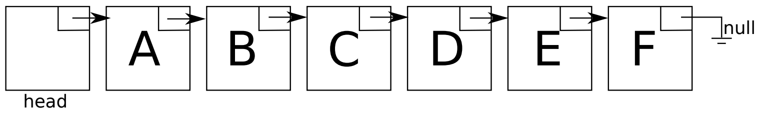
Question 1

Assume you are given a Simple Linked List (i.e. not a doubly linked list) containing an even number of elements. For example

$$L = \texttt{[A B C D E F]}.$$

- a) Draw the linked node structure of L, including a head node.



- b) Describe an O(N) algorithm (in pseudocode or your own words) that flips every adjacent pair of elements in the list. For instance, the algorithm should convert L into `[ B A D C F E]`. Assume the list implementation does not provide an iterator.

```
prev = head;
a = prev.next;

while (a != null) {
    b = a.next;
    prev.next = b;
    a.next = b.next;
    b.next = a;
    prev = a;
    a = a.next;
}
```

Other solutions: Use recursion instead of a for loop. Swap the data stored in the nodes without changing the pointer structure.

Question 2

Perform the following sequence of operations on an initially empty Circular Array Queue of size 3. Show the layout of the array and the value of the front and back marker after each step.

`enqueue(A)`, `enqueue(B)`, `dequeue()`, `enqueue(C)`, `dequeue()`, `enqueue(D)`.

	0	1	2
enqueue(A)	f,b A		
enqueue(B)	f A	b B	
dequeue()		f,b B	
enqueue(C)		f B	b C
dequeue()			f,b C
enqueue(D)	b D		f C

Question 3

Analyze the running time of the following Java method in Big-O notation. Provide as tight a bound as possible. Then answer the following questions: What is the return value of `foo(11)`? What does the method `foo` compute in general?

```
public static String foo (Integer n) {
     if (n <= 1) return n.toString();
     Integer bar = n % 2;
     return foo(n / 2) + bar.toString() ;
}
```

(1) *Let N be the value of the main call to* `foo`*, so when we first call foo* $n = N$*. The method does not contain any loops, so the only thing affecting the runtime is the recursive call. We only need to figure out how often* `foo` *is called. The first recursive call passes* $N/2$ *to* `foo`*, the next recursive call passes* $\frac{(N/2)}{2} = N/4$ *to* `foo` *etc. We need* $\log_2 N$ *times before we reach the base case. The running time is therefore* $O(logN)$ *(This is the same pattern we have seen for binary search.)*

Here is a slighty more formal analysis of the running time: We can write the running time of foo as a recurrence relation: $T(N) = T(N/2) + 1$*. The 1 term stands for the constant time spent in the method body, the* $T(N/2)$ *is the time spent on the recursive call. We can now substitute the recurrence relation repeatedly.*

$$T(N) = T(N/2) + 1 = T(N/4) + 1 + 1 = T(N/8) + 1 + 1 + 1$$
$$= T(N/8) + 3 = T(N/2^k) + k$$

Since we are only interested in Big-O, we can round up N *to the next highest power of 2,* $N = 2^m$*.*

$$T(N) = T(2^m) = T(2^m/2^k) + k.$$

To reach the base case $T(1)$ *we need* $m = k$*. Then*

$$T(N) = T(2^m/2^m) + m = T(1) + m = O(m)$$

and since $N = 2^m$*,* $m = \log_2 N$*. Therefore* $T(N) = O(\log N)$*.*

(2) *The return value of* `foo(11)` *is the string "1011". The method computes a binary representation for the integer* n*.*

Question 4

```
public static void foo(LinkedList<Integer> l1, LinkedList<Integer> l2){
    for (Integer b : l1) {
        if (b > -1)
            System.out.println(l2.get(b));
    }
}
```

a) Assume that l1 has N elements, what should the input look like so that the method runs in $O(N)$ (this is the best-case input).

If all elements x in `l1` *are $x \leq -1$, no* `get` *operation is ever called on* `l2`*. We spend only constant time on each element in* `l1` *and the running time is $O(N)$.*

b) What is the big-O running time in the worst case? Provide as tight a bound as possible.

In the worst case, all elements in `l1` *are the index to the last element in* `l2`*. Let M be the length of* `l2`*. The runtime is then $O(N \cdot M)$.*

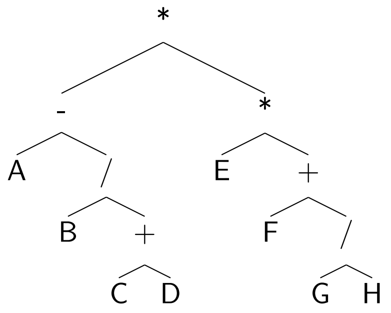
Question 5

Draw an expression tree for the following expression and give the expression in postfix and prefix notation.

$$(A - B/(C + D)) * E * (F + G/H)$$

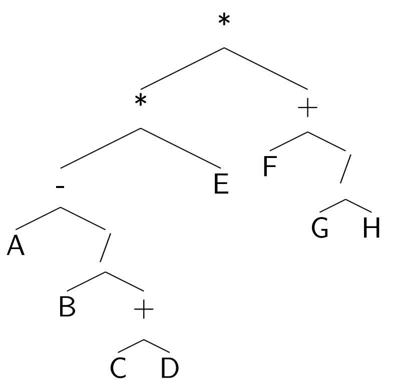
There are two possible trees, depending on the associativity of `*`.

Right-associative:



Pre-fix: * - A / B + C D * E + F / G H
Post-fix: A B C D + / - E F G H / + * *

Left-associative:



Pre-fix: * * - A / B + C D E + F / G H
Post-fix: A B C D + / - E * F G H / + *

Question 6

(a) Write the following arithmentic expression in post-order notation. $5 + 3 * (2 + 4) - 13$.

*5 3 2 4 + * + 13 -*
or
*5 3 2 4 + * 13 - +*

(b) Use a stack to evaluate the post-order expression. Show the push and pop operations used for each input character, and the resulting stack after each step. *Solution for 5 3 2 4 + * + 13 -*

```
1-4. Input: 5 3 2 4
   push(5), push(3), push(2), push(4)

  top->   4
          2
          3
          5

5. Input +
   pop()->4,  pop()->2,   push(2+4)

    top-> 6
          3
          5

6. Input *
   pop()->6,  pop()->3,   push(3*6)

   top->18
        5

7. Input +
   pop()->18,  pop()->5,   push(5+18)

   top->23

8. Input 13
   push(13)

   top->13
        23

9. Input -
   pop()->13,   pop->23(),  push(23-13)
   top->10

10. pop result
    pop() -> 10
```

Question 7

Prove by induction that any full Binary Tree has an odd number of nodes.

Induction over the number of nodes N.

Base case: For a single node $N = 1$ is obviously odd.

Inductive step: In a full binary tree, each non-leaf node has two children. The left child is the root of the subtree T_r and the left child is the root of the subtree T_l. Assume the inductive hypothesis is true for T_r and T_l, then the number of nodes N_r in T_r is odd and the number of nodes N_l in T_l is odd. $N_l + N_r$ must be even, since the sum of two odd numbers is even (see below). Adding the parent node, we get $N_l + N_r + 1$, which is odd. $\square$.

In the proof above we used the lemma that the sum of two odd numbers is even. Even though this lemma seems trivial, here is a proof (we would not expect you to prove this on an actual exam): An odd number, by definition, is any number that can be written as $2k + 1$, where k is an integer. If we add two odd numbers $2k + 1$ and $2j + 1$ we get

$$2k + 1 + 2j + 1$$

$$= 2(k + j) + 2$$

$$= 2(k + j + 1)$$

This number can be divided by 2 without a remainder, so it is even.

Question 8

An anagram is a word play in which the letters of one word are rearranged to form another word. For some word pairs, a single stack can be used to convert word A into word B. You can only scan the input word once (left to right) and you need to print the characters in the output word in sequence (left to right). For each of the following word pairs, show a sequence of push and pop operations that successfully transforms the input word into the output word, or explain why the conversion is impossible.

- aple, pale

  ```
  push(a), push(p), pop()->p, pop()->a, push(l), pop()->l, push(e), pop()->e
  ```

- insult, sunlit

  ```
  push(i), push(n), push(s), pop()->s, push(u), pop()->u, pop()->n, push(l),
   pop()->l, pop()->i, push(t), pop()->t
  ```

- rescued, reduced

  There is a typo in this part. Clearly 'rescued' cannot be transformed into 'reduced' because the words don't contain the same letters.

  A more interesting question would have been: rescued, reduces

  This is still not possible. We start by 'copying' re. Then we need to push 'scued' to the stack, because 'd' has to be the next letter that is popped and written to the output.

```
push(r), pop()->e, push(e), pop()->e,
push(s), push(c), push(u), push(e), push(d),
pop()->d
```

The next letter we need to write to the output is 'u', but it is now blocked by 'e' on the stack.

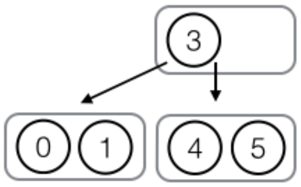
```
top-> e
      u
      c
      s
```

- rescued, secured

```
push(r), push(e), push(s), pop()->s, pop()->e,
push(c), pop()->c, push(u), pop()->u,
pop()->r,
push(e), pop()->e
push(d), pop()->e
```

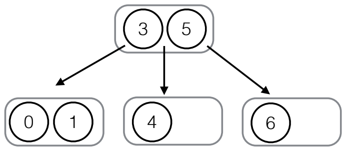
Question 9

Consider a 3-ary B-Tree:

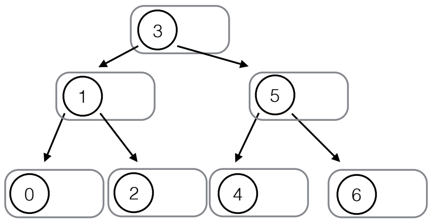


Perform the following sequence of operations. Draw the new B-tree after each step.
(1) insert 6. (2) insert 2. (3) insert 7. (4) remove 4.

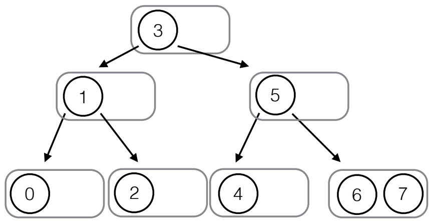
insert 6



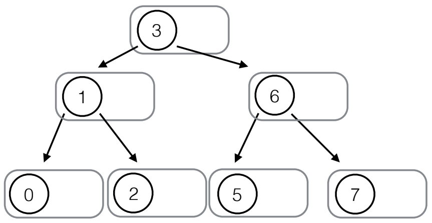
insert 2



insert 7



delete 4

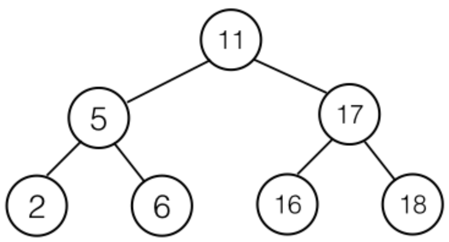


Question 10

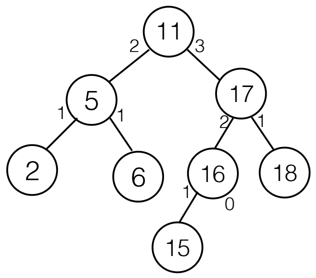
Perform the following sequence of insertions on the AVL Tree on the right.

`insert(15)`, `insert(14)`, `insert(13)`, `insert(12)`

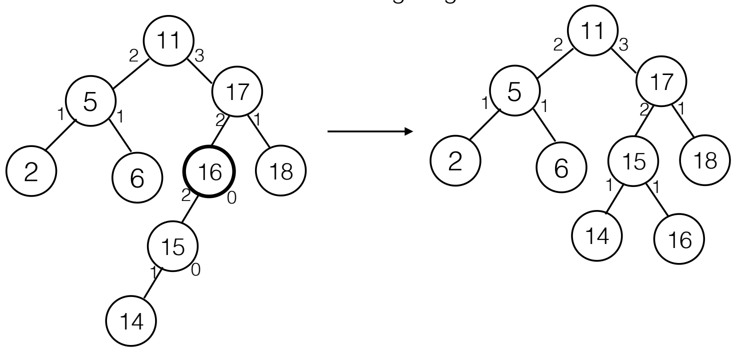
For every insertion, specify for which node the balance condition is violated (if any) and state what type of rotation is needed to restore balance. Draw the new tree after each insertion. Label every node with the height of its left and right subtree.



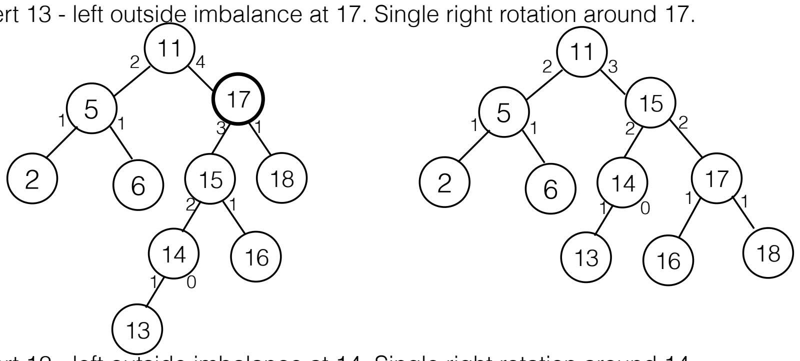
insert 15 - balanced



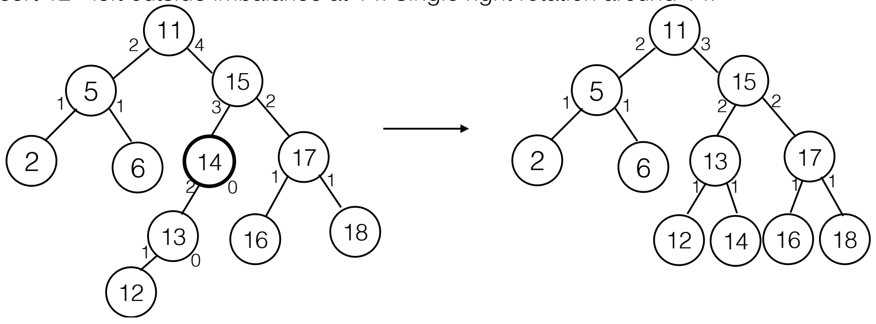
insert 14 - left outside imbalance at 16. Single right rotation around 16.



insert 13 - left outside imbalance at 17. Single right rotation around 17.



insert 12 - left outside imbalance at 14. Single right rotation around 14.



Question 11

The Hash Table on the right uses linear probing with $f(i) = 2i$.

- Fill the table by inserting the following integer keys (in order) using the hash function $hash(x) = x\%11$.
  16, 28, 6, 38, 5, 11, 49
- What is the load factor of the hash table?

The midterm will not contain questions about probing based collision handling, but for the sake of completeness we include the solution below (questions about hash functions and separate chaining are fair game).
The load factor is $7/11$.

index	key
0	11
1	
2	49
3	
4	
5	16
6	28
7	38
8	6
9	5
10	