Week 10

Things to Note …

- Mid-semester test result
- Assignment 2 under way

In This Lecture …

- Search Tree Algorithms ([S] Ch.12.5-12.6,12.8-12.9)

Coming Up …

- Balanced Search Trees ([S] Ch.13)

Mid-semester Test Statistics

21.25 – 25.0	55
18.75 – 21.0	60
16.25 – 18.5	43
12.5 – 16.0	48
$<$ 12.5	21

Final exam: Thursday, 9 November, 8:45am (2 hours)

Assignment 2 Tips

Mandatory style requirements:

- structured code (no **`break/continue`**)
- good commenting (must explain your code)

Test your program thoroughly

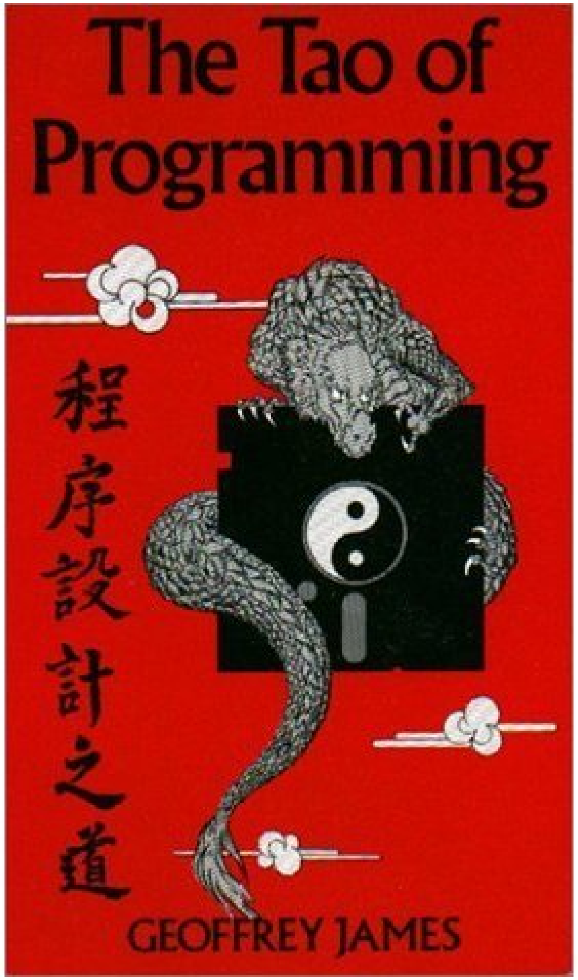
- Just passing the **`dryrun`**-tests is not enough
- Try to break your program with your own test cases
  - can use arbitrary strings, of course (e.g. **`abc, acc, cbc, cc,`** …)

Do not forget to add time complexity *O(…)* for task 1 and *O(…)* for task 2, *not* for every function

- with explanation

The Tao of Programming

Next in a series of advices from the Tao of Programming …



Thus spake the Master Programmer:

"When a program is being tested, it is too late to make design changes."

The Tao of Programming (cont)

Book 4
Chapter 4.1

A program should be light and agile, its subroutines connected like a string of pearls.

The spirit and intent of the program should be retained throughout.

There should be neither too little nor too much. Neither needless loops nor useless variables; neither lack of structure nor overwhelming rigidity.

If the program fails in these requirements, it will be in a state of disorder and confusion. The only way to correct this is to rewrite the program.

Searching

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 (cont)

Since searching is a very important/frequent operation, many approaches have been developed to do it

Linear structures: arrays, linked lists, files

Arrays = random access. Lists, files = sequential access.

Cost of searching:

	Array	List	File
Unsorted	O(n) (linear scan)	O(n) (linear scan)	O(n) (linear scan)
Sorted	O(log n) (binary search)	O(n) (linear scan)	O(log n) (*seek, seek>/$>, …*)

- *O(n)* … linear scan (search technique of last resort)
- *O(log n)* … binary search, search trees (trees also have other uses)

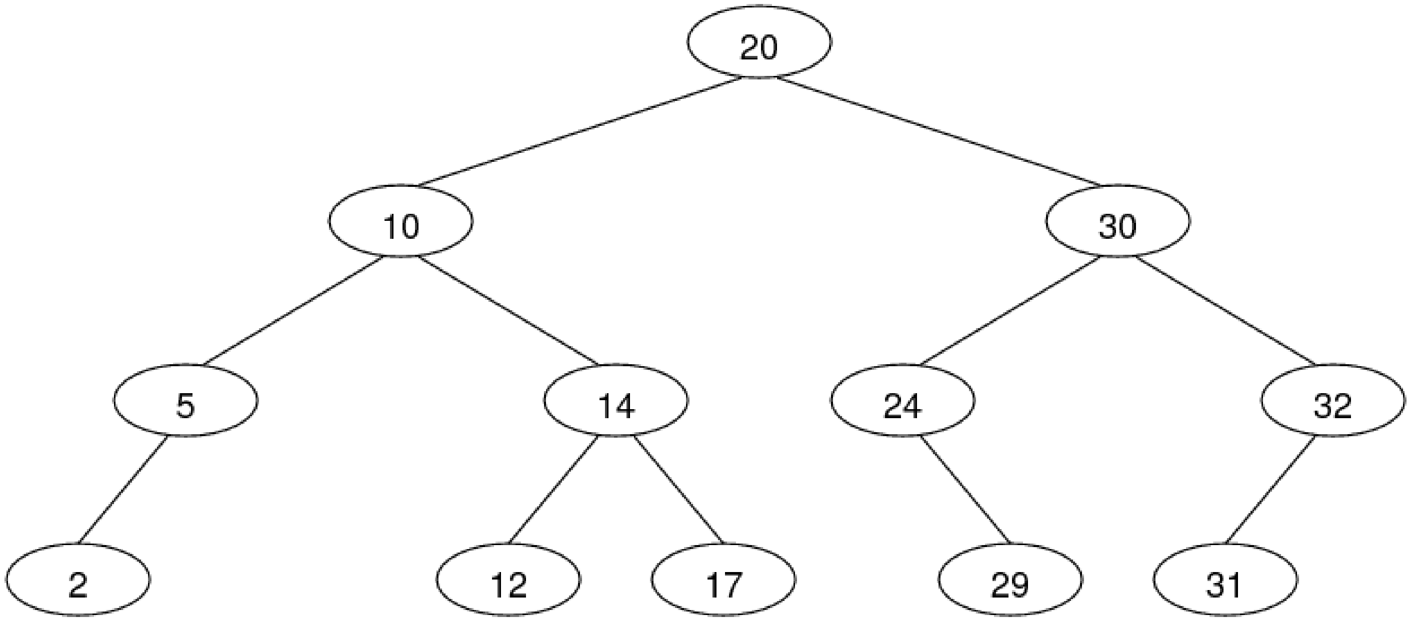
Also (cf. COMP9021): hash tables (*O(1)*, but only under optimal conditions)

Searching (cont)

Maintaining the order in sorted arrays and files is a costly operation.

Search trees are as efficient to search but more efficient to maintain.

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

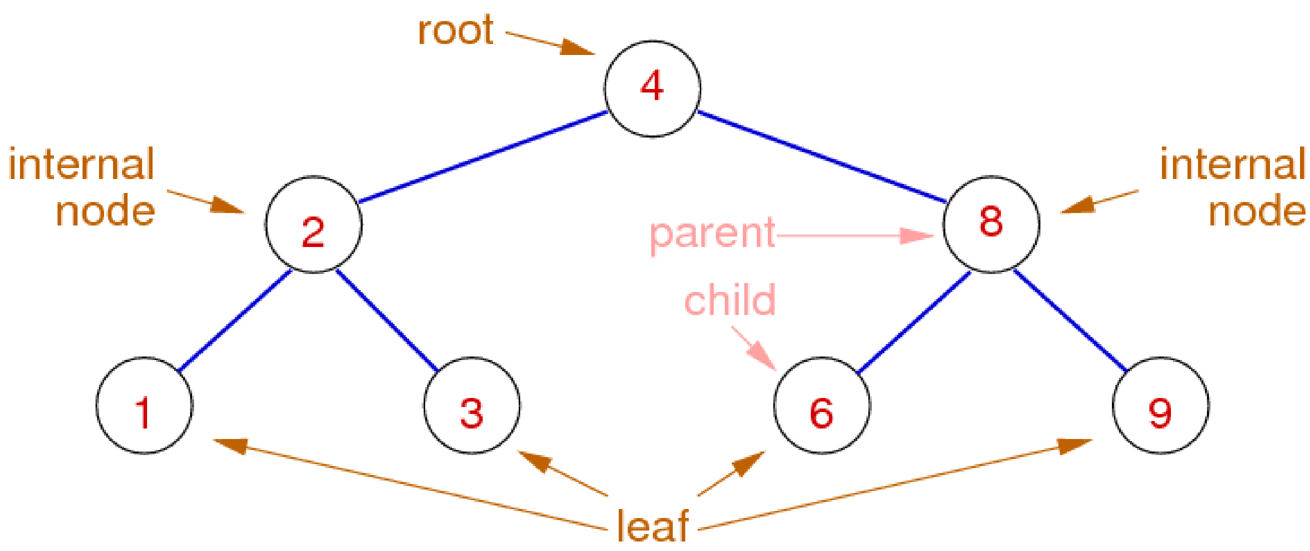


Tree Data Structures

Trees

Trees are connected graphs

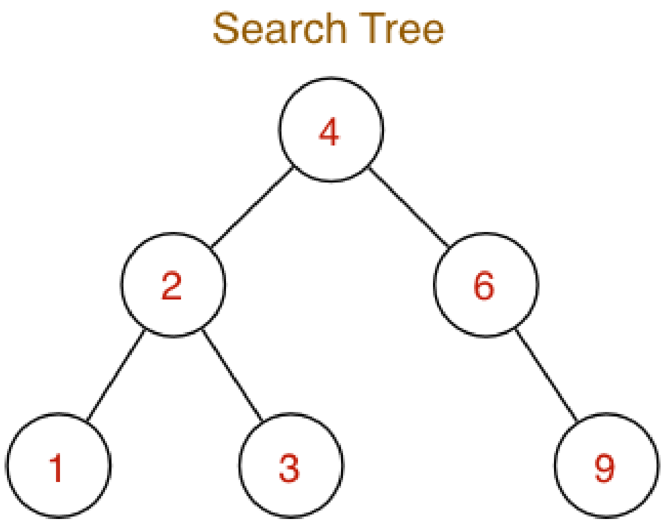
- consisting of nodes and edges (called *links*), with no cycles (no "up-links")
- each node contains a data value (or key+data)
- each node has links to ≤ *k* other child nodes (*k*=2 below)

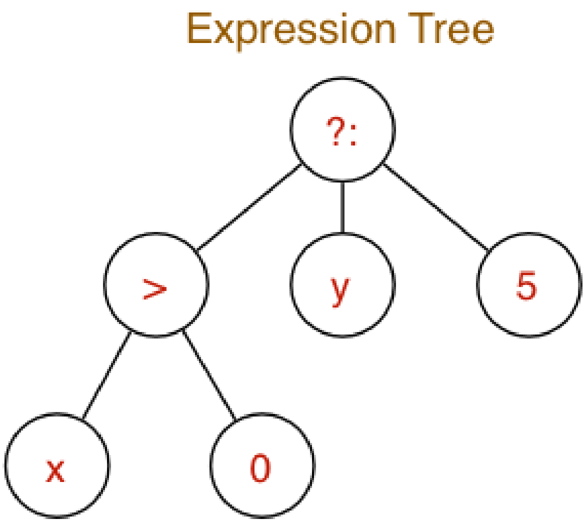


Trees (cont)

Trees are used in many contexts, e.g.

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

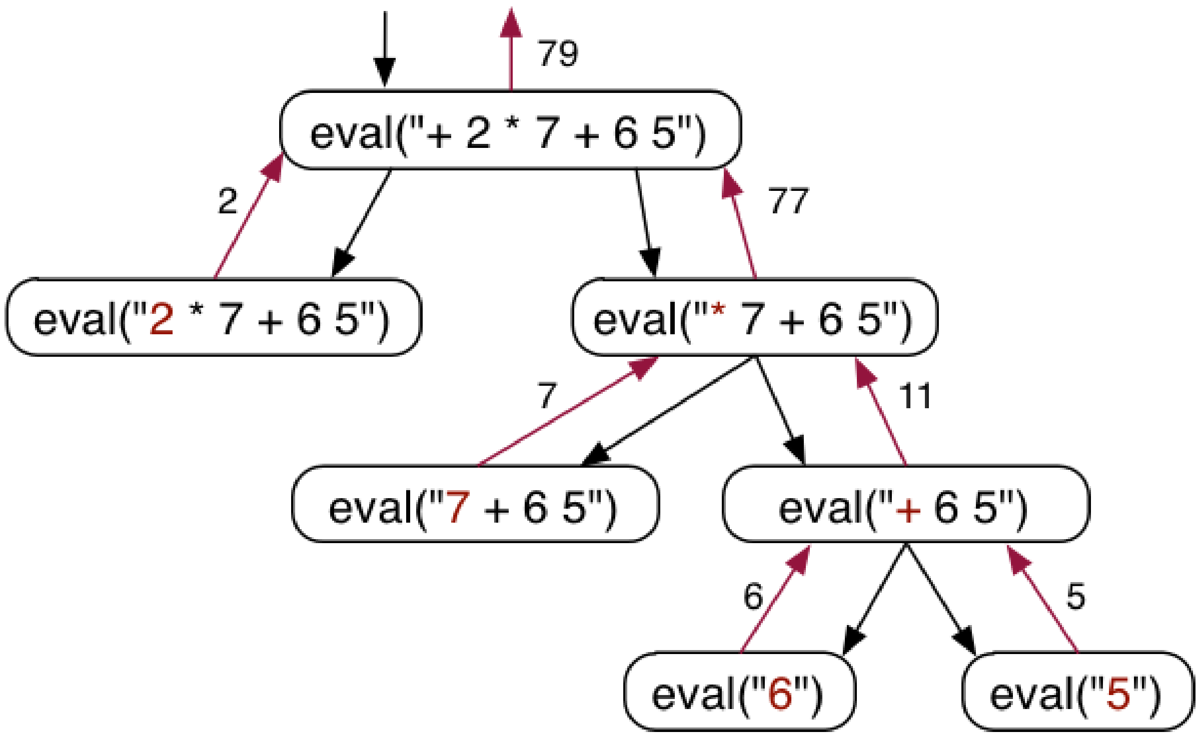




Trees (cont)

Trees can be used as a data structure, but also for illustration.

E.g. showing evaluation of a prefix arithmetic expression

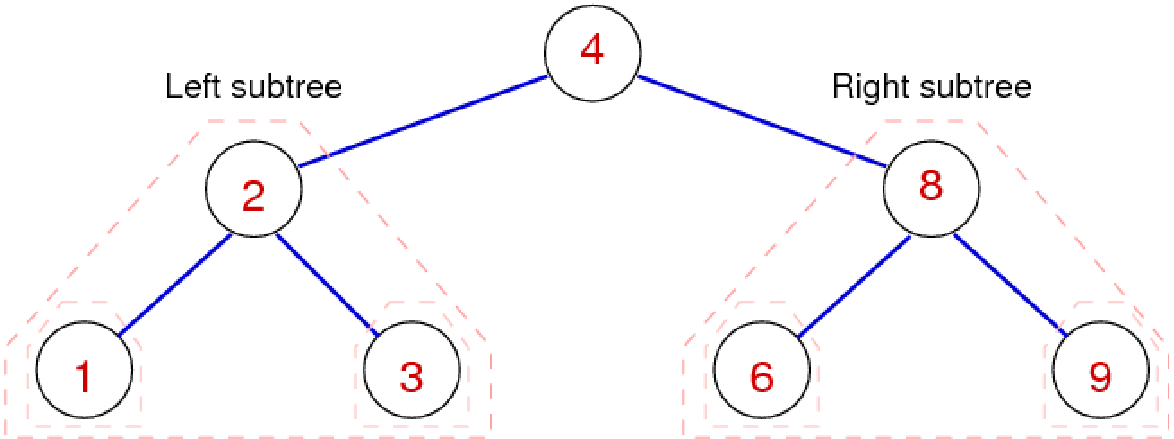


Trees (cont)

Binary trees (*k=2* children per node) can be defined recursively, as follows:

A *binary tree* is either

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



Trees (cont)

Other special kinds of tree

- *m*-ary tree: each internal node has exactly *m* children
- 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

Search Trees

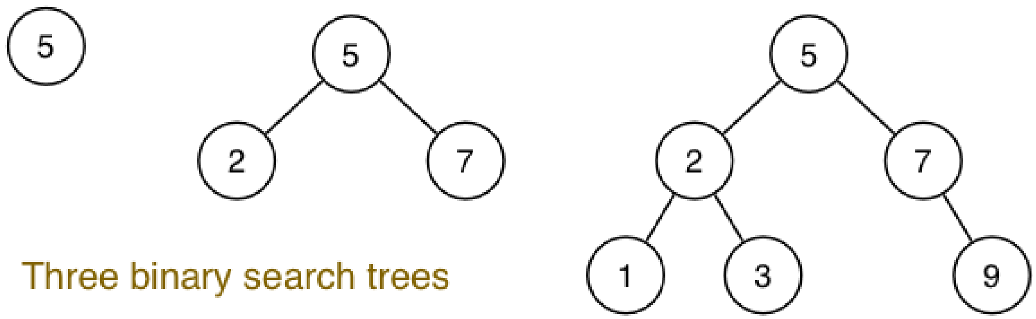
Binary Search Trees

Binary search trees (or BSTs) have the characteristic properties

- 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

(perfectly) balanced trees have the properties

- #nodes in left subtree = #nodes in right subtree
- this property applies over all nodes in the tree



Three binary search trees

Binary Search Trees (cont)

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(), …

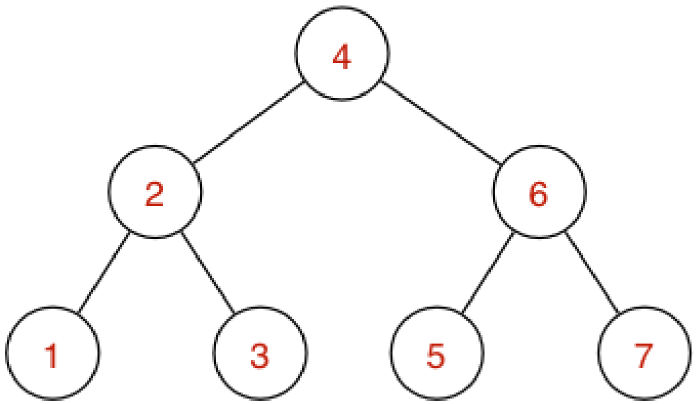
Notes:

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

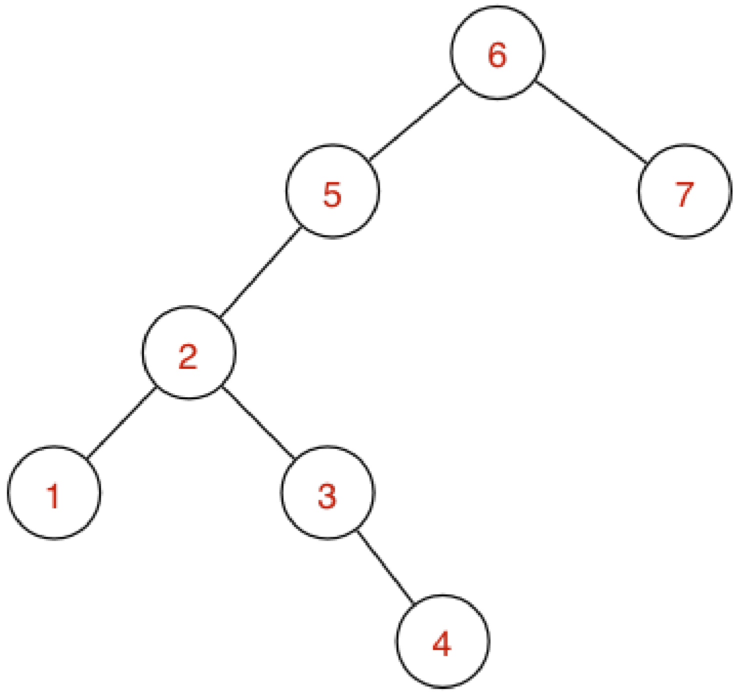
Binary Search Trees (cont)

Examples of binary search trees:

Balanced Tree



Non-balanced Tree

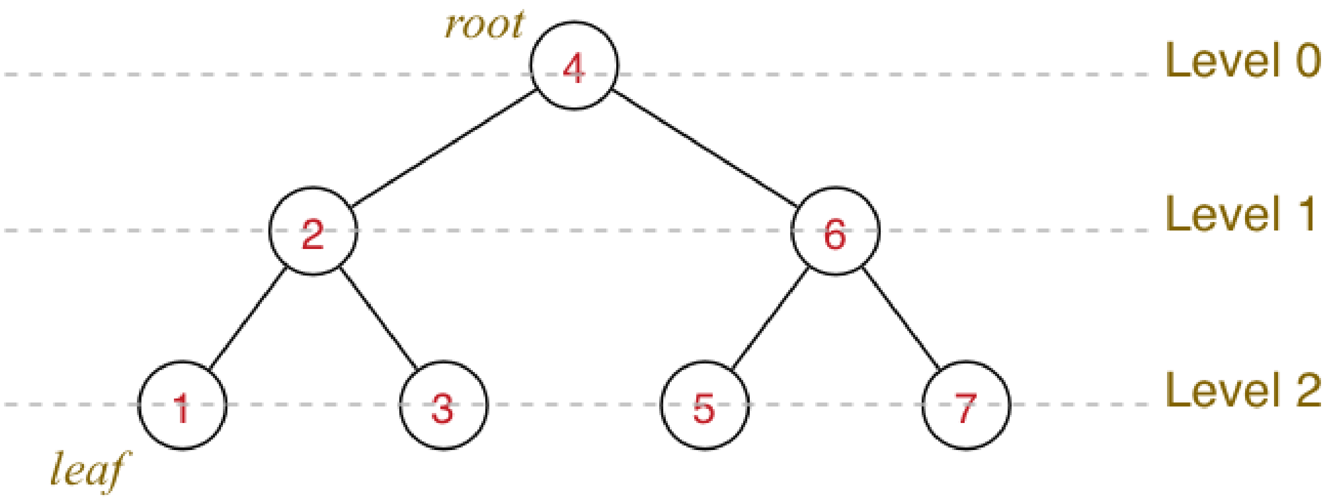


Shape of tree is determined by order of insertion.

Binary Search Trees (cont)

Level of node = path length from root to node

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



Height-balanced tree: ∀ nodes: height(left subtree) = height(right subtree)

Time complexity of tree algorithms is typically *O(height)*

Exercise #1: Insertion into BSTs

For each of the sequences below

- start from an initially empty binary search tree
- show tree resulting from inserting values in order given

(a) 4 2 6 5 1 7 3

(b) 6 5 2 3 4 7 1

(c) 1 2 3 4 5 6 7

Assume new values are always inserted as new leaf nodes

(a) the balanced tree from 3 slides ago (height = 2)

(b) the non-balanced tree from 3 slides ago (height = 4)

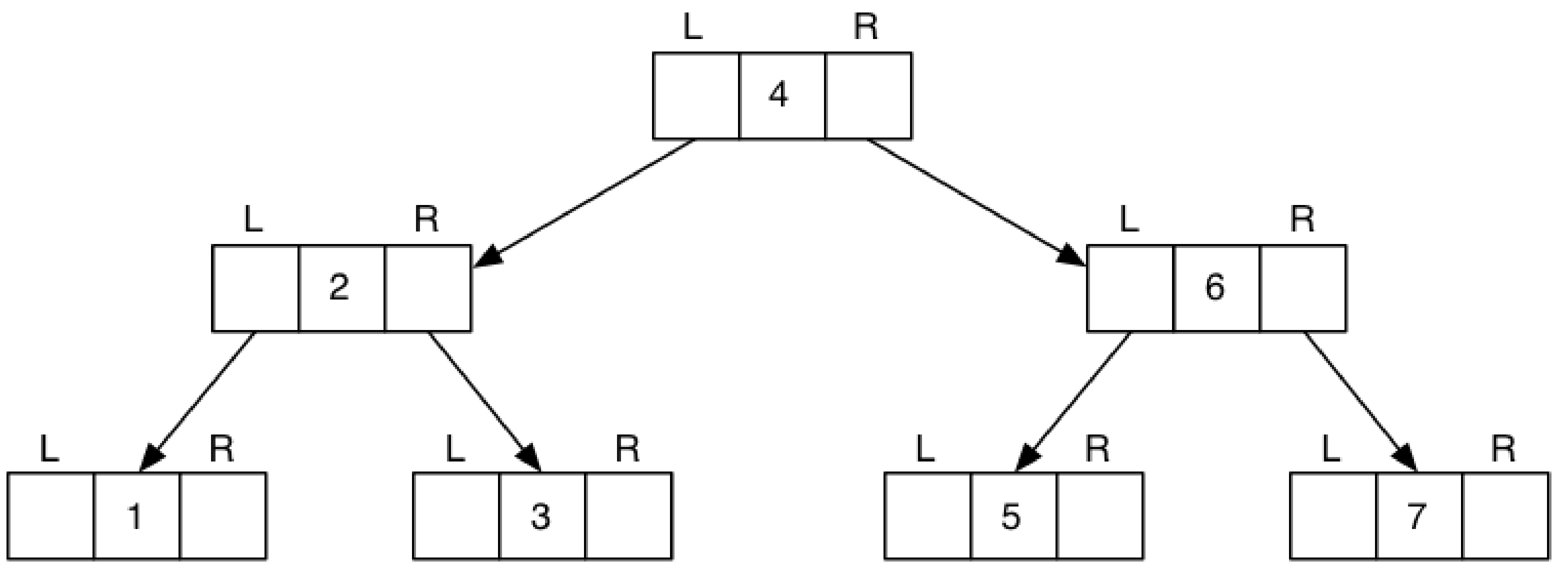
(c) a fully degenerate tree of height 6

Representing BSTs

Binary trees are typically represented by node structures

- containing 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 (cont)

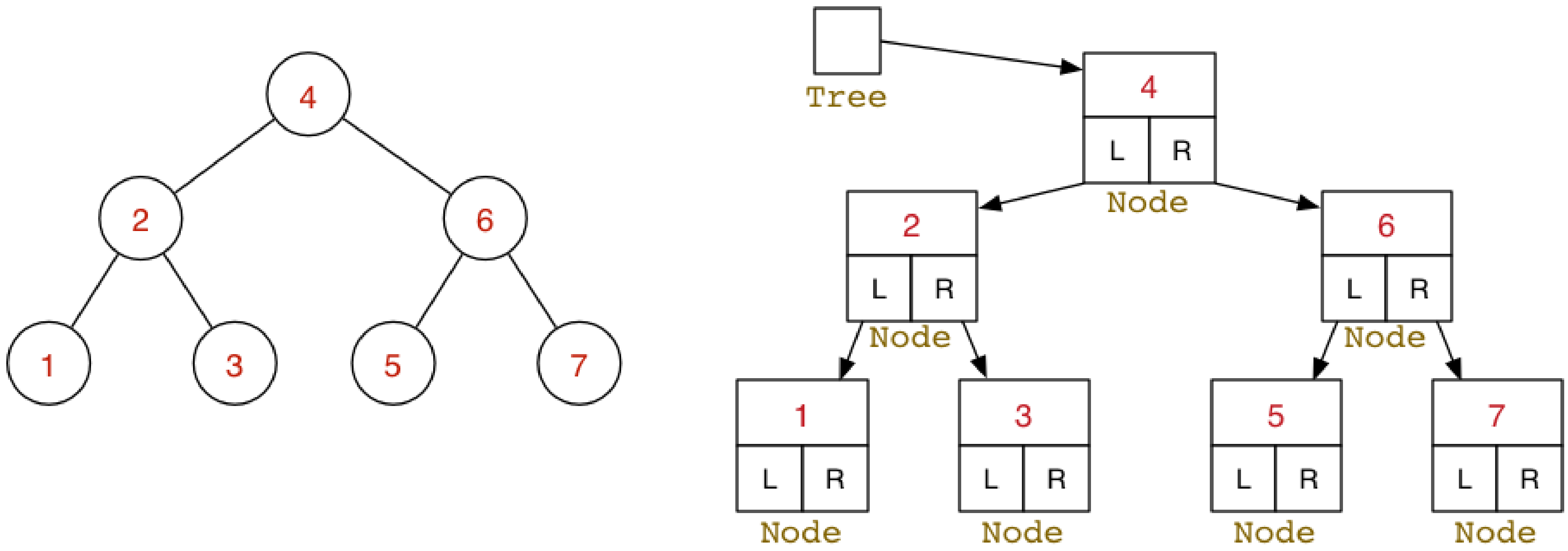
Typical data structures for trees …

```
// 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(tree)  ((tree)->data)
#define left(tree)  ((tree)->left)
#define right(tree) ((tree)->right)
```

We ignore items ⇒ **`data`** in **`Node`** is just a key

Representing BSTs (cont)

Abstract data vs concrete data …



Tree Algorithms

Searching in BSTs

Most tree algorithms are best described recursively:

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

Insertion into BSTs

Insert an item into appropriate subtree:

```
insertAtLeaf(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
      return insertAtLeaf(left(tree),item)
   else if item > data(tree) then
      return insertAtLeaf(right(tree),item)
   else
      return tree     // avoid duplicates
   end if
```

Tree Traversal

Iteration (traversal) on …

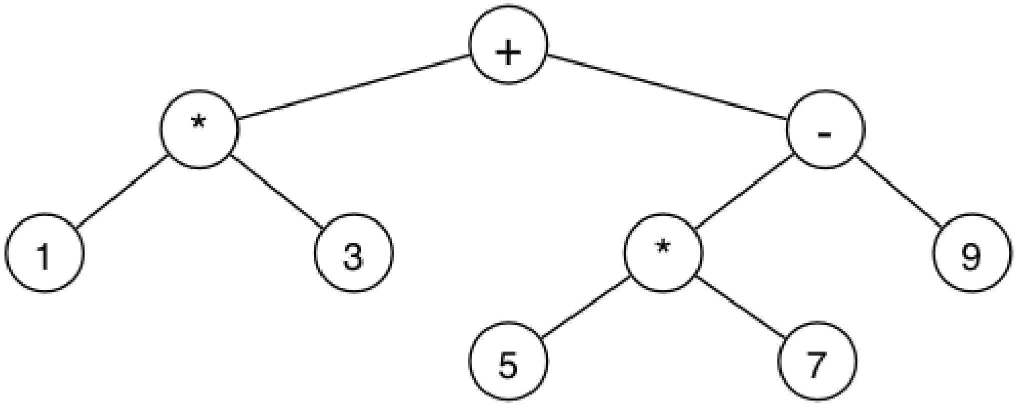
- `List`s … visit each value, from first to last
- `Graph`s … visit each vertex, order determined by DFS/BFS/…

For binary `Tree`s, 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 (cont)

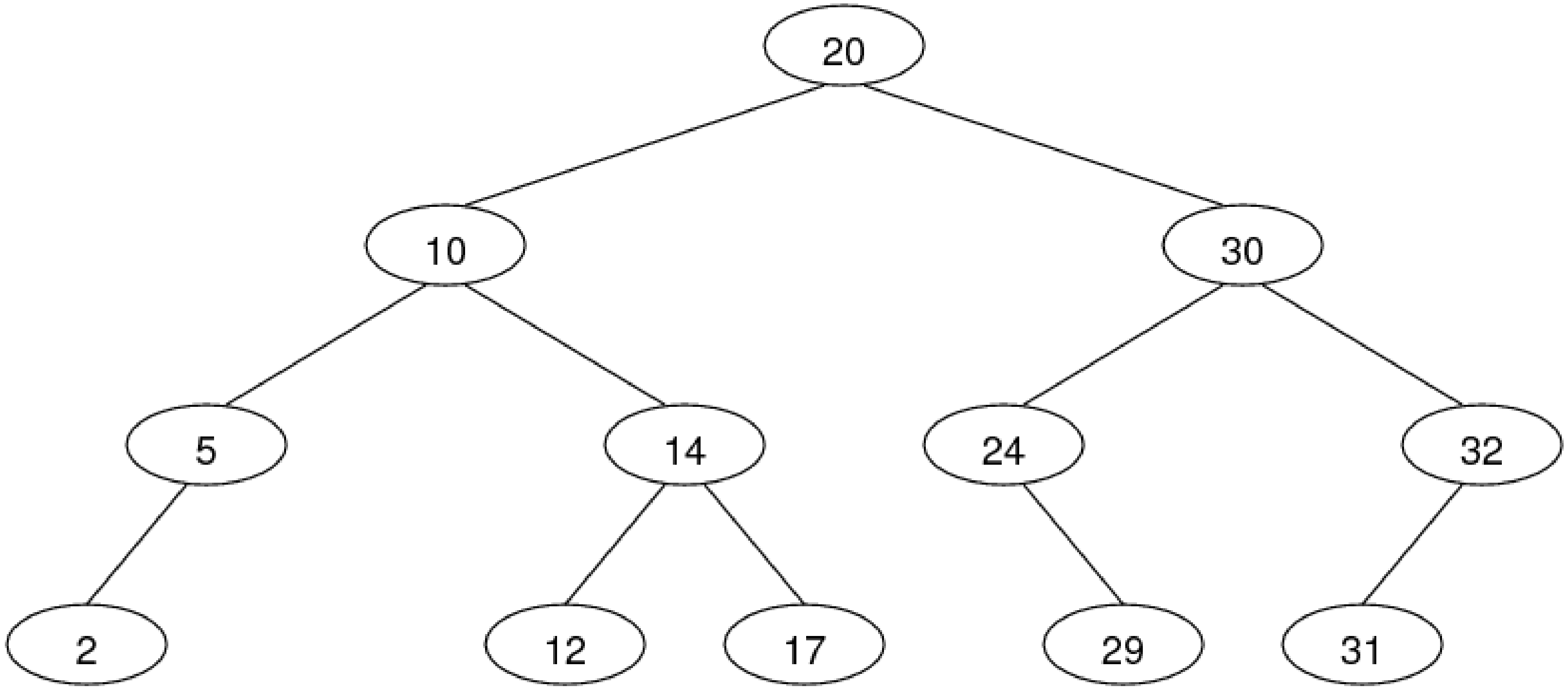
Consider "visiting" an expression tree like:



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)

Exercise #2: Tree Traversal

Show NLR, LNR, LRN 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

Exercise #3: Non-recursive traversals

Write a non-recursive *preorder* traversal algorithm.

Assume that you have a stack ADT available.

```
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: $\mathtt{t = joinTrees(t_1,t_2)}$

- 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 (cont)

Method for performing tree-join:

- find the min node in the right subtree ($\mathbf{t_2}$)
- replace min node by its right subtree
- elevate min node to be new root of both trees

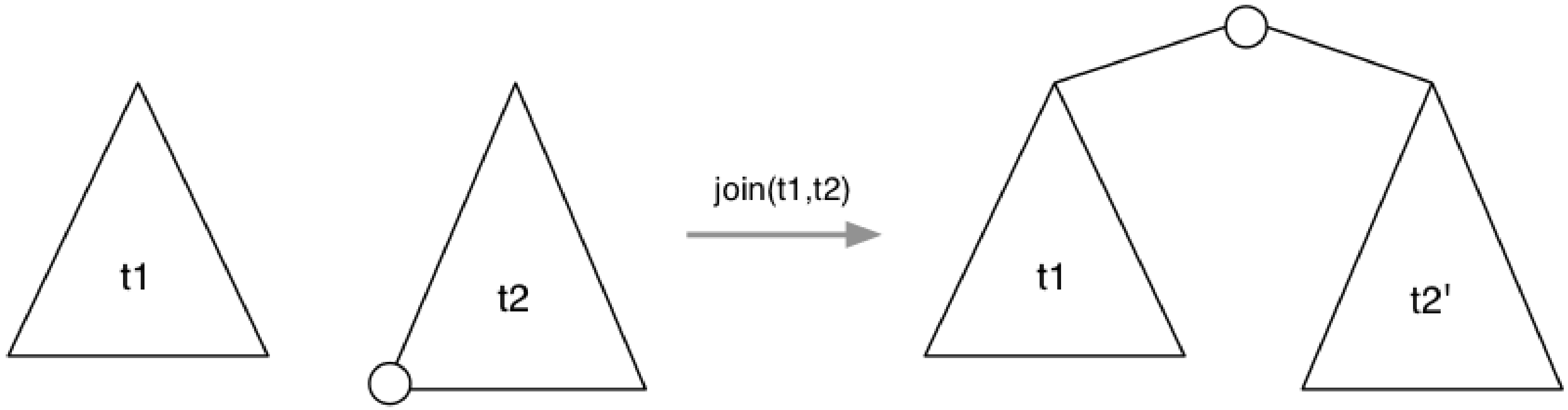
Advantage: doesn't increase height of tree significantly

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

Variation: choose deeper subtree; take root from there.

Joining Two Trees (cont)

Joining two trees:



Note: t2' may be less deep than t2

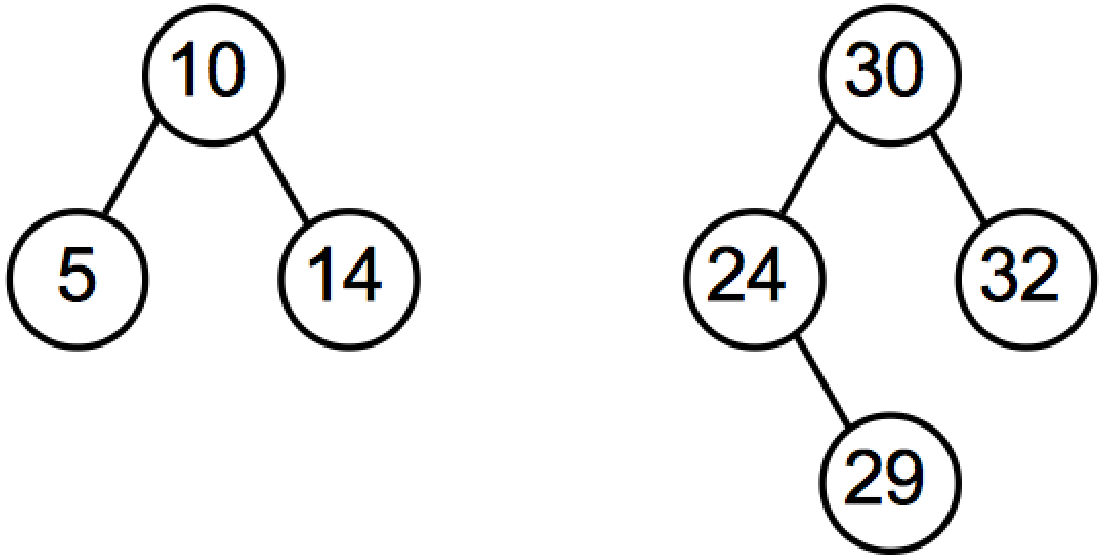
Joining Two Trees (cont)

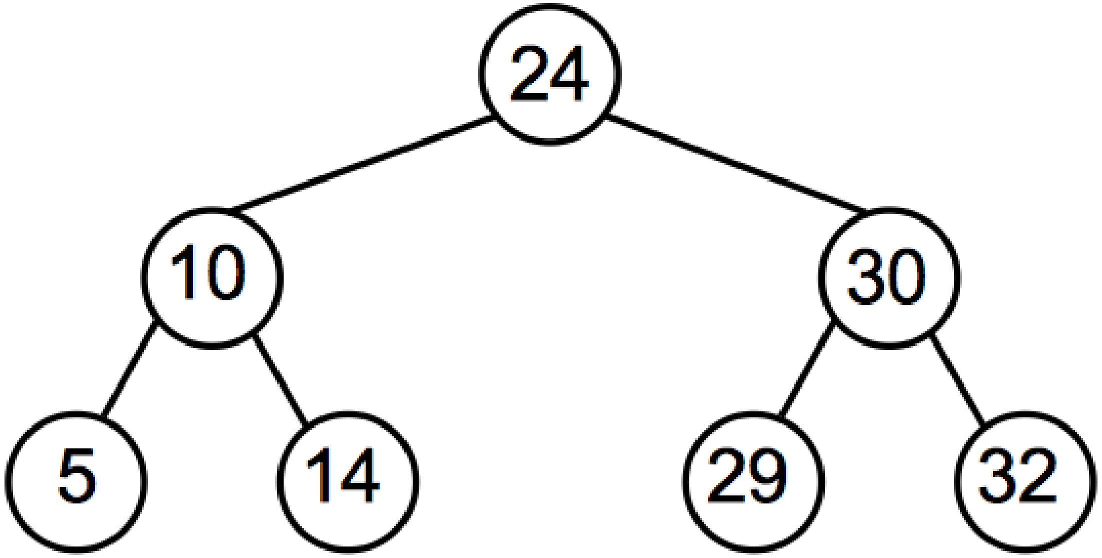
Implementation of tree-join:

```
joinTrees(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
```

Exercise #4: Joining Two Trees

Join the trees




24
10
30
5
14
29
32

Deletion from BSTs

Insertion into a binary search tree is easy.

Deletion from a binary search tree is harder.

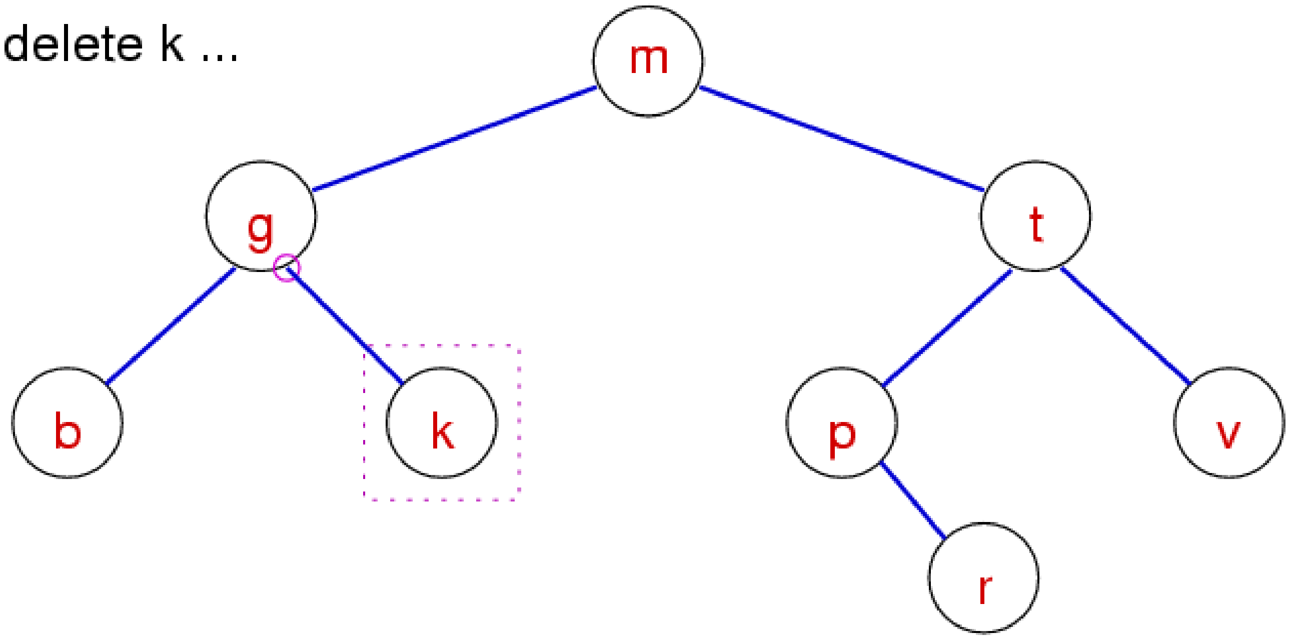
Four cases to consider …

- 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 (cont)

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

delete k ...

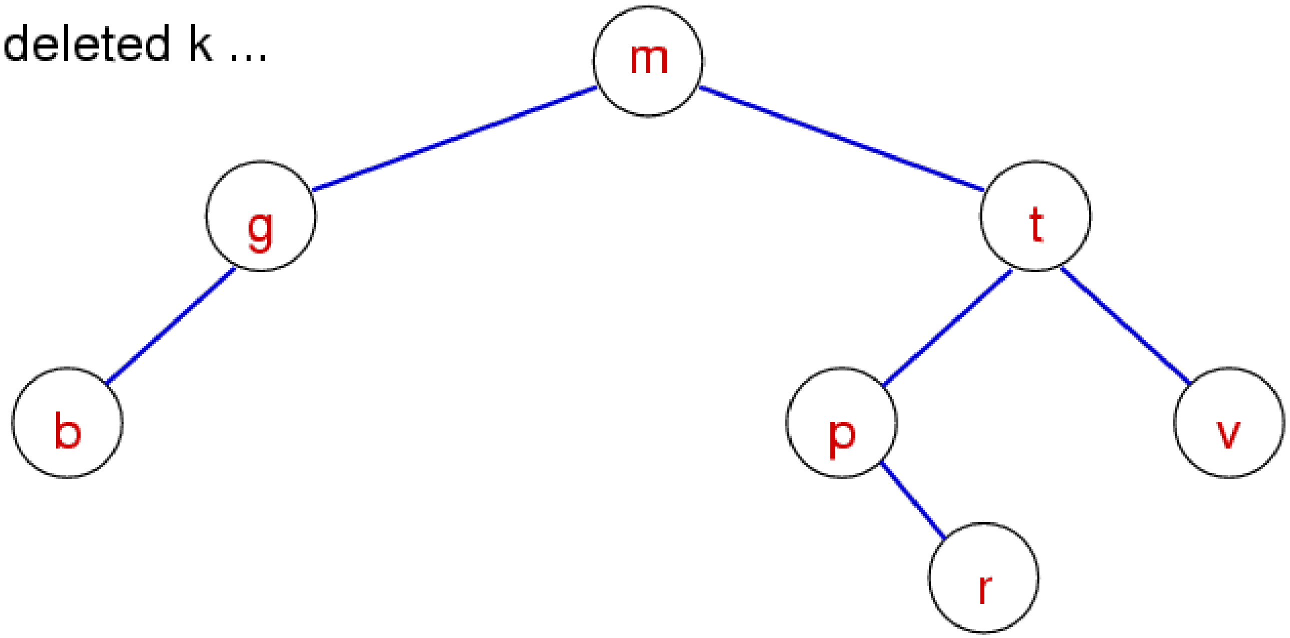


Just delete the item

Deletion from BSTs (cont)

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

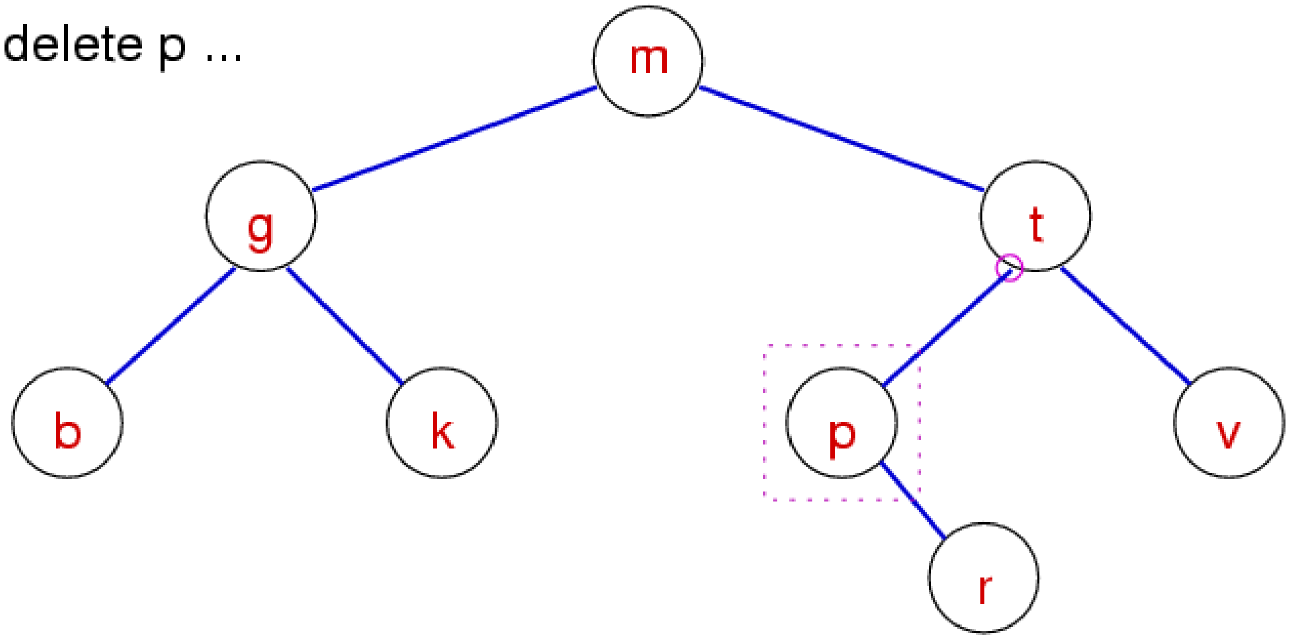
deleted k ...



Deletion from BSTs (cont)

Case 3: item to be deleted has one subtree

delete p ...

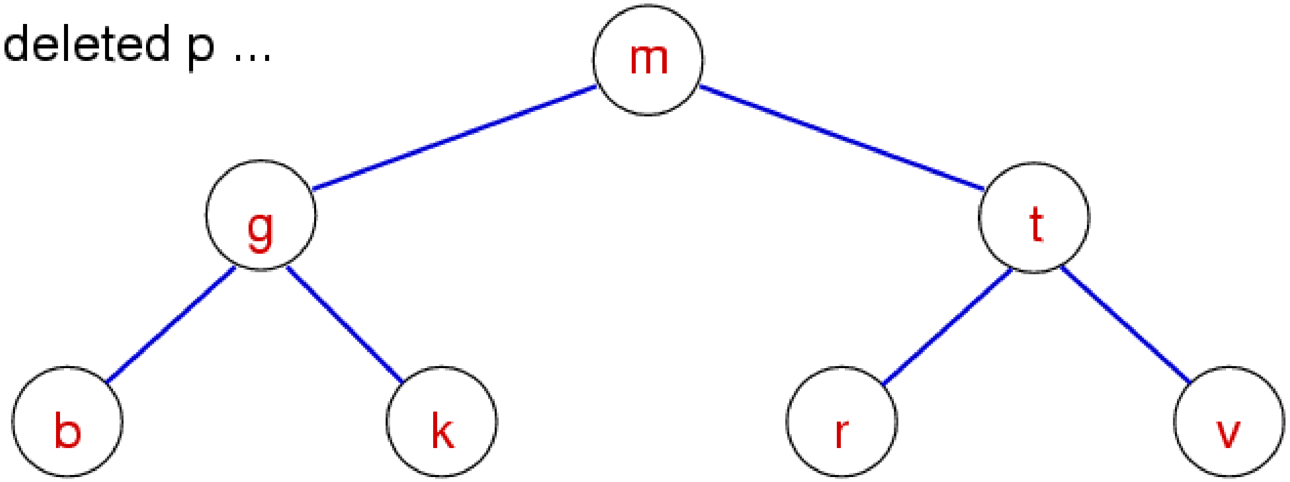


Replace the item by its only subtree

Deletion from BSTs (cont)

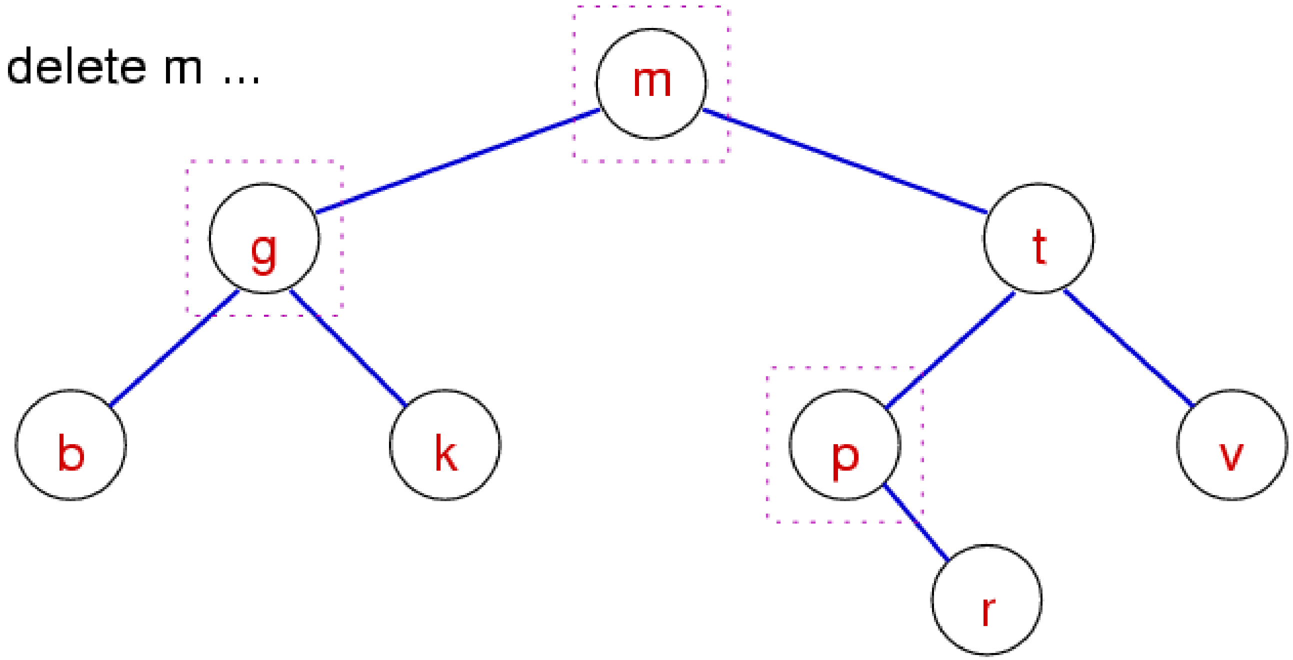
Case 3: item to be deleted has one subtree

deleted p ...



Deletion from BSTs (cont)

Case 4: item to be deleted has two subtrees

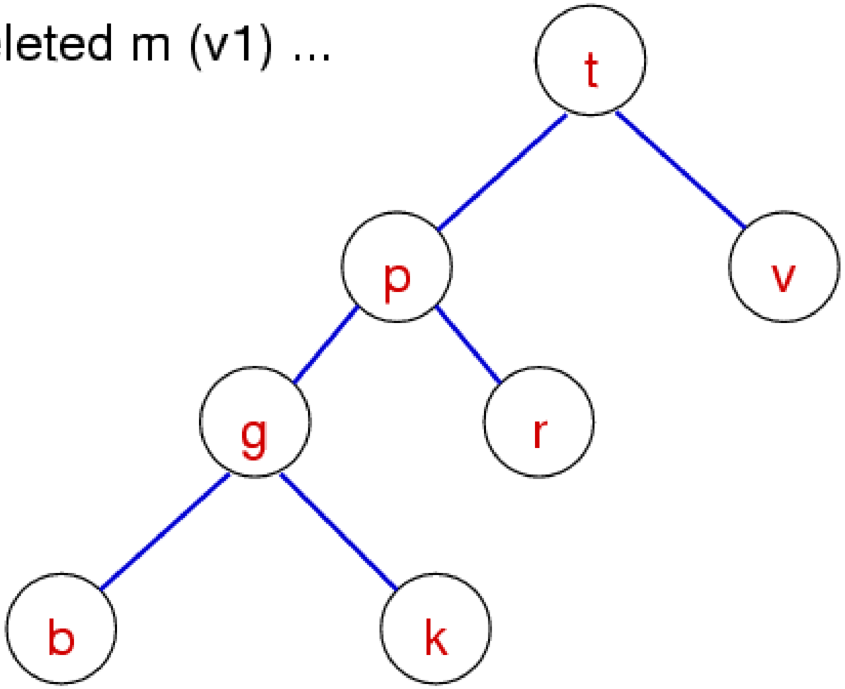


Version 1: right child becomes new root, attach left subtree to min element of right subtree

Deletion from BSTs (cont)

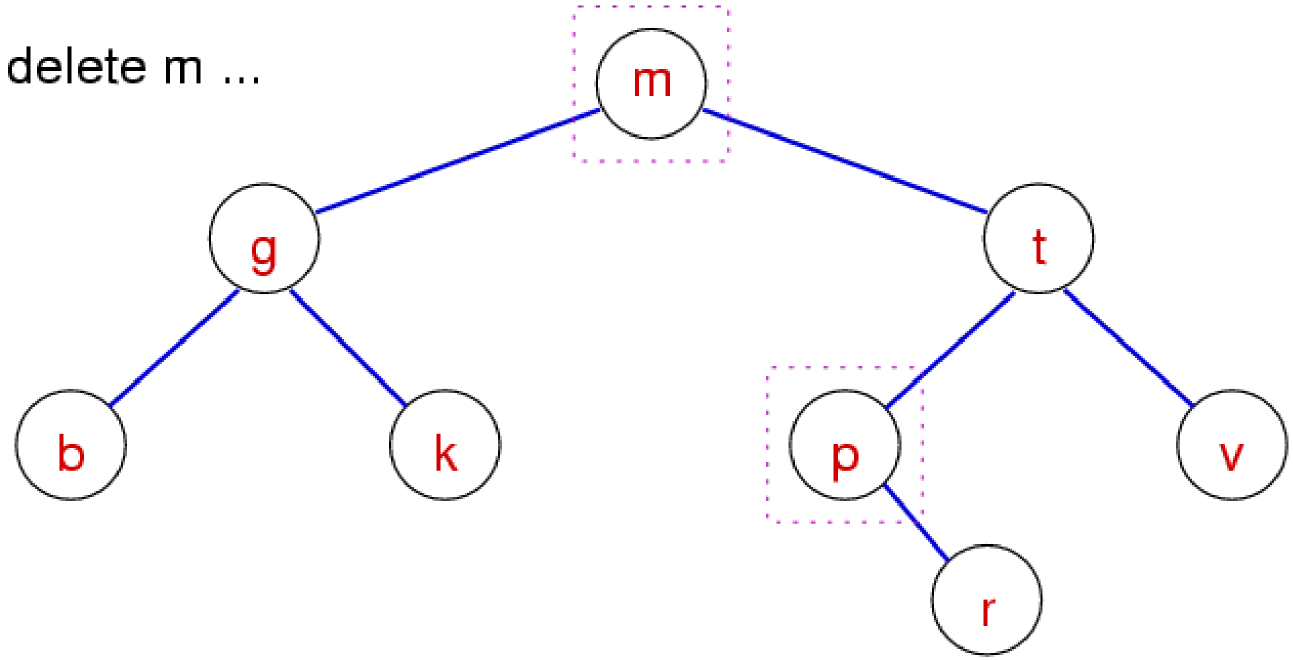
Case 4: item to be deleted has two subtrees

deleted m (v1) ...



Deletion from BSTs (cont)

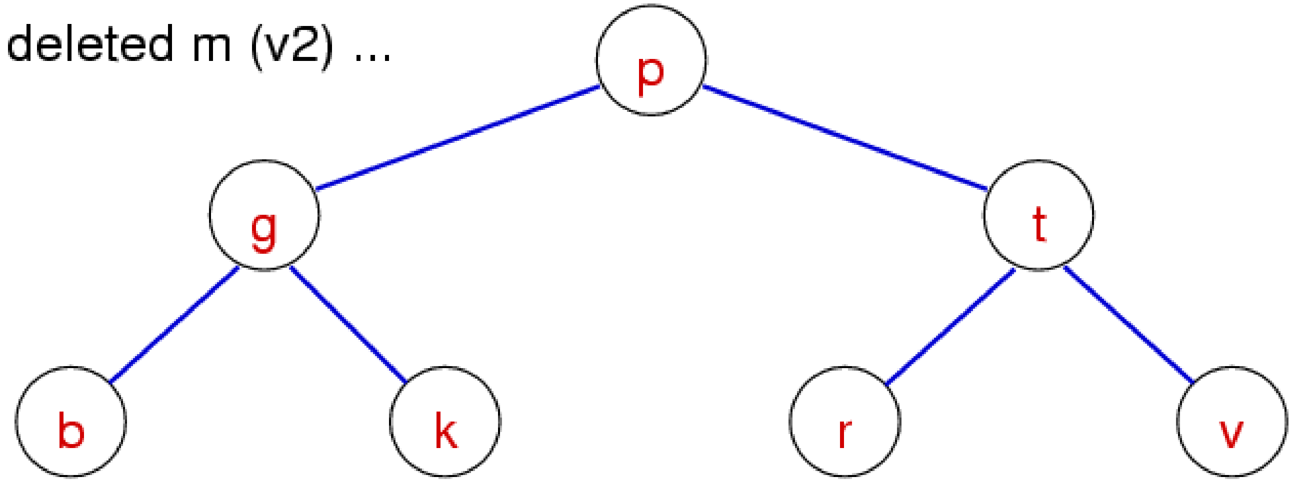
Case 4: item to be deleted has two subtrees



Version 2: *join* left and right subtree

Deletion from BSTs (cont)

Case 4: item to be deleted has two subtrees



Deletion from BSTs (cont)

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=joinTrees(left(t),right(t))      // 2 children
|   |  |  end if
|   |  |  free memory allocated for t
|   |  |  t=new
|   |  end if
|   end if
|   return t
```

Balanced BSTs

Balanced Binary Search Trees

Goal: build binary search trees which have

- minimum height ⇒ minimum worst case search cost

Perfectly balanced tree with N nodes has

- abs(#nodes(LeftSubtree) - #nodes(RightSubtree)) < 2, for every node
- height of $log_2 N$ ⇒ worst case search $O(log\ N)$

Three *strategies* to improving worst case search in BSTs:

- randomise — reduce chance of worst-case scenario occuring
- amortise — do more work at insertion to make search faster
- optimise — implement all operations with performance bounds

Operations for Rebalancing

To assist with rebalancing, we consider new operations:

Left rotation

- move right child to root; rearrange links to retain order

Right rotation

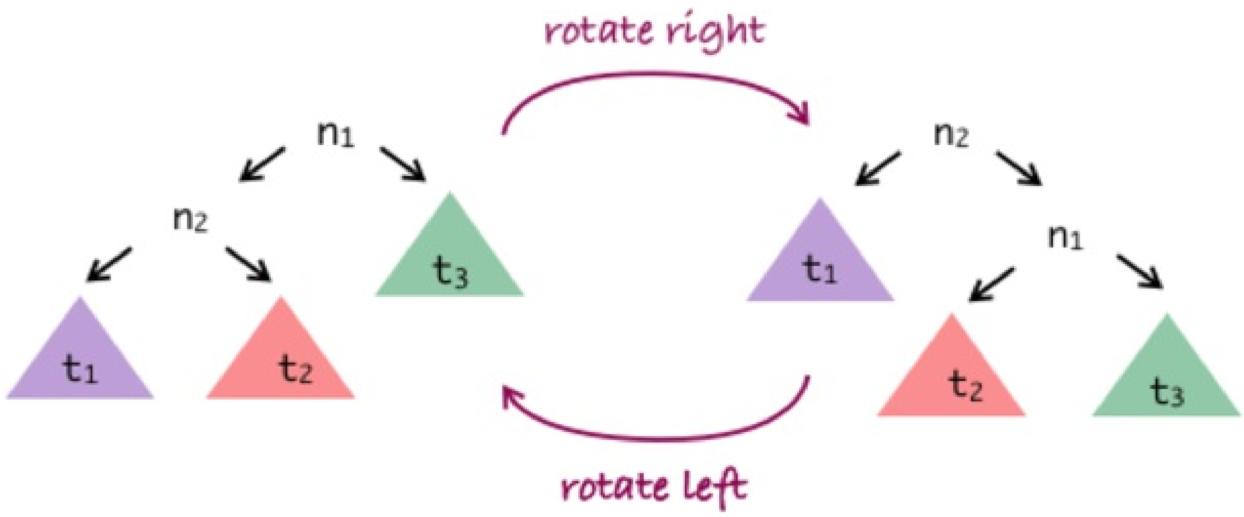
- move left child to root; rearrange links to retain order

Insertion at root

- each new item is added as the new root node

Tree Rotation

In tree below: $t_1 < n_2 < t_2 < n_1 < t_3$



Tree Rotation (cont)

Method for rotating tree T right:

- N_1 is current root; N_2 is root of N_1's left subtree
- N_1 gets new left subtree, which is N_2's right subtree
- N_1 becomes root of N_2's new right subtree
- N_2 becomes new root

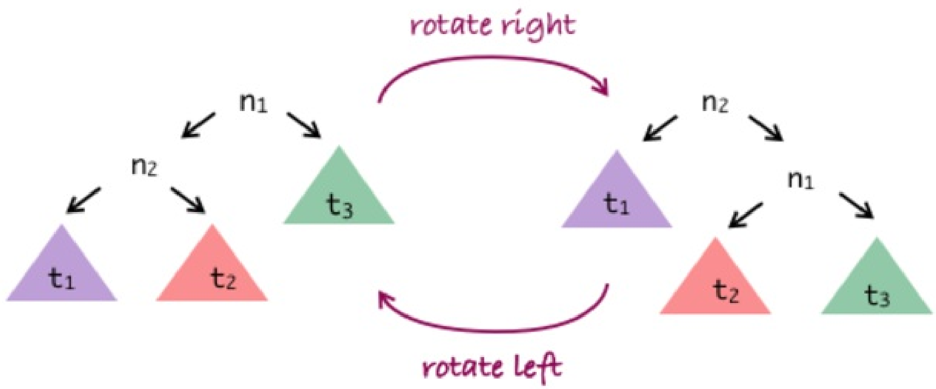
Left rotation: swap left/right in the above.

Cost of tree rotation: *O(1)*

Tree Rotation (cont)

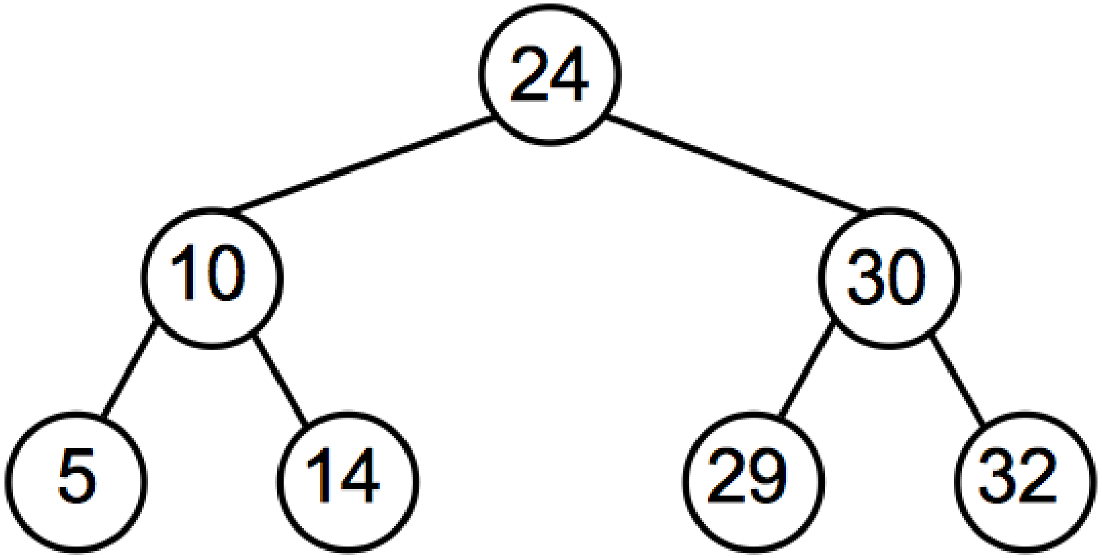
Algorithm for right rotation:

```
rotateRight(n₁):
│  Input  tree n₁
│  Output n₁ rotated to the right
│
│  if n₁ is empty ∨ left(n₁) is empty then
│     return n₁
│  end if
│  n₂=left(n₁)
│  left(n₁)=right(n₂)
│  right(n₂)=n₁
│  return n₂
```

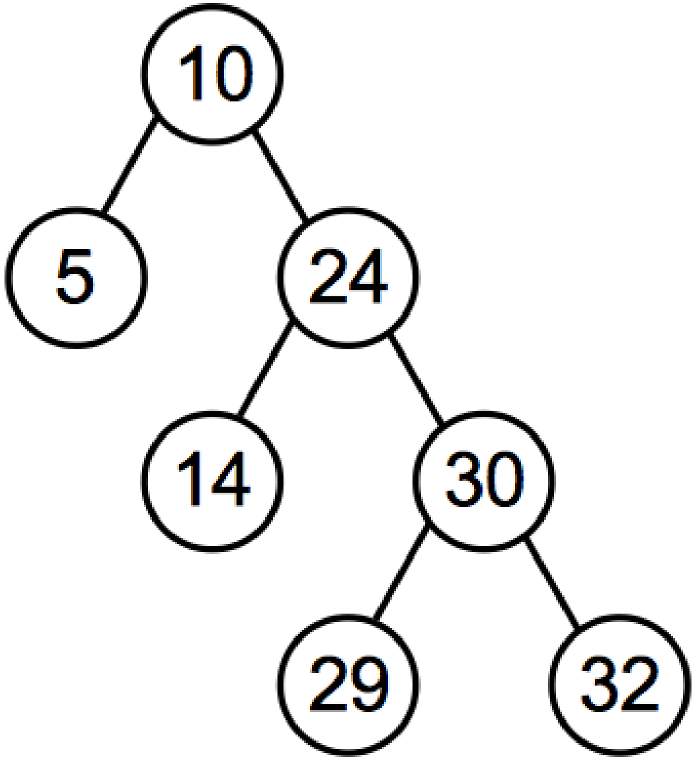


Exercise #5: Tree Rotation

Consider the tree **`t`**:

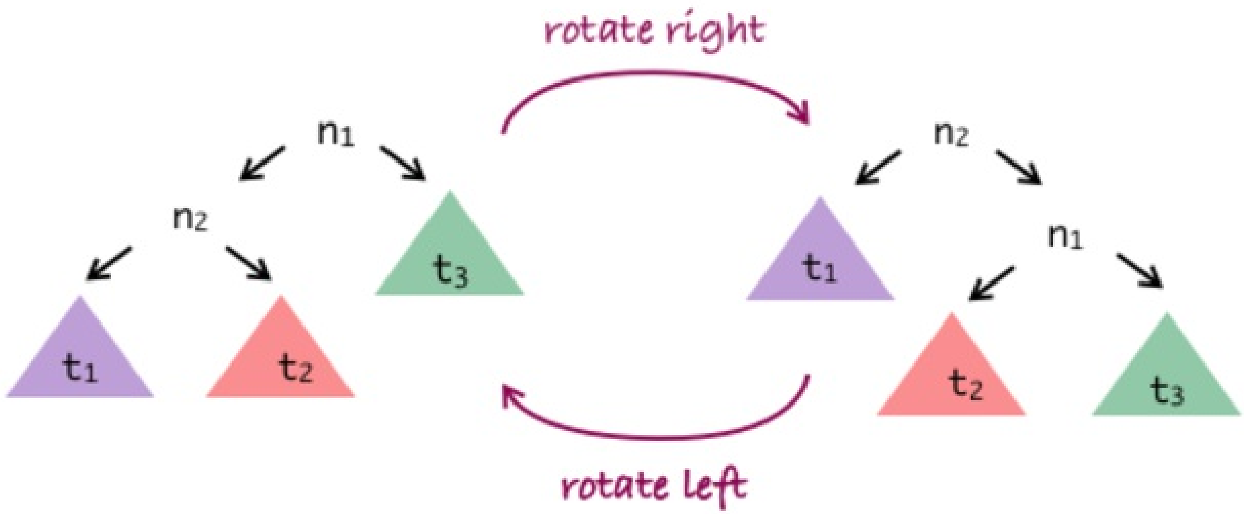


Show the result of **`rotateRight(t)`**


10
5
24
14
30
29
32

Exercise #6: Tree Rotation

Write the algorithm for left rotation



```
rotateLeft(n₂):
|   Input  tree n₂
|   Output n₂ rotated to the left
|
|   if n₂ is empty ∨ right(n₂) is empty then
|      return n₂
|   end if
|   n₁=right(n₂)
|   right(n₂)=left(n₁)
|   left(n₁)=n₂
|   return n₁
```

Insertion at Root

Previous description of BSTs inserted at leaves.

Different approach: insert new item at root.

Potential disadvantages:

- large-scale rearrangement of tree for each insert

Potential advantages:

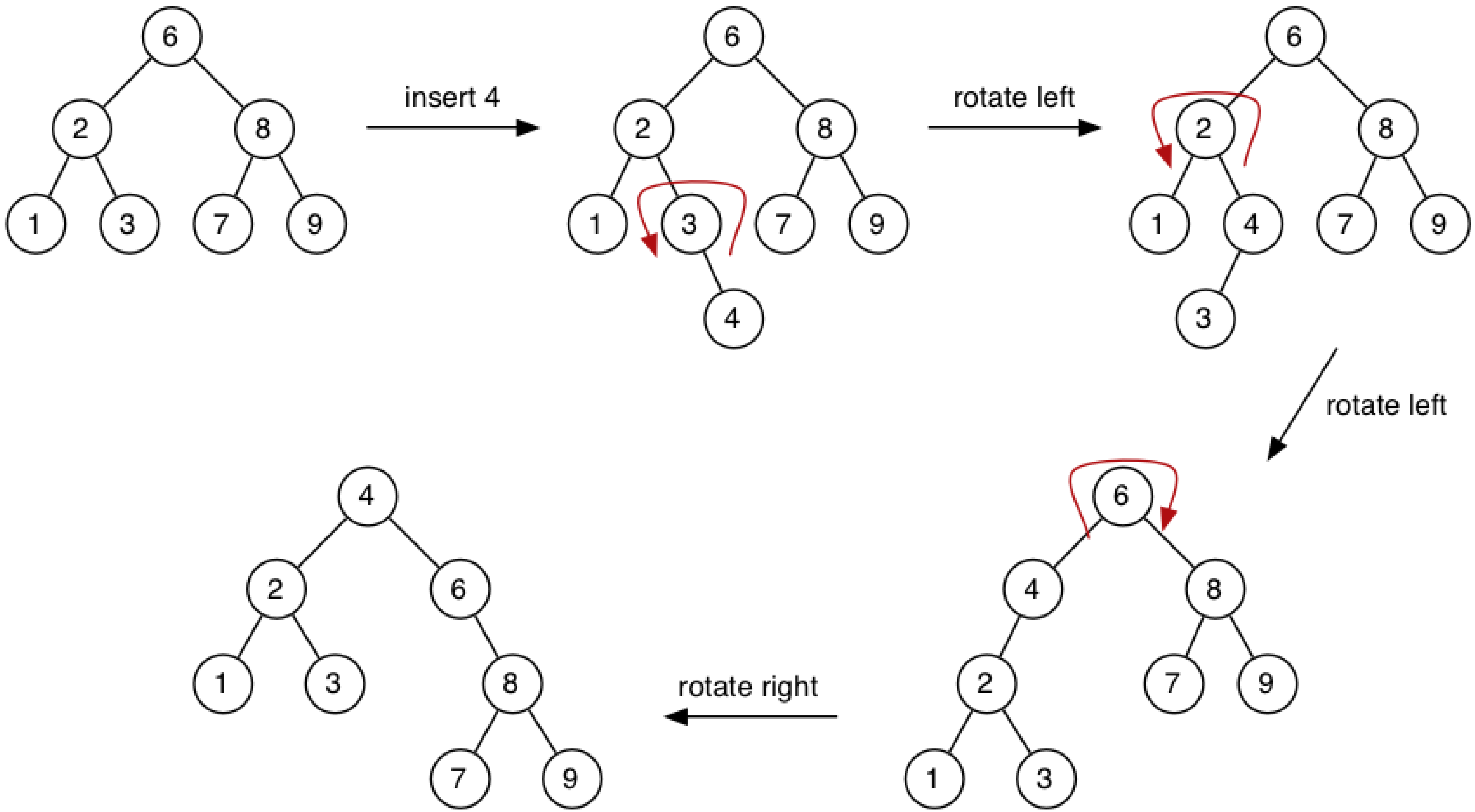
- recently-inserted items are close to root
- low cost if recent items more likely to be searched

Insertion at Root (cont)

Method for inserting at root:

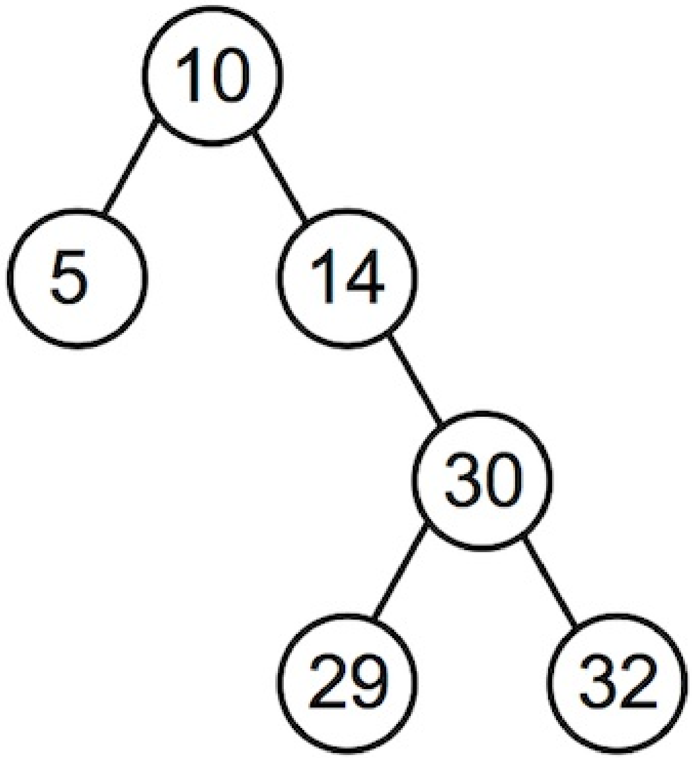
- base case:
  - tree is empty; make new node and make it root
- recursive case:
  - insert new node as root of appropriate subtree
  - lift new node to root by rotation

Insertion at Root (cont)

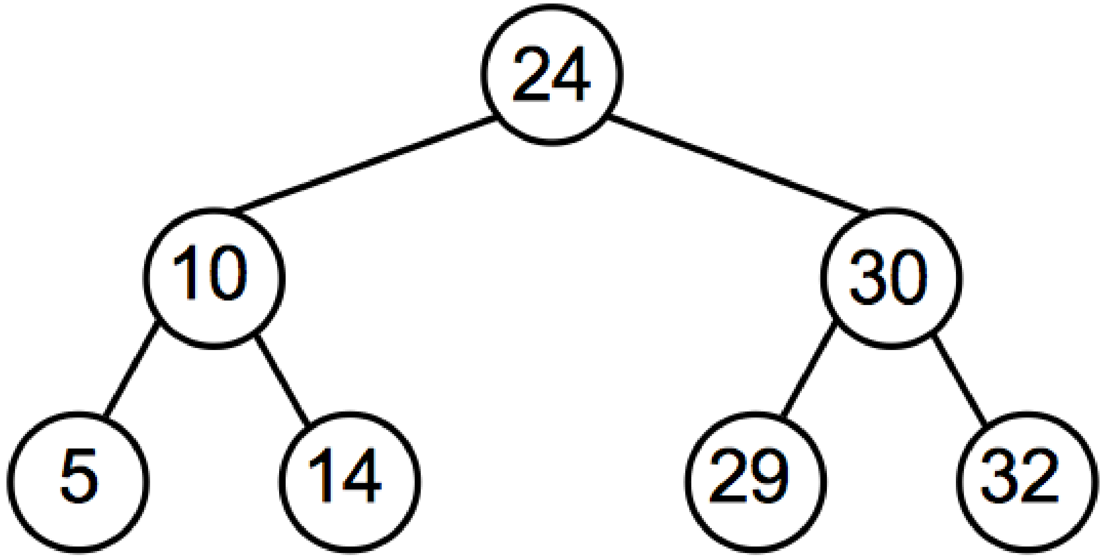


Exercise #7: Insertion at Root

Consider the tree **`t`**:



Show the result of **`insertAtRoot(t,24)`**


24
10
30
5
14
29
32

Insertion at Root (cont)

Analysis of insertion-at-root:

- same complexity as for insertion-at-leaf: *O(height)*
- tendency to be balanced, but no balance guarantee
- benefit comes in searching
  - for some applications, search favours recently-added items
  - insertion-at-root ensures these are close to root
- could even consider "move to root when found"
  - effectively provides "self-tuning" search tree

Rebalancing Trees

An approach to balanced trees:

- insert into leaves as for simple BST
- periodically, rebalance the tree

Question: how frequently/when/how to rebalance?

```
NewTreeInsert(tree,item):
|  Input  tree, item
|  Output tree with item randomly inserted
|
|  t=insertAtLeaf(tree,item)
|  if #nodes(t) mod k = 0 then
|     t=rebalance(t)
|  end if
|  return t
```

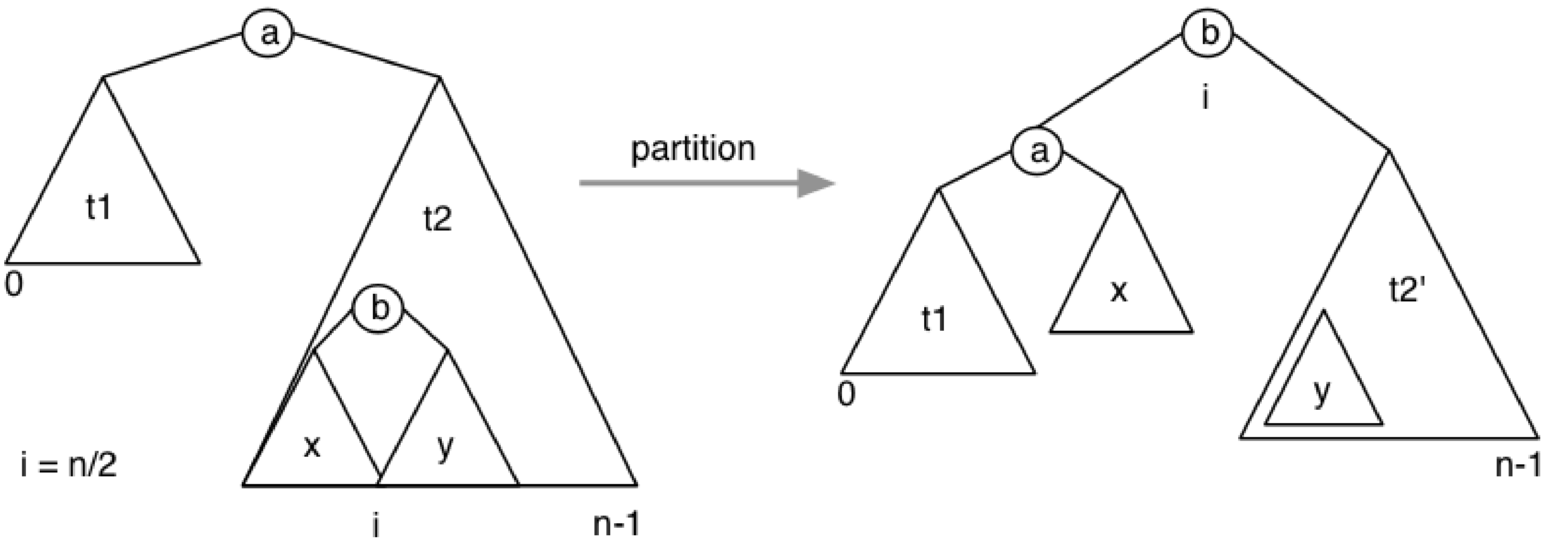
E.g. rebalance after every 20 insertions ⇒ choose *k=20*

Note: To do this efficiently we would need to change tree data structure and basic operations:

```
typedef struct Node {
   int  data;
   int  nnodes;      // #nodes in my tree
   Tree left, right; // subtrees
} Node;
```

Rebalancing Trees (cont)

How to rebalance a BST? Move median item to root.



Rebalancing Trees (cont)

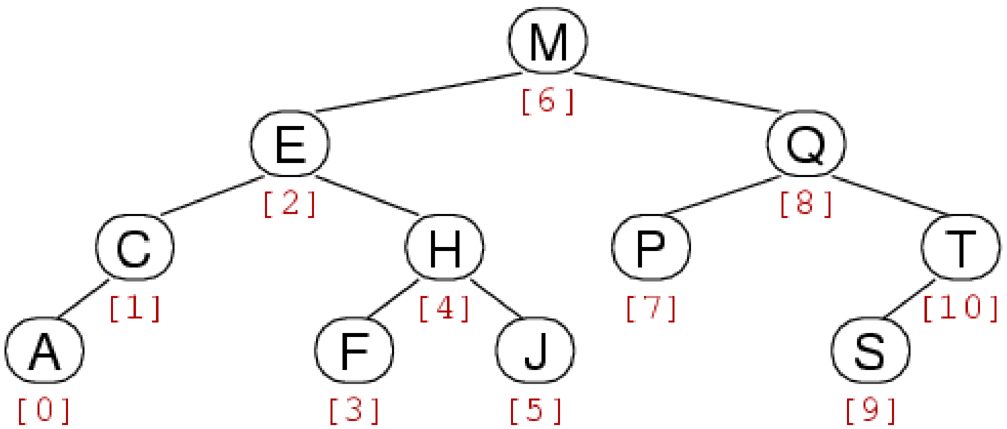
Implementation of rebalance:

```
rebalance(t):
|  Input  tree t with n nodes
|  Output t rebalanced
|
|  if n≥3 then
|  |  t=partition(t,⌊n/2⌋)         // put node with median key at root
|  |  left(t)=rebalance(left(t))   // then rebalance each subtree
|  |  right(t)=rebalance(right(t))
|  end if
|  return t
```

Rebalancing Trees (cont)

New operation on trees:

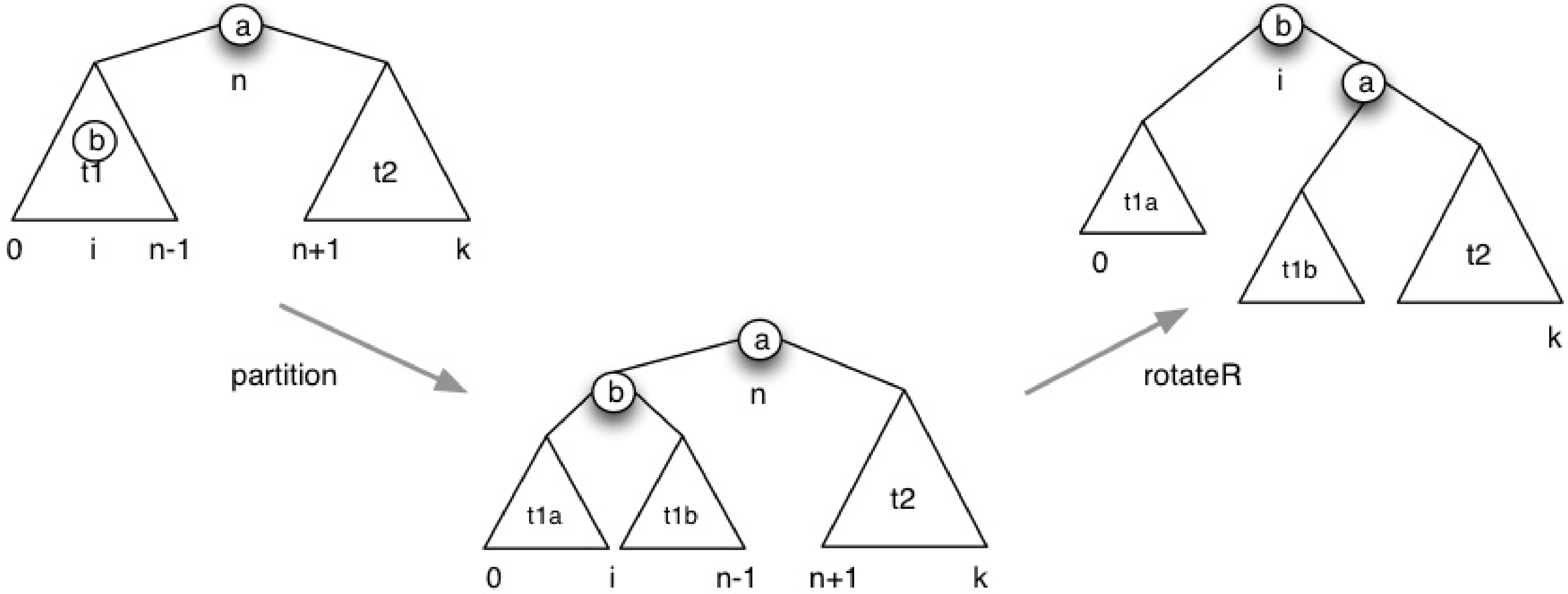
- `partition(tree,i)`: re-arrange tree so that element with index *i* becomes root



For tree with *N* nodes, indices are *0 .. N-1*

Rebalancing Trees (cont)

Partition: moves i th node to root



Rebalancing Trees (cont)

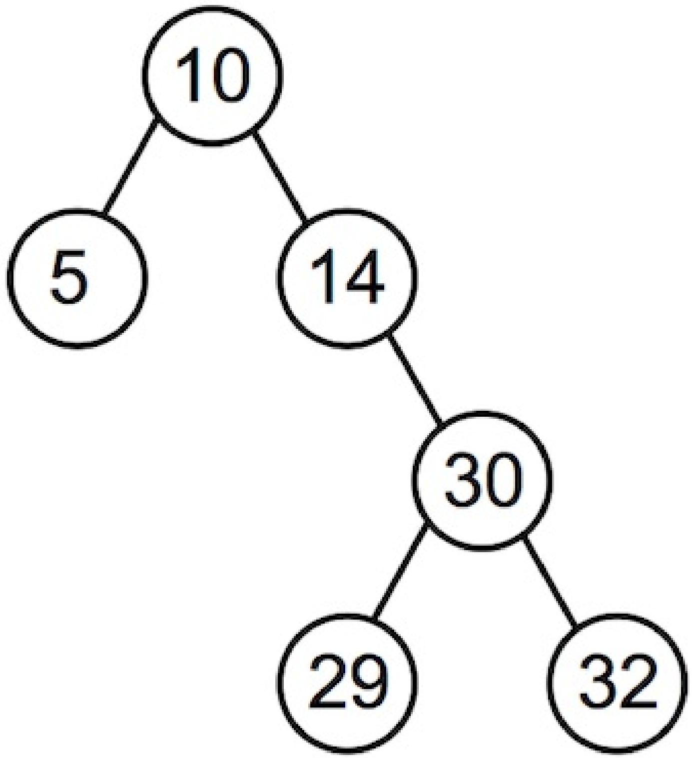
Implementation of partition operation:

```
partition(tree,i):
|  Input  tree with n nodes, index i
|  Output tree with i^th item moved to the root
|
|  m=#nodes(left(tree))
|  if i < m then
|     left(tree)=partition(left(tree),i)
|     tree=rotateRight(tree)
|  else if i > m then
|     right(tree)=partition(right(tree),i-m-1)
|     tree=rotateLeft(tree)
|  end if
|  return tree
```

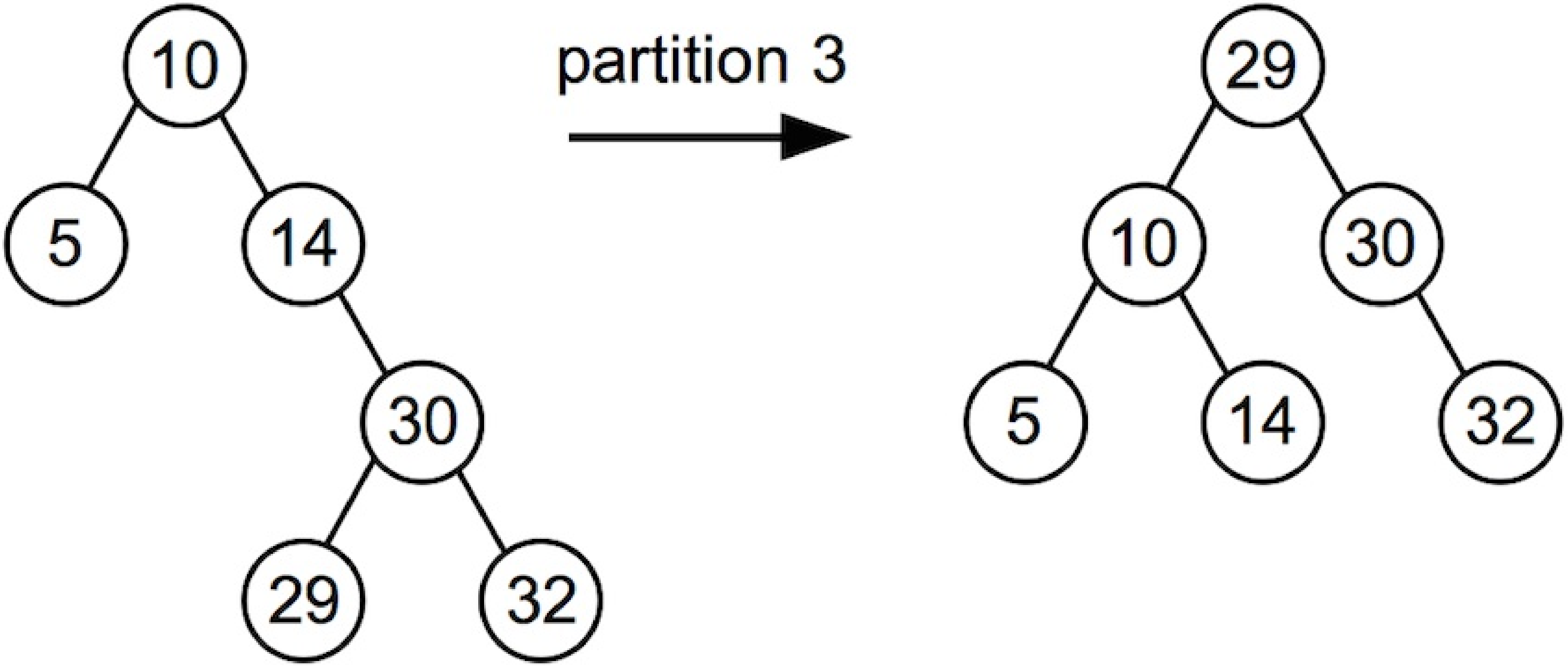
Note: size(tree) = n, size(left(tree)) = m, size(right(tree)) = n-m-1 (why -1?)

Exercise #8: Partition

Consider the tree **`t`**:



Show the result of **`partition(t,3)`**


10
5
14
30
29
32
partition 3
29
10
30
5
14
32

Rebalancing Trees (cont)

Analysis of rebalancing: visits every node ⇒ *O(N)*

Cost means not feasible to rebalance after each insertion.

When to rebalance? … Some possibilities:

- after every *k* insertions
- whenever "imbalance" exceeds threshold

Either way, we tolerate worse search performance for periods of time.

Does it solve the problem? … Not completely ⇒ Solution: real balanced trees (next week)

Application of BSTs: Sets

Trees provide efficient search.

Sets require efficient search

- to find where to insert/delete
- to test for set membership

Logical to implement a set ADT via **`BSTree`**

Application of BSTs: Sets (cont)

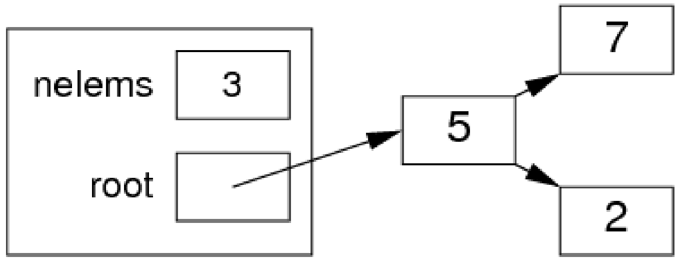
Assuming we have `Tree` implementation

- which precludes duplicate key values
- which implements

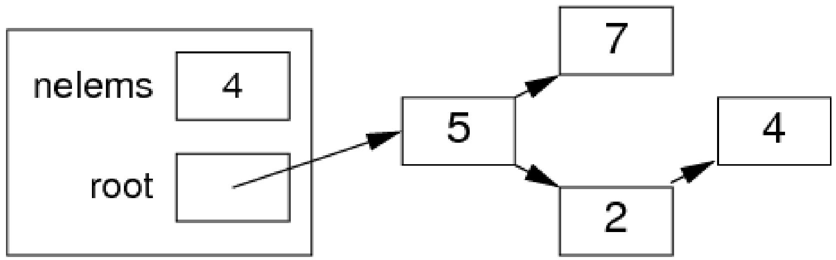
then `Set` implementation is

- `SetInsert(Set,Item)` ≡ `TreeInsert(Tree,Item)`
- `SetDelete(Set,Item)` ≡ `TreeDelete(Tree,Item.Key)`
- `SetMember(Set,Item)` ≡ `TreeSearch(Tree,Item.Key)`

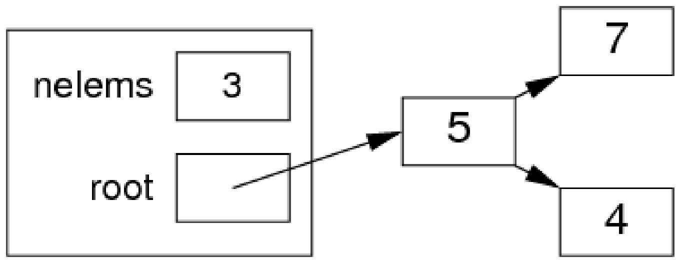
Application of BSTs: Sets (cont)



After SetInsert(s,4):



After SetDelete(s,2):



Application of BSTs: Sets (cont)

Concrete representation:

```
#include <BSTree.h>

typedef struct SetRep {
   int    nelems;
   Tree   root;
} SetRep;

Set newSet() {
   Set S = malloc(sizeof(SetRep));
   assert(S != NULL);
   S->nelems = 0;
   S->root = newTree();
   return S;
}
```

Summary

- Binary search tree (BST) data structure
- BST insertion and deletion
- Other tree operations
  - tree rotation
  - tree partition
  - joining trees

- Suggested reading:
  - Sedgewick, Ch.12.5-12.6,12.8-12.9