Data Structures – Brett Bernstein

Lecture 10: BitSets and Packages

Exercises

1. Suppose you are using a HashMap<String, Integer>. What is the runtime of put and get assuming you have a reasonable hash function?

2. Suppose you make a Java HashMap that uses ArrayLists as the keys. What happens if you add an element to one of the ArrayLists currently being used as a key?

3. Suppose you need to be able to translate user logins to emails and vice-versa. Assume both user logins and emails are Strings that are unique across the organization. What can be done to allow lookups in both directions to be done in \( \Theta(1) \)?

4. Given an ArrayList of some type of objects, show how to create a new list without the duplicate entries.

5. Give a simple efficient implementation of a Set whose possible values will be between 0 and 30. Below we have included the Set ADT:

```java
//Stores elements but disregards duplicates (according to .equals).
public interface Set<T>
{
    //Adds t to the set
    void add(T t);
    //Determines if t is in the set
    boolean contains(T t);
    //Removes t from the set if it is the set.
    //Otherwise does nothing.
    void remove(T t);
    //Returns the number of elements in the set
    int size();
}
```

Solutions

1. Expected \( \Theta(1) \) running time. This signifies that we will usually have a small number of collisions, so we will compare our key to a small number of entries. A more accurate runtime would be \( \Theta(L) \) where \( L \) is the length of our key. This indicates that even though we will make a small number of key comparisons, each comparison will take \( \Theta(L) \) time. In other words, we have not assumed equals is \( \Theta(1) \) and have incorporated it into our runtime.
2. The hashCo de of the ArrayList will probably change, and thus it will likely be in the wrong bucket list. In light of this, never modify an ArrayList that is currently being used as a key in a hashtable. This can be made a little easier by using Collections.unmodifiableList (look it up in the Java API if you are curious).

3. Make two HashMaps taking Strings to Strings. The first maps logins into emails. The second maps emails into logins. When ever you add a new user, you put an entry in both maps.

4. We can stick all of our elements into a HashSet and then build our new list. Code follows:

```java
import java.util.ArrayList;
import java.util.HashSet;

public class RemoveDuplicates {

    public static ArrayList<String> removeDups(
        ArrayList<String> list)
    {
        HashSet<String> set = new HashSet<>();
        for (String s : list) set.add(s);
        ArrayList<String> ret = new ArrayList<>();
        for (String s : set) ret.add(s);
        return ret;
    }

    public static <T> ArrayList<T> removeDups2(
        ArrayList<T> list)
    {
        HashSet<T> set = new HashSet<>();
        set.addAll(list);
        ArrayList<T> ret = new ArrayList<>();
        ret.addAll(set);
        return ret;
    }

    public static <T> ArrayList<T> removeDups3(
        ArrayList<T> list)
    {
        //HashSet<T> set = new HashSet<>((list);
        //return new ArrayList<>((set);
        return new ArrayList<>(new HashSet<>(list));
    }
}
```
The method removeDups implements a String version of what we described above. In
removeDups2 we use a generic parameter on the method to allow for any type, and we
use the method addAll to remove some code. In removeDups3 we remove the addAll
by using a constructor which does the same thing in one step.

5. Here we can just use the implementation from Lab 3: a boolean array. Each entry of
the array corresponds to a value between 0 and 30. If the $i^{th}$ entry is true, then the
corresponding element is in the set, otherwise it isn’t.

**Binary Representation of Numbers**

Given any non-negative integer we can represent it in base 2 (i.e., binary):

\[
\begin{align*}
5 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101_2 \\
9 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1001_2 \\
15 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1111_2,
\end{align*}
\]

where the subscript 2 just means we are writing the number in base 2. Typically we write
numbers in base 10 (decimal):

\[
237 = 2 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0 = 237_{10}.
\]

In Java we can write integers in binary by prefacing them with 0b:

5 == 0b101
9 == 0b1001
15 == 0b1111

For Java ints (which are 32-bit values), all representable non-negative integers are stored
in the lowest 31 bits (in base 2) with the highest bit fixed at 0. Negative numbers will
all have the highest bit set to 1. To handle negative numbers the computer uses the two’s
complement representation: to negate a number we flip all of the bits and then add 1. For
example, the representation of 1 is 00…001. To get the two’s complement representation
of $-1$ we first flip to get 11…110 and then add 1 to get 11…111. If we repeat this process
we get 1 again. Since 5 is 00…0101 then $-5$ is 11…1011. Although the two’s complement
formula may seem unintuitive at first, it actually makes a lot of sense. Consider the value
$-1$ we looked at above. What happens when you add 1 to it? As we go through the addition
bit by bit we will keep getting a 0 and carrying a 1. The final carried 1 is lost and we obtain
the value 0. In other words, by simple binary addition we have $-1 + 1 = 0$. That is, all of the
standard arithmetic operations can be applied to numbers represented in two’s complement
and they yield the correct results.

The long data type is just like an int, but it uses 64-bits. Thus the non-negative values
will use the lowest 63-bits with the highest bit 0, and the negatives will have the highest
bit set to 1. If you have a long or int and want to see its representation in bits you can use
Integer.toBinaryString and Long.toBinaryString.
Bitwise Operations

In addition to the usual arithmetic operations, Java supports bitwise operations that work on the individual bits of the values. These are bitwise complement (~), bitwise-and (&), bitwise-or (|), and bitwise-xor (^). There is also shift left (<<), signed shift right (>>) and unsigned shift right (>>>). Each of these operations work separately on each bit, so knowing how they work on single bit inputs shows you how they work in general. Complement a bit means to flip it, so the bitwise complement flips all of the bits of a number. Letting $p$ and $q$ denote bits, the following table shows how the operations work bitwise.

| p | q | ~p | p&q | p|q | p^q |
|---|---|----|-----|----|----|
| 0 | 0 | 1  | 0   | 0  | 0  |
| 0 | 1 | 1  | 0   | 1  | 1  |
| 1 | 0 | 0  | 0   | 1  | 1  |
| 1 | 1 | 0  | 1   | 1  | 0  |

Acting on ints, the bitwise operators work on every bit simultaneously. For example,

\[
\begin{align*}
5 \text{ & } 6 &= 0b101 \text{ & } 0b110 = 0b100 = 4 \\
5 \text{ | } 6 &= 0b101 \text{ | } 0b110 = 0b111 = 7 \\
5 \text{ } \wedge \text{ } 6 &= 0b101 \wedge 0b110 = 0b011 = 3 \\
-5 &= \neg 0b101 = 0b111...010 = -6 \\
-5+1 &= \neg 0b101 + 1 = 0b111...011 = -5
\end{align*}
\]

The shift operators work as you would expect. If you write $x<<k$ it will shift the bits of $x$ to the left by $k$ bits. To do this it shifts in $k$ zero bits on the right (lowest order) side. If you write $x>>>(k$ it shifts $x$ to the right by $k$ bits by shifting in $k$ zero bits on the left side. If you write $x>>k$ then you shift to the right by $k$ bits, but the bits you shift in on the left will match the highest order bit (i.e., preserving the sign of the number). For example,

\[
\begin{align*}
5 \text{ } << \text{ } 2 &= 0b101 \text{ } << \text{ } 2 = 0b10100 \\
5 \text{ } >> \text{ } 2 &= 0b101 \text{ } >> \text{ } 2 = 1 \\
5 \text{ } >>> \text{ } 2 &= 0b101 \text{ } >>> \text{ } 2 = 1 \\
-5 \text{ } << \text{ } 2 &= 0b11...1011 \text{ } << \text{ } 2 = 0b11...101100 \\
-5 \text{ } >> \text{ } 2 &= 0b11...1011 \text{ } >> \text{ } 2 = 0b11...1011 \\
-5 \text{ } >>> \text{ } 2 &= 0b11...1011 \text{ } >>> \text{ } 2 = 0b0011...10
\end{align*}
\]

Binary Exercises

1. Write the 32-bit binary representations of the following values: 37, 15, -37, -15.

2. Compute the following expressions (assuming 32-bit ints):

\[
37 \text{ & } 15, 37 \text{ | } 15, 37 \text{ } \wedge \text{ } 15, -37
\]

3. What ints will be equal to their negative?
4. What operation is equivalent to left shifting by \( k \) bits? What operation is equivalent to signed right shifting by \( k \) bits (i.e., \( x >>= 3 \))?  

5. Let \( x \) be an arbitrary integer. What are the values of the following expressions?  
   
   (a) \( x^x \)  
   (b) \( x << 3 \)  
   (c) \( x & 0 \)  
   (d) \( x & 1 \)  

6. What is the int value \( y \) so that \( x & y == x \) for any int \( x \)?

**Binary Solutions**

1.  
   
   \[
   \begin{align*}
   37 & = 0b00...0100101 \\
   15 & = 0b00...0011111 \\
   -37 & = 0b11...1011011 \\
   -15 & = 0b11...1110001 \\
   
   \end{align*}
   \]

2.  
   
   \[
   \begin{align*}
   37 & \& 15 = 0b00...0000101 = 5 \\
   37 & | 15 = 0b00...0101111 = 47 \\
   37 & ^ 15 = 0b00...01011010 = 42 \\
   \neg 37 & = 0b11...1011010 = -38 \\
   \end{align*}
   \]

3. 0 and \(-2^{31}\).

4. Left shifting by \( k \) bits is the same as multiplying by \( 2^k \). Right shifting is equivalent to dividing by \( 2^k \). This is only true since we used \( >> \) instead of \( >>> \). If a number is negative and we use the unsigned right shift then it isn’t equivalent to division (it turns out to be equivalent to division if we treat the integer as unsigned, but we won’t get into this).

5. (a) 0  
   (b) \( 8x \)  
   (c) 0  
   (d) 1 if \( x \) is odd and 0 if \( x \) is even.

6. \( y = -1 \), i.e., every bit 1.
BitSets (also called Bitmasks)

Our goal is to improve upon our earlier data structure for storing subsets of the integers $0, 1, \ldots, N$ where $N \leq 63$. To do this we will use a single long variable where each bit represents one of the numbers. For instance, the long $5L$ (the L makes the number have type long) which has binary representation $101_2$ will represent the set $\{0, 2\}$. If the bit in position $i$ is 1, then we will say $i$ is in the set. If the bit in position $i$ is 0, then we will say $i$ is not in the set. For this reason we will often want to obtain information about the bit in position $i$ of a given long. It will be helpful to use the quantity $(1L<<i)$ which is entirely zero bits except for a 1 in position $i$ (i.e., the $(i+1)$th bit). For example, $(1L<<3)==0b1000$.

BitSet Exercises

1. What does $x^\sim(1<<3)$ do to the long variable $x$?

2. Assume the long variables $x, y$ represent some subsets of $0, 1, \ldots, 63$ using the convention that the $i$th bit determines whether the subset contains $i$. Show how to implement the following methods in Java code:
   
   (a) Determine if the set (corresponding to $x$) contains the number $i$.
   (b) Add $i$ to the set (corresponding to $x$), or do nothing if it already contains it.
   (c) Remove $i$ from the set (corresponding to $x$) if it contains it, otherwise do nothing.
   (d) Union the two sets corresponding to $x$ and $y$.
   (e) Intersect the two sets corresponding to $x$ and $y$.
   (f) Remove all elements of the set corresponding to $y$ from the set corresponding to $x$ (ignore the elements from $y$ but not in $x$).

3. Let the long variable $x$ represent a subset of $0, 1, \ldots, 63$. Show how to count the number of elements in the set.

4. Let $n = 20$. Suppose there is a function `void process(int bitset)` which performs the required work on the given subset of $0, 1, \ldots, n$. Show how to call process on all subsets of $0, 1, \ldots, n$.

5. How can we make a bitset for the values $0, \ldots, N$ where $N > 63$?

BitSet Solutions

1. Flips the 4th bit (i.e., the bit in position 3).

2. (a) $(x \& (1L<<i)) != 0$
   (b) $x = (x | (1L<<i))$ or we could write $x |= 1L<<i$
   (c) $x = x \& (\neg(1L<<i))$ or $x &= \neg(1L<<i)$
(d) $x \mid y$
(e) $x \& y$
(f) $x \& (\neg y)$

3. Below we give some different implementations:

```
import java.util.Random;

public class CountBits {
    public static int countOnes(long x) {
        int ret = 0;
        for (int i = 0; i < 64; ++i) {
            long b = 1L << i;
            if ((x & b) != 0) ret++;
        }
        return ret;
    }
    public static int countOnes2(long x) {
        int ret = 0;
        while (x != 0) {
            if ((x & 1) != 0) ret++;
            x = (x>>>1); //Must use >>>
            //Could also write x >>>= 1;
        }
        return ret;
    }
    public static int countOnes3(long x) {
        int ret = 0;
        while (x != 0) {
            ret++;
            x = x&(x-1); //Sneakily zeros out lowest 1
        }
        return ret;
    }
    public static int countOnes4(long x) {
        return Long.bitCount(x);
    }

    public static void main(String[] args) {
        long x = new Random().nextLong();
        System.out.println(x);
    }
}
```
System.out.println(Long.toBinaryString(x));
System.out.println(countOnes(x));
System.out.println(countOnes2(x));
System.out.println(countOnes3(x));
System.out.println(countOnes4(x));
}
}

The first two are fairly standard. The only trick with the second is that if you don’t use $\gg\gg$ it will loop infinitely on negative numbers. The third one uses the sneaky operation $x \& (x-1)$ that zeros out the lowest bit that is 1. This reduces the loop to only iterate $k$ times, where $k$ is the number of 1’s (instead of 64 times like the other loops could). The last one shows that the Java library has this operation.

4. \texttt{for} (\texttt{int} i = 0; i < (1<<21); ++i) \texttt{process(i);} \\

5. Use an array of longs where each corresponds to 64 values in the range. This is already implemented for you in the Java class BitSet.

\section*{Java Packages}

Packages are used to organize code into bundles that can be treated as separate units. Putting your code in a package in Eclipse is easy. You just right-click on your project and create a new package. At the top of each of your source files will be the line \texttt{package PackageName;} For example, if you look at the top of the Java ArrayList class you will see the line \texttt{package java.util;} A period in a package name indicates subpackaging. Thus ArrayList belongs to the subpackage util which belongs to the package java. On the file system, each subpackage corresponds to a different folder. Thus code in the package brett.ds2016.util will be in the folder brett/ds2016/util/ off of my project’s src folder in the workspace. Up to this point in the class we have never created our own packages. Thus all of our code is in the src folder directly in what is called the default (unnamed) package. By always creating new projects for each lab or assignment we have separated our code neatly, but code in one project cannot use code in another project. Thus every project is on an island. If instead we made a single project with multiple packages then the code would be separated, but would have access to each other via import statements.

Now that we know what packages are, we can finish learning about access modifiers. To obtain access to the public classes in a different package we must use an import statement at the top of our code. You have already done this countless times in your programs. If you declare a class but you do not mark it as public it has package access. This means that the only code that has access to it lies in the same package. The same goes for variables and methods (static or instance) that you declare in a class. If you don’t specify an access modifier then they have package access and can only be accessed within the same package.
If you mark a field or method as protected then it is accessible within the same package, or in any descendant class (possibly in another package).