PRIORITY QUEUES + BSTS

David Kauchak
CS 62 – Fall 2026

1

Admin

Autocomplete assignment

Checkpoint in lab tomorrow

Hex-A-Pawn assignment out tomorrow

Schedule next week:

- No class Tuesday
- Wed lab asynchronous:
  - May work in pairs
  - Must submit by end of the day Thursday

2

Admin

Declaring the CS major

3

Binary heap

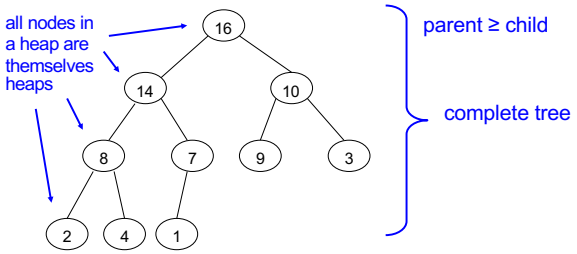
A binary tree where the value of a parent is greater than or equal to the value of its children

Additional restriction: the tree must be **complete**!

Max heap vs. min heap

4

Binary heap - references



5

Binary heap - array

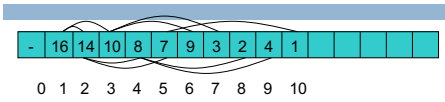
```
PARENT(i)
    return ⌊i/2⌋

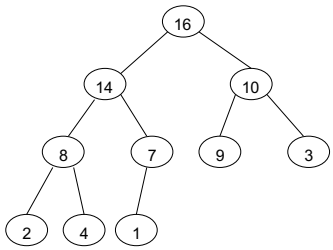
LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

6

Binary heap - array





7

Heaps summarized

Very good at extracting min/max (depending on heap ordering)

	best	worst	average
max			
extractMax		O(log n)	
insert		O(log n)	
change node		O(log n)	

8

Heaps summarized

Very good at extracting min/max (depending on heap ordering)

	best	worst	average
max	O(1)	O(1)	O(1)
extractMax	O(1)	O(log n)	O(log n)
insert	O(1)	O(log n)	O(log n)
change node	O(1)	O(log n)	O(log n)

9

Heapsort

Could we sort data with a heap?

What would be the runtime (best, average, worst)?

	best	worst	average
max	O(1)	O(1)	O(1)
extractMax	O(1)	O(log n)	O(log n)
insert	O(1)	O(log n)	O(log n)
change node	O(1)	O(log n)	O(log n)

10

Heapsort

Build a heap out of the data (e.g., insert n items into heap)

Call extractMin n times and add to answer

11

Heapsort runtime

Build a heap out of the data (e.g., insert n items into heap)

Call extractMin n times and add to answer

Best case?

Worst case?

Average case?

12

Heapsort runtime

Best case? O(n) – when all items have the same value

Worst case? O(n log n)

Average case? O(n log n)

	best	worst	average
max	O(1)	O(1)	O(1)
extractMax	O(1)	O(log n)	O(log n)
insert	O(1)	O(log n)	O(log n)
change node	O(1)	O(log n)	O(log n)

13

Heapsort

Build a heap out of the data (e.g., insert n items into heap*)

Call extractMin n times and add to answer

Stable? No.

In-place? Recursive implementation, no, but can be done without too much trouble iteratively

*there are better ways than this, but we won't cover them in this class)

14

Sorting summarized

	in-place?	stable?	Best	Average	Worst	Notes
Selection	X		$O(n^2)$	$O(n^2)$	$O(n^2)$	n swaps
Insertion	X	X	O(n)	$O(n^2)$	$O(n^2)$	use for partially ordered
Merge		X	O(n log n)	O(n log n)	O(n log n)	guaranteed, stable
Quick	X		O(n log n)	O(n log n)	$O(n^2)$	fastest in practice
Heap	X		O(n)	O(n log n)	O(n log n)	guaranteed, in-place

15

Priority Queues

Queues work well for keeping track of sequential ordering when everything is equivalent (e.g., waiting in line to get lunch!)

Some queues everything is not equivalent (e.g., ER waiting room)

Priority queues support add/remove **ordered by a weight/priority**

16

Priority Queues

Applications?

17

Priority Queues

Applications?

- process scheduling (e.g., 'top' command)
- network traffic scheduling
- Many algorithms
  - Search algorithms (A*)
  - Shortest paths algorithms (Dijsktra's)
  - Minimum spanning trees (Prim's)
  - Huffman codes

18

Priority queue interface

```
public interface PriorityQueue <E extends Comparable<E>>{
    /**
     * Returns the smallest value in the queue if non-empty
     *
     * @return the smallest value in the priority queue
     */
    public E extractMin();

    /**
     * Adds the specified item to the priority queue
     *
     * @param data data item to be added
     */
    public void add(E data);

    /**
     * Returns the number of elements in the queue
     *
     * @return size of the queue
     */
    public int size();

    public boolean isEmpty();
}
```

19

Priority queue

two key methods:

- add
- extractMin (highest priority)

How can we do this?

See how many options you can come up with that have *different* runtimes for operations!

20

Option 1: unordered ArrayList

add:

extractMin:

21

Option 1: unordered ArrayList

add: add to the end of the ArrayList

extractMin: search for the smallest, return and remove it

Worst case running times?

22

Option 1: unordered ArrayList

add: add to the end of the ArrayList
O(1) amortized

extractMin: search for the smallest, return and remove it
O(n)

Worst case running times?

23

Option 1b: unordered LinkedList

add: add to the front or end (with rail reference) of the linked list

extractMin: search for the smallest, return and remove it

Worst case running times?

24

Option 1b: unordered LinkedList

add: add to the front or end (with rail reference) of the linked list

O(1)

extractMin: search for the smallest, return and remove it

O(n)

Worst case running times?

25

Option 1b: unordered LinkedList

```
public class SimpleLinkedListPriorityQueue<E extends Comparable<E>> implements Priority
    private LinkedList<E> pq = new LinkedList<E>();

    public void add(E data) {
        pq.add(data);
    }

    public E extractMin(){
        E min = pq.get(0);

        for( E temp: pq ){
            if( temp.compareTo(min) < 0 ){
                min = temp;
            }
        }

        pq.remove(min);
        return min;
    }

    public boolean isEmpty() {
        return pq.size() == 0;
    }

    public int size() {
        return pq.size();
    }
}
```

26

Option 2: sorted order linked list

add:

extractMin:

27

Option 2: sorted order linked list

add: search for the correct location and insert

extractMin: remove and return the first thing from the list

Worst case running times?

28

Option 2: sorted order linked list

add: search for the correct location and insert
O(n)

extractMin: remove and return the first thing from the list
O(1)

Worst case running times?

29

Option 3: heap

add/insert

extractMin

Worst case running times?

30

Option 3: heap

add/insert
O(log n)

extractMin
O(log n)

Worst case running times?

31

Priority queues summarized

Different scenarios may benefit from different implementations

Priority queue ≠ heap

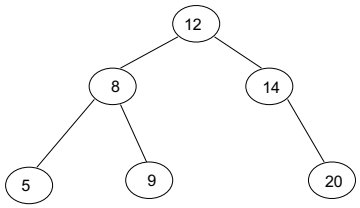
	add	extractMin
unordered linked list	O(1)	O(n)
sorted linked list	O(n)	O(1)
heap	O(log n)	O(log n)

32

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)



33

Operations

Search – Does the key exist in the tree

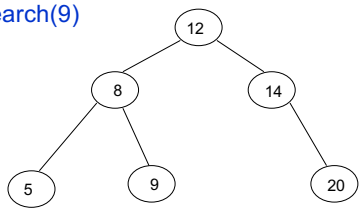
Insert – Insert the key into tree

Delete – Delete the key from the tree

34

Finding an element

Search(9)

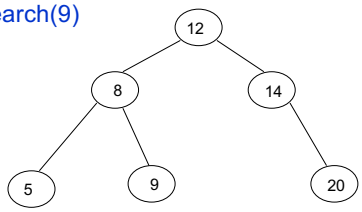


How do we find an element?

35

Finding an element

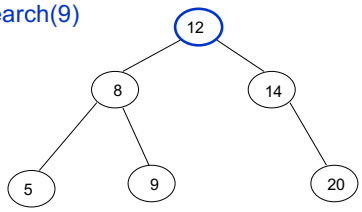
Search(9)



36

Finding an element

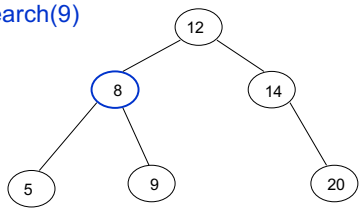
Search(9)



37

Finding an element

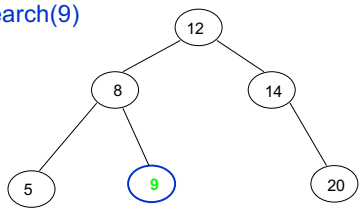
Search(9)



38

Finding an element

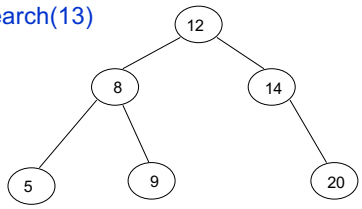
Search(9)



39

Finding an element

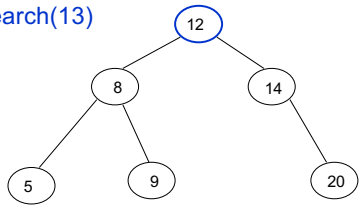
Search(13)



40

Finding an element

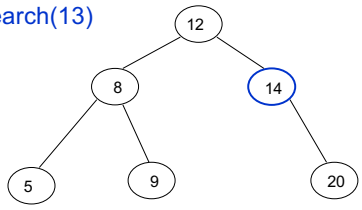
Search(13)



41

Finding an element

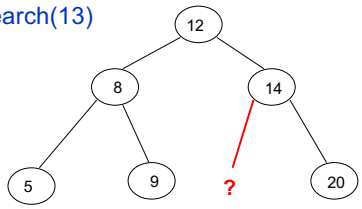
Search(13)



42

Finding an element

Search(13)



43

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);
        }

    }
}
```

44

Finding an element

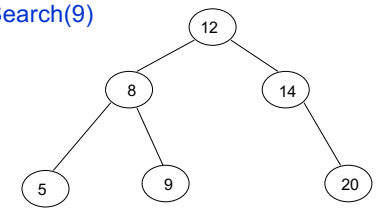
```
private boolean iterativeSearch(Node c, E item) {
    while( c != null && item.compareTo(c.value) != 0 ) {
        if( item.compareTo(c.value) < 0 ){
            c = c.left;
        }else {
            c = c.right;
        }
    }

    return c != null;
}
```

45

Finding an element

Search(9)

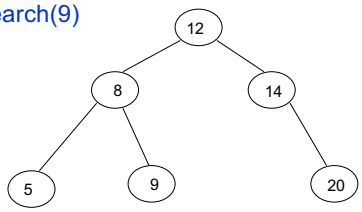


What is the worst case running time of search?

46

Finding an element

Search(9)

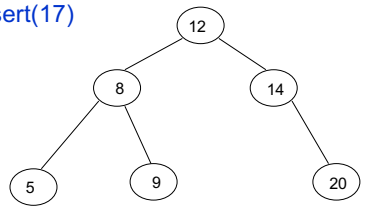


Worst case, have to search to the lowest leaf
O(height)

47

Inserting

Insert(17)

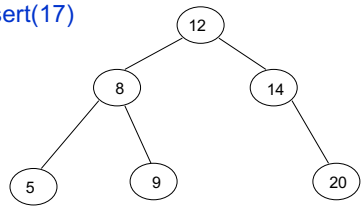


How do we insert an element?

48

Inserting

Insert(17)

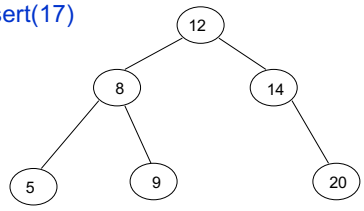


Search until we find a null, add there

49

Inserting

Insert(17)

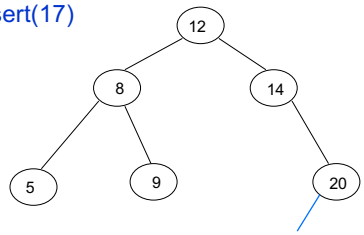


Where would 17 go?

50

Inserting

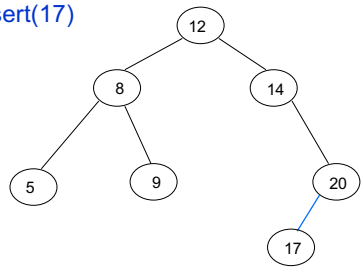
Insert(17)



51

Inserting

Insert(17)



52

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);
        }
    }
}
```

53

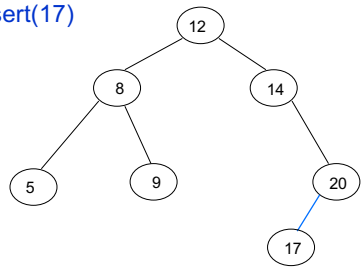
Inserting

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

54

Inserting

Insert(17)

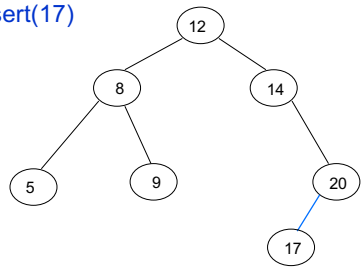


What is the worst-case running time of insert?

55

Inserting

Insert(17)



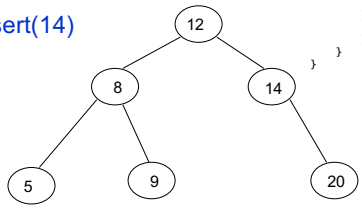
Worst case, have to search to the lowest leaf
O(height)

56

Inserting duplicates

```
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);
        }
    }
}
```

Insert(14)

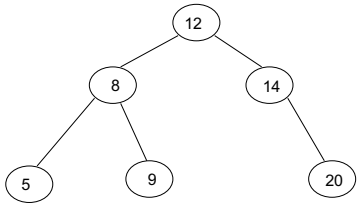


Where would it go?

57

Visiting all nodes

In sorted order

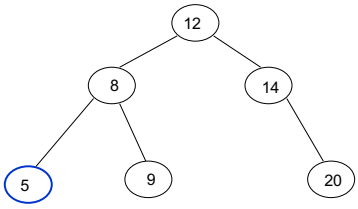


58

Visiting all nodes

In sorted order

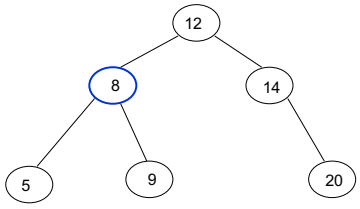
5



59

Visiting all nodes

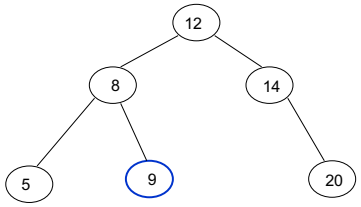
5, 8



60

Visiting all nodes

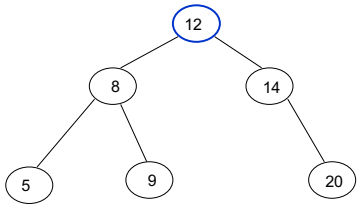
5, 8, 9



61

Visiting all nodes

5, 8, 9, 12

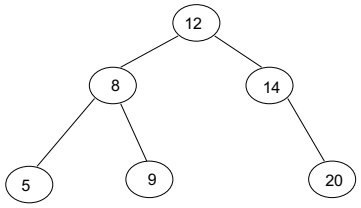


62

Visiting all nodes

What's happening?

5, 8, 9, 12

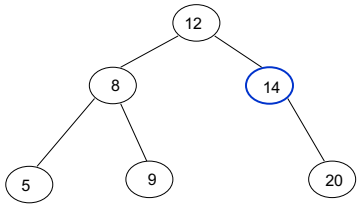


63

Visiting all nodes

InOrder traversal

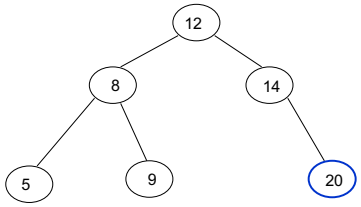
5, 8, 9, 12, 14



64

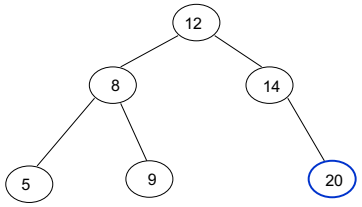
Visiting all nodes

5, 8, 9, 12, 14, 20



65

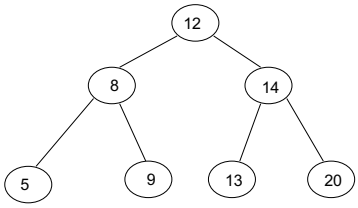
Min/Max?



66

Successor and predecessor

Predecessor(12)? 9



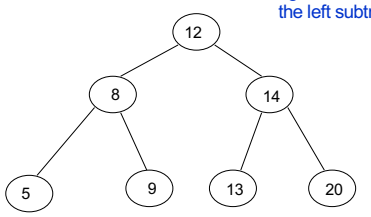
67

Successor and predecessor

Predecessor in general?

largest node of all those smaller than this node

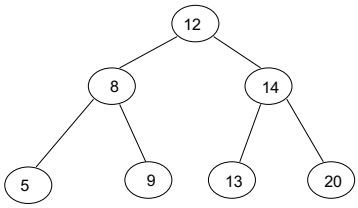
rightmost element of the left subtree



68

Successor

Successor(12)? 13



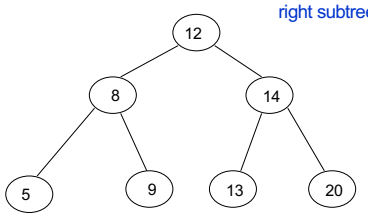
69

Successor

Successor in general?

smallest node of all those larger than this node

leftmost element of the right subtree



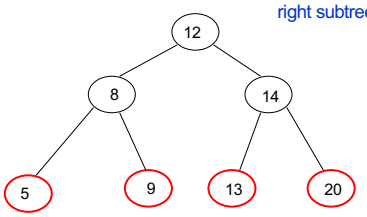
70

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



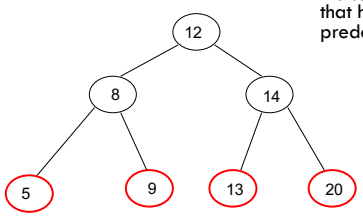
71

Successor

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

node is the largest

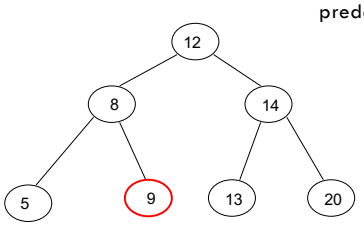
the successor is the node that has x as a predecessor



72

Successor

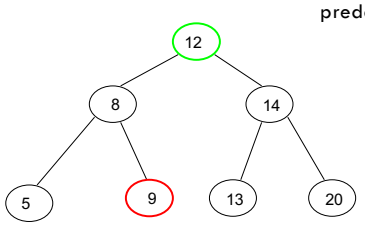
successor is the node that has x as a predecessor



73

Successor

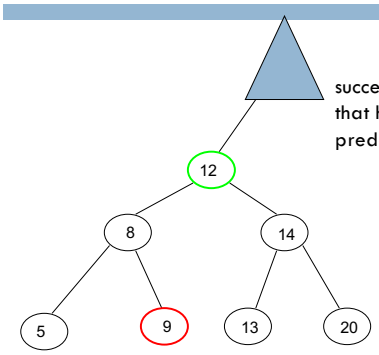
successor is the node that has x as a predecessor



74

Successor

successor is the node that has x as a predecessor

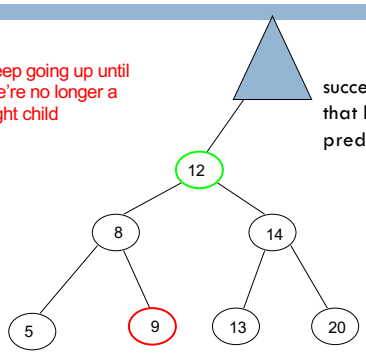


75

Successor

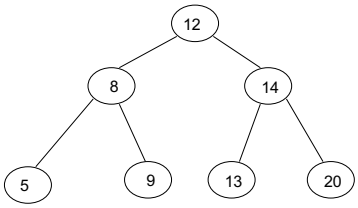
keep going up until we're no longer a right child

successor is the node that has x as a predecessor



76

Deletion



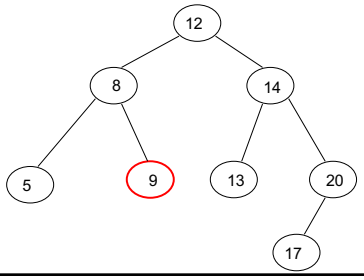
Three cases!

77

Deletion: case 1

No children

Just delete the node

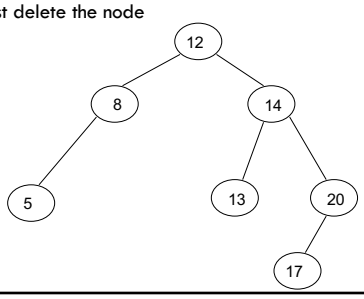


78

Deletion: case 1

No children

Just delete the node

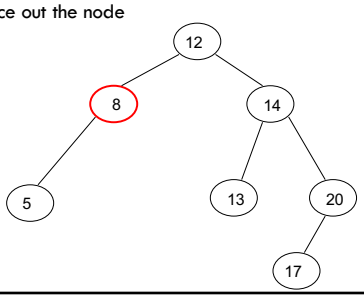


79

Deletion: case 2

One child

Splice out the node

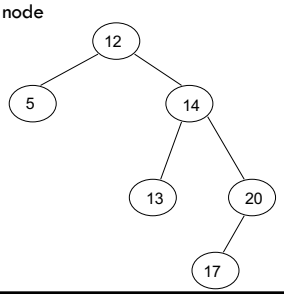


80

Deletion: case 2

One child

Splice out the node

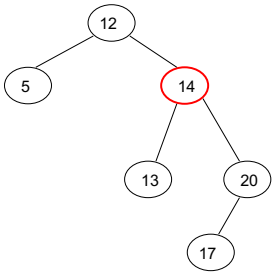


81

Deletion: case 3

Two children

Replace x with the smallest value of the right subtree



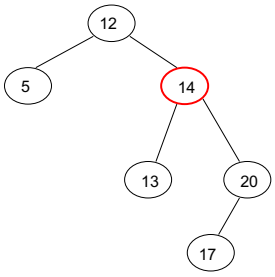
How does this maintain the search tree property?

82

Deletion: case 3

Two children

Replace x with the smallest value of the right subtree



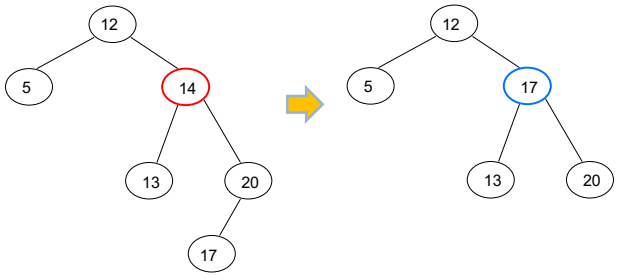
- Larger than everything to the left
- Smaller than everything to the right

83

Deletion: case 3

Two children

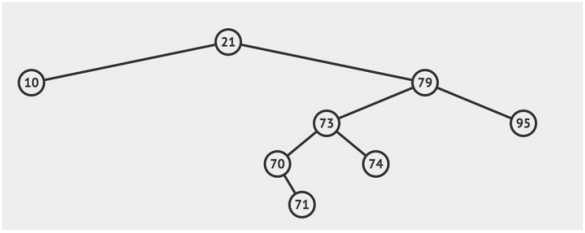
Replace x with the smallest value of the right subtree



84

Deletion

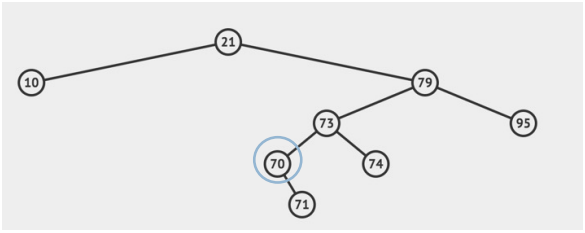
Delete 21



85

Deletion

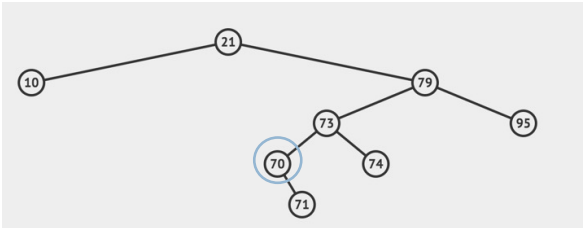
Min of the right subtree



86

Deletion

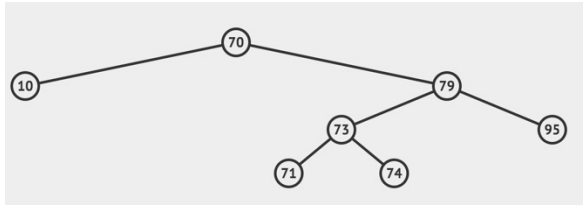
Replace the value: involves a case 2 deletion



87

Deletion

Replace the value: involves a case 2 deletion



88

Deletion: case 3

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

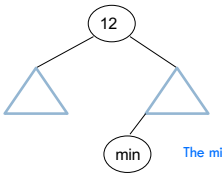
Why?

89

Deletion: case 3

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

Why?



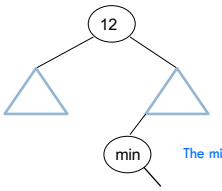
The minimum cannot have a left child

90

Deletion: case 3

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

Why?

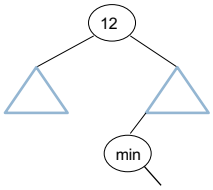


The minimum cannot have a left child

91

Deletion: case 3

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

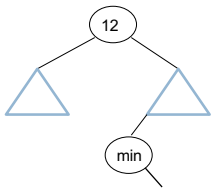


What is the worst case running time of delete?

92

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)

93

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) ☹

98

Why BSTs?

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

Why BSTs?

99

Why BSTs?

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

Min/Max

Successor/predecessor

Inorder traversal

order statistics (5^{th} largest element, etc.)

100

Operations

Search – Does the key exist in the tree

Insert – Insert the key into tree

Delete – Delete the key from the tree

101