选择排序

```
void SelectPass( SqList &L, int i ) {
RcdType  W；
j = i；
for ( k=i+1; k<=L.length; k++ )
    if ( L.r[k].key < L.r[j].key )  j = k ；
if ( i != j )
   { W=L.r[j]；  L.r[j] =L.r[i]；  L.r[i] = W；  }
} // SelectPass
```

选择排序

```
void SelectSort (SqList &L) {
        RcdType  W；
     for (i=1; i<L.length; ++i) {
        j = i；
        for ( k=i+1; k<=L.length; k++ )
            if ( L.r[k].key < L.r[j].key )  j =k ；
        if ( i!=j ) { W=L.r[j];L.r[j] =L.r[i];L.r[i] = W;}
     }
} // SelectSort
```

插入排序

```
void InsertPass( SqList &L, int i ) {
    L.r[0] = L.r[i];          // 复制为哨兵
    for ( j=i-1;  L.r[0].key < L.r[j].key;  --j )
       L.r[j+1] = L.r[j];         // 记录后移
    L.r[j+1] = L.r[0];          // 插入到正确位置
} // InsertPass
```

插入排序

```
void InsertSort ( SqList &L) {
      // 对顺序表L作插入排序
      for ( i=2; i<=L.length; ++i )
        if ( L.r[i].key < L.r[i-1].key )
          L.r[0] = L.r[i];                    // 复制为哨兵
          for ( j=i-1;  L.r[0].key < L.r[j].key;  --j )
            L.r[j+1] = L.r[j];                // 记录后移
          L.r[j+1] = L.r[0];                  // 插入到正确位置
        } // if
} // InsertSort
```

起泡排序

```
void BubbleSort( SqList &L ){
    RcdType  W；
      i = L.length;
      while (i >1) {
        lastExchangeIndex = 1;
        for (j = 1; j < i; j++){
          if (L.r[j+1].key < L.r[j].key) {
            W=L.r[j]；  L.r[j] =L.r[j+1]；  L.r[j+1] = W；
            lastExchangeIndex = j;
          } //if
        } //for
        i = lastExchangeIndex;
      } // while
} // BubbleSort
```

快速排序

```
int Partition ( RcdType R[], int low, int high) {
      R[0] = R[low];            // 将枢轴记录移至数组的闲置分量
      pivotkey = R[low].key;    // 枢轴记录关键字
      while (low<high) {        // 从表的两端交替地向中间扫描
        while(low<high&& R[high].key>=pivotkey)
        --high;
        R[low++] = R[high];     // 将比枢轴记录小的记录移到低端
        while (low<high && R[low].key<=pivotkey)
        ++low;
        R[high--] = R[low];     // 将比枢轴记录大的记录移到高端
      } //while
      R[low] = R[0];            // 枢轴记录移到正确位置
      return low;               // 返回枢轴位置
} // Partition
```

快速排序

```
void QSort (RedType R[],  int s,  int t ) {
      // 对记录序列R[s..t]进行快速排序
      if (s < t-1) {                  // 长度大于1
        pivotloc = Partition(R, s, t);// 对 R[s..t]
        QSort(R, s, pivotloc-1);      // 对低子序列递归排序
        QSort(R, pivotloc+1, t);      // 对高子序列递归排序
      } // if
} // Qsort

void QuickSort( SqList & L) {
      QSort(L.r, 1, L.length);
} // QuickSort
```

归并排序

```
void Merge (RcdType SR[], RcdType TR[], int i, int m, int n)
{
    // 将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n]
    for (j=m+1, k=i;  i<=m && j<=n;  ++k) {
      if (SR[i].key<=SR[j].key)  TR[k] = SR[i++];
      else TR[k] = SR[j++];
    }
    while (i<=m) TR[k++] = SR[i++];
    while (j<=n) TR[k++] = SR[j++];
} // Merge
```

归并排序

```
void Msort ( RcdType SR[], RcdType TR1[], int s, int t ) {
    // 对SR[s..t]进行归并排序，排序后的记录存入TR1[s..t]。
    RcdType TR2[t-s+1];
    if (s==t)  TR1[s] = SR[s];
    else {
      m = (s+t)/2;
      Msort (SR, TR2, s, m);
      Msort (SR, TR2, m+1, t);
      Merge (TR2, TR1, s, m, t);
    } // else
} // MSort
void MergeSort (SqList &L) {
     MSort(L.r, L.r, 1, L.length);
} // MergeSort
```

二路归并

```
void mergepass(RcdType SR[],RcdType TR[],int s,int n)
{
        i=1;
        while(i+2s-1<=n) {
                merge(SR,TR,i,i+s-1,i+2*s-1);
                i +=2*s;
        }
        if(i+s-1<n) //最后还有两个集合，但不等长
                merge(SR,TR,i,i+s-1,n);
        else //最后还有一个集合
                while(i<＝n){TR[i]=SR[i];i++};
}
```

二路归并

```
void mergesort((Sqlist &L)
{
        RcdType TR[L.length+1];
        s=1;
        while（s<n）{
                mergepass(L.r,TR,s,n);
                s*=2;
                mergepass(TR,L.r,s,n);
                s*=2;
        }
}
```

堆排序（一）

```
void HeapAdjust (HeapType &H, int s, int m) {
  /*已知H.r[s..m]中记录的关键字除H.r[s].key之外均满足堆的定义，本函数依据关键字的大小对H.r[s]进行调整，使H.r[s..m]成为一个大顶堆(对其中记录的关键字而言)*/
    rc = H.r[s];                                  // 暂存根结点的记录
    for ( j=2*s; j<=m; j*=2 ) {     // 沿key较大的孩子向下筛选
      if ( j<m && H.r[j].key<H.r[j+1].key )  ++j;  // j为key较大孩子记录的下标
      if ( rc.key >= H.r[j].key )  break; // 不需要调整
      H.r[s] = H.r[j];  s = j;    // 把大关键字记录往上调
    }
    H.r[s] = rc;                  // 回移筛选下来的记录
} // HeapAdjust
```

堆排序（二）

```
void HeapSort ( HeapType &H ) {
    // 对顺序表H进行堆排序。
    for ( i=H.length/2; i>0; --i )  // 把H.r[1..H.length]建成大顶堆
        HeapAdjust ( H, i, H.length );
    w=H.r[1] ; H.r[1]= H.r[H.length]; H.r[H.length]=w;
                                        //交换"堆顶"和"堆底"的记录
    for ( i=H.length-1; i>1; --i ) {
      HeapAdjust(H, 1, i);   // 从根开始调整，将H.r[1..i] 重新调整为大顶堆
      w=H.r[1]; H.r[1]=H.r[i]; H.r[i]=w;    // 使已有序的记录堆积到底部
    }
} // HeapSort
```

基数排序

```
void RadixPass( RcdType A[], RcdType B[], int n, int i ) {
  for ( j=0; j<RADIX; ++j ) count[j] = 0;
  for ( k=0; k<n; ++k ) count[ A[k].keys[i] ] ++;
  for ( j=1; j<RADIX; ++j ) count[j] = count[j-1] + count[j];
  for ( k=n-1; k>=0; --k ) {
     j = A[k].keys[i];
     B[ count[j]-1 ] = A[k];
     count[j]--;
   }// for
}// RadixPass
```

基数排序

```
void RadixSort( SqList &L ) {
  RcdType C[L.length];
  i= bitsnum-1;
  while ( i >= 0 ) {
    RadixPass( L.r, C, L.length, i );     i--;
    if (i >=0 ) {
      RadixPass( C, L.r, L.length, i );    i--;
    }
    else
     for ( j=0;  j<l.length; ++j ) L.r[j] = C[j];
  }// while
}// RadixSort
```

二叉排序树排序

```
void BSTSort( SqTable &L ){
BiTree  T = NULL;       // 初始化二叉排序树为空树
for ( i=1;  i<L.length;  ++i)
    Insert_BST( T, L.r[i] ); // 按顺序表L构造二叉排序树
    i = 0;
    InOrder( T，Output(T, L, i) ); // 中序遍历二叉排序树
} // BSTSort
void Output ( BiTree T,  SqTable &L,  int & i ){
    L.r[++i]=T->data;
}
```