# Data Structures

**Binary Search Trees**

**Teacher : Wang Wei**

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. 殷人昆,　　　　数据结构
3. 金远平,　　　　数据结构
4. http://inside.mines.edu/~dmehta/



## Search structure

- The most common objective of computer is to store and retrieve data
- An efficient ways to organize collections of data records
  - Be stored and retrieved quickly
  - Such as **dictionary**
- Dictionary is a collection of record pairs <element, key>
  - Each pair has a key and an associated element
  - Assumption no two pair have the same key
- Dictionary provides operations for storing records, searching records and removing records from the collection

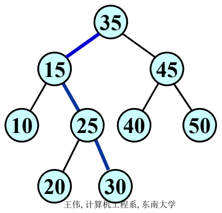


## Search Problem

- Suppose
  - Have a dictionary *D* of *n* record pairs *<element,key>*
    <$e_1$,$k_1$>,<$e_2$,$k_2$>,…,<$e_n$,$k_n$>
- Search for records might wish to search for the Key
  - Example 1 : given a particular key value *K* , find an element with key value $k_j$ = *K*
  - Example 2 : find the fifth smallest element…
  - …
- Result of a search
  - Successful : is found the record pair with $k_j$ in *D*
  - Unsuccessful : is not found or no such record pair exists in *D*

# Binary Search Tree （BST）

- Definition
  - A binary tree
  - Each node has a (key, value) pair
  - For every node x
    - all keys in the *left* subtree of x are smaller than that in x
    - all keys in the *right* subtree of x are greater than that in x






# Class Definition

```
#include <iostream.h>
#include <stdlib.h>
template <class E,class K>
struct BSTNode
{                                       //二叉树结点类
   E data;                                  //数据域
   BSTNode<E,K> *left, *right;          //左子女和右子女

  //  ….

};
```



```
template <class E,class K>
class BST {
public:
    BST() { root = NULL; }            //构造函数
    BST(K value);          //构造函数
    ~BST() {};                //析构函数
    bool Search (const K x) const
                  { return Search(x,root) != NULL; } //搜索
    BST<E>& operator = (const BST<E,K>& R); //重载：赋值
    void makeEmpty() { makeEmpty (root); root = NULL;} //置空
    void PrintTree() const { PrintTree (root); }          //输出
    E Min() { return Min(root)->data; }         //求最小
     E Max() { return Max(root)->data; }        //求最大
     bool Insert (const E & e1)
                          { return Insert(e1, root); } //插入新元素
     bool Remove (const K x)
                          { return Remove(x, root);} //删除含x的结点
```

```
private:
    BSTNode<E,K> *root;  //根指针
    K RefValue;                          //输入停止标志
    BSTNode<E,K> *
        Search (const K x, BSTNode<E,K> *ptr);      //递归：搜索

    void makeEmpty (BSTNode<E,K> *& ptr);     //递归：置空
    void PrintTree (BSTNode<E,K> *ptr) const;   //递归：打印
    BSTNode<E,K> *
        Copy (const BSTNode<E,K> *ptr);            //递归：复制

    BSTNode<E,K>* Min (BSTNode<E,K>* ptr); //递归：求最小
    BSTNode<E,K>* Max (BSTNode<E,K>* ptr); //递归：求最大

    bool Insert (const E& e1, BSTNode<E,K>*& ptr); //递归：插入
    bool Remove (const K x, BSTNode<E,k>*& ptr); //递归：删除
};
```



```
//Recursive :
//在以ptr为根的二叉搜索树中搜索含x的结点
//若找到，则函数返回该结点的地址，否则函数返回NULL值
template<class E,class K>
BSTNode<E,K>* BST<E,K>::
        Search (const K x, BSTNode<E,K> *ptr)
{
    if (ptr == NULL) return NULL;
    else if (x < ptr->data) return Search(x, ptr->left);
    else if (x > ptr->data) return Search(x, ptr->right);
    else return ptr;                              //搜索成功
};
```



```
//Iterative :
// 作为对比，在当前以ptr为根的二叉搜索树中搜索含x的结点
//若找到，则函数返回该结点的地址，否则函数返回NULL值
  if (ptr == NULL) return NULL;
  BSTNode<E>* temp = ptr;
  while (temp != NULL) {
      if (x == temp->data) return temp;
      if (x < temp->data) temp = temp->left;
     else temp = temp->right;
  }
  return NULL;
```

## Insertion Operation

```
template <class E,class K>
bool BST<E,K>::Insert (const E& e1, BSTNode<E,K> *& ptr )
{  // 私有函数：
  // 在以ptr为根的二叉搜索树中插入值为e1的结点
  // 若在树中已有含e1的结点, 则不插入
      if (ptr == NULL) {          //新结点作为叶结点插入
        ptr = new BstNode<E>(e1);       //创建新结点
        if (ptr == NULL)
          { cerr << "Out of space" << endl;  exit(1); }
         return true;
      }
     else if (e1 < ptr->data) Insert (e1, ptr->left); //左子树插入
     else if (e1 > ptr->data) Insert (e1, ptr->right); //右子树插入
     else return false;                        //x已在树中,不再插入
  };
```



```
template <class E, class K>
BST<E,K>::BST (K value)
{
 //输入一个元素序列, 建立一棵二叉搜索树
   E x;
   root = NULL;  RefValue = value;              //置空树
   cin >> x;                                    //输入数据
   while ( x.key != RefValue) {
                                       //RefValue是一个输入结束标志
      Insert (x, root);  cin >> x;         //插入,再输入数据
   }
}
```



## Deletion Operation

- When remove a node from a BST
  - ✓ Deletion of a leaf
    - ✓ Its parent is set to 0, and the node disposed
  - ✓ Deletion of a nonleaf that has only one child
    - ✓ The node is disposed, and the single-child takes the place of the node
      - ✓ left child replace the disposed node
      - ✓ right child replace the disposed node
  - ✓ Deletion of a nonleaf that has two children
    - ✓ The node is replaced by either the largest node in its left subtree or the smallest one in its right subtree
    - ✓ Then the replacing node be proceed to remove from the subtree from which it was taken

## Deletion Operation

```
//在以 ptr 为根的二叉搜索树中删除含 x 的结点
template <class E,class K >
bool BST<E,K>::Remove (const K x, BstNode<E,K> *& ptr)
{
   BstNode<E> *temp;
   if (ptr != NULL)
   {
       if (x < ptr->data) Remove (x, ptr->left);
                                        //在左子树中执行删除
        else if (x > ptr->data) Remove (x, ptr->right);
                                              //在右子树中执行删除
```



```
else if (ptr->left != NULL && ptr->right != NULL)
{                    //ptr指示关键码为x的结点，它有两个子女
    temp = ptr->right;
                     //到右子树搜寻中序下第一个结点
    while (temp->left != NULL)
         temp = temp->left;
    ptr->data = temp->data;
                    //用该结点数据代替根结点数据
      Remove (ptr->data, ptr->right);
}
```



```
  else {        //ptr指示关键码为x的结点有一个子女
          temp = ptr;
           if (ptr->left == NULL)  ptr = ptr->right;
          else ptr = ptr->left;
          delete temp;  // disposed
          return true;
     }
   }
   return false;
}
```

# Data Structures

Thread Binary Trees

**Teacher : Wang Wei**

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. 殷人昆, 数据结构
3. 金远平, 数据结构
4. http://inside.mines.edu/~dmehta/



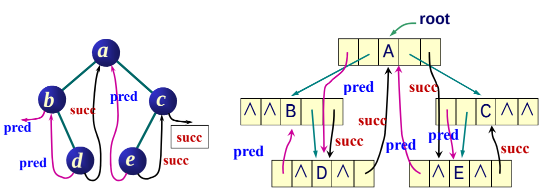
## Threaded Binary Tree

- Using the threads and without an additional stack
  - Perform an *inorder* traversal
  - Find the *inorder* successor of any arbitrary node
  - Perform an *preorder* traversal
  - Perform an *postorder* traversal



## Thread Binary Tree and Nodes

| pred | leftChild | data | rightChild | succ |
|---|---|---|---|---|




- predecessor thread pointer pre
- successor thread pointer succ

## Nodes sturcture

- Distinguish between threads and normal pointer
  - Adding two Boolean fields : Ltag and Rtag
  - Let *t* be a pointer to a tree node
  - If *t->Ltag==true*, then *t->leftChild* contains a thread; otherwise contains a pointer to the left child
  - If *t->Rtag==true*, then *t->rightChild* contains a thread; otherwise contains a pointer to the right child

| leftChild | Ltag | data | Rtag | rightChild |
|---|---|---|---|---|



## Inorder Thread Binary Tree

```
template <class T>
void ThreadTree<T>::createInorderThread()
{
    ThreadNode<T> *pre = NULL;            //前驱结点指针
     if (root != NULL) {                  //非空二叉树, 线索化
        createInorderThread (root, pre);  //中序遍历线索化二叉树
         pre->rightChild = NULL;
        pre->Rtag = 1;                    //后处理中序最后一个结点
     }
}
```



```
template <class T>
void ThreadTree<T>::createInThread(ThreadNode<T> *current,
                                   ThreadNode<T> *& pre)
{
    //通过中序遍历, 对二叉树进行线索化
    if (current == NULL) return;
    createInThread (current->leftChild, pre);  //递归, 左子树线索化
     if (current->leftChild == NULL)
    {
        //建立当前结点的前驱线索
        current->leftChild = pre;
        current->Ltag = 1;
     }
```

```
//建立前驱结点的后继线索
if (pre != NULL && pre->rightChild == NULL)
{   pre->rightChild = current;
    pre->Rtag = 1;
 }
 pre = current;     //前驱跟上,当前指针向前遍历
 createInThread (current->rightChild,  pre); //递归, 右子树线索化
}
```

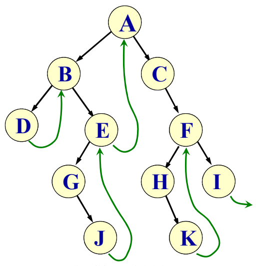


## Finding the *inorder* successor of current Node

**if (current->Rtag ==1)   successor is current->rightChild**

**else //current->Rtag == 0**

**the *inorder* successor is the first node of the right subtree of current node**





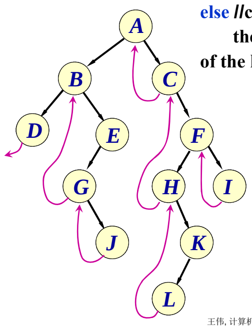

## Finding the *inorder* predecessor of current Node

**if (current->Ltag ==1)**

**successor is current->leftChild**

**else //current->Ltag == 0**

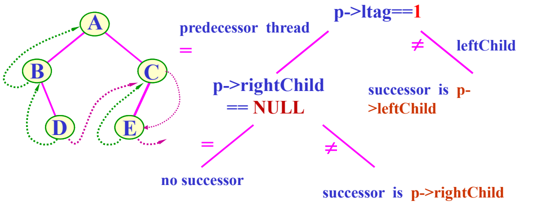
**the *inorder* predecessor is the last node of the left subtree of current node**



## Preorder Thread Binary Tree

- Finding the *preorder* successor of the node *p*

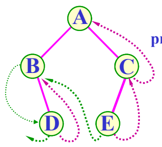
Preorder : A B D C E

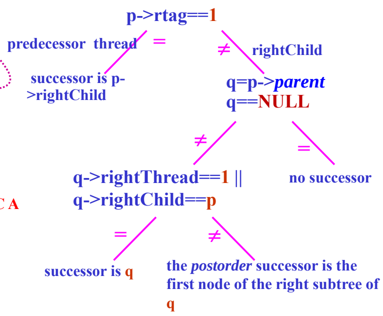





## Postorder Thread Binary Tree

- **Finding the *postorder* successor of the node *p***




Postorder : D B E C A



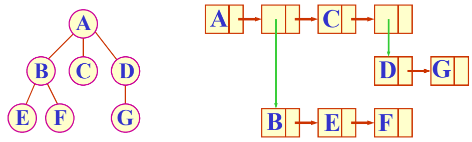
# Data Structures

Trees and Forests

**Teacher : Wang Wei**

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. 殷人昆, 数据结构
3. 金远平, 数据结构
4. http://inside.mines.edu/~dmehta/

## 1. Generalized List Representation



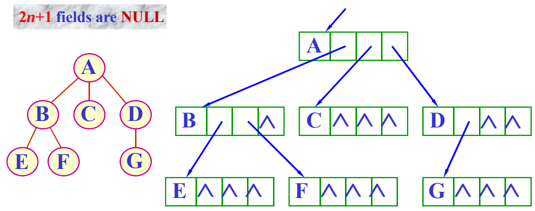

A(B(E, F), C, D(G))

utype field not shown



## 2. Parent-Child Representation

$2n+1$ fields are NULL




Each node having a fixed size

| | | | | | |
|---|---|---|---|---|---|
| data | $child_1$ | $child_2$ | $child_3$ | …… | $child_d$ |



## 3. Child-Sibling Representation

- Two specialized fixed-node-size representation
- Node structure

| | | |
|---|---|---|
| data | firstChild | nextSibling |

↓

Left Child-Right Sibling

| | |
|---|---|
| data | |
| firstChild | nextSibling |

## 4. Degree-Two Representation

- Using binary tree
  - Rotate the right-sibling pointers in a left child-sibling tree clodkwise by 45 degrees
- Node structure

| firstChild | data | nextSibling |
|---|---|---|



## Abstract Data Type of Tree

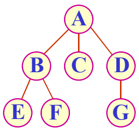
```
template <class T>
class Tree {
/*
   树是由n (≥0) 个结点组成的有限集合
   position 是树中结点的地址
   在顺序存储方式下是下标型; 在链表存储方式下是指针型
   T 是树结点中存放数据的类型, 要求所有结点的数据类型都是一致的
*/
public:
   Tree ();
   ~Tree ();
   /*   member functions   */
};
```



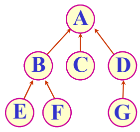
```
BuildRoot (const T& value);                //建立树的根结点
 position FirstChild(position p);
                          //返回 p 第一个子女地址, 无子女返回 0
 position NextSibling(position p);
                          //返回 p 下一兄弟地址, 若无下一兄弟返回 0
 position Parent(position p);
                          //返回 p 双亲结点地址, 若 p 为根返回 0
T getData(position p);                     //返回结点 p 中存放的值
bool InsertChild(position p, T& value);
//在结点 p 下插入值为 value 的新子女, 若插入失败, 函数返回false, 否则返回
true
bool DeleteChild (position p, int i);
//删除结点 p 的第 i 个子女及其全部子孙结点;若失败, 返回false, 否则返回true
 void DeleteSubTree (position t);
                                           //删除以 t 为根结点的子树
bool IsEmpty ();
                          //判树空否, 若空则返回true, 否则返回false
void Traversal (void (*visit)(position p));
                                           //遍历以 p 为根的子树
```

## 5. Parent Representation

- **One possible representation for sets**
  - Each set is represented as a tree
- **Linked the nodes from the children to the parent**
  - Array representation with parent field

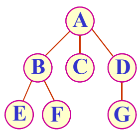


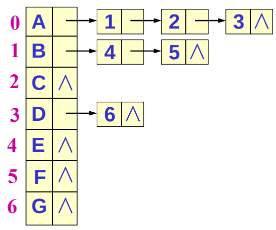

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| data | A | B | C | D | E | F | G |
| parent | −1 | 0 | 0 | 0 | 1 | 1 | 3 |






## 6. Children List Representation

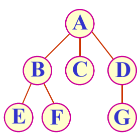


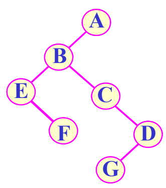






## Tree Traversal

- *tree preorder*
- *tree inorder*
- *tree level-order*

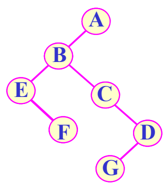



Left Child-Right Sibling

## *tree preorder*

- *Preorder* traversal of the tree is equivalent to visiting the nodes of the binary tree in *preorder*

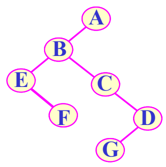



ABEFCDG



## *tree inorder*

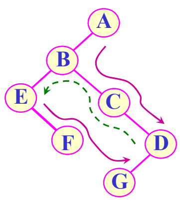
- ***Ineorder*** **traversal of the tree is equivalent to visiting the nodes of the binary tree in *inorder***




EFBCGDA



## *tree level-order*

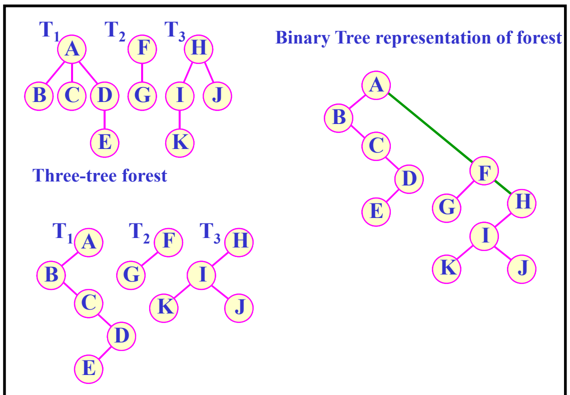



ABCDEFG

## Transforming a Forest into a Binary tree

- Using Child-Sibling Representation
  - Transforming a arbitrary Tree into a Binary Tree
  - Transforming a Forest into a Binary Tree






Binary Tree representation of trees



## Forest Traversal

- *forest preorder*
  - is equivalent **preorder of binary tree**
- *forest inorder*
  - is equivalent **inorder of binary tree**
- *forest level-order*
  - do not necessarily yield the same result **for level-order of binary tree**

## definition of a forest

- If $F$ is a forest of trees, then the binary tree corresponding to this forset, denoted by

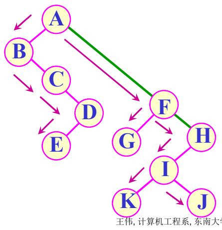
  $F = \{\{T_1 = \{ r_1, T_{11}, \ldots, T_{1k} \}, T_2, \ldots, T_m\}$

(1) is empty if n=0
(2) has root equal to root($T_1$) $r_1$
(3) has left subtree equal to $\{T_{11}, \ldots, T_{1k}\}$ , where $T_{11}, \ldots, T_{1k}$ are the subtrees of root($T_1$)
(4) has right subtree $\{T_2, \ldots, T_m\}$



## *forest preorder*

- if F = Ø , then return
- else // in forest preorder
  - ✓ Visit the root $r_1$ of the first tree of $T_1$
  - ✓ Traverse the subtree of the first tree $\{T_{11}, \ldots, T_{1k}\}$
  - ✓ Traverse the remaining trees of F $\{T_2, \ldots, T_m\}$

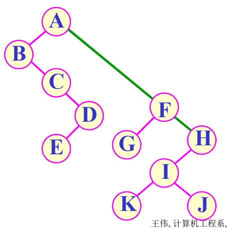



ABCDE FG HIKJ



## *forest inorder*

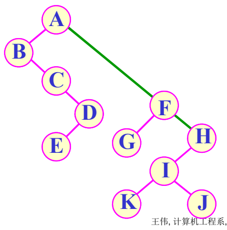
- **if F = Ø , then return**
- **else // in forest inorder**
  - ✓ **Traverse the subtree of the first tree $\{T_{11}, \ldots, T_{1k}\}$**
  - ✓ **Visit the root $r_1$ of the first tree of $T_1$**
  - ✓ **Traverse the remaining trees of F $\{T_2, \ldots, T_m\}$**




BCEDAGFKIJH

## *forest level-order*

- if F = Ø , then return
- else // in forest inorder
  - ✓ Nodes are visited by level, beginning with the roots of each tree in the forest
  - ✓ Within each level, nodes are visited from left to right




AFH BCDGIJ EK



---

# Data Structures

Union-Find Set

Teacher : Wang Wei


1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. 殷人昆, 数据结构
3. 金远平, 数据结构
4. http://inside.mines.edu/~dmehta/




---

## Disjoint Sets

- Given a set {1, 2, …, n} of n elements
- Initially each element is in a different set
  - {1}, {2}, …, {n}
- Assume
  - The elements of the sets are the numbers *0,1,2,…,n-1*
  - The sets being represented are pairwise disjoint
- Example
  - $S_1$ = {0,6,7,8}
  - $S_2$ = {1,4,9}
  - $S_3$ = {2,3,5}

## Initial a Union-Find Set (UFS)

- Each node is represented as a tree
- Using an array parent [] to represent the tree nodes
- parent[i] is the element that is the parent of element i

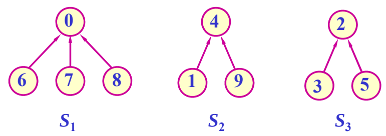
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| parent | −1 | −1 | −1 | −1 | −1 | −1 | −1 | −1 | −1 | −1 |

- The root nodes *parent [i] = -1*



## Parent Representation for Disjoint Sets

- One possible representation for sets
  - Each set is represented as a tree
- Linked the nodes from the children to the parent
  - Array representation with parent field

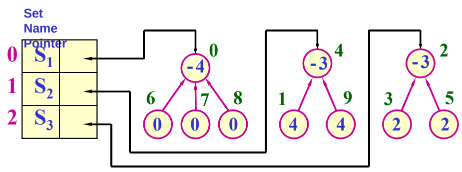



**tree representation of disjoint sets**

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| parent | −4 | 4 | −3 | 2 | −3 | 2 | 0 | 0 | 0 | 4 |



Let $S_1$= {0, 6, 7, 8}, $S_2$= {1, 4, 9}, $S_3$= {2, 3, 5}




- Each root has a pointer to the set name
- Parent links to the root of its tree and use the pointer to the set name
- In *Union* and *Find* algorithms
  - just identify sets by the roots of the trees
  - ignore the actual set names

## Constructor Function

```
// 构造函数:
// sz 是集合元素个数，双亲数组的范围为parent[0]-parent[size-1]

UFSets::UFSets(int sz)
{   size = sz;                          //集合元素个数
    parent = new int[size];             //创建双亲数组
    for(int i = 0; i < size; i++)
        parent[i] = -1;                 //每个自成单元素集合
};
```



## Operations of UFS

- Operator
  - Union(root1, root2) //合并操作
    - Combines two sets into one
      - each of the n elements is in exactly one set at any time
  - Find(i) //查找操作
    - Identifies the set that contains a particular element i
  - UFSets(s) //构造函数



## Strategy for *Find*

- Find(i)
  - start at the node that represents element i which given by parent[i]
  - follow parent fields until a root node whose parent field is null is reached
  - return element in this root node
  - Follow the tree, each node must have a parent pointer

```
int UFSets::Find(int i)
{   // Recursive Find, 搜索并返回包含元素x的树的根
    if (parent[i] < 0) return i;        //根的parent[]值小于0
    else return ( Find(parent[i]) );
};
//
int UFSets::Find(int i)  // Nonrecursive Find
{    while (parent[i] >= 0)
       i = parent[i];        // move up the tree
   return i;
}
```

## Strategy for *Union*

- Union(i,j)
  - i and j are the roots of two different trees, i != j
  - to unite the trees, make one tree a subtree of the other
    - parent[j] = i

```
void UFSets::Union(int Root1, int Root2)
{  // Recursive Union, 求两个不相交集合Root1与Root2的并
    parent[Root1] += parent[Root2];
    parent[Root2] = Root1;        //将Root2连接到Root1下面
};
```

王伟, 计算机工程系, 东南大学 55

## Time Complexity

- The time taken a union operator is O(1)
- The $n-1$ unions can be processed in time $O(n)$
- The time taken a find operator of the element $i$ is $O(i)$
- The total time need to process the $n$ finds is

$$O(\sum_{i=1}^{n} i) = O(n^2)$$

- *Find* and *Union* functions are very easy
- Their performance characteristics are not every good
  - Such as the degenerate tree (退化树)

王伟, 计算机工程系, 东南大学 56

## Abstract Data Type of UFS

```
//集合中的各个子集合互不相交
const int DefaultSize = 10;
class UFSets
{
 public:
    UFSets (int sz = DefaultSize);             //构造函数
    ～UFSets() { delete [] parent; }           //析构函数
    UFSets& operator = (UFSets& R);            //集合赋值
    void Union (int Root1, int Root2);         //子集合并
    int Find (int x);                          //查找x的根
 private:
    int *parent;              //集合元素数组(双亲表示)
    int size;                 //集合元素的数目
};
```

王伟, 计算机工程系, 东南大学 57