**2022212153 陈祥烨 第十一章作业**

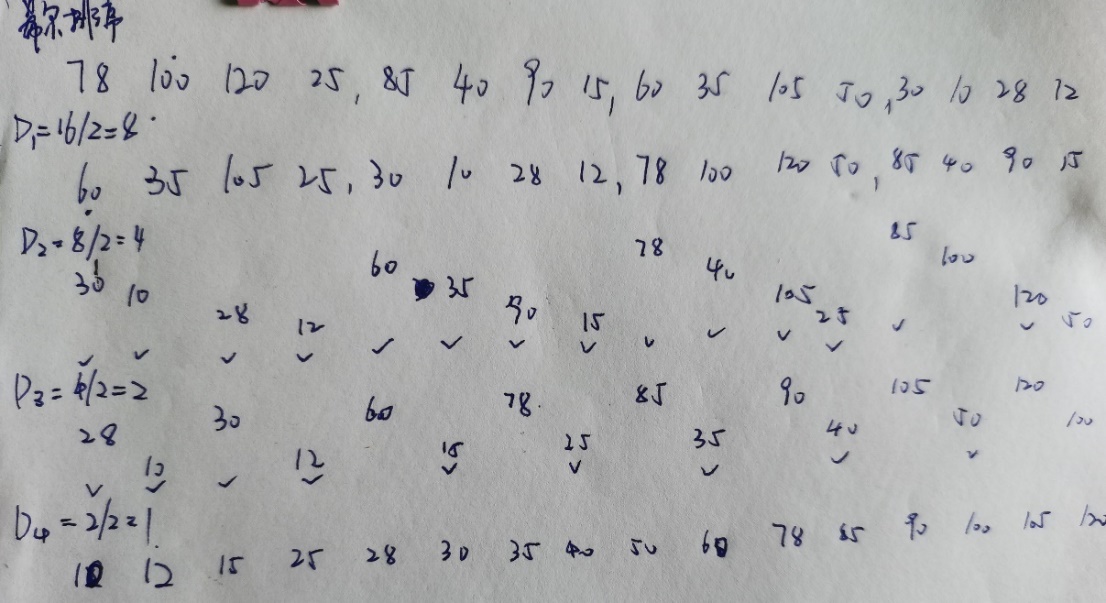
**第一题**

**题目：**

**3.对下面数据表，写出采用希尔排序算法排序的每趟的结果。 (2\*)**

**(78 100 120 25 85 40 90 15 60 35 105 50 30 10 28 12)**

**手绘结果：**

****

**思路：**

将待排序列划分为若干组，分组一般为上次步长的一半，在每组内进行直接插入排序，以使整个序列基本有序，然后再对整个序列进行直接插入排序。

**代码：**

#include<iostream>

using namespace std;

void Print(int A[], int n)

{

for (int i = 0; i < n; i++)

{

cout << A[i] << " ";

}

cout << endl;

}

void Shell\_Sort(int A[],int n)

{

int d = n / 2; //步长

for(;d>0;d /= 2)

{

for (int i = d; i < n; i++) //分组

{

//插入排序

int x = A[i];

int j = i - d;

for (;j >= 0 && x < A[j]; j -= d)

{

A[j + d] = A[j];

}

A[j + d] = x;

}

cout << "步长为：" << d << endl;

Print(A, n);

}

}

void test()

{

int A[16] = { 78, 100, 120, 25, 85, 40, 90, 15, 60, 35, 105, 50, 30, 10, 28, 12};

Shell\_Sort(A, 16);

}

int main()

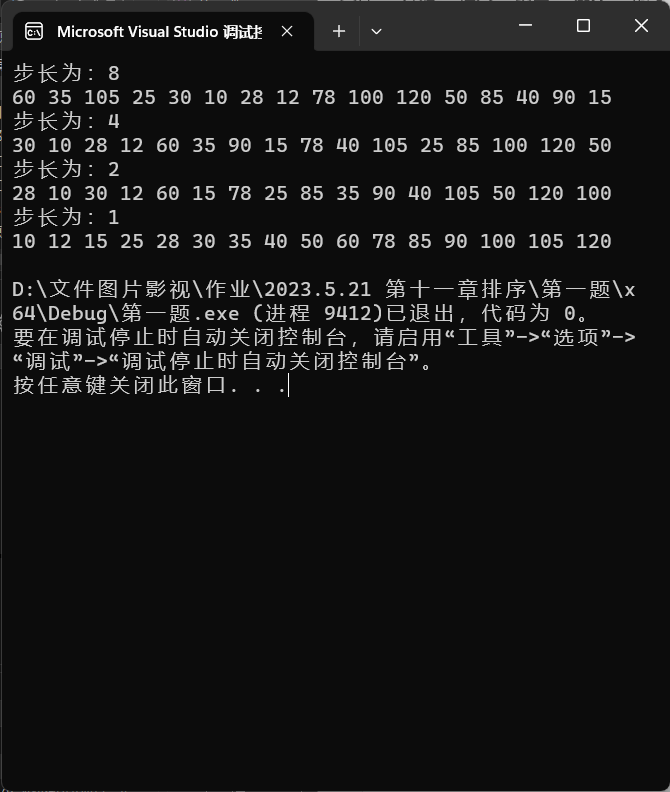
{

test();

return 0;

}

**测试：**

****

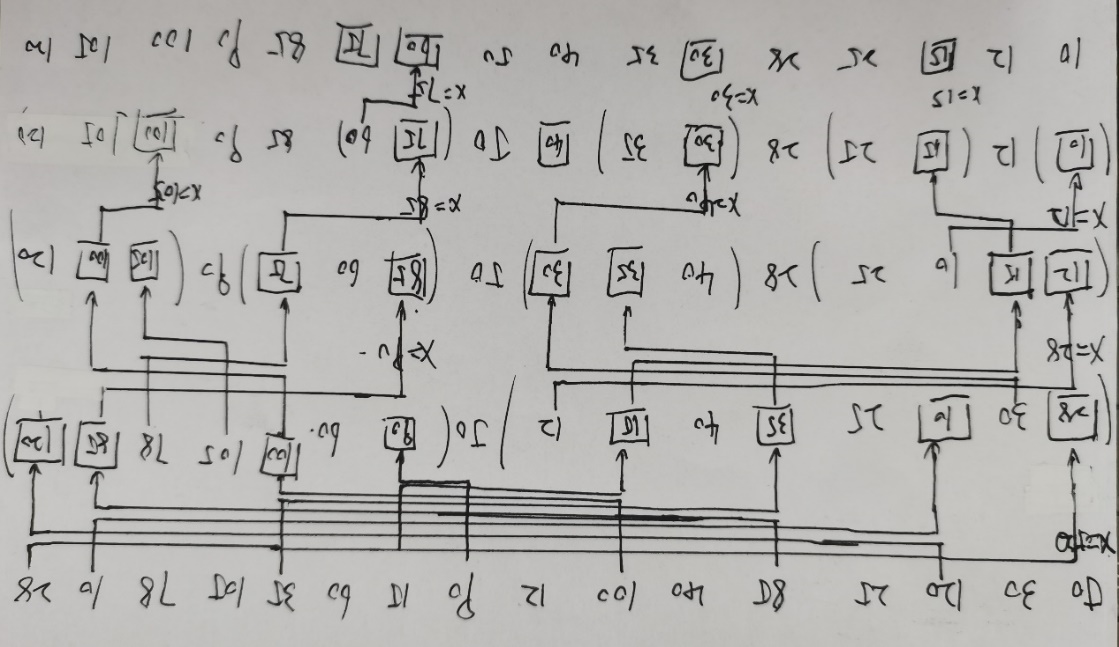
**第二题**

**题目：**

**5. 对下面数据表，写出采用快速排序算法排序的每趟的结果，并标明每趟的数据移动情况。 (3\*)**

**(50 30 120 25 85 40 100 12 90 15 60 35 105 78 10 28)**

**手绘过程：**

****

**思路：**

1. 选择第一个做中间元素，将该元素放到辅助空间，将原来的位置空出来；
2. 从右边找一个比中间元素小的数放到空位上，将该元素原位置空出来，大的数保持位置不变（如果没有小的数，说明所有数都大于中间元素）；
3. 从左边找一个比中间元素大的数放到空位上，将该元素原位置空出来，小的数保持位置不变（如果没有大的数，说明所有数都小于中间元素）；
4. 重复②③操作，直到两边的空位重合，此时将中间元素放入空位中。然后将中间元素两边化为两个部分，依次从①开始；

**代码：**

#include<iostream>

using namespace std;

void Print(int A[], int n)

{

for (int i = 0; i < n; i++)

{

cout << A[i] << " ";

}

cout << endl;

}

//对数组A中下标为s～t的子表进行划分

void partition(int A[], int s, int t, int\* cutpoint){

int x = A[s]; //保存中间元素，腾出空位

int i = s, j = t; //更新上下标

while (i != j) {

while (i<j && A[j] > x)

j--;

if (i < j) {

A[i] = A[j]; i = i + 1;

}

while (i < j && A[i] < x)

i++;

if (i < j) {

A[j] = A[i]; j = j - 1;

}

}

A[i] = x;

\*cutpoint = i;

}

//对数组A中下标从s到t的元素组成的子表快速排序

void Quick\_sort(int A[], int s, int t) {

if (s < t) {

int\* i = new int(0);

partition(A, s, t, i); //划分

cout << "快速排序 从 " << s << " 到 " << t << " :" << endl;

Print(A, 16);

Quick\_sort(A, s, \*i - 1); //对前面子表快速排序

Quick\_sort(A, \*i + 1, t); //对后面子表快速排序

}

}

void test(){

int A[16] = { 50, 30, 120, 25, 85, 40, 100, 12, 90, 15, 60, 35, 105, 78, 10, 28};

Quick\_sort(A, 0, 15);

}

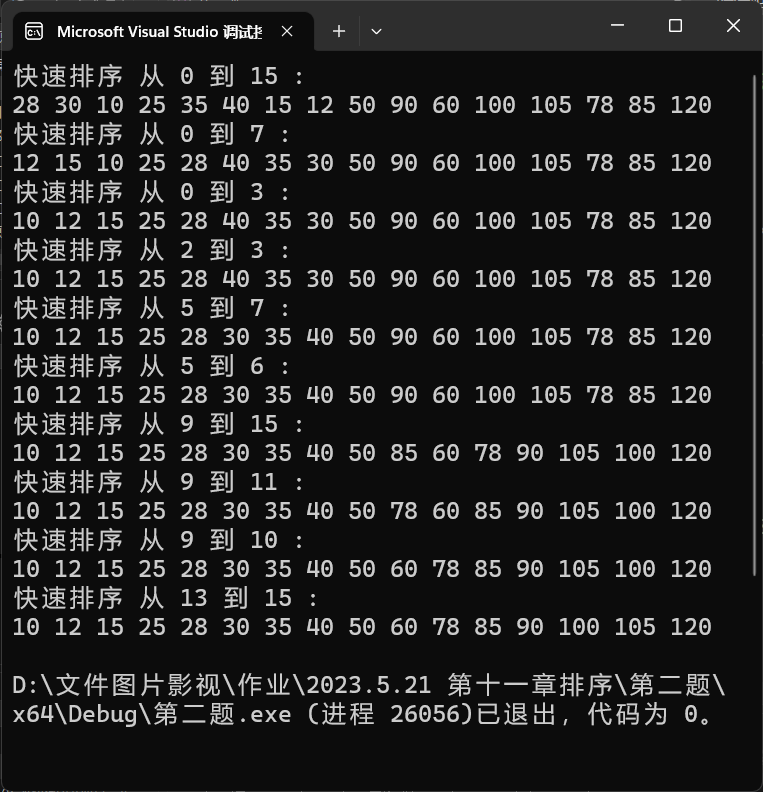
int main(){

test();

return 0;

}

**测试：**

****

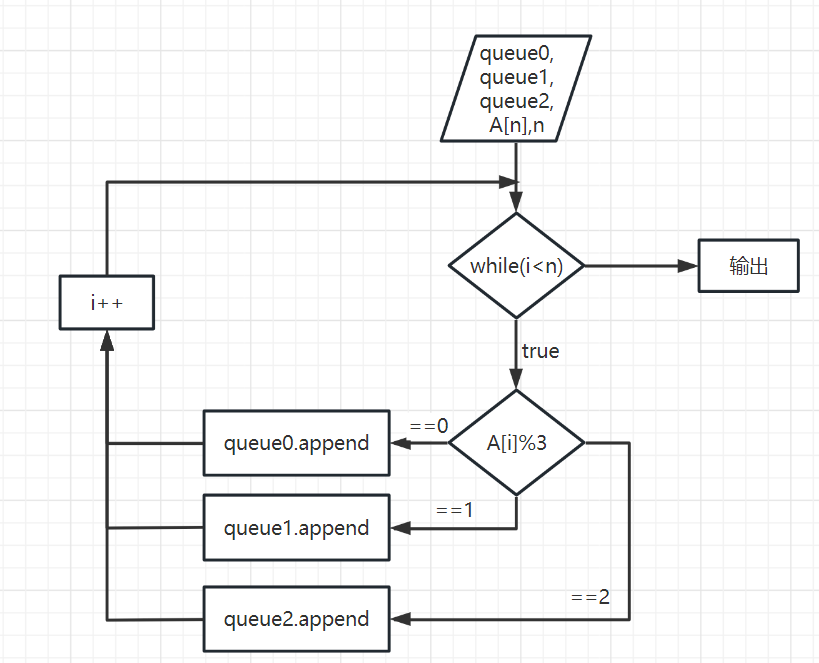
**第三题**

**题目：**

**7. 已知数组A[n]中的元素为整型，设计算法将其调整为三部分，其中左边所有元素为3的倍数，中间所有元素除3余1，右边所有元素除3余2，并要求时间复杂度为O(n)。 (3\*)**

**思路：**

应用桶排序的思想。遍历数组，除以3求余数，会有0,1,2三个不同的结果，按照结果分类，装到不同的队列里，全部遍历结束后，将按照题目所给顺序依次遍历。符合数目虽大，但分类情况少的桶排序的特点。按照一定顺序输出，故这里用队列做为“桶”的结构。



**代码：**

**Bucket Sort.cpp**

#include<iostream>

#include"Queue.h"

using namespace std;

void Print(int A[], int n){

for (int i = 0; i < n; i++){

cout << A[i] << " ";

}

cout << endl;

}

void Bucket\_Sort(int A[], int n){

Queue queue0;

Queue queue1;

Queue queue2;

for (int i = 0; i < n; i++){

if (A[i] % 3 == 0)queue0.append(A[i]);

else if (A[i] % 3 == 1)queue1.append(A[i]);

else queue2.append(A[i]);

}

int index = 0;

while (queue0.get\_front(A[index])) {

queue0.serve();

index++;

}

while (queue1.get\_front(A[index])) {

queue1.serve();

index++;

}

while (queue2.get\_front(A[index])) {

queue2.serve();

index++;

}

}

void test1(){

int A[10] = { 46,1,2,3,4,5,6,7,8,9 };

cout << "46,1,2,3,4,5,6,7,8,9" << endl;

Bucket\_Sort(A, 10);

Print(A, 10);

cout << endl;

}

void test2() {

int A[10] = { 0,1,2,74,4,5,6,45,8,9 };

cout << "0,1,2,74,4,5,6,45,8,9" << endl;

Bucket\_Sort(A, 10);

Print(A, 10);

cout << endl;

}

void test3() {

int A[9] = { 1,2,3,4,5,45,7,8,66 };

cout << "1,2,3,4,5,45,7,8,66" << endl;

Bucket\_Sort(A,9);

Print(A, 9);

cout << endl;

}

int main(){

test1();

test2();

test3();

return 0;

}

**Queue.h**

#pragma once

#include<iostream>

using namespace std;

struct QNode{

int data;

QNode\* next;

QNode(){

next = NULL;

}

QNode(int x){

data = x;

next = NULL;

}

};

class Queue{

private:

int \_count;

QNode\* \_head;

QNode\* \_rear;

public:

Queue();

Queue(int A[],int n);

Queue(Queue& queue);

~Queue();

bool Empty() const; //队空

int get\_length() const; //获得长度

QNode\* get\_head() const; //获得头结点指针

bool get\_front(int& x)const; //获取首元素

void append(const int x); //入队

bool serve(); //出队

private:

void Destroy(QNode\* node); //删除该结点以及以后的结点

};

Queue::Queue(){

\_count = 0;

\_head = new QNode;

\_rear = \_head;

}

Queue::Queue(int A[], int n) {

\_count = 0;

\_head = new QNode;

\_rear = \_head;

for (int i = 0; i < n; i++) {

QNode\* node = new QNode(A[i]);

\_rear->next = node;

\_rear = node;

}

\_count = n;

}

Queue::Queue(Queue& queue){

\_count = queue.get\_length();

\_head = new QNode;

\_rear = \_head;

for (QNode\* cur = queue.get\_head()->next; cur != NULL; cur = cur->next) {

QNode\* node = new QNode(cur->data);

\_rear->next = node;

\_rear = node;

}

}

Queue::~Queue()

{

Destroy(\_head);

\_head = NULL;

\_rear = \_head;

\_count = 0;

}

bool Queue::Empty() const {

if (\_count == 0)return true;

else return false;

}

int Queue::get\_length() const {

return \_count;

}

bool Queue::get\_front(int& x)const {

if (\_head->next == NULL)return false;

x = \_head->next->data;

return true;

}

QNode\* Queue::get\_head() const {

return \_head;

}

void Queue::append(const int x) {

QNode\* node = new QNode(x);

\_rear->next = node;

\_rear = node;

\_count++;

}

bool Queue::serve() {

if (Empty())return false;

QNode\* temp = \_head->next;

\_head->next = \_head->next->next;

delete temp;

\_count--;

if (Empty())\_rear = \_head;

return true;

}

void Queue::Destroy(QNode\* node) {

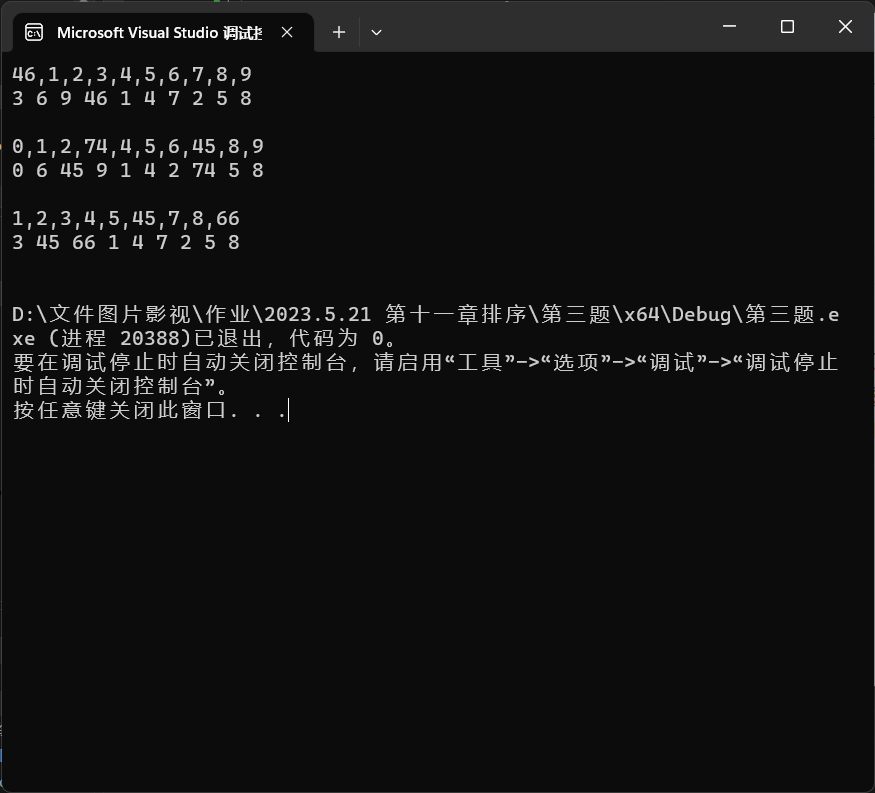
if (node == NULL) return;

Destroy(node->next);

delete node;

}

**测试：**

****

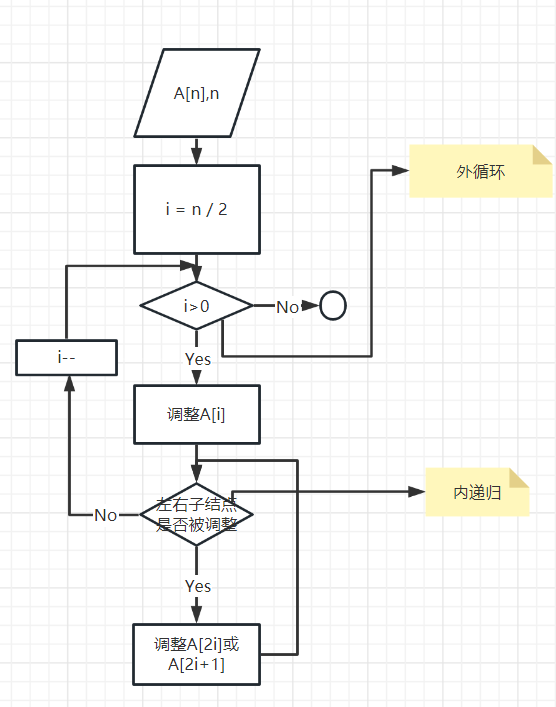
**第四题**

**题目：**

**10. 将下面数据表分别调整为大根堆和小根堆。 (2\*)**

**(50 30 120 25 85 40 100 12 90 15 60 35 105 78 10 28)**

**思路：**

****

**代码：**

#include<iostream>

using namespace std;

//---------------------------工具包------------------

void Print(int A[], int n) {

for (int i = 1; i <= n; i++) {

cout << A[i] << " ";

}

cout << endl;

}

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

//-------------------------大根堆调整----------------------------

void adjust\_big(int A[], int n, int i, bool& left, bool& right) {

if (2 \* i > n) { //左孩子为空

return;

}

else {

if (2 \* i + 1 > n) { //右孩子为空，左孩子不空

if (A[2 \* i] > A[i]) {

int temp = A[2 \* i];

A[2 \* i] = A[i];

A[i] = temp;

left = 1;

}

}

else { //左右孩子都不空

if (A[i] >= A[2 \* i] && A[i] >= A[2 \* i + 1]) { //根最大

return;

}

else if (A[2 \* i] >= A[2 \* i + 1]) { //左孩子最大

swap(A[i], A[2 \* i]);

left = 1;

}

else { //右孩子最大

swap(A[i], A[2 \* i + 1]);

right = 1;

}

}

}

}

void traversal\_big(int A[], int n,int i) { //内部递归

bool left = 0, right = 0; //判断左右孩子是否被调整，0为未调整，1为已被调整，被调整的话则以此为根重新调整

adjust\_big(A, n, i,left,right);

if (left == 1) traversal\_big(A, n, i \* 2);

if (right == 1) traversal\_big(A, n, i \* 2 + 1);

}

void heap\_sort\_big(int A[], int n) { //外部循环

int i = n / 2 ;

for (; i > 0; i--) {

traversal\_big(A, n, i);

}

}

//-------------------------小根堆调整----------------------------

void adjust\_small(int A[], int n, int i, bool& left, bool& right) {

if (2 \* i > n) { //左孩子为空

return;

}

else {

if (2 \* i + 1 > n) { //右孩子为空，左孩子不空

if (A[2 \* i] > A[i]) {

int temp = A[2 \* i];

A[2 \* i] = A[i];

A[i] = temp;

left = 1;

}

}

else { //左右孩子都不空

if (A[i] <= A[2 \* i] && A[i] <= A[2 \* i + 1]) { //根最小

return;

}

else if (A[2 \* i] <= A[2 \* i + 1]) { //左孩子最小

swap(A[i], A[2 \* i]);

left = 1;

}

else { //右孩子最小

swap(A[i], A[2 \* i + 1]);

right = 1;

}

}

}

}

void traversal\_small(int A[], int n, int i) {

bool left = 0, right = 0; //判断左右孩子是否被调整，0为未调整，1为已被调整，被调整的话则以此为根重新调整

adjust\_small(A, n, i, left, right);

if (left == 1) traversal\_small(A, n, i \* 2);

if (right == 1) traversal\_small(A, n, i \* 2 + 1);

}

void heap\_sort\_small(int A[], int n) {

int i = n / 2;

for (; i > 0; i--) {

traversal\_small(A, n, i);

}

}

void test() {

cout << "A:" << endl;

int A[17] = {0, 50, 30, 120, 25, 85, 40, 100, 12, 90, 15, 60, 35, 105, 78, 10, 28 };

Print(A, 16);

heap\_sort\_big(A, 16);

cout << "大根堆：";

Print(A, 16);

heap\_sort\_small(A, 16);

cout << "小根堆：";

Print(A, 16);

cout << "B:" << endl;

int B[16] = { 0,12,15,30,80,100,46,78,33,90,86,64,55,120,230,45 };

Print(B, 15);

heap\_sort\_big(B,15);

cout << "大根堆：";

Print(B, 15);

heap\_sort\_small(B, 15);

cout << "小根堆：";

Print(B, 15);

}

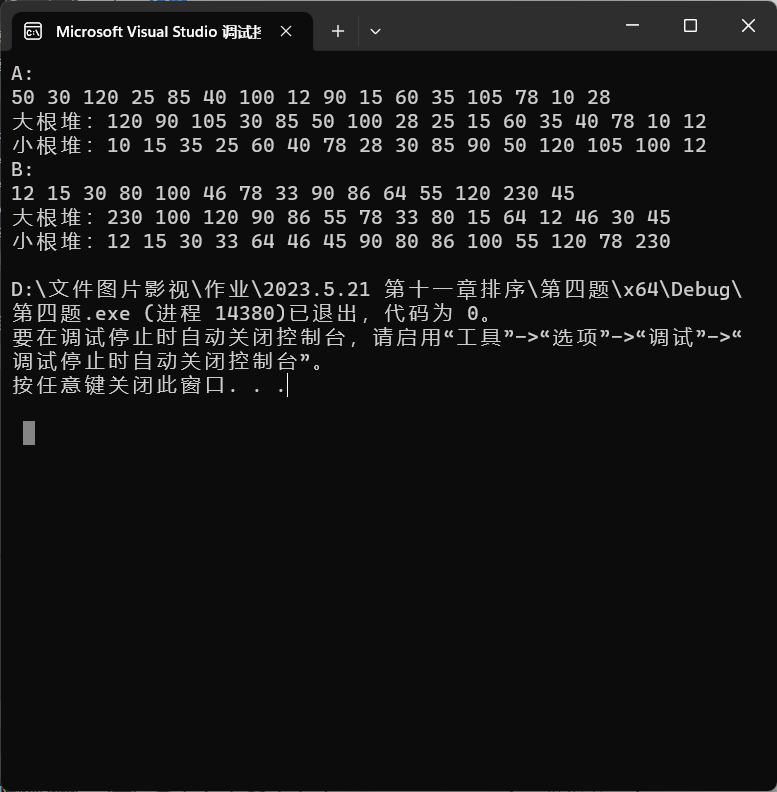
int main() {

test();

return 0;

}

**测试：**

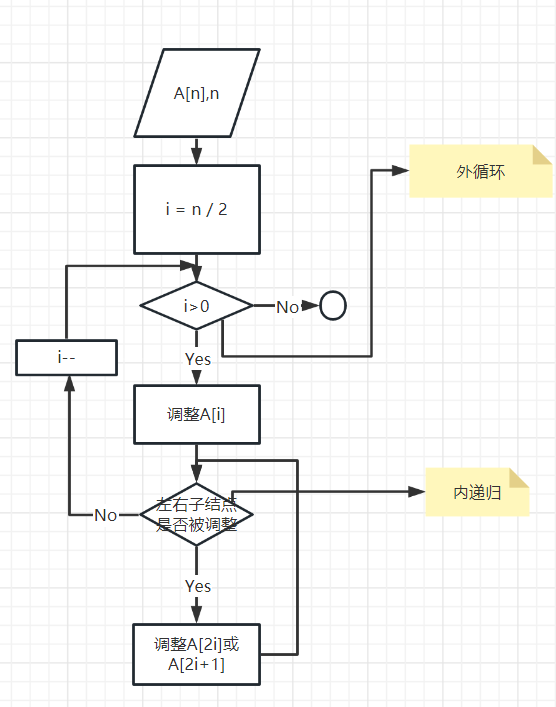
****

**第五题**

**题目：**

**11. 由初始建堆过程的讨论可知，调整过程可以是递归形式的，请写出这一递归形式的算法。 (3\*)**

**思路：**

****

**代码：**

#include<iostream>

using namespace std;

//---------------------------工具包------------------

void Print(int A[], int n) {

for (int i = 1; i <= n; i++) {

cout << A[i] << " ";

}

cout << endl;

}

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

//-------------------------大根堆调整----------------------------

void adjust\_big(int A[], int n, int i, bool& left, bool& right) {

if (2 \* i > n) { //左孩子为空

return;

}

else {

if (2 \* i + 1 > n) { //右孩子为空，左孩子不空

if (A[2 \* i] > A[i]) {

int temp = A[2 \* i];

A[2 \* i] = A[i];

A[i] = temp;

left = 1;

}

}

else { //左右孩子都不空

if (A[i] >= A[2 \* i] && A[i] >= A[2 \* i + 1]) { //根最大

return;

}

else if (A[2 \* i] >= A[2 \* i + 1]) { //左孩子最大

swap(A[i], A[2 \* i]);

left = 1;

}

else { //右孩子最大

swap(A[i], A[2 \* i + 1]);

right = 1;

}

}

}

}

void traversal\_big(int A[], int n,int i) { //内部递归

bool left = 0, right = 0; //判断左右孩子是否被调整，0为未调整，1为已被调整，被调整的话则以此为根重新调整

adjust\_big(A, n, i,left,right);

if (left == 1) traversal\_big(A, n, i \* 2);

if (right == 1) traversal\_big(A, n, i \* 2 + 1);

}

void heap\_sort\_big(int A[], int n) { //外部循环

int i = n / 2 ;

for (; i > 0; i--) {

traversal\_big(A, n, i);

}

}

//-------------------------小根堆调整----------------------------

void adjust\_small(int A[], int n, int i, bool& left, bool& right) {

if (2 \* i > n) { //左孩子为空

return;

}

else {

if (2 \* i + 1 > n) { //右孩子为空，左孩子不空

if (A[2 \* i] > A[i]) {

int temp = A[2 \* i];

A[2 \* i] = A[i];

A[i] = temp;

left = 1;

}

}

else { //左右孩子都不空

if (A[i] <= A[2 \* i] && A[i] <= A[2 \* i + 1]) { //根最小

return;

}

else if (A[2 \* i] <= A[2 \* i + 1]) { //左孩子最小

swap(A[i], A[2 \* i]);

left = 1;

}

else { //右孩子最小

swap(A[i], A[2 \* i + 1]);

right = 1;

}

}

}

}

void traversal\_small(int A[], int n, int i) {

bool left = 0, right = 0; //判断左右孩子是否被调整，0为未调整，1为已被调整，被调整的话则以此为根重新调整

adjust\_small(A, n, i, left, right);

if (left == 1) traversal\_small(A, n, i \* 2);

if (right == 1) traversal\_small(A, n, i \* 2 + 1);

}

void heap\_sort\_small(int A[], int n) {

int i = n / 2;

for (; i > 0; i--) {

traversal\_small(A, n, i);

}

}

void test() {

cout << "A:" << endl;

int A[17] = {0, 50, 30, 120, 25, 85, 40, 100, 12, 90, 15, 60, 35, 105, 78, 10, 28 };

Print(A, 16);

heap\_sort\_big(A, 16);

cout << "大根堆：";

Print(A, 16);

heap\_sort\_small(A, 16);

cout << "小根堆：";

Print(A, 16);

cout << "B:" << endl;

int B[16] = { 0,12,15,30,80,100,46,78,33,90,86,64,55,120,230,45 };

Print(B, 15);

heap\_sort\_big(B,15);

cout << "大根堆：";

Print(B, 15);

heap\_sort\_small(B, 15);

cout << "小根堆：";

Print(B, 15);

}

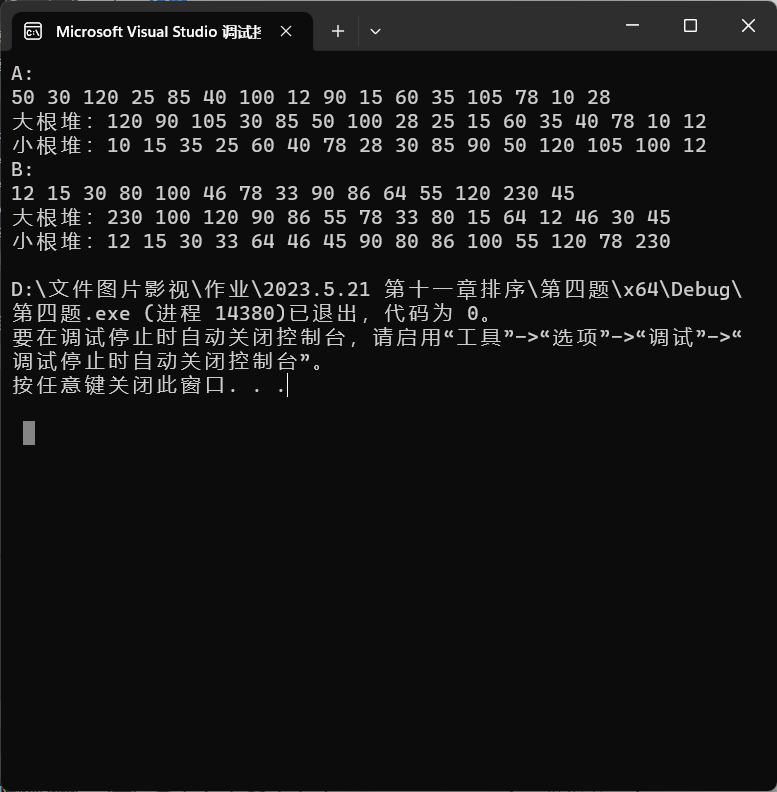
int main() {

test();

return 0;

}

**测试：**

****

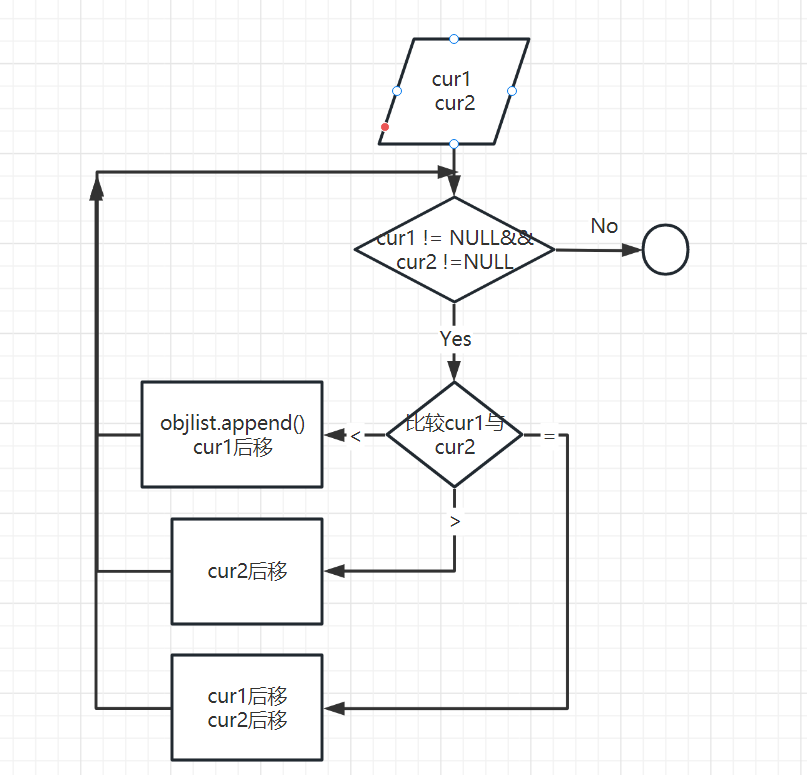
**第六题**

**题目：**

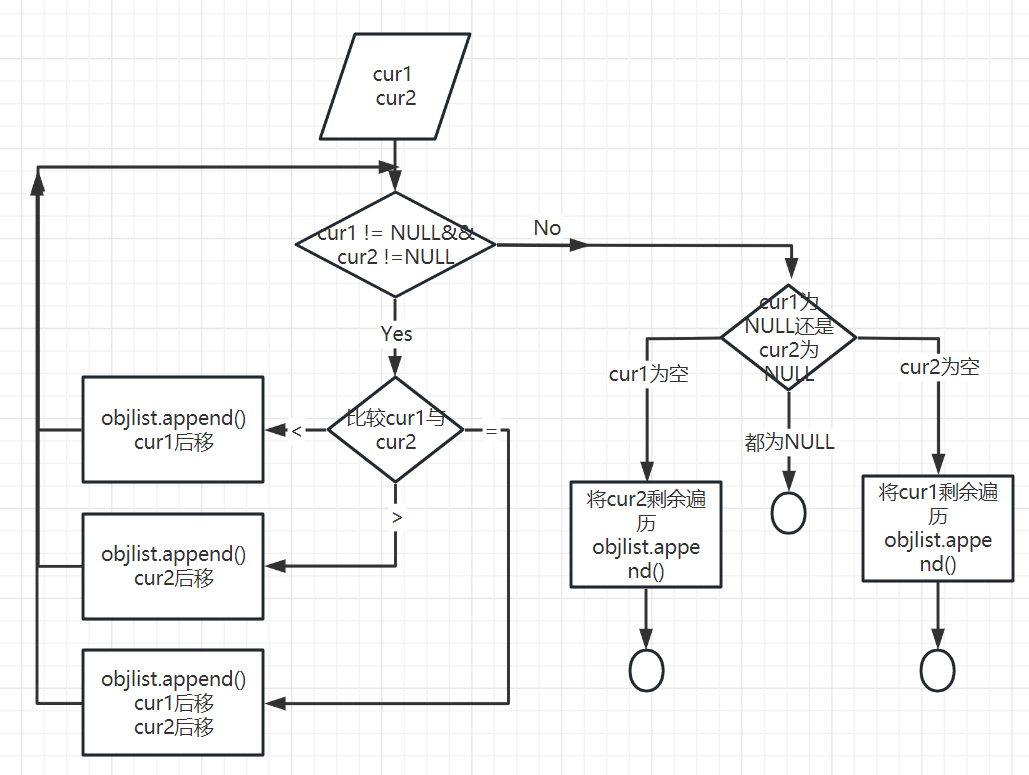
**13. 如果递增有序的数据表A、B分别代表一个集合，设计算法以求解此类集合交、并差等运算。 (3\*)**

**思路：**

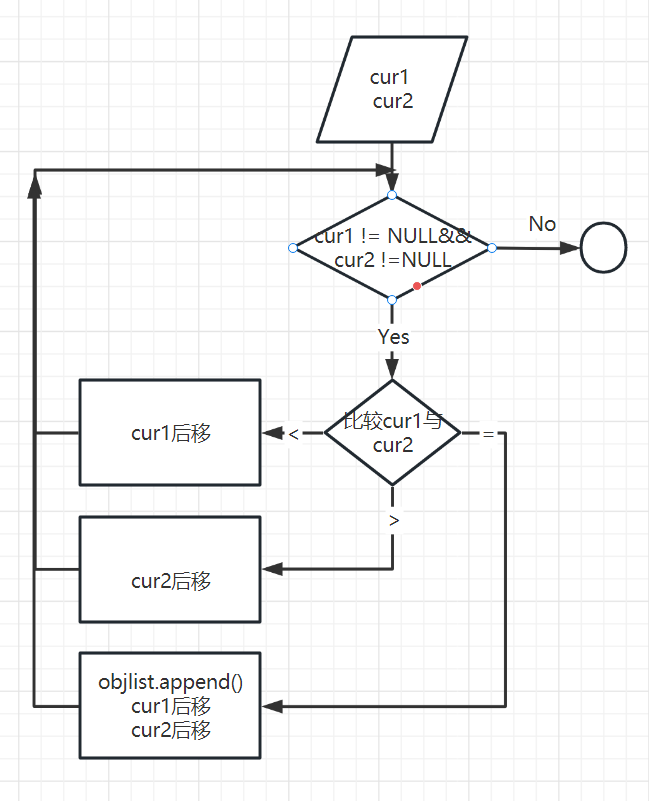
差集



交集



并集



**代码：**

**Linked List.h**

#pragma once

#include<iostream>

using namespace std;

struct Node

{

int data;

Node\* next;

};

class List //单链表

{

public:

//构造

List(); //默认构造函数

List(List& list); //拷贝构造函数

List(int A[], int n); //数组构造函数

~List(); //析构函数

//获取链表成员变量

int length()const; //求长度

Node\* get\_head() { return head; } //读取链表表头指针

//操作结点

void append(int n); //尾增

//操作链表

void print()const; //遍历输出

void get\_intersection(List list, List& objlist); //交集

void get\_union(List list, List& objlist); //并集

void get\_sub(List list, List& objlist); //减集

private:

int count;

Node\* head;

Node\* rear;

};

void List::get\_sub(List list, List& objlist) {

Node\* cur1 = head->next;

Node\* cur2 = list.get\_head()->next;

while (cur1 != NULL && cur2 != NULL) {

if (cur1->data < cur2->data) {

objlist.append(cur1->data);

cur1 = cur1->next;

}

else if (cur1->data > cur2->data) {

cur2 = cur2->next;

}

else {

cur1 = cur1->next;

cur2 = cur2->next;

}

}

}

void List::get\_union(List list, List& objlist) {

Node\* cur1 = head->next;

Node\* cur2 = list.get\_head()->next;

while (cur1 != NULL && cur2 != NULL) {

if (cur1->data < cur2->data) {

objlist.append(cur1->data);

cur1 = cur1->next;

}

else if(cur1->data > cur2->data){

objlist.append(cur2->data);

cur2 = cur2->next;

}

else {

objlist.append(cur1->data);

cur1 = cur1->next;

cur2 = cur2->next;

}

}

if (cur1 == NULL && cur2 == NULL) return;

else if (cur1 == NULL) {

while (cur2 != NULL) {

objlist.append(cur2->data);

cur2 = cur2->next;

}

}

else {

while (cur1 != NULL) {

objlist.append(cur1->data);

cur1 = cur1->next;

}

}

}

void List::get\_intersection(List list, List& objlist)

{

if (count == 0 || list.length() == 0)return; //空链表特殊处理

Node\* cur1 = head->next;

Node\* cur2 = list.get\_head()->next;

for (; cur1 != NULL && cur2 != NULL; cur1 = cur1->next) //cur1遍历

{

Node\* temp = cur2;

bool isadd = 0;

for (; cur2->data <= cur1->data && cur2 != NULL; cur2 = cur2->next)

{

if (cur1->data == cur2->data) //找到相同元素，添加

{

objlist.append(cur1->data);

isadd = 1;

cur2 = cur2->next;

break;

}

}

if (isadd == 0)cur2 = temp; //未找到，cur2回归

}

}

void List::append(int n)

{

Node\* node = new Node;

node->data = n;

node->next = NULL;

rear->next = node;

rear = node;

count++;

}

void List::print()const

{

if (count == 0)

{

cout << " is NULL!" << endl;

return;

}

Node\* PA = head;

while (PA->next != NULL)

{

PA = PA->next;

cout << PA->data << " ";

}

cout << endl;

}

List::List()

{

head = new Node;

head->next = NULL;

rear = head;

count = 0;

}

List::List(List& list)

{

head = new Node;

head->next = NULL;

rear = head;

count = 0;

Node\* PA = list.get\_head()->next;

Node\* PB = head; //设置当前表，即B表尾指针

while (PA != NULL) //在A表中还有元素时

{

Node\* s = new Node; //产生结点

s->data = PA->data; //复制结点的值

PB->next = s; // 插入复制的结点到表尾

PB = s; //重新指示表尾，以便下一个元素插入

rear = s;

count++; //继续复制A表的下一个元素

PA = PA->next;

PB->next = NULL; //将B表尾结点的后继指针置为空

}

}

List::List(int A[], int n)

{

head = new Node;

head->next = NULL;

rear = head;

Node\* PA = head;

for (int i = 0; i < n; i++)

{

Node\* s = new Node;

s->data = A[i];

s->next = NULL;

PA->next = s;

PA = s;

}

rear = PA;

count = n;

}

List::~List()

{

while (head != rear)

{

Node\* temp = head;

head = head->next;

delete temp;

}

delete head;

}

int List::length() const

{

return count;

}

**test.cpp**

#include"Linked List.h"

void test1()

{

cout << "test01:" << endl;

int A[10] = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };

int B[13] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 20, 30 };

List list1(A, 10);

List list2(B, 13);

cout << "list1:"; list1.print();

cout << "list2:"; list2.print();

List objlist;

list1.get\_intersection(list2, objlist);

cout << "交集:"; objlist.print();

}

void test2()

{

cout << "test02:" << endl;

int A[10] = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };

int B[13] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 20, 30 };

List list1(A, 10);

List list2(B, 13);

cout << "list1:"; list1.print();

cout << "list2:"; list2.print();

List objlist;

list1.get\_union(list2, objlist);

cout << "并集:"; objlist.print();

}

void test3()

{

cout << "test03:" << endl;

int A[10] = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };

int B[13] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 20, 30 };

List list1(A, 10);

List list2(B, 13);

cout << "list1:"; list1.print();

cout << "list2:"; list2.print();

List objlist;

list1.get\_sub(list2, objlist);

cout << "差集:"; objlist.print();

}

int main()

{

test1();

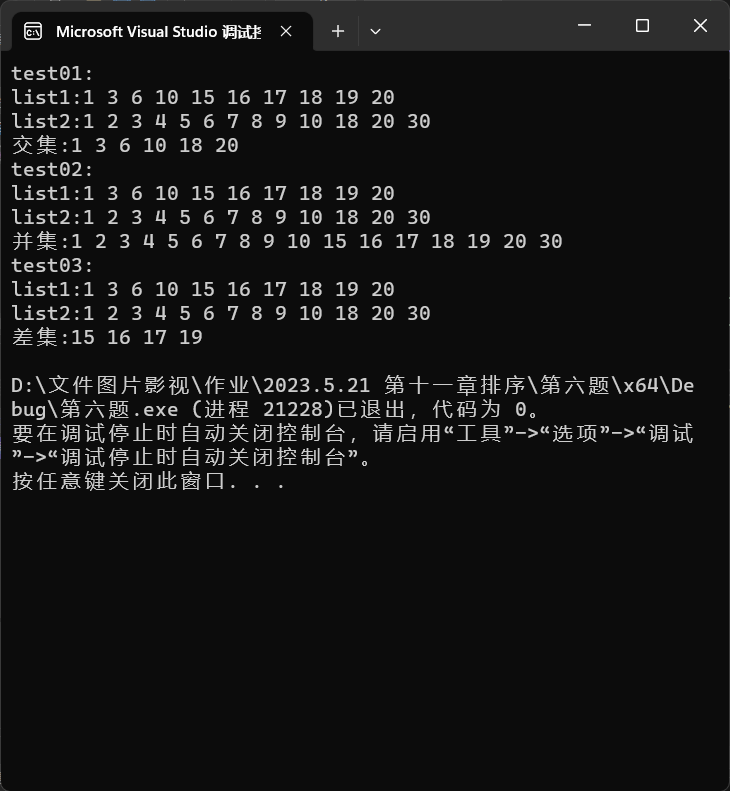
test2();

test3();

return 0;

}

**测试：**

****