1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

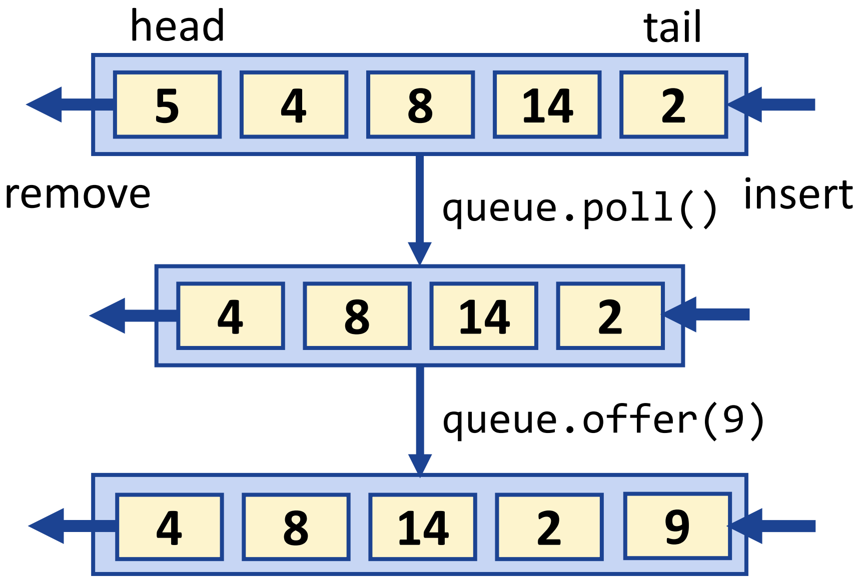
Lecture 35: Queues and sets

References and learning objectives

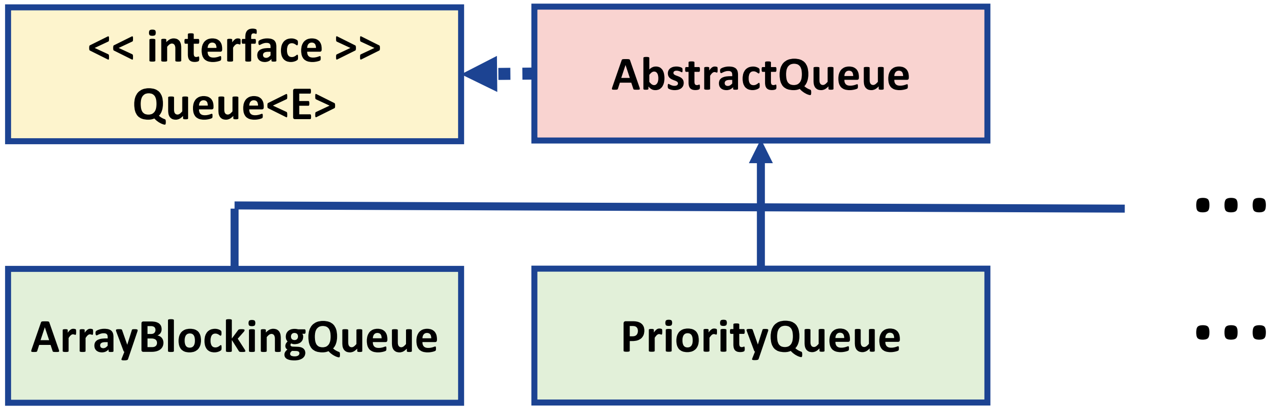
- Today’s sessions are mostly based on:
  - Evans & Flanagan, **Java in a Nutshell**, *7th edition*, 2018. O'Reilly Media.
  - Deitel, **Java How to Program**, 2018, Chapter 16.
- After today’s session, you should be able to:
  - Implement basic data structures and algorithms in Java applications using queues, sets, and maps defined in Java collection framework
  - Implement custom comparators for search and sorting algorithms using Java collection framework

Queues

- Usually, elements in queues are ordered in FIFO (*First In, First Out*) manner
- Insert elements by calling **`queue.offer(`**`data`**`)`**
- Access top element by calling **`queue.peek()`**
- Remove top element by calling **`queue.poll()`**



Different queues



- First-In, First-Out ordered queue bounded to the maximum capacity

- Unbounded queue using natural ordering of elements

- `ConcurrentLinkedQueue`
- `DelayQueue`
- etc ...

PriorityQueue example

```java
import java.util.*;
public class PriorityQueueExample {
  public static void main(String[] args) {
    PriorityQueue<Double> queue = new PriorityQueue<>();
    queue.offer(9.2);
    queue.offer(5.1);
    queue.offer(12.7);
    queue.offer(0.8);
    while(queue.size() > 0) {
      System.out.printf("%.1f\n", queue.peek());
      queue.poll();
    }
  }
}
```

Note ordering of elements

```
$ java PriorityQueueExample
0.8
5.1
9.2
12.7
$
```

UNIVERSITY OF ABERDEEN

ArrayBlockingQueue example

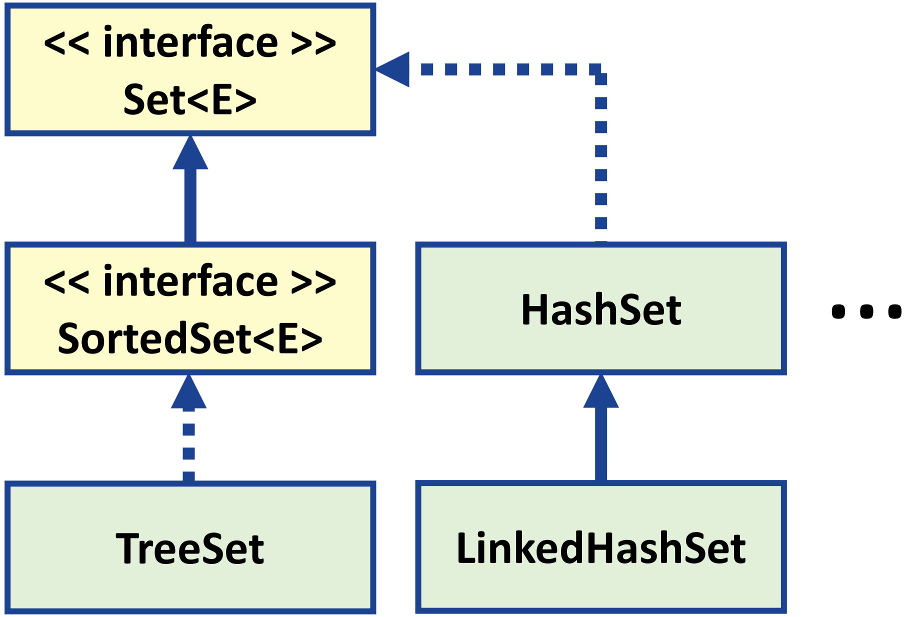
```java
import java.util.concurrent.*;
import java.util.*;
public class ArrayBlockingQueueExample {
  public static void main(String[] args) {
    Random random = new Random();
    ArrayBlockingQueue<Double> queue = new ArrayBlockingQueue<>(5);
    while(queue.offer(random.nextDouble())) {}
    while(queue.size() > 0) {
      System.out.printf("%.2f\n", queue.peek());
      queue.poll();
    }
  }
}
```

Maximum capacity set to 5

```
$ java ArrayBlockingQueueExample
0.94
0.98
0.54
0.52
0.62
$
```

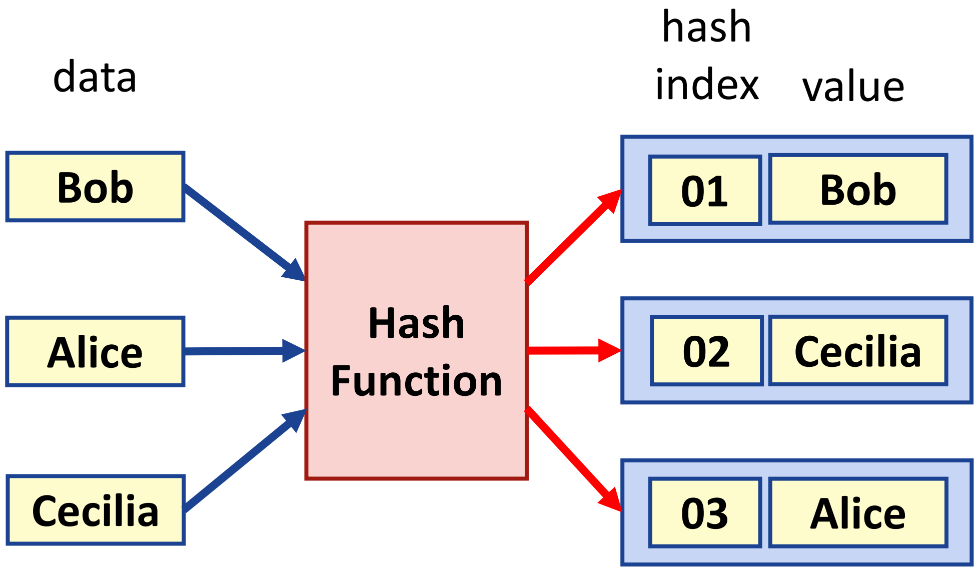
Sets

- `Set` is a collection that contains no duplicate elements
- When iterating over a `Set`, the order of elements is not guaranteed unless it is a sorted collection



HashSet

- Indices in a **HashSet** are computed by the *hash function*
- Elements are accessed by iterating over the set
- Check if set contains the element by calling **`set.contains()`**
- Add elements by calling **`set.add(data)`**



HashSet example

```java
import java.util.*;
public class HashSetExample {
  public static void main(String[] args) {
    String[] names = {"Bob","Alice","Bob","Cecilia","David","Frank"};
    List<String> list = Arrays.asList(names);
    Set<String> set = new HashSet<>(list);
    System.out.println("The name list without duplicates:");
    for(String name : set) {
      System.out.println(name);
    }
  }
}
```

```
$ java HashSetExample
The name list without duplicates:
Cecilia
Bob
Alice
David
Frank
$
```

LinkedHashSet example

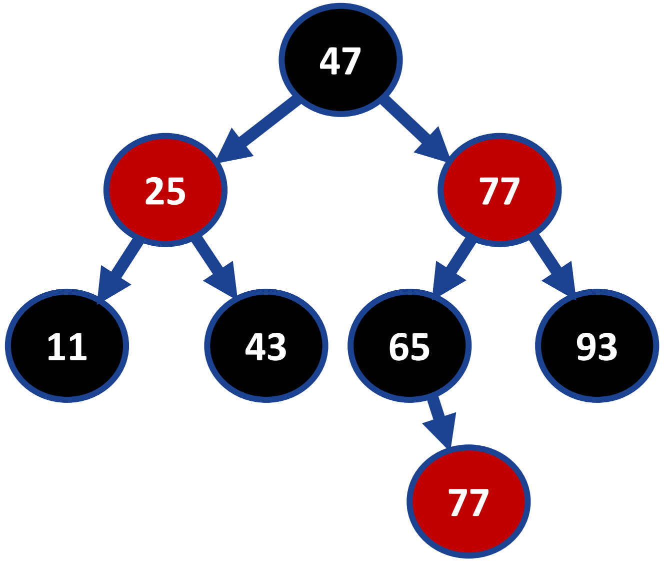
```java
import java.util.*;
public class LinkedHashSetExample {
  public static void main(String[] args) {
    String[] names = {"Bob","Alice","Bob","Cecilia","David","Frank"};
    List<String> list = Arrays.asList(names);
    Set<String> set = new LinkedHashSet<>(list);
    System.out.println("The name list without duplicates:");
    for(String name : set) {
      System.out.println(name);
    }
  }
}
```

LinkedHashSet is similar to HashSet, except that the order of elements is preserved

```
$ java LinkedHashSetExample
The name list without duplicates:
Cecilia
Bob
Alice
David
Frank
$
```

TreeSet

- Ideal for handling large quantities of sorted data
- Stores data in a *red-black self-balancing binary tree*
- Check if set contains the element by calling **`set.contains()`**
- Access elements by calling **`tree.first()`**, **`tree.last()`**, *or* iterating over the set
- Add elements by calling **`tree.add(`**`data`**`)`**

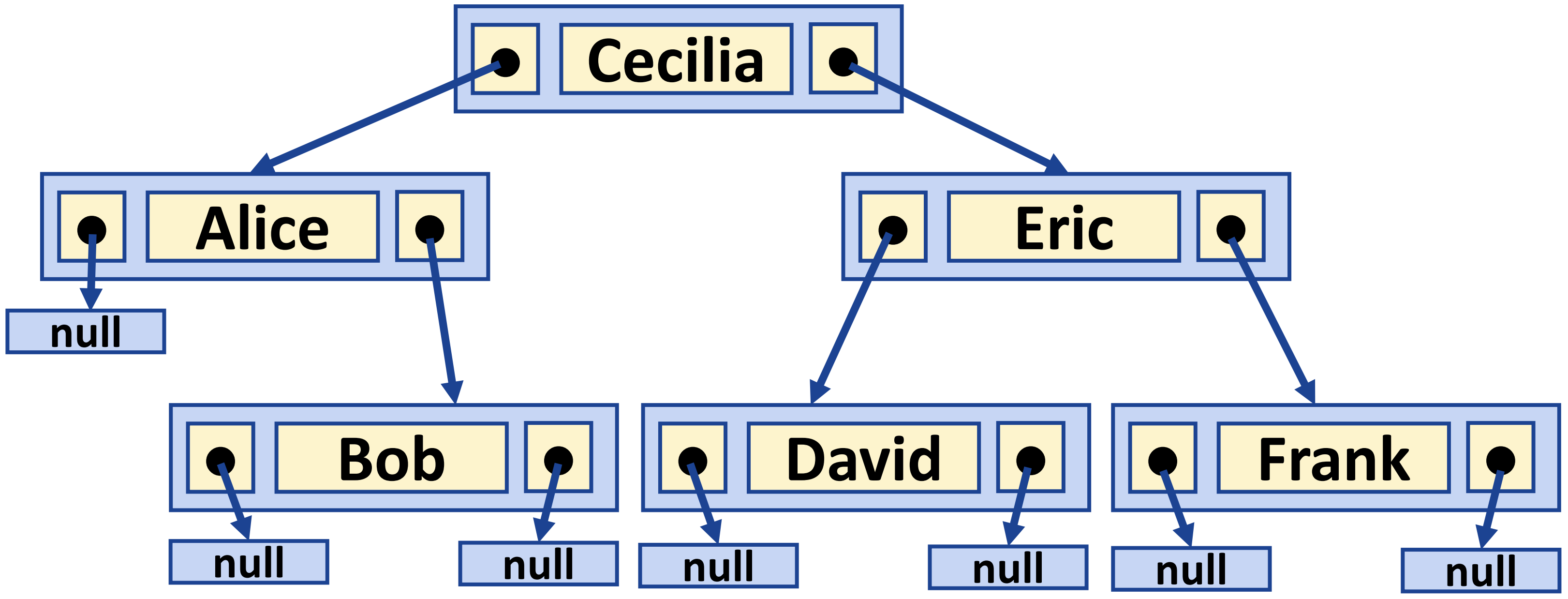


TreeSet example

```java
import java.util.*;
public class TreeSetExample {
  public static void main(String[] args) {
    String[] names = {"Bob","Alice,"Cecilia","David",“Eric","Frank"};
    List<String> list = Arrays.asList(names);
    SortedSet<String> set = new TreeSet<String>(list);
    System.out.println("Names before Cecilia:");
    for(String name : set.headSet("Cecilia")) {
      System.out.println(name);
    }
    System.out.println("Names from Cecilia:");
    for(String name : set.tailSet("Cecilia")) {
      System.out.println(name);
    }
    System.out.println("First name: " + set.first());
    System.out.println("Last name: " + set.last());
  }
}
```

```
$ java TreeSetExample
Names before Cecilia:
Alice
Bob
Names from Cecilia:
Cecilia
David
Eric
Frank
First name: Alice
Last name: Frank
$
```

Internal representation of the example



Remarks on collections

- Usually, the same task can be done using different types of collections
- However, different collections have different pros and cons
  - Complexity of searching, adding, and removing elements
  - Different methods for sorting, shuffling, reversing, etc. available
- Collections provide efficient built-in data structures and algorithms for many tasks: no need to reinvent the wheel!

Questions, comments?