COSC 419 Assignment 1

Martin Wallace

February 26, 2018

1. **Steps taken per runtime for various sizes of n**

Table 1

	10	100	1000
$\frac{1}{2}n^2$	50.0	5000.0	500000.0
$3n\ log_2 n$	99.7	1993.2	29897.4
$n\sqrt{n}$	31.6	1000.0	31622.8
$2^{\frac{n}{4}}$	5.7	33554432.0	1.8e+75

If the system is expected to support input sizes of ten thousand or more then you would want to use the algorithm with complexity O($3n\ log_2 n$).

2. **An interface for an excel like table**

```java
//Table.java
package assignment1;
import java.io.FileNotFoundException;
public interface Table<T>{
    public void populateFromCSVFile(String csvFilename) throws
        FileNotFoundException, InvalidCSVException;
    public int height();
    public int width();
    public T[] getRow(int rowNum);
    public T[] getCol(int colNum);
    public T[][] getFullTable();
    public void set(int row, int col, T data);
}
```

3. **Hashmaps for a shopping list**

```java
package assignment1;

import java.util.HashMap;
```

```java
import java.util.Map;
import java.util.Scanner;

public class Q3 {
    public static void main(String[] args) {
        /*
            For this question we will assume that the input looks
                like
            n - Followed by n lines of:
            Item Price
            m - Followed by M lines of:
            Quantity Item
            Assuming that the data is coming from stdin
         */
        Scanner input = new Scanner(System.in);
        Map<String, Double> prices = new HashMap<>();
        double total = 0;
        int n = Integer.parseInt(input.nextLine());
        while(n-->0) {
            String item = input.next();
            double price = Double.parseDouble(input.nextLine());
            prices.put(item, price);
        }
        int m = Integer.parseInt(input.nextLine());
        while(m-->0) {
            int quantity = Integer.parseInt(input.next());
            String item = input.nextLine().trim();
            if(prices.containsKey(item)) {
                total += prices.get(item) * quantity;
            }else {
                throw new RuntimeException("Undefined grocery item");
            }
        }
        System.out.println(String.format("The total of the shopping
            list is: $%.2f", total));
        System.exit(0);
    }
}
```

The output of running the program:

```
$ java Q3 << cat EOF
5
Book     8.95
Pen      0.99
Eraser   0.50
Case     3.75
Backpack 29.99
4
```

```
1 Backpack
6 Pen
2 Eraser
1 Book
EOF
The total of the shopping list is: $45.88
$
```

---

4. **Evaluate the run time of a code fragment**

   for $i \leftarrow 0$ to n - 1 **do**
       for j $\leftarrow$ 0 to $n \cdot n$ - 1 **do**
           for k $\leftarrow$ 0 to j - 1 **do**
               $s \leftarrow s + 1$

   Given that the outer loop **for** $i \leftarrow 0$ to n - 1 **do** will run exactly n times we know that the each piece of code inside this loop will run exactly n times. Therefor we can say that the run time of the entire expression is O(n) $\cdot$ O(inner loops).

   Given that the first inner loop **for** j $\leftarrow$ 0 to $n \cdot n$ - 1 **do** will run exactly n^2 times we can say that the run time of the entire expression is O(n) $\cdot$ O(n^2) $\cdot$ O(inner loop).

   Given that the last inner loop **for** k $\leftarrow$ 0 to j - 1 **do** will run on average $\frac{1}{2}n^2$ times.

   Then we can say that the run time of the entire code fragment is O(n) $\cdot$ O(n^2) $\cdot$ O($\frac{1}{2}n^2$), or O(n^5).

5. **The evil (and cheap) king**

   Given that the king has n bottles of wine, only 1 bottle is poisoned, and it will take exactly one month to kill anyone who drinks a drop of wine, we can determine which glass of wine was poisoned in one month using exactly ceil($log_2 n$) taste testers.

   (a) First label each bottle from *1..n* with numbers *0..n-1* respectively.

   (b) Set $m \leftarrow ceil(log_2 n)$.

   (c) Next gather your m taste testers and brand them (literally, because you are an evil king) with a number from *0..m-1*.

   (d) Now for each bottle with number i we will have each taste tester with a number j, such that $(1 << j)$ & $i = 1$, drink a drop from the bottle. Or less formally j is the index of a bit which is set to 1 in the binary representation of i.

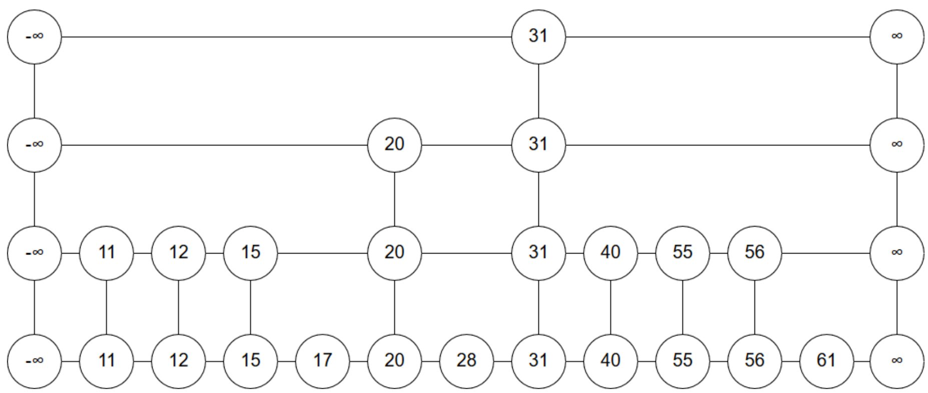
   (e) One month later gather of all of taste testers who died and put their numbers into set D. Now the number on the label of the poisoned bottle of wine is equal to

$$\sum_{j \in D} 2^j$$

For example, if testers with numbers 0, 1 and 2 assigned to them died, we would know that the poisoned bottle was bottle number 7 ($2^0 + 2^1 + 2^2$). However, if taste tester number 3 died instead of number 0, then we would know that the number of the poisoned bottle would be 14 ($2^1 + 2^2 + 2^3$).

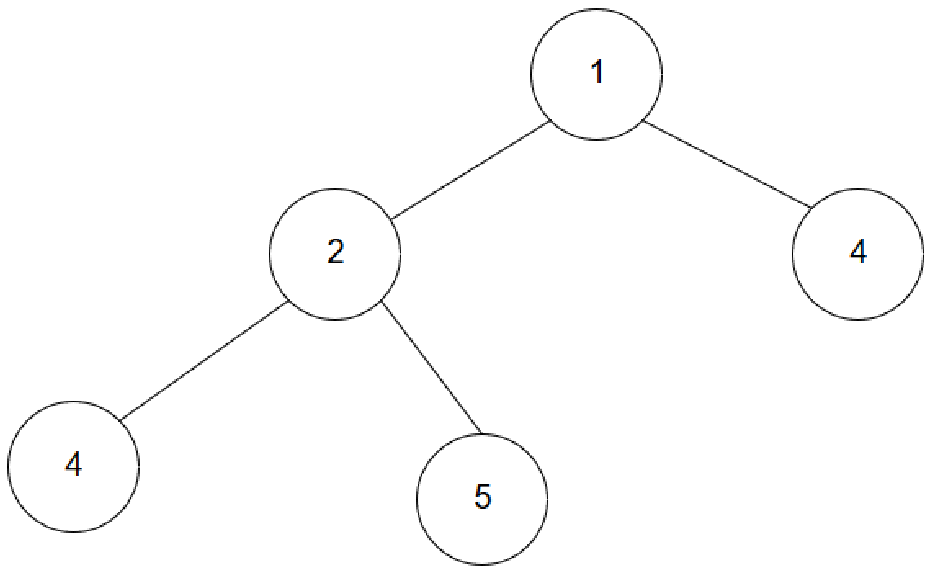
6. **Skip Lists**
   The results of inserting 12, 20, 40 with the next 12 flips being HTHHTHTHTTHH

Figure 1: Final Result of Skip List

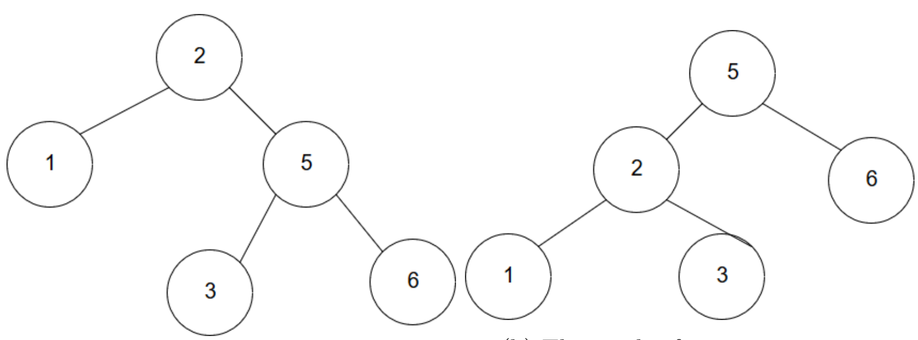


7. **Heaps**
   Vladimir is wrong that a preorder traversal will always list its keys in nondecreasing order. For example in the figure below, the preorder traversal would be 1,2,4,5,3. Since $5 > 3$ this breaks the nondecreasing order that Vladimir claims.

Figure 2: A heap that breaks Vladimir's assumptions



8. **Search Trees**
   Margherita is wrong that the order of insertion into a binary tree does not matter. As seem below.



(a) The result of inserting 2,5,1,6,3

(b) The result of inserting 5,2,3,1,6

Figure 3: Two distinct binary trees with the same set of keys

9. **Ternary Trees**

   (a) Given that a full ternary tree with depth 0 has 1 node, depth 1 has 4 nodes and depth 2 has 13 nodes. We can see that each level of depth f has exactly 3^f nodes at that level. So a full tree of depth d would have

$$\sum_{i=0}^{d} 3^i$$

nodes. This can be written in the closed form of

$$\frac{3^{d+1}-1}{2}$$

(b) Given that the tree is zero-indexed, we can find the left, centre and right children of any given index using the following three formulas:

- $left(i) \rightarrow i \cdot 3 + 1$
- $centre(i) \rightarrow i \cdot 3 + 2$
- $right(i) \rightarrow i \cdot 3 + 3$

(c) $depth(i) \rightarrow floor(log_3((i+1) * 2))$

10. **Search Trees**
Show the results of inserting 15,6,12,1,14,11,4,13,3,9,10,2,5,7,8 in the given order into various trees.

Figure 4: Items inserted into a binary tree

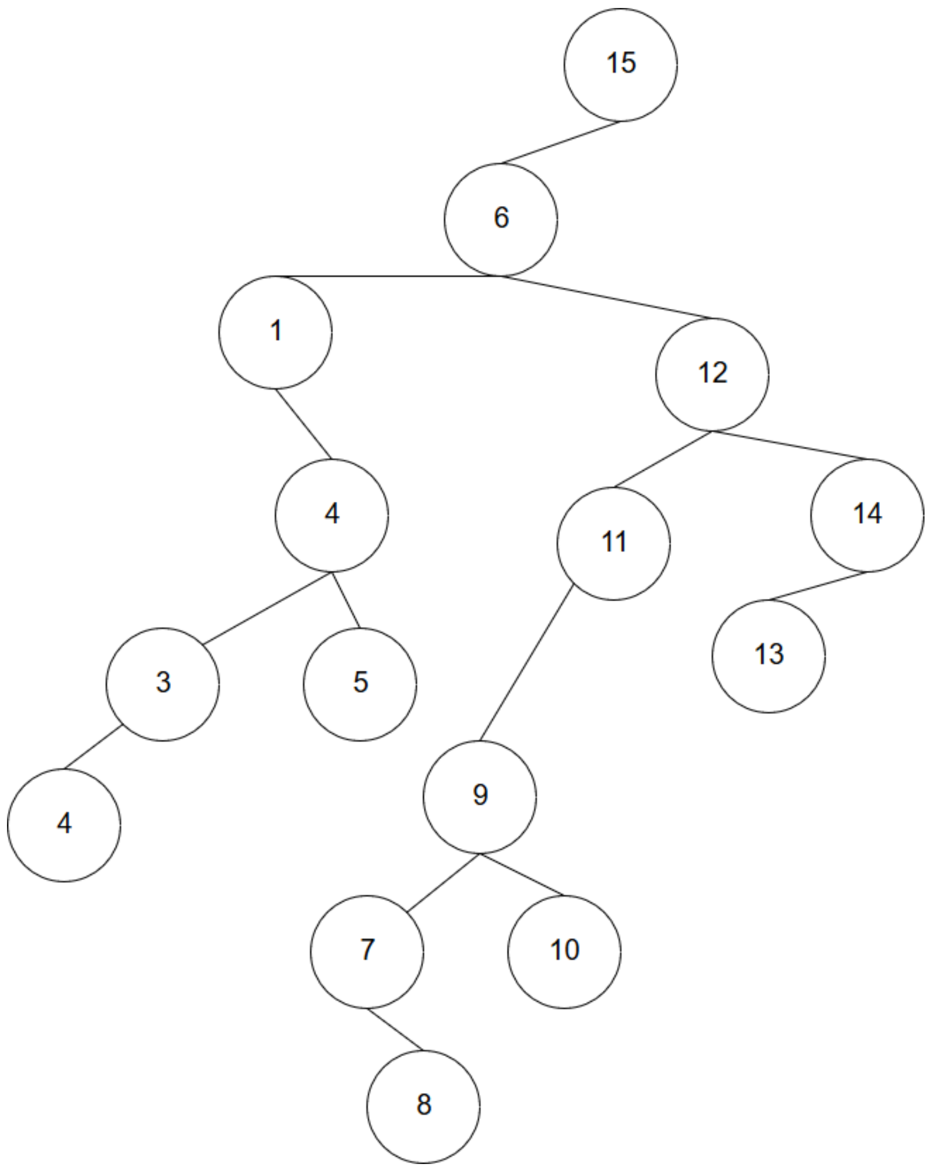


Figure 5: Items inserted into a red black tree

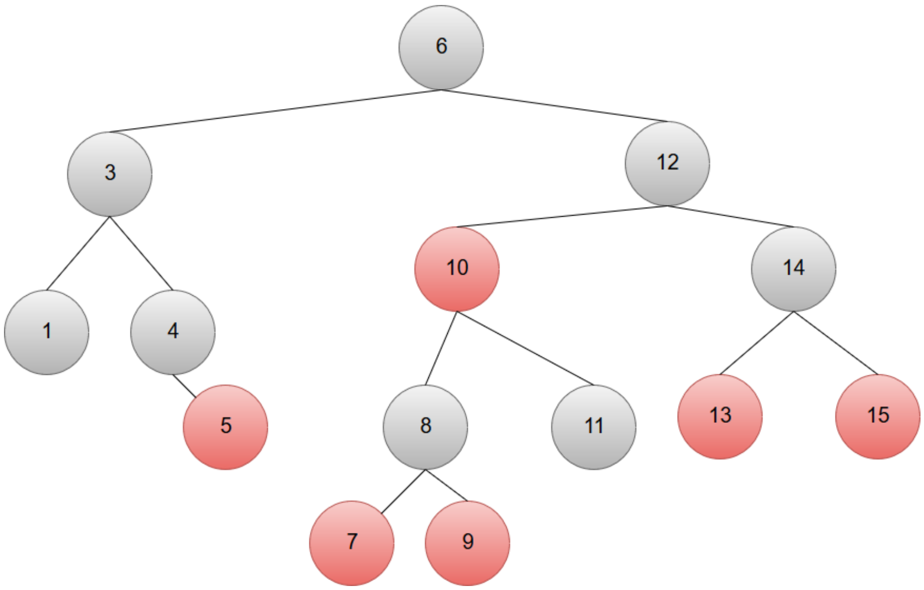


Figure 6: Items inserted into a splay tree

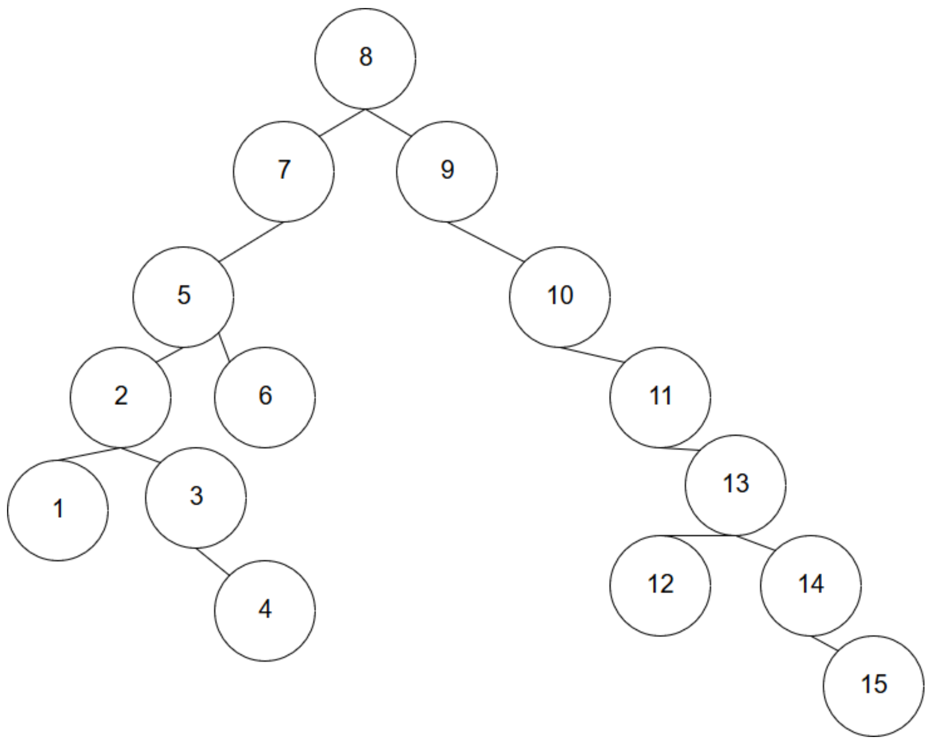
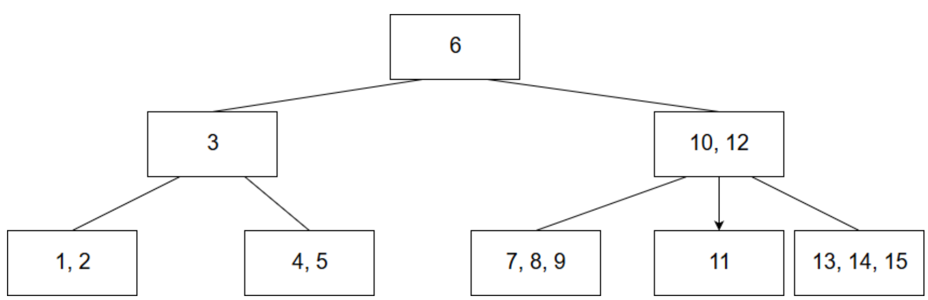


Figure 7: Items inserted into a 2-4 tree



Show the results after each step of removing 5,8,6 for each tree

Figure 8: Binary tree after removing 5

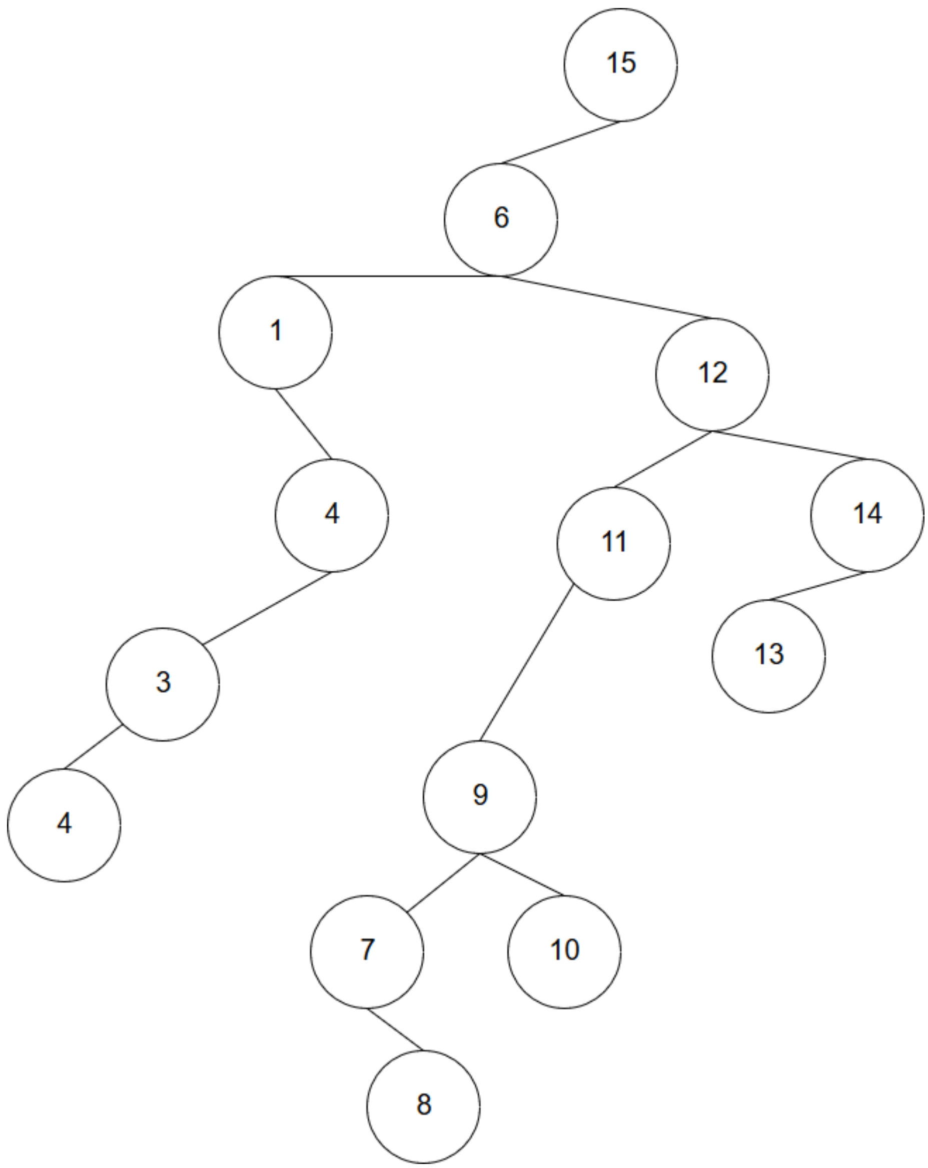


Figure 9: Binary tree after removing 8

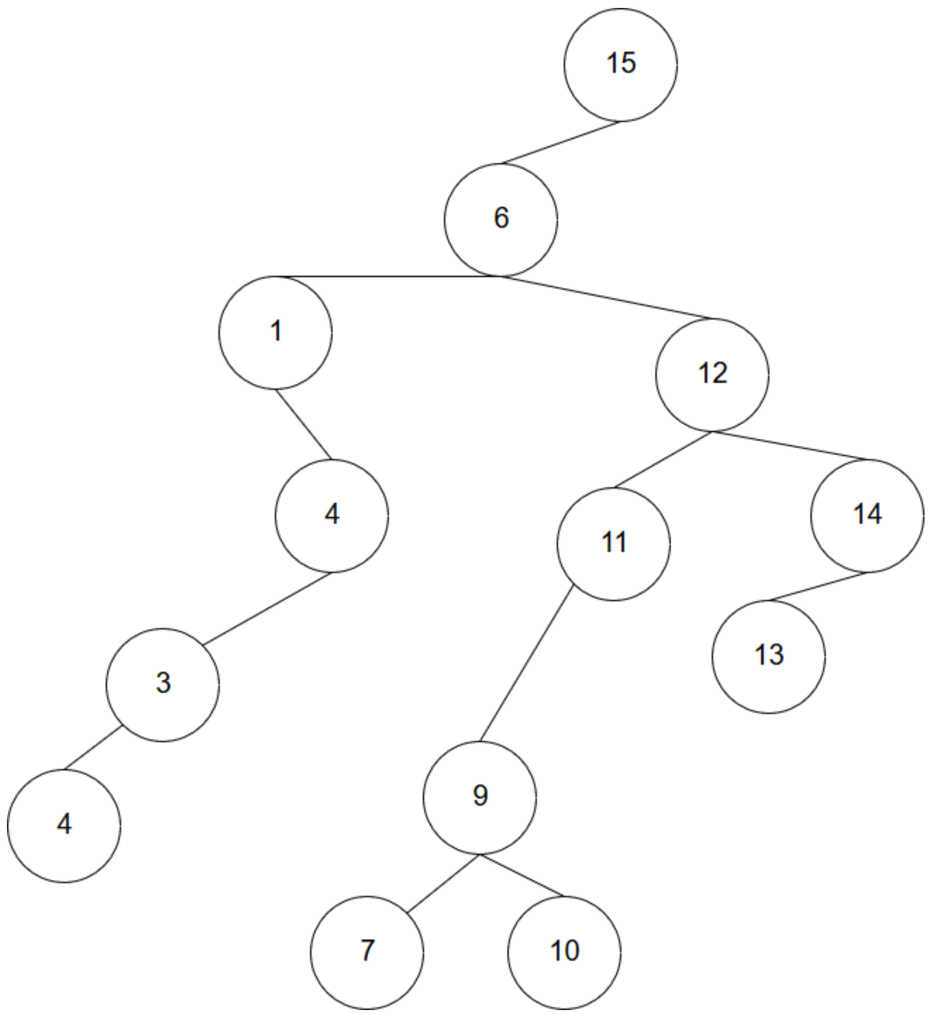


Figure 10: Binary tree after removing 6

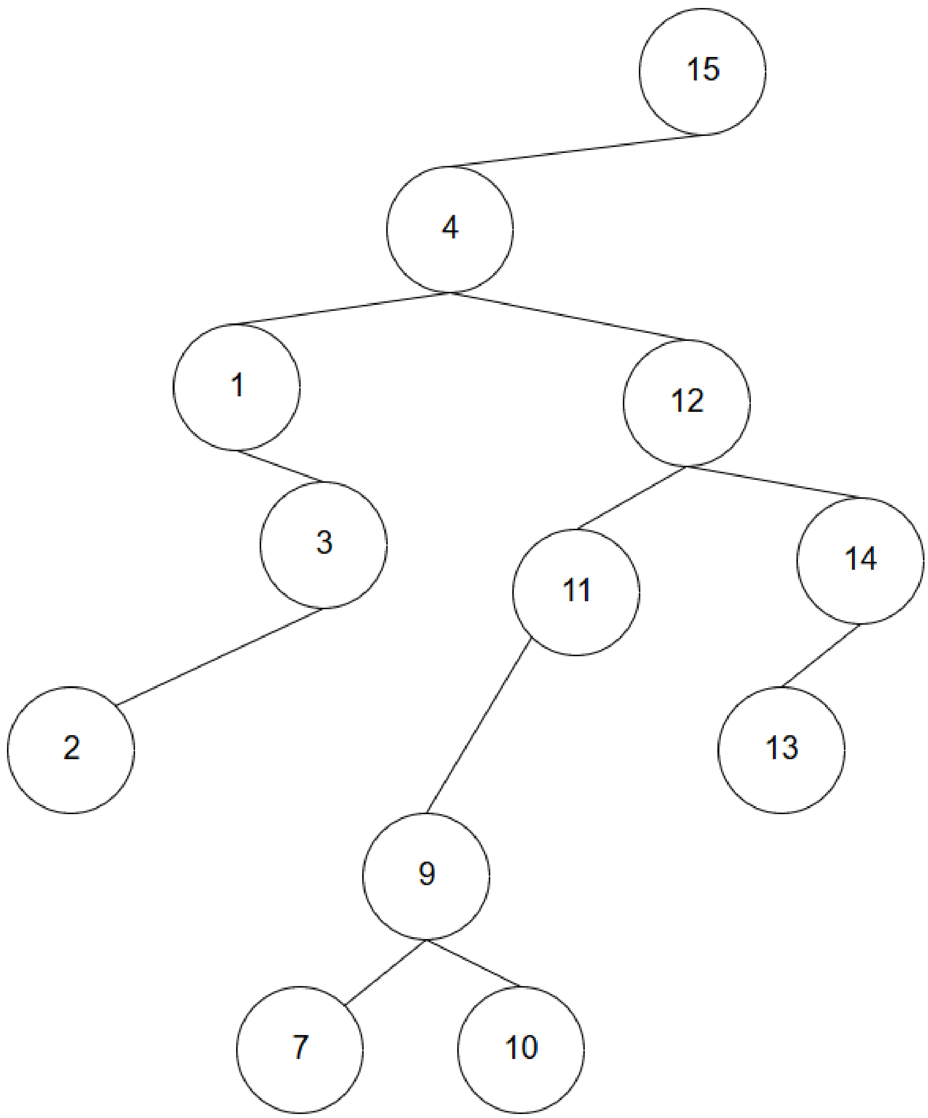


Figure 11: Red black tree after removing 5

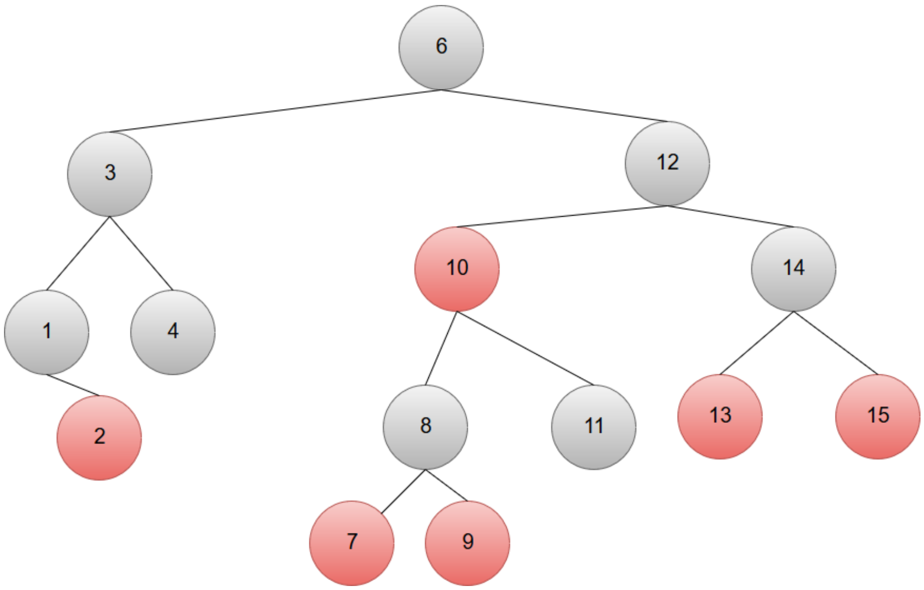


Figure 12: Red black tree after removing 8

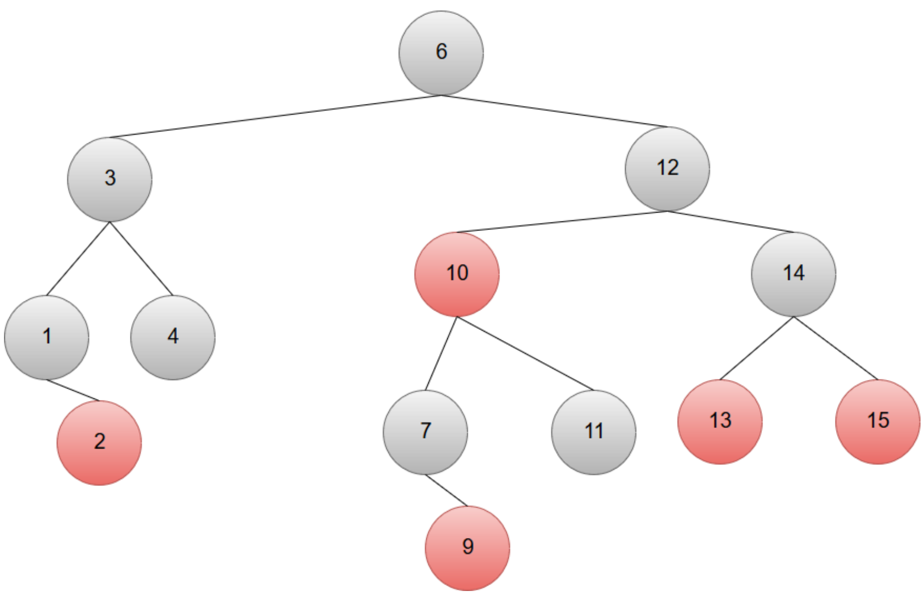


Figure 13: Red black tree after removing 6

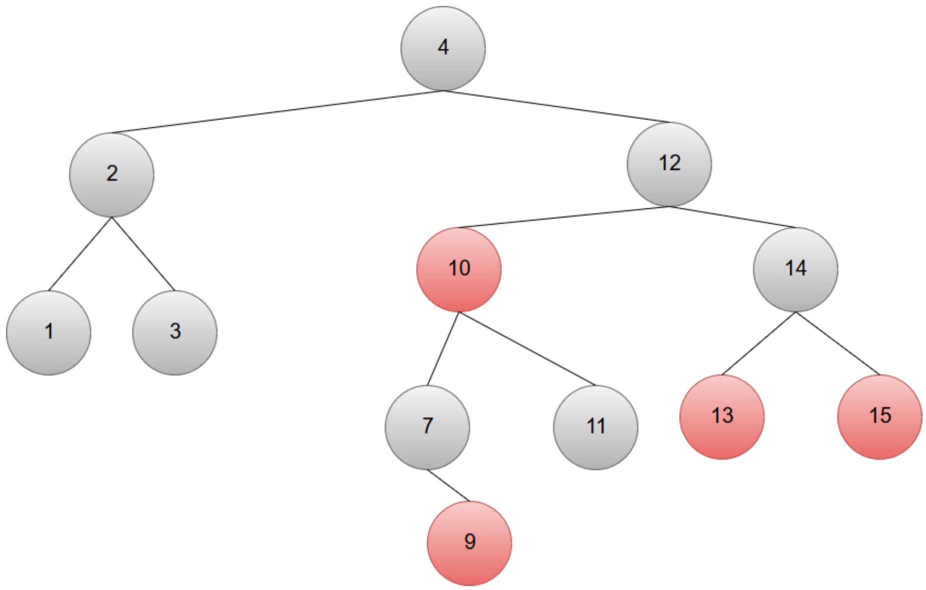


Figure 14: Splay tree after removing 5

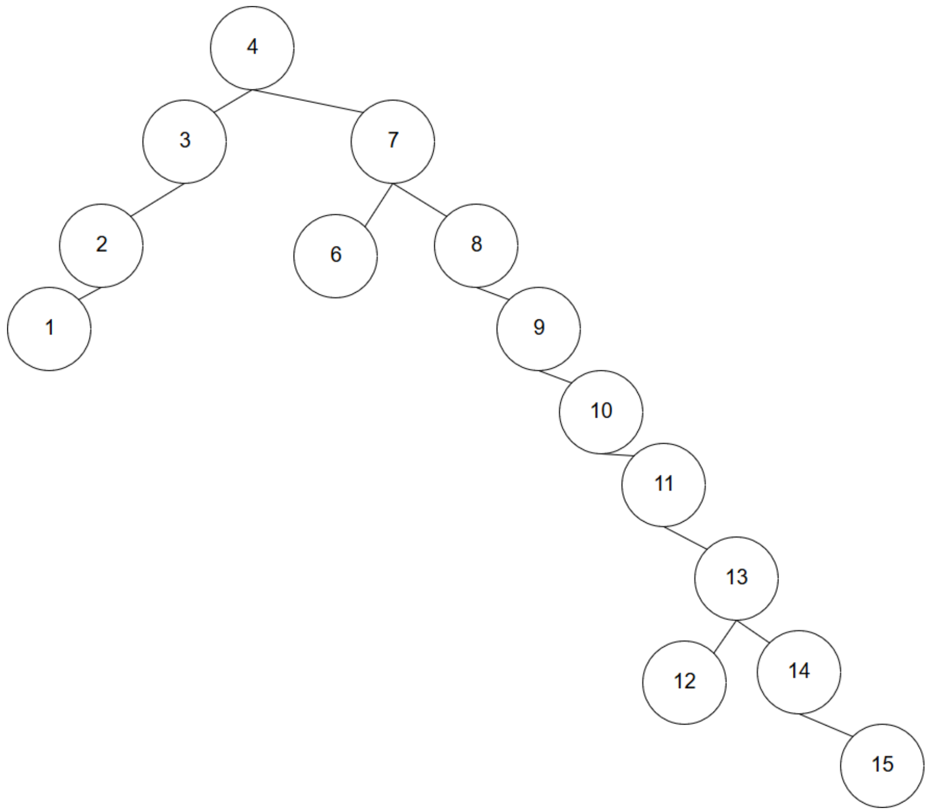


Figure 15: Splay tree after removing 8

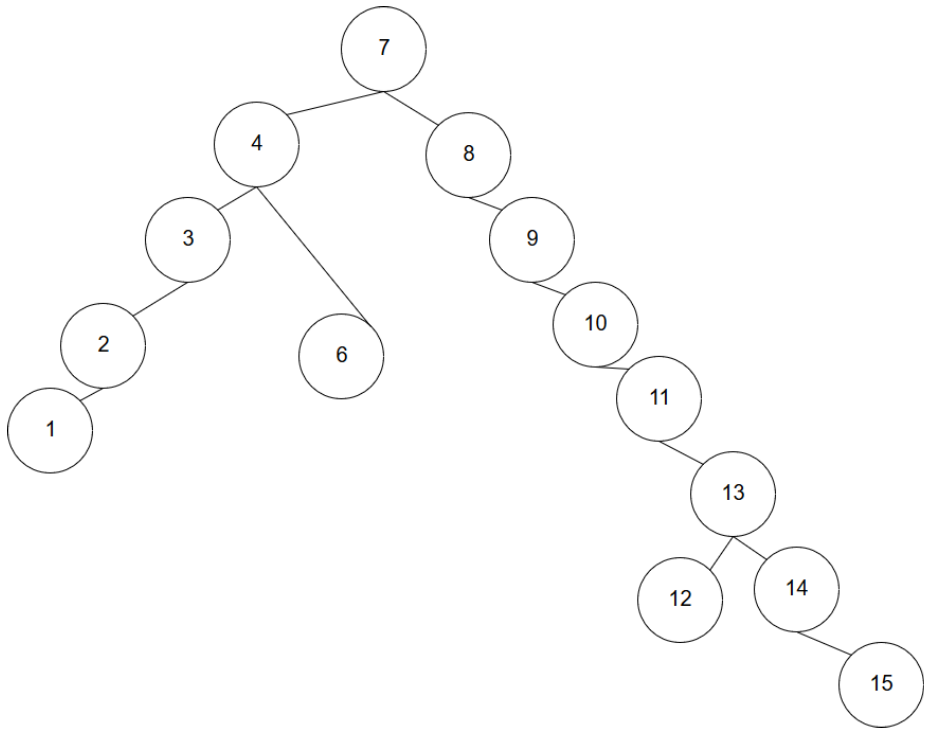


Figure 16: Splay tree after removing 6

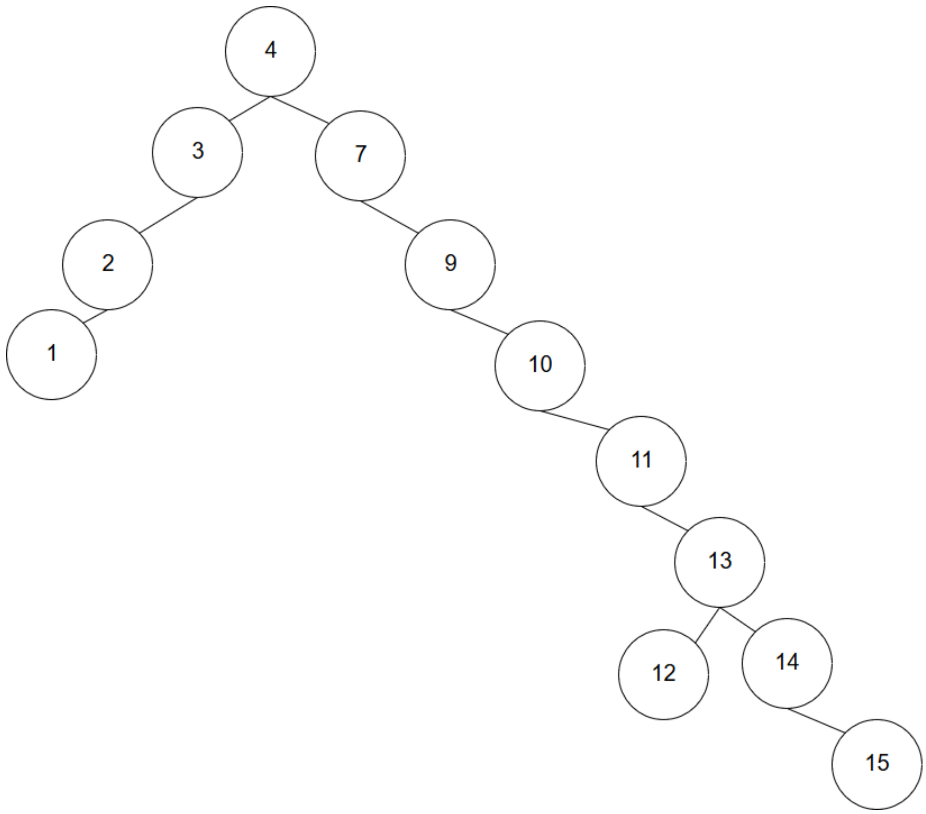


Figure 17: 2-4 tree after removing 5

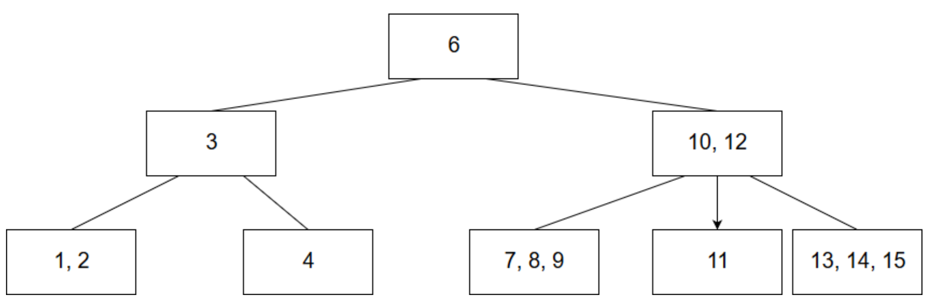


Figure 18: Splay tree after removing 8

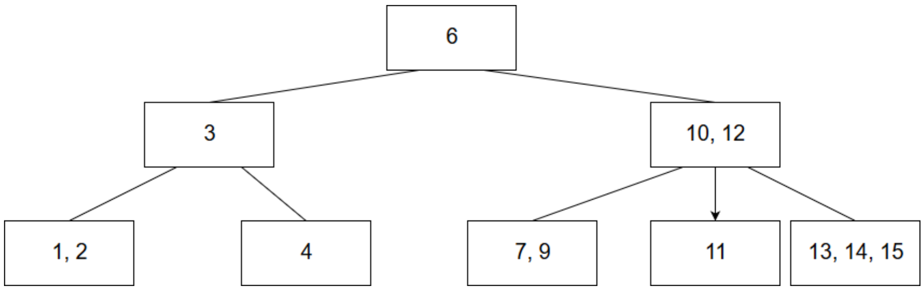
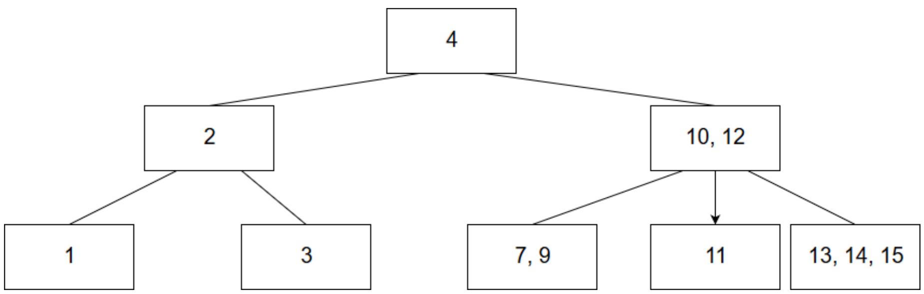
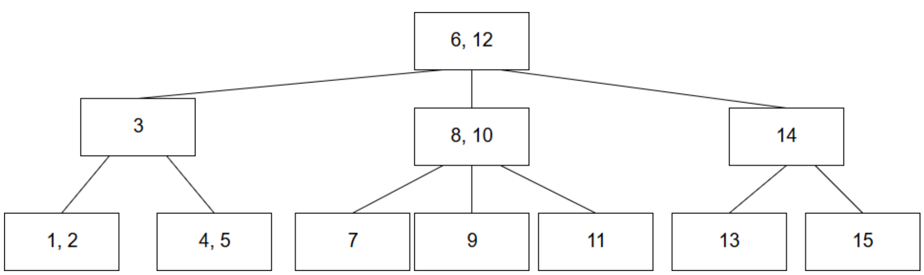


Figure 19: Splay tree after removing 6



11. **2-3 Tree**
    Show the results of inserting 15,6,12,1,14,11,4,13,3,9,10,2,5,7,8 into a 2-3 Tree

Figure 20: 2-3 Tree after inserts

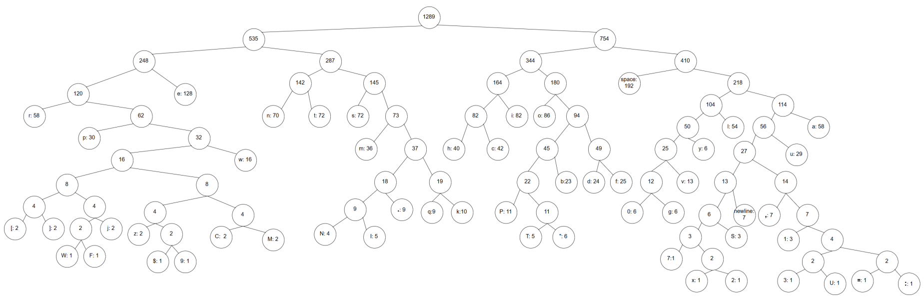


12. **Huffman Code**

    (a) The given text contains 1289 characters, which means that if we needed 7 bytes to for each character we would require a total of 9023 bytes.

(b) Full Huffman Tree (larger version available at end of document)

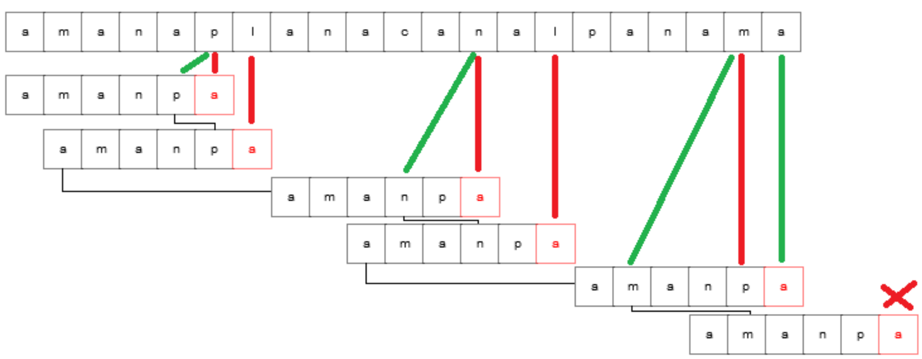
Figure 21: 2-3 Tree after inserts



(c) The number of bits required to store the text in a prefix code was 5882 which is only 65.2% of the number of bits required for the uncompressed version.

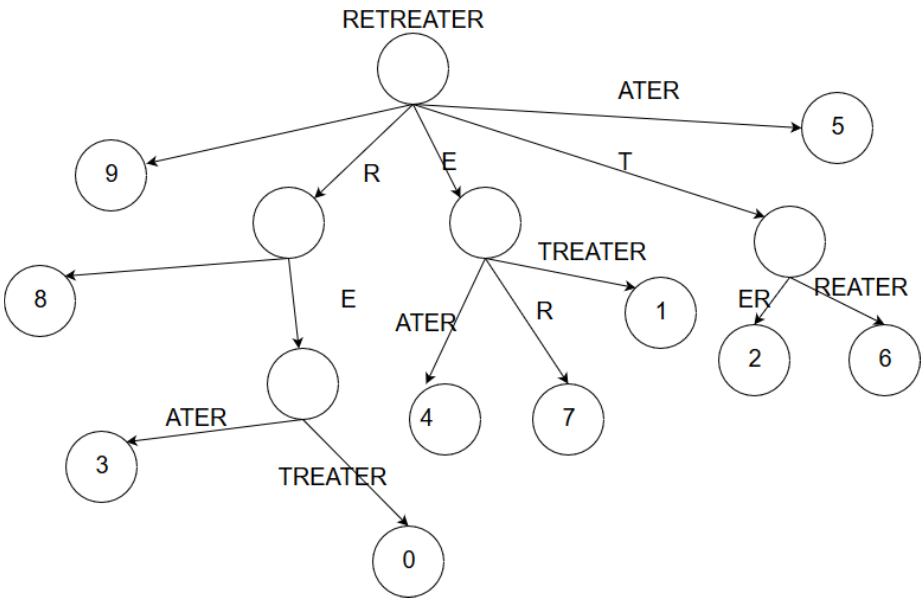
13. **Pattern Matching**

Figure 22: Boyer Moore Algorithm searching for amanpa in amanaplanacanalpanama



14. **Tries**

Figure 23: The compressed suffix trie of RETREATER



15. **2-4 Tree Contains Method**

```
package assignment1;
public class TwoFourTree {
    static class Node {
        int size;
        int[] keys = new int[3];
        Node[] children = new Node[4];
    }
    public static boolean find(Node r, int k) {
        // Make sure node is valid
        if (r == null || r.size == 0)
            return false;
        for(int i = 0; i < r.size; i++)
            if(k == r.keys[i])
                return true;
            else if(k < r.keys[i])
                return find(r.children[i], k);
        // If we made it here then k > all keys
        // call find on rightmost child
        return find(r.children[r.size], k);
    }
}
```

Figure 24: 2-3 Tree after inserts