hashCodes & Priority Queue ADT

15-121 Fall 2020

Margaret Reid-Miller

Today

- hashCodes
- Priority Queues
- Implementation: Heaps
- Heapsort (maybe)

Runtime

What is the worst-case runtime for contains, add, remove using a hash table?

What is the best-case runtime?

What is the expected runtime?

Load Factor

Load Factor: (number of elements) / (length of array)

What is the expected size of a bucket?

What is a good load factor?

What can you do when the load factor gets too big?

Java Objects

All Java objects have the following 3 methods:

`boolean` **`equals`**`(Object)`

`String` **`toString`**`()`

`int` **`hashcode`**`()` – the value returned may be negative or much larger than the hash table length. Use _____ to convert to a valid hash table index.

What does the `Object` class methods do?

`equals`: checks if same memory address

`toString`: returns "memory address" as a string

`hashCode`: returns "memory address" as an int

If you override `equals` you must also override `hashCode`

HashSet and HashMap:

- uses an object's `hashCode` method to determine the bucket index, and
- then uses the object's `equals` method to see if the object is in the bucket.

Requirement:

If `obj1.equals(obj2)` **then**

`obj1.hashCode() == obj2.hashCode()`

- The two work together and both are necessary for HashSet and HashMap to work correctly.

Object hashCode

Is the default Object hashcode method sufficient? No!

For example, suppose you write

```
Map m = new HashMap();
m.put(new Point(3, 5),  "max");
String label = m.get(new Point(3, 5));
```

What value does label have? `null`

There are two `Point` instances. One added to the map. The other to retrieve its associated label.

Although the two points are equal, they have two different hashCodes!

Rule: hashCode

Rule: Whenever you write your own class and you want to use instances of the class in a `HashSet` or `HashMap`, you **must** write your own `hashCode` method for you class.

Example: Point class

```
public class Point {
  private int x;
  private int y;
  private int greyScale; // for internal use only

  public boolean equals(Object obj) {
     if (obj instanceof Point) {
        Point other = (Point) obj;
        return this.x == other.x
           &&  this.y == other.y;
     }
     return false;
  }
  public int hashCode() {  ????
```

Hash function properties

Desired properties:

1. The hash function should be fast to compute: O(1)
2. Limited number of collisions:
   a) Given two elements, the probability that they hash to the same index is low. (We would like unequal objects have unequal hash codes.)
   b) When many elements are added to the table, they should appear “evenly” distributed.
3. To be **valid**, it **must** hash two objects of equal value to the same index.

For `Point` class which hashCodes are valid? Good?

```
public int hashCode() { return x; }
```

```
public int hashCode() { return x*y; }
```

```
public int hashCode() { return x+y; }
```

Valid? Good?

```
public int hashCode() { return 47; }
```

```
public int hashCode() {
                return x*Math.random(); }
```

```
public int hashCode() { return x*1000 + y;}
```

Valid? Good?

```
public int hashCode(){
          return (x + " " + y).hashCode; }


public int hashCode() {
      return x*10000 + y*100 + greyScale; }
```

HashCode advice

- **Rule:** When writing a `hashCode` method, **do not** use fields that are not included the `equals` method.

- **Rule-of-thumb**: **Include all fields and their subparts** that are used in the `equals` method in its hash code computation to minimize collisions.

If `x.equals(y)`, must
`hashCode(x) == hashCode(y)`?

If `hashCode(x) == hashCode(y)` must
`x.equals(y)`?

Worst-case runtime complexity of Map/Set implementations

implementation	contains	add/remove	restriction
Unsorted array	O(n)	O(n)	
Unsorted linked list	O(n)	O(n)	
Sorted array	O(log n)	O(n)	Comparable
Sorted linked list	O(n)	O(n)	Comparable
Binary tree	O(n)	O(n)	
Binary search tree	O(n)	O(n)	Comparable
Balance BST	O(log n)	O(log n)	Comparable
Hash table – expected	O(1)	O(1)	Need valid hash function
Hash table – worst-case one bucket	O(n)	O(n)	

HashSet vs TreeSet

Advantages of HashSet (HashMap)

Near constant time: expected O(1)

* Don't have to be Comparable

Advantages of TreeSet (TreeMap)

More operations than HashSet: fast min, max, range

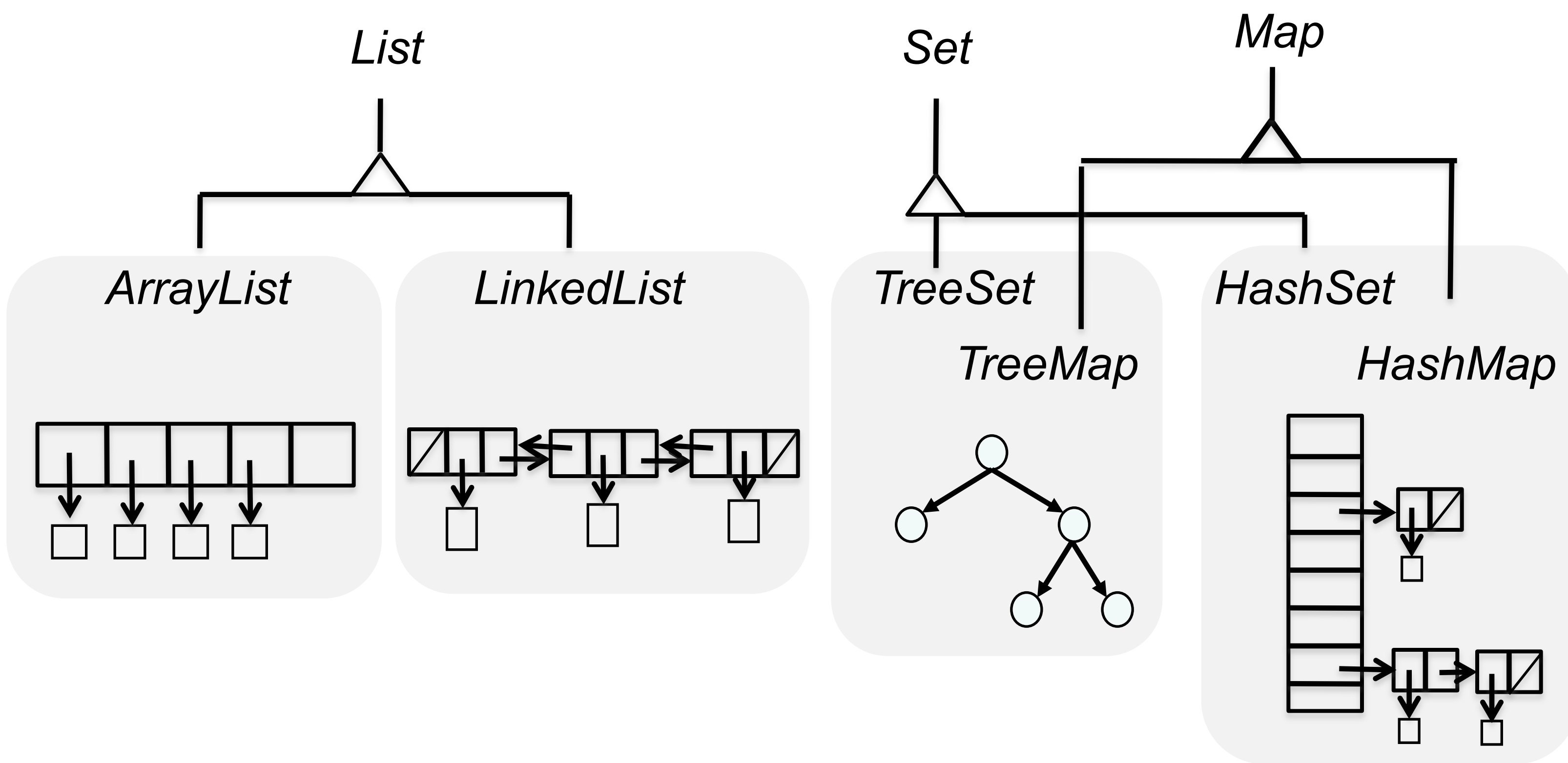
* TreeSet iterator gives values in natural order

Don't need to write a hash function

(No need to tune trade off between space and time)

But worst-case runtime: O(log n)

Java collections in one slide



D. Feinberg

Priority Queues

Binary Heaps

ADT vs Data structures

Abstract Data Types: List, Set, Map, Stack, Queue (In Java, we typically define an interface for ADTs.)

Data Structures: array, dynamic array, sorted dynamic array, linked list, doubly-linked list, binary search tree, hash table, etc.

(Not always a clear distinction, though)

Priority Queue ADT

Priority Queue has the following operations:

isEmpty
add (with priority)
remove (highest priority)
peek (at highest priority)

```
public interface PriorityQueue {
    boolean isEmpty();
    void add (Comparable obj);
    Comparable removeMin();
    Comparable peekMin();
}
```

Towards a PQ implementation

What data structures have we seen that has an O(log n) worst-case runtime to add?

- O(log n) often suggests a balanced binary tree (not necessarily a search tree).

If we can peek at the highest priority in O(1) runtime in the worst case, where must be the highest priority item?

Towards a PQ implementation

Where would you expect to find the 2nd highest priority item?

Does it matter in which subtree, left or right, that the 2nd highest item is?

- No. The only requirement is that it should be the highest priority item in its subtree.

Introducing Binary Heaps

Binary heaps are a data structure with two properties:

1. Shape
2. Order

(Aside: When a program runs, memory is divided into two parts:

stack – stores values of parameters and local variables.

heap – stores objects and arrays.

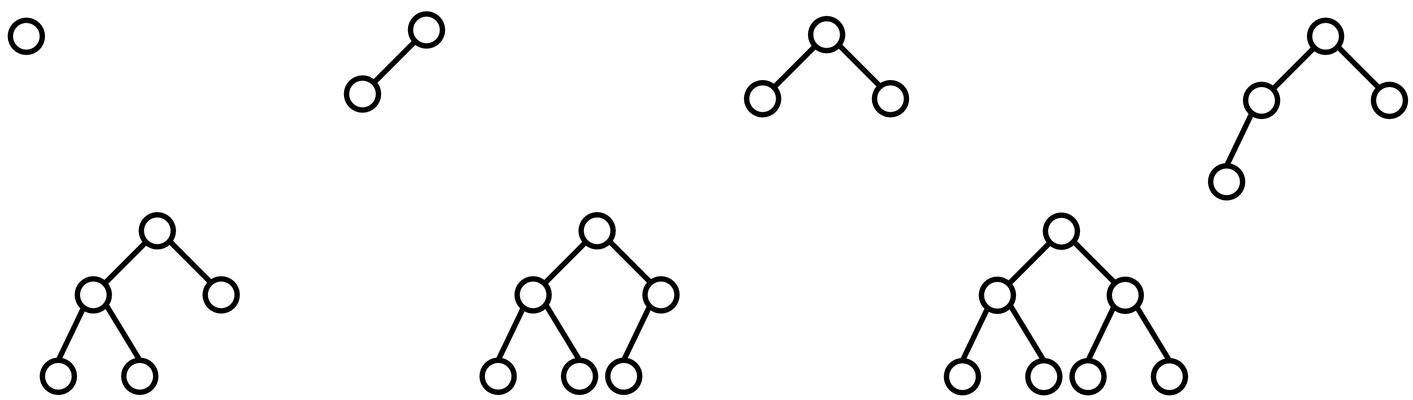
Heap data structures have no relation to the memory heap.)

The binary heap Shape property

A binary heap is a **complete binary tree** –

- all levels are completely filled, except the bottom level which is filled from left to right.

The 7 smallest heap shapes:



The binary heap Order Property

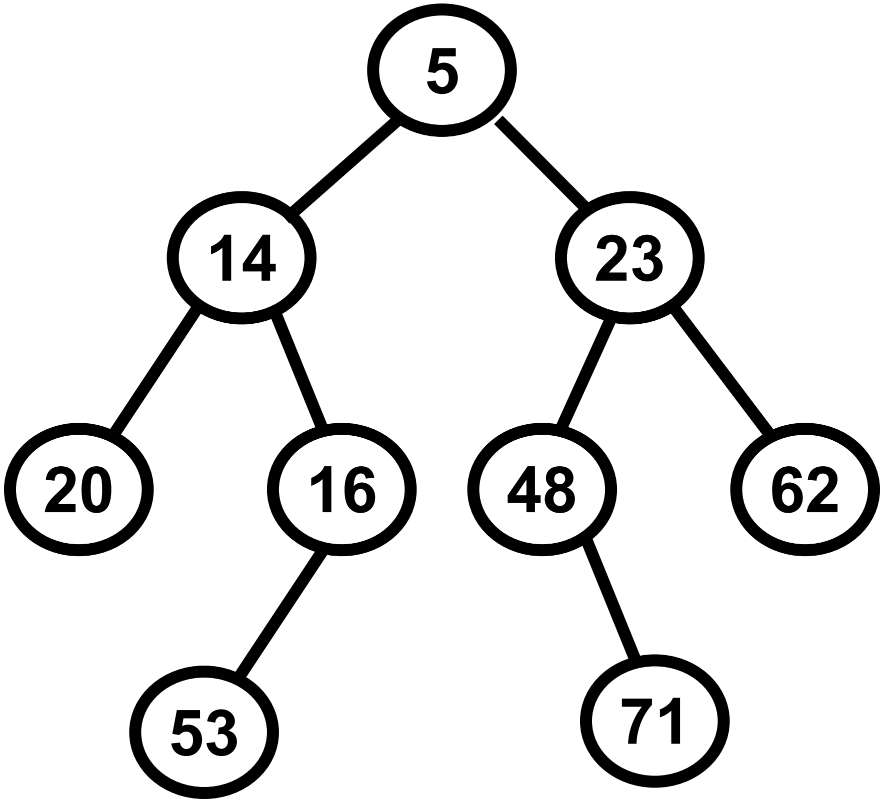
Min-Heap: parent <= children for all nodes

(the highest priority is the minimum)

Max-Heap: parent >= children for all nodes

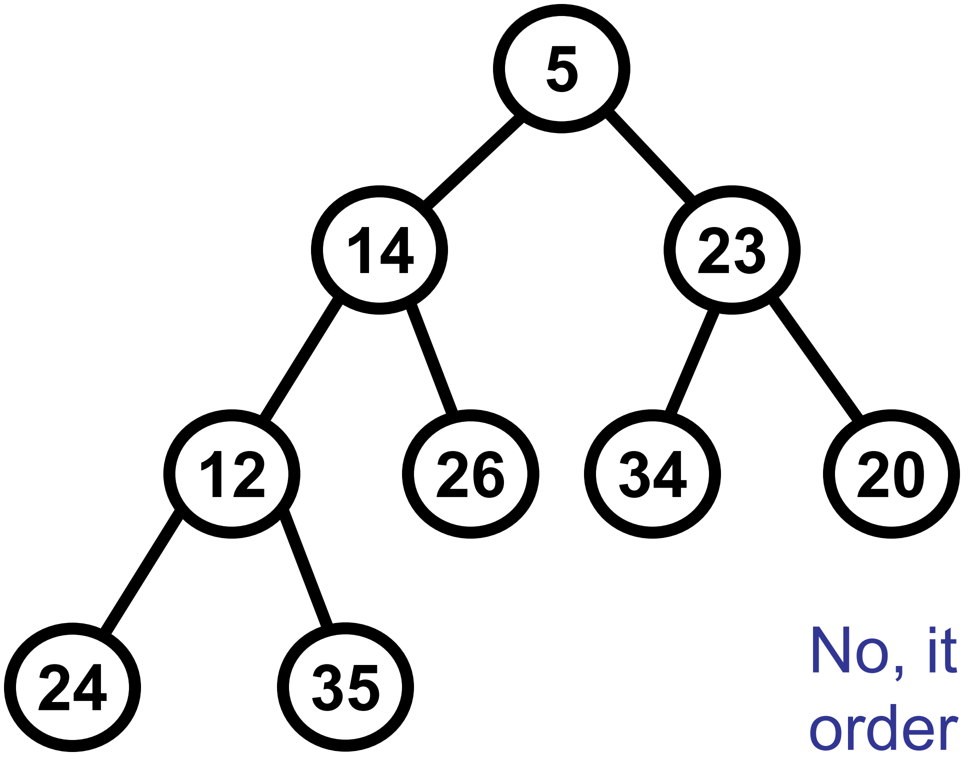
(highest priority is the maximum)

Is it a min-heap?



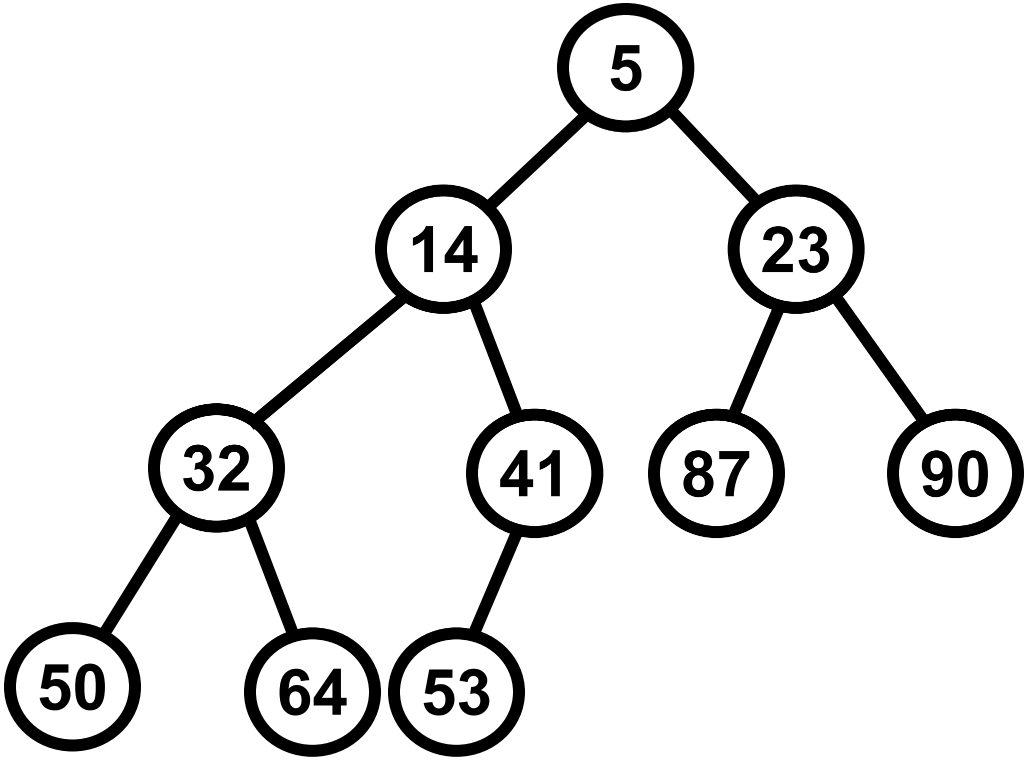
No, it violates the shape property.

Is it a min-heap?



No, it violates the order property.

Is it a min-heap?



Yes.

Possible Heaps

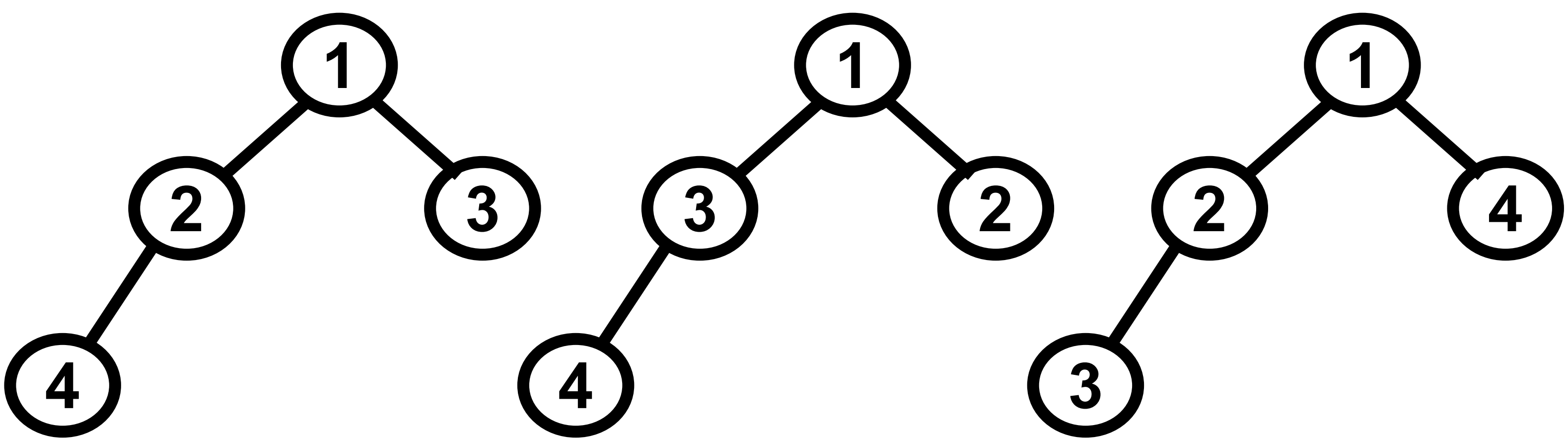
What are all possible min-heaps on elements 1, 2, 3, 4?

What shape can the tree have?

What value(s) can the root have?

What must be a child of the root?

Can 4 be a child of the root?



Exercise: What are all the min-heaps on 1,2,3,4,5?

Add an element to heap

Step 1: Maintain the **shape** property first.

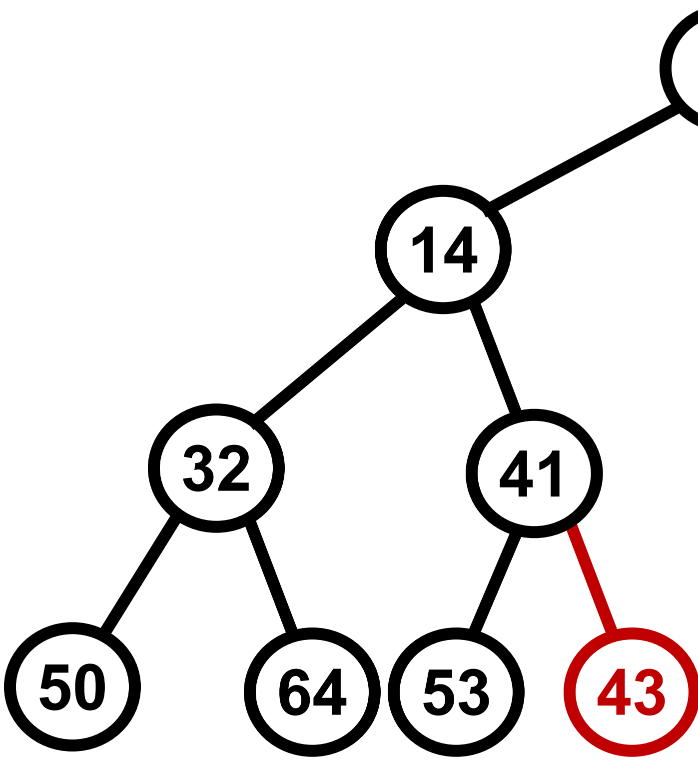
Where must the new element go to keep the tree complete? Ignore that it might violate the order property.

Step 2: Then restore the **order** property.

To where must we move the new element?

Add to a min-heap

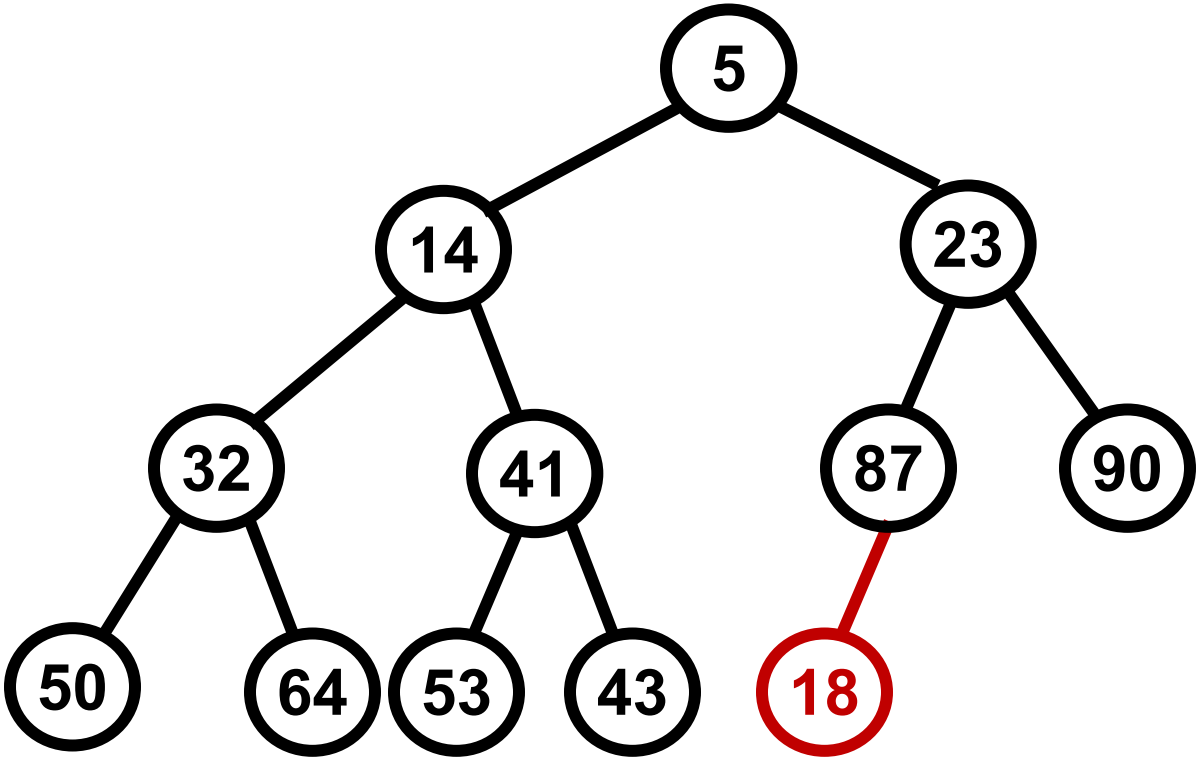
Add 43



1. Shape property
2. Order property

Add to a min-heap

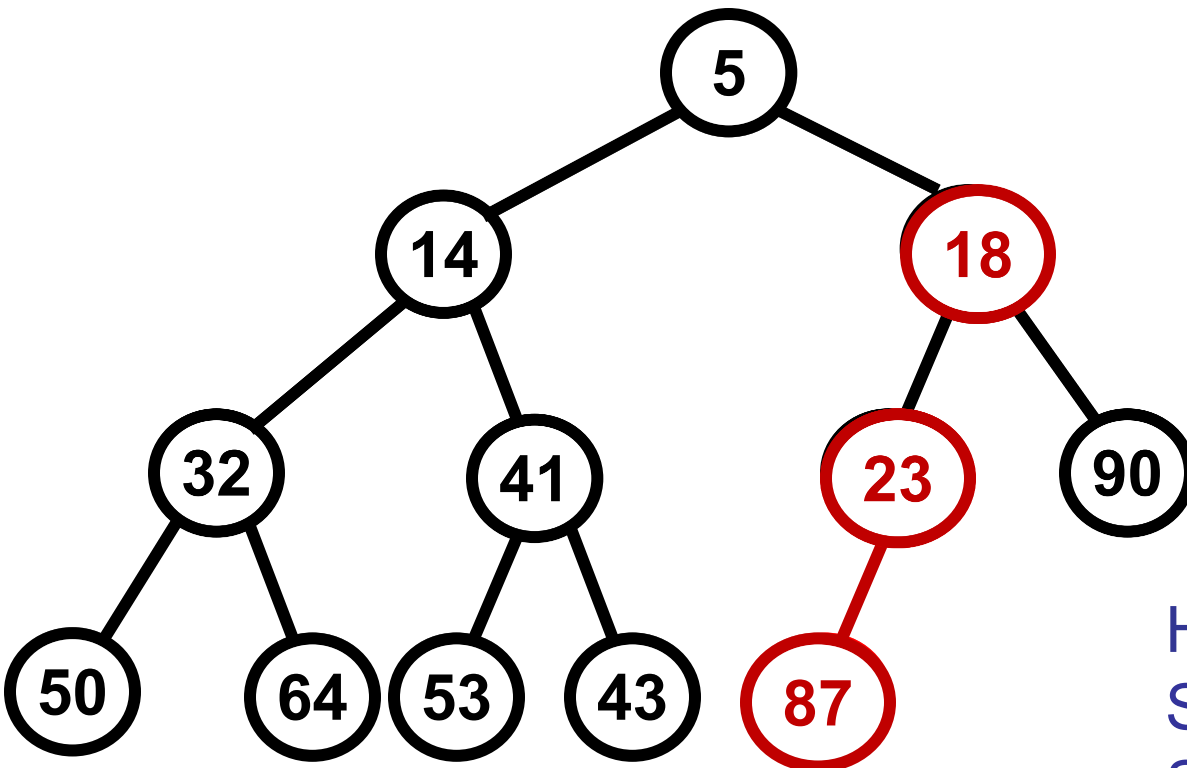
add 18



1. Add leaf
2. Heapify up:
   (see next slide)

Add to a min-heap

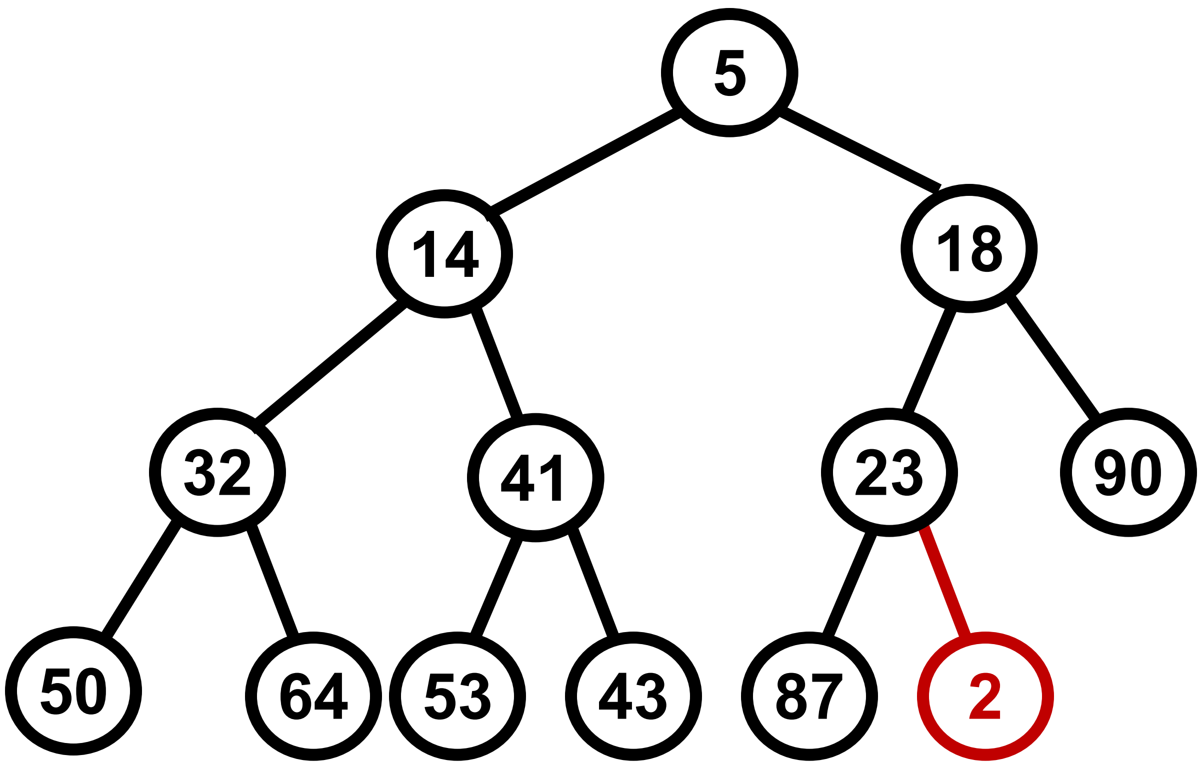
18 added



Heapified up:
Swapped 18 & 87
Swapped 18 & 23

Add to a min-heap

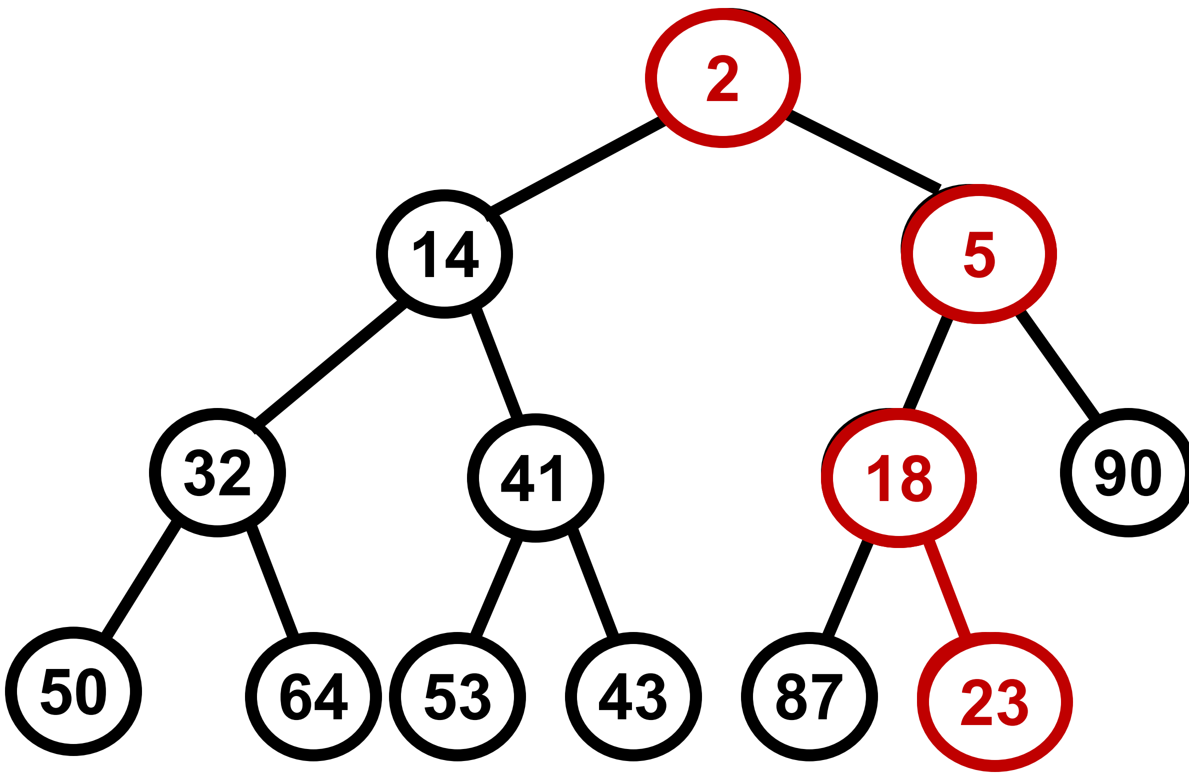
add 2



1. Add leaf

2. Heapify up:
(see next slide)

Add to a min-heap
2 added



Heapified up:
Swapped 2 & 23
Swapped 2 & 18
Swapped 2 & 5

Remove the minimum

Step 1: Maintain the **shape** property first

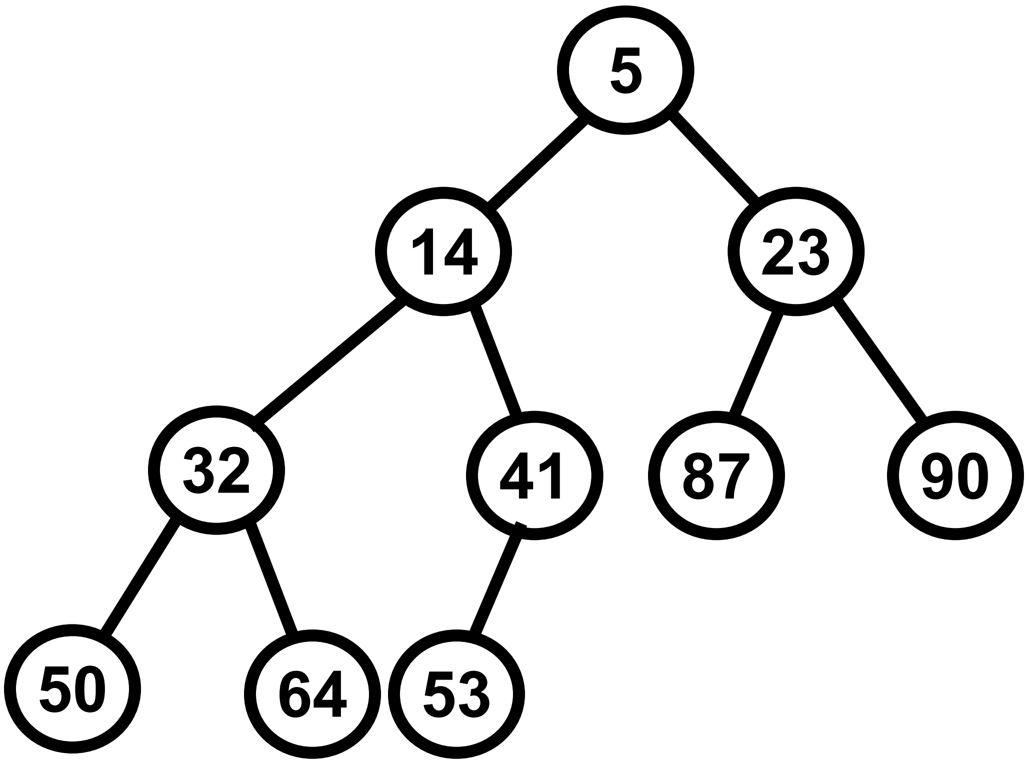
What element should we use to replace the root we just removed?

Step 2: Then restore the **order** property

To where must we move the new root?

Removing from a min-heap

Remove min (5)



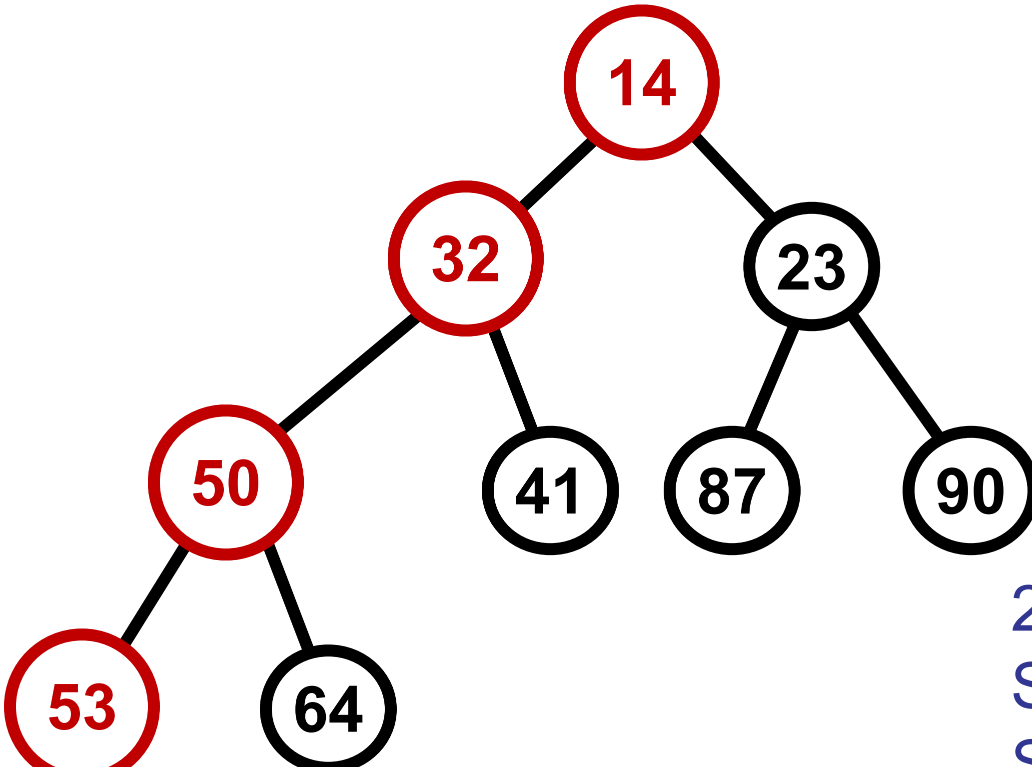
returnValue 5

1. Shape property:
   Put 53 at the root

(continued)

Removing from a min-heap

Remove min (5)

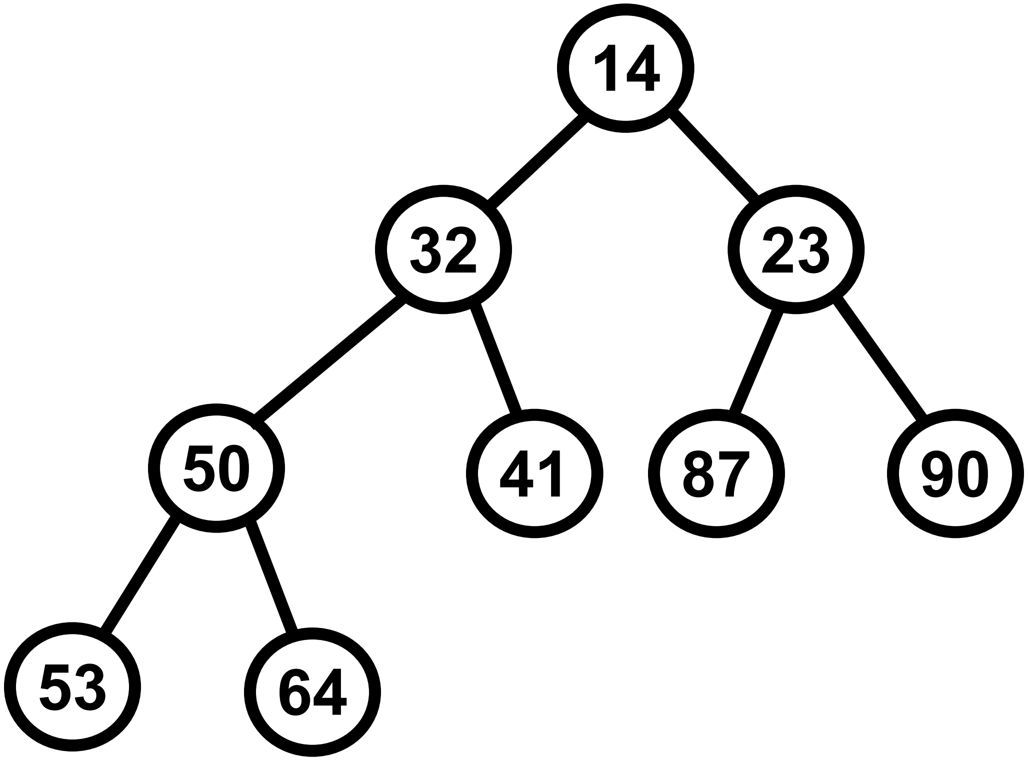


returnValue 5

2. heapify down:
Swap 53 & 14 (not 23)
Swap 53 & 32 (not 41)
Swap 53 & 50 (not 64)

Removing from a min-heap

Remove min (14)



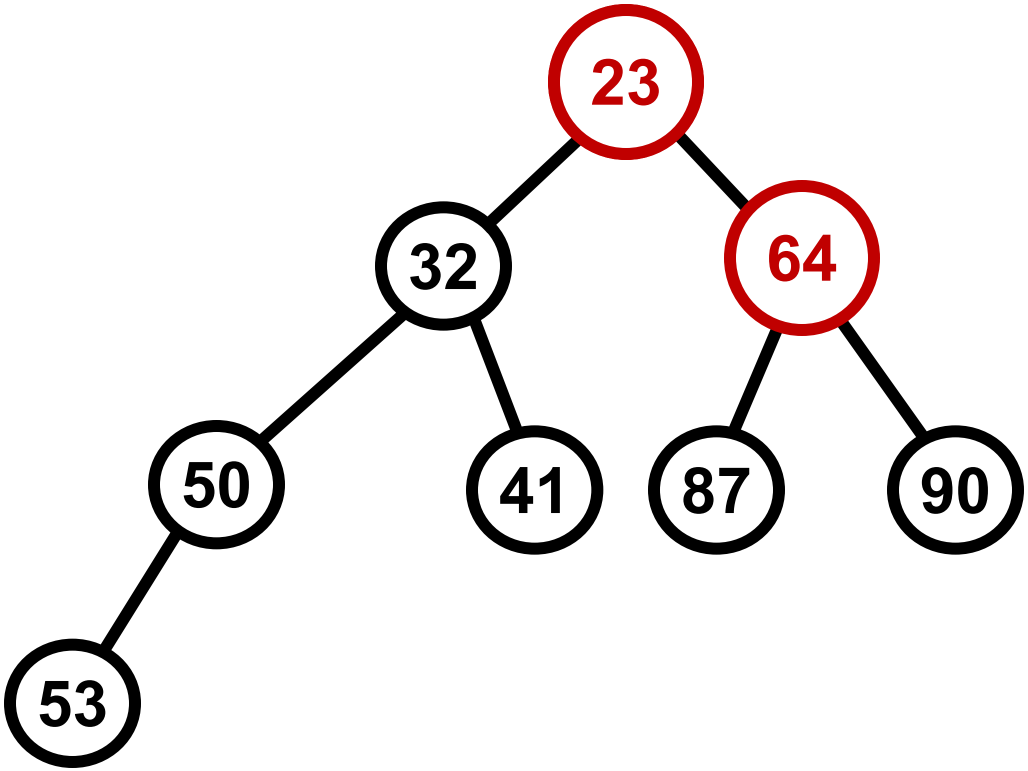
returnValue 14

1. Put 64 at the root

(continued)

Removing from a min-heap

Remove min (14)



returnValue 14

1. Put 64 at the root
2. heapify down:
Swap 64 & 23 (not 32)

Exercise

- Build a min-heap with 12, 6, 4, 8, 10, 9.
- Repeatedly remove the minimum until empty.

If the data structure is a binary tree

Add Problem:

How can I find where to put the new element?

How do we find the parent of a child?

Remove Problem:

How can I find the element to put at the root?

Towards a data structure

Suppose we number the nodes of the binary heap as follows. Do you see a relationship between a node and its children? A node and its parent?

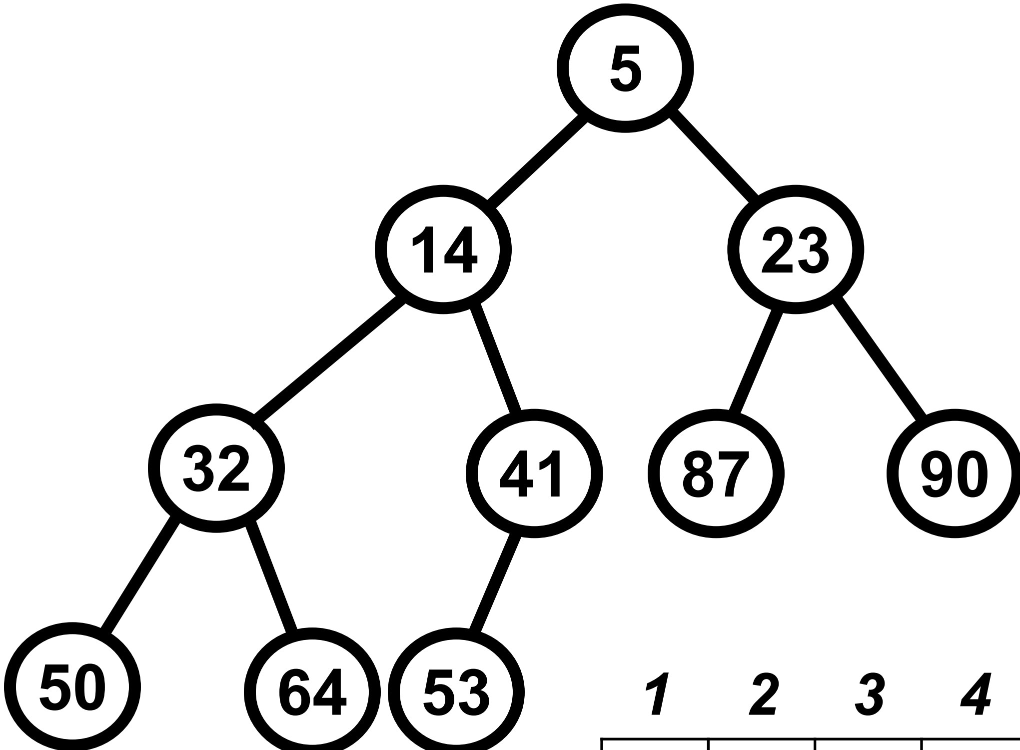
```
              1
         /         \
      2                3
    /   \            /   \
   4     5         6      7
  / \   / \       / \    / \
 8   9 10  11   12  13  14  15
```

For a node numbered i

- left child is 2*i
- right child is 2*i + 1
- parent is i / 2 (integer division)

Using this indexing we can store a binary tree in an array (starting at index 1).

ArrayList implementation



For a node i
- left child is 2*i
- right child is 2*i + 1
- parent is i / 2

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
5	**14**	**23**	**32**	**41**	**87**	**90**	**50**	**64**	**53**

Binary heaps runtime complexity

What is the height of the binary heap? O(log n) – ALWAYS

Runtime (min-heap)**:**

isEmpty: O(1)

peekMin: O(1)

add: best: O(1) – sometimes add a large element

expected: O(1) – most nodes are at bottom 2 layers

worst: O(log n) – sometimes move up to root

removeMin: O(log n) – always move a large element from the root down (usually to bottom 2 layers)

Heap Sort

If we add n values to an empty min-heap and then we remove all the values from a heap, in what order will they be removed?

Smallest to largest. We just invented Heap Sort!

Heap Sort Runtime:

1. Build the heap: n * O(log n)
2. Repeatedly remove the min: n * O(log n)

Total: O(n log n): best, expected, and worst case

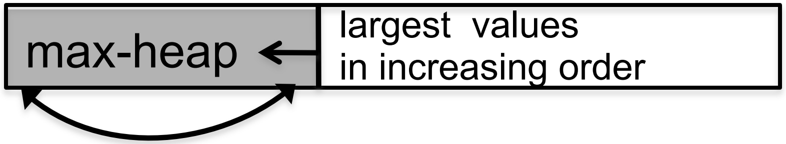
What other sort has the same worst-case runtime? Merge sort

What is the disadvantage of merge sort? Not in place

Heap Sort (in place)

1. Build a max-heap by adding each successive element in the array to the heap.

2. Remove the maximum and put it at the last index, remove the next maximum and put it at 2nd to last index, and so on. In particular, repeatedly swap the root with last element in the heap and heapify down the new root to restore the heap one size smaller.

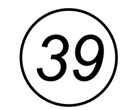


parent of j = (j-1)/2

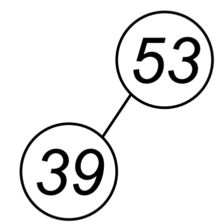
1. Building the max-heap

ADD NEXT VALUE TO HEAP AND FIX HEAP

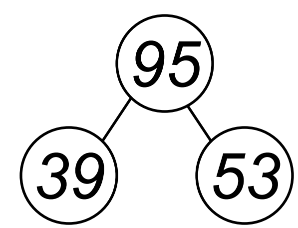
0	1	2	3	4	5	6
39	53	95	72	61	48	83



0	1	2	3	4	5	6
53	39	95	72	61	48	83



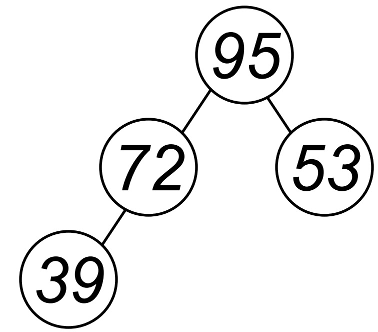
0	1	2	3	4	5	6
95	39	53	72	61	48	83



parent of j = (j-1)/2

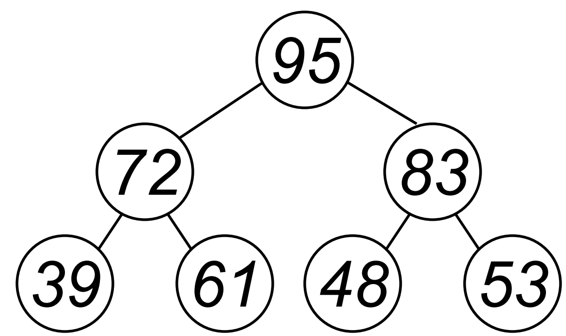
1. Building the max-heap (cont'd)

0	1	2	3	4	5	6
95	**72**	**53**	**39**	**61**	**48**	**83**



CONTINUE UNTIL THE HEAP IS COMPLETED...

0	1	2	3	4	5	6
95	**72**	**83**	**39**	**61**	**48**	**53**



children of j = (j+1)*2-1, (j+1)*2

2. Sorting from the heap

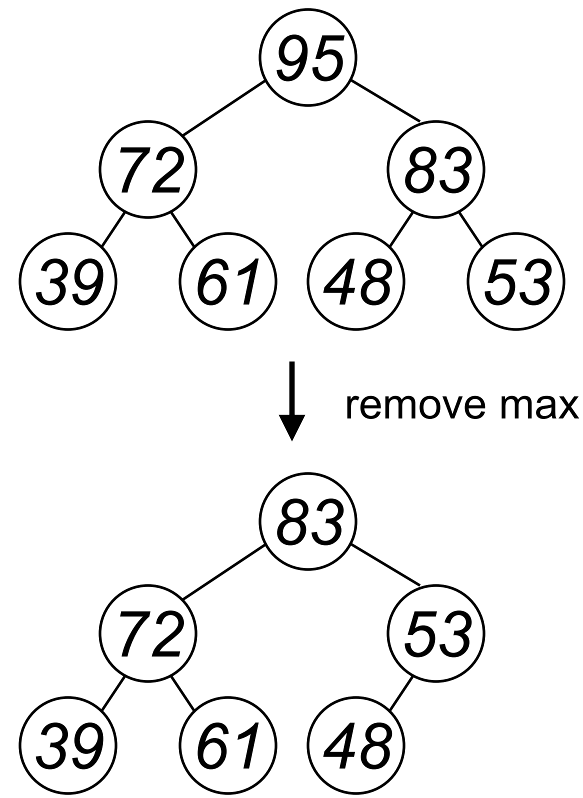
0	1	2	3	4	5	6
95	72	83	39	61	48	53

SWAP THE MAX OF THE HEAP WITH THE LAST VALUE OF THE HEAP:

0	1	2	3	4	5	6
53	72	83	39	61	48	95

FIX THE HEAP (NOT INCLUDING MAX):

0	1	2	3	4	5	6
83	72	53	39	61	48	95



children of j = (j+1)*2-1, (j+1)*2

2. Sorting from the heap (cont'd)

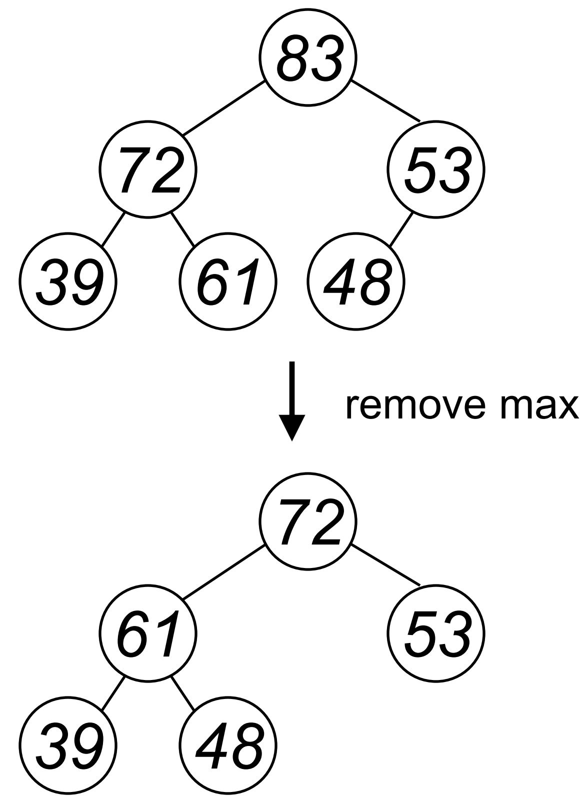
0	*1*	*2*	*3*	*4*	*5*	*6*
83	**72**	**53**	**39**	**61**	**48**	**95**

SWAP THE MAX OF THE HEAP
WITH THE LAST VALUE OF THE HEAP:

0	*1*	*2*	*3*	*4*	*5*	*6*
48	**72**	**53**	**39**	**61**	**83**	**95**

FIX THE HEAP (NOT INCLUDING MAX):

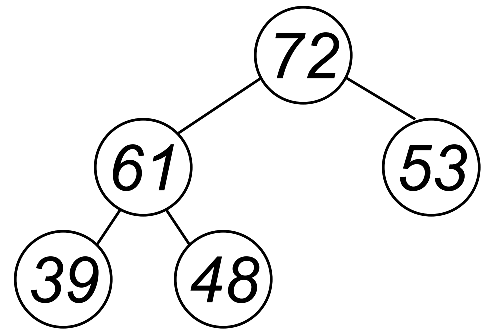
0	*1*	*2*	*3*	*4*	*5*	*6*
72	**61**	**53**	**39**	**48**	**83**	**95**



children of j = (j+1)*2-1, (j+1)*2

2. Sorting from the heap (cont'd)

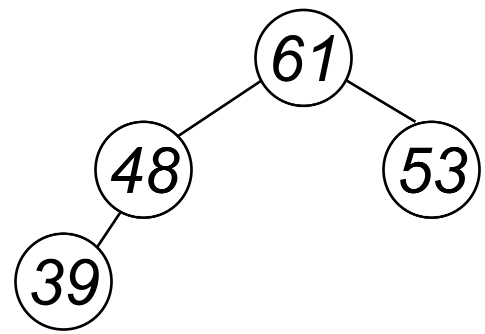
0	1	2	3	4	5	6
72	61	53	39	48	83	95



SWAP THE MAX OF THE HEAP
WITH THE LAST VALUE OF THE HEAP:

0	1	2	3	4	5	6
48	61	53	39	72	83	95

remove max



FIX THE HEAP (NOT INCLUDING THAT MAX):

0	1	2	3	4	5	6
61	48	53	39	72	83	95

REPEAT UNTIL THE HEAP HAS 1 NODE LEFT