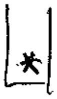
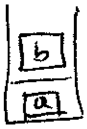
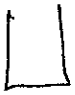
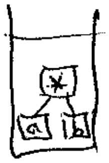
Assignment 4

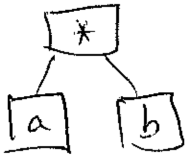
1. Infix to expression tree: a*b/(c+d−e*(f+g)/h)

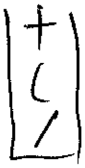
After 3 tokens: opstack: 

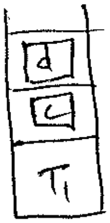
exptree stack: 

/ has same precedence as *, so pop * from opstack, 2 expressions from expstack, merge them with * as op:

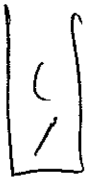
opstack: 
exp stack: 

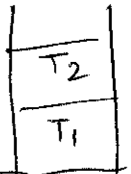
T1: 

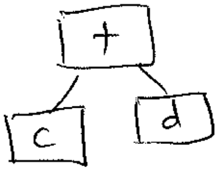
After 5 tokens: op stack: 

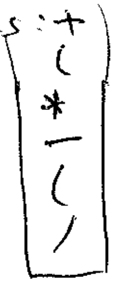
expstack: 

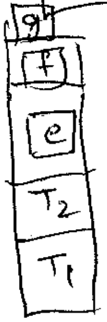
− has same precedence as +, pop + from stack, 2 expressions from exp stack and merge them with + as operator.

Opstack: 

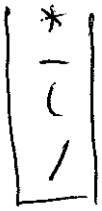
expstack: 

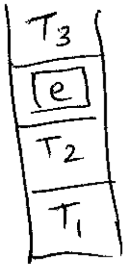
T2: 

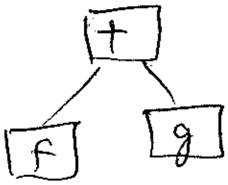
After 7 tokens:
opstack: 

expstack: 

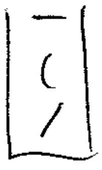
When) is processed, pop opstack up to (:

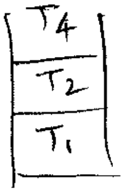
opstack: 

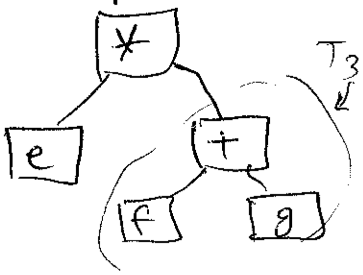
exp: 

T3: 

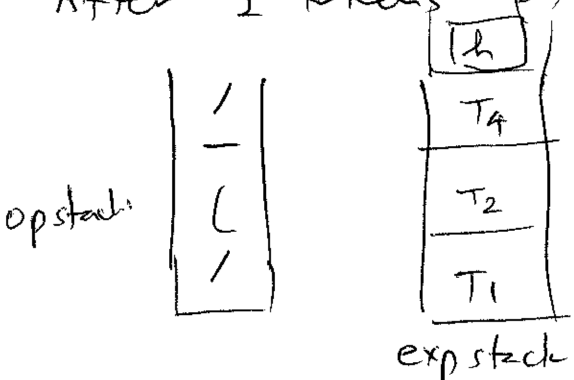
/ has same precedence as *, pop * and 2 exp:



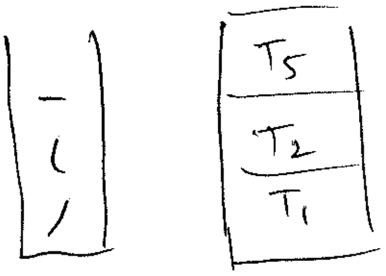
exp: 

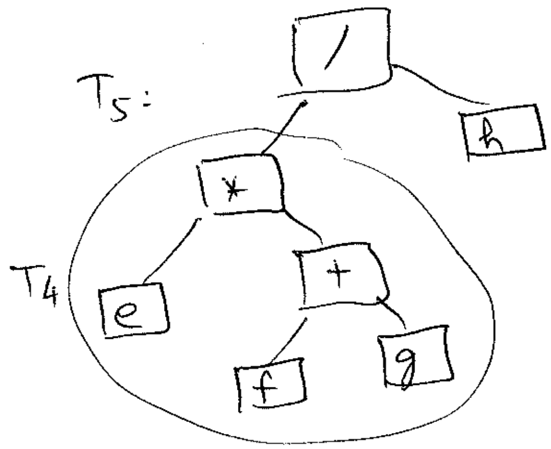
T4: 

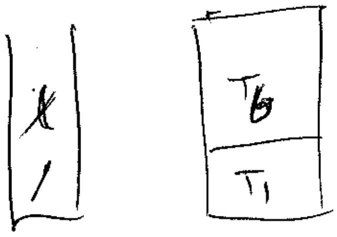
After 2 tokens ~~6~~/ h~~$~~:

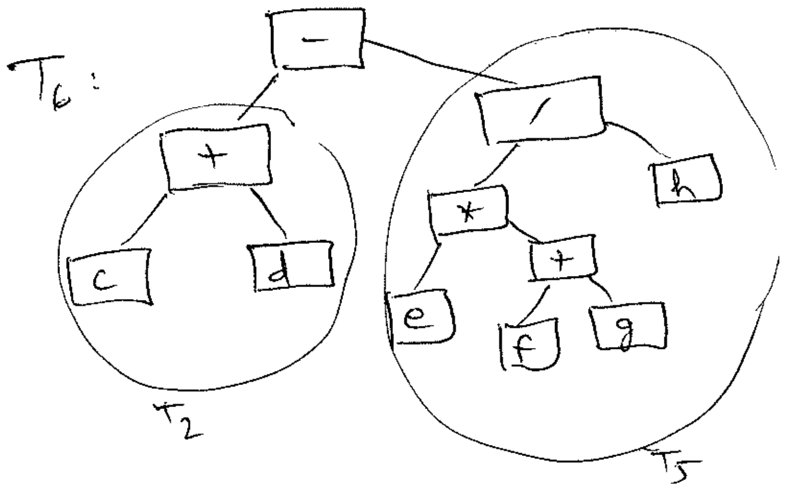


When processing), pop operators from stack and process them until) is removed:

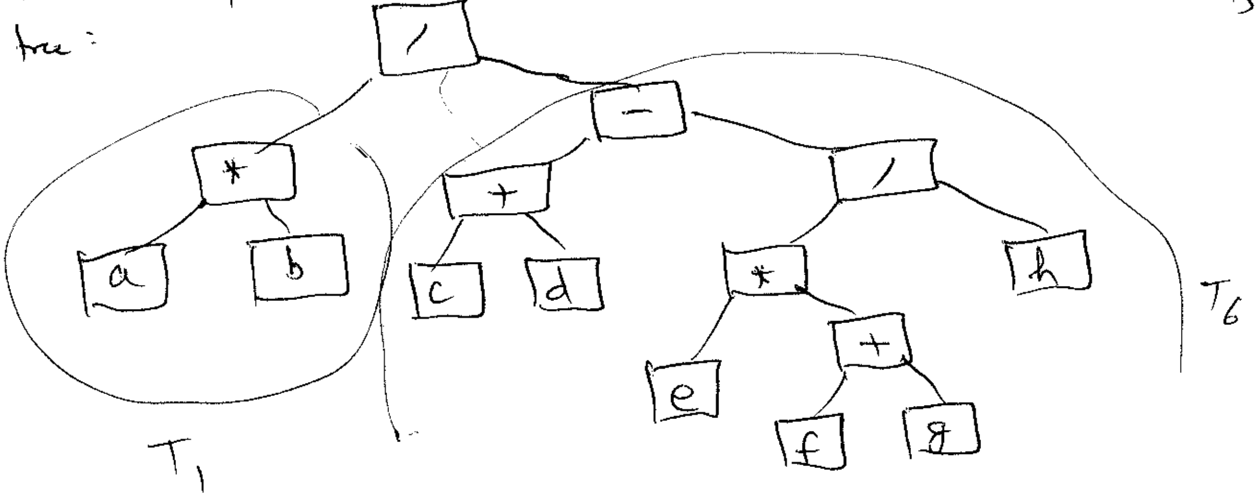








Input is done: pop remaing operators and process them.
Final tree:



Solution to Q2 on Assignment 4

```
// return elements at index s, s+k, s+2k, s+3k, ...
List filter( List l, int s, int k ): // RT = O(n).
  List result ← empty list of int
  if l.size() >= 1+s then // result’s size > 0
      Iterator it ← l.iterator();
      for i ← 0 to s do
        x ← it.next()
      result.add(x)
      count ← 0
      // LI: s+q*k+count elements processed
      while it.hasNext() do
        count ← count + 1
        x ← it.next()
        if count = k then
            result.add(x)
            count ← 0
  return result
```

Binary Seach Trees in Java Library

TreeSet, TreeMap — Red Black trees
(extension of BST's)

TreeSet: implementation of interface Set.

Usage: Set<Integer> s = new TreeSet<>();
s.add(x); // add new integer x to set
// duplicates are rejected

TreeSet<Vertex> for user defined class Vertex:

(i) Vertex class defines: (a) Comparable<Vertex>:
public int compareTo(Vertex other)
in Vertex class — natural ordering

or (b) Comparator<Vertex> passed as a parameter
Set<Vertex> s = new TreeSet(comp)
comp = object that implements
Comparator<Vertex>

- user defined ordering:
public int compare(Vertex one, Vertex two)

Convention: $a \leq b$? a.compareTo(b) or compare(a,b

$a < b$: return -1 (any negative no) $a = b$: 0 $a > b$: return $+1$ (any positive no

Note: It is customary to provide:

```
boolean equals (Object other) ,  — a.equals(b)?
   int  hashCode ( )  — used in ~~[illegible]~~ hash tables.
```

```
s.remove(x);      // remove x from s

s.contains(x);    // does x appear in set?

s.iterator();  ← create an iterator to go
                 through elements of s
                   in sorted order of keys.
```

Usage of iterator:

explicit:

```
Iterator<Integer> it
          =  s.iterator();
// s = TreeSet<Integer>
while (it.hasNext()) {
      Integer x = it.next();
          :
       it.remove(); // if needed
}
```

implicit

```
// for classes that are Iterable.
  for (Integer x : s) {
               // process x
         :
  }
```

Implicit iterator does not allow you to remove current object.

TreeMap : Map of tuples K, V (Key-value pairs)

K = Keys that have no duplicates, no Null keys. K → V

V = no restrictions.

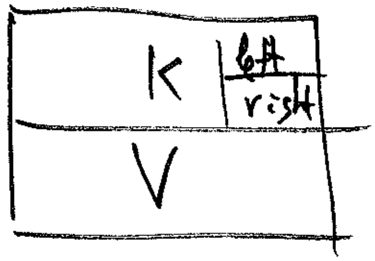
Usage: Map<Vertex, Integer> m = new TreeMap<>(

//map using natural ordering of Vertex class.

Can also create TreeMap<>(comp) for user defined ordering.

↑ Comparator<Vertex>

BST : Each Entry is



Operations:

m. get (key) : value stored with ~~key~~ K key. null if there is no such key.

m. put (key, value) : (a) if key exists already: replace its old value by value. old value is returned by put.

(b) if Key is new, add a new entry (key, value) return null.

Iterate over map:

for(Map.Entry ~~Map~~ <Vertex, Integer> e : m.entrySet( )) {

⋮

e.getKey( )
e.getValue( )

Iterate over objects in order of keys.

Additional ops: m.containsKey(key)
m.containsValue(value) ← O(n)
don't use.

Key based operations of TreeMap/TreeSet:
O(log n) per operation (n = # of keys in tree)

Iteration: O(log n) per op — worst case
O(1) per op — amortized.