Assignment No: 03

Name: S. M. Salah Uddin Kadir **ID: 1800503**

---

1. Use a stack to eliminate recursion from the following procedure.

```
Procedure reverse (L: list){
      X: element_type
      If not list_empty(L) then {
         X = list_retrieve(first(L), L)
         Delete (list_first(L), L)
         reverse(L)
         list_insert(x, list_end(L), L)
      }
}
```

<u>Solution:</u>

```
Procedure reverse (L: list){
      X: element_type
      S:Stack
      while not list_empty(L) {
         X = list_retrieve(first(L), L)
         Delete (list_first(L), L)
         Push(X, S)
       }

       while not Empty(S) {
         X = Top(S)
         Pop(S)
         list_insert(X, list_end(L), L)
      }
}
```

2. Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section. (Hint: List the possibilities when n = 3.)

<u>Solution:</u>
Suppose we have 3 keys {1, 2, 3}.

For the 1st case,
We can create five different binary search trees using these three keys.

```
1              1            2              3              3
 \              \          / \            /              /
  2              3        1   3          1              2
   \            /                         \            /
    3          2                           2          1
```

As each binary search tree is equally likely to be chosen, so, the probability for each tree to be chosen is 1/5.

For the 2nd case,
If we choose keys randomly to build the binary tree, then we can choose keys in six different ways. The sequences will be,
{1, 2, 3},
{1, 3, 2},
{2, 1, 3},
{2, 3, 1},
{3, 1, 2}, and
{3, 2, 1}
So, the probability for each tree to be chosen is 1/6.

So, the probability distribution for randomly chosen binary search tree is different with the probability distribution for randomly built binary search tree.

3. Binary search trees with equal keys
Equal keys pose a problem for the implementation of binary search trees.

a) What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?

<u>Solution:</u>
The condition to insert items into binary tree,

```
if z.key < x.key
        x = x.left
else
        x = x.right
```

Now, if we add only identical elements, then the item will be added only to the rightmost leaf, and will build the right skewed tree.

So, to insert an element to the tree will take Θ(n) time, and to insert n items into the tree will take Θ(n^2) time.

We can write it as, $\sum_{i=1}^{n} i \in \theta(n^2)$

3. We propose to improve TREE-INSERT by testing before line 5 to determine whether z.key = x.key and by testing before line 11 to determine whether z.key = y.key.

If equality holds, we implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, in which we compare the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)

b) Keep a boolean flag x.b at node x, and set x to either x.left or x.right based on the value of x.b, which alternates between FALSE and TRUE each time we visit x while inserting a node with the same key as x.

__**Solution:**__
If we create a binary tree by altering the identical elements, then it will create a balanced binary tree. The height of balanced binary tree is log (n).

So, to insert an element to the tree will take Θ(lg(n)) time, and to insert n items into the tree will take Θ(nlg(n)) time.

We can write it as, $\sum_{i=1}^{n} \lg(n) \in \theta(nlg(n))$

c) Keep a list of nodes with equal keys at x, and insert z into the list.

__**Solution:**__
This will take linear time. As all keys are equal, so, keeping a list of nodes with equal keys at x, the height of the tree will be 0. So, a single insertion into a list can be done in constant time.

d) Randomly set x to either x.left or x.right. (Give the worst-case performance and informally derive the expected running time.)

__**Solution:**__
After choosing items randomly, then, in a worst case, it may create either left skewed tree or right skewed tree. In this case, the time complexity will be O(n^2).

If we think, the probability of picking elements for left is nearly half, and for the right is remaining, then the tree will be almost balanced. So, the depth of the tree will be approximately lg(n), and the expected running time will be nlg(n).

[4] Suppose that we use an open-address hashing table of size m to store n ≤ m/2 items. Assuming uniform hashing, show that for i = 1, 2, …, n, the probability is at most 2^{-k} that the ith insertion requires strictly more than k probes.

<u>Solution:</u>

The size of the hashing table is m, and number of items need to store is n, where n <= m/2. The index for each probe is computed uniformly from among all the possible indices. Since we have n <= m/2, we know that there are at least half of the indices empty at any stage.

So, for more than k probes to be required, we would need that in each of k first probes, we probed a vertex that already had an entry, this has probability less than 1/2. So, the probability of it happening each time is at most 2^{-k}.

[5] Let p1 >= p2 >= … >= pn be the access frequencies of n names in a sequential list (implemented as a table). (a) Assuming that only successful searches occur, prove that among all n! permutations of names in the table those with access frequencies in monotonic non-increasing order have minimum average search time. (b) If unsuccessful searches are included, is the statement in (a) true? Justify your answer.

<u>Solution:</u>

(a)

In case of successful searching, the average searching cost of a sequential list of n items is,

C = (1/total number of accessing frequency of the list) $*$ $\sum_{i=1}^{n}(\text{i } * \text{ accessing frequency of the ith item})$

Suppose, we have a list of 5 items

A->B->C->D->E

The corresponding access frequency of those items are,

3, 4, 1, 7, and 2

So, the total access cost will be for this list,

3*1 + 4*2 + 1*3 + 7*4 + 2*5 = 52

And average cost will be, 52/17 = 3.059

Now, if we sort the list based on frequency,

D->B->A->E->C

So, the total access cost will be for the sorted list,

7*1 + 4*2 + 3*3 + 2*4 + 1*5 = 37

And the average cost will be, 37/17 = 2.176

So, after sorting the list based on frequency, the average cost has been decreased.

Now, if we swap the 1st item with the 2nd item from the sorted list, then it will add by 3 with the total cost.
Again, if we swap 3rd item with the 5th item from the sorted list, then it will add by 4 with the total cost.

Actually, any position changing of the items of the sorted list will increase the total searching cost. So, we get the minimal average searching cost after sorting the list in non-increasing order based on accessing frequency of an item.

(b)
After adding unsuccessful searching costs, the first statement still will be true.

As the total searching cost of the sorted list will be always less than the total searching cost of the unsorted list even though after adding the unsuccessful searching cost, so we will get always minimal average searching cost in the sorted list.