Árvores e Mapas

Luís Lopes

DCC-FCUP

Estruturas de Dados

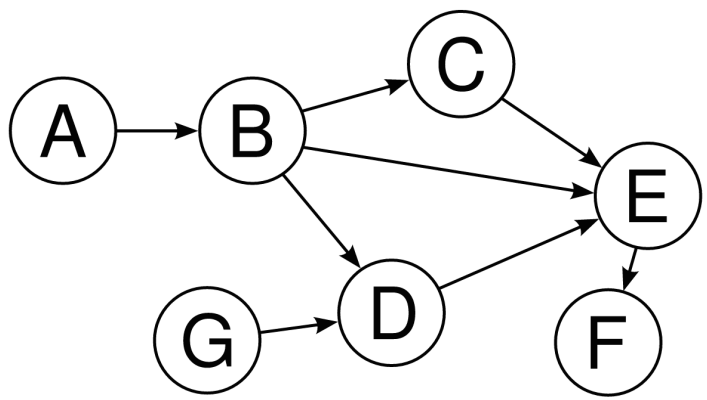
Estruturas não lineares

Os arrays e as listas são exemplos de estruturas de dados lineares, cada elemento tem:

- um predecessor único (excepto o primeiro elemento da lista);
- um sucessor único (excepto o último elemento da lista).

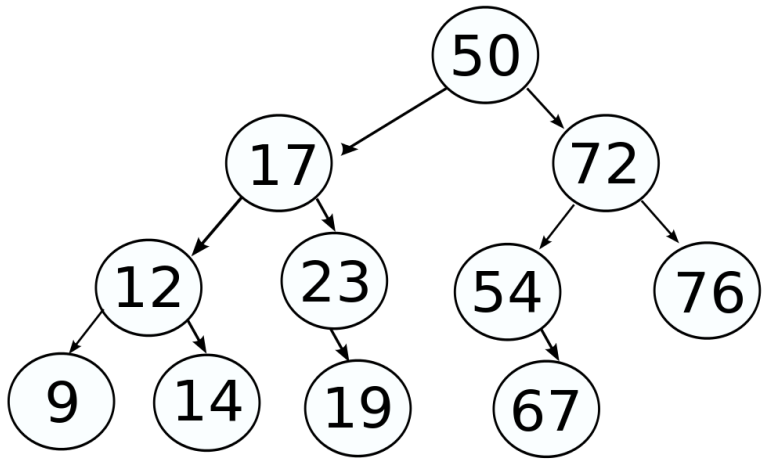
Existem outros tipos de estruturas?

- um grafo é uma estrutura de dados não-linear, pois cada um dos seus elementos, designados por nós, podem ter mais de um predecessor ou mais de um sucessor.



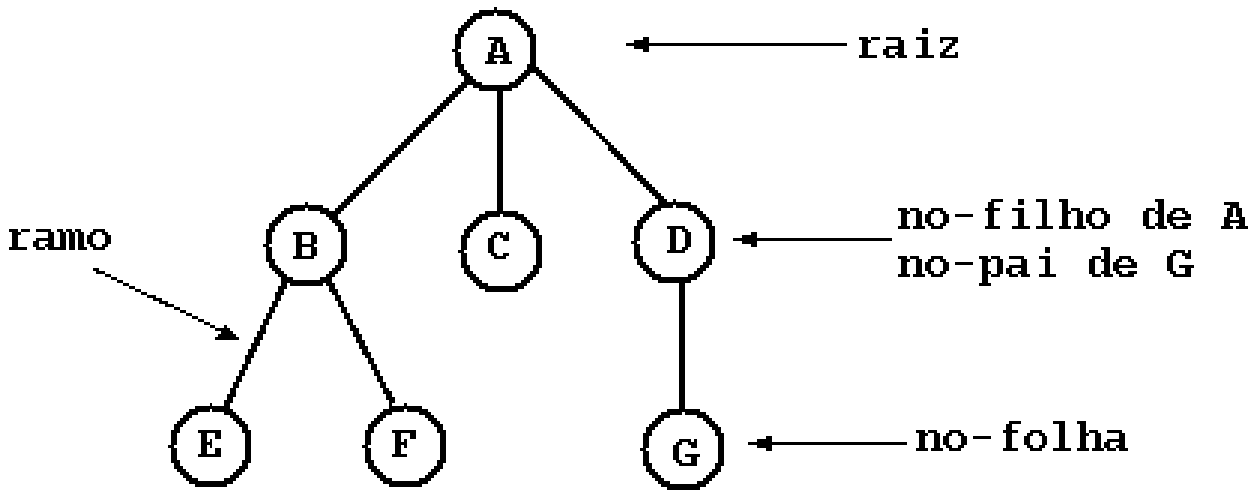
Árvores

- Uma árvore é um grafo especial;
- cada elemento, designado por nó, tem zero ou mais sucessores, mas apenas um predecessor (excepto o primeiro nó, a que se dá o nome de raiz da árvore);
- são estruturas particularmente adequadas para representar informação organizada em hierarquias;
- e.g., a estrutura de directórios (ou pastas) de um sistema de ficheiros.



Terminologia

- Ao predecessor (único) de um nó, chama-se nó-pai;
- os seus sucessores são os nós-filho;
- o grau de um nó é o número sub-árvores (ou nós-filho) que descendem desse nó;
- um nó-folha não tem filhos, tem grau 0;
- um nó-raiz não tem pai;
- os arcos que ligam os nós, chamam-se ramos.

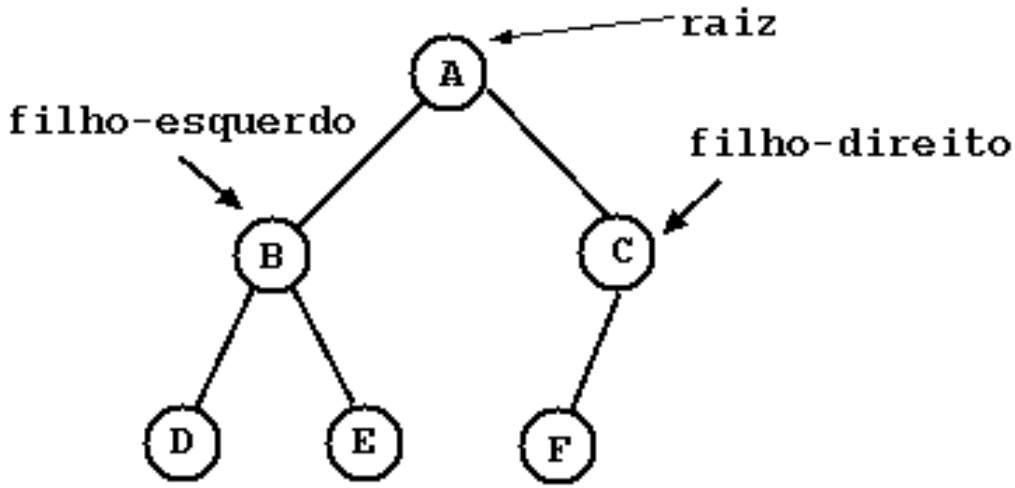


Terminologia

- Chama-se caminho a uma sequência de ramos entre dois nós;
- *uma propriedade importante das árvores é que existe apenas um caminho entre dois quaisquer nós*
- o comprimento de um caminho é o número de ramos nele contido;
- a profundidade de um nó é o comprimento do caminho desde a raiz até esse nó (a profundidade da raiz é zero);
- a profundidade de uma árvore é o comprimento do caminho desde a raiz até ao seu nó mais profundo.

Árvores Binárias

Uma árvore-binária é uma árvore de aridade 2, isto é, cada nó possui no máximo dois filhos, designados por o filho-esquerdo e o filho-direito.



Tamanho da Árvore e Profundidade

- Como determinar o número de nós n de uma árvore binária perfeita de profundidade k?
- O número de nós total é a soma dos nós dos níveis de 0 a k, i.e.

nível 0	–	$2^0 = 1$ nó
nível 1	–	$2^1 = 2$ nós
...	...	...
nível k	–	2^k nós
n	–	$2^0 + 2^1 + 2^2 + \ldots + 2^k = \sum_{j=0}^{k} 2^j$

Por indução, $n = 2^{k+1} - 1$.

Tamanho da Árvore e Profundidade

- Conhecido o número de nós de uma árvore binária perfeita, determinar a sua profundidade k:
- $n = 2^{k+1} - 1 \Leftrightarrow k = log_2(n+1) - 1$
- nota: $10 = log_2(1024)$, $20 = log_2(1048576)$
- portanto, apesar de uma árvore binária poder conter muitos nós, a distância da raiz a qualquer folha é relativamente pequena;
- isto é excelente pois significa que, com algum cuidado, podemos definir algoritmos sobre árvores (e.g., inserção, procura, remoção) que requerem apenas percorrer um caminho cujo tamanho é logarítmico no número de nós da árvore.

Árvores Binárias de Pesquisa

Em inglês: Binary Search Tree (BST)

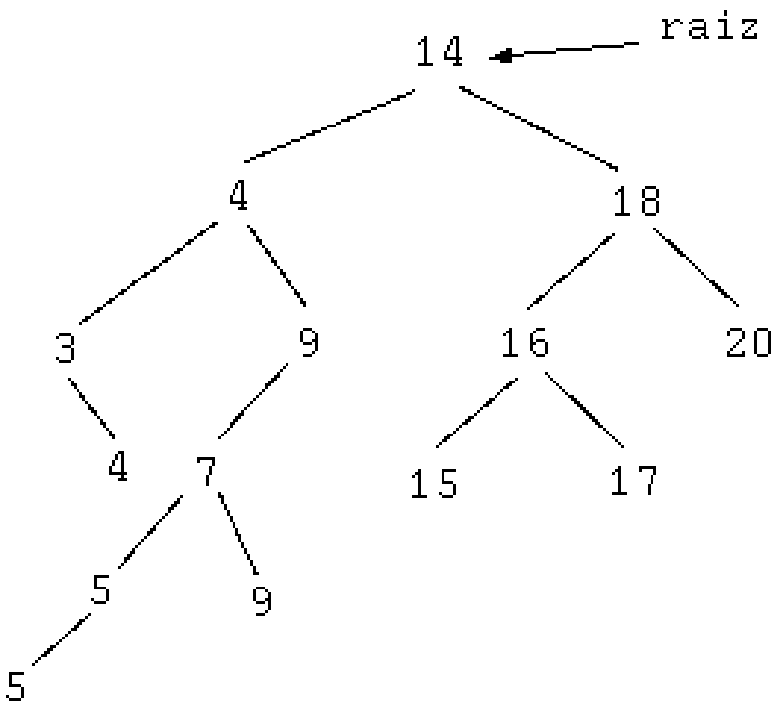
Uma árvore binária T diz-se de pesquisa se:

- T for vazia, ou
- cada nó de T contém uma chave que satisfaz as condições seguintes:
  - todas as chaves (se existirem) na sub-árvore esquerda da raiz precedem a chave da raiz;
  - a chave da raiz precede as chaves (se existirem) na sub-árvore direita;
  - as sub-árvores esquerda e direita da raiz também são árvores de pesquisa.

Árvores Binárias de Pesquisa

Munidos da definição, podemos agora construir uma árvore binária de pesquisa:

- o primeiro valor fica na raiz da árvore.
- os seguintes são à esquerda ou direita da raiz, obedecendo à relação de ordem, como folhas e em níveis cada vez mais baixos.



Sequência de valores dada:
14,18,4,9,7,16,3,5,4,17,20,15,9,5

Operações do Tipo Abstracto de Dados `BSTree`

As operações associadas a uma árvore binária de pesquisa são:

`isEmpty(BSTree<K,V>)`	a árvore está vazia?
`size(BSTree<K,V>)`	número de nós
`depth(BSTree<K,V>)`	profundidade
`insert(BSTree<K,V>, K, V)`	adiciona par chave/valor
`contains(BSTree<K,V>, K)`	a chave está na árvore?
`get(BSTree<K,V>, K)`	o valor associado à chave
`remove(BSTree<K,V>, K)`	remove o nó associado a chave

Implementação: BSTNode.java

```
class BSTNode<K extends Comparable<? super K>,V> {
  K            key;
  V            val;
  BSTNode<K,V> left;
  BSTNode<K,V> right;

  BSTNode(K k, V v, BSTNode<K,V> l, BSTNode<K,V> r) {
     key   = k;
     val   = v;
     left  = l;
     right = r;
  }
  BSTNode(K k, V v) {
     key   = k;
     val   = v;
     left  = null;
     right = null;
  }
}
```

Implementação: BSTree.java

```
class BSTree<K extends Comparable<? super K>,V> {
  BSTNode<K,V> root;

  BSTree(BSTNode<K,V> node) {
    root = node;
  }

  BSTree() {
    root = null;
  }
}
```

Implementação: LibBSTree.java

```
public static ... boolean isEmpty(BSTree<K,V> t) {
  return (t.root == null);
}
public static ... int size(BSTree<K,V> t) {
    return size(t.root);
}
private static ... int size(BSTNode<K,V> n) {
  if ( n == null )
    return 0;
  else
    return 1 + size(n.left) + size(n.right);
}
public static ... int depth(BSTree<K,V> t) {
  return depth(t.root);
}
private static ... int depth(BSTNode<K,V> n) {
  if ( n == null )
    return 0;
  else
    return 1 + Math.max( depth(n.left), depth(n.right));
}
```

Implementação: LibBSTree.java

```
public static ...
void insert(BSTree<K,V> t, K key, V val) {
  t.root = insert(t.root, key, val);
}

private static ...
BSTNode<K,V> insert(BSTNode<K,V> n, K key, V val) {
  if ( n == null )
    return new BSTNode<K,V>(key,val);
  else
    int cmp = key.compareTo(n.key);
    if ( cmp == 0 )
      n.val = val;
    else if (cmp < 0)
      n.left = insert(n.left, key, val);
    else
      n.right = insert(n.right, key, val);
  return n;
}
```

Procura de valores

Dada uma árvore binária de pesquisa `t` e uma chave `k`, pretende-se verificar se `t` contém `k`.
Quando se pensa em percorrer uma árvore, deve pensar-se numa solução recursiva em que temos de lidar com três casos:

1. árvore vazia, pelo que a árvore `t` não contém `k`
2. nó corrente contém `k`
3. continuar a procura numa das sub-árvores, de acordo com a relação de ordem entre `k` e a chave do nó corrente.

Implementação: LibBSTree.java

```
public static <K extends Comparable<? super K>,V>
boolean contains(BSTree<K,V> t, K key) {
  return contains(t.root, key);
}

private static <K extends Comparable<? super K>,V>
boolean contains(BSTNode<K,V> n, K key) {
  if ( n == null )
    return false;
  else
    if ( key.compareTo(n.key) == 0 )
      return true;
    if ( key.compareTo(n.key) < 0 )
      return contains(n.left, key);
    else
      return contains(n.right, key);
}
```

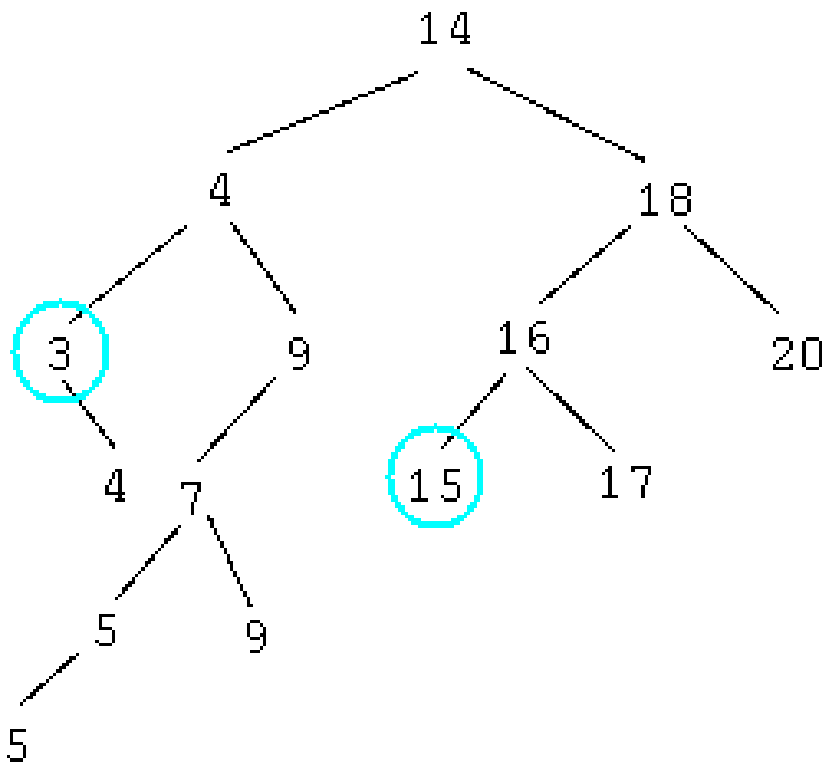
Implementação: LibBSTree.java

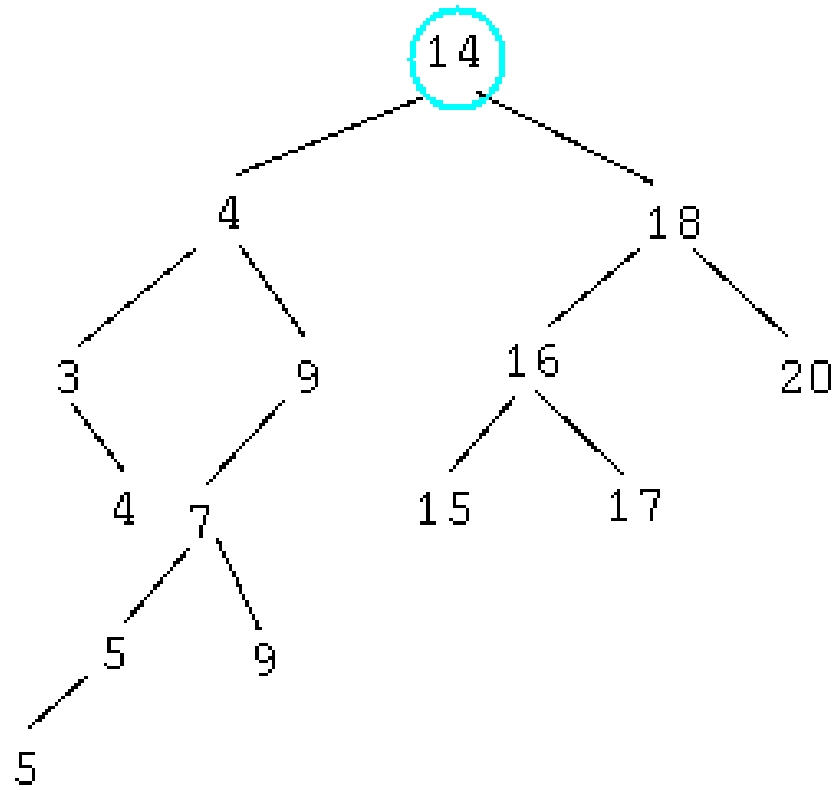
```
public static <K extends Comparable<? super K>,V>
V get(BSTree<K,V> t, K key) {
  return get(t.root, key);
}

private static <K extends Comparable<? super K>,V>
V get(BSTNode<K,V> n, K key) {
  if ( n == null )
    return null;
  else
    if ( key.compareTo(n.key) == 0 )
      return n.val;
    if ( key.compareTo(n.key) < 0 )
      return get(n.left, key);
    else
      return get(n.right, key);
}
```

Remover valores

Casos especiais para o nó a remover:

- nó folha (e.g. remover o 15)
- nó-interior com apenas um filho (e.g. remover o 3)
- nó interior com dois filhos (e.g. remover o 14)





Remover valores

A operação de remover um nó que contém uma chave `k` de uma árvore `t`, caso tal nó exista, é um pouco mais complexa pois a árvore final tem de satisfazer as propriedades de uma árvore binária de pesquisa.

1. localizar o nó com a chave `k`, seja `n` esse nó;
2. se `n` é um nó-folha, simplesmente remove-se o nó;
3. se `n` é um nó-interior, é necessário mais cuidado para não ficarmos com 2 árvores desconexas:
   - se `n` só tiver um filho, a sub-árvore pendurada nesse nó toma o lugar de `n`
   - se `n` tiver dois filhos, então devemos procurar o nó com menor chave entre os descendentes do filho-direito (ou o maior dos descendentes do filho-esquerdo) para tomar o lugar de `n`.

Implementação: LibBSTree.java

```
public static ...
void remove(BSTree<K,V> t, K key) {
  t.root = remove(t.root, key);
}
private static ...
BSTNode<K,V> remove(BSTNode<K,V> n, K key) {
  if ( n != null )
    if ( key.compareTo(n.key) < 0 )
      n.left = remove(n.left, key);
    else if ( key.compareTo(n.key) > 0 )
      n.right = remove(n.right, key);
    else if ( n.left == null )
      n = n.right;
    else
      if ( n.right == null )
        n = n.left;
      else
        n.right = removeMin(n, n.right);
  return n;
}
```

Implementação: LibBSTree.java

```
private static ...
BSTNode<K,V> removeMin(BSTNode<K,V> n1, BSTNode<K,V> n2) {
  if ( n2.left == null ) {
    n1.key = n2.key;
    n1.val = n2.val;
    return n2.right;
  } else {
    n2.left = removeMin(n1, n2.left);
    return n2;
  }
}
```

Complexidade das BST

- Para uma árvore binária de pesquisa com n nós:
- o comprimento médio do caminho da raiz até uma folha é $\mathcal{O}(\log n)$, correspondente à profundidade da árvore;
- verificar se v pertence à árvore é assim $\mathcal{O}(\log n)$;
- os métodos `insert()`, `contains()` e `remove()` são $\mathcal{O}(\log n)$;
- é comum que ao inserirmos n valores aleatórios, a árvore não seja completa, pode até ficar uma sequência ordenada. Neste caso cada inserção é $\mathcal{O}(n)$.

Complexidade das BST

- Uma árvore binária em média estará próxima de uma árvore completa ou de uma sequência?
- isto decide se o tempo médio é $\mathcal{O}(\log n)$ ou $\mathcal{O}(n)$
- é possível demonstrar que a complexidade amortizada é $\mathcal{O}(\log n)$.

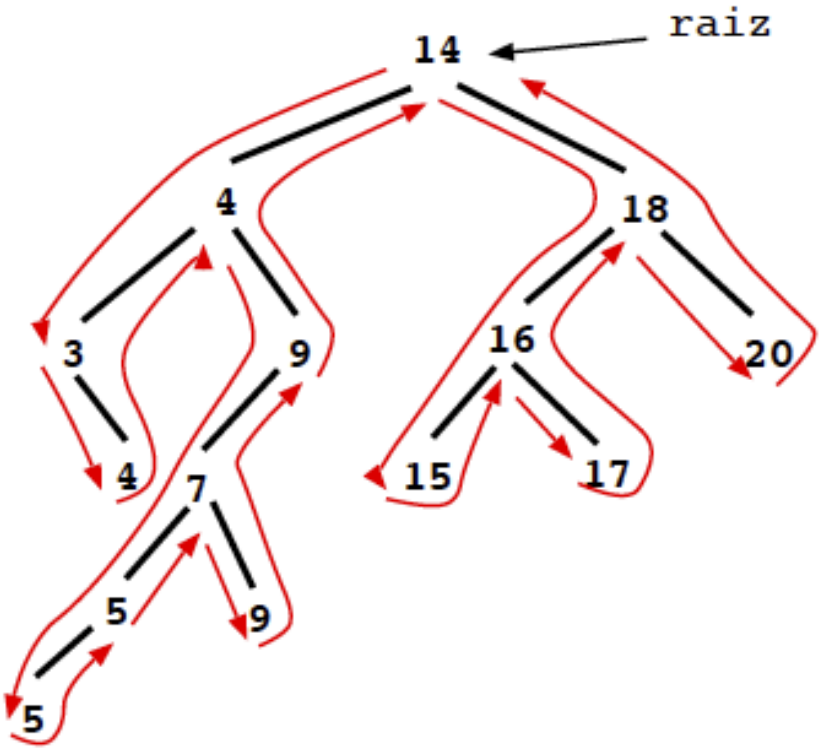
Visita em profundidade

O método de pesquisa **inorder** é particularmente útil para pesquisar os nós de uma árvore de pesquisa binária em que a ordem dos valores (chaves de pesquisa) é crescente. Assim como permite listar por ordem a sequência de valores da árvore.
O método de pesquisa **inorder** (ou **depth-first**) consiste em visitar a árvore do seguinte modo:

1. sub-árvore esquerda
2. raiz
3. sub-árvore direita

Visita em profundidade (depth-first)

A pesquisa inorder equivale a fazer-se o seguinte percurso:



visita inorder:

3,4,4,5,5,7,9,9,
14,15,16,17,18,20

Visita em profundidade (depth-first)

```
public static <K extends Comparable<? super K>,V>
String toString(BSTree<K,V> t) {
  return toString(t.root);
}

private static <K extends Comparable<? super K>,V>
String toString(BSTNode<K,V> n) {
  if ( n != null )
    return toString(n.left) +
           "(" + n.key + "," + n.val + ")" +
           toString(n.right);
  return "";
}
```

Visitas Preorder e Postorder

Método preorder:

1. visitar a raiz
2. pesquisar em preorder a sub-árvore esquerda
3. pesquisar em preorder a sub-árvore direita

Método postorder:

1. pesquisar em postorder a sub-árvore esquerda
2. pesquisar em postorder a sub-árvore direita
3. visitar a raiz

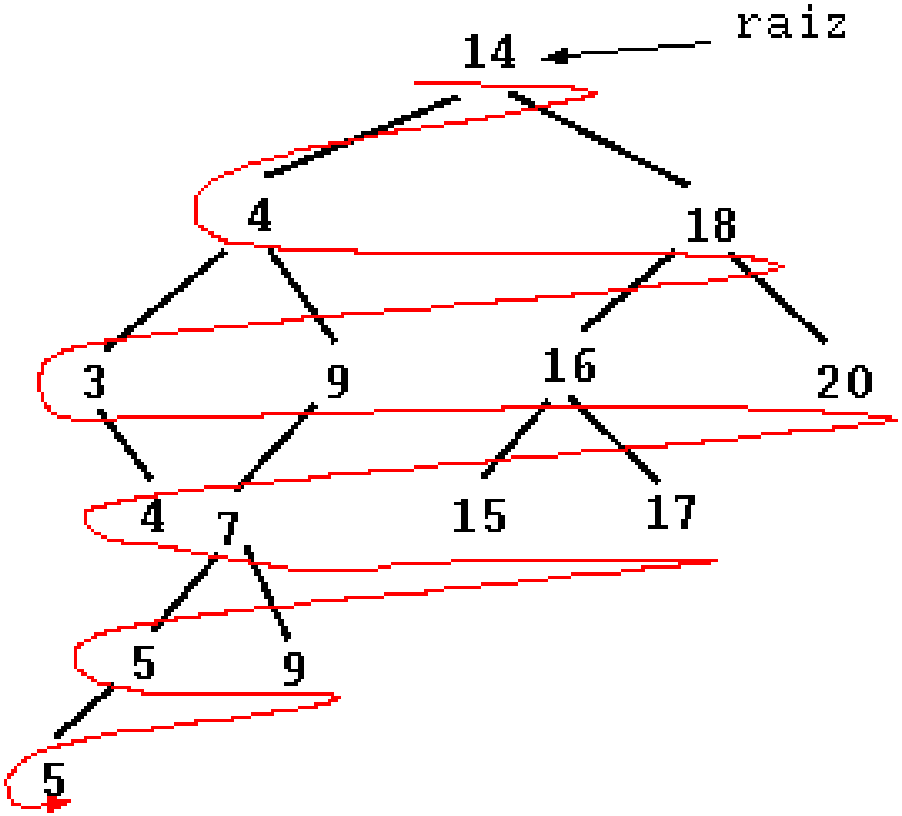
Exercício: implemente estes métodos de pesquisa.

Visita em profundidade não recursiva (com pilha)

```
public static ...
void visitDepthFirst(BSTree<K,V> t) {
  visitDepthFirst(t.root);
}

private static ...
void visitDepthFirst(BSTNode<K,V> n) {
  Stack<BSTNode<K,V>> stack = new Stack<BSTNode<K,V>>();
  BSTNode<K,V> np = n;
  while (( np != null ) || !stack.isEmpty() ) {
    while ( np != null ) {
      stack.push( np );
      np = np.left;
    }
    if ( !stack.isEmpty() ) {
      np = stack.pop();
      System.out.println("(" + n.key + "," + n.val + ")");
      np = np.right;
    }
  }
}
```

Visita em largura (breadth-first)

Um outro método de percorrer os nós de uma árvore é fazê-lo por níveis, i.e. em largura-primeiro (ou breadth-first).




```
visita breadth-first:
   14,4,18,3,9,16,20,
   4,7,15,17,5,9,5
```

Visita em largura não recursiva (com fila)

```
public static <K extends Comparable<? super K>,V>
void visitBreadthFirst(BSTree<K,V> t) {
  visitBreadthFirst(t.root);
}

private static <K extends Comparable<? super K>,V>
void visitBreadthFirst(BSTNode<K,V> n) {
  Queue<BSTNode<K,V>> queue =
    new LinkedList<BSTNode<K,V>>();
  queue.add(n);
  while (!queue.isEmpty()) {
    BSTNode<K,V> np = queue.remove();
    System.out.println("(" + np.key + "," + np.val + ")");
    if ( np.left != null )
      queue.add(np.left);
    if ( np.right != null )
      queue.add(np.right);
  }
}
```

Mapas

Um mapa é uma estrutura de dados que mantém uma colecção de pares (chave,valor) de tal forma que cada chave ocorre uma única vez na colecção. São também conhecidos como arrays associativos ou dicionários.

Exemplo: uma agenda telefónica

```
{(Nuno,960234576), (Clara,960112343), (Laura,919245333) }
```

Problema: como implementar de forma eficiente operações de inserção, pesquisa e remoção de elementos?

Um solução: implementar o mapa usando uma árvore binária de pesquisa!

Operações do Tipo Abstracto de Dados `Map`

As operações associadas a um mapa são:

`size(Map<K,V>)`	número de elementos no mapa
`put(Map<K,V>, K, V)`	adiciona par com chave/valor
`get(Map<K,V>, K)`	retorna o valor associado a chave
`contains(Map<K,V>, K)`	verifica se chave ocorre
`remove(Map<K,V>, K)`	remove par associado a chave

Implementação: Map.java

```
class Map<K extends Comparable<? super K>,V> {
  BSTree<K,V> tree;

  Map() { tree = new BSTree<K,V>(); }
}
```

Implementação: LibMap.java

```
class LibMap<K extends Comparable<? super K>,V> {

  public static ... int size(Map<K,V> m) {
    return LibBSTree.size(m.tree);
  }
  public static ... void put(Map<K,V> m, K key, V val) {
    LibBSTree.insert(m.tree, key, val);
  }
  public static ... V get(Map<K,V> m, K key) {
    return LibBSTree.get(m.tree, key);
  }
  public static ... boolean contains(Map<K,V> m, K key) {
    return LibBSTree.contains(m.tree, key);
  }
  public static ... void remove(Map<K,V> m, K key) {
    LibBSTree.remove(m.tree, key);
  }

}
```

Implementação: TestMap.java

```
public class TestMap {
  public static void main(String[] args) {
    String  phoneList =
      "Nuno 960234576  Clara 960112343
       Laura 919245333 Jorge 939122227";
    Scanner in = new Scanner(phoneList);
    Map<String,Integer> m1 = new Map<String,Integer>();
    while(in.hasNext())
      LibMap.put(m1, in.next(), in.nextInt());
    System.out.println("size:  " + LibMap.size(m1));
    System.out.println(LibMap.toString(m1));
    System.out.println("Laura's phone is " +
      LibMap.get(m1, "Laura"));
    System.out.println("Keys -> " + LibMap.keys(m1));
    Map<Integer,String> m2 = LibMap.invert(m1);
    System.out.println("the inverted map is as follows...");
    System.out.println(LibMap.toString(m2));
    System.out.println("Keys -> " + LibMap.keys(m2));
  }
}
```