插入排序

```cpp
void insertSort(int data[], int n) {
    int buffer = 0;
    for (int i = 1; i < n; i++) {      // 从第二个元素起进行插入操作
        buffer = data[i];              // 缓存该元素
        int j = i - 1;                 // 初始化比较元素位置为前一个
        while(j >= 0 && buffer < data[j]){ //该次比较继续进行条件
            data[j + 1] = data[j];     //交换元素
            j -= 1;                    //继续下一个位置比较
        }
        data[j + 1] = buffer;          //比较停止，将插入值插入对应位置
    }
}
```

//稳定算法

复杂度：最好情况 O（n），最坏/平均情况：O（n^2）

归并排序

```cpp
template <typename T> //向量归并排序
void Vector<T>::mergeSort ( Rank lo, Rank hi ) { //0 <= lo < hi <= size
if ( hi - lo < 2 ) return; //单元素区间自然有序，否则...
   int mi = ( lo + hi ) / 2; //以中点为界
   mergeSort ( lo, mi ); mergeSort ( mi, hi ); //分别排序
   merge ( lo, mi, hi ); //归并
}
template <typename T> //有序向量的归并
void Vector<T>::merge ( Rank lo, Rank mi, Rank hi ) { //各自有序的子向量[lo, mi)和[mi, hi)
   T* A = _elem + lo; //合并后的向量A[0, hi - lo) = _elem[lo, hi)
   int lb = mi - lo; T* B = new T[lb]; //前子向量B[0, lb) = _elem[lo, mi)
   for ( Rank i = 0; i < lb; B[i] = A[i++] ); //复制前子向量
   int lc = hi - mi; T* C = _elem + mi; //后子向量C[0, lc) = _elem[mi, hi)
   for ( Rank i = 0, j = 0, k = 0; (j<lb)||(k <lc);) { //B[j]和C[k]中的小者续至A末尾
      if ( ( j < lb ) && ( ! ( k < lc ) || ( B[j] <= C[k] ) ) ) A[i++] = B[j++];
      if ( ( k < lc ) && ( ! ( j < lb ) || ( C[k] <  B[j] ) ) ) A[i++] = C[k++];
   }
   delete [] B; //释放临时空间B
} //归并后得到完整的有序向量[lo, hi)
```

//分治递归，稳定算法，需辅助空间 O（n）

复杂度：两路归并 merge O(n)

设排序时间为 T(n)，对长度为 n 的向量归并排序，需完成 2 长度为

n/2 向量的归并排序和一两路归并：T(n) = 2*T(n/2) + O(n)

边界条件：T(1) = 1 => T(n) = O(nlogn)

冒泡排序

```
template <typename T>
void Vector<T>::bubbleSort(Rank lo, Rank hi) {
        int times = 0; bool exchange = true; // 从第一趟开始
        int nSort = hi - lo;
        while (times < nSort && exchange) {
                exchange = false;       // 某趟是否有交换的标志，初始为无交换
                for (int j = hi-1; j > lo + times; j--)
                                        //从最后元素开始到第一个未排序元素
                        if (_elem[j - 1]>_elem[j]) {//若需要交换则置换元素
                                T temp = _elem[j - 1];
                                _elem[j - 1] = _elem[j];
                                _elem[j] = temp;
                                exchange = true;
                        }
                times++;
        }
```

//稳定算法

复杂度：最好情况 O（n），最坏/平均情况：O（n^2）

选择排序：

```
template <typename T>
void Vector<T>::selectSort(Rank lo, Rank hi) {
        for (Rank i = hi - 1; i > lo; i--) { // 从后往前
                int max = i;
                for (Rank j = 0; j < i + 1; j++) {
                                        // 遍历前面未排序，选择最大元素
                        if (_elem[j] > _elem[max])
                                max = j;
                }
                if (max != i) {                 // 交换
                        T temp = _elem[i];
                        _elem[i] = _elem[max];
                        _elem[max] = temp;
                }
        }
}
```

//不稳定

复杂度：最好最坏均为 O（n^2）

希尔排序

```
void ShellSort(int data[], int count){
    int step = 0;
    int auxiliary = 0;
    for (step = count / 2; step > 0; step /= 2){  // 从数组第step个元素开始
        for (int i = step; i < count; i++){      // 每个元素与自己组内的数据进行直接插入排序
            if (data[i] < data[i - step]){      // 插入排序的第一次判断
                auxiliary = data[i];            // 需要往前插入，对待插入数据进行缓存
                int j = i - step;
                while (j >= 0 && data[j] > auxiliary){ //对同组前面数据检测，所大则循环后移
                    data[j + step] = data[j];
                    j -= step;
                }
                data[j + step] = auxiliary;          // 插入数据
            }
        }
    }
}
```

排序方法	平均情况	最好情况	最差情况	辅助空间	稳定性
希尔排序	$O(n\log^2 n)\sim O(n^2)$	$O(n\log^2 n)$	$O(n^2)$	$O(1)$	不稳定

快速排序

partition

```
void quickSort(int data[], int l, int r){
    if (l< r){
        int pivotL = l, pivotR = r, x = data[l];
        while (pivotL < pivotR){
            while (pivotL<pivotR && data[pivotR]>x) pivotR --;
                  //从右向左找第一个小于x的数
            if (pivotL<pivotR) data[pivotL ++] = data[pivotR];

            while (pivotL<pivotR && data[pivotL]<x) pivotL++;
                 //从左向右找第一个大于等于x的数
            if (pivotL<pivotR) data[pivotR --] = data[pivotL];
        }
        data[pivotL] = x;
        quickSort(data, l, pivotL - 1);    // 递归调用处理左子序列
        quickSort(data, pivotL + 1, r);    // 递归调用处理右子序列
    }
}
```

//不稳定，存储开销 O（logn）

复杂度：最好/平均情况 O（nlogn）最坏情况 O（n^2）

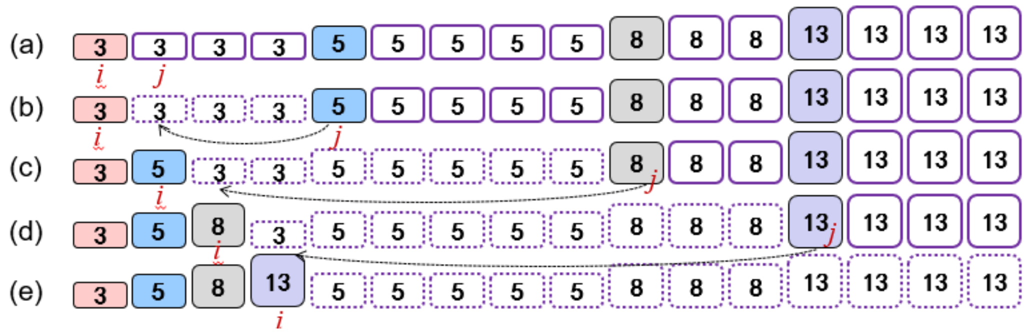
//研究表明，当排序序列长度<25 时，采用直接插入排序要比快速排序至少快 10%

有序向量唯一化（高效）

■ 唯一化(高效版)

如何实现时间复杂度O(n)?

```
template <typename T>
int Vector<T>::uniquify() {                //有序向量重复元素剔除算法（高效版）
    Rank i = 0, j = 0;                     //各对互异“相邻”元素的秩
    while ( ++j < _size )                  //逐一扫描，直至末元素
        if ( _elem[i] != _elem[j] )        //跳过雷同者
            _elem[++i] = _elem[j];         //发现不同元素时，向前移至紧邻于前者右侧
    _size = ++i;
    shrink();                              //直接截除尾部多余元素
    return j - i;                          //向量规模变化量，即被删除元素总数
}
```

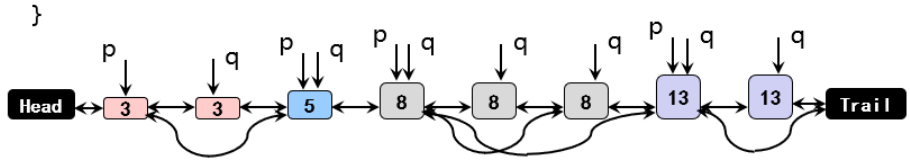


有序向量二分查找

```
// 二分查找算法（版本A）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= _size
template <typename T>
static Rank binSearch ( T* A, T const& e, Rank lo, Rank hi ) {
    while ( lo < hi ) {                         //每步迭代可能做两次比较判断，三个分
        Rank mi = ( lo + hi ) >> 1;             //以中点为轴点
        if( e < A[mi] ) hi = mi;                //深入前半段[lo, mi)继续查找
        else if ( A[mi] < e ) lo = mi + 1;      //深入后半段(mi, hi)继续查找
        else return mi;                         //在mi处命中
    }                                           //成功查找可以提前终止
    return -1;                                  //查找失败
} //多个命中元素时，不能保证返回秩最大者；查找失败时，简单地返回-1，而不能指示失败位置
```

有序列表唯一化（高效）

```
template <typename T>
int List<T>::uniquify() {                          //成批剔除重复元素，效率更高
    if ( _size < 2 ) return 0;                     //平凡列表自然无重复
    int oldSize = _size;                           //记录原规模
    ListNodePosi(T) p = first();
    ListNodePosi(T) q;                             //p为各区段起点，q为其后继
    while ( trailer != ( q = p->succ ) )           //反复考查紧邻的节点对(p, q)
        if ( p->data != q->data ) p = q;           //若互异，则转向下一区段
        else remove ( q );                         //否则（雷同），删除后者
    return oldSize - _size;                        //列表规模变化量，即被删除元素总数
}
```

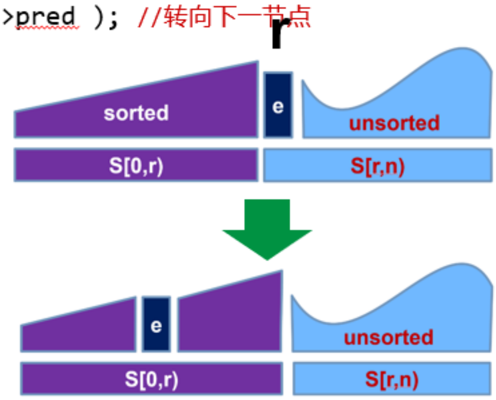


比较相邻两元素是否一致，指针p和q分别指向相邻节点，若二者相同则删除q，否则转向下一对邻节点，O(n)复杂度

列表排序

■ 插入排序

```
template <typename T> //列表的插入排序算法：对起始于位置p的n个元素排序
void List<T>::insertionSort ( ListNodePosi(T) p, int n )
{
      for ( int r = 0; r < n; r++ ) { //逐一为各节点
           insertA ( search ( p->data, r, p ), p->data );
                                            //查找适当的位置并插入
           p = p->succ; remove ( p->pred ); //转向下一节点
   }
}
```



- ✓ 有多个元素命中时，search接口总返回其中最大者，故排序之后重复元素保持其顺序，属稳定算法
- ✓ 查找算法所需复杂度在O(1)到O(n)间浮动
- ✓ 最好情况O(n)，最坏情况O(n²),总体复杂度O(n²)

■ 选择排序

```
template <typename T> //列表的选择排序算法：对起始于位置p的n个元素排序
void List<T>::selectionSort ( ListNodePosi(T) p, int n )
{  ListNodePosi(T) head = p->pred; ListNodePosi(T) tail = p;
   for ( int i = 0; i < n; i++ ) tail = tail->succ;
                                       //待排序区间为(head, tail)
   while ( 1 < n ) {            //在至少还剩两个节点之前，在待排序区间内
      ListNodePosi(T) max = selectMax ( head->succ, n );
                                      //找出最大者（歧义时后者优先）
      insertB ( tail, remove ( max ) );
                          //将其移至无序区间末尾（作为有序区间新的首元素）
      tail = tail->pred;
      n--;
   }
}
```

SelectMax需遍历整个无序前缀，故复杂度为O(n)，加上常数复杂度的移位，外层再进行n次的循环，总复杂度为O(n²)

■ 归并排序：归并函数部分

不需额外空间，属于就地排序

```
template <typename T>
//有序列表的归并：当前列表中自p起的n个元素，与列表L中自q起的m个元素归并
void List<T>::merge ( ListNodePosi(T) & p, int n, List<T>& L,
ListNodePosi(T) q, int m ) {
      ListNodePosi(T) pp = p->pred;
                              //借助前驱（可能是header），以便返回前 ...
      while ( 0 < m ) //在q尚未移出区间之前
      if ( ( 0 < n ) && ( p->data <= q->data ) )
                                    //若p仍在区间内且v(p) <= v(q)，则
         { if ( q == ( p = p->succ ) ) break; n--; }
                                   //p归入合并的列表，并替换为其直接后继
      else                         //若p已超出右界或v(q) < v(p)，则
         { insertB ( p, L.remove ( ( q = q->succ )->pred ) ); m--; }
                                                //将q转移至p之前
      p = pp->succ;                         //确定归并后区间的（新）起点
}
```

将一有序列表L中起始于节点q,长度为m的子序列，与当前有序列表中起始于节点p，长度为n的子列表做二路归并。复杂度O（m+n）

```
template <typename T> //列表的归并排序算法：对起始于位置p的n个元素排序
void List<T>::mergeSort ( ListNodePosi(T) & p, int n ) {
   if ( n < 2 ) return;      //若待排序范围已足够小，则直接返回；否则...
   int m = n >> 1;                                  //以中点为界
   ListNodePosi(T) q = p;
   for ( int i = 0; i < m; i++ ) q = q->succ; //均分列表
   mergeSort ( p, m );
   mergeSort ( q, n - m );                       //对前、后子列表分别排序
   merge ( p, m, *this, q, n - m );              //归并
}                           //注意：排序后，p依然指向归并后区间的（新）起点
```

- ✓ **两路归并merge时间复杂度：O(n)**
- ✓ **均分列表复杂度：O(n)**
- ✓ **对长度为n的向量归并排序，需完成2长度为n/2向量的归并排序，一两路归并，一均分操作：**

  $T(n) = 2*T(n/2) + mO(n)$
- ✓ **边界条件：$T(1) = 1$**

$T(n) = 2*T(n/2) + mO(n)$

$T(n)/n=T(n/2)/(n/2)+O(m)$

$=T(n/4)/(n/4)+O(2m)$

$=T(n/2^k)/(n/2^k)+O(km)$

当$n=2^k$时，$k=\log n$,

$T(n)/n=mO(\log n)$,m为常数

$T(n)=O(n\log n)$

总结：列表排序

排序方法	最好时间	平均时间	最坏时间	辅助空间	稳定性
插入	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
冒泡	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定
归并	O(nlgn)	O(nlgn)	O(nlgn)	O(1)	稳定

- 插入排序：逐个×(查找+插入)=O(n)×(O(n)+O(1)) = $O(n^2)$
  二分查找无法降低至O(logn) 插入后无移动代价
- 选择排序：逐个×(选择+插入)=O(n)×(O(n)+O(1)) = $O(n^2)$
- 冒泡排序：逐遍×逐个×相邻交换=O(n)×O(n)×O(1) = $O(n^2)$
- 归并排序：递归复杂度公式证明O(nlogn) 就地排序，归并排序更适合用列表实现

栈混洗

```
bool stackPermutation(B[1,n]){
        Stack S;
        int i = 1;
        for k=1 to n {
            while(S.empty()||B[k]!=S.top())
                if(i>n) return false;
                else S.push(i++);
            S.pop();
        }
        return true;
}
```

表达式求值

```
float evaluate ( char* S, char*& RPN ) {
   Stack<float> opnd; Stack<char> optr;
   char* expr = S;
   optr.push ( '\0' ); displayProgress(expr, S, opnd, optr, RPN);
   while ( !optr.empty() ) {
      if ( isdigit ( *S ) ) {
         readNumber ( S, opnd ); append ( RPN, opnd.top() );
      } else //若当前字符为运算符，则
         switch ( orderBetween ( optr.top(), *S ) ) {
            case '<': //栈顶运算符优先级更低时
               optr.push ( *S ); S++; //计算推迟，当前运算符进栈
               break;
            case '=':
               optr.pop(); S++;
               break;
            case '>': {
               char op = optr.pop(); append ( RPN, op );
               if ( '!' == op ) { //若属于一元运算符
                  float pOpnd = opnd.pop();
                  opnd.push ( calcu ( op, pOpnd ) );
               } else { //对于其它（二元）运算符
                  float pOpnd2 = opnd.pop(), pOpnd1 = opnd.pop();
                  opnd.push ( calcu ( pOpnd1, op, pOpnd2 ) ); /
               }
               break;
            }
            default: exit(-1); //逢语法错误，不做处理直接退出
         } //switch
     displayProgress(expr, S, opnd, optr, RPN);
   } //while
   return opnd.pop(); //弹出并返回最后的计算结果
}
```

二叉树层次遍历 BFS

```
template <typename T> template <typename VST> //元素类型、操作器
void BinNode<T>::travLevel ( VST& visit ) { //二叉树层次遍历算法
   Queue<BinNodePosi> Q;               //辅助队列
   Q.enqueue ( this );                 //根节点入队
   while ( !Q.empty() ) {              //在队列再次变空之前，反复迭代
      BinNodePosi x = Q.dequeue();
      visit ( x->data );               //取出队首节点并访问之
      if ( HasLChild ( *x ) ) Q.enqueue ( x->lc ); //左孩子入队
      if ( HasRChild ( *x ) ) Q.enqueue ( x->rc ); //右孩子入队
   }
}
```

初始化时令根入队，随后进入循环。每一步迭代中，取出队首节点，然后其左右孩子入队。一旦试图在下一迭代前发现队列为空，遍历即告完成。复杂度O(n)

二叉树先/中/后序遍历 DFS

先序遍历（迭代）

```
template <typename T, typename VST> void
travPre_I1 ( BinNodePosi x, VST& visit ) {
   Stack<BinNodePosi(T)> S;
   if ( x ) S.push ( x );
   while ( !S.empty() ) {
      x = S.pop(); visit ( x->data );
      if ( HasRChild ( *x ) )
            S.push ( x->rc );
      if ( HasLChild ( *x ) )
            S.push ( x->lc );
   }
}
```

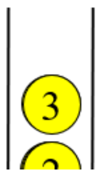
先序遍历（递归）

```
template <typename T, typename VST>
void travPre
( BinNodePosi x, VST& visit )
{
   if ( !x ) return;
   visit ( x->data );
   travPre ( x->lc, visit );
   travPre ( x->rc, visit );
}
```

复杂度：时间 O（n），空间 O（h）

中序遍历（迭代）

```
template <typename T, typename VST> void
travIn_I2 (BinNodePosi x, VST& visit){
   Stack<BinNodePosi(T)> S;
   while ( true )
      if ( x ) {
         S.push ( x );
         x = x->lc;
      }
      else if ( !S.empty() ) {
         x = S.pop();
         visit ( x->data );
         x = x->rc;
      }
      else
         break;
}
```



二叉树重构

```
void Recon(BinNode* p, int pre0, int pre1, int in0, int in1){
  设置当前子树根节点的值 p->data = PreSeq [pre0];
  确定中序序列根节点位置 int root_in = find(InSeq,p->data);
  计算左右子树节点数目 int nL, nR;
  左子树的前序序列起始位置 int preL0 = pre0 + 1;
  左子树的前序序列终止位置 int preL1 = preL0+nL-1;
  左子树的中序序列起始位置 int inL0 = in0;
  左子树的前序序列终止位置 int inL1 = in0+nL-1;
  同样计算右子树的preR0,preR1,inR0,inR1;
  if (nL>0) { //递归调用
      产生新的左根节点pL; Recon(pL,preL0,preL1,inL0,inL1);}
  if (nR>0) {//递归调用
      产生新的左根节点pR; Recon(pR,preR0,preR1,inR0,inR1);}
```

二叉搜索树的查找

复杂度：O（logn）

二叉搜索树的插入/构建

```
bool Insert(int x, BSTNode *&p) {
    if (p == NULL) {
        p = new BSTNode;
        p->data = x;
        p->left = NULL;
        p->right = NULL;
        return true;
    }
    else if (x < p->data)
        Insert(x, p->left);
    else if (x > p->data)
        Insert(x, p->right);
    else return false;
};
```

复杂度：插入时间 O（h）/O（logn） 构建

二叉搜索树的删除

```
bool Remove(int x, BSTNode* &p) {          //在以p为根节点的树中删除元素x
    if (p != NULL) {        //若递归的各层子树不为空，说明顺利删除返回true
        if (x < p->data) Remove(x, p->left);    //递归进入左子树进行删除
        else if (x > p->data) Remove(x, p->right); //递归进入右子树删除
        else if (p->left == NULL||p->right == NULL){//若至多有一个孩子
            BSTNode* temp = p;                // 以temp指向即将删除节点
            if (p->left == NULL)
                p = p->right;        //若左子树空，p地址更新为其右孩子地址
            else p = p->left;        //若右子树空，p地址更新为其左孩子地址
            delete temp;             // 删除节点
        }
        else {                       //左右子树皆存在
            BSTNode* temp = p->right;
            while (temp->left!=NULL) temp = temp->left; //递归寻找右孩
            p->data = temp->data;               //更新该被删除节点的关键码
            Remove(p->data, p->right); // 递归调用删除右子树中的p->data
        }
        return true;                 // 删除成功，返回true
    }
    return false;                    // 树中无关键码x，删除失败
};
```

复杂度：O（logn）

■ 查找、插入、删除复杂度分析

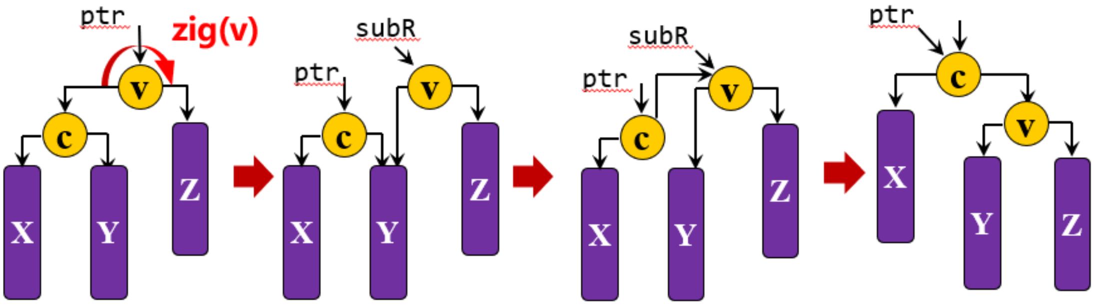
二分查找下复杂度为O(logn)

数据结构	无序向量	无序列表	有序向量	有序列表	二叉搜索树
search(x)	O(n)	O(n)	O(logn) 二分查找	O(n)	O(logn)
insert(x)	O(1) 末端插入	O(1) 末端插入	O(n) 查找|移位	O(n) 查找	O(logn)
remove(x)	O(n) 查找|移位	O(n) 查找	O(n) 查找|移位	O(n) 查找	O(logn)

可否改进有序向量查找的结构，避免移位操作？
利用二叉搜索树（保持平衡性），
可实现平均复杂度为O(logn)的搜索、插入、删除

AVL 平衡化旋转：

■ 核心操作：左单旋与右单旋

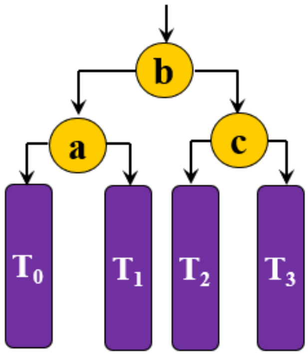



```
void RotateR(Node *& ptr) {//左子树比右子树高，旋转后新根在ptr
    Node *subR = ptr;//要右旋转的结点
    ptr = subR->left;
    subR->left = ptr->right;//转移ptr右边负载
    ptr->right = subR;//ptr成为新根
    ptr->bf = subR->bf = 0; // 修改v和c的平衡因子
};
```

左单旋操作完全对称，同理可进行

AVL 树 3+4 组装：

```
template <typename T> BinNodePosi(T) BST<T>::connect34 (
   BinNodePosi(T) a, BinNodePosi(T) b, BinNodePosi(T) c,
   BinNodePosi(T) T0, BinNodePosi(T) T1, BinNodePosi(T) T2,
BinNodePosi(T) T3
) { // 在确定a,b,c节点及四棵子树情况下组装重构平衡子树
   a->lc = T0; if ( T0 ) T0->parent = a;
   a->rc = T1; if ( T1 ) T1->parent = a;
   updateHeight ( a ); // 组装左子树并更新高度
   c->lc = T2; if ( T2 ) T2->parent = c;
   c->rc = T3; if ( T3 ) T3->parent = c;
   updateHeight ( c ); // 组装右子树并更新高度
   b->lc = a; a->parent = b;
   b->rc = c; c->parent = b;
   updateHeight ( b ); // 组装子树根节点
   return b; //返回该子树新的根节点
}
```



```
template <typename T> BinNodePosi(T)
BST<T>::rotateAt ( BinNodePosi(T) v ) { //v为非空孙辈节点
    BinNodePosi(T) p = v->parent;
    BinNodePosi(T) g = p->parent; //视v、p和g相对位置分四种情况
    if ( IsLChild ( *p ) ) /* zig */
      if ( IsLChild ( *v ) ) { /* zig-zig */
        p->parent = g->parent; //向上联接
        return connect34 (v, p, g, v->lc, v->rc, p->rc, g->rc);
      }
      else { /* zig-zag */
        v->parent = g->parent; //向上联接
        return connect34 (p, v, g, p->lc, v->lc, v->rc, g->rc);
      }
    else  /* zag */
       … … … …
}
```

KD 树

■ 2d-树范围查询实现

```
void KdTree::Search_R(TreeNode* pNode, int d, const RecArea& R,
std::vector<PointStruct>& result) //递归搜索R范围内的点，返回于result
{
    if (!pNode) return;                    // 节点为空返回
    int dimension = d % 2;                 // 获得当前分割轴方向
    int flag = R.intersect(dimension, *pNode); //判断节点与R交叠情况
    if (flag < 0)                          // 只有左子树相交的情况
        Search_R(pNode->m_pLeft, d + 1, R, result);
    else if (flag > 0)                     // 只有右子树相交的情况
        Search_R(pNode->m_pRight, d + 1, R, result);
    else {                                 // 左右子树都相交的情况
        RecArea smallerR, biggerR;
        R.split(dimension, *pNode, smallerR, biggerR); // 划分区域
        Search_R(pNode->m_pLeft, d + 1, smallerR, result);
        Search_R(pNode->m_pRight, d + 1, biggerR, result);
                              // 分别在左右子树搜索，搜索区域划分为小区域
        if (R.include(*pNode)) result.push_back(*pNode);
    } // 后序遍历判断，节点是否属于R
}
```

2d-树最近邻查询实现

```
void KdTree::recurSearch(TreeNode* pNode, int depth, const Coordinate&
point, PointStruct& nearest){
    if (!pNode) return;
    int dimension = depth % 2;
    if (point.distance(*pNode) < point.distance(nearest))
        nearest = *pNode;  // 查询过程，即递归中的先序遍历，先访问根节点数据
    if (point.isSmaller(dimension, *pNode)){
        recurSearch(pNode->m_pLChild, depth + 1, point, nearest);
        if (point.distance(nearest) >= point.distance(dimension,*pNode))
            recurSearch(pNode->m_pRChild, depth + 1, point, nearest);
    }   // 分情况进行左右子树的查询与回溯，目标位于左子树则左子树必进入，
    else{  右子树视回溯过程的垂直距离
        recurSearch(pNode->m_pRChild, depth + 1, point, nearest);
        if (point.distance(nearest) >= point.distance(dimension,*pNode))
        recurSearch(pNode->m_pLChild, depth + 1, point, nearest);
    }
}
```

先序遍历，必进入其中一棵子树分支，另一个分支进入与否依垂直距离判断

堆合并法/Floyd

```
template <class ElemType>
void MyHeap<ElemType>::makeHeap(){
//建堆的过程就是一个不断调整堆的过程，循环调用函数percolateDown调整子树
    if (numCounts < 2) return;
    //第一个需要调整的子树的根节点地址
    int parent = (numCounts) / 2 -1;
    while (1){
        percolateDown(parent, heapDataVec[parent]);
        if (0 == parent) return; // 到达根节点，返回
        --parent;
    }
}
```

■ 复杂度比较：蛮力算法 vs 堆合并法

- ✓ 设高度为 h, 规模为 $n=2^{h+1}-1$ 的满二叉树, 深度为 i 的节点共 2^i 个
- ✓ 蛮力算法：每个节点时间正比于其**深度**

$$\sum_j depth(j) = S(h) = \sum_{i=0}^{h} i \times 2^i$$

$$\rightarrow 2S(h) = \sum_{i=0}^{h} i \times 2^{i+1} = \sum_{i=1}^{h+1} (i-1) \times 2^i$$

$$\rightarrow S(h) = \sum_{i=1}^{h+1} (i-1) \times 2^i - \sum_{i=0}^{h} i \times 2^i = h \cdot 2^{h+1} - 2^{h+1} + 2$$

$$\rightarrow S(h) = (\log_2(n+1) - 2)(n+1) + 2 = O(nlogn)$$

- ✓ 堆合并法：每个节点时间正比于其**高度**

$$\sum_j height(j) = \sum_{i=0}^{h} \left((h-i) \times 2^i\right) = 2^{h+1} - (h+2)$$

$$= n - log_2(n+1) = O(n)$$

■ 堆排序实现

```
template <class ElemType>
void MyHeap<ElemType>::sortHeap()
{
    makeHeap();
    while (numCounts > 0)
        delMax(); // 即为每次循环的置换、下滤过程
```

串 KMP 算法

■ Next表的计算

```
int* buildNext ( char* P ) { //构造模式串P的next表
   size_t m = strlen ( P ), j = 0; //“主”串指针
   int* N = new int[m]; //next表
   int t = N[0] = -1; //模式串指针
   while ( j < m - 1 )
      if ( 0 > t || P[j] == P[t] ) { //匹配
         j ++; t ++;
         N[j] = t;
      }
      else //失配
         t = N[t];
   return N;
}
```

■ 复杂度分析

- ✓ 令k=2*i-j
- ✓ 由while循环的分析，每进行一次while循环，k至少+1
- ✓ k的最大值可能为$2*i_{max}-j_{min}=2n+1$，故while循环至多执行2n+1次。为此，复杂度为O(n)

```
int match ( char* P, char* T ) {                         //KMP算法
   int* next = buildNext ( P );                          //复杂度O(m)
   int n = ( int ) strlen ( T ), i = 0;                  //复杂度O(1)
   int m = ( int ) strlen ( P ), j = 0;                  //复杂度O(1)
   while ( j < m  && i < n )
      if ( 0 > j || T[i] == P[j] )
         { i ++;  j ++; }     //i，j每次加1，等价于k每次加1
      else
         j = next[j];         //i不变，j至少减1，故k至少加1
   delete [] next;                                       //复杂度O(1)
   return i - j;
}
```

图 BFS（队列）

```
void Graph<Tv, Te>::BFS ( int v, int& clock ) {
   Queue<int> Q;
   status ( v ) = DISCOVERED;
   Q.enqueue ( v );
   dTime ( v ) = ++clock;
   while ( !Q.empty() ) {
      int v = Q.dequeue();
      for ( int u = firstNbr ( v );
             -1 < u; u = nextNbr ( v, u ) )
         if ( UNDISCOVERED == status ( u ) ) {
            status ( u ) = DISCOVERED;
            Q.enqueue (u);
```

```
template <typename Tv, typename Te> //广度优先搜索BFS算法（全图）
void Graph<Tv, Te>::bfs ( int s ) {
   reset(); int clock = 0; int v = s; //初始化
   do //逐一检查所有顶点
      if ( UNDISCOVERED == status ( v ) ) //一旦遇到尚未发现的顶点
         BFS ( v, clock ); //即从该顶点出发启动一次BFS
   while ( s != ( v = ( ++v % n ) ) ); //按序号检查，故不漏不重
}
template <typename Tv, typename Te> //广度优先搜索BFS算法（单个连通域）
void Graph<Tv, Te>::BFS ( int v, int& clock ) {
   Queue<int> Q; status ( v ) = DISCOVERED; Q.enqueue ( v ); //初始化起点
   while ( !Q.empty() ) { //在Q变空之前，不断
      int v = Q.dequeue(); dTime ( v ) = ++clock; //取出队首顶点v
      for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v邻居
         if ( UNDISCOVERED == status ( u ) ) { //若u尚未被发现，则
            status ( u ) = DISCOVERED; Q.enqueue ( u ); //发现该顶点
            type ( v, u ) = TREE; parent ( u ) = v; //引入树边拓展支撑树
         } else { //若u已被发现，或者甚至已访问完毕，则
            type ( v, u ) = CROSS; //将(v, u)归类于跨边
         }
      status ( v ) = VISITED; //至此，当前顶点访问完毕
   }
}
```

时间复杂度O(n+e)，可应用于连通域分解、边权值相同下的最短路径

图 DFS（栈）

```
void Graph<Tv, Te>::DFS ( int v) {
   Stack<int> S;
   S.push ( v );
   while ( !S.empty() ) {
      int v = S.pop();
      status ( v ) = DISCOVERED;
      for ( int u = firstNbr ( v );
              -1 < u; u = nextNbr ( v, u ) )
         if ( UNDISCOVERED == status ( u ) )
            S.push (u);
         //   status (u) = DISCOVERED;
         }
```

```
template <typename Tv, typename Te> //深度优先搜索DFS算法（全图）
void Graph<Tv, Te>::dfs ( int s ) {
   reset(); int v = s;                         //初始化
   do                                          //逐一检查所有顶点
      if ( UNDISCOVERED == status ( v ) )      //一旦遇到尚未发现的顶点
         DFS ( v );                            //即从该顶点出发启动一次DFS
   while ( s != ( v = ( ++v % n ) ) );        //按序号检查，故不漏不重
}

template <typename Tv, typename Te> //深度优先搜索DFS算法（单个连通域）
void Graph<Tv, Te>::DFS ( int v) {
   status (v) = DISCOVERED;     //设置v已被访问，此处可加代码，如打印等
   for (int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ))
       if (UNDISCOVERED == status ( u ) )
            DFS ( u );         //深搜v所有未访问过邻居u
   status (v) = VISITED;
}
```

各顶点被访问到（将顶点标记为 DISCOVERED）的次序，类似于

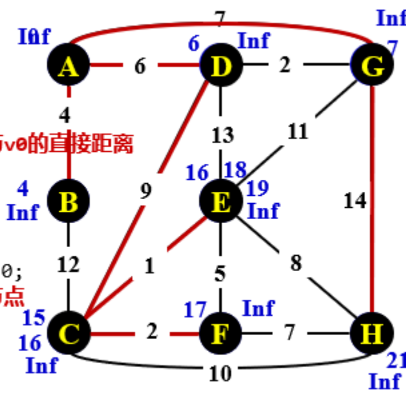
树的先序遍历；各顶点被访问完（将顶点标记为 VISITED）的次序，类似于树的后序遍历

最小生成树 Prim

```
void minSpanTreePrim(MGraph g){
    int min, i, j, k; int pred[MAXVEX];int cutcost[MAXVEX];
    cutcost[0] = 0; //0节点直接加入生成树，其割边权重为0
    precede[0] = 0; //0节点前驱为自己
    for (i = 1; i<g.numVertexes; i++){
        //将节点0加入后，更新其他每个节点的割边权重
        cutcost[i] = g.arc[0][i]; //割边权重更新(初始化)
        pred[i] = 0; // 初始化各节点的前驱为节点0
    }
    for (i = 1; i<g.numVertexes; i++){ //循环nV-1次,查找最小割边加入节点加入
        min = INFINITY; // 设置最小
        for (j = 1; j<g.numVertexes; j++){ // 循环节点1至节点nV-1，判断哪个节点加入生成树
            if (cutcost[j] != 0 && cutcost[j]<min){
                min = cutcost[j]; // 更新最小割边权重为j的割边权重
                k = j; // 设置j为当前生成树准备加入的节点
            }
        }
        printf("(%d,%d)", pred[k], k); //k加入生成树，并输出其前驱
        cutcost[k] = 0;//表示该点k已经加入生成树
        for (j = 1; j<g.numVertexes; j++){//该循环检查在k加入后，各顶点割边是否权重是否更新
            if (cutcost[j] != 0 && g.arc[k][j]<cutcost[j]){
                cutcost[j] = g.arc[k][j]; // j点的割边权重更新
                pred[j] = k; // 设置j点的割边连接的点k
            }
        }
```

时间复杂度O(n²)，可通过优先级队列降低

最短路径树




```
void Dijkstra(MGraph g,int v0){
    int min,i,j,k; bool inTree[MAXVEX];
    int mindist[MAXVEX]; //当前每个节点的最短路径
    int pred[MAXVEX];    //每个节点的前驱
    for (int i = 0; i < g.numVertexes; ++i){
        mindist[i] = g.arc[v0][i]; //节点的初始最短距离为与v0的直接距离
        inTree[i] = false;     //初始都未用过该点
        if (mindist[i] == INFINITY) pred[i] = -1;
        else pred[i] = v0;  //设置前驱
    }
    inTree[v0] = true;  //节点v0为起始顶点  mindist[v0] = 0;
    for (i = 1; i < g.numVertexes; i++){//循环nV-1次加新节点
        min = INFINITY;
        for (j = 0; j < g.numVertexes; ++j)
        if ((!inTree[j]) && mindist[j]<min){
            k = j; // 保存当前节点号
            min = mindist[j]; //更新最小长度
        }
        inTree[k] = true; // 设置节点k进入最短路径树
        printf("(%d,%d,%d)", pred[k], k, mindist[k]); //k加入最短路径树，并输出其前驱
        for (j = 0; j < g.numVertexes; j++) // 循环k的所有邻域节点进行
        if ((!inTree[j]) && g.arc[k][j]<INFINITY){ //k的邻域节点
            if (mindist[k] + g.arc[k][j] < mindist[j]){ //通过k点找更短的路径
                mindist[j] = mindist[k] + g.arc[k][j];  //更新j的最短路径
                pred[j] = k;    //记录j的前驱顶点为k
            }
```

时间复杂度O(n²)，可通过优先级队列降低

■ Dijkstra的优先级队列实现

```
int Dij(int Star, int End){
	Node P, Pn;
	P.end = Star; P.weight = 0;
	priority_queue<Node> Q;
	Q.push(P);              //把顶点放入优先级队列
	while (!Q.empty()){  //至多e次提取(但实际复杂度为n)
	    P = Q.top();  Q.pop(); //提取优先级最高顶点(维护堆序性, 下滤, 复杂度O(logn))
	    if (visited[P.end]) continue;   // 若该顶点被访问过, 则返回
	    visited[P.end] = true; // 设置该顶点访问标记
	    if (P.end == End) return P.weight; // 找到目标节点则返回退出函数
	    int nEdge = G[P.end].size(); // 该顶点的邻域表个数
	    for (int i = 0; i< nEdge; i++){
	        Pn.end = G[P.end][i].end; // 取出第i个邻域顶点的秩
	        Pn.weight = G[P.end][i].weight + P.weight; // 对应权重修改
	        if (!visited[Pn.end])     // 若该邻域顶点未被访问, 则放入队列
	                Q.push(Pn); // 放入该顶点进入优先级队列, 不对重复顶点进行合并,
	    }                       // 每个顶点可能重复放入, 队列中元素至多为边的数目e
	}
	return -1;
}
```

整体时间复杂度: O(eloge)=O(elogn)

■ Bellman和Ford算法实现

```
bool Bellman_Ford(){
    for (int i = 1; i <= nodenum; ++i) //初始化
        dist[i] = (i == original ? 0 : MAX);
    for (int k = 1; k <= nodenum - 1; ++k) //k次迭代
        for (int j = 1; j <= edgenum; ++j)
                          // 对原公式n²次松弛, 简化为e次边的松弛
            if (dist[edge[j].v]>dist[edge[j].u]+edge[j].cost){
                dist[edge[j].v]=dist[edge[j].u]+edge[j].cost;
                pre[edge[j].v] = edge[j].u;
            }
    bool negative = false; //判断是否含有负权回路
    for (int j = 1; j <= edgenum; ++j)
              // 再做一次迭代看是否有任意边可改进, 若是则有负权和回路
        if(dis[edge[j].v] > dis[edge[j].u]+edge[j].cost){
            negative = true; break;}
    return negative;
}
```

```
for (k = 1; k <= n; k++)
   for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
         if (e[i][j]>e[i][k]+e[k][j]){
             e[i][j]=e[i][k]+e[k][j];
             p[i][j]=p[i][k];
         }
```

拓扑排序

```cpp
void Graph<Tv, Te>::TS () {
   Stack<int> S;
   for(int i=0; i<n; i++)
       if(V[i].inDegree==0) S.push(i);
   if(S.size()==0) return false;
   while ( !S.empty() ) {
       int s = S.pop();
       status ( s ) = DISCOVERED;
       for ( int u = firstNbr ( s );
              -1 < u; u = nextNbr ( s, u ) )
           if ( UNDISCOVERED == status ( u ) )
                if((--V[u].inDegree)==0) S.push(u);
        }
   }
   for(int i=0; i<n; i++)
       if(status (s) != DISCOVERED) return false;
   return true;
}
```