第一章

~~老师没发第一章的答案，所以只能随便搞一下了。~~

`算法的五个基本特征`:有穷性 确切性 输入 输出 可行性

在线性表的单链表存储结构中，每一个结点有两个域，一个是`数据域` ,用于存储数据元素值本身，另一个是`指针域` ，用于存储后继结点的地址。

`数据结构` ：相互之间存在一种或多种特定关系的数据元素的集合。

第二章

3.在一个单链表中，已知*q*结点是p结点的前驱结点，若在*q*和p之间插入*s结点， 则执行 __ 。

A. s->next=p->next; p->next=s;

B. p->next=s->next; s->next=p;

C. q->next=s; s->next=p;

D. p->next=s; s->next=q;

选C，要把s插入到q和p的中间，q是p的前驱，所以就这样。

`q->next` 是一个指针 `q->next=s` 就是让他指向s

5.设单链表中指针p指向结点m，指针f指向将要插入的新结点x，当x插在链表的结点m之后时，只要先修改 __ 后修改p->next=f即可。

A. f->next=p;

B. f->next=p->next;

C. p->next=f->next;

D. f=NULL;

很显然选`B` ，是把`f` 插在`p` 后面，所以`f->next=p->next,p->next=f`

这样就是，不会修改p后面的结点，只是在后面插入了一个f。

6.在双向链表中，删除p所指的结点时需修改指针 ___ 。

A. ((p->next) ->next) ->next=p; p->next=(p->next) ->next;

B. (p->prior) ->next=p->next; (p->next) ->prior=p->prior;

C. p->prior=(p->prior) ->prior; ((p->prior) ->prior) ->next=p;

D. ((p->prior) ->prior) ->next=p; p->prior=(p->prior) ->prior;

要删除p所指的结点，其实就是删除p，那我们需要p的前驱的后继变成p的后继，p的后继的前

驱是p的前驱。
即`(p->prior)->next=p->next, (p->next)->prior=p->prior`，所以选B

不带头结点的单链表判空条件是 `head==NULL`
带头结点的单链表判空条件是 `head->next==NULL`
非空循环单链表的尾结点p满足 `p->next==head`

1. 线性表有两种存储结构：顺序表和链表。试问：

(1)两种存储表示各有哪些主要优缺点?
(2)如果有n个线性表同时并存，并且在处理过程中各表的长度会动态发生变化，线性表的总数也会自动地改变。在此情况下，应选用哪种存储结构?为什么?
(3)若线性表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取线性表中的元素，那么，应采用哪种存储结构?为什么?
解答：
(1)顺序存储是按索引（隐含的）直接存取数据元素，方便灵活，效率高，但插入、删除操作时将引起元素移动，因而降低效率；链接存储内存采用动态分配，利用率高，但需增设指示结点之间有序关系的指针域，存取数据元素不如顺序存储方便，但结点的插入、删除操作十分简单。
(2)应选用链接表存储结构。其理由是，链式存储结构用一组任意的存储单元依次存储线性表里各元素，这里存储单元可以是连续的，也可以是不连续的。这种存储结构，在对元素作插入或删除运算时，不需要移动元素，仅修改指针即可。所以很容易实现表的容量扩充。
(3)应选用顺序存储结构。其理由是，每个数据元素的存储位置和线性表的起始位置相差一个和数据元素在线性表中的序号成正比的常数。由此，只要确定了起始位置，线性表中任一数据元素都可随机存取，所以线性表的顺序存储结构是一种随机存取的存储结构。而链表则是一种顺序存取的存储结构。

2. 对链表设置头结点的作用是什么?(至少说出两条好处)

解答：
(1) 对带头结点的链表，在表的任何结点之前插入结点或删除表中任何结点，所要做的都是修改前一结点的指针域，因为任何元素结点都有前驱结点。若链表没有头结点，则首元素结点没有前驱结点，在其前插入结点或删除该结点时操作会复杂些。
(2) 对带头结点的链表，表头指针是指向头结点的非空指针，因此空表与非空表的处理是一样的。

1. 编写下列算法，假定单链表的头指针用L表示，类型为linklist。

(1)将一个单链表中的所有结点按相反次序链接。（5分）
(2)删除单链表中第i个（i≥1）结点。（5分）

(3)删除单链表中由指针p所指向的结点。（5分）
(4)从带有附加表头结点的循环单链表中删除其值等于x的第一个结点。（5分）
(5)在单链表中指针p所指结点之前插入一个值为x的新结点。（5分）
(6)从循环单链表中查找出最小值。（5分）
(7)根据一维数组A(1:n)中顺序存储的具有n个元素的线性表建立一个带有头结点的单链表。（5分）

```cpp
#include <climits>
#include<iostream>
using namespace std;

struct List{
    struct linklist{
        int data;
        linklist* next;
        linklist(int val):data(val),next(nullptr){}
    };//定义结构体
    linklist* L;//作为头结点
    List(){
        L=new linklist(0);
        L->next=nullptr;
    }

    List(int a[],int l,int r){
        L=new linklist(0);
        linklist* curr=L;
        for(int i=l;i<=r;i++){
            linklist* now=new linklist(a[i]);
            curr->next=now;
            curr=now;
        }
    }

    void reverse(){
        //反转单链表
        //思路是把每一个结点的next指向原本的前驱
        linklist* prev=nullptr;
        linklist* curr=L->next;
        linklist* next=nullptr;
        while(curr!=nullptr){
            next=curr->next;
            curr->next=prev;
            prev=curr;
            curr=next;
        }
        L->next=prev;
```

```
        //curr表示当前的结点，prev表示前一个结点，next表示下一个结点
        //每次把当前结点的后继改成原本的前驱，就能反转
        //最后把最后一个结点摄制成最后一个结点
    }

    int min_val(){
        //查找最小值
        int res=INT_MAX;
        linklist* p=L->next;
        while(p!=nullptr){
            res=min(res,p->data);
            p=p->next;
        }
        return res;
    }

    void print(){
        linklist* p=L->next;
        while(p!=nullptr){
            cout<<p->data<<' ';
            p=p->next;
        }
        cout<<'\n';
    }

    ~List(){
        delete L;
    }
};
int main()
{
    // int n;
    // cin>>n;
    // int* a=new int[n+1];
    int a[]={0,6,7,8,9,10};
    // for(int i=1;i<=n;i++)
    //     cin>>a[i];
    List L(a,1,5);
    L.print();
```

```
    L.reverse();
    L.print();
    cout<<L.min_val()<<'\n';
}
```

循环单链表和普通单链表不一样的地方只有，循环的是尾结点的next指向head。

第三章

第五章

```cpp
#include <iostream>
#include <stack>
using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

void preorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;  // 如果树为空，直接返回
    }

    stack<TreeNode*> s;  // 使用栈来模拟递归
    s.push(root);          // 将根节点入栈

    while (!s.empty()) {
        TreeNode* node = s.top();  // 获取栈顶元素
        s.pop();                   // 弹出栈顶元素
        cout << node->val << " ";  // 访问当前节点

        // 先将右子树入栈，再将左子树入栈
        if (node->right != nullptr) {
            s.push(node->right);
        }
        if (node->left != nullptr) {
            s.push(node->left);
        }
    }
}

int main() {
    // 构造二叉树
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
```

```
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(7);

    // 进行先序遍历
    cout << "Preorder Traversal: ";
    preorderTraversal(root);  // 输出: 1 2 4 5 3 6 7
    cout << endl;

    return 0;
}
```

先序遍历

相当于dfs，然后先根再左再右

中序遍历

先左子树，再根节点，再右子树

后序遍历

先左子树，后右子树，再根节点

第六章

Prim算法和**Kruskal算法**都是求解最小生成树（MST，Minimum Spanning Tree）的经典算法，它们的核心目标是从一个连通图中选择一部分边，确保不形成环且边权之和最小。虽然两者都能得到最小生成树，但它们的实现方法和思路有所不同。

1. Prim算法

Prim算法是基于顶点的贪心算法。其核心思想是：从一个起始顶点出发，不断地向生成树中添加最小权重的边，直到所有顶点都被包含进来为止。

步骤：

1. 从任意一个顶点开始（可以选择图的任意一个顶点作为起点）。
2. 将起始顶点标记为已访问。
3. 在所有与当前生成树的顶点相连的边中，选择权重最小的边，并将它连接的一个新顶点加入到生成树中。
4. 重复步骤3，直到生成树包含图中的所有顶点。

算法流程：

1. 选一个起始顶点，将它加入生成树。
2. 找到从已访问的顶点集合到未访问的顶点集合中权值最小的边，加入生成树。
3. 重复步骤2，直到所有顶点都被访问。

时间复杂度：

- 如果使用邻接矩阵：`O(V^2)`，其中 `V` 为图中的顶点数。
- 如果使用邻接表和最小堆：`O(E log V)`，其中 `E` 为图中的边数，`V` 为图中的顶点数。

2. Kruskal算法

Kruskal算法是基于边的贪心算法。其核心思想是：按边的权值从小到大排序，然后从最小的边开始，逐步将边加入到生成树中，前提是不能形成环。

步骤：

1. 将图中的所有边按权值从小到大排序。
2. 初始化每个顶点的集合，每个顶点一开始都属于一个单独的集合。
3. 按照权值的顺序遍历每一条边，如果该边连接的两个顶点属于不同的集合，则将这条边加入生成树，并将这两个集合合并；如果它们已经在同一个集合中，则跳过这条边。
4. 重复步骤3，直到生成树包含 `V-1` 条边。

算法流程：

1. 将所有的边按权值排序。
2. 遍历边集合，若边的两个顶点属于不同的集合，则将该边加入生成树，并合并两个集合。

3. 重复步骤2，直到生成树包含 `V-1` 条边。

时间复杂度：

- 排序边的时间复杂度为 `O(E log E)`，其中 `E` 是边的数量。
- 使用并查集时，查找和合并操作的时间复杂度为 `O(α(V))`，其中 `α` 是阿克曼函数的反函数，几乎可以认为是常数。

比较：

- **Prim算法**：
    - 适用于稠密图（边多的图），尤其是当图的顶点数 `V` 较小且边数 `E` 较多时，性能较好。
    - 是基于顶点的贪心算法，逐步扩展生成树。
    - 实现时通常使用优先队列来找到最小的边。
- **Kruskal算法**：
    - 适用于稀疏图（边少的图），尤其是当图的边数 `E` 较小且顶点数 `V` 较多时，性能较好。
    - 是基于边的贪心算法，逐步选择