数据结构

UNIkeEN，2023.1

> 版权声明：以下所有内容大部分来自张同珍老师的PPT，内容选择参考2022-2023-1总结课提及考试范围。（整理的时候也在速通，若有错误实属合理现象）

CH2 线性表

线性结构：线性结构的数据元素之间形成有序序列。邻接关系一对一，即每个结点至多有一个直接前驱，至多有一个直接后继。所有结点按一对一的邻接关系构成的整体就是线性结构。

线性表：一种仅由元素的相互位置确定它们之间相互关系的线性结构。

线性表的**规模或长度**是指线性表中元素的个数。特别地：当元素的个数为零时，该线性表称为**空表**。

2.1 顺序实现

线性表的顺序实现

```
template <class elemType>
class seqList{
    private:
        elemType *elem; // 顺序表存储数组，存放实际的数据元素。
        int len;        // 顺序表中的元素个数，亦称表的长度。
        int maxSize;    // 顺序表的的最大可能的长度。
        void doubleSpace(); //私有函数，做内部工具
    public:
        seqList(int size=INITSIZE); //初始化顺序表
        //表为空返回true,否则返回false。
        bool isEmpty() const { return (len == 0); }
        //表为满返回true，否则返回false。
        bool isFull() const { return (len == maxSize); }
        int length() const { return len; }  //表的长度，即实际存储元素的个数。
        elemType get(int i) const { return elem[i]; } //返回第i个元素的值
        //返回值等于e的元素的序号，无则返回-1.
        int find (const elemType &e) const;
        //在第i个位置上插入新的元素（值为e），使原来的第i个元素成为第i+1个元素。
        void insert(int i, const elemType &e);
        // 若第i个元素存在，删除并将其值放入e指向的空间。
        void remove(int i, elemType &e );
        void clear() { len=0; }; //清除顺序表，使得其成为空表
        ~seqList() { delete []elem; }; //释放表占用的动态数组
};
```

- 构造函数

```
template <class elemType>
seqList<elemType>::seqList(int size)
//初始化顺序表
{
    elem = new elemType[size];//申请动态数组
    if (!elem) throw illegalSize(); //连续空间，判定申请是否成功
    maxSize = size;
    len = 0;
}
```

- doubleSpace函数

```
template <class elemType>
void seqList<elemType>::doubleSpace()
{   int i;
    elemType *tmp = new elemType[2*maxSize];
    if (!tmp) throw illegalSize();

    for (i=0; i<len; i++)
        tmp[i] = elem[i];

    delete []elem;   elem = tmp;
    maxSize = 2*maxSize;
}
```

- 查找函数，时间复杂度 O(n)

```
template <class elemType>
int seqList<elemType>::find (const elemType &e)const
// 返回值等于e的元素的序号，无则返回-1
{
    int i;
    elem[0] = e;  //哨兵位先置为待查元素
    for (i=len-1; i>=0; i--)
        if (elem[i]==e) break;
    return i;
}
```

- 插入函数，时间复杂度 O(n)

```
template <class elemType>
void seqList<elemType>::insert (int i, const elemType &e)
{
    if (len==maxSize) doubleSpace(); //空间满了，无法插入元素
    for (int k=len; k>i; k--)
        elem[k]=elem[k-1];
    elem[i]=e;
    len++;
}
```

- 删除函数

```
template <class elemType> //注意五步口诀法
void seqList<elemType>::remove (int i, elemType &e )
{
    for (int k=i; k<len-1; k++)
        elem[k]=elem[k+1];
    len--;
}
```

- 常见错误

1. 混淆len和maxSize的含义，前者是实际元素的个数，后者是存储空间的大小，也是最多能存多少元素的限制。
2. seqList对象作为函数形参，直接用对象形式而不用引用形式, 这样即浪费空间又引起seqList类的复制构造函数的执行，降低运行效率。

3. insert函数实现中忘记检查位置i的合理性、忘了检查表中是否有空间可以支持插入一个新的元素等，造成算法不完整。

2.2 链接实现

顺序表插入、删除时间代价的分析，可以看出其时间复杂度是线性阶的，而且会引起大量已存储元素的位置移动。

改进方法：采用链式结构

2.2.1 单链表

- 头指针head指向了头结点。头结点并不是线性表中的一部分，它的指针字段next给出了首结点的地址。（引入头结点，是为考虑删除链表第一个结点/在表头插入一个结点的特殊情况）
- 线性表中最后一个结点的指针字段next的值为NULL。
- 提供一个单链表只需要给出头结点的地址即头指针，顺着头指针head，可以很方便地逐个访问单链表中的所有结点。
- 它的任何一个结点包含了一个存储元素**数据值**的字段和一个存储该结点的直接**后继结点地址**的指针字段。

类定义

```
template <class elemType>
class linkList; //类的前向说明

template <class elemType>
class node
{   friend class linkList<elemType>;
     private:
        elemType data;
        node *next;
     public:
        node():next(NULL){};
        node(const elemType &e, node *N=NULL)
        {  data = e; next = N; };
};

template <class elemType>
class linkList
{   private:
        node<elemType> *head;
    public:
        linkList();  //构造函数，建立一个空表
        bool isEmpty ()const; //表为空返回true,否则返回false。
        bool isFull ()const {return false;}; //表为满返回true,否则返回false。
        int length ()const;  //表的长度
        elemType get(int i)const;//返回第i个元素的值
        //返回值等于e的元素的序号，从第1个开始，无则返回0.
        int find (const elemType &e )const;
        //在第i个位置上插入新的元素（值为e）。
        void insert (int i, const elemType &e );
        //若第i个元素存在，删除并将其值放入e指向的空间。
        void remove (int i, elemType &e);
        void reverse(); //元素就地逆置
        void clear (); //清空表，使其为空表
        ~linkList();
};
```

基本操作

```
template <class elemType>
linkList<elemType>::linkList() //构造函数，建立一个空表
{
    head = new node<elemType>();
}

template <class elemType>
bool linkList<elemType>::isEmpty ()const //表为空返回true,否则返回false。
{
    if (head->next==NULL) return true;
    return false;
}
```

插入函数

当P已经指向了插入位置的前一个结点时，插入操作和结点个数无关，时间复杂度为O(1)。

```
template <class elemType>
void linkList<elemType>::insert (int i, const elemType &e )
//在第i个位置上插入新的元素（值为e）。
{
    if (i<1) return;//参数i越界
    int j=0; node<elemType>  *p=head;

    while (p&&j<i-1)
    {j++;  p=p->next;}

    if (!p) return; //参数i越界
    p->next = new node<elemType>(e, p->next); //一句话插入，填空题可能会考
}
```

头部插入法：

```
head->next = new node(x, head->next);
```

删除函数

当P已经指向了待删除结点的前一个结点时，删除操作和结点个数无关，时间复杂度为O(1)。

删除函数与插入相似，主要代码如下：

```
node *q=p->next;
p->next = q->next;
delete q;
```

查找操作

- 找值为x的结点，顺首结点逐个向后检查、匹配。单链表和顺序表中，时间复杂度都是O(n)。
- 找第k个结点，顺序表O(1) ，链表O(n)。

找值为x的结点：

```
template <class elemType>
int linkList<elemType>::find (const elemType &e )const
//返回值等于e的元素的序号，从第1个开始，无则返回0.
{   int i=1;
    node<elemType> *p = head->next;

    while (p)
    {   if (p->data==e) break;
        i++;   p=p->next;
    }
    if (p) return i;
    return 0;
}
```

找第k个结点：

```
template <class elemType>
elemType linkList<elemType>::get(int i )const
//返回第i个元素的值,首元素为第1个元素
{
    if (i<1) throw outOfBound();
    int j=1;
    node<elemType> *p = head->next;
    while (p&&j<i) {p=p->next; j++;}
    if (p) return p->data;
}
```

清空表

```
template <class elemType>    //P、Q兄弟协同法
void linkList<elemType>::clear () //清空表，使其为空表
{
    node<elemType> *p,*q;
    p=head->next;
    head->next=NULL;
    while (p)
    {   q=p->next;
        delete p;
        p=q;
    }
}
```

- 常见错误
  - 指针p未被初始化或者为空，读取其指向的字段如p->data，如循环检查p所指的结点中值是否x，可用 `p=head->next; while (p && p->data!=x) p=p->next。`
  - 插入时，p首先指向了首结点p=head->next ，应该指向头结点p=head 。
  - p原本指向了一个结点，但其指向的结点空间已经释放，仍要读取其所指结点的字段。如 `p=head; delete p; p=p->next; p->next` 非法访问了不能访问的内存空间。

反转操作▲

> 2014-2015-1，T2-4 将单链表除头结点之外的其余结点逆序连接

```
void reverse(){
 node *p, *q;
 p = head -> next;
 if (!p) return;
 head -> next = NULL;
 while (p){
     q = p;
     p = p->next;
     q -> next = head -> next;
     head -> next = q;
 }
}
```

2.2.2 单循环链表

末结点的next不再指向空，而是指向头结点。从表中任何一个结点出发，都可以顺next指针访问到所有结点。

为了循环方便，**不带头结点的单循环链表居多，head直接指向首结点。**

- 典型应用：约瑟夫环

2.2.3 双向链表，双循环链表

每个结点有prior和next两个指针，分别指向直接前驱和直接后继结点。

双向链表可以引入头结点和尾结点两个不存放数据的空结点，双向循环链表没有。

CH3 栈和队列

3.1 栈

- 如果元素到达线性结构的时间越晚，离开的时间就越早，这种线性结构称为**栈（Stack）或堆栈**；
- 因为元素之间的关系是由到达、离开的时间决定的，因此栈通常被称为时间有序表。而到达和离开的含义就是插入和删除操作，因此栈可以看作是插入和删除操作位置受限的线性表。

> LIFO，Last In First Out

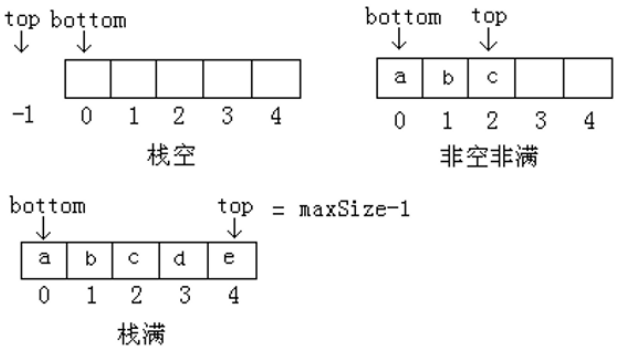
基本名词：

- 栈的首部（元素最早到达的部分）称为**栈底（bottom）**，栈结构的尾部（元素最晚到达的部分）称为**栈顶（top）**
- 为了保证栈的先进后出或后进先出的特点，元素的插入和删除操作都必须在栈顶进行。元素从栈顶删除的行为，称为**出栈或者弹栈**操作（pop）；元素在栈顶位置插入的行为，称为**进栈或者压栈**操作（push）。
- 注：pop操作一般会返回被pop的值。相当于删除并返回stack.top

3.1.1 顺序栈（数组模拟栈）

栈的顺序存储即使用连续的空间存储栈中的元素。可以将栈底放在数组的0下标位置，进栈和出栈总是在栈的同一端（栈顶）进行，顺序方式存储的栈称为**顺序栈。**

> 注意，数组模拟栈同样有最长长度与类似doubleSpace增广空间的函数



bottom总是0，不需要记忆
栈空标志：top=-1
栈满标志：top=maxSize-1

函数initialize(seqStack)、isEmpty、isFull、top、pop、destroy（~seqStack）的时间复杂度均为O(1)。

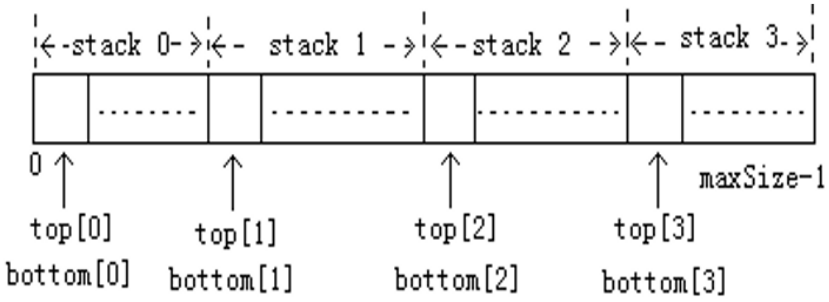
push因某时可能扩大空间，造成O(n)时间消耗，但按照分期"付款式"法，分摊到单次的插入操作，时间复杂度仍为O(1)。——**均摊法**

共享栈

在实际应用中，有时需要同时使用多个数据类型相同的栈。栈中的元素个数因进栈、出栈操作动态地变化，所有栈不一定同时达到栈满，有时一些栈满而另一些栈尚余空间。为了提高空间使用率，可以在同一块连续的空间中设置多个栈。多个栈间共享空间，这些栈称为**"共享栈"**。

共享栈的特点是每个栈拥有一个连续的小空间, 所有共享栈拥有一个大的连续空间。

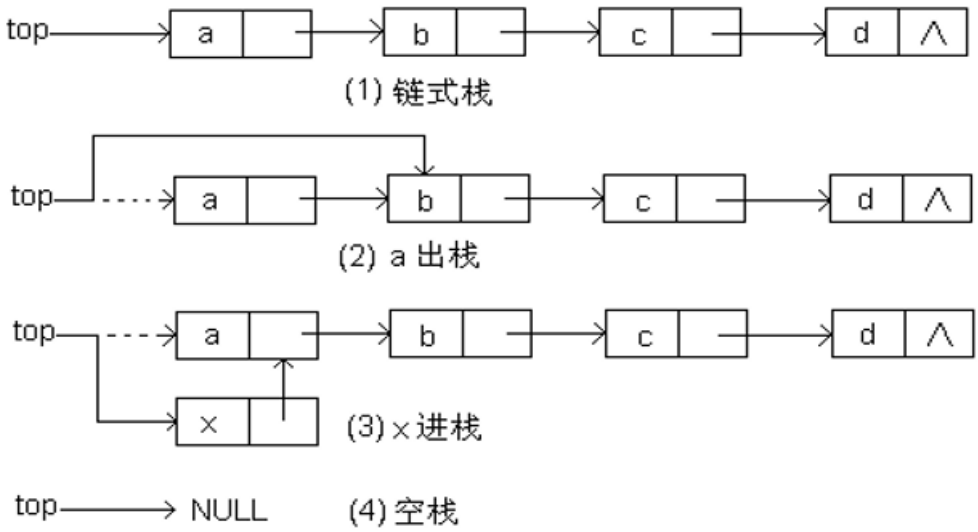
top指向实际栈顶的后一个位置



假设有m个栈，第i个栈空的条件：top[i]=bottom[i];
第i个栈栈满条件为：当i<m-1时，top[i]=bottom[i+1];
当i=m-1时，top[i]=maxSize。

3.1.2 链式栈

- 用不连续的空间和附加指针来存储元素及元素间的关系。
- 栈顶指针top指向处于栈顶的结点，即单链表中的首结点。



析构函数将栈中的所有结点清除，空间回收，时间复杂度为O(n)。

构造函数、isEmpty、isFull、top、push、pop的时间复杂度均为O(1)。

```
class outOfBound{}; //异常类

template <class elemType>
class linkStack;
template <class elemType>
class Node
{  friend class linkStack<elemType>;
    private:
        elemType data;
        Node *next;
    public:
        Node(){next = NULL;}
        Node(const elemType &x, Node *p=NULL)
        { data = x; next = p; }
};

template <class elemType>
class linkStack
{
    private:
        Node<elemType> *Top;
    public:
        linkStack(){ Top = NULL; }; //初始化栈，使其为空栈
        bool isEmpty(){ return (Top==NULL); }; //栈为空返回true,否则返回false。
        elemType top();
        void push(const elemType &e);
        void pop();
        ~linkStack();
};

template <class elemType>
elemType linkStack<elemType>::top()
{
    if (!Top) throw outOfBound();//栈空
    return Top->data;
}
```

```
template <class elemType>
void linkStack<elemType>::push(const elemType &e)
{  Top = new Node<elemType>(e, Top);  }

template <class elemType>
void linkStack<elemType>::pop()
{
    Node<elemType> *tmp;
    if (!Top) throw outOfBound();//栈空
    tmp = Top; //用tmp记住原栈顶结点空间，用于弹栈后的空间释放
    Top = Top->next; //实际将栈顶结点弹出栈
    delete tmp;//释放原栈顶结点空间
}

template <class elemType>
linkStack<elemType>::~linkStack()
{
    Node<elemType> *tmp;
    while (Top)
    {
        tmp = Top;
        Top=Top->next;
        delete tmp;
    }
}
```

3.1.3 栈的应用

括号匹配

1. 首先创建一个以字符为数据类型的栈。
2. 从源程序中读入字符。
3. 如果字符为结束符，转向（4）。
    - 如果读入的是开括号，将其进栈。
    - 如果读入的是闭括号但栈是空的，说明少开括号，报错并结束。
    - 如果读入的是闭括号但栈不空，将栈中的开括号出栈。如果出栈的开括号和读入的闭括号不是同种类型（如一个为小括号，一个为中括号），说明不匹配，报错并结束。
4. 继续从源程序中读入下一个符号，转向（3）。
5. 如果栈非空，说明开括号多了，报错并结束；
6. 否则括号配对成功，结束。

表达式计算

算术表达式中运算符出现在两个操作数之间，这种形式称为**中缀式**，运算符在前称为**前缀式或波兰式**，运算符在后称**后缀式或逆波兰式**。

中缀式有利于人的理解，但不便于计算机处理；前缀式不便于人理解，但可去掉括号；后缀式不便于人理解，可去掉括号，更便于计算机计算。在编译时，编译器首先要把中缀式转换成后缀式。

中缀式转后缀式算法分析：

- 对一个中缀表达式，从左至右顺序读入各操作数、运算符。
- 当读入的是操作数时，直接输出 (如：输出到屏幕或追加到保存后缀式的字符串中)；
- 当读入的是操作符时，当读入的运算符优先级高时，因不知是否后续读入的操作符优先级更高，只能将本次读入的运算符暂存，继续读入中缀式。当读入的运算符优先级低时，才可能计算刚才暂存的运算符(即输出)；在暂存机构中，越是后面存入的操作符，优先级越高，越先出来进行计算，这种结构就是栈，处于栈顶的运算符的优先级最高。
- 相同运算符，先进栈的优先级高于后进栈的。

- 表达式中的括号也可以看作是一种操作符，其中开括号具有两面性：即将进入栈的开括号，优先级最高；已经在栈顶的开括号，优先级最低。括号在后缀式中是要消失的，消失靠闭括号。当读入一个闭括号时，计算之前进栈的运算符，直到遇到一个开括号，然后开闭括号双双消失即可。

计算后缀式：

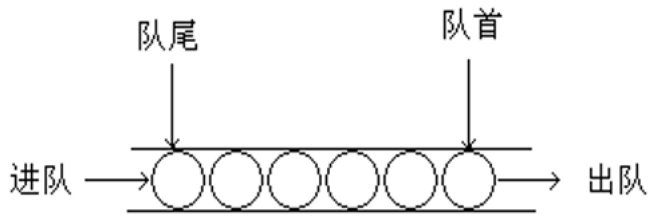
1. 声明一个操作数栈，依次读入后缀式中的字符。
2. 若读到的是操作数，将其进栈；
3. 若读到的是运算符，将栈顶的两个操作数出栈。后弹出的操作数为被操作数，先弹出的为操作数。将出栈的两个操作数完成运算符所规定的运算后将结果进栈。
4. 继续读入后缀式中的字符，如上处理，最后直到后缀式中所有字符读入完毕。
5. 当完成以上操作后，栈中只剩一个操作数，弹出该操作数，它就是表达式的计算结果。

3.2 队列

- 如果元素到达线性结构的时间越早，离开的时间就越早，这种线性结构称为**队（Queue）或者队列。**
- 同栈一样，为时间有序表，且可以看作是插入和删除操作位置受限的线性表。

> FIFO，First In First Out

- 可以将队列想象为一段管道，元素从一端流入，从另一端流出。流入端称为**队尾**，流出端称为**队首。**
- 元素从队首删除的操作，称为**出队**（deQueue）；元素在队尾位置插入的操作，称为**进队**（enQueue）。

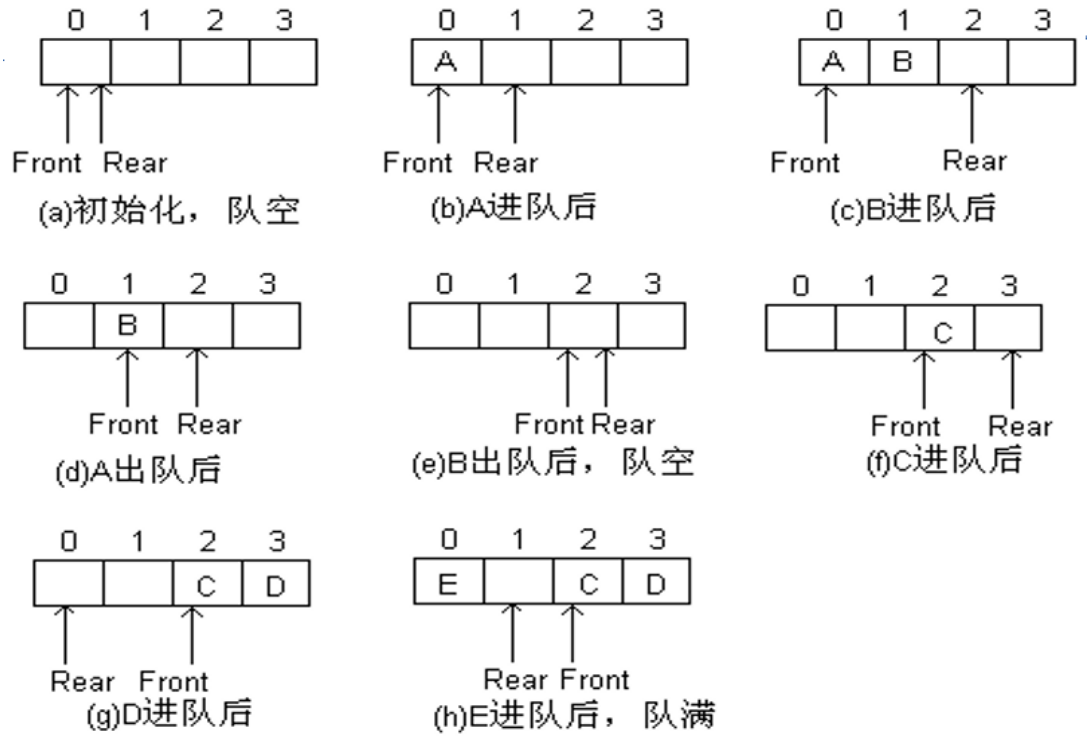


3.2.1 顺序队列

- 队首位置固定方式，缺点是出队会引起大量数据移动
- 队首位置不固定方式，所有操作都是O（1），但浪费空间。

较好的解决方法是：**循环队列。**队首指针Front给出的是实际队首元素的地址，队尾指针Rear给出的是实际队尾元素的下一个空间地址。为防止队列后移造成的空间浪费，附加一个循环特征，称**顺序循环队列。**即：Rear指向的下标可能小于Front指向的下标。进队则队尾后移，出队则队首后移。

- 队空标志： Rear==Front
- 队满标志：(Rear+1)%maxSize == Front




```
class illegalSize{};
class outOfBound{};

template <class elemType>
class seqQueue
{    private:
         elemType *array;
         int maxSize;
         int Front, Rear;
         void doubleSpace(); //扩展队队列元素的存储空间为原来的2倍
     public:
         seqQueue(int size=10); //初始化队列元素的存储空间
         bool isEmpty(); //判断队空否，空返回true，否则为false
         bool isFull();  //判断队满否，满返回true，否则为false
```

```
        elemType front(); //读取队首元素的值，队首不变
        void enQueue(const elemType &x); //将x进队，成为新的队尾
        void deQueue(); //将队首元素出队
        ~seqQueue(); //释放队列元素所占据的动态数组
};

template <class elemType>
seqQueue<elemType>::seqQueue(int size) //初始化队列元素的存储空间
{
    array = new elemType[size]; //申请实际的队列存储空间
    if (!array) throw illegalSize();
    maxSize = size;
    Front = Rear = 0;
}

template <class elemType>
bool seqQueue<elemType>::isEmpty()  //判断队空否，空返回true，否则为false
{return Front == Rear;}

template <class elemType>
bool seqQueue<elemType>::isFull() //判断队满否，满返回true，否则为false
{return (Rear+1)%maxSize == Front;}

template <class elemType>
elemType seqQueue<elemType>::front() //读取队首元素的值，队首不变
{
    if (isEmpty()) throw outOfBound();
    return array[Front];
}

template <class elemType>
void seqQueue<elemType>::enQueue(const elemType &x)  //将x进队，成为新的队尾
{  if (isFull())  doubleSpace();
    array[Rear]=x;
    Rear = (Rear+1)%maxSize;
}

template <class elemType>
void seqQueue<elemType>::deQueue() //将队首元素出队
{  if (isEmpty()) throw outOfBound();
    Front = (Front+1)%maxSize;
}

...
```

3.2.2 链式队列

用单链表存储队列中的元素及元素关系。其中队首指针Front指向队首结点、队尾指针Rear指向队尾结点。队空的条件为Front=Rear=NULL，队满为假。

3.2.3 优先队列

有时进入队列中的元素具有优先级（优先级可用一个优先数表示，一般优先数越小优先级越高），出队时是按照优先级越高的元素出队越早，优先级越低出队越晚，优先级相同者按先进先出的原则处理，这种队列称**优先队列**。优先队列可使用顺序存储或链式存储的方式。

有两种存放策略：

- 进队按时间顺序存放，出队是优先级高者出队。
- 进队按优先级顺序存放，出队是队首出队。

策略一

- 进队时，按照下标由小到大的顺序即直接将元素放队尾，时间为O(1)。
- 出队时，从所有元素中找到优先级最高的元素，然后删除。

  为避免整个队列的后移，造成空间的浪费，当有元素出队时将队列中最后一个元素移到出队元素所在的存储位置，这样队列始终从0下标开始到某个下标终止，中间不会出现空隙。故不需要像普通队列顺序存储时使用循环技术。查找最高优先级元素时间为O(n)
- 设队尾Rear指针指向实际队尾元素的后一单元。

  则队空的条件为：Rear == 0;　队满的条件为：Rear == maxSize

策略二

- 元素按照优先数由小到大排列。
- 元素进队时，先在队列中找到合适的插入位置，移动后面的元素，将新进元素插入，时间复杂度为O(n)。
- 出队时，删除队首即0下标元素即可，（顺序存储）为了避免后面元素的移动，可以采用顺序循环队列，时间复杂度为O(1)。
- 设队尾Rear指针指向实际队尾元素的后一单元。

  则队空的条件为：Rear == Front;

  队满的条件为：(Rear+1）%maxSize==Front。

CH4 树

4.1 树的定义与基本概念

两种定义方式

- 在一个元素集合中，如果元素呈上下层次关系。对一个结点而言，上层元素为其直接前驱且直接前驱唯一，下层元素为其直接后继且直接后继可以有多个，这样的结构就是**树结构。**
- **树**是有限个(n>0)元素组成的集合。在这个集合中，有一个结点称为根；如果有其他的结点，这些结点又被分为若干个互不相交的非空子集，每个子集又是一棵树，称为根的子树；每个子树都有自己的根，子树的根称为树根结点的孩子结点。

各类结点

- 一个结点的子树的根称为这个结点的**孩子结点**或**儿子结点。**相对之，该结点为**父结点。**
- 父结点的父结点称为**祖父结点**，根节点到树中某结点路径上所有结点称为其**祖先结点**，相对之有**子孙结点。**
- 同一父结点的结点互为**兄弟结点**，同一祖父不同父亲互为**堂兄弟结点。**

度和高度

- 树中每个结点拥有的孩子结点的个数称为该**结点的度**
  - 度为0的结点称**叶子结点**或**终端结点**
  - 度不为0的结点称**非叶子结点**或**中间结点**或**非终端结点。**
  - **树的度**是树中每个结点的度的最大值。
- 树中结点具有层次关系。**根的层次**数通常规定为1，其余结点的层次数是其父结点的层次数加1。树中所有结点的层次数的最大值就是**树的高度**

其他定义

- 对树中的任意一个结点，如果其孩子结点都被规定了一定的顺序，如谁是第一个孩子、谁是第二个孩子等，这棵树就称**有序树**。如果结点的孩子没有规定顺序，称为**无序树**。
- 两棵及以上的树称为**森林**(Forest)。
- 在树中，父结点可以看作是孩子结点的直接前驱、孩子结点可以看作是父结点的直接后继，直接前驱是唯一的、直接后继可以有多个。

4.2 二叉树

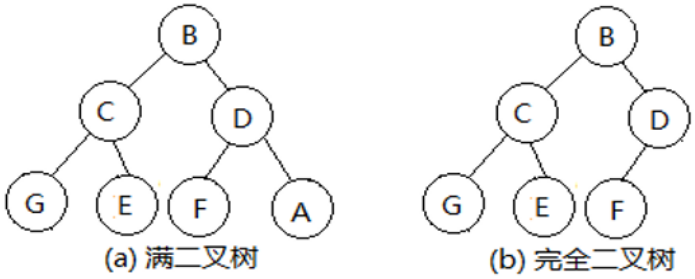
4.2.1 二叉树的定义与性质

二叉树是有限个(n≥0)结点的集合。它或者为空，或者有一个结点作为根结点，其余结点分成左右两个互不相交的子集作为根结点的左右子树，每个子树又是一棵二叉树。

- 二叉树中结点个数可以为0，允许一棵空二叉树存在，这是作为工具能更方便地使用；而树中结点个数不能为0，必须至少是1。
- 二叉树中左右孩子要明确指出是左还是右，即便只有一个孩子，也要指明它是左子还是右子；有序树中孩子只是进行了排序，没有左右之分，当某个结点只有一个孩子时，不需要确定其是左是右。

特殊二叉树

- 如果一棵k层二叉树中每一层结点数量都达到了最大值，该二叉树称为**满二叉树**。
- 如果一棵二叉树有k层，其中k-1层都是满的，第k层可能缺少一些结点，但缺少的结点是自右向左的，这样的二叉树称为**完全二叉树**。



结点高度与深度▲

结点高度是到叶结点的最长距离，深度是到根的距离

> 2014-2015-1，T1-8
>
> 二叉树有N个节点，高度为h，在其中插入一个新的节点，高度发生改变的节点个数最多为O(h)
>
> - 只会改变插入路径上的高度。在根节点插父节点只会改变所有结点深度，不会改变任意结点高度。

性质

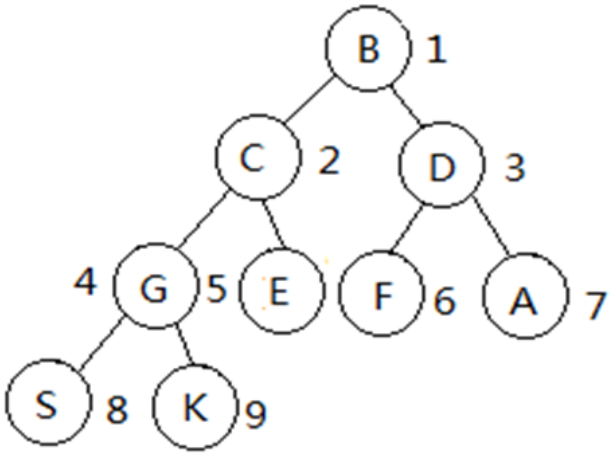
- 一棵非空二叉树的第i层上最多有2^{i-1}个结点（i≥1）
- 一棵高度为k的二叉树，最多具有2^k-1个结点。
- 对于一棵非空二叉树，如果叶子结点个数为n0，度数为2的结点个数为n2，则有: n0 = n2 + 1。
- 具有n个结点的完全二叉树的高度$k=[log_2 n]+1$。
- 如果对一棵有n个结点的完全二叉树中的所有结点按层次自上而下、每一层自左而右依次对其编号。若根结点的编号为1，则编号为i的结点（1≤i≤n），有以下性质：

  （1）如果i = 1，该结点是二叉树的根结点；如果i>1，其父亲结点的编号为[i/2] 。

  （2）如果2i > n，编号为i的结点无左儿子；否则，其左儿子的编号为2i。

（3）如果2i+1>n，编号为i的结点无右儿子；否则，其右儿子的编号为2i+1。

- 对于完全二叉树，如果一个结点是某个结点j的左子，则其编号有2j和j的关系；如果一个结点是某个结点j的右子，则其编号有2j+1和j的关系，故如果一个非根结点的编号是i，其父结点的编号就是[i/2]。



4.2.2 二叉树的存储

顺序存储

优点是数组访问简单，缺点是需要事先预估出数据最大规模。

- 用一组连续的空间即数组来存储二叉树中的结点。
- 每个结点除了包括元素值，还包括表达二叉树父子关系的字段。可设置四个字段： data、left、right、parent，其中left、right、parent为其左、右孩子及父结点在数组中的下标。
- 当一个结点没有孩子结点时，结点的left、right字段设置为-1；当一个结点没有父结点时，结点的parent字段设置为-1。
- 结点在存储时，可以随意地按照任何顺序将它们存储在数组中，元素之间的关系全靠字段left、right和parent来维系。

对于完全二叉树可以更简单。先对结点按照二叉树层次自上而下、自左向右进行编号，编号从0开始逐步加一，然后将结点存储在下标和其编号相同的数组分量中。（非完全树也可使用此方法，相对于完全二叉树，空余的结点留空或用特殊字符代替）

链式存储

两种形式：标准形式（不存储parent），广义标准形式（存储parent）

```
template <class elemType>
class BTree;
template <class elemType>
class Node
{
    friend class BTree<elemType>;
    private:
        elemType data;       Node *left, *right;
        int leftFlag;  //用于标识是否线索，0时left为左孩子结点，1时为前驱线索
        int rightFlag; //用于标识是否线索，0时right为右孩子结点，1时为后继线索
    public:
        Node(){left=NULL; right=NULL; leftFlag = 0; rightFlag=0;};
        Node(const elemType &e, Node* L=NULL, Node *R=NULL)
        {  data = e;
           left=L; right=R; leftFlag = 0; rightFlag=0;
        };
};
```

```
template <class elemType>
class BTree
{
    private:
        Node<elemType> *root;
        elemType stopFlag;

        int Size (Node<elemType> *t); //求以t为根的二叉树的结点个数。
        int Height (Node<elemType> *t); //求以t为根的二叉树的高度。
        void DelTree(Node<elemType> *t);//删除以t为根的二叉树
        void PreOrder(Node<elemType> *t);
        // 按前序遍历输出以t为根的二叉树的结点的数据值
        void InOrder(Node<elemType> *t);
        // 按中序遍历输出以t为根的二叉树的结点的数据值
        void PostOrder(Node<elemType> *t);
        // 按后序遍历输出以t为根的二叉树的结点的数据值
    public:
        BTree(){root=NULL;}
        void createTree(const elemType &flag);//创建一棵二叉树
        bool isEmpty () { return (root == NULL);}// 二叉树为空返回true，否则返回false
        Node<elemType> * GetRoot(){ return  root; }
        int Size (); //求二叉树的结点个数。
        int Height (); //求二叉树的高度。
        void DelTree();//删除二叉树
        void PreOrder();// 按前序遍历输出二叉树的结点的数据值
        void InOrder();// 按中序遍历输出二叉树的结点的数据值
        void PostOrder();// 按后序遍历输出二叉树的结点的数据值
        void LevelOrder();// 按中序遍历输出二叉树的结点的数据值
};
```

- 建立非空二叉树

  借助队列。根主动进队，根出队时创建并链接根的孩子（第二层所有结点），并使第二层结点全部进队。

  第二层结点出队时，又创建并链接第三层所有结点，如此循环。

```
template <class elemType>
void BTree<elemType>::createTree(const elemType &flag)
//创建一棵二叉树
{
    seqQueue<Node<elemType>*> que;
    elemType e, el, er;
    Node<elemType> *p, *pl, *pr;

    stopFlag = flag;
    cout<<"Please input the root: ";
    cin>>e;
        if (e==flag) { root = NULL; return;}
    p = new Node<elemType>(e);
    root = p; //根结点为该新创建结点
    que.enQueue(p);
    while (!que.isEmpty())
    {   p = que.front();  //获得队首元素并出队
        que.deQueue();
        cout<<"Please input the left child and the right child of "<<p->data
            <<" using "<<flag<<" as no child: ";
        cin>>el>>er;
```

```
        if (el!=flag) //该结点有左孩子
        {   pl = new Node<elemType>(el);
            p->left = pl;
            que.enQueue(pl);
        }
        if (er!=flag) //该结点有右孩子
        {   pr = new Node<elemType>(er);
            p->right = pr;
            que.enQueue(pr);
        }
    }
}
```

- 求属性操作
  - **Size**操作：如果二叉树为空，返回0，否则返回根的个数1加上根的左、右子树中结点个数；
  - **Height**操作：如果二叉树为空，返回0，否则返回根的高度1加上根的左、右子树高度中的最大值。

  均使用递归实现

以Size操作为例：

- 递归算法中参数需要体现出规模的变化，这里用一个结点地址，表示以该结点为根的子树。
- 从类的属性中可以看出，root为私有成员，故带参数的递归函数并不方便外部函数调用，故该递归函数设置为私有。
- 不带参数的size/height函数，虽不利于递归，但有利于外部函数调用，故设置为公有。
- 公有成员函数size/height可以通过调用私有成员函数size/height得以实现。

```
template <class elemType>   //公有
int BTree<elemType>::Size()
{ return Size(root); }

template <class elemType>  //私有
int BTree<elemType>::Size (Node<elemType> *t)
//得到以t为根二叉树结点个数，递归算法实现。
{  if (!t) return 0;
    return 1+Size(t->left)+Size(t->right);
}
```

```
template <class elemType>
int BTree<elemType>::Height()
{ return Height(root); }

template <class elemType>
int BTree<elemType>::Height(Node<elemType> *t)
//得到以t为根二叉树的高度，递归算法实现。
{   int hl, hr;
    if (!t) return 0;
    hl = Height(t->left);
    hr = Height(t->right);
    if (hl>=hr) return 1+hl;
    return 1+hr;
}
```

- 递归删除树

```
template <class elemType>
void BTree<elemType>::DelTree()
{  DelTree(root);    root = NULL;   }

template <class elemType>
void BTree<elemType>::DelTree(Node<elemType> *t)
//删除以t为根的二叉树，递归算法实现
{  if (!t) return;
    DelTree(t->left);  DelTree(t->right);
    delete t;
}
```

4.2.3 二叉树的遍历

层次遍历：如果二叉树为空，遍历操作为空。否则，从第一层开始，从上而下，逐层访问每一层结点。对同一层结点，自左向右逐一访问。

前序遍历：如果二叉树为空，遍历操作为空。否则，先访问根结点，然后前序遍历根的左子树，再前序遍历根的右子树。可简记为："根左右"。

中序遍历：如果二叉树为空，遍历操作为空。否则，先中序遍历根的左子树，然后访问根结点，最后中序遍历根的右子树。可简记为："左根右"。

后序遍历：如果二叉树为空，遍历操作为空。否则，先后序遍历根的左子树，然后后序遍历根的右子树，最后访问根结点。可简记为："左右根"。

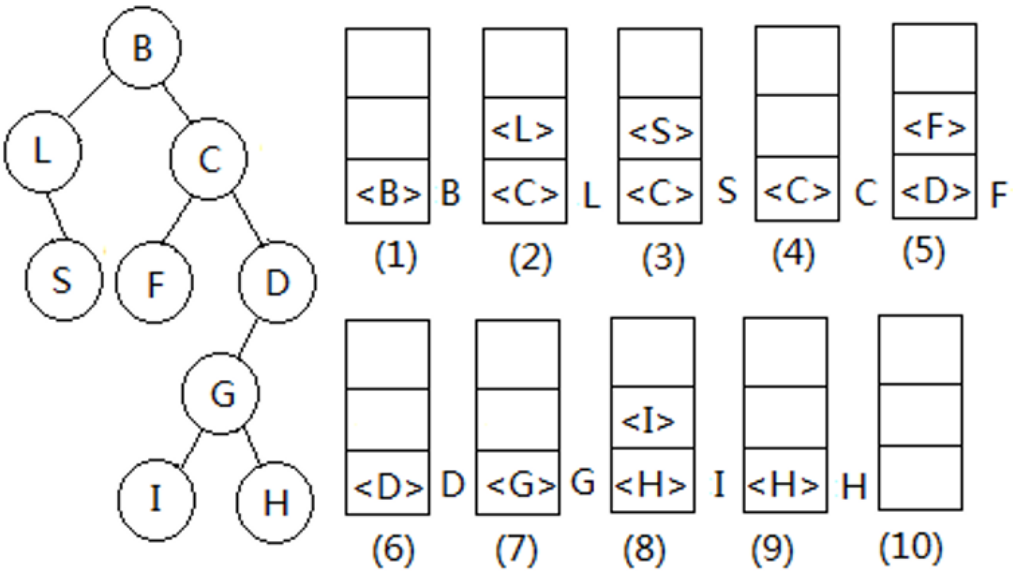
- 以前序遍历为例，递归形式：

```
template <class elemType>
void BTree<elemType>::PreOrder(Node<elemType> *t)
//前序遍历以t为根二叉树递归算法的实现。
{  if (!t) return;
    cout << t->data;

    PreOrder(t->left);
    PreOrder(t->right);
}
```

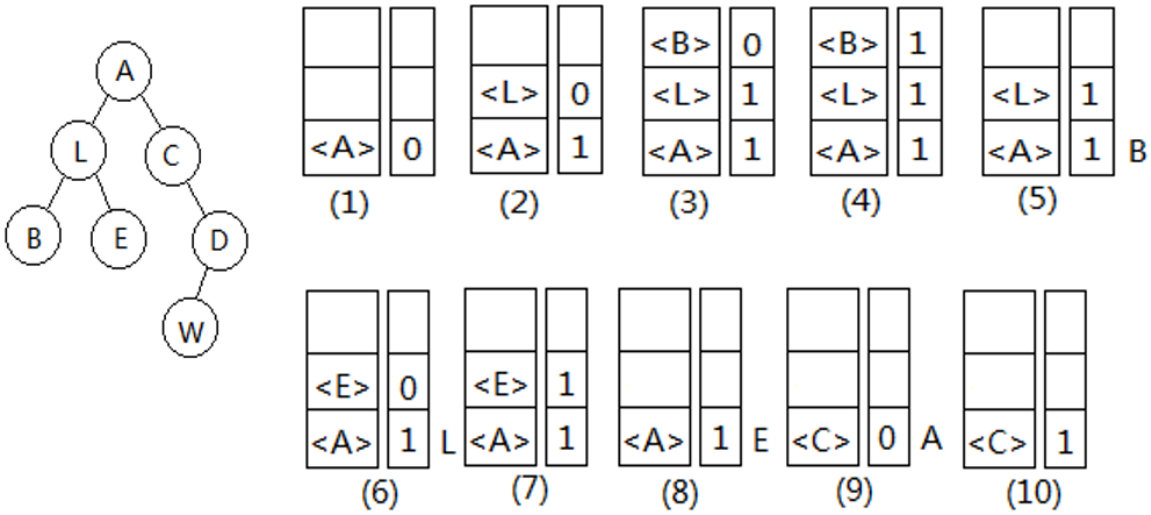
修改为中序、后序遍历，只需将递归语句和输出语句的位置进行变换。

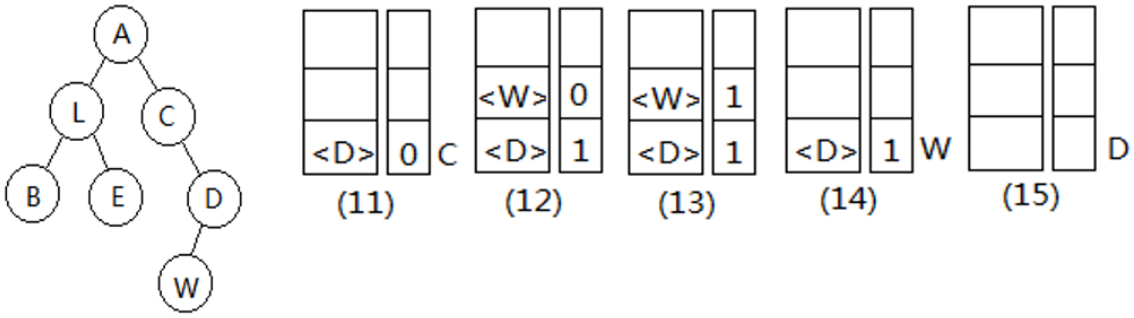
- 非递归形式，将递归时系统内部的栈拉至前台，显式地使用栈。时间复杂度O(n)。




```
template <class elemType>
void BTree<elemType>::PreOrder()
//前序遍历的非递归算法实现
{
    if (!root) return;
    Node<elemType> *p;
    seqStack<Node<elemType> *> s;
    s.push(root);
    while (!s.isEmpty())
    {  p=s.top(); s.pop();
        cout << p->data;
        if (p->right) s.push(p->right);
        if (p->left)  s.push(p->left);
    }
    cout << endl;
}
```

对于中序的非递归实现，需要为元素引入一个标志计算是"第几次出栈"。第二次出栈时才输出并将右子进栈，否则再次进栈并将左子进栈。时间复杂度还是O（n）






```
template <class elemType>
void BTree<elemType>::InOrder()
//中序遍历的非递归算法实现。
{  if (!root) return;
    seqStack<Node<elemType> *> s1;
    seqStack<int> s2;
    Node<elemType> *p;
    int flag;
    int zero=0, one=1;
    p=root;
    s1.push(p); s2.push(zero);
    while (!s1.isEmpty())
    {   flag = s2.top(); s2.pop();
        p = s1.top(); //读取栈顶元素
        if (flag==1)
        {   s1.pop();
            cout << p->data;
            if (!p->right) continue;
            //有右子压右子，没有进入下一轮循环
            s1.push(p->right);
            s2.push(zero);
        }
        else
        {   s2.push(one);
            if (p->left) //有左子压左子
            {   s1.push(p->left);
                s2.push(zero);
            }
        }
    }
    cout<<endl;
}
```

> ztz例程使用第二个栈同步存储元素出栈次数，yyu例程将node外包装结构体存储出栈次数

后序遍历的非递归算法，第三次出栈才输出。第一次出栈再次进栈并将左子进栈；第二次出栈再次进栈并将右子进栈。时间复杂度还是O（n）

- 层次遍历（非递归实现）

和前序遍历非递归算法思路类似，只是把**辅助工具由栈换为队列**。用一个队列完全接管结点，由它的进、出队顺序来决定结点的访问次序。时间复杂度是O（n）

```
template <class elemType>
void BTree<elemType>::LevelOrder()
```

```
//层次遍历二叉树算法的实现。
{   seqQueue<Node<elemType> *> que;
    Node<elemType> *p;

    if (!root) return; //二叉树为空
    que.enQueue(root);
    while (!que.isEmpty())
        {
            p = que.front();
            que.deQueue();
            cout << p->data;

            if (p->left) que.enQueue(p->left);
            if (p->right)que.enQueue(p->right);
        }
    cout << endl;
}
```

4.2.4 二叉线索树

二叉树中有n+1个左右指针域是空的。把二叉树中空指针域利用起来：如果一个结点的左指针域为空，就把某种遍历序列中这个结点的直接前驱结点地址存于左指针域；如果一个结点的右指针域空着，就把某种遍历序列中这个结点的直接后继结点地址存于右指针域。在空指针域上加载了某种遍历线索的二叉树就叫**线索树**。

- 在一棵中序线索树上实现中序遍历操作就可以摆脱对栈的依赖。

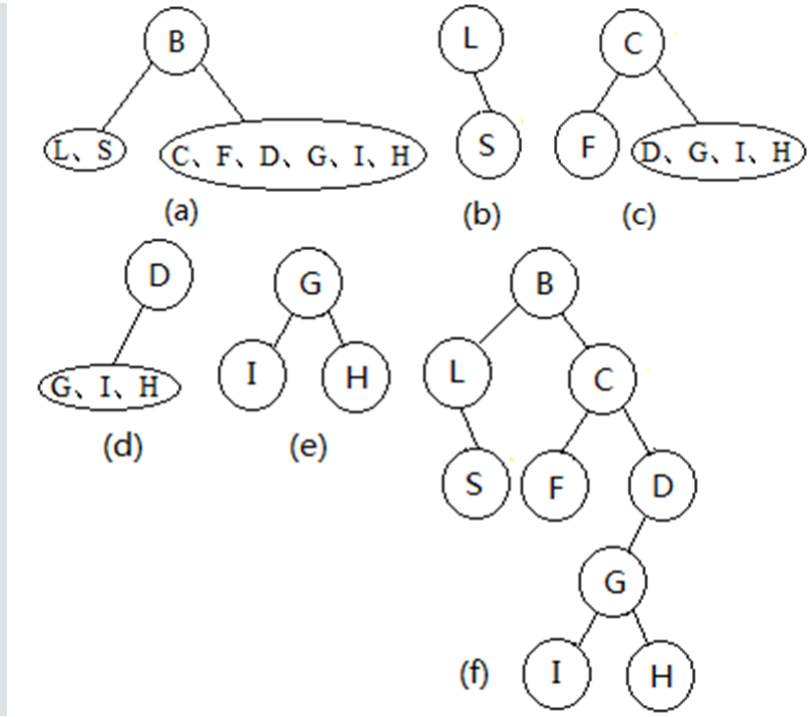
4.2.5 遍历序列确定二叉树（二叉树的重构）

- 已知一棵**完全二叉树**的层次遍历，能唯一确定这个完全二叉树。
- 已知一棵**满二叉树**的前序、中序、后序遍历之一，能唯一确定这棵满二叉树。
- 已知一个**一般二叉树**的前序、中序、后序遍历之一，是**不能唯一确定**这棵二叉树的。

- 当给了一棵二叉树的前序和中序遍历序列，能唯一确定一棵二叉树；
- 当给了一棵二叉树的后序和中序遍历序列，能唯一确定一棵二叉树。
- 当给了一棵二叉树的前序和后序遍历序列，**不能唯一确定**一棵二叉树。

例：根据前序、中序序列确定二叉树

> 已知一棵二叉树的前序和中序序列为：
>
> 前序序列：B、L、S、C、F、D、G、I、H
>
> 中序序列：L、S、B、F、C、I、G、H、D



- 两个序列分别在前序数组和中序数组中。
- 由前序数组定出根结点值后，在中序数组找到根结点值所在的位置，由此找到了前序和中序序列中根的左子树下标范围，同样也找到了右子树的下标范围。
- 根据根结点的值创建根结点空间，并分别利用两个数组和其左子树下标范围、右子树下标范围递归确定其左、右子树，将左右子树的根地址写入根结点的左右孩子指针中

```
template <class elemType>
Node<elemType> *BTree<elemType>::buildTree(elemType pre[], int pl, int pr,
                                           elemType mid[], int ml, int mr)
//pre数组存储了前序遍历序列，pl为序列左边界下标，pr为序列右边界下标。
//min数组存储了中序遍历序列，ml为序列左边界下标，mr为序列右边界下标。
{  Node<elemType> *p, *leftRoot, *rightRoot;
    int i, pos, num;
    int lpl, lpr, lml, lmr; //左子树中前序的左右边界、中序的左右边界
    int rpl, rpr, rml, rmr; //右子树中前序的左右边界、中序的左右边界
    if (pl>pr) return NULL;
    p = new Node<elemType>(pre[pl]); //找到子树的根并创建结点
    if (!root) root = p;

    //找根在中序中的位置和左子树中结点个数
    for (i=ml; i<=mr; i++)
        if (mid[i]==pre[pl]) break;
    pos = i;    //子树根在中序中的下标
    num = pos-ml; //子树根的左子树中结点的个数
    //找左子树的前序、中序序列下标范围
    lpl = pl+1; lpr = pl+num;
    lml = ml; lmr = pos-1;
    leftRoot = buildTree(pre, lpl, lpr, mid, lml, lmr);

    //找右子树的前序、中序序列下标范围
    rpl = pl+num+1; rpr = pr;
    rml = pos+1; rmr = mr;
    rightRoot = buildTree(pre, rpl, rpr, mid, rml, rmr);
    p->left = leftRoot;
    p->right = rightRoot;
    return p;
```

```
}
```

ACMOJ-1049，使用数组＋补满为完全二叉树（空余部分用初始化时特殊字符代替），利用下标关系存储。

```
void buildTree(char *pre, char *mid, int len, int idx){
    if (len<=0) return;
    t[idx] = pre[0];
    int i=0;
    while(mid[i]!=pre[0]){i++;}
    buildTree(pre+1, mid, i, 2*idx);
    buildTree(pre+i+1, mid+i+1, len-i-1, 2*idx+1);
}
```

4.3 最优二叉树

4.3.1 哈夫曼算法

二叉树中任意两个结点间的**路径长度**为其路径上的分支总数。

二叉树的路径长度为根到树中各个结点的路径长度之和。

特别地：如果二叉树中叶子结点上带有权值，**二叉树的加权路径长度**特指从根结点到各个叶子结点路径上的分支数乘以该叶子的权值之和。

- 哈夫曼算法就是利用了**权值越大，越靠近根**的思想，逐步比较结点的权值，构造出了**哈夫曼树**，哈夫曼树是一棵最优二叉树。
- 哈夫曼算法的具体步骤为：对于给定的一个带权结点集合U

  1.如果U中只有一个结点，操作结束，否则转向2）。

  2.在集合中选取两个权值最小的结点x、y，构造一个新的结点z。新结点z的权值为结点x、y的权值之和。在集合U中删除结点x和y并加入新结点z，然后转向1）。

4.3.2 最优二叉树性质

- 设T为带权w1≤w2≤...≤wt的一组结点集合构成的最优二叉树，则
  1. 带权w1，w2的叶子nw1，nw2是兄弟。
  2. 以叶子nw1，nw2为儿子结点的内部结点，其路径长度最长。
- 设T为带权w1≤w2≤...≤wt的一组结点集合构成的最优二叉树，若将以带权w1和w2的叶子为儿子的内部结点改为带权w1+w2的叶子，得到一棵新树T1，则T1也是最优二叉树。
- 假定树中叶子结点有n个，最优二叉树中结点总数为n+n-1=2n-1个。

4.3.3 哈夫曼算法实现

二叉树用**顺序存储法**：用一个数组存储哈夫曼结点。哈夫曼结点（叶子、中间结点）**HuffmanNode**：

包含5个字段：data为结点的值，weight为结点的权值，parent为结点的父结点地址（下标），left和right为结点的左、右孩子的地址（下标）。

（**特殊地**：数组的0下标分量空出来不用，从下标为1的数组分量开始存储数据）

开辟动态数组时，考虑到0下标分量不用，需要为数组申请2n个连续的结点空间。初始时，数组中只有叶子结点，其parent、left、right都设置为0。叶子结点在数组中**从后往前**依次存储，前面空余的n-1个数组分量作为存储即将构造的中间结点用。

对一组带权结点U={(A，3)，(B，8)，(C，10)，(D，12)，(E，50)，(F，4)}，按照哈夫曼算法构造了一棵最优二叉树。

index	1	2	3	4	5	6	7	8	9	10	11
data						F	E	D	C	B	A
weight	87	37	22	15	7	4	50	12	10	8	3
parent	0	1	2	2	4	5	1	3	3	4	5
left	2	4	9	5	11	0	0	0	0	0	0
right	7	3	8	10	6	0	0	0	0	0	0

> 仅计算WPL时，仅需存储weight，parent。

时间复杂度分析：

- 为n个叶子结点设置初始状态，时间消耗O(n)。
- 执行了n-1次创建新的中间结点操作，每次创建要在所有元素(元素个数在n和2n之间)中两次去找权值最小的元素，时间耗费为2n。

算法总的时间复杂度为O(n^2)。

4.3.4 哈夫曼编码

同离散。从根结点到哈夫曼树叶子结点，向左走为0、向右走为1

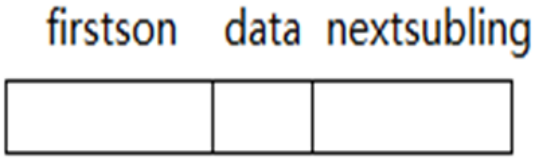
算法实践中从叶子追溯到根获得逆序哈夫曼编码。（yyu，P202）

时间复杂度分析：算法包含了两重循环，外循环次数为叶子结点的个数n，内循环串行地做了两件事：一个是从叶子逐步追溯到根获取哈夫曼编码的逆序，一个是逐步弹栈获取哈夫曼编码。内循环的两个操作消耗的时间都最多是哈夫曼树的高度，而哈夫曼树的形态、高度取决于这组字符的频度分布。**最好时，哈夫曼可能达到的树高是log_2n**；最差时，哈夫曼树的树高会达到n，因此求哈夫曼编码算法的时间复杂度最好为O(n log_2n)，最差为O(n^2)。

4.4 树和森林

4.4.1 孩子兄弟表示法

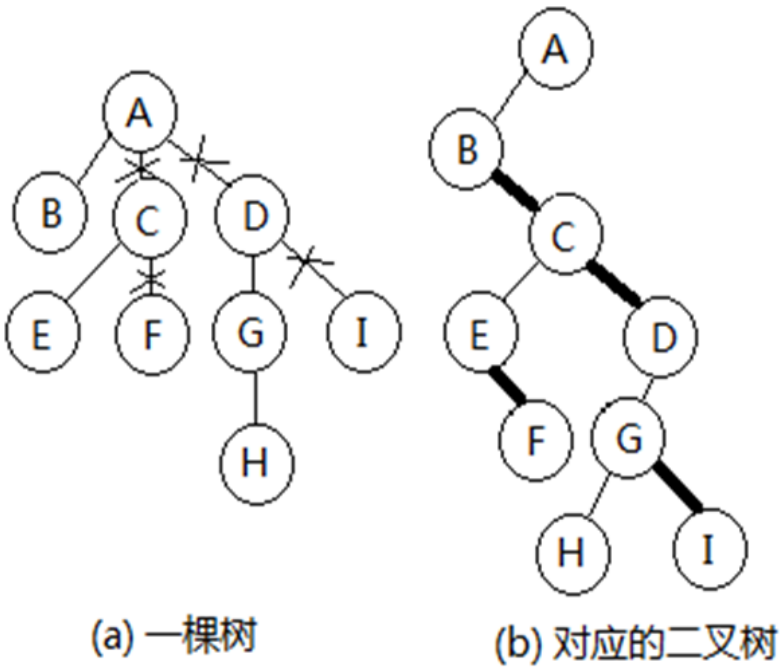
每个结点除了保存数据，还保存了该结点的最大孩子结点地址和最大弟弟的结点地址。



data字段保存了结点数据、**firstchild**字段保存了最大孩子结点的地址、**nextsibling**字段保存了最大弟弟的结点地址。以下为了方便，有时称firstchild为结点的左分支、左子或左手，称nextsibling为右分支、右子或右手。

4.4.2 树到二叉树的转换

树的根就是二叉树的根。对树中每个结点，保留其到最大孩子的分支，对其余孩子删除其到父结点的分支，逐个降级，增加其左侧哥哥（最小哥哥）到它的右分支，将它链到最小哥哥结点的右分支上去。

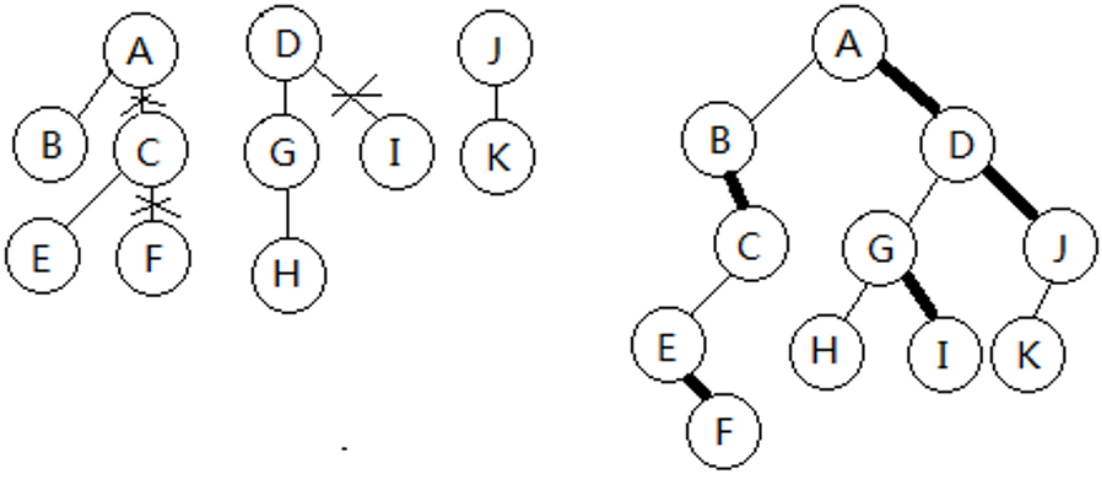


(a) 一棵树　　(b) 对应的二叉树

4.4.3 森林到二叉树的转换

将森林中的每棵树转换为对应的二叉树，每棵二叉树的根就是它对应树的根。将每棵树的根看作兄弟，即二叉树的根为兄弟。

将第一棵二叉树的根作为森林对应的二叉树的根。其余二叉树的根由其最小的哥哥结点用nextsibling字段链接。



(a) 森林　　(b) 对应的二叉树

CH5 图

5.1 图的基本概念

度

- 有向图中一个顶点的**出度**是指由该顶点射出的有向边的条数，一个顶点的**入度**则是射入该顶点的有向边的条数。
- 无向图中一个顶点的**度**是指邻接于该顶点的边的总数。

完全图

- 有n个顶点的无向图中，如果任意两个顶点间都有边相连，此时边的条数最多，达到$C_n^2 = n(n-1)/2$条，这样的图称为**无向完全图**；对有向图而言，边的条数最多为$P_n^2 = n(n-1)$，这样的图称为**有向完全图**。

简单路径/回路

- 如果一条路径上除了第一个顶点和最后一个顶点可能相同之外，其余各顶点都不相同，这样的路径称为**简单路径**。简单路径上如果第一个顶点和最后一个顶点相同，则该路径也称为**简单回路或简单环**。

连通与生成树

- 在一个图中，如果顶点i和j之间有路径存在，称顶点i、j 之间是**连通**的。在一个无向图中，如果任意两个顶点对之间都是连通的，称该无向图G是**连通图**。无向图的极大连通子图称为**连通分量**。
- 在一个有向图G中，如果任意两个顶点对之间都是连通的，称有向图G是**强连通图**。有向图的极大连通子图，称**强连通分量**。
- 连通图的**生成树**是指它的**极小连通子图**，该连通子图包含连通图的所有n个顶点，但只含它的n-1条边。如果去掉一条边，这个子图将不连通
- 一个连通图的生成树并不唯一。

5.2 图的存储表示

5.2.1 邻接矩阵

a[i,j]=1时表示边<vi,vj>存在。

性质

- 在有向图中，其邻接矩阵某一行中所有1的个数，就是相应行顶点的出度；而某一列中所有1的个数，就是相应列顶点的入度。
- 在无向图中，某一行中所有1的个数或者某一列中所有1的个数，就是相应顶点的度。
- 无向图中，同一条边在邻接矩阵中出现两次，无向图的邻接矩阵是以主对角线为轴对称的，主对角线全为零，因此在存储无向图时可以只存储它的上三角矩阵或下三角矩阵。
- 一般来说，边的总数即便远远小于n^2，也需n^2个内存单元来存储边的信息，空间消耗大。

```
template <class verType, class edgeType>
class Graph
{  private:
        int verts, edges;   //图的实际顶点数和实际边数
        int maxVertex;      //图顶点的最大可能数量
        verType *verList;   // 保存顶点数据的一维数组
        edgeType **edgeMatrix; //保存邻接矩阵内容的二维数组
        edgeType noEdge;       //无边的标志，一般图为0， 网为无穷大MAXNUM
        bool directed;          //有向图为1，无向图为0
    public:
        //初始化图结构g，direct为是否有向图标志，e为无边数据
        Graph(bool direct, edgeType e);      ~Graph();
        int numberOfVertex()const{ return verts; }; // 返回图当前顶点数
        int numberOfEdge()const{ return edges; }; // 返回图当前边数
        //返回顶点为vertex值的元素在顶点表中的下标
        int getVertex(verType vertex)const;
        //判断某两个顶点间是否有边
        bool existEdge(verType vertex1,verType vertex2)const;
        void insertVertex(verType vertex ); //插入顶点
        void insertEdge(verType vertex1, verType vertex2, edgeType edge); //插入
边
        void removeVertex(verType vertex);  //删除顶点
```

```
        void removeEdge(verType vertex1, verType vertex2); //删除边
        //返回顶点vertex的第一个邻接点,如果无邻接点返回-1
        int getFirstNeighbor(verType vertex ) const;
        //返回顶点vertex1相对vertex2的下一个邻接点，如果无下一个邻接点返回-1
        int getNextNeighbor(verType vertex1, verType vertex2)const;
        void disp()const; //显示邻接矩阵的值
};

//构造函数
template <class verType, class edgeType>
Graph<verType, edgeType>::Graph(bool direct, edgeType e)
{  int i, j;
    //初始化属性 direct为是否有向图标志，e为无边数据
    directed = direct;    noEdge = e;    verts = 0;   edges = 0;
    maxVertex = DefaultNumVertex;
    //为存顶点的一维数组和存边的二维数组创建空间
    verList = new verType[maxVertex];
    edgeMatrix = new edgeType*[maxVertex];
    for (i=0; i<maxVertex; i++)
        edgeMatrix[i] = new edgeType[maxVertex];

    //初始化二维数组，边的个数为0
    for (i=0; i<maxVertex; i++)
        for (j=0; j<maxVertex; j++)
            if (i==j)
                edgeMatrix[i][j] = 0;//对角线元素
            else
                edgeMatrix[i][j] = noEdge; //无边
}

//析构函数
template <class verType, class edgeType>
Graph<verType, edgeType>::~Graph()
{
    int i;
    delete []verList;
    for (i=0; i<maxVertex; i++)
        delete []edgeMatrix[i];
    delete []edgeMatrix;
}

//判断某两个顶点是否有边
template <class verType, class edgeType>
bool Graph<verType, edgeType>::
existEdge(verType vertex1,verType vertex2)
const
{  int i, j;
    //找到vertex1和vertex2的下标
    for (i=0; i<verts; i++)
        if (verList[i]==vertex1)
            break;
    for (j=0; j<verts; j++)
            if (verList[j]==vertex2)
                break;

    if (i==verts || j==verts)   return false;
    if (i==j) return false;
    if (edgeMatrix[i][j] == noEdge) return false;
```

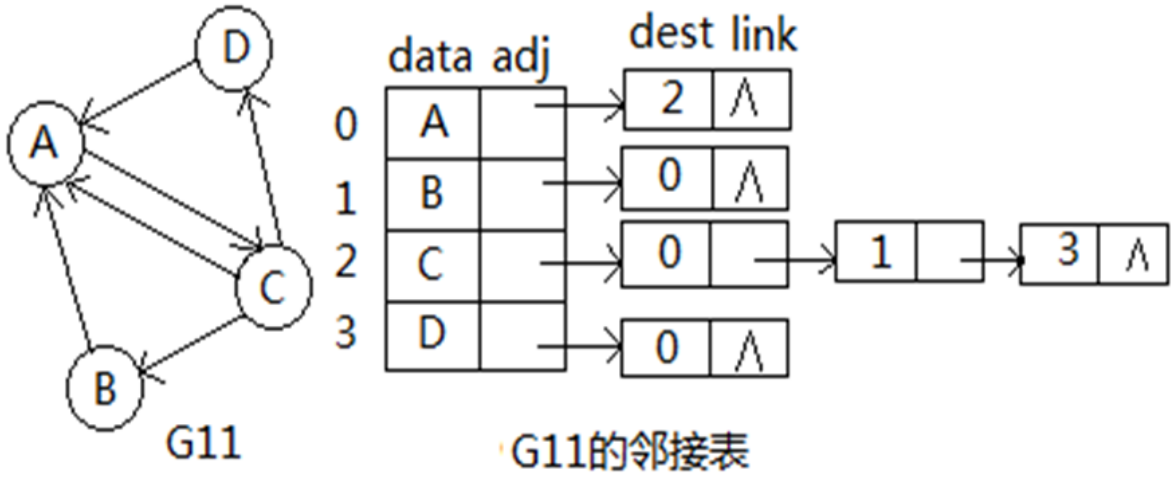
```
    return true;
}

//删除顶点
template <class verType, class edgeType>
void Graph<verType, edgeType>::removeVertex(verType vertex)
{   int i, j, k;
    //找到该顶点在顶点表中的下标
    for (i=0; i<verts; i++)
        if (verList[i]==vertex)  break;
    if (i==verts) return
    //在顶点表中删除顶点
    for (j=i; j<verts-1; j++)
        verList[j] = verList[j+1];
    //计数删除顶点射出的边,边数减少
    for (j=0; j<verts; j++)
        if ( (j!=i) && (edgeMatrix[i][j]!= noEdge))
            edges--;
    //如果是有向图，计数删除顶点射入的边,边数减少
    if (directed)
    {   for (k=0; k<verts; k++)
          if (( (k!=i) && edgeMatrix[k][i]!= noEdge))
                edges--;
    }
    //第i行之后所有行上移
    for (j=i; j<verts-1; j++)
    {   for (k=0; k<verts; k++)
        {   edgeMatrix[j][k] = edgeMatrix[j+1][k];   }
    }
    //第i列之后所有列前移
    for (j=i; j<verts-1; j++)
    {
        for (k=0; k<verts; k++)
             edgeMatrix[k][j] = edgeMatrix[k][j+1];
    }
     verts--;
}
```

5.2.2 邻接表

对于无向图，邻接于同一个顶点的所有边形成一条单链表；对于有向图，自同一个顶点出发的所有边形成一条单链表。顶点信息可以用一个一维数组来存储，这个数组称为**顶点表**，保存边信息的单链表称为**边表**。一个图可以由顶点表和边表共同表示，这种方法称为**邻接表表示法**。

- 方便计算出度，不方便计算入度



- 顶点表也可以使用单链表，无需预估顶点个数，方便插入顶点。
- **逆邻接表**，射向同一个顶点的所有边形成一条单链表。方便计算入度，不方便计算出度。

```
//顶点结点、边结点设计
template <class edgeType>
struct edgeNode
{  int dest; //边指向的下标
    edgeType weight;
    edgeNode *link;
};
template <class verType, class edgeType>
struct verNode
{  verType data;
    edgeNode<edgeType> *adj;
};
```

5.3 图的遍历

5.3.1 深度优先遍历（DFS）

访问方式类似于二叉树的前序访问，访问方式如下：

1.选中第一个未被访问过的顶点。

2.访问、对顶点加已访问标志。

3.依次从顶点的未被访问过的第一个、第二个、第三个..... 邻接顶点出发，依次进行深度优先搜索。即转向2。（**标一个递归一个**）

4.如果还有顶点未被访问过，选中其中一个作为起始顶点，转向2。

如果所有的顶点都被访问到，结束。

- 同一个图的深度优先遍历结果并不唯一

注意：

- 图可能不连通，从一个顶点开始做深度优先遍历可能只能访问到部分顶点，此时需要重新选择尚未访问的顶点，从它开始再次开始深度优先遍历。
- 一个顶点可能和其他多个顶点邻接，故以它为起始顶点做深度优先遍历前需检查是否已经访问过。如果未访问过，遍历才能进行。

```
//基于邻接表DFS，递归
template <class verType, class edgeType>
```

```
void Graph<verType, edgeType>::DFS(int start, bool visited[])const
{  edgeNode<edgeType> *p;
    cout<<verList[start].data<<'\t';     visited[start] = true;
    p = verList[start].adj;
    while (p)
    {  if (!visited[p->dest]) DFS(p->dest, visited);
        p = p->link;
    }
}
template <class verType, class edgeType>
void Graph<verType, edgeType>::DFS()const
{   bool *visited;     int i;
    visited = new bool[verts];
    for (i=0; i<verts; i++) visited[i]=false;
    for (i=0; i<verts; i++)
    {  if (!visited[i]) DFS(i, visited);
        cout<<endl;
     }
}
```

时间复杂度:

DFS()中第一个for循环初始化visited数组，时间为O(n);

第二个for循环中的每一次循环体的执行都有一个顶点被访检查，一共有n个顶点，每个顶点又通过DFS(int start, bool visited[])遍历了它的边表，因此总的时间复杂度为O(n+e)。

非递归实现:

类似二叉树前序遍历的非递归算法。建立一个栈，选一个顶点进栈，然后反复进行以下操作：如果栈不空，弹出访问，第一个未被访问的邻接点进栈，第二个未被访问的邻接点进栈，…，最后一个未被访问的邻接点进栈。

```
//基于邻接表DFS，非递归
template <class verType, class edgeType>
void Graph<verType, edgeType>::DFS()const
{  seqStack<int> s;
    edgeNode<edgeType> *p;   bool *visited;      int i, start;

    //为visited创建动态数组空间，并置初始访问标志为false。
    visited = new bool[verts];   if (!visited) throw illegalSize();
    for (i=0; i<verts; i++)   visited[i]=false;
    //逐一找到未被访问过顶点，做深度优先遍历
    for (i=0; i<verts; i++)
    {  if (visited[i]) continue;
        s.push(i);
        while (!s.isEmpty())
        {   start = s.top(); s.pop();
            if (visited[start]) continue;
            cout<<verList[start].data<<'\t';
            visited[start] = true;
            p = verList[start].adj;
            while (p)
            {
                if (!visited[p->dest])
                s.push(p->dest);
                p = p->link;
            }
```

```
        }
        cout<<'\n';
    }
}
```

5.3.2 广度优先遍历（BFS）

访问方式类似于二叉树的层次遍历。：

1.选中第一个未被访问的顶点。

2.访问、对顶点置已访问过的标志。

3.依次对顶点的未被访问过的第一个、第二个、第三个.....第 m 个邻接顶点 W1 、W2、W3..... Wm 进行访问且进行标记。

4.依次对顶点 W1 、W2、W3..... Wm 转向操作3。（**全标完再开始递归**）

5.如果还有顶点未被访问，任选其中一个顶点作为起始顶点，转向2。

如果所有的顶点都被访问到，遍历结束。

利用队列，程序首先将所有顶点的访问标志初始化为false，然后进入外层for循环。在外循环中，顺序找未被访问过的顶点作起始顶点，将起始顶点进队，然后反复执行以下循环：顶点出队，如果未访问过，访问之并将它所有未被访问过的邻接点进队，反复循环，直到队空。继续下一轮外循环，直到所有的顶点都被检查过

```
template <class verType, class edgeType>
void Graph<verType, edgeType>::BFS()const//广度优先遍历
{
    seqQueue<int> q;
    edgeNode<edgeType> *p;
    bool *visited;   int i, start;

    //为visited创建动态数组空间，并置初始访问标志为false。
    visited = new bool[verts];
    if (!visited) throw illegalSize();
    for (i=0; i<verts; i++) visited[i]=false;
    //逐一找到未被访问过顶点，
    //做广度优先遍历
    for (i=0; i<verts; i++)
    {     if (visited[i]) continue;
          q.enQueue(i);
          while (!q.isEmpty())
          {   start = q.front(); q.deQueue();
              if (visited[start]) continue;
              cout<<verList[start].data<<'\t';
              visited[start] = true;
              p = verList[start].adj;
              while (p)
              {  if (!visited[p->dest])
                  q.enQueue(p->dest);
                  p = p->link;
              }
          }
     cout<<'\n';
     }
}
```

时间复杂度：O（n+e）

5.3.3 图的连通性

做广度优先遍历时，增加一个计数器。在最外层循环“入队”操作处计数器+1，若遍历结束计数器为1（只从一个顶点出发过），则图连通（即有1个连通支）

计数器的值为连通器个数

有向图的连通性

- 当有向图的强连通分量只有一个时，说明它是强连通图。
- 当有向图的强连通分量不止一个时，说明它不是强连通图。

  对一个强连通分量来说，要求每一对顶点间都有路径可达，比如顶点i和j，不光要从i能到j，还要求从j能到i。

欧拉回路

如果图中一条路径经过了每条边一次且仅一次，这条路径称**欧拉路径**。如果一条欧拉路径的起终点相同，是一个回路，称**欧拉回路**，具有欧拉回路的图称**欧拉图**(简称E图)。具有欧拉路径但不具有欧拉回路的图称**半欧拉图**。

性质：

- 一个无向连通图中，如果度为奇数的顶点超过了2个，则欧拉路径是不存在的。
- 一个无向连通图中，如果除了两个顶点的度是奇数而其他顶点的度都是偶数，则从一个度为奇数的顶点出发一定能找到一条**欧拉路径**，经过每条边一次且仅一次的路径回到另外一个度为奇数的顶点。
- 一个无向连通图中，如果顶点的度都是偶数，则从任意一个顶点出发都能经过每条边一次且仅一次并回到原来的顶点（形成**欧拉回路**）

求解：

1.任选一个顶点v，从该顶点出发开始**深度优先搜索**，搜索路径上都是由未访问过的边构成，搜索中访问这些边，最后直到回到顶点v且v没有尚未被访问的边，此时便得到了一个回路，此回路为当前结果回路。

2.在搜索路径上另外找一个尚有未访问边的顶点，继续如上操作，找到另外一个回路，将该回路拼接在当前结果回路上，形成一个大的、新的结果回路。

3.如果在新的结果回路中，还有中间某结点有尚未访问的边，回到2）；如果没有任何中间顶点尚余未访问的边，访问结束，当前结果回路即欧拉回路。

5.4 AOV网和AOE网

5.4.1 拓扑排序

把有向无环图中的结点按下述规则排序：如果有一条u到v的路径，那么在拓扑排序中v必须出现在u之后。典型应用：课程先修关系问题。

实现：

- 首先在图中，找到入度为0的顶点，将这些顶点全部入栈，然后反复循环判断栈是否空，非空则执行以下操作：
- 顶点出栈，如果由该顶点射出了m条有向边，射入的这m个邻接点的入度减一（相当于该顶点对其m个邻接顶点的先修约束已经消失），在各邻接点入度减一的过程中，一旦发现哪个邻接点的入度变为0，将它进栈，然后再次回到循环，直到栈空。

```
template <class verType, class edgeType>
void Graph<verType, edgeType>::topoSort()const
{   int *inDegree;      seqStack<int> s;   int i, j;

    //创建空间并初始化计算每个顶点的入度,
    //邻接矩阵每一列元素相加,加完入度为零的压栈
    inDegree = new int[verts];
    for (j=0; j<verts; j++)
    {   inDegree[j] = 0;
        for (i=0; i<verts; i++)
        {     if ((i!=j)&&(edgeMatrix[i][j]!=noEdge))   inDegree[j]++;   }
        if (inDegree[j]==0) s.push(j);
    }
    //逐一处理栈中的元素
    while (!s.isEmpty())
    {   i = s.top(); s.pop();
        cout<<i<<"  ";

        //将i射出的边指示的邻接点入度减一，减为零时压栈
        for (j=0; j<verts; j++)
            if ((j!=i)&&(edgeMatrix[i][j]!=noEdge))
            {    inDegree[j]--;
                 if (inDegree[j]==0) s.push(j);
            }
    }
    cout<<endl;
}
```

很明显，算法的时间代价是O(n^2)。如果图用邻接表来存储，时间代价为O(n+e)。

5.4.2 关键路径

AOE网将活动赋予边之上，顶点表达了活动发生后到达的某种状态或事件。某个状态或事件既意味着前面所有的活动结束，也意味着后面的活动可以开始。AOE网的一个典型应用是工程问题。

关键子工程即**关键活动**会形成一条从总体工程开始和完工之间的路径，这条路径便是**关键路径。**

- 求每个**顶点事件的最早发生时间**，即从起点到达顶点所需要的最短时间。
- 求每个**顶点事件的最迟发生时间**，即从起点到达顶点所能容忍的最长时间。
- 求每个**活动的最早开始时间**，即每个边表示的活动最早何时能开始。
- 求每个**活动的最迟开始时间**，即每个边表示的活动最晚何时必须开始。
- 当某活动的最早开始时间和最迟开始时间相同时，这些活动便是**关键活动。**

如果一个顶点有若干条边射入，即说明该顶点表示的事件须当从起点到经由这些边到达该顶点的全部路径上的活动都完成才能发生，因此事件的最早发生时间是最长路径所消耗的时间。

如果一个工程终点的最早时间已知，这个最早时间就是工程需要的总的最短工期，为了达到这个工期目标，可以设定这个时间就是终点事件的最迟发生时间，然后对余下的顶点**倒推**回去，可以获得其余顶点事件的最迟发生时间。

使用邻接矩阵存储时，算法的时间代价是O(n^2)

CH6 查找

6.1 静态查找

内存中的一组数据相对稳定、鲜有变化，就说这组数据是**静态的**，它被称为**静态查找表**。（一般数据采用顺序存储）

6.1.1 顺序查找

常用方法是：把0下标位置设置成一个**哨兵位**，当要查找数据x时，首先将x存入哨兵位，然后从n下标开始向前一直找到0下标。

加哨兵位的好处是避免了逐个检查数组元素时还要首先检测下标是否越界，用浪费一个空间的方式换取时间，提高了效率。

- 时间复杂度O（n）

6.1.2 折半查找

当元素序列有序时，可以使用更高效的查找方法---折半查找法。

- 首先比较中间位置元素，比较成功，查找结束；
- 比较不成功
    1. 如果待查找元素小于中间位置元素，在前半段使用上述同样方法继续查找。特殊地，如果前半段没有了，即长度为0，说明不存在待查找元素
    2. 如果待查找元素大于中间位置元素，在后半段使用上述同样方法继续查找。特殊地，如果后半段没有了，也说明不存在待查找元素。
- 时间复杂度O($log_2 n$)

6.1.3 分块查找

折半查找虽具有好的性能，但要求顺序存储时按照关键字排序，这在元素动态变化时耗时较多。顺序存储虽然也可以应对元素的动态变化，但查找效率低。

综合两者提出了**分块查找**方法。分块查找是把一个大的线性表分解成若干块，块中数据可以无序，任意存放，但块间必须有序。

- 要建立一个索引表，把每块中最大关键字值作为索引表的关键字，从小到大顺序存放在一个辅助数组中。查找时，可以先在索引表中折半查找，确定要找的元素所在的块，然后在块中采用顺序查找，即可找到对应的元素
- 一般适用于数据量很大，无法一次调入内存的情况。

6.2 二叉查找树

6.2.1 二叉查找树定义

对二叉树中的每个结点而言，其左子树上的所有结点都比它小，其右子树上的所有结点都比它大。（为简化起见，假设数据序列中没有元素相等的情况。）

注意是左右子树上的所有结点！不只是左右子结点

6.2.2 二叉查找树查找

递归实现：

```
template <class elemType>
bool binarySearchTree<elemType>::search(const elemType &x, Node<elemType> *t)
const
{    if (!t) return false;
     if (x == t->data) return true;
     if (x < t->data)
        return search(x, t->left);
     else
        return search(x, t->right);
}
```

非递归实现：

```
template <class elemType>
bool binarySearchTree<elemType>::search(const elemType &x) const
{    if (!root) return false;
     Node<elemType> *p;     p = root;
     while (p)
     {   if (x == p->data) return true;
        if (x < p->data) p = p->left;
        else p = p->right;
     }
     return false;
}
```

时间复杂度为树的高度

6.2.3 二叉查找树插入

递归实现：

```
template <class elemType> //递归算法实现
void binarySearchTree<elemType>::insert
(const elemType &x, Node<elemType> *&t) //参数t用了引用形式，完成了新结点的父子关联!!
{   if (!t) { t = new Node<elemType>(x); return; }
    if (x == t->data) return; //已存在，结束插入
    if (x < t->data)
        insert(x, t->left);
    else
        insert(x, t->right);
}
```

非递归实现：

- 如果根为空，创建新结点作为根结点。如果根不为空，设置当前结点为根结点。
- 将待插入元素和当前结点比较，值相等则无需插入；
- 若值小于当前结点，且当前结点无左子，创建新结点作为其左子，否则将其左子作为当前结点，继续比较；
- 若值大于当前结点，且当前结点无右子，创建新结点作为其右子，否则将其右子作为当前结点，继续比较

时间复杂度为树的高度

6.2.4 二叉查找树的删除

递归实现：

- 根为空，删除结束。
- 否则和根结点值比较，相同则实施**三种删除**；
- 不相同但比根结点值小，在以其左子为根的二叉查找树中继续删除；
- 不相同但比根结点值大，在以其右子为根的二叉查找树中继续删除。

三种删除：

1. 待删除结点为叶子。释放待删除结点空间，父子链置为空。
2. 待删除结点有唯一孩子。用唯一孩子替代待删除结点位置，释放待删除结点空间。
3. 待删除结点有两个孩子，找替身结点（左子树上的最大结点；即沿左子一路右子找下去。或右子树上

   上最小结点。即沿右子一路左子找下去）。用替身结点值取代待删除结点值，删去原替身结点，它最多只有一个孩子。问题转化为前两种

```
template <class elemType> //递归算法实现
void binarySearchTree<elemType>::remove(const elemType &x, Node<elemType> *&t)
{
    if (!t) return;
    if (x < t->data)   remove(x, t->left);
    else
        if (x > t->data )  remove(x, t->right);
        else
        {
            if (!t->left && !t->right) //叶子结点
            {    delete t;  //释放待删除结点
                 t = NULL;  //父结点和叶子的链接断开
                 return;
            }

            if (!t->left || !t->right)//只有一个孩子
            {
                Node<elemType> *tmp;
                tmp = t;
                t = (t->left)? t->left : t->right;  //父结点链接其唯一孩子结点
                delete tmp;//释放待删除结点
                return;
            }
            //待删除结点有两个孩子的情况
            Node<elemType> *p, *substitute;
            p = t->right;
            while (p->left) p = p->left;
            substitute = p;

            t->data = substitute->data;
            remove(substitute->data, t->right); //删除原替换结点
        }
}
```

消耗时间最多为树高度的两倍。

6.3 平衡二叉查找树（AVL树）

- 结点的**平衡因子**：一个结点的平衡因子等于其左子树的高度减去其右子树的高度。
- 如果一棵二叉树中所有结点的平衡因子的绝对值不超过1，即为-1，0，1这三种情况，这棵二叉树就称为**平衡二叉树。**
- 平衡二叉树的目标是**限制二叉树的高度。但平衡二叉查找树并不能直接和树高最矮划等号**

> AVL树的查找和普通二叉查找树相同。

6.3.1 AVL树插入

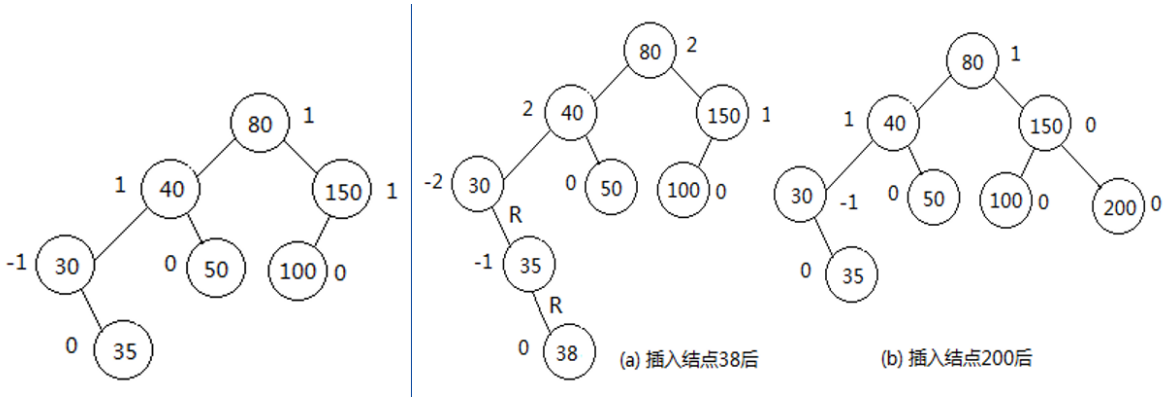
当元素插入到AVL树中时，就可能打破原有的平衡。

新插入结点的平衡因子为0，一路自下而上往祖先结点传导。

- 如果传导来自于左子树，说明左子树高度增加了1，父结点平衡因子加1；
- 如果传导来自于右子树，说明右子树高度增加了1，父结点平衡因子减1。

父结点平衡因子变化后，如果结果变为0，说明原本的左右子树一边高、一边低，现在低的长高了，变得和高的一样了，以父结点为根的子树高度没有变化，自下而上的传导行为停止，祖父包括更上层祖先结点的平衡因子保持不变；

父结点平衡因子变化后，如果结果变为非0，依然按照传导来自左子树加1、右子树减1的原则向祖父结点传导，直到某一层祖先结点的平衡因子变为0、+2或者-2、或者到达根结点。

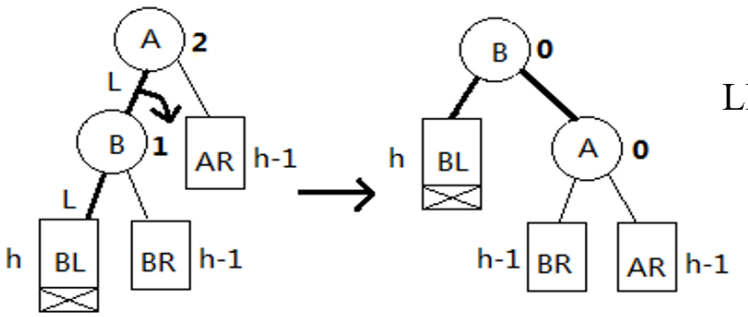


二叉查找树中一旦有一个结点的平衡因子不在-1，0，1的范围内，二叉树就不再平衡。在向上的传导过程中，平衡因子第一个超过-1，0，1范围的结点称为**冲突结点。**

一旦发现冲突结点，暂停沿祖先向上的传导，先对二叉树在冲突结点附近实施调整，直到它变得平衡。

冲突结点和来自插入方向的子结点、孙子结点关系分4种类型

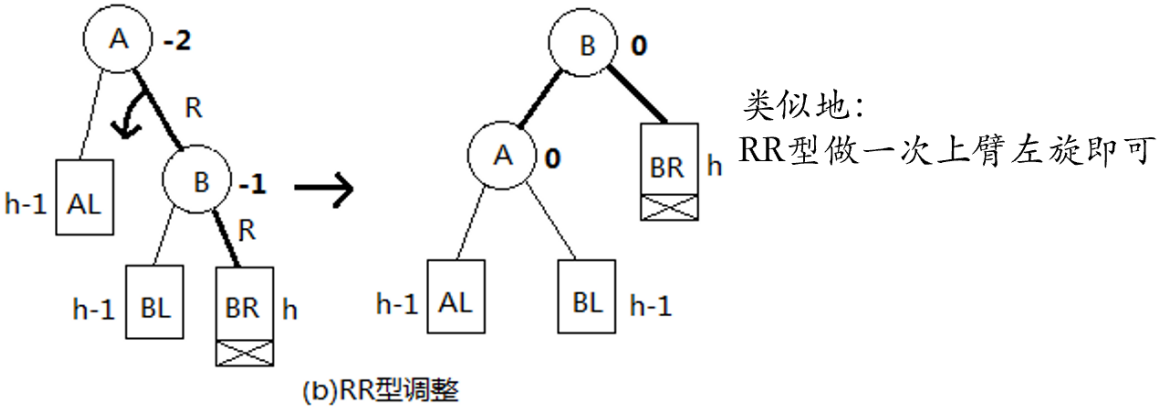
（LL、RR、LR、RL），各自处理方式如下：



LL型做一次上臂右旋即可

(a) LL型调整

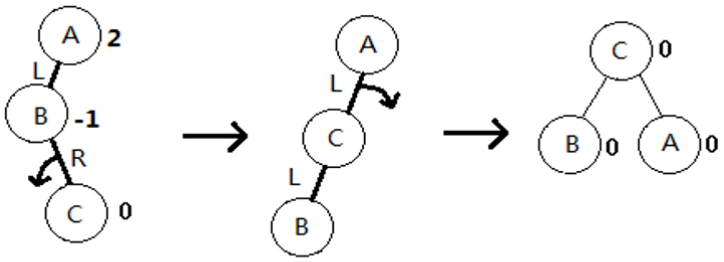
```
template <class KEY, class OTHER>
void AVLTree<KEY, OTHER>::LL(AVLNode *&t)
{
    AVLNode *t1=t->left;
    t->left=t1->right;
    t1->right=t;
    t->height = max(height(t->left), height(t->right))+1;
    t1->height = max(height(t1->left), height(t))+1;
    t=t1;
}
```



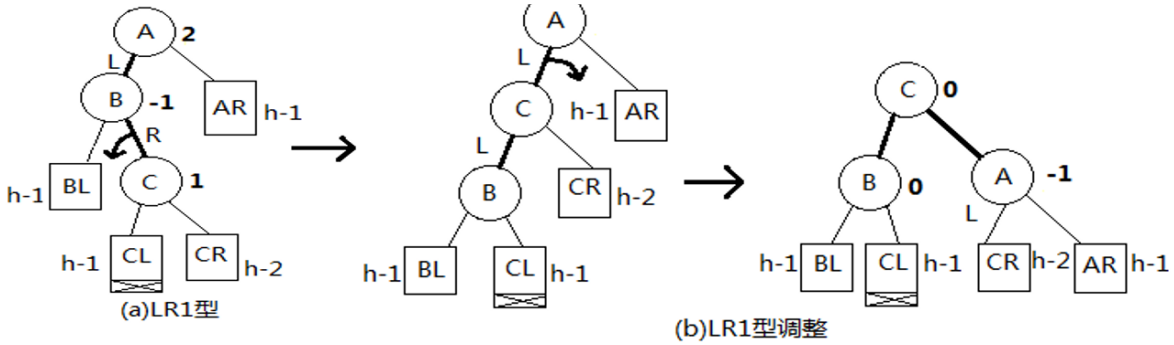
(b)RR型调整

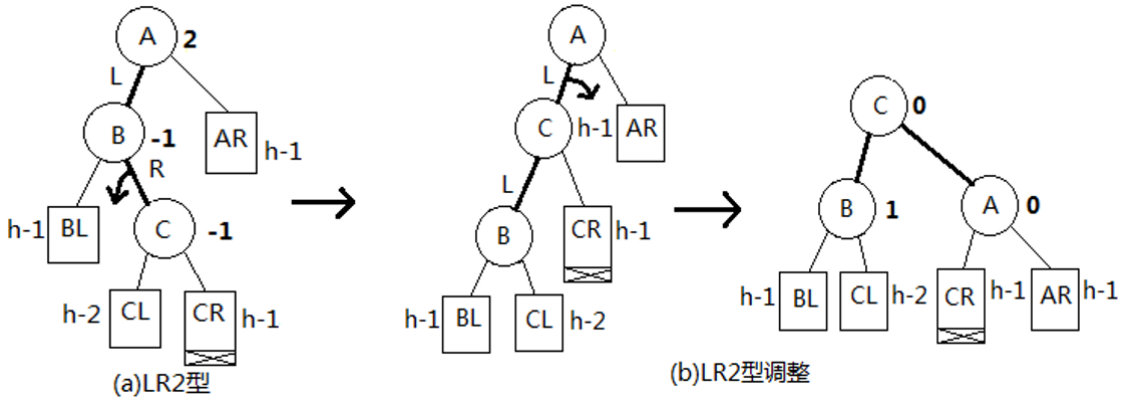
```
template <class KEY, class OTHER>
void AVLTree<KEY, OTHER>::RR(AVLNode *&t)
{
    AVLNode *t1=t->right;
    t->right=t1->left;
    t1->left=t;
    t->height = max(height(t->left), height(t->right))+1;
    t1->height = max(height(t1->right), height(t))+1;
    t=t1;
}
```

- LR型先RR（t->left）再LL（t）



（a）LR0型　　（b）LR0型调整





- RL型先LL（t->right）再RR（t）

6.3.2 AVL树删除

> 不想学了

6.3.3 AVL树高度

一棵由N个结点组成的AVL树，它的高度H小于等于 1.44log(N+1)-0.328

设具有n个结点的二叉平衡树的最大高度为H，则有：F(H+2)-1≤n≤F(H+3)-1，其中F(m)为斐波那契数列{0，1，1，2，3，5，8，13，21，...}。

6.4 B和B+树

- 上面各种查找都是基于数据元素能够全部载入内存来讨论其存储结构和算法的。
- 在实际应用中，数据量往往很大，存储在外存储器上的文件中，数据无法一次性载入内存，数据的存储常是按其写入时间的先后顺序存储。为了加快数据的查找速度，**建立索引文件**是最普遍的做法。
- 索引文件，通常按关键字顺序存储了关键字和所属数据在原始文件中地址的对应关系，查找时首先在索引文件中按关键字查找，找到待查关键字后，通过对应的地址信息，到数据文件就能读取到目

标数据。

6.4.1 B树的定义

B树是一种存储在外存储器中的动态查找表。M阶的B树须满足如下定义：

1.或者为空、或者只有一个根结点、或者除了根还有多个结点。

2.根结点如果有子，则至少有两个儿子、至多有M个儿子。

3.除了根结点外，每个非叶子结点至少有[M/2]（这是向上取整）个儿子，至多有M个儿子。

非叶子结点结构如下：

$$(n, A_0, K_1, R_1, A_1, K_2, R_2, A_2, \cdots, K_n, R_n, A_n)$$

其中：n为结点中关键字的个数，K_i为关键字，R_i为关键字为K_i的数据在原始文件中的地址，A_{i-1}为在树中 关键字值小于K_i的结点的地址，A_i为在树中关键字值大于K_i的结点的地址。n个关键字就意味着该结点有**n+1个孩子**（所以关键字个数[M/2]-1 ~ M-1）

4.叶子结点都在同一层上，且不带任何信息，可以视做空结点、表示查找失败。

6.4.2 B树的查找

- B树每层只查找一个结点，如果B树作为原始数据文件的索引文件驻留在外存储器上，走向下一层时，只需根据指向的地址将B树中的一个结点读入内存，即对应着一次磁盘的访问，因此B树中一个结点的大小通常也取一次磁盘读取的数据量（称一个数据块）。
- 外存储器访问速度比内存访问速度要慢得多，降低B树的层次就能减少读取外存储器的次数。B树因为是多路搜索树，孩子的数量大于2，所以它比二叉查找树高度要少。

6.4.3 B树的插入

- 首先找到最后一层空结点位置，在其父结点处插入一个关键字，如果父结点中关键字个数在插入前已达上限（M-1），就需逐级向上进行结点分裂，直到某层结点中关键字个数少于上限。树的高度，就是在所有结点逐个插入的过程中分裂结点并向上建立新结点形成的
- 首先从根结点逐层向下移动查找，需要比较的次数最差为B树的高度。插入后如果引起结点分裂，最差情况是每一层都分裂、上移，直到根结点。所以总的时间消耗是B树高度的两倍。

6.4.4 B树的删除

在原始数据文件中删除数据，只需要在其B树索引文件中删除该数据关键字。

原始文件中数据可以暂时不做处理，留在后面做定期批量数据维护时清理。

删除首先也要进行查找，查找待删数据关键字在B树中的哪个结点上。然后根据结点的情况，将删除按照以下几种情况分别处理：

- 待删关键字在最下非叶子结点层，再下层就是空结点了。
    - 如果所在结点中原本关键字个数就大于[M/2]：直接删除关键字，并将结点前的关键字个数减1即可。
    - 如果所在结点中原本关键字个数等于[M/2]，且左、右兄弟结点中有孩子个数非最小值：从关键字个数非下限的兄弟处借过来一个。如果从左边兄弟结点处借，借最大关键字；如果从右边兄弟结点处借，借最小值。借来的关键字和父结点的一个关键字交换，将换来的父结点关键字追加到删除关键字的结点中。
    - 如果所在结点中原本关键字个数等于[M/2]，且左、右兄弟结点孩子个数均为最小值：删除关键字的结点和左兄弟或者右兄弟结点合并，将父结点中介于两个合并孩子间的关键字下移加入合并结点。如果父结点中关键字下移后，关键字个数少于[M/2]-1个，调整继续上移。
- 待删的关键字在中间层，下层仍为非叶子结点层。

在B树中找到待删关键字，顺关键字左侧子树中找到最大关键字或者顺着关键字右侧子树中找到最小关键字作为替身替代之，然后转为情况1.

6.4.5 B+树

B树是一棵多线索查找树，所有的数据关键字都挂在了树的非叶子结点中，每个关键字右侧字段（空心箭头）标明了其对应数据记录在原始数据文件中的具体地址，因此根据关键字和作为索引的B树的帮助，就能非常方便地在数据文件中找到它。

B树的缺点也是明显的，因每个关键字都在B树上，造成B树过于庞大；另外如果要按关键字大小顺序访问所有数据，B树没有任何优势。为了解决这个问题，引入了B+树的概念。

M阶的B+树定义如下：

1.或者为空、或者只有一个根结点、或者除了根还有多个结点。

2.每个非叶子结点至多有M个儿子。

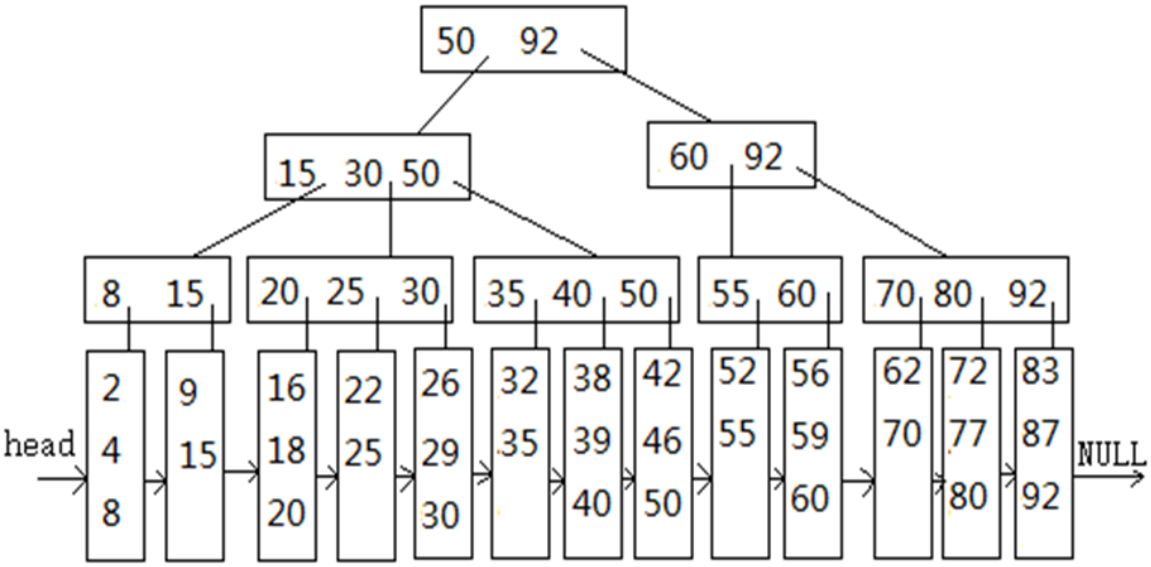
3.除了根结点外，每个非叶子结点至少有[M/2]儿子。

4.非叶子结点结构如下：$(n, K_1, A_1, K_2, A_2, \cdots, K_n, A_n)$

n为关键字个数，A_i为关键字小于等于K_i的孩子结点地址。

显然，有k个孩子的结点有k个关键字，非叶子结点并不包含所有数据的关键字。

5.叶子结点和中间结点不同，它存储了数据关键字和它在原始数据文件中的地址信息。所有叶子包含了全部关键字信息，它的关键字个数最少[M/2]个，最多M个。



时间复杂度比较：

- B树关键字和对应数据记录地址信息可能分布在非叶子结点的任何结点中，所以查找可能停止在任何一层，时间消耗小于等于树的高度，最差为树的高度。
- 而B+树由于关键字和对应数据记录地址关系全部存储在叶子结点中，查找必须从根到达叶子结点，时间消耗一定是树的高度。

B+树便于按序遍历：

用一个指针变量（头指针）保存第一个叶子结点的地址，每个叶子结点尾部增加一个指针，指向右边相邻叶子结点的地址，最右一个叶子结点中的该指针指向空，这样便形成了一个单链表。当需要按照关键字大小顺序遍历原始数据文件中的所有数据时，顺着头指针访问单链表，便能实现原始数据的顺序遍历任务。

6.5 Hash方法

哈希方法希望抛弃比较，具体做法是对数据关键字key，通过一个函数映射H(key)计算出数据在内存中的存储地址，这种方法称为**哈希查找。**

理想的哈希函数是将不同的数据分别映射到不同的地址，查找会达到最理想的时间复杂度O(1)。

- 负载因子越大，空间利用率越高，哈希函数就越难找。
- 负载因子越小，冲突的可能性越小，但空间利用率越低。

好的哈希函数满足计算速度快；哈希到的地址均匀、冲突少；哈希表的负载因子高，尽量减少空间的浪费。

6.5.1 常用的Hash函数

1.直接寻址法： 哈希函数为H(key) = a*key+b。

如元素关键字序列{100，200，330，520，600，815}， 通过函数H(key) = key/100-1，将以上序列映射到{0，1，2，4，5，7}下标分量中。哈希表的大小可以取m=8，负载因子为∂=6/8=0.75。

2.除留余数法：哈希函数为H(key) = key mod p。

通过对关键字除以p取余数计算出地址。如元素关键字序列{35，192，64，5，76，653}， 通过函数H(key) = key mod 7，将以上序列映射到{0，3，1，5，6，2}下标分量中。此例哈希表的大小可以取m=7，负载因子为∂=6/7=0.86。**函数中p通常取大于元素个数n的最小的素数**，因p大于n，能保障元素全部入表，p取素数是为了尽量减少规律性的空间浪费。

3.数据分析法：数据的关键字中如果有一些位上数据分布比较均匀，能区分出不同的数据元素，就可以将这些位取出来作为数据元素的存储地址用。

4.平方取中法：如果关键字分布不均匀，也可以将关键字首先平方，有时平方后的结果中的几位就会变得均匀，这时再用直接定址或者数据分析法获得合适的哈希地址。如关键字136，平方后为18496，取其中的第2、3位得49，49便为关键字136的数据元素的哈希地址。

5.折叠法：当关键字位数相比于数据元素的个数大得多，也可以将其按照哈希表大小分割成若干段，并将这些段相加，得到的和作为哈希地址。

6.5.2 解决冲突

在考虑到空间负载因子的情况下，很难找到一个函数能将所有数据都映射不同的地址上去，势必会存在两个关键字值不同的数据却得到了相同的哈希地址，这就叫**冲突**。一旦发生了冲突，就要采取一定的措施解决冲突了。

冲突时重新计算哈希地址，再计算出的地址仍然在原来的哈希表中，因此称为**闭哈希表**。相反，如果冲突时在原来的哈希表外寻找一个存储地址，则称为**开哈希表**

1.线性探测法：当冲突发生时用(H(key)+i)%m的方法重新定址。其中i为冲突的次数，m为哈希表的大小。

2.二次探测法：当冲突发生时，按照$(H(key)(^{+}_{-})i^2)$解决冲突，第1次冲突，取$(H(key)+1^2)$；第2次冲突，取 $(H(key)-1^2)$；第3次冲突，取$(H(key)+2^2)$，以此类推。

这种方法较线性方法比，位置移动的幅度大，不容易造成二次冲突。

CH7 排序

7.1 概念与效率

稳定与不稳定排序

待排序数据中如果有关键字值相同的元素，经过某种排序算法后其相对先后位置在排序前后没有变化，这种排序称**稳定排序**；反之称**不稳定排序。**

如数据Ri和Rj关键字值相同，排序前Ri在Rj之前，稳定排序后一定保持Ri在Rj之前；，不稳定排序后，有可能变为Ri在Rj之后。

- 冒泡排序、插入排序、归并排序、基数排序都是稳定排序
- 而希尔排序、选择排序、堆排序、（一般认为）快速排序都是不稳定排序。

内排序与外排序

待排序数据可全部一次性载入内存，排序只和内存打交道，在程序中的具体表现就是数据可以全部放入声明的一组变量中，该排序操作称为**内排序。**

如果待排序数据不能一次性全部载入内存，在排序过程中还需要进行内、外存之间的数据交换，在程序中的具体表现是数据只能分批从文件中读入内存变量中，该排序称为**外部排序。**

常用算法时间复杂度

- 冒泡排序、插入排序、选择排序、快速排序的最坏情况下时间复杂度都是O(n^2)
- 特别地在数据原本正序的基础上冒泡排序和插入排序时间复杂度可以达到O(n)
- 基数排序和堆排序在一般情况下时间复杂度就能达到O(n)。
- 归并排序、堆排序最坏情况、快速排序最好情况时间复杂度能达到O(n log_2n)。

各种常用排序算法							
类别	排序方法	时间复杂度			空间复杂度	稳定性	复杂性
		平均情况	最好情况	最坏情况	辅助存储		
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	简单
	希尔排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定	复杂
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定	复杂
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	简单
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$~$O(n)$	不稳定	复杂
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定	复杂
基数排序		$O(d(r+n))$	$O(d(r+n))$	$O(d(r+n))$	$O(rd+n)$	稳定	复杂
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数							

7.2 冒泡排序

基本有序情况下优选（2017-2018-1 T2-8）

```
void bubbleSort(elemType a[], int n)
{  int i, j;   bool  change = true;
   elemType tmp;
   for (j=n-1; j>0&&change; j--)
   {    change = false;
    for (i=0; i<j; i++)
       if (a[i]>a[i+1])
       {   tmp = a[i];  a[i] = a[i+1];
           a[i+1] = tmp;  change = true;
        } //if
}}
```

7.3 插入排序

基本有序情况下优选（2017-2018-1 T2-8）

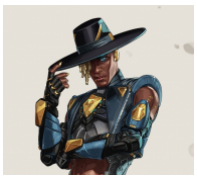
```
template <class elemType>
void insert(elemType a[], int n, const elemType &x)
//n为有序表a中当前元素的个数，x为待插入新元素
{
    int i;
    //从后往前找第一个不比x大的元素，大者后移一位
    for (i=n-1; i>=0; i--)
        if (a[i]<=x) break;
        else a[i+1] = a[i];

    a[i+1] = x; //在腾出的位置上存新元素x
}


template <class elemType>
void insertSort(elemType a[], int n)
{   int i;
    elemType tmp;

    //将第i个元素插入到前i-1个元素的有序序列中
    for (i=1; i<n; i++)
    {   tmp = a[i];   insert(a, i, tmp);  }
}
```

7.4 希尔排序



预处理，使序列比较有序，然后进行插入排序

将原始序列按不同步长分成若干子序列，分别进行插入排序，使序列变比较有序，降低插入排序时间消耗。

```
template <class elemType>
void shellSort(elemType a[], int n)
{   int step, i, j;
     elemType tmp;
     for (step=n/2; step>0; step/=2)
        for (i=step; i<n; i++)
         {   tmp = a[i];
               j = i;
          while ((j-step>=0)
                    &&(tmp<=a[j-step]))
             {   a[j] = a[j-step];
                  j-=step;
             }
             a[j] = tmp;
         }
}
```

值相等的元素在预处理可能分在不同的子序列中，经过在各自子序列中位置的调整，原本的相对前后位置就可能发生改变，因此希尔排序是**不稳定排序。**

7.5 归并排序

- 基于将两个有序序列归并为一个有序序列的方法。

```
template <class elemType>
void merge(elemType a[], int low, int mid, int high)
{    int i, j, k;    elemType *c;
     //创建实际空间存储合并后结果
     c=new elemType[high-low+1];
     i = low;   j = mid+1;   k = 0;
     //两个有序序列中元素的比较合并
     while ((i<=mid)&&(j<=high))
     {     if (a[i]<=a[j])
           {   c[k]=a[i];   i=i+1;  }
            else
           {  c[k]=a[j];   j=j+1; }
           k=k+1;
     }
     //若a序列中i未越界，抄写剩余元素
     while (i<=mid)
     {      c[k]=a[i];     i=i+1;    k=k+1;    }

     //若b序列中j未越界，抄写剩余元素
     while (j<=high)
     {     c[k]=a[j];     j=j+1;    k=k+1;    }

     for (i=0;i<high-low+1; i++)   a[i+low] = c[i];
     delete []c;
}
```

在两两合并算法中，对前后两个有序序列中元素比较时，后者元素大才能胜出，因此值相同的元素在合并中能保持原本的相对前后位置，合并排序是一个**稳定排序。**

7.6 选择排序

选择排序的思想：两两比较为基础。从左到右，为有序序列中每个位置选择合适的元素。

具体为：在下标0-n-1范围找出最小值，换到0下标位置；在下标1-n-1范围找出最小值，换到1下标位置 ... 在下标n-2-n-1范围找出最小值，换到n-2下标位置.

- 直接选择排序中，如选第一个时，是原第一个和选到的元素**位置互换**，不是从原第一个到选到元素后移！

```
template <class elemType>
void selectSort(elemType a[], int n)
{   int i, j,minIndex;      elemType temp;

    for (i=0; i<n; i++)
    {   //为第i个位置找合适的数据
        minIndex = i;
        for (j=i+1; j<n; j++)
```

```
            if (a[j]<a[minIndex])   minIndex = j;
        //将minIndex位置上的数据
        //和位置i上数据交换
        if (minIndex == i) continue;
        temp = a[i];
        a[i] = a[minIndex];
        a[minIndex] = temp;
    }
}
```

7.7 堆排序与堆 ▲

堆的概念:

- 一个完全二叉树中，任意一个结点的值比其左右子结点值都大，称为**大顶堆**。
- 一个完全二叉树中，任意一个结点的值比其左右子结点值都小，称为**小顶堆**。

  大顶堆和小顶堆都称为堆。

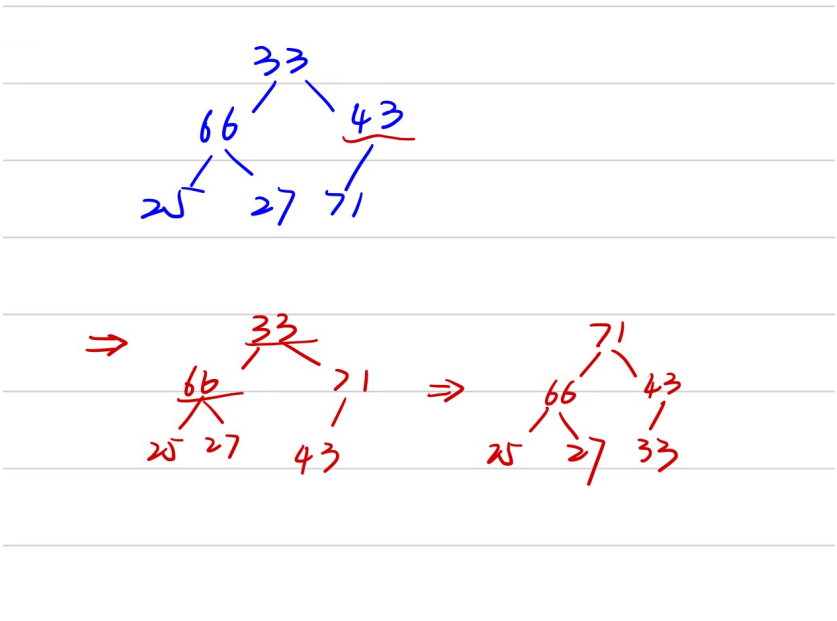
堆排序:

- 将存于数组中的序列看作是一棵完全二叉树的顺序存储。
- 按照堆的概念调整之，使之成为一个大顶堆。
- 摘取大顶，换到待处理元素最后位置
- 继续调整新的根使之满足大顶堆概念，得到次大元素，
- 继续后移，直到序列中元素全部有序。

建堆过程（以非递减排序，建立大顶堆为例）：

- 按顺序从数组建立完全二叉树
- 从第一个非叶子结点开始过滤。（非叶结点是从后往前的）

> 2014-2015-1，T1-12
>
> 从（33，66，43，25，27，71）非递减堆排序建立初始堆：



最大/最小堆插入:

- 先插入到最后，然后再上下调整

时间复杂度:

- 创建一个大小为k堆时间复杂度为O(k)
- 在一个大小为k的堆中添加或删除元素时间复杂度都是O(logk)

7.8 快速排序

```
void qsort(int a[],int start,int end){
        int mid;
        if (start>=end) return; //待排数字只有0或1个时结束递归
        mid=divid(a,start,end);
        qsort(a,start,mid-1);
        qsort(a,mid+1,end);
}

int divid(int a[],int start,int end){ //选数为标志，拆分
int k=a[start];  //书中选择每组第一个数，这样对较乱数据有很好时间效率，但本身有序时时间效率差
    do{
        while (start<end && a[end]>=k) end--;
        if (start<end){a[start]=a[end];start++;}
        while (start<end && a[start]<=k) start++;
        if (start<end){a[end]=a[start];end--;}
    }while (start!=end);
a[start]=k; //置入中间
return start;
}
```

7.9 外排序，置换选择、K路归并

元素个数太多，无法一次性载入内存。

根据内存容量的大小一次调入一定量的数据，形成一个数据序列，该序列在内存中可以按照某种内排序的方法进行排序，然后将排好的序列写入外存，之后再调入其他未排序的数据进入，以此类推。

最终在外存上原始的待排序序列分割成了多个有序序列，之后再设法将数据分段调入内存，进行有序数据段的归并。

置换选择法

用于优化生成初始归并段。对有m个初始归并段的k路归并，其时间消耗为$log_k m$，m越小时间花费越小。

CH8 历年代码题

2013-2014-1

- 图的邻接矩阵转邻接表存储
- 求结点x在二叉树中的双亲结点（伪代码）
- 求以二叉链表表示的二叉树中所有叶子节点数（伪代码）
- 顺序散列表查找关键字x
- 由前序、中序遍历确定二叉树

2014-2015-1

- 二叉查找树删结点x，保持高度不变
- 二分查找（非递归算法）
- 数据进栈
- 将单链表逆序连接
- 在规模为N的无序数组找第k答元素（伪代码，堆）
- DFS寻找顶点间路径

2015-2016-1

- 前序遍历二叉树（非递归）
- 奇偶冒泡交换排序
- 寻找最小化堆最大元素下标
- 删除绝对值相等的重复结点

2017-2018-1

- 深度优先遍历图
- 判断字符串a是否是b的前缀
- 清空二叉查找树/保留以x为根节点子查找树
- 二叉查找树插入（递归）
- 二叉查找树查找（非递归）
- 单链表找倒数第k个
- 双向冒泡排序

2018-2019-1

- 寻找二叉查找树第k大结点
- 拓扑排序
- 奇偶冒泡交换排序（同2015-2016-1）

2020-2021-2

- 输出二叉树结点x全部祖先结点
- 二叉查找树删除节点x（类似2014-2015-1）
- 优先级队列入队
- 输出图中长度M路径
- 字符解密
- 判断二叉树是否为完全二叉树

2021-2022-2

- 二叉查找树插入（非递归）
- 括号匹配
- 单链表寻找元素
- 拓扑排序（同2018-2019-1）
- 二叉查找树查找（递归）
- 求二叉查找树高度（递归）
- 单链表找倒数第k个（同2017-2018-1）