Polynomial.h

合并同类项（对顺序表也就是数组的应用） 关键字：系数，指数

```
Polynomial Polynomial::operator+(Polynomial B) {
    Polynomial C;

    int a = start;
    int b = B.start;
    float c; // coefficient of result polynomial

    C.start = free;
    while ( a <= finish && b <= B.finish ){
      switch ( compare ( termArray[a].exp,termArray[b].exp) ) {          //compare exponents
        case '=':                                  //with the same exponents
           c = termArray[a].coef + termArray[b].coef;
           if ( abs(c) >=1.0E-6)
               NewTerm ( c, termArray[a].exp );
           a++;
           b++;
           break;
        case '>' :
           NewTerm ( termArray[b].coef,termArray[b].exp );
           b++;
           break;
       case '<':
           NewTerm ( termArray[a].coef,termArray[a].exp );
           a++;
           break;
       }
    }
     for (    ; a <= finish; a++ )
           NewTerm ( termArray[a].coef,termArray[a].exp );

     for (    ; b <= B.finish; b++ )
           NewTerm ( termArray[b].coef,termArray[b].exp );
     C.finish = free-1;
     return C;
 }
```

Sparse.h

稀疏矩阵（转置代码） 关键字：sparseMatrix

```
sparse_Matrix sparse_Matrix::Transpose() {
	int i,k,currentp;
	sparse_Matrix *b;
	b = (sparse_Matrix *) malloc(sizeof(sparse_Matrix));
	b->cols = rows;
	b->rows = cols;
	b->terms = terms;
	if(b->terms > 0) {
		currentp = 0;
		for(k=0;k<b->rows;k++)      /*select kth row */
			for(i=0;i<terms;i++)
				if(elem[i].get_col() == k)
				{
					b->elem[currentp].set_row(k);
					b->elem[currentp].set_col(elem[i].get_row());
					b->elem[currentp].set_value(elem[i].get_value());
					currentp++;
				}
	}
	return *b;
}
```

括号匹配　关键字：Balancing Symbols

```
int matching(string &exp) {
//exp is a pointer to a string to check
    int state = 1,i=0;
    char e;
    stack<char> s;
    while ( i<exp.length() && state )
        switch (exp[i]) {
            case '[':
            case '(':
                s.push(exp[i]);
                 //push the open character onto stack
                 i++;
                 break;
            case ')':
                if ( !s.empty() && s.top() == '(') {
                    s.pop();    i++; }
                else
                    state = 0;                              //an error occurs
                break;
            case ']':
               if ( !s.empty() && s.top() == '['){
              s.pop();    i++; }
               else
              state = 0;                              // an error occurs
               break;
            default:
               i++; break;
    }       //end of while
    if ( state && s.empty() )    return 1;
    else    return 0;
}
```

四则运算 关键字：（Postorder traversal 后序遍历） abb+-

```
Boolean Calculator::Get2Operands(double &left,double &right){
    if(s. empty( )){
        cerr<<“Missing Operand!”<<endl; return False;
    }
    right = s.top( );s.pop();
    if(s.empty( ))
        cerr<<“Missing Operand!”<<endl; return False;
    }
    left = s.top( );s.pop();
    return True;
  }
void Calculator::DoOperator(char op){
    double left,right;Boolean result;
    result = Get2Operand(left,right);
    if(result == TRUE)
        switch( op ) {
         case’+’:s.push(left + right);break;
         case’-’:s.push(left - right);break;
         case’*’:s.push(left * right);break;
         case’/’:
              if(abs(right) <= 1E-6){
                  cerr << “Divide by 0”<<endl; Clear( );
                 }
             else s.push(left / right);break;
         case’^’:s.push(left ^ right);break;
     }
    else clear( );    }
double Calculator::Run( ){
    char ch; double newoperand,ret;
    while(cin >> ch, ch != ‘=‘) {
        swith( ch ) {
            case’+’:
             case’-’:
             case’*’:
             case’/’:
             case’^’:
                 DoOperator(ch),break;
             default:
                 cin.putback(ch);
                 cin >> newoperand;
                 AddOperand(newoperand);
                 break;     }        }
      ret = s.top();s.pop();return ret;          }
```

Binary Heap 堆

ADT of MinHeap（以最小堆为例 若是最大堆，则将大小于号对调）

```
template <class Type> void MinHeap<Type> ::     //最小堆的向下调整
FilterDown ( int start, int EndOfHeap ) {
    int i = start,    j = 2*i+1;          // j 是 i 的左子女
   Type temp = heap[i];
    while ( j <= EndOfHeap ) {
        if ( j < EndOfHeap && heap[j].key >
            heap[j+1].key )   j++;      //两子女中选小者
        if ( temp.key <= heap[j].key ) break;
        else { heap[i] = heap[j];   i = j;    j = 2*j+1; }
     }
     heap[i] = temp;
}
template <class Type> int MinHeap<Type> ::     //最小堆的插入元素
Insert ( const Type &x ) {
//在堆中插入新元素 x
    if ( CurrentSize == MaxHeapSize )       //堆满
        { cout << "堆已满" << endl;   return 0; }
    heap[CurrentSize] = x;               //插在表尾
   FilterUp (CurrentSize);              //向上调整为堆
    CurrentSize++;                        //堆元素增一
    return 1;
}
template <class Type> void MinHeap<Type> ::    //最小堆的向上调整
FilterUp ( int start ) {
//从 start 开始,向上直到 0,调整堆
    int j = start,   i = (j-1)/2;      // i 是 j 的双亲
    Type temp = heap[j];
    while ( j > 0 ) {
        if ( heap[i].key <= temp.key ) break;
      else {   heap[j] = heap[i];   j = i;   i = (i -1)/2; }
    }
    heap[j] = temp;
}
template <class Type> int MinHeap <Type> ::   //删除最小堆根元素（即最小元素）
RemoveMin ( Type &x ) {
    if ( !CurrentSize )
      { cout << " 堆已空 " << endl;   return 0; }
    x = heap[0];                    //最小元素出队列
    heap[0] = heap[CurrentSize-1];
    CurrentSize--;                  //用最小元素填补
    FilterDown ( 0, CurrentSize-1 );   //从 0 号位置开始自顶向下调整为堆
    return 1;    }
```

ADT of HuffmanTree 哈夫曼树

```
template <class Type>
void HuffmanTree (Type *fr,   int n, ExtBinTree <Type> & newtree ) {
     ExtBinTree <Type> & first, & second;
     ExtBinTree <Type> Node[DafualtSize];
     MinHeap < ExtBinTree <Type> > hp;   //最小堆
     if ( n > DefaultSize ) {
          cout << "大小 n" << n << "超出了数组边界 "
                    << endl;     return;
     }
     for ( int i = 0; i < n; i++ ) {
          Node[i].root→data.key = fr[i];
           Node[i].root→leftChild =
                         Node[i].root→rightChild = NULL;
     }        //传送初始权值
     hp.MinHeap ( Node, n );
     for ( int i = 0; i < n-1; i++ ) {
      //建立霍夫曼树的过程，做 n-1 趟
           hp.RemoveMin ( first );         //选根权值最小的树
          hp.RemoveMin ( second );   //选根权值次小的树
          newtree = new ExtBinTree <Type>
                               ( first, second );      //建新的根结点
          hp.Insert ( newtree );          //形成新树插入
     }
}
```

折半查找 Binary search

```
template <class Type> int   orderedList <Type>::
BinarySearch ( const Type & x ) const {
//折半搜索的迭代算法
     int high =  ____CurrentSize-1_,   low = 0,   mid;
     while (  _______low <= high______) {
          mid = ( low + high ) / 2;
          if ( Element[mid].getKey ( ) < x )
                    _________low = mid + 1_______;
          else if ( Element[mid].getKey ( ) > x )
                         ___high = mid - 1_______;
          else return mid;
     }
     return -1;
}
```

Binary Search Tree　　二叉搜索树

```
template <class Type>         //二叉搜索树的递归的搜索算法
BstNode<Type> * BST<Type> :: Find
      (const Type & x, BstNode<Type> * ptr ) const {
  if ( ptr == NULL ) return NULL;           //搜索失败
  else if ( x < ptr→data )              //在左子树递归搜索
                        return Find ( x,  _ptr→leftChild_ );
else if ( x > ptr→data ) //在右子树递归搜索
                        return Find ( x,  __ptr→rightChild__ );
else return ptr;             //相等,搜索成功
}
template <class Type>     //二叉搜索树的迭代的搜索算法
BstNode <Type> * BST <Type> :: Find
      (const Type & x,    BstNode<Type> * ptr ) const {
      if ( ptr != NULL ) {
            BstNode<Type> * temp = ptr;
            while ( temp != NULL ) {
            if ( temp→data == x ) return temp;    //成功
            if ( temp→data < x )
                        temp = temp→rightChild;      //右子树
                  else temp = temp→leftChild;     //左子树
            }
      }
      return NULL;                       //搜索失败
}
template <class Type> void BST<Type>::      //二叉搜索树插入算法
Insert (const Type & x, BstNode<Type> * & ptr) {
          if ( ptr == NULL )       {                  //空二叉树
          ptr = new BstNode<Type> (x);         //创建含  x  结点
          if ( ptr == NULL )
               { cout << "Out of space" << endl;    exit (1); }
      }
      else if ( x < ptr→data )                  //在左子树插入
                       Insert ( x, ptr→leftChild );
      else if ( x > ptr→data )                //在右子树插入
                       Insert ( x, ptr→rightChild );
}
template <class Type> void BST<Type> ::    //二叉树删除算法
Remove (const Type &x, BstNode<Type> * &ptr) {
      BstNode<Type> * temp;
      if ( ptr != NULL )
          if ( x < ptr→data ) Remove ( x, ptr→leftChild );
          else if ( x > ptr→data )
                         Remove ( x, ptr→rightChild );
```

```
        else if ( ptr→leftChild != NULL &&    ptr→rightChild != NULL )
         {      temp = Min ( ptr→rightChild );
             ptr→data = temp→data;
             Remove ( ptr→data, temp );    }
      else {
         temp = ptr;
         if ( ptr→leftChild == NULL ) //只有左子树
                   ptr = ptr→rightChild;
         else if ( ptr→rightChild == NULL )//  只有右子树
                   ptr = ptr→leftChild;
         delete temp;
          }
  }
```

先序遍历 Preorder traversal 第六章 树

```
template <class Type> void BinaryTree<Type>::
    PreOrder ( BinTreeNode <Type> *current ) {
    if ( current != NULL ) {
       cout << current→data;
       PreOrder ( current→leftChild );
       PreOrder ( current→rightChild );       }         }
```

中序遍历 Inorder traversal

```
template <class Type>
void BinaryTree <Type>::InOrder ( ) {
    InOrder ( root );
}
template <class Type> void BinaryTree <Type>::
    InOrder ( BinTreeNode <Type> *current ) {
        if ( current != NULL ) {
           InOrder ( current→leftChild );
           cout << current→data;
           InOrder ( current→rightChild );       }       }
```

后序遍历 Postorder traversal

```
template <class Type> void
    BinaryTree <Type>::PostOrder ( ) {
    PostOrder ( root );
}
template <class Type> void BinaryTree<Type>::
PostOrder ( BinTreeNode <Type> *current ) {
    if ( current != NULL ) {
       PostOrder ( current→leftChild );
       PostOrder ( current→rightChild );
       cout << current→data;
    }   }
```

DFS　Depth-First Traversals 深度优先遍历

```
template<class NameType, class DistType>
void Graph <NameType, DistType> :: DFS ( ) {
    int * visited = new int [NumVertices];
    for ( int i = 0; i < NumVertices; i++ )
        visited [i] = 0;   //访问标记数组 visited 初始化
    DFS (0, visited );
    delete [ ] visited;          //释放 visited
}
template<class NameType, class DistType>
void Graph<NameType, DistType> ::
DFS ( const int v, int visited [ ] ) {
    cout << GetValue (v) << ‘ ’;   //访问顶点 v
    visited[v] = 1;                //顶点 v 作访问标记
    int w = GetFirstNeighbor (v);
      //取 v 的第一个邻接顶点 w
    while ( w != -1 ) {            //若邻接顶点 w 存在
        if ( !visited[w] ) DFS ( w, visited );
        //若顶点 w 未访问过, 递归访问顶点 w
        w = GetNextNeighbor ( v, w );     //取顶点 v 的排在 w 后面的下一个邻接顶点
      }     }
```

BFS　readth-First Traversals 度优先遍历

```
template<class NameType, class DistType>
void Graph <NameType, DistType> ::
BFS ( int v ) {
    int * visited = new int[NumVertices];
    for ( int i = 0; i < NumVertices; i++ )
       visited[i] = 0;          //visited 初始化
    cout << GetValue (v) << ' ';    visited[v] = 1;
    Queue<int> q;
    q.EnQueue (v);   //访问 v, 进队列
     while ( !q.IsEmpty ( ) ) {                //队空搜索结束
        v = q.DeQueue ( );                  //不空, 出队列
        int w = GetFirstNeighbor (v);       //取顶点 v 的第一个邻接顶点 w
          while ( w != -1 ) {      //若邻接顶点 w 存在
       if ( !visited[w] ) {    //若该邻接顶点未访问过
           cout << GetValue (w) << ‘ ’;          //访问
           visited[w] = 1;    q.EnQueue (w);     //进队
            }
       w = GetNextNeighbor (v, w);
            //取顶点 v 的排在 w 后面的下一邻接顶点
            }     //重复检测 v 的所有邻接顶点
       }     //外层循环，判队列空否
   delete [ ] visited;     }
```

快速排序　QuickSort

```
template <class Type>
void QuickSort ( datalist<Type> &list, const int left, const int right ) {
//在待排序区间 left~right 中递归地进行快速排序
    if ( left < right) {
        int pivotpos = Partition ( list, left, right ); //划分
        QuickSort ( list, left, pivotpos-1);
         //在左子区间递归进行快速排序
        QuickSort ( list, pivotpos+1, right );
         //在左子区间递归进行快速排序
    }
}
template <class Type>
int Partition ( datalist<Type> &list, const int low,
            const int high ) {
int pivotpos = low;       //基准位置
Swap(List.Vector[low],List.Vextor[(low+high)/2]);
Element<Type> pivot = list.Vector[low];
    for ( int i = low+1; i <= high; i++ )
        if ( list.Vector[i].getKey ( ) < pivot.getKey( )
            && ++pivotpos != i   )
            Swap ( list.Vector[pivotpos], list.Vector[i] );
            //小于基准对象的交换到区间的左侧去
    Swap ( list.Vector[low], list.Vector[pivotpos] );
    return pivotpos;
}
```

排 序 方 法	比较次数		移动次数		稳定性	附加存储	
	最好	最差	最好	最差		最好	最差
直接插入排序	n	n^2	0	n^2	√	1	
折半插入排序	n $\log_2 n$		0	n^2	√	1	
起泡排序	n	n^2	0	n^2	√	1	
快速排序	n$\log_2 n$	n^2	n $\log_2 n$	n^2	×	$\log_2 n$	n^2
简单选择排序	n^2		0	n	×	1	
锦标赛排序	n $\log_2 n$		n $\log_2 n$		√	n	
堆排序	n $\log_2 n$		n $\log_2 n$		×	1	
归并排序	n $\log_2 n$		n $\log_2 n$		√	n	