Data Structures 1

(Notes by Matt Fontaine, very lightly edited by Arup Guha)

Overview

The purpose of this lecture is to give a basic understanding of data structures that are useful to contest programming. It will cover a basic understanding of how they work, useful methods for the data structures, and when they can be applied. The runtime for each method will also be provided for runtime calculation purposes.

Arrays

Most of you already know how to use arrays. One helpful piece in contest programming is the Arrays class. It contains many static methods easing programming of arrays and ultimately makes your life easier. Here we will look at a few useful methods for getting the most out of arrays.

```
Arrays.fill(array, someValue);
Arrays.fill(array, fromInclusive, toExclusive, value);
```

Fill is a useful method in the arrays class. It, like many of the static methods in Arrays, is implemented several times in java to be useful on all its primitives. (e.g. long, int, char, byte, float, double, boolean)

I find this method particularly useful when filling in a memo table with -1 for dynamic programming solutions or filling distance arrays with large values for BFS or dijkstra's algorithm. The method simply sets all values in the array to a new value. The runtime is the same as a for loop over the array filling with values.

```
Arrays.copyOf(array, length);
Arrays.copyOfRange(array, fromInclusive, toExclusive);
```

These two methods allow you to copy each array or a portion of each array. If length is longer than the original array, the copy will be padded with zeros.

```
Arrays.toString(array);
```

This method is useful for debugging your program. Sometime it is useful to know the values of one of your arrays in the middle of your program. This handy function gets a string representation of the array for printing. All the primitives work great. If you have an array of objects then each object must have a toString method. You can implement that in the following way:

```
class MyClass
{
   public int toString()
   {
      return "string representation of class";
   }
}
```

```
Arrays.sort(array);
Arrays.sort(array, fromInclusive, toExclusive);
```

This method is useful when trying to sort an array of data. For primitives, the array will be put in increasing order of value. Unfortunately, you cannot change the order arrays of primitives are sorted. Sorts in Java run in $O(n \log n)$ runtime. Sorting a large number elements will run in time for most problems.

For arrays of objects you can specify a comparator. This comparator will allow you to sort the array in a custom way ignoring the objects compareTo method.

This can be done in line in java, in this example we will reverse the order integer is compared:

```
Integer[] array = new Integer[N];
Arrays.sort(array, new Comparator(){
      public int compare(Integer a, Integer b){
         return b.compareTo(a);
      }
   });
```

If you are building a custom object, it is also possible to implement your own compareTo function in the object.

For example:

```
class Point implements Comparable<Point>
{
   int x, y;

   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   // This method sorts points by x values and, if there is a tie, y
   // values.
   public int compareTo(Point rhs)
   {
      if (x == rhs.x)
         return Integer.compare(y, rhs.y);
      return Integer.compare(x, rhs.x);
   }
}
```

Once you build this class, if you have an array of Point called mypts, you would call Arrays.sort(mypts). If you had an ArrayList of Point called ptlist you would call Collections.sort(ptlist).

ArrayList

ArrayList is a useful data structure when the size you need for the structure is not known. In practice, the structure is faster than its LinkedList counterpart. So it is generally preferable in contest programming to use an ArrayList for runtime purposes. ArrayList has a little more overhead due to object creation and extra memory used. This is due to ArrayLists requiring that they contain objects only and not primitives.

Collections in java are generally implemented with generics. This allows you to specify the type of object that is contained in the collection:

```
ArrayList<Type> myList = new ArrayList<Type>();
```

As a specific example you can create a list of integers like so:

```
ArrayList<Integer> myList = new ArrayList<Integer>();
```

Java 7 also allows the inference of types in the constructor. So the same line of code can be written with less typing:

```
ArrayList<Integer> myList = new ArrayList<>();
```

You can add things to the end of an ArrayList by the add function and get a specific index with the get function. Here are examples using arrays:

```
myList.add(4);
Integer val = myList.get(3);
```

You can also have a loop over integers and have them unbox to their primitives:

```
for (int val : myList)
{
   // Do stuff with val
}
```

It is also worth noting that all Collections in java have a toString method that behaves similar to Arrays.toString(). To print a collection just pass that collection to println:

```
System.out.println(myList);
```

Iterators

For all Collections in java, you can obtain an iterator over the collection. The iterator for loop shown above is just a quick shorthand for iterating over a collection. Under the hood of this for loop an iterator is present that goes over the list. Sometimes an iterator is still necessary. Usually if you want to make removals to elements in the collection while iterating over the collection.

As an example using our list of integers above, we will write code that removes all even integers from our list:

```
Iterator<Integer> it = myList.iterator();
```

```
while (it.hasNext())
{
   int curValue = it.next();
   if (curValue % 2 == 0)
      it.remove();
}
```

Iterators are available for all Collections in the java Collections library and contain three main methods:

```
it.hasNext();
it.next();
it.remove();
```

The method hasNext returns true if there is any element next in the iterator and false otherwise. The method next returns the next element in this iteration. Remove will remove the last element returned by next. Use iterators when trying to clean up a collection. Note, you can have multiple iterators at a time. Each iterator allows you to keep track of multiple locations in the structure.

Pitfalls for Collections

Java allows for autoboxing of primitives when removing them from collections. Having object duplicates of primitives such as Integer for the int primitive or Long for the long primitive creates several hard to catch bugs in contest programming.

For example, consider the following code:

```
ArrayDeque<Integer> deque = new ArrayDeque<Integer>();
// Insert code filling the deque here
while (deque.size() > 1 && deque.peekFirst() == deque.peekLast())
{
      deque.pollFirst();
      deque.pollLast();
}
```

So the above code tries to loop through a deque while there are at least two elements and checks to see if the front of the deque is equal to the back. If so it tries to repeatedly remove such duplicate start and ends.

Unfortunately, the code does not work as intended. The operator == in java will compare the object reference values of the two integers in the deque. If the objects are not the two exact same objects then the code may not work as intended.

So this seems like a fairly harmless bug but has caught many Java contestants. The reason this bug is so hard to catch is java catches Integer objects for all integers in the range [-128, 127] for performance reasons. This means any Integer object in this range will be exactly the same object! This is due to those objects occurring frequently. This is bad news for testing because many contestants will only make cases using small integer values. This also means the wrong answer will not be caught by testing easily.

Here are two fixes to the code to make it work as intended:

```
ArrayDeque<Integer> deque = new ArrayDeque<Integer>();
```

```
// Insert code filling the deque here
while (deque.size() > 1 && deque.peekFirst().equals(deque.peekLast()))
{
      deque.pollFirst();
      deque.pollLast();
}
```

```
ArrayDeque<Integer> deque = new ArrayDeque<Integer>();
// Insert code filling the deque here
while             (deque.size()             >             1             &&
       deque.peekFirst().intValue() == deque.peekLast().intValue())
{
      deque.pollFirst();
      deque.pollLast();
}
```

Another common pitfall is the overloaded remove operator in ArrayList. This usually creates problems for ArrayLists of Integer type:

```
myList.remove(7);
myList.remove(new Integer(7));
```

These two lines of code do different things. The first line of code will remove the integer at index 7 in the list. The second line of code will remove the first occurrence of an integer equal to 7 in the list. This can create tricky bugs to track down in java.

Collections

The Collections class, similar to the Arrays class, contains a handful of handy functions for dealing with all types of collections.

Here are a few of my favorites:

```
Collections.sort(myCollection);
Collections.shuffle(myCollection);
Collections.reverse(myCollection);
```

The sort function behaves in a similar manner to the

Collections also has a prebuild comparator for reversing the natural order of any object relative to its compareTo method. You can pass this comparator to any function that takes in a comparator, even Arrays.sort!

```
Collections.sort(myList, Collections.reverseOrder());
Arrays.sort(myArray, Collections.reverseOrder());
```

This is a useful trick for sorting arrays in reverse order.

ArrayDeque

This data structure in java is useful in implementing Stacks, Queues and Deques. Though these data structures are very simple in concept they are incredibly useful in a large number of problems.

As ArrayDeque implements three different structures it has many methods that do similar things.

Collection methods

```
container.size()     // Return the size of this deque
container.isEmpty()  // Return true if this container has elements
container.toString() // Return a string representation of this deque
```

Stack methods

```
stk.push(val);           // Add value to top of stack
int val = stk.pop();     // Remove and return top of the stack
int val = stk.peek();    // Return the top of the stack w/o removal
```

Queue methods

```
q.offer(val);       // Add value to the end of the queue
q.poll();           // Remove and return the first element
q.peek();           // Return the first element w/o removal
```

Deque methods

```
dq.addFirst(val);      // Add to the front of the deque
dq.addLast(val);                // Add to the end of the deque
dq.pollFirst();        // Remove from the front of the deque
dq.pollLast();         // Remove from the back of the deque
dq.peekFirst();        // Return the first element w/o removal
dq.peekLast();         // Return the last element w/o removal
```

As there is overlap for the functionality it is worth mentioning, **push** does the same as **addFirst**. The method **offer** does the same thing as **addLast**. If you are trying to be explicit and not confuse yourself it is probably best to use the addFirst and addLast methods and their counterparts if you are mixing add types. If you are just using the deque as a stack or queue, using the named versions of these functions will make conceptual sense on their own.

PriorityQueue

A priority queue in java is implemented using a data structure called a heap. This lecture will not explain how a heap works but you can read up on that structure on your own time. Instead, we will explain the methods a priority queue contains and the order runtime of each method.

The purpose of the priority queue is to provide quick access to the smallest element in the queue based on some sorting. It works similar to a regular queue except additions with lower priority values are handled first.

For the most part in a priority queue you will be working with four main methods:

```
q.add(newValue);
q.poll();
q.peek();
q.size();
```

Add

The add method for a priority queue behaves in $O(\log n)$ runtime. The n in this runtime is the number of elements in the priority queue at the time of addition. Here log means log base 2. If you plug 1,000,000 in to the log function of your calculator, you will see this is not a large number of steps. This makes priority queues a nice choice for speeding up problems where you need to know either the largest or smallest element you would like to process next.

The purpose of this method is to add an element to the container. Note that the runtime of $O(\log n)$ is worse than the runtime $O(1)$. Essentally we are paying for the fast lookup of the smallest element by making adding elements slower. Sometimes this is a worthy tradeoff.

Poll

This method takes the smallest element in the queue based on each objects compareTo method and removes it from the queue. The method call takes $O(\log n)$ time as well as the structure must rearrange itself to figure out the next smallest element.

Peek

This method allows you access to the smallest element in the queue without removing it. It works the same as ArrayDeque's peek method and even runs in $O(1)$ time!

Example 1

If you want to try implementing this problem, you can find the problem description here on codeforces:

http://codeforces.com/contest/523/problem/D

In this problem we have **m** processors and **n** tasks, you will assign the first available processor to the first task as they appear. For each task, you want to print out the time the task will be completed.

Solution

```
// Read in the number of tasks and the number of processors
int n = in.nextInt();
int m = in.nextInt();

// First add all processors in the time they are available
PriorityQueue<Long> q = new PriorityQueue<>();
for (int i=0; i<m; i++)
   q.add(0L);

// For each task determine the next available process and print
// when the task will be completed.
for (int i=0; i<n; i++)
{
   int timeArrived = in.nextInt();
   int taskDuration = in.nextInt();
   long nextAvailable = Math.max(timeArrived, q.poll());
   long completion = taskDuration + nextAvailable;
   System.out.println(completion);
   q.add(completion);
}
```

Example 2

In this problem you want to read in a list of values. After reading in each value print the median of the list so far. There can be up to 10^5 values.

How can we use PriorityQueues to efficiently solve this problem?

Sets

Before learning about the set data structure, it is prudent to explore the concept of sets in mathematics. This way you understand the mathematical collection from which this data structure originates.

In mathematics, sets are unordered collections without duplicates. Objects that are contained inside sets are called elements of that set.

When writing out sets mathematically we can use the following notation (called the roster method):

$$\{apple, orange, peaches, grapes\}$$
$$\{1, 2, 3, 4, 5\}$$

Notice that the two following sets are equal under our definition:

$$\{1, 3, 2\} = \{2, 1, 3\}$$

Two sets A and B are equal if the elements in A are all contained in set B and all elements of set B are contained in set A. Also notice that each element may be contained only once in the set. We cannot have the following set:

$$\{orange, orange\}$$

There are some other useful definitions to know about sets. A set A is called a subset of set B if all elements in set A are also in set B. We use the following notation to denote subset:

$$A \subseteq B$$

An alternative definition of set equality is $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

Another piece of notation from set theory is the element of symbol. We use this to denote when one element is contained in a set. So the following is a true statement:

$$apple \in \{apple, peach, orange\}$$

Sets also have a few handy operations:

Set Union

Given two sets A and B, the union of those sets is defined as the set containing all elements in either A or B. Union is denoted $A \cup B$.

$$\{apple, orange, peach\} \cup \{apple, tomato, carrot\} = \{apple, orange, peach, tomato, carrot\}$$

Set Intersection

Given two sets A and B, the intersection of those sets is defined as the set containing all elements in both A and B. Intersection is denoted $A \cap B$.

$$\{apple, orange, peach\} \cap \{apple, tomato, carrot\} = \{apple\}$$

Set Difference

Given two sets A and B, the set difference of those sets is defined as the set containing all elements in A but not in B. Set difference is denoted $A - B$.

$$\{apple, orange, peach\} - \{apple, tomato, carrot\} = \{orange, peach\}$$

Set theory goes much deeper than this but a brief introduction to these concepts will greatly aid in your understanding of not only the data structures below but more advanced concepts based off Set Theory such as Inclusion-Exclusion, the Disjoint Sets data structure, and brute force concepts enumerating sets.

TreeSet

TreeSets are useful is representing ordered sets. Ordered sets are subject to some natural ordering. Each object you add to a TreeSet must implement the Comparable interface including the compareTo method.

Useful methods

```
ts.add(val);     // Adds the specified value to the TreeSet
ts.remove(val); // Removes the specified value from the TreeSet
ts.contains(val); // Returns if the specified value is in the TreeSet
ts.first();     // Returns the smallest element in the TreeSet
ts.last();      // Returns the largest element in the TreeSet
ts.pollFirst(); // Returns and removes the smallest element
ts.pollLast();  // Returns and removes the largest element
ts.lower(val);  // Returns the first element strictly lower than val
```

```
ts.higher(val); // Returns the first element strictly higher than val

ts.floor(val);     // Returns the first value ≤ val
ts.ceiling(val);   // Returns the first value ≥ val

// Keeps all elements shared between ts and other collection
ts.retainAll(otherCollection);

// Remove all elements shared between ts and other collection
ts.removeAll(otherCollection);

// Checks if ts contains all elements of other collection
ts.containsAll(otherCollection);
```

The methods {**add**, **remove**, **pollFirst**, **pollLast**, **lower**, **higher**, **floor**, **ceiling**} all behave in $O(\log n)$ time. The methods **first** and **last** run in $O(1)$ time.

Now that you have learned some of the TreeSet methods, can you figure out how to implement the operations union, intersection and set difference easily? What about properties like subset or element of?

Let's say you have a set of numbers, can you determine the closest number in the set to some query number quickly using the methods above?

HashSets

This class is a way of implementing the unordered form of Sets. HashSet doesn't have as many fancy functions as TreeSet but makes up for it by being faster in most cases.

The concept of hashing is a standard concept taught in college level computer science courses. The idea is to take an object of some kind and encode it into an integer. That integer represents some element in a table (usually implemented by an array). This is a great idea until you end up with two objects with the same hash value. Such an occurrence is called a collision. If you are interested in this concept further, including how collisions are resolved, go look it up online. There are many resources on how this concept is implemented.

In order to go into a hash set the object must override both the **hashcode** and **equals** method from object. The hashcode generate an index for our object and equals must tests if two objects of the same type are indeed perfectly equal. (Equals is used to resolve collisions) If the hashing function is good enough, then collections will occur infrequently yielding $O(1)$ performance. If you use Integer, Long, String, or most build in java classes, these functions will already be implemented for you.

HashSets are a nice alternative if you don't want to sort the data you are hashing. They are also a nice choice for strings as many of the comparisons only happen on the **hashcode** and not the **equals** method.

Useful methods

```
hs.add(val);     // Adds the specified value to the HashSet
hs.remove(val); // Removes the specified value from the HashSet
hs.contains(val); // Returns if the specified value is in the HashSet
```

Maps

Similar to TreeSet and HashSet there are two types of Maps in java that are useful, TreeMap and HashMap. These two types of data structures, Sets and Maps, are very much related.

Maps are like more general arrays. They take in an object called a key and spit out a value. Unlike arrays, the key can be any object supported by the map.

Using generics for defining a map is a little different as you must specify both the key and the value's type for the map. Here is example usage for TreeMap and HashMap in java:

```
TreeMap<String, Integer> map1 = new TreeMap<String, Integer>();
HashMap<String, Integer> map2 = new HashMap<String, Integer>();
```

In Java 7 and later, we can use the same shortcut to cut out some of that typing:

```
TreeMap<String, Integer> map1 = new TreeMap<>();
HashMap<String, Integer> map2 = new HashMap<>();
```

In this example, our key is a String and it maps to an Integer. So we could use map as a way of mapping someone's last name to their age. There is a helpful method to aid in that mapping called **put**.

```
map.put("John", 16);
```

In this example, we take the string john and assign the value 16 to that position in the map. If we use array syntax (not allowed by Java) the code would look something like this:

```
map["John"] = 16
```

You can also retrieve the mapping using the **get** method:

```
int value = map.get("John");
```

At the end of this call, value would contain 16.

Maps serve as a way of creating associations between two types of data.

The difference between TreeMap and HashMap are similar to the differences between TreeSet and HashSet. Many of the same functions that exist for the Set version also exist for maps operating on the keys. (For example, **lower()** becomes **lowerKey()**) So you can take advantage of many of the same benefits of the set structures in the maps.