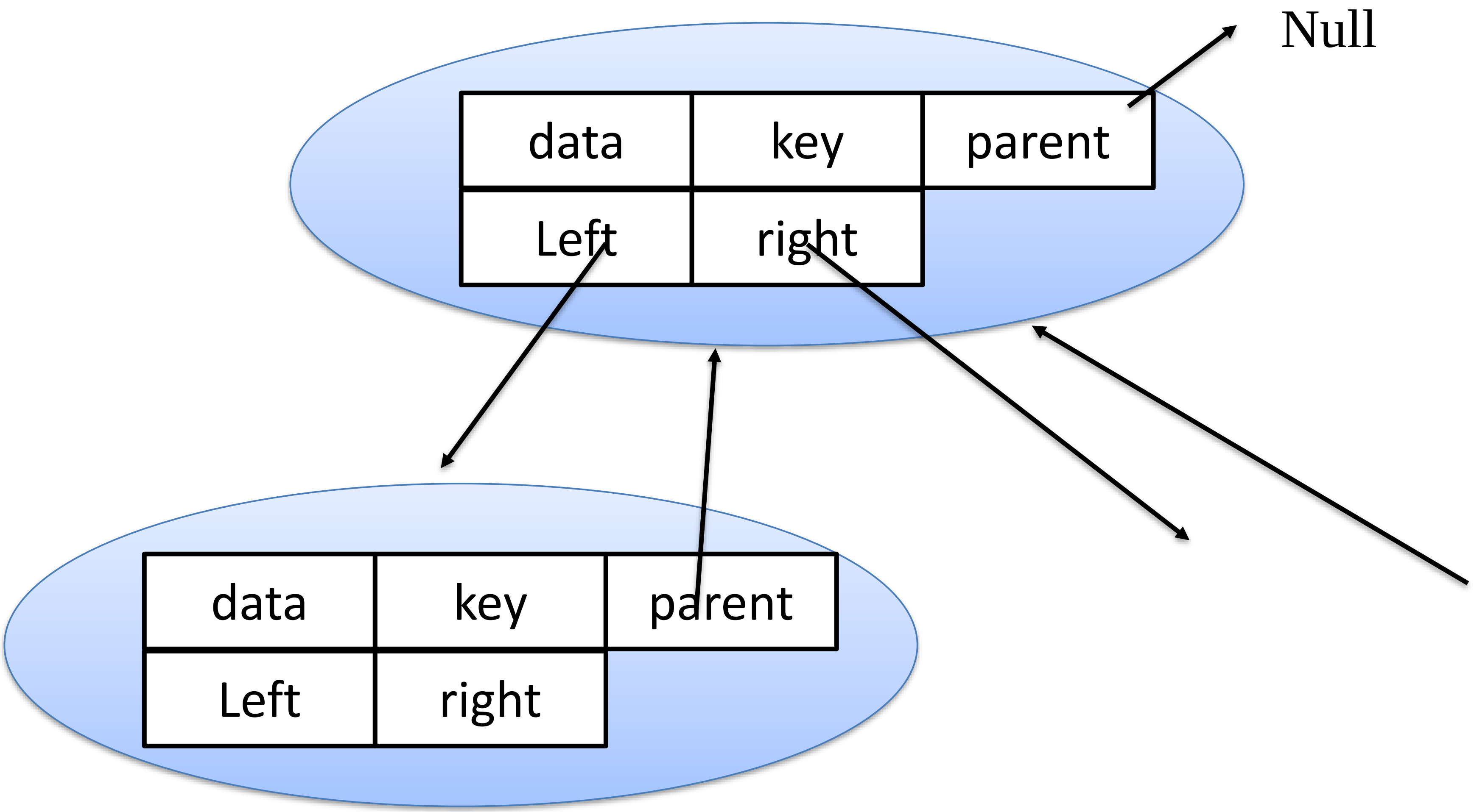
COT 6405 Introduction to Theory of Algorithms

Topic 13. Binary Search Tree

Binary Search Trees

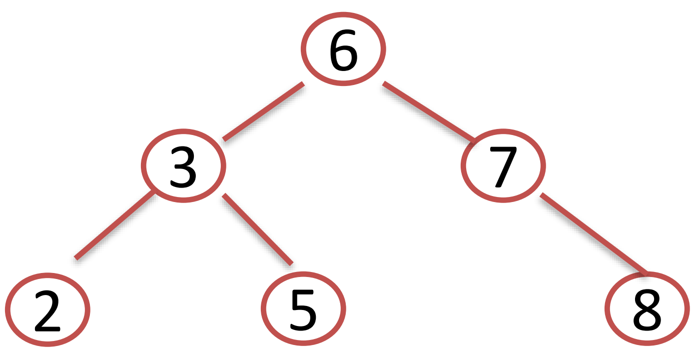
- Binary Search Trees (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, nodes have:
  - **key**: an identifying field inducing a total ordering
  - **left**: pointer to a left child (may be NULL)
  - **right**: pointer to a right child (may be NULL)
  - **p**: pointer to a parent node (NULL for root)

Node implementation

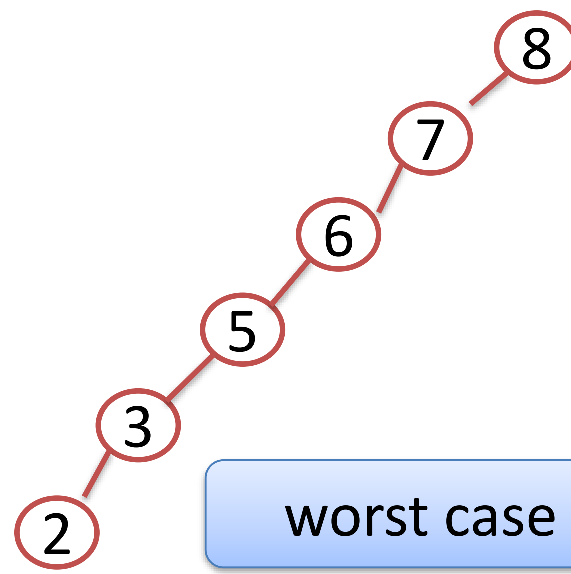


Binary Search Trees

- BST property: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key < x.key. If y is a node in the right subtree of x, then y.key > x.key. Different BSTs can constructed to represent the same set of data



Average case O(lgn)



worst case O(n)

Walk on BST

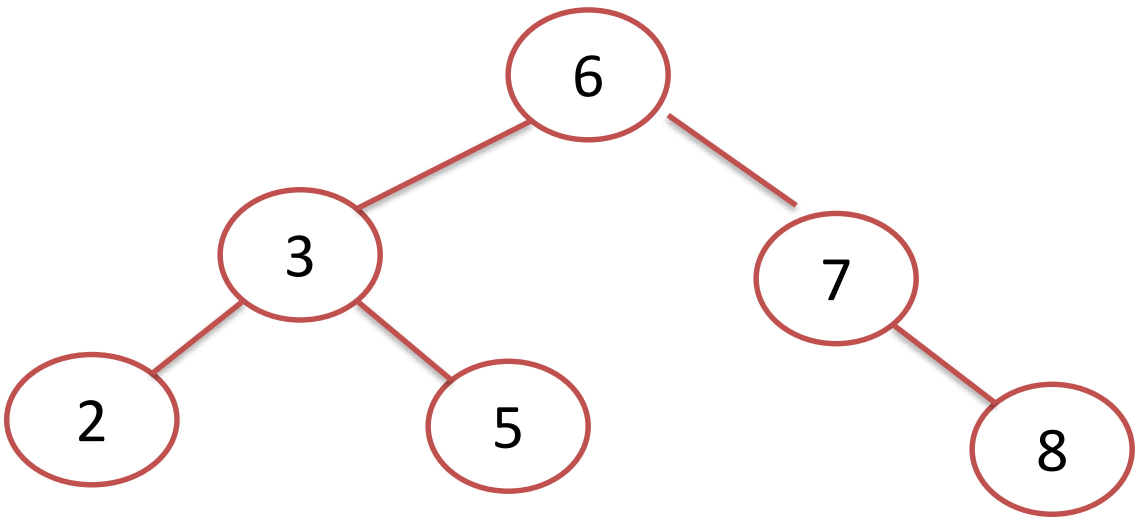
- A: prints elements in sorted (increasing) order

```
InOrderTreeWalk(x)
        InOrderTreeWalk(x.left);
        print(x);
        InOrderTreeWalk(x.right);
```

- This is called an inorder tree walk
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

Example

- What is the result for in-order walk, pre-order walk, and post-order walk?



In order: 2 3 5 6 7 8
Pre order: 6 3 2 5 7 8
Post order: 2 5 3 8 7 6

Analyze a tree walk in recursion

- Theorem: If *x* is the root of an *n*-node tree, then the call INORDER-TREE-WALK(*x*) takes $\Theta(n)$ time.
- Proof: suppose left subtree of x has k nodes and right subtree has n - k - 1 nodes. The running time T(n) is $T(n) = T(k) + T(n - k - 1) + d$, where d reflects the time to execute INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

Proof (Cont'd)

- We use the substitution method to show that T(n) = O(n). Assume T(k) $\leq ck$
- T(n) = T(k) + T(n - k - 1) + d

  $\leq ck + c(n - k - 1)$ +d

  = cn – c +d

  $\leq cn$ if c $\geq$ d

Use the same method, we can prove that T(n) = Ω (n). Thus, T(n) = $\Theta(n)$

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 = x.key)
          return x;
     if (k < x.key)
          return TreeSearch(x.left, k);
     else
          return TreeSearch(x.right, k);
```

Operations on BSTs: Search

- Here's another function that does the same

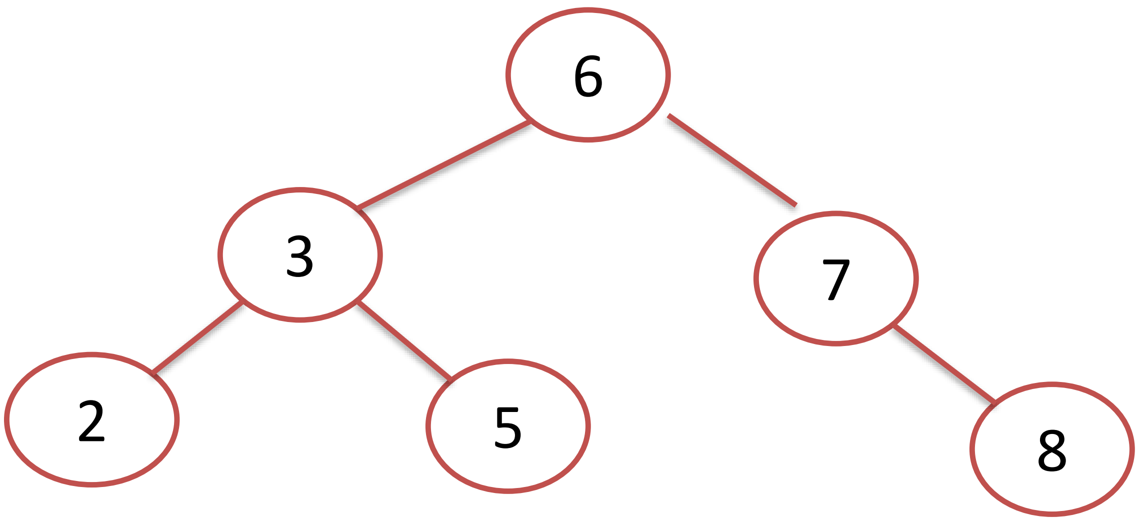
Iterative-Tree-Search **(x, k)**

```
while (x != NULL and k != x.key)
    if (k < x.key)
        x = x.left;
    else
        x = x.right;
return x;
```

- Which of these two functions is more efficient?

Example

- Search for 5 and 8



BST Operations: Minimum

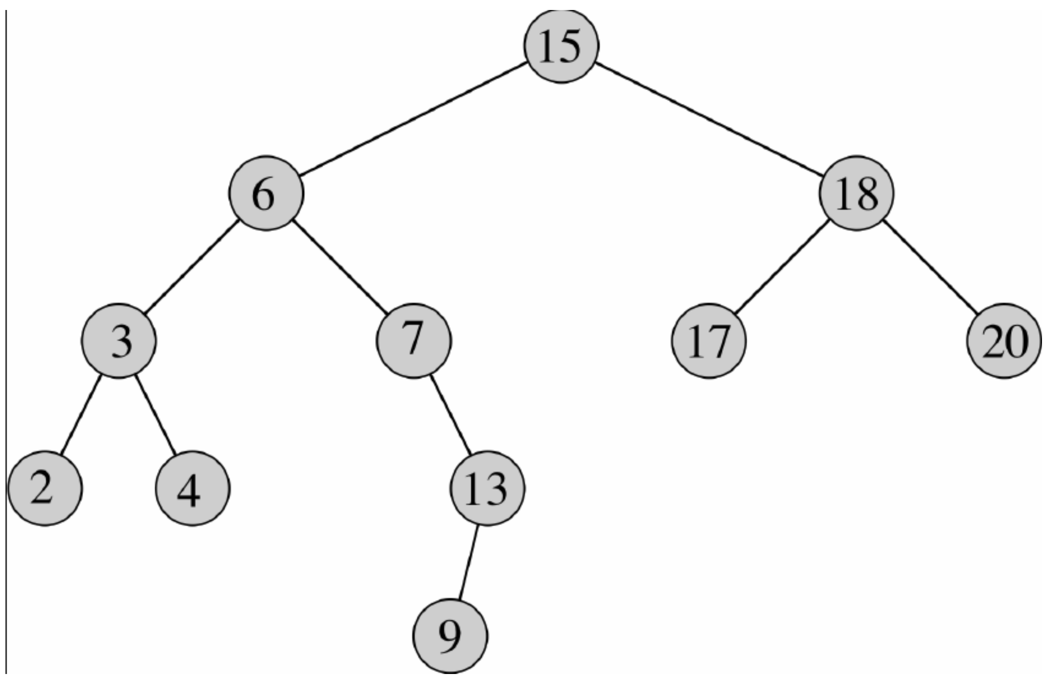
- How can we implement a Minimum() query?

```
TREE_MINIMUM(x)
        while x.lef <> NIL
              x = x.left
         Return x
```

- What is the running time?
- Minimum→ Find the leftmost node in tree
- Maximum → find the rightmost node in the tree

BST Operations: Successor

- Successor of x: the smallest key greater than *key*[*x*].
- What is the successor of node 3? Node 15? Node 13?
- What are the general rules for finding the successor of node x? (hint: two cases)

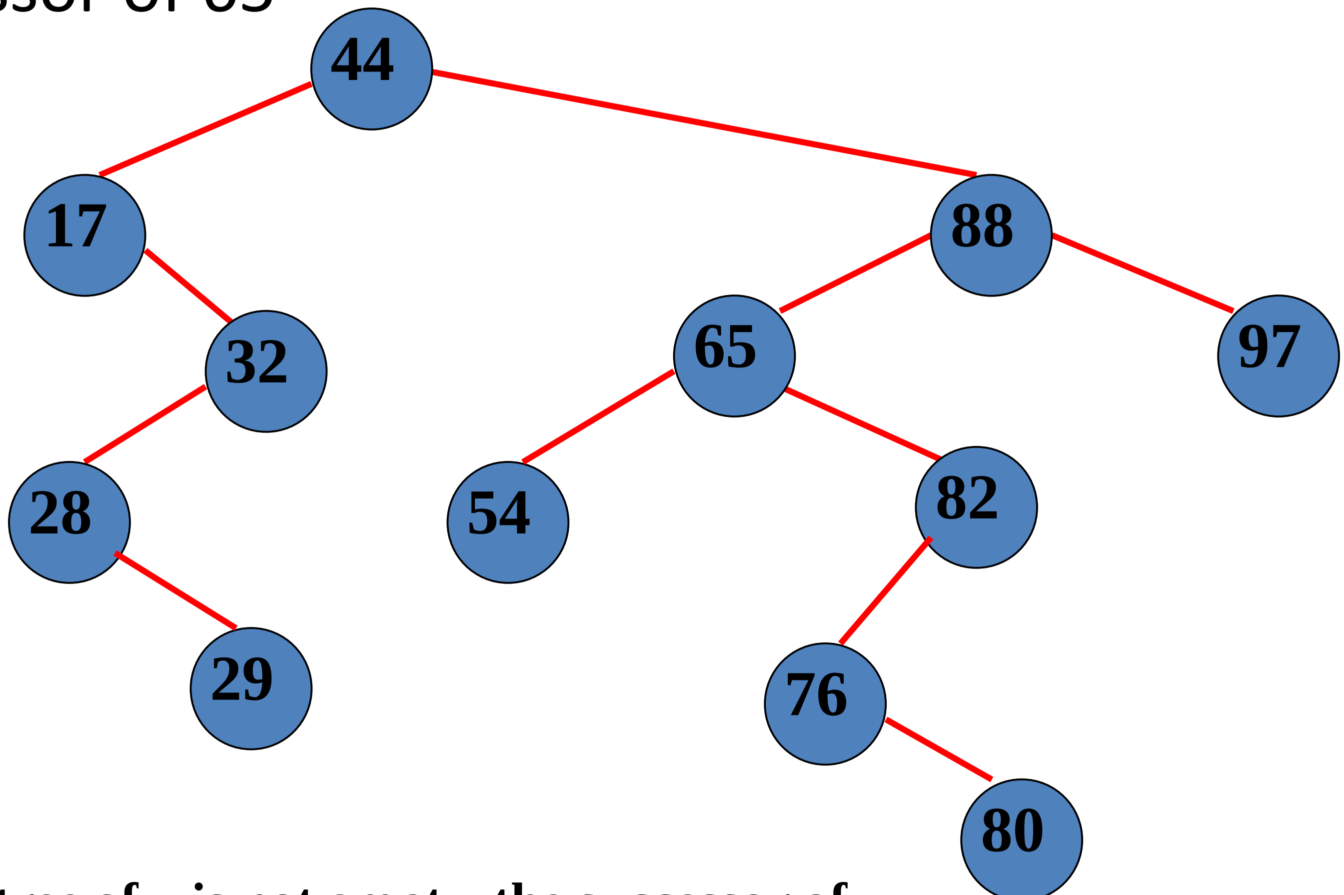


BST Operations: Successor

- Two cases:
  - x has a right subtree: its successor is minimum node in right subtree
  - x has no right subtree: x must be on the left sub tree of the successor such that x <= successor. So the successor is the first ancestor of x whose left child is an ancestor of x (or x)
    - Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.

BST: Find Successor of a Node

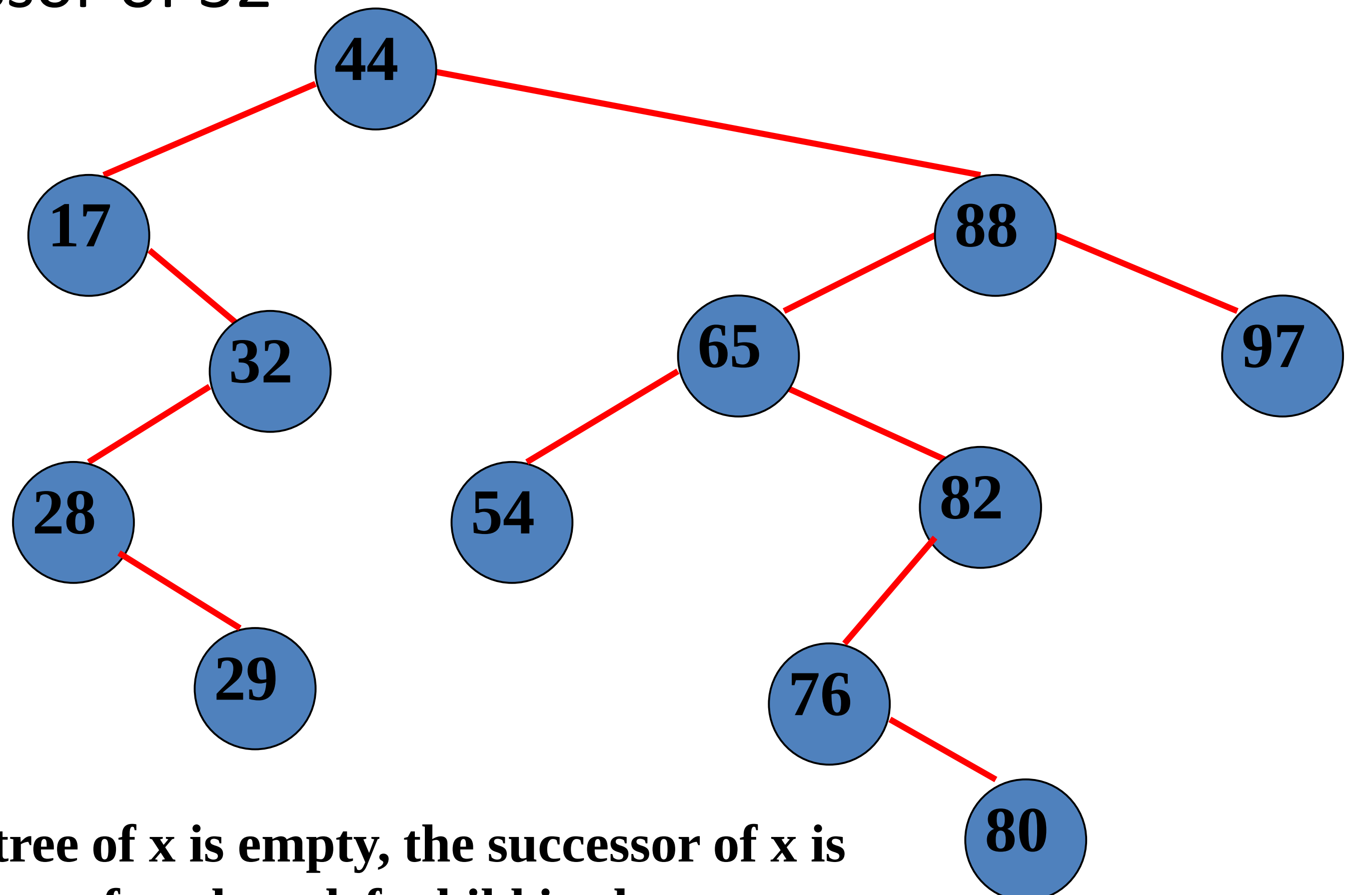
Find Successor of 65



If the right sub-tree of x is not empty, the successor of x is the leftmost node in its right sub-tree.

BST: Find Successor of a Node

Find Successor of 32



If the right sub-tree of x is empty, the successor of x is the lowest ancestor of x whose left child is also an ancestor of x

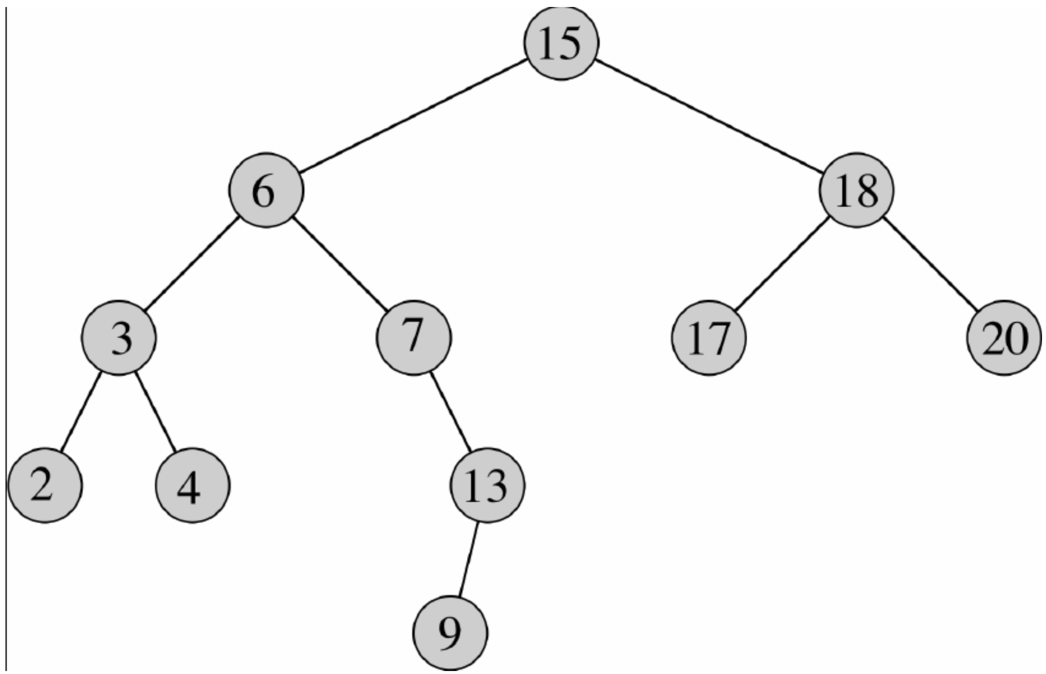
Find Successor Algorithm for BST with pointers to parents

// Returns node in BST that is the successor
// of node x, or NIL if no successor

```
Tree-Successor( x )
        if  x.right ≠ NIL
                then  return Tree-Minimum( x.right )
        y = x.p
        while (y ≠ NIL and x == y.right)  //to left up
                x = y
                y = y.p  // move up one node
        return y
```

BST Operations: predecessor

- Predecessor of x: the greatest key smaller than *key*[*x*].
- What is the Predecessor of node 6? Node 7?



BST Operations: predecessor

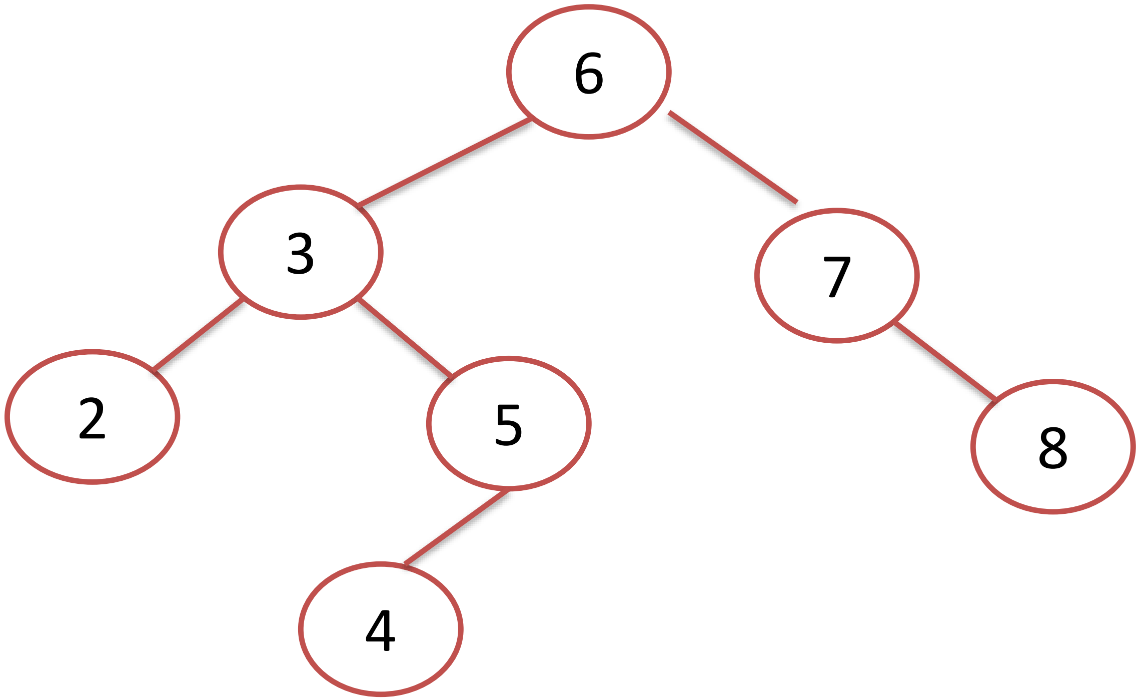
- Two cases:
  - x has a left subtree: its predecessor is maximum node in left subtree
  - x has no left subtree: x must be on the right sub tree of the predecessor such that x >= predecessor. So the predecessor is the first ancestor of x whose right child is an ancestor of x (or x)

Operations of BSTs: Insert

- Adds an element x to the tree
  - → the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Use a “trailing pointer” to keep track of where you came from
    - like inserting into singly linked list

BST Example: Insert C

- Example: Insert 4



Iterative Insertion Algorithm for BST with pointers to parents

```
Tree-Insert ( T, z )              // Inserts node z into BST T
        y = NIL
        x = root[ T ]
        while  x ≠ NIL
                y = x
                if z.key < x.key
                        then x = x.left
                else x = x.right
        z.p = y
        if y == NIL          // If tree T was empty
                then T.root = z          // New node is root
        else if z.key < y.key
                then y.left = z
        else y.right = z
```

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

Sorting With Binary Search Trees

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

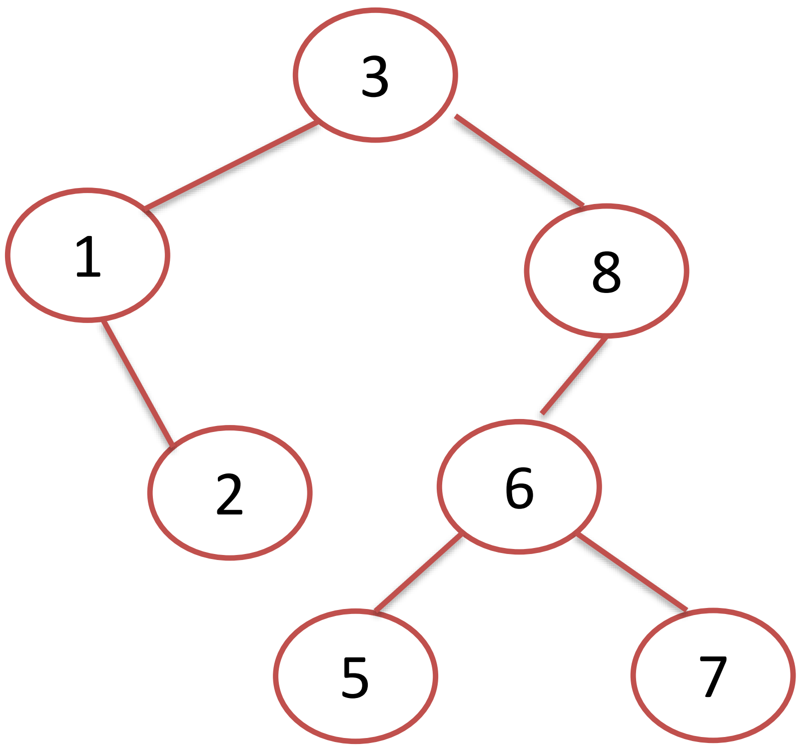
```
BSTSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

- What will be the running time in the
  - Worst case?
  - Average case?

BSTsort example

- Example:

3	1	8	2	6	5	7



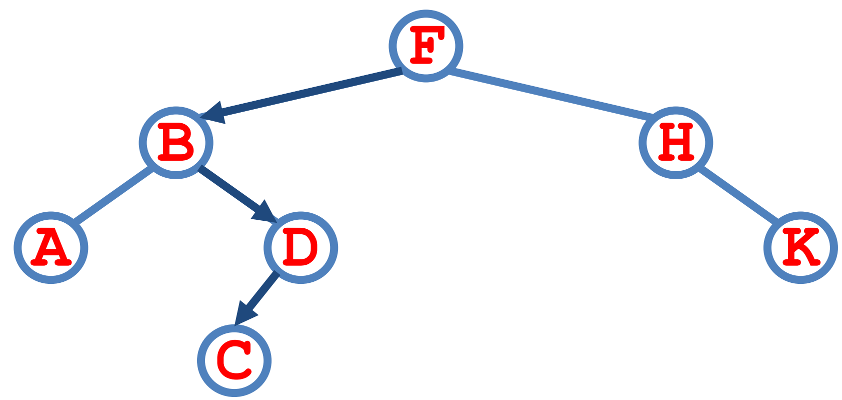
It's similar to quicksort with Pivot = A[0]!

Sorting with BSTs

- Which do you think is better, quicksort or BSTsort? Why?
- Answer: quicksort
  - Sorts in place (why BSTsort is not in place?)
  - Doesn't need to build data structure

BST Operations: Delete

- Several cases:
  - x has no children:
    - Remove x
    - Set parent's link NULL
  - x has one child:
    - Replace x with its child
    - Set the child's link NULL
  - x has two children:
    - replace x with its successor
    - Perform case 0 or 1 to delete the successor



Example: delete K or H or B

BST Operations: Delete

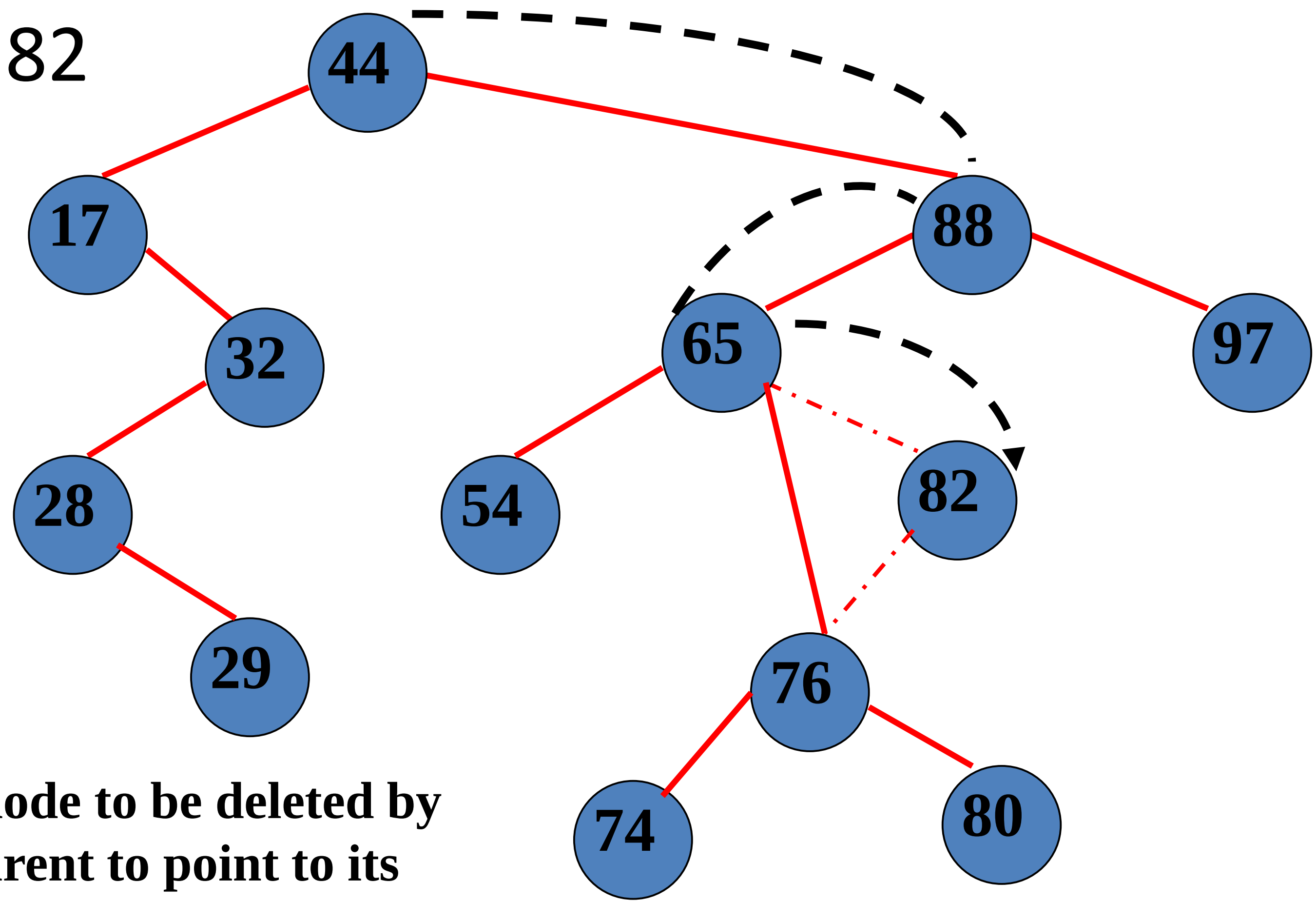
- Why will case 2 always go to case 0 or case 1?
- Answer: because when x has 2 children, its successor is the minimum on its right subtree
  - The successor is the leftmost node on the right subtree
  - The successor either has no children or has the right child node only
  - Why can't the successor have two children nodes or the left child node?
    - Because the successor should be the smallest

Case 2: BST property

- How to prove that replacing x with its successor still maintains the BST property?
  - Nodes on the left subtree are smaller than the successor
  - Nodes on the right subtree are greater than the successor
  - If the successor is the left child of the parent, it is smaller than the parent.
  - Otherwise, it is larger than the parent

Deletion of Node with Zero or One Child

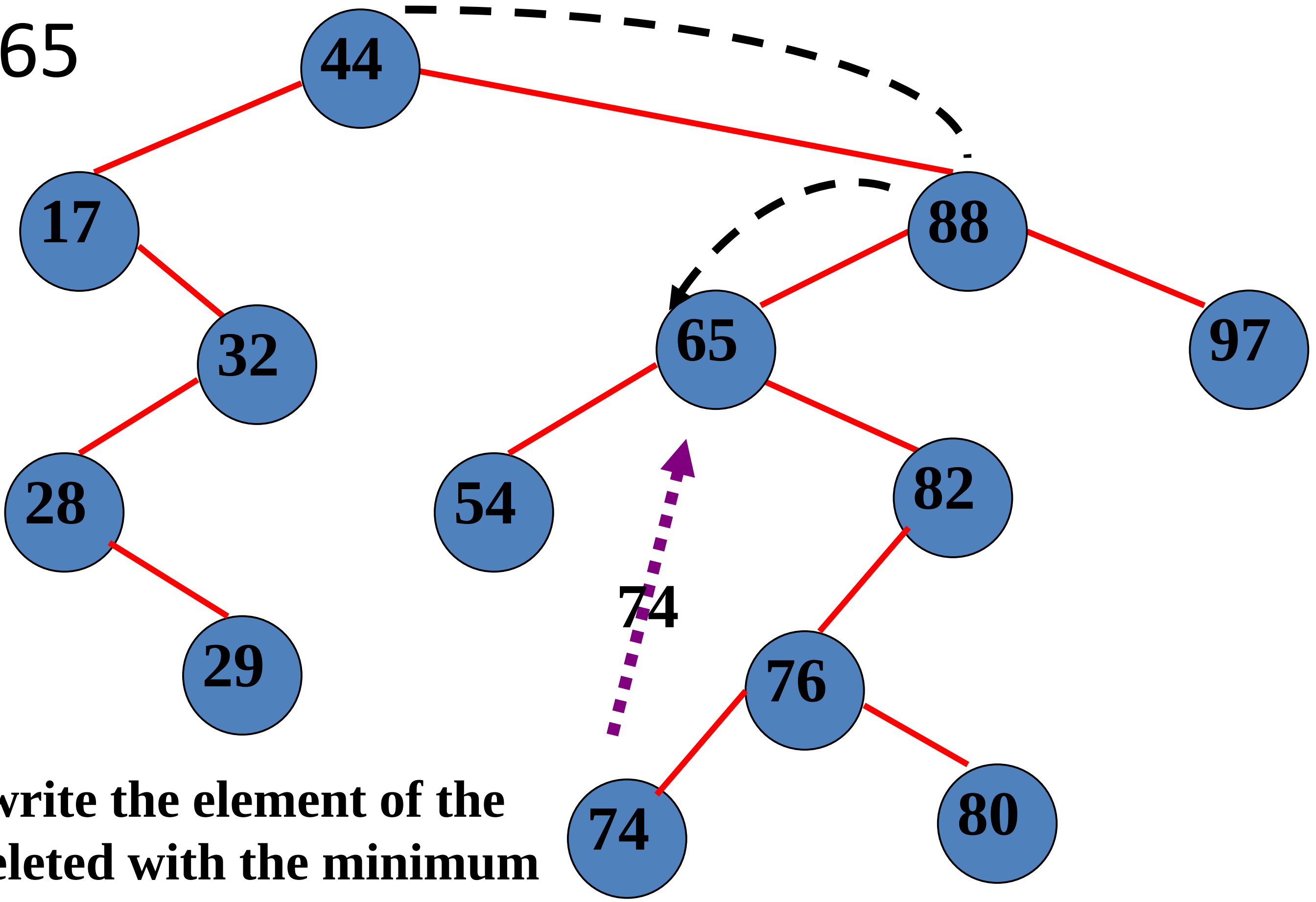
Delete 82



Bypass the node to be deleted by setting its parent to point to its child

Deletion of Node with Two Children

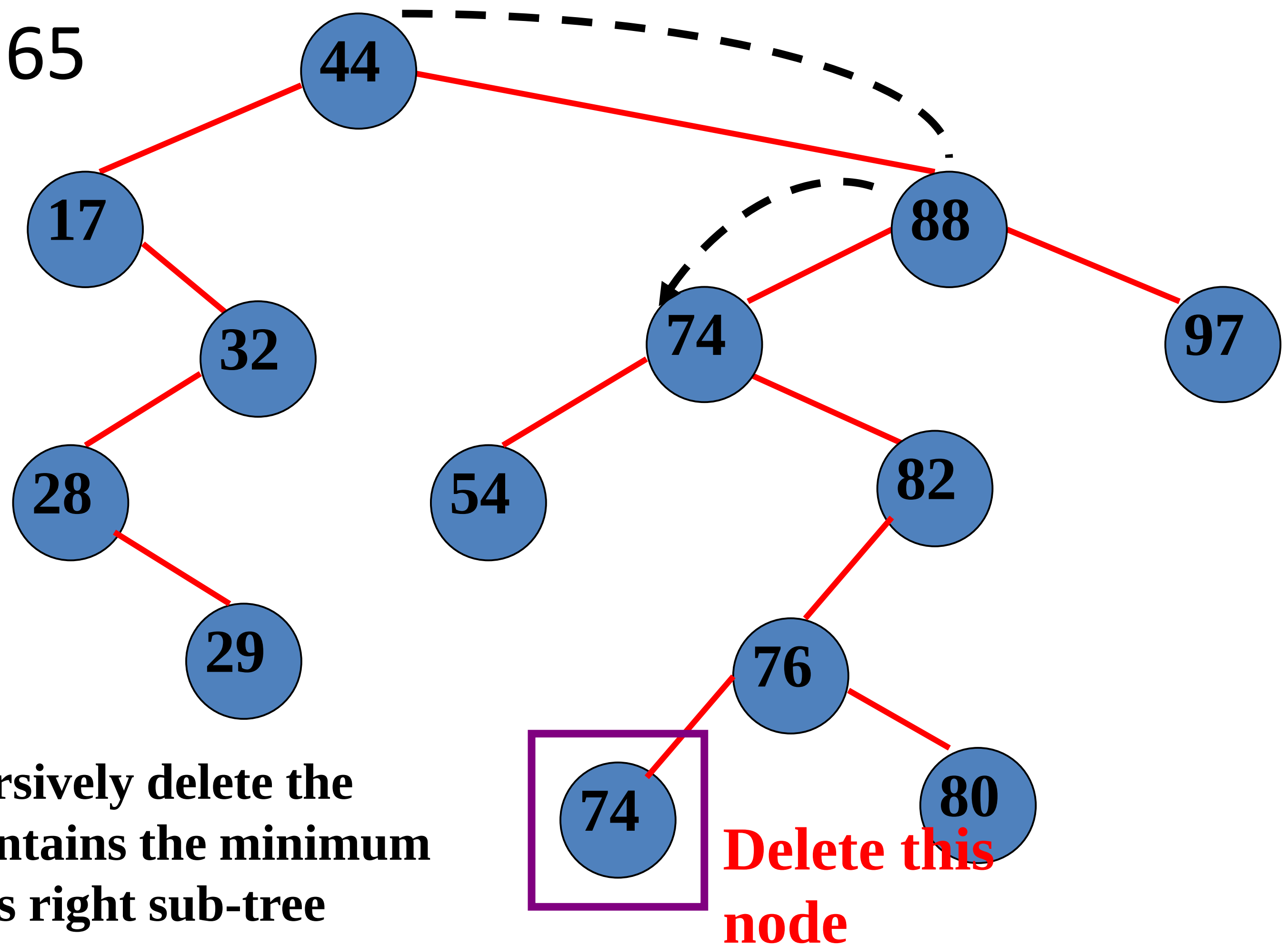
Delete 65



Step1: **Overwrite the element of the node to be deleted with the minimum element of its right sub-tree**

Deletion of Node with Two Children

Delete 65



Step2: **Recursively delete the node that contains the minimum element of its right sub-tree**

```
Tree-Delete (T, z ) // Deletes node z from BST T
        x = NIL
        if z.left == NIL or z.right == NIL
                then y = z
                else  y = Tree-Successor( z )
        if y.left ≠ NIL
                then x = y.left
                else  x = y.right
        if x ≠ NIL
                then x.p =y.p
        if y.p == NIL
                then root[ T ] = x
        else if y == p[ y ] .left
                then p[ y ] .left = x
                else  p[ y ] .right = x
        if y ≠ z
                then z.key = y.key
        return y
```

BST Summary

- BST is one of the most useful tools for maintaining dynamic sets
- Performance bound → O(h), tree height h
- Difference between BST and Min/Max heap