Técnicas de Busca e Ordenação (TBO)

Árvores Binárias de Busca

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

(Material baseado nos slides dos Professores Eduardo Zambon e Giovanni Comarela)

Introdução

- Uma estrutura que já vem sendo estudada há bastante tempo no curso é a árvore binária de busca *(BST – binary search tree)*.
- Alguns programadores consideram a BST uma das estruturas de dados mais importantes.
- **Aula de hoje**: mostrar como uma tabela de símbolos pode ser implementada como uma BST.
- **Objetivos**: compreender as aplicações e o funcionamento das BSTs.

Referências

Chapter 12 – Symbol Tables and Binary Search Trees
R. Sedgewick

Árvores binárias de busca

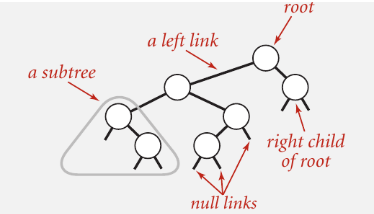
Definição: uma BST é uma árvore binária com ordenação simétrica.

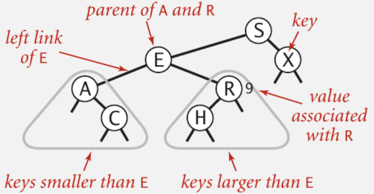
Uma árvore binária é:

- Vazia.
- Duas árvores binárias disjuntas (esquerda e direita).

Ordenação simétrica. Cada nó tem uma chave aonde:

- Toda a chave da subárvore da esquerda é menor.
- Toda a chave da subárvore da direita é maior.





BSTs: operações fundamentais

Busca. Comece da raiz, comparando as chaves:

- Se menor, vá para subárvore da esquerda.
- Se maior, vá para subárvore da direita.
- Se igual, *search hit*.
- Se nulo, *search miss*.

Inserção. Comece da raiz, comparando as chaves:

- Se menor, vá para subárvore da esquerda.
- Se maior, vá para subárvore da direita.
- Se igual, retorne (não admite valores repetidos).
- Se nulo, insira.

Ver arquivo `32DemoBinarySearchTree.mov`.

BST: representação em C

Definição em C: Uma BST é um ponteiro para um nó raiz (*root*).

Um nó é composto por quatro campos:

- Uma chave e um valor.
- Ponteiros para as subárvores da esquerda e da direita.

Chaves precisam ser comparáveis. Chaves e valores possuem uma constante nula para cada um.

```
typedef struct node Node;

struct node {
    Key key;       // Sorted by key.
    Value val;     // Associated data.
    Node *l, *r;   // Left and right subtrees.
};

Node *root; // Root of BST.
```

Busca em BST: implementação em C

Função `ST_get`:

- Retorna o valor associado a uma chave.
- Ou `NULL_Value` se a chave não existe.

Versão recursiva:

```
static Value rec_get(Node *n, Key key) {
    if (n == NULL) { return NULL_Value; }
    int cmp = compare(key, n->key);
    if      (cmp < 0) { return rec_get(n->l, key); }
    else if (cmp > 0) { return rec_get(n->r, key); }
    else /*cmp == 0*/ { return n->val; }
}

Value ST_get(Key key) {
    return rec_get(root, key);
}
```

Busca em BST: implementação em C

Função `ST_get`:

- Retorna o valor associado a uma chave.
- Ou `NULL_Value` se a chave não existe.

Versão não-recursiva:

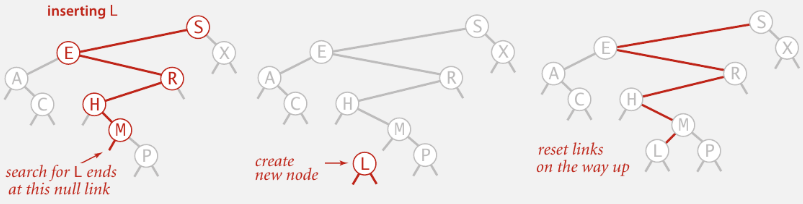
```
Value ST_get(Key key) {
    Node *n = root;
    while (n != NULL) {
        int cmp = compare(key, n->key);
        if      (cmp < 0) { n = n->l; }
        else if (cmp > 0) { n = n->r; }
        else /*cmp == 0*/ { return n->val; }
    }
    return NULL_Value;
}
```

Custo: Número de comparações é $= 1+$ profundidade do nó.

Inserção em BST

Função `ST_put`:

- Armazena o par (chave, valor) passado.
- Chave já existe $\Rightarrow$ atualiza o valor.
- Chave não existe $\Rightarrow$ cria novo nó.



Inserção em BST: implementação em C

Função `ST_put`:

- Armazena o par (chave, valor) passado.
- Chave já existe ⇒ atualiza o valor.
- Chave não existe ⇒ cria novo nó.

Versão recursiva:

```
static Node* rec_put(Node *n, Key key, Value val) {
    if (n == NULL) { return create_node(key, val); }
    int cmp = compare(key, n->key);
    if      (cmp < 0) { n->l = rec_put(n->l, key, val); }
    else if (cmp > 0) { n->r = rec_put(n->r, key, val); }
    else /*cmp == 0*/ { n->val = val; }
    return n;
}

void ST_put(Key key, Value val) {
    root = rec_put(root, key, val);
}
```

Inserção em BST: implementação em C

Versão não-recursiva:

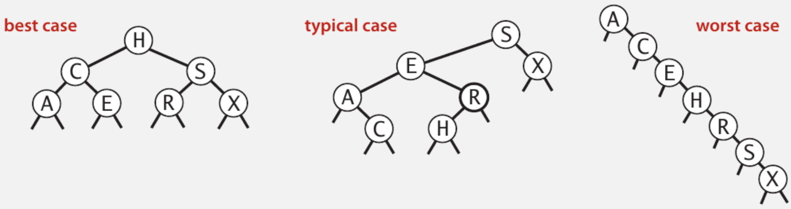
```
void ST_put(Key key, Value val) {
    if (root == NULL) { root = create_node(key, val); return; }

    Node *parent = NULL, *n = root;
    while (n != NULL) {
        parent = n;
        int cmp = compare(key, n->key);
        if      (cmp < 0) { n = n->l; }
        else if (cmp > 0) { n = n->r; }
        else /*cmp == 0*/ { n->val = val; return; }
    }

    Node *new = create_node(key, val);
    int cmp = compare(key, parent->key);
    if   (cmp < 0) { parent->l = new; }
    else           { parent->r = new; }
}
```

Custo: Número de comparações é $= 1+$ profundidade do nó.

Forma da árvore

Observações:

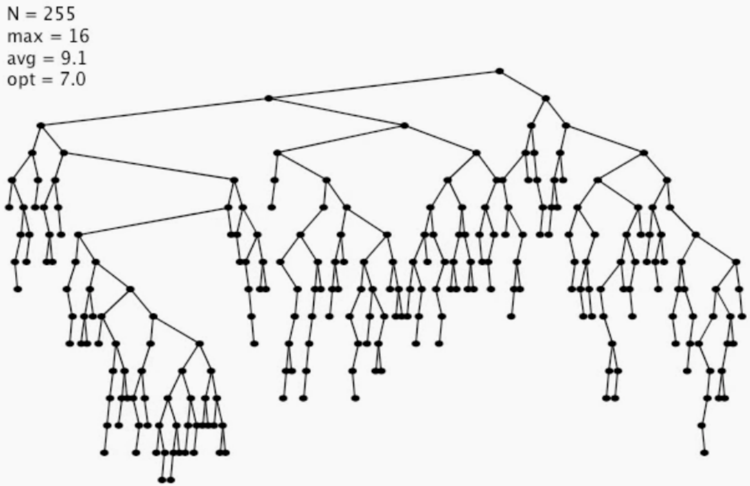
- Muitas BSTs correspondem ao mesmo conjunto de chaves.
- Número de comparações para busca/inserção é igual a 1 + profundidade do nó.



Conclusão: A forma da árvore depende da ordem de inserção das chaves.

Inserção aleatória em BST: visualização

Inserção de chaves em uma ordem aleatória.



Veja também o Laboratório de Árvores e Recursão.

Ordenação com uma BST

Q: Que algoritmo de ordenação é esse?

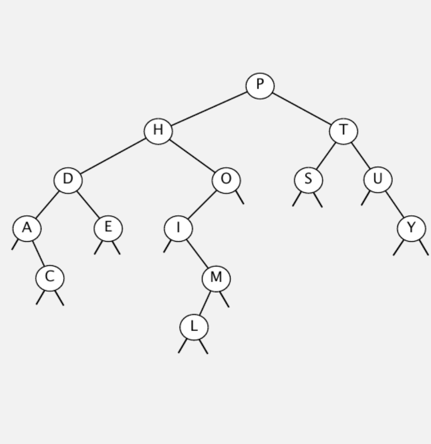
```
0. Embaralhe as chaves.
1. Insira todas as chaves na BST.
2. Caminhe in-order na BST.
```

A: Isso não é um algoritmo de ordenação se há chaves duplicadas!

Q: OK, e se não existir duplicação? Quais são as propriedades desse algoritmo?

Correspondência entre BSTs e *quick sort*

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



Observação: existe uma correspondência de 1-1 entre a BST e o particionamento do *quick sort* se o *array* não possui chaves duplicadas.

BSTs: análise matemática

Proposição: Se N chaves distintas são inseridas em uma BST de forma aleatória, o número esperado de comparações para busca/inserção é $\sim 2 \ln N$.

Justificativa: Essa é a complexidade do particionamento do *quick sort*, e existe uma correspondência de 1-1.

Proposição [Reed, 2003]: Se N chaves distintas são inseridas de forma aleatória, a altura esperada da BST é $\sim 4.311 \ln N$.

Mas... O pior caso ainda tem altura N.

(Embora a chance disso ocorrer seja exponencialmente pequena com a inserção aleatória.)

Implementações de tabelas de símbolos: sumário

implementation	guarantee		average case	
	search	insert	search hit	insert
sequential search (unordered list)	N	N	½ N	N
binary search (ordered array)	lg N	N	lg N	½ N
BST	N	N	1.39 lg N	1.39 lg N

Why not shuffle to ensure a (probabilistic) guarantee of 4.311 ln N?

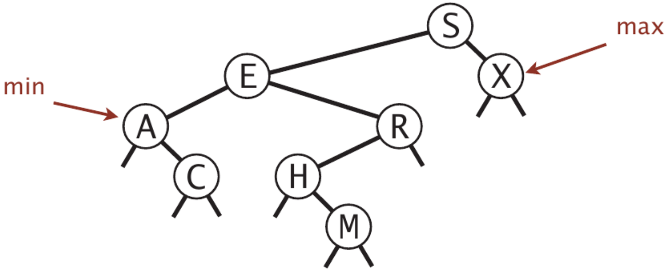
Part I

Operações Ordenadas

Mínimo e máximo

Mínimo: menor chave na tabela.

Máximo: maior chave na tabela.

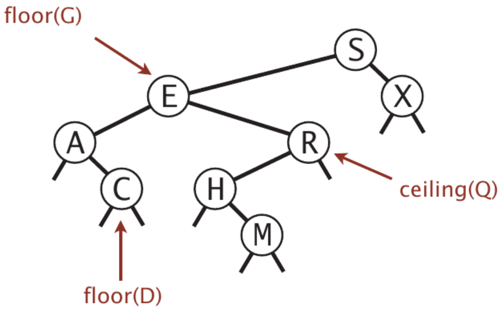


Q: Como encontrar o mínimo / máximo?

Floor e *ceiling*

Floor: Maior chave $\leq$ que uma chave dada.

Ceiling: Menor chave $\geq$ que uma chave dada.



Q: Como encontrar *floor* e *ceiling*?

Calculando *floor*

Caso 1:

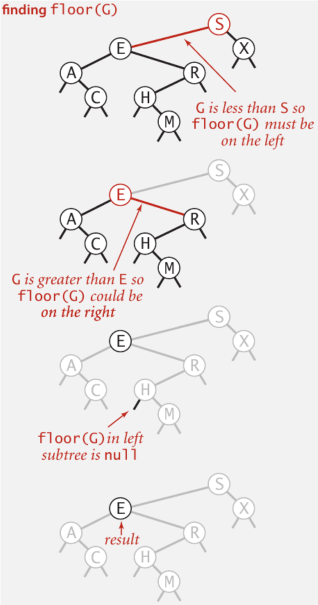
- Chave k é igual à chave do nó.
- O *floor* de k é k.

Caso 2:

- Chave k é menor que a chave do nó.
- O *floor* de k está na subárvore da esquerda.

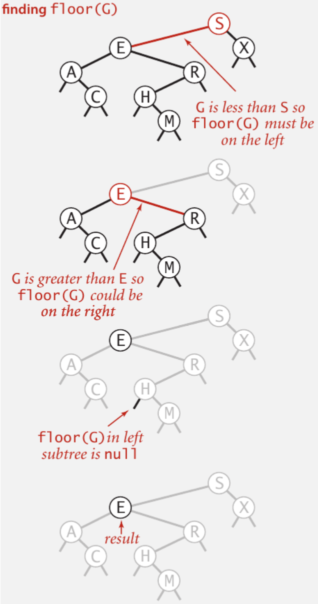
Caso 3:

- Chave k é maior que a chave do nó.
- O *floor* de k está na subárvore da direita.
- Mas só se houver alguma chave à direita $\leq k$.
- Caso contrário é a chave do nó.



Calculando *floor*

```
static Node* rec_floor(Node *n, Key key) {
    if (n == NULL) { return NULL; }
    int cmp = compare(key, n->key);
    if (cmp == 0) { return n; }
    if (cmp <  0) {
        return rec_floor(n->l, key);
    }
    Node *t = rec_floor(n->r, key);
    if (t != NULL) { return t; }
    else           { return n; }
}

Key ST_floor(Key key) {
    Node *n = rec_floor(root, key);
    if (n == NULL) { return NULL_Key; }
    else           { return n->key; }
}
```

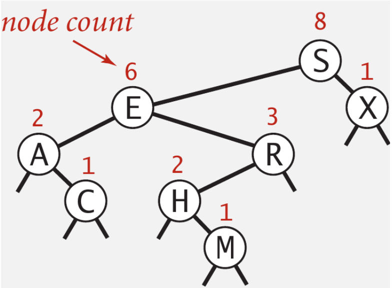


Código para `ceiling` é bem similar.

Rank

Q: Como implementar *rank* de forma eficiente?

A: Em cada nó, armazenar o tamanho da subárvore que começa nesse nó.



Implementação da BST: tamanho das subárvores

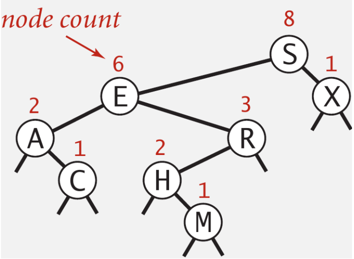
```
struct node {
    Key key;        // Sorted by key.
    Value val;      // Associated data.
    Node *l, *r;    // Left and right subtrees.
    int size;       // Number of nodes in subtree.
};
```

```
static int size(Node *n) {
  if (n == NULL) { return 0; }
  else       { return n->size; }
}
```

```
int ST_size() {
  return size(root);
}
```

```
static Node* rec_put(Node *n, Key key, Value val) {
    if (n == NULL) { return create_node(key, val, 1); }
    int cmp = compare(key, n->key);
    if      (cmp < 0) { n->l = rec_put(n->l, key, val); }
    else if (cmp > 0) { n->r = rec_put(n->r, key, val); }
    else /*cmp == 0*/ { n->val = val; }
    n->size = size(n->l) + size(n->r) + 1;
    return n;
}
```

Rank: Quantas chaves $< k$?




```
static int rec_rank(Node *n, Key key) {
  if (n == NULL) { return 0; }
  int cmp = compare(key, n->key);
  if      (cmp < 0) { return rec_rank(n->l, key); }
  else if (cmp > 0) { return 1+size(n->l)+rec_rank(n->r, key); }
  else /*cmp == 0*/ { return size(n->l); }
}
int ST_rank(Key key) {
    return rec_rank(root, key);
}
```

Caminhamento na BST

In-order traversal:

- Caminhe recursivamente pela subárvore da esquerda.
- Visite a chave do nó.
- Caminhe recursivamente pela subárvore da direita.

```
static void rec_traverse(Node *n, void (*visit)(Key,Value)) {
    if (n == NULL) { return; }
    rec_traverse(n->l, visit);
    visit(n->key, n->val);
    rec_traverse(n->r, visit);
}

void ST_traverse(void (*visit)(Key,Value)) {
    rec_traverse(root, visit);
}
```

Propriedade: *in-order traversal* na BST visita os nós em ordem crescente.

Implementações de tabelas de símbolos: sumário

	sequential search	binary search	BST
search	N	$\lg N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N

h = height of BST
(proportional to log N
if keys inserted
in random order)

Part II

Remoção em BSTs

Implementações de tabelas de símbolos: sumário

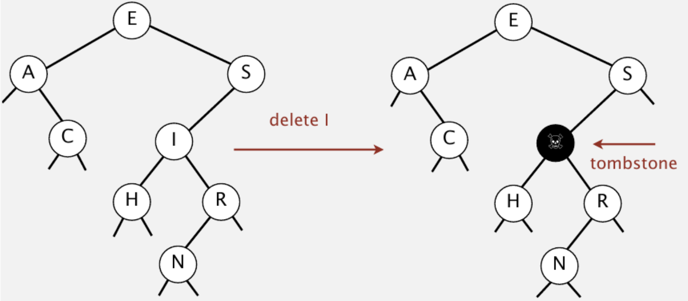
implementation	guarantee			average case			ordered ops?
	search	insert	delete	search hit	insert	delete	
sequential search (linked list)	N	N	N	½ N	N	½ N	
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✔
BST	N	N	N	1.39 lg N	1.39 lg N	? ? ?	✔

A seguir: Remoção de chaves em BSTs.

Remoção em BSTs: método preguiçoso

Para remover um nó com uma dada chave:

- Coloque o valor como `NULL_Value`.
- Deixe a chave na árvore para guiar a busca (mas não considere a chave igual a nada).



Custo: $\sim 2 \ln N'$ por operação (inserção, busca ou remoção), onde N' é o número total de chaves (incluindo lápides).

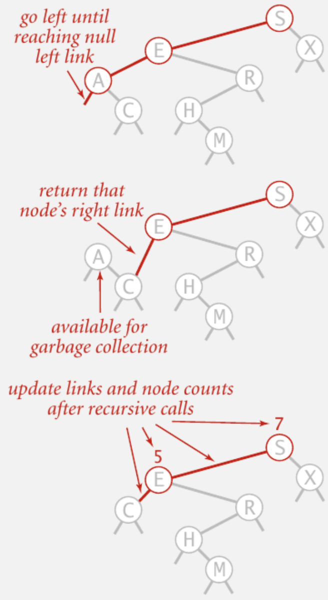
Solução ineficiente: desperdício de memória.

Removendo o mínimo

Para remover a chave mínima:

- Vá para esquerda até encontrar um nó com o filho esquerdo nulo.
- Substitua esse nó pelo seu filho da direita.
- Atualize a contagem de nós da subárvore.

```
Node* rec_delmin(Node *n, bool del) {
    if (n->l == NULL) {
        Node *r = n->r;
        if (del) free(n);
        return r;      }
    n->l = rec_delmin(n->l, del);
    n->size = size(n->l) + size(n->r) + 1;
    return n;
}
void ST_delmin() {
    root = rec_delmin(root, true);
}
```

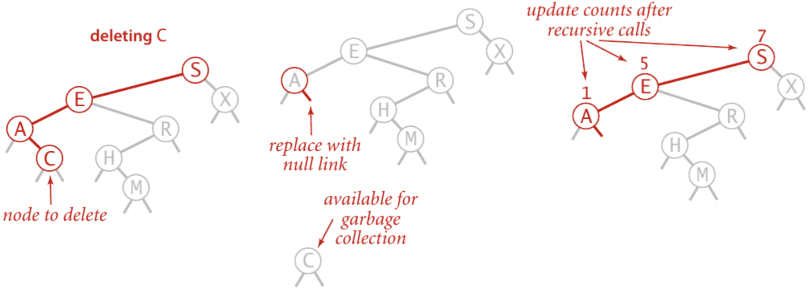


Removendo um nó da BST

Hibbard *deletion* [T. Hibbard, 1962]:

- Garante que as alturas das subárvores envolvidas mudam de no máximo 1.
- Para remover um nó com a chave *k*: busque pelo nó *t* que contém *k*.

Caso: 0 filhos. Apague *t* colocando o ponteiro do pai para nulo.

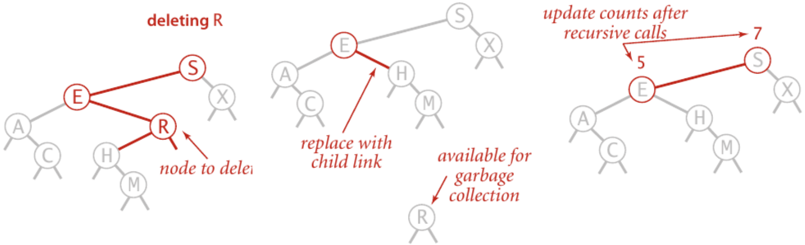


Removendo um nó da BST

Hibbard *deletion* [T. Hibbard, 1962]:

- Garante que as alturas das subárvores envolvidas mudam de no máximo 1.
- Para remover um nó com a chave *k*: busque pelo nó *t* que contém *k*.

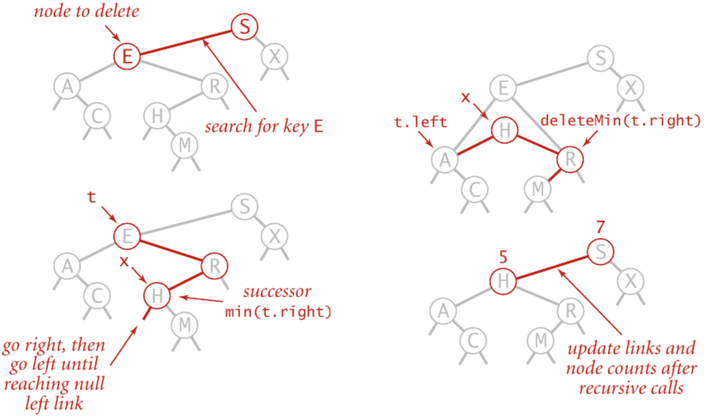
Caso: 1 filho. Apague *t* substituindo-o pelo filho.



Removendo um nó da BST

Caso: 2 filhos.

- Encontre o sucessor x de t. (x não tem filho à esquerda.)
- Remova x da subárvore da direita de t. (Mas não apague!)
- Coloque x no lugar de t. (Continua sendo uma BST.)

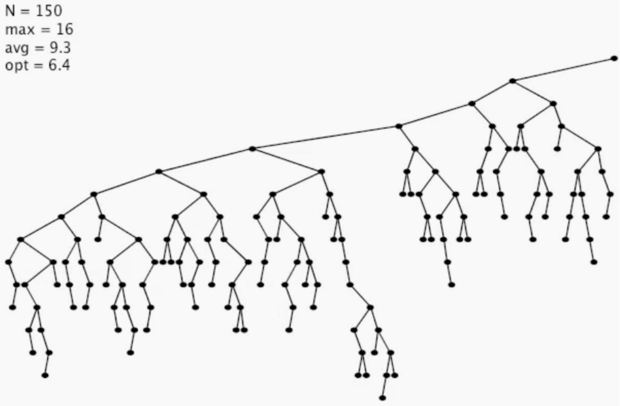


Hibbard *deletion*: implementação em C

```c
static Node* rec_delete(Node *n, Key key) {
    if (n == NULL) { return NULL; }
    int cmp = compare(key, n->key);
    if      (cmp < 0) { n->l = rec_delete(n->l, key); }
    else if (cmp > 0) { n->r = rec_delete(n->r, key); }
    else /*cmp == 0*/ {
        if (n->r == NULL) { Node *l = n->l; free(n); return l; }
        if (n->l == NULL) { Node *r = n->r; free(n); return r; }
        Node *t = n;
        n = rec_min(t->r);
        n->r = rec_delmin(t->r, false);
        n->l = t->l;
        free(t);
    }
    n->size = size(n->l) + size(n->r) + 1;
    return n;
}

void ST_delete(Key key) {
    root = rec_delete(root, key);
}
```

Hibbard *deletion*: análise

Solução não é satisfatória: escolha do sucessor é arbitrária e portanto não simétrica.



Consequência: a árvore não é mais aleatória! $\Rightarrow \sqrt{N}$ per op.

Em aberto: método simples e eficiente de remoção em BSTs.

Implementações de tabelas de símbolos: sumário

implementation	guarantee			average case			ordered ops?
	search	insert	delete	search hit	insert	delete	
sequential search (linked list)	N	N	N	½ N	N	½ N	
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✔
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔

other operations also become √N
if deletions allowed

Próxima aula: Desempenho logarítmico garantido para todas as operações.