PFAFB: pointers and references

- In Java everything is a pointer or a reference
  - What does this mean for parameters?
  - Mean for for Map<String,ArrayList<String>>

- How do you implement HashMap (Hashtable)?
  - Properties of keys?
  - Comparable v. Hashable
- What about LinkedList?
  - O(1) add to front, how?
  - Inefficient find n°, why?
- Use versus build

What is a pointer?

- An indirect reference, not the thing, pointer-to-thing
  - What is a forwarding address when you move?
  - What’s a sign that says Gone Fishin’?

ArrayList<Color> x = new ArrayList<Color>();
ArrayList<Color> y = x;
// y = new ArrayList<Color>();
x.add(new Color(255,255,0));
// y.add(x.get(0));
y.get(0).makeBlue();

- An indirect reference

Links and Pointers and Recursion

- Map<String,ArrayList<String>> map;
  - What does this do? map.get("fruit").add("apple");
  - Different from this code:
- Understand linked lists from the bottom up
  - Not as clients of java.util.LinkedList
  - Using linked lists to implement different structures
  - Using linked lists to leverage algorithmic improvements
- Self-referential structures and recursion
  - Why recursion works well with linked-structures
- Setting up the DNA-linked-list assignment

Review Map Contents

- Map<String,ArrayList<String>> map;
  - What does this do? map.get("fruit").add("apple");
  - Different from this code:
    - ArrayList<String> list = map.get("fruit");
    - list.add("apple");
  - Different from this code?
    - ArrayList<String> list = map.get("fruit");
    - list.add("apple");
    - map.put("fruit",list);
- Map<String, Integer> map;
  - map.put("word", map.get("word")+1);
What will we do with Pointers?

- Why assignment = to parameter have no affect?
  - What about `param.changeMe()`?
  - What about "change-and-return"?

- Study LinkedList and linked lists from basics
  - Useful to understand C, C++
  - Useful in understanding trees
  - Required in other courses, interviews, etc.
  - Low-level abstraction, high-order abstraction

- Linked lists and Trees: similar but different

Foundations for [Hash|Tree] [Set|Map]

- Typically linked lists used to implement hash tables
  - List of frames for film: clip and insert without shifting
  - Nodes that link to each other, not contiguous in memory
  - Self-referential, indirect references, confusing?

- Why use linked lists?
  - Insert and remove without shifting, add element in constant time, e.g., O(1) add to back
    - Contrast to ArrayList which can double in size
  - Master pointers and indirection
  - Leads to trees and graphs: structure data into information

Linked lists as recombinant DNA

- Splice three GTGATAATTC strands into DNA
  - Use strings: length of result is N + 3*10
  - Generalize to N + B*S (# breaks x size-of-splice)

- We can use linked lists instead
  - Use same GTGATAATTC if strands are immutable
  - Generalize to N+ S + B, is this an improvement?

Getting in front

- Suppose we want to add a new element
  - At the back of a string or an ArrayList or a ...
  - At the front of a string or an ArrayList or a ...
  - Is there a difference? Why? What's complexity?

- Suppose this is an important problem: we want to grow at the front (and perhaps at the back)
  - Think editing film clips and film splicing
  - Think DNA and gene splicing

- Self-referential data structures to the rescue
  - References, reference problems, recursion, binky
Goldilocks and the Hashtable

- A hashtable is a collection of buckets
  - Find the right bucket and search it
  - Bucket organization?
    - Array, linked list, search tree

How do we compare times? Methods?

Structuring Data: The inside story

- How does a hashtable work? ArrayListHash.java
  - What happens with put(key, value) in a HashMap?
  - What happens with get(key)?
  - What happens with remove(key)?

Contrast LinkedList and ArrayList

- ISimpleList, SimpleLinkedList, SimpleArrayList
  - Meant to illustrate concepts, not industrial-strength
  - Very similar to industrial-strength, however

- ArrayList -- why is access O(1) or constant time?
  - Storage in memory is contiguous, all elements same size
  - Where is the 1st element? 40th? 360th?
  - Doesn’t matter what's in the ArrayList, everything is a pointer or a reference (what about null?)
Linked lists, CDT and ADT

- As an ADT
  - A list is empty, or contains an element and a list
  - ( ) or (x, (y, ( ) ) )

- As a picture

- CDT (concrete data type) pojo: plain old Java object

  ```java
  public class Node{
      Node p = new Node();
      String value;
      Node next;
  }
  ```

What about LinkedList?

- Why is access of Nth element linear time?
  - Keep pointer to last, does that help?

- Why is adding to front constant-time O(1)?

ArrayLists and linked lists as ADTs

- As an ADT (abstract data type) ArrayLists support
  - Constant-time or O(1) access to the k-th element
  - Amortized linear or O(n) storage/time with add
    - Total storage is 2n, why? (for n elements)
    - Add in middle (or where…) is "expensive", what and why?

- Linked lists as ADT
  - Constant-time or O(1) insertion/deletion anywhere, but...
    - Linear or O(n) time to find where, sequential search

- Good for sparse structures: when data are scarce, allocate exactly as many list elements as needed, no wasted space/copying (e.g., what happens when ArrayList grows?)

Linked list applications continued

- If programming in C, there are no “growable-arrays”, so typically linked lists used when # elements in a collection varies, isn’t known, can’t be fixed at compile time
  - Could grow array, potentially expensive/wasteful especially if # elements is small.
  - Also need # elements in array, requires extra parameter
  - With linked list, one pointer accesses all elements

- Simulation/modeling of DNA gene-splicing
  - Given list of millions of CGTA... for DNA strand, find locations where new DNA/gene can be spliced in
    - Remove target sequence, insert new sequence
Building linked lists

- Add words to the front of a list (draw a picture)
  - Create new node with next pointing to list, reset start of list

```
public class Node {
    String value;
    Node next;
    Node(String s, Node link){
        value = s;
        next = link;
    }
}
```

// … declarations here

```
Node list = null;
while (scanner.hasNext()) {
    list = new Node(scanner.next(), list);
}
```

- What about adding to the end of the list?

Dissection of add-to-front

- List initially empty
- First node has first word

```
list = new Node(word, list);
```

- Each new word causes new node to be created
- New node added to front
- Rhs of operator = completely evaluated before assignment

Standard list processing (iterative)

- Visit all nodes once, e.g., count them or process them

```
public int size(Node list){
    int count = 0;
    while (list != null) {
        count++;
        list = list.next;
    }
    return count;
}
```

- What changes if we generalize meaning of process?
  - Print nodes?
  - Append "s" to all strings in list?

Building linked lists continued

- What about adding a node to the end of the list?
  - Can we search and find the end?
  - If we do this every time, what’s complexity of building an N-node list? Why?

- Alternatively, keep pointers to first and last nodes
  - If we add node to end, which pointer changes?
  - What about initially empty list: values of pointers?
    - Will lead to consideration of header node to avoid special cases in writing code

- What does code to remove a node look like?
  - First node
  - Other nodes (why different?)