# Assignment 3: Ordered Collection Data Types

CS 301

February 8, 2018

Now that we've explored some data structures from the outside, we're going to try to build some of our own. We'll focus on various data types for organizing ordered collections—that is, different kinds of lists.

We will create our data structures as objects, and as we do so, we want to think about keeping the big O running time of their methods as low as possible.

1. First we'll build a type of list, called a linked list. Linked lists are made up of nodes. Each node contains one list element, along with a link to the next node in the list. (Hence the name.) Create a `Linked_List` object that implements a linked list. It should use a helper class `Node`. It should implement the following methods:

   **List()** creates a new list that is empty. It needs no parameters and returns an empty list.

   **add(item)** adds a new item to the beginning of the list. It needs the item and returns nothing.

   **remove(item)** removes the item from the list. It needs the item and modifies the list. If the item isn't in the list, it raises an error.

   **search(item)** searches for the item in the list. It needs the item and returns a boolean value.

   **isEmpty()** tests to see whether the list is empty. It needs no parameters and returns a boolean value.

   **size()** returns the number of items in the list. It needs no parameters and returns an integer.

   **append(item)** adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing.

   **index(item)** returns the position of item in the list. It needs the item and returns the index. If the item isn't in the list, it raises an error.

   **insert(pos,item)** adds a new item to the list at position `pos`. It needs the item and returns nothing. If the list is too short, then it raises an error.

   **pop()** removes and returns the last item in the list. It needs nothing and returns an item. If the list is empty, it raises an error.

   **pop(pos)** removes and returns the item at position pos. It needs the position and returns the item. If the list is too short, then it raises an error.

2. A *Doubly-linked list* is just like a linked list, except that each node is linked in two directions—to the following node, and to the previous node. The head of the list should be linked both to the beginning and to the end. Thus, it allows traversal in both directions. Modify your code from one to implement a doubly-linked list.

3. Do you think that python's internal representation of a list is a linked-list, a doubly-linked list, or something else? Why or why not? Insert a comment in your code to explain your thinking about this.

4. A *stack* is like a list, except that items can only be added or removed from one end (the "top" of the stack). It is like a stack of books—you can add a book to the top of the stack, or remove a book from the top of the stack. Stacks are referred to as LIFO—"Last in, First out"—data structures. Implement a stack. You don't have to redo all of the work you did to create a linked list; you can use one of you your list structures or python's built-in list structure to implement your stack. You should think about about what would make the most sense. Your stack should implement the following methods:

   **Stack()** creates a new stack that is empty. It needs no parameters and returns an empty stack.

   **push(item)** adds a new item to the top of the stack. It needs the item and returns nothing.

   **pop()** removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

   **peek()** returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.

   **isEmpty()** tests to see whether the stack is empty. It needs no parameters and returns a boolean value.

   **size()** returns the number of items on the stack. It needs no parameters and returns an integer.

   In comments, describe the running time of each method of your implementation.

5. A *queue* is just like a stack, except that it has a FIFO—"First In, First Out"—structure. Implement a queue. It should have the following methods:

   **Queue()** creates a new queue that is empty. It needs no parameters and returns an empty queue.

   **enqueue(item)** adds a new item to the rear of the queue. It needs the item and returns nothing.

   **dequeue()** removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.

   **isEmpty()** tests to see whether the queue is empty. It needs no parameters and returns a boolean value.

   **size()** returns the number of items in the queue. It needs no parameters and returns an integer.

   In comments, describe the running time of each method of your implementation.

6. A *deque* is a "double-ended queue." It combines the features of a stack and a queue. Implement a deque in python. It should have the following methods:

**Deque()** creates a new deque that is empty. It needs no parameters and returns an empty deque.

**addFront(item)** adds a new item to the front of the deque. It needs the item and returns nothing.

**addRear(item)** adds a new item to the rear of the deque. It needs the item and returns nothing.

**removeFront()** removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.

**removeRear()** removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.

**isEmpty()** tests to see whether the deque is empty. It needs no parameters and returns a boolean value.

**size()** returns the number of items in the deque. It needs no parameters and returns an integer.

In comments, describe the running time of each method of your implementation.

7. As an application, look up "Reverse Polish Notation" on wikipedia. Write a function that inputs a string containing an expression in reverse polish notation, evaluates it, and outputs the answer, also as a string.