CS 162P
Single-Linked Lists

Linked Lists

Linked lists are a basic data structure used to keep track of information like arrays or Python lists. They can be used to store data, or more abstract data types such as stacks, queues, and double-ended queues can be implemented with them.

They are different from Python lists in several ways. In a Python list or array, the data is stored contiguously in memory and it is possible to use indexing to access any location in a single step. In linked lists, the data is in links that are not stored contiguously, and it is necessary to walk down the links from the head or starting point to find a given item. The memory in Python lists and arrays is allocated in a single block, so when they grow it is necessary to create a new larger block and copy the data over. Linked lists grow and shrink dynamically as new links are created and added to the linked list or old links are removed.

There are two basic types of linked lists, single-linked lists where there is a forward connection between two links and double-linked lists with a connection forward and backwards. This document focuses on the simplest one, a single-linked list. It shows how items can be added or removed from the head, or front of the linked list, and how it can be searched to find a given item.

It then shows how to modify the linked list to add items at the tail and how a linked list can be used to implement a stack or a queue.

Basic structure

A linked list is a set of links joined using references between them. It can be imagined as a chain where each link of the linked list is represented by a single link in the chain.

Imagine building a paper chain. Before the first link is created, there is nothing. The first link is created and held in one hand. Then, a second link is looped through the first, and the first is dropped, no longer relevant. As each subsequent link is added to the front of the chain, the end of chain (which is the first link that was added) gets further away. The only links that are tracked are the new one that is being created and the previous one currently held in hand.

Continuing with the paper chain example, to figure out how many links of each color were in the chain, the most efficient way is to start at the link being held and move through the chain, one link at a time, counting the links of each color, until the end is reached.

When creating a LinkedList class, the class contains a variable, which is a reference to the start of the linked list, typically called the head. Each link then contains some data and a pointer to the next item, typically labeled next. Finally, the last link in the linked list points to **None** to terminate it.

Like the paper chain, a linked list starts at the head and continues, link by link, to the end. Traversing the linked list can only go in a single direction, from each link to the next link, because that is the only data stored in each link.

Link Class

Since a linked list is composed of links, it is necessary to define the structure of a link first. A link is a single object in memory and is defined as an independent class.

A link contains at least two things. One is the value being stored in it (often an object) and the other is a reference or pointer to the next link, normally named next. The Link class is not used independently, instead it is only used in the LinkedList class.

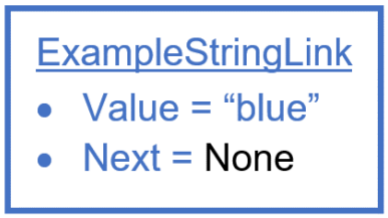
For a single-linked list as this document describes, only those two things are necessary. To keep the following coding examples simple, the value stored in the link will be an integer. Remember that while the values are integers in these examples, the link value can be any data type that has been defined.

```
# definition of a Link Class
class Link:
   # init method that initializes the class variables
   def __init__(self, value):
      self.__value = value
      self.__next = None

   # setter to allow us to change the next value
   def setNext(self, next):
      self.__next = next

   # getters to allow us to access the class variables
   def getNext(self):
      return self.__next

   def getValue(self):
      return self.__value
```



Note that this link does not have a method to set a new value. A common convention for linked lists is that values are not changed, the link is instead replaced. The __init__ function provides a default value for next so that it does not have to specified when creating a new link.

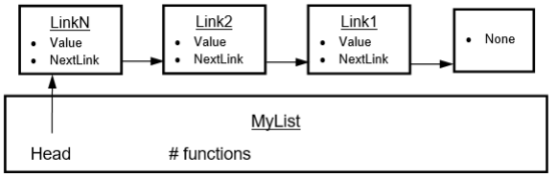
LinkedList Class

By itself, a link is not very useful. It is necessary to create a LinkedList class that includes the methods used to maintain the linked list. The LinkedList class for a single-linked list consists of a variable head that contains the address of the first link, and a series of functions which use head to interact with and access the links in the linked list. This is an important shift from structures that can be accessed via index (such as Python lists and arrays), where each individual element can be directly accessed. Because of this difference, all access to links and their values (other than the first one) require stepping through intermediate links and is O(N).

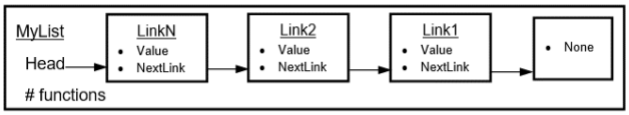
Note that all languages access list elements via pointer, regardless of whether it is done explicitly (C++) or implicitly (Python, C#).

While technically this means that a linked list does not actually contain any of the link elements, conceptually and diagrammatically a linked list is considered to be all of the links accessible via its head.

Technical Representation



Conceptual Representation



For the purpose clear communication, this conceptual representation of a linked list will be used throughout this document rather than the technical one.

For a single-linked list with a single end, the only data stored in the LinkedList class is the variable for head. The methods this document covers are __init__, addHead, getHead, removeHead, findValue, and findRemove. In addition, a __repr__ method is defined to allow displaying a logical representation of the linked list.

```
class LinkedList:
    # the only class variable is the head and it starts out as None
    def __init__(self):

    # add a new link containing value at the head of the list
    def addHead(self, value):

    # return the value contained in the first link of the list
    def getHead(self):

    # remove the first link in the list
    def removeHead(self):

    # return true if the list is empty
    def isEmpty(self):

    # return True if value is present in the list
    def findValue(self, value):

    # return True if value is present in the list
    # and remove that link from the list
    def findRemove(self, value):

    # return a printable representation of the list
    def __repr__(self):
```

Initialization method

The __init__ method sets the **head** to **None** to initialize it.

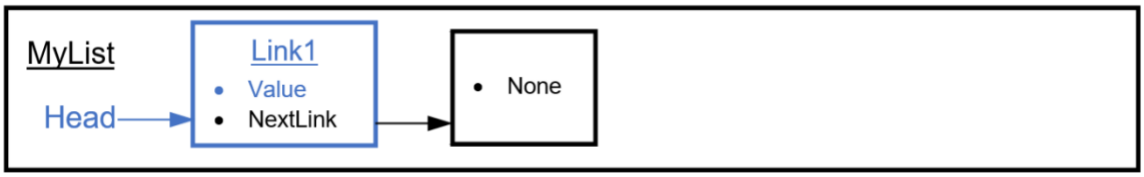
```
    # the only class variable is the head and it starts out as None
    def __init__(self):
        self.__head = None
```

Adding a value at head

The first method to consider is adding a new link to the linked list. Since **head** is the only variable in the LinkedList class and all new links must be added via it the method is named addHead.

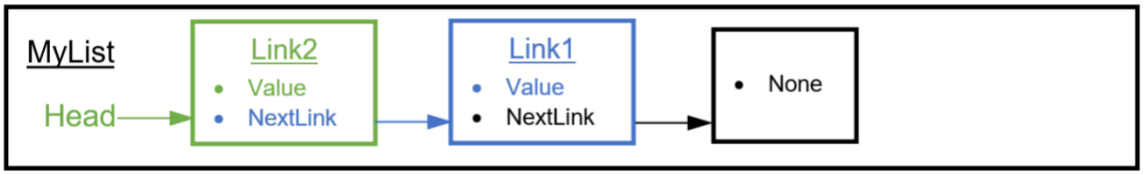
The following diagrams show the two cases to be considered when adding a new link to a linked list. The first is adding a value to an empty list, where head points to **None**.

The first link is added by creating a new link, and then **head** is set to point to it. The Link class described above automatically sets the link's **next** value equal to **None**, so nothing further is required.



To make the connections visually easy to follow, in these diagrams all pointers are colored to match the link (or in the case of **None**, the lack of a link) that they are currently pointing to.

The second case for adding a link is when a link is added to a non-empty list. Since there is already a link in the list, when the new link is created its **next** value must be set to point to the link that **head** points to before **head** can be set to point to the new link.



It is always best practice to consider possible special cases when creating code. In this circumstance it is easily demonstrated that no special case code is required. In the first case, **head** contains the end of the list indicator (**None**) and in the second case **head** contains the address of a link. This allows the code to be reduced to a single case.

Since the same basic action is performed in both cases, the code simply needs to create a new link, set the new link's next value to the value stored in head, and then set head to the new link.

```
# add a new link containing value at the head of the list
def addHead(self, value):

    # create a new link containing the value
    temp = Link(value)

    # update its next field to contain whatever head contained
    temp.setNext(self.__head)

    # update head to point to the new link
    self.__head = temp
```

This creates a new link in memory and adds it to the front of the linked list. This is part of the dynamic nature of a linked list, as new links are added the memory usage increases.

Since the size of the linked list, the number of links, does not affect the number of steps in this method, it is O(1).

Accessing the first link

This version of a linked list implements two methods which directly access the link at head. The first returns the value contained in that link and is named getHead.

Again, both an empty linked list and a non-empty linked list must be considered. Because this method reaches into a link object to extract a value and there is nothing inside of **None** that could be extracted, a special case must be handled with when coding this method.

```
# return the value contained in the first link of the list
def getHead(self):

    # if list is empty, throw an exception
    if self.__head is None:
        # raise an exception

    # return the value from inside head
    return self.__head.getValue()
```

Removing from the head

The second method for accessing the linked list from head removes the first link without returning a value. Since it removes the link pointed to by head, it is named removeHead.

Again, the method must deal with the special case of an empty linked list. It is impossible to remove something from nothing, so if the linked list is empty the code should raise an exception.

Again, it does not matter if it has only a single link or multiple links. The value contained in the variable **next** in the first link is stored in **head**, this is either **None** or the address of the next link in the list.

```
def removeHead(self):

    # if list is empty, throw an exception
    if self.__head is None:
        # raise an exception

    # change head to point at the next link in the list
    self.__head = self.__head.getNext()
```

Since the number of links in the linked list does not affect the number of steps in this method, it is also O(1).

Detecting an empty list

Since Python emphasizes code readability, it is desirable to replace the test for an empty linked list with a method that communicates the intent of the code more clearly. Here is a simple method that does so.

```
# return true if the list is empty
def isEmpty(self):
   return self.__head == None
```

Searching

Since all accesses to a linked list require starting at the head, it is not possible to index into a given location. Therefore, all searches are linear, starting at the head then checking each link for the desired value and quitting when it is found, or the end is reached.

This means that all searches on linked lists are O(N).

findValue

A linear search can be done with a simple loop, using a variable that starts at the link pointed to by **head** and is then updated to walk down the list until the terminating **None** is found.

```
# return True if value is present in the list
def findValue(self, value):

   # start at the head
   current = self.__head

   # continue until the end
   while current is not None:

      # found It, go home happy
      if current.getValue() == value:
         return True

      # not found, continue with next link
      current = current.getNext()

   # end of list, nothing found
   return False
```

findRemove

To delete a link containing a value, the simplest way is to find it and delete it. The basic search is similar in nature; simply add code to delete the link when it is found instead of just returning **True**. The problem is, when searching, current points to the link being examined and, unfortunately, when deleting a link, it is necessary to update the link before it in the linked list.

The way to solve this problem is to have two variables that walk down the linked list in tandem. Current tracks the link to change and next looks at the link pointed to by current which is the one

being compared to the search value. Note that it is necessary to create a special case for the situation when the link to remove is the first one in the list.

```
# return True if value is present in the list
# and remove that link from the list
def findRemove(self, value):

   # special case empty list
   if self.isEmpty():
      return False

   # special case head is value
   if self.__head.getValue() == value:
      self.removeHead()
      return True

   # now walk down list, starting at the first link after head
   # at each link, you are actually checking the next one
   current = self.__head
   next = current.getNext()

   # while the next link exists, check it out
   while next is not None:

      # found it, now update linked list to delete it
      if next.getValue() == value:
         current.setNext(next.getNext())
         return True

      # not found, keep looking
      current = next
      next = next.getNext()

   return False
```

Compare this with an array where it is necessary to move all values with larger indexes down one to remove the desired value. In both cases, the search is O(N). In a linked list, it only takes a single step to remove the link and compress the linked list, while in an array it takes O(N) steps to move the remaining items. So, even if they are the same order, the linked list is faster.

Displaying the Linked List

For debugging and understanding purposes, here is a simple function that returns a printable version of the linked list and the values that are stored in it.

```
# return a printable representation of the list
def __repr__(self):
   current = self.__head
   nodes = []
   while current is not None:
      nodes.append(current.getValue())
      current = current.getNext()
   nodes.append("None")
   return " -> ".join(nodes)
```

Adding to Tail

A useful extension to the above class is to add a variable, **tail**, that points to the last link in the linked list. This allows adding a value at the end of the list without walking down the entire linked list to get there.

Note that four methods need to be modified to do this; __init__ to set the new variable, addHead to set tail on an empty linked list, removeHead to set tail when removing the last item from a linked list, and the new method addTail. Note that all methods have been updated to use the isEmpty method.

```
    # the only class variables are head and tail, they start out as None
    def __init__(self):
        self.__head = None
        self.__tail = None

    # add a new link containing value at the head of the list
    def addHead(self, value):

        temp = Link(value)

        if self.isEmpty():
            self.__tail = temp
        else:
            temp.setNext(self.__head)

        self.__head = temp

    # remove the first link in the list if not empty
    # if this is the last link, update tail
    def removeHead(self):

        if self.isEmpty():
            # raise exception

        self.__head = self.__head.getNext()

        if self.isEmpty():
            self.__tail = None

    # add a new link containing value at the tail of the list
    # if this is the first link, update head also
    def addTail(self, value):

        temp = Link(value)

        if self.isEmpty():
            self.__head = temp
        else:
            self.__tail.setNext(temp)

        self.__tail = temp
```

FIFO and LIFO

Two common data structures are a stack (LIFO) and a queue (FIFO). In a stack, items are added in such a way that only the most recently added item can be retrieved, then the previously added one, and so forth. Imagine a stack of plates where plates can be added to the top of the stack or removed from the top, but not inserted in the middle. In a queue, items are added at the end of the queue and removed from the front, for example the line of people waiting to check out in a store.

With a double-ended, single-linked list as described here, it is trivial to implement either of these two classes using class composition. Notice that the only difference between them is where new items are added. The Stack adds them to the head and the Queue adds them to the tail.

```
class Stack:
   def __init__(self):
      self.__theList = LinkedList

   def addItem(self, value):
      self.__theList.addHead(value)

   def getItem(self):
      return self.__theList.getHead()

   def removeItem(self):
      self.__theList.removeHead()


class Queue:
   def __init__(self):
      self.__theList = LinkedList

   def addItem(self, value):
      self.__theList.addTail(value)

   def getItem(self):
      return self.__theList.getHead()

   def removeItem(self):
      self.__theList.removeHead()
```