Aula 15
Coleções “Informadas”
Tabelas de Símbolos e Árvores de Pequisa Binária

Algoritmos e Estruturas de Dados

Tabelas de Símbolos

Tabelas de Símbolos

- Caso particular de coleção informada
- Em que guardamos sempre um par de objetos
  - *<chave,valor>*
- Usamos a informação da chave para guardar o par
- Permite pesquisar por um objeto através de uma chave de forma muito eficiente

Observação:
Tabelas de símbolos são também designadas de Dicionários, ou Mapas

Tabelas de Símbolos

- API Genérica Tabela de Símbolos

public interface SymbolTable<Key, Value>		
`void`	`put(Key k, Value v)`	coloca o par <k,v> na tabela. O valor v não pode ser *null*. **Não existem chaves duplicadas na tabela**. Se for inserido um par<k,v> para uma chave k já existente na tabela, o novo valor v irá substituir o valor antigo.
`Value`	`get(Key k)`	retorna o valor associado à chave recebida. Não remove o valor da tabela. Se a chave não existir na tabela deverá retornar *null*
`void`	`delete(Key k)`	remove a chave e o seu valor da tabela
`boolean`	`contains(Key k)`	retorna *true* se a chave existir na tabela e *false* caso contrário
`boolean`	`isEmpty()`	retorna *true* se a tabela estiver vazia, e *false* caso contrário
`int`	`size()`	retorna o número de pares <chave,valor> guardados na tabela
`Iterable<Key>`	`keys()`	devolve um iterador para todas as chaves da tabela

Tabela de Símbolos

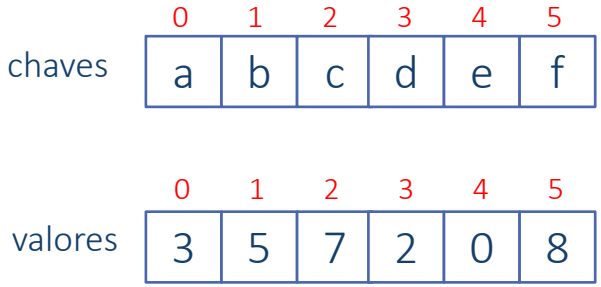
Como *array* ordenado

Implementação como array ordenado

- A implementação é praticamente igual à implementação de um array ordenado, mas:
  - Trabalhamos com 2 arrays alinhados
    - *1 array ordenado pela chave*
    - *1 array com os valores a guardar em cada índice correspondente*
  - Não podemos colocar 2 vezes a mesma chave no array ordenado

Implementação como array ordenado

- Chaves e valores guardados em 2 arrays paralelos
- Arrays ordenados pela chave
  - Chaves têm que ser comparáveis
- Ex:
  - put <a,3>, put <b,5>, put <c,7>, put <d,2>, put <e,0>, put<f,8>



Método put

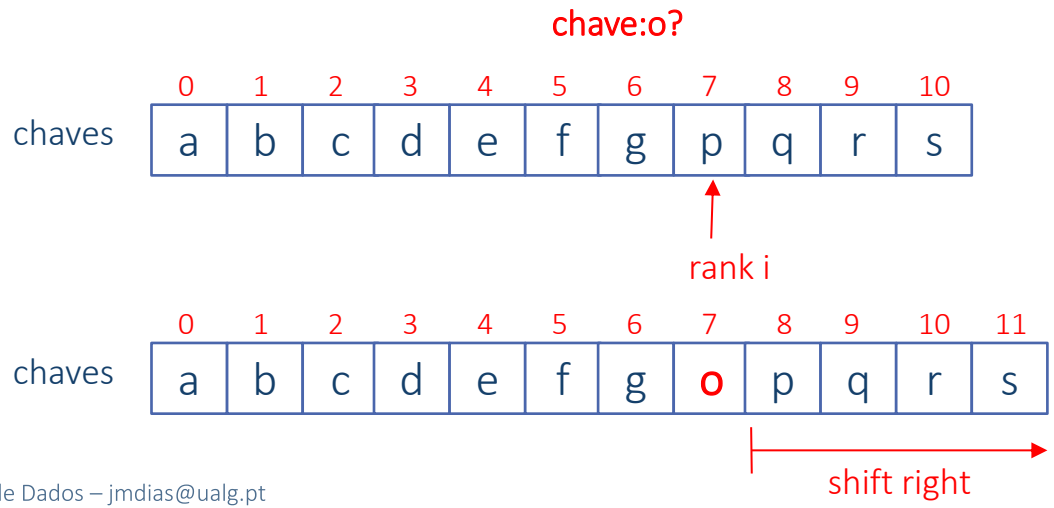
- Inserir um par <chave,valor>
  - Usar o método rank para determinar a posição da chave
  - Se a chave já lá estiver

    Alterar o valor associado à chave para valor
  - Se a chave não estiver

    Temos que a adicionar

    Mover tudo o resto para a direita

    Incluindo o array de valores



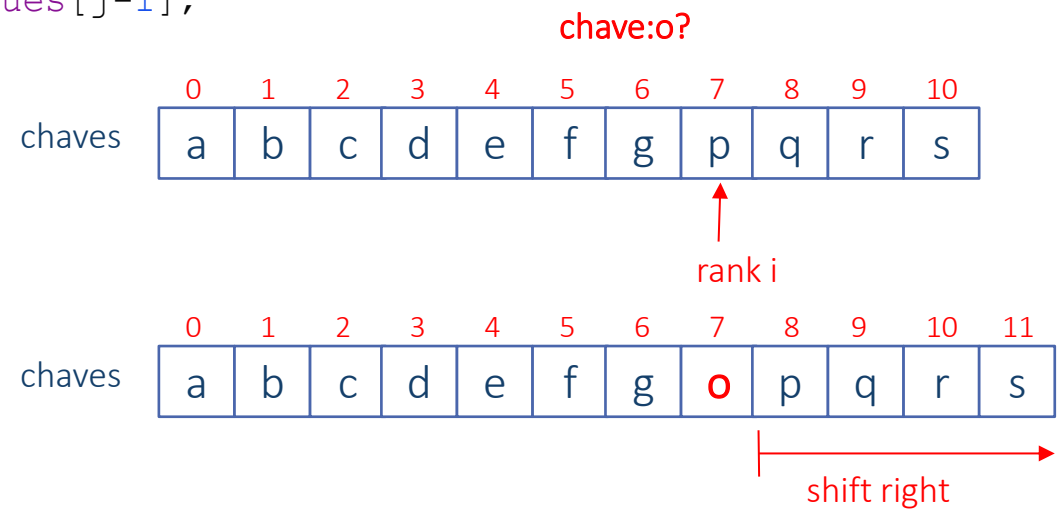
Método put

```
public void put(Key k, Value v)
{
    int i = rank(k);
    if(i < this.size && k.compareTo(this.keys[i])==0)
    {
        this.values[i] = v;
        return;
    }

    for(int j = this.size; j > i; j--)
    {
        this.keys[j] = this.keys[j-1];
        this.values[j] = this.values[j-1];
    }

    this.keys[i] = k;
    this.values[i] = v;
    this.size++;

    return;
}
```

O método rank devolve-nos a posição onde a chave deve ser colocada, para que o array de chaves continue ordenado

Já existe uma chave igual à que estamos a tentar colocar, portanto apenas temos de substituir o valor associado à chave

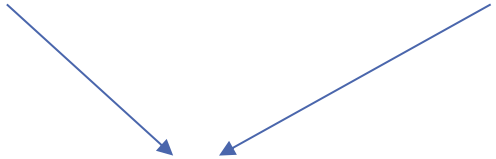
shift right



Complexidade Temporal

- Tem os mesmos problemas que um array ordenado
- Inserção não é boa por causa da operação de shift

	Caso Médio		Pior Caso	
	Inserção	Pesquisa	Inserção	Pesquisa
BinarySearch Array	n	$\log_2 n$	2n	$\log_2 n$



A inserção não é tão simpática por causa dos shifts

Árvores de Pesquisa

Binária

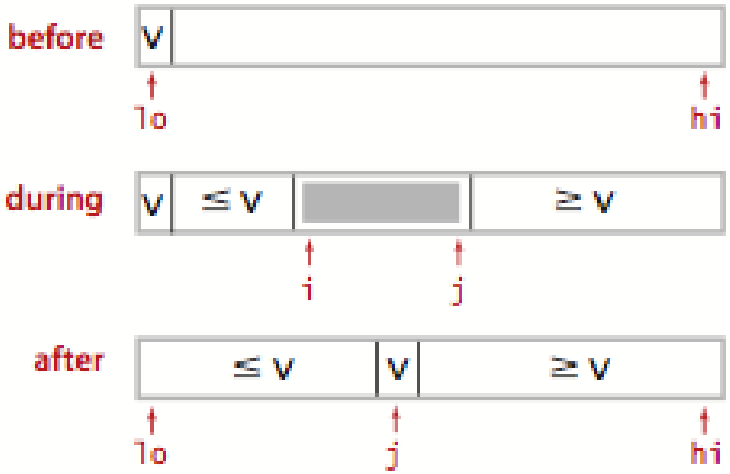
Pesquisa Binária com Arrays

- O algoritmo de pesquisa binária com array
  - Performance interessante
- No entanto,
  - Problema com inserção devido à utilização de arrays
  - (shifts fazem com que complexidade seja O(n))

- Ideia:
  - Tirar partido da flexibilidade das listas ligadas
  - Tirar partido da pesquisa binária
  - Tirar partido da ideia de partition do quicksort

Pesquisa Binária com Árvores

- Ideia:
  - Tirar partido da flexibilidade das listas ligadas/árvores
  - Tirar partido da pesquisa binária
  - Tirar partido da ideia de partition do quicksort



Quicksort partitioning overview

Árvore de Pesquisa Binária

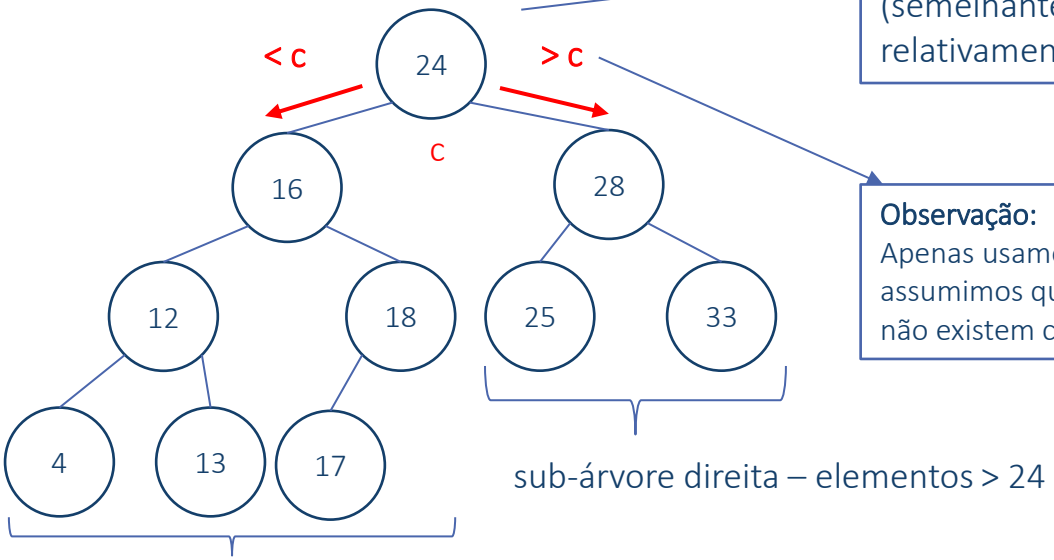
- *Def:*
  - *Árvore binária onde:*

    cada nó contem uma chave, ou um par <chave,valor>

    a chave de um nó é sempre:

    Maior que as chaves da subárvore esquerda

    Menor que as chaves da subárvore direita



Observação:
O pai funciona como um pivô (semelhante ao pivô do quicksort) relativamente aos filhos

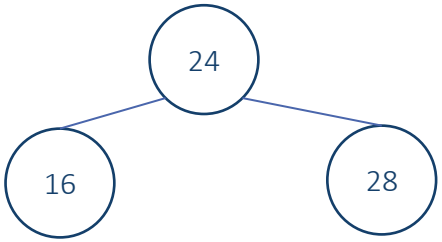
Observação:
Apenas usamos > em vez de >= porque assumimos que numa tabela de símbolos não existem chaves repetidos.

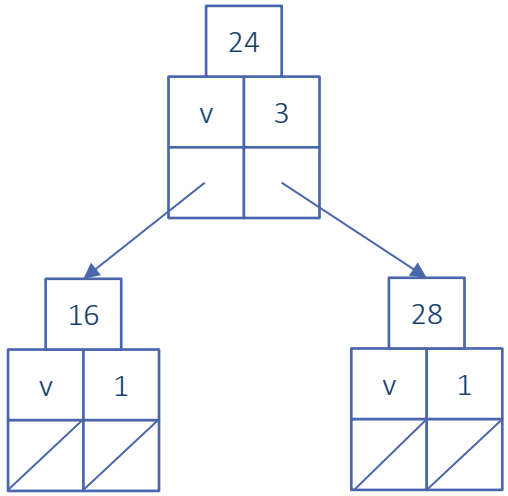
Nó de uma árvore de pesquisa binária

```
private class Node
{
    private Key key;
    private Value value;
    private Node left;
    private Node right;
    private int size;

    public Node(Key k, Value v, int size)
    {
        this.key = k;
        this.value = v;
        this.size = size;
    }
}
```

Esta classe não é genérica pq está definida internamente dentro da classe BinarySearchTree (que é genérica).

Se quisermos definir esta classe fora, tem que ser uma classe genérica <Key,Value>

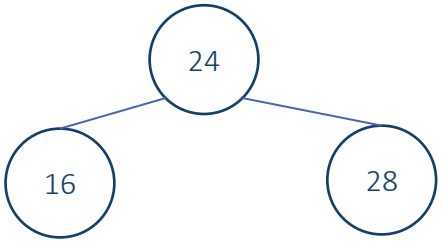


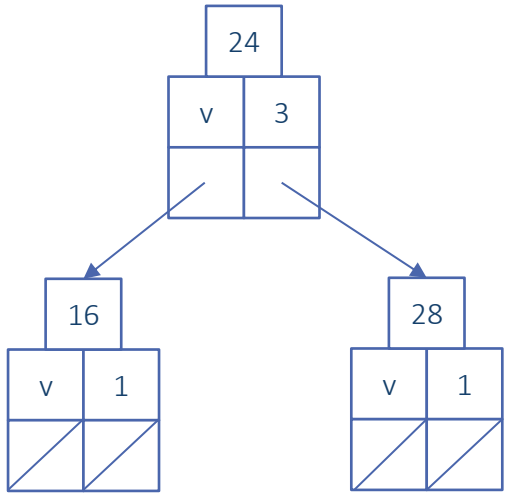


representação por caixas e ponteiros

Árvore de pesquisa binária

```
public class BinarySearchTree<Key extends Comparable<Key>, Value> {

    private Node root;

    public BinarySearchTree()
    {
        this.root = null;
    }

    private int size(Node n)
    {
        if(n==null) return 0;
        else return n.size;
    }

    public int size()
    {
        return size(this.root);
    }
    …
}
```





representação por caixas e ponteiros

Método *get(Key k)*

- Parecido ao *get* para pesquisa binária em arrays
  - Mas em vez de calcularmos o mid do array…

  - Começar pelo correspondente à raiz da árvore
  - Se chave nó = k

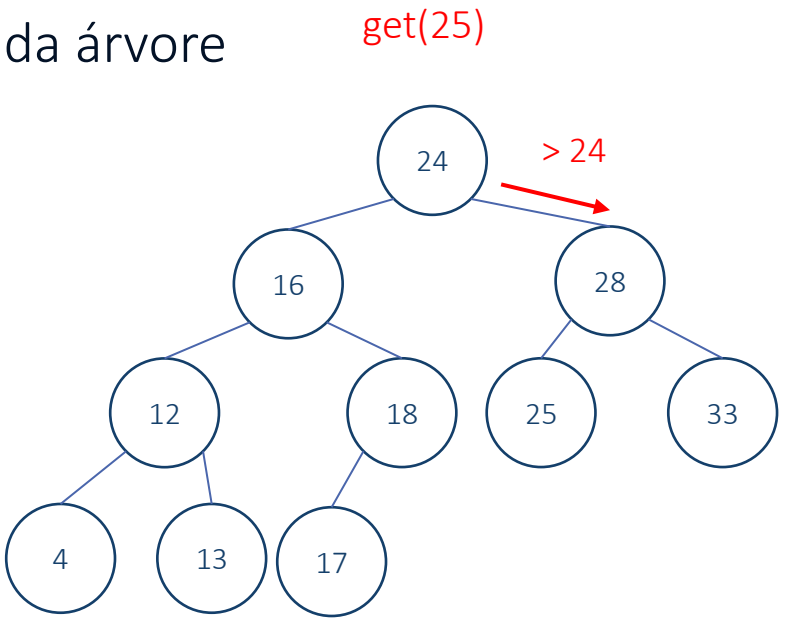
    Retornamos o valor guardado no nó
  - Se k < chave do nó

    Procuramos na subárvore esquerda
  - Se k > chave do nó

    Procuramos na subárvore direita

  - Se chegarmos a uma árvore vazia

    Não encontrámos a chave, retornar null



Método *get(Key k)*

- Parecido ao *get* para pesquisa binária em arrays
  - Mas em vez de calcularmos o mid do array…

  - Começar pelo correspondente à raiz da árvore
  - Se chave nó = k

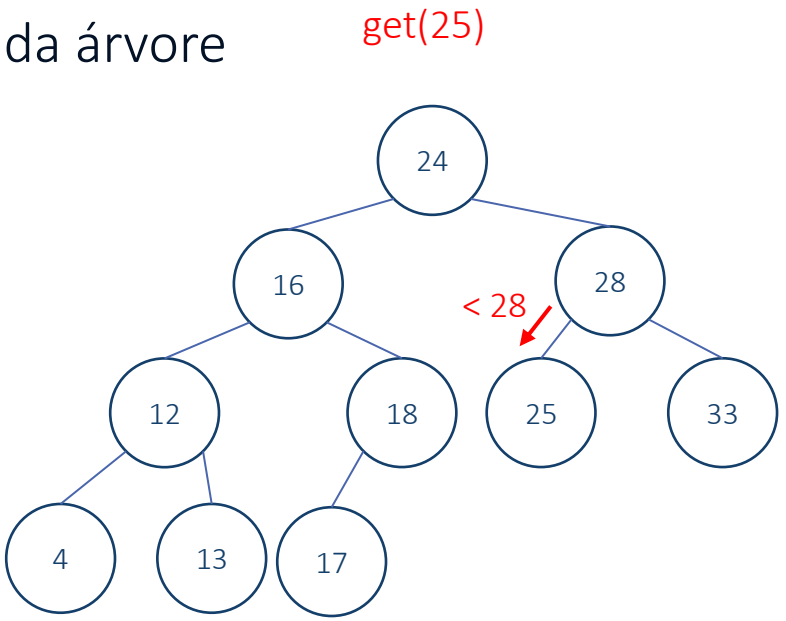
    Retornamos o valor guardado no nó
  - Se k < chave do nó

    Procuramos na subárvore esquerda
  - Se k > chave do nó

    Procuramos na subárvore direita

  - Se chegarmos a uma árvore vazia

    Não encontrámos a chave, retornar null



Método *get(Key k)*

- Parecido ao *get* para pesquisa binária em arrays
  - Mas em vez de calcularmos o mid do array…

  - Começar pelo correspondente à raiz da árvore
  - Se chave nó = k

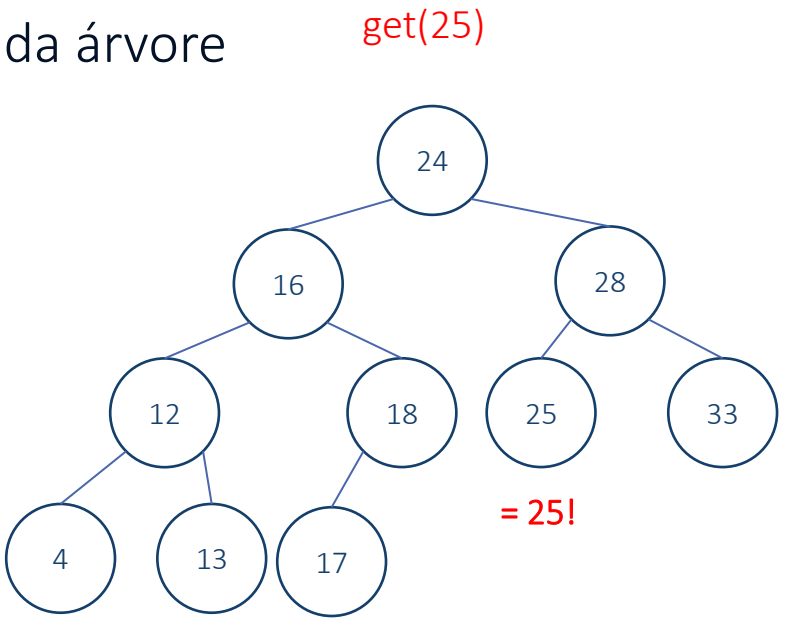
    Retornamos o valor guardado no nó
  - Se k < chave do nó

    Procuramos na subárvore esquerda
  - Se k > chave do nó

    Procuramos na subárvore direita

  - Se chegarmos a uma árvore vazia

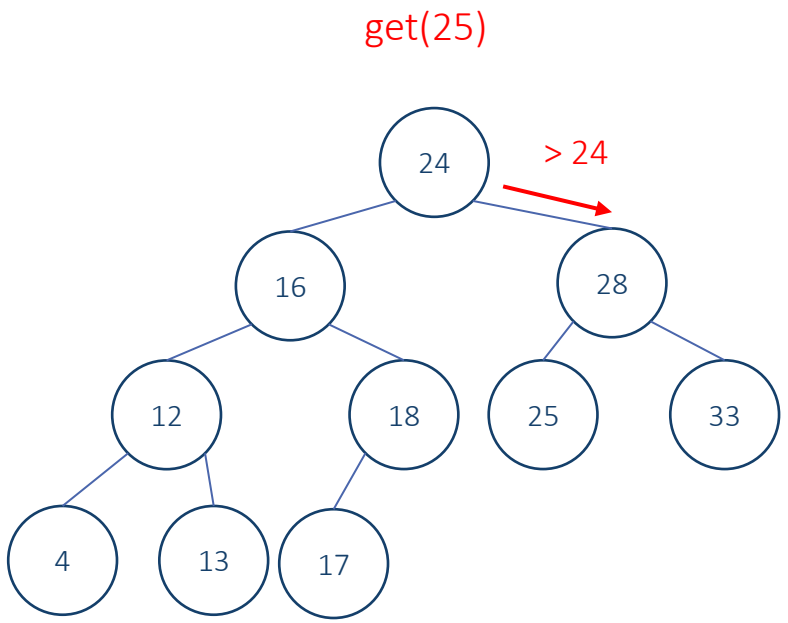
    Não encontrámos a chave, retornar null



Método *get(Key k)*

Observação: é possível implementar este método sem usar recursão, mas reparem que as árvores são estruturas naturalmente recursivas, e portanto é fácil implementar métodos recursivos sobre elas.

```
public Value get(Key k)
{
    return get(this.root, k);
}

private Value get(Node n, Key k)
{
    if(n == null) return null;
    int cmp = k.compareTo(n.key);
    if (cmp < 0) return get(n.left,k);
    else if(cmp > 0) return get(n.right,k);
    else return n.value;
}
```



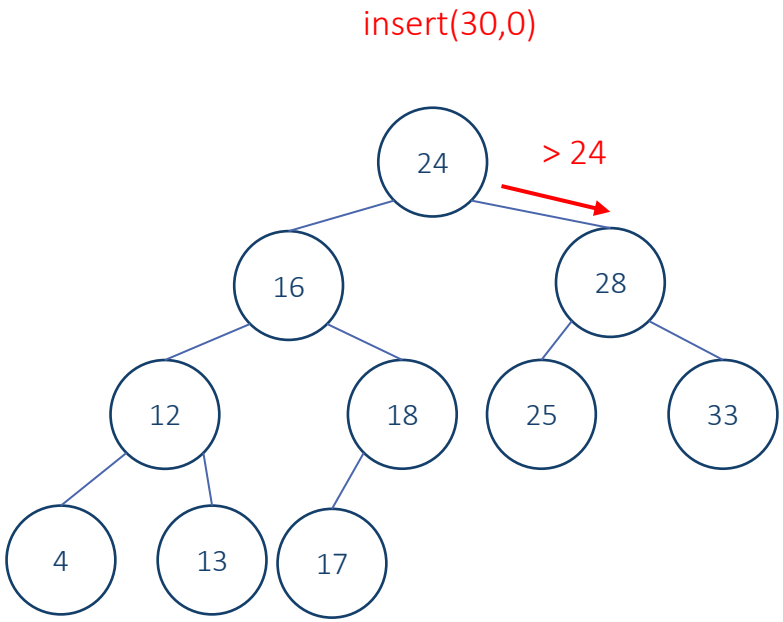
Método *put(Key k, Value v)*

- Começar pelo correspondente à raiz da árvore
- Se chave nó = k

  Alterar o valor guardado no nó para v
- Se k < chave do nó

  Colocar à esquerda
- Se k > chave do nó

  Colocar à direita



- Se chegarmos a uma árvore vazia

  Criar e colocar um novo nó com o par <key,valor>

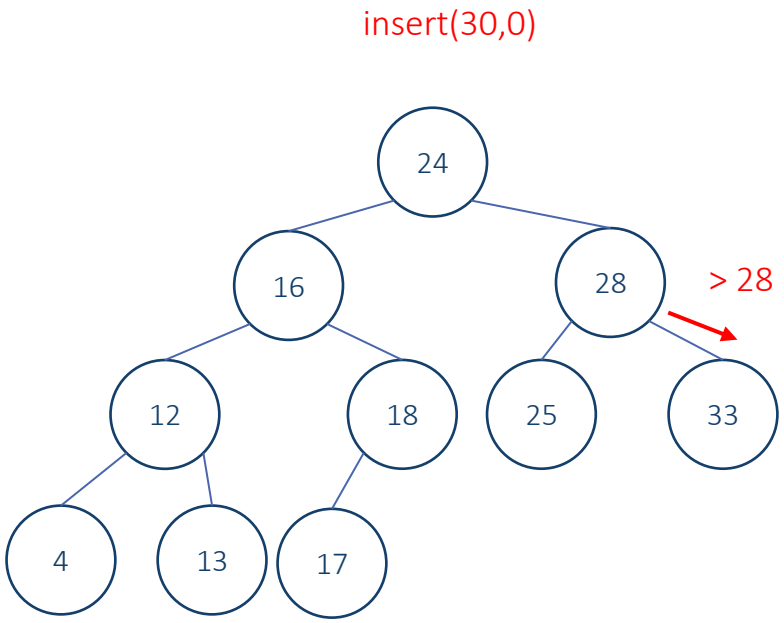
Método *put(Key k, Value v)*

- Começar pelo correspondente à raiz da árvore
- Se chave nó = k

  Alterar o valor guardado no nó para v
- Se k < chave do nó

  Colocar à esquerda
- Se k > chave do nó

  Colocar à direita



- Se chegarmos a uma árvore vazia

  Criar e colocar um novo nó com o par <key,valor>

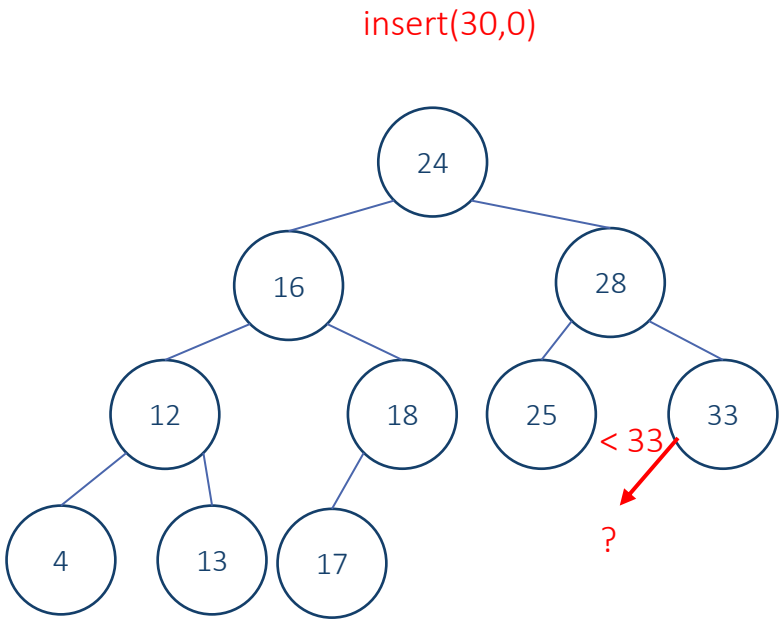
Método *put(Key k, Value v)*

- Começar pelo correspondente à raiz da árvore
- Se chave nó = k

  Alterar o valor guardado no nó para v
- Se k < chave do nó

  Colocar à esquerda
- Se k > chave do nó

  Colocar à direita



- Se chegarmos a uma árvore vazia

  Criar e colocar um novo nó com o par <key,valor>

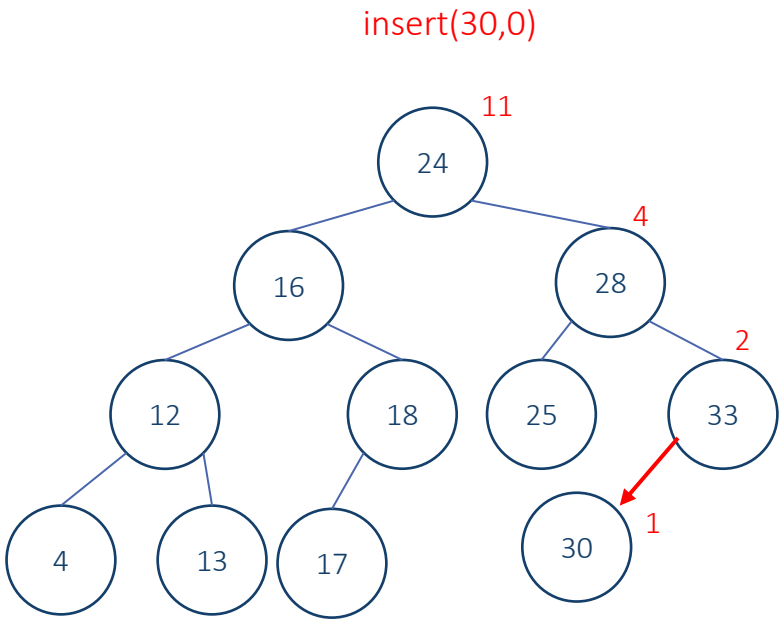
Método *put(Key k, Value v)*

- Começar pelo correspondente à raiz da árvore
- Se chave nó = k

  Alterar o valor guardado no nó para v
- Se k < chave do nó

  Colocar à esquerda
- Se k > chave do nó

  Colocar à direita

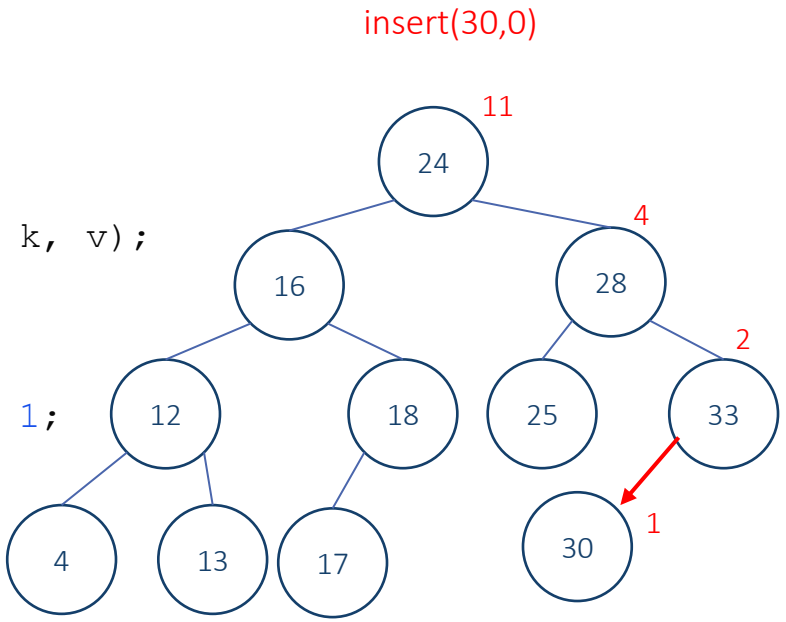


- Se chegarmos a uma árvore vazia

  Criar e colocar um novo nó com o par <key,valor>

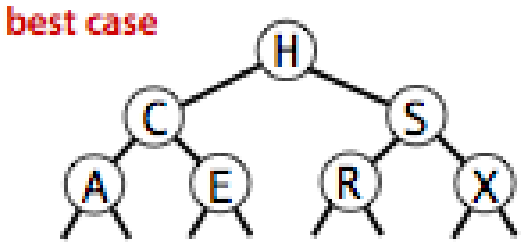
Método *put(Key k, Value v)*

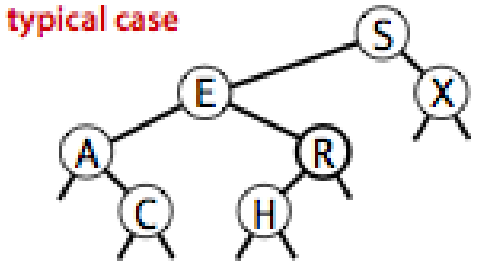
```
public void put(Key k, Value v)
{
    this.root = put(this.root, k, v);
}

private Node put(Node n, Key k, Value v)
{
    if(n == null) return new Node(k,v,1);

    int cmp = k.compareTo(n.key);
    if(cmp <0) n.left = put(n.left, k, v);
    else if(cmp > 0) n.right = put(n.right, k, v);
    else n.value = v;

    //update the node size
    n.size = size(n.left) + size(n.right) + 1;

    return n;
}
```

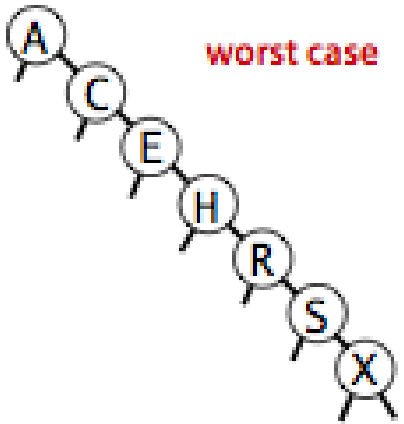


Complexidade Temporal

- Depende muito da estrutura da árvore
- e da ordem pela qual os elementos são adicionados na árvore

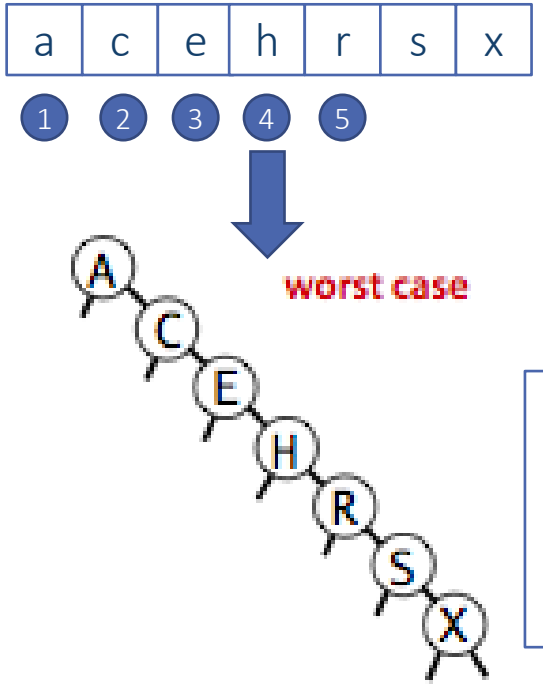






Pior caso

- Elementos colocados na árvore por ordem
  - **Problema parecido ao Quicksort**

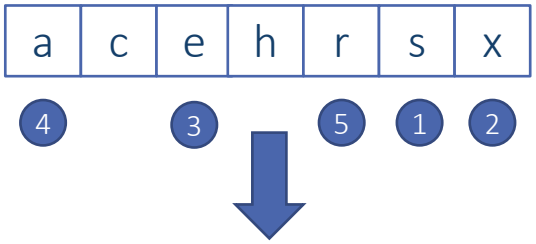


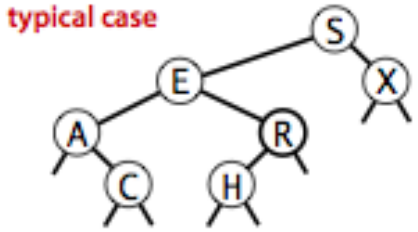
- get
  - T(n) ~ n
- put
  - T(n) ~ n

Observação:
No pior caso, a nossa árvore transforma-se em algo parecido a uma lista, e a eficiência corresponde à eficiência de percorrer uma lista

Caso médio

- Elementos colocados na árvore de forma aleatória

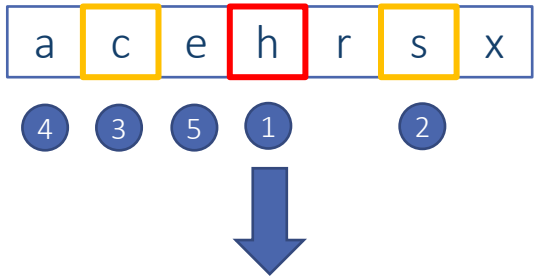




- Análise semelhante ao quicksort
- get
  - T(n) ~ 2 ln n
    ~ 1.39 $\log_2$ n
- put
  - T(n) ~ 2 ln n
    ~ 1.39 $\log_2$ n

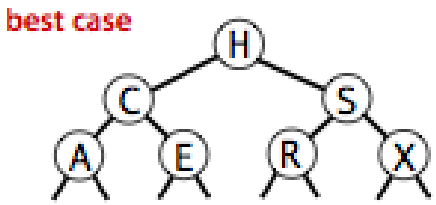
Melhor caso

- Elementos colocados na árvore
  - De forma recursiva
  - Pela mediana de cada subarray



Na prática isto não se faz, pois é muito díficil garantir esta ordem de forma natural

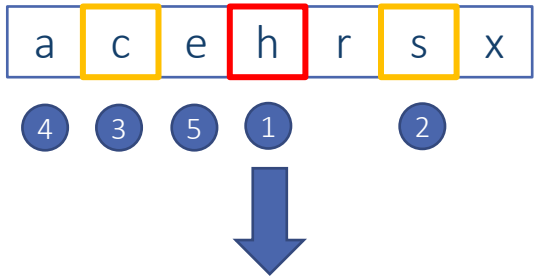
Para conseguir isto teríamos de saber à priori todos os elementos que vão ser colocados, e teríamos de ordenar o array primeiro

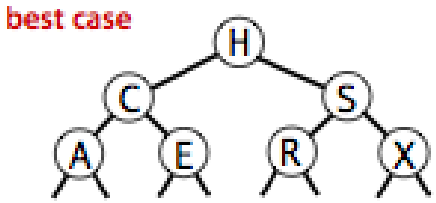


árvore perfeitamente balanceada

Melhor caso

- Elementos colocados na árvore
  - De forma recursiva
  - Pela mediana de cada subarray





- Árvore perfeitamente balanceada
  - Profundidade = $\log_2 n$
- get
  - $T(n) \sim \log_2 n$
- put
  - $T(n) \sim \log_2 n$

Complexidade Temporal

- Melhor caso é dificil de conseguir em problemas reais
- Muito eficiente em média
  - Caso médio é apenas 39% mais lento que melhor caso
- Pior caso é mau ☹

	Caso Médio		Pior Caso	
	Inserção	Pesquisa	Inserção	Pesquisa
Array Pesquisa Binária	n	$\log_2 n$	2n	$\log_2 n$
Árvore Pesquisa Binária	$1.39 \log_2 n$	$1.39 \log_2 n$	n	n

Observação: O pior caso acontece nas mesmas situações do quicksort, e por isso as soluções são semelhantes.

No entanto, enquanto que no quicksort o quickfix funciona garantidamente, aqui não. Por uma razão muito simples: No quicksort já sabemos todos os elementos a ordenar, por isso podemos fazer um shuffle. No caso de uma árvore de pesquisa muitas vezes não temos controlo sobre a ordem de inserção.