Section 5 Overview

Mini Assignment:

- A is connected, acyclic (like a tree!)
    - connected: we can't easily tell from the matrix without hardcore math, since you could have two nodes connected to each other but disconnected from another portion of the graph.
    - acyclic: can't easily tell from the matrix without hardcore math
- B is unconnected, cyclic
    - unconnected: can easily tell from the matrix because there's a row and column with all Fs
    - cyclic: we can see at least one cycle pretty easily, 3 is connected to itself, however still a good idea to draw it out because there could be other cycles that are not immediately obvious

Tree Induction

- Go over binary tree induction from HW5

Graphs

Given vertices v1 and v2 in a undirected Graph G, compute whether there exists a path between vertices.

```
existsPath(v1, v2):
      Q = [] //Queue
      v1.visited = true
      Q.enqueue(v1)
      while Q not empty:
            cur = Q.dequeue()
            if cur == v2:
                  return true
            for neighbor in cur's adjacent nodes:
                  if not neighbor.visited:
                        neighbor.visited = true
                        Q.enqueue(neighbor)
      return false
```

- Runtime: O(|E| + |V|)
- BFS is more efficient than DFS -- BFS guarantees that first time you reach a node is the *shortest* path to that node because you're expanding outward from the source in levels.

Cycles

Given a singly linked list, return true if there is a cycle (return false otherwise).

What if you weren't allowed to store additional information for each node (i.e. space-complexity constraints)?

Solutions involving decorations will work and run in O(n), but fail the memory constraint.

The method this uses is having a runner (hare), which is a common method for solving interview questions about linked lists.

```
public boolean isCycle (Node head){
      if (head == null) {
            return false;
      }
      Node tortoise = head;
      Node hare = head.next;
      while (hare != null){
            if (tortoise == hare) {
                        return true;
            }

            if (hare.next == null) {
                        return false;
            }

            hare = hare.next.next;
            tortoise= tortoise.next;
      }
      return false;
}
```

Generics

Why use generics?

1. By providing types as a parameter, there is stronger type checking at compile-time. Fixing errors at compile time is easier than fixing errors at run-time, in which an error may occur anywhere within the lifecycle of the program
2. Eliminate Casts (example below)
3. Creating generic algorithms - algorithms can be used on collections of types

```
List list = new ArrayList();
list.add("hello");
Benefits:
String s = (String) list.get(0);
```

vs

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```
// This is the only acceptable way to do it. Don't do the first way.

Example:

```
public class Box {
      private Object _object;
      // constructor elided
      public void set(Object object) {
            _object = object;
      }
      public Object get() {
            return _object;
      }
}
```

Generic Box:

```
/**
 * Generic version of the Box class.
 * @param <T>     the type of the value being boxed
 */
public class Box<T> {
      // T stands for "Type"
      private T _t;
      // constructor elided
      public void set(T t) {
            _t = t;
      }
      public T get() {
            return _t;
      }
}
```

How would you instantiate something of type Box?

```
Box<Integer> box1 = new Box<Integer>(); // this Box holds Integers
Box<String> box2 = new Box<String>(); // this Box holds Strings
```

- Examples in Heap:
    - MyLinkedHeapTree<E> uses Position<E>
    - MyHeap creates a MyLinkedHeapTree<MyHeapEntry<K,V>>
    - MyHeapEntry has a reference to its position in a variable of type Position<MyHeapEntry<K,V>>
- So what's the point?
    - Generics allow us to have one set of code that works for any type and guarantees that the different classes' types match
    - In the visualizer we use integers/strings, but the point of making our heap is to allow it to be used in the future, and therefore we don't want to restrict the types that our heap can be used by.

Selection Review

- No new information here, check out the Selection lecture on the website for a review!

Optional Problems:

Runner Reinforcement:
Given a linked list of unknown length, find the midpoint of the list without counting how many nodes there are the list then traversing to the middle--that would be the naive solution and too boring.

Can assume that if the length of the list is even, then either of the "middle" nodes is OK to return.i

```
def findMiddle(node head):
      node runner = head;
      node current = head;

      while runner != null and runner.next != null:
            runner = runner.next.next
            current = current.next

      return current
```

Is a graph 2-colorable?

A graph is k-colorable if every node can be assigned one of k colors such that no node has a neighbor that is the same color as itself.

```
//all nodes in g are initially uncolored
//assume g is connected
isTwoColorable(graph g):
      let v be a random vertex in g
      Q = [ ] //Queue
      v.color = A
      Q.enqueue(v)
      while Q not empty:
            vertex cur = Q.dequeue;
            for all edges e of cur:
                  w = node on opposite side of e
                  if cur and w are the same color
                        return false
                  if w is uncolored
                        give w the opposite color of cur
                        Q.enqueue(w)
      return true
```