|  |  |  |
| --- | --- | --- |
| 软件学院实验报告 | | |
| 姓名： 韩泽旭 学号： 2410250044 专业： 软件工程 年级：2024级 | | |
| 课程名称 | | 数据结构 |
| 实验名称 | | 实验1 集合的交、并、差 |
| 实验的准备阶段 | 实验内容 | （1）实验目的  通过该实验，让学生复习巩固C语言中的分支结构、循环结构、数组、链表、函数的调用等有关内容，体会到用数组存储集合时，需要记录集合元素的个数，否则输出结果会出现数据越界现象。  （2）实验内容  通过键盘，分别输入集合A和B中的数据元素，要求数据元素类型为整数类型，输出两个集合的交、并、差、是否是子集。  （3）实验要求  a)从程序完善性上考虑，集合元素输入时，要有检查元素重复的功能，每个集合中不允许有重复的元素。集合可以用数组存储（可以用固定数组，也可以用动态数组）；也可以用链表存储；或者用结构体定义也可以。  b)实现交、并、差、判断A是否是B的子集运算时，分别把代码写成函数的形式，即实现交运算的函数，实现并运算的函数，实现差运算的函数、实现是否是子集函数，在主函数中分别调用4个函数。  c)使用菜单形式对应各个操作，应允许用户反复查看结果，想结束程序时，输入负数结束，使其编成一个完整的小软件。菜单参考示例如下：  1---输入集合A和B  2---求集合A交B  3---求集合A并B  4---求集合A-B  5---判断A是否是B的子集  退出，输入一个负数！  注意：菜单建议不要做成清屏和刷屏的效果，否则后面截图会比较麻烦。  （4）验收/测试用例  如：  输入： A={4,2,1,3,5} B={6,3,5,7,4,8}  要注意输入的过程中，每输入一个元素都要检查输入的这个元素是否和前面的元素重复，如果重复，要求用户重新输入当前元素。验收测试时要测试这种重复的情况。  输出：A交B={3， 4， 5} A并B={1，2，3，4，5，6，7,8}  A-B={1, 2} A不是B的子集  注意：每个同学测试截图的时候，数据要修改一下，不要全用上面的数据。 |
| 实验类型 | 验证性 |
| 实验的重点、难点 | 重点： 数组或线性表的使用  难点： 去重操作和函数调用 |
| 实验环境 | VC++6.0 或 DEV |
| 实验的实施阶段 | 实验步骤及完成任务情况 | 1. 设计思想   大致结构  总体上程序按照这个结构安排文件：  由main.cpp加载所有的内容（最顶层菜单）  包含以下四组内容：  {  cFixedArray：C语言风格固定数组  cDynamicArray：C语言风格动态数组  cDyLinkedList：C语言风格动态链表  cppSet：C++风格Set集合  }  为了复制粘贴函数较为方便，四组内容各自用namespace包裹，打包在prototype.h文件里  每一组内容里面都有自己的main函数，用于调用小菜单，决定进行什么操作  C语言风格内容中包含了一个cSafeInput头文件，用于对int和short输入进行预处理，C++风格内容也引入了cppSafeInput,使用模板控制不同类型的输入  为了测试方便，在constData头文件中提供四个数组便于直接调用代替键盘输入  去重算法  去重首先要排序，ai提供了C语言中使用的排序函数：qsort,排序虽然写了冒泡和选择两种，但是默认选择ai提供的qsort  去重时使用两个索引操作同一个数组，如果前后两个元素不一样，就正常更新索引，如果元素一样，就其中一个索引先向前找到不同的元素，然后一点点挪，一个一个覆盖，直到最前面的索引超出边界  至于链表，输入的时候就去重了，先利用previous指针挂在head上，然后一个一个找哪个和自己数据一样大，一样大的就删除，没有一样大的就把数据挂在previous的下一个上  C++的集合会自动去重  交集  我自己写的是创建一个大小是max（arr1,arr2）的数组，  然后双重循环，一对多挨个比对，中途统计一共有多少位，最后输出  ai写的更快更好的方法是创建两个索引，分别指向两个数组，大小不一样的话小的就往前挪一位，相等就记录，直至结束  Set的话有给定的函数，不用自己写，以下相同  并集  双索引，对应元素较小就添加并向前挪一个，一样的就添加并挪一个，天然过滤掉了添加两次的问题，并且在循环结束了以后把两个数组/链表剩下的内容全部加入新数组/链表  差集  双索引，不一样的就录入，一样的跳过，循环结束了以后把剩下的所有元素全都录入  判断A是否是B的子集  双索引，如果这个集合A首先出现了更小的数，说明不是子集，如果没有更小的，就向后正常寻找，发现元素一样的就正常向后挪一位，最后如果是子集，那就已经挪出去了，变成NULL了，但是如果有额外的元素，就会一直挪不干净，检测到返回false   1. 主要源代码  |  | | --- | | main.cpp  #include "cSafeInput.h"  #include "prototypes.h"  int main(int argc, char \*argv[])  {  re\_full\_menu:;  printf("1.C fixed array.\n2.C dynamic array.\n"  "3.C dynamic linked list.\n4.C++.\n");  switch(hd\_input())  {  case 1: cFixedArray::main\_CFA();break;  case 2: cDynamicArray::main\_CDA();break;  case 3: cDyLinkedList::main\_CDLL();break;  case 4: cppSet::main\_CppS();break;  default: goto full\_exit;  }  goto re\_full\_menu;  full\_exit:;  return 0;  } | | prototypes.h  //  // Created by comardom on 25-9-13.  //  #ifndef PROTOTYPES\_H  #define PROTOTYPES\_H  namespace cFixedArray  {  int main\_CFA();  }  namespace cDynamicArray  {  int main\_CDA();  }  namespace cDyLinkedList  {  int main\_CDLL();  }  namespace cppSet  {  int main\_CppS();  }  #endif //PROTOTYPES\_H | | connstData.h  //  // Created by comardom on 25-9-11.  //  #ifndef CONSTDATA\_H  #define CONSTDATA\_H  #define ARR\_SIZE 10  const int TEST\_INT\_ARR\_A[ARR\_SIZE] = {2, 5, -3, 0, 7, 5, 10, -8, 1, 4};  // 包含负数、零，并与A有交集  const int TEST\_INT\_ARR\_B[ARR\_SIZE] = {-8, 4, 1, 9, 0, 2, -6, 15, 12, 11};  const int TEST\_INT\_SON[ARR\_SIZE/2] = {1, 2, 3, 4, 5};  const int TEST\_INT\_FATHER[ARR\_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  #endif //CONSTDATA\_H | | cSafeInput.h  //  // Created by comardom on 25-9-11.  //  #ifndef CSAFEINPUT\_H  #define CSAFEINPUT\_H  #include <limits.h>  #include <stdio.h>  inline void clear\_stdin()  {//用于清空缓冲区  int ch;  // 循环读取输入流中的所有字符，直到换行符或 EOF  while ((ch = getchar()) != '\n' && ch != EOF)  {  ;// 什么都不做，仅用于清空缓冲区  }  }  //简单说一下哈，就是所有的输入都用我自己写的函数代替了，里面包含了输入检查  //原生scanf容易把自己卡死，getchar说实话太累了  inline void d\_input(int \*n)  {  int tmp=0;  bool is\_num = true;  goto\_d:;  is\_num=scanf("%d",&tmp);  if(!is\_num || tmp>INT\_MAX || tmp<INT\_MIN)  {  printf("Incorrect input! Plz enter again.\n");  clear\_stdin();  goto goto\_d;  }  \*n=tmp;  }  inline void hd\_input(short \*n)  {  short tmp=0;  bool is\_num = true;  goto\_hd:;  is\_num=scanf("%hd",&tmp);  if(!is\_num || tmp>SHRT\_MAX || tmp<SHRT\_MIN)  {  printf("Incorrect input! Plz enter again.\n");  clear\_stdin();  goto goto\_hd;  }  \*n=tmp;  }  inline short hd\_input()  {  short tmp=0;  bool is\_num = true;  goto\_hd:;  is\_num=scanf("%hd",&tmp);  if(!is\_num || tmp>SHRT\_MAX || tmp<SHRT\_MIN)  {  printf("Incorrect input! Plz enter again.\n");  clear\_stdin();  goto goto\_hd;  }  return tmp;  }  #endif //CSAFEINPUT\_H | | cppSafeInput.h  //  // Created by Comardom on 25-9-6.  //  #ifndef NUM\_UND\_STR\_SAFE\_INPUT\_H  #define NUM\_UND\_STR\_SAFE\_INPUT\_H  #include <iostream>  #include <limits>  #include <string>  // 清空输入缓冲区  void clear\_stdin();  // 数字输入函数模板  template <typename T>  void num\_input\_s(T& n);  // 字符串输入函数  void string\_input\_s(std::string& s);  // --- 以下是函数的完整实现 ---  inline void clear\_stdin()  {  std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  }  template <typename T>  void num\_input\_s(T &n)  {  goto\_re\_input:;  std::cin.clear();  std::cin >> n;  if (std::cin.good() && n >= std::numeric\_limits<T>::min() && n <= std::numeric\_limits<T>::max())  {  clear\_stdin();  }  else  {  std::cout << "Error. Plz enter again." << std::endl;  std::cin.clear();  clear\_stdin();  goto goto\_re\_input;  }  }  inline short hd\_input\_s()  {  goto\_re\_input:;  std::cin.clear();  short tmp = 0;  std::cin >> tmp;  if (std::cin.good() && tmp >= std::numeric\_limits<short>::min() && tmp <= std::numeric\_limits<short>::max())  {  clear\_stdin();  }  else  {  std::cout << "Error. Plz enter again." << std::endl;  std::cin.clear();  clear\_stdin();  goto goto\_re\_input;  }  return tmp;  }  inline void string\_input\_s(std::string &s)  {  std::cin.clear();  clear\_stdin();  std::getline(std::cin, s);  }  #endif // NUM\_UND\_STR\_SAFE\_INPUT\_H | | cFixedArray.cpp  //  // Created by comardom on 25-9-11.  //  //通过键盘，分别输入集合A和B中的数据元素，要求数据元素类型为整数类型，输出两个集合的交、并、差、是否是子集。  #include <stdlib.h>  #include "constData.h"  #include "cSafeInput.h"  namespace cFixedArray  {  int main\_CFA()  {  void init\_array(int \*arr, int name, bool use\_const\_data);  void remove\_duplicates(int \*arr, int \*n);  void print\_array(const int \*arr);  void bubble\_sort(int \*arr, int n);  void selection\_sort(int \*arr, int n);  void print\_arrOut(int \*arrOut, int digits, short purpose);  void intersection\_slow(const int \*arrA,const int \*arrB);  void intersection\_fast(const int \*arrA, int sizeA, const int \*arrB, int sizeB);  void union\_func(const int \*arrA, int sizeA, const int \*arrB, int sizeB);  void choose\_sort(int \*arrA, int \*arrB, int sizeA, int sizeB, short howSort);  void difference(const int \*arrA, int sizeA, const int \*arrB, int sizeB, bool isA\_minus\_B);  bool is\_subset(const int \*arrA, int sizeA, const int \*arrB, int sizeB);  void init\_array2(int \*arr, int name);  //这是普通const  // int arrA[ARR\_SIZE], arrB[ARR\_SIZE];  // int sizeA = ARR\_SIZE, sizeB = ARR\_SIZE;  // init\_array(arrA, 0,true);  // init\_array(arrB, 1,true);  //这是子集const  int arrA[ARR\_SIZE/2], arrB[ARR\_SIZE];  int sizeA = ARR\_SIZE/2, sizeB = ARR\_SIZE;  init\_array2(arrA,0);  init\_array2(arrB,1);  choose\_sort(arrA, arrB,sizeA,sizeB, 2);  remove\_duplicates(arrA, &sizeA);  remove\_duplicates(arrB, &sizeB);  re\_menu:;  printf("Fixed array.\n1. intersection\_slow\n"  "2. intersection\_fast\n""3. union\n4.difference\n5.if A is sub of B\n6.if B is sub of A\n");  switch(hd\_input())  {  case 1:  intersection\_slow(arrA, arrB);break;  case 2:  intersection\_fast(arrA,sizeA,arrB,sizeB);break;  case 3:  union\_func(arrA, sizeA, arrB, sizeB);break;  case 4:  difference(arrA, sizeA, arrB, sizeB, true);  case 5:  printf(is\_subset(arrA, sizeA, arrB, sizeB)?"True\n":"False\n");break;  case 6:  printf(is\_subset(arrB, sizeB, arrA, sizeA)?"True\n":"False\n");break;  default: goto exit\_menu;  }  goto re\_menu;  exit\_menu:;  return 0;  }  // 负责初始化数组，只进行输入，不去重  void init\_array(int \*arr, int name, bool use\_const\_data)  {  const char\* setName = (name == 0) ? "A" : "B";  if (use\_const\_data)  {  const int\* constArr = (name == 0) ? TEST\_INT\_ARR\_A : TEST\_INT\_ARR\_B;  for (int i = 0; i < ARR\_SIZE; i++)  {  arr[i] = constArr[i];  }  printf("Static %s.\n", setName);  }  else  {  printf("In %d nums to %s.\n", ARR\_SIZE, setName);  for (int i = 0; i < ARR\_SIZE; i++)  {  d\_input(arr + i);  }  }  }  void init\_array2(int \*arr, int name)  {  const char\* setName = (name == 0) ? "SON" : "FATHER";  const int\* constArr = (name == 0) ? TEST\_INT\_SON: TEST\_INT\_FATHER;  if (name == 0)  {  for (int i = 0; i < ARR\_SIZE/2; i++)  {  arr[i] = constArr[i];  }  }  else  {  for (int i = 0; i < ARR\_SIZE; i++)  {  arr[i] = constArr[i];  }  }  printf("Static %s.\n", setName);  }  // 这是一个新的辅助函数，用于在排序后去重  void remove\_duplicates(int \*arr, int \*n)  {  if(\*n <= 1){return;}  int j = 0;  for(int i = 1; i < \*n; i++)  {  if(arr[i] != arr[j])  {  j++;  arr[j] = arr[i];  }  }  \*n = j + 1; // 更新数组的有效大小  }  void print\_array(const int \*arr)  {  for(int i=0;i<ARR\_SIZE;i++)  {  if(i==0)  {  printf("This: \n");  }  printf("%d ",\*(arr+i));  if(i==ARR\_SIZE-1)  {  printf("\b\n");  }  }  }  void bubble\_sort(int \*arr, const int n)  {  int tmp=0;  for (int i=0;i<n-1;i++)  {  for (int j=0;j<n-i-1;j++)  {  if (\*(arr+j)>\*(arr+j+1))  {  tmp=\*(arr+j);  \*(arr+j)=\*(arr+j+1);  \*(arr+j+1)=tmp;  }  }  }  }  void selection\_sort(int \*arr, int n)  {//选择排序真的很复杂  int temp = 0;  for (int i = 0; i < n - 1; i++)  {  // 找到未排序部分的最小元素的索引  int min\_idx = i;  for (int j = i + 1; j < n; j++)  {  if (arr[j] < arr[min\_idx])  {  min\_idx = j;  }  }  // 将找到的最小元素与当前位置的元素交换  temp = arr[min\_idx];  arr[min\_idx] = arr[i];  arr[i] = temp;  }  }  int compare\_ints(const void \*a, const void \*b)  {  // ReSharper disable once CppCStyleCast  const int arg1 = \*(const int\*)a;  // ReSharper disable once CppCStyleCast  const int arg2 = \*(const int\*)b;  return (arg1 - arg2);  }  //上下两个函数绑定，用于qsort,不如模板  void stdlib\_q\_int\_sort(void \*base, size\_t num, size\_t size)  {  qsort(base, num, size, compare\_ints);  }  void print\_arrOut(const int \*arrOut, const int digits, const short purpose)  {  if(digits)  {  switch(purpose)  {  case 0:printf("The intersection is: \n{ ");break;  case 1:printf("The union is: \n{ ");break;  case 2:printf("The difference is: \n{ ");  default:;  }  for(int i=0;i<digits;i++)  {  printf("%d, ",\*(arrOut+i));  }  printf("\b\b }\n");  }  else  {  printf("∅\n");  }  }  void intersection\_slow(const int \*arrA,const int \*arrB)  {  int arrOut[ARR\_SIZE]={0};  short digits = 0;  for(int i=0;i<ARR\_SIZE;i++)  {  for(int j=0;j<ARR\_SIZE;j++)  {  if(\*(arrA+i)==\*(arrB+j))  {  arrOut[digits]=\*(arrA+i);  digits++;  }  }  }  print\_arrOut(arrOut,digits,0);  }  void choose\_sort(int \*arrA, int \*arrB, int sizeA, int sizeB, short howSort)  {  switch(howSort)  {  case 0:  bubble\_sort(arrA, sizeA);  bubble\_sort(arrB, sizeB);  break;  case 1:  selection\_sort(arrA, sizeA);  selection\_sort(arrB, sizeB);  break;  case 2:  stdlib\_q\_int\_sort((void\*)arrA, sizeA, sizeof(int));  stdlib\_q\_int\_sort((void\*)arrB, sizeB, sizeof(int));  break; // Don't forget this break  default:  // It's good practice to handle invalid input  // You could print an error message or do nothing  break;  }  }  void intersection\_fast(const int \*arrA, const int sizeA, const int \*arrB, const int sizeB)  {  int arrOut[ARR\_SIZE] = {0};  // 假设 arrA 和 arrB 已经排序且去重  int i = 0, j = 0, k = 0;int digits = 0;  while (i < sizeA && j < sizeB)  {  if (arrA[i] < arrB[j])  {  i++;  }  else if(arrA[i] > arrB[j])  {  j++;  }  else  {  arrOut[k++] = arrA[i++];  j++;  }  }  print\_arrOut(arrOut,digits,0);  }  void union\_func(const int \*arrA, const int sizeA, const int \*arrB, const int sizeB)  {  int arrOut[ARR\_SIZE \* 2] = {0};  int i = 0, j = 0, digits = 0;  while (i < sizeA && j < sizeB)  {  if (arrA[i] < arrB[j])  {  arrOut[digits++] = arrA[i++];  }  else if(arrA[i] > arrB[j])  {  arrOut[digits++] = arrB[j++];  }  else  {  arrOut[digits++] = arrA[i++];  j++;  }  }  while (i < sizeA) {arrOut[digits++] = arrA[i++];}  while (j < sizeB) {arrOut[digits++] = arrB[j++];}  print\_arrOut(arrOut, digits, 1);  }  void difference(const int \*arrA, const int sizeA, const int \*arrB, const int sizeB, bool isA\_minus\_B)  {  int arrOut[ARR\_SIZE] = {0}; // 静态数组  int i = 0, j = 0, digits = 0;  // 确定基准数组和比较数组  const int \*base\_arr = isA\_minus\_B ? arrA : arrB;  const int \*compare\_arr = isA\_minus\_B ? arrB : arrA;  const int base\_size = isA\_minus\_B ? sizeA : sizeB;  const int compare\_size = isA\_minus\_B ? sizeB : sizeA;  while (i < base\_size && j < compare\_size)  {  if (base\_arr[i] < compare\_arr[j])  {  // 基准数组的当前元素是独有的，添加到差集  arrOut[digits++] = base\_arr[i++];  }  else if (base\_arr[i] > compare\_arr[j])  {  // 比较数组的当前元素不是我们需要的，跳过  j++;  }  else  {  // 找到共同元素，跳过  i++;  j++;  }  }  // 将基准数组中剩余的独有元素全部添加到差集  while (i < base\_size)  {  arrOut[digits++] = base\_arr[i++];  }  print\_arrOut(arrOut, digits, 2);  }  bool is\_subset(const int \*arrA, const int sizeA, const int \*arrB, const int sizeB)  {  //ai帮我重写的，确实没有那么绕远了  // 如果 A 的大小大于 B 的大小，那么 A 不可能是 B 的子集  if (sizeA > sizeB)  {  return false;  }  int i = 0; // arrA 的指针  int j = 0; // arrB 的指针  // 假设 arrA 和 arrB 已经排序且去重  while (i < sizeA && j < sizeB)  {  if (arrA[i] < arrB[j])  {  // 如果 A 的当前元素小于 B 的当前元素，说明 A 的这个元素在 B 中不存在。  // 因为数组已经排序，所以 arrA[i] 永远找不到匹配了。  return false;  }  else if (arrA[i] > arrB[j])  {  // B 的当前元素比 A 的小，继续在 B 中寻找下一个元素。  j++;  }  else // arrA[i] == arrB[j]  {  // 找到匹配，同时移动两个指针。  i++;  j++;  }  }  // 循环结束后，如果 i 等于 sizeA，说明 A 的所有元素都在 B 中找到了匹配。  return (i == sizeA);  }  } | | cDynamicArray.cpp  // // Created by comardom on 25-9-13. // // // Created by comardom on 25-9-11. // //通过键盘，分别输入集合A和B中的数据元素，要求数据元素类型为整数类型，输出两个集合的交、并、差、是否是子集。  #include <stdlib.h> #include "constData.h" #include "cSafeInput.h" namespace cDynamicArray {  //直接照搬的固定数组然后改的malloc  int main\_CDA()  {  void init\_array(int \*arr, int name,int size, bool use\_const\_data);  void remove\_duplicates(int \*arr, int \*n);  void print\_array(const int \*arr);  void bubble\_sort(int \*arr, int n);  void selection\_sort(int \*arr, int n);  void print\_arrOut(int \*arrOut, int digits, short purpose);  void intersection\_slow(const int\* arrA, int sizeA, const int\* arrB, int sizeB);  void intersection\_fast(const int \*arrA, int sizeA, const int \*arrB, int sizeB);  void union\_func(const int \*arrA, int sizeA, const int \*arrB, int sizeB);  void choose\_sort(int \*arrA, int \*arrB, int sizeA, int sizeB, short howSort);  void difference(const int \*arrA, int sizeA, const int \*arrB, int sizeB, bool isA\_minus\_B);  bool is\_subset(const int \*arrA, int sizeA, const int \*arrB, int sizeB);  void init\_array2(int \*arr, int name);   //这是普通const  // int arrA[ARR\_SIZE], arrB[ARR\_SIZE];  // int sizeA = ARR\_SIZE, sizeB = ARR\_SIZE;  // init\_array(arrA, 0,true);  // init\_array(arrB, 1,true);   //这是子集const  int\* arrA;int\* arrB;  int sizeA = ARR\_SIZE/2, sizeB = ARR\_SIZE;  arrA = (int\*)malloc(sizeA \* sizeof(int));  arrB = (int\*)malloc(sizeB \* sizeof(int));  init\_array2(arrA,0);  init\_array2(arrB,1);   choose\_sort(arrA, arrB,sizeA,sizeB, 2);  remove\_duplicates(arrA, &sizeA);  remove\_duplicates(arrB, &sizeB);  re\_menu:;  printf("Dynamic array.\n1. intersection\_slow\n"  "2. intersection\_fast\n""3. union\n4.difference\n5.if A is sub of B\n6.if B is sub of A\n");  switch(hd\_input())  {  case 1:  intersection\_slow(arrA,sizeA, arrB,sizeB);break;  case 2:  intersection\_fast(arrA,sizeA,arrB,sizeB);break;  case 3:  union\_func(arrA, sizeA, arrB, sizeB);break;  case 4:  difference(arrA, sizeA, arrB, sizeB, true);  case 5:  printf(is\_subset(arrA, sizeA, arrB, sizeB)?"True\n":"False\n");break;  case 6:  printf(is\_subset(arrB, sizeB, arrA, sizeA)?"True\n":"False\n");break;  default: goto exit\_menu;  }  goto re\_menu;  exit\_menu:;  free(arrA);  free(arrB);  return 0;  }   // 负责初始化数组，只进行输入，不去重  void init\_array(int \*arr, int name,int size, bool use\_const\_data)  {  const char\* setName = (name == 0) ? "A" : "B";   if (use\_const\_data)  {  const int\* constArr = (name == 0) ? TEST\_INT\_ARR\_A : TEST\_INT\_ARR\_B;  for (int i = 0; i < size; i++)  {  arr[i] = constArr[i];  }  printf("Static %s.\n", setName);  }  else  {  printf("In %d nums to %s.\n", size, setName);  for (int i = 0; i < size; i++)  {  d\_input(arr + i);  }  }  }   void init\_array2(int\* arr, int name)  {  const char\* setName = (name == 0) ? "SON" : "FATHER";  const int\* constArr = (name == 0) ? TEST\_INT\_SON: TEST\_INT\_FATHER;  if (name == 0)  {  for (int i = 0; i < ARR\_SIZE/2; i++)  {  arr[i] = constArr[i];  }  }  else  {  for (int i = 0; i < ARR\_SIZE; i++)  {  arr[i] = constArr[i];  }  }  printf("Static %s.\n", setName);  }   // 这是一个新的辅助函数，用于在排序后去重  void remove\_duplicates(int \*arr, int \*n)  {  if (\*n <= 1){return;}  int j = 0;  for (int i = 1; i < \*n; i++)  {  if (arr[i] != arr[j])  {  j++;  arr[j] = arr[i];  }  }  \*n = j + 1; // 更新数组的有效大小  }    void print\_array(const int \*arr, int size)  {  for(int i=0;i<size;i++)  {  if(i==0)  {  printf("This: \n");  }  printf("%d ",\*(arr+i));  if(i==size-1)  {  printf("\b\n");  }  }  }    void bubble\_sort(int \*arr, const int n)  {  int tmp=0;  for (int i=0;i<n-1;i++)  {  for (int j=0;j<n-i-1;j++)  {  if (\*(arr+j)>\*(arr+j+1))  {  tmp=\*(arr+j);  \*(arr+j)=\*(arr+j+1);  \*(arr+j+1)=tmp;  }  }  }  }    void selection\_sort(int \*arr, int n)  {//选择排序真的很复杂  int temp = 0;  for (int i = 0; i < n - 1; i++)  {  // 找到未排序部分的最小元素的索引  int min\_idx = i;  for (int j = i + 1; j < n; j++)  {  if (arr[j] < arr[min\_idx])  {  min\_idx = j;  }  }  // 将找到的最小元素与当前位置的元素交换  temp = arr[min\_idx];  arr[min\_idx] = arr[i];  arr[i] = temp;  }  }    int compare\_ints(const void \*a, const void \*b)  {  // ReSharper disable once CppCStyleCast  const int arg1 = \*(const int\*)a;  // ReSharper disable once CppCStyleCast  const int arg2 = \*(const int\*)b;  return (arg1 - arg2);  }  //上下两个函数绑定，用于qsort,不如模板  void stdlib\_q\_int\_sort(void \*base, size\_t num, size\_t size)  {  qsort(base, num, size, compare\_ints);  }    void print\_arrOut(const int \*arrOut, const int digits, const short purpose)  {  if(digits)  {  switch(purpose)  {  case 0:printf("The intersection is: \n{ ");break;  case 1:printf("The union is: \n{ ");break;  case 2:printf("The difference is: \n{ ");  default:;  }   for(int i=0;i<digits;i++)  {  printf("%d, ",\*(arrOut+i));  }  printf("\b\b }\n");  }  else  {  printf("∅\n");  }  }    void intersection\_slow(const int\* arrA, int sizeA, const int\* arrB, int sizeB)  {  int arrOut[sizeA>sizeB?sizeA:sizeB]={0};  short digits = 0;  for(int i=0;i<sizeA;i++)  {  for(int j=0;j<sizeB;j++)  {  if(\*(arrA+i)==\*(arrB+j))  {  arrOut[digits]=\*(arrA+i);  digits++;  }  }  }  print\_arrOut(arrOut,digits,0);  }    void choose\_sort(int \*arrA, int \*arrB, int sizeA, int sizeB, short howSort)  {  switch(howSort)  {  case 0:  bubble\_sort(arrA, sizeA);  bubble\_sort(arrB, sizeB);  break;  case 1:  selection\_sort(arrA, sizeA);  selection\_sort(arrB, sizeB);  break;  case 2:  stdlib\_q\_int\_sort((void\*)arrA, sizeA, sizeof(int));  stdlib\_q\_int\_sort((void\*)arrB, sizeB, sizeof(int));  break; // Don't forget this break  default:  // It's good practice to handle invalid input  // You could print an error message or do nothing  break;  }  }   void intersection\_fast(const int \*arrA, const int sizeA, const int \*arrB, const int sizeB)  {  int max\_size = (sizeA < sizeB) ? sizeA : sizeB;  int\* arrOut = (int\*)malloc(max\_size \* sizeof(int));  int i = 0, j = 0, k = 0;int digits = 0;  while (i < sizeA && j < sizeB)  {  if (arrA[i] < arrB[j])  {  i++;  }  else if(arrA[i] > arrB[j])  {  j++;  }  else  {  arrOut[k++] = arrA[i++];  j++;  }  }  print\_arrOut(arrOut,digits,0);  free(arrOut); // 释放内存  }    void union\_func(const int \*arrA, const int sizeA, const int \*arrB, const int sizeB)  {  int arrOut[sizeA+sizeB] = {0};  int i = 0, j = 0, digits = 0;  while (i < sizeA && j < sizeB)  {  if (arrA[i] < arrB[j])  {  arrOut[digits++] = arrA[i++];  }  else if(arrA[i] > arrB[j])  {  arrOut[digits++] = arrB[j++];  }  else  {  arrOut[digits++] = arrA[i++];  j++;  }  }  while (i < sizeA) {arrOut[digits++] = arrA[i++];}  while (j < sizeB) {arrOut[digits++] = arrB[j++];}   print\_arrOut(arrOut, digits, 1);  }     void difference(const int \*arrA, const int sizeA, const int \*arrB, const int sizeB, bool isA\_minus\_B)  {  int arrOut[sizeA>sizeB?sizeA:sizeB] = {0}; // 静态数组  int i = 0, j = 0, digits = 0;   // 确定基准数组和比较数组  const int \*base\_arr = isA\_minus\_B ? arrA : arrB;  const int \*compare\_arr = isA\_minus\_B ? arrB : arrA;  const int base\_size = isA\_minus\_B ? sizeA : sizeB;  const int compare\_size = isA\_minus\_B ? sizeB : sizeA;   while (i < base\_size && j < compare\_size)  {  if (base\_arr[i] < compare\_arr[j])  {  // 基准数组的当前元素是独有的，添加到差集  arrOut[digits++] = base\_arr[i++];  }  else if (base\_arr[i] > compare\_arr[j])  {  // 比较数组的当前元素不是我们需要的，跳过  j++;  }  else  {  // 找到共同元素，跳过  i++;  j++;  }  }   // 将基准数组中剩余的独有元素全部添加到差集  while (i < base\_size)  {  arrOut[digits++] = base\_arr[i++];  }   print\_arrOut(arrOut, digits, 2);  }    bool is\_subset(const int \*arrA, const int sizeA, const int \*arrB, const int sizeB)  {  //ai帮我重写的，确实没有那么绕远了  // 如果 A 的大小大于 B 的大小，那么 A 不可能是 B 的子集  if (sizeA > sizeB)  {  return false;  }   int i = 0; // arrA 的指针  int j = 0; // arrB 的指针   // 假设 arrA 和 arrB 已经排序且去重  while (i < sizeA && j < sizeB)  {  if (arrA[i] < arrB[j])  {  // 如果 A 的当前元素小于 B 的当前元素，说明 A 的这个元素在 B 中不存在。  // 因为数组已经排序，所以 arrA[i] 永远找不到匹配了。  return false;  }  else if (arrA[i] > arrB[j])  {  // B 的当前元素比 A 的小，继续在 B 中寻找下一个元素。  j++;  }  else // arrA[i] == arrB[j]  {  // 找到匹配，同时移动两个指针。  i++;  j++;  }  }   // 循环结束后，如果 i 等于 sizeA，说明 A 的所有元素都在 B 中找到了匹配。  return (i == sizeA);  } } | | cDyLinkedList.cpp  #include <stdio.h>  #include <stdlib.h>  #include <stddef.h>  #include "cSafeInput.h"  //  // Created by comardom on 25-9-13.  //  namespace cDyLinkedList  {  //这个链表是跟着ai写的  //链表内部数据与连接  typedef struct Node {  int data; // 存储数据元素  Node\* next; // 指向下一个节点的指针  };  //链表本身  typedef struct LinkedList {  Node\* head;  int size;  };  int main\_CDLL()  {  void init\_list(LinkedList\* list);  void insert\_sorted(LinkedList\* list, int value);  void print\_list(const LinkedList\* list);  void free\_list(LinkedList\* list);  void intersection\_slow(const LinkedList\* listA, const LinkedList\* listB);  void intersection\_fast(const LinkedList\* listA, const LinkedList\* listB);  void union\_func(const LinkedList\* listA, const LinkedList\* listB);  void difference(const LinkedList\* listA, const LinkedList\* listB);  bool is\_subset(const LinkedList\* listA, const LinkedList\* listB);  LinkedList listA, listB;  init\_list(&listA);  init\_list(&listB);  int countA, countB;  int value;  printf("In list A's size.\n");  d\_input(&countA);  printf("In list A.\n");  for (int i = 0; i < countA; i++)  {  printf("A[%d]: ", i);  d\_input(&value);  insert\_sorted(&listA, value);  }  printf("In list B's size.\n");  d\_input(&countB);  printf("In list B.\n");  for (int i = 0; i < countB; i++)  {  printf("B[%d]: ", i);  d\_input(&value);  insert\_sorted(&listB, value);  }  print\_list(&listA);print\_list(&listB);  re\_list\_menu:;  printf("Dynamic linked list.\n1. intersection\_slow\n"  "2. intersection\_fast\n""3. union\n4.difference\n5.if A is sub of B\n"  "6.if B is sub of A\n");  switch(hd\_input())  {  case 1:  intersection\_slow(&listA, &listB);break;  case 2:  intersection\_fast(&listA,&listB);break;  case 3:  union\_func(&listA, &listB);break;  case 4:  difference(&listA,&listB);  case 5:  printf(is\_subset(&listA,&listB)?"True\n":"False\n");break;  case 6:  printf(is\_subset(&listB,&listA)?"True\n":"False\n");break;  default: goto exit\_list\_menu;  }  goto re\_list\_menu;  exit\_list\_menu:;  free\_list(&listA);  free\_list(&listB);  return 0;  }  void init\_list(LinkedList\* list)  {  // ReSharper disable once CppZeroConstantCanBeReplacedWithNullptr  list->head = NULL;  list->size = 0;  }  void insert\_sorted(LinkedList\* list, const int value)  {  //分配一个新node,包含连接和数据  Node\* new\_node = (Node\*)malloc(sizeof(Node));  if (new\_node == NULL){printf("Memory error.\n");}  //给node注入数据  new\_node->data = value;  //注入连接  new\_node->next = NULL;  //创建一个指向头的临时指针  Node\* current = list->head;  //流转（遍历）使用的指针  Node\* previous = NULL;  //如果如果如果 内存状态正常&&输入旳值比当前找到的值大  while (current != NULL && value > current->data)  {  //向后挪一个  previous = current;  current = current->next;  }  //元素如果一样就删掉  if (current != NULL && current->data == value)  {  free(new\_node); // 释放重复节点的内存  return;  }  //head是NULL,则如果前一个是NULL,代表挂在head上  if (previous == NULL)  {  //插入到头部  //先连接后面的  new\_node->next = list->head;  //再重连  list->head = new\_node;  }  else  {  //插入到中间或尾部  //先连接新节点  new\_node->next = previous->next;  //再断开旧节点并且串起来  previous->next = new\_node;  }  //手动增加大小  list->size++;  }  Node\* get\_node\_at\_index(const LinkedList\* list, const int index)  {  if(index < 0 || index >= list->size)  {  printf("Error of index.\n");  return NULL;  }  //临时指针  Node\* current = list->head;  int count = 0;  while(current)  {  //找到第i个节点，返回其指针  if (count == index){return current;}  // 移动到下一个节点  current = current->next;  count++;  }  return NULL; // 理论上不会执行到这里，除非链表大小错误  }  void print\_list(const LinkedList\* list)  {  //如果不存在  if (!list->size)  {  printf("∅\n");  return;  }  printf("{ ");  //创建一个临时指针用于读取数据  const Node\* current = list->head;  //只要存在就执行  while(current)  {  //读取数据  printf("%d ", current->data);  //往后挪  current = current->next;  }  printf("}\n");  }  void free\_list(LinkedList\* list)  {  //临时目击者  Node\* current = list->head;  //大记忆恢复术！打不死就接着打  while(current)  {  //先把头指针杀了就读不到后面的东西了，  //所以给留一个后继  Node\* next\_node = current->next;  //弄死  free(current);  //攻守之势异也  current = next\_node;  }  //Reversio in nihilum  list->head = NULL;  list->size = 0;  }  void intersection\_slow(const LinkedList\* listA, const LinkedList\* listB)  {  int sizeA = listA->size;  int sizeB = listB->size;  LinkedList listOut;  init\_list(&listOut);  int i = 0, j = 0, k = 0;int digits = 0;  while (i < sizeA && j < sizeB)  {  if(get\_node\_at\_index(listA,i)->data < get\_node\_at\_index(listB,j)->data)  {  i++;  }  else if(get\_node\_at\_index(listA,i)->data > get\_node\_at\_index(listB,j)->data)  {  j++;  }  else  {  insert\_sorted(&listOut,get\_node\_at\_index(listA,i)->data);  i++;j++;k++;  }  }  printf("Intersection:\n");  print\_list(&listOut);  free\_list(&listOut);  }  void intersection\_fast(const LinkedList\* listA, const LinkedList\* listB)  {  const Node\* ptrA = listA->head;  const Node\* ptrB = listB->head;  LinkedList listOut;  init\_list(&listOut);  while (ptrA && ptrB)  {  if (ptrA->data < ptrB->data)  {  // A 的元素较小，移动 A 的指针  ptrA = ptrA->next;  }  else if (ptrA->data > ptrB->data)  {  // B 的元素较小，移动 B 的指针  ptrB = ptrB->next;  }  else  {  // 找到相同元素，插入到结果链表  insert\_sorted(&listOut, ptrA->data);  // 同时移动两个指针  ptrA = ptrA->next;  ptrB = ptrB->next;  }  }  printf("Intersection:\n");  print\_list(&listOut);  free\_list(&listOut);  }  void union\_func(const LinkedList\* listA, const LinkedList\* listB)  {  const Node\* ptrA = listA->head;  const Node\* ptrB = listB->head;  LinkedList listOut;  init\_list(&listOut);  while (ptrA && ptrB)  {  if (ptrA->data < ptrB->data)  {  insert\_sorted(&listOut, ptrA->data);  // A 的元素较小，移动 A 的指针  ptrA = ptrA->next;  }  else if (ptrA->data > ptrB->data)  {  insert\_sorted(&listOut, ptrB->data);  // B 的元素较小，移动 B 的指针  ptrB = ptrB->next;  }  else  {  // 找到相同元素，插入到结果链表  insert\_sorted(&listOut, ptrA->data);  // 同时移动两个指针  ptrA = ptrA->next;  ptrB = ptrB->next;  }  }  //以下为ai修复内容  // 修复点1: 处理 listA 中剩余的元素  while (ptrA)  {  insert\_sorted(&listOut, ptrA->data);  ptrA = ptrA->next;  }  // 修复点2: 处理 listB 中剩余的元素  while (ptrB)  {  insert\_sorted(&listOut, ptrB->data);  ptrB = ptrB->next;  }  printf("Union:\n");  print\_list(&listOut);  free\_list(&listOut);  }  void difference(const LinkedList\* listA, const LinkedList\* listB)  {  const Node\* ptrA = listA->head;  const Node\* ptrB = listB->head;  LinkedList listOut;  init\_list(&listOut);  while (ptrA && ptrB)  {  if (ptrA->data < ptrB->data)  {  insert\_sorted(&listOut, ptrA->data);  // A 的元素较小，移动 A 的指针  ptrA = ptrA->next;  }  else if (ptrA->data > ptrB->data)  {  // B 的元素较小，移动 B 的指针  ptrB = ptrB->next;  }  else  {  ptrA = ptrA->next;  ptrB = ptrB->next;  }  }  //ai修复  // 修复点: 处理 listA 中剩余的元素  // 这些元素肯定是 A 独有的，因为 listB 已经遍历完了  while (ptrA)  {  insert\_sorted(&listOut, ptrA->data);  ptrA = ptrA->next;  }  printf("Difference:\n");  print\_list(&listOut);  free\_list(&listOut);  }  bool is\_subset(const LinkedList\* listA, const LinkedList\* listB)  {  int sizeA = listA->size;  int sizeB = listB->size;  LinkedList listOut;  init\_list(&listOut);  // 如果 A 的大小大于 B 的大小，那么 A 不可能是 B 的子集  if (sizeA > sizeB)  {  return false;  }  const Node\* ptrA = listA->head;  const Node\* ptrB = listB->head;  while (ptrA && ptrB)  {  if (ptrA->data < ptrB->data)  {  // 如果 A 的当前元素小于 B 的当前元素，说明 A 的这个元素在 B 中不存在。  // 因为已经排序，所以 arrA[i] 永远找不到匹配了。  return false;  }  else if (ptrA->data > ptrB->data)  {  // B 的当前元素比 A 的小，继续在 B 中寻找下一个元素。  ptrB = ptrB->next;  }  else  {  ptrA = ptrA->next;  ptrB = ptrB->next;  }  }  // 如果 ptrA 已经遍历到末尾，说明 A 的所有元素都在 B 中找到了匹配  return (ptrA == NULL);  }  } | | cppSet.cpp  // // Created by comardom on 25-9-13. // #include <iostream> #include <set> #include <algorithm> #include <iterator> #include "cppSafeInput.h" namespace cppSet {  int main\_CppS()  {  void get\_user\_set(std::set<int>& my\_set, const std::string& set\_name);  void print\_set(const std::set<int>& my\_set);   std::set<int> setA, setB;  get\_user\_set(setA, "A");  get\_user\_set(setB, "B");  print\_set(setA);print\_set(setB);   re\_set\_menu:;  printf("C++ set.\n1. intersection\n"  "2. union\n3.difference\n4.if A is sub of B\n5.if B is sub of A\n");  switch(hd\_input\_s())  {  case 1:  {  std::set<int> intersection;  set\_intersection(setA.begin(), setA.end(), setB.begin(), setB.end(),  inserter(intersection, intersection.begin()));  print\_set(intersection);break;  }  case 2:  {  std::set<int> union\_set;  set\_union(setA.begin(), setA.end(), setB.begin(), setB.end(),  inserter(union\_set, union\_set.begin()));  print\_set(union\_set);break;  }  case 3:  {  std::set<int> difference;  set\_difference(setA.begin(), setA.end(), setB.begin(), setB.end(),  inserter(difference, difference.begin()));  print\_set(difference);break;  }  case 4:  {  std::cout <<includes(setB.begin(), setB.end(),  setA.begin(), setA.end()) << std::endl;break;  }  case 5:  {  std::cout <<includes(setA.begin(), setA.end(),  setB.begin(), setB.end()) << std::endl;break;  }  default:{ goto exit\_set\_menu;}  }  goto re\_set\_menu;  exit\_set\_menu:;  return 0;  }   void get\_user\_set(std::set<int>& my\_set, const std::string& set\_name)  {  int count;  std::cout << "Set in " << set\_name << "'s size";  num\_input\_s(count);   std::cout << "Set in " << count << " nums for " << set\_name << std::endl;  int value;  for (int i = 0; i < count; ++i)  {  std::cout << set\_name << "[" << i << "]: ";  num\_input\_s(value);  my\_set.insert(value);  }  }  void print\_set(const std::set<int>& my\_set)  {  std::cout << "{ ";  for (const int elem : my\_set)  {  std::cout << elem << " ";  }  std::cout << "}" << std::endl;  } } | |
| 实验结果的处理阶段 | 实验结果 |  |
| 实验结果总结 | 第一个：不同文件相同名称可能打架，所以使用namespace隔离  第二个：发现namespace隔离导致编译出错，则把头文件提出去，并且使用std::的形式而不是using namespace std；  第三个：输入难以控制，那么就引入头文件来控制输入，检查大小和合法性 |