**山东大学计算机科学与技术学院  
数据结构与算法课程设计报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201900180091 | 姓名：张朝阳 | | 班级： 19.1 |
| 上机学时： 6 | | 日期：2021.06.01 | |
| 课程设计题目： 二项堆的实现与分析 | | | |
| 软件环境：Visual Studio 2019  Qt Creator4.9 | | | |
| 报告内容：  1.需求描述  **1.1 问题描述**  认识二项树、二项堆数据结构，并能应用该结构解决实际问题。    **1.2 基本要求**  ①设计二项堆ADT，其上的基本操作包括：  Make Heap ():初始化一个空堆；  Find-Min():返回一个指向最小关键字元素的指针  Union(H):与堆H进行合并，合并后的结果保存在当前堆，H变为空  Insert(x):插入元素x  Extract-Min():从堆中删除最小关键字元素，并返回指向删除元素的指针。  Decrease Key (x,k):将元素x的关键字赋予新值k;  Delete(x):从堆中删除元素x;  ②实现二项堆ADT，包括实现二项堆的存储结构以及其上的基本操作，并分析基本操作的时间复杂性。  ③实现二项堆ADT的基本操作展示。  **1.3 输入说明**  输入界面设计：  输入界面设计了UI界面输入（为了演示操作） 以及美化的CMD界面输入（功能性更全）。  UI输入：设计了UI输入输出界面，可以输入插入删除更新的值，以及pop等操作，方便操作演示。  输入样例：    可以输入（插入）一定的数据以及操作，并且可视化展示二项堆结构：    点击合并，生成第三个堆，来可视化合并操作：    CMD输入：    如果选择1：对拍完成后会有一个效率对比的选项    如果选择2：需要输入路径    如果选择3：会有操作提示，并且可以dot作图可视化：      输入异常的处理：   1. 拒绝删除、更新不存在数据，拒绝不符合Extract语义的操作（新值要小/大）      1. 拒绝不存在的操作：如上图   **1.4 输出说明**  输出界面设计：  输出样例：  UI界面输出：展示操作合并：  Qt-3  对于1堆删除最小值，更新5->1后：    CMD窗口输出：   1. 测试数据   CMD-2   1. 对拍   CMD-3  3.与HBLT对比数据：并且测试性能，生成对比图展示：  CMD-4CMD-6  4.输入路径的输出：    2.分析与设计  **2.1 问题分析**  二项堆的实现与分析所执行的任务包括：  显示功能界面------------------------------------欢迎模块  按要求进行选择性输入数据、生成数据、性能对比----输入模块  二项堆的基本操作----------------------------基本功能模块  Dot 做二项堆的图------------------------------可视化模块  与普通队列进行对拍------------------------------对拍模块  数据生成器----------------------------------数据生成模块  与HBLT对比性能-----------------------------性能分析模块  输出操作结果，各种遍历方式----------------------输出模块  -欢迎模块：  void welcome()；  MainWindow::MainWindow(QWidget \*parent)  ·inital()在屏幕显示文字表示的欢迎界面，并且提供操作提示；  ·在屏幕显示UI欢迎界面，并提供输入控件。  -输入模块：  Void datai(）；(i从1-4)  void MainWindow::insert()//插入一个数据  void MainWindow::delete()//删除一个数据  void MainWindow::update()//更新一个数据  ·data1()输入一百个数据集并且进行对拍  ·data2()输入指定路径下的文件  ·data3()手动输入数据，可以用来测试最大堆  ·data4()利用数据生成器创建一个新的数据集合，并且运行，输出。  -基本功能模块：  void Make\_Heap(Compare c = Compare())  BinaryHeapNode<K,V>\* Find\_Min();  void Union(Bheap& Heap);  void insert(pair<K,V> x);  pair<K, V>\* Extract\_Min();  void Decrease\_Key(K oldkey, K newk);  void Delete(K KEY);  ·分别实现初始化空堆（指定比较方式） 寻找最小值，合并，插入，删除最小值，更新旧值，删除。  -可视化模块:  void visualize(string filename);//遍历该二项堆，生成二项堆结构图片。  -对拍模块:  使用FC命令进行对拍  -数据生成模块:  Makedata1(ofstream& fout, int total\_ops, int total\_nums)  makedata2(ofstream& fout, int total\_ops, int total\_nums)  生成不同操作的数据集，1是每一种操作都可以输出，2是只有简单操作，例如push pop top  -性能分析模块:  void comparetime();//同样样例，HBLT与BinaryHeap时间做对比  Testtime.py 作图  -输出模块：  void Post(bnode\* x);//后序遍历  void preTravel(bnode\* x);//前序遍历  void preT();//前序遍历  void levelTravel();//层次遍历  //不同的遍历方式输出二项堆的信息  void outputPost() { Post(Root); }  void outputpre() { preT(); };  **2.2 主程序设计**    Welcome 进行显示欢迎界面(颜色美化) 并且展示输入提示    QT下：展示UI界面 并检查槽      **2.3 设计思路**  **可视化模块：**  **Void visualize(string filename)设计思路：**  输入 输出的文件名，并且采用前序遍历，每次指定作图时的位置即可，让同一层的保证在同一层，自然就符合二项堆的结构。  **UI界面的设计思路:**  UI界面设计如图：需要设计输入数据的空间，通过使用两个堆，分别构建，最后可以合并两个堆，展示，便于演示操作。    **数据生成模块：**  **Void visualize(string filename)设计思路：**  对每个操作编号ABCDE，然后再生成对应的随机数据，即可。    **基础模块：**  **void Make\_Heap(Compare c = Compare()) 的设计思路：**  初始化一个空堆，并且可以指定比较方法，来确定最大/最小堆。  **BinaryHeapNode<K,V>\* Find\_Min()的设计思路：**  返回一个指向最小关键字元素的指针.每个二项树都是一个最小堆，所以只需要扫描一遍跟表，就能确定最值。  **BinaryHeapNode<K, V>\* Find(bnode\* t,K KEY)的设计思路：**  定位某KEY的指针，需要从根开始进行前序遍历，如果Key相等，则返回对应的指针。即如果有重复的KEY,则返回第一个找到的Key的指针。  **void Union(Bheap& Heap)的设计思路：**  合并两个堆  ①合并根表 即使用两个指针，分别扫两个堆的根表，进行排序，即类似于有序链表的合并。 序是按二项树的度由小到大排序的。  ②排好序之后，将新链表中"根节点度数相同的二项树"连接起来，直到所有根节点度数都不相同：  ③合并的具体情况：  x是根链表的当前节点，sibling\_x是x的下一个(兄弟)节点。  Case 1: x->degree != sibling\_x->degree  即，"当前节点的度数"与"下一个节点的度数"不等时。此时，不需要执行任何操作，继续查看后面的节点。  Case 2: x->degree == sibling\_x->degree == sibling\_x->sibling->degree  即，"当前节点的度数"、"下一个节点的度数"和"下下一个节点的度数"都相等时。此时，暂时不执行任何操作，还是继续查看后面的节点。实际上，这里是将"下一个节点"和"下下一个节点"等到后面再进行整合连接。  Case 3: x->degree ==sibling\_x->degree != sibling\_x->sibling->degree && x->key <= sibling\_x->key  即，"当前节点的度数"与"下一个节点的度数"相等，并且"当前节点的键值"<="下一个节点的度数"。此时，将"下一个节点(对应的二项树)"作为"当前节点(对应的二项树)的左孩子"。  Case 4: x->degree == sibling\_x->degree != sibling\_x->sibling->degree&& x->key > sibling\_x->key  即，"当前节点的度数"与"下一个节点的度数"相等，并且"当前节点的度数">"下一个节点的度数"。此时，将"当前节点(对应的二项树)"作为"下一个节点(对应的二项树)的左孩子"。  **void insert(pair<K,V> x)的设计思路：**  插入一个键值对，首先判断是否二项堆为空，空的话，直接让根节点等于这个键值对的节点，否则就以该插入节点、原二项堆的比较函数 来建一个二项堆，进行两个堆的合并即可插入。  **pair<K, V>\* Extract\_Min()的设计思路：**  从堆中删除最小关键字元素，并返回指向删除元素的指针。 即优先队列的top+pop操作。  顺着根表进行扫描，找到最值。然后删除该节点。  ①此时只有根表只有一个节点，或者根即所求，则需要修改根的指针。  ②删除最小值节点：（先逆转再合并）  (01) 将"该节点所在的二项树"进行反转。反转的意思，就是将根的所有孩子独立出来，并将这些孩子整合成二项堆，将该二项堆记为child。  (02) 将child和heap进行合并操作。  最后然会最小值节点的键值对指针即可  **void Decrease\_Key(K oldkey, K newk)设计思路：**  将元素X的关键字赋予新值K。（其实优先队列里面没有这种操作）  ①先找到oldkey对应的节点。  ②如果没找到，则输出not exist can't decrease! 如果找到了，则更新新值。  ③然后进行堆的调整，从该节点开始向上走，如果值小，则交换上去，只需要交换值即可不需要交换节点。  **void Delete(K KEY)设计思路：**  从堆中删除元素X（K = key）  首先需要找该节点，如果找不到则输出一个not exist! can't delete  找到了，则将该节点进行Decrease\_Key()把当前节点的值，赋新值INT\_MIN,然后自然会调整到根表，然后extract\_min，就删除了该节点(必为最小值节点)。  **欢迎模块：**  **Void welcome()的设计思路；**  提示可输入的数据集以及对应的操作，以及输入的内容即可。 使用SetConsoleTextAttribute()设置颜色，更美观。  **MainWindow::MainWindow(QWidget \*parent) 的设计思路：**  提示以及输入数据的个数，界面简洁清晰  **输入模块：**  **void data1()设计思路：**  自动输入数据生成器生成的一百个数据，每个数据集首先输入一个total\_nums代表初始化二项堆的数据个数，然后一个total\_opnums，即要进行多少次操作，之后进行输入total\_nums个数据，初始化。之后输入total\_opnums个操作，进行操作。  然后进行输出，之后和标准优先队列的输出进行对拍，看是否正确。  **void data2()设计思路：**  输入指定路径的数据集。输入格式同data1。之后输出再CMD  **void data3()设计思路：**  手动输入数据，格式也是同data1() cmd输出 可测试最大堆、各种操作、以及可视化输出当前二项堆的结构。  **void data4()设计思路：**  输入初始化数据个数以及操作数，随机数生成对应的数据，数据输入到test.txt。然后测试test.txt 输出在cmd窗口。  **输出模块：**  **void Post()设计思路：**  后序遍历：每次先进入cur的左孩子，知道最深处，然后回溯，输出节点信息，并且进入兄弟节点继续递归和回溯，输出。即“左 兄弟 根”  **void preTravel(bnode\* x)设计思路：**  前序遍历：每次先输出该节点的信息，然后进入左孩子，如果有兄弟，继续兄弟链表上遍历。  **void levelTravel()设计思路：**  层次遍历：使用队列，首先遍历该节点和兄弟链表，之后再进入该节点的左孩子，插入队列。类似于二叉树的层次遍历，只不过二叉树是右孩子，这里是兄弟，多个兄弟其实可以看作一个右孩子。  **2.4 数据及数据类(型)定义**  二项堆节点定义如下：即使用父亲兄弟链表组织树形结构，类似于文件系统的结构。    二项堆节点构成二项树 二项树节点数据结构如下：采用比较函数模板来实现比较方法的适配，可以实现最大堆，之后封装了文件输出流，使用putflag确定输出文件还是CMD，还有size 以及 整个堆的degree    **2.5.算法设计及分析**  **基础模块：**  **MakeHeap(Compare c = Compare())算法复杂度：O（1）**  {  Root,size,heap\_degree初始化。  Compare = c;//比较方法初始化  }  **BinaryHeapNode<K,V>\* Find\_Min()算法复杂度：O(log(n)):**    **BinaryHeapNode<K, V>\* Find(bnode\* t,K KEY) 的算法复杂度：O(N)**    **void Union(Bheap& Heap)的算法复杂度：O(log(n))**    **void insert(pair<K,V> x)的算法复杂度：O(log(n)):**    **pair<K, V>\* Extract\_Min() 复杂度O(log(n))**    **void Decrease\_Key(K oldkey, K newk)算法：复杂度O(logn)**    **void Delete(K KEY)的算法：复杂度为O（logn）**    **可视化模块：**  **void visualize(string filename)的算法复杂度为O（n）**  template<typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::visualize(string filename) {  //dot作图 先声明节点 然后指定颜色  int i = -1;  fstream fo(filename+".dot", ios::out);  fo << "digraph g{" << "\n";  bnode\* r = Root; string str;  if (r != NULL) {  str += "{rank=\"same\";" + to\_string(r->key) + ';';  }  while (r != NULL) {  queue<bnode\* > q;  q.push(r);  while (q.size()) {  string ss= "{rank=\"same\";";  bnode\* e = q.front(); q.pop();  bnode\* child = e->LeftChild;  while (child != NULL) {  fo << e->key << "->" << child->key<< "[color = blue]";  q.push(child);  if (child->sibiling != NULL) {  fo << ";\n";  }  else fo << "[weight=10][color=blue]; \n";  ss += to\_string(child->key) + ";";  child = child->sibiling;  }  ss += "};\n"; fo << ss << endl;  }  bnode\* temp = r; r = r->sibiling;  if (r != NULL) {  fo << temp->key << "->" << r->key << "[color=blue];" << '\n' << endl;  str += to\_string(r->key) + ";";  }  }  str += "};\n";  fo << str << endl;  if (i != -1)  fo << i << "[label=\" dest: " << i << "\",style=filled, fillcolor=green]" << endl;  fo << "}" << endl;  string sss = "dot -Tjpg " + filename + ".dot -o " + filename + ".jpg";  system(sss.c\_str());  }  **输出模块：**  **void Post(bnode\* x)的算法复杂度为O（n）：**    **void preTravel(bnode\* x)的算法复杂度为O（n）：**    **void levelTravel(bnode\* x)的算法复杂度为O（n）：**    3.测试  **-测试输入：**  使用各种大小数据进行测试，包括自建的层次化的100个样例，包括中小数据，简单数据、一般数据、大数据  简单数据：    中小数据：    大数据、复杂数据：      关于容错数据的处理：详见1.3输入格式  对于不存在的数据也有相应的处理，详情请看下方测试输出：  **-测试输出**  小数据、简单数据的输出（左1）：    异常数据的处理：    图形化输出：    **-测试中的问题及解决：**  问题：STL优先队列里没有二项堆的更新、删除指定元素的操作  解决：如果想模拟二项堆的删除指定元素，那必须将优先队列依次出队保存，直到找到元素或者全部出队为止，最后还需要把原来的不匹配的元素重新push回，这样导致写出来跑自建的1e5的样例时运行速度就很慢。 所以使用vector来模拟了STL优先队列，可以使用函数find、max\_element、erase等直接操作，写起来简单，复杂度也低。  问题：二项堆作为优先队列的一种构成方法，其实是允许重复元素的。但是这样导致dot作图会出现自环  解决：自环的原因是我dot作图把key作为节点的标识了，相同key自然看作一个节点，所以出现自环，只需要另外赋节点标识就可以了，或者再演示操作时，进了不插入重复元素。  4. 分析与探讨  **-测试结果分析：**   1. 各个操作时间复杂度：      1. 关于二项堆结构的分析:   二项堆是一个实现优先队列性能比较好的数据结构，平均性能都是logn级别的。  原因在于，与HBLT类似，合并操作是整个数据结构的核心之处，插入删除都是可以用合并来组织的。所以复杂度都会再log n级别   1. 关于二项堆的复杂度问题：   ①部分操作是O（n+log n） 比如decrease\_key、erase 是因为需要首先找到该元素才能进行操作，必然要进行一个find操作，而find只有遍历完每个节点才能知道到底这个元素存不存在，所以会有一个O（n） ②真正分析的时候，应该不考虑find这个先决条件，而是直接考虑操作的复杂度，所以都是O(log n) ③并且 实现优先队列，其实优先队列也不支持这几个操作，只需要有pop top push 对应于 extract\_min find\_min insert就可以了，而这些复杂度都是绝对的O(log n)   1. 关于二项堆的存储结构   这里使用的是父亲兄弟链表，比较经典的数据结构，能够很好的组织二项堆。  而更优化的存储结构，尝试了一下数组存储。类似于下图，想用一个类似于树状数组的组织方法，但是这种方法的核心即：找一个寻找方法代替二项堆节点的sibiling 以及 parent 以及 leftchild 但是尝试了很多方法也未果，而且即使部分成功的计算方法也比较麻烦，但是数组应该是能组织这种二进制树的结构的。     1. 对于合并操作的理解   合并操作是基础，合并操作其实是实现了二进制的加法：  如果说现在有一个七个节点的二项堆，那么他必然有一个四度、一个二度、一个一度二项树，即0111 。同理有一个13个节点的二项堆：他就是1101 两者合并之后是二十个节点，即:10100  通过结构验证一下，看是不是二项树能够对应：即一个16度、一个四度二项树：       1. 关于二项堆部分操作的优化思考：   Find\_Min操作：O(log n)是因为要扫描一遍根表，然后找最小值。但是如果我们在合并的时候，进行根表合并时，维护一个minPtr 合并的同时顺带更新minPtr 还有就是erase、insert、update的时候如果有元素交换到了根表，就看一下是不是比minPtr的key还小 如果是，则更新，最后Find\_min 只需要直接返回minPtr的key就好了。   1. 关于和他很相似的HBLT的性能对比：   HBLT的核心操作也是合并，但是他是O(logm + log n) 即 log m\*n 而二项树是 O(log (m + n) )所以复杂度一般比HBLT，程序里也进行了性能图表分析。   1. 附录：实现源代码   **注：包括以下文件:BinaryHeap.h BinaryHeap.cpp 二项堆实现**  **Makedata.h 数据生成器 STLPQ.h 标准STL的优先队列测试**  **HBLT.h 左高树的效率对比分析**  **Main.cpp 主函数以及测试、输入的一些函数**  **Qt实现源代码未放入报告，而是在代码部分。** BinaryHeap.h: #pragma once  #include<bits/stdc++.h>  using namespace std;  //允许重复值，如果多个重复的数据的话，就只对第一个进行操作。  //二项堆节点 兄弟父亲链表 组织结构  template < typename K, typename V>  struct BinaryHeapNode  {  BinaryHeapNode\* parent;  BinaryHeapNode\* LeftChild;  BinaryHeapNode\* sibiling;  int degree;  K key;//值  V data;//堆应该是个维护字典的结构  //默认构造  BinaryHeapNode() {  parent = LeftChild = sibiling = NULL;  degree = 0; key = data = INT\_MIN; //毕竟是要排序  }  BinaryHeapNode(pair<K,V> p) {  parent = LeftChild = sibiling = NULL;  degree = 0; key = p.first; data = p.second;//毕竟是要排序  }  //copy 构造  BinaryHeapNode(BinaryHeapNode& bhn) {  parent = bhn.parent;  LeftChild = bhn.LeftChild;  sibiling = bhn.sibiling;  degree = bhn.degree;  key = bhn.key;  data = bhn.data;  }  //构造  BinaryHeapNode(BinaryHeapNode\* p, BinaryHeapNode\* lc,  BinaryHeapNode\* sib, int deg, K k,V d) {  parent = p;  LeftChild = lc;  sibiling = sib;  degree = deg;  key = k;  data = d;  }  //重载等号  BinaryHeapNode& operator=(BinaryHeapNode& bhn) {  if (&bhn == this) {//排除  return \*this;  }  delete this->parent;  delete this->LeftChild;  delete this->sibiling;  this->parent = bhn.parent;  this->LeftChild = bhn.LeftChild;  this->sibiling = bhn.sibiling;  this->degree = bhn.degree;  this->key = bhn.key;  this->key = bhn.data;  return \*this;  }  //重载<号  bool operator<(const BinaryHeapNode& bhn) {  return this->key < bhn.key;  }  };  //适配！！！ 还定义了比较方法 可以最大堆！  template <typename K, typename V, typename Compare = less<K> >  class BinaryHeap  {  public:  //简化 一个是Bnode 一个是Bheap  typedef BinaryHeapNode<K, V> bnode;  typedef BinaryHeap Bheap;  ofstream fout;  ifstream fin;  string inname;  string outname;  int putflag;  private:  bnode\* Root;//根节点，最小值节点  Compare compare;//键比较器，默认小于，为最小堆!  int heap\_degree;  int size;    public:  BinaryHeap(bnode\* h, Compare c = Compare())  :Root(h), compare(c), heap\_degree(0),size(0), putflag(1) {}  BinaryHeap(Compare c = Compare())  :Root(nullptr), compare(c), heap\_degree(0), size(0), putflag(1) {}  BinaryHeap(string inn, string outn, Compare c = Compare())  {  Root = nullptr; compare = c; heap\_degree = 0; size = 0;  outname = outn;  inname = inn;  fout.open(outname);  fin.open(inname);  if (!fin.is\_open() || !fout.is\_open()) {  cout << "文件打开失败!\n";  return;  }  }  void Make\_Heap(Compare c = Compare()) {  Root = NULL; compare = c;  heap\_degree = 0; size = 0;  }//初始化一个空堆  void setcompare(Compare c) {  compare = c;  }  void lineAtTail(bnode \*&tail,bnode \*cur) {//尾端插入  if (Root == NULL) {  Root = cur;  tail = cur;  }  else {  tail->sibiling = cur;  tail = tail->sibiling;  }  }  void treeLink(bnode\* l, bnode\* r)  {//进行两颗等度数的二项树链接  l->parent = r;  l->sibiling = r->LeftChild;  r->LeftChild = l;  r->degree ++;  }  bnode\* find\_pre(bnode\* cur) {//插找某节点的 左兄弟  bnode\* pre = NULL;  if (cur->parent == NULL)  pre = Root;  else if (cur->parent->LeftChild == cur) {//无前驱  return pre;  }  else pre = cur->parent->LeftChild;  while (pre->sibiling != cur) {  pre = pre->sibiling;  }  return pre;  }  bool check\_exist(pair<K, V> p);  BinaryHeapNode<K,V>\* Find\_Min();//返回一个指向最小关键字元素的指针  BinaryHeapNode<K, V>\* Find(bnode\* t,K KEY);//找到关键字K的指针  void Union(Bheap& Heap);//与堆H合并，合并的结果保存在当前堆中,H变为空  void Merge\_Root(Bheap& Heap);//合并根表  void reverse();//逆置 用于删除  void insert(pair<K,V> x);//插入元素X  void push(pair<K, V> x) { insert(x); }  pair<K, V>\* Extract\_Min();//从堆中删除最小关键字元素，并返回指向删除元素的指针  void pop() {  Extract\_Min();  }  void top() {  Find\_Min();  }  void Decrease\_Key(K oldkey, K newk);//将元素X的关键字赋予新值K    void Delete(K KEY);//从堆中删除元素X  void Post(bnode\* x);//后序遍历  void preTravel(bnode\* x);//前序遍历  void preT();//前序遍历  void levelTravel();//层次遍历  void outputPost() { Post(Root); }  void outputpre() { preT(); };    void visualize(string filename);  //优先队列里面 应该只用到insert Extract MIN FINDmin FIND Delete Decrease\_Key  }; BinaryHeap.cpp: #include"BinaryHeap.h"  #include<bits/stdc++.h>  using namespace std;  /\* 返回关键字最小的节点  O(logn) \*/  template <typename K, typename V, typename Compare>  BinaryHeapNode<K,V>\* BinaryHeap<K, V, Compare>::Find\_Min()  {//最小的一定在根表上 扫描  if (Root == NULL) {  if(putflag)  cout << "empty\n";  else fout<< "empty\n";  return NULL;  }  //最小值指针  bnode\* minPtr = NULL;  //扫描指针  bnode\* scan = Root;  K min\_k = INT\_MAX;  while (scan != NULL) {  if (compare(scan->key, min\_k)) {  min\_k = scan->key;  minPtr = scan;  }  scan = scan->sibiling;  }  return minPtr;  }  /\* 合并两个二项堆  O(logn) \*/  template <typename K, typename V, typename Compare>  void BinaryHeap<K,V,Compare>::Union(Bheap& Heap)  {//进行堆合并  //判断非空  /\*if (Heap.Root == NULL) {  return;  }  if (Root == NULL) {  Root = Heap.Root;  Heap.Root = NULL;  return;  }\*/  //先合并根表  Merge\_Root(Heap);  if (Root == NULL) {  cout << "根表连接失败！\n";  return;  }  //扫描指针  bnode\* prex = NULL;  bnode\* cur = Root;  bnode\* next = Root->sibiling;  while (next != NULL) {//遍历每个二项树的根  if ((cur->degree != next->degree) || (next->sibiling != NULL  && next->sibiling->degree == cur->degree)) {  //当前树和下一棵树度不等， 或者三个连续的二项树度相等。  //都是要指针迁移  prex = cur; cur = next;  }  else if (compare(cur->key, next->key)) {  //如果两棵树的度相等 且cur的根key 小  cur->sibiling = next->sibiling;  treeLink(next, cur);//next 作为 cur左孩子  }  else {//next key小 cur 作为next 的左孩子  if (prex == NULL) Root = next;  else  prex->sibiling = cur->sibiling;  treeLink(cur, next);  cur = next;  }next = cur->sibiling;  }  }  /\* 合并根表 度数非递减  O(logn)  \*/  template <typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::Merge\_Root(Bheap& Heap)  {  if (Heap.Root == NULL) {  return;  }  if (Root == NULL) {  Root = Heap.Root;  Heap.Root = NULL;  return;  }  //使用两个指针扫描  bnode\* pa = Root, \*pb = Heap.Root, \*tail = NULL;  Root = NULL; Heap.Root = NULL;  while (pa != NULL && pb != NULL) {  //不断链接 until有一堆为空  if (pa->degree <= pb->degree) {  lineAtTail(tail, pa);//尾端插入  pa = pa->sibiling;  }  else {  lineAtTail(tail, pb);  pb = pb->sibiling;  }  }  //看那个还有剩下的  if (pa != NULL) {  tail->sibiling = pa;  }  else if (pb != NULL) {  tail->sibiling = pb;  }  else {//都没剩下  tail->sibiling = NULL;  }  }  /\* 插入一个新元素  O(logn)  \*/  template <typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::insert(pair<K, V> x)  {  /\*if (check\_exist(x)) {  cout<< "该元素已经存在,禁止插入！\n";  return;  }\*/  size++;  //新建一个节点  bnode\* cur = new bnode(x);  if (Root == NULL) Root = cur;  else {  //以插入节点新建一个堆  Bheap H(cur, compare);  Union(H);  }  }  /\* 从堆中删除最小/大 KEY元素，并且返回删除元素的指针  O(logn)  返回pair指针 \*/  template <typename K, typename V, typename Compare>  pair<K,V>\* BinaryHeap<K, V, Compare>::Extract\_Min()  {  //首先找到最小值minptr  if (Root == NULL) {  if(putflag)  cout << "empty\n";  else fout<< "empty\n";  return NULL;  }  bnode\* minptr = NULL, \* pre = NULL, \* cur = Root;  K mink = Root->key;//初始 根即最值  while (cur->sibiling != NULL) {//跟表上找最值  if (compare(cur->sibiling->key, mink)) {  mink = cur->sibiling->key;  pre = cur;  }  cur = cur->sibiling;  }  if (pre == NULL) {//只有一个元素 或者根为所求  minptr = Root;  Root = Root->sibiling;  }  else {//否则  minptr = pre->sibiling;  pre->sibiling = minptr->sibiling;  }  //以最小值的孩子节点建一个堆  Bheap h(minptr->LeftChild, compare);//最小值 除去根的 堆  h.reverse();//反转  Union(h);//再合并回去  pair<K, V> rv = pair<K, V>(minptr->key, minptr->data);  delete minptr;  size--;  return &rv;  }  /\*对被删除的部分进行逆置 方便合并  O(logn)  \*/  template <typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::reverse() {  bnode\* cur = Root,\*rcur;  Root = NULL;  while (cur != NULL) {  cur->parent = NULL;//第一层  rcur = cur;  cur = cur->sibiling;  if (Root == NULL) {  Root = rcur;  rcur->sibiling = NULL;  }  else {  rcur->sibiling = Root;  Root = rcur;  }  }  }  /\* 元素X关键字赋予新值(减小)  O(logn)  \*/  template <typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::Decrease\_Key(K oldkey, K newk)  {//赋值后 还需 调整本二项树  //判断是否符合decrease要求  if (!compare(newk, oldkey)) {  if (putflag == 1)  cout << "new key is greater! Refuse!\n";  else fout << "new key is greater! Refuse!\n";  return;  }  //看看是否有该节点  bnode\* x = Find(Root,oldkey);  if (x == NULL) {  if(putflag == 1)  cout << "not exist can't decrease!\n";  else fout << "not exist can't decrease!\n";  return;  }    x->key = newk;  //从此节点开始，向上调整，比较，如果小就交换上去。  bnode\* z = x->parent, \* y = x;  while (z != NULL && z->key > y->key) {  swap(z->key, y->key);  y = z;  z = y->parent;  }  }  /\* 删除关键字为Key的节点  O(logn)+O(n)  \*/  template <typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::Delete(K KEY)  {  bnode\* x = Find(Root,KEY);  if (x == NULL) {  if(putflag == 1)  cout << "not exist! can't delete！\n";  else  fout << "not exist! can't delete！\n";  return ;  }  Decrease\_Key(x->key, INT\_MIN);  Extract\_Min();    }  template<typename K, typename V, typename Compare>  BinaryHeapNode<K, V>\* BinaryHeap<K, V, Compare>::Find(bnode\* t, K KEY)  {  //前序遍历进行寻找  bnode\* cur = t, \* x = NULL;  while (cur != NULL) {  if (cur->key == KEY)  return cur;  else {  x = Find(cur->LeftChild, KEY);  if (x != NULL) {//递归找  return x;  }  cur = cur->sibiling;  }  }  return NULL;//没找到  }  template<typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::Post(bnode\* x)  {//后序遍历  bnode\* cur = x;  while (cur != NULL) {  Post(cur->LeftChild);  cout << "key is " << cur->key << "\ndata is " << cur->data  << "\ndegree is " << cur->degree << "\n";  cur = cur->sibiling;  }  }  template<typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::preTravel(bnode\* x)  {//前序遍历  bnode\* cur = x;  if (cur != NULL) {  cout << "key is " << cur->key << "\ndata is " << cur->data  << "\ndegree is " << cur->degree << "\n";  if (cur->LeftChild != NULL) {  for (bnode\* t = cur->LeftChild; t != NULL; t = t->sibiling) {  preTravel(t);  }  }  }  }  template<typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::preT()  {  for (bnode\* t = Root; t != NULL; t = t->sibiling) {  preTravel(t);  }  }  template<typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::levelTravel()  {//层次遍历  queue< bnode\* > q; q.push(Root);  while (!q.empty()) {  bnode\* cur = q.front(); q.pop();  bnode\* t = cur;  while (t->sibiling !=NULL)  {  q.push(t->sibiling);  t = t->sibiling;  }  if (cur->LeftChild != NULL) q.push(cur->LeftChild);  cout << "key is " << cur->key << "\ndata is " << cur->data  << "\ndegree is " << cur->degree << "\n";  }  }  template<typename K, typename V, typename Compare>  bool BinaryHeap<K, V, Compare>::check\_exist(pair<K,V> p)  {  queue< bnode\* > q; q.push(Root);  while (!q.empty()) {  bnode\* cur = q.front(); q.pop();  if (cur->key == p.first) {  return true;  }  bnode\* t = cur;  while (t->sibiling != NULL)  {  q.push(t->sibiling);  t = t->sibiling;  }  if (cur->LeftChild != NULL) q.push(cur->LeftChild);  }  return false;  }  template<typename K, typename V, typename Compare>  void BinaryHeap<K, V, Compare>::visualize(string filename) {  //dot作图 先声明节点 然后指定颜色  int i = -1;  fstream fo(filename+".dot", ios::out);  fo << "digraph g{" << "\n";  bnode\* r = Root;  string str;  if (r != NULL) {  str += "{rank=\"same\";" + to\_string(r->key) + ';';  }  while (r != NULL) {  queue<bnode\* > q;  q.push(r);  while (q.size()) {  string ss= "{rank=\"same\";";  bnode\* e = q.front(); q.pop();  bnode\* child = e->LeftChild;  while (child != NULL) {  fo << e->key << "->" << child->key<< "[color = blue]";  q.push(child);  if (child->sibiling != NULL) {  fo << ";\n";  }  else fo << "[weight=10][color=blue]; \n";  ss += to\_string(child->key) + ";";  child = child->sibiling;  }  ss += "};\n";  fo << ss << endl;  }  bnode\* temp = r;  r = r->sibiling;  if (r != NULL) {  fo << temp->key << "->" << r->key << "[color=blue];" << '\n' << endl;  str += to\_string(r->key) + ";";  }  }  str += "};\n";  fo << str << endl;  if (i != -1)  fo << i << "[label=\" dest: " << i << "\",style=filled, fillcolor=green]" << endl;  fo << "}" << endl;  string sss = "dot -Tjpg " + filename + ".dot -o " + filename + ".jpg";  system(sss.c\_str());  } Main.cpp: #include<bits/stdc++.h>  #include"BinaryHeap.h"  #include"HBLT.h"  #include"BinaryHeap.cpp"  #include"STLPQ.h"  #include<windows.h>  #include"makedata.h"  using namespace std;  void comparetime() {  for (int i = 1; i <= 80; i++) {  string inname = "F:\\vscpp\\BinaryHeap\\BinaryHeap\\input\\" + to\_string(i) + ".txt";  string outname = "F:\\vscpp\\BinaryHeap\\BinaryHeap\\outputhblt" + to\_string(i) + ".txt";  cout << "\t\t\t\t\t 正在测试HBLT " << to\_string(i) << " 个数据...\n";  maxHBLT<int> bh;  ifstream fin(inname);  ofstream fout(outname);  LARGE\_INTEGER start\_time; //开始时间  LARGE\_INTEGER end\_time; //结束时间  double dqFreq; //计时器频率  LARGE\_INTEGER freq; //计时器频率  QueryPerformanceFrequency(&freq);  dqFreq = (double)freq.QuadPart;  QueryPerformanceCounter(&start\_time); //计时开始  int n, opn; fin >> n >> opn;  for (int j = 1; j <= n; j++) {  int num; fin >> num;  bh.push(num);  }  for (int j = 1; j <= opn; j++) {  char op; fin >> op;  if (op == 'A') {  int num; fin >> num;  bh.push(num);  }  else if (op == 'B')  {  fout << bh.front() << "\n";  }  else if (op == 'C')  {  fout << bh.front() << "\n";  bh.pop();  }    }  QueryPerformanceCounter(&end\_time); //计时end  fin.close();  fout.close();  ofstream fo("cost2.txt", ios::app);  double run\_time = (end\_time.QuadPart - start\_time.QuadPart) / dqFreq \* 1000;  if (i <= 80)//只测前80个  fo << i << " " << run\_time << "\n";  fo.close();  }  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "\t\t\t\t\t 图表马上就来...\n";  cout << "\t\t\t\t\t HBLT 合并复杂度是:O(log(mn)) Bheap 复杂度是O(logn)\n";  system("testtime.py");  }  void showops()  {  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "\n\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*二项堆操作\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";  cout << "\n\n请输入下列框中的字母:" << endl;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_GREEN);  cout << "\t\t\t\t\t [A] insert x" << endl;  cout << "\t\t\t\t\t [B] Extract\_MIN" << endl;  cout << "\t\t\t\t\t [C] Find\_Min" << endl;  cout << "\t\t\t\t\t [D] Delete x" << endl;  cout << "\t\t\t\t\t [E] Decrease\_Key oldk newk" << endl;  cout << "\t\t\t\t\t [F] visualize" << endl;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";  cout << "\n";  }  void check()  {//用system 对拍  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_BLUE);  system("check.exe");  }  //跑100个数据集并对拍  void data1()  {//计划输入100个样例 然后 对拍  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "\t\t\t\t\t 正在测试100个数据集...\n";  int putf = 0;//0写入文件 1cmd  for (int i = 1; i <= 100; i++) {  string inname = "F:\\vscpp\\BinaryHeap\\BinaryHeap\\input\\" + to\_string(i) + ".txt";  string outname= "F:\\vscpp\\BinaryHeap\\BinaryHeap\\output\\output" + to\_string(i) + ".txt";  cout << "\t\t\t\t\t 正在测试 " << to\_string(i) << " 个数据...\n";  BinaryHeap<int, int, less<int> > bh(inname,outname);  bh.putflag = 0;  LARGE\_INTEGER start\_time; //开始时间  LARGE\_INTEGER end\_time; //结束时间  double dqFreq; //计时器频率  LARGE\_INTEGER freq; //计时器频率  QueryPerformanceFrequency(&freq);  dqFreq = (double)freq.QuadPart;  QueryPerformanceCounter(&start\_time); //计时开始  int n, opn; bh.fin >> n >> opn;  for (int j = 1; j <= n; j++) {  int num; bh.fin >> num;  bh.insert(make\_pair(num,num));  }  for (int j = 1; j <= opn; j++) {  char op; bh.fin >> op;  if (op == 'A') {  int num; bh.fin >> num;  bh.insert(make\_pair(num, num));  }  else if(op == 'B')  {  bh.fout << bh.Extract\_Min()->first << "\n";  }  else if (op == 'C')  {  bh.fout <<bh.Find\_Min()->key<<"\n";  }  else if (op == 'D')  {  int num; bh.fin >> num;  bh.Delete(num);  }  else if (op == 'E')  {  int oldk, newk; bh.fin >> oldk >> newk;  bh.Decrease\_Key(oldk, newk);  }  }  QueryPerformanceCounter(&end\_time); //计时end  bh.fin.close();  bh.fout.close();  ofstream fo("cost1.txt", ios::app);  double run\_time = (end\_time.QuadPart - start\_time.QuadPart) / dqFreq \* 100;  if(i<=80)//只测前80个  fo << i << " " << run\_time << "\n";  fo.close();  }  cout << "\t\t\t\t\t 测试完成，进行对拍";  cout << "\t\t\t\t\t 正在进行对拍...\n";  check();  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "\t\t\t\t\t 是否需要展示和HBLT的效率对比图？(Y/N)\n";  char see; cin >> see;  if (see == 'Y') {  comparetime();  }  else {  cout << "\t\t\t\t\t 测试完毕！\n";  }  }  //输入指定路径下的文件 进行输入cmd输出  void data2() {  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN | FOREGROUND\_BLUE);  cout << "\t\t\t\t 请输入路径: (例如:F:\\vscpp\\BinaryHeap\\BinaryHeap\\input\\1.txt)\n";  cout << "\t\t\t\t ";  string inn; cin >> inn;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "结果如下： \n";  BinaryHeap<int, int, less<int> > bh;  bh.inname = inn; bh.fin.open(bh.inname);  bh.putflag = 1;  int n, opn; bh.fin >> n >> opn;  for (int j = 1; j <= n; j++) {  int num; bh.fin >> num;  bh.insert(make\_pair(num, num));  }  for (int j = 1; j <= opn; j++) {  char op; bh.fin >> op;  if (op == 'A') {  int num; bh.fin >> num;  bh.insert(make\_pair(num, num));  }  else if (op == 'B')  {  cout << bh.Extract\_Min()->first << "\n";  }  else if (op == 'C')  {  cout << bh.Find\_Min()->key << "\n";  }  else if (op == 'D')  {  int num; bh.fin >> num;  bh.Delete(num);  }  else if (op == 'E')  {  int oldk, newk; bh.fin >> oldk >> newk;  bh.Decrease\_Key(oldk, newk);  }  }  bh.fin.close();  }  //手动输入数据 cmd输出 可测试最大堆  void data3() {  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "需要使用最大二项堆[0]还是最小二项堆[1]\n";  int flag; cin >> flag;  if (flag == 0) {  BinaryHeap<int, int, greater<int> > bh;  bh.putflag = 1;  cout << "\t\t\t\t请输入初始化数据个数以及操作个数\n";  int n, m; cin >> n >> m;  cout << "\t\t\t\t请输入初始化数据\n";  for (int i = 1; i <= n; i++) {  int num; cin >> num;  bh.insert(make\_pair(num, num));  }  showops();  for (int i = 1; i <= m; i++) {  char op; cin >> op;  if (op == 'A') {  int num; cin >> num;  bh.insert(make\_pair(num, num));  }  else if (op == 'B')  {  auto s = bh.Extract\_Min();  if (s != NULL)  cout << s->first << "\n";  else  cout << "已空！！请输入数据！\n";  }  else if (op == 'C')  {  cout << bh.Find\_Min()->key << "\n";  }  else if (op == 'D')  {  int num; cin >> num;  bh.Delete(num);  }  else if (op == 'E')  {  int oldk, newk; cin >> oldk >> newk;  bh.Decrease\_Key(oldk, newk);  }  else if (op == 'F') {  cout << "\t\t\t\t请输入图片的名字\n";  string name; cin >> name;  bh.visualize(name);  cout << "\t\t\t\t已生成图片在本地，请查看！\n";  }  else {  cout << "\t\t\t\t输入有误 重新输入！\n";  }  }  return;  }  else {  BinaryHeap<int, int, less<int> > bh;  bh.putflag = 1;  cout << "\t\t\t\t请输入初始化数据个数以及操作个数\n";  int n, m; cin >> n >> m;  cout << "\t\t\t\t请输入初始化数据\n";  for (int i = 1; i <= n; i++) {  int num; cin >> num;  bh.insert(make\_pair(num, num));  }  showops();  for (int i = 1; i <= m; i++) {  char op; cin >> op;  if (op == 'A') {  int num; cin >> num;  bh.insert(make\_pair(num, num));  }  else if (op == 'B')  {  auto s = bh.Extract\_Min();  if (s != NULL)  cout << s->first << "\n";  else  cout << "已空！！请输入数据！\n";  }  else if (op == 'C')  {  cout << bh.Find\_Min()->key << "\n";  }  else if (op == 'D')  {  int num; cin >> num;  bh.Delete(num);  }  else if (op == 'E')  {  int oldk, newk; cin >> oldk >> newk;  bh.Decrease\_Key(oldk, newk);  }  else if (op == 'F') {  cout << "\t\t\t\t请输入图片的名字\n";  string name; cin >> name;  bh.visualize(name);  cout << "\t\t\t\t已生成图片在本地，请查看！\n";  }  else {  cout << "\t\t\t\t输入有误 重新输入！\n";  }  }  }  }  //选择性的建一个新数据集 然后对他进行操作 cmd输出  void data4() {  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_BLUE);  cout << "\t\t\t\t\t 输入初始化数据个数以及操作数" << endl;  int nn, mm; cin >> nn >> mm;  ofstream fout("test.txt");  makedata2(fout, mm, nn);  fout.close();  cout << "\t\t\t\t\t 新数据集创建完毕，请在本地检查test.txt\n";  cout << "\t\t\t\t\t 下面测试test.txt 输出在cmd窗口\n";  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "结果如下： \n";  BinaryHeap<int, int, less<int> > bh;  bh.inname = "test.txt"; bh.fin.open(bh.inname);  bh.putflag = 1;  int n, opn; bh.fin >> n >> opn;  for (int j = 1; j <= n; j++) {  int num; bh.fin >> num;  bh.insert(make\_pair(num, num));  }  for (int j = 1; j <= opn; j++) {  char op; bh.fin >> op;  if (op == 'A') {  int num; bh.fin >> num;  bh.insert(make\_pair(num, num));  }  else if (op == 'B')  {  cout << bh.Extract\_Min()->first << "\n";  }  else if (op == 'C')  {  cout << bh.Find\_Min()->key << "\n";  }  else if (op == 'D')  {  int num; bh.fin >> num;  bh.Delete(num);  }  else if (op == 'E')  {  int oldk, newk; bh.fin >> oldk >> newk;  bh.Decrease\_Key(oldk, newk);  }  }  bh.fin.close();  }  void stdput()  {  cout << "正在生成标准输出......\n";  for (int i = 81; i <= 100; i++) {  cout << "正在生成数据集 "<<to\_string(i)<<"的标准输出\n";  string inn = "F:\\vscpp\\BinaryHeap\\BinaryHeap\\input\\" + to\_string(i) + ".txt";  ifstream fin(inn);  string outt = "F:\\vscpp\\BinaryHeap\\BinaryHeap\\outputSTD\\outputstd" + to\_string(i) + ".txt";  ofstream fout(outt);  if (!fin.is\_open() || !fout.is\_open()) {  cout << "文件打开失败\n";  return;  }  int n, m;  fin >> n >> m;  putSTD1(fin, fout, m, n);  fin.close(); fout.close();  }  cout << "标准输出完成!\n";  }  void welcome() {//欢迎  fir:SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "\n\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*二项堆实现与分析\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";  cout << "\n\n请输入下列框中的数字:" << endl;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_BLUE);  cout << "\t\t\t\t\t [0] 退出" << endl;  cout << "\t\t\t\t\t [1] 测试100个自建样例" << endl;  cout << "\t\t\t\t\t [2] 输入指定路径下的文件" << endl;  cout << "\t\t\t\t\t [3] 手动输入数据并进行可视化展示" << endl;  cout << "\t\t\t\t\t [4] 自建数据集,并对拍" << endl;  cout << "\n\t\t\t\t\t 更详情的可视化展示请见Qt版本" << endl;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN);  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";  while (1)  {  int op;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED |  FOREGROUND\_GREEN | FOREGROUND\_BLUE);  cout << "\n\n\t\t\t\t\t\t\t 请输入:";  //lf: cin >> op;  cin >> op;  //if (cin.fail()) //判断输入类型  //{  // cout << "\t\t\t\t\t\t 数据类型错误,请输入整数:";  // cin.clear(); //清除错误标记  // cin.sync(); //清除缓冲区  // goto lf; //重新输入  //}  if (op == 0) {  cout << "\t\t\t\t\t\t 程序已退出!\n";  }  else if (op == 1) {  data1();  }  else if (op == 2) {  data2();  }  else if (op == 3) {  data3();  }  else if (op == 4) {  data4();  }  else  {  cout << "\t\t\t\t\t\t 数据类型错误,请输入整数:";  }  }  }  int main()  {  welcome();  return 0;  } Makedata.h: #pragma once  #include<bits/stdc++.h>  #include<fstream>  #include <stdlib.h>  #include <time.h>  using namespace std;  //写个数据生成器 BUT 感觉 很容易找不到  // 解决： 存一下输入  // 1 - 80 只有pop push top  // 81 -- 100 pop push top delete decrease  // 写操作指令到哪个文件中 并且一共多少条指令 初始插入多少个数字  //到时候和 HBLT对比效率  //和STL的优先队列对拍  //  // 81 -- 100 pop push top delete decrease  void makedata2(ofstream& fout, int total\_ops, int total\_nums) {  fout << total\_nums << " ";  fout << total\_ops << "\n";  srand((unsigned)time(NULL));  if (!fout.is\_open()) {  cout << "文件打开失败！\n";  return;  }  vector<int> vc;  for (int i = 1; i <= total\_nums; i++) {  int num = rand() % 100000000;  fout << rand() % 100000000 << " ";  vc.push\_back(num);  }  fout << "\n";  for (int i = 1; i <= total\_ops; i++) {  //insert Extract MIN FINDmin Delete Decrease\_Key  // A B C D E  // 0 1 2 3 4  //  int opnum = rand() % 5;  if (opnum == 0) {  fout << 'A' << " ";  int num = rand() % 100000000;  vc.push\_back(num);  fout << num << "\n";  }  else if (opnum == 1) {  fout << 'B' << "\n";  }  else if (opnum == 2) {  fout << 'C' << "\n";  }  else if (opnum == 3) {  fout << 'D' << " ";  int pos = rand() % vc.size();  int num = vc[pos];  fout << num << "\n";  }  else if (opnum == 4) {  fout << 'E' << " ";  int pos = rand() % vc.size();  int num = vc[pos];  fout << num << " ";  num = rand() % 100000000;  vc.push\_back(num);  fout << num << "\n";  }  }  }  // 1 - 80 只有pop push top  void makedata1(ofstream& fout, int total\_ops, int total\_nums) {  fout << total\_nums << " ";  fout << total\_ops << "\n";  srand((unsigned)time(NULL));  if (!fout.is\_open()) {  cout << "文件打开失败！\n";  return;  }  vector<int> vc;  for (int i = 1; i <= total\_nums; i++) {  int num = rand() % 100000000;  fout << rand() % 100000000 << " ";  vc.push\_back(num);  }  fout << "\n";  for (int i = 1; i <= total\_ops; i++) {  //insert Extract MIN FINDmin  // A B C  // 0 1 2  int opnum = rand() % 3;  if (opnum == 0) {// push  fout << 'A' << " ";  int num = rand() % 100000000;  vc.push\_back(num);  fout << num << "\n";  }  else if (opnum == 1) {// top + pop  fout << 'B' << "\n";  }  else if (opnum == 2) {// top  fout << 'C' << "\n";  }  }  } STLPQ.h： #pragma once  #include<bits/stdc++.h>  using namespace std;  //力求正确性 不求快速！  void putSTD1(ifstream &fin, ofstream &fout,int total\_ops, int total\_nums) {  //使用STL优先队列进行  priority\_queue<int, vector<int>, greater<int> > pq;  vector<int> temps; int flag = 0;  for (int i = 1; i <= total\_nums; i++) {  //check no 重复数据  flag = 0;  int num; fin >> num;  //允许重复数据  pq.push(num);  }  //ops  for (int i = 1; i <= total\_ops; i++) {  char op; fin >> op;  //insert Extract MIN FINDmin Delete Decrease\_Key  // A B C D E  // 0 1 2 3 4  if (op == 'A') {  int num; fin >> num;  pq.push(num);  }    else if (op == 'B') {  fout << pq.top()<<"\n"; pq.pop();  }    else if (op == 'C') {  fout << pq.top()<<"\n";  }    else if (op == 'D') {//delete  int num; fin >> num;  flag = 0;  for (int j = 1; j <= pq.size(); j++) {  int temp = pq.top(); pq.pop();  if (num == temp) {  flag = 1;  break;  }  else  temps.push\_back(temp);  }  //都放回去回去 除了要删除的  for (auto& x : temps) {  if (x != num)  pq.push(x);  }  temps.clear();  if (flag == 0) {  fout << "not exist! can't delete！\n";  }  }    else if (op == 'E') {  //Decrease\_Key()  int oldk, newk; fin >> oldk >> newk;  //check 有没有oldk  flag = 0;  for (int j = 1; j <= pq.size(); j++) {  int temp = pq.top(); pq.pop();  if (oldk == temp) {  flag = 1;  break;  }  else temps.push\_back(temp);  }  //都放回去回去 除了要修改的  for (auto& x : temps) {  if (x != oldk)  pq.push(x);  }  pq.push(newk);  temps.clear();  if (flag == 0) {  fout << "not exist can't decrease!\n";  continue;  }  if (oldk > newk) {  fout << "new key is greater ! Refuse!\n";  continue;  }  }  }  }  //力求正确性 不求快速！  void putSTD2(ifstream& fin, ofstream& fout, int total\_ops, int total\_nums) {  //使用STL优先队列进行  vector<int> pq;  for (int i = 1; i <= total\_nums; i++) {  //check no 重复数据  int num; fin >> num;  //允许重复数据  pq.push\_back(num);  }  //ops  for (int i = 1; i <= total\_ops; i++) {  char op; fin >> op;  //insert Extract MIN FINDmin Delete Decrease\_Key  // A B C D E  // 0 1 2 3 4  if (op == 'A') {  int num; fin >> num;  pq.push\_back(num);  }  else if (op == 'B') {  auto pos = max\_element(pq.begin(), pq.end());  fout << \*pos << "\n";  pq.erase(pos);  }  else if (op == 'C') {  auto pos = max\_element(pq.begin(), pq.end());  fout << \*pos << "\n";  }  else if (op == 'D') {//delete  int num; fin >> num;  int flag = 0;  for (auto j = pq.begin(); j != pq.end(); j++) {  if (num == \*j) {  pq.erase(j);  flag = 1;  break;  }  }  if (flag == 0) {  fout << "not exist! can't delete！\n";  }  }  else if (op == 'E') {  //Decrease\_Key()  int oldk, newk; fin >> oldk >> newk;  //先看大小  if (oldk > newk) {  fout << "new key is greater ! Refuse!\n";  continue;  }  //check 有没有oldk  int flag = 0;  for (auto j = pq.begin(); j != pq.end(); j++) {  if (oldk == \*j) {  \*j = newk;  flag = 1;  break;  }  }  if (flag == 0) {  fout << "not exist can't decrease!\n";  continue;  }  }  }  } HBLT.h: #pragma once  #include<iostream>  #include<queue>  using namespace std;  template<class T>  struct binaryTreeNode//定义二叉树节点  {  T element;//当前节点数值  binaryTreeNode<T>\* leftChild;//左子节点指针  binaryTreeNode<T>\* rightChild;//右子节点指针  binaryTreeNode()  {  leftChild = NULL;  rightChild = NULL;  }  binaryTreeNode(const T& theElement)  {  element = theElement;  leftChild = NULL;  rightChild = NULL;  }  binaryTreeNode(const T& theElement, binaryTreeNode<T>\* LEFT, binaryTreeNode<T>\* RIGHT)  {  element = theElement;  leftChild = LEFT;  rightChild = RIGHT;  }  };  template<class T>  class maxHBLT//最大HBLT类  {  private:  void meld(binaryTreeNode<pair<int, T>>\*& x, binaryTreeNode<pair<int, T>>\*& y)//递归合并两个左高树，合并结果为x，此函数需要封装  {  //递归终点  if (y == nullptr)  return;  if (x == nullptr)  {  x = y;  return;  }  //保证x根节点所存数据值比y大  if (x->element.second < y->element.second)  swap(x, y);  //x与y的合并转化为x的右子树与y的合并，从而进行递归  meld(x->rightChild, y);  //递归结束之后需要对左高树的形状以及各个节点的s()进行维护  if (x->leftChild == nullptr)//x的左子树为空，则交换左右子树  {  x->leftChild = x->rightChild;  x->rightChild = nullptr;  x->element.first = 1;  }  else  {  if (x->leftChild->element.first < x->rightChild->element.first)//x的左子树的s()小于右子树的s()，两者交换，并重新计算x的s()  swap(x->leftChild, x->rightChild);  x->element.first = x->rightChild->element.first + 1;//左高树根节点的s()就是从根节点一直向右走到达外部节点的距离  }  }  binaryTreeNode<pair<int, T>>\* root;//根节点，其元素为一个数对，前项表示根节点的s()，后项表示根节点存储的数据  int treeSize;//节点总个数  public:  maxHBLT()//空树  {  root = nullptr;  treeSize = 0;  }  T front()//获取最大值（根节点的数据值）  {  return root->element.second;  }  //插入：相当于原树与一个只有一个元素的最大HBLT树合并  void push(const T& theElement)  {  binaryTreeNode<pair<int, T>>\* newNode = new binaryTreeNode<pair<int, T>>(pair<int, T>(1, theElement));  meld(root, newNode);  treeSize++;  }  //删除：相当于根节点的左右子树合并  void pop()  {  if (treeSize == 0)  {  cout << "树为空" << endl;  return;  }  binaryTreeNode<pair<int, T>>\* left = root->leftChild;  binaryTreeNode<pair<int, T>>\* right = root->rightChild;  delete root;  root = left;  meld(root, right);  treeSize--;  }  //合并两个最大HBLT树  void meld(maxHBLT<T>& theHBLT)  {  meld(root, theHBLT.root);  treeSize += theHBLT.treeSize;  theHBLT.root = nullptr;  theHBLT.treeSize = 0;  }  //将一个数组转化为最大HBLT树  void initialize(T\* theElements, int theSize)  {  if (theSize < 1)  {  cout << "初始化无效" << endl;  return;  }  queue<binaryTreeNode<pair<int, T>>\*> queue;  delete root;  for (int i = 1; i <= theSize; i++)//先将数组中的元素转化为孤立的树的节点，存入队列中  queue.push(new binaryTreeNode<pair<int, T>>(pair<int, T>(1, theElements[i])));  for (int i = 1; i <= theSize - 1; i++)//然后循环取出两个节点，合并，然后再存入队列中  {  binaryTreeNode<pair<int, T>>\* node1 = queue.front();  queue.pop();  binaryTreeNode<pair<int, T>>\* node2 = queue.front();  queue.pop();  meld(node1, node2);  queue.push(node1);  }  root = queue.front();//最终所有节点合并成一个最大HBLT树  treeSize = theSize;  }  //输出树（检查用）  void print()  {  int size = treeSize;  for (int i = 0; i < size; i++)  {  cout << front() << " ";  pop();  }  cout << endl;  }  }; | | | |