**江苏科技大学**

**《数据结构》实验报告**

**（2023/2024学年第1学期）**

**学生姓名：**

**学生学号：**

**院 系： 计算机学院**

**专 业： 计算机科学与技术**

**考核得分：**

2023 年 12 月

**实验一 线性表的操作**

**一、实验目的**

掌握线性表的基本操作在存储结构上的实现，其中以单链表的操作作为重点。

**二、实验题目**

1.以**单链表**作为存储结构，实现线性表的就地逆置。

2. 创建一个**非递减序（有重复值）的单链表**，实现删除值相同的多余结点。

**三、实验预习的核心算法伪代码**

1.

**procedure reverse():**

**pre := null**

**head := first.next**

**if head == null:**

**return**

**nxt := head.next**

**while head != null:**

**head.next := pre**

**pre := head**

**head := nxt**

**if nxt != null:**

**nxt := nxt.next**

**first.next := pre**

**2.**

**procedure unique():**

**head := first.next**

**pre := null**

**val := -114514**

**while head != null:**

**if head.data == val:**

**tmp := head**

**head := head.next**

**pre.next := head**

**delete tmp**

**else:**

**pre := head**

**val := head.data**

**head := head.next  
四、实验源程序**

**#include <iostream>**

**using namespace std;**

**struct Node**

**{**

**int data;**

**Node \*next;**

**};**

**class LinkList**

**{**

**private:**

**Node \*first;//dummy head**

**public:**

**LinkList(int a[],int n){//insert in head**

**first=new Node;**

**Node\*r=first,\*s=nullptr;**

**for(int i=0;i<n;i++){**

**s=new Node;**

**s->data=a[i];**

**r->next=s;**

**r=s;**

**}**

**r->next=nullptr;**

**}**

**LinkList(){**

**first=new Node;**

**first->next=nullptr;**

**}**

**~LinkList(){**

**Node\*p=first;**

**while(first!=nullptr){**

**first=first->next;**

**delete p;**

**p=first;**

**}**

**}**

**int get(int i){**

**Node\*p=first->next;**

**int count=i;**

**while(p!=nullptr && count<i){**

**p=p->next;**

**count++;**

**}**

**if(p==nullptr)**

**throw "invaild index";**

**else**

**return p->data;**

**}**

**void insert(int i,int x){**

**Node \*p=first,\*s=nullptr;**

**int count=0;**

**while (p!=nullptr && count<i-1){**

**p=p->next;**

**count++;**

**}**

**if(p==nullptr)throw"invaild index";**

**else{**

**s=new Node;**

**s->data=x;**

**s->next=p->next;**

**p->next=s;**

**}**

**}**

**int erase(int i){**

**int x;**

**Node \*p=first,\*q=nullptr;**

**int count=0;**

**while(p!=nullptr && count<i-1){**

**p=p->next;**

**count++;**

**}**

**if(p==nullptr || p->next==nullptr)**

**throw "invaild index";**

**else{**

**q=p->next;**

**x=q->data;**

**p->next=q->next;**

**delete q;**

**return x;**

**}**

**}**

**int locate(int x){**

**Node \*p=first->next;**

**int count=1;**

**while(p!=nullptr){**

**if(p->data==x)**

**return count;**

**p=p->next;**

**count++;**

**}**

**return 0;**

**}**

**void printList(){**

**Node\*p=first->next;**

**while (p!=nullptr){**

**cout<<p->data<<'\t';**

**p=p->next;**

**}**

**cout<<'\n';**

**}**

**void reverse(){**

**Node\*pre=nullptr;**

**Node\*head=first->next;**

**if(head==nullptr)return;**

**Node\*nxt=head->next;**

**while(head!=nullptr){**

**head->next=pre;**

**pre=head;**

**head=nxt;**

**if(nxt!=nullptr)**

**nxt=nxt->next;**

**}**

**first->next=pre;**

**}**

**void unique(){**

**Node\*head=first->next;**

**Node\*pre=nullptr;**

**int val= -114514;**

**while(head!=nullptr){**

**if(head->data==val){**

**Node\*tmp=head;**

**head=head->next;**

**pre->next=head;**

**delete tmp;**

**}else{**

**pre=head;**

**val=head->data;**

**head=head->next;**

**}**

**}**

**}**

**};**

**int main(){**

**int arr[]={1,2,3,3,4,5,5,5,6,7,8,9};**

**LinkList l1(arr,sizeof(arr)/sizeof(int));**

**l1.printList();**

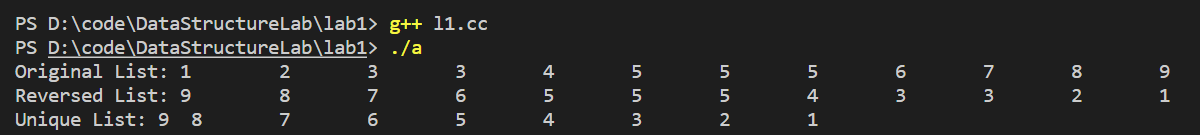
**l1.unique();**

**l1.printList();**

**return 0;**

**}  
五、实验结果与分析（分析出错的原因与调试过程等）**

**运行结果：**



**六、实验心得**

设计链表时处理头结点的边界情况总是一件麻烦事，稍有不慎就会得到一个segment fault，但是本次实验中虚拟头结点的使用就大大简化了这个问题。单链表算法的实现总之就是首先要保证当前这个节点的前一个节点是被记录的，这样才能做相应的增改操作，然后草稿纸上画图，理清楚指针变动的关系，最后处理边界情况。

**实验二 栈和队列的操作**

**一、实验目的**

掌握栈和队列的存储结构、操作特性及实现方法。

**二、实验题目**

1.设从键盘输入一个整数序列：a1, a2, …，an，编写程序实现：采用**链栈**结构存储输入的整数，当ai ≠-1时，将ai进栈；当ai=-1时，输出栈顶整数并出栈。算法应对异常情况给出相应的信息。

2.设以**不带头结点的循环链表**表示**队列**，并且只设一个指针指向队尾结点，

但不设头指针。编写相应的入队和出队程序。

**三、实验预习的核心算法伪代码.**

**链栈的入栈：**

**procedure push(x):**

**p := new Node**

**head := first.next**

**p.next := head**

**p.data := x**

**first.next := p**

**链栈的出栈：**

**procedure pop():**

**if first.next == null:**

**throw "empty stack!"**

**head := first.next**

**first.next := head.next**

**delete head**

**链栈返回栈顶元素：**

**function top():**

**if first.next == null:**

**throw "empty stack!"**

**return first.next.data**

**检测链栈为空**

**function empty():**

**return first.next == null**

**链队列入队**

**procedure push(x):**

**if tail == null:**

**tail := new Node**

**tail.data := x**

**tail.next := tail**

**else:**

**head := tail.next**

**p := new Node**

**p.data := x**

**p.next := head**

**tail.next := p**

**tail := p**

**链队列出队**

**procedure pop():**

**if tail == null:**

**throw "empty queue!"**

**else:**

**head := tail.next.next**

**if tail.next == tail:**

**delete tail**

**tail := null**

**else:**

**delete tail.next**

**tail.next := head**

**返回链队列最前端节点**

**function front():**

**if tail == null:**

**throw "empty queue!"**

**return -114514**

**else:**

**return tail.next.data**

**判断链队列为空**

**function empty():**

**return tail == null  
四、实验源程序**

**1.链栈：**

**#include <iostream>**

**using namespace std;**

**struct Node{**

**int data;**

**Node\*next;**

**};**

**class LinkStack{**

**private:**

**Node\*first;**

**public:**

**LinkStack(){**

**first=new Node;**

**first->next=nullptr;**

**}**

**void push(int x){**

**Node\*p=new Node;**

**Node\*head=first->next;**

**p->next=head;**

**p->data=x;**

**first->next=p;**

**}**

**void pop(){**

**if(first->next==nullptr)**

**throw "empty stack!";**

**Node\*head=first->next;**

**first->next=head->next;**

**delete head;**

**}**

**int top(){**

**if(first->next==nullptr)**

**throw "empty stack!";**

**return first->next->data;**

**}**

**bool empty(){**

**return first->next==nullptr;**

**}**

**~LinkStack(){**

**Node\*p=first;**

**while(first!=nullptr){**

**first=first->next;**

**delete p;**

**p=first;**

**}**

**}**

**};**

**int main(){**

**int t;cin>>t;**

**LinkStack ls;**

**while(t--){**

**int x;cin>>x;**

**if(x!=-1)**

**ls.push(x);**

**else{**

**cout<<ls.top()<<'\n';**

**ls.pop();**

**}**

**}**

**return 0;**

**}**

**2.链队列：**

**#include <iostream>**

**struct Node{**

**int data;**

**Node\*next;**

**};**

**class LinkQueue{**

**private:**

**Node\*tail;**

**public:**

**LinkQueue(){**

**tail=nullptr;**

**}**

**void push(int x){**

**if(tail==nullptr){**

**tail=new Node;**

**tail->data=x;**

**tail->next=tail;**

**}else{**

**Node\*head=tail->next;**

**Node\*p=new Node;**

**p->data=x;**

**p->next=head;**

**tail->next=p;**

**tail=p;**

**}**

**}**

**void pop(){**

**if(tail==nullptr){**

**throw "empty queue!";**

**}else{**

**Node\*head=tail->next->next;**

**if(tail->next==tail){**

**delete tail;**

**tail=nullptr;**

**}else{**

**delete tail->next;**

**tail->next=head;**

**}**

**}**

**}**

**int front(){**

**if(tail==nullptr){**

**throw "empty queue!";**

**return -114514;**

**}**

**else**

**return tail->next->data;**

**}**

**bool empty(){**

**return tail==nullptr;**

**}**

**~LinkQueue(){**

**Node\*p;**

**while(tail!=nullptr){**

**p=tail;**

**tail=tail->next;**

**delete p;**

**}**

**}**

**};**

**int main(){**

**LinkQueue lq;**

**int arr[]={1,2,3,4,5,6,7,8,9};**

**std::cout<<"input sequence:";**

**for(int a: arr){**

**lq.push(a);**

**std::cout<<a<<' ';**

**}**

**std::cout<<'\n';**

**std::cout<<"LinkQueue sequeue:";**

**while(!lq.empty()){**

**std::cout<<lq.front()<<" ";**

**lq.pop();**

**}**

**std::cout<<'\n';**

**return 0;**

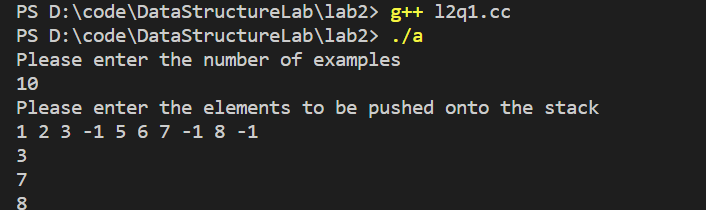
**}  
五、实验结果与分析**

**链栈：**

**输入1 2 3 -1 5 6 7 -1 8 -1**

**得到输出3 7 8**

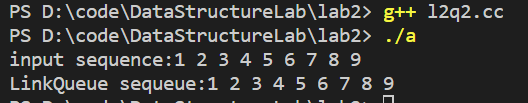
**当第一次遇到-1时栈顶元素为3，弹出3，接着将5,6,7入栈，弹出7，将8入栈，弹出8**



**链队列：**

**构造一个{1,2,3,4,5,6,7,8,9}序列并将其插入链队列中**

**根据队列先进先出的原则输出应当与输入序列一致。**

 **六、实验心得**

比起顺序表栈和队列实现，使用链表主要有以下优势：1.不需要限定死一个最大的容量上限 2.对于队列如果频繁地入队出队，使用顺序表实现会造成大量浪费的空间。

但是使用链表也存在了一些缺点，因为每一个节点都是new和delete的，而new和delete操作本身较为耗时，如果处理不当可能还造成内存泄漏，频繁delete还可能出现内存碎片。

面对实际工程可以考虑一种更系统化的方向案，那就使用一个编写一个内存管理器去管理内存，例如可以批量申请内存构造内存池，当申请内存时从内存池中拿出一块内存，当内存释放将内存归还内存池。

**实验三 二叉树**

**一、实验目的**

熟悉二叉树的结构，掌握二叉树的操作及实现。

**二、实验题目**

编写程序实现以下功能：

1．建立一棵二叉树，所有值均为整数（采用二叉链表存储方式）；

2．输出该二叉树的前序、中序、后序遍历序列；

3. 把所有值为负数的结点修改为正数。

**三、实验预习的核心算法伪代码**

**构造二叉树：**

**function createBinaryTree(array, length, position):**

**if array[position] == FIN:**

**node = null**

**if position != length - 1:**

**position = position + 1**

**else:**

**node = new BiNode**

**node.data = array[position]**

**position = position + 1**

**node.left = createBinaryTree(array, length, position)**

**node.right = createBinaryTree(array, length, position)**

**return node**

**前序遍历：**

**procedure preOrderTraversal(node):**

**if node != null:**

**print(node.data)**

**preOrderTraversal(node.left)**

**preOrderTraversal(node.right)**

**中序遍历：**

**procedure inOrderTraversal(node):**

**if node != null:**

**inOrderTraversal(node.left)**

**print(node.data)**

**inOrderTraversal(node.right)**

**后序遍历：**

**procedure postOrderTraversal(node):**

**if node != null:**

**postOrderTraversal(node.left)**

**postOrderTraversal(node.right)**

**print(node.data)**

**取节点正值**

**procedure bi\_abs(node):**

**if node != null:**

**bi\_abs(node.left)**

**if node.data < 0:**

**node.data = -node.data**

**bi\_abs(node.right)**

**四、实验源程序**

**#include <iostream>**

**#include <functional>**

**using namespace std;**

**#define FIN -114514**

**struct BiNode{**

**int data;**

**BiNode \*left,\*right;**

**};**

**class BiTree{**

**public:**

**BiTree(int a[],int len,int &pos){root=create(a,len,pos);}**

**void preOrder(){\_\_preOrder(root);cout<<'\n';}**

**void inOrder(){\_\_inOrder(root);cout<<'\n';}**

**void postOrder(){\_\_postOrder(root);cout<<'\n';}**

**void bi\_abs(){**

**function<void(BiNode\*)>dfs=[&dfs](BiNode\*p){**

**if(p==nullptr)**

**return;**

**dfs(p->left);**

**if(p->data<0)**

**p->data=-p->data;**

**dfs(p->right);**

**};**

**dfs(root);**

**}**

**~BiTree(){release(root);}**

**private:**

**BiNode \*root;**

**BiNode \*create(int a[],int len,int &pos){**

**BiNode\*bt;**

**if(a[pos]==FIN){**

**bt=nullptr;**

**if(pos!=len-1)**

**pos++;**

**}**

**else{**

**bt=new BiNode;**

**bt->data=a[pos];**

**pos++;**

**bt->left=create(a,len,pos);**

**bt->right=create(a,len,pos);**

**}**

**return bt;**

**}**

**void release(BiNode\*bt){**

**if(bt==nullptr)**

**return;**

**release(bt->left);**

**release(bt->right);**

**delete bt;**

**}**

**void \_\_preOrder(BiNode\*bt){**

**if(bt==nullptr)**

**return;**

**cout<<bt->data<<' ';**

**\_\_preOrder(bt->left);**

**\_\_preOrder(bt->right);**

**}**

**void \_\_inOrder(BiNode\*bt){**

**if(bt==nullptr)**

**return;**

**\_\_inOrder(bt->left);**

**cout<<bt->data<<' ';**

**\_\_inOrder(bt->right);**

**}**

**void \_\_postOrder(BiNode\*bt){**

**if(bt==nullptr)**

**return;**

**\_\_postOrder(bt->left);**

**\_\_postOrder(bt->right);**

**cout<<bt->data<<' ';**

**}**

**};**

**int main() {**

**int a[] = {1, 2, -3, FIN, FIN, -4, FIN, FIN, 5, 6, FIN, FIN, 7, 8, FIN};**

**int pos = 0;**

**BiTree bt(a, sizeof(a) / sizeof(int), pos);**

**cout << "Original Binary Tree:" << endl;**

**cout << "Pre-order: ";**

**bt.preOrder();**

**cout << "In-order: ";**

**bt.inOrder();**

**cout << "Post-order: ";**

**bt.postOrder();**

**cout << "Binary Tree After Applying bi\_abs,Pre-order:" << endl;**

**bt.bi\_abs();**

**bt.preOrder();**

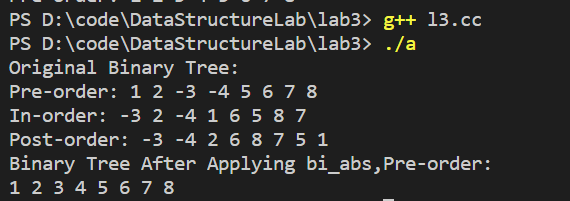
**return 0;**

**}  
五、实验结果与分析**

使用如下序列构建二叉树：1, 2, -3, FIN, FIN, -4, FIN, FIN, 5, 6, FIN, FIN, 7, 8, FIN

得到前序中序后序遍历结果如下：

然后调用bi\_abs()成员函数用于取所有节点值的绝对值



**六、实验心得**

1. 这个实现的API设计中体现了面向对象的思想——封装。有些时候向外部提供的API可能需要其他的辅助函数来实现，此时就可以将真正完成功能的函数访问权限设置为public，而将辅助函数设置为private
2. 这个实现的构造函数我采用的是使用数组序列将值传入而不是一个个由用户输入。但这样就存在一个问题就是需要一个变量在递归的过程中维护此时数组迭代到的下标，我这里直接使用了一个引用，其实可以再做一次函数的封装减少用户调用的参数。
3. 这个终止符号FIN其实存在问题，我定义的是一个很小的负值，但是实际肯定不能这么简单地定义，不然用户存储就受到了限制。这里实际工程中需要更多的变量来维护树的相关信息。

**实验四 图**

**一、实验目的**

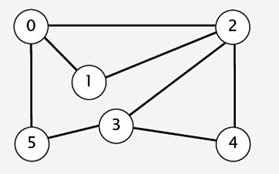
熟悉图的存储结构，掌握图的遍历及实现。

**二、实验题目**

编写程序实现以下功能：

1．创建一个无向图（采用邻接矩阵或者邻接表方式存储）；

2．分别输出从结点0开始的一个深度优先遍历序列和一个广度优先遍历序列。



**三、实验预习的核心算法伪代码**

**创建图：**

**class Graph:**

**nodes = []**

**adjacency\_list = {}**

**procedure addNode(node):**

**nodes.append(node)**

**adjacency\_list[node] = []**

**procedure addEdge(node1, node2):**

**adjacency\_list[node1].append(node2)**

**adjacency\_list[node2].append(node1)**

**深度优先搜索**

**procedure DFS(node, visited):**

**if node not in visited:**

**print(node)**

**visited.add(node)**

**for neighbor in adjacency\_list[node]:**

**DFS(neighbor, visited)**

**procedure DFS\_start():**

**visited = set()**

**for node in nodes:**

**if node not in visited:**

**DFS(node, visited)**

**广度优先搜索**

**procedure BFS(start\_node):**

**queue = Queue()**

**visited = set()**

**queue.enqueue(start\_node)**

**visited.add(start\_node)**

**while not queue.isEmpty():**

**current\_node = queue.dequeue()**

**print(current\_node)**

**for neighbor in adjacency\_list[current\_node]:**

**if neighbor not in visited:**

**queue.enqueue(neighbor)**

**visited.add(neighbor)**

**四、实验源程序**

**#include <iostream>**

**#include <vector>**

**#include <functional>**

**#include <queue>**

**using namespace std;**

**typedef pair<int,int> edge;**

**class Graph{**

**public:**

**Graph(vector<edge> &es){**

**vertexNum=es.size();**

**g.resize(vertexNum);**

**for(auto[u,v]:es){**

**g[u].push\_back(v);**

**g[v].push\_back(u);**

**}**

**}**

**void dfs(){**

**vector<bool>vis(vertexNum,0);**

**function<void(int,vector<bool>&)> \_dfs=[&](int u,vector<bool>&vis){**

**vis[u]=1;**

**cout<<u<<' ';**

**for(int v:g[u])**

**if(!vis[v])**

**\_dfs(v,vis);**

**};**

**\_dfs(0,vis);**

**cout<<'\n';**

**}**

**void bfs(){**

**queue<int>q;**

**q.push(0);**

**vector<bool>vis(vertexNum,0);**

**vis[0]=1;**

**while(!q.empty()){**

**int u=q.front();**

**q.pop();**

**cout<<u<<' ';**

**for(int v:g[u])**

**if(!vis[v]){**

**vis[v]=1;**

**q.push(v);**

**}**

**}**

**cout<<'\n';**

**}**

**private:**

**vector<vector<int>> g;**

**int vertexNum;**

**};**

**int main(){**

**vector<edge> es{{0,1},{0,2},{0,5},{1,2},{2,3},{2,4},{3,4},{3,5}};**

**Graph g(es);**

**cout << "DFS Traversal: ";**

**g.dfs();**

**cout << "BFS Traversal: ";**

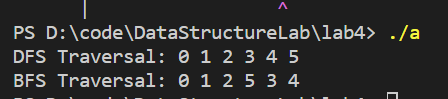
**g.bfs();**

**return 0;**

**}  
五、实验结果与分析**

对于深度优先搜索采用递归的方式，对于选取的初始点搜索与其相连的边，对每一条边一直向下搜索直到到尽头开始返回。

对于宽度优先搜索采用一个队列维护待访问的节点。对每一个节点将其相邻的边存储到队列中并打上已访问的标记，直到队列为空

 **六、实验心得**

**实验五 查找**

**一、实验目的**

熟悉查找的基本过程，掌握常用查找算法设计技巧。

**二、实验题目**

1.设计顺序表的顺序查找算法，将哨兵设在下标高端。

2.构造一棵二叉排序树，从小到大输出树中所有值大于x的结点值。

**三、实验预习的核心算法伪代码**

**顺序表查找：**

**function linearSearch(array, target):**

**for each element in array:**

**if element equals target:**

**return index of element**

**return length**

**二叉排序树：**

**class BiSortTree:**

**Node:**

**int data**

**Node left, right**

**BiSortTree:**

**Node root**

**BiSortTree():**

**root = null**

**procedure insert(value):**

**root = insertRec(root, value)**

**function insertRec(root, value):**

**if root is null:**

**root = new Node(value)**

**else if value < root.data:**

**root.left = insertRec(root.left, value)**

**else:**

**root.right = insertRec(root.right, value)**

**return root**

**procedure printUpperX(x):**

**printUpperXRec(root, x)**

**function printUpperXRec(node, x):**

**if node is not null:**

**if node.data >= x:**

**printUpperXRec(node.left, x)**

**print(node.data)**

**printUpperXRec(node.right, x)**

**else:**

**printUpperXRec(node.right, x)  
四、实验源程序**

**#include <iostream>**

**#include <cstring>**

**using namespace std;**

**#define MAXLEN 100**

**class LinearFind{**

**public:**

**LinearFind(int a[],size\_t len):length(len){**

**memcpy(data,a,sizeof(int)\*len);**

**}**

**size\_t find(int k){**

**data[length]=k;**

**int pos=0;**

**while(data[pos]!=k)**

**pos++;**

**return pos;**

**}**

**private:**

**int data[MAXLEN];**

**size\_t length;**

**};**

**int main(){**

**int arr[13]={1,1,4,5,1,4,1,9,1,9,8,1,0};**

**cout<<LinearFind(arr,13).find(8)<<'\n';**

**return 0;**

**}**

**二叉排序树：**

**#include <iostream>**

**struct BiNode {**

**BiNode\* left;**

**BiNode\* right;**

**int val;**

**};**

**class BiSortTree {**

**public:**

**BiSortTree(int a[], int n) : root(nullptr) {**

**for (int i = 0; i < n; i++)**

**root = \_\_insert(root, a[i]);**

**}**

**~BiSortTree() {**

**\_\_release(root);**

**}**

**void printUpperX(int x){**

**\_\_printUpperX(root,x);**

**}**

**private:**

**BiNode\* root;**

**void \_\_release(BiNode\* bt) {**

**if (bt == nullptr)**

**return;**

**\_\_release(bt->left);**

**\_\_release(bt->right);**

**delete bt;**

**}**

**BiNode\* \_\_insert(BiNode\* bt, int x) {**

**if (bt == nullptr) {**

**BiNode\* s = new BiNode;**

**s->val = x;**

**s->left = nullptr, s->right = nullptr;**

**bt = s;**

**return bt;**

**}**

**if (bt->val > x)**

**bt->left = \_\_insert(bt->left, x);**

**else**

**bt->right = \_\_insert(bt->right, x);**

**return bt;**

**}**

**void \_\_printUpperX(BiNode\* bt , int x) {**

**if (bt != nullptr) {**

**if (bt->val <= x){**

**\_\_printUpperX(bt->right, x);**

**}else{**

**\_\_printUpperX(bt->left, x);**

**std::cout << bt->val << " ";**

**\_\_printUpperX(bt->right, x);**

**}**

**}**

**}**

**};**

**int main() {**

**int arr[] = {5, 3, 7, 1, 4, 6, 8};**

**int n = sizeof(arr) / sizeof(arr[0]);**

**BiSortTree bs(arr, n);**

**std::cout<<"upper than 5: ";**

**bs.printUpperX(5);**

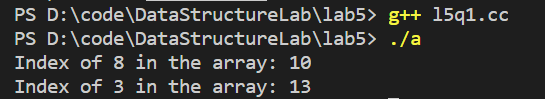
**std::cout<<"\nupper than 2: ";**

**bs.printUpperX(2);**

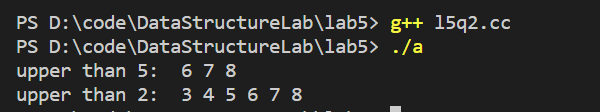
**return 0;**

**}  
五、实验结果与分析**

**给出序列1,1,4,5,1,4,1,9,1,9,8,1,0 构造顺序表查找类，并查找 8和 3 的位置，其中 8 查找成功，3 查找失败返回数组末尾位置**



**给出序列5, 3, 7, 1, 4, 6, 8构造二叉树排序类，并输出大于2和大于5的序列：**

 **六、实验心得**

**实验六 排序**

**一、实验目的**

掌握排序的基本概念，比较基于不同存储结构下排序的算法设计过程。

**二、实验题目**

设待排序的记录序列用单链表作存储结构，编写直接插入排序和简单选择排序的程序。

**三、实验预习的核心算法伪代码**

**插入排序：**

**procedure insertSort():**

**if first is null or first.next is null:**

**return**

**Node ni := first.next**

**Node sorted := null**

**while ni is not null:**

**Node next := ni.next**

**if sorted is null or sorted.data > ni.data:**

**ni.next := sorted**

**sorted := ni**

**else:**

**Node nj := sorted**

**while nj.next is not null and nj.next.data < ni.data:**

**nj := nj.next**

**ni.next := nj.next**

**nj.next := ni**

**ni := next**

**first.next := sorted**

**选择排序：**

**procedure selectSort():**

**if first is null or first.next is null:**

**return**

**Node head := new Node**

**Node tail := head**

**while first.next is not null:**

**Node nmin := null**

**Node nmin\_pref := null**

**int min\_val := INFINITY**

**for Node ni := first; ni.next is not null; ni := ni.next:**

**if ni.next.data < min\_val:**

**nmin := ni.next**

**nmin\_pref := ni**

**min\_val := ni.next.data**

**tail.next := nmin**

**tail := tail.next**

**nmin\_pref.next := nmin.next**

**delete first**

**first := head  
四、实验源程序**

**#include <iostream>**

**struct Node {**

**int data;**

**Node\* next;**

**};**

**class LinkList {**

**private:**

**Node\* first; // dummy head**

**public:**

**LinkList(int a[], int n) { // insert in head**

**first = new Node;**

**Node\* r = first, \* s = nullptr;**

**for (int i = 0; i < n; i++) {**

**s = new Node;**

**s->data = a[i];**

**r->next = s;**

**r = s;**

**}**

**r->next = nullptr;**

**}**

**LinkList() {**

**first = new Node;**

**first->next = nullptr;**

**}**

**~LinkList() {**

**Node\* p = first;**

**while (first != nullptr) {**

**first = first->next;**

**delete p;**

**p = first;**

**}**

**}**

**int get(int i) {**

**Node\* p = first->next;**

**int count = 0;**

**while (p != nullptr && count < i) {**

**p = p->next;**

**count++;**

**}**

**if (p == nullptr)**

**throw "invalid index";**

**else**

**return p->data;**

**}**

**Node\* get\_ptr(int i){**

**Node\* p = first->next;**

**int count = 0;**

**while (p != nullptr && count < i) {**

**p = p->next;**

**count++;**

**}**

**if (p == nullptr)**

**throw "invalid index";**

**else**

**return p;**

**}**

**void insert(int i, int x) {**

**Node\* p = first, \* s = nullptr;**

**int count = 0;**

**while (p != nullptr && count < i - 1) {**

**p = p->next;**

**count++;**

**}**

**if (p == nullptr)**

**throw "invalid index";**

**else {**

**s = new Node;**

**s->data = x;**

**s->next = p->next;**

**p->next = s;**

**}**

**}**

**int erase(int i) {**

**int x;**

**Node\* p = first, \* q = nullptr;**

**int count = 0;**

**while (p != nullptr && count < i - 1) {**

**p = p->next;**

**count++;**

**}**

**if (p == nullptr || p->next == nullptr)**

**throw "invalid index";**

**else {**

**q = p->next;**

**x = q->data;**

**p->next = q->next;**

**delete q;**

**return x;**

**}**

**}**

**int locate(int x) {**

**Node\* p = first->next;**

**int count = 1;**

**while (p != nullptr) {**

**if (p->data == x)**

**return count;**

**p = p->next;**

**count++;**

**}**

**return 0;**

**}**

**void printList(){**

**Node\*p=first->next;**

**while (p!=nullptr){**

**std::cout<<p->data<<' ';**

**p=p->next;**

**}**

**std::cout<<'\n';**

**}**

**void insertSort();**

**void selectSort();**

**};**

**void LinkList::insertSort() {**

**if(first==nullptr || first->next==nullptr)**

**return;**

**Node\* ni=first->next,\*sorted=nullptr;**

**while(ni!=nullptr){**

**Node\*next=ni->next;**

**if(sorted==nullptr || sorted->data > ni->data ){//考虑头插**

**ni->next=sorted;**

**sorted=ni;**

**}**

**else{**

**Node\* nj=sorted;//这里不可以用first,原链关系已被打乱**

**while(nj->next!=nullptr && nj->next->data < ni->data){**

**nj=nj->next;**

**}**

**ni->next=nj->next;**

**nj->next=ni;**

**}**

**ni=next;**

**}**

**first->next=sorted;**

**}**

**void LinkList::selectSort() {**

**if(first==nullptr || first->next==nullptr)**

**return;**

**Node\*head=new Node;**

**Node\*tail=head;**

**while(first->next!=nullptr){**

**Node\*nmin=nullptr;**

**Node\*nmin\_pref=nullptr;**

**int min\_val=114514;**

**for(Node\*ni=first;ni->next!=nullptr;ni=ni->next){**

**if(ni->next->data < min\_val){**

**nmin=ni->next;**

**nmin\_pref=ni;**

**min\_val=ni->next->data;**

**}**

**}**

**tail->next=nmin;**

**tail=tail->next;**

**nmin\_pref->next=nmin->next;**

**}**

**delete first;**

**first=head;**

**}**

**const int N = 105;**

**void solve(){**

**int t;scanf("%d",&t);**

**while(t--){**

**int a[N],n;**

**scanf("%d",&n);**

**for(int i=0;i<n;i++)**

**scanf("%d",&a[i]);**

**LinkList list(a,n);**

**list.selectSort();**

**list.printList();**

**}**

**}**

**int main() {**

**int arr1[]={4,2,1,5,2,5,7};**

**LinkList l1(arr1,sizeof(arr1)/sizeof(arr1[0]));**

**l1.selectSort();**

**std::cout<<"selectSort:";**

**l1.printList();**

**LinkList l2(arr1,sizeof(arr1)/sizeof(arr1[0]));**

**l2.insertSort();**

**std::cout<<"insertSort:";**

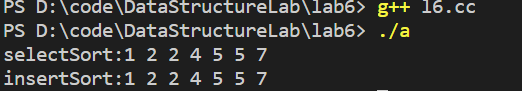
**l2.printList();**

**return 0;**

**}**

**五、实验结果与分析**

**使用4,2,1,5,2,5,7序列构造两个链表，再分别使用链表的成员函数插入排序和选择排序并输出排序后的结果**



**六、实验心得**

链表的排序涉及到大量指针的变动和边界情况的调整，如果直接在原链表上进行指针转化情况会比较复杂。但是考虑到插入排序和选择排序本质上都是从前往后逐步构造出一个有序的序列，所以我们考虑使用一个sorted 指针用于维护新序列，当确认某个节点的位置时，将其从原链表中删除并加入新链表中，最后将first指针指向新链表。

这里为了简化代码在遍历寻找的时候可以使用一个trick,就是将当前节点的下一个节点视作目标节点，这样可以省去记录前一个位置的指针。

此外在第一层循环时要谨记nodei指向的位置可能在链表增删的时候改变，所以要先记录下nodei 的下一个位置用于之后的迭代。

总之这个实验室比较有挑战性的实验，在调试过程中可能因为数组的越界出现大量的段错误。所以确定好方法尤为重要。