数据结构

线性表

定义

线性表简称表，是n个数据元素的有限序列，线性表中数据元素的个数称为线性表的长度。长度等于零的线性表称为空表。

顺序表

要点：

- 用一段**地址连续**的存储单元
- **依次存储**线性表中的数据元素

举例：数组、向量

存取访问

公式：Loc(ai) = Loc(a1) + (i -1)×c

随机存取：在**O(1)**时间内存取数据元素

顺序表的实现

注：教材中i指的是序号而非下标

```
const int MaxSize = 100;
template <typename DataType>
class SeqList {
public:
    SeqList( );
    SeqList(DataType a[ ], int n);
    ~SeqList( );
    int Length( );
    DataType Get(int i);
    int Locate(DataType x );
    void Insert(int i, DataType x);
    DtaType Delete(int i);
    int Empty( );
    void PrintList( );
private:
    DataType data[MaxSize];
    int length;
};
```

建立

```
template <typename DataType>
SeqList<DataType> :: SeqList(DataType a[ ], int n) {
    if (n > MaxSize) throw “参数非法”;
    for (int i = 0; i < n; i++)
        data[i] = a[i];
    length = n;
}
```

插入

```
template <typename DataType>
void SeqList<DataType> :: Insert(int i, DataType x) {
    if (length == MaxSize) throw "上溢";
    if (i < 1 || i > length + 1) throw "插入位置错误";
    for (int j = length; j >= i; j--) // 从后往前运动
        data[j] = data[j - 1]; //第j个元素存在数组下标为j-1处
    data[i - 1] = x;
    length++;
}
```

插入的平均时间复杂度：**O(n)**

顺序表的优缺点

优点：

- 无需为表示表中元素之间的逻辑关系而增加额外的存储空间
- 随机存取：可以快速地存取表中任一位置的元素

缺点：

- 表的容量难以确定，表的容量难以扩充
- 插入和删除操作需要移动大量元素

单链表

要点：

- 逻辑次序和物理次序**不一定**相同
- 元素之间的逻辑关系用**指针**表示

存储思想：用一组任意的存储单元存放线性表

存储访问

- 头结点：在第一个元素结点之前附设一个类型相同的结点
- 头指针：指向第一个结点的存储地址
- 尾标志：终端结点的指针域为空

单链表的实现

结点定义

```
template <typename DataType>
struct Node {
    DataType data;
    Node *next;
};
```

链表定义

```
template<class T>
class LinkList {
public:
    LinkList();
    LinkList(T a[], int n);
    ~LinkList();
    void PrintList();
    T Get(int i);
    void Insert(int i, T x);
    void Delete(int i);
public:
    Node<T> *first; // 头结点
};
```

遍历

重点之一，需要初始化一个**工作指针**进行操作。链表很多操作需要创造一个工作指针。

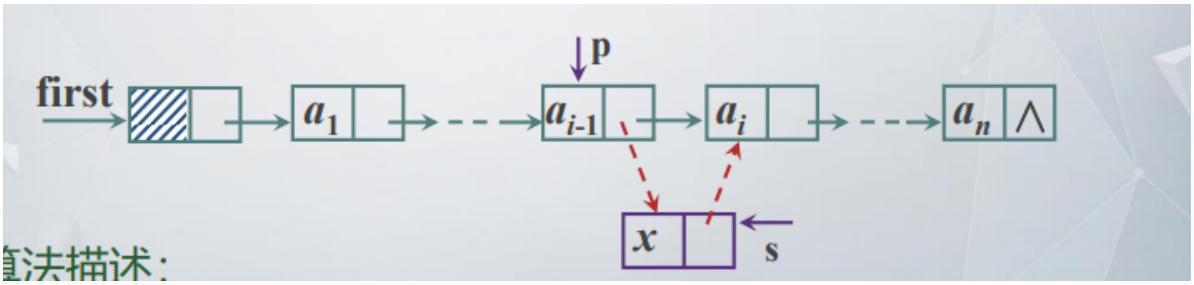
```
template <typename DataType>
void LinkList<DataType> :: PrintList( ) {
    Node<DataType> *p = first->next; //工作指针p初始化
    while (p != nullptr) {
        cout << p->data << "\t";
        p = p->next; //工作指针p后移，注意不能写作p++
    }
    cout << endl;
}
```

按位查找

```
template <typename DataType>
DataType LinkList<DataType> :: Get(int i) {
    Node<DataType> *p = first->next; //工作指针p初始化
    int count = 1; //累加器count初始化
    while (p != nullptr && count < i) {
        p = p->next; //工作指针p后移
        count++;
    }
    if (p == nullptr) throw "查找位置错误";
    else return p->data;
}
```

插入

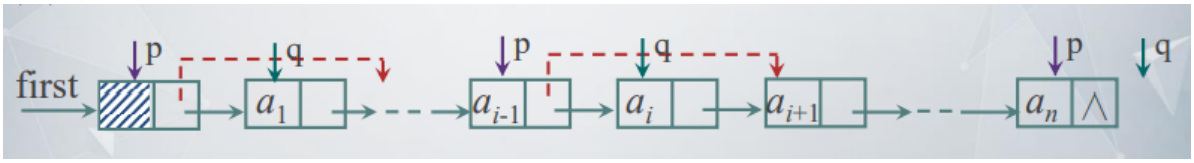
此插入是从第i个元素前面插入。




```
Node<DataType> *s = new Node<Datatype>;
s->data = x;
s->next = p->next;
p->next = s;
```

```
void LinkList<DataType> :: Insert(int i, DataType x) {
    Node<DataType> *p = first, *s = nullptr; //工作指针p初始化
    int count = 0;
    while (p != nullptr && count < i - 1) { //查找第i - 1个结点
        p = p->next; //工作指针p后移
        count++;
    }
    if (p == nullptr) throw "插入位置错误"; //没有找到第i-1个结点
    else {
        s = new Node<DataType>;
        s->data = x; //申请结点s，数据域为x
        s->next = p->next;
        p->next = s; //将结点s插入到结点p之后
    }
}
```

删除




```
q=p->next; x=q->data;
p->next=q->next; delete q;
```

表尾的特殊情况： 虽然被删结点不存在， 但其前驱结点却存在!

```
template <typename DataType>
DataType LinkList<DataType> :: Delete(int i) {
    DataType x;
    int count = 0;
    Node<DataType> *p = first, *q = nullptr; //工作指针p指向头结点
    while (p != nullptr && count < i - 1) { //查找第i-1个结点
        p = p->next;
        count++;
    }
    if (p == nullptr || p->next == nullptr) throw "删除位置错误";
```

```
    else {
        q = p->next;
        x = q->data; //暂存被删结点
        p->next = q->next; //摘链
        delete q;
        return x;
    }
}
```

建立

单链表的建立分为**头插法**和**尾插法**，一起使用这两种方法可以实现链表的倒置。

头插法：插在头结点的后面

first指针不动，不断将其next指向新节点。

```
template <typename DataType>
LinkList<DataType> :: LinkList(DataType a[ ], int n) {
    first = new Node<DataType>;  // 头结点初始化
    first->next = nullptr; //初始化一个空链表
    for (int i = 0; i < n; i++) {
        Node<DataType> *s = nullptr;
        s = new Node<DataType>;
        s->data = a[i];
        s->next = first->next;  // 重要的两步
        first->next = s; //将结点s插入到头结点之后
    }
}
```

尾插法：插在尾结点的后面

为方便在尾结点后面插入结点，设指针rear指向尾结点。

first指针不动，通过尾指针rear不断向后移动达到尾插法效果。

```
template <typename DataType>
LinkList<DataType> :: LinkList(DataType a[ ], int n) {
    first = new Node<DataType>; //生成头结点
    Node<DataType> *r = first, *s = nullptr; //尾指针初始化
    for (int i = 0; i < n; i++) {
        s = new Node<DataType>;
        s->data = a[i];
        r->next = s;
        r = s; //将结点s插入到终端结点之后
    }
    r->next = nullptr; //单链表建立完毕，将终端结点的指针域置空
}
```

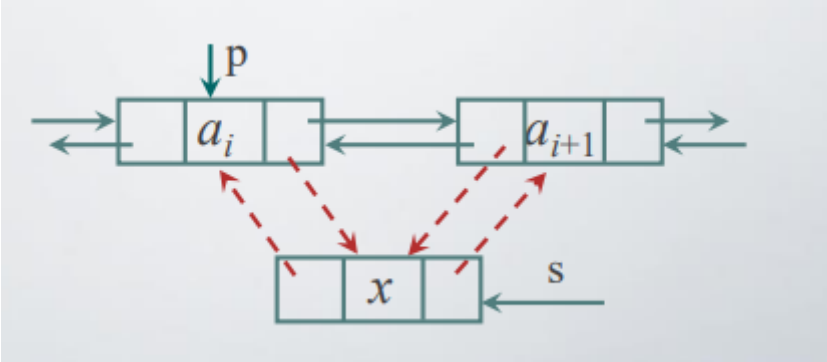
双链表

双链表是为了快速访问前驱元素。

```
template <typename DataType>
sruct DulNode {
    DataType data;
    DulNode< DataType> *prior, *next;
};
```

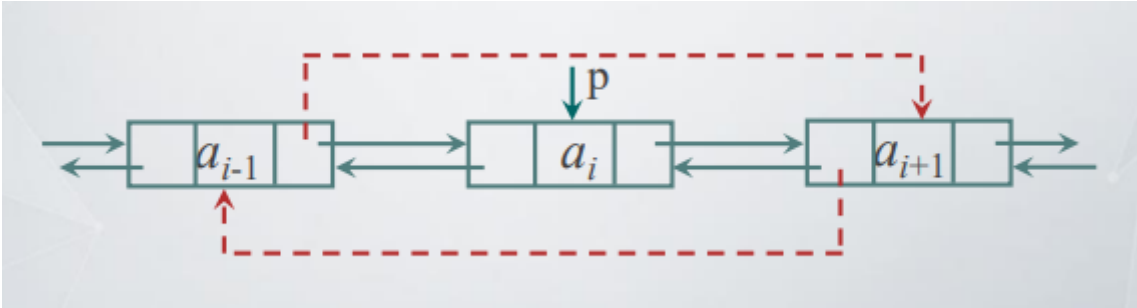
双链表的实现

插入




```
s = new DulNode;
s->prior = p;
s->next = p->next;
p->next->prior = s;
p->next = s;
```

删除




```
(p->prior)->next = p->next;
(p->next)->prior = p->prior
delete p;
```

循环链表

定义：将链表首尾相接

循环单链表：将终端结点的指针由空指针改为指向头结点

循环链表的实现

建立

头插法：

```
template <typename DataType>
LinkList<DataType> :: LinkList(DataType a[ ], int n) {
    first = new Node<DataType>;  // 头结点初始化
    first->next = first; //将头节点的next指向自己 修改处
    for (int i = 0; i < n; i++) {
        Node<DataType> *s = nullptr;
        s = new Node<DataType>;
        s->data = a[i];
        s->next = first->next;  // 重要的两步
        first->next = s; //将结点s插入到头结点之后
    }
}
```

尾插法：

```
template <typename DataType>
LinkList<DataType> :: LinkList(DataType a[ ], int n) {
    first = new Node<DataType>; //生成头结点
    Node<DataType> *r = first, *s = nullptr; //尾指针初始化
    for (int i = 0; i < n; i++) {
        s = new Node<DataType>;
        s->data = a[i];
        r->next = s;
        r = s; //将结点s插入到终端结点之后
    }
    r->next = first; //循环单链表建立完毕，将终端结点的指针域指向头结点 修改处
}
```

插入、查找等

这些操作最主要更改的地方是：

```
p != nullptr => p != first
p->next != nullptr => p->next != first
```

性能对比

按位查找：

- 顺序表：O(1)，随机存取
- 链表：O(n)，顺序存取

插入和删除：

- 顺序表：O(n)，平均移动表长一半的元素
- 链表：不用移动元素，合适位置的指针——O(1)

栈和队列

栈

- 栈：限定仅在一端进行插入和删除操作的线性表
- 栈顶（top）：允许插入和删除的一端称为栈顶
- 栈底（bottom）：另一端称为栈底

特点：后进先出

顺序栈

```
const int StackSize = 10;
template <typename DataType>
class SeqStack {
public:
    SeqStack( );
    ~SeqStack( );
    void Push( DataType x );
    DataType Pop( );
    DataType GetTop( );
    int Empty( );
private:
    DataType data[StackSize];
    int top;
};
```

入栈

```
template <typename DataType>
void SeqStack<DataType> :: Push(DataType x) {
    if (top == StackSize - 1) throw "上溢";
    data[++top] = x;  // 重点 出栈为top--
}
```

链表栈

定义：和单链表类似，但是没有头结点，使用链头作为栈顶。

```
template <typename DataType>
class LinkStack {
public:
    LinkStack( );
    ~LinkStack( );
    void Push(DataType x);
    DataType Pop( );
    DataType GetTop( );
    int Empty( );
private:
    Node<DataType> *top;
};
```

入栈

```
template <typename DataType>
void LinkStack<DataType> :: Push(DataType x) {
    Node<DataType> *s = nullptr;
    s = new Node<DataType>;
    s->data = x; //申请结点s数据域为x
    s->next = top;
    top = s; //将结点s插在栈顶
}
```

队列

- 队列：只允许在表的一端进行插入操作，在另一端进行删除操作
- 队尾：允许**插入**的一端，相应地，位于队尾的元素称为队尾元素
- 队头：允许**删除**的一端，相应地，位于队头的元素称为队头元素

特点：先进先出

顺序队列

使用一个队尾指针rear指向队尾。入队时只需在队尾进行追加元素即可，时间性能为O(1)。但是出队的操作需要移动n-1个元素，即所有元素向前移动，时间性能为O(n)。

也可使用front指针指向队首元素，入队时rear+1，出队时front+1。
且约定：front指向队首元素的前一个元素，rear指向队尾元素的位置。（rear-front=队长）

问题：产生假溢出

循环队列

循环队列将队列看成为首尾相接的循环结构，操作语句为**rear = (rear + 1) % QueueSize**

为了区分判定队满和队空的条件(front==rear)，一般浪费一个数组元素单元，让队头队尾位置相差1，即队满的条件为 (rear + 1) % QueueSize == front

```
const int QueueSize = 100;
template <typename DataType>
class CirQueue {
public:
    CirQueue( );
    ~ CirQueue( );
    void EnQueue(DataType x);
    DataType DeQueue( );
    DataType GetQueue( );
    int Empty( );
private:
    DataType data[QueueSize]; // QueueSize在此
    int front, rear;
};
```

初始化

```
CirQueue<DataType> :: CirQueue( ) {
    front = rear = -1;  // 也可以是QueueSize - 1 即队尾
}
```

判空

```
int CirQueue<DataType> :: Empty( ) {
    if (rear == front) return 1; // 与队满区分
    else return 0;
}
```

判满

```
int CirQueue<DataType> :: Full( ) {
    if ((rear + 1) % QueueSize == front) return 1; // 与队空区分
    else return 0;
}
```

入队

```
template <typename DataType>
void CirQueue<DataType> :: EnQueue(DataType x) {
    if ((rear + 1) % QueueSize == front) throw "上溢";  // 队满
    rear = (rear + 1) % QueueSize; //队尾指针在循环意义下加1
    data[rear] = x; //在队尾处插入元素
}
```

出队

```
template <typename DataType>
DataType CirQueue<DataType> :: DeQueue( ) {
    if (rear == front) throw "下溢"; // 队空
    front = (front + 1) % QueueSize; //队头指针在循环意义下加1
    return data[front]; //读取并返回出队前的队头元素
}
```

链队列

- 链队列：队列的链接存储结构
- 链头作为队头，出队时间为O(1)
- 链尾作为队尾，入队时间为O(n) => 设置队尾指针解决 O(1)

```
template <typename DataType>
class LinkQueue
{
public:
 LinkQueue( );
 ~LinkQueue( );
 void EnQueue(DataType x);
 DataType DeQueue( );
 DataType GetQueue( );
 int Empty( );
private:
 Node<DataType> *front, *rear;
};
```

入队

```
void LinkQueue<DataType> :: EnQueue(DataType x) {
    Node<DataType> *s = nullptr;
    s = new Node<DataType>; //申请结点s
    s->data = x;
    s->next = nullptr;
    rear->next = s;
    rear = s;
}
```

出队

```
DataType LinkQueue<DataType> :: DeQueue( ) {
    DataType x;
    Node<DataType> *p = nullptr;
    if (rear == front) throw "下溢";
    p = front->next;
    x = p->data;
    front->next = p->next;
    if (p->next == nullptr) rear = front; // 判断是否为空
    delete p;
    return x;
}
```

字符串

- 串长：串中所包含的字符个数
- 空串：长度为 0 的串
- 子串：串中任意个连续的字符组成的子序列
- 主串：包含子串的串
- 子串的位置：子串的第一个字符在主串中的序号

BF算法

基本思想：从主串S的第一个字符开始和**模式T**的第一个字符进行比较， 若相等，则继续比较两者的后续字符；否则，从主串S的第二个字符开始和模式T 的第一个字符进行比较，重复上述过程，直到T中的字符全部比较完毕，则说明本趟匹配成功；或S中字符全部比较完，则说明匹配失败。

实现

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

int BF(char S[], char T[]) {
    int i = 0, j = 0;
    while (S[i] != '\0' && T[j] != '\0') {
        if (S[i] == T[j]) {
            i++, j++;
        } else {
            i = i - j + 1; // i的位置加一
            j = 0;
        }
    }
    if (T[j]=='\0')
        return i - j; // 返回数组的位置 非序号
    else
        return -1;
}

int main(){
    char s[] = "I love you";
    char t[] = "No";

    cout<<BF(s,t);
}
```

性能分析

最好情况：不成功的匹配都发生在串 T 的第 1 个字符。

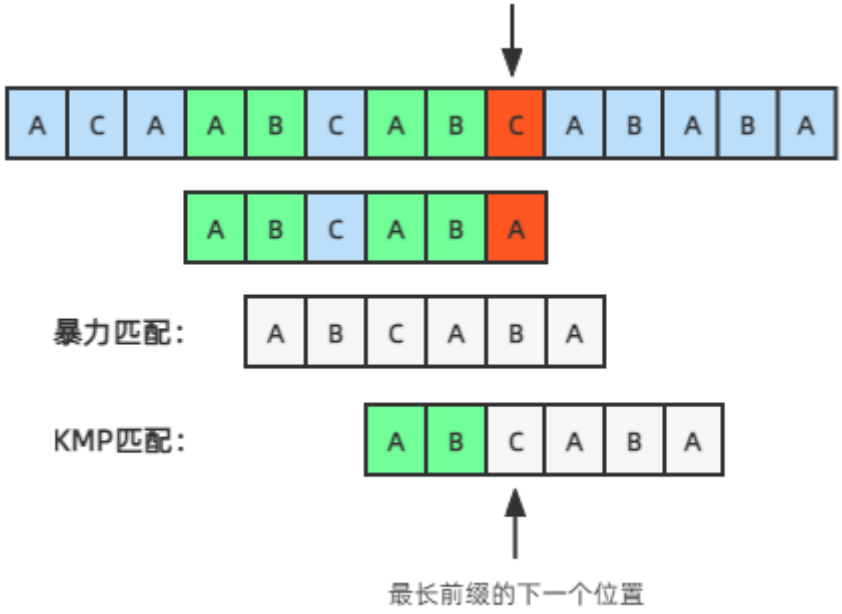
时间复杂度：O(n+m)

最坏情况：不成功的匹配都发生在串 T 的最后一个字符

时间复杂度：O(m*n)

KMP算法

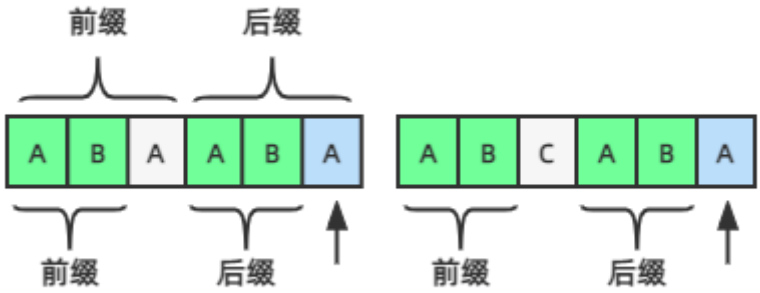
(32条消息) 可能是全网最清晰的KMP算法讲解_卡了个卡的博客-CSDN博客



一个字符串的前n个字符和后n个字符，分别是它的前缀和后缀，如果它的**前缀和后缀相同**，那不就可以对齐了吗？并且我们要找长度最长的那个前后缀才能实现效率最大化，注意：一定是**前缀和后缀相同且长度最长且非字符串本身**，后面简称为**最长前缀。**

KMP的核心思想是：**某个位置匹配失败时，移动到这个位置之前的字符串的最长前缀的下一个字符继续匹配。**

next数组



先看上图左边部分，箭头所指蓝色位置之前的字符串的最长前缀是AB，长度是2，我们对比这个最长前缀的下一个字符A和蓝色位置A相同，那么整个字符串的前后缀就变成了ABA，长度为2+1=3，发现规律了吗？仔细想想看：蓝色的A之前是最长后缀，白色的A之前是最长前缀，而这个最长前后缀是相同的，都是AB，那么如果第三个字符也相同，这三个字符连起来不是也相同吗？长度就是2+1=3！

也就是说，**如果要计算到某个位置字符串的最长前缀长度，我们只需要将这个字符和它之前字符串的最长前缀的下一个字符对比，如果相同，则它的最长前缀长度就是前面字符串的最长前缀长度+1。**

如果不相同呢？看上图右边部分，蓝色的A不等于最长前缀的下一个字符C，我们应该往前回溯，字符C前面的字符串AB的最长前缀长度是0（即后面代码里的k=next[k]），我们就用这个前缀的下一个字符，即第1个字符A和当前字符比较，如果相同，则整个字符串的最长前缀长度为0+1，如果不同，则继续往前回溯，直到第一个字符。也就是，**不断往前回溯，用前面字符串的最长前缀的下一个字符和当前字符对比。**

注意：next数组保存的不是最长前缀的长度，而是对应的下标，也就是长度-1，如果长度是0，则保存`-1`。

next数组保存的是模式串各个位置的字符匹配失败时应该往前回溯的位置，其值等于到该位置字符串的最长前缀的长度-1，应该从它的下一个位置（即这个值+1）开始匹配。

```
int* getNext(char T[]) {
    int length = sizeof(T) / sizeof(char);
    nextArr[0] = -1;
    int k = -1;

    for (int i = 1; i < length; i++) {

        // 当前字符和最长前缀下一个字符不同，则往前回溯
        while (k > -1 && T[i] != T[k + 1]) {
            k = nextArr[k];
        }

        // 当前字符和当前字符前面字符串的最长前缀的下一个字符相同，则k+1
        if (T[i] == T[k + 1]) {
            k++;
        }

        nextArr[i] = k;
    }

    return nextArr;
}
```

没看懂... 期末再看看吧...

To be continue...

树和二叉树

树

树的基本术语

- 结点的度：结点所拥有的**子树的个数**
- 树的度：树中各结点度的**最大值**
- 叶子结点：度为 0 的结点，也称为终端结点
- 分支结点：度不为 0 的结点，也称为非终端结点
- 路径：结点序列 n1 , n2 , ..., nk 称为一条由 n1 至 nk 的路径，当且 仅当满足如下关系：结点 ni是 ni+1 的双亲（1<=i<k）
- 路径长度：路径上经过的边的个数
- 结点所在层数：根结点的层数为 1；对其余结点，若某结点在 第 k 层，则其孩子结点在第 k+1 层
- 树的深度（高度）：树中所有结点的最大层数
- 树的宽度：树中每一层结点个数的最大值

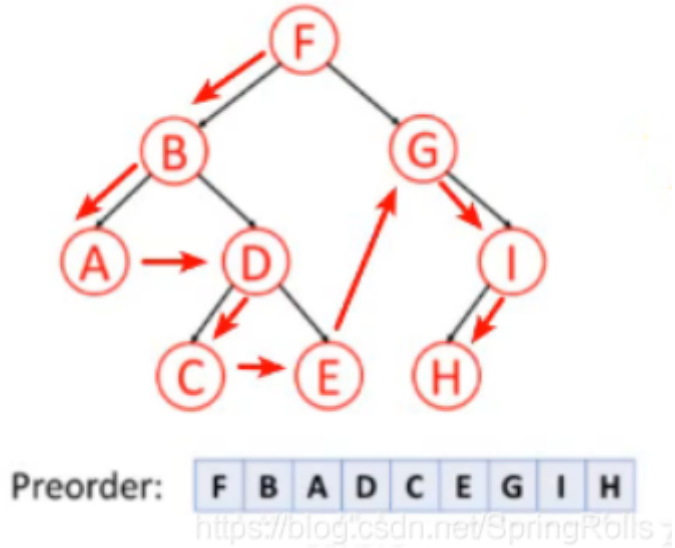
树的性质

- 对于一棵具有n个结点的树，其所有结点的度之和为**n-1** (根节点没有线进入)
- n = n0 + n1 + n2 + n3 + ...　n = B + 1 (B = n1 + 2n2 + 3n3 + ...) n为结点数

树的遍历

前序

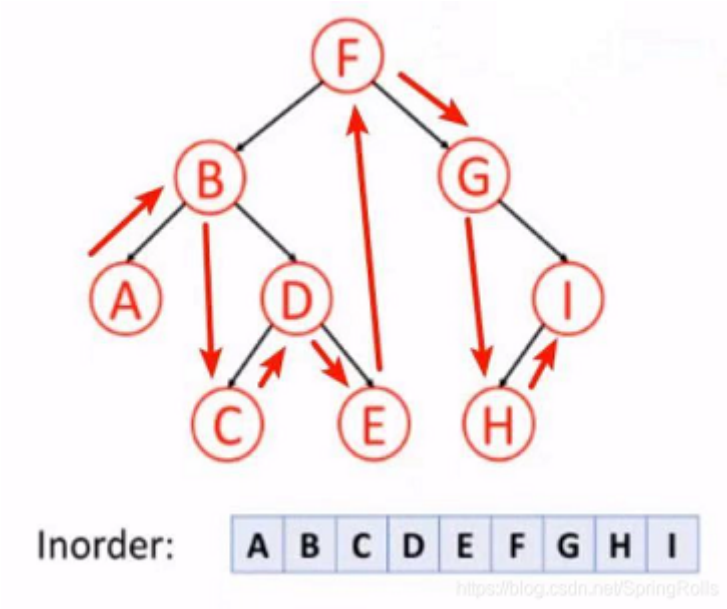
- 访问根结点
- 前序遍历根结点的左子树
- 前序遍历根结点的右子树




```
void preTrav(BiTree* root) {
  if (root) {
    cout << root->key << " ";
    preTrav(root->left);
    preTrav(root->right);
  }
}
```

中序

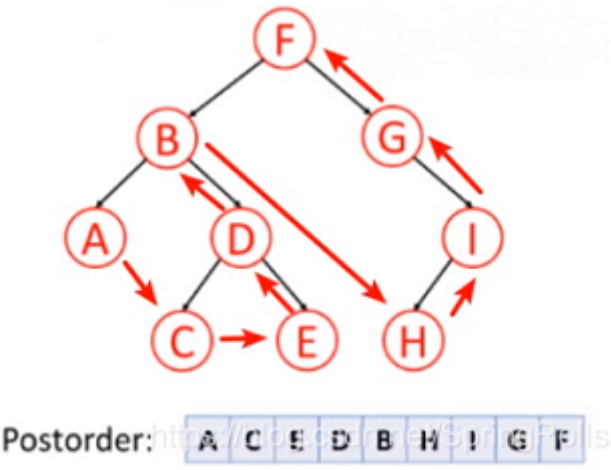
- 中序遍历根结点的左子树
- 访问根结点
- 中序遍历根结点的右子树




```
void midTrav(BiTree* root) {
  if (root) {
    midTrav(root->left);
    cout << root->key << " ";
    midTrav(root->right);
  }
}
```

后序

- 后序遍历根结点的左子树
- 后续遍历根结点的右子树
- 访问根结点




```
void lastTrav(BiTree* root) {
  if (root) {
    lastTrav(root->left);
    lastTrav(root->right);
    cout << root->key << " ";
  }
}
```

根据前序(后序)与中序遍历确定二叉树

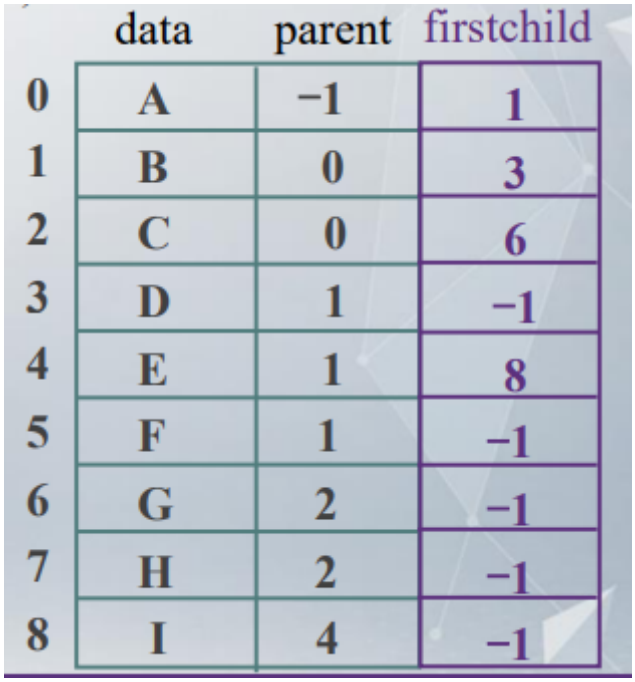
只知道前序和后序遍历是无法确定一个二叉树的。

通过前序的第一个结点可以知道其为根结点，再从中序中找到根结点。中序根结点左边是左子树，右边是右子树。

树的存储结构

双亲表示法

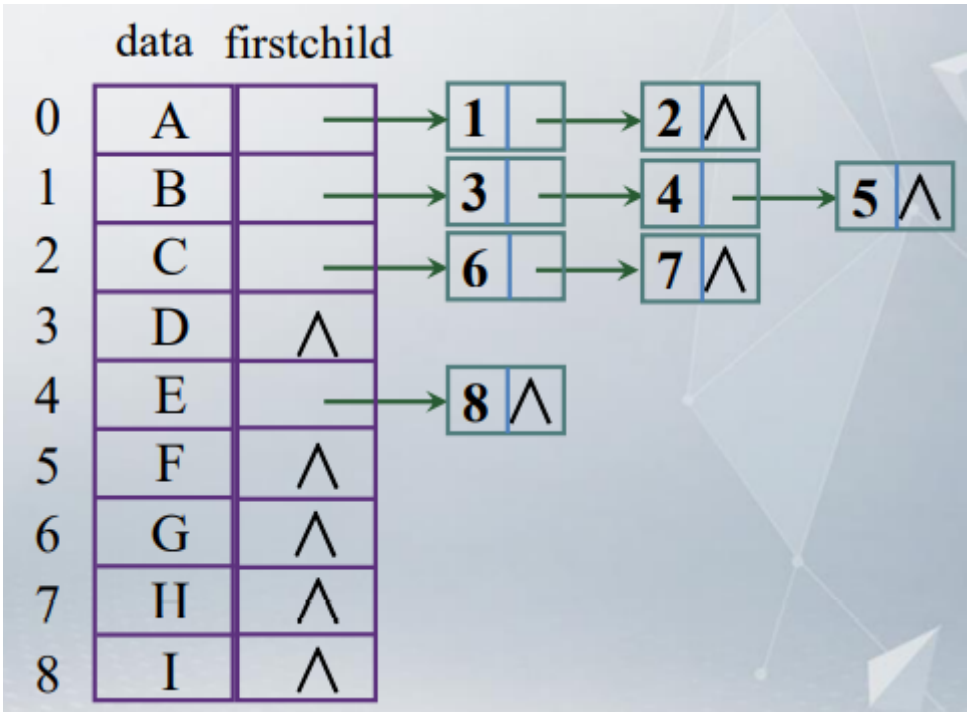
用**一维数组**存储树中各个结点（一般按层序存储）的数据信息以及该结点的双亲在数组中的下标。



	data	parent	firstchild
0	A	−1	1
1	B	0	3
2	C	0	6
3	D	1	−1
4	E	1	8
5	F	1	−1
6	G	2	−1
7	H	2	−1
8	I	4	−1

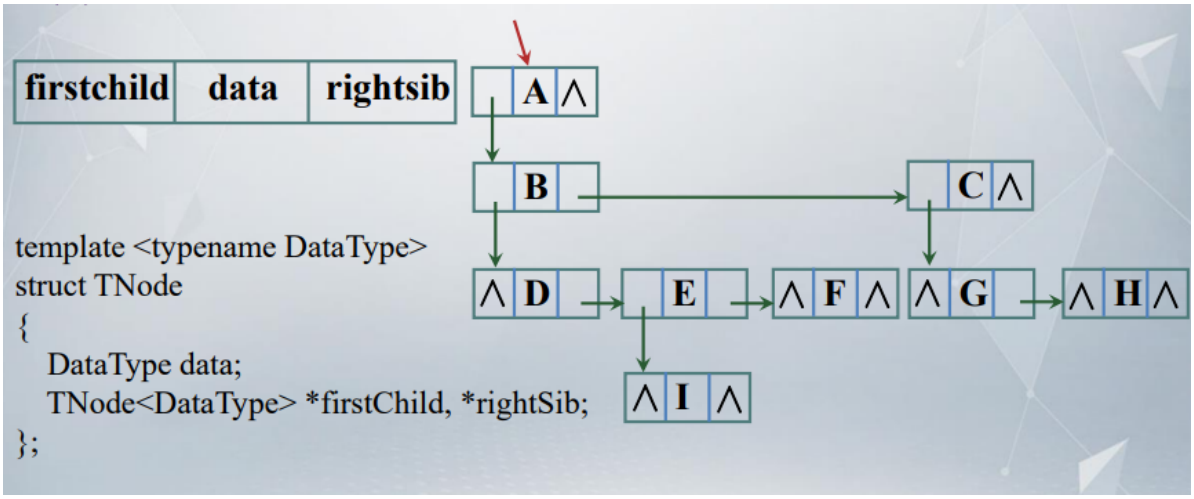
孩子表示法

最佳方式：将结点的所有孩子构成一个**单链表。**



孩子兄弟表示法（二叉链表）

链表中的每个结点包括数据域和分别指向该结点的**第一个孩子**和**右兄弟**的指针。



二叉树

二叉树是度小于等于 2 的树。

- 每个结点最多有两棵子树
- 二叉树是有序的，其次序不能任意颠倒

斜树是树结构的特例，是从树结构**退化成了线性结构。**

- 左斜树：所有结点都只有左子树的二叉树
- 右斜树：所有结点都只有右子树的二叉树

满二叉树：所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上的二叉树。

- 叶子只能出现在最下一层
- 只有度为 0 和度为 2 的结点
- 在同样深度的二叉树中结点个数最多
- 在同样深度的二叉树中叶子结点个数最多

完全二叉树：在满二叉树中，从最后一个结点开始，连续去掉任意个结点得到的二叉树

- 叶子结点只能出现在**最下两层**且最下层的叶子结点都集中在二叉树的**左面**
- 完全二叉树中如果有度为 1 的结点，只可能有一个，且该结点只有**左孩子**
- 深度为 k 的完全二叉树在 k-1 层上一定是满二叉树
- 在同样结点个数的二叉树中，完全二叉树的深度最小

二叉树的性质

- 在一棵二叉树中，如果叶子结点数为 n0，度为 2 的结 点数为 n2，则有:
- 二叉树的**第 i 层**上最多有

  个结点（i≥1）
- 一棵**深度为 k** 的二叉树中，最多有

  个结点
- 具有 n 个结点的完全二叉树的深度为
- 对一棵具有 n 个结点的**完全二叉树**中从 1 开始按层序编号，对于任意的序号为 i（1≤i≤n）的结点（简称结点 i），有:

- 如果 i > 1，则结点 i 的双亲结点的序号为 i/2，否则结点 i 无双亲结点
- 如果 2i≤n，则结点 i 的左孩子的序号为 2i，否则结点 i 无左孩子
- 如果 2i+1≤n，则结点 i 的右孩子的序号为2i+1，否则结点 i 无右孩子

图

定义：由顶点的有穷非空集合和顶点之间边的集合组成

- 无向完全图：无向图中，任意两个顶点之间都存在边
  - 有n×(n-1)/2条边
- 有向完全图：有向图中，任意两个顶点之间都存在方向相反的两条弧
  - 有n×(n-1)条边

- 顶点的度：在**无向图**中，顶点 v 的度是指依附于该顶点的边数，通常记为TD (v)
- 顶点的入度：在**有向图**中，顶点 v 的入度是指以该顶点为弧头（指向）的弧的数目，记为ID (v)
- 顶点的出度：在**有向图**中，顶点 v 的出度是指以该顶点为弧尾（指出）的弧的数目，记为OD (v)
- 在具有 n 个顶点、e 条边的**无向图**中，各顶点度之和等于边数的两倍（2e）
- 在具有 n 个顶点、e 条边的**有向图**中，各顶点入读等于出度等于边数（e）

- 连通分量：非连通图的极大连通子图 **连通分量是对无向图的一种划分**
- 强连通顶点：在有向图中，如果从顶点vi到顶点vj和从顶点vj到顶点vi均有路径，则称顶点vi和vj是强连通的 有向图两个顶点各自出发均联通

图的存储结构

图一般采用邻接矩阵的方式进行存储。

邻接矩阵

```
template <class DataType>
class MGraph {
    public:
        MGraph(DataType a[ ], int n, int e);     //构造函数，建立具有n个顶点e条边的图
        ~MGraph( ) { }                          //析构函数为空
        void DFSTraverse(int v);                  //深度优先遍历图
        void BFSTraverse(int v);                   //广度优先遍历图
    private:
        DataType vertex[MaxSize];              //存放图中顶点的数组
        int arc[MaxSize][MaxSize];              //存放图中边的数组
        int vertexNum, arcNum;                 //图的顶点数和边数
};

/////////////补充1/////////
template <class T>
MGraph<T>::MGraph(T a[], int n, int e) {
    vertexNum = n;
    arcNum = e;
    for (int i = 0; i < vertexNum; i++) // 存储顶点
        vertex[i] = a[i];
    for (int i = 0; i < vertexNum; i++)
        for (int j = 0; j < vertexNum; j++)
```

```
            arc[i][j] = 0;
    for (int k = 0; k < arcNum; k++) {
        int i, j;
        cout << "请输入边的两个顶点的序号：";
        cin >> i >> j;
        arc[i][j] = 1; // 边集为1
        arc[j][i] = 1;
    }
}
```

邻接表

对于稀疏图，邻接矩阵会浪费大量空间。因此采用邻接表的形式存储数据。

- 边表（邻接表）：顶点 v 的所有邻接点链成的单链表
- 顶点表：所有边表的**头指针**和**存储顶点信息的一维数组**
-
```
struct EdgeNode { // 边集结点
    int adjvex; // 邻接点顶点的编号
    EdgeNode *next;
} ;

template <typename DataType>
struct VertexNode { // 顶点结点
    DataType vertex;
    EdgeNode *firstEdge;
};

template <typename DataType>
class ALGraph {
    public:
        ALGraph(DataType a[ ], int n, int e);
        ~ALGraph( );
        void DFTraverse(int v);
        void BFTraverse(int v);
    private:
        VertexNode<DataType> adjlist[MaxSize]; // 顶点数组
        int vertexNum, edgeNum;
};

// 图的建立
template <typename DataType>
ALGraph<DataType> :: ALGraph(DataType a[ ], int n, int e) {
    int i, j, k;
    EdgeNode *s = nullptr;
    vertexNum = n;
    edgeNum = e;
    for (i = 0; i < vertexNum; i++) { //输入顶点信息，初始化顶点表
        adjlist[i].vertex = a[i];
        adjlist[i].firstEdge = nullptr;
    }
    for (k = 0; k < edgeNum; k++) { //依次输入每一条边
        cin >> i >> j; //输入边所依附的两个顶点的编号
        s = new EdgeNode;
        s->adjvex = j; //生成一个边表结点s
        // 头插法
        s->next = adjlist[i].firstEdge; //将结点s插入到第i个边表的表头
```

```
            adjlist[i].firstEdge = s;
        }
    }

    // 图的销毁
    template <typename DataType>
    ALGraph<DataType> :: ~ALGraph( ) {
        EdgeNode *p = nullptr, *q = nullptr;
        for (int i = 0; i < vertexNum; i++) {
            p = q = adjlist[i].firstEdge;
            while (p != nullptr) {
                p = p->next;
                delete q;
                q = p;
            }
        }
    }
```

图的遍历

深度优先搜索 DFS

先对顶点数据进行打印，之后递归搜索。

```
int visited[MaxSize] = {0};
template <class T>
void MGraph<T>::DFSTraverse(int v) {
    cout << vertex[v]; // 打印对应编号顶点数据
    visited[v] = 1; // 设置为已访问
    for (int j = 0; j < vertexNum; j++)
        if (arc[v][j] == 1 && visited[j] == 0)
            DFSTraverse(j);
}
```

```
template <typename DataType>
void ALGraph<DataType> :: DFTraverse(int v) {
    int j;
    EdgeNode *p = nullptr;
    cout << adjlist[v].vertex;
    visited[v] = 1;
    p = adjlist[v].firstEdge;
    while (p != nullptr) {
        j = p->adjvex;
        if (visited[j] == 0)
            DFTraverse(j);
        p = p->next;
    }
}
```

广度优先搜索 BFS

```
template <class T>
void MGraph<T>::BFSTraverse(int v) {
    int Q[MaxSize]; // 定义队列
    int front = -1, rear = -1; // 头指针 尾指针
    cout << vertex[v]; // 打印顶点数据
    visited[v] = 1; // 第一个顶点设置为访问
    Q[++rear] = v; // 第一个顶点进入队列
    while (front != rear) { // 当队列不为空时
        int w = Q[++front]; // 取尾指针指向的顶点
        for (int j = 0; j < vertexNum; j++) { // 搜索取出来的顶点的周围顶点
            if (arc[w][j] == 1 && visited[j] == 0) { // 如果有边且未访问
                cout << vertex[j]; // 打印邻接点数据
                visited[j] = 1; // 邻接点设置为已访问
                Q[++rear] = j; // 邻接点入队
            }
        }
    }
}
```

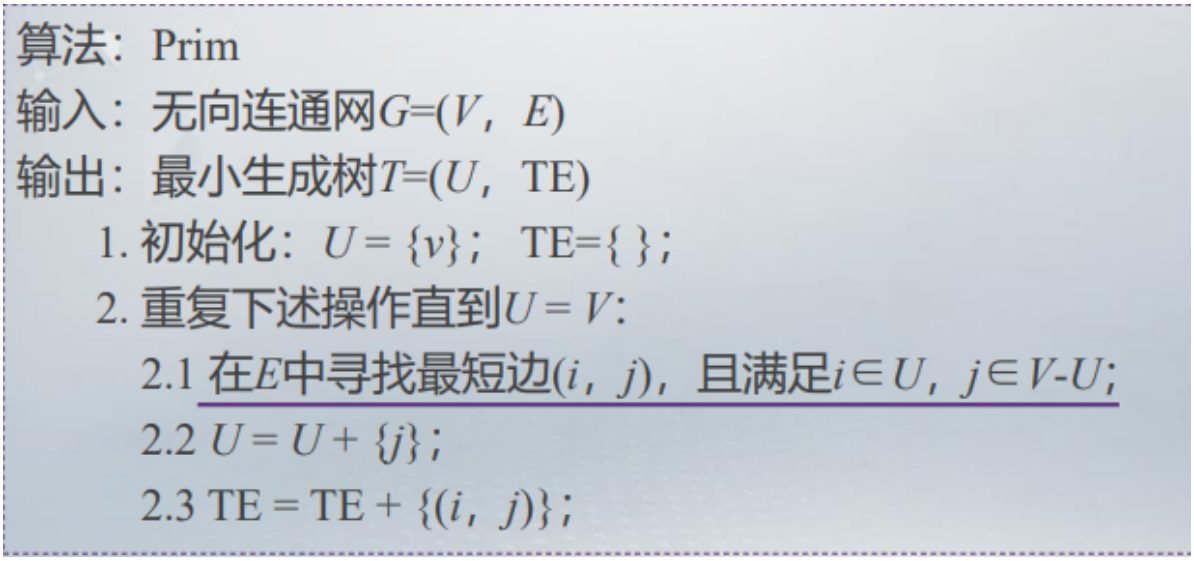
```
template <typename DataType>
void ALGraph<DataType> :: BFTraverse(int v) {
    int w, j, Q[MaxSize];
    int front = -1, rear = -1;
    EdgeNode *p = nullptr;
    cout << adjlist[v].vertex;
    visited[v] = 1;
    Q[++rear] = v;
    while (front != rear) {
        w = Q[++front];
        p = adjlist[w].firstEdge;
        while (p != nullptr) {
            j = p->adjvex; // 取邻接点的编号
            if (visited[j] == 0) {
                cout << adjlist[j].vertex;
                visited[j] = 1;
                Q[++rear] = j;
            }
            p = p->next;
        }
    }
}
```

最小生成树

连通图的生成树是包含图中全部顶点的一个极小连通子图，在图的所有生成树中，代价最小的生成树称为**最小生成树。**

Prim算法

加点法



Prim算法的关键是如何**找到链接U和V-U的最短边来扩充生成树T**

总共需要迭代n-1次

存储结构

- 图的存储结构：需要不断读取任意两个顶点之间边的权值 --> 邻接矩阵
- 候选最短边集：
  - adjvex[n] --> 候选最短边的邻接点
  - lowcost[n] --> 候选最短边的权值
  - adjvex[i] = j; lowcast[i] = w; 候选最短边(i, j) **(j,i)** 的权值为w

顶点的更新过程下：

- lowcast[i] = min{lowcast[i], 边(i, j)的权值} **非联通状态赋值为无限**
- adjvex[i] = j (如果边(i,j)的权值 < lowcast[i])

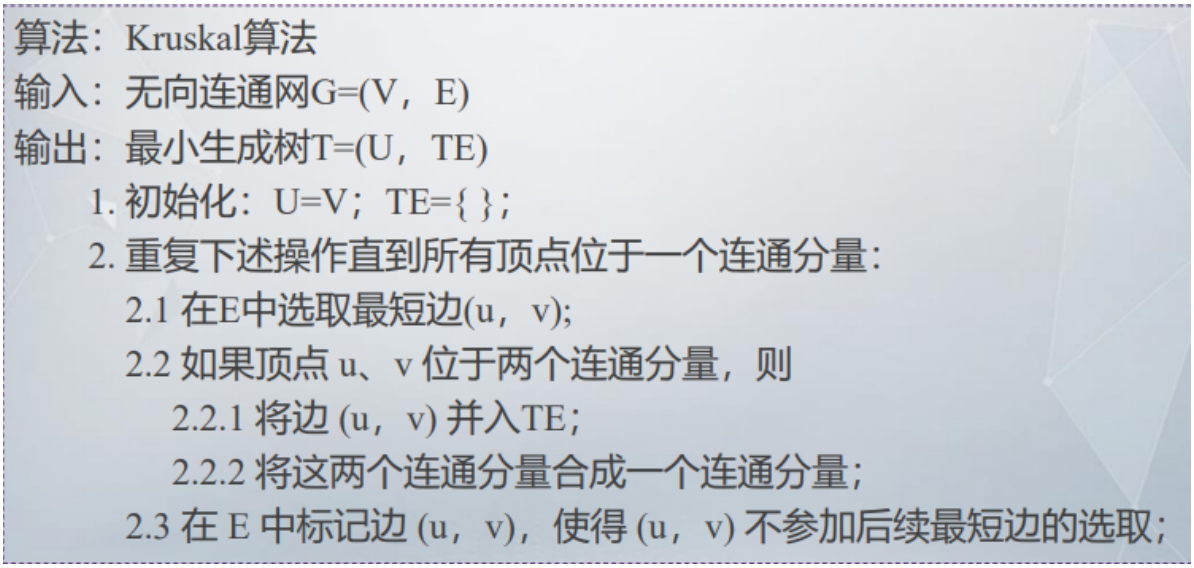
```
int MinEdge(int lowcast[], int vertexNum) {
    int minest = 0x7fffffff;
    int index = -1;
    for (int i = 0; i < vertexNum; i++) {
        if (lowcast[i] != 0 && lowcast[i] < minest) {
            minest = lowcast[i];
            index = i;
        }
    }
    return index;
}

template <class DataType>
void MGraph<DataType>::Prim(int v) {
    int adjvex[MaxSize], lowcast[MaxSize];
    for (int i = 0; i < vertexNum; i++) {
        lowcast[i] = arc[i][v];
        adjvex[i] = v;
    }
    lowcast[v] = 0; // 将顶点v加入集合U
    for (int k = 1; k < vertexNum; k++) { // 总共迭代n-1次
```

```
            int j = MinEdge(lowcast, vertexNum);
            cout << "<" << j << "," << adjvex[j] << ">" << lowcast[j] << endl;
            lowcast[j] = 0; // 顶点j加入集合U
            for (int i = 0; i < vertexNum; i++) { // 调整辅助数组
                if (arc[i][j] < lowcast[i] && arc[i][j] != 0) { // 用无限会更好
                    lowcast[i] = arc[i][j];
                    adjvex[i] = j;
                }
            }
        }
    }
```

Kruskal算法

加边法



Kruskal算法中心思想是：将T中的顶点各自构成一个联通分量，然后按照**边的权值由小到大的顺序，依次考察边集E中的各条边**。若被考察的两个顶点属于两个不同的连通分量，则将此边加入到TE中，同时把两个连通分量链接为一个连通分量。若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路。

存储结构

- 图的存储结构：边集数组表示法

- 连通分量的顶点所在的集合：由于涉及到集合的查找和合并等操作，考虑采用查并集来实现。查并集是将集合中的元素组织成树的形式。**当合并两个集合时，只需要将一个集合的根结点作为另一个集合根结点的孩子。**
- 查并集的存储结构：双亲表示法 parent[n]
  - 设元素parent[i]表示顶点i的父节点的下标
  - 初始时令parent[i] = -1，表示顶点没有父节点
  - 对于边(u，v)，设vex1和vex2分别表示两个顶点所在集合的根的下标，如果vex1≠vex2，则顶点u和顶点v一定位于两个集合
  - 令parent[vex2] = vex1即可合并两个集合

```
#include<iostream>
using namespace std;
const int MaxVertex = 10;
const int MaxEdge = 100;

struct Edge{
    int from;
    int to;
    int weight;
};

template <typename DataType>
class EdgeGraph {
    public:
        EdgeGraph(DataType a[ ], int n, int e);
        ~EdgeGraph( );
        void Kruskal( );
    private:
        int FindRoot(int parent[ ], int v);
        DataType vertex[MaxVertex];
        struct Edge edge[MaxEdge];
        int vertexNum, edgeNum;
};

template <typename DataType>
void EdgeGraph<DataType>::Kruskal() {
    int vex1, vex2;
```

```
    int parent[vertexNum]; // 使用父节点表示法存储查并集
    for (int i = 0; i < vertexNum; i++) {
        parent[i] = -1;
    }
    for (int i = 0, num = 0; num < vertexNum - 1; i++) {
        // 会有很多边 而且有一部分边是不满足条件的
        vex1 = FindRoot(parent, edge[i].from);
        vex2 = FindRoot(parent,edge[i].to);
        if(vex1!=vex2){
            cout<<"("<<edge[i].from<<","<<edge[i].to<<")"<<edge[i].weight;
            parent[vex1] = vex1; // 合并边集
            num++; // 加入图中的边的个数 最后加到n-1
        }
    }
}

template <typename DataType>
int EdgeGraph<DataType>::FindRoot(int parent[], int v){
    int t = v;
    while(parent[t]>-1) // 不断寻找父节点
        t = parent[t];
    return t; // 即使没有找到父节点也不会返回-1 而是返回自身的下标
}
```

最短路径

定义：最短路径是指两顶点之间经历的边上权值之和最短的路径

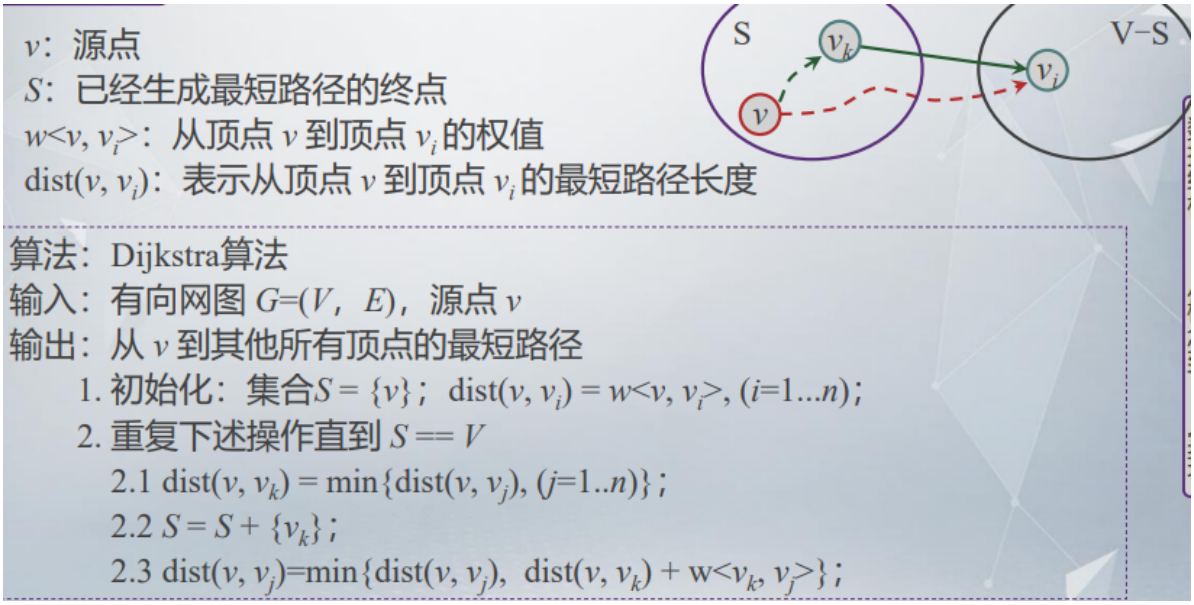
Dijkstra算法

基本思想：设置一个集合S存放已经找到最短路径的顶点，S的初始状态只包含源点v，对vi∈V-S，假设从源点v到vi的有向边为最短路径。以后每求得一条最短路径v, ..., vk，就将vk加入集合S中，并将路径v, ..., vk , vi与原来的假设相比较，取路径长度较小者为最短路径。重复上述过程，直到集合V中全部顶点加入到集合S中。

中心思想：从一点出发，不断寻找从原点到可扩展点的最短路径

两个规律：

- 最短路径是按路径长度递增的次序产生的
- 最短路径上只可能经过已经产生终点的顶点



存储结构

- 图的存储结构：邻接矩阵
- 最短路径存储结构
  - dist[n]：存储当前最短路径的长度 从原点到点n的最短长度 **整型数组**
  - path[n]：存储当前的最短路径，即顶点序列 **字符串数组**

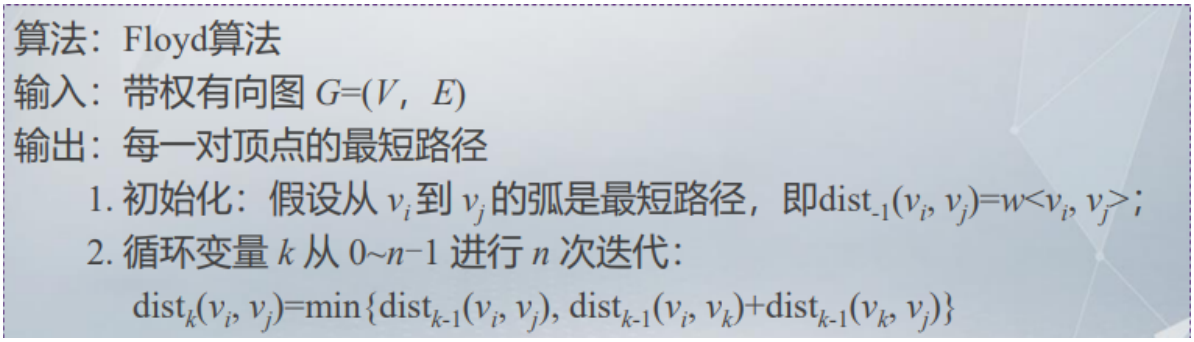
```
int isVisited[MaxSize];
int Min(int dist[], int vertexNum) {
    int minest = 255;
    int index = -1;
    for (int i = 0; i < vertexNum; i++) {
        if (dist[i] != 0 && dist[i] < minest) {
            minest = dist[i];
            index = i;
        }
    }
    return index;
}

template<class T>
void MGraph<T>::Dijkstra(int v) {
    int num, dist[MaxSize];
    string path[MaxSize];
    int infinite = 255;  //  假设255为不相连
    for (int i = 0; i < vertexNum; i++) {
        dist[i] = arc[v][i];
        if (dist[i] != infinite) { // 当连通时
            path[i] = "";
            path[i] = path[i] + vertex[v] + vertex[i];
        } else {
            path[i] = "";
        }
    }
    for (num = 1; num < vertexNum; num++) { // 总共有n-1条边
        int k = Min(dist, vertexNum); // 在dist数组中寻找最小值并返回下标
        cout << path[k] <<","<<dist[k]<<";"<<endl;
        for (int i = 0; i < vertexNum; i++) {
            if (dist[i] > dist[k] + arc[k][i]) { // 到达下标为i的点的距离
```

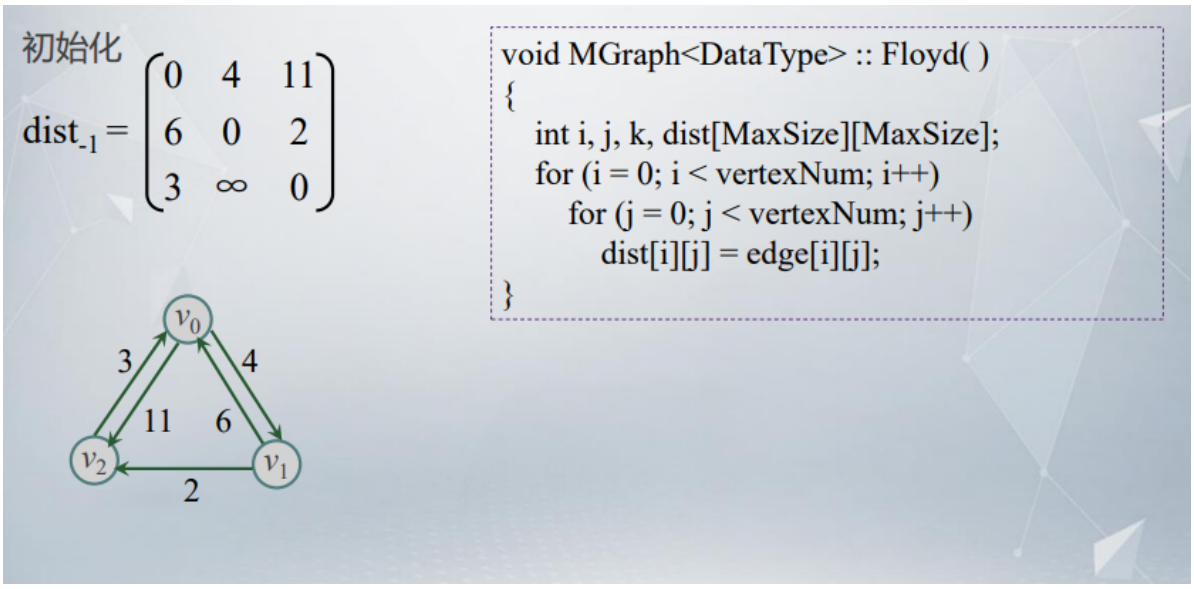
```
                dist[i] = dist[k] + arc[k][i];
                path[i] = path[k] + vertex[i];
            }
        }
        dist[k] = 0;
    }
}
```

Floyd算法

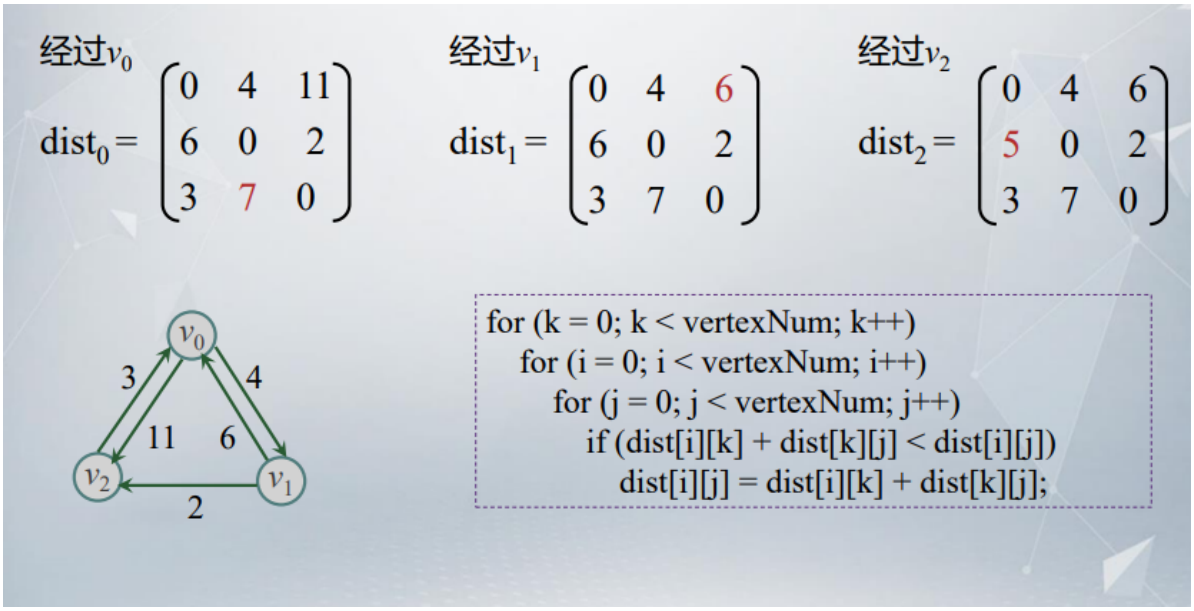
Floyd算法用于求每一对顶点之间的最短路径问题。



初始化



运行中



最外层是k个顶点进行遍历。更换的条件是如果当前大小大于进过中途点则更新。

```
template<class DataType>
void MGraph<DataType> :: Floyd( ) {
    int i, j, k, dist[MaxSize][MaxSize];
    for (i = 0; i < vertexNum; i++) {
        for (j = 0; j < vertexNum; j++)
            dist[i][j] = edge[i][j];
    }
    for (k = 0; k < vertexNum; k++)
        for (i = 0; i < vertexNum; i++)
            for (j = 0; j < vertexNum; j++)
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
}
```

AOV网

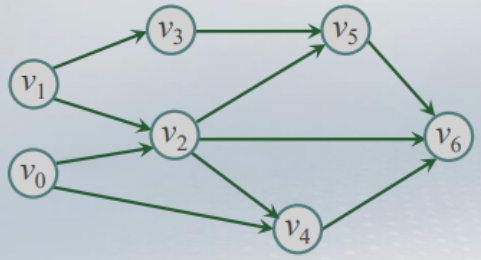
定义：顶点表示活动的网，在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系

AOV网中出现回路意味着活动之间的优先关系是矛盾的

拓扑排序

设有向图G=(V，E)具有 n 个顶点，则顶点序列 v0 , v1 ，..., vn-1 称为一个拓扑序列，当且仅当满足下列条件：若从顶点 vi 到 vj有一条路径，则在顶点序列中顶点 vi 必在顶点 vj之前

算法：拓扑排序TopSort
输入：AOV网 $G=(V, E)$
输出：拓扑序列
1. 重复下述操作，直到输出全部顶点，或AOV网中不存在没有前驱的顶点
1.1 从AOV网中选择一个没有前驱的顶点并且输出；
1.2 从AOV网中删去该顶点，并且删去所有以该顶点为尾的弧；



拓扑序列：v_0 v_1 v_2 v_3 v_4 v_5 v_6

存储结构

- 图的存储结构：邻接表
- 如何求顶点入度？在顶点表中增加入度域
- 如何查找没有前驱的顶点？设置栈或队列

具体过程：

- 将入度为0的顶点入栈
- 弹出并输出栈顶
- 重复以上过程直至栈空

```
void TopSort(){
    int count = 0;
    int S[MaxSize], top = -1;
    EdgeNode *p = nullptr;
    for(int i=0;i<vertexNum;i++) // 扫描顶点表
        if(adjlist[i].in==0) // 入读为0压入栈
            S[++top] = i;
    while(top!=-1){
        int j = S[top--]; // 取出顶点
        cout<<adjlist[i].vertex;
        count++;
        p = adjlist[j].firstEdge; // 工作指针p初始化
        while(p!=nullptr){
            int k = p->adjvex;
            adjlist[k].in--;
            if(adjlist[k].in==0)
                S[++top] = k;
            p = p->next;
        }
    }
    if(count<vertexNum)
        cout<<"有回路";
}
```

关键路径

关键路径：AOE网中从源点到终点的最长路径

AOE网具有以下两个性质：

- 只有在进入某顶点的各活动都已经结束，该顶点所代表的事件才能发生
- 只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始

设带权有向图 G=(V，E)含有 n 个顶点 e 条边，设置 4 个一维数组：

- 事件的最早发生时间 ve[n]
  - ve[0] = 0
  - ve[k] = max{ve[j] + len} (∈p[k])
- 事件的最迟发生时间 vl[n]
  - vl[n-1] = ve[n-1]
  - vl[k] = min{vl[j]-len}（∈s[k]）s[k]： 所有从 vk发出的有向边
- 活动的最早开始时间 ae[e]
  - ae[i] = ve[k]
  - al[i] = vl[j] - len
- 活动的最晚开始时间 al[e]
  - ae[i] = ve[k]
  - al[i] = vl[j] - len

查找技术

关键码：可以标识一个记录的某个数据项

键值：关键码的值

静态查找： 不涉及插入和删除操作的查找

动态查找： 涉及插入和删除操作的查找。当查找不成功时，要插入被查找的记录。

静态查找

顺序查找

改进：通过将第一个元素设置为哨兵进而改良查找速度

```
int LineSearch :: SeqSearch2(int k) {
    int i = n;
    data[0] = k;
    while (data[i] != k)
        i--;
    return i;
}
```

- 查找的时间复杂度为**O(n)**
- 不考虑元素的有序性，插入删除的时间复杂度为**O(1)**

折半查找

折半查找（对半查找、二分查找）的基本思想：在有序表（假设为递增）中， 取中间记录作为比较对象，**若给定值与中间记录相等，则查找成功**；若给定值小于中间记录，则在有序表的左半区继续查找；若给定值大于中间记录，则在有序表的右半区继续查找。不断重复上述过程，直到查找成功，或查找区域无记录，查找失败

非递归算法

```
int LineSearch::BinSearch1(int k) {
    int mid, low = 1, high = n;
    while (low <= high) {
        mid = (low + high) / 2;
        if (data[mid] == mid)
            return mid;
        else if (k < data[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
}
```

递归算法

```
int LineSearch::BinSearch2(int low, int high, int k) {
    if (data[mid] == k)
        return mid;
    else if (k < data[mid])
        BinSearch2(low, mid - 1, k);
    else
        BinSearch2(mid + 1, high, k);
}
```

- 查找时间复杂度为**O(logn)**
- 插入删除的时间复杂度为**O(n)**

动态查找

二叉排序树

定义：或者是一棵空的二叉树，或者是具有下列性质的二叉树

- 若它的左子树不空，则**左子树上所有结点的值均小于根结点的值**
- 若它的右子树不空，则**右子树上所有结点的值均大于根结点的值**
- 它的左右子树也都是二叉排序树

查找的过程：

- 若 bt 是空树，则查找失败
- 若k = bt->data，则查找成功
- 若k < bt->data，则在 bt 的左子树上查找
- 若k > bt->data，在 bt 的右子树上查找

查找

```
BiNode<int>* BiSortTree::SearchBST(BiNode<int>*bt, int k){
    if(bt== nullptr)
        return nullptr;
    if(bt->data == k)
        return bt;
    else  if(k<bt->data)
        return SearchBST(bt->lchild, k);
    else
        return SearchBST(bt->rchild, k);
}
```

插入

```
BiNode<int>* BiSortTree::InsertBST(BiNode<int>*bt, int x) {
    if (bt == nullptr) {
        BiNode<int>*s = new BiNode<int>;
        s->data = x;
        bt = s;
        return bt; // 返回叶子结点的地址
    } else if (x < bt->data)
        bt->lchild = InsertBST(bt->lchild, x);
    else
        bt->rchild = InsertBST(bt->rchild, x);
}
```

构造

```
BiSortTree::BiSortTree(int a[ ], int n) {
    root = nullptr;
    for (int i = 0; i < n; i++)
        root = InsertBST(root, a[i]);
}
```

删除

二叉排序树的删除比较麻烦。删除后的二叉树仍要保持二叉排序树的特性。

不失一般性，设待**删除结点为p，其双亲结点为f**，且 **p 是 f 的左孩子**

被删除的结点是叶子结点

操作：将双亲结点中相应指针域的值改为空

```
f->lchild = nullptr;
```

被删除的结点只有左子树或者只有右子树

操作：将双亲结点中相应指针域指向被删除结点的左子树（或右子树）

```
f->lchild = p->rchild;
```

被删除的结点既有左子树也有右子树

操作：找到大于p的最小者或者小于p的最大者

以其左子树中的最大值结点替换之，然后再删除该结点（delete的是查找到最大结点）

特殊情况：左子树中的最大值 结点是被删结点的孩子

```
if (p == par)
 par->lchild = s->lchild;
```

左子树的右链一定是最大的

右子树的左链一定是最小的

```
void BiSortTree::DeleteBST(BiNode<int> *p, BiNode<int> *f ) {
    if ((p->lchild == nullptr) && (p->rchild == nullptr)) { //p为叶子
        f->lchild = NULL;
        delete p;
        return;
    }
    if (p->rchild == nullptr) { //p只有左子树
        f->lchild = p->lchild;
        delete p;
        return;
    }
    if (p->lchild == nullptr) { //p只有右子树
        f->lchild = p->rchild;
        delete p;
        return;
    }
    BiNode<int> *par = p, *s = p->lchild; /*p的左右子树均不空*/
    while (s->rchild != nullptr) { /*查找左子树的最右下结点*/
        par = s;
        s = s->rchild;
    }
    p->data = s->data;
    // 特殊情况：左子树中的最大值结点是被删结点的孩子
    if (par == p) par->lchild = s->lchild;
    else par->rchild = s->lchild;
    delete s;
}
```

平衡二叉树

平衡因子：该结点的**左子树的深度减去右子树的深度**

平衡二叉树：或者是一棵空的二叉排序树，或者是具有下列性质 的二叉排序树：

- 根结点的左子树和右子树的**深度最多相差 1**
- 根结点的左子树和右子树也都是平衡二叉树

最小不平衡子树：以距离插入结点最近的、且**平衡因子的绝对值大于 1** 的结点为根的子树

一旦产生最小不平衡子树，就进行调整 => 不影响其他结点

对于新结点插入的位置分为LL、RR、LR、RL四种类型（只走两步）。其中前两种调整一次，后两种调整两次。根据扁担原理和旋转优先进行调整。

- **扁担原理**：将根结点看成是扁担中肩膀的位置
- **旋转优先**：旋转下来的**结点作为新根结点的孩子**（撞到另外一棵树的结点转化到另一颗树上）

LL型

结点x插在了根结点的左孩子的左子树上。

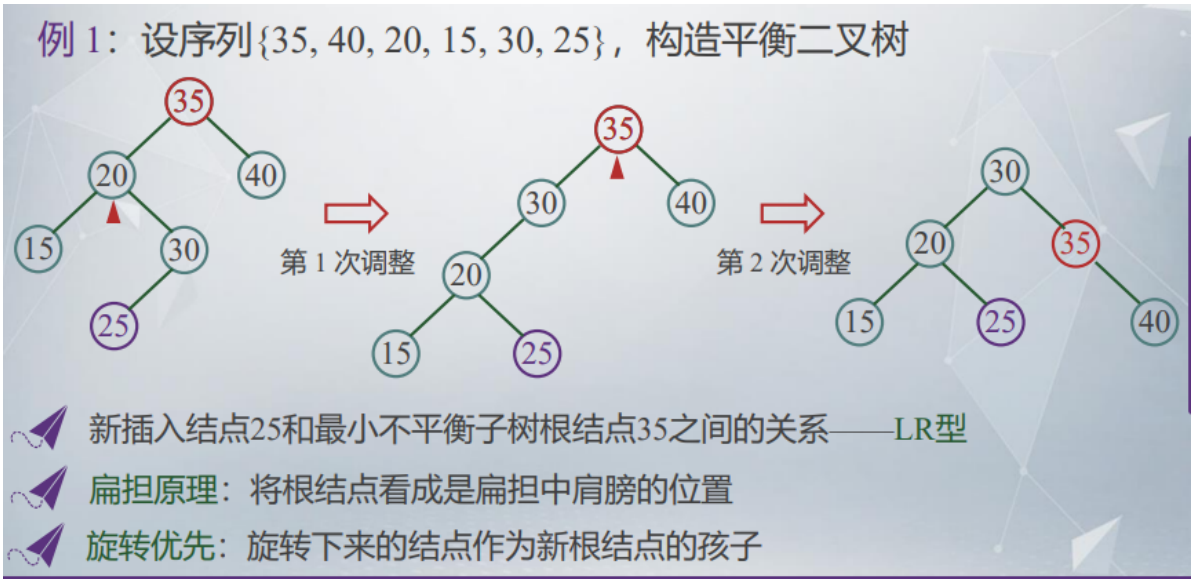
顺时针旋转一次

RR型

逆时针旋转一次

LR型

结点x插在了根结点35的左孩子的右子树上。



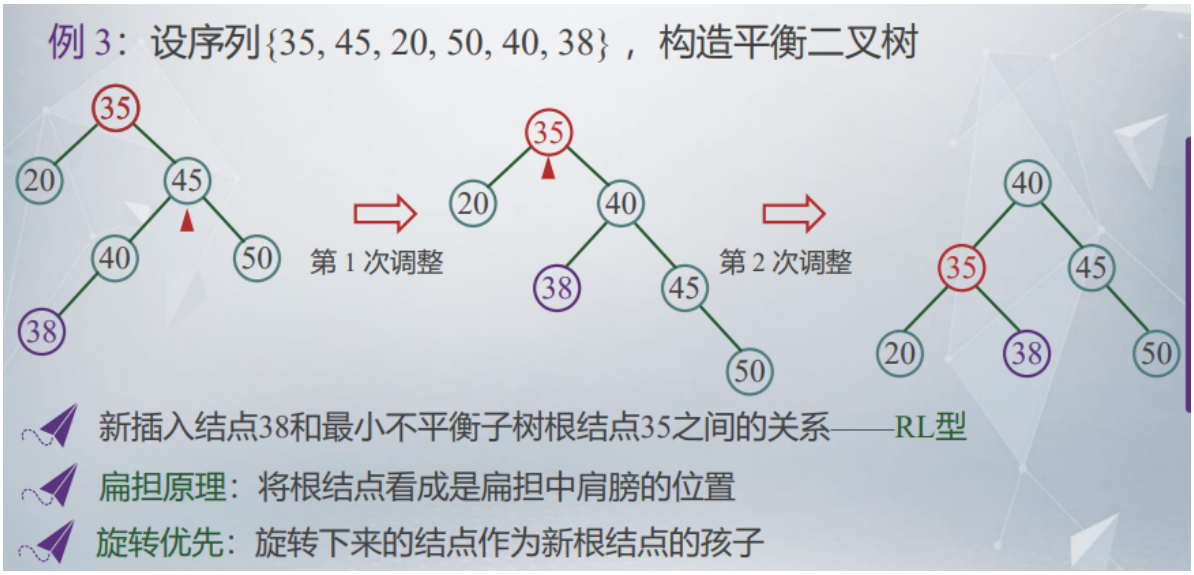
第一次调整：

根结点不动，先逆时针旋转**调整根结点的左子树**。支撑点由20调整到30。

第二次调整：

将支撑点由35调整顺时针到30。图中右子树的子树没有撞击到其他子树，无需调整位置。

RL型



结点x插在了根结点35的右孩子的左子树上。

第一次调整：

根结点不动，先顺时针旋转**调整根结点的右子树**。支撑点由45调整到40。

第二次调整：

将支撑点由35调整逆时针到40。对于子树的调整，还可以看大小关系进行调整。

散列查找（哈希表）

散列的基本思想：在记录的关键码和存储地址之间建立一个确定的对应关系，通过计算得到待查记录的地址。

- 散列表：采用散列技术存储查找集合的连续存储空间
- 散列函数：将关键码映射为散列表中适当存储位置的函数
- 散列地址：由散列函数所得的存储地址

散列函数

直接定址法

散列函数是关键码的线性函数，即：

平方取中法

对关键码平方后，按散列表大小，取中间的若干位作为散列地址。

除留余数法

适用于：最简单、最常用，不要求事先知道关键码的分布

处理冲突

开放定址法

用开放定址法处理的冲突得到的散列表叫做**闭散列表**

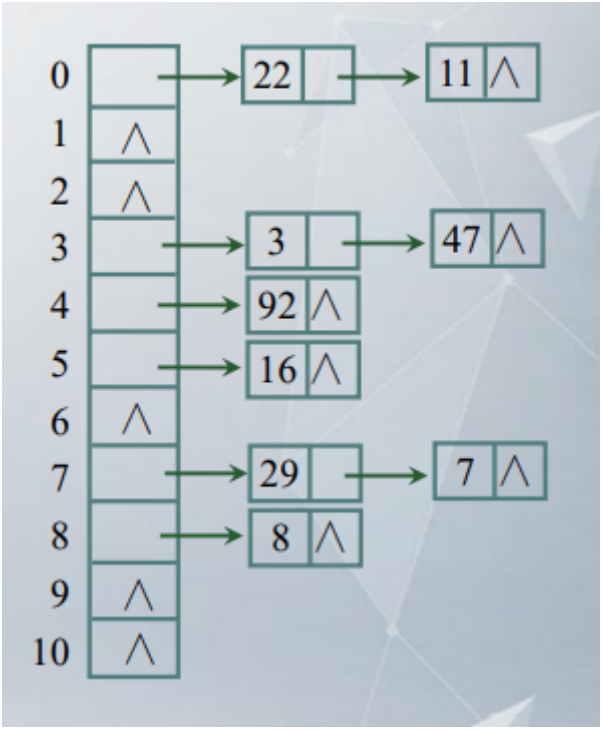
```
// 查找算法
int HashTable1 :: Search(int k) {
    int i, j = H(k); //计算散列地址
    i = j; //设置比较的起始位置
    while (ht[i] != 0) {
        if (ht[i] == k) return i; //查找成功
        else i = (i + 1) % m; //向后探测一个位置
    }
    return -1; //查找失败
}
```

拉链法

同义词子表：所有散列地址相同的记录构成的单链表。

开散列表：用拉链法处理冲突得到的散列表。

开散列表中存储同义词子表的**头指针**，开散列表不会出现堆积现象




```
Node<int>* ht[MaxSize]; // 开散列表
// 构造算法
j = H(k); // 哈希函数得出来的key
Node<int> *p = ht[j];
while (p != nullptr) {
    if (p->data == k) break;
    else p = p->next;
}
if (p == null) {
    q = new Node<int>;
```

```
    q->data = k;
    q->next = ht[j];
    ht[j] = q;
}
```

```
// 查找算法
Node<int> * HashTable2 :: Search(int k) {
    int j = H(k);
    Node<int> *p = ht[j];
    while (p != nullptr) {
        if (p->data == k) return p;
        else p = p->next;
    }
    return nullptr;
}
```

排序技术

如果在排序前，ri的位置在rj之间，排序后位置**相对性依然不变**，则称这种排序算法是**稳定**的；否则称为**不稳定**的。

插入排序

直接插入排序

基本思想：依次将待排序序列中的每一个记录插入到已排好序的序列中，直到全部记录都排好序。

```
void InsertSort(int r[ ], int n) {
    int i,j;
    for (i = 1; i < n; i++) { // 对于长度为n的数组 排序执行n-1次
        int temp = r[i];
        // 与待查元素之前的元素进行比较
        for (j = i - 1; j >= 0 && temp < r[j]; j--) {
            r[j + 1] = r[j];
        }
        r[j + 1] = temp;
    }
}
```

时间复杂度为O(n^2)

希尔排序

希尔排序是对直接插入排序的一种改进。

改进点：

- 若待排序记录基本有序，直接插入排序的效率很高
- 待排序记录个数较少时效率也很高

基本思想：将整个待排序序列分割成若干个子序列，在子序列内部分别进行插入排序，待到序列整体有序的时候再对全体进行直接插入排序。

相距d非常重要。对相距d的元素进行排序。

```
void ShellSort(int r[ ], int n) {
    int i, j, d;
    for (d = n / 2; d >= 1; d = d / 2) { // 增量为d进行直接插入排序
        for (i = d; i < n; i++) { // 原来为1的地方改成d
            int temp = r[i];
            for (j = i - d; j >= 0 && temp < r[j]; j = j - d) {
                r[j + d] = r[j];
            }
            r[j + d] = temp;
        }
    }
}
```

交换排序

冒泡排序

基本思想：两两比较相邻记录，如果反序则交换，知道没有反序记录为止。

```
void bubble_sort(int array[], int length) {
    // 定义临时变量
    int temp;
    // 循环遍历数组
    for (int i = 0; i < length; i++) {
        for (int j = 0; j < length - i - 1; j++) { //
            // 如果相邻两个元素的顺序错误
            if (array[j] > array[j + 1]) {
                // 交换位置
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}
```

上述的冒泡排序没有进行任何优化。对此，我们可以设置exchange变量记录冒泡排序最后一次交换记录的位置，从此之后的所有记录均以有序。

```
void bubble_sort(int array[], int n) {
    int j, exchange, bound, temp;
    exchange = n - 1; // 第一趟冒泡排序区间是[0~n-1]
    while (exchange != 0) {
        bound = exchange; // 确定边界
        exchange = 0;
        for (j = 0; j < bound; j++) {
            if (array[j > array[j + 1]]) {
                temp = array[j];
                array[j] = array[j + 1];
```

```
                array[j + 1] = temp;
                exchange = j;
            }
        }
    }
}
```

快速排序

改进点：记录的比较和移动从两端向中间进行，值较大的记录一次就能从前面移动到后面，值较小类似。记录移动的距离较远，从而减少比较和移动次数。

基本思想：首先确定一个**轴值**，将待排序记录划分成两部分，左侧记录均小于或等于轴值，右侧记录均大于或等于轴值（最终情况）。然后分别对这两部分重复上述过程，直到整个序列有序。

一般而言，轴值选取第一个记录。

```
// 划分算法
int Partition(int arr[], int first, int last) {
    int i = first, j = last, temp;
    while (i < j) {
        while (i < j && arr[i] <= arr[j])
            j--; // 右侧扫描
        // 跳出循环说明产生异常情况 => 交换元素
        if (i < j) {
            Swap(arr[i], arr[j]);
            i++;
        }

        while (i < j && arr[i] <= arr[j])
            i++; // 左侧扫描
        // 跳出循环说明产生异常情况 => 交换元素
        if (i < j) {
            Swap(arr[i], arr[j]);
            j--;
        }
    }
    return i; // i为轴值记录的最终位置
}

void QuickSort(int arr[], int first, int last) {
    if (first >= last)
        return ; // 区间长度为，递归结束
    else {
        int pivot = Partition(arr, first, last);
        QuickSort(arr, first, pivot - 1); // 对左侧子序列进行快速排序
        QuickSort(arr, pivot + 1, last); // 对右侧子序列进行快速排序
    }
}
```

时间复杂度为**O(nlogn)**

选择排序

简单选择排序

基本思想：第i趟排序在待排序序列ri~rn中选取最小的记录，并和第i个记录交换作为有序序列的第i个记录。

```
void SelectSort(int arr[], int n) {
    int i, j, index;
    for (i = 0; i < n - 1; i++) {
        index = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[index])
                index = j; // 寻找到最小值的下标

        if (index != i) {
            Swap(arr[i], arr[index]);
        }
    }
}
```

简单选择排序是一种不稳定的排序方法。时间复杂度O(n^2)

———— Jerry