PROGRAMMING AND DATA STRUCTURES

BINARY TREES (HEAP)

HOURIA OUDGHIRI

SPRING 2023

OUTLINE

- Characteristics of the Heap
- Operations on the Heap
- Implementation of the Heap class

STUDENT LEARNING OUTCOMES

At the end of this chapter, you should be able to:

- Describe the properties of the Heap
- Trace operations on the Heap
- Implement the Heap generic data structure
- Use the Heap data structure
- Evaluate the complexity of the operations on the Heap

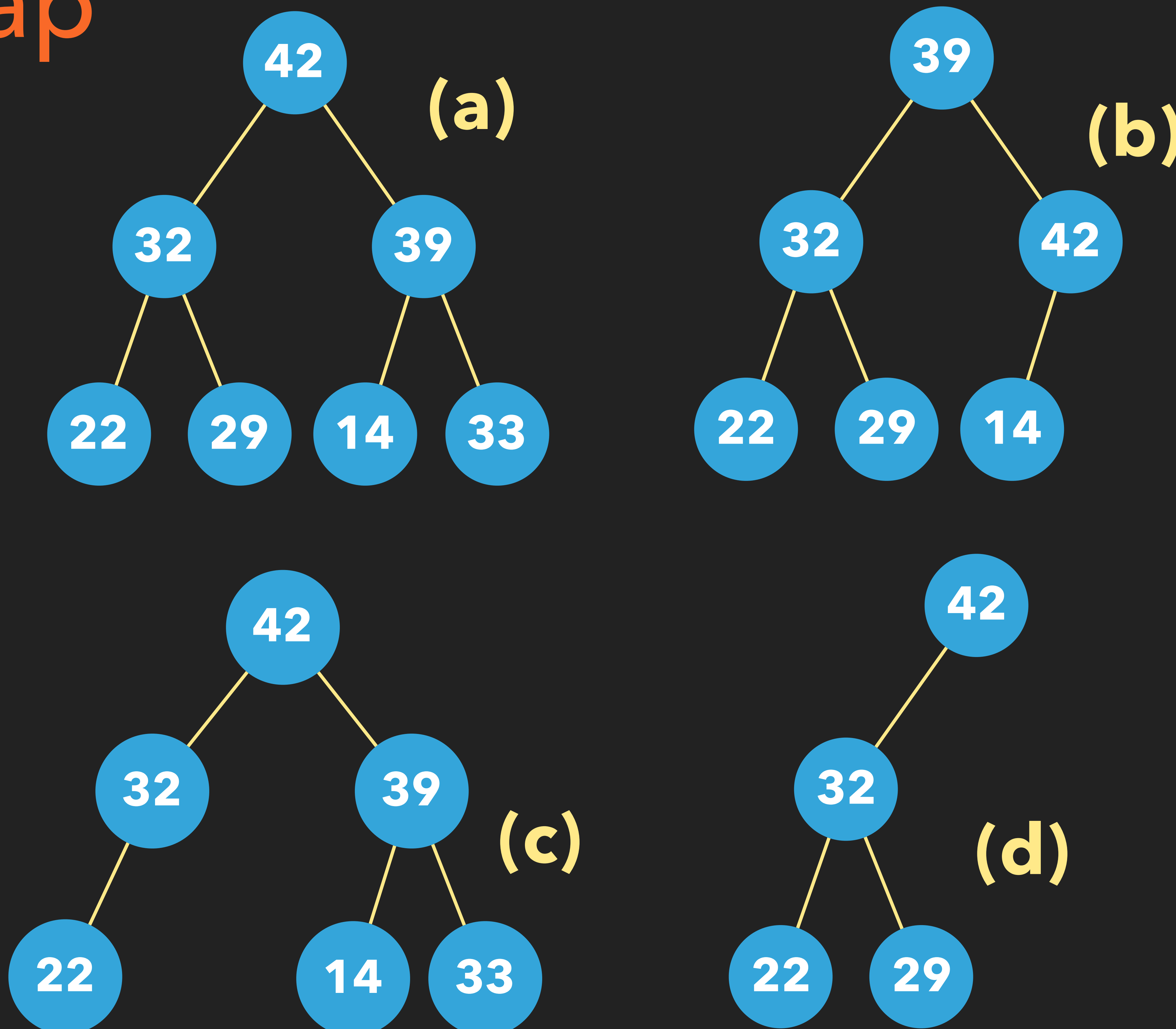
Heap

- Special binary tree used to order data (not to retrieve data)
- Used for efficient sorting (heap sort)
- Used to implement the priority queue

Heap

- Properties of the heap
  - Property 1: Complete binary tree
    All the levels are filled except the last level
    All leaves on the last level are placed leftmost
  - Property 2: every node is greater than or equal to any of its children (Max Heap) [Min Heap: less than or equal]

Heap

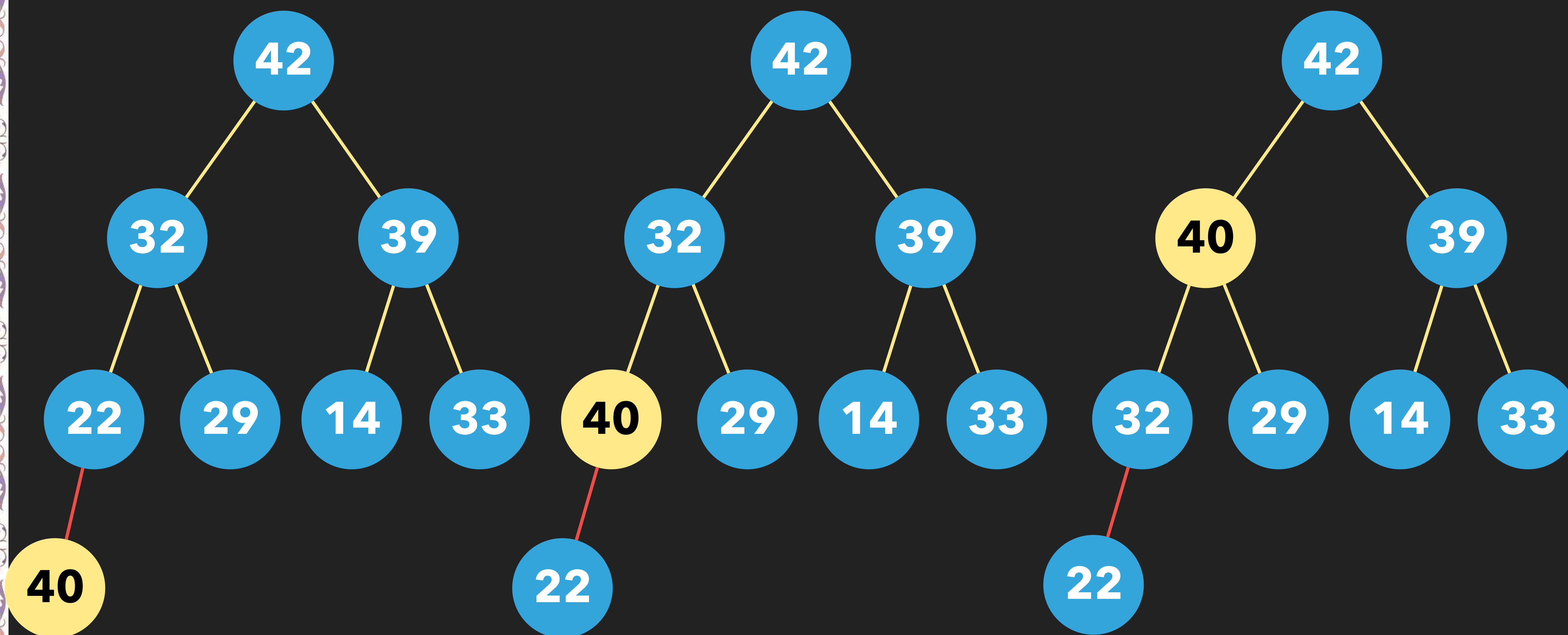

42
(a)
32
39
22
29
14
33
39
(b)
32
42
22
29
14
42
32
39
(c)
22
14
33
42
32
(d)
22
29

Heap

- Two main operations on the Heap
  - Adding a new node while keeping the heap properties
  - Removing a node while keeping the heap properties

Heap (add)

- Adding a new node to the heap (40)



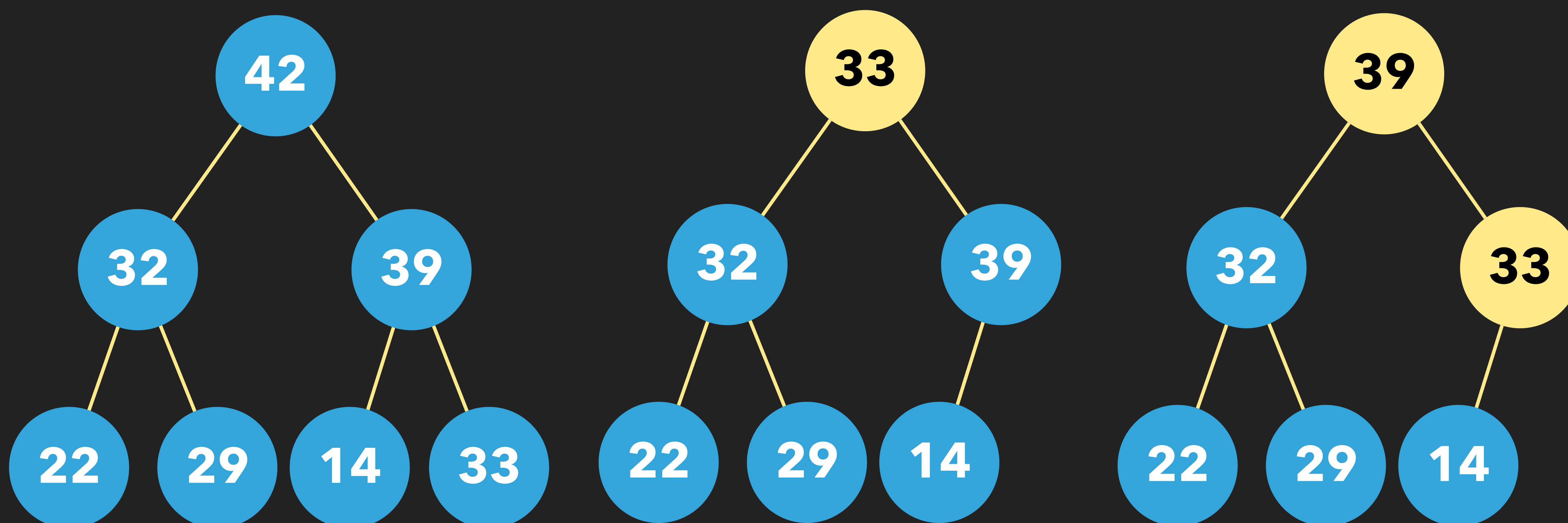
Heap (add)

- Adding a new node to the heap

```
Algorithm add
  Add the new node at the end of the heap
  Current node = added node
  While (current node > its parent)
    Swap current node with its parent
    Current node becomes the parent
End
```

Heap (remove)

- Removing a node from the heap (42)
- Always the root



Heap (remove)

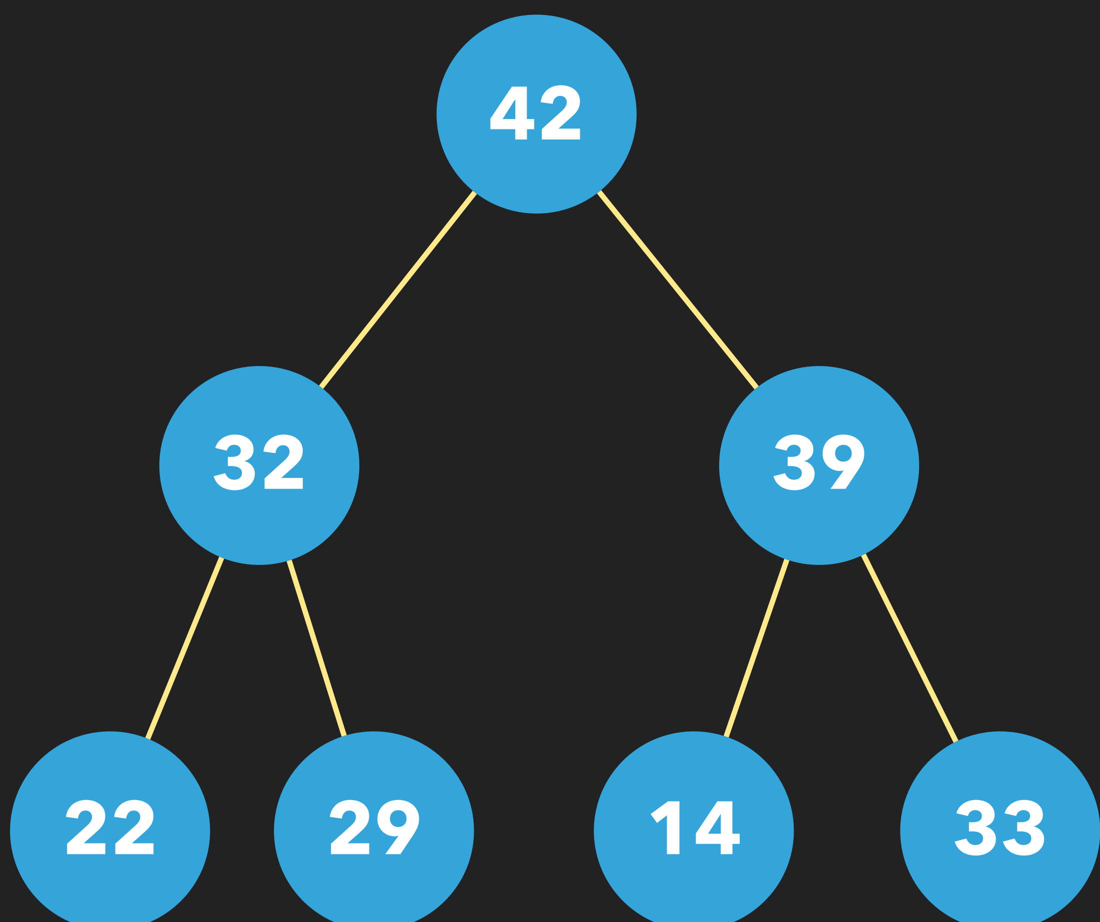
- Removing a node from the heap (root)

```
Algorithm remove

  Copy the value of the last node to the root

  Current node = root

  While (current node < its children)

    Swap current node with the largest
    of its children

    Current node becomes the largest child

End
```

Heap implementation

- Heap implementation
  - ArrayList to store the heap nodes
  - Easy access to children and parent

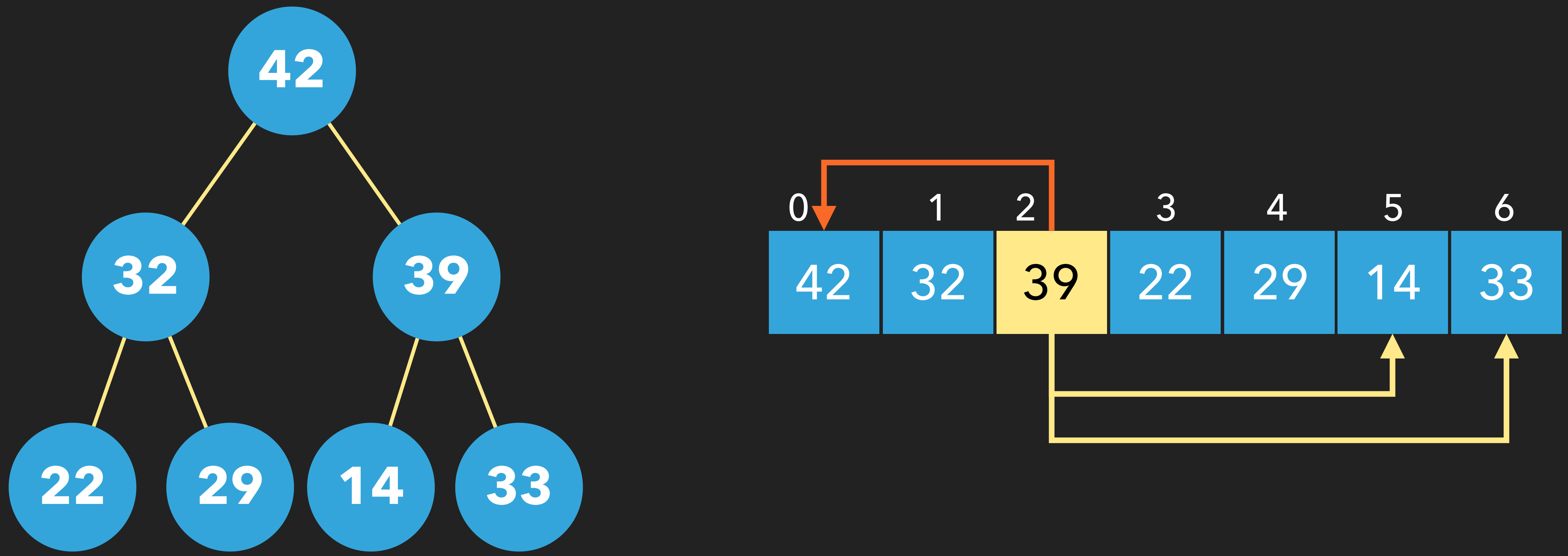
Heap implementation



ArrayList with the nodes of the heap

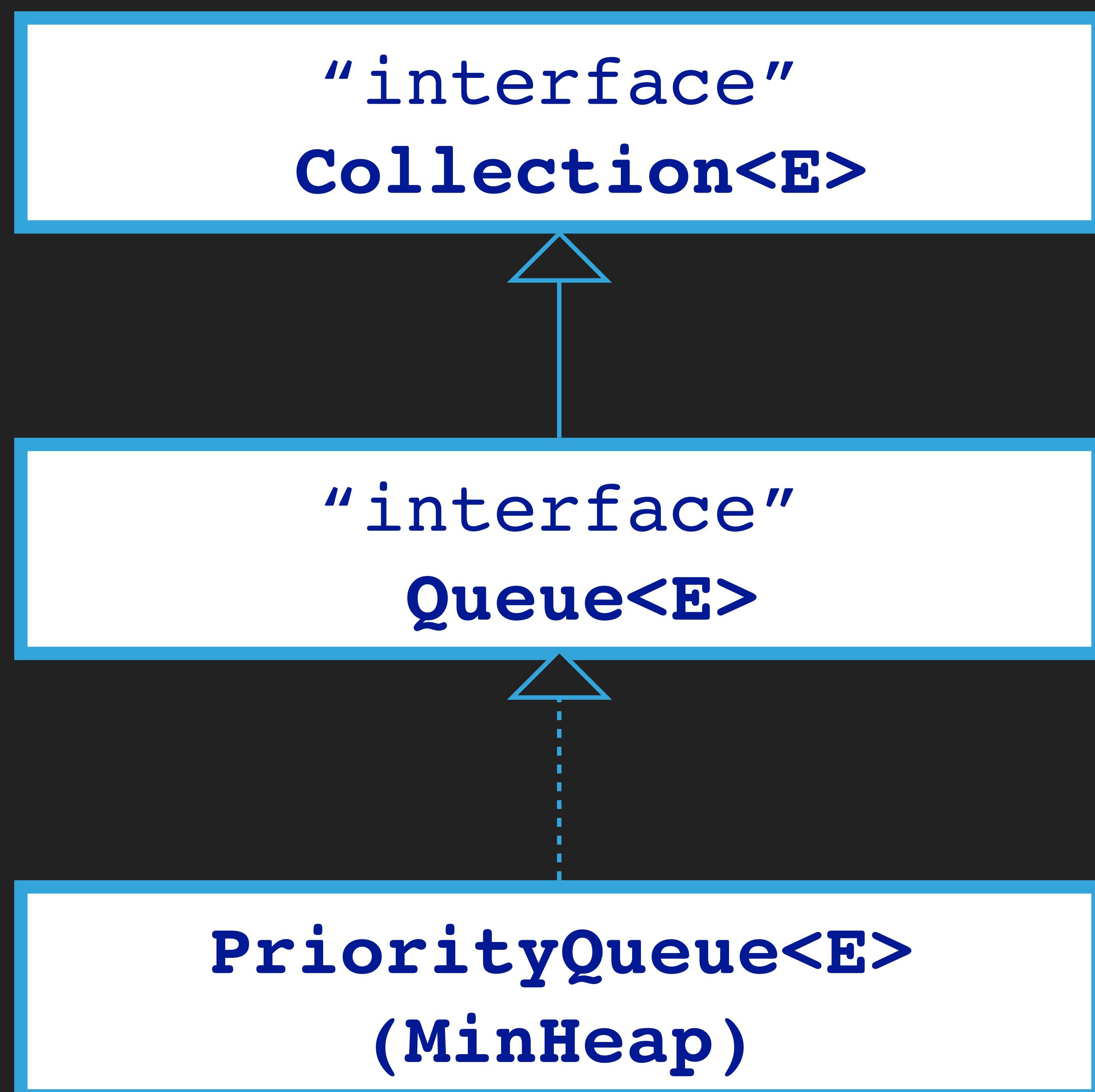
0	1	2	3	4	5	6
42	32	39	22	29	14	33

Heap implementation




```
IndexOf(Parent) = (IndexOf(current) - 1) / 2

IndexOf(Left child) =  2 * IndexOf(current) + 1
IndexOf(Right child) = 2 * IndexOf(current) + 2
```

Heap implementation



Heap implementation

Heap<E extends Comparable<E>>
-list: ArrayList<E>
+Heap() +add(E): void +remove(): E +contains(E): boolean +size(): int +isEmpty(): boolean +clear(): void +toString(): String

Heap implementation

Heap.java

```
public class Heap<E extends Comparable<E>> {
    private ArrayList<E> list;
    public Heap(){
        list = new ArrayList<>();
    }
    public int size(){
        return list.size();
    }
    public boolean isEmpty(){
        return list.isEmpty();
    }
    public void clear(){
        list.clear();
    }
    public String toString(){
        return list.toString();
    }
```

Heap implementation

Heap.java

```
public boolean contains(E value) {
        for(int i=0; i<list.size(); i++) {
            if(list.get(i).equals(value))
                return true;
        }
        return false;
}
```

Heap implementation

Heap.java

```
public void add(E value) {
        list.add(value);
        int currentIndex = list.size()-1;
        while(currentIndex > 0) {
            int parentIndex = (currentIndex-1)/2;
            E current = list.get(currentIndex);
            E parent = list.get(parentIndex);
            if(current.compareTo(parent) > 0) {
                list.set(currentIndex, parent);
                list.set(parentIndex, current);
            }
            else
                break;
            currentIndex = parentIndex;
        }
    }
```

Heap implementation

Heap.java

```
public E remove() {
        if(list.size() == 0) return null;
        E removedItem = list.get(0);
        list.set(0, list.get(list.size()-1));
        list.remove(list.size()-1);
        int currentIndex = 0;
        while (currentIndex < list.size()) {
            int left = 2 * currentIndex + 1;
            int right = 2 * currentIndex + 2;
            if (left >= list.size())
                break;
            int maxIndex = left;
            E max = list.get(maxIndex);
            if (right < list.size())
            if(max.compareTo(list.get(right)) < 0)
                maxIndex = right;
            E current = list.get(currentIndex);
            max = list.get(maxIndex);
            if(current.compareTo(max) < 0){
                list.set(maxIndex, current);
                list.set(currentIndex, max);
                currentIndex = maxIndex;
            }
            else
                break;
        }
        return removedItem;
    }
```

Heap implementation

Test.java

```
public class Test {
    public static void main(String[] args) {
        Heap<String> heap = new Heap<>();
        heap.add("Kiwi");
        heap.add("Strawberry");
        heap.add("Apple");
        heap.add("Banana");
        heap.add("Orange");
        heap.add("Lemon");
        heap.add("Watermelon");
        System.out.println("Heap: " + heap.toString());
        System.out.println("Removed: " + heap.remove());
        System.out.println("Heap: " + heap.toString());
        System.out.println("Heap contains Pear?: " +
                                    heap.contains("Pear"));
    }
}
```

Heap implementation

- Performance of the Heap operations

Method	Complexity
`Heap()`	**O(1)**
`size()`	**O(1)**
`clear()`	**O(1)**
`isEmpty()`	**O(1)**
`add(E)`	**O(log n)**
`remove(E)`	**O(log n)**
`contains(E)`	**O(n)**
`toString()`	**O(n)**

Summary

- Heap - Special binary tree
- Operations: Add and Remove mainly
- Implementation - Using an ArrayList
- Performance of the operations on the Heap (logarithmic complexity for add and remove)
- Heap is a balanced binary tree always
  (height = `log`(number of nodes))