Balancing Search Trees

❖ Balancing Binary Search Trees

Observation: order of insertion into a tree affects its height

- worst case: keys inserted in ascending/descending order
  (effectively have a linked list, so search cost is *O(n)*)
- best case (for at-leaf insertion): keys inserted in pre-order
  (tree height ⇒ search cost is *O(log n)* ; tree is balanced)
- average case: keys inserted in random order
  (tree height ⇒ search cost is *O(log n)*; but cost ≥ best case)

Goal: build binary search trees which have

- minimum height ⇒ minimum worst case search cost

❖ ... Balancing Binary Search Trees

Perfectly-balanced tree with *N* nodes has

- ∀ nodes, `abs(#nodes(LeftSubtree) - #nodes(RightSubtree)) < 2`
- height of $log_2 N \Rightarrow$ 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

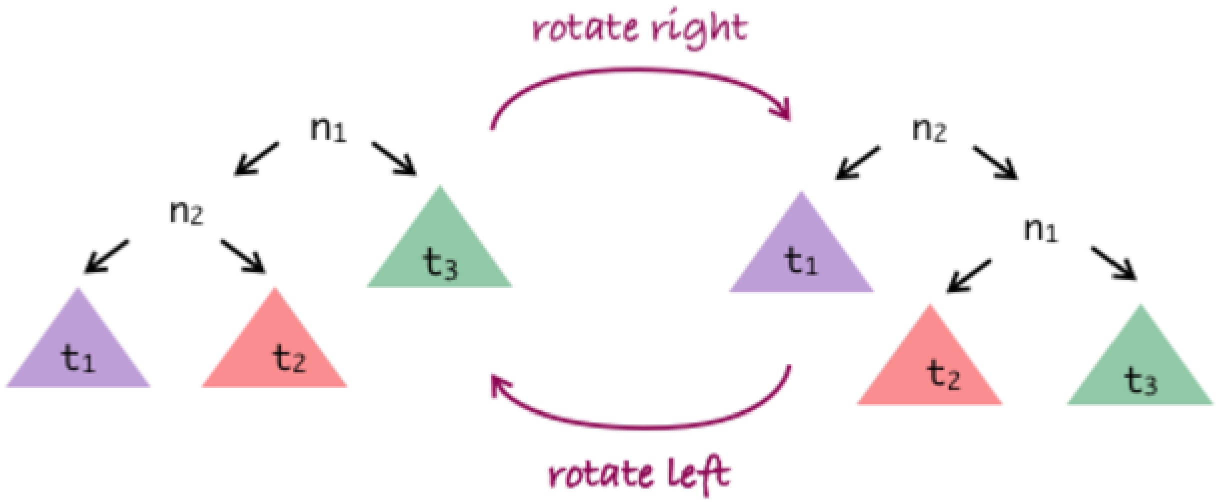
- each new item is added as the new root node

Partition

- rearrange tree around specified node (push it to root)

❖ Tree Rotation

Rotation operations:



Note: tree is ordered, $t_1 < n_2 < t_2 < n_1 < t_3$

❖ ... 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
- n_2's left subtree is unchanged

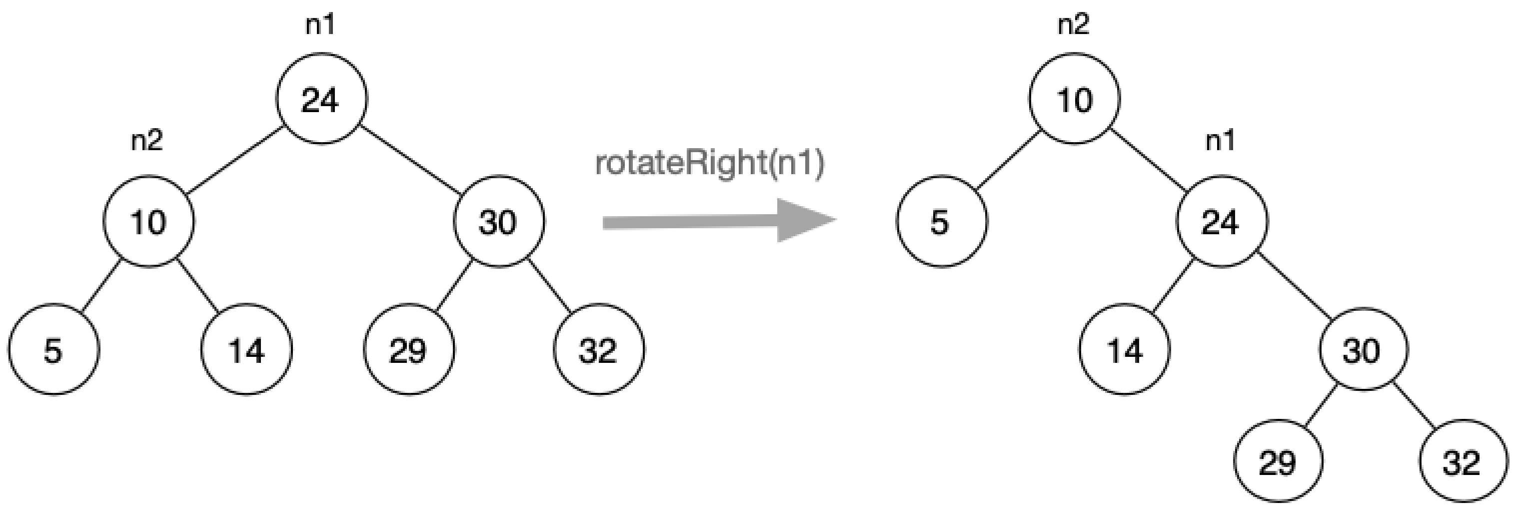
Left rotation: swap left/right in the above.

Rotation requires simple, localised pointer rearrangemennts

Cost of tree rotation: *O(1)*

❖ ... Tree Rotation

Example of right rotation:



❖ ... Tree Rotation

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
```

❖ ... Tree Rotation

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
```

❖ ... Tree Rotation

Cost considerations for tree rotation

- the rotation operation is cheap *O(1)*
- if applied appropriately, will tend to improve tree balance

Sometimes rotation is applied from leaf to root, along one branch

- cost of this is *O(height)*
- payoff is improved balance which reduces height
- reduced height pushes search cost towards *O(log n)*

❖ Insertion at Root

Previous discussion of BSTs did insertion at leaves.

Different approach: insert new item at root.

Potential disadvantages:

- large-scale rearrangement of tree for each insert (apparently)

Potential advantages:

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

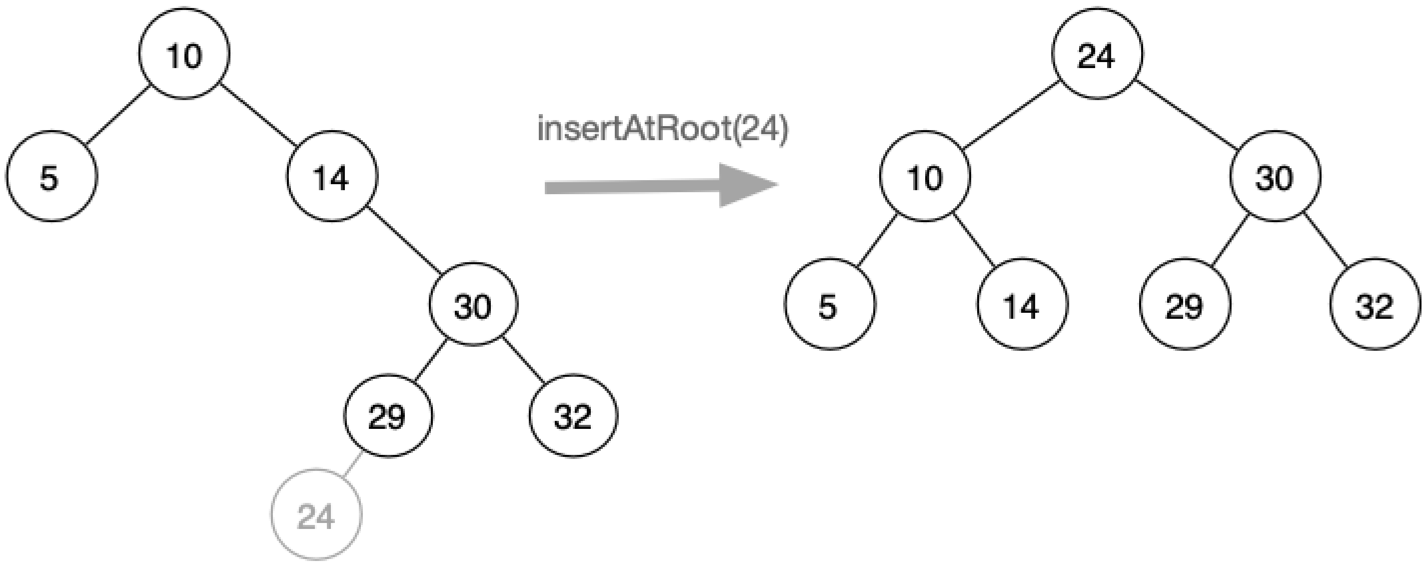
❖ ... Insertion at Root

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

Example of inserting at root:



❖ ... Insertion at Root

Algorithm for inserting at root:

```
insertAtRoot(t, it):
|  Input tree t, item it to be inserted
|  Output modified tree with item at root
|
|  if t is empty tree then
|     t = new node containing item
|  else if item < root(t) then
|     left(t) = insertAtRoot(left(t), it)
|     t = rotateRight(t)
|  else if it > root(t) then
|     right(t) = insertAtRoot(right(t), it)
|     t = rotateLeft(t)
|  end if
|  return t;
```

❖ ... Insertion at Root

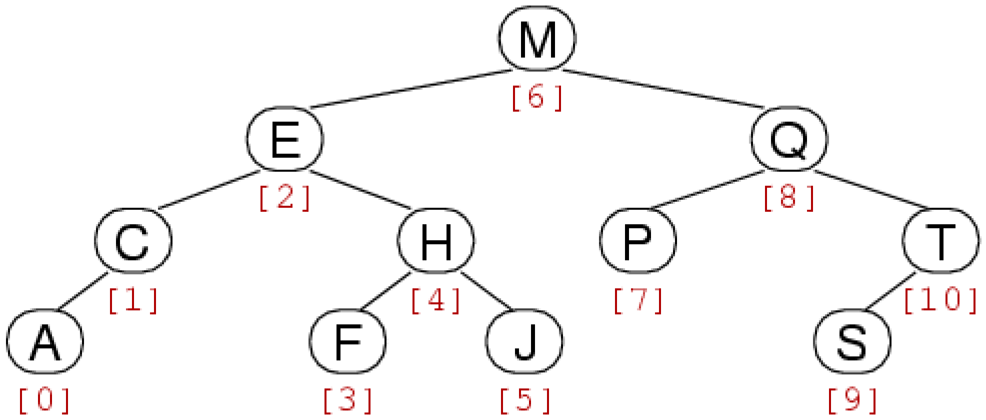
Analysis of insertion-at-root:

- same complexity as for insertion-at-leaf: *O(height)*
    - but cost is effectively doubled ... traverse down, rotate up
- 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

❖ Tree Partitioning

Tree partition operation `partition(tree,i)`

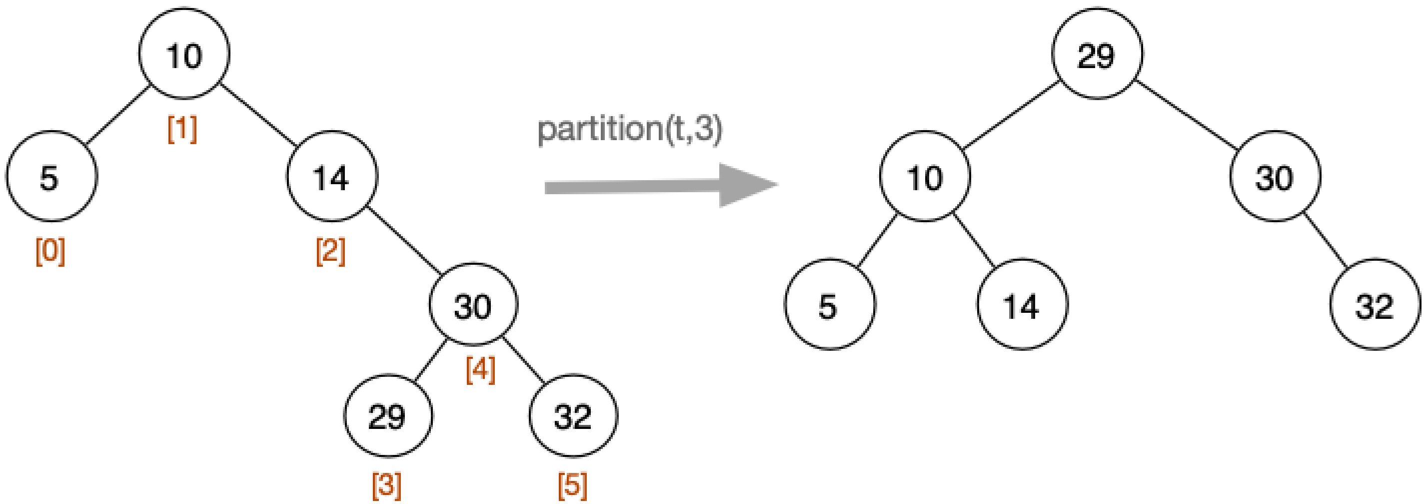
- re-arranges tree so that element with index *i* becomes root



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

❖ ... Tree Partitioning

Example of partition:



❖ ... Tree Partitioning

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

❖ ... Tree Partitioning

Analysis of tree partitioning

- no requirement for search (using element index instead)
- after each recursive partitioning step, one rotation
- overall cost similar to insert-at-root

Benefits

- tends to improve balance $\Rightarrow$ improves search cost

❖ Periodic Rebalancing

An approach to maintaining balance:

- insert at leaves as before; periodically, rebalance the tree

```
| 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
```

When to rebalance? e.g. after every *k* insertions

❖ ... Periodic Rebalancing

A problem with this approach ...

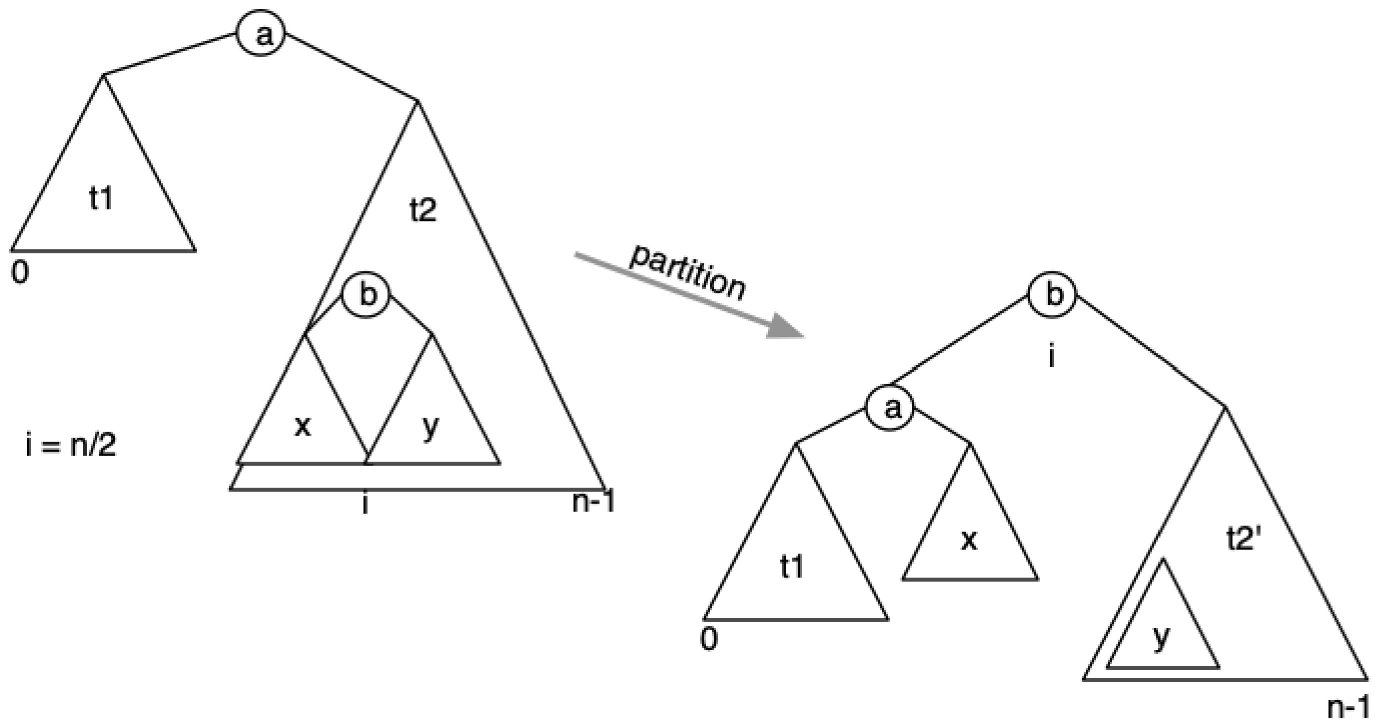
- operation #nodes() has to traverse whole (sub)tree
- to improve efficiency, change node structure

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

But maintaining nnodes requires extra work in other operations

❖ ... Periodic Rebalancing

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



❖ ... Periodic Rebalancing

Implementation of rebalance:

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

❖ ... Periodic Rebalancing

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)

❖ Randomised BST Insertion

Reminder: order of insertion can dramatically affect shape of tree

Tree ADT has no control over order that keys are supplied.

We know that inserting in random order gives $O(\log_2 n)$ search

Can the algorithm itself introduce some randomness?

In the hope that this randomness helps to balance the tree ...

❖ ... Randomised BST Insertion

Approach: normally do leaf insert, randomly do root insert.

```
insertRandom(tree,item)
|  Input  tree, item
|  Output tree with item randomly inserted
|
|  if tree is empty then
|     return new node containing item
|  end if
|  // p/q chance of doing root insert
|  if random() mod q < p then
|     return insertAtRoot(tree,item)
|  else
|     return insertAtLeaf(tree,item)
|  end if
```

E.g. 30% chance ⇒ choose *p=3, q=10*

❖ ... Randomised BST Insertion

Cost analysis:

- similar to cost for inserting keys in random order: *$O(log_2 n)$*
- does not rely on keys being supplied in random order

Approach can also be applied to deletion:

- standard method promotes inorder successor to root
- for the randomised method ...
  - promote inorder successor from right subtree, OR
  - promote inorder predecessor from left subtree

❖ An 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 binary search tree.

❖ ... An Application of BSTs: Sets

Assuming we have BST implementation with type **`Tree`**

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

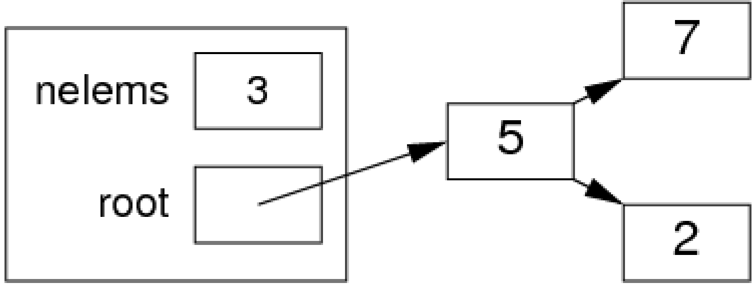
then **`Set`** implementation is

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

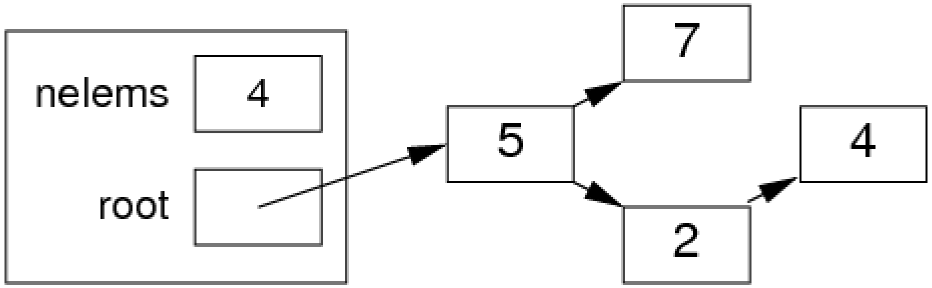
What about union? and intersection?

❖ ... An Application of BSTs: Sets

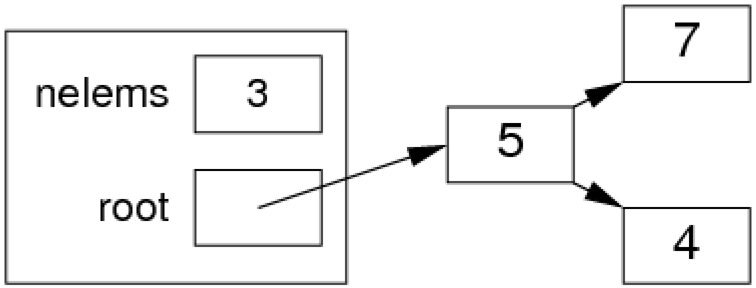
Sets implemented via Trees:



After SetInsert(s,4):



After SetDelete(s,2):



❖ ... An Application of BSTs: Sets

Concrete representation:

```
#include <Tree.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;
}
```

Produced: 8 Jun 2020