(KEY)
Linked List Review

1. Explain the differences between a Single, Doubly and Circular Linked List
   Answer: In a single linked list, each node has a reference to the next node in the list. It only allows traversal in one direction (forward). In Doubly LinkedList, each node has references to both the next and the previous nodes, which allows traversal in both forward and backward directions. A circular linked list is like a singly linked list, but the last node's next pointer points back to the first node, forming a closed loop.

2. What are some public members we can include in the Linked Lists' Class? What about private members?
   Answer: Some Public member function like adding, removing, searching an element, and traversing. Private members should contain head and tail (if it needed or provided), sometimes even size function are kept private for the LinkedList.

3. Why do we save the head and tail as private members?
   Answer: So we can ensure that these pointers are modified only through the function that are public that uses these ptrs.

4. What kind of memory allocation do we use with Linked Lists?
   Answer: Uses dynamic memory allocation where each node is allocated in a heap.

5. What are the differences between Linked Lists and Arrays?
   Answer: Linked Lists use dynamic allocation(heap), size can be flexible (not fixed). Arrays are allocated in contiguous memory, size is fixed.

6. Name the advantages and disadvantages of using Linked Lists.
   Answer: Advantage: 0(1) access to the head and tail (if used), flexible size. Disadvantage: 0(n) random access, traversal is inefficient, higher overhead due to dynamic allocation due to each node allocation.

7. Implement the addAtHead(), addAtEnd(), and addAtMiddle() functions.
   addAtHaed()

```
void LinkedList::addToFront(int val)
{
    Node* newNode = new Node(val);

    if (isEmpty())
    {
        head = newNode;
    }
    else
    {
        newNode->next = head;
        head = newNode;
    }

    size++;
}
```

addAtEnd()

```
void LinkedList::addToEnd(int val)
{
    if (isEmpty())
    {
        addToFront(val);
    }
    else
    {
        Node* newNode = new Node(val);
        Node* current = head;

        while (current->next != nullptr)
        {
            current = current->next;
        }

        current->next = newNode;
        size++;
    }
}
```

addAtMiddle()

```
void LinkedList::addAt(int index, int val)
{
    if (index <= 0)
    {
        addToFront(val);
    }
    else if (index >= size)
    {
        addToEnd(val);
    }
    else
    {
        // index > 0 && index < size if we enter here

        Node* newNode = new Node(val);
        Node* current = head;

        for (int i = 0; i < index - 1; i++)
        {
            current = current->next;
        }

        newNode->next = current->next;
        current->next = newNode;
        size++;
```

8. Implement the removeAtHead(), removeAtEnd(), and removeAtMiddle() [Based on Value and Position] functions.
   removeAtHead()

```
void LinkedList::deleteAtBeg()
{
    if(!isEmpty())
    {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

removeAtEnd()

```
void LinkedList::deleteAtEnd()
{
    if(!isEmpty())
    {
        Node* temp;
        Node* curr;

        //if only 1 element in linked list
        if(head -> next == nullptr)
        {
            delete head;
            head = nullptr;
        }
        else
        {
            curr = head;

            while(curr -> next -> next != nullptr)
            {
                curr = curr -> next;
            }

            temp = curr -> next;
            curr -> next = nullptr;
            delete temp;
        }
    }
}
```

removeAtMiddle()

```
void LinkedList::deleteAtPos(int pos)
{
    if(!isEmpty())
    {
        Node* temp;
        Node* curr;

        //if only 1 element in linked list
        if(head -> next == nullptr)
        {
            delete head;
            head = nullptr;
        }

        else
        {
            curr = head;

            for(int i = 1; i < pos - 1; i++)
            {
                curr = curr -> next;
            }

            temp = curr -> next;
            curr -> next = temp -> next;
            delete temp;
        }
    }
}
```

9. Name the difference between the 2 snippets of code:

```
void LinkedList::print(Node* curr)
{
    if(curr != nullptr)
        return;

    print(curr -> next);
    cout << curr -> data << " ";
}
```

```
void LinkedList::print(Node* curr)
{
    if(curr != nullptr)
        return;

    cout << curr -> data << " ";
    print(curr -> next);
}
```

In order

Reverse order

10. Implement a Linked List function that reverses a Linked List.

```
void LinkedList::reverse() {
    node* curr = head;
    node* prev = nullptr;
    node* nextNode = nullptr;

    while(curr != nullptr) {
        // nextNode is always one ahead of curr
        nextNode = curr->next;

        // change the link to prev (where the reverse comes in)
        curr->next = prev;

        // increment prev and curr
        prev = curr;
        curr = nextNode;
    }

    // update head
    head = prev;
}
```

Recursion Review

1. Write a recursive function which returns the factorial of a given number.

factorial.cpp > factorial(int)

```
int factorial(int input){
    if (input <=1)
        return 1;
    return input*factorial(input-1);
}
```

2. Given a number n, print n-th Fibonacci number.

```
int fib(int num){
  if(num <= 1 ){
    return num;
  }

 return fib(num -1) + fib(num-2);
}
```

3. Write a recursive function which given the number n, returns the sum of all numbers 0 to n.

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

4. Given a string, write a recursive function that checks if the string is a palindrome.

```
bool Palindrome(string s, int start, int end){
  //edge case
  if(s.length() <= 2){
    return true;
  }
  //base case
  if(start >= end){
    return true;
  }else if(s[start] == s[end]){
    //recursive case
    return Palindrome(s, start+1, end-1);
  }else{
    return false;
  }
}
```

5. Write a recursive function which prints every other node in a Linked List.

```
void printEveryOtherNode(Node* head) {
    int count = 1; // Initialize count to 1 to print the first node

    while (head != NULL) {
        if (count % 2 != 0) {
            cout << head->data << " ";
        }
        head = head->next;
        count++;
    }
}
```

6. Write the output of the following recursive function.

```
void fun(int x)
{
    if(x >= 0)
    {
        fun(x-1);
        --x;
        fun(x-1);
        cout << x << " ";
    }
}

int main(){

    fun(3);
```

Output: -1 0 -1 1 -1 0 2

Sorting Algorithms Review

1. Write the Bubble Sort Algorithm Implementation

a. Both v1 and v2
b. What is the Time Complexity? Best: O(n) from v2, Worst: O(n^2)
c. Explain in words how the algorithm works
Compares two adjacent elements and swaps them until they are in the intended order. For v2 it uses a bool variable to see if swap occurred in the pass, if the swap never happened in a pass that mean the array is sorted.

2. Write the Selection Sort Algorithm Implementation
    a. What is the Time Complexity? Best and Worst: O(n^2)
    b. Explain in words how the algorithm works
    Chooses the smallest element in an unordered array in each iteration and swap with the current element until it sorted.

3. Write the Insertion Sort Algorithm Implementation
    a. What is the Time Complexity Best: O(n), Worst: O(n^2)
    b. Explain in words how the algorithm works
    It places an unsorted element at its suitable place every iteration by checking the array and placing it at the right place(shifting).

4. Perform All Sorts on the Following Unsorted Array (Tracing)
    a. [12, 7, 5, 1, 3]

Bubble Sort: [12, 7, 5, 1, 3]
Pass 1:
[7, 12, 5, 1, 3]
[7, 5, 12, 1, 3]
[7, 5, 1, 12, 3]
[7, 5, 1, 3, 12]
Pass 2:
[5, 7, 1, 3, 12]
[5, 1, 7, 3, 12]
[5, 1, 3, 7, 12]
Pass 3:
[1, 5, 3, 7, 12]
[1, 3, 5, 7, 12]

Selection Sort: [12, 7, 5, 1, 3]
Pass 1:
[1, 7, 5, 12, 3]
Pass 2:
[1, 3, 5, 12, 7]
Pass 3:
[1, 3, 5, 7, 12]

Insertion Sort: [12, 7, 5, 1, 3]
Pass 1:
[7, 12, 5, 1, 3]
Pass 2:
[5, 7, 12, 1, 3]
Pass 3:
[1, 5, 7, 12, 3]
Pass 4:
[1, 3, 5, 7, 12]

Arrays Review

1. What are the main differences between statically and dynamically fixed arrays?
   Answer: Statically fixed arrays have a fixed size, size determined at compile time, while dynamically fixed arrays can resize themselves at runtime according to the user's need. Statically fixed arrays often use stack memory, but dynamic arrays allocate memory on the heap.

2. What do the "new" and "delete" keywords do?
   Answer: Operator new allocates memory (a variable) of the designated type and returns a pointer to it.
   Operator delete is used to destroy dynamic variables, so that its memory space.

3. Name the differences between stack and heap memory. Which does static memory use? Which does dynamic memory use?
   Answer: Stack memory is fast and small whereas heap memory is slower and larger. Static memory is used for global and static variables and arrays with fixed lifetimes, while heap is used for dynamic memory allocation and allows for flexible memory management during program execution.

4. What are the advantages of using dynamic arrays over static ones?
   Answer: Dynamic arrays can be resized themselves as needed, while static arrays have a fixed size that may lead to memory waste or size limitations. Dynamic arrays have better memory management compared to static arrays.

Time Complexity Review

1. Order the time complexities from fastest to slowest.

O(n * log(n)), O(n), O(n!), O(n^2), O(1), O(log(n)), O(2^n)
Answer: O(1), O(log(n)), O(n), O(n*log(n)), O(n^2), O(2^n), O(n!)

2. Review “Week 4 – Big O Notation” Slides for Time Complexity Examples

3. Name the Time Complexities to the following:

0(1)

```
for(int i = 0; i < 10000; i++)
{
    for(int i = 0; i <= i; i *= 2)
    {
        cout << "Hello World" << endl;
    }
}
```

```
void fun(int n)
{
    if(n <= 0)
        return;

    fun(n - 1);
    fun(n - 2);
}
```

```
void fun2(int n)
{
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j *= 2)
            cout << "Hello World" << endl;
}
```

```
void fun3(int n)
{
    for(int i = 0; i < n; i++)
        for(int j = 0; j < 100; j *= 2)
            cout << "Hello World" << endl;
}
```

```
void fun4(int n)
{
   if(n <= 0)
    return;

    fun4(n - 1);
}
```

```
void fun5(int n)
{
   if(n <= 0)
    return;

    fun5(n/2);
}
```

Fun: O(2^n)
Fun2: O(nlog(n))
Fun3: O(n)
Fun4: O(n)
Fun5: O(log(n))