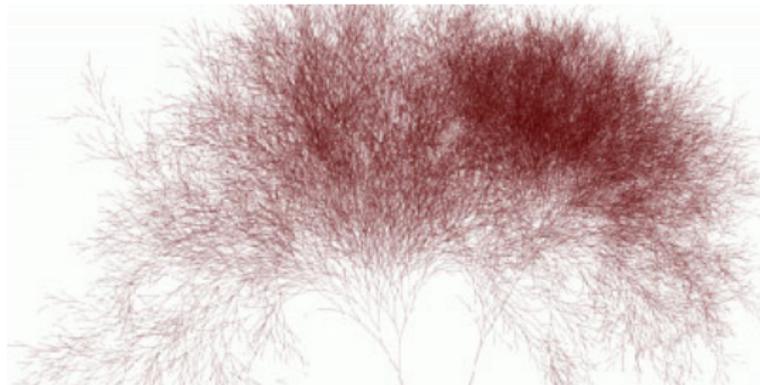


Melhores momentos

AULA 19

Árvores binárias de busca



Fonte: <http://infosthetics.com/archives/>

PF 15

<http://www.ime.usp.br/~pf/algoritmos/aulas/binst.html>

Árvores binárias de busca

Considere uma **árvore binária** cujos nós têm um campo **chave** (como **int** ou **String**, por exemplo).

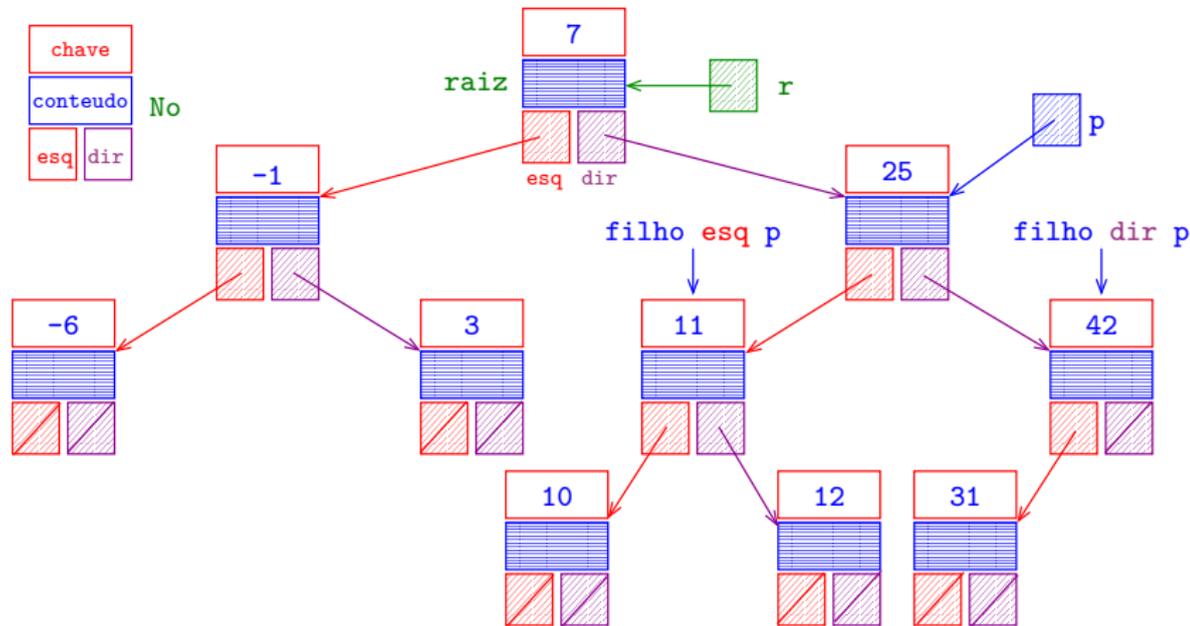
```
typedef struct celula No;
struct celula {
    int conteudo;           /* tipo devia ser Item */
    int chave;             /* tipo devia ser Chave */
    No *esq;
    No *dir;
};
typedef No *Arvore;
```

Árvores binárias de busca

Uma **árvore binária** deste tipo é de **busca** (em relação ao campo **chave**) se, para cada nó **x**, **x.chave** é

1. **maior ou igual** à chave de qualquer nó na subárvore **esquerda** de **x** e
2. **menor ou igual** à chave de qualquer nó na subárvore **direita** de **x**.

Ilustração de uma árvore binária de busca



in-ordem (e-r-d): -6 -1 3 7 10 11 12 25 31 42

AULA 20

Busca

Recebe um inteiro **k** e uma **árvore de busca r** e retorna um nó cuja chave é **k**; se tal nó não existe, retorna **NULL**.

Busca

Recebe um inteiro **k** e uma **árvore de busca r** e retorna um nó cuja chave é **k**; se tal nó não existe, retorna **NULL**.

```
No *busca(Arvore r, int k) {  
    if (r == NULL || r->chave == k)  
        return r;  
  
    if (r->chave > k)  
        return busca(r->esq, k);  
    return busca(r->dir, k);  
}
```

Busca versão iterativa

Recebe um inteiro **k** e uma árvore de busca **r** e retorna um nó cuja chave é **k**; se tal nó não existe, retorna **NULL**.

Busca versão iterativa

Recebe um inteiro `k` e uma árvore de busca `r` e retorna um nó cuja chave é `k`; se tal nó não existe, retorna `NULL`.

```
No *busca(Arvore r, int k) {
    while (r != NULL && r->chave != k)
        if (r->chave > k)
            r = r->esq;
        else
            r = r->dir;
    return r;
}
```

Inserção

Recebe uma árvore de busca r e um nó novo.
Insere o nó no lugar correto da árvore de modo que a árvore continue sendo de busca e retorna o endereço da nova árvore.

Inserção

Recebe uma árvore de busca `r` e um nó `novo`.
Insere o nó no lugar correto da árvore de modo que a árvore continue sendo de busca e retorna o endereço da nova árvore.

```
No *new(int chave, int conteudo, No *esq, No *dir) {  
    No *novo = mallocSafe(sizeof *novo);  
    novo->chave = chave;  
    novo->conteudo = conteudo;  
    novo->esq = esq;  
    novo->dir = dir;  
    return novo;  
}
```

Inserção

```
Arvore insere(Arvore r, No *novo) {
    No *f, /* filho de p */
    No *p; /* pai de f */
    if (r == NULL) return novo;
    f = r;
    while (f != NULL) {
        p = f;
        if (f->chave > novo->chave)
            f = f->esq;
        else
            f = f->dir;
    }
}
```

Inserção

```
/* novo sera uma folha
   novo sera filho de p */
if (p->chave > novo->chave)
    p->esq = novo;
else
    p->dir = novo;
return r;
}
```

Remoção

Recebe uma árvore de busca não vazia r . Remove a sua raiz e rearranja a árvore de modo que continue sendo de busca e retorna o endereço da nova árvore.

Remoção

Recebe uma árvore de busca não vazia r . Remove a sua raiz e rearranja a árvore de modo que continue sendo de busca e retorna o endereço da nova árvore.

```
Arvore removeRaiz(Arvore r) {  
    No *p, *q;  
    if (r->esq == NULL) {  
        q = r->dir;  
        free(r);  
        return q;  
    }  
}
```

Remoção

Recebe uma árvore de busca não vazia r . Remove a sua raiz e rearranja a árvore de modo que continue sendo de busca e retorna o endereço da nova árvore.

```
Arvore removeRaiz(Arvore r) {
```

```
    No *p, *q;
```

```
    if (r->esq == NULL) {
```

```
        q = r->dir;
```

```
        free(r);
```

```
        return q;
```

```
    }
```

```
/* busca nó q com maior valor na subarvore r->esq */
```

Remoção

```
p = r;    q = r->esq;
while (q->dir != NULL) {
    p = q;
    q = q->dir;
}
```

Remoção

```
p = r;    q = r->esq;
while (q->dir != NULL) {
    p = q;
    q = q->dir;
}
/* q: nó anterior a r na ordem e-r-d */
if (p != r) {
    p->dir = q->esq;
    q->esq = r->esq;
}
q->dir = r->dir;
free(r);
return q;
}
```

Remoção de nó arbitrário

Recebe um inteiro k e uma árvore de busca r e retorna a raiz da árvore obtida da remoção de um nó cuja chave é k ; se tal nó não existe, retorna a própria árvore r .

Remoção de nó arbitrário

Recebe um inteiro **k** e uma **árvore de busca r** e retorna a raiz da árvore obtida da remoção de um nó cuja chave é **k**; se tal nó não existe, retorna a própria árvore **r**.

```
No *remove(Arvore r, int k) {  
    if (r == NULL) return r;  
    if (r->chave == k) return removeRaiz(r);
```

Remoção de nó arbitrário

Recebe um inteiro **k** e uma **árvore de busca r** e retorna a raiz da árvore obtida da remoção de um nó cuja chave é **k**; se tal nó não existe, retorna a própria árvore **r**.

```
No *remove(Arvore r, int k) {  
    if (r == NULL) return r;  
    if (r->chave == k) return removeRaiz(r);  
    if (r->chave > k)  
        r->esq = remove(r->esq, k);  
    else  
        r->dir = remove(r->dir, k);  
    return r;  
}
```

Consumo de tempo

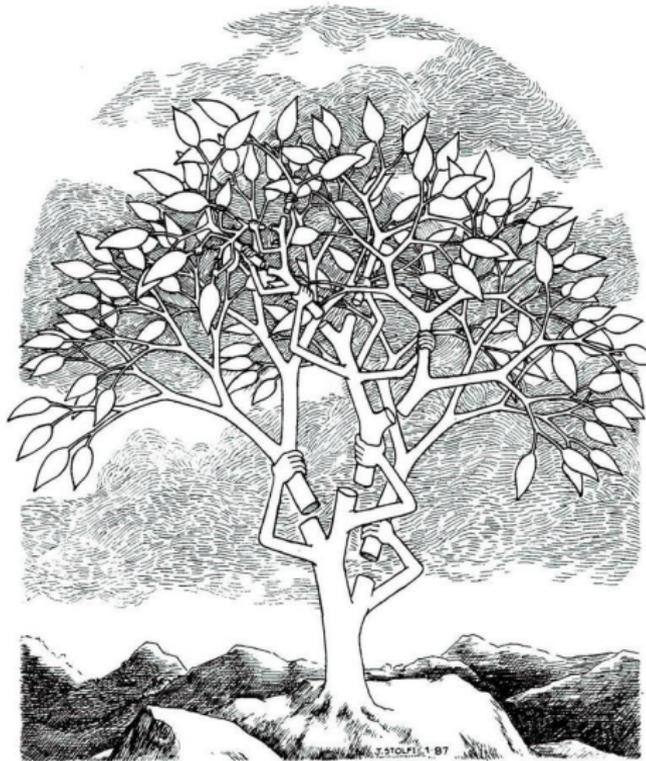
O consumo de tempo das funções **busca**, **insere** e **remove** é, no pior caso, proporcional à **altura** da **árvore**.

Consumo de tempo

O consumo de tempo das funções **busca**, **insere** e **remove** é, no pior caso, proporcional à **altura** da **árvore**.

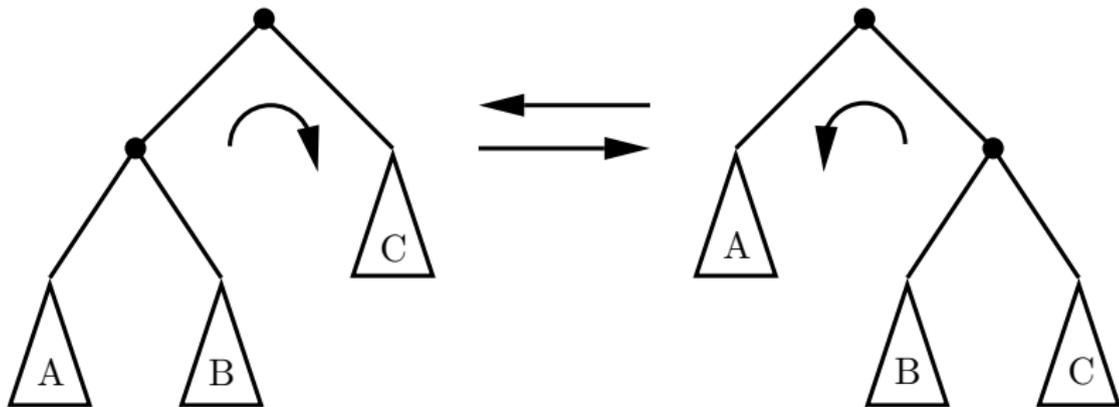
Conclusão: interessa trabalhar com **árvores balanceadas**:
árvores **AVL**, árvores **rubro-negras**, **treaps** ...

Self balancing trees

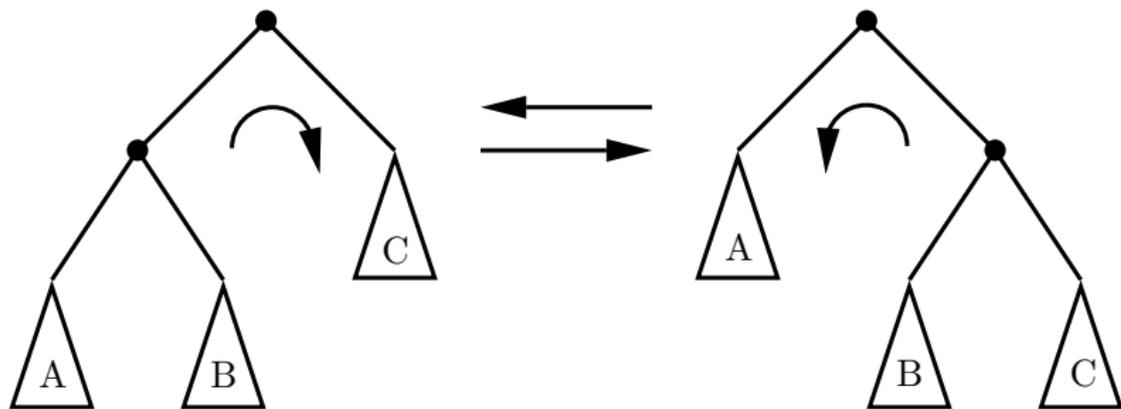


autor: Jorge Stolfi

rotações



rotações



Observe que a ordem se mantém:

se era **ABB**, então continua **ABB**.

Rotação para direita

Recebe uma árvore r com
 $r \neq \text{NULL}$ e $r \rightarrow \text{esq} \neq \text{NULL}$.

```
Arvore rotacaoDir(Arvore r) {  
    No *q = r->esq;  
    r->esq = q->dir;  
    q->dir = r;  
    return q;  
}
```

Rotação para direita

Recebe uma árvore r com
 $r \neq \text{NULL}$ e $r \rightarrow \text{esq} \neq \text{NULL}$.

```
Arvore rotacaoDir(Arvore r) {  
    No *q = r->esq;  
    r->esq = q->dir;  
    q->dir = r;  
    return q;  
}
```

Consumo de tempo: constante, ou seja, $O(1)$.

Escreva o `rotacaoEsq`.

Treap

Uma **treap** (= *tree + heap*) é uma árvore de busca binária **aleatorizada**, onde cada nó tem um campo extra chamado **prior** (= *priority*).

Treap

Uma **treap** (= *tree + heap*) é uma árvore de busca binária **aleatorizada**, onde cada nó tem um campo extra chamado **prior** (= *priority*).

A treap, como a árvore de busca binária, guarda um conjunto de **objetos** (*itens*), cada um dotado de uma **chave** (= *key*) e de um **valor** (= *value*).

Treap

Uma **treap** (= *tree + heap*) é uma árvore de busca binária **aleatorizada**, onde cada nó tem um campo extra chamado **prior** (= *priority*).

A treap, como a árvore de busca binária, guarda um conjunto de **objetos** (*itens*), cada um dotado de uma **chave** (= *key*) e de um **valor** (= *value*).

As chaves podem ser números inteiros ou *strings* ou outro tipo de dados.

Treap

Uma **treap** (= *tree + heap*) é uma árvore de busca binária **aleatorizada**, onde cada nó tem um campo extra chamado **prior** (= *priority*).

A treap, como a árvore de busca binária, guarda um conjunto de **objetos** (*itens*), cada um dotado de uma **chave** (= *key*) e de um **valor** (= *value*).

As chaves podem ser números inteiros ou *strings* ou outro tipo de dados.

As prioridades satisfazem a propriedade de um heap:
todo nós deve ter prioridade maior que a prioridade dos seus filhos.

Busca em treap

Como fica a busca?

Busca em treap

Como fica a busca? Igual!

Busca em treap

Como fica a busca? Igual!

E remoção?

Busca em treap

Como fica a busca? Igual!

E remoção? Quase igual!

Como fica a prioridade do nó que substituiu a raiz?

Busca em treap

Como fica a busca? **Igual!**

E remoção? **Quase igual!**

Como fica a prioridade do nó que substituiu a raiz?

Faça rotações para restaurar as condições de heap.

Busca em treap

Como fica a busca? **Igual!**

E remoção? **Quase igual!**

Como fica a prioridade do nó que substituiu a raiz?

Faça rotações para restaurar as condições de heap.

Consumo de tempo:

O consumo de tempo de **busca** e **remove** numa treap é, no pior caso, proporcional à sua **altura**.

Altura de uma treap

O consumo de tempo de `busca` e `remove` numa treap é, no pior caso, proporcional à sua `altura`.

Qual é a altura de uma treap?

Altura de uma treap

O consumo de tempo de `busca` e `remove` numa treap é, no pior caso, proporcional à sua `altura`.

Qual é a altura de uma treap?

Depende das prioridades aleatórias...

Altura de uma treap

O consumo de tempo de `busca` e `remove` numa treap é, no pior caso, proporcional à sua `altura`.

Qual é a altura de uma treap?

Depende das prioridades aleatórias...

É uma variável aleatória!

Inserção em treaps

Insere-se o novo nó exatamente como antes.

Inserção em treaps

Insere-se o novo nó exatamente como antes.

Atribui-se uma prioridade escolhida aleatoriamente de maneira uniforme de um conjunto grande.

Inserção em treaps

Insere-se o novo nó exatamente como antes.

Atribui-se uma prioridade escolhida aleatoriamente de maneira uniforme de um conjunto grande.

Por meio de rotações,
se simula um `sobe_heap` caso necessário.

Inserção em treaps

Insere-se o novo nó exatamente como antes.

Atribui-se uma prioridade escolhida aleatoriamente de maneira uniforme de um conjunto grande.

Por meio de rotações,
se simula um `sobe_heap` caso necessário.

Ou seja, enquanto o nó tem prioridade maior que o seu pai, executa-se uma rotação apropriada para que ele fique acima do pai.

Inserção em treaps

Implementação com apontador `pai`.

```
typedef struct celula Celula;
struct celula {
    int conteudo;           /* tipo devia ser Item */
    int chave;             /* tipo devia ser Chave */
    int prior;             ◁ ◁ ◁
    Celula *pai;
    Celula *esq;
    Celula *dir;
};
typedef Celula No;
typedef No *Treap;
```

Inserção

Recebe uma *treap* r e um nó *novo*.

Insere o nó no lugar correto da *treap*, executa um *sobe_heap* e *retorna* o endereço da nova *treap*.

Inserção

Recebe uma treap `r` e um nó `novo`.

Insere o nó no lugar correto da treap, executa um `sobe_heap` e retorna o endereço da nova treap.

```
No *new(int chave, int conteudo, No *esq, No *dir) {  
    No *novo = mallocSafe(sizeof *novo);  
    novo->chave = chave;  
    novo->conteudo = conteudo;  
    novo->prior = random();  
    novo->pai = NULL;  
    novo->esq = esq;  
    novo->dir = dir;  
    return novo;  
}
```



Rotação com apontador pai

Recebe treap r e nó $q \neq \text{NULL}$ com $q \rightarrow \text{pai} \neq \text{NULL}$.

Rotação com apontador pai

Recebe treap r e nó $q \neq \text{NULL}$ com $q \rightarrow \text{pai} \neq \text{NULL}$.

```
Treap rotaciona(Treap r, No *q) {
    No *p = q->pai;
    q->pai = p->pai;
    if (q == p->esq) {      ▷ rotação p/direita
        p->esq = q->dir;
        if (q->dir != NULL) q->dir->pai = p;
        q->dir = p;
        p->pai = q;
    } else                 ▷ rotação p/esquerda ...
        return r;
}
```

Inserção

```
Treap insere(Treap r, No *novo) {  
    No *f, *p; /* pai de f */  
    if (r == NULL) return novo;  
    f = r;  
    while (f != NULL) {  
        p = f;  
        if (f->chave > novo->chave)  
            f = f->esq;  
        else  
            f = f->dir;  
    }  
}
```

Inserção

`/* novo será uma folha, filho de p */`

```
novo->pai = p;
if (p->chave > novo->chave)
    p->esq = novo;
else
    p->dir = novo;

/* ajusta a árvore pelas prioridades */
q = novo->pai;
while (q != NULL && q->prior < novo->prior) {
    r = rotaciona (r, novo);
    q = novo->pai;
}
return r;
}
```

Qual é a altura de uma treap?

Depende das prioridades aleatórias...

É uma variável aleatória!

Pode-se mostrar que o valor esperado da altura de uma treap é $O(\lg n)$, onde n é o número de nós na treap.

Qual é a altura de uma treap?

Depende das prioridades aleatórias...

É uma variável aleatória!

Pode-se mostrar que o valor esperado da altura de uma treap é $O(\lg n)$, onde n é o número de nós na treap.

Consumo de tempo:

O consumo de tempo **esperado** de **busca**, **insere** e **remove** numa treap é $O(\lg n)$.