Foundations of Computer Science (Semester 2) – 2015

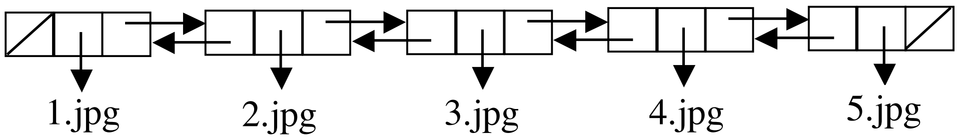
Assessed Exercise Sheet 2 – 10% of Continuous Assessment Mark

Deadline : 11pm Sunday 1st February, via Canvas

Question 1 (24 marks)

Suppose you have a series of photograph 1.jpg, 2.jpg, …, N.jpg and wish to store their locations as a doubly linked list as described in the lectures.

Produce a graphical representation for the N = 5 case.



For the general N case, state in words (not pseudocode) an efficient ordered list of operations that will insert an additional photograph x.jpg into the list at a particular given point near the middle.

1. Create a new 3-cell with data element that is the location of x.jpg.
2. Copy the rear pointer of the cell that now precedes it into the rear pointer of the new cell.
3. Change the rear pointer of the cell that now precedes it to point to the new cell.
4. Copy the front pointer of the cell that now follows it into the front pointer of the new cell.
5. Change the front pointer of the cell that now follows it to point to the new cell.

Suppose the photograph locations were instead stored in a simple array. State in words an ordered list of operations that will insert an additional photograph x.jpg into the array at a particular given point near the middle.

1. Copy the content of array position N into position N+1
2. Repeat for array positions N-1 down to the position where the new photograph is needed.
3. Insert the location of x.jpg into the required position.

Comment on the different time complexities of the doubly linked list and array versions.

The doubly linked list version has constant time complexity (5 operations). The array version has linear time complexity (~N/2 operations) and is therefore usually much slower.

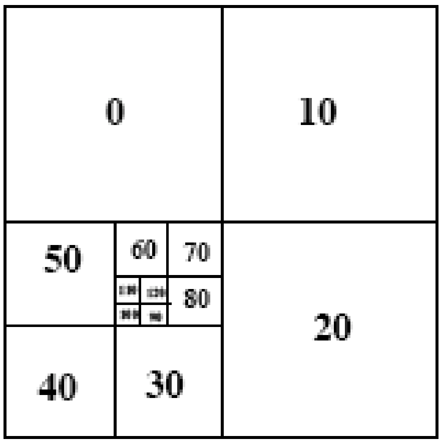
Question 2 (16 marks)

In the lecture notes (section 2.2) we looked at a procedure `last(L)` that returned the last item in the given list `L`. Modify that to create a recursive procedure `getItem(i,L)` that returns item `i` in a list `L`, where `i` is an integer greater than zero.

```
   getItem(i,L) {
      if ( isEmpty(L)) )
         error('Error: List is too short.')
      elseif  ( i == 1 )
         return first(L)
      else return getItem(i-1,rest(L))
  }
```

Question 3 (24 marks)

A *quadtree* was defined in the lectures in terms of primitive constructors `baseQT(value)` and `makeQT(luqt,ruqt,llqt,rlqt)`, selectors `lu(qt)`, `ll(qt)`, `ru(qt)` and `rl(qt)`, and condition `isValue(qt)`. Suppose a gray-scale picture is represented by such a quadtree with values in the range 0…255, for example:



Write a procedure `flip(qt)`, that uses the above primitive quadtree operators, to flip the picture about the horizontal line through its centre.

```
   flip(qt) {
      if (isValue (qt) )
         return qt
      else return makeQT(flip(ll(qt)), flip(rl(qt)),
                         flip(lu(qt)), flip(ru(qt)) )
   }
```

Write another procedure `avevalue(qt)`, that uses the above primitive quadtree operators, to return the average gray-scale value across the whole picture.

```
   avevalue(qt) {
      if (isValue(qt) )
         return qt
      else return (avevalue(lu(qt)) + avevalue(ru(qt))
                  + avevalue(ll(qt)) + avevalue(rl(qt)))/4
   }
```

Question 4 (20 marks)

Suppose you already have the primitive binary tree procedures `isempty(bt)`, `root(bt)`, `left(bt)`, and `right(bt)`. Write a procedure `isLeaf(bt)` that returns `true` if the binary tree `bt` is a leaf node, and `false` if it is not.

```
isLeaf(bt) {
   return ( not isempty(bt)
            and isempty(left(bt)) and isempty(right(bt)) )
}
```

Then write a recursive procedure `numLeaves(bt)` that returns the number of leaves in the given binary tree `bt`. It is allowed to call any of the primitive binary tree procedures and also your `isLeaf(bt)` procedure.

```
numLeaves(bt) {
   if( isempty(bt) )
       return 0
   elseif ( isLeaf(bt) )
       return 1
   else return (numLeaves(left(bt)) + numLeaves(right(bt)))
}
```

Question 5 (16 marks)

It is often important to know whether two given binary trees are the same. Write a procedure `equalBinTree(t1,t2)` which returns `true` if the given binary trees `t1` and `t2` are the same, and `false` otherwise. You can assume that you have access to the basic binary tree procedures `isempty(t)`, `root(t)`, `left(t)`, and `right(t)`. [Hint: Remember that you can only directly test the equality of numbers, e.g. node values.]

```
equalBinTree(t1,t2) {
   if ( isEmpty(t1) and isEmpty(t2) )
      return true
   elseif ( isEmpty(t1) or isEmpty(t2) )
      return false
   else return ( (root(t1) == root(t2) ) and
                  equalBinTree(left(t1),left(t2)) and
                  equalBinTree(right(t1),right(t2)) )
}
```

What is the time complexity of your algorithm?

Linear in n, or O(n), where n is the number of nodes in the smallest tree.