**Homework 2**

1. Swap two adjacent elements by adjusting only the links (and not the data) using
2. Singly linked list
3. Doubly linked list

For singly linked lists:

// beforeP is the cell before the two adjacent cells that are to be swapped.

// Error checks are omitted for clarity.

public static void swapWithNext( Node beforep )

{

Node p, afterp;

p = beforep.next;

afterp = p.next; // Both p and afterp assumed not null.

p.next = afterp.next;

beforep.next = afterp;

afterp.next = p;

}

**(b)** For doubly linked lists:

// p and afterp are cells to be switched. Error checks as before.

public static void swapWithNext( Node p )

{

Node beforep, afterp;

beforep = p.prev;

afterp = p.next;

p.next = afterp.next;

beforep.next = afterp;

afterp.next = p;

p.next.prev = p;

p.prev = afterp;

afterp.prev = beforep;

}

1. Implement the contains routine for MyLinkedList

public boolean contains( AnyType x )

{

Node<AnyType> p = beginMarker.next;

while( p != endMarker && !(p.data.equals(x) ))

{

p = p.next;

}

return (p != endMarker);

}

1. The Josephus problem is the following game: N people, numbered 1 to N , are sitting in a circle. Starting at person 1, a hot potato is passed, After M passes, the person holding the potato is eliminated, the circle closes ranks, and the game continues with the person who was sitting after the eliminated person picking up the hot potato. The last remaining person wins. Thus if M =0 and N=5, players are eliminated in order and player 5 wins. If M=1 and N=5, the prder of elimination is 2,4,1,5.
2. Write a program to solve the Josephus problem for general values of M and N. Try to make your program as efficient as possible. Make sure you dispose of cells.

[7 Points]

1. What is the running time of your program?

[3 Points]

**Using ArrayList:**

A basic algorithm is to iterate through the list, removing every *M*th item. This can be improved by two observations. The first is that an item *M* distance away is the same as an item that is only *M* mod *N* away. This is useful when *M* is large enough to cause iteration through the list multiple times. The second is that an item *M* distance away in the forward direction is the same as an item (*M – N*) away in the backwards direction. This improvement is useful when *M* is more than *N*/2 (half the list). The solution shown below uses these two observations. Note that the list size changes as items are removed. The worst case running time is *O*(*N\*N*min(*M*, *N*)), though with the improvements given the algorithm might be significantly faster. If *M* = 1, the algorithm is quadratic (mainly due to each remove taking linear time).

public static void pass(int m, int n)

{

int i, j, mPrime, numLeft;

ArrayList<Integer> L = new ArrayList<Integer>();

for (i=1; i<=n; i++)

L.add(i);

ListIterator<Integer> iter = L.listIterator();

Integer item=0;

numLeft = n;

mPrime = m % n;

for (i=0; i<n; i++)

{

mPrime = m % numleft;

if (mPrime <= numLeft/2)

{

if (iter.hasNext())

item = iter.next();

for (j=0; j<mPrime; j++)

{

if (!iter.hasNext())

iter = L.listIterator();

item = iter.next();

}

}

else

{

for (j=0; j<numLeft-mPrime; j++)

{

if (!iter.hasPrevious())

iter = L.listIterator(L.size());

item = iter.previous();

}

}

System.out.print("Removed " + item + " ");

iter.remove();

if (!iter.hasNext())

iter = L.listIterator();

System.out.println();

for (Integer x:L)

System.out.print(x + " ");

System.out.println();

numLeft--;

}

System.out.println();

}

**Using SinglyLinkedList**

Using singly linked list without iterator not a circular list, the pointer ptr goes to first node each time it reaches the end of the list Running time O(N\* M) clearly due to 2 for loops, each remove is constant time.

public static void pass\_LinkedList(int m, int n){

//Create a empty singly linkedlist.

singlyLinkedList L = new singlyLinkedList(); // has a pointer head which is initially null and eventually points to the first node

for (int i=1; i<=n; i++)

L.insert(i); // adds element i to the end of the list

singlyLinkedList.Node ptr = L.head; // Initialize a pointer to the head.

for(int i=1;i<=n;i++){

for(int j=1;j<m;j++) {

if(ptr.next !=null)

ptr=ptr.next;

//skip all way the way to m-1 th element

else

ptr = L.head; // Back to head if end of the list reached

}//end of j loop

if(ptr.next!=null)

{

System.out.println("\nRemoved " + ptr.next.data);

ptr.next = ptr.next.next;

//we have essentially unreferenced mth element.

ptr=ptr.next;

if(ptr==null)

ptr=L.head; //to avoid null reference in the next round

}

else

{

System.out.println("\nRemoved " + L.head.data);

L.head=L.head.next;

// we have essentially unreferenced mth element.

ptr=L.head;

}

}//End of i loop

}

1. What is the running time of the following code?

public static List<Integer> makeList( int N)

{

ArrayList<Integer> lst = new ArrayList<>();

for( inti =0; i < N; i++)

{

lst.add(i);

lst.trimToSize();

}

}

*O*(*N*2). The trim method reduces the size of the array, requiring each add to resize it. The resize takes *O*(*N*) time, and there are *O*(*N*) calls.

1. The following routine removes the first half of the list passed as a parameter:

public static void removeFirstHalf(List<?> lst)

{

int theSize = lst.size() /2

for( inti =0; i < theSize; i++ )

lst.remove(0);

}

1. Why is theSize saved prior to entering the for loop?
2. What is the running time of removeFirstHalf if lst is an ArrayList?
3. What is the running time of removeFirstHalf if lst is a LinkedLIst?
4. Does using an iterator make removeFirstHalf faster for either type of List

**(a)** Because the remove call changes the size, which would affect the loop.

**(b)** *O*(*N*2). Each remove from the beginning requires moving all elements forward, which takes *O*(*N*) time.

**(c)** *O*(*N*). Each remove can be done in *O*(1) time.

**(d)** No. Since it is always the first element being removed, finding the position does not take time, thus an iterator would not help.

1. Write a function in pseudocode named removeDuplicates(), which takes a singly linked list sorted in increasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60. ( delete the node inside the method do not use remove\_node routine)

**Input:**  
You have to complete the method which takes **1** argument: the **head** of the  linked list  .   
**Output:**  
Your function should return a pointer to the head of linked list with no duplicate element.

[10 Points]

Node removeDuplicates(Node head)  
{

Node current = head;  
 while(current != null && current.next != null){  
 if(current.next.data == [current.data](http://disq.us/url?url=http%3A%2F%2Fcurrent.data%3AIFnhU4EOF-kXB6e2gKUxenRFluQ&cuid=3952990" \o "current.data))

[current.next](http://disq.us/url?url=http%3A%2F%2Ffastptr.next%3AUhdsQlZdOtgzinboHi214mxtSMg&cuid=3952990) = current.next.next

else

current = current.next  
 }

return head;

}