Algorithms and Data
Structures
Lecture VII

This Lecture

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

Dictionaries

- *Dictionary* ADT – a dynamic set with methods:
  - **Search(S, k)** *– a query method that returns a pointer x to an element where x.key = k*
  - **Insert(S, x)** *– a modifier method that adds the element pointed to by x to S*
  - **Delete(S, x)** *– a modifier method that removes the element pointed to by x from S*
- An element has a *key* part and a *satellite data* part

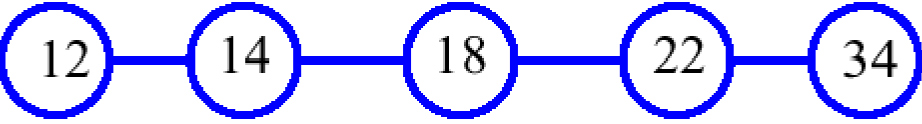
Ordered Dictionaries

- In addition to dictionary functionality, we want to support priority-queue-type operations:
  - **Min(S)**
  - **Max(S)**
- *Partial-order* supported by priority-queues is not enough. We want to support
  - **Predecessor(S, k)**
  - **Successor(S, k)**

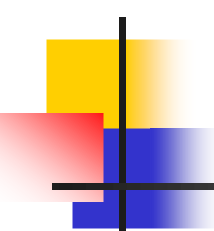
A List-Based Implementation

- Unordered list
  - searching takes $O(n)$ time
  - inserting takes $O(1)$ time

- Ordered list

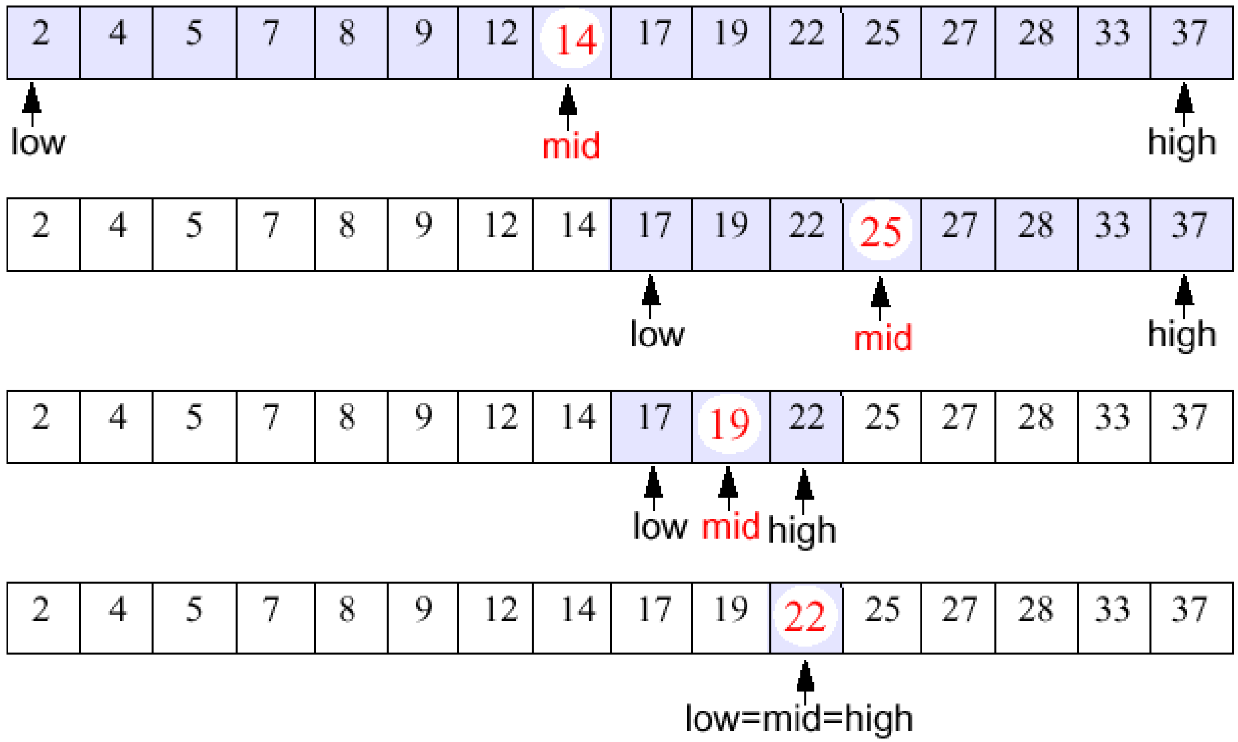


  - searching takes $O(n)$ time
  - inserting takes $O(n)$ time
  - Using array would definitely improve search time.



Binary Search

- Narrow down the search range in stages
  - findElement(22)



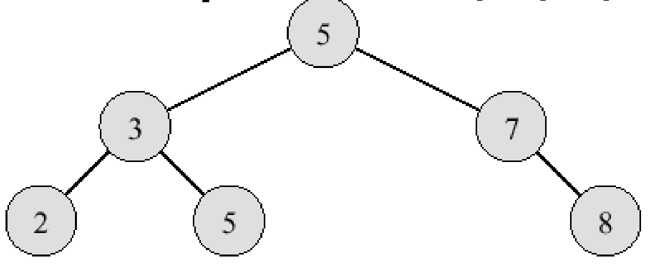
Running Time

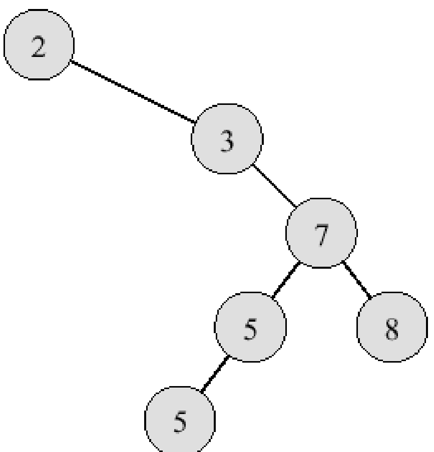
- The range of candidate items to be searched is halved after comparing the key with the middle element
- Binary search runs in $O(\lg n)$ time (remember recurrence...)

Comparisons performed	Size of array range to search
0	N
1	$N/2$
i	$N/2^i$
$log_2 i$	1

Binary Search Trees

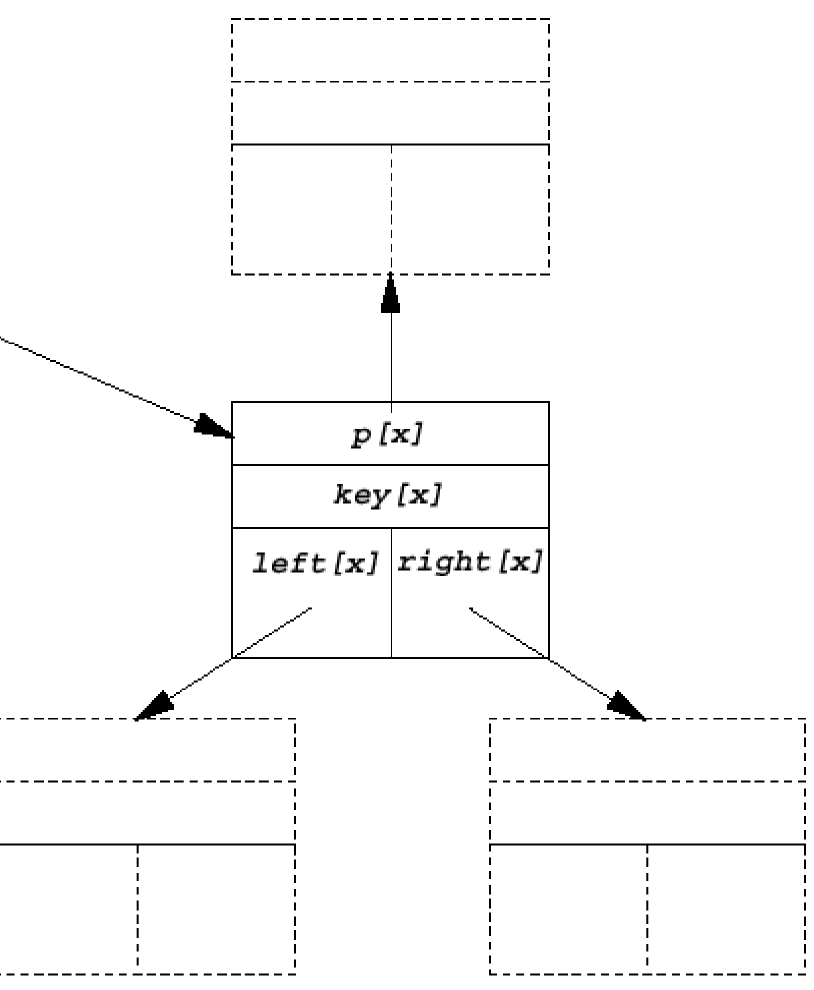
- A binary search tree is a binary tree T such that
  - each internal node stores an item (k,e) of a dictionary
  - keys stored at nodes in the **left subtree** of v are **less than or equal** to k
  - keys stored at nodes in the **right subtree** of v **are greater than or equal** to k
- Example sequence 2,3,5,5,7,8





The Node Structure

- Each node in the tree contains
  - *key*[*x*] – key
  - *left*[*x*] – pointer to left child
  - *right*[*x*] – pt. to right child
  - *p*[*x*] – pt. to parent node



Tree Walks

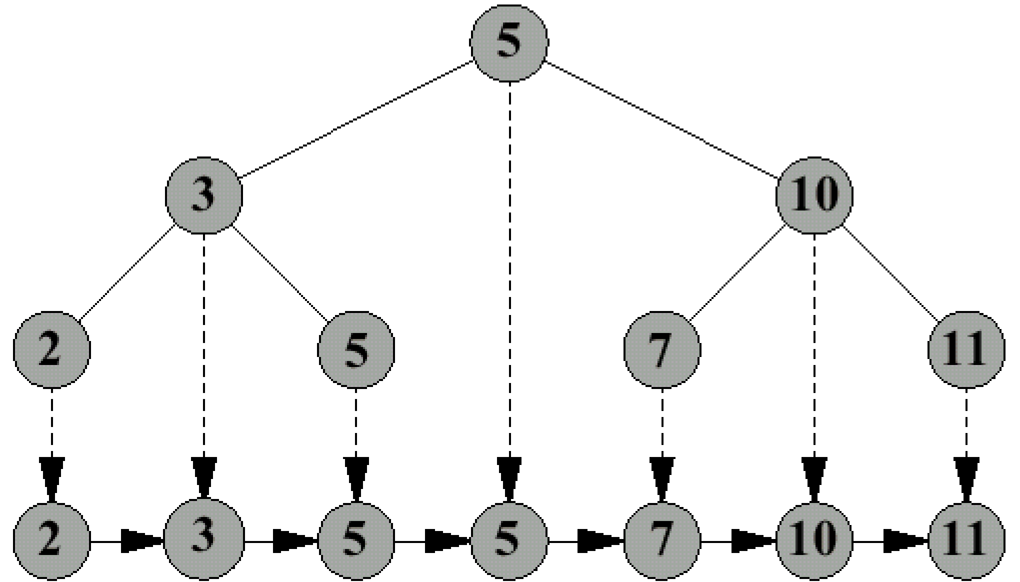
- Keys in the BST can be printed using "tree walks"
- Keys of each node printed between keys in the left and right subtree – *inroder* tree traversal

```
InorderTreeWalk(x)
01 if x ≠ NIL then
02     InorderTreeWalk(left[x])
03     print key[x]
04     InorderTreeWalk(right[x])
```

- Prints elements in monotonically increasing order
- Running time $\Theta(n)$

Tree Walks (2)

- ITW can be thought of as a projection of the BST nodes onto a one dimensional interval



Tree Walks (3)

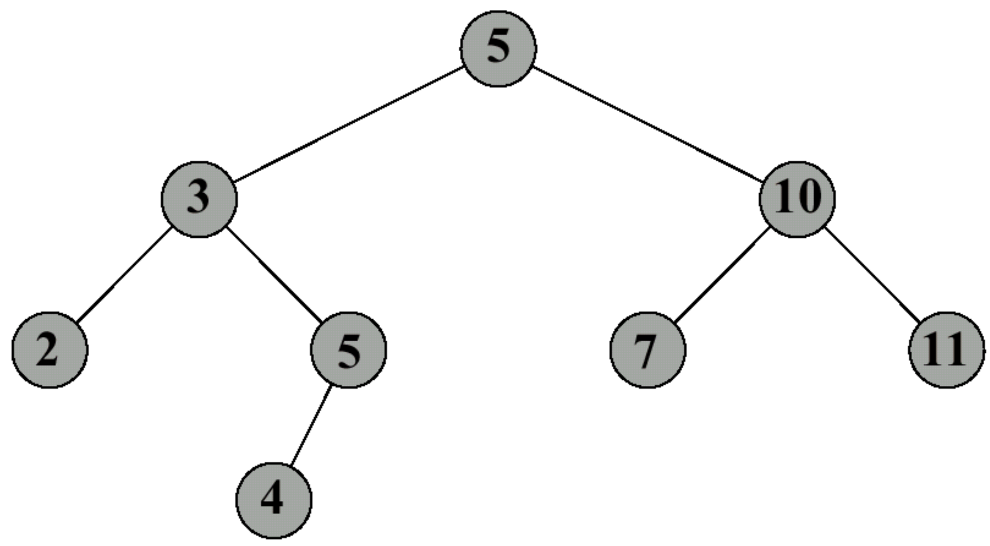
- A preorder tree walk processes each node before processing its children
- A postorder tree walk processes each node after processing its children

Tree Walks (4)

- printing an arithmetic expression; specialization of an inorder traversal (infix, postfix notation)
  - print "(" before traversing the left subtree
  - print ")" after traversing the right subtree

((((3 × (1 + (4 + 6))) + (2 + 8)) × 5) + (4 × (7 + 2)))

Searching a BST



- To find an element with key *k* in a tree *T*
  - compare *k* with key[root[*T*]]
  - if *k* < key[root[*T*]], search for *k* in left[root[*T*]]
  - otherwise, search for *k* in right[root[*T*]]

Pseudocode for BST Search

- Recursive version

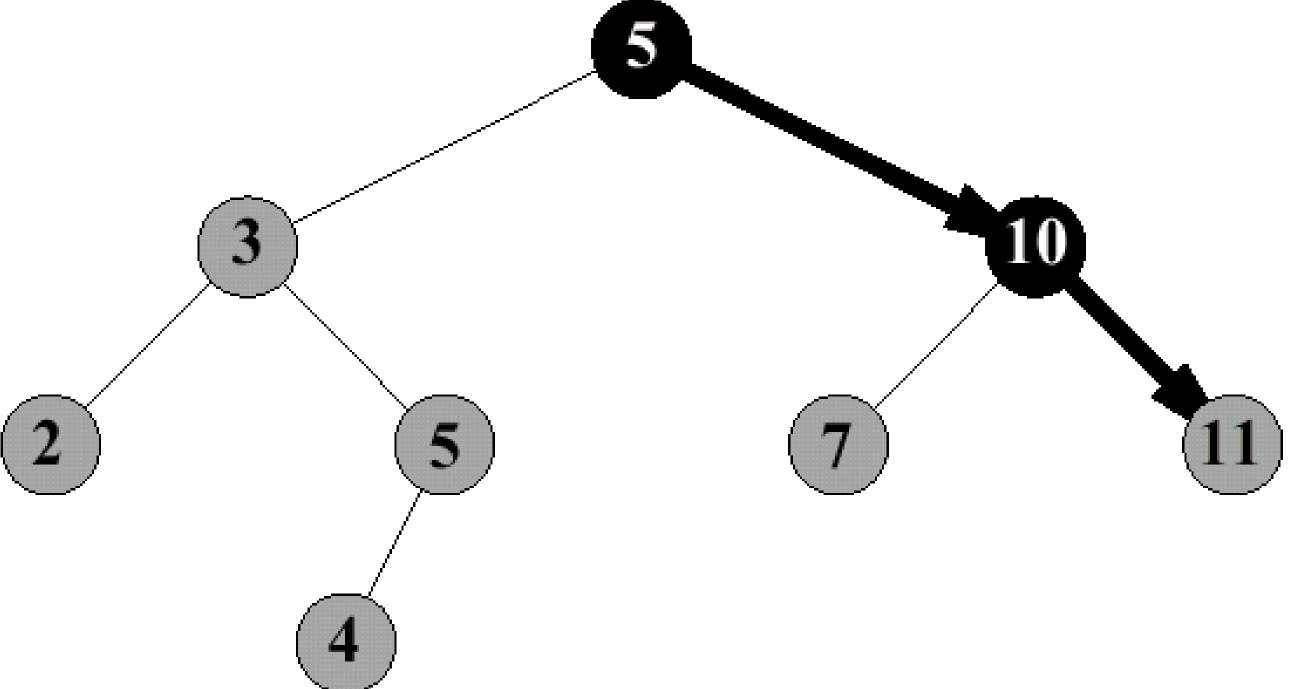
```
Search(T,k)
01 x ← root[T]
02 if x = NIL then return NIL
03 if k = key[x] then return x
04 if k < key[x]
05   then return Search(left[x],k)
06   else return Search(right[x],k)
```

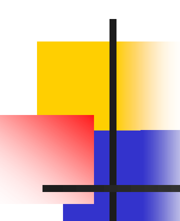
- Iterative version

```
Search(T,k)
01 x ← root[T]
02 while x ≠ NIL and k ≠ key[x] do
03   if k < key[x]
04     then x ← left[x]
05     else x ← right[x]
06 return x
```

Search Examples

- Search(*T*, 11)



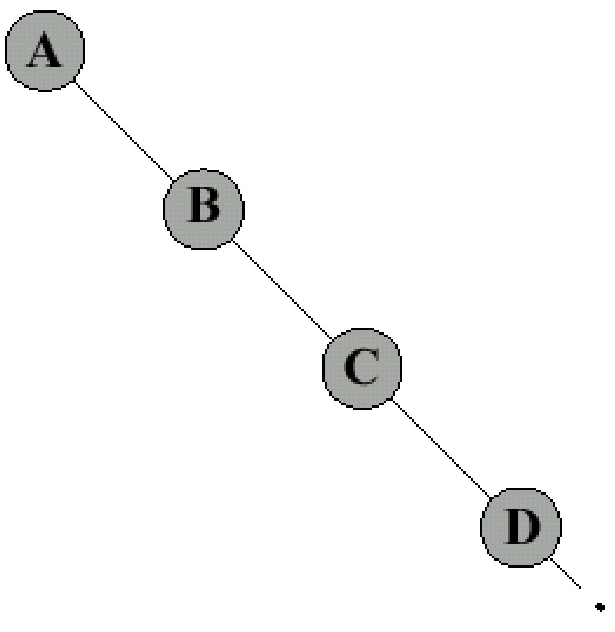


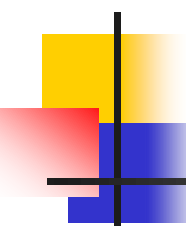
Search Examples (2)

- Search(T, 6)

Analysis of Search

- Running time on tree of height h is $O(h)$
- After the insertion of n keys, the worst-case running time of searching is $O(n)$





BST Minimum (Maximum)

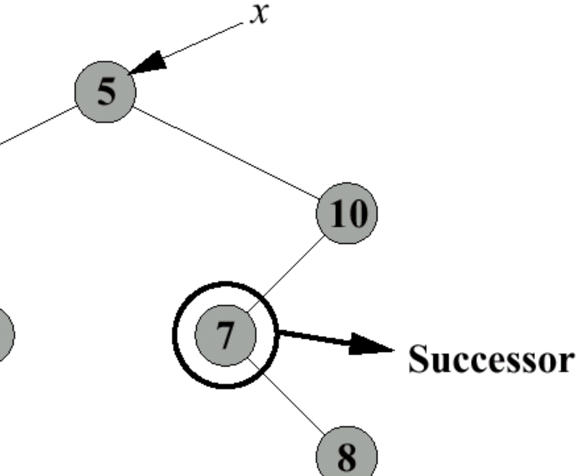
- Find the minimum key in a tree rooted at x (compare to a solution for heaps)

```
TreeMinimum(x)
01 while left[x] ≠ NIL
02    do x ← left[x]
03 return x
```

- Running time $O(h)$, i.e., it is proportional to the height of the tree

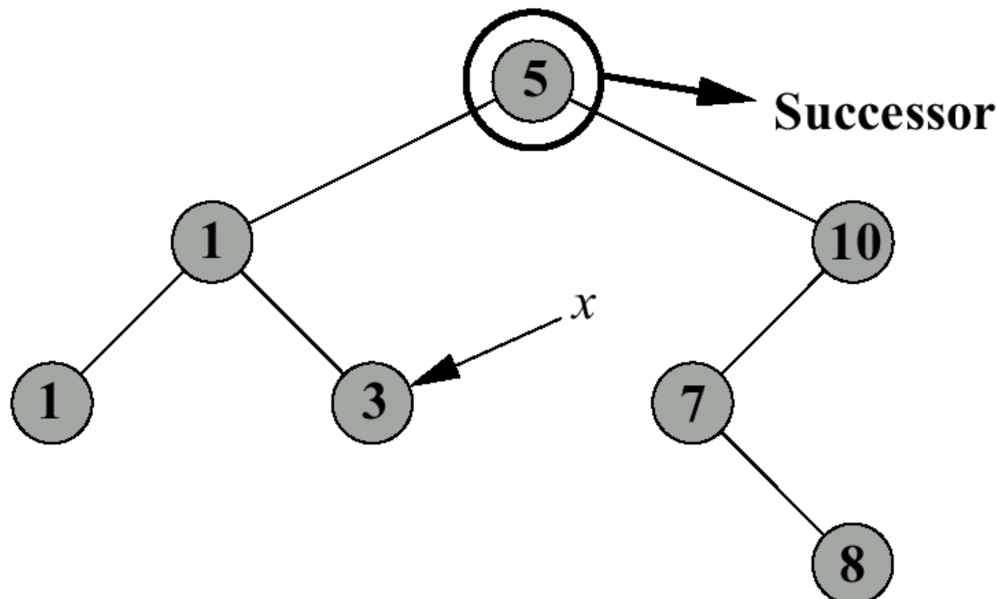
Successor

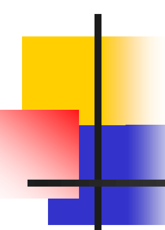
- Given *x*, find the node with the smallest key greater than key[*x*]
- We can distinguish two cases, depending on the right subtree of *x*
- Case 1
  - right subtree of x is nonempty
  - successor is leftmost node in the right subtree (Why?)
  - this can be done by returning TreeMinimum(right[x])



Successor (2)

- Case 2
  - the right subtree of *x* is empty
  - successor is the lowest ancestor of *x* whose left child is also an ancestor of *x* (Why?)





Successor Pseudocode

```
TreeSuccessor(x)
01 if right[x] ≠ NIL
02   then return TreeMinimum(right[x])
03 y ← p[x]
04 while y ≠ NIL and x = right[y]
05   x ← y
06   y ← p[y]
03 return y
```

- For a tree of height h, the running time is $O(h)$

BST Insertion

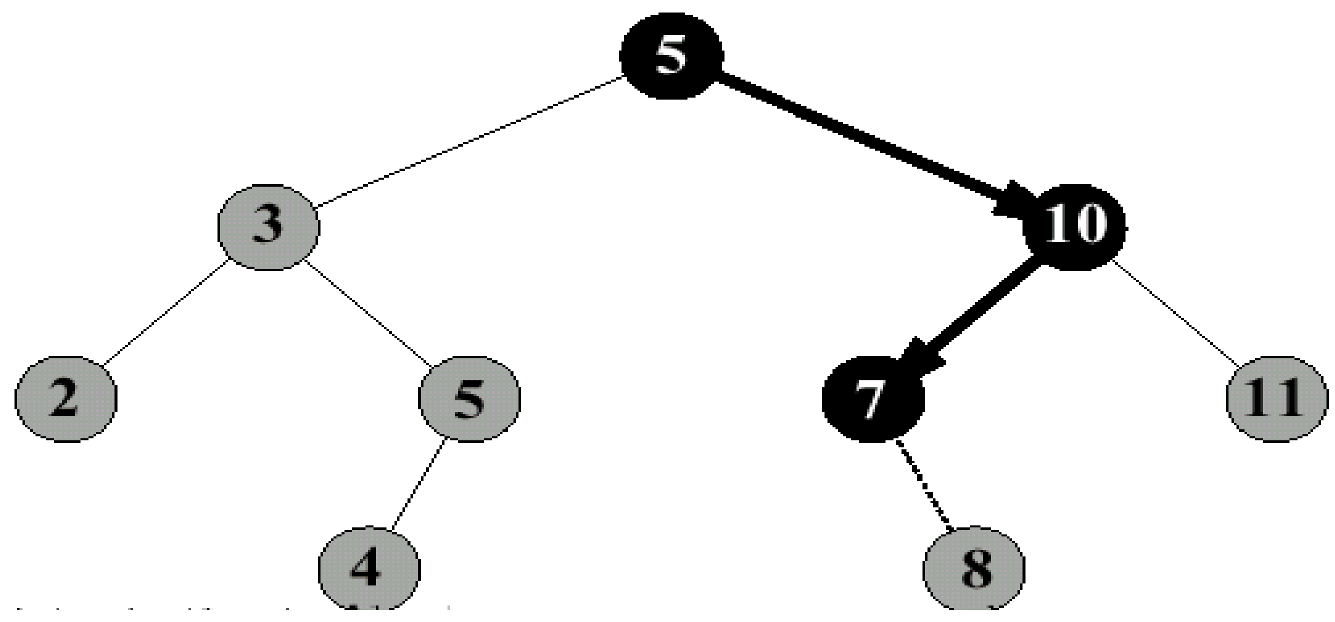
- The basic idea is similar to searching
  - take an element z (whose left and right children are NIL) and insert it into T
  - find place in T where z belongs (that's similar to search…),
  - and add z there
- The running on a tree of height h is $O(h)$, i.e., it is proportional to the height of the tree

BST Insertion Pseudo Code

```
TreeInsert(T,z)
01 y ← NIL
02 x ← root[T]
03 while x ≠ NIL
04   y ← x
05   if key[z] < key[x]
06     then x ← left[x]
07     else x ← right[x]
08 p[z] ← y
09 if y = NIL
10   then root[T] ← z
11   else if key[z] < key[y]
12     then left[y] ← z
13     else right[y] ← z
```

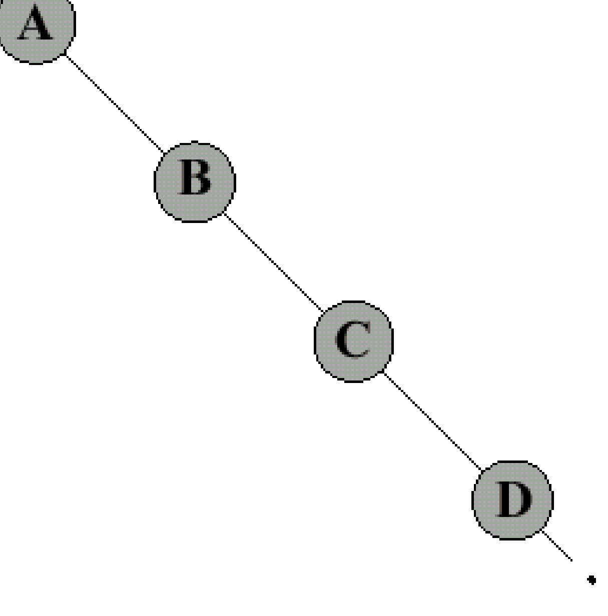
BST Insertion Example

- Insert 8



BST Insertion: Worst Case

- In what kind of sequence should the insertions be made to produce a BST of height n?



BST Sorting

- Use TreeInsert and InorderTreeWalk to sort list of *n* elements, *A*

```
TreeSort(A)
01 root[T] ← NIL
02 for i ← 1 to n
03   TreeInsert(T,A[i])
04 InorderTreeWalk(root[T])
```

BST Sorting (2)

- Sort the following numbers
  5 10 7 1 3 1 8
- Build a binary search tree

5 → 5, 10 → 5, 10, 7 . . .

- Call InorderTreeWalk

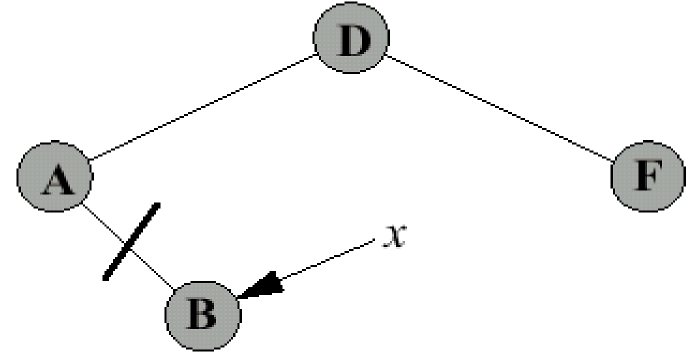
1 1 3 5 7 8 10

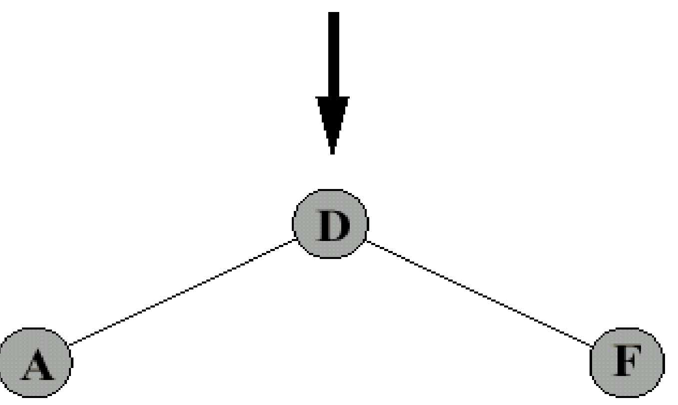
Deletion

- Delete node *x* from a tree *T*
- We can distinguish three cases
  - *x* has no children
  - *x* has one child
  - *x* has two children

Deletion Case 1

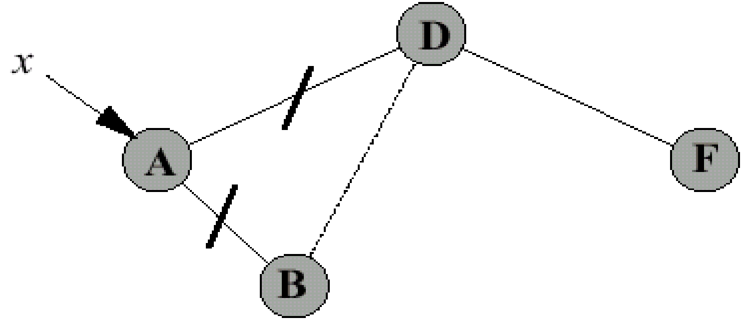
- If *x* has no children – just remove *x*

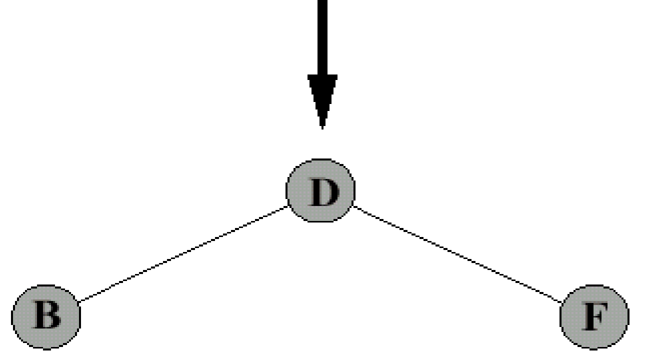




Deletion Case 2

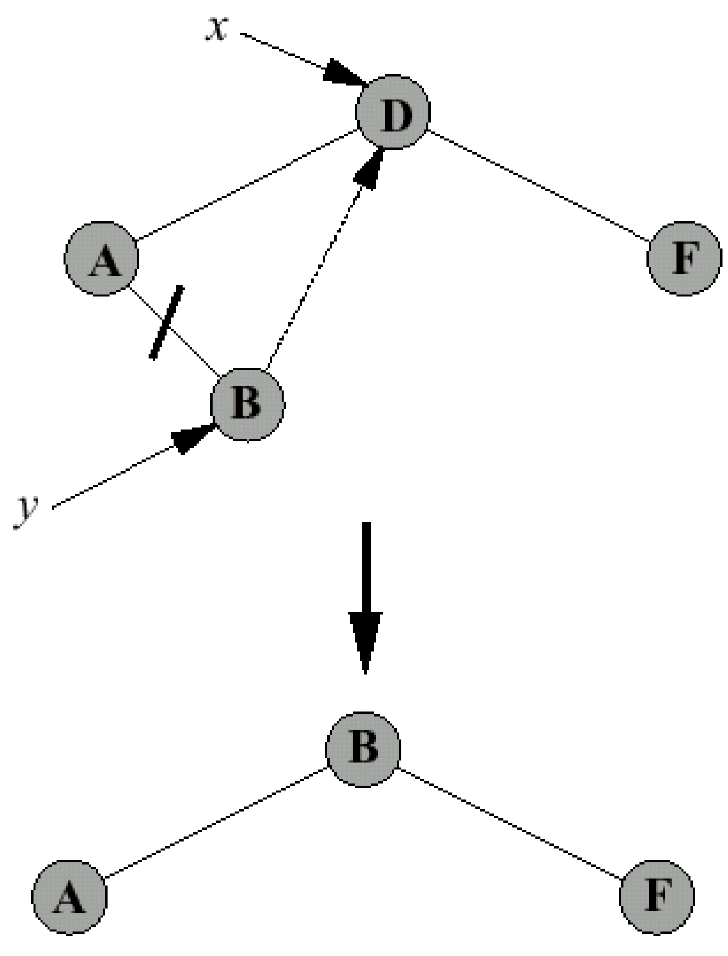
- If *x* has exactly one child, then to delete *x*, simply make p[x] point to that child





Deletion Case 3

- If *x* has two children, then to delete it we have to
  - find its successor (or predecessor) *y*
  - remove y (note that y has at most one child – why?)
  - replace x with y

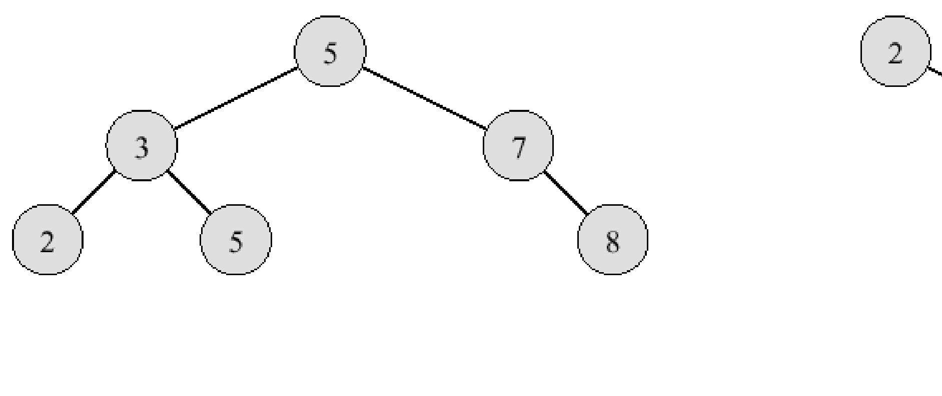


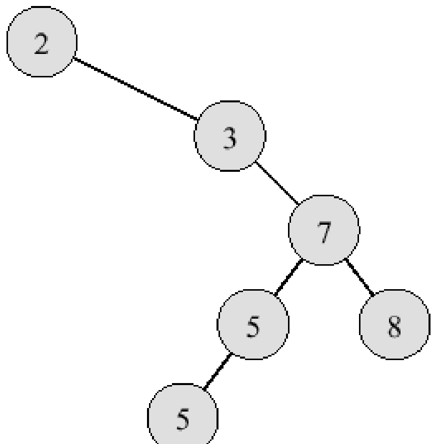
Delete Pseudocode

```
TreeDelete(T,z)
01 if left[z] = NIL or right[z] = NIL
02   then y ← z
03   else y ← TreeSuccessor(z)
04 if left[y] ≠ NIL
05   then x ← left[y]
06   else x ← right[y]
07 if x ≠ NIL
08   then p[x] ← p[y]
09 if p[y] = NIL
10   then root[T] ← x
11   else if y = left[p[y]]
12     then left[p[y]] ← x
13     else right[p[y]] ← x
14 if y ≠ z
15   then key[z] ← key[y] //copy all fileds of y
16 return y
```

Balanced Search Trees

- Problem: worst-case execution time for dynamic set operations is $\Theta(n)$
- Solution: balanced search trees guarantee small height!





Next lecture

- Balanced Binary Search Trees:
  - Red-Black Trees