CSCI 136
Data Structures & Advanced Programming

Spring 2020

Instructors: Bill & Dan

Administrative Details

- Lab 6: Two Towers
  - Lab handout has detailed instructions
  - Please Collaborate!
  - See lab-specific videos for more details

Last Time

- Iterators
  - General purpose mechanism for traversals
- `Iterator` interface (Java)
- `AbstractIterator` class (structure5)
  - Adds `get()` and `reset()`

Today's Outline

- Brainstorm some “nifty” Iterators
- Bit operations
  - Useful for data structures in general, but
    - …necessary for Lab 6
- `BIterator.java`: an iterator for enumerating the individual bits in the binary representation of an Integer

Some “Nifty” Iterators

- Iterators aren’t limited to simply traversing the elements of a data structure
    - We can define arbitrary “traversals”
    - We can “generate” elements that are not stored anywhere in memory
- An iterator just gives us an interface that we can fill in however we want!

ReverseIterator.java

- Goal:
  - Take an iterator `it` and return its values in the opposite order that it yields them
- Implementation:
  - Problem: Iterators progress in one direction only
    - `next()` but no `previous()`
  - Any ideas?

SkipIterator.java

- Goal:
  - Take an iterator `it` and a value `val`
  - Return sequential values from `it` as long as they don't match `val`
- Implementation:
  - `next()` and `hasNext()`
    - Pre-calculate the values in preparation for the `next()` call
  - What if last value in `it` is equal to `val`?

Biterator.java

- Goal:
  - Take a number `n`, and yield its bits (0 or 1) from least significant bit to most significant bit
- Implementation:
  - Think back to Lab 3
- We will revisit this at the end of lecture, after covering bit operations

Representing Numbers

- Humans usually think of numbers in base 10
- But even though we write `int x = 23;` the computer stores `x` as a sequence of `1`s and `0`s
- Recall Lab 3:

```
public static String numInBinary(int n) {
        if (n <= 1)
            return "" + n%2;

        return printInBinary(n/2) + n%2;
}
```

- 00000000 00000000 00000000 00010111

numInBinary(int n)

- What was our strategy for writing (recursive) `printInBinary` for Lab 3?
  - Use mod to isolate the least significant bit
  - Divide by 2 and recurse

```
public static String numInBinary(int n) {
        if (n <= 1)
            return "" + n%2;

        return printInBinary(n/2) + n%2;
}
```

Bitwise Operations

- We can use *bitwise* operations to manipulate the `1`s and `0`s in the binary representation
  - Bitwise 'and': `&`
  - Bitwise 'or': `|`
- Also useful: bit shifts
  - Bit shift left: <<
  - Bit shift right: >>

& and |

- Given two integers `a` and `b`, the *bitwise or* expression `a | b` returns an integer s.t.
  - At each bit position, the result has a `1` if that bit position had a `1` in EITHER `a` OR `b`
  - `3 | 6 = ?` 011 | 110 = 111
- Given two integers `a` and `b`, the *bitwise and* expression `a & b` returns an integer s.t.
  - At each bit position, the result has a `1` if that bit position had a `1` in BOTH `a` AND `b`
  - `3 & 6 = ?` 011 & 110 = 010

>> and <<

- Given two integers `a` and `i`, the expression `(a << i)` returns $(a * 2^{i})$
  - Why? It shifts all bits left by `i` positions
  - `1 << 4 = ?` 00001 << 4 = 10000
- Given two integers `a` and `i`, the expression `(a >> i)` returns $(a / 2^{i})$
  - Why? It shifts all bits right by `i` positions
  - `1 >> 4 = ?` 00001 >> 4 = 00000
  - `97 >> 3 = ?` (97 = 1100001)
    1100001 >> 3 = 1100
- Be careful about shifting left and "overflow"!!!

Revisiting numInBinary(int n)

- How would we rewrite a recursive `numInBinary` using bit shifts and bitwise operations?

```
public static String numInBinary(int n) {
    if (n <= 1) // no non-zero digits
        return "" + n;
    return numInBinary(n >> 1) + (n & 1);
}
```

Revisiting numInBinary(int n)

- How would we write an iterative `printInBinary` using bit shifts and bitwise operations?

```
public static String printInBinary(int n,
                                   int width) {
    String result = "";
    for(int i = 0; i < width; i++)
        if ((n & (1<<i)) == 0)
            result = 0 + result;
        else
            result = 1 + result;
    return result;
}
```

BIterator.java

- Goal:
  - Take a number `n`, and yield its bits (0 or 1) from least significant bit to most significant bit
- Implementation:
  - Store `n`
  - Each `next()` isolates the LSB and shifts
  - `hasNext()`?
  - `reset()`?

General Rules for Iterators

1. Understand order of data structure
2. **Always call hasNext() before calling next()!!!**
3. Use remove with caution!
4. Don't add to structure while iterating:
   see `TestIterator.java`

Up Next

- Two Towers Lab
  - Use the bitwise representation of a Java Long to represent a "subset" of blocks
  - Iterate through all possible subsets of blocks to find the subset that optimizes the problem
    - Find the "most even" stacking