

COMP2119 Introduction to Data Structures and Algorithms

Assignment 2 - Algorithm Analysis, Data Structures and Hashing

Cheng Ho Ming, Eric (3036216734) [Section 1C, 2024]

October 21, 2024

1. (a) Jeff's code perform a linear search on the list of integers A and finds out the index k of the first element on the list which disrupts the ascending order of the list.

He first initializes an variable k , which is a integer that stores the index of the first element that disrupts the ascending order of the list.

Then, he uses an for-loop with counter i to iterate through the list of integers A from the first element (with index 0) to the second last element (with index $n - 2$). During each iteration, the program compares the current element (i) with the next element ($i + 1$). If the current element is greater than the next element, he assigns the index of the current element to k .

Finally, he returns the value of k .

Let the cost of the k -th line of the code section above be c_k . The worst-case time complexity will be:

$$O(c_3 + c_4 + (n - 2 + 1)(c_5 + c_6)) = O(n)$$

- (b) The pseudocode of another algorithm $reOrder2(A, n)$ is as follows:

```
function reOrder2(A, n)
    left = 0
    right = n-1
    while (left < right - 1) do
        middle = (left+right) div 2 // integer division
        if A[left] < A[middle] then
            left = middle
        end if
        if A[middle] < A[right] then
            right = middle
        end if
    end while
    return right
end function
```

The code performs a binary search over the list of integers A .

The list A can be divided into two separate ascending sequences of numbers. The code will find the index of the last element of the first sequence, store it in variable $left$, and find the index of the first element of the second sequence and store it in variable $right$.

It does this by selecting the element in the middle of the list A , and determining whether such element belongs to the first sequence or the second sequence.

- * If the element belongs to the first sequence, it updates the value of $left$ to the index of the middle element.

- * If the element belongs to the second sequence, it updates the value of $right$ to the index of the middle element.

Ultimately, during each selection, the element selected must belong to either one of the sequences, given the list A is valid.

The maximum number of times we can select the middle element is $\log_2(n)$, before we reach the point that $left$ and $right$ are adjacent to each other. Therefore, the worst case time complexity will be:

$$O(c_2 + c_3 + c_4 + (\log_2(n))(c_5 + c_6 + c_7 + c_9 + c_{10}) + c_{13}) = O(\log_2(n)) = O(\log(n))$$

2. (a) `#include <stdlib.h>`

```
struct Node {
    int data;
    struct Node* next;
};
typedef struct Node Node;

Node* mergeTwoSortedLinkedList(Node* first, Node* second) {
    if (first == NULL) return second;
    if (second == NULL) return first;
    if (first->data < second->data) {
        first->next = mergeTwoSortedLinkedList(first->next, second);
        return first;
    }
    else {
        second->next = mergeTwoSortedLinkedList(first, second->next);
        return second;
    }
}
```

(b) `#include <stdlib.h>`

```
struct Node {
    int data;
    struct Node* next;
};
typedef struct Node Node;

Node* getCycleBeginsPtr(Node* head) {
    if (head == NULL) return NULL;
    Node* one_step = head;
    Node* two_step = head;
    while (two_step != NULL && two_step->next != NULL) {
        one_step = one_step->next;
        two_step = two_step->next->next;
        if (one_step == two_step) {
            Node* cycle_begins = head;
            while (cycle_begins != one_step) {
                cycle_begins = cycle_begins->next;
                one_step = one_step->next;
            }
            return cycle_begins;
        }
    }
    return NULL;
}
```

3. (a) `#include <vector>`

```
#define MAX_VAL 10
class FreqStack {
private:
    std::vector<int> stack;
    int freq[MAX_VAL] = {0};
    int maxFreq(void) {
        int max = 0;
        for (int i = 0; i < MAX_VAL; i++) {
            if (freq[i] > max) {
                max = freq[i];
            }
        }
        return max;
    }
public:
    FreqStack() {
        stack = std::vector<int>();
    }
    void push(int val) {
        stack.push_back(val); // O(1) operation
        freq[val]++;
    }
    int pop(void) {
        int _maxFreq = maxFreq();
        int i = stack.size() - 1;
        while (i > 0) {
            if (freq[stack[i]] == _maxFreq) {
                freq[stack[i]]--;
                break;
            }
            i--;
        }
        int _val = stack[i];
        stack.erase(stack.begin() + i); // O(n) operation
        return _val;
    }
};
```

- (b) In my implementation, the worst-case time complexity of *fstack.push* would be $O(1)$, as both the *stack.push_back* and *freq[val]++* operations are $O(1)$ (constant time) operations.

For the *fstack.pop* function:

```
int pop(void) {
    int _maxFreq = maxFreq(); // O(n) (maxFreq() is an linear search function)
    int i = stack.size() - 1; // O(1)
    while (i > 0) { // O(n)
        if (freq[stack[i]] == _maxFreq) { // O(1) (O(n) inside a while loop)
            freq[stack[i]]--; // O(1) (O(n) inside a while loop)
            break; // O(1) (O(n) inside a while loop)
        }
        i--; // O(1) (O(n) inside a while loop)
    }
    int _val = stack[i]; // O(1)
    stack.erase(stack.begin() + i); // O(n) (erase() involves shifting
    // elements to fill the blank)
    return _val; // O(1)
}
```

Let the cost of the k -th line of the code section above be c_k . The worst case time complexity will

be:

$$O(nc_2 + c_3 + n(c_4 + c_5 + c_6 + c_7 + c_9) + c_9 + c_{11} + nc_{12} + c_{13}) = O(n)$$

4. (a) Inserting $\{17, 94, 86, 22, 98, 79, 54, 38\}$ to the hash table with size 7, using the hash function $h(x) = x \bmod 7$ and collision handling by chaining.

```
# Initial state:
[NIL, NIL, NIL, NIL, NIL, NIL, NIL]
# Insert 17 at  $h(17) = 17 \bmod 7 = 3$ :
[NIL, NIL, NIL, [17], NIL, NIL, NIL]
# Insert 94 at  $h(94) = 94 \bmod 7 = 3$ 
[NIL, NIL, NIL, [17, 94], NIL, NIL, NIL]
# Insert 86 at  $h(86) = 86 \bmod 7 = 2$ 
[NIL, NIL, [86], [17, 94], NIL, NIL, NIL]
# Insert 22 at  $h(22) = 22 \bmod 7 = 1$ 
[NIL, [22], [86], [17, 94], NIL, NIL, NIL]
# Insert 98 at  $h(98) = 98 \bmod 7 = 0$ 
[[98], [22], [86], [17, 94], NIL, NIL, NIL]
# Insert 79 at  $h(79) = 79 \bmod 7 = 2$ 
[[98], [22], [86, 79], [17, 94], NIL, NIL, NIL]
# Insert 54 at  $h(54) = 54 \bmod 7 = 5$ 
[[98], [22], [86, 79], [17, 94], NIL, [54], NIL]
# Insert 38 at  $h(38) = 38 \bmod 7 = 3$ 
[[98], [22], [86, 79], [17, 94, 38], NIL, [54], NIL]
```

As an empty slot or a NIL pointer access should also be counted as one inspection, an unsuccessful search on $h(?) = 0$ would take 2 inspections (1 for accessing 98 and 1 for NIL pointer access), for example.

Average number of slots inspected for an unsuccessful search after the keys are inserted would be:

$$\frac{2 + 2 + 3 + 4 + 1 + 2 + 1}{7} = \frac{15}{7} \approx 2.14 \text{ (correct to 3 significant figures)}$$

- (b) After insertion, the hash table would look like this:

```
[22, 86, 79, NIL, NIL, 54, 17, NIL, 38, 94, 98]
```

The average number of slots inspected for an unsuccessful search after the keys are inserted would be:

$$\frac{2 + 2 + 5 + 1 + 1 + 4 + 4 + 1 + 3 + 3 + 6}{11} = \frac{32}{11} \approx 2.91 \text{ (correct to 3 significant figures)}$$

- (c) After insertion, the hash table would look like this:

```
[22, 54, 79, NIL, NIL, 38, 17, NIL, 94, 86, 98]
```

Since it is a double hash function, the probing sequence can be different even if the first inspected slot is the same.

In this case, we notice that for $x = 0$ or 11, their first inspected slots are both 0 (as $h(0, 0) = h(11, 0) = 0$), but their next inspected slots are $h(0, 1) = 2$ and $h(11, 1) = 1$ respectively.

Looking at the function, we can see the second hash function $h_2(x) = 2 - (x \bmod 2)$ returns 2 if x is even and 1 if x is odd.

Since we are calculating average, we assume x has equal chance (50%) for being even or odd.

The average number of slots inspected for an unsuccessful search after the keys are inserted would be:

$$\begin{aligned} & \frac{\frac{4+3}{2} + \frac{3+2}{2} + \frac{2+2}{2} + 1 + 1 + \frac{3+2}{2} + \frac{2+5}{2} + 1 + \frac{7+4}{2} + \frac{6+4}{2} + \frac{5+3}{2}}{11} \\ &= \frac{63}{22} \approx 2.86 \text{ (correct to 3 significant figures)} \end{aligned}$$

P.S. (Dear TA) I am sorry that copying the code from the pdf does not preserve indentation. Do you know any workarounds for this? I am using `\usepackage{minted}` for code snippets.