Module 7 - Overview

Introduction

Module Learning Outcomes

After successful completion of this module, you be able to do the following:

1. Write a LinkedList class.
2. Write Stack and Queue classes.
3. Describe what None is and why it's useful

Key questions:

* What is an ADT?
* On what basis would we choose between different implementations of an ADT?
* What's the key difference between a stack and a queue?

Explorations

Use the pages within this module to explore the following concepts:

* Exploration: [Linked lists, stacks, queues](https://canvas.oregonstate.edu/courses/1915078/pages/exploration-linked-lists-stacks-queues) (CLO 2, MLOs 1-3)
* Video Demo: [Linked Lists](https://canvas.oregonstate.edu/courses/1915078/pages/video-demo-linked-lists) (CLO 2, MLOs 1-3)
* [Module 7 exercise solutions](https://canvas.oregonstate.edu/courses/1915078/pages/module-7-exercise-solutions)

Optional Resources

* *Problem Solving with Algorithms and Data Structures Using Python*
  + [Chapter 4 Sections 3-5Links to an external site.](https://runestone.academy/ns/books/published/pythonds3/BasicDS/toctree.html)
  + [Chapter 4 Sections 10-12Links to an external site.](https://runestone.academy/ns/books/published/pythonds3/BasicDS/WhatIsaQueue.html)
  + [Chapter 4 Sections 21-23Links to an external site.](https://runestone.academy/ns/books/published/pythonds3/BasicDS/ImplementinganUnorderedListLinkedLists.html)

Task List

Complete the following assignments and other tasks:

* Read the Exploration pages and do the interactive exercises on those pages (CLO 2, MLOs 1-3).
* Do [Assignment 7](https://canvas.oregonstate.edu/courses/1915078/assignments/9227007), which gives you more practice with writing recursive functions (CLO 2, MLOs 1-3).
* Take [Quiz 7](https://canvas.oregonstate.edu/courses/1915078/quizzes/2859163) (CLO 2, MLOs 1-3).

Exploration: Linked lists, stacks, queues

ADTs and data structures

We’ve seen some built-in Python data structures like lists, dictionaries and sets. Data structures provide a way to organize data and perform a certain group of operations on it. We can conceptually separate the logical interface from its implementation. An **abstract data type** (ADT) defines the kinds of data that are stored and the operations that are available to perform on that data. A **data structure** provides a specific implementation of an ADT. Different ADTs are better suited for different contexts and can be very specific to a particular use case. An ADT can also be implemented as a data structure in different ways that optimize certain operations over others. Let’s take a look at an ADT you’re somewhat familiar with.

Lists, linked lists

Lists are a common data structure used to organize data sequentially. You’ve seen that Python comes with a built-in list type. There are two main approaches to implementing a list. One is to store all of the elements contiguously in memory. The other is to allow elements to be discontiguous, and have each **node** in the list refer to the next node in the list. Python’s built-in list type takes the first approach, but we can implement a list type that takes the second approach, known as a **linked list**. First we create a Node class with two data members: *data*, which holds the value we’re storing in the Node, and *next*, which refers to the next Node in the list.

The data members of the Node class in this example are not private. The reason for that is because Node is a trivial class that contains only two data members and no methods (besides init), so there's not a need for encapsulation. Another way of putting it is that there's no need to separate interface from implementation because there is no interface (public methods of the class).

Then we define a Linked\_list class with methods that let the user create and manipulate a list of Nodes. The only data member of the Linked\_list class is *head*, which we use to keep track of the first node in the list. Here we implement just a few of the most common list operations. The exercises will ask you to implement a few others.

A screenshot of a computer program

Description automatically generated with low confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

* The constructor initializes *head* to None because there are no nodes in the list yet.
* Performing an operation on an empty data structure is often a special case we need to make sure is handled correctly. The **add** method first checks to see if the list is empty (by checking whether *head* is None). If it is, then the method creates a new node containing the parameter value and assigns it to be the head. If the list was not empty, the method sets *current* equal to *head*, so they refer to the same node. Then while there’s still another node in the list, it sets *current* to refer to that node. When it arrives at the last node in the list, it creates a new node containing the parameter value and makes the (formerly) last node refer to it (with its *next* data member).
* The **display** method sets *current* equal to *head* and then advances through the list as in the add method, but it prints out the value in each node as it goes. In order to print all the values on a single line, it uses an optional parameter that makes it print a space at the end instead of a newline. There’s an empty print after the loop to print a newline.
* The **remove** method first checks to see if the list is empty. If so, it simply returns. Next it checks to see if the head should be removed. If so, it makes *head* refer to the next node (or None). If the list is not empty, and the head is not being removed, then the method advances through the list until either *current* falls off the end or the value is found. The variable *previous* also advances through the list one step behind *current*. If the value was found, then *current* is spliced out of the list by making the node behind *current* (which is *previous*) refer to the node ahead of *current* (which is *current.next*). If the value was not found, then the method does nothing.
* The **is\_empty** method simply returns True if *head* equals None and returns False otherwise.

The add, display, and remove methods all advance through the list in a similar way, but their loop conditions are a bit different:

while current.next is not None: #add  
while current is not None: #display  
while current is not None and current.data != val: #remove

* The **add** method looks a step ahead to see if it’s on the last node. It needs a reference to the last node in the list in order to make it refer to a new node. If it lets *current* fall off the list, it loses that reference.
* The **display** method doesn’t look ahead. It’s fine if *current* falls off the list because this method doesn't need to change the list.
* The **remove** method does need to be able to change the list, but only if the value is found. If *current* falls off the list, that means the value wasn’t found, so the list doesn’t need to change. Notice the need for the short-circuit evaluation of **and**. The condition first checks that *current* is not None. If it is None, then the second part of the condition is not evaluated, because we already know the condition is false. This prevents us from accessing a data member that doesn’t exist (because None isn’t a node, so it doesn’t have *data*).

You might think that "is not None" seems odd, because it sounds like a double negative, but it's technically not. "None" is the name of a special Python value that is used (perhaps confusingly) to represent the absence of a value. It's a useful marker (or "sentinel value") that we can check for before trying to do something with a variable that might not make sense. For example, in this LinkedList class, we use None to mark the end of the list. Trying to go past the end of the list in the methods wouldn't make sense.

Would "if x:" mean the same thing as "if x is not None:"? Not quite. "if x:" will evaluate as true if x is not any "falsy" value. None is a falsy value, but so are: False, 0, 0.0, "" (the empty string), [] (the empty list) , () (the empty tuple), {} (the empty dictionary), and a few others. "if x is not None:" is more specific.

Is there any reason to favor either the contiguous memory approach or the discontiguous memory approach? Yes – that choice depends on which operations you want to optimize. When the elements are contiguous in memory, that allows us to index into the list in constant time, O(1). On the other hand, inserting into or deleting from the list will be O(n), because all of the elements after the insertion or deletion point have to be shifted over to remain contiguous in memory. With a linked list, getting the value at a particular node is O(n), since we have to start at the beginning of the list and step through it node-by-node. On the other hand, inserting into or deleting from a linked list can be done in O(1) if we already have a reference to the node before the insertion or deletion point.

<https://cfvod.kaltura.com/p/391241/sp/39124100/thumbnail/entry_id/0_vq1ik81k/version/100032/width/608/height/372>

Try playing around with the above linked list example in [Python TutorLinks to an external site.](https://pythontutor.com/) (which, as mentioned in the last module, is a helpful site for visualizing what's happening when you run Python code).

Stacks and queues

You can think of **Stacks** and **queues** as more restricted types of list. Specifically, they restrict the way that data are accessed, added to, or removed from a list. Why would we restrict ourselves? Why not just use a list? Restricting our interface to the small set of operations that are needed for a stack or queue helps to simplify and focus our use of the data structure to solve a particular problem.

A stack, like a list, is an ordered sequence of values. The operations provided by a stack are:

* push: adds a value to the “top” of the stack
* pop: removes the value at the “top” of the stack
* peek: returns the value at the “top” of the stack without removing it from the stack
* is\_empty: returns True if the stack is empty, returns False otherwise

A queue is similarly an ordered sequence of values, but with different operations provided:

* enqueue: adds a value to the “end” of the queue
* dequeue: removes a value from the “beginning” of the queue
* is\_empty: returns True if the queue is empty, returns False otherwise

The big difference between a stack and a queue is in how values are added and removed. A stack works like a stack of plates. If you want to add a plate to the stack, it goes on top. The only plate you can remove from the stack is the one on top (you don't want to break any plates, do you?). The last one added is the first one removed, so we say that stacks are LIFO (last in, first out). A queue works like a group of people waiting in line. People join at the back of the line and are removed at the front of the line. The first one added is the first one removed, so we say that queues are FIFO (first in, first out). Both stacks and queues can be implemented using either kind of list. Let’s look at implementations for both that use Python’s built-in lists.

A screenshot of a computer program

Description automatically generated with low confidence

A screenshot of a computer program

Description automatically generated with medium confidence

Notice that enqueue and push both add an element to the end of the list, but pop removes an element from the end, whereas dequeue removes an element from the front.

We could have implemented Stack and Queue using inheritance instead of composition, but that would be incorrect. It's not true that a stack is-a list or that a queue is-a list. Lists have many operations that don't apply to stacks or queues, so composition is the more appropriate choice.

Exercises

As you work on these questions, pay attention to the process of reading a description of requirements and figuring out a specific list of tasks. This is an important skill that you can practice on these simple exercises before you try the requirements of the assignments.

1. Write a LinkedList method named *contains*, that takes a value as a parameter and returns True if that value is in the linked list, but returns False otherwise.

You can use this starter code instead of writing the entire class from scratch: [linked\_lists\_1\_exercise\_starter\_code.py](https://canvas.oregonstate.edu/courses/1915078/files/98541891/preview)[Download linked\_lists\_1\_exercise\_starter\_code.py](https://canvas.oregonstate.edu/courses/1915078/files/98541891/download?download_frd=1)

Be sure to rename the above file to LinkedList.py before you run these tests in PyCharm: [linked\_lists\_1\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98541976/preview)[Download linked\_lists\_1\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98541976/download?download_frd=1)

2. Write a LinkedList method named *insert* that takes as parameters a value and a position (in that order). A position of zero means that the new value will become the new first element. A position of one means it will become the new second element, etc. A position >= the length of the list means it will be added at the end of the list.

You can use this starter code instead of writing the entire class from scratch:

[linked\_lists\_2\_exercise\_starter\_code.py](https://canvas.oregonstate.edu/courses/1915078/files/98541926/preview)[Download linked\_lists\_2\_exercise\_starter\_code.py](https://canvas.oregonstate.edu/courses/1915078/files/98541926/download?download_frd=1)

Be sure to rename the above file to LinkedList.py before you run these tests in PyCharm. These tests need to\_regular\_list() method given in the starter code, make sure to copy this method from the starter code if your code does not has it.

[linked\_lists\_2\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98541854/preview)[Download linked\_lists\_2\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98541854/download?download_frd=1)

3. Write a LinkedList method named *reverse*, that takes no parameters (besides *self*) and doesn't return a value, but just reverses the order of the nodes in the linked list. The reverse method should **not** change the *data* value each node holds - it must rearrange the order of the nodes (by changing the *next* value each node holds).You must include the to\_regular\_list() method given in the starting code, which is used by the tests.

You can use this starter code instead of writing the entire class from scratch: [linked\_lists\_3\_starter\_code.py](https://canvas.oregonstate.edu/courses/1915078/files/98541950/preview)[Download linked\_lists\_3\_starter\_code.py](https://canvas.oregonstate.edu/courses/1915078/files/98541950/download?download_frd=1)

Be sure to rename the above file to LinkedList.py before you run these tests in PyCharm:

[linked\_lists\_3\_exercise\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98541966/preview)

Review Video Demo:

<https://media.oregonstate.edu/media/t/0_oiwwp70r>