Data Structures - Session 2, July 5

- Homework 1 is due at 11:59 on Wednesday. It will be submitted via zip file on Canvas.
- Midterm will be on Monday of the fourth week. That week's homework will be due the following Wednesday.
- Homework 5 will be given somewhat less time to complete, since it will be smaller.
- TA office hours: Tuesdays from 7pm to 9pm, Wednesdays from 1 to 3, Thursdays from 7pm to 9pm, and Fridays from 6 to 8pm.

Big-O

- Big-O allows the abstraction regardless of hardware or types of data that the algorithm is being run on
- Runtime analysis example:

  ```
  int sum = 0;
  for (int i =0;i<a.length;i++)
  sum=+ a[i];
  ```

  - declaring the variable sum counts as an operation
  - as does the initialization of i = 0
  - the comparison of i to a.length
  - i++ is another operation
  - these operations are all constants compared to the variable aspect of the cost
- $T(n) = O(f(n))$ such that there exist some constants c & n_o where $T(n) \leqslant cf(n)$ where $n \leqslant n_o$
- Big-O provides an upper bound complexity estimate
- Big omega definition: $T(n) = \Omega(g(n))$ if there exists some constants c & no such that T(n) >= cg(n) where n >= no
- Big theta definition: $T(n) = \Theta(h(n))$ if
- $T(n) = O(h(n))$ &
- $T(n) = \Omega(h(n))$

Linear search example:

- The best case scenario would have the searched element at the beginning of the list, giving the algorithm a Big-O(1)
- The worst case scenario would have the searched element at the end of the list, or not in it at all, giving the algorithm a Big-O(n)

- The average scenario would not be Big-O $(n/2)$, as the constant would be 1/2.

Relative speeds

- constant c would be the fastest costs
- $logn$ would grow very slowly and be the next fastest
- log^2n, log^3n, so on
- linear time algorithms n
- $nlogn$, n^2logn, etc
- n^2, n^3, so on
- exponential algorithms, 2^n
- $n!$ is the slowest/greatest cost

1. If we have two separate algorithms, $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$
   - we can say that $T_1(n) + T_2(n) = O(f(n) + g(n))$
   - $T_1(n) * T_2(n) = O(f(n) * g(n))$ - an algorithmic example of this is nested for loops
2. If $T(n)$ is a polynomial of degree k, then:
   - $T(n) = \Theta(n^k)$
3. $log^k(n) = O(n)$

Example

```
for (i=0;i<n;i++)
  for (j=i;j<nj++)
      x++;
```

- the first iteration of the loop will have n, second will have (n - 1), etc
- the total sum would be n(n + 1)/2
- the sum evaluated would be n^2/2 + n /2, which would equal Big-O(n^2)

If-statement Example

```
if(condition) {
    alg1;
} else {
    alg2;
}
```

- in the worst case scenario, it will execute the most expensive algorithm between the 2. Sometimes, the frequency with which either algorithm will occur is known, and using the most expensive algorithm everytime will not be correct

Abstract Data Type (ADT)

- an abstract data type in Java would closely correspond to an interface

Collections

- a Collection is an abstract data type
- extending an interface includes the methods in that interface into the subclass interface
- extending Iterable interface will include the hasNext(), next(), and remove()

Lists

- a list is an abstract data type as a grouping, or ordering, of elements. In a list, we might want to be able to get(i), set(i), add, or remove. These are all intrinsic behaviors of a list.
- a List should have all of the attributes of a Collection, so the List interface extends the Collection interface
- List interface includes add/removal methods by index, as well as get/set methods by indices
- A ListIterator allows the user to move forward and backward, since there is ordering with a list. They also have the Iterators under the Collection interface.

ArrayList

- implements the List interface, with a size variable
- the size variable is incremented whenever adding an element to the ArrayList
- we create an array of an initial size with "empty" variables in it
- searching for an element in an ArrayList would always be Big-O(1), but inserting is always an expensive operation (especially if inserting at the beginning of the ArrayList)

Singly linked lists

- a singly linked list is built out of node objects
- a linked list node is a class that defines two fields: data and next pointer `java LLNode<AnyType> {     AnyType data;     LLNode<AnyType> next; }`
- the next field is a reference to another linked list node, therefore the linked list node points to the next linked list node
- having a reference to the first node will provide a reference to every other node in the linked list
- placing an object at the end of the list would be Big-O(n), since we only have the reference to the first object
- placing an object at the front of the list would just be Big-O(1)
- searching or setting would have a worst case scenario of Big-O(n)
- inserting into a linked list during interations would have a Big-O(1) based on where the iterator is

Doubly linked lists

- also has a 'previous' field, adding an extra piece of data that each node must have

- Since you can move in both directions, there is a reference to both the first and last elements of the list
- the first and last nodes of a doubly linked list don't actually store data (as sentinel nodes) and are the only references stores
- addAfter Example

  ```
  addAfter(k)
  t = new Node(30); // new node
  t.prev = k; // redirects previous reference from t
  t.next = k.next; // redirects next node from t to k's forward reference
  k.next = t; // redirects k's forward reference to t
  t.next.prev = t; // redirects end node's previous reference to t
  ```