1. This exam is closed book and no calculating devices of any type will be allowed. You are allowed, however, to prepare in advance the back of the hand-out info page with whatever notes you wish to place on it, and you may use this page during the exam. When the exam is over, submit this sheet with the rest of your exam.

2. Print your full name and your email address in the boxes above.

3. Print your name at the top of every page.

4. Please write clearly and legibly. If I can’t read your answer, I can’t give you credit.

5. Remember, these are NOT necessarily in order of difficulty. Please read all the problems first, and don’t allow yourself to get stuck on a single problem.

<table>
<thead>
<tr>
<th>#</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. (20 points) Consider our singly linked list class. Code a new function for this class called `size` which returns the size of the list. For example, if the list is

5 -> 2 -> 9 -> NULL

Your function should return the value 3. (You are NOT allowed to add a private variable or change any other functions - you should only write this one new function.)

Below is a small portion of our class (with a few of the functions for reference purposes), along with the spot for your function on the beginning of the next page.

```cpp
/* Singly Linked List Class */
template <typename Object>
class SLinkedList {

private:

    /* Node class for singly linked lists*/
    class SNode {
    private:
        Object elem;
        SNode * next;
    };

    //private variable list
    SNode * head; //pointer to the head of the list

public:

    SLinkedList() : head(NULL){}

    bool empty() const {
        return (head == NULL);
    }

    const Object& front() const {
        if (head == NULL) throw runtime_error("Head is NULL");
        return head->elem;
    }

    void removeFront() {
        SNode* old = head;
        head = old->next;
        delete old;
    }

    void addFront(const Object& e) {
        SNode* v = new SNode;
        v->elem = e;
    }

};
```
v->next = head;
head = v;
}

int size() {
    // your code goes here
    int count = 0;
    SNode* v = head;
    while (v != NULL) {
        count++;
        v = v->next;
    }
    return count;
}
2. (10 points) Choose one answer for each of the following questions.

(a) In our code from class, what is the running time of the function `addFront` in a singly linked list? Sol: i
   i. $O(1)$
   ii. $O(\log n)$
   iii. $O(n)$
   iv. $O(n^2)$

(b) In our code from class, what is the *amortized* running time of the function `push_back` in a vector? Sol: i
   i. $O(1)$
   ii. $O(\log n)$
   iii. $O(n)$
   iv. $O(n^2)$

(c) What is the worst case running time of bubble sort? Sol: iv
   i. $O(\log n)$
   ii. $O(n)$
   iii. $O(n \log n)$
   iv. $O(n^2)$

(d) What is the worst case running time of merge sort? Sol: iii
   i. $O(\log n)$
   ii. $O(n)$
   iii. $O(n \log n)$
   iv. $O(n^2)$

(e) In a binary search, what is the worst case running time of the function `find`? Sol: ii
   i. $O(1)$
   ii. $O(\log n)$
   iii. $O(\sqrt{n})$
   iv. $O(n)$
3. (10 points) Consider Vector and List data structures that we have learned.

(a) Which data structure is efficient in terms of insertion and deletion? Describe the reason with comparing the two data structures.
Sol: Insertion and Deletion in List is very efficient as compared to vector because to insert an element in list at start, end or middle, internally just a couple of pointers are swapped.
Whereas, in vector insertion and deletion at start or middle will make all elements to shift by one. Also, if there is insufficient contiguous memory in vector at the time of insertion, then a new contiguous memory will be allocated and all elements will be copied there.

(b) Which data structure is efficient in terms of random access? Describe the reason with comparing the two data structures.
Sol: As List is internally implemented as doubly linked list, therefore no random access is possible in List. It means, to access 15th element in list we need to iterate through first 14 elements in list one by one.
Whereas, vector stores elements at contiguous memory locations like an array. Therefore, in vector random access is possible i.e. we can directly access the 15th element in vector using operator .
4. (20 points) Suppose we start with an initially empty `std::list<int>` called `mylist`. Fill in the missing values in the second column below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Contents of mylist after current operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mylist.insert(mylist.begin(), 11)</code></td>
<td>(11)</td>
</tr>
<tr>
<td><code>mylist.push_front(5)</code></td>
<td>(5, 11)</td>
</tr>
<tr>
<td><code>mylist.push_back(7)</code></td>
<td>(5, 11, 7)</td>
</tr>
<tr>
<td><code>mylist.reverse()</code></td>
<td>(7, 11, 5)</td>
</tr>
<tr>
<td><code>mylist.insert(mylist.end(),8)</code></td>
<td>(7, 11, 5, 8)</td>
</tr>
<tr>
<td><code>mylist.pop_front()</code></td>
<td>(11, 5, 8)</td>
</tr>
<tr>
<td><code>mylist.insert(mylist.begin(),2)</code></td>
<td>(2, 11, 5, 8)</td>
</tr>
<tr>
<td><code>mylist.insert(mylist.end()/-,-,6)</code></td>
<td>(2, 11, 5, 8, 6)</td>
</tr>
<tr>
<td><code>mylist.pop_back()</code></td>
<td>(2, 11, 5, 8)</td>
</tr>
</tbody>
</table>
5. (20 points) Write a function which accepts as input a reference to a list of integers. Your task is to remove all nodes whose values are greater than 100, and then to add a new node at the end of the list, whose value is equal to the sum of all the values in all the nodes you removed. For example, if the list had contained

4->502->10->12->7->33->5->821->11->103->7->NULL

then your resultant list should be

4->10->12->7->33->5->11->7->1426->NULL

You may use any of the C++ STL functions to do this. In particular, iterators and the list functions associated with them (such as begin(), operator++, operator* and insert) may prove useful.

```cpp
void SumOfRemoval(list<int>& theList) {
    // your code goes here
    int sumRemoves = 0;
    list<int>::iterator it;
    for (it = theList.begin(); it != theList.end(); it++) {
        if (*it > 100) {
            sumRemoves += *it;
            theList.erase(it);
        }
    }
    theList.push_back(sumRemoves);
}
```
6. (20 points) Complete the recursive power function `power(x, n)` that runs in \(O(\log n)\) time using divide and conquer to multiply \(x\) exactly \(n\) times. Describe shortly how the function is \(O(\log n)\) time.

```c
int power(int x, int n)
{
    // your code goes here
    int temp;
    if (n == 0)
        return 1;
    temp = power(x, n/2);
    if (n%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
```

The recursive power function using divide and conquer is \(\log(n)\) because it just solves half of the power recursion.
(scratch paper)