Week 03b: Search Tree Data Structures

Searching

1/49

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

2/49

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)

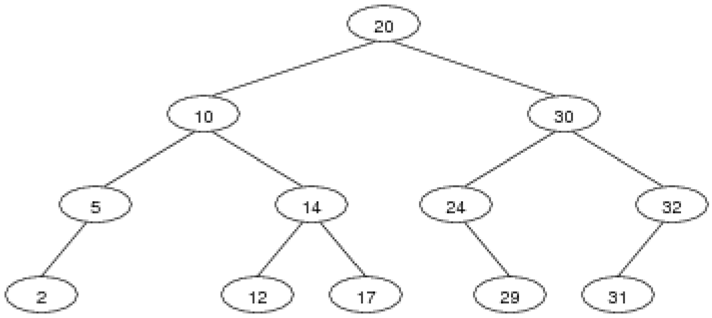
... Searching

3/49

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]`:



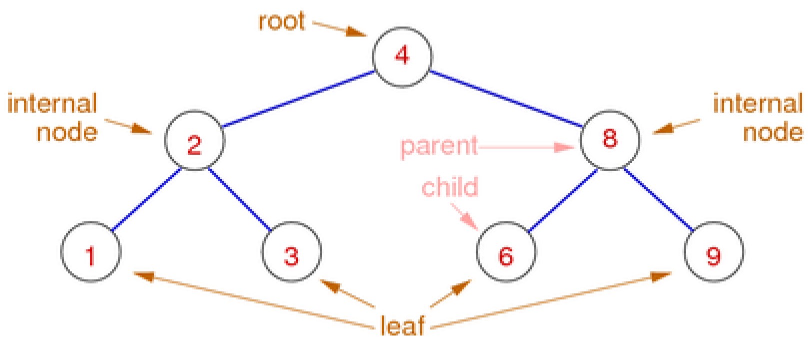
Tree Data Structures

Trees

5/49

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)

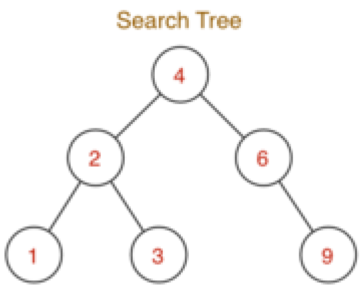


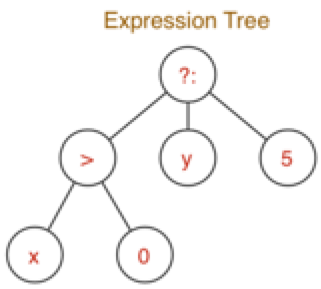
... Trees

6/49

Trees are used in many contexts, e.g.

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



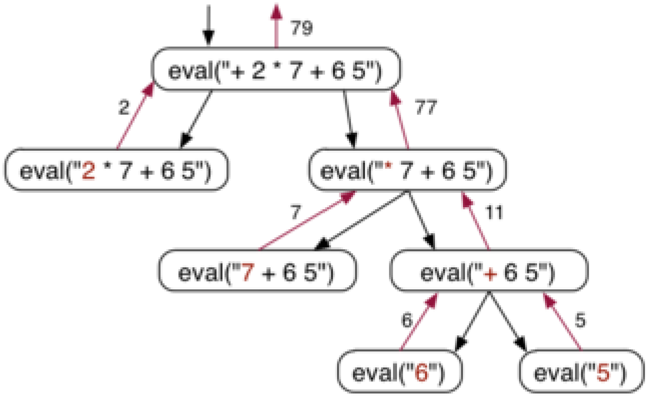


... Trees

7/49

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

E.g. showing evaluation of a prefix arithmetic expression



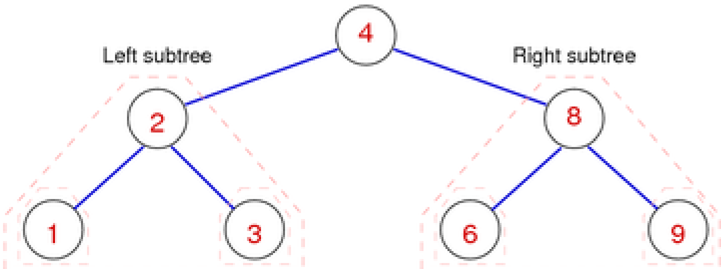
... Trees

8/49

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

9/49

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

10/49

Binary Search Trees

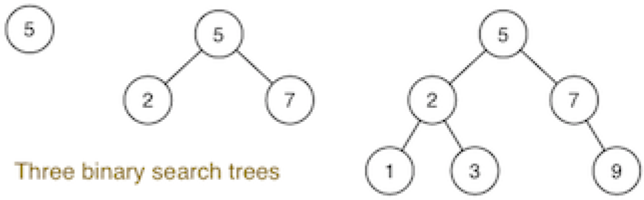
11/49

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

12/49

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(), ...

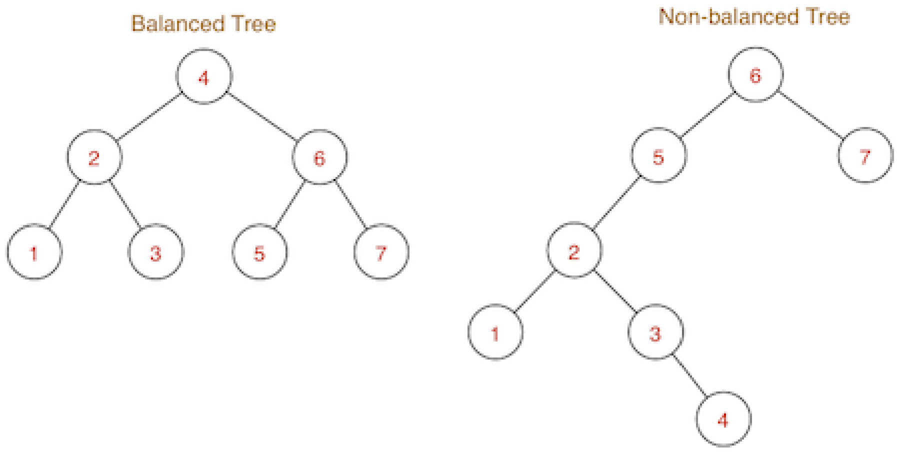
Notes:

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

... Binary Search Trees

13/49

Examples of binary search trees:



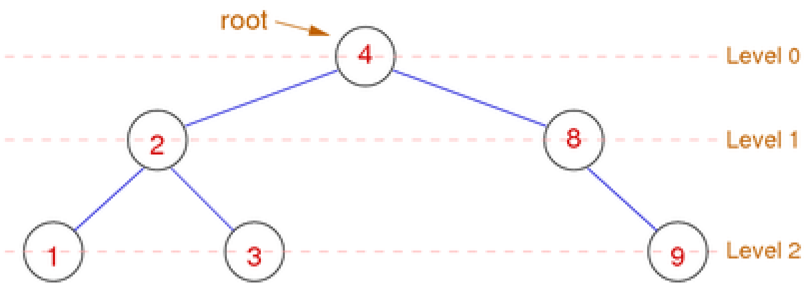
Shape of tree is determined by order of insertion.

... Binary Search Trees

14/49

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

15/49

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

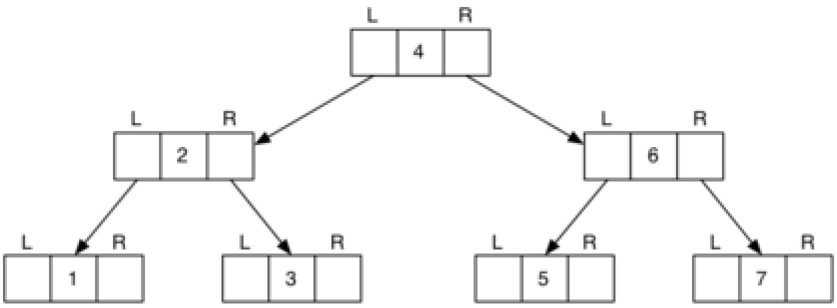
Representing BSTs

17/49

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

18/49

Typical data structures for trees ...

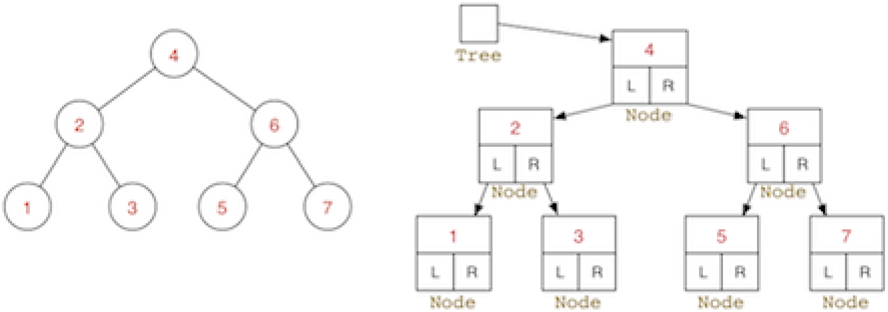
```
// a Tree is represented by a pointer to its root node
typedef struct Node *Tree;

// 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

19/49

Abstract data vs concrete data ...



Tree Algorithms

Searching in BSTs

21/49

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

22/49

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

23/49

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:

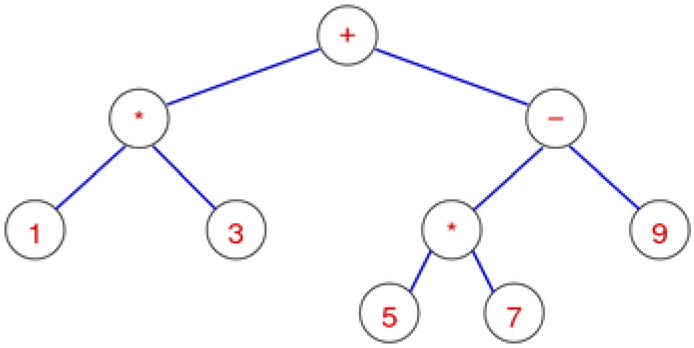
- *preorder* (NLR) ... visit root, then left subtree, then right subtree
- *inorder* (LNR) ... visit left subtree, then root, then right subtree
- *postorder* (LRN) ... visit left subtree, then right subtree, then root
- *level-order* ... visit root, then all its children, then all their children

---

... Tree Traversal

24/49

Consider "visiting" an expression tree like:

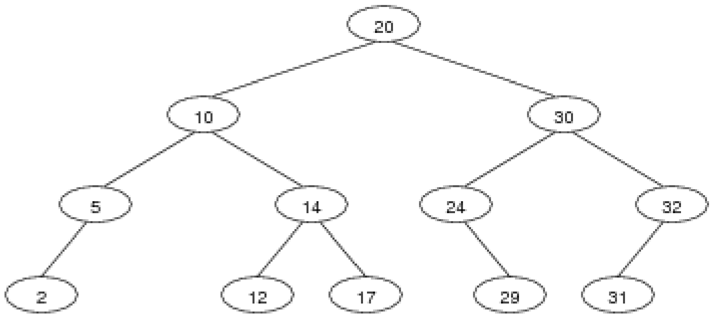


NLR: + * 1 3 - * 5 7 9 (prefix-order: useful for building tree)
LNR: 1 * 3 + 5 * 7 - 9 (infix-order: "natural" order)
LRN: 1 3 * 5 7 * 9 - + (postfix-order: useful for evaluation)
Level: + * - 1 3 * 9 5 7 (level-order: useful for printing tree)

Exercise #2: Tree Traversal

25/49

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

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: `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

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

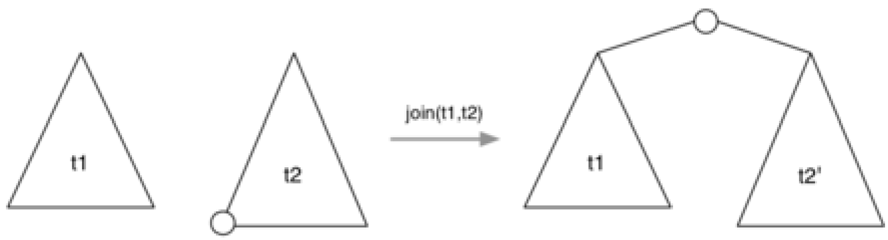
$x \leqslant$ height(t) $\leqslant x+1$, where x = max(height(t_1),height(t_2))

Variation: choose deeper subtree; take root from there.

... Joining Two Trees

31/49

Joining two trees:



Note: t2' may be less deep than t2

... Joining Two Trees

32/49

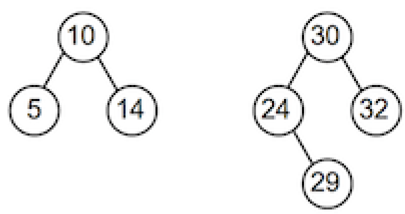
Implementation of tree-join

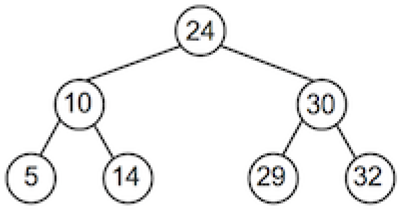
```
joinTrees(t1,t2):
|  Input  trees t1,t2
|  Output t1 and t2 joined together
|
|  if t1 is empty then return t2
|  else if t2 is empty then return t1
|  else
|  |  curr=t2, parent=NULL
|  |  while left(curr) is not empty do     // find min element in t2
|  |     parent=curr
|  |     curr=left(curr)
|  |  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

33/49

Join the trees





Deletion from BSTs

35/49

Insertion into a binary search tree is easy.

Deletion from a binary search tree is harder.

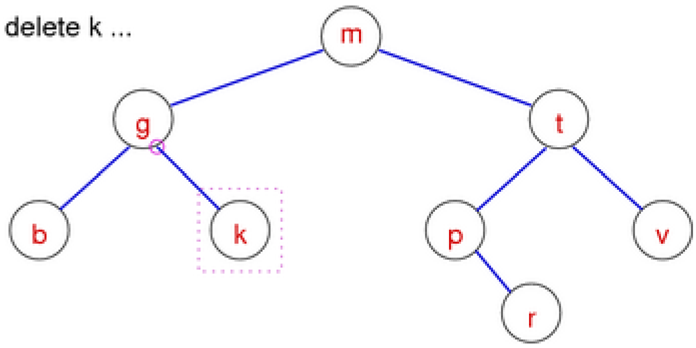
Four cases to consider ...

- 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

36/49

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

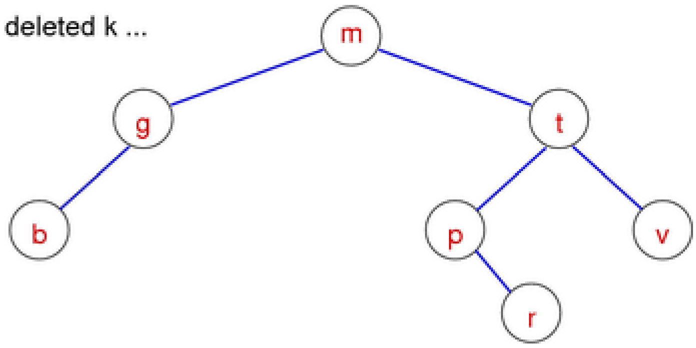


Just delete the item

... Deletion from BSTs

37/49

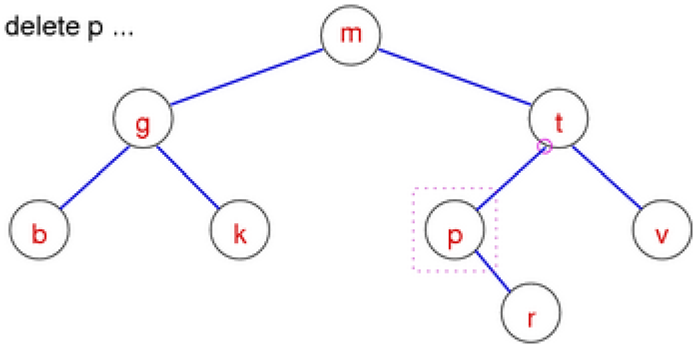
Case 2: item to be deleted is a leaf (zero subtrees)



... Deletion from BSTs

38/49

Case 3: item to be deleted has one subtree

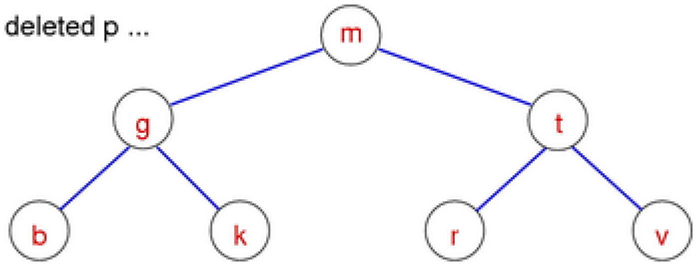


Replace the item by its only subtree

... Deletion from BSTs

39/49

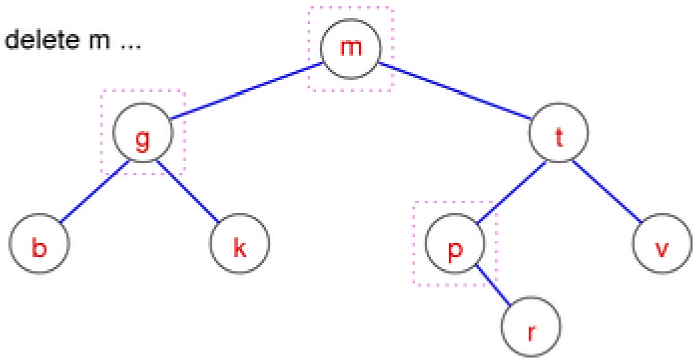
Case 3: item to be deleted has one subtree



... Deletion from BSTs

40/49

Case 4: item to be deleted has two subtrees



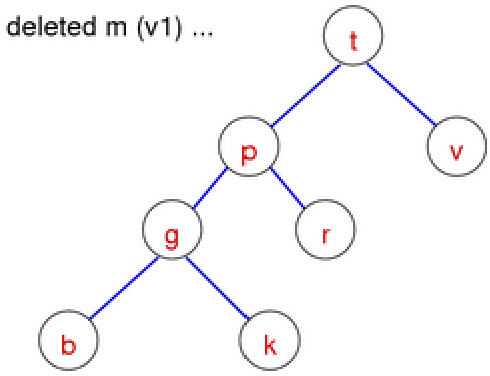
Version 1: right child becomes new root, attach left subtree to min element of right subtree

---

... Deletion from BSTs

41/49

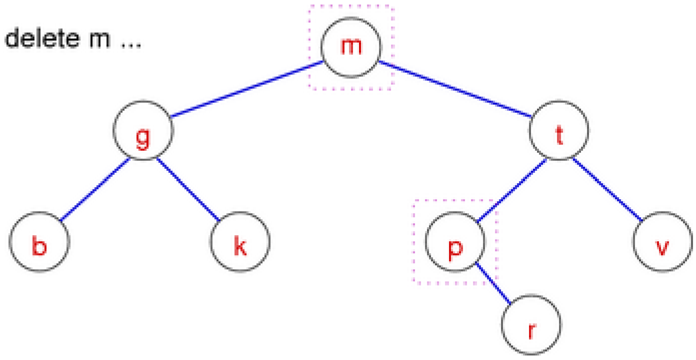
Case 4: item to be deleted has two subtrees



... Deletion from BSTs

42/49

Case 4: item to be deleted has two subtrees

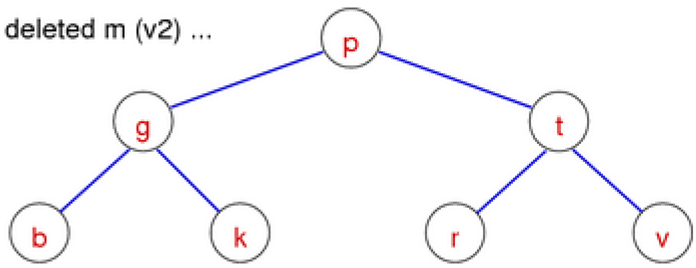


Version 2: *join* left and right subtree

... Deletion from BSTs

43/49

Case 4: item to be deleted has two subtrees



... Deletion from BSTs

44/49

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
```

Application of BSTs: Sets

45/49

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

46/49

Assuming we have `Tree` implementation

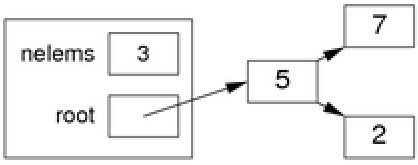
- which precludes duplicate key values
- which implements insertion, search, deletion

then `Set` implementation is

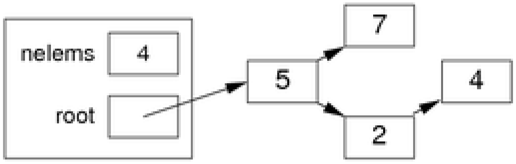
- `SetInsert(Set,Item)` ≡ `TreeInsert(Tree,Item)`
- `SetDelete(Set,Item)` ≡ `TreeDelete(Tree,Item.Key)`
- `SetMember(Set,Item)` ≡ `TreeSearch(Tree,Item.Key)`

... Application of BSTs: Sets

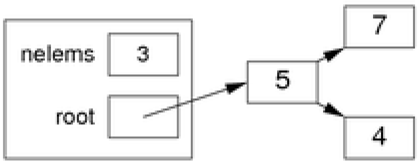
47/49



After SetInsert(s,4):



After SetDelete(s,2):



... Application of BSTs: Sets

48/49

Concrete representation:

```
#include <BSTree.h>

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

typedef Set *SetRep;

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

Summary

49/49

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

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

Produced: 6 Jan 2020