CPS 112
Homework 6
Papa Nii Vanderpuye

iterators.py is my solution to part 1 and part 2. In iterators.py, I implemented a subclass of **ForwardIterator** called **ForwardAssignableIterator**. This class represents an iterator that allows one to additionally set the item in the position pointed to by the iterator. It will inherit the methods of its superclass, so I added the abstract method **setItem(item)**, which sets the item at the current position.

In the same module I implemented the class **PythonListFAIterator**. It takes two parameters and stores them as instance variables; the list the iterator is working with, and the index at which it should start. The **getNext** method increases the current index by one. The **getItem** and **setItem** methods should **get** and **set** the item associated with the current index. If the current index is too big, **getItem** returns None and **setItem** does nothing. The **getLoc** method returns the current index. The **clone** method returns a new iterator pointing to the same list/index as the current iterator.

linkedlist.py is my solution to part 2. In the module, I implemented the class **LinkedListFAIterator.** Its constructor takes one parameter and stores it as an instance variable, the node the iterator should start pointing to. The **getNext** method changes the current node to the current node's next node. The **getItem** and **setItem** methods get and set the item in the current node. The **getLoc** method returns the current node. The **clone** method should return a new iterator pointing to the same node as this iterator. I modified the **getStartIterator()** method of the LinkedList class to return a new **LinkedListFAIterator** pointing to the head of the list.

iteratorsorts.py is my solution to Part 3. In the module I implemented the selection sort algorithm, using forward iterators. My function **selectionSort** takes a single parameter, a **ForwardAssignableIterator** object. It uses selection sort to sort the items in the container associated with the iterator. My implementation finds the minimum element and puts it at the beginning of the list. In the selection sort two clones are created first, one to assign the previous clones and one to use as the iterator. the one used for the iterator goes through the list attempting to find a minimum, once it attains a new minimum, it saves itself as a previous clone, and clones itself, so the the new clone will continue the iteration to look for another minimum. When that is done, the previous clone, which should be the minimum, and the original iterators swap the values in their positions.

sorttest.py is my solution to part 3. I created a main function that generates 10 random 20-element lists, a random 1-element list, and an empty list for both Python lists and LinkedLists. It prints the lists before and after sorting them with my selection sort function.

iterators.py is my soltution to Part 4. In *iterators.py*, I created an abstract subclass of **ForwardAssignableIterator** called **BidirectionalAssignableIterator**. It defines the interface for an iterator that can move forward or backward in a structure. I added one abstract method, **getPrevious()**, which moves the iterator to the previous location in the collection. I also implemented a concrete subclass of **BidirectionalAssignableIterator** called **PythonListBAIterator**. It is like the **PythonListFAIterator**, except it can move forward and backward through the list.

iteratorsorts.py is my soltution to Part 5. In it I created the insertion sort algorithm but using bidirectional iterators instead of list indices. My **insertionSort** function takes one parameter, a bidirectional iterator pointing to the beginning of the list. It should sort the items in the container using the insertion sort algorithm we studied in class. It clones the iterator twice and uses one as the clone that checks and compares the values iterates through to a end new value every time the iterator moves to a bigger subset. and it uses the other clone, which is right behind the first clone in terms of iteration, swaps its value when the current value is bigger that the value in the first clones position, this essentially shifts all values that are bigger than the compared avlue till it comes across a middle position to insert the compared value. I modified *sorttest.py* to additionally generate 10 random 20-element Python lists, a random 1-element Python list, and an empty list, and print them before and after sorting using my insertion sort function.

doublylinkedlist.py is my soltution to Part 6. In it, I created a concrete subclass of **BidirectionalAssignableIterator** called **DoublyLinkedListBAIterator**, representing a bidirectional iterator in a doublylinked list. It is like the **LinkedListFAIterator,** except it can move forward and backward through the list. It has three methods: **getStartIterator()** which returns a new bidirectional iterator pointing to the head, **getEndIterator()** which returns a new bidirectional iterator pointing to the tail, and **__iter__()** which adds for loop support. **clone()** Returns a bidirectional iterator pointing to the head. I modified the main function in *sorttest.py* to create 10 random 20-element doubly-linked lists, a random 1-element doubly-linked list, and an empty doubly-linked list and print them before and after sorting them with my insertion sort function.