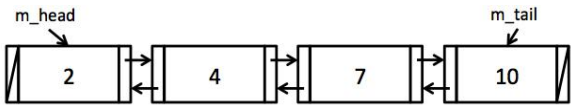
Name: Nashir Jammohamed **ID:** 1325605

Cautionary Rant: It is a good idea to check your work by actually programming any problem. But, it is critically important to complete these, or any exercise, without the aid of your computer **first**. You will not have access to a computer during the exam or during most technical interviews with any large software development company.

Problem 1: Doubly Linked List Class

We will build a sorted doubly linked list. The following shows an example of a list with 4 elements, where the nodes are sorted in the increasing order of their values. The `m_prev` pointer of the head node and `m_next` pointer of the tail node both point to `nullptr`. If the list is empty, head and tail pointers both point to `nullptr`.



Assume the following declaration of `Node` structure and `SortedLinkedList` class.

```
struct Node {
        int value;
        Node *prev;
        Node *next;
};
```

```
class SortedLinkedList {
public:
        SortedLinkedList();
        bool insert(const int &value);
        const Node *search(const int &value) const;
        void remove(Node *node);
        int size() const { return m_size; }
        void printIncreasingOrder() const;

private:
        Node * m_head;
        Node *m_tail;
        int m_size;
}
```

a. Implement SortedLinkedList().

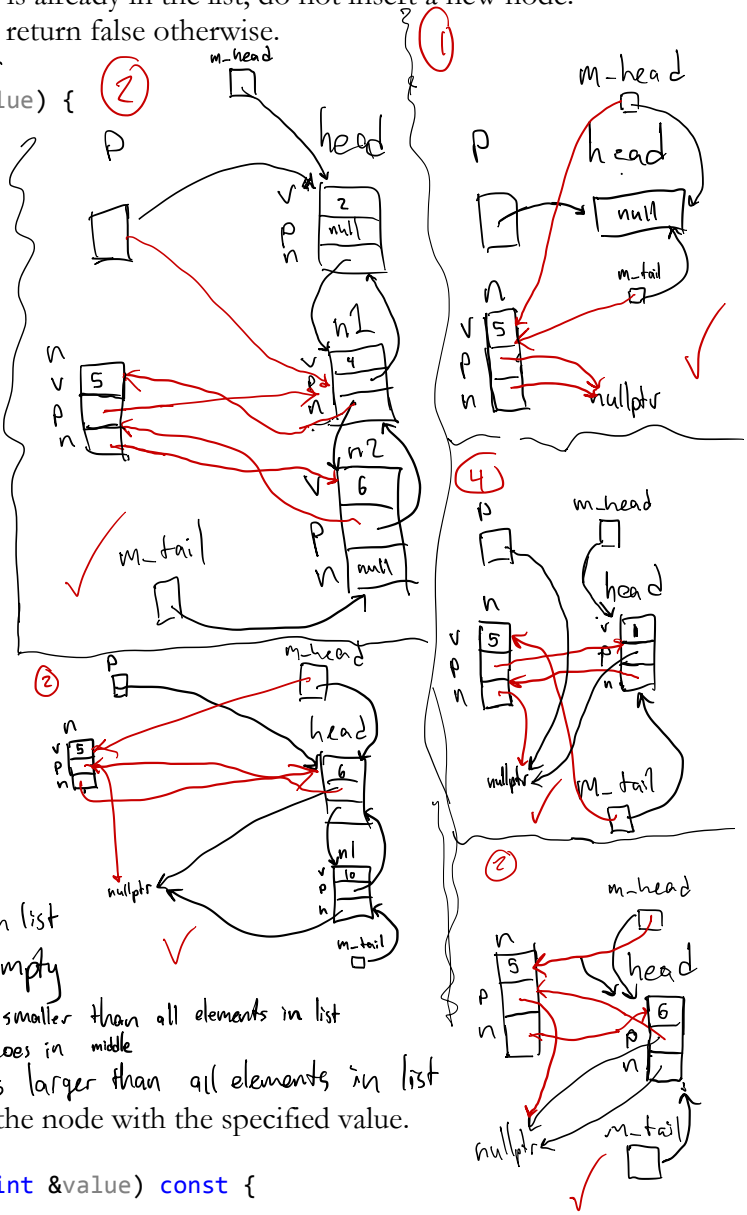
```
SortedLinkedList::SortedLinkedList() {
        m_size = 0;
        m_head = m_tail = nullptr;

}
```

Name: Nashir Jammohamed **ID:** 1325605

b. Implement `insert()`. If a node with the same value is already in the list, do not insert a new node. Return true if a new node is successfully inserted, and return false otherwise.

```
bool SortedLinkedList::insert(const int &value) {
    m_size++;
    if (m_head == nullptr){
        Node *n = new Node;
        n->value = value;
        n->next = n->prev = nullptr;
        m_head = m_tail = n;
        return true;
    }
    if (m_head->value > value){
        Node *n = new Node;
        n->value = value;
        n->prev = nullptr;
        n->next = m_head;
        m_head->prev = n;
        m_head = n;
        return true;
    }
    Node *p = m_head;
    while (p != nullptr){
        if (p->value == value){
            m_size--;
            return false;
        }
        if (p->next != nullptr && p->next->value > value){
            Node *n = new Node;
            n->value = value;
            n->next = p->next;
            n->prev = p;
            p->next = n;
            n->next->prev = n;
            return true;
        }
        p = p->next;
    }
    Node *n = new Node;
    n->value = value;
    n->next = nullptr;
    m_tail->next = n;
    n->prev = m_tail;
    m_tail = n;
    return true;
}
```



* if value is in list
(1) if list is empty
(2) if value is smaller than all elements in list
(3) if value goes in middle
(4) if value is larger than all elements in list

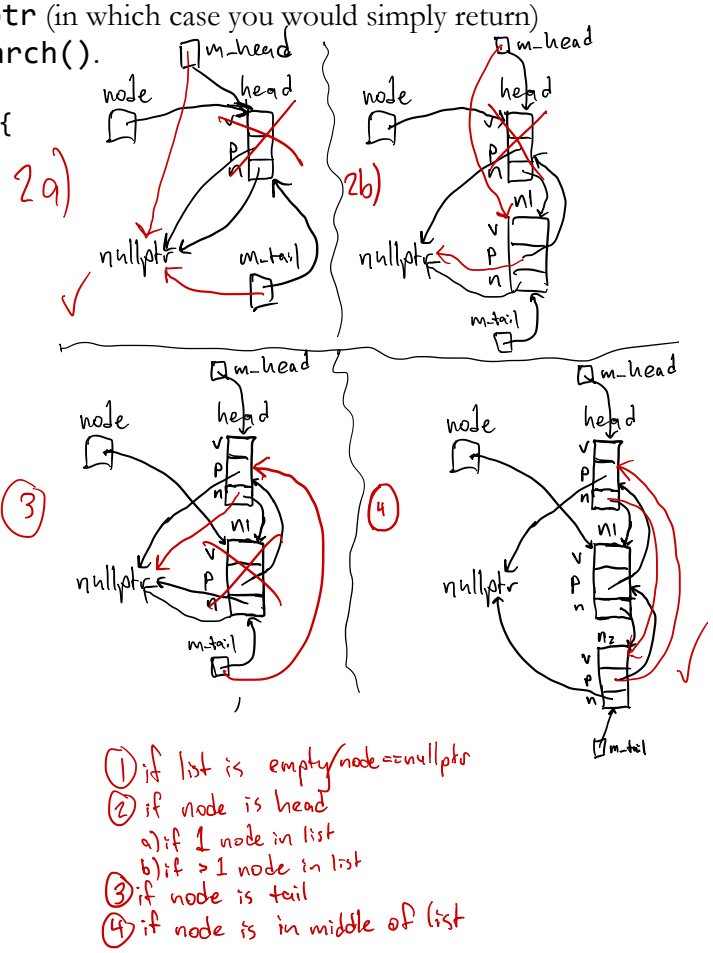
c. Implement `search()`, which returns the pointer to the node with the specified value.

```
const Node *SortedLinkedList::search(const int &value) const {
    Node *p = m_head;
    while (p != nullptr){
        if (p->value == value)
            return p;
        p = p->next;
    }
    return nullptr;      //if p is not found or if list is empty
}
```

Name: Nashir Jammohamed **ID:** 1325605

d. Implement `remove()`. Assume `node` is either `nullptr` (in which case you would simply return) or a valid pointer to a Node in the list, as found in `search()`.

```
void SortedLinkedList::remove(Node *node) {
    if (node == nullptr)
        return;
    else if (node == m_head){
        if (m_head == m_tail)
            m_head = m_tail = nullptr;
        else {
            m_head = node->next;
            m_head->prev = nullptr;
        }
    }
    else if (node == m_tail){
        m_tail = node->prev;
        m_tail->next = nullptr; //node->prev->next
    }
    else { //interior node
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    delete node;
    m_size--;
}
```



1. if list is empty/node==nullptr
2. if node is head
   a) if 1 node in list
   b) if > 1 node in list
3. if node is tail
4. if node is in middle of list

e. Implement `printIncreasingOrder()`, which prints the values stored in the list in the increasing order, one value in each line.

```
void SortedLinkedList::printIncreasingOrder() const {
    Node *p = m_head;
    while (p != nullptr){
        std::cout << p->value << std::endl;
        p = p->next;
    }
}
```

f. Program your implementation in a single file named sdll.h and include that with your quiz 2 submission archive.

Name: Nashir Jammohamed **ID:** 1325605

Problem 2: Recursion

a. Write a recursive function that computes base^{exp} (base raised to the power of exp). What is the Big-O of this function?

```
// no negative exponents!
int recursivePow(int base, int exp) {
    if (exp < 1)          // 1
        return 1;         // 2
    return (base * recursivePow(base, exp-1)); // 3
}
```

base = 3, exp = 3

3 · v(3,2)
3 · v(3,1)
3 · v(3,0)
1

27 ✓

① O(1) · n
② O(1) · n
③ O(1) · n

} O(n) time & space complexity

b. Write a function **powerThree** that, given a non-negative number **n**, returns 3^n (3^n, or "3 raised to power n") recursively, assuming 3^n is something that can be represented as an integer. Do not use a loop, and do not use the character **'*'** anywhere in your code. What is the Big-O of this function?

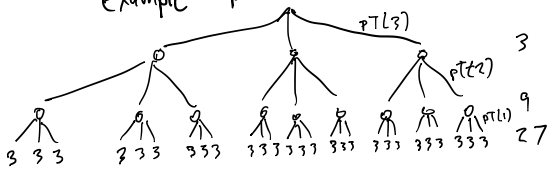
solution #1

```
int powerThree(int n) {
    if (n < 1)
        return 1;
    if (n == 1)
        return 3;
    return (powerThree(n-1) + powerThree(n-1) + powerThree(n-1));
}
```

solution #2

```
int powerThree(int n){
    if (n < 1)                          // O(1)·n
        return 1;                       // O(1)·n
    int sum = powerThree(int n);        // O(1)·n
    return (sum + sum + sum);           // O(1)·n
}
```

example parent call (n = 4)



$3^{(n-1)}$, n>0

n <= 1 => O(1)

O(n)

O(3^n)

Name: Nashir Janmohamed **ID:** 1325605

Problem 3: Big-O

a. Suppose Algorithm A and B perform the same task. For any given input size n, algorithm A executes in f(n)=0.003n2 operations, and algorithm B executes f(n)=250n operations. Specifically, when does Algorithm A perform better than B?

$f_A(n) = 0.003n^2 \quad f_B(n) = 250n \quad f_A(n) = f_B(n) = \frac{0.003n^2}{0.003n} = \frac{250n}{0.003n} \Rightarrow n = 83{,}333.\overline{3}$ (f_A & f_B intersect)

for $n \leq 83{,}333$ f_A performs better than f_B

for $n > 83{,}333$ f_B performs better than f_A

b. An algorithm that is $O(n^2)$ takes 10 seconds to complete when n=100. How long would you expect it to take when n=500?

$n^2 = 10s$ when $n = 100 \Rightarrow$ 10,000 elements = 10 seconds $\Rightarrow 10^3$ elements/second

$n^2 = x s$ when $n = 500 \Rightarrow 2.5\times10^5$ elements = x seconds $\Rightarrow 2.5\times10^5$ elements $\times \frac{1 \text{ second}}{10^3 \text{ elements}} = 250s$

c. What is the Big-O of the following code segment:

```
for (int i = 0; i < n; i++)
      for (int j = 0; j < 4 * i; j++)
            sum++;
```

$\frac{n(4(n-1))}{2} = \frac{4n^2-4n}{2} = 2n^2 - 2n$

$f(n) = 1 + 3n + 3(2n^2 - 2n) = 6n^2 - 6n + 1$

$O(n^2)$

d. What is the Big-O of the following function?

```
int gobidygoop(int n, int p) {
      int ac = 1;
      for (int i = 0; i < n; i++) {
            int k = p;
            while (k > 1) {
                  ac *= i + k;
                  k /= 4;
            }
      }
}
```

Annotations: $\log_4 p \cdot n \cdot 1$ (each line of the while loop)

$f(n) = 1 + 1 + 3n + 3n\log_4 p$

$O(n + n\log_4 p)$

Name: Nashir Jammohamed **ID:** 1325605

e. Consider the following pseudo code:

```
Get value for n          // 1
Set the value of k to 1  // 1
While k is less than or equal to n  // 1*n
    Set the value of j to twice of k // 1* n
    While j is greater or equal to 1  // n*log2n
        Print the value of j          // 1 * n * log2n
        Set the value of j to one half its former value; // log2n *n
    Increase k by 1                   // 1*n
```

What is the Big-O of this pseudocode? What does this print if n is 4?

$O(n \log_2 n)$

n=4 k=1 j=2 j=1 j=0 k=2 j=4 j=2 j=1 j=0 k=3 j=6 j=3 j=1 j=0 k=4 j=8 j=4 j=2 j=1 j=0

Output

2 1 4 2 1 6 3 1 8 4 2 1

n1 n2 n3 n4

9, 12, 15, 18, 21

0-1 1-3 3-7 7-15 15-31?

n=0, 2
n=1, 11
n=2, 23
n=3, 35
n=4, 50

n=5, 65
n=6, 80
n=7, 95
n=8, 113
n=9, 131

n=10, 149
n=11, 167
n=12, 185
n=13, 203
n=14, 221
n=15, 239
n=16, 266

Problem 5: Sorting

a. Fill out the following table or sorting properties, if there is no special condition for a particular case then leave it blank:

Sorting Algorithm	Selection	Insertion	Bubble	Quick	Merge
Average Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Worse Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log_2 n)$
Condition for Worse				input is already (or mostly) sorted or in reverse order	
Best Complexity	$O(n^2)$	$O(n)$	$O(n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Condition for Best		input is already (or mostly) sorted	input is already (or mostly) sorted		

Name: Nashir Jammohamed **ID:** 1325605

b. Sort the following array using the Mergesort algorithm. Show each recursive step, including the merge.