==========

Problem 1:

The time for insert, find, delete are all O(n), which is the same with one item per link.

"Find" may take slightly less time because you can compare the first data of each LinkedNode to look for the correct Node storing the data.

"Delete" and "Insert" may take longer time because you have to change the location of other values in one array to maintain the structure, though the array length is just 8 and it takes O(1) time to do that. That runtime is determined by the structure you want to maintain, e.x., you may want the data to be maintained sorted.

Overall, the runtime order does not change.
==========

Problem 2:

```
Stack: []
push(15)    Stack: [15]
push(28)    Stack: [15,28]
pop()       Stack: [15] Output: 28
push(3)     Stack: [15,3]
push(81)    Stack: [15,3,81]
pop()       Stack: [15,3] Output: 81
pop()       Stack: [15] Output:3
push(9)     Stack: [15,9]
push(1)     Stack: [15,9,1]
pop()       Stack: [15,9] Output: 1
push(7)     Stack: [15,9,7]
push(4)     Stack: [15,9,7,4]
pop()       Stack: [15,9,7] Output: 4
pop()       Stack: [15,9] Output : 7
push(4)     Stack: [15,9,4]
pop()       Stack: [15,9] Output : 4
```

==========

Problem 3:

For array implementation:
  Advantage:
    no need to move data to keep structure in every delete and every insert. Less times to reallocate spaces.
  Disadvantage:
    Access data is harder because array[5] may points to a deleted value. Find data is less efficient because deleted values have to be checked. Storage space have small utilization ratio.
For linked list implementation:
  Advantage:
    No need to free or reallocate spaces every time an insert or delete happens. Insert is more efficient if data can be inserted into location marked "deleted". Delete is faster too.
  Disadvantage:
    The same with array implementation. Search takes longer time because deleted items have to be checked. Space has smaller utilization ratio.
For binary trees:
  Advantage:
    Similar to advantages of arrays and linked lists: Less times to reallocate spaces, delete and insert are more efficient.
  Disadvantage:
    Structure will be destroyed. Deleted items have to be compared for search. Re-organizing the whole tree after deleting and inserting a lot of items may be hard because it may need to totally re-build the whole tree.

==========

Problem 4:

Why do I need induction to prove this? I can prove it by:
  Let n0 be the number of leaves, n1 be the number of degree-one nodes, n2 be
  the number of degree-two nodes, then

  n0 + n1 + n2 = the total number of nodes
  n1 + 2 * n2 = the total number of links

  Notice that each node, except the root, has one incoming link, so

  the total number of nodes = the total number of links + 1

  Thus, n0 + n1 + n2 = n1 + 2 * n2 + 1, which is n2 = n0 - 1,
==========

Problem 5:

I do this by hand. See the pages attached behind.

==========

Problem 6:

Do a breadth-first search over the tree, i.e., level-order traverse, set the
root depth to be 0. We can use a queue to do the traverse, for each node with
depth i added to the queue, set its children to be depth i+1.

This algorithm traverse the tree only once, and set the depth of each node level
by level. So it takes O(n) runtime.

==========

Problem 7:

Use a stack to do that. The runtime is O(n), because each node is added to the
stack and visited once.

```
Initialize a stack
ADD root node to stack
WHILE stack is not empty
  WHILE the top node in stack has left child
    ADD the left child of the top node into the stack
  END_WHILE
  n = pop the stack
  visit node n
  IF n has right child
    ADD the right child of n into the stack
  END_IF
END_WHILE
```

==========

Problem 8, Problem 9 and Problem 10:

I do this by hand. See the pages attached behind.

==========

Problem 11:

The runtime is O(n), because each node is traversed once in each tree.

```
public boolean isSimilar(BST a, BST b){
```

```
  if(a == null && b == null) return true;

  if(a == null) return false;
  if(b == null) return false;

  return isSimilar(a->left, b->left) && isSimilar(a->right, b->right);
}
```

==========

Problem 12:

Runtime:

	AVL tree	Splay tree
Insert:	O(log n) average & worst	O(log n) average & O(n) worst
Search:	O(log n) average & worst	O(log n) average & O(n) worst
Delete:	O(log n) average & worst	O(log n) average & O(n) worst

Splay tree may be linear, in which case it takes O(n) to operate. However, by adding a randomized variant, the expected cost of Splay tree is O(log n)

Implementation:

AVL tree: rotate the tree after inserting or deleting, so that it is balanced: left subtree and right subtree differ in depth by at most one

Splay tree: rotate the tree after searching or inserting or deleting, so that the accessed node is closer to the root.

Splay tree changes the structure whenever search, insert and delete happens, so that recent accessed nodes are quick to access again.

AVL tree does not change the structure when search happens. It rotates for balancing the tree instead of moving accessed date to the root.

Splay tree is easier to implement than AVL trees

Splay tree is not good for multi-thread data accessing because structure is changing for each data accessing.

Splay tree works well with nodes containing identical keys—contrary comparing to AVL trees. All Splay tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms.