Generics, Time Measurements, and Binary Heaps

Lecture 10 – Various left-overs!

Dr Jonas Lundberg

Jonas.Lundberg@lnu.se

March 4, 2019

Agenda

- Generic Classes
- Time and Memory Measurements
- Priority Queues
- Binary Heap

Reading Instructions
Horstmann: Sections 15.7 - 15.8, Chapter 16
Liang: Chapter 19, 24.6

Generic Classes – Introduction

- **Generics:** Classes with type parameters
  - Instead of implementing `IntList`, a list class for integers only ...
  - ... we implement `List<T>` and instantiate it as `List<Integer>` (or as `List<String>`, `List<Double>`, `List<Student>`, `...`)
  - ⇒ One class definition can generate many different types of objects.
- Generic classes was introduced in Java version 5 (JDK 1.5)
- Generic classes was introduced at the same time in C#
- A generic class is instantiated by providing a concrete type as a parameter to the constructor call.
- It might be more than one type parameter.
- The Java Library has a number of generic data structures
  - `ArrayList<T>`, `LinkedList<T>`, `Stack<T>`, `...`
  - `HashSet<T>`, `HashMap<K,V>`

Using Generics GenericMain.java

```
List<Integer> list = new ArrayList<Integer>();  // A generic integer list
for (int i=1; i<=5; i++)
    list.add(i);                               // add 1,2,3,4,5. Converted to Integer

for (int i=0; i<list.size(); i++) {
    int n = list.get(i);                      // Integer --> int
    System.out.print(" "+n);
}
System.out.println("\n");                      // line break

List raw = new ArrayList();                    // A "raw" integer list
for (int i=1; i<=5; i++)
    raw.add(i);                                // add 1,2,3,4,5. Converted to Integer

for (int i=0; i<raw.size(); i++) {
    int n = (Integer) raw.get(i);              // Object --> Integer --> int
    System.out.print(" "+n);
}
```

Use if possible the generic version. Thus avoiding 1) down casting when accessing data 2) Warnings from Eclipse.

Why Generics?

Why generics, why not use `Object` as a parameter?

- Use `Object` $\Rightarrow$ repeated down casts $\Rightarrow$ type checking at run-time
- Generic classes $\Rightarrow$ static type checking $\Rightarrow$ errors found at compile time

Using generic classes is simple $\Rightarrow$ **Use them!**

Implementing Generics

- Implementing generic classes is a bit more tricky

```
public class A<T> {
   public void a(T t) { "do something with t"}
   public T b(T t) { "return something of type T"}
}
```

- Remember: It must work on every possible type of object.

A Simple Example - `GenValue.java`

```
public class GenValue<T> {
    private T value;

    public GenValue(T val) { value = val; }

    public void setValue(T val) { value = val; }
    public T getValue() { return value; }

    @Override
    public String toString() { return "GV("+value.toString()+")"; }

    ... missing methods
}
```

Usage:

```
GenValue<Integer> i1 = new GenValue<Integer>(7);
i1.setValue(23);
System.out.println("Integers: "+i1.toString()+"\t"+i1.getValue());

GenValue<String> s1 = new GenValue<String>("Hello");
System.out.println("\nStrings: "+s1.toString()+"\t"+s1.getValue());
```

Example - `GenValue.java` (cont.)

```
public class GenValue<T> {
    private T value;

    public GenValue(T val) { value = val; }

    @Override
    public boolean equals(Object o) {  // Why not T as parameter type?
        if (o instanceof GenValue) {  // Compare with raw version
            GenValue other = (GenValue) o;
            return other.value.equals(value);
        }
        return false;
    }
}
```

Usage:

```
GenValue<String> s1 = new GenValue<String>("Hello");
GenValue<String> s2 = new GenValue<String>("World!");

String msg = s1.equals(s2) ? "equal" : "not equal";
System.out.println("They are "+msg);
```

Implementing Generic Classes

- Generic class ⇒ Class with type parameters (one or more)

  ```
  public class GenValue<T> {
  ...
  }
  ```

  E and T are often used but any upper case letter will do.
- Using generic classes: Exact type is decided when we create a new object

  ```
  GenValue<Integer> i1 = new GenValue<Integer>(7);

  GenValue<String> s1 = new GenValue<String>("Hello");
  ```
- Must work for all parameter types T ⇒ expect only class Object properties
  ⇒ Methods like `equals()`, `hashCode()`, `toString()`, `clone()`
- The possible types can be limited by *constraining* the type parameter
  ⇒ An increased set of methods to play around with. (Next Example)

Generic Methods - `ArrayUtil.java` (cont.)

```
public class ArrayUtil {                              // Non-generic class
    public static <T> void print(T[] arr) {           // Generic method
        for (T t : arr)
            System.out.print(t + " ");
        System.out.println();  // Line break
    }
    public static <E extends Comparable<E> >  E findMin(E[] arr) {
        E min = arr[0];
        for (int i=0; i<arr.length; i++) {
            if (arr[i].compareTo(min) < 0)  // compareTo available since all
                min = arr[i];               // types must implement Comparable
        }
        return min;
    }
}
```

Usage:

```
String[] strings =  {"Hej","Hola","Hello","Ciao"};
ArrayUtil.print(strings);
String min = ArrayUtil.findMin(strings);
System.out.println("Min: "+min);
```

Generic Methods

- A non-generic class might have generic methods

```
public class ArrayUtil {                              // Non-generic class
    public static <T> void print(T[] arr) {    // Generic method
        for (T t : arr)
            System.out.print(t + " ");
        System.out.println();  // Line break
    }
}
```

- Notice: Type parameter `<T>` inserted after modifiers (`public static`) but before return type (`void`)
- Using generic methods:

```
String[] strings = new String[] {"Hej","Hola","Hello","Ciao"};
ArrayUtil.print(strings);
```

$\Rightarrow$ No type specification `<String>` needed. The compiler deduces that it is type `String`

- Generic methods are seldom used. However, they might come in handy for static utility methods like `print` and `findMin` in example `ArrayUtil`.

Constraining Type Parameters

```
public static <E extends Comparable<E>>  E findMin(E[] arr) {
    E min = arr[0];
    for (int i=0; i<arr.length; i++) {
        if (arr[i].compareTo(min) < 0)  // compareTo available since all
            min = arr[i];               // types must implement Comparable
    }
    return min;
}
```

- `<E extends Comparable<E> >` ⇒ Any class E implementing Comparable
- We are decreasing the number of possible parameter types
- We are increasing the number of available methods to use

A Generic List (GenericList.java)

Non-generic interface

```
public interface IntList extends Iterable<Integer> {
   public void add(int n);          // Add integer n to list.
   public boolean contains(int n);  // true if n is in list, otherwise false.
   public int get(int index);       // Get integer at position index.
   public int size();               // Number of integers currently stored.
   public String toString();        // String of type "[ 7 56 -45 68 ... ]"
   public void addAll(Iterable<Integer> col); // Append iterable collection
                                              // of elements to the list.
}
```

Generic interface

```
public interface GenericList<T> extends Iterable<T> {
   public void add(T t);          // Add element t to list.
   public boolean contains(T t);  // Returns true if element t is in list, othe
   public T get(int index);       // Get element at position index.
   public int size();             // Number of elements currently stored.
   public String toString();      // String displaying  list content on one lin
   public void addAll(Iterable<T> col); // Append iterable collection
                                        // of elements to the list.
}
```

My Generic List (GenericArrayList.java)

```
public class GenericArrayList<T> implements GenericList<T> {
   private int size = 0;
   private T[] values;

   public GenericArrayList() {values = (T[]) new Object[8];}

   public void add(T t) {
      values[size++] = t;
      if (size == values.length) { // increase size
         resize();
   }

   private void  resize() {   // Private help method
      T[] tmp = (T[]) new Object[2*values.length];
      System.arraycopy(values,0,tmp,0,values.length);
      values = tmp;
   }
```

Method addAll(Iterable<T> col)

- addAll(Iterable<T> col) ⇒ Add elements using any collection class implementing the Iterable interface
- For Example: ArrayList, HashSet
- Implementation is straight forward

```
public void addAll(Iterable<T>  col) {
     for (T t : col)
          add(t);      // Our own add method (with resizing)
}
```

Usage:

```
GenericList<String> strList = new GenericArrayList<String>();
ArrayList<String> sL = new ArrayList<String>();     // Implements Iterable
... add strings to ArrayList

strList.addAll(sL);        // Add ArrayList content to strList
System.out.println(strList);
```

It is sometimes very easy ...

Converting a non-generic class to a generic version is sometimes very easy.

```
// Non-generic
public boolean contains(int n) {
   for (int i=0;i<size;i++) {
      int v = values[i];
      if (v == n)
         return true;
   }
   return false;
}
```

```
// Generic
public boolean contains(T t) {
   for (int i=0;i<size;i++) {
      T v = values[i];
       if (v .equals(t))
            return true;
   }
   return false;
}
```

Suggestion

- When new to generics, start by implementing a non-generic version.

Non-Generic Iterators

```
public class ArrayIntList implements Iterable<Integer> {
    private int size = 0;
    private int[] values;

    ... many methods are dropped

    public Iterator<Integer> iterator() { // Required by Iterable interface
        return new ListIterator();
    }

    private class ListIterator implements Iterator<Integer> {
        private int count = 0;

        public Integer next() { return values[count++]; }
        public boolean hasNext() { return count<size; }
        public void remove() {
            throw new RuntimeException("remove() is not implemented");
        }
```

The Iterator (GenericList.java)

```
public class GenericArrayList<T> implements GenericList<T> {
   private int size = 0;
   private T[] values;

   public Iterator<T> iterator() {  // Implements Iterable in GenericList<T>
      return new ListIterator<T>(values);  // Create iterator of type <T>
   }                                       // using ListIterator<X> below

   private class ListIterator<X> implements Iterator<X> {
      private int count = 0;        // Notice: We are not using same type T
      private X[] elements;         // as the enclosing class GenericArrayList

      public ListIterator(X[] xElements) {elements = xElements;}

      public X next() {return elements[count++];}
      public boolean hasNext() {return count<size;}

      public void remove() {  // Marked as optional in Iterator<X>
         throw new RuntimeException("remove() is not implemented");
      }
   }
```

Generics: Readings and Exercises

- Chapter 16 in book by Horstmann
- Sun's Generics Tutorial

  `http://java.sun.com/docs/books/tutorial/java/generics`
- Study generic examples provided in the lecture material

In general, Sun's tutorials can be found at

`http://java.sun.com/docs/books/tutorial/java`

Exercises in Assignment 4

1. A generic linked queue.
2. Update JUnit queue test to handle generic queue
3. A generic array-based queue (VG Exercise)

Feel free to reuse your own Assignment 2 solution. But handle comments from teaching assistants first!

A 10 Minute Break ...

ZZZZZZZZZZZZZZZZZZZZZZ...

Measure Time

The class java.lang.System supports time measurements

- Example 1: Using `static long currentTimeMillis()`

  ```
  long before = System.currentTimeMillis();
      ... do something
  long after = System.currentTimeMillis();
  long estimatedTime = after-before;    // time in milliseconds
  ```

- Example 2: Using `static long System.nanoTime()`

  ```
  long before = System.nanoTime();
      ... do something
  long after = System.nanoTime();
  long estimatedTime = after-before;    // time in nanoseconds
  ```

- `currentTimeMillis()` vs `nanoTime()`
  - `currentTimeMillis()` has an interpretation
    $\Rightarrow$ time in milliseconds since January 1, 1970
  - `System.nanoTime()` has no interpretation. This method can only be used to measure elapsed time. According to Java: *Returns the current value of the most precise available system timer, in nanoseconds.*
  - `nanoTime` are using the operating system clock $\Rightarrow$ accuracy might change from one OS to another

Measure Memory

The class java.lang.Runtime supports memory measurements

- `static Runtime getRuntime()`
  ⇒ get access to current runtime enviroment
- `long totalMemory()`
  ⇒ Memory (in bytes) allocated for the JVM
- `long freeMemory()`
  ⇒ Memory (in bytes) currently available for the JVM
- `void gc()` ⇒ Run garbage collection
- Example:

```
Runtime runtime = Runtime.getRuntime();
runtime.gc()                           // Clear memory
long usedMemory = runtime.totalMemory() - runtime.freeMemory();
long mbytes = usedMemory/1000000;     // Used memory (in MB)
```

- Increase the JVM heap space to increase available memory
- We use JVM arguments: `-Xmx2048m -Xms2048m` in the MathSet competition
  `-Xmx` ⇒ Max memory, `-Xms` ⇒ Initial memory
  ⇒ In MathSet competition we use 2GB of memory already from the start
- In Eclipse: Run ⇒ Run Configurations ⇒ Arguments ⇒ VM Arguments

Assignment 4 – Exercise 3

- Problem: Concatenate a large number of strings
- Two possible solutions:
  1. Use the concatenation operator +
     ⇒ Repeated `str = str + " .... ";`
  2. Append strings to a StringBuilder

     ```
     StringBuilder buf = new StringBuilder();
     buf.append(" ... ");

     String result = buf.toString();
     ```
- Which approach is the fastest?
- Exercise 3: How many concatenations in 1 seconds?
  - Case 1: Concatenate short string (only one character)
  - Case 2: Concatenate long strings (a row with 80 characters)

  We want number of concatenations and total length of final string for each case, using each approach ⇒ 8 numbers in total

Advices

- Do not trust time measurements of less than (say) 30 ms
- Solution: Repeat experiment many times and compute average values

```
long before = System.nanoTime();
for (int i=0; i<10; i++ ) {
    ... do something
}
long after = System.nanoTime();
long estimatedTime = (after-before)/10;    // average time
```

- Measuring time for fast operations like binary search and hashing is very difficult.
  - Requires very large input sizes to get trustworthy time measurements
  - Large sizes ⇒ memory problems
  - Memory problems ⇒ garbage collection becomes a problem
- Remember
  - Your experiment is not the only activity running on the computer
  - Try to shut down other applications
  - However, the operating system will still be working in the background

Repeated runs give very different result ⇒ something is wrong!

Assignment 4 - Exercise 4 and 5

Exercise 4

- Evaluate performance of you sorting algorithms from Assignment 2
- Compare Insertion Sort with Merge Sort for both integers and strings
- How long arrays can be sorted in 1 second?

Exercise 5

- Write a short report for Exercises 3 and 4. For each exercise
  1. Experimental Setup: How did you perform your experiment? Did you compute an average over several runs? What type of strings where used when sorting strings? Motivate your decisions!
  2. Experimental Results: Present table of experimental results. Explain the presented data.
  3. Discussion, Conclusion, and Future Work: Are you satisfied with all your experiments? Explain problems/outliers. What are your conclusions? Do you find situations where one algorithm clearly outperfoms the other? What additional experiments should you have done (given a bit more time)? Try to motivate all your statements.

We expect about 1 page for each experiment $\Rightarrow$ 2 pages in total

Example: A simple Experiment

Task: How long integer lists can be sorted in 1 second using the sort method in the class java.util.Collections?

Experimental Setup

- Machine: MacBook Pro with an Intel Core i7 processor (2.2GHz) with 8GB of memory.
- Java version: JavaSE-1.8
- List version: java.util.ArrayList
- Clock: System.currentTimeMillis()
- JVM Memory size (VM Argumeents): -Xmx4096m -Xms4096m.
- Always compute average of 10 runs
- Always generate data (10 random integer lists) before any measurement
- Run garbage collection before each measurement
- Warm up: 5 unsorted integer lists

Example: Strategy

1. A test few runs ⇒ find list size N that roughly takes 1 second to sort
2. Find interval $[N_{min}, N_{max}]$ definitely containing N
3. Make time measurements for (say) 10 sizes covering the interval

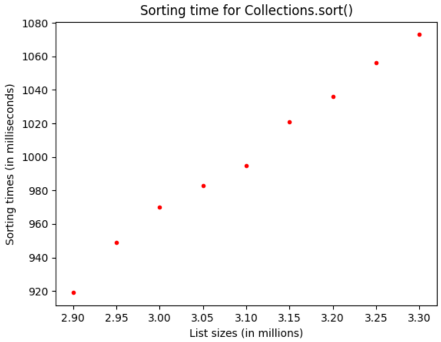
In my case: $N \approx 3.1$ million, interval is [2.9, 3.3] million

Experiment output

```
Warming up
Size = 3100000 ==> Time = 1015, Memory = 769 Mbytes
.. four more warm-up runs

Actual Runs
Size = 2900000 ==> Time = 919, Memory = 724 Mbytes
Size = 2950000 ==> Time = 949, Memory = 734 Mbytes
... four more measurement runs
Size = 3250000 ==> Time = 1056, Memory = 801 Mbytes
Size = 3300000 ==> Time = 1073, Memory = 811 Mbytes
```

Example: Results



- No problematic outliers ⇒ experiment is OK!
- From figure: Time 1000 (milliseconds) ⇒ Size 3.1 (millions)
- Better estimate if we use regressions methods (machine learning) and add more measurements

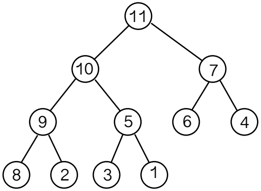
Priority Queues

A *priority queue* is a data structure similar to a regular queue where:

- Each element has a priority associated with it
- An element with high priority is served before an element with low priority
- Same priority ⇒ they are served in insertion order
- Most significant operations:
  - `insert` ⇒ add an element to the queue with an associated priority.
  - `pullHighest` ⇒ remove (and return) element with the highest priority
- From now on ...
  - A priority is a positive integer
  - Value 1 ⇒ lowest priority
  - Highest value ⇒ highest priority

Binary Heaps

Binary heap ⇒ a binary tree representation suitable for priority queues



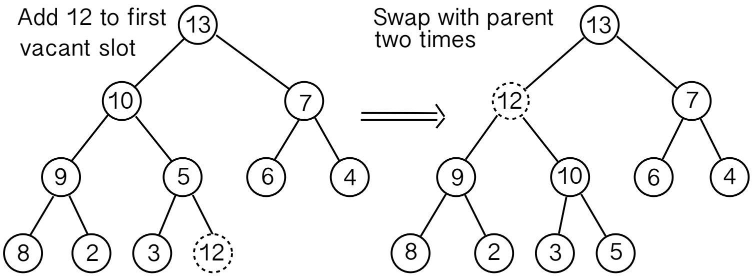
A binary heap is a binary tree with two special properties

1. *Almost completely filled*: All nodes are filled except the last level that might have some nodes missing toward the right.
2. *Heap ordering*: The priority in a node is always higher (or equal to) than the priority of its children.

Notice

- The element with highest priority is stored in the root
- Not the same type of tree as binary search trees (BST)
  - The heap shape is very regular whereas BSTs can have an arbitrary shape
  - Left and right child store elements with lower priority than the node itself.

Insert New Element

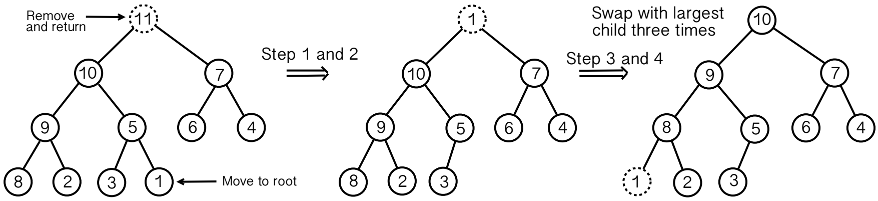


Algorithm for inserting new element

1. Add node/element to first vacant slot at the end of the tree
2. As long as parent node has lower priority, swap place with parent node. Repeat process until parent has higher priority or we have reached the root node.

Step 2 is called *percolation up*. (Straight forward to implement.)

Pull Highest Priority Element

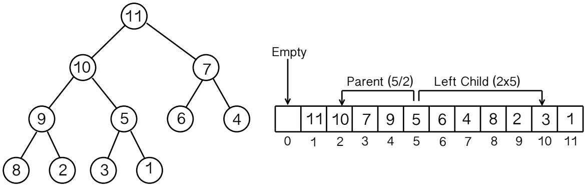


Algorithm for popping element with highest priority

1. Extract root node value (and return it later on)
2. Move value of last node into the root node
3. If root have lower priority than any of its children, swap place with largest child.
4. Repeat swap with largest child until all children have lower priority, or no more children

Steps 3 and 4 are called *percolation down*. (A bit tricky to implement.)

Array Based Binary Heaps



An array can be used to store the elements in a binary heap

- Position 0 is never used
- The root node is always in position 1
- Elements are added layer by layer
- For an element in array position N it holds:
  - The parent is in position $N/2$
  - The left child is in position $2 \cdot N$
  - The right child is in position $2 \cdot N + 1$
- Hence, both `insert` and `pullHighest` can be done on the array.
  - `insert` $\Rightarrow$ percolation up $\Rightarrow$ move upwards in the tree using $N/2$
  - `pullHighest` $\Rightarrow$ percolation down $\Rightarrow$ move down using $2 \cdot N$ or $2 \cdot N + 1$

Binary Heaps - Summary

- Binary heap is an implementation technique for priority queues
  (Just like linked list is a technique to implements lists)
- We *think* of the binary heap as an almost complete binary tree (with a certain heap ordering)
- insert $\Rightarrow$ add at first vacant slot and percolate up to find correct position
- pullHighest $\Rightarrow$ remove root, move last to root, and percolate down to find correct position
- We *implement* binary heaps using arrays. For a node in array position N:
  - Parent is in position $N/2$
  - Left child is in position $2 \cdot N$
  - Right child is in position $2 \cdot N + 1$
- Both `insert` and `pullHighest` can be done on the array.
  - `insert` $\Rightarrow$ percolation up $\Rightarrow$ move upwards in the tree using $N/2$
  - `pullHighest` $\Rightarrow$ percolation down $\Rightarrow$ move downwards in the tree using $2 \cdot N$ (left child) and $2 \cdot N + 1$ (right child)

Read textbook (Horstmann) or Google Internet for more details.

Assignment 4 – Exercise 6

Task: Implement a class BinaryIntHeap containing the methods below following the standard rules for how to implement a binary heap. Write also a JUnit test case to check the correctness of your heap implementation.

```
public BinaryIntHeap()       // Constructs an empty heap
public void insert(int n)    // Add n to heap
public int pullHighest()     // Return and remove element
                             // with highest priority
public int size()            // Current heap size
public boolean isEmpty()     // True if heap is empty
```

Notice: In this very simple approach the element and the priority are the same. More general tasks will be handled in the next exercise.

Assignment 4 – Exercise 7 (VG Exercise)

A *Priority Queue* is a data structure that allows the processing of a number of *Tasks* based on some priority.

1. Design two interfaces (or abstract classes) named `PriorityQueue` and `Task` that together describes a priority queue in general.
2. Provide a concrete priority queue implementation named `BinaryHeapQueue` and a concrete task implementation named `WorkTask`. In addition to a priority (positive integer), a WorkTask also comes with a work description (a string).

We also expect you to write a small program WorkMain that demonstrates how to use your priority queue.

Notice: Good design implies flexibility and extendability. Hence, try to make it easy to replace the WorkTask with another type of task that also implements the Task interface. Also, it should be easy to switch from one PriorityQueue implementation (e.g BinaryHeapQueue) to another.