Algorithms & Data Structures
Exercise Sheets

Weeks 7-8: Heaps and Binary Search Trees

Exercises

(1) (Week 7) Using pen and paper, illustrate how a *min-heap* is built from the array

$$[59, 44, 79, 17, 54, 32, 31, 12, 7, 4, 1]$$

using *heapify*. You must illustrate all steps of the heapify process. Then illustrate first what happens if *remove* is called on the resulting heap and then what happens if *insert(2)* is called on the heap (before the remove).

(2) (Week 7) Modify the heap implementation we saw on class to implement a *max-heap* in which the maximum element is stored in the root. Implement the `priority_queue` interface discussed in class using the max-heap, and provide testing code.

(3) (Week 7) What is the running time of *heapsort* on an array A of length N that is already sorted in increasing order? What about decreasing order?

(4) (Week 8) Using pen and paper, illustrate how an AVL tree is build with the following elements $[10, 20, 15, 25, 30, 16, 18, 19]$ (i.e.10 is inserted first, then 20 ...). You must illustrate all rotations etc. Then illustrate what happens when 30 is deleted and then 18

(5) (Week 8) A node in a binary tree is an only-child if it has a parent node but no sibling node (Note: The root does **not** qualify as an only child).
The *loneliness-ratio* (LR) of a given binary tree T is defined as the following ratio:

$$LR(T) = \frac{\text{The number of nodes in T that are only children}}{\text{The number of nodes in T}} \tag{1}$$

(a) Argue that for any nonempty AVL tree T, we have that $LR(T) \leq 1/2$

(b) Is it true for any binary tree T,that if $LR(T) \leq 1/2$ then $height(T) = log(n)$?

(6) (Week 8) The `Set` is an ordered container that does not allow duplicates. Write an implementation of the `Set` class using the `BinarySearchTree` implementation we saw in class as the internal data structure. Add to each node a link to the parent node to make traversal easier. The supported operations are:

```
iterator insert(const Object& x);
iterator find(const Object& x ) const;
iterator erase(iterator& itr);
```

The `insert` function returns an iterator to that either represents the newly inserted item or the existing duplicate item that is already stored in the container. For searching, the `find` routine returns an iterator representing the location of the item (or the endmarker if the search fails). The `erase` function removes the object at the position given by the iterator, returns an iterator representing the element that followed `itr` immediately prior to the call to erase, and invalidates `itr`.

(7) (Week 8) Design and implement a linear-time algorithm that verifies that the height information in an AVL tree is correct i.e. the balance property is in order for all nodes.