York University

CSE 2011 Summer 2015 – Midterm

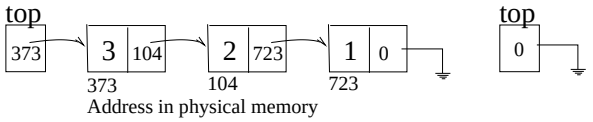
Instructor: Jeff Edmonds

0 Art therapy question: When half done the exam, draw a picture of how you are feeling.

1. (71% in 2015)
Consider a stack implemented using a singling linked list with a variable *top* pointing at the first node.

   (a) Imagine constructing a new stack and then pushing 1, 2, and then 3 onto it. Draw a picture of what the stack then looks like. Show where pointers are pointing both with arrows AND by giving actual values to the variables and showing in the drawing what these values represent.

   - Answer:



   (b) Give code for the constructor *stack*() which sets up an empty stack.

   - Answer:
     top = null;
     % Note this variable is not allocated here. This is done at the beginning of the class.
     % Java would probably automatically give *top* the value *null* when the variable was allocated. But I am doing here just to be safe and clear.

   (c) Give code for *stack.push*(*int* x) which pushes the value x onto this stack.

   - Answer: See Invariants slide pg 230.
     Node temp = new Node();
     temp.info = x;
     temp.next = top;
     top = temp;

   (d) Give code for *int stack.pop*() which pops the value x off this stack.

   - Answer: See Invariants slide pg 234.
     int x = top.info;
     top = top.next;
     return(x);
     Note Java automatically garbage collects the node when nobody is pointing at it.

   (e) What special code is needed to handle the case when the element is being pushed onto an empty stack?

   - Answer: No special code is needed. (In the slides extra code was needed to maintain the invariant for the *last* pointer.)

   (f) Give the bigO (or Theta) of the running time of this push operation as a function of the number n of nodes in the stack. Why?

   - Answer: $\Theta(1)$, because the number of operations is bounded by the constant 4, independent of the number of nodes n.

   (g) Draw (in (a)) a picture of the stack after you have popped all three of these items. Again show both arrows and actual values.

   - Answer: See above.

(h) Seeing the code, $x = 5;\ y = x;\ z = y;$, I can trace the history of the value 5. It was first assigned to the variable x. Then it was copied from x to y and then from y to z.

After you have popped the last item off the stack, what value is stored in the variable *top*?

Trace the history of this value backwards in time. In your above code, indicate where this value first came from and which lines of code copied it from one variable to the next until finally got to *top* as it is now.

- Answer: The value is the *null* pointer, which is actually the integer zero. When the stack was constructed, its precondition dictates that the stack initially be empty. In this case the variable *top* should contain the null pointer. Java automatically initializes variables to zero, but but my line $top = null$; did it to be safe. When the first item was pushed, the code $temp.next = top$; moves this null pointer to the next field of the last node in the linked list. There it stays until this last item is popped. At this point, the code $top = top.next$; returns it to *top*.

(i) Given the linked list as described above, give code *int stack.last*() that returns the value in the last node in the stack, i.e. furthest from *top*.
(Note an official stack would not have such an operation.)

- Answer:
  Node walk = top;
  while(walk.next != null) walk = walk.next;
  return(walk.info);

(j) Give the big-Oh (or Theta) of the running time of this program as a function of the number n of nodes in the stack. Why?

- Answer: $\Theta(n)$, because the code must walk down the length of the linked list visiting each of the n nodes.

2. (68% in 2015) Invariants:

(a) What is loop invariant?

- Answer: A picture of what must be true at the top of the loop

(b) Loop Invariant Steps: Your goal is to prove that no matter what the input is, as long as it meets the precondition, and no matter how many times your iterative algorithm iterates, as long as eventually the exit condition is met, then the post condition is guarantee to be achieved. According to Jeff's steps, what are three most important things to prove in order to accomplish this. Explain each in your own words. Also state each as an $A \Rightarrow B$ statement.

- Answer:
  i. Establish the Loop Invariant:
  Our computation has just begun and all we know is that we have an input instance that meets the Pre Condition.
  Being lazy, we want to do the minimum amount of work.
  And to prove that it follows that the Loop Invariant is then true.
  $\langle pre{-}cond \rangle\ \&\ code_{pre\text{-}loop}\ \Rightarrow\ \langle loop{-}invariant \rangle$
  ii. Maintain the Loop Invariant:
  We arrived at the top of the loop knowing only that the Loop Invariant is true and the Exit Condition is not.
  We must take one step (iteration) (making some kind of progress).
  And then prove that the Loop Invariant will be true when we arrive back at the top of the loop.
  $\langle loop{-}invariant' \rangle\ \&\ not\, \langle exit{-}cond \rangle\ \&\ code_{loop}\ \Rightarrow\ \langle loop{-}invariant'' \rangle$

iii. Obtain the Post Condition:
We know the Loop Invariant is true because we have maintained it.
We know the Exit Condition is true because we exited.
We do a little extra work.
And then prove that it follows that the Post Condition is true.
$\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{–}loop}$ $\Rightarrow$ $\langle post{-}cond \rangle$

(c) How is a system invariant same and different?

- Answer:
  - Same:
    - The importance of invariants is the same.
    - Both have invariants that must be maintained.
  - Differences:
    - An algorithm must terminate with an answer,while systems and data structures may run forever.
    - An algorithm gets its full input at the beginning, while data structures gets a continuous stream of instructions from the user.

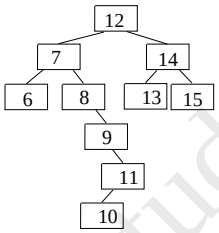
(d) Give three or four sentences to explain how the system invariant is defined for the previous *Stack* question and how invariant steps are accomplished.

- Answer:
  i. The picture in question (a) will act as the system invariant for this stack implementation.
  ii. The *constructor* code in (b) needs to establish this invariant.
  iii. The *push* and *pop* code in (c,d) needs to maintain this invariant.
  iv. There is no to terminate the system invariant or any post condition to obtain.

3. (68% in 2015) Consider the following **Binary Search Tree**.

(a) Starting from 11 label the nodes $10, 9, 8, \ldots$ in the order that they would be visited if you repeatedly called *previous* as coded in assignment 2 and $12, 13, 14, \ldots$ if you repeatedly called *next*.

- Answer:



(b) How do you know that value at node 11 is at most the value at node 12?

- Answer: Because 11 is in the left subtree of the node 12. In a Binary Search Tree, EACH node v has a value that is greater or equal to the values at each of the nodes in its left subtree and less than or equal to the values at each of the nodes in its right subtree.

(c) Part of the instructions for the *next* routine was the following.

If your current node does not have a right child, travel *up and left* as far as you can and then *up and right*. (Going "up and left" means testing of you are the right child of your parent and if so going to your parent.) Said another way follow the path up parent parent parent. Stop the first time that the node you came from is the left node of where you currently are.

For example, this should get you from node 11 to node 12.

Give the code for just this part of the assignment. No error correcting needed.

- Answer:

```
if(super.rightChild(position)!= null)
        ...
else
        while( super.parent(position)!=null &&
                super.rightChild(super.parent(position))==position )
                        position = super.parent(position);
        position = super.parent(position);
```

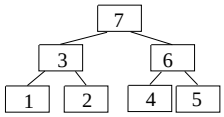
or (enough for full marks)

```
while( position.parent.right == position ) position = position.parent;
position = position.parent;
```

4. (60% in 2015)
Read the string 4 3 2 + 5 *. When you see a number push it on the stack. When you see an operator, pop two numbers and perform the operation on them, and push the result. What is on the stack at the end?

   - Answer: 4, (3+2)*5=25

5. Write a recursive program that takes as input a binary tree and prints out its nodes in the order shown in the following figure. What is the name of this order?



   - Answer:

algorithm $PostFixOrder(tree)$
⟨*pre−cond*⟩: $tree$ is binary tree.
⟨*post−cond*⟩: Its nodes are printed in post fix order

```
begin
    if( tree! = null )
        PostFixOrder(tree.left)
        PostFixOrder(tree.right)
        Print(tree.info)
    }
end algorithm
```

6. Jeff feels that a major cause of students through to grad school not understanding computer science material is because they don't understand first order logic. Example:
$\forall x,\ \exists y,\ x + y = 0.$
This states that every number x has an additive inverse $y = -x$.

   Jeff uses the following game to prove a first order logic statement. Read the expression left to right. If a variable is quantified with an exists ($\exists$), then the prover produces an object for that variable. If it quantified with a forall ($\forall$), then the adversary does. The prover wins the game, if the inner statement is true. The statement is true, if the prover can always win. The proof of our example is as follows.

Proof of $\forall x,\ \exists y,\ x + y = 0$:
Let x be an arbitrary real number.
Let y be $-x$.
Note that $x + y = x + (-x) = 0$.

A computational problem is a function P from inputs I to required output $P(I)$.
An algorithm is a function A from inputs I to actual output $A(I)$.
$A(I) = P(I)$ *and* $Time(A, I) \leq 3|I|^2$
states that algorithm A solves problem P on instance I in at most $3n^2$ time.

For each of the following statements, check the first order logic statement that expresses the statement. Hint: Consider (but do not write) the game played to prove that the statement is true.

(a) Problem P is computable.
   i. $\forall I\ \exists A\ A(I) = P(I)$.
   ii. $\forall A\ \exists I\ A(I) = P(I)$.
   iii. $\exists I\ \forall A\ A(I) = P(I)$.
   iv. $\exists A\ \forall I\ A(I) = P(I)$.

(b) Problem P is uncomputable.
   i. $\forall I\ \exists A\ A(I) \neq P(I)$.
   ii. $\forall A\ \exists I\ A(I) \neq P(I)$.
   iii. $\exists I\ \forall A\ A(I) \neq P(I)$.
   iv. $\exists A\ \forall I\ A(I) \neq P(I)$.

(c) Algorithm A's running time is $\mathcal{O}(1)$. (Ignore the n_0 part.)
   i. $\forall I\ \exists c\ Time(A, I) \leq c$.
   ii. $\forall c\ \exists I\ Time(A, I) \leq c$.
   iii. $\exists I\ \forall c\ Time(A, I) \leq c$.
   iv. $\exists c\ \forall I\ Time(A, I) \leq c$.

- Answer: iv,ii,iv

7. Consider the function $f(n) = 9n^3 \log^{100}(n) + 5 \cdot n^{5.5} \log^2(n)(\sin(n^2) + 5)$.
What is $\Theta(f(n))$? Why?
Is it one of the following: $\Theta(1)$, $2^{\Theta(n)}$, $\log^{\Theta(1)}(n)$, or $n^{\Theta(1)}$. Why? What is this class called?

   - Answer: $f(n) = \Theta(n^{5.5} \log^2(n))$. Theoretically at least the $\log^{100}(n)$ will get dominated by the $n^{5.5}$. The $\log^2(n)$ can't be dropped because it is not a constant, but $(\sin(n^2) + 5)$ can because it is bounded by 6.
   For big n, this is bounded between $n^{5.5}$ and $n^{5.6}$. Hence, $f(n) \in n^{\Theta(1)}$. This is a polynomial function.

8. Running Time: Give the Theta (bigO) of time complexity (running time) of each of these programs. Explain your work.

algorithm $Ex0(\langle x_1, x_2, \ldots, x_n \rangle)$

$\langle pre\text{-}cond \rangle$**:** x is an array of n integers.

$\langle post\text{-}cond \rangle$**:** Some number of "Hi"s are printed

```
begin
    loop i = 1...n {
        m = n
        while( m > 1 ) {
            Print("Hi")
            Print("Hi")
            m = m/2
        }
        Print("Hi")
    }
end algorithm
```

algorithm $Ex1(\langle x_1, x_2, \ldots, x_n \rangle)$

$\langle pre\text{-}cond \rangle$**:** x is an array of n integers.

$\langle post\text{-}cond \rangle$**:** Some number of "Hi"s are printed

```
begin
    m = n
    while( m > 1 ) {
        loop i = 1...m {
            Print("Hi")
            Print("Hi")
        }
            Print("Hi")
        m = m/2
    }
end algorithm
```

algorithm $Ex2(N)$

$\langle pre\text{-}cond \rangle$**:** N is an integers.

$\langle post\text{-}cond \rangle$**:** Some number of "Hi"s are printed

```
begin
    loop i = 1...N {
        loop i = 1...N {
            Print("Hi")
            Print("Hi")
        }
    }
end algorithm
```

algorithm $Ex3(\langle x_1, x_2, \ldots, x_n \rangle)$

$\langle pre\text{-}cond \rangle$**:** x is an array of n integers.

$\langle post\text{-}cond \rangle$**:** Some number of "Hi"s are printed

```
begin
    if( n ≤ 1 )
        Print("Hi")
    else
        loop i = 1...n^3
            Print("Hi")
        end loop
        Ex3(⟨1 × x_1, x_2, ..., x_{n/2}⟩)
        Ex3(⟨2 × x_1, x_2, ..., x_{n/2}⟩)
        Ex3(⟨3 × x_1, x_2, ..., x_{n/2}⟩)
        Ex3(⟨4 × x_1, x_2, ..., x_{n/2}⟩)
        Ex3(⟨5 × x_1, x_2, ..., x_{n/2}⟩)
        Ex3(⟨6 × x_1, x_2, ..., x_{n/2}⟩)
        Ex3(⟨7 × x_1, x_2, ..., x_{n/2}⟩)
        Ex3(⟨8 × x_1, x_2, ..., x_{n/2}⟩)
    end if
end algorithm
```

- Answer:

  $Ex0$**:** The outer loop iterates n times. The inner loop needs to divide by two $\log n$ times to get m from n down to 1. The number of "Hi"s printed just just changes the constant. $Time(n) = \Theta(n \log n)$.

  $Ex1$**:** On the i^{th} iteration of the outer loop m has the value $\frac{n}{2^i}$ and hence the inner loop iterates this number of times. This is a geometric sum which is dominated by the biggest term. $Time(n) = \sum_{i=0}^{\log n} \Theta(\frac{n}{2^i}) = \Theta(n)$.

  $Ex2$**:** N is a value. The size of this input is $n = \log_2(N)$. This gives $N = 2^n$. $Time(n) = \Theta(N^2) = \Theta((2^n)^2) = \Theta(2^{2n})$. Any answer $2^{\Theta(n)}$ is fine.

  $Ex3$**:** The work the top level does is n^3 giving $c = 3$. The number of friends is $a = 8$. The size of their input is $\frac{n}{2}$ and hence $b = 2$. This gives the recurrence relation $T(n) = 8T(n/2) + n^3$. $\frac{\log(a)}{\log(b)} = \frac{\log(8)}{\log(2)} = 3 = c$. Hence the work is the same at all levels giving $Time(n) = \Theta(n^3 \log n)$.