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：201705130113 | 姓名： 黄瑞哲 | | 班级：计科17.3 |
| 实验题目：队列 | | | |
| 实验学时：2 | | 实验日期：2018.11.15 | |
| 实验目的：  1、掌握队列结构的定义与实现；  2、掌握队列结构的使用。 | | | |
| 软件环境：  Visual Studio Community 2017 | | | |
| 1. 实验内容（题目内容，输入要求，输出要求）   （1）、创建队列类，采用链式描述；  （2）、实现卡片游戏 。假设桌上有一叠扑克牌，依次编号为 1-n（从最上面开始）。当至少还有两张的时候，可以进行操作：把第一张牌扔掉，然后把新的第一张放到整叠牌的最后。输入 n，输出每次要扔掉的牌，以及最后剩下的牌。   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）   建立队列，按顺序依次插入1-n个数，当队列长度大于1时每次输出队首元素并出队，再将队首元素插入队尾并出队，最后再输出队首元素。   1. 测试结果（测试输入，测试输出，结果分析）   测试输入：n=7  测试输出：  throw: 1  throw: 3  throw: 5  throw: 7  throw: 4  throw: 2  remain:6  结果：与手动模拟结果一致，符合题目要求。   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   对于链表描述的队列，如果front指向链表头，back指向链表尾，则所有操作的复杂度为O(1),但如果front指向了链表尾而back指向了链表头，则出队操作的复杂度变成了O(n),但如果链表为双向链表则复杂度不变。   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）   /\*exe7.cpp\*/  #include "pch.h"  #include <iostream>  using namespace std;  int main()  {  int n;  linkedQueue<int>q;  cin >> n;  for (int i = 1; i <= n; ++i) q.push(i);  while (q.size() > 1)  {  cout << "throw: " << q.front() << endl;  q.pop();  q.push(q.front());  q.pop();  }  cout << "remain: " << endl;  while (!q.empty())  {  cout << q.front() << endl;  q.pop();  }  return 0;  }  /\*pch.h\*/  #ifndef PCH\_H  #define PCH\_H  // TODO: 添加要在此处预编译的标头  #include <queue/linkedQueue.h>  #endif //PCH\_H  /\*linkedQueue.h\*/  #ifndef \_\_linkedQueue\_H\_  #define \_\_linkedQueue\_H\_  #include "queue.h"  #include <linearlist/chain.h>  #include <stdexcept>  using namespace std;  template<typename T>  class linkedQueue:public queue<T>  {  public:  linkedQueue() :queueFront(nullptr), queueBack(nullptr), queueSize(0) {};  linkedQueue(const linkedQueue<T>&);  ~linkedQueue();  bool empty() const { return queueSize == 0; }  int size() const { return queueSize; }  T front() const;  T back() const;  void pop();  void push(const T&);  void clear();  linkedQueue<T>& operator=(const linkedQueue<T>&);  protected:  chainNode<T>\* queueFront;  chainNode<T>\* queueBack;  int queueSize;  };  template<typename T>  linkedQueue<T>::linkedQueue(const linkedQueue<T>& q)  {  if (q.empty())  {  queueFront = queueBack = nullptr;  queueSize = 0;  }  else  {  queueFront = queueBack = new chainNode<T>(q.queueFront->element);  chainNode<T>\* sourceNode = q.queueFront->next;  while (sourceNode != nullptr)  {  queueBack->next = new chainNode<T>(sourceNode->element);  queueBack = queueBack->next;  sourceNode = sourceNode->next;  }  queueSize = q.queueSize;  }  }  template<typename T>  linkedQueue<T>::~linkedQueue()  {  while (queueFront != nullptr)  {  chainNode<T>\* nextNode = queueFront->next;  delete queueFront;  queueFront = nextNode;  }  queueBack = nullptr;  }  template<typename T>  T linkedQueue<T>::front() const  {  if (queueSize < 1) throw out\_of\_range("the queue is empty");  return queueFront->element;  }  template<typename T>  T linkedQueue<T>::back() const  {  if (queueSize < 1) throw out\_of\_range("the queue is empty");  return queueBack->element;  }  template<typename T>  void linkedQueue<T>::pop()  {  if (queueSize < 1) throw out\_of\_range("the queue is empty");  chainNode<T>\* nextNode = queueFront->next;  delete queueFront;  queueFront = nextNode;  queueSize--;  }  template<typename T>  void linkedQueue<T>::push(const T& theElement)  {  chainNode<T>\* newNode = new chainNode<T>(theElement);  if (queueSize == 0) queueFront = newNode;  else queueBack->next = newNode;  queueBack = newNode;  queueSize++;  }  template<typename T>  void linkedQueue<T>::clear()  {  while (queueFront != nullptr)  {  chainNode<T>\* nextNode = queueFront->next;  delete queueFront;  queueFront = nextNode;  }  queueBack = nullptr;  queueSize = 0;  }  template<typename T>  linkedQueue<T>& linkedQueue<T>::operator=(const linkedQueue<T>& q)  {  if (this == &q) return \*this;  clear();  if (q.empty())  {  queueFront = queueBack = nullptr;  queueSize = 0;  }  else  {  queueFront = queueBack = new chainNode<T>(q.queueFront->element);  chainNode<T>\* sourceNode = q.queueFront->next;  while (sourceNode != nullptr)  {  queueBack->next = new chainNode<T>(sourceNode->element);  queueBack = queueBack->next;  sourceNode = sourceNode->next;  }  queueSize = q.queueSize;  }  return \*this;  }  #endif //\_\_linkedQueue\_H\_  /\*queue.h\*/  #ifndef \_\_QUEUE\_H\_  #define \_\_QUEUE\_H\_  template<typename T>  class queue  {  public:  virtual ~queue() {};  virtual bool empty() const = 0;  virtual int size() const = 0;  virtual T front() const = 0;  virtual T back() const = 0;  virtual void pop() = 0;  virtual void push(const T&) = 0;  virtual void clear() = 0;  };  #endif //\_\_QUEUE\_H\_  /\*chain.h\*/  #ifndef \_\_CHAIN\_H\_  #define \_\_CHAIN\_H\_  #include "linearList.h"  #include <stdexcept>  using namespace std;  template<typename T>  struct chainNode  {  T element;  chainNode<T>\* next;  chainNode(const T& element, chainNode<T>\* next = nullptr)  {  this->element = element;  this->next = next;  }  chainNode(const chainNode<T>\*& c)  {  element = c->element;  next = nullptr;  }  };  template<typename T>  class chain  :virtual public linearList<T>  {  public:  chain(int = 10);  chain(const chain<T>&);  ~chain();  bool empty() const;  int size() const;  int find(const T&) const;  void erase(int);  void insert(int, const T&);  void clear();  void push\_back(const T&);  chain<T>& operator=(const chain<T>&);  T& operator[](int);  const T& operator[](int) const;  class iterator;  class const\_iterator;  iterator begin() { return iterator(pHead->next); }  iterator end() { return iterator(nullptr); }  const\_iterator begin() const { return const\_iterator(pHead->next); }  const\_iterator end() const { return const\_iterator(nullptr); }  class iterator  {  public:  typedef forward\_iterator\_tag iterator\_category;  typedef T value\_type;  typedef ptrdiff\_t difference\_type;  typedef T\* pointer;  typedef T& reference;  iterator(chainNode<T>\* theNode = nullptr) :node(theNode) {};  T& operator\*() { return node->element; }  T\* operator->() { return &node->element; }  iterator& operator++()  {  node = node->next;  return \*this;  }  iterator operator++(int)  {  iterator old = \*this;  node = node->next;  return old;  }  bool operator==(const iterator right) const { return node == right.node; }  bool operator!=(const iterator right) const { return node != right.node; }  protected:  chainNode<T>\* node;  };  class const\_iterator  {  public:  typedef forward\_iterator\_tag iterator\_category;  typedef T value\_type;  typedef ptrdiff\_t difference\_type;  typedef T\* pointer;  typedef T& reference;  const\_iterator(chainNode<T>\* theNode) :node(theNode) {};  const T& operator\*() { return node->element; }  const T\* operator->() { return &node->element; }  const\_iterator& operator++()  {  node = node->next;  return \*this;  }  const\_iterator operator++(int)  {  const\_iterator old = \*this;  node = node->next;  return old;  }  bool operator==(const const\_iterator right) const { return node == right.node; }  bool operator!=(const const\_iterator right) const { return node != right.node; }  protected:  chainNode<T>\* node;  };  protected:  chainNode<T>\* pHead;  chainNode<T>\* pTail;  int listSize;  void checkIndex(int) const;  };  template<typename T>  chain<T>::chain(int initialCapacity)  {  if (initialCapacity < 1) throw out\_of\_range("the initial Capacity of arrayList must > 0");  listSize = 0;  pHead = new chainNode<T>(0);  pTail = pHead;  }  template<typename T>  chain<T>::chain(const chain<T>& c)  {  pHead = new chainNode<T>(c.pHead->element);  pTail = pHead;  chainNode<T>\* sourceNode = c.pHead->next;  chainNode<T>\* currentNode = pHead;  while (sourceNode != nullptr)  {  pTail = currentNode->next = new chainNode<T>(sourceNode->element);  currentNode = currentNode->next;  sourceNode = sourceNode->next;  }  listSize = c.listSize;  }  template<typename T>  chain<T>::~chain()  {  chainNode<T>\* currentNode = pHead->next;  chainNode<T>\* deleteNode;  while (currentNode != nullptr)  {  deleteNode = currentNode;  currentNode = currentNode->next;  delete deleteNode;  }  delete pHead;  }  template<typename T>  bool chain<T>::empty() const { return listSize == 0; }  template<typename T>  int chain<T>::size() const { return listSize; }  template<typename T>  int chain<T>::find(const T& theElement) const  {  int index = 0;  chainNode<T>\* currentNode = pHead->next;  while (currentNode != nullptr)  {  if (currentNode->element == theElement) return index;  currentNode = currentNode->next;  ++index;  }  return -1;  }  template<typename T>  void chain<T>::erase(int theIndex)  {  checkIndex(theIndex);  chainNode<T>\* deleteNode;  chainNode<T>\* pre = pHead;  for (int i = 0; i < theIndex; ++i) pre = pre->next;  if (theIndex == listSize - 1) pTail = pre;  deleteNode = pre->next;  pre->next = pre->next->next;  --listSize;  delete deleteNode;  }  template<typename T>  void chain<T>::insert(int theIndex, const T& theElement)  {  if (theIndex < 0 || theIndex>listSize) throw out\_of\_range("illegalIndex");  chainNode<T>\* pre = pHead;  for (int i = 0; i < theIndex; ++i) pre = pre->next;  pre->next = new chainNode<T>(theElement, pre->next);  if (theIndex == listSize) pTail = pre->next;  ++listSize;  }  template<typename T>  void chain<T>::clear()  {  chainNode<T>\* currentNode = pHead->next;  chainNode<T>\* deleteNode;  while (currentNode != nullptr)  {  deleteNode = currentNode;  currentNode = currentNode->next;  delete deleteNode;  }  listSize = 0;  pHead->next = nullptr;  pTail = pHead;  }  template<typename T>  void chain<T>::push\_back(const T& theElement)  {  pTail->next = new chainNode<T>(theElement, pTail->next);  pTail = pTail->next;  listSize++;  }  template<typename T>  chain<T>& chain<T>::operator=(const chain<T>& c)  {  if (this == &c) return \*this;  clear();  chainNode<T>\* currentNode = pHead;  chainNode<T>\* sourceNode = c.pHead->next;  while (sourceNode != nullptr)  {  pTail = currentNode->next = new chainNode<T>(sourceNode->element);  currentNode = currentNode->next;  sourceNode = sourceNode->next;  }  listSize = c.listSize;  return \*this;  }  template<typename T>  T& chain<T>::operator[](int index)  {  checkIndex(index);  chainNode<T>\* currentNode = pHead->next;  for (int i = 0; i < index; ++i) currentNode = currentNode->next;  return currentNode->element;  }  template<typename T>  const T& chain<T>::operator[](int index) const  {  checkIndex(index);  chainNode<T>\* currentNode = pHead->next;  for (int i = 0; i < index; ++i) currentNode = currentNode->next;  return currentNode->element;  }  template<typename T>  void chain<T>::checkIndex(int theIndex) const  {  if (theIndex < 0 || theIndex >= listSize)  throw out\_of\_range("the index is out of range");  }  #endif //\_\_CHAIN\_H\_  /\*linearList.h\*/  #ifndef \_\_LINEARLIST\_H\_  #define \_\_LINEARLIST\_H\_  template<typename T>  class linearList  {  public:  virtual ~linearList() {};  virtual bool empty() const = 0;  virtual int size() const = 0;  virtual int find(const T&) const = 0;  virtual void erase(int) = 0;  virtual void insert(int, const T&) = 0;  virtual void clear() = 0;  virtual void push\_back(const T& ) = 0;  virtual T& operator[](int) = 0;  virtual const T& operator[](int) const = 0;  };  #endif //\_\_LINEARLIST\_H\_ | | | |