Plan for the Week

- **Review Big-Oh**
  - General concepts for analytical analyses
  - Examples of "loop-counting"

- **Introduction to Linked Lists**
  - Building blocks for data structures, prelude to trees, so-called "self-referential", but not

- **Markov, APTs, Fall Break, Midterm**
  - Resources and Practice

Concrete Implementation: Linked List

Pointers, References, Structures

- **Why assignment = to parameter have no affect?**
  - What about `param.changeMe()`?
  - What about "change-and-return"?
  - No change at all? Toward Java 8/Functional

- **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

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 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

Structuring Data: The inside story

- How does a HashSet work? SimpleHashStringSet, almost the same as HashMap
  - What happens with \textit{add} (key) in a HashSet?
  - What happens with \textit{contains} (key)?
  - What happens with \textit{remove} (key)?

- In diagram below, what's in each cell of myTable?
  - ArrayList: advantages? Disadvantages?

Set Implementations

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Unique</th>
<th>Array</th>
<th>Util.hash</th>
<th>HashArray</th>
<th>HashLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Melville</td>
<td>14353</td>
<td>4103</td>
<td>0.43</td>
<td>0.15</td>
<td>0.216</td>
<td>0.104</td>
</tr>
<tr>
<td>hawthorne</td>
<td>85753</td>
<td>13542</td>
<td>1.84</td>
<td>0.21</td>
<td>0.288</td>
<td>0.188</td>
</tr>
<tr>
<td>kjv10</td>
<td>823135</td>
<td>32674</td>
<td>14.77</td>
<td>0.71</td>
<td>0.558</td>
<td>0.584</td>
</tr>
</tbody>
</table>

- Array: search entire array for each add
- Class java.util.HashSet
- HashArray: buckets are ArrayList objects
- HashLink: buckets are low-level linked lists

Set Implementations, SetStress.java

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Util.hash</th>
<th>HashArray</th>
<th>HashLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>0.857</td>
<td>0.037</td>
<td>0.023</td>
<td>0.020</td>
</tr>
<tr>
<td>20,000</td>
<td>3.384</td>
<td>0.015</td>
<td>0.014</td>
<td>0.010</td>
</tr>
<tr>
<td>30,000</td>
<td>6.884</td>
<td>0.024</td>
<td>0.016</td>
<td>0.024</td>
</tr>
<tr>
<td>40,000</td>
<td>14.833</td>
<td>0.012</td>
<td>0.030</td>
<td>0.019</td>
</tr>
</tbody>
</table>

- Can we run without edit/recompile/run cycle?
  - Benefits? Drawbacks?
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{
    String value;
    Node next;
    Node p = new Node();
    p.value = "hello";
    p.next = null;
}
```

What about LinkedList?

- Why is access of N\textsuperscript{th} 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 front or middle is "expensive", what???
- 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,

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, can't just pass array, need size!!
  - With linked list, one pointer accesses all elements
Building linked lists

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

```java
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

```java
Node(String s, Node link) {
    info = s;
    next = link;
}
```

- 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

```java
public int size(Node list) {
    int count = 0;
    while (list != null) {
        count += 1;
        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

- Special cases: empty list (null) one node list
Removing Node from list, "cat"

```
public Node remove(Node list, String s) {
    Node start = list;
    while (list.next != null) {
        if (list.value.equals(s)) {
            list.next = list.next.next;
            break;
        }
        list = list.next;
    }
    return start;
}
```

List Questions

Why is the parameter in remove method Object and not String?

Linked List idioms

- Sometimes check list == null and list.next == null
  - Short-circuit evaluation in how to do this?
- First node can be tricky to process, e.g., remove
  - Has no node before it.
  - Solution: put a "header" or "empty" node first
- Typically loop: while(list != null)
  - Can be useful to do while (list.next != null)
  - Must be sure list != null in writing this!!!