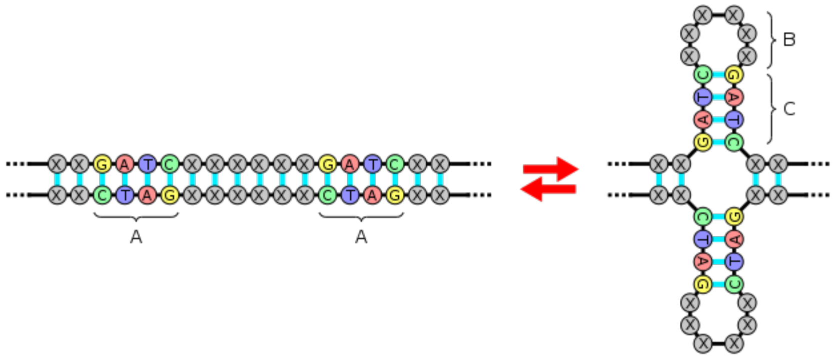
Assignment 2
Genetic Palindromes, Linked Lists

Release Date	Due Date
January 30, 2020	February 13, 2020

Objectives

- Understanding the mechanisms of linked lists in depth
- Sorting linked lists
- Explicitly manipulating pointers and references



Taken from Wikimedia / Wikipedia

Problem Specification

In this assignment you will check to see if a given DNA (or RNA) double-strand is palindromic and search for palindromic strands in a genomic sequence. A palindrome in languages is a word or string that can be read the same way forward or backwards (e.g. racecar, wow, noon, madam, 10101). Palindromic sequences in DNA refer to a short run of bases (typically 3 to 5 in length), followed by their complementary bases in reverse order, we will call these genetic palindromes.

DNA strands can be represented using strings of letters referring to the bases of the strand. There are four common types of bases: adenine (A), cytosine (C), guanine (G), and thymine (T), and there is also uracil (U) in RNA. A and T are complements, A and U are complements, and C and G are complements.

The meaning of palindrome in the context of genetics is slightly different from the definition used for words and sentences. Since a double helix (see the above figure) is formed by two paired antiparallel strands of nucleotides that run in opposite directions, and the nucleotides always pair in the same way (adenine (A) with thymine (T) in DNA or uracil (U) in RNA; cytosine (C) with guanine (G)), a (single-stranded) nucleotide sequence is said to be a palindrome if it is equal to its reverse complement. For example, the DNA sequence ACCTAGGT is palindromic because its nucleotide-by-nucleotide complement is TGGATCCA, and reversing the order of the nucleotides in the complement gives the original sequence. The RNA sequence "UAGCTA" is genetic palindrome because, read backwards, it reads “ATCGAU”, the complement to the original sequence.

Palindromic sequences are important for genetic research, specifically in researching restrictive enzymes. They are also linked with diseases including several cancers, mental retardation, X-linked recessive diseases and many physical abnormalities. Hence, identifying these in a genome sequence is an important task.

Note: In this assignment you are NOT allowed to use Java (or other languages) Collections API such as LinkedList, ArrayList, List, Stack, Queue, etc.

Write an application in Java, Python or C++ to determine whether genomic sequences contain genetic palindromes using your implementation of linked lists as follows:

1) Storing a genome sequence in a string variable, say myGseq, and then searching for genetic palindromes by iterating through the characters of myGseq, may be a good solution that uses array-like random-access manipulations! However…
2) Since sample sizes and genomic sequences can be arbitrarily large, storing them in a string variable may not be feasible. Therefore, we will store them one character (i.e., base) per node in a linked list and manipulate the linked list to identify or search genetic palindromes. In order to get extensive experience with linked lists and pointers/references, you will use your implementation of linked lists following the ideas[1] given after the assignment submission section. Any other methods used will receive zero credit.
3) Your linked list node’s data portion will have at least one attribute, called *nucleotide* of type *char*.
4) Read the DNA / RNA sequences from a text file. One genomic sequence per line. Note that there might be only one strand per line to simplify test cases.
5) Identify all genetic palindromes of length four to seventeen in a given genomic sequence. Store all the identified genetic palindromes in a new linked list using the *Append()* method of linked-list. Separate each genetic palindrome by a node containing the non-base character ‘-‘. Print the data of this linked list.
6) For each genetic palindrome that is found in the file, find how many other similar genetic palindromes exist in the file. Note that a pair of strands are similar when they have the same length and the i^{th} nucleotide of a strand in the pair is either a base or its complement.
7) Test and measure time complexities of your implementations of *isGeneticPalindrome(), findGeneticPalindromes()* and *countSimilarGeneticPalindromes()* and output the results. See sample inputs and outputs.
8) All your functions, including *isGeneticPalindrome(),* should have input validation.
   a. *Example: myLL.storeStrand(“TAT”) would store the characters {T,A,T} in the linked-list.*
   b. *Example: myLL.storeStrand(“hi_mom”) would report an error and gracefully exit or continue processing next input. Your message to mom would never be seen :(*
9) To test *InsertSort() (*and *sortedInsert()),* sort the genomic sequence after storing it in the linked list and print the first and the last genomic sequences in the file and their corresponding sorted sequences.

[1] We have liberally borrowed ideas of the linked lists from Nick Parlante@Stanford and palindromes in genetics from Wikipedia.

Sample Inputs and Outputs

```
Please enter the file name containing DNA/RNA Sequences: mySequences.txt
   .. read mySequences.txt
time to create linked lists from sequences: 20 milliseconds

DNA sequence 1: TATA

“TATA” is a genetic palindrome.
This sequence does not contain any genetic palindromes.

time to test whether a genomic subsequence in Sequence 1 is palindromic: 2 milliseconds
time to find all palindromic subsequences of length [4, 17] in Sequence 1: 5 milliseconds

DNA sequence 2: Lorem Ipsom
“Lorem Ipsom” contains characters that do not denote DNA molecules. Skipped.

DNA sequence 3: TTTACCT

“TTTACCT” is not a palindromic sequence. However, it has the following genetic palindromes
of length > 3: TTACC

time to test whether a genomic subsequence in Sequence 3 is palindromic: 2.5 milliseconds
time to find all palindromic subsequences of length [4, 17] in Sequence 3: 6 milliseconds


“TATA” has 20 similar sequences in the file, and
it took 10 milliseconds to determine that.

“TTACC” has 5 similar sequences in the file, and
it took 11 milliseconds to determine that.
```

Design Requirements

Code Documentation

For this assignment, you must include documentation for your code. This include how to compile and run your program. Also, you **must** give outputs of your program with the following double-stranded DNA: "TAGUCTA”, “CTAGUCTAA”, “TTAAATTCCTTTAGG”, “AGTCCGATCCGT”, and “AAAAAATTTTTTGGGGGGCCCCCC” at the minimum. If you don’t provide outputs for these five inputs, it means your program does not execute properly.

You must also include documentation for your code as generated by JavaDoc (and similar tools if using some other language). You should have JavaDoc comments for every class, constructor, and method. By default, JavaDoc should output html documentation to a subfolder within your project (/dist/javadoc). Make sure this folder is included when you zip your files for submission. You do not need to submit a hard copy of this documentation.

Hint: http://stackoverflow.com/questions/4468669/how-to-generate-javadoc-html-in-eclipse

Coding Conventions and Programming Standards

You must adhere to all conventions in the CS 3310 Java coding standard. This includes the use of white spaces for readability and the use of comments to explain the meaning of various methods and attributes. Be sure to follow the conventions for naming files, classes, variables, method parameters and methods. Read the material linked from our class web-pages (in case you can't recall programming styles and conventions from your CS1 and CS2 courses).

Testing

Make sure you test your application with several different values capturing different cases, to make sure it works. That is test your program not just on the inputs we specified but other representatiove test data that covers your program's logic as well.

Assignment Submission

- Generate a .zip file that contains all your files, including:
    - Signed Plagiarism Declaration
    - Source code files
    - Include any input or output files
    - Documentation of your code – e.g. using Javadoc if using Java
    - A brief report (in a pdf file) on your observations of comparing theoretical vs empirically observed time complexities. Note this report will include (a) a brief description of problem statement(s), (b) algorithms descriptions (if these are standard, commonly known algorithms, then just mention their names along with customization to your specific solution(s), otherwise give the pseudo-code of your algorithms, (c) theoretically derived complexities of the algorithms used in your code, (d) table(s) of the observed time complexities, and (e) plots comparing theoretical vs. empirical along with your observations (e.g. do theoretical agree with your implementation, why? Why not?).
- Don't forget to follow the naming convention specified for submitting assignments

Linked List Specifications using Java-like syntax

Linked List Ground Rules

All of the linked list code in this document uses the "classic" doubly linked list structure: A single head pointer points to the first node in the list. Each node contains a single `next` pointer to the next node and a single `prev` pointer to the previous node. The `next` pointer of the last node is NULL and `prev` pointer of the first node is NULL. The empty list is represented by a NULL head pointer. All of the nodes are allocated in the heap.

You will practice list manipulations using an object-oriented (OO) style of coding, for some functionality we may use non-OO style of coding. Essentially, in OO style – list manipulation functions are implemented as methods of the class and in non-OO style (aka conventional) they are implemented as independent units. So there might be two versions of the functions. We suggest, you first implement the functions using whatever style you are most comfortable with and then copy-paste-modify-test those methods' implementations using the other style. Examples in the exposition below use data type "int" but note that for your assignment it is "char" (the first letter of the base), so make appropriate change or use say "generics" if you are using Java. In fact, an enumerated type may be even better for the data portion of your linked-list.

For a few of the problems, the solutions present the temporary "dummy node" variation (see below), but most of the code deals with linked lists in their plain form. In the text, brackets {} are used to describe lists — the list containing the numbers 1, 2, and 3 is written as {1, 2, 3}. The drawing show singly-linked lists rather than doubly-linked lists for clarity and brevity. The node and list types used are (add any other attributes as needed).

```
Node {char nucleotide; Node prev; Node next}

List {Node head}
```

Basic Utility Functions

Implement following basic utility functions...

- `int Length(Node head); int Length();`
  Returns the number of nodes in the list pointed by head or the current list.

- `List BuildOneTwoThree();`
  Allocates and returns the list {1, 2, 3}. Used by some of the example code to build lists to work on. Note one attribute of List must be `Node head` where head points to the head node of the list.

- `List Push(Node lPtr, char newData); List Push (char newData);` Given a `char` and a reference to the pointer in L (i.e., the attribute head of class List, L.head), add a new node at the head of the list with the standard 3-step-link-in: create the new node, set its `.next` to point to the current head, set L's head field to the new node, and finally return the new list. If lPtr is specified, then push newData at lPtr. (If you are not sure of how this method works, the first few problems may be helpful warm-ups.) This is also implemented as a method of linked list class.

Use of the Basic Utility Functions

This sample code demonstrates the basic utility functions being used.

```
void BasicsCaller()
   { List L;

     int len;


     L = BuildOneTwoThree(); // Start with {1, 2, 3}


   L.Push(13); // using OO-style Push 13 on the front, yielding {13, 1, 2, 3}


   L = L.Push((L.head().next()), 42);   // Push 42 into the second position
                                       // yielding {13, 42, 1, 2, 3}
   len = Length(L.head());             // or L.Length(); Computes that the length is 5.
}
```

Linked List Coding Techniques

The following list presents the most common techniques you may want to use in solving the linked list problems. The first few are basic. The last few are only necessary for the more advanced problems.

1. Iterate Down a List

A very frequent technique in linked list code is to iterate a pointer over all the nodes in a list. Traditionally, this is written as a `while` loop. The head pointer is copied into a local variable `current` which then iterates down the list. Test for the end of the list with `current!=NULL`. Advance the pointer with `current=current.next`. For example, you will need iteration to implement `Length()` function. Alternately, some people prefer to write the loop as a `for` which makes the initialization, test, and pointer advance more centralized, and so harder to omit...

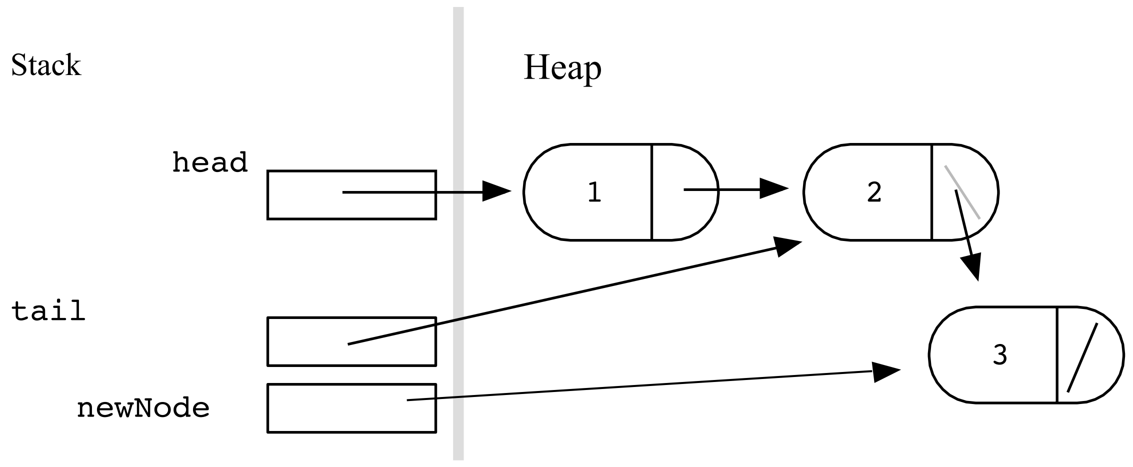
```
  for (current = head; current != NULL; current = current.next) {
…}
```

2. Build — At Head With Push()

The easiest way to build up a list is by adding nodes at its "head end" with Push(). The code is short and it runs fast — lists naturally support operations at their head end. The disadvantage is that the elements will appear in the list in the reverse order that they are added. If you don't care about order, then the head end is the best.

3. Build — With Tail Pointer

What about adding nodes at the "tail end" of the list? Adding a node at the tail of a list most often involves locating the last node in the list, and then changing its `next` field from NULL to point to the new node, such as the `tail` variable in the following example of adding a "3" node to the end of the list {1, 2}...



This is just a special case of the general rule: to insert or delete a node inside a list, you need a pointer to the node just *before* that position, so you can change its `next` field. Many list problems include the sub-problem of advancing a pointer to the node before the point of insertion or deletion. The one exception is if the operation falls on the first node in the list — in that case the head pointer itself must be changed. The following examples show the various ways code can handle the single head case and all the interior cases.

4. Build — Special Case + Tail Pointer

Consider the problem of building up the list {1, 2, 3, 4, 5} by appending the nodes to the tail end. The difficulty is that the very first node must be added at the head pointer, but all the other nodes are inserted after the last node using a tail pointer. The simplest way to deal with both cases is to just have two separate cases in the code. Special case code first adds the head node {1}. Then there is a separate loop that uses a tail pointer to add all the other nodes. The tail pointer is kept pointing at the last node, and each new node is added at `tail.next`. The only "problem" with this solution is that writing separate special case code for the first node is a little unsatisfying. Nonetheless, this approach is a solid one for production code — it is simple and runs fast.

5. Build — Temporary Dummy Node

This is a slightly unusual technique that can be used to shorten the code: Use a temporary dummy node at the head of the list during the computation. The trick is that with the dummy, every node appears to be added after the `.next` field of some other node. That way the code for the first node is the same as for the other nodes. The tail pointer plays the same role as in the previous example. The difference is that now it also handles the first node as well.

Some linked list implementations keep the dummy node as a permanent part of the list. For this "permanent dummy" strategy, the empty list is not represented by a NULL pointer. Instead, every list has a heap allocated dummy node at its head. Algorithms skip over the dummy node for all operations. That way the dummy node is always present to provide the above sort of convenience in the code. Programmers prefer the temporary strategy shown here, but it is a little peculiar since the temporary dummy node is allocated in the stack, while all the other nodes are allocated in the heap. For production code, coders do not use either type of dummy node. The code should just cope with the head node boundary cases.

6. Build — Local References

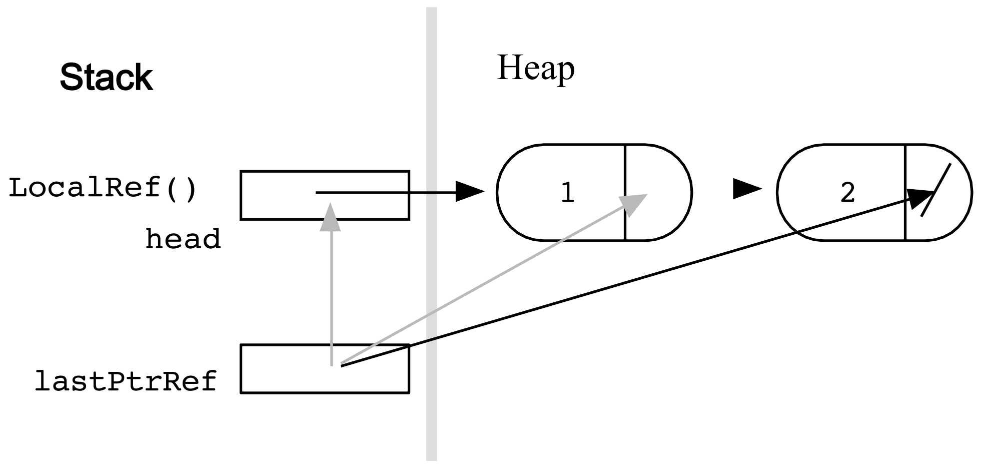
Finally, here is a tricky way to unify all the node cases without using a dummy node at all. For this technique, we use a local "reference pointer" which always points to the last *pointer* in the list instead of to the last node. All additions to the list are made by following the reference pointer. The reference pointer starts off pointing to the head pointer. Later, it points to the `.next` field *inside* the last node in the list. (A detailed explanation follows.) Following uses C syntax instead of Java (in case you did not know, C is good for “low” level programming or programming resource constraint devices).

```c
struct node* BuildWithLocalRef() {
   struct node* head = NULL;
   struct node** lastPtrRef= &head; // Start out pointing to the head pointer
   int i;


   for (i=1; i<6; i++) {
      Push(lastPtrRef, i);    // Add node at the last pointer in the list
      lastPtrRef= &((*lastPtrRef)->next); // Advance to point to the
                                          // new last pointer
   }


   // head == {1, 2, 3, 4, 5};
   return(head);
}
```

This technique is short, but the inside of the loop is scary. This technique is rarely used, but it's a good way to see if you really understand pointers. Here's how it works...

1) At the top of the loop, `lastPtrRef` points to the last *pointer* in the list. Initially it points to the head pointer itself. Later it points to the `.next` field inside the last node in the list.
2) `Push(lastPtrRef, i);` adds a new node at the last pointer. The new node becomes the last node in the list.
3) `lastPtrRef= &((*lastPtrRef)->next);` Advance the `lastPtrRef` to now point to the `.next` field inside the new last node that `.next` field is now the last pointer in the list.

Here is a drawing showing the state of memory for the above code just before the third node is added. The previous values of `lastPtrRef` are shown in gray...



This technique is never required to solve a linked list problem, but it can be one of the alternative solutions for some of the advanced problems. The code is shorter this way, but the performance is probably not any better.

Unusual Techniques

Both the temporary-stack-dummy and the local-reference-pointer techniques are a little unusual. They are cute, and they let us play around with yet another variation in pointer intensive code. They use memory in unusual ways, so they are a nice way to see if you really understand what's going on. However, I probably would not use them in production code.

Linked List Problems

Here are 8 linked list problems arranged in order of difficulty. The first few are quite basic. Each problem starts with a basic definition of what needs to be accomplished. Many of the problems also include hints or drawings to get you started. The drawing show singly-linked lists to keep it simple.

It's easy to just passively sweep your eyes over someone’s solution — verifying its existence without lettings its details touch your brain. To get the most benefit from these problems, you need to make an effort to think them through. Whether or not you solve the problem, you will be thinking through the right issues, and the given solution will make more sense.

Great programmers can visualize data structures to see how the code and memory will interact. Linked lists are well suited to that sort of visual thinking. Use these problems to develop your visualization skill. Make memory drawings to trace through the execution of code. Use drawings of the pre- and post-conditions of a problem to start thinking about a solution.

"The will to win means nothing without the will to prepare." - Juma Ikangaa, marathoner

(also attributed to Bobby Knight)

1 — Count()

Write a Count() function that counts the number of times a given `int` occurs in a list. Of course for your application, you need to change data type to char to count number of bases (A, T, C, G, or U). The code for this has the classic list traversal structure.

```
void CountTest() {
   List myList = BuildOneTwoThree();      // build {1, 2, 3}


   int count = myList.Count(1);     // returns 1 since there's 1 '2' in the list
}




/*
 Given a list and an int, return the number of times that int occurs
 in the list.
*/

int Count(int searchFor) {// as part of List's method
// Your code

}
```

2 — GetNth()

Write a GetNth() function that takes an integer index and returns the data value stored in the list at that index position. Also myGenome.GetNth() method that operates on the linked list myGenome and takes an integer index and returns the data value stored in the node at that index position. GetNth() uses the C numbering convention that the first node is index 0, the second is index 1, ... and so on. So for the list {42, 13, 666} GetNth() with index 1 should return 13. The index should be in the range [0..length- 1]. If it is not, GetNth() should assert() fail (or you could implement some other error case strategy).

```
void GetNthTest() {
   List myList = BuildOneTwoThree();            // build {1, 2, 3}
   int lastNode = myList.GetNth(2);             // returns the value 3
}
```

Essentially, GetNth() is similar to an `array[i]` operation — the client can ask for elements by index number. However, GetNth() on a list is much slower than [ ] on an array. The advantage of the linked list is its much more flexible memory management — we can Push() at any time to add more elements and the memory is allocated as needed.

```
// Given a list and an index, return the data
// in the nth node of the list. The nodes are numbered from 0.
// Assert fails if the index is invalid (outside 0..lengh-1).
int GetNth(int index) {//OO-style implementation
// Your code
```

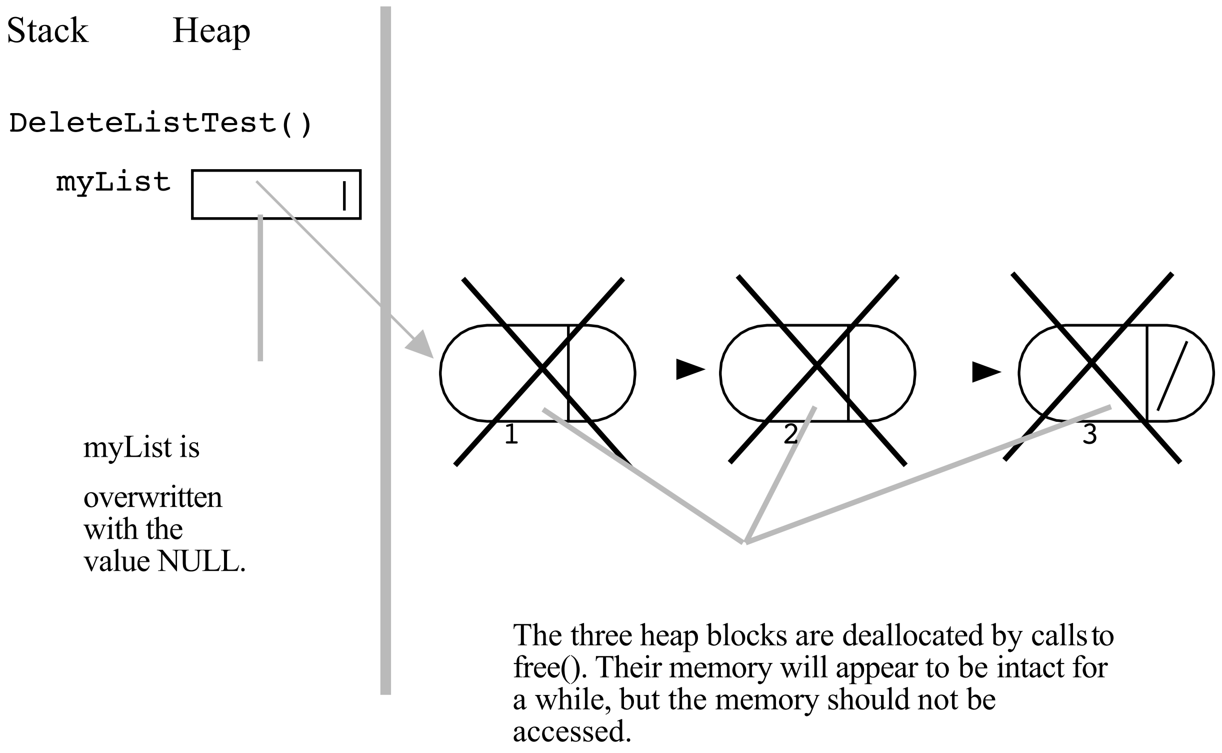
3 — DeleteList()

Write a function DeleteList() that takes a list, deallocates all of its memory and sets its head pointer to NULL (the empty list).

```
void DeleteListTest() {
   List myList = BuildOneTwoThree();            // build {1, 2, 3}


   myList = myList.DeleteList(); // deletes the three nodes and sets myList to NULL
}
```

Post DeleteList() Memory Drawing

The following drawing shows the state of memory after DeleteList() executes in the above sample. Overwritten pointers are shown in gray and deallocated heap memory has an 'X' through it. Essentially DeleteList() just needs to call free() once for each node and set the head pointer to NULL, if free() ior equivalent s available in the language, otherwise sets it to NULL and let garbage collector collect the freed up memory.


Stack
Heap
DeleteListTest()
myList
1
2
3
myList is
overwritten with the value NULL.
The three heap blocks are deallocated by calls to free(). Their memory will appear to be intact for a while, but the memory should not be accessed.

DeleteList()

The implementation also needs to be careful not to access the `next` field in each node after the node has been deallocated.

```
List DeleteList() {
// Your code
```

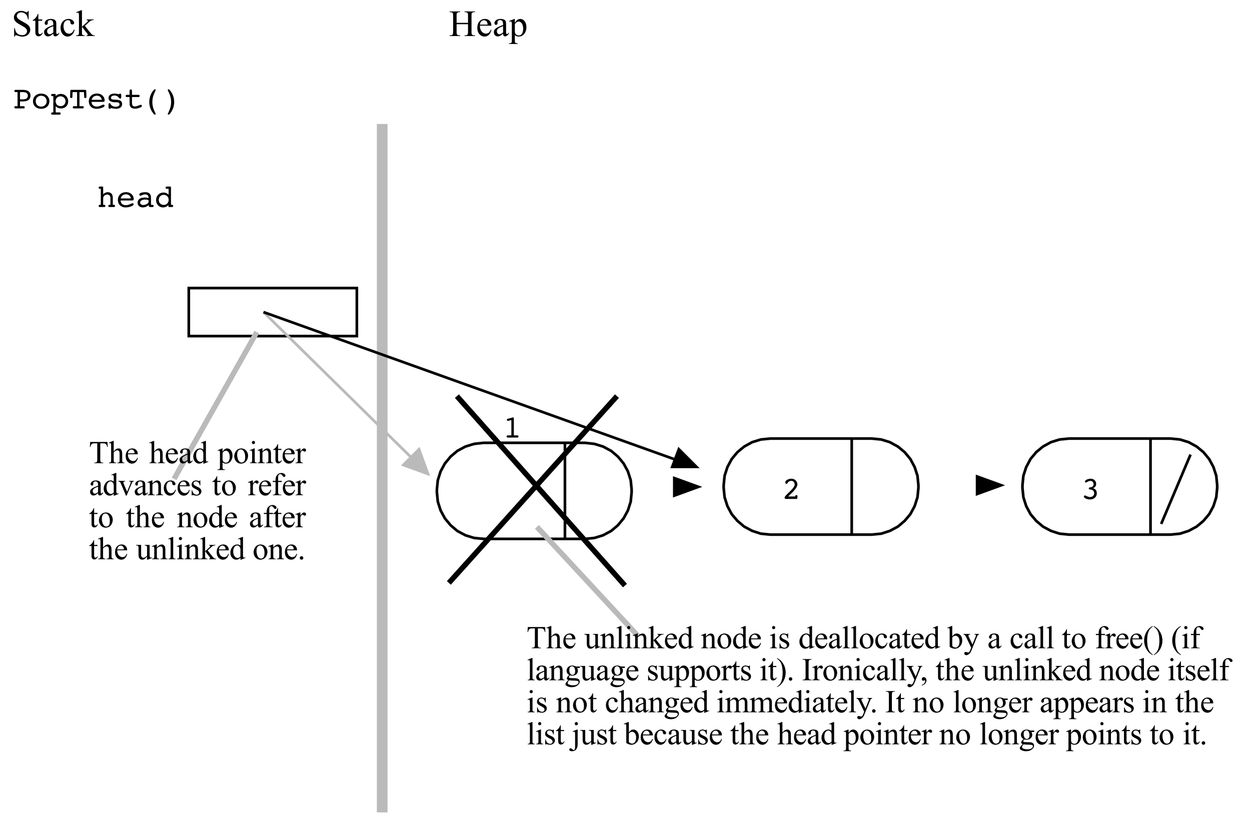
4 — Pop()

Write a Pop() function that is the inverse of Push(). Pop() takes a non-empty list, deletes the head node, and returns the deleted node's data. If all you ever used were Push() and Pop(), then our linked list would really look like a stack. However, we provide more general functions like GetNth() which what make our linked list more than just a stack. Pop() should assert() fail if there is not a node to pop. Here's some sample code which calls Pop()....

```
void PopTest() {
   List myL = BuildOneTwoThree();// build {1, 2, 3}
   int a = myL.Pop(); // deletes "1" node and returns 1
   int b = myL.Pop();   // oo-style deletes "2" node and returns 2
   int c = myL.Pop();   // deletes "3" node and returns 3
   int len = myL.Length(); // the list is now empty, so len == 0
}
```

Pop() Unlink

Pop() is a bit tricky. Pop() needs to unlink the front node from the list and deallocate it with a call to free() or set it to null. A good first step to writing Pop() properly is making the memory drawing for what Pop() should do. Below is a drawing showing a Pop() of the first node of a list. The process is basically the reverse of the 3-Step-Link-In used by Push() (would that be "Ni Knil Pets-3"?). The overwritten pointer value is shown in gray, and the deallocated heap memory has a big 'X' drawn on it...


Stack
Heap
PopTest()
head
1
2
3
The head pointer advances to refer to the node after the unlinked one.
The unlinked node is deallocated by a call to free() (if language supports it). Ironically, the unlinked node itself is not changed immediately. It no longer appears in the list just because the head pointer no longer points to it.

Pop()

```
/*
 The opposite of Push(). Works on a non-empty
 list and removes the front node, and returns the
 data which was in that node.
*/
int Pop() {
// your code...
```

5 — InsertNth()

A more difficult problem is to write a function InsertNth() which can insert a new node at any index within a list. Push() is similar, but can only insert a node at the head end of the list (index 0). The caller may specify any index in the range [0..length], and the new node should be inserted so as to be at that index.

```
void InsertNthTest() {
   List myL = new myList();   // start with the empty list


   myL.InsertNth(0, 13);      // build {13)
   myL.InsertNth(1, 42);      // build {13, 42}
   myL.InsertNth(1, 5);             // build {13, 5, 42}


   myL.DeleteList();    // clean up after ourselves
}
```

InsertNth() is complex — you will want to make some drawings to think about your solution and afterwards, to check its correctness.

```
/*
 A more general version of Push().
 Given a list, an index 'n' in the range 0..length,
 and a data element, add a new node to the list so
 that it has the given index.
*/
List InsertNth(int index, int data) {
// your code...
```

6 — SortedInsert()

Write a SortedInsert() function which given a list that is sorted in increasing order, and a single node, inserts the node into the correct sorted position in the list. While Push() allocates a new node to add to the list, SortedInsert() takes an existing node, and just rearranges pointers to insert it into the list. There are many possible solutions to this problem.

```
void SortedInsert(Node newNode) {
// Your code...
```

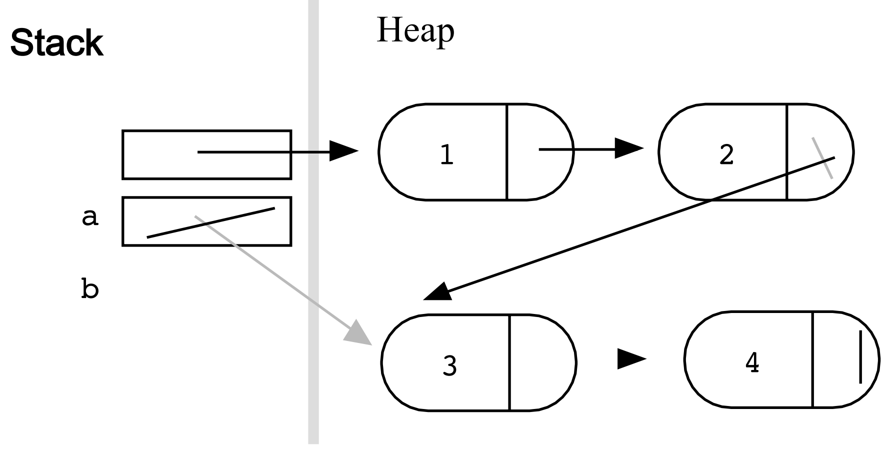
7 — InsertSort()

Write an InsertSort() function which given a list, rearranges its nodes so they are sorted in increasing order. It should use SortedInsert().

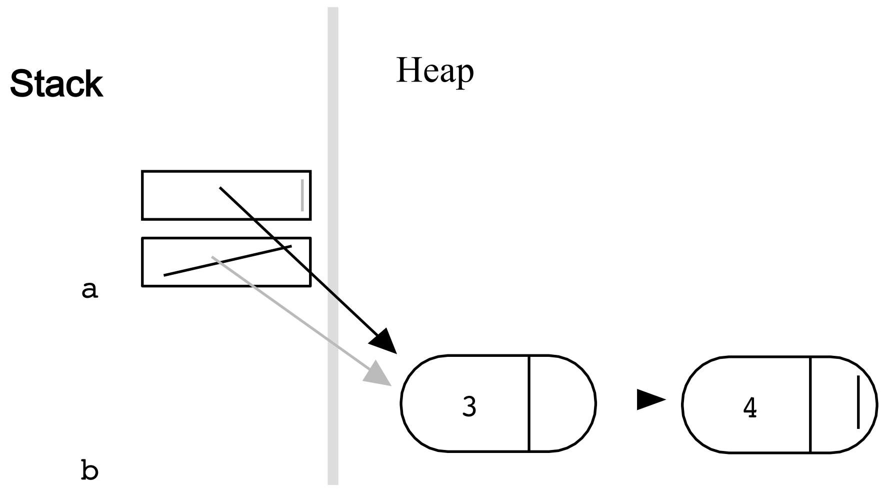
```
// Given a list, change it to be in sorted order (using SortedInsert()).
List InsertSort() { // Your code
```

8 — Append()

Write an Append() function that takes two lists, 'a' and 'b', appends 'b' onto the end of 'a', and then sets 'b' to NULL (since it is now trailing off the end of 'a'). Here is a drawing of a sample call to Append(a, b) with the start state in gray and the end state in black. At the end of the call, the 'a' list is {1, 2, 3, 4}, and 'b' list is empty.



It turns out that both of the head pointers accessed by a.Append(b) may need to be changed. The second 'b' parameter's head is always set to NULL. When is 'a' changed? That case occurs when the 'a' list starts out empty. In that case, the 'a' head must be changed from NULL to point to the 'b' list. Before the call 'b' is {3, 4}. After the call, 'a' is {3, 4}.



```
// Append 'b' onto the end of 'a', and then set 'b's head to NULL.
List Append(List second) {
// Your code...
```