Practice Exam 1

Question 1

A.) You are given two singly-linked lists, possibly of different lengths, that contain distinct elements without repeats. Write the code for a function CountMatches() that returns the number of elements that match between the two singly-linked lists.

For example, consider the lists:

List1=> 1 -> 2 -> 3 -> 4

List2=> 2 -> 4 -> 6 -> 1 -> 5

There are 3 matches between the two lists. (1, 2, and 4)

The Nodes in these singly-linked list will contain a single integer value named value and a Node pointer named next.

Note: If there are no matches or either of the lists is empty the result should be 0.

B.) What is the time complexity for your function?

Solution 1

A.)
```
int CountMatches(Node* first, Node* second){
    int count = 0;
    Node* current;
    while(first != nullptr){
        current = second;
        while(current != nullptr){
            if(first->value == current->value){
                count++;
            }
            current = current->next;
        }
        first = first->next;
    }
    return count;
}
```

B.) O(n*m) for lists of length n and m

Question 2

Implement a recursive function in good C++ code called SumOfDigits(). It should take, as a parameter, an integer and returns the sum of the digits.

For example:
SumOfDigits(123) => 6
SumOfDigits(19) => 10
SumOfDigits(5) => 5

Solution 2

```
int SumOfDigits(int number){
    if(number < 10)
        return number;

    return number % 10 + SumOfDigits(number/10);
}
```

Question 3

A.) Implement a function in C++, called Reverse() that takes as a parameter the Node pointer to the head of a singly-linked list. The function should reverse the order of the linked list; i.e. the elements should be in reverse order when the function is finished.

Write good working C++ code to implement this function.

An empty list and a list with a single Node will remain unchanged:
An original list:

1 -> 2 -> 3 -> 4

will become:

4 -> 3 -> 2 -> 1

The Node structure contains multiple data elements of different types as well as a Node* named next.

Note: You must reverse the order of the Nodes in the list and not the values within the Nodes.

Question 3

B.) What is the time complexity of your solution for a singly-linked list with N elements?

C.) You realize this would be much easier if the list was doubly linked. Explain and show why with code demonstrating how you would reverse a doubly-linked list.

D.) What is the time complexity of this solution for a doubly-linked list with N elements?

Solution 3

A.) Method 1:

```
void Reverse(Node* &head){
    Node* current = head, *working = nullptr;
    if(current == nullptr){
        return;
    }
    head = nullptr;
    while(current != nullptr){
        working = current;
        current = current->next;
        working->next = head;
        head = working;
    }
}
```

Solution 3

A.) Method 2:

```
void Reverse(Node* &head){

        Node* current = head;
        Node *prev = NULL, *next = NULL;

        while (current != NULL) {
            next = current->next;
            current->next = prev;
            prev = current;
            current = next;
        }
        head = prev;
    }
```

Solution 3

B.) O(n) for n Nodes in the linked list

C.) It will be slightly easier in that we only need to swap the pointers.

```
void Reverse(Node* &head){
    Node* current = head, Node* temp;
    while(current != nullptr){
        temp = current->next;
        current->next = current->prev;
        current->prev = temp;
        current = temp;
        head = current;
    }
}
```

D.) O(n) for n Nodes in the linked list

Question 4

A.) It is often useful to randomize the order of a list of elements. Implement a function in C++, called Shuffle() that takes as a parameter the Node pointer to the head of a doubly-linked list.
The function should split the list in half (rounding down) and then interleave nodes from the first half with nodes from the second half, see the example below.

Write good working C++ code to implement this function.

An empty list and a list with a single Node will remain unchanged:
An original list:

1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8 <-> 9

Splits to:

1 <-> 2 <-> 3 <-> 4 5 <-> 6 <-> 7 <-> 8 <-> 9

and will become:

1 <-> 5 <-> 2 <-> 6 <-> 3 <-> 7 <-> 4 <-> 8 <-> 9

The Node structure may contain multiple data elements of different types as well as a Node pointer named next and a Node pointer named prev.

Note: You must change the order of the Nodes in the list and not the values within the Nodes.

Question 4

B.) What is the time complexity of your solution for a doubly-linked list with N elements?

C.) You think this might be easier if the values were stored in an array instead of a doubly-linked list, explain why or why not. Show your reasons with code demonstrating how you would shuffle an array of elements instead.

D.) What is the time complexity of this solution for an array with N elements?

Solution 4

A.)

```
void Shuffle(Node* head) {
    int count = 0;
    if (head == nullptr || head->next == nullptr) {
        return;
    }

    Node* current = head;
    while (current != nullptr) {
        count++;
        current = current->next;
    }

    Node* second = head;
    for (int i = 0; i < count / 2; i++)
        second = second->next;

    second->prev->next = nullptr;

    Node* first = head->next;
    current = head;

    while (first != nullptr && second != nullptr) {
        current->next = second;
        second->prev = current;
        current = current->next;
        second = second->next;
        current->next = first;
        first->prev = current;
        current = current->next;
        first = first->next;
    }
    current->next = second;
    second->prev = current;

}
```

Solution 4

B.) O(n) once through all the nodes.

C.) Much easier but requires O(n) space for the copy of the array

```
void Shuffle(Node* nodes[], int size) {
    if (size < 2) {
        return;
    }

    Node** working = new Node * [size];
    for (int i = 0; i < size; i++) {
        working[i] = nodes[i];
    }
    int index = 0;
    for (int first = 0, second = size / 2; first < size / 2; first++, second++) {
        nodes[index++] = working[first];
        nodes[index++] = working[second];
    }
}
```

D.) O(n) once through all the elements