数据结构笔记

Chunyin Chan

目录

1 绪论

数据: 客观描述事物属性的数, 字符及所有能被输入到计算机中并被程序识别和处理的符号集合

数据元素: 数据的基本单位, 作为一个整体进行考虑和处理. 根据业务确定具体构成.

数据对象: 具有相同属性的数据元素的集合. **不强调数据元素之间的关系**

数据类型: 1) 原子类型: 不可再分. 2) 结构类型: Struct. 3) 抽象数据类型 ADT: 数学化的语言, 定义数据的取值范围, 结构形式, 操作 (逻辑结构, 数据运算). 不关心存储结构, 与具体实现无关.

数据结构: 相互之间存在一种或多种特定**关系**的**数据元素**的集合. **三要素**: 逻辑结构, 存储结构, 数据的运算

1.1 数据元素三要素

1. 逻辑结构: 数据元素之间的逻辑关系, 与存储无关. 线性结构 (线性表); 非线性结构 (集合, 树, 图 …).

注 哈希表属于存储结构, 有序表 (有序线性表) 属于逻辑结构

1) 集合: 元素之间只有**同属于一个集合**的关系

2) 线性结构: **只存在一对一**关系, 除首末节点, 其余节点只有唯一前驱和唯一后继

3) 树形结构: **一对多**关系

4) 图状结构/网状结构: **多对多**关系

2. 存储结构: 也称为物理结构, 是数据结构在计算机中的表示 (映像). **影响**: (a) 存储空间分配的方便程度; (b) 对数据运算的速度. **主要有**: 顺序存储, 链式存储, 索引存储, 散列存储

1) 顺序存储: 逻辑上相邻, 物理上也相邻. 优点: 可以随机存取; 缺点: 只能使用一整块存储单元

2) 链式存储: 逻辑上相邻, 物理上不要求相邻. 使用指针表示元素之间的逻辑关系. 优点: 不会出现碎片现象; 缺点: 只能顺序存取; 指针占据额外存储空间

3) 索引存储: 建立附加的索引表. 索引表中的每项称为**索引项**. 一般形式: **(关键字, 地址)**. 优点: 检索速度快; 缺点: 索引表占据额外空间, 增删数据也要修改索引表, 耗费较多时间

4) 散列存储: 通过关键字直接计算出存储地址. 优点: 查增删效率高; 缺点: 较差的散列函数会造成冲突, 解决冲突会带来时间空间开销

3. 数据的运算: 运算的**定义**针对**逻辑结构**; **实现**针对**存储结构**.

注 不同的数据结构, 其逻辑结构, 存储结构不一定不同. EX. 链表实现队列和栈; 图的存储又邻接表, 邻接矩阵

注 存储数据时, 要存储数据的值和数据元素之间的关系. 数据的操作方法, 数据元素类型, 存取方法隐含在实现或逻辑结构中, 或可以通过上下文推断, 不需要显式存入.

1.2 算法

五个重要特征: 缺少任何一个都不是算法

1) 有穷性: 算法要在有穷步完成, 每一步要在有穷时间内完成.

2) 确定性: 每条指令要有确切含义, 无歧义. 相同输入得到相同输出.

3) 可行性: 能够通过代码实现.

4) 输入: 0 或多个输入

5) 输出: **1 或多个**输出

好的算法: 只要满足上述五个特征, 就算是算法

1) 正确性: 正确解决问题; 2) 可读性; 3) 健壮性: 对非法数据有反应或处理; 4) 高效率和低存储量需求: 时间空间复杂度低.

注 程序可以无穷, 只要不关掉.

确定性的无歧义 EX. 48, 76, 76 选择最大的元素 -> 同值时确定最大 -> 顺序前/后最大 (稳定排序)

1.2.1 时间复杂度

对于问题规模 n, 时间复杂度记为:

$$T(n) = O(f(n))$$

$$O(1) < O(\log_2^n) < O(n) < O(n\log_2^n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

1.2.2 空间复杂度

对于问题规模 n, **只考虑与问题规模 n 相关的额外空间/辅助空间**. 空间复杂度记为:

$$S(n) = O(f(n))$$

注 算法原地工作: 所需的辅助空间为 $O(1)$

1.2.3 空间复杂度计算

1. $S(n) = O$(递归深度)

```
Function fun(n):
   ...
   return fun(n-1)
```

2. $S(n) = n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2} -> O(n^2)$

```
Function fun(n):
   ...
   int a[n]
   fun(n-1)
   return
```

1.2.4 时间复杂度计算

1. 循环: 作表

```
int sum = 0;
for (int i=1; i<n; i *= 2)
   for (int j=0; j<i; j++)
      sum++;
```

令基本运算 sum++的执行次数为 x, 有:

i	j	x
1	0	2^0
2^1	$0 \sim 1$	2^1
2^2	$0 \sim 2^2 - 1$	2^2
$\cdots$	$\cdots$	$\cdots$
2^c	$0 \sim 2^c - 1$	2^c

运行次数之和 $\sum x = \frac{1(1-2^c)}{1-2} = 2^c - 1$

$2^c < n < 2^{c+1}$

$c < \log_2^n < c + 1$

$\log_2^n - 1 < c < \log_2^n$

故: $\sum x < n - 1$

故: $T(n) = O(n)$

2. 循环主体中变量参与循环条件判断

```
// 1
int i {1};
while (i <= n)
   i *= 2;
// 2
int y {5}:
while ((y+1) * (y+1) < n)
   y++;
```

1. 令基本运算 i *= 1的执行次数为 x, 则 $2^x \leq n$, 解得 $x \leq \log_2^n$, 故 $T(n) = O(\log_2^n)$

2. 令基本运算 y++的执行次数为 x, 则 $y = x + 5$, $(x + 5 + 1)^2 < n$, 解得 $x < \sqrt{n} - 6$, 故 $T(n) = O(\sqrt{n})$

3. 递归: 递推求解, 可能需要使用数学归纳 EX.

$$T(n) = 1 + T(n-1) = 1 + 1 + T(n-2) = \cdots = n - 1 + T(1)$$

即 $T(n) = O(n)$

2 线性表

线性表: 具有**相同**数据类型的 n 个**数据元素**的**有限序列**.

n 为**表长**, n=0 时, 线性表为**空表**. 将线性表命名为 L, 表示为 $L = (a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n)$. i 为位序, 1-index, 表示**第 i 个元素**; 数组索引为 0-index

表头元素: a_1, **表尾元素**: a_n. 表头元素表尾元素均唯一.

除第一个元素, 每个元素有且仅有一个直接前驱; 除最后一个元素, 每个元素有且只有一个直接后继.

线性表的基本操作: 创销增删查改

```
Initial(&L) // 初始化表, 创建一个空的线性表
Length(L) // 求表长. 返回: 表的长度(数据元素个数)
LocateElem(L, e) // 按值查找. 返回: L中元素e的位序
GetElem(L, i) // 按位查找. 返回: L中第i个元素
ListInsert(&L, i, e) // 插入. 将e插入到L的第i个位置
ListDelete(&L, i, &e) // 删除. 删除L中第i个元素, 并将删除的元素通过e返回
PrintList(L) // 打印表
Empty(L) // 判空. 返回: empty -> true; not empty -> false
DestroyList(&L) // 销毁. 销毁线性表并释放L所占空间
```

2.1 顺序表

顺序表: 线性表的顺序存储. 用一组**地址连续的单元**依次存储线性表中的数据元素. 逻辑上相邻, 物理上相邻.

特点: 1) 逻辑顺序与物理顺序相同; 2) 可随机存取; 3) 存储密度大; 4) 拓展容量不方便; 5) 插入删除需要移动大量元素, 时间开销大

优点: 1) 存储密度大; 2) 可以随机访问: 即通过元素首地址和元素序号可以在 O(1) 时间内找到指定元素

缺点: 1) 元素的插入和删除需要移动大量元素, 插入平均需要移动 $n/2$ 个元素, 删除平均需要移动 $(n-1)/2$ 个元素; 2) 顺序存储分配需要一段连续的存储空间

结构

```
// 静态分配
typedef struct
{
  ElemType data[MAXSIZE];
  int maxsize; // 最大长度, 为了方便操作加上, 若保证能访问宏定义MAXSIZE可以省略
  int length; // 当前长度
}SqList;

// 动态分配
typedef struct
{
  ElemType *data;
  int maxsize; // 当前最大长度
  int length; // 当前长度
}SeqList;
```

初始化表

```
// 初始化静态分配顺序表
void Initial(SqList &L)
{
  L.maxsize = MAXSIZE;
  L.length = 0;
}

// 初始化动态分配顺序表
bool Initial(SeqList &L)
{
  L.maxsize = MAXSIZE;
  L.length = 0;
  if ((L.data = new Type[MAXSIZE]))
    return true;
  return false;
}
```

销毁表

```
// 销毁动态分配顺序表
bool DestroyList(SeqList &L)
{
  L.maxsize = 0;
  L.length = 0;
  delete L.data; // free space
  return true;
}
```

扩展表

```
// 扩展表
bool IncreaseList(SeqList &L, int n)
{
  n += L.maxsize; // new data size
  if (Type* new_data = new Type[n]) // try to request space
  {
    for (int i=0; i<L.length; i++) // move elements
      new_data[i] = L.data[i];
    delete L.data; // free space
    L.data = new_data; // change data pointer
    L.maxsize = n; // new data size
    return true;
  }
  else return false;
}
```

查找: 按值查找: 最好 $O(1)$, 平均/最坏 $O(n)$; 按位查找: $O(1)$

```
// 按值查找
template <typename T>
int LocateElem(T L, Type e)
{
  // return order of elemet
  for (int i=0; i<L.length; i++)
    if (L.data[i] == e) return i+1;
  return -1;
}

// 按位查找
template <typename T>
Type GetElem(T L, int i)
{
  // param i: order of element
  if (i > L.length) // out of range
    return -INT_MAX;
  else return L.data[i-1];
}
```

插入 删除: 最好 $O(1)$, 平均/最坏: $O(n)$

```
// 插入元素
template <typename T>
bool ListInsert(T &L, int i, Type e)
{
  // param i: order of element
  // invalid i
  if (i < 1 || i > L.maxsize || L.length == L.maxsize)
    return false;
  for (int j=L.length; j>=i; j--) // move element [i, length] backward
    L.data[j] = L.data[j-1];
  L.data[i-1] = e; // insert element
  L.length++; // update length
  return true;
}

// 删除元素
template <typename T>
bool ListDelete(T &L, int i, Type &e)
{
  // param i: order of element
  if (i < 1 || i > L.length) // invalid i
    return false;
  e = L.data[i-1]; // save deleting element
  for (int j=i; j<L.length; j++) // move element[i+1, length] forward
    L.data[j-1] = L.data[j];
  L.length--; // update length
  return true;
}
```

表长 打印 判空

```
template <typename T> // 表长
int Length(T L)
{
  return L.length;
}

template <typename T> // 打印表
void PrintList(T L)
{
  for (int i=0; i<L.length; i++)
    cout << L.data[i] << " ";
  cout << endl;
}

template <typename T> // 判空
bool Empty(T L)
{
  return (L.length == 0);
}
```

2.2 线性表的链式表示 链表

2.2.1 单链表

单链表: 线性表的链式存储. 通过一组任意的存储单元来存储线性表中的数据元素. 对每个链表节点, 存放一个**指向后继的指针**. 单链表结点结构: `data: 数据域，存数据; next: 指针域，存后继地址`

单链表的元素离散分布, 属于**非随机存取的存储结构**. 通常用头指针 `L`或者 `head`标识单链表, 指出链表的起始地址

链表为空表: 不带头结点: `L == nullptr`; 带头结点 `L->next ==nullptr`

头结点: 在单链表第一个数据节点之前附加的结点; **首元结点**: 第一个数据结点

引入头结点的优势: 1) 链表第一个位置上的操作和其他位置上的操作一致, 无需特殊处理; 2) 无论链表是否为空, 头指针都是指向头结点的非空指针, 统一空表和非空表的处理.

结构

```
typedef struct LNode{
  Type data;
  LNode* next;
}LNode, *LinkList;
```

初始化表

```
// 带头结点
bool InitList(LinkList &L)
{
  L = new LNode;
  L->next = nullptr;
  return true;
}

// 不带头结点
bool Initial_nohead(LinkList &L)
{
  L = nullptr;
  return true;
}
```

求表长 打印 判空 求表长: $O(n)$

```
int Length(LinkList L)
{
  int n {0};
  LNode *p = L->next;
  while (p->next != nullptr)
  {
    n++;
    p = p->next;
  }
  return n;
}

void Print(LinkList L)
{
  LNode *p = L->next;
  while (p)
  {
    cout << p->data << " -> ";
    p = p->next;
  }
  cout << "end" << endl;
}

bool Empty(LinkList L)
{
  if (L->next) return false;
  else return true;
}
```

查找 $O(n)$

```
// 按位查询
LNode* GetElem(LinkList L, int n)
{
  /*
  n : 位序
  */
  int i {0}; // 头结点计数为0
  LNode* p = L;
  while (p && i < n)
  {
    p = p->next;
    i++;
  }
  return p;
}

// 按值查询
LNode* LocateElem(LinkList L, Type x)
{
  /*
  返回目标结点指针
  找不到则返回空指针
  */
  LNode* p = L->next;
  while (p && p->data != x)
    p = p->next;
  return p;
}
```

后插 给定位序: $O(n)$; 给定结点指针: $O(1)$

```
// 给定位序
bool ListInsert_backward(LinkList &L, int n, Type x)
{
  int i {0};
  LNode *p = L;
  // find n
  while (p && i < n)
  {
    p = p->next;
    i++;
  }
  if (p)
  {
    LNode *node = new LNode;
    node->data = x;
    node->next = p->next;
    p->next = node;
    return true;
  }
  else return false;
}

// 给定结点
bool ListInsert_backward_node(LNode* node, Type x)
{
  if (node)
  {
    LNode *p = new LNode;
    p->data = x;
    p->next = node->next;
    node->next = p;
    return true;
  }
  else return false;
}
```

前插 给定位序: $O(n)$; 给定结点指针: $O(1)$

```
// 给定位序
bool ListInsert_forward(LinkList &L, int n, Type x)
{
  int i {0};
  LNode* p = L;
  // find n-1
  while(p->next && i < n-1)
  {
    p = p->next;
    i++;
  }
  if (p->next)
  {
    // new node
    LNode *node = new LNode;
    node->data = x;
    // modify pointer
    node->next = p->next;
    p->next = node;
    return true;
  }
  else return false;
}

// 给定结点
bool ListInsert_forward_node(LNode* node, Type x)
{
  if (node)
  {
    LNode* s = new LNode;
    s->data = x;
    s->next = node->next;
    node->next = s;
    // swap data
    Type t = node->data;
    node->data = s->data;
    s->data = t;
    return true;
  }
  else return false;
}
```

删除结点

```
// 给定位序
bool ListDelete(LinkList &L, int n, Type &e)
{
  /*
  delete n-th node
  copy data to e
  */
  int i {0};
  LNode *p = L;
  // find n - 1
  while (p && i < n-1)
  {
    p = p->next;
    i++;
  }
  if (p->next && i == n-1)
  {
    LNode *t = p->next;
    p->next = t->next;
    e = t->data;
    delete t;
    return true;
  }
  else return false;
}
```

销毁表

```
bool ListDestroy(LinkList &L)
{
  /*
  retain head node
  */
  LNode *p = L->next;
  LNode *q;
  while (p)
  {
    q = p->next;
    delete p;
    p = q;
  }
  L->next = nullptr;
  return true;
}
```

头插法建立单链表 插入一个结点 $O(1)$, 表长为 n, 总计 $O(n)$. 链表元素顺序与读入相反, 可用于**逆置链表**.

```
LinkList BuildList_HeadInsert(Type x[], int n)
{
  LNode *L = new LNode;
  L->next = nullptr;
  for(int i=0; i<n; ++i)
  {
    LNode *s = new LNode;
    s->data = x[i];
    s->next = L->next;
    L->next = s;
  }
  return L;
}
```

尾插法建立单链表 $O(n)$. 需要附设一个尾指针 `r`; 链表元素顺序与读入顺序相同

```
LinkList BuildList_TailInsert(Type x[], int n)
{
  LNode *L = new LNode;
  LNode *end = L;
  for (int i=0; i<n; ++i)
  {
    LNode *s = new LNode;
    s->data = x[i];
    end->next = s;
    end = s;
  }
  end->next = nullptr;
  return L;
}
```

2.2.2 双链表

双链表: 在单链表的基础上增加一个指针, 共有两个指针 `prior, next`, 分别指向前驱和后继.

结构

```
typedef struct DNode{
  Type data;
  DNode *prior, *next;
}DNode, *DLinkList;
```

初始化

```
bool Initial_Dlink(DLinkList &L)
{
  DNode* head = new DNode;
  head->data = 0;
  head->prior = nullptr;
  head->next = nullptr;
  L = head;
  return true;
}
```

前插 给定位序: $O(n)$; 给定结点指针: $O(1)$

```
bool ListInsert_forward(DLinkList &L, int n, Type x)
{
  if (n < 1) return false; // n out of range
  DNode* node = new DNode;
  node->data = x;
  // handle n = 1 and list's length = 0
  if (!(L->next) && n == 1)
  {
    node->next = nullptr;
    node->prior = L;
    L->next = node;
    return true;
  }
  DNode* p = L->next;
  int i {1};
  // search n-th node
  while (i < n && p)
  {
    p = p->next;
    i++;
  }
  if (i < n || !p) return false; // n out of range
  // insert forward
  p = p->prior; // point to previous node, n-1
  node->next = p->next;
  node->prior = p;
  p->next->prior = node;
  p->next = node;

  return true;
}

bool ListInsert_forward_node(DNode* node, Type x)
{
  /*
  NOT allow inserting node before head node
  */
  if (!(node->prior)) return false; // node is head
      node
  DNode* p = new DNode;
  p->data = x;
  p->next = node;
  p->prior = node->prior;
  node->prior->next = p;
  node->prior = p;

  return true;
}
```

后插 给定位序: $O(n)$; 给定结点指针: $O(1)$

```
bool ListInsert_backward(DLinkList &L, int n, Type x)
{
  if (n < 0) return false; // n out of range
  DNode* node = new DNode;
  node->data = x;
  // handle n == 0 and list's length = 0
  if (n == 0 && !(L->next))
  {
    node->next = nullptr;
    node->prior = L;
    L->next = node;
  }
  DNode* p = L->next;
  int i {1};
  // search n-th node
  while (i < n && p)
  {
    p = p->next;
    i++;
  }
  // n out of range
  if (i < n || p == L) return false;
  node->next = p->next;
  node->prior = p;
  p->next->prior = node;
  p->next = node;

  return true;
}

bool ListInsert_backward_node(DNode* node, Type x)
{
  DNode* p = new DNode;
  p->data = x;
  p->next = node->next;
  p->prior = node;
  node->next->prior = p;
  node->next = p;

  return true;
}
```

删除结点 给定位序: $O(n)$; 给定结点指针: $O(1)$

```
bool ListDelete(DLinkList &L, int n)
{
  if (n < 1) return false; // n out of range
  DNode* p = L->next;
  int i{1};
  while (i < n && p)
  {
    p = p->next;
    i++;
  }
  // n out of range
  if (i < n || !p) return false;
  p->next->prior = p->prior;
  p->prior->next = p->next;
  delete p;

  return true;
}
```

前向遍历

```
void traverse_forward(DNode* node)
{
  /*
  skip head node
  */
  DNode* p = node;
  while (p)
  {
    if (!(p->prior)) break;; // access head node; if allow to access head node, omment out this line
    cout << p->data << " ; ";
    p = p->prior;
  }
  cout << "end" << endl;
}
```

后向遍历

```
void traverse_backward(DNode* node)
{
  /* skip head node */
  DNode* p = node;
  while (p)
  {
    if (!(p->prior)) p = p->next; // access head node; if allow to access head node, comment out this line
    cout << p->data << " ; ";
    p = p->next;
  }
  cout << "end" << endl;
}
```

2.2.3 循环单链表

将尾结点的 next指向头结点. 若设置尾指针, 则在表头和表尾插入元素均只需要 $O(1)$; 设置头指针, 表头插入 $O(1)$, 表尾插入 $O(n)$

判空: L->next ==L; **判尾结点**: p->next ==L

2.2.4 循环双链表

头结点的 prior指向尾结点, 尾结点的 next指向头结点.

判空: L->next == L && L->prior == L; **判尾结点**: p->next == L

2.3 静态链表

静态链表: 用数组来表述线性表的链式存储结构.

结点结构: data: 数据域; next: 指针域, 其中, next 称为**游标**, 指向下一个元素的数组下标, 尾结点的 next = -1. 空闲元素的 next = -2

适用: 1) 不支持指针的语言; 2) 数据元素数量不变 (EX. 文件分配表 FAT)

优点: 1) 增删不需要移动大量元素

缺点: 1) 容量固定; 2) 无随机存取性

结构

```
typedef struct
{
  Type data;
  int next; // next: >0: pointer; -1: end of list; -2: available node
  int size;
}SNode, *SLinkList;
```

初始化表

```
bool Initial_SLinkList(SLinkList &L, int capacity)
{
  L = new SNode[capacity];
  if (!L) return false; // not enough memory
  for (int i=0; i<capacity; i++) L[i].next = -2; // initial next field
  L[0].next = -1; // head node
  L->size = capacity;
  return true;
}
```

插入结点 $T(n)=O(n)$

```
bool SListInsert(SLinkList &L, int n, Type x)
{
  if (n < 1) return false; // n out of range
  // find available node
  int node {-1};
  for (int i=1; i<L->size; i++)
  {
    if (L[i].next == -2) { node = i; break; }
  }
  if (node == -1) return false; // full list
  // find n-1 node
  int prior {0}; int idx {0};
  while (idx < n - 1 && prior != -1)
  {
    prior = L[prior].next;
    idx++;
  }
  // invalid n
  if (idx < n - 1) return false; // invalid n

  // modify next and data field
  L[node].data = x;
  L[node].next = L[prior].next;
  L[prior].next = node;

  return true;
}
```

删除结点 $T(n)=O(n)$

```
bool SListDelete(SLinkList &L, int n)
{
  // n out of range
  if (n < 1) return false;
  // find n-1
  int prior {0}; int cur = L[prior].next;
  int idx {0}; // order of prior
  while (idx < n-1 && prior != -1)
  {
    prior = cur;
    cur = L[cur].next;
    idx++;
  }
  // invalid n
  if (idx < n - 1 || prior == -1) return false;
  // modify pointer field in prior
  L[prior].next = L[cur].next;
  // delete n node
  L[cur].next = -2;

  return true;
}
```

打印表

```
void Print_SList(SLinkList L)
{
  cout << "-----Details of List-----" << endl;
  cout << "format: Data (next)" << endl;
  string data; int next;
  for (int i=0; i<L->size; i++)
  {
    next = L[i].next;
    data = next != -2 ? to_string(L[i].data) : "none";
    cout << data << "(" << next << ")" << " -> ";
  }
  cout << "end" << endl;
  cout << "-----order output-----" << endl;
  int p = L[0].next;
  while (p != -1)
  {
    cout << L[p].data << " -> ";
    p = L[p].next;
  }
  cout << "end" << endl;
}
```

2.4 顺序表和链表的比较

比较两种数据结构: 1) 逻辑结构; 2) 物理结构; 3) 运算 (创销增删查改)

存取: 顺序表可以顺序存取和随机存取; 链表只能顺序存取

逻辑结构与物理结构: 顺序表: 逻辑相邻物理相邻; 链表: 逻辑相邻物理不一定相邻

数据运算: **查找**: 链表: $O(n)$; 顺序表: 有序 $O(n\log^n)$, 无序 $O(n)$; **插入删除**: 顺序表需要移动元素, $O(n)$; 链表视情况而定, 时间复杂度主要在定位到目标结点上.

存储空间分配: 顺序表的元素数量不能超出预先分配的大小, 动态扩增顺序表大小会带来大量时间开销, 且内存中若无足够连续空间, 会导致扩增失败; 链表可以在需要时申请空间, 但存储密度较低.

选用考量: 1) **存储**: 难以估计长度, 使用链表; 2) **运算**: 若经常**按位查找/按序号查找 (业务场景偏向静态)**: 顺序表; 经常**插入删除 (业务场景偏向动态)**: 链表.

3 栈

栈: 受限的线性表, **只允许在一端进行插入删除的线性表**. **栈顶**: 允许进行插入删除的一端; **栈底**: 固定的, 不允许进行插入删除; **空栈**: 不含任何元素的空表. 操作特性: **后进先出, Last In First Out, LIFO**

卡特兰数: 当 n 个不同元素进栈时, 出栈的可能序列有 $\frac{1}{n+1}\binom{2n}{n}$.

栈的基本操作

```
bool InitialStack(SqStack &s); // 初始化栈
bool StackEmpty(SqStack s); // 判断栈是否为空
bool Push(SqStack &s, Type x); // 将x压入栈顶
bool Pop(SqStack &s, Type &x); // 从栈顶弹出元素, 用x返回
bool GetTop(SqStack s, Type &x); // 读取栈顶元素, 非空则用x返回
bool DestroyStack(SqStack &s); // 销毁栈
```

顺序栈结构

```
typedef struct{
  Type data[MAXSIZE]; // 元素
  int top;    // 栈顶指针
}SqStack;
```

顺序栈基本操作

```
// 初始化栈
bool InitialStack(SqStack &s)
{
  s.top = -1;
  return true;
}

// 判断栈是否为空
bool StackEmpty(SqStack s)
{
  return s.top == -1;
}

// 将x压入栈顶
bool Push(SqStack &s, Type x)
{
  if (s.top == MAXSIZE) return false; // full stack
  s.data[++s.top] = x;
  return true;
}

// 从栈顶弹出元素, 用x返回
bool Pop(SqStack &s, Type &x)
{
  if (s.top == -1) return false; // empty stack
  x = s.data[s.top--];
  return true;
}
```

```
29
30 // 读取栈顶元素，非空则用x返回
31 bool GetTop(SqStack s, Type &x)
32 {
33   if (s.top == -1) return false; // empty satck
34   x = s.data[s.top];
35   return true;
36 }
```

s.top 为顺序栈的栈顶指针, 初始化时有 s.top=-1 和 s.top=0 两种初始化方法, 两种初始化方法的入栈, 出栈, 判空, 判满操作不同.

	s.top = -1	s.top = 0
入栈	s.data[++s.top] = x	s.data[s.top++] = x
出栈	x = s.data[s.top–]	x = s.data[–s.top]
判空	s.top == -1	s.top == 0
判满	s.top == MAXSIZE - 1	s.top == MAXSIZE

共享栈: 同一块连续内存区域 M[0...MAXSIZE-1], M[0] 时栈 0 的栈底, M[MAXSIZE-1] 是栈 1 的栈底.

判栈空: 栈 0: `top0 == -1`; 栈 1: `top1 == MAXSIZE`

入栈: 栈 0: `data[++top0] = x`; 栈 1: `data[--top1] = x`

出栈: 栈 0: `x = data[top0--]`; 栈 1: `x = data[top1++]`

栈满: `top1 - top0 == 1`

共享栈是为了更好地利用空间, 只有在整个存储空间都满时才会发生上溢. 存取的 T(n) = O(1). 共享栈减少了上溢的可能性, 相比于两个栈独自分配空间, 共享栈合并了两个栈的空间, 相比于独立空间更大, 因此, 一个栈可以使用另一个栈的存储空间, 减少了上溢的可能性.

上溢: 存储区域满但继续写入, 浮点数中过大设为正无穷; **下溢**: 存储区域已空仍弹出元素, 浮点数中过小归零

链栈

链栈结构

```
1 typedef struct LinkNode{
2   Type data; // 数据域
3   struct LinkNode *front, *rear; // 头尾指针
4 }LinkNode, LiStack;
```

链栈不存在栈满上溢的情况, 并规定**所有的操作都在单链表的表头进行**, 通常链栈不设置头结点. head指向栈顶, 表尾为栈底. 因此, 入栈出栈操作即在单链表的头结点前插入结点或释放当前头结点.

4 队列

队列 (Queue), 简称队, 一种操作受限的线性表, 只允许**在表的一端插入, 在另一端删除**. 插入元素: 入队, 删除元素: 出队. **先进先出, First In First Out, FIFO**

队头 (front), 队首: 允许删除的一端; **队尾 (rear)**: 允许插入的一端; 空队列

顺序存储结构

```
typedef struct{
  Type data[MAXSIZE];
  int front, rear; // 队头和队尾指针
} SqQueue;
```

基本操作

```
bool InitQueue(SqQueue &Q); // 初始化队列
bool QueueEmoty(SqQueue Q); // 判断队空
bool EnQueue(SqQueue &Q, Type x); // 元素x入队
bool DeQueue(SqQueue &Q, Type &x); // 元素出队并用x返回
bool GetHead(SqQueue Q, Type &x); // 读取队头元素并用x返回
```

4.1 顺序队列

会假溢出, 几乎不用.

初始化: `Q.front = 0; Q.rear = 0;` 入队: `Q.data[Q.rear++] = x` 出队: `x =Q.data[Q.front++]`

不能使用 `Q.rear == MAXSIZE` 来判断队满, 只要累计入队次数达到 MAXSIZE, Q.rear 就会指向 MAXSIZE, 造成假溢出.

4.2 循环队列

存储上与顺序队列一致, 操作上增加对 MAXSIZE 的取余操作使得在逻辑上将队列视作环状.
初始化: `Q.front = 0; Q.rear = 0;`
队头指针加一: `Q.front = (Q.front + 1) % MAXSIZE`
队尾指针加一: `Q.rear = (Q.rear + 1) % MAXSIZE`
队列长度: `(Q.rear + MAXSIZE - Q.front) % MAXSIZE` (Q.rear + MAXSIZE 使得 rear 在数值上大于 front, 再取余数保留 rear 和 front 差值的绝对值大小)

使用顺序表的处理思路, 会出现初始化和队满时均有 `Q.rear == Q.front`, 因此该条件不能判断队满, 判断队满有 3 种思路:

(1) **闲置一个存储单元**来区分队空和队满, 队列可用长度-1, 较常用.
队满: `(Q.rear + 1) % MAXSIZE == Q.front`
队空: `Q.rear == Q.front`
队中元素个数: `(Q.rear + MAXSIZE - Q.front) % MAXSIZE`

(2) **结构体中增加成员 size**, 初始化为 0, 插入成功 +1, 删除成功-1.

队满: `size == MAXSIZE`

队空: `size == 0`

该方案中, 队满队空都可能出现 `Q.rear == Q.front`

(3) **结构体中增加标志位 tag**, 删除成功置 0, 插入成果置 1.

队满: `tag == 1 && Q.rear == Q.front`

队空: `tag == 0 && Q.rear == Q.front`

基本操作

```
// 初始化队列
bool InitQueue(SqQueue &Q)
{
  Q.front = Q.rear = 0;
  return true;
}

// 判断队空
bool QueueEmpty(SqQueue Q)
{
  return Q.rear == Q.front;
}

// 元素x入队
bool EnQueue(SqQueue &Q, Type x)
{
  // if full
  if ((Q.rear + 1) % MAXSIZE == Q.front) return false;
  Q.data[Q.rear] = x; // enqueue
  Q.rear = (Q.rear + 1) % MAXSIZE; // update pointer
  return true;
}

// 元素出队并用x返回
bool DeQueue(SqQueue &Q, Type &x)
{
  // if empty
  if (Q.front == Q.rear) return false;
  x = Q.data[Q.front];
  Q.front = (Q.front + 1) % MAXSIZE; // update pointer
  return true;
}

// 读取队头元素并用x返回
bool GetHead(SqQueue Q, Type &x)
{
  // if empty
  if (Q.front == Q.rear) return false;
  x = Q.data[Q.front];
  return true;
}
```

4.3 链队列

通常将链队列设计为一个**带头结点的单链表**, 并追踪头结点指针和尾结点指针. 适合数据元素变动较大的情形, 不存在队满且溢出的问题.

链队列结构

```
typedef struct LinkNode{
  Type data;
  struct LinkNode *next;
} LinkNode;

typedef struct LinkQueue{
  LinkNode *front, *rear;
} LinkQueue;
```

基本操作

```
// 初始化
bool InitQueue(LinkQueue &Q)
{
  LinkNode *head = new LinkNode;
  if (!head) return false; // not enough memory
  head->next = nullptr;
  Q.front = Q.rear = head;
  return true;
}

// 判空
bool QueueEmpty(LinkQueue Q)
{
  return (Q.rear == Q.front);
}

// 元素x入队
bool EnQueue(LinkQueue &Q, Type x)
{
  LinkNode *node = new LinkNode;
  // new node
  if (!node) return false; // memory fail to allocate
  node->data = x;
  node->next = nullptr;
  // modify pointer
  Q.rear->next = node;
  Q.rear = node;
  return true;
}
```

```
// 出队，并用x返回
bool DeQueue(LinkQueue &Q, Type &x)
{
  // if empty
  if (QueueEmpty(Q)) return false;
  LinkNode *p = Q.rear->next;
  x = p->data;
  // release node
  Q.rear->next = p->next;
  if (Q.rear == p) Q.rear = Q.front; // only one data node
  delete p;
  return true;
}
```

4.4 双端队列

两端都可以输入和输出的队列.

输入受限的双端队列: 只能一端输入, 两端都能删除.

输出受限的双端队列: 只能一端删除, 两端都能插入.

一般考察输出顺序是否合法, 一个元素在输出时, 主要考察输入输出顺序是否合法. 先将输入序列分为可以立刻输入输出的部分和输入后延迟输出的部分, 延迟输出的部分写出输出顺序, 按照输入序列从早到晚判断是否合法.

EX. 输入受限. 输入: 1 2 3 4 5. 输出: 4 2 1 3 5 合法. 5 输入后立刻输出. 右边输出的输出顺序: 3124, Inp1, RInp2, LInp3, RInp 4.

输出: 4 1 3 2 5 非法. 5 输入后立刻输出, 右边输出的输出顺序: 2314, Inp1, LInp2, 3 不能插入 1 2 之间, 非法

4.5 栈和队列的应用

4.5.1 括号匹配

利用栈实现匹配:

1. 初始化一个空栈;
2. 遇到左括号, 入栈;
3. 遇到右括号:
   (1) 栈顶为匹配的左括号, 出栈, 找到一个匹配对;
   (2) 栈顶非匹配的左括号或栈空(EX. [ )), 非法序列结束.
4. 序列读入完: (1) 栈空, 匹配; (2) 栈非空, 不匹配

```cpp
bool bracket_pair(std::string s)
{
  using namespace std;
  // initial stack
  SqStack stack;
  InitialStack(stack);
  vector <char> lbrackets {'(', '[', '{'};
  vector <char> rbrackets {')', ']', '}'};
  char e;

  for (char c : s)
  {
    auto it = find(lbrackets.begin(), lbrackets.end(), c);
    // left bracket
    if (it != lbrackets.end()) Push(stack, c); // push in stack
    // right brackt
    else {
      auto it = find(rbrackets.begin(), rbrackets.end(), c);
      if (it == rbrackets.end()) continue; // not a bracket
      GetTop(stack, e);
      // pair bracket
      if (c == ')' && e == '(') Pop(stack, e);
      else if (c == ']' && e == '[') Pop(stack, e);
      else if (c == '}' && e == '{') Pop(stack, e);
      else return false; // unpair bracket occur
    }
  }
  if (StackEmpty(stack)) return true;
  else return false;
}
// string s1 {"((((([()])))))"}; // true
// string s2 {"{{[()[]]}}"}; // true
// string s3 {"[][]("}; // false
```

4.5.2 算术表达式格式转换

前缀表达式和后缀表达式 (逆波兰式) 不必须要括号, 中缀表达式必须要括号. 前缀和后缀表达式只有操作数和运算符. 前中后缀是符号的前中后缀. 中缀按照可能运算顺序的不同, 可以有多个不同的前缀和后缀表达式
EX. 中缀: `3 + 4 * (2 - 1)`, 前缀: `+ 3 * 4 - 2 1`, 后缀: `3 4 2 1 - * +`

中缀转前缀: **先操作符后操作数**, 递归. `3 + 4 * (2 - 1)`, 得 `+ (3) (4 * (2-1))`
得到: `+ 3 前缀(4 * (2-1)) -> + 3 * 4 前缀(2 -1) -> + 3 * 4 - 2 1`

中缀转后缀: **先操作数后操作符**, 递归. 得 `(3) (4 * (2 -1)) +`
得到: `3 后缀(4 * (2 -1)) + -> 3 4 后缀(2 - 1) * + -> 3 4 2 1 - * +`

计算机转换为后缀表达式, 使用栈:
依次遍历表达式各字符:
1. 遇到操作数: 直接加入表达式;
2. 遇到括号: 若 (, 入栈; 若), 依次弹出栈内各操作符加入运算表达式直至遇到 (;
3. 遇到操作符: 若栈顶的操作符优先度更高或相等, 弹出, 直至优先级低于自身 (先算优先级高的) 或遇到 (; 若栈顶优先级低于自身, 自身入栈.
EX. 2 * 3 + (4 * (5 - 2)):

```
 2 ->  2  -> 2 3 -> 2 3 * -> 2 3 * -> 2 3 * -> 2 3 * 4 -> 2 3 * 4 -> 2 3 * 4 -> 2 3 * 4 5
[ ]-> [*] -> [*] ->  [ ]  ->  [+]  -> [+(]  ->   [+(]  ->   [+(*] ->  [+(*(] -> [+(*(]
-> 2 3 * 4 5 -> 2 3 * 4 5 2 -> 2 3 * 4 5 2 - -> 2 3 * 4 5 2 - * -> 2 3 * 4 5 2 - * +
->  [+(*(-]  ->    [+(*(-]  ->     [+(*]     ->      [+]        ->      []
```

```
// 123 + 45 * 67 / (89 - 12) * 3 + 456 * (78 - 9) // 123 45 67 * 89 12 - / 3 * + 456 78 9 - * +
std::string expression_infix2post(std::string e)
{
  using namespace std;
  string post {""};
  SqStack stack;
  InitialStack(stack);

  int n = e.length();
  int begin, end; // split string
  begin = end = 0;
  string temp, top;
  int flag; //0: operator, 1: digit, 2: bracket

  vector<char> operators {'+', '-', '*', '/'};
  vector<char> digits {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
  map<string, int> precedence {
    {"+", 1}, {"-", 1}, {"*", 2}, {"/", 2}, };

  while (end < n) {
    // split expression
    auto it = find(operators.begin(), operators.end(), e[end]);
    if (it != operators.end()) { // operator
      temp = e[end]; end++;
      flag = 0;
    }
```

```
    else if (e[end] == ' ') { // space
      end++;
      continue;
    }
    else if (e[end] == '(' || e[end] == ')') { // bracket
      temp = e[end]; end++;
      flag = 2;
    }
    else { //digit
      begin = end;
      while (end < n && find(digits.begin(), digits.end(), e[end]) != digits.end())
        end++;
      temp = e.substr(begin, end - begin); // (end - 1) - begin + 1
      flag = 1;
    }

    // action
    if (flag == 2) { // bracket
      if (temp == "(") Push(stack, temp);
      else {
        GetTop(stack, top);
        while (top != "(") {
          post += " "; post += top;
          Pop(stack, top); GetTop(stack, top);
        } // while
        Pop(stack, top); // pop '('
      } //else
    }
    else if (flag == 1) { // digit
      post += " "; post += temp;
    }
    else { // operator
      while (!StackEmpty(stack)) {
        GetTop(stack, top);
        if (top == "(") break;
        if (precedence[top] >= precedence[temp]) {
          post += " "; post += top;
          Pop(stack, top);
        }
        else break;
      }
      Push(stack, temp);
    }
  }
  // clear stack
  while (!StackEmpty(stack)) {
    Pop(stack, top);
    post += " "; post += top;
  }
  return post;
}
```

4.5.3 表达式求值

中缀表达式求值: 双栈, 操作数栈 operand, 操作符栈 operator.

遍历各个字符:

1. 如果是数字: 压入operand;
2. 如果是括号:
   (1) 左括号, 压入operator;
   (2) 按照运算方法, 不断弹出一个操作符, 两个操作数计算,
   并将结果压入operand, 直到operator栈顶为(, 弹出并抛弃;
3. 如果是操作符:
   (1) operator栈顶优先级高于等于自身, 弹出操作符<op>, 依次弹出两个操作数Y和X,
   计算X<op>Y, 将结果压入operand;
   (2) 优先级低于自身, 自身压入operator;

遍历结束:

(1) operator非空, 不断退栈, 按照弹出<op>YX, 压入X<op>Y的原则清理双栈
(2) operator空, operand为结果

后缀表达式求值: 单栈

遍历各字符:

1. 如果是操作数, 压入栈;
2. 如果是操作符<op>, 依次弹出Y和X, 计算X<op>Y并压入栈

遍历完序列: 栈顶为结果

```
// 123 + 45 * 67 / (89 - 12) * 3 + 456 * (78 - 9) // 31704.5
double calculate_infix(std::string infix)
{
  using namespace std;

  string post = expression_infix2post(infix);
  stack<double> stk;
  int n = post.length();
  string temp;
  double X, Y;
  int begin, end;
  begin = end = 0;

  vector<char> operators {'+', '-', '*', '/'};
  vector<char> digits {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

  while (end < n) {
    // split expression
    auto it = find(operators.begin(), operators.end(), post[end]);
    if (it != operators.end()) { // operator
      temp = post[end]; end++;
      Y = stk.top(); stk.pop();
      X = stk.top(); stk.pop();
      if (temp == "+") stk.push(X + Y);
      else if (temp == "-") stk.push(X - Y);
      else if (temp == "*") stk.push(X * Y);
      else if (temp == "/") stk.push(X / Y);

    }
    else if (post[end] == ' ') { // space
      end++;
      continue;
    }
    else { //digit
      begin = end;
      while (end < n && find(digits.begin(), digits.end(), post[end]) != digits.end())
        end++;
      temp = post.substr(begin, end - begin); // (end - 1) - begin + 1
      stk.push(stod(temp));
    }
  }
  return stk.top();
}
```

---

5 数组和特殊矩阵

数组: n 个相同类型元素组成的线性表. **数组元素**: 每个数据元素. **下标**: 每个元素在数组中的序号. **维界**: 下标的取值范围

数组的存储: A[0...n-1]: $LOC(a_i) = LOC(a_0) + i \times L \ \ (0 \le i < n)$

假设: 行下标的范围为 $[0...r]$, 列下标的范围为 $[0...c]$, 每个数组元素占 L 字节

行优先: 第 i+1 行有 j 个, 前 i 行每行 c+1 列存满. $LOC(a_{i,j}) = LOC(a_{0,0}) + L \times [(c+1) \times i + j]$

列优先: 第 j+1 列有 i 个, 前 j 列每列 r+1 行存满. $LOC(a_{i,j}) = LOC(a_{0,0}) + L \times [(r+1) \times j + i]$

5.1 特殊矩阵压缩存储

对称矩阵

将 n 阶二维矩阵存储在一维矩阵 $B[n(n+1)/2]$

由于 $a_{i,j} = a_{j,i}$, 可以只存储对角线和一个三角部分, 下标无区别.
$a_{i,j}$ 在一维数组中的序号为: $1 + 2 + \cdots + i - 1 + j = \frac{i(i-1)}{2} + j$
对于上三角的元素, 交换 i, j 可得到相对应的元素下标 `B[k]`.

0-index:

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1, & i \ge j \text{下三角} \\ \frac{j(j-1)}{2} + i - 1, & i < j \text{上三角} \end{cases}$$

1-index:

$$k = \begin{cases} \frac{i(i-1)}{2} + j, & i \ge j \text{下三角} \\ \frac{j(j-1)}{2} + i, & i < j \text{上三角} \end{cases}$$

三角矩阵

仅上三角或下三角元素非 0, 其余为 0, 存储在一维 $B[n(n-1)/2+1]$

上三角

$a_{i,j}$ 在一维数组中的序号为:

$n+(n-1)+\cdots+(n-(i-1)+1)+(j-i+1)=\frac{(i-1)(2n-i+2)}{2}+j-i+1$

即前 i-1 行 + 第 i 行第 j 个 (从对角线 i 到 j)

0-index, B[k]:

$$k=\begin{cases}\frac{(i-1)(2n-i+2)}{2}+j-i, & i\le j\text{上三角}\\ \frac{n(n-1)}{2}, & i>j\text{下三角, 为 0}\end{cases}$$

1-index, B[k]:

$$k=\begin{cases}\frac{(i-1)(2n-i+2)}{2}+j-i+1, & i\le j\text{上三角}\\ \frac{n(n-1)}{2}+1, & i>j\text{下三角, 为 0}\end{cases}$$

下三角

$a_{i,j}$ 在一维数组中的序号为: $1+2+\cdots+i-1+j=\frac{i(i-1)}{2}+j$. 即前 i-1 行 + 第 i 行第 j 个

0-index, B[k]:

$$k=\begin{cases}\frac{n(n+1)}{2}, & i\le j\text{上三角}\\ \frac{i(i-1)}{2}+j-1, & i>j\text{下三角, 为 0}\end{cases}$$

1-index, B[k]:

$$k=\begin{cases}\frac{n(n-1)}{2}, & i\le j\text{上三角}\\ \frac{i(i-1)}{2}+j, & i>j\text{下三角, 为 0}\end{cases}$$

三对角矩阵 (带状矩阵)

对 n 阶矩阵任意元素 $a_{i,j}$, 若 $|i-j|>1,\ a_{i,j}=0$

$a_{i,j}$ 的序号: [前 i-1 行] + [i 行 j 号]: $[3(i-1)-1]+[j-i+2]=2i+j-2$

故: $a_{i,j}$, `b[[k]]`: 0-index: $k=2i+j-3$; 1-index: $k=2i+j-2$

从序号反求 i, j: 元素在 0-index 数组的**下标**为 k: $3(i-1)-1-1<3(i-1)-1\le k\le 3i-1-1<3i-1$

得: $i\le\frac{k+1}{3}+1\implies i=\lfloor\frac{k+1}{3}+1\rfloor$, 根据 $k=2i+j-3$ 或 $k=2i+j-2$ 求出 j.

稀疏矩阵

非零元素显著少于矩阵元素个数. 使用三元组或十字链表存储.

三元表: 存储非零元素个数, 矩阵行数和列数, 非零元素详情.

EX.

i	j	$a_{i,j}$
0	6	13

6 串 (string)

是由零个或多个字符组成的有限序列. 记为 $S =' a_1a_2 \dots a'_n$ $(n \geq 0)$.

串名: S; **串的长度**: n; **空串**: $n = 0$ 的串 (可表示为 $\emptyset$); **子串**: 串中任意多个连续的字符组成的子序列, 0 个字符也是子串; **主串**: 包含子串的串; **字符在串中的位置**: 字符在串中的序号; **子串在主串中的位置**: 子串第一个字符在主串中的序号; **空格串**: 由一个或多个空格组成的串, 空格亦为字符, 空格串不是空串; **模式串**: 模式匹配算法中, 要在主串中搜索的字符串

基本操作

```
StrAssign(&T, chars); // 赋值, chars赋值给T
StrCopy(&T, S); // S复制到T
StrEmpty(S); // 判断S是否为空串, 空则返回true, 否则返回false
StrCompare(S, T); // 比较. 返回S[i] - T[i].
  // S[i] T[i]指向第一个失配字符, 返回在ASCII或UTF-8等编码表中的编码的差值(字典序)
  // 当其中S[i], T[i]其中一个为空时, 较长的串大
StrLength(S); // 求串长
SubString(&Sub, S, pos, len); // 将S从pos开始长度为len的子串赋给Sub
Concat(&T, S1, S2); // 串联接, T = S1 + S2
Index(S, T); // 返回T在S中的位置, 无则返回0
ClearString(&S); // 将S清空为空串
DestroyString(&S); // 销毁S
```

其中, 串赋值 StrAssign, 串联接 Concat, 串比较 StrCompare, 求串长 StrLength, 求子串 SubString 为串的最小操作子集.

存储结构

有三种:

1. **定长顺序存储**: 此时记录串的长度有两类方式: (1) 新增结构体成员 int len 记录; (2) 空置字符数组 ch[0] 用以记录长度 len, 但是 char 占据 8 bit, 最大数值为 255; (3) 不显式记录, 在字符结尾使用空字符 \0作为结束标志, 结束标志不计入长度. 对于此类方式, 超出最大长度的字符会被舍去 (**截断**). (约定闲置 ch[0] 以统一序号和下标)

```
typedef struct {
  char ch[MAXSIZE];
  int len;
} SString;
```

2. **堆分配存储**: 动态分配空间存储串

```
typedef struct {
  char *ch;
  int len;
} SString;
```

3. **块链存储**: 链式存储, 每个结点有一个或这个多个字符. **块**: 每个结点; **块链结构**: 整个链表. 最后一个结点为存满的使用 # 填充. 若结点字符少, 存储密度低, 通过增加每个结点存储的字符数提高存储密度.

6.1 串的朴素匹配算法

朴素模式匹配是一种暴力算法. 从主串和模式串的第一个字符开始逐个匹配, 若发生失配, 主串指针跳转到本轮匹配起始字符的下一字符, 模式串指针跳转到首字符重新开始匹配. 假设主串长为 n, 模式串长为 m, 则最多在主串中匹配 n-m+1 个子串. 时间复杂度为 $O(mn)$, 但是在一般情况下, 实际时间复杂度近似 $O(m+n)$(每轮最终会失配的匹配一般只比较少量字符就会发生失配).

```
// 主串S, 子串T
int Index(std::string S, std::string T)
{
  int i{1}; int j {1};
  while (i <= S.length() && j <= T.length()){
    if (S[i-1] == T[j-1]) { // 继续比较下一字符
      i++; j++;
    }
    else {
      i = i - j + 2; // i - j + 1 + 1
      j = 1;
    }
  } // while
  if (j > T.length()) return i - T.length(); // (i + 1) - T.len
  return 0;
}
```

6.2 KMP 算法

KMP 仅在主串和子串有大量的部分匹配情况下, 才会快于普通算法. 时间复杂度为 $O(m+n)$. 主要优点是主串指针不会回溯.

KMP 通过计算 next 数组来控制主串中失配的字符下一次与模式串中哪一个字符匹配. 算法的设计思想是: 当主串中当前已经匹配相等的字符中, 存在后缀与模式串的前缀相同, 则可以将模式串向右滑动, 将已匹配的后缀与模式串前缀对齐, 以此减少需要匹配的字符. 主串的指针无需回溯, 模式串向右滑动的位数只与模式串本身有关.

对于第一位字符失配, 固定 `next[1] = 0`, 同时指针 ij 均自增; 对于第二位字符失配, 固定 `next[2] = 1`. 即: next 数组前两位固定 `next[1] = 0; next[2] = 1;`

6.2.1 手算 next 数组

考虑模式串 abaabcaba. 每次在失配字符之前画一条竖线. 此处使用 ()标出匹配的前后缀, 有匹配则当前 next 为前缀之后第一个字符的序号, 无匹配前后缀则当前 next 为 1

```
01        01 1       011   2      0112   2     01122   3    011223 1   0112231   2
abaabcaba ab\aabcaba ab(a)\abcaba aba(a)\bcaba aba(ab)\caba abaabc\aba abaabc(a)\ba
abaabcaba ab\aabcaba (a)ba\abcaba (a)baa\bcaba (ab)aab\caba abaabc\aba (a)baabca\ba
01122312   3
abaabc(ab)\a
(ab)aabcab\a
```

i	1	2	3	4	5	6	7	8	9
模式串	a	b	a	a	b	c	a	b	a
$next[i]$	0	1	1	2	2	3	1	2	3

6.2.2 算法实现

对于 next 数组. 当计算 next[i] 时, 有两种情况: (1) T[i-1] 与 T[next[i-1]] 相同; (2) T[i-1] 与 T[next[i-1]] 不同:

(1) T[i-1] 与 T[next[i-1]] 相同: 则 T[i] 应该与 T[next[i-1]] 的下一位比较, 因此有: `next[i] = next[i-1] + 1`

(2) T[i-1] 与 T[next[i-1]] 不同: 考虑到, next[i] 的意义是: 当 S[j] 与 T[i] 失配时, S[j] 应该与 T[next[i]] 重新开始比较. 因此, T[i-1] 与 T[next[i-1]] 不同, 即 T[i-1] 与 T[next[i-1]] 失配, 因此, T[i-1] 应该与 T[next[next[i-1]]] 重新开始比较. 由此可以类推: T[i-1] 最终需要与 T[next …next[i-1]] 比较, 此时, 两者应该相同或者后者已经等于 0.

```
// 计算next
void get_next(std::string S, int next[])
{
  int i {1}; int j {0};
  next[1] = 0;
  while (i < S.length()){
    if (j == 0 || S[i-1] == S[j-1]){ // S[i-1] == S[next[i-1]]
      next[i+1] = j + 1; // next[i+1] = next[i] + 1
      i++; j++;
    }
    else j = next[j]; // S[i-1] != S[next[i-1]]; S[next[next[i-1]]]
  }
}
```

```cpp
// 主串S, 子串T
int KMP_Index(std::string S, std::string T)
{
   int i {1}; int j {1};
   int next[T.length()+1];
   next[0] = -1;
   get_next(T, next);

   while (i <= S.length() && j <= T.length()){
      if (j == 0 || S[i-1] == T[j-1]){ // 比较下一位
         i++; j++;
      }
      else j = next[j];
   } // while
   if (j > T.length()) return i - T.length(); // (i + 1) - T.len
   return 0;
}
```

6.3 KMP 算法的优化

对于 next 数组, 假设 S[i] 与 T[j] 失配后, S[i] 与 T[next[j]] 匹配. 当 `T[j] == T[next[j]]`时, 这一步匹配必然失配. 因此, KMP 的优化思想是: 当 `T[j] == T[next[j]`时, 递归地修正 `next[j] = next[next[j]]`, 直至 `next[j] == 0` 或者 `T[j] != T[next[j]]`, 再执行 S[i] 与 T[next[j]] 的匹配. 因此, 在开始匹配前, 计算出依据该思想修正的 nextval 数组代替 next 数组进行匹配.

6.3.1 手算 nextval 数组

首先算出 next 数组, 固定 $nextval[1] = 0$, 从第 2 位开始算起.

若 S[i] 与 S[next[i]] 不等, 则 `S[nextval[i]] = S[next[i]]`

若 S[i] 与 S[next[i]] 相等, 则 `S[nextval[i] = S[nextval[nextval[i]]]`. 即令`j = next[i]`, 递归地执行`j = next[j]]`直至`next[j] = 0`或者`S[i] != S[j]`

EX. 串 abaabcaba

```
next    = 0, 1, 1, 2, 2, 3, 1, 2, 3
nextval = 0, 1, 0, 2, 1, 3, 0, 1, 0
```

6.3.2 算法实现

优化的 KMP 算法与 KMP 算法仅在 next 数组和 nextval 数组计算上有差异, 比较逻辑无异.

```
// 计算nextval
void get_nextval(std::string S, int nextval[])
{
  int i {1}; int j {0};
  nextval[1] = 0;
  while (i < S.length()){
    if (j == 0 || S[i-1] == S[j-1]) { // S[i-1] == S[nextval[i-1]]
      i++; j++;
      if (S[i-1] != S[j-1]) nextval[i] = j; // S[i] != S[nextval[i]]
      else nextval[i] = nextval[j]; // S[i] == S[nextval[i]], recursive
    } // if
    else j = nextval[j]; // S[i-1] != S[nextval[i-1]]; S[nextval[nextval[i-1]]]
  } // while
}

// 主串S, 子串T
int KMP_val_Index(std::string S, std::string T)
{
  int i {1}; int j {1};
  int nextval[T.length()+1];
  nextval[0] = -1;
  get_nextval(T, nextval);

  while (i <= S.length() && j <= T.length()){
    if (j == 0 || S[i-1] == T[j-1]) { // 比较下一位
      i++; j++;
    }
    else j = nextval[j];
  } // while
  if (j > T.length()) return i - T.length();
  return 0;
}
```

7 树

树: n 个结点的有限集; **空树**: n=0; 任何一个非空树有且仅有一个**根**.
树的定义是 **递归**的; 树的根节点没有前驱, 其余结点有且仅有一个前驱; 树中所有结点都可以有零个或多个后继.
树用于表示具有**层次结构**的数据.

基本术语
祖先: 结点 K 的祖先是从根节点到 K 路径上所有其他结点; **子孙**: A 是 B 的祖先, B 是 A 的子孙;
双亲: 全部祖先中最接近结点的祖先; **孩子**: A 是 B 的祖先, B 是 A 的孩子; **兄弟**: 具有相同双亲的结点;
堂兄弟: 双亲在同一层的结点 (一般用于指代有几个结点在同一层)
结点的层次: 根节点是第 1 层, 其孩子是第 2 层, 从上往下数以此类推;
结点的深度: 结点所在的层次; **树的高度 (深度)**: 树中结点的最大层数;
结点的高度: 以该节点为根节点的子树的高度 (从下往上数); **结点的度**: 一个结点的孩子个数;
树的度: 树中结点的最大度数 **分支节点 (非终端结点)**: 度大于 0 的结点;
叶子结点 (终端结点): 度为 0 的结点 **有序树**: 树中各个结点的各个子树从左到右有次序, 不能互换;
无序树: 树中结点的各子树可以互换左右次序
路径: 由两个结点之间经过的结点序列 (EX. A 到 B 的路径 A->C->D->B). 只能从上到下, 有向边.
路径长度: 路径上所经过的边的个数;
树的路径长度: 分为 **内部路径长度 IPL**: 所有结点的深度之和; 和 **外部路径长度**: 所有叶子结点的深度之和

数值特征:

度为 m 的树与 m 叉树
度为 m 的树: 任意结点的度 $\leq m$, **至少一个**结点的度为 m
m 叉树: 任意结点的度 $\leq m$, 允许所有结点的度 $< m$, 可以是空树

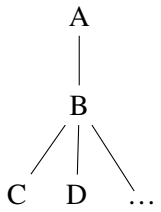
树的结点数 n = 所有结点的度数之和 $+1$

度为 m 的树, 第 i 层上最多有 m^{i-1} $(i \geq 1)$ 个结点. 第一层 1 个, 第 2 层 m 个, 第 3 层 m^2 个

高度为 h 的 m 叉树, 至多有 $\frac{m^h-1}{m-1}$ 个结点. $m^0 + m^1 + \cdots + m^h = \frac{1-m^h}{1-m}$

度为 m, 具有 n 个结点的树的最小高度为 $\lceil \log_m [n(m-1)+1] \rceil$. 前 h-1 层最大, 共 $\frac{m^{h-1}-1}{m-1}$
因此 $\frac{m^{h-1}-1}{m-1} < n \leq \frac{m^h-1}{m-1}$, $h - 1 < \log_m [n(m-1)+1] \leq h$

度为 m, 共有 n 个结点的最大高度 h 为 $n - m + 1$



前 $h-1$ 层每层一个, 最后一层 m 个 (度为 m).
$(h-1) + m = n$

7.1 二叉树

每个结点中至多有两课子树 (不存在度 >2 的结点), 子树有左右之分不可颠倒 (有序树).

二叉树与度为 2 的树的区别: (a) 二叉树可以**为空**, 度为 2 至少有 3 个结点, 其中一个度为 2; (b) 二叉树的左右次序确定, 不可更改; 度为 2 的树中, 左右次序是相对于另一个孩子而言的, 若只有一个结点, 则不需要区分左右次序

特殊二叉树

1. **满二叉树**: 高度为 h, 具有 2^h-1 个结点的树. (a) 只有**度为 0 和度为 2** 的结点; (b) 仅最后一层有叶节点; (c) 每层都含有最多的结点; 叶节点集中在最下一层

满二叉树层序编号: 按照从上到下从左到右从 1 开始编号: 对于编号为 i 的结点, 双亲为 $\lfloor i/2 \rfloor$, 左孩子 $2i$, 有孩子 $2i+1$ (若有).

2. **完全二叉树**: 高度为 h, 有 n 个结点, 每个结点都与高度为 h 的满二叉树中编号为 $1 \sim n$ 的结点一一对应. 可以视为从满二叉树删去若干最底层最靠右的一些连续叶结点. (a) 只有最后 **2 层**有叶结点; (b) 最多只有一个**度为 1** 的结点; (c) 孩子双亲的编号关系同满二叉树; (d) 编号 $i \le \lfloor n/2 \rfloor$ 的是分支节点, $i > \lfloor n/2 \rfloor$ 为叶子结点

完全二叉树推算不同度的结点数: 记有 n 个结点的二叉树中, 度为 0, 1, 2 的结点分别有 n_0, n_1, n_2. 则 $n_0+n_1+n_2 = n_1+2n_2+1, \implies n_0 = n_2+1$

假设二叉树有 n 个结点:

(1) n 为奇数, $n=2k-1$, 有 $n=(n_0+n_2)+n_1=(2n_2+1)+n_1, \implies n_1=0, n_2=k-1, n_0=k$

(2) n 为偶数, $n=2k$, $n=(2n_2+1)+n_1 \implies n_1=1, n_2=k-1, n_0=k$

3. **二叉排序树**: 对任意一个结点, 其左子树上所有结点的关键字小于自身关键字, 其右子树上所有结点的关键字大于自身关键字. 其左右子树亦是一颗二叉排序树

4. **平衡二叉树**: 树中任意一个结点的左右子树的高度差的绝对值不超过 1

5. **正则二叉树**: 只有度为 0 或 2 的结点

数值性质

1. $n_0 = n_2 + 1$

2. 非空二叉树第 k 层最多有 2^{k-1} 个结点 $(k \ge 1)$

3. 高度为 h 的二叉树最多有 2^h-1 个结点 $(h \ge 1)$

4. 具有 $n(n>0)$ 个结点的**完全二叉树**, 高度为 $\lfloor \log_2 n \rfloor + 1$ 或 $\lceil \log_2(n+1) \rceil$

5. 对**完全二叉树**从 **1** 开始, 从上到下, 从左到右开始编号. 对于编号为 i 的结点:

(a) 双亲 $\lfloor i/2 \rfloor$, 左孩子 $2i$, 右孩子 $2i+1$

(b) 最后一个分支节点 $\lfloor n/2 \rfloor$, 若 $i < \lfloor n/2 \rfloor$ 为分支结点, 否则为叶结点

(c) 若有度为 1 的结点, 只可能是最后一个分支结点 $\lfloor n/2 \rfloor$, 且该结点只有左孩子没有右孩子

(d) 层序编号后, 若某节点为叶结点或只有左孩子, 则编号大于该结点的结点均为叶结点

(e) 结点 i 所在层次是 $\lfloor \log_2 i \rfloor + 1$ 或者 $\lceil \log_2(i+1) \rceil$. ($2^{h-1}-1+1 \le i \le 2^h-1 < 2^h$ 或 $2^{h-1}-1 < i \le 2^h-1$)

6. 对**完全二叉树**从 **0** 开始, 从上到下, 从左到右开始编号. 对于编号为 i 的结点: (编号为 i 的结点, 对应从 1 开始编号是编号为 $i+1$ 的结点, 代换)

(a) 双亲 $\lfloor \frac{i-1}{2} \rfloor$, 左孩子 $2i+1$, 右孩子 $2i+2$

(b) 最后一个分支节点 $\lfloor n/2 \rfloor - 1$, 若 $i < \lfloor n/2 \rfloor - 1$ 为分支结点, 否则为叶结点

(c) 若有度为 1 的结点, 只可能是最后一个分支结点 $\lfloor n/2 \rfloor - 1$, 且该结点只有左孩子没有右孩子

(d) 层序编号后, 若某节点为叶结点或只有左孩子, 则编号大于该结点的结点均为叶结点

(e) 结点 i 所在层次是 $\lfloor \log_2(i+1) \rfloor + 1$ 或者 $\lceil \log_2(i+2) \rceil$.

7.1.1 存储结构

顺序存储

对于**完全二叉树**或者**满二叉树**, 适合使用**顺序存储**. 树中结点的序号可以唯一地表示结点之间的逻辑关系.

顺序存储结构中, 需要在数组元素结构体中新增 bool 成员表示该结点是否为空. 对于任意一棵二叉树, 按照顺序存储时, 需要对照相同高度的完全二叉树在没有结点的地方添加空结点, 会造成存储空间浪费. 最坏情况下, n 个结点的单支树, 需要占用 $2^n - 1$ 个存储空间.

判断是否有孩子: 计算在若为完全二叉树的情况下, 该结点所在位置的编号 i 以及孩子编号, 再利用 bool 类型成员查询孩子编号的数组元素是否为空元素

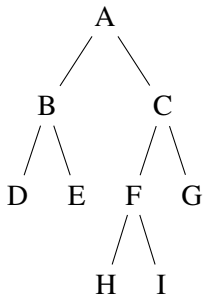
链式存储

一般采用链式存储. 一般分为不带双亲指针的二叉链表和带双亲指针 (指向双亲) 的三叉链表. 对于 n 个结点的链表, 有 $2n$ 个指针域, 只有 $n-1$ 个结点被指向, 因此有 $n+1$ 个空链域 (用于组成线索二叉树)

```
typedef struct BiTNode {
  type data;          // 数据域
  struct BiTNode *lchild, *rchild; // 左右孩子指针
} BiTNode, *BiTree;
```

7.1.2 二叉树的遍历

二叉树遍历是寻找一种方式使得每个结点都被访问一次而且只被访问一次. **根 (N), 左孩子 (L), 有孩子 (R)**. 常见的遍历有: 先序遍历 (先根遍历), 中序遍历 (中根遍历), 后序遍历 (后根遍历), 层序遍历. 先中后序遍历使用递归实现, 层序遍历借助队列实现.



先序遍历 NLR: A B D E C F H I G
中序遍历 LNR: D B E A H F I C G
后序遍历 LRN: D E B H I F G C A
层序遍历: A B C D E F G H I

每个结点均访问一次且只访问一次, **时间复杂度**: $T(n) = O(n)$; **空间复杂度**: 空间复杂度与使用的递归工作栈的深度有关, $S(n) = O(h+1) = O(h)$, h 为树的深度, 在**最坏情况下**, 树为单支树, 结点数为 n, 树深度为 n, 此时 $S(n) = O(n)$

```cpp
// 先序遍历
void PreOrder(BiTree T)
{
  if (!T) return ; // empty node
  visit(T);
  PreOrder(T->lchild);
  PreOrder(T->rchild);
}

// 中序遍历
void InOrder(BiTree T)
{
  if (!T) return ; // empty node
  InOrder(T->lchild);
  visit(T);
  InOrder(T->rchild);
}

// 后序遍历
void PostOrder(BiTree T)
{
  if (!T) return ; // empty node
  PostOrder(T->lchild);
  PostOrder(T->rchild);
  visit(T);
}

// 层序遍历
void LayerOrder(BiTree T)
{
  std::queue<BiTNode*> q;
  q.push(T);
  while (!q.empty()){
    if (q.front()->lchild) q.push(q.front()->lchild);
    if (q.front()->rchild) q.push(q.front()->rchild);
    visit(q.front());
    q.pop();
  }
}
```

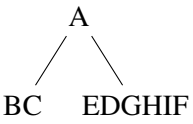
7.1.3 由遍历序列重建二叉树

只给出四种遍历序列中的任意一种都不能唯一确定二叉树; 给出**中序遍历序列**和其余任意一种遍历序列, 可以唯一确定一颗二叉树. 关键在于交替使用两个对立确定根节点跟左右子树的结点, 递归逐层重建.

先序和中序

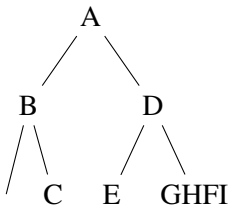
先序: ABCDEFGHI; 中序: BCAEDGHFI

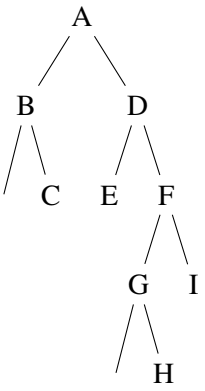
1. root: A, left: BC, right: EDGHFI



2. LEFT: root: B, right: C

RIGHT: root: D, left: E, right: GHFI

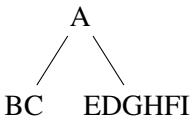


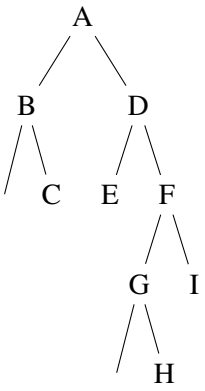


后序和中序

后序: CBEHGIFDA, 中序: BCAEDGHFI

1. root: A, left: BC, right: EDGHFI

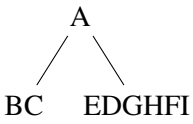


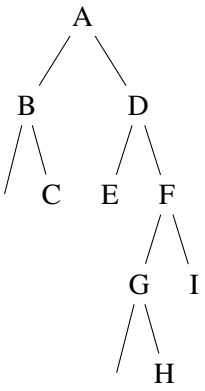


层序和中序

层序: ABDCEFGIH, 中序: BCAEDGHFI

1. root: A, left: BC, right: EDGHFI





7.2 线索二叉树

结点结构

lchild	ltag	data	rtag	rchild

约定 tag 域的数据: $tag = \begin{cases} 0, \text{指向该结点的孩子} \\ 1, \text{指向该结点的前驱或后继} \end{cases}$

左孩子指针作为线索指向前驱, 有孩子指针作为线索指向后继

```
typedef struct ThreadNode { // 线索链表
  type data;           // 数据域
  struct ThreadNode *lchild, *rchild; // 左右孩子指针
  int ltag, rtag;      // 标志位
} ThreadNode, *ThreadTree;
```

线索链表: 含有左右线索标志位的二叉链表; **线索**: 指向结点前驱和后继的指针; **线索二叉树**: 加上线索的二叉树

7.2.1 二叉树线索化

线索化的特别处理

先序线索化: 需要预先检查线索标志位避免循环递归. 当前结点的左右孩子的前驱为当前结点, 在递归线索化中, 如果不预先判断左右孩子指针指向真正孩子还是前驱后继, 当孩子指向前驱时, 会进入死循环.

中序线索化: 置空遍历过程最后一个结点的右孩子指针. 最后一个结点必然无右结点, 需要结束线索化后置空.

后序线索化: 无序特别处理. 最后一个遍历的结点是根节点.

```
// 中序线索化
void InThread(ThreadTree &p, ThreadTree &pre)
{
  InThread(p->lchild, pre); // 递归线索化左子树
  if (!p->lchild) { // 无左孩子, 指向前驱
    p->lchild = pre;
    p->ltag = 1;
  }
  if (pre && !pre->rchild) { // 前驱右孩子为空, 指向当前结点
    pre->rchild = p;
    pre->rtag = 1;
  }
  pre = p; // 更新上一个访问结点指针
  InThread(p->rchild, pre); // 递归线索化右子树
}

// 中序线索化主过程
void CreateInThread(ThreadTree &T)
{
  ThreadNode *pre = nullptr;
  if (T) {
    InThread(T, pre);
    pre->rchild = nullptr; // 最后一个结点必无右孩子, 特别处理
    pre->rtag = 1;
  }
}
```

```cpp
// 先序线索化
void PreThread(ThreadTree &p, ThreadTree &pre)
{
  if (!p->lchild) { // 无左孩子, 指向前驱
    p->lchild = pre;
    p->ltag = 1;
  }
  if (pre && !pre->rchild) { // 前驱右孩子为空, 指向当前
    pre->rchild = p;
    pre->rtag = 1;
  }
  pre = p; // 更新指针
  if (p->ltag == 0) PreThread(p->lchild, pre); // 检查线索标志位
  if (p->rtag == 0) PreThread(p->rchild, pre); // 检查线索标志位
}

// 先序线索化主过程
void CreatePreThread(ThreadTree &T)
{
  ThreadNode *pre = nullptr;
  if (T) {
    PreThread(T, pre);
    pre->rchild = nullptr; // 最后一个结点必无右孩子, 特别处理
    pre->rtag = 1;
  }
}
```

```cpp
// 后序线索化
void PostThread(ThreadTree &p, ThreadTree &pre)
{
  PostThread(p->lchild, pre);
  PostThread(p->rchild, pre);
  if (!p->lchild) { // 无左孩子, 指向前驱
    p->lchild = pre;
    p->ltag = 1;
  }
  if (pre && !pre->rchild) { // 前驱无右孩子, 指向当前
    pre->rchild = p;
    pre->rtag = 1;
  }
  pre = p;
}

// 后序线索化主过程
void CreatePOstThread(ThreadTree &T)
{
  ThreadNode *pre = nullptr;
  if (T) {
    PostThread(T, pre); // 后序遍历最后一个结点是根节点, 不用特别处理
  }
}
```

带头结点的线索二叉树: 额外定义一个头结点. 其前驱指向树的根节点, 其后继指向最后一个遍历的结点; 最后一个遍历的结点的右孩子指向头结点, 第一个遍历的结点的左孩子指向头结点. 带头结点的线索二叉树类似于双向循环链表, 允许从树中任一结点开始, 向后或者向前遍历整棵树.

7.2.2 非线索二叉树的遍历

中序遍历

后继: 右子树的最左下结点

前驱: 左子树的最右下结点

先序遍历

后继: (a) 有左孩子: 左孩子; (b) 无左孩子: 右孩子

前驱: 需要先找到双亲结点 p.

(a) 当前结点为左孩子: p;

(b) 当前为右孩子: (b1) p 无左孩子: p ;(b2) p 有左孩子: p 左子树的最右下结点

后序遍历

后继: 需要先找到双亲结点 p.

(a) 当前是 p 的左孩子: (a1) 若 p 有右孩子: p 右子树的最左下结点; (a2) p 无右孩子: p

(b) 当前是 p 的右孩子: p

前驱: (a) 有右孩子: 右孩子; (b) 无右孩子: 左孩子

7.2.3 线索二叉树的遍历

先序和后序需要寻找双亲结点, 一般使用带双亲指针的三叉链表实现, 在二叉链表中实现复杂, 不予考虑.

```
// 中序序列第一个结点
ThreadNode* InFirstNode(ThreadTree p)
{
   if (!p) return nullptr;
   // 寻找最左下结点
   while (p->ltag == 0) p = p->lchild;
   return p;
}

// 中序后继
ThreadNode* InNextNode(ThreadNode *p)
{
   if (!p) return nullptr;
   if (p->rtag == 0) return InFirstNode(p->rchild);
   return p->rchild;
}
```

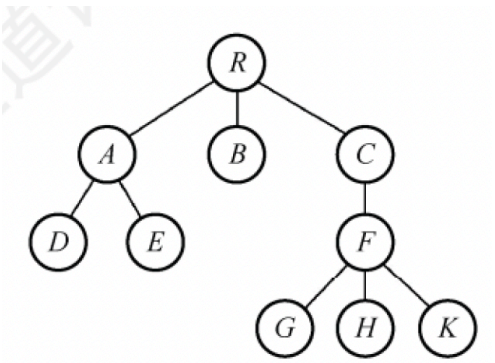
```
// 中序前驱
ThreadNode* InPrevNode(ThreadNode *p)
{
   if (!p) return nullptr;
   if (p->ltag == 1) return p->lchild;
   p = p->lchild; // 左子树的最右下结点
   while (p->rtag == 0) p = p->rchild;
   return p;
}

// 中序遍历
void InOrder(ThreadTree T)
{
   ThreadNode *p = InFirstNode(T);
   while (p) {
      visit(p);
      p = InNextNode(p);
   }
}
```

7.3 树与森林

7.3.1 树的存储结构

树能够使用顺序存储结构和链式存储结构, 但必须能够唯一地表示出树中各结点的逻辑关系 (由于一棵树的结点数量不确定, 只存储数据不存储逻辑关系, 不能够连接各结点)



(a) 一棵树

双亲表示法

顺序存储, 使用一组连续空间来存储每个结点, 在每个结点中设置一个**伪指针**, 指向**其双亲结点在数组中的位置**. 根节点的伪指针域为**-1**.

在**存储森林**时, 将每棵树根节点的 parent 设为-1 即可.

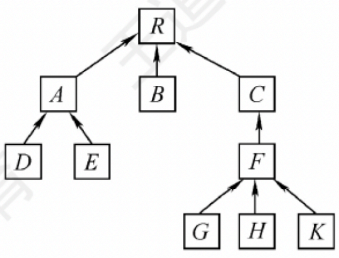
优点: 找双亲快; 缺点: 找孩子不方便, 只能遍历整棵树. 适用于寻找双亲多, 寻找孩子少的情景.

```
typedef struct {
  type data; // 数据元素
  int parant; // 伪指针, 指向双亲
} PTNode;

typedef struct {
  PTNode nodes[MAX_TREE_SIZE]; // 存储结点
  int n;            // 结点数
} PTree;
```

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

(b) 双亲表示



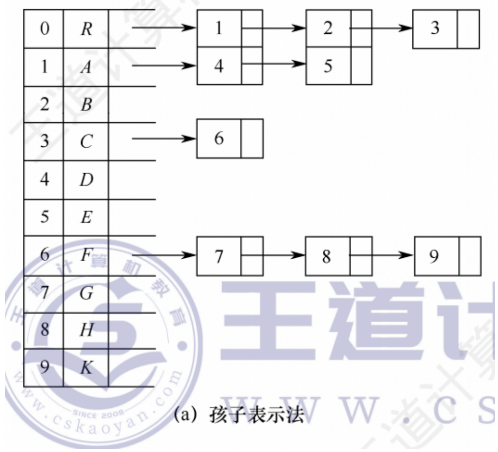
(c) 双亲指针图示

孩子表示法

混合使用**顺序存储和链式存储**. 每个结点均存入线性表, 每个结点的直接后继以单链表的形式存在双亲结点之后. n 个结点有 n 个孩子链表, 叶结点的孩子链表为空表. n 个头指针组成一个单链表.

存储森林时, 需要在将结构体中封装记录全部根节点位置的成员.

优缺点与双亲相反: 优点: 寻找孩子非常方便; 缺点: 寻找双亲需要遍历 n 个结点中孩子链表指向的 n 个孩子链表结点. 适用于寻找孩子多, 寻找双亲少的场景.

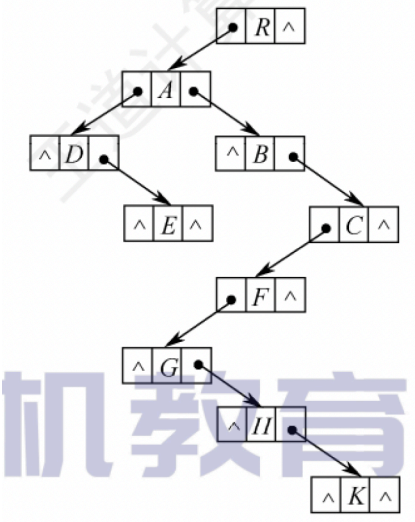


(a) 孩子表示法

孩子兄弟表示法 (二叉树表示法)

以**二叉链表**作为树的存储结构. **结点组成: 结点值, 指向结点第一个孩子的指针, 指向结点下一个兄弟结点的指针**.

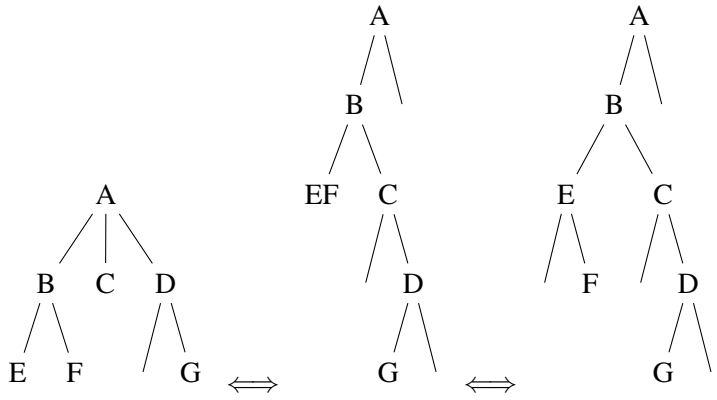
优点: 可以方便实现树转换为二叉树的操作, 易于查找节点的孩子等; 缺点: 从当前结点查找双亲结点比较麻烦. 若增设指向双亲结点的指针 parent, 查找双亲结点也会很方便.



(b) 孩子兄弟表示法

7.3.2 树转换为二叉树

左孩子, 右兄弟. 每个结点左指针指向第一个孩子, 右指针指向下一个兄弟. 转换时可以逐层处理以防遗漏.



7.3.3 森林与二叉树互相转换

森林转换为二叉树

(1) 森林中每棵树转换为二叉树

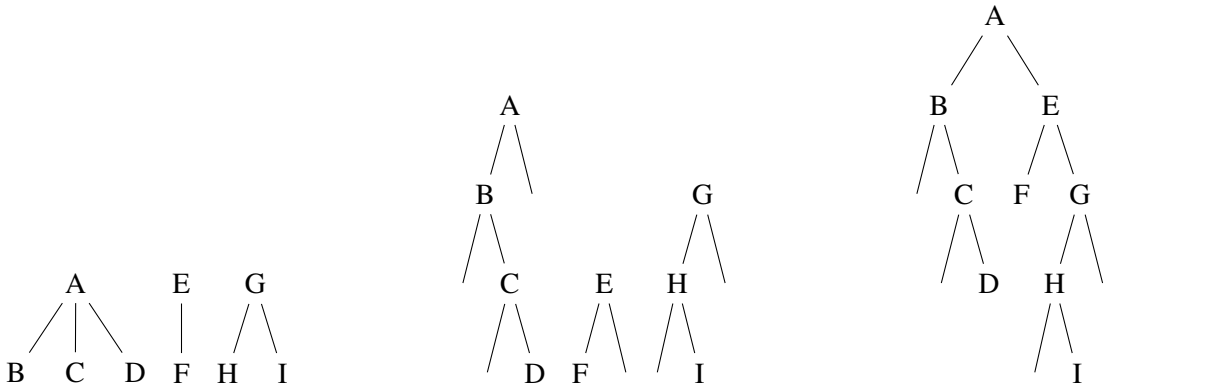
(2) 每棵树的根视为兄弟关系, 合并二叉树

二叉树转换为森林

(1) 将根节点和根节点左子树单独取出, 断开右指针. 所取出的部分为第一颗树

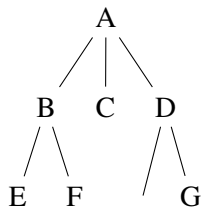
(2) 对剩余部分重复步骤 (1), 直到只剩下一颗不存在右子树的二叉树

(3) 将分离出的二叉树转换为树



(a) 森林　　**(b)** 各自转换为二叉树　　**(c)** 合并为二叉树

7.3.4 树的遍历



先根遍历, 遍历序列与**二叉树的先序遍历**序列相同

(1) 访问根节点;

(2) 依照顺序访问每课子树, 子树访问顺序依然为先根后子树

遍历序列: ABEFCDG

后根遍历, 遍历序列与**二叉树的中序遍历**序列相同

LNR, 全部子树在根节点的左子树; 在根节点的左孩子中, 本身与左子树为第一颗子树, 其第一个右孩子及其左子树为第二颗子树, 其第二个右孩子及其左子树为第三棵树 …)

(1) 依次遍历每棵子树, 遍历时遵循先子树后根结点的顺序;

(2) 访问根节点

遍历序列: EFBCGDA

层次遍历: 按照层序依次访问各结点

遍历序列: ABCDEFG

7.3.5 森林的遍历

先序遍历森林, 等同于对每棵树依次进行先根遍历, 或对应二叉树的先序遍历

(1) 访问森林中第一棵树的根节点;

(2) 先序访问第一棵树中根节点的子树森林;

(3) 先序访问除第一棵树外剩余树构成的森林

遍历序列: ABCDEFGHI

中序遍历森林, 等同于对每棵树依次进行后根遍历, 或对应二叉树的中序遍历

(1) 中序遍历第一棵树根节点的子树森林;

(2) 访问第一棵树的根节点;

(3) 中序遍历除第一棵树外剩余树构成的子树森林

遍历序列: BCDAFEHIG

树	森林	对应二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

7.4 哈夫曼树 (最优二叉树) 与哈夫曼编码

结点的权: 树中结点被赋予的有意义的一个值;
结点的带权路径长度: 从树的根到该结点的 路径长度 与 结点的权 的乘积
树的带权路径长度 (WPL): 树中所有 叶结点 的 带权路径长度 之和

哈夫曼树 (最优二叉树): 对于拥有 n 个带权叶结点的二叉树中, WPL 最小的二叉树称为哈夫曼树.

对于相同的 n 个带权叶结点, **哈夫曼树不唯一, 但是 WPL 一定相同**. 哈夫曼树对负权重结点依然适用

构造哈夫曼树:

不断在森林中取出当前权值最小的两棵树, 将其结点权重相加作为新树根节点的权重, 所选取的两棵树分别作为新树的左右子树, 将新树加入森林, 直至森林只剩下一棵树:
(1) 将 n 个结点作为 n 棵只含有一个结点的二叉树, 构成森林 F
(2) 构造一个新结点, 从 F 选择权值最小的两棵树分别作为新节点的左右子树; 令新结点的权值为两棵子树根节点权值之和
(3) 在 F 中删除所选择的两棵树, 并将新树加入到 F
(4) 重复 (2)(3), 直至 F 中只剩下一棵树

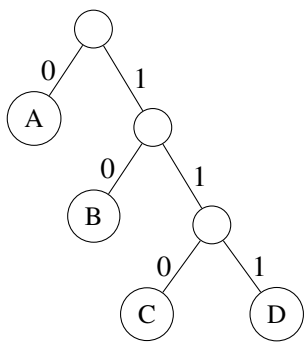
哈夫曼树特性:
1. 权值越小的结点到根节点的路径长度越大
2. 每个最初的结点最终都成为叶结点
3. 每次都选择两棵子树, 因此, 新建了 $n-1$ 个结点, 结点总数为 $2n-1$; 且不存在度为 1 的结点, 结点只为 0 或 2

哈夫曼编码

哈夫曼编码属于**可变长度编码**, 即对不同字符使用不等长的二进制编码表示. 该编码对频率高的字符赋予短编码, 对频率低的字符赋予长编码. 可以减少字符平均编码, 以达到压缩数据的目的. 此外, **固定长度编码**: 每个字符用相等的二进制编码表示. 若使用定长编码, 要求所有字符结点在同一层.

此外, 哈夫曼编码属于**前缀编码**, 即没有一个编码是另一个编码的前缀. 若要确定字符集是否为前缀编码, 即寻找是否存在不符合前缀编码规则的编码: 在同一棵树上操作, 对每一个字符序列, 0 则移动到左孩子, 1 则移动到右孩子, 不存在现成结点则创建, 当结束本序列遍历时, 若停留在分支结点或解码的路径在此前已存在, 则为非前缀编码.

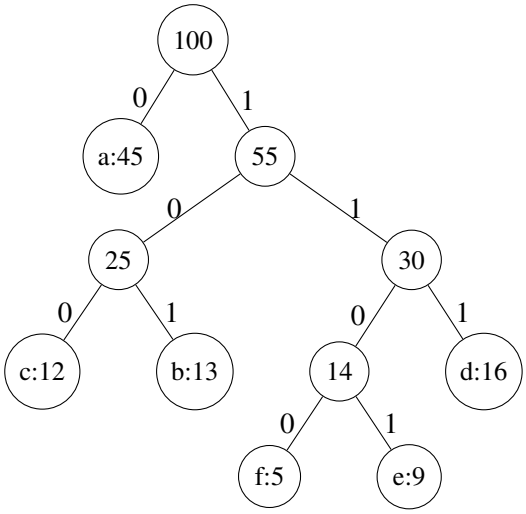
构造哈夫曼编码时, 将字符出现频次作为结点的权值, 依次构造哈夫曼树, 并约定边的值为左 0 右 1(左 1 右 0 也可以); 解码时, 按照编码序列寻找对应结点.



编码: A(0), B(10), C(110), D(111)

计算压缩率

假设存在字符集, 字符(出现频次)如下: a(45), b(13), c(12), d(16), e(9), f(5). 所构造的哈夫曼树如下:



哈夫曼编码: a(0), b(101), c(100), d(111), e(1101), f(1100)

定长编码: a(000), b(001), c(010), d(011), e(100), f(101)

哈夫曼编码需要的存储空间, 即 $WPL = 1 \times 45 + 3 \times (12 + 13 + 16) + 4 \times (5 + 9) = 224$

定长编码所需存储空间: $3 * (45 + 13 + 12 + 16 + 9 + 5) = 300$

压缩率: $\frac{300-224}{300} = 25\%$

8 图

基本定义

数学定义: 图 G 由顶点 V 和边 E 组成, 记为 $G=(V,E)$. $V(G)$ 表示 G 中顶点的非空子集, $E(G)$ 表示 G 的边. $V=\{v_1,v_2,\dots\}$, $E=\{(u,v)|u\in V,v\in V\}$. **图的顶点数 (图的阶数)**: $|V|$, **图的边数**: $|E|$

图不可以是空图, 图**至少有一个结点**. 顶点集 V 一定非空, 边集**可以为空**

弧: 有向边, 顶点的有序对, 记为 $<v,w>$, $v\to w$, **弧尾**: v, **弧头**: w. 称: $<v,w>$ 为 v 到 w 的弧, 或 v 邻接**到** w

边: 无向边. 记为 (v,w) 或者 (w,v). v,w 互为 **邻接点**, 称: 边 (v,w) 依附于 w 和 v, 或者边 (v,w) 与 v,w 相关联

简单图 (存在有向无向之分): 1. 不存在重复边; 2. 不允许顶点到自身的边

复杂图: 存在重复边或者存在顶点到自身的边

顶点的度:

无向图: 顶点 v 的度是依附于 v 的边的数量, 记为 $TD(v)$. **无向图**全部顶点的度之和为边数的 2 倍 (一条边连接两个顶点), 即 $\sum_{i=1}^{n}TD(v_i)=2|E|$

有向图: 区分**入度**, **出度**, **度**.

入度: 以 v 为终点的边的数量, 记作 $ID(v)$. **出度**: 以 v 为起点的边的数量, 记作 $OD(v)$.

度: 入度与出度之和, 记作 $TD(v)=ID(v)+OD(v)$. 有向图中: $\sum_{i=1}^{n}ID(v_i)=\sum_{i=1}^{n}OD(v_i)=|E|$

路径: 顶点 v_p 到 v_q 之间的序列 $v_p,v_1,v_2,\dots,v_q$

路径长度: 路径上边的数目

回路 (环): 首末顶点相同的路径. 若一个图有 n 个顶点, 有多于 $n-1$ 条边, 则该图**一定有环**

简单路径: 顶点不重复出现的路径

简单回路: 除了首末顶点之外, 其他顶点不重复出现的回路

距离 (区分有向无向): 从 u **到** v 的**最短**路径长度. 若 u,v 之间不存在路径, 则距离为无穷 ∞

子图: 若 $G=(V,E),G'=(V',E')$, V' 是 V 的子集, E' 是 E 的子集, 则 G' 是 G 的子图. 若要构成子图, 则 E' 涉及的顶点必须全在 V' 中, 否则不构成图, 因此, 不是 VE 的任何子集都能构成 G 的子图

生成图: G' 是 G 的子图, 且 $V(G')=V(G)$, 即子图包含原图的全部顶点

连通: **无向图中**, 顶点 v 和 w 之间存在**路径**.

连通图: 无向图中任意两个顶点都是连通的; **非连通图**: 不是连通图的无向图. **极大连通子图 (连通分量)**: 1. 连通图; 2. 包含尽可能多的顶点和边 (**不能被任何另外一个连通子图所包含**)

对于有 n 个顶点的无向图, 若为**连通图**, **最少有** n-1 条边 (一个顶点连接所有); 若保证为**非连通图**, **最多有** $\binom{n-1}{2}$ 条边 (n-1 个顶点两两相连, 孤立一个顶点)

强连通: **有向图**中, 对于顶点 v, w, 从 v 到 w 和从 w 到 v 的路径都存在 (不一定有 $<v,w>$ 或 $<w,v>$), 则称这两个顶点强连通

强连通图: 有向图中, 任意两个顶点都是强连通的

强连通分量: 1. 强连通图; 2. 包含尽可能多的顶点和边. 若一个图是强连通图, **最少有** n 条边 (构成环)

生成树: **连通图**中包含图中全部顶点的一个**极小连通子图**. 1. 包含全部顶点; 2. 边尽可能少. n 个顶点的图, 其生成树只能有 n-1 条边. 生成树可能有多个

生成森林: **非连通图**中, 连通分量的生成树构成非连通图的生成森林

边的权值: 每条边都可以标注的有意义的数值

网 (带权图): 边有权值的图

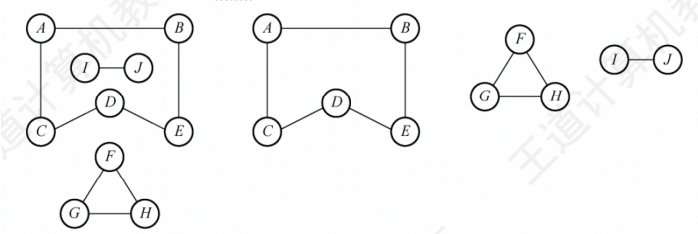
带权路径长度: 路径上所有边的权值之和

完全图 (简单完全图): 对于 n 个顶点的**无向图**, 有 $\binom{n}{2} = \frac{n(n-1)}{2}$ 条边的图 (任意两个顶点之间有边); 对于 n 个顶点的**有向图**, 有 $2\binom{n}{2} = n(n-1)$ 条边的图 (任意两个顶点之间有方向相反的两条边). n 个顶点, 无向图, 边的条数 $\in [0, \binom{n}{2}]$; 有向图, 边的条数 $[0, n(n-1)]$

稠密图: 边很多的图; **稀疏图**: 边很少的图. 一般, 满足 $|E| < |V| \log_2 |V|$ 可视为稀疏图 (没有绝对的判断标准)

有向树: 一个顶点的入度为 0, 其余顶点的入度均为 1 的有向图 (不是强连通图)

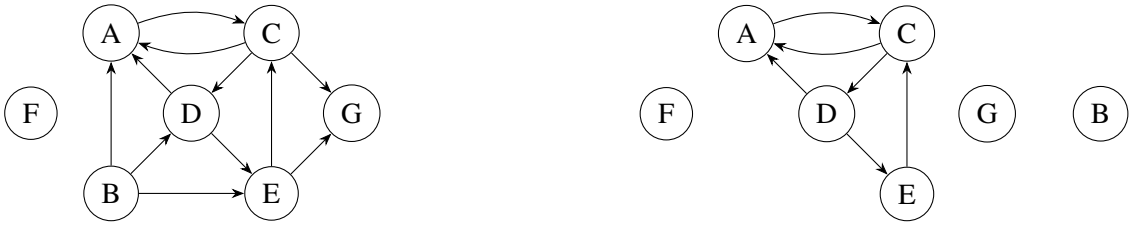
树: 1. 无向图; 2. 不存在回路; 3. 连通;



(a) 无向图G_4 (b) G_4的三个连通分量

找强连通分量

1. 分离出孤立顶点
2. 分离出没有出度的顶点和与其相连的边
3. 分离出没有入度的顶点和与其相连的边



(a) 有向图 **(b)** 4 个强连通分量

8.1 图的存储

8.1.1 邻接矩阵法

顺序存储. 用一个一维数组存储图中顶点的信息, 用一个二维数组存储图中边的信息, 其中, 二维数组称为**邻接矩阵**. 在简单应用中, 可以忽略一维数组, 不存储顶点信息. 邻接矩阵的空间复杂度为 $O(|V|^2)$. **稠密图**适合使用邻接矩阵图中, 顶点自身与自身不认为有边连接. 记有 n 个顶点的图之顶点为 $v_1, v_2, \ldots, v_n$

非带权图

$$A[i][j] = \begin{cases} 1, < v_i, v_j > 或(v_i, v_j)是图中的边 \\ 0, < v_i, v_j > 或(v_i, v_j)不是图中的边 \end{cases}$$

带权图

$$A[i][j] = \begin{cases} w_{ij}, < v_i, v_j > 或(v_i, v_j)是图中的边 \\ 0或\infty, < v_i, v_j > 或(v_i, v_j)不是图中的边 \end{cases}$$

注意 i, j 的先后关系在有向图中不可以对换. **有向图**中 `A[i][j]`是边 `i->j`. **无向图**中, 邻接矩阵是一个**唯一的**对称矩阵 (上下三角压缩存储)

1. ∞ 可以定义为 `MAXINT`; 2. 部分情况下, 自身与自身 (对角线): 0, 无边: ∞

数据结构

```
struct MGgraph {
  VertexType vex[MaxVertexNum]; // 顶点表
  EdgeType edge[MaxVertexNum][MaxVertexNum]; // 邻接矩阵(边表)
  int vexnum, arcnum;         // 图当前的顶点数和边数
};
```

邻接矩阵有关计算

1. **无向图**: 顶点的度 $TD(v_i)$ 为矩阵中第 i 行或者第 i 列非零 (∞) 元素的个数

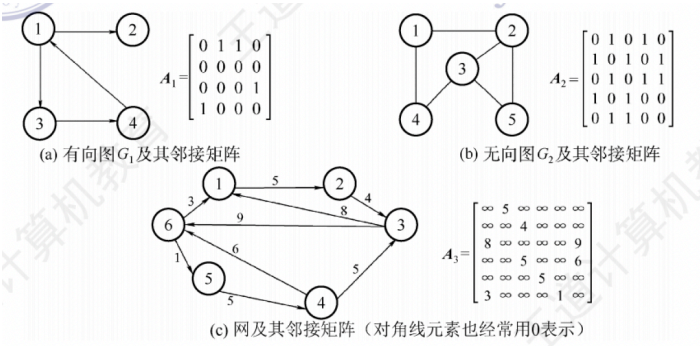
2. **有向图**: 顶点 v_i 的**入度** $ID(v_i)$ 为第 i **列**非零 (∞) 元素的个数; **出度** $OD(v_i)$ 为第 i **行**非零 (∞) 元素的个数

3. 设图 G 的领接矩阵为 A, 则 $A^n[i][j]$ 表示有顶点 i 到顶点 j, 长度为 n 的路径的条数

EX. $A^2[1][4] = a_{11}a_{14} + a_{12}a_{24} + a_{13}a_{34} + a_{14}a_{44}$, 一个因子为 1, 则 `i->k->j`路径存在

邻接矩阵容易确定任意两个顶点之间是否有边连接, 但是确定图中有几条边, 需要按行和按列进行遍历每一个元素. $T(n) = O(|V|^2)$.

邻接矩阵计算顶点的度 $T(n) = O(|V|)$



(a) 有向图G_1及其邻接矩阵

(b) 无向图G_2及其邻接矩阵

(c) 网及其邻接矩阵（对角线元素也经常用0表示）

8.1.2 邻接表法

顺序存储 + 链式存储. 当存储稀疏图时, 可以减少大量空间浪费. 对每个顶点建立一个单链表, 称为**边表** (有向图中称为**出边表**), 边表表示依附于顶点 v_i 的边或者以 v_i 为尾的弧 (有向图). 顶点的数据信息和边表的头指针使用**顺序存储**.

若为**无向图**, $S(n) = O(|V| + 2|E|)$(每条边出现两次); **有向图**: $S(n) = O(|V| + |E|)$

```
// 边表结点
struct ArcNode
{
  int adjvex;     // 该边指向的顶点
  struct ArcNode *nextarc; // 下一条边
};
// 顶点表结点
struct VNode
{
  VertexType data; // 顶点信息
  ArcNode *firstarc; // 指向第一条边或者弧的指针
};
// 邻接表
struct ALGraph
{
  VNode vertices[MaxVertexNum]; // 边表
  int vexnum, arcnum; // 顶点数和边数
};
```

计算度

1. **无向图**: 遍历顶点 v_i 的边表
2. **有向图**: **出度**: 遍历顶点 v_i 的出边表; **入度**: 遍历全部顶点的出边表

注 1. 有向图找出边方便, 找入边麻烦. 无向图找边方便. 若要确定两个顶点中是否右边, 需要在一个顶点的边表中找另一个顶点, 效率低

2. 图的邻接表不唯一, 各边结点的链接次序任意

8.1.3 十字链表

十字链表用于存储**有向图**, **顺序 + 链式存储**. 每条弧用弧结点存, 每个顶点用顶点节点存, 顶点结点为顺序存储. 十字链表**不唯一**. $S(n) = O(|V| + |E|)$. 从任意顶点出发, 能够快速遍历出边和全部入边, 求度方便.

弧结点

tailvex	headvex	hlink	tlink	(info)

顶点结点

data	firstin	firstout

tailvex: 弧尾顶点编号

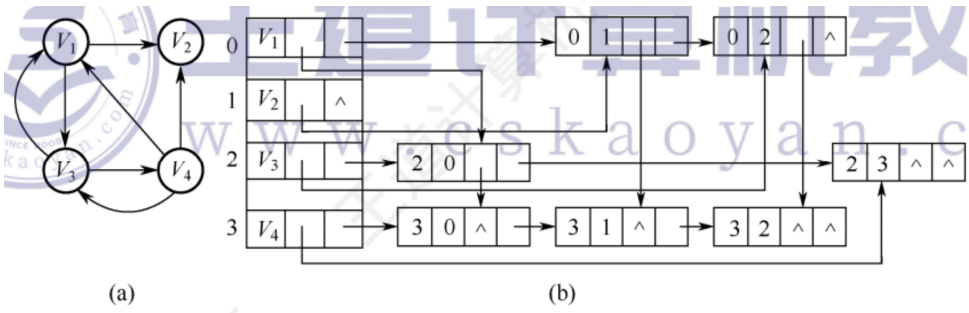
headvex: 弧头顶点编号

hlink: 相同弧头的下一条弧 (顶点的下一条入边)

tlink: 相同弧尾的下一条弧 (顶点的下一条出边)

firstin: 顶点的第一条入边 (以该顶点为弧头的弧)

firstout: 顶点的第一条出边 (以该顶点为弧尾的弧)



画图或编写算法时, 可以先把出边全部处理完, 再以此链接每个结点的入边表.

8.1.4 邻接多重表

邻接多重表用于存储**无向图, 顺序 + 链式存储**. 边存储于边结点中, 顶点存储于顶点结点中 (顺序). 容易求得顶点和边的各种信息, 但是求两个顶点之间是否存在边和删除边时, 需要分别从两个结点开始遍历边表, 效率低.

由于每条边依附于两个顶点, 因此, 每个边结点同时链接在两个顶点的链表中. 无向图的邻接多重表和邻接表的唯一区别: **邻接表**中同一条边有分属于两个链表中的两个结点, **邻接多重表中**只有一个结点

边结点

ivex	ilink	jvex	jlink	(info)

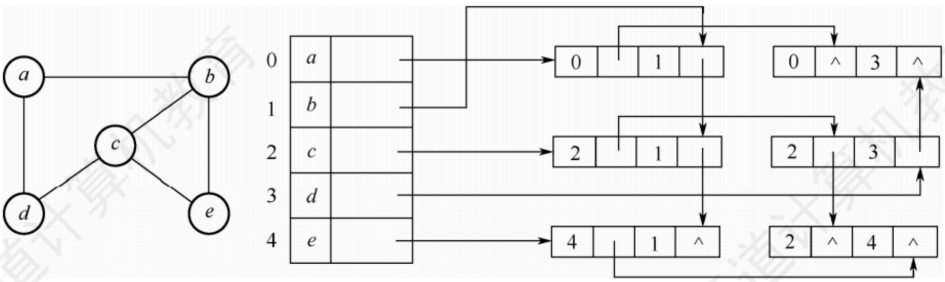
顶点结点

data	firstedge

ivex, jvex: 顶点 i, j 的编号

ilink, jlink: 依附于顶点 i, j 的下一条边

firstedge: 依附于该顶点的第一条边



画图或编写算法时, 可以每个顶点依次新增或链接完全衣服于该结点的边. 注意结点每个成员的意义, 链接边的时候注意链接到成员 ilink 还是 jlink.

图存储方式总结

	邻接矩阵	邻接表	十字链表	邻接多重表
空间复杂度	$O(\|V\|^2)$	无向图: $O(\|V\|+2\|E\|)$ 有向图: $O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$
找相邻边	遍历行/列, $T(n)=O(\|V\|)$	有向图找出边遍历结点出边表 找入度遍历整个邻接表	方便	方便
删除边或顶点	删除边方便 删除顶点需要移动大量元素	无向图中删除边或者顶点都要遍历两个顶点的边表, 不方便	方便	方便
适用	稠密图	都行	有向图	无向图
表示方式	唯一	不唯一	不唯一	不唯一

8.2 图的基本操作

Adjacent (G, x, y): 判断图 G 是否存在边 <x, y> 或 (x, y)。

邻接矩阵: $O(1)$; 邻接表: 最好 $O(1)$, 最坏 $O(|V|-1)=O(|V|)$

Neighbors (G, x): 列出图 G 中与结点 x 邻接的边。

邻接矩阵: $O(|V|)$; 邻接表: $O(1)\sim O(|V|)$. 其中有向图, 出边 $O(1)\sim O(|V|)$, 入边: $O(|E|)$

InsertVertex (G,x): 在图 G 中插入顶点 x 。

$T(n)=O(1)$

DeleteVertex (G, x): 从图 G 中删除顶点 x 。

邻接矩阵: $O(|V|)$(顶点 x 的行列全部置零, 再标记顶点已删除); 邻接表: $O(1)\sim O(|E|)$.

AddEdge (G, x, y): 若无向边 (x, y) 或有向边 <x, y> 不存在，则向图 G 中添加该边。

$T(n)=O(1)$

RemoveEdge (G, x, y): 若无向边 (x, y) 或有向边 <x, y> 存在，则从图 G 中删除该边。

邻接矩阵: $O(|1|)$; 邻接表: $O(1)\sim O(|V|)$.

FirstNeighbor (G, x): 求图 G 中顶点 x 的第一个邻接点，若有则返回顶点号。若 x 没有邻接点或图中不存在 x, 则返回 -1 。

邻接矩阵: $O(1)\sim O(|V|)$; 邻接表: 无向图: $O(1)$, 有向图: 出边 $O(1)$, 入边 $O(1)\sim O(|E|)$.

NextNeighbor(G,x, y): 假设图 G 中顶点 y 是顶点 x 的一个邻接点，返回除 y 外顶点 x 的下一个邻接点的顶点号，若 y 是 x 的最后一个邻接点，则返回 -1 。

邻接矩阵: $O(|1|)\sim O(|V|)$; 邻接表: $O(1)$.

Get_edge_value (G, x, y): 获取图 G 中边 (x, y) 或 <x, y> 对应的权值。

时间在于找边, 复杂度同 Adjacent

Set_edge_value (G,x, y ,v): 设置图 G 中边 (x, y) 或 <x, y> 对应的权值为 v 。

时间在于找边, 复杂度同 Adjacent

8.3 图的遍历

图在遍历过程中, 需要一个辅助数组 `bool visited[numVertives]`来记录顶点是否已经访问过.

8.3.1 广度优先搜索 BFS

类似于树的层序遍历. 对于每个结点, 先访问, 标记为已访问后, 入队; 结点出队时, 访问其全部邻接的未访问结点. BFS 需要借助辅助队列实现.

```
1  bool visited[MaxVertexNum];
2  SqQueue Q;
3  void BFSTraverse(Graph G)
4
5    int i;
6    for (i=0; i<G.vexnum; i++) // 初始化辅助数组
7      visited[i] = false;
8    InitQueue(Q);
9    for (i=0; i<G.vexnum; i++) // 对每个连通分量都调用一次
10     if (!visited[i])
11       BFS(G, i);
12 }
```

邻接表实现

```
void BFS (ALGraph G, int i)
{
  int v, w; // 工作结点
  ArcNode* p; // 用于访问邻接点
  // 访问初始结点
  visit(G, i);
  visited[i] = true;
  EnQueue(Q, i);
  while (!QueueEmpty(Q)) {
    DeQueue(Q, v); // 队首结点
    for (p=G.vertices[v].firstarc; p; p=p->nextarc) {
        // 遍历全部邻接点
      w = p->adjvex;
      if (!visited[w]) { // 未被访问
        visit(G, w);
        visited[w] = true;
        EnQueue(Q, w);
      }
    }
  }
}
```

邻接矩阵实现

```
void BFS(MGgraph G, int i)
{
  int v, w; // 工作节点
  // 访问初始结点
  visit(G, i);
  visited[i] = true;
  EnQueue(Q, i);
  while (!QueueEmpty(Q)) {
    DeQueue(Q, v); // 队首结点
    for (w=0; w<G.vexnum; w++) { // 遍历全部邻接点
      if (!visited[w] && G.edge[v][w] == 1) { //
          未被访问且<v, w>or(v, w)有边
        visit(G, w);
        visited[w] = true;
        EnQueue(Q, w);
      }
    }
  }
}
```

单源最短路径

对于**非带权图和权重均相等的图**, 可以使用 BFS 搜索单源最短路径. 对于带权图, 只能使用 Floyd 和 Dijkstra.

```
int d[MaxVertexNum]; // 记录源点到该点的最短路径长度
int path[MaxVertexNum]; // 记录该点在最短路径上的直接前驱
void BFS_Min_Path(Graph G, int u){
  int w; // 工作结点
  for (int i=0; i<G.vexnum; i++) // 初始化路径长度为无穷
    d[i] = INT32_MAX;
  visited[u] = true;
  d[u] = 0;
  EnQueue(Q, u);
  while (!QueueEmpty(Q)) {
    DeQueue(Q, u);
    for (w=FirstNeighbor(G, u); w>0; w=NextNeighbor(G, u, w)) {
      if (!visited[w]) { // 未被访问
        visited[w] = true;
        EnQueue(Q, w);
        d[w] = d[u] + 1; // 在上一层长度基础上+1
        path[w] = u; // 记录w的前驱
      }
    }
  }

}
```

性能分析

空间复杂度: 需要借助辅助队列, 最坏情况下 $|V|-1$ 个结点在第二次入队时入队. $S(n)=O(|V|)$
时间复杂度:
邻接矩阵: $T(n)=O(|V|+|V|^2)=O(|V|^2)$ (遍历顶点 + 遍历边表);
邻接表: 有向: $T(n)=O(|V|+|E|)$, 无向: $T(n)=O(|V|+2|E|)$

广度优先生成树

将遍历过程分层, 可以得到遍历树, 称为广度优先生成树. 该遍历树是指定顶点为根的树中, 高度最小的生成树. 同时, 层次也反映了从源点出发到各个顶点的最短距离. 对于同一个图, 邻接矩阵的表示是唯一的, 广度优先生成树也是唯一的; 邻接表不唯一, 其广度优先生成树亦不唯一. 对于非 (强) 连通图, 遍历时会产生广度优先生成森林.

8.3.2 深度优先搜索 DFS

DFS 是一个递归算法, 需要使用递归工作栈.

```
void DFSTraverse (Graph G)
{
  int i;
  for (i=0; i<G.vexnum; i++) // 初始化辅助数组
    visited[i] = false;
  for (i=0; i<G.vexnum; i++) // 对每个连通分量都调用一次
    if (!visited[i])
      DFS(G, i);
}
```

邻接表实现

```
void DFS(ALGraph G, int i)
{
  ArcNode* p; // 工作结点
  int j;
  visit(G, i);
  visited[i] = true;
  for (p=G.vertices[i].firstarc; p; p=p->nextarc) { // 访问全部邻接点
    j = p->adjvex;
    if (!visited[j]) // 未被访问
      DFS(G, j);
  }
}
```

邻接矩阵实现

```
void DFS (MGgraph G, int i)
{
  int j; // 工作顶点
  visit(G, i);
  visited[i] = true;
  for(j=0; j<G.vexnum; j++) { // 遍历各邻接点
    if (!visited[j] && G.edge[i][j] == 1) // 未被访问且有边
      DFS(G, j);
  }
}
```

性能分析

空间复杂度: 需要使用递归工作栈, $S(n) = O(|V|)$
时间复杂度:
邻接矩阵: $T(n) = O(|V|^2)$
邻接表: 有向图: $T(n) = O(|V| + |E|)$; 无向图: $T(n) = O(|V| + 2|E|)$

深度优先生成树

(强) 连通图会产生深度优先生成树; (强) 非连通图会生成深度优先生成森林.

8.3.3 图的遍历与连通性

Traverse 过程中, BFS/DFS 的调用次数 = 图的连通分量个数

对于**无向图**: 只需要一次遍历就能访问全部, 则是连通图; 否则为非连通图, 并且每次遍历能够访问该连通分量的全部顶点

对于有向图: 由于分为强连通分量和非强连通分量, 在非强连通分量中, 一次遍历不一定能访问该子图的全部顶点.