Haggis for Advanced Higher
Algorithms and Data Structures with Object Orientation
Greg Michaelson
May 2015

Contents

0. Introduction....2
1. Linear array....2
1.1. Linear Search....2
1.2. Binary search – ascending order - iterative....2
1.3. Binary search – ascending order – recursive....3
1.4. Swap....3
1.5. Bubble sort – ascending order....4
1.6. Bubble sort – ascending order - with success check....4
1.7. Quicksort – ascending order....4
1.8. Insert – ascending order....5
1.9. Delete – ascending order....5
2. Stack....6
2.1. Push....6
2.2. Pop....6
3. Queue....7
3.1. Join....7
3.2. Leave....7
4. Linked list – iterative/update....8
4.1. Show....8
4.2. Insert – ascending order....8
4.3. Delete- ascending order....9
5. Linked list – recursive/copy....9
5.1. Show....9
5.2. Insert – ascending order....10
5.3. Delete – ascending order....10
5.4. Sort – ascending order....10
6. Exercises....10
6.1. Linear array....10
6.2. Stack....11
6.3. Queue....11
6.4. Linked list – iterative/update....12

6.5. Linked list – recursive/copy.....12

0. Introduction

The following algorithms and data structures exemplify many of those specified for study in the Scottish Qualifications Authority Advanced Higher in Computing Science.

The notation used is Haggis, a neutral reference language intended originally for setting assessments at SQA National 5, Higher and Advanced Higher.

I'd like to thank:
- Ian King, Richard Connor and Quintin Cutts;
- the participants in the Haggis Workshop at the University of Strathclyde on 11th May 2015;

for their help.

If you find any egregious errors, please do tell me!

Greg Michaelson
School of Mathematical and Computer Sciences
Heriot-Watt University

G.Michaelson@hw.ac.uk

1. Linear array

1.1. Linear Search

```
FUNCTION linearSearch(ARRAY OF INTEGER a,
                      INTEGER length,
                      INTEGER v) RETURNS INTEGER
   DECLARE I INITIALLY 0
   DECLARE found INITIALLY FALSE
   WHILE NOT found AND i<length DO
      IF v=a[i] THEN
         SET found TO TRUE
      ELSE
         SET i TO i+1
      END IF
   END WHILE
   RETURN i
END FUNCTION
```

1.2. Binary search – ascending order - iterative

```
FUNCTION binarySearch(ARRAY OF INTEGER a,
                      INTEGER length,
                      INTEGER v) RETURNS INTEGER
   DECLARE left INITIALLY 0
   DECLARE right INITIALLY length-1
   DECLARE middle INITIALLY 0
   DECLARE result INITIALLY -1
   DECLARE found INITIALLY FALSE
   WHILE NOT found AND left <= right DO
      SET middle TO (left+right)/2
      IF a[middle]=v THEN
         SET result TO middle
         SET found TO TRUE
      ELSE
         IF a[middle]>v THEN
            SET right TO middle-1
         ELSE
            SET left TO middle+1
         END IF
      END IF
   END WHILE
   RETURN result
END FUNCTION
```

1.3. Binary search – ascending order – recursive

```
FUNCTION recBinarySearch(ARRAY OF INTEGER a,
                         INTEGER left,
                         INTEGER right,
                         INTEGER v) RETURNS INTEGER
   DECLARE middle INITIALLY (left+right)/2
   IF left>right THEN
      <not found action>
   ELSE
      IF a[middle]=v THEN
         RETURN middle
      ELSE
         IF a[middle]<v THEN
            RETURN recBinarySearch(a,left,middle,v)
         ELSE
            RETURN recBinarySearch(a,middle+1,right,v)
         END IF
      END IF
   END IF
END FUNCTION
```

1.4. Swap

```
PROCEDURE swap(ARRAY OF INTEGER a,
```

```
                INTEGER i,
                INTEGER j)
    DECLARE temp INITIALLY a[i]
    SET a[i] TO a[j]
    SET a[j] TO temp
 END PROCEDURE
```

1.5. Bubble sort – ascending order

```
PROCEDURE bubbleSort(ARRAY OF INTEGER a,
                     INTEGER length)
   FOR i FROM length-2 TO 0 STEP -1 DO
      FOR j FROM 0 TO i DO
         IF a[j]>a[j+1] THEN
            swap(a,j,j+1)
         END IF
      END FOR
   END FOR
END PROCEDURE
```

1.6. Bubble sort – ascending order - with success check

```
PROCEDURE fastBubbleSort(ARRAY OF INTEGER a,
                         INTEGER length)
   DECLARE swaps INITIALLY true
   DECLARE i INITIALLY length-2
   WHILE swaps AND i>=0 DO
      SET swaps TO false
      FOR j FROM 0 TO i DO
         IF a[j]> a[j+1] THEN
            swap(a,j,j+1)
            SET swaps TO true
         END IF
      END FOR
      SET i TO i-1
   END WHILE
END PROCEDURE
```

1.7. Quicksort – ascending order

```
PROCEDURE quickSort(ARRAY OF INTEGER a,
                    INTEGER left,
                    INTEGER right)
   IF left<right THEN
      DECLARE middle INITIALLY partition(a,left,right)
      quickSort(a,left,middle)
      quickSort(a,middle+1,right)
```

```
   END IF
END PROCEDURE

FUNCTION partition(ARRAY OF INTEGER a,
                   INTEGER left,
                   INTEGER right) RETURNS INTEGER
   DECLARE l INITIALLY left
   DECLARE r INITIALLY right
   DECLARE pivot INITIALLY a[l]
   WHILE l<r DO
      WHILE a[l]<pivot DO
         SET l TO l+1
      END WHILE
      WHILE a[r]>pivot DO
         SET r TO r-1
      END WHILE
      IF l<r THEN
         swap(a,l,r)
         SET l TO l+1
         SET r TO r-1
      END IF
   END WHILE
   RETURN l
END FUNCTION
```

1.8. Insert – ascending order

```
PROCEDURE insert(ARRAY OF INTEGER a,
                 INTEGER next,
                 INTEGER length,
                 INTEGER v)
   DECLARE i INITIALLY 0
   IF next=length THEN
      <array full action>
   ELSE
      WHILE i<next AND v>a[i] DO
         SET i TO i+1
      END WHILE
      FOR j FROM next TO i+1 STEP -1 DO
         SET a[j] TO a[j-1]
      END FOR
      SET a[i] TO v
      SET next TO next+1
   END IF
END PROCEDURE
```

1.9. Delete – ascending order

```
PROCEDURE delete(ARRAY OF INTEGER a,
```

```
                  INTEGER next,
                  INTEGER v)
   DECLARE i INITIALLY 0
   DECLARE found INITIALLY false
   WHILE NOT found AND i<next DO
      IF v=a[i] THEN
         SET found TO true
      ELSE
         SET i TO i+1
      END IF
   END WHILE
   IF found THEN
      FOR j FROM i TO next-2 DO
         SET a[j] TO a[j+1]
      END FOR
      SET next TO next-1
   ELSE
      <not found action>
   END IF
END PROCEDURE
```

2. Stack

```
CLASS stack IS {ARRAY OF INTEGER s,
               INTEGER sp,
               INTEGER size}
METHODS

CONSTRUCTOR (INTEGER sz)
   DECLARE THIS.size INITIALLY size
   DECLARE THIS.s INITIALLY [0]*size
   DECLARE THIS.sp INITIALLY 0
END CONSTRUCTOR
```

2.1. Push

```
PROCEDURE push(INTEGER V)
   IF THIS.sp=THIS.size THEN
      <stack overflow action>
   ELSE
      SET THIS.s[THIS.sp] TO v
      SET THIS.sp TO THIS.sp+1
   END IF
END PROCEDURE
```

2.2. Pop

```
FUNCTION pop() RETURNS INTEGER
```

```
   IF THIS.sp=0 THEN
      <stack underflow action>
   ELSE
      SET THIS.sp TO THIS.sp-1
      RETURN THIS.s[THIS.sp]
   END IF
END FUNCTION

END CLASS
```

3. Queue

```
CLASS queue IS {ARRAY OF INTEGER q,
                INTEGER qp,
                INTEGER size}
METHODS

CONSTRUCTOR (INTEGER sz)
   DECLARE size INITIALLY sz
   DECLARE q INITIALLY []*size
   DECLARE qp INITIALLY 0

END CONSTRUCTOR
```

3.1. Join

```
PROCEDURE join(INTEGER v)
   IF THIS.qp=THIS.size THEN
      <queue overflow action>
   ELSE
      SET THIS.q[THIS.qp] TO v
      SET THIS.qp TO THIS.qp+1
   END IF
END PROCEDURE
```

3.2. Leave

```
FUNCTION leave() RETURNS INTEGER
   DECLARE result INITIALLY <whatever>
   IF THIS.qp=0 THEN
      <queue underflow action>
   ELSE
      SET result TO THIS.q[0]
      FOR i FROM 0 TO THIS.qp-2 DO
         SET THIS.q[i] TO THIS.q[i+1]
      END FOR
      SET THIS.qp TO THIS.qp-1
      RETURN result
```

```
   END IF
END FUNCTION

END CLASS
```

4. Linked list – iterative/update

```
RECORD cell IS {INTEGER value,
                cell next}

CLASS list IS {cell first}
METHODS
```

4.1. Show

```
PROCEDURE show()
   DECLARE f INITIALLY THIS.first
   WHILE f!=[] DO
      SEND f.value TO DISPLAY
      SET f TO f.next
   END WHILE
END PROCEDURE
```

4.2. Insert – ascending order

```
PROCEDURE insert(INTEGER v)
   IF THIS.first=[] THEN
      SET THIS.first TO cell(v,[])
   ELSE
      IF v<THIS.first.value THEN
         SET THIS.first TO cell(v,THIS.first)
      ELSE
         DECLARE f INITIALLY THIS.first
         DECLARE done INITIALLY false
         WHILE NOT done DO
            IF f.next=[] THEN
               SET f.next TO cell(v,[])
               SET done TO true
            ELSE
               IF v<f.next.value THEN
                  SET f.next TO cell(v,f.next)
                  SET done TO true
               ELSE
                  SET f TO f.next
               END IF
            END IF
         END WHILE
      END IF
```

```
   END IF
END PROCEDURE
```

4.3. Delete- ascending order

```
PROCEDURE delete(INTEGER v)
   IF first=[] THEN
      <not found action>
   ELSE
      IF THIS.first.value=v THEN
         SET THIS.first TO THIS.first.next
      ELSE
         DECLARE f INITIALLY THIS.first
         DECLARE done INITIALLY false
         WHILE NOT done DO
            IF f.next=[] THEN
               <not found action>
            ELSE
               IF f.next.value=v THEN
                  SET f.next TO f.next.next
                  SET done TO true
               ELSE
                  SET f TO f.next
               END IF
            END IF
         END WHILE
      END IF
   END IF
END PROCEDURE

END CLASS
```

5. Linked list – recursive/copy

```
CLASS recList WITH {INTEGER value, recList next}
METHODS
```

5.1. Show

```
PROCEDURE show()
   IF THIS!=[] THEN
      SEND THIS.value TO DISPLAY
      THIS.next.show()
   END IF
END PROCEDURE
```

5.2. Insert – ascending order

```
FUNCTION insert(INTEGER v) RETURNS recList
   IF THIS=[] THEN
      RETURN recList(v,[])
   ELSE
      IF v<THIS.value THEN
         RETURN recList(v,THIS)
      ELSE
         RETURN recList(THIS.value,THIS.next.insert(v))
      END IF
   END IF
END FUNCTION
```

5.3. Delete – ascending order

```
FUNCTION delete(INTEGER v) RETURNS recList
   IF THIS=[] THEN
      <not found action>
   ELSE
      IF THIS.value=v THEN
         RETURN THIS.next
      ELSE
         RETURN recList(THIS.value,THIS.next.delete(v))
      END IF
   END IF
END FUNCTION
```

5.4. Sort – ascending order

```
FUNCTION sort() RETURNS recList
   IF THIS=[] THEN
      RETURN []
   ELSE
      RETURN (THIS.next.sort()).insert(THIS.value)
   END IF
END FUNCTION

END CLASS
```

6. Exercises

For each of the following, trace the changes to the variables and associated data structures:

6.1. Linear array

```
1. DECLARE b INITIALLY [1,2,3,4,5,6,7,8,9,10]
```

```
2. DECLARE r INITIALLY 0
3. SET r TO linearSearch(b,10,5)
4. SET r TO linearSearch(b,10,11)
5. SET r TO binarySearch(b,10,9)
6. SET r TO binarySearch(b,10,11)
7. SET r TO recBinarySearch(b,0,9,9)
8. SET r TO recBinarysearch(b,0,9,11)

9. DECLARE c INITIALLY [4,1,3,2,5]
10.  bubbleSort(c,5)
11.  fastBubbleSort(c,5)  # with original c
12.  quickSort(c,0,4)        # with original c

13.  DECLARE d INITIALLY []*4
14.  DECLARE n INITIALLY 0
15.  insert(d,n,4,4)
16.  insert(d,n,4,1)
17.  insert(d,n,4,3)
18.  insert(d,n,4,4)
19.  insert(d,n,5,5)
20.  delete(d,n,2)
21.  delete(d,n,5)
```

6.2. Stack

```
1. DECLARE ss INITIALLY stack(3)
2. ss.push(2)
3. ss.push(1)
4. ss.push(3)
5. ss.push(4)
6. SET r TO ss.pop()
7. SET r TO ss.pop()
8. SET r TO ss.pop()
9. SET r TO ss.pop()
```

6.3. Queue

```
1. DECLARE qq INITIALLY queue(3)
2. qq.join(2)
3. qq.join(1)
4. qq.join(3)
5. qq.join(4)
6. SET r TO qq.leave()
7. SET r TO qq.leave()
8. SET r TO qq.leave()
9. SET r TO qq.leave()
```

6.4. Linked list – iterative/update

```
1. DECLARE l1 INITIALLY list([])
2. l1.insert(1)
3. l1.insert(4)
4. l1.insert(2)
5. l1.insert(3)
6. l1.show()
7. l1.delete(4)
8. l1.delete(1)
9. l1.delete(5)
```

6.5. Linked list – recursive/copy

```
1. DECLARE l2 initially recList([])
2. SET l2 TO l2.insert(1)
3. SET l2 TO l2.insert(4)
4. SET l2 TO l2.insert(2)
5. SET l2 TO l2.insert(3)
6. l2.show()
7. SET l2 TO l2.delete(4)
8. SET l2 TO l2.delete(1)
9. SET l2 TO l2.delete(5)
```