Family name: _____

Given Name: _____

User name: _____

York University
Faculty of Pure and Applied Science/Faculty of Arts
Department of Computer Science

COSC1030.03 Section P
Midterm Test -- Solution
1997 February 19

Instructions

1 This is a closed book test.
2 Answer each question in the space provided.
3 All questions are to be attempted.
4 The test time is 50 minutes.
5 Where descriptive answers are requested, use complete sentences and paragraphs.
6 Use diagrams to enhance your answers.
7 Comment all program text.
8 Tests written in pencil will not be considered for remarking.
9 The value of each question is indicated below. They are graded on the university standard A,B,C,... scale.

Question	Maximum	Grade
1	9	
2	9	
3	9	
4	9	
5	9	
6	9	
7	9	
Total	63	

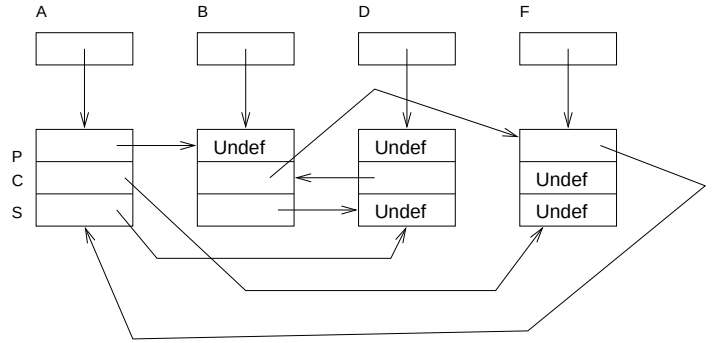
Question 1 -- Pointers

You are given the following declarations and statements.

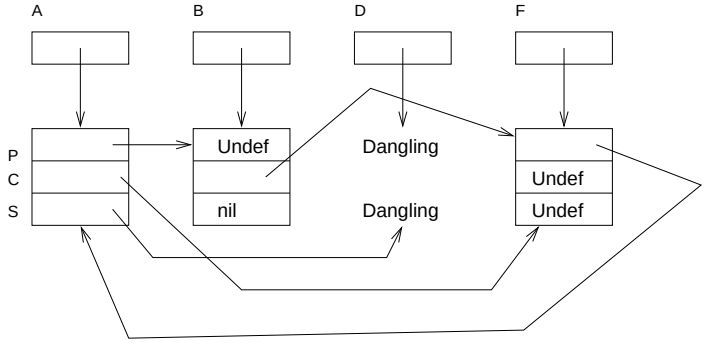
```
class PD
  export var P, var C, var S
  P : ^PD
  C : ^PD
  S : ^PD
end PD
```

```
var A, B, D, F : ^ PD
new A
new B
new D
new F
A->P := B
A->C := F
A->S := D
A->P->S := D
A->P->S->C := B
B->C := F
F->P := A
free B->S
```

Before the free statement



After the free statement



For each of the following circle the correct answer after the above program text has executed.

Field B->C->P->C has the value A B D << F >> Undefined Error

Field B->P->P->P has the value A B D F Undefined << Error>>

Field A->P->C->C has the value A B D F << Undefined >> Error

Field B->S is a dangling pointer YES << NO >> Meaningless Undecidable

Field A->P->C->P->S is a dangling pointer << YES >> NO Meaningless Undecidable

How many dangling pointers are there? 0 1 << 2 >> 3 4 5

How many nil fields are there? 0 << 1 >> 2 3 4 5

How many undefined fields are there? 0 1 2 << 3 >> 4 5

How many more free statments are needed to finalize the computed structure? 0 1 2 << 3 >> 4 5

Question 2 -- Big O

For each of the following algorithms circle the correct big O for execution time if the size of the input is P.

Algorithm 1: O(1) O(log P) O((log P)^2) O(P log P) O(P) << O(P^2) >>

Algorithm 2: O(1) O(log P) O((log P)^2) << O(P log P) >> O(P) O(P^2)

Algorithm 3: O(1) O(log P) << O((log P)^2) >> O(P log P) O(P) O(P^2)

Algorithm 1

```
i :=1
loop exit when i >= P
  j := 1
  loop exit when j >= P
    j += 2
  end loop
  i += 2
end loop
```

Algorithm 2

```
i := P
loop exit when i <= 1
  j := P
  loop exit when j <= 1
    j -= 2
  end loop
  i := i div 2
end loop
```

Algorithm 3

```
i := 1
loop exit when i >= P
  j := P
  loop exit when j <= 1
    j := j div 2
  end loop
  i := 2*i
end loop
```


Explain what is meant by the O(f(n)) of space complexity of an algorithm.

>>> n is the size of the input to the algorithm. As the size increases the amount of space the algorithm needs follows the function f(n). With big O only the highest order term is retained and multiplicative constants are replaced with the value 1.

Give an example of an algorithm with O(N) of space complexity.

>>> An algorithm to reverse a sequence of characters. All the characters must be saved before the first output character can be printed. Any algorithm where the input must be saved before output is generated.

Question 3 -- Loop invariants

What is the loop invariant for each of the following loops?

Given a list of integers in var P : array 1..Max of int. A loop that does a sequential search for value V from index J to index K inclusive, where Max ≥ J ≥ K ≥ 1.

>>> for i: J .. K-1 :: P(i+i .. J) ≠ V Note: the search is from high index to low!

This is a simplification of the TotalWork = WorkDone + WorkToDo, see the following, by eliminating the "redundant" (double entry bookkeeping) Search subsequences on each part of the equal sign and implying indirectly the key may be in the remainder of the the sequence. Searched(P(i+1..J)) is the equivalent of the above relationship.

for i: J .. K-1 :: Search(P(K..J)) = Searched(P(i+i..J)) and Search(P(K..i))

Given a list of integers in var P : array 1..Max of int. A loop that modifies P(J..K) by adding the corresponding integers in P(J+L..K+L); that is P(i) := P(i) + P(i+L).
Assume 1 ≤ J ≤ K ≤ Max and L ≥ 0.

>>> for i: 0 .. K-J+1 :: P(J .. J+(i-1))' = P(J .. J+(i-1)) + P(J+L .. J+(i-1)+L)

Note the explicit statement of the work done in terms of input and output (work done) by the loop. In the case of the search (previous case) the input is the array part being searched and the output is the relationship that the input does not contain V.

A loop to copy a file line by line.

>>> for i: 1..#lines+1 :: outputLine(1 .. i-1) = inputLine(1 .. i-1)

Note the explicit statement of the work done in terms of input and output (work done) by the loop.

Question 4 -- Proving programs correct

Prove the following algorithm to sum the integers from 1 to B inclusive is correct.

Precondition: $B \geq 1$ Postcondition: $R = \sum_{i=1}^{B} i$ Loop invariant: $R = \sum_{i=1}^{T} i$

```
R := 1
T := 1
%----------------------------- 1
loop exit when T = B
  T += 1
  R += T
%----------------------------- 2
end loop
%----------------------------- 3a
%----------------------------- 3b
```

1 -- Establish the loop invariant is correct after initialization for the loop

Substitute R and T into the loop invariant -- $1 = \sum_{i=1}^{1} i$

The sum from 1..1 is 1. So the loop invariant is true.

2 -- Establish the loop invariant it true after the loop body executes assuming the exit condition is false and the loop invariant is true. Substitute the new value for R (R' = R + T') and the new value for T (T' = T + 1) into the loop invariant.

$R' = \sum_{i=1}^{T'} i$ becomes $R + T + 1 = \sum_{i=1}^{T+1} i$

$R + T + 1 = T + 1 + \sum_{i=1}^{T} i$ Use meaning of sigma

$R = \sum_{i=1}^{T} i$ Simplify.

So loop invariant is still true. The condition is not needed in this case.

3a -- Is the postcondition satisfied. Loop invariant must still be true.

$R = \sum_{i=1}^{T} i$

$R = \sum_{i=1}^{B} i$ Substitute, T = B, from the exit condition which is true.

The postcondition is true

3b -- Does the loop exit?

$T = 1 \leq B$ by the precondition and loop initialization

T increments by 1, so must eventually reach exactly equal to B (property of counting)

The test occurs before any increment of T so the loop exits even when B = 1
The loop must exit.
□

Question 5 -- Linked lists

Given the following ListNode class and List class, complete the definition of the procedure `FreeNodes` in the List class for singly linked lists.

```
class ListNode
    import DataClass in "..."
    export var theData, var key,  var next, Finalize , Initialize
    var key     : int             %**** Assumption, Correct export list
    var theData : ^DataClass      %**** Intent is to write a program.
    var next    : ^ListNode

    procedure Initialize()
        next := nil   ;   new theData   ;   theData->Initialize()
    end Initialize

    procedure Finalize()   theData->Finalize()   end Finalize
end ListNode

class List
    var front : ^ListNode
    ... % Other program text which you do not need.
```

FreeNodes frees all nodes having the input key value, `withKey`
Precondition: `self` has been initialized
Postcondition: `listNode` in `self` ⇒ (`listNode->key` ≠ `withKey` ⇒ `listNode` in `self'`
and `listNode->key = withKey` ⇒ `free listNode`)

```
    procedure FreeNodes(withKey : int)
        var prevPtr : ^ListNode := nil    % Node previous to one being checked.
        var currPtr : ^ListNode := front  % Node being checked.
        var freePtr : ^ListNode           % Remember node to free.

        loop exit when currPtr = nil      % Keep going to end of list.
            if currPtr->key = withKey then
            % Handle the free node case.  First remove the node from the list.
                freePtr := currPtr        % Remember the node to free
                if prevPtr = nil then
                % Node to remove is at the front of the list
                    front := currPtr->next
                else % Node to remove is not at the front of the list
                    prevPtr->next := currPtr->next
                end if
                currPtr := currPtr->next  % Advance to next node to check.
                freePtr->Finalize()       % Now it is save to free the node
                free freePtr

            % Handle the keep node case.  Just advance the pointers.
            else prevPtr := currPtr
                 currPtr := currPtr->next
            end if
        end loop
    end FreeNodes
end List
```

Question 6 -- Recursive functions

Write a recursive function, using binary split and no global variables, to sum the elements of an array.

Precondition initial call to Sum: P(1..N) contains the integers to add.

Postcondition of initial call to Sum: Result = $\sum_{i=1}^{N} P(i)$

Fill in the following for the function Sum for any call, including recursive ones.

Precondition: P(A..B) contains the integers to add

Postcondition: Result = $\sum_{i=A}^{B} P(i)$

```
function Sum(P : ArrayType , A, B : int) : int
    var midIndex : int                % Index to middle of the array.
    if A = B then result P(A) end if    % Base case, only one number.
    midIndex := A + (B - A) div 2       % Find split index.
    result Sum(P, A, midIndex) + Sum(P, midIndex+1, B)   % Recursive case.
end Sum
```

Note: the computation for `midIndex` can be simplified to be `(A + B) div 2` provided the indices `A` and `B` are never negative. For example the above program works if `ArrayType` is the following.

```
type ArrayType : array -5..5 of int
```

If you substitute `midIndex := (A + B) div 2`, the program will have infinite recursion as `midIndex` will become -4 when `A = -5` and `B = -4`, instead of becoming -5 as required by the "result" statement. The reason being `div` truncates the fraction part, thus moving the result towards 0 instead of towards -infinity. For negative `A` values this is away from `A` instead of towards `A`.

Question 7 -- ADT design

Describe the objects in the design of an abstract data type.

>>>>>>>>
There are two types of obects.
-- The real objects correspond to data in which the user is interested. For example, numbers stored in a sparse matrix.

-- The meta objects contain data about the user data. For example, the row and column header lists for a sparse matrix.
<<<<<<<<

Describe guidelines for designing the operations for an abstract data type.

>>>>>>>>
The operations should be fine grained enough to implement the most atomic state changes that do not violate the ADT invariants.

The operations should be complete, minimal and orthoganal so all valid state changes can be done with no side effects among the operations to change their behaviour and there are no extra operations.

More than the minimal set of operations can be provided to simplify use of the ADT and make frequent combinations of base operations more efficient.

Operations can be divided into five classes.

-- Enquiry: Returns meta data; no change in state

-- Read: Returns user data; usually no change in state but is combined with state changes that modify metadata such as advancing a pointer to the next character in a file on a get.

-- Write: Stores user data; usually no state change on replace; state changes on adding new information.

-- Reorganize: Does not change user data but does change metadata. Operations needed to make efficient use of space or time.

-- Test: Operations to test the correctness of implementation. Useful for regression testing.
<<<<<<<<

Explain when a class should be used to implement an abstract data type and when a module should be used to implement an abstract data type.

>>> Modules are used when only one instance of an ADT is needed. For example, the debug flags in in Assignment 2 or a clock in a simulation program.

>>> Classes are used when you want more than one instance of an ADT. For example, matrix elements in a sparse matrix, or sparse matrices themselves, stacks, etc.