# Trees, Search Trees


- Searching
- Tree Data Structures
- Binary Search Trees
- Insertion into BSTs
- Representing BSTs
- Searching in BSTs
- Insertion into BSTs
- Tree Traversal
- Joining Two Trees
- Deletion from BSTs

# ❖ Searching

Search is an extremely common application in computing

- given a (large) collection of items and a key value
- find the item(s) in the collection containing that key
    - item = (key, $val_1$, $val_2$, …)  (i.e. a structured data type)
    - key = value used to distinguish items  (e.g. student ID)

Applications:  Google,  databases, .....

## ❖ ... Searching

Many approaches have been developed for the "search" problem

Different approaches determined by properties of data structures:

- arrays: linear, random-access, in-memory
- linked-lists: linear, sequential access, in-memory
- files: linear, sequential access, external

Search costs:

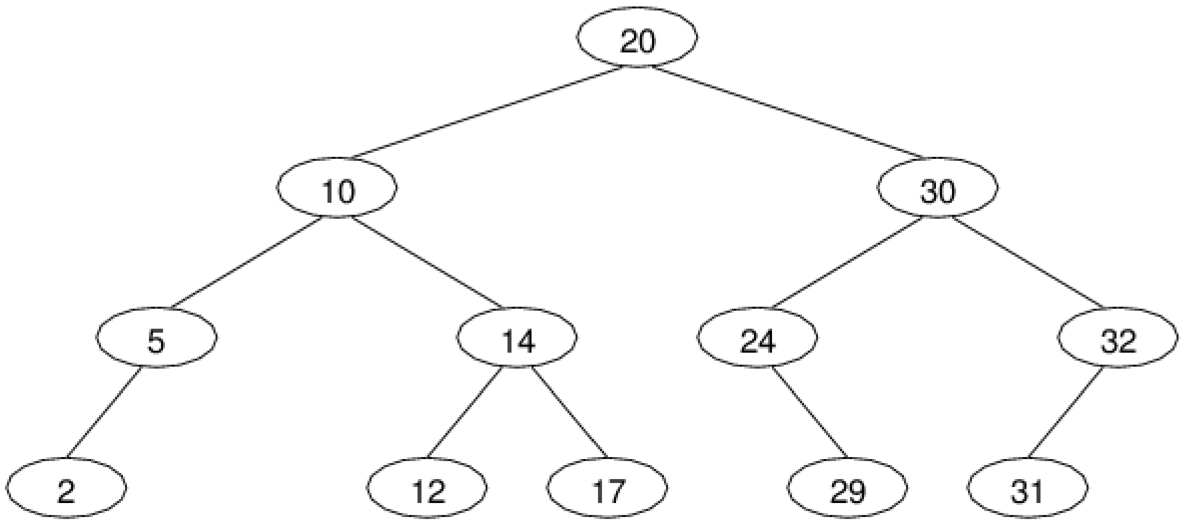
| | Array | List | File |
|---|---|---|---|
| Unsorted | O(n)<br>(linear scan) | O(n)<br>(linear scan) | O(n)<br>(linear scan) |
| Sorted | O(log n)<br>(binary search) | O(n)<br>(linear scan) | O(log n)<br>(lseek,lseek,...) |

## ❖ ... Searching

Maintaining arrays and files in sorted order is costly.

Search trees are efficient to search but also efficient to maintain.

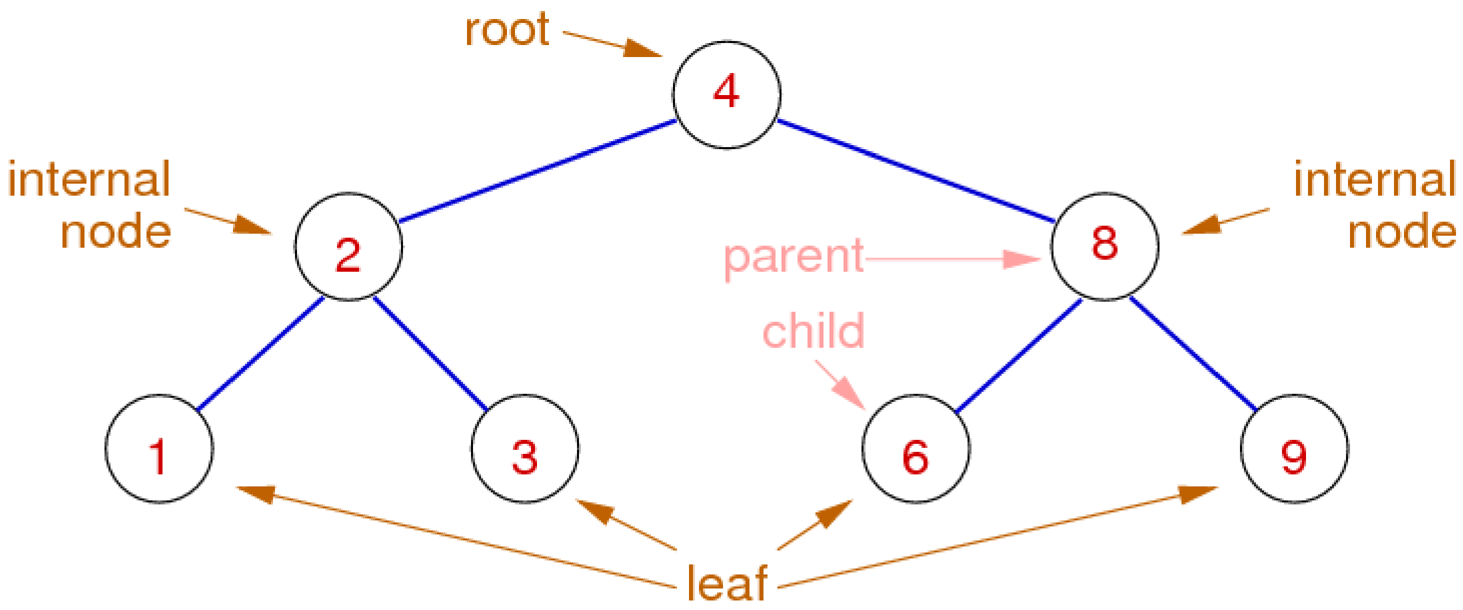
Example: the following tree corresponds to the sorted array
**[2,5,10,12,14,17,20,24,29,30,31,32]**:



# ❖ Tree Data Structures

Trees are connected graphs

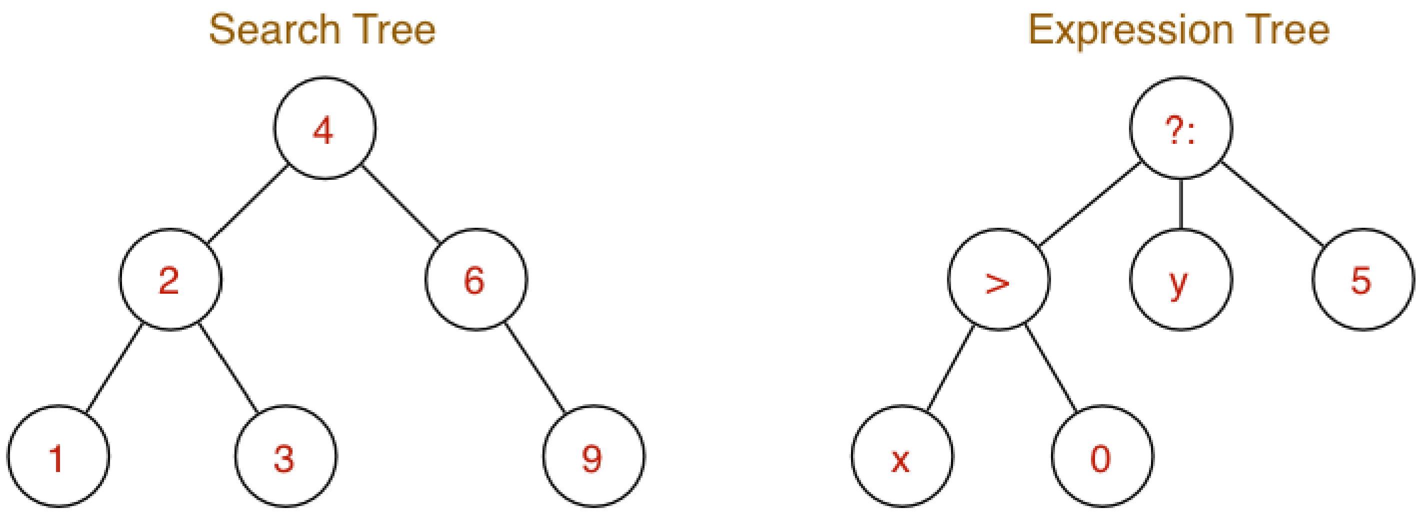
- with nodes and edges (called *links*), but no cycles (no "up-links")
- each node contains a data value (or key+data)
- each node has links to ≤ *k* other child nodes (*k=2* below)



# ❖ ... Tree Data Structures

Trees are used in many contexts, e.g.

- representing hierarchical data structures (e.g. expressions)
- efficient searching (e.g. sets, symbol tables, ...)



## ❖ ... Tree Data Structures

Real-world example: organisational structure

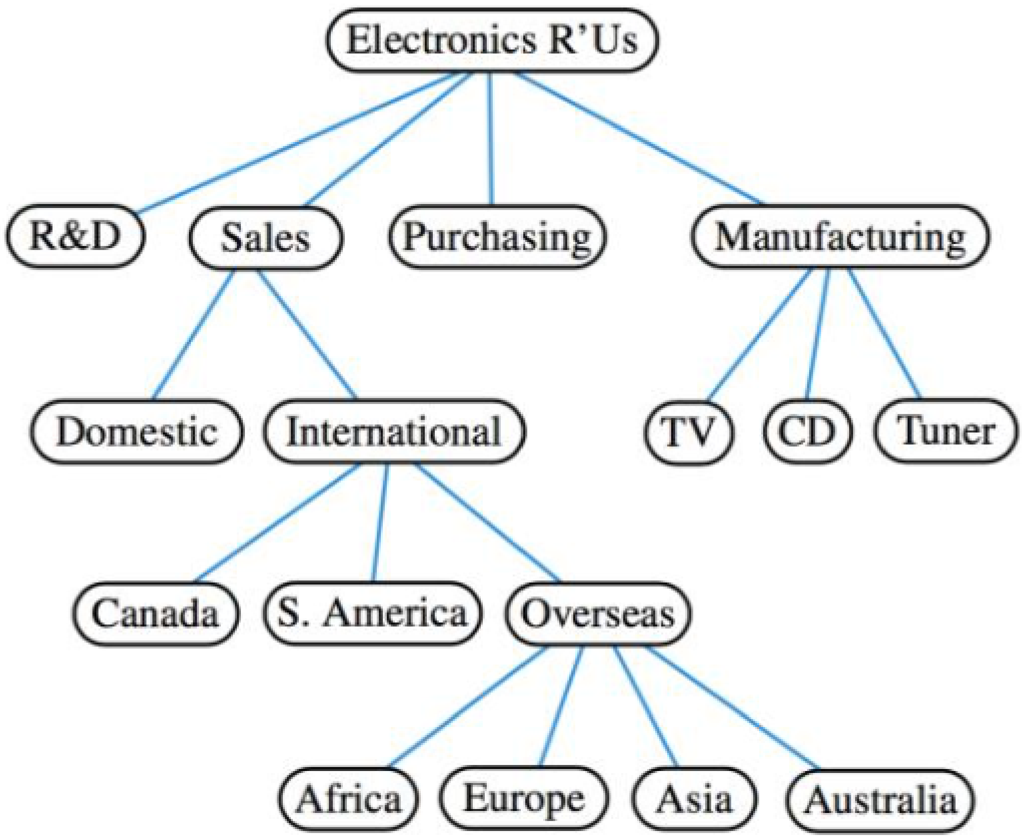



Diagram from "Data Structures and Algorithms in Java" (6th ed) by Goodrich et al

## ❖ ... Tree Data Structures

Real-world example: hierarchical file system (e.g. Linux)

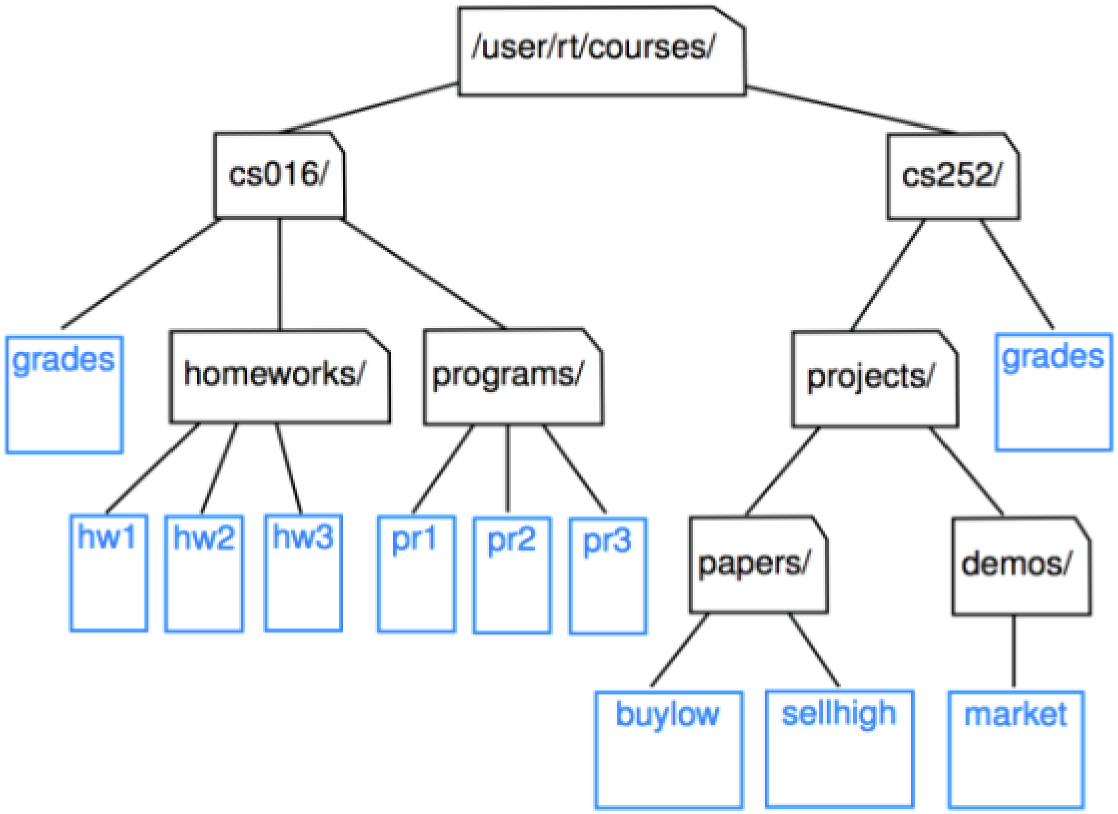



Diagram from "Data Structures and Algorithms in Java" (6th ed) by Goodrich et al

# ❖ ... Tree Data Structures

Real-world example: structure of a typical book

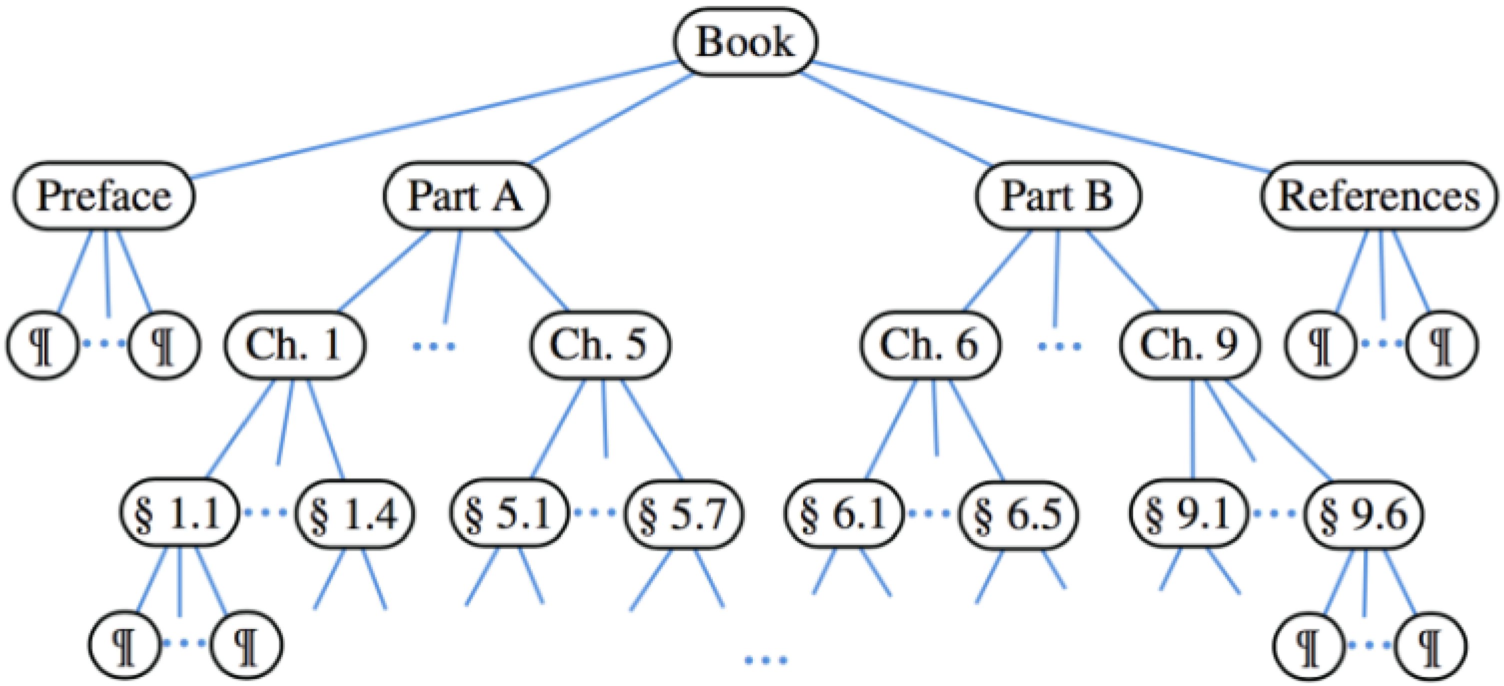



Diagram from "Data Structures and Algorithms in Java" (6th ed) by Goodrich et al

# ❖ ... Tree Data Structures

Real-world example: a decision tree

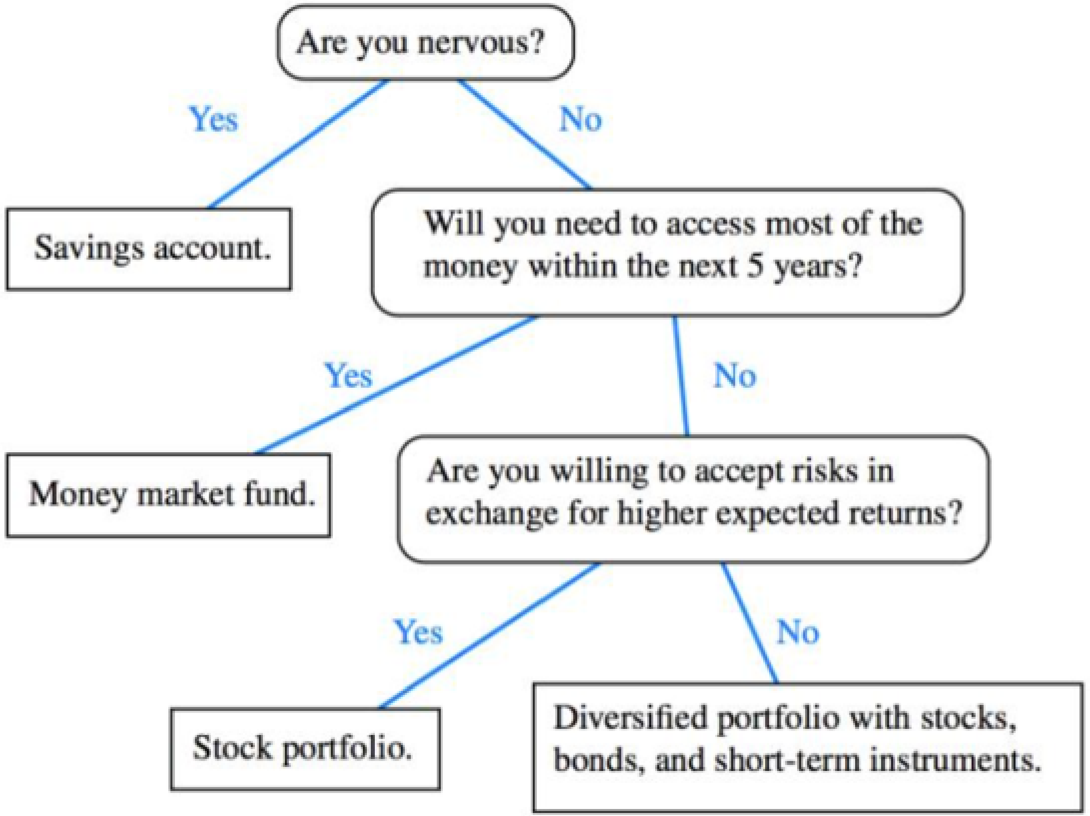
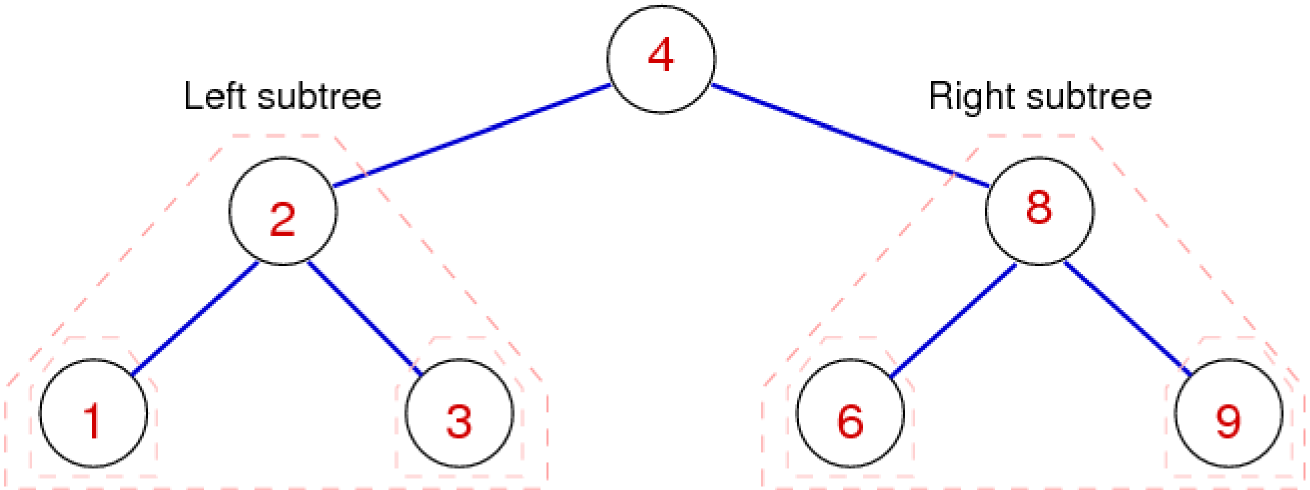



Diagram from "Data Structures and Algorithms in Java" (6th ed) by Goodrich et al

# ❖ ... Tree Data Structures

A *binary tree* is either

- empty (contains no nodes)
- consists of a node, with two subtrees
  - node contains a value (typically key+data)
  - left and right subtrees are *binary trees* (recursive)



# ❖ ... Tree Data Structures

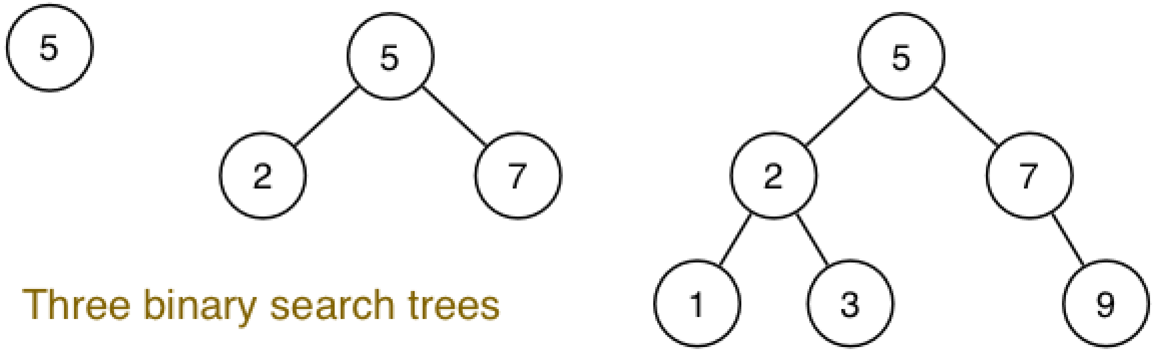
Other special kinds of tree

- *m*-ary tree: each internal node has exactly $m$ children
- B-tree: each internal node has $n/2 \leq$ #children $\leq n$
- Ordered tree: all left values < root, all right values > root
- Balanced tree: has ≅minimal height for a given number of nodes
- Degenerate tree: has ≅maximal height for a given number of nodes

## ❖ Binary Search Trees

Binary search trees (or BSTs) are ordered trees

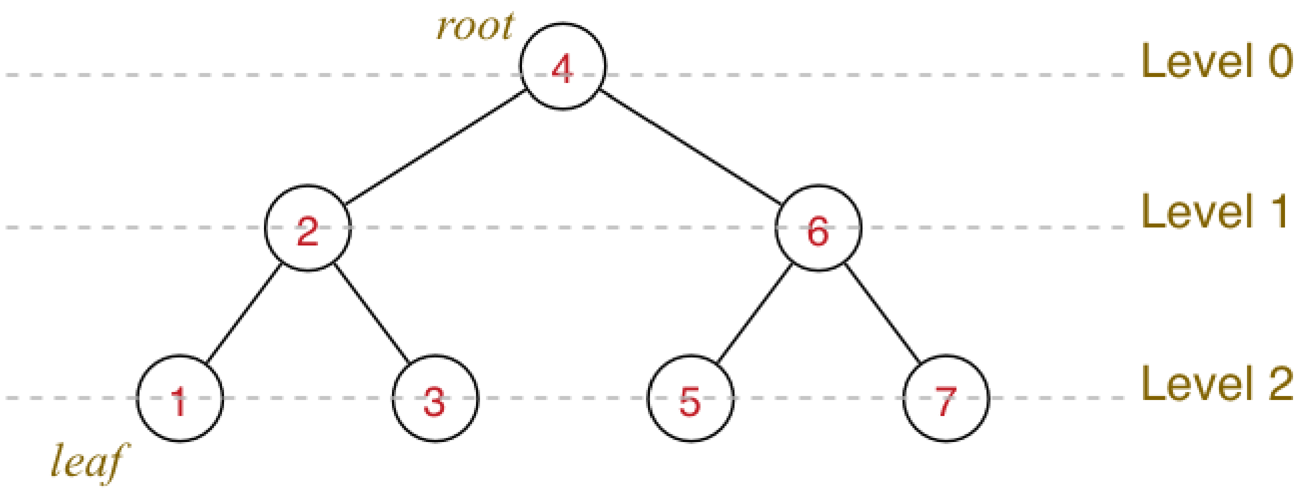
- each node is the root of 0, 1 or 2 subtrees
- all values in any left subtree are less than root
- all values in any right subtree are greater than root
- these properties applies over all nodes in the tree



# ❖ ... Binary Search Trees

Level of node = path length from root to node

Height (or depth) of tree = max path length from root to leaf



# ❖ ... Binary Search Trees

Some properties of trees ...

Ordered

- ∀ nodes: max(left subtree) < root < min(right subtree)

Perfectly-balanced tree

- ∀ nodes: #nodes(left subtree) = #nodes(right subtree)

Height-balanced tree

- ∀ nodes: height(left subtree) = height(right subtree)

Note: time complexity of tree algorithms is typically *O(height)*

# ❖ ... Binary Search Trees

Operations on BSTs:

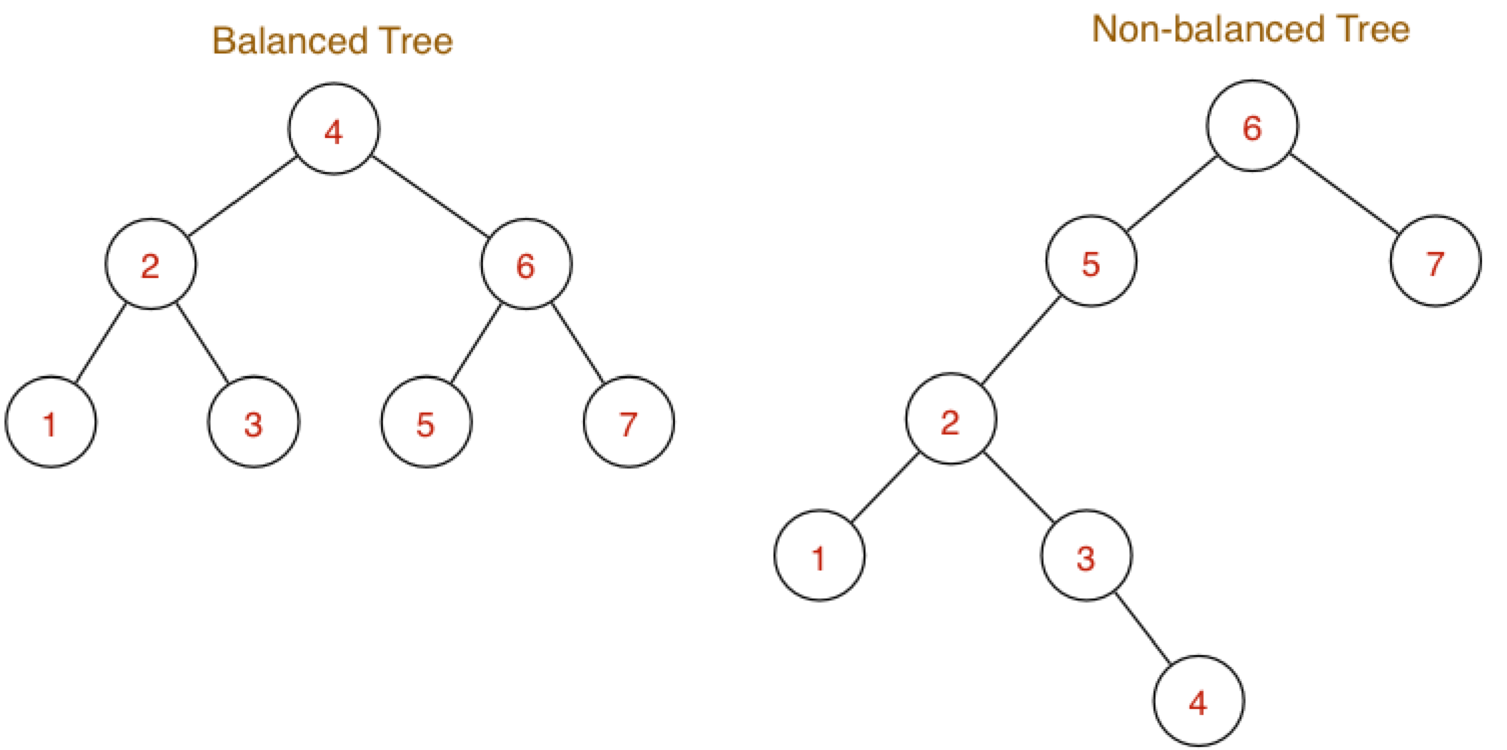
- insert(Tree,Item) … add new item to tree via key
- delete(Tree,Key) … remove item with specified key from tree
- search(Tree,Key) … find item containing key in tree
- plus, "bookkeeping" … new(), free(), show(), …

Notes:

- nodes contain `Item`s; we generally show just `Item.key`
- keys are unique (not technically necessary)

# ❖ ... Binary Search Trees

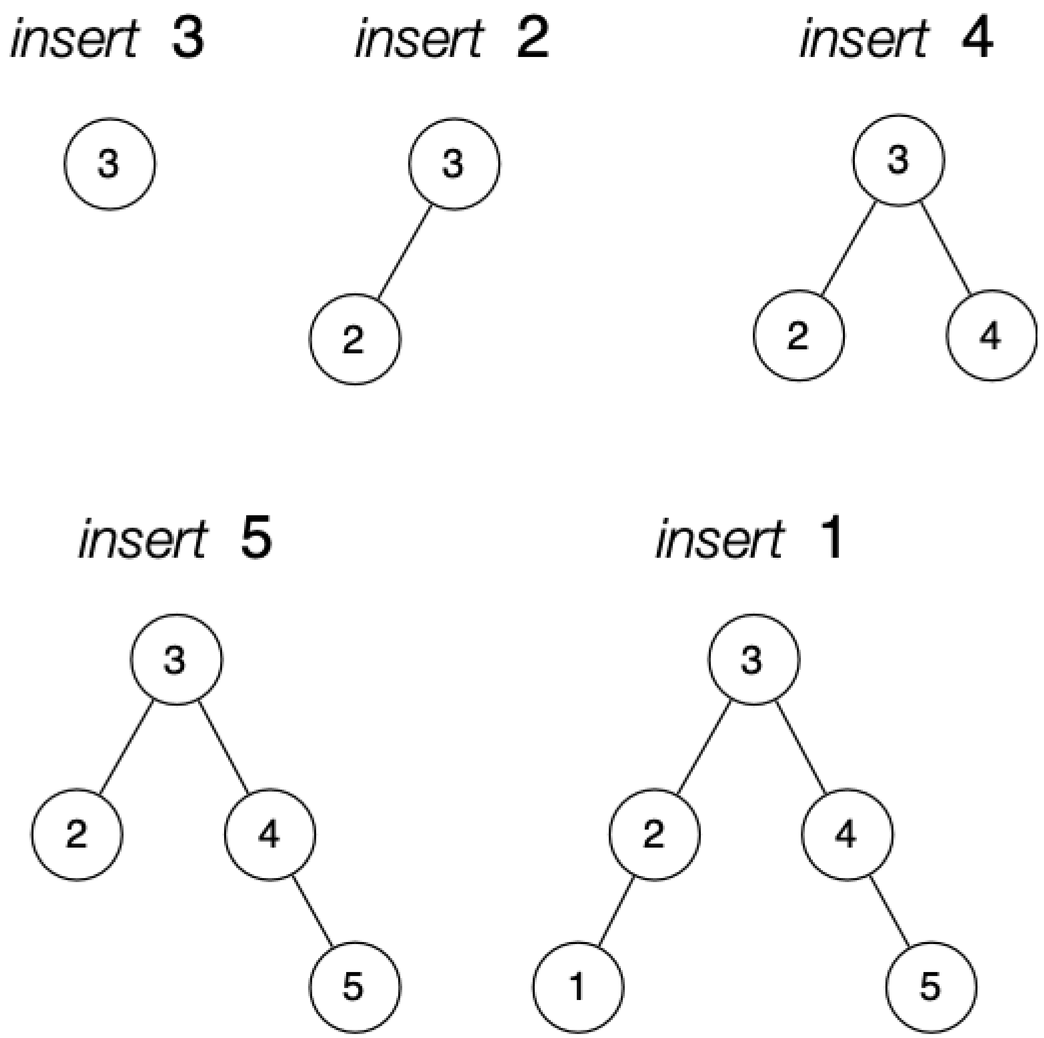
Examples of binary search trees:



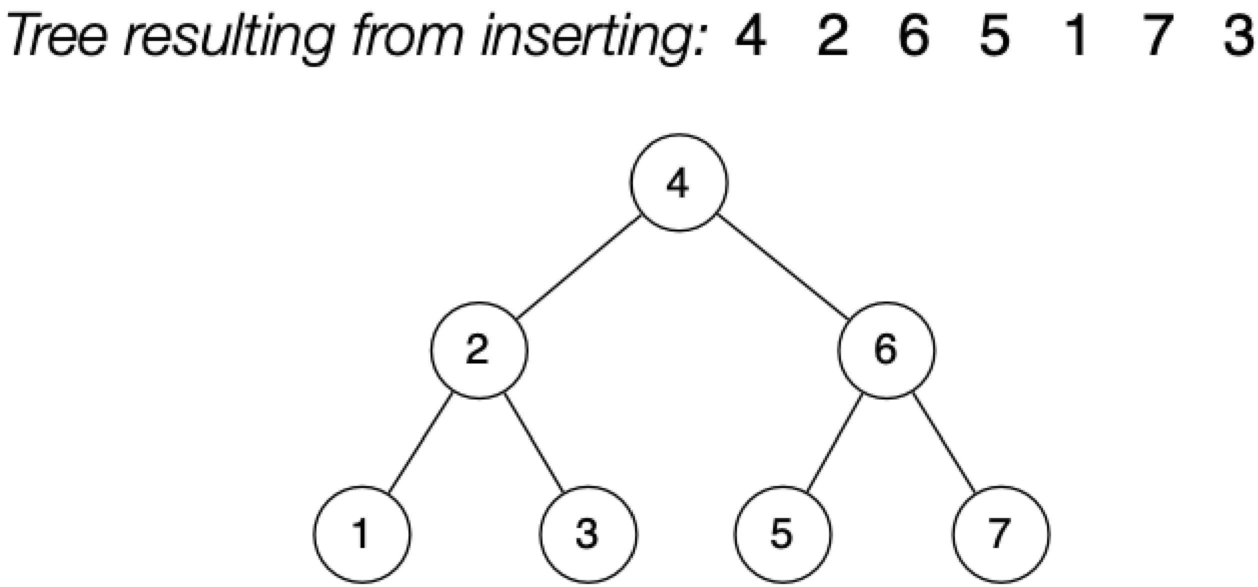

Shape of tree is determined by order of insertion.

# ❖ Insertion into BSTs

Steps in inserting values into an initially empty BST

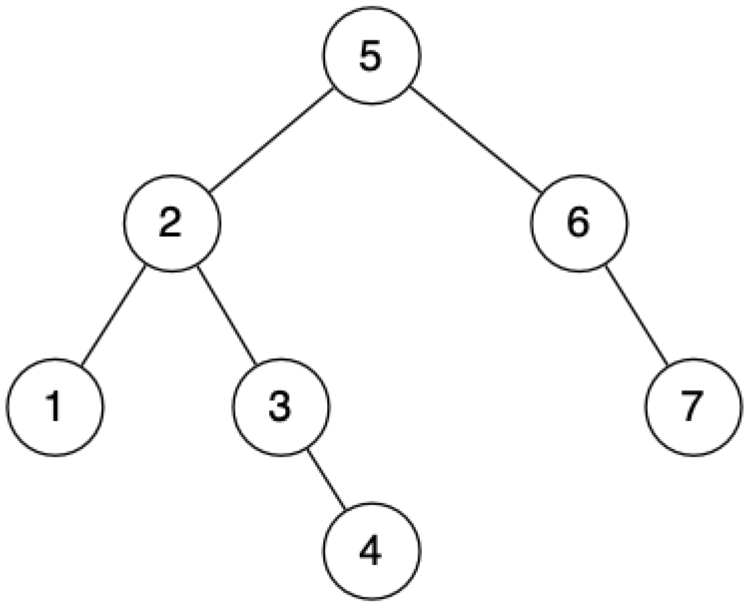


## ❖ ... Insertion into BSTs



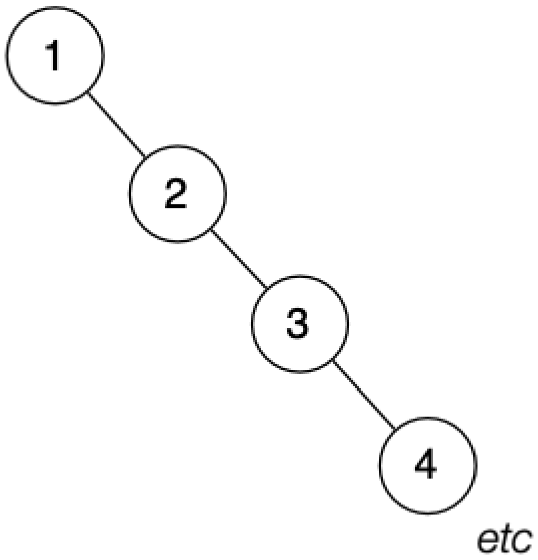
## ❖ ... Insertion into BSTs

*Tree resulting from inserting:* 5 6 2 3 4 7 1



## ❖ ... Insertion into BSTs

*Tree resulting from inserting:* 1 2 3 4 5 6 7

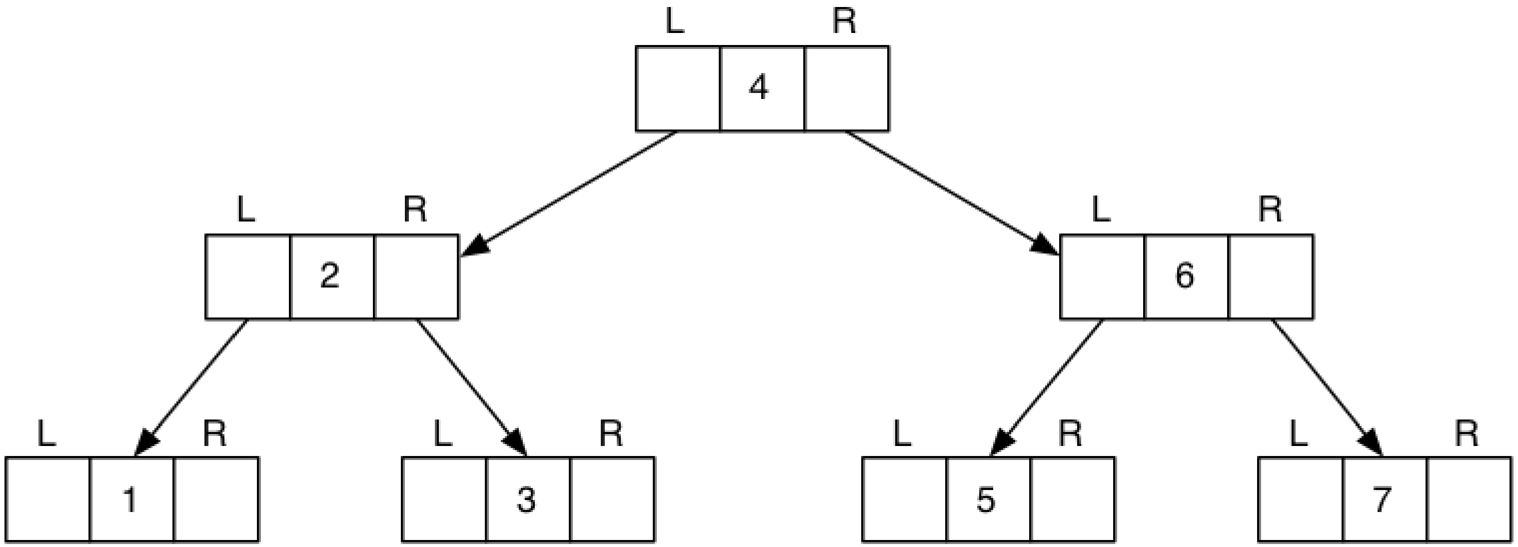


## ❖ Representing BSTs

Binary trees are typically represented by node structures

- where each node contains a value, and pointers to child nodes

Most tree algorithms move *down* the tree.
If upward movement needed, add a pointer to parent.



## ❖ ... Representing BSTs

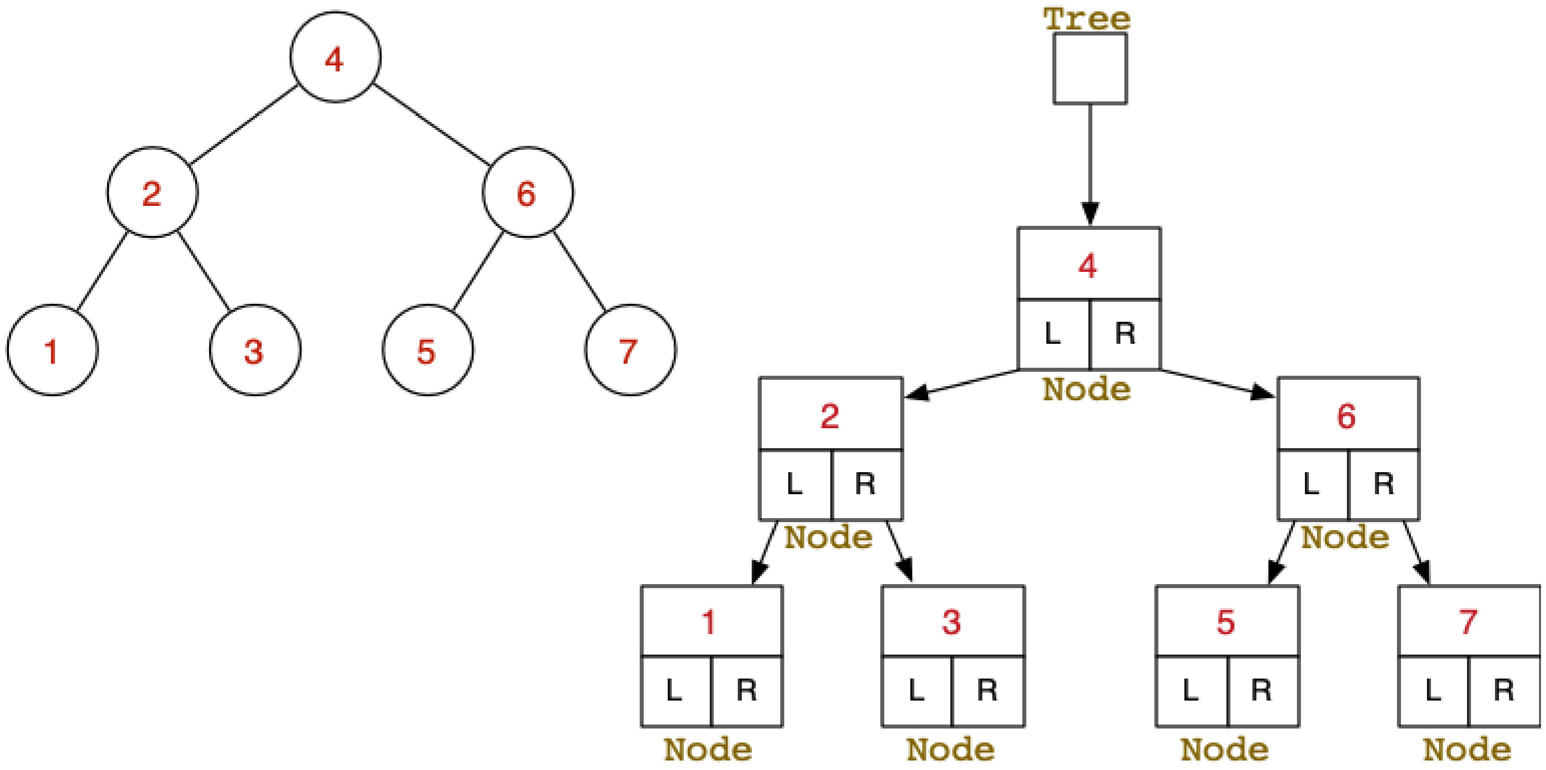
Typical data structures for trees ...

```
// a Tree is represented by a pointer to its root node
typedef struct Node *Tree;

// a Node contains its data, plus left and right subtrees
typedef struct Node {
   int  data;
   Tree left, right;
} Node;

// some macros that we will use frequently
#define data(node)  ((node)->data)
#define left(node)  ((node)->left)
#define right(node) ((node)->right)
```

Here we use a simple definition for **`data`** ... just a key

# ❖ ... Representing BSTs

Abstract data vs concrete data ...



# ❖ Searching in BSTs

Most tree algorithms are best described recursively:

```
TreeContains(tree,key):
|  Input  tree, key
|  Output true if key found in tree, false otherwise
|
|  if tree is empty then
|     return false
|  else if key < data(tree) then
|     return TreeContains(left(tree),key)
|  else if key > data(tree) then
|     return TreeContains(right(tree),key)
|  else             // found
|    return true
|  end if
```

## ❖ Insertion into BSTs

Insert an item into a tree; item becomes new leaf node

```
TreeInsert(tree,item):
|  Input  tree, item
|  Output tree with item inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item < data(tree) then
|     left(tree) = TreeInsert(left(tree),item)
|     return tree
|  else if item > data(tree) then
|     right(tree) = TreeInsert(right(tree),item)
|     return tree
|  else
|     return tree    // avoid duplicates
|  end if
```

## ❖ Tree Traversal

Iteration (traversal) on ...

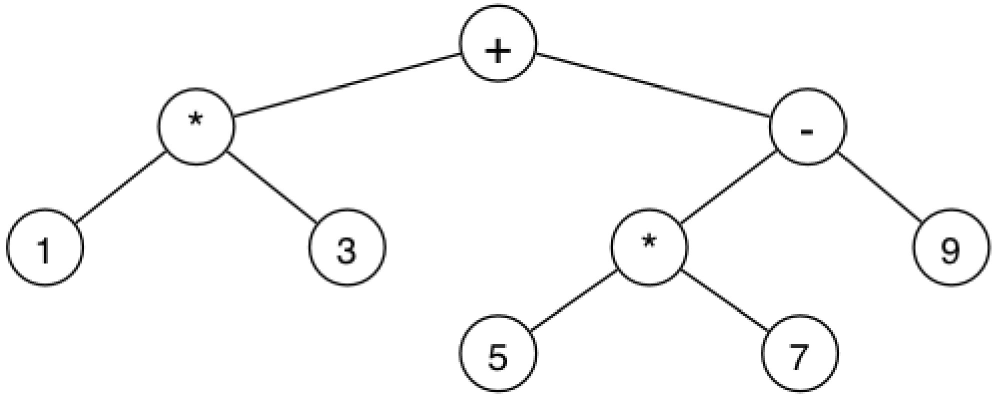
- `Lists` ... visit each value, from first to last
- `Graphs` ... visit each vertex, order determined by DFS/BFS/...

For binary `Trees`, several well-defined visiting orders exist:

- preorder (NLR) ... visit root, then left subtree, then right subtree
- inorder (LNR) ... visit left subtree, then root, then right subtree
- postorder (LRN) ... visit left subtree, then right subtree, then root
- level-order ... visit root, then all its children, then all their children

# ❖ ... Tree Traversal

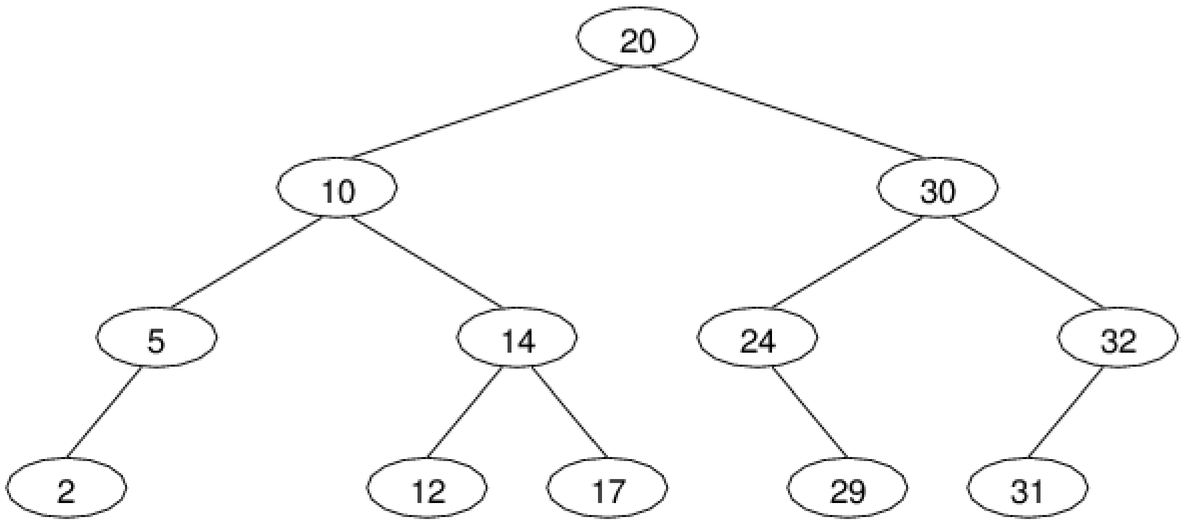
Consider "visiting" an expression tree like:




NLR: + * 1 3 - * 5 7 9 (prefix-order: useful for building tree)
LNR: 1 * 3 + 5 * 7 - 9 (infix-order: "natural" order)
LRN: 1 3 * 5 7 * 9 - + (postfix-order: useful for evaluation)
Level: + * - 1 3 * 9 5 7 (level-order: useful for printing tree)

## ❖ ... Tree Traversal

Traversals for the following tree:




NLR (preorder): 20 10 5 2 14 12 17 30 24 29 32 31

LNR (inorder): 2 5 10 12 14 17 20 24 29 30 31 32

LRN (postorder): 2 5 12 17 14 10 29 24 31 32 30 20

## ❖ ... Tree Traversal

Pseudocode for NLR traversal

```
showBSTreePreorder(t):
|   Input tree t
|
|   if t is not empty then
|   |   print data(t)
|   |   showBSTreePreorder(left(t))
|   |   showBSTreePreorder(right(t))
|   end if
```

Recursive algorithm is very simple.

Iterative version less obvious ... requires a Stack.

## ❖ ... Tree Traversal

Pseudocode for NLR traversal (non-recursive)

```
showBSTreePreorder(t):
|  Input tree t
|
|  push t onto new stack S
|  while stack is not empty do
|  |  t=pop(S)
|  |  print data(t)
|  |  if right(t) is not empty then
|  |     push right(t) onto S
|  |  end if
|  |  if left(t) is not empty then
|  |     push left(t) onto S
|  |  end if
|  end while
```

# ❖ Joining Two Trees

An auxiliary tree operation ...

Tree operations so far have involved just one tree.

An operation on two trees: `t = TreeJoin(t1,t2)`

- Pre-conditions:
  - takes two BSTs; returns a single BST
  - max(key($t_1$)) < min(key($t_2$))
- Post-conditions:
  - result is a BST (i.e. fully ordered)
  - containing all items from $t_1$ and $t_2$

## ❖ ... Joining Two Trees

Method for performing tree-join:

- find the min node in the right subtree (**`t`**$_2$)
- replace min node by its right subtree (possibly empty)
- elevate min node to be new root of both trees

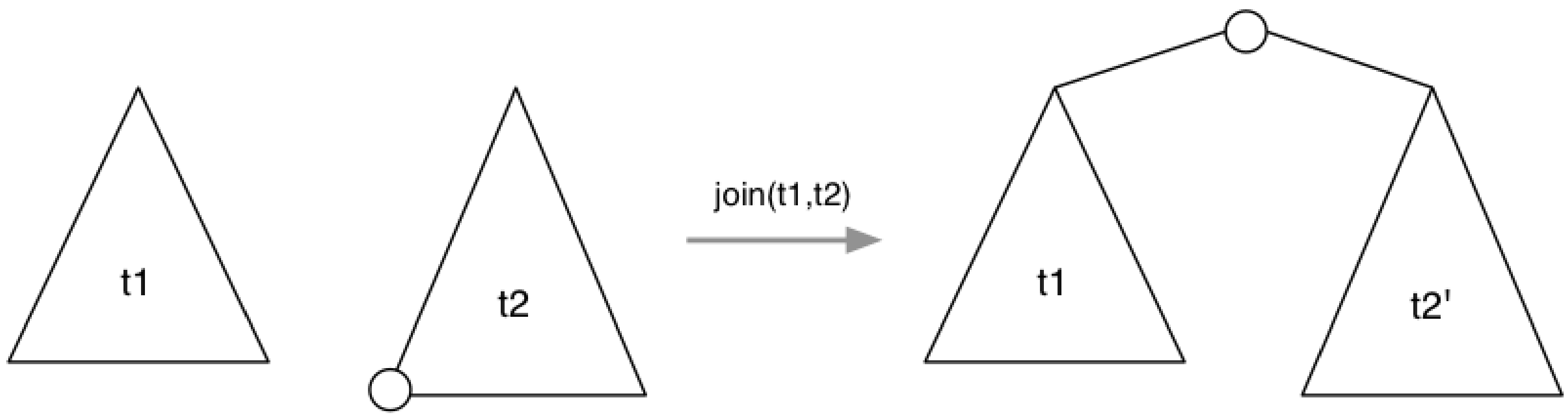
Advantage: doesn't increase height of tree significantly

$x \le \text{height}(\mathbf{t}) \le x+1$, where $x = \max(\text{height}(\mathbf{t_1}),\text{height}(\mathbf{t_2}))$

Variation: choose deeper subtree; take root from there.

## ❖ ... Joining Two Trees

Joining two trees:




Note: t2' may be less deep than t2

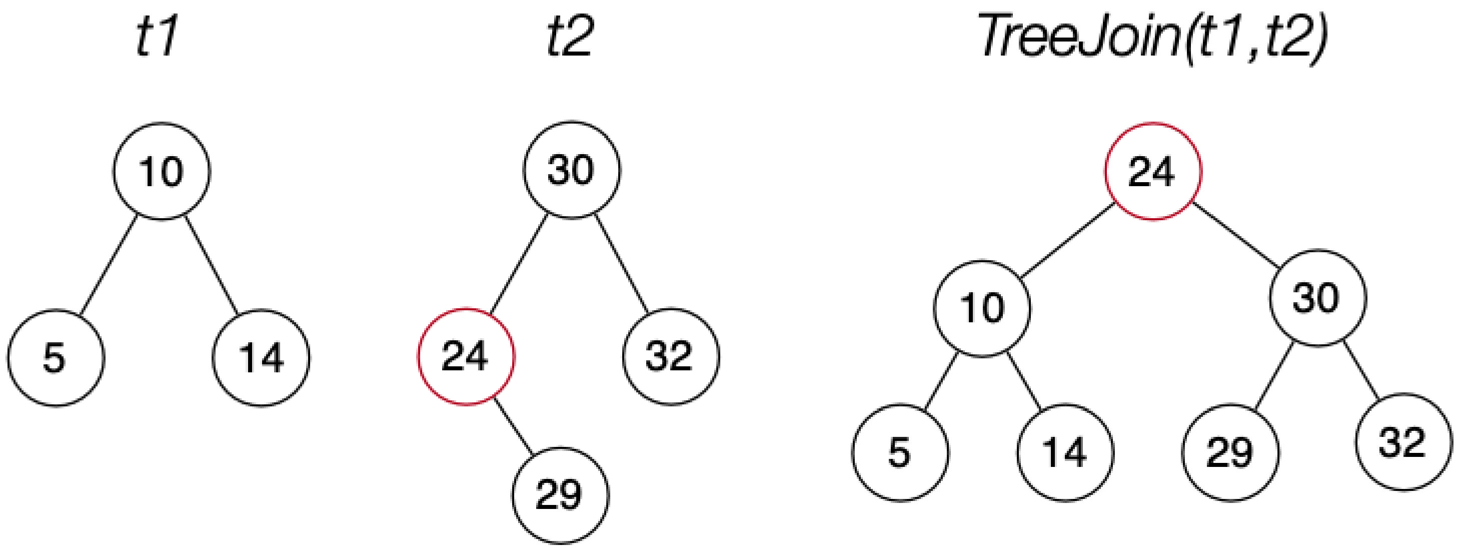
# ❖ ... Joining Two Trees

Implementation of tree-join:

```
TreeJoin(t₁,t₂):
|  Input  trees t₁,t₂
|  Output t₁ and t₂ joined together
|
|  if t₁ is empty then return t₂
|  else if t₂ is empty then return t₁
|  else
|  |  curr=t₂, parent=NULL
|  |  while left(curr) is not empty do     // find min element in t₂
|  |     parent=curr
|  |     curr=left(curr)
|  |  end while
|  |  if parent≠NULL then
|  |     left(parent)=right(curr)  // unlink min element from parent
|  |     right(curr)=t₂
|  |  end if
|  |  left(curr)=t₁
|  |  return curr                  // curr is new root
|  end if
```

# ❖ ... Joining Two Trees

Example tree join:



## ❖ Deletion from BSTs

Insertion into a binary search tree is easy.

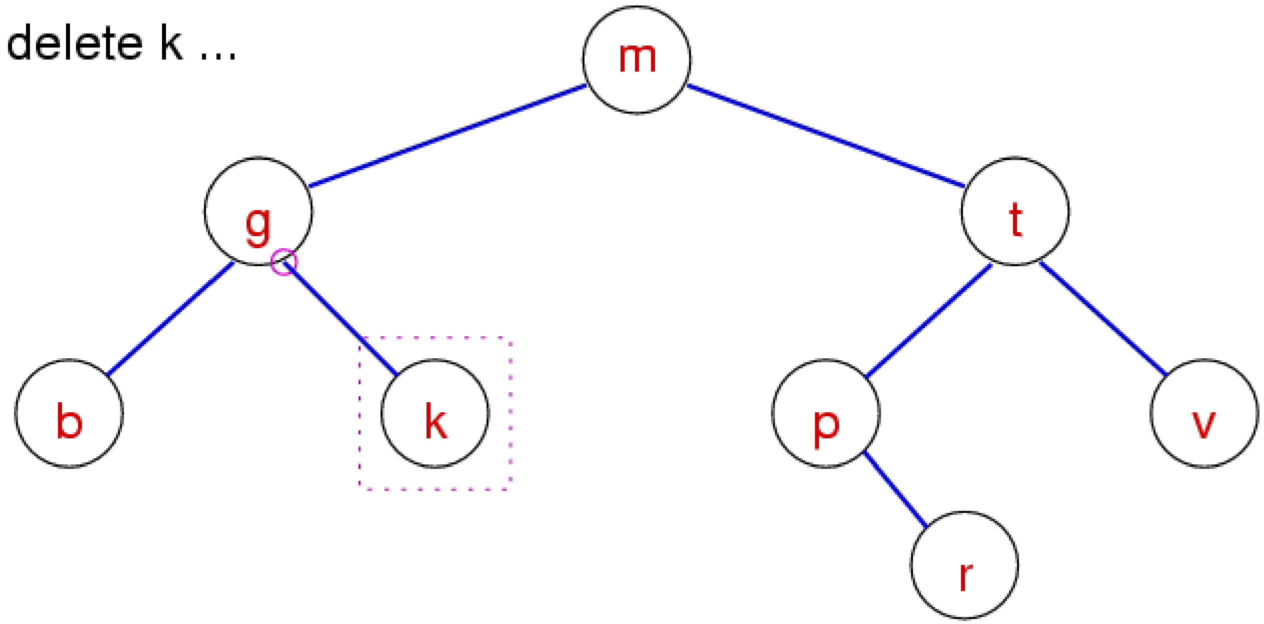
Deletion from a binary search tree is harder.

Four cases to consider …

- empty tree … new tree is also empty
- zero subtrees … unlink node from parent
- one subtree … replace by child
- two subtrees … replace by successor, join two subtrees

# ❖ ... Deletion from BSTs

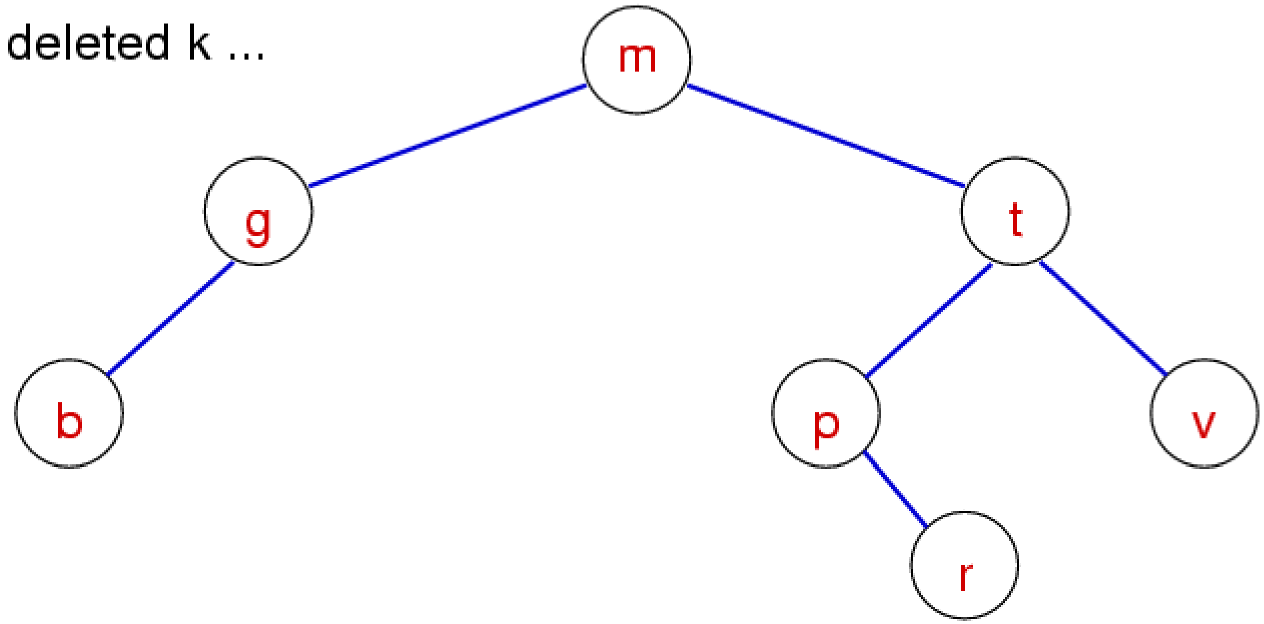
Case 2: item to be deleted is a leaf (zero subtrees)




Just delete the item

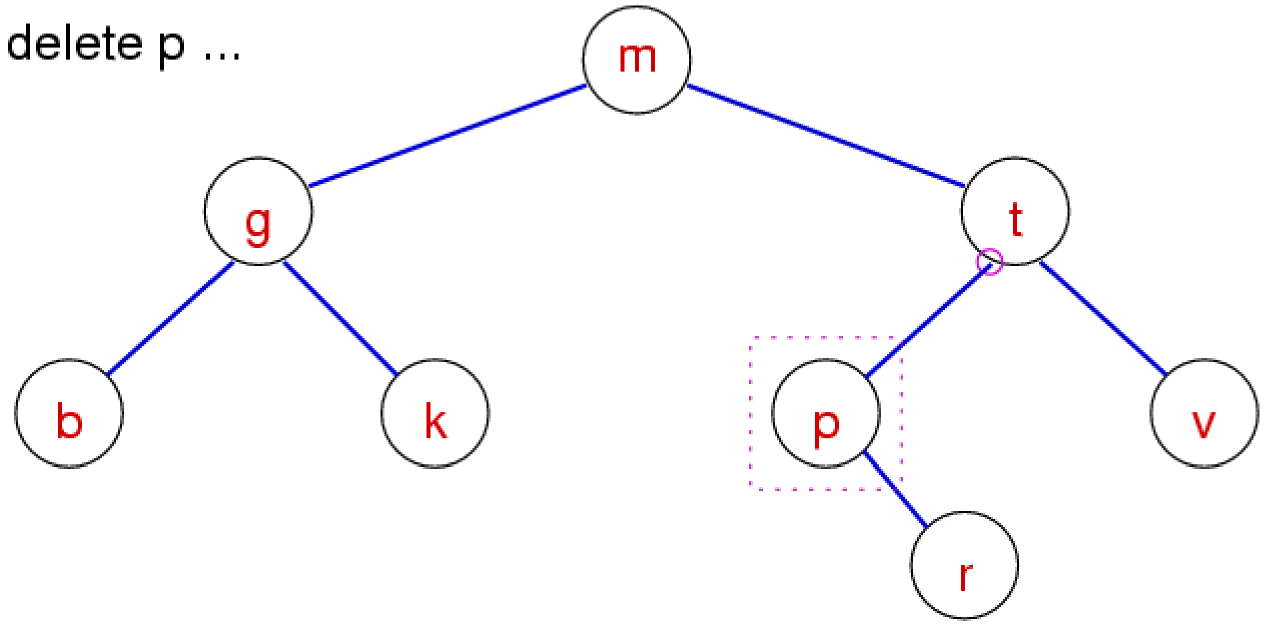
# ❖ ... Deletion from BSTs

Case 2: item to be deleted is a leaf (zero subtrees)



# ❖ ... Deletion from BSTs

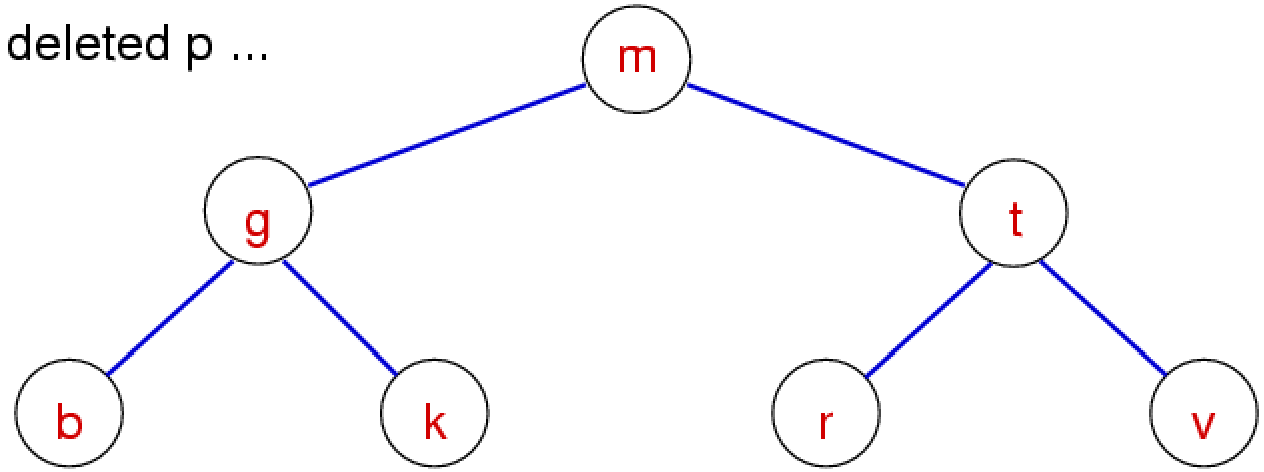
Case 3: item to be deleted has one subtree




Replace the item by its only subtree

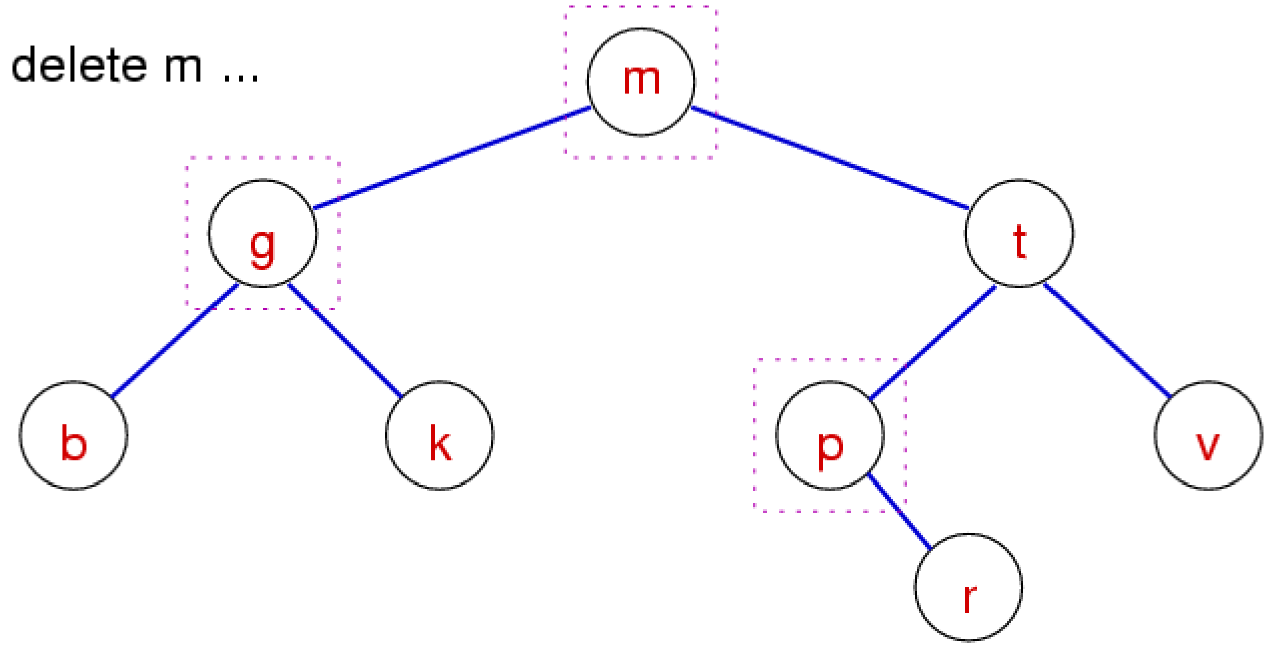
# ❖ ... Deletion from BSTs

Case 3: item to be deleted has one subtree



# ❖ ... Deletion from BSTs

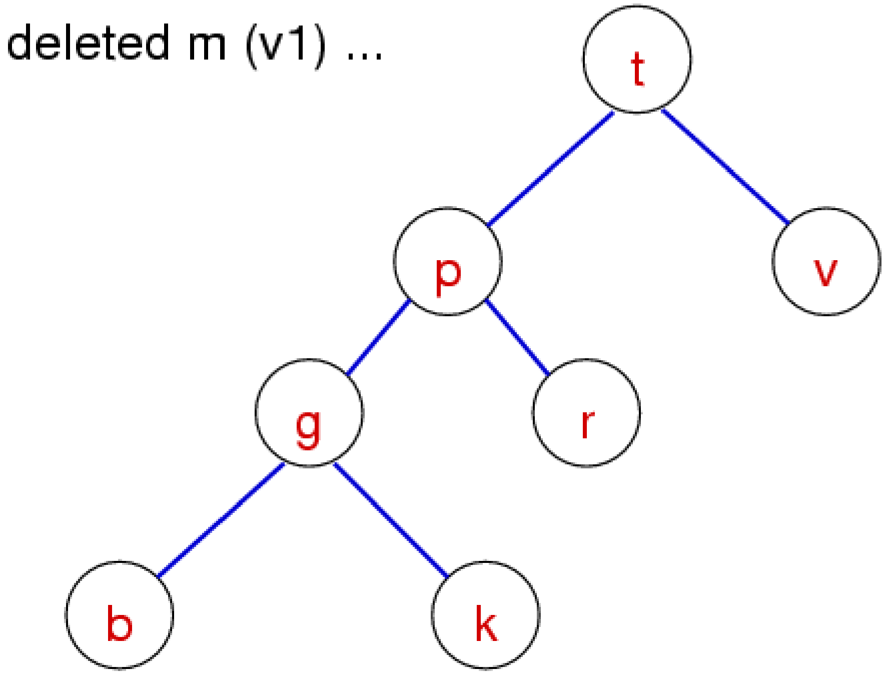
Case 4: item to be deleted has two subtrees




Version 1: right child becomes new root, attach left subtree to min element of right subtree

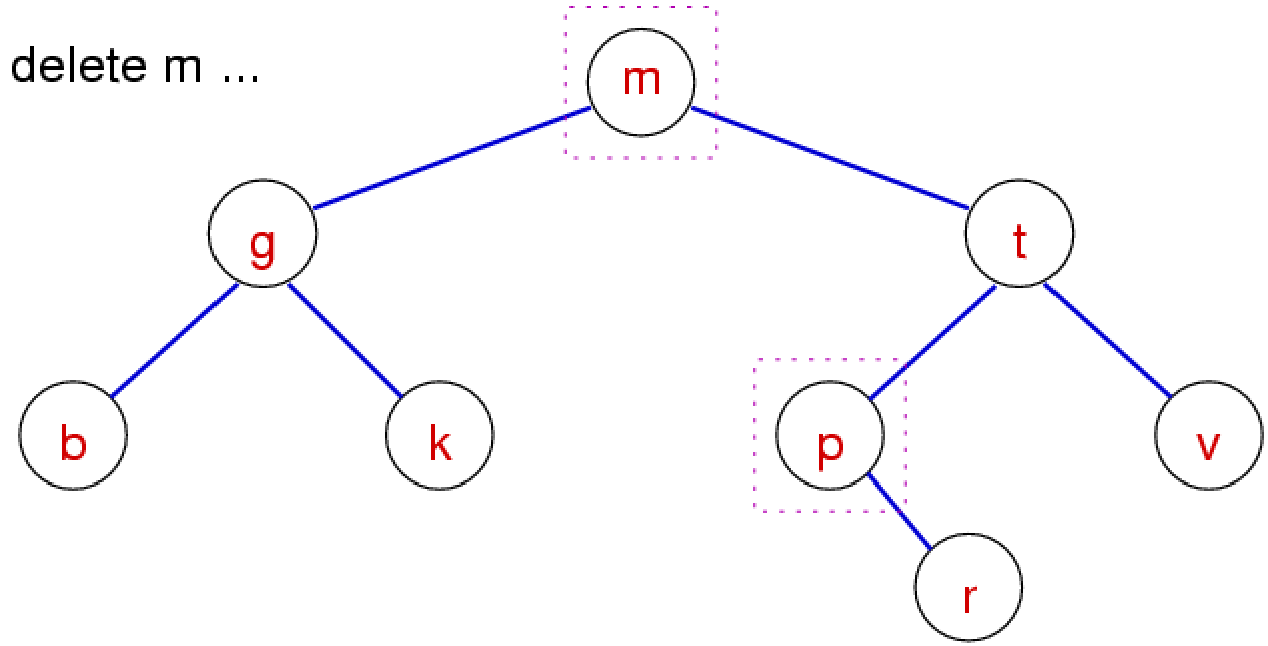
## ❖ ... Deletion from BSTs

Case 4: item to be deleted has two subtrees



# ❖ ... Deletion from BSTs

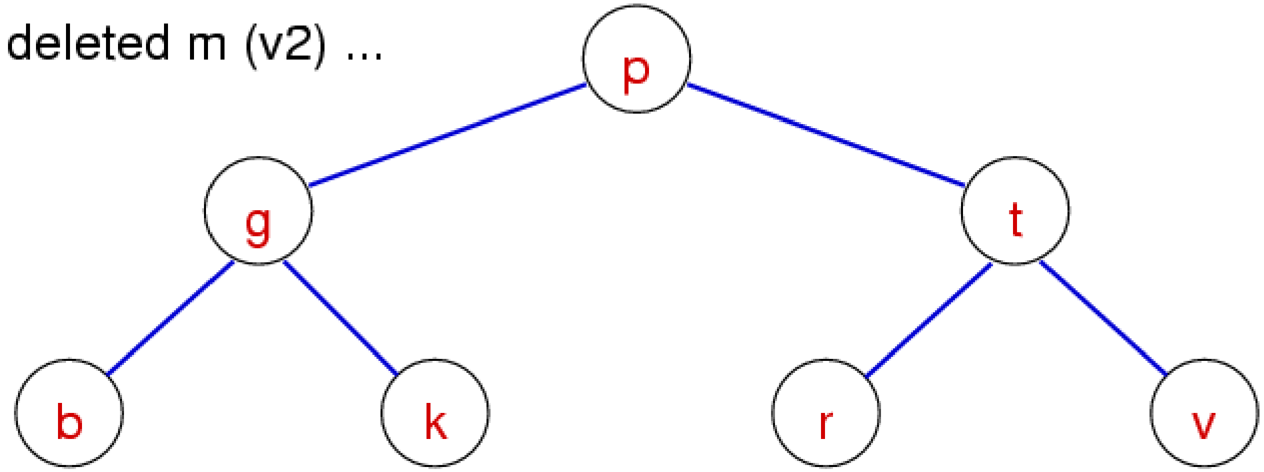
Case 4: item to be deleted has two subtrees




Version 2: *join* left and right subtree

# ❖ ... Deletion from BSTs

Case 4: item to be deleted has two subtrees



# ... Deletion from BSTs

Pseudocode (version 2):

```
TreeDelete(t,item):
|  Input  tree t, item
|  Output t with item deleted
|
|  if t is not empty then          // nothing to do if tree is empty
|  |  if item < data(t) then       // delete item in left subtree
|  |     left(t)=TreeDelete(left(t),item)
|  |  else if item > data(t) then  // delete item in left subtree
|  |     right(t)=TreeDelete(right(t),item)
|  |  else                         // node 't' must be deleted
|  |  |  if left(t) and right(t) are empty then
|  |  |     new=empty tree                   // 0 children
|  |  |  else if left(t) is empty then
|  |  |     new=right(t)                     // 1 child
|  |  |  else if right(t) is empty then
|  |  |     new=left(t)                      // 1 child
|  |  |  else
|  |  |     new=TreeJoin(left(t),right(t))  // 2 children
|  |  |  end if
|  |  |  free memory allocated for t
|  |  |  t=new
|  |  end if
|  end if
|  return t
```

Produced: 7 Jun 2020