1

(a)

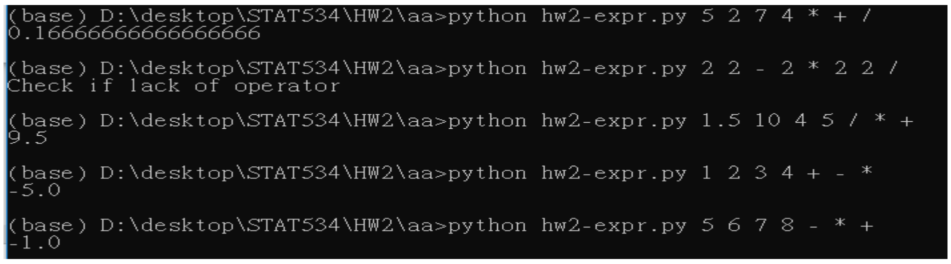
```
# error checking
Set argc = len(argv) - 1
if argc == 0: # no input argument
    print(“There is no input to compute”)
elif argc > (a const): # the const depending on the memory of cpu
    print(“Too many input argument for stack!”)
elif (number of digit – 1 < number of operators):
    print(“Stack is empty”) # meaning that there are too many operators.
# set a dict for operation.
ops = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.truediv}

for i in range(1,argc):
    if argv[i] is a number:
        push
    if argv[i] is operators:
        if argv[i] == ‘+’ or argv[i] == ‘*’:
            push(ops[argv[i]](pop(),pop()))
# since ‘plus’ and ‘multiply’ have commutative properties, we don’t care about the
#sequence. Simply take out whatever two on the top and compute
        if argv[i] == ‘-’ or argv[i] == ‘/’:
            num = pop()
            push(ops[argv[i]],pop(),num)
# we can not ignore the sequence so take it out first and compute.
        Print(“ans: ”, whatever in the stack)
```

1(b)

Implement in hw2-expr.py

This is tested by python 3.7

1(c)

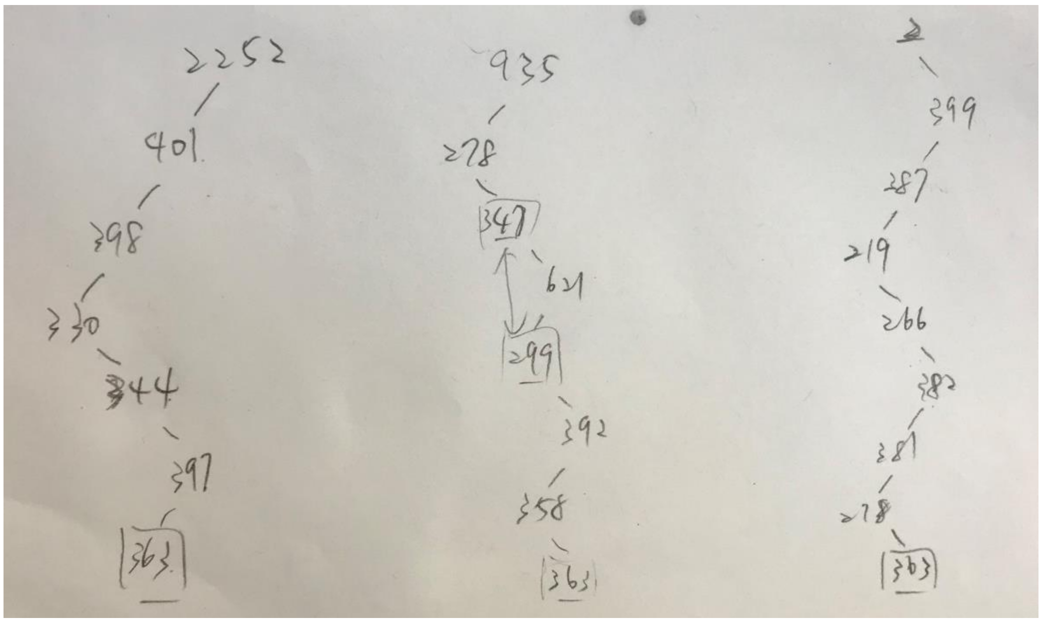


```
(base) D:\desktop\STAT534\HW2\aa>python hw2-expr.py 5 2 7 4 * + /
0.16666666666666666

(base) D:\desktop\STAT534\HW2\aa>python hw2-expr.py 2 2 - 2 * 2 2 /
Check if lack of operator

(base) D:\desktop\STAT534\HW2\aa>python hw2-expr.py 1.5 10 4 5 / * +
9.5

(base) D:\desktop\STAT534\HW2\aa>python hw2-expr.py 1 2 3 4 + - *
-5.0

(base) D:\desktop\STAT534\HW2\aa>python hw2-expr.py 5 6 7 8 - * +
-1.0
```

2.(a)

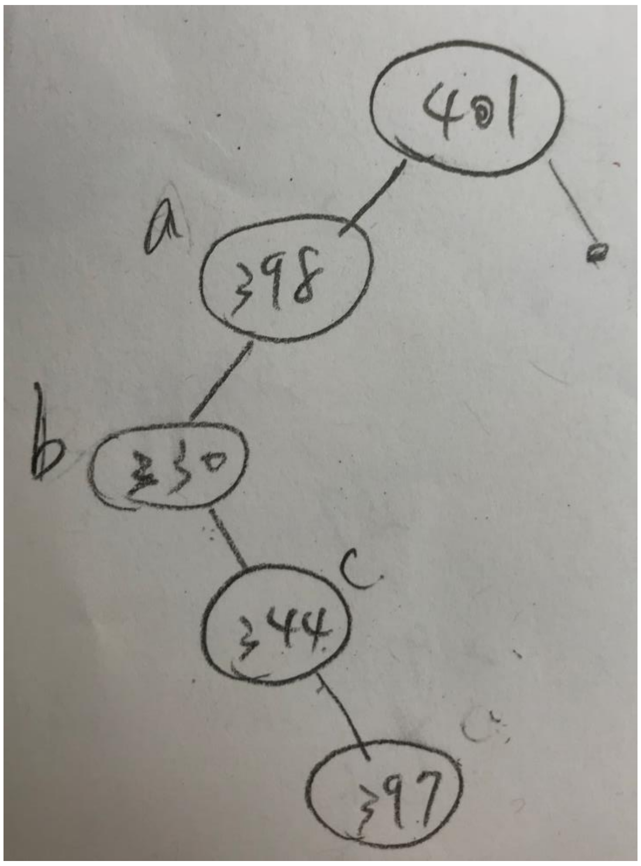
Binary search tree properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.



The second one is wrong, since it disobey binary tree’s property, which is right subtree’s key is larger than their ancestors. 299 is not greater than 347.
However the first one is weird as well since from the problem statement, the number should be 1 to 1000. The root is 2252.

2.(b)



A: The key to the left of the search path.

B: The keys on the search path.

C: The key to the right of the search path.

This is the counter example. Say we want to search for 397, which is leaf of this tree. Also, a =398, b=330, and c=344. All of them satisfy the properties prof. Bunyan $a \in A, b \in B, c \in C$. However, $b < c < a$, which does not satisfy the property $a \leq b \leq c$

3(a)

The insertion time complexity will be decided by the height of the tree, and since there will be multiple identical keys k, we know that the tree will be a balanced tree. The height of the tree will be $\log_2 n$; therefore, the asymptotic performance is $O(\log_2 n)$

3(b)

If we want to delete **any** node in the tree, there will be three cases.

I. Delete the leaf
II. Delete the node with only one child.
III. Delete the node with two children.

For the first case, we need to go deep to the leave and get the node, which will take $O(\log_2 n)$ and connect the parent to NULL, taking $O(1)$. Therefore, total $\rightarrow$ $O(\log_2 n)$

For the second case, we need to go find the node we want to delete, taking $O(\log_2 n)$ in the worst case, and put the only child to replace it, taking $O(1)$. Therefore, total $\rightarrow O(\log_2 n)$

For the third case, we need to go find the node we want to delete, taking $O(\log_2 n)$ in the worst case, and put the successor to replace it, in this case since all the keys are k, taking $O(1)$ to find it. Therefore, total $\rightarrow O(1)$

3(c)

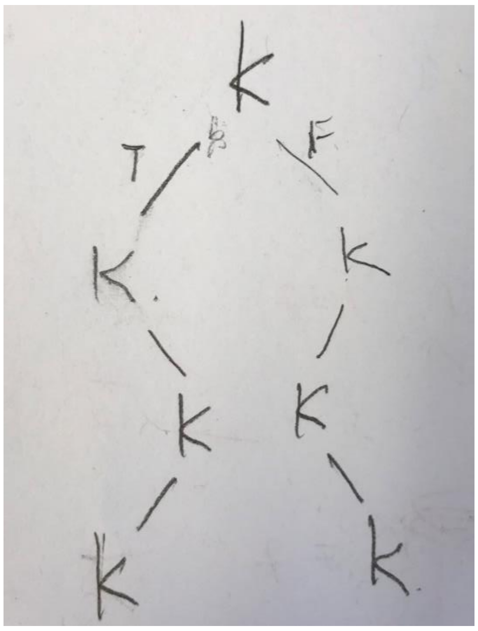
Rule:

Flag b == True: go left.

Flag b == False: go right.

Flag switch during insertion every time it is checked.

Here's my tree:



(d)

By induction,

m = 1: number of nodes is 1, depth of tree is 1, thus the insertion time is O(2^0*1)

m = 2: number of nodes is 3, depth of tree is 2, thus the insertion time is O(2^1*2)

m = 3: number of nodes is 7, depth of tree is 3, thus the insertion time is O(2^2*3)

m = m: number of nodes are $2^m - 1$, depth of tree is m, thus the insertion time is $\mathrm{O}(2^{m-1} * \mathrm{m})$

sum up from m = 1 to m the total run time will be $\mathrm{O}(m^3)$

(e)

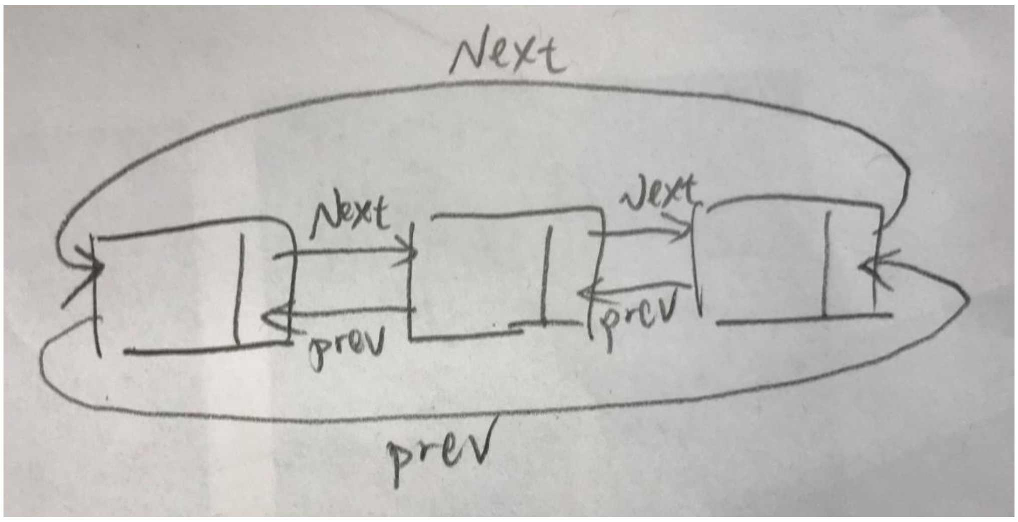
Since it's doubly link list, whenever I insert a node, I can just put it in the end and link to the root. Therefore it's $\mathrm{O}(1)$

4(a)

No, a singly linked list can not have multiple loops. It is because the use of linked list is to use pointer to point out the next address and show the value so we will only point out one address for the next value. However, for multiple loops, it is inevitable that we will need to have multiple addresses in whichever node. This will cause a problem that we can not precisely point out which to go next.

4(b)

Yes, doubly linked list can have loop. Doubly linked list means that we have a link to run back and forth of two nodes. Say if the last node did not point to NULL but simply point back to the head. It means that we can run back and forth between the tail and head but did not ruin the property of doubly linked list. Look at the picture below:



4(c)

We can simply divide the situation into 4 cases.

I. Odd number of nodes and odd number of link in the loop.
II. Odd number of nodes and even number of link in the loop.
III. Even number of nodes and odd number of link in the loop.
IV. Even number of nodes and even number of link in the loop.

And we set two pointers named “ptr1” and “ptr2”. ptr1 moves 1 node per iteration and ptr2 moves 2 nodes per iteration.

Because of least common multiple (L.C.M), we can tell that the ptr1 and ptr2 must meet at the same node if there is a loop in the linked list. This is true for all the four cases. Therefore, we can use this to decide whether there is a loop in the linked list.

Pseudo:

```
ptr1 = head of the list.
ptr2 = head of the list
def loop_check(link list):
    while(ptr1 != address of link list end):
        ptr1 -> next
        ptr2 -> next -> next
        if (ptr1 == ptr2):    // means that the pointers meet at the same address.
            print(“Loop, we got you!”)
            return 1
check = loop_check(link list)
if check != 1:
    print(“There is no loop in this link list”)
```

Runtime analysis:

Whether the linked list has odd or even number of link, the pointer will meet eventually no later than reaching to the end. Assume the link has n elements. Run the whole linked list is the worst case and will be bounded by $O(n)$