BinarySearch.h

// 二分查找（折半查找）：对于已排序，若无序，需要先排序  
  
// 非递归  
int BinarySearch(vector<int> v, int value)  
{  
 if (v.size() <= 0)  
 return -1;  
 int low = 0;  
 int high = v.size() - 1;

int mid = 0;  
 while (low <= high)  
 {  
 mid = low + (high - low) / 2;  
 if (v[mid] == value)  
 return mid;  
 else if (v[mid] > value)  
 high = mid - 1;  
 else  
 low = mid + 1;  
 }  
 return -1;  
}  
  
// 递归  
int BinarySearch2(vector<int> v, int value, int low, int high)  
{  
 if (low > high)  
 return -1;  
 int mid = low + (high - low) / 2;  
 if (v[mid] == value)  
 return mid;  
 else if (v[mid] > value)  
 return BinarySearch2(v, value, low, mid - 1);  
 else  
 return BinarySearch2(v, value, mid + 1, high);  
}

BSTSearch.h

二叉搜索树的查找算法:  
在二叉搜索树b中查找x的过程为：  
1. 若b是空树，则搜索失败，否则：  
2. 若x等于b的根节点的数据域之值，则查找成功；否则：  
3. 若x小于b的根节点的数据域之值，则搜索左子树；否则：  
4. 查找右子树。   
  
// 在根指针T所指二叉查找树中递归地查找其关键字等于key的数据元素，若查找成功，  
// 则指针p指向該数据元素节点，并返回TRUE，否则指针指向查找路径上访问的最终  
// 一个节点并返回FALSE，指针f指向T的双亲，其初始调用值为NULL  
Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree &p){  
  
 if(!T) { //查找不成功  
 p=f;  
 return false;  
 }  
 else if (key == T->data.key) { //查找成功  
 p=T;  
 return true;  
 }  
 else if (key < T->data.key) //在左子树中继续查找  
 return SearchBST(T->lchild, key, T, p);  
 else //在右子树中继续查找  
 return SearchBST(T->rchild, key, T, p);  
}

BubbleSort.h

（无序区，有序区）。从无序区通过交换找出最大元素放到有序区前端。  
选择排序思路：  
1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。  
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。  
3. 针对所有的元素重复以上的步骤，除了最后一个。  
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。  
  
// 冒泡排序  
void BubbleSort(vector<int>& v) {  
 int len = v.size();  
 for (int i = 0; i < len - 1; ++i)  
 for (int j = 0; j < len - 1 - i; ++j)  
 if (v[j] > v[j + 1])   
 swap(v[j], v[j + 1]);  
}  
  
// 模板实现冒泡排序  
template<typename T> //整數或浮點數皆可使用,若要使用物件(class)時必須設定大於(>)的運算子功能  
void bubble\_sort(T arr[], int len) {  
 for (int i = 0; i < len - 1; i++)  
 for (int j = 0; j < len - 1 - i; j++)  
 if (arr[j] > arr[j + 1])  
 swap(arr[j], arr[j + 1]);  
}  
  
// 冒泡排序（改进版）  
void BubbleSort\_orderly(vector<int>& v) {  
 int len = v.size();  
 bool orderly = false;  
 for (int i = 0; i < len - 1 && !orderly; ++i) {  
 orderly = true;  
 for (int j = 0; j < len - 1 - i; ++j) {  
 if (v[j] > v[j + 1]) { // 从小到大  
 orderly = false; // 发生交换则仍非有序  
 swap(v[j], v[j + 1]);  
 }  
 }  
 }  
}

BucketSort.cpp

#include<iterator>  
#include<iostream>  
#include<vector>  
using std::vector;  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
桶排序：将值为i的元素放入i号桶，最后依次把桶里的元素倒出来。  
  
桶排序序思路：  
1. 设置一个定量的数组当作空桶子。  
2. 寻访序列，并且把项目一个一个放到对应的桶子去。  
3. 对每个不是空的桶子进行排序。  
4. 从不是空的桶子里把项目再放回原来的序列中。  
  
假设数据分布在[0，100)之间，每个桶内部用链表表示，在数据入桶的同时插入排序，然后把各个桶中的数据合并。  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
  
const int BUCKET\_NUM = 10;  
  
struct ListNode{  
 explicit ListNode(int i=0):mData(i),mNext(NULL){}  
 ListNode\* mNext;  
 int mData;  
};  
  
ListNode\* insert(ListNode\* head,int val){  
 ListNode dummyNode;// 假的  
 ListNode \*newNode = new ListNode(val);  
 ListNode \*pre,\*curr;  
 dummyNode.mNext = head;  
 pre = &dummyNode;  
 curr = head;  
 while(NULL!=curr && curr->mData<=val){  
 pre = curr;  
 curr = curr->mNext;  
 }  
 newNode->mNext = curr;  
 pre->mNext = newNode;  
 return dummyNode.mNext;  
}  
  
  
ListNode\* Merge(ListNode \*head1,ListNode \*head2){  
 ListNode dummyNode;  
 ListNode \*dummy = &dummyNode;  
 while(NULL!=head1 && NULL!=head2){  
 if(head1->mData <= head2->mData){  
 dummy->mNext = head1;  
 head1 = head1->mNext;  
 }else{  
 dummy->mNext = head2;  
 head2 = head2->mNext;  
 }  
 dummy = dummy->mNext;  
 }  
 if(NULL!=head1) dummy->mNext = head1;  
 if(NULL!=head2) dummy->mNext = head2;  
   
 return dummyNode.mNext;  
}  
  
void BucketSort(int n,int arr[]){  
 vector<ListNode\*> buckets(BUCKET\_NUM,(ListNode\*)(0));  
 for(int i=0;i<n;++i){  
 int index = arr[i]/BUCKET\_NUM;  
 ListNode \*head = buckets.at(index);  
 buckets.at(index) = insert(head,arr[i]);  
 }  
 ListNode \*head = buckets.at(0);  
 for(int i=1;i<BUCKET\_NUM;++i){  
 head = Merge(head,buckets.at(i));  
 }  
 for(int i=0;i<n;++i){  
 arr[i] = head->mData;  
 head = head->mNext;  
 }  
}

CountSort.cpp

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
计数排序：统计小于等于该元素值的元素的个数i，于是该元素就放在目标数组的索引i位（i≥0）。  
  
计数排序基于一个假设，待排序数列的所有数均为整数，且出现在（0，k）的区间之内。  
如果 k（待排数组的最大值） 过大则会引起较大的空间复杂度，一般是用来排序 0 到 100 之间的数字的最好的算法，但是它不适合按字母顺序排序人名。  
计数排序不是比较排序，排序的速度快于任何比较排序算法。  
时间复杂度为 O（n+k），空间复杂度为 O（n+k）  
  
算法的步骤如下：  
  
1. 找出待排序的数组中最大和最小的元素  
2. 统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项  
3. 对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）  
4. 反向填充目标数组：将每个元素 i 放在新数组的第 C[i] 项，每放一个元素就将 C[i] 减去 1  
  
  
#include <iostream>  
#include <vector>  
#include <algorithm>  
  
using namespace std;

// 计数排序  
void CountSort(vector<int>& vecRaw, vector<int>& vecObj)  
{  
 // 确保待排序容器非空  
 if (vecRaw.size() == 0)  
 return;  
  
 // 使用 vecRaw 的最大值 + 1 作为计数容器 countVec 的大小  
 int vecCountLength = (\*max\_element(begin(vecRaw), end(vecRaw))) + 1;  
 vector<int> vecCount(vecCountLength, 0);  
  
 // 统计每个键值出现的次数  
 for (int i = 0; i < vecRaw.size(); i++)  
 vecCount[vecRaw[i]]++;  
   
 // 后面的键值出现的位置为前面所有键值出现的次数之和  
 for (int i = 1; i < vecCountLength; i++)  
 vecCount[i] += vecCount[i - 1];  
  
 // 将键值放到目标位置  
 for (int i = vecRaw.size(); i > 0; i--)

// 此处逆序是为了保持相同键值的稳定性  
 vecObj[--vecCount[vecRaw[i - 1]]] = vecRaw[i - 1];  
}  
  
int main()  
{  
 vector<int> vecRaw = { 0,5,7,9,6,3,4,5,2,8,6,9,2,1 };  
 vector<int> vecObj(vecRaw.size(), 0);  
  
 CountSort(vecRaw, vecObj);  
  
 for (int i = 0; i < vecObj.size(); ++i)  
 cout << vecObj[i] << " ";  
 cout << endl;  
  
 return 0;  
}

FibonacciSearch.cpp

// 斐波那契查找   
   
#include "stdafx.h"   
#include <memory>   
#include <iostream>   
using namespace std;   
   
const int max\_size=20;//斐波那契数组的长度   
   
/\*构造一个斐波那契数组\*/   
void Fibonacci(int \* F)   
{   
 F[0]=0;   
 F[1]=1;   
 for(int i=2;i<max\_size;++i)   
 F[i]=F[i-1]+F[i-2];   
}

int main()   
{   
 int a[] = {0,16,24,35,47,59,62,73,88,99};   
 int key=100;   
 int index=FibonacciSearch(a,sizeof(a)/sizeof(int),key);   
 cout<<key<<" is located at:"<<index;   
 return 0;   
}

/\*定义斐波那契查找法\*/   
/\*a为要查找的数组,n为要查找的数组长度,key为要查找的关键字\*/

int FibonacciSearch(int \*a, int n, int key)   
{   
 int low=0;   
 int high=n-1;   
   
 int F[max\_size];   
 Fibonacci(F);//构造一个斐波那契数组F   
   
 int k=0;   
 while(n>F[k]-1)//计算n位于斐波那契数列的位置   
 ++k;   
   
 int \* temp;//将数组a扩展到F[k]-1的长度   
 temp=new int [F[k]-1];   
 memcpy(temp,a,n\*sizeof(int));   
   
 for(int i=n;i<F[k]-1;++i)   
 temp[i]=a[n-1];   
   
 while(low<=high)   
 {   
 int mid=low+F[k-1]-1;   
 if(key<temp[mid])   
 {   
 high=mid-1;   
 k-=1;   
 }   
 else if(key>temp[mid])   
 {   
 low=mid+1;   
 k-=2;   
 }   
 else   
 {   
 if(mid<n)   
 return mid; //若相等则说明mid即为查找到的位置   
 else   
 return n-1; //若mid>=n则说明是扩展的数值,返回n-1   
 }   
 }   
 delete [] temp;   
 return -1;   
}

HeapSort.cpp  
#include <iostream>  
#include <algorithm>  
using namespace std;  
// 堆排序：（最大堆，有序区）。从堆顶把根卸出来放在有序区之前，再恢复堆。  
void max\_heapify(int arr[], int start, int end) {  
 //建立父结点指标和子结点指标  
 int dad = start;  
 int son = dad \* 2 + 1;  
 while (son <= end) { //若子结点指标在范围內才做比较  
 if (son + 1 <= end && arr[son] < arr[son + 1])

//先比较兩個子结点大小，选择最大的  
 son++;  
 if (arr[dad] > arr[son])

//如果父结点大於子结点代表调整完成，直接跳出函数  
 return;  
 else { //否則交换父子內容再继续子结点和孙结点比较  
 swap(arr[dad], arr[son]);  
 dad = son;  
 son = dad \* 2 + 1;  
 }  
 }  
}

void heap\_sort(int arr[], int len) {  
 //初始化，i從最後一個父结点開始調整  
 for (int i = len / 2 - 1; i >= 0; i--)  
 max\_heapify(arr, i, len - 1);  
 //先將第一個元素和已经排好的元素前一位做交换，再從新調整(刚调整的元素之前的元素)，直到排序完畢  
 for (int i = len - 1; i > 0; i--) {  
 swap(arr[0], arr[i]);  
 max\_heapify(arr, 0, i - 1);  
 }  
}  
int main() {  
 int arr[] = { 3, 5, 3, 0, 8, 6, 1, 5, 8, 6, 2, 4, 9, 4, 7, 0, 1, 8, 9, 7, 3, 1, 2, 5, 9, 7, 4, 0, 2, 6 };  
 int len = (int) sizeof(arr) / sizeof(\*arr);  
 heap\_sort(arr, len);  
 for (int i = 0; i < len; i++)  
 cout << arr[i] << ' ';  
 cout << endl;  
 return 0;  
}

InsertionSearch.h

//插值查找  
int InsertionSearch(int a[], int value, int low, int high)  
{  
 int mid = low+(value-a[low])/(a[high]-a[low])\*(high-low);  
 if(a[mid]==value)  
 return mid;  
 if(a[mid]>value)  
 return InsertionSearch(a, value, low, mid-1);  
 if(a[mid]<value)  
 return InsertionSearch(a, value, mid+1, high);  
}

InsertSort.h

/\*  
（有序区，无序区）。把无序区的第一个元素插入到有序区的合适的位置。对数组：比较得少，换得多。  
  
插入排序思路：  
1. 从第一个元素开始，该元素可以认为已经被排序  
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描  
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置  
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置  
5. 将新元素插入到该位置后  
6. 重复步骤2~5  
  
\*/  
// 插入排序  
void InsertSort(vector<int>& v)  
{  
 int len = v.size();  
 for (int i = 1; i < len - 1; ++i) {  
 int temp = v[i];  
 for(int j = i - 1; j >= 0; --j)  
 {  
 if(v[j] > temp)  
 {  
 v[j + 1] = v[j];  
 v[j] = temp;  
 }  
 else  
 break;  
 }  
 }  
}

MergeSort.h

// 归并排序：把数据分为两段，从两段中逐个选最小的元素移入新数据段的末尾。可从上到下或从下到上进行。  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 迭代版  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
//整數或浮點數皆可使用,若要使用物件(class)時必須設定"小於"(<)的運算子功能  
template<typename T>  
void merge\_sort(T arr[], int len) {  
 T\* a = arr;  
 T\* b = new T[len];  
 for (int seg = 1; seg < len; seg += seg) {  
 for (int start = 0; start < len; start += seg + seg) {  
 int low = start, mid = min(start + seg, len), high = min(start + seg + seg, len);  
 int k = low;  
 int start1 = low, end1 = mid;  
 int start2 = mid, end2 = high;  
 while (start1 < end1 && start2 < end2)  
 b[k++] = a[start1] < a[start2] ? a[start1++] : a[start2++];  
 while (start1 < end1)  
 b[k++] = a[start1++];  
 while (start2 < end2)  
 b[k++] = a[start2++];  
 }  
 T\* temp = a;  
 a = b;  
 b = temp;  
 }  
 if (a != arr) {  
 for (int i = 0; i < len; i++)  
 b[i] = a[i];  
 b = a;  
 }  
 delete[] b;  
}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 递归版  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
template<typename T>  
void merge\_sort\_recursive(T arr[], T reg[], int start, int end) {  
 if (start >= end)  
 return;  
 int len = end - start, mid = (len >> 1) + start;  
 int start1 = start, end1 = mid;  
 int start2 = mid + 1, end2 = end;  
 merge\_sort\_recursive(arr, reg, start1, end1);  
 merge\_sort\_recursive(arr, reg, start2, end2);  
 int k = start;  
 while (start1 <= end1 && start2 <= end2)  
 reg[k++] = arr[start1] < arr[start2] ? arr[start1++] : arr[start2++];  
 while (start1 <= end1)  
 reg[k++] = arr[start1++];  
 while (start2 <= end2)  
 reg[k++] = arr[start2++];  
 for (k = start; k <= end; k++)  
 arr[k] = reg[k];  
}  
//整數或浮點數皆可使用,若要使用物件(class)時必須設定"小於"(<)的運算子功能  
template<typename T>   
void merge\_sort(T arr[], const int len) {  
 T \*reg = new T[len];  
 merge\_sort\_recursive(arr, reg, 0, len - 1);  
 delete[] reg;  
}

SequentialSearch.h

// 顺序查找  
int SequentialSearch(vector<int>& v, int k) {  
 for (int i = 0; i < v.size(); ++i)  
 if (v[i] == k)  
 return i;  
 return -1;  
}

QuickSort.h

/\*  
  
（小数，基准元素，大数）。在区间中随机挑选一个元素作基准，将小于基准的元素放在基准之前，大于基准的元素放在基准之后，再分别对小数区与大数区进行排序。  
  
快速排序思路：  
1. 选取第一个数为基准  
2. 将比基准小的数交换到前面，比基准大的数交换到后面  
3. 对左右区间重复第二步，直到各区间只有一个数  
  
\*/  
  
// ----------------------------------------------------  
  
// 快速排序（递归）  
void QuickSort(vector<int>& v, int low, int high) {  
 if (low >= high) // 结束标志  
 return;  
 int first = low; // 低位下标  
 int last = high; // 高位下标  
 int key = v[first]; // 设第一个为基准  
  
 while (first < last)  
 {  
 // 将比第一个小的移到前面  
 while (first < last && v[last] >= key)  
 last--;  
 if (first < last)  
 v[first++] = v[last];  
  
 // 将比第一个大的移到后面  
 while (first < last && v[first] <= key)  
 first++;  
 if (first < last)  
 v[last--] = v[first];  
 }  
 // 基准置位  
 v[first] = key;  
 // 前半递归  
 QuickSort(v, low, first - 1);  
 // 后半递归  
 QuickSort(v, first + 1, high);  
}

// ----------------------------------------------------  
// 模板实现快速排序（递归）  
template <typename T>  
void quick\_sort\_recursive(T arr[], int start, int end) {  
 if (start >= end)  
 return;  
 T mid = arr[end];  
 int left = start, right = end - 1;  
 while (left < right) {  
 while (arr[left] < mid && left < right)  
 left++;  
 while (arr[right] >= mid && left < right)  
 right--;  
 std::swap(arr[left], arr[right]);  
 }  
 if (arr[left] >= arr[end])  
 std::swap(arr[left], arr[end]);  
 else  
 left++;  
 quick\_sort\_recursive(arr, start, left - 1);  
 quick\_sort\_recursive(arr, left + 1, end);  
}  
template <typename T> //整數或浮點數皆可使用,若要使用物件(class)時必須設定"小於"(<)、"大於"(>)、"不小於"(>=)的運算子功能  
void quick\_sort(T arr[], int len) {  
 quick\_sort\_recursive(arr, 0, len - 1);  
}

// ----------------------------------------------------  
  
// 模板实现快速排序（迭代）  
struct Range {  
 int start, end;  
 Range(int s = 0, int e = 0) {  
 start = s, end = e;  
 }  
};  
template <typename T> // 整數或浮點數皆可使用,若要使用物件(class)時必須設定"小於"(<)、"大於"(>)、"不小於"(>=)的運算子功能  
void quick\_sort(T arr[], const int len) {  
 if (len <= 0)  
 return; // 避免len等於負值時宣告堆疊陣列當機  
 // r[]模擬堆疊,p為數量,r[p++]為push,r[--p]為pop且取得元素  
 Range r[len];  
 int p = 0;  
 r[p++] = Range(0, len - 1);  
 while (p) {  
 Range range = r[--p];  
 if (range.start >= range.end)  
 continue;  
 T mid = arr[range.end];  
 int left = range.start, right = range.end - 1;  
 while (left < right) {  
 while (arr[left] < mid && left < right) left++;  
 while (arr[right] >= mid && left < right) right--;  
 std::swap(arr[left], arr[right]);  
 }  
 if (arr[left] >= arr[range.end])  
 std::swap(arr[left], arr[range.end]);  
 else  
 left++;  
 r[p++] = Range(range.start, left - 1);  
 r[p++] = Range(left + 1, range.end);  
 }  
}

RadixSort.h

// 基数排序：一种多关键字的排序算法，可用桶排序实现。  
  
int maxbit(int data[], int n) //辅助函数，求数据的最大位数  
{  
 int maxData = data[0]; ///< 最大数  
 /// 先求出最大数，再求其位数，这样有原先依次每个数判断其位数，稍微优化点。  
 for (int i = 1; i < n; ++i)  
 {  
 if (maxData < data[i])  
 maxData = data[i];  
 }  
 int d = 1;  
 int p = 10;  
 while (maxData >= p)  
 {  
 //p \*= 10; // Maybe overflow  
 maxData /= 10;  
 ++d;  
 }  
 return d;  
/\* int d = 1; //保存最大的位数  
 int p = 10;  
 for(int i = 0; i < n; ++i)  
 {  
 while(data[i] >= p)  
 {  
 p \*= 10;  
 ++d;  
 }  
 }  
 return d;\*/  
}

void radixsort(int data[], int n) //基数排序  
{  
 int d = maxbit(data, n);  
 int \*tmp = new int[n];  
 int \*count = new int[10]; //计数器  
 int i, j, k;  
 int radix = 1;  
 for(i = 1; i <= d; i++) //进行d次排序  
 {  
 for(j = 0; j < 10; j++)  
 count[j] = 0; //每次分配前清空计数器  
 for(j = 0; j < n; j++)  
 {  
 k = (data[j] / radix) % 10; //统计每个桶中的记录数  
 count[k]++;  
 }  
 for(j = 1; j < 10; j++)  
 count[j] = count[j - 1] + count[j]; //将tmp中的位置依次分配给每个桶  
 for(j = n - 1; j >= 0; j--) //将所有桶中记录依次收集到tmp中  
 {  
 k = (data[j] / radix) % 10;  
 tmp[count[k] - 1] = data[j];  
 count[k]--;  
 }  
 for(j = 0; j < n; j++) //将临时数组的内容复制到data中  
 data[j] = tmp[j];  
 radix = radix \* 10;  
 }  
 delete []tmp;  
 delete []count;  
}

SelectionSort.h

/\*  
  
（有序区，无序区）。在无序区里找一个最小的元素跟在有序区的后面。对数组：比较得多，换得少。  
  
选择排序思路：  
1. 在未排序序列中找到最小（大）元素，存放到排序序列的起始位置  
2. 从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾  
3. 以此类推，直到所有元素均排序完毕  
  
\*/  
// 选择排序  
void SelectionSort(vector<int>& v) {  
 int min, len = v.size();  
 for (int i = 0; i < len - 1; ++i) {  
 min = i;  
 for (int j = i + 1; j < len; ++j) {  
 if (v[j] < v[min]) { // 标记最小的  
 min = j;  
 }  
 }  
 if (i != min) // 交换到前面  
 swap(v[i], v[min]);  
 }  
}  
  
// 模板实现  
template<typename T>   
void Selection\_Sort(std::vector<T>& arr) {  
 int len = arr.size();  
 for (int i = 0; i < len - 1; i++) {  
 int min = i;  
 for (int j = i + 1; j < len; j++)  
 if (arr[j] < arr[min])  
 min = j;  
 if(i != min)  
 std::swap(arr[i], arr[min]);  
 }  
}

ShellSort.h

// 希尔排序：每一轮按照事先决定的间隔进行插入排序，间隔会依次缩小，最后一次一定要是1。  
template<typename T>  
void shell\_sort(T array[], int length) {  
 int h = 1;  
 while (h < length / 3) {  
 h = 3 \* h + 1;  
 }  
 while (h >= 1) {  
 for (int i = h; i < length; i++) {  
 for (int j = i; j >= h && array[j] < array[j - h]; j -= h){  
 std::swap(array[j], array[j - h]);  
 }  
 }  
 h = h / 3;  
 }  
}