COMP2119 Introduction to Data Structures and Algorithms
Assignment 2 - Algorithm Analysis, Data Structures and Hashing

Due Date: Oct 21, 2024 7:00pm

Question 1 - What have You Done? [30%]

One day, handsome Jeff finished grading n assignment scripts. He sorted the scripts in ascending order of UID (assume all UIDs are unique). To inspect his work, Ben read k scripts from the top, and put them back at the bottom of the pile without changing the order of the k scripts.

Suppose the UID of the n scripts are stored in list A as integers with the current placing order. The first element in A represents UID of the script on the top. To arrange the papers back in order, Jeff needs to find out the value of k.

You may assume that n is in power of 2, and $0 \leq k \leq n-1$.

Example:
For input $A_1 = [43, 60, 71, 3, 14, 23, 26]$ and $n = 7$, the output will be $k = 4$. Ben has read scripts with UID 3, 14, 23, 26.

Your Tasks:

(a) [10%] Jeff has written the function reOrder1(A, n) to solve the problem. Explain briefly how his code solve the problem and analyze it's worst case time complexity.

Algorithm 1.1 Pseudocode of reOrder1

```
1:  function reOrder1(A, n)
2:
3:      k = 0
4:      for (i = 0 to n − 2) do
5:          if (A[i] > A[i + 1]) then
6:              k = n − i − 1
7:          end if
8:      end for
9:
10:     return k
11:
12: end function
```

(b) [20%] If n is large, it would be a nightmare to run Jeff's algorithm by human efforts. Write the pseudocode of another algorithm reOrder2(A, n) to solve the problem. The algorithm must be asymptotically faster than reOrder1(A, n).

Briefly explain what the code does, and analyze the algorithm's worst case time complexity.

Solution:

(a) Explanation (5%):
The algorithm iterates through the list and compare the value between the current element and it's next one. If the value of the next element is smaller than the current element, we have found k by subtracting n with the index of the next element.

Time complexity (5%):
The time complexity would be $O(n)$.

The algorithm iterates through the list for n times. In each iteration, the operations cost $O(1)$ time. As a result, the for loop runs for $O(n)$ time. The operations outside the for loop runs for $O(1)$ time. Summing up, the overall worst case time complexity is $O(n)$.

(b) Pseudocode & Explanation (15%):

Algorithm 1.2 Pseudocode of reOrder2

```
1:  function reOrder2(A, n)                    ▷ Find max gain in subarray S[left : right]
2:
3:      k = 0
4:      b = n/2
5:
6:      if (n == 1) then                                              ▷ Base Case
7:          return 0
8:      end if
9:
10:     if (A[0] > A[b]) then                  ▷ Find k recursively in one half of A
11:         k = b + reOrder2(A[0 : b], b)
12:     else
13:         k = reOrder2(A[b : n], b)
14:     end if
15:
16:     return k
17:
18: end function
```

The algorithm compares the value between the element at the beginning and the middle of the list. If the middle one is smaller, it adds up half length of the list to k and recursively apply the algorithm in the first half of the list (i.e. at least half of the papers in the list is read by Ben). Or else, it continue the search in the later half of the list. For the base case, it returns 0 as the answer because there is only one element in the list.

Time complexity (5%):
Let $T(n)$ be the running time of the algorithm.

$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + c_1 \text{ , where } T(1) = c_2 \\
&= T(\frac{n}{4}) + c_1 + c_1 \\
&= T(1) + \log n(c_1) \\
&= c_2 + c_1 \log n \\
&\rightarrow O(\log n)
\end{aligned}
$$

Question 2 - Linked List [30%]

Implement a linked list and solve the following sub-problems with your implementation. Feel free to select any programming language you prefer.

(a) [15%] Merging: Implement a function that takes two sorted linked lists and merges them into a single sorted linked list.

(b) [15%] Cycle detection: A cycle in a linked list occurs when a node's next pointer points back to a previous node, creating a loop, while the beginning of a cycle is the first node in that loop where traversal starts repeating.

Implement a function that detects if there is a cycle in the linked list. If a cycle exists, it returns the node where the cycle begins.

Solution:

```
class Node:
    def __init__(self, data):
        # Initialize a node with data and a pointer to the next node
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        # Initialize the linked list with a head pointer
        self.head = None

    # Sub-problem 1: Merge Two Sorted Linked Lists
    def merge_sorted_lists(self, list2):
        # Create a dummy node to simplify merging process
        dummy = Node(0)
        tail = dummy  # This will point to the last node in the merged list

        # Pointers for both lists
        l1, l2 = self.head, list2.head

        # Traverse both lists and merge them based on their values
        while l1 and l2:
            if l1.data < l2.data:
                # If current node in list1 is smaller, attach it to the merged list
                tail.next = l1
                l1 = l1.next  # Move to the next node in list1
            else:
                # If current node in list2 is smaller or equal, attach it to the merged list
                tail.next = l2
                l2 = l2.next  # Move to the next node in list2
            tail = tail.next  # Move the tail pointer to the last node

        # Attach the remaining nodes from either list
        tail.next = l1 if l1 else l2
        return dummy.next  # Return the merged list (skipping the dummy node)

    # Sub-problem 2: Detect a Cycle in the Linked List
    def detect_cycle(self):
        # Use two pointers: slow and fast
        slow = fast = self.head
```

```
        # Traverse the list with slow moving 1 step and fast moving 2 steps
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

            # If they meet, a cycle exists
            if slow == fast:
                # Reset slow to find the start of the cycle
                slow = self.head
                while slow != fast:
                    slow = slow.next
                    fast = fast.next
                return slow  # Return the node where the cycle begins
        return None  # No cycle found

# Example Usage
# Create first sorted linked list: 1 -> 3 -> 5
list1 = LinkedList()
list1.append(1)
list1.append(3)
list1.append(5)

# Create second sorted linked list: 2 -> 4 -> 6
list2 = LinkedList()
list2.append(2)
list2.append(4)
list2.append(6)

# Merge the two sorted lists
merged_list = list1.merge_sorted_lists(list2)
current = merged_list
print("Merged Sorted List:")
while current:
    print(current.data, end=" -> ")
    current = current.next
print("None")

# Detect cycle
cycle_list = LinkedList()
cycle_list.append(1)
cycle_list.append(2)
cycle_list.append(3)
cycle_list.head.next.next.next = cycle_list.head  # Creating a cycle

cycle_node = cycle_list.detect_cycle()
print("Cycle starts at node:", cycle_node.data if cycle_node else "No cycle")
```

Question 3 - Maximum Frequency Stack [20%]

Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack.

Specifically, for a *FreqStack* class: *FreqStack*() constructs an empty frequency stack (called *fstack*). *fstack.push*(*val*) pushes an integer *val* onto the top of the stack. *fstack.pop*() removes and returns the most frequent element in the stack. If there are multiple different elements of the same highest frequency, the element closest to the stack's top is removed and returned.

Suppose at most N elements will be inserted into *fstack* and each *val* is a non-negative integer smaller than 10.

Example:

```
FreqStack fstack = new FreqStack();
fstack.push(5); // The stack is [5]
fstack.push(7); // The stack is [5,7]
fstack.push(5); // The stack is [5,7,5]
fstack.push(7); // The stack is [5,7,5,7]
fstack.push(4); // The stack is [5,7,5,7,4]
fstack.push(5); // The stack is [5,7,5,7,4,5]
fstack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,5,7,4].
fstack.pop(); // return 7, as 5 and 7 are the most frequent, but 7 is closest to the
    top. The stack becomes [5,7,5,4].
fstack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,4].
fstack.pop(); // return 4, as 4, 5 and 7 are the most frequent, but 4 is closest to
    the top. The stack becomes [5,7].
```

Your Tasks:

(a) [15%] Implement the *FreqStack* class as described above.

(b) [5%] Analyse the worst-case time complexity of *fstack.push* and *fstack.pop* of your implementation.

Solution:

(a)
```python
# Use a hash table to record the frequency of each possible val
class FreqHashTable:
    def __init__(self) -> None:
        # Initialize frequency of each possible val to zero
        self.frequency: List[int] = [0] * 10

    def insert(self, val: int) -> None:
        # Insert causes increase of recorded frequency by 1
        self.frequency[val] += 1

    def delete(self, val: int) -> None:
        # Delet causes decrease of recorded frequency by 1
        self.frequency[val] -= 1

    def search(self, val: int) -> int:
        # Return the recorded frequency
```

```
            return self.frequency[val]


    # A standard stack that stores val in a list
    class Stack:
        def __init__(self) -> None:
            self.data = [0] * N
            self.tos = -1

        def push(self, val: int) -> None:
            self.tos += 1
            self.data[self.tos] = val

        def pop(self) -> int:
            self.tos -= 1
            return self.data[self.tos + 1]

        def empty(self) -> bool:
            return self.tos >= 0

    class FreqStack:
        def __init__(self) -> None:
            self.freq = FreqHashTable()
            # Use a hash table to access each stack;
            # Each stack stores elements of the same frequency
            self.stack_hash_table = [Stack() for i in range(N)]
            self.max_freq = 0

        def push(self, val: int) -> None:
            # Calculate and record the frequency of current val
            self.freq.insert(val)
            # Get the frequency of val
            current_freq = self.freq.search(val)
            # Push val to the stack whose elements share the frequency of current_freq
            self.stack_hash_table[current_freq].push(val)
            # Calculate maximum frequency so that we can easily access the stack
            # with maximum frequency
            self.max_freq = max(self.max_freq, current_freq)

        def pop(self) -> int:
            # Access the stack with maximum frequency and pop an element from its top
            val = self.stack_hash_table[self.max_freq].pop()
            # Re-calculate the frequency of val
            self.freq.delete(val)
            if self.stack_hash_table[self.max_freq] == 0:
                self.max_freq -= 1
            return val
```

(b) Time complexity of *fstack.push* & *fstack.pop* of the above implementation are both $O(1)$.

Question 4 - Hash Table [20%]

Insert the keys $\{17, 94, 86, 22, 98, 79, 54, 38\}$ into the following hash tables.

For each case, calculate the average number of slots inspected for an unsuccessful search after the keys are inserted (An empty slot or a NIL pointer access should also be counted as one inspection.). Assume that each slot in a hash table has equal chance to be accessed at the first probe in the search process.

(a) [5%] A hash table of size 7 with a single hash function of $h(x) = x \bmod 7$. Collisions are handled by chaining.

(b) [7%] A hash table of size 11 with a single hash function of $h(x) = x \bmod 11$. Collisions are handled by open addressing with linear probing, using the function $f(i) = 3i$.

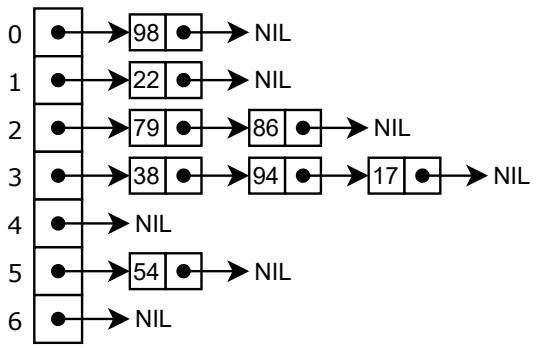
(c) [8%] A hash table of size 11 with double hashing using the probe sequence $h(k, i) = (h_1(k) + i\ h_2(k)) \bmod 11$, where $h_1(x) = x \bmod 11$ and $h_2(x) = 2 - (k \bmod 2)$.

Solution:

(a) Calculating hash values:

key	17	94	86	22	98	79	54	38
hash	3	3	2	1	0	2	5	3

Hash table:



Number of inspections for each slot in an unsuccessful search:

slot	0	1	2	3	4	5	6
count	2	2	3	4	1	2	1

The average is $\frac{2+2+3+4+1+2+1}{7} = \frac{15}{7} = 2.14$.

(b) Calculating hash values:

key	17	94	86	22	98	79	54	38
hash	6	6	9	0	10	2	10	5

Hash table:

Number of inspections for each slot in an unsuccessful search (need to probe until reaching an empty slot):

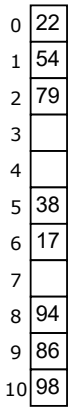
slot	0	1	2	3	4	5	6	7	8	9	10
count	2	2	5	1	1	4	4	1	3	3	6

The average is $\frac{2+2+5+1+1+4+4+1+3+3+6}{11} = \frac{32}{11} = 2.91$.

(c) Calculating hash values:

key	17	94	86	22	98	79	54	38
h_1	6	6	9	0	10	2	10	5
h_2	1	2	2	2	2	1	2	2

Hash table:



Number of inspections for each slot in an unsuccessful search (there are two cases, with equal chances):

slot	0	1	2	3	4	5	6	7	8	9	10
count(h_2=1)	4	3	2	1	1	3	2	1	7	6	5
count(h_2=2)	3	2	2	1	1	2	5	1	4	4	3

The average is $\frac{4+3+2+1+1+3+2+1+7+6+5+3+2+2+1+1+2+5+1+4+4+3}{22} = \frac{63}{22} = 2.86$.

Submission

Please submit your assignment (one **PDF** file) to moodle by the deadline. Make sure the content is readable. Feel free to contact the TAs if you encounter any difficulty in this assignment. We are happy to help you!

Enquiries for Grading

If you have any issues regarding to the grading of the assignment, please contact the following graders for help:

Q1: Jeff Siu (Email: jeffsiu1@hku.hk)
Q2: Ye Tian (Email: yetiansh@connect.hku.hk)
Q3, 4: XiuXian Guan (Email: guanxx@connect.hku.hk)