Overview

- Homework Problem 2
- Dynamic Search Problems

- REMINDER: Exam 1 is one week from today.
- NOTE: We'll have an in-class review on Wednesday.

DYNAMIC Search Problems

- Input: Set of "keyed" records S
- Operations:
  - Add record to S
  - Delete record from S
  - Find record in S with key=*x* (if one exists)

Dictionary Data Structure

Data structure that supports add, delete, find for set of keyed records.

Binary search tree

Balanced binary search tree

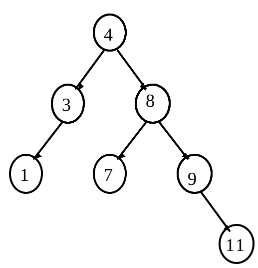
General search tree

Hash Table

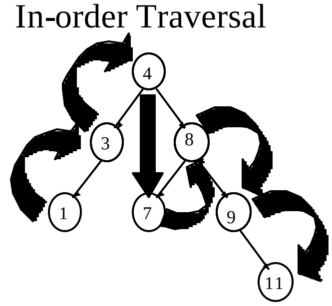
Binary Search Tree for S

- T is a binary tree
- 1-1 relationship between the nodes in T and the records in S
- BST Property: For any node X in T
  - If node Y is in the left subtree of X then Y.key $\leq$ X.key
  - If node Y in the right subtree of X thenY.key $\geq$ X.key

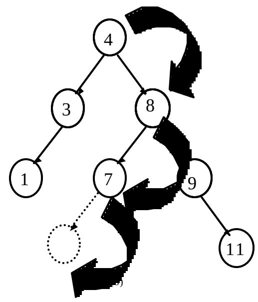
BST



Sorting with BST: In-order Traversal

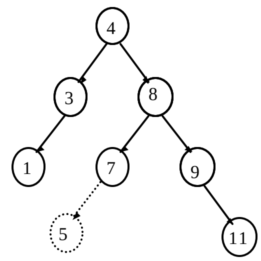


Insert (5)



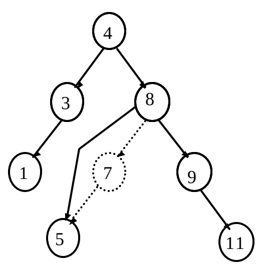
9/25/00 7

Delete(5)
(Leaf is easy)



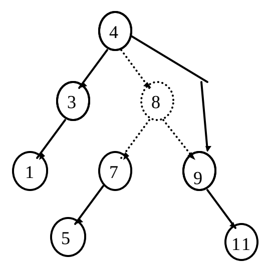
9/25/00 CS140(8) 8

Delete(7)
(Node with 1 child is easy)



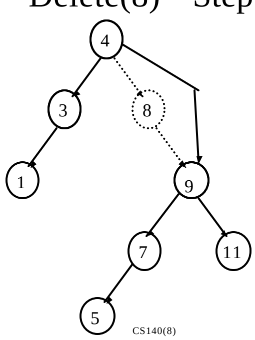
9/25/00 CS140(8) 9

Delete(8) - Step 1



9/25/00 CS140(8) 10

Delete(8) – Step 2



9/25/00 CS140(8) 11

Operation Run Time

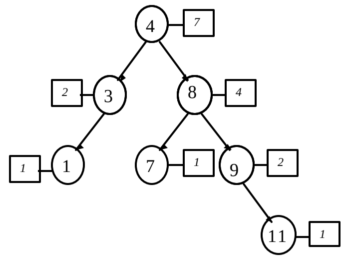
- Search(x) – O(h)
- Insert(x) – O(h)
- Delete(x) – O(h)

9/25/00 CS140(8) 12

Rank

- How can we add rank-finding to the dictionary data structure?
- For each node x in T:
  - count(x) is the number of nodes in the subtree rooted at x.

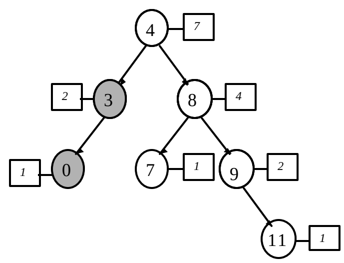
BST with rank capability



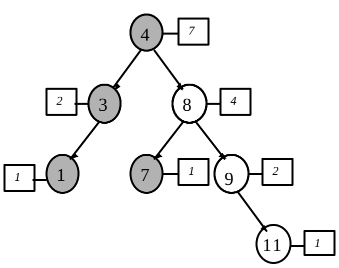
Operation Run Time

• Search(x)	– O(h)
• Insert(x)	– O(h)
• Delete(x)	– O(h)
• Rank(x)	– O(h)
• FindKeyOfRank(i)	– O(h)

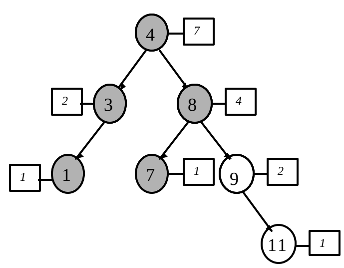
Rank(3)=2



Rank(7)=4

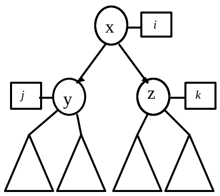


Rank(8)=5



Compute rank at root

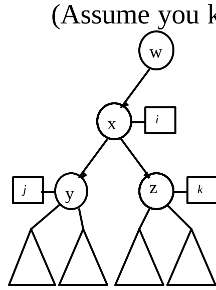
Rank(x)=______



Compute rank at internal node

(Assume you know rank(w))

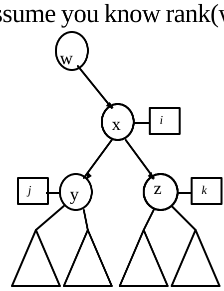
Rank(x)=______



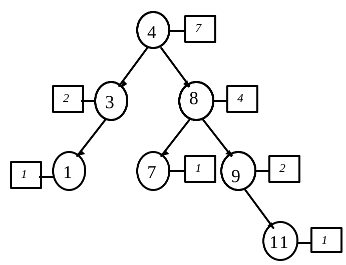
Compute rank at internal node

(Assume you know rank(w))

Rank(x)=______



FindKeyOfRank(5)=8



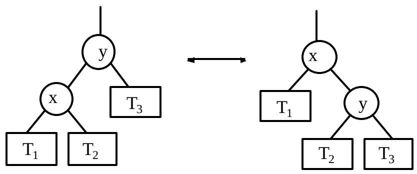
Operation Run Time

- Search(x) – O(h)
- Insert(x) – O(h)
- Delete(x) – O(h)
- Rank(x) – O(h)
- FindKeyOfRank(i) – O(h)

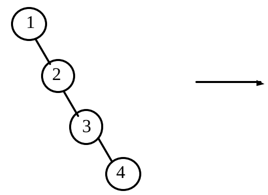
Keeping a good balance …

- Search trees: O(h) time per operation
- “Balanced trees” insure O(log n) time per operation.
- Different approaches to balance:
  – Red/black trees
  – 2-3 trees
  – AVL trees
  – Treaps

Rotations



Example



Different balancing rules…

- Red/black trees
- 2-3 trees
- AVL trees
- Treaps

Treaps: Step 1

- S={1,3,5,7,9 }
- Each element of S is assigned a unique "heap key":

  T=(1,15), (3,10), (5,30), (7,0), (9,25)

Treaps: Step 2

- S={1,3,5,7,9 }
- Each element of S is assigned a unique "H key":

  T=(1,15), (3,10), (5,30), (7,0), (9,25)
- Build tree where
  - S-key satisfy BST Property
  - H-key satisfy Heap Property

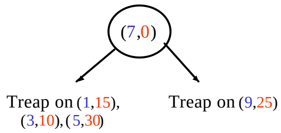
Treap

T=(1,15), (3,10), (5,30), (7,0), (9,25)

- Root:
- Left subtree:
- Right subtree:

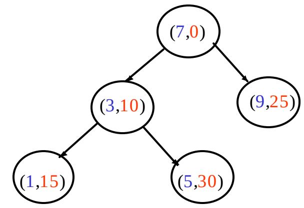
Treap

T=(1,15), (3,10), (5,30), (7,0), (9,25)



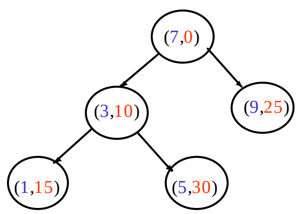
Treap

T=(1,15), (3,10), (5,30), (7,0), (9,25)



Treap

T=(1,15), (3,10), (5,30), (7,0), (9,25)



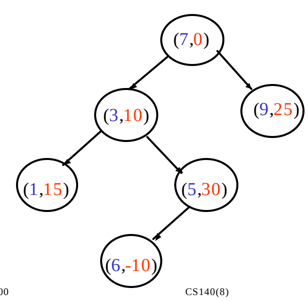
Treaps

- Claim: If the heap keys are unique then the treap is unique.
- Proof:

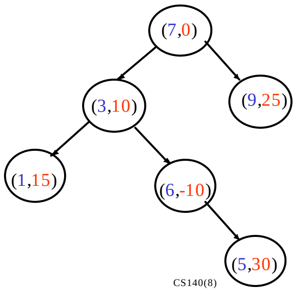
Treaps

- Claim: If the heap keys are chosen uniformly at random from [-B,B], where B>>n, then
  - With high probability the keys will be unique.
  - The expected height of the treap is O(lg(n)).

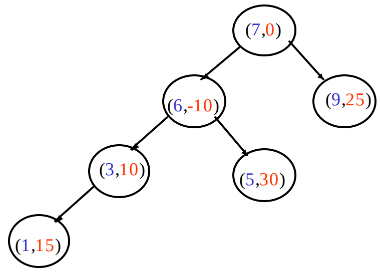
Treap: Insert (6,-10)



Treap: Rotate



Treap: Rotate



Treap: Rotate