1. 线性表：线性表是具有相同特性的数据元素的一个有限序列。每个中间节点有唯一前驱，唯一后继

顺序表：以物理相邻体现逻辑相邻的线性表

链表：用指针或游标链接体现逻辑链接的线性表

1. 使用顺序表：

时间：

元素具有相邻线性逻辑关系，需要大量查找操作，少量插入删除操作或插入删除不需要移动后面元素的位置

空间：

线性表长度变化不大

使用链表：

时间：

元素具有链接逻辑关系，需要大量插入删除操作，需要少量查找操作

空间：

线性表长度变化大

顺序表优点：

时间：

由于物理相邻体现逻辑相邻，可以通过首元素地址直接算出所要位置的元素地址，查找时间复杂度低。按位置查找O(1)，若有序可用二分查找来按值查找时间复杂度O(logn)

空间：

没使用指针，游标，不用花费附加开销

顺序表缺点：

时间：

由于物理相邻体现逻辑相邻，插入删除操作需要移动后面的元素来保持物理相邻。时间复杂度O(n)

空间：

若长度变化大，需要指针+malloc/realloc/free动态分配内存，不支持自定义构造函数和析构函数；或者指针+new/delete+拷贝，时间耗费高

链表优点：

时间：

由于用指针或游标链接体现逻辑链接，插入删除操作不需要移动后续元素的位置。不算查找时间复杂度O(1)，而查找操作一般在其他地方已经操作。

空间：

支持动态改变链表长度

链表缺点：

时间：

由于用指针或游标链接体现逻辑链接，没有物理相邻，查找需要遍历链表。按位置查找时间复杂度O(n)，按值查找无法用二分法时间复杂度O(n)

空间：

需要额外的空间存储指针/游标

1. 插入平均：(n+0)/2=n/2

删除平均：(n-1+0)/2=(n-1)/2

因素：插入删除的位置，总的元素数量

1. 有序： 每个中间节点都有且只有唯一前驱和唯一后继

不一定物理相邻

有序表：每个中间节点都有且只有唯一前驱和唯一后继

def binary\_search(L, left, right, x):

    if left >= right - 1:

        return right

    mid = int((left + right) / 2)

    if L[mid] < x:

        return binary\_search(L, mid, right, x)

    elif L[mid] > x:

        return binary\_search(L, left, mid, x)

    else:

        return mid

x\_index = binary\_search(L, 0, length, x)

for index in range(x\_index + 1, length + 1)[::-1]:

    L[index] = L[index - 1]

L[x\_index] = x

6.

int ListLength(List\* L) {

List\* pointer = L;

int length = 0;

for (; pointer != nullptr; length++) {

pointer = pointer->next;

}

return length;

}

7.

List\* remove\_repetitive(unordered\_set<int> visited\_value, List\* head, List\* pointer, List\* last\_pointer = nullptr) {

if (pointer == nullptr) {

return head;

}

int value = pointer->value;

if (visited\_value.find(value) != visited\_value.end()) {

if (last\_pointer != nullptr) {

last\_pointer->next = pointer->next;

delete pointer;

return remove\_repetitive(visited\_value, head, last\_pointer->next, last\_pointer);

}

else {

head = pointer->next;

delete pointer;

return remove\_repetitive(visited\_value, head, head->next, head);

}

}

else {

visited\_value.insert(value);

return remove\_repetitive(visited\_value, head, pointer->next, pointer);

}

}

unordered\_set<int> visited\_value(ListLength(L));

L = remove\_repetitive(visited\_value, L, L);

8.

void remove\_between(vector<int> & A, int x, int y) {

vector<int>::iterator iter = A.begin();

while (iter != A.end()) {

if (\*iter >= x && \*iter <= y) {

iter = A.erase(iter);

}

else {

iter++;

}

}

}

9.

List\* merge(List\* pointer1, List\* pointer2, List\* tail = nullptr) {

if (pointer1 == nullptr && pointer2 == nullptr) {

return tail;

}

if (pointer1 != nullptr && (pointer2 == nullptr || pointer1->value < pointer2->value)) {

List\* pointer1\_next = pointer1->next;

pointer1->next = tail;

return merge(pointer1\_next, pointer2, pointer1);

}

else {

List\* pointer2\_next = pointer2->next;

pointer2->next = tail;

return merge(pointer1, pointer2\_next, pointer2);

}

}