ECS 32B: Conceptual Homework #2

Instructor: Aaron Kaloti

Summer Session #2 2020

1 Identification

Enter the members of your pair. (You can partner with at most one other student.) If you are not partnered with anyone, then leave the second box empty. You can remove the use of `\vspace` in the `.tex` file.

Pair member #1:

Zhikuan Quan (ID:917800911)

Pair member #2:

Bohao Zou (ID: 917796070)

2 Problems

2.1 Linked List Indexing

Explain why linked lists do not support constant time indexing.

Indexing the linked lists needs linear time. This is because the linked list is unordered in the memory. It needs to find the element one by one by traversing all the elements from the first one to the one with specific index.

2.2 Unordered List `add()`

Consider the implementation of `add()` for the linked list implementation of an unordered list on slide #30 of the linear data structures slide deck. Could we flip the order of the lines `temp.setNext(self.head)` and `self.head = temp`? If *not*, explain.

No, if we flip the order of the lines, the head point will direct to the temp node itself. The function `setNext()` will set the next node to the head node. So, we can not get the next node if we flip.

2.3 Stack-Less Balanced Symbols

2.3.1 Balanced Parentheses

Recall the balanced parentheses example from the lecture on stacks (and from your primary textbook). Could we have replaced the stack with a single integer variable instead? If so, explain how we would adjust the approach to the balanced parentheses problem (i.e. when and how would the integer variable change? how can it be used to detect that the given string of parentheses is unbalanced?). If not, explain why.

Yes, we can use a single integer variable instead. If we detect the "(", we add the integer 1. if we detect ")", we add the integer -1. If the integer is 0 but we do not iterate parentheses string over, the parentheses string is not balance. If we iterate parentheses string over but the integer variable is not 0, the parentheses string is not balanced. If we iterate parentheses string over and the integer is 0, it is a balanced string.

2.3.2 Balanced Symbols

Repeat the previous problem, except for the more general balanced symbols example (that you were supposed to read about in section 4.7 of the primary textbook) instead.

```
def judge(string):
    """
    This function is to judge if the symbol string is balanced.
    balanced string: (), {}, [], (){}[], []{}.
    unbalanced string: (}, ({)}, [](((( 

    :param string: The input symbol string
    :return: If the symbol string is balanced.
    """
    jNum = 0
    length = len(string) - 1
    for i,s in enumerate(string):
        if s == "(":
            jNum += 1
            if (i + 1) <= length and string[i + 1] != ")":
                return False
        elif s == ")":
            jNum -= 1
        elif s == "[":
            jNum += 2
            if (i + 1) <= length and string[i + 1] != "]":
                return False
        elif s == "]":
            jNum -= 2
        elif s == "{":
            jNum += 3
            if (i + 1) <= length and string[i + 1] != "}":
                return False
        else:
            jNum -= 3
    if jNum == 0:
        return True
    else:
        return False
```

In the code above, we can use single integer jNum to check the balanced symbols.

2.4 Binary Search on an Ordered List

When we talked about ordered lists during lecture, we added a small speed-up to the `search()` method that did not affect the worst-case time complexity. Why did we not use binary search? Should we have? If we had implemented the ordered list with a Python list instead of a linked list, could we have used binary search there instead/too? How, if at all, would that have improved the worst-case time complexity of `search()`?

Why did we not use binary search? This is because if we need to find the medium element, we need to iterate all the elements in the linked list. So, we do not need to use binary search.
We could use binary search if we implement ordered list with a python list. This will improved the worst-care time complexity to $log(n)$.

2.5 Binary Search Midpoint Calculation

In our Python implementation of binary search, why do we not calculate `midpoint` by using the formula `midpoint = last // 2`.

It will work in the first time judgement. However, during the second judgement, if we judge the target element in the right side of the list, the value `last` would not change. The value `first` changed to `midpoint + 1`. It is a wrong mid value in the second judgement if using last //2 (which is not the midpoint of right-half list).

2.6 Unordered List `remove()`

2.6.1 Worst-Case Time Complexity

As stated on slide #40, if we use a Python list as an unordered list (think of just a normal Python list of integers, with nothing special about it), `remove(item)` takes linear time, *regardless* of the location of `item`. Why does the location of `item` not matter? Why would removing the element at the front of the unordered list (again, represented as a Python list) take about as long (i.e. linear time) as removing the element at the end?

The location of item does not matter because it is an unordered list, which means we must do search processing and the search process will take linear time to iterate. In this case, removing the element at the front takes same time as removing the element at the end.

2.6.2 Removing the Last Node

Explain how the `remove()` method for the linked list representation of an unordered list (see slide #37 of the linear data structures slide deck) works if the node-to-remove is the last node in the linked list.

For the last node, we have self.next = None. If the target is the last one then we will directly link the previous one to None.

2.6.3 Removing the Only Node

Consider the same implementation of `remove()` that was considered in the previous question. Explain how that implementation works if the node-to-remove is the *only node* in the linked list.

For the only node, we will set the point of head to Null Point (self.head = None).

2.7 A More Time-Efficient Queue

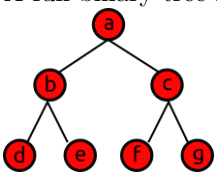
Recall that for the implementation of a queue that we examined during lecture, `enqueue()` runs in linear time in the worst case. Suppose we wanted to implement a queue (i.e. a queue class) such that *all* of its operations, including `enqueue()`, take constant time, and suppose that we were assured that never more than 100 elements would be inserted into an instance of this queue *ever*. For example, we could see that 50 elements are inserted into an instance of the queue, followed by 4 pops, and then followed by 50 more insertions, but after this, we are assured that there won't be a 101st insert. How can we implement such a queue that has constant time operations? You should explain your approach and explain how each of `enqueue()`, `dequeue()`, and `size()` would be implemented and how they each take constant time. Explain how your approach might work in a small hypothetical scenario, e.g. a mix of 1-2 calls to `enqueue()` and 1-2 calls to `dequeue()`.

Hint: Use a Python list as the underlying implementation. Take advantage of a concept called "lazy deletion". To "lazily delete" from a Python list is to "trick" your code into thinking that an element was deleted (usually from the front or back) of the list by changing some variables in your code that keep track of where the start or end of the Python list is.

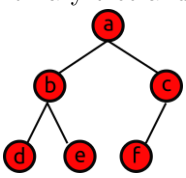
```
class Quene:

    def __init__(self):
        self.items = [] # initialization
        self.start = 0 # the start point of the list
        self.num = 0 # the number(size) of items in list after the start point

    def enqueue(self, item):
    """
    Adding the new item at the end of the list.
    (.append in python list takes constant time in the worst case)
    """
        self.items.append(item)
        self.num += 1

    def dequeue(self):
    """
    Returning the item at the start of the list and then move the start point to the next index.
    e.g. the list (1)2345, where () represents the start point. When calling dequeue(), it returns 1
    and then the list becomes 1(2)345 with self.num = 4 representing the lenth of the rest list. If we
     call dequeue() again, it will return 2 and then the list becomes 12(3)45 with self.num = 3.
    (For the python list, indexing the list takes constant time in the worst case)
    """
        if self.num == 0:
            return None
        val = self.items[self.start]
        self.start += 1
        self.num -= 1
        return val

    def size(self):
    """
    Directly return the size of the list (after the start point).
    """
        return self.num
```

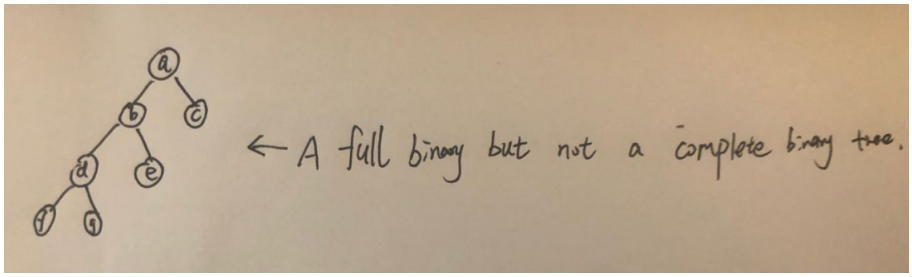
2.8 Full Binary Trees vs. Complete Binary Trees

A full binary tree is a binary tree in which each node that is not a leaf has two children. Below is an example.



A complete binary tree is a binary tree in which every level, except possibly the last (which has all of its node shifted as far left as possible), is filled. Give ***and explain*** an example that shows that a full binary tree is not always complete. (By "explain", I mean that you should explain how your example shows what you want it to show.) If you wish, you can upload an image, even if the image is handwritten, using `includegraphics`; see how it's used in the LaTeX file to include the full binary tree and complete binary tree images, and note that the file extension must be omitted.)





This is a full binary tree since each nodes that are not a leaf have two children. However, it is not a complete binary tree because in the level 1 and 2 are not filled.