Assignment 5: Sample Solutions

1. (a) The following procedure searches for key x in a B-S array A of size $2^{k+1} - 1$:

Algorithm 1 MEMBER(x, A, k)

```
1: found ← FALSE; j ← k
2: while j ≥ 0 and not(found) do
3:   if A[2^j] ≠ ∞ then
4:     binary search in A[2^j, 2^{j+1} − 1] for key x
5:     if search is successful then
6:       found ← TRUE
7:     end if
8:   end if
9:   j ← j − 1
10: end while
11: return found
```

The idea here is to look for key x in all of the non-null blocks, *in order of decreasing size*, using a binary search within each block.

(b) In the worst case for a successful MEMBER query (or in *all* cases for an unsuccessful MEMBER query) all of the non-null blocks are searched at a total cost of $\sum_{0 \le j \le k}(1 + \lg 2^j) = \sum_{0 \le j \le k}(1 + j) = \Theta(k^2)$. Since $k = \Theta(\lg n)$, this is $\Theta((\lg n)^2)$.

For the average case we compute the total running time for all possible successful MEMBER queries (i.e all elements of S) and divide by n.

As suggested, we assume for simplicity that the cost of performing binary search on block i is $b_i i$. If the query corresponds to one of the 2^j elements in the non-null block j, our algorithm searches all larger blocks first. Thus the total cost can be expressed as $\sum_{0 \le j \le k} b_j 2^j \sum_{j \le i \le k} b_i i$. Since $b_i \le 1$, this cost is bounded above by :

$$\begin{aligned}\sum_{0\le j\le k}(k(k-j+1))2^j &= k2^{k+1}\sum_{0\le j\le k}\frac{k-j+1}{2^{k-j+1}}\\ &= k2^{k+1}\sum_{1\le s\le k+1}\frac{s}{2^s}\\ &< k2^{k+2} \qquad\qquad (\text{since } \sum_{s\ge 1}\frac{s}{2^s}=2)\end{aligned}$$

Thus, the total cost is $O(n \lg n)$ and the average cost, over all possible successful queries, is $O(\lg n)$.

(c) We can implement INSERT by successive merge operations, in order of increasing block size, which reflects the successive "carry" steps when incrementing a binary counter. The following procedure inserts a new element x into an existing B-S array:

Algorithm 2 INSERT(x, A)

```
1: */ find index i of the first (smallest) null block
2: j ← 1; i ← 0
3: while A[j] ≠ ∞ do
4:    j ← j + j; i ← i + 1
5: end while
6: */ now merge all smaller blocks, plus the new element x, into block i
7: A[j] ← x
8: s ← 0
9: while s < i do
10:   merge A[2^s : 2^{s+1} − 1] with A[j : j + 2^s − 1] into A[j : j + 2^{s+1} − 1]
11:   A[2^s : 2^{s+1} − 1] ← ∞
12:   s ← s + 1
13: end while
```

(d) In the worst case we need to merge all of the elements into one new block. Since merging cost is linear in the total size of the blocks being merged, and the blocks double in size at each step, the total cost for merging is $\Theta(n)$ in the worst case.

It is straightforward to do an accounting-type amortized analysis to show that the amortized cost of INSERT (over a sequence of n INSERT operations, starting from the empty B-S array) os $O(\lg n)$. Simply assign each element $\lg n$ tokens, each of which pays for a unit of work in a single merge step. Since individual elements y can participate in at at most $\lg n$ merges in total (each merge doubles the size of the block containing y) these tokens suffice to pay for all of the required work.

(e) Since the most recently inserted elements occupy the smallest blocks, it makes sense–if our objective is to reduce the search cost for the most recently inserted elements–to modify our search to consider blocks in order of *increasing* size. In this case, using our standard B-S array, we must perform binary search on all of the non-empty blocks up to and including the block containing the query value x.

(f) Assuming that x is the i-th most recently inserted element, the worst case occurs when $i \in [2^j,\ 2^{j+1} - 1]$ and n (the total number of keys) is of the form $n = 2^k + 2^j - 1$. In this case, blocks 0 through $j - 1$ are all non-empty, but x lies in block k. The total cost in this case is $\Theta(k + j^2) = \Theta(\lg n + (\lg i)^2)$.

(g) The idea here is to keep an array with *two* blocks of each (power of 2) size. (Think of splitting the blocks of a standard B-S array in half.) It is easy to maintain the invariant that at least one of the two blocks of each size , up to the size of the largest non-empty block, is non-empty. This can be viewed as a different kind of binary representation of n, where $n = \sum_{0 \le i \le t} c_i 2^i$ and each $c_i \in \{1, 2\}$.
With this modification the i-th most recently inserted element must occupy a block of size no more than $2^{\lfloor \lg i \rfloor}$. Hence the cost of a MEMBER query for the i-th most recently inserted element is at most $O((\lg i)^2)$, independent of n. Note that the asymptotic cost of INSERT does not change with this modification.

2. (a) If G is represented as an adjacency list, it is easy to check if $|E| = |V| - 1$, a *necessary* condition for G to be a tree, in $O(V)$ time. If G passes that test, then it remains to check that G is connected, which is straightforward to do in $O(V + E)(= O(V))$ time, using depth-first or breadth-first search.

(b) Consider the adversary strategy that responds $a_{1,j} = a_{j,1} = 1$, for $j = 2, \ldots, |V|$ and $a_{i,j} = 0$, otherwise. Faced with this adversary any algorithm must look at every entry (up to symmetry) in the adjacency matrix. Otherwise, if the adversary changes any one un-probed entry in the matrix, the corresponding graph changes from being a tree to not being a tree.

3. (a) Suppose that the edge colours are labeled $1, \ldots c$. We can transform the graph G by replacing each vertex v by $2c$ vertices, v_j^{in}, v_j^{out}, for $1 \le j \le c$, and each edge (u, v) with colour j by the edge (u_j^{out}, v_j^{in}) with weight 0. In addition, we add edges (u_j^{in}, u_j^{out}) with weight 0, for $1 \le j \le c$, and edges (u_j^{in}, u_k^{out}) with weight 1, for all $j \ne k$, $1 \le j, k \le c$. Finally, we add edges of weight 0 from a new vertex s^* to all vertices s_j^{out}, and edges of weight 0 from all vertices t_j^{in} to a new vertex t^*, for $1 \le j \le c$.
If G has n vertices and m edges then the new graph G' has $2cn$ vertices and $m + c^2$ edges.

In G' there is a path of weight w from s^* to t^* if and only if there is a path in G from s to t with exactly w colour transitions. (Each edge of weight 1 in G' corresponds to a colour transition in G, and vice versa.)

(b) As we will see later, the *minimum colour path problem* is **NP**-hard. No truly efficient general algorithm is known for this problem: the only available approaches involve essentially brute-force: try all possible paths in G from s to t. One could imagine trying all possible values of k and all subsets of colours of size k and asking if a path exists using only colours in that subset. What would be the cost of this approach as a function of n and c?