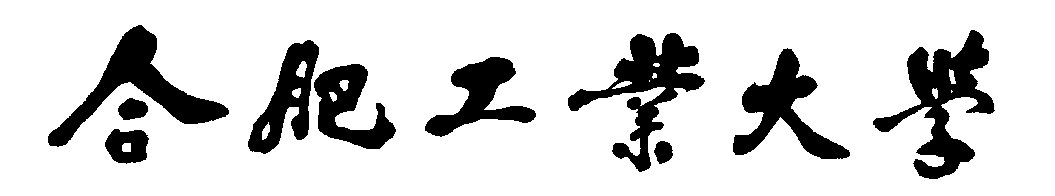
****

计算机与信息学院

数据结构实验报告

|  |  |
| --- | --- |
| 专 业 班 级 | 物联网一班 |
| 学生姓名及学号 | 敬成超 2023212388 |
| 课程教学班号 |  |
| 任 课 教 师 | 胡学钢 |
| 实验指导教师 |  |
| 实验地点 | C栋304 |
| 2023 ~2024 学年第二学期 | |

实验序号及名称：实验 一 单链表实验

实验时间∶ 3月 20

|  |
| --- |
| 预习内容 |
| 一、实验目的和要求∶  （1）理解线性表的链式存储结构。  （2）熟练掌握动态链表结构及有关算法的设计。  （3）根据具体问题的需要，设计出合理的表示数据的链  表结构，并设计相关算法。 |
| 二、实验任务∶  本次实验中的链表结构均为**带头结点的单链表**。  **要求：在实现带头结点的单链表的类基础上，完成下列内容**：  **设计算法实现下列问题的求解。**  **<1>求链表中第i个结点的指针（函数），若不存在，则返回NULL。**  实验测试数据基本要求：  第一组数据：链表长度n≥10，i分别为5，n，0，n+1，n+2  第二组数据：链表长度n=0，i分别为0，2  **<2>在第i个结点前插入值为x的结点。**  实验测试数据基本要求：  第一组数据：链表长度n≥10，x=100, i分别为5,n,n+1,0,1,n+2  第二组数据：链表长度n=0，x=100，i=5  **<3>删除链表中第i个元素结点。**  实验测试数据基本要求：  第一组数据：链表长度n≥10，i分别为5,n,1,n+1,0  第二组数据：链表长度n=0， i=5  **<4>在一个递增有序的链表L中插入一个值为x的元素，并保持其递增有序特性。**  实验测试数据基本要求：  链表元素为 （10,20,30,40,50,60,70,80,90,100）,  x分别为25，85，110和8  **<5>将单链表Ｌ中的奇数项和偶数项结点分解开，并分别连成一个带头结点的单链表，然后再将这两个新链表同时输出在屏幕上，并保留原链表的显示结果，以便对照求解结果。**  实验测试数据基本要求：  第一组数据：链表元素为 （1,2,3,4,5,6,7,8,9,10,20,30,40,50,60）  第二组数据：链表元素为 （10,20,30,40,50,60,70,80,90,100）  **求两个递增有序链表L1和L2中的公共元素，并以同样方式连接成链表L3。**  实验测试数据基本要求：  第一组  第一个链表元素为 （1, 3, 6, 10, 15, 16, 17, 18, 19, 20）  第二个链表元素为 （1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 20, 30）  第二组  第一个链表元素为 （1, 3, 6, 10, 15, 16, 17, 18, 19, 20）  第二个链表元素为 （2, 4, 5, 7, 8, 9, 12, 22）  第三组  第一个链表元素为 （）  第二个链表元素为 （1, 2, 3, 4, 5, 6, 7, 8, 9, 10） |
| 三、实验准备方案，包括以下内容：  （硬件类实验：实验原理、实验线路、设计方案等）  （软件类实验：所采用的核心方法、框架或流程图及程序清单）  **核心方法**   1. **Node类**：定义链表中的节点，通常包含数据和指向下一个节点的指针。 2. **LinkedList类**：定义单链表本身，包含指向头节点的指针和链表长度。 3. **插入操作**（Insert）：在链表的特定位置添加新节点。 4. **删除操作**（Delete）：从链表中移除特定节点。 5. **搜索操作**（Search）：查找链表中包含特定数据的节点。 6. **遍历操作**（Traverse）：从头部开始，按顺序访问链表中的每个节点。 7. **显示操作**（Display）：打印链表中的所有元素。   **框架**   1. **类的定义**：定义Node和LinkedList类。 2. **构造函数**：初始化链表，设置头节点为null，长度为0。 3. **成员函数**：实现插入、删除、搜索、遍历和显示等操作。 4. **辅助函数**：如获取链表长度、判断链表是否为空等。 |

|  |
| --- |
| 实验内容 |
| 一、实验用仪器、设备：电脑，配置了环境的visual studio 2022 |
| 二、实验内容与步骤（过程及数据记录）：   1. **设计链表节点**：首先设计一个链表节点类或结构体，通常包含数据部分和指向下一个节点的指针。 2. **实现链表类**：创建一个链表类，包含指向头节点的指针和链表的长度等属性。 3. **初始化链表**：实现一个构造函数，用于初始化链表，设置头节点为null，并初始化链表长度为0。 4. **添加节点**：   **头部添加**：实现一个方法，将新节点添加到链表头部。  **尾部添加**：实现一个方法，将新节点添加到链表尾部。这通常需要遍历整个链表找到最后一个节点。   1. **删除节点**：   **按值删除**：实现一个方法，根据节点的值删除链表中的节点。  **按位置删除**：实现一个方法，根据节点的位置（索引）删除节点。   1. **查找节点**：实现一个方法，根据节点的值或位置查找节点。 2. **显示链表**：实现一个方法，打印出链表中的所有节点数据，以便于观察链表的状态。 3. **更新节点**：实现一个方法，根据节点的位置或值更新节点的数据。 4. **合并链表**：如果需要，实现一个方法，将两个有序链表合并为一个有序链表。 5. **测试链表功能**：编写测试用例，测试链表的各种操作，确保功能正确实现。   #include <iostream>  #include <memory>  // 定义链表结点结构  struct Node {  int data;  std::shared\_ptr<Node> next;  Node(int val) : data(val), next(nullptr) {}  };  // 获取链表中第i个结点的指针  std::shared\_ptr<Node> getNodeAtPosition(const std::shared\_ptr<Node>& head, int i) {  if (i < 0) {  return nullptr;  }  std::shared\_ptr<Node> current = head;  int count = 0;  while (current != nullptr) {  if (count == i) {  return current;  }  count++;  current = current->next;  }  return nullptr;  }  // 打印链表  void printList(const std::shared\_ptr<Node>& head) {  std::shared\_ptr<Node> current = head;  while (current != nullptr) {  std::cout << current->data << " -> ";  current = current->next;  }  std::cout << "NULL" << std::endl;  }  int main() {  // 创建链表  auto head = std::make\_shared<Node>(1);  auto current = head;  for (int i = 2; i <= 10; i++) {  current->next = std::make\_shared<Node>(i);  current = current->next;  }  // 测试第一组数据  int testIndices1[] = { 5, 10, 0, 11, 12 };  for (int i = 0; i < 5; i++) {  auto result = getNodeAtPosition(head, testIndices1[i]);  if (result != nullptr) {  std::cout << "Node at position " << testIndices1[i] << ": " << result->data << std::endl;  }  else {  std::cout << "Node at position " << testIndices1[i] << ": NULL" << std::endl;  }  }  // 创建空链表  std::shared\_ptr<Node> emptyHead = nullptr;  // 测试第二组数据  int testIndices2[] = { 0, 2 };  for (int i = 0; i < 2; i++) {  auto result = getNodeAtPosition(emptyHead, testIndices2[i]);  if (result != nullptr) {  std::cout << "Node at position " << testIndices2[i] << ": " << result->data << std::endl;  }  else {  std::cout << "Node at position " << testIndices2[i] << ": NULL" << std::endl;  }  }  return 0;  }  #include <iostream>  #include <memory>  // 定义链表结点结构  struct Node {  int data;  std::shared\_ptr<Node> next;  Node(int val) : data(val), next(nullptr) {}  };  // 在链表第i个结点前插入值为x的结点  void insertNodeBeforePosition(std::shared\_ptr<Node>& head, int i, int x) {  if (i < 0) {  std::cout << "Invalid position: " << i << std::endl;  return;  }  auto newNode = std::make\_shared<Node>(x);  if (i == 0) {  newNode->next = head;  head = newNode;  return;  }  std::shared\_ptr<Node> current = head;  int count = 0;  while (current != nullptr && count < i - 1) {  current = current->next;  count++;  }  if (current == nullptr) {  std::cout << "Invalid position: " << i << std::endl;  return;  }  newNode->next = current->next;  current->next = newNode;  }  // 打印链表  void printList(const std::shared\_ptr<Node>& head) {  std::shared\_ptr<Node> current = head;  while (current != nullptr) {  std::cout << current->data << " -> ";  current = current->next;  }  std::cout << "NULL" << std::endl;  }  int main() {  // 创建链表  auto head = std::make\_shared<Node>(1);  auto current = head;  for (int i = 2; i <= 10; i++) {  current->next = std::make\_shared<Node>(i);  current = current->next;  }  // 测试第一组数据  int testIndices1[] = { 5, 10, 11, 0, 1, 12 };  int x = 100;  for (int i = 0; i < 6; i++) {  insertNodeBeforePosition(head, testIndices1[i], x);  std::cout << "After inserting " << x << " before position " << testIndices1[i] << ": ";  printList(head);  }  // 创建空链表  std::shared\_ptr<Node> emptyHead = nullptr;  // 测试第二组数据  int testIndex2 = 5;  insertNodeBeforePosition(emptyHead, testIndex2, x);  std::cout << "After inserting " << x << " before position " << testIndex2 << ": ";  printList(emptyHead);  return 0;  }  #include <iostream>  #include <memory>  // 定义链表结点结构  struct Node {  int data;  std::shared\_ptr<Node> next;  Node(int val) : data(val), next(nullptr) {}  };  // 删除链表中第i个元素结点  void deleteNodeAtPosition(std::shared\_ptr<Node>& head, int i) {  if (i < 0) {  std::cout << "Invalid position: " << i << std::endl;  return;  }  if (head == nullptr) {  std::cout << "List is empty, cannot delete from empty list." << std::endl;  return;  }  if (i == 0) {  head = head->next;  return;  }  std::shared\_ptr<Node> current = head;  int count = 0;  while (current != nullptr && count < i - 1) {  current = current->next;  count++;  }  if (current == nullptr || current->next == nullptr) {  std::cout << "Invalid position: " << i << std::endl;  return;  }  current->next = current->next->next;  }  // 打印链表  void printList(const std::shared\_ptr<Node>& head) {  std::shared\_ptr<Node> current = head;  while (current != nullptr) {  std::cout << current->data << " -> ";  current = current->next;  }  std::cout << "NULL" << std::endl;  }  int main() {  // 创建链表  auto head = std::make\_shared<Node>(1);  auto current = head;  for (int i = 2; i <= 10; i++) {  current->next = std::make\_shared<Node>(i);  current = current->next;  }  // 测试第一组数据  int testIndices1[] = {5, 10, 1, 11, 0};  for (int i = 0; i < 5; i++) {  deleteNodeAtPosition(head, testIndices1[i]);  std::cout << "After deleting node at position " << testIndices1[i] << ": ";  printList(head);  }  // 创建空链表  std::shared\_ptr<Node> emptyHead = nullptr;  // 测试第二组数据  int testIndex2 = 5;  deleteNodeAtPosition(emptyHead, testIndex2);  std::cout << "After deleting node at position " << testIndex2 << ": ";  printList(emptyHead);  return 0;  }  #include <iostream>  struct ListNode {  int val;  ListNode\* next;  ListNode(int x) : val(x), next(nullptr) {}  };  void insertSorted(ListNode\*& head, int x) {  ListNode\* newNode = new ListNode(x);  // 如果链表为空或者新节点应该插入在链表头部  if (head == nullptr || head->val >= x) {  newNode->next = head;  head = newNode;  return;  }  // 找到插入位置  ListNode\* current = head;  while (current->next != nullptr && current->next->val < x) {  current = current->next;  }  // 插入新节点  newNode->next = current->next;  current->next = newNode;  }  void printList(ListNode\* head) {  ListNode\* current = head;  while (current != nullptr) {  std::cout << current->val << " ";  current = current->next;  }  std::cout << std::endl;  }  int main() {  // 创建初始链表  ListNode\* head = new ListNode(10);  ListNode\* current = head;  int values[] = { 20, 30, 40, 50, 60, 70, 80, 90, 100 };  for (int val : values) {  current->next = new ListNode(val);  current = current->next;  }  // 测试插入  int testValues[] = { 25, 85, 110, 8 };  for (int x : testValues) {  insertSorted(head, x);  printList(head);  }  // 释放内存  while (head != nullptr) {  ListNode\* temp = head;  head = head->next;  delete temp;  }  return 0;  }  #include <iostream>  using namespace std;  // 定义链表节点结构  struct ListNode {  int val;  ListNode\* next;  ListNode(int x) : val(x), next(nullptr) {}  };  // 创建链表  ListNode\* createList(const int arr[], int n) {  if (n == 0) return nullptr;  ListNode\* head = new ListNode(arr[0]);  ListNode\* current = head;  for (int i = 1; i < n; ++i) {  current->next = new ListNode(arr[i]);  current = current->next;  }  return head;  }  // 打印链表  void printList(ListNode\* head) {  ListNode\* current = head;  while (current != nullptr) {  cout << current->val << " ";  current = current->next;  }  cout << endl;  }  // 分解链表为奇数项和偶数项  void splitList(ListNode\* head, ListNode\*& oddHead, ListNode\*& evenHead) {  if (head == nullptr) return;  oddHead = new ListNode(0); // 带头结点的单链表  evenHead = new ListNode(0); // 带头结点的单链表  ListNode\* oddTail = oddHead;  ListNode\* evenTail = evenHead;  ListNode\* current = head;  int index = 1;  while (current != nullptr) {  if (index % 2 != 0) {  oddTail->next = current;  oddTail = oddTail->next;  }  else {  evenTail->next = current;  evenTail = evenTail->next;  }  current = current->next;  index++;  }  oddTail->next = nullptr;  evenTail->next = nullptr;  // 去掉头结点  oddHead = oddHead->next;  evenHead = evenHead->next;  }  int main() {  // 第一组数据  int arr1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60 };  ListNode\* head1 = createList(arr1, sizeof(arr1) / sizeof(arr1[0]));  cout << "原链表1: ";  printList(head1);  ListNode\* oddHead1 = nullptr;  ListNode\* evenHead1 = nullptr;  splitList(head1, oddHead1, evenHead1);  cout << "奇数项链表1: ";  printList(oddHead1);  cout << "偶数项链表1: ";  printList(evenHead1);  // 第二组数据  int arr2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };  ListNode\* head2 = createList(arr2, sizeof(arr2) / sizeof(arr2[0]));  cout << "原链表2: ";  printList(head2);  ListNode\* oddHead2 = nullptr;  ListNode\* evenHead2 = nullptr;  splitList(head2, oddHead2, evenHead2);  cout << "奇数项链表2: ";  printList(oddHead2);  cout << "偶数项链表2: ";  printList(evenHead2);  return 0;  }  #include <iostream>  using namespace std;  // 定义链表节点结构  struct ListNode {  int val;  ListNode\* next;  ListNode(int x) : val(x), next(nullptr) {}  };  // 创建链表  ListNode\* createList(const int arr[], int n) {  if (n == 0) return nullptr;  ListNode\* head = new ListNode(arr[0]);  ListNode\* current = head;  for (int i = 1; i < n; ++i) {  current->next = new ListNode(arr[i]);  current = current->next;  }  return head;  }  // 打印链表  void printList(ListNode\* head) {  ListNode\* current = head;  while (current != nullptr) {  cout << current->val << " ";  current = current->next;  }  cout << endl;  }  // 找到两个递增有序链表的公共元素，并连接成链表L3  ListNode\* findCommonElements(ListNode\* L1, ListNode\* L2) {  ListNode dummy(0);  ListNode\* tail = &dummy;  ListNode\* p1 = L1;  ListNode\* p2 = L2;  while (p1 != nullptr && p2 != nullptr) {  if (p1->val == p2->val) {  tail->next = new ListNode(p1->val);  tail = tail->next;  p1 = p1->next;  p2 = p2->next;  }  else if (p1->val < p2->val) {  p1 = p1->next;  }  else {  p2 = p2->next;  }  }  return dummy.next;  }  int main() {  // 第一组数据  int arr1[] = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };  int arr2[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 20, 30 };  ListNode\* L1 = createList(arr1, sizeof(arr1) / sizeof(arr1[0]));  ListNode\* L2 = createList(arr2, sizeof(arr2) / sizeof(arr2[0]));  cout << "第一个链表: ";  printList(L1);  cout << "第二个链表: ";  printList(L2);  ListNode\* L3 = findCommonElements(L1, L2);  cout << "公共元素链表: ";  printList(L3);  // 第二组数据  int arr3[] = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };  int arr4[] = { 2, 4, 5, 7, 8, 9, 12, 22 };  ListNode\* L4 = createList(arr3, sizeof(arr3) / sizeof(arr3[0]));  ListNode\* L5 = createList(arr4, sizeof(arr4) / sizeof(arr4[0]));  cout << "第一个链表: ";  printList(L4);  cout << "第二个链表: ";  printList(L5);  ListNode\* L6 = findCommonElements(L4, L5);  cout << "公共元素链表: ";  printList(L6);  // 第三组数据  int arr5[10] = {};  int arr6[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  ListNode\* L7 = createList(arr5, sizeof(arr5) / sizeof(arr5[0]));  ListNode\* L8 = createList(arr6, sizeof(arr6) / sizeof(arr6[0]));  cout << "第一个链表: ";  printList(L7);  cout << "第二个链表: ";  printList(L8);  ListNode\* L9 = findCommonElements(L7, L8);  cout << "公共元素链表: ";  printList(L9);  return 0;  } |

|  |
| --- |
|  |
| 三、实验结果分析、思考题解答∶  实验结果再预期内，没有思考题 |
| 四、感想、体会、建议∶  感想与体会   1. **数据结构的重要性**：单链表是数据结构中的基础内容，通过实验可以深刻理解数据结构在计算机科学中的重要性。 2. **指针的使用**：单链表的操作涉及大量的指针操作，这有助于加深对指针的理解和使用技巧。 3. **编程实践的价值**：通过实际编写代码来实现单链表的创建、插入、删除等操作，可以提高编程能力和问题解决能力。 4. **调试技巧的提升**：在实现单链表的过程中，可能会遇到各种错误，这需要耐心调试，从而提升调试技巧。   建议   1. **理解基本概念**：在进行实验之前，确保对单链表的基本概念（如节点、指针、头节点等）有清晰的理解。 2. **逐步实现**：不要试图一次性实现所有功能，而是逐步进行，先实现基本的创建和遍历，再逐步添加插入、删除等功能。 3. **注重代码的可读性**：编写清晰、结构化的代码，使用有意义的变量名和函数名，便于自己和他人阅读和理解。 4. **测试用例的设计**：设计全面的测试用例，包括正常情况和边界情况，确保代码的正确性和鲁棒性。 5. **错误处理**：在实现插入、删除等操作时，要考虑各种可能的错误情况，并进行适当的错误处理。 6. **参考资料**：利用书籍、在线教程和论坛等资源，遇到问题时可以查找解决方案或寻求帮助。 7. **代码复用**：如果可能，将单链表的基本操作封装成函数或类，便于复用和维护。 |
| 实验成绩∶  指导教师签名：  年 月 日 |