

数据结构

宏定义

1. 线性表

顺序表(Sequence List)

初始化

获取元素

查找元素

插入

删除

销毁, 清空, 检查为空

顺序表的头文件

单链表(Single Linked List)

初始化

创建链表(头插法) $O(n)$

创建链表(尾插法) $O(n)$

获取元素 $O(n)$

查找元素 $O(n)$

插入元素 $O(n)$

删除元素 $O(n)$

总结插入和删除操作算法的不同

销毁链表

单链表头文件

循环链表(Circular Linked List)

双向链表(Double Linked List)

初始化

创建双向链表

插入和删除

线性表玩具

线性表合并

有序表合并(并归排序的基础)

有序链表合并

多项式创建 and 多项式相加

2. 栈和队列(Stack and Queue)

顺序栈(Sequence Stack)

栈的类型定义

栈的初始化

入栈(Push)

出栈(Pop)

栈的其他操作

测试代码

链栈(Linked Stack)

链栈的类型定义

链栈初始化

链栈Push

链栈Pop

链栈的其他操作

测试代码

Stack.h

栈与递归

函数的调用过程

循环队列

顺换队列初始化

顺换队列入队

循环队列出队

循环队列其他操作

测试代码

链队

链队初始化

链队入队

链队出队

栈和队列玩具

进制转化

括号的匹配

10以内的计算器

3. 字符串, 数组, 广义表

字符串匹配

BF算法(Brute Force)

KMP算法

求next数组

字符串匹配玩具

数组

广义表(Generalized List)

4. 树、二叉树、森林

定义

树的基本术语

二叉树的定义

二叉树的性质

二叉树定理

完全二叉树性质及定理

链式二叉树

存储结构

遍历方式(递归)

遍历方式(非递归)

先序遍历创建二叉树

测试(创建, 遍历)

层次遍历算法

复制二叉树

求深度和节点数

销毁二叉树(递归和非递归)

头文件

线索二叉树 Threaded Binary Tree(了解即可)

数据类型定义

树和森林

树的存储结构

1. 双亲表示法

2. 孩子链表

3. 孩子兄弟表示法(树转化二叉树的基础)

树与二叉树之间相互转换

树转二叉树

二叉树转树

森林与二叉树之间的转化

森林转二叉树

二叉树转森林

树和森林的遍历

树的遍历

森林的遍历

哈夫曼树

术语

最优二叉树

构造哈夫曼树

step和定理

代码实现

哈夫曼编码

代码实现

整体头文件

5. 图(Graph)

图的定义和术语

完全图

稀疏图(Sparse Graph)和稠密图(Dense Graph)

顶点的度(degree)

路径(path)

连通图(Connected Graph)

连通子图和连通分量(Connected Component)

极小连通子图和生成树(Spanning Tree)

图的存储结构

邻接矩阵(Adjacency Matrix)

无向图

有向图

带权图(网) weighted Graph

代码实现

无向无权图

无向带权图

有向带权图

有向无权图

邻接表(Adjacency List)

代码实现

无向无权图

邻接表和邻接矩阵的比较

十字链表

邻接多重链表

图的遍历

深度优先 (Depth First Search)

遍历矩阵

遍历邻接表

广度优先 (Breadth First Search)

遍历矩阵

遍历邻接表

算法效率

最小生成树 Spanning Tree

Prim 算法

代码实现

邻接矩阵

邻接表

Kruskal 算法

代码实现

邻接矩阵

邻接表

最短路径

Dijkstra 算法

代码实现

邻接矩阵

Floyd算法

代码实现

邻接矩阵

有向无环图(Directed Acycline Graph)

AOC和AOE

拓扑排序

代码实现

邻接表

关键路径

代码实现

邻接表

图的头文件

define_Graph.h

AMGraph.h

ALGraph.h

数据结构

宏定义

在本笔记中用到的宏定义，头文件为define.h

```
#ifndef __DEFINE_H
#define __DEFINE_H

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
typedef int Status;

#endif
```

1. 线性表

• 顺序表(Sequence List)

特点：逻辑上相邻的数据元素，其物理次序也是相邻的

线性表中第 $i + 1$ 个数据元素的存储位置 $LOC(a_{i+1})$ 和第 i 个元素满足下列关系

$$\begin{aligned} LOC(a_{i+1}) &= LOC(a_i) + l \\ LOC(a_i) &= LOC(a_0) + (i - 1) \times l \\ l &: \text{代表每个元素所占的存储单元} \end{aligned}$$

循序表的存储结构：

```
#define SQLMAXSIZE 100
typedef int SqlElemType;
typedef struct __Sqlist {
    SqlElemType *base;
    int length;
} Sqlist;
```

数组的动态定义，base指向数组的第一元素的地址

- 初始化

```

Status InitSL(Sqlist *L, int length) {
    L->base = (SqlElemType *)malloc(sizeof(SqlElemType) * SQLMAXSIZE);
    if (!L->base)
        return OVERFLOW;
    L->length = 0;

    for (int i = 1; i < length + 1; i++) {
        SqlElemType e;
        scanf(" %d", &e);
        SqlInsert(L, i, e);
    }
    return OK;
}

```

- 获取元素

```

Status GetElem(Sqlist *L, int position, SqlElemType *e) {
    if (position < 1 || position > L->length)
        return ERROR;
    *e = L->base[position - 1];
    return OK;
}

```

- 查找元素

```

int LocateElem(Sqlist *L, SqlElemType e) {
    for (int i = 0; i < L->length; i++) {
        if (e == L->base[i])
            return i + 1;
    }
    return 0; // 0代表查找元素不在循序表中
}

```

平均查找长度ASL(Average Search Length)

$$ASL = \sum_{i=1}^n p_i C_i$$

p_i 为查找第*i*个元素成功的概率

C_i 为查找第*i*个元素需要比较的次数

$$\text{可知 } p_i = \frac{1}{n}$$

C_i 为第*i*个元素在表中位置

$$ASL = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}$$

因此可知*LocateElem*的时间复杂度为 $O(n)$

- 插入

```

Status SqlInsert(Sqlist *L, int position, SqlElementType e) {
    if (position < 1 || position > L->length + 1)
        return ERROR;
    if (L->length == SQLMAXSIZE)
        return OVERFLOW;
    for (
        int i = L->length - 1; i >= position - 1;
        i--) { //注意需要把数组中的元素全部向右移动，需要从数组最右边的元素开始移动
        L->base[i + 1] = L->base[i];
    }
    L->base[position - 1] = e;
    L->length++;
    return OK;
}

```

E_{ins} 表示插入元素所需要移动元素次数的期望值(平均次数)

$$E_{ins} = \sum_{i=1}^{n+1} = (p_i)(n - i + 1)$$

假设在各个位置上插入元素的概率相等 $p_i = \frac{1}{n+1}$

$$E_{ins} = \frac{1}{n+1} \sum_{i=1}^{n+1} (n - i + 1) = \frac{n}{2}$$

因此 $O(n)$

- 删除

```

Status SqlDelete(Sqlist *L, int position, SqlElementType *e) {
    if (position < 1 || position > L->length)
        return ERROR;
    for (int i = position; i < L->length; i++) {
        L->base[i - 1] = L->base[i];
    }
    *e = L->base[position - 1];
    L->length--;
    return OK;
}

```

$$E_{del} = \sum_{i=1}^n p_i (n - i) = \frac{1}{n} \sum_{i=1}^n (n - i) = \frac{n-1}{2}$$

$O(n)$

- 销毁, 清空, 检查为空

```

Status SqlDelete(Sqlist *L, int position, SqlElementType *e) {
    if (position < 1 || position > L->length)
        return ERROR;
    for (int i = position; i < L->length; i++) {
        L->base[i - 1] = L->base[i];
    }
}

```

```

    }

    *e = L->base[position - 1];
    L->length--;
    return OK;
}

Status SqlDestroy(Sqlist *L) {
    if (!L->base)
        return ERROR;
    else {
        free(L->base);
        return OK;
    }
}

void SqlClear(Sqlist *L) { L->length = 0; }

Status SqlIsEmpty(Sqlist *L) {
    if (0 == L->length)
        return TRUE;
    else
        return FALSE;
}

```

- 顺序表的头文件

```

#include "define.h"
#include <stdio.h>
#include <stdlib.h>

#ifndef __SEQUENCELIST_H
#define __SEQUENCELIST_H

#define SQLMAXSIZE 100
typedef int SqlElemType;
typedef struct __SqlList {
    SqlElemType *base;
    int length;
} SqlList;

Status InitSL(SqlList *L, int length);
Status GetElem(SqlList *L, int position, SqlElemType *e);
int LocateElem(SqlList *L, SqlElemType e);
Status SqlInsert(SqlList *L, int position, SqlElemType e);
Status SqlDelete(SqlList *L, int position, SqlElemType *e);
Status SqlDestroy(SqlList *L);
void SqlClear(SqlList *L);
Status SqlIsEmpty(SqlList *L);
void MergeList(SqlList *La, SqlList *Lb);
void Traverse(SqlList *L);
void MergeList_Seq(SqlList *La, SqlList *Lb, SqlList *Lc);

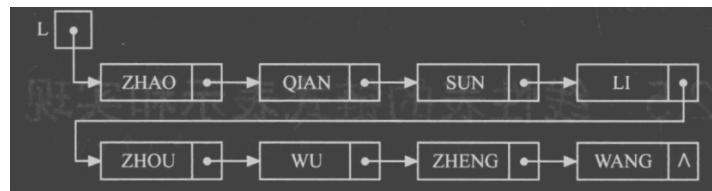
```

#endif

• 单链表(Single Linked List)

- 单链表由头节点(不存放数据只存放下个节点的地址)和n个节点组成,
- 每个节点分为两个域: 数据域和指针域(存放下个节点的地址)
- 第n个节点的指针域为NULL

如下图所示



```
typedef int LlElemtype;
typedef struct _LNode {
    LlElemtype data; //存放单个节点的数据
    _LNode *next;   //存放下个节点的地址
} LNode, *LinkList;
```

- 初始化

```
Status InitLL(LinkList *L) // L是个二级指针
{
    (*L) = (LinkList)malloc(sizeof(LNode));
    (*L)->next = NULL;
    return OK;
}
```

- 创建链表(头插法) $O(n)$

```
void CreatLL_H(LinkList L, int n)
    //用此方法创建的链表, 遍历的顺序和创建的顺序相反
{
    printf("Please input %d numbers:", n);
    for (int i = 0; i < n; i++) {
        LinkList p = (LinkList)malloc(sizeof(LNode));
        int data;
        scanf(" %d", &data); // %d前面的空格代表清除制表符回车等符号
        p->data = data;
        p->next = L->next;
        L->next = p;
    }
}
```

- 创建链表(尾插法) $O(n)$

```
void CreatLL_R(LinkList L, int n) {
    printf("Please input %d numbers:", n);
    LinkList ptail;
    ptail = L;
    for (int i = 0; i < n; i++) {
        LinkList pnew;
        pnew = (LinkList)malloc(sizeof(LNode));
        int data;
        scanf(" %d", &data);
        pnew->data = data;
        pnew->next = NULL;
        ptail->next = pnew;
        ptail = pnew;
    }
}
```

- 获取元素 $O(n)$

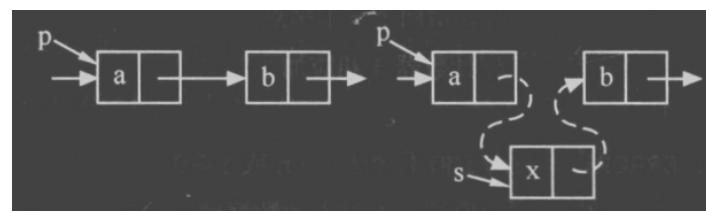
```
Status GetElem(LinkList L, int position, LlElmtype *e) {
    LinkList p = L->next;
    int i = 1; //使i和p的位置同步, 即i代表着p在链表中的位置
    if (position < 1 || !p)
        return ERROR;
    while (p && i < position) { //此处不可 i<=position, 因为while成立时, 内部会i++
        p = p->next;
        i++;
    }
    *e = p->data;

    return OK;
}
```

- 查找元素 $O(n)$

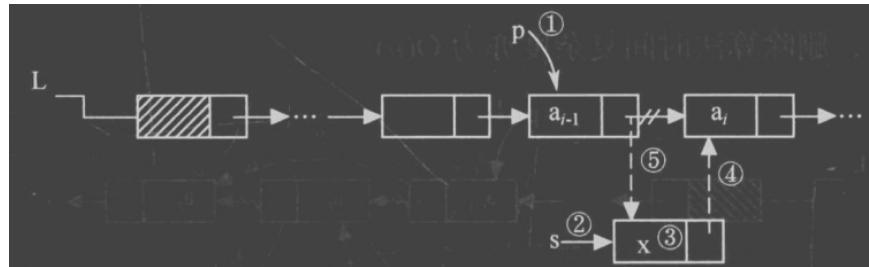
```
LinkList LocateElem(LinkList L, LlElmtype e) {
    LinkList p = L->next;
    while (p && p->data != e) {
        p = p->next;
    }
    if (!p)
        return NULL; //如果p的地址为空 说明e不在链表中
    return p;
}
```

- 插入元素 $O(n)$



想在 a, b 之间插入 c , 需要先知道 a 节点的地址

如图所示, 如果想要在位置 i 插入节点, 则需要知道位置 $i - 1$ 节点的位置

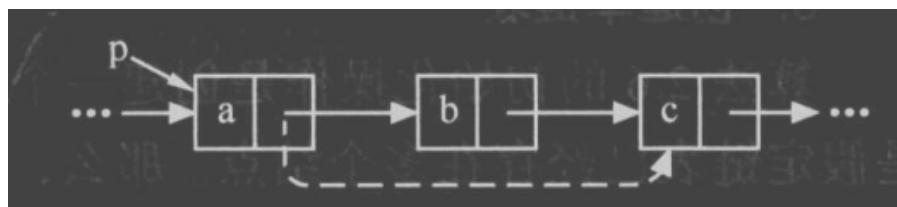


- 注意因为插入操作和GetElem操作不同
- 要从0开始, p 要从 L 开始
- 如果从1和 L 开始的话, 无法再位置1插入元素

```
Status LInsert(LinkList L, int position, LlElemtpe e) {
    LinkList p = L; // 注意因为插入操作和GetElem操作不同
    int i = 0; // i要从0开始, p要从L开始
    // 如果从1和L开始的话, 无法再位置1插入元素
    while (p && i < position - 1) { // 查找插入节点位置的前一个节点
        i++;
        p = p->next;
    }
    if (!p || i > position - 1)
        return ERROR;

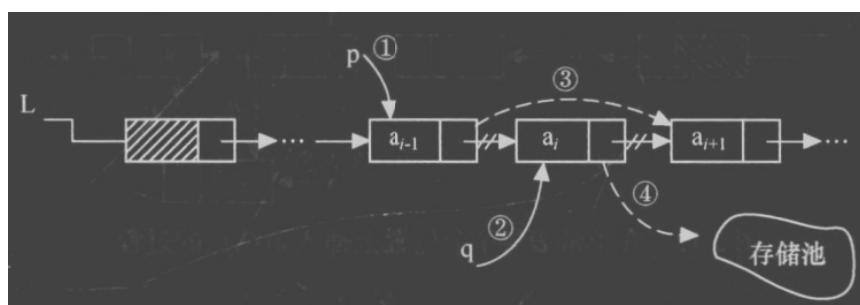
    LinkList pnew = (LinkList)malloc(sizeof(LNode));
    pnew->data = e;
    pnew->next = p->next;
    p->next = pnew;
    return OK;
}
```

- 删除元素 $O(n)$



想要删除 b , 则必须先知道 a 的地址

- 注意因为插入操作和GetElem操作不同
- 要从0开始, p 要从 L 开始
- 如果从1和 L 开始的话, 无法再位置1插入元素



```

Status LlDelete(LinkList L, int position, LlElmtype *e) {
    LinkList p = L;
    int i = 0;

    while (p && i < position - 1) {
        //查找position-1位置节点的地址
        i++;
        p = p->next;
    }
    if (i > position - 1 || !p || !p->next)
        return ERROR;
    //注意是 多增加了判断条件!(p->next) 当节点数为n, 删除的位置为n+1时会返回error

    LinkList pfree = p->next;
    *e = pfree->data;
    p->next = pfree->next; //此处可改写成 p->next = p->next->next;
    free(pfree);
    return OK;
}

```

- 总结插入和删除操作算法的不同

```

while (p){
    p = p->next;
}//最终p的值为NULL

while(p->next){
    p = p->next;
}//最终p的值为最后一个节点的地址
//-----插入-----
//如果插入操作的position不合法, 即position > n+1(n为链表长度), 那么p一定会指向NULL, 此时按照
退出条件!p可以返回ERROR
if (!p || i>position-1) return ERROR;
//但是如果采用:
while(p->next)
//则最终会指向链表最后一个节点, 即使position不合法, 那么也会在最后一个节点后方插入新节点
//所以使用:
while(p)

//-----删除-----
//如果删除操作的positon不合法, 即position>链表长度, p会指向, 最后一个节点的地址(position ==
n+1时)或是NULL(position > n+1), 那么下面的代码会出错
LinkList pfree = p->next;
//如果p指向最后一个节点, 此时pfree指向NULL。如果p指向NULL, 此时pfree指向非法空间(不受主程序控
制), 从而导致下面代码报错
*e = pfree->data;
//所以需要增加一个判断条件

```

```

if (i>position-1 || !p || !p->next) return ERROR;
//必须保证 !p 要在 !p->next的左边, 即position > n+1 的情况
//这是因为如果 !p->next 在 !p 的左边, 如果p指向NULL, 那么NULL->next会报错

```

- 销毁链表

```

Status LlDestroy(LinkList *L) {
    if (!(*L))
        return ERROR;
    LinkList p = *L;
    while (p) {
        LinkList pfree = p; // pfree保存要释放的节点地址
        p = p->next;      //此行和下行的顺序不能反
        free(pfree);
        pfree = NULL;
    }
    *L = NULL;
    return OK;
}

```

- 单链表头文件

```

#include "define.h"
#include <stdio.h>
#include <stdlib.h>

#ifndef __LINKLIST_H
#define __LINKLIST_H

typedef int LlElemtype;
typedef struct __LNode {
    LlElemtype data; //存放单个节点的数据
    __LNode *next;   //存放下个节点的地址
} LNode, *LinkList;

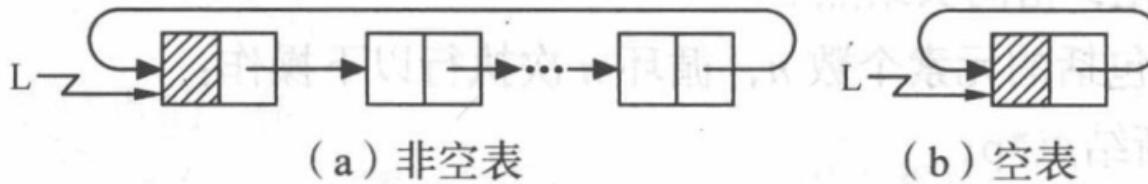
Status InitLL(LinkList *L);
void CreatLL_H(LinkList L, int n);
void CreatLL_R(LinkList L, int n);
Status GetElem(LinkList L, int position, LlElemtype *e);
LinkList LocateElem(LinkList L, LlElemtype e);
Status LlInsert(LinkList L, int position, LlElemtype e);
Status LlDelete(LinkList L, int position, LlElemtype *e);
void Traverse(LinkList L);
Status LlDestroy(LinkList *L);
void Merge_LinkedList(LinkList *La, LinkList *Lb, LinkList *Lc);
#endif

```

• 循环链表(Circular Linked List)

循环链表的特点：

- 最后一个节点的指针域指向头节点，整个表链形成一个环
- 由此，从表中任意节点出发，可以找到其他节点



和单链表很像，区别就是最后一个节点的next域指向头节点

• 双向链表(Double Linked List)

有两个指针域，一个指向直接前驱，另一个指向直接后继

数据类型

```
typedef int DouLElemtype;
typedef struct __DouLinkNode {
    DouLElemtype data;
    __DouLinkNode *prior;
    __DouLinkNode *next;
} DouLinkNode, *DouLinkedList;
```

- 初始化

```
void InitDL(DouLinkedList *L) {
    *L = (DouLinkedList)malloc(sizeof(DouLinkNode));
    (*L)->next = NULL;
    (*L)->prior = NULL;
}
```

- 创建双向链表

```
void CreatDL_H(DouLinkedList L, int length) {
    for (int i = 0; i < length; i++) {
        DouLinkedList pnew = (DouLinkedList)malloc(sizeof(DouLinkNode));
        int data;
        printf("(for %d)Please input the data:", i + 1);
        scanf("%d", &data);
        pnew->data = data;
        pnew->next = L->next;
        pnew->prior = L;
        L->next = pnew;
    }
}
```

```

    }

}

void CreatDL_R(DouLinkList L, int length) {
    DouLinkList ptail = L;
    for (int i = 0; i < length; i++) {
        DouLinkList pnew = (DouLinkList)malloc(sizeof(DouLinkNode));
        int data;
        printf("(for %d)Please input the data:", i + 1);
        scanf("%d", &data);
        pnew->data = data;
        pnew->next = NULL;
        pnew->prior = ptail;
        ptail->next = pnew;
        ptail = pnew;
    }
}

```

- 插入和删除

```

void DlInsert(DouLinkList L, int position, DouLElemtype e) {
    int i = 0;
    DouLinkList p = L;
    while (p->next && i < position - 1) {
        i++;
        p = p->next;
    }
    DouLinkList pnew = (DouLinkList)malloc(sizeof(DouLinkNode));
    pnew->data = e;
    pnew->next = p->next;
    pnew->prior = p;
    p->next = pnew;
    p->next->prior = pnew;
}

void DlDelete(DouLinkList L, int position, DouLElemtype *e) {
    DouLinkList p = L;
    int i = 0;
    while (p->next && i < position - 1) {
        p = p->next;
        i++;
    }
    DouLinkList pfree = p->next;
    *e = pfree->data;
    p->next = p->next->next;
    p->next->next->prior = p;
    free(pfree);
    pfree = NULL;
}

```

• 线性表玩具

- 线性表合并

已知两个集合

$$A = (7, 5, 3, 11)$$

$$B = (2, 6, 3)$$

求出合并后集合 $A = (7, 5, 3, 11, 2, 6)$

```
void MergeList(Sqlist *La, Sqlist *Lb) {
    for (int i = 1; i < Lb->length + 1; i++) {
        SqlElemType e;
        GetElem(Lb, i, &e);
        if (!LocateElem(La, e)) {
            La->base[La->length++] = e;
        }
    }
}

void Traverse(Sqlist *L) {
    for (int i = 0; i < L->length; i++) {
        printf("%d ", L->base[i]);
    }
}
```

```
#include <SequenceList.h>

int main(void) {
    Sqlist La, Lb;
    Sqlist *pa = &La;
    Sqlist *pb = &Lb;
    InitSL(pa, 4);
    InitSL(pb, 3);

    MergeList(pa, pb);

    Traverse(pa);

    system("pause");
    return 0;
}
/*
7 5 3 11
2 6 3
*/
```

- 有序表合并(并归排序的基础)

$$A = (3, 5, 8, 11)$$

$$B = (2, 6, 8, 11, 15, 20)$$

合并成新集合

$$C = (2, 3, 5, 6, 8, 8, 9, 11, 11, 15, 20)$$

```
void MergeList_Seq(Sqlist *La, Sqlist *Lb, Sqlist *Lc) {
    Lc->length = La->length + Lb->length;
    SqlElemType *pa = La->base, *pa_last = pa + La->length - 1;
    // pa指向La->base的首地址, pa_last指向base中最后一个元素的地址, 下面同理
    SqlElemType *pb = Lb->base, *pb_last = pb + Lb->length - 1;
    SqlElemType *pc = Lc->base;

    while (pa <= pa_last && pb <= pb_last) {
        //当pa>pa_last时说明, 有集合中的元素已经全部加入到Lc中
        if (*pa < *pb)
            *(pc++) = *(pa++);
        else
            *(pc++) = *(pb++);
    }

    while (pa <= pa_last)
        *(pc++) = *(pa++); //判断La的元素是否全部加入Lc中, 下面同理
    while (pb <= pb_last)
        *(pc++) = *(pb++);
}
```

```
#include <SequenceList.h>
```

```
int main(void) {
    Sqlist La, Lb, Lc;
    InitSL(&La, 4);
    InitSL(&Lb, 7);
    InitSL(&Lc, 0);

    MergeList_Seq(&La, &Lb, &Lc);
    Traverse(&Lc);

    system("pause");
    return 0;
}
/*
3 5 8 11
2 6 8 9 11 15 20
*/
```

- 有序链表合并

```
void Merge_LinkedList(LinkList *La, LinkList *Lb, LinkList *Lc) {
    *Lc = *La; //让Lc使用La的头节点进行合并
    LinkList pa = (*La)->next, pb = (*Lb)->next; // pa, pb分别表示合并时所指节点
```

```

LinkList pc = *Lc; // pc表示Lc的尾节点

while (pa && pb) {
    if (pa->data < pb->data) {
        pc->next = pa;
        pc = pa;
        pa = pa->next;
    } else {
        pc->next = pb;
        pc = pb;
        pb = pb->next;
    }
}
pc->next = pa ? pa : pb;
// pc->next不需要NULL,因为合并时一定会剩下一串节点, 只需指向该剩下的节点就OK
free(*Lb);
*La = *Lb = NULL;
}

```

```

#include "LinkList.h"

int main(void) {
    LinkList La, Lb, Lc;
    InitLL(&La), InitLL(&Lb), InitLL(&Lc);
    CreatLL_R(La, 4);
    CreatLL_R(Lb, 7);
    Merge_LinkedList(&La, &Lb, &Lc);

    Traverse(Lc);

    system("pause");
    return 0;
}
/*
3 5 8 11
2 6 8 9 11 15 20
*/

```

- 多项式创建 and 多项式相加

创建一个多项式，并按照指数的高低排序

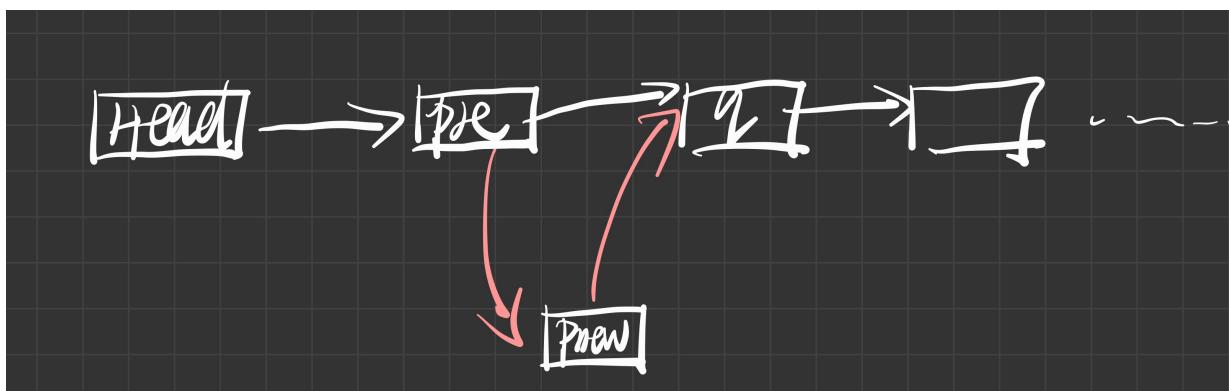
多项式结构体

```

typedef struct __PolyNode {
    double coefficient;
    int exponent;
    __PolyNode *next;
} PolyNode, *Polynomial;

```

多项式创建



核心变量为 `Polynomial q; Polynimial pre;`

核心语句为 `while(q && q->exponent < pnew->exponent)`

```

void InitPolynomial(Polynomial *p, int length) {
    *p = (Polynomial)malloc(sizeof(PolyNode)); //先初始化头节点
    (*p)->next = NULL;

    printf("Please input the coefficient and exponent:");
    for (int i = 0; i < length; i++) {
        Polynomial pnew = (Polynomial)malloc(sizeof(PolyNode));
        scanf(" %lf", &(pnew->coeffcient));
        scanf(" %d", &(pnew->exponent));
        Polynomial q = (*p)->next; // q为指向比pnew->exponent大的节点
        Polynomial pre = (*p);      // pre指向q的直接前驱节点

        while (q && q->exponent < pnew->exponent) { //第1次的for循环不会执行
            //直到找到一个节点的exponent大于pnew->exponent,如果没找到q指向NULL
            pre = q;
            q = q->next;
        }
        pnew->next = q; //因为q->exponent > pnew->exponent
        pre->next = pnew;
    }
}

```

主程序

```

#include <stdio.h>
#include <stdlib.h>

typedef struct __PolyNode {
    double coeffcient;
    int exponent;
    __PolyNode *next;
} PolyNode, *Polynomial;

void InitPolynomial(Polynomial *p, int length);
void Traverse(Polynomial P);
Polynomial AddPolynomial(Polynomial pa, Polynomial pb);
int main(void) {
    Polynomial p1, p2, p3;
    InitPolynomial(&p1, 3);
    InitPolynomial(&p2, 4);
    p3 = AddPolynomial(p1, p2);
    Traverse(p1);
}

```

```

        system("pause");
        return 0;
    }
/*
x^6+2x^2+3x^5
-2x^2+3x^5+2x^6+x^3
-----
input:
1 6 2 2 3 5
-2 2 3 5 2 6 1 3
-----
output:
(1.0x^3)+(6.0x^5)+(3.0x^6)
*/

```

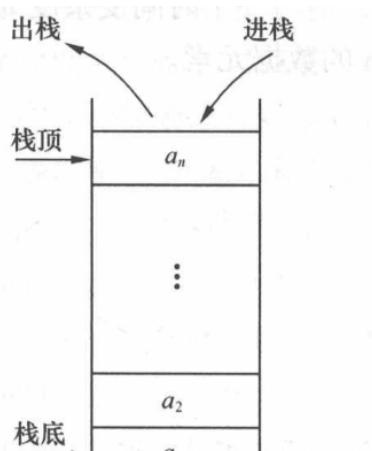
2. 栈和队列(Stack and Queue)

- 顺序栈(Sequence Stack)

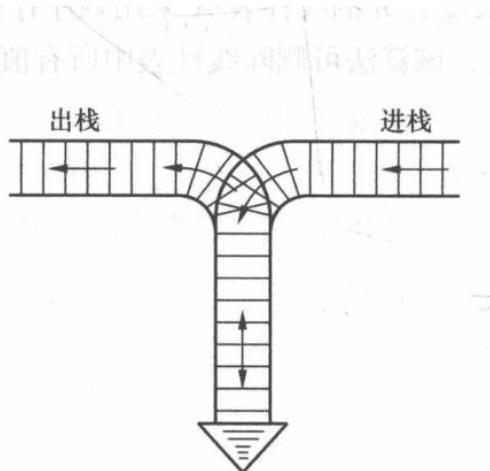
栈是限定仅在表尾进行插入或删除操作的线性表，表末端为栈顶(Top)，表头称为栈底(Bottom)，不含元素称为空栈

(用顺序表存储的栈更常见)

因此栈又称为后进先出(Last in First out, LIFO)的线性表，如下图



(a) 栈的示意图



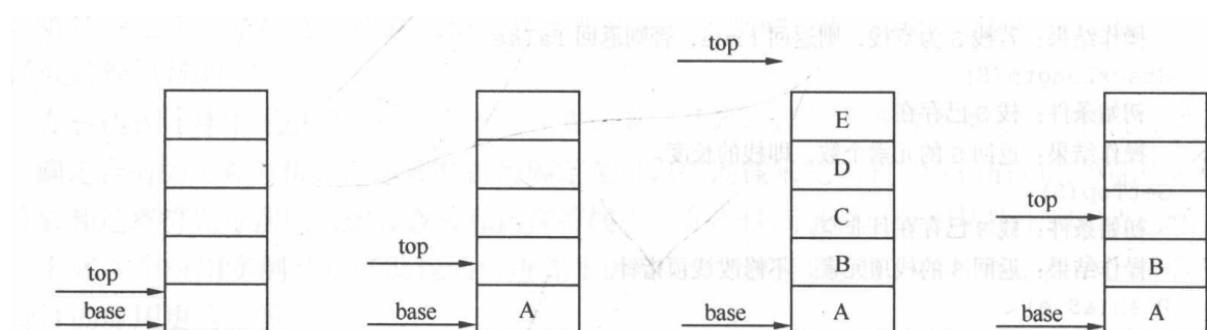
(b) 用铁路调度站表示栈

- 栈的类型定义

```
#define MAXSTACK 100
typedef char StackElemType; //栈数据类型
typedef struct _SqStack //顺序栈, 最常用
{
    StackElemType *base;
    StackElemType *top;
    int stacksize;
} SqStack;
```

- 栈的初始化

```
Status InitStack(SqStack *S) {
    S->base = (StackElemType *)malloc(sizeof(StackElemType) * MAXSTACK);
    if (!S->base)
        exit(OVERFLOW);
    S->top = S->base;           //栈顶和指向栈底
    S->stacksize = MAXSTACK;    //栈的容量
    return OK;
}
```



观察上图，发现top指向内存空间不存放元素

- 入栈(Push)

因为top指向的内存不存放空间，当为base分配的空间存满元素时，top = 分配空间的最后一个元素的地址+1，此时表示栈满，即 `top-base = stacksize`

```
Status Push(SqStack *S, StackElemType e) {
    if (S->top - S->base == S->stacksize)
        return ERROR; //表示此时栈满
    *(S->top++) = e; // top指向内存单元存放e, 且top指向下一个内存单元
    return OK;
}
```

- 出栈(Pop)

当 `base == top` 时, 表示栈空

```
Status Pop(SqStack *S, StackElemType *e) {
    if (S->base == S->top)
        return ERROR; // 栈空
    *e = *(--(S->top));
    return OK;
}
```

- 栈的其他操作

```
Status IsEmpty(SqStack *S) {
    if (S->base == S->top)
        return TRUE;
    else
        return FALSE;
}

Status IsFull(SqStack *S) {
    if (S->top - S->base == S->stacksize)
        return TRUE;
    else
        return ERROR;
}

StackElemType GetTop(SqStack *S) {
    if (!IsEmpty(S))
        return *(S->top - 1);
}
```

- 测试代码

```
#include <Stack.h>

int main(void)
{
    SqStack S;
    InitStack(&S);
    Push(&S, 'A');
    Push(&S, 'B');
    Push(&S, 'C');
    Push(&S, 'C');
    while (!IsEmpty(&S))
    {
        StackElemType e;
        Pop(&S, &e);
        printf("%c ", e);
    }

    system("pause");
}
```

```

    return 0;
}

```

• 链栈(Linked Stack)

由于栈的主要操作是对栈顶进行Push和Pop，所以选用top作为链表的头节点

- 链栈的类型定义

```

typedef struct __StackNode //链栈
{
    StackElemType data;
    __StackNode *next;
} StackNode, *LinkStack;

```

- 链栈初始化

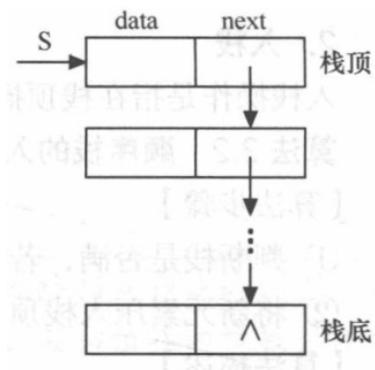
```

Status InitStack(LinkStack *S) {
    *S = NULL;
    //(*S)->next = NULL; 不需要此行代码
    return OK;
}

```

- 链栈Push

在第一次Push时，第一个节点的next指向NULL

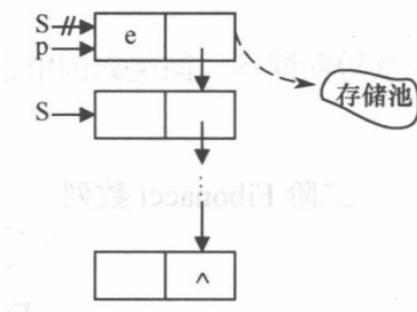


```

Status Push(LinkStack *S, StackElemType e) {
    LinkStack pnew = (LinkStack)malloc(sizeof(StackNode));
    pnew->data = e;
    pnew->next =
        *S; //在第一次Push时, 第一个节点的next指向NULL, 因为初始化了*S=NULL;
    *S = pnew;
    return OK;
}

```

- 链栈Pop



```

Status Pop(LinkStack *S, StackElemType *e) {
    LinkStack pfree = *S; //临时保存栈顶节点S
    *e = (*S)->data;
    *S = (*S)->next;
    free(pfree); //出栈后释放
    return OK;
}

```

- 链栈的其他操作

```

StackElemType GetTop(LinkStack *S) {
    if (*S)
        return (*S)->data;
}

Status IsEmpty(LinkStack *S) {
    if (!(*S))
        return TRUE;
    else
        return FALSE;
}

```

- 测试代码

```
#include "Stack.h"

int main(void)
{
    LinkStack S;
    InitStack(&S);
    Push(&S, 'E');
    Push(&S, 'A');
    Push(&S, 'C');
    Push(&S, 'H');
    Push(&S, 'R');
    while (!IsEmpty(&S))
    {
        StackElemType e;
        Pop(&S, &e);
        printf("%c ", e);
    }
    system("pause");
    return 0;
}
```

- Stack.h

```
#include "define.h"
#ifndef _STACK_H
#define _STACK_H
//-----
#define MAXSTACK 100
typedef char StackElemType; //栈数据类型
typedef struct __SqStack //顺序栈, 最常用
{
    StackElemType *base;
    StackElemType *top;
    int stacksize;
} SqStack;

Status InitStack(SqStack *S);
Status Push(SqStack *S, StackElemType e);
Status Pop(SqStack *S, StackElemType *e);
Status IsEmpty(SqStack *S);
Status IsFull(SqStack *S);
StackElemType GetTop(SqStack *S);
//-----
typedef struct __StackNode //链栈
{
    StackElemType data;
    __StackNode *next;
} StackNode, *LinkStack;

Status InitStack(LinkStack *S);
Status Push(LinkStack *S, StackElemType e);
Status Pop(LinkStack *S, StackElemType *e);
StackElemType GetTop(LinkStack *S);
```

```
Status IsEmpty(LinkStack *S);
#endif
```

• 栈与递归

- 函数的调用过程

调用前系统完成：

1. 将实参，返回地址(下行代码地址)等传递给被调用函数
2. 为被调用函数的局部变量分配空间
3. 将控制转移到被调用函数的入口

调用后，系统完成：

1. 保存被调用函数的计算结果(返回值)
2. 释放被调用函数的数据区
3. 依照被调用函数保存的返回地址，将控制转移到调用函数

递归调用函数时，如下图

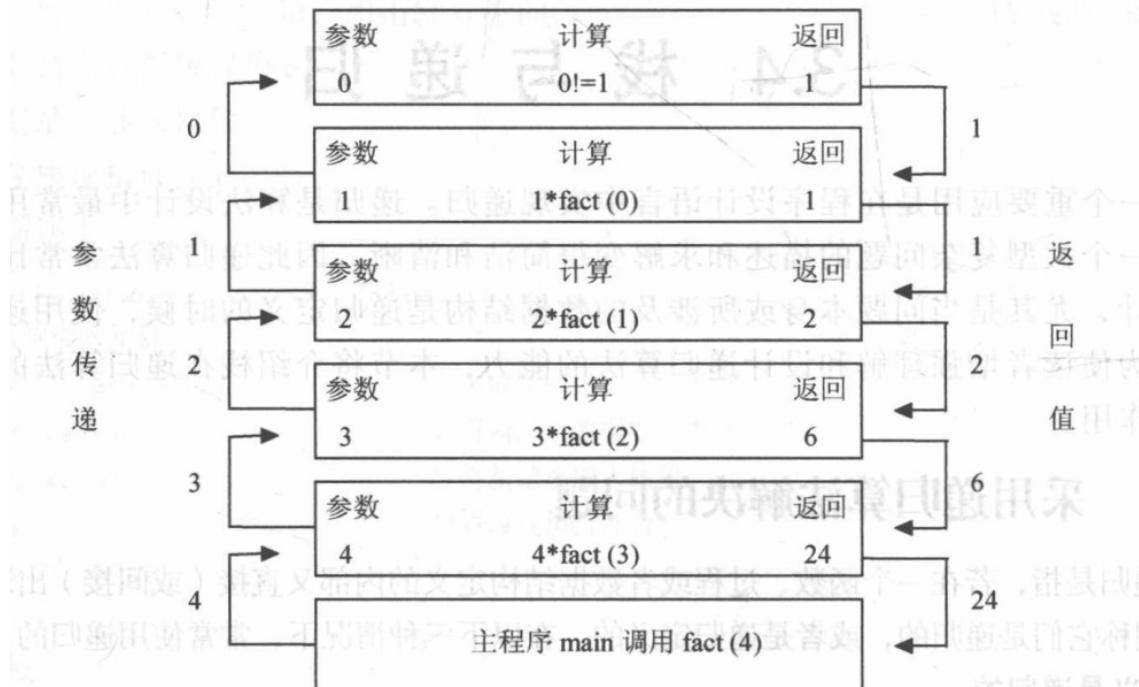


图 3.7 求解 $4!$ 的过程

- 按照调用顺序依次把各个函数入栈
- 当栈顶函数满足return条件时，依次出栈(按照FISLO原则)，并且返回值从上向下传递
- 直到主程序调用的`fact(4)`出栈后，递归完成

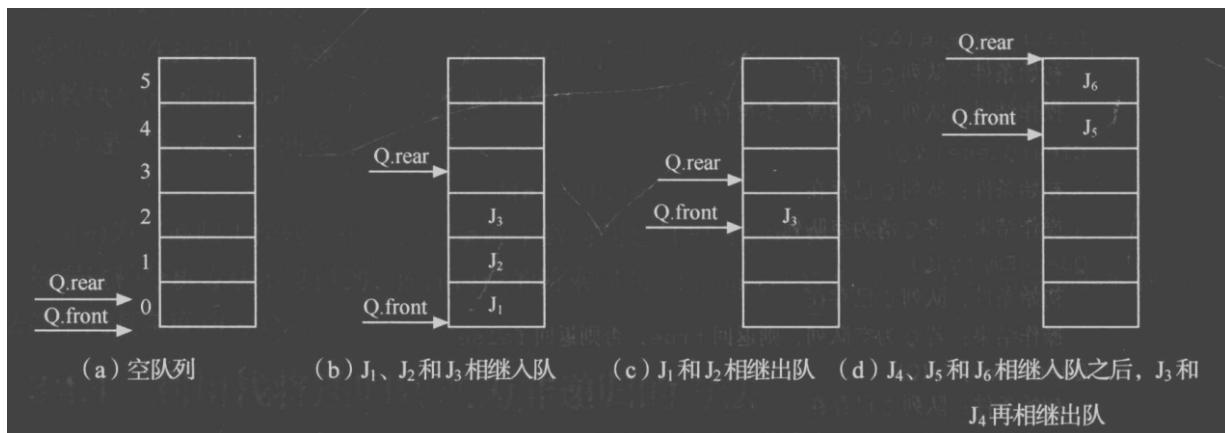
• 循环队列

定义：只能在表的一端进行插入运算，在表的另一端进行删除运算的线性表

先进先出(First in First out)原则

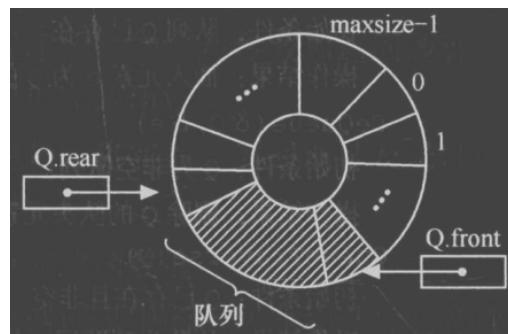
数据类型定义

```
#define QMAXSIZE 100
typedef char QElemType;
typedef struct _SqQueue {
    QElemType *base;
    int front, rere; //front为队头下标, rere为队尾下标(rere下标的位置不存放元素)
} SqQueue;
//入队rere+1,出队front+1,但是此种情况存在问题,如下图
```



如上图, (d)虽然数组中的空间没有满,但是rear却不能继续增加,假溢出

解决方法：把base数组想象成一个环形的循环队列如下图



此时如果 `front == rere` 表示队空, 而 `front == (rere+1) % QMAXSIZE` 时表示队满

- 顺序队列初始化

把 front 和 rere 初始化为0

```
Status InitQueue(SqQueue *Q) {
    if (!(Q->base = (QElemType *)malloc(sizeof(QElemType) * QMAXSIZE)))
        exit(OVERFLOW);
    Q->front = Q->rere = 0;
    return OK;
}
```

- 顺换队列入队

```
Status EntryQ(SqQueue *Q, QElemType e) {
    if ((Q->rere + 1) % QMAXSIZE == Q->front) //判断是否队满
        return ERROR;
    Q->base[Q->rere] = e;
    Q->rere = (Q->rere + 1) % QMAXSIZE; // Q->rere++ 错误写法
    return OK;
}
```

- 循环队列出队

```
Status OutQ(SqQueue *Q, QElemType *e) {
    if (Q->rere == Q->front) //判断是否队满
        return ERROR;
    *e = Q->base[Q->front];
    Q->front = (Q->front + 1) % QMAXSIZE; // Q->front++; 为错误写法
    return OK;
}
```

- 循环队列其他操作

```
Status IsEmpty(SqQueue *Q) {
    if (Q->front == Q->rere)
        return TRUE;
    else
        return FALSE;
}

Status IsFull(SqQueue *Q) {
    if ((Q->rere + 1) % QMAXSIZE == Q->front)
        return TRUE;
    else
        return FALSE;
}

Status GetFront(SqQueue *Q, QElemType *e) {
    if (IsEmpty(Q))
        return ERROR;
    *e = Q->base[Q->front];
    return OK;
}

int LengthQueue(SqQueue *Q) {
    return (Q->rere - Q->front + QMAXSIZE) % QMAXSIZE;
}
```

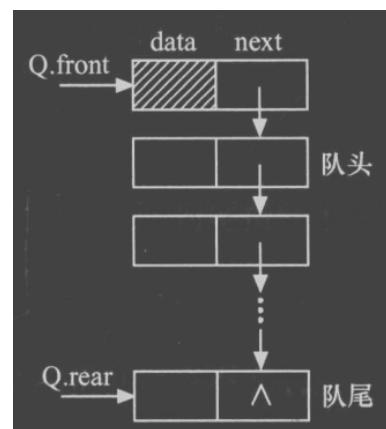
- 测试代码

```
#include "Queue.h"

int main(void) {
    SqQueue Q;
    InitQueue(&Q);
    EntryQ(&Q, 'A');
    EntryQ(&Q, 'B');
    EntryQ(&Q, 'N');
    EntryQ(&Q, 'M');
    printf("%d\n", LengthQueue(&Q));
    QELEMType e;
    while (!IsEmpty(&Q)) {
        OutQ(&Q, &e);
        printf("%c ", e);
    }
    system("pause");
    return 0;
}
```

• 链队

链队结构类似于链表，不同于链表的头指针，用两个指针域 front rere 来表示队列，如下图



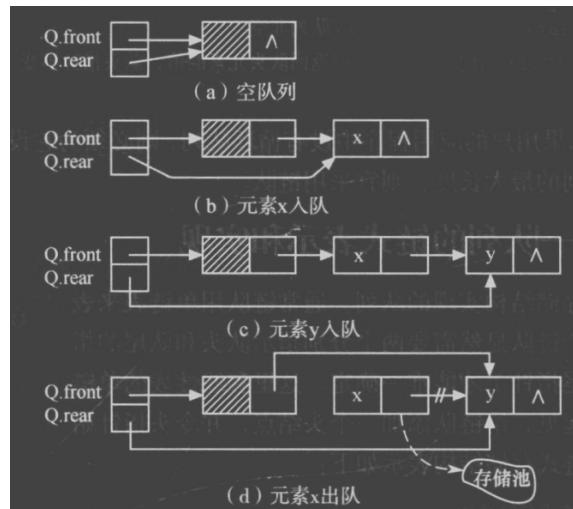
数据类型定义：

```
typedef struct __QueueNode {
    QELEMType data;
    __QueueNode *next;
} QueueNode;
typedef struct __LinkedQueue {
    QueueNode *front; //front相当于链表的头指针
    QueueNode *rere; //rere指向整个链表的最后一个节点
} LinkedQueue;
```

- 链队初始化

```
Status InitQueue(LinkedQueue *Q) {
    // front 和 rear 指向同一节点
    if (!(Q->front = Q->rear = (QueueNode *)malloc(sizeof(QueueNode))))
        exit(OVERFLOW);
    Q->front->next = NULL; // 使该节点next域为NULL
    return OK;
}
```

- 链队入队



```
Status EntryQ(LinkedQueue *Q, QElemType e) {
    QueueNode *pnew = (QueueNode *)malloc(sizeof(QueueNode));
    if (!pnew)
        exit(OVERFLOW);
    pnew->data = e;
    pnew->next = NULL;
    Q->rear->next = pnew;
    Q->rear = pnew;
    return OK;
}
```

- 链队出队

```

Status OutQ(LinkedQueue *Q, QElemType *e) {
    if (Q->front == Q->rere)
        return ERROR;
    QueueNode *pfree = Q->front->next;
    *e = pfree->data;
    Q->front->next = pfree->next;
    //如果删除的节点为队尾, 那么释放pfree之后, rere指向未知存储空间
    if (Q->rere == pfree)
        Q->rere = Q->front;
    free(pfree);
    return OK;
}

```

• 栈和队列玩具

- 进制转化

一个进制转换函数, 有一个参数n(十进制), 要求输出n的8进制

```

//需要提前修改 StackElemType的类型
void Convert_8(int n) {
    SqStack S;
    InitStack(&S);

    int temp = n;
    while (temp) {
        Push(&S, temp % 8);
        temp = temp / 8;
    }

    while (!IsEmpty(&S)) {
        int i;
        Pop(&S, &i);
        printf("%d", i);
    }
    printf("\n");
}

```

- 括号的匹配

输入括号(), 判断括号是否匹配成功, #字符表示输入结束

例: ([()]) 成功 [((())]] 失败

写法1:

```

Status Matching_Parentheses(void) {
    SqStack S;

```

```

InitStack(&S);
char ch, pop;
int flag = 1;
scanf(" %c", &ch);

while (flag && ch != '#') {
    switch (ch) {
        // 注意:不可以写成 case '(' || '[': 如果这样写 ||运算符只会返回1
        case '(':
            Push(&S, ch);
            break;

        case '[':
            Push(&S, ch);
            break;

        case ')':
            // 若栈为空, 则说明有多余的右括号, 则匹配失败
            if (!IsEmpty(&S) && GetTop(&S) == '(') {
                Pop(&S, &pop);
            } else {
                flag = 0;
            }
            break;

        case ']':
            if (!IsEmpty(&S) && GetTop(&S) == '[') {
                Pop(&S, &pop);
            } else {
                flag = 0;
            }
            break;
    }

    scanf(" %c", &ch);
}

if (flag && IsEmpty(&S))
    return TRUE;
else
    return FALSE;
}

```

写法2:

```

bool Matching_Parenthese(void) {
    SqStack S;
    InitStack(&S);
    char x;
    char buffer[1024];
    scanf("%s", &buffer);
    char *ptr = buffer;
    bool flag = true;

    while (*ptr != '\0' && flag) {
        if (*ptr == '(' || *ptr == '[') {
            Push(&S, *(ptr++));

```

```

} else if (*ptr == ')') {
    if (!IsEmpty(&S) && GetTop(&S) == '(') {
        Pop(&S, &x);
        ptr++;
    } else
        flag = false;
} else if (*ptr == ']') {
    if (!IsEmpty(&S) && GetTop(&S) == '[') {
        Pop(&S, &x);
        ptr++;
    } else
        flag = false;
}
}

if (flag && IsEmpty(&S)) {
    return true;
} else {
    return false;
}
}

```

- 10以内的计算器

写出一个只有加减乘除的计算器，要求每一步计算结果小于10

运算符顺序要求如下(#代表结束标志，且#运算级最小)

θ_2	+	-	*	/	()	#
θ_1	>	>	<	<	<	>	>
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

```

char Evaluate() {
    SqStack opnd; //运算数栈
    SqStack optr; //运算符栈
    InitStack(&opnd);
    InitStack(&optr);
    Push(&optr, '#'); //先把#结束表示压入运算符栈
    char ch, theta;
    char a, b;
    scanf(" %c", &ch);

    while ('#' != ch || GetTop(&optr) != '#') {
        //先判断是否为运算符
        if (!IsOperator(ch)) {
            Push(&opnd, ch);
            scanf(" %c", &ch);
        }
    }
}

```

```

} else {
    //比较栈顶运算符和输入运算符比较
    switch (Precede(GetTop(&optr), ch)) {
        //输入运算符大于栈顶运算符，则入栈
        case '<':
            Push(&optr, ch);
            //开始读取下一个运算符
            scanf(" %c", &ch);
            break;
        //如果输入运算符小于栈顶运算符，弹出optr栈顶运算符，并弹出opnd的两个运算数，进行运算
        case '>':
            Pop(&optr, &theta);
            Pop(&opnd, &a);
            Pop(&opnd, &b);
            //把运算结果入栈
            Push(&opnd, operate(a, theta, b)); //
            //注意此时并没有对输入运算符ch进行任何操作，所以不用往后读取字符,
            //即不需要scanf("%c", &ch);
            break;
        //如果相等则说明括号匹配结束
        case '=':
            Pop(&optr, &theta);
            scanf(" %c", &ch);
            break;
    }
}
//返回opnd栈顶，表达式的最终结果
return GetTop(&opnd);
}

```

测试代码:

```

#include "Stack.h"
#define OPERATORMAX 7

char operators[OPERATORMAX] = { '+', '-', '*', '/', '(', ')', '#'};
int LocateChar(char *array, char theta);
char operate(char a, char theta, char b);
char Precede(char theta1, char theta2);
char Evaluate(void);
int main(void) {

    printf("%c ", Evaluate());
    // printf("%c", Precede('(', '/'));
    // printf("%c", operate('2', '*', '4'));
    system("pause");
    return 0;
}

char Precede(char theta1, char theta2) {

    int index1 = LocateChar(operators, theta1);
    int index2 = LocateChar(operators, theta2);

```

```

char precedences[OPERATORMAX][OPERATORMAX] = {
    {'>', '>', '<', '<', '>', '>'}, {'>', '>', '<', '<', '<', '>', '>'},
    {'>', '>', '>', '>', '<', '>'}, {'>', '>', '>', '>', '<', '>', '>'},
    {'<', '<', '<', '<', '<', '='}, {'>', '>', '>', '>', '0', '>', '>'},
    {'<', '<', '<', '<', '0', '='}};

return precedences[index1][index2];
}

char operate(char a, char theta, char b) {
    int val_a = atoi(&a);
    int val_b = atoi(&b);
    int result;
    switch (theta) {
        case '+':
            result = val_a + val_b;
            break;
        case '-':
            result = val_a - val_b;
            break;
        case '*':
            result = val_a * val_b;
            break;
        case '/':
            result = val_a / val_b;
            break;
    }
    char val[24];
    itoa(result, val, 10);
    return val[0];
}

int LocateChar(char *array, char theta) {
    for (int i = 0; i < OPERATORMAX; i++) {
        if (array[i] == theta)
            return i;
    }
    return -1;
}

bool IsOperator(char a) {
    if (-1 == LocateChar(operators, a)) {
        return false;
    } else
        return true;
}

```

3. 字符串, 数组, 广义表

- 字符串匹配

字符串的存储结构可以分为两类：

- 顺序存储结构：

```
typedef struct __SString {
    char ch[MAXLEN + 1]; //为了代码理解方便，不使用数组0号位置
    int length;
} SString;
```

- 链式存储结构：

```
#define CHUNKSIZE 80 //每一节点的最大长度
typedef struct __Chunk {
    char ch[CHUNKSIZE + 1];
    __Chunk *next;
} Chunk;

typedef struct __LString {
    Chunk *head, *tail;
    int curlen; //当前字符串长度
} LString;
#endif
```

本笔记只记录顺序存储结构的字符串

输入字符串函数和初始化

```
int GetLength(SString *S) {
    int cnt = 0;
    int i = 1; //因为不使用0号位置，从1开始

    while (S->ch[i] != '\0') {
        i++;
        cnt++;
    }

    return cnt;
}

void InputString(SString *S) {
    scanf(" %s", &S->ch[1]);
    S->length = GetLength(S);
}
```

BF算法(Brute Force)

```
int BF_Match(SString S, SString T, int position) {
    int i = position;
    int j = 1;
```

```

while (i <= S.length && j <= T.length) {
    if (S.ch[i] == T.ch[j]) {
        i++;
        j++;
    } else {
        // i-j+1 是回溯到初始匹配位置
        i = i - j + 1 + 1;
        j = 1;
    }
}
//如果匹配成功, 那么此时i和j都会自增+1, 此时j>T.length
if (j > T.length)
    return i - T.length;
else
    return 0;
}

```

BF算法效率分析

令主串长度为 n , 子串长度为 m , 若从主串的第 i 个位置开始与模式串匹配成功

则在前 $i-1$ 趟匹配中字符总共比较了 $i-1$ 次

若第 i 趟匹配成功, 则比较次数为 m , 总比较次数为 $i-1+m$

$$\sum_{i=1}^{n-m+1} p_i(i-1+m) = \frac{1}{n-m+1} \sum_{i=1}^{n-m+1} i-1+m = \frac{1}{2}(m+n)$$

即 $O(n+m)$

最坏情况：

假设从主串对的第 i 个位置开始与模式串匹配成功, 则在前 $i-1$ 趟中总共比较了 $(i-1) \times m$ 次

若第 i 趟匹配成功, 则第 i 趟需要 m 次匹配, 总比较次数为 $(i-1) \times m + m = im$

$$\sum_{i=1}^{n-m+1} p_i(im) = \frac{1}{n-m+1} \sum_{i=1}^{n-m+1} = \frac{1}{2}m(n-m+2)$$

$O(nm)$

KMP算法

令字符串 S , 和模式串 T

(1) 匹配到 t_k 和 S_i 时失匹, 则 $t_1 \dots t_{k-1} = s_{i-k+1} \dots s_{i-1}$

s 下标由来: $[(i-1-(k-1)+1)]$ $k-1$ 为失匹之前成功匹配的子串长度

(2) j 指向 T 中失配字符, 则有 $t_{j-k+1} \dots t_{j-1} = s_{i-k+1} \dots s_{i-1}$

t 下标由来: $[(j-1-(k-1)+1)]$

(3) 由(1)(2)可知, $t_1 \dots t_{k-1} = t_{j-k+1} \dots t_{j-1}$

由(3)可知, 如果 s_i 和 t_j 失匹, 无需从 t_1 开始和 s_i 比较, 只需要把 t_k 滑动到 s_i 的位置比较即可

令 $next[j] = k$ 表示在匹配中 t_j 失匹时, 让 t_k 与 s_i 比较

可得 $next[j]$ $\begin{cases} 0 & j=1, \text{ 因为当 } t_1 \text{ 失匹时字符串来到 } t_0 \text{ 位置(空串), 有 } t_0 = s_i \dots s_{i-1} \text{ (空串)} \\ \max\{k | 1 < k < j \text{ 且 } t_1 \dots t_{k-1} = t_{j-k+1} \dots t_{j-1}\} & \\ 1 & t_1 \dots t_{k-1} \neq t_{j-k+1} \dots t_{j-1}, \text{ 需要从 } t_1 \text{ 开始重新比较} \end{cases}$

求next数组

求 $next$ 数组的过程实际上是 *induction* 的过程，即如果 i 存在，那么 $i - 1$ 必然存在
也是模式串的字串匹配模式串的过程

设 $next[j] = k$, 即有 $t_1 \dots t_{k-1} = t_{j-k+1} \dots t_{j-1}$

第一种情况 $t_k = t_j$

此时则有 $t_1 t_2 \dots t_{k-1} t_k = t_{j-k+1} \dots t_{j-1} t_j$

那么则有 $next[j + 1] = k + 1 = next[j] + 1$

第二种情况 $t_k \neq t_j$

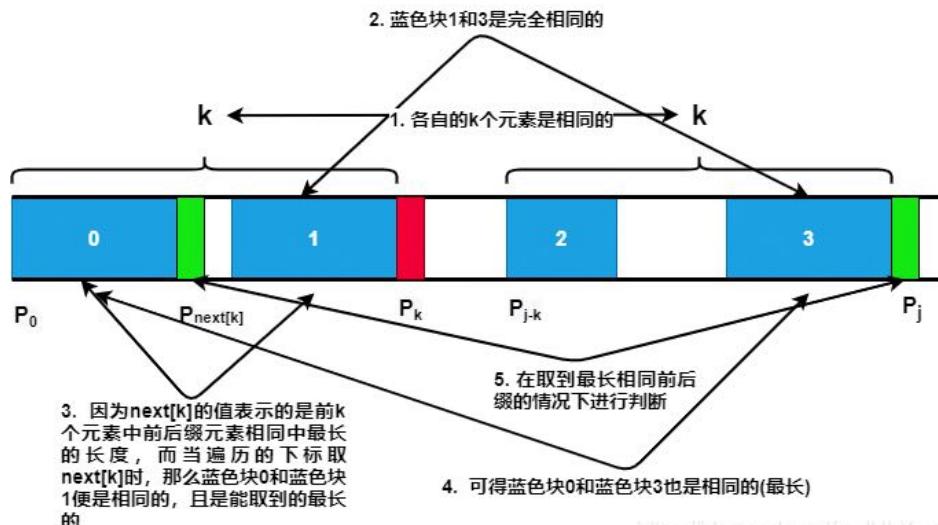
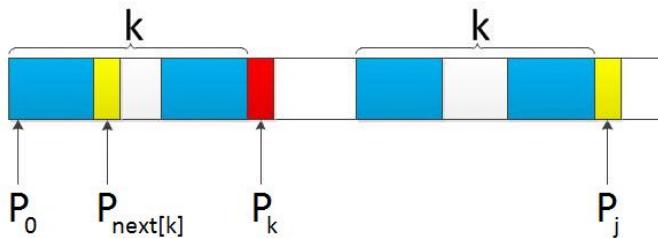
说明在 k 处失匹，则回溯到 $next[k] = k'$

(1) 如果 $t_{k'} = t_j$, 如下图

那么 $next[j + 1] = k' + 1 = next[k] + 1$

(2) 如果 $t_{k'} \neq t_j$

那么一直回溯 $next[next[next[k]]] \dots$, 若出现 $t_{k'} = t_j$ 则同上，若未出现 $next[j + 1] = 1$



https://blog.csdn.net/y_JULY_y

未优化前，求next数组

```
void GetNext(SSString T, int *next) {
    next[1] = 0;
    int j = 0;
    int i = 1;

    while (i < T.length) {
        if (0 == j || T.ch[i] == T.ch[j]) {
            i++;
            j++;
            next[i] = j;
        } else {
            j = next[j];
        }
    }
}
```

考虑如下问题，如果回溯的 $next[j]$ 也等于 t_j 会出现什么问题

会出现无意义比较，解决办法让 $next[j]$ 也进行回溯 $next[j] = next[next[j]]$

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

a	b	a	c	a	b	a	b	c
---	---	---	---	---	---	---	---	---

a	b	a	b
-1	0	0	1

优化后

```
void GetNext(SSString T, int *next) {
    next[1] = 0;
    int j = 0;
    int i = 1;
    //! 注意:while里面 不能写i<=T.length, 因为如果这么写, 那么i经过++后, i=T.length+1
    //! 因为操作的内存是动态的, 这样会导致内存泄漏
    //! 所以必须是i<T.length, 这样最后i才会指向T.length
    while (i < T.length) {
        if (0 == j || T.ch[i] == T.ch[j]) {
            i++, j++;
            //在未优化之前, 如果ch[i]匹配失败, 那么会回溯到ch[j]上
            //如果ch[j] == ch[i]那么说明匹配仍然会失败, 因为ch[i]匹配失败
            if (T.ch[i] == T.ch[j]) {
                //在i++, j++之后提前判断是否相等
                next[i] = next[j];
            } else {
                next[i] = j;
            }
        } else {
            j = next[j];
        }
    }
}
```

KMP算法

```
int KMP_Match(SSString S, SSString T, int position) {
    int *next = (int *)malloc(sizeof(int) * (T.length + 1));
    GetNext(T, next);

    int i = position;
    int j = 1;

    while (i <= S.length && j <= T.length) {
        if (j == 0 || S.ch[i] == T.ch[j]) {
            i++;
            j++;
        }
```

```

    } else {
        j = next[j];
    }
}
free(next);
if (j > T.length)
    return i - j;
else
    return 0;
}

```

测试代码

```

#include "SString.h"

int main(void) {
    SString S, T;
    InputString(&S);
    InputString(&T);
    int i = KMP_Match(S, T, 1); //把KMP改成BF就是BF算法
    printf("%d", i);

    system("pause");
    return 0;
}

```

字符串匹配玩具

输入一个病毒序列长度为 m , 从文件中读取病人样本, 确认病人是否被感染

注意: 病毒序列是环状的

比如病毒序列为: xyz , 那么 yzx, zxy 都属于病毒

思路: 利用一个数组存储病毒序列, 并把该序列扩大到 $2m$ 长度, 依此比较

实现代码

```

#include "SString.h"
#include <fstream>
#define FILENAME "Virus.txt"
//获取文件中样本个数
int File_GetNumber();

int main(void) {
    //创建结构体Virus存储病毒信息
    SString Virus;
    //输入病毒序列
    InputString(&Virus);
    //获取文件中样本个数
    int number = File_GetNumber();
    // id用户标记文件中第几个样本
    int id = 1;
    //创建文件输入对象
    std::ifstream ifs(FILENAME);
    //判断文件是否打开成功
    if (!ifs.is_open()) {
        printf("Failed!\n");
    }
}

```

```

    exit(0);
}

//如果打开成功,扩大病毒序列, 如 xyz 扩大称 xyzxyz
for (int i = Virus.length, j = 1; j <= Virus.length; j++) {
    Virus.ch[i + j] = Virus.ch[j];
}

//因为病毒序列被扩大, 所以要重新设置'\0'
Virus.ch[2 * Virus.length + 1] = '\0';

while (number--) {
    //创建样本结构体
    SString sample;
    //设置检测是否被感染标记flag
    int flag = false;
    // ifs读文件, 并输入到sample上面
    ifs >> &sample.ch[1];
    //初始化sample的长度
    sample.length = GetLength(&sample);
    //设置临时结构体temp, 用于获取不同的病毒序列
    SString temp;
    temp.length = Virus.length;
    //把不同的序列赋值给temp
    for (int i = 0; i < Virus.length; i++) {
        for (int j = 1; j <= Virus.length; j++) {
            temp.ch[j] = Virus.ch[j + i];
        }
    }
    //为temp设置结束标识符
    temp.ch[Virus.length + 1] = '\0';
    //此时已经获取了病毒序列储存在temp中, 把病毒样本和病毒序列比对
    flag = KMP_Match(sample, temp, 1);
    if (flag) { //如果flag!=0说明检测成功
        printf("%d infected\n", id++);
        break;
    }
}
if (0 == flag)
    printf("%d Not infected\n", id++);
}

system("pause");
return 0;
}

int File_GetNumber() {
    std::ifstream ifs(FILENAME);
    char buffer[1024];
    int cnt = 0;
    // ifs会一直读取字符输入到buff数组上, 直到遇到空格等制表符停止
    while (ifs >> buffer) {
        cnt++;
    }
    return cnt;
}

```

• 数组

声明格式：数据类型变量名称[行数][列数]

如： $\text{int num}[5][8]$

一个二维数组也可以被定义成一维数组

例如：

```
typedef elemtype array2[m][n];
等价于
typedef elemtype array1[n];
typedef array1 array2[m];
```

三维数组：每个元素都是二维数组，且二维数组中的每个元素又都是一维数组

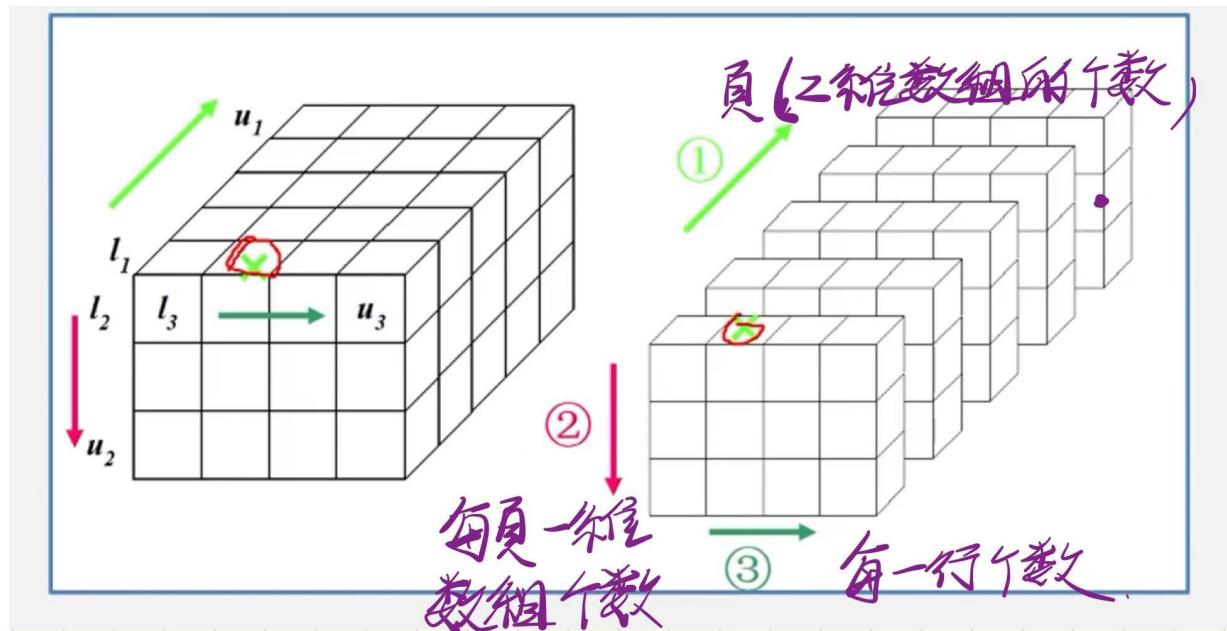
n 维数组：每个元素都是 $n - 1$ 维数组，且 $n - 1$ 维数组的每个元素都是 $n - 2$ 维数组

... 2维数组中的每个元素都是一维数组

若有三维数组 $a[m_1][m_2][m_3]$ 各元素维 m_1, m_2, m_3

则 $LOC(i_1, i_2, i_3) = a + i_1 \times m_2 \times m_3 + i_2 \times m_3 + i_3$

可以抽象成下图



若有 n 维数组，各个维度的元素个数为： $m_1, m_2, m_3 \dots, m_n$

则 $LOC(i_1, i_2, i_3, \dots, i_n) = a + \left(\sum_{j=1}^{n-1} \prod_{k=j+1}^n m_k \right) + i_n$

• 广义表(Generalized List)

广义表通常记作： $LS = (a_1, a_2, \dots, a_n)$

LS 为表名通常用大写字母表示

n 为长度

a_i 为表的元素

表头 : 若 LS 非空 ($n \geq 1$), 则第一元素 a_1 就是表头

$$\text{head}(LS) = a_1$$

表尾 : 除表头之外的其他元素组成的表

$$\text{tail}(LS) = (a_2, a_3, \dots, a_n)$$

注意 : 表尾不是一个元素, 而是一个子表

例 :

$$A = () \text{ 空表, } n = 0$$

$$B = (()), n = 1, \text{ head} \text{ 和 } \text{tail} \text{ 都是 } (), \text{ 但并不是同一 } ()$$

$$C = (a, (b, c)), n = 2, \text{head}(C) = a, \text{tail}(C) = ((b, c))$$

$$E = (C, D), n = 2, \text{head}(E) = C, \text{tail}(E) = (D)$$

$$F = (a, F), n = 2, \text{head}(F) = a, \text{tail}(F) = (F) = ((a, F)) = ((a, (a, F))) \dots$$

广义表的性质:

(1) 广义表中的元素有相对次序 : 一个直接前驱和一个直接后继

(2) 广义表的长度定义为最外层所包含的元素个数 (3) 广义表的深度定义为该广义表展开后所含括号的层数

如 : $A = (b, c)$ 深度1, $B = (A, d)$ 深度2, $C(f, B, h)$ 深度3

注意 : 原子的深度为0, 空表的深度为1

(4) 在广义表中可以为其他广义表共享, 如上述广义表 B 就共享 A

(5) 广义表可以是一个递归的表, 如上述 F 广义表

注意 : 递归表的长度是有限的, 但是深度是 ∞

广义表的基本运算

$$D = (E, F) = ((a, (b, c)), F)$$

$$\text{GetHead}(D) = E \quad \text{GetTail}(D) = (F)$$

$$\text{GetHead}(E) = a \quad \text{GetTail}(E) = ((b, c))$$

$$\text{GetHead}(((b, c))) = (b, c) \quad \text{GetTail}(((b, c))) = ()$$

$$\text{GetHead}((b, c)) = b \quad \text{GetTail}((b, c)) = (c)$$

$$\text{GetHead}((c)) = c \quad \text{GetTail}((c)) = ()$$

4. 树、二叉树、森林

• 定义

树是 $n (n \geq 0)$ 个节点的有限集合

当 $n = 0$ 时为空树

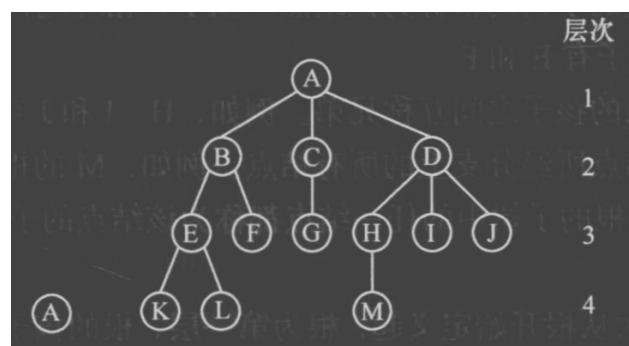
有且仅有一个称之为根的节点

令一颗树为 T

除根节点外, 可分为 $m (m > 0)$ 个互不相交的有限集, T_1, T_2, \dots, T_m , 其中一个集合又是一棵树

称其为 T 的子树

如下图



• 树的基本术语

节点的度(*degree*)

节点拥有的子树个数成为节点的度

如上图 $\text{degree}(A) = 3, \text{degree}(C) = 1, \text{degree}(F) = 0$

树的度(*degree*)

树的度是树内各节点度的最大值

上图的度为3

层次(*Level*)

节点的层次从根开始定义，根为第一层，根的孩子为第二层

树中任意节点的层次 = 它的双亲层次 + 1

高度(*Height*)

树中节点的最大层次成为树的高度

如上图 $\text{Height}(T) = \max(\text{Level}(t)) = 4$

森林

是 $m(m \geq 0)$ 颗互不相交的树的集合

对任意一棵树而言，其子树组成森林

• 二叉树的定义

是 $n(n \geq 0)$ 个节点组成的集合， $n = 0$ 时为空树

对于非空树 T

有且仅有一个根节点

除根结点外，有互不相交的 T_l, T_r 两棵子树(可以为空树)，分别称其为 T 的左右子树

• 二叉树的性质

(1) 每个节点最多有两棵子树(即 $\text{degree}(T) \leq 2$)

(2) 子树有左右之分，不可颠倒

• 二叉树定理

定理1

在二叉树的第 i 层上，最多有 2^{i-1} 个节点

定理2

深度为 k 的二叉树，最多有 $2^k - 1$ 个节点

Proof

$$\sum_{i=1}^k 2^{i-1} = 2^0 + 2^1 + \cdots + 2^{k-1} = \frac{2^k - 2^0}{2 - 1} = 2^k - 1$$

定理3

$$n_0 = n_2 + 1$$

Proof

令一颗二叉树 T 有 n 个节点，其中 n_1, n_2, n_0 分别为 $degree = 1, 2, 0$ 的节点个数

$$\text{即有 } n = n_0 + n_1 + n_2 \quad (1)$$

可知 T 的根无双亲节点，而其他的节点都有双亲，令这些节点的个数为 B

$$\text{即 } B = n - 1 = 0 \cdot n_0 + 1 \cdot n_1 + 2 \cdot n_2 = n_1 + 2n_2$$

$$\text{由(1)可知 } n - 1 = n_0 + n_1 + n_2 - 1 = n_1 + 2n_2$$

$$\text{所以 } n_0 = n_2 + 1$$

• 完全二叉树性质及定理

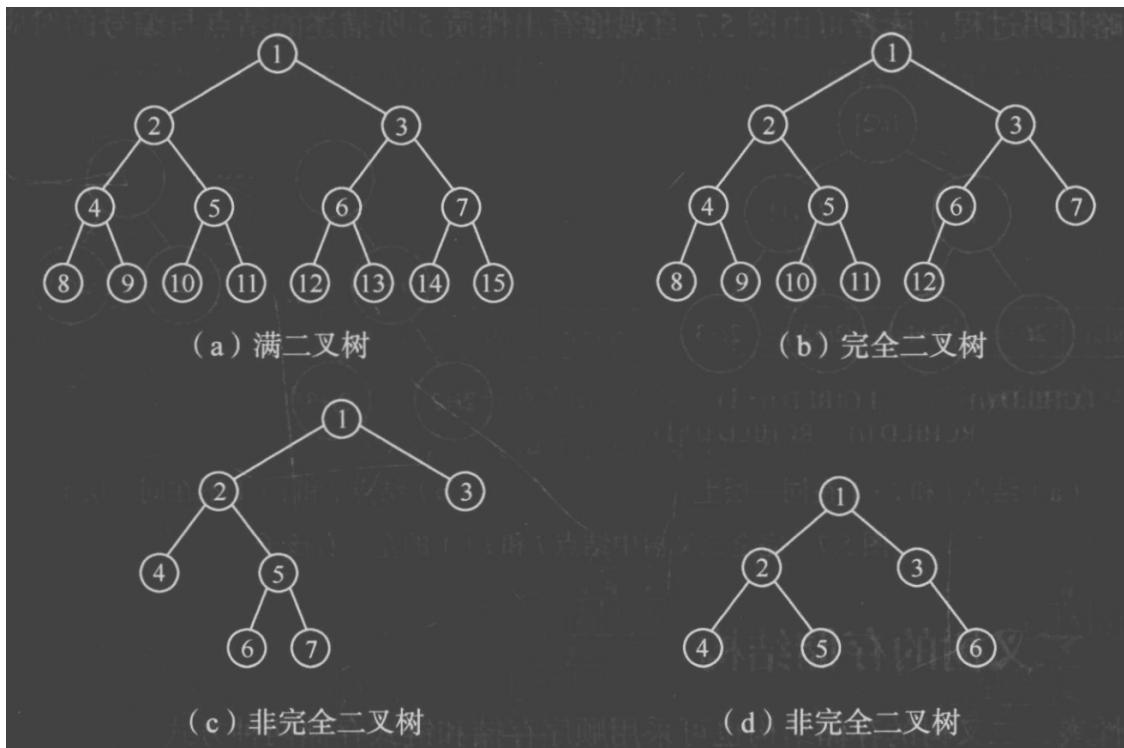
定义：

深度为 k 的，由 n 个节点的二叉树

当且仅当其每一个节点都与深度为 k 的满二叉树中编号从1到 n 的节点一一对应时

称其为完全二叉树

如下图



特点

对完全二叉树的任意节点

若其右分支的子孙最大层次为： l ，则其左分支的子孙最大层次为： l 或 $l + 1$

定理1

若一个完全二叉树有 n 个节点，深度为 k ，则 $k = \lfloor \log_2^n \rfloor + 1$

Proof :

根据完全二叉树性质可知：

$$2^{k-1} - 1 < n \leq 2^k - 1 \implies 2^{k-1} \leq n < 2^k$$

对其 \log_2 化：

$$k - 1 \leq \log_2^n < k$$

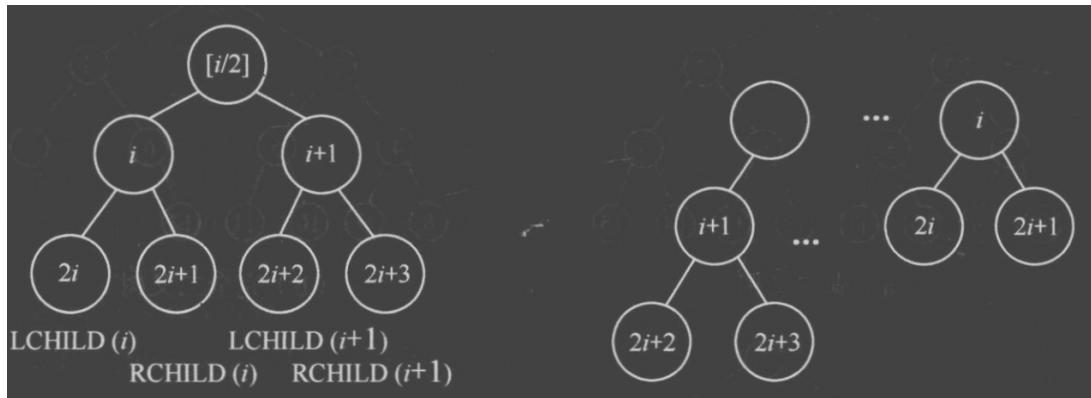
因为 $k \in \mathbb{Z}^+$, 所以 $k = \lfloor \log_2^n \rfloor + 1$

定理2

一颗完全二叉树 T , 有 n 个节点，若按照节点层次(从左到右给其编号)，则对于任意节点 i ：

$$\begin{cases} i = 1 & \text{无双亲} \\ i \neq 1 \text{ and } i \leq n & i \text{ 的双亲为 } \lfloor i/2 \rfloor \\ 2i > n & \text{则 } i \text{ 无左孩子} \\ 2i \leq n & 2i \text{ 为 } i \text{ 的左孩子} \\ 2i + 1 > n & \text{则 } i \text{ 无右孩子} \\ 2i + 1 \leq n & \text{则 } 2i + 1 \text{ 为 } i \text{ 的右孩子} \end{cases}$$

如下图



• 链式二叉树

- 存储结构

```
typedef char BitreeElemType;
typedef struct __BiNode {
    BitreeElemType data;
    __BiNode *lchild, *rchild;
} BiNode, *BiTree;
```

- 遍历方式(递归)

```
void InOrder(BiTree T) {
    if (!T)
        return;
    InOrder(T->lchild);
    printf("%c ", T->data);
```

```

    InOrder(T->rchild);
}

void PreOrder(BiTree T) {
    if (!T)
        return;
    printf("%c ", T->data);
    PreOrder(T->lchild);
    PreOrder(T->rchild);
}

void PostOrder(BiTree T) {
    if (!T)
        return;
    PostOrder(T->lchild);
    PostOrder(T->rchild);
    printf("%c ", T->data);
}

```

- 遍历方式(非递归)

```

void InOrder_unrec(BiTree T) {
    SqStack S;
    InitStack(&S);
    BiTree p = T;
    // BiTree pop = NULL;
    while (p || !IsEmpty(&S)) {
        if (p) {
            // 栈顶元素始终是p的parent节点
            Push(&S, p);
            p = p->lchild;
        } else {
            Pop(&S, &p); // 把栈顶元素弹出给p, 此时变成了原p的parent
            // 此时p的左孩子为空, 根据in-order规则, 输出根节点p, 然后再以相同的方式遍历他的右孩子
            printf("%c", p->data);
            p = p->rchild; // 输出根节点后, 遍历其右子树
        }
    }
}

```

- 先序遍历创建二叉树

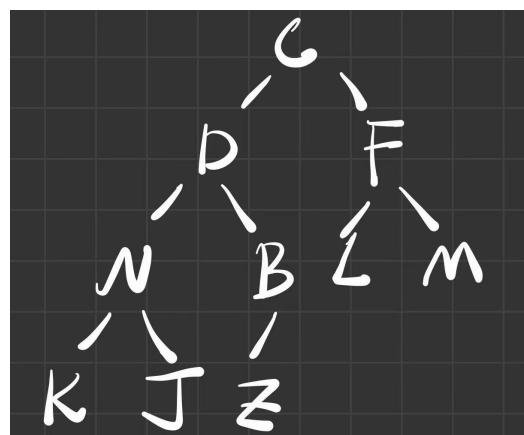
```

void Creat_BiTree_Pre(BiTree *T) {
    //根据输出字符识别虚空节点，'#' 代表虚空节点
    char e;
    scanf(" %c", &e); //输入字符
    if ('#' == e)
        *T = NULL; //设置虚空节点
    else {
        *T = (BiTree)malloc(sizeof(BiNode));
        (*T)->data = e;
        Creat_BiTree_Pre(&(*T)->lchild);
        Creat_BiTree_Pre(&(*T)->rchild);
    }
}

```

- 测试(创建, 遍历)

输入如下一棵树, 按照中序遍历方式输出



```

#include "BinaryTree.h"

int main(void) {
    BiTree T = NULL;
    Creat_BiTree_Pre(&T);
    InOrder_unrec(T); //非递归
    // InOrder(T); 递归调用
    system("pause");
    return 0;
}
// 输入数据: CDNK##J##BZ###FL##M##
// 输出: KNJDZBCLFM

```

- 层次遍历算法

```

void LevelOrder(BiTree T) {
    BiTree temp = NULL;
    SqQueue Q;

```

```

InitQueue(&Q);
if (T) // 先入队根
    EntryQ(&Q, T);
while (!IsEmpty(&Q)) {
    OutQ(&Q, &temp);           // temp暂存弹出值
    printf("%c", temp->data); // 输出
    if (temp->lchild)         // 在入队temp的左孩子和右孩子
        EntryQ(&Q, temp->lchild);
    if (temp->rchild)
        EntryQ(&Q, temp->rchild);
}
}

/*
input: CDNK##J##BZ###FL##M##
-----
output: CDFNBLMKJZ
*/

```

- 复制二叉树

```

void Copy(BiTree *Tnew, const BiTree T) {
    if (!T) {
        *Tnew = NULL;
        return;
    } else {
        *Tnew = (BiTree)malloc(sizeof(BiNode));
        (*Tnew)->data = T->data;
        Copy(&(*Tnew)->lchild, T->lchild);
        Copy(&(*Tnew)->rchild, T->rchild);
    }
}

```

- 求深度和节点数

```

int Depth(BiTree T) {
    if (!T)
        return 0;
    else {
        return Depth(T->lchild) > Depth(T->rchild) ? Depth(T->lchild) + 1
                                                       : Depth(T->rchild) + 1;
    }
}

int Nodes(BiTree T) {
    if (!T)
        return 0;
    else
        return 1 + Nodes(T->lchild) + Nodes(T->rchild);
}

```

- 销毁二叉树(递归和非递归)

```
//递归的方式
void Destroy(BiTTree *root) {
    //销毁操作必须按照后续遍历的顺序
    if (!(*root))
        return;
    else {
        Destroy(&(*root)->lchild);
        Destroy(&(*root)->rchild);
        free(*root);
        *root = NULL;
    }
}
```

```
void Destroy_unrec(BiTTree *root) {
    //此算法的思路是:创建一个链栈, 分别依次入栈
    //<根节点>,<根节点的左孩子>,<左孩子的左孩子>,...<最后一个左孩子>
    //利用lchild充当链栈的next.
    //当lchild发生变法时, 树的结构必然发生变化, 所以需要一个current指针来指向入栈节点
    //当所有根节点左子树的所有左孩子入栈完毕, 使current指向栈顶top
    //判断栈顶节点是否有右孩子, 如果有则current指向它的右孩子, 并把current入栈
    //如果没有右孩子, 则出栈, 并释放栈顶
    //重复此步骤
    BiTree top = NULL;           //初始化一个链栈
    BiTree temp;                 //暂存current
    BiTree current = (*root)->lchild; //因为要把root入栈, current指向root的左孩子
    (*root)->lchild = top; //因为lchild充当链栈的next, 所以lchild指向top
    top = *root;                //此时root变成栈顶

    while (top) {
        while (current) { //如果current不为空, 则入栈
            temp = current; //暂存current
            current = current->lchild;
            temp->lchild = top; //入栈
            top = temp;
        }
        //此时已经把root的左孩子, 左孩子的左孩子.....入栈, 栈顶为树的最左路径的最后一个节点
        current = top;      // current 和 top 指向同一个节点
        if (current->rchild) { //如果栈顶有右孩子
            temp = current;
            // current指向它的右孩子, 然后再次循环上面的步骤
            current = current->rchild;
            //因为栈顶的右孩子地址已经被current保存, 所以可以把该指针指向NULL
            temp->rchild = NULL;
        } else { // 如果栈顶元素没有左孩子
            //此时top和current指向同一节点
            top = top->lchild;
            free(current);
            current = NULL;
        }
        //回到上面的循环时, 因为current为空, 所以不会有元素入栈, 并且current会指向栈顶
    }
}
```

```

    }
}

*root = NULL; //销毁完毕，把root指向空
}

```

- 头文件

```

#include "Queue.h"
#include "Stack.h"
#include "define.h"

#ifndef __BINARYTREE_H
#define __BINARYTREE_H

typedef char BitreeElemType;
typedef struct _BiNode {
    BitreeElemType data;
    _BiNode *lchild, *rchild;
} BiNode, *BiTree;

void Creat_BiTree_Pre(BiTree *T);
void InOrder(BiTree T);
void InOrder_unrec(BiTree T);
void PreOrder(BiTree T);
void PostOrder(BiTree T);
void LevelOrer(BiTree T);
void Copy(BiTree *Tnew, const BiTree T);
int Depth(BiTree T);
int Nodes(BiTree T);
void Destroy_unrec(BiTree *root);
void Destroy(BiTree *root);
#endif

```

• 线索二叉树 Threaded Binary Tree(了解即可)

问题，如何寻找特点遍历顺序中二叉树节点的前驱和后继？

办法：

- (1)通过遍历，浪费时间
- (2)给结构体内增加前驱和后继指针，增加存储负担
- (3)利用二叉链表中的空指针域

定理：

如果一个二叉树有 n 个节点，那么空指针域为 $n + 1$ 个

Proof :

总指针域为 $2n$ 个，除去根节点，一个有 $n - 1$ 个节点，即需要 $n - 1$ 个指针域

$$2n - (n - 1) = n + 1$$

线索二叉树的定义：

如果某个节点的左孩子为空，那么利用左孩子指针域，把它指向它的前驱

如果右孩子为空，则指向它后继

这种改变指向的指针称为线索

- 数据类型定义

```
typedef struct BiThrNode {
    int data;
    int ltag, rtag; //为了区分左右孩子指向为空还是前驱或后继，新增ltag, rtag
    BiThrNode * lchild, *rchild;
} *BiThrTree;
```

$$\begin{cases} ltag = 0 & \text{左孩子为空} \\ ltag = 1 & \text{左孩子指向前驱} \end{cases} \quad \begin{cases} rtag = 0 & \text{右孩子为空} \\ rtag = 1 & \text{右孩子指向前驱} \end{cases}$$

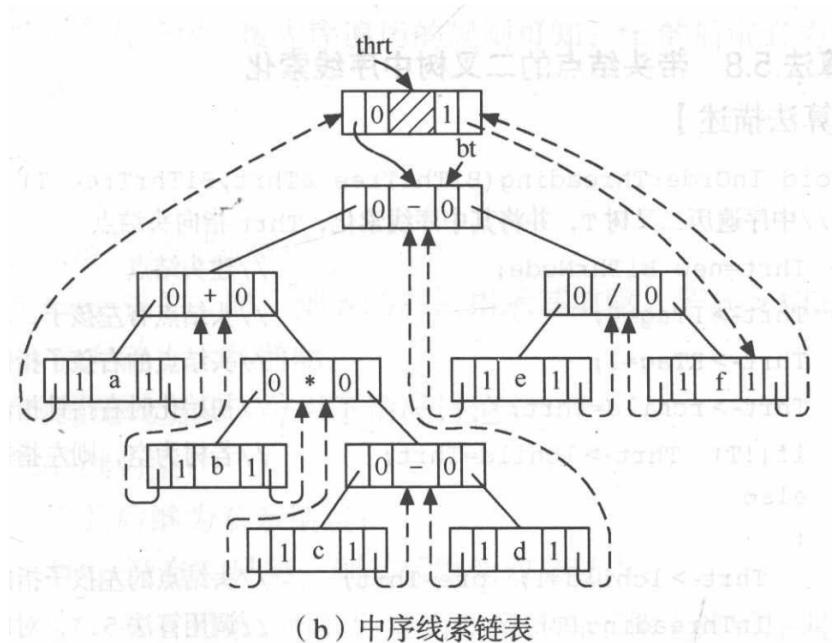
如果按照遍历的顺序，那么序列中第一个元素必然没有前驱

把该节点的左孩子指向一个头指针

该头指针的左孩子指向根节点，右孩子指向序列最后一个节点

一个中序遍历为： $a + b * c - d - e / f$ 的线索二叉树

如下图



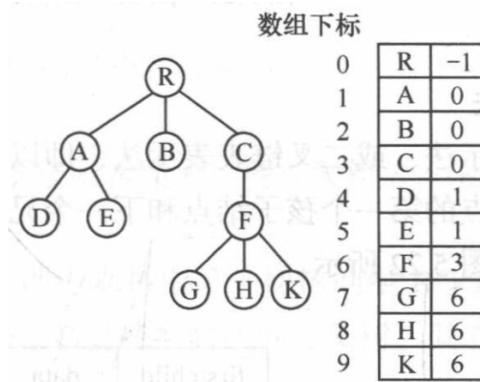
• 树和森林

- 树的存储结构

1. 双亲表示法

```
struct PTNode {
    DataType data;
    int parent; //存放parent在数组中的位置
};
```

```
struct PTree {
    PTNode[MAXSIZE];
    int root; //根节点位置
    int n; //当前节点个数
};
```



2. 孩子链表

```
//孩子节点结构
struct CTNode {
    int chlid;
    CTNode *next;
};
```

```
//双亲结点结构
struct CTBox {
    DataType data;
    int parent; //可有可无，看具体需求
    CTNode * child;
};
```

```
//整体树结构
struct CTree {
    CTBox[MAXSIZE];
    int root;
    int n;
};
```

0	A	•	3	—	5	Λ		
1	B	Λ						
2	C	•	6	Λ				
3	D	Λ						
4	R	•	0	—	1	—	2	Λ
5	E	Λ						
6	F	•	7	—	8	—	9	Λ
7	G	Λ						
8	H	Λ						
9	K	Λ						

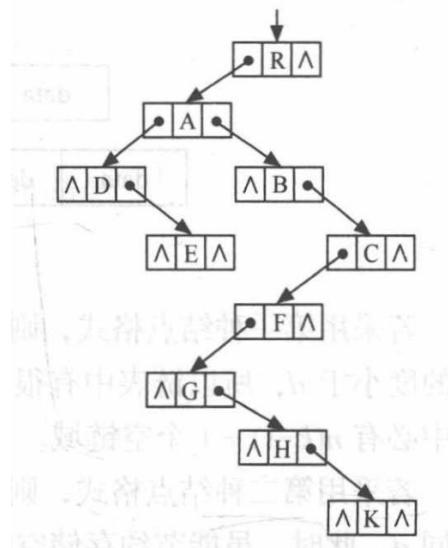
(a) 孩子链表

0	4	A	•	3	—	5	Λ		
1	4	B	Λ						
2	4	C	•	6	Λ				
3	0	D	Λ						
4	-1	R	•	0	—	1	—	2	Λ
5	0	E	Λ						
6	2	F	•	7	—	8	—	9	Λ
7	6	G	Λ						
8	6	H	Λ						
9	6	K	Λ						

(b) 带双亲的孩子链表

3. 孩子兄弟表示法(树转化二叉树的基础)

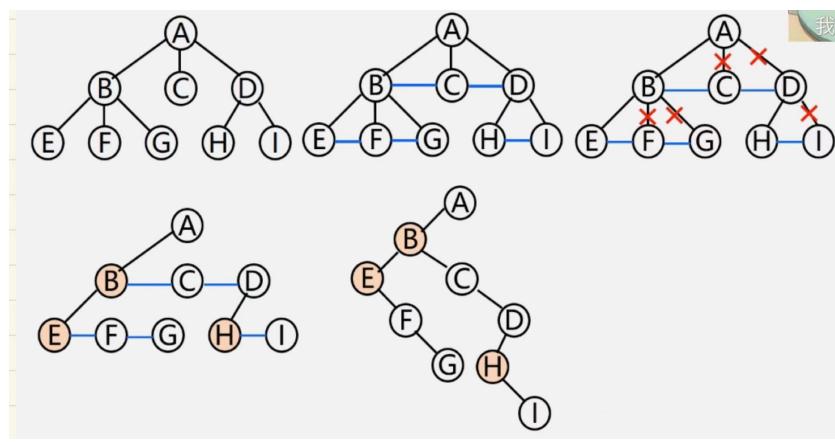
```
//节点的child指针域指向它的第一个孩子, sibling指向第一个兄弟
struct CSNode {
    DataType data;
    CSNode * child, *sibling;
};
```



树与二叉树之间相互转换

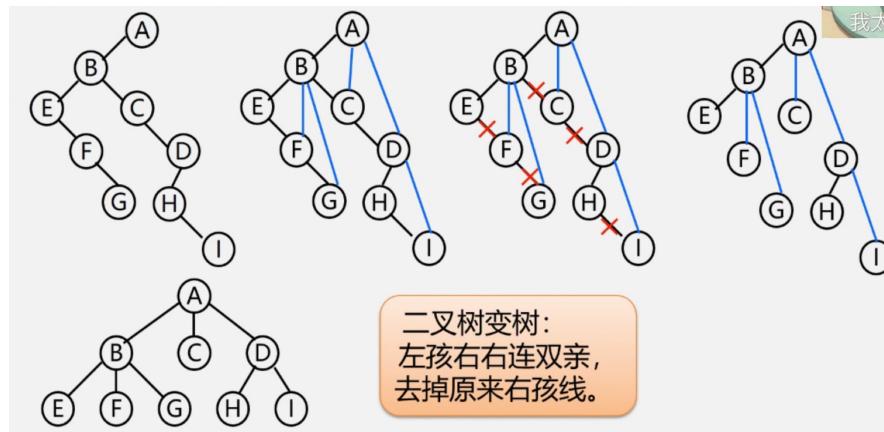
树转二叉树

- (1) 在兄弟之间连线
- (2) 对于任意一个节点 k , 除了其左孩子外, 去除掉所有孩子与其的关系
- (3) 以根节点的左孩子为中心, 顺时针旋转 45° , 再与根节点相连



二叉树转树

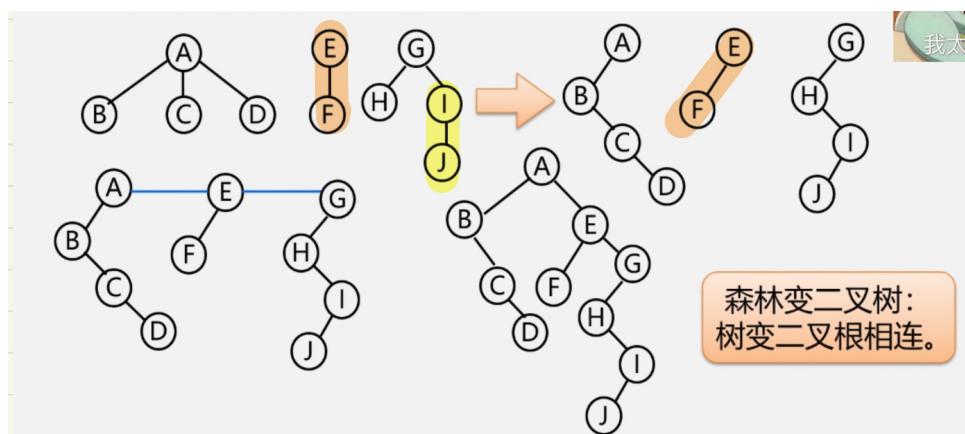
- (1)若 k 节点是双亲节点的左孩子，则将 k 的右孩子，右孩子的右孩子...连接到 k 的双亲结点上
- (2)抹掉所有双亲结点和右孩子之间关系
- (3)整理成树状



- 森林与二叉树之间的转化

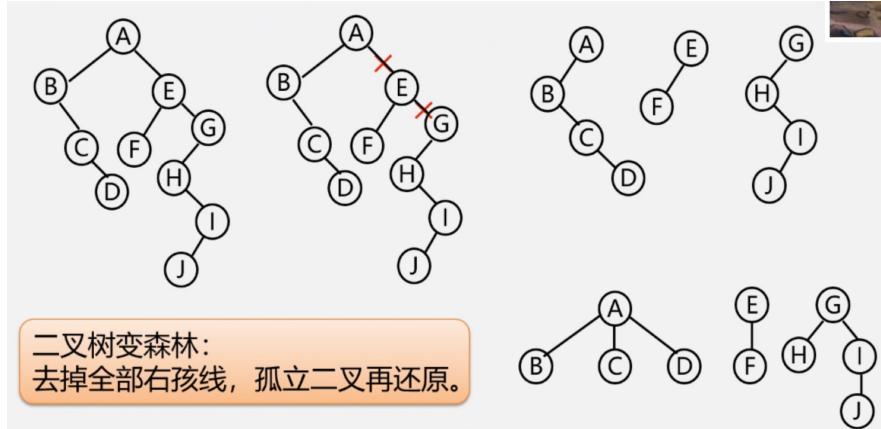
森林转二叉树

- (1)将森林中的各棵树转化为二叉树
- (2)链接这些二叉树的根节点
- (3)第一棵树的根为二叉树的根，进行调整



二叉树转森林

- (1)若 r 是二叉树的根，则取消所有 r 的右孩子，右孩子的右孩子，…之间的关系
- (2)此时有 m 颗孤立的二叉树，把这些二叉树转成树



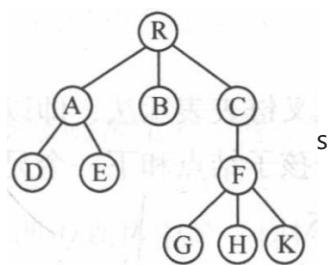
- 树和森林的遍历

树的遍历

先根遍历：若树不为空，则先访问根节点，然后再依次先根遍历遍历各个子树

后根遍历：若树不为空，先依次后根遍历各个子树，再访问根节点

层次遍历：自上到下，从左到右



先根遍历为： $RADEBCFGHK$

后根遍历： $DEABGHKFCR$

森林的遍历

假设有森林 F ，有 n 棵互不相交的树 $\{f_1, f_2, \dots, f_n\}$

将森林看成三部分

- (1)森林中第一棵树的根节点
- (2)森林中第一棵树的所有子树
- (3)森林中2到 n 颗树

先序遍历：

- (1)访问 f_1 的根节点
- (2)先序遍历 f_1 的所有子树
- (3)先序遍历 f_2 到 f_n

即从左到右依次对 f_1, f_2, \dots, f_n 进行先根遍历

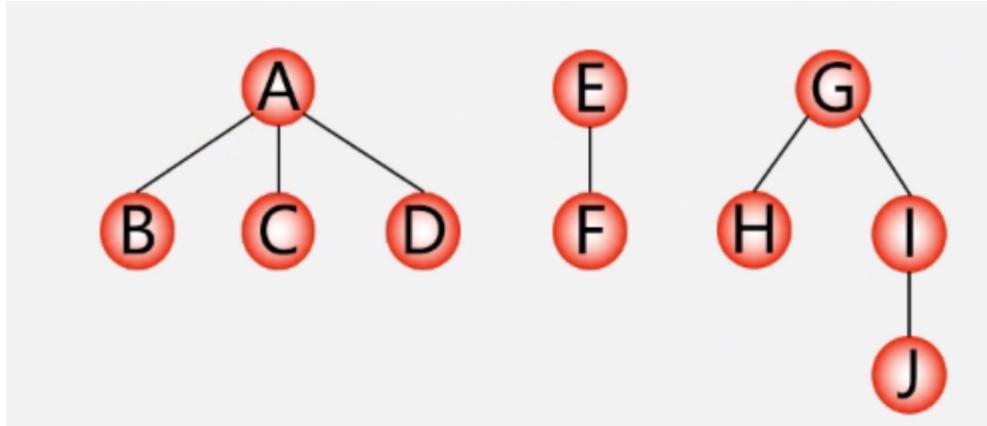
中序遍历：

(1) 中序遍历 f_1 的所有子树

(2) 访问 f_1 的根节点

(3) 中序遍历 f_2 到 f_n

即从左到右依次对 $f_1, f_2 \dots, f_n$ 进行后根遍历



对上图进行先序遍历和中序遍历

先序 : $ABCDEFHIJ$

中序 : $BCDAFEHJIG$

• 哈夫曼树

- 术语

路径：

从树中一个节点到另一个节点之间的分支，组成两个节点间的路径

节点的路径长度：

两节点间路径上的分支数

树的路径长度：

从树根到每一个节点的路径长度之和，记作 TL

节点数目相同的二叉树中，完全二叉树是路径长度最短的二叉树

权(*weight*)：

将树中节点赋给一个含有某种意义的值，这个值叫 *weight*

结点的带权路径长度：

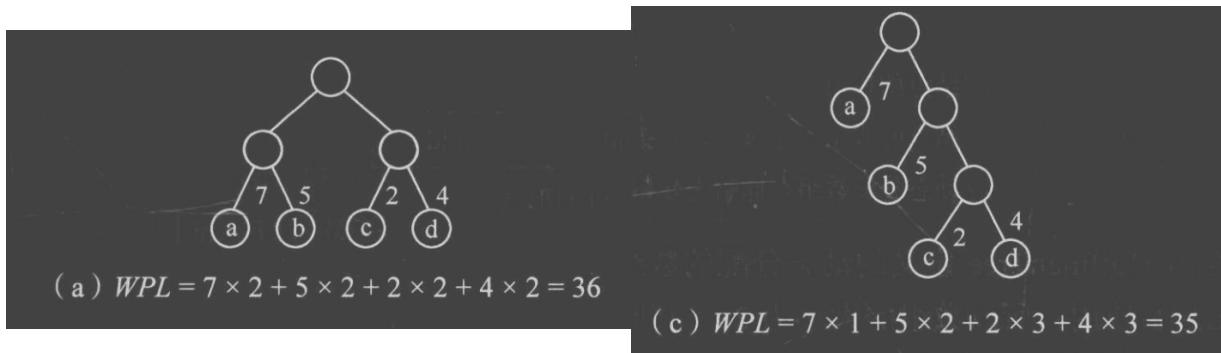
从根节点到该节点之间的路径长度与该节点的乘积，即 $weight \times length$

树的带权路径长度(*Weighted Path Length*)：

树中所有叶子节点的带权路径长度之和

$$WPL = \sum_{k=1}^n w_k l_k$$

如下图



- 最优二叉树

哈夫曼树，又称最优二叉树，即带权路径 WPL 最短的树

注意：

是 $degree$ 相同的树比较之下

完全二叉树(包括满二叉树)不一定是哈夫曼树

哈夫曼树中权值越大的叶子节点离根节点越近

具有相同带权路径的哈夫曼树，节点位置不唯一

- 构造哈夫曼树

- step和定理

(1)根据 n 个给定的权值 $\{w_1, w_2, \dots, w_n\}$ 构成 n 个只有根节点的森林 $F = \{T_1, T_2, \dots, T_n\}$ (森林里全是根)

(2)在 F 中选取两个权值最小的树 T_{min}, T_{max} ，构造一颗新的二叉树 T_{new}

$degree(T_{new}) = degree(T_{min}) + degree(T_{max})$

(3)在 F 中删除 T_{min}, T_{max} ，把 T_{new} 加入 F 中

(4)重复如上步骤，直到 F 中只有一棵树

定理1：

包含 n 个节点的树，需要经过 $n - 1$ 次合并才能形成哈夫曼树

定理2：

把一颗含有 n 个节点的树，转化成哈夫曼树，那么这个哈夫曼树一共有 $2n - 1$ 个

$$n + n - 1 (\text{定理1})$$

- 代码实现

数据类型定义

```
typedef struct __HTNode {
    int weight;
    int parent, lchild, rchild;
} HTNode, *HTree;
```

前置函数，选取两个最小值

```
void Select_Min(const HTree T, int length, int *e1, int *e2) {
    int min1, min2;
    min1 = min2 = INT_MAX;
    int pos1, pos2;
    pos1 = pos2 = 0;

    for (int i = 1; i < length + 1; ++i) {
        if (T[i].parent == 0) { //! parent==0 说明在森林中
            if (T[i].weight < min1) {
                min2 = min1;
                pos2 = pos1;
                min1 = T[i].weight;
                pos1 = i;
            } else if (T[i].weight < min2) {
                min2 = T[i].weight;
                pos2 = i;
            }
        }
    }

    *e1 = pos1;
    *e2 = pos2;
}
```

Creat_Huffman

```
void Creat_Huffman(HTree *T, int n) {
    if (n <= 1)
        return;
    int m = 2 * n - 1; //定理2

    *T = (HTree)malloc(sizeof(HTNode) * (m + 1)); // 0号位置不存元素

    for (int i = 1; i < m + 1; ++i) { //进行初始化
        (*T)[i].lchild = (*T)[i].rchild = 0;
        (*T)[i].parent = 0;
    }

    for (int i = 1; i < n + 1; ++i) { //赋值weight
        scanf(" %d", &(*T)[i].weight);
    }
    //-----初始化完毕，开始构造-----
    int min1, min2; //表示第一小和第二小的位置
    for (int i = n + 1; i < m + 1; ++i) { // n+1号位置为新构造的节点下标
        Select_Min(*T, i - 1, &min1, &min2); //! 核心语句 动态的选择大小

        (*T)[min1].parent = (*T)[min2].parent = i; //合并
        (*T)[i].lchild = min1; //新节点的左右孩子
        (*T)[i].rchild = min2;
        (*T)[i].weight = (*T)[min1].weight + (*T)[min2].weight; //赋值权值
    }
}
```

测试代码

```
#include "HuffmanTree.h"
#define NODES 8
int main(void) {
    HTree T = NULL;
    Creat_Huffman(&T, NODES);

    for (int i = 1; i < 2 * NODES; ++i) {
        printf("weight = %d\tparent = %d\tlchild = %d\trchild = %d\n", T[i].weight,
               T[i].parent, T[i].lchild, T[i].rchild);
    }

    system("pause");
    return 0;
}
/*
*/
input: 7 19 2 6 32 3 21 10
-----
output:
weight = 7      parent = 11      lchild = 0      rchild = 0
weight = 19     parent = 13      lchild = 0      rchild = 0
weight = 2       parent = 9       lchild = 0      rchild = 0
weight = 6       parent = 10      lchild = 0      rchild = 0
weight = 32      parent = 14      lchild = 0      rchild = 0
weight = 3       parent = 9       lchild = 0      rchild = 0
weight = 21      parent = 13      lchild = 0      rchild = 0
weight = 10      parent = 11      lchild = 0      rchild = 0
weight = 5       parent = 10      lchild = 3      rchild = 6
weight = 11      parent = 12      lchild = 9      rchild = 4
weight = 17      parent = 12      lchild = 1      rchild = 8
weight = 28      parent = 14      lchild = 10     rchild = 11
weight = 40      parent = 15      lchild = 2      rchild = 7
weight = 60      parent = 15      lchild = 12     rchild = 5
weight = 100     parent = 0       lchild = 13     rchild = 14
*/

```

• 哈夫曼编码

讨论的背景

在远程通信中传递字符串时，需要转换成二进制的字符串

即，让待传递字符串中出现次数多的字符采用尽可能短的编码
这样的话二进制字符串的编码就会缩短

但，由于二进制只有0和1，所以有可能二进制代码转成字符时，出现二义性
所以要设计长度不等的编码，则必须使任一字符的编码都不是另一个字符编码的前缀
如上形式的编码称为：前缀码

哈夫曼编码

哈夫曼编码是总长最短的前缀码

- (1)统计字符集中每个字符在电文中出现的平均概率(概率越大, 编码要求最短)
- (2)利用哈夫曼树的特点: 权值越大的叶子离根越近; 将每个字符的概率值最为权值, 构造哈夫曼树
 - (3)在哈夫曼树的每个分支上标0和1, 左分支为0, 右分支为1

为什么哈夫曼编码是最短的前缀码?

根据哈夫曼树的特性, 原树中的 n 个节点, 在哈夫曼树中变成了叶子
叶子节点不会是另一个叶子的双亲或是祖先, 所以是前缀码

哈夫曼树的性质

性质1: 哈夫曼编码是前缀码

性质2: 哈夫曼编码是最优前缀码

- 代码实现

```
HuffmanCode Create_HuffmanCode(const HTree HT, int n) {
    // HC和哈夫曼树一样, 不使用0号下标
    HuffmanCode HC = (char **)malloc(sizeof(char *) * (n + 1));
    char *temp_string = (char *)malloc(sizeof(char) * n); //此数组使用0号下标
    temp_string[n - 1] = '\0'; //因为存放字符串所以最后一个位置 '\0'

    for (int i = 1; i < n + 1; ++i) {
        int parent = HT[i].parent; //需要向上回溯
        int current = i;           //回溯中当前节点
        int start = n - 1;         //数组中最后一个位置, 即'\0'
        while (parent) {

            if (current == HT[parent].lchild) //如果是左孩子, 那么为'0'
                temp_string[--start] = '0';
            else
                temp_string[--start] = '1'; //右孩子为 '1'

            current = parent;
            parent = HT[parent].parent;
        }
        //计算长度: 因为start表示字符串的起始下标, n-1表示末尾结束符'\0', 所以 length
        //= n-1-start+1=n-start;
        HC[i] = (char *)malloc(sizeof(char) * (n - start)); //根据长度分配空间
        strcpy(HC[i], &temp_string[start]); //拷贝字符串
    }

    free(temp_string); //释放堆空间
    return HC;
}
```

测试代码

构建如下哈夫曼编码

$$D = \{A, B, C, D, E, F, G\}$$

$$Weight = \{40, 30, 15, 5, 4, 3, 3\}$$

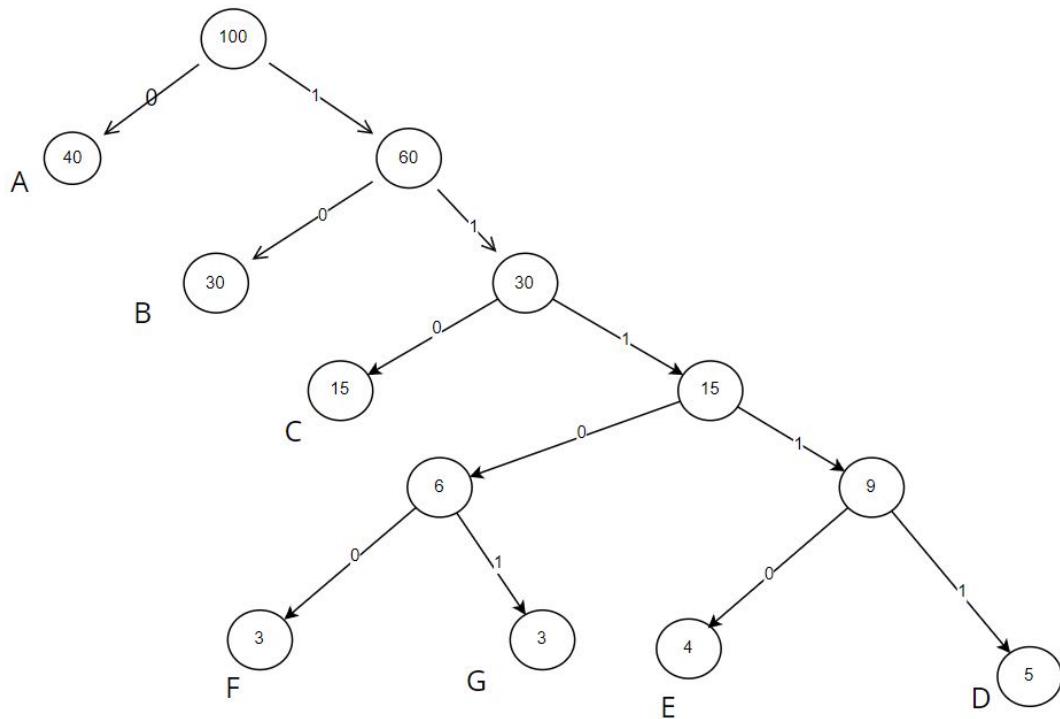
```
#include "HuffmanTree.h"
```

```
#define NODES 7
int main(void) {
    HTree T = NULL;
    Creat_Huffman(&T, NODES);

    HuffmanCode HC = Creat_HuffmanCode(T, NODES);
    int a = 65; // ASCII 65 是'A'
    for (int i = 1; i < NODES + 1; ++i) {
        printf("%c=%s\n", a, HC[i]);
        ++a;
    }

    system("pause");
    return 0;
}
/*
input: 40 30 15 5 4 3 3
-----
output:
A=0
B=10
C=110
D=11111
E=11110
F=11100
G=11101
*/

```



- 整体头文件

```
#include "define.h"
#include "string.h"
#ifndef __HUFFMANTREE_H
#define __HUFFMANTREE_H

typedef struct __HTNode {
    int weight;
    int parent, lchild, rchild;
} HTNode, *HTree;
typedef char **HuffmanCode;

void Select_Min(const HTree T, int length, int *e1, int *e2);
void Creat_Huffman(HTree *T, int n);

HuffmanCode Creat_HuffmanCode(const HTree HT, int n);
#endif
```

5. 图(Graph)

• 图的定义和术语

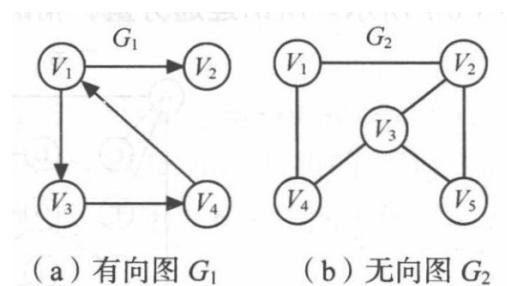
图 : $G(V, E)$

v : 顶点的有限非空集合

E : 边的有限集合

图分为有向图(Digraph)和无向图

如下图



- 完全图

完全图(Complete Graph)

任意两个点都有一条边相连

若有 n 个顶点的无向完全图，则 $edges = \frac{n(n - 1)}{2}$

若有 n 个顶点的有向完全图，则 $edges = n(n - 1)$

- 稀疏图(Sparse Graph)和稠密图(Dense Graph)

稠密图 : $e < n \log n$

稠密图 : 有较多边的图

- 顶点的度(degree)

$\text{degree}(v_x)$: 与该顶点相关联的边的数量

Digraph 中, 顶点的度 = $\text{indegree} + \text{outdegree}$

问题 : 当 Digraph 中, 仅有一个顶点的 indegree 为 0, 其余顶点的 $\text{indegree} = 1$, 此时图为什么形状

答 : 树, 有向树

- 路径(path)

路径 : 接续的边构成的顶点序列

路径长度 : 路径上边的数量或权值之和

回路(loop) : 第一个顶点和最后一个顶点相同的路径

simple 路径 : 路径上的顶点均不相同

simple 回路 : 除路径起点和终点可以相同外, 其余顶点均不相同的路径

- 连通图(Connected Graph)

若连通图 :

在无向图 $G = (V, E)$ 中

若对任何两个顶点 v, u , 都存在 (v, u) 路径, 则 G 是连通图

弱连通图(Weakly Connected) :

若把有向图 G 中所有的边替换成无向边, 此时得到的图为 G 的基图

若它的基图为连通图, 则 G 为若连通

强连通(Strongly Connected) :

任取有向图 $G = (V, E)$ 中两个顶点 v, u , 若 v 和 u 中间存在路径, 则 G 为强连通

- 连通子图和连通分量(Connected Component)

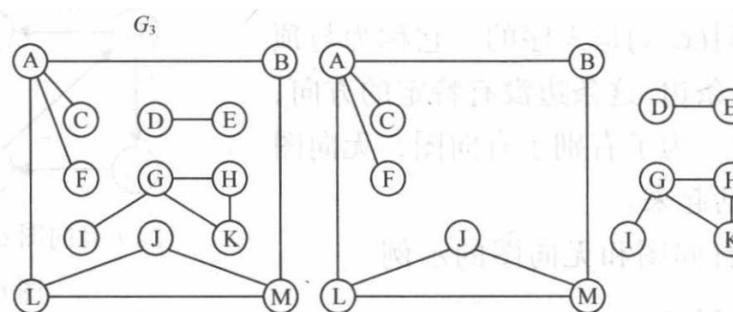
无向图 G 的极大连通子图称为 G 的连通分量

极大连通子图 : 若无向图 $G = (V, E)$ 的子图 $G_1 = (V_1, E_1)$ 为连通图

任取 $(v \in V) \wedge (v \notin V_1)$ 即 $v \in (V - V_1)$

若把 v 加入到 V_1 中, 如果 G_1 不再连通, 则称 G_1 为 G 的极大连通子图

如下图

(a) 无向图 G_3 (b) G_3 的 3 个连通分量

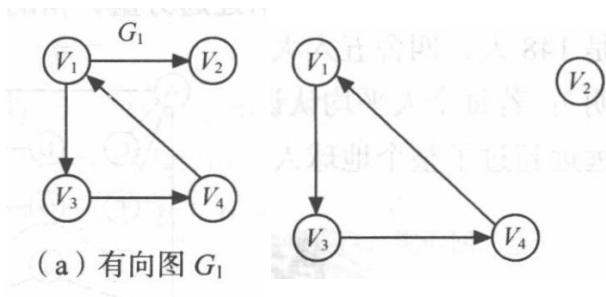
强连通分量：

有向图 G 的极大强连通子图称为 G 的连通分量

若有向图 $G = (V, E)$ 的子图 $G_1 = (V_1, E_1)$ 为连通图

任取 $(v \in V) \wedge (v \notin V_1)$ 即 $v \in (V - V_1)$

若把 v 加入到 V_1 中，如果 G_1 不再连通，则称 G_1 为 G 的极大强连通子图



- 极小连通子图和生成树(Spanning Tree)

若子图 G_1 是 G 的连通子图，在改子图中删除任意一条边， G_1 不再连通，则称 G_1 是 G 的极小连通子图

注意：极小连通子图中不存在loop，极大连通子图可以存在loop

生成树：

若无向图 $G = (V, E)$ ， V 中所有的点构成的极小连通子图就是 G 的生成树

• 图的存储结构

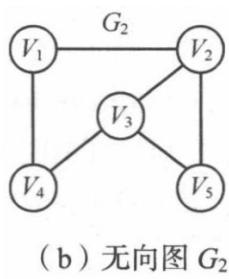
- 邻接矩阵(Adjacency Matrix)

若有图 $G = (V, E)$ ，有 n 个顶点，则对应 $n \times n$ 矩阵 A

$$A_{(v_1, v_2)} = \begin{cases} 1 & (v_1, v_2) \in E \\ 0 & (v_1, v_2) \notin E \end{cases}$$

注意：无向图的邻接矩阵为对称矩阵，而有向图的邻接矩阵未必

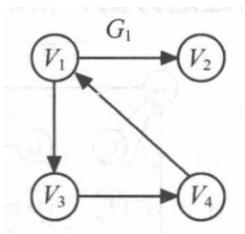
无向图



G_2 的邻接矩阵为：

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \text{degree}(v_i) = i\text{行中1的个数}$$

有向图



$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \begin{aligned} \text{degree}^+(v_i) &= \text{第}i\text{列1的个数} \\ \text{degree}^-(v_i) &= \text{第}i\text{行1的个数} \end{aligned}$$

带权图(网) weighted Graph

若有带权图 $W = (V, E)$, 有 n 个顶点, 则对应 $n \times n$ 矩阵 A

$$A_{(v_1, v_2)} = \begin{cases} \text{weight} & (v_1, v_2) \in E \\ \infty & (v_1, v_2) \notin E \end{cases}$$

代码实现

数据类型定义

```
#define MAXWEIGHT 99999 //最大权值
#define MAXVERTEX 20 //最大定点数
typedef char VetextType; //顶点用字符表示
typedef int MatrixType; //矩阵类型

typedef struct __AMGraph {
    char vertex[MAXVERTEX];
    MatrixType edge[MAXVERTEX][MAXVERTEX];
    int vertices, edges;
} AMGraph;
```

无向无权图

```
void Creat_unAMGraph_unweightd(AMGraph *G) {
    //初始化点和边个数
    printf("Please input the number of vertices:");
    scanf(" %d", &G->vertices);
    printf("Please input the number of edges:");
    scanf(" %d", &G->edges);
    //输入各个顶点的名字
```

```

printf("Please input the name of vertices(just like A B C):");
for (int i = 0; i < G->vertices; ++i) {
    scanf(" %c", &G->vertex[i]);
}
//把矩阵初始化
for (int i = 0; i < G->vertices; ++i)
    for (int j = 0; j < G->vertices; ++j)
        G->edge[i][j] = 0;

char v1, v2;           //一条边的顶点
int index_v1, index_v2; //边的顶点的下标
for (int i = 0; i < G->edges; ++i) {
    printf("(for %d)Please input the edge(just like A B):", i + 1);
    scanf(" %c %c", &v1, &v2);
    index_v1 = Locate_vertex(G, v1);
    index_v2 = Locate_vertex(G, v2);

    G->edge[index_v1][index_v2] = 1;
    G->edge[index_v2][index_v1] = 1;
}
}

```

测试代码

```

#include "Graph.h"

int main(void) {
    AMGraph G;
    Creat_unAMGraph_unweightd(&G);
    print_Matrix(&G);

    system("pause");
    return 0;
}

/*
5 6
A B C D E
A B
A D
B C
D C
C E
B E
-----
0 1 0 1 0
1 0 1 0 1
0 1 0 1 1
1 0 1 0 0
0 1 1 0 0
*/

```

无向带权图

```

void Creat_unAMGraph_weightd(AMGraph *G) {
    //初始化点和边个数
    printf("Please input the number of vertices:");
    scanf(" %d", &G->vertices);
    printf("Please input the number of edges:");
    scanf(" %d", &G->edges);
    //输入各个顶点的名字
    printf("Please input the name of vertices(just like A B C):");
    for (int i = 0; i < G->vertices; ++i) {
        scanf(" %c", &G->vertex[i]);
    }
    //把矩阵初始化
    for (int i = 0; i < G->vertices; ++i)
        for (int j = 0; j < G->vertices; ++j)
            G->edge[i][j] = MAXWEIGHT;

    char v1, v2;           //一条边的顶点
    int index_v1, index_v2; //边的顶点的下标
    int weight;
    for (int i = 0; i < G->edges; ++i) {
        printf("(for %d)Please input the edge(just like A B):", i + 1);
        scanf(" %c %c %d", &v1, &v2, &weight); //相比无向无权图只多了一个weight
        index_v1 = Locate_vertex(G, v1);
        index_v2 = Locate_vertex(G, v2);

        G->edge[index_v1][index_v2] = weight;
        G->edge[index_v2][index_v1] = weight;
    }
}

```

有向带权图

```

void Creat_AMGraph_weightd(AMGraph *G) {
    //初始化点和边个数
    printf("Please input the number of vertices:");
    scanf(" %d", &G->vertices);
    printf("Please input the number of edges:");
    scanf(" %d", &G->edges);
    //输入各个顶点的名字
    printf("Please input the name of vertices(just like A B C):");
    for (int i = 0; i < G->vertices; ++i) {
        scanf(" %c", &G->vertex[i]);
    }
    //把矩阵初始化
    for (int i = 0; i < G->vertices; ++i)
        for (int j = 0; j < G->vertices; ++j)
            G->edge[i][j] = MAXWEIGHT;

    char v1, v2;           //一条边的顶点
    int index_v1, index_v2; //边的顶点的下标
    int weight;
    for (int i = 0; i < G->edges; ++i) {

```

```

printf("(for %d)Please input the edge(just like A B):", i + 1);
scanf(" %c %c %d", &v1, &v2, &weight);
index_v1 = Locate_vertex(G, v1);
index_v2 = Locate_vertex(G, v2);

G->edge[index_v1][index_v2] = weight;
}
}

```

有向无权图

```

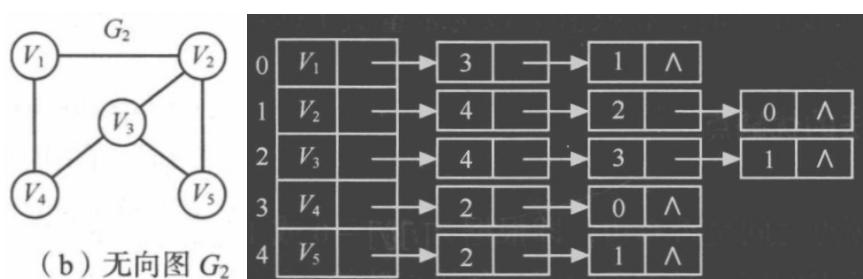
void Creat_AMGraph_unweightd(AMGraph *G) {
//初始化点和边个数
printf("Please input the number of vertices:");
scanf(" %d", &G->vertices);
printf("Please input the number of edges:");
scanf(" %d", &G->edges);
//输入各个顶点的名字
printf("Please input the name of vertices(just like A B C):");
for (int i = 0; i < G->vertices; ++i) {
    scanf(" %c", &G->vertex[i]);
}
//把矩阵初始化
for (int i = 0; i < G->vertices; ++i)
    for (int j = 0; j < G->vertices; ++j)
        G->edge[i][j] = 0;

char v1, v2;           //一条边的顶点
int index_v1, index_v2; //边的顶点的下标
int weight;
for (int i = 0; i < G->edges; ++i) {
    printf("(for %d)Please input the edge(just like A B):", i + 1);
    scanf(" %c %c %d", &v1, &v2, &weight);
    index_v1 = Locate_vertex(G, v1);
    index_v2 = Locate_vertex(G, v2);

    G->edge[index_v1][index_v2] = 1; //有向图不是对称矩阵
}
}

```

- 邻接表(Adjacency List)



数据类型定义

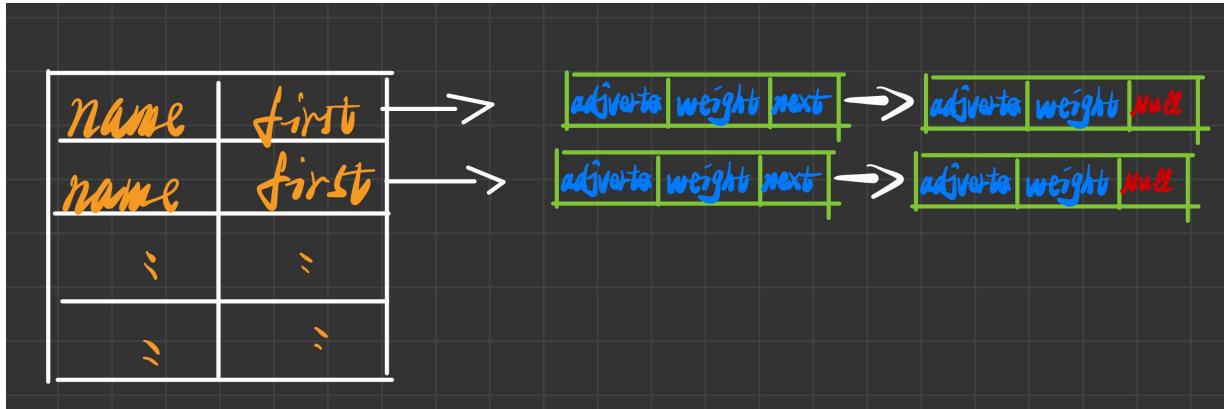
```

typedef struct __EdgeNode {
    int adjvertex;
    __EdgeNode *next;
    int weight;
} EdgeNode;

typedef struct __ALGNode {
    VertexType name;
    EdgeNode *first;
} ALGNode;

typedef struct __ALGraph {
    ALGNode vertex[MAXVERTEX];
    int edges, vertices;
} ALGraph;
//如下图

```



代码实现

无向无权图

无向有权，有向无权，有向有权图的创建方法和此方法类似，只需改动weight和新建节点个数即可

```

void Creat_unALGraph_unweighted(ALGraph *G) {
    //初始化点和边个数
    printf("Please input the number of vertices:");
    scanf(" %d", &G->vertices);
    printf("Please input the number of edges:");
    scanf(" %d", &G->edges);
    //输入各个顶点的名字
    printf("Please input the name of vertices(just like A B C):");
    for (int i = 0; i < G->vertices; ++i) {
        scanf(" %c", &G->vertex[i].name);
        G->vertex[i].first = NULL;
    }

    char v1, v2;           //一条边的顶点
    int index_v1, index_v2; //边的顶点的下标

    for (int i = 0; i < G->edges; ++i) {
        printf("(for %d)Please input the edge(just like A B):", i + 1);
        scanf(" %c %c", &v1, &v2);
        index_v1 = Locate_vertex(G, v1);
        index_v2 = Locate_vertex(G, v2);
    }
}

```

```

//创造新节点1
EdgeNode *pnew1 = (EdgeNode *)malloc(sizeof(EdgeNode));
pnew1->adjvertex = index_v2;
//头插法
pnew1->next = G->vertex[index_v1].first;
G->vertex[index_v1].first = pnew1;
//创造新节点2
EdgeNode *pnew2 = (EdgeNode *)malloc(sizeof(EdgeNode));
pnew2->adjvertex = index_v1;
//头插法
pnew2->next = G->vertex[index_v2].first;
G->vertex[index_v2].first = pnew2;
}
}

```

测试代码

```

#include "ALGraph.h"

int main(void) {
    ALGraph G;
    Creat_unALGraph_unweighted(&G);
    print_ALG_unweighted(&G);

    system("pause");
    return 0;
}

/*
*/
/*
5 6
A B C D E
A B
A D
B C
D C
C E
B E
-----
A:A--D  A--B
B:B--E  B--C  B--A
C:C--E  C--D  C--B
D:D--C  D--A
E:E--B  E--C
*/

```

- 邻接表和邻接矩阵的比较

邻接矩阵

- 优点
 - 便于判断顶点间是否有边
 - 便于计算各个顶点的度

- 缺点

- 不便于插入和删除顶点
- 不便于统计边数，需要扫描矩阵才能计算 $O(n^2)$
- 空间复杂度较高，但如果 n 较大时，可以采用上三角或下三角矩阵(因为矩阵是对称的)

邻接表

- 优点

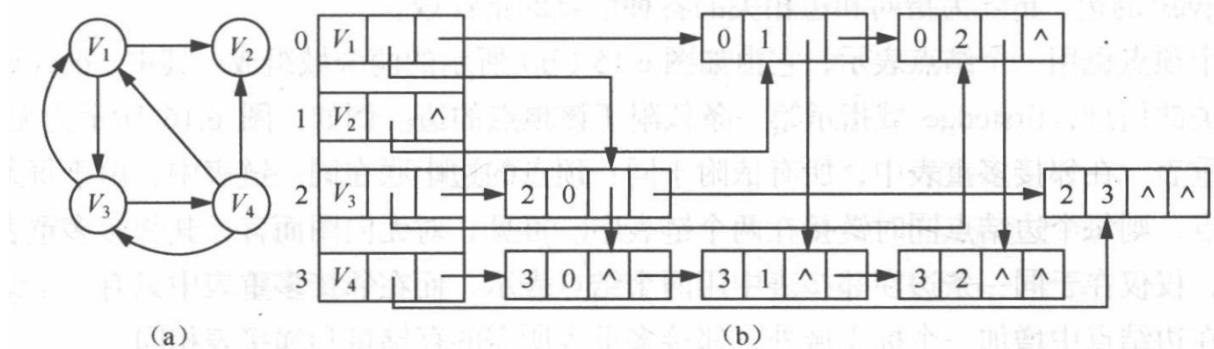
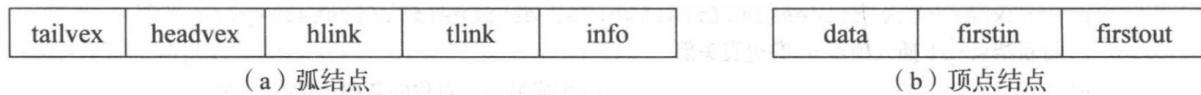
- 便于增加和删除顶点
- 便于统计边的数量，按顶点顺序扫描所有边即可。 $O(n + e)$
- 空间效率高，无向图 $O(n + 2e)$ ，有向图 $O(n + e)$

- 缺点

- 不便于判断两顶点间是否有边(相对于矩阵的随机取值而言)
- 不便于计算各个顶点的度。
- 对于无向图， $degree(v_i) =$ 第*i*个表的节点个数
- 对无向图
 $degree(v_i) = degree^+(v_i) + degree^-(v_i)$ ，入度为第*i*个表的节点个数，但是出度却要历遍所有的表

- 十字链表

十字链表可以解决用邻接链表储存的有向图求顶点degree的问题



根据上图可知，一条边即是一个顶点的入度，也是另外一个顶点的出度

设 e 是 (v_1, v_2) 的一条边

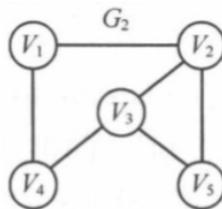
当建立边节点 e 时，使 v_1 的 $firstout$ 域指向 e ，使 v_2 的 $firstin$ 指向 e

当 v_1 再次有出度边 e_2 时，使 e_2 的 $hlink$ 域指向 v_1 的 $firstout$ 所指节点， $firstout$ 指向 e_2

当 v_2 再次有入度边 e_3 时，使 e_3 的 $tlink$ 域指向 v_2 的 $firstin$ 所指节点， $firstin$ 指向 e_3

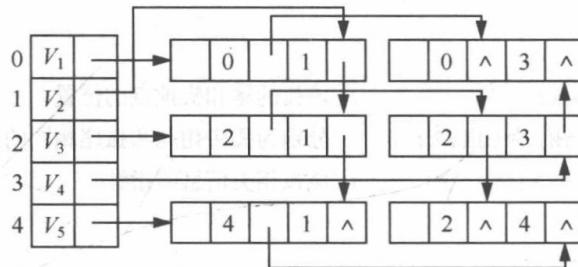
- 邻接多重链表

用于解决用邻接表存储的无向图每条边都要存储两遍的问题

(b) 无向图 G_2

mark	ivex	ilink	jvex	jlink	info	data	firistedge
(a) 边结点						(b) 顶点结点	

图 6.15 边结点和顶点结点

图 6.16 无向图 G_2 的邻接多重表

如上图

*mark*记录该边是否被搜索过*ivex*, *jvex*分别表示边 $e = (v_i, v_j)$ 顶点的下标*ilink*, *jlink*分别表示 v_i 的下一条边节点, v_j 的下一条边节点令 $e = (v_i, v_j)$ 使 e 的*ilink*指向 v_i 的*first*域, e 的*jlink*指向 v_j 的*first*域 e 的*ivex* = v_i 的下标, e 的*jvex* = v_j 的下标使 v_i 和 v_j 的*first*指向 e

遍历

令 v_i 的*first*为 p while($p \neq \text{NULL}$)打印 p $p = p - > \text{ilink}$

• 图的遍历

图中可能存在loop, 且图的任何一点都有可能和其他顶点相连

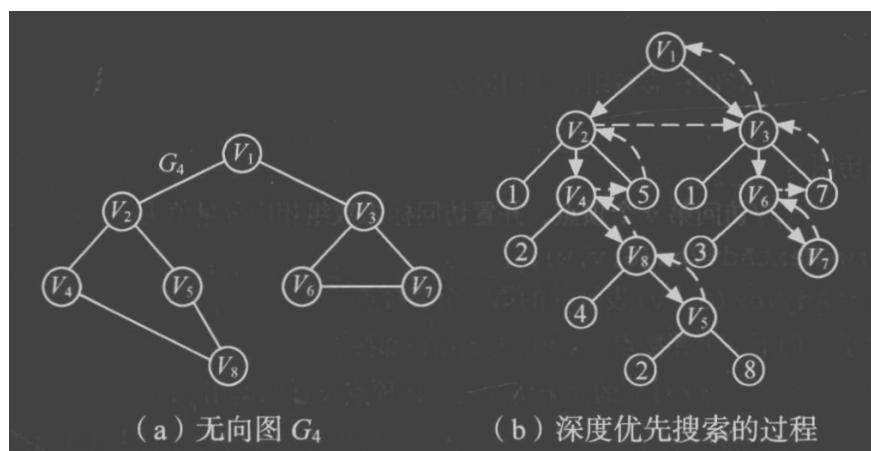
在访问完某个顶点之后, 可能会沿着某些边又回到了曾经访问过的顶点

解决思路:

设置辅助数组 $vist[n]$, 用来标记顶点是否被访问过

$$\begin{cases} vist[i] = 0 & i\text{顶点未被访问过} \\ vist[i] = 1 & i\text{顶点被访问过} \end{cases}$$

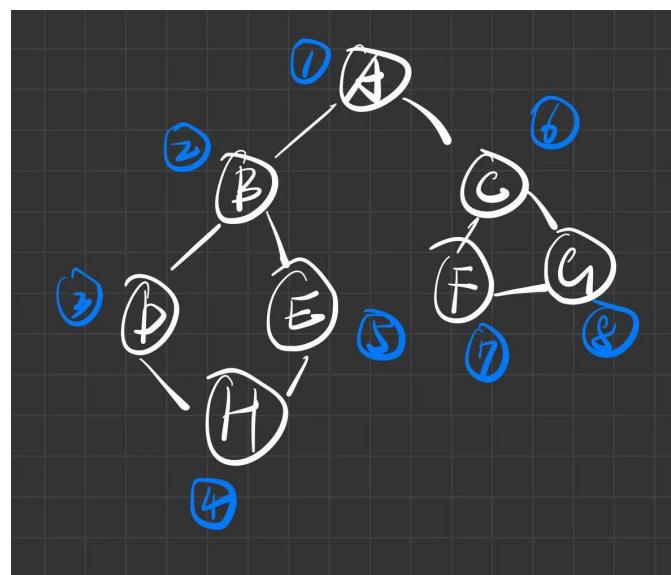
- 深度优先 (Depth First Search)



算法描述：

先访问 v_1 , 再访问 v_1 的邻接点 v_2
再访问 v_2 的邻接点 v_4, \dots, v_8, v_5
开始出栈, 控制节点再次来到 v_1
因为 v_2 已经被访问过, 所以开始访问 v_3, v_6, v_7
全部出栈完, 控制再次回到 v_1
 v_1 出栈, 结束

代码实现下图



注意上图中的当访问完 E 之后, 并不是直接回到 A
而是退回到 H, D, B , 需要依次出栈

遍历矩阵

```
void DFS_AM(AMGraph *G, int v, bool *visit) {
    printf("%c ", G->vertex[v]); //先遍历顶点
    visit[v] = true; //访问标记

    for (int k = 0; k < G->vertices; ++k) {
        if (G->edge[v][k] && !visit[k]) //如果v, k之间存在边, 并且k顶点并未被访问过
            DFS_AM(G, k, visit);
    }
}
```

```

}

void DFS_AMGraph(AMGraph *G, VertexType v) {           //封装函数
    bool *visit = (bool *)malloc(sizeof(bool) * G->vertices); //为visit分配空间
    memset(visit, false, sizeof(bool) * G->vertices);        //初始化为false

    int index = Locate_vertex(G, v); //找到下标
    DFS_AM(G, index, visit);       //以此下标为顶点出发，遍历
    free(visit);
}

```

测试代码

```

#include "AMGraph.h"

int main(void) {
    AMGraph G;
    Creat_unAMGraph_unweightd(&G);
    DFS_AMGraph(&G, 'A');

    system("pause");
    return 0;
}

/*
8 9
A B C D E F G H
A B
A C
B D
D H
B E
E H
C F
C G
F G
-----
A B D H E C F G
*/

```

遍历邻接表

注意:使用邻接表时，遍历的顺序和邻接矩阵不一样，因为创建邻接表使用头插法(如果使用尾插法顺序则一样)

```

void DFS_AL(ALGraph *G, int v, bool *visit) {
    printf("%c ", G->vertex[v].name);
    visit[v] = true;

    EdgeNode *p = G->vertex[v].first;
    while (p) {
        int adj = p->adjvertex;
        if (!visit[adj]) // 只需判断visit数组，因为当p不为空时，adjvertex必然存在

```

```

DFS_AL(G, adj, visit);

    p = p->next;
}
}

void DFS_ALGraph(ALGraph *G, VertexType v) { //封装代码
    bool *visit = (bool *)malloc(sizeof(bool) * G->vertices);
    memset(visit, false, sizeof(bool) * G->vertices);

    int index = Locate_vertex(G, v);
    DFS_AL(G, index, visit);
}

```

测试代码

```

#include "ALGraph.h"

int main(void) {
    ALGraph G;
    Creat_unALGraph_unweighted(&G);
    DFS_ALGraph(&G, 'A');

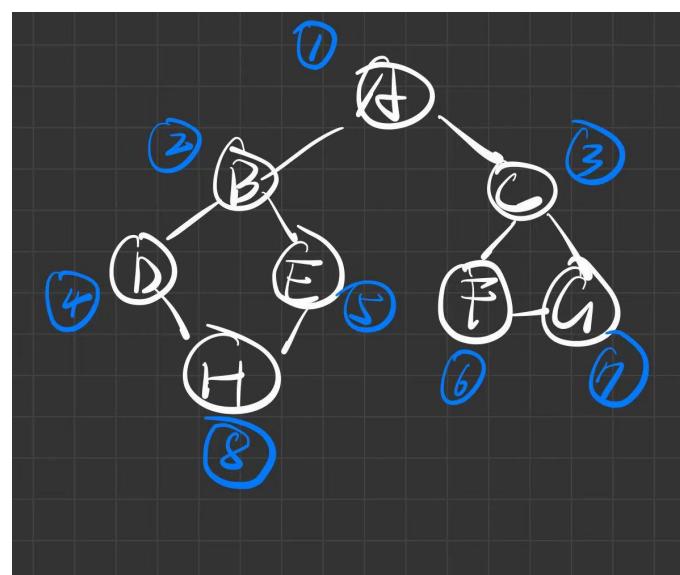
    system("pause");
    return 0;
}
// 
/*
8 9
A B C D E F G H
A B
A C
B D
D H
B E
E H
C F
C G
F G
-----
A C G F B E H D
*/

```

- 广度优先 (Breadth First Search)

从图的 v_1 出发，首先访问 v_1 ，然后访问 v_1 的所有邻接点， $v_{i1}, v_{i2}, \dots, v_{in}$
 然后按照 $v_{i1}, v_{i2}, \dots, v_{in}$ 的顺序，依次访问他们的邻接点

实现遍历下图



代码实现

遍历矩阵

```

void BFS_AM(AMGraph *G, int v, bool *visit) {
    SqQueue Q;
    InitQueue(&Q);
    EntryQ(&Q, v); //先让顶点入队
    visit[v] = true; //入队时设置visit状态
    int pop; //用于接收队列弹出数据
    while (!IsEmpty(&Q)) {
        OutQ(&Q, &pop);
        printf("%c ", G->vertex[pop]); //出队时，打印

        for (int k = 0; k < G->vertices; ++k) {
            if (G->edge[pop][k] && !visit[k]) {
                EntryQ(&Q, k);
                visit[k] = true;
                //! 设置visit状态，不可以放在printf后面，因为for循环可能造成重复入队
            }
        }
    }
}

void BFS_AMGraph(AMGraph *G, VertexType v) {
    bool *visit = (bool *)malloc(sizeof(bool) * G->vertices); //为visit分配空间
    memset(visit, false, sizeof(bool) * G->vertices); //初始化为false

    int index = Locate_vertex(G, v); //找到下标
    BFS_AM(G, index, visit); //以此下标为顶点出发，遍历
    free(visit);
}

```

测试代码

```

#include "AMGraph.h"

int main(void) {
    AMGraph G;

```

```

Creat_unAMGraph_unweightd(&G);
BFS_AMGraph(&G, 'A');

system("pause");
return 0;
}

/*
8 9
A B C D E F G H
A B
A C
B D
D H
B E
E H
C F
C G
F G
-----
A B C D E F G H
*/

```

遍历邻接表

注意:使用邻接表时, 遍历的顺序和邻接矩阵不一样, 因为创建邻接表使用头插法(如果使用尾插法顺序则一样)

```

void BFS_AL(ALGraph *G, int v, bool *visit) {
    SqQueue Q;
    InitQueue(&Q);
    EdgeNode *p; //用于遍历邻接表
    EntryQ(&Q, v); //入队顶点
    visit[v] = true; //设置顶点visit数组状态
    int pop; //用于接收队列弹出数据
    while (!IsEmpty(&Q)) {
        OutQ(&Q, &pop);
        printf("%c ", G->vertex[pop].name); //出队后打印
        p = G->vertex[pop].first; //利用p遍历

        while (p) {
            if (!visit[p->adjvertex]) { //判断p遍历中的邻接点是否被遍历过
                EntryQ(&Q, p->adjvertex); //如果没有被遍历过, 入队
                visit[p->adjvertex] = true; //同时设置visit状态
            }
            p = p->next;
        }
    }
}

void BFS_ALGraph(ALGraph *G, VertexType v) {
    bool *visit = (bool *)malloc(sizeof(bool) * G->vertices); //为visit分配空间
    memset(visit, false, sizeof(bool) * G->vertices); //初始化为false

    int index = Locate_vertex(G, v); //找到下标

```

```
BFS_AL(G, index, visit);           //以此下标为顶点出发, 遍历
free(visit);
}
```

测试代码

```
#include "ALGraph.h"
```

```
int main(void) {
    ALGraph G;
    Creat_unALGraph_unweighted(&G);
    BFS_ALGraph(&G, 'A');

    system("pause");
    return 0;
}
/*
8 9
A B C D E F G H
A B
A C
B D
D H
B E
E H
C F
C G
F G
-----
A C B G F E D H
*/
```

- 算法效率

可知, 邻接矩阵的时间效率为 $O(n^2)$, 邻接表的时间效率为 $O(n + e)$

• 最小生成树 Spanning Tree

- Prim 算法

令 $G = \{V, E\}$ 为连通图带权，且 $U = \emptyset$

设起点为 v_0 , 把 v_0 加入到 U 中

选取 $v_i \in (V - U) \wedge Weight(v_0, v_i)$ 为最小

输出 (v_0, v_i) 边

把 v_i 加入 U 中

选取 $v_j \in (V - U) \wedge Weight(v_i, v_j)$ 为最小

输出 (v_i, v_j) 边

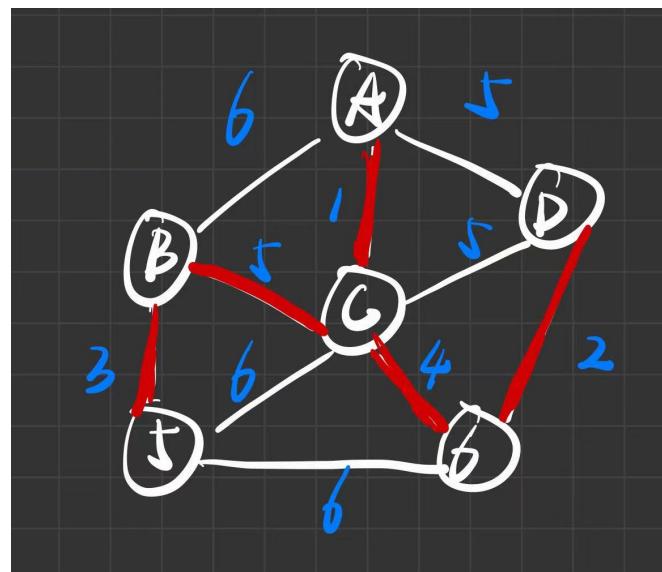
...

直到存在 $n - 1$ 条边

此时输出的边即为 *Spanning Tree*

代码实现

实现下图的Spanning Tree



需要辅助集合U

```
struct Uset {
    int adjvertex; //下标为i的点的邻接点
    int weight;    //当前权值
};
```

邻接矩阵

前置算法

```

int Min_Uset(Uset *U, int n) {
    int min = INT_MAX;
    int pos = 0;

    for (int i = 0; i < n; ++i) {
        if (U[i].weight != 0 && U[i].weight < min) {
            //! 核心语句 U[i].weight != 0 说明不在U中, 即V-U
            min = U[i].weight;
            pos = i;
        }
    }
    return pos;
}

```

```

void MST_Prim(AMGraph *G, VertexType v) {
    int u = Locate_vertex(G, v);
    Uset *U = (Uset *)malloc(sizeof(Uset) * G->vertices);
    //动态为U分配空间, 大小为n(顶点个数)
    for (int i = 0; i < G->vertices; ++i) {
        //初始化U集合, 因为先把顶点v放入U集合中, 所以U[i].adjvertex为
        //顶点v的下标, 即u。如果i与点v存在边
        U[i].adjvertex = u;
        //如果i与点v存在边, 赋值, 如果不存在weight为99999
        U[i].weight = G->edge[u][i];
    }
    U[u].weight = 0; // weight = 0 说明此点已经加入U集合

    for (int i = 1; i < G->vertices; ++i) { //选取n-1条边
        int min = Min_Uset(U, G->vertices); //在V-U中选取权值最小的边
        int u_0 = U[min].adjvertex; // u_0为最小边的邻接点
        printf("%c->%c ", G->vertex[u_0], G->vertex[min]); //输出此两点
        U[min].weight = 0; //把该边的顶点加入集合U

        for (int j = 0; j < G->vertices; ++j) { //更新U集合
            if (G->edge[min][j] < U[j].weight) {
                //如果成立, 使U的weight, 和adjvertex和min相关
                U[j].weight = G->edge[min][j];
                U[j].adjvertex = min;
            }
        }
    }

    free(U);
}

```

邻接表

```

void MST_Prime(ALGraph *G, VertexType v) {
    int u = Locate_vertex(G, v);
    Uset *U = (Uset *)malloc(sizeof(Uset) * G->vertices);

```

```

for (int i = 0; i < G->vertices; ++i) {
    //邻接表的weight不会为最大值，所以初始化为最大值
    U[i].weight = MAXWEIGHT;
    U[i].adjvertex = u;
}
U[u].weight = 0; //加入U集合
EdgeNode *p = G->vertex[u].first;
while (p) {
    //遍历点v的邻接表，把weight放入U中
    U[p->adjvertex].weight = p->weight;
    p = p->next;
}

for (int i = 1; i < G->vertices; ++i) {
    // n-1个边
    int min = Min_Uset(U, G->vertices); // 权值最小边
    int u_0 = U[min].adjvertex;
    printf("%c->%c ", G->vertex[u_0].name, G->vertex[min].name);
    U[min].weight = 0; //加入集合U中

    EdgeNode *p_min = G->vertex[min].first;
    while (p_min) {
        //因为邻接表的特性，不需要遍历所有顶点，只需遍历min的邻接表即可
        if (p_min->weight < U[p_min->adjvertex].weight) {
            U[p_min->adjvertex].adjvertex = min;
            U[p_min->adjvertex].weight = p_min->weight;
        }
        p_min = p_min->next;
    }
}

free(U);
}

```

测试代码

```

#include "ALGraph.h"

int main(void) {
    ALGraph G;
    Creat_unALGraph_weighted(&G);
    MST_Prime(&G, 'A');

    system("pause");
    return 0;
}
/*
6 10
A B C D E F
A B 6
A D 5
A C 1
B C 5
C D 5
B E 3
E C 6

```

```

C F 4
D F 2
E F 6
-----
A->C C->F F->D C->B B->E
*/

```

- Kruskal 算法

定理1：

若无向有权无环图 $G = \{V, E\}$ 为 A 图的最大连通子图

$\forall v_i \in V$, 作为连通依赖点, 则 V 集合所有点必然存在一条通往 v_i 的路径
单个顶点的连通依赖点为自身(注意： (v, v) 表示为 *loop*, 不可以这么表示)

Proof :

令 v_i 为连通依赖点, 且 $\exists v_k \in V$ 使 v_k, v_i 之间并不存在路径

Contradiction by G 为连通图

引理：

若 $G = \{V_G, E_G\}, H = \{V_H, E_H\}$ 分别为 A 的最大连通子图

若 $\forall v_i \in V_G, \forall v_j \in V_H$, 则 $v_i \neq v_j$

Proof :

利用 *Contradiction*

若 $v_i = v_j$, 则说明 V_G 中所有的点都有一条通往 v_j 的路径

Contradiction by G 为最大连通子图(子图 H 同理)

Kruskal 算法：

若存在无向带权图 $G = \{V, E\}$ 且 G 连通

令集合 $MST = \emptyset$

对 E 进行排序, 取 $\min(E) = e_1$, 把 e_1 加入 MST 中

循环对 e_2, e_3, \dots, e_n 进行如下操作

如果 e_i 的连通依赖点 $\neq \forall e_j \in MST$ 的联通点

则把 e_i 加入到 MST

直到遍历所有边

MST 集合中为 G 的 *Minimal Spanning Tree*

代码实现

数据类型定义

```

struct Eset { //边集合
    int start; //起点
    int end;   //终点
    int weight;
};

void Sort_Eset(Eset *E, int length); //排序函数

```

采用冒泡排序，可以灵活变换

```
void Sort_Eset(Eset *E, int length) {
    bool flag = true; //排序flag
    for (int i = 0; i < length - 1 && flag; ++i) { //如果未发生交换则说明有序
        flag = false; //第一次设置为false
        for (int j = 0; j < length - 1 - i; ++j) {
            if (E[j].weight > E[j + 1].weight) {
                flag = true; //如果发生交换，为true
                Eset temp = E[j];
                E[j] = E[j + 1];
                E[j + 1] = temp;
            }
        }
    }
}
```

邻接矩阵

```
void InitEset(Eset *E, AMGraph *G) {
    //遍历上三角矩阵
    Eset *p = E;
    for (int i = 0; i < G->vertices; ++i) // i为array
        for (int k = i + 1; k < G->vertices; ++k) { // k为column
            if (G->edge[i][k] < MAXWEIGHT) { //小于说明存在
                (p)->start = i;
                (p)->end = k;
                (p)->weight = G->edge[i][k];
                ++p;
            }
        }
    p = NULL;
}
```

```
void MST_Krusal(AMGraph *G) {
    Eset *E = (Eset *)malloc(sizeof(Eset) * G->edges);
    InitEset(E, G);
    Sort_Eset(E, G->edges);
    //此时E集合有序
    //建立V集合存放连通依赖点
    int V[G->vertices]; //例如V[i] = k;
    //表示顶点i依赖于k点(即必有一条到k的路径)
    for (int i = 0; i < G->vertices; ++i) //清掉所有边
        V[i] = i; //现在V集合的连通分量就是自己，即所有顶点无邻接点

    for (int j = 0; j < G->edges; ++j) {
        int v_1 = E[j].start; // v_1和v_2分别为这个边的顶点
        int v_2 = E[j].end;

        int component_v_1 = V[v_1]; // v_1所属的连通分量
        int component_v_2 = V[v_2]; // v_2所属的连通分量
        if (component_v_1 != component_v_2) { //说明不会形成loop
            if (component_v_1 < component_v_2) {
                V[component_v_2] = component_v_1;
            } else {
                V[component_v_1] = component_v_2;
            }
        }
    }
}
```

```

printf("%c->%c ", G->vertex[v_1], G->vertex[v_2]);
//链接这两个顶点，并且输出
//因为链接了两个顶点，那么此时构成一条新的连通分量，所以要更新连通分量依赖点
for (int k = 0; k < G->vertices; ++k) {
    if (V[k] == component_v_2) //如果k顶点依赖于v_2的连通分量
        //那么在连接之后，所有的顶点都依赖于v_1的连通分量
        V[k] = component_v_1;
}
}

free(E);
}

```

测试代码

```

#include "AMGraph.h"

int main(void) {
    AMGraph G;
    Creat_unAMGraph_weightd(&G);

    MST_Krusal(&G);
    system("pause");
    return 0;
}

/*
/*
6 10
A B C D E F
A B 6
A D 5
A C 1
B C 5
C D 5
B E 3
E C 6
C F 4
D F 2
E F 6
-----
A->C D->F B->E C->F B->C
*/

```

邻接表

```

void InitEset(Eset *E, ALGraph *G) {
    //! 邻接表和邻接矩阵不一样，矩阵是连续的，而邻接表在存储上不是连续的
    //! ALGraph需要为有向图
    Eset *p_E = E;

    for (int i = 0; i < G->vertices; ++i) {
        EdgeNode *p = G->vertex[i].first;

```

```

while (p) {
    p_E->start = i;
    p_E->end = p->adjvertex;
    p_E->weight = p->weight;

    p = p->next;

    ++p_E;
}
}
p_E = NULL;
}

```

```

void MST_Krusal(ALGraph *G) {
    Eset *E = (Eset *)malloc(sizeof(Eset) * G->edges);
    InitEset(E, G);
    Sort_Eset(E, G->edges);

    for (int w = 0; w < G->edges; ++w) {
        printf("(%d)start=%d, end=%d, weight=%d\n", w, E[w].start, E[w].end,
               E[w].weight);
    }
    int V[G->vertices];
    for (int i = 0; i < G->vertices; ++i)
        V[i] = i;

    for (int j = 0; j < G->edges; ++j) {
        int v_1 = E[j].start;
        int v_2 = E[j].end;

        int component_v_1 = V[v_1];
        int component_v_2 = V[v_2];

        if (component_v_1 != component_v_2) {
            printf("%c->%c ", G->vertex[v_1].name, G->vertex[v_2].name);

            for (int k = 0; k < G->vertices; ++k) {
                if (component_v_2 == V[k])
                    V[k] = component_v_1;
            }
        }
    }

    free(E);
}

```

测试代码

```

#include "ALGraph.h"

int main(void) {
    ALGraph G;
    Creat_ALGraph_weighted(&G); //! 需要用有向图来表示无向图
}

```

```

MST_Krusal(&G);

system("pause");
return 0;
}

/*
6 10
A B C D E F
A B 6
A D 5
A C 1
B C 5
C D 5
B E 3
E C 6
C F 4
D F 2
E F 6
-----
A->C D->F B->E C->F B->C
*/

```

• 最短路径

- Dijkstra 算法

若有无向带权网 $G = \{V, E\}$

以 $v_0 \in V$ 为顶点, 求出所有 $v_i \in V$ 和 v_0 间的最路径

令 $D(v_i)$ 为 v_i 到 v_0 间的路径长度, 集合 $S = \emptyset$

若 (v_0, v_k) 存在, 则初始化 $D(v_k)$, 若不存在则 $D(v_k) = \infty$

遍历所有点之后, 把 v_0 加入到 S 中

$$D(v_n) = \min \left(D(v_{k1}), D(v_{k2}), \dots, D(v_{kn}) \right) \text{ with } v_{ki} \notin S$$

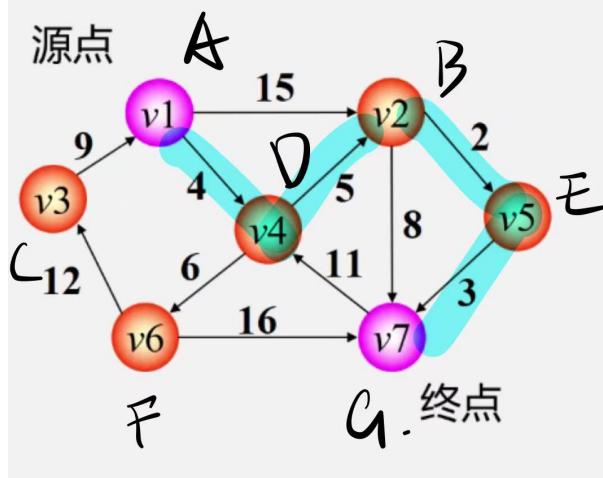
把 v_n 加入到 S 中, 若 v_z 为 v_n 的邻接点, 且 $D(v_n) + weight(v_n, v_z) < D(v_z)$

则 $D(v_z) = D(v_n) + weight(v_n, v_z)$

重复上步骤, 直到 $S = V$

用代码实现此图

第一类问题：两点间最短路径



代码实现

邻接矩阵

```

void ShortestPath(AMGraph *G, int v, int *path) {
    //! 数组path[i]表示i的最短前驱
    int D[G->vertices]; //集合D表示各个顶点到v的距离
    bool S[G->vertices]; //集合S, 如果i点在S中则为true

    for (int i = 0; i < G->vertices; ++i) {
        //初始化D,S,path
        D[i] = G->edge[v][i]; //如果不存在边, 则为MAXWEIGHT
        S[i] = false;           //初始化S为空集

        if (D[i] < MAXWEIGHT) //说明边存在
            path[i] = v;       //设置前驱
        else
            path[i] = -1;     //如果不存在边, -1
    }

    S[v] = true;
    D[v] = 0;
    //初始化完成

    for (int i = 1; i < G->vertices; ++i) {
        int min_index;
        int min = MAXWEIGHT;
        for (int j = 0; j < G->vertices; ++j) {
            //从D中选出路径最短点, 且不在集合S中
            if (!S[j] && D[j] < min) {
                min_index = j;
                min = D[j]; //求出D[j]最小的点
            }
        }
        S[min_index] = true; //把该点加入到S中

        for (int k = 0; k < G->vertices; ++k) {
            if (!S[k] && (G->edge[min_index][k] + D[min_index] < D[k])) {
                //遍历min_index的邻接点, 进行判断
                D[k] = G->edge[min_index][k] + D[min_index];
            }
        }
    }
}

```

```

        path[k] = min_index;
    }
}
}

void ShortestPath_Dijkstra(AMGraph *G, VertexType v) {
    //封装的函数，表示以v为起点，求出所有点与v的最短路径
    int v_index = Locate_vertex(G, v);
    int *path = (int *)malloc(sizeof(int) * G->vertices);
    ShortestPath(G, v_index, path);
    // for (int i = 0; i < G->vertices; ++i)

    for (int i = 0; i < G->vertices; ++i) {
        if (i != v_index) {
            print_path(G, path, v_index, i);
            printf("\n");
        }
    }

    free(path);
}

void print_path(AMGraph *G, int *path, int start, int end) {
    SqStack S;
    InitStack(&S);
    printf("%c->", G->vertex[start]);
    Push(&S, end); //先入栈end
    int flag = end; //用于判断是否输出"->""
    end = path[end];
    while (end != start) {
        Push(&S, end);
        end = path[end];
    }

    int head = start;
    int pop;
    while (!IsEmpty(&S)) {

        Pop(&S, &pop);
        printf("%c(%d)", G->vertex[pop], G->edge[head][pop]);
        if (pop != flag)
            printf("->");
        head = pop;
    }
}
}

```

测试代码

```

#include "AMGraph.h"

int main(void) {
    AMGraph G;
    Creat_unAMGraph_weightd(&G);
    ShortestPath_Dijkstra(&G, 'A');
}

```

```

    system("pause");
    return 0;
}
/*
*/
7 11
A B C D E F G
A B 15
A C 9
A D 4
C F 12
F G 16
F D 6
D B 5
B G 8
B E 2
E G 3
D G 11
-----
A->D(4)->B(5)
A->C(9)
A->D(4)
A->D(4)->B(5)->E(2)
A->D(4)->F(6)
A->D(4)->B(5)->E(2)->G(3)
*/

```

- Floyd算法

本质上为暴力算法

令矩阵 $D = \text{存储图 } G = \{V, E\}$ 的权值，其中 G 有 n 个顶点

$D[i][j]$ 表示 i 到 j 的 weight, $D[x][y]$ 表示 x 到 y 的 weight

遍历矩阵所有元素，向 (i, j) 之间插入点 $k \in V$, 如果 $\text{weight}(i, k) + \text{weight}(k, j) < \text{weight}(i, j)$

则 $\text{weight}(i, j) = \text{weight}(i, k) + \text{weight}(k, j)$

遍历结束后，矩阵 D 记录 (x, y) 的最短路径

代码实现

邻接矩阵

```

int **Path_Matrix(AMGraph *G) {
    // path[i][j] 表示从 i 到 j 的路径上, j 点的直接前驱
    int n = G->vertices;
    int **path = (int **)malloc(sizeof(int *) * n); // 先分配一维指针数组
    for (int i = 0; i < G->vertices; ++i)           // path[i] 分配空间;
        path[i] = (int *)malloc(sizeof(int) * n);
    // 此时 path 为二维数组

```

```

int D[n][n];

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
        D[i][j] = G->edge[i][j]; // copy给D
        if (D[i][j] < MAXWEIGHT) //点i,j如果为邻接点
            path[i][j] = i; //此时的路径即为i, j之间的边, 所以j的前驱为i
        else
            path[i][j] = -1;
    }

for (int k = 0; k < n; ++k) //尝试在两点之间加入点k
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            if (D[i][k] + D[k][j] < D[i][j]) {
                D[i][j] = D[i][k] + D[k][j];
                path[i][j] = path[k][j]; //更新path, 即j的前驱, 向前递归
            }
        }
}

return path;
}

```

```

void ShortestPath_Floyd(AMGraph *G, VertexType v1, VertexType v2) {
    int start = Locate_vertex(G, v1);
    int end = Locate_vertex(G, v2);

    int **path = Path_Matrix(G);

    SqStack S;
    InitStack(&S);
    Push(&S, end); //先入栈末端点
    int prior = path[start][end];

    while (prior != start) {
        //向前回溯, 直到prior == start
        Push(&S, prior);
        prior = path[start][prior];
    }

    int head = start; // head用于向前追溯
    int pop;
    printf("%c->", G->vertex[start]);
    while (!IsEmpty(&S)) {
        Pop(&S, &pop);
        printf("%c(%d)", G->vertex[pop], G->edge[head][pop]);
        if (pop != end)
            printf("->");
        head = pop; //向前追溯
    }

    for (int i = 0; i < G->vertices; ++i)
        //因为path本质上是一维指针数组, 每个指针又指向一块空间, 所以逐个释放
        free(path[i]);
    free(path); //还需要释放path
}

```

```
path = NULL;
}
```

• 有向无环图(Directed Acycline Graph)

- AOC和AOE

AOC网

用一个有向图表示一个工程的各个子工程及其相互制约关系
其中顶点表示活动，弧表示优先制约关系

Activity on vertex

AOE网

弧表示活动，以顶点表示活动开始或结束事件
Activity on edge

- 拓扑排序

在AOV网没有回路的前提下，将全部活动排列成一个线性序列

AOV网中边 (i, j) 存在，则在这个序列中 i 一定排在 j 的前面

对AOV网进行如上排序，即为拓扑排序

算法思路：

若AOV图中 $\text{indegree}(v_i) = 0$ ，则去除掉所有和 v_i 相关的边

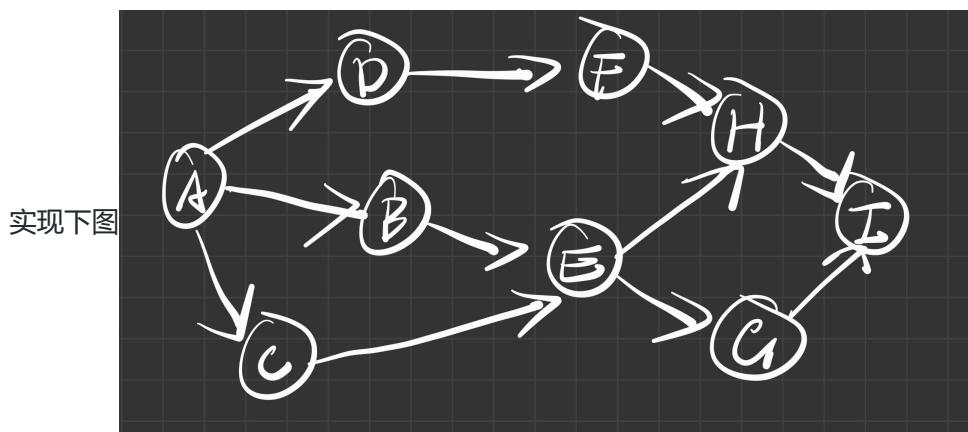
令 $T = \emptyset$ ，把 v_i 加入到 T 中

更新所有 $v \in V$ 的 indegree

直到 $T = V$

此时 T 的序列则为拓扑序列

代码实现



邻接表

前置函数

```
void InDegree(ALGraph *G, int *a) {
    //获取所有点的indegree
    for (int i = 0; i < G->vertices; ++i) {
        EdgeNode *p = G->vertex[i].first;

        while (p) {
            ++a[p->adjvertex];
            p = p->next;
        }
    }
}
```

```
int *Get_Topo(ALGraph *G) {
    int *indegree = (int *)malloc(sizeof(int) * G->vertices);
    memset(indegree, 0, sizeof(int) * G->vertices); //主要对堆数据初始化
    int *topo = (int *)malloc(sizeof(int) * G->vertices);
    memset(topo, 0, sizeof(int) * G->vertices);

    InDegree(G, indegree); //获取全部点的indegree

    SqStack S;
    InitStack(&S);

    for (int i = 0; i < G->vertices; ++i)
        if (!indegree[i]) //如果indegree==0 入栈
            Push(&S, i);

    int m = 0; //用于控制topo下标
    int pop; //用于接收栈弹出值
    while (!IsEmpty(&S)) {
        Pop(&S, &pop); //弹出
        topo[m++] = pop; //栈顶即为拓扑序列顶点
        EdgeNode *p = G->vertex[pop].first;

        while (p) {
            //因为在逻辑上删除了pop点, 所以更新它邻接点的indegree
            --indegree[p->adjvertex];

            if (!indegree[p->adjvertex]) //判断indegree==0
                Push(&S, p->adjvertex);

            p = p->next;
        }
    }

    if (m < G->vertices) {
        //如果m=G->vertices 说明G为AOV
        //如果不说明有回路
        free(indegree);
        free(topo);
        return NULL; //返回空
    }
}
```

```

} else {
    free(indegree);
    return topo;
}
}

```

```

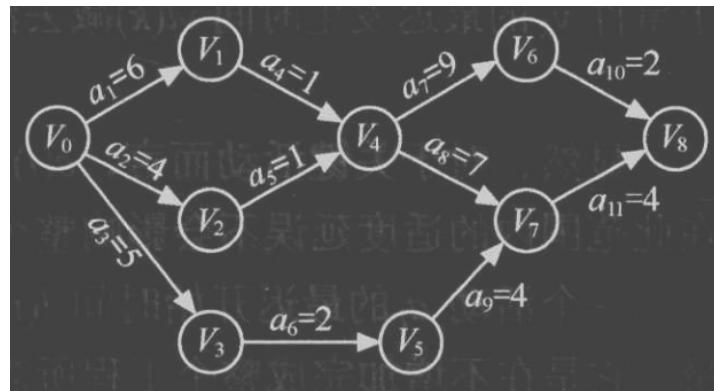
void TopoSort(ALGraph *G) {
    int *topo = Get_Topo(G);
    if (!topo) { //如果为NULL
        printf("The Graph is not AVO!\n");
        return;
    }

    for (int i = 0; i < G->vertices; ++i) {
        printf("%c ", G->vertex[topo[i]].name);
    }

    free(topo); //注意释放堆区数据
    topo = NULL;
}

```

• 关键路径



如上图AOE网，源点 v_0 (表示事件整体的开始)，汇点 v_8 (表示事件整体结束)

其他的点则表示一个活动的结束，同时也表示另外一个活动的开始

关 v_i 表示时间，而 a_i 表示活动， $w(a_i)$ 表示活动所需要的时间

令 $ve(i)$ 表示 v_i 的最早开始时间，即 v_i 时间之前的活动必须完成， v_i 才能开始

即 $ve(i) = \max(w(k, i) + ve(k))$, k 为*i*的直接前驱

令 $vl(i)$ 表示 v_i 的最迟开始时间，即 v_i 的后继点 v_m 需要为 $\min(v_j, v_m)$ 留出时间， v_j 为 v_m 的直接前驱

若已知 $vl(m)$ ，则 $vl(i) = \min(vl(m) - w(i, m))$

活动时间 $a_k = w(v_i, v_j)$ 表示边 (v_i, v_j) 的 weight

$e(a_k)$ 表示, a_k 发生的最早时间, 只有 v_i 发生了, a_k 才能发生

所以 $e(a_k) = ve(i)$

注意:

$ve(0) = 0$, 即源点 v_0 的最早发生时间为 0

$vl(n - 1) = ve(n - 1)$, 即汇点 v_{n-1} 的发生最早时间 = v_{n-1} 的最迟时间

由此可知 $ve(i)$ 从源点开始向前递归, $vl(i)$ 从后向前递归

$l(a_k)$ 表示, a_k 的最迟发生时间

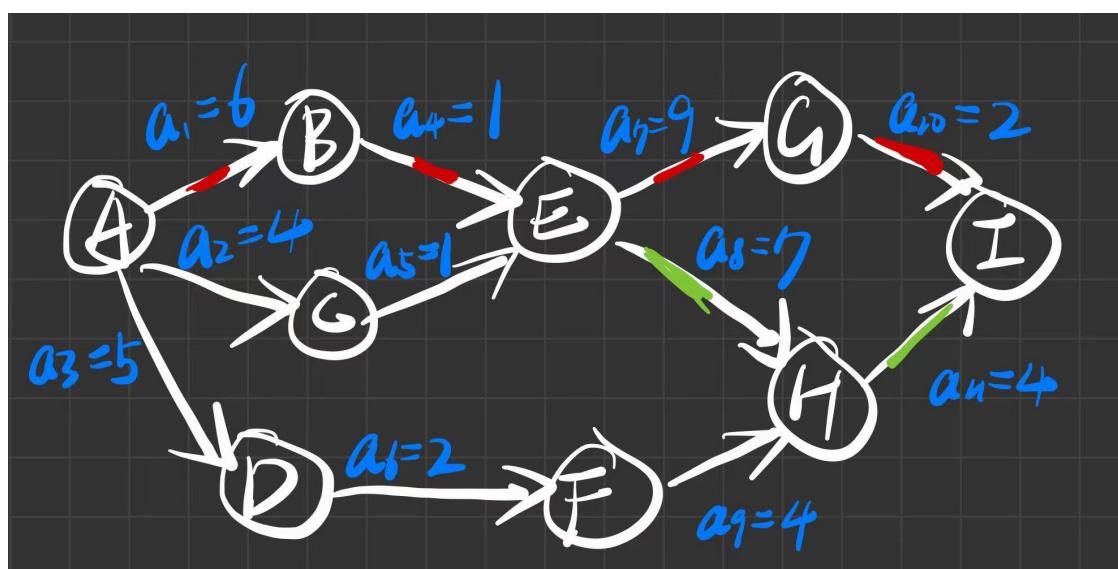
$l(a_k) = el(j) - w(v_i, v_j)$

若 $l(a_k) = e(a_k)$ 则说明活动 a_k 没有缓冲时间, 为关键路径

如果 $l(a_k) - e(a_k) = z(z \neq 0)$, 这说明活动 a_k 有 $\leq z$ 时间可以调整

代码实现

实现下图



邻接表

```

void CriticalPath(ALGraph *G) {
    int *topo = Get_Topo(G);
    if (!topo) {
        printf("The Graph is not AVO");
        return;
    }

    int n = G->vertices;
    int ve[n], vl[n]; // ve表示i顶点的最早发生时间, vl表示i顶点的最晚发生时间
    memset(ve, 0, sizeof(int) * n); // 初始化为0

    for (int i = 0; i < n; ++i) {
        int k = topo[i]; // 按照拓扑序列遍历各点的邻接点
    }
}

```

```

EdgeNode *p = G->vertex[k].first;
while (p) {
    if (ve[p->adjvertex] < ve[k] + p->weight) //求出最大值
        ve[p->adjvertex] = ve[k] + p->weight;
    p = p->next; //接着判断下一个邻接点
}
//把所有的顶点的最晚发生时间初始化为汇点的最晚发生时间
for (int i = 0; i < n; ++i)
    vl[i] = ve[topo[n - 1]]; // topo[n-1]的ve是最大的, 为下面求vl最小值做准备
//-----
for (int i = n - 1; i >= 0; --i) {
    //从汇点从后向前遍历拓扑序列
    int k = topo[i];

    EdgeNode *p = G->vertex[k].first;

    while (p) {
        if (vl[k] > vl[p->adjvertex] - p->weight)
            vl[k] = vl[p->adjvertex] - p->weight; //要vl的最小值
        p = p->next;
    }
}
//-----
for (int i = 0; i < n; ++i) {

    EdgeNode *p = G->vertex[i].first;
    while (p) {
        int j = p->adjvertex;
        int e = ve[i]; // 活动的最发生时间 = 该边的前驱节点i的最早发生时间
        //活动的最晚发生时间=该边前驱节点的最晚发生时间-该边的weight
        int l = vl[j] - p->weight;

        if (e == l) //如果等于即为关键路径
            printf("%c->%c ", G->vertex[i].name, G->vertex[j].name);

        p = p->next;
    }
}

free(topo);
}

```

• 图的头文件

- define_Graph.h

```

#include "Queue.h"
#include "Stack.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```
#define MAXWEIGHT 99999 //最大权值
#define MAXVERTEX 20 //最大定点数

typedef char VertexType; //顶点用字符表示
typedef int MatrixType; //矩阵类型

struct Uset {
    int adjvertex; //下标为i的点的邻接点
    int weight; //当前权值
};

int Min_Uset(Uset *U, int n);

struct Eset {
    int start;
    int end;
    int weight;
};

void Sort_Eset(Eset *E, int length);
```

- AMGraph.h

```
#include "define_Graph.h"

#ifndef __GRAPH_H
#define __GRAPH_H

typedef struct __AMGraph {
    VertexType vertex[MAXVERTEX];
    MatrixType edge[MAXVERTEX][MAXVERTEX];
    int vertices, edges;
} AMGraph;

int Locate_vertex(AMGraph *G, char v);
void print_Matrix_weighted(const AMGraph *G);
void print_Matrix_unweighted(const AMGraph *G);
void Creat_unAMGraph_unweightd(AMGraph *G);
void Creat_unAMGraph_weightd(AMGraph *G);
void Creat_AMGraph_weightd(AMGraph *G);
void Creat_AMGraph_unweightd(AMGraph *G);

void DFS_AM(AMGraph *G, int v, bool *visit);
void DFS_AMGraph(AMGraph *G, VertexType v);

void BFS_AM(AMGraph *G, int v, bool *visit);
void BFS_AMGraph(AMGraph *G, VertexType v);

void MST_Prim(AMGraph *G, VertexType v);

void InitEset(Eset *E, AMGraph *G);
void MST_Krusal(AMGraph *G);

void ShortestPath(AMGraph *G, VertexType v);
```

```

void ShortestPath_Dijkstra(AMGraph *G, VertexType v);

void ShortestPath(AMGraph *G, int v, int *path);
void ShortestPath_Dijkstra(AMGraph *G, VertexType v);
void print_path(AMGraph *G, int *path, int start, int end);

int **Path_Matrix(AMGraph *G);
void ShortestPath_Floyd(AMGraph *G, VertexType v1, VertexType v2);
#endif

```

- ALGraph.h

```

#include "define_Graph.h"

#ifndef __ALGRAPH_H
#define __ALGRAPH_H

typedef struct __EdgeNode {
    int adjvertex;
    __EdgeNode *next;
    int weight;
} EdgeNode;

typedef struct __ALGNode {
    VertexType name;
    EdgeNode *first;
} ALGNode;

typedef struct __ALGraph {
    ALGNode vertex[MAXVERTEX];
    int edges, vertices;
} ALGraph;

int Locate_vertex(const ALGraph *G, VertexType v);
void print_ALG_unweighted(const ALGraph *G);
void print_ALG_weighted(const ALGraph *G);
void Creat_unALGraph_unweighted(ALGraph *G);
void Creat_ALGraph_unweighted(ALGraph *G);
void Creat_unALGraph_weighted(ALGraph *G);
void Creat_ALGraph_weighted(ALGraph *G);
void DFS_AL(ALGraph *G, int v, bool *visit);
void DFS_ALGraph(ALGraph *G, VertexType v);

void BFS_AL(ALGraph *G, int v, bool *visit);
void BFS_ALGraph(ALGraph *G, VertexType v);

void MST_Prime(ALGraph *G, VertexType v);

void InitEset(Eset *E, ALGraph *G);
void MST_Krusal(ALGraph *G);

void InDegree(ALGraph *G, int *a);
int *Get_Topo(ALGraph *G);
void TopoSort(ALGraph *G);

```

```
void CriticalPath(ALGraph *G);  
#endif
```