算法结构

|  |  |  |
| --- | --- | --- |
| 顺序结构 | | |
| 选择结构 | | |
| ┗ |  | |
|  | ┣ | “单分支”选择结构 |
|  | ┣ | “双分支”选择结构 |
|  | ┗ | “多分支”选择结构 |
| 循环结构 | | |
| ┣ |  | |
| ┃ | ┣ | 当型循环结构（入口条件循环） |
| ┃ | ┗ | 直到型循环结构（退出条件循环） |
| ┗ |  | |
|  | ┣ | 不确定循环 |
|  | ┗ | 计数循环 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

查找

|  |
| --- |
| 顺序查找 |
| 二分查找 |
| 分块查找 |
| 哈希查找 |

排序

|  |
| --- |
| void print\_indent(int indent=0)  {  int i;  for(i=0;i!=indent;++i)  {  printf(" ");  }  }  void print\_array(int\*p,int n,int highlight1=-1,int highlight2=-1)  {  int i;  for(i=0;i!=n;++i)  {  if(i==highlight1||i==highlight2)  {  printf("[%d]",p[i]);  }  else  {  printf(" %d ",p[i]);  }  }  }  void print\_heap(int\*p,int n,int highlight1=-1,int highlight2=-1,int indent=0)  {  int i1,i2;  print\_indent(indent);printf("/\n");  for(i1=1;i1<=n;i1\*=2)  {  print\_indent(indent);printf("|");print\_indent(i1\*2-2);  for(i2=0;i2!=i1&&i1-1+i2<n;++i2)  {  if(i1-1+i2==highlight1||i1-1+i2==highlight2)  {  printf("[%d]",p[i1-1+i2]);  }  else  {  printf(" %d ",p[i1-1+i2]);  }  }  printf("\n");  }  print\_indent(indent);printf("\\");  }  void sort(){}  int main()  {  int i,max=20;  const int n=10;  int a[n];  srand(static\_cast<unsigned int>(time(NULL)));  for(i=0;i!=n;++i)  {  a[i]=rand()%max;  }  print\_array(a,n);printf("\n");  sort();  print\_array(a,n);printf("\n");  } |

|  |  |  |  |
| --- | --- | --- | --- |
| 插入排序   |  | | --- | | 直接插入排序  / | | 折半插入排序  二分查找  int insert\_position(int\*p,int n,int key)  {  int min=0,mid,max=n-1;  while(min<=max)  {  mid=(min+max)/2;  if(p[mid]==key)  {  return mid;  }  if(p[mid]>key)  {  max=mid-1;  }  else  {  min=mid+1;  }  }  mid=(min+max)/2;  if(p[mid]<key)  {  return mid+1;  }  else  {  return mid;  }  }  void sort(int\*p,int n)  {  int i,j,pos,temp\_swap;  for(i=1;i!=n;++i)  {  pos=insert\_position(p,i,p[i]);  printf("(i=%d)",i);print\_array(p,n,i,pos);printf("\n");  temp\_swap=p[i];  for(j=i;j!=pos;--j)  {  p[j]=p[j-1];  }  p[pos]=temp\_swap;  printf ("(i=%d)",i);print\_array(p,i+1,pos);printf("\n");  }  } | | 希尔排序 | |
| 交换排序   |  | | --- | | 冒泡排序  void sort(int\*p,int n)  {  int i,flag=n-1,next\_flag;//i:[0,flag)  int temp;  while(flag!=0)  {  next\_flag=0;  printf("flag=%d\n",flag);  for(i=0;i!=flag;++i)  {  if(p[i]>p[i+1])  {  printf("(i=%d)",i);print\_array(p,flag+1,i,i+1);printf ("\n");  temp=p[i];p[i]=p[i+1];p[i+1]=temp;  printf("(i=%d)",i);print\_array(p,flag+1,i,i+1);printf ("\n");  next\_flag=i;  }  }  flag=next\_flag;  printf("flag=%d\n",flag);  printf("\n");  }  } | | 快速排序  分治法  void split(int\*p,int n,int\*i1,int\*i2,int indent=0)重复元素少  {  print\_indent(indent);printf("/\n");  print\_indent(indent);printf("|");print\_array(p,n,n-1);printf("\n");  \*i2=0; \*i1=n-2;  print\_indent(indent);printf("|");printf("(i2=%d i1=%d)\n",\*i2,\*i1);  int temp\_swap;  while(1)  {  while(\*i2!=n-1&&p[\*i2]<=p[n-1])  {  ++\*i2;  }  while(\*i1!=-1&&p[\*i1]>=p[n-1])  {  --\*i1;  }  if(\*i2>=\*i1)break;  print\_indent(indent);printf("|/");print\_array(p,n,\*i2,\*i1);printf("(i2=%d i1=%d)\n",\*i2,\*i1);  temp\_swap=p[\*i2]; p[\*i2]=p[\*i1]; p[\*i1]=temp\_swap; ++\*i2; --\*i1;  print\_indent(indent);printf("|\\");print\_array(p,n,\*i2,\*i1);printf("(i2=%d i1=%d)\n",\*i2,\*i1);  }  print\_indent(indent);printf("|/");print\_array(p,n,\*i2,\*i1);printf("(i2=%d i1=%d)\n",\*i2,\*i1);  temp\_swap=p[\*i2]; p[\*i2]=p[n-1]; p[n-1]=temp\_swap; ++\*i2;  print\_indent(indent);printf("|\\");print\_array(p,n,\*i2,\*i1);printf("(i2=%d)\n",\*i2);  print\_indent(indent);printf("\\\n");  }  void split(int\*p,int n,int\*i1,int\*i2,int indent=0)重复元素多  {  print\_indent(indent);printf("/\n");  print\_indent(indent);printf("|");print\_array(p,n,n-1);printf("\n");  \*i2=0; \*i1=n-2;  print\_indent(indent);printf("|");printf("(i2=%d i1=%d)\n",\*i2,\*i1);  int temp\_swap,i\_temp;  while(1)  {  while(\*i2!=n-1&&p[\*i2]<=p[n-1])  {  ++\*i2;  }  while(\*i1!=-1&&p[\*i1]>p[n-1])  {  --\*i1;  }  if(\*i2>=\*i1)break;  print\_indent(indent);printf("|/");print\_array(p,n,\*i2,\*i1);printf("(i2=%d i1=%d)\n",\*i2,\*i1);  temp\_swap=p[\*i2]; p[\*i2]=p[\*i1]; p[\*i1]=temp\_swap; ++\*i2; --\*i1;  print\_indent(indent);printf("|\\");print\_array(p,n,\*i2,\*i1);printf("(i2=%d i1=%d)\n",\*i2,\*i1);  }  i\_temp=0;  while(1)  {  while(i\_temp!=\*i2&&p[i\_temp]<p[n-1])  {  ++i\_temp;  }  while(\*i1!=-1&&p[\*i1]==p[n-1])  {  --\*i1;  }  if(i\_temp>=\*i1)break;  print\_indent(indent);printf("|/");print\_array(p,n,i\_temp,\*i1);printf("(i\_temp=%d i1=%d)\n",i\_temp,\*i1);  temp\_swap=p[i\_temp]; p[i\_temp]=p[\*i1]; p[\*i1]=temp\_swap; ++i\_temp; --\*i1;  print\_indent(indent);printf("|\\");print\_array(p,n,i\_temp,\*i1);printf("(i\_temp=%d i1=%d)\n",i\_temp,\*i1);  }  print\_indent(indent);printf("|/");print\_array(p,n,\*i2,\*i1);printf("(i2=%d i1=%d)\n",\*i2,\*i1);  temp\_swap=p[\*i2]; p[\*i2]=p[n-1]; p[n-1]=temp\_swap; ++\*i2;  print\_indent(indent);printf("|\\");print\_array(p,n,\*i2,\*i1);printf("(i2=%d)\n",\*i2);  print\_indent(indent);printf("\\\n");  }  void sort(int\*p,int n,int indent=0)  {  int i1,i2;  print\_indent(indent);printf("(n=%d)",n);printf("\n");  split(p,n,&i1,&i2,indent);  if(i1>0)sort(p,i1+1,indent+2);  if(n-i2-1>0)sort(p+i2,n-i2,indent+2);  } | |
| 选择排序   |  | | --- | | 简单选择排序  void sort(int\*p,int n)  {  int i,j,i\_min,temp\_swap;  for(i=0;i!=n-1;++i)  {  i\_min=i;  for(j=i+1;j!=n;++j)  {  if(p[j]<p[i\_min])  {  i\_min=j;  }  }  if(i\_min!=i)  {  printf("(i=%d i\_min=%d)",i,i\_min);print\_array(p,n,i,i\_min);printf("\n");  temp\_swap=p[i];p[i]=p[i\_min];p[i\_min]=temp\_swap;  printf("(i=%d i\_min=%d)",i,i\_min);print\_array(p,n,i,i\_min);printf("\n");  }  }  } | | 堆排序  堆  void heap\_refresh(int\*p,int n,int current)  {  int child=current\*2+1,temp\_swap;  for(;child<n;current=child,child=child\*2+1)  {  if(child+1<n&&p[child]<p[child+1])++child;  if(p[current]>=p[child])return;  print\_heap(p,n,current,child,4);printf("(current=%d child=%d)",current,child);printf("\n");  temp\_swap=p[current];p[current]=p[child];p[child]=temp\_swap;  print\_heap(p,n,current,child,4);printf("(current=%d child=%d)",current,child);printf("\n\n");  }  }  void sort(int\*p,int n)  {  int i,temp\_swap;  for(i=n/2-1;i!=0;--i)  {  print\_heap(p,n,i,-1);printf("\n");  heap\_refresh(p,n,i);  }  printf("\n");  for(i=n;i!=1;--i)  {  print\_heap(p,i);printf("\n");  heap\_refresh(p,i,0);  print\_array(p,n,0,i-1);printf("\n");  temp\_swap=p[i-1];p[i-1]=p[0];p[0]=temp\_swap;  print\_array(p,n,0,i-1);printf("\n");  }  } | |
| 归并排序  分治法  void sort(int\*p,int n,int indent=0)  {  int i,i1,i2;  int mid=(n-1)/2,n1=mid+1,n2=n-n1;  print\_indent(indent);print\_array(p,n1);printf("+");print\_array(p+n1,n2);printf("(n=%d)\n",n);  if(n1>1)sort(p,n1,indent+3);  if(n2>1)sort(p+n1,n2,indent+3);  print\_indent(indent);print\_array(p,n1);printf("+");print\_array(p+n1,n2);printf("(n=%d)\n",n);  int\*p\_temp=new int[n]();  i=i1=i2=0;  while(i1!=n1&&i2!=n2)  {  if(p[i1]<(p+n1)[i2])  {  p\_temp[i]=p[i1];  print\_indent(indent+2);print\_array(p,n1,i1);print\_array(p+n1,n2,i2);printf("(i1=%d i2=%d)\n",i1,i2);  ++i1;  }  else  {  p\_temp[i]=(p+n1)[i2];  print\_indent(indent+2);print\_array(p,n1,i1);print\_array(p+n1,n2,i2);printf("(i1=%d i2=%d)\n",i1,i2);  ++i2;  }  ++i;  print\_indent(indent+2);print\_array(p\_temp,i,i-1);printf("\n");  }  for(;i1!=n1;++i1,++i)  {  p\_temp[i]=p[i1];  }  for(;i2!=n2;++i2,++i)  {  p\_temp[i]=(p+n1)[i2];  }  for(i=0;i!=n;++i)  {  p[i]=p\_temp[i];  }  print\_indent(indent);print\_array(p,n);printf("(n=%d)\n",n);  delete[]p\_temp;  } |
| 分配排序   |  | | --- | | 基数排序  前缀和 | | 桶排序  void sort(int\*p,int n,int max)//[0,max)  {  int\*p\_temp=new int[max]();  int i,j=0;  for(i=0;i!=n;++i)  {  ++(p\_temp[p[i]]);  }  for(i=0;i!=max;++i)  {  while(p\_temp[i]>0)  {  p[j]=i;  ++j;  --(p\_temp[i]);  printf("(i=%d p2[i]=%d)",i,p\_temp[i]);print\_array(p,j,j-1);printf("\n");  }  }  delete[]p\_temp;  } | |

顺序统计量

|  |
| --- |
| int main()  {  int i,j,max=100;  const int n=10;  int a[n];  srand(time(NULL));  for(i=0;i!=n;++i)  {  a[i]=rand()%max;  }  print\_array(a,n);printf("\n");  i=rand()%10;  printf("(i=%d)\n\n",i);  printf("%d\n\n",select(a,n,i));  } |

|  |
| --- |
| int split(int\*p,int n,int indent=0)  {  print\_indent(indent);printf("/\n");  int i,j,temp\_swap;  print\_indent(indent);printf("|");print\_array(p,n,-1,n-1);printf("\n");  i=0;j=n-2;  print\_indent(indent);printf("|");printf("(i=%d j=%d)\n",i,j);  while(1)  {  while(i!=n-1&&p[i]<=p[n-1])  {  ++i;  }  while(j!=-1&&p[j]>=p[n-1])  {  --j;  }  if(i<j)  {  temp\_swap=p[i];p[i]=p[j];p[j]=temp\_swap;  print\_indent(indent);printf("|");print\_array(p,n,i,j);printf("(i=%d j=%d)",i,j);printf("\n");  }  else  {  break;  }  }  print\_indent(indent);printf("|");printf("(i=%d j=%d)\n",i,j);  temp\_swap=p[i];p[i]=p[n-1];p[n-1]=temp\_swap;  print\_indent(indent);printf("|");print\_array(p,n,i,n-1);printf("\n");  print\_indent(indent);printf("\\\n");  return i;  }  int select(int\*p,int n,int i,int indent=0)//i:[0,n)  {  if(n==1)  {  return p[0];  }  print\_indent(indent);printf("(i=%d)\n",i);  print\_indent(indent);print\_array(p,n);printf("\n");  int n2=split(p,n,indent+1);  print\_indent(indent);print\_array(p,n,n2);printf("\n");  if(i==n2)  {  return p[n2];  }  else  {  if(i<n2)  {  print\_indent(indent);print\_array(p,n,0,n2-1);printf("\n");  return select(p,n2,i,indent+1);  }  else  {  print\_indent(indent);print\_array(p,n,n2+1,n-1);printf("\n");  return select(p+n2+1,n-n2-1,i-n2-1,indent+1);  }  }  } |

数论

|  |
| --- |
| gcd  #include<stdio.h>  #include<stdlib.h>  #include<time.h>  int gcd1(int i,int j,int&n\_recursion)//i>=j  {  ++n\_recursion;  int r;  if((r=i%j)==0)  {  return j;  }  else  {  return(r<j-r)  ?gcd1(j-r,r,n\_recursion)  :gcd1(r,j-r,n\_recursion);  }  }  int gcd2(int i,int j,int&n\_recursion)  {  ++n\_recursion;  return j?gcd2(j,i%j,n\_recursion):i;  }  int main()  {  srand(time(NULL));  int i,temp1,temp2,n\_recursion;  for(i=20;i!=40;++i)  {  temp1=rand()%1000;  temp2=rand()%1000+1;  n\_recursion=0;  printf("(%d,%d)=%d\t",temp1,temp2,gcd1(temp1,temp2,n\_recursion));  printf("%d\t",n\_recursion);  n\_recursion=0;  printf("(%d,%d)=%d\t",temp1,temp2,gcd2(temp1,temp2,n\_recursion));  printf("%d\n",n\_recursion);  }  } |
| ex\_gcd  #include<stdio.h>  #include<stdlib.h>  #include<time.h>  void print\_indent(int indent=0)  {  int i;  for(i=0;i!=indent;++i)  {  printf(" ");  }  }  void ex\_gcd(int a,int b,int&x,int&y,int&gcd,int indent=0)  {  if(a%b==0)  {  gcd=b;  x=0;  y=1;  print\_indent(indent);printf("%d \* %d + %d \* %d = %d\n",a,x,b,y,gcd);  }  else  {  int tempx,tempy;  ex\_gcd(b,a%b,tempx,tempy,gcd,indent+2);  print\_indent(indent);printf("%d \* %d + (%d - %d / %d) \* %d = %d\n",b,tempx,a,a,b,tempy,gcd);  x=tempy;  y=tempx-a/b\*tempy;  print\_indent(indent);printf("%d \* %d + %d \* %d = %d\n",a,x,b,y,gcd);  }  }  int main()  {  int i,a,b,x,y,gcd;  srand(time(NULL));  for(i=0;i!=5;++i)  {  a=rand()%500+200;  b=rand()%500+200;  ex\_gcd(a,b,x,y,gcd);  printf("\n");  }  } |
|  |
|  |

1. 数据结构
   1. 可合并堆
      1. 斜堆

|  |
| --- |
| /\*  \* 可合并堆/斜堆：两堆合并时，将较大堆合并到较小堆的右子树，然后交换较小堆的左右子树（防止右子树一直变大）  \*/  #include<stdio.h>  #include<stdlib.h>  struct skew\_heap  {  skew\_heap\*p\_left,\*p\_right;  int key;  skew\_heap(int key):key(key),p\_left(NULL),p\_right(NULL){}  };  skew\_heap\*merge(skew\_heap\*p1,skew\_heap\*p2)  {  if(p1==NULL)  {  return p2;  }  if(p2==NULL)  {  return p1;  }  if(p1->key>p2->key)  {  skew\_heap\*p\_temp=p1;p1=p2;p2=p\_temp;  }  p1->p\_right=merge(p1->p\_right,p2);  skew\_heap\*p\_temp=p1->p\_left;p1->p\_left=p1->p\_right;p1->p\_right=p\_temp;  return p1;  }  void insert(skew\_heap\*&p,int key)  {  p=merge(p,new skew\_heap(key));  }  int min(skew\_heap\*p)  {  return p->key;  }  int delete\_min(skew\_heap\*p)  {  int temp=p->key;  p=merge(p->p\_left,p->p\_right);  return temp;  }  void print(skew\_heap\*p,int indent)  {  if(p->p\_left!=NULL)  {  print(p->p\_left,indent+1);  }  for(int i=0;i!=indent;++i)  {  putchar(' ');  }  printf("%d\n",p->key);  if(p->p\_right!=NULL)  {  print(p->p\_right,indent+1);  }  }  int main(void)  {  skew\_heap\*p=NULL;  int i;  for(i=0;i!=30;++i)  {  insert(p,rand());  }  print(p,0);  return 0;  } |

* + 1. 左偏树

|  |
| --- |
| /\*  \* 可合并堆/左偏树：每个结点增加1个npl(NULL Path Length)，表示向右下方走多少步后到达NULL。  \* merge中，将较大堆和较小堆的右子树merge后，将较小堆的左子树和右子树调整成npl小的在右。使树总是左偏，不会退化成单支树。  \*/  #include<stdio.h>  #include<stdlib.h>  #define NPL(x) (x==NULL?0:x->npl)  struct left\_heap  {  left\_heap\*p\_left,\*p\_right;  int key;  int npl;  left\_heap(int key):key(key),npl(1),p\_left(NULL),p\_right(NULL){}  };  left\_heap\*merge(left\_heap\*p1,left\_heap\*p2)  {  if(p1==NULL)  {  return p2;  }  if(p2==NULL)  {  return p1;  }  if(p1->key>p2->key)  {  left\_heap\*p\_temp=p1;p1=p2;p2=p\_temp;  }  p1->p\_right=merge(p1->p\_right,p2);  if(NPL(p1->p\_left)<NPL(p1->p\_right))  {  left\_heap\*p\_temp=p1->p\_left;p1->p\_left=p1->p\_right;p1->p\_right=p\_temp;  }  p1->npl=NPL(p1->p\_right)+1;  return p1;  }  void insert(left\_heap\*&p,int key)  {  p=merge(p,new left\_heap(key));  }  int min(left\_heap\*p)  {  return p->key;  }  int delete\_min(left\_heap\*p)  {  int temp=p->key;  p=merge(p->p\_left,p->p\_right);  return temp;  }  void print(left\_heap\*p,int indent)  {  if(p->p\_left!=NULL)  {  print(p->p\_left,indent+2);  }  for(int i=0;i!=indent;++i)  {  putchar(' ');  }  printf("%d\n",p->key);  if(p->p\_right!=NULL)  {  print(p->p\_right,indent+2);  }  }  int main(void)  {  left\_heap\*p=NULL;  int i;  for(i=0;i!=30;++i)  {  insert(p,rand());  }  print(p,0);  return 0;  } |

* + 1. （随机堆）

|  |
| --- |
| /\*  \* 可合并堆/随机堆：随机选择较小堆的左右子树和较大堆合并。  \*/  #include<stdio.h>  #include<stdlib.h>  struct rand\_heap  {  rand\_heap\*p\_left,\*p\_right;  int key;  rand\_heap(int key):key(key),p\_left(NULL),p\_right(NULL){}  }; |

* + 1. （二项堆）

|  |
| --- |
| /\*  \* 可合并堆/二项堆：二项树的森林。二项树有2^k个结点，2^(k+1)个结点的二项树的结构：1个2^k个结点的二项树的根结点添加1个孩子，这个孩子是另1个2^k个结点的二项树（画图）。  \* 二项堆：根据结点数的二进制形式中为1的那些位，相应地拥有二项树。  \* 实现：左儿子右兄弟  \*/ |

* 1. 二叉查找树
     1. Treap

|  |
| --- |
| /\*  \* Treap是一种平衡二叉树，每个结点有key和priority两个属性，priority是随机生成的  \* Treap是关于priority的堆，关于key的二叉排序树  \*/  #include<stdio.h>  #include<stdlib.h>  struct treap  {  treap\*p\_left,\*p\_right;  int key,priority;  treap(int key):key(key),priority(rand()%100),p\_left(NULL),p\_right(NULL){}  };  void left\_rotate(treap\*&p)  {  treap\*p\_temp=p->p\_right;  p->p\_right=p\_temp->p\_left;  p\_temp->p\_left=p;  p=p\_temp;  }  void right\_rotate(treap\*&p)  {  treap\*p\_temp=p->p\_left;  p->p\_left=p\_temp->p\_right;  p\_temp->p\_right=p;  p=p\_temp;  }  void insert(treap\*&p,int key)  {  treap\*p\_new=new treap(key);  if(p==NULL)  {  p=p\_new;  }  else  {  if(key<p->key)  {  insert(p->p\_left,key);  if(p->priority>p->p\_left->priority)  {  right\_rotate(p);  }  }  else  {  insert(p->p\_right,key);  if(p->priority>p->p\_right->priority)  {  left\_rotate(p);  }  }  }  }  void del(treap\*&p,int key)  {  if(p->key==key)  {  if(p->p\_left==NULL)  {  p=p->p\_right;  }  else if(p->p\_right==NULL)  {  p=p->p\_left;  }  else  {  if(p->p\_left->priority<p->p\_right->priority)  {  right\_rotate(p);  del(p->p\_right,key);  }  else  {  left\_rotate(p);  del(p->p\_left,key);  }  }  }  }  void print(treap\*p,int indent)  {  if(p->p\_left!=NULL)  {  print(p->p\_left,indent+2);  }  for(int i=0;i!=indent;++i)  {  putchar(' ');  }  printf("%d,%d\n",p->key,p->priority);  if(p->p\_right!=NULL)  {  print(p->p\_right,indent+2);  }  }  int main(void)  {  treap\*p=NULL;  int i;  for(i=0;i!=30;++i)  {  insert(p,rand()%50);  }  print(p,0);  return 0;  } |

* + 1. splay树

|  |
| --- |
| /\*  \* splay树：查找操作，将查到的结点经过多次旋转移到根节点，理由是查找的结点很有可能再被查找。  \* 根据祖父结点和父节点分，旋转有6种情况：根节点的左子结点、根节点的右子结点、（从上到下）LL、LR、RL、RR  \*/  #include<stdio.h>  #include<stdlib.h>  struct splay\_node  {  splay\_node\*p\_parent;  int value;  splay\_node\*p\_left,\*p\_right;  splay\_node(int value):p\_parent(NULL),value(value),p\_left(NULL),p\_right(NULL){}  };  void rotate\_left(splay\_node\*&p\_root,splay\_node\*p\_node)//p\_node的父结点向左下移动。前提：p\_node的父结点存在，p\_node是父结点的右孩子  {  splay\_node\*p\_parent=p\_node->p\_parent,\*p\_grandparent=p\_parent->p\_parent;  p\_parent->p\_right=p\_node->p\_left;  if(p\_node->p\_left!=NULL)  {  p\_node->p\_left->p\_parent=p\_parent;  }  p\_parent->p\_parent=p\_node;  p\_node->p\_left=p\_parent;  p\_node->p\_parent=p\_grandparent;  if(p\_grandparent!=NULL)  {  if(p\_grandparent->p\_left==p\_parent)  {  p\_grandparent->p\_left=p\_node;  }  else  {  p\_grandparent->p\_right=p\_node;  }  }  else  {  p\_root=p\_node;  }  }  void rotate\_right(splay\_node\*&p\_root,splay\_node\*p\_node)//p\_node的父结点向右下移动。前提：p\_node的父结点存在，p\_node是父结点的左孩子  {  splay\_node\*p\_parent=p\_node->p\_parent,\*p\_grandparent=p\_parent->p\_parent;  p\_parent->p\_left=p\_node->p\_right;  if(p\_node->p\_right!=NULL)  {  p\_node->p\_right->p\_parent=p\_parent;  }  p\_parent->p\_parent=p\_node;  p\_node->p\_right=p\_parent;  p\_node->p\_parent=p\_grandparent;  if(p\_grandparent!=NULL)  {  if(p\_grandparent->p\_left==p\_parent)  {  p\_grandparent->p\_left=p\_node;  }  else  {  p\_grandparent->p\_right=p\_node;  }  }  else  {  p\_root=p\_node;  }  }  void up\_LL(splay\_node\*&p\_root,splay\_node\*p\_node)//前提：p\_node的父结点存在，p\_node的祖父结点存在，父结点是祖父结点的左孩子，p\_node是父结点的左孩子  {  rotate\_right(p\_root,p\_node->p\_parent);  rotate\_right(p\_root,p\_node);  }  void up\_RR(splay\_node\*&p\_root,splay\_node\*p\_node)//前提：p\_node的父结点存在，p\_node的祖父结点存在，父结点是祖父结点的右孩子，p\_node是父结点的右孩子  {  rotate\_left(p\_root,p\_node->p\_parent);  rotate\_left(p\_root,p\_node);  }  void up\_LR(splay\_node\*&p\_root,splay\_node\*p\_node)//前提：p\_node的父结点存在，p\_node的祖父结点存在，父结点是祖父结点的左孩子，p\_node是父结点的右孩子  {  rotate\_left(p\_root,p\_node);  rotate\_right(p\_root,p\_node);  }  void up\_RL(splay\_node\*&p\_root,splay\_node\*p\_node)//前提：p\_node的父结点存在，p\_node的祖父结点存在，父结点是祖父结点的右孩子，p\_node是父结点的左孩子  {  rotate\_right(p\_root,p\_node);  rotate\_left(p\_root,p\_node);  }  splay\_node\*\*search(splay\_node\*&p,int value,splay\_node\*&p\_parent)  {  if(p==NULL||p->value==value)  {  return &p;  }  else if(p->value>value)  {  return search(p->p\_left,value,p\_parent=p);  }  else  {  return search(p->p\_right,value,p\_parent=p);  }  }  void splay\_tree(splay\_node\*&p\_root,splay\_node\*p\_node)//将p\_node提升至成为根结点。前提：p\_root不等于NULL，p\_node不等于p\_root（真的需要提升时才调用）  {  while(p\_root!=p\_node&&p\_root->p\_left!=p\_node&&p\_root->p\_right!=p\_node)  {  //将p\_node提升2层，前提：p\_node不在第0层，p\_node不在第1层  if(p\_node->p\_parent==p\_node->p\_parent->p\_parent->p\_left)  {  if(p\_node==p\_node->p\_parent->p\_left)  {  up\_LL(p\_root,p\_node);  }  else  {  up\_LR(p\_root,p\_node);  }  }  else  {  if(p\_node==p\_node->p\_parent->p\_left)  {  up\_RL(p\_root,p\_node);  }  else  {  up\_RR(p\_root,p\_node);  }  }  }  //将p\_node提升1层，前提：p\_node在第1层  if(p\_root->p\_left==p\_node)  {  rotate\_right(p\_root,p\_node);  }  else if(p\_root->p\_right==p\_node)  {  rotate\_left(p\_root,p\_node);  }  }  void search\_splay(splay\_node\*&p\_root,int value)  {  splay\_node\*p\_parent,\*\*p\_p\_find=search(p\_root,value,p\_parent);  if(\*p\_p\_find!=NULL&&\*p\_p\_find!=p\_root)//查到结果，且结果不在第0层（真的需要提升时才调用）  {  splay\_tree(p\_root,\*p\_p\_find);  }  }  void insert(splay\_node\*&p\_root,int value)  {  splay\_node\*p\_parent=NULL,\*\*p\_p\_find=search(p\_root,value,p\_parent);  if(\*p\_p\_find==NULL)  {  \*p\_p\_find=new splay\_node(value);  (\*p\_p\_find)->p\_parent=p\_parent;  }  }  void remove(splay\_node\*&p\_root,int value)  {  splay\_node\*p\_parent,\*\*p\_p\_find=search(p\_root,value,p\_parent);  if(\*p\_p\_find!=NULL)  {  //if(\*p\_p\_find!=p\_root)//查到结果，且结果不在第0层（真的需要提升时才调用）  //{  // splay\_tree(p\_root,\*p\_p\_find);  //}  if((\*p\_p\_find)->p\_left==NULL)  {  splay\_node\*p\_temp=\*p\_p\_find;  \*p\_p\_find=p\_temp->p\_right;  if(p\_temp->p\_right!=NULL)  {  p\_temp->p\_right->p\_parent=p\_temp->p\_parent;  }  delete p\_temp;  }  else if((\*p\_p\_find)->p\_right==NULL)  {  splay\_node\*p\_temp=\*p\_p\_find;  \*p\_p\_find=p\_temp->p\_left;  if(p\_temp->p\_left!=NULL)  {  p\_temp->p\_left->p\_parent=p\_temp->p\_parent;  }  delete p\_temp;  }  else  {  splay\_node\*p\_temp=(\*p\_p\_find)->p\_left;  while(p\_temp->p\_right!=NULL)  {  p\_temp=p\_temp->p\_right;  }  if(p\_temp==(\*p\_p\_find)->p\_left)  {  p\_temp=\*p\_p\_find;  \*p\_p\_find=p\_temp->p\_left;p\_temp->p\_left->p\_parent=p\_temp->p\_parent;  (\*p\_p\_find)->p\_right=p\_temp->p\_right;p\_temp->p\_right->p\_parent=\*p\_p\_find;  delete p\_temp;  }  else  {  p\_temp->p\_parent->p\_right=p\_temp->p\_left;  if(p\_temp->p\_left!=NULL)  {  p\_temp->p\_left->p\_parent=p\_temp->p\_parent;  }  p\_temp->p\_left=(\*p\_p\_find)->p\_left;  p\_temp->p\_right=(\*p\_p\_find)->p\_right;  p\_temp->p\_parent=(\*p\_p\_find)->p\_parent;  \*p\_p\_find=p\_temp;p\_temp=p\_temp->p\_left->p\_parent;  (\*p\_p\_find)->p\_left->p\_parent=\*p\_p\_find;  (\*p\_p\_find)->p\_right->p\_parent=\*p\_p\_find;  delete p\_temp;  }  }  }  }  void print(splay\_node\*p,int indent)  {  if(p->p\_left!=NULL)  {  print(p->p\_left,indent+2);  }  for(int i=0;i!=indent;++i)  {  putchar(' ');  }  printf("%d\n",p->value);  if(p->p\_right!=NULL)  {  print(p->p\_right,indent+2);  }  }  int main(void)  {  splay\_node\*p=NULL;  int i;  for(i=0;i!=30;++i)  {  insert(p,rand()%50);  }  print(p,0);  putchar('\n');  int temp;  while(true)  {  scanf("%d",&temp);  remove(p,temp);  print(p,0);putchar('\n');  }  return 0;  } |
| //https://blog.csdn.net/mu0206mu/article/details/70185944  //splay树不要父结点指针的实现 |

* 1. 并查集

|  |
| --- |
| /\*  \* 并查集：1个一维数组，第i个元素存储第i个结点的前驱的下标  \*/  #include<stdio.h>  #include<stdlib.h>  #include<time.h>  #include<string.h>  void decircle(size\_t\*pre,size\_t n)  {  for(size\_t i=0;i!=n;++i)  {  size\_t i1=i,i2=pre[i1],i3=pre[i2];  while(true)  {  pre[i1]=i3;  if(pre[i3]==i1)  {  pre[i3]=i3;  break;  }  if(i3==i1)  {  pre[i2]=i3;  break;  }  i1=i2;i2=i3;i3=pre[i3];  }  }  }  size\_t search(size\_t\*pre,size\_t index)  {  size\_t root=index;  while(pre[root]!=root)  {  root=pre[root];  }  size\_t i1=index,i2;  while(i1!=root)  {  i2=pre[i1];  pre[i1]=root;  i1=i2;  }  return root;  }  void join(size\_t\*pre,size\_t index1,size\_t index2)  {  pre[index2]=index1;  }  void print(size\_t\*pre,size\_t n)  {  for(size\_t i=0;i!=n;++i)  {  for(int j=0;j!=pre[i];++j)  {  putchar(' ');  putchar(' ');  }  printf("%u\n",i);  }  }  void print2(size\_t\*p1,size\_t\*p2,size\_t n)  {  for(size\_t i=0;i!=n;++i)  {  for(int j=0;j!=p1[i];++j)  {  putchar(' ');  putchar(' ');  }  printf("%u(%u)\n",i,p2[i]);  }  }  int main(void)  {  size\_t pre[30],n=30;  srand(time(NULL));  for(int i=0;i!=n;++i)  {  pre[i]=rand()%n;  }  size\_t pre2[30];memcpy(pre2,pre,n\*sizeof(size\_t));  decircle(pre2,n);  print2(pre,pre2,n);  int temp;  while(true)  {  scanf("%d",&temp);  search(pre,temp);  print(pre,n);putchar('\n');  }  return 0;  } |

* 1. 集合计数问题

|  |
| --- |
| /\*  \* 集合计数问题：启发式合并算法  \*/  #include<stdio.h>  #include<stdlib.h>  struct node  {  size\_t pre;  int n;  };  const size\_t n=10000;  struct node nodes[n];  size\_t get\_root(size\_t index)  {  if(nodes[index].pre!=index)  {  nodes[index].pre=get\_root(nodes[index].pre);  }  return nodes[index].pre;  }  void merge(size\_t index1,size\_t index2)  {  size\_t root1=get\_root(index1),root2=get\_root(index2);  if(root1!=root2)  {  if(nodes[root1].n<nodes[root2].n)  {  nodes[root1].pre=root2;  nodes[root2].n+=nodes[root1].n;  }  else  {  nodes[root2].pre=root1;  nodes[root1].n+=nodes[root2].n;  }  }  }  int main(void)  {  for(size\_t i=0;i!=n;++i)  {  nodes[i].pre=i;  nodes[i].n=1;  }  size\_t index\_max=0,n\_max=0;  int temp1,temp2;  while(scanf("%d%d",&temp1,&temp2)!=-1)  {  merge(temp1,temp2);  if(temp1>index\_max)  {  index\_max=temp1;  }  if(temp2>index\_max)  {  index\_max=temp2;  }  }  for(size\_t i=0;i!=index\_max;++i)  {  if(nodes[i].n>n\_max)  {  n\_max=nodes[i].n;  }  }  printf("%d",(int)n\_max);  return 0;  } |

* 1. 二分图的识别

|  |
| --- |
| /\*  \* 二分图的判别：上色法找奇环。每次先有1个没遍历过的结点，上色后入队，出队后对所有相邻结点上色后（没遍历过的）入队，遇到相邻结点颜色相同就return false  \*/  #include<stdio.h>  #include<stdlib.h>  const int n=8;  int node\_color[n]={};//0:none 1:2:not none  int edges[n][n]={};  int node\_traversed[n]={};  struct node  {  int index;  struct node\*p\_next;  };  struct queue  {  struct node\*p\_start,\*p\_end;  };  void queue\_init(struct queue&q)  {  q.p\_start=q.p\_end=new node();  q.p\_start->p\_next=NULL;  }  void queue\_in(struct queue&q,int index)  {  q.p\_end->p\_next=new node;  q.p\_end=q.p\_end->p\_next;  q.p\_end->index=index;q.p\_end->p\_next=NULL;  }  int queue\_out(struct queue&q)  {  int index;  if(q.p\_start->p\_next==q.p\_end)  {  index=q.p\_end->index;  q.p\_end=q.p\_start;  delete q.p\_start->p\_next;q.p\_start->p\_next=NULL;  return index;  }  else  {  struct node\*p=q.p\_start->p\_next;  q.p\_start->p\_next=q.p\_start->p\_next->p\_next;  index=p->index;  delete p;  return index;  }  }  void queue\_dest(struct queue&q)  {  struct node\*p;  while(q.p\_start!=NULL)  {  p=q.p\_start;  q.p\_start=q.p\_start->p\_next;  delete p;  }  }  bool queue\_is\_empty(struct queue&q)  {  return q.p\_start==q.p\_end;  }  bool valid(int index)  {  struct queue q1;  queue\_init(q1);  queue\_in(q1,index);  while(!queue\_is\_empty(q1))  {  int index=queue\_out(q1);  node\_traversed[index]=true;  for(int i=0;i!=n;++i)  {  if(edges[index][i]!=0)  {  if(node\_color[i]==0)  {  node\_color[i]=3-node\_color[index];  }  else if(node\_color[i]==node\_color[index])  {  return false;  }  if(!node\_traversed[i])  {  queue\_in(q1,i);  }  }  }  }  return true;  }  int main(void)  {  int temp1,temp2;  while(scanf("%d%d",&temp1,&temp2)!=-1)  {  edges[temp1][temp2]=edges[temp2][temp1]=1;  }  int i;  for(i=0;i!=n;++i)  {  if(0==node\_traversed[i])  {  node\_color[i]=1;  if(!valid(i))  {  break;  }  }  }  if(i==n)  {  printf("valid");  }  else  {  printf("invalid");  }  return 0;  } |

* 1. 平衡二叉树
     1. （红黑树）
     2. avl树

|  |
| --- |
| /\*  \* avl树，使用平衡因子的版本  \* https://blog.csdn.net/freeelinux/article/details/52204851  \*/  #include<stdio.h>  #include<stdlib.h>  struct avl\_node;  struct node  {  struct avl\_node\*\*value;  struct node\*p\_next;  node(struct avl\_node\*\*value):value(value),p\_next(NULL){}  };  void stack\_init(struct node\*&p)  {  p=NULL;  }  void stack\_push(struct node\*&p,struct avl\_node\*\*value)  {  struct node\*p\_temp=new node(value);  p\_temp->p\_next=p;  p=p\_temp;  }  struct avl\_node\*\*stack\_pop(struct node\*&p)  {  struct avl\_node\*\*value=p->value;  struct node\*p\_temp=p;  p=p->p\_next;  delete p\_temp;  return value;  }  void stack\_dest(struct node\*&p)  {  struct node\*p\_temp;  while(p!=NULL)  {  p\_temp=p;  p=p->p\_next;  delete p\_temp;  }  }  bool stack\_is\_empty(struct node\*&p)  {  return p==NULL;  }  struct avl\_node\*\*stack\_top(struct node\*&p)  {  return p->value;  }  struct avl\_node  {  int value;  int balance\_factor;//左减右  struct avl\_node\*p\_left,\*p\_right;  avl\_node(int value):value(value),balance\_factor(0),p\_left(NULL),p\_right(NULL){}  };  void rotate\_left(avl\_node\*\*p\_p\_node)  {  avl\_node\*p\_temp=(\*p\_p\_node)->p\_right;  (\*p\_p\_node)->p\_right=p\_temp->p\_left;  p\_temp->p\_left=(\*p\_p\_node);  (\*p\_p\_node)=p\_temp;  }  void rotate\_right(avl\_node\*\*p\_p\_node)  {  avl\_node\*p\_temp=(\*p\_p\_node)->p\_left;  (\*p\_p\_node)->p\_left=p\_temp->p\_right;  p\_temp->p\_right=(\*p\_p\_node);  (\*p\_p\_node)=p\_temp;  }  void insert(avl\_node\*\*p\_p\_root,int value)  {  if(\*p\_p\_root==NULL)  {  \*p\_p\_root=new avl\_node(value);  }  else  {  avl\_node\*\*p\_p\_temp=p\_p\_root,\*\*p\_p\_temp\_parent=NULL;  struct node\*p;stack\_init(p);  while(true)  {  p\_p\_temp\_parent=p\_p\_temp;  stack\_push(p,p\_p\_temp\_parent);  if(value<(\*p\_p\_temp)->value)  {  p\_p\_temp=&((\*p\_p\_temp)->p\_left);  }  else if(value==(\*p\_p\_temp)->value)  {  stack\_dest(p);  return;  }  else  {  p\_p\_temp=&((\*p\_p\_temp)->p\_right);  }  if(\*p\_p\_temp==NULL)  {  \*p\_p\_temp=new avl\_node(value);  break;  }  }  while(!stack\_is\_empty(p))  {  p\_p\_temp\_parent=stack\_pop(p);  if(&((\*p\_p\_temp\_parent)->p\_left)==p\_p\_temp)  {  ++((\*p\_p\_temp\_parent)->balance\_factor);  }  else  {  --((\*p\_p\_temp\_parent)->balance\_factor);  }  //刚才p\_p\_temp\_parent的1个子树（p\_p\_temp）高度增加1，可能出现：  // p\_p\_temp\_parent没失衡且不可能影响上方结点(break)  // p\_p\_temp\_parent没失衡且可能影响上方结点(p\_p\_temp=p\_p\_temp\_parent，下一轮循环一开始p\_p\_temp\_parent=stack.pop)  // p\_p\_temp\_parent失衡(-2,+2)，处理后，不可能影响上方结点(break)  if((\*p\_p\_temp\_parent)->balance\_factor==0)  {  break;  }  else if((\*p\_p\_temp\_parent)->balance\_factor==-1||(\*p\_p\_temp\_parent)->balance\_factor==1)  {  p\_p\_temp=p\_p\_temp\_parent;  }  else//！！！画图  {  if ((\*p\_p\_temp)->balance\_factor == (\*p\_p\_temp\_parent)->balance\_factor/2)  {  if ((\*p\_p\_temp)->balance\_factor == 1)//结点左偏重，父结点左偏重  {  rotate\_right(p\_p\_temp\_parent);  (\*p\_p\_temp\_parent)->balance\_factor = (\*p\_p\_temp\_parent)->p\_right->balance\_factor = 0;  }  else//结点右偏重，父结点右偏重  {  rotate\_left(p\_p\_temp\_parent);  (\*p\_p\_temp\_parent)->balance\_factor = (\*p\_p\_temp\_parent)->p\_left->balance\_factor = 0;  }  }  else  {  if ((\*p\_p\_temp)->balance\_factor == 1)//结点左偏重，父结点右偏重  {  rotate\_right(&((\*p\_p\_temp\_parent)->p\_right));  rotate\_left(p\_p\_temp\_parent);  }  else//结点右偏重，父结点左偏重  {  rotate\_left(&((\*p\_p\_temp\_parent)->p\_left));  rotate\_right(p\_p\_temp\_parent);  }  switch ((\*p\_p\_temp\_parent)->balance\_factor)  {  case-1:  (\*p\_p\_temp\_parent)->p\_left->balance\_factor = 1;  (\*p\_p\_temp\_parent)->p\_right->balance\_factor = 0;  break;  case 0://（画图）4个子树高度都是0时，就是这种情况  (\*p\_p\_temp\_parent)->p\_left->balance\_factor = (\*p\_p\_temp\_parent)->p\_right->balance\_factor = 0;  break;  case 1:  (\*p\_p\_temp\_parent)->p\_left->balance\_factor = 0;  (\*p\_p\_temp\_parent)->p\_right->balance\_factor = -1;  break;  }  (\*p\_p\_temp\_parent)->balance\_factor = 0;  }  break;  }  }//while  stack\_dest(p);  }  }  void remove(avl\_node\*\*p\_p\_root,int value)  {  avl\_node\*\*p\_p\_temp=p\_p\_root,\*\*p\_p\_temp\_parent=NULL;  struct node\*p;stack\_init(p);  while(true)  {  if (\*p\_p\_temp == NULL)  {  stack\_dest(p);  return;  }  else if(value==(\*p\_p\_temp)->value)  {  break;  }  else  {  p\_p\_temp\_parent=p\_p\_temp;  stack\_push(p,p\_p\_temp\_parent);  if(value<(\*p\_p\_temp)->value)  {  p\_p\_temp=&((\*p\_p\_temp)->p\_left);  }  else  {  p\_p\_temp=&((\*p\_p\_temp)->p\_right);  }  }  }  //p\_p\_temp的左右子树都不为空时，转化成删除p\_p\_temp的中序前驱：  // p\_p\_temp.value=p\_p\_temp的中序前驱.value  // p\_p\_temp=p\_p\_temp的中序前驱  if((\*p\_p\_temp)->p\_left!=NULL&&(\*p\_p\_temp)->p\_right!=NULL)  {  avl\_node\*\*p\_p\_temp2=p\_p\_temp;  //向左下走1步  {  p\_p\_temp\_parent = p\_p\_temp;  stack\_push(p, p\_p\_temp\_parent);  p\_p\_temp = &((\*p\_p\_temp)->p\_left);  }  //向右下走至少0步  while((\*p\_p\_temp)->p\_right!=NULL)  {  p\_p\_temp\_parent=p\_p\_temp;  stack\_push(p,p\_p\_temp\_parent);  p\_p\_temp=&((\*p\_p\_temp)->p\_right);  }  (\*p\_p\_temp2)->value=(\*p\_p\_temp)->value;  }  //现在开始删除p\_p\_temp指向的结点  avl\_node\*p\_delete=\*p\_p\_temp;  avl\_node\*\*p\_p\_temp\_child;  if((\*p\_p\_temp)->p\_left!=NULL)  {  p\_p\_temp\_child=&((\*p\_p\_temp)->p\_left);  }  else  {  p\_p\_temp\_child=&((\*p\_p\_temp)->p\_right);  }  if(p\_p\_temp\_parent==NULL)//删的是根结点且根结点孩子数<=1  {  \*p\_p\_root=\*p\_p\_temp\_child;  }  else  {  bool is\_left1=&((\*p\_p\_temp\_parent)->p\_left)==p\_p\_temp;  if(is\_left1)  {  (\*p\_p\_temp\_parent)->p\_left=\*p\_p\_temp\_child;  }  else  {  (\*p\_p\_temp\_parent)->p\_right=\*p\_p\_temp\_child;  }  while(!stack\_is\_empty(p))  {  //每轮循环：刚才p\_p\_temp\_parent1个子树高度减小1  //is\_left1指出p\_p\_temp\_parent哪个子树高度减小1  //可能出现：  // p\_p\_temp\_parent没失衡(0 -> -1,+1)且不可能影响上方结点(break)  // p\_p\_temp\_parent没失衡(-1,+1 -> 0)且可能影响上方结点(p\_p\_temp\_child=p\_p\_temp\_parent)  // p\_p\_temp\_parent失衡(-2,+2)，  // 较高子树平衡因子是0，处理后不可能影响上方结点(break)  // 较高子树平衡因子不是0，处理后可能影响上方结点(p\_p\_temp\_child=p\_p\_temp\_parent)  p\_p\_temp\_parent=stack\_pop(p);  if(is\_left1)  {  --((\*p\_p\_temp\_parent)->balance\_factor);  }  else  {  ++((\*p\_p\_temp\_parent)->balance\_factor);  }  if(!stack\_is\_empty(p))  {  avl\_node\*\*p\_p\_temp\_grand\_parent=stack\_top(p);  if(&((\*p\_p\_temp\_grand\_parent)->p\_left)==p\_p\_temp\_parent)  {  is\_left1=true;  }  else if(&((\*p\_p\_temp\_grand\_parent)->p\_right)==p\_p\_temp\_parent)  {  is\_left1=false;  }  }  if((\*p\_p\_temp\_parent)->balance\_factor==-1||(\*p\_p\_temp\_parent)->balance\_factor==1)  {  break;  }  else if((\*p\_p\_temp\_parent)->balance\_factor!=0)//p\_p\_temp\_parent失衡(-2,+2)  {  //p\_p\_temp\_child=p\_p\_temp\_parent的较高子树  if((\*p\_p\_temp\_parent)->balance\_factor>0)  {  p\_p\_temp\_child=&((\*p\_p\_temp\_parent)->p\_left);  }  else  {  p\_p\_temp\_child=&((\*p\_p\_temp\_parent)->p\_right);  }  //！！！画图  if((\*p\_p\_temp\_child)->balance\_factor==0)  {  if((\*p\_p\_temp\_parent)->balance\_factor==2)  {  rotate\_right(p\_p\_temp\_parent);  (\*p\_p\_temp\_parent)->balance\_factor=-1;  (\*p\_p\_temp\_parent)->p\_right->balance\_factor=1;  }  else  {  rotate\_left(p\_p\_temp\_parent);  (\*p\_p\_temp\_parent)->balance\_factor=1;  (\*p\_p\_temp\_parent)->p\_left->balance\_factor=-1;  }  break;  }  else if((\*p\_p\_temp\_child)->balance\_factor==(\*p\_p\_temp\_parent)->balance\_factor/2)  {  if((\*p\_p\_temp\_parent)->balance\_factor==2)//结点左偏重，子结点左偏重  {  rotate\_right(p\_p\_temp\_parent);  (\*p\_p\_temp\_parent)->balance\_factor=(\*p\_p\_temp\_parent)->p\_right->balance\_factor=0;  }  else//结点右偏重，子结点右偏重  {  rotate\_left(p\_p\_temp\_parent);  (\*p\_p\_temp\_parent)->balance\_factor=(\*p\_p\_temp\_parent)->p\_left->balance\_factor=0;  }  }  else  {  if((\*p\_p\_temp\_parent)->balance\_factor==2)//结点左偏重，子结点右偏重  {  rotate\_left(&((\*p\_p\_temp\_parent)->p\_left));  rotate\_right(p\_p\_temp\_parent);  }  else//结点右偏重，子结点左偏重  {  rotate\_right(&((\*p\_p\_temp\_parent)->p\_right));  rotate\_left(p\_p\_temp\_parent);  }  switch((\*p\_p\_temp\_parent)->balance\_factor)  {  case-1:  (\*p\_p\_temp\_parent)->p\_left->balance\_factor=1;  (\*p\_p\_temp\_parent)->p\_right->balance\_factor=0;  break;  case 0:  (\*p\_p\_temp\_parent)->p\_left->balance\_factor=(\*p\_p\_temp\_parent)->p\_right->balance\_factor=0;  break;  case 1:  (\*p\_p\_temp\_parent)->p\_left->balance\_factor=0;  (\*p\_p\_temp\_parent)->p\_right->balance\_factor=-1;  break;  }  (\*p\_p\_temp\_parent)->balance\_factor=0;  }  }  p\_p\_temp\_child=p\_p\_temp\_parent;  }//while  }  delete p\_delete;  stack\_dest(p);  }  void print(avl\_node\*p,int indent)  {  if(p->p\_left!=NULL)  {  print(p->p\_left,indent+2);  }  for(int i=0;i!=indent;++i)  {  putchar(' ');  }  printf("%d[%d]\n",p->value,p->balance\_factor);  if(p->p\_right!=NULL)  {  print(p->p\_right,indent+2);  }  }  int main(void)  {  avl\_node\*p\_root=NULL;  for(int i=0;i!=50;++i)  {  insert(&p\_root,rand()%50);  }  print(p\_root,0);  int value;  while(true)  {  scanf("%d",&value);  remove(&p\_root,value);  //system("cls");  print(p\_root,0);  }  return 0;  } |
| #include<stdio.h>  #include<stdlib.h>  struct avl\_node;  struct node  {  struct avl\_node\*value;  struct node\*p\_next;  node(struct avl\_node\*value):value(value),p\_next(NULL){}  };  void stack\_init(struct node\*&p)  {  p=NULL;  }  void stack\_push(struct node\*&p,struct avl\_node\*value)  {  struct node\*p\_temp=new node(value);  p\_temp->p\_next=p;  p=p\_temp;  }  struct avl\_node\*stack\_pop(struct node\*&p)  {  struct avl\_node\*value=p->value;  struct node\*p\_temp=p;  p=p->p\_next;  delete p\_temp;  return value;  }  void stack\_dest(struct node\*&p)  {  struct node\*p\_temp;  while(p!=NULL)  {  p\_temp=p;  p=p->p\_next;  delete p\_temp;  }  }  bool stack\_is\_empty(struct node\*&p)  {  return p==NULL;  }  struct avl\_node\*stack\_top(struct node\*&p)  {  return p->value;  }  struct avl\_node  {  int value;  int balance\_factor;//左减右  struct avl\_node\*p\_left,\*p\_right;  avl\_node(int value):value(value),balance\_factor(0),p\_left(NULL),p\_right(NULL){}  };  void rotate\_left(avl\_node\*&p\_node)  {  avl\_node\*p\_temp=p\_node->p\_right;  p\_node->p\_right=p\_temp->p\_left;  p\_temp->p\_left=p\_node;  p\_node=p\_temp;  }  void rotate\_right(avl\_node\*&p\_node)  {  avl\_node\*p\_temp=p\_node->p\_left;  p\_node->p\_left=p\_temp->p\_right;  p\_temp->p\_right=p\_node;  p\_node=p\_temp;  }  void insert(avl\_node\*&p\_root,int value)  {  if(p\_root==NULL)  {  p\_root=new avl\_node(value);  }  else  {  avl\_node\*p\_temp=p\_root,\*p\_temp\_parent=NULL;  struct node\*p;stack\_init(p);  while(true)  {  p\_temp\_parent=p\_temp;  stack\_push(p,p\_temp\_parent);  if(value<p\_temp->value)  {  p\_temp=p\_temp->p\_left;  if(p\_temp==NULL)  {  p\_temp=p\_temp\_parent->p\_left=new avl\_node(value);  break;  }  }  else if(value==p\_temp->value)  {  stack\_dest(p);  return;  }  else  {  p\_temp=p\_temp->p\_right;  if(p\_temp==NULL)  {  p\_temp=p\_temp\_parent->p\_right=new avl\_node(value);  break;  }  }  }  while(!stack\_is\_empty(p))  {  p\_temp\_parent=stack\_pop(p);  if(p\_temp\_parent->p\_left==p\_temp)  {  ++(p\_temp\_parent->balance\_factor);  }  else  {  --(p\_temp\_parent->balance\_factor);  }  if(p\_temp\_parent->balance\_factor==0)  {  break;  }  else if(p\_temp\_parent->balance\_factor==-1||p\_temp\_parent->balance\_factor==1)  {  p\_temp=p\_temp\_parent;  }  else//！！！画图  {  if (p\_temp->balance\_factor == p\_temp\_parent->balance\_factor/2)  {  if (p\_temp->balance\_factor == 1)//结点左偏重,父结点左偏重  {  rotate\_right(p\_temp\_parent);  p\_temp\_parent->balance\_factor = p\_temp\_parent->p\_right->balance\_factor = 0;  }  else//结点右偏重,父结点右偏重  {  rotate\_left(p\_temp\_parent);  p\_temp\_parent->balance\_factor = p\_temp\_parent->p\_left->balance\_factor = 0;  }  }  else  {  if (p\_temp->balance\_factor == 1)//结点左偏重,父结点右偏重  {  rotate\_right(p\_temp\_parent->p\_right);  rotate\_left(p\_temp\_parent);  }  else//结点右偏重,父结点左偏重  {  rotate\_left(p\_temp\_parent->p\_left);  rotate\_right(p\_temp\_parent);  }  switch (p\_temp\_parent->balance\_factor)  {  case-1:  p\_temp\_parent->p\_left->balance\_factor = 1;  p\_temp\_parent->p\_right->balance\_factor = 0;  break;  case 0://（画图）4个子树高度都是0时，就是这种情况  p\_temp\_parent->p\_left->balance\_factor = p\_temp\_parent->p\_right->balance\_factor = 0;  break;  case 1:  p\_temp\_parent->p\_left->balance\_factor = 0;  p\_temp\_parent->p\_right->balance\_factor = -1;  break;  }  p\_temp\_parent->balance\_factor = 0;  }  break;  }  }//while  //局部变量被rotate()更新过，是正确的。树上的指针还需要更新：  if(stack\_is\_empty(p))  {  p\_root=p\_temp\_parent;  }  else  {  p\_temp=stack\_top(p);  if(p\_temp\_parent->value<p\_temp->value)  {  p\_temp->p\_left=p\_temp\_parent;  }  else  {  p\_temp->p\_right=p\_temp\_parent;  }  }  stack\_dest(p);  }  }  void remove(avl\_node\*&p\_root,int value)  {  avl\_node\*p\_temp=p\_root,\*p\_temp\_parent=NULL;  struct node\*p;stack\_init(p);  while(true)  {  if (p\_temp == NULL)  {  stack\_dest(p);  return;  }  else if(value==p\_temp->value)  {  break;  }  else  {  p\_temp\_parent=p\_temp;  stack\_push(p,p\_temp\_parent);  if(value<p\_temp->value)  {  p\_temp=p\_temp->p\_left;  }  else  {  p\_temp=p\_temp->p\_right;  }  }  }  if(p\_temp->p\_left!=NULL&&p\_temp->p\_right!=NULL)//p\_temp.value=p\_temp的中序前驱.value,p\_temp=p\_temp的中序前驱  {  avl\_node\*p\_temp2=p\_temp;  {  p\_temp\_parent = p\_temp;  stack\_push(p, p\_temp\_parent);  p\_temp = p\_temp->p\_left;  }  while(p\_temp->p\_right!=NULL)  {  p\_temp\_parent=p\_temp;  stack\_push(p,p\_temp\_parent);  p\_temp=p\_temp->p\_right;  }  p\_temp2->value=p\_temp->value;  }  //现在开始删除p\_temp指向的结点  avl\_node\*p\_temp\_child;  if(p\_temp->p\_left!=NULL)  {  p\_temp\_child=p\_temp->p\_left;  }  else  {  p\_temp\_child=p\_temp->p\_right;  }  if(p\_temp\_parent==NULL)//删的是根结点且根结点孩子数<=1  {  p\_root=p\_temp\_child;  }  else  {  avl\_node\*p\_temp\_grand\_parent;  int is\_left0;  bool is\_left1=p\_temp\_parent->p\_left==p\_temp;  if(is\_left1)  {  p\_temp\_parent->p\_left=p\_temp\_child;  }  else  {  p\_temp\_parent->p\_right=p\_temp\_child;  }  while(!stack\_is\_empty(p))  {  p\_temp\_parent=stack\_pop(p);  if(is\_left1)  {  --(p\_temp\_parent->balance\_factor);  }  else  {  ++(p\_temp\_parent->balance\_factor);  }  if(!stack\_is\_empty(p))  {  p\_temp\_grand\_parent=stack\_top(p);  if(p\_temp\_grand\_parent->p\_left==p\_temp\_parent)  {  is\_left0=-1;  is\_left1=true;  }  else if(p\_temp\_grand\_parent->p\_right==p\_temp\_parent)  {  is\_left0=1;  is\_left1=false;  }  }  else  {  //p\_temp\_grand\_parent未初始化  is\_left0=0;  }  if(p\_temp\_parent->balance\_factor==-1||p\_temp\_parent->balance\_factor==1)//p\_temp\_parent本来平衡因子是0，现在1个子树变矮。不会再对上层结点产生任何影响了。  {  break;  }  else if(p\_temp\_parent->balance\_factor!=0)//p\_temp\_parent本来平衡因子是-1或1，现在变成-2或2  {  if(p\_temp\_parent->balance\_factor>0)//p\_temp\_child=p\_temp\_parent的较高子树  {  p\_temp\_child=p\_temp\_parent->p\_left;  }  else  {  p\_temp\_child=p\_temp\_parent->p\_right;  }  if(p\_temp\_child->balance\_factor==0)  {  if(p\_temp\_parent->balance\_factor==2)  {  rotate\_right(p\_temp\_parent);  p\_temp\_parent->balance\_factor=-1;  p\_temp\_parent->p\_right->balance\_factor=1;  }  else  {  rotate\_left(p\_temp\_parent);  p\_temp\_parent->balance\_factor=1;  p\_temp\_parent->p\_left->balance\_factor=-1;  }  break;  }  else if(p\_temp\_child->balance\_factor==p\_temp\_parent->balance\_factor/2)  {  if(p\_temp\_parent->balance\_factor==2)//子结点左偏重，结点左偏重  {  rotate\_right(p\_temp\_parent);  p\_temp\_parent->balance\_factor=p\_temp\_parent->p\_right->balance\_factor=0;  }  else//子结点右偏重，结点右偏重  {  rotate\_left(p\_temp\_parent);  p\_temp\_parent->balance\_factor=p\_temp\_parent->p\_left->balance\_factor=0;  }  }  else  {  if(p\_temp\_parent->balance\_factor==2)//子结点右偏重，结点左偏重  {  rotate\_left(p\_temp\_parent->p\_left);  rotate\_right(p\_temp\_parent);  }  else//子结点左偏重，结点右偏重  {  rotate\_right(p\_temp\_parent->p\_right);  rotate\_left(p\_temp\_parent);  }  switch(p\_temp\_parent->balance\_factor)  {  case-1:  p\_temp\_parent->p\_left->balance\_factor=1;  p\_temp\_parent->p\_right->balance\_factor=0;  break;  case 0:  p\_temp\_parent->p\_left->balance\_factor=p\_temp\_parent->p\_right->balance\_factor=0;  break;  case 1:  p\_temp\_parent->p\_left->balance\_factor=0;  p\_temp\_parent->p\_right->balance\_factor=-1;  break;  }  p\_temp\_parent->balance\_factor=0;  }  //局部变量被rotate()更新过，是正确的。树上的指针还需要更新：  switch (is\_left0)  {  case-1:  p\_temp\_grand\_parent->p\_left = p\_temp\_parent;  break;  case 1:  p\_temp\_grand\_parent->p\_right = p\_temp\_parent;  break;  }  p\_temp\_child=p\_temp\_parent;  }  }//while  //局部变量被rotate()更新过，是正确的。树上的指针还需要更新：  if(stack\_is\_empty(p))  {  p\_root=p\_temp\_parent;  }  else  {  switch (is\_left0)  {  case-1:  p\_temp\_grand\_parent->p\_left = p\_temp\_parent;  break;  case 1:  p\_temp\_grand\_parent->p\_right = p\_temp\_parent;  break;  }  }  }  delete p\_temp;  stack\_dest(p);  }  void print(avl\_node\*p,int indent)  {  if(p->p\_left!=NULL)  {  print(p->p\_left,indent+2);  }  for(int i=0;i!=indent;++i)  {  putchar(' ');  }  printf("%d[%d]\n",p->value,p->balance\_factor);  if(p->p\_right!=NULL)  {  print(p->p\_right,indent+2);  }  }  int main(void)  {  avl\_node\*p\_root=NULL;  for(int i=0;i!=50;++i)  {  insert(p\_root,rand()%50);  }  print(p\_root,0);  int value;  while(true)  {  scanf("%d",&value);  remove(p\_root,value);  //system("cls");  print(p\_root,0);  }  return 0;  } |
| /\*  \* avl树，使用树高的版本  \*/  //略 |