1 Vectors and amortized analysis

1. Consider a vector class which has indexing, the usual push back and a `pop_back` operation, which causes the last element to be removed from the vector. Show that any sequence of n operations can be implemented in $O(n)$ time total, i.e. such that the amortized complexity of each operation is $O(1)$. You must ensure that at all times the allocated memory must be at most some α times the memory in use. You can chose any constant α. Hint: $\alpha = 2$ will not work if you double the size whenever a push_back causes the vector size to go above the currently allocated size.

2. Suppose I use an array of length n of `int`s to represent an n bit number. Thus each element of the array will only take values 0 or 1, and the ith element will represent the ith least significant bit. Initially the number is initialized to all zeroes. Two operations are allowed: `increment` which increments the value stored, and `print` which prints the value. Note that when you increment, there will be carries and in general many bits will change. Show that starting at zero if you perform N increment operations, the total work you do is $O(N)$, i.e. the amortized cost of each `increment` operation is $O(1)$. What is the maximum possible cost (time taken) for an `increment` operation?

2 Binary trees and binary search trees

Most of the problems below are about binary trees, but you are expected to be able to solve them.

The description will often say, input is "a tree" – which really means the input is a pointer to the root of the tree. Also, unless specified, we will mean binary trees, i.e. trees in which each vertex has at most two children.

Most problems will be solved using an appropriate recursive function: to solve the problem on the tree recurse on one or both the subtrees. If you indeed use such recursive functions in your solution, you should state precisely what your function does, when it is called on any node, not just the root. You should write down preconditions for the function to work correctly, if any such preconditions are necessary.

In the exercises below, n will denote the number of nodes in the tree, and h the height of the tree.

Some of the exercises ask you to add members to the struct `Node`. If need be you may yourself add more members if your computation needs it.

1. Write a function which given a tree, returns n the number of nodes. Remember that a node is a leaf if it has no children. Your function should run in time $O(n)$.

2. Write a function which given a tree, returns the number of leaves in it. Remember that a node is a leaf if it has no children. Your function should run in time $O(n)$.

3. Write a function which given a tree returns a vector `v` such that `v[i]` gives the number of nodes at a distance `i` from the root. You might find it useful to have two function, one function which the rest of the world calls, but which in turn calls a recursive function that does all the work. Be careful in specifying which parameters are passed by reference, if any. Remember that if you pass a vector by value, it is copied and that takes time proportional to its length. Your function should run in time $O(n)$.

4. A function which recurses on both the subtrees of a tree will visit every vertex, and is said to *traverse* the tree. Here is the simplest such function.

```
void traverse(Node* r){
  if(r == NULL) return;
  traverse(r->left);
  traverse(r->right);
}
```

   Of course, to be useful, you will need to put code to do something on the nodes perhaps, or do some computation and maybe return a result.

For now we use the `traverse` function to define 3 different ways to number the n vertices with distinct numbers between 0 and $n-1$. In a preorder numbering, node u receives a smaller number than node v if `traverse` is called before on u than on v. In inorder numbering, node u must receive a larger number than the numbers assigned to all the nodes in its left subtree, and smaller than those in the right. In postorder numbering, node u receives a smaller number than node v if the call on u returns before the call on v returns. This is just a generalization of what you saw in exercises 8, 9, 10 of text1.

Suppose struct `Node` has additional integer members `preorder, inorder, postorder`. Write functions that take the tree as argument and sets the values of the members above to reflect the numberings defined above.

5. Suppose we are given the preorder, inorder, and postorder numbers for each node u. Show that in $O(1)$ time we can compute (a) number of nodes in the subtree rooted at u, (b) the depth of u, i.e. the length of the path from the root to u, (c) given two nodes u, v determine whether u is an ancestor of v.

6. Write a program which draws the tree such that the x coordinate of each node is proportional to the inorder number, and y coordinate proportional to the depth. Your function should run in time $O(n)$.

7. Let $R_n(i)$ denote the number obtained by reversing the binary representation of the n bit number i. For example, $R_4(12)$ is 3, because the binary representations of 12, 3 are 1100, 0011, and thus reverses of each other. Suppose we have an initially empty search tree into which in step i we insert $R_n(i)$ for $i = 1$ to $i = 2^n - 1$. What is the height of this tree?

8. Suppose n distinct numbers are inserted into a binary search tree in a randomly chosen order. What is the probability that the tree will have height n?

9. Suppose we add an integer member `size` to `struct Node`. We would like to have the invariant that `v.size` is equal to the size of the subtree beneath node `v`. Show how the `insert` function update `v.size` of relevant nodes suitably so that the invariant is maintained. The insert function should run in time $O(h)$ as before, where h is the height of the tree.

10. Using the member `size` as described above, show how to find the rth smallest key in the set, given r and the root of the tree. Your function should run in time $O(h)$.

The next set of problems you will be able to solve after the lecture on Wednesday.

1. Give an algorithm that prints the elements in the set falling in a given interval $[a, b]$. Your algorithm should run in time $O(h + m)$, where m is the number of elements in the set that are in the given interval.

2. Suppose you are given a possibly unbalanced binary search tree on n nodes. Show that in time $O(n)$ you can construct a binary search tree with the same set of keys having height $\log n$.