CSC263H
Data Structures and Analysis

Prof. Bahar Aameri & Prof. Marsha Chechik

Winter 2024 – Week 3

Dictionary ADT:

- **Objects**: A collection of **key-value pairs** (keys are *unique*).
- **Operations**:
  - *Search*(D,k): return x in D s.t. $x.key = k$, or NIL if no such x is in D.
  - *Insert*(D,x): insert x in D; if some y in D has $y.key$ equal to $x.key$, replace y by x.
  - *Delete*(D,x): remove x from D.

Note: k is a key, x is a node.

Unsorted List:

- *Search*(D, k): $\Theta(n)$
- *Insert*(D, x): $\Theta(1)$
- *Delete*(D, x): $\Theta(n)$

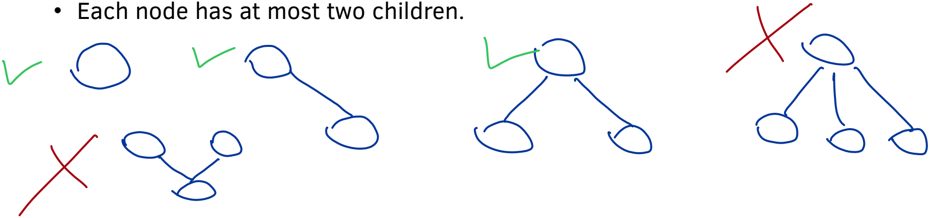
Sorted List (by keys):

- *Search*(D,k): $\Theta(\log n)$
- *Insert*(D, x): $\Theta(n)$
- *Delete*(D, x): $\Theta(n)$

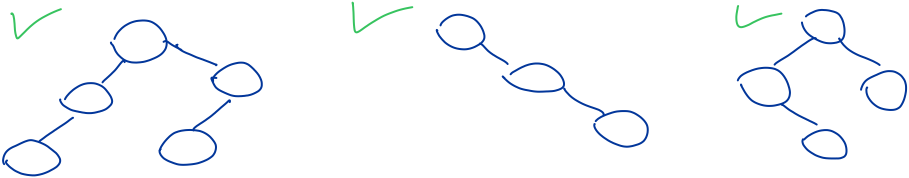
- Searching the list randomly is slow and sub-optimal.
- Using binary search on a sorted list, we can reduce the run-time of the search to $\mathcal{O}(\log n)$.
- Is there a data structure that stores and organizes keys in a dictionary in such a way?

Binary Search Tree (BST): A binary tree that satisfies the binary search tree property: for every node x, $x.key$ is greater than every key in left *sub-tree* of x, and $x.key$ is less than every key in right *sub-tree* of x.

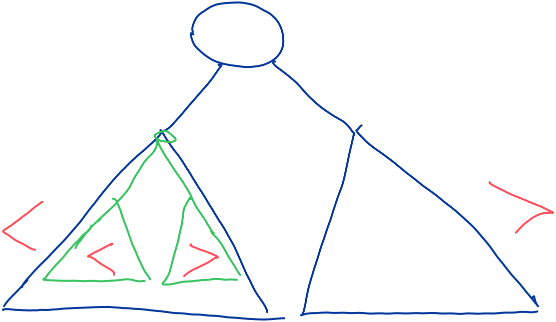
- Each node has at most two children.



- Nodes don't have to be full, unlike binary heaps.

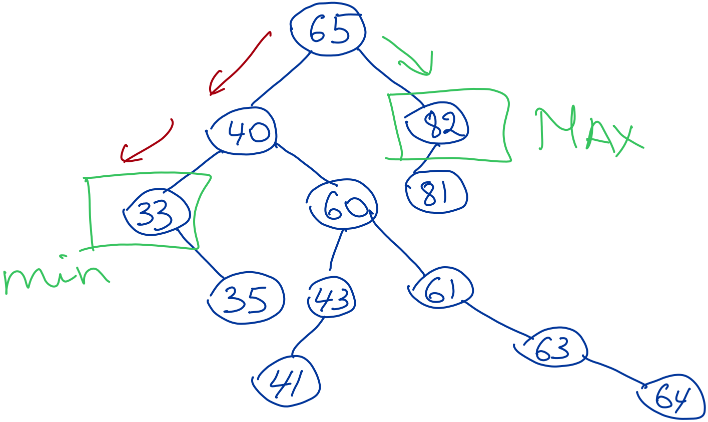


BST property is recursive!



Properties of Binary Search Trees (BSTs)

- **Minimum** value of a BST: left-most node with *no left child*
- **Maximum** value of a BST: right-most node with *no right child*



Properties of Binary Search Trees (BSTs)

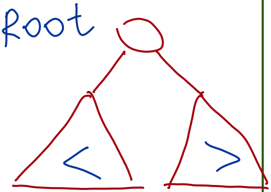
Information at each node x:

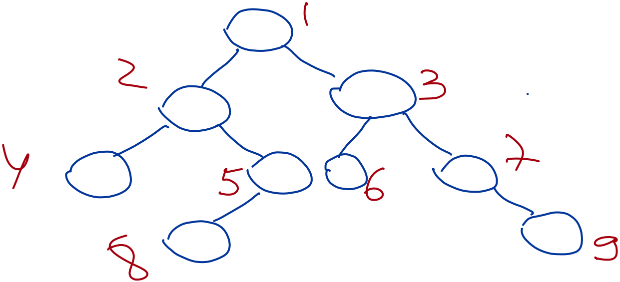
- $x.key$: the key
- $x.left$: the left child (node)
- $x.right$: the right child (node)
- $x.p$: the parent (node)

Because of BST property, we can say that the keys in a BST are **sorted**.

How to obtain a sorted list from a BST?

- Inorder traversal (Depth First: Left, Root, Right)
- Preorder traversal (Depth First: Root, Left, Right)
- Postorder traversal (Depth First: Left, Right, Root)
- Level-by-level traversal (Breadth-First)





InorderTraversal(x): Passing a tree by passing its root node

print all keys in BST rooted at x in ascending order

if $x \neq$ NIL
InorderTraversal (x. left)
Print (x. key)
InorderTraversal (x. right)

Worst-case Running Time: $\Theta(n)$

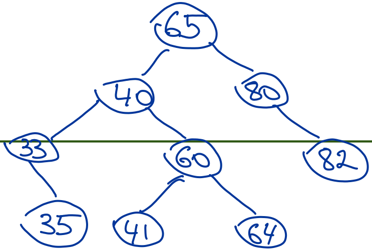
BSTs: Finding Minimum

BSTMin(x): Return the node with the minimum key in the tree rooted at x.

BSTMin(x):

1. Start from the $\boldsymbol{x}$.
2. keep going left, until the left child is **NIL**.
3. **return** the final node.

Worst-case Running Time: $\Theta(n)$




```
BSTMin(x) :
    while x.left ≠ NIL :
        x = x.left,
    return x
```

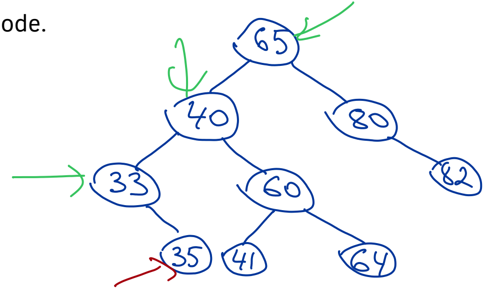
Search(D,k): return x in D s.t. $x.key = k$, or NIL if no such x is in D.

BSTSearch(root, k):

1. Start from the **root**.
2. If k is smaller than the key of the current node, go left.
3. If k is larger than the key of the current node, go right.
4. If equal, **return** the current node.
5. If going to **NIL**, not found.

33

34



Worst-case Running Time:

$\Theta(h)$

h can be as big as n

BSTs: Search

```
BSTSearch(root, k):
    if root == NIL:
        return NIL
    else if root.key == k:
        return root
    else if root.key > k:
        return BST Search(root.left, k)
    else :
        return BST Search(root.right, k)
```

Insert(D,x): insert x in D; if some y in D has $y.key$ equal to $x.key$, replace y by x.

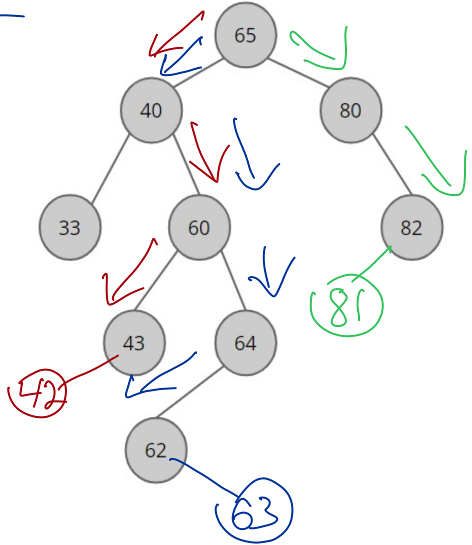
BSTInsert(root, x):

1. Start from the **root**.
2. Go down, left and right like what we do in *BSTSearch*
3. When next position is NIL, insert there.
4. If find equal key, replace the node.

Worst-case Running Time: $\Theta(h) = \Theta(n)$

h can be as big as n

Insert 42, 81, 63



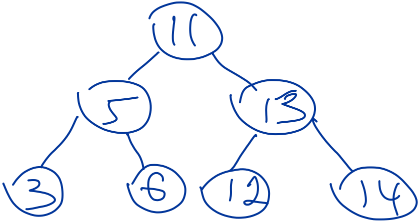
BSTs: Insert

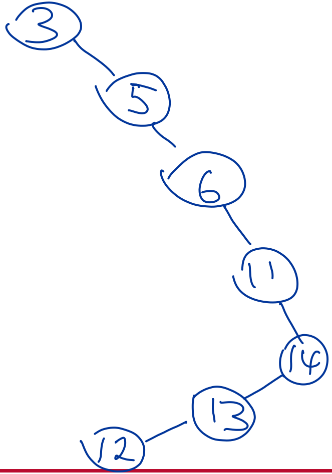
```
BSTInsert(root, x):
    if root == NIL:
        root = x
    else if root.key > x.key:
        root.left = BSTInsert(root.left, x)
    else if root.key < x.key:
        root.right = BSTInsert(root.right, x)
    else: # x.key == root.key
        replace root with x        # update x.left, x.right
    return root
```

If inserting the same set of data, but in a different sequence, will the shape of the BST be the same?

Insert sequence 1: 11, 5, 13, 12, 6, 3, 14

Insert sequence 2: 3, 5, 6, 11, 14, 13, 12





Successor(x): Find the node which is the successor of x in the <u>*sorted list*</u> obtained by <u>*inorder traversal*</u> (i.e., node with the smallest key larger than x).

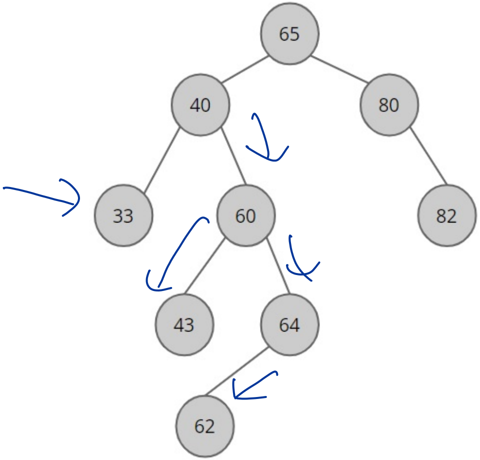
Successor of 33 is 40

Successor of 40 is 43

Successor of 60 is 62

Successor of 64 is 65

Successor of 65 is 80



BSTs: Successor

Case 1: x has a *right* child.

- *Successor*(x) must be the ***minimum*** in the ***right subtree*** of x.

Case 2: x does **not** have a *right* child.

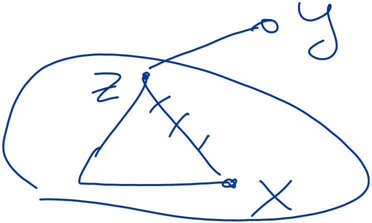
BSTs: Successor

Case 1: x has a *right* child.

- *Successor*(x) must be the ***minimum*** in the ***right subtree*** of x.

Case 2: x does **not** have a *right* child.

- x is the maximum in some subtree A (because x has no right child).
- Maximum of a subtree is the last node visited in the subtree in an inorder traversal. So, the successor y of x is visited right after finishing A.
- Right after finishing visiting a left subtree, its parent is visited.
  So A must be the **left subtree of y**.



BSTs: Successor

Case 1: x has a *right* child.

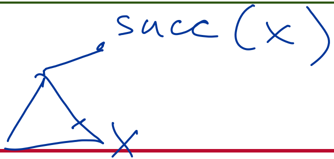
- *Successor*(x) must be the ***minimum*** in the ***right subtree*** of x.

Case 2: x does **not** have a *right* child.

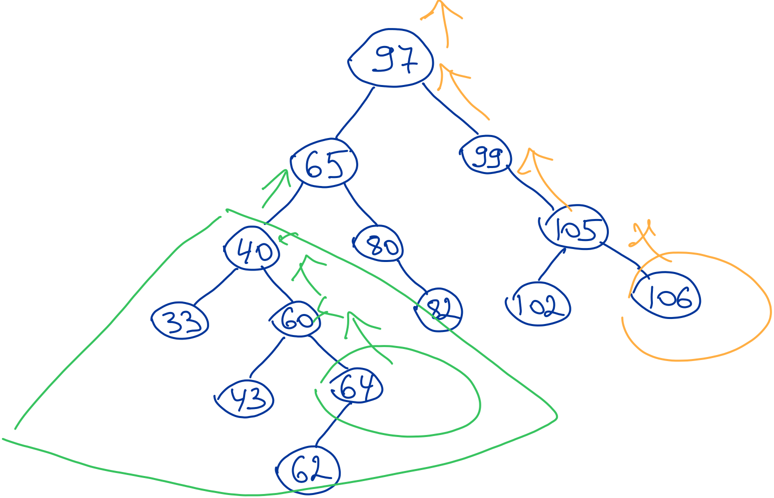
- x is the maximum in some subtree A (because x has no right child).
- So, the successor y of x is visited right after finishing A in inorder traversal.
- So A must be the **left subtree of y**.

- Go up to $x.p$.
- If x is a right child of $x.p$, keep going up.
- If x is a left child of $x.p$, stop, $x.p$ is the guy!

what if we reach the root of the tree?



Then x is already Max so it does not have a successor



65 is Successor
of 64

```
Successor(x):
    if x.right ≠ NIL:
        return BSTMinimum(x.right)
    y = x.p
    while y ≠ NIL and x == y.right:        #x is right child
        x = y
        y = y.p        #keep going up
    return y
```

$h \in O(n)$

Worst case running time:

Case 1: BST Minimum — takes $\Theta(h)$ or $\Theta(n)$

Case 2: Going from a leaf at height h to root. Also $\Theta(n)$

BSTs: Delete

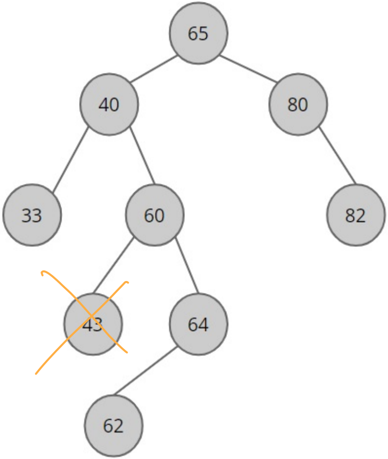
Delete(D,x): remove x from D.

Case 1: x has no child.

- Just delete it

Case 2: x has one child.

Case 3: x has two children.



BSTs: Delete

Delete(D,x): remove x from D.

Case 1: x has no child.

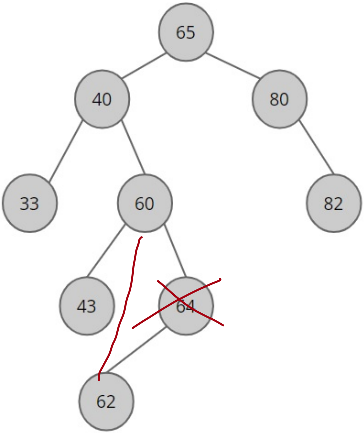
- Just delete it

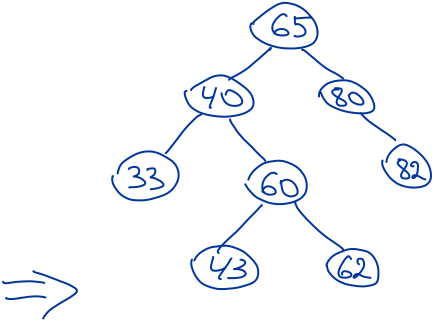
Case 2: x has one child.

- Delete x;
- **Promote** x's only child to x's spot, together with the the child's subtree.

Case 3: x has two children.

Delete 64





```
BSTTransplant(root, x, y):
    # promote y to x's position
    if x.p == NIL :        # if x is the root
        root = y        # y replaces x as root
    else if x == x.p.left:        # if x is its parent's left child
        x.p.left = y        # make y the new left child of the parent
    else:        # if x is its parent's right child
        x.p.right = y        # make y the new right child of the parent
    if y ≠ NIL :
        y.p = x.p        # update the parent of y
```

BSTs: Delete

Delete(D,x): remove x from D.

Case 1: x has no child.

- Just delete it

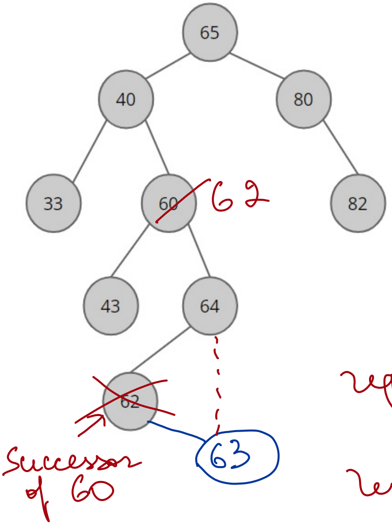
Case 2: x has one child.

- Delete x;
- **Promote** x's only child to x's spot, together with the the child's subtree.

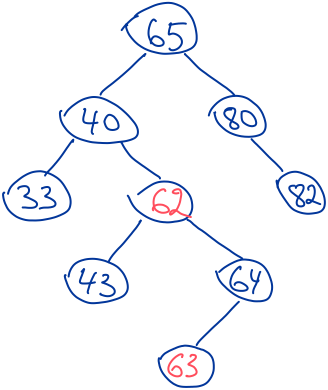
Case 3: x has two children.

- Delete x;
- Replace x by its successor y;
- **Remove y** from its original position;
  - y must be the minimum of right subtree of x.
    So y has at most one child.

Delete 60



$\Rightarrow$



replace 60 by 62

remove old node 62

Side Note: Thinking Process

I can see that it works, pretty clever!
But HOW ON EARTH can I come up with this kind of clever algorithms!

Thinking Process:

- Understand the BST property (the invariant).
- predict the final shape of the tree.
- see how to get there.

BSTDelete$(root, x)$:
    if $root$ == NIL:
        pass # do nothing
    else if $root.key == x.key$:
        $root$ = *BSTExtractMin*$(root.right)$ # or $root$ = *BSTExtractMax*$(root.left)$
    else if $root.key > x.key$:
        BSTDelete$(root.left, x)$
    else:
        BSTDelete$(root.right, x)$

Successor of x.key

predecessor of x.key

$\Theta(n)$

Worst-case Running Time:

- BSTSearch: $\Theta(h)$
- BSTInsert: $\Theta(h)$
- BSTDelete: $\Theta(h)$

But $h \in O(n)$ so all three are $\Theta(n)$

A BST is **NOT** necessarily complete (unlike binary heap).

After Lecture

- Review BSTs in the Course Notes (first part of Chapter 3) and CLRS (CLRS Chapter 12.3).

- Relevant exercises in the Course Notes and CLRS.

- Complete implementation of BSTExtractMin and BSTExtractMax.