算法设计与问题求解

绪论

计算机的优势：速度、记忆、精确。
算法：按部就班解决一个问题或完成某个目标的过程。
算法设计的要求：正确性、具体步骤、确定性、有限性、终止性
算法的评价标准：时间复杂度和空间复杂度

C与C++的差异

C语言——面向过程
C++——面向对象
C++相较C语言增加了bool基本类型，其输出为true或false。
C++引入了命名空间，“::”操作符是域解析操作符。

> **注意**
>
> 输入操作符“>>”在读入下一个输入项前会忽略前一项后面的空格。

函数重载与模板

函数的重载：C++允许多个函数拥有相同的名字，只要它们的参数列表不同即可。
函数模板：建立一个通用函数，其返回值类型和形参类型不具体指定，用一个虚拟的类型来代替。

面向对象初步

类：构造数据类型，成员可以是变量或者函数。
对象：类定义出来的变量。
class：类的声明。
public：类的公有成员，通过当前类创建的对象都可以访问。
private：类的私有成员，只能由该类中的成员函数访问。

string类

需要头文件<string>
相较于C语言中的string，C++中的string类型的变量结尾没有'/0'。

若干数学问题的算法

求两个数的最小公倍数

质数筛

多项式乘除

多项式插值问题

拉格朗日插值法

牛顿插值法

非线性方程求解

二分法

牛顿迭代法

线性方程组求解

雅可比迭代法

高斯消去法

一元线性回归

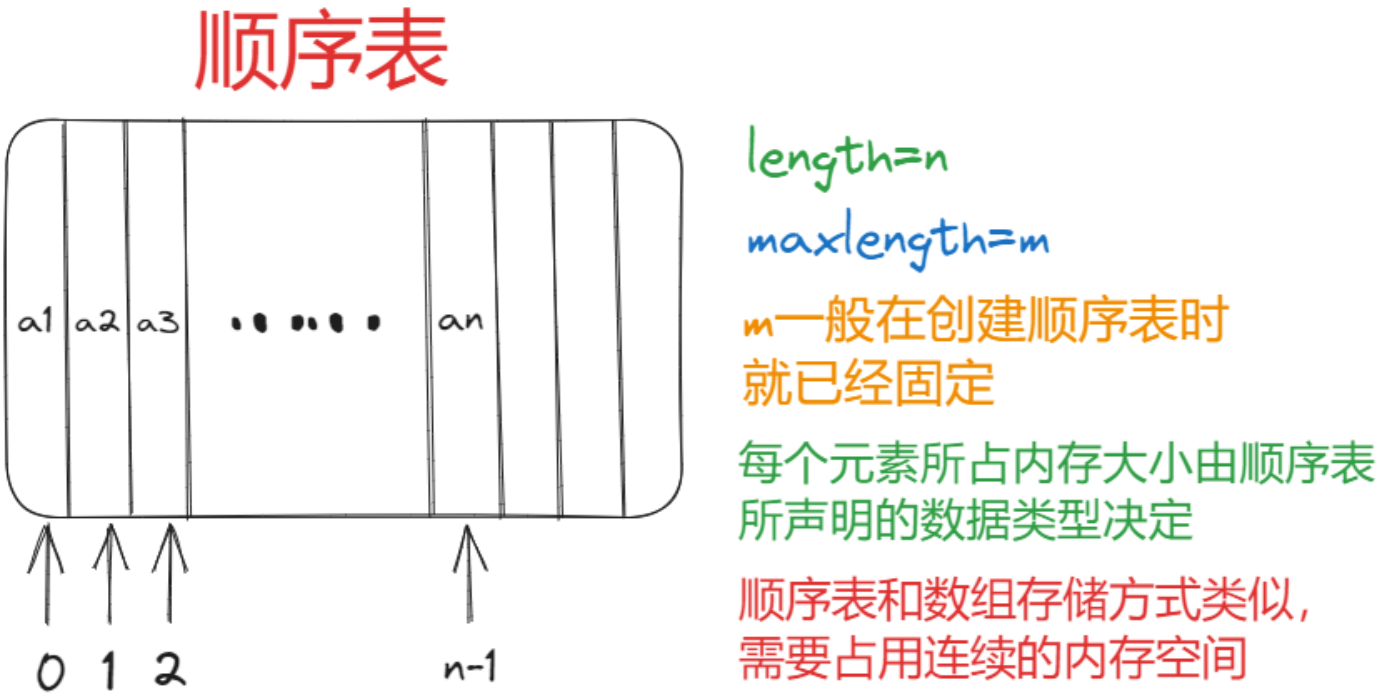
数据结构

> ⚠ **Warning**
>
> 使用前先判断存储结构是否为空，是否为满

线性数据结构

线性表

顺序表



采用顺序存储结构的线性表称为顺序表。**逻辑上相邻的数据元素，其存储位置也彼此相邻。**
插入：从最后一个元素开始依次往后移动一个位置，直到pos位置为止，然后向pos位置存入新元素
删除：从pos位置的后一个元素开始，依次向后直到最后一个元素为止，将每个元素前移一个位置

以下给出一种c++顺序表的可能实现方式(申请静态内存)

```
#include<iostream>

using namespace std;

const int MaxSize=100;
template<class DataType>//类模板的特化，DataType是一个类型参数
class SeqList
{
        public:
                SeqList()//默认构造函数，初始化length为0
                {
                        length=0;
                }
                SeqList(DataType a[],int n);//带参构造函数，用数组a的前n个元素初始化
SeqList
                ~SeqList(){}//析构函数，静态内存在程序使用完后自动释放，故不需要进行操作
                int Length()//返回SeqList长度
                {
                        return length;
                }
```

```
                DataType Get(int i);//返回SeqList第i个元素的值
                int Locate(DataType x);//查找值为x的元素在SeqList中的位置，返回下标
                void Insert(int i,DataType x);//在SeqList的第i个位置插入值为x的元素
                DataType Delete(int i);//删除SeqList的第i个位置的元素，并返回其值
                void PrintList();//输出SeqList中的所有元素
        private:
                DataType data[MaxSize];//存储SeqList中的元素的数组
                int length;//SeqList的长度
};

template<class DataType>
SeqList<DataType>::SeqList(DataType a[],int n)
{
        if(n>MaxSize)
        throw"wrong parameter";
        for(int i=0;i<n;i++)
            data[i]=a[i];
        length=n;
}

template<class DataType>
DataType SeqList<DataType>::Get(int i)
{
        if(i<1||i>length)
        throw"wrong location";
        else
        return data[i-1];
}

template<class DataType>
int SeqList<DataType>::Locate(DataType x)
{
        for(int i=0;i<length;i++)
            if(data[i]==x)
            return i+1;
        return 0;
}

template<class DataType>
void SeqList<DataType>::Insert(int i,DataType x)
{
        if(length>=MaxSize)
        throw"Overflow";
        if(i<1||i>length+1)//注意和Delete中条件不同
        throw"Location";
        for(int j=length;j>=i;j--)
```

```
        data[j]=data[j-1];
    data[i-1]=x;
    length++;
}

template<class DataType>
DataType SeqList<DataType>::Delete(int i)
{
        int x;
        if(length==0)
        throw"Underflow";
        if(i<1||i>length)//注意和Insert中条件不同
        throw"Location";
        x=data[i-1];
        for(int j=i;j<length;j++)
            data[j-1]=data[j];
        length--;
        return x;
}

template<class DataType>
void SeqList<DataType>::PrintList()
{
        for(int i=0;i<length;i++)
            cout<<data[i]<<endl;
}

int main()
{
        SeqList<int> p;
        p.Insert(1,5);
        p.Insert(2,9);
        p.PrintList();
        p.Insert(2,3);
        cout<<p.Length()<<endl;
        p.PrintList();
        cout<<p.Get(3)<<endl;
        p.Delete(2);
        p.PrintList();
        return 0;
}
```

虽然以上的代码已经能够实现顺序表，但在工程上，我们需要用到更多的数据，这个时候使用静态内存往往不是最佳选择，有可能会发生爆栈。我们往往会使用动态内存来存放更多的数据。（程序

的存储结构如图所示)

值得注意的是，我们平时定义的静态变量所占用的内存会在程序执行结束后自动回收，而申请的动态内存不会，需要自己手动释放，否则会造成内存泄漏

以下再给出另外一种申请动态内存实现顺序表的代码

```
#include <iostream>

using namespace std;

template<class DataType>
class SeqList
{
public:
    SeqList();  // 默认构造函数
    ~SeqList();  // 析构函数

    DataType* Create(int length);  // 创建长度为length的顺序表
        int Length();  // 返回顺序表的长度
        int MaxLength();  // 返回顺序表的最大容量
    DataType Get(int i);  // 获取指定位置的元素值
    int Locate(DataType x);  // 查找指定元素的位置

    void Insert(int i, DataType x);  // 在指定位置插入元素
    DataType Delete(int i);  // 删除指定位置的元素
    void PrintList();  // 打印顺序表中的元素

private:
```

```
    DataType* data;  // 存储顺序表元素的数组指针
    int length;  // 顺序表的当前长度
    int maxLength;  // 顺序表的最大长度
};

template<class DataType>
SeqList<DataType>::SeqList()
{
        length=0;  // 初始化顺序表长度
        data=NULL;  // 初始化顺序表指针
}

template<class DataType>
SeqList<DataType>::~SeqList()
{
    delete []data;  // 释放存储顺序表元素的数组内存
}

template<class DataType>
DataType* SeqList<DataType>::Create(int length)
{
    this->length=0;  // 初始化顺序表长度为0
    maxLength=length;  // 初始化顺序表最大长度为length
    data=new DataType[maxLength];  // 创建动态数组，长度为maxLength
    if(data==NULL)
        throw "Memory allocation failure";  // 内存分配失败，抛出异常
    return data;  // 返回指向动态数组的指针
}

template<class DataType>
int SeqList<DataType>::Length()
{
    return length;  // 返回顺序表的长度
}

template<class DataType>
int SeqList<DataType>::MaxLength()
{
    return maxLength;  // 返回顺序表的最大容量
}

template<class DataType>
DataType SeqList<DataType>::Get(int i)
{
    if(i<1||i>length)
        throw "wrong location";  // 抛出位置错误异常
```

```
    else
        return data[i-1];  // 返回指定位置的元素值
}

template<class DataType>
int SeqList<DataType>::Locate(DataType x)
{
    for(int i=0;i<length;i++)
        if(data[i]==x)
            return i+1;  // 返回指定元素的位置
    return 0;  // 如果未找到指定元素，返回0
}

template<class DataType>
void SeqList<DataType>::Insert(int i, DataType x)
{
    DataType* newData=new DataType[length+1];  // 创建新的动态数组，长度加1
    for(int j=0;j<i-1;j++)
        newData[j]=data[j];  // 复制插入位置之前的元素到新数组
    newData[i-1]=x;  // 在指定位置插入新元素
    for(int j=i;j<=length;j++)
        newData[j]=data[j-1];  // 复制插入位置之后的元素到新数组
    delete []data;  // 释放原数组内存
    data=newData;  // 更新数组指针
    length++;  // 长度加1
}

template<class DataType>
DataType SeqList<DataType>::Delete(int i)
{
    if(length==0)
        throw "Underflow";  // 抛出下溢异常
    if(i<1||i>length)
        throw "Location";  // 抛出位置错误异常
    DataType* newData=new DataType[length-1];  // 创建新的动态数组，长度减1
    DataType x=data[i-1];  // 保存要删除的元素值
    for(int j=0;j<i-1;j++)
        newData[j]=data[j];  // 复制删除位置之前的元素到新数组
    for(int j=i;j<length;j++)
        newData[j-1]=data[j];  // 复制删除位置之后的元素到新数组
    delete []data;  // 释放原数组内存
    data=newData;  // 更新数组指针
    length--;  // 长度减1
    return x;  // 返回删除的元素值
}
```

```
template<class DataType>
void SeqList<DataType>::PrintList()
{
    for(int i=0;i<length;i++)
        cout<<data[i]<<endl;  // 打印顺序表中的元素值
}

int main()
{
    SeqList<int> p;  // 创建SeqList类的实例p
        int* list=p.Create(9);
        p.Insert(1,6);
        p.Insert(2,99);
        cout<<p.Get(1)<<" "<<p.Get(2)<<endl;
        cout<<p.Length()<<endl;
        p.Delete(1);
        cout<<p.Length()<<endl;
        p.PrintList();
    return 0;
}
```

链表

单向链表

插入：将新结点指针指向第i个位置的结点，将第i-1位置结点的指针指向新结点
删除：将第i个结点的指针保存到第i-1个结点的指针域上，释放结点

以下给出一个示例代码：

```
#include<iostream>

using namespace std;

template<class T>
class LinkList
{
        public:
                LinkList();//构造函数
                ~LinkList();//析构函数

            int Length();//返回链表长度
            T Get(int pos);//返回第pos个位置的数据
            int Find(T x);//返回数据为x的位置
```

```
            void Insert(int pos,T x);//在第pos个位置插入元素x
            T Delete(int pos);//删除第pos个位置的元素，并返回其元素值
            void Print();//打印链表
        private:
                int length;//链表长度
                struct Node//定义结点
                {
                        T data;//数据域
                        Node* next;//指针域
                };
                Node* head;//头指针
};

template<class T>
LinkList<T>::LinkList()
{
        head=new Node;
        head->next=NULL;
        length=0;
}

template<class T>
LinkList<T>::~LinkList()
{
        Node* temp;
        while(head)
        {
                temp=head;
                head=head->next;
                delete temp;
        }
}

template<class T>
int LinkList<T>::Length()
{
        if(!head->next)
            return 0;
        int i=0;
        Node* temp=head->next;
        while(temp)
        {
                i++;
                temp=temp->next;
        }
        return i;
```

```
}

template<class T>
T LinkList<T>::Get(int pos)
{
        if(pos<0||pos>length-1)
            throw"wrong location";
        Node* temp=head->next;
        for(int i=0;i<pos;++i)
            temp=temp->next;
        return temp->data;
}

template<class T>
int LinkList<T>::Find(T x)
{
        int i=0;
        Node* temp=head;
        while(temp)
        {
                if(temp->data==x)
                    return i;
                temp=temp->next;
                i++;
        }
        return -1;
}

template<class T>
void LinkList<T>::Insert(int pos, T x)
{
   if(pos<0||pos>length)
       throw"wrong location";
   Node* temp=head;
   if(pos==0)
   {
       Node* p=new Node;
       if (p==NULL)
           throw"memory allocation failed";
       p->data=x;
       p->next=head->next;
       head->next=p;
       length++;
       return;
   }
   if(pos==length)
```

```
    {
        Node* p=new Node;
        if(p==NULL)
            throw"memory allocation failed";
        p->data=x;
        p->next=NULL;
        if(head->next==NULL)
            head->next=p;
        else
        {
            Node* temp=head->next;
            while(temp->next)
                temp=temp->next;
            temp->next=p;
        }
        length++;
        return;
    }
    for(int i=0;i<pos-1;++i)
        temp=temp->next;
    Node* p=new Node;
    if(p==NULL)
        throw"memory allocation failed";
    p->data=x;
    p->next=temp->next;
    temp->next=p;
    length++;
}

template<class T>
T LinkList<T>::Delete(int pos)
{
        if(pos<=0||pos>length-1)
            throw"wrong location";
        if(length==0)
            throw"list is empty";
        Node* p=head;
        Node* q;
        int i=0;
        while(p!=NULL&&i<pos)
        {
                q=p;
                p=p->next;
                i++;
        }
        q->next=p->next;
```

```
        T data=p->data;
        delete p;
        length--;
        return data;
}

template<class T>
void LinkList<T>::Print()
{
        Node* temp=head->next;
        while(temp)
        {
                cout<<temp->data<<" ";
                temp=temp->next;
        }
        cout<<endl;
}

int main()
{
        LinkList<int> s;//特化模板，声明数据类型为int的链表s
        s.Insert(0,154);//以下为测试用例
        s.Insert(1,99);
        s.Insert(2,48);
        cout<<s.Length()<<endl;
        s.Print();
        cout<<s.Get(2)<<endl;
        cout<<s.Find(99)<<endl;
        cout<<s.Delete(1)<<endl;
        cout<<s.Length()<<endl;
        s.Print();
        return 0;
}
```

单向循环链表

表尾指针指向表头

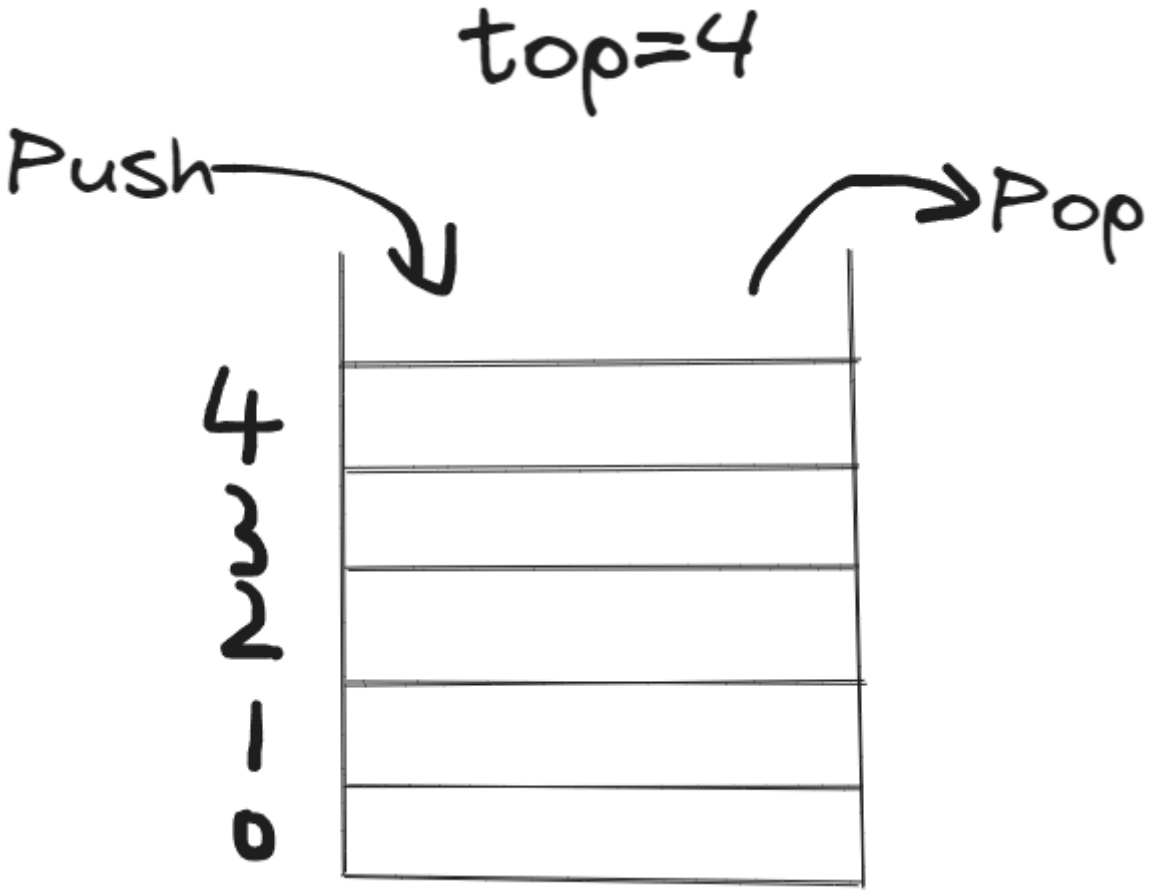
双向链表

每个链表结点既有指向下一个元素的指针，又有指向前一个元素的指针。

双向循环链表

栈

只能在一端进行插入和删除操作的特殊线性表。
特点：先进后出，后进先出
操作：创建空栈，清空栈，入栈（push），出栈（pop），读取栈顶元素
一般将数组的0下标单元作为栈底，将栈顶元素的下标存储在栈顶指针top中。
注意：初始化时把栈顶指针的值赋为-1，表示空栈



以下给出栈的示例代码

```
#include<iostream>

using namespace std;

template <class T>
class STACK
{
        public:
                STACK(int length);//创建栈
                ~STACK();//销毁栈
                void Push(T data);//进栈
                T Pop();//出栈，返回出栈元素
                T Get();//读栈顶，返回栈顶元素
                int Length();//得到栈的长度
        private:
```

```
                struct Stack
                {
                        T* array;
                        int top;
                };
            Stack* stack;
                int maxLength;
};

template <class T>
STACK<T>::STACK(int length)
{
        stack=new Stack;
        if(stack)
        {
                stack->array=new T[length];
                if(stack->array==NULL)
                    throw"memory allocation failed";
                maxLength=length;
                stack->top=-1;
        }
}

template <class T>
STACK<T>::~STACK()
{
        delete []stack->array;
        delete stack;
}

template <class T>
void STACK<T>::Push(T data)
{
        if(stack->top<maxLength-1)
        {
                stack->top++;
                stack->array[stack->top]=data;
        }
        else
            throw"Stack is full.";
}

template <class T>
T STACK<T>::Pop()
{
        if(stack->top>=0)
```

```
        {
                T data=stack->array[stack->top];
                stack->top--;
                return data;
        }
        else
            throw"Stack is empty.";
}

template <class T>
T STACK<T>::Get()
{
        if(stack->top>=0)
            return stack->array[stack->top];
        else
            throw"Stack is empty.";
}

template <class T>
int STACK<T>::Length()
{
        return stack->top+1;
}

int main()
{
        STACK<int> s(20);
        int num[3];
        for(int i=0;i<3;i++)
        {
                cin>>num[i];
                s.Push(num[i]);
        }
        cout<<endl;
        while(s.Length()>0)
                cout<<s.Pop()<<endl;

        return 0;
}
```

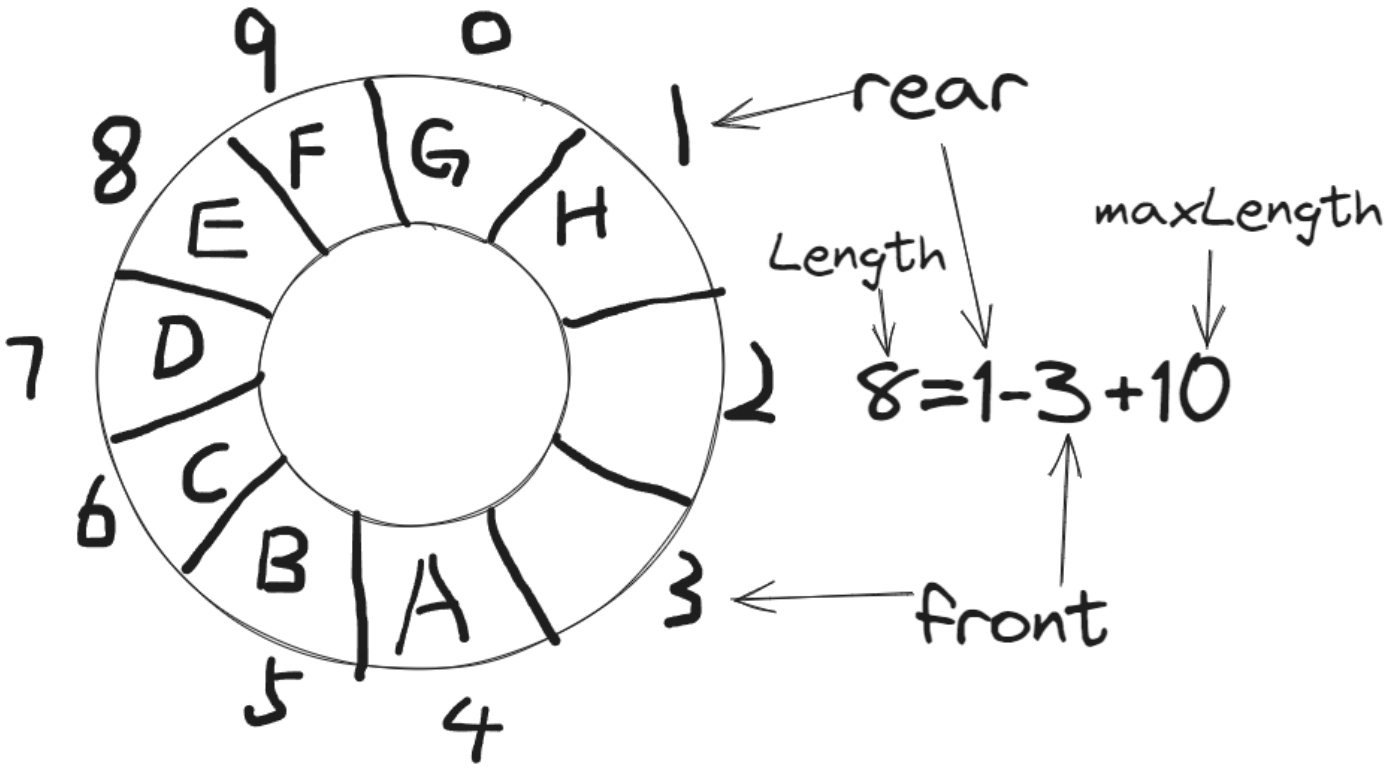
队列

只能在表的一端进行插入（队尾）、在另一端进行删除操作（队头）。
采用顺序存储结构，通过移动队头和队尾指针来获取元素。

特点：先进先出，后进后出
操作：创建空队列、求队列长度、清空队列、入队、出队、读队头元素

循环队列

帮助解决队列假溢出的问题。
与循环链表不同，采用数学方法将rear或front指针从数组空间的最大下标位置移动到最小下标位置。
队空条件：rear == front
队满条件：(rear+1)%M == front，M为最大数组空间



以下给出示范代码：

```
#include<iostream>

using namespace std;

template <class T>
class Queue
{
        public:
                Queue(int length);
            ~Queue();
            int Length();
            void Enqueue(T data);
            T Dequeue();
```

```
            T Front();
            void Print();
        private:
                struct queArray
                {
                        T* array;
                    int front;
                    int rear;
                };
                int maxLength;
                queArray* queue;
};

template <class T>
Queue<T>::Queue(int length)
{
        queue=new queArray;
        if(queue)
        {
                queue->array=new T[length];
                if(queue->array==NULL)
                    throw"memory allocation failed";
                queue->front=0;
                queue->rear=0;
                maxLength=length;
        }
}

template <class T>
Queue<T>::~Queue()
{
        delete []queue->array;
        delete queue;
}

template <class T>
int Queue<T>::Length()
{
        return queue->rear>=queue->front?queue->rear-queue->front:queue->rear-
queue->front+maxLength;
}

template <class T>
void Queue<T>::Enqueue(T data)
{
        if((queue->rear+1)%maxLength==queue->front)
```

```
            cout<<"Queue is full.";
        else
        {
                queue->rear=(queue->rear+1)%maxLength;
                queue->array[queue->rear]=data;
        }
}

template <class T>
T Queue<T>::Dequeue()
{
        if(Length()>0)
        {
                queue->front=(queue->front+1)%maxLength;
                return queue->array[queue->front];
        }
        else
        throw"Queue is empty.";
}

template <class T>
T Queue<T>::Front()
{
        if(Length()>0)
            return queue->array[(queue->front+1)%maxLength];
        else
                throw"Queue is empty.";
}

template <class T>
void Queue<T>::Print()
{
        int newfront=queue->front;
        int newrear=queue->rear;
        while(newfront!=newrear)
        {
                newfront=(newfront+1)%maxLength;
                cout<<queue->array[newfront]<<endl;
        }
}

int main()
{
        Queue<int> p(10);
        p.Enqueue(84);
        p.Enqueue(75);
```

```
        p.Enqueue(39);
        cout<<p.Dequeue()<<endl;
        p.Print();
    p.Enqueue(13);
        p.Enqueue(8);
        p.Enqueue(47);
        p.Print();

        return 0;
}
```

数据结构是本课程难度较高的内容，要求同学们能够在对不同存储方式原理了解的基础上，将其转化为代码实现。同学们在学习这部分内容时切忌眼高手低，应多写多试来熟悉数据结构的代码实现。以下是我强烈推荐的这部分内容的b站课程，希望大家都能够掌握这部分知识。
【【强烈推荐】深入浅出数据结构 - 顶尖程序员图文讲解 - UP主翻译校对 (已完结)】
https://www.bilibili.com/video/BV1Fv4y1f7T1?vd_source=1dfdb49439c4714973b55e47cf074e41

非线性数据结构

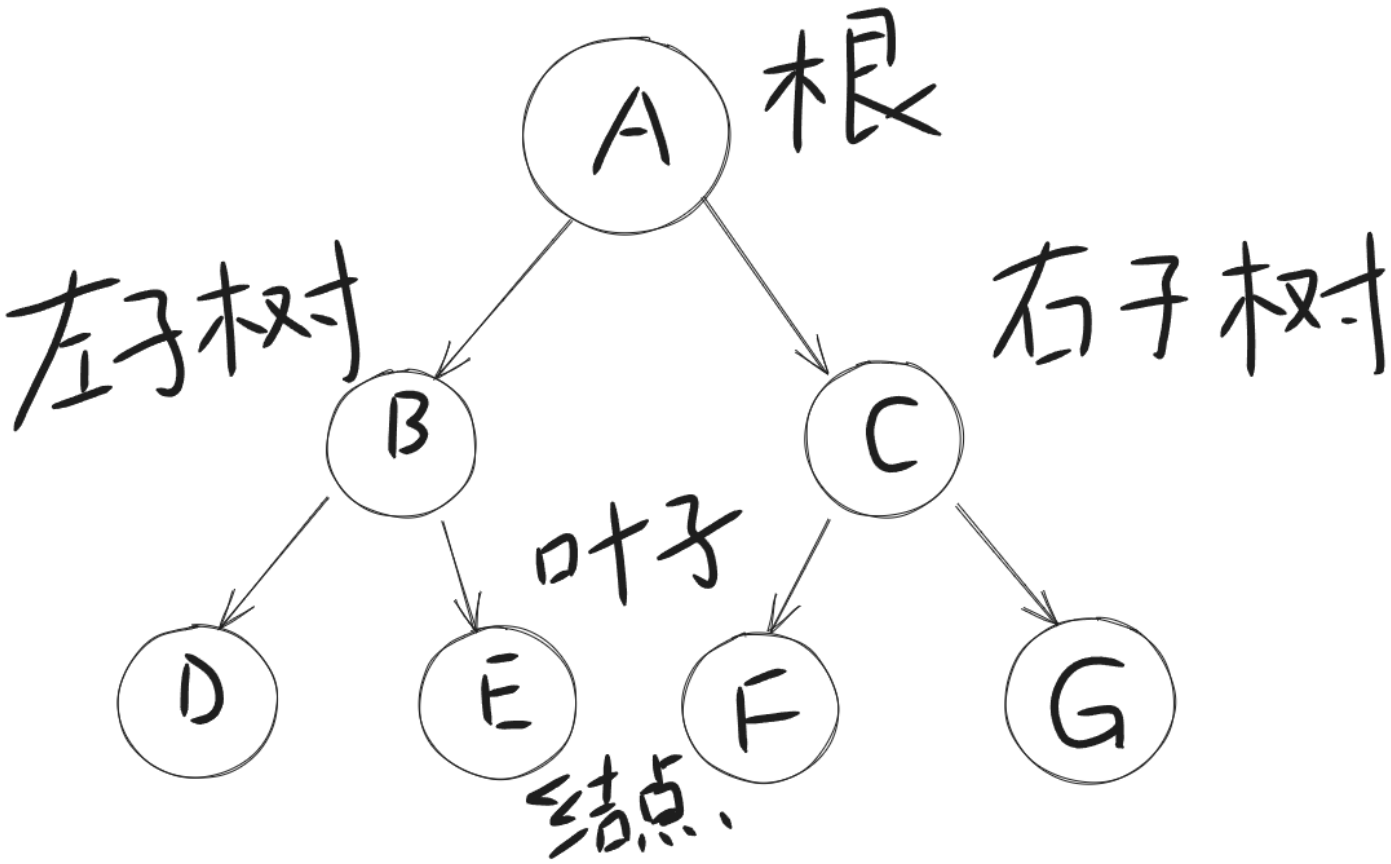
树

树是一种递归定义的数据结构。树是一个或多个结点组成的有限集合T，其中第一个特定结点称为根。处在每个子树底端的结点称为叶子结点，度为零。
结点的度：结点拥有的非空子树的个数。
树的度：树中所有结点的度中的最大值。
结点的层次：根结点的层次为1，其子结点的层次为2，依此类推。
树的深度：树中结点所在的最大层次。
有序树和无序树：树中各结点的子树看成自左向右有序的，则称为有序树，反之称为无序树。

二叉树

二叉树是每个结点最多有两个子树的树结构。
二叉树是有序树，结点的子树分别称为左子树和右子树。

满二叉树：当二叉树每个分支结点的度都是2，且所有叶子结点都在同一层上，则称其为满二叉树。



完全二叉树：从满二叉树叶子所在的层次中，自右向左连续删除若干叶子所得到的二叉树称为完全二叉树。满二叉树可看作是完全二叉树的一个特例。
二叉树的第i层至多有2^{i-1}个结点；深度为k的二叉树至多有2^k-1个结点。

完全二叉树

将完全二叉树的每个结点从上到下，每一层从左至右进行1至n的编号。
性质：1.若i = 1，则该结点是二叉树的根，否则，编号为$[i/2]$的结点为结点i的父结点；
2.若$2*i>n$，则该结点无左子树。否则，编号为$2*i$的结点为结点i的左子树；
3.若$2*i+1>n$，则该结点无右子树，否则，编号为$2*i+1$的结点为结点i的右子树。
可以将二叉树通过补充虚结点的方式变为完全二叉树进行储存。

平衡二叉树

如果对于二叉树的每个结点，左子树和右子树之间的高度差不大于k（大多数时候k是1），这种二叉树叫做平衡二叉树。

二叉搜索树（BST）

对于二叉树的每个结点，所有左子树上的结点值都比该结点值小，所有右子树上的结点值都比该结点值大。（适合递归）

遍历

先序遍历

首先访问根，然后按先序遍历方式访问左子树，再按先序遍历方式访问右子树。

中序遍历

首先访问左子树，然后访问根，再访问右子树。

后序遍历

首先访问左子树，然后访问右子树，再访问根。

以下给出二叉搜索树的示例

```
#include<iostream>
#include<algorithm>
#include<queue>
using namespace std;

struct BSTNode{
    int data;
    BSTNode* left;
    BSTNode* right;
    int height;
    BSTNode(int
    value):data(value),left(NULL),right(NULL),height(0){}
};

int FindHeight(BSTNode* root){
    if(root==NULL) return -1;
    return max(FindHeight(root->left),FindHeight(root->right))+1;
}

int GetBalanced(BSTNode* root){
    if(root==NULL) return 0;
    return FindHeight(root->left)-FindHeight(root->right);
}

BSTNode* GetNewNode(int data){
    BSTNode* newNode=new BSTNode();
    newNode->data=data;
    newNode->left=newNode->right=NULL;
    return newNode;
}
```

```
BSTNode* Insert(BSTNode* root,int data){
    if(root==NULL){
        root=GetNewNode(data);
    }
    else if(data<=root->data){
        root->left=Insert(root->left,data);
    }
    else{
        root->right=Insert(root->right,data);
    }
    return root;
}

bool Search(BSTNode* root,int data){
    if(root==NULL) return false;
    else if(root->data==data) return true;
    else if(data<=root->data) return Search(root->left,data);
    else return Search(root->right,data);
}

BSTNode* FindMin(BSTNode* root){
    if(root==NULL){
        cout<<"Error:Tree is empty."<<endl;
        return NULL;
    }
    while(root->left!=NULL){
        root=root->left;
    }
    return root;
}

BSTNode* FindMax(BSTNode* root){
    if(root==NULL){
        cout<<"Error:Tree is empty."<<endl;
        return NULL;
    }
    while(root->right!=NULL){
        root=root->right;
    }
    return root;
}

int FindHeight(BSTNode* root){
    if(root==NULL){
        return -1;
```

```
    }
    return max(FindHeight(root->left),FindHeight(root->right))+1;
}

void Preorder(BSTNode* root){
    if(root==NULL) return;
    cout<<root->data<<" ";
    Preorder(root->left);
    Preorder(root->right);
}

void Inorder(BSTNode* root){
    if(root==NULL) return;
    Inorder(root->left);
    cout<<root->data<<" ";
    Inorder(root->right);
}

void Postorder(BSTNode* root){
    if(root==NULL) return;
    Postorder(root->left);
    Postorder(root->right);
    cout<<root->data<<" ";
}

BSTNode* Delete(BSTNode* root,int data){
    if(root==NULL) return root;
    else if(data<root->data) root->left=Delete(root->left,data);
    else if(data>root->data) root->right=Delete(root->right,data);
    else{
        if(root->left==NULL&&root->right==NULL){
            delete root;
            root=NULL;
        }
        else if(root->left==NULL){
            BSTNode* temp=root;
            root=root->right;
            delete temp;
        }
        else if(root->right==NULL){
            BSTNode* temp=root;
            root=root->left;
            delete temp;
        }
        else{
            BSTNode* temp=FindMin(root->right);
```

```
            root->data=temp->data;
            root->right=Delete(root->right,temp->data);
        }
    }
    return root;
}

int main(){
    BSTNode* root=NULL;
    root=Insert(root,15);
    root=Insert(root,10);
    root=Insert(root,20);
    root=Insert(root,25);
    root=Insert(root,8);
    root=Insert(root,12);
    root=Insert(root,11);
    Preorder(root);
    cout<<endl;
    Inorder(root);
    cout<<endl;
    Postorder(root);
    cout<<endl;
    Delete(root,10);
    Preorder(root);
    return 0;
}
```

AVL树（平衡二叉搜索树）

查找、插入、删除时间复杂度均为$O(\log_2 n)$

哈夫曼树

以一些带有固定权值的结点作为叶子所构造的，具有最小带权路径长度的二叉树。
示例：

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <sstream>

using namespace std;

// 哈夫曼树节点定义
```

```
struct HuffmanNode {
    char data;
    int weight;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(char data, int weight) : data(data), weight(weight),
left(nullptr), right(nullptr) {}
};

// 自定义的比较函数对象，用于确定节点的优先级
struct CompareNodes {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->weight > b->weight; // 按照权重值进行比较，构建最小堆
    }
};

// 递归构建哈夫曼树
HuffmanNode* buildHuffmanTree(vector<int>& frequencies) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, CompareNodes> pq;

    // 创建叶节点并插入优先队列
    for (int i = 0; i < frequencies.size(); i++) {
        char data = 'A' + i;
        int weight = frequencies[i];
        pq.push(new HuffmanNode(data, weight));
    }

    // 构建哈夫曼树
    while (pq.size() > 1) {
        HuffmanNode* leftNode = pq.top();
        pq.pop();
        HuffmanNode* rightNode = pq.top();
        pq.pop();

        int combinedWeight = leftNode->weight + rightNode->weight;
        HuffmanNode* parentNode = new HuffmanNode('\0', combinedWeight);
        parentNode->left = leftNode;
        parentNode->right = rightNode;

        pq.push(parentNode);
    }

    return pq.top();
}
```

```
// 递归生成哈夫曼编码
void generateHuffmanCode(HuffmanNode* root, string currentCode,
unordered_map<char, string>& codes) {
    if (root == nullptr) {
        return;
    }

    if (root->left == nullptr && root->right == nullptr) {
        codes[root->data] = currentCode;
    }

    generateHuffmanCode(root->left, currentCode + "0", codes);
    generateHuffmanCode(root->right, currentCode + "1", codes);
}

int main() {
    vector<int> frequencies = { 1, 2, 4, 8, 16, 32, 64, 128 };

    HuffmanNode* root = buildHuffmanTree(frequencies);

    unordered_map<char, string> codes;
    generateHuffmanCode(root, "", codes);

    string encodedString;
    for (int i = 0; i < frequencies.size(); i++) {
        char data = 'A' + i;
        encodedString += codes[data];
    }

    cout << "Encoded string: " << encodedString << endl;

    return 0;
}
```

图

路径长度：非带权图的路径长度是指此路径上边或弧的条数。
连通图：在无向图中，若从顶点v_i到v_j有路径，则顶点v_i与v_j是连通的。如果图中任意一对顶点都是连通的，则称此图为连通图。
利用一维数组存储顶点信息。
利用二维数组存储顶点间边或弧的信息。此二维数组称为邻接矩阵。
邻接表是数组与链表结合的存储形式。
邻接表有两种结点：头结点和表结点（类似链表结点）

图的搜索

> ⚠ 注意
>
> 不能重复访问！！！

深度优先（DFS）

广度优先（BFS）

图的最短路径

路径的开始顶点为源点，路径的最后一个顶点为终点。
设顶点集$V=\{0,1,\ldots,n-1\}$，并假定所有边上的权值均是表示长度的非负实数。
设$G=<V,E,W>$是带权的图，其中V是顶点集合，E是边集合，W是权值集合，$W(i,j)$为从结点i到结点j的直连距离
优先队列PQueue：是指顶点出队列时，首先找到其中距离源点最近的元素，再将这个元素交换到头部，最后将队头元素出队列

贪心算法

（1）可行的
（2）局部最优
（3）不可取消

活动规划和马踏棋盘

道路规划和最小生成树

在一个有N个点的含权的连通图中，找出其中的$N-1$条边，它们恰好连接所有的N个点，并且这$N-1$条边的边权之和是所有方案中最小的。
Prim算法示例

```
#include<iostream>
#define MAX 100
#define Infinity 65535
using namespace std;

//定义图的邻接表
struct EdgeNode{
        int no;//边的序号
```

```
        char info;//边的名称
        int weight;//边的权值
        struct EdgeNode* next;//下一个边
};

//定义结点
struct VexNode{
        char info;//结点名称
        struct EdgeNode* link;//与之相连的端点
};

void CreateGraph(VexNode* adjlist,const int nVex,const int nEdge){
        for(int i=1;i<=nVex;i++){
                cout<<"请输入结点"<<i<<"的名称：";
                cin>>adjlist[i].info;
                adjlist[i].link=NULL;
        }
        //每输入一条边，在邻接表中添加两个对应的结点
        EdgeNode* p1;
        EdgeNode* p2;
        int v1,v2;
        int weight;
        for(int i=1;i<=nEdge;i++){
                cout<<"请输入边"<<i<<"的两端的结点序号：";
                cin>>v1>>v2;
                cout<<"此边的权值：";
                cin>>weight;
                p1=new EdgeNode;
                p2=new EdgeNode;
                p1->no=v1;
                p1->weight=weight;
                p1->info=adjlist[v1].info;
                p1->next=adjlist[v2].link;
                adjlist[v2].link=p1;
                p2->no=v2;
                p2->weight=weight;
                p2->info=adjlist[v2].info;
                p2->next=adjlist[v1].link;
                adjlist[v1].link=p2;
        }
}

int CreateMST(VexNode* adjlist,int* parent,const int n,const int startVex){
        bool visited[MAX];
        int lowcost[100];//存储从开始结点到结点j的最小花费
```

```
        for(int i=1;i<=n;i++){
                visited[i]=false;
                lowcost[i]=Infinity;
                parent[i]=startVex;
        }
        //最小生成树的权值总和
        int sum=0;
        EdgeNode *p,*q;
        p=adjlist[startVex].link;
        visited[startVex]=true;
        while(p!=NULL){
                lowcost[p->no]=p->weight;
                p=p->next;
        }
        //寻找最小的权值加入
        int minCost;
        for(int i=1;i<n;i++){
                minCost=Infinity;
                int k;
                for(int j=1;j<=n;j++){
                        if(minCost>lowcost[j]&&!visited[j]){
                                minCost=lowcost[j];
                                k=j;
                        }
                }
                //总权值
                sum+=minCost;
                visited[k]=true;
                q=adjlist[k].link;
                while(q!=NULL){
                        if(!visited[q->no]&&q->weight<lowcost[q->no]){
                                lowcost[q->no]=q->weight;
                                parent[q->no]=k;
                        }
                        q=q->next;
                }
        }
        return sum;
}

int main(){
        VexNode adjlist[MAX];

        int nVex;
        int nEdge;
        int startVex;
```

```
        int parent[MAX];//结点的前驱结点
        cout<<"请输入结点数：";
        cin>>nVex;
        cout<<"请输入边数：";
        cin>>nEdge;
        cout<<"请输入从哪一个结点开始：";
        cin>>startVex;

        CreateGraph(adjlist,nVex,nEdge);
        int sum=CreateMST(adjlist,parent,nVex,startVex);

        cout<<"最小生成树的边集为："<<endl;
        for(int i=1;i<=nVex;i++){
                if(i!=startVex)
                    cout<<"("<<adjlist[parent[i]].info<<","
<<adjlist[i].info<<")"<<"";
        }
        cout<<endl;
        cout<<"最小生成树的权值为："<<sum<<endl;
        return 0;
}
```

Kruskal算法示例

```
#include <iostream>
#define MAX 100

using namespace std;

struct Edge{
        int no;
        int x;
        int y;
        int weight;
        bool selected;
};

int FindSet(int x,int* parent){
        if(x!=parent[x]){
                parent[x]=FindSet(parent[x],parent);
        }
        return parent[x];
}

void UnionSet(int x,int y,int w,int &sum,int* mstRank,int* parent){
```

```
        if(x==y)
            return;
        if(mstRank[x]>mstRank[y])
            parent[y]=x;
        else{
                if(mstRank[x]==mstRank[y])
                    mstRank[y]++;
                parent[x]=y;
        }
        sum+=w;
}

void FastSort(Edge* edge,int begin,int end){
        if(begin<end){
                int i=begin-1,j=begin;
                edge[0]=edge[end];
                while(j<end){
                        if(edge[j].weight<edge[0].weight){
                                i++;
                                Edge temp1=edge[i];
                                edge[i]=edge[j];
                                edge[j]=temp1;
                        }
                        j++;
                }
                Edge temp2=edge[end];
                edge[end]=edge[i+1];
                edge[i+1]=temp2;
                FastSort(edge,begin,i);
                FastSort(edge,i+2,end);
        }
}

int main(){
        Edge edge[MAX];

        int mstRank[MAX];

        int parent[MAX];

        int n;
        int sum=0;
        cout<<"请输入边的个数：";
        cin>>n;

        int weight;
```

```
        for(int i=1;i<=n;i++){
                edge[i].no=i;
                cout<<"请输入第"<<i<<"条边的二个端点序号：";
                cin>>edge[i].x>>edge[i].y;
                cout<<"这条边的权值为：";
                cin>>edge[i].weight;
                edge[i].selected=false;

                parent[edge[i].x]=edge[i].x;
                parent[edge[i].y]=edge[i].y;
                mstRank[edge[i].x]=0;
                mstRank[edge[i].y]=0;
        }

        FastSort(edge,1,n);
        for(int i=1;i<=n;i++){
                int x,y;
                x=FindSet(edge[i].x,parent);
                y=FindSet(edge[i].y,parent);
                if(x!=y){
                        edge[i].selected=true;
                        UnionSet(x,y,edge[i].weight,sum,mstRank,parent);
                }
        }
        cout<<"最小生成树的边集为："<<endl;
        for(int i=1;i<=n;i++){
                if(edge[i].selected){
                        cout<<"序号："<<edge[i].no<<" "<<"端点1："
<<edge[i].x<<"，端点2："<<edge[i].y<<endl;
                }
        }
        cout<<"最小生成树的权值为："<<sum<<endl;

        return 0;
}
```

动态规划

动态规划算法是通过自底向上，递归地从子问题的最优解逐步构造出整个问题的最优解的，而备忘录方法的递归方式是自顶向下的。
当一个问题的所有子问题都至少需要解一次时，动态规划算法比备忘录方法好。当部分子问题不必求解时，用备忘录方法比较好。
特点：（1）最优化原理

（2）无后效性
（3）重叠子问题

挖金矿问题

0-1背包问题

最长公共子序列问题

第十周第一题

背包问题与0-1背包问题类似，所不同的只是在选择物品i装入背包时，可以**选择物品的一部分**而不一定要全部，1≤i≤n。以上面的例子为例，假设选中物品3，在0-1背包问题里，必须要把物品3的总重量60斤全部放进背包，而背包问题，则可以选择放进少于60斤的物品3，可以不必把60斤全放入背包。

输入：物品个数、各物品的价值和重量

输出：物品建议取法

样本输入：

（请输入物品个数:----------**注：此行不显示**）

3

（请输入各物品的价值 :----------**注：此行不显示**）

40 90 240

（请输入各物品的重量 :----------**注：此行不显示**）

20 30 60

样本输出：

（物品建议取法:----------**注：此行不显示**）

x[0] x[1] x[2]

0.000 1.000 1.000

解

```cpp
#include<iostream>
#include<map>//调用STL库里的映射

using namespace std;

int Fill(int W,map<double,int>& bag,int wei[]){
        int RW=W;//定义背包的剩余空间
        int Value=0;//背包所装物品的总价值
        //映射的排序为键值从小到大，因此我们可以利用迭代器实现倒序输出，即尽可能多装入更有价值的物品
        for(auto it=bag.rbegin();it!=bag.rend();it++){
                if(wei[it->second]<=RW){
                        Value+=it->first*wei[it->second];//添加新装入物品的价值
                        RW-=wei[it->second];//减少背包的剩余容量
                }
                else{
                        Value+=RW*it->first;//剩余的容量全部装入这种物品
                        wei[it->second]=RW;//更新这种物品装入背包的重量
                        //背包已经装满，其他物品都装不下了，重置其他物品装入背包的重量
                        RW=0;
                        it++;
                        for(;it!=bag.rend();it++){
                                wei[it->second]=0;
                        }
                        break;//装完物品，跳出for循环
                }
        }
        return Value;
}

int main(){
        int count,W;
        cout<<"请输入背包的容量："<<endl;
        cin>>W;
        cout<<"请输入物品的种类："<<endl;
        cin>>count;

        int val[100];
        int wei[100];
        map<double,int>bag;//创建映射

        for(int i=0;i<count;i++){
                cout<<"请输入第"<<i+1<<"件物品的价值："<<endl;
```

```
                cin>>val[i];
        }

        for(int i=0;i<count;i++){
                cout<<"请输入第"<<i+1<<"件物品的重量："<<endl;
                cin>>wei[i];
                bag.insert(pair<double,int>(static_cast<double>
(val[i]/wei[i]),i));//把每件物品的单位价值和对应物品的序号插入映射
        }

        int V=Fill(W,bag,wei);
        for(int i=0;i<count;i++){
                cout<<"第"<<i+1<<"件物品取："<<wei[i]<<endl;
        }
        cout<<"背包所装物品的总价值为："<<V;
        return 0;
}
```

通过这个代码，我们可以看到容器的优越性。除了map映射外，还有其他不同的容器如vector，queue，list，stack等。这些容器可以极大地简化代码，尤其是在数据结构方面，它们各自发挥着重要的作用。

第十周第二题

一辆汽车加满油后可行驶n公里。旅途中有k个加油站。设计一个有效算法，指出应

在哪些加油站停靠加油，使沿途加油次数最少。对于给定的n(n <= 5000)和k(k <= 1000)个加油站位置，编程计算最少加油次数。

输入：

第一行有2个正整数n和k，表示汽车加满油后可行驶n公里，旅行中途有k个加油站。另外，第0 个加油站表示出发地（汽车油已加满），第k+1 个加油站为目的地。接下来的1 行中，有k+1 个整数，表示第k个加油站与第k-1 个加油站之间的距离。

输出：

输出编程计算出的最少加油次数。如果无法到达目的地，则输出“无解！”。

样本输入：

（汽车加满油后可行驶n公里和k个加油站:----------**注：此行不显示**）

600 6

（7个加油站彼此之间的距离:----------**注：此行不显示**）

300 200 100 500 200 100 90

样本输出：

最少加油次数：2

三解

```
#include<iostream>

using namespace std;

int Judge(int n,int k,int s[]){
    int num=0;//加油次数
        int temp=n;//剩余油可行走的距离
    for(int i=0;i<k;i++){
                if(temp-s[i]<0){
                        cout<<"无解！";
                        break;//跳出for循环
                }
                else{
                        if(temp-s[i]<s[i+1]){
                                num++;//在这个加油站加油
                                temp=n;//更新油量
                        }
                        else
                                temp-=s[i];//消耗油量
                }
        }
        return num;
}

int main(){
        int n,k;
        cin>>n;
        cin>>k;
        int s[100];
        for(int i=0;i<=k;i++){
                cin>>s[i];
        }
```

```
        cout<<Judge(n,k,s);
        return 0;
}
```

第十二周第一题

在主城站街很久之后，小萌决定不能就这样的浪费时间虚度青春，他打算去打副本。

这次的副本只有一个BOSS，而且BOSS是不需要击杀的，只需要和它比智力.....

BOSS会列出一正整数的序列，由小萌先开始，然后两个人轮流从序列的任意一端取数，取得的数累加到积分里，当所有数都取完，游戏结束。

假设小萌和BOSS都很聪明，两个人取数的方法都是最优策略，问最后两人得分各是多少。

输入

整数个数N

用空格隔开的N个正整数（1 ≤ a[i] ≤ 1000）

输出

只有一行，用空格隔开的两个数，小萌的得分和BOSS的得分。

解

```
#include<iostream>

using namespace std;

void Compare(int player,int boss,int sum){
        if(player>=(sum-player))
            boss=sum-player;
        else{
                boss=player;
                player=sum-player;
        }
        cout<<player<<" "<<boss;
}
```

```
int main(){
        int N;
        int a[1000];
        int player=0;
        int boss=0;
        int sum=0;

        cin>>N;
        if(N%2==0){
                for(int i=0;i<N;i++){
                    cin>>a[i];
                    sum+=a[i];
            }
            for(int i=0;i<N;i+=2)
                    player+=a[i];
        }
        else{
                for(int i=0;i<N;i++){
                    cin>>a[i];
                    sum+=a[i];
            }
            for(int i=0;i<=N;i+=2)
                    player+=a[i];
        }

        Compare(player,boss,sum);
        return 0;
}
```

本题的逻辑为，只要小萌选择一端开始取数，双方的取数序数就被确定，只有奇数或偶数两种情况，因此在取数前比较奇数项和与偶数项和大小即可确定如何取数。

第十二周第三题

一个有N个整数元素的一维数组A[0]、A[1]、…、A[N-1]，求其中**连续的**子数组和的最大值？不需要返回子数组的具体位置、数组中包含：正、负、零整数、子数组不能空。

例如：

int A[] = { 1, -1, 2, 3, -4, 4 };

符合条件的子数组为{1, -1, 2, 3}或{2,3}或{2, 3, -4, 4}，即答案为5；

解

```
#include<iostream>
#include<vector>

using namespace std;

int max_subarray_sum(vector<int>&nums){//将容器作为参数传入函数
    int max_sum=0;//储存最大子列和
    int current_sum=0;//储存当前子列和

    for(int num:nums){//基于范围的for循环，遍历容器中的所有元素
        current_sum+=num;
        if(current_sum<0)//此时本次子列和进行下去无意义，结束本次子列和计算
                current_sum=0;//重置子列和
                if(current_sum>max_sum)
                        max_sum=current_sum;//更新最大子列和
        }

        return max_sum;
}

int main(){
        int k;
        cin>>k;

        vector<int>nums(k);
        for(int i=0;i<k;i++)
                cin>>nums[i];

        cout<<max_subarray_sum(nums);
        return 0;
}
```

本题与第二周第六题几乎完全一样。通过比较动态规划法和穷举法可以发现，前者的时间复杂度明显小于后者。因此随着数据的增多，后者所花费的时间成本会变得越来越难以接受，所以我们在写程序时要学会思考，写完后还需要回头比较，使用更优的方法。

总结

贪心算法将会是编程部分的重点，而动态规划出现的概率较小，即使出现也只会在压轴题。这里列出几点个人的心得，帮助大家写好程序。
1.命名要规范。无论是常量的命名还是函数的命名都需要有一定逻辑，否则一旦程序出现的常量过多，非常容易造成混乱。
2.学会使用函数简化主函数。当一个程序步骤过多时，我们可以通过把复杂的过程放在函数里来简化主函数，这样可以帮助自己和他人迅速理清代码逻辑。
3.学会写注释。注释可以体现自己的思路，即使部分代码写不出来或者写不对，也可以展示自己的思路，避免大片空白。
4.学会自己调试代码。积累平时写代码时所犯的错误，如变量未初始化、数组越界等，学会看懂编译器的报错。除此之外还可以通过断点来查看循环的哪一步出错。
5.思路提醒。在没有好的思路时可以先采用列举法，观察特征，归纳出一般的方法。对于部分题，其思路往往是从数学角度出发分析问题。

遗传算法（GA）

概念

优点：

（1）群体搜索，易于并行化处理
（2）不是盲目穷举，而是启发式搜索
（3）适应度函数不受连续、可微等条件的约束，适用范围很广

应用：

（1）函数优化
（2）组合优化
（3）自动控制
（4）机器人
（5）图像处理

计算框架：

（1）初始化
（2）个体评价
（3）选择运算
（4）交叉运算

（5）变异运算
（6）终止条件判断

设计

编码

通过某种机制把求解问题抽象为由特定符号按一定顺序排成的串。

适应度函数

适应度函数值越大，解的质量越好。

遗传算子

轮盘赌选择方法

（1）计算群体中所有个体的适应度函数值（需要解码）。
（2）利用比例选择算子的公式，计算每个个体被选中遗传到下一代群体的概率。
（3）采用模拟赌盘操作（即生成0到1之间的随机数与每个个体遗传到下一代群体的概率进行匹配）来确定各个个体是否遗传到下一代群体中。

$$P_i = \frac{F_i}{\sum_{i=1}^{n} F_i}$$

⚠ 注意

交叉运算是遗传算法区别于其他进化算法的重要特征。
遗传算法中的变异运算是产生新个体的辅助方法，它决定了遗传算法的局部搜索能力，同时保持种群的多样性。交叉运算和变异运算的相互配合，共同完成对搜索空间的全局搜索和局部搜索。

人工神经网络

定义

人工神经网络是在没有认识清楚人脑的工作机理的情况下，试图从模拟人脑的神经网络结构出发，人为地设计出一种信息处理机，使之具有和人类相似的信息处理能力，从而实现对人类智能的模拟。**其基础为神经元。**

$$y_j(t) = f(\sum_{i=1}^{n} w_{ij}x_i - \theta_j)$$

函数$f(\sum_{i=1}^{n} w_{ij}x_i - \theta_j)$表达了神经元的输入输出特性。往往采用0和1二值函数或Sigmoid函数。Sigmoid函数是一个在生物学中常见的S型的函数，也称为S型生长曲线。Sigmoid函数定义为：

$$S(x) = \frac{1}{1 + e^{-x}}$$

应用

一般分为学习（训练）和工作（计算）两个阶段。

主要的学习算法有：指导学习算法、无指导学习算法和强化学习算法三种。

人工神经网络具有良好的自学习、自适应和自组织能力以及大规模并行、分布式信息存储和处理等特点，这使得它非常适合于处理那些需要同时考虑多个因素的、不完整的、不准确的信息处理问题。

考试形式

- **判断题（10分，共5道）**
- **选择题（20分，共10道）**
- **编程题（60分，共3道）**
- **填空题（10分，共5空）**
  个人认为编程部分大概率会出一道数学问题（可能从第二章前三个里抽），一道贪心算法，可能会建立一个简单的线性结构，小概率出一道动态规划。
  填空部分大概率为数据结构的填空，小概率抽取数学问题的算法填空核心部分。

备考建议

对于数据结构，最好还是从理解其逻辑出发。通过画图的方式可以避免死记硬背。对于线性数据结构，要求掌握其代码实现，切勿眼高手低。