山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201700140056 | 姓名：李港 | | 班级：跟18.2（17.4） |
| 实验题目：实验八 散列表 | | | |
| 实验学时：2h | | 实验日期：2019.11.07 | |
| 实验目的：  1、掌握散列表结构的定义和实现。  2、掌握散列表结构的应用。 | | | |
| 软件开发工具：  Virtual Studio 2019 | | | |
| 1. **实验内容**    1. 分别使用线性开型寻址和链表散列解决溢出，创建散列表类。    2. 使用散列表设计实现一个字典，假设关键字为整数且D为961，在字典中插入随机产生的500个不同的整数，实现字典的建立和搜索操作。\*实现字典的删除。 2. **数据结构与算法描述（整体思路描述，所需要的数据结构与算法）**   **总体思路：**   * 1. 线性开型寻址使用原生数组存储元素。   2. 链表散列使用链表存储元素   3. 两种结构均提供插入、删除、查询等操作,两种结构相同功能函数的参数与返回值相同，确保兼容性。   4. 两种结构的某些函数因 OJ 的要求不同而有细微区别。   5. 使用模板元编程的方法提供不同类型的哈希函数。   6. int型哈希函数直接返回原数值。   7. 在基本数据结构方面，线性开型寻址的散列表比链表散列的散列表简单。   8. 在实现功能方面，链表散列的散列表比线性开型寻址的散列表简单。   **数据结构：**   * 1. 哈希表采用原生数组作为底层数据结构   2. 链表散列使用链表类作为底层数据结构，一个链表散列的散列表类是多个链表的集合。链表提供插入，删除等操作，散列表根据哈希函数分发这些操作到不同链表中。   3. 使用mypair类来存储键值对，提供构造函数；线性开型寻址和链表散列的散列表类使用相同的mypair类。   4. 链表使用mynode作为节点。   **算法：**   * 1. 线性开型寻址：      1. 搜索：         1. 先获得关键词的桶编号         2. 若该处为空，则搜索失败         3. 若该处元素关键词不需要的关键词，则向后寻找         4. 若将表搜索一遍仍未找到目标元素，则搜索失败      2. 插入：应当注意，线性开型寻址的散列表存在无法插入的情况         1. 首先获得关键词对应的桶编号，并不断向后对比         2. 对比过程中，若桶中键值对关键词与要插入的关键词相同，则修改对应数据         3. 若找不到关键词但有剩余空间则插入新的键值对         4. 若找遍数组都没找到对应关键词，也没有剩余空间，则插入失败      3. 删除：需要注意删除元素后的数组整理过程         1. 首先按照搜索的方法找到对应位置，然后删除元素         2. 接下来索引不断增加，将后续元素进行整理            1. 若索引指向位置的键值对的关键词不在它应该在的位置上（通过\_getPosByKey得到），则将此元素移动到它该在的位置上，并清空原来的位置。            2. 若索引指向位置为空，则说明该表已整理好，返回移动的元素的个数。   2. 链表散列：      1. 搜索：         1. 先获得关键词的链表编号         2. 再从相应链表中寻找该元素         3. 找到则返回数据指针，找不到返回空指针      2. 插入：         1. 将键值对插入到相应的链表中         2. 若关键词已存在则修改关键词对应的数据      3. 删除：         1. 获得关键词对应的链表编号         2. 在链表中删除该元素            1. 若链表中存在此元素，则删除之            2. 若链表中不存在此元素，则删除失败  1. **测试结果（测试输入，测试输出）**    1. 验收展示： 插入完毕后，屏幕上依次打印：查找过程，整个散列表，删除过程，整个散列表； 可见代码结果正确。       1. 线性开型寻址的散列表类：       2. 链表散列的散列表类：    2. 平台提交       1. 线性开型寻址      * + 1. 链表散列：      1. **分析与探讨（结果分析，若存在问题，探讨解决问题的途径）**   本实验最终结果正确，在实验过程中有以下问题或心得：   * 1. 使用OJ给出的样例运行正常，但OJ就是通不过，后来添加了更多测试样例才开始正常debug。   2. 在线性开型寻址中有个公有函数getPosByKey与私有函数\_getPosByKey，某些函数错误地将\_getPosByKey调用成了\_getPosByKey。导致debug很久。   3. 后来又发现线性开型寻址erase函数在删除元素后将后续元素前移的逻辑出现了错误。   4. 即使正确前移，在oj上提交还是错误，后来发现是delete之后没有将size减一，导致用到size的函数运行出错。   5. 进行总结，发现上述bug的出现是因为没有提前注意的缘故，考虑到我没办法听课，应当提前在网络上搜索了解各种易出错的坑。   6. 链表散列的散列表比线性开型寻址的散列表简单一些，由此可见，基本结构简单，实现功能的代码不一定简单。算法与数据结构是相辅相成的。  1. **附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）**   文件1 hashchain.h  # pragma once  #include<iostream>  /\*对应各种数据类型的哈希函数，此处只写int的哈希函数\*/  template <class K> class myhash;  template<>  class myhash<int> {  public:  int operator()(const int keyin) const { return int (keyin); }  };  /\*用于存储键值的类\*/  template <class K, class E>  class mypair {  public:  K key;  E data;  mypair (K keyin, E datain) :key (keyin), data (datain) {}  };  /\*用于构成链表的结点类\*/  template <class K, class E>  struct mynode {  mypair< K, E> element;  mynode< K, E>\* next;  mynode (mypair< K, E> pairin) :element (pairin), next (nullptr) {}  mynode (mypair< K, E> pairin, mynode< K, E>\* nextin) :element (pairin), next (nextin) {}  };  /\*链表类  template<class K, class E>  class myChain {  protected:  mynode<K, E>\* \_head; //保存链表头  int \_size; //保存元素个数  public:  myChain (); //构造函数  ~myChain (); //析构函数  bool empty () const; //返回是否为空  int size () const; //返回元素数目  mypair< K, E>\* find (const K& keyin) const; //查找元素，找不到返回空指针  bool erase (const K& keyin); //删除元素，无此元素返回false，删除成功返回true  void insert (mypair< K, E>& pairin); //插入函数，在插入时确保有序  void output (ostream& out) const; //输出链表元素  \*/  template<class K, class E>  class myChain {  protected:  mynode<K, E>\* \_head;  int \_size;  public:  myChain () :\_head (nullptr), \_size (0) {}  ~myChain () {  while (\_head != nullptr) {  mynode<K, E>\* nextNode = \_head->next;  delete \_head;  \_head = nextNode;  }  }  bool empty () const { return \_size == 0; }  int size () const { return \_size; }  mypair< K, E>\* find (const K& keyin) const {  mynode<K, E>\* c\_node = \_head;  /\*一直寻找直到找到对应元素\*/  for (;;) {  if (c\_node == nullptr || c\_node->element.key == keyin) {  break;  }  c\_node = c\_node->next;  }  /\*如果值不为空，则\*/  if (c\_node != nullptr && c\_node->element.key == keyin) {  return &c\_node->element;  } else {  return nullptr;  }  }  bool erase (const K& keyin) {  mynode<K, E>\* nodep = \_head;  mynode<K, E>\* pre\_nodep = nullptr;  while (nodep != nullptr && nodep->element.key < keyin) {  pre\_nodep = nodep;  nodep = nodep->next;  }  if (nodep != nullptr && nodep->element.key == keyin) {  if (pre\_nodep == nullptr) \_head = nodep->next;  else pre\_nodep->next = nodep->next;  delete nodep;  \_size--;  return true;  } else {  return false;  }  }  /\*插入函数，在插入时确保有序\*/  void insert (mypair< K, E>& thePair) {  mynode<K, E>\* nodep = \_head;  mynode<K, E>\* pre\_nodep = nullptr;  while (nodep != nullptr && nodep->element.key < thePair.key) {  pre\_nodep = nodep;  nodep = nodep->next;  }  /\*查看是否已有此元素\*/  if (nodep != nullptr && nodep->element.key == thePair.key) {  //已有则插入则修改该位置元素  nodep->element.data = thePair.data;  return;  } else {  //无则插入新元素  mynode<K, E>\* newNode = new mynode<K, E> (thePair, nodep);  if (pre\_nodep == nullptr) \_head = newNode;  else pre\_nodep->next = newNode;  \_size++;  return;  }  }  /\*输出链表元素\*/  void output (std::ostream& out) const {  mynode<K, E>\* c\_node = \_head;  while (c\_node != nullptr) {  out << c\_node->element.key << ":" << c\_node->element.data << " ";  c\_node = c\_node->next;  }  }  };  /\*输出链表元素的重载函数\*/  template <class K, class E>  std::ostream& operator<<(std::ostream& out, const myChain<K, E>& in) {  in.output (out);  return out;  }  /\*哈希链表类  template<class K, class E>  class myhashChains {  protected:  myChain<K, E>\* \_chains; //我的链表  myhash<K> \_myhash; //自定义的哈希函数  int \_size; //元素个数  int \_divisor; //除数，链表的个数  int inline \_getPosByKey (const K& keyin)const; //获取该关键词的理论位置，注意该返回值需要配个各函数进行分别分析  public:  myhashChains (int divisorin = 20); //构造函数  ~myhashChains (); //析构函数  bool empty () const; //返回是否为空  int size () const; //返回元素个数  E\* find (const K& keyin) const; //查找关键词，返回值的指针  bool insert (mypair<K, E> pairin); //插入键值对，返回插入成功失败与否  bool erase (const K& keyin); //删除元素的包装  void output (ostream& out) const; //输出元素  int getLengthByKey (const K& keyin) const; //通过关键词获取长度，为了OJ而增加  \*/  template<class K, class E>  class myhashChains {  protected:  myChain<K, E>\* \_chains;  myhash<K> \_myhash;  int \_size;  int \_divisor;  int inline \_getPosByKey (const K& keyin)const {  return \_myhash (keyin) % \_divisor;  }  public:  myhashChains (int divisorin = 20) {  \_size = 0;  \_divisor = divisorin;  \_chains = new myChain<K, E>[\_divisor];  }  ~myhashChains () { delete[] \_chains; }  /\*查找关键词，返回值的指针\*/  E\* find (const K& keyin) const {  mypair< K, E>\* temp = (\_chains[\_getPosByKey (keyin)].find (keyin));  if (temp == nullptr) {  return nullptr;  } else {  return &(temp->data);  }  }  /\*插入键值对，返回插入成功失败与否\*/  bool insert (mypair<K, E> pairin) {  int pos = \_getPosByKey (pairin.key);  int size = \_chains[pos].size ();  \_chains[pos].insert (pairin);  /\*链表长度扩大说明插入了，没有扩大说明覆盖了\*/  if (\_chains[pos].size () > size) {  \_size++;  return true;  } else {  return false;  }  }  /\*删除元素的包装\*/  bool erase (const K& keyin) { return \_chains[\_getPosByKey (keyin)].erase (keyin); }  /\*输出元素\*/  void output (std::ostream& out) const {  for (int i = 0; i < \_divisor; i++) {  if (\_chains[i].size () == 0) {  std::cout << "Chain" << i << ": nothing ";  } else {  std::cout << "Chain" << i << ":" << \_chains[i] << " ";  }  }  }  /\*通过关键词获取长度，为了OJ而增加\*/  int getLengthByKey (const K& keyin) const {  return \_chains[\_getPosByKey (keyin)].size ();  }  bool empty () const { return \_size == 0; }  int size () const { return \_size; }  };  /\*输出重载\*/  template <class K, class E>  std::ostream& operator<<(std::ostream& out, const myhashChains<K, E>& in) {  in.output (out);  return out;  }  文件2 hashchaintest.cpp  #include<iostream>  #include"hashchain.h"  using namespace std;  #define LGRAND(min,max) ((rand()%(max-min+(int)1))+(int)min )  int main () {  myhashChains<int,int> chain(961);  /\*随机获取50个不重复的数字\*/  for (int i = 0; i < 500;) {  mypair<int, int> temp (i, LGRAND (0, 50000));  if (chain.find (temp.data) == nullptr) {  chain.insert (temp);  i++;  }  }  cout << "find:\n";  for (int i = 0; i < 500;i++) {  cout <<"find:"<<i<<" "<< \*chain.find(i) << endl;  }  cout << "delete:\n";  for (int i = 0; i < 500; i++) {  chain.erase (i) ;  cout << "delete:"<<i<<"\n";  }  cout << "delete done\n";  cout << endl << endl << chain << endl << endl;  return 0;  }  文件3 hashtable.h  # pragma once  #include<iostream>  /\*对应各种数据类型的哈希函数，此处只写int的哈希函数\*/  template <class K> class myhash;  template<>  class myhash<int> {  public:  int operator()(const int theKey) const {  return int (theKey);  }  };  /\*用于存储键值的类\*/  template <class K, class E>  class mypair {  public:  K key;  E data;  mypair (K keyin, E datain) :key (keyin), data (datain) {}  };  /\*template<class E, class K>  class myhashTable {  public:  struct pair;  protected:  myhash<K> myhashf;  int \_divisor; //散列函数的除数  pair\*\* \_table\_head; //散列数组  int \_getPosByKey (const K& keyin) const; //返回下标，不含有返回-1  public:  myhashTable (int divisorin = 20); //  int insertGetPos (const K& keyin, const E& datain);//元素存在则返回-1，不出在则返回插入后的下标  int deleteGetNum (const K& keyin); //通过关键词删除键值对，返回移动次数或-1\*  friend std::ostream& operator << (std::ostream& out, myhashTable& in)  int getPosByKey (const K& keyin) const; //通过关键词获取该元素的理论位置  \*/  template<class E, class K>  class myhashTable {  //public:  // typedef struct mypair<K,E> {  // K key;  // E data;  // mypair<K,E> (K keyin, E datain) :key (keyin), data (datain) {}  // }mypair<K,E>;  protected:  myhash<K> myhashf;  int \_divisor; //散列函数的除数  mypair<K,E>\*\* \_table\_head; //散列数组  /\*返回下标，不含有返回-1\*/  int \_getPosByKey (const K& keyin) const {  int i = keyin % \_divisor; //找 k对应散列表的位置  int j = i;  do {  if (!(\_table\_head[j] != nullptr && \_table\_head[j]->key != keyin)) return j; //找到或者为空，则返回  j = (j + 1) % \_divisor;  } while (j != i); //若又回到i，则退出循环  return j;  }  public:  int getPosByKey (const K& keyin) const {  int pos = \_getPosByKey (keyin);  if (\_table\_head[pos] == nullptr) {  return -1;  } else {  return pos;  }  }  E\* find (K& keyin) {  int temp = getPosByKey (keyin);  if (temp== -1) {  return nullptr;  } else {  return &(\_table\_head[temp]->data);  }  }  myhashTable (int divisorin = 20) :\_divisor (divisorin) {  \_table\_head = new mypair<K,E> \* [\_divisor];  for (int i = 0; i < \_divisor; i++) {  \_table\_head[i] = nullptr;  }  }  ~myhashTable () {  for (int i = 0; i < \_divisor; i++) {  }  delete[] \_table\_head;  }  int erase (const K& keyin) {  return deleteGetNum (keyin);  }  /\*通过关键词删除键值对，返回移动次数或-1\*/  int deleteGetNum (const K& keyin) {  int delete\_pos = \_getPosByKey (keyin);  if (\_table\_head[delete\_pos] == nullptr) {  return -1;  } else if (\_table\_head[delete\_pos]->key == keyin) {  \_table\_head[delete\_pos] = nullptr;//内存溢出  int current\_pos = delete\_pos;//当前元素当前的实际位置  int last\_pos = delete\_pos;//  int original\_pos;//当前元素一般情况下本应该在的位置  int move\_count = 0;  for (;;) {  current\_pos++;//将下一个元素设置为当前元素  current\_pos = current\_pos % \_divisor; //当前元素的实际位置  if (\_table\_head[current\_pos] == nullptr) { //当前桶为空或，为空说明前面的元素都在正确位置，直接删除并返回  return move\_count;  } else {  original\_pos = myhashf (\_table\_head[current\_pos]->key) % \_divisor;//当前元素本应该在哪里  if (current\_pos != \_getPosByKey (\_table\_head[current\_pos]->key)) {  \_table\_head[last\_pos] = \_table\_head[current\_pos];//移动指针，无需delete  \_table\_head[current\_pos] = nullptr;//内存溢出  move\_count++;  last\_pos = current\_pos;  } else {  continue;  }  }  };  }  }  /\*元素存在则返回-1，不出在则返回插入后的下标\*/  int insertGetPos (const K& keyin, const E& datain) {  int b = \_getPosByKey (keyin);  if (\_table\_head[b] == nullptr) {  \_table\_head[b] = new mypair<K,E> (keyin, datain);  return b;  } else {  return -1;  }  }  int insert (mypair<K,E>& pairin) {  return insertGetPos (pairin.key, pairin.data);  }  friend std::ostream& operator << (std::ostream& out, myhashTable& in) {  for (int i = 0; i < in.\_divisor; i++) {  if (in.\_table\_head[i] == nullptr) {  out << "null" << " ";  } else {  out << in.\_table\_head[i]->key << " ";  }  }  return out;  }  };  文件4 hashtable.cpp  #include<iostream>  #include"hashtable.h"  using namespace std;  #define LGRAND(min,max) ((rand()%(max-min+(int)1))+(int)min )  int main () {  myhashTable<int, int> table (961);  /\*随机获取50个不重复的数字\*/  for (int i = 0; i < 500;) {  mypair<int, int> temp (i, LGRAND (0, 50000));  if (table.find (temp.data) == nullptr) {  table.insert (temp);  i++;  }  }  cout << "find:\n";  for (int i = 0; i < 500; i++) {  cout << "find:" << i << " " << \*table.find (i) << endl;  }  cout << "delete:\n";  for (int i = 0; i < 500; i++) {  table.erase (i);  cout << "delete:" << i << "\n";  }  cout << "delete done\n";  cout << endl << endl << table << endl << endl;  return 0;  } | | | |