计算机科学技术学院实验报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **课程名称** | 面向对象程序设计 | | | **学 号** | 2405111B08 |
| **实验项目** | 实验三 - 为用链表实现的线性表添加赋值和拷贝 | | | **姓 名** | 王子睿 |
| **学 时** | 2 | **项目性质** | 综合型 | **班 级** | 2405111B |
| **指导教师** | 蒋振刚 | **实验地点** | 南实训424 | **日 期** | 2025年4月7日 |
| 1. **实验目的和要求**   实验目的：了解深拷贝和浅拷贝的区别。  实验要求：  在线性表类DList的基础之上，扩充：  1）拷贝构造函数；  2）赋值运算；  3）测试代码验证 以上的操作。 | | | | | |
| 1. **实验环境**   Visual Studio Code  MinGW-w64 | | | | | |
| 1. **实验内容与过程**   // 第3个实验：为用链表实现的线性表添加赋值和拷贝。  // 目的：了解深拷贝和浅拷贝的区别。  // 要求：在线性表类DList的基础之上，扩充：  // 1）拷贝构造函数；  // 2）赋值运算；  // 3）测试代码验证 以上的操作。  #include <iostream>  class DList  {  private:      struct Node      {          int data;          Node \*next;      };      Node \*head;      Node \*tail;      int currentSize;      bool isShallowCopy;  public:      DList()      {          head = nullptr;          tail = nullptr;          currentSize = 0;          isShallowCopy = false;      }      // Deep copy (copy constructor)      DList(const DList &list)      {          head = nullptr;          tail = nullptr;          currentSize = 0;          isShallowCopy = false;          Node \*current = list.head;          while (current != nullptr)          {              insert(currentSize, current->data);              current = current->next;          }      }      // Shallow copy (assignment operator)      DList &operator=(const DList &list) // NEW      {          if (this == &list)          {              return \*this; // Handle self-assignment          }          // Free existing resources          Node \*current = head;          while (current != nullptr)          {              Node \*temp = current;              current = current->next;              delete temp;          }          head = list.head;          tail = list.tail;          currentSize = list.currentSize;          isShallowCopy = true;          return \*this;      }      ~DList() // EDITED      {          // If it's a shallow copy, do not delete the nodes,          // They will be deleted by the original list's destructor.          if (!isShallowCopy)          {              Node \*current = head;              while (current != nullptr)              {                  Node \*temp = current;                  current = current->next;                  delete temp;              }          }      }      void insert(int index, int value)      {          // Range check          if (index < 0 || index > currentSize)          {              std::cout << "Index out of range." << std::endl;              return;          }          // Create a new node          Node \*newNode = new Node;          newNode->data = value;          // Insert at the beginning          if (index == 0)          {              newNode->next = head;              head = newNode;              if (currentSize == 0)              {                  tail = newNode;              }          }          // Insert at the end          else if (index == currentSize)          {              tail->next = newNode;              tail = newNode;              tail->next = nullptr;          }          // Insert in the middle          else          {              Node \*current = head;              for (int i = 0; i < index - 1; i++)              {                  current = current->next;              }              newNode->next = current->next;              current->next = newNode;          }          currentSize++;      }      void remove(int index)      {          // Range check          if (index < 0 || index >= currentSize)          {              std::cout << "Index out of range." << std::endl;              return;          }          // Remove the first node          if (index == 0)          {              Node \*temp = head;              head = head->next;              delete temp;              if (currentSize == 1)              {                  tail = nullptr;              }          }          // Remove the last node          else if (index == currentSize - 1)          {              Node \*current = head;              for (int i = 0; i < index - 1; i++)              {                  current = current->next;              }              Node \*temp = tail;              tail = current;              tail->next = nullptr;              delete temp;          }          // Remove in the middle          else          {              Node \*current = head;              for (int i = 0; i < index - 1; i++)              {                  current = current->next;              }              Node \*temp = current->next;              current->next = current->next->next;              delete temp;          }          currentSize--;      }      void display()      {          Node \*current = head;          while (current != nullptr)          {              std::cout << current->data << " ";              current = current->next;          }          std::cout << std::endl;      }      void reverse()      {          Node \*current = head;          Node \*prev = nullptr;          Node \*next = nullptr;          while (current != nullptr)          {              next = current->next;              current->next = prev;              prev = current;              current = next;          }          head = prev;      }      int &operator[](int index) const      {          // Range check          if (index < 0 || index >= currentSize)          {              std::cout << "Index out of range." << std::endl;          }          Node \*current = head;          for (int i = 0; i < index; i++)          {              current = current->next;          }          return current->data;      }      friend std::ostream &operator<<(std::ostream &os, const DList &list)      {          Node \*current = list.head;          while (current != nullptr)          {              os << current->data << " ";              current = current->next;          }          return os;      }      friend std::istream &operator>>(std::istream &is, DList &list)      {          int value;          while (is >> value)          {              list.insert(list.currentSize, value);          }          return is;      }      const int size() const      {          return currentSize;      }      const int find(int value) const      {          Node \*current = head;          for (int i = 0; i < currentSize; i++)          {              if (current->data == value)              {                  return i;              }              current = current->next;          }          return -1;      }  };  int main()  {      DList list;                     // Create a list with size 10      std::cin >> list;               // Input 10 integers      std::cout << list << std::endl; // Display the list      DList deepCopyList(list);       // Deep copy      std::cout << "Deep Copy: " << deepCopyList << std::endl;      DList shallowCopyList;      shallowCopyList = list; // Shallow copy      std::cout << "Shallow Copy: " << shallowCopyList << std::endl;      list[0] = 100; // Modify the original list      std::cout << "After modifying original list:" << std::endl;      std::cout << "Original List: " << list << std::endl;      std::cout << "Deep Copy: " << deepCopyList << std::endl;      std::cout << "Shallow Copy: " << shallowCopyList << std::endl;      return 0;  }  3.drawio | | | | | |
| 1. **实验结果与分析**   001  主函数创建了一个容量为10的动态线性表，然后依次测试了从cin中输入数据、输出数据到cout、创建表的深拷贝、创建表的浅拷贝、更改原表的第一个元素、测试深拷贝、测试浅拷贝。在创建浅拷贝列表的时候，需要将isShallowCopy更改为否，以避免析构时尝试删除已经被原表释放的实体。 | | | | | |
| 1. **实验心得**   通过深拷贝浅拷贝的测试，我进一步理解了数据在内存中存储的工作原理及其在应用中的重要性。特别是在赋值等常见的拷贝操作中，需要区分深拷贝和浅拷贝，避免引用更改后的元素，从而出现不必要的错误，让我对数据结构的存储有了更深入的认识。在编写析构函数时，执行到delete语句时抛出了Unknown signal错误。通过排查分析，我发现了原表执行完毕析构函数后，其浅拷贝对象仍然在析构函数中尝试释放节点。为了区分浅拷贝对象，避免浅拷贝对象执行析构函数，我新建了一个isShallowCopy成员，将浅拷贝对象的该成员置为假，在析构函数的条件判断时跳过释放部分。 | | | | | |
| 1. **教师评语** | | | | | |
| 1. **实验成绩**   教师签名： 蒋振刚 批阅日期： 2025 年 4 月 12 日 | | | | | |

注：项目性质为 演示型、验证型、设计型、综合型和创新型。