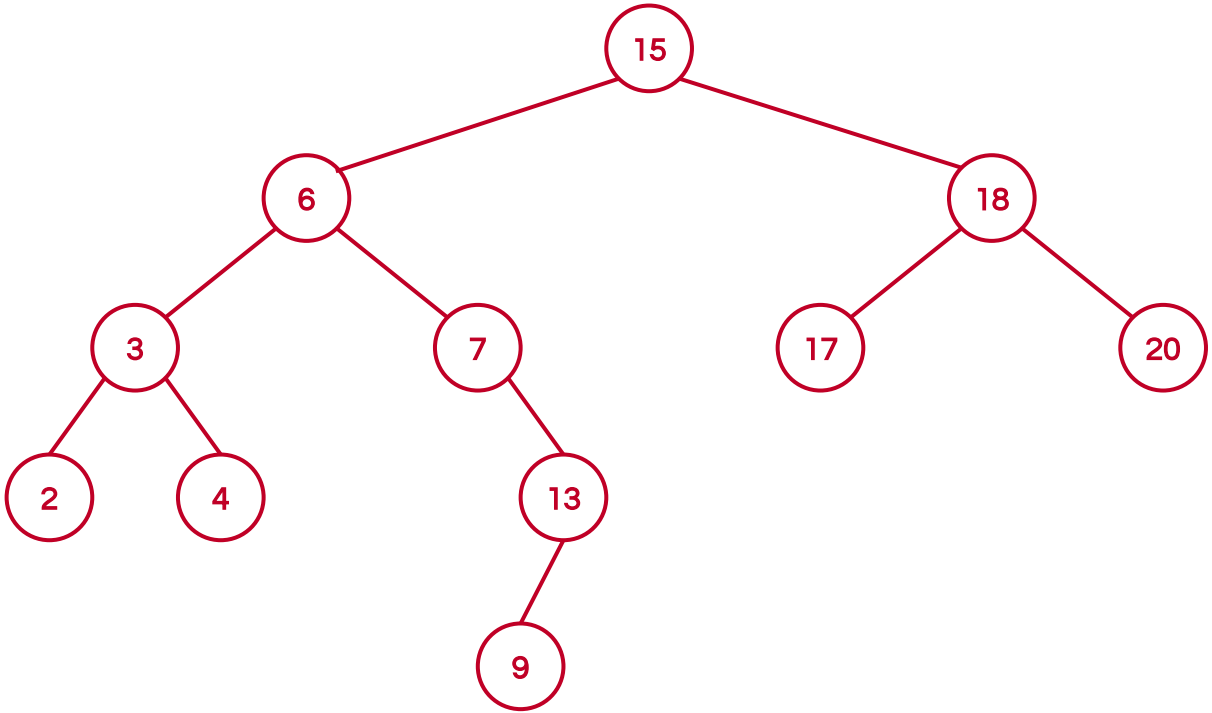
QUASE NO FIM

AGUENTEM SÓ MAIS UM POUCO

Encontro 27 de Desafios de Programação

ÁRVORE DE BUSCA BINÁRIA



ÁRVORE DE BUSCA BINÁRIA

```
node *find(node *r, int key);
```

(devolve endereço do nó com tal chave, ou NULL se não encontrar)

```
void add(node **r, int key);
```

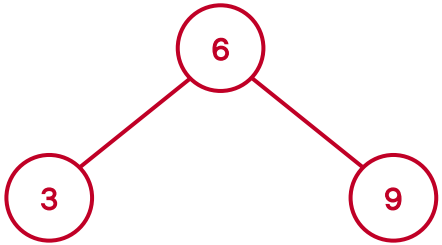
(adiciona nó com tal chave, ou não faz nada se já existir)

```
void remove(node **r, int key);
```

(remove nó com tal chave, ou não faz nada se não encontrar)

ÁRVORE DE BUSCA BINÁRIA

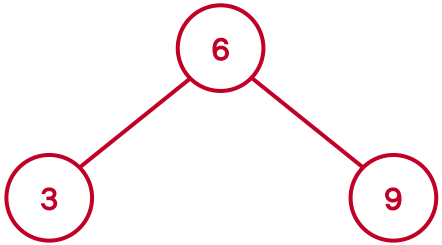
Todas as operações têm complexidade proporcional à altura da árvore...



encontrar	*O(altura)*
remover (já encontrou)	*O(altura)*
adicionar	*O(altura)*

ÁRVORE DE BUSCA BINÁRIA

Todas as operações têm complexidade proporcional à altura da árvore...

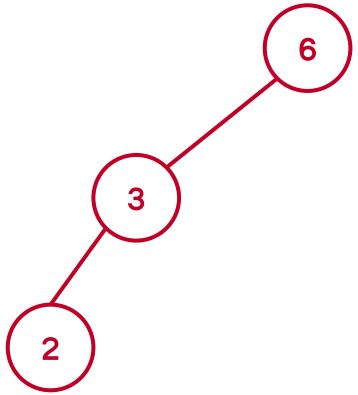


encontrar	*O(altura)*
remover (já encontrou)	*O(altura)*
adicionar	*O(altura)*

← Pode precisar de um maximum, lembra?

ÁRVORE DE BUSCA BINÁRIA

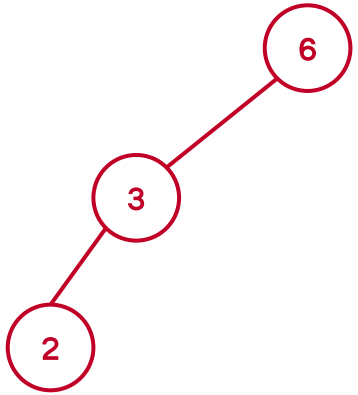
Todas as operações têm complexidade proporcional à altura da árvore, então se ela não estiver balanceada...



encontrar	*O(n)*
remover (já encontrou)	*O(n)*
adicionar	*O(n)*

ÁRVORE DE BUSCA BINÁRIA

Todas as operações têm complexidade proporcional à altura da árvore, então se ela não estiver balanceada...



encontrar	*O(n)*
remover (já encontrou)	*O(n)*
adicionar	*O(n)*

meh

ÁRVORE DE BUSCA BINÁRIA BALANCEADA

ÁRVORE DE BUSCA BINÁRIA BALANCEADA

as duas clássicas:

Árvores AVL	Árvores Rubro-Negras

ÁRVORE DE BUSCA BINÁRIA BALANCEADA

as duas clássicas:

Árvores AVL	Árvores Rubro-Negras
Guarda nos nós as alturas de cada subárvore.	Guarda em cada nó uma cor. (vermelho/preto)

ÁRVORE DE BUSCA BINÁRIA BALANCEADA

as duas clássicas:

Árvores AVL	Árvores Rubro-Negras
Guarda nos nós as alturas de cada subárvore. Após adicionar ou remover um nó, faz certos consertos para garantir que, para cada nó, a diferença de altura entre suas subárvores não é maior do que 1.	Guarda em cada nó uma cor. (vermelho/preto) Após adicionar ou remover um nó, faz certos consertos para garantir que a árvore continua seguindo certas regras baseadas nas cores. Essas regras garantem altura $O(\lg n)$.

ÁRVORE DE BUSCA BINÁRIA BALANCEADA

as duas clássicas:

Árvores AVL	Árvores Rubro-Negras
• busca a menor altura possível • consertos mais caros	• altura *O(lg n)*, mas não a menor possível • consertos mais baratos

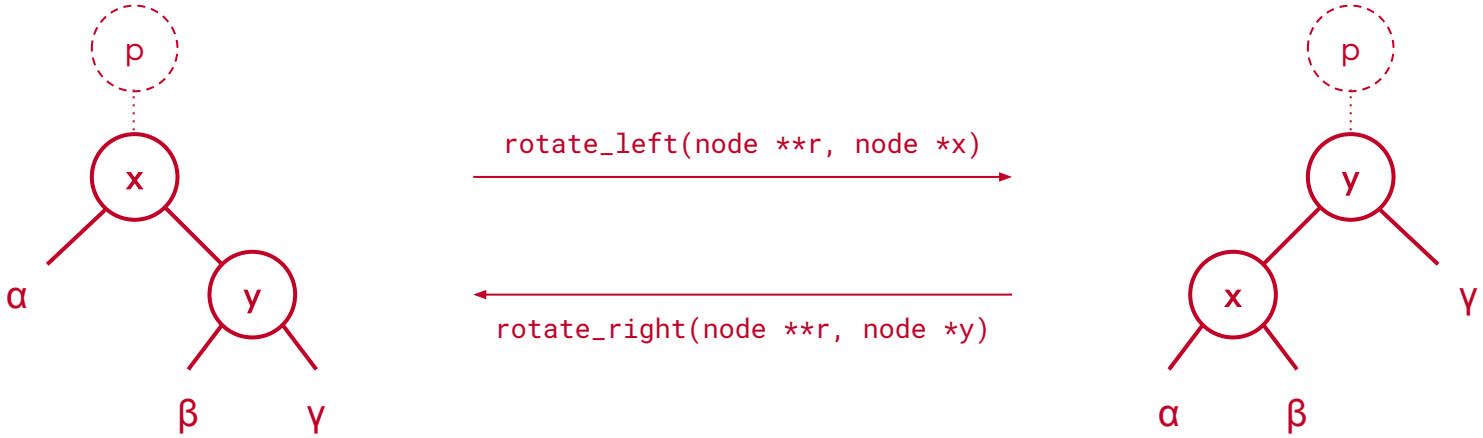
ÁRVORE DE BUSCA BINÁRIA BALANCEADA

as duas clássicas:

Árvores AVL	Árvores Rubro-Negras
para muitas buscas e poucas modificações	para poucas buscas e muitas modificações

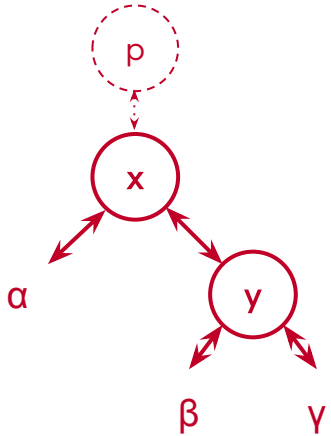
mas em ambas as árvores a operação básica dos consertos é a mesma!

ROTAÇÕES


p
x
α
y
β
γ
rotate_left(node **r, node *x)
rotate_right(node **r, node *y)
p
y
x
γ
α
β

ROTAÇÕES

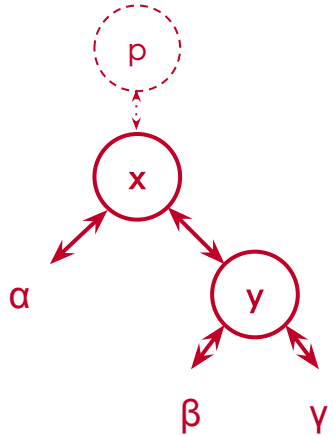
```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```



ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

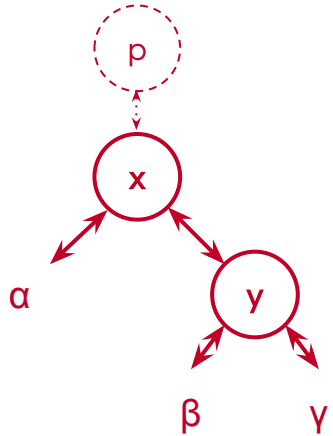
(estamos supondo que x e y existem)



ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

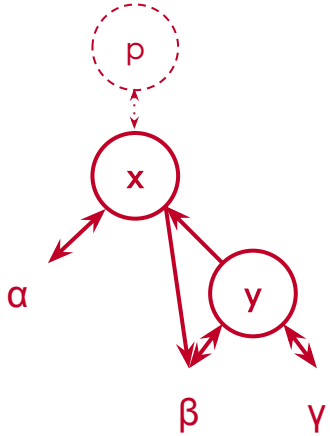
```
  // pendura β em x
```



ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

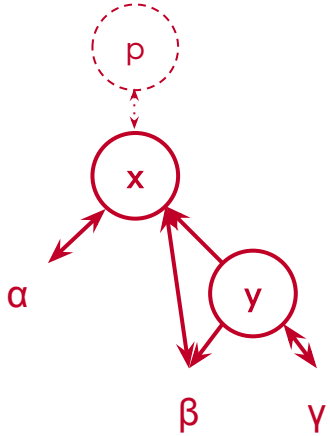
```
  // pendura β em x
  x->right = y->left;
```



ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

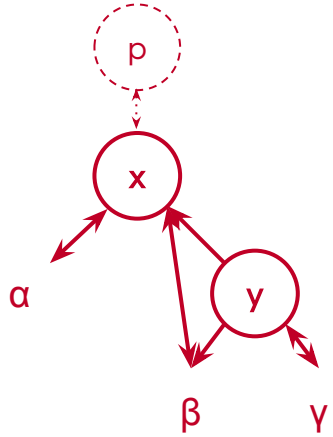
```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```



ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```

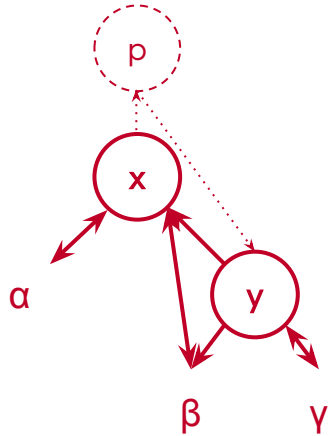



```
  // pendura y em p
```

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```

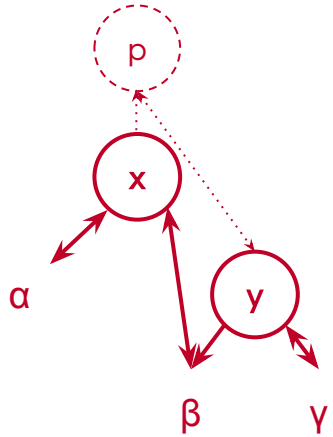



```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
```

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```

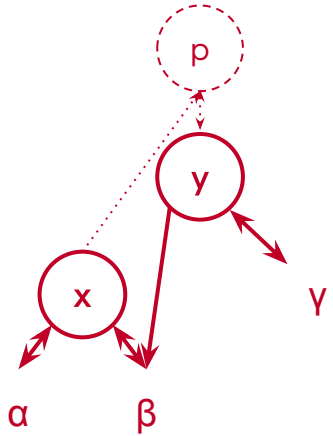



```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
  y->parent = p;
```

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```



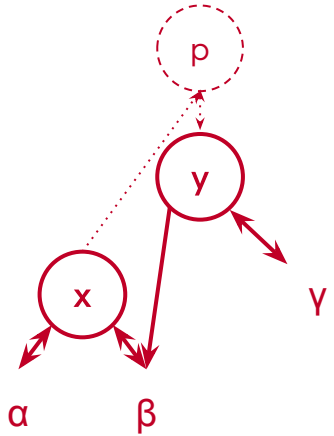

```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
  y->parent = p;
```

(pausa para reorganizar o desenho)

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```



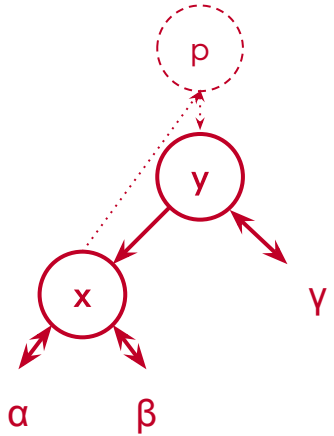

```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
  y->parent = p;
```

```
  // pendura x em y
```

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```



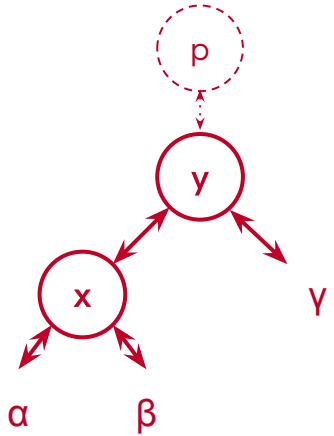

```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
  y->parent = p;
```

```
  // pendura x em y
  y->left = x;
```

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```



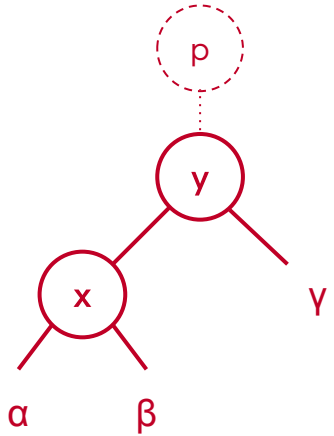

```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
  y->parent = p;
```

```
  // pendura x em y
  y->left = x;
  x->parent = y;
```

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```




```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
  y->parent = p;
```

```
  // pendura x em y
  y->left = x;
  x->parent = y;
}
```

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  y->left->parent = x;
```

(sobrou um errinho, consegue encontrar?)

```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
  y->parent = p;
```

```
  // pendura x em y
  y->left = x;
  x->parent = y;
}
```

ROTAÇÕES

```
void rotate_left(node **r, node *x) {
  node *p = x->parent;
  node *y = x->right;
```

```
  // pendura β em x
  x->right = y->left;
  if(y->left != NULL) {
    y->left->parent = x;
  }
```

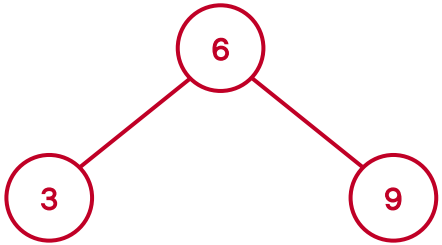
(a subárvore beta pode não existir!)

```
  // pendura y em p
  if(p == NULL) {
    *r = y;
  }
  else if(p->left == x) {
    p->left = y;
  }
  else {
    p->right = y;
  }
  y->parent = p;
```

```
  // pendura x em y
  y->left = x;
  x->parent = y;
}
```

ÁRVORE DE BUSCA BINÁRIA BALANCEADA

Todas as operações têm complexidade proporcional à altura da árvore, então se ela estiver balanceada...

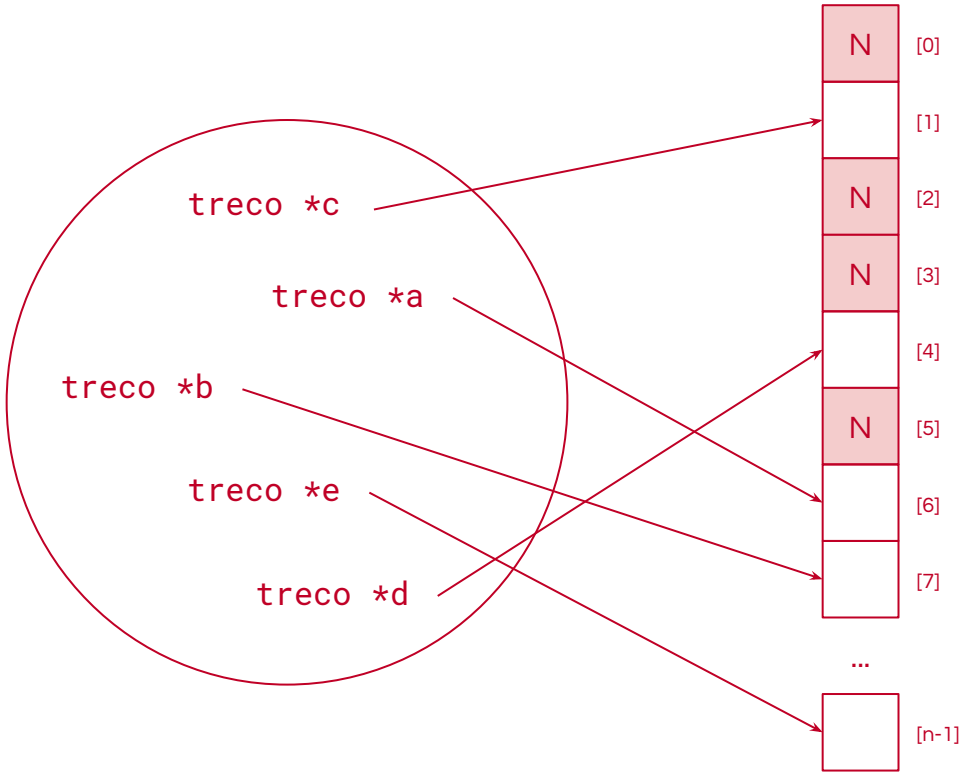


encontrar	*O(lg n)*
remover (já encontrou)	*O(lg n)*
adicionar	*O(lg n)*

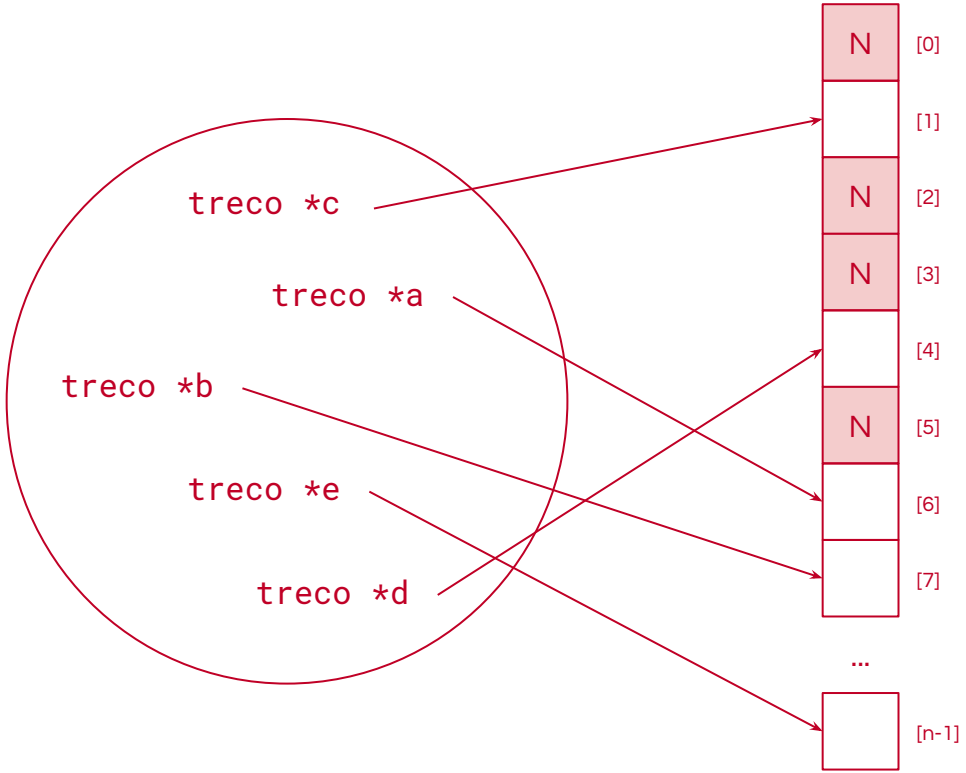
podemos considerar árvores de busca binária balanceadas como uma “versão eficiente” do conceito de listas ligadas (nós que apontam para outros nós)

existe uma “versão eficiente”
do conceito de vetores?
(acesso direto indexado)

TABELAS DE ESPALHAMENTO



TABELAS DE ESPALHAMENTO



Proposta Preliminar:

inicializar:

```
treco **hash;
hash = new treco*[n];
for(int i = 0; i < n; i++)
  hash[i] = NULL;
```

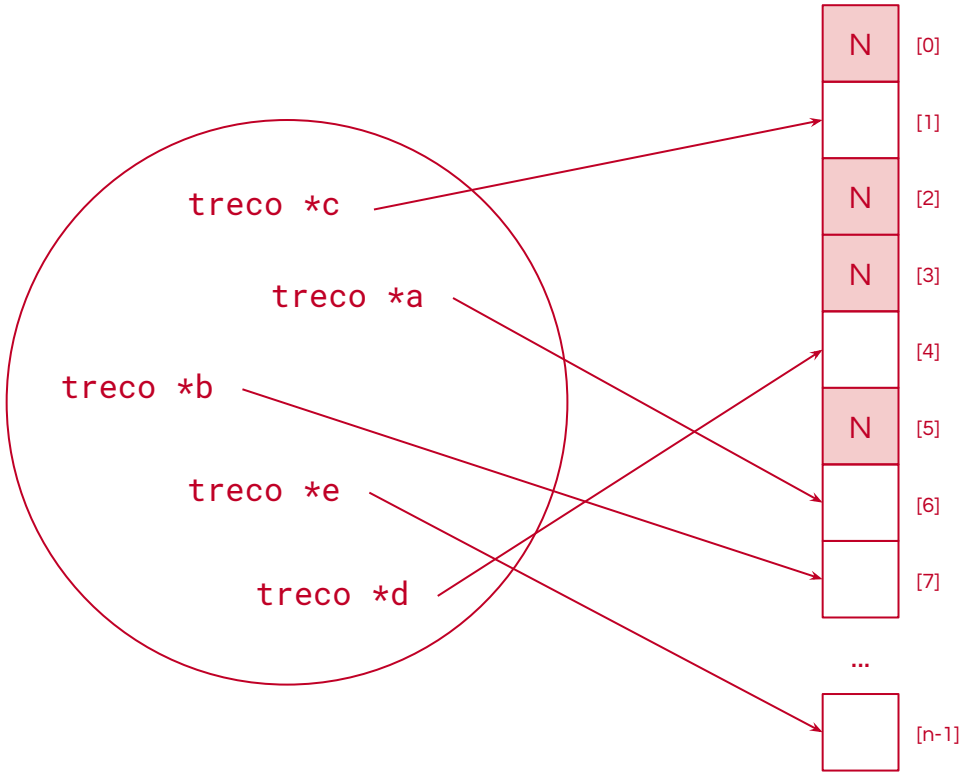
adicionar:

```
hash[t->key] = t;
```

remover:

```
hash[t->key] = NULL
```

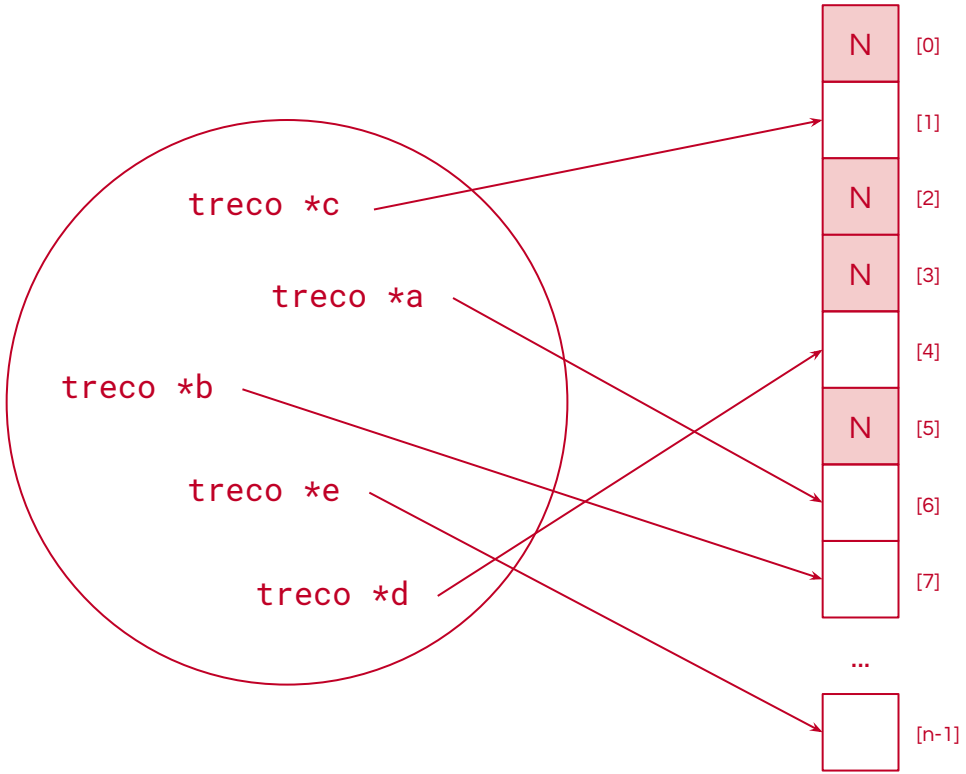
TABELAS DE ESPALHAMENTO



Problema:

como reduzir consumo de memória se o universo de chaves for muito grande?

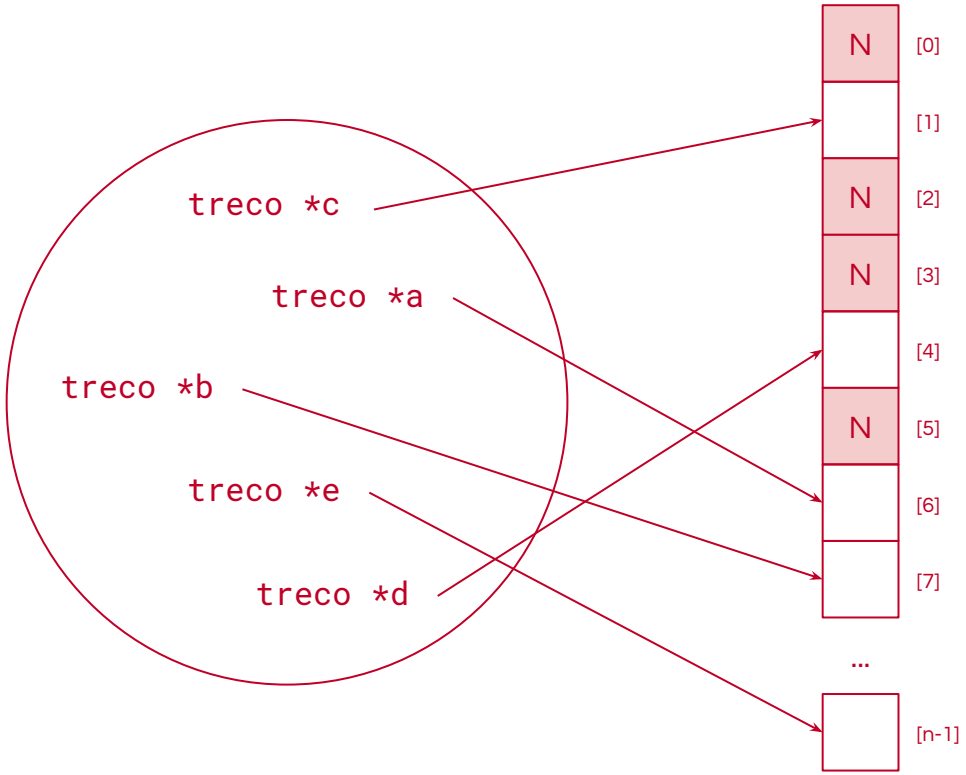
TABELAS DE ESPALHAMENTO



Problema:
como reduzir consumo de memória se o universo de chaves for muito grande?

Solução:
função hash que mapeia conjunto maior para conjunto menor

TABELAS DE ESPALHAMENTO

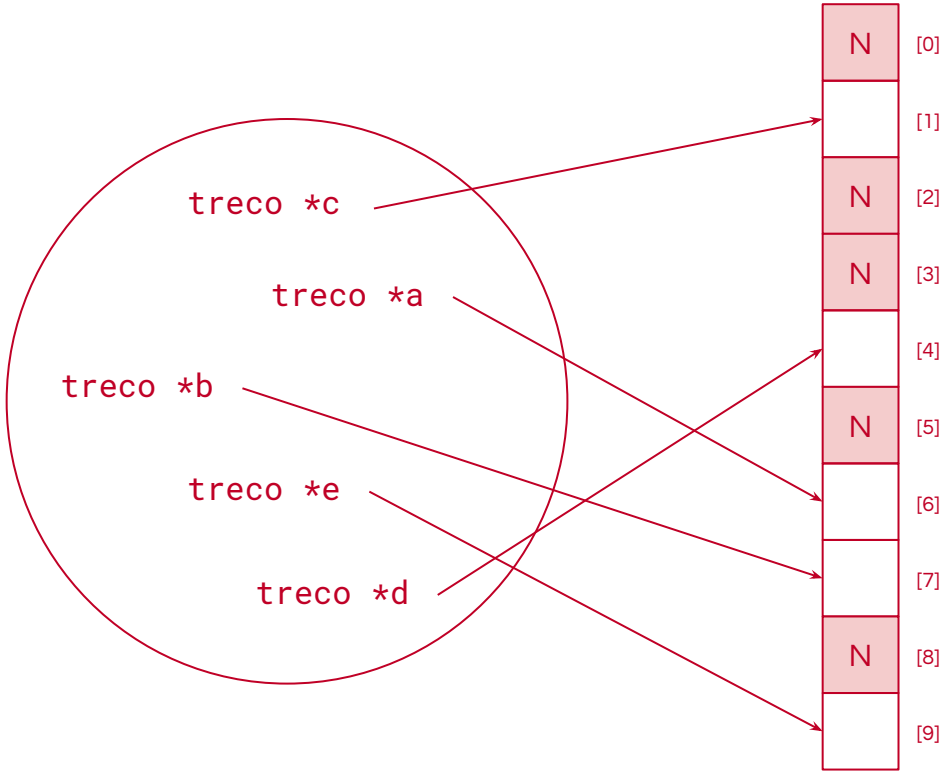


Problema:
como reduzir consumo de memória se o universo de chaves for muito grande?

Solução:
função hash que mapeia conjunto maior para conjunto menor

Exemplo:
resto da divisão

TABELAS DE ESPALHAMENTO



Nova Proposta:

```
const int MAX = 10;
```

inicializar:

```
treco **hash;
hash = new treco*[MAX];
for(int i = 0; i < MAX; i++)
  hash[i] = NULL;
```

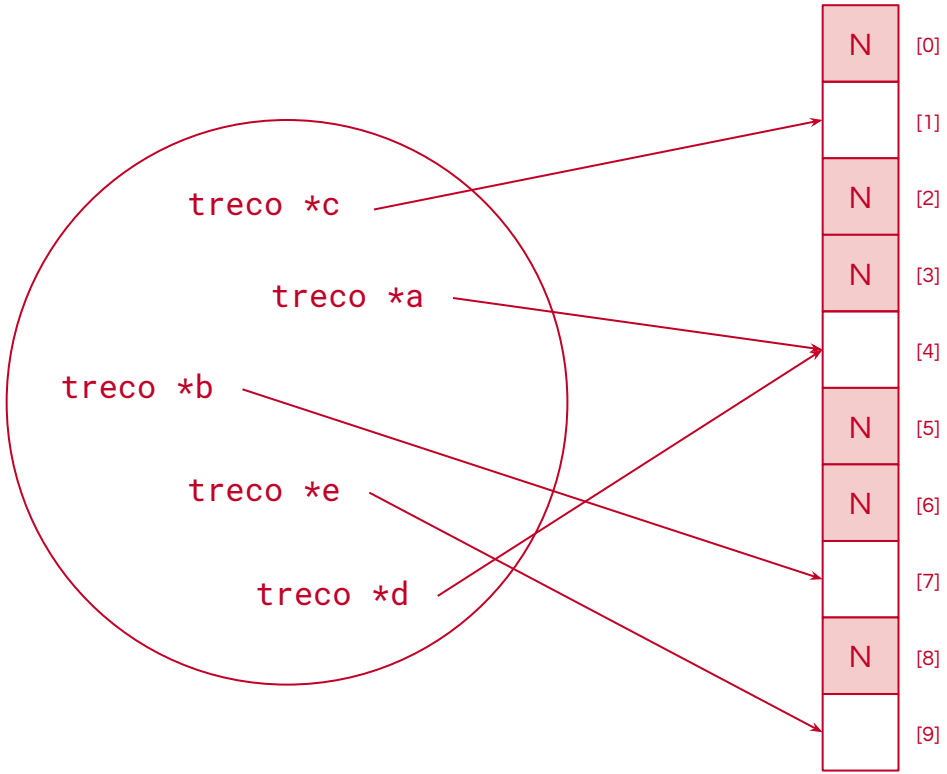
adicionar:

```
hash[t->key % MAX] = t;
```

remover:

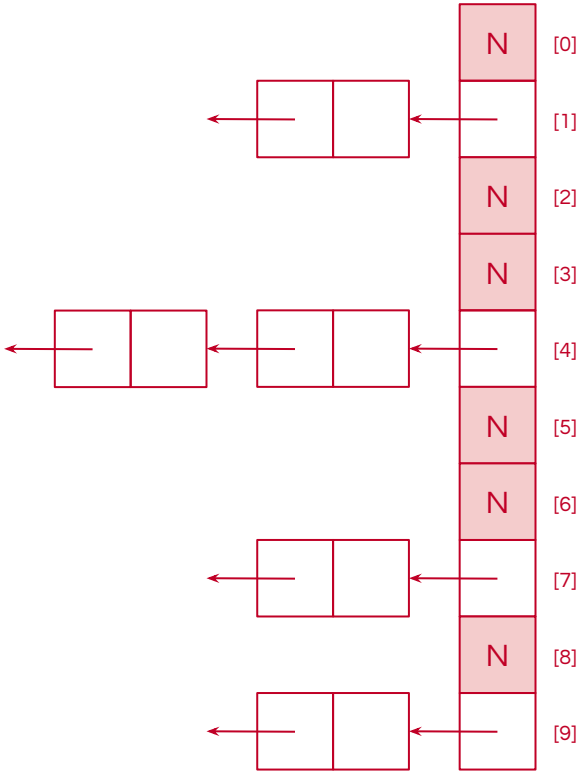
```
hash[t->key % MAX] = NULL
```

TABELAS DE ESPALHAMENTO



Problema:
o que fazer quando dois elementos são mapeados para o mesmo hash?

TABELAS DE ESPALHAMENTO



Problema:
o que fazer quando dois elementos são mapeados para o mesmo hash?

Solução:
tratamento de colisão usando listas ligadas

TABELAS DE ESPALHAMENTO

Eita, mas então essa estrutura não é grande coisa!

encontrar	$O(n)$
remover (já encontrou)	$O(1)$
adicionar	$O(1)$

TABELAS DE ESPALHAMENTO

Eita, mas então essa estrutura não é grande coisa!

No **pior caso** de fato não é...

encontrar	*O(n)*
remover (já encontrou)	*O(1)*
adicionar	*O(1)*

TABELAS DE ESPALHAMENTO

Eita, mas então essa estrutura não é grande coisa!

No pior caso de fato não é...

E no **caso médio**?

encontrar	*O(?)*
remover (já encontrou)	*O(1)*
adicionar	*O(1)*

TABELAS DE ESPALHAMENTO

Eita, mas então essa estrutura não é grande coisa!

No pior caso de fato não é...

E no caso médio? Depende de uma **boa função hash**!

encontrar	*O(?)*
remover (já encontrou)	*O(1)*
adicionar	*O(1)*

TABELAS DE ESPALHAMENTO

Eita, mas então essa estrutura não é grande coisa!

No pior caso de fato não é...

E no caso médio? Depende de uma boa função hash!

Uma boa função hash é aquela que **espalha bem** os elementos...

TABELAS DE ESPALHAMENTO

Eita, mas então essa estrutura não é grande coisa!

No pior caso de fato não é...

E no caso médio? Depende de uma boa função hash!

Uma boa função hash é aquela que espalha bem os elementos, ou seja, a probabilidade da função devolver cada um dos m elementos possíveis é $1/m$.

TABELAS DE ESPALHAMENTO

Eita, mas então essa estrutura não é grande coisa!

No pior caso de fato não é...

E no caso médio? Depende de uma boa função hash!

Uma boa função hash é aquela que espalha bem os elementos, ou seja, a probabilidade da função devolver cada um dos m elementos possíveis é $1/m$.

Note que definir uma função assim não é trivial! Por exemplo, não adianta definir uma função que divide o universo de chaves em partes iguais se algumas chaves são mais frequentes que outras!

TABELAS DE ESPALHAMENTO

Mas se conseguimos definir a função...

encontrar	$O(n/m)$
remover (já encontrou)	$O(1)$
adicionar	$O(1)$

TABELAS DE ESPALHAMENTO

Mas se conseguimos definir a função e partimos da premissa (perfeitamente aceitável na prática) de que $m = n/k$ com k constante...

encontrar	$O(n/(n/k))$
remover (já encontrou)	$O(1)$
adicionar	$O(1)$

TABELAS DE ESPALHAMENTO

Mas se conseguimos definir a função e partimos da premissa (perfeitamente aceitável na prática) de que $m = n/k$ com k constante, então a complexidade no caso médio é constante!

encontrar	$O(k)$
remover (já encontrou)	$O(1)$
adicionar	$O(1)$

TABELAS DE ESPALHAMENTO

Mas se conseguimos definir a função e partimos da premissa (perfeitamente aceitável na prática) de que $m = n/k$ com k constante, então a complexidade no caso médio é constante!

Se k não for absurdo (e na prática não é), essa complexidade é excelente!

encontrar	$O(1)$
remover (já encontrou)	$O(1)$
adicionar	$O(1)$

TABELAS DE ESPALHAMENTO

Mas se conseguimos definir a função e partimos da premissa (perfeitamente aceitável na prática) de que $m = n/k$ com k constante, então a complexidade no caso médio é constante!

Se k não for absurdo (e na prática não é), essa complexidade é excelente!

encontrar	$O(1)$
remover (já encontrou)	$O(1)$
adicionar	$O(1)$

← É um almoço muito, muito barato.

TABELAS DE ESPALHAMENTO

Mas se conseguimos definir a função e partimos da premissa (perfeitamente aceitável na prática) de que $m = n/k$ com k constante, então a complexidade no caso médio é constante!

Se k não for absurdo (e na prática não é), essa complexidade é excelente!

encontrar	$O(1)$
remover (já encontrou)	$O(1)$
adicionar	$O(1)$

É um almoço muito, muito barato, **mas ainda não é grátis**

A PERGUNTA FINAL

qual é a vantagem de árvores de busca binária em relação a tabelas de espalhamento?