Assignment 2

Mingzhe Wang
wangm235@mcmaster.ca

Xing Li
li64@mcmaster.ca

March 20, 2021

Contents

1 Question 1

As the BinarySearchST is already sorted before inserting a new key, we only need to add one statement to compare the new key and keys[N-1]. If the new key is larger than all keys in the table, we can append it as the last element and skip the rank and element shift. In this way, it only tasks constant time and building a table by calling put() for keys that are in order takes linear time.

```
1  public class BinarySearchST<Key extends Comparable<Key>,
       Value>
2   {
3       private Key[] keys;
4       private Value[] vals;
5       private int N;
6
7       public BinarySearchST(int capacity)
8       {
9           keys = (Key[]) new Comparable[capacity];
10          vals = (Value[]) new Object[capacity];
11      }
12
13      public int size()
14      {  return N;  }
15
16      public Value get(Key key)
17      {
18          if (isEmpty()) return null;
19          int i = rank(key);
20          if (i < N && keys[i].compareTo(key) == 0) return
                vals[i];
21          else return null;
22      }
23
24      public int rank(Key key)
25      {
26          int lo = 0, hi = N - 1;
27          while (lo <= hi)
28          {
29                  int mid = lo + (hi - lo) / 2;
30                  int cmp = key.compareTo(keys[mid]);
31                  if (cmp < 0) hi = mid - 1;
32                  else if (cmp > 0) lo = mid + 1;
33                  else return mid;
34          }
```

```
35          return lo;
36      }
37
38      public void put(Key key, Value val)
39      {
40          int i;
41          if (keys[N-1].compareTo(key) < 0) i = N;
42          else i = rank(key);
43          //  when i == N, the if and for statements below
                takes constant time
44          if (i < N && keys[i].compareTo(key) == 0)
45          { vals[i] = val; return; }
46          for (int j = N; j > i; j--)
47          { keys[j] = keys[j-1]; vals[j] = vals[j-1]; }
48          keys[i] = key;
49          vals[i] = val;
50          N++;
51      }
52
53  }
```

2 Question 2

Algorithm 1 Reverse BST

```
1:  procedure ReverseTree(root)
2:      root = reverse(root)
3:
4:  procedure Reverse(node)
5:      if node == Nil then
6:          return Nil
7:      left = reverse(node.left)
8:      right = reverse(node.right)
9:      node.left = right
10:     node.right = left
11:     return node
```

3 Question 3

Algorithm 2 Inorder Tree Walk

```
1: procedure InorderTraverse(root)
2:
3:     procedure PushNodes(node)
4:         current_node = node
5:         while current_node ≠ Nil do
6:             Push current_node
7:             current_node = current_node → left
8:
9:     if root == Nil then
10:        return
11:    Initiate an empty stack S
12:    PushNodes(root)
13:    while S is not empty do
14:        current = Pop()
15:        visit(current)
16:        if current → right ≠ Nil then
17:            PushNodes(current → right)
```

4 Question 4

The largest possible number of internal nodes in a red-black tree can be reached when the hight of the red-black tree is the maximum. At that time its height is twice of the black hight k: 2k. And for a tree of hight 2k, the maximum number of internal nodes is $2^{2k} - 1$. Therefore, the largest possible number of internal nodes in a red-black tree with black height k is $2^{2k} - 1$.

The smallest possible number of internal nodes in a red-black tree can be reached when all nodes are black. At that time its height is the black hight: k. And for a tree of hight k, the smallest number of internal nodes is $2^k - 1$. This is because of the constraint that the number of black nodes of all paths from leaf to root have to be k. Therefore, the smallest possible number of internal nodes in a red-black tree with black height k is $2^k - 1$.

5 Question 6

5.1 Question 6 (a)

```
54
55 public class LinearProbingHashST<Key, Value>
56 {
57     private int N;           // number of key-value pairs
           in the symbol table
58     private int M = 11;      // size of linear probing
           table
59     private Key[] keys;      // the keys
60     private Value[] vals;    // the values
61
62     public LinearProbingHashST()
63     {
64         keys = (Key[])   new Object[M];
65         vals = (Value[]) new Object[M];
66     }
67
68     private int mapAlphabet(String letter)
69     {
70         return (letter.charAt(0) - 'A' + 1);
71     }
72
73     private int hash1(Key key)
74     {
75         return mapAlphabet((String) key) % M;
76     }
77
78     private int hash2(Key key)
79     {
80         double A = 0.6180339887;
81         return (int) Math.floor(M * ((mapAlphabet((String)
               key) * A) % 1));
82     }
83
84     private int hash(Key key, int i)
85     {
86         return (hash1(key) + i * hash2(key)) % m;
87     }
88
89     public void put(Key key, Value val)
90     {
91         int i;
```

```
92          for (i = 0; keys[hash(key, i)] != null; i++) {
93              if (keys[hash(key, i)].equals(key)) {
94                  vals[hash(key, i)] = val;
95                  return;
96              }
97          }
98          keys[hash(key, i)] = key;
99          vals[hash(key, i)] = val;
100         N++;
101     }
102
103     public Value get(Key key)
104     {
105         for (int i = 0; keys[hash(key, i)] != null; i++) {
106             if (keys[hash(key, i)].equals(key))
107                 return vals[hash(key, i)];
108         }
109         return null;
110     }
111 }
```

5.2 Question 6 (b)

Note: below, $hash(k, i) = (hash1(k) + i * hash2(k)) \mod 11$.

- insert E
  hash("E", 0) = 5. That index is empty, therefore insert.

- insert A
  hash("A", 0) = 1. That index is empty, therefore insert.

- insert S
  hash("S", 0) = 8. That index is empty, therefore insert.

- insert Y
  hash("Y", 0) = 3. That index is empty, therefore insert.

- insert Q
  hash("Q", 0) = 6. That index is empty, therefore insert.

- insert U
  hash("U", 0) = 10. That index is empty, therefore insert.

- insert T
  hash("T", 0) = 9. That index is empty, therefore insert.

- insert I
  hash("I", 0) = 9. That index is full, therefore next probing.
  hash("I", 0) = 4. That index is empty, therefore insert.

- insert O
  hash("O", 0) = 4. That index is full, therefore next probing.
  hash("O", 1) = 6. That index is full, therefore next probing.
  hash("O", 2) = 8. That index is full, therefore next probing.
  hash("O", 3) = 10. That index is full, therefore next probing.
  hash("O", 4) = 1. That index is full, therefore next probing.
  hash("O", 5) = 3. That index is full, therefore next probing.
  hash("O", 6) = 5. That index is full, therefore next probing.
  hash("O", 7) = 7. That index is empty, therefore insert.

- insert N
  hash("N", 0) = 3. That index is full, therefore next probing.
  hash("N", 1) = 10. That index is full, therefore next probing.
  hash("N", 2) = 6. That index is full, therefore next probing.
  hash("N", 3) = 2. That index is empty, therefore insert.

Trace for each insertion is showed below:

```
1  null,null,null,null,null,E,null,null,null,null,null,
2  null,A,null,null,null,E,null,null,null,null,null,
3  null,A,null,null,null,E,null,null,S,null,null,
4  null,A,null,Y,null,E,null,null,S,null,null,
5  null,A,null,Y,null,E,Q,null,S,null,null,
6  null,A,null,Y,null,E,Q,null,S,null,U,
7  null,A,null,Y,null,E,Q,null,S,T,U,
8  null,A,null,Y,I,E,Q,null,S,T,U,
9  null,A,null,Y,I,E,Q,O,S,T,U,
10 null,A,N,Y,I,E,Q,O,S,T,U,
```

6 Question 7

Proof. Let P(n) be the statement that every connected graph with n vertices has a vertex whose removal will not disconnect the graph, we will prove it with weak induction.

Base Case: n = 2.
When n = 2, removing any vertex will leave a single vertex and it's connected with itself. Therefore, P(2) holds.

Induction Step: When P(n) holds for all $n \geq 2$, we will prove P(n+1) holds. For a connected graph with n + 1 vertices, we can arbitrarily pick n vertices. Since it's a connected graph, the n vertices are connected. With the induction hypothesis, in the n connected vertices, we can find a vertex whose removal will not disconnect the graph, named as V_i. We can also name the vertex which is not picked as V_j.
1. If V_j is adjacent with another node in the n vertices other than V_i, then V_i is the one we can remove.
2. If V_j not adjacent with any node in the n vertices other than V_i, then it has to be adjacent V_i. Otherwise the graph with n + 1 vertices is not connected. In this situation, we can remove V_j instead. Because it's only connected to V_i, the removal won't disconnect the rest graph with n vertices.
Therefore, P(n) holds for $\forall n \geq 2 \in \mathbb{N}$ by weak induction. □

In the algorithm below, we assume that the graph has at least two vertices. Otherwise, it makes no sense to find such a vertex.

Algorithm 3 Find Vertex vis DFS

```
1: procedure Get_Vertex(V)
2:     Initiate an empty stack S
3:     Push V
4:     while S is not empty do
5:         current_v ← S.top()
6:         Mark current_v as visited
7:         if all vertices adjacent to current_v are visited then
8:             return current_v
9:         else
10:            Push one of the unvisited vertices into stack
```

7 Question 8

To produce a topological order, we need to put all start nodes (with indegree 0) into the queue first. After that, if we used only the given algorithm, we could have generated wrong topological order for some graph as shown below.

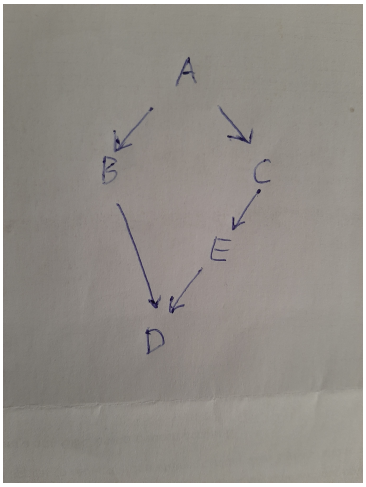

Figure 1: Directed Acyclic Graph

With BFS, we'll put A in the queue and mark its distance as 0. Then we'll pick A and push B and C into the queue and mark the distance of B and C as 1. After that, we'll pick A and push D into the queue. However, D should be behind E in topological order. This situation happens because in the algorithm we don't check the indegree of each node, which is the fault of the given algorithm based on BFS.