TREES – PART III

THE HUFFMAN CODE

- Developed by David Huffman in 1952.
- Uses binary trees to compress data.
- Prefix codes are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character.

CHARACTER CODES

- Each character in a normal uncompressed text file is represented in the computer by one byte (for the venerable ASCII code) or two bytes (for the newer Unicode, which is designed to work for all languages).
- Every character requires the same number of bits.

Character	Decimal	Binary
A	65	01000000
B	66	01000001
C	67	01000010
…	…	…
X	88	01011000
Y	89	01011001
Z	90	01011010

CHARACTER CODES

- The most common approach to compress text files is to reduce the number of bits that represent the most used characters.
- In English, E is often the most common letter.
- Suppose we use just two bits for E, say 01.
- We cannot code every letter in the alphabet using only 2 bits since there are four combinations only: 00, 01, 10, 11.
- We cannot use two bits for most commonly used letters either, since these two bits can appear at the beginning of a longer code used for some other character.

CHARACTER CODES

- E → 01
- X → 01011000
- It is not clear if the initial 01 is for E or just the beginning of the code X.
- **Rule 1: No code can be prefix of any other code.**

CHARACTER CODES

- We cannot assume that the E is the most commonly used letter in every text.
- That is why we should make up a table that shows how many times each letter appears. This is called a frequency table.
- The frequency table for the message SUSIE SAYS IT IS EASY :

Character	Count
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Space	4
Linefeed	1

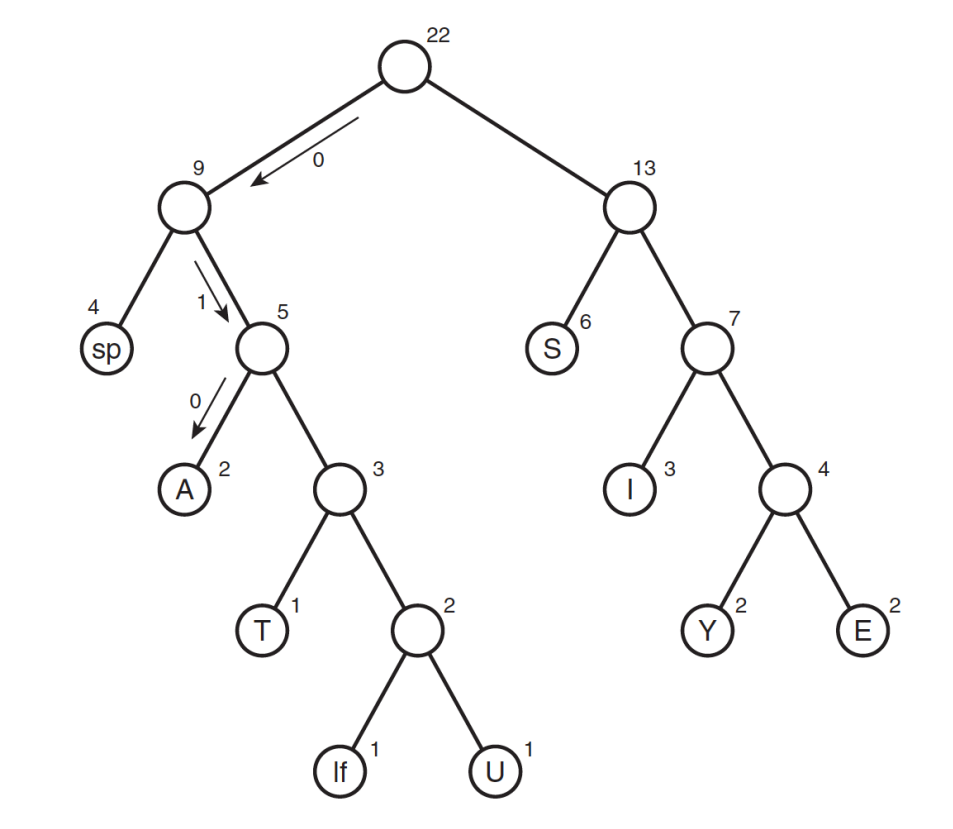
CHARACTER CODES

- We might encode the characters in the Susie message as:

Character	Code
A	010
E	1111
I	110
S	10
T	0110
U	01111
Y	1110
Space	00
Linefeed	01110

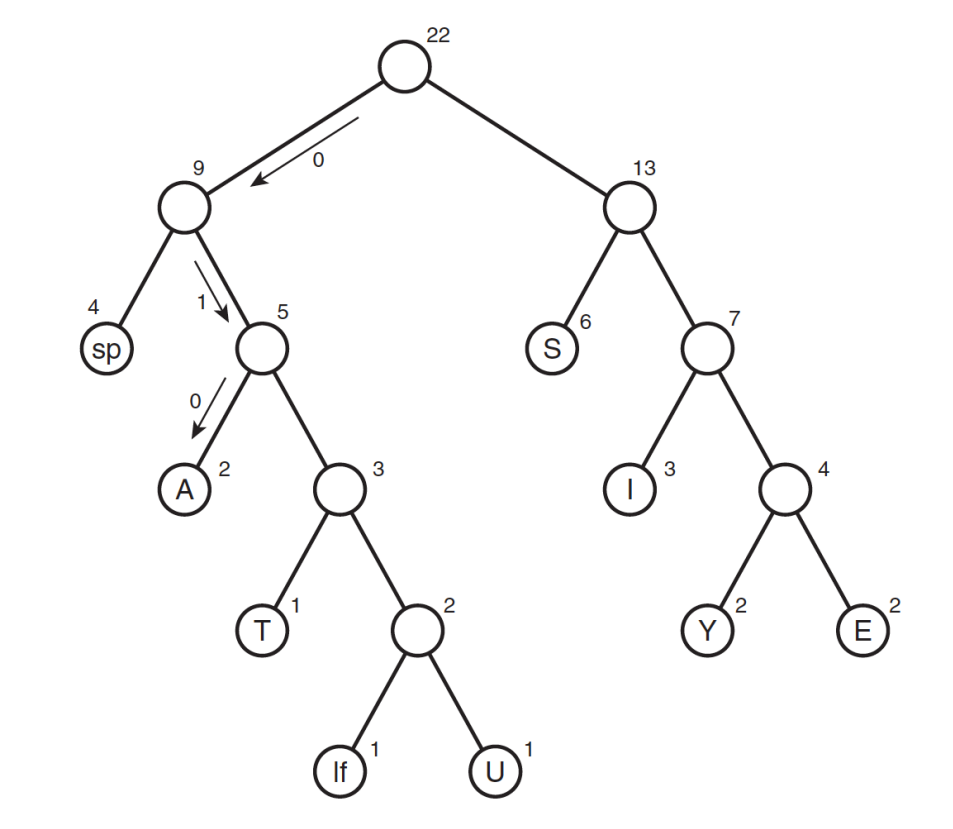
The entire message is coded as:
10 01111 10 110 1111 00 10 010 1110 10 00 110 0110 00 110 10 00 1111 010 10 1110 01110

DECODING WITH THE HUFFMAN TREE



- The characters in the message appear in the tree as the leaf nodes.
- The higher their frequency in the message, the higher they appear in the tree.
- The number outside each circle is the frequency.
- The number outside interior nodes are the sums of the frequencies of their children.

DECODING WITH THE HUFFMAN TREE



Using the tree to decode the message:

- For each character, you start at the root.
- If you see a 0 bit you go left, if you see a 1 bit you go right.

CREATING THE HUFFMAN TREE

1. Make a Node object for each character used in the message. For our Susie example that would be nine nodes. Each node has two data items: the character and that character's frequency in the message.
2. Make a tree object for each of these nodes. The node becomes the root of the tree.
3. Insert these trees in a priority queue . They are ordered by frequency, with the smallest frequency having the highest priority. That is, when you remove a tree, it's always the one with the least-used character.

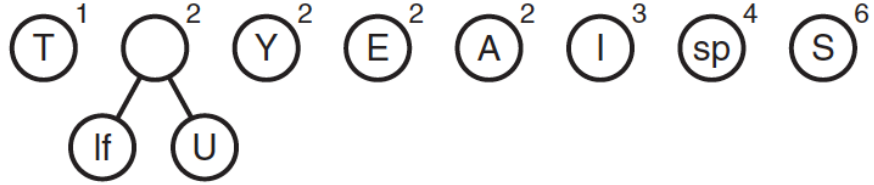
CREATING THE HUFFMAN TREE

4. Remove two trees from the priority queue, and make them into children of a new node. The new node has a frequency that is the sum of the children's frequencies; its character field can be left blank.
5. Insert this new three-node tree back into the priority queue.
6. Keep repeating steps 1 and 2. The trees will get larger and larger, and there will be fewer and fewer of them. When there is only one tree left in the queue, it is the Huffman tree and you're done.

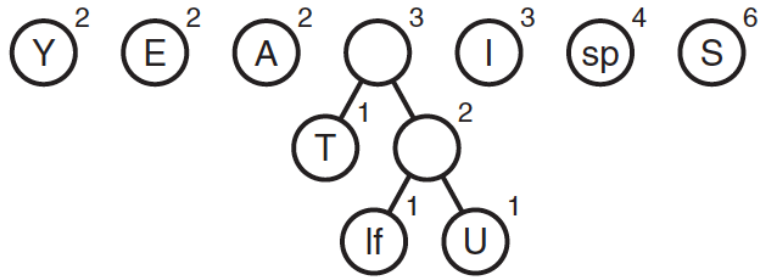
a)


If 1
U 1
T 1
Y 2
E 2
A 2
I 3
sp 4
S 6

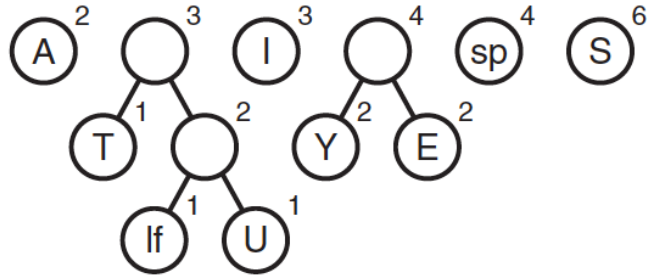
b)


T 1
2
Y 2
E 2
A 2
I 3
sp 4
S 6
If
U

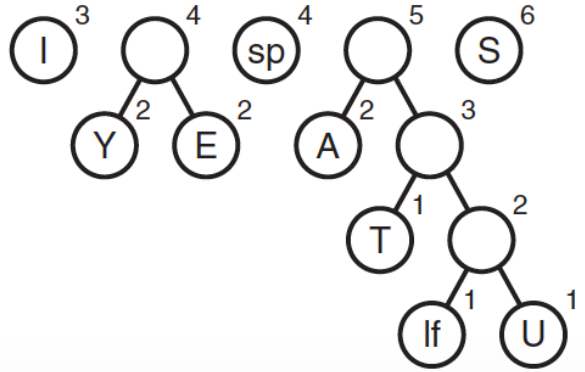
c)


Y 2
E 2
A 2
3
I 3
sp 4
S 6
T 1
2
If 1
U 1

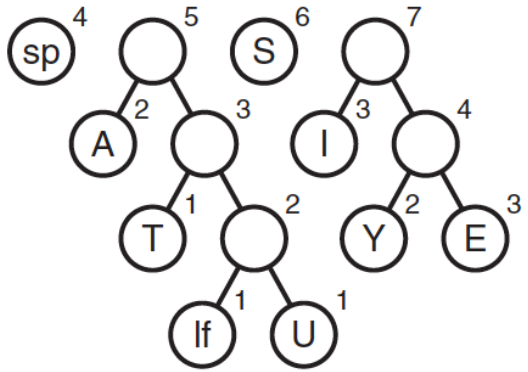
d)


A 2
3
I 3
4
sp 4
S 6
T 1
2
Y 2
E 2
If 1
U 1

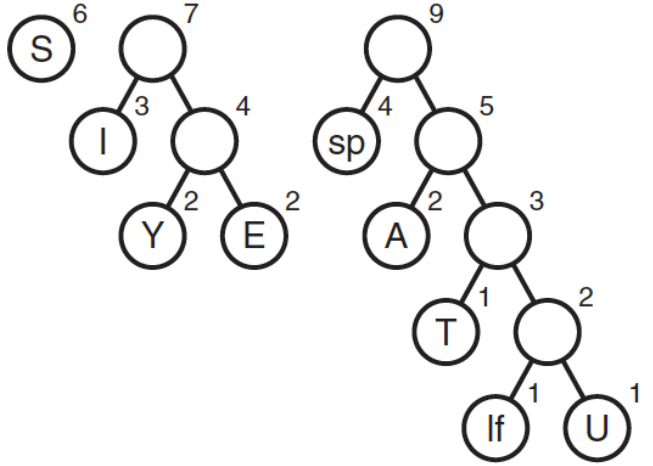
e)


I 3
4
sp 4
5
S 6
Y 2
E 2
A 2
3
T 1
2
If 1
U 1

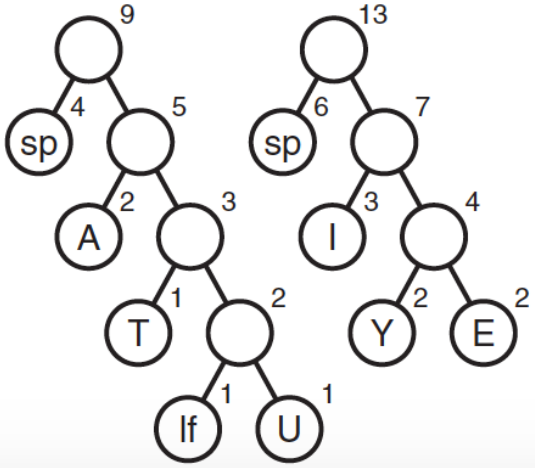
f)


sp 4
5
S 6
7
A 2
3
I 3
4
T 1
2
Y 2
E 3
If 1
U 1

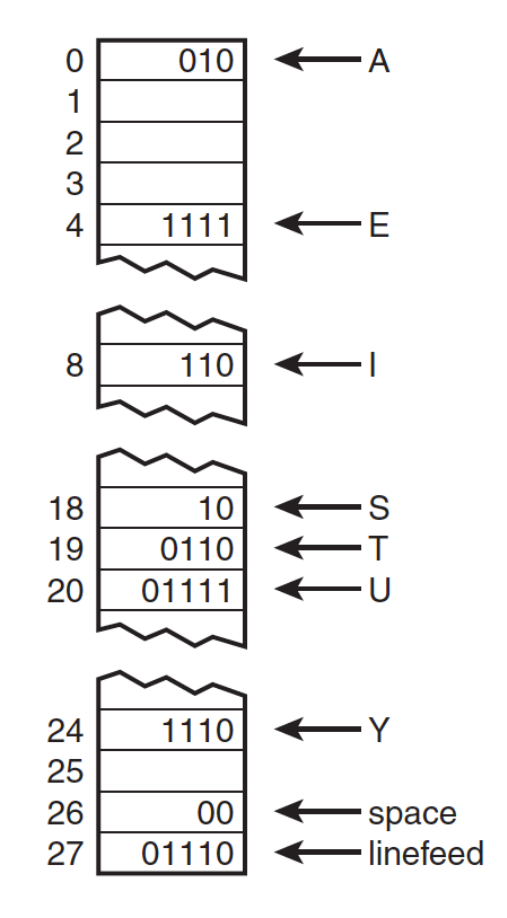
g)


S 6
7
I 3
4
9
sp 4
5
Y 2
E 2
A 2
3
T 1
2
If 1
U 1

h)


9
13
sp 4
5
sp 6
7
A 2
3
I 3
4
T 1
2
Y 2
E 2
If 1
U 1

CODING THE MESSAGE



- We start by creating a code table, which lists the Huffman code alongside each character.
- To simplify the discussion, let's assume that, instead of the ASCII code, our computer uses a simplified alphabet that has only uppercase letters with 28 characters. A is 0, B is 1, and so on up to Z, which is 25.
- We number these characters so their numerical codes run from 0 to 27.
- Not every cell contains a value, only those that appear in the message.
- For each character in the original message, we use its code as an index into the code table.
- We then repeatedly append the Huffman codes to the end of the coded message until it's complete.

CREATING HUFFMAN CODE

- The process is like decoding a message. We start at the root of the Huffman tree and follow every possible path to a leaf node.
- As we go along the path, we remember the sequence of left and right choices, recording a 0 for a left edge and a 1 for a right edge.
- When we arrive at the leaf node for a character, the sequence of 0s and 1s is the Huffman code for that character.
- We put this code into the code table at the appropriate index number.

BALANCED BINARY TREES

- A binary tree is balanced if the heights of the left and right subtrees of every node differ by at most 1.
- The efficiency of the search, insertion, and deletion operations depend on the height of the tree.
- In the best case, a binary tree of size n has a height of log n, but in the worst case, there is one node per level, resulting in a height of n.
- Thus, it would be to our advantage to try to build a binary search tree that has height log n.

BALANCED BINARY TREES

- Red-Black Trees
- AVL Trees
- 2-3 Trees

RED-BLACK TREES

- Red-black tree is a self-balancing Binary Search Tree where every node follows these rules:
  - Every node is either red or black.
  - The root is always black.
  - If a node is red, its children must be black (although the converse isn't necessarily true).
  - Every path from the root to a leaf, or to a null (None) child, must contain the same number of black nodes.
- The number of black nodes on a path from root to leaf is called the **black height**.

SIMULATION

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

AVL TREE

- The AVL tree, which was invented by G. M. **A**del'son-**V**elskii and Y. M. **L**andis in 1962, improves on the binary search tree by always guaranteeing the tree is height balanced, which allows for more efficient operations.
- With each node in an AVL tree, we associate a balance factor, which indicates the height difference between the left and right branch.
- The balance factor can be one of three states:
  - left high: When the left subtree is higher than the right subtree.
  - equal high: When the two subtrees have equal height.
  - right high: When the right subtree is higher than the left subtree.

AVL TREE

- The balance factors of the tree nodes in our illustrations are indicated by symbols:
  - > for a left high state,
  - = for the equal high state,
  - < for a right high state.
- When a node is out of balance, we will use either << or >> to indicate which subtree is higher.

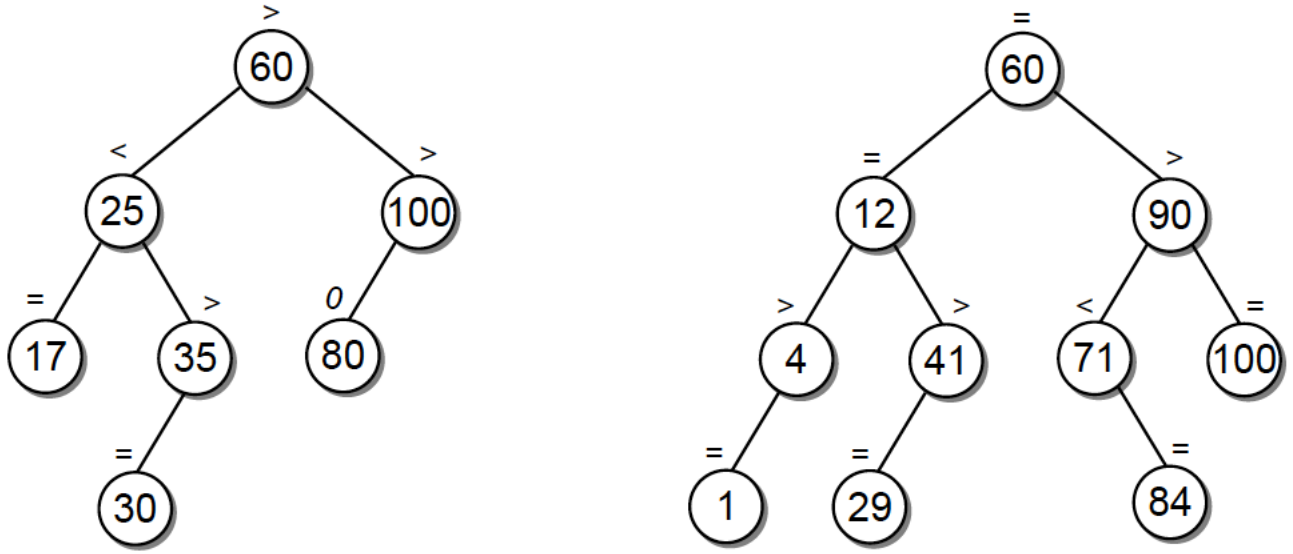


Figure 14.14: Examples of balanced binary search trees.

AVL TREE - INSERTIONS

- The search and traversal operations are the same with an AVL tree as with a binary search tree.
- The insertion and deletion operations have to be modified in order to maintain the balance property of the tree as new keys are inserted and existing ones removed.
- This height is sufficient for providing O(log n) time operations even in the worst case.

AVL TREE - INSERTIONS

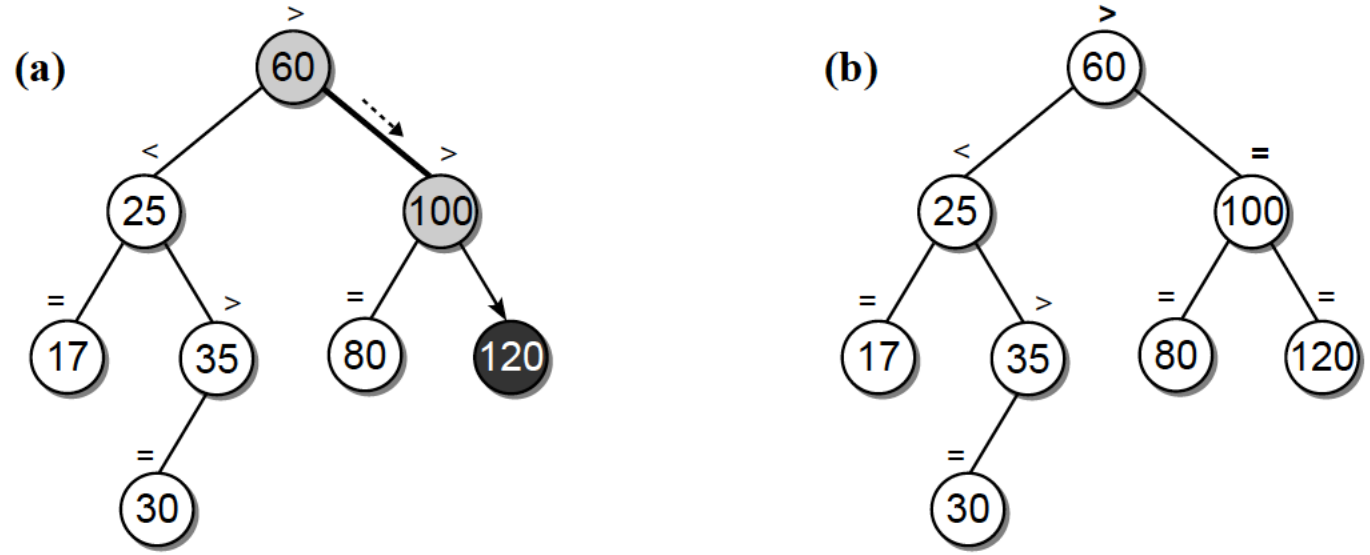


Figure 14.15: A simple insertion into an AVL tree: (a) with key 120 inserted; and (b) the new balance factors.

AVL TREE - INSERTIONS

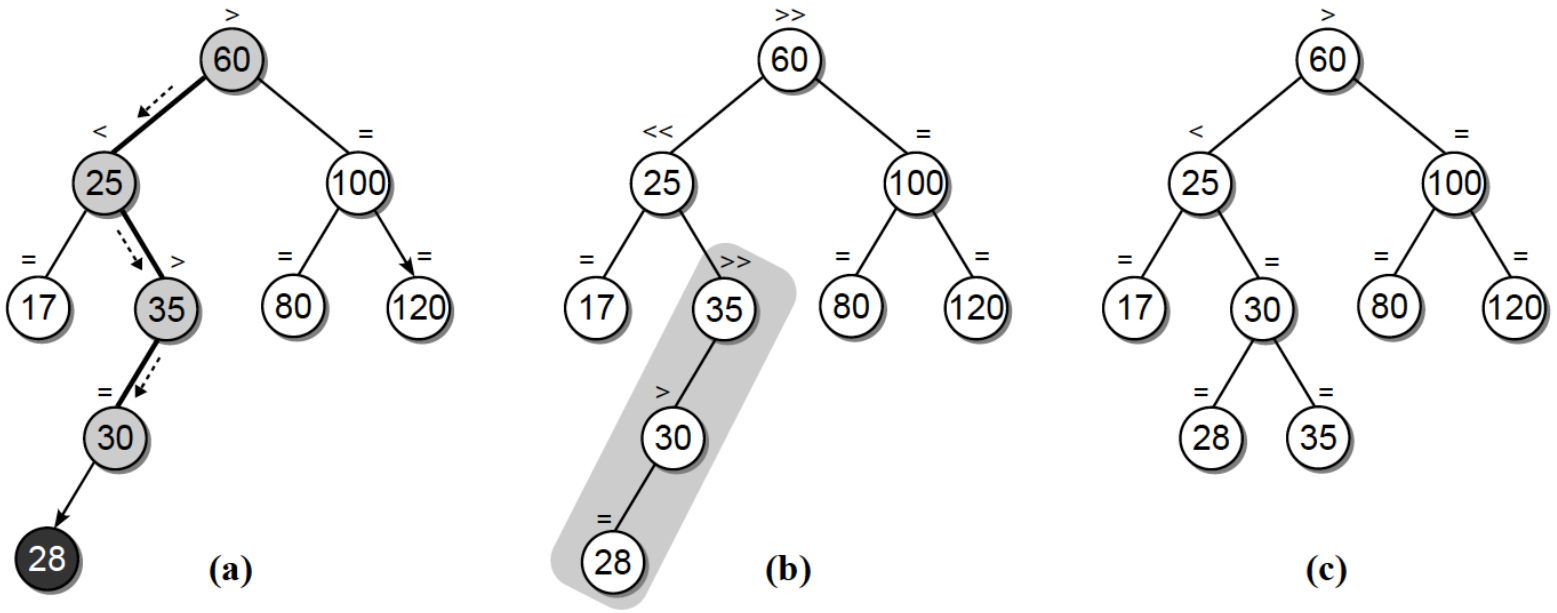
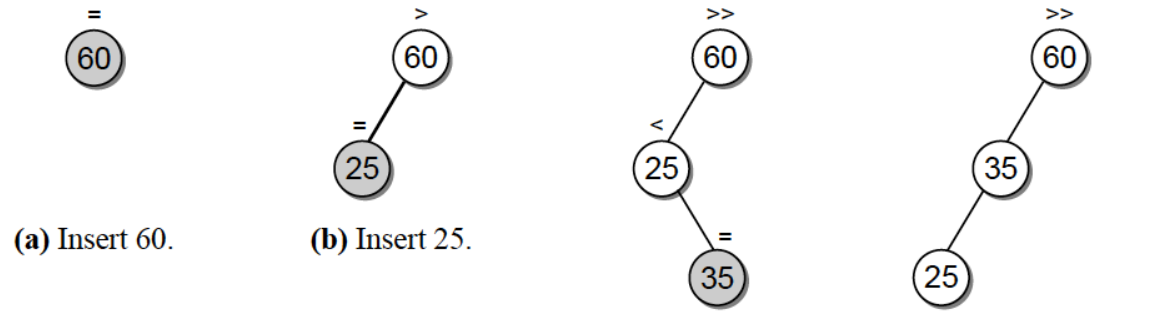
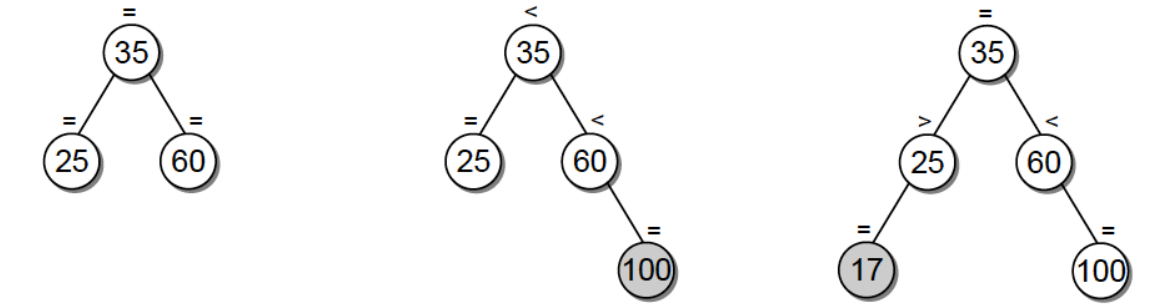


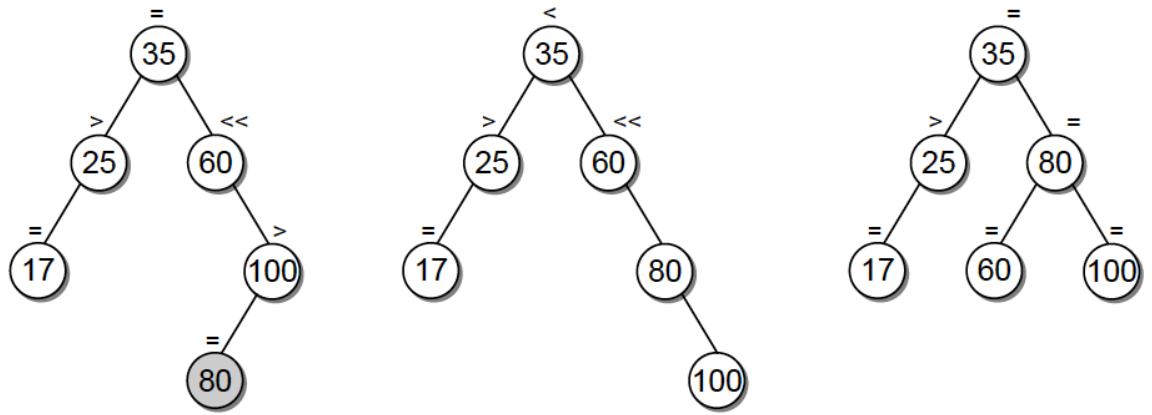
Figure 14.16: An insertion that causes the AVL tree to become unbalanced: (a) the new key is inserted; (b) the balance factors showing an out-of-balance tree; and (c) the subtree after node 35 is rearranged.



(a) Insert 60. **(b)** Insert 25. **(c)** Insert 35. **(d)** Left rotate at 25.



(e) Right rotate at 60. **(f)** Insert 100. **(g)** Insert 17.



(h) Insert 80. **(i)** Right rotate at 100. **(j)** Left rotate at 60.

Figure 14.20: Building an AVL tree from the list of keys `[60, 25, 35, 100, 17, 80]`. Each tree shows the results after performing the indicated operation.

AVL TREE - DELETION

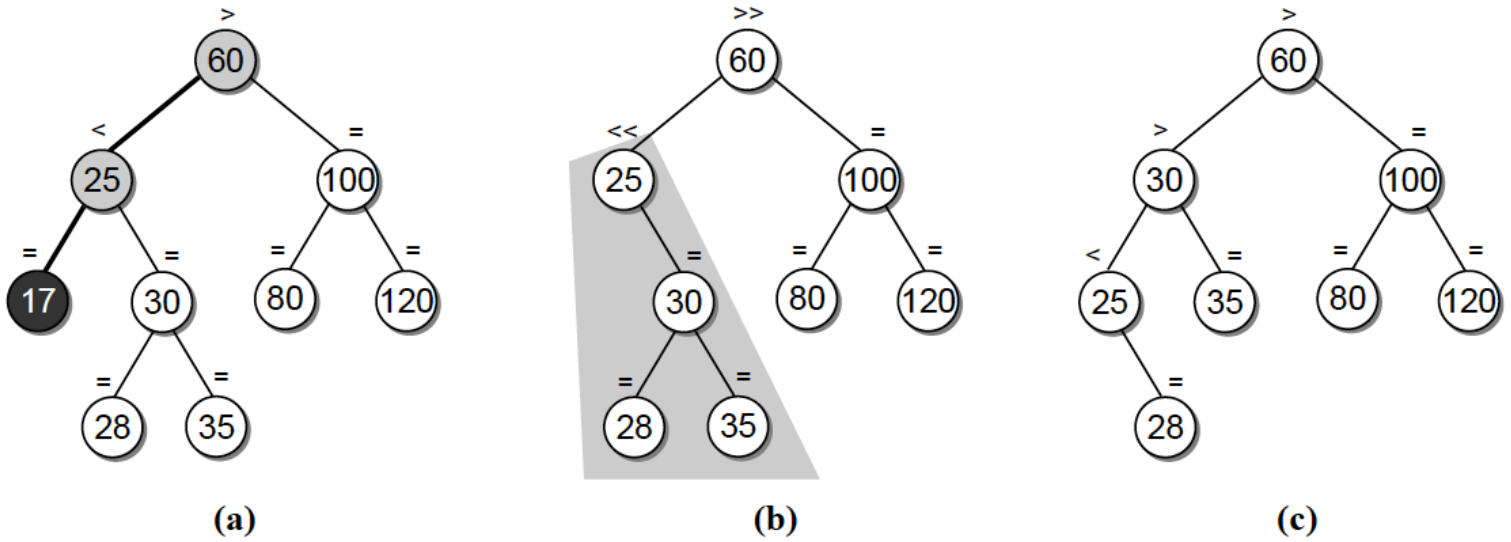


Figure 14.21: A deletion that causes the AVL tree to become unbalanced: (a) the node is located; (b) the balance factors change showing an out-of-balance tree; and (c) the tree after a left rotation.

2-3 TREE

- The 2-3 tree is a multi-way search tree that can have up to three children.
- It provides fast operations that are easy to implement.
- The tree gets its name from the number of keys and children each node can contain.

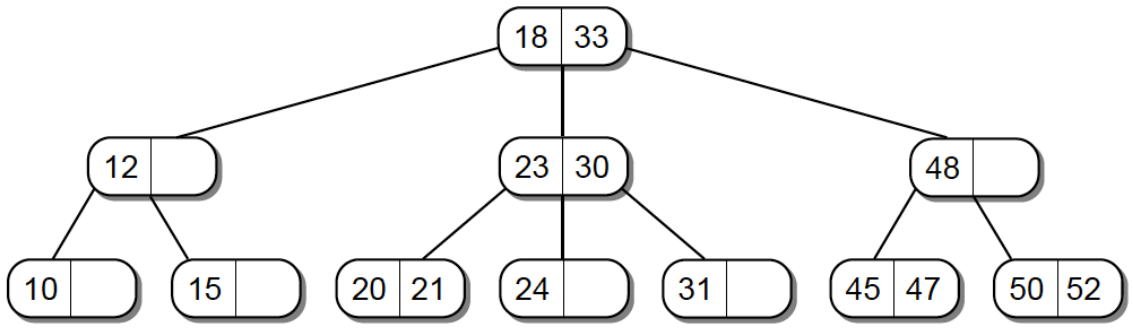


Figure 14.22: A 2-3 tree with integer search keys.

2-3 TREE

- A 2-3 tree is a search tree that is always balanced and whose shape and structure is defined as follows:
  - Every node has capacity for one or two keys (and their corresponding payload), which we term key one and key two.
  - Every node has capacity for up to three children, which we term the left, middle, and right child.
  - All leaf nodes are at the same level.
  - Every internal node must contain two or three children. If the node has one key, it must contain two children; if it has two keys, it must contain three children.

2-3 TREE

- All keys less than the first key of node V are stored in the left subtree of V .
- If the node has two children, all keys greater than the first key of node V are stored in the middle subtree of V .
- If the node has three children: (1) all keys greater than the first key of node V but less than the second key are stored in the middle subtree of V ; and (2) all keys greater than the second key are stored in the right subtree.

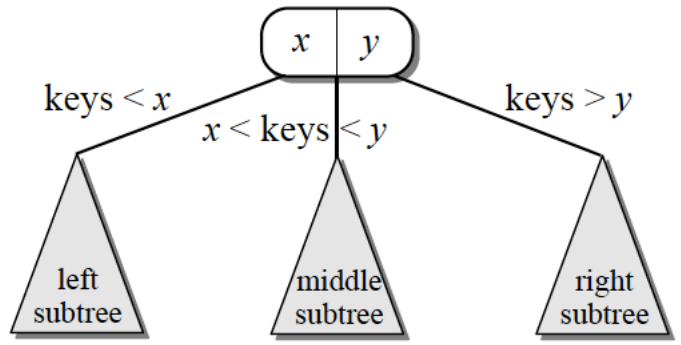


Figure 14.23: A search property of a 2-3 tree.

2-3 TREE - INSERTIONS

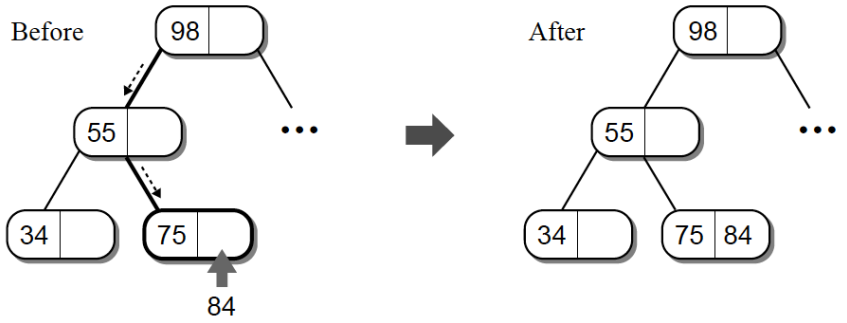


Figure 14.25: Inserting key 84 into a 2-3 tree with space available in the leaf node.

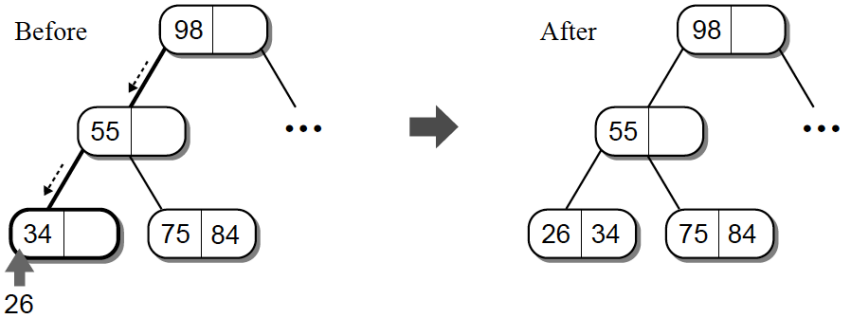


Figure 14.26: Inserting key 26 into a 2-3 tree with space available in the leaf node.

2-3 TREE - INSERTIONS

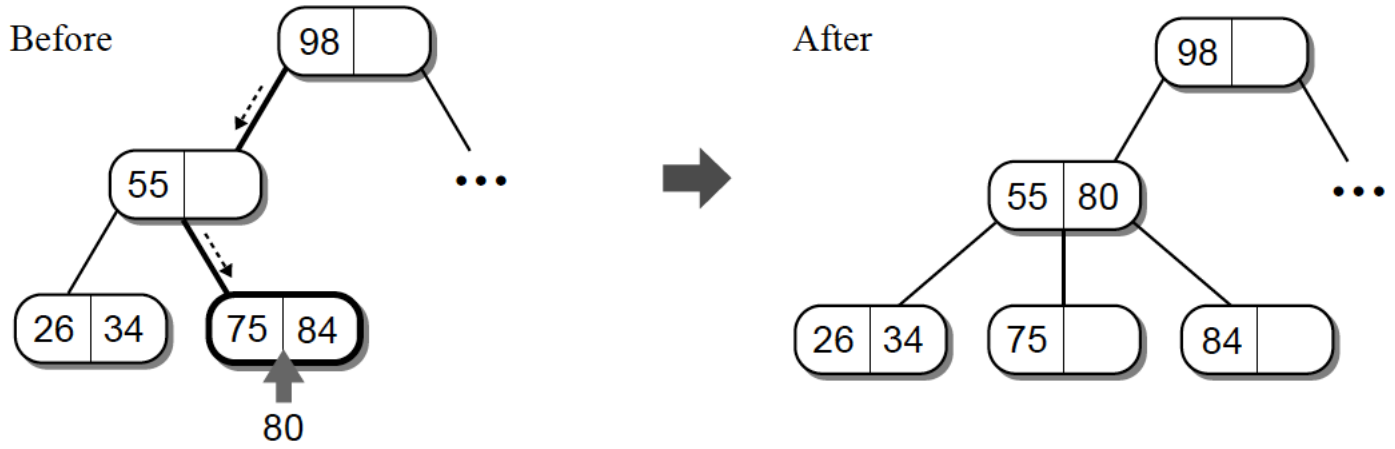


Figure 14.27: Inserting a key into a 2-3 tree with a full leaf node.