

1. (a) A **linked list** provides flexible allocation of memory using smaller blocks of nodes. The arrays require allocation of a contiguous block of memory. The linked list requires the extra 8 bytes to hold a reference to each node. LL's are also prone to programmer errors if LL is not manipulated correctly.  
 (b) An **array implementation of a stack** of Nodes of size M each =  $24 + N \cdot M$ . The LL implementation requires  $(8(\text{item}) + 8(\text{next}) + 16(\text{obj})) = 32$  bytes for each node (assuming node is not an inner class. Otherwise add 4). So we need  $32N + N \cdot M$  (this is for the actual objects referenced by the Item)
  
2. **Amortized Cost** (a) best case – stack has enough space –  $O(1)$   
 worst case – stack has no space. Need to double the space and write all the elements –  $O(N)$   
 Average case –  $O(1)$  can show this as amortized cost  
  
 (b) Assume the array size is  $i$  when needed to resize by 2. We need to find the cost of doing  $M$  operations. Suppose when the array size is  $i = 2K$  (since we are incrementing by 2, it is always a multiple of 2). The biggest cost is when the array goes from  $i = 2k$  to  $i = 2K+1$ . You will have to copy the old array of size  $i = 2K$  into the new array. This will have to be done  $M/2$  times. So we have  $\text{Sum}(2k, k=1, 2, \dots, M/2) \sim \frac{1}{4} M^2$ . hence the average cost of  $M$  operations is  $M^2/M = M$
  
3. **Algorithm Design: Counting Inversions.** Divide and conquer. Suppose we have an array of size  $N$ . Break the array into two arrays of size  $N/2$ . Say we have  $A = [A1 \ A2]$ . Suppose you want to know the number of inversions between  $A1$  and  $A2$ . If we sort  $A1$  (say in  $N \log N$  time), We can find the inversions between  $A1$  and  $A2$  in linear time. So we write a recursive equation to describe our algorithm. Let  $T(N)$  be the cost to find inversions in an array of size  $N$ . Then we can describe the relation as  $T(N) = 2 T(N/2) + cN$  (some linear cost to find the inversions between the two arrays)  
 By solving  $T(N) = 2 T(N/2) + N$  we get  $T(N) \sim O(N \log N)$
  
4. **Finding Cycles.** In this problem we will assume that we do not know the size of the list (if so, it is easy to find by counting). Now define two pointers,  $\text{ptr1}$  and  $\text{ptr2}$  that startd from first and advance by one node and two nodes respectively.  
 **$\text{ptr1} = \text{ptr1.next}$  and  $\text{ptr2} = \text{ptr2.next.next}$**   
 We will argue that if there is a cycle,  $\text{ptr1}$  will at some point will meet  $\text{ptr2}$  and if there is no cycle,  $\text{ptr1}$  and  $\text{ptr2}$  will never meet ( $\text{ptr2}$  will end sooner). How is this possible? Just intuition is good enough here. No mathematical proof necessary. For those who are

mathematically inclined, Since 1 and 2 are relatively prime, we can find two integers p and q such that,  $1.p + 2.q = 1(\text{mod } N)$ . So we claim no matter when they enter the loop, at some point they will come together at 1 mod N (1 mod N is when a number is divided by N, the remainder is 1).

eg: suppose the cycle length is 3.

ptr1 will hit the following sequence: 1,2,3,**1**,2,3,**1**,2,3

ptr2 will hit the following sequence: 1 3 2 **1** 3 2 **1** 3 2

**Follow up question:** What is we decide to send ptr1 by 2 and ptr2 by 3? Will the argument still work?

## 5. Sorting with Just Two Keys

Consider any array with just 2 keys: a b a a b a a b a a a .. b

Have a right pointer to last element and a left pointer to the first element. Consider an algorithm like this. Let's make it easier by knowing what the bigger element is. if the keys are uniformly distributed, you should be able to find this quite fast in constant time. Now here is the algorithm

- i. if right = max , move left by 1
- ii. if left = min, move right by 1
- iii. if right < left, swap(left, right), right—and left++
- iv. if right > left, left++, right--

(b) What happens to insertion and selection sorts if there are only two keys and they are uniformly distributed? (note that we do not modify the selection and insertion sorts. we just need to analyze what happens)

**Selection Sort** : no change in complexity, since we find min and swap, It would still be  $\sim \frac{1}{2} N^2$

**Insertion Sort:** Since keys are equally likely, in each iteration of insertion sort, say you are trying to insert into a sorted array of size i, we can assume that half the A's are in the beginning of sorted part and the other half are B's. So if you are inserting an A, you will need to do no more than  $i/2$  work and if you are inserting a B, you will do no work. Assuming half of the unsorted is A and other half is B (uniformly distributed assumption), we will do  $i/2$  work for  $(n-i)/2$  A elements, and 1 work for  $(n-i)/2$  , B elements. So we have a total of =  $\text{sum}( i/2*(n-i)/2 + 1*(n-i)/2)$  work to do. Left as an

exercise (do not worry since this is all additional stuff, that is not required to pass the course)