Imperative Programming (Sheet #3)

Marius Gavrilescu

8.3 For identical elements the quicksort will be $O(n^2)$.

```
PROCEDURE swap(VAR a, b : INTEGER);
VAR t : INTEGER;
BEGIN
  t := a;
  a := b;
  b := t;
END swap;

PROCEDURE partition(l, r : INTEGER) : INTEGER;
VAR x : T;
    i, k : INTEGER;
BEGIN
  x := a[l];
  k := l + 1;
  FOR i := l + 1 TO r - 1 DO
    IF less(a[i], x) THEN swap(a[k], a[i]); k := k + 1; END;
  END;
  swap(a[k-1], a[l]);
  FOR i := k TO r - 1 DO
    IF a[i] = x THEN swap(a[k], a[i]); k := k + 1; END;
  END;
  RETURN k - 1;
END partition;
```

9.1
```
PROCEDURE add(VAR a, b, c : MATRIX; n, m : INTEGER);
VAR i, j : INTEGER;
BEGIN
  FOR i := 0 TO n - 1 DO
    FOR j := 0 TO m - 1 DO
      c[i,j] := a[i,j] + b[i,j];
    END;
  END;
END add;
```

Yes, it can implement $A := A + B$ by calling $add(a, b, a)$.

9.2
```
PROCEDURE rotate(VAR x, y : MATRIX; n, m : INTEGER);
  VAR i, j : INTEGER;
  BEGIN
    FOR i := 0 TO n - 1 DO
      FOR j := 0 TO m - 1 DO
        x[i,j] = y[i,j];
      END
    END;

    FOR i := 0 TO n - 1 DO
      FOR j := i + 1 TO m - 1 DO
        swap(x[i,j], x[j,i])
      END
    END;

    FOR i := 0 TO (n - 1) DIV 2 DO
      FOR j := 0 TO m - 1 DO
        swap(x[i,j], x[n - i - 1, j])
      END
    END;
  END rotate;
```

9.3
```
PROCEDURE multiply(VAR a, b, c : MATRIX; n, m, p : INTEGER);
  VAR i, j, k : INTEGER;
  BEGIN
    FOR i := 0 TO n - 1 DO
      FOR j := 0 TO p - 1 DO
        c[i,j] := 0;
        FOR k := 0 TO m - 1 DO
          c[i,j] := c[i,j] + a[i,k] * b[k,j];
        END
      END
    END
  END multiply;
```

9.5
```
VAR counter : INTEGER;

PROCEDURE Increment(x : T);
BEGIN
  counter := counter + 1;
END Increment;

PROCEDURE length(xs : List) : INTEGER;
BEGIN
  counter := 0;
  return IterList(Increment, xs);
END length;

VAR tempList : List;

PROCEDURE PushFront(x : T);
VAR l;
BEGIN
  NEW(l);
  l.head := x;
  l.tail := tempList;
  tempList := l;
END

PROCEDURE reverse(xs : List) : List;
BEGIN
  tempList := NIL;
  IterList(PushFront, xs);
  return tempList;
END reverse;
```

11.1
```
PROCEDURE Delete*(k : KEY);
VAR i : INTEGER;
BEGIN
  i := FIND(k);
  IF i = size THEN RETURN END;
  pair[i].key := pair[size-1].key;
  pair[i].value := pair[size-1].value;
  size := size - 1
END Delete;
```

11.2
```
PROCEDURE Find(k : KEY): INTEGER;
  VAR s, e : INTEGER;
BEGIN
  s := 0;
  e := size - 1;
  WHILE s < e DO
    m := (s + e) DIV 2;
    IF pair[m].key < k THEN s := m + 1 ELSE e := m END;
  END;
  IF pair[s].key = m THEN RETURN s END;
  RETURN size
END Find;

PROCEDURE Store*(k : KEY; v : VALUE);
  VAR i : INTEGER;
BEGIN
  i := Find(k);
  IF i = size THEN
    ASSERT(size < MAXKEY); size := size + 1; pair[i].key := k;
    WHILE (i > 0) & (pair[i-1].key > pair[i].key) DO
      swap(pair[i-1], pair[i]);
      i := i - 1;
    END
  END;
  pair[i].value := v
END Store;

PROCEDURE Delete*(k : KEY);
BEGIN
  i := FIND(k);
  IF i = size THEN RETURN END;
  WHILE i < size DO
    pair[i].key := pair[i+1].key;
    pair[i].value := pair[i+1].value;
    i := i + 1;
  END;
  size := size - 1
END Delete;
```

11.3
```
PROCEDURE Delete*(k : KEY);
VAR p, q : List;
BEGIN
  p := Find(k);
  IF p = NIL THEN RETURN END;
  IF p = list THEN list := list.next; RETURN END;
  q := list;
  WHILE q.next # p DO q := q.next END;
  q.next := p.next;
END Delete;
```