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
Some Vocabulary

- **Abstract Data Type or ADT**
  - Treating a structure through an API or interface rather than via its implementation
  - Examples with ArrayList, HashSet, and more

- **Concrete Data Type or CDT**
  - The implementation that conforms to an interface, or realizes an ADT

- **ADT is, in some ways, old terminology**
  - API and Interfaces are more current
ArrayList and LinkedList as ADTs

● As an ADT (abstract data type) ArrayList supports
  ➢ Constant-time or O(1) access to the k-th element
  ➢ Amortized linear or O(n) storage/time with add
    • Total storage used in n-element vector is approx. 2n, spread over all accesses/additions (why?)
  ➢ Add/remove in middle is "expensive" O(n), why?

● What's underneath here? How Implemented?
  ➢ Concrete: array – contiguous memory, must be contiguous to support random access
  ➢ Element 20 = beginning + 20 x size of a pointer
ArrayList and LinkedList as ADTs

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

- **Linked good for add/remove at front**
  - Splicing into middle, also for 'sparse' structures

- **What's underneath? How Implemented**
  - Low-level linked lists, self-referential structures
  - More memory intensive than array: two pointers
Remove Middle in Pictures

- **Find middle element:** happens instantly or \( O(1) \)
  - \( \text{alist(location)} + \frac{n}{2} \times \text{sizeof(pointer)} \) since \( \text{ArrayList} \) holds pointers

- **Shifting requires moving \( \frac{n}{2} \) pointers, but they are all contiguous in memory:** *cache performance*

```
for(int k=middle; ...
    a[k] = alist[k+1]
```
Remove Middle in Pictures

- Find middle element: have to follow pointers between elements
  - Follow $n/2$ pointers, but all over memory, so takes time to move from memory->cache->use
- Removing middle: instantaneous, no shifting, just re-assign a couple of pointers (back pointers too)
  - Blue points to Yellow
Using O-notation as specifications

● **Linked-list is O(1) to insert/delete, O(n) to find, in an n-element list**
  - What does this mean? Does it require a specific implementation?
  - O-notation is typically worst-case upper-bound

● **Hashing is O(1) to insert/delete/find**
  - This isn't worse case, worst case is terrible!
  - Doesn't this sort of depend on N? Memory?
More on O-notation, big-Oh

- **O-notation** is an upper-bound, this means that $N$ is $O(N)$, but it is also $O(N^2)$; we try to provide *tight* bounds. Formally:
  - A function $g(N)$ is $O(f(N))$ if there exist constants $c$ and $n$ such that $g(N) < cf(N)$ for all $N > n$
Notations for measuring complexity

- **O-notation/big-Oh**: $O(n^2)$ is used in algorithmic analysis, e.g., Compsci 330 at Duke. Upper bound in the limit
  - Correct to say that linear algorithm is $O(n^2)$, but useful?

- **Omega is lower bound**: $\Omega(n \log n)$ is a lower bound for comparison based sorts
  - Can't do better than that, a little hard to prove
  - We can still engineer good sorts: TimSort!
Simple examples of array/loops: O?

```java
for(int k=0; k < list.length; k += 1) {
    list[k] += 1;  // list.set(k, list.get(k)+1);
}
//-----

for(int k=0; k < list.length; k += 1)
    for(int j=k+1; j < list.length; j += 1)
        if (list[j].equals(list[k]))
            matches += 1;
//--

for(int k=0; k < list.length; k += 1)
    for(int j=k+1; j < list.length; j *= 2)
        value += 1;
```
Loops explained

- Let N be the # elements in list
  - Loop iterates N times
  - Each time does $O(1)$ work – not dependent on N
- Complexity of code or runtime analysis is: $O(N)$

```java
for(int k=0; k < list.length; k += 1) {
    list[k] += 1;
}
```
Loops explained II

- **Let N be the # elements in list**
  - Outer loop iterates N times
  - Each time does the work of the inner loop
- **Inner loop statement is O(1), the inner loop iterates exactly N-(k+1) times, so inner most statement:**
  - \((N-1) + (N-2) + \ldots + 2 + 1 = O(N^2)\)

```java
for(int k=0; k < list.length; k += 1)
    for(int j=k+1; j < list.length; j += 1)
        if (list[j].equals(list[k]))
            matches += 1;
```
Loops explained III

- Let \( N \) be the \# elements in list
  - Outer loop iterates \( N \) times
  - Each time does the work of the inner loop
- Inner loop statement is \( O(1) \), the inner loop iterates exactly \( \log_2(N-(k+1)) \) times
  - \( \log_2(N) \) * \( N \) is an upper bound, \( O(N \log N) \)
  - \( \log(N-1) + \log(N-2) + \ldots + \log(1) = \log((N-1)!) = \)
    - \( O(N \log N) \)
- http://stackoverflow.com/questions/2095395/is-logn-%CE%98n-logn

```java
for(int k=0; k < list.length; k += 1)
    for(int j=k+1; j < list.length; j *= 2)
        value += 1;
```
Big-O questions

http://bit.ly/201fall16-sept30-1

- How do check work? Look for patterns? Use concrete values for N
Multiplying and adding big-Oh

- Suppose we do a linear search then do another one
  - What is the complexity? $O(n) + O(n)$
  - If we do 100 linear searches? $100 \cdot O(n)$
  - If we do $n$ searches on an array of size $n$? $n \cdot O(n)$

- Binary search followed by linear search?
  - What are big-Oh complexities? Sum?
  - What about 50 binary searches? What about $n$ searches?
What is big-Oh about?

- **Intuition:** avoid details when they don’t matter, and they don’t matter when input size (N) is big enough
  - Use only leading term, ignore coefficients
    
    \[
    y = 3x \\
    y = 6x - 2 \\
    y = 15x + 44 \\
    y = x^2 \\
    y = x^2 - 6x + 9 \\
    y = 3x^2 + 4x
    \]

- **The first family is O(n), the second is O(n^2)**
  - Intuition: family of curves, generally the same shape
  - Intuition: linear function: double input, double time, quadratic function: double input, quadruple the time
Some helpful mathematics

● \(1 + 2 + 3 + 4 + \ldots + N\)
  \(\frac{N(N+1)}{2},\) exactly \(= \frac{N^2}{2} + \frac{N}{2}\) which is \(O(N^2)\) why?

● \(N + N + N + \ldots + N\) (total of \(N\) times)
  \(N^2\) which is \(O(N^2)\)
  Adding \(N+ \ldots + N\), \(2N\) or \(3N\) or \(\ldots\) \(22N\) times? Still \(O(N^2)\)

● \(1 + 2 + 4 + \ldots + 2^N\)
  \(2^{N+1} - 1 = 2 \times 2^N - 1\) which is \(O(2^N)\) – in terms of last term, call it \(X\), this is \(O(X)\)

● \(O(1) + O(2) + \ldots + O(N-1) + O(N) = O(1 \times 2 \times \ldots (N-1) \times N)\)
  \(O(N!)\) which is \(O(N \log N)\)
Grace Murray Hopper (1906-1992)

- “third programmer on world's first large-scale digital computer”
  - US Navy: Admiral, YouTube!

“It's better to show that something can be done and apologize for not asking permission, than to try to persuade the powers that be at the beginning”

- ACM Hopper award given for contributions before 35
  2004: Jennifer Rexford
  2010: Craig Gentry: [http://www.youtube.com/watch?v=qe-zmHoPW30](http://www.youtube.com/watch?v=qe-zmHoPW30)
  2011: Luis von Ahn
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 `add(key)` in a HashSet?
  - What happens with `contains(key)`?
  - What happens with `remove(key)`?

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

### SetDriver.java

<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

[https://git.cs.duke.edu/201fall16/building-arrays/tree/master](https://git.cs.duke.edu/201fall16/building-arrays/tree/master)
## 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?**

[https://git.cs.duke.edu/201fall16/building-arrays/tree/master](https://git.cs.duke.edu/201fall16/building-arrays/tree/master)
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**
  
  ![Linked list diagram]

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

```java
public class Node{
    String value;
    Node next;
}
```

```java
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;
    }
}
```

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
list = new Node(word, list);
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;
}
```
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!!!
Link Questions


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