CS 2302 Data Structures

Lab Report #2

Due: October 2nd, 2020
Professor: Olac Fuentes
TA: Anindita Nath
Student: Manuel Gutierrez Jr

Introduction

For this lab, we were asked to work with recursion, Linked List, quicksort, selection sort, time complexity and code methods that use this data structure to print out specific elements. For this lab, it is key to remember the basics of how Linked List references work and how you can use them to implement different sorting algorithms, It is also important to know how to implement quicksort and selection sort to work with the following programs. Link list are required for all the lab and quicksort makes half of the lab and selection sort is the other half. The main objective of this lab is to get a deeper understanding of how references work and how different sorting algorithms have different time complexity and being able to implement linked list and sorting algorithms to get a specific node

Proposed Solution Design and Implementation

Operation #1:

For this operation I used a zybooks example to reference back selection sort as it had been quite a while since I implemented it. Selection sort works by constantly swapping the lowest value to the left most place in the sorted array. For example when taking the numbers 2,5,1,0 you would swap 0 with 2 because it is the least number and 2 is the leftmost place. I continued this till the first value in the array was sorted correctly and no more values needed to be swapped. The question was to get a value k where k represented the kth value from the end. This part was rather tricky and I could not find a way to implement it as part of the selection sort so I created a helper method that would count the length of the list of L and then run another pass where it would stop in when the for loop iterated to match length of the list - k. This approach works but you need to iterate through the list twice to get the length of the link list and once to get the value k over all the time complexity of it is n^2 + 2n or O(n^2).

Operation #2:

For this solution my approach was the exact same as question 1. I sorted the list because it made it easier on my to delete the nodes at the beginning as an example i would take in a list of 2,4,3,1 and sort it with selection sort from there i would have a sorted list of 1,2,3,4 from there i would change the reference of the head k + 1 times and return the last deleted node. This was rather simple when i had a sorted list as the least element is the beginning. Unfortunately this does take n^2 + n times because it needs to be sorted (it could take n log n if it was sorted with .sort but i had already implemented selection sort and it would be a waste not to use already written code.

Operation #3:

Quick Starting a linked list was excruciating because quicksort works best on list because of its randomness to divide and concor. So I did what any sane computer science did. I disregarded time complexity and implemented a helper function that could turn a linked list into an array. I then implemented quicksort to deal with the list and it became easier to implement, debug and overall better in my opinion though this did cost me the number of elements in the list. This is a huge disadvantage as a linked list could have a thousand elements and it would take 100 times to convert to a an array and n log n times to sort it with quicksort and then turn it into a link list n times over all the time complexity s still O(n log n) but in reality it is 2n + n log n. After that i made a helper method that could give you the element k by having a counter that traverses the linked list and then traverses it again with the length of the list - k just as question 1 would do

Operation #4:

This problem was rather difficult and I tried my best. What I did was partition it only once with a while loop to technically partition it multiple times and I started with the end to the start in hopes that I could partition it fasted by concerning the end to the start. I only make one recursive call so there are less activation records. I partitioned it with length /2 because that is what the pdf recommended and Sadly I don believe it worked because it was stuck inside an infinite loop and I did not have enough time to debug so I left it with the assumption that it did work when only calling it once.

Experimental Results

Operation #1,#2#3,#4:

For this operation, I used Dr.fuentes code with a given random list and with that random link list I ran my experiments by determining the kth element from the end. I used k equal to zero as the Pl told me to just use my own design for testing as no other code was provided and I figured it was the easiest to test. I also used a print statement to print the given list and check my answers. Additionally Dr.fuentes code made sure that all the other code was working and provided a results if all the numbers matched it was correct this is from what i understood. The reason I say all experimental results where the same is because they are. They are a repetition of using Dr.fuentes code to implement linked list and then test it with print statements

Conclusion

This lab helped me have a better understanding of how linked lists behave and how references are used in dealing with linked lists. There were some cases where having a link list was difficult such as in quicksort it is rather difficult to use references and a list of elements is better when dealing with this though it may add extra time and space complexity the implementation is easier. Additionally I learned the importance of having the sorting algorithms memorized by heart and knowing how to implement them i spent some time remembering them while looking at the zybooks for the class. They do have very different implementations when dealing with them in a link list. I think all sorting algorithms have an asier implementation when it is an array of elements or a list of elements rather than a referenced based data structure like a linked list. Over all I did learn a vast amount of knowledge from problem 4 as a I never knew that by using while loops instead of recursion I could shorten the time complexity of an algorithm if it would have been recursively only.

Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import time
import singly_linked_list as sll

def index_of(L,k):
   count = 0
   t = L.head
   while t!=None:
       if t.data == k:
           return count
       t = t.next
       count +=1
   return -1

def random_list(n):
   L = sll.List()
   L.extend(list(np.random.randint(0, high=10*n, size=n, dtype=int)))
   return L

def length_of_L(L,k):
   count = 0
   t = L.head
   while t!=None:
       t = t.next
       count +=1
   return element_k(L,count-k)
```

```python
def element_k(L,count):
   t = L.head
   tempCount = 1
   while t!= None:
       if(tempCount == count):
           return t.data
       tempCount+=1
       t = t.next
   return -1


def selectionsort(L):
   temp = L.head
   while (temp != None):
       least_in_list = temp
       next_element = temp.next

       while (next_element != None):
           if (least_in_list.data > next_element.data):
               least_in_list = next_element
           next_element = next_element.next

       swap_variable = temp.data
       temp.data = least_in_list.data
       least_in_list.data = swap_variable
       temp = temp.next

   return L

def select_selectionsort(L,k):
   L = selectionsort(L)
   return length_of_L(L, k)

def select_min(L,k):
   L = selectionsort(L)
   replaceCounterForK = k
   prev = L.head
   current = L.head.next
```

```
   while(replaceCounterForK != 0):
       L.head,prev = None,current
       current = current.next
   return current.data

'''
This was my attempt to quicksorting with a link list and it will take a long tiem
because a link list has pointers and quicsort is rather random
def partician(start,end):
   if(start == end or start == None or end == None):
       return start

   pivot_prev = start
   current_node = start
   pivot = end.data
   counter = start.data

   while(start != end):
       if(pivot > counter):
           pivot_prev.data,current_node.data,start.data =
current_node,start,current_node
       start = start.data

   temp = current_node
   current_node.data = pivot
   end.data = temp

   return pivot_prev

def quicksort(start,end):
   if(start == end):
       return

   pivot_prev = partician(start, end)
   quicksort(start, pivot_prev)

   if(pivot_prev != None and pivot_prev == start):
       quicksort(pivot_prev.next, end)
```

```
   elif(pivot_prev != None and pivot_prev.next != None):
       quicksort(pivot_prev.next.next, end)

   return
'''

def partition(array, start, end):
   pivot = array[start]
   low = start + 1
   high = end

   while start!= 0:
       while low <= high and array[high] >= pivot:
           high = high - 1
       while low <= high and array[low] <= pivot:
           low = low + 1
       if low <= high:
           array[low], array[high] = array[high], array[low]
       else:
           break
   array[start], array[high] = array[high], array[start]
   return high

def quicksort(arr, start, end):
   if start >= end or start == 0 or end ==0:
       return

   part = partition(arr, start, end)
   quicksort(arr, start, part-1)
   quicksort(arr, part+1, end)

def select_quicksort(L,K):
   arr = []
   t = L.head
   while(t != None):
       arr.append(t.data)
       t = t.next

   quicksort(arr,0,len(arr)-1)
```

```
    L = sll.List()
    for i in arr:
        L.append(i)
    return length_of_L(L, k)

def select_modified_quicksort(L,k):
    arr = []
    t = L.head
    while(t != None):
        arr.append(t.data)
        t = t.next
    quicksort_one_call(arr,0,len(arr)-1)

    L = sll.List()
    for i in arr:
        L.append(i)
    return length_of_L(L, k)


def quicksort_one_call(arr,start,end):
    while(start < end):
        part = partition(arr, end, start)
        quicksort_one_call(arr, end, part)
        end = part+1


if __name__ == "__main__":
    reps = 3
    first_n, last_n, step_n = 10, 20, 1
    times, sizes = [], []
    for n in range(first_n, last_n, step_n):
        sum_time = 0
        results = []
        for r in range(reps):
            np.random.seed(seed=n+r) # To obtain the same results in every experiment
            L = random_list(n)
            start = time.time()
            index = index_of(L,2302)
```

```
            results.append(index)
            #select_selectionsort(L,k). Sort L using selection sort, then return the
element in position k.
            k = 0
            kth_sorting_smallest = select_selectionsort(L, k)
            print('kth_smallest element in list using
select_selectionsort',kth_sorting_smallest)
            kth_smallest = select_min(L,k)
            elapsed_time = time.time() - start
            print('kth_smallest element in list using select_min',kth_smallest)
            sum_time += elapsed_time
            kth_smallest_quicksort = select_quicksort(L, k)
            print("Kth_sallest elemtn in the list using
quicksort",kth_smallest_quicksort)
            #kth_smallest_select_modofied_quicksort = select_modified_quicksort(L, k)
            #print("Kth_sallest elemtn in the list using select odofied
quicksort",kth_smallest_select_modofied_quicksort)
        times.append(sum_time/reps) # Display average time per repetition
        sizes.append(n)
        print('List length: {:3}, running time: {:7.5f}
seconds'.format(sizes[-1],times[-1]))
        print('Results:',results) # Print results to verify that all algorithms return
the same value for the same input

    #plt.close('all')  # Uncomment to close all previous figures prior to drawing a new
one
    fig, ax = plt.subplots()
    plt.plot(sizes,times)
    ax.set_xlabel('n')
    ax.set_ylabel('running time (seconds)')
    fig.suptitle('Running time for index_of function', fontsize=16)  # Replace by your
fuction's name
```

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class