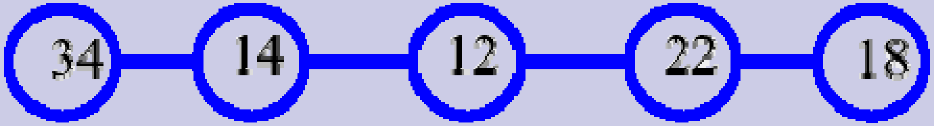
Ordered Dictionaries

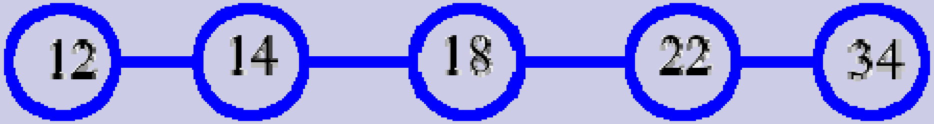
- In addition to dictionary functionality, we want to support following operations:
  - **Min()**
  - **Max()**
  - **Predecessor(S, k)**
  - **Successor(S, k)**
- For this we require that there should be a total order on the keys.

A List-Based Implementation

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

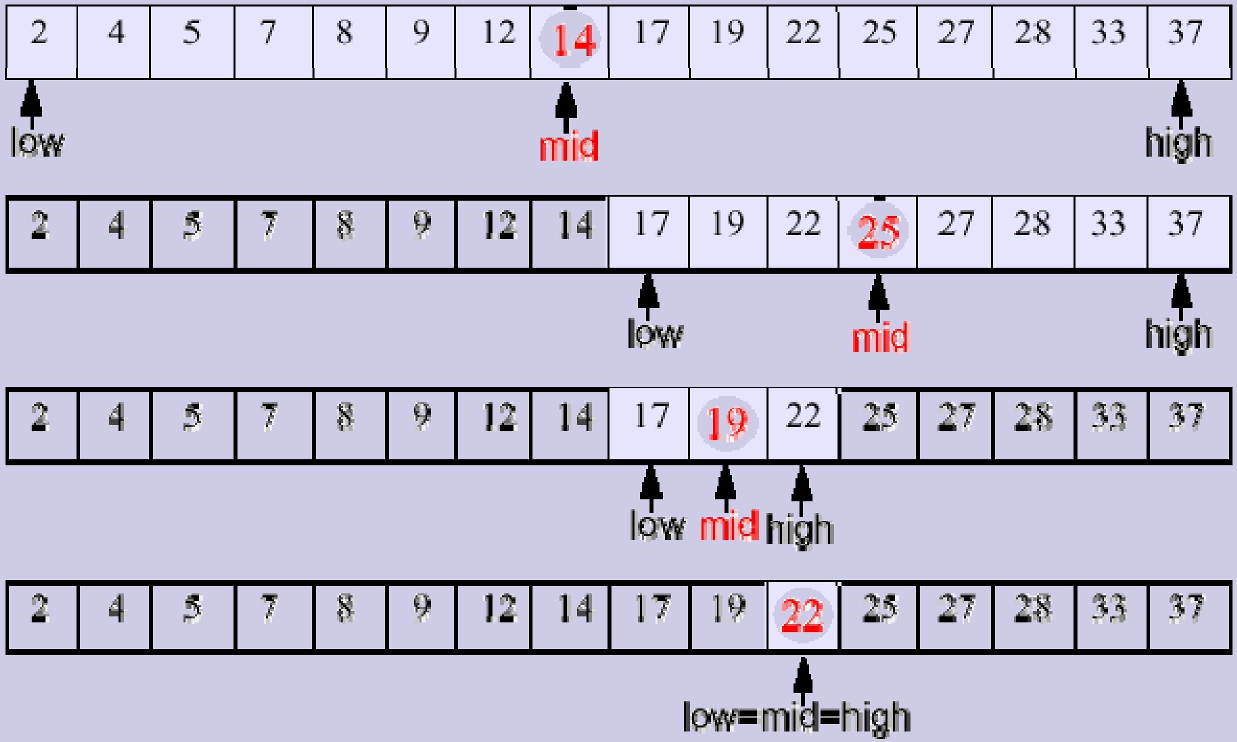


- Ordered list
  - searching takes $O(n)$ time
  - inserting takes $O(n)$ time
  - Using array would definitely improve search time.



Binary Search

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

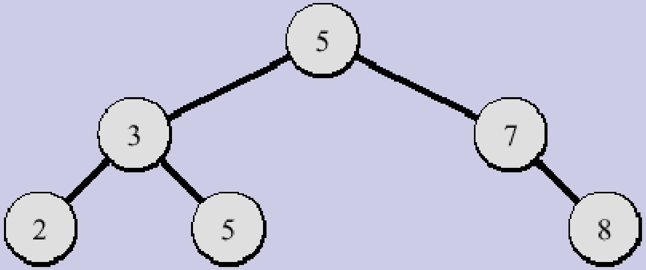


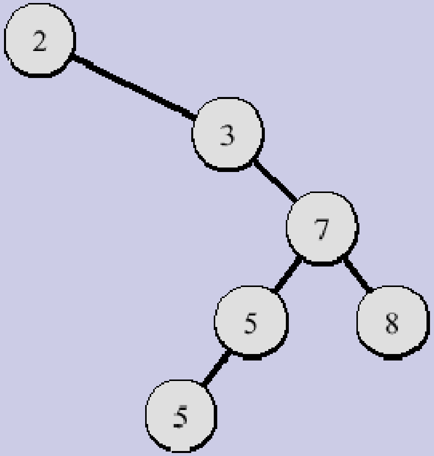
Running Time

- 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...)
- What about insertion and deletion?

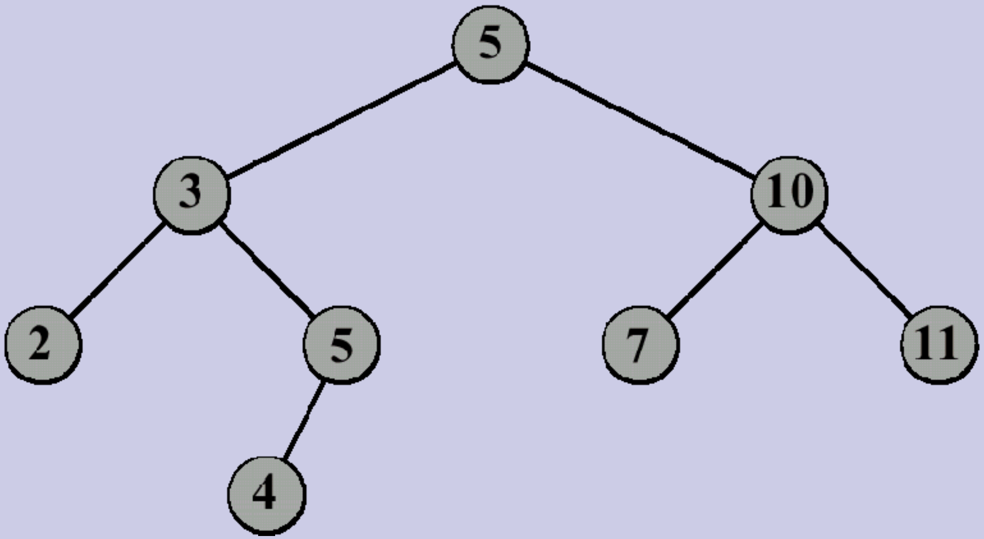
Binary Search Trees

- 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





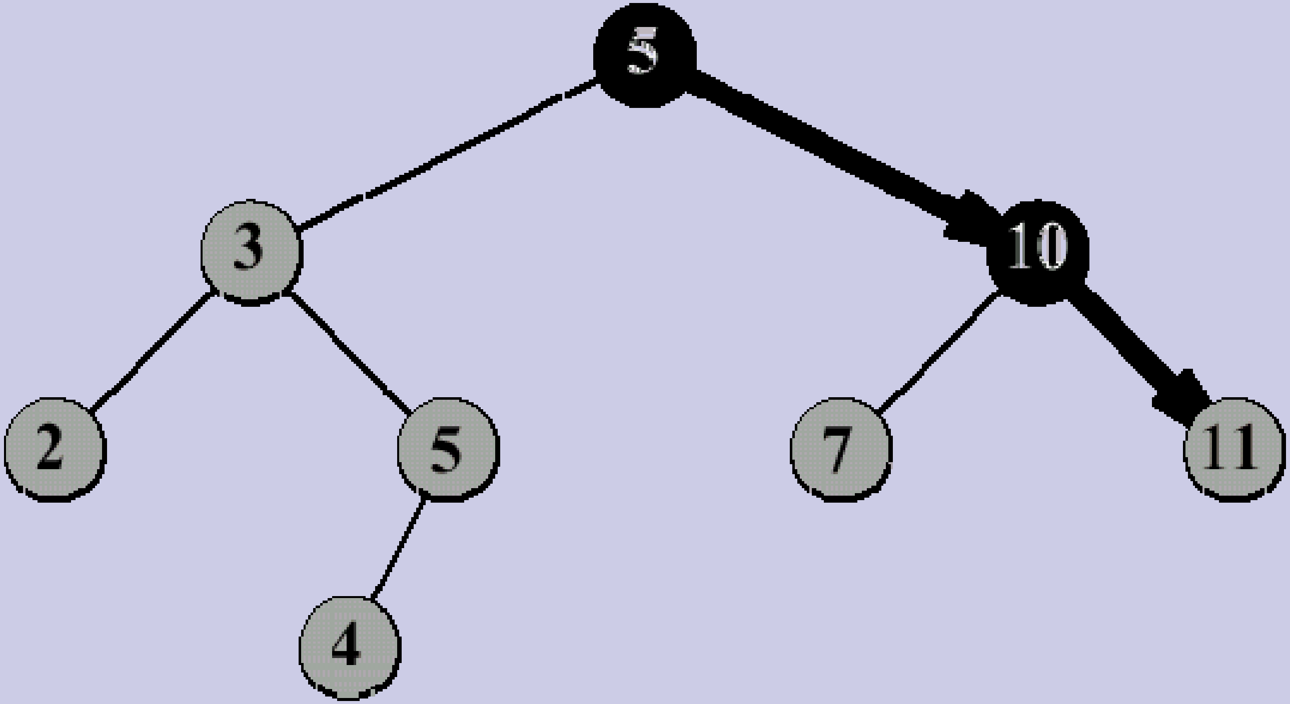
Searching a BST



- 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*]]

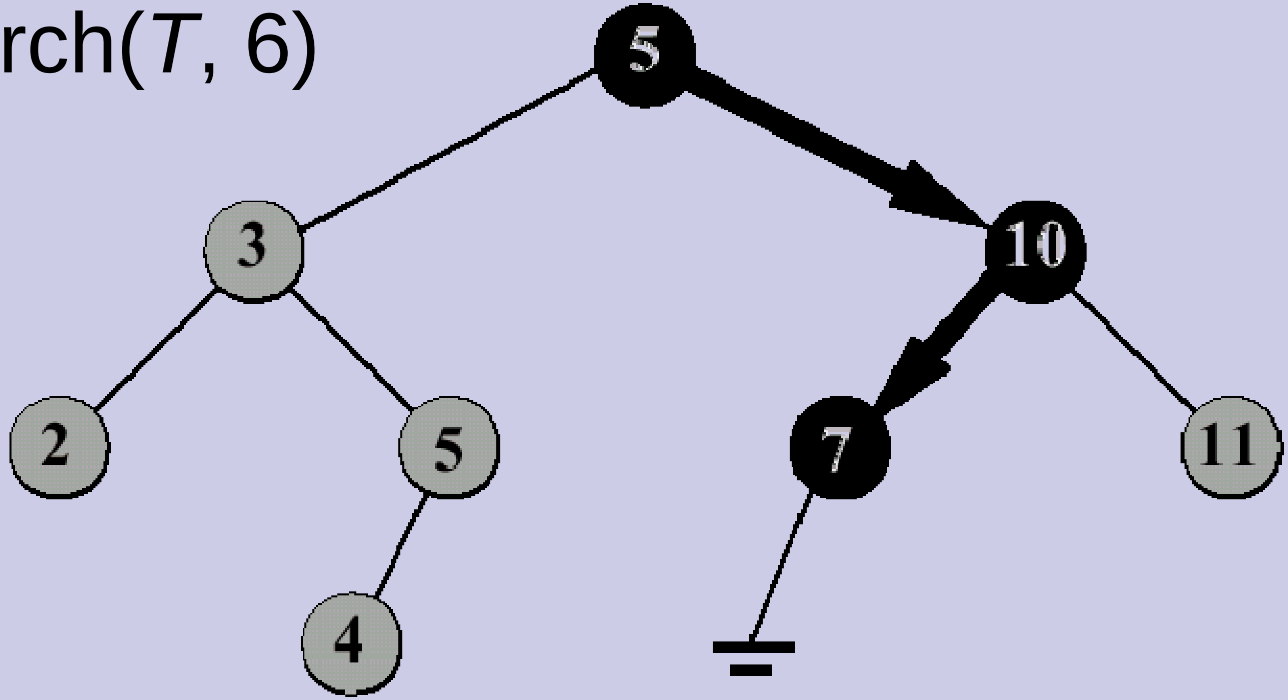
Search Examples

- Search(*T*, 11)



Search Examples (2)

- Search(T, 6)



Pseudocode for BST Search

- Recursive version

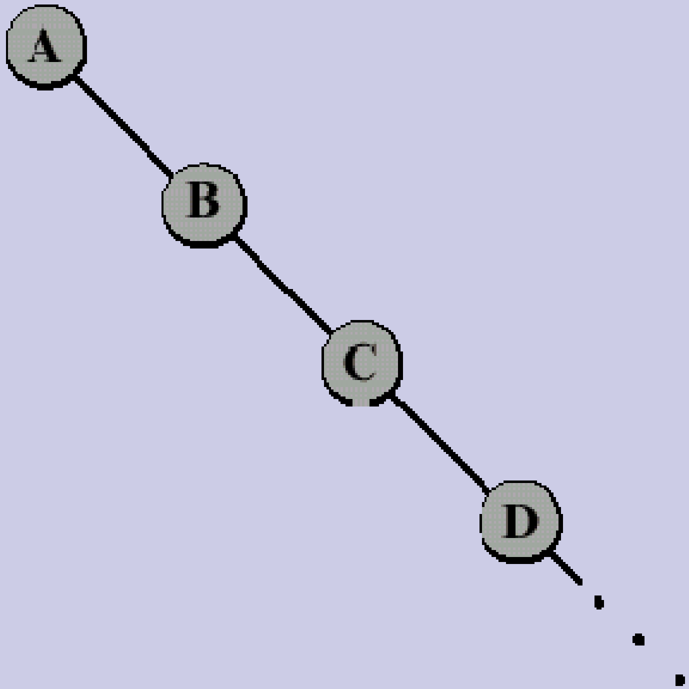
```
Search(T,k)
x ← root[T]
if x = NIL then return NIL
if k = key[x] then return x
if k < key[x]
  then return Search(left[x],k)
  else return Search(right[x],k)
```

- Iterative version

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

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)

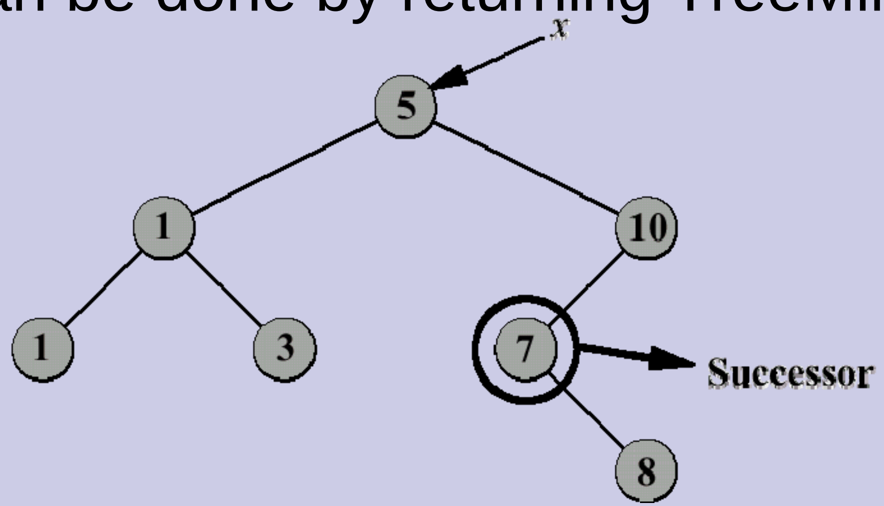
- Find the minimum key in a tree rooted at x

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

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

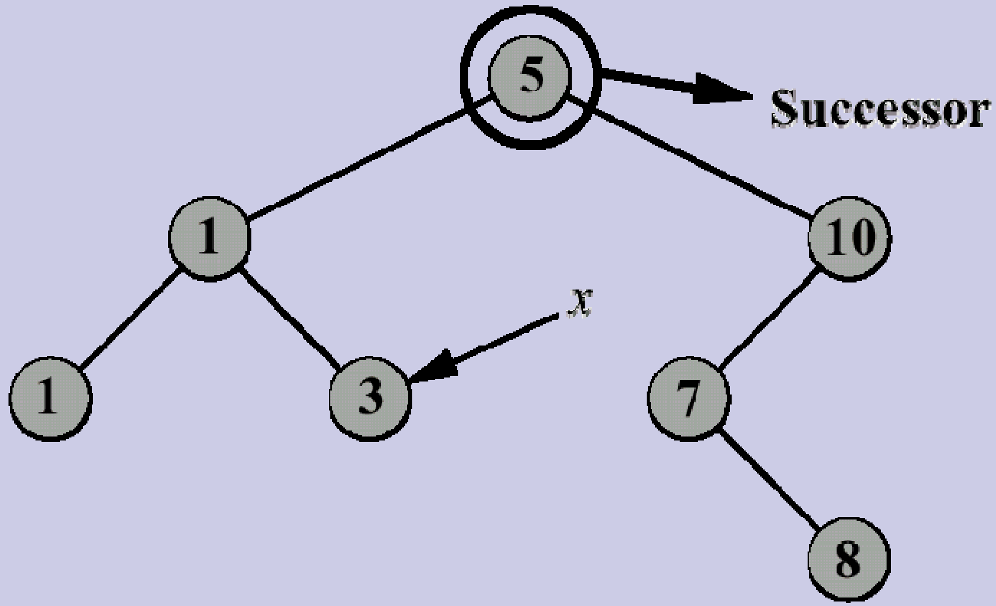
Successor

- 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)
if right[x] ≠ NIL
  then return TreeMinimum(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
  x ← y
  y ← p[y]
return y
```

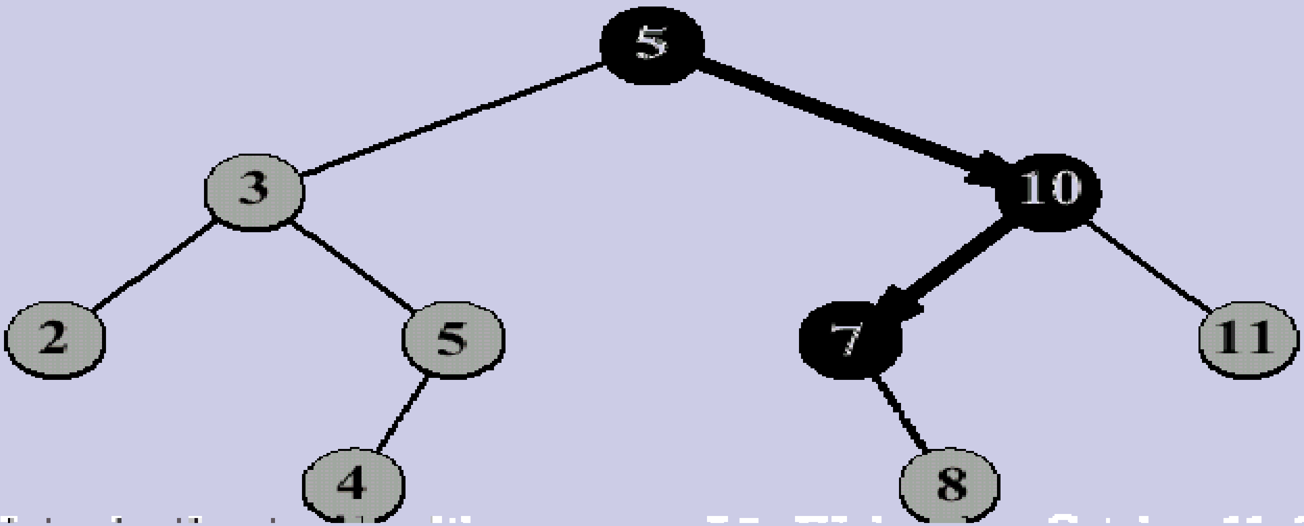
- For a tree of height *h*, the running time is *O*(*h*)

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 (as if searching for z),
  - 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 Example

- Insert 8



BST Insertion Pseudo Code

```
TreeInsert(T,z)
y ← NIL
x ← root[T]
while x ≠ NIL
  y ← x
  if key[z] < key[x]
    then x ← left[x]
    else x ← right[x]
p[z] ← y
if y = NIL
  then root[T] ← z
  else if key[z] < key[y]
    then left[y] ← z
    else right[y] ← z
```

BST Insertion: Worst Case

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