1. Compare the two implementations of the linear list, and explain when you should use the sequential list and when you should use the linked list.

Linear lists are a fundamental data structure used in computer science and programming. There are two primary ways to implement linear lists**: using an array-based structure called a sequential list** or using **a node-based structure called a linked list**. Here are the differences between the two and when to use each:

Sequential List:

A sequential list is an ordered collection of elements where the elements are stored in contiguous memory locations, and each element is accessed using its position (index) in the list. Sequential lists provide **efficient random access to elements,** which means you can easily retrieve an element from the list using its index. **However, inserting or deleting an element in a sequential list requires shifting all the elements in the list to make space for the new or removed element.** This process can be inefficient if you need to insert or delete elements frequently.

When to use Sequential List:

Sequential lists are best suited for applications where **the size of the list is fixed or changes infrequently,** and **random access to elements is a common operation**. They are **well suited for small lists,** where the overhead of managing pointers in a linked list outweighs the benefits of random access.

Linked List:

A linked list is an ordered collection of elements where each element (node) is a separate object that contains a reference (pointer) to the next element in the list. Linked lists provide **efficient insertion and deletion of elements,** as only the pointers need to be updated. **However, accessing an element in a linked list requires traversing the list from the beginning to the desired location.** This process can be inefficient if you need to access elements randomly or frequently.

When to use Linked List:

**Linked lists are best suited for applications where the size of the list is variable or changes frequently, and frequent insertion or deletion of elements is required.** They are well suited for large lists, where the overhead of managing pointers is less significant compared to the benefits of efficient insertions and deletions.

**In summary, if you need efficient random access to elements, use a sequential list. If you need efficient insertion and deletion of elements, use a linked list.**

1. Try to explain the function of head pointer and head node.

The head node is the actual first node in the list. It is often represented as a pointer to the first node, called the "head pointer" **which is initialized when we create an linkedList instance**. The head pointer points to the memory address of the first node in the list.

The head pointer is necessary because it provides a way to access the first node in the list. When you **want to traverse or manipulate a linked list, you typically start at the head** and follow the pointers to access the other nodes in the list.

1. **In the double linked list and single circular linked list, if you know that the pointer points to the base node, can you delete the node? If it can be deleted, what is the time complexity?**

Yes, if a pointer points to the base node in a double linked list or a single circular linked list, the node can be deleted. The time complexity for deleting a node from a double linked list is O(1) as it involves updating the pointers of th**e previous and next nodes** to skip over the node being deleted.

In a single circular linked list, the time complexity for deleting a node from the base node is O(n), where n is the total number of nodes in the list, **as one has to traverse through the entire list to find the previous node that points to the base node** and update its pointer accordingly.

**dLinkList<T>**

currPos->prev->next = NULL;

delete currPos;

--currentLength;

**sLinkList<T>**

node \*pos, \*delp;

prev = **move**(*find\_index\_by\_pointer*(CurrPos) - 1);

delp = prev ->next;

prev ->next = NULL; // 绕过delp

delete delp;

--currentLength;

1. What is the capacity of the sequence table? What is the length of the sequence table? Why is there no concept of capacity in the linked list?

In an array-based sequence table, the capacity refers to the maximum number of elements that can be stored in the table without resizing it. The length of the table refers to the number of elements currently present in the table. The capacity and length of an array-based sequence table are closely related; the length cannot exceed the capacity, and when the length approaches the capacity, the table needs to be resized to enable additional elements to be stored.

On the other hand, in a linked list, there is no concept of capacity because the linked l**ist dynamically allocates memory as necessary for each new element added to the list.** The length of the linked list can change dynamically as elements are added or removed, without requiring any resizing operations. Because of this ability to grow and shrink dynamically, linked lists are often used in situations where the number of elements to be stored is unknown or may change over time, **and where efficient memory utilization is critical.**

**6. Try to briefly describe the main advantages and disadvantages of the list and the main advantages and disadvantages of the linked list.**

**List Advantages:**

Lists provide efficient access to elements using an index.

Lists can be easily sorted or searched using built-in functions or libraries.

**Disadvantages**:

Inserting or deleting elements from the beginning or middle of a list can be slow, as all the following elements need to be shifted.

Lists have a fixed size, so resizing can be inefficient and memory-intensive.

**Linked list Advantages:**

Linked lists can easily grow or shrink in size without having to move elements in memory.

Inserting or deleting elements anywhere in the list is fast and efficient.

Linked lists can be easily traversed in either direction.

**Disadvantages**:

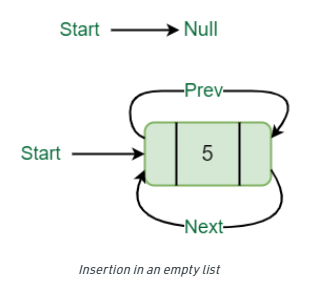
Linked lists use more memory than arrays or lists because each element needs a separate node.

Accessing an element in the middle of the list can be slower than in a list, as each element needs to be traversed from the beginning.

Linked lists cannot be efficiently sorted or searched using built-in functions or libraries.

**11.If head points to the tail of the double circular linked list, the double circular linked list is empty initially, that is, the value of head is NULL. Write the statement used to insert the first node in the double circular linked list.**

<https://www.geeksforgeeks.org/insertion-in-doubly-circular-linked-list/>



void insert(int data) {

Node\* new\_node = new Node(data);

if (this->head == nullptr) {

this->head = new\_node;

this->head->next = this->head;

this->head->prev = this->head;

}

else {

Node\* tail = this->head->prev;

tail->next = new\_node;

new\_node->prev = tail;

new\_node->next = this->head;

this->head->prev = new\_node;

}

}

1. Given two ascending linked lists of length m and n respectively, if they are merged into a descending linked list of length m+n, what is the time complexity in the worst case?

**See /code/hmw/hmw1/ex1-12.cpp**

**二 程序 设计 题**

1. **See LinkedList.cpp /\* erase nodes whoes values are between [i, j] \*/**

template <class T>

void sLinkList<T>::erase(int i, int j) {

node \*pre, \*delp;

int cnt = 0;

pre = head;

while (pre->next != NULL) {

if (i <= pre->next->data && pre->next->data <= j) {

delp = pre->next;

pre->next = delp->next;

delete delp;

++cnt;

} else

pre = pre->next; // fixed bug when have several tmp[i] in a raw

}

currentLength -= cnt;

}

1. **Reverse linkedList:**

**See LinkedList.cpp**  *(https://www.geeksforgeeks.org/reverse-a-linked-list/)*

template <class T>

void sLinkList<T>::reverse() {

node\* curr = head ->next; // always assign to head->next

node\* prev = NULL;

node\* next = NULL;

// line 1 and 4 simple update curr to next one, line 2 makes curr to look back,

line 3 assigns looking back curr node to prev

while(curr!=NULL){

next = curr->next;

curr->next = prev;

prev = curr;

curr = next;

}

head->next = prev;

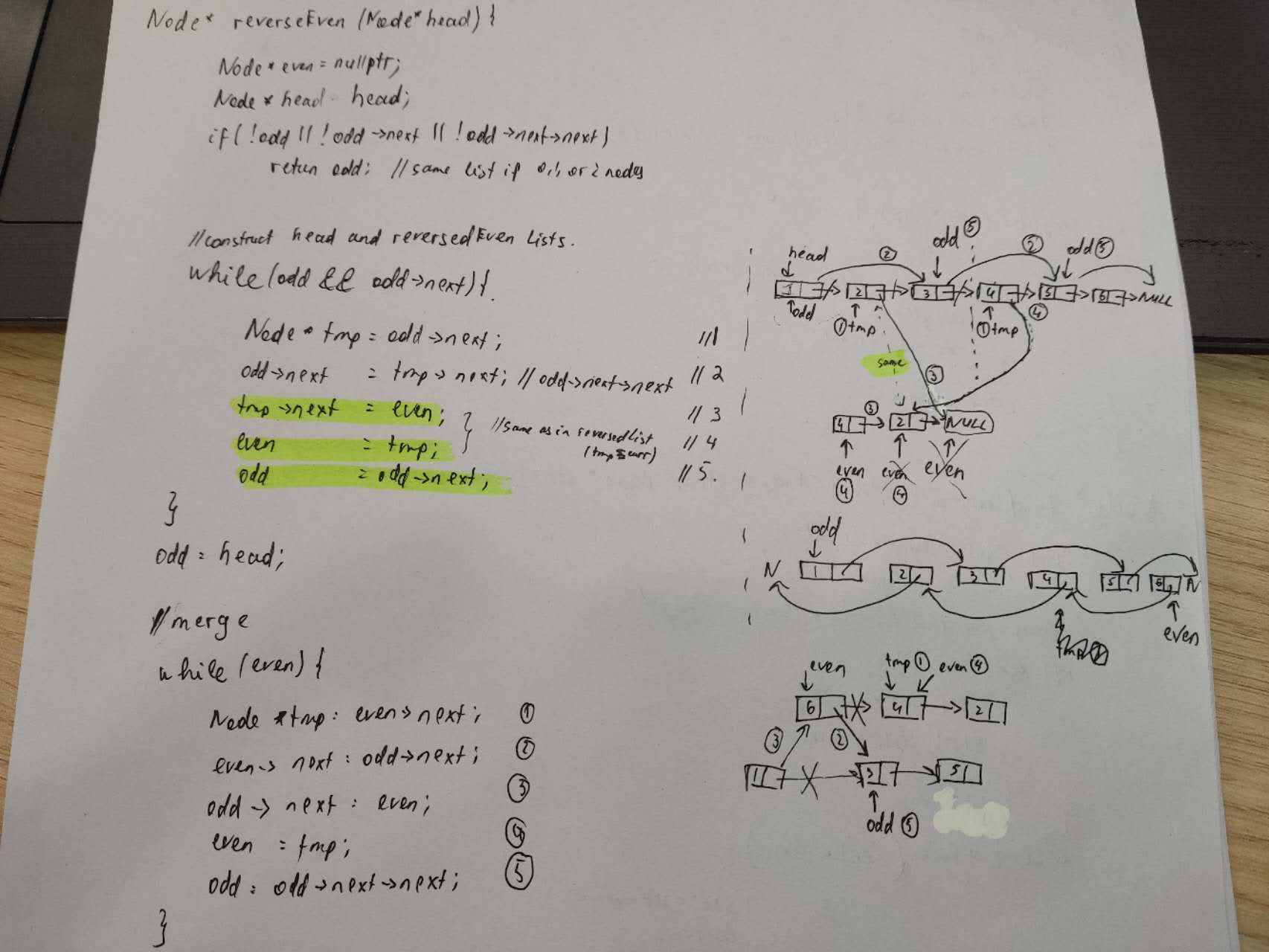
}

1. -----
2. See /code/hmw/hmw1/ex3-10.cpp

**Ex4)** Singly linked list with header nodes (a1, a2,...an-1,an), n is an even number, design algorithm to converts it to (a1,an,a2,an-1,...)

See /code/hmw/hmw1/ex4

https://www.geeksforgeeks.org/reverse-the-order-of-all-nodes-at-even-position-in-given-linked-list/?ref=rp



**Ex5)**

Probability of finding the element and not find the element in this function is the same, data is equally distributed, do time complexity analysis

int seqList<T>::search(const T &x) const

{

int i = 0;

for (i = 0; i < currentLength && data[i] != x; ++i);

if (i == currentLength)

return -1; // success

else

return i; // fail

};

