CSCI 33500 - Spring 2016 - Answers to Homework #2, Covering chapters 3 and 4.

For this assignment, follow these pseudocode guidelines:
- all type declarations are considered implicit (but no penalty for including include them).
- brackets should still be present to define the scope of procedures/loops.
- use indentation to make these scopes easy to understand.
- make sure that * pointers are used correctly and when needed.
- be explicit and add comments liberally.

1 Chapter 3 - List, Stack, and Queues

1.1 Given a *simple* linked list L of size N, write a procedure `reverse(L)` that returns the list L in reverse order, using only the basic list operations. Give an analysis of the runtime of your procedure.
(The basic list operations are defined in chapter 3.3 / page 81-83, and are `size()`, `clear()`, `empty()`, `front()`, `push_front(x)`, `pop_front()`, `begin()` and the associated iterator methods. Note that `push_back(x)`, `pop_back()`, `end()` are not available for a simple liked list.)

Answer: Pseudocode (iterative version):

```
reverse_iter(L)
{
   iterL = L.begin();
   rev = List(); // Empty List
   while( *iterL != nullptr)
   {
      // for each element of the iterator,
      // push it to front of the rev list.
      rev.push_front(*iterL);
      iterL++;
      // the last element of L will be put to the front of L last
      // thus the list will be reversed.
   }
   return( rev)
}
```

Runtime analysis: this procedure goes through each element of the list exactly once, and all the statements in the loop are O(1), so the total runtime is O(N).

For the recursive version, use an accumulator (also called Continuation Passing Style). Base case: when the list is empty.

1.2 Given two ***sorted*** simple linked lists, L1 and L2, of size N and M respectively, write a procedure to compute L1 XOR L2, the list of elements that are in either L1 or L2 but not in both, using only the basic list operations. Give an analysis of the runtime of your procedure.

Answer: Pseudocode (iterative version):

```
exclusiveor_iter(L1, L2)
{
   iterL1 = L1.begin();
   iterL2 = L2.begin();
   xor_list = List(); // Empty List
   while(iterL1 != nullptr && iterL2 != nullptr )
   {
      // because the lists are sorted, we know the smallest
      // of these values is in only one of the lists
      if ( *iterL1 < *iterL2)
      {
         xor_list.push_front(*iterL1);
         iterL1++ ;
      }
      else if ( *iterL1 > *iterL2))
      {
         xor_list.push_front(*iterL2);
         iterL2++ ;
      }
      else // they are both equal, dont add either and increment both
         iterL1++; iterL2++;
   }
   // at this point one of the iterators will point to nullptr
   // we still need to add the leftover elements from the other list
   while(iterL1 != nullptr)
      xor_list.push_front(*iterL1);
   while(iterL2 != nullptr)
      xor_list.push_front(*iterL2);
   // because of the while termination condition
   // only one of the previous loops will be executed

   return( xor_list );
}
```

Runtime analysis: Each iteration of the loop increments at least one of the iterators. At the end, each iterator is at null. The run time is then O(N+M).
For the recursive version, use front() and pop_front() instead of iterators. Base case: when one of the lists is empty.

1.3 Transform the following *infix* expressions to their *postfix* form (see section 3.6.3 / page 105 of the book, PEMDAS order of operation):
`a * x ^ 3 + b * x ^ 2 + c * x + d`
`d + x * ( c + x * ( b + x * a ) )`

For one of these transformations (of your choice), draw the stack and output as it changes for each stack operation. Follow the example given in page 109 of the book.

Answer: infix `a * x ^ 3 + b * x ^ 2 + c * x + d`
to postfix `a x 3 ^ * b x 2 ^ * + c x * + d +`

input	stack	output	comments
a		a	operands go straight to output
*	*	a	first operator is pushed
x	*	a x	
ˆ	* ˆ	a x	operator is of higher priority, push
3	* ˆ	a x 3	
+	+	a x 3 ˆ *	pop until end or higher priority operator
b	+	a x 3 ˆ * b	
*	+ *	a x 3 ˆ * b	
x	+ *	a x 3 ˆ * b x	
ˆ	+ * ˆ	a x 3 ˆ * b x	
2	+ * ˆ	a x 3 ˆ * b x 2	
+	+	a x 3 ˆ * b x 2 ˆ * +	pop until end or higher priority operator
c	+	a x 3 ˆ * b x 2 ˆ * + c	
*	+ *	a x 3 ˆ * b x 2 ˆ * + c	
x	+ *	a x 3 ˆ * b x 2 ˆ * + c x	
+	+	a x 3 ˆ * b x 2 ˆ * + c x * +	
d	+	a x 3 ˆ * b x 2 ˆ * + c x * + d	
end		a x 3 ˆ * b x 2 ˆ * + c x * + d +	end is read, pop all stack

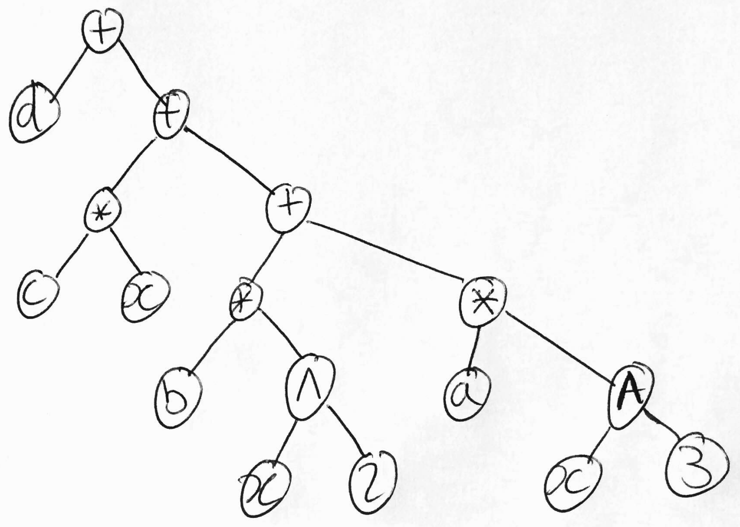
infix	`d + x * ( c + x * ( b + x * a ) )`
to postfix	`d x c x b x a * + * + * +`

2 Chapter 4 - Trees

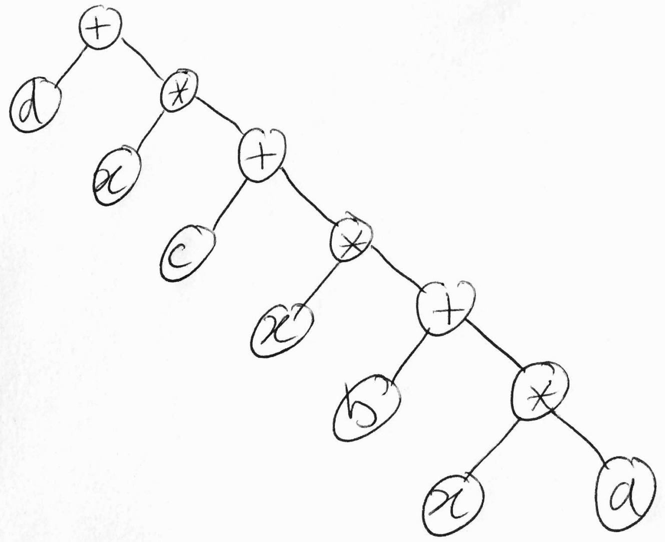
2.1 Draw the binary trees corresponding to the expressions of question 1.3. Parentheses should not appear.

Answer:

Tree for $a*x\wedge 3+b*x\wedge 2+c*x+d$.



Tree for $d+x*(c+x*(b+x*a))$

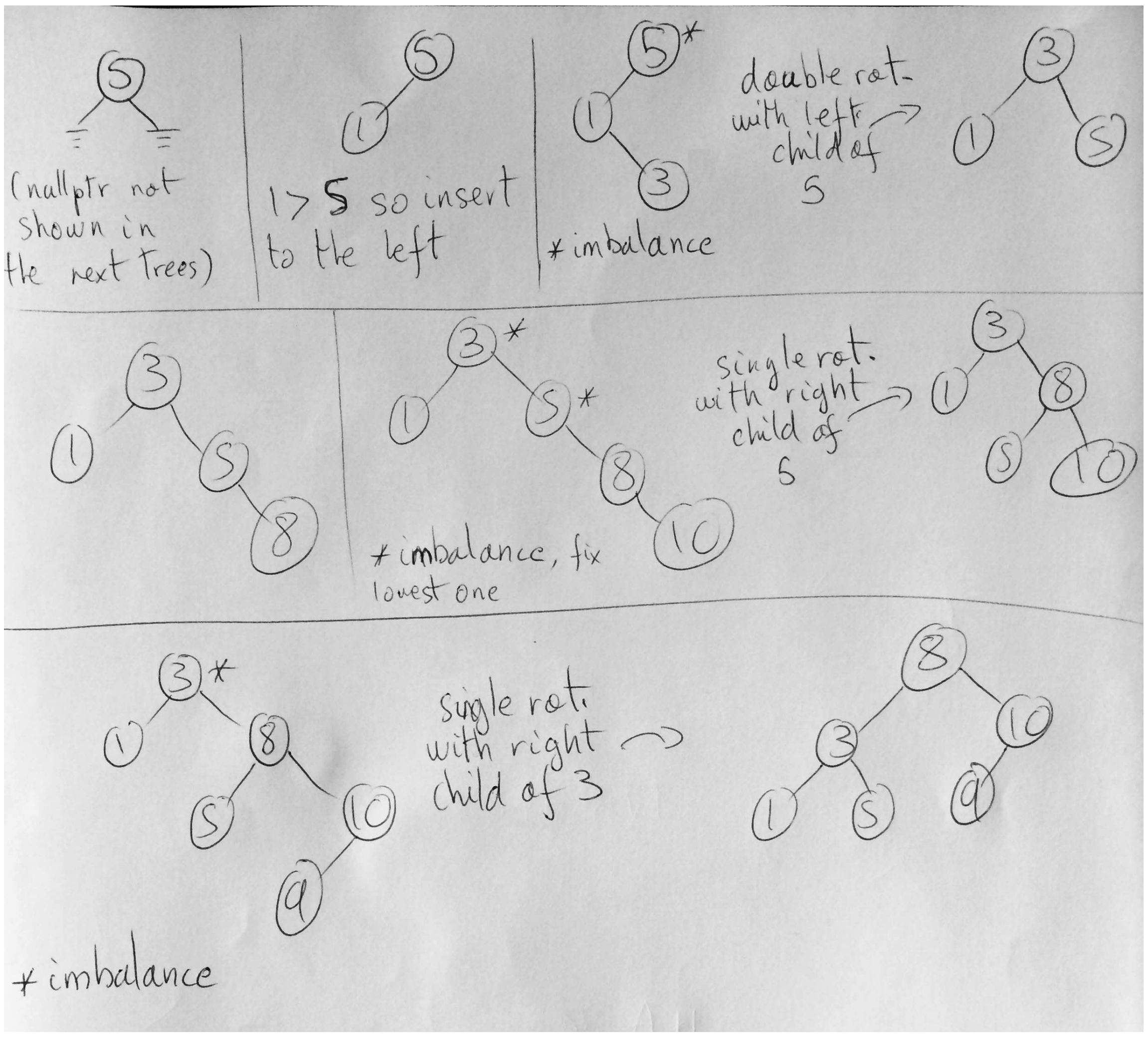


2.2 Prove by *induction* that in a tree of N nodes, there are N+1 `nullptr` links representing children.

Anwer: Let F(N) be the `nullptr` links in a tree with with N nodes.
Base case: for N=0, there is 1 `nullptr` link, the hypothesis is true.
another base case: for N=1, there are 2 `nullptr` links, the hypothesis is also true.
for any base case where $N \geq 2$, you need to verify the hypothesis for all the possible trees of that size.
Induction: assume that for a given $N \geq 0$, F(N) = N+1
Let's prove that F(N+1) = N+2
When I add a node to a tree of size N, I remove one `nullptr` from the parent and add two from the node added.
So F(N+1) = F(N)-1+2
by the induction hypothesis (that F(N) = N+1), we have:
F(N+1) = N+1-1+2 = N+2 Q.E.D.
By induction, the hypothesis is true for all $N \geq 0$.

2.3 Show the steps and final result of inserting $5, 1, 3, 8, 10, 9$ (in this order) into an initially empty AVL tree. At each step, show the tree before and after balancing, and state if you are performing a simple/double rotation with the left/right child.

Answer:



(nullptr not shown in the next trees)
1 > 5 so insert to the left
* imbalance
double rot. with left child of 5
single rot. with right child of 5
* imbalance, fix lowest one
single rot. with right child of 3
* imbalance

3 Extra Credit

Write recursive versions for each of procedure you've written.