Data Structures and Algorithms – (COMP SCI 2C03)
Winter, 2021
Assignment-II

Due at 11:59pm on March 15th, 2021

- **No late assignment accepted.**
- Make sure to submit a version of your assignment ahead of time to avoid last minute uploading issues.
- Submit one assignment solution as a PDF file per group on Avenue.
- If the solution submitted by two groups is the same. Both teams will get a zero mark on the assignment.
- Present your algorithms in Java or Pseudocode (Pseudocode is preferred).
- It is advisable to start your assignment early.

This assignment consists of 8 questions, and is worth 55 marks.

1. Modify BinarySearchST (Algorithm 3.2 in textbook on page 379) so that inserting a key that is larger than all keys in the table takes constant time (so that building a table by calling put() for keys that are in order takes linear time). [6 marks]
   Answer: See Figure 1 and 2.

2. Given a standard binary search tree (BST) (where each key is greater than the keys in its left subtree and smaller than the keys in its right subtree), design a linear-time algorithm to transform it into a reverse BST (where each key is smaller than the keys in its left subtree and

greater than the keys in its right subtree). The resulting tree shape should be symmetric to the original one. It is not required of you to output the tree from your code (if you implement it in JAVA). Marks will be given based on the logic and correctness of the algorithm. [7 marks]

Answer:

```
procedure INVERT_TREE(root) {
    if {(root == NULL)} then return NULL
    left = invertTree(root.left)
    right = invertTree(root.right)
    root.left = right
    root.right = left
    return root
}

procedure main() {
    root = Node(5)
    root.left = Node(4)
    root.right = Node(6)
    root.left.left = Node(3)
    root.left.right = Node(8)
    root.right.left = Node(9)
    root.right.right = Node(7)
    INVERT_TREE(root)
}

After INVERT_TREE(root) is called, the tree should be as follows:
root.val = 5
root.left.val = 6
root.right.val = 4
root.left.left.val = 8
root.left.right.val = 3
root.right.left.val = 7
root.right.right.val = 9
```

3. Give a non-recursive algorithm that performs an inorder tree walk for a tree containing n nodes. The algorithm should run in $O(n)$ time. [6

marks]

Answer:

```
procedure INORDER_TRAVERSAL(root) {
    S = NULL
    curr = root
    while {(curr != NULL || !S.isEmpty())} do {
          while {(curr != NULL)} do {
               S.push(curr)
               curr = curr.left
          }
          curr = S.pop()
          VISIT(curr)
          curr = curr.right
    }
}
```

In this case, VISIT() can be any method of accessing the value of curr. E.g. print(curr.val).

The algorithm pushes and pops only the left nodes on to the stack once. Hence the inner while loop runs for at most n times. Furthermore, the outer while loop at each iteration prints a node, and hence is executed n times. Therefore, the algorithm runs in $O(n)$ time.

4. What is the largest possible number of internal nodes in a red-black tree with black height k? What is the smallest possible number? [5 marks]

   Answer: When all the links in the left leaning red-black tree (LLRBT) are black, the LLRBT is the same as a 2-3 tree consisting of only 2-node nodes with the tree height = black height = k. Thus, the LLRBT is a balanced binary tree having height k. The number of nodes in such a tree = $2^k - 1$. Hence, the smallest possible number of internal nodes in a LLRBT of black height k is $2^k - 1$.

   ~~The LLRBT of black height k has max. height = $2k$, when it has alternate red and black links in at least one path from the root to a leaf node. A balanced LLRBT with height $2k$ would have the maximum number of internal nodes = $2^{2k} - 1$. Therefore, the largest possible number of internal nodes in a red-black tree with black height k is $2^{2k} - 1$.~~

The LLRBT of black height k, has maximum internal nodes, when it is equivalent to a 2-3 tree of height k consisting of all 3-nodes. Each 3-node that is an internal node in the 2-3 tree will contribute to 2 internal nodes in its equivalent LLRBT, and each leaf node in the 2-3 tree will contribute to only one internal node in its equivalent LLRBT. Hence, the maximum number of internal nodes in an LLRBT with black height $k = 2(\frac{3^k-1}{2}) + 3^k = 2.3^k - 1$.

6. (a) Modify LinearProbingHashST (given on page 470 of the textbook) to use a second hash function to define the probe sequence. Specifically, first replace (i + 1) % M (both occurrences) by (i + k) % M where k is a nonzero key-dependent integer that is relatively prime to M (it is easier to choose M as prime) - $(i + k)\%M$ would be your first hash function $h_1(k)$. Then, device a second 'good' hash function $h_2(k)$ to identify successive probe positions. You are not required to work on this question in JAVA. Simply modifying the `hash`, `put` and the `get` functions to accommodate double hashing will suffice. However, for those interested in implementing it in JAVA, the code for LinearProbingHashST can be found at - `https://algs4.cs.princeton.edu/34hash/LinearProbingHashST.java.html` [7 marks]

   Alternate version of the question: Modify LinearProbingHashST (given on page 470 of the textbook) to use a second hash function to define the probe sequence. Specifically, first replace (i + 1) % M (both occurrences) by $(h_1(k) + ih_2(k))\%M$ where k is a nonzero key-dependent integer that is relatively prime to M (it is easier to choose M as prime). $h_1(k)$ would be your first hash function. You may choose to use the $hash()$ function defined on page 470 of the textbook as $h_1(k)$. You are then required to device a second 'good' hash function $h_2(k)$ to identify successive probe positions. You are not required to work on this question in JAVA. Simply modifying the `hash`, `put` and the `get` functions to accommodate double hashing will suffice.

   Answer: See Figure 3 and 4.

   (b) Give a trace of the process of inserting the keys E A S Y Q U T I O N in that order into an initially empty table of size M =11, using the hash functions described in 6(a). For this question, since the keys are drawn from the English alphabet, you may choose a map-

ping from the set of English Alphabets to natural numbers and then insert the keys in the hash table. For instance, you choose the following mapping $MAP : \{A, B, \ldots, Y, Z\} \rightarrow \{1, 2, \ldots 25, 26\}$. Alternatively, it is also acceptable to use the hashcode() method for the keys provided in JAVA (`key.hashCode()`) as shown on page 470.[5 marks]

Answer: See Figure 5, 6 and 7.

7. Prove that every connected graph (with at least two vertices) has a vertex whose removal (including all adjacent edges) will not disconnect the graph, and write a DFS method that finds such a vertex. Hint : Consider a vertex whose adjacent vertices are all marked. [6 marks]

Proof: The proof is by induction.

Base case: G has two connected vertices v and w. When either of vertices are deleted (along with the edge connecting them), G is left with only one vertex v (or w). A graph with only one vertex remains connected. Therefore the base case is satisfied.

Inductive case: Suppose G has k connected vertices, and the graph has a vertex v whose removal (including all adjacent edges) will not disconnect the graph. From G, we can construct another graph G' by adding a vertex w and an edge connecting w with any other vertex $\neq w$ in G'. If w is connected to v, then w is the vertex whose removal will not disconnect G' (as the resulting graph after the removal would be G and by induction hypothesis, it is connected). If w is connected to any other vertex, then v is the vertex whose removal will not disconnect G'.

Therefore we have shown that that every connected graph, with at least two vertices, has a vertex whose removal (including all adjacent edges) will not disconnect the graph. □

Algorithm:

```
Initialize marked[0..V-1] to false
flag = true                ▷ flag is a switch we use to identify the vertex
vertex ▷ vertex – is the vertex whose removal does not disconnect G
procedure DepthFirstPaths(G, s)
    dfs(G, s)
procedure DFS_Indentify_Vertex(G, v, flag)
    marked[v] = true
    for each w ∈ G.adj(v) do
```

```
        if !marked[w] then
            flag = false
            dfs(G, w)
    if flag then vertex = v
```

8. Explain why the following algorithm does not necessarily produce a topological order: Run BFS, and label the vertices by increasing distance to their respective source. [6 marks]

   Answer: For this question, it suffices to show a counter example and explain how the algorithm produces incorrect result. Consider the directed graph given in Figure 8. The topological order for the graph is A B D C. When we run BFS with A as the source, since it is at a distance 0 from it self, we label it as zero. After A is evaluated, the vertices in its adjacency list; that is B and C are marked and evaluated next in that order. Hence B is labelled '1' and C is labelled '2' (you could label both as 1, however this way there is a defined ordering). Similarly D and E are labelled 3 and 4 respectively. Hence the vertex order generated by this algorithm is A B C D. This is clearly not a topological order as C appears before D, when D should appear before C due the presence of the directed edge $D \rightarrow C$. Therefore this algorithm does not work. In fact, when you have directed edges from the vertices away from the source, to the vertices closer to the source, the algorithm fails.

```
public class BinarySearchST<Key extends Comparable<Key>,
Value> {
    private Key[] keys;
    private Value[] vals;
    private int N=0;

    public BinarySearchST(int capacity) {
        keys = (Key[]) new Comparable[capacity];
        vals = (Value[]) new Object[capacity];
    }

    public int size() {
        return N;
    }

    public Value get(Key key) {
        if (isEmpty()) return null;
        int i = rank(key
        if (i < N && keys[i].compareTo(key) == 0) return
vals[i];
        return null;
    }

    public int rank(Key key) {
        int lo = 0, hi = n-1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int cmp = key.compareTo(keys[mid]);
            if      (cmp < 0) hi = mid - 1;
            else if (cmp > 0) lo = mid + 1;
            else return mid;
```

Figure 1: Solution to Q1 (Part1)

```
        }
        return lo;
    }


    public void put(Key key, Value val)  {
        if (n == keys.length) resize(2*keys.length);

        if (keys[N-1].compareTo(key) == 0) {
            vals[N-1] = val;
            return;
        }

        if (keys[N-1].compareTo(key) < 0) {
            keys[N] = key;
            vals[N] = val;
            N++;
            return;
        }

        int i = rank(key);

        if (i < n && keys[i].compareTo(key) == 0) {
            vals[i] = val;
            return;
        }

        for (int j = n; j > i; j--)  {
            keys[j] = keys[j-1];
            vals[j] = vals[j-1];
        }
        keys[i] = key;
        vals[i] = val;
        N++;
    }
}
```

Figure 2: Solution to Q1 (Part2)

```
  private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % M;
    }

    // here I choose prime number 7 and use it to compute
the second hash
    private int hash2(Key key) {
        return 7 - (key.hashCode() & 0x7fffffff) % 7;
    }

    public void put(Key key, Value val) {
        if (N >= M / 2) resize(2 * M);
        int k = hash2(key);
        int curr = hash(key)
        for (int i = 0; keys[curr] != null; i++) {
            curr = (hash(key) + i * (hash2(key))) % M;
            if (keys[curr].equals(key)) {
```

Figure 3: Solution to Q6a (Part1)

```
                vals[curr] = val;
                return;
            }
        }

        keys[curr] = key;
        vals[curr] = val;
        N++;
    }

    public Value get(Key key) {
        int curr = hash(key)
        int k = hash2(key);
        for (int i = 0; keys[curr] != null; i++) {
            curr = (hash(key) + i * (hash2(key))) % M;
            if (keys[curr].equals(key))
                return vals[curr];
        }

        return null;
    }
```

Figure 4: Solution to Q6a (Part2)

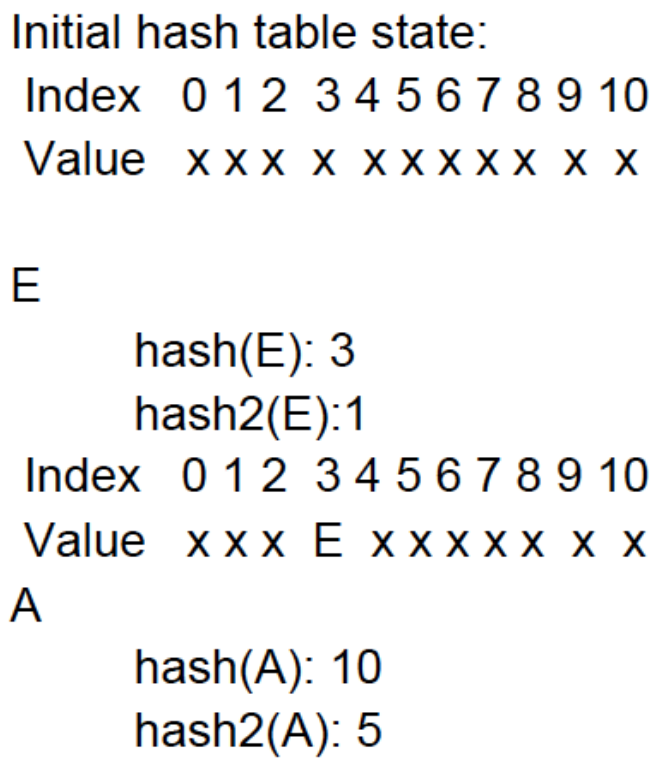


Figure 5: Solution to Q6a (Part1)

Index 0 1 2 3 4 5 6 7 8 9 10
Value x x x E x x x x x x A
S
hash(S): 6
hash2(S): 1

Index 0 1 2 3 4 5 6 7 8 9 10
Value x x x E x x S x x x A
Y
hash(Y): 1
hash2(Y): 2
Index 0 1 2 3 4 5 6 7 8 9 10
Value x Y x E x x S x x x A
Q
hash(Q): 4
hash2(Q): 3
Index 0 1 2 3 4 5 6 7 8 9 10
Value x Y x E Q x S x x x A
U
hash(U): 8
hash2(U): 6

Index 0 1 2 3 4 5 6 7 8 9 10
Value x Y x E Q x S x U x A
T
hash(T): 7
hash2(T): 7

Index 0 1 2 3 4 5 6 7 8 9 10
Value x Y x E Q x S T U x A
I
hash(I): 7
hash2(I): 4
Collision found
Index 0 1 2 3 4 5 6 7 8 9 10
Value I Y x E Q x S T U x A
O

Figure 6: Solution to Q6a (Part2)

hash(O): 2
hash2(O): 5

Index	0	1	2	3	4	5	6	7	8	9	10
Value	I	Y	O	E	Q	x	S	T	U	x	A

N

hash(N): 1
hash2(N): 6

Index	0	1	2	3	4	5	6	7	8	9	10
Value	I	Y	O	E	Q	x	S	T	U	N	A

Figure 7: Solution to Q6a (Part3)

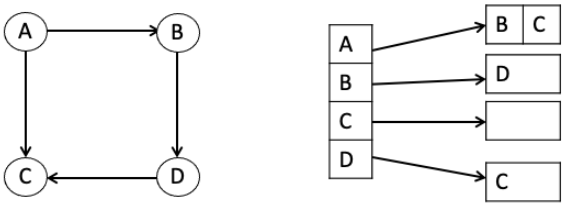
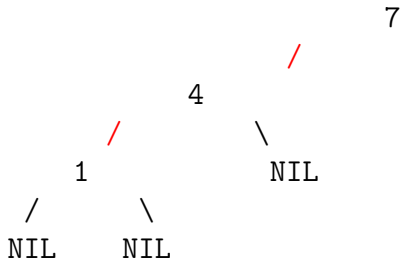


Figure 8: Example graph for Q8 solution

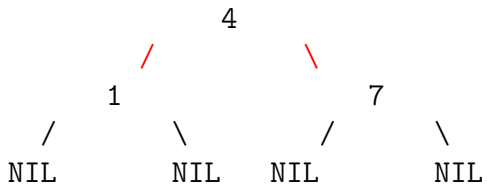
Bonus Question: Q5 Show that red-black BSTs are not memoryless: for example, if you insert a key that is smaller than all the keys in the tree and then immediately delete the minimum, you may get a different tree. [7 marks]

Answer: Suppose we have the following simple tree:

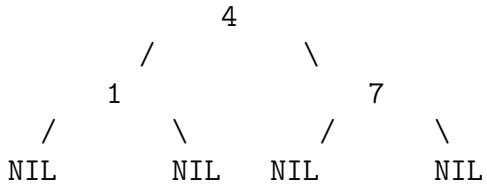
We insert 1 into this tree, so we end up with the following intermediate tree:



The tree then rotates to maintain balance:



The red edges are then flipped to black edges as follows:



We delete node 1 and we can clearly see that our new tree is different: