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：实验四 链表 | | | |
| 实验学时：2h | | 实验日期：2019.10.10 | |
| 实验目的：  1、掌握线性表结构、链式描述方法（链式存储结构）、链表的实现。  2、掌握链表迭代器的实现与应用。 | | | |
| 软件开发工具：  Virtual Studio 2019 | | | |
| 1. **实验内容** 2. 创建线性表类：存储结构使用单链表；提供操作:    1. 自表首插入元素    2. 删除指定元素    3. 搜索表中是否有指定元素    4. 输出链表   2、接收键盘录入的一系列整数作为节点的元素值，创建链表。输出链表内容。  3、输入一个整数，在链表中进行搜索，输出元素的索引。如果不存在输出-1。  4、设计实现链表迭代器，使用链表迭代器实现链表的反序输出。  5、创建两个有序链表，使用链表迭代器实现链表的合并。   1. **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. 本次实验建立chain.h头文件，编写chain类。   2. chain类使用泛型技术。   3. 枚举常见错误，如下标越界，并使用throw抛出错误，便于调试。   4. 提供增删改查，获取长度等功能。      1. 增：insert      2. 删：del      3. 改：重载“[]”运算符。      4. 查：find 成功则返回元素位置，失败则返回-1   5. 在[]、insert、del等函数中要有越界判断功能。   6. 实现单向迭代器，提供前置++、后置++，+n等功能。   7. 编写merge，reverse等较为高级的功能。   **数据结构：**本实验是链表实验，采用动态申请的node结构体作为链表结点。  **算法：**   * 1. 本实验较为关键的部分是指针与迭代器的操作，merge与reverse等复杂操作均需要合理地使用指针与迭代器。   2. 本实验中使用迭代器进行reverse输出的部分采用了递归思想。      1. 成员函数中设置父子函数      2. 父函数用于提供对外接口，并向子函数传参      3. 子函数实现递归，从而实现反向输出   3. merge函数考虑了对有序链表的操作      1. 对比两个源链表迭代器数值的大小，将大（或小）的元素添加到新链表中      2. 最终必然有一个链表未完全复制，在扫尾工作将其完全复制即可  1. **测试结果（测试输入，测试输出）**   **1.课堂检查：**  输出：  测试增:  613 451 286 366 798 856 438 239 720 39  isempty: 0  size: 10  测试改:  1226 902 572 732 1596 1712 876 478 1440 78  isempty: 0  size: 10  测试查:  data 78 in: 9  data 572 in: 2  data 573 in: -1  测试 iterator 反向输出:  78 1440 478 876 1712 1596 732 572 902 1226  测试删:  isempty: 1  size: 0  测试merge:  0 1 2 3 4 5 6 7 8 9  0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9  isempty: 0  size: 20  **2.平台提交**  输入：  **9**  **1 3**  **2 3**  **1 1**  **1 2**  **3 4**  **3 2**  **4**  **5**  **6**  **3 1 3 5**  **4 4 7 8 10**  输出：  **-1**  **0**  **2 1**  **1 2**  **1 3 4 5 7 8 10**  结果正确   1. **分析与探讨（结果分析，若存在问题，探讨解决问题的途径）**   实验结果正确，在实验过程中遇到了一些问题，学习到了若干新的细节：   * 1. 模板类重命名不能用typedef，应该用： template <class T> using arrayList = vector<T>;   2. 链表的指针操作 在插入，删除，查找等问题中，注意边界的设置与识别，如：      1. insert的下标可以等于链表长度。      2. 迭代器遍历链表有 n 次取值与 n-1 次自增，编写代码时应注意区别。   3. 类中使用成员函数作为默认形参，则该函数应当为static函数。   4. VS可用 #pragma warning(disable:4996) 禁止不必要的安全性报错   5. 链表的指针操作非常灵活，在链表中常出现多个指针相互配合的情况，非常有趣。  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 链表头文件  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* chiain.cpp \*  \* Copyright (C) 2019 TriAlley lg139@139.com. \*  \* @brief 链表头文件 \*  \* @license GNU General Public License (GPL) \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma once  #include<iostream>  using std::endl;  using std::cout;  using std::ostream;  namespace TA {  template<class T>  class chain {  public:  typedef enum err { index\_out\_of\_range }err;  struct node {  T data;  node\* next;  node ( const T& data ) {  this->data = data;  this->next = nullptr;  }  node ( const T& data, node\* next ) {  this->data = data;  this->next = next;  }  };  /\*迭代器的实现\*/  class chainiterator {  public:  chainiterator ( node\* theNode = nullptr ) {  inode = theNode;  }  // dereferencing operators  T& operator\*() const { return inode->data; }  T\* operator->() const { return &inode->data; }  chainiterator& operator++() {  inode = inode->next;  return \*this;  }  chainiterator operator++( int ) {  chainiterator old = \*this;  inode = inode->next;  return old;  }  chainiterator operator+( int i ) {  for ( int j = 0; j < i; j++ ) inode = inode->next;  return \*this;  }  bool operator!=( const chainiterator right ) const {  return inode != right.inode;  }  bool operator==( const chainiterator right ) const {  return inode == right.inode;  }  protected:  node\* inode;  };  chain () {  head = nullptr;  size = 0;  }  chain ( const chain<T>& in\_list ) {  size = in\_list.size;  if ( size == 0 ) {  head = nullptr;  return;  }  /\*复制的准备部分\*/  node\* source\_node = in\_list.head;  head = new node ( source\_node->data );  source\_node = source\_node->next;  node\* target\_node = head;  /\*复制的主体部分\*/  while ( source\_node != nullptr ) {  target\_node->next = new node ( source\_node->data );  target\_node = target\_node->next;  source\_node = source\_node->next;  }  /\*复制的收尾部分\*/  target\_node->next = nullptr;  }  ~chain () {  node\* nextNode;  while ( head != nullptr ) {  nextNode = head->next;  delete head;  head = nextNode;  }  }  void clear () {  node\* nextNode;  while ( head != nullptr ) {  nextNode = head->next;  delete head;  head = nextNode;  size--;  }  }  /\*增删改查部分\*/  void insert ( int index, const T& target ) {  /\*插入下标允许与size相同，故此处不使用统一的下标检查函数\*/  if ( index < 0 || index > size ) {  throw index\_out\_of\_range;  } else if ( index == 0 ) {//若插入到0位置，则直接建立新的结点，将原head设置为新head的next  head = new node ( target, head );  } else {//若插入到其他位置，则循环到所需位置再建立新结点  node\* p = head;  for ( int i = 0; i < index - 1; i++ ) {  p = p->next;  }  p->next = new node ( target, p->next );  }  size++;  }  void del ( int index ) {  \_checkIndex ( index );  node\* delete\_node;//待删除结点的指针  if ( index == 0 ) {//若删除0位。则建立新head并直接删除旧head  delete\_node = head;  head = head->next;  } else {//若删除其它位置，则循环找到所需元素并删除  node\* p = head;  for ( int i = 0; i < index - 1; i++ ) {  p = p->next;  }  delete\_node = p->next;  p->next = p->next->next;  }  size--;  delete delete\_node;  }  /\*重载中括号，达到“改”的目的\*/  T& operator []( int index ) const {  \_checkIndex ( index );  node\* current\_node = head;  for ( int i = 0; i < index; i++ )  current\_node = current\_node->next;  return current\_node->data;  }  /\*返回目的元素的下标，找不到则返回-1\*/  int find ( const T& target ) const {  node\* current\_node = head;  int index = 0;  while ( current\_node != nullptr &&  current\_node->data != target ) {  current\_node = current\_node->next;  index++;  }  if ( current\_node == nullptr ) {  return -1;  } else {  return index;  }  }  /\*返回链表是否为空\*/  bool empty () const { return size == 0; }  /\*返回链表长度\*/  int listSize () const { return size; }  /\*返回指向链表头的迭代器\*/  chainiterator begin () { return chainiterator ( head ); }  /\*返回指向链表尾的迭代器（此处为nullptr）\*/  chainiterator end () { return chainiterator ( nullptr ); }  /\*将链表翻转-使用迭代器\*/  /\*父函数，向子函数提供参数\*/  void reverse () {  \_reverse ( begin (), end () );  cout << endl;  }  void \_reverse ( chainiterator head, chainiterator end ) {  if ( head != end ) {  chain<int>::chainiterator temp = head;  \_reverse ( ++head, end );  cout << \*temp << " ";  }  }  /\*  \* brief：将两个输入的链表合并  \* in:  \* chain<int>& a：输入的链表之一  \* chain<int>& b：输入的链表之一  \* out:  \* chain<int>& ：返回函数调用者的引用  \*/  chain<int>& merge ( chain<int>& a, chain<int>& b ) {  clear ();  /\*合并的主体部分\*/  chain<int>::chainiterator ai = a.begin ();  chain<int>::chainiterator bi = b.begin ();  while ( ( ai != a.end () ) && ( bi != a.end () ) ) {  if ( \*ai <= \*bi ) {  insert ( listSize (), \*ai );  ai++;  } else {  insert ( listSize (), \*bi );  bi++;  }  }  //如果index等于listsize，说明该数组移动完毕；否则未复制完毕，继续复制剩余元素。  //a，b线性表必只剩一个没有复制完毕，因此此处两个判断只有一个能执行  if ( ai != a.end () ) {  while ( ai != a.end () ) {  insert ( listSize (), \*ai );  ai++;  }  }  if ( bi != a.end () ) {  while ( bi != a.end () ) {  insert ( listSize (), \*bi );  bi++;  }  }  return \*this;  }  /\*  \* brief：将链表内容输出到流  \* in:  \* ostream& out ：需要输出到的流的引用  \* const chain& c ：需要被输出的链表的引用  \* out:  \* ostream& 流的引用  \*/  friend ostream& operator<<( ostream& out, const chain& c ) {  node\* current\_node = c.head;  for ( int i = 0; i < c.size; i++ ) {  out << current\_node->data << " ";  current\_node = current\_node->next;  }  return out;  }  protected:  /\*检查下标是否越界，越界则 throw index\_out\_of\_range\*/  void \_checkIndex ( int index ) const {  if ( index < 0 || index >= size ) {  throw index\_out\_of\_range;  }  }  node\* head; //链表头指针  int size; //链表长度  };  }  文件2 链表测试  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* main.cpp \*  \* Copyright (C) 2019 TriAlley lg139@139.com. \*  \* @brief 链表测试 \*  \* @license GNU General Public License (GPL) \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include"chain.h"  #include<iostream>  using namespace std;  using namespace TA;  #define LGRAND(min,max) ((rand()%(max-min+(int)1))+(int)min )  chain<int>\* merge( chain<int>& a, chain<int>& b) {  /\*建立新链表\*/  chain<int>\* temp = new chain<int>;  chain<int>::chainiterator ai = a.begin();  chain<int>::chainiterator bi = b.begin();  while ((ai!=a.end()) && (bi != a.end())) {  if (\*ai>= \*bi) {  temp->insert(temp->listSize(),\*ai);  ai++;  } else {  temp->insert(temp->listSize(), \*bi);  bi++;  }  }  //如果index等于listsize，说明该数组移动完毕；否则未复制完毕，继续复制剩余元素。  //a，b线性表必只剩一个没有复制完毕，因此此处两个判断只有一个能执行  if (ai != a.end()) {  while (ai != a.end()) {  temp->insert(temp->listSize(), \*ai);  ai++;  }  }  if (bi != a.end()) {  while (bi != a.end()) {  temp->insert(temp->listSize(), \*bi);  bi++;  }  }  return temp;  }  int main() {  srand(0);  int num = 10;  chain<int> c;  cout << "测试增:\n";  for (int i = 0; i < num; i++) {  c.insert(0, (int)LGRAND(1, 1000));  }  cout << c << endl  <<"isempty: "<<c.empty()<<endl  <<"size: "<<c.listSize() <<endl;  cout << "\n测试改:\n";  for (int i = 0; i < 10; i++) {  c[i] = c[i] \* 2;  }  cout << c << endl  << "isempty: " << c.empty() << endl  << "size: " << c.listSize() << endl;  cout << "\n测试查:\n";  cout << "data 78" << " in: " << c.find(78) << endl;  cout << "data 572" << " in: " << c.find(572) << endl;  cout<<"data 573"<<" in: "<<c.find(578)<<endl;  cout << "\n测试 iterator 反向输出:\n";  c.reverse ();  cout << "\n测试删:\n";  for (int i = 0; i < 10; i++) {  c.del(0);  }  cout << c << endl  << "isempty: " << c.empty() << endl  << "size: " << c.listSize() << endl;  cout << "\n测试merge:\n";  chain<int> a;  for (int i = 0; i < num; i++) {  a.insert(0, i);  }  cout << a << endl  << "isempty: " << a.empty() << endl  << "size: " << a.listSize() << endl;  chain<int>\* temp = merge(a, a);  cout << \*temp << endl  << "isempty: " << temp->empty() << endl  << "size: " << temp->listSize() << endl;  delete temp;  } | | | |