数据结构与算法常见代码

李明哲

计算学部金牌讲师团

```
// QuickSort
int Partition(ElemType A[], int low, int high)
{
    ElemType pivot = A[low];
    while (low < high) {
        while (low < high && A[high] >= pivot) {
            high--;
        }
        A[low] = A[high];
        while (low < high && A[low] <= pivot) {
            low++;
        }
        A[high] = A[low];
    }
    A[low] = pivot;
    return low;
}
void QuickSort(ElemType A[], int low, int high)
{
    if (low < high) {
        int pivotpos = Partition(A, low, high);
        QuickSort(A, low, pivotpos - 1);
        QuickSort(A, pivotpos + 1, high);
    }
    return;
}

// QuickSort
void quicksort(int l, int r)
{
    if (l < r) {
        int x = a[(l + r) >> 1];
        int i = l - 1, j = r + 1;
        while (i < j) {
            do {
                i++;
            } while (a[i] < x);
            do {
                j--;
            } while (a[j] > x);
            if (i < j) {
                swap(a[i], a[j]);
            }
        }
        quicksort(l, j);
        quicksort(j + 1, r);
    }
    return;
}

// MergeSort
int a[maxn];
int b[maxn];
void mergesort(int l, int r)
{
    if (l < r) {
        int mid = (l + r) >> 1;
        mergesort(l, mid);
        mergesort(mid + 1, r);
        int pl = l;
        int pr = mid + 1;
        int ptr = l;
        while (pl <= mid || pr <= r) {
            if (pl <= mid && (a[pl] < a[pr] || pr > r)) {
                b[ptr++] = a[pl];
                pl++;
            }
            else {
                b[ptr++] = a[pr];
                pr++;
            }
        }
        for (int i = l; i <= r; i++) {
            a[i] = b[i];
        }
    }
    return;
}

// UFSets
#define SIZE 100
int fa[SIZE];
void Initial(int fa[])
{
    for (int i = 0; i < SIZE; i++) {
        fa[i] = -1;
    }
    return;
}
int Find(int fa[], int x)
{
    while (fa[x] >= 0) {
        x = fa[x];
    }
    return x;
}
void Union(int fa[], int x, int y)
{
    if (x != y) {
        fa[y] = x;
    }
    return;
}
```

```c
// PreInPostOrder
void PreInPostOrder(BiTree T)
{
    if (T != NULL) {
        //visit(T);//PreOrder
        PreInPostOrder(T->lchild);
        //visit(T);//InOrder
        PreInPostOrder(T->rchild);
        //visit(T);//PostOrder
    }
}

// PreInOrder
void PreInOrder(BiTree T)
{
    InitStack(S);
    BiTree p = T;
    while (p || IsEmpty(S)) {
        if (p) {
            //visit(p);//PreOrder
            Push(S, p);
            p = p->lchild;
        }
        else {
            Pop(S, p);
            //visit(p);InOrder
            p = p->rchild;
        }
    }
    return;
}

// PostOrder
void PostOrder(BiTree T)
{
    InitStack(S);
    BiTNode *p = T;
    BiTNode *r = NULL;
    while (p || !IsEmpty(S)) {
        if (p) {
            Push(S, p);
            p = p->lchild;
        }
        else {
            GetTop(S, p);
            if (p->rchild && p->rchild != r) {
                p = p->rchild;
            }
            else {
                Pop(S, p);
                visit(p);
                r = p;
                p = NULL;
            }
        }
    }
    return;
}

// Binary_Search
int Binary_Search(SSTable L, ElemType key)
{
    int low = 0, high = L.TableLen - 1;
    while (low < high) {
        int mid = (low + high) / 2;
        if (L.elem[mid] == key) {
            return mid;
        }
        else if (L.elem[mid] > key) {
            high = mid - 1;
        }
        else {
            high = mid + 1;
        }
    }
    return -1;
}

// MaxHeap
void BuildMaxHeap(ElemType A[]. int len)
{
    for (int i = len / 2; i >= 1; i--) {
        HeadAdjust(A, i, len);
    }
    return;
}
void HeadAdjust(ElemType A[], int k, int len)
{
    A[0] = A[k];
    for (int i = 2 * k; i <= len; i *= 2) {
        if (i <= len - 1 && A[i] < A[i + 1]) {
            i++;
        }
        if (A[0] >= A[i]) {
            break;
        }
        else {
            A[k] = A[i];
            k = i;
        }
    }
    A[k] = A[0];
    return;
}
void HeapSort(ElemType A[], int len)
{
    BuildMaxHeap(A, len);
    for (int i = len; i > 1; i--) {
        Swap(A[i], A[1]);
        HeadAdjust(A, 1, i - 1);
    }
    return;
```

```
}

// Kruskal
int Kruskal(Graph a[])
{
    sort(a + 1, a + 1 + m);
    int sum = 0;
    for(int i = 1; i <= m; i++){
        int fx = find(a[i].x);
        int fy = find(a[i].y);
        if(fx != fy){
            f[fx] = fy;
            sum += a[i].z;
        }
    }
    return sum;
}

// Dijkstra
void dijkstra(void)
{
    memset(dist, 0x3f, sizeof(dist));
    dist[1] = 0;
    for (int i = 1; i <= n; i ++) {
        int t = -1;
        for (int j = 1; j <= n; j++) {
            if (!selected[j] && (dist[j] < dist[t] || t == -1)) {
                t = j;
            }
        }
        selected[t] = 1;
        for (int j = 1; j <= n; j++) {
            dist[j] = min(dist[j], dist[t] + g[t][j]);
        }
    }
    return;
}

// Floyd
void floyd(int n)
{
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            if (i != k && fdis[i][k] != 0x7f7f7f7f) {
                for (int j = 1; j <= n; j++) {
                    if (fdis[i][k] + fdis[k][j] < fdis[i][j]) {
                        fdis[i][j] = fdis[i][k] + fdis[k][j];
                    }
                }
            }
        }
    }
    return;
}
```