Huffman Coding and Red-Black Tree

Subhabrata Samajder

IIIT, Delhi

Summer Semester,

23rd June, 2022

Huffman Coding

Huffman Coding

- Are used to compress information.
  - Like WinZip (although WinZip doesn't use the Huffman algorithm).
  - JPEGs do use Huffman as part of their compression process.

- **Basic idea:** Instead of storing characters in a file as 8-bit ASCII value, store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits
  - On average this should decrease the filesize (usually by 1/2).

Huffman Coding (Cont.)

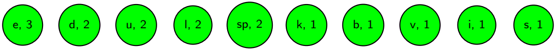
An Example: Consider the string,

"duke blue devils"

- Do a frequency count of the characters:

e	d	u	l	space	k	b	v	i	s
3	2	2	2	2	1	1	1	1	1

- Next we use a Greedy algorithm to build up a Huffman Tree.
  - Start with nodes for each character.



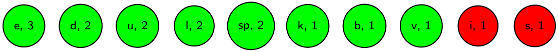
Huffman Coding (Cont.)

- Pick the nodes with the smallest frequency and combine them together to form a new node.
  - The selection of these nodes is the Greedy part.

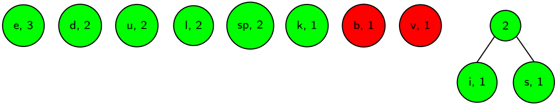
- Remove the two selected nodes are from the set and replace it with a combined node.

- Continue until only 1 node left in the set.

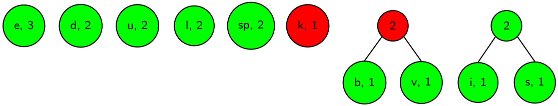
Huffman Coding (Cont.)



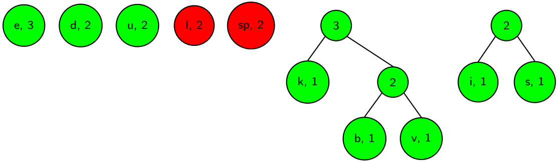
Huffman Coding (Cont.)



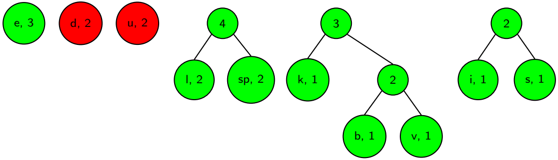
Huffman Coding (Cont.)



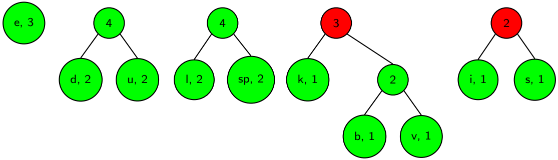
Huffman Coding (Cont.)



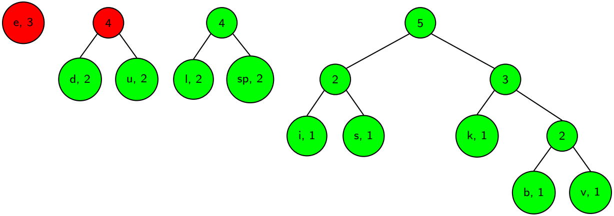
Huffman Coding (Cont.)



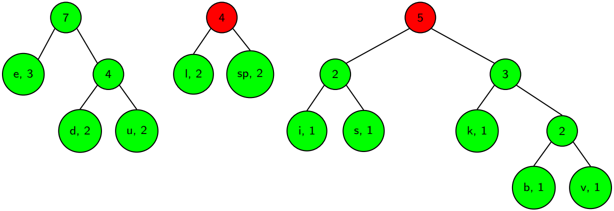
Huffman Coding (Cont.)



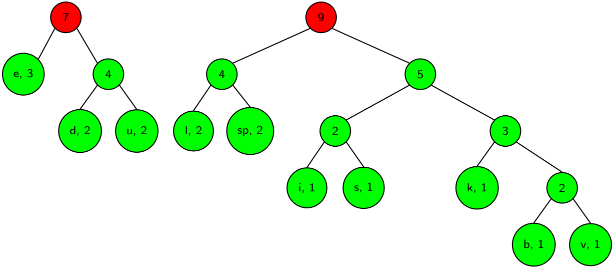
Huffman Coding (Cont.)



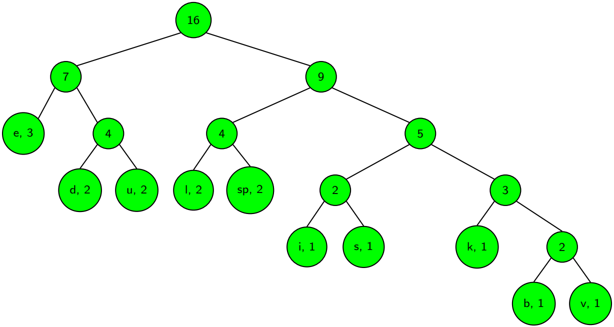
Huffman Coding (Cont.)



Huffman Coding (Cont.)



Huffman Coding (Cont.)



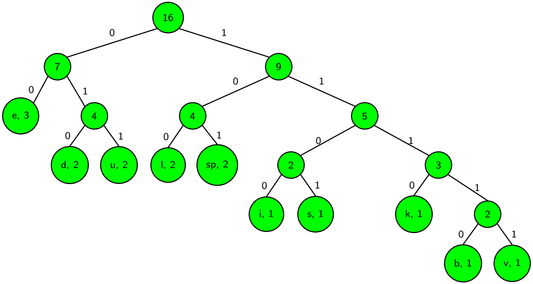
Huffman Coding: Assign Codes

- **Assign codes to the tree:**
  - 0: For left child.
  - 1: For Right child.

- A traversal of the tree from root to leaf gives the Huffman code for that particular leaf character.

- **Note:** No code is the prefix of another code.

Huffman Coding: Code Assignment



e	d	u	l	space	i	s	k	b	v
00	010	011	100	101	1100	1101	1110	11110	11111

Huffman Coding: Compression

- Use these codes to encode the string.
- Encoding of "duke blue devils":

010 011 1110 00 101 11110 100 011 00 101 010 00 11111 1100 100 1101

- Grouped then into bytes:

01001111 10001011 11101000 11001010 10001111 11100100 1101*xxxx*

- $\therefore$ it takes 7 bytes of space.
- In contrast, the uncompressed string takes 16 characters $\times$ 1 byte/char = 16 bytes.

Huffman Decoding: Uncompression

- Reading the compressed file bit by bit.
  - Start at the root of the tree.
  - If a 0 is read, head left.
  - If a 1 is read, head right.
  - When a leaf is reached decode that character and start over again at the root of the tree

- $\therefore$ Huffman table information needs to be saved as a header in the compressed file.
  - Doesnt add a significant amount of size to the file for large files (which are the ones you want to compress anyway).
  - Or we could use a fixed universal set of codes/freqencies.

Homework

- Implement Huffman encoding and decoding in C.
- Using it create a compression software that takes as input a text file and outputs a compressed file with the Huffman encoding table in the header of the file.
- Also create a uncompression software that will take the compressed file created above and output the original text file.

Red-Black Tree (RBT)

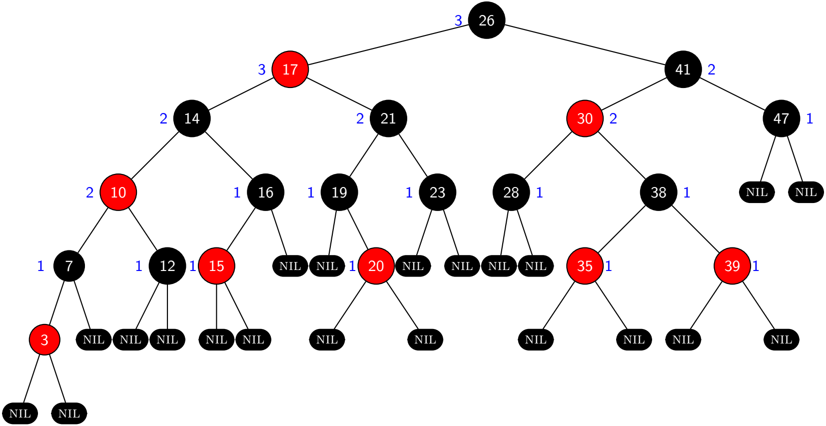
Properties

- Another example of balanced tree.
- Like AVL tree, ensures basic dynamic-set operations take $\mathcal{O}(\log n)$ in the worst case.
- It is a BST satisfying the following properties.
  1. Every node is either **red** or **black**.
  2. Root is **black**.
  3. Every leaf is **black**.

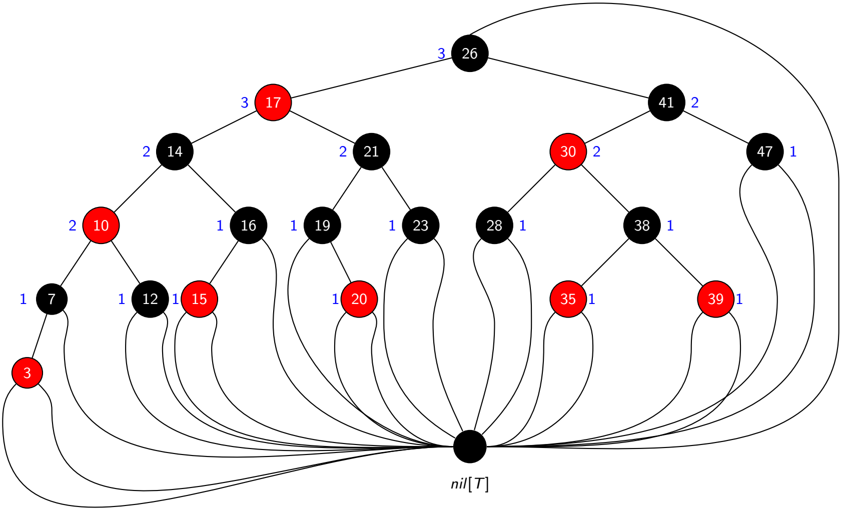
     Note: Only the null pointers are considered as leaf nodes. All other nodes are considered as internal nodes.
  4. If a node is **red** the both its children are **black**.
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

An Example Tree

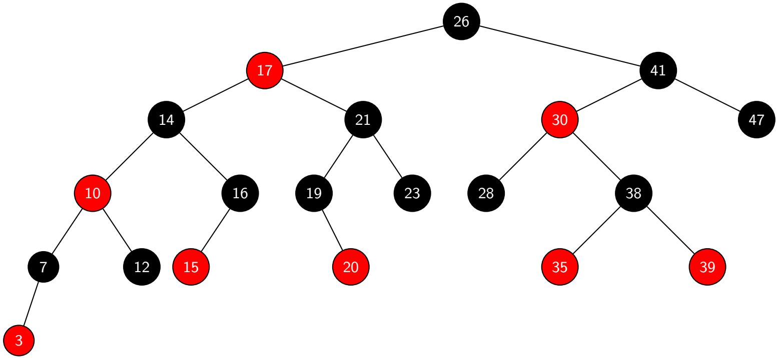
Black-height: # black node on any path from, but not including, a node x down to a leaf. It is denoted by $bh(x)$.



An Example Tree



An Example Tree



RBT Node in C

```
typedef struct RBTNode {
  int nKey;
  int nColour;              // 0 - Red and 1 - black
  struct RBTNode *pParent;
  struct RBTNode *pLeft;
  struct RBTNode *pRight;
} RBTNode;
```

Sentinel $nil[T]$**:**

- It is a RBTNode.
- Color = Black, i.e., nColour = 1.
- nKey, pParent, pLeft, pRight can be set to arbitrary values.

Lemma

Lemma

A RBT with n internal nodes has height at most $2\lg(n+1)$.

Lemma

Lemma

A RBT with n internal nodes has height at most $2\lg(n+1)$.

Claim: A subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Lemma

Proof of the claim: The proof is by induction on the height of x.

Base Case ($bh(x) = 0$): Then x must be a leaf ($nil[T]$) node $\Rightarrow$ # of internal nodes is at least $2^0 - 1 = 0$.

Inductive Step: Height of the children of x is either $bh(x)$ or $bh(x) - 1$.

Then by the inductive hypothesis a sub-tree rooted at x contains at least

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 \ = \ 2^{bh(x)} - 1$$

internal nodes.

Lemma

Proof of the Lemma: Let h be the height of the tree.

By Property 4, at least half of the nodes on any simple path from the root to a leaf, not including the root, must be black
$\Rightarrow \; bh(x) \geq h/2$.

$\therefore \; n \; \geq 2^{h/2} - 1 \quad \Rightarrow \quad h \; \leq \; \lg(n+1)$.

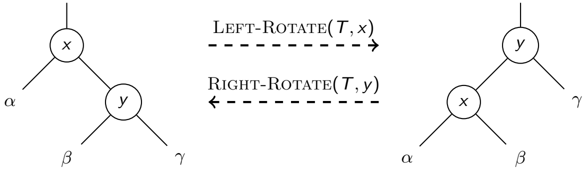
Rotations in a RBT

Rotations

- Insert and Delete operations may violate the red-black property.
- Can be restored by
  - changing the colours of some of the nodes and
  - also change the pointer structure.
- The pointer structure is changed via the following two types of rotation.
  - Left rotation.
  - Right rotation.

Left and Right Rotations

Left-Rotate: $right[x] \neq nil[T]$ and $p[root] = nil[T]$.
Right-Rotate: $left[x] \neq nil[T]$ and $p[root] = nil[T]$.



Complexity: $\mathcal{O}(1)$.

LEFT-ROTATE(T, x)

```
BEGIN
  y ← right[x]            // Set y
  right[x] ← left[y]      // Turn y's left sub-tree into x's right sub-tree

  if left[y] ≠ nil[t]
    then p[left[y]] ← x

  p[y] ← p[x]             // Link x's parent to y

  if p[x] = nil[T]
    then root[T] ← y
  else if x = left[p[x]]
    then left[p[x]] ← y
  else
    right[p[x]] ← y

  left[y] ← x
  p[x] ← y
END
```

LEFT-ROTATE(T, x)

BEGIN

```
y ← right[x]                  // Set y
right[x] ← left[y]            // Turn y's left sub-tree into x's right sub-tree

if left[y] ≠ nil[t]
   then p[left[y]] ← x

p[y] ← p[x]                   // Link x's parent to y

if p[x] = nil[T]
   then root[T] ← y
else if x = left[p[x]]
   then left[p[x]] ← y
else
   right[p[x]] ← y

left[y] ← x
p[x] ← y
```

END

Homework: Write the algorithm for the RIGHT-ROTATE.

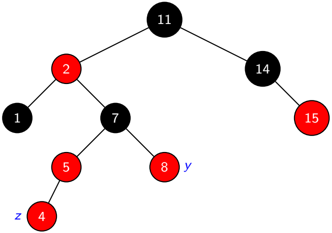
RBT: Insertion

Insertion

- Insert the z into the RBT T as if it was a BST.
- Colour the node z as red.

  Note: Colouring z as red does not effect the black height!
- Call an auxiliary procedure RB-INSERT-FIXUP to recolour nodes and perform rotations.

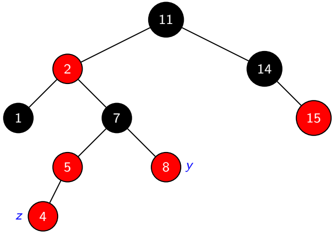
RB-INSERT-FIXUP(T, z)



Recall: Properties of a RBT:

1. Every node is either red or black.
2. Root is black.
3. Every leaf is black.
4. If a node is red the both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

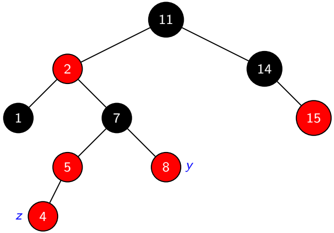
RB-Insert-Fixup(T, z)



Recall: Properties of a RBT:

1. Every node is either red or black.
2. Root is black.
3. Every leaf is black.
4. If a node is red the both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

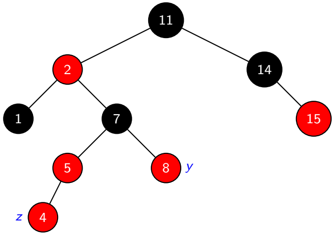
RB-INSERT-FIXUP(T, z)



Recall: Properties of a RBT:

1. Every node is either red or black.
2. Root is black.
3. Every leaf is black.
4. If a node is red the both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

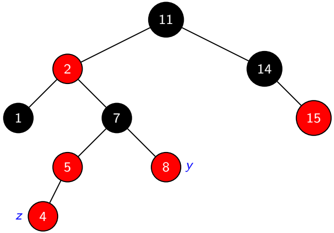
RB-INSERT-FIXUP(T, z)



Recall: Properties of a RBT:

1. Every node is either red or black.
2. Root is black.
3. Every leaf is black.
4. If a node is red the both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

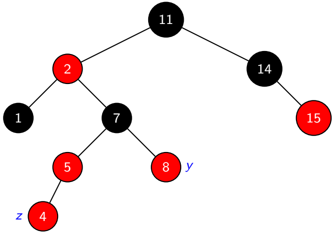
RB-INSERT-FIXUP(T, z)



Recall: Properties of a RBT:

1. Every node is either red or black.
2. Root is black.
3. Every leaf is black.
4. If a node is red the both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

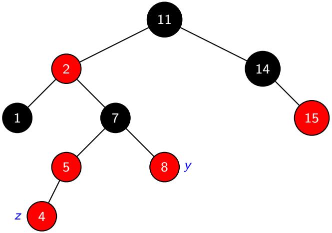
RB-INSERT-FIXUP(T, z)



Recall: Properties of a RBT:

1. Every node is either red or black.
2. Root is black.
3. Every leaf is black.
4. If a node is red the both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

RB-INSERT-FIXUP(T, z)

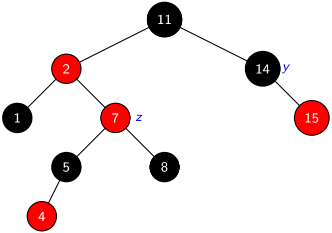


The following cases may arise:

- **Case 1:** z's uncle is red.

1. **while** $color[p[z]] = \text{RED}$

RB-INSERT-FIXUP(T, z)

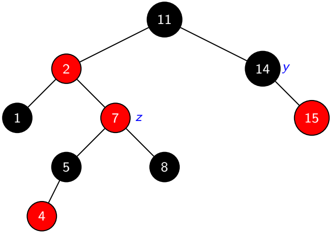


The following cases may arise:

- **Case 1:** z's uncle is red.

```
while color[p[z]] = RED
    if p[z] = left[p[pz]]
        y ← right[p[p[z]]]
        if colour[y] = RED
            colour[p[z]] ← BLACK
            colour[y] ← BLACK
            colour[p[p[z]]] ← RED
            z ← p[p[z]]
```

RB-INSERT-FIXUP(T, z)

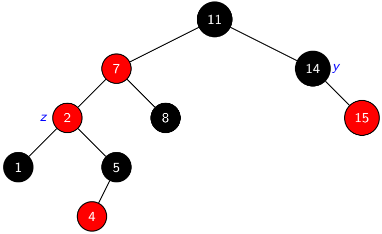


The following cases may arise:

- **Case 1:** z's uncle is red.
- **Case 2:** z's uncle is black and z is a right child.

```
while color[p[z]] = RED
    if p[z] = left[p[pz]]
        y ← right[p[p[z]]]
        if colour[y] = RED
            colour[p[z]] ← BLACK
            colour[y] ← BLACK
            colour[p[p[z]]] ← RED
            z ← p[p[z]]
```

RB-INSERT-FIXUP(T, z)

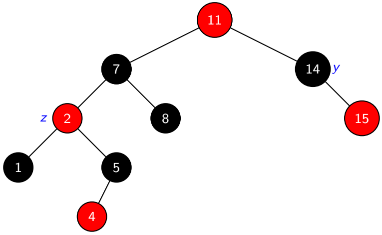


The following cases may arise:

- **Case 1:** z's uncle is red.
- **Case 2:** z's uncle is black and z is a right child.

```
while color[p[z]] = RED
    if p[z] = left[p[pz]]
        y ← right[p[p[z]]]
        if colour[y] = RED
            colour[p[z]] ← BLACK
            colour[y] ← BLACK
            colour[p[p[z]]] ← RED
            z ← p[p[z]]
        else if z = right[p[z]]
            z ← p[z]
            LEFT-ROTATE(T, z)
```

RB-INSERT-FIXUP(T, z)

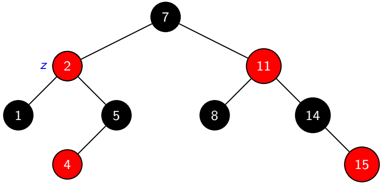


The following cases may arise:

- **Case 1:** z's uncle is red.
- **Case 2:** z's uncle is black and z is a right child.
- **Case 3:** z's uncle is black and z is a left child.
- **Note:**
  - Properties 2 is violated.
  - Properties 4 is violated for the node 11.

```
while color[p[z]] = RED
    if p[z] = left[p[pz]]
        y ← right[p[p[z]]]
        if colour[y] = RED
            colour[p[z]] ← BLACK
            colour[y] ← BLACK
            colour[p[p[z]]] ← RED
            z ← p[p[z]]
        else if z = right[p[z]]
            z ← p[z]
            LEFT-ROTATE(T, z)
            colour[p[z]] ← BLACK
            colour[p[z]] ← RED
```

RB-INSERT-FIXUP(T, z)

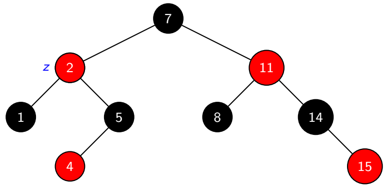


The following cases may arise:

- **Case 1:** z's uncle is red.
- **Case 2:** z's uncle is black and z is a right child.
- **Case 3:** z's uncle is black and z is a left child.
- **Note:**
  - Properties 2 is violated.
  - Properties 4 is violated for the node 11.

```
while color[p[z]] = RED
    if p[z] = left[p[pz]]
        y ← right[p[p[z]]]
        if colour[y] = RED
            colour[p[z]] ← BLACK
            colour[y] ← BLACK
            colour[p[p[z]]] ← RED
            z ← p[p[z]]
        else if z = right[p[z]]
            z ← p[z]
            LEFT-ROTATE(T, z)
            colour[p[z]] ← BLACK
            colour[p[z]] ← RED
            RIGHT-ROTATE(T, p[p[z]])
```

RB-INSERT-FIXUP(T, z)

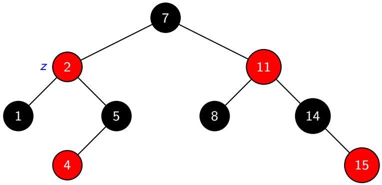


The following cases may arise:

- **Case 1:** z's uncle is red.
- **Case 2:** z's uncle is black and z is a right child.
- **Case 3:** z's uncle is black and z is a left child.
- **Note:**
  - Properties 2 is violated.
  - Properties 4 is violated for the node 11.

1. **while** $color[p[z]] = \text{RED}$
2. **if** $p[z] = left[p[pz]]$
3. $y \leftarrow right[p[p[z]]]$
4. **if** $colour[y] = \text{RED}$
5. $colour[p[z]] \leftarrow \text{BLACK}$
6. $colour[y] \leftarrow \text{BLACK}$
7. $colour[p[p[z]]] \leftarrow \text{RED}$
8. $z \leftarrow p[p[z]]$
9. **else if** $z = right[p[z]]$
10. $z \leftarrow p[z]$
11. LEFT-ROTATE(T, z)
12. $colour[p[z]] \leftarrow \text{BLACK}$
13. $colour[p[z]] \leftarrow \text{RED}$
14. RIGHT-ROTATE($T, p[p[z]]$)
15. **else** (same as Line 3 to 14 with "right" and "left" interchanged)

RB-INSERT-FIXUP(T, z)



The following cases may arise:

- **Case 1:** z's uncle is red.
- **Case 2:** z's uncle is black and z is a right child.
- **Case 3:** z's uncle is black and z is a left child.
- **Note:**
  - Properties 2 is violated.
  - Properties 4 is violated for the node 11.

```
1  while color[p[z]] = RED
2      if p[z] = left[p[pz]]
3          y ← right[p[p[z]]]
4          if colour[y] = RED
5              colour[p[z]] ← BLACK
6              colour[y] ← BLACK
7              colour[p[p[z]]] ← RED
8              z ← p[p[z]]
9          else if z = right[p[z]]
10             z ← p[z]
11             LEFT-ROTATE(T, z)
12             colour[p[z]] ← BLACK
13             colour[p[z]] ← RED
14             RIGHT-ROTATE(T, p[p[z]])
15     else (same as Line 3 to 14 with
       "right" and "left" interchanged)
16 colour[root[T]] ← BLACK
```

Thank You for your kind attention!

Books and Other Materials Consulted

1. Huffman Coding part taken from the following website.

2. Red-Black Tree part taken from Chapter 13 of the *Introduction to Algorithms* book by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Questions!!