Data Structures and Algorithms
Week 3 problem sheet

## Linked Lists

1. Give three points of differences between an array and a linked list.

Array holds a fixed number of elements, linked list can grow and shrink

Array has direct access to elements, linked list need to follow from the head
Array is allocated a fixed amount of memory, linked list uses nodes allocated in the heap

1. Draw the **stack** and **queue** data structures with linked list implementations for each step in the following sequence:
* add(1), add(2), remove, add(3), add(4),
remove, remove, add(5).

top: 1 |null

top 2 🡒 1 | null

top 1 | null

top 3 🡒 1 | null

top 4 🡒 3 🡒 1 | null

top 3 🡒 1 | null

top 1 | null

top 5 🡒 1 | null (final)

Queue:

head, tail: 1 | null

head 1 🡒 tail 2 | null

head, tail: 2 | null

head 2 🡒 tail 3 |null

head 2 🡒 3 🡒 tail: 4 | null

head 3 🡒 tail: 4 | null

head, tail: 4 | null

head 4 🡒 tail: 5 | null

1. Let Link be an object with two member variables:
* public class Link {
 char item;
 Link next;
}
* Assume that the variable first is a reference to a Link object containing 'a', whose next is a link object containing 'b', whose next is null.
* 
* Which of the following code snippets successfully reverses this structure, so as to give the following list?
* 
* Draw a picture of the result of each code snippet to explain your answer.
	1. first.next = null;
	first.next.next = first;
	first = first.next;

first: a 🡒 b | null (start)

first a | null b | null

Then gives null pointer exception (thrown when first.next.next called)

* 1. first.next.next = first;
	first = first.next;
	first.next.next = null;

This is the correct answer.

first a 🡒 b | null (start)

b 🡒 first: a 🡒 (b)

first: b 🡒 a 🡒 (b)

first: b 🡒 a | null (final)Thi

* 1. Link temp = first;
	first = temp.next;
	first.next = temp;
	temp = null;

first: a 🡒 b | null (start)

first, temp: a 🡒 b | null

temp: a 🡒 first: b | null

first: b 🡒 temp: a 🡒 (b)

first: b 🡒 a 🡒 (b) (final)

- not a correct answer, results in a cyclic list.

* 1. Link temp = first.next;
	temp.next = first;
	first.next = null;

first: a 🡒 b | null (start)

first, temp: a 🡒 b | null

first, temp: a 🡒 b 🡒 (a)

first, temp: a | null b

- not a correct answer, it shortens the list to one element.

1. A Cyclic Linked Implementation of a Queue
* An (unbounded) queue can be implemented cyclically based on a linked representation. In this case, rather than referencing “null”, the successor of the last item in the queue references the beginning of the queue. While this is not necessary to prevent memory erosion, it does mean that rather than having two references to the beginning and the end of the queue, only a *single* reference is needed.
* Write a cyclic linked implementation of a queue called QueueLinked using this approach.
* Your queue ADT must implement the QueueADT interface.
* Fully document your code.

This exercise is to give you practice writing code – we do not provide a smaple answer.

If you *have* attempted the code, and wish to get feedback on your attempt, feel free to email the lecturer.

1. A double-ended queue (*deque*) of characters differs from a standard queue in that it allows objects to be added and deleted from both ends of the queue. Contrast this to a standard queue, where objects can only be added to the end of the queue and removed from the front.
* Write a singly linked-list implementation of the deque ADT in Java.
* Your implementation should contain the following methods:
	+ DequeCharCyclic(s): create an empty deque of size s.
	+ isEmpty(): return true iff the deque is empty, false otherwise.
	+ isFull(): return true iff the deque is full, false otherwise.
	+ pushLeft(c): add character c as the left-most character in the deque, or throw an Overflow exception if the deque is full.
	+ pushRight(c): add character c as the right-most character in the deque, or throw an Overflow exception if the deque is full.
	+ peekLeft(): return the left-most character in the deque, or throw an Underflow exception if the deque is empty.
	+ peekRight(): return the right-most character in the deque, or throw an Underflow exception if the deque is empty.
	+ popLeft(): remove and return the left-most character in the deque, or throw an Underflow exception if the deque is empty.
	+ popRight(): remove and return the right-most character in the deque, or throw an Underflow exception if the deque is empty.

This exercise is to give you practice writing code – we do not provide a smaple answer.

If you *have* attempted the code, and wish to get feedback on your attempt, feel free to email the lecturer.

1. **Challenge:** See extra questions on dequeues at <http://teaching.csse.uwa.edu.au/units/CITS2200/Tutorials/tutorial04.html>

## Trees



1. (source: Weiss 18.1) For the tree shown above determine:
	1. Which node is the root

answer: A

* 1. Which nodes are leaves

answer: G, H, I, L, M, K

* 1. What is the tree’s height

answer: 4

* 1. What are the results of traversing the tree using
		1. preorder

answer: A B D G H E I J L M C F K

* + 1. postorder

answer: G H D I L M J E B K F C A

* + 1. inorder

answer: G D H B I E L J M A C F K

* + 1. level order

answer: A B C D E F G H I J K L M

(levels shown here as extra spaces)

## Binary trees

1. State whether each of the following statements is TRUE or FALSE. Why?
	1. The height of a binary tree is $O\left(logn\right)$

FALSE – the tree could be unbalanced with one side longer than log n. Worst case tree height is n.

* 1. The proper descendants of a node’s ancestors are also descendants of that node.

FALSE eg above A and B are ancestors of D, B is a proper descendent of A but is not a descendent of D

* 1. A binary tree always has more external nodes than internal nodes.

FALSE. Example above has 9 internal nodes and 4 external nodes (leaves) from13 nodes in total

* 1. Every tree has a root node.

TRUE (we assume “null” alone does not represent a tree).

1. (Source: Weiss 18.9) Write efficient methods (and give their Big-Oh running times) that take a reference to a binary tree root t and compute
	1. The number of leaves in t
	2. The number of nodes in t that contain one non-null child
	3. The number of nodes in t that contain two non-null children

Answer: Code will be similar to the following:



1. (Source: Weiss 18.10) Suppose a binary tree stores ints. Write efficient methods (and give their Big-Oh running times) that take a reference to a binary tree root t and compute
	1. The number of even data items
	2. The sum of all the items in the tree

Answer: Code will be similar to the following:

**public** **static** **int** treeSum(BinaryTreeNodeHomework<Integer> t)

 {

 **if** (t == **null**) {

 **return** 0;

 }

 **return** (t.value + *treeSum*(t.left) + *treeSum*(t.right));

 }

 **public** **static** **int** evenNodes(BinaryTreeNodeHomework<Integer> t)

 {

 **int** e;

 **if** (t == **null**) {

 **return** 0;

 }

 //check if the current node is even or not

 **if** (t.value % 2 == 0) { e = 1; } **else** { e = 0; }

 //then count up even nodes in the rest of the tree

 **return** (e + *evenNodes*(t.left) + *evenNodes*(t.right));

}

## Binary search trees

1. (Source: Weiss 19.1) Show the result of inserting 3, 1, 4, 6, 9, 2, 5, and 7 in an initially empty binary search tree. Then show the result of deleting the root.

Answer:

When deleting the root: we replace the root with the largest node from the left hand side, in this case 2.

1. (Source: Weiss 19.2) Draw all binary search trees that can result from inserting permutations of 1, 2, 3, and 4. How many trees are there? What are the probabilities of each tree’s occurring if all permutations are equally likely?

There are 24 permutations (=different orderings) of these numbers (4 x 3 x 2 x 1 = 24). Each insert order (permutation) gives rise to a particular tree, but sometimes the same tree occurs for 2 different permutations. For example, inserting 3124 and 3241 and 3214 all make the same tree.

In parentheses below is listed the number of possible insertion sequences that give the same tree.

1234 (1), 1243 (1), 1324 (2), 1423 (1), 1432 (1),
2143 (3), 2134 (3),
3214 (3), 3124 (3),
4321 (1), 4312 (1), 4213 (2), 4123 (1), 4132 (1).

14 trees are possible, their probabilities are calculated from the number in brackets as eg, 1/24, 2/24 or 3/24

1. (Source: Weiss 19.15) Implement the BinaryTree methods find, findMin, and findMax recursively.

Note this is a Binary Tree but not (necessarily) a Binary *Search* Tree so you can’t rely on the nodes being in L < P < R order and have to search the whole tree.

 findMin is shown here. findMax and find are similar.

 /\*\*

 \* Compare two tree nodes and return the one with the min value

 \* **@param** n1 binary tree node 1

 \* **@param** n2 binary tree node 2

 \* **@return** the node with the minimum value

 \*/

 **public** **static** BinaryTreeNodeHomework<Integer> minPair(

 BinaryTreeNodeHomework<Integer> n1,

 BinaryTreeNodeHomework<Integer> n2 ) {

 **if** (n1==**null** && n2==**null**) {

 **throw** **new** IllegalArgumentException("at least one node for minPair must be non null");

 }

 **if** (n1==**null**) { **return** n2; }

 **if** (n2==**null**) { **return** n1; }

 **if** (n1.value <= n2.value) {

 **return** n1;

 } **else** {

 **return** n2;

 }

 }