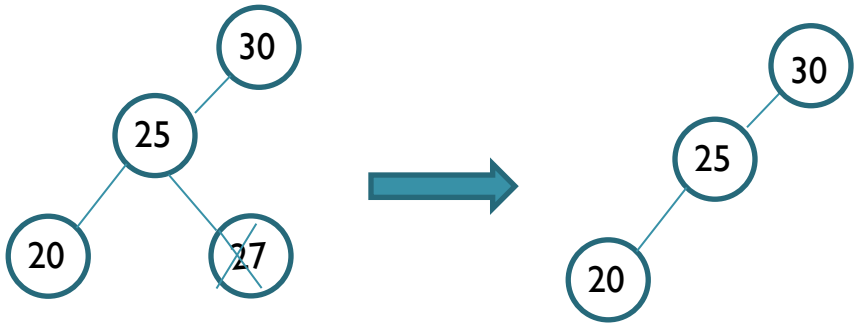
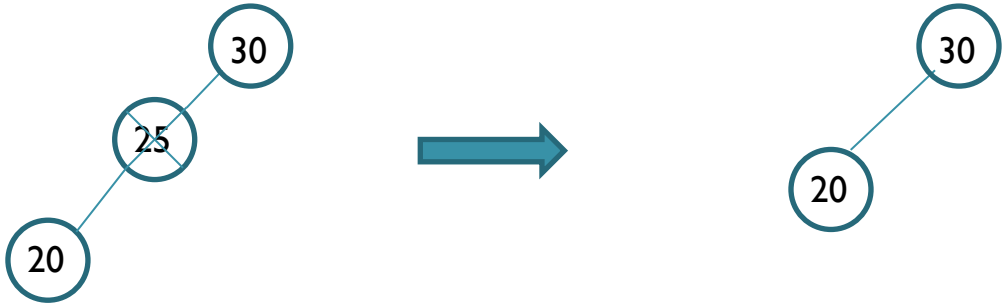
Binary Trees (contd.)

Dictionary operations for BST - delete

(a) If node to be removed is a leaf, we can just remove it from the tree

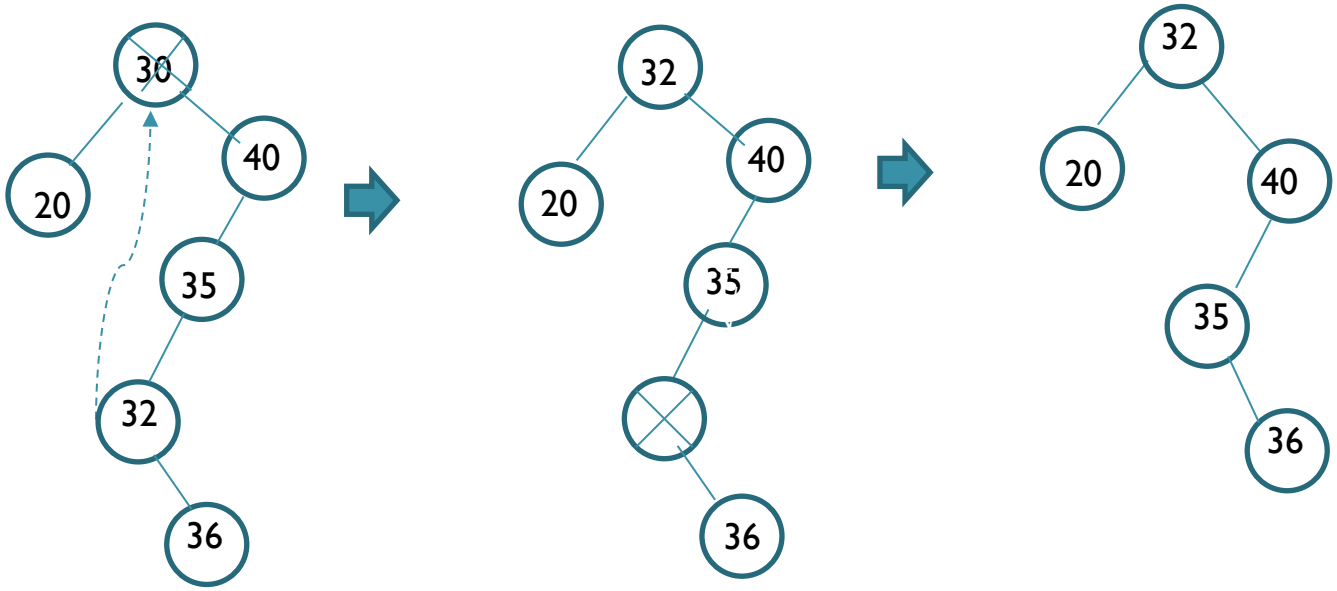


(b) If a node has only one child, then make the child the child of its parent



Dictionary operations for BST - delete

(c) when node to be removed has 2 children (replace it with inorder successor or minimum key in right subtree i.e. left most node)



remove() implementation

```
/** Remove a record from the tree.
    @param k Key value of record to remove.
    @return The record removed, null if there is none. */
public E remove(Key k) {
  E temp = findhelp(root, k);   // First find it
  if (temp != null) {
    root = removehelp(root, k); // Now remove it
    nodecount--;
  }
  return temp;
}

private BSTNode<Key,E> getmin(BSTNode<Key,E> rt) {
  if (rt.left() == null) return rt;
  return getmin(rt.left());
}

private BSTNode<Key,E> deletemin(BSTNode<Key,E> rt) {
  if (rt.left() == null) return rt.right();
  rt.setLeft(deletemin(rt.left()));
  return rt;
}
```

Source : textbook

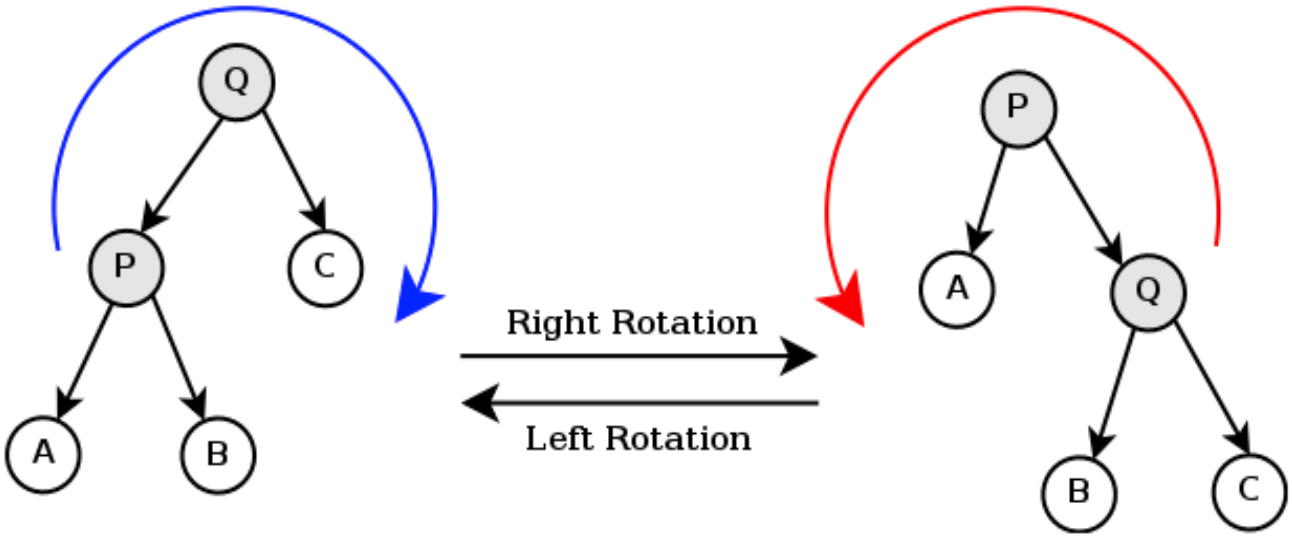
```
/** Remove a node with key value k
    @return The tree with the node removed */
private BSTNode<Key,E> removehelp(BSTNode<Key,E> rt,Key k) {
  if (rt == null) return null;
  if (rt.key().compareTo(k) > 0)
    rt.setLeft(removehelp(rt.left(), k));
  else if (rt.key().compareTo(k) < 0)
    rt.setRight(removehelp(rt.right(), k));
  else { // Found it
    if (rt.left() == null) return rt.right();
    else if (rt.right() == null) return rt.left();
    else { // Two children
      BSTNode<Key,E> temp = getmin(rt.right());
      rt.setElement(temp.element());
      rt.setKey(temp.key());
      rt.setRight(deletemin(rt.right()));
    }
  }
  return rt;
}
```

Source : textbook

Balanced Binary Search Trees

- Satisfies some **height balancing property** at every node of the tree
- For example in AVL tree, balance factor at every node is $-1, 0\ or\ 1$
- Recursive structure
- Height balancing property typically guarantees logarithmic bounds on tree height
- Insertions and removals restructure trees to guarantee this property with minimum overhead
- They involve **rotation** operations.

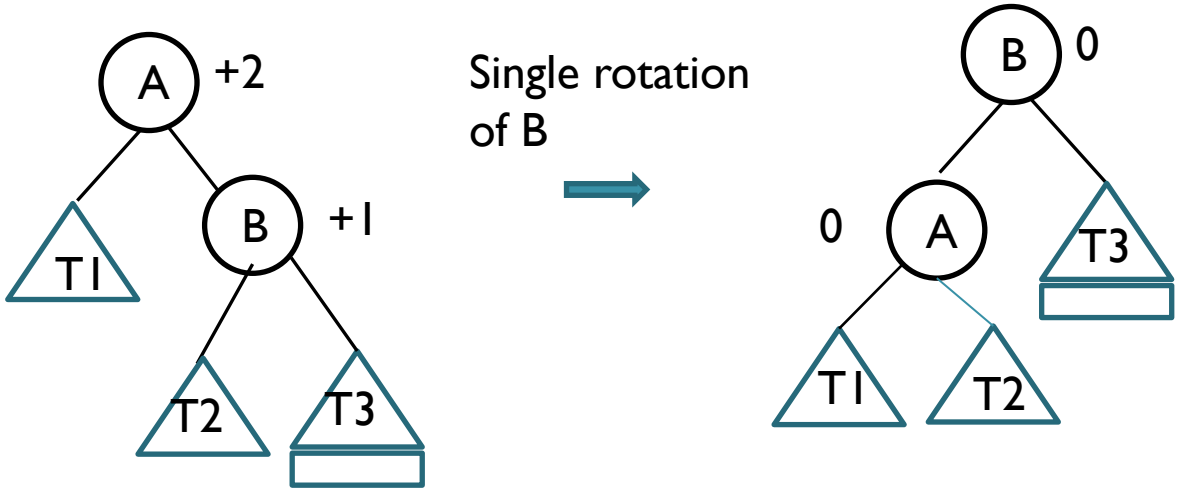
Left/Right rotations



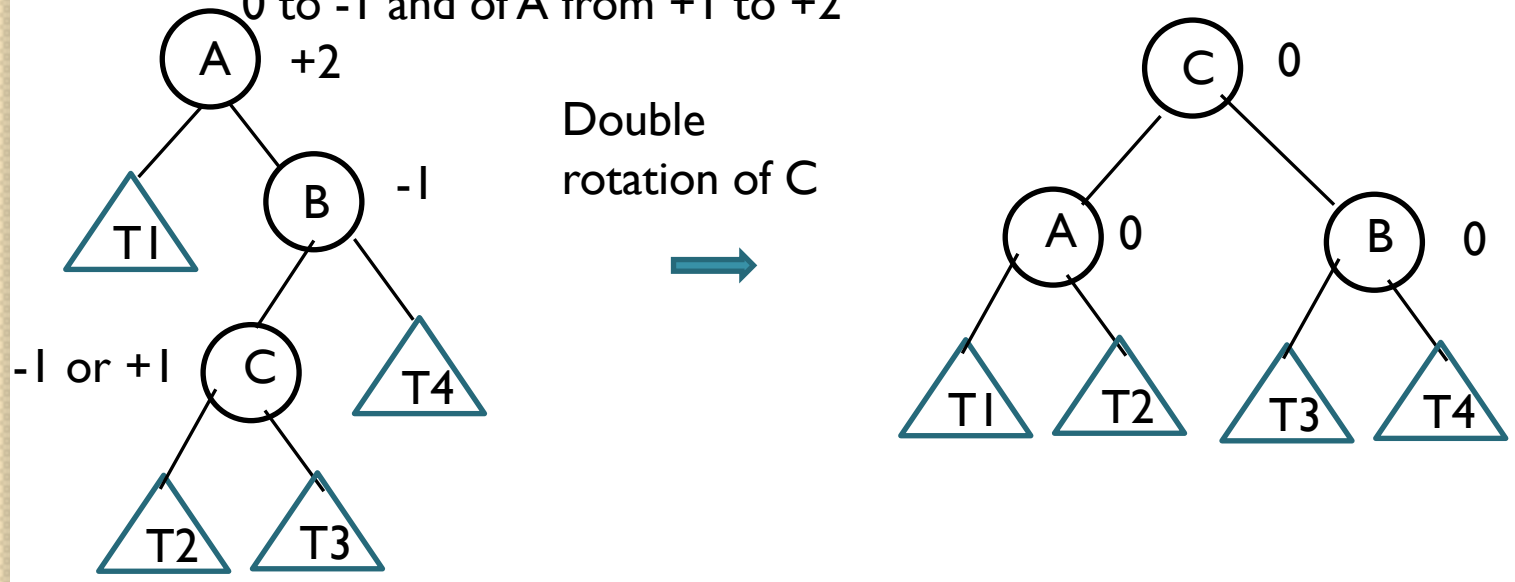
Left rotation:
Q = P.right
P.right = Q.left // B
Q.left= P
(also should return Q to parent to adjust its child pointer and its parent)

Right rotation:
P = Q.left
Q.left = P.right //B
P.right = Q
(also should return P to parent to adjust its child pointer and its parent)

Case a: Insertion into T3 changes BF of B rom 0 to +1 and of A from +1 to +2



Case b: Insertion into T2 or T3 changes BF of B from 0 to -1 and of A from +1 to +2



Max-Priority Queue ADT

Reference: "Data Structures and Algorithm Analysis", C. Shaffer, pp. 170–177.

- Allows a totally ordered set of elements to be stored in such a way that "maximum" element can be extracted efficiently – self-reorganizing structure
- Useful for task scheduling, efficient sorting
- Operations:

  insertElement(e) – insert element in queue

  removeMax() – remove and return largest

  maxElement() – return max element
- In a min-PQ, minimum elements are of interest
- In Java PriorityQueue<E> is a class based on unbounded heap

PQ – simple array implementation

- Two approaches:

  (a) Keep array unordered

  (b) Keep array sorted at all times

- Unordered array

  (a) insertElement(E) – add element to end of array – O(1) time

  (b) maxElement() – O(n) time even in best-case

  (c) removeMax() – O(n) time even in best-case

  basis of **Selection Sort** $O(n^2)$ even in best-case

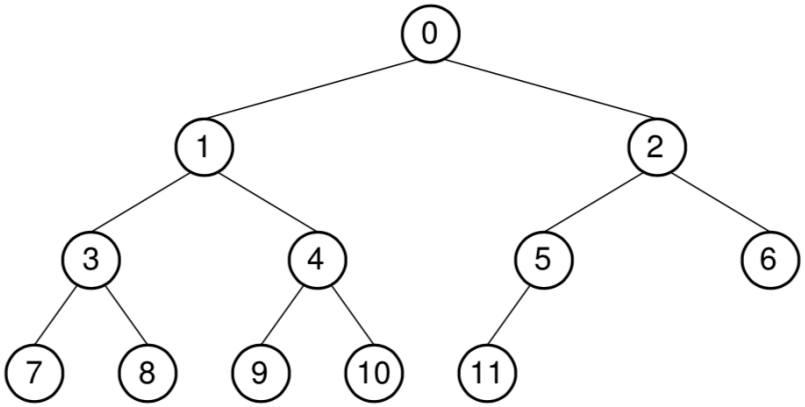
- Sorted array

(a) insertElement(E) - find position to insert and shift elements to right – O(1) time best-case and O(n) time in worst-case

(b) maxElement(E) – O(1) time

(c) removeMax() – O(1) time (basis of **Insertion Sort** $O(n^2)$ in worst-case and O(n) time in best-case)

Binary Tree array implementation



(a)

Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

(b)

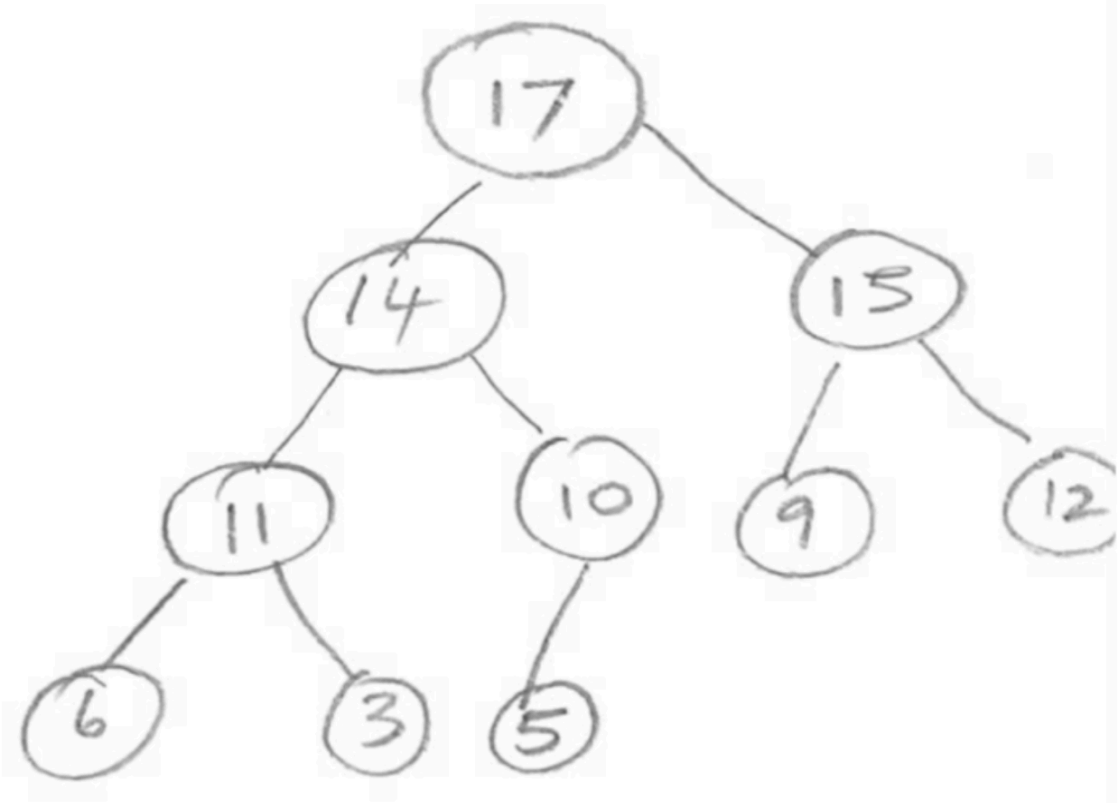
Figure 5.12 A complete binary tree and its array implementation. (a) The complete binary tree with twelve nodes. Each node has been labeled with its position in the tree. (b) The positions for the relatives of each node. A dash indicates that the relative does not exist.

- $\text{Parent}(r) = \lfloor (r-1)/2 \rfloor$ if $r \neq 0$.
- $\text{Left child}(r) = 2r + 1$ if $2r + 1 < n$.
- $\text{Right child}(r) = 2r + 2$ if $2r + 2 < n$.

Max-PQ Binary heap

- It is a complete binary tree with array implementation. In the tree we fill every level from left to right before proceeding to next level
- All elements stored in internal nodes only. Ignore external nodes in discussion.
- The element at a node (except root) is always <= element stored in its parent node – **heap property**
- **Theorem** 1 : Root element of a subtree is the maxiimum element in that subtree (use induction)
- Recursive structure
- **Theorem 2** :The height of an n-element heap is at most [log n] . Note # of nodes of a heap of height h is at least 2^h
- Easy to see that **maxElement**() takes O(1) time.

Max heap example



Array implementation

17	14	15	11	10	9	12	6	3	5

Max Heap Implementation

```
import java.util.ArrayList;

/** Max-heap implementation */
public class MaxHeap<E extends Comparable<E>> {
  private ArrayList<E> Heap;
  private int n; // Number of things in heap

  /** Constructor for initially empty heap */
  public MaxHeap() {
    Heap = new ArrayList<E>();
    n = 0;
  }

/** @return Current size of the heap */
public int heapsize() { return n; }

public boolean isEmpty() { return n == 0; }
```

Max Heap Implementation

```
  /** @return True if pos a leaf position, false otherwis
  public boolean isLeaf(int pos) {
    return (pos >= n / 2) && (pos < n);
  }
```

```
/** @return Position for left child of pos */
public int leftchild(int pos) {
  assert pos < n/2 : "Position has no left child";
  return 2*pos + 1;
}

/** @return Position for right child of pos */
public int rightchild(int pos) {
  assert pos < (n-1)/2 : "Position has no right child";
  return 2*pos + 2;
}

/** @return Position for parent */
public int parent(int pos) {
  assert pos > 0 : "Position has no parent";
  return (pos-1)/2;
}
```

Source : Dr. Calvin & textbook

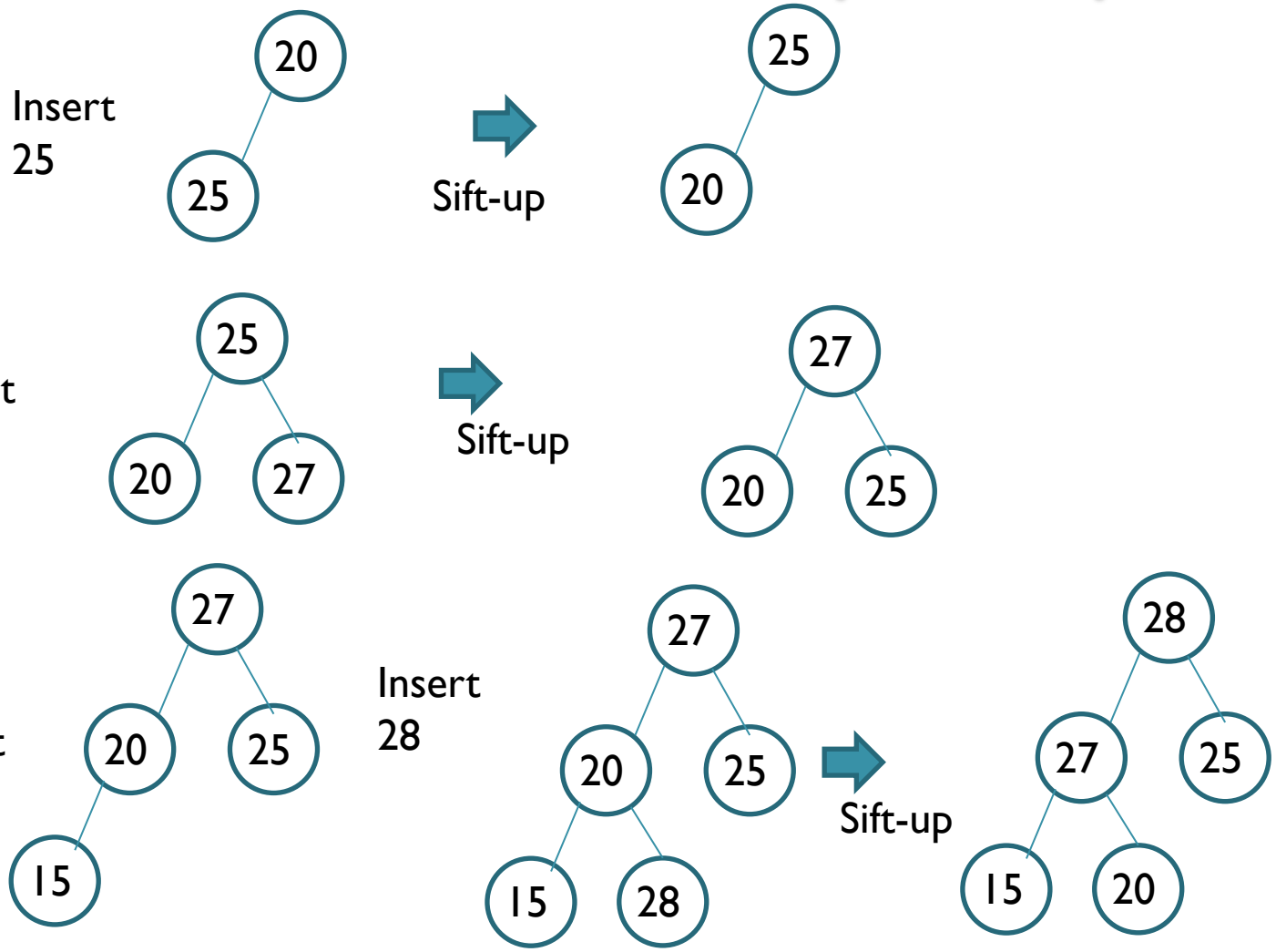
Max Heap Implementation

```
/** Insert val into heap */
public void insert(E val) {
  Heap.add(val); // Start at end of heap
  int curr = n;
  n++;
  // Now sift up until curr's parent's key > curr's key
  while ((curr != 0)
    &&(Heap.get(curr).compareTo(Heap.get(parent(curr))))
    swap(curr, parent(curr));
    curr = parent(curr);
  }
}
```

Insert() complexity is O(log n)

```
/** Heapify contents of Heap */
public void buildheap() {
  for (int i = n / 2 - 1; i >= 0; i--)
    siftdown(i);
}
```

Insert into max-heap example



Max Heap Implementation

```
/** Put element in its correct place */
private void siftdown(int pos) {
  assert (pos >= 0) && (pos < n) : "Illegal heap positio
  while (!isLeaf(pos)) {
    int j = leftchild(pos);
    if ((j < (n - 1)) &&
        (Heap.get(j).compareTo(Heap.get(j + 1)) < 0))
      j++; // j is now index of child with greater value
    if (Heap.get(pos).compareTo(Heap.get(j)) >= 0)
      return;
    swap(pos, j);
    pos = j; // Move down
  }
}
```

Source : Dr. Calvin & textbook

Max Heap Implementation

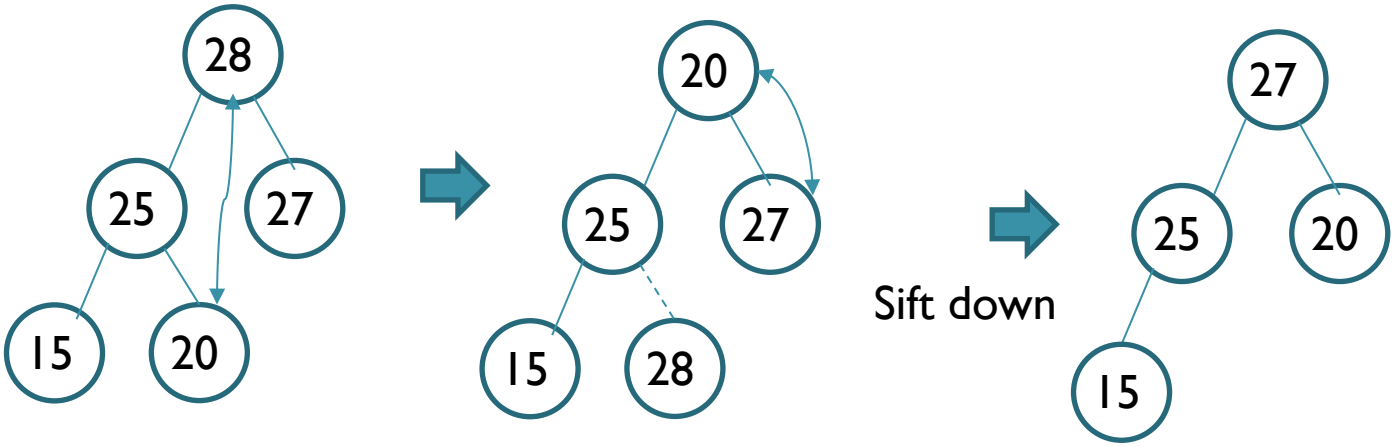
```
  public E removeMax() {
    assert n > 0 : "Removing from empty heap";
    swap(0, --n); // Swap maximum with last value
    if (n != 0) // Not on last element
      siftdown(0); // Put new heap root val
                   //  in correct place
    return Heap.get(n);
  }

  private void swap(int i, int j) {
    E temp = Heap.get(j);
    Heap.set(j, Heap.get(i));
    Heap.set(i, temp);
  }
}
```

removeMax complexity is O(log n)

Source : Dr. Calvin & textbook

removemax example



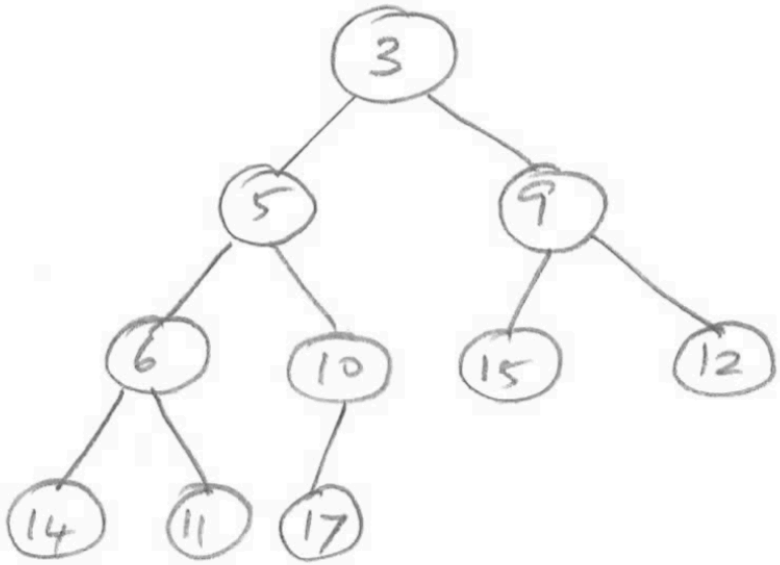
removeMax()

28 is returned by max and not in heap anymore

Heap Building Analysis

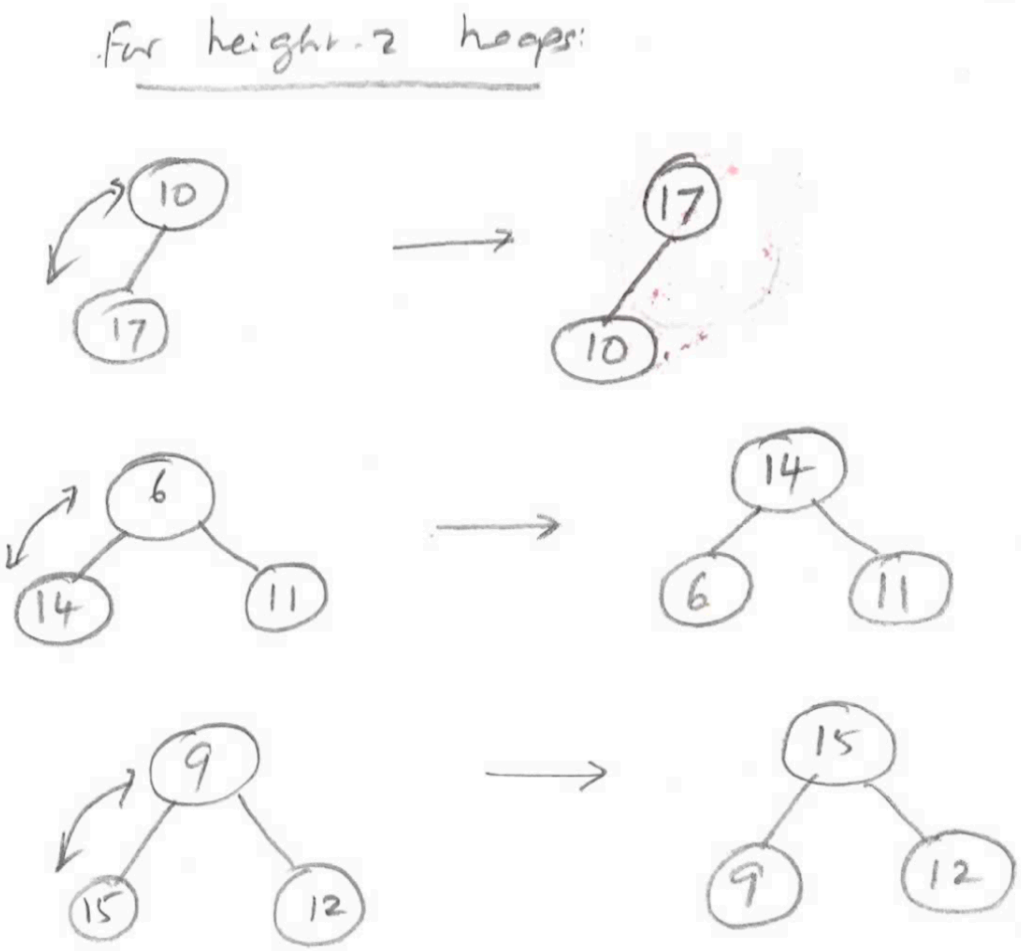
- Insert into the heap one value at a time:
  - Sift each new value up the tree to where it belongs
  - $$\sum_{i=1}^{n} \log(i) = \Theta(n \log n)$$
- Starting with full array, work from bottom up
  - Since nodes in bottom half form a heap, just need to process from midpoint of array down.
  - At most $\lceil n/2^{h+1} \rceil$ nodes of height h.
  - $$\sum_{h=1}^{\lg n} \lceil n/2^{h+1} \rceil O(h) = \Theta(n)$$

Heapify example (1)

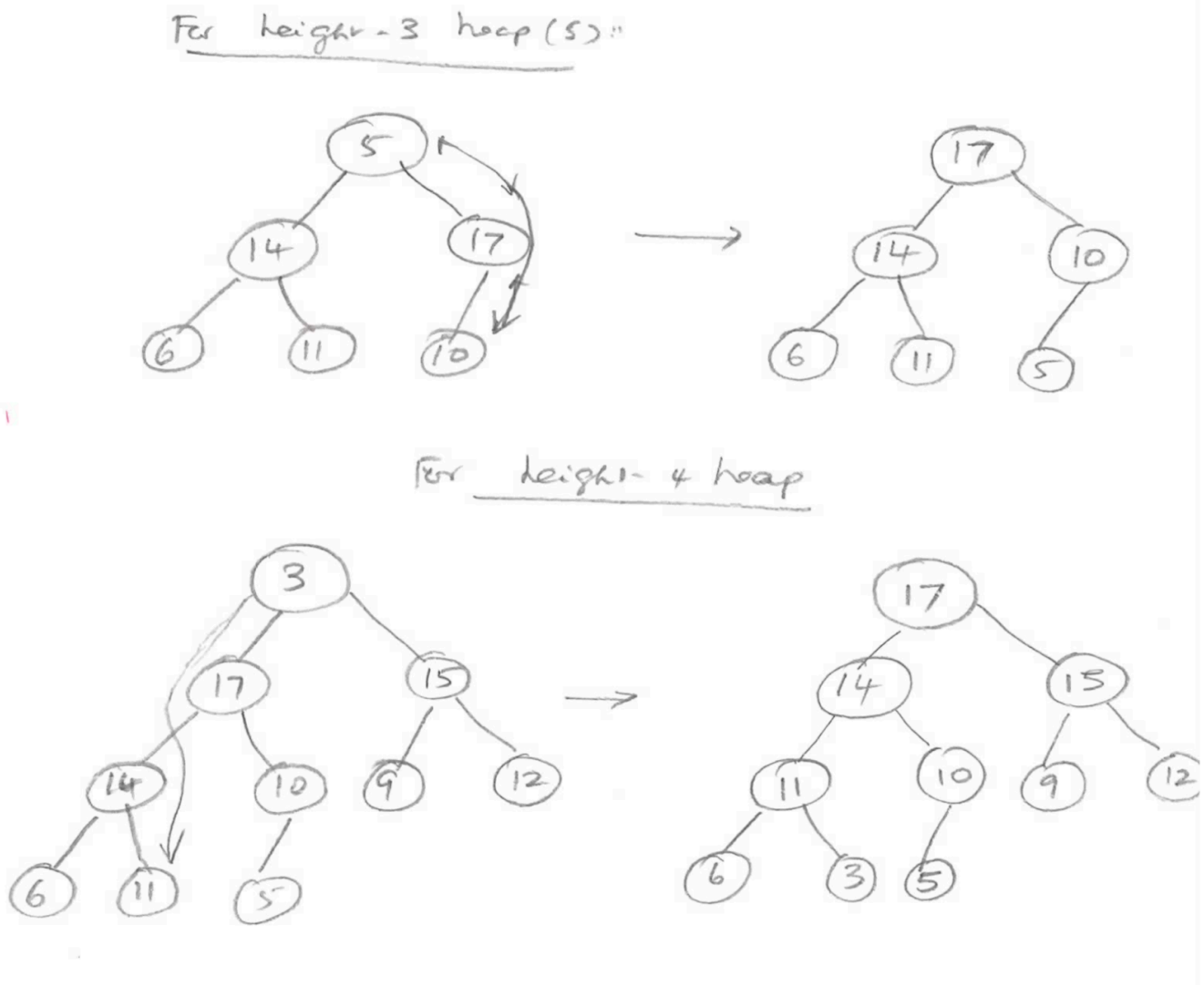


MAX-HEAPs of height 1

Heapify example (2)



Heapify example (3)



Huffman coding

- A prefix code is an encoding scheme where a code for one character cannot be a prefix of code for another character

Needed to avoid ambiguity during decoding

Example : 011 and 01110 cannot be part of a prefix code

- Prefix code can be represented as a binary tree where

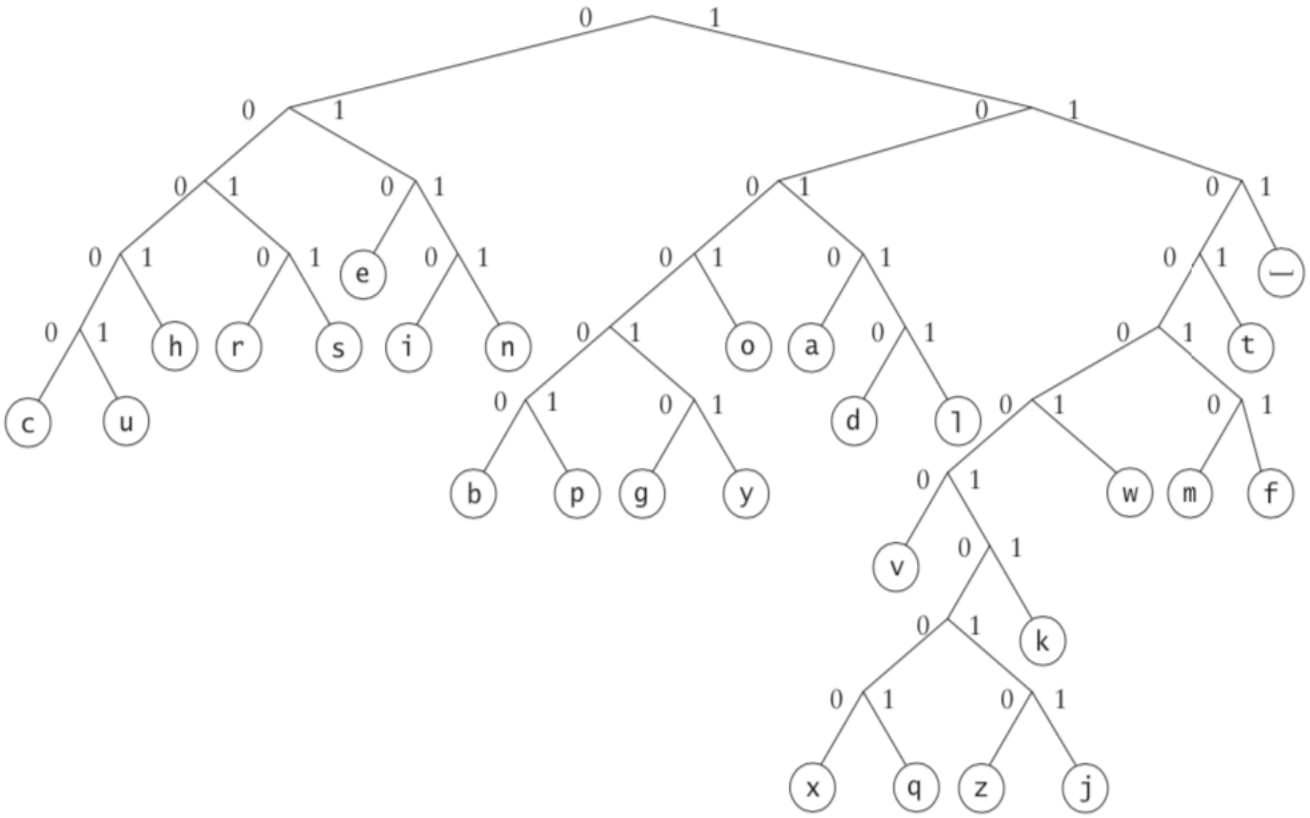
(a) leaves represent characters to be encoded

(b) edges from a node are labeled with 0 and 1 respectively.

(c) binary code for a character is obtained by concatenating edge labels in the path from root to the corresponding leaf.

- Easy to show it is a prefix-code.
- Minimizes the expected length of compressed string saving space. Huffman coding base compression algorithms are used as part of JPEG and MP3.
- Achieves by using shorter codes for high frequency characters and longer ones for low frequency characters.

Huffman Tree Based on Frequency of Letters in English Text



Source : Elliot Koffman et. al

Binary coding optimization problem

Find a tree that minimizes expected path length

$$\min_{T} \sum_{c \in C} freq(c) * \ d_T(c)$$

where

$freq(c)$ is the frequency of character c in the character set C and

$d_T(c)$ is the depth (number of edges in the path from root) of leaf node corresponding to character c in the binary tree T

Source : Dr. Calvin & textbook

Letter	Freq	Code	Bits
C	32	1110	4
D	42	101	3
E	120	0	1
K	7	111101	6
L	42	110	3
M	24	11111	5
U	37	100	3
Z	2	111100	6

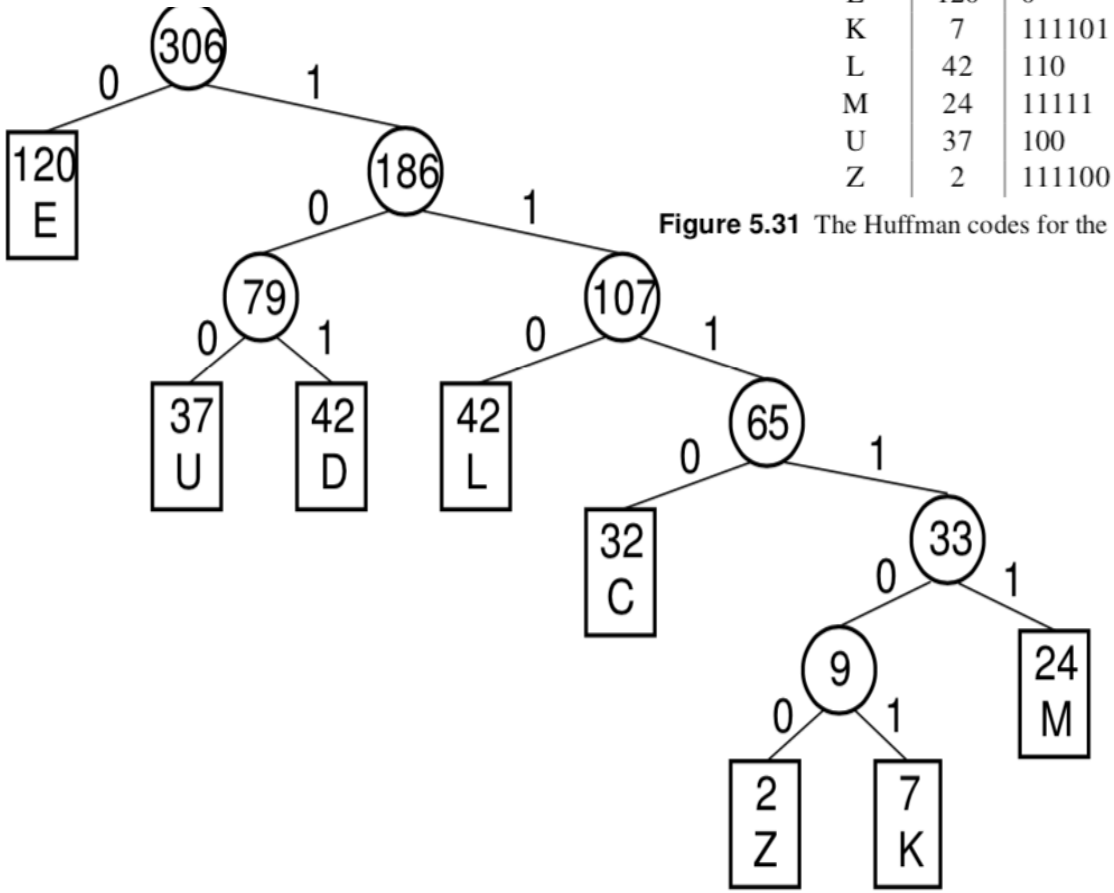


Figure 5.31 The Huffman codes for the letters of Figure 5.24.

Source : textbook