Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Algoritmos e Estruturas de Dados I

Algoritmos de Busca

Mirtha Lina Fernández Venero
Sala 529-2, Bloco A
mirtha.lina@ufabc.edu.br
http://professor.ufabc.edu.br/~mirtha.lina/aedi.html

23 de março de 2019

Agenda

Introdução

Busca: Operação de recuperar uma informação numa coleção de dados

- Cada elemento da coleção é chamado de **registro**
- Cada registro inclui um (ou vários) campo **chave** que é usado no critério ou condição de busca
- Usualmente existe uma **ordem total** entre chaves
- A busca pode terminar com sucesso (*hit*, *match*) ou não (*miss*). Pode retornar a posição do registro, seu o valor (i.e. todo o registro ou parte dele), todos os registros, etc

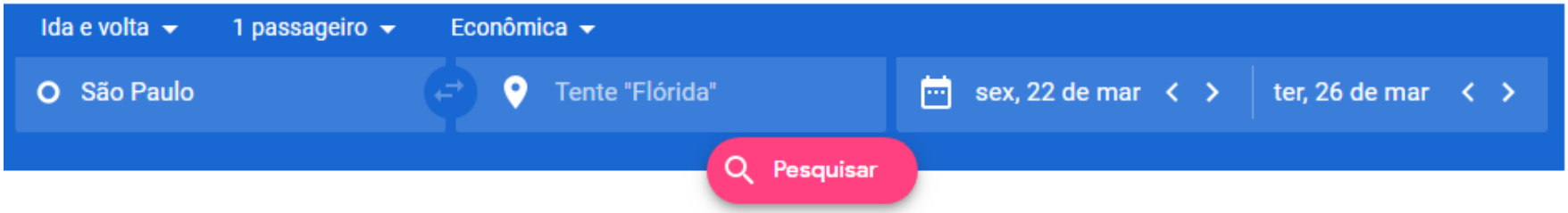
Exemplos?:

Introdução

Busca: Operação de recuperar uma informação numa coleção de dados

- Cada elemento da coleção é chamado de **registro**
- Cada registro inclui um (ou vários) campo **chave** que é usado no critério ou condição de busca
- Usualmente existe uma **ordem total** entre chaves
- A busca pode terminar com sucesso (*hit*, *match*) ou não (*miss*). Pode retornar a posição do registro, seu o valor (i.e. todo o registro ou parte dele), todos os registros, etc

Exemplos?: palavra no dicionário, informação na web ou site



Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Agenda

Busca sequencial - dados não ordenados

Busca numa coleção de registros não ordenados

- Percorrer estrutura de dados (vetores ou listas) até acharmos a chave ou chegarmos no final.

Busca sequencial - dados não ordenados

Busca numa coleção de registros não ordenados

- Percorrer estrutura de dados (vetores ou listas) até acharmos a chave ou chegarmos no final.

```
typedef struct LinkedNode LinkedNode;
struct LinkedNode {
  Key key;
  Value value;
  LinkedNode *next;
};

LinkedNode *search(LinkedNode *first, Key key) {
  while ( first != NULL && first->key != key )
    first = first->next;
  return first;
}
```

Busca sequencial - dados não ordenados

Busca numa coleção de registros não ordenados

- Percorrer estrutura de dados (vetores ou listas) até acharmos a chave ou chegarmos no final. **Custo** $O(n)$. Dá para melhorar?

```
typedef struct LinkedNode LinkedNode;
struct LinkedNode {
  Key key;
  Value value;
  LinkedNode *next;
};

LinkedNode *search(LinkedNode *first, Key key) {
  while ( first != NULL && first->key != key )
    first = first->next;
  return first;
}
```

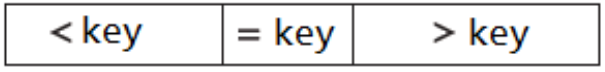
Busca sequencial - dados não ordenados

- Para **vetores** pode ser usada uma sentinela que evita o teste de fim de lista

```
typedef struct Node Node;
struct Node {
  Key key;
  Value *value;
};

int searchArr(Node *arr, int n, Key key) {
  int i = 0;
  arr[n].key = key;
  while (arr[i].key != key)
    i++;
  if (i < n)
    return i;
  return -1;
}
```

Busca sequencial - dados ordenados

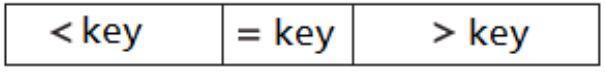


search hit

search miss

- Percorrer a estrutura de dados (vetor ou lista) até acharmos a chave **ou uma maior do que ela** ou chegarmos no final

Busca sequencial - dados ordenados

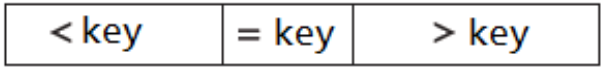


search hit

search miss

- Percorrer a estrutura de dados (vetor ou lista) até acharmos a chave **ou uma maior do que ela** ou chegarmos no final
- O que muda no código anterior? Custo no caso pior?

Busca sequencial - dados ordenados

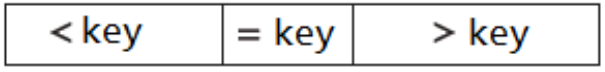


search hit

search miss

- Percorrer a estrutura de dados (vetor ou lista) até acharmos a chave **ou uma maior do que ela** ou chegarmos no final
- O que muda no código anterior? Custo no caso pior? $O(n)$. Dá para melhorar?

Busca sequencial - dados ordenados



search hit

search miss

- Percorrer a estrutura de dados (vetor ou lista) até acharmos a chave **ou uma maior do que ela** ou chegarmos no final
- O que muda no código anterior? Custo no caso pior? $O(n)$. Dá para melhorar?

Se as chaves estão ordenadas, por que começar no início? por que não pular algumas chaves?

Busca num dicionário

- Abrir numa pagina qualquer (posição aleatória)
- Se achamos o que procuramos, pronto!
- Se o que procuramos está antes então descartar o restante
- Se o que procuramos está depois então descartar o anterior
- Até acharmos o que procuramos ou perceber que não está

Universidade Federal do ABC

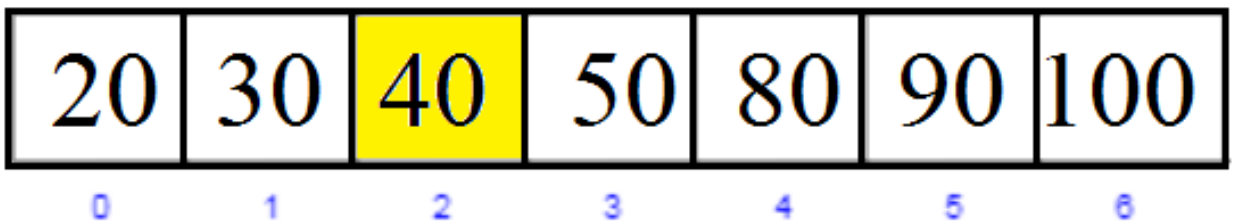
CMCC Centro de Matemática, Computação e Cognição

Agenda

Busca binária

Aplica o principio da **dicotomia**

- Começar na metade do arranjo
- Se achamos a chave procurada (k), pronto!
- Se k é menor que o elemento do médio, descartar metade superior
- Se k é maior que o elemento do médio, descartar metade inferior
- Aplicar recursivamente até acharmos k ou o arranjo ficar vazio



Searching Sorted List Visualization https://www.cs.usfca.edu/~galles/visualization/Search.html

Busca binária (variante recursiva)

```
50 int binarySearch(Node *arr, int left, int right, Key key) {
51   if (left > right)
52     return -1;          //miss
53
54   int mid = left + (right - left) / 2;
55
56   if (arr[mid].key < key)
57     // key must be in right subarray
58     return binarySearch(arr, mid + 1, right, key);
59
60   if (arr[mid].key > key)
61     // key must be in left subarray
62     return binarySearch(arr, left, mid - 1, key);
63
64   return mid;           //hit
65 }
66
```

Ver considerações sobre a corretude da versão iterativa em Requiem for a Bug - Verifying Software, Part 2: Formal Verification through SPARK 2014, Johannes Kanig, Electronic Design, 2015

Busca binária - Análise do algoritmo - Caso pior

Sempre descarta metade do arranjo até `left > right`, i.e. a chave procurada não está no vetor

$$T(n) = T(n/2) + O(1)$$

Busca binária - Análise do algoritmo - Caso pior

Sempre descarta metade do arranjo até `left > right`, i.e. a chave procurada não está no vetor

$$T(n) = T(n/2) + O(1)$$

Relembrando a variante do Teorema Mestre:

$$T(n) = aT(\frac{n}{b}) + n^c$$

Busca binária - Análise do algoritmo - Caso pior

Sempre descarta metade do arranjo até `left > right`, i.e. a chave procurada não está no vetor

$$T(n) = T(n/2) + O(1)$$

Relembrando a variante do Teorema Mestre:

$$T(n) = aT(\frac{n}{b}) + n^c$$

Neste caso $a = 1, b = 2, \log_b^a = 0 = c$. Logo, aplica-se o caso:

2. $\log_b^a = c \Rightarrow T(n) = \Theta(n^c * \log_b^n)$

$$\Rightarrow T(n) = \Theta(\log n)$$

Busca sequencial vs Busca binária

Técnica	Ordem	Busca	Inserção	Remoção
Busca Sequencial	Não	N	N	N
Busca Binária	Sim	log(N)	N	N

- Busca sequencial é simples porém ineficiente
- A busca binária tem custo $O(\log n)$ porém o vetor precisa estar ordenado. Sua eficiência da busca binária está baseada na indexação em tempo constante de qualquer elemento
- Se o vetor é pequeno a busca linear pode ser mais eficiente que ordenar e usar busca binária
- Inserções e remoções arbitrárias têm custo $O(n)$

Busca sequencial vs Busca binária

Técnica	Ordem	Busca	Inserção	Remoção
Busca Sequencial	Não	N	N	N
Busca Binária	Sim	log(N)	N	N

- Busca sequencial é simples porém ineficiente
- A busca binária tem custo $O(\log n)$ porém o vetor precisa estar ordenado. Sua eficiência da busca binária está baseada na indexação em tempo constante de qualquer elemento
- Se o vetor é pequeno a busca linear pode ser mais eficiente que ordenar e usar busca binária
- Inserções e remoções arbitrárias têm custo $O(n)$

Existem estruturas não lineares que permitem a busca em tempo $O(\log n)$ e inserções/remoções mais eficientes?

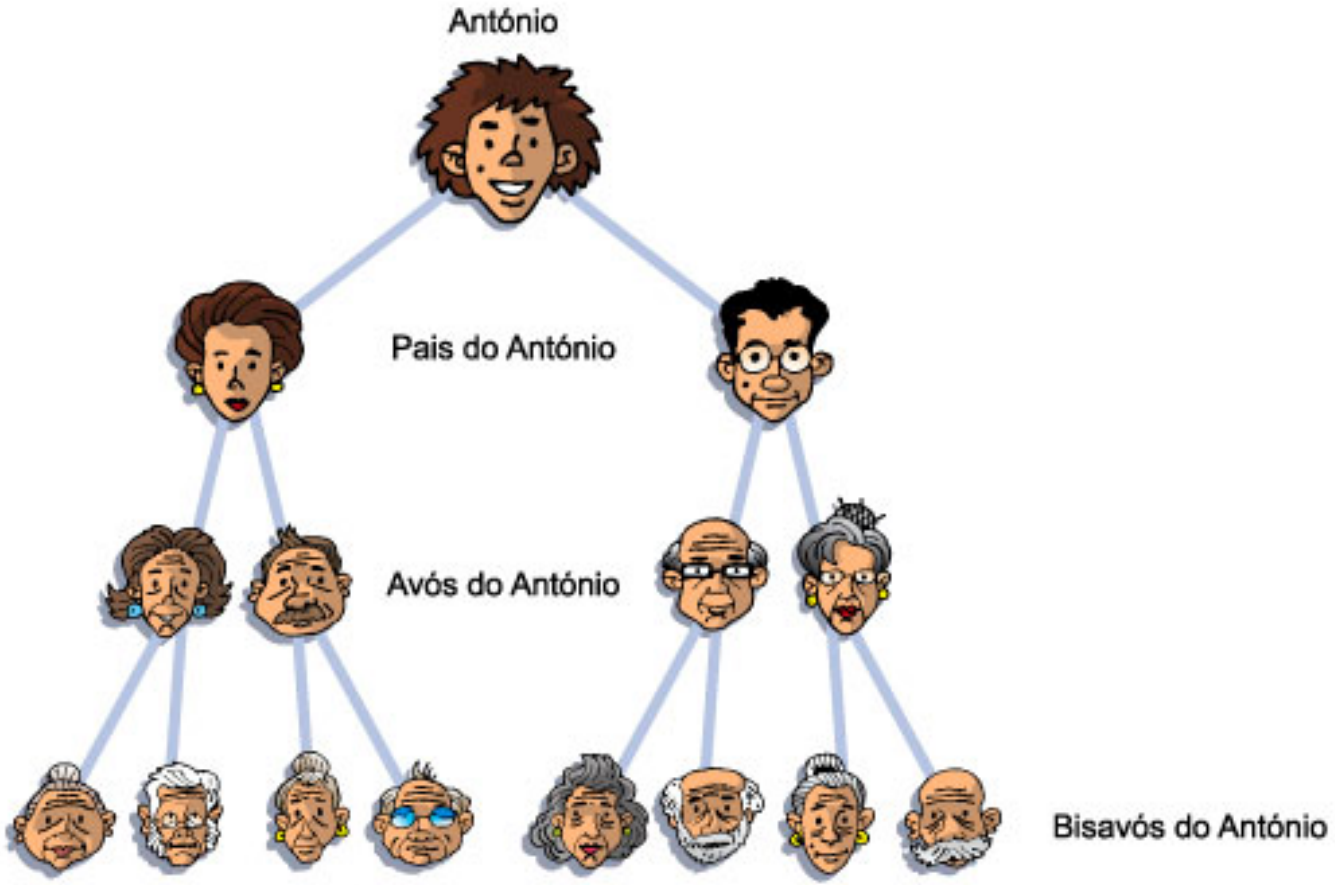
Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Agenda

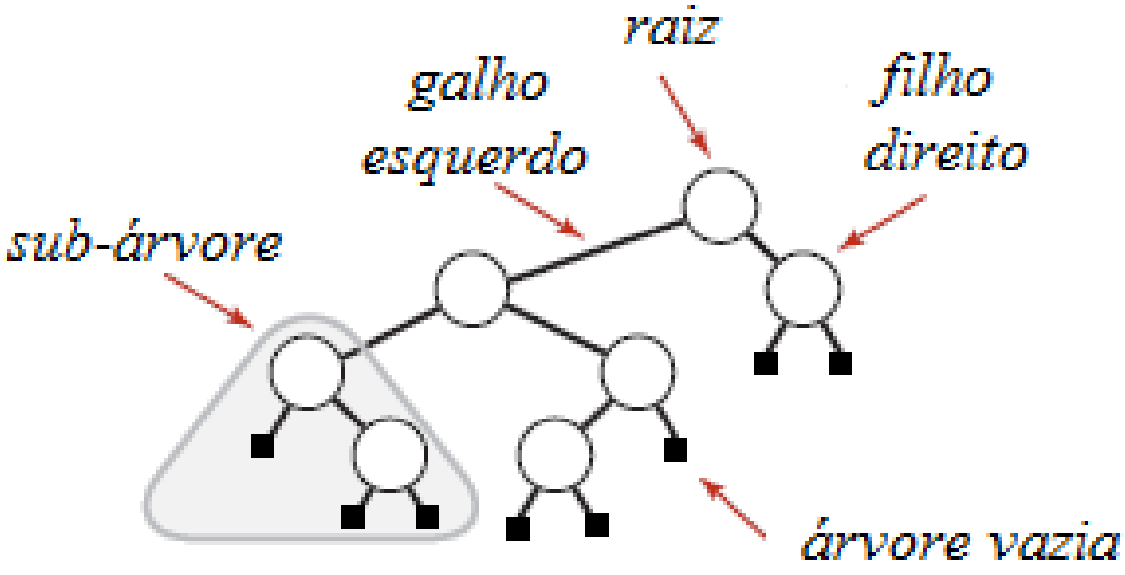
Árvore binária - Definição recursiva

1. Um único nó (folha)
2. Um nó raiz com duas sub-árvores binárias disjuntas



Árvore binária em Computação - Definição recursiva

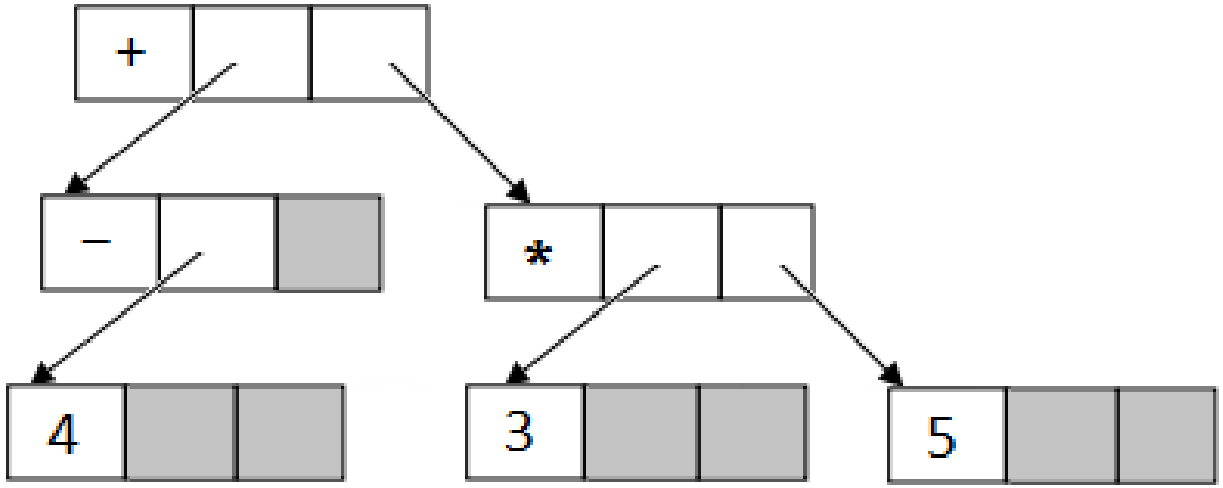
1. Árvore vazia
2. Um nó com duas sub-árvores binárias disjuntas



Árvore binária em Computação - Definição recursiva

1. Árvore vazia
2. Um nó com duas sub-árvores binárias disjuntas

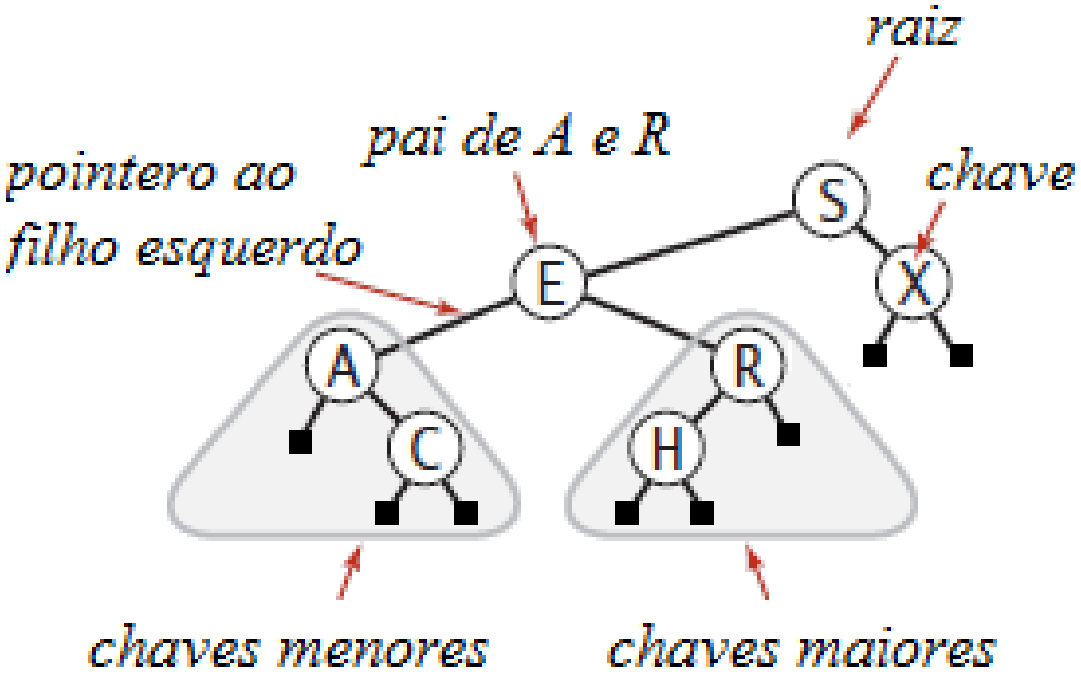
```
typedef struct TreeNode TreeNode;
struct TreeNode {
  Value value;
  TreeNode *pLeft, *pRight;
};
```



Árvore Binária de Busca (ABB)

Inventada por P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard em 1960, é uma árvore na qual a chave de cada nó é

- maior que qualquer chave na sub-árvore da esquerda
- menor que qualquer chave na sub-árvore da direita

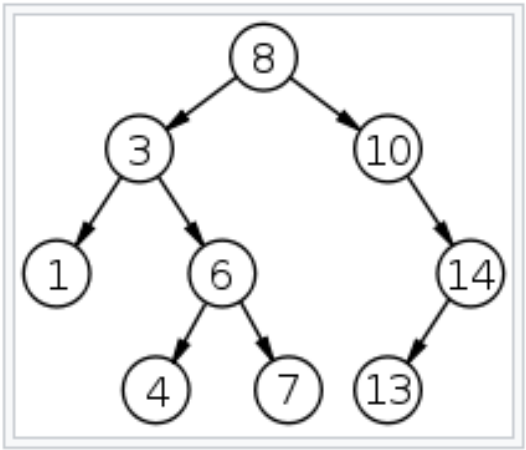


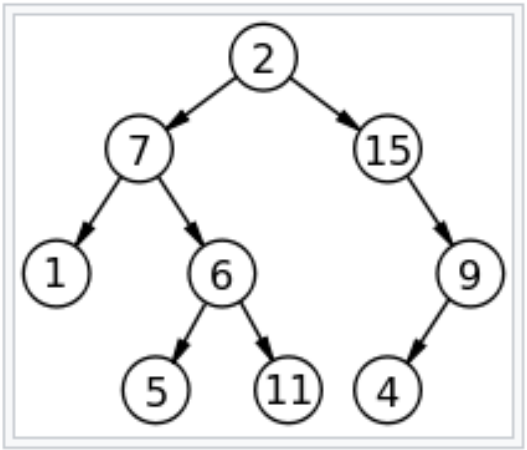
Árvore Binária de Busca (ABB)

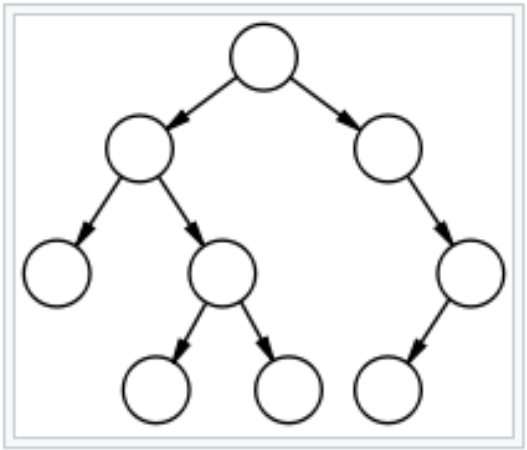
Inventada por P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard em 1960, é uma árvore na qual a chave de cada nó é

- maior que qualquer chave na sub-árvore da esquerda
- menor que qualquer chave na sub-árvore da direita

Exemplo: Qual das seguintes árvores binárias **não** é de busca? Preencha os nós da árvore da direita para obter uma ABB com as mesmas chaves daquela que não é







Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Árvore Binária de Busca (ABB)

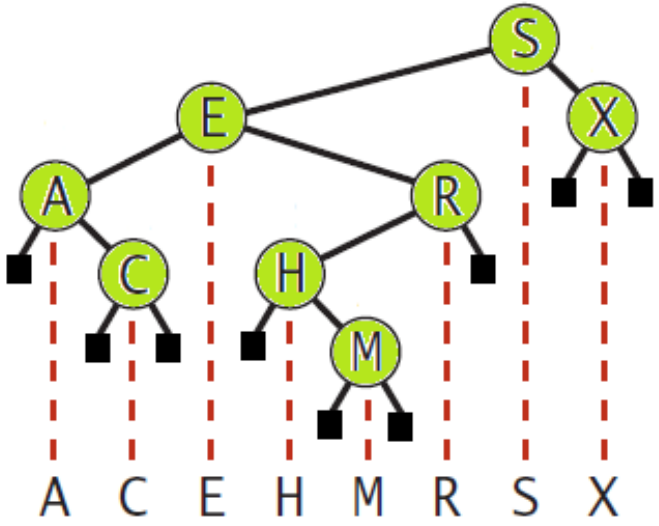
Inventada por P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard em 1960, é uma árvore na qual a chave de cada nó é

- maior que qualquer chave na sub-árvore da esquerda
- menor que qualquer chave na sub-árvore da direita

```
typedef struct TreeNode TreeNode;
struct TreeNode {
  Key key;
  Value value;
  TreeNode *pLeft, *pRight;
};
```

Ordem Linear duma Árvore Binária

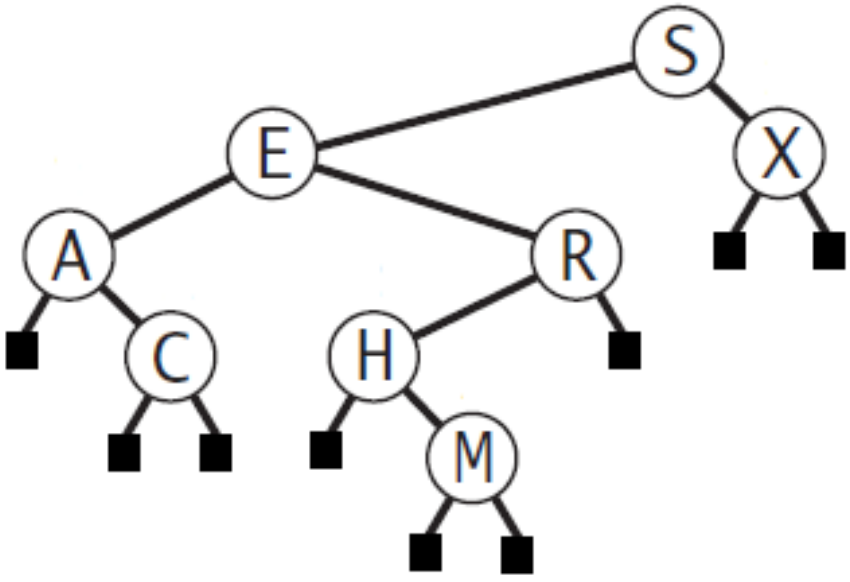
- As árvores são estruturas de dados não lineares
- Para obter uma ordem linear é preciso percorrer todos seus nós
- Existem vários percursos, e.g. em ordem, pre-ordem, pós-ordem
- Nas ABBs o percurso em ordem gera as chaves ordenadas



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

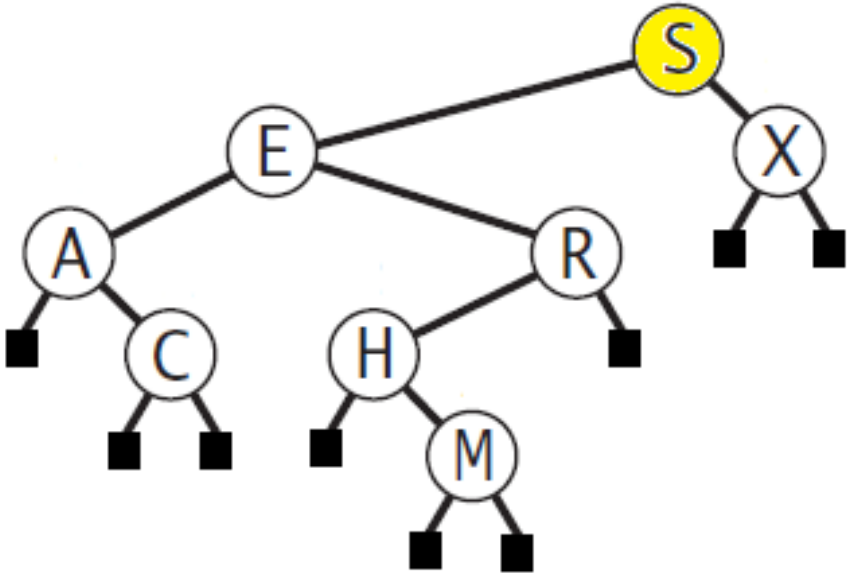
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

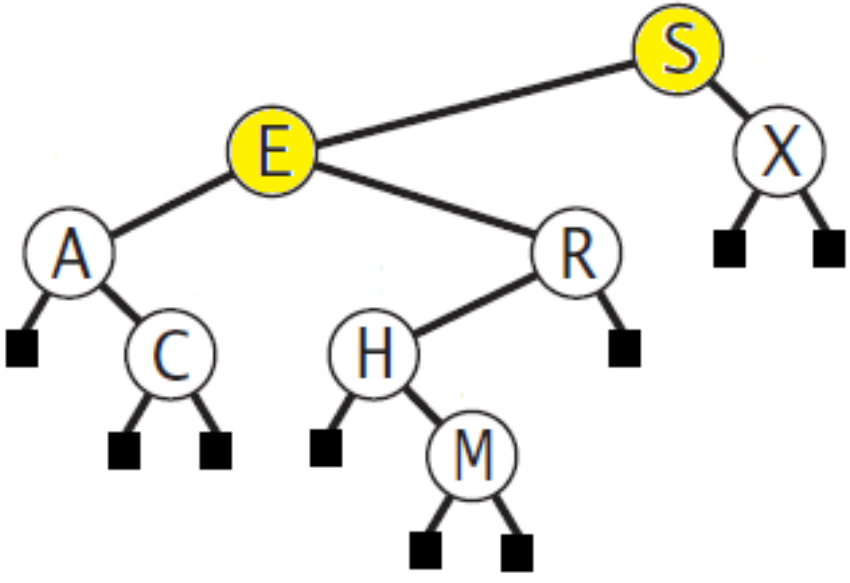
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

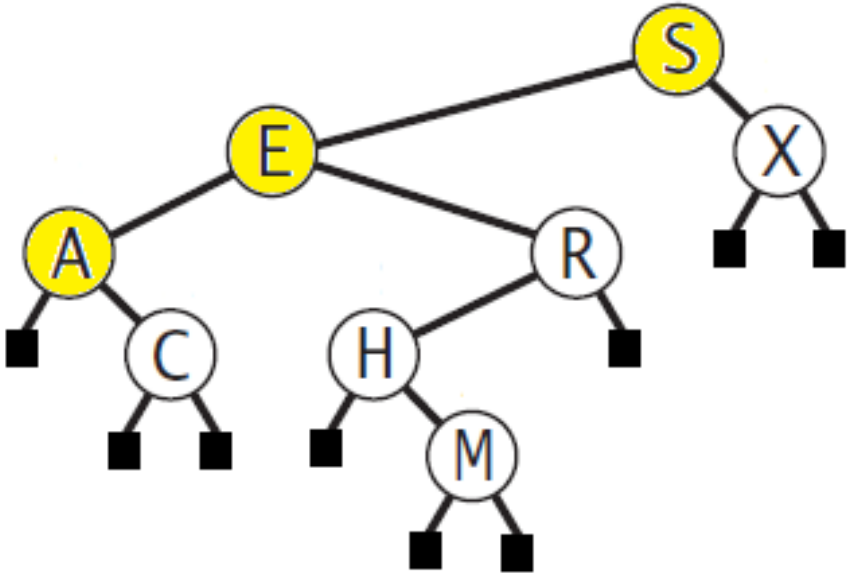
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

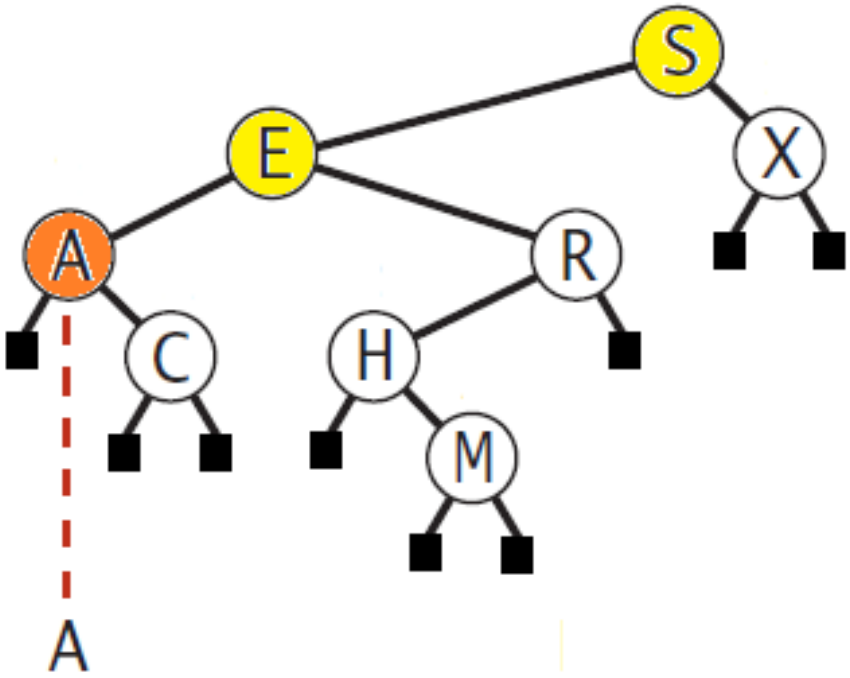
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

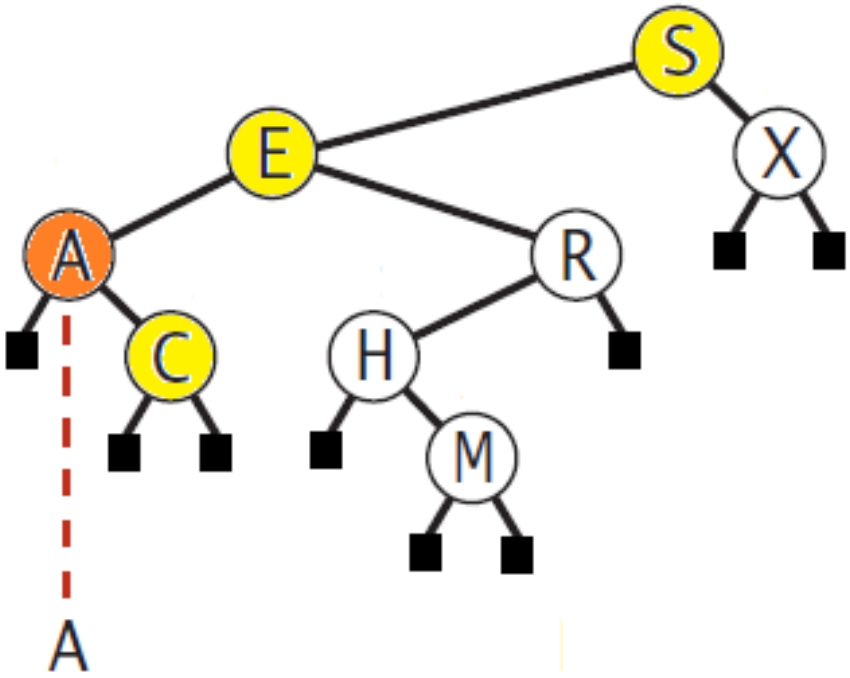
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

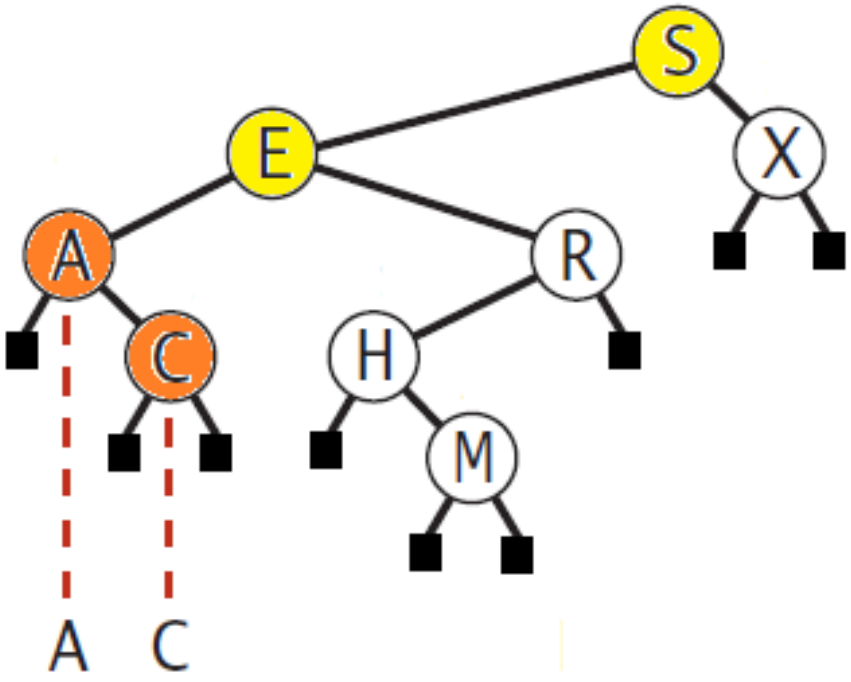
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

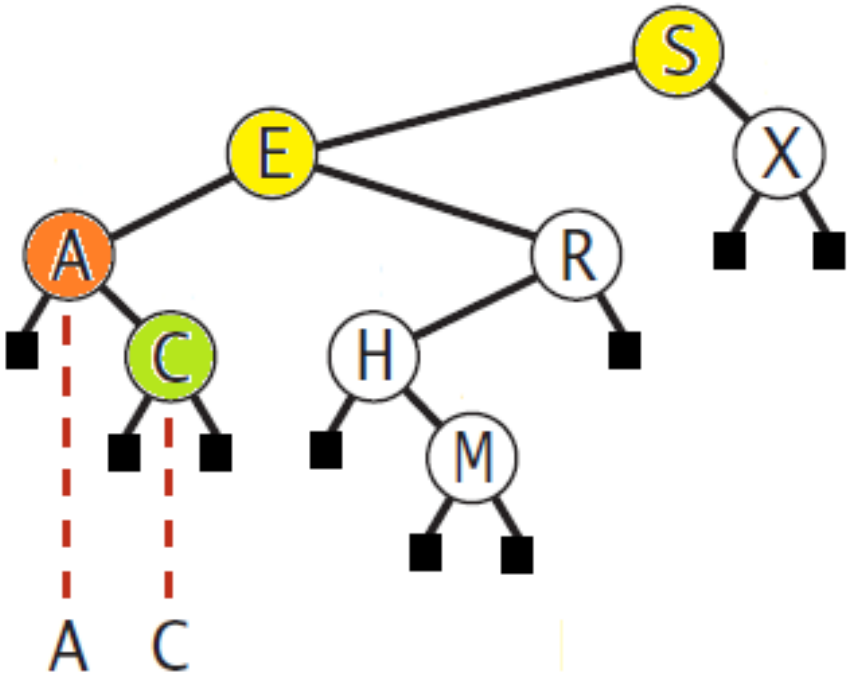
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

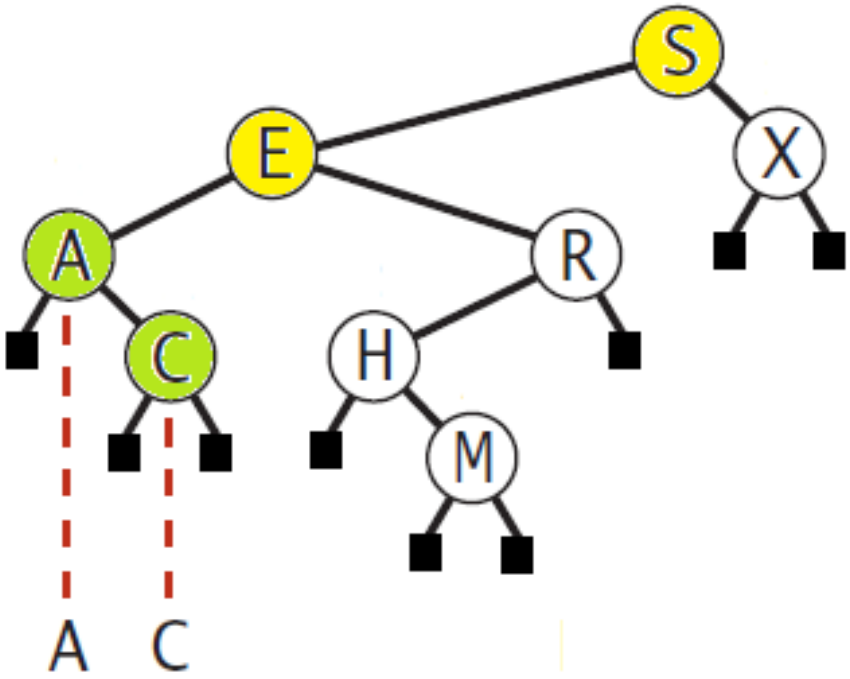
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

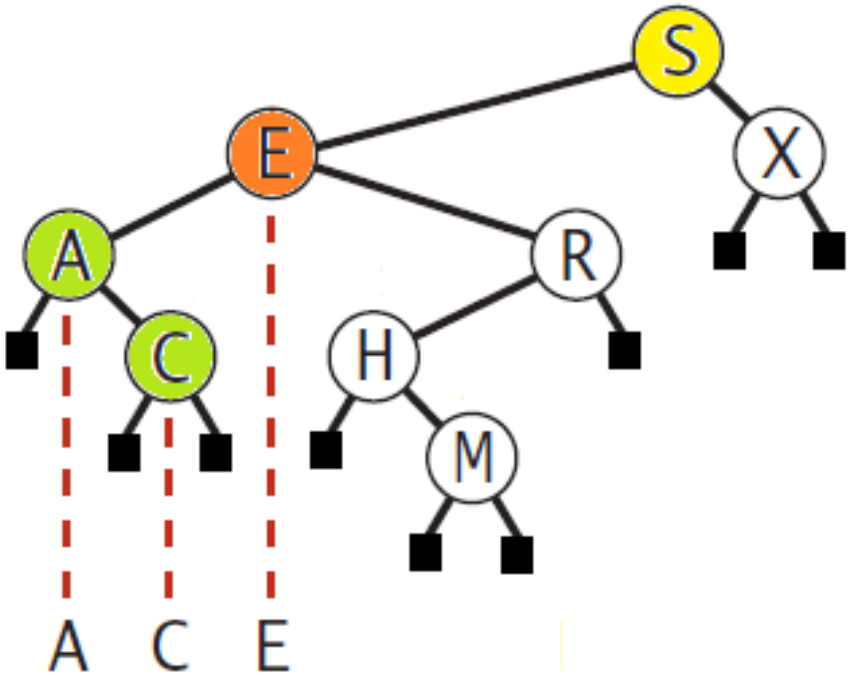
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

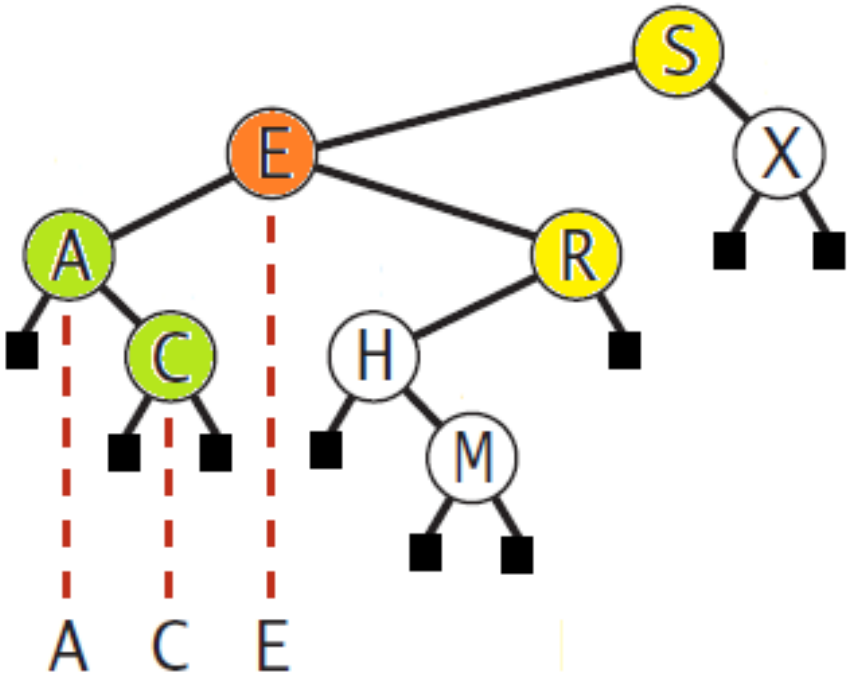
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

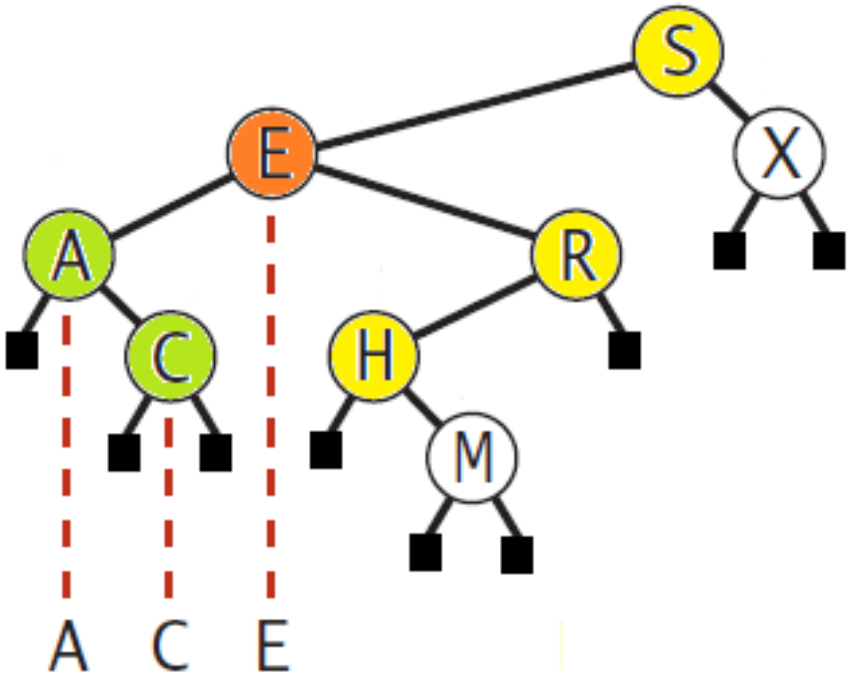
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

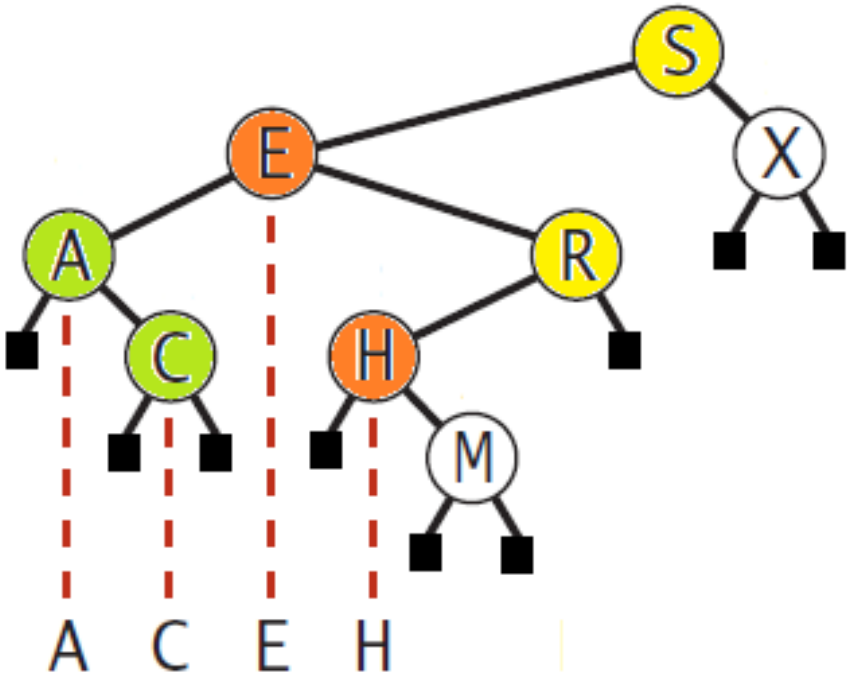
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

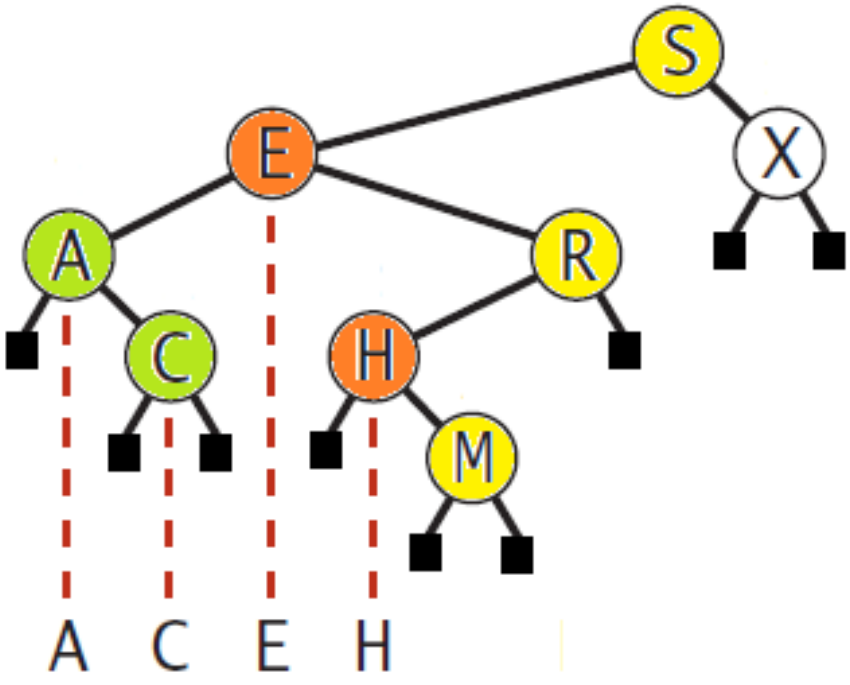
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

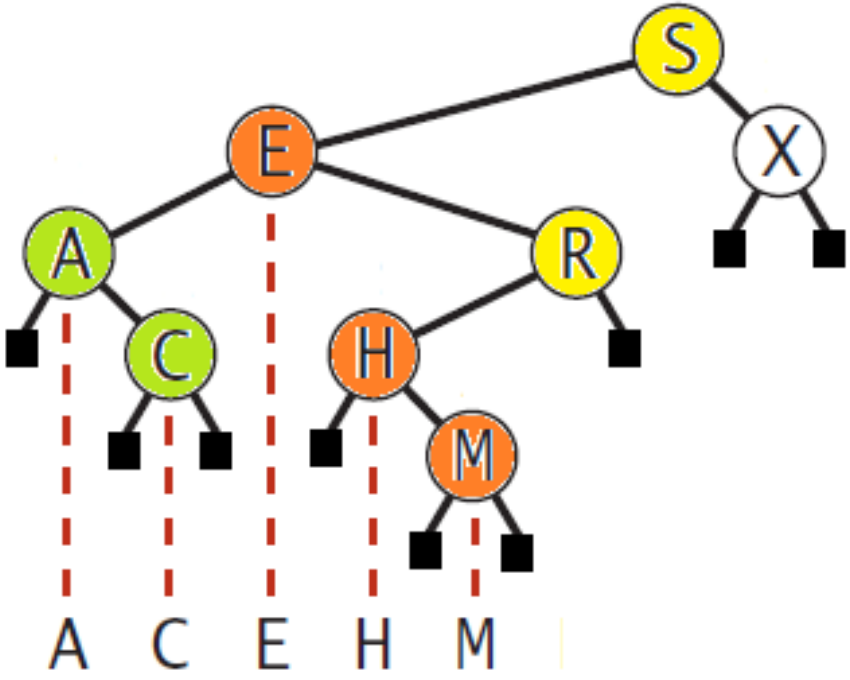
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

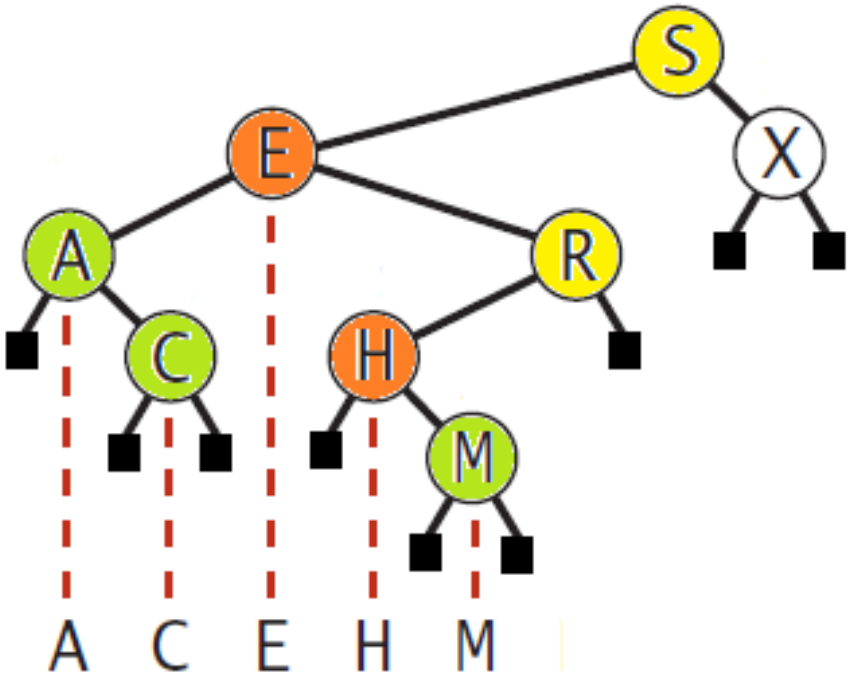
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

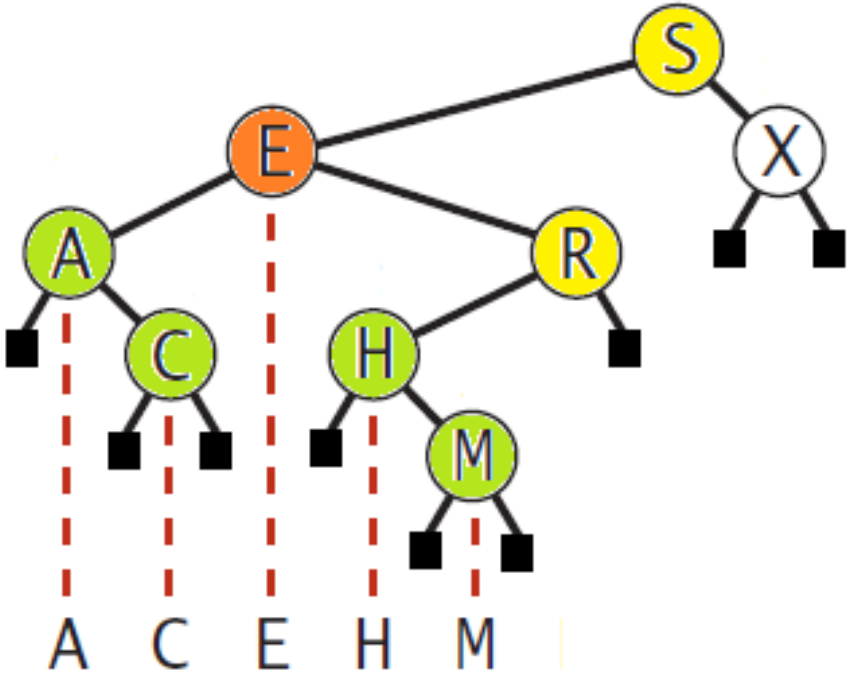
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

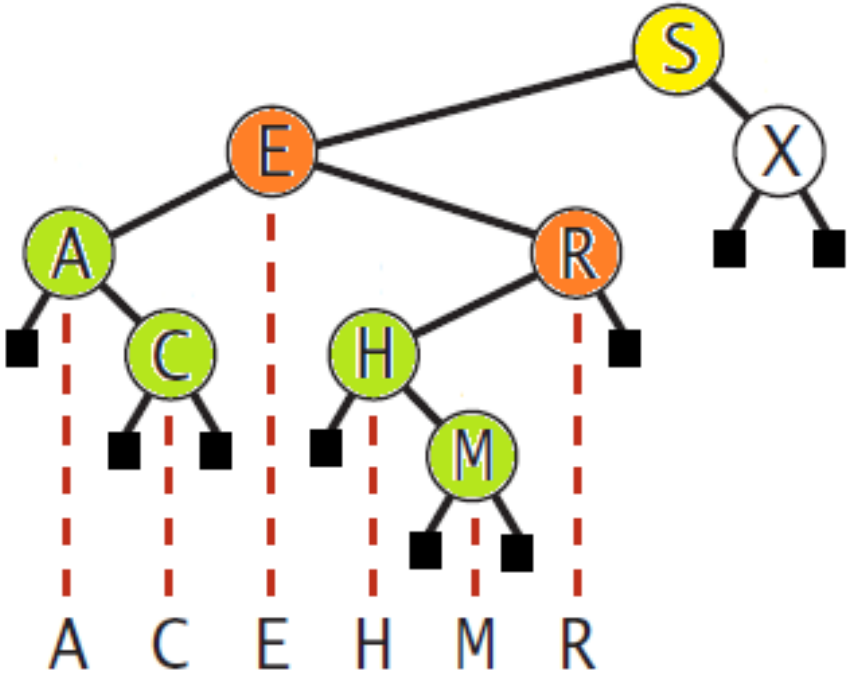
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

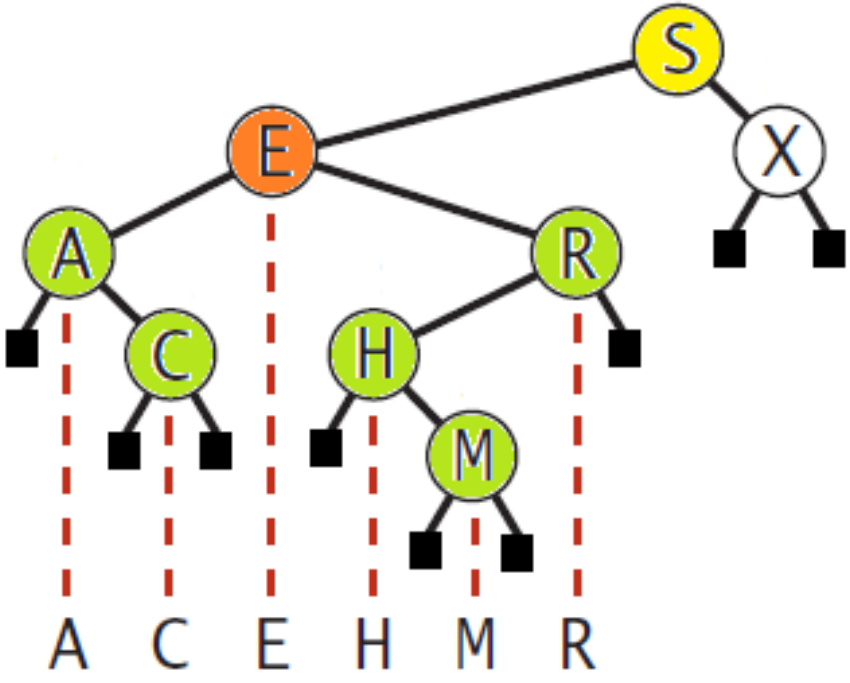
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

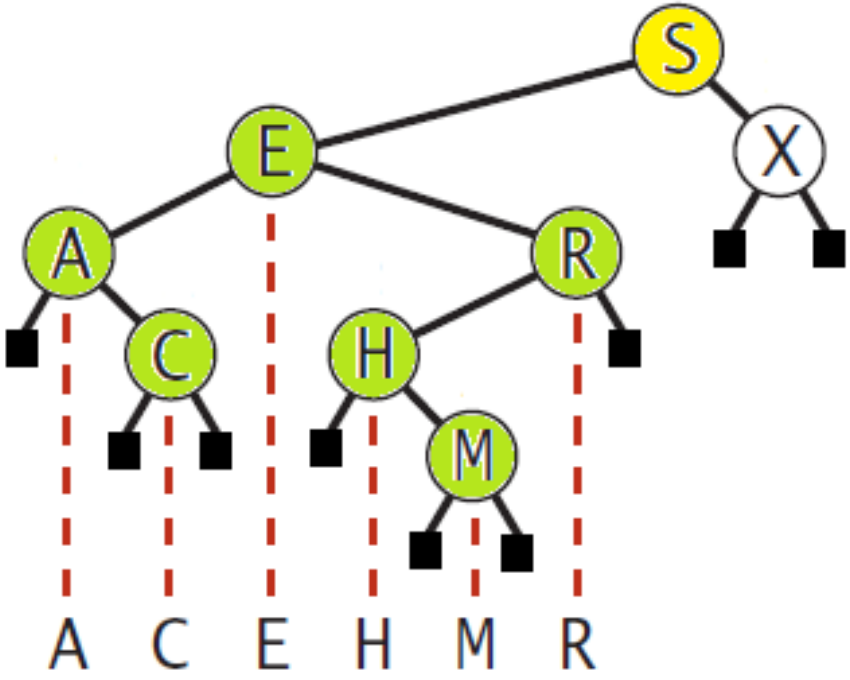
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

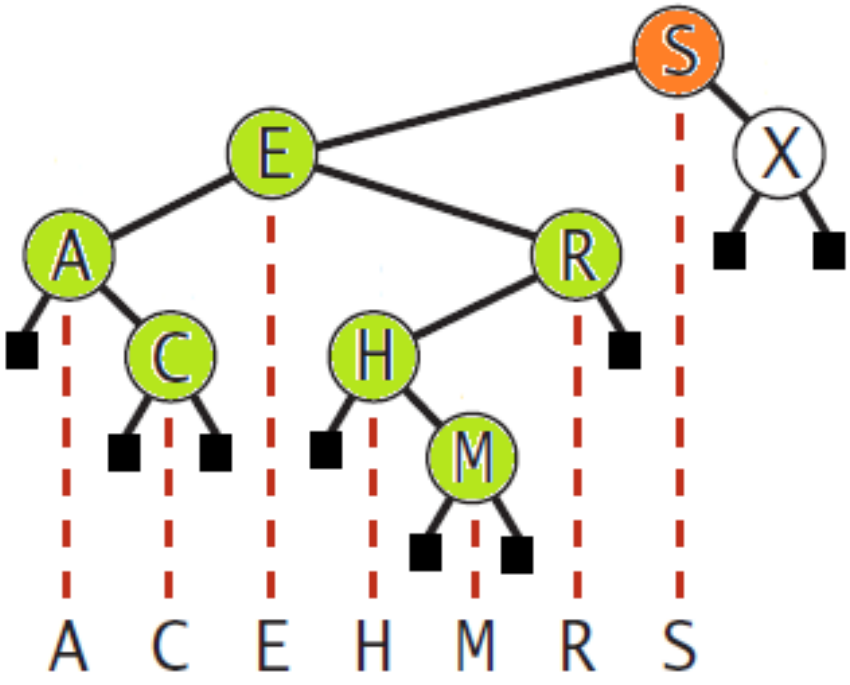
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

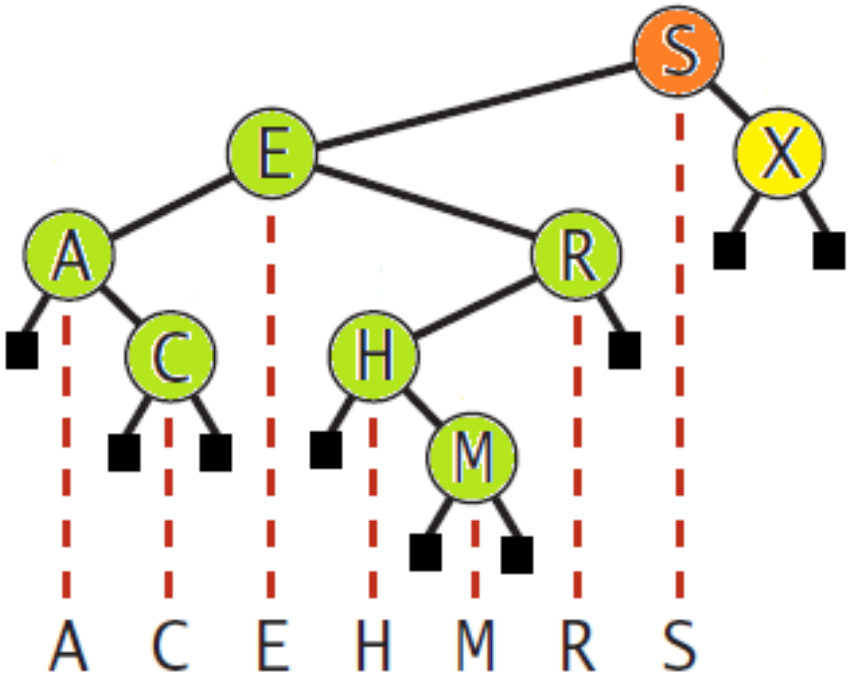
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

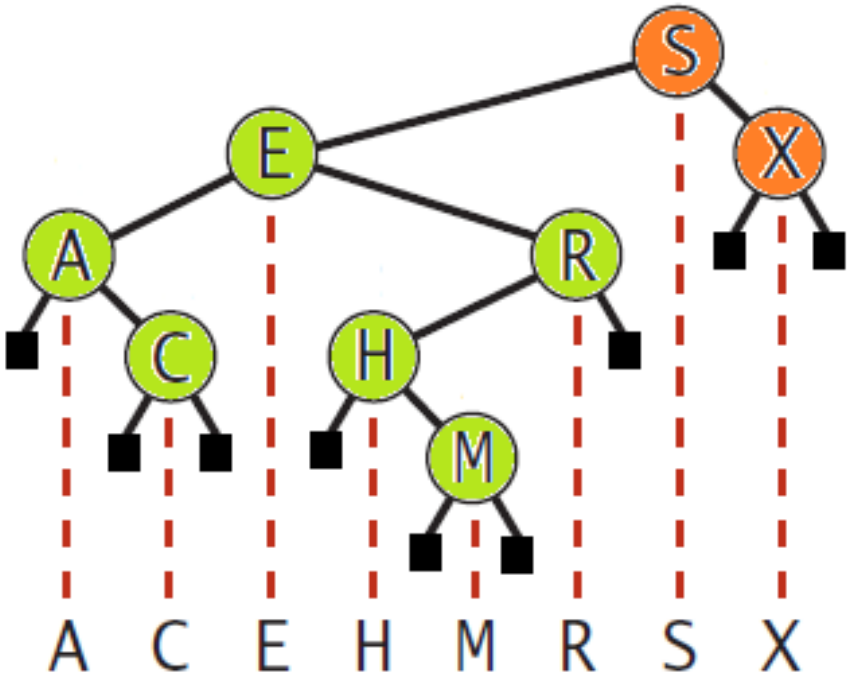
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

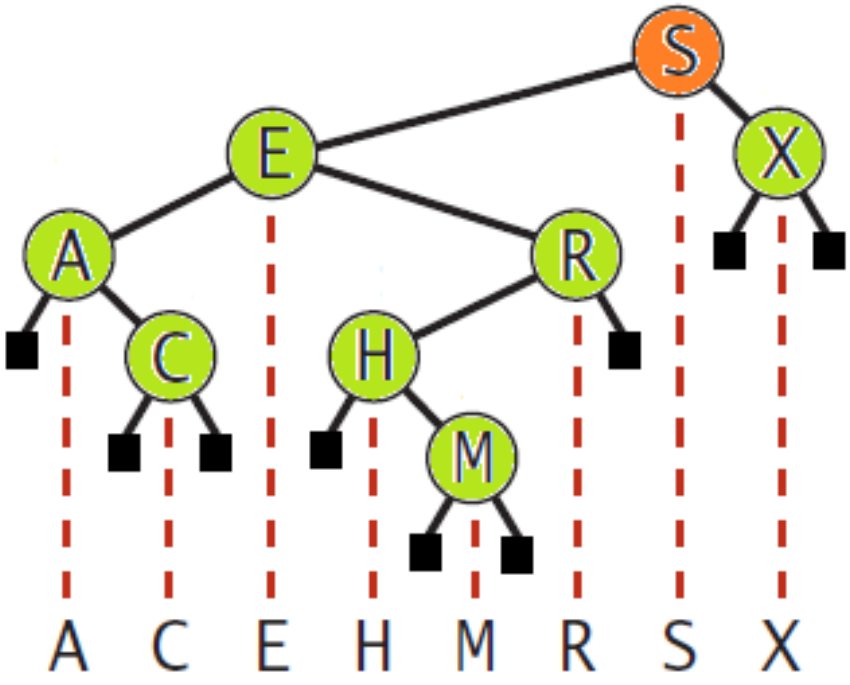
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

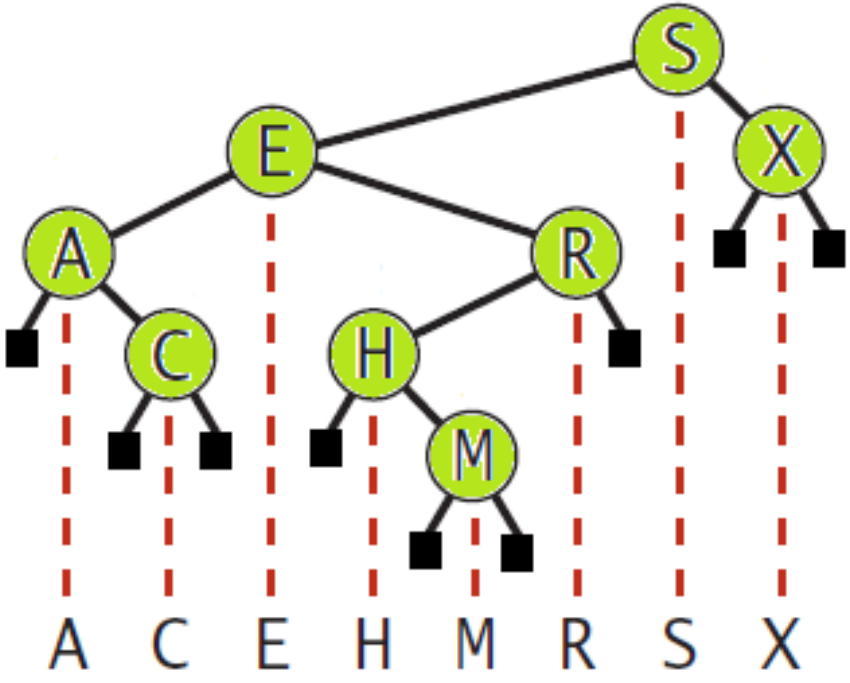
```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



Percurso em ordem duma Árvore Binária de Busca

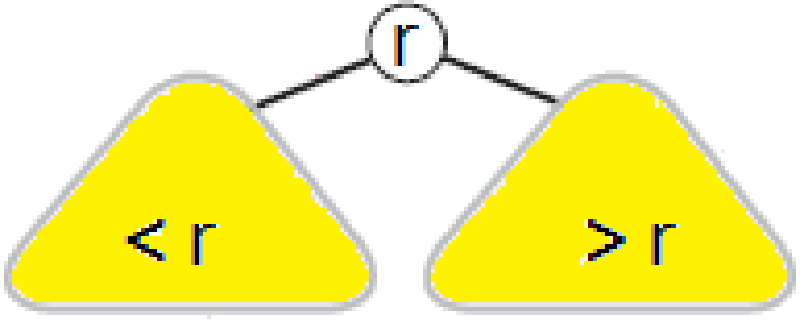
- percorrer recursivamente a sub-árvore da esquerda
- imprimir a chave e/ou o valor da raiz
- percorrer recursivamente a sub-árvore da direita

```
void inOrder(TreeNode *t) {
  if (t == NULL)
    return;
  inOrder(t->pLeft);
  printf("%d", t->key);
  inOrder(t->pRight);
}
```



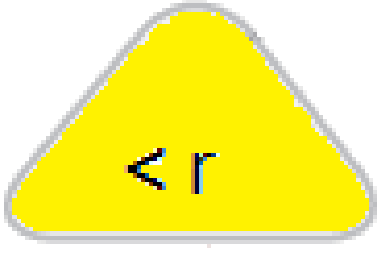
Busca nas ABBs

- Começar na raiz da árvore
- Se a chave procurada (k) está na raiz (r), pronto!
- Se k é menor que r, procurar continuar a busca só na sub-árvore da esquerda. Se k é maior que r, procurar continuar a busca só na sub-árvore da direita
- Aplicar recursivamente até acharmos k ou a árvore ficar vazia



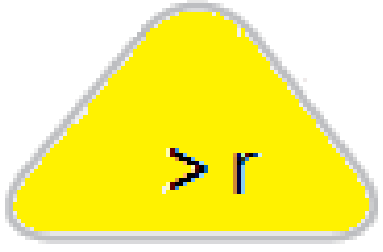
Busca nas ABBs

- Começar na raiz da árvore
- Se a chave procurada (k) está na raiz (r), pronto!
- Se k é menor que r, procurar continuar a busca só na sub-árvore da esquerda. Se k é maior que r, procurar continuar a busca só na sub-árvore da direita
- Aplicar recursivamente até acharmos k ou a árvore ficar vazia



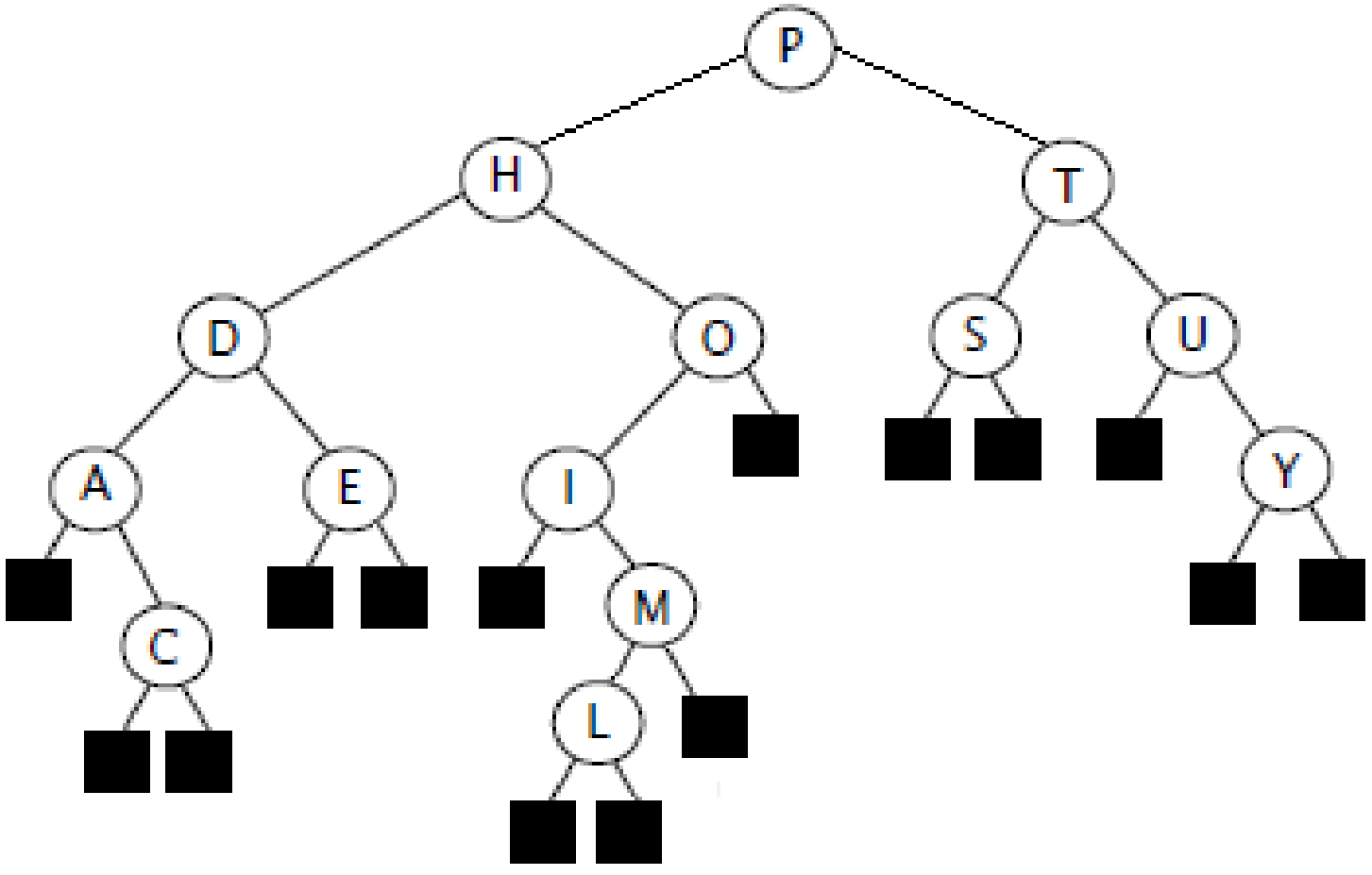
Busca nas ABBs

- Começar na raiz da árvore
- Se a chave procurada (k) está na raiz (r), pronto!
- Se k é menor que r, procurar continuar a busca só na sub-árvore da esquerda. Se k é maior que r, procurar continuar a busca só na sub-árvore da direita
- Aplicar recursivamente até acharmos k ou a árvore ficar vazia



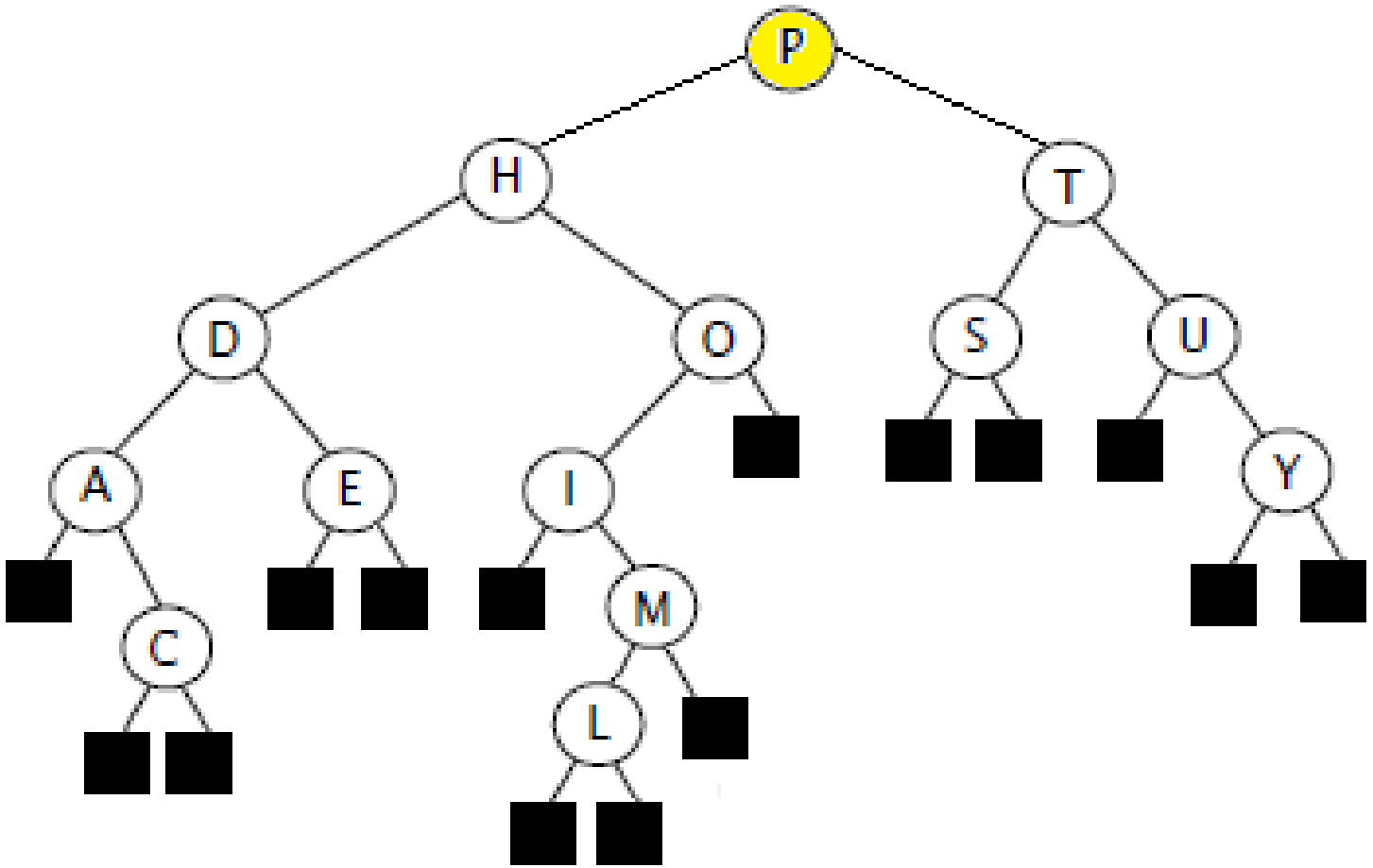
Busca nas ABBs -Exemplo: buscar **M**

- Começar na raiz da árvore



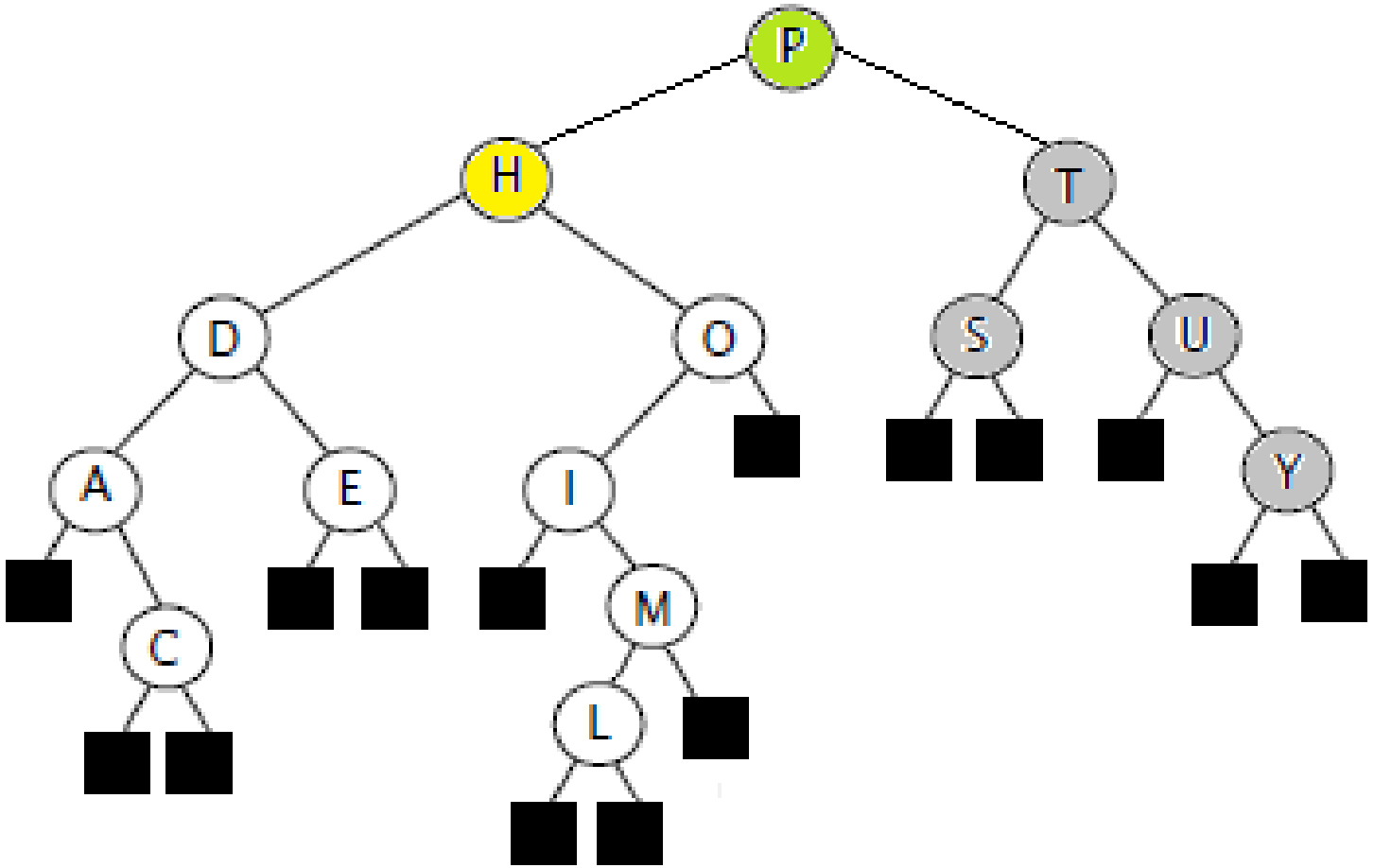
Busca nas ABBs -Exemplo: buscar **M**

- M $<$ P $\Rightarrow$ buscar à esquerda



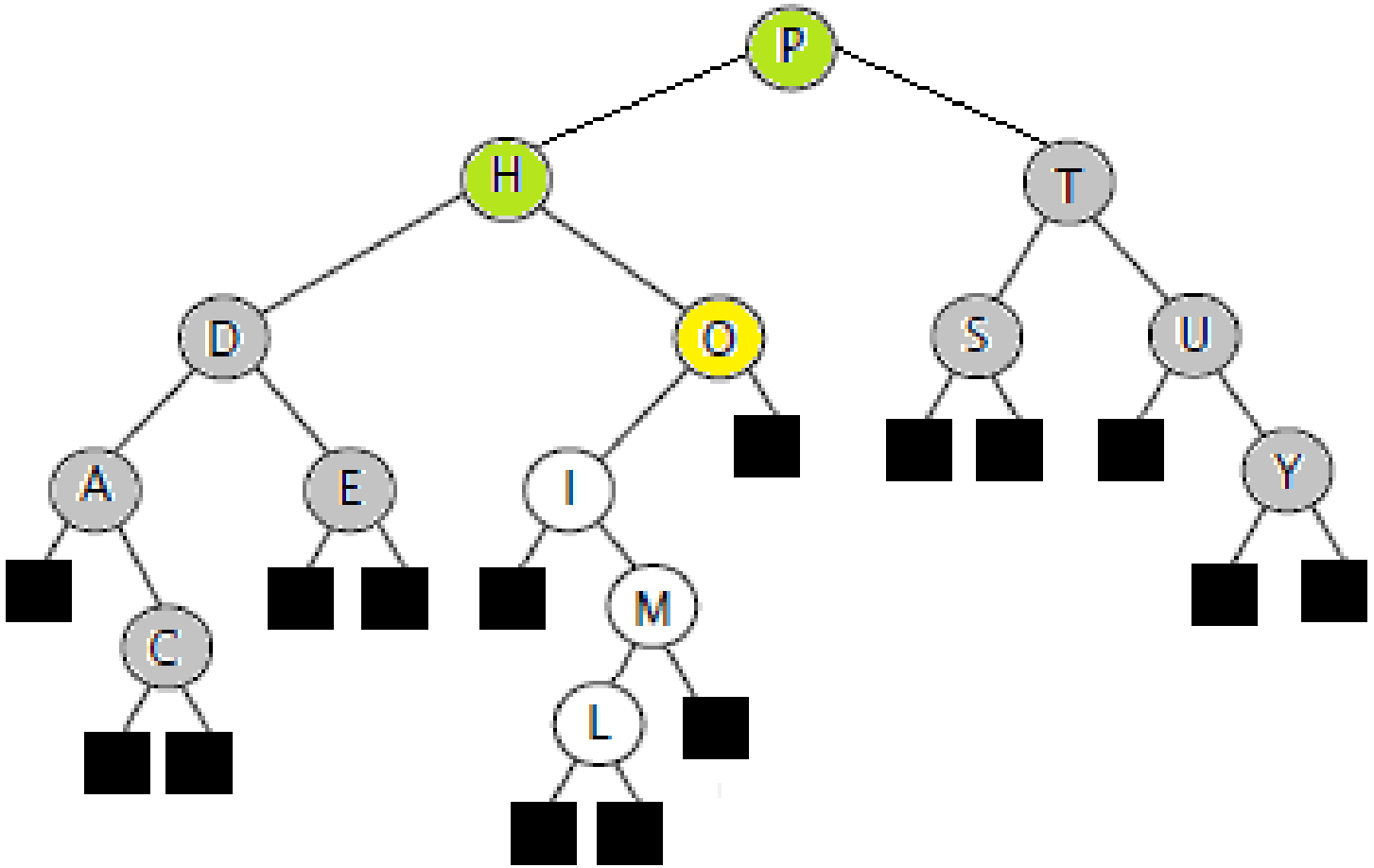
Busca nas ABBs -Exemplo: buscar **M**

- M $>$ H $\Rightarrow$ buscar à direita



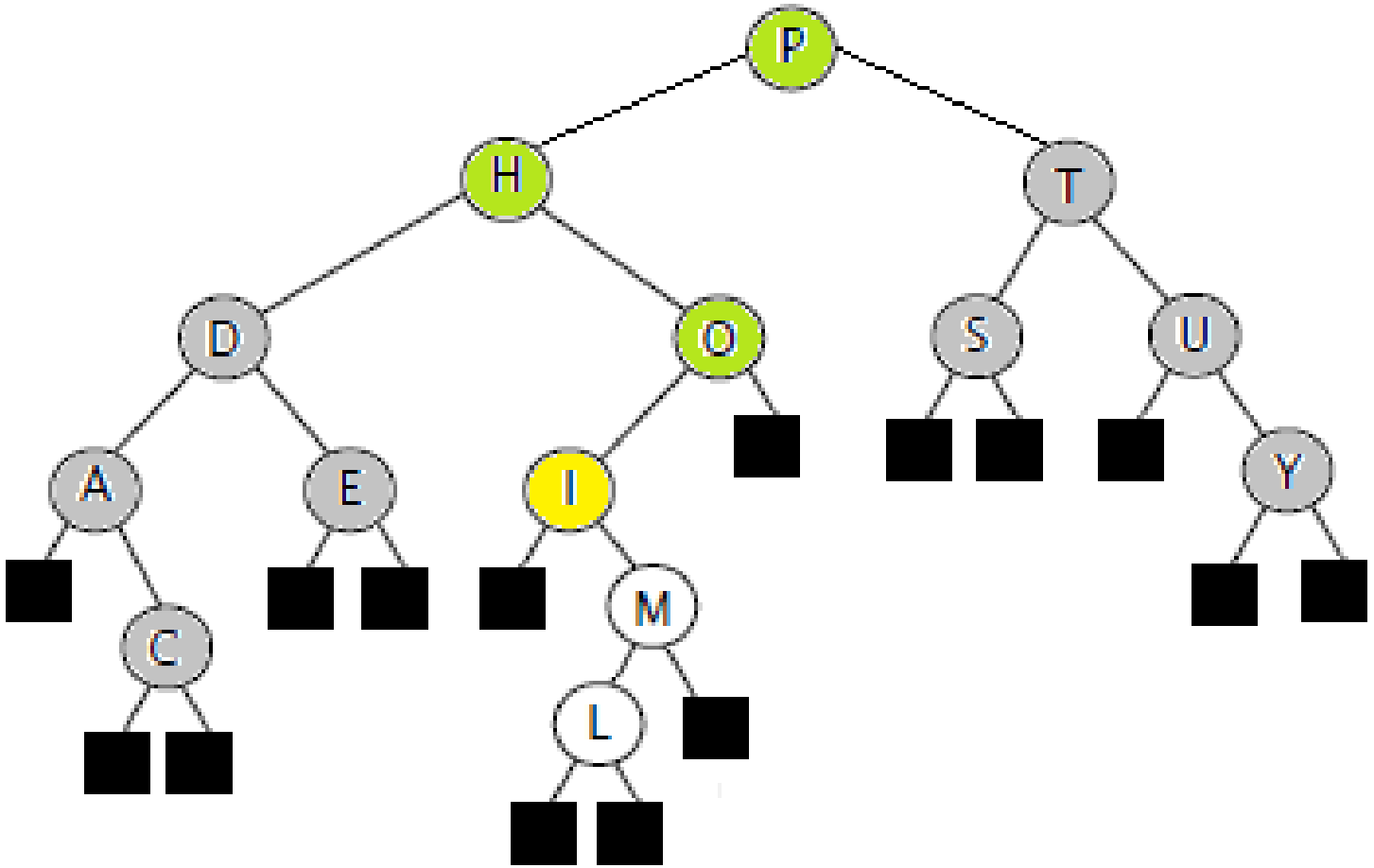
Busca nas ABBs -Exemplo: buscar **M**

- M $<$ O $\Rightarrow$ buscar à esquerda



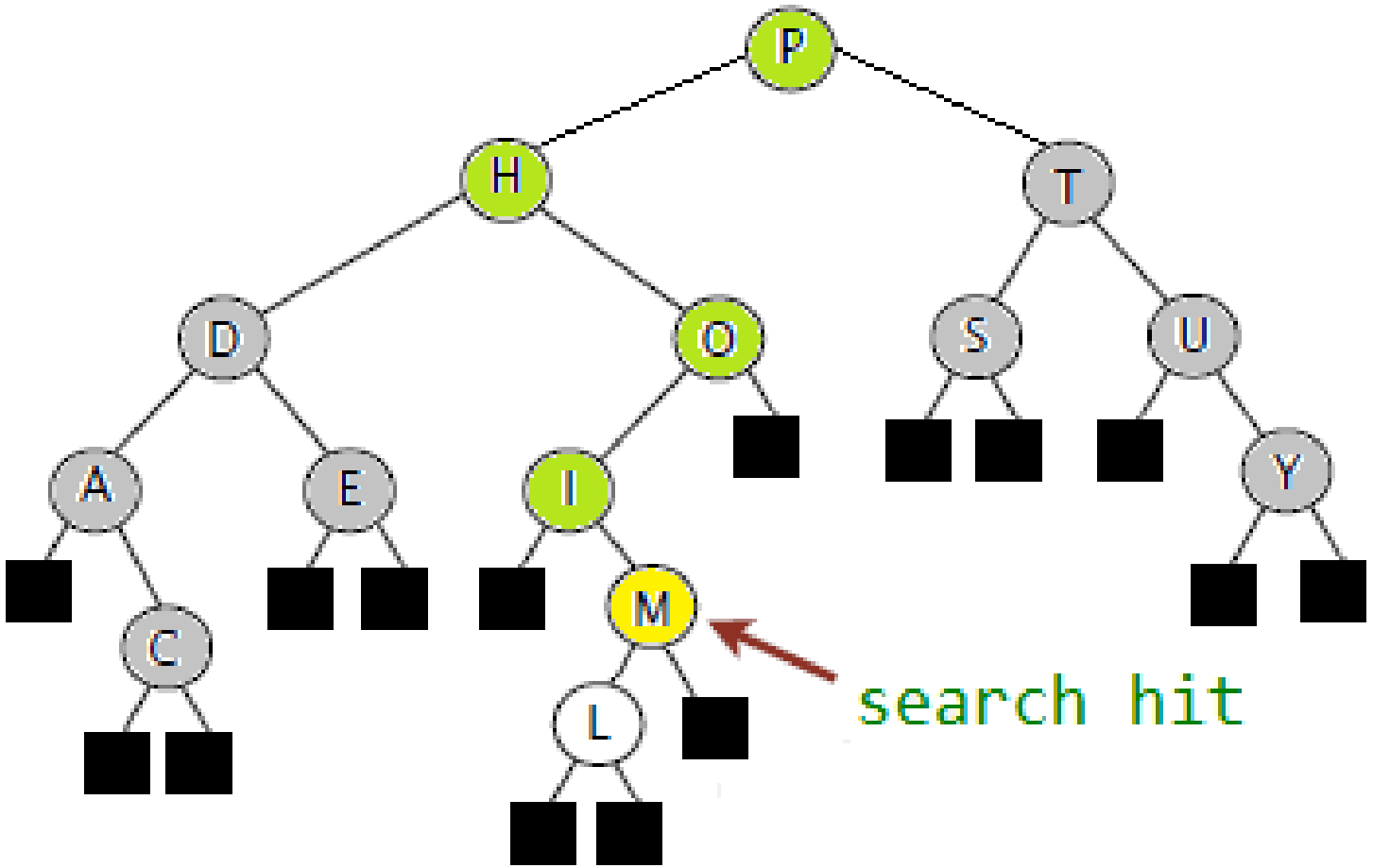
Busca nas ABBs -Exemplo: buscar **M**

- M $>$ I $\Rightarrow$ buscar à direita



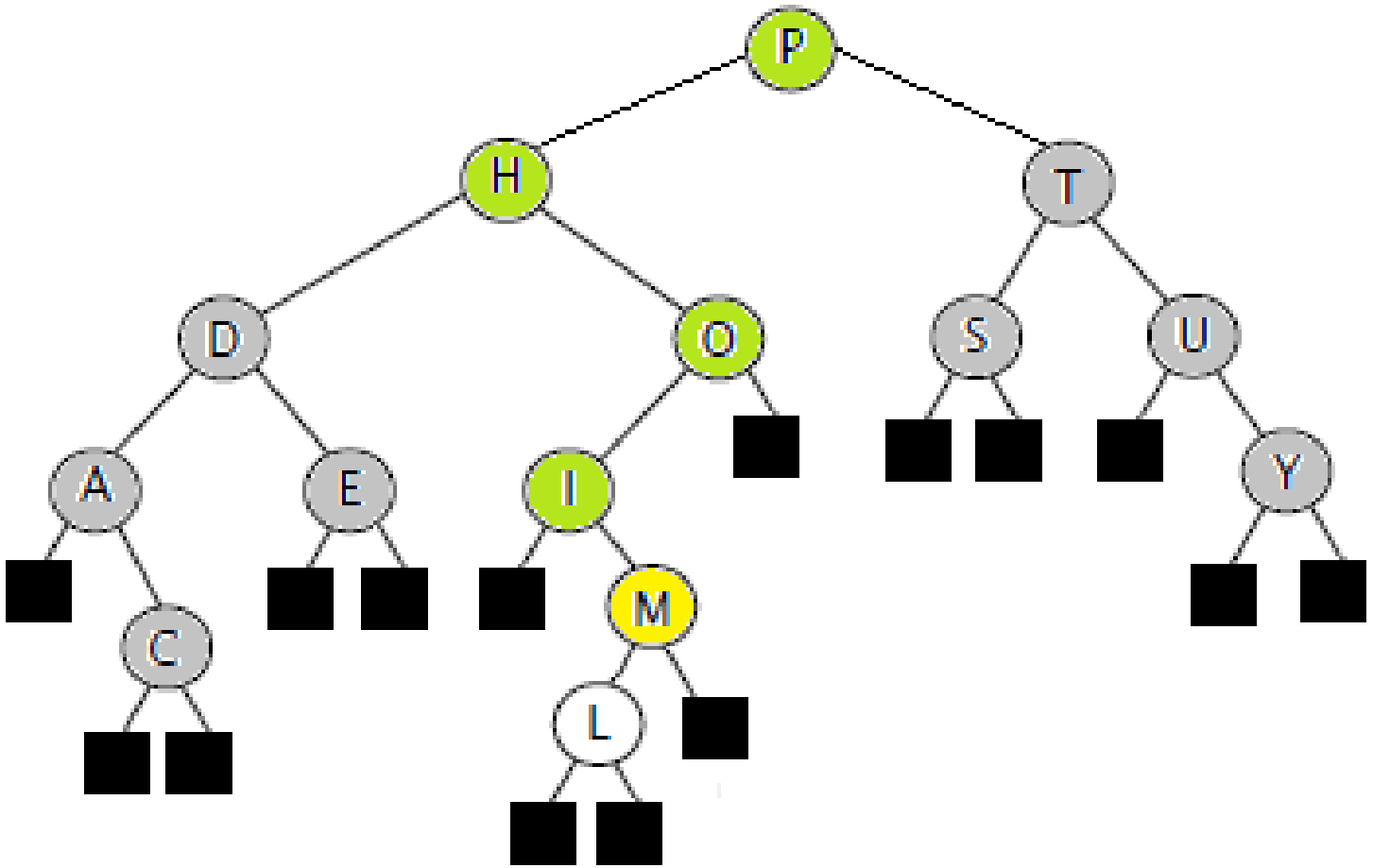
Busca nas ABBs -Exemplo: buscar **M**

- Chave encontrada!



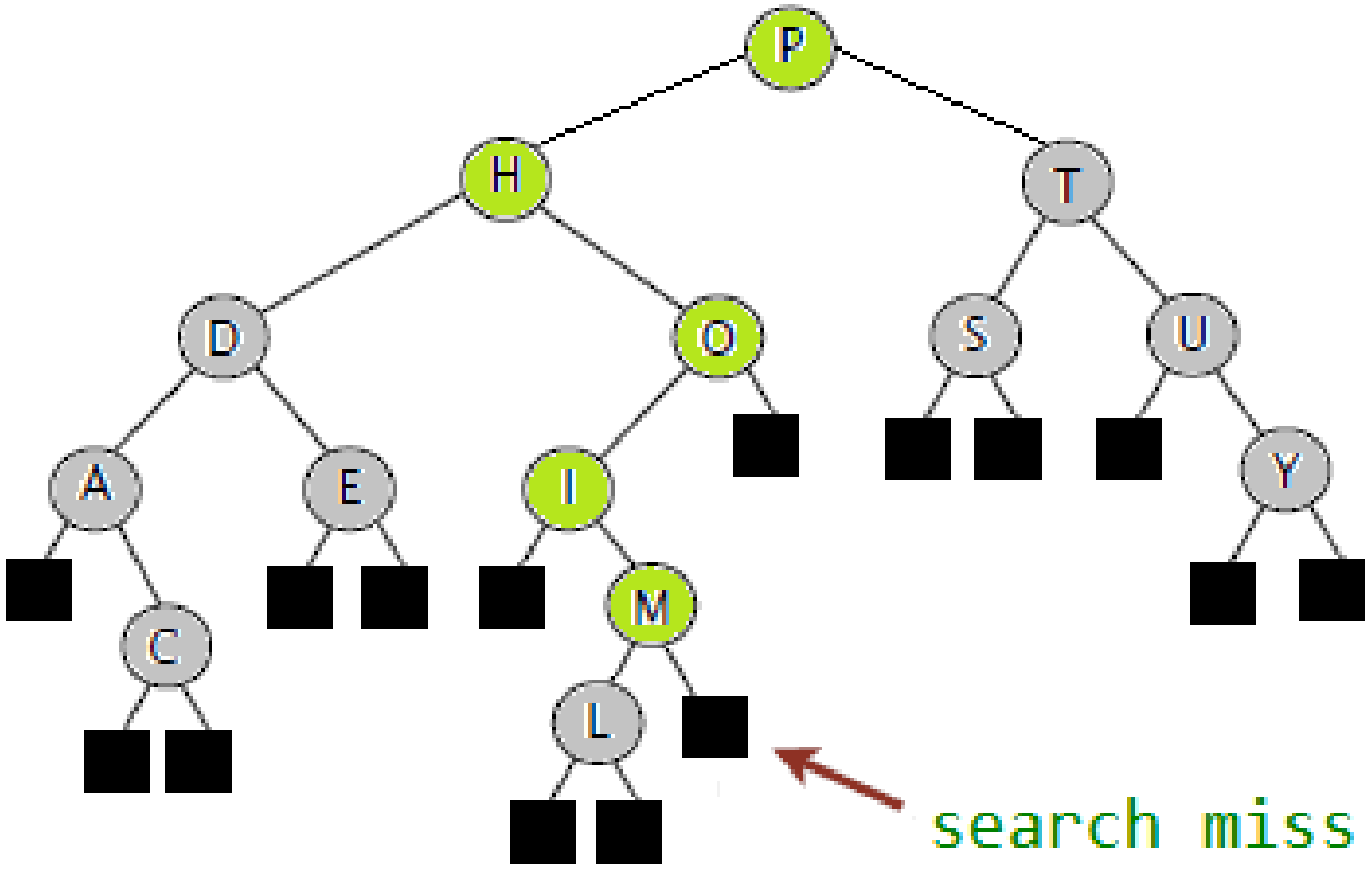
Busca nas ABBs -Exemplo: buscar **N**

- N $>$ M $\Rightarrow$ buscar à direita



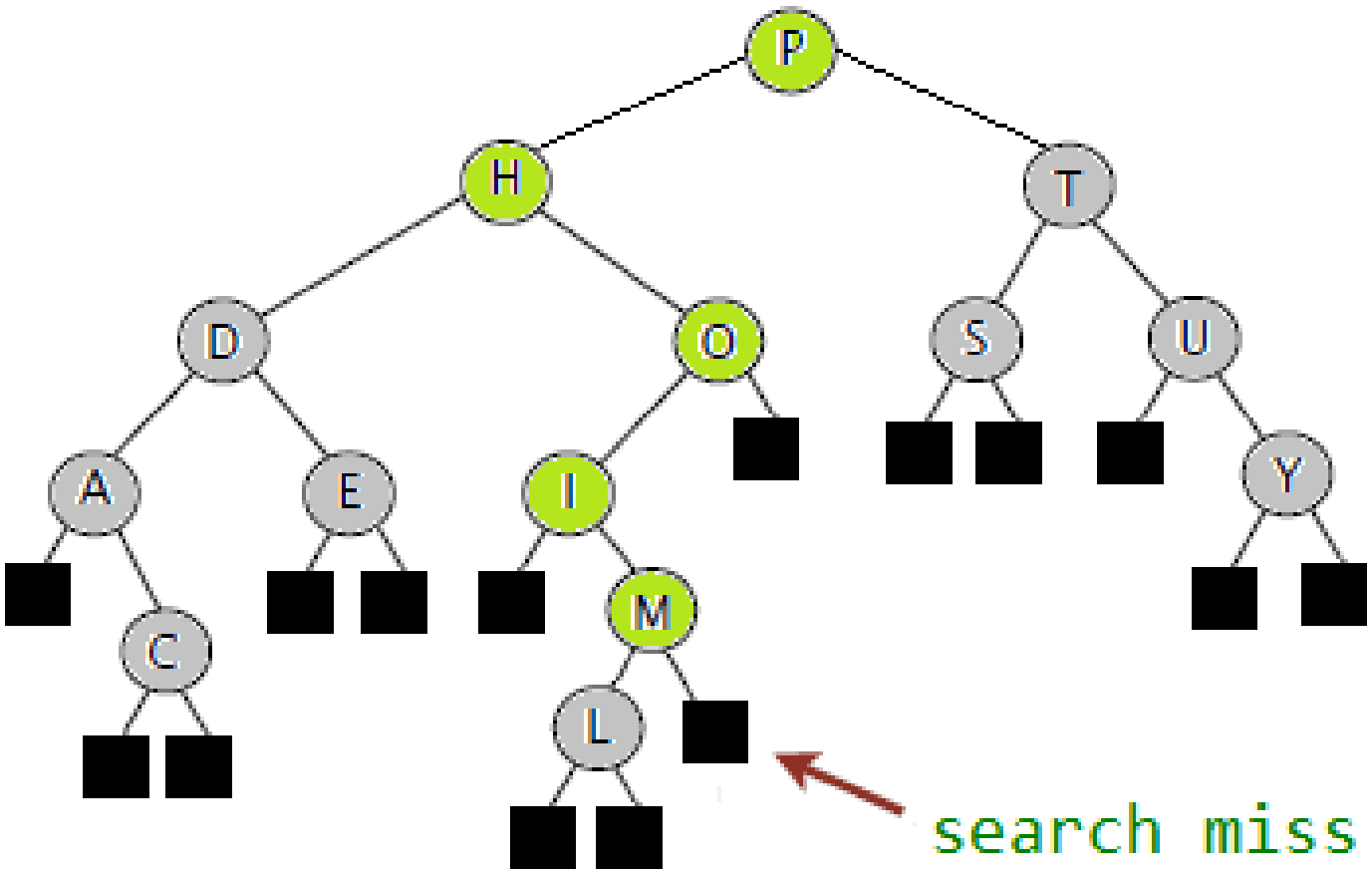
Busca nas ABBs -Exemplo: buscar **N**

- Chave não encontrada!



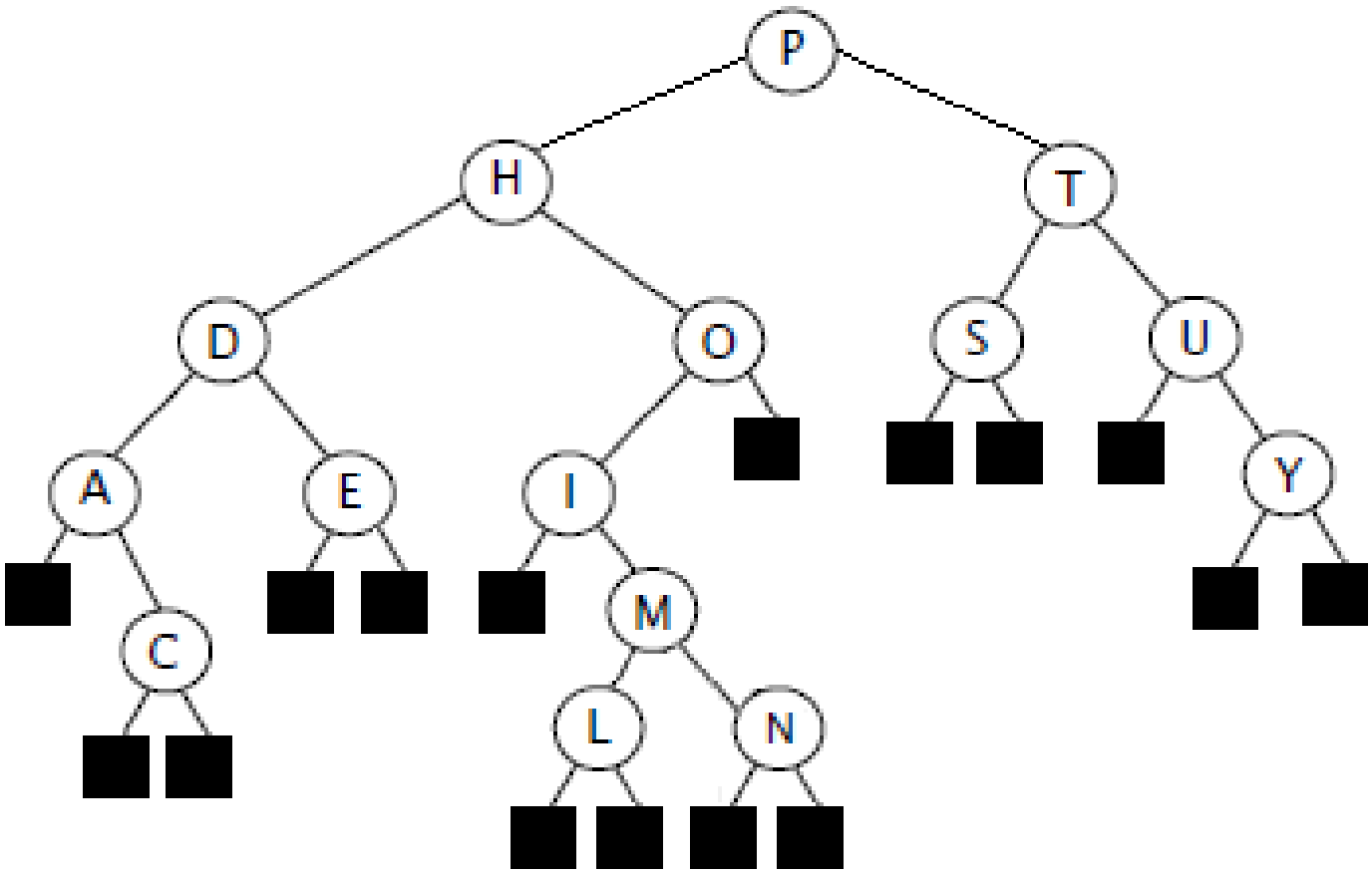
Inserção nas ABBs - Exemplo: inserir **N**

- A forma de uma ABB depende da ordem das inserções e remoções. O nova chave é sempre inserida numa folha



Inserção nas ABBs - Exemplo: inserir **N**

- A forma de uma ABB depende da ordem das inserções e remoções. O nova chave é sempre inserida numa folha



Inserção nas ABBs

- A forma de uma ABB depende da ordem das inserções e remoções. O nova chave é sempre inserida numa folha

Exercício: Desenhe a árvore que resulta de inserir os seguintes nós numa ABB: 15, 4, 20, 17, 19, 1, 8, 25, 12, 28, 2.

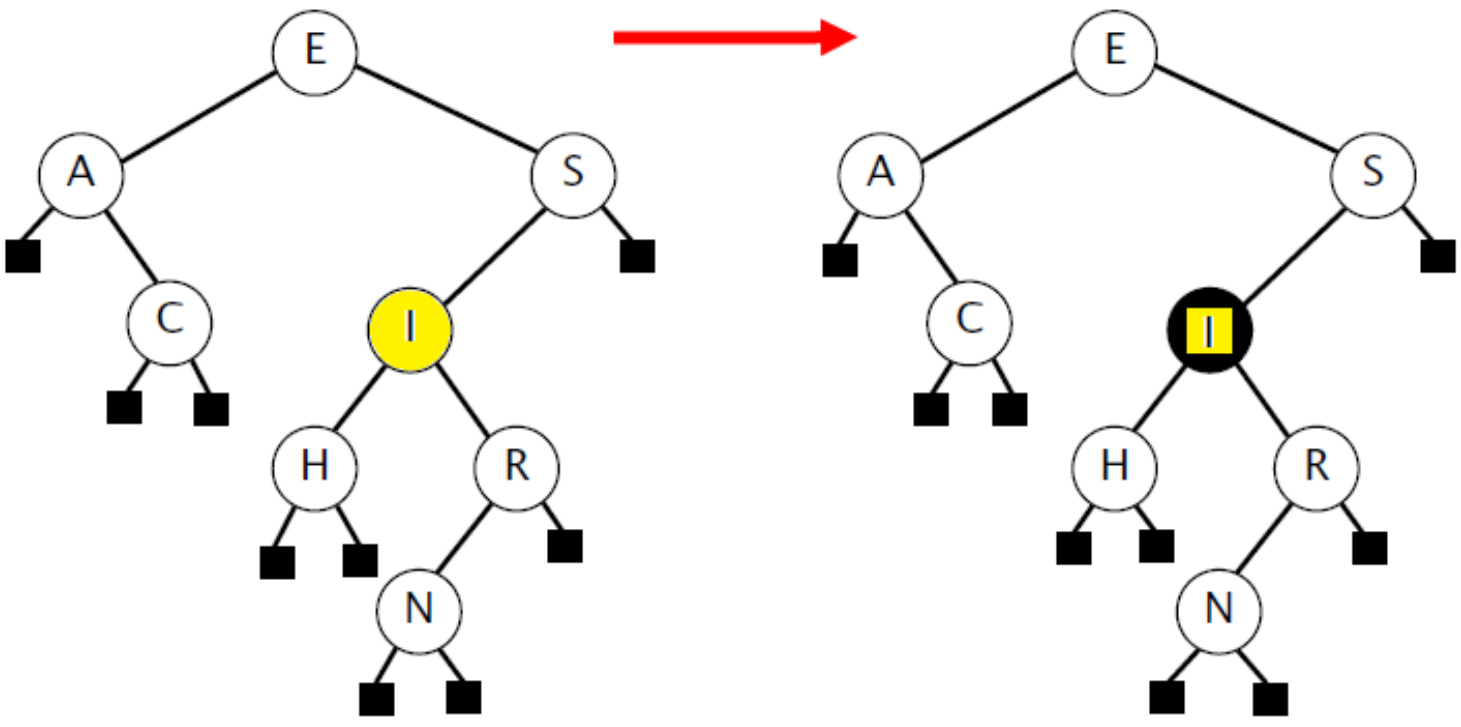
Binary Search Tree Visualization https://www.cs.usfca.edu/~galles/visualization/BST.html

Inserção nas ABBs - Variante recursiva

```
TreeNode *insert(TreeNode *t, Key key, Value value) {
  if (t == NULL) {
    t = malloc(sizeof(TreeNode));
    if (t != NULL) {
      t->key = key; t->value = value;
      t->pRight = t->pLeft = NULL;
    }
  }
  else if (key < t->key)
                t->pLeft = insert(t->pLeft, key, value);
        else if (key > t->key)
                t->pRight = insert(t->pRight, key, value);
              else
                t->value = value;
  return t;
}
```

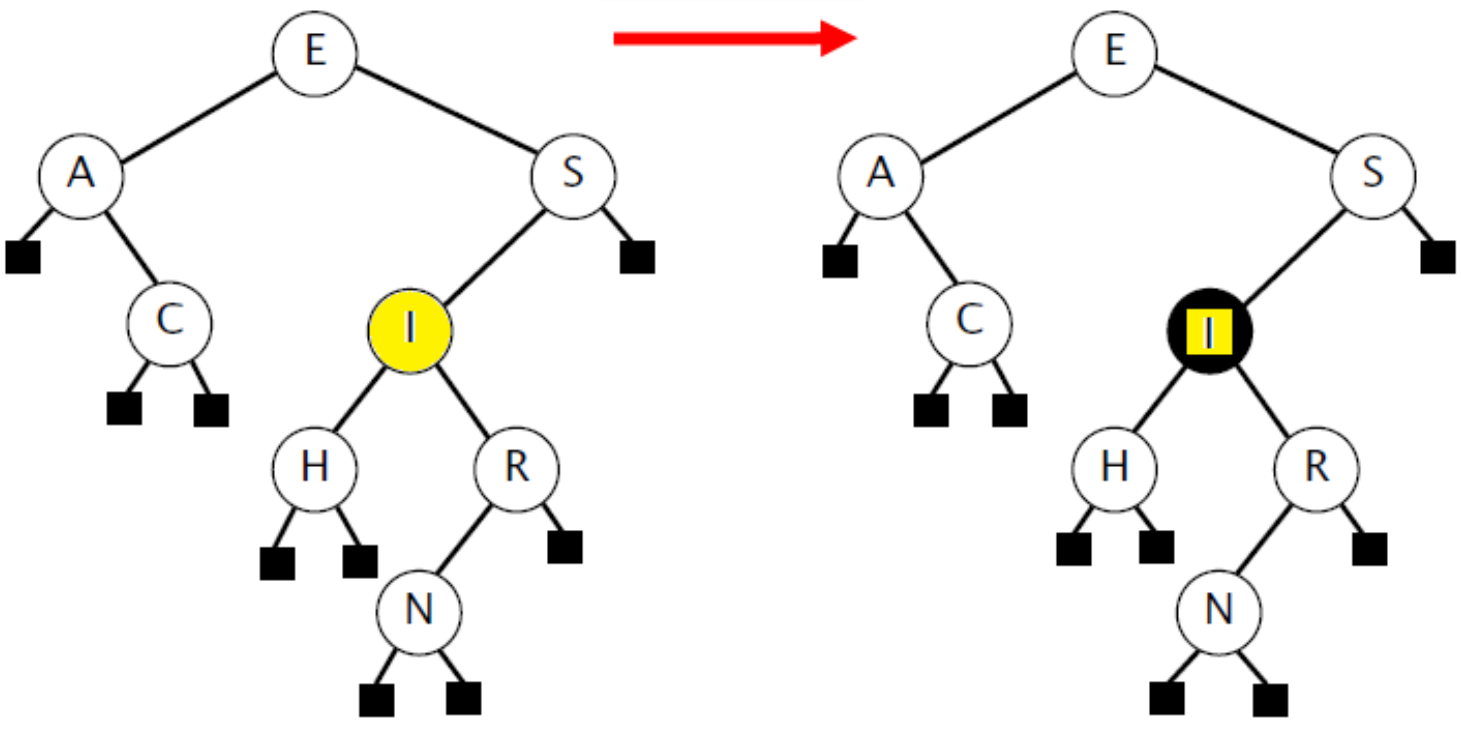
Remoção Preguiçosa nas ABBs

- Buscar o nó com a chave e atribuir `null` ao campo valor.
- Considerar a chave para guiar a busca; porém na igualdade retornar `null`



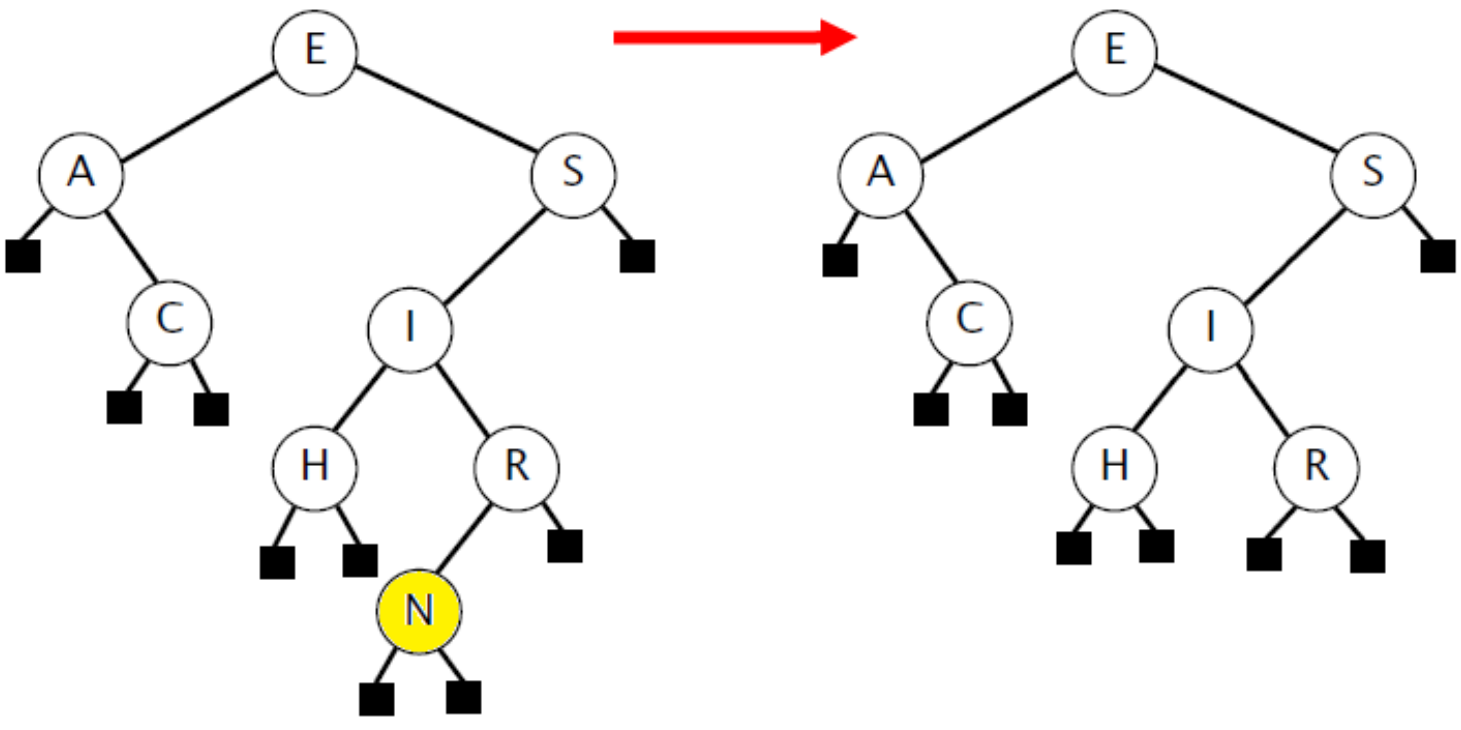
Remoção Preguiçosa nas ABBs

- Custo busca, inserção, remoção $O(\log N_t)$ onde N_t é o número de total chaves inseridas
- Desperdiço de memória; precisa de coleta de lixo



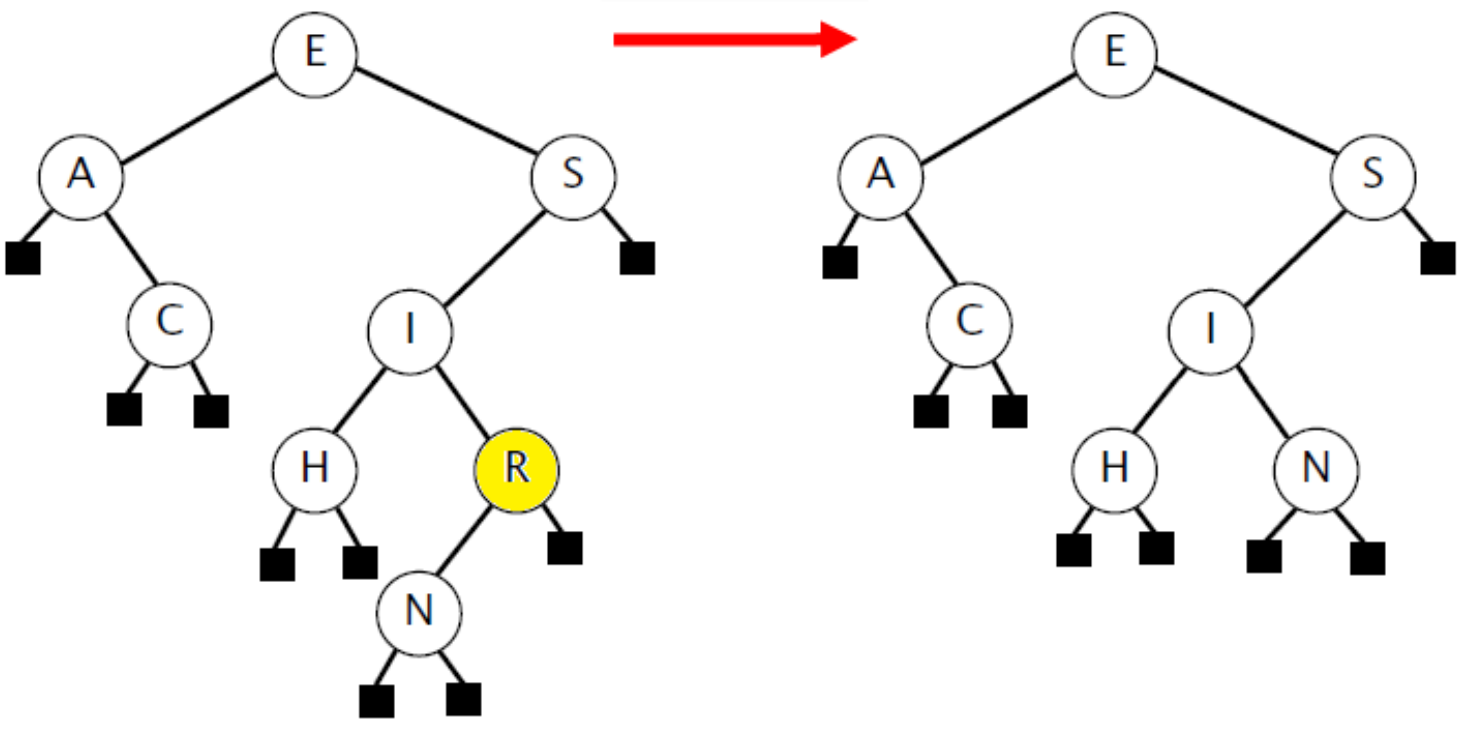
Remoção por cópia nas ABBs (T. Hibbard e D. Knuth)

Caso fácil: Se o nó a remover n tem zero ou um filho $f \Rightarrow$ Atualizar o link do pai do nó com `null` ou f resp.



Remoção por cópia nas ABBs (T. Hibbard e D. Knuth)

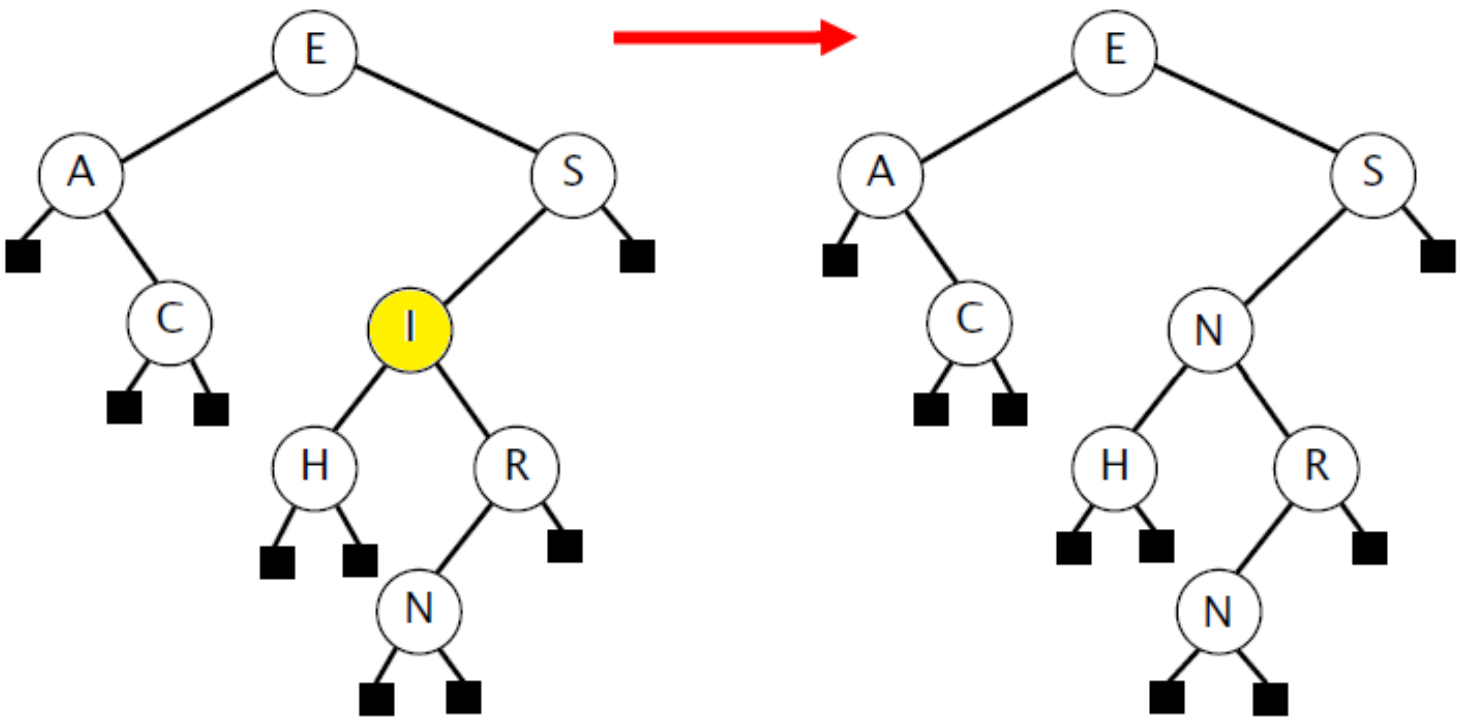
Caso fácil: Se o nó a remover n tem zero ou um filho $f \Rightarrow$ Atualizar o link do pai do nó com `null` ou f resp.



Remoção por cópia nas ABBs (T. Hibbard e D. Knuth)

Caso difícil: Se o nó a remover n tem dois filhos $\Rightarrow$ reduzir ao caso anterior

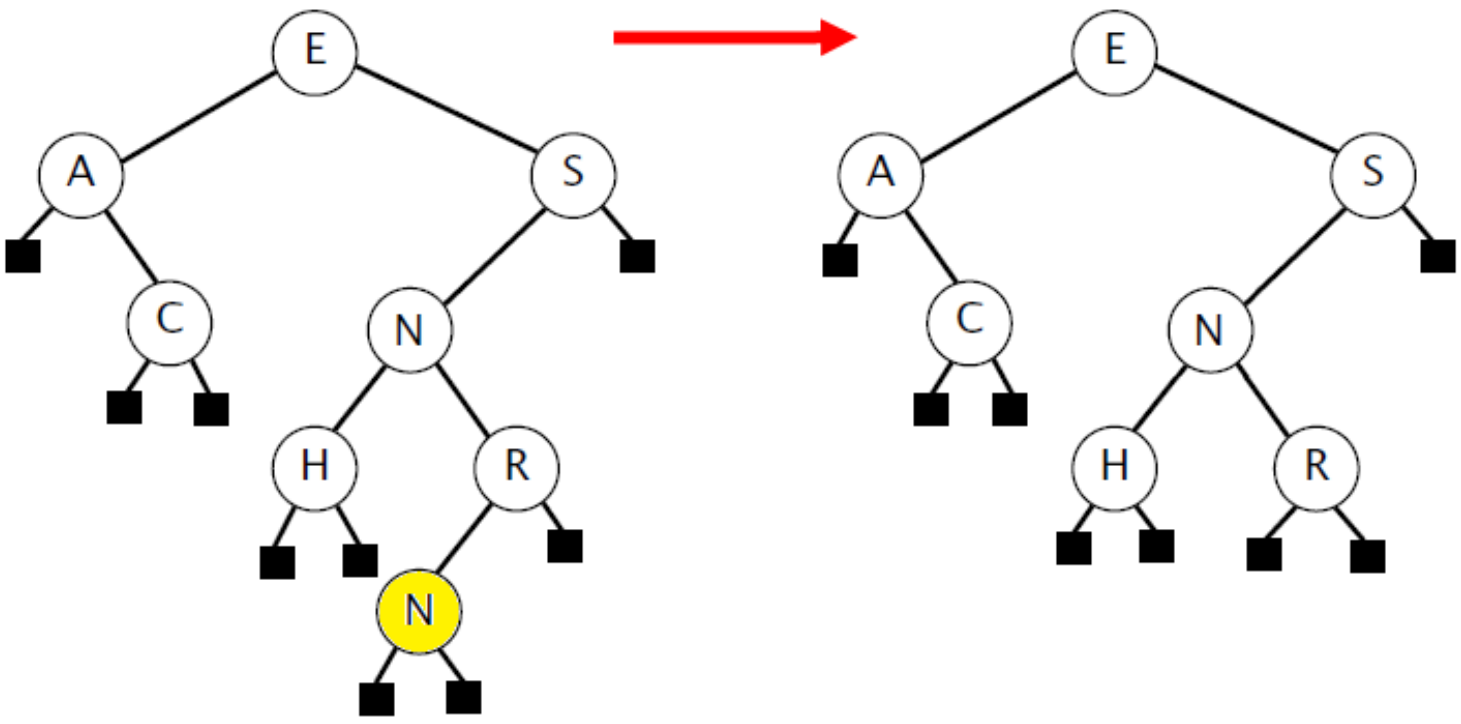
1. Colocar em n seu sucessor $succ(n) = s = min(n.right)$



Remoção por cópia nas ABBs (T. Hibbard e D. Knuth)

Caso difícil: Se o nó a remover n tem dois filhos $\Rightarrow$ reduzir ao caso anterior

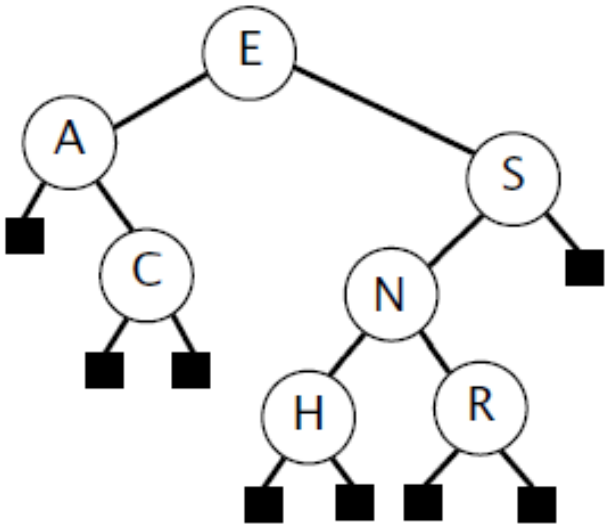
2. Remover s (cumpre que $s.left$==`null`)



Remoção por cópia nas ABBs (T. Hibbard e D. Knuth)

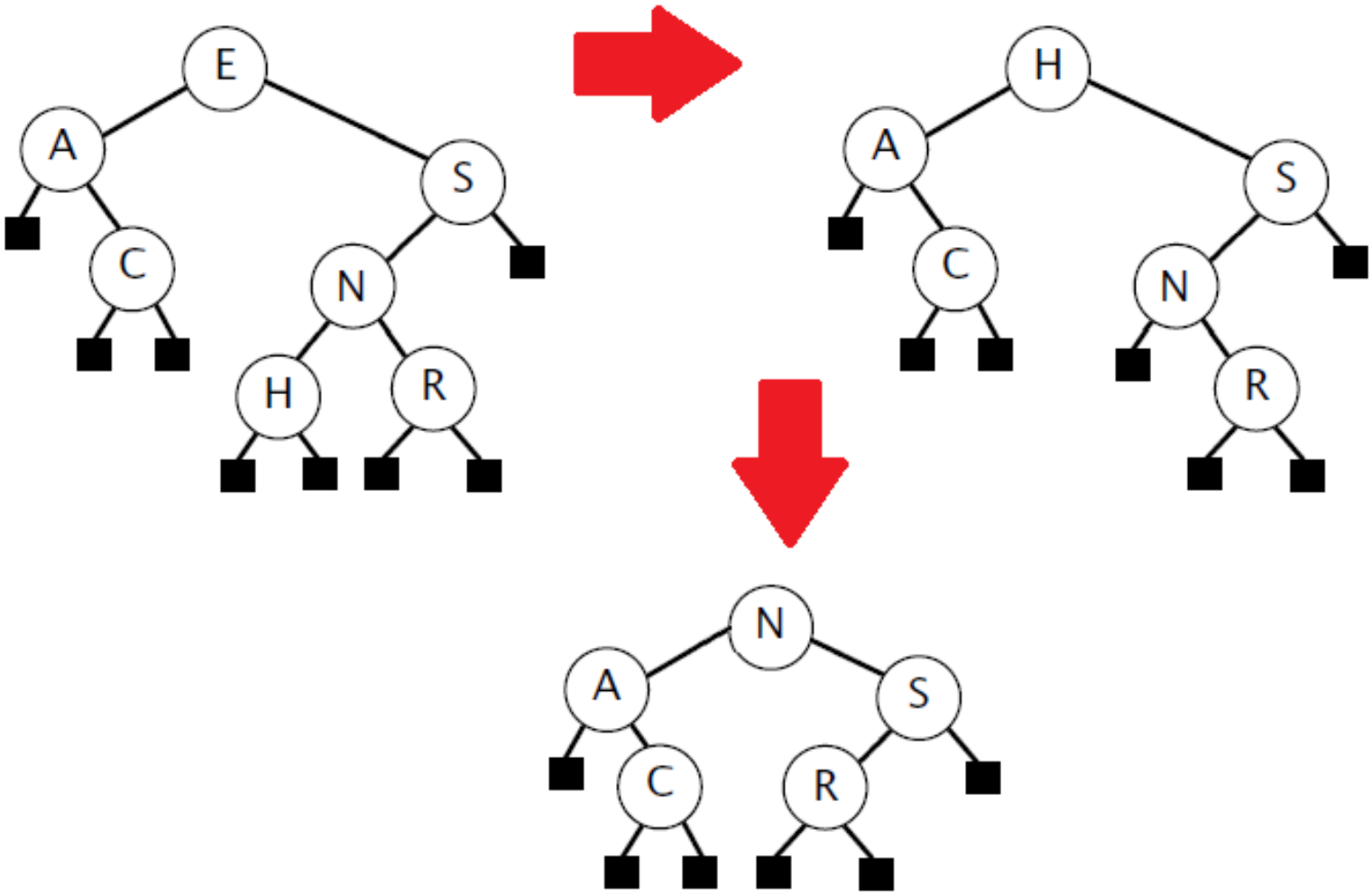
```
40  TreeNode* deleteNode(TreeNode* root, int key) {
41    if (root == NULL) return NULL;
42    if (key < root->key)                          // search in the left
43      root->left = deleteNode(root->left, key);
44    else if (key > root->key)                     // search in the right
45      root->right = deleteNode(root->right, key);
46    else {                                        // hit
47      TreeNode *tmp = root;
48      if (root->left == NULL || root->right == NULL) {    // easy case
49        root = root->left ? root->left : root->right;
50        free(tmp);
51      }
52      else {                                      // hard case
53        tmp = minValueNode(root->right);
54        root->key = tmp->key;                     // copy the successor
55        root->right = deleteNode(root->right, tmp->key);
56     }
57    }
58    return root;
59  }
```

Exemplo: Remover os nós E e H



https://www.cs.usfca.edu/~galles/visualization/BST.html

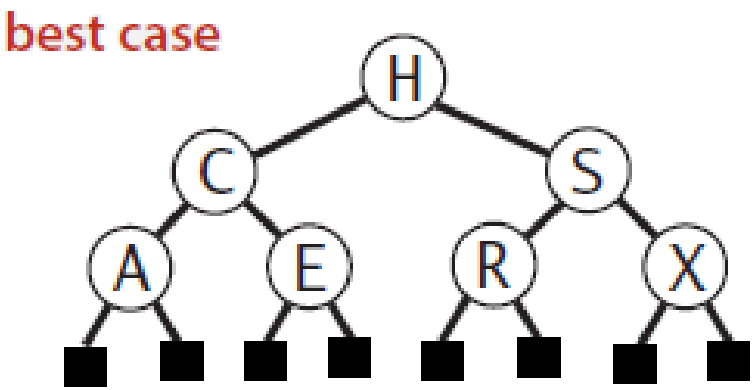
Exemplo: Remover os nós E e H



https://www.cs.usfca.edu/~galles/visualization/BST.html

Análise da Busca, Inserção e Remoção nas ABBs

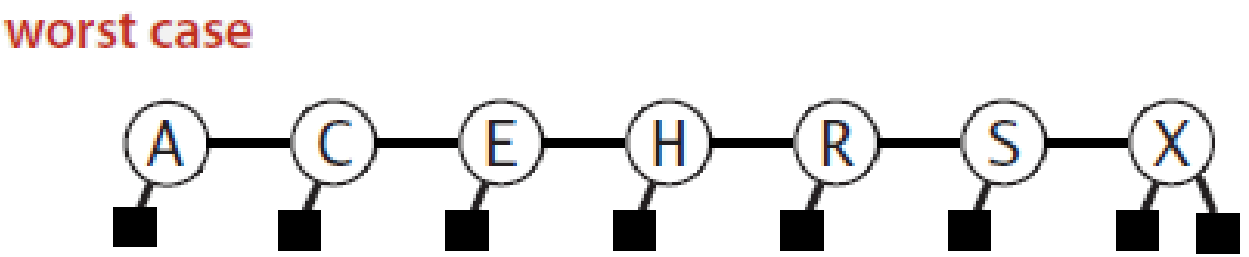
- A forma de uma ABB depende da ordem das inserções e remoções que sempre acontecem nas folhas
- Se N chaves distintas são inseridas numa ABB em ordem aleatória, o custo da busca é $\boldsymbol{O(\log N)}$



The Height of a Random Binary Search Tree, Bruce Reed, Journal of the ACM, v 50, No. 3, 2003, pp. 306-332.

Análise da Busca, Inserção e Remoção nas ABBs

- A forma de uma ABB depende da ordem das inserções e remoções que sempre acontecem nas folhas
- Se N chaves distintas são inseridas numa ABB em ordem aleatória, o custo da busca é $\boldsymbol{O(\log N)}$
- O caso pior é quando as chaves são inseridas ordenadas (crescente ou decrescente). Neste caso, árvore tem a forma duma lista linear; logo, o custo da busca é $\boldsymbol{O(N)}$



The Height of a Random Binary Search Tree, Bruce Reed, Journal of the ACM, v 50, No. 3, 2003, pp. 306-332.

Conclusões

- A forma de uma ABB depende da ordem das inserções e remoções que sempre acontecem nas folhas
- Se N chaves distintas são inseridas numa ABB em ordem aleatória, o custo da busca é $\boldsymbol{O(\log N)}$
- O caso pior é quando as chaves são inseridas ordenadas (crescente ou decrescente). Neste caso, árvore tem a forma duma lista linear; logo, o custo da busca é $\boldsymbol{O(N)}$

Técnica	Ordem	Busca	Inserção	Remoção
Busca Sequencial	Não	N	N	N
Busca Binária	Sim	log(N)	N	N
ABB	Sim	h	h	h
???	Sim	log(N)	log(N)	log(N)

The Height of a Random Binary Search Tree, Bruce Reed, Journal of the ACM, v 50, No. 3, 2003, pp. 306-332.

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Agenda

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Referências Bibliográficas

- Robert Sedgewick, Algorithms, 4th Edition, Addison-Wesley, 2011, Slides http://algs4.cs.princeton.edu/lectures/
- Introduction to Algorithms, 3rd Edition. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, 2009
- The Art of Computer Programming 3rd Edition,Donald Knuth, Section 6.1: Sequential Searching, 1997
- Projeto de Algoritmos, 2da Edição, Nivio Ziviani, 2007
- Requiem for a Bug - Verifying Software: Testing and Static Analysis, Johannes Kanig, Electronic Design (ED), 2014
- Requiem for a Bug - Verifying Software, Part 2: Formal Verification through SPARK 2014, Johannes Kanig, ED, 2015

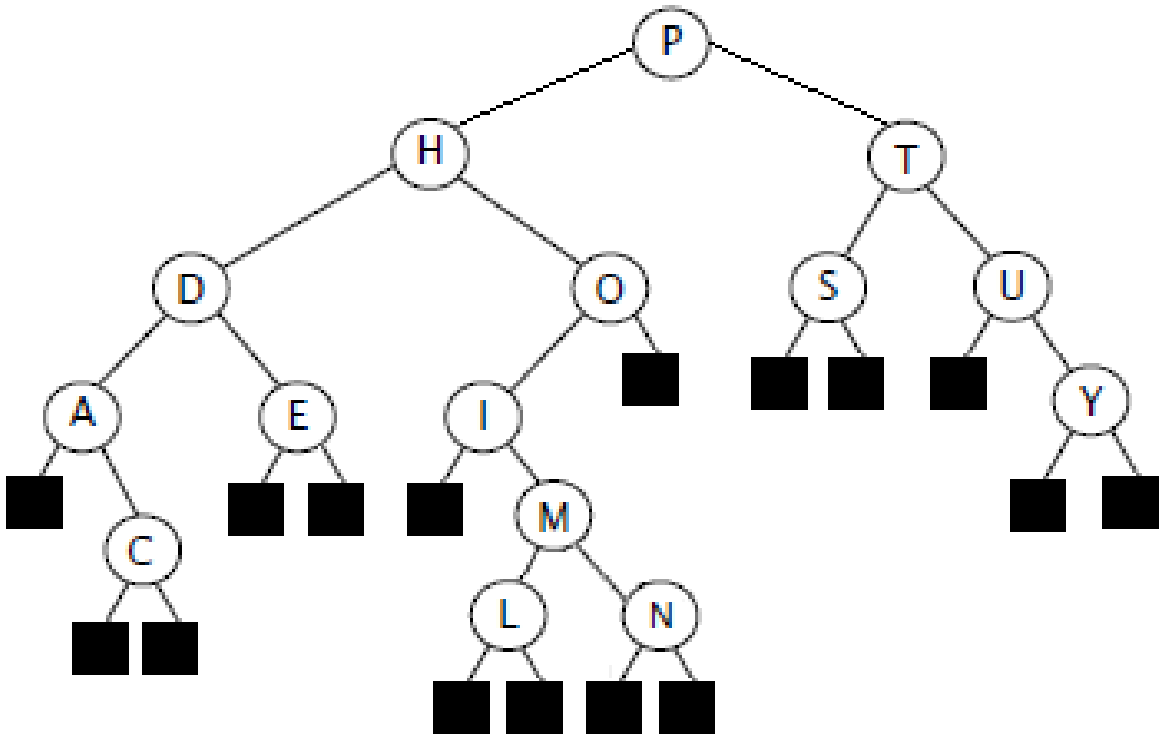
Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Agenda

Exercício 1- Remoção por cópia usando o antecessor

Quais modificações na remoção de T. Hibbard e D. Knuth devem ser feitas para usar o **antecessor** em lugar do sucessor? Mostre a ABB resultante de remover os nós D, H e P (nessa ordem) da seguinte ABB usando o antecessor.



Exercício 2- Remoção por fusão

Na remoção por **fusão**, para um nó com duas sub-árvores, uma das duas subárvores do nó é extraída e anexada à outra subárvore. Escreva uma função que implemente uma remoção por fusão.