ICE-2231
(Data Structure)

Lecture on
Chapter-4: Tree and Graphs

By

Dr. M. Golam Rashed

(golamrashed@ru.ac.bd)

Department of Information and Communication Engineering (ICE)
University of Rajshahi, Rajshahi-6205, Bangladesh

Trees and Graphs: Basic terminology, Binary trees, Binary tree representation, Tree traversal algorithms, Extended binary tree, Huffman codes/algorithm, Graphs, Graph representation, Shortest path algorithm and transitive closure, Traversing a graph.

- Linear Data Structures
  - ✓ Strings
  - ✓ Arrays
  - ✓ Linked Lists
  - ✓ Stacks, and
  - ✓ Queues
- Nonlinear Data Structures
  - ✓ Tree,
  - ✓ Graphs

Tree Data Structure

- Tree structure is mainly used to represent data containing a hierarchical relationship between elements. For example....
    - Records,
    - Family tress, and
    - Tables of contents.
- In this chapter, we investigate a special kind of tree, called a ***binary tree***
    - It can be maintained in the computer easily

Binary Tree

- T is a finite set of elements , called *Nodes,* such that
  - T is empty (called null tree or empty tree), or
  - T contains a distinguished node R, called a root of T, and the remaining nodes form an ordered pair of disjoint binary trees T1, and T2
- Any node of T has 0, 1 , or 2 children.

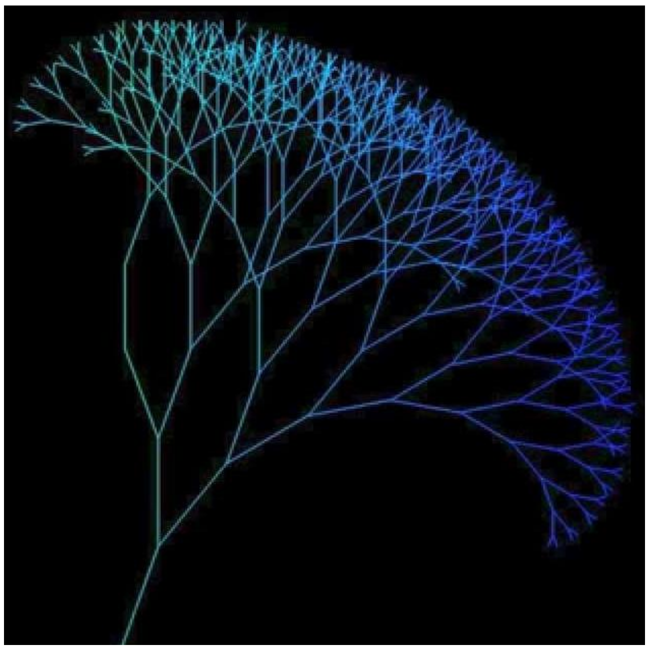
Basic Binary Tree Concept

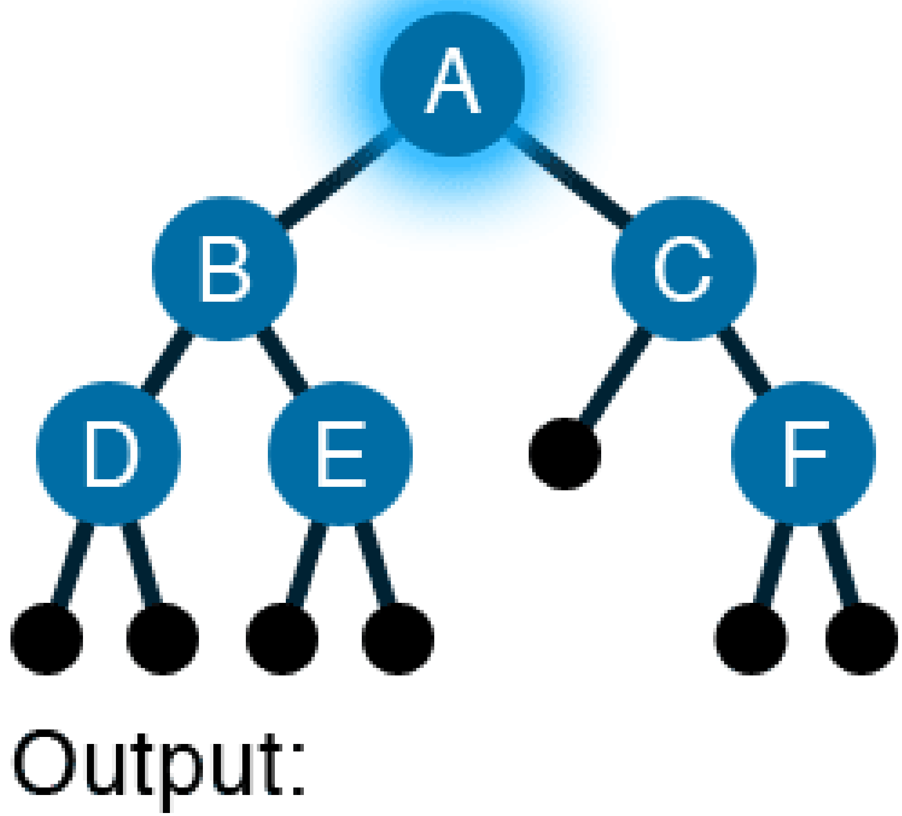
- If T does contain a Root R , then the two trees T_1 and T_2 are called, respectively , the left and right sub-trees of R.
- If T_1 is nonempty, then its root is called the left successor of R; similarly,
- If T_2 is nonempty, then its root is called the right successor of R;

Binary Tree Illustration

ICE 2231

- A Binary Tree T is frequently presented by mean of a diagram





See Example 7.1 and 7.2 for understanding

Tree Terminology

- Suppose N is a node in T with left successor S1 and right successor S2.
    - N is called the parent (or father) of S1 and S2.
- Analogously, S1 is called the left child (or son) of N, and S2 is called the right child (or son) of N
- S1 and S2 are said to be siblings (or brothers)
- Two or more nodes with the same parents are called siblings.

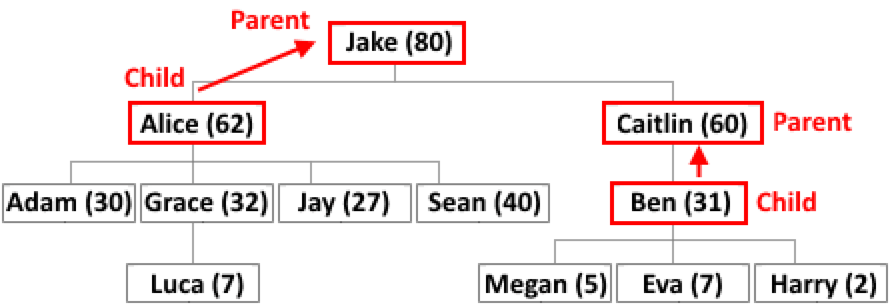
Tree Terminology

ICE 2231

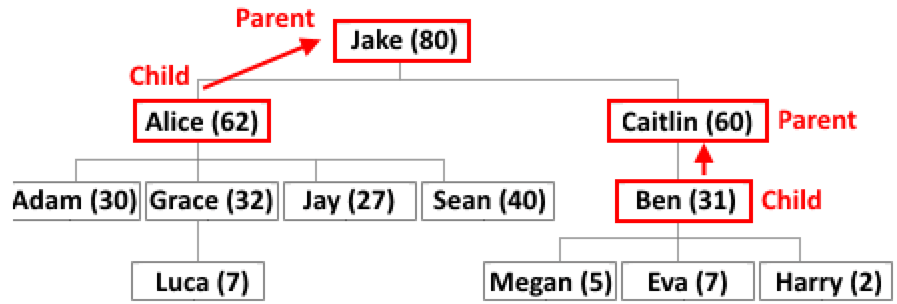
- An ancestor is any node in the path from the root to the node.
- A descendant is any node in the path below the parent node; that is, all nodes in the paths from a given node to a leaf are descendants of that node.

Tree Terminology

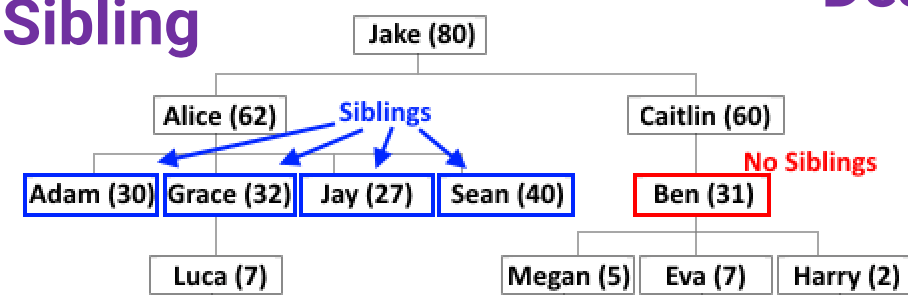
Parent



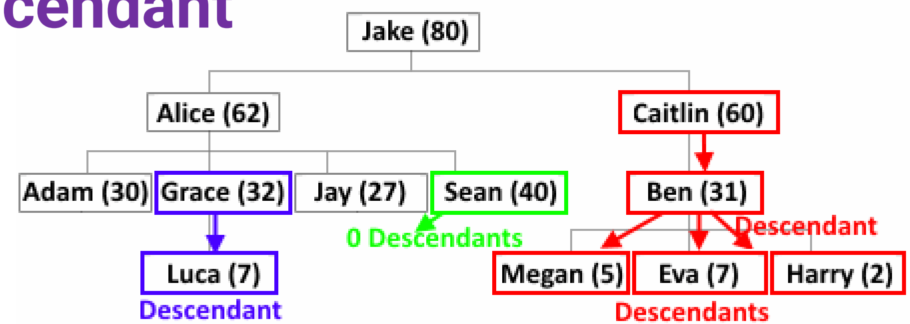
Child



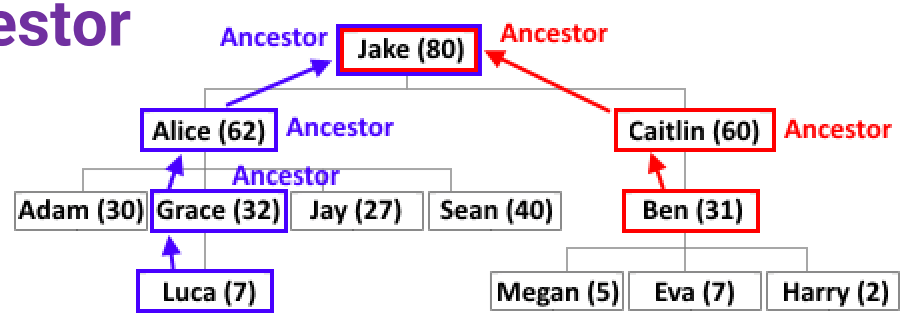
Sibling



Descendant



Ancestor

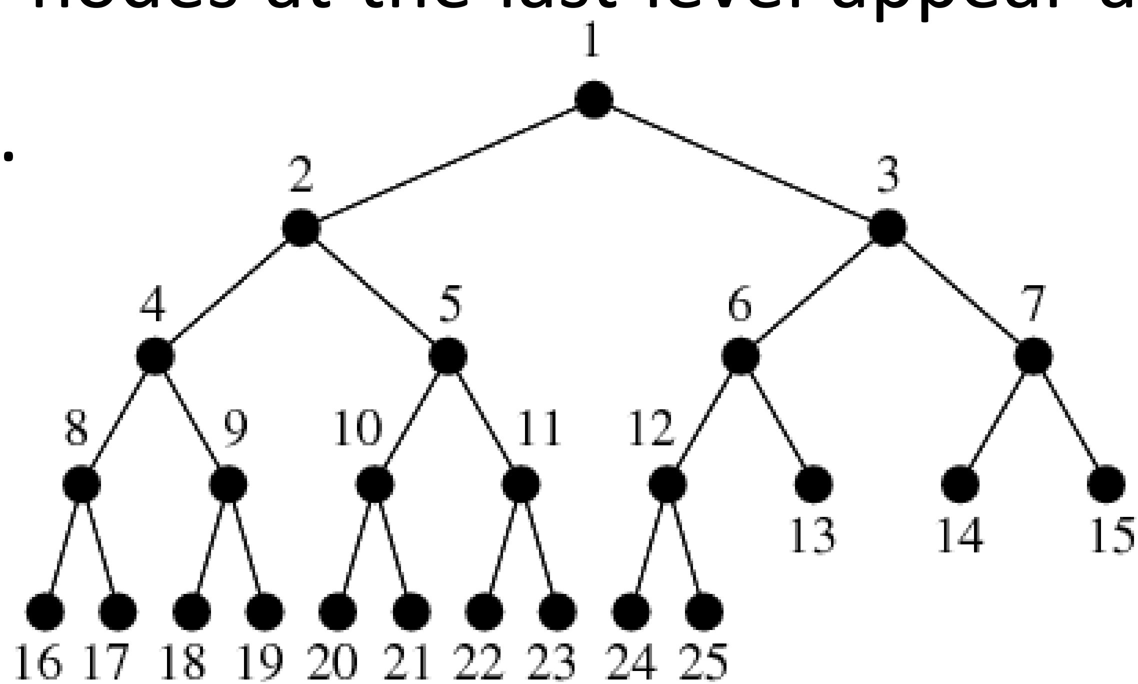


Tree Terminology

- A path is a sequence of nodes in which each node is adjacent to the next node.
- The level or dept of a node is its distance from the root. The root is at level 0, its children are at level 1, etc.
- The height of the tree is the level of the leaf in the longest path from the root plus 1.
- A subtree is any connected structure below the root. The first node in the subtree is known is the root of the subtree.

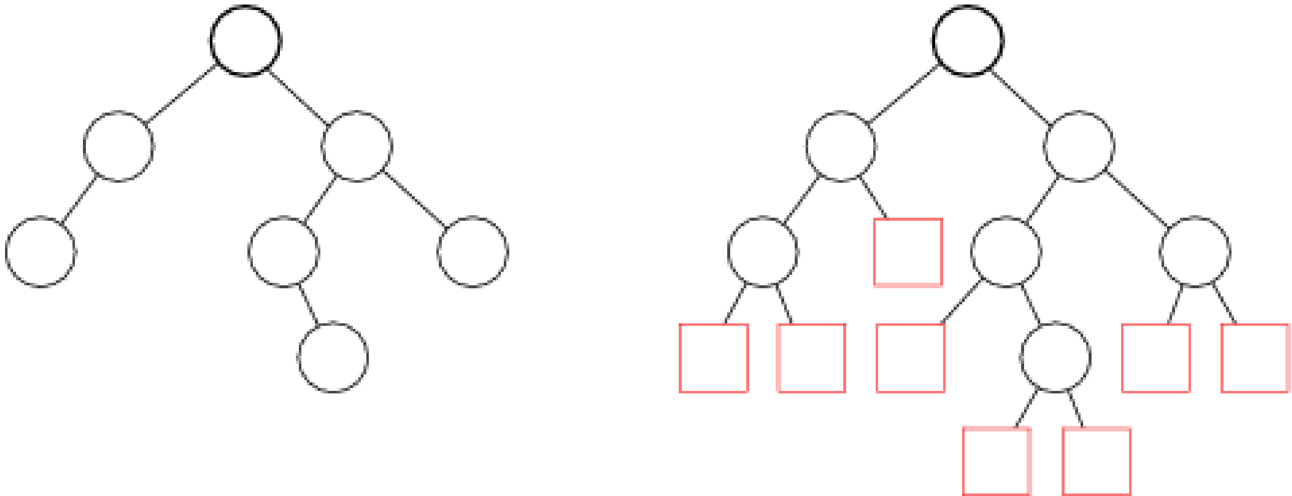
Complete Tree

- A tree is said to be **complete if all its levels, except possibly the last have the maximum number of possible nodes**, and
- if all the nodes at the last level appear as far left as possible.



Extended Binary Tree: 2-Tree

- A binary tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children.
- In such a case, the nodes, with 2 children are called internal nodes, and the node with 0 children are called external node.



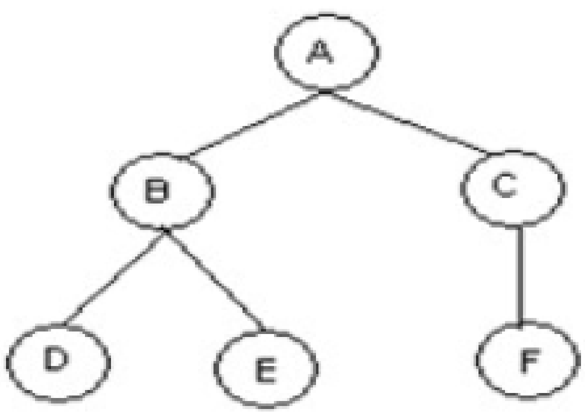
Representing Binary Tree in Memory

- Let T be a binary tree.
- There are two ways of representing T in memory:
  - First and usual was is called link representation of T
  - The second way uses a single array, called the sequential representation of T
- The main requirement of any representation of T is that one should have direct access to the root R of T and, given any node N of T, one should have direct access to the children of N.

Linked Representation of Binary Trees

Linked Representation of Binary Tree

- Use Three parallel arrays Info, Left and Right and a pointer variable Root.
- Info[K]: Contains data at node N.
- Left[K]: Contains location of left child of N
- Right[K]: Contains location of right child of N
- Root: Contains location of Root
- Example:



Root

	Info	Left	Right
1	C	0	10
2	D	0	0
3			
4	E	0	0
5	A	7	1
6			
7	B	2	4
8			
9			
10	F	0	0

Figure: Binary Tree T and its linked representation.

Sequential Representation of Binary Trees

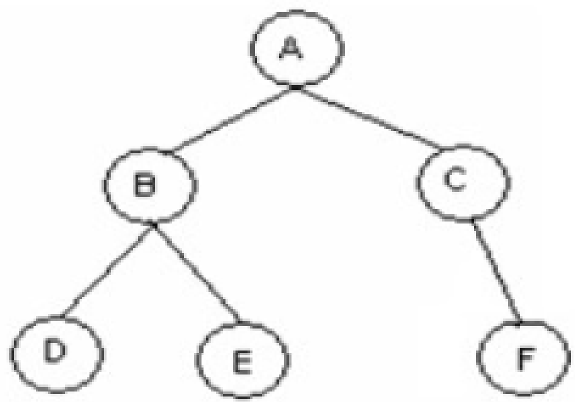
- Use only a single liner array Tree.

(a)Tree[1] represents the Root of T.

(b)If node N is in Tree[K], then its left child is in Tree[2K] and right child is in Tree[2K+1].

(c)Tree[1] = NULL indicates T is empty.

- Example:



Tree =

1	2	3	4	5	6	7	8	9	10
A	B	C	D	E		F			

Figure: Binary Tree T and its sequential representation.

Binary Tree Traversal Methods

- In a traversal of a binary tree, each element of the binary tree is visited exactly once.
- During the visit of an element, all action (display, evaluate the operator, etc.) with respect to this element is taken.

Binary Tree Traversal Methods

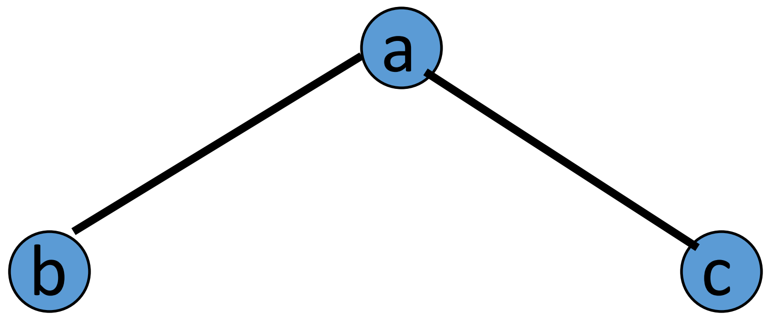
- There are three standard ways of traversing a binary tree T with Root R:
  - Preorder
    - Process the Root R
    - Traverse the left subtree of R in preorder
    - Traverse the right subtree of R in preorder
  - Inorder
    - Traverse the left subtree of R in preorder
    - Process the root R
    - Traverse the right subtree of R in inorder
  - Postorder
    - Traverse the left subtree of R in postorder
    - Traverse the right subtree of R in postorder
    - Process the root R

Binary Tree Traversal Methods

Preorder Traversal

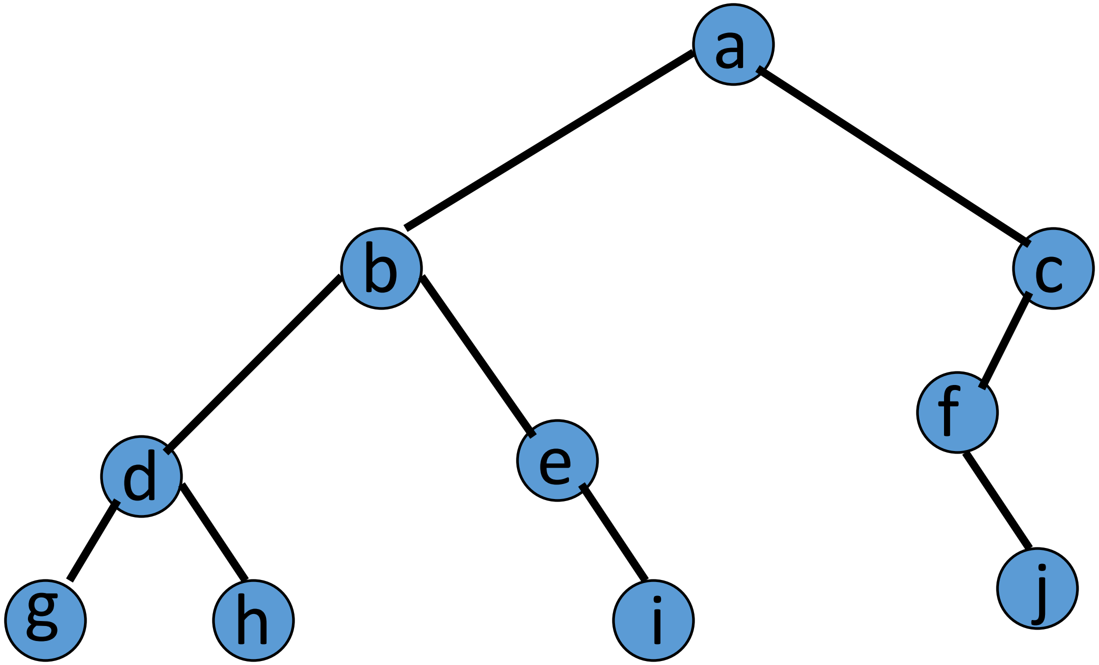
```
preOrder(treePointer ptr)
{
          (ptr !=       )
    {
          visit(ptr);
          preOrder(ptr->leftChild);
          preOrder(ptr->rightChild);
    }
}
```

Preorder Example (Visit = print)



a b c

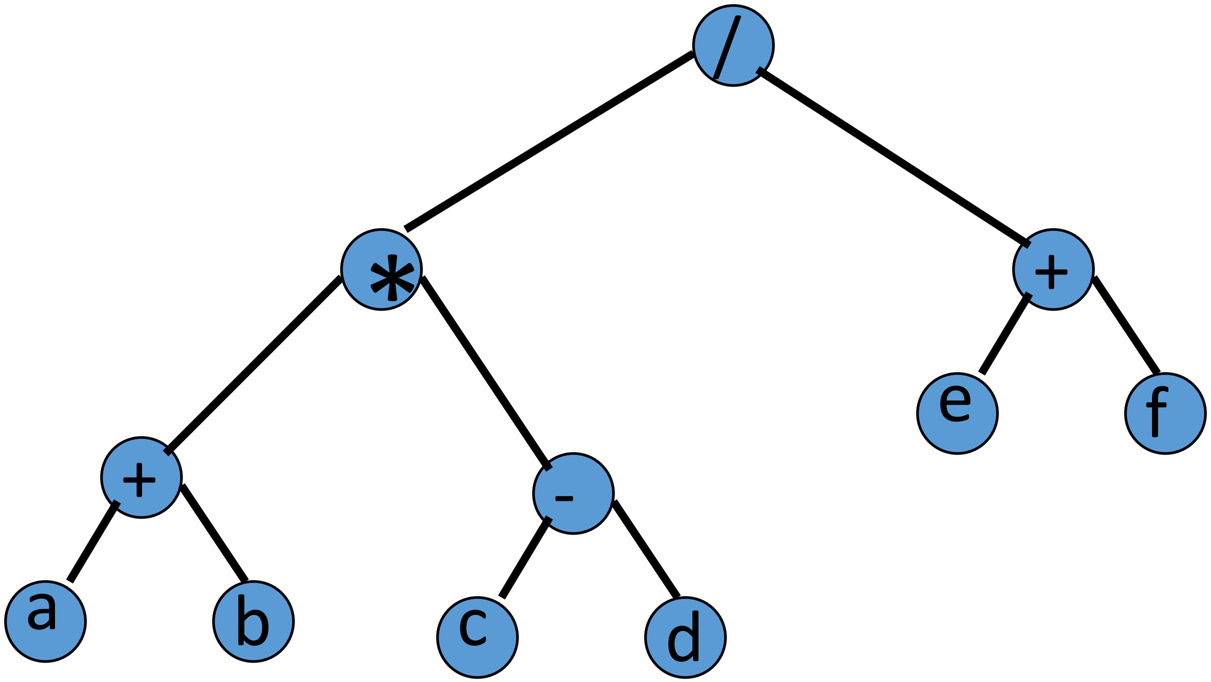
Preorder Example (Visit = print)



a b d g h e i c f j

Preorder Of Expression Tree

ICE 2231



/ *+ab- c d+e f

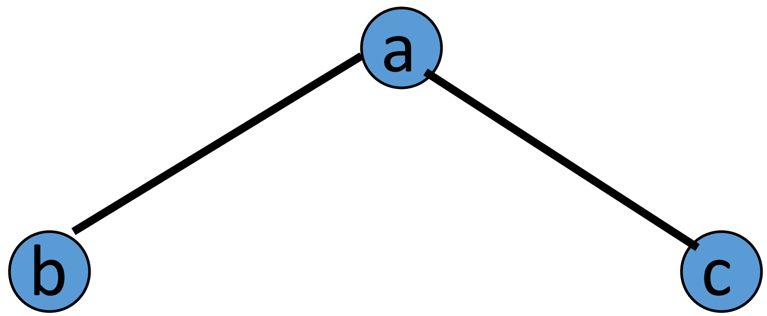
Gives prefix form of expression!

Inorder Traversal

ICE 2231

```
inOrder(treePointer ptr)
{
        (ptr !=      )
    {
        inOrder(ptr->leftChild);
        visit(ptr);
        inOrder(ptr->rightChild);
    }
}
```

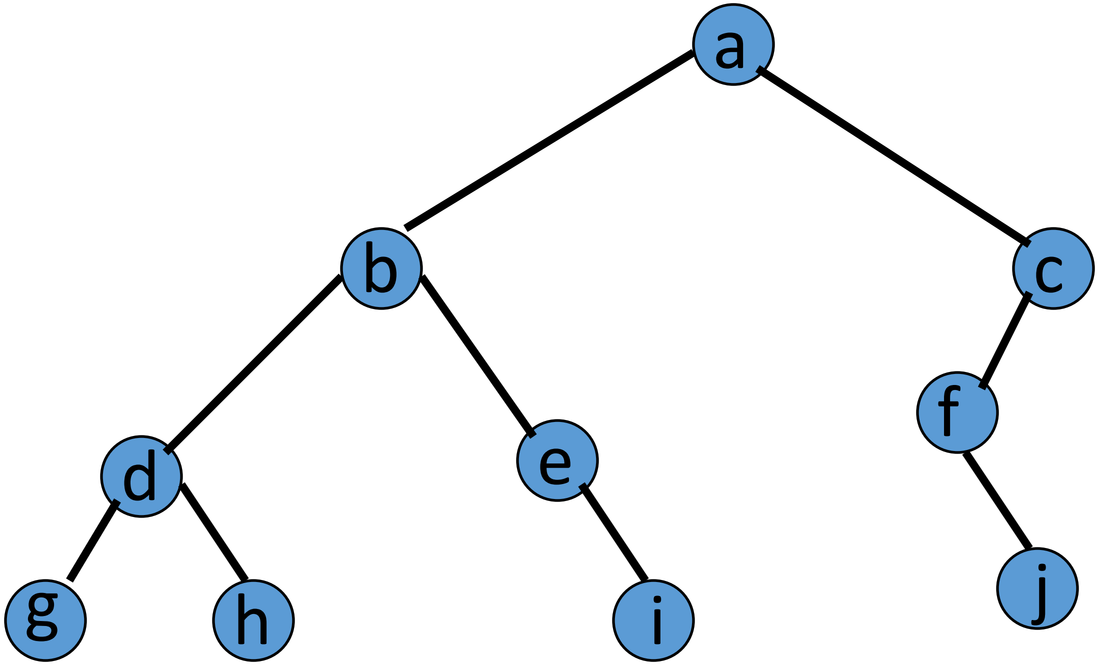
Inorder Example (Visit = print)



b a c

Inorder Example (Visit = print)

ICE 2231

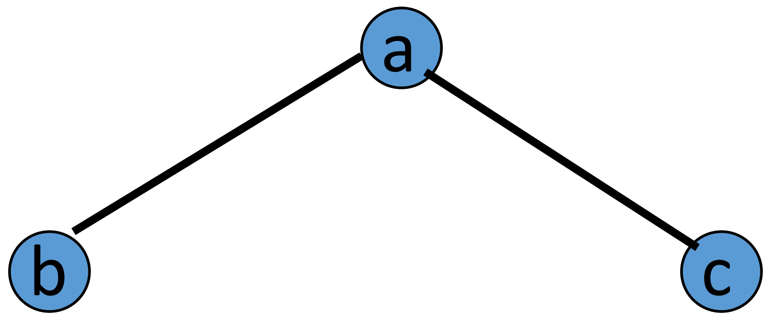


g d h b e i a f j c

Postorder Traversal

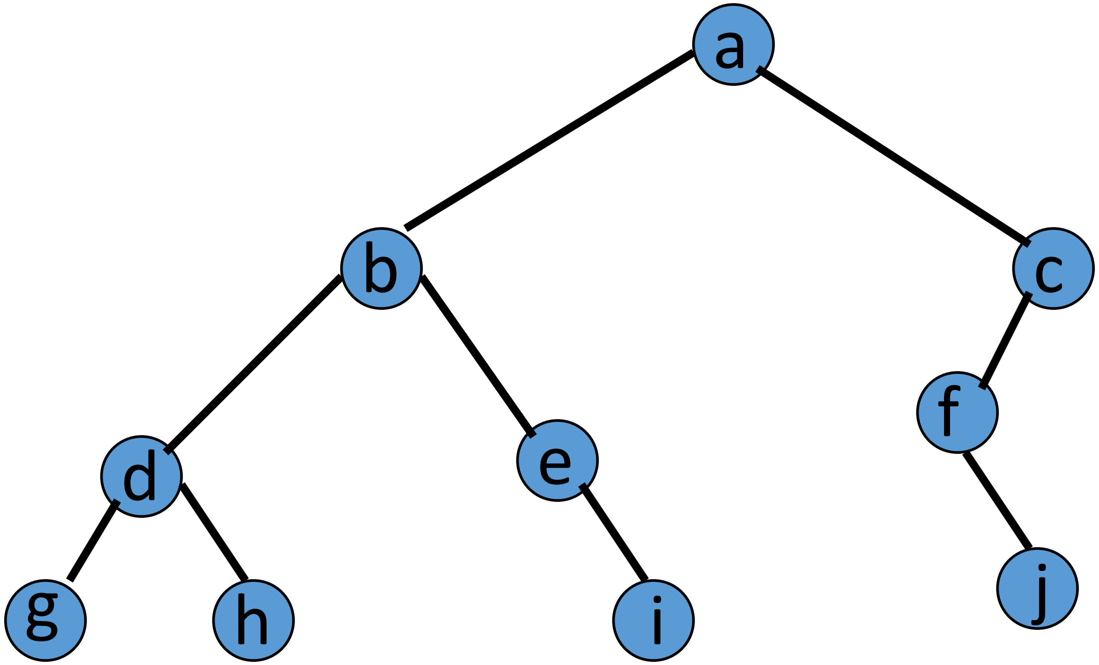

```
postOrder(treePointer ptr)
{
        (ptr != NULL)
    {
        postOrder(ptr->leftChild);
        postOrder(ptr->rightChild);
        visit(ptr);
    }
}
```

Postorder Example (Visit = print)



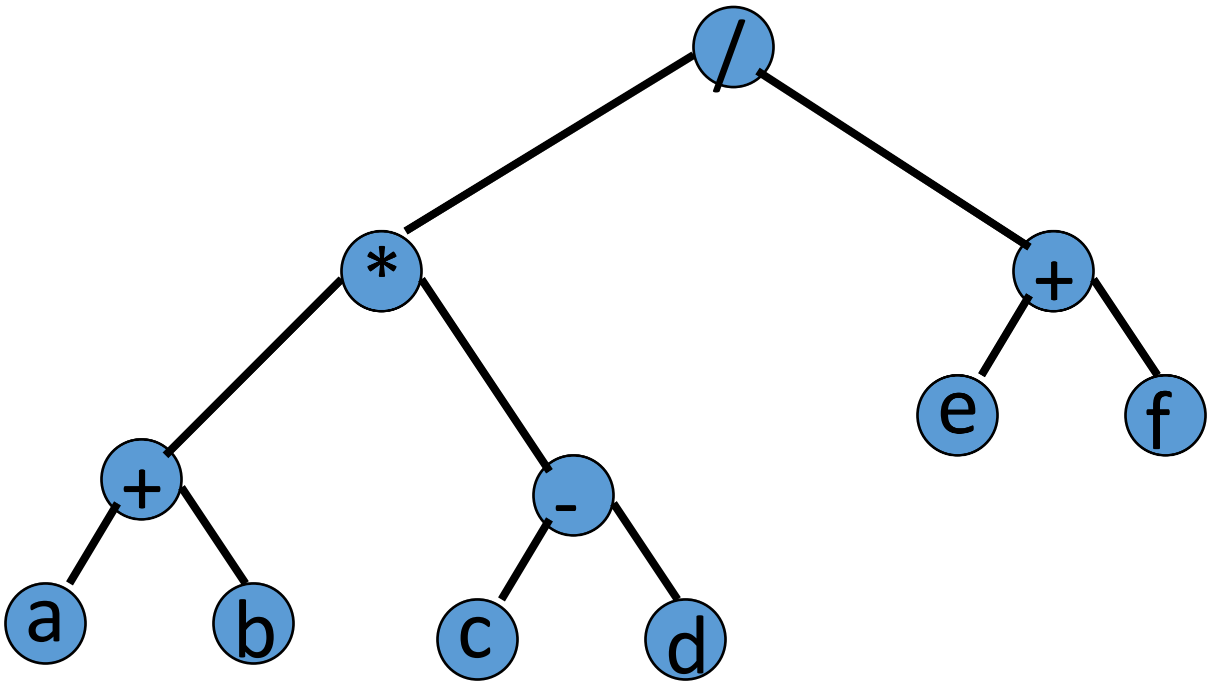
b c a

Postorder Example (Visit = print)



ghdi ebj f c a

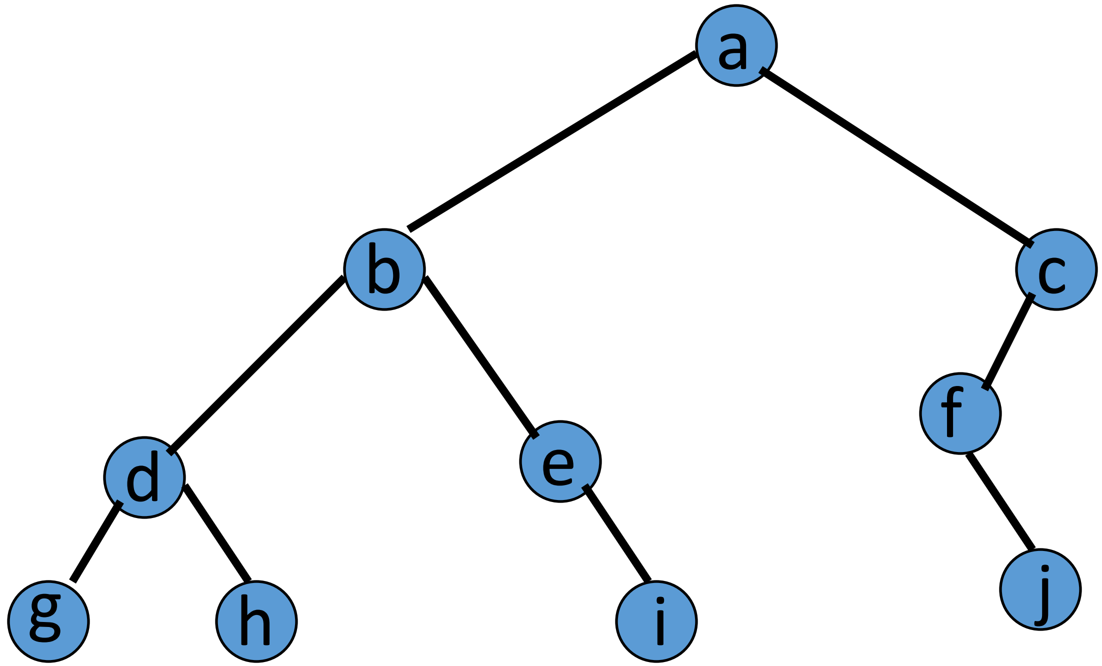
Postorder Of Expression Tree



a b + c d - * e f + /

Gives postfix form of expression!

Traversal Applications



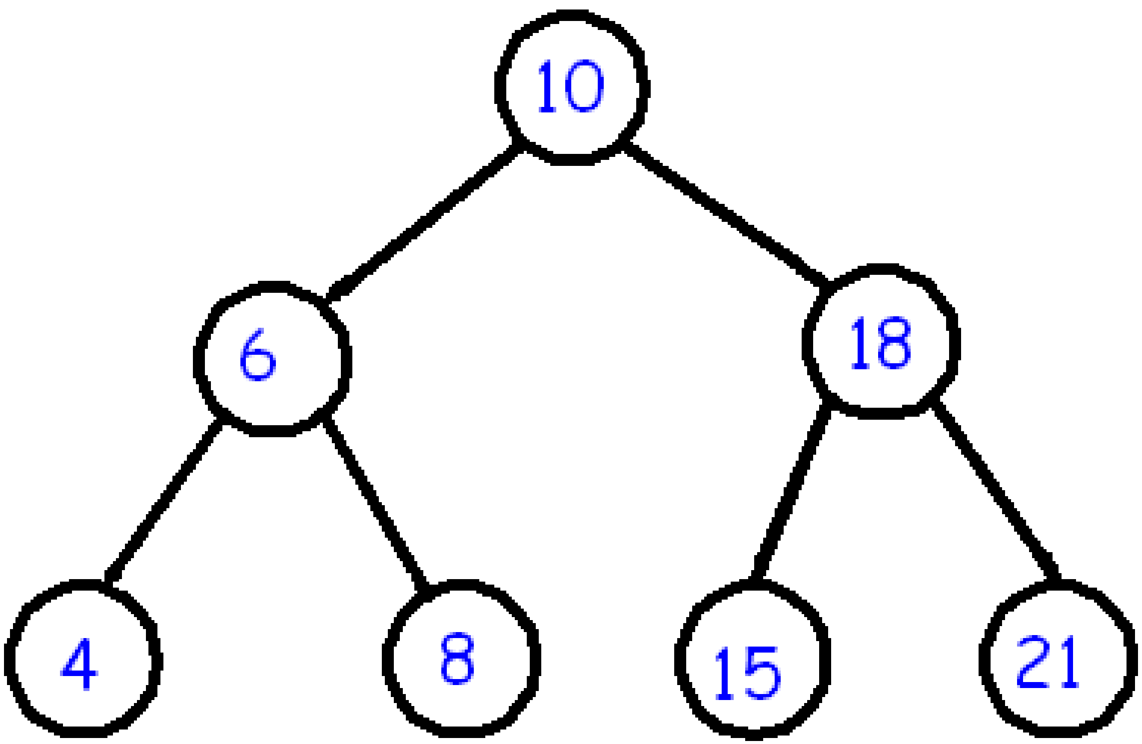
- Make a clone.
- Determine height.
- Determine number of nodes.

Traversal Applications

- Go through the following related example:
  - Example 7.5
  - Example 7.6
  - Example 7.7

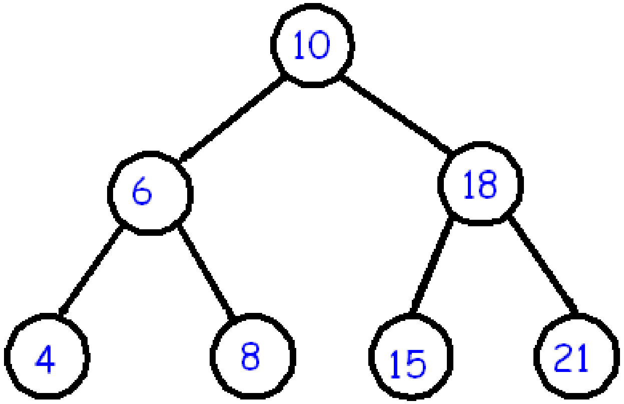
Binary Search Trees

- We consider a particular kind of a binary tree called a Binary Search Tree (**BST**).
- The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving.



Binary Search Trees

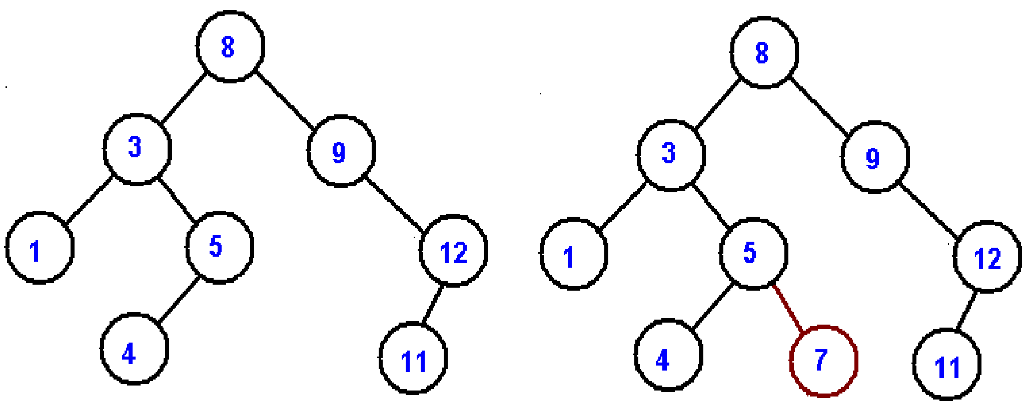
- A **BST** is a binary tree where nodes are ordered in the following way:
  - each node contains one key (also known as data)
  - the keys in the left sub-tree are less then the key in its parent node, in short L < P;
  - the keys in the right sub-tree are greater the key in its parent node, in short P < R;
  - duplicate keys are not allowed.



- In the following tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10.
- Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes:

Searching and Inserting in Binary Search Trees

- Suppose **T** is a **BST** and an **ITEM** of information is given.
- The searching and inserting will be given by a single search and insertion algorithm which is quite simple.
- We start at the root and recursively go down the tree searching for a location of **ITEM** in **T**, or inserts **ITEM** as new node in its appropriate place T.
- If the element **ITEM** to be inserted is already in the tree, we are done (we do not insert duplicates)



before insertion after insertion

Searching and Inserting in BST (Algorithm)

SearchingAndInsertingAlgorithm (T, ITEM, N)

(a) Compare ITEM with the root node N of T

i. If ITEM<N, proceed to the left child of N

ii. If ITEM>N, proceed to the right child of N

(b) Repeat step (a) until one of the following occurs:

i. We meet a node N such that ITEM=N. In this case the search is Successful.

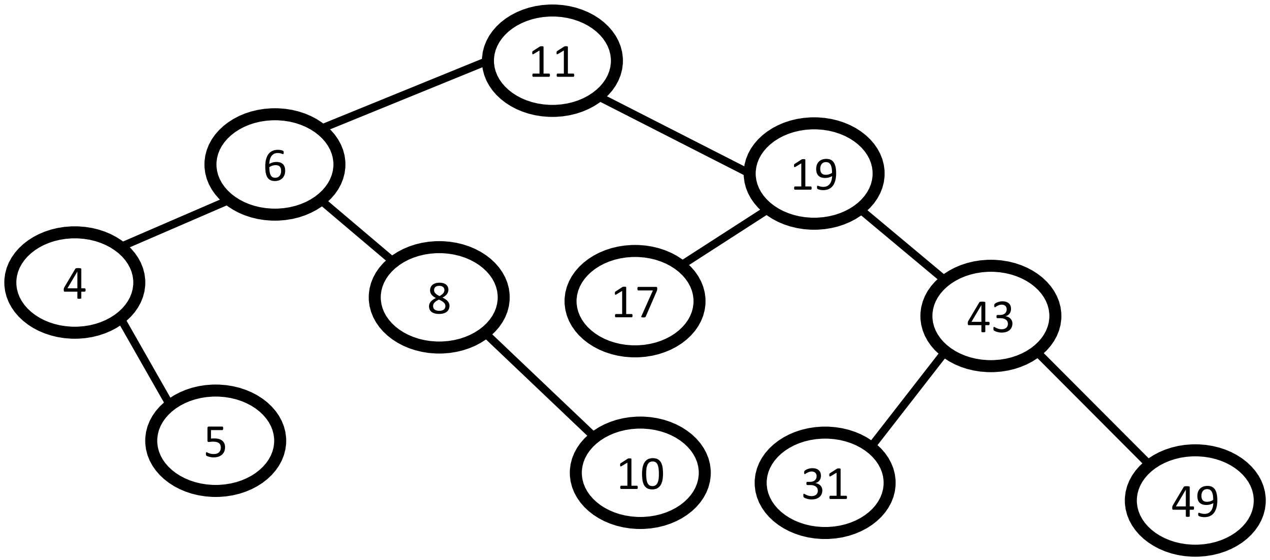
ii. We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

Searching and Inserting in Binary Search Trees

Exercise. Given a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

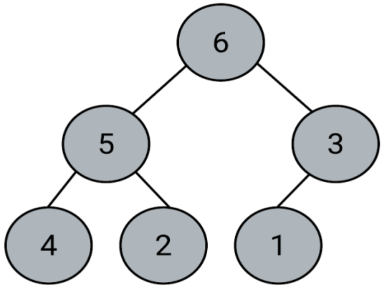
Draw a binary search tree by inserting the above numbers from left to right.



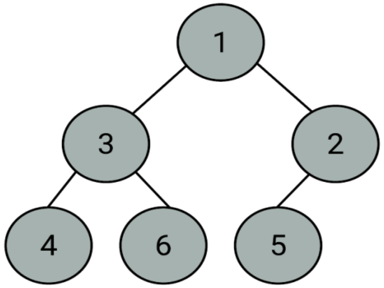
Look at Example 7.14, 7.15

HEAP; MAXHEAP, MINHEAP

- Suppose **H** is a complete binary tree with n elements.
- **H** is called a **HEAP** or a **MAXHEAP**, if each node of **N** of **H** has the following property:
  - The value at **N** is greater than or equal to the value at each of the children of **N**,
  - Accordingly, the value at **N** is greater than or equal to the value at any of the children of **N**.
- **MINHEAP**: The value of N less than or equal to the value at any of the children of N.



Max Heap

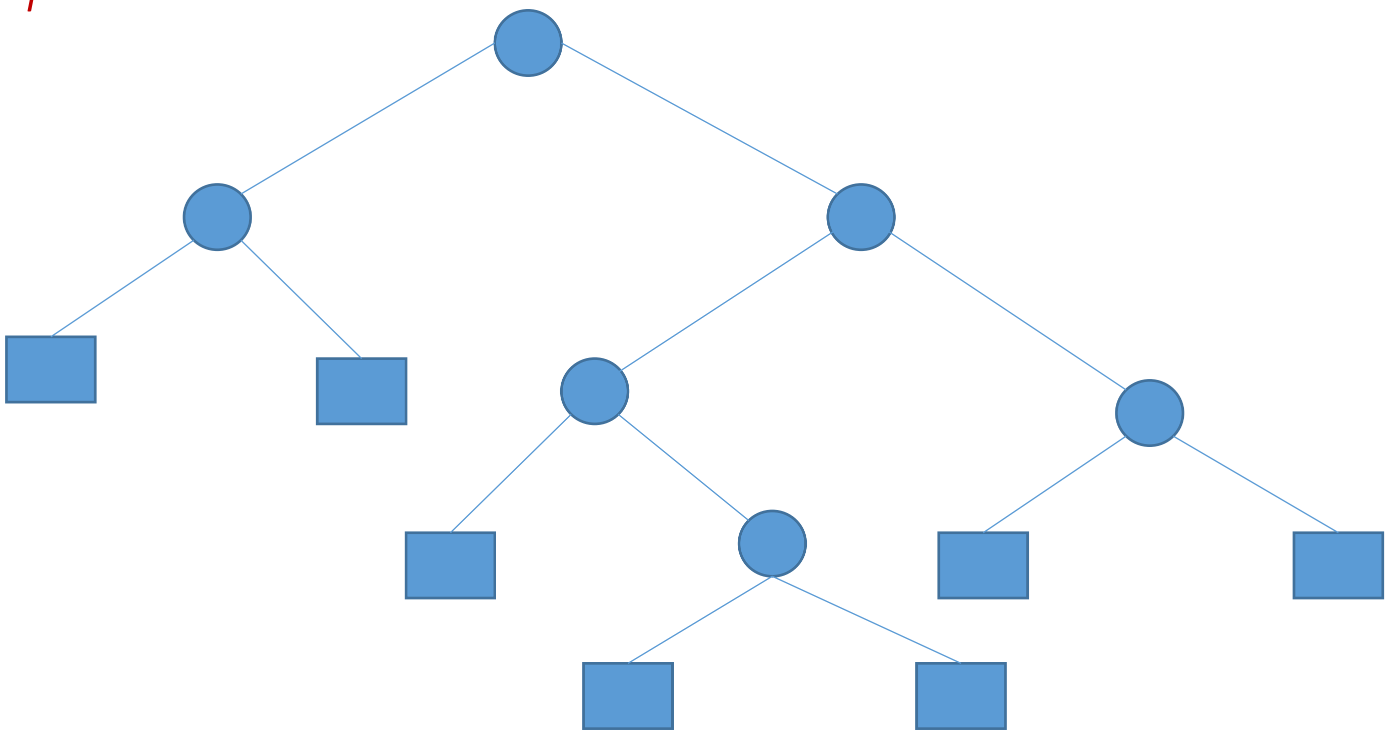


Min heap

Path Lengths: Huffman’s Algorithm

Extended Binary Tree

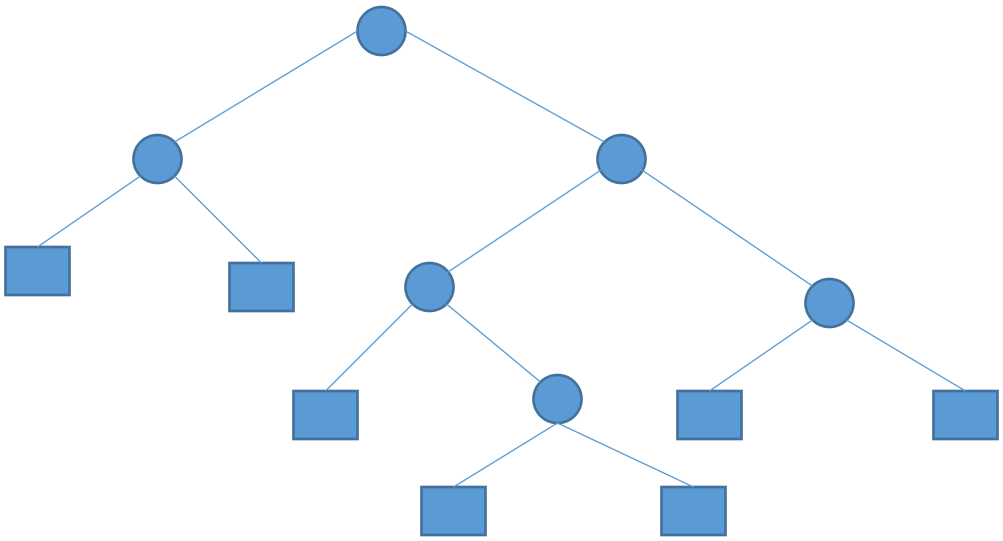
- Extended binary tree or 2-tree is binary tree with either 0 or 2 children
- Leaves are external nodes and others are internal nodes
- $N_E = N_I + 1$



Path Length

- External Path Length L_E: Sum of all path lengths summed over each path from the root *R* to an external node.
- Internal Path Length L_I: Definition is same
- $L_E = 2+2+3+4+4+3+3=21$
- $L_I = 0+1+1+2+3+2=9$
- $L_E = L_I + 2*n$

n -> is the number of internal nodes



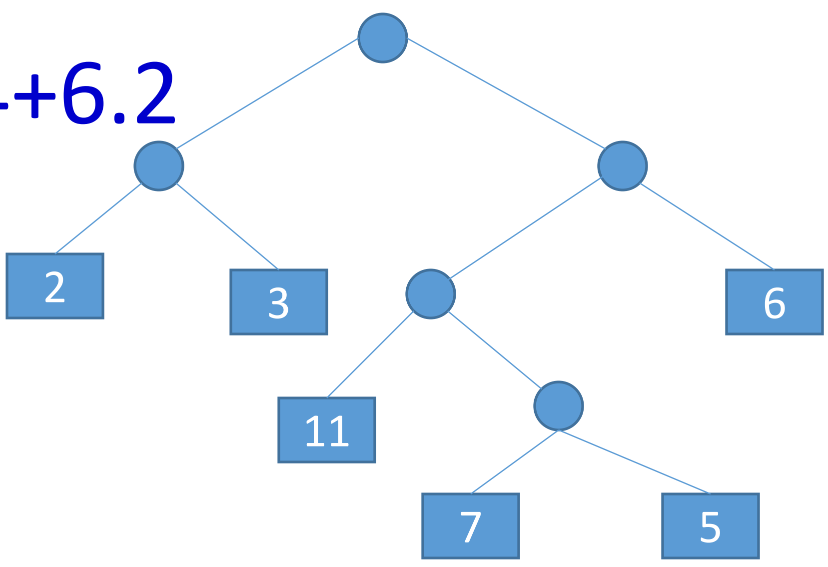
Weighted Path Length

- Suppose each external node is assigned a weight w. The weighted path length is defined to be the sum of the weighted path length:

$$P = w_1L_1 + w_2L_2 + \ldots\ldots + w_nL_n$$

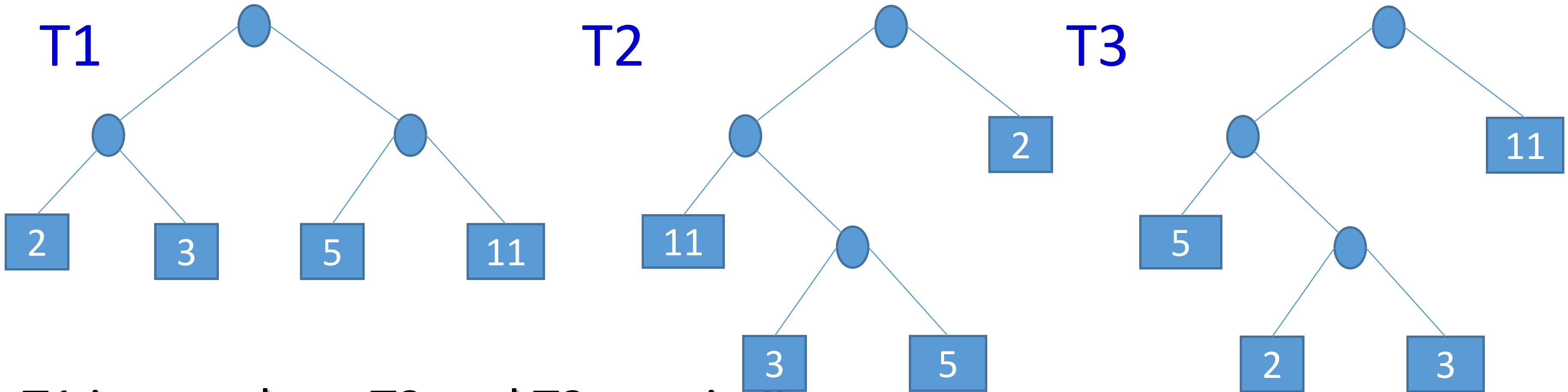
- P = 2.2+3.2+11.3+7.4+5.4+6.2

  = **103**



Minimum Path Length

- Weighted Path Length



- T1 is complete; T2 and T3 are similar
- P1 = 2.2+3.2+5.2+11.2 = 42
- P2 = 2.1+3.3+5.3+11.2 = 48
- P3 = 2.3+3.3+5.2+11.1 = 36

- The quantities P1 and P3 indicate that the complete tree need not give a minimum length P.
- The quantities P2 and P3 indicates that similar trees need not give the same length.

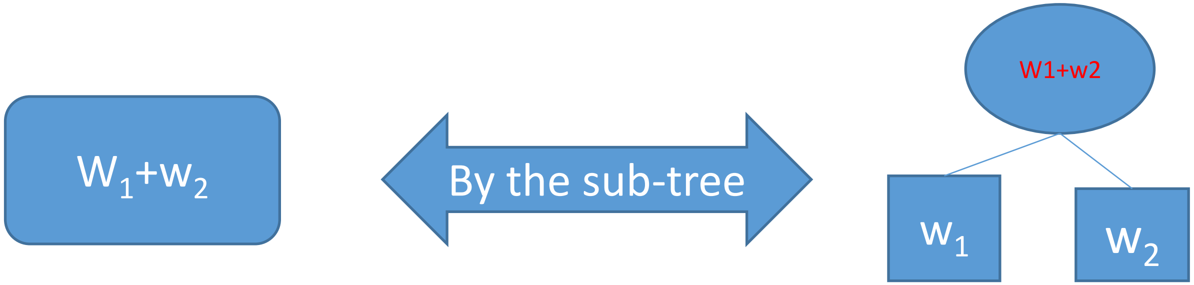
General Problem of finding minimum-weighted path length

- For a list of n weights $w_1, w_2, \ldots\ldots, w_n$, we want to find a minimum weighted path length tree. Here, w_i represents weights of external nodes.
- The tree may not be unique
- Huffman gave an algorithm to find such a tree

Huffman's Algorithm

ICE 2231

- **Algorithm: Suppose w_1 and w_2 are two minimum weights among $w_1, w_2, \ldots\ldots, w_n$**
  - **Find a tree T' which gives a solution for n-1 weights $w_1 + w_2, w_3, \ldots\ldots, w_n$**
  - **Then in the tree T' replace the external node as:**



  - **The new 2-tree is the desired solution**
  - **Apply this algorithm recursively to find the final tree**

Illustration of Huffman's Algorithm

ICE 2231

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

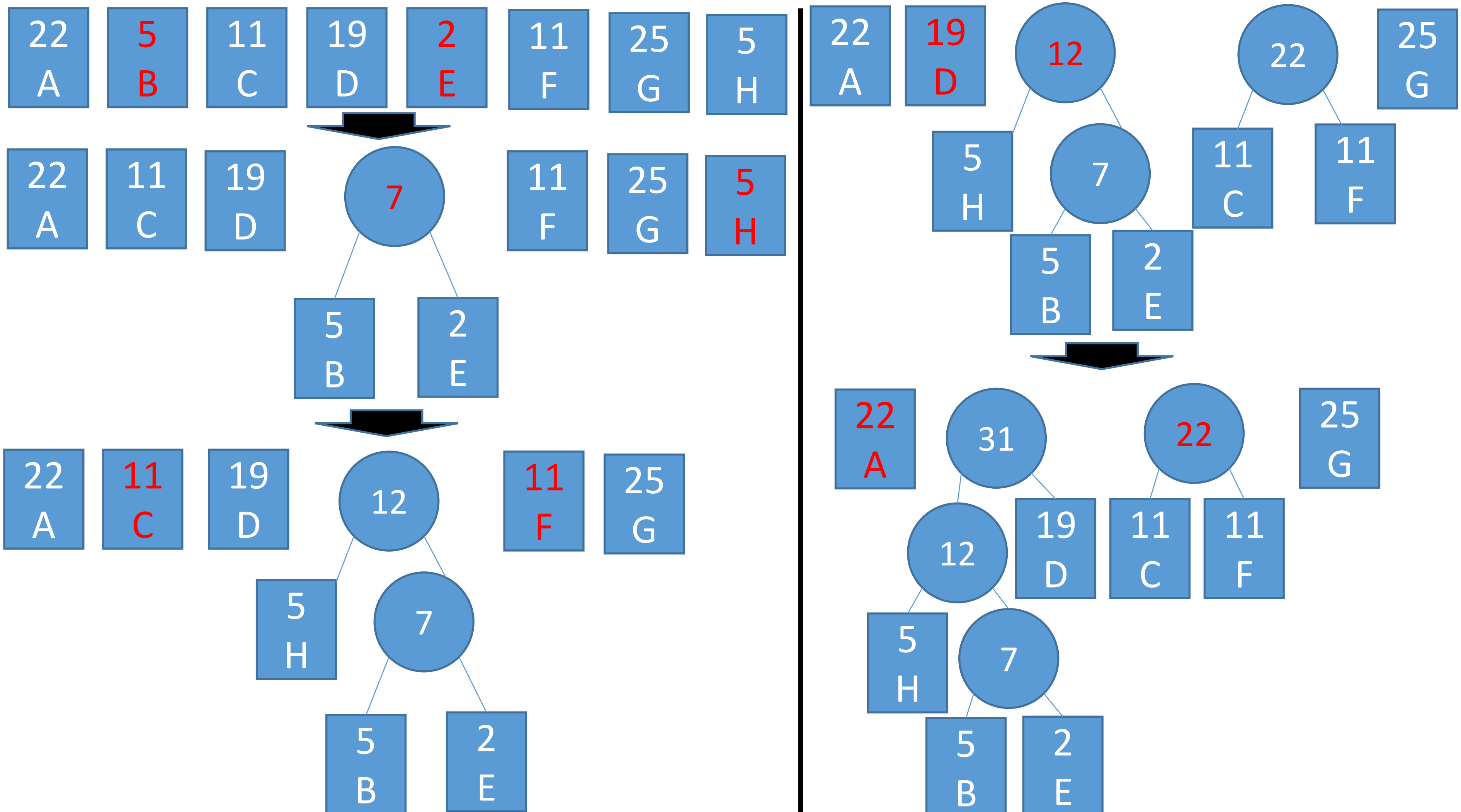
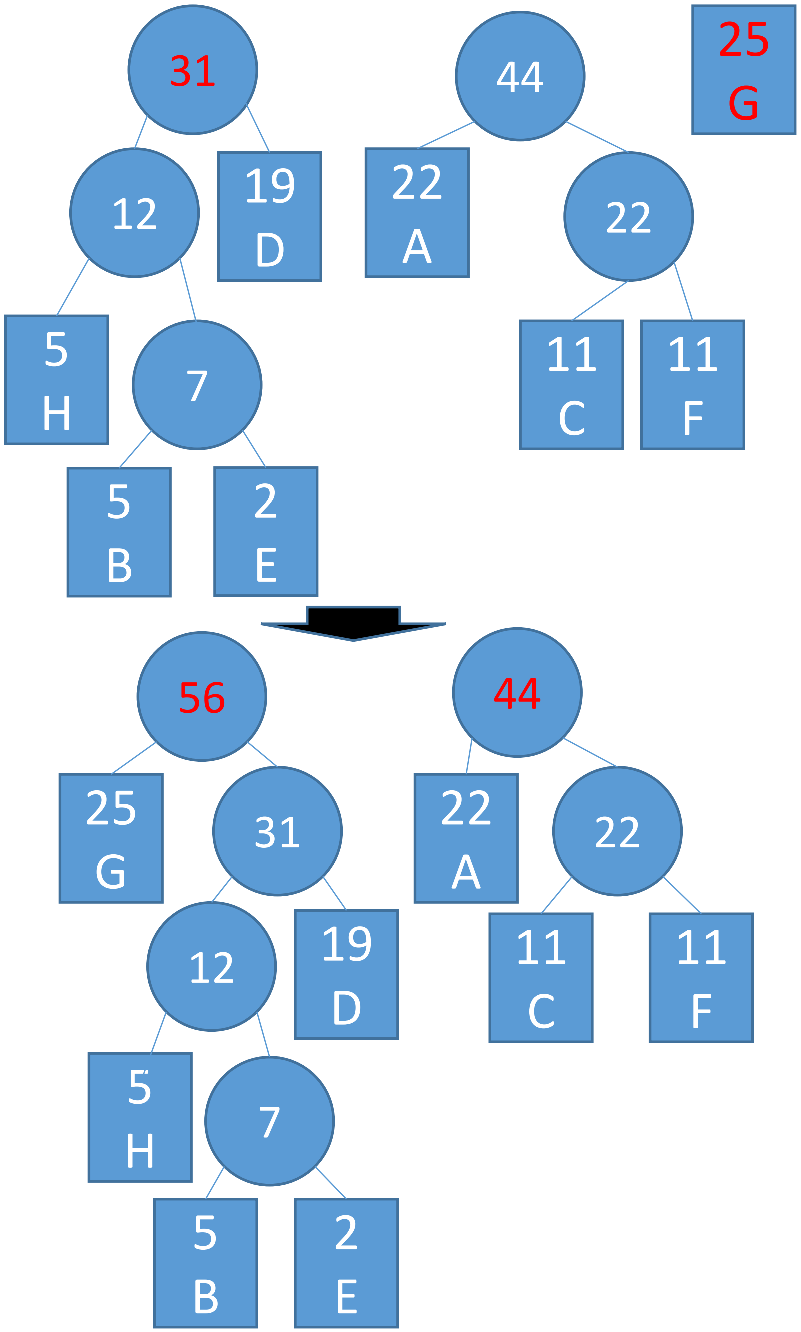
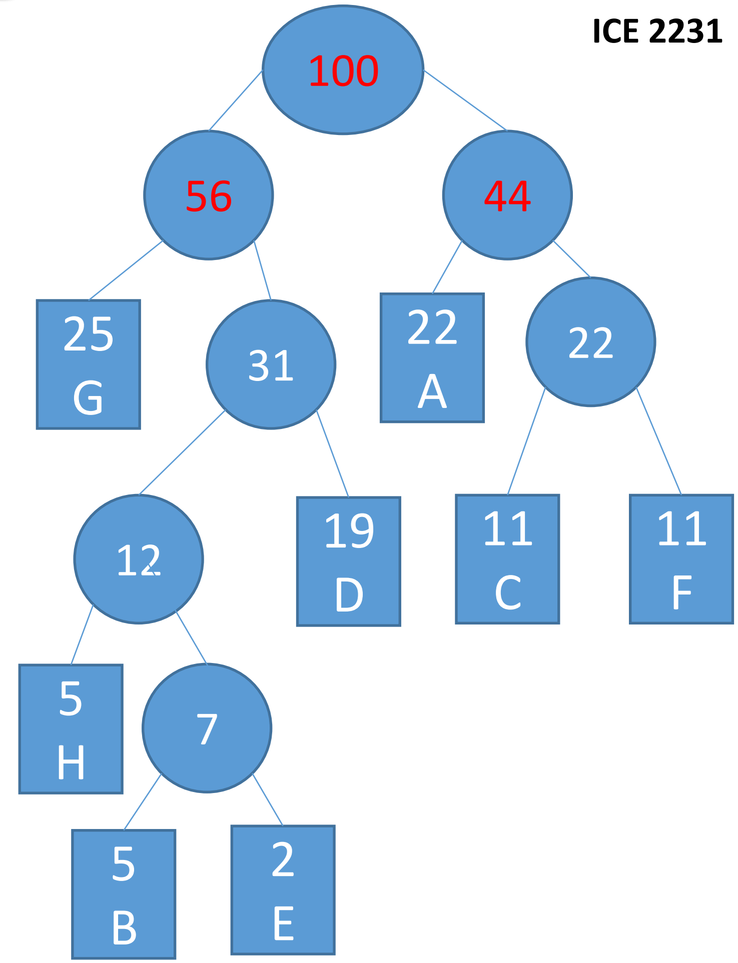


Illustration of Huffman's Algorithm

ICE 2231





Huffman Coding

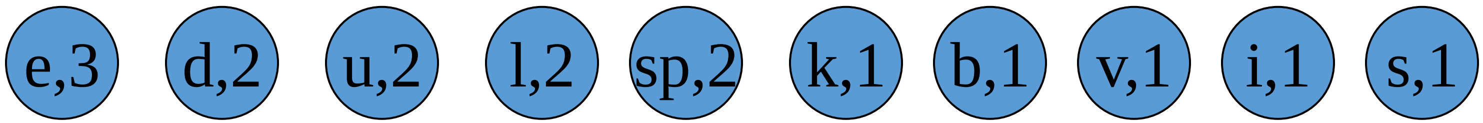
- Huffman codes can be used to compress information
  - Like WinZip – although WinZip doesn't use the Huffman algorithm
  - JPEGs do use Huffman as part of their compression process
- The basic idea is that instead of storing each character in a file as an 8-bit ASCII value, we will instead store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits
  - On average this should decrease the filesize (usually ½)

Huffman Coding

- As an example, lets take the string:

  "duke blue devils"

- We first to a frequency count of the characters:
  - e:3, d:2, u:2, l:2, space:2, k:1, b:1, v:1, i:1, s:1
- Next we use a Greedy algorithm to build up a Huffman Tree
  - We start with nodes for each character

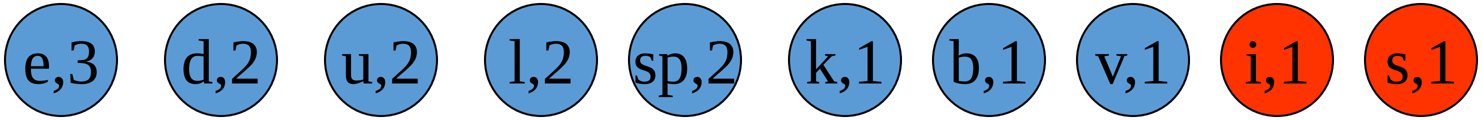


Huffman Coding

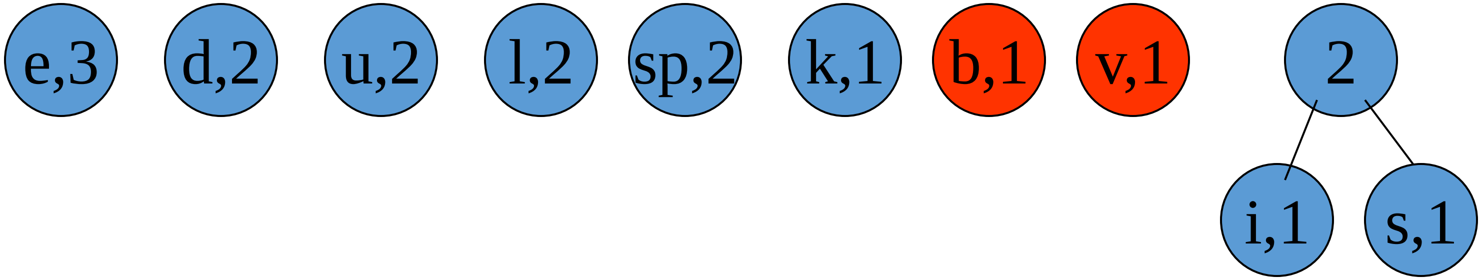
- We then pick the nodes with the smallest frequency and combine them together to form a new node
  - The selection of these nodes is the Greedy part
- The two selected nodes are removed from the set, but replace by the combined node
- This continues until we have only 1 node left in the set

Huffman Coding

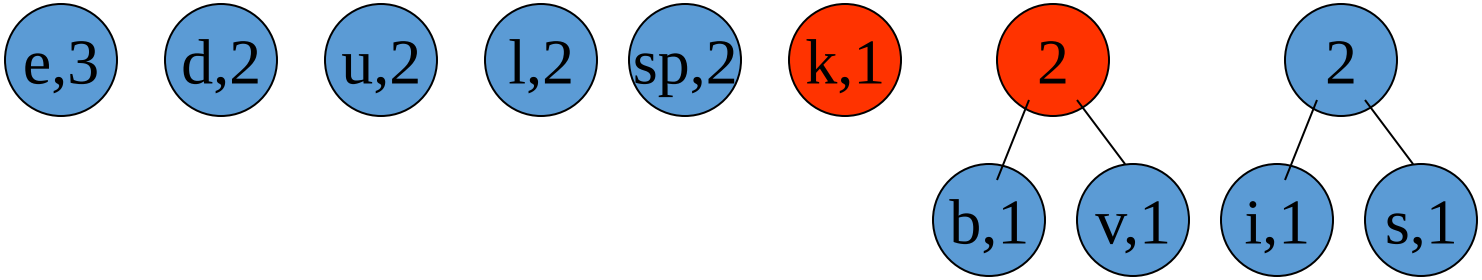
ICE 2231



Huffman Coding

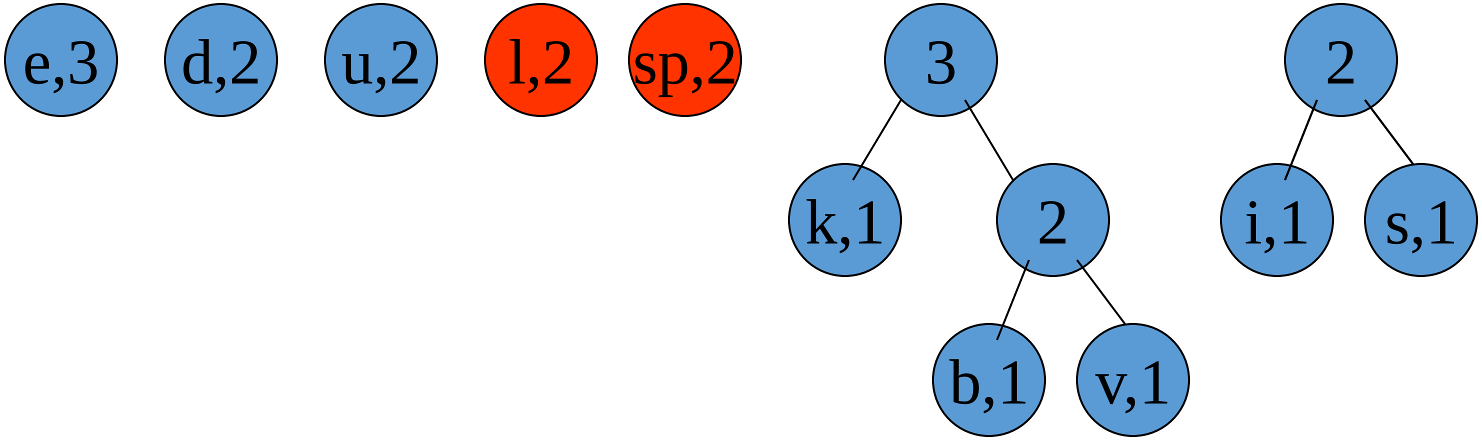


Huffman Coding

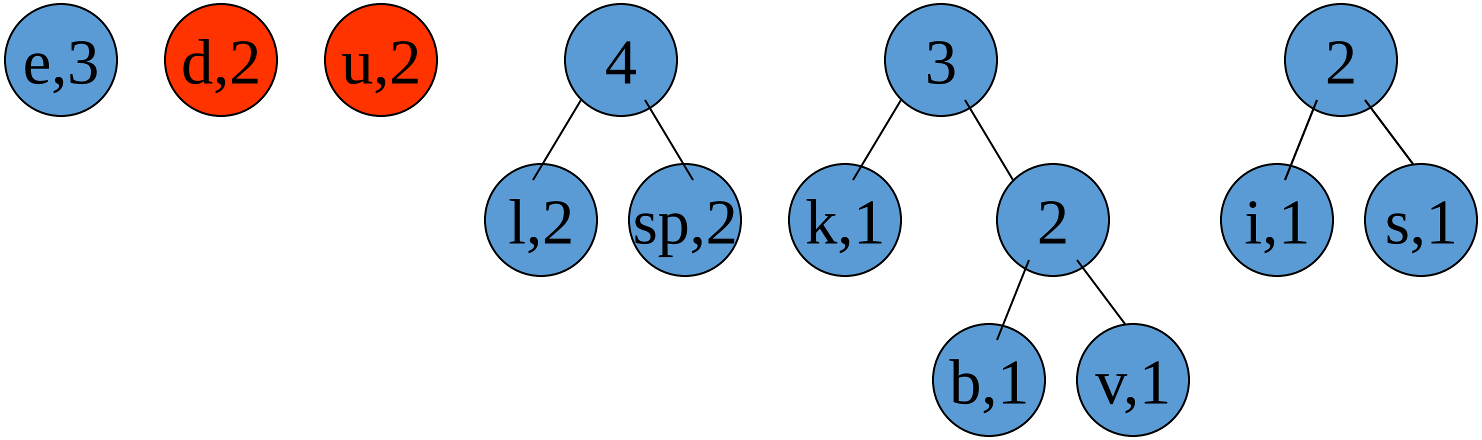


Huffman Coding

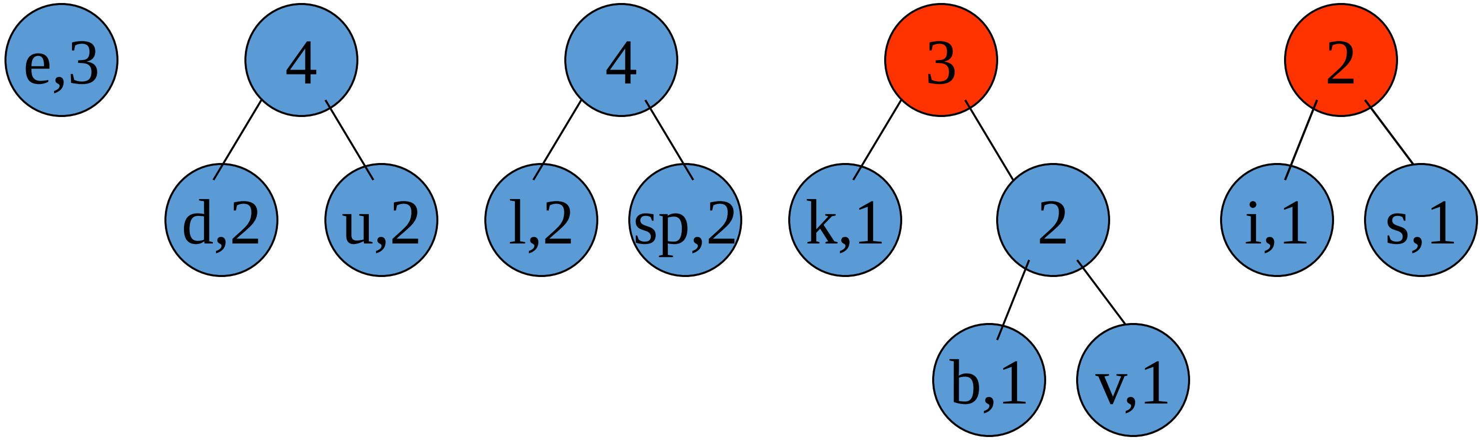
ICE 2231



Huffman Coding

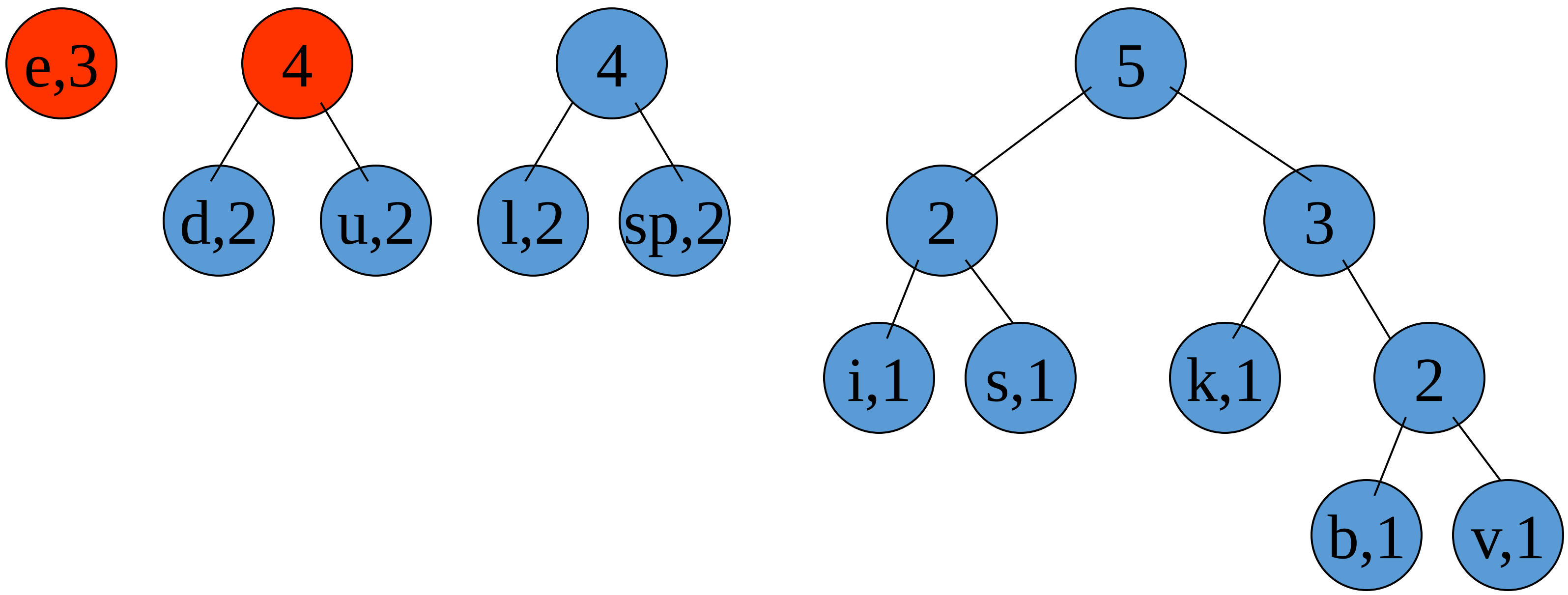


Huffman Coding



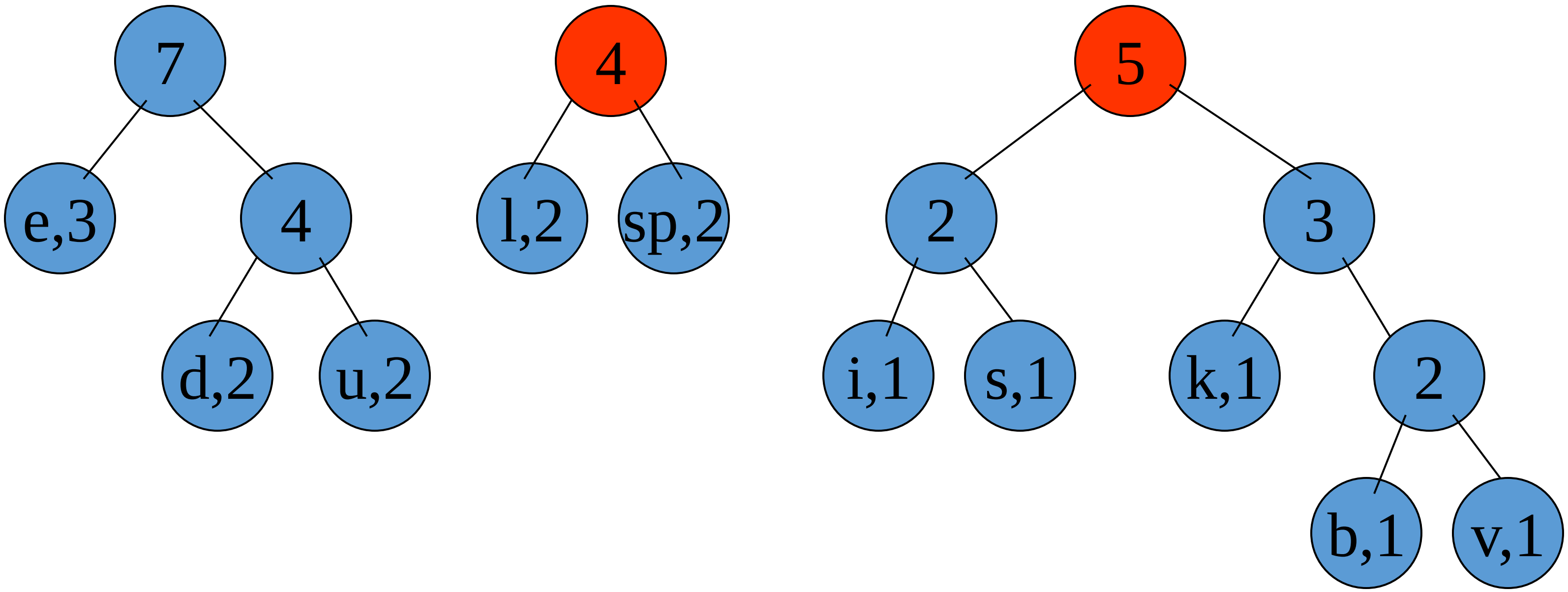
Huffman Coding

ICE 2231



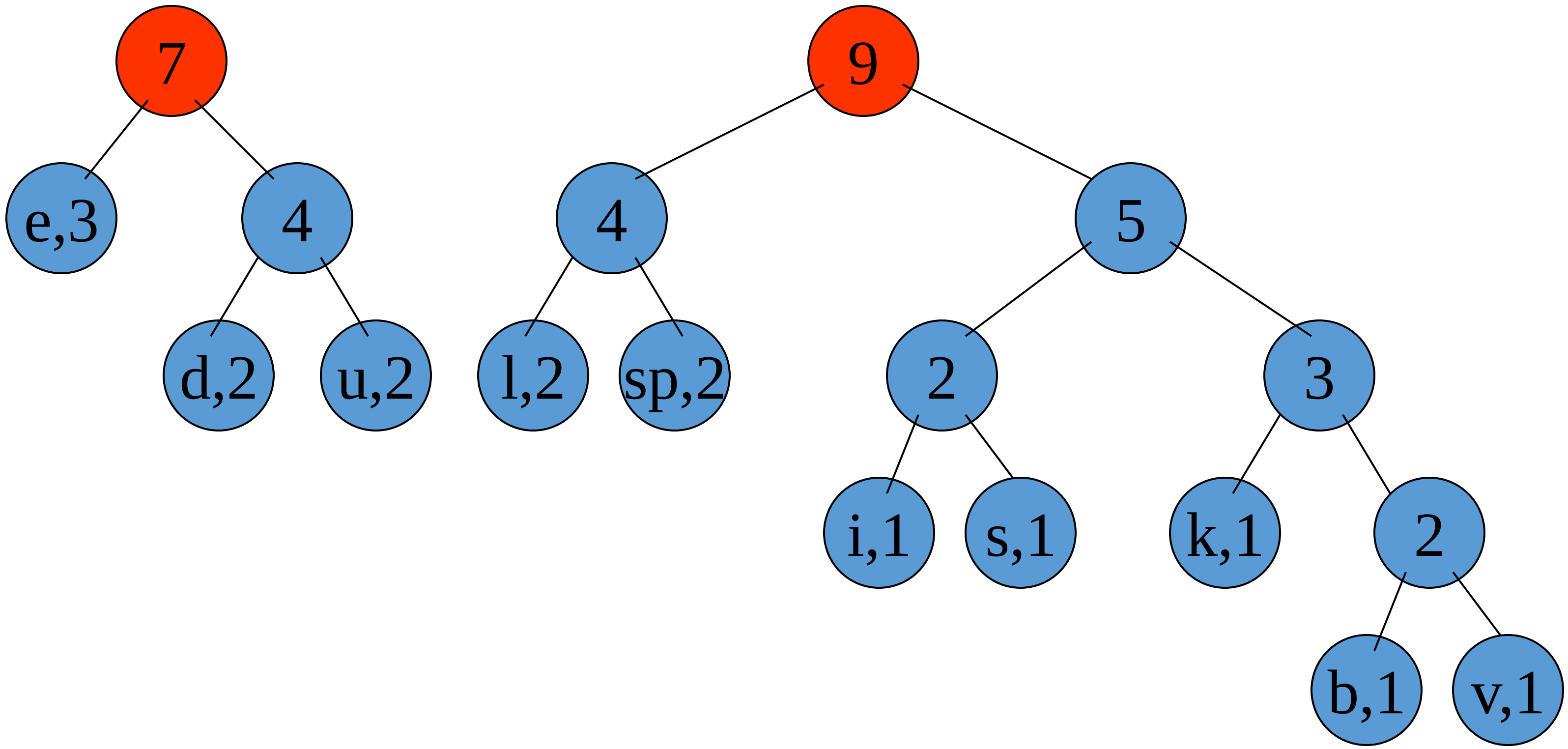
Huffman Coding

ICE 2231

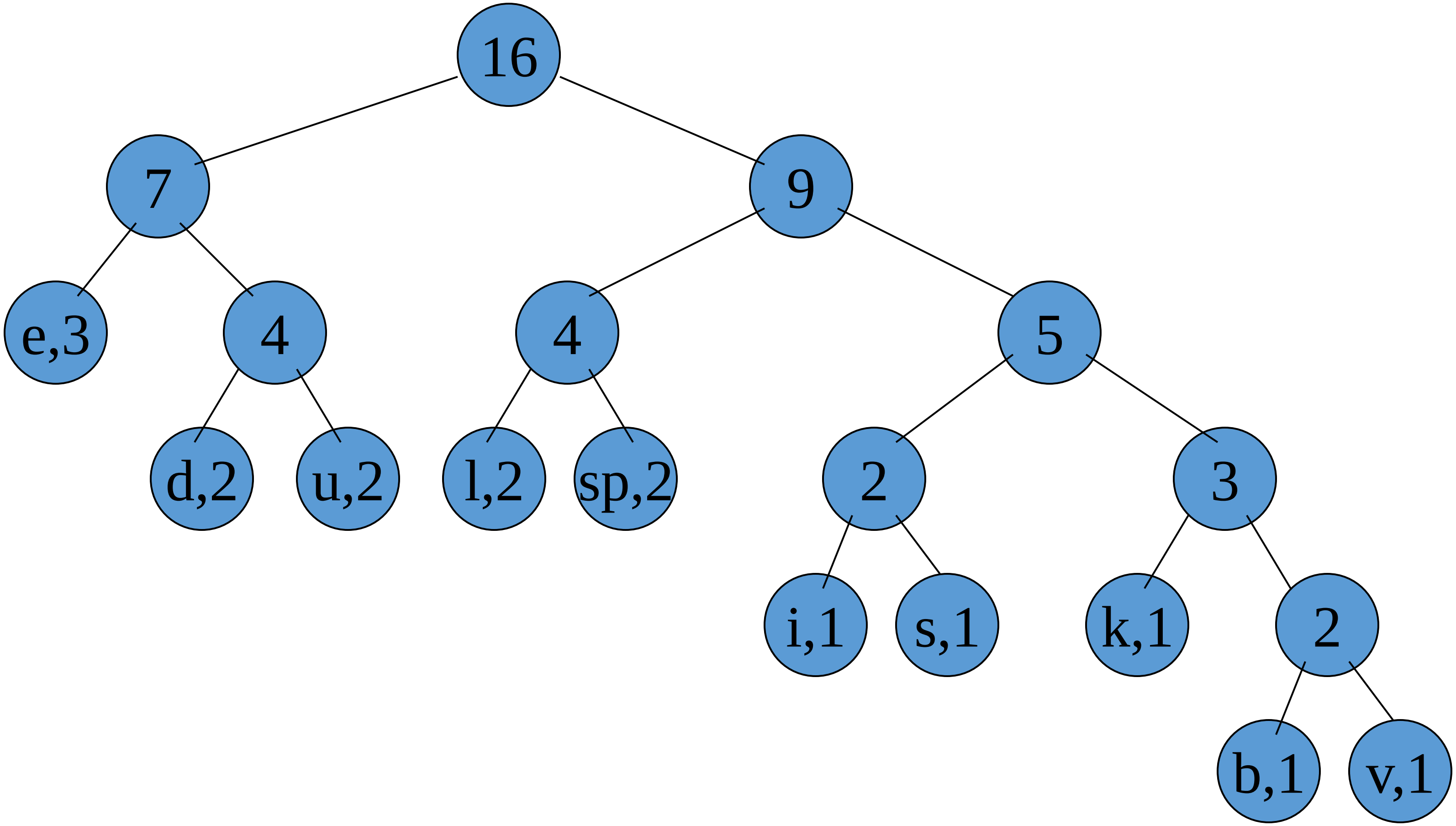


Huffman Coding

ICE 2231



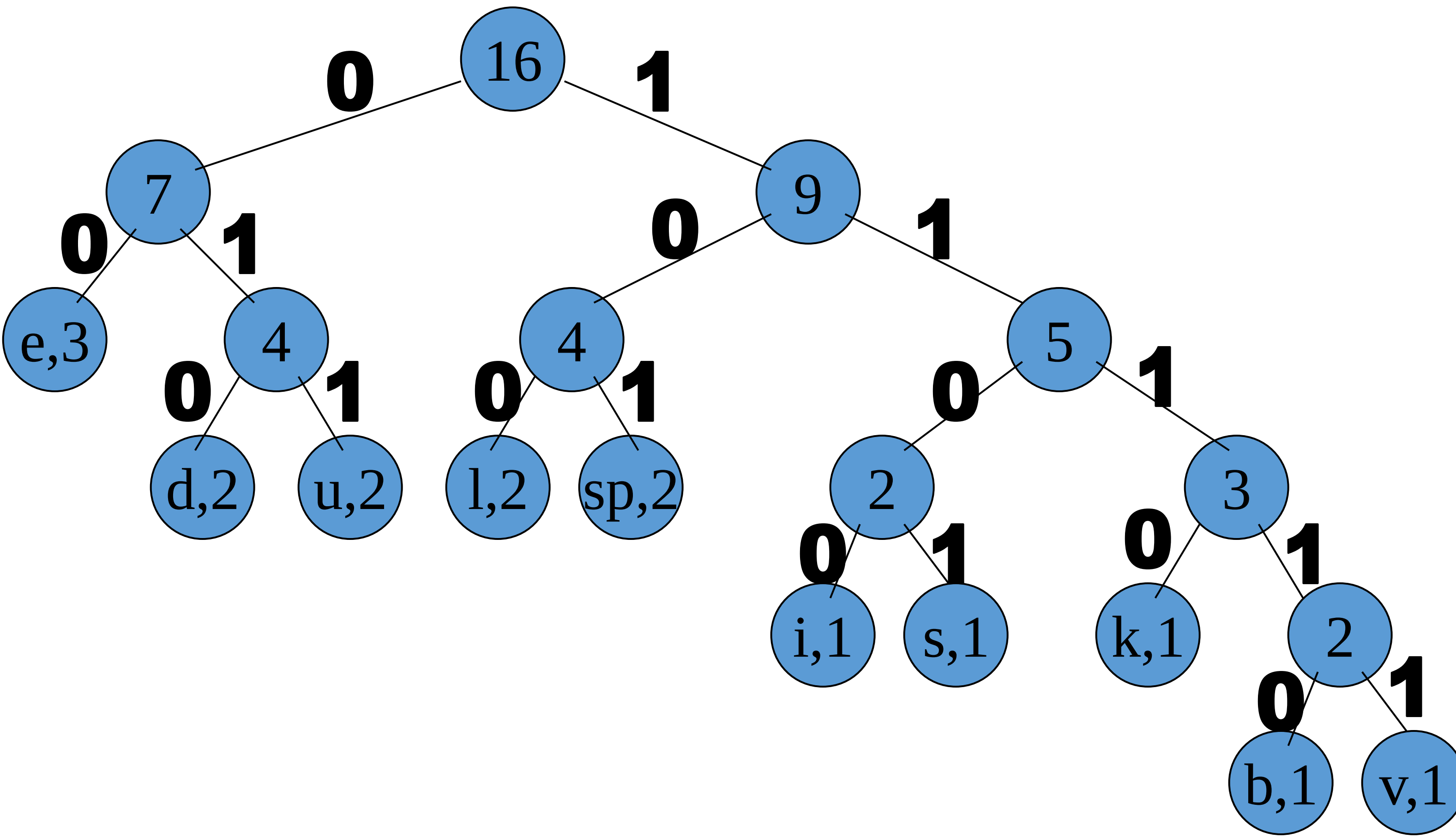
Huffman Coding



Huffman Coding

- Now we assign codes to the tree by placing a 0 on every left branch and a 1 on every right branch
- A traversal of the tree from root to leaf give the Huffman code for that particular leaf character
- Note that no code is the prefix of another code.

Huffman Coding



e	00
d	010
u	011
l	10
sp	101
i	1100
s	1101
k	1110
b	11110
v	11111

Huffman Coding

- These codes are then used to encode the string
- Thus, “duke blue devils” turns into:
- **010 011 1110 00 101 11110 100 011 00 101 010 00 11111 1100 100 1101**
- When grouped into 8-bit bytes:

01001111 10001011 11101000 11001010 10001111 11100100 1101xxxx

- Thus it takes 7 bytes of space compared to 16 characters * 1 byte/char = 16 bytes uncompressed

Huffman Coding

- Uncompressing works by reading in the 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
- Thus, we need to save Huffman table information as a header in the compressed file
  - Doesn’t 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

Solved Problems on Tree:

7.1 , 7.2, 7.3, 7.7, 7.8, 7.14