For this lab, our task was to sort a linked list with different sorting algorithms, and to return the element in the middle after sorting the list. The code to create the list, and to execute some basic functions for a linked list were already given by the professor. The first thing I had to do, was to fill the linked with random numbers, so I would have something to sort, otherwise, I would have a linked list already sorted. Then, for the sorting algorithms, I just had to follow the logic of each of them. Finally, I created a method to return the middle element after the list is sorted.

For the sorting algorithms, I decided that it would be better if I created individual functions for each of them. To implement each of the, I just tried to follow the logic behind them.

**Bubble sort:** For example in bubble sort, I know it is supposed to compare the current element with the next one to see if they are in order, and swap them in case they are not, and repeat that process until the list is sorted. Knowing this, I first checked for an empty list, or for a list with only one element. After that, I just have a while loop to execute the comparison and swapping process. I also have a Boolean variable that will repeat the process in the entire list until it is completely sorted, otherwise the process will only be executed once.

**Merge sort:** For this one, I know that the list needs to be divided in half, until we have lists with only one element, then it compares the individual elements, and puts them in order into the final list. In this case, I first divided the original list into two lists, then, I send those two lists to this same function, and I store the results in two new other lists, finally I compare the results of these new lists, to see if they where should they go, and I append them to the final list.

**Quick sort:** In this algorithm, a pivot is selected, usually is the element in the middle, then we make new lists out of the elements around the pivot (one for the ones at the right, and one for the ones at the left), we repeat the process until we have lists with only one element. Then, we can put them back in order, based on the pivot that will be added to one of the lists, until we have our final ordered list. For the code, I first select a pivot, in this case I decided to use the head of the list because is more efficient than getting the element in the middle, in the end, both the head and the element in the middle are random numbers. After that, I create two new lists, one for elements bigger than the pivot, and the other one for elements smaller than the pivot. Then I repeat this process, and append my results in the final list.

To test these functions I only generated my linked list with random numbers, and I passed this list to the method I wanted to test, printing the list before and after the sorting algorithm, and returning the element in the middle, if the list is sorted, and the element returned is the real middle element, then I know the method is completed.

From doing this lab I better understood how do linked list work in python, and how to understand the. I also learned how to implement different sorting algorithms that can be useful in the future.

**Source code:**

|  |
| --- |
| from random import random |
|  |  |
|  | #Node Functions |
|  | class Node(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self, item, next=None): |
|  | self.item = item |
|  | self.next = next |
|  |  |
|  | #List Functions |
|  | class List(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self): |
|  | self.head = None |
|  | self.tail = None |
|  |  |
|  | #Returns the element in the middle possition of a linked list after being sorted |
|  | def MidPoint(L): |
|  | t=L.head |
|  | len=getLength(L) |
|  | for i in range(len): |
|  | if i==int(len/2): |
|  | return(t.item) |
|  | t=t.next |
|  |  |
|  | #This method makes an exact copy of a given linked list |
|  | def CopyList(L): |
|  | NewList = List() |
|  | t = L.head |
|  | while t is not None: |
|  | Append(NewList, t.item) |
|  | t = t.next |
|  | return NewList |
|  |  |
|  | #Checks is a linked list is empty |
|  | def IsEmpty(L): |
|  | return L.head == None |
|  |  |
|  | def Append(L,x): |
|  | # Inserts x at end of list L |
|  | if IsEmpty(L): |
|  | L.head = Node(x) |
|  | L.tail = L.head |
|  | else: |
|  | L.tail.next = Node(x) |
|  | L.tail = L.tail.next |
|  |  |
|  | def PrintList(L): |
|  | # Prints list L's items in order using a loop |
|  | temp = L.head |
|  | while temp is not None: |
|  | print(temp.item, end=' ') |
|  | temp = temp.next |
|  | print() # New line |
|  |  |
|  | #Returns the length of a linked list |
|  | def getLength(L): |
|  | if IsEmpty(L): |
|  | return 0 |
|  | temp=L.head |
|  | count=0 |
|  | while temp is not None: |
|  | count+=1 |
|  | temp=temp.next |
|  | return count |
|  |  |
|  | #Bubble sort |
|  | def BS(L): |
|  | keepDoing=True |
|  | while keepDoing: |
|  | t=L.head |
|  | while t.next is not None: |
|  | if t.item>t.next.item: |
|  | temp=t.item |
|  | t.item=t.next.item |
|  | t.next.item=temp |
|  | keepDoing=False |
|  | t=t.next |
|  | keepDoing=not keepDoing |
|  | PrintList(L) |
|  | t=L.head |
|  | len=(getLength(L)) |
|  | for i in range(len): |
|  | if i==int(len/2): |
|  | print(t.item) |
|  | t=t.next |
|  |  |
|  | #Merge sort |
|  | def MS(L): |
|  | len = getLength(L) |
|  | if len <= 1: |
|  | return L |
|  | L1 = List() |
|  | L2 = List() |
|  | t = CopyList(L) |
|  | count=0 |
|  | for i in range(len//2): |
|  | Append(L1, t.head.item) |
|  | t.head = t.head.next |
|  | count+=1 |
|  | while count < len: |
|  | Append(L2, t.head.item) |
|  | t.head = t.head.next |
|  | count+=1 |
|  | LeftL = List() |
|  | LeftL = MS(L1) |
|  | RightL = List() |
|  | RightL = MS(L2) |
|  | TotalL = List() |
|  | while getLength(TotalL)!=len: |
|  | if IsEmpty(RightL): |
|  | Append(TotalL, LeftL.head.item) |
|  | LeftL.head = LeftL.head.next |
|  | elif IsEmpty(LeftL): |
|  | Append(TotalL, RightL.head.item) |
|  | RightL.head = RightL.head.next |
|  | elif RightL.head.item < LeftL.head.item: |
|  | Append(TotalL, RightL.head.item) |
|  | RightL.head = RightL.head.next |
|  | else: |
|  | Append(TotalL, LeftL.head.item) |
|  | LeftL.head = LeftL.head.next |
|  | return TotalL |
|  |  |
|  | #Quick sort |
|  | def QS(L): |
|  | if getLength(L)<=1: |
|  | return L |
|  | piv = L.head.item |
|  | L1 = List() |
|  | L2 = List() |
|  | t = L.head.next |
|  | while t is not None: |
|  | if t.item < piv: |
|  | Append(L1, t.item) |
|  | else: |
|  | Append(L2, t.item) |
|  | t = t.next |
|  | SmallSide = List() |
|  | SmallSide = QS(L1) |
|  | BigSide = List() |
|  | BigSide = QS(L2) |
|  | t2 = List() |
|  | if IsEmpty(SmallSide): |
|  | Append(t2, piv) |
|  | t2.head.next = BigSide.head |
|  | t2.tail = BigSide.tail |
|  | return t2 |
|  | elif IsEmpty(BigSide): |
|  | Append(SmallSide, piv) |
|  | return SmallSide |
|  | else: |
|  | Append(SmallSide, piv) |
|  | SmallSide.tail.next = BigSide.head |
|  | SmallSide.tail = BigSide.tail |
|  | return SmallSide |
|  |  |
|  | #Modified quick sort |
|  | def MQS(L, median): |
|  | piv = L.head.item |
|  | L1 = List() |
|  | L2 = List() |
|  | t = L.head.next |
|  | while t is not None: |
|  | if t.item < piv: |
|  | Append(L1, t.item) |
|  | else: |
|  | Append(L2, t.item) |
|  | t = t.next |
|  | if getLength(L1)<median: |
|  | return MQS(L2, median-getLength(L1)-1) |
|  | elif getLength(L1)>median: |
|  | return MQS(L1, median) |
|  | else: |
|  | return piv |
|  |  |
|  | L = List() |
|  | for i in range(5): |
|  | x=int(random()\*10) |
|  | Append(L,x) |