Lecture - 10
on
Data Structures(Trees)

Properties of Heaps

- Heaps are binary trees that are ordered from bottom to top, so that a traversal along any leaf-to-root path will visit the keys

  in ascending order: max heaps : **key(parent) >= key(child)]**

  in descending order: min heaps : **key(parent) $\leq$ key(child)**

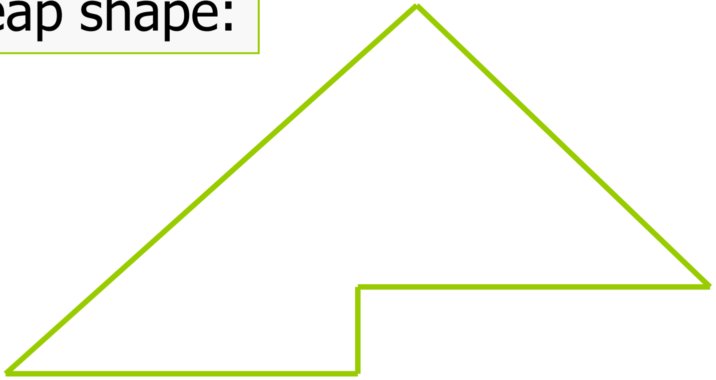
- Heaps are (1) to implement priority queues,

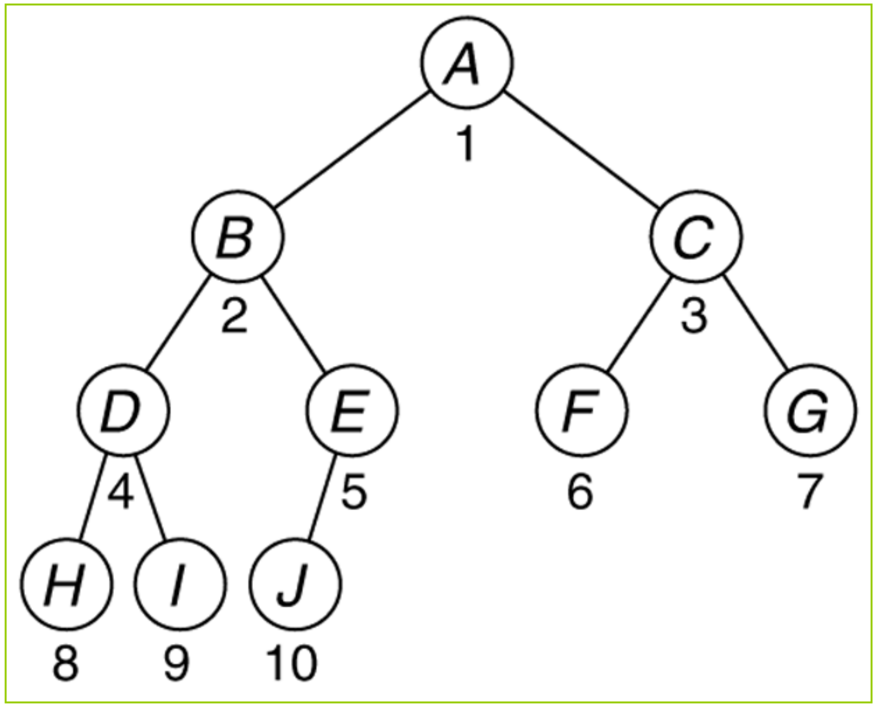
  (2) to implement the Heap sort algorithm.

- The parent of node number i is numbered (i/2) and its two children are numbered 2i and (2i + 1).
- A heap is a complete binary tree.

Heap

Heap shape:





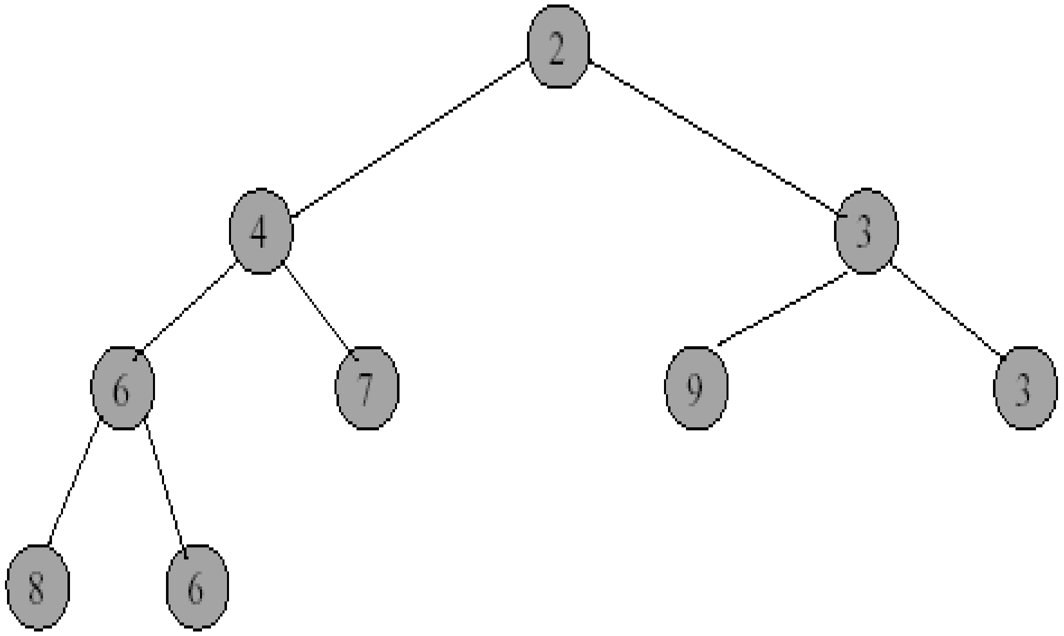
Min Heap with 9 Nodes

A **minimal heap** (descending heap) is an almost complete binary tree in which the value at each parent node is less than or equal to the values in its child nodes.

Obviously, the minimum value is in the root node.

Note, too, that any path from a leaf to the root passes through the data in descending order.

Here is an example of a minimal heap:



Complete binary tree with 9 nodes

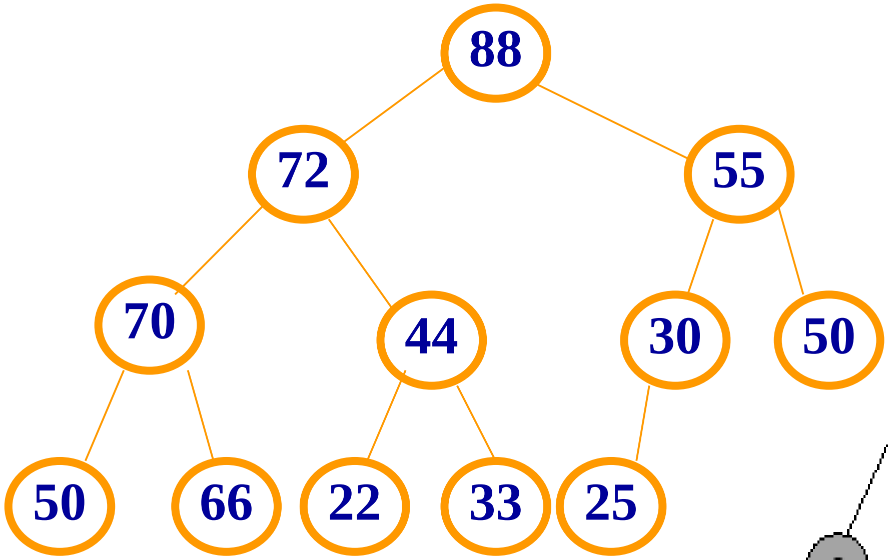
Max Heap

A **Max heap** (Ascending heap) is an almost complete binary tree in which the value at each parent node is greater than or equal to the values in its child nodes.

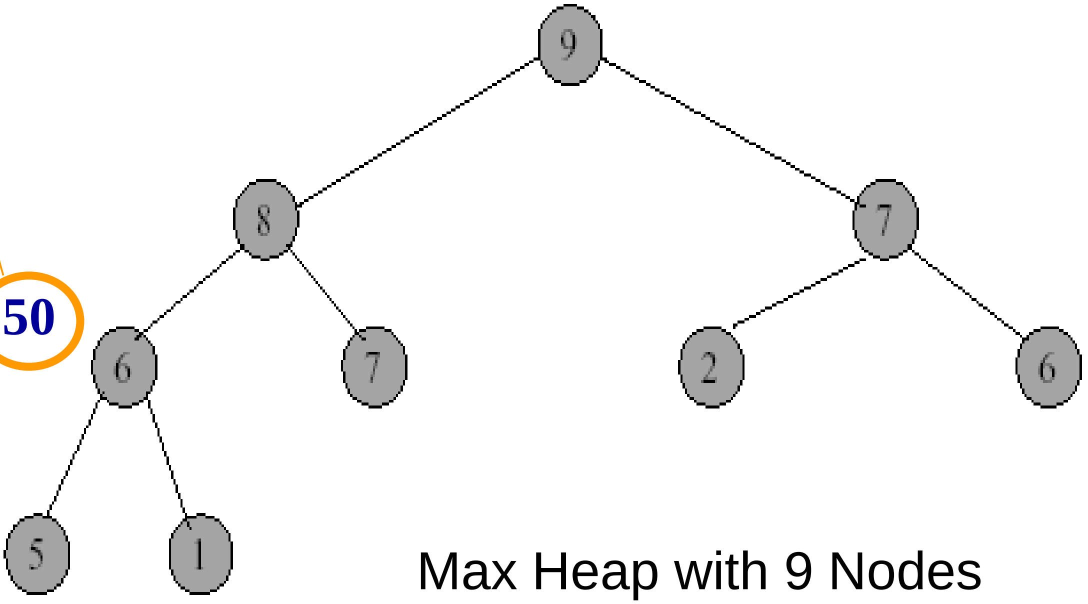
Obviously, the maximum value is in the root node.

Note, too, that any path from a leaf to the root passes through the data in Ascending order.

Here is an example of a max heap:

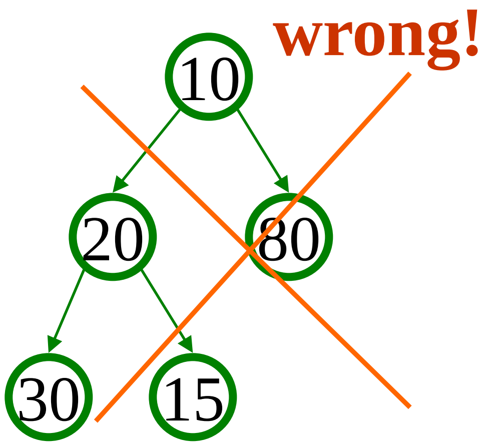


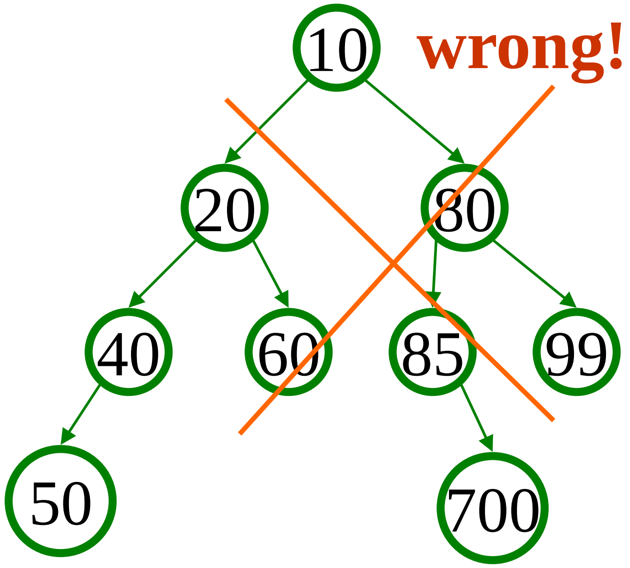
Max Heap with 12 Nodes

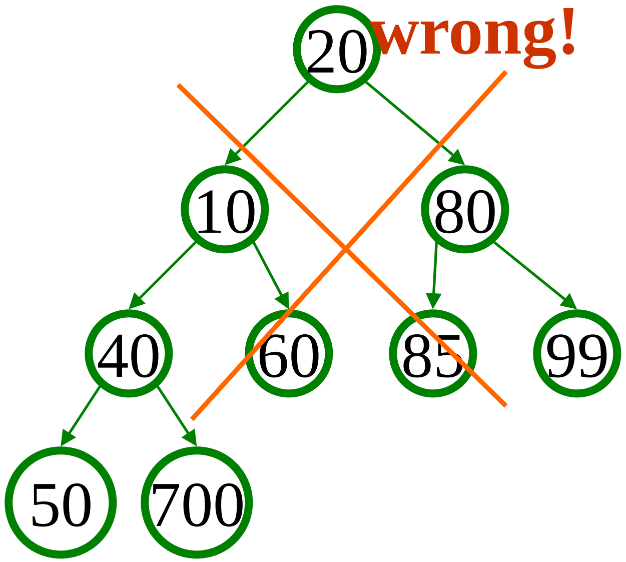


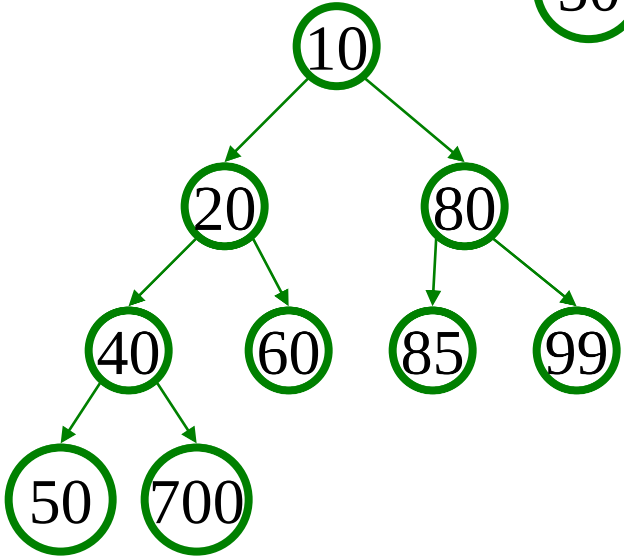
Max Heap with 9 Nodes

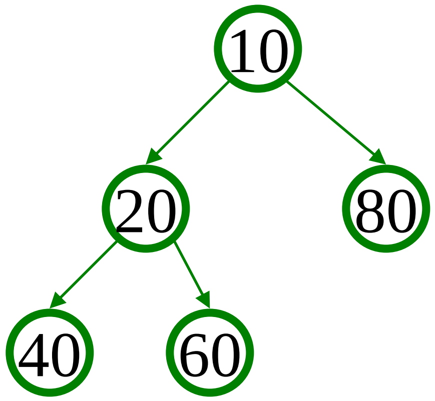
Which are min-heaps?

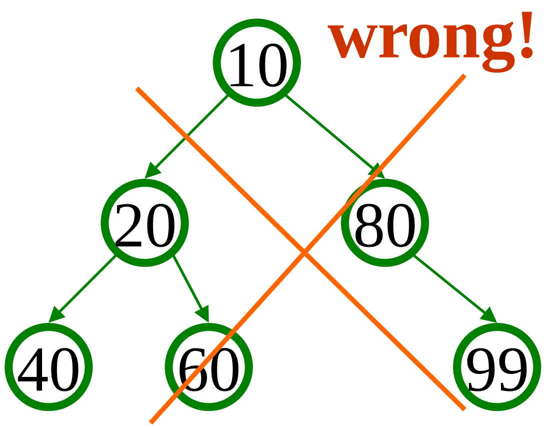




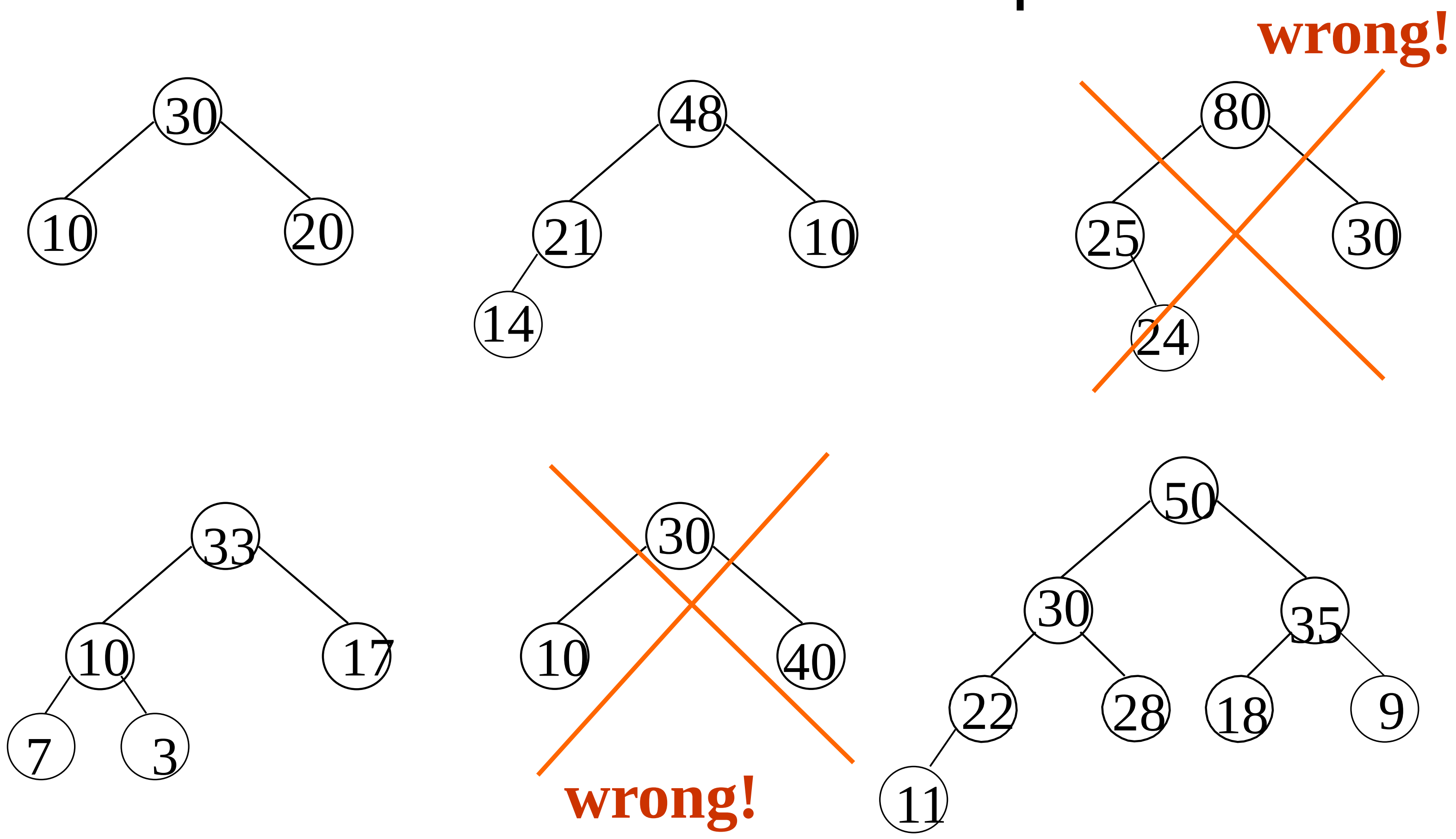




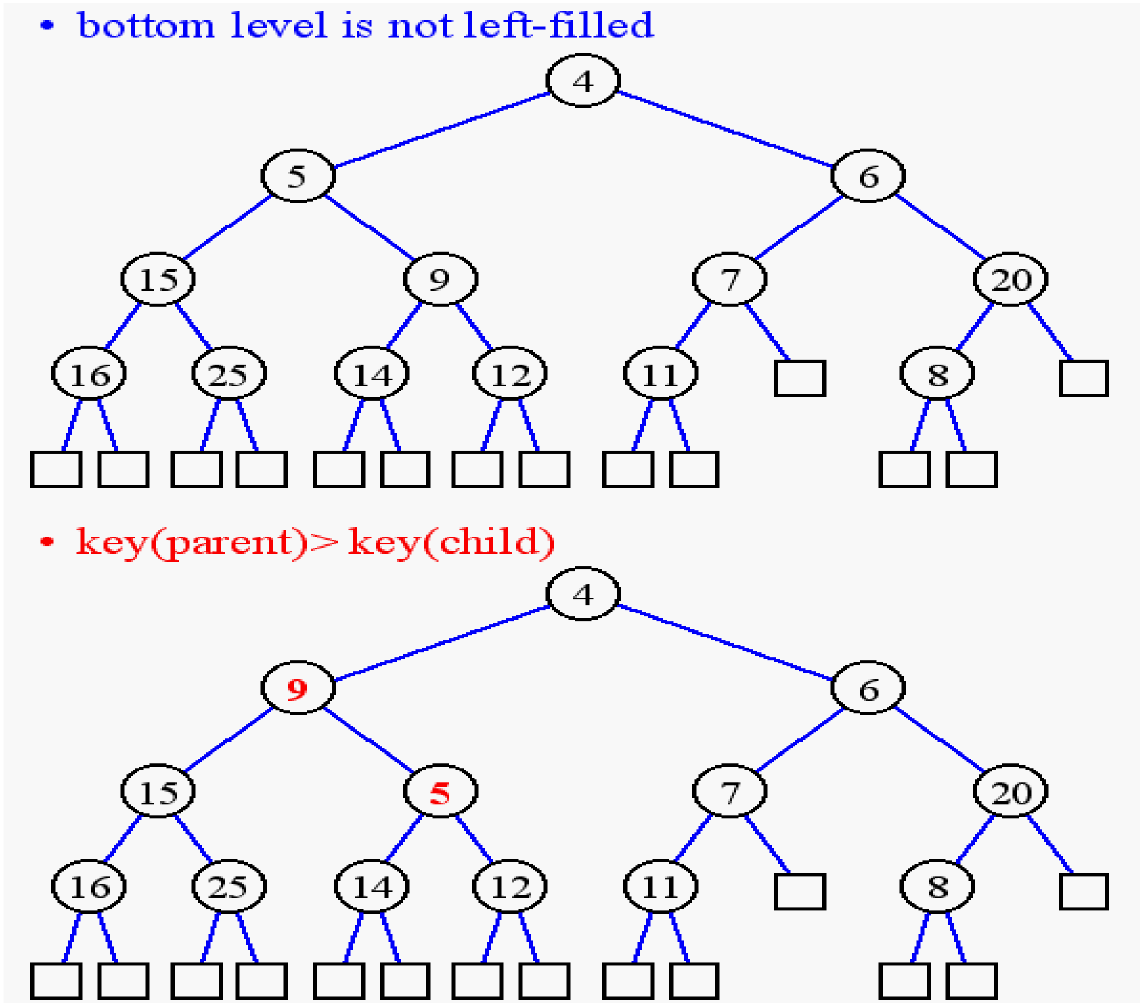




Which are max-heaps?



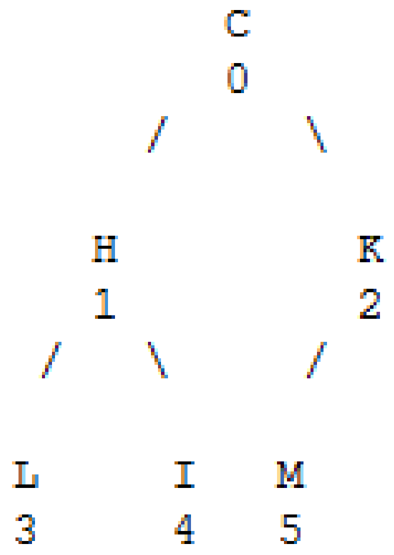
Heap or Not a Heap?



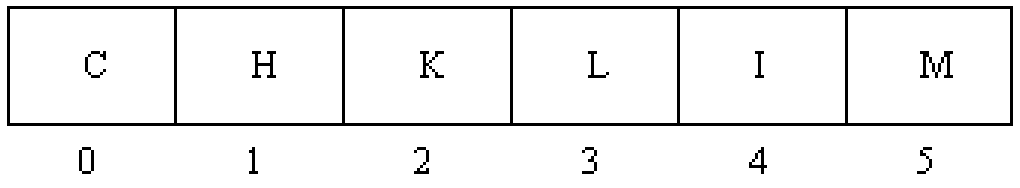
Heap Implementation

- The typical storage method for a heap, or any almost complete binary tree, works as follows.

- Begin by numbering the nodes level by level from the top down, left to right.
- when implementing a complete binary tree, we actually can "cheat" and just use an array
  - index of root = 1 (leave 0 empty for simplicity)
  - for any node *n* at index *i*,
    - index of *n*.left = 2*i*
    - index of *n*.right = 2*i* + 1
- For example, consider the following heap. The numbering has been added below the nodes.



- Then store the data in an array as shown below:



C	H	K	L	I	M
0	1	2	3	4	5

Heap Implementation (Cont..)

- Using arrays
- Parent = k ;
- Children = 2k , 2k+1;

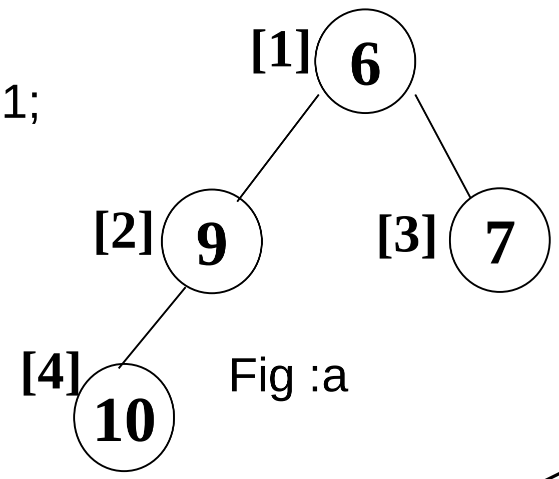


Fig :a

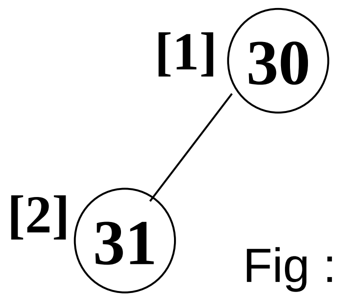


Fig :b

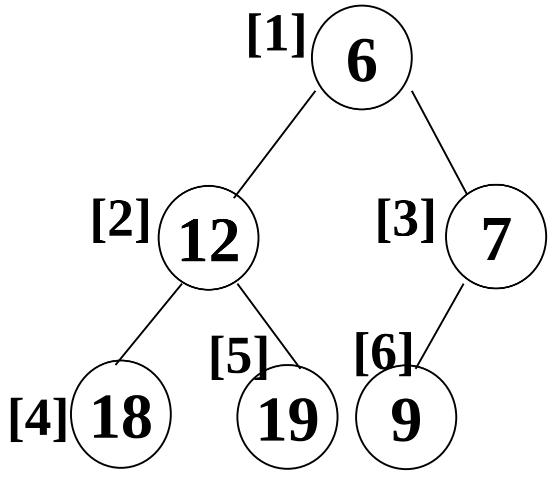


Fig :c

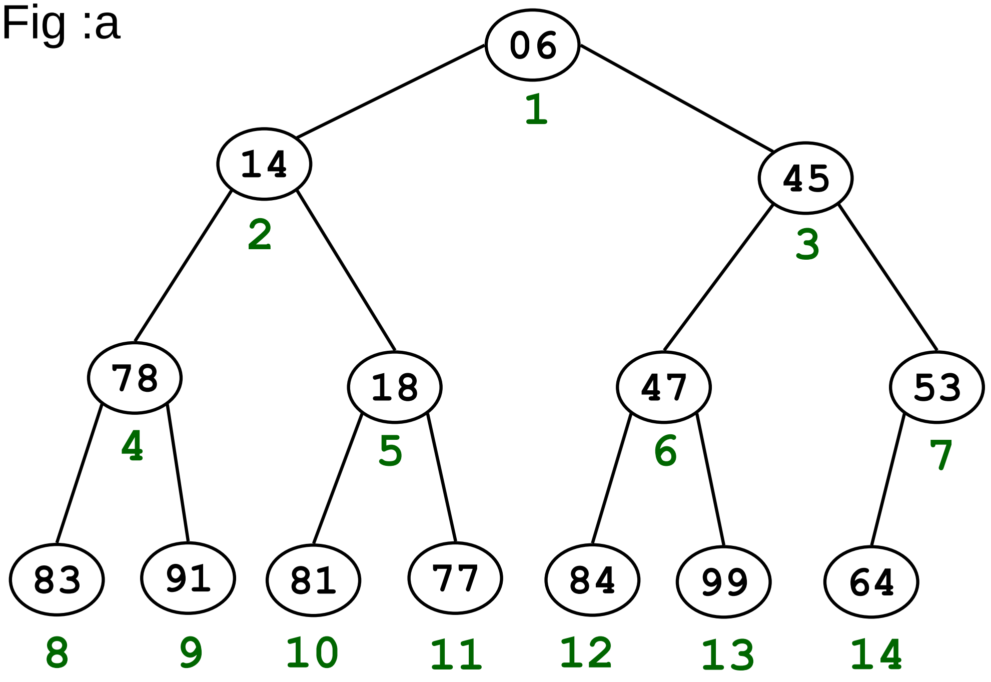
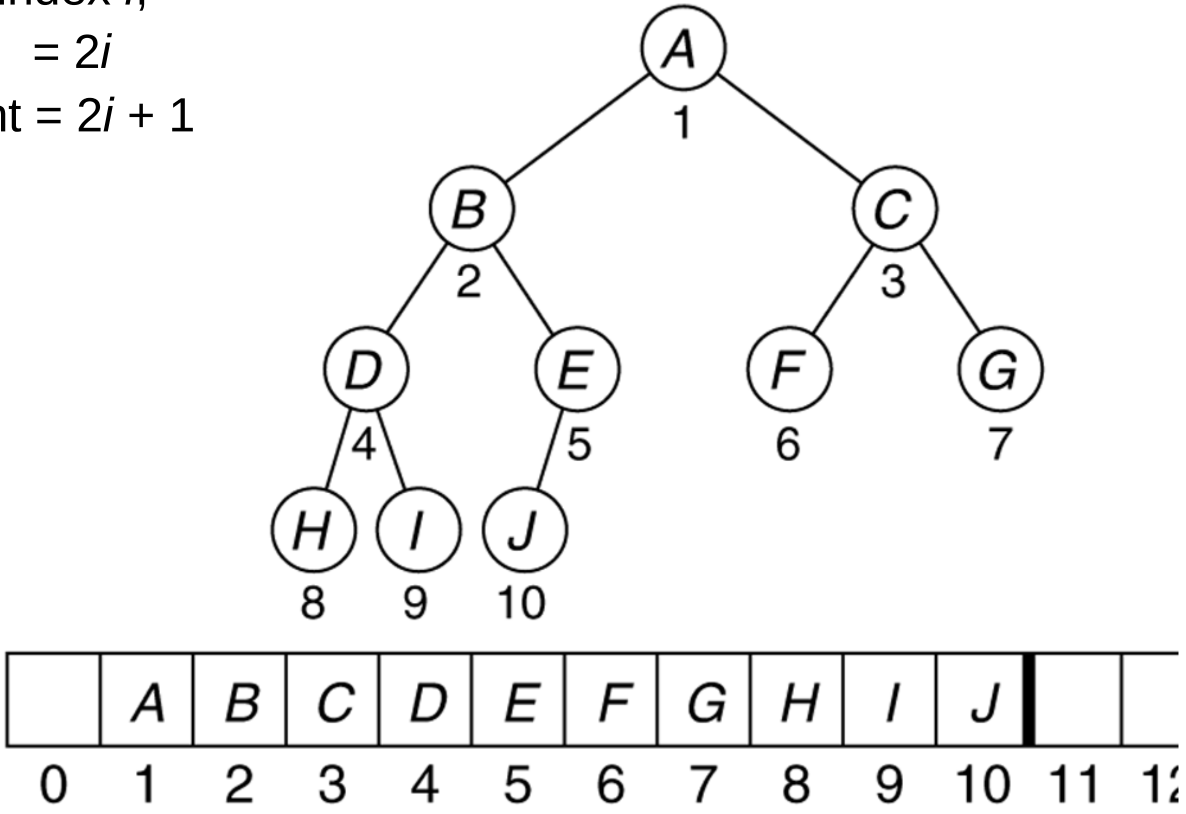


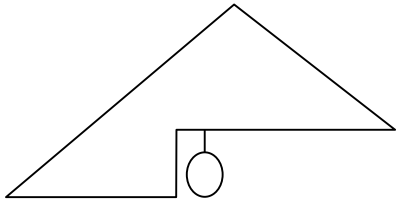
Fig :d

Heap Implementation (Cont..)

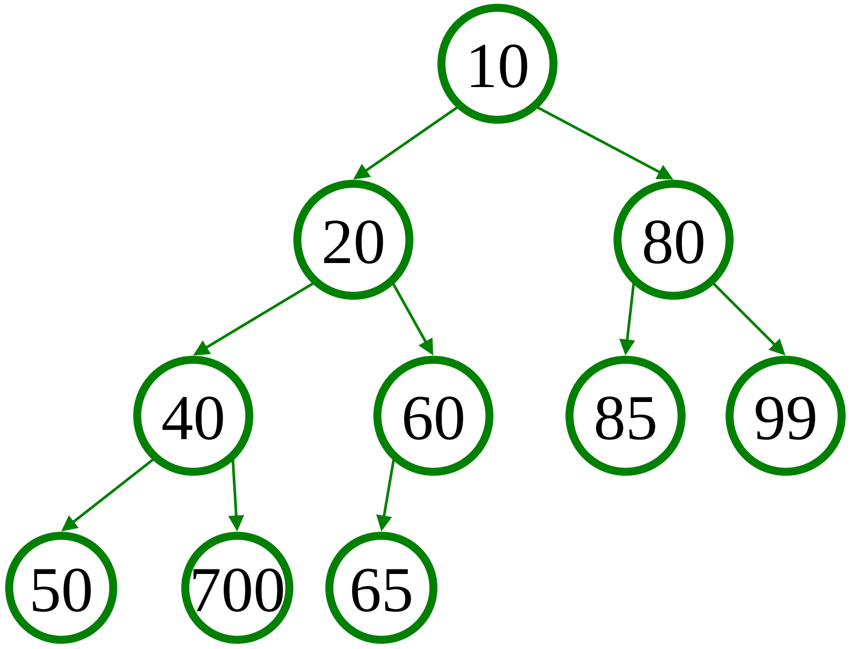
- when implementing a complete binary tree, we actually can "cheat" and just use an array
  - index of root = 1 (leave 0 empty for simplicity)
  - for any node *n* at index *i*,
    - index of *n*.left = 2*i*
    - index of *n*.right = 2*i* + 1

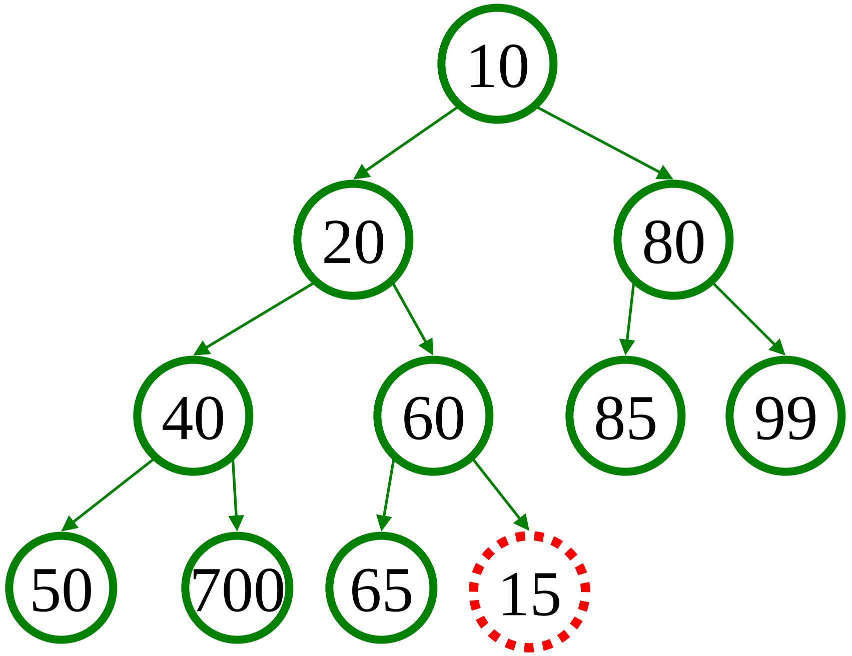


Adding to a heap



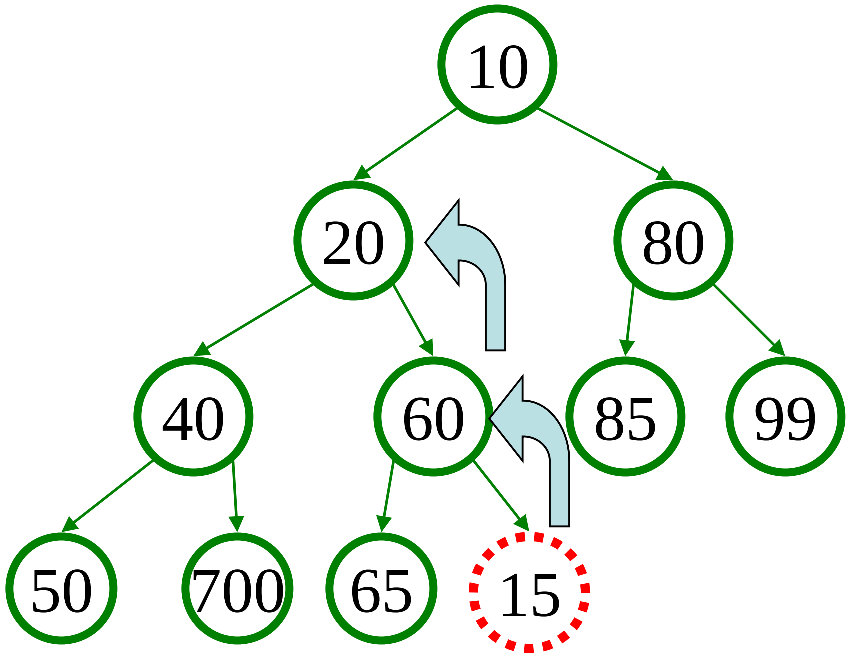
- when an element is added to a heap, it should be initially placed as the rightmost leaf (to maintain the completeness property)
  - heap ordering property becomes broken!

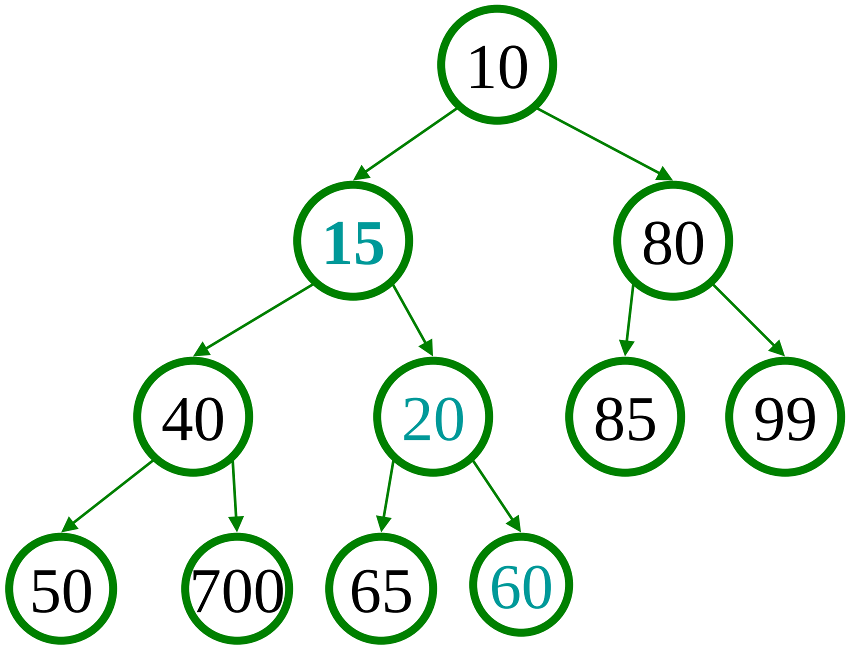




Adding to a min heap, cont..

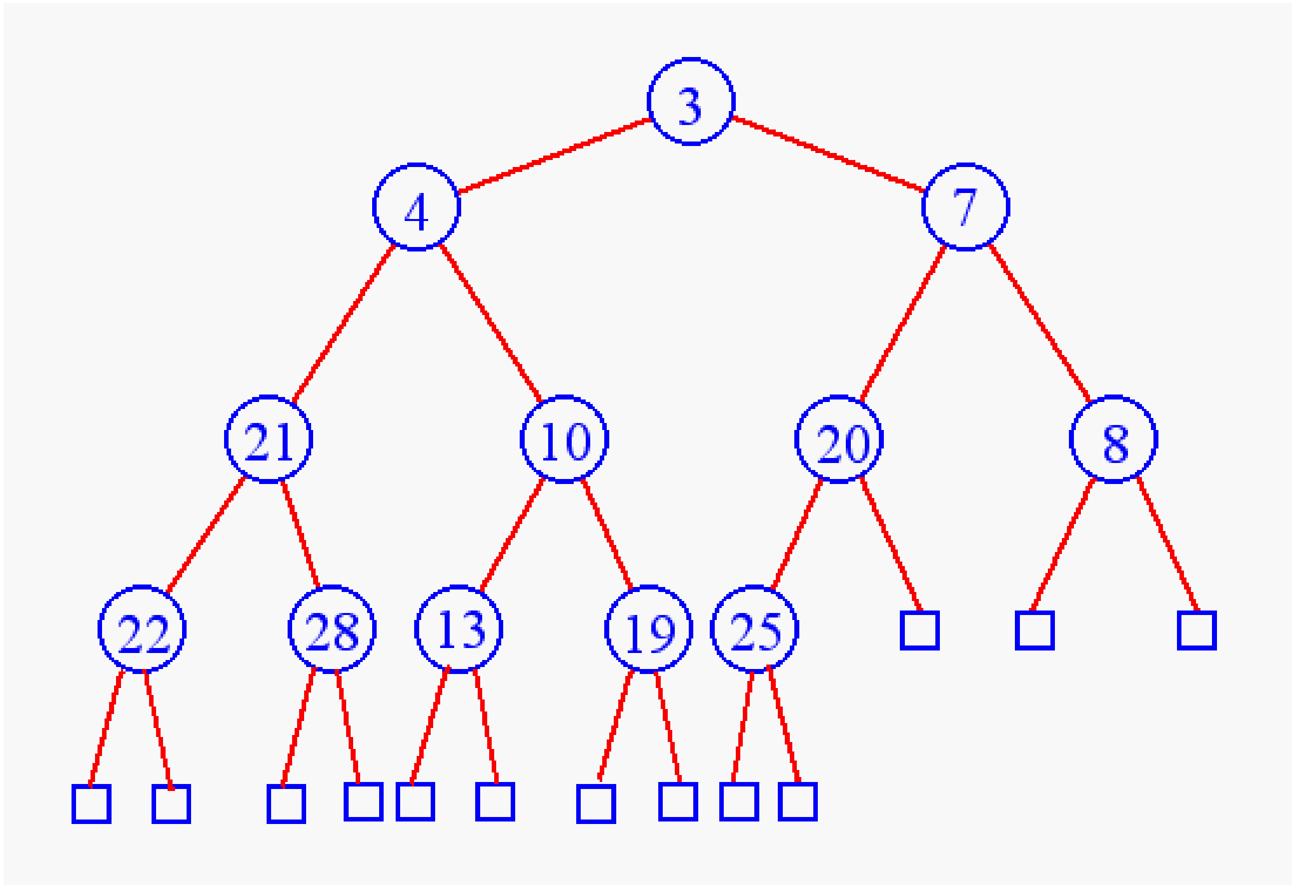
- To restore heap ordering property, the newly added element must be shifted upward ("bubbled up") until it reaches its proper place
  - bubble up by swapping with parent





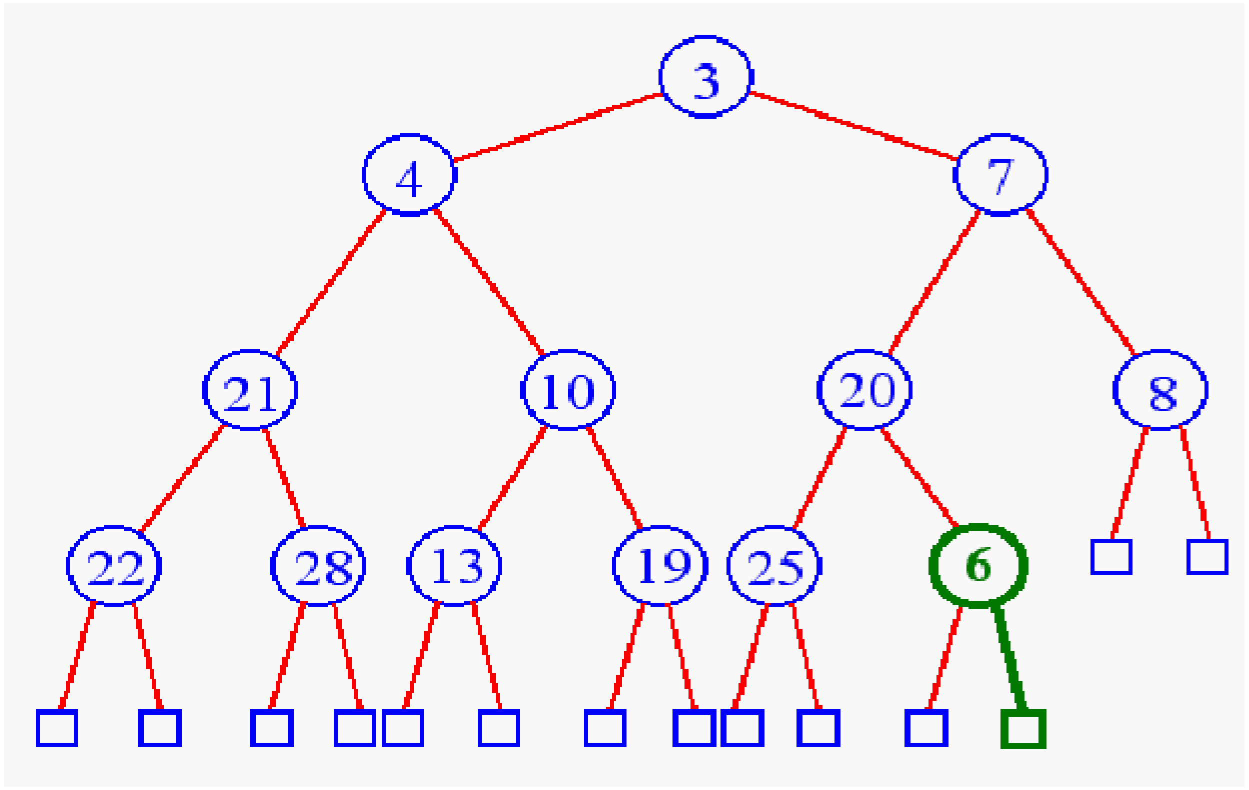
Adding to a min heap, cont..

- Insert 6



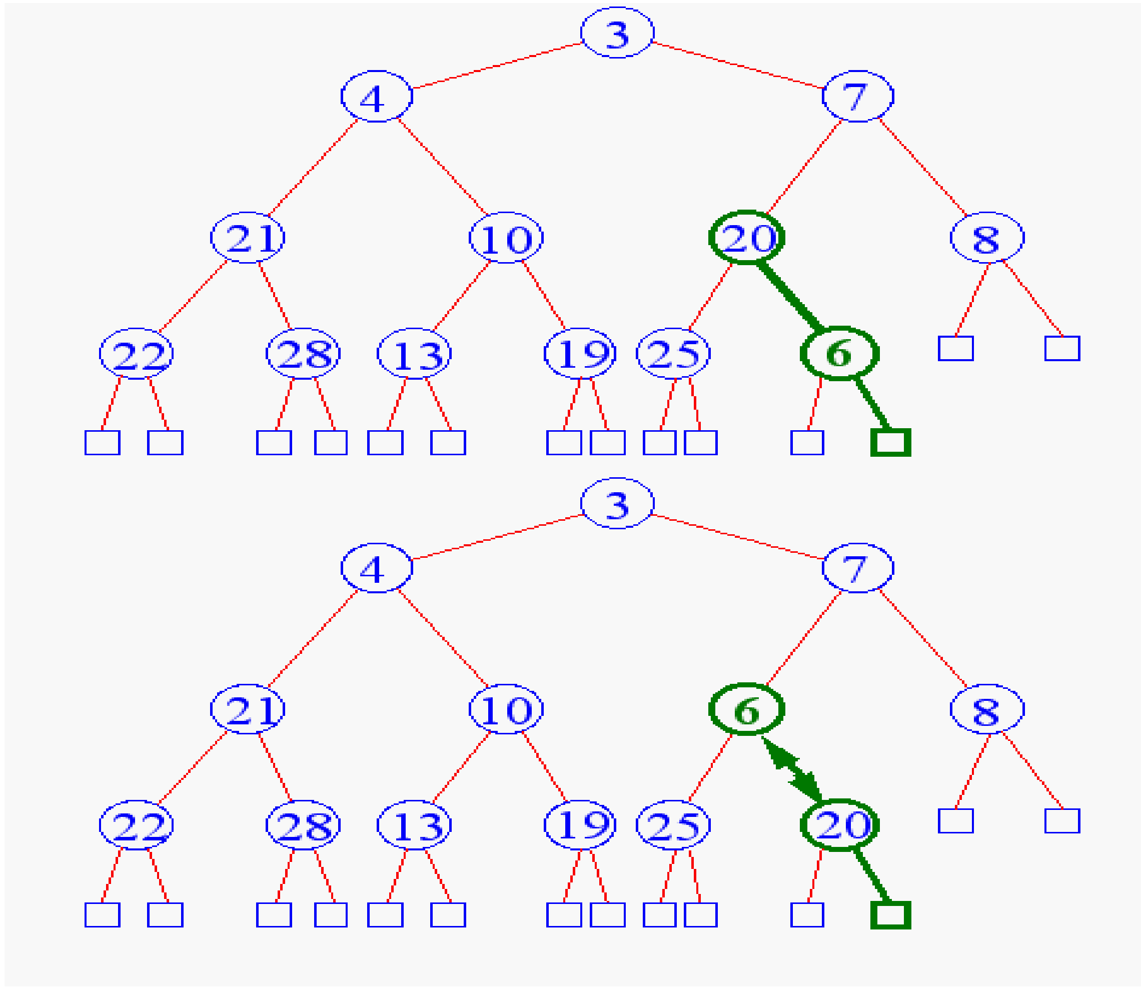
Adding to a min heap, cont..

- Add key in next available position

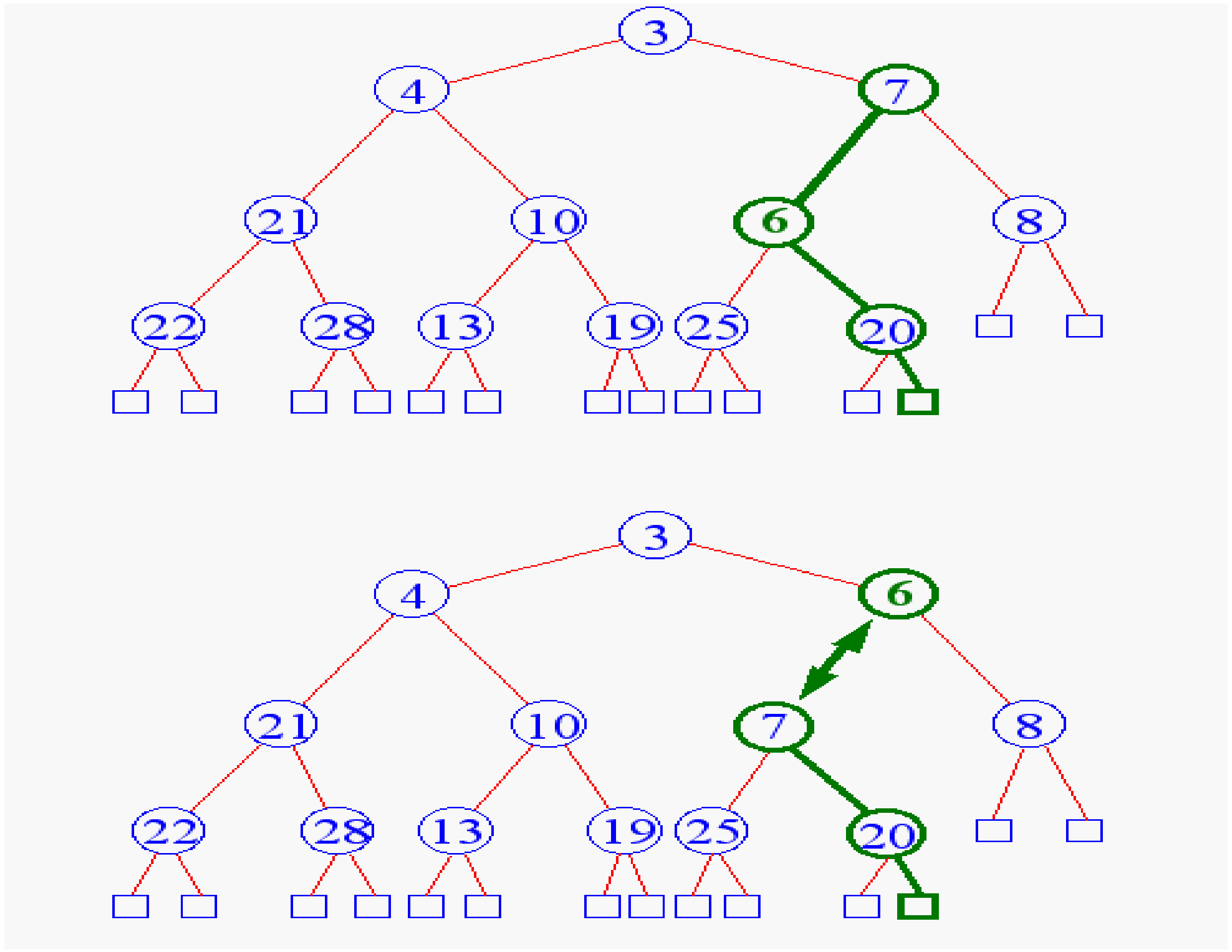


Adding to a min heap, cont..

- Begin Unheap

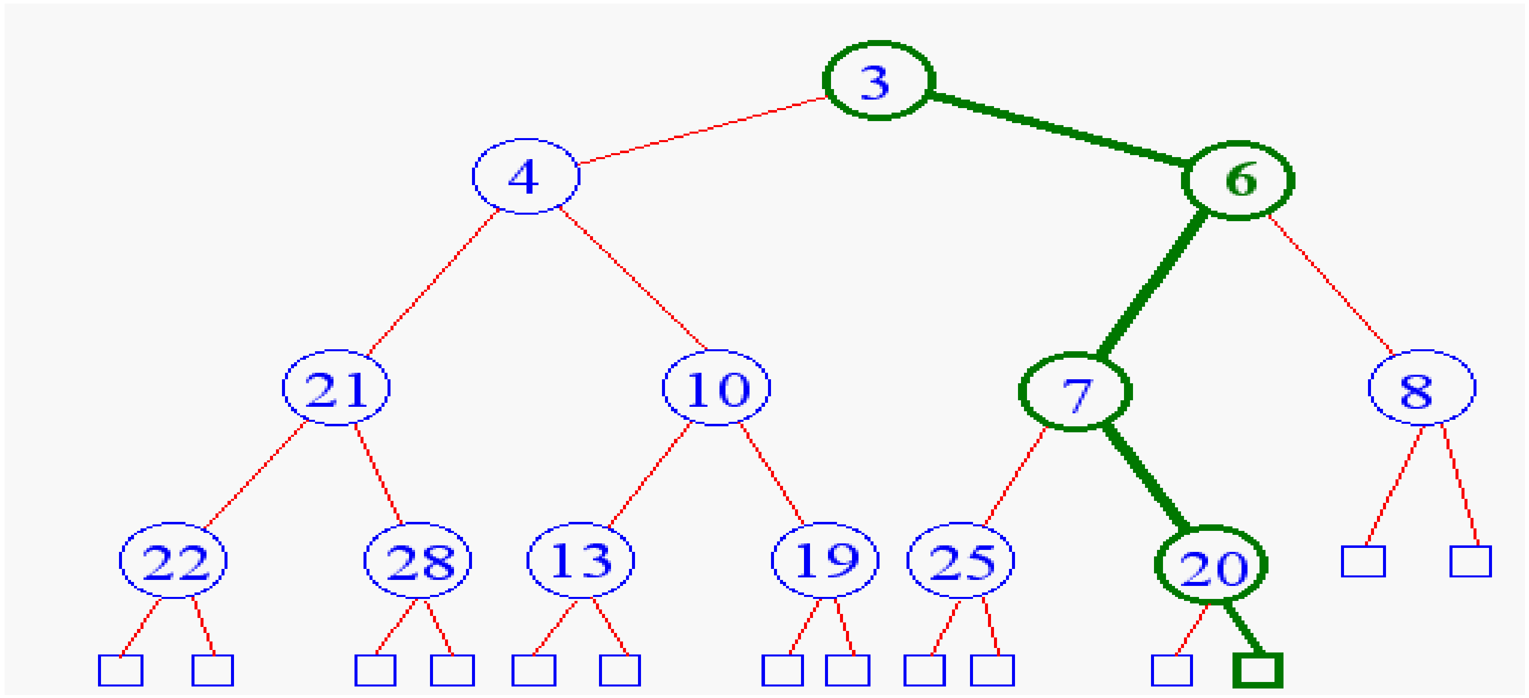


Adding to a min heap, cont..



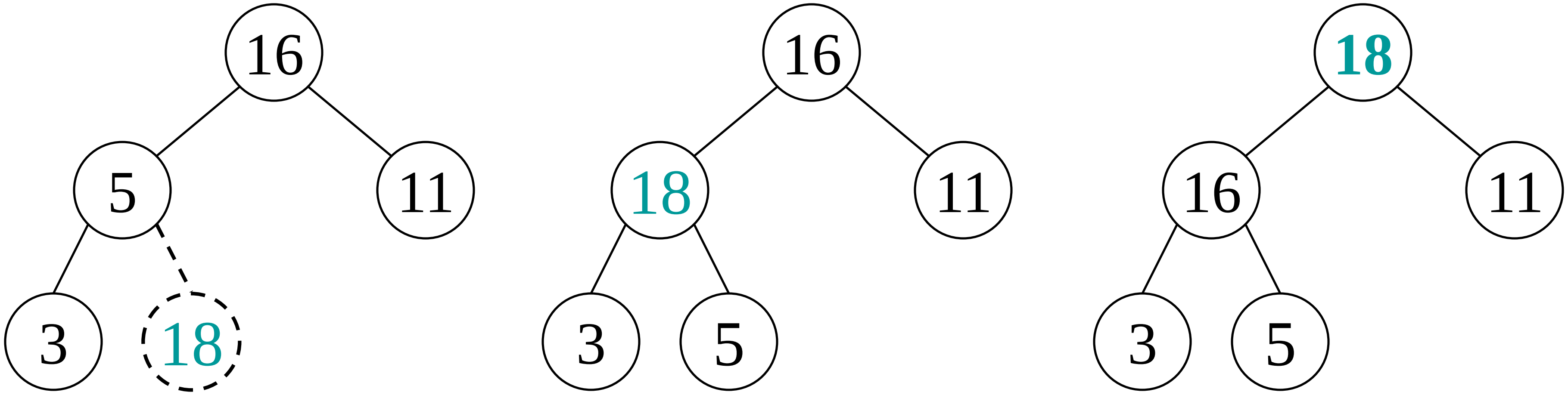
Adding to a min heap, cont..

- Terminate unheap when
  - reach root
  - key child is greater than key parent



Adding to a max-heap

- same operations, but must bubble up *larger* values to top

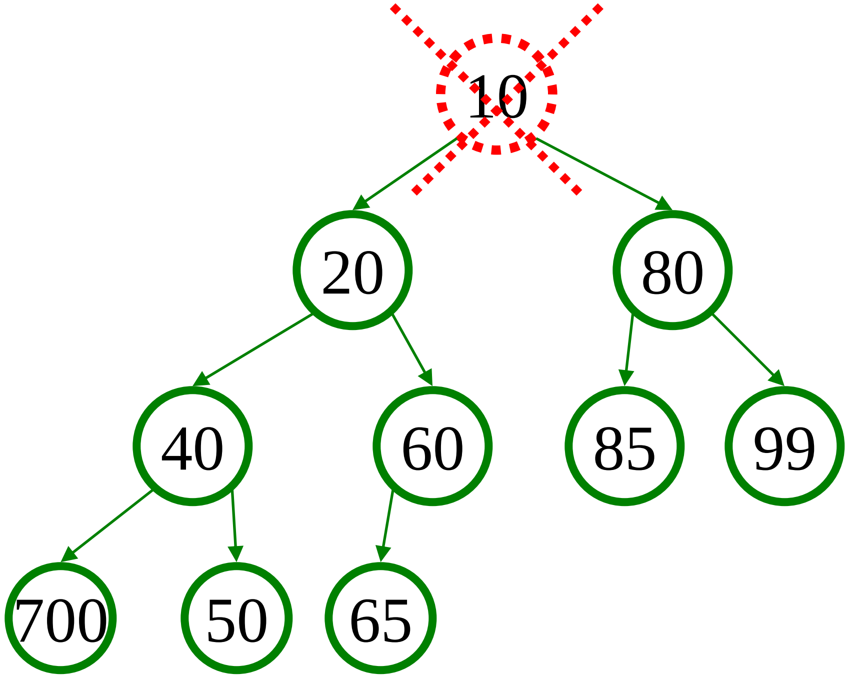


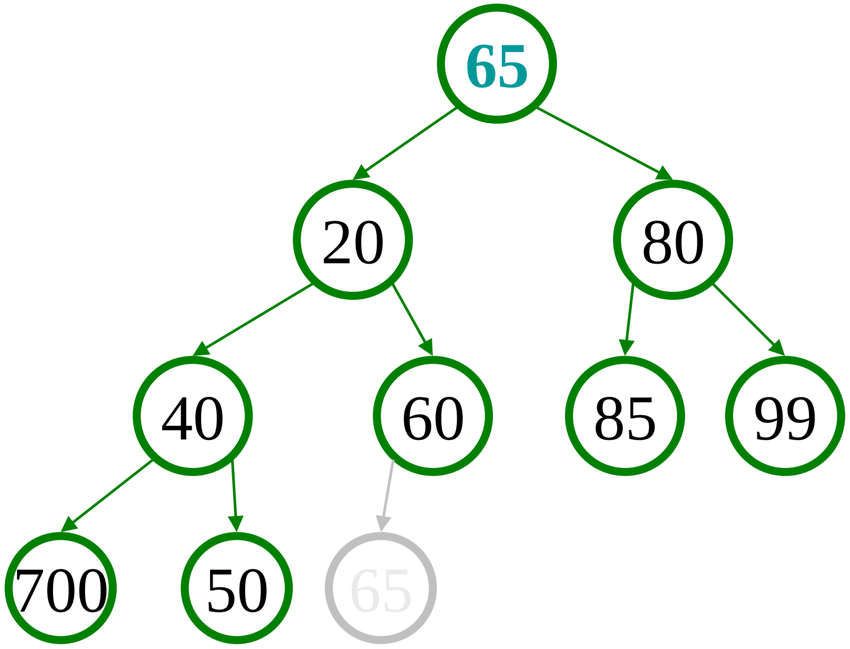
Removing from a Heap

- We always remove the item from the root. That way we always get the smallest item. The problem is then how to adjust the binary tree so that we again have a heap (with one less item).
- The algorithm works like this:
- First, remove the root item and replace it temporarily with the item in the last position.
- Call this replacement the target.
- A FilterDown routine is then used to check the path from the root to a leaf for the correct position for this target.
- The path is chosen by always selecting the smaller child at each node. For example, let's remove the C from this heap:

Removing from a min-heap

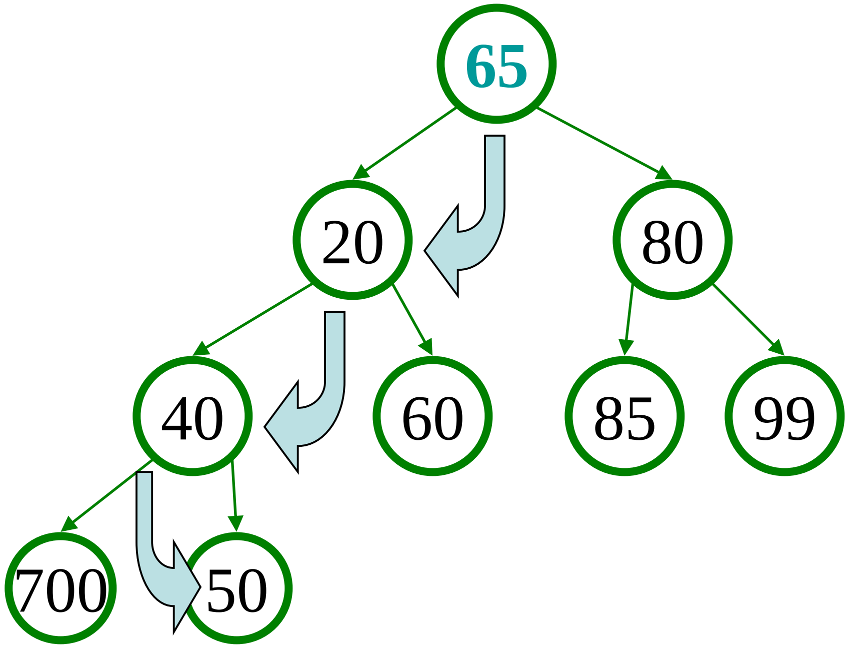
- min-heaps only support `remove` of the min element (the root)
  - must remove the root while maintaining heap completeness and ordering properties
  - intuitively, the last leaf must disappear to keep it a heap
  - initially, just swap root with last leaf (we'll fix it)

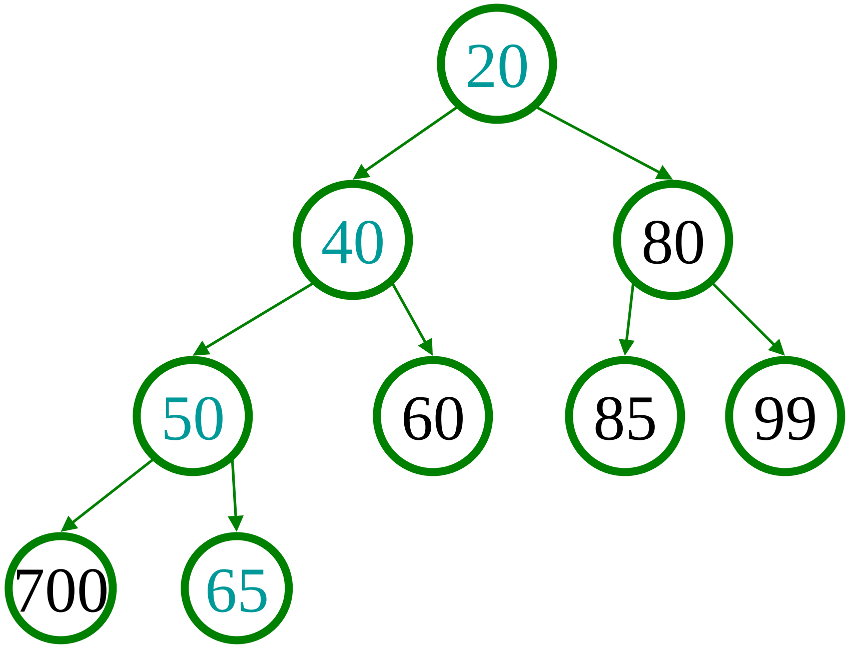




Removing from heap, cont..

- must fix heap-ordering property; root is out of order
  - shift the root downward ("bubble down") until it's in place
  - swap it with its smaller child each time
    - What happens if we don't always swap with the smaller child?



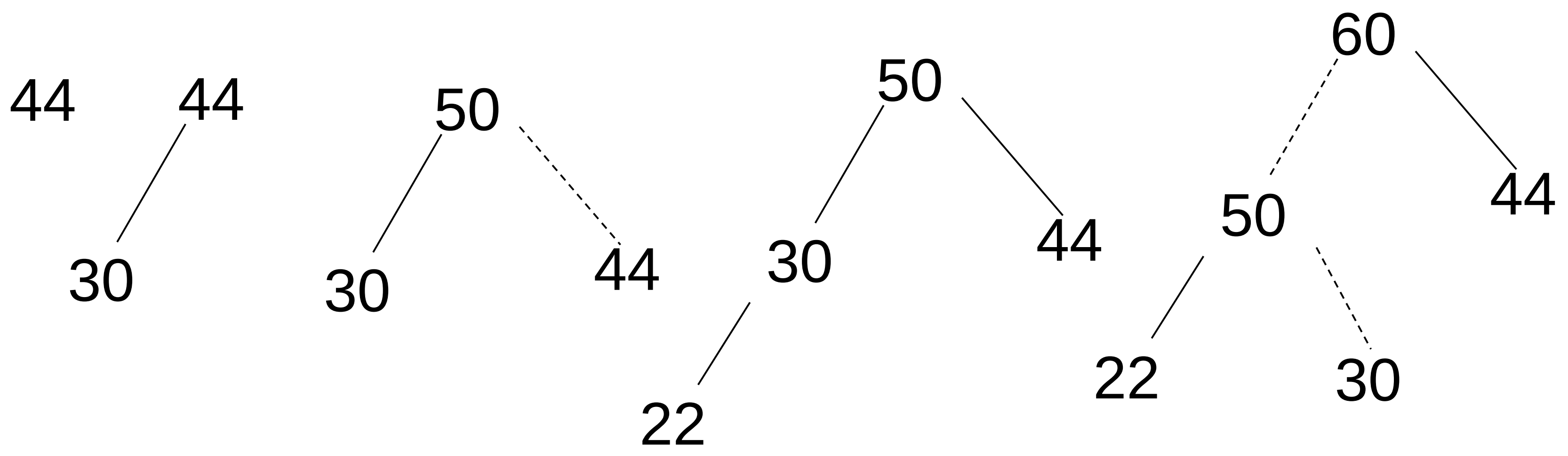


Building a Heap

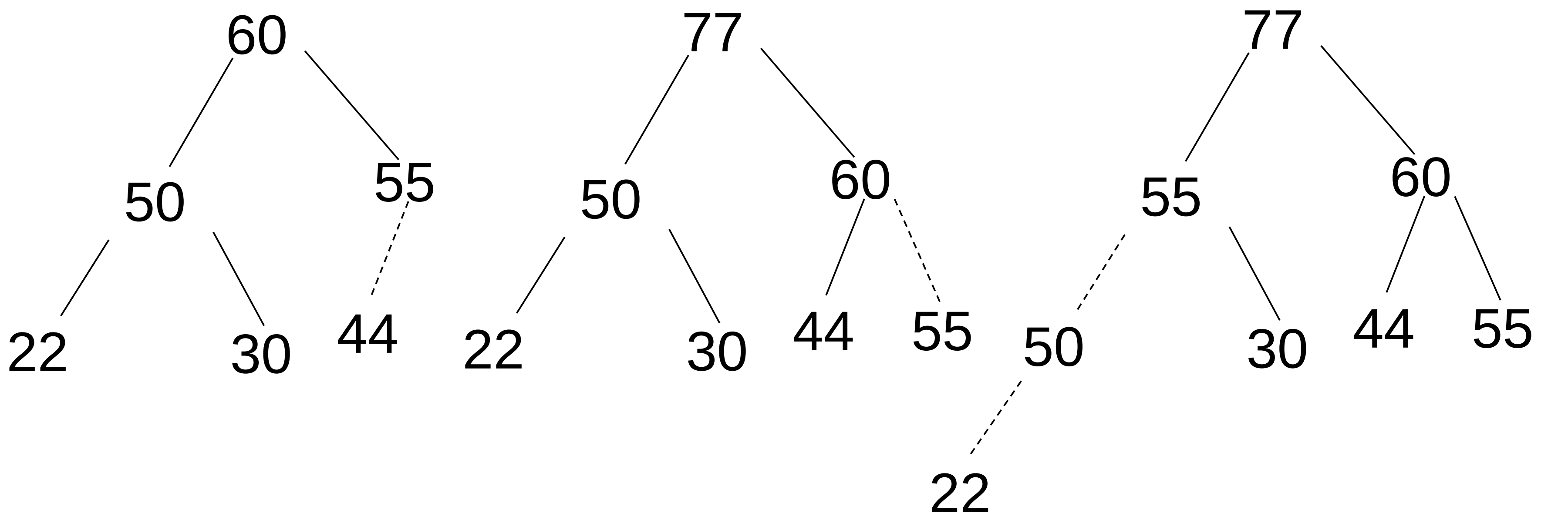
EXAMPLE 7.21

- Build a heap H from the following list of numbers:

  44, 30, 50, 22, 60, 55,77, 55



Building a Heap



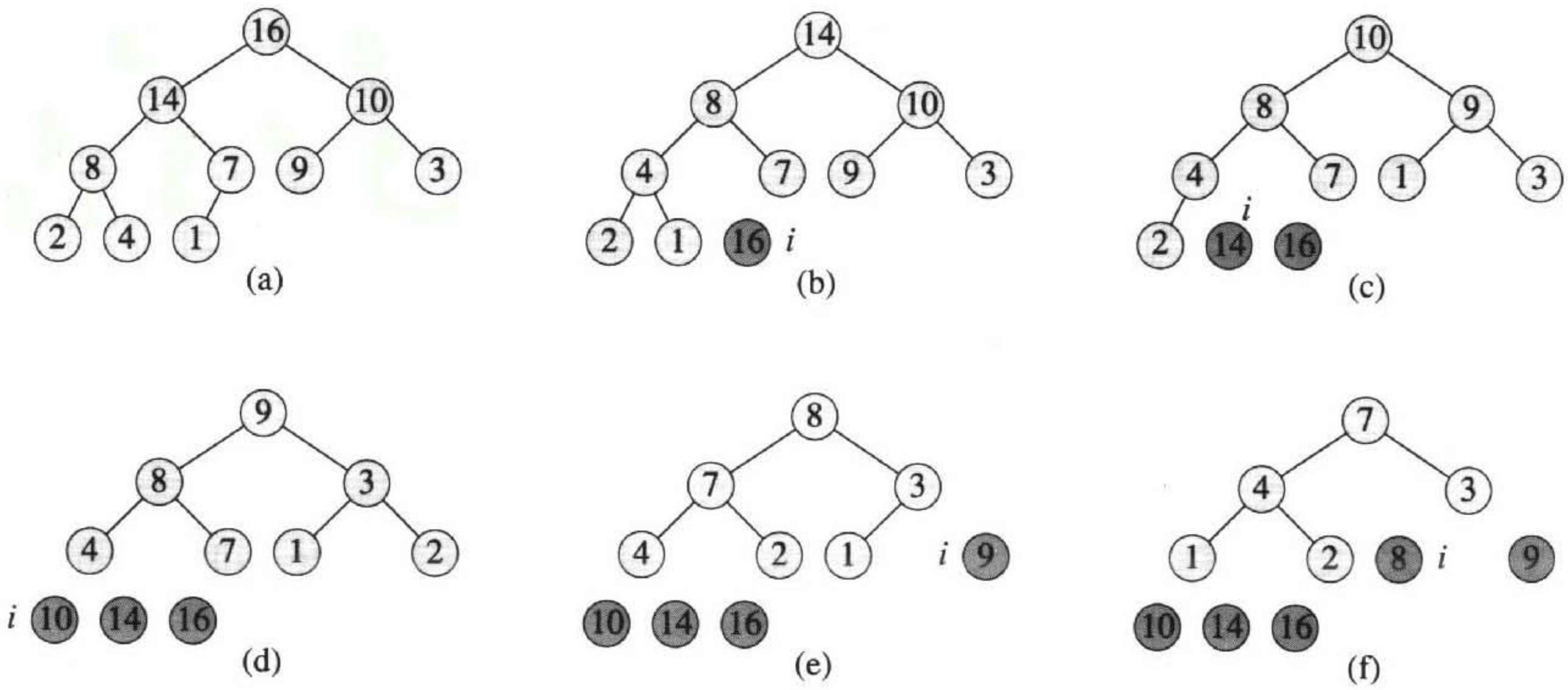
Applications of Heaps

- Sort (heap sort)
- Machine scheduling
- Huffman codes

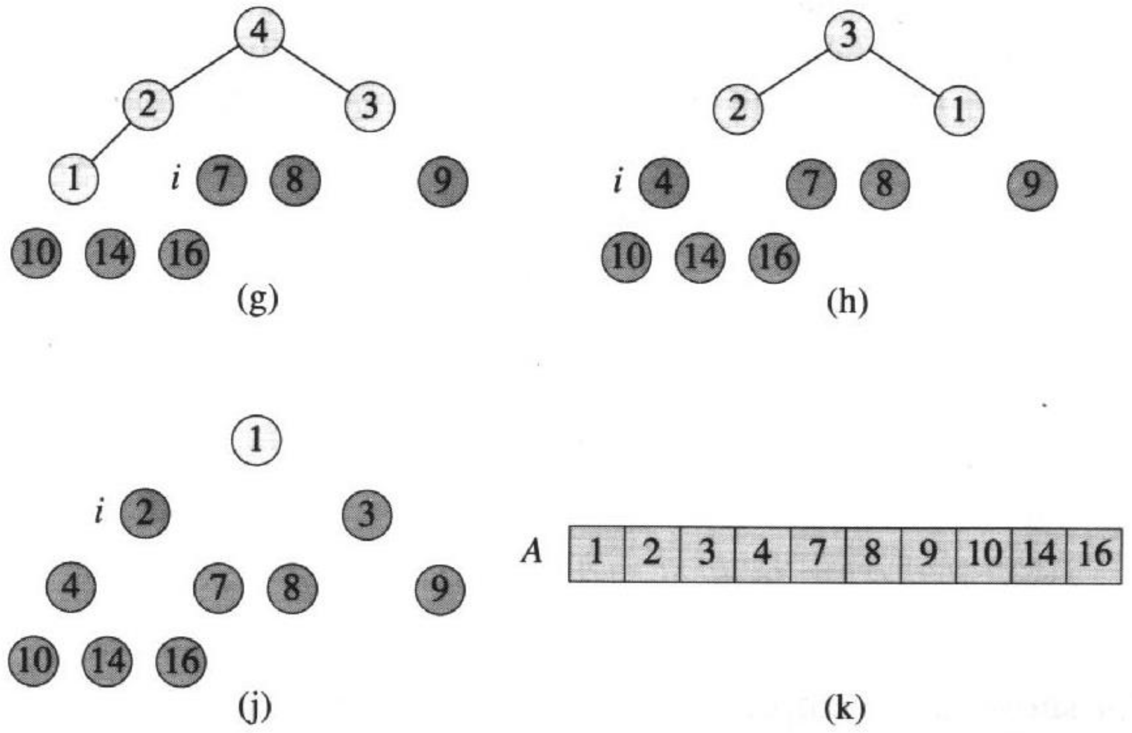
Heapsort

- Heapsort is performed by somehow creating a heap and then removing the data items one at a time.
- The heap could start as an empty heap, with items inserted one by one.
- However, there is a relatively easy routine to convert an array of items into a heap, so that method is often used. This routine is described below.
- Once the array is converted into a heap, we remove the root item (the smallest), readjust the remaining items into a heap,
- and place the removed item at the end of the heap (array).
- Then we remove the new item in the root (the second smallest), readjust the heap, and place the removed item in the next to the last position, etc.

Heap Sort



Heap Sort



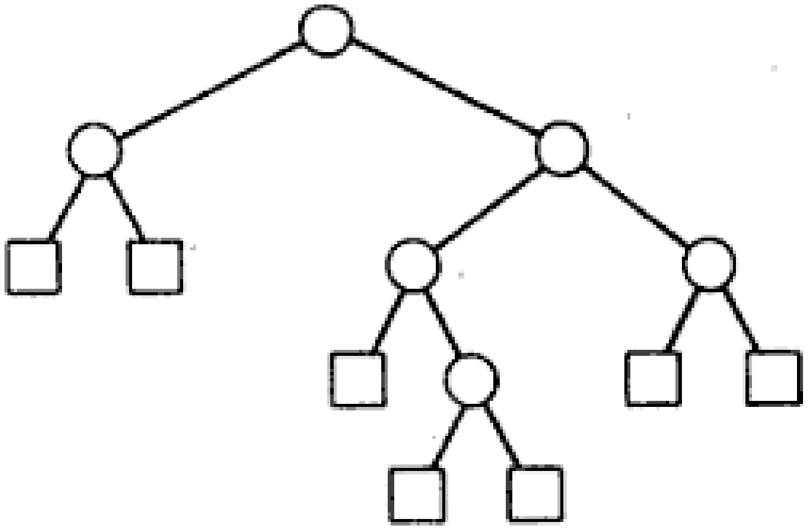
Huffman Tree Properties

- Huffman tree is a binary tree with minimum weighted external path length for a given set of frequencies (weights)

- A Huffman tree is a binary tree of integers with two properties:
  1. Each internal node is the sum of its children.
  2. Its weighted external path length is minimal.

10.8 PATH LENGTHS, HUFFMAN'S ALGORITHM

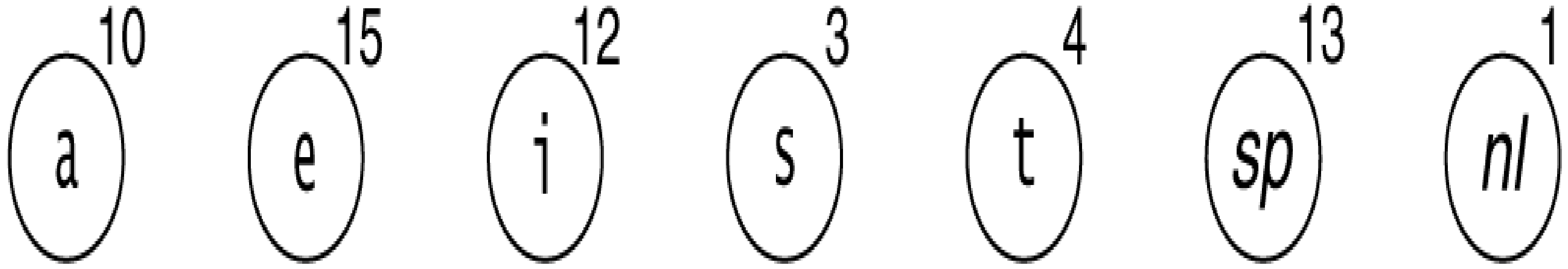
Let T be an extended binary tree or 2-tree (Section 10.3). That is, T is a binary tree where each node N has either 0 or 2 children. The nodes with no children are called *external nodes*, and the nodes with two children are called *internal nodes*. Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes. Moreover, if T has n external nodes, then T has $n - 1$ internal nodes. Figure 10-15 shows a 2-tree with seven external nodes and hence $7 - 1 = 6$ internal nodes.



Huffman Algorithm

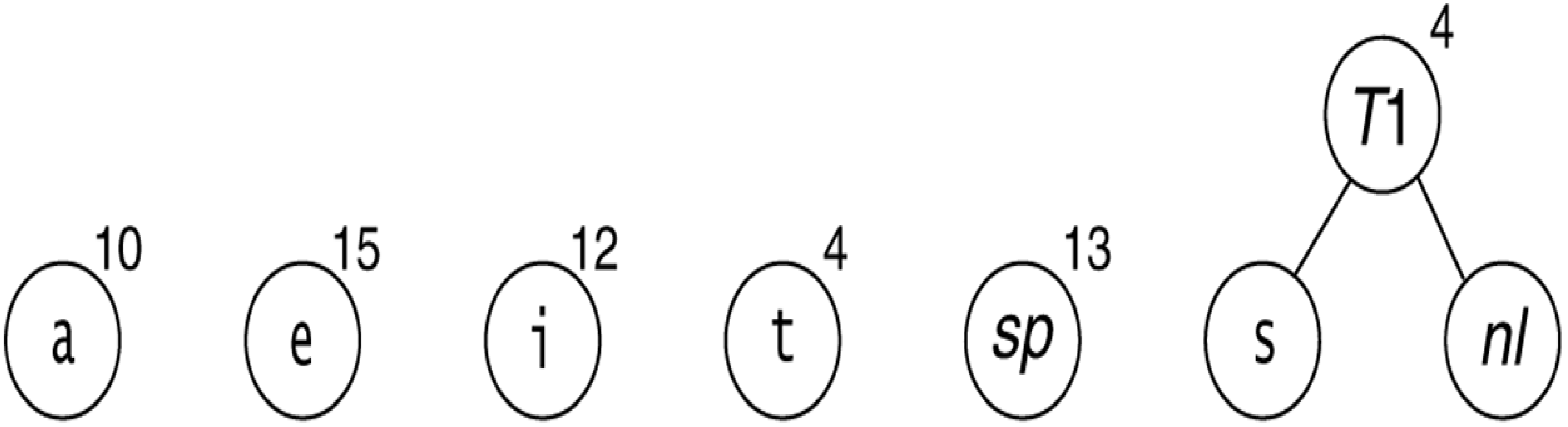
- Initialize a forest, F, to have K single node trees in it, one tree per character, also storing the character's weight
- while (|F| > 1)
  - Find the two trees, T1 and T2, with the smallest weights
  - Create a new tree, T, whose weight is the sum of T1 and T2
  - Remove T1 and T2 from the F, and add them as left and right children of T
  - Add T to F

Huffman's Algorithm Example - I



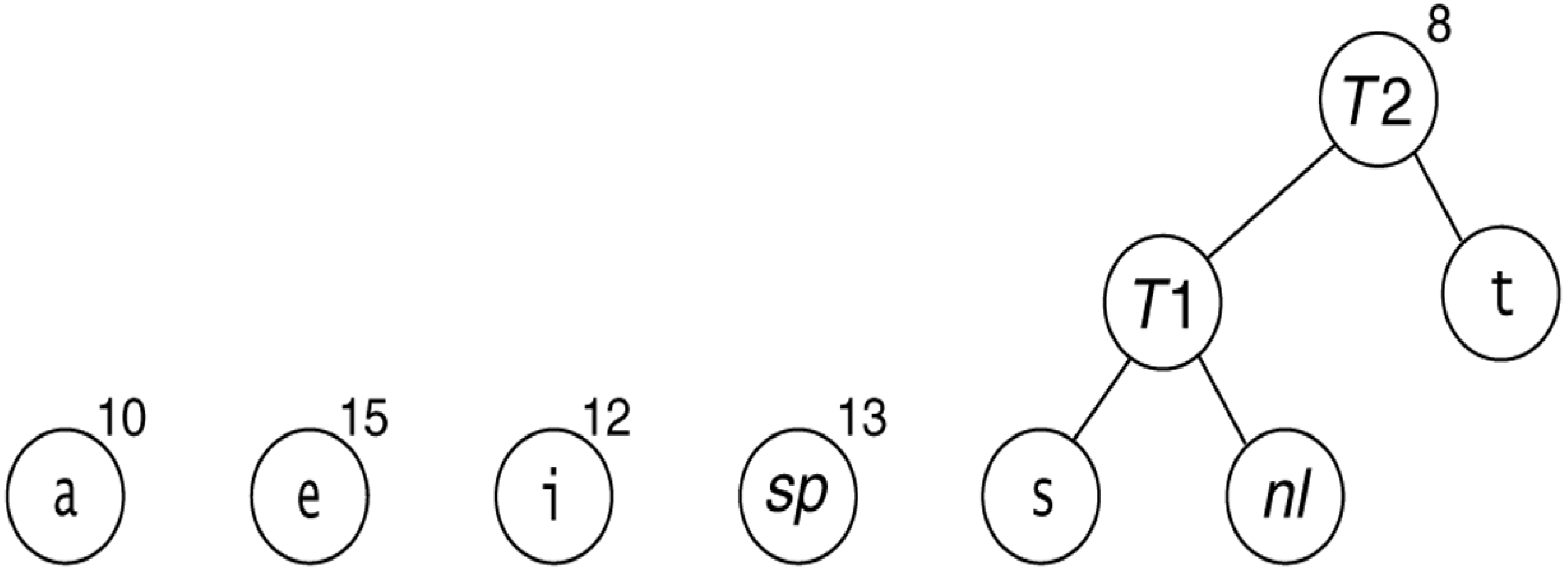
Building a Huffman's Tree

Huffman’s Algorithm Example - II



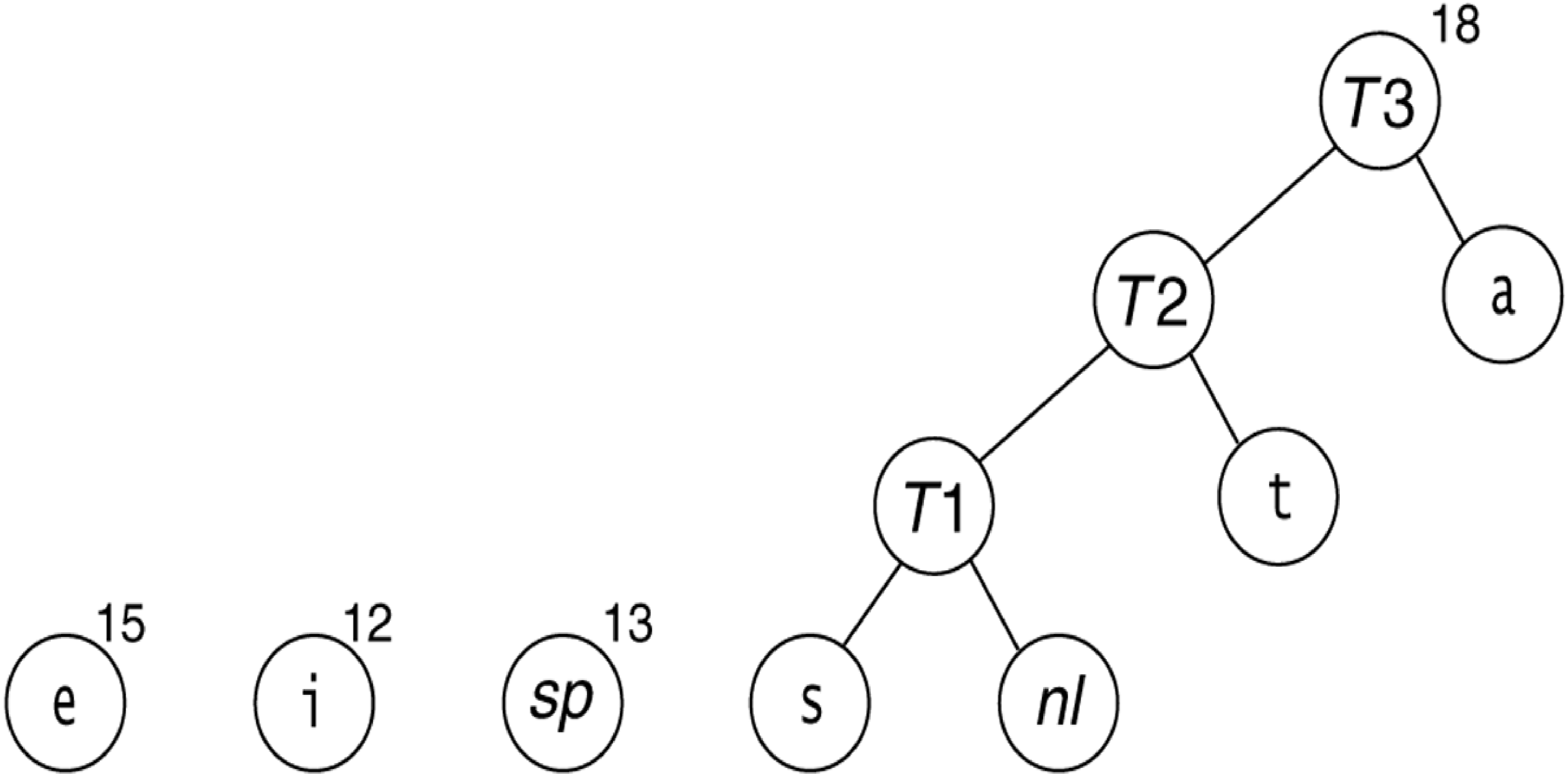
Building a Huffman’s Tree

Huffman's Algorithm Example - III



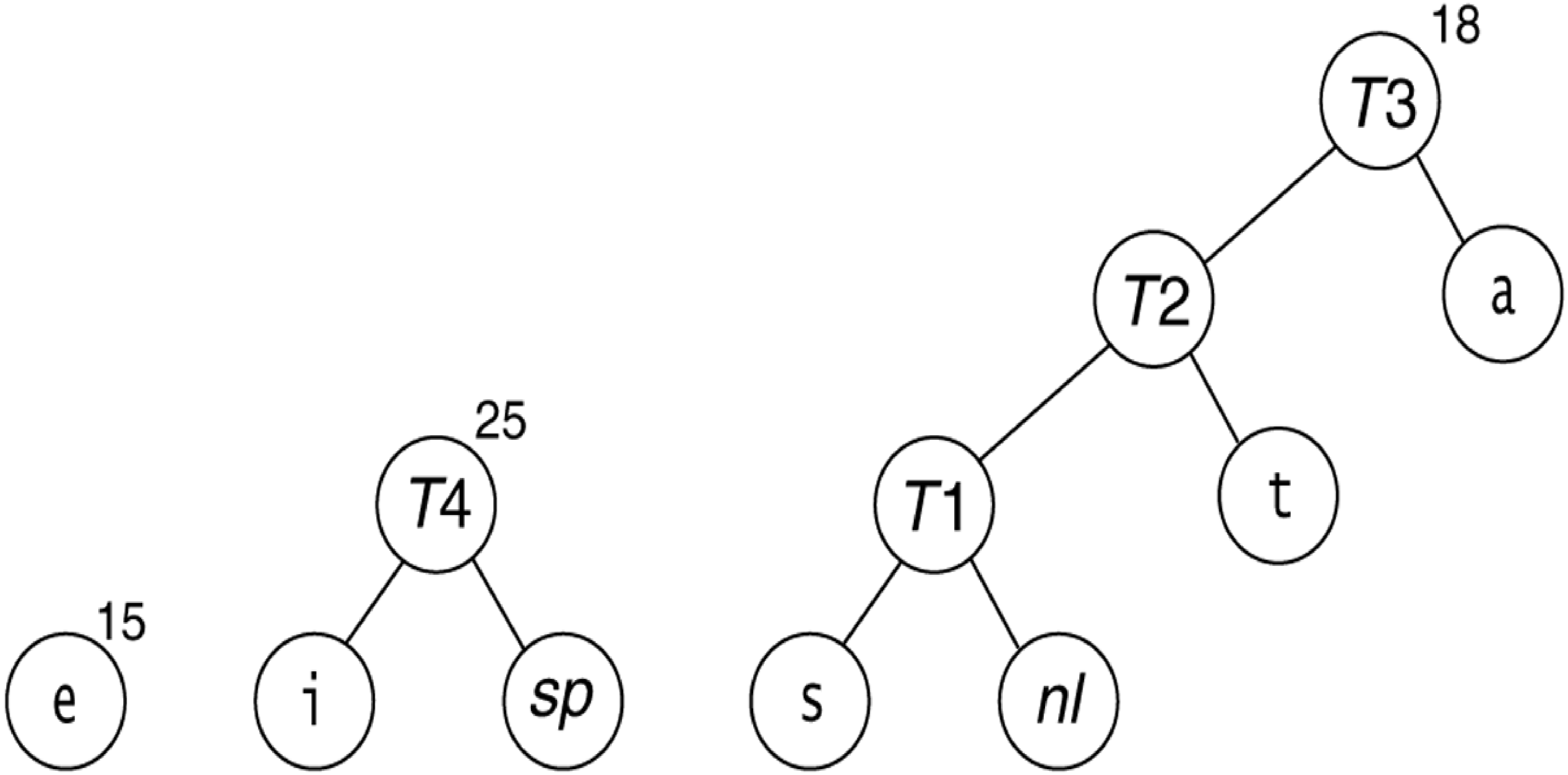
Building a Huffman's Tree

Huffman’s Algorithm Example - IV



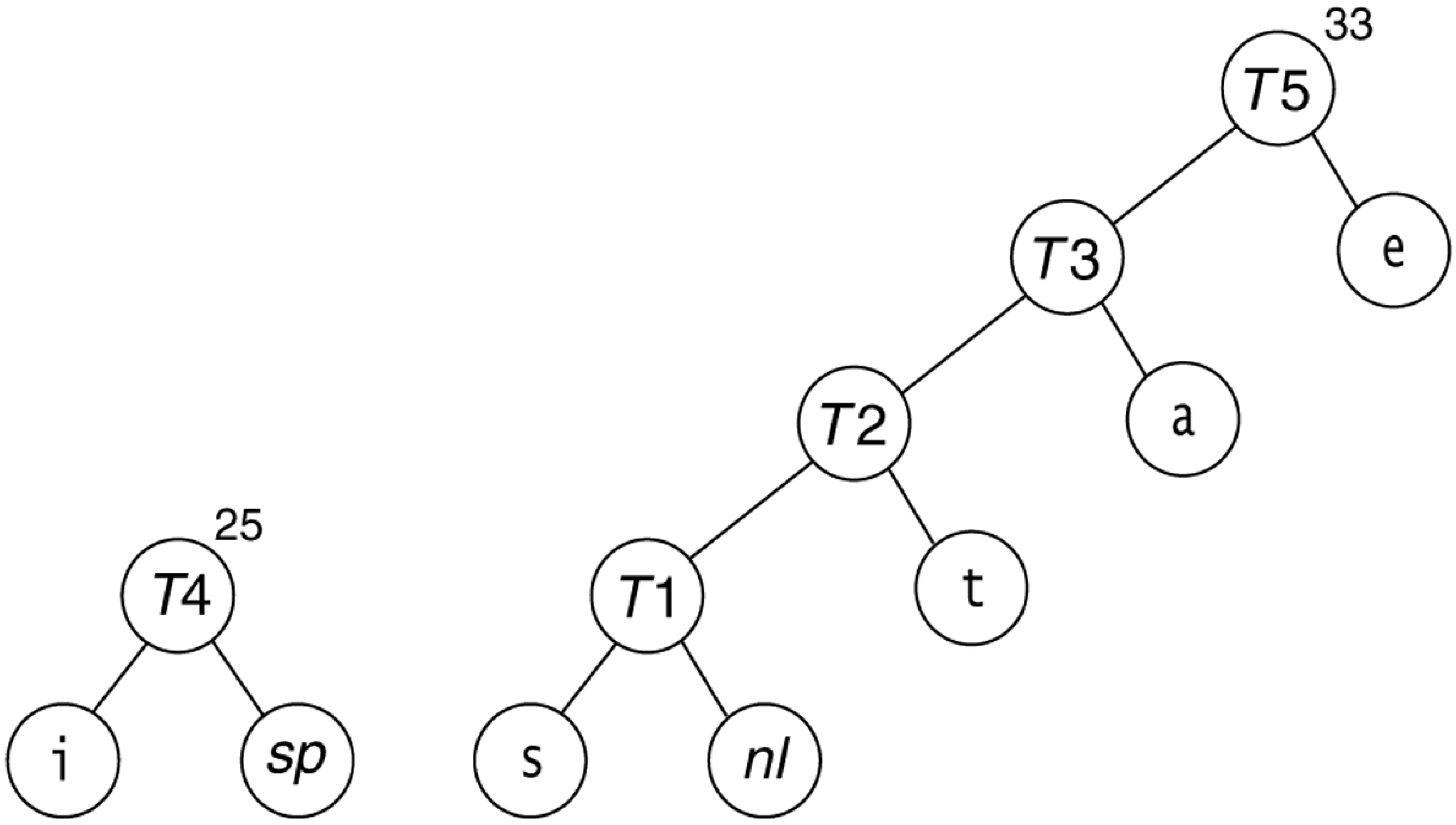
Building a Huffman’s Tree

Huffman's Algorithm Example - V



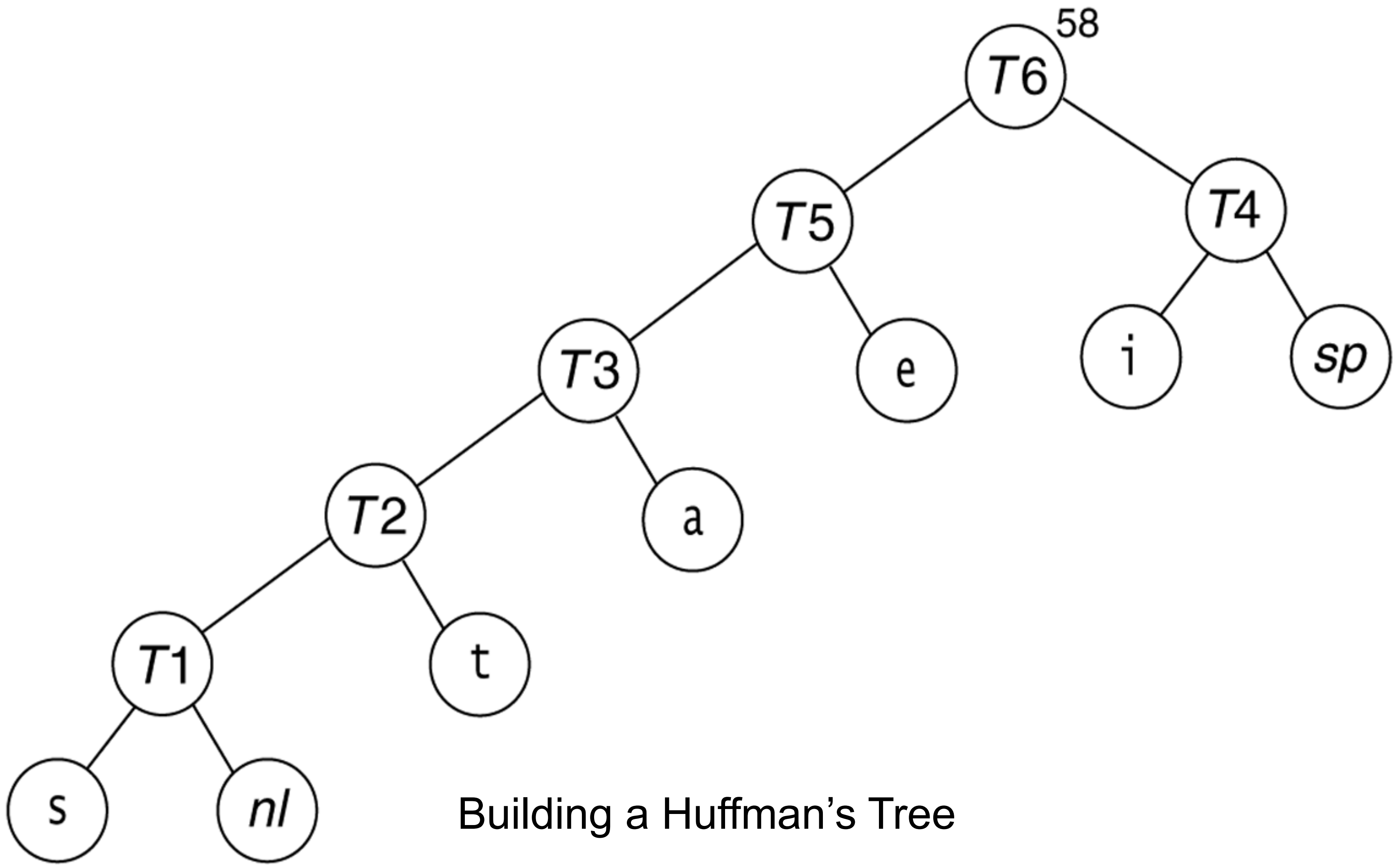
Building a Huffman's Tree

Huffman's Algorithm Example - VI



Building a Huffman's Tree

Huffman's Algorithm Example-VII



Building a Huffman's Tree

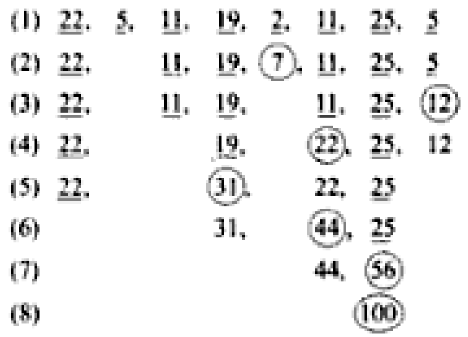
EXAMPLE 10.12 Let A, B, C, D, E, F, G, H be eight data items with the following assigned weights:

Data item:	A	B	C	D	E	F	G	H
Weight:	22	5	11	19	2	11	25	5

Construct a 2-tree T with a minimum weighted path length P using the above data as external nodes.

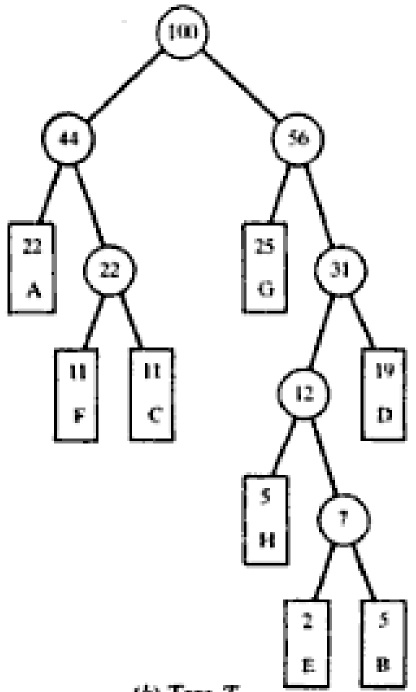
Apply the Huffman algorithm. That is, repeatedly combine the two subtrees with minimum weights into a single subtree as shown in Fig. 10-17(a). For clarity, the original weights are underlined, and a circled number indicates the root of a new subtree. The tree T is drawn from Step (8) backward yielding Fig. 10-17(b). (When splitting a node into two parts, we have drawn the smaller node on the left.) The path length P follows:

$$P = 22(2) + 11(3) + 11(3) + 25(2) + 5(4) + 2(5) + 5(5) + 19(3) = 280$$



(1)	$\underline{22}$,	$\underline{5}$,	$\underline{11}$,	$\underline{19}$,	$\underline{2}$,	$\underline{11}$,	$\underline{25}$,	$\underline{5}$
(2)	$\underline{22}$,		$\underline{11}$,	$\underline{19}$,	$\textcircled{7}$,	$\underline{11}$,	$\underline{25}$,	$\underline{5}$
(3)	$\underline{22}$,		$\underline{11}$,	$\underline{19}$,		$\underline{11}$,	$\underline{25}$,	$\textcircled{12}$
(4)	$\underline{22}$,			$\underline{19}$,		$\textcircled{22}$,	$\underline{25}$,	12
(5)	$\underline{22}$,			$\textcircled{31}$,		22,	$\underline{25}$	
(6)				31,		$\textcircled{44}$,	$\underline{25}$	
(7)						44,	$\textcircled{56}$	
(8)							$\textcircled{100}$	

(a) Huffman algorithm



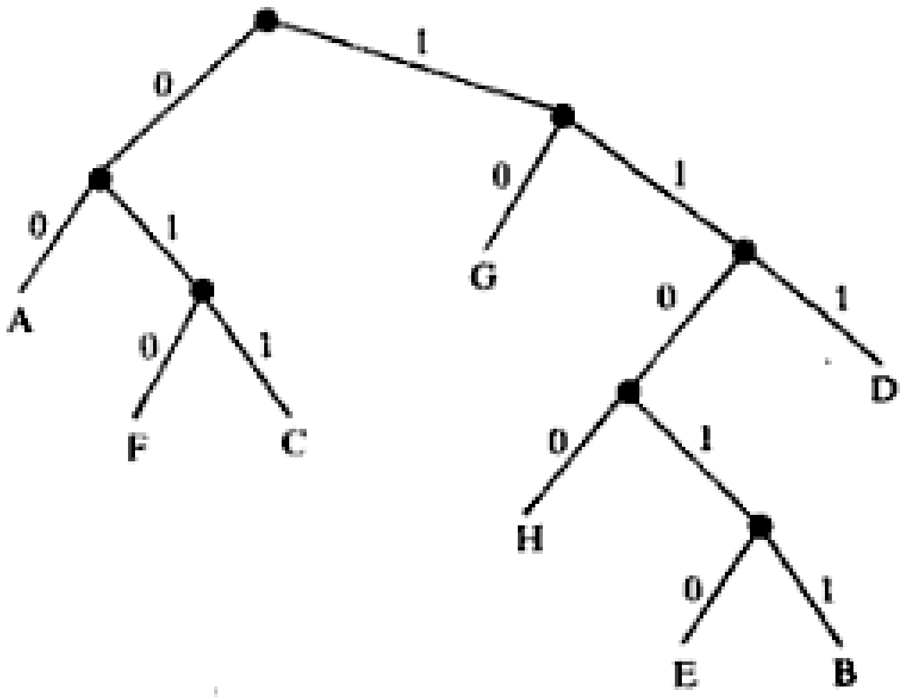
(b) Tree T

Huffman’s coding Example

EXAMPLE 10.13 Consider again the eight data items *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H* in Example 10-12. Suppose the weights represent the percentage probabilities that the items will occur. Assigning bit labels to the edges in the Huffman tree in Fig. 10-17(*b*), we obtain the tree *T* in Fig. 10-19. The reader can verify that the tree *T* yields the following code:

A: 00	*B*: 11011,	*C*: 011,	*D*: 111
E: 11010,	*F*: 010	*G*: 10,	*H*: 1100

This is an efficient coding of the data items.

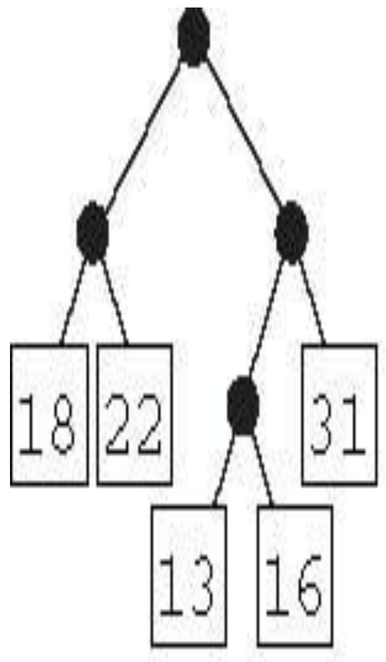


Weighted Path Lengths

Suppose T is a 2-tree with n external nodes, and suppose each external node is assigned a (nonnegative) weight. The *weighted path length* (or simply *path length*) P of the tree T is defined to be the sum

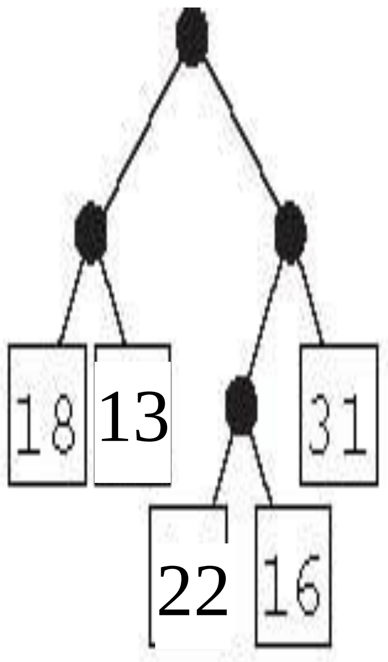
$$P = W_1L_1 + W_2L_2 + \cdots + W_nL_n$$

where W_i is the weight at an external node N_i and L_i is the length of the path from the root R to the node N_i. (The path length P exists even for nonweighted 2-trees where one simply assumes the weight 1 at each external node.)



18+22+31=71

13+16=29



18+13+31=62

22+16=38

WEPL = 2(71) + 3(29) = 229

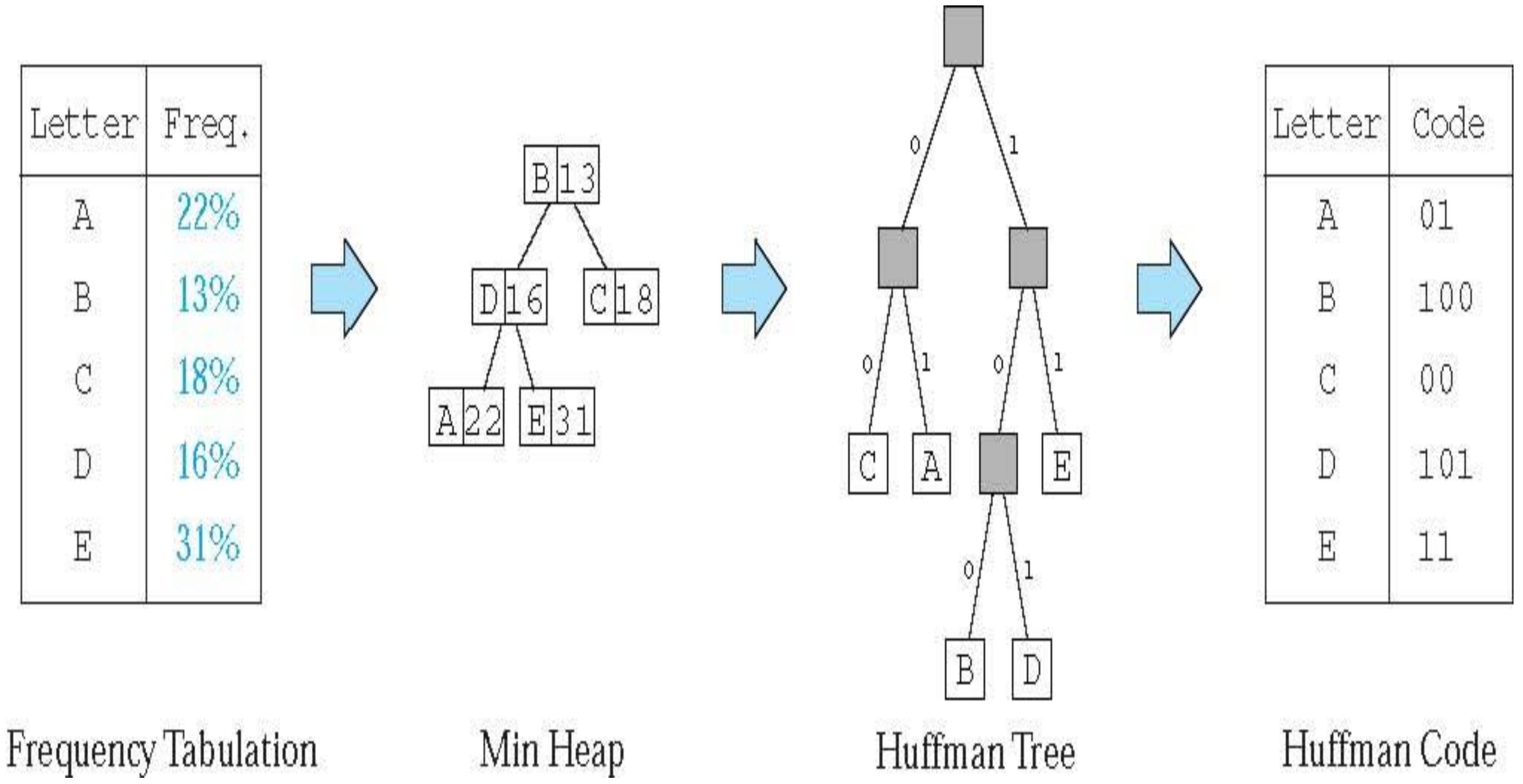
WEPL = 2(62) + 3(38) = 238

Huffman Codes

Huffman code is a technique for compressing data.

- Huffman codes is a text compression method, which relies on the relative frequency (i.e., the number of occurrences of a symbol) with which different symbols appear in a text
- Uses extended binary trees
- Huffman tree is a binary tree with minimum weighted external path length for a given set of frequencies (weights)

Huffman Algorithm



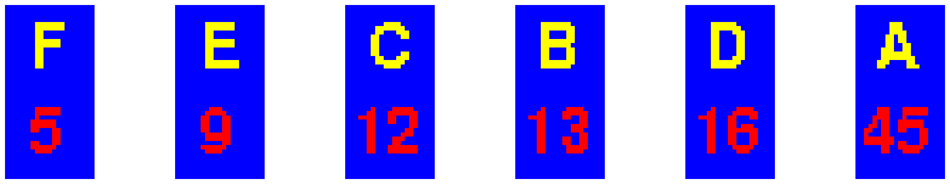
Letter	Freq.
A	22%
B	13%
C	18%
D	16%
E	31%

Letter	Code
A	01
B	100
C	00
D	101
E	11

Frequency Tabulation Min Heap Huffman Tree Huffman Code

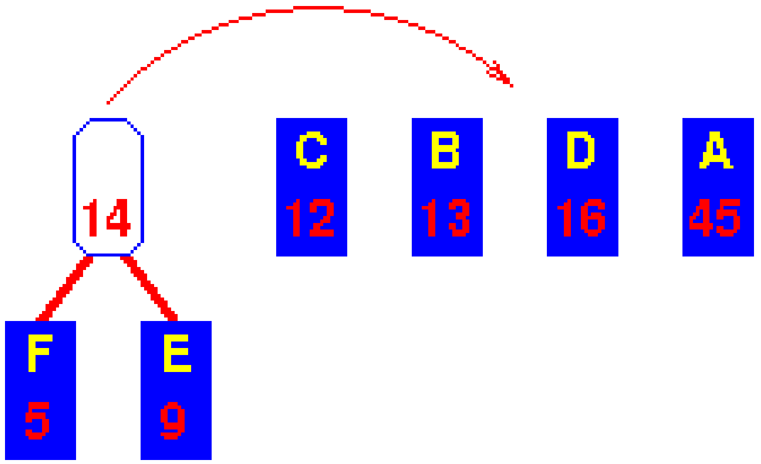
Huffman Encoding - Operation

Initial sequence
Sorted by frequency



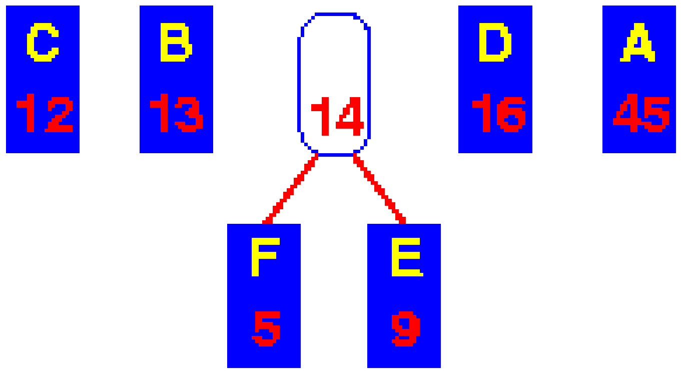
Combine lowest two into sub-tree

Move it to correct place



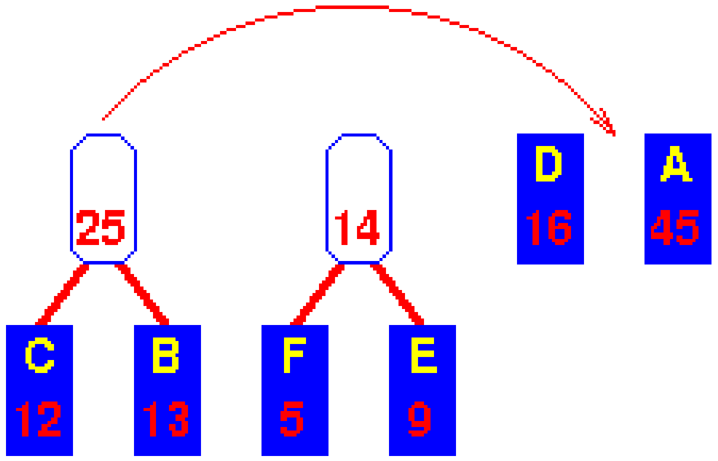
Huffman Encoding - Operation

After shifting sub-tree to its correct place ...



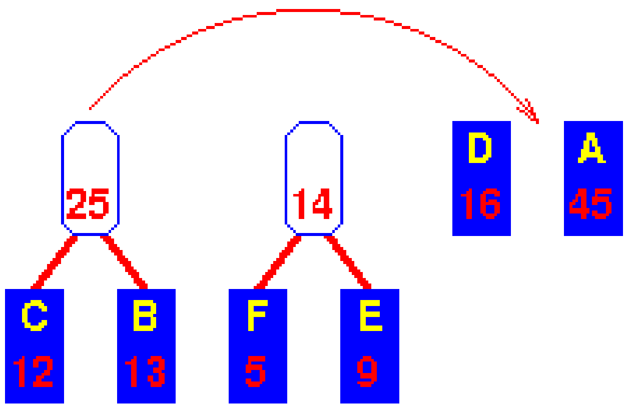
Combine next lowest pair

Move sub-tree to correct place



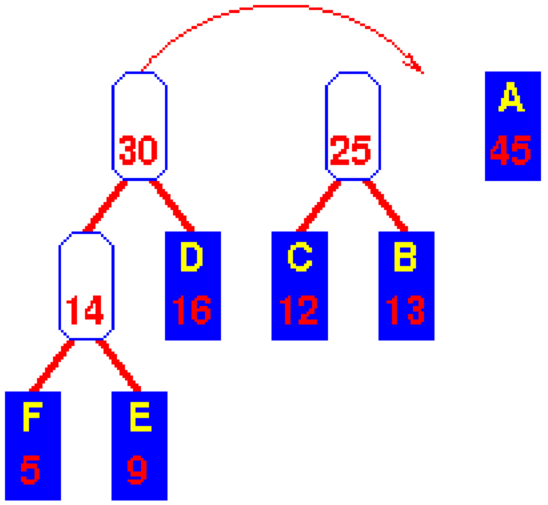
Huffman Encoding - Operation

Move the new tree to the correct place ...

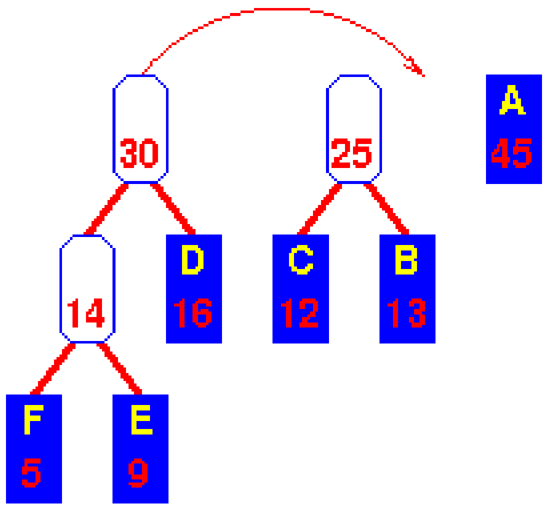


Now the lowest two are the "14" sub-tree and D

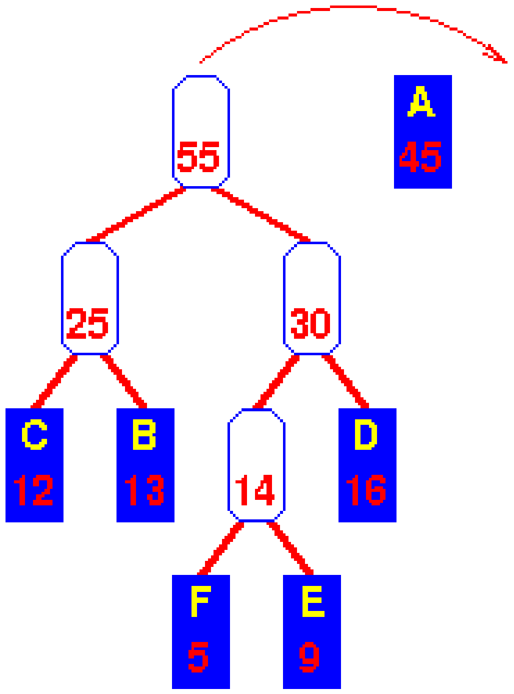
Combine and move to correct place



Huffman Encoding - Operation



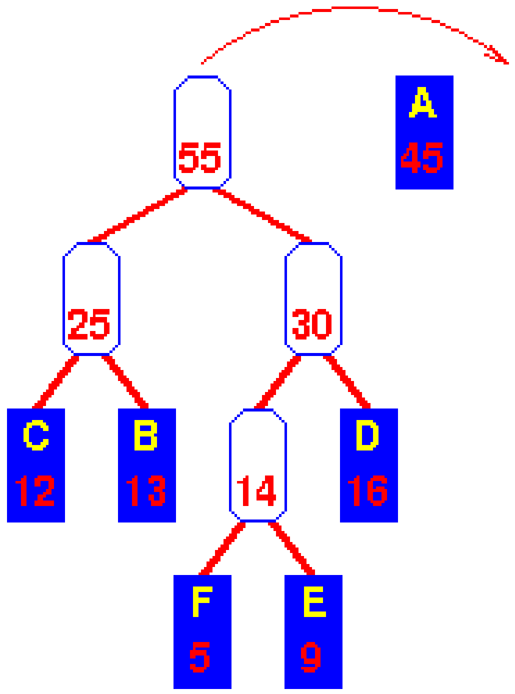
Move the new tree to the correct place ...



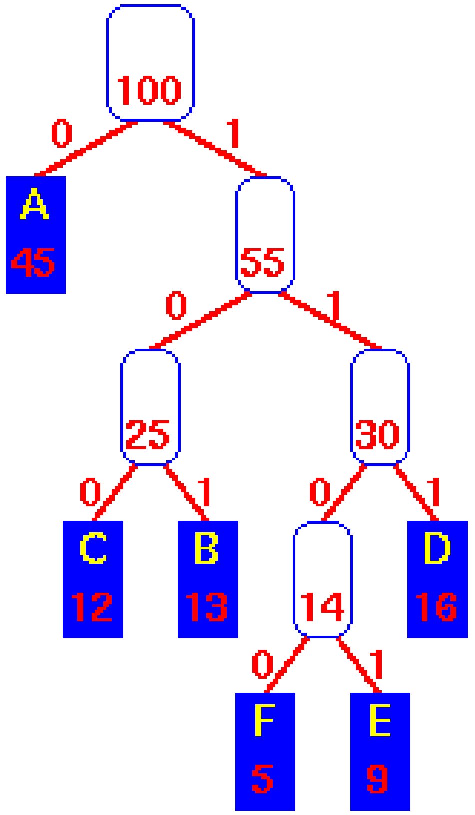
Now the lowest two are the the “25” and “30” trees

Combine and move to correct place

Huffman Encoding - Operation

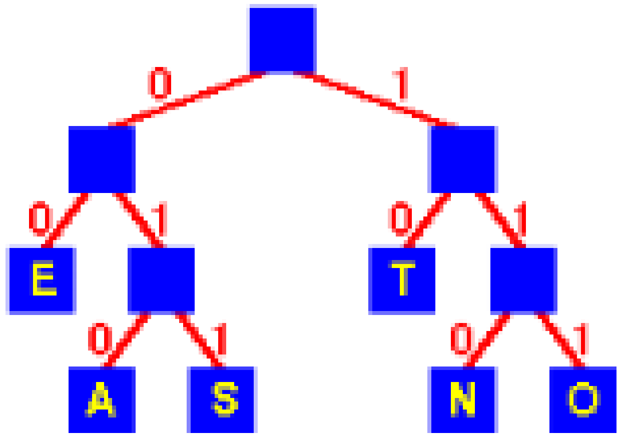


Combine last two trees



Huffman Encoding - Operation

- Encoding: Concatenate the codewords representing each characters of the file.



- String Encoding TEA 10 00 010
- SEA 011 00 010
- TEN 10 00 110

General Trees

A general tree (tree) is defined to be a nonempty finite set T of elements, called nodes, such that:

•T contains a distinguished element R, called the root of T

•The remaining elements of T form an ordered collection of zero or more disjoint trees T1, T2, …….., Tm.

•Trees T1, T2, …….., Tm are called subtrees of the root R, and the roots of T1, T2, …….., Tm are called successors of R.

Figure 10-20 is a picture of a general tree *T* with 13 nodes,

A, B, C, D, E, F, G, H, J, K, L, M, N

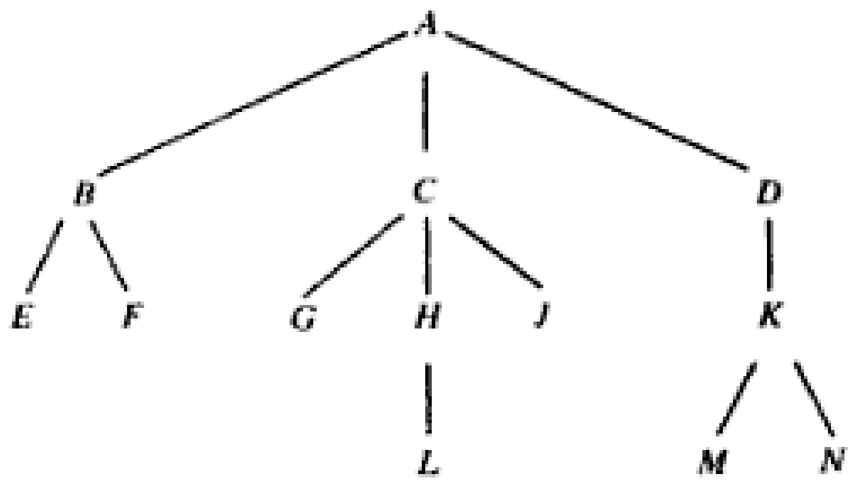
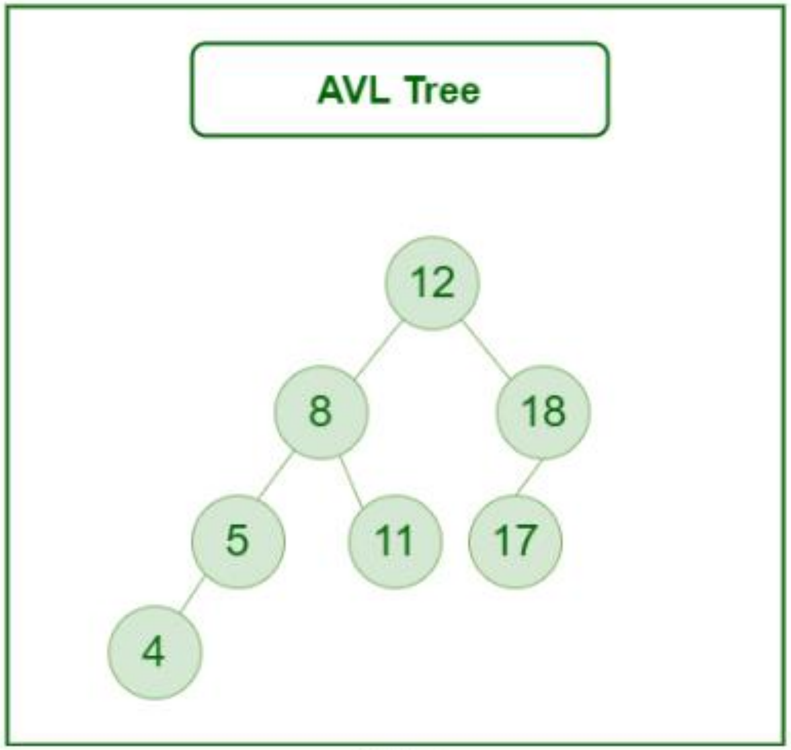


Fig. 10-20

AVL Tree Data Structure

*An **AVL tree** defined as a self-balancing **[Binary Search Tree](#)** (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.*



AVL tree

Operations on an AVL Tree

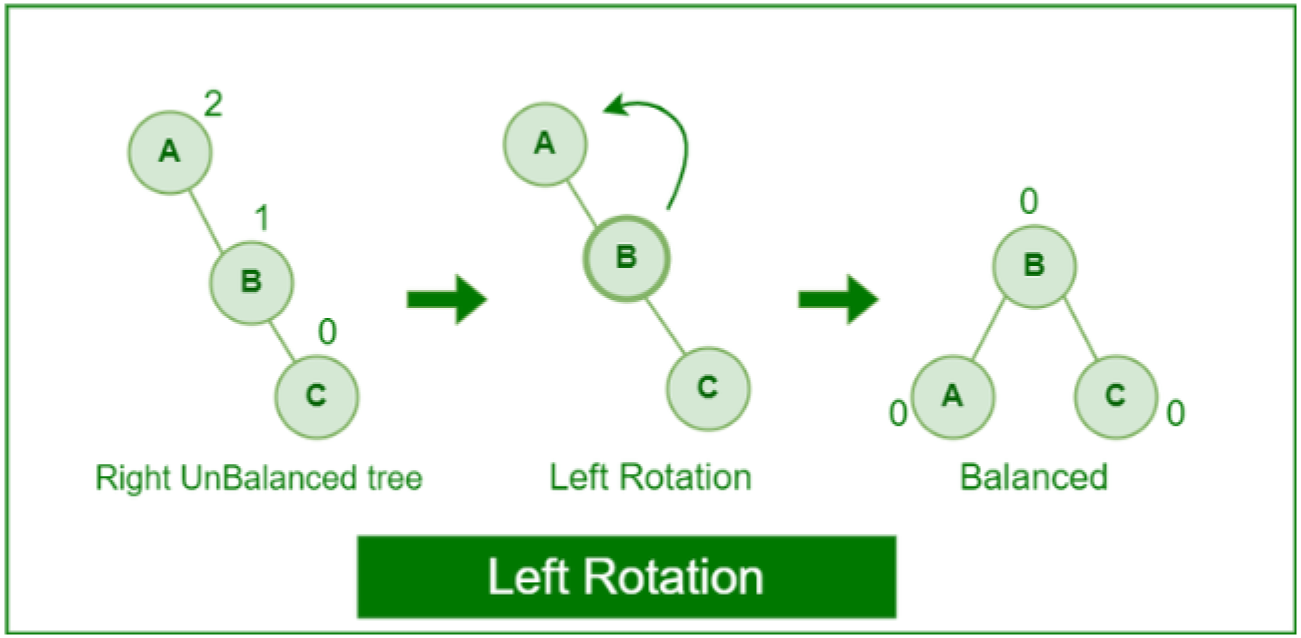
- Insertion
- Deletion
- Searching

Operations on an AVL Tree

Rotating the subtrees in an AVL Tree:

Left Rotation:

When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



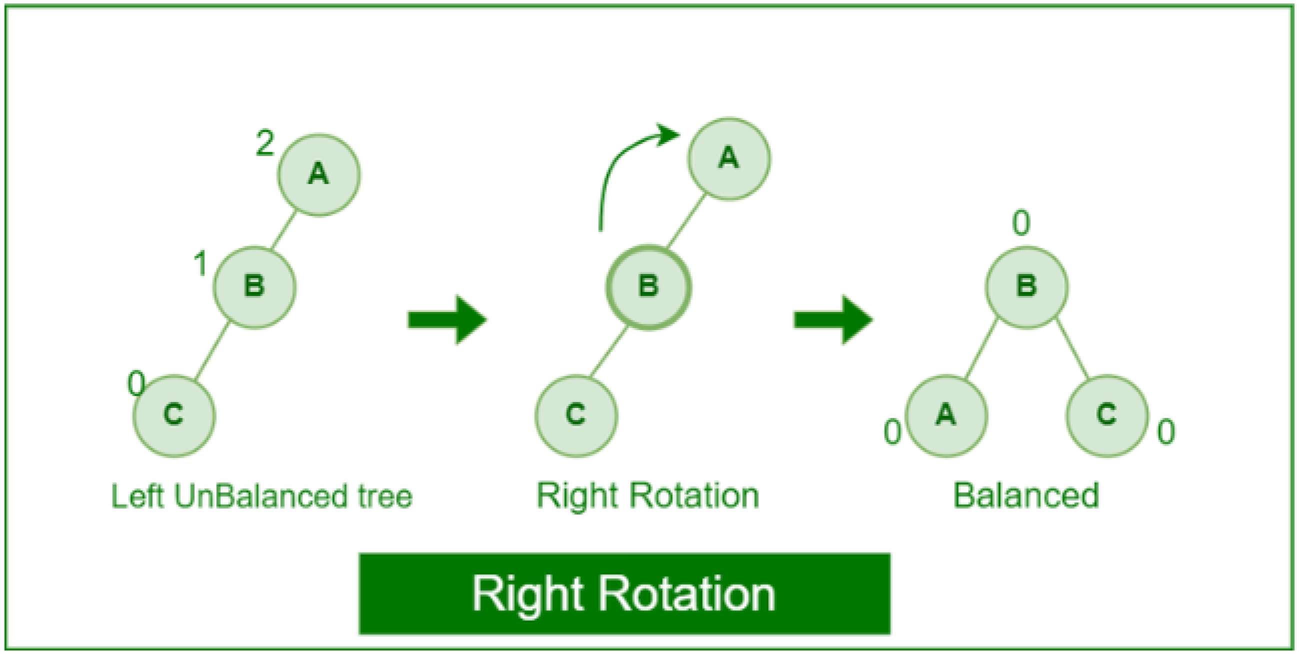
Left-Rotation in AVL tree

Operations on an AVL Tree

Rotating the subtrees in an AVL Tree:

Right Rotation:

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



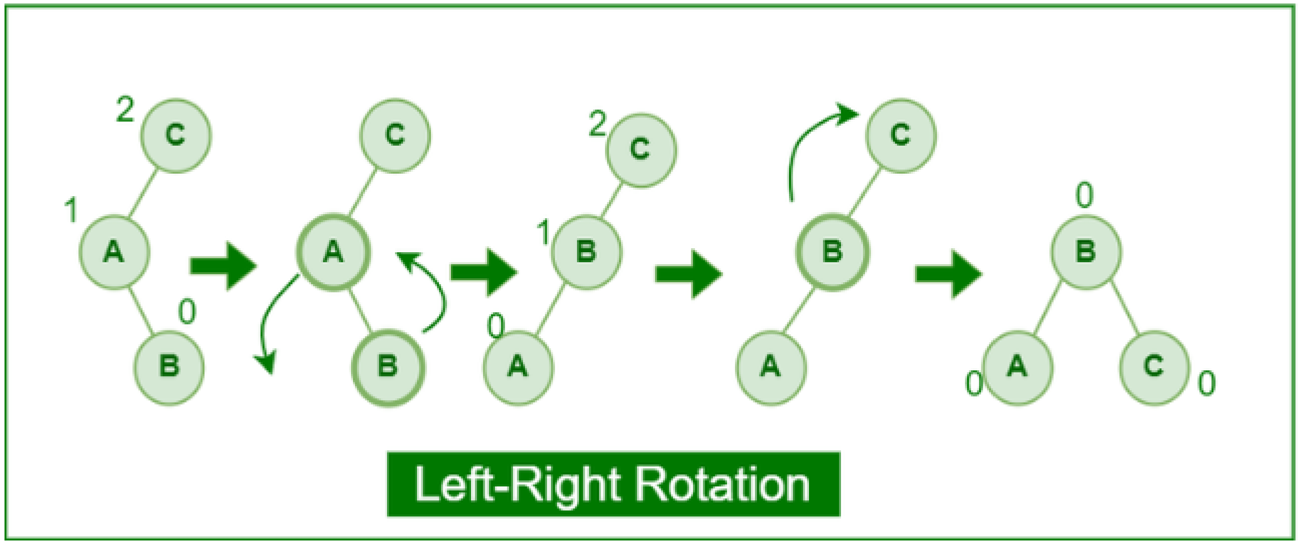
Right Rotation in AVL tree

Operations on an AVL Tree

Rotating the subtrees in an AVL Tree:

Left-Right Rotation:

A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



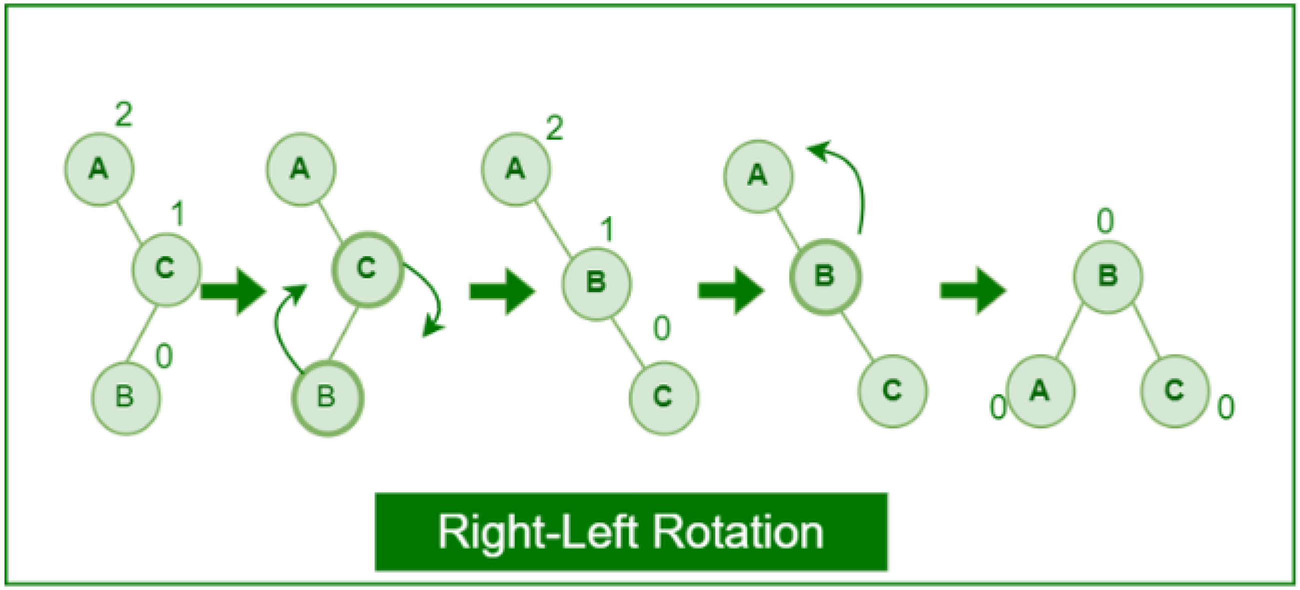
Left-Right Rotation in AVL tree

Operations on an AVL Tree

Rotating the subtrees in an AVL Tree:

Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



Right-Left Rotation in AVL tree

Applications of AVL Tree

1.It is used to index huge records in a database and also to efficiently search in that.

2.For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.

3.Database applications, where insertions and deletions are less common but frequent data lookups are necessary

4.Software that needs optimized search.

5.It is applied in corporate areas and storyline games.

Advantages of AVL Tree

- AVL trees can self-balance themselves.
- It is surely not skewed.
- It provides faster lookups than Red-Black Trees
- Better searching time complexity compared to other trees like binary tree.
- Height cannot exceed log(N), where, N is the total number of nodes in the tree.

Disadvantages of AVL Tree

It is difficult to implement.
It has high constant factors for some of the operations.
Less used compared to Red-Black trees.
Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
Take more processing for balancing.