

COS 226 Data Structures and Algorithms
Spring 2016 - Flipped Lecture Handout

Week 2 Flipped Activities

1. Understanding Linked Lists and Array Implementation of Stacks

- (a) What are the advantages and disadvantages of using a linked list vs array when implementing a stack?
- (b) Discuss the exact memory requirements for implementing a stack of N nodes using an array and implementing the same with a linked list. Assume that each node takes a total of M bytes.

2. Amortized Analysis

- (a) What is the best, worst and amortized cost of push and pop operations in a resizing array using doubling principle.
- (b) Suppose array resizing is done by increasing the array size by 2 when the array is full. What is the average cost of M push operations?

3. Algorithm Design

Inversions. Design a subquadratic algorithm that counts the number of inversions in an array.

4. Cycles in Linked Lists

Design an algorithm that can detect a cycle in a linked list. You can only use a constant amount of extra memory and the algorithm must run in linear time.

5. Sorting with Just two keys

- (a) Device a sorting algorithm that runs in linear time, if the array only has two distinct keys.

- (b) Formulate and validate hypothesis about the running time of insertion and selection sort for arrays that contain just two key values, assuming that they are equally likely to occur.

6. **Queue using Circular Linked List** Write a queue (of Strings) implementation that uses a circular linked list (i.e. `last.next = first`). Show the implementation of the methods `enqueue` and `dequeue`.

```
private class Node
{
    String item;
    Node next;
}
private Node first, last;
void enqueue(String item);\
String dequeue();\
```

7. Randomized Queues and Deques Assignment

Write a generic data type for a deque and a randomized queue. The goal of this assignment is to implement elementary data structures using arrays and linked lists, and to introduce you to generics and iterators.

Deque. A double-ended queue or deque (pronounced "deck") is a generalization of a stack and a queue that supports adding and removing items from either the front or the back of the data structure. Create a generic data type Deque that implements the following API:

```
public class Deque<Item> implements Iterable<Item> {
    public Deque()                // construct an empty deque
    public boolean isEmpty()       // is the deque empty?
    public int size()              // return the number of items on the deque
    public void addFirst(Item item) // add the item to the front
    public void addLast(Item item)  // add the item to the end
    public Item removeFirst()        // remove and return the item from the front
    public Item removeLast()        // remove and return the item from the end
    public Iterator<Item> iterator() // return an iterator over items in order from front to end
    public static void main(String[] args) // unit testing (required)
}
```

Performance requirements. Your deque implementation must support each deque operation (including construction) in constant worst-case time and use space linear in the number of items currently in the deque. Additionally, your iterator implementation must support each operation (including construction) in constant worst-case time.

Randomized queue. A randomized queue is similar to a stack or queue, except

that the item removed is chosen uniformly at random from items in the data structure. Create a generic data type RandomizedQueue that implements the following API:

```
public class RandomizedQueue<Item> implements Iterable<Item> {
    public RandomizedQueue() // construct an empty randomized queue
    public boolean isEmpty()  // is the queue empty?
    public int size()         // return the number of items on the queue
    public void enqueue(Item item) // add the item
    public Item dequeue()        // remove and return a random item
    public Item sample()         // return a random item (but do not remove it)
    public Iterator<Item> iterator() // return an independent iterator over items in random order
    public static void main(String[] args) // unit testing (required)
}
```

Performance requirements. Your randomized queue implementation must support each randomized queue operation (besides creating an iterator) in constant amortized time and use space linear in the number of items currently in the queue. That is, any sequence of M randomized queue operations (starting from an empty queue) must take at most cM steps in the worst case, for some constant c . Additionally, your iterator implementation must support `next()` and `hasNext()` in constant worst-case time and construction in linear time; you may use a linear amount of extra memory per iterator.