GENERAL NOTES

ANDREW S. HARD[1]

January 12, 2015

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 *Department of Physics, University of Wisconsin, Madison, United States of America*

1 INTRODUCTION

As a fifth year graduate student, I have come to the realization that significant portions of my skillset and knowledge base are highly specialized. I intend to pursue a career outside of my field of study (experimental high-energy particle physics). In order to enhance my future job prospects, I decided that it would be useful to review basic concepts in computer science, statistics, mathematics, and machine learning. Most of the derivations and explanations will be borrowed from other sources. A list of references is currently being compiled.

2 DATA STRUCTURES

2.1 Java Basics

In Java, there are **primitive types** and **reference types**. Primitive types are like `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, `byte`. Reference types are **arrays** and **classes**.

When you declare a variable with a reference type, you get space for a reference or **pointer** to that type, not for the type itself. Use `new` to get space for the type itself. Non-primitive types are really pointers. Assigning 1 variable to another can cause **aliasing** (two or more different names reference the same object).

For each **primitive type** there is a "box class" or "wrapper class". For instance, the `Integer` class contains one `int` value. Java **autoboxing** converts between the primitive and wrapper classes as appropriate.

Java also has **visibility modifiers**, which specify how code can be accessed.

- *public* accessible to any code
- *private* accessible to code within same class
- *protected* accessible to code within same class or subclasses
- *public* accessible to code within same package (directory)

Numerical comparisons can be performed using the "<", ">", "=" signs for `int`, `float`, `double` primitives. However, the **Comparable** interface should be used for objects in Java (`String`, `Integer`, `Double`). The `compareTo` method returns negative value if "less than", zero if "equal to", and positive if "greater than".

2.2 Complexity

Complexity is a way to gauge the performance of an algorithm: how much of the computing resources are used when a program executes. **Complexity** measures how the resource requirements scale as the problem gets larger. The time requirement is basically proportional to the number of basic operations such as '+', '*', one assignment, one logical test, one read, or one write. The **problem size** is related to the number of important factors. In sorting, for example, this could be the number of elements to sort.

We are usually interested in the worst case number of operations, not the exact number of operations, which can vary. The average and best cases

do provide useful information, however. We use the *big-O* notation to write complexity: $O(N)$ is linear complexity, $O(N^2)$ is quadratic complexity, etc.

2.3 Abstract Data Types

Abstract Data Types (**ADTs**) separate the *specification* of a data structure from the *implementation* of that data structure. The specification refers to the methods and operations that can be performed on it, while the implementation refers to how those features are programmed. The advantage to using ADTs is that code is more reusable: the implementation of certain code can be changed without changing the programs that use the code. An ADT corresponds to a class or many implementing classes. Operations on an ADT are the class's **public methods**. Example operations on an ADT include: initialize, add data, access data, remove data.

ADTs have 2 parts.

- public/external - conceptual picture & conceptual operations
- private/internal - the representation (how the structure is stored) and the implementation of operations (code)

To implement an interface in Java: (1) include "`implements InterfaceName`" in the class declaration, changing `InterfaceName` as appropriate, (2) define a public method for the class for each method signature in the interface.

To specify that something is a subclass, use `extends`. This says that the current class definition inherits from the specified class. For instance, `MyException` inherits from the `Exception` class in the example below.

```
public class MyException extends Exception {...};
```

2.4 Exceptions

Errors can arise due to *user error* (e.g. wrong inputs) or *programmer error* (e.g. buggy code). It is desirable to pass the error up from low-level methods to a level where it can be handled properly. In C++, methods usually just return a special value to indicate an error. Requires calling code to check for error. In Java, expections allow error testing in the code.

The idea is to **throw** an **exception** when an error is detected. The code causing the error stops executing immediately and control is transferred to the **catch clause** for that type of exception of the first enclosing **try block** that has such a clause. The try block might be in the current method or in a method that called the current one. The exception is passed up the call chain, all the way up to `main` or the Java virtual machine if necessary (at which point the program stops and an error message appears). The code below provides an example.

```
try {
  // code that might cause exceptions.
} catch( ExceptionType1 id1 ) {
  // statement to handle this exception type.
} catch( ExceptionType2 id2 ) {
  // statement to handle this exception type.
} finally {
```

```
    // statement to execute every time this try block executes.
  }
```

Each `catch` clause specifies one type of exception and provides a name for it, like method objects. The `finally` clause is optional. It *always* executes if the `try` block is entered. Note: you can have zero catch clauses and still have a finally clause.

Java's standard libraries have several built-in exceptions: `ArithmeticException`, `ClassCastException`, `IndexOutOfBoundsException`, `NullPointerException`, `FileNotFoundException`...

All exceptions are either **checked exceptions** or **unchecked exceptions**. If a method includes code that could cause a checked exception to be thrown: (1) the exception must be declared in the method header using a **throws** clause, or (2) the code that might cause the exception to be thrown must be in a try block with a catch clause for that exception. Otherwise you will get a compiler error. Unchecked exceptions do not need to be listed in the throws clause of a method. The code below provides an example.

```
  public static void main( string[] args ) throws FileNotFoundException,
EOFException {
    // those two exceptions might be thrown here.
  }
```

Unchecked exceptions are subclasses of `RuntimeException`, while checked exceptions are subclasses of `Exception` only, not `RuntimeException`. Most built-in Java exceptions are unchecked, with the exception of I/O exceptions, which are checked. user defined exceptions should usually be checked.

Java exceptions are objects, and as such a new exception is made by defining a new instantiable class. It must be a subclass of `Throwable`, usually a subclass of `Exception`. An example is given below.

```
public class EmptyStackException extends Exception {
  public EmptyStackException() { super(); }
  public EmptyStackException(string message) { super(message); }
};
```

At the point in code where an error is detected, the exception is thrown using a `throw` statement.

```
if (...)  { throw new EmptyStackException(); }
```

2.5 List Structure

2.5.1 *Basic List*

A **List** structure is an ordered collection of items of some element type *E*. One advantage of lists over arrays is that they are variable size. A disadvantage is that you cannot have an empty list with non-zero size. Lists only hold objects, whereas arrays hold any type (even primitive types such as `int`, `char`).

The table below provides the public interface for the `ListADT`.

`void add(int pos, E item);`	Add an item at a specified position
`void add(E item);`	Append item to end of list
`boolean contains(E item);`	Check if list contains item
`int size();`	Get the number of items in the list
`boolean isEmpty();`	Check if list contains zero items
`E get(int pos);`	Get the item at the specified position
`E remove(int pos);`	Remove, return item at specified pos.

To instantiate `ListADT`, for a list of `Integer` type items named `number` using the `ArrayList` class:

```
ListADT<Integer> numbers = new ArrayList<Integer>();.
```

`Iterator` is a defined interface in `java.util`. Iterating accesses each item in a list in turn. The interface is `Iterable`. Every java class that implements Iterable (from `java.lang` provides an `iterator` method that returns an `Iterator` object for that collection. For example, the `List<E>` interface has an `iterator()` method:

```
Iterator<string> itr = words.iterator(); //words is a List<String>
```

The easiest way to implement an iterator for a class is usually to define a new class. For example, in the `SimpleArrayList` class, the `iterator()` method should return a new `SimpleArrayListIterator` object.

Here's a simple overview of an iterator class:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class SimpleArrayListIterator implements Iterator<Object>
{
    // methods:
    public boolean hasNext(){...}
    public Object next(){...}
    public void remove(){...}
};
```

2.5.2 *Linked Lists*

Linked lists are a type of list in which each item in the list contains the item data as well as a reference to the next element in the list. The list is built using `Listnode` objects that store data as well as the next listnode. The `getNext()` method returns the next item in the list, which can also be set using the `setNext(Listnode<E> n)` method.

Linked lists often have a **header node** – a dummy node at the front of the list. Header nodes do NOT contain data, the only include a reference for the next item. A **tail pointer** references the last listnode.

There are many variants of linked lists, including doubly linked lists (which have a reference to the previous and next elements) and circularly linked lists (where the tail node `next` is linked to the first node).

2.6 Stacks and Queues

Stacks and queues are like lists, but with more limited operations. They can be implemented using an array or linked list.

2.6.1 *Stacks*

A **stack** is a stack of items. Only the top item can be accessed. Items are added to the top of a stack. It is a basic **LIFO** structure: a last-in-first-out data type. A basic stack interface is provided below.

```
public interface StackADT<E> {
  boolean isEmty(); //return true iff empty
  void push(E obj); // add object to top of stack
  E pop() throws EmptyStackException; //remove
  E peek() throws EmptyStackException; // see
};
```

2.6.2 *Queue*

A **queue** is a structure similar to a line or queue. Items go in one side and out the other. It is a basic **FIFO** structure: a first-in-first-out data type. Items are added to the *rear* of the queue and removed from the *front* of the queue.

```
public interface QueueADT<E> {
  boolean isEmty(); //return true iff empty
  void enqueue(E obj); // add object to end of queue
  E dequeue() throws EmptyQueueException; //remove from front
};
```

2.7 Recursion

A method is **recursive** if it can call itself either directly or indirectly. Recursive code is often *simpler*, but not faster or more efficient. There are a few rules associated with recursion.

- *Recursion rule 1*: recursive methods must have a **base case** - a condition under which no recursive call is made. This prevents infinite recursion.
- *Recursion rule 2*: every recursive method **make progress** towards a base case.

At runtime, a stack of activation records (AR) is maintained (one for each active method). This is also named the **call stack**. It includes the method's parameters, local variables, and return address. When a method is called, its AR is pushed onto the stack. When too many methods are called, a **stack overflow** error is produced.

A data structure is recursive if it can be defined in terms of itself.

2.8 Searching in an Array

There are two basic approaches to searching in an array: **sequential** search and **binary** search. Sequential search involves looking at each value in turn. It is faster for sorted arrays $O(N)$. Binary search (for sorted array) involves looking at the middle item, comparing with the value of interest, then eliminating half of the array from the search $O(\log(N))$.

2.9 Tree Structure

So far, the structures discussed have been **linear data structures**. These incude lists, stacks, queues, and other structures in which items are stored with one item following another. **Trees** are an example of a **nonlinear data structure**. More than one item can follow from a single item. Furthermore, the number of following items can vary from one item to another. Trees are useful because they can provide *fast access* to information in a database.

Some tree terminology is provided in the table below.

Node	A single element of the tree
Edge	Arrow connecting two nodes
Root	Top-most node of the tree (has no **parent**)
Leaf	bottom nodes (has no **children**)
Ancestors	The preceding nodes in a tree
Descendants	The succeeding nodes in the tree
Path	A sequence of conected nodes
Path Length	Number of nodes in the path
Height	Length of *longest* path from root to a leaf
Depth	Path length from a given node back to root

2.9.1 *Binary Trees*

Binary trees are a special type of tree in which (1) each node has 0, 1, or 2 children and (2) each child is either a *right* or *left* child.

Programming a binary tree usually requires a tree class and a tree node class. The `BinaryTreenode` class only requires 3 fields: `data`, `leftChild`, and `rightChild`. A generic tree, by comparison, could just use a list of nodes for the children.

There are several ways to traverse a tree, as shown in the table below. V=view/visit, L=go to left child, R=go to right child.

Name	*for binary tree*	*for other tree*
pre-order	VLR	VC
post-order	LRV	CV
level-order	left to right from top to bottom	
in-order	LVR	(binary only)

2.9.2 *Binary Search Trees*

A **binary search tree** (BST) is a type of binary tree that stores sorted values. For every node n with a **key value** (and maybe data):

- all keys in n's left sub-tree are less than the key in n,
- all keys in n's right sub-tree are greater than the key in n.

A binary search tree has efficient implementations of the insert, contains, remove, and print methods. Example structures for the node and tree classes are provided below.

```
class BSTnode<K> {
  private K key;
  private BSTnode<K> left, right;
  public BSTnode(K key, BSTnode<K> left, BSTnode<K> right) {...} ;
  //accessors:
  public K getKey(){...}
  public BSTnode<K> getLeft(){...}
  public BSTnode<K> getRight(){...}
  //mutators:
  void setKey(K key){...}
  void setLeft(BSTnode<K> left){...}
  void setRight(BSTnode<K> right){...}
};

public class BST<K extends Comparable<K> > {
  private BSTnode<K> root;// root of BST
  public void insert(K key){...}
  public void delete(K key){...}
  public boolean lookup(K key){...}
  public void print(PrintStream p){...}
};
```

Of course, it is also possible to design a BST that stores data along with each key.

The `lookup(K key)` method is an example of a recursive algorithm. The base cases are an empty tree, which returns false, or value is in the root node, which returns true. Check the root, left, and right subtrees. The worst-case lookup for a BST is $O(N)$, and corresponds to a stalky shape. Best case performance for a BST lookup is $O(\log(N))$ for a full, balanced tree.

Question: how do you **delete** a node in a tree with children? And with what do you replace the deleted node? Possibilities:

- the largest value in the left subtree (**in-order predecessor**).
- the smallest value in the right subtree (**in-order successor**).

The Java standard library implements a BST with the `TreeSet` and `HashSet` classes. Use `Map` and `TreeMap` to associate data with each key.

A few additional tree types are listed below.

- **Full tree**: all the leaves have the same depth.
- **Complete tree**: full to depth H-1, depth H filled from left to right.
- **Height-balanced tree**: difference in heights of *left* and *right* subtrees is at most 1.
- **Balanced tree**: has a height of $O(\log(N))$, where N=# nodes.

2.9.3 *Red-Black Trees*

Red-black trees (RBT) are a type of balanced tree. In order to preserve balance, the `insert` and `delete` operations may restructure the tree. Each node of the tree has a **color** (red or black), and 3 properties hold:

- **Root property**: the root of the tree is black.
- **Red property**: children of a red node are black.
- **Black property**: for each node with at least 1 null child, the number of black nodes on the path from the root to the null child is the same.

We want to perform insert, lookup, delete, and print operations on this tree. Lookup and print for the RBT are identical to those of the BST. The lookup, insert, delete operations have $O(\log(N))$ complexity for the worst-case.

Here's an overview of the RBT insert algorithm for node with key K.

- Use the BST insert algorithm to add K to the tree.
- Color the node containing K **red**.
- Restore the RBT properties if necessary.

Insertion can lead to a red property violation **RPV**. Here's how to deal with that.

- *Case 1*: if K's parent P is black, then do nothing (no RPV).
- *Case 2a*: if P's sibling is black or null, do a **trinode restructuring**.
- *Case 2b*: if P's sibling S is red, do a **recoloring** of P, S, G (unless G is the root).

2.10 Graph Structure

Define the **depth-first search**, **breadth-first search**, etc.

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph [**?**]. The original form of Dijkstra's algorithm has time complexity $O(|V|^2)$, where $|V|$ is the number of nodes. However, there is a priority-queue implementation of the algorithm that has worst-case performance of $O(|E| + |V| \log |V|)$, where $|E|$ is the number of edges.

Dijkstra's algorithm is known as a uniform-cost search, and is a type of best-first search.

2.11 Hash Maps