Design and Analysis of Algorithms

CSE 5311

Lecture 10 Binary Search Trees

Junzhou Huang, Ph.D.

Department of Computer Science and Engineering

Recall: Dynamic Sets

- data structures rather than straight algorithms
- In particular, structures for *dynamic sets*
  - Elements have a *key* and *satellite data*
  - Dynamic sets support *queries* such as:
    - ***Search(S, k), Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x)***
  - They may also support *modifying operations* like:
    - ***Insert(S, x), Delete(S, x)***

Motivation

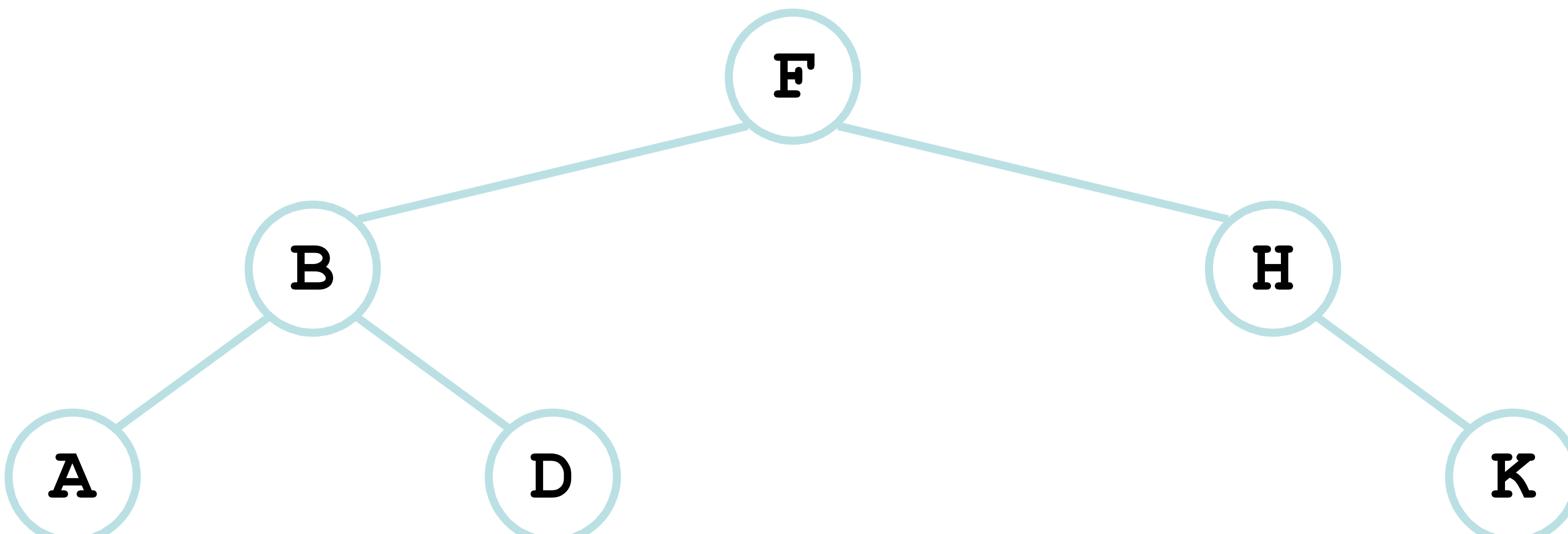
- Given a sequence of values:
  - How to get the max, min value efficiently?
  - How to find the location of a given value?
  - …
- Trivial solution
  - Linearly check elements one by one
- Searching Tree data structure supports better:
  - SEARCH, MINIMUM, MAXIMUM,
  - PREDECESSOR, SUCCESSOR,
  - INSERT, and DELETE operations of dynamic sets

Binary Search Trees

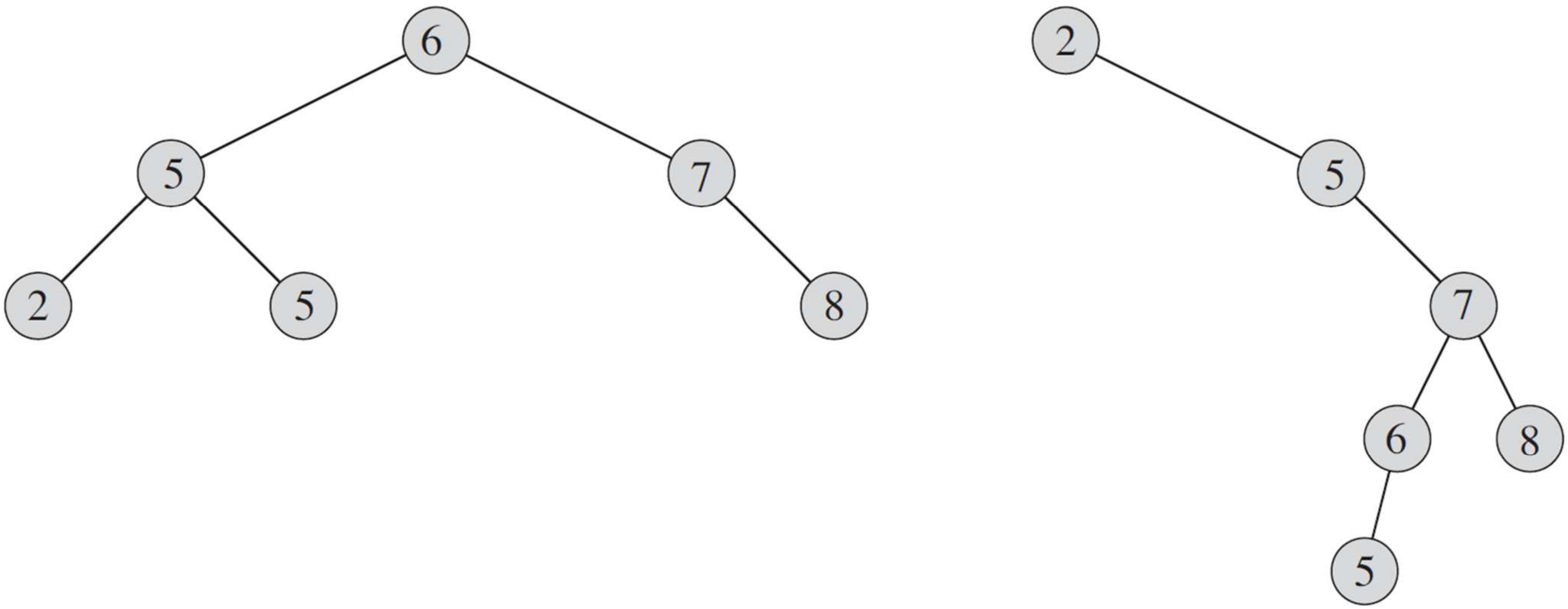
- ***Binary Search Trees*** **(BSTs) are an important data structure for dynamic sets**
  - Each node has at most two children
- Each node contains:
  - key and data
  - left: points to the left child
  - right: points to the right child
  - p(parent): point to parent
- Binary-search-tree property:
  - y is a node in the left subtree of x: $y.key \leq x.key$
  - y is a node in the right subtree of x: $y.key \geq x.key$
  - Height: ***h***

Binary Search Trees

- BST property:
  $key[leftSubtree(x)] \leq key[x] \leq key[rightSubtree(x)]$
- Example:



Examples



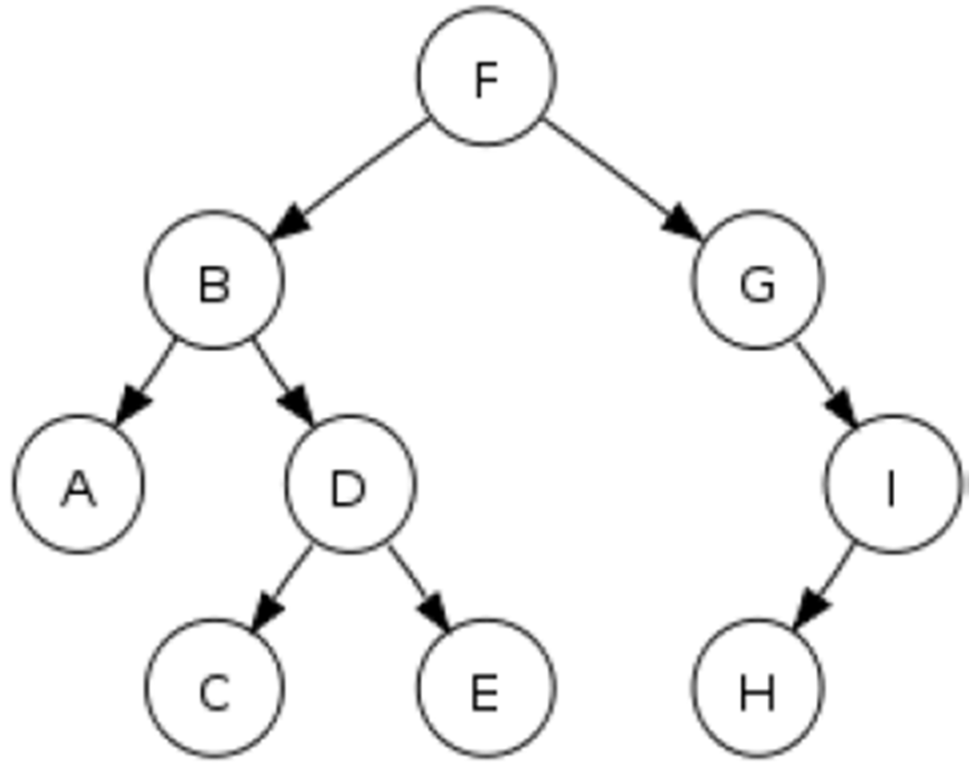
(a) (b)

Print out Keys

- Preorder tree walk
  - *P*rint key of node before printing keys in subtrees (node left right)
- Inorder tree walk
  - Print key of node after printing keys in its left subtree and before printing keys in its right subtree (left node right)
- Postorder tree walk
  - *P*rint key of node after printing keys in subtrees (left right node)

Example

- Preorder tree walk
  - F, B, A, D, C, E, G, I, H
- Inorder tree walk
  - A, B, C, D, E, F, G, H, I
  - Sorted (why?)
- Postorder tree walk
  - A, C, E, D, B, H, I, G, F



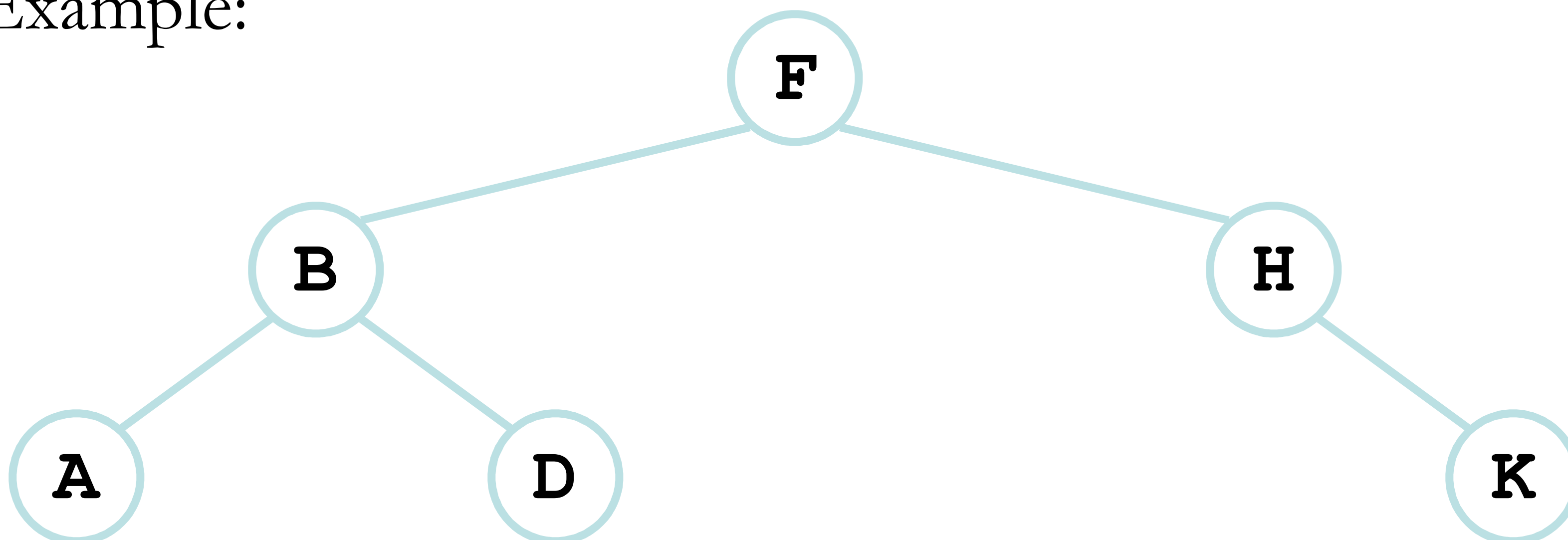
Inorder Tree Walk

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

- Inorder tree walk
  - *Visit and print each node once*
  - *Time:* $\Theta(n)$

Inorder Tree Walk

- Example:



- ***How long will a tree walk take?***
- ***Prove that inorder walk prints in monotonically increasing order***

Operations

- Querying operations
  - Search: get node of given key
  - Minimum: get node having minimum key
  - Maximum: get node having maximum key
  - Successor: get node right after current node
  - Predecessor: get node right before current node
- Updating operations
  - Insertion: insert a new node
  - Deletion: delete a node with given key

Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

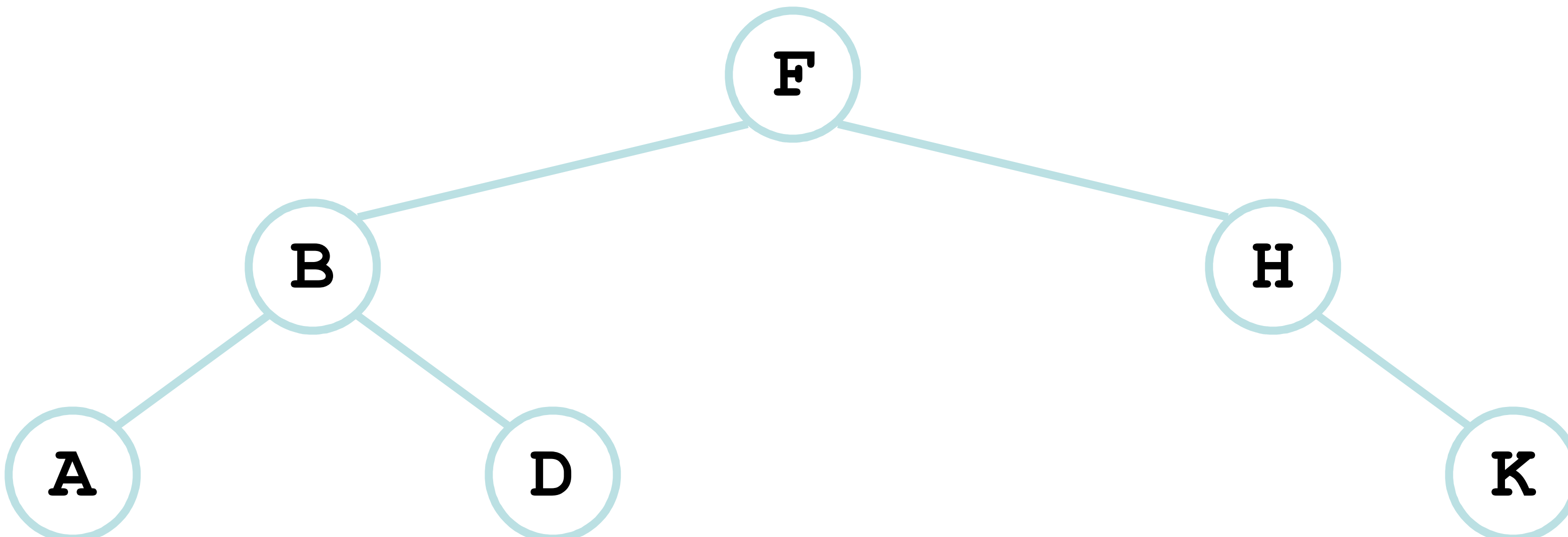
```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

Time = the length of path from root to found node

Time: O(h)

BST Search: Example

- Search for *D* and *C*:



Operations on BSTs: Search

- Here's another function that does the same:

```
TreeSearch(x, k)
  while (x != NULL  and  k != key[x])
    if (k < key[x])
      x = left[x];
    else
      x = right[x];
  return x;
```

- *Which of these two functions is more efficient?*

Operations: Minimum and Maximum

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

- Minimum: left most node
- Maximum: right most node
- Time: O(h)

Operations of BSTs: Insert

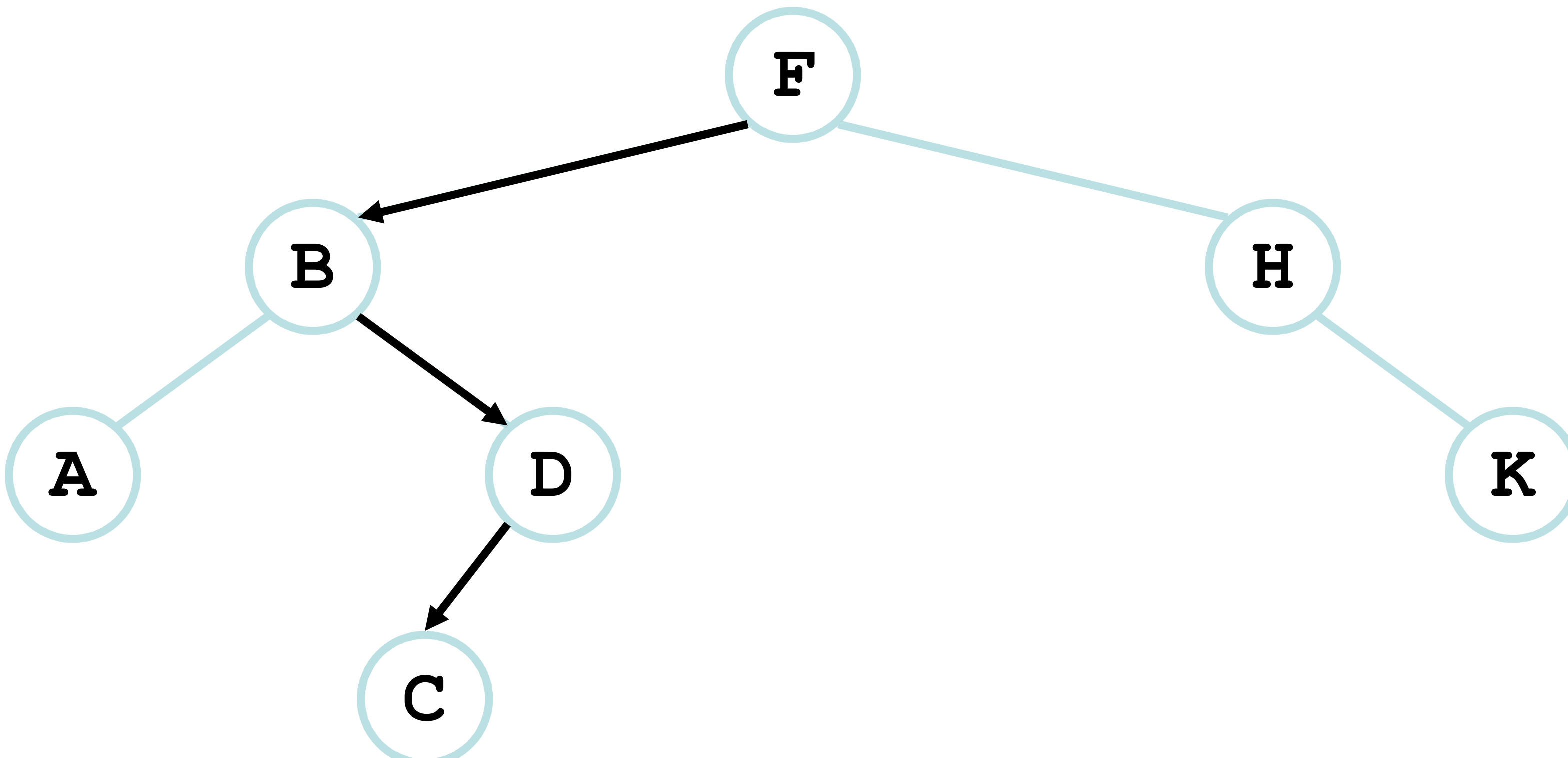
- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a "trailing pointer" to keep track of where you came from (like inserting into singly linked list)
  - Time: O(h)

Operations of BSTs: Insert

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

BST Insert: Example

- Example: Insert *C*



BST Search/Insert: Running Time

- *What is the running time of TreeSearch() or TreeInsert()?*
- A: O(h), where h = height of tree
- *What is the height of a binary search tree?*
- A: worst case: h = O(n) when tree is just a linear string of left or right children
  - We'll keep all analysis in terms of h for now
  - Later we'll see how to maintain h = O(lg n)

Sorting With Binary Search Trees

- Informal code for sorting array A of length *n*:

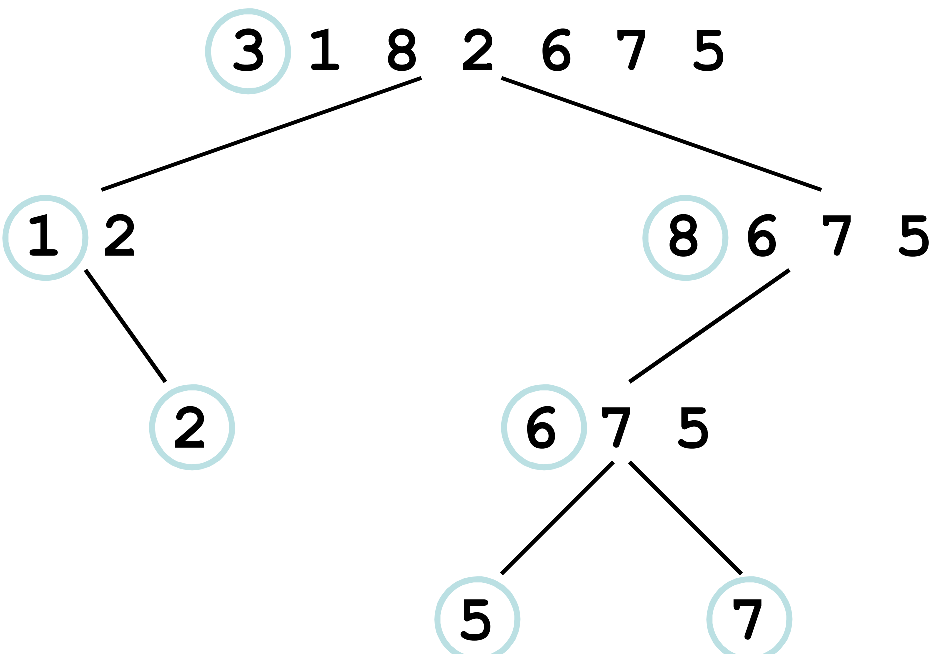
```
BSTSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

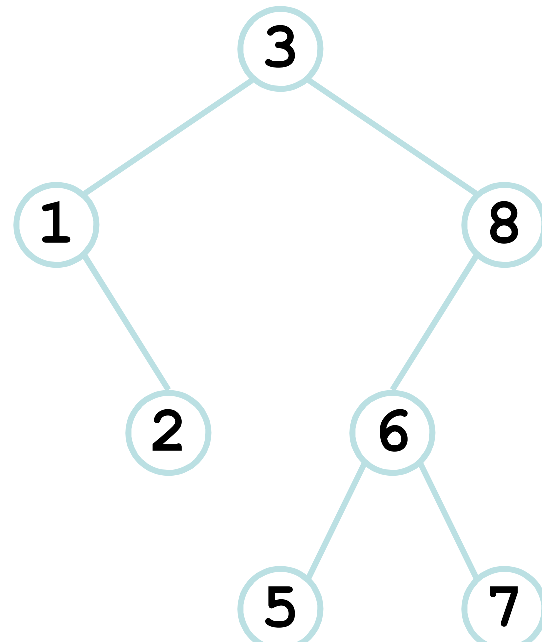
- *Argue that this is $\Omega(n \lg n)$*
- *What will be the running time in the*
  - *Worst case?*
  - *Average case? (hint: remind you of anything?)*

Sorting With BSTs

- **Average case analysis**
  - It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```





Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
  - Same comparisons as quicksort, different order!
    - Example: consider inserting 5

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- ***Which do you think is better, quicksort or BSTsort? Why?***

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTSort? Why?*
- A: quicksort
  - Better constants
  - Sorts in place
  - Doesn't need to build data structure

More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue
- *What operations must a priority queue have?*
  - Insert
  - Minimum
  - Extract-Min

BST Operations: Successor

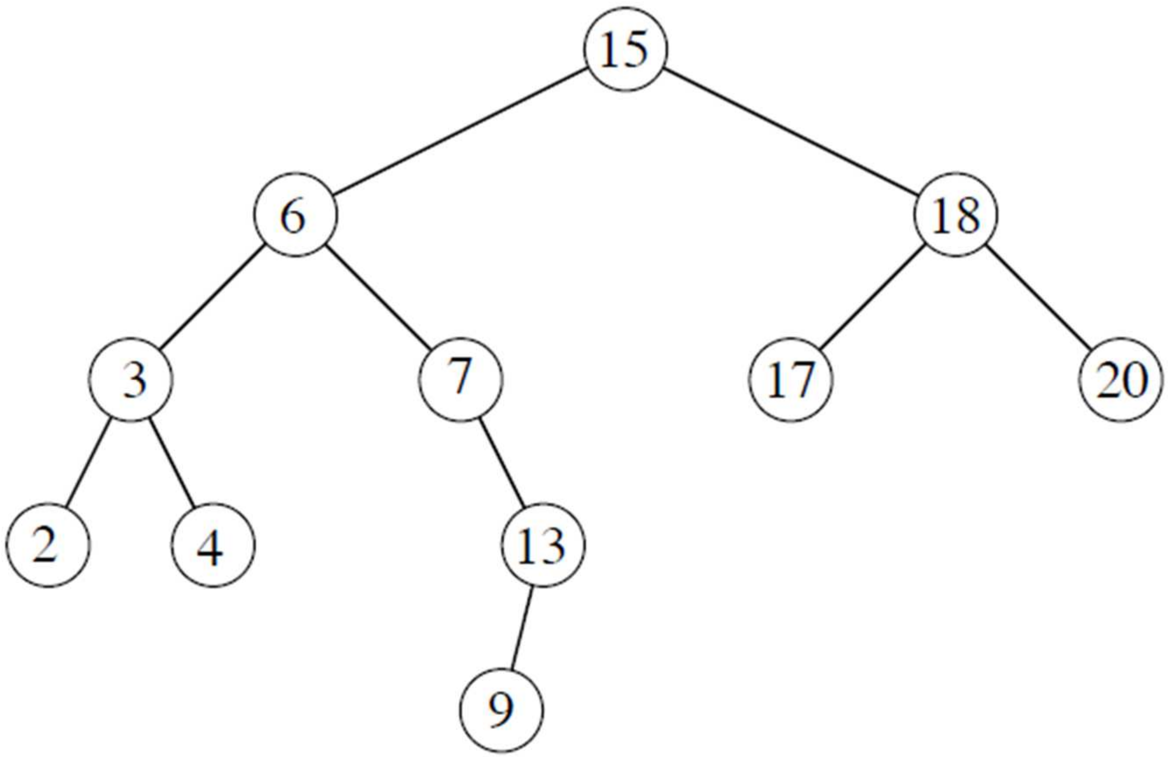
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

Successor in right subtree (lines 1–3)

Go up in left direction until turn right (lines 4–6)

- Time: O(h)

Example



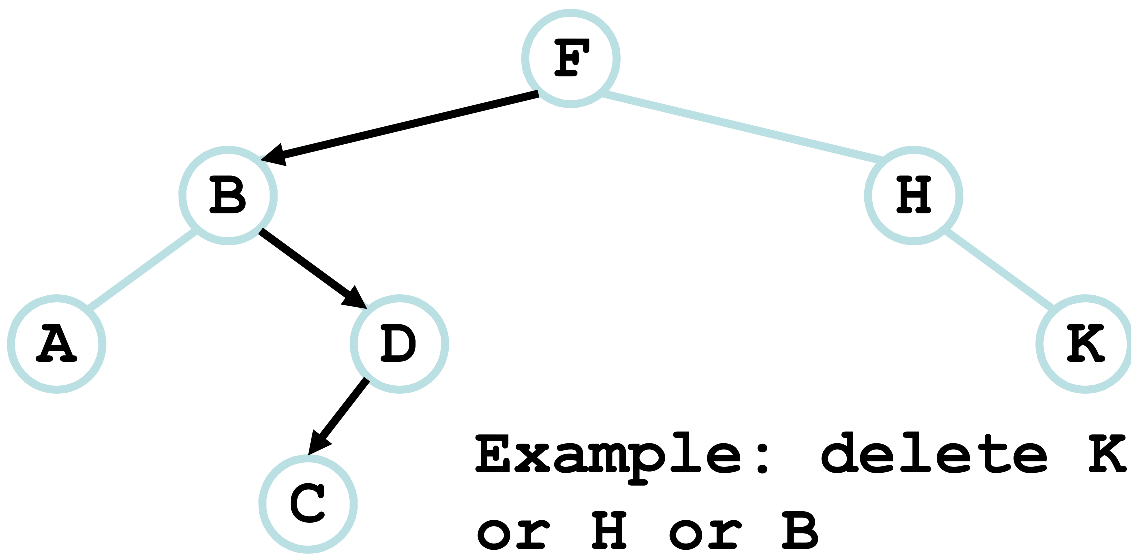
- Successor of 15 is 17
- Successor of 13 is 15

BST Operations: Successor

- **Two cases:**
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.
- **Predecessor: similar algorithm**

BST Operations: Delete

- Deletion is a bit tricky
  - Key point: choose a node in subtree rooted at x to replace the deleted node x
  - Node to replace x: predecessor or successor of x
- 3 cases:
  - x has no children:
    - ➢Remove x
  - x has one child:
    - ➢Splice out x
  - x has two children:
    - ➢Swap x with successor
    - ➢Perform case 1 or 2 to delete it



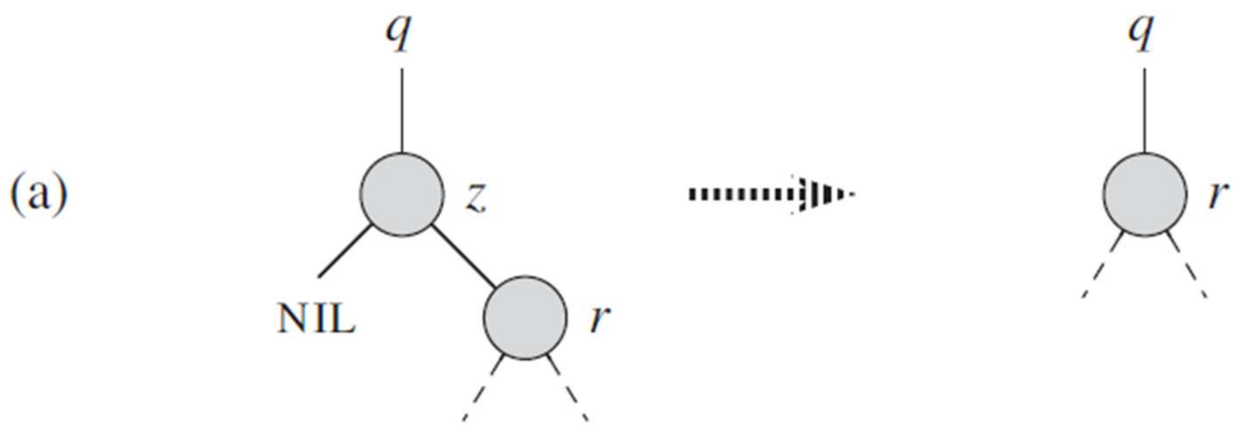
Example: delete K or H or B

BST Operations: Delete

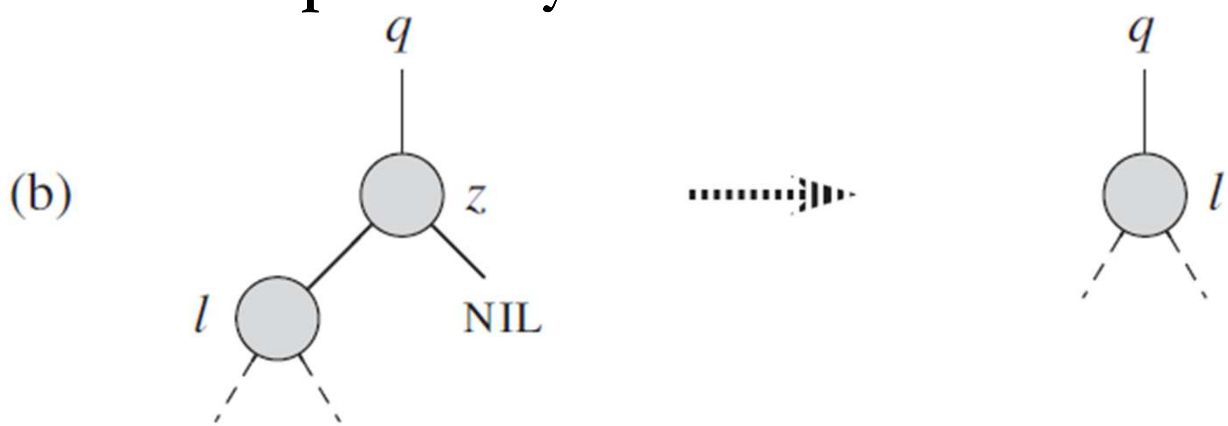
- *Why will case 2 always go to case 0 or case 1?*
- A: because when x has 2 children, its successor is the minimum in its right subtree
- *Could we swap x with predecessor instead of successor?*
- A: yes. *Would it be a good idea?*
- A: might be good to alternate

- Up next: guaranteeing a O(lg n) height tree

Has one child

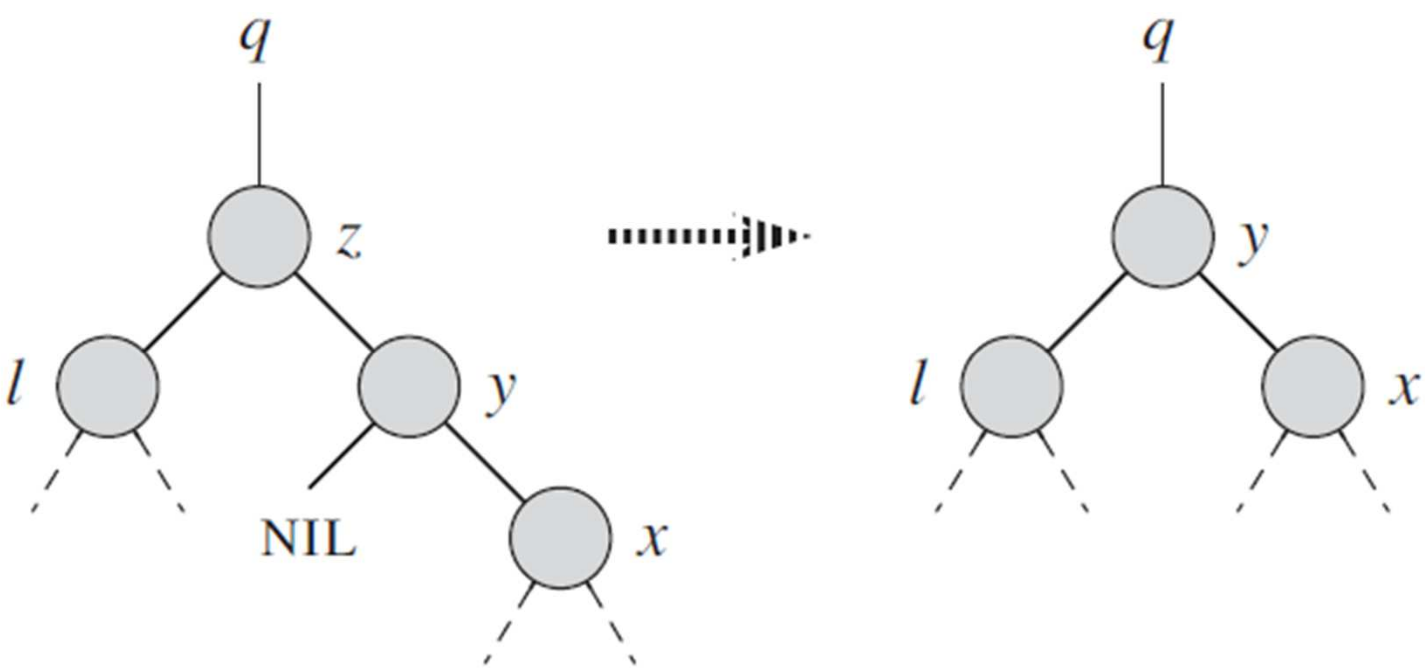


Replace *z* by its child



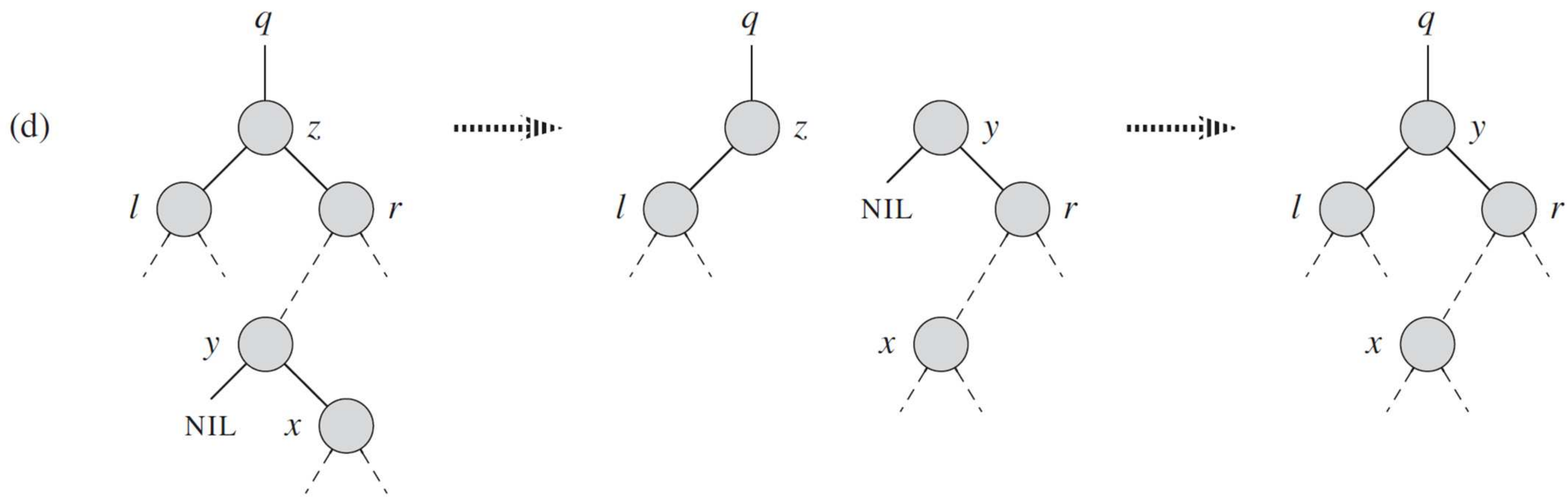
Right child has no left subtree

(c)



Replace *z* by its successor y

Right child has left subtree



1. Find successor y of z
2. Replace y by its child
3. Replace z by y

Replace a mode by its Child

- Replace the subtree rooted at node u with the subtree rooted at node v
- Running time: $O(1)$

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

Deletion Algorithm

- Main running time: find z's successor
- Time: $O(h)$

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Summary

- Binary search tree stores data hierarchically
- Support SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE operations
- Running time of all operation is $O(h)$
- Question: What is the lower bound of h? How to achieve it?