Week 7: Search Tree Data Structures

Searching

1/63

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

2/63

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)

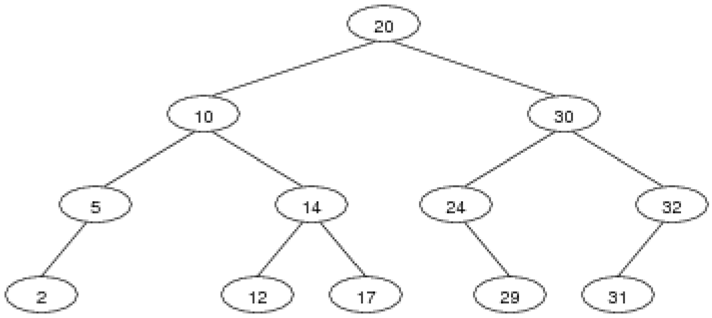
... Searching

3/63

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]`:

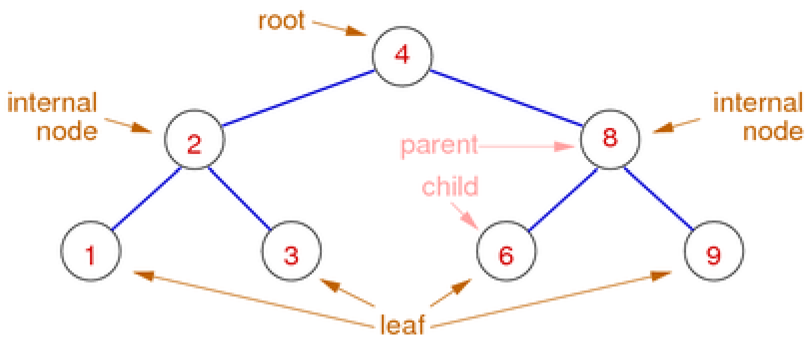


Tree Data Structures

4/63

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)

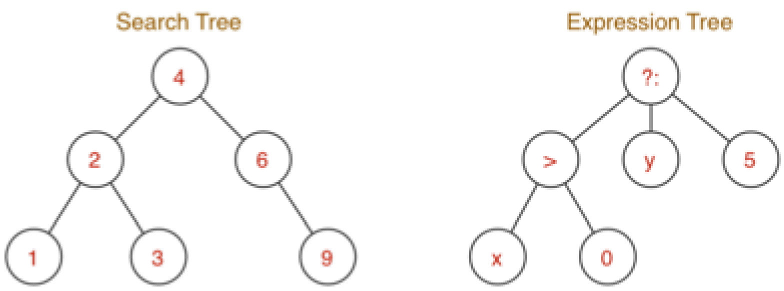


... Tree Data Structures

5/63

Trees are used in many contexts, e.g.

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

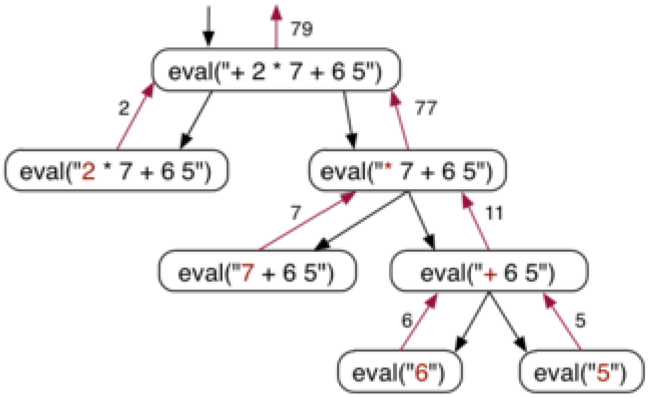


... Tree Data Structures

6/63

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

E.g. showing evaluation of a prefix arithmetic expression



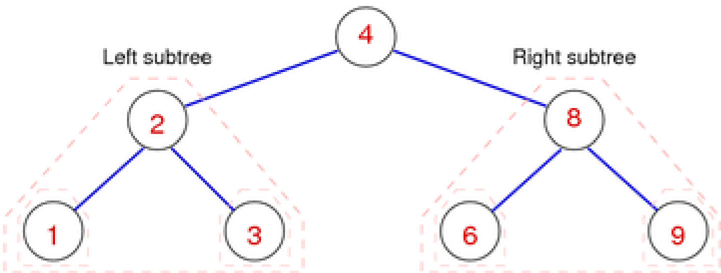
... Tree Data Structures

7/63

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*



... Tree Data Structures

8/63

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

9/63

Binary Search Trees

10/63

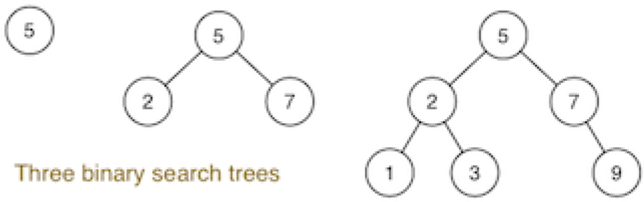
Binary search trees (or *BSTs*) have the characteristic properties

- 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



... Binary Search Trees

11/63

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(), …

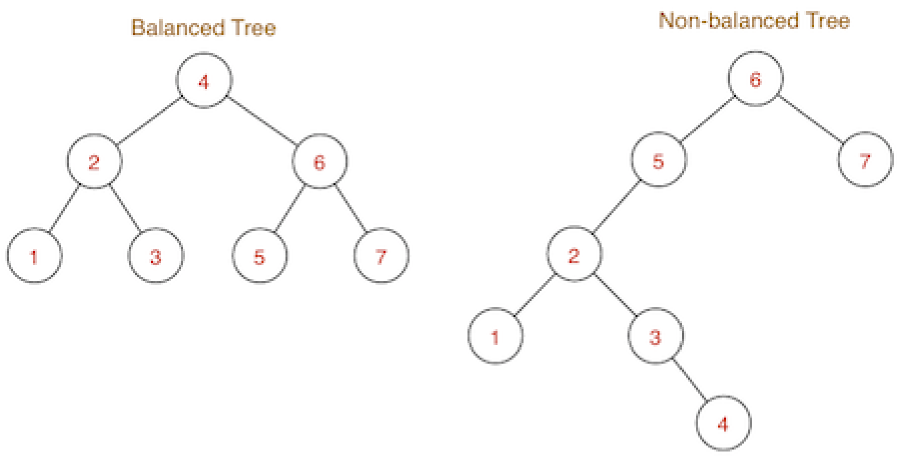
Notes:

- in general, nodes contain `Items`; we just show `Item.key`
- keys are unique (not technically necessary)

... Binary Search Trees

12/63

Examples of binary search trees:



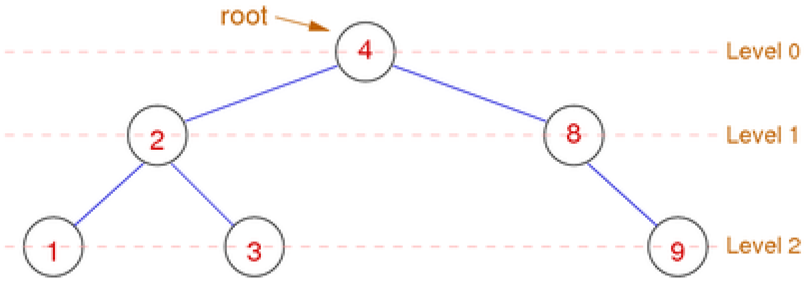
Shape of tree is determined by order of insertion.

... Binary Search Trees

13/63

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)

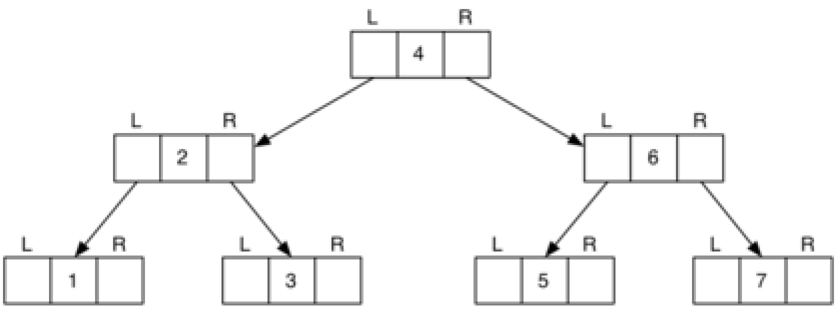
(c) a fully degenerate tree of height 6

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

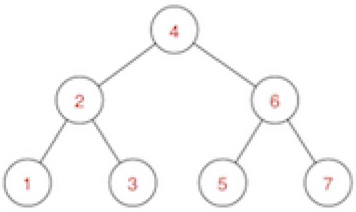
Typical data structures for trees …

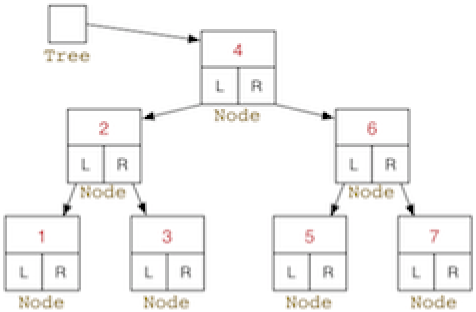
```
// a Tree is represented by a pointer to its root node
typedef struct Node *Tree;

// 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 $\Rightarrow$ `data` in `Node` is just a key

... Representing BSTs

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

21/63

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

22/63

Iteration (traversal) on …

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

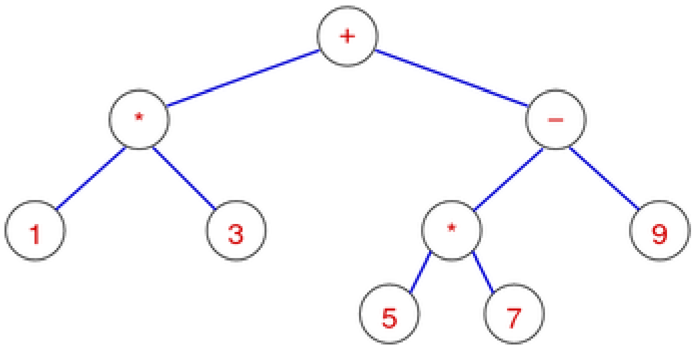
For binary `Trees`, several well-defined visiting orders exist:

- *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

23/63

Consider "visiting" an expression tree like:

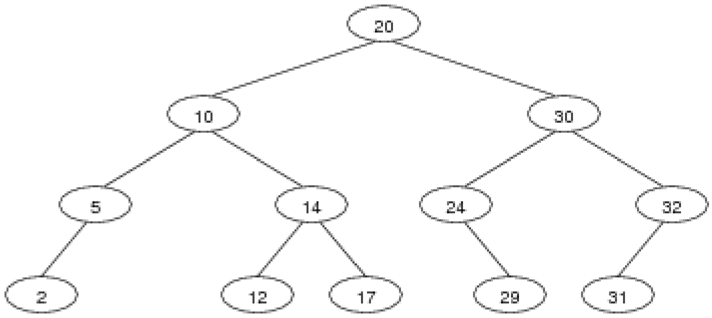


NLR: + * 1 3 - * 5 7 9 (prefix-order: useful for building tree)
LNR: 1 * 3 + 5 * 7 - 9 (infix-order: "natural" order)
LRN: 1 3 * 5 7 * 9 - + (postfix-order: useful for evaluation)
Level: + * - 1 3 * 9 5 7 (level-order: useful for printing tree)

Exercise #2: Tree Traversal

24/63

Show NLR, LNR, LRN traversals for the 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

26/63

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

28/63

An auxiliary tree operation …

Tree operations so far have involved just one tree.

An operation on two trees: `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

29/63

Method for performing tree-join:

- find the min node in the right subtree (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

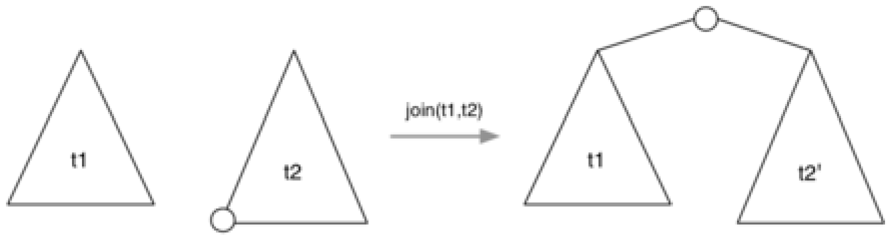
$x \le height(t) \le x+1$, where $x = max(height(t_1),height(t_2))$

Variation: choose deeper subtree; take root from there.

... Joining Two Trees

30/63

Joining two trees:



Note: t2' may be less deep than t2

... Joining Two Trees

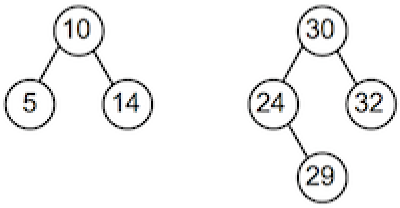
31/63

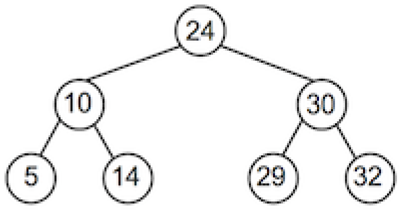
Implementation of tree-join

```
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





Deletion from BSTs

Insertion into a binary search tree is easy.

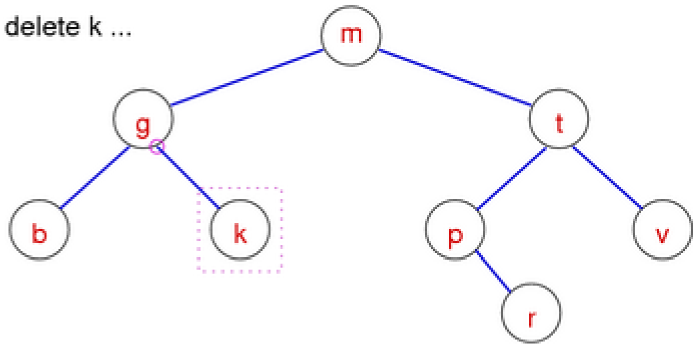
Deletion from a binary search tree is harder.

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

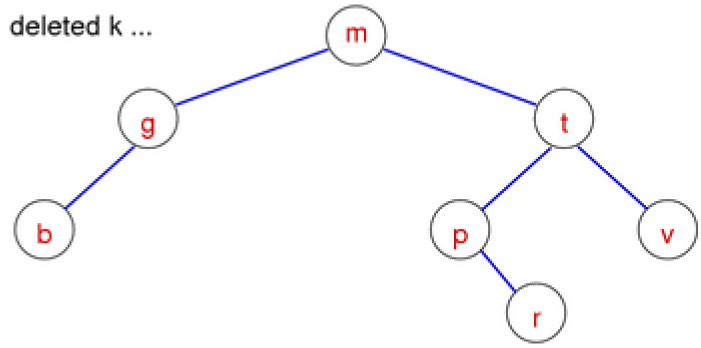
Case 2: item to be deleted is a leaf (zero subtrees)



Just delete the item

... Deletion from BSTs

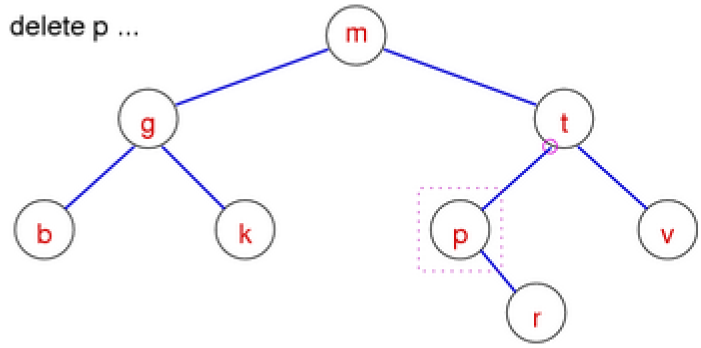
Case 2: item to be deleted is a leaf (zero subtrees)



... Deletion from BSTs

37/63

Case 3: item to be deleted has one subtree

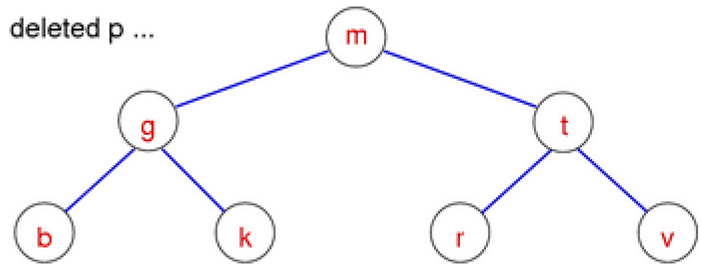


Replace the item by its only subtree

... Deletion from BSTs

38/63

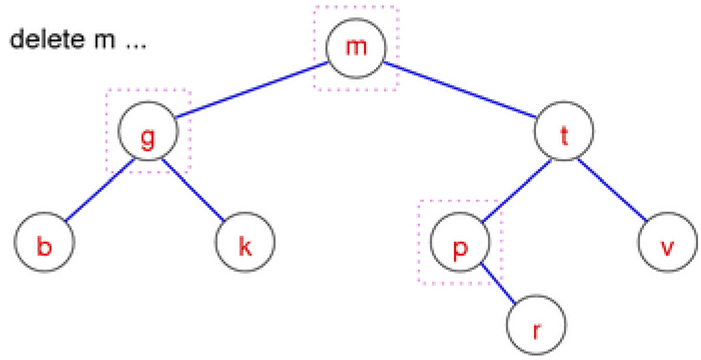
Case 3: item to be deleted has one subtree



... Deletion from BSTs

39/63

Case 4: item to be deleted has two subtrees

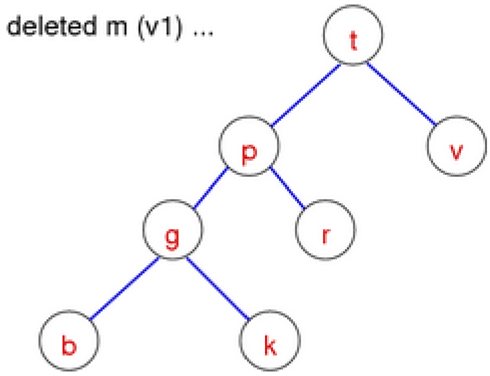


Version 1: right child becomes new root, attach left subtree to min element of right subtree

... Deletion from BSTs

40/63

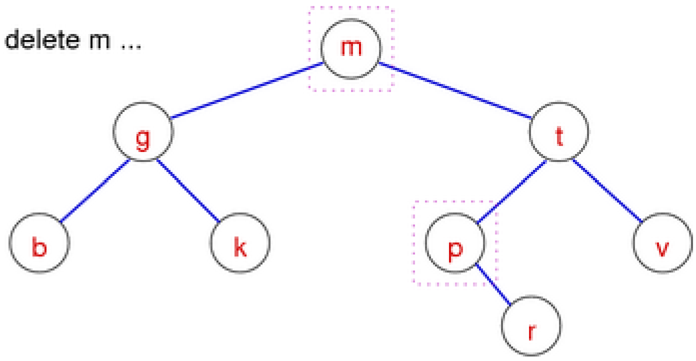
Case 4: item to be deleted has two subtrees



... Deletion from BSTs

41/63

Case 4: item to be deleted has two subtrees

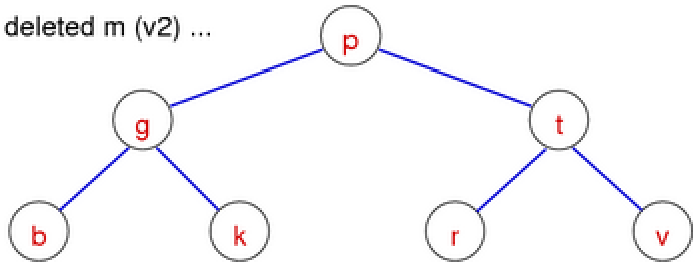


Version 2: *join* left and right subtree

... Deletion from BSTs

42/63

Case 4: item to be deleted has two subtrees



... Deletion from BSTs

43/63

Pseudocode (version 2 for case 4)

```
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 right 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 Binary Search Trees

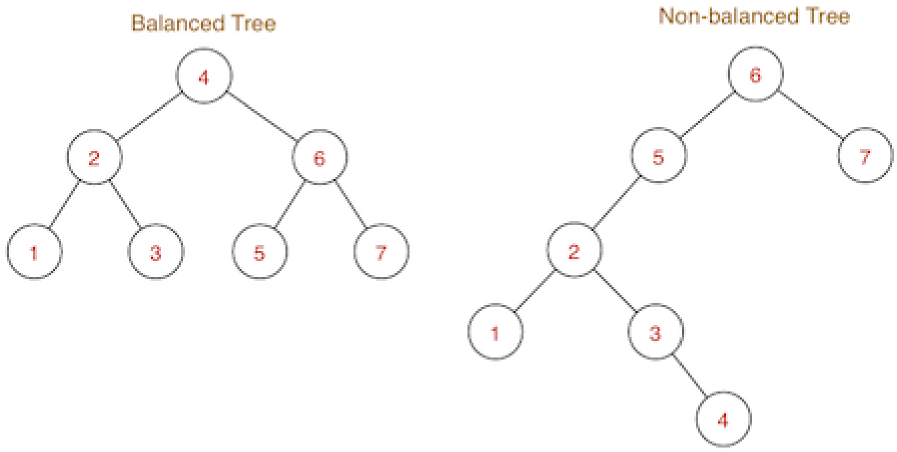
44/63

Goal: build binary search trees which have

- minimum height ⇒ minimum worst case search cost

Best balance you can achieve for tree with N nodes:

- abs(#nodes(LeftSubtree) - #nodes(RightSubtree)) ≤ 1, for every node
- height of $log_2 N$ ⇒ worst case search $O(log\ N)$



Operations for Rebalancing

45/63

To assist with rebalancing, we consider new operations:

Left rotation

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

Right rotation

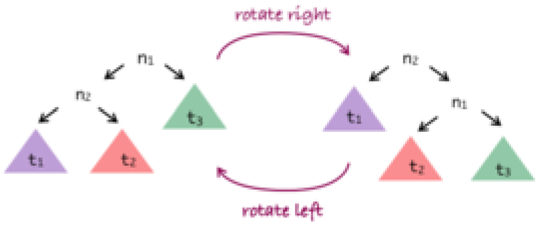
- move left child to root; rearrange links to retain order

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

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

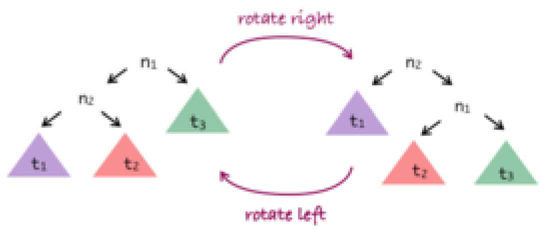
Left rotation: swap left/right in the above.

Cost of tree rotation: *O(1)*

... Tree Rotation

Algorithm for right rotation:

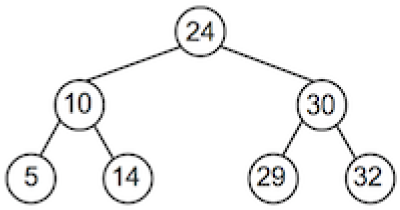
```
rotateRight(n1):
|  Input  tree n1
|  Output n1 rotated to the right
|
|  if n1 is empty or left(n1) is empty then
|     return n1
|  end if
|  n2=left(n1)
|  left(n1)=right(n2)
|  right(n2)=n1
|  return n2
```



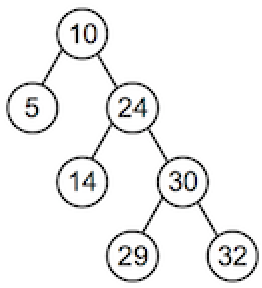
Exercise #5: Tree Rotation

Consider the tree `t`:

N1 is current root; N2 is root of N1's left subtree
N1 gets new left subtree, which is N2's right subtree N1 becomes root of N2's new right subtree
N2 becomes new root

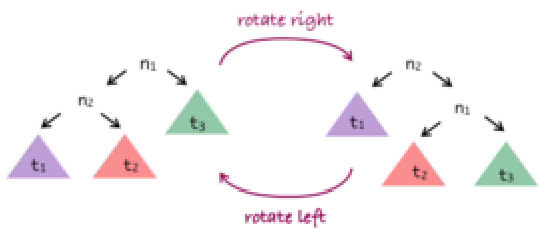


Show the result of `rotateRight(t)`



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 or right(n2) is empty then
|     return n2
|  end if
|  n1=right(n2)
```

```
|   right(n₂)=left(n₁)
|   left(n₁)=n₂
|   return n₁
```

Insertion at Root

53/63

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

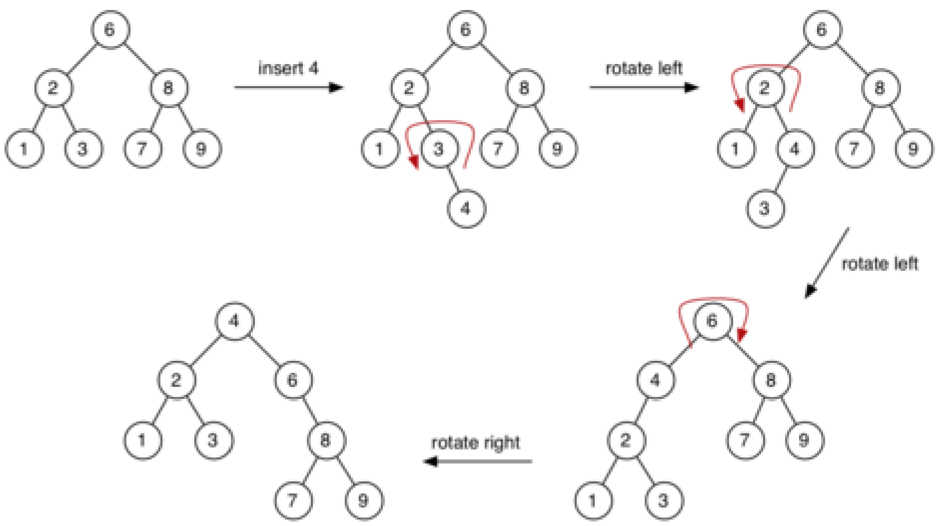
54/63

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

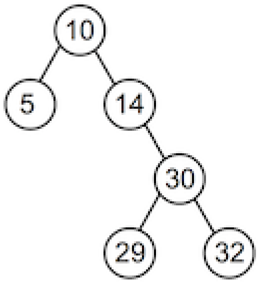
55/63



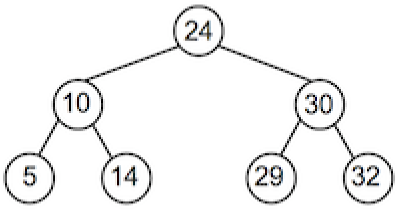
Exercise #7: Insertion at Root

56/63

Consider the tree `t`:



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



... Insertion at Root

58/63

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

  ⇒ Real-balanced trees (week 8)

Application of BSTs: Sets

59/63

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

60/63

Assuming we have `Tree` implementation

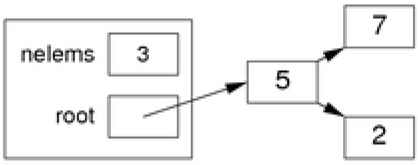
- which precludes duplicate key values
- which implements insertion, search, deletion

then `Set` implementation is

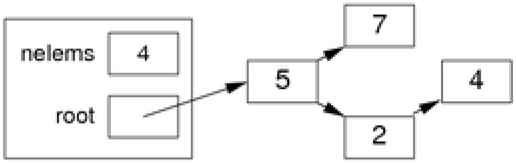
- `SetInsert(Set,Item)` ≡ `TreeInsert(Tree,Item)`
- `SetDelete(Set,Item)` ≡ `TreeDelete(Tree,Item.Key)`
- `SetMember(Set,Item)` ≡ `TreeSearch(Tree,Item.Key)`

... Application of BSTs: Sets

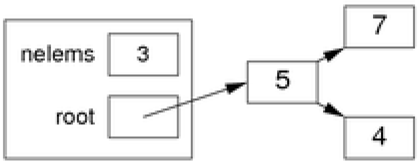
61/63



After SetInsert(s,4):



After SetDelete(s,2):



... Application of BSTs: Sets

62/63

Concrete representation:

```
#include "BSTree.h"

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

typedef SetRep *Set;

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

Summary

63/63

- Binary search tree (BST) data structure
- Tree traversal
- Basic BST operation: insertion, join, deletion, rotation

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

Produced: 17 Jul 2020