中山大學
SUN YAT-SEN UNIVERSITY

数据结构及算法

笔记整理

姓名：刘斯宇
学号：17341110

目录

☑ **第一章--线性表**

☑ **第二章--栈**

☑ **第三章--队列**

1 线性表

线性表的定义

(1) 由结点集 N, 以及定义在节点集 N 上的线性关系 r 所组成的线性结构，这些结点称为线性表的元素。

(2) 零个或多个数据元素的有限序列。

线性表的数序语言描述

若将线性表记作 $(a_1,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$，则标中 a_{i-1} 领先于 a_i，称 a_{i-1} 是 a_i 的直接前驱元素，s_{i+1} 是 a_i 的直接后继元素。

Properties: 线性表

(1) 结点集 N 中有一个唯一的开始结点，它没有前驱，但有一个唯一的后继；

(2) 对于有限集 N, 它存在一个唯一的终止结点，该结点有一个唯一的前驱而没有后继；

(3) 其它的结点皆称为内部结点，每一个内部结点既有一个唯一的前驱，也有一个唯一的后继；

(4) 线性表所包含的结点个数称为线性表的长度，它是线性表的一个重要参数；长度为 0 的线性表称为空表；

线性表的分类

- 线性表的顺序存储结构，指的是用一段地址连续的存储单元依次存储线性表的数据元素
- 线性表的链式存储结构，指的是用一组任意的存储单元存储线性表的数据元素，这组存储单元可以是连续的，也可以是不连续的。

1.1 顺序表

特点

逻辑上相邻的元素，在物理位置上也相邻。

1.1.1 函数的实现

插入 insert

```
1 int InsertElem(sqlist *L,DataType x,int i)
```

Algorithm 1 InsertElem(i,e)

Input: i: location;e : ElementType;$1 \leq i \leq L.last + 1$
Output: Insert element e at location i and add 1 to Length of L

```
{
    int j;
    if (((*L).last)>=maxsize - 1)
    {
        printf( “overflow\n” );
        return NULL;
    } //溢出
    else
        if((i<1)||(i>((*L).last)+1)
        {
            printf( “error\n” );
            return NULL;
        } //非法位置
        else
        {
            for (j=(*L).last;j>=i;j--)
                (*L).data[j+1]=(*L).data[j];
                (*L).data[i]=x;
                (*L).last=(*L).last+1;
        }
    return(1);
}
```

复杂度分析

最坏：i=1，移动次数为 n

最好：i= 表长 +1，移动次数为 0

平均：等概率的情况下，平均移动次数为 $\frac{0+1+2+\cdots+n}{n+1} = \frac{n}{2}$

删除 delete

Algorithm 2 DeleteElem(i)

Input: i: location;$1 \leq i \leq L.last+1$
Output: delete the element at location i.

```
int DeleteElem(sqlist *L,int i)
{
    int j;
    if ((i<1)||(i>(*L).last+1))
    {
        printf("error\n");
        return NULL;
    } //非法位置
    else
    {
        for(j=i;j<=(*L).last;j++)
            (*L).data[j-1]=(*L).data[j];
        (*L).last--;//表长减1
    }
    return(1);
}
```

复杂度分析

最坏：i=1，移动次数为 n-1

最好：i= 表长，移动次数为 0

平均：等概率的情况下，平均移动次数为 $\frac{0+1+2+\cdots+n-1}{n}=\frac{n-1}{2}$

按值查找

```
int LocateElem_e(SqList L, DataType e)
{
    i=1;
```

Algorithm 3 LocateElem_e(e)

Input: element e;L is not empty;If e is in List L then return location i.
Output: location of element e;

```
    while ( i<=L.last && e != L.data[i-1])
        ++i;
    if (i<=L.last)
        return i;
    else
        return 0;

    }
```

按位置查找

Algorithm 4 LocateElem_e(e)

Input: location i;$1 \leq i \leq L.last+1$
Output: the element at location i.

```
DataType LocateElem_i(SqList L, int i)
{
    j=1;
    If ((i<1)||(i>((*L).last))
    {
        printf( "error\n" );
        return NULL;
    } //非法位置

    return (*L).data[i];

}
```

顺序表的优缺点比较

优点

顺序表的结构简单

顺序表的存储效率高，是紧凑结构，无须为表示节点间的逻辑关系而增加额外的存储空间

顺序表是一个直接存取结构（随机存储结构）

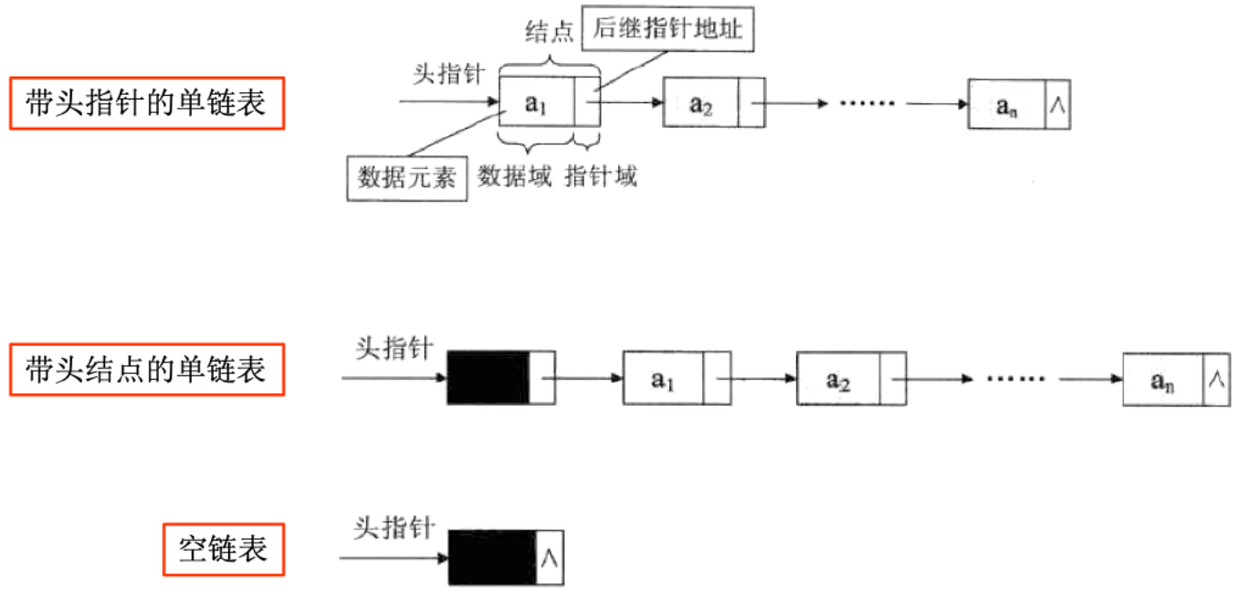
缺点

在顺序表中进行插入和删除操作时，需要移动数据元素，算法效率较低。

对长度变化较大的线性表，或者要预先分配较大空间或者要经常扩充线性表，给操作带来不方便。

1.2 链表

链表的几种表达形式




```
template <class T> class Link {
    public:
        T data;// 用于保存结点元素的内容
```

```
4         Link<T> *next;// 指向后继结点的指针
5         Link(const T info, const Link<T>* nextValue = NULL)
6         {
7             data = info;
8             next = nextValue;
9         }
10        Link(const Link<T>* nextValue)
11        {
12            next = nextValue;
13        }
14};
```

1.2.1 函数的实现

按值查找

```
1  LNode *Locate_Node(LNode *L, int key)
2  /* 在以L为头结点的单链表中查找值为key的第一个结点 */
3  {
4      LNode *p=L->next;
5      while ( p!=NULL&& p->data!=key)
6          p=p->next;
7      if (p==NULL)
8      {
9          printf("所要查找的结点不存在!!\n");
10         return NULL;
11     }
12     return p;
13
14 }
```

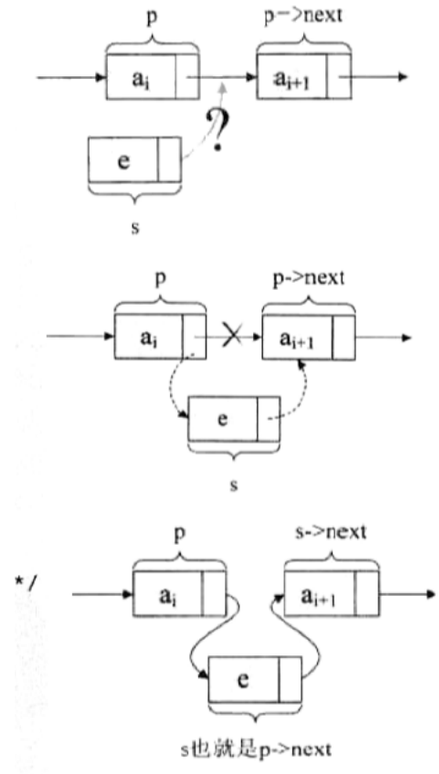
> **Tip** 算法的执行与形参 key 有关，平均时间复杂度为 O(n)。

插入

Algorithm 5 链表的插入

Input: i: 向单链表中插入第 i 个数据
Output: 插入好的单链表

1: 声明一结点 p 指向链表第一个结点，初始化 j 从 1 开始
2: 当 j<i 的时候遍历链表，让 p 的指针向后移动，不断指向下一结点，j 累加 1
3: 若到链表末尾 p 为空，则说明第 i 个元素不存在。
4: 否则查找成功，在系统中生成一个空结点 s
5: 将数据元素 e 赋值给 s->data
6: 单链表的插入标准语句 s->next=p->next;p->next=s;
return 成功;




```
1    bool LinkListInsert(LinkList&L,int i,int e)
2    {
3        int j=0;
4        LinkList p,s;
5        p=L->next;
```

```
6           while(p)
7           {
8               j++;
9               if(j<i-1)
10                  p=p->next;
11              else
12                  break;
13          }
14          if(j>i-1||!p)
15          return false;
16          s=(LNode*)malloc(sizeof(LNode));
17          s->data=e;
18          s->next=p->next;
19          p->next=s;
20          return true;
21      }
```

结点的删除

顺序表的优缺点比较

按序号删除：删除单链表中的第 i 个结点。

为了删除第 i 个结点 a_i，必须找到结点的存储地址。该存储地址是在其直接前趋结点 a_{i-1} 的 next 域中，因此，必须首先找到 a_{i-1} 的存储位置 p，然后令 p->next 指向 a_i 的直接后继结点，即把 a_i 从链上摘下。最后释放结点 a_i 的空间，将其归还给“存储池”。

按值删除

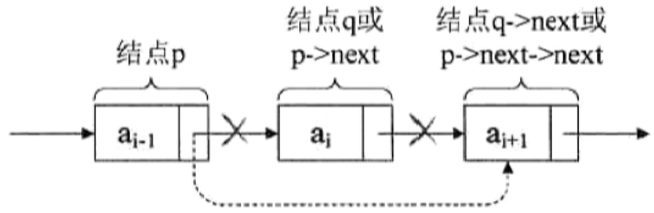
删除单链表中值为 key 的第一个结点。

与按值查找相类似，首先要查找值为 key 的结点是否存在？若存在，则删除；否则返回 NULL。

Algorithm 6 链表的删除

Input: i: 单链表需要删除的第 i 个数据
Output: 插入好的单链表

1: 声明一结点 p 指向链表第一个结点，初始化 j 从 1 开始
2: 当 j<i 的时候遍历链表，让 p 的指针向后移动，不断指向下一结点，j 累加 1
3: 若到链表末尾 p 为空，则说明第 i 个元素不存在。
4: 否则查找成功，将预删除的结点 p->next 赋值给 q
5: 单链表的删除标准语句 p->next=q->next
6: 释放 q 结点
return 成功;




```
1
2     bool LinklistDelete(LinkList&L,int i,int &e)
3     {
4         LinkList p,q;
5         int j=0;
6         p=L->next;
7         while(p)
8         {
9             j++;
10            if(j<i-1)
11                p=p->next;
12            else
13                break;
14        }
15        if(j>i-1||!p)
16            return false;
17        q=p->next;
18        p->next=q->next;
```

```
19          e=q->data;
20          free(q);
21          return true;
22
23      }
```

复杂度分析

设单链表的长度为 n

最坏：i=n+1，虽然不存在，但是其前驱是存在的。

平均：等概率的情况下，平均移动次数为 $\frac{0+1+2+\cdots+n}{n+1} = \frac{n}{2}$

链表的合并

```
1  LNode *Merge_LinkList(LNode *La, LNode *Lb)
2  /* 合并以La, Lb为头结点的两个有序单链表 */
3      {
4      LNode *Lc, *pa , *pb , *pc, *ptr ;
5      Lc=La ; pc=La ;
6      pa=La->next ; pb=Lb->next ;
7      while (pa!=NULL && pb!=NULL)
8      {
9          if (pa->data < pb->data)
10         {
11             pc->next=pa ;
12             pc=pa ;
13             pa=pa->next ;
14         }
15     /* 将pa所指的结点合并, pa指向下一个结点 */
16
17         if (pa->data > pb->data)
18         {
```

```
19              pc->next=pb ;
20              pc=pb ;
21              pb=pb->next ;
22          }
23      /* 将pa所指的结点合并，pa指向下一个结点 */
24      if (pa->data == pb->data)
25      {
26          pc->next=pa ;
27          pc=pa ;
28          pa=pa->next ;
29          ptr=pb ;
30          pb=pb->next ;
31          free(ptr) ;
32      }
33  /* 将pa所指的结点合并，pb所指结点删除 */
34
35      }
36      if (pa!=NULL)
37          pc->next=pa ;
38      else
39          pc->next=pb ;
40      free(Lb) ;
41      return(Lc) ;
42      /*将剩余的结点链上*/
43  }
```

Tip 若 La，Lb 两个链表的长度分别是 m，n，则链表合并的时间复杂度为 O(m+n)。

链表的优缺点比较

优点

插入、删除操作方便，且比较快，不会导致元素的移动，因为元素增减，只需要调整指针。

可动态添加删除、大小可变

链表各个结点在内存空间不需要连续，而且有多少数据就用多少内存，空间空间利用率高。

缺点

查找不方便，不适合随机查找，访问结点效率低，只能从头结点依次访问

链表 VS 顺序表

顺序表不适用的场合

经常插入删除时，不宜使用顺序表

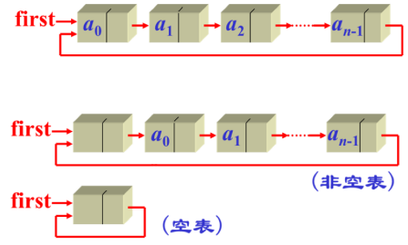
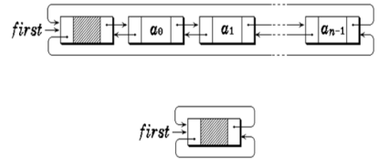
线性表的最大长度也是一个重要因素

链表不适用的场合

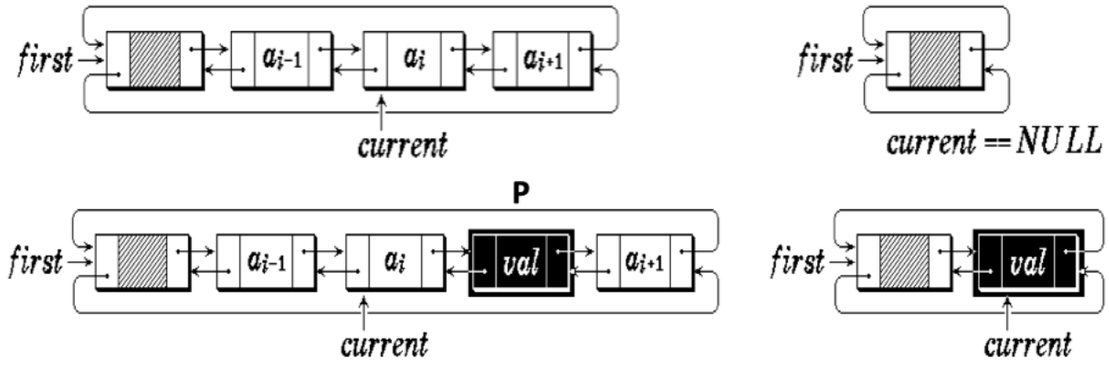
当读操作比插入删除操作频率大时，不应选择链表

当指针的存储开销，和整个结点内容所占空间相比其比例较大时，应该慎重选择

其他类型的链表

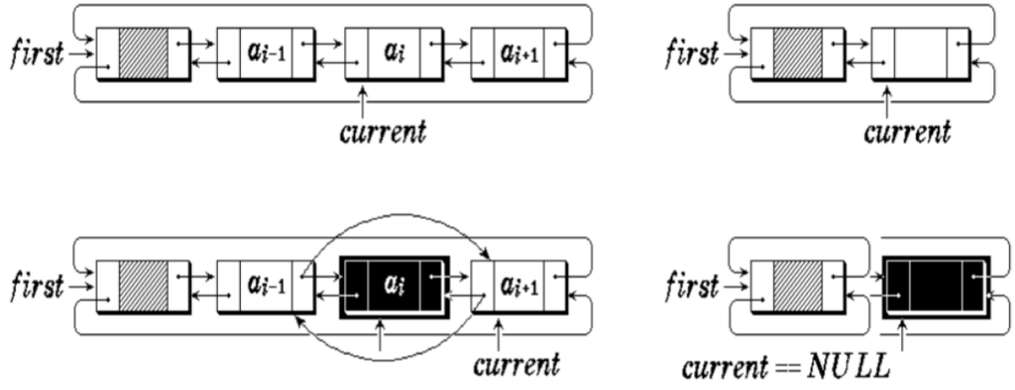
种类	表示
循环链表	
双向链表	

双向链表的插入




```
1    p->prior = current;
2    p->next = current->next;
3    current->next = p;
4    p->next->prior = p;
```

双向链表的删除




```
1    current->next->prior = current->prior;
2    current->prior->next = current->next;
```

1.3 作业整理

例题 1

表所表示的元素是否有序? 如有序，则有序性体现于何处? 链表所表示的元素是否一定要在物理上是相邻的? 有序表的有序性又如何理解?

解答

链表所表示的元素有序; 体现在逻辑上的有序; 链表所表示的元素不一定要在物理上是相邻; 链表的有序性并不是通过物理上的有序来实现的，而是通过逻辑上的有序来实现的，就是通过一个指针来实现其有序性。

2 栈

栈的定义

限定仅在表尾进行插入和删除操作的线性表

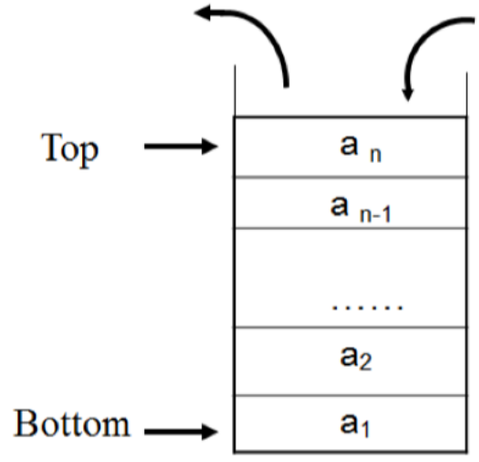
后进先出 (LIFO)

按照存储表示方法的不同，栈有两种实现--顺序栈和链式栈。

2.1 顺序栈

顺序栈的定义

栈的顺序存储结构是利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针 top 只是栈顶元素在顺序栈中的位置。这里以 top=-1 表示空栈。top=maxsize-1 表示栈已经满了。



栈的类抽象

```
Const int maxstack = 10;
Class Stack {
public:
    Stack();
    bool empty() const;
```

```
    Error_code pop();
    Error_code top(Stack_entry &item) const;
    Error_code push(const Stack_entry &item);
private:
    int count;
    Stackentry entry[maxstack];

}
```

压栈操作 push

```
bool arrStack<T>::push(const T item) {
    if (top == mSize-1) // 栈已满
    {
        cout << "栈满溢出" << endl;
        return false;
    }
    else // 新元素入栈并修改栈顶指针
    {
        st[++top] = item;
        return true;
    }
}
```

出栈操作 pop

```
bool arrStack<T>::pop(T & item) { // 出栈的顺序实现
    if (top == -1) // 栈为空
    {
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
```

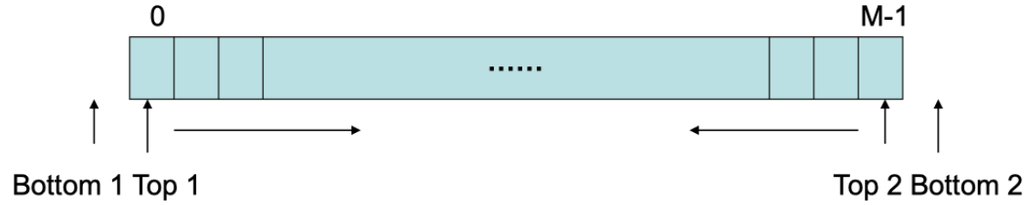
```
6       }
7       else // 返回栈顶元素并修改栈顶指针
8       {
9           item = st[top--];
10          return true;
11      }
12  }
```

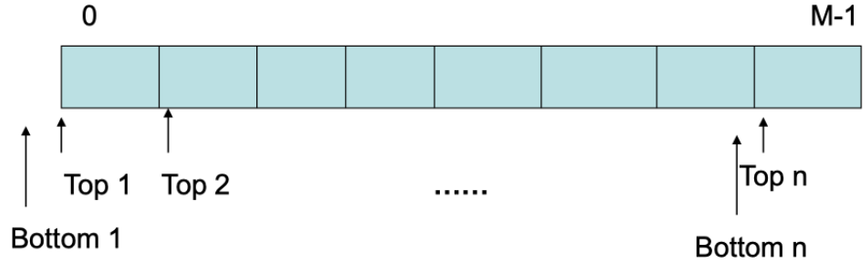
2.1.1 共享栈

共享栈的定义

两个或多个栈使用同一段存储空间。

咦两个栈为例子，第一个栈从数组头开始存储，第二个栈从数组尾开始，两个栈向中间拓展。



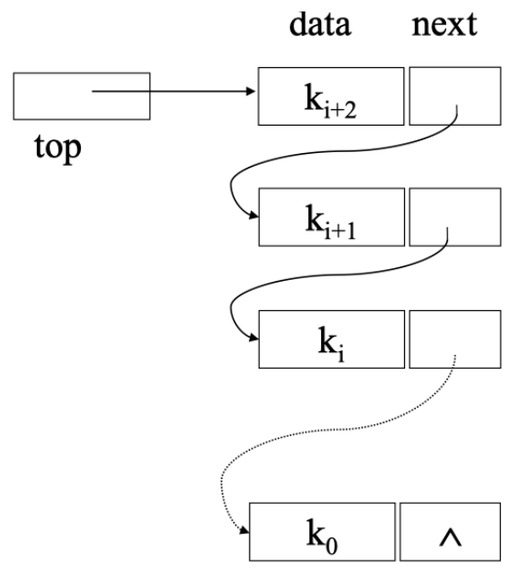


2.2 链式栈

链式栈的定义

用单链表方式存储的栈

指针的方向从栈顶向下链接



压栈操作

```
bool lnksStack<T>:: push(const T item) {
    Link<T>* tmp = new Link<T>(item, top);
    top = tmp;
    size++;
    return ture
}
```

出栈操作

```
bool lnkStack<T>:: pop(T& item) {
    Link <T> *tmp;
    if (size == 0) {
        cout << "栈为空, 不能执行出栈操作"<< endl;
        return false;
        }
        item = top->data;
        tmp = top->next;
```

```
9           delete top;
10          top = tmp;
11          size--;
12          return true;
13  }
```

顺序栈 VS 链式栈

时间效率

所有操作都只需常数时间

顺序栈和链式栈在时间效率上难分伯仲

空间效率

顺序栈须说明一个固定的长度

链式栈的长度可变，但增加结构性开销

3 队列

队列的定义

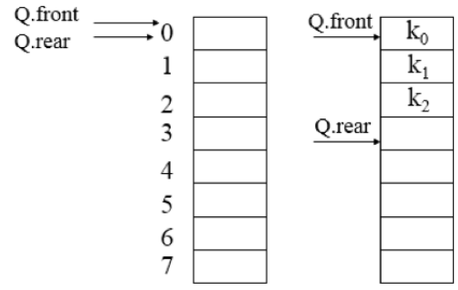
队列是一种限制访问点的线性表。所有的插入在表的一端进行，所有的删除都在表的另一端进行，按照到达的顺序来释放元素。按照存储形式分为顺序队列和链式队列两种。

3.1 顺序队列

顺序队列的定义

使用顺序表来实现队列。

在非空队列里，队首指针始终指向队头元素，而队尾指针始终指向队尾元素的下一位置。



队列空

Properties: 顺序队列

入队：rear = rear + 1

出队：front = front + 1

队空：rear = front

队满：rear-front = m

3.1.1 函数实现

入队

```
1 void Queue:EnQueue(ValueType item)
2 {
3     //判队列满，否则队列溢出异常，退出运行
```

```
    assert(full());
    curr_len++;
    //在队列尾端加入队列
    Qlist[curr_len]=item;
    rear=(rear+1)%maxsize;
}
```

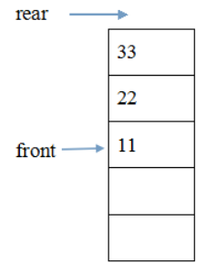
出队

```
void Queue:EnQueue()
{
    ValueType item;
    //判断队列非空，否则队列已空，异常退出运行
    assert(empty());
    temp=Qlist[front];
    curr_len--;
    front=(front+1)%maxsize;
    return temp;
}
```

顺序队列存在的问题

顺序队列中存在“假溢出”现象。因为在入队和出队操作中，头、尾指针只增加不减小，致使被删除元素的空间永远无法重新利用。

如下图所示，队列中明明还有空间，却因为 rear 已经指向最后，造成无法入队问题，这是假溢出。



3.1.2 循环队列

循环队列的定义

为充分利用向量空间，克服“假溢出”现象的方法是：将为队列分配的向量空间看成为一个首尾相接的圆环，并称这种队列为循环队列 (Circular Queue)。

Properties: 循环队列

循环队列为空：front = rear

循环队列满：(rear+1)%maxsize = front

无法通过 front=rear 来判断队列“空”还是“满”

循环队列 VS 顺序队列

队列主要是用于保存中间数据，而且保存的数据满足先产生先处理的特点。非循环队列可能存在数据假溢出，即队列中还有空间，可是队满的条件却成立了。为此，改为循环队列，这样克服了假溢出。

但并不能说循环队列一定优于非循环队列，因为循环队列中出队元素的空间可能被后来进队的元素占用，如果算法要求在队列操作结束后利用进队的所有元素实现某种功能，这样循环队列就不适合了，这种情况下需要使用非循环队列。

4 串

串的定义

字符串 (String) 是零个或多个字符组成的有限序列。

串中所包含的字符个数称为该串的长度。长度为零的串称为空串 (Empty String)，它不包含任何字符。

4.1 KMP 算法

KMP 的算法流程

Algorithm 7 Framework of ensemble learning for our system.

Input: 假设现在文本串 S 匹配到 i 位置，模式串 P 匹配到 j 位置

1: 如果 j = -1，或者当前字符匹配成功（即 S[i] == P[j]），都令 i++，j++，继续匹配下一个字符；

2: 如果 j != -1，且当前字符匹配失败（即 S[i] != P[j]），则令 i 不变，j = next[j]。此举意味着失配时，模式串 P 相对于文本串 S 向右移动了 j - next [j] 位。(换言之，当匹配失败时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的 next 值，即移动的实际位数为：j - next[j]，且此值大于等于 1。)

具体流程

- 寻找前缀后缀最长公共元素长度

Tip 对于 $P = p_0p_1...p_{j-1}p_j$，寻找模式串 P 中长度最大且相等的前缀和后缀。如果存在 $p_0p_1...p_{k-1}p_k = p_{j-k}p_{j-k+1}...p_{j-1}p_j$，那么在包含 p_j 的模式串中有最大长度为 k+1 的相同前缀后缀。

模式串	a	b	a	b
最大前缀后缀公共元素长度	0	0	1	2

- 求 next 数组

Tip next 数组考虑的是除当前字符外的最长相同前缀后缀，所以通过第 1 步骤求得各个前缀后缀的公共元素的最大长度后，只要稍作变形即可：将第 1 步骤中求得的值整体右移一位，然后初值赋为-1

模式串	a	b	a	b
next数组	-1	0	0	1

- 根据 next 数组进行匹配

> **Tip** 匹配失配，j = next [j]，模式串向右移动的位数为：j - next[j]。换言之，当模式串的后缀 $p_{j-k}p_{j-k+1},...,p_{j-1}$ 跟文本串 $s_{i-k}s_{i-k+1},...,s_{i-1}$ 匹配成功，但 p_j 跟 s_i 匹配失败时，因为 next[j] = k，相当于在不包含 p_j 的模式串中有最大长度为 k 的相同前缀后缀，即 $p_0p_1...p_{k-1} = p_{j-k}p_{j-k+1}...p_{j-1}$，故令 j = next[j]，从而让模式串右移 j - next[j] 位，使得模式串的前缀 $p_0p_1,...,p_{k-1}$ 对应着文本串 $s_{i-k}s_{i-k+1},...,s_{i-1}$，而后让 p_k 跟 s_i 继续匹配。

例题 2

给定文本串“BBC ABCDAB ABCDABCDABDE”，和模式串“ABCDABD”，用 KMP 算法给出详细的过程。

解答

- 求出模式串的 next 数组

模式串	A	B	C	D	A	B	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

- 最开始匹配时

```
BBC ABCDAB ABCDABCDABDE
    ABCDABD
```

- P[1] 跟 S[5] 匹配成功，P[2] 跟 S[6] 也匹配成功, ...，直到当匹配到字符 D 时失配（即 S[10] != P[6]），由于 j 从 0 开始计数，故数到失配的字符 D 时 j 为 6，且字符 D 对应的 next 值为 2，所以向右移动的位数为：j - next[j] = 6 - 2 =4 位

```
BBC ABCDAB ABCDABCDABDE
    ABCDABD
```

- 向右移动 4 位后，C 再次失配，向右移动：j - next[j] = 2 - 0 = 2 位

```
BBC ABCDAB ABCDABCDABDE
      ABCDABD
```

- 移动两位之后，A 跟空格不匹配，再次后移 1 位

```
BBC ABCDAB ABCDABCDABDE
        ABCDABD
```

- D 处失配，向右移动 j - next[j] = 6 - 2 = 4 位

```
BBC ABCDAB ABCDABCDABDE
          ABCDABD
```

- 匹配成功，过程结束。

```
BBC ABCDAB ABCDABCDABDE
              ABCDABD
```

4.1.1 算法的实现

如何求 next 数组呢?

```
void GetNext(char* p,int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀, p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++k;
            ++j;
            next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
}
```

```
int KmpSearch(char* s, char* p)
{
        int i = 0;
        int j = 0;
        int sLen = strlen(s);
        int pLen = strlen(p);
        while (i < sLen && j < pLen)
        {
```

```
9                   //如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），
                        都令i++，j++
10                  if (j == -1 || s[i] == p[j])
11                  {
12                          i++;
13                          j++;
14                  }
15                  else
16                  {
17                          //如果j != -1，且当前字符匹配失败（即S[i] != P[
                                j]），则令 i 不变，j = next[j]
18                          //next[j]即为j所对应的next值
19                          j = next[j];
20                  }
21          }
22          if (j == pLen)
23                  return i - j;
24          else
25                  return -1;
26  }
```

5 数组

数组的定义

数组是一组偶对 (下标值，数据元素值) 的集合。在数组中，对于一组有意义的下标，都存在一个与其对应的值。一维数组对应着一个下标值，二维数组对应着两个下标值，如此类推。数组是由 n(n>1) 个具有相同数据类型的数据元素 $a_1, a_2, \ldots, a_n$ 组成的有序序列，且该序列必须存储在一块地址连续的存储单元中。

- 数组中的数据元素具有相同数据类型。
- 数组是一种随机存取结构，给定一组下标，就可以访问与其对应的数据元素。
- 数组中的数据元素个数是固定的。

数组的顺序表示和实现

行优先顺序 (Row Major Order) ——将数组元素按行排列，第 i+1 个行向量紧接在第 i 个行向量后面。

第 m 行中的每个元素对应的 (首) 地址是：$\mathrm{LOC}\left[a_{mj}\right] = \mathrm{LOC}\left[a_{11}\right] + (m-1) \times n \times l + (j-1) \times l \quad j = 1, 2, \ldots, n$

对 n 维数组 A=($a_{j1j_2\ldots j_n}$)，若每个元素占用的存储单元数为 l(个)

$$\begin{aligned}\mathrm{LOC}\left[a_{j_1 j_2 \ldots j_n}\right] = \mathrm{LOC}\left[a_{11\ldots1}\right] + [&(b_2 \times \ldots \times b_n) \times (j_1 - 1) \\ &+ (b_3 \times \ldots \times b_n) \times (j_2 - 1) + \ldots \\ &+ b_n \times (j_{n-1} - 1) + (j_n - 1)] \times l\end{aligned}$$

列优先顺序 (Column Major Order) ——将数组元素按列向量排列，第 j+1 个列向量紧接在第 j 个列向量之后

例题 3). 数组

二维数组 M 的成员是 6 个字符（每个字符占一个存储单元）组成的字符串，行下标 i 的范围从 0 到 8，列下标 j 的范围从 0 到 9。若 M 按行优先方式存储时，元素 M[8][5] 的起始地址与当 M 按列优先方式存储时的哪个元素的起始地址一致?

解答

- 按行优先方式存储时，M[8][5] 的存储地址为 M + (8 x 10 + 5) x 6 = M + 510，
- M[i][j] 按列优先存储时的地址为 M + (i + j x 9) x 6，当 i = 4, j = 9 时，其值为 M + 510。

5.1 特殊矩阵的存储

5.1.1 对称矩阵

对称矩阵的定义

对一个 n 阶方阵 $A=(a_{ij})_{n*n}$ 中的元素满足性质：

$$a_{ij}=a_{ji} \quad 1\le i,j\le n$$

例子：

$$A=\begin{pmatrix} 1 & 5 & 1 & 3 & 7 \\ 5 & 0 & 8 & 0 & 0 \\ 1 & 8 & 9 & 2 & 6 \\ 3 & 0 & 2 & 5 & 1 \\ 7 & 0 & 6 & 1 & 3 \end{pmatrix} \qquad A=\begin{pmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \cdots & \cdots & \cdots & \cdots & \\ a_{n1} & a_{n2} & \cdots & & a_{nn} \end{pmatrix}$$

按照行优先的排列方式，对称矩阵的压缩存储为

K	1	2	3	4	...	n(n-1)/2	...	n(n+1)/2
sa	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	...	a_{n1}	a_{n2}	...	a_{nn}

对称矩阵元素 a_{ij} 保存在向量 sa 中时的下标值 k 与 (i,j) 之间的对应关系是：

$$\mathrm{k}=\begin{cases} \mathrm{i}\times(\mathrm{i}-1)/2+\mathrm{j} & i\ge j \\ \mathrm{j}\times(\mathrm{j}-1)/2+\mathrm{i} & i<j \end{cases} \quad 1\leqq \mathrm{i},\mathrm{j}\leqq \mathrm{n}$$

5.1.2 三角矩阵

三角矩阵的定义

以主对角线划分，三角矩阵有上三角和下三角两种。
上三角矩阵的下三角（不包括主对角线）中的元素均为常数 c (一般为 0)。下三角矩阵正好相反，它的主对角线上方均为常数。

例子：

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ c & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \\ c & c & \cdots & a_{nn} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & c & \cdots & c \\ a_{21} & a_{22} & \cdots & c \\ \cdots & \cdots & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

5.1.3 对角矩阵

三角矩阵的定义

矩阵中，除了主对角线和主对角线上或下方若干条对角线上的元素之外，其余元素皆为零。即所有的非零元素集中在以主对角线为了中心的带状区域中

例子：

$$A=\begin{pmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & \cdots & 0 \\ & \cdots & \cdots & \cdots & \cdots & \\ 0 & \cdots & 0 & a_{n-1\,n-2} & a_{n-1\,n-1} & a_{n-1\,n} \\ 0 & \cdots & 0 & 0 & a_{n\,n-1} & a_{n\,n} \end{pmatrix}$$

Properties: 对角矩阵

- 当 $|i-j| > (k-1)/2$ 时，$a_{ij} = 0$

按照行优先的排列方式，对称矩阵的压缩存储为

K	1	2	3	4	5	6	7	8	…	3n-3	3n-2
sa	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	…	$a_{n\ n-1}$	a_{nn}

5.1.4 稀疏矩阵

三角矩阵的定义

设矩阵 A 是一个 num 的矩阵中有 s 个非零元素，设 $\delta = s/(num)$，称 为稀疏因子，如果某一矩阵的稀疏因子 δ 满足 $\delta \leq 0.05$ 时称为稀疏矩阵。

例子：

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 24 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵的表示

稀疏矩阵可以用三元组来表示，矩阵中的每个元素都是由行序号和列序号唯一确定的。因此，我们需要用三项内容表示稀疏矩阵中的每个非零元素，即形式为：(i,j,value)，其中，i 表示行序号，j 表示列序号，value 表示非零元素的值，通常将它称为三元组。

例子：

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 24 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(1,2,12), (1,3,9),
(3,1,-3), (3,8,4),
(4,3,24), (5,2,18),
(6,7,-7), (7,4,-6)

稀疏矩阵的转置

- 将矩阵的行、列下标值交换。即将三元组表中的行、列位置值 i 、j 相互交换；
- 重排三元组表中元素的顺序。即交换后仍然是按行优先顺序排序的。

暴力算法求解

```
void TransMatrix(TMatrix a , TMatrix b)
    {  int p , q , col ;
       b.rn=a.cn ;  b.cn=a.rn ;  b.tn=a.tn ;
           /*   置三元组表b.data的行、列数和非0元素个数  */
       if  (b.tn==0)   printf(" The Matrix A=0\n" );
       else
           {  q=0;
              for  (col=1; col<=a.cn ; col++)
                 /*  每循环一次找到转置后的一个三元组  */
                 for  (p=0 ;p<a.tn ; p++)
                 /*  循环次数是非0元素个数  */
                 if  (a.data[p].col==col)
                  {  b.data[q].row=a.data[p].col ;
                     b.data[q].col=a.data[p].row ;
                     b.data[q].value=a.data[p].value;
                      q++ ;
                   }
           }
    }
```

算法分析：

本算法主要的工作是在p和col的两个循环中完成的，故算法的时间复杂度为O(cn x tn)，即矩阵的列数和非0元素的个数的乘积成正比。

稀疏矩阵的快速转置

- 直接按照稀疏矩阵 A 的三元组表 a.data 的次序依次顺序转换，并将转换后的三元组放置于三元组表 b.data 的恰当位置。
- 若能预先确定原矩阵 A 中每一列的 (即 B 中每一行) 第一个非 0 元素在 b.data 中应有的位置，则在作转置时就可直接放在 b.data 中恰当的位置。因此，应先求得 A 中每一列的非 0 元素个数。
- 附设两个辅助向量 num[ ] 和 cpot[ ]
  - num[col]：统计 A 中第 col 列中非 0 元素的个数；
  - cpot[col]：指示 A 中第一个非 0 元素在 b.data 中的恰当位置。

$$\begin{cases} \text{cpot}[1] = 1 \\ \text{cpot}[\text{col}] = \text{cpot}[\text{col}-1] + \text{num}[\text{col}-1] \quad 2 \leqq \text{col} \leqq \text{a.cn} \end{cases}$$

例子：

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 24 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵示例

num[col]和cpot[col]的值表

col	1	2	3	4	5	6	7	8
num[col]	1	2	2	1	0	1	1	1
cpot[col]	1	2	4	6	6	7	8	9

例题 4

用稀疏矩阵的快速转置算法完成下面矩阵的转置。

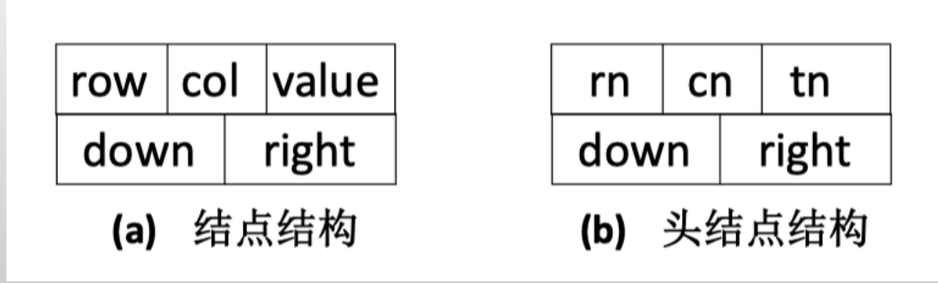
5.1.5 十字链表

十字链表的定义

对于稀疏矩阵，当非 0 元素的个数和位置在操作过程中变化较大时，采用链式存储结构表示比三元组的线性表更方便。

矩阵中非 0 元素的结点所含的域有：行、列、值、行指针 (指向同一行的下一个非 0 元)、列指针 (指向同一列的下一个非 0 元)。其次，十字交叉链表还有一个头结点。

表示：



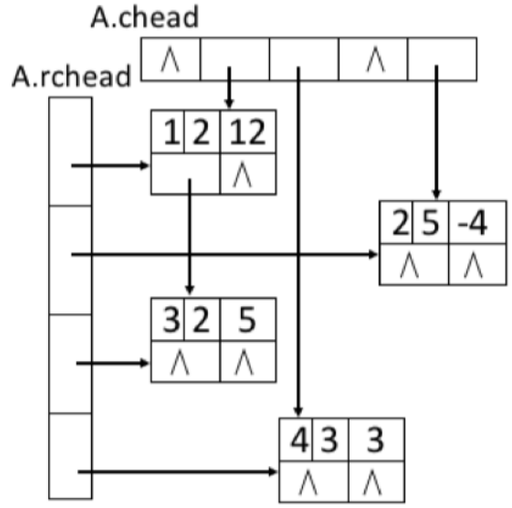
Properties: 十字链表

(1) 由定义知，稀疏矩阵中同一行的非 0 元素的由 right 指针域链接成一个行链表，由 down 指针域链接成一个列链表。则每个非 0 元素既是某个行链表中的一个结点，同时又是某个列链表中的一个结点，所有的非 0 元素构成一个十字交叉的链表，称为十字链表。

(2) 此外，还可用两个一维数组分别存储行链表的头指针和列链表的头指针。

$$A=\begin{pmatrix} 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{pmatrix}$$

(a) 稀疏矩阵



(b) 稀疏矩阵的十字交叉链表

5.2 广义表

广义表的定义

- 广义表是线性表的推广和扩充，在人工智能领域中应用十分广泛。
- 我们把线性表定义为 n($n \geq 0$) 个元素 $a_1, a_2, \ldots, a_n$ 的有穷序列，该序列中的所有元素具有相同的数据类型且只能是原子项 (Atom)。所谓原子项可以是一个数或一个结构，是指结构上不可再分的。若放松对元素的这种限制，容许它们具有其自身结构，就产生了广义表的概念。
- 广义表 $LS = (a_1, a_2, \ldots, a_n)$, 其中其中 a_i 或者是原子项，或者是一个广义表。LS 是广义表的名字，n 为它的长度。若 a_i 是广义表，则称为 LS 的子表。习惯上：原子用小写字母，子表用大写字母。
- 广义表的长度：元素的数目。
- 广义表的表头：非空广义表中第一个元素。
- 广义表的表尾：除表头元素之外，其余元素构成的表。
- 广义表的深度：广义表中括号的重数。

例题 5

写下表

解答

	长度	表头	表尾	深度
A=()	0			1
B=(e)	1	e	()	1
C=(a,(b,c,d))	2	a	((b,c,d))	2
D=(A,B,C)	3	()	(B,C)	3
E=(a,E)	2	a	(E)	无穷大

6 树

6.1 树

非线性结构的定义

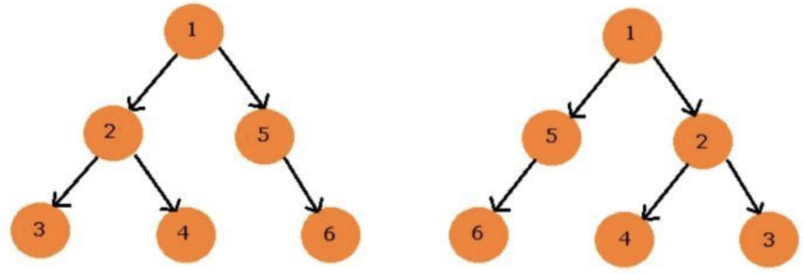
所谓非线性结构是指，在该结构中至少存在一个数据元素，有两个或两个以上的直接前驱（或直接后继）元素。相对应于线性结构，非线性结构的逻辑特征是一个结点元素可能对应多个直接前驱和多个后继。

树型结构和图型就是其中十分重要的非线性结构，可以用来描述客观世界中广泛存在的层次结构和网状结构的关系，如家族谱、城市交通等。

树的定义

- 树 (Tree) 是 n($n \geq 0$) 个结点的有限集合 T，若 n=0 时称为空树 (Empty)，否则：
  - (1) 有且只有一个特殊的称为树的根 (Root) 结点；
  - (2) 若 n>1 时，其余的结点被分为 m(m>0) 个互不相交的子集 $T_1, T_2, T_3 \ldots T_m$ ，其中每个子集本身又是一棵树，称其为根的子树 (Subtree)。
- 结点 (node)：一个数据元素及其若干指向其子树的分支。
- 结点的度 (degree)：结点所拥有的子树的个数称为该结点的度。
- 树的度：树中各结点度的最大值称为该树的度
- 叶子 (left) 结点：树中度为 0 的结点称为叶子结点 (或终端结点)。
- 非叶子结点：度不为 0 的结点称为非叶子结点 (或非终端结点或分支结点)。除根结点外，分支结点又称为内部结点。
- 一个结点的子树的根称为该结点的孩子结点 (child) 或子结点；相应地，该结点是其孩子结点的双亲结点 (parent) 或父结点。
- 同一双亲结点的所有子结点互称为兄弟结点。
- 规定树中根结点的层次为 1，其余结点的层次等于其双亲结点的层次加 1。
- 若某结点在第 l ($l \geq 1$) 层，则其子结点在第 $l+1$ 层。
- 双亲结点在同一层上的所有结点互称为堂兄弟结点 (Cousins)。
- 从根结点开始，到达某结点 p 所经过的所有结点成为结点 p 的层次路径 (有且只有一条)。
- 结点 p 的层次路径上的所有结点（p 除外）称为 p 的祖先 (ancester) 。
- 以某一结点为根的子树中的任意结点称为该结点的子孙结点 (descent)。
- 树的深度 (depth)：树中结点的最大层次值，又称为树的高度
- 有序树和无序树：对于一棵树，若其中每一个结点的子树（若有）具有一定的次序，则该树称为有序树，否则称为无序树。
- 森林 (forest)：是 m($m \geq 0$) 棵互不相交的树的集合。显然，若将一棵树的根结点删除，剩余的子树就构成了森林。

Tip 究竟什么是有序树呢？这个概念不怎么好理解，换句话说就是兄弟节点有顺序的树，称为有序树. 就是说每一层的兄弟结点们都是按照同样的规则进行排列。下面两棵树都是有序树。

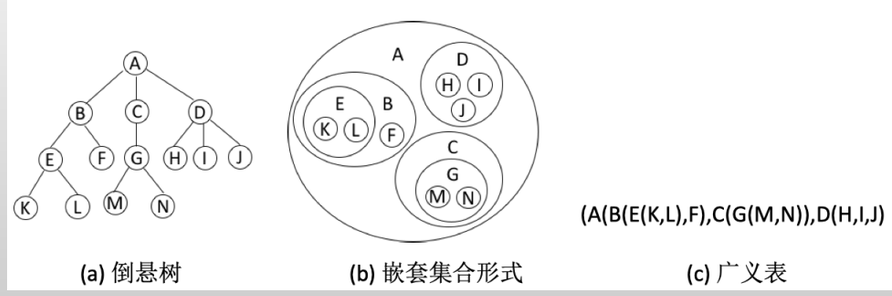


6.2 树的表示

树的几种表示

- 倒悬树。
- 嵌套集合。是一些集合的集体，对于任何两个集合，或者不相交，或者一个集合包含另一个集合。
- 广义表形式。

举例：



(a) 倒悬树　(b) 嵌套集合形式　(c) 广义表

6.3 二叉树

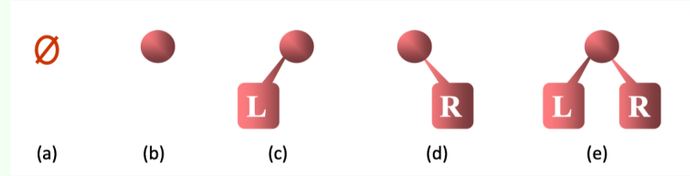
树的定义

- 二叉树是由 n($n \geq 0$) 个结点的有限集合构成，此集合或者为空集，或者由一个根结点及两棵互不相交的左右子树组成，并且左右子树都是二叉树。
- 这是一个递归定义。二叉树可以是空集合，根可以有空的左子树或空的右子树。
- 二叉树不是树的特殊情况，它们是两个概念。

Properties: 二叉树与树的区别

- 二叉树与无序树不同
  - 二叉树中，每个结点最多只能有两棵子树，并且有左右之分。二叉树并非是树的特殊情形，它们是两种不同的数据结构。
- 二叉树与度数为 2 的有序树不同
  - 在有序树中，虽然一个结点的孩子之间是有左右次序的，但是若该结点只有一个孩子，就无须区分其左右次序。
  - 而在二叉树中，即使是一个孩子也有左右之分。
- 二叉树结点的子树要区分左子树和右子树，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。这是二叉树与树的最主要的差别。

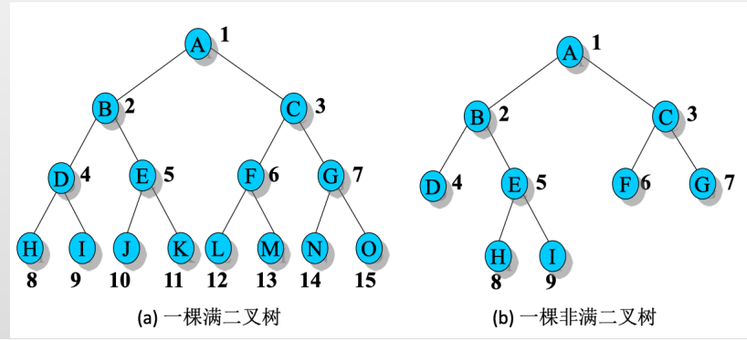
  二叉树的 5 种基本形态如图所示



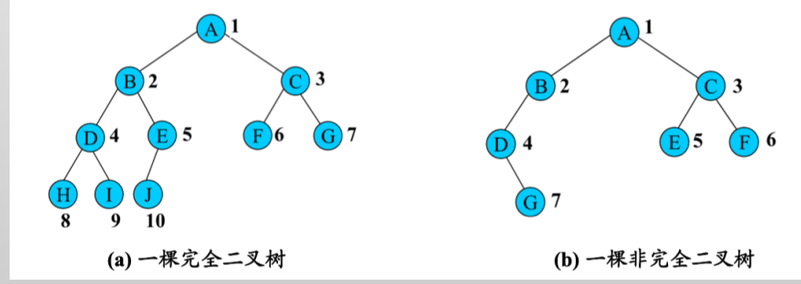
二叉树相关的概念

- 满二叉树：在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶结点都在同一层上，这样的一棵二叉树称作满二叉树。
- 完全二叉树：一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i $(1 \leq i \leq n)$ 的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。
- 一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。

举例：



(a) 一棵满二叉树　　(b) 一棵非满二叉树



(a) 一棵完全二叉树　　**(b) 一棵非完全二叉树**

例题 6

达式树

解答

Properties: 二叉树的性质

- 在二叉树的第 i 层上至多有 2^{i-1} 个结点 ($i \geq 1$)。

  证明： 当 i=1 时，只有一个根结点，$2^{i-1} = 2^0 = 1$ 成立。
  假设第 i-1 层上至多有 2^{i-2} 个结点, 由于二叉树每个结点的度最大为 2，故在第 i 层上最大结点数为第 i-1 层上最大结点数的二倍，即 $2 * 2^{i-2} = 2^{i-1}$

- 深度为 k 的二叉树至多有 2^{k-1} 个结点 ($k \geq 1$)。

  证明：
  $$\mathbf{M} = \sum_{\mathbf{i}=1}^{\mathbf{k}} \mathbf{x_i} \leq \sum_{\mathbf{i}=1}^{\mathbf{k}} 2^{\mathbf{i}-1} = \mathbf{2^k - 1}$$

- 对任何一棵二叉树，如果其叶结点数为 n_0 ，度为 2 的结点数为 n_2 ，则 $n_0 = n_2 + 1$。

  证明： 设 n 为二叉树的结点总数，n_1 为二叉树中度为 1 的结点数，则有：$n = n_0 + n_1 + n_2$, 在二叉树中，除根结点外，其余结点都有唯一的一个进入分支。设 B 为二叉树中的分支数，那么有：$B = n - 1$, 这些分支是由度为 1 和度为 2 的结点发出的，一个度为 1 的结点发出一个分支，一个度为 2 的结点发出两个分支，所以有：$B = n_1 + 2n_2$, 所以 $n_0 = n_2 + 1$。

- 具有 n 个结点的完全二叉树的深度 k 为 $\lfloor \log_2 n + 1 \rfloor$。

  证明： 根据完全二叉树的定义和性质 2 可知，当一棵完全二叉树的深度为 k、结点个数为 n 时，有 $2^{k-1} - 1 < n \leq 2^k - 1$, 即 $2^{k-1} \leq n < 2^k$, 两边同时取对数有 $k - 1 \leq \log_2 n < k$，由于 k 是整数，则 $k = \lfloor \log_2 n + 1 \rfloor$

- 如果对一棵有 n 个结点的完全二叉树的结点按层序编号 (从第 1 层到第 $\lfloor \log_2 n + 1 \rfloor$ 层，每层从左到右)，则对任一结点 i($1 \leq i \leq n$)，有
  - 如果 i = 1，则结点 i 无双亲，是二叉树的根；如果 i>1，则其双亲是结点 $\lfloor \frac{i}{2} \rfloor$
  - 如果 2i>n，则结点 i 为叶结点，无左孩子；否则，其左孩子是结点 2i。
  - 如果 2i+1>n，则结点 i 无右孩子；否则，其右孩子是结点 2i+1。

证明：

- 对于 i = 1，由完全二叉树的定义，其左孩子是结点 2，若 2>n，即不存在结点 2，此是，结点 i 无孩子。结点 i 的由孩子也只能是结点 3，若结点 3 不存在，即 3>n，此时结点 i 无右孩子。
- 对于 i>1，可分为两种情况：
  - 设第 j($1 \leq j \leq \lfloor \frac{i}{2} \rfloor$) 层的第一个结点的编号为 i，由二叉树的性质 2 和定义知 $i = 2^{j-1}$ 。结点 i 的左孩子必定为的 j+1 层的第一个结点，其编号为 $2^j = 2 * 2^{j-1} = 2^i$。如果 $2^i > n$，则无左孩子；其右孩子必定为第 j+1 层的第二个结点，编号为 2i+1。若 2i+1>n，则无右孩子。
  - 假设第 j($1 \leq j \leq \lfloor \frac{i}{2} \rfloor$) 层上的某个结点编号为 i($2e(j-1) \leq i \leq 2ej-1$)，且 2i+1<n，其左孩子为 2i，右孩子为 2i+1，则编号为 i+1 的结点时编号为 i 的结点的右兄弟或堂兄弟。若它有左孩子，则其编号必定为 2i+2 = 2*(i+1)：若它有右孩子，则其编号必定为 2i+3 = 2*(i+1)+1。
- 当 i = 1 时，就是根，因此无双亲，当 i>1 时，如果 i 为左孩子，即 2Œ(i/2)=i，则 i/2 是 i 的双亲；如果 i 为右孩子，i = 2p+1，i 的双亲应为 p，p = (i-1)/2=$\lfloor i/2 \rfloor$。

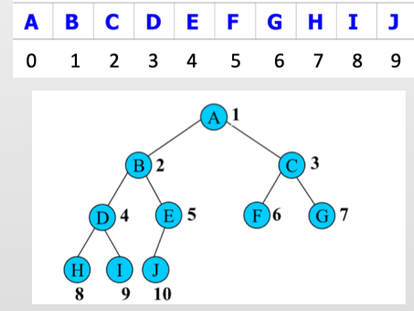
6.4 树的存储

顺序存储结构

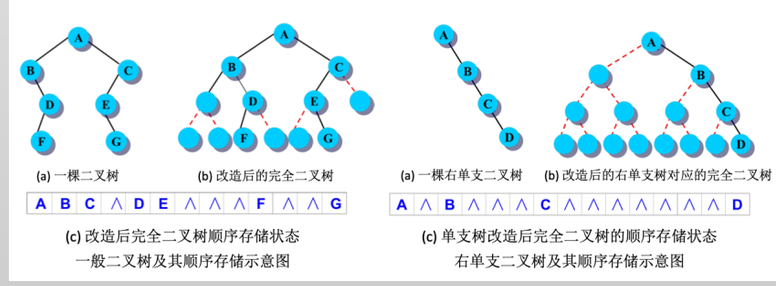
- 用一维数组存储存放二叉树中的结点。一般是按照二叉树结点从上至下、从左到右的顺序存储。
- 这样结点在存储位置上的前驱后继关系并不一定就是它们在逻辑上的邻接关系，然而只有通过一些方法确定某结点在逻辑上的前驱结点和后继结点，这种存储才有意义。
- 因此，依据二叉树的性质，完全二叉树和满二叉树采用顺序存储比较合适，树中结点的序号可以唯一地反映出结点之间的逻辑关系，这样既能够最大可能地节省存储空间，又可以利用数组元素的下标值确定结点在二叉树中的位置，以及结点之间的关系。
- 对于一般的二叉树，如果仍按从上至下和从左到右的顺序将树中的结点顺序存储在一维数组中，则数组元素下标之间的关系不能够反映二叉树中结点之间的逻辑关系，只有增添一些并不存在的空结点，使之成为一棵完全二叉树的形式，然后再用一维数组顺序存储。显然，这种存储对于需增加许多空结点才能将一棵二叉树改造成为一棵完全二叉树的存储时，会造成空间的大量浪费，不宜用顺序存储结构。

举例：

- 完全二叉树的顺序存储示意



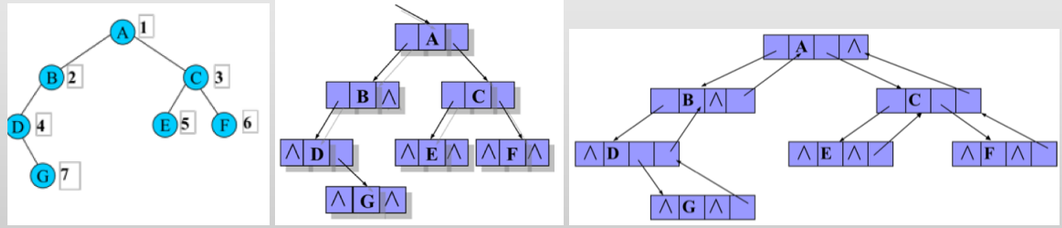
- 一般二叉树的顺序存储示意



链式存储结构

- 二叉树的链式存储结构是用链表来表示一棵二叉树，即用链来指示着元素的逻辑关系。通常有二叉链表和三叉链表两种形式。
- 二叉链表：链表中每个结点由三个域组成，除了数据域外，还有两个指针域，分别用来给出该结点左孩子和右孩子所在的链结点的存储地址。data 域存放结点的数据；lchild 与 rchild 分别存放指向左孩子和右孩子的指针，当左孩子或右孩子不存在时，相应指针域值为空 (用 或 NULL 表示)。
- 三叉链表: 每个结点由四个域组成，其中，data、lchild 以及 rchild 三个域的意义同二叉链表结构；parent 域为指向该结点双亲结点的指针。这种存储结构既便于查找孩子结点，又便于查找双亲结点；但是，相对于二叉链表存储结构而言，它增加了空间开销。

举例：



6.5 二叉树的操作

6.5.1 先序遍历

先序遍历定义

若二叉树为空，遍历结束。否则，

- 访问根结点；
- 先序遍历根结点的左子树；
- 先序遍历根结点的右子树。

递归写法

```
1 void PreOrder(BiTree bt) // 递归调用的结束条件
```

```
{
    if (bt==NULL) return;

    Visite(bt->data);// 访问结点的数据域

    PreOrder(bt->lchild);// 先序递归遍历bt的左子树

    PreOrder(bt->rchild); // 先序递归遍历bt的右子树
}
```

先序遍历的非递归实现

思路

用栈来帮助实现遍历路线。先序遍历是在深入时遇到结点就访问。

- 在沿左子树深入时，深入一个结点，入栈一个结点。
- 在入栈之前访问之，当沿左分支深入不下去时，则返回，即从堆栈中弹出前面压入的结点；

```
template <class T> void BinaryTree<T>::PreOrder(void (*visit)(
   BinTreeNode<T> *t)) { stack<BinTreeNode<T>*> S;

BinTreeNode<T> *p = root;
S.Push (NULL);
 while (p != NULL) {
    visit(p); // 访问结点
    if (p->rightChild != NULL)
        S.Push (p->rightChild); // 预留右指针在栈中
    if (p->leftChild != NULL)
        p = p->leftChild; //进左子树
    else
        S.Pop(p); //左子树为空
 }
```

```
14 }
```

6.5.2 中序遍历

中序遍历定义

若二叉树为空，遍历结束。否则，

- 中序遍历根结点的左子树；
- 访问根结点；
- 中序遍历根结点的右子树。

递归写法

```
1 void InOrder(BiTree bt) // 递归调用的结束条件
2 {
3     if (bt==NULL) return;
4     InOrder(bt->lchild);// 中序递归遍历bt的左子树
5     Visite(bt->data);// 访问结点的数据域
6     InOrder(bt->rchild); // 中序递归遍历bt的右子树
7 }
```

中序遍历的非递归实现

思路

用栈来帮助实现遍历路线。中序遍历是在从左子树返回时遇到结点访问

- 在沿左子树深入时，深入一个结点，入栈一个结点。
- 访问该结点，然后从该结点的右子树继续深入；

```
1 template <class T> void BinaryTree<T>::InOrder(void (*visit) (
    BinTreeNode<T> *t)) { stack<BinTreeNode<T>*> S;
2 BinTreeNode<T> *p = root;
3 do {
```

```
        while (p != NULL) //遍历指针向左下移动
        {
            S.Push (p); //该子树沿途结点进栈
            p = p->leftChild;
        }
        if (!S.IsEmpty()) //栈不空时退栈
        {
            S.Pop (p);
            visit (p); //退栈, 访问
            p = p->rightChild; //遍历指针进到右子女 }
    } while (p != NULL || !S.IsEmpty ());

}
```

6.5.3 后序遍历

中序遍历定义

若二叉树为空，遍历结束。否则，

- 后序遍历根结点的左子树;
- 后序遍历根结点的右子树。
- 访问根结点;

递归写法

```
void PostOrder(BiTree bt) // 递归调用的结束条件
{
    if (bt==NULL) return;
    PostOrder(bt->lchild);// 后序递归遍历bt的左子树
    PostOrder(bt->rchild); // 后序递归遍历bt的右子树
}
```

后序遍历的非递归实现

思路

用栈来帮助实现遍历路线。后序遍历是在从右子树返回时遇到结点访问

- 在沿左子树深入时，深入一个结点，入栈一个结点。
- 将此结点再次入栈，然后从该结点的右子树继续深入，与前面类同，仍能深入一个结点入栈一个结点，深入不下去再返回，直到第二次从栈里弹出该结点，才访问之。

```
template <class T> struct stkNode { BinTreeNode<T> *ptr; // 树结点指针
enum tag {L, R}; // 退栈标记
stkNode (BinTreeNode<T> *N = NULL) :
    ptr(N), tag(L) { } // 构造函数
};
//tag = L, 表示从左子树退回还要遍历右子树;
//tag = R, 表示从右子树退回要访问根结点。


void BinaryTree<T>::PostOrder (void (*visit) (BinTreeNode<T> *t) {
    Stack< stkNode<T> > S; stkNode <T> w;
    BinTreeNode<T> * p = root; // p是遍历指针
    do {
        while (p != NULL)
        {
            w.ptr = p;
            w.tag = L;
            S.Push (w);
            p = p->leftChild;
        }
        int continue1 = 1;// 继续循环标记, 用于R
        while (continue1 && !S.IsEmpty ()) {
            S.Pop (w);
            p = w.ptr;
```

```
25              switch (w.tag) // 判断栈顶的tag标记
26                  case L:
27                      w.tag = R;
28                      S.Push (w);
29                      continue1 = 0;
30                      p = p->rightChild;
31                      break;
32                  case R:
33                      visit (p);
34                      break;
35                  }
36
37          } while (!S.IsEmpty ()); // 继续遍历其他结点
38          cout << endl;
39  };
```

Properties: 一些性质

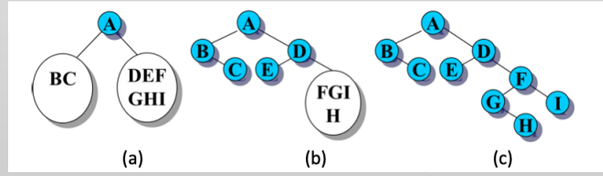
- (1) 任意一棵二叉树结点的先序序列和中序序列都是唯一的。
- (2) 二叉树的先序序列和中序序列可唯一地确定一棵二叉树。
- (3) 二叉树的后序序列和中序序列可唯一地确定一棵二叉树。
- (4) 二叉树的先序序列和后序序列不可唯一确定一棵二叉树。

例题 7

知一棵二叉树的先序与中序序列分别为 ABCDEFGHI 和 BCAEDGHFI，试恢复该二叉树。

解答

- 首先，由先序序列可知，结点 A 是二叉树的根结点。
- 其次，根据中序序列，在 A 之前的所有结点都是根结点左子树的结点，在 A 之后的所有结点都是根结点右子树的结点，由此得到图 (a) 所示的状态。
- 然后，再对左子树进行分解，由先序序列得知 B 是左子树的根结点，又从中序序列知道，B 的左子树为空，B 的右子树只有一个结点 C。
- 接着对 A 的右子树进行分解，得知 A 的右子树的根结点为 D；而结点 D 把其余结点分成两部分，即左子树为 E，右子树为 F、G、H、I，如图 (b) 所示。接下去的工作就是按上述原则对 D 的右子树继续分解下去，最后得到如图 (c) 的整棵二叉树。



思路总结:

- 先根据先序序列的第一个元素建立根结点；
- 然后在中序序列中找到该元素，确定根结点的左、右子树的中序序列；
- 再在先序序列中确定左、右子树的先序序列；
- 最后由左子树的先序序列与中序序列建立左子树，由右子树的先序序列与中序序列建立右子树。

特例: 相同的先序遍历和后序遍历，但是可以构造出不同的树的例子

6.5.4 层次遍历

7 参考文献

References

1. https://zhuanlan.zhihu.com/p/47354495 树性结构：有序树和二叉树