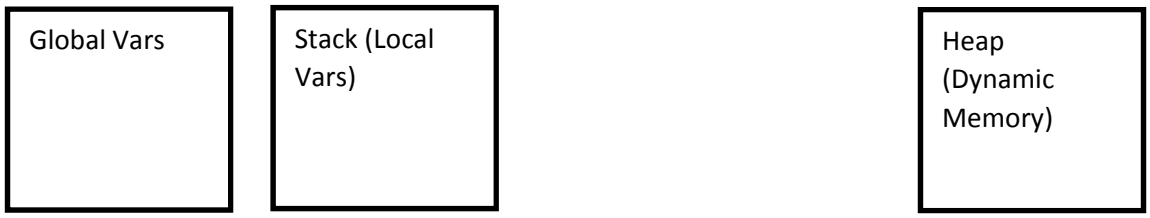
Chapter 7 – Linked Lists

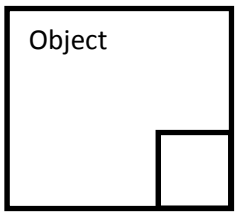
In order to use linked lists, you need to learn how to allocate memory dynamically on the heap and you need to learn about objects. Recall that local variables are temporary which get destroyed upon exit of the method. Objects that are built only of multiple native data types and/or objects are called structures. Structures that include a set of methods that operate only on its objects are called classes. By having methods that only work with the object, it opens up more naming options without the worry of name collision. For example, we could have a rectangular shape and a circle shape. Both shapes can use ***x*** and ***y*** as variables; and both can have a method called ***draw*** without any confusion because it will call the rectangular draw if it's a rectangle and the circle draw if it's a circle. This is one of the great features of classes with object oriented languages. Since methods belong to the class, it makes autocomplete for methods names easy because the list of possible methods is smaller.

With classes, we find that the 3 language quickly diverge. C++ and Java have classes called class. Objective-C also has classes but look quite different. C++ creates classes from scratch, so there is no overhead, but there is also no support. Java and Objective-C have classes derived from a base class: Object and NSObject, respectively. The base objects provide some functionality; most importantly is garbage collection or automatic reference counting (ARC). This means you don't have to manage the memory yourself.

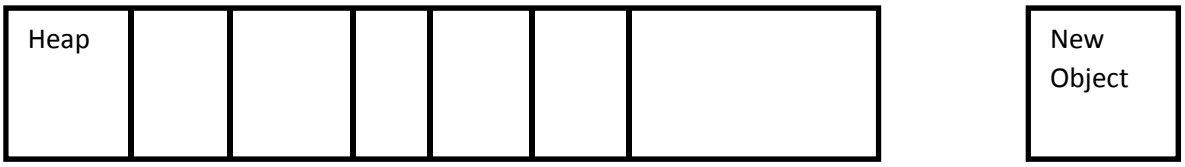
C++ requires that you free up objects when you don't need to use them anymore. If you don't free up memory and run a program for a long time, you will end up using all the memory and the program will crash. If you try to use a pointer from a deallocated object, you will get errors. Memory management is generally easy for small program, but more difficult for larger programs where objects may get used in unexpected ways.



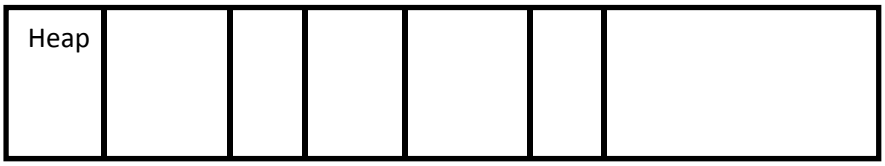
For ARC, when an object pointer gets assigned, the object counter increments; and whenever an object pointer gets changed to something else or goes out of scope, the object counter decrements. When the counter goes to zero, it gets deallocated.



A potential problem with manual memory management and ARC is that objects are allocated to specific memory locations. When objects are allocated and deallocated there is a potential for memory fragmentation. Too much fragmentation can lead to an out-of-memory error.



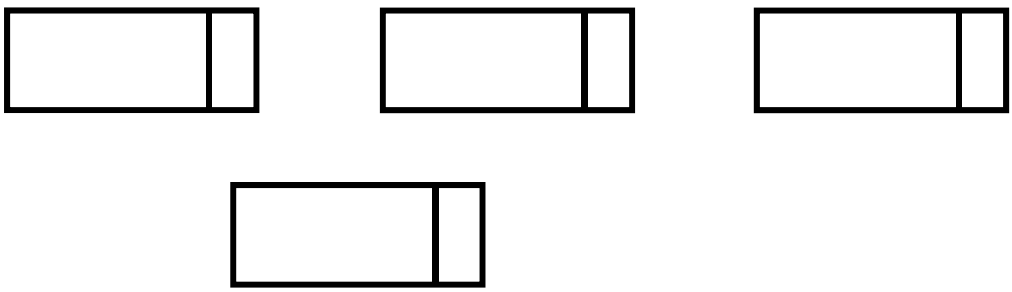
Java handles memory quite differently. It uses handles rather than pointers to memory. It also has garbage collection instead of counting. Handles are a little slower than pointers because they are pointers to pointers. The first pointer never changes, it is the handle. Garbage collection works by keeping track of all live objects, objects that are being pointed to from the roots. Objects that aren't accessible from the root are dead and the memory is available to use. The advantage of handles is that when memory gets fragmented, JVM (Java Virtual Machine) can defragment memory by removing all the free space holes. The handle to the actual object memory location is free to change and the handle still works. The disadvantage is you don't control when the JVM will or will not defragment memory which means it could stall your program at an inconvenient time.



Enough with the background, let's look at some code before continuing. We will look at the syntax for creating a class and how to create and delete objects. Recursion can be an elegant solution, but in general you want to avoid it if possible because for large lists, it will be a big hit for the stack.

> **Exercise 7.1: Using Exercise_7_1 as a template, create your own class with some member variables. Create some objects with your class. Assign the variables some values and print them out. There is no sample printout for this exercise.**

Now we will look at creating a class that can be used for linked lists. This is also known as a single linked list, as opposed to a double linked list which we will learn later. All we have to add to our class is a pointer that points to the next object. Before we start with code, let's look at the concept.



To make inserting and deleting easier, we will use a "Head" node. It does not contain any information; it simply serves as a pointer to the linked list.

Let's have a quick look at the class definition for *Node*. Notice that there are private member variables and methods. These are things we don't want other code to look at or use; they are only meant to be used internally for the class. The reasons for this will be explained at a later time.

It is useful to have a constructor method that initializes all the member variables so that they are not undefined. The pointer typically gets set to *0, NULL, null,* or *nil*; depending on the language you are using. This is the default constructor; there is also a constructor that sets the next pointer. Although this can be done manually (if it wasn't private), we don't want other code incorrectly changing the pointer.

Printing the nodes is done by skipping over the head node, then checking the pointer. If it's *NULL*, we stop. If not, we print the number then advance the pointer to the next node and repeat checking the pointer.

addNodeToEnd – this will go through the links until the next pointer is *NULL*. Then change the next pointer to the passed-in node.

getNode – we skip the head node and go through the links until the pointer is *NULL* or the number matches the passed-in value. If there is a match, it will return the node pointer otherwise return *NULL*.

getNodeIndex – surprisingly, this one is a little more difficult than the previous. We skip the head node and go through the links until the pointer is *NULL* or the number of nodes travelled matches the passed-in value. If you encounter a *NULL* pointer first, then you just return *NULL*.

sortNode – we start at the head node and loop until the pointer is *NULL*. If the next pointer is null or the passed-in number is less than the number of the next node, then we link in the passed-in node between the current node and next node.

deleteNode – deleting is actually the hardest task. This will go through the links until the next pointer is *NULL*. If the passed-in number matches the number of the next node, then we set the next pointer to the next pointer of the deleted node and return true. If we don't find a match, we return *false*. Bonus if you properly free up memory (refer to *deleteAll*).

> **Exercise 7.2: Complete all the empty methods in the sample code. Do NOT change any of the code in main or the method definitions; it is your test data. Below is how your printout should look like.**

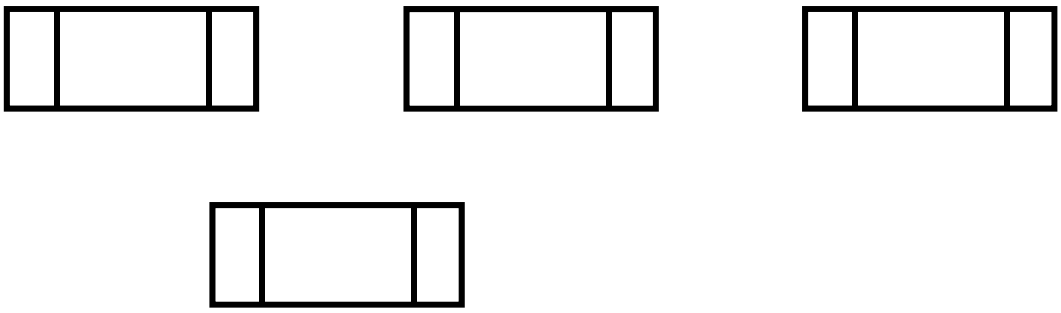
```
Print list using loops:
13 18 23
Print list using recursion:
13 18 23
Print list in reverse using recursion:
23 18 13
Get 13: 25e4e6db
Get 10: null

Get index 0: 25e4e6db
Get index 2: 56f0474c
Get index 3: null
```

```
Add a sorted item:
13 18 20 23
Add a sorted item:
6 13 18 20 23
Deleting node with 18: true
6 13 20 23
Deleting node with 17: false
6 13 20 23
Deleting node with 23: true
6 13 20
Deleting all:
```

In general, linked lists aren't used much. The problems are that there is no random access like arrays; and searching is very slow for a sorted list, *O(n)*, compared to other methods that are *O(lg n)*. You might use a linked list to keep track of items that are just used sequentially or items that are added and removed without being accessed much. Linked lists may be chosen over arrays because some arrays have fixed sizes where linked lists can always add or remove nodes. The main reason we learn linked lists is that they are a good introduction to using the memory (heap) and pointers. Although linked lists aren't used too much, the use of links is important for other similar but slightly more complicated structures such as trees; trees are a very much used and efficient structure in programming.

To make linked lists a little easier to use, we can add a link to the previous node; this is called a double linked list. The API is identical for most of the methods. This is because the implementation should not affect how the linked list is used. It is identical except for one constructor and the print routines are not recursive because of the extra overhead to pass parameters.



Exercise 7.3: Complete all the empty methods in the sample code. Do NOT change any of the code in main or the method definitions; it is your test data. The output is identical for the routines you need to write. All checks for *NULL* need to be changed to checks for the head node.