CS146 Data Structures and Algorithms

THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

INTRODUCTION TO

ALGORITHMS

THIRD EDITION
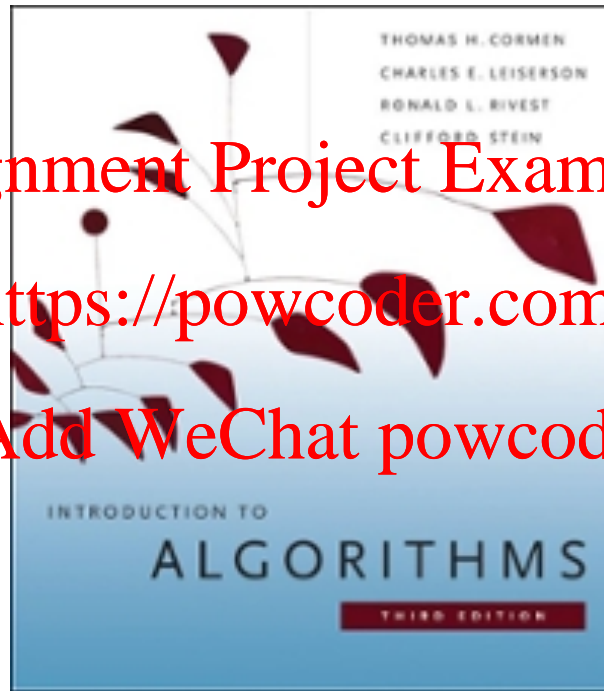
Chapter 12: Binary Search Tree

BST: Dynamic Sets

- Next few lectures will focus on 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:
    - o **Search(S, k), Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x)**

  - They may also support *modifying operations* like:
    - o **Insert(S, x), Delete(S, x)**

- Basic operations take time proportional to the height of the tree – *O(h)*.

Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets

- Represented by a linked data structure of nodes.

- In addition to satellite data, elements have:

  - *key*: an identifying field inducing a total ordering

  - *left*: pointer to a left child : root of left subtree (may be NULL)

  - *right*: pointer to a right child : root of right subtree (may be NULL)
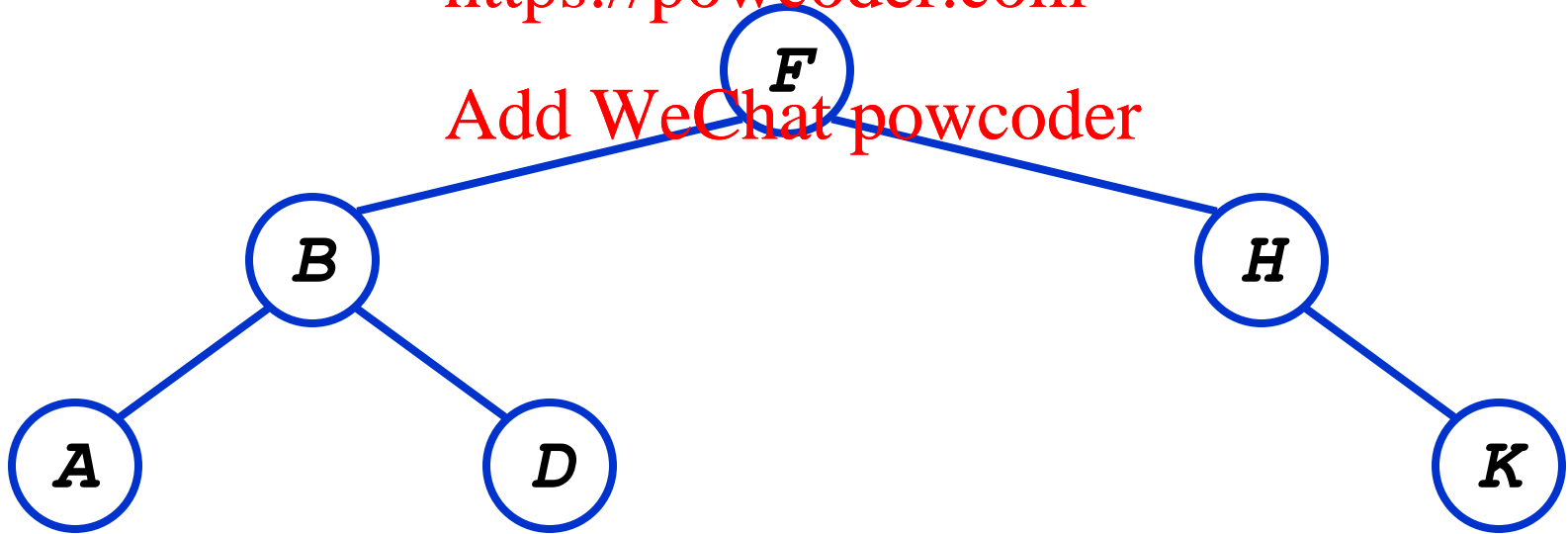
  - *p*: pointer to a parent node (NULL for root)

Binary Search Trees

- BST property:
  key[leftSubtree(x)] ≤ key[x] ≤ key[rightSubtree(x)]

- Example:

Binary Search Tree Property

- Stored keys must satisfy the *binary search tree* property.

  - ∀ y in left subtree of x, then $y.key \leq x.key$

  - ∀ y in right subtree of x, then $y.key \geq x.key$.

56

26

200

18

28

190

213

12

24

27
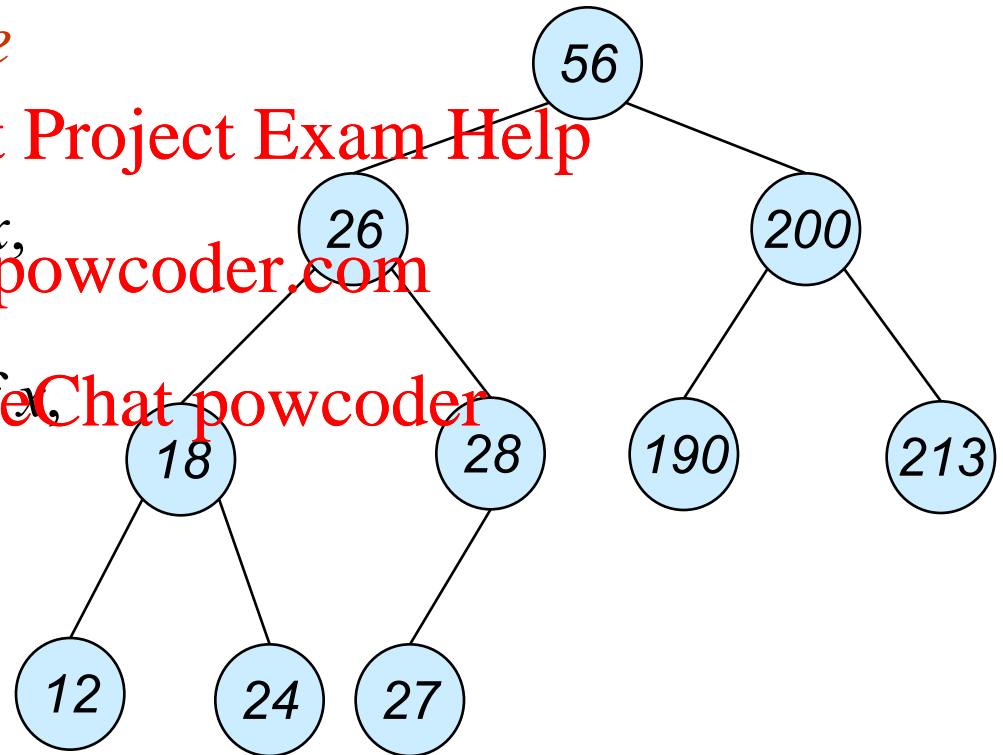
Inorder Traversal

The binary-search-tree property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively.
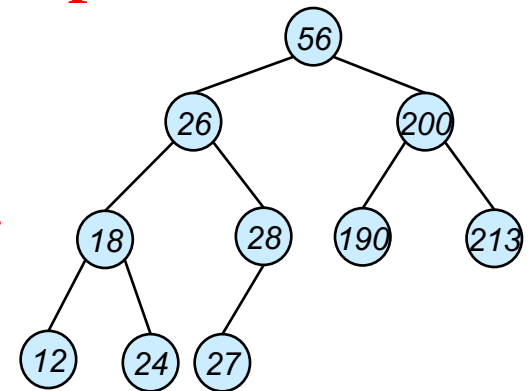
Inorder-Tree-Walk (x)

1. **if** $x \neq$ NIL
2.   **then** Inorder-Tree-Walk($x.left$)
3.   print $x.key$
4.   Inorder-Tree-Walk($x.right$)

◆   How long does the walk take?

◆   Can you prove its correctness?

Correctness of Inorder-Walk

- Must prove that it prints all elements, in order, and that it terminates.

- By induction on size of tree. Size = 0: Easy.

- Size > 1:
  - Prints left subtree in order by induction.
  - Prints root, which comes after all elements in left subtree (still in order).
  - Prints right subtree in order (all elements come after root, so still in order).

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Querying a Binary Search Tree

- All dynamic-set search operations can be supported in $O(h)$ time.

- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)

- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

- A binary tree with n nodes (leaf nodes and internal nodes, including the root node) and height h is balanced if the following is true: $2^{h-1} \leq n < 2^h$. Otherwise it is unbalanced. For example, a binary tree with height 4 can have between 8 and 15 nodes (between 1 and 8 leaf nodes) to be balanced.

Tree Search

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

1. **if** *x* == NIL *or* *k* == *x.key*

2.   **return** *x*

3. **if** *k* < *x.key*

4.   **return** Tree-Search(*x.left*, *k*)

5. **else return** Tree-Search(*x.right*, *k*)

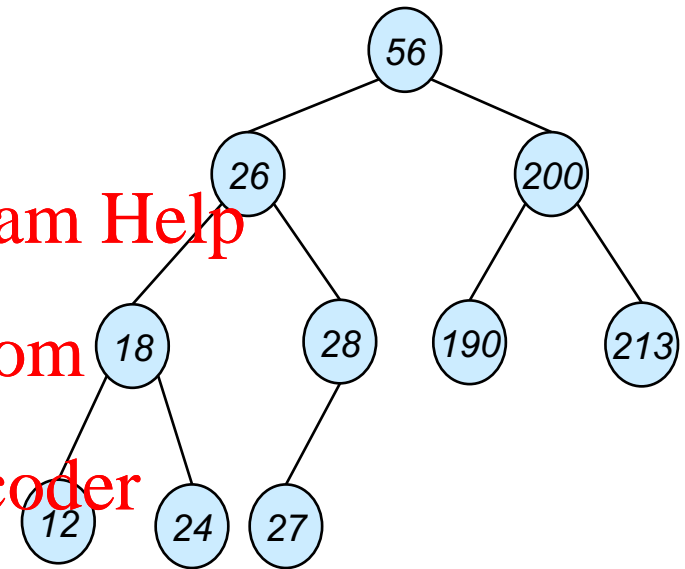**Running time:** *O(h)*
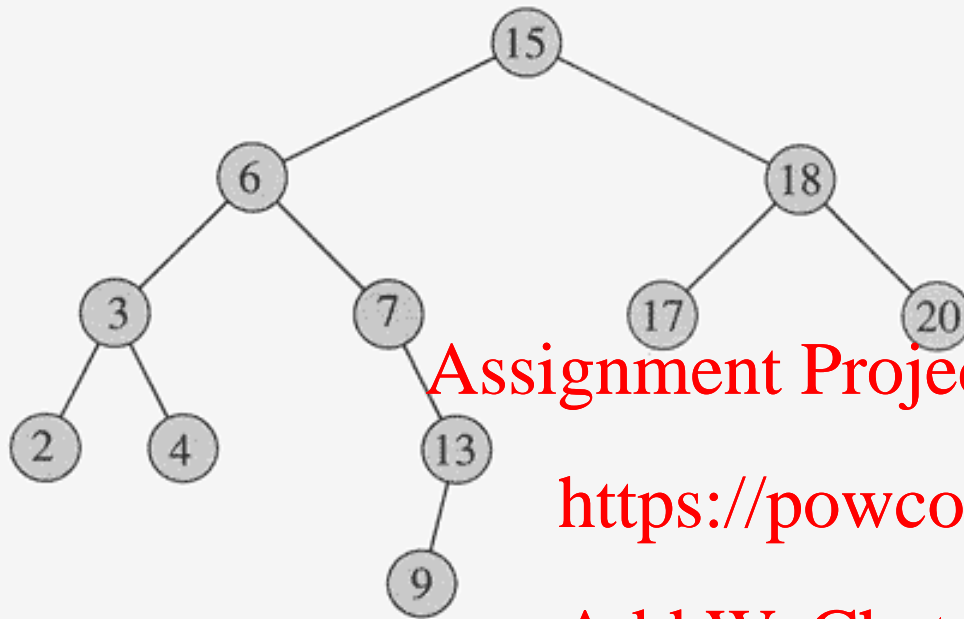
Can we do Tree Search using Iterative approach?
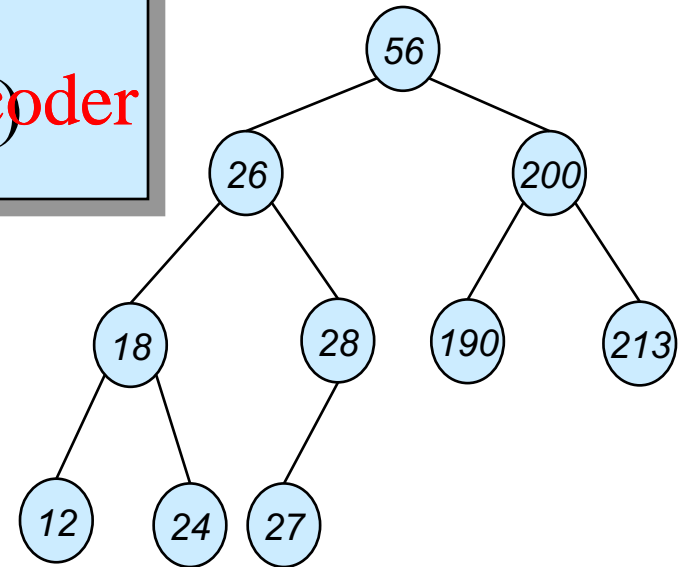
Iterative Tree Search

Iterative-Tree-Search(x, k)

1. **while** $x \neq NIL$ **and** $k \neq x.key$
2.   **if** $k < x.key$
3. $x = x.left$
4.   **else** $x = x.right$
5. **return** x

- The iterative tree search is more efficient on most computers.

- The recursive tree search is more straightforward.

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

How to finding Min & Max

◆The binary-search-tree property guarantees that:

» The minimum is located at the left-most node.

» The maximum is located at the right-most node.

Tree-Minimum(x)	Tree-Maximum(x)
1. **while** $x.left \neq NIL$	1. **while** $x.right \neq NIL$
2. $x = x.left$	2. $x = x.right$
3. **return** x	3. **return** x

Q: How long do they take?

Successor and Predecessor

- Successor of node *x* is the node *y* such that *key*[*y*] is the smallest key greater than *key*[*x*].

- The successor of the largest key is NIL.

- Search consists of two cases.

  - If node *x* has a nonempty right subtree, then *x*'s successor is the minimum in the right subtree of *x*.

  - If node *x* has an empty right subtree, then:

    - o As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.

    - o *x*'s successor *y* is the node that *x* is the predecessor of (*x* is the maximum in *y*'s left subtree).

    - o In other words, *x*'s successor *y*, is the lowest ancestor of *x* whose left child is also an ancestor of *x*.
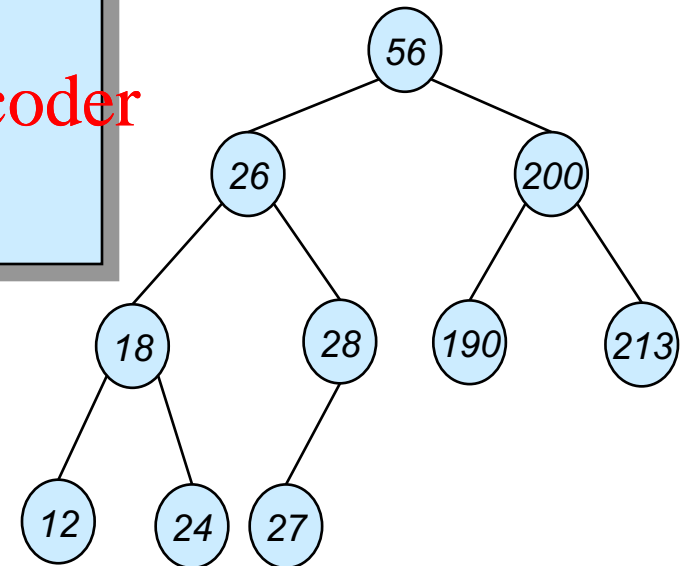
Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** *x.right* ≠ *NIL*

2.   **return** Tree-Minimum(*x.right*)

3. y = *x.p*

4. **while** *y* ≠ *NIL* **and** *x* == *y.right*

5.   *x* = *y*

6.   *y* = *y.p*

7. **return** *y*

BST allows to determine the successor of a node without ever comparing keys.

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder
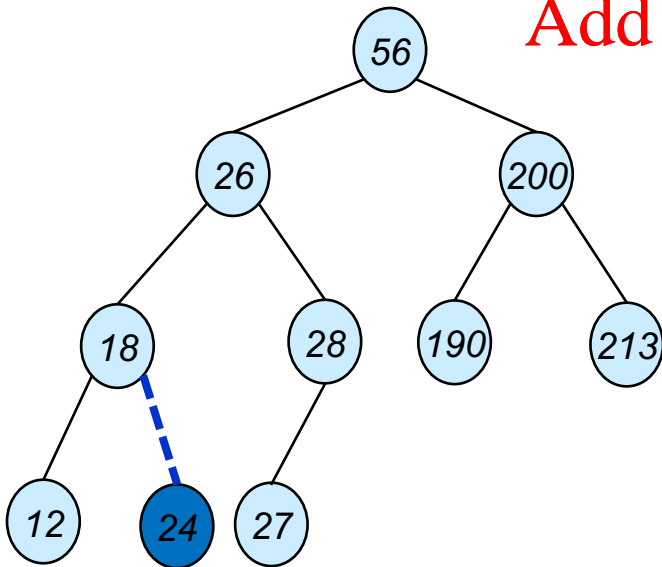
Code for **predecessor** is symmetric.

Running time: O(h)
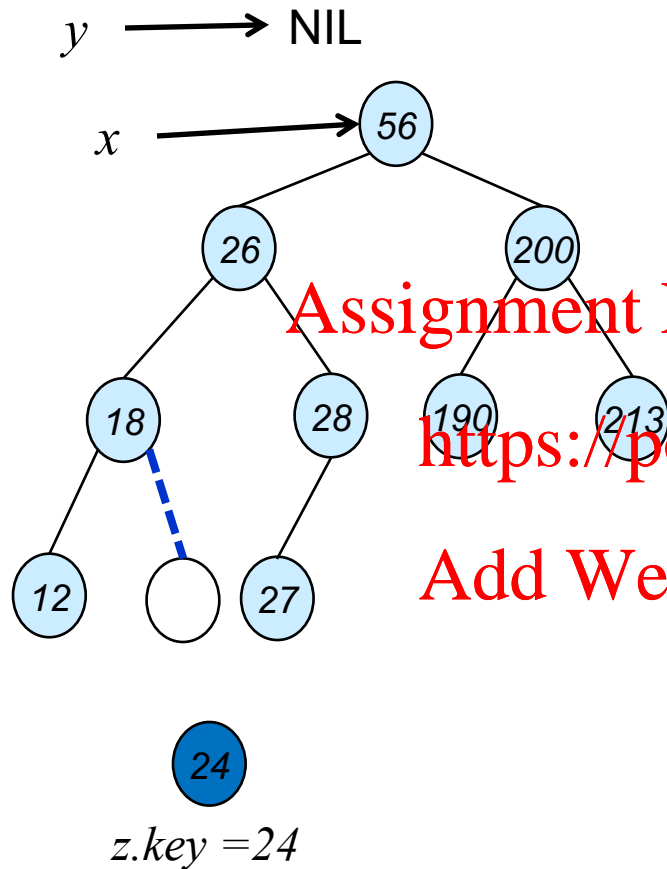
BST Insertion – Pseudocode

- Change the dynamic set represented by a BST.

- Ensure the binary-search-tree property holds after change.

- Insertion is easier than deletion.

Tree-Insert(T, z)

1. y = NIL   /* trailing pointer, p of x */
2. x = $T.root$
3. **while** $x \neq$ NIL
4. y = x
5.   **if** $z.key < x.key$
6. x = $x.left$
7.   **else** x = $x.right$
8. $z.p$ = y
9. **if** y == NIL
10. $T.root$ = z. //tree T was empty
11. **else if** $z.key < y.key$
12. $y.left$ = z
13. **else** $y.right$ = z

BST: Insertion (Initialization)

$y \longrightarrow$ NIL

$x \longrightarrow$ 56

26   200

18   28   190   213

12   27

24

z.key =24

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Tree-Insert(*T*, *z*)
1. *y* = NIL   /* trailing pointer, *p* of *x* */
2. *x* = *T.root*
3. **while** *x* ≠ NIL
4.   *y* = *x*
5.   **if** *z.key* < *x.key*
6.   *x* = *x.left*
7.   **else** *x* = *x.right*
8. *z.p* = *y*
9. **if** *y* == NIL
10.   *T.root* = *z*. //tree *T* was empty
11. **else if** *z.key* < *y.key*
12.   *y.left* = *z*
13. **else** *y.right* = *z*

BST: Insertion (After While Loop)
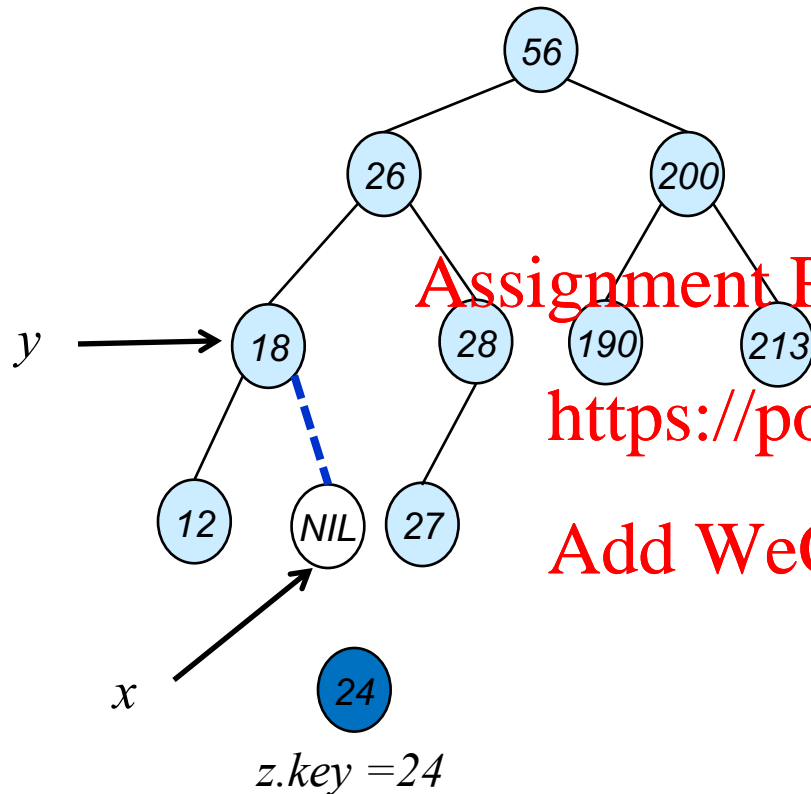
56
26   200
18   28 190 213
12 NIL 27
24

z.key = 24

y →

x →

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Tree-Insert(T, z)
1. $y =$ NIL   /* trailing pointer, p of x */
2. $x = T.root$
3. **while** $x \neq$ NIL
4. $y = x$
5.   **if** $z.key < x.key$
6. $x = x.left$
7.   **else** $x = x.right$
8. $z.p = y$
9. **if** $y ==$ NIL
10. $T.root = z$. //tree T was empty
11. **else if** $z.key < y.key$
12. $y.left = z$
13. **else** $y.right = z$

BST: Insertion (Line 9 ~ 13)
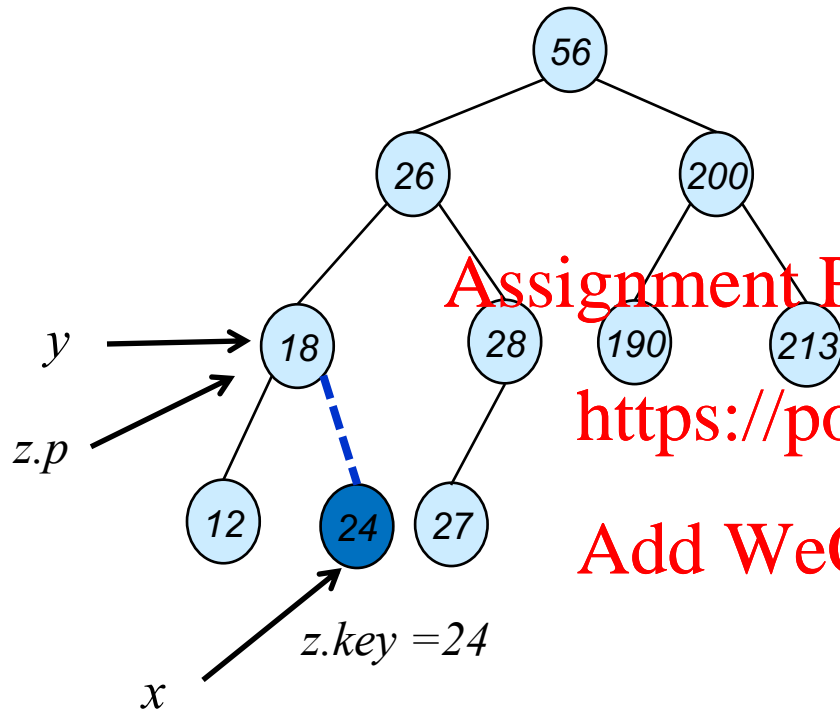
Tree-Insert(*T*, *z*)

1. *y* = NIL   /* trailing pointer, *p* of *x* */
2. *x* = *T.root*
3. **while** *x* ≠ NIL
4.   *y* = *x*
5.   **if** *z.key* < *x.key*
6.   *x* = *x.left*
7.   **else** *x* = *x.right*
8. *z.p* = *y*
9. **if** *y* == NIL
10.   *T.root* = *z*. //tree *T* was empty
11. **else if** *z.key* < *y.key*
12.   *y.left* = *z*
13. **else** *y.right* = *z*

y

z.p

x

z.key =24

56
26   200
18   28   190   213
12   24   27

BST Animation1   BST Animation2

BST: Insertion

- 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31
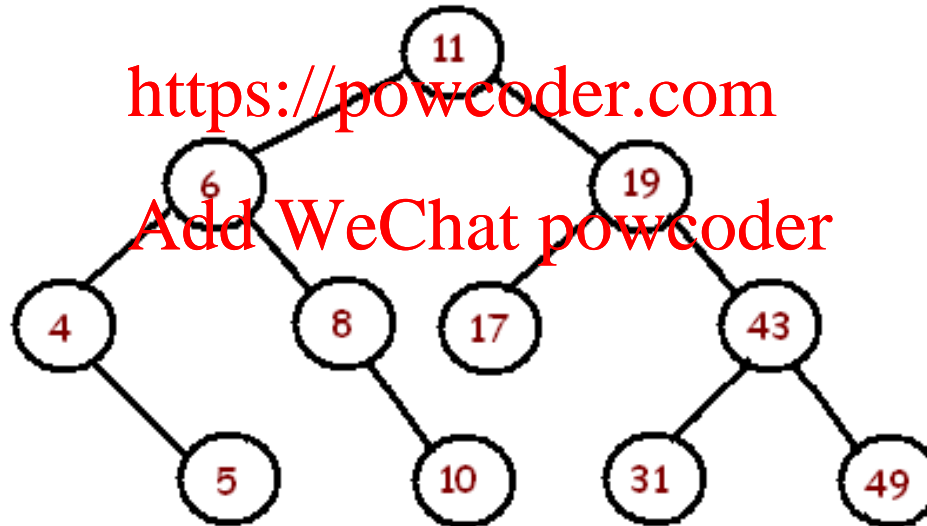
Analysis of Insertion

- Initialization: $O(1)$

- While loop in lines 3-7 searches for place to insert z, maintaining parent y.
  This takes $O(h)$ time.

- Lines 8-13 insert the value: $O(1)$

$\Rightarrow$ TOTAL: $O(h)$ time to insert a node.

Tree-Insert(T, z)
1. $y = $ NIL
2. $x = T.root$
3.   **while** $x \neq$ NIL
4. $y = x$
5.   **if** $z.key < x.key$
6. $x = x.left$
7.   **else** $x = x.right$
8. $z.p = y$
9.   **if** $y ==$ NIL
10. $T.root = z$. //tree T was empty
11.   **else if** $z.key < y.key$
12. $y.left = z$
13.   **else** $y.right = z$

Exercise: Sorting Using BSTs

BSTSort (*A*)

   for *i* ← 1 to *n*

      do Tree-Insert(*A*[*i*])

Inorder-Tree-Walk(*root*)

- What are the worst case and best case running times?

- In practice, how would this compare to other sorting algorithms?

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 Ω(n lg n)*

- *What will be the running time in the*

  - *Worst case?*

  - *Average case? (hint: remind you of anything?)*

Sorting With BSTs
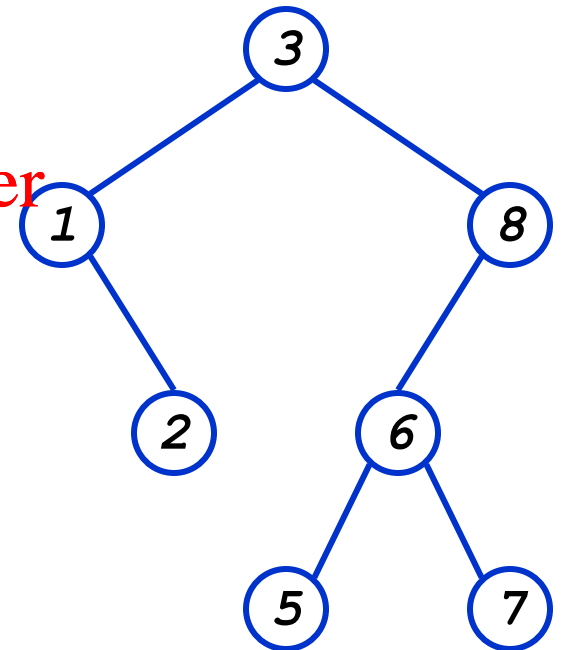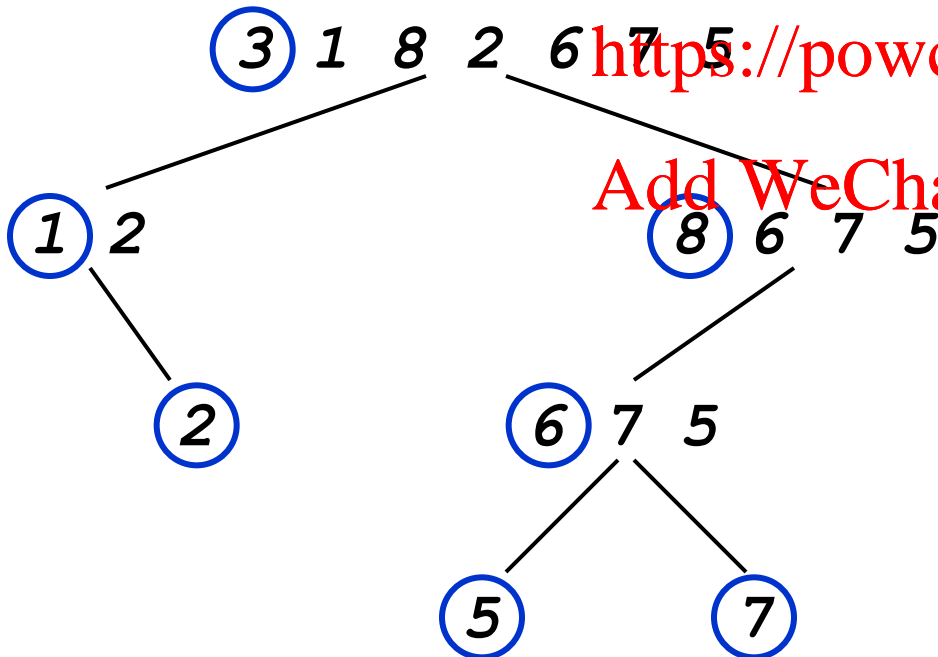
- Average case analysis
  - It's a form of quicksort!

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

3 1 8 2 6 7 5

1 2

2

8 6 7 5

6 7 5

5   7

3
1   8
2   6
    5   7

Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order

  - In previous example

    - o Everything was compared to 3 once

    - o Then those items < 3 were compared to 1 once

    - o Etc.

  - Same comparisons as quicksort, different order!

    - o 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

Tree-Delete (*T, x*)

- Deletion is a bit tricky

- 3 cases:

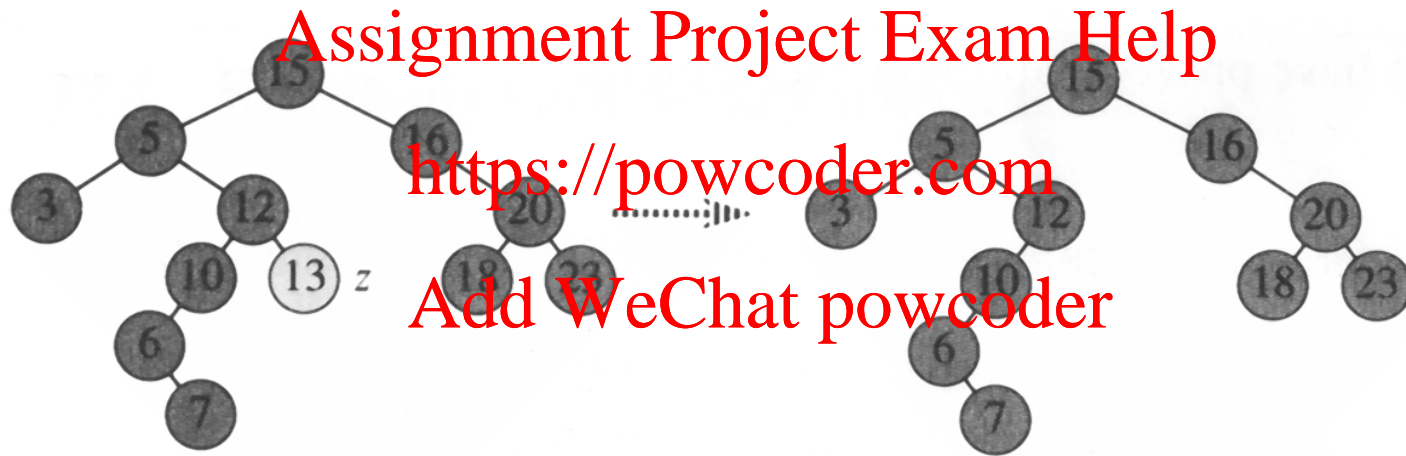if *x* has no children   ♦ case *a*

   then remove *x*

if *x* has one child   ♦ case *b*

   then make *x.p* point to child

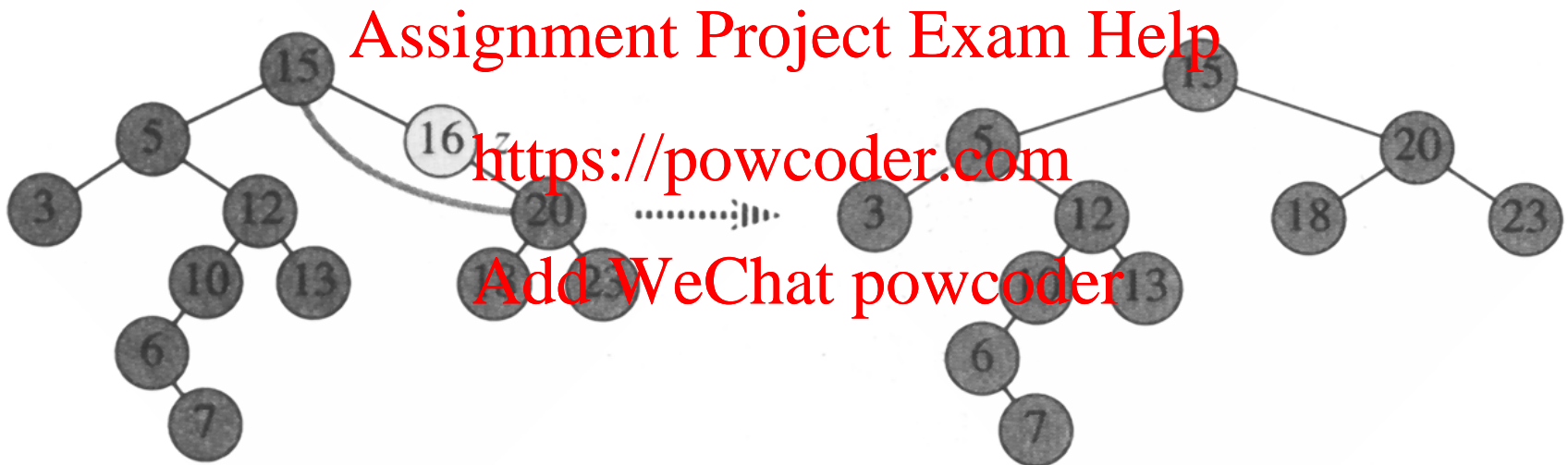if *x* has two children (subtrees)   ♦ case *c*

   then swap *x* with its successor

       perform case *a* or case *b* to delete it

$\Rightarrow$ TOTAL: $O(h)$ time to delete a node

Case a: z has no children

Assignment Project Exam Help

https://powcoder.com

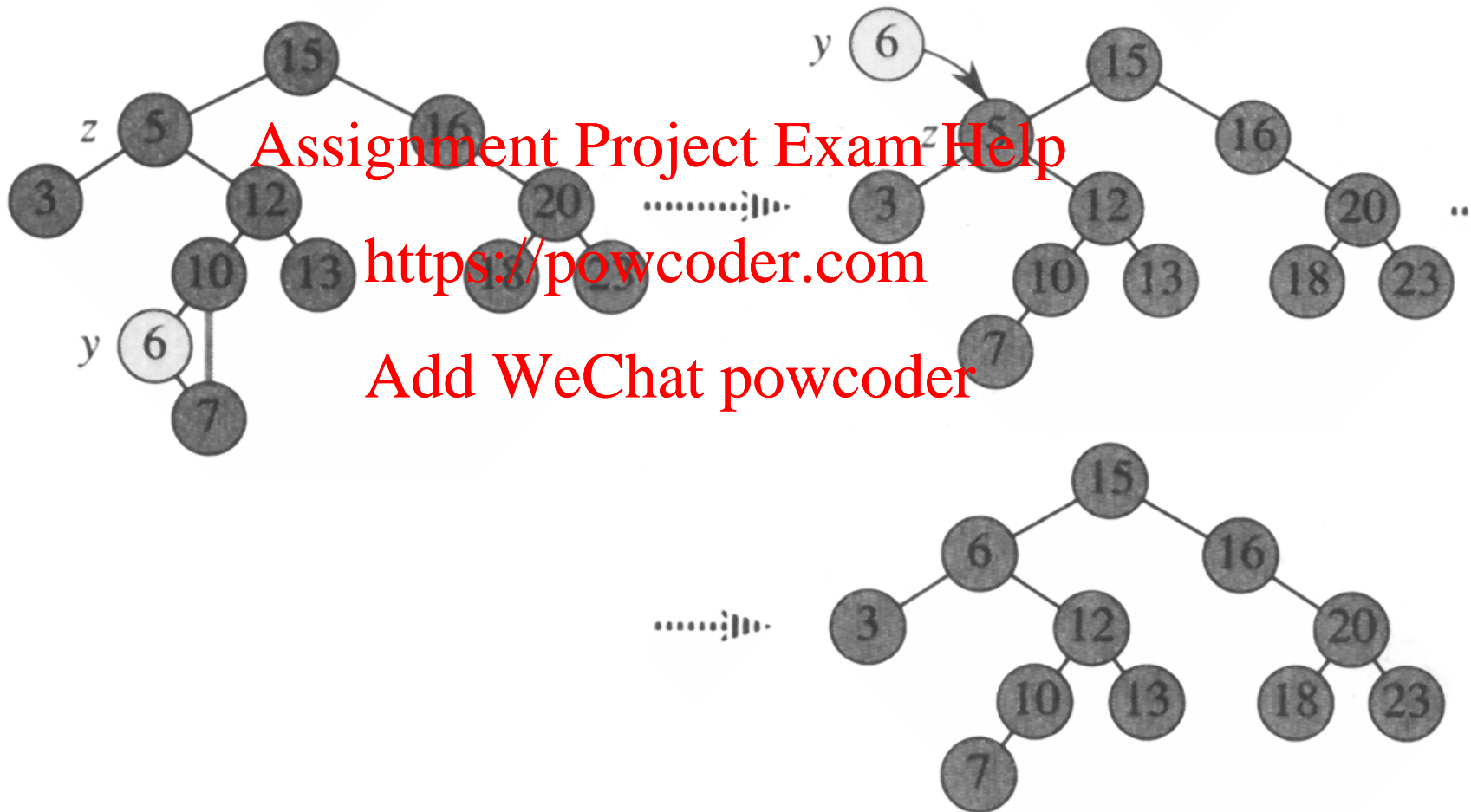Add WeChat powcoder

Case b: z has only one child

Case c: z has two children

Tree-Delete(T, z)
/* Determine which node to splice out: either *z* or *z*'s successor. */
1 **if** *z.left* == NIL or *z.right* == NIL
2   *y = z*
3 **else** *y* = Tree-Successor(*z*)
/* Set *x* to a non-NIL child of *y*, or to NIL if *y* has no children. */
4 **if** *y.left* ≠ NIL
5   *x = y.left*
6 **else** *x = y.right*   Assignment Project Exam Help
/* *y* is removed from the tree by manipulating pointers of *y.p* and *x* */
7 **if** *x* ≠ NIL   https://powcoder.com
8.   *x.p = y.p*
9 **if** *y.p* == NIL   Add WeChat powcoder
10   *root*[*T*] = *x*
11 **else if** *y* == *y.p.left*
12   *y.p.left = x*
13 **else** *y.p.right = x*
/* If *z*'s successor was spliced out, copy its data into *z* */
14 **if** *y* ≠ *z*
15   *z.key = y.key*
16   copy y's satellite data into z
17   **return** *y*

Subtree Replacement - Transplant

- To move subtree around within the subtree.

- Replace one substree rooted at node u as a child in its parent with another sybtree rooted at node v.

- Node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child.

Transplant(T, u, v)

/* *Handle u is root of T* */

1. if $u.p == NIL$

2. $T.root = v$

/* *if u is a left child* */

3.   else if $u == u.p.left$

4. $u.p.left = v$

/* *if u is a right child* */

5.   else $u.p.right = v$

/* *update v.p if v is non-NIL* */

6.   if $v \neq NIL$

7. $v.p = u.p$

Subtree Replacement - Transplant

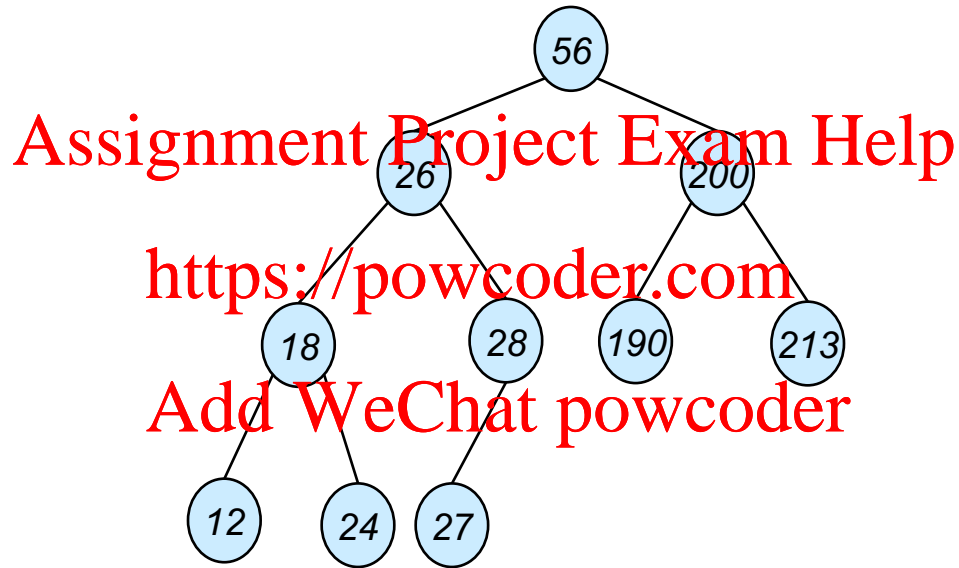**Transplant(*T, u, v*)**

/* *Handle u is root of T* */

1. **if** *u.p == NIL*   /* **if u doesn't have a parent => u is the root** */

2. *T.root = v*   /* **then v must replace u as the root of the tree T** */

/* *if u is a left child* */

3. **else if** *u == u.p.left*   /* **if u is a left subtree of its parent** */

4. *u.p.left = v*   /* **then v must replace u as the left subtree of u's parent** */

/* *if u is a right child* */

5. **else** *u.p.right = v*   /* **otherwise u is a right subtree and v must replace u as the right subtree of u's parent** */

/* *update v.p if v is non-NIL* */

6. **if** *v ≠ NIL*   /* **if v has replaced u (and thus is not NIL)** */

7. *v.p = u.p*   /* **v must have the same parent as u** */

Transplant(*T, 56, 200*)?

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

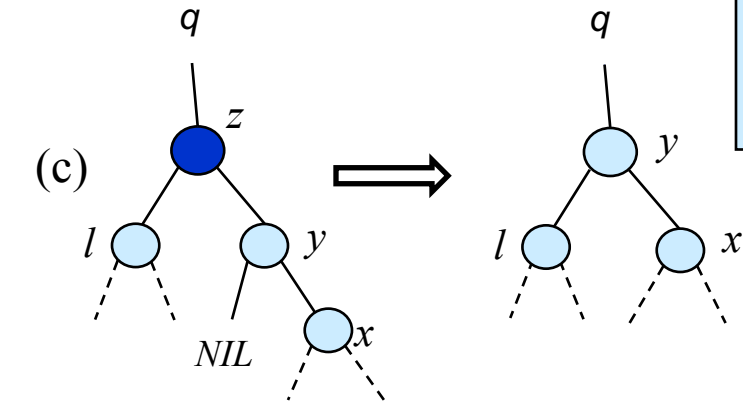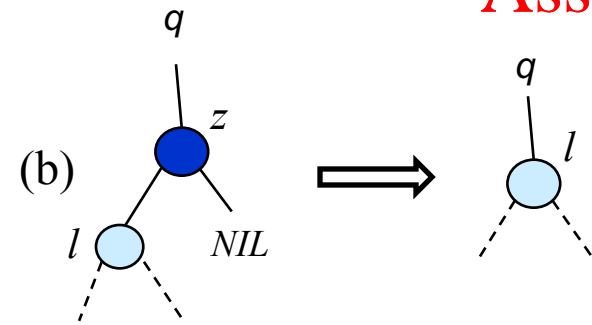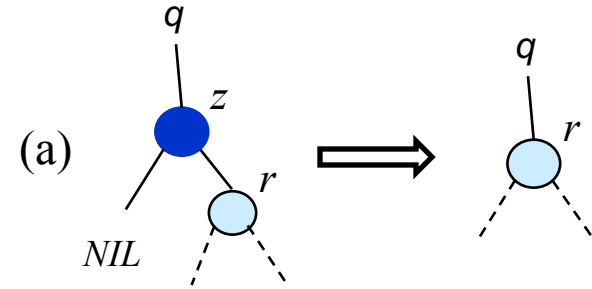# Deletion Using Transplant(T, *u, v*)

Tree-Delete(T, z)

   /* (a) *z* has no left child */

1 **if** *z.left* == NIL

2.   *Transplant(T, z, z.right)*

   /* (b) *z* has a left child, but no right child */

3 **else if** *z.right* == NIL

4   *Transplant(T, z, z.left)*

   /* *z* has two children */

5 **else** *y = Tree-Minimum(z.right)* /* find successor */

6   **if** *y.p ≠ z*   /* (d) y lies within y's right subtree but is not the root of this subtree */

7   *Transplant(T, y, y.right)*

8   *y.right = z.right*

9   *y.right.p = y*

   /* (c) if *y* is *z*'s right child */

10   *Transplant(T, z, y)*

11   *y.left = z.left*   /* replace *y*'s left child by *z*'s left child */

12   *y.left.p = y*

TOTAL: O(h) time to delete a node   BST Animation

Deletion Using Transplant(T, *u, v*)

(a)

(b)

(c)

Tree-Delete(T, z)
    /* (a) *z* has no left child */
1   **if** *z.left* == NIL
2.   *Transplant(T, z, z.right)*
    /* (b) *z* has a left child, but no right child */
3   **else if** *z.right* == NIL
4   *Transplant(T, z, z.left)*
    /* *z* has two children */
5   **else** *y = Tree-Minimum(z.right)* /* find *z*'s successor */
6   **if** *y.p ≠ z* /* (d) y lies within y's right subtree but is not the root of this subtree */
7   *Transplant(T, y, y.right)*
8   *y.right = z.right*
9   *y.right.p = y*
    /* (c) if *y* is *z*'s right child */
10   *Transplant(T, z, y)*
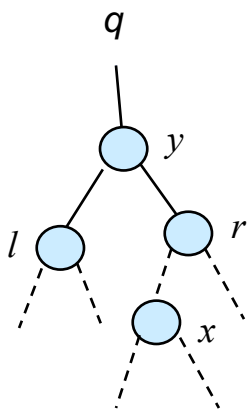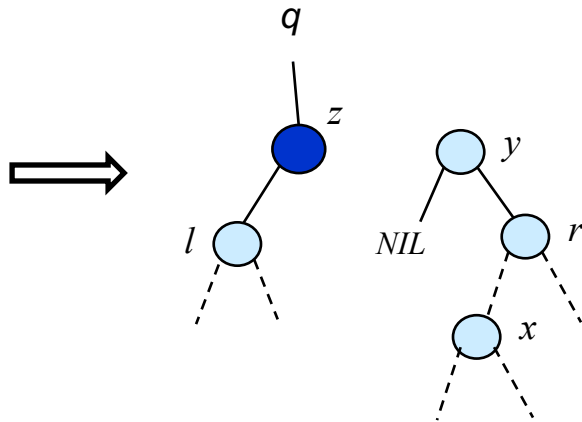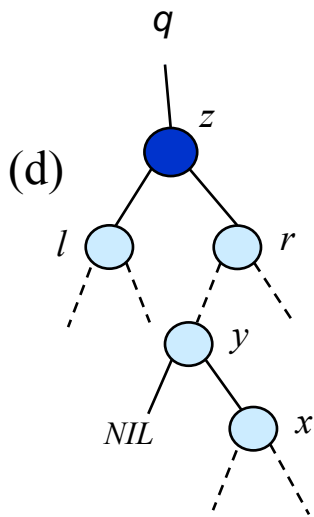11   *y.left = z.left*   /* replace *y*'s left child by *z*'s left child */
12   *y.left.p = y*

Deletion Using Transplant(T, *u, v*)


```
Tree-Delete(T, z)
    /* (a) z has no left child */
1   if  z.left  == NIL
2.          Transplant(T, z, z.right)
    /* (b) z has a left child, but no right child */
3   else if z.right == NIL
4           Transplant(T, z, z.left)
    /* z has two children */
5     else y = Tree-Minimum(z.right)  /* find z's successor */
6         if y.p ≠ z /* (d) y lies within z's right subtree but is not the
                  root of this subtree */
7               Transplant(T, y, y.right)
8               y.right = z.right
9               y.right.p = y
    /* (c) if y is z's right child */
10        Transplant(T, z, y)
11        y.left = z.left     /* replace y's left child by z's left child */
12        y.left.p = y
```

Theorem 12.3

- The dynamic-set operations, INSERT and DELETE can be made to run in O(h) time on a binary search tree of height h.

The End

- The primary purpose of BST is dynamic-set operations: search, insert, and delete.
    - Dynamic operations guarantee a O(lg n) height
    - Comparison based – inorder sorting v.s. Quicksort
    - BST Sorting O(n lg n)

- *Which do you think is better, quicksort or BSTSort? Why?*

- A: quicksort
    - Better constant performance
    - Sorts in place
    - Doesn't need to build data structure tree