BINARY SEARCH TREES

David Kauchak
CS 62 – Spring 2021

1

Administrative

Autocomplete

Lab tomorrow: Q&A + work session (make some progress on assignment!)

Midterm 2 next week

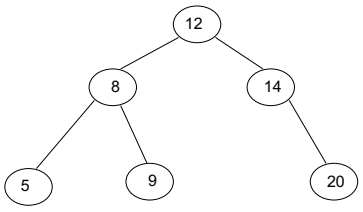
Pre-pre enrollment

2

Binary Search Trees

BST – A binary tree where each each node has a value, and every node's value is:

- Greater than all values in its left subtree. (everything left is smaller)
- Less than or equal to all values in its right subtree. (everything right is larger)



3

Operations

Search – Does the key exist in the tree

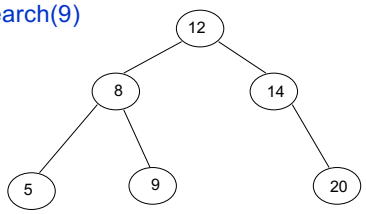
Insert – Insert the key into tree

Delete – Delete the key from the tree

4

Finding an element

Search(9)

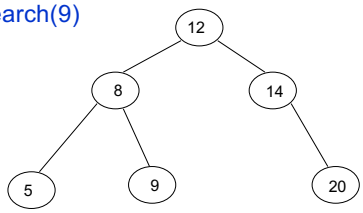


How do we find an element?

5

Finding an element

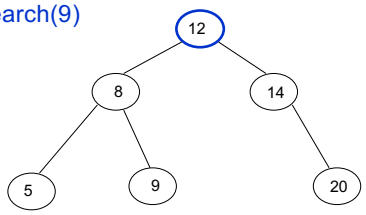
Search(9)



6

Finding an element

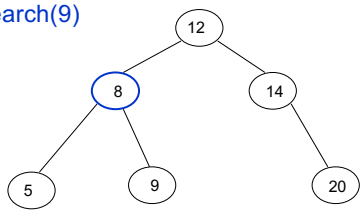
Search(9)



7

Finding an element

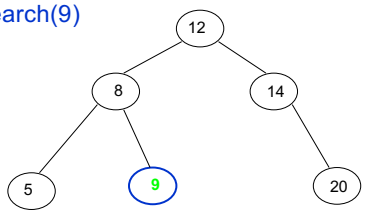
Search(9)



8

Finding an element

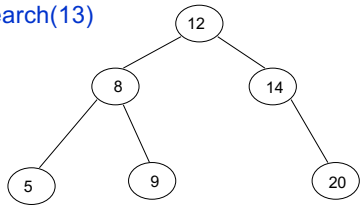
Search(9)



9

Finding an element

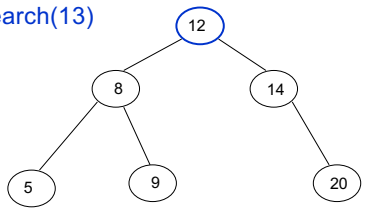
Search(13)



10

Finding an element

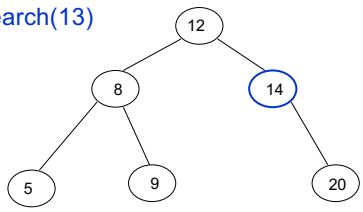
Search(13)



11

Finding an element

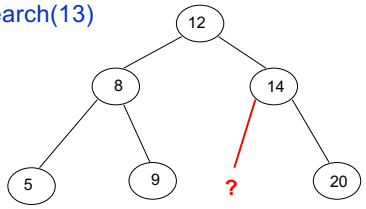
Search(13)



12

Finding an element

Search(13)



13

Finding an element

```
public boolean search(E item){
    return search(root, item);
}

private boolean search(Node c, E item) {
    if( c == null ) {
        return false;
    } else {
        int cmp = item.compareTo(c.value);

        if( cmp == 0 ){
            return true;
        } else if( cmp < 0 ){
            return search(c.left, item);
        }else {
            return search(c.right, item);
        }

    }
}
```

14

Finding an element

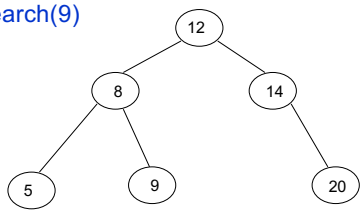
```
private boolean iterativeSearch(Node c, E item) {
    while( c != null && item.compareTo(c.value) != 0 ) {
        if( item.compareTo(c.value) < 0 ){
            c = c.left;
        }else {
            c = c.right;
        }
    }

    return c != null;
}
```

15

Finding an element

Search(9)

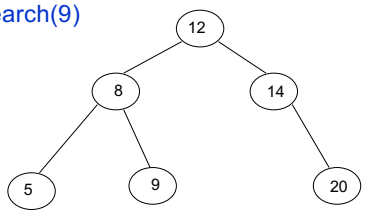


What is the worst case running time of search?

16

Finding an element

Search(9)

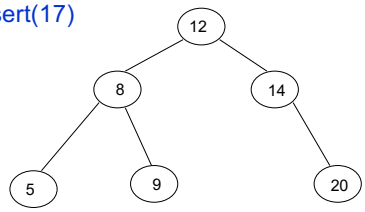


Worst case, have to search to the lowest leaf
O(height)

17

Inserting

Insert(17)

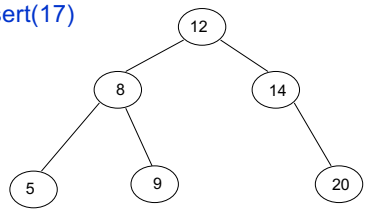


How do we insert an element?

18

Inserting

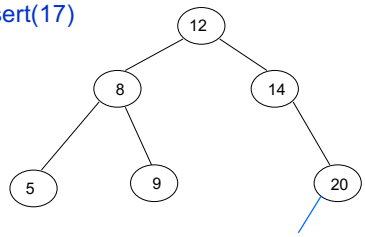
Insert(17)



19

Inserting

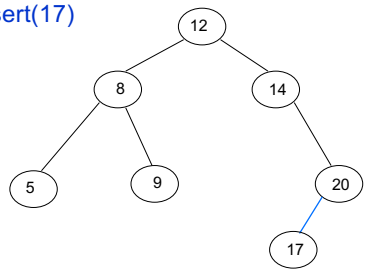
Insert(17)



20

Inserting

Insert(17)



21

Inserting

```
private void insert(Node c, E item) {
    if( item.compareTo(c.value) < 0) {
        if( c.left == null ) {
            c.left = new Node(item);
        } else {
            insert(c.left, item);
        }
    } else {
        if( c.right == null ) {
            c.right = new Node(item);
        } else {
            insert(c.right, item);
        }
    }
}
```

22

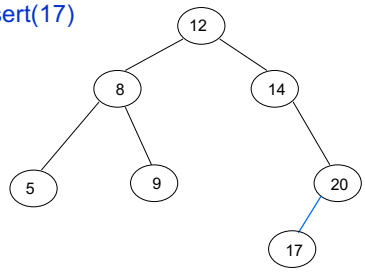
Inserting

```
public void insert(E item){
    if( root == null ) {
        root = new Node(item);
    } else {
        insert(root, item);
    }
}
```

23

Inserting

Insert(17)

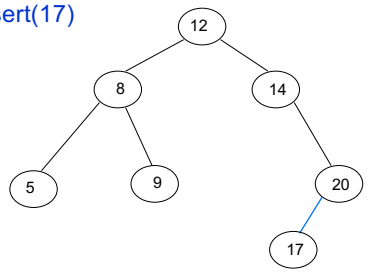


What is the worst case running time of insert?

24

Inserting

Insert(17)

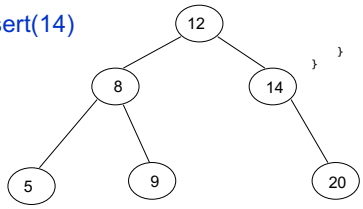


Worst case, have to search to the lowest leaf
O(height)

25

Inserting duplicate

Insert(14)

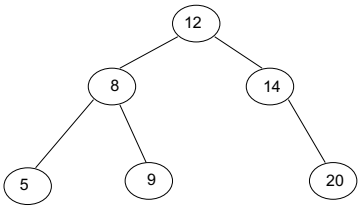



```
private void insert(Node c, E item) {
    if( item.compareTo(c.value) < 0) {
        if( c.left == null ) {
            c.left = new Node(item);
        } else {
            insert(c.left, item);
        }
    } else {
        if( c.right == null ) {
            c.right = new Node(item);
        } else {
            insert(c.right, item);
        }
    }
}
```

26

Visiting all nodes

In sorted order

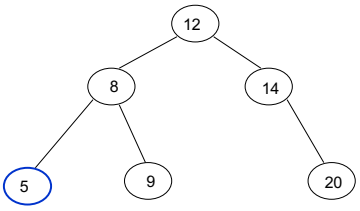


27

Visiting all nodes

In sorted order

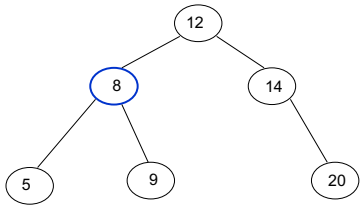
5



28

Visiting all nodes

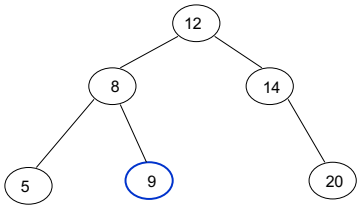
5, 8



29

Visiting all nodes

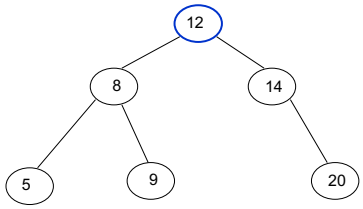
5, 8, 9



30

Visiting all nodes

5, 8, 9, 12

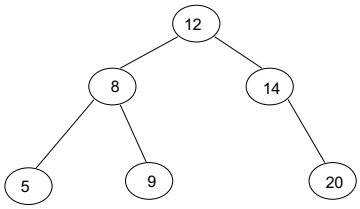


31

Visiting all nodes

What's happening?

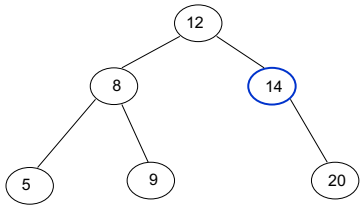
5, 8, 9, 12



32

Visiting all nodes

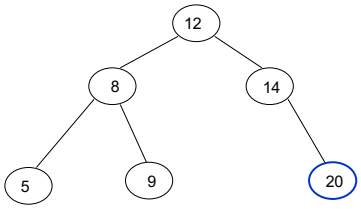
5, 8, 9, 12, 14



33

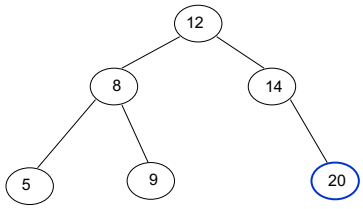
Visiting all nodes

5, 8, 9, 12, 14, 20



34

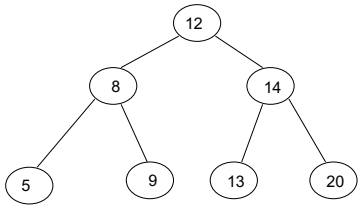
Min/Max?



35

Successor and predecessor

Predecessor(12)? 9

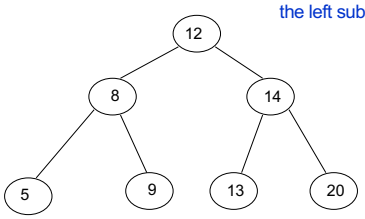


36

Successor and predecessor

Predecessor in general? largest node of all those smaller than this node

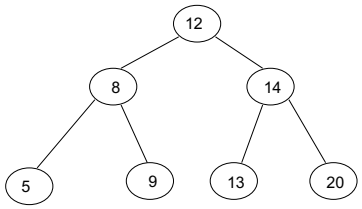
rightmost element of the left subtree



37

Successor

Successor(12)? 13

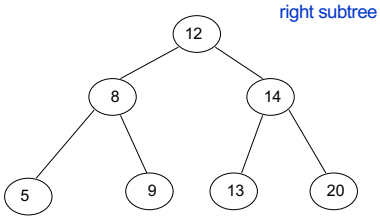


38

Successor

Successor in general? smallest node of all those larger than this node

leftmost element of the right subtree

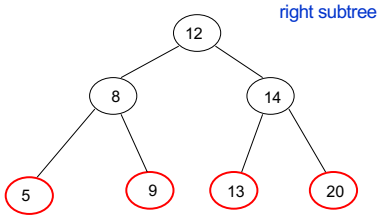


39

Successor

What if the node doesn't have a right subtree? smallest node of all those larger than this node

leftmost element of the right subtree



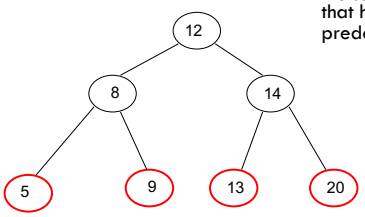
40

Successor

What if the node doesn't have a right subtree?

node is the largest

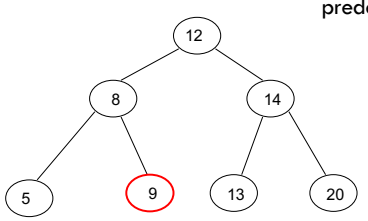
the successor is the node that has x as a predecessor



41

Successor

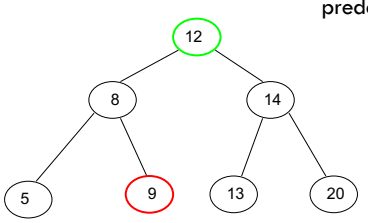
successor is the node that has x as a predecessor



42

Successor

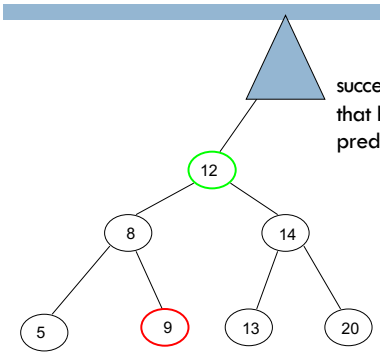
successor is the node that has x as a predecessor



43

Successor

successor is the node that has x as a predecessor

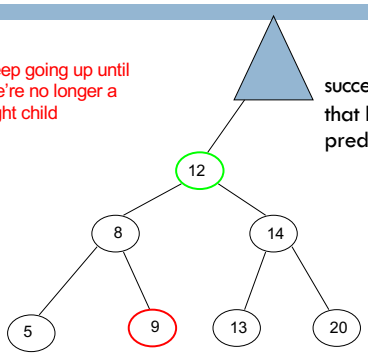


44

Successor

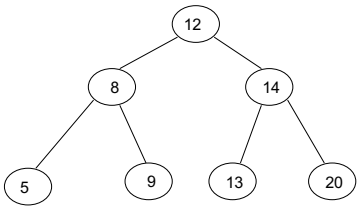
keep going up until we're no longer a right child

successor is the node that has x as a predecessor



45

Deletion



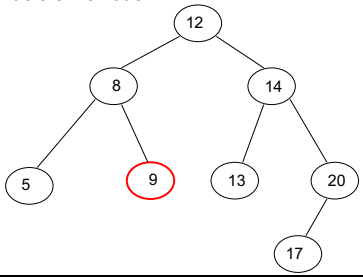
Three cases!

46

Deletion: case 1

No children

Just delete the node

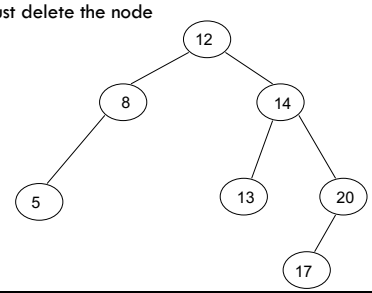


47

Deletion: case 1

No children

Just delete the node

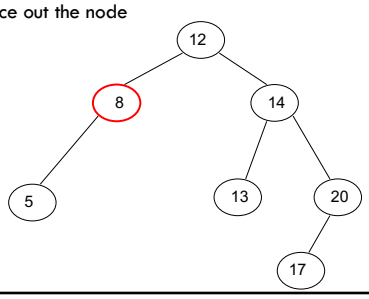


48

Deletion: case 2

One child

Splice out the node

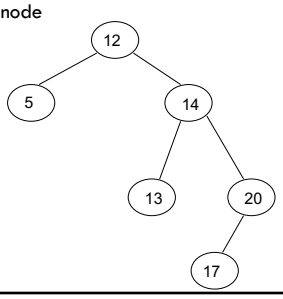


49

Deletion: case 2

One child

Splice out the node

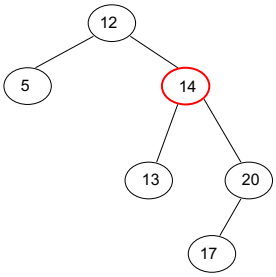


50

Deletion: case 3

Two children

Replace x with the smallest value of the right subtree



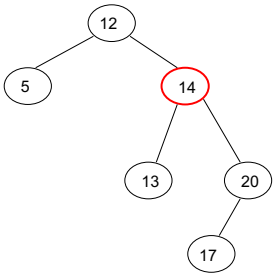
How does this maintain the search tree property?

51

Deletion: case 3

Two children

Replace x with the smallest value of the right subtree



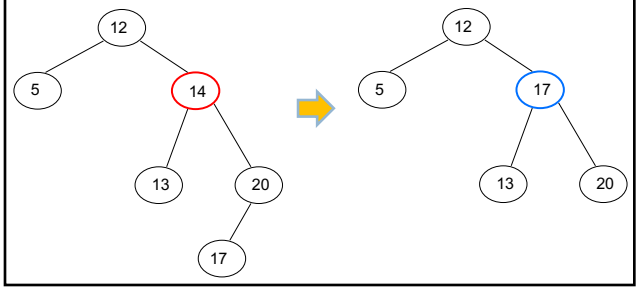
- Larger than everything to the left
- Smaller than everything to the right

52

Deletion: case 3

Two children

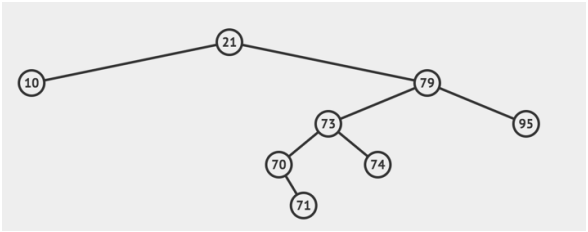
Replace x with the smallest value of the right subtree



53

Deletion

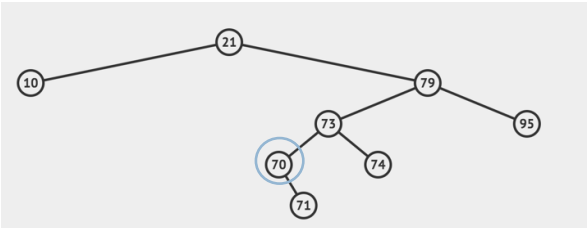
Delete 21



54

Deletion

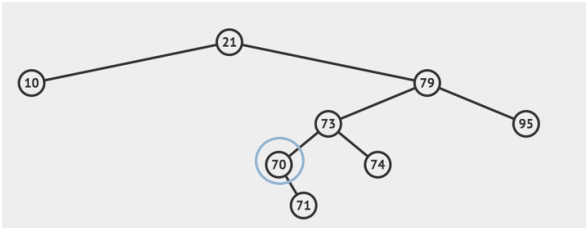
Min of the right subtree



55

Deletion

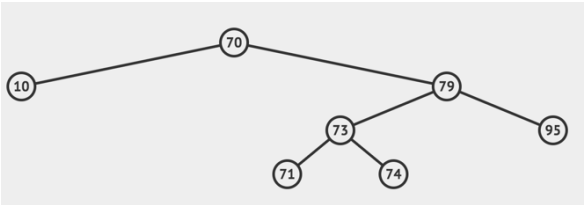
Replace the value: involves a case 2 deletion



56

Deletion

Replace the value: involves a case 2 deletion



57

Deletion: case 3

The min of the right subtree will always be either a case 1 deletion or a case 2 deletion

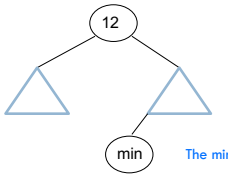
Why?

58

Deletion: case 3

The min of the right subtree will always be either a case 1 deletion or a case 2 deletion

Why?



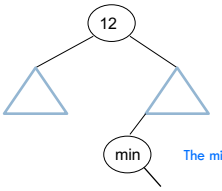
The minimum cannot have a left child

59

Deletion: case 3

The min of the right subtree will always be either a case 1 deletion or a case 2 deletion

Why?

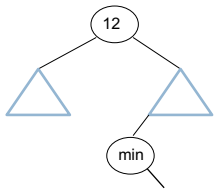


The minimum cannot have a left child

60

Deletion: case 3

The min of the right subtree will always be either a case 1 deletion or a case 2 deletion

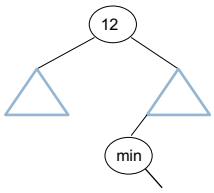


What is the worst case running time of delete?

61

Deletion: case 3

The min of the right subtree will always be either a case 1 deletion or a case 2 deletion



Case 1 and Case 2: O(1)
Case 3: Find min and do a case 1 or case 2 delete
O(height)

62

Height of the tree

Most of the operations take time O(height)

Trees built from random data have height O(log n)

Two problems:

- We can't always insure random data
- What happens when we delete nodes and insert others after building a tree?

Worst case height for binary search trees is O(n) ☹

67

Why BSTs?

Hashtables are fast at search/insert/delete, O(1)

Why BSTs?

68

Why BSTs?

Hashtables are fast at search/insert/delete, O(1)

Min/Max

Successor/predecessor

Inorder traversal

order statistics (5th largest element, etc.)

69

Operations

Search – Does the key exist in the tree

Insert – Insert the key into tree

Delete – Delete the key from the tree

70