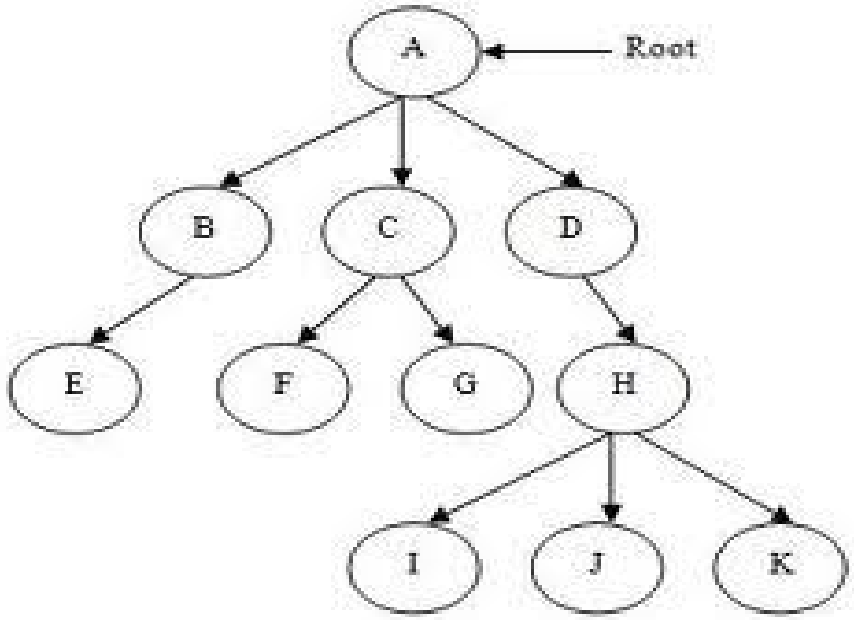
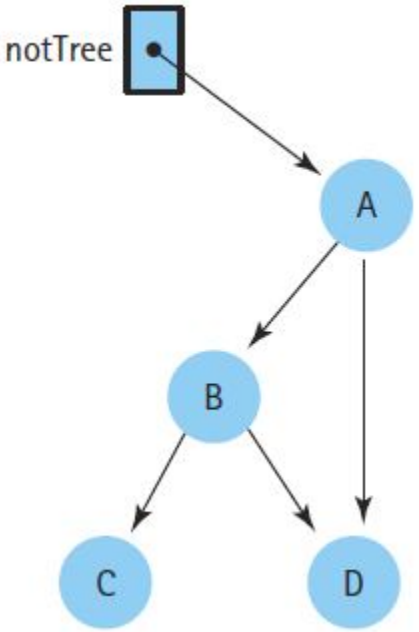
# Tree

# Overview

- A **tree** is a nonlinear **data structure**
- A **tree** can be empty with no nodes or a **tree** is a **structure** consisting of one node called the root and zero or one or more subtrees.
- ***A Tree is a graph which does not contain any cycle, so there is exactly one path from the root to each node in a tree.***

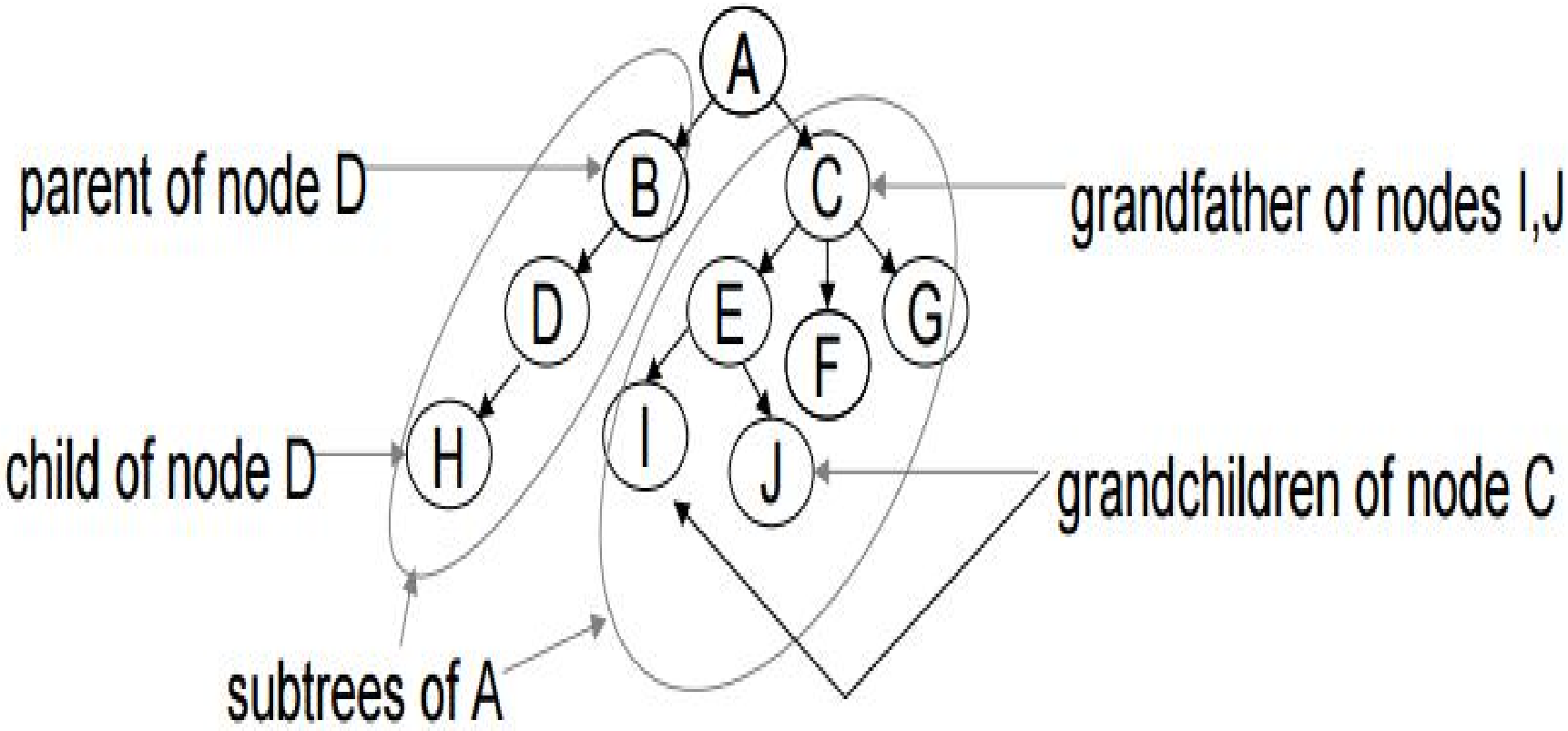


# Terminology

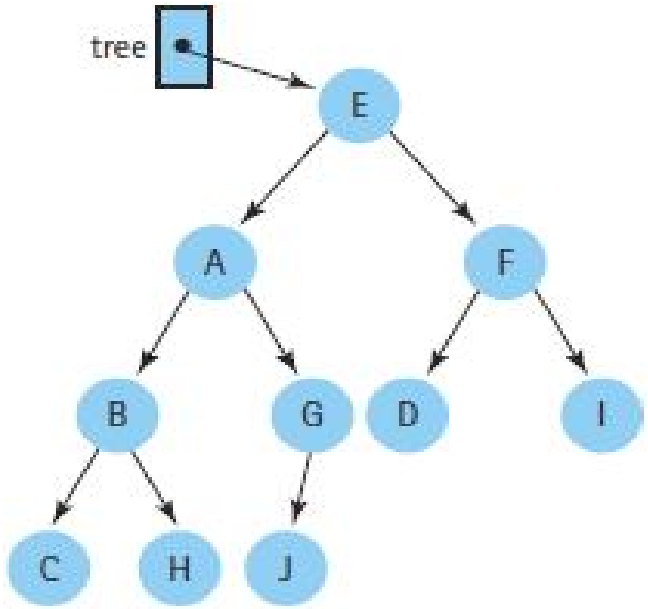
- **Nodes** are vertices and **Edges** are connections between nodes
- **Root** is top most node or base of tree where from tree starts
- Parent, Child, Siblings (children of same parent), Grand parent, Grand child
- **Ancestor** – a node A is said to be ancestor of B if A is either parent of B or parent of some ancestor of B
- **Descendent** – a node B is said to be descendent of A if B is either child of A or child of some other descendent of A
- **Every node except the root has one parent**
- **Sub-tree** – Part of tree
- **Leaf node** – node which does not have any child

# Example


A
B
C
D
E
F
G
H
I
J
parent of node D
child of node D
grandfather of nodes I,J
grandchildren of node C
subtrees of A

# Terminology(Cont…)

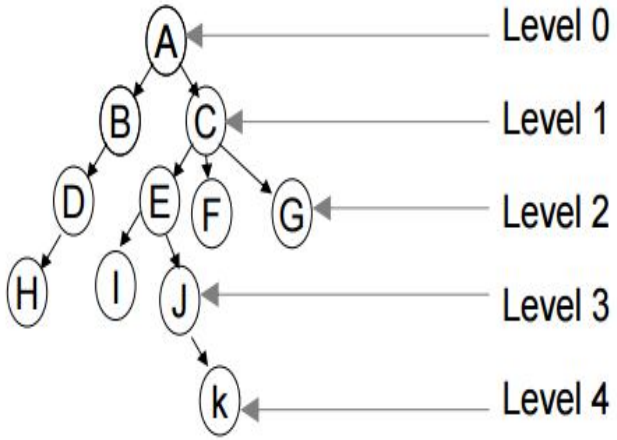
- **Degree** of a node is number of nodes connected to it
- In a directed tree :
  - **In-degree :** Number of edges coming toward to particular node
  - **Out-degree : N**umber of edges going out from particular node
- **Internal node** – out degree>0
- **Size:** The number of nodes in a tree.



# Height and Depth

- **Height** of a node is the length of a longest path from this node to a leaf
- All leaves are at height zero
- Height of a tree is the height of its root (maximum level)
- **Dept*h*** of a node is the length of path from root to this node
- Root is at depth zero
- **Depth of a tree is the depth of its deepest leaf that is equal to the height of this tree**

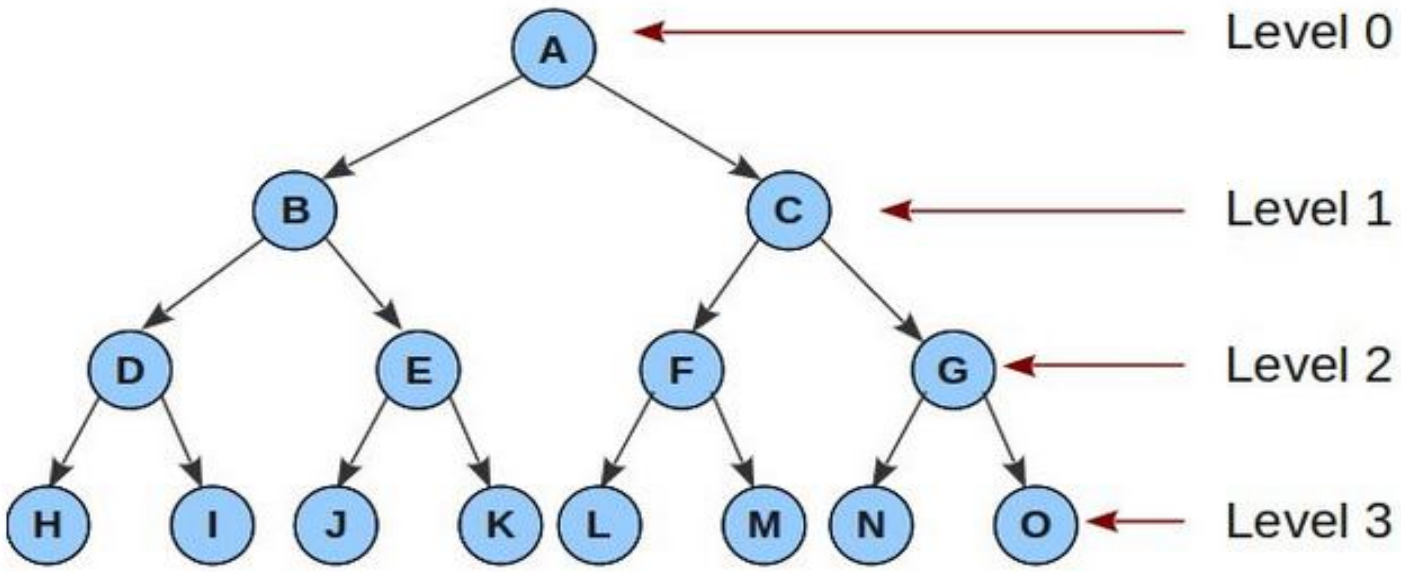
- The height of the tree is 4.
- The height of node C is 3.



# Binary Tree

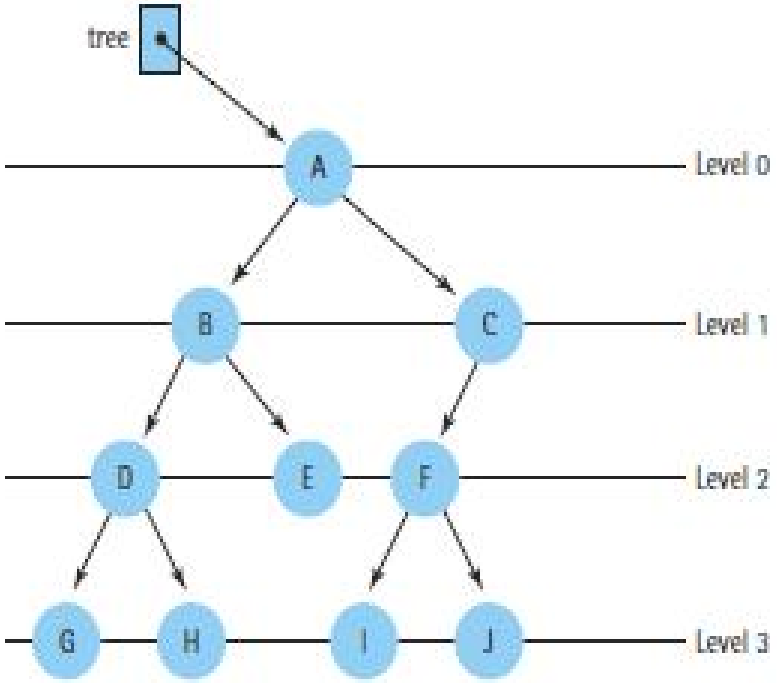
- Tree with no node of out degree greater than two
- Out degree of any node can be 0,1 or 2
- Any node can have at most two children

Example of level and depth of a binary tree:

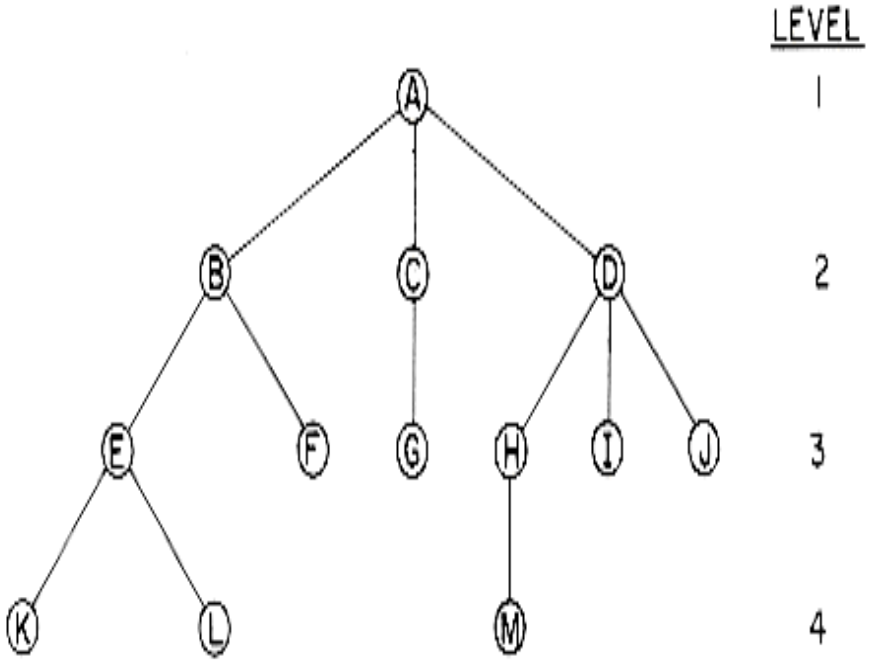



Depth fo the above tree (d) = 3

# Binary Tree (Cont…)



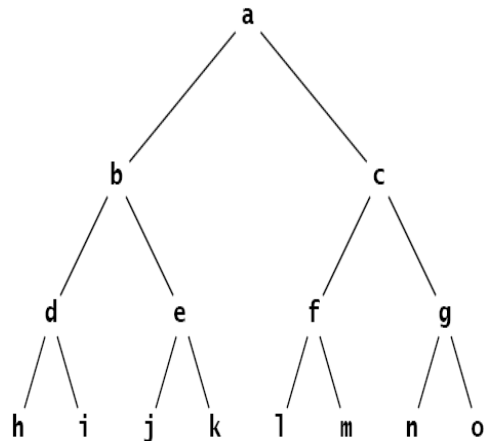

**Binary tree**




**Not a binary tree**

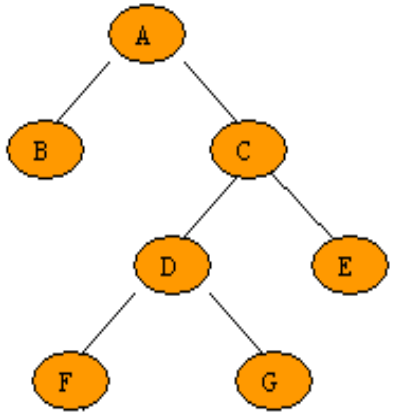
# Full Binary Tree

➢A full binary tree is a binary tree in which every node other than the leaves has two children. This is also called ***strictly binary tree*** or **proper binary tree** or **2-tree**.



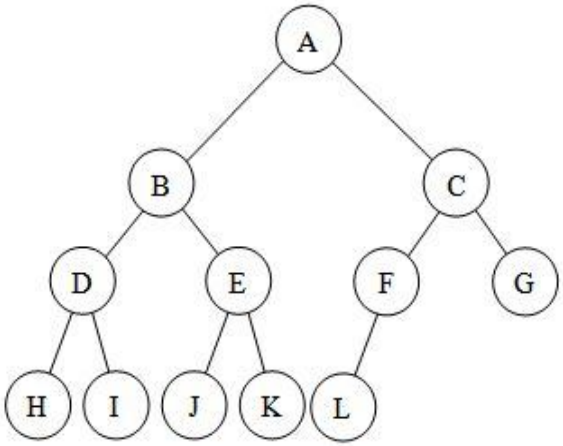

Is this Full Binary tree

YES




Is this Full Binary tree

YES




Is this Full Binary tree

NO

# Some Important Facts

- **Let T be a nonempty, full binary tree Then:**

  (a) If T has I internal nodes, the number of leaves is L = I + 1.

  (b) If T has I internal nodes, the total number of nodes is N = 2I + 1.

  (c) If T has a total of N nodes, the number of internal nodes is I = (N – 1)/2.

  (d) If T has a total of N nodes, the number of leaves is L = (N + 1)/2.

  (e) If T has L leaves, the total number of nodes is N = 2L – 1.

  (f) If T has L leaves, the number of internal nodes is I = L – 1.

  Basically, number of nodes N, the number of leaves L, and the number of internal nodes I are related in such a way that if you know any one of them, you can determine the other two.

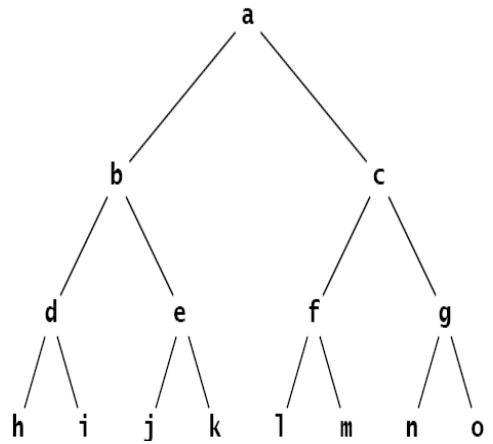
# Some Important Facts

1.Maximum number of leaves in the full binary tree of height h = $2^h$ and $2^h - 1$ internal nodes

2. Total no. of nodes =Total number of leaves + Total number of internal nodes =$2^{h+1} - 1$

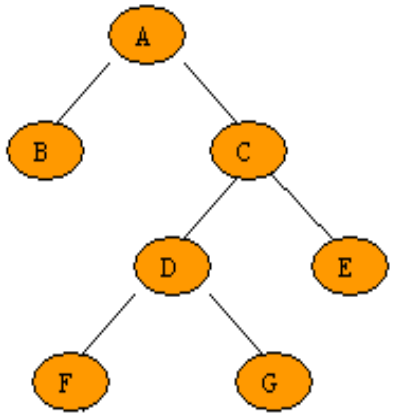
# Complete Binary Tree

➢A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



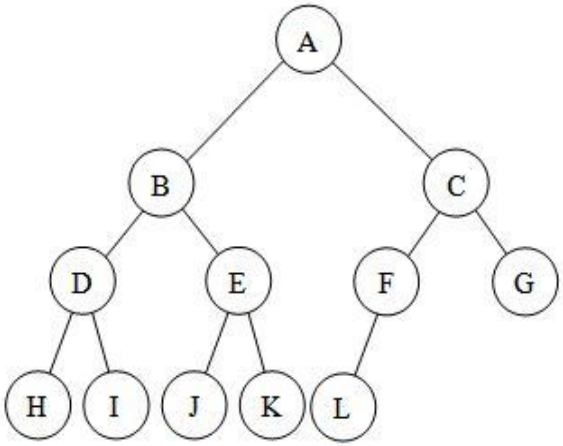

Is this complete Binary tree

YES




Is this complete Binary tree

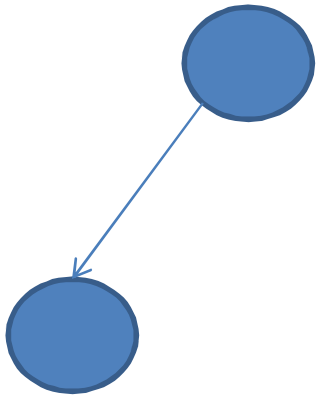
NO




Is this complete Binary tree

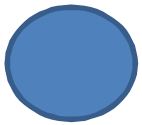
YES

# Complete Binary Tree



Is this complete Binary tree

YES



Is this complete Binary tree

YES

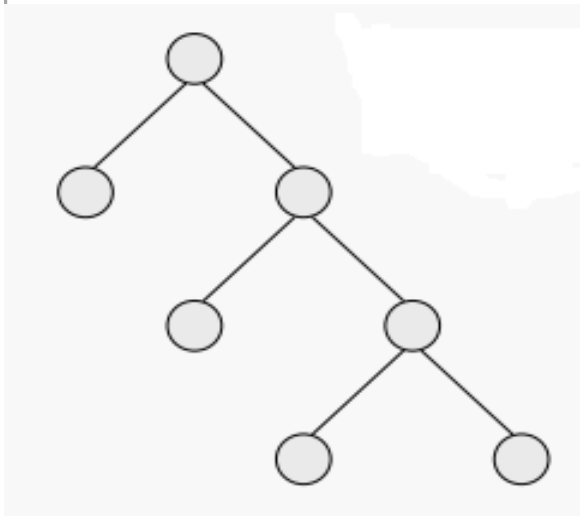
But this binary tree is complete.
It has only one node, the root.

Is this complete Binary tree

YES

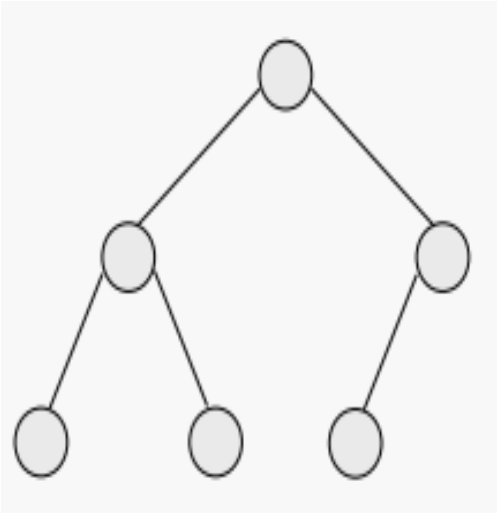
It is called the empty tree, and it has no nodes, not even a root.

# More Examples



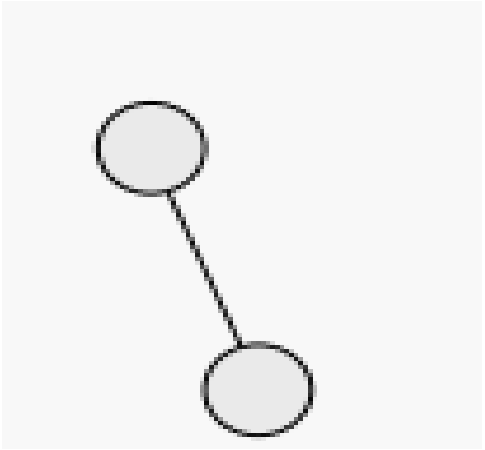
Is this Full or complete

Full but Not Complete



Is this Full or complete

Complete but not full



Is this Full or complete

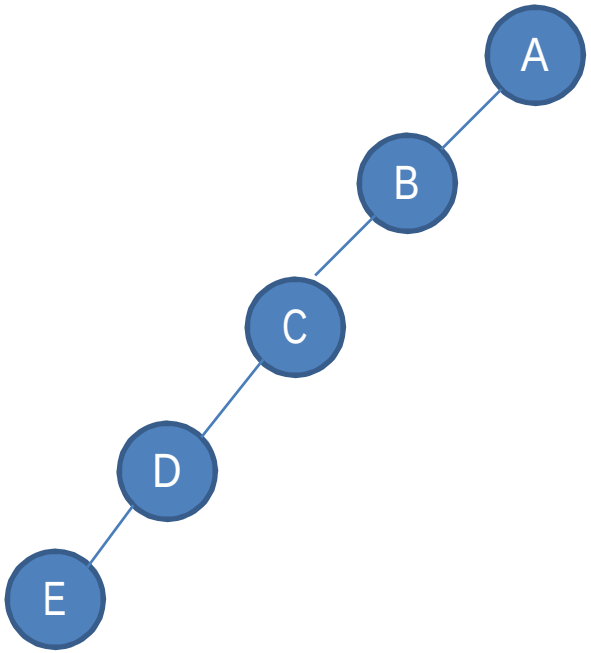
Neither Full nor complete

**Perfect binary tree** :- Every node except the leaf nodes have two children and every level (last level too) is completely filled.
i.e. a **binary tree** where **each level** contains the ***maximum*** **number of nodes**

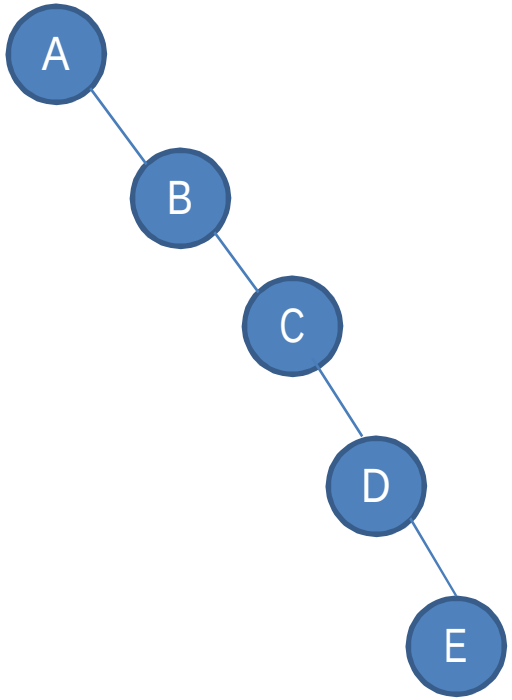
Height of **Perfect binary tree** if no. of nodes are given = $\log_2(n+1)-1$

# Skewed tree

## Left Skewed Tree




## Right Skewed Tree




Height = N – 1

# Representation of binary tree

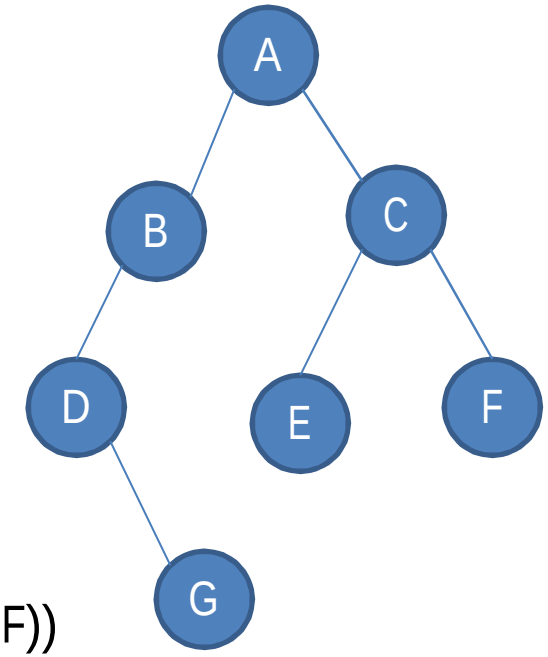
Binary tree can be represented in two ways –

1. Array
2. Linked list

***Array Representation*** –

Three arrays to be maintained.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Data | A | B | C | D | E | F | G |
| Left Child | 1 | 3 | 4 | -1 | -1 | -1 | -1 |
| Right Child | 2 | -1 | 5 | 6 | -1 | -1 | -1 |

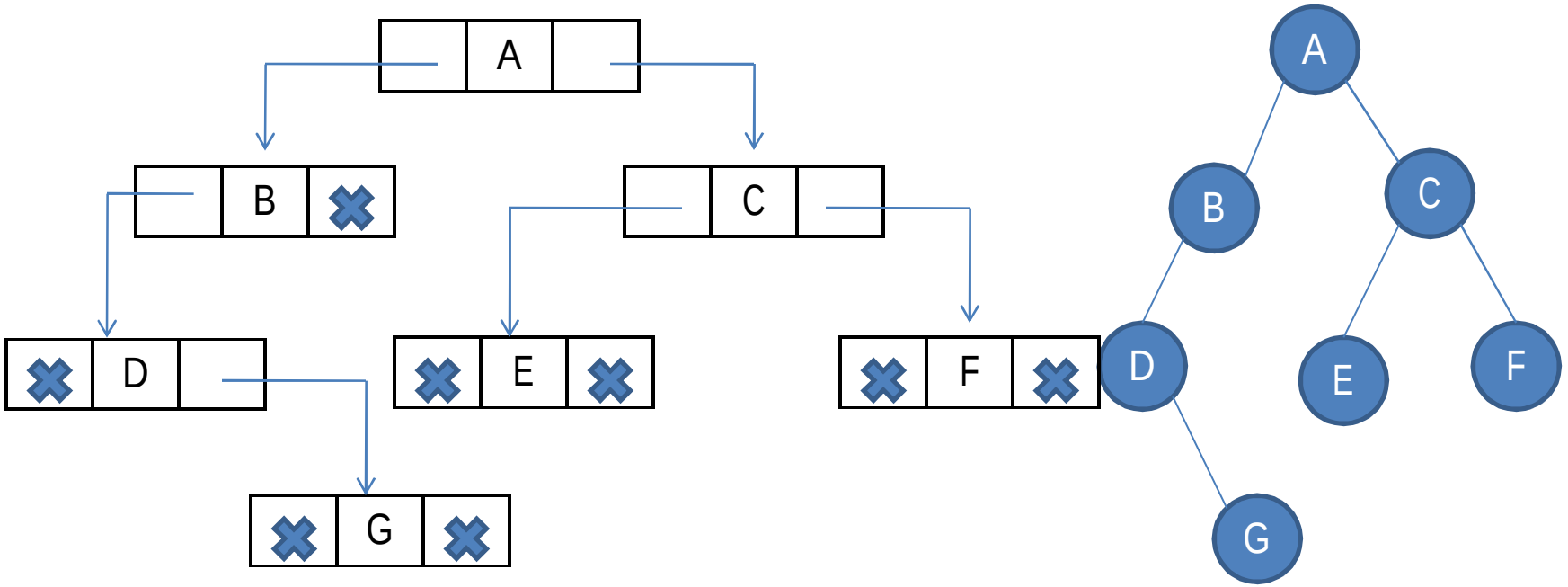



Note: Parenthetical representation A(B(D( null G) null )C(EF))

# Representation of binary tree (Cont…)

***Linked List Representation –***

```
struct Node{
        struct node *lc;
        int data;
        struct node *rc;
};
```



# Operations on Tree

1. Traversing
2. Searching
3. Insertion
4. Deletion

# Traversal

- Traversal is the process of visiting every node exactly once
- Three recursive techniques for binary tree traversal
  - Pre order traversal

    1. Visit the root
    2. Traverse left subtree
    3. Traverse right subtree

  - In order traversal

    1. Traverse left subtree
    2. Visit the root
    3. Traverse right subtree

  - Post order traversal

    1. Traverse left subtree
    2. Traverse right subtree
    3. Visit the root

# PreOrder (NLR)

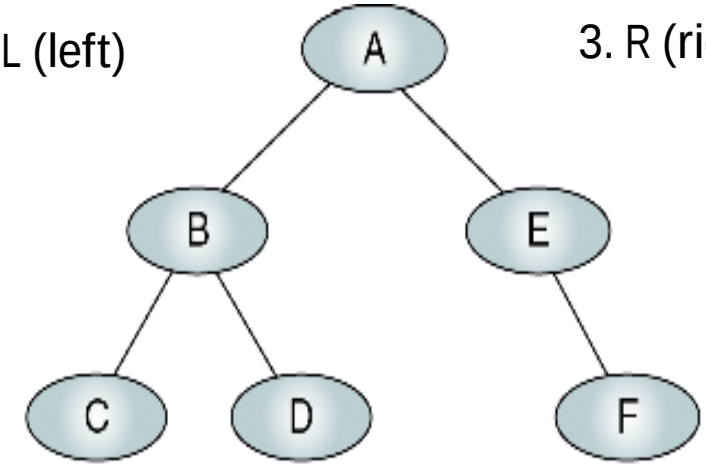
**Algorithm Preorder (root):** Traverse a binary tree in node-left-right sequence.

1. If (root is not null)
2. process (root)
3. Preorder (leftSubtree)
4. Preorder (rightSubtree)

1. N (node)

2. L (left)

3. R (right)

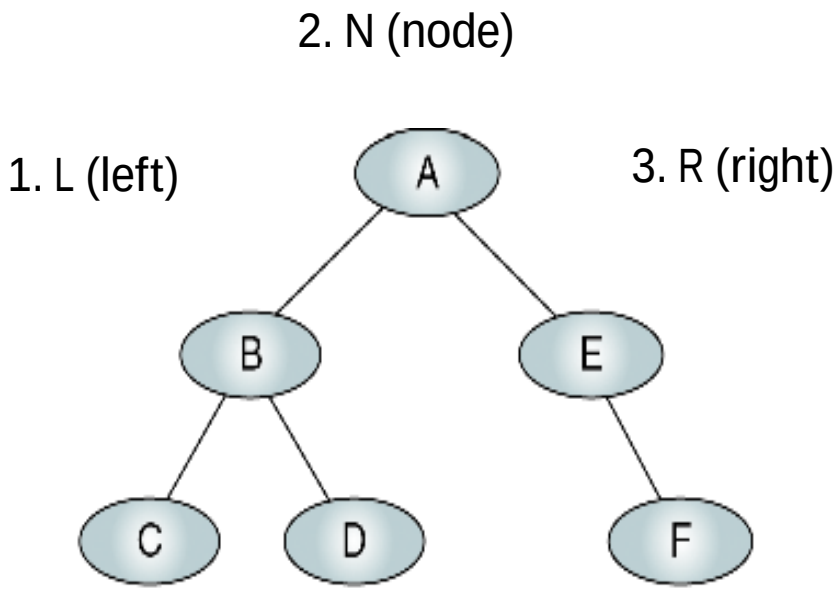



**Preorder – A B C D E F**

# InOrder (LNR)

**Algorithm Inorder (root):** Traverse a binary tree in left-node-right sequence.

1. If (root is not null)
2. Inorder (leftSubtree)
3. process (root)
4. Inorder (rightSubtree)

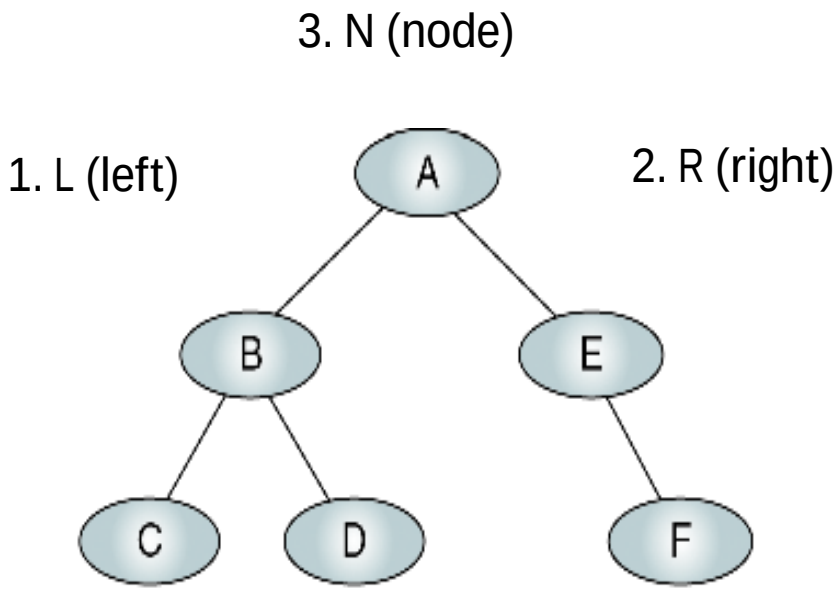



**Inorder – C B D A E F**

# PostOrder (LRN)

**Algorithm Postorder (root):** Traverse a binary tree in left-right-node sequence.

1. If (root is not null)
2. Postorder (leftSubtree)
3. Postorder (rightSubtree)
4. process (root)




**Postorder – C D B F E A**

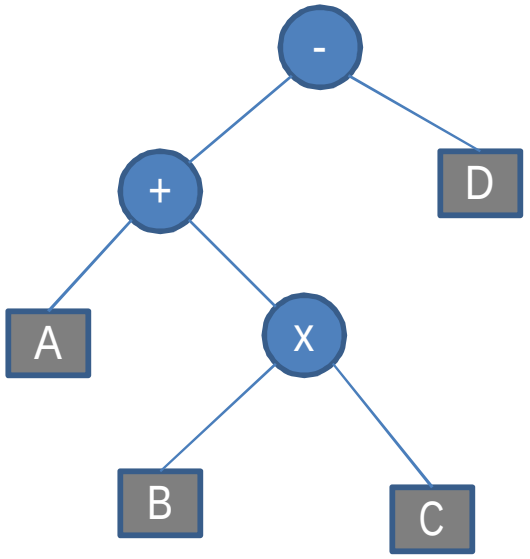
# Expression Tree

- An expression tree is a binary tree with the following properties:
  1. Each leaf is an operand
  2. The root and internal nodes are operators
  3. Sub trees are sub expressions, with the root being an operator

1. Infix traversal
2. Prefix traversal
3. Postfix traversal

Example – Infix expression A + (B x C) - D

Infix expression tree

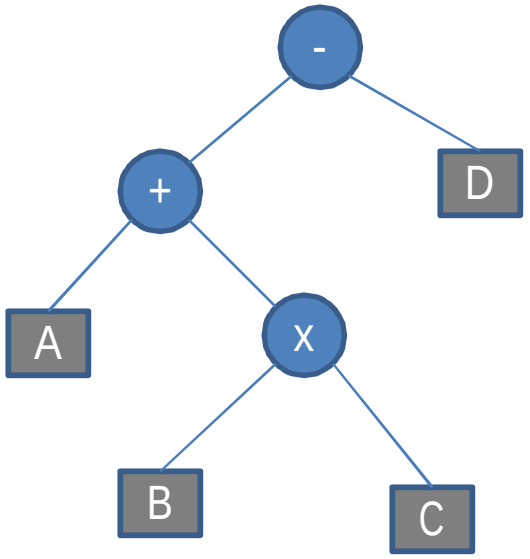


# Prefix traversal

***Algorithm Prefix(tree):*** Print prefix expression for a given expression tree

1. If (tree is not empty)
2. print token
3. Prefix (leftSubtree)
4. Prefix (rightSubtree)

Prefix expression - + A x B C D

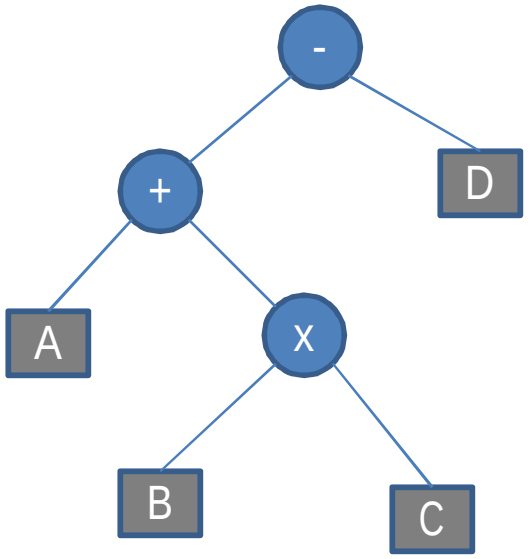


# Postfix traversal

***Algorithm Postfix(tree):*** Print postfix expression for a given expression tree

1. If (tree is not empty)
2. Postfix (leftSubtree)
3. Postfix (rightSubtree)
4. print token

Postfix expression A B C x + D -

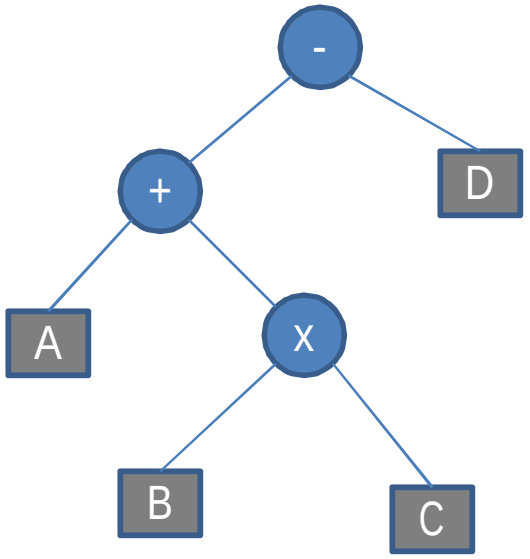


# Infix traversal

***Algorithm Infix(tree):*** Print infix expression for a given expression tree

```
1.  If (tree is not empty)
2.          If (token is operand)
3.                  print token
4.          else
5.                  print left parenthesis
6.                  Infix (leftSubtree)
7.                  print token
8.                  Infix (rightSubtree)
9.                  print right parenthesis
```

Infix expression ((A + (B x C)) – D)



# Exercises

Draw Expression Tree –

1. A + B * C - D / E
2. A * B - ( C + D ) + E
3. A + (B * C – (D / E ↑ F) * G) * H

PostFix Traversal –

1. A B C * + D E / -
2. A B * C D + - E +
3. A B C * D E F ↑ / G * - H * +

PreFix Praversal–

1. - + A * B C / D E
2. + - * A B + C D E
3. + A * - * B C * / D ↑ E F G H

# Problem

Which of the following is a true about Binary Trees
**(A)** Every binary tree is either complete or full.
**(B)** Every complete binary tree is also a full binary tree.
**(C)** Every full binary tree is also a complete binary tree.
**(D)** No binary tree is both complete and full.
**(E)** None of the above

**Answer: (E)**

**A binary tree T has 20 leaves. The number of nodes in T having two children is**

**(A)** 18
**(B)** 19
**(C)** 17
**(D)** Any number between 10 and 20

**Answer: (B)**

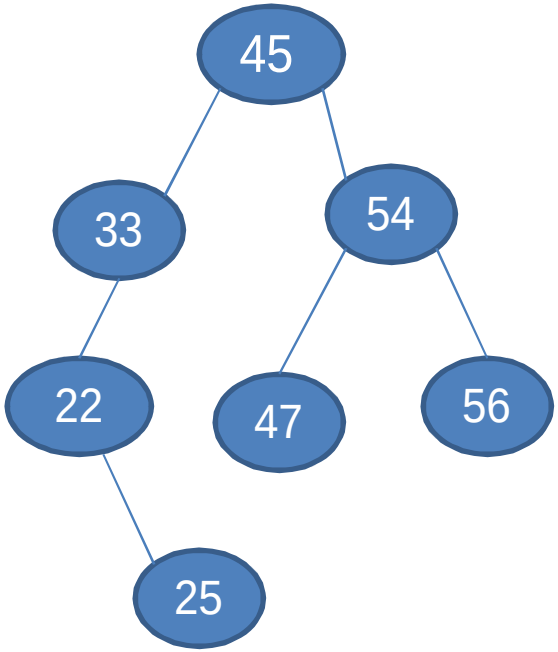
# Binary Search Tree (BST)

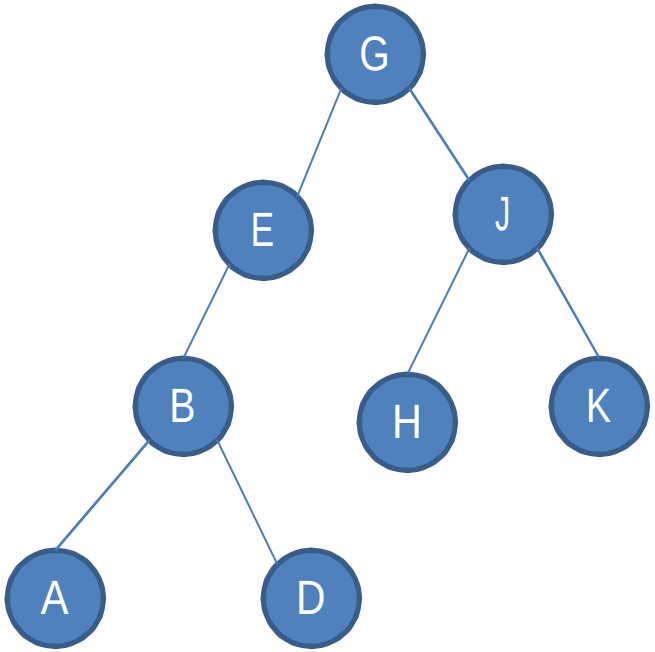
Binary tree with following properties –

1. Every element has a unique key.
2. The left sub tree of a node contains only nodes with keys less than the node's key.
3. The right sub tree of a node contains only nodes with keys greater than the node's key.
4. The left and right sub trees are also binary search trees.

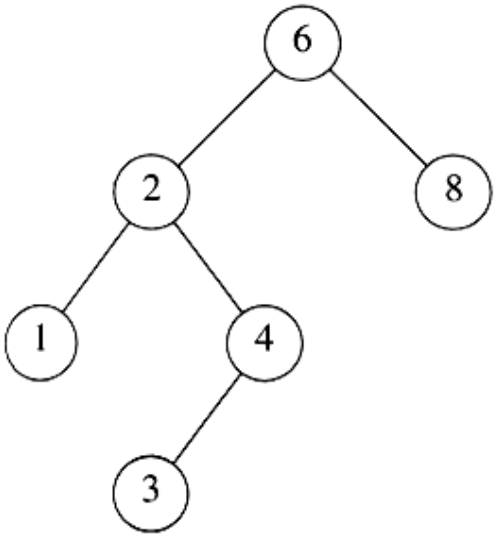
- Binary Search Trees (BST) are type of binary trees with a special organization of data
- This data organization leads to O(log n) complexity for searches, insertions and deletions in certain types of the BST (balanced trees). But for worst case it would be O(n).
- For getting all advantages of BST, it should be AVL or height balance tree.

# BST Examples

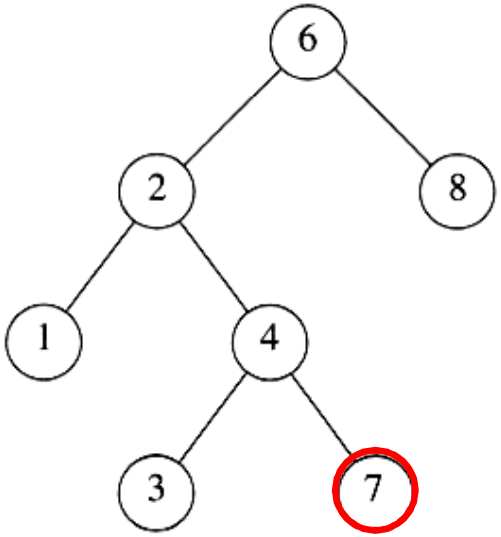


# BST Examples



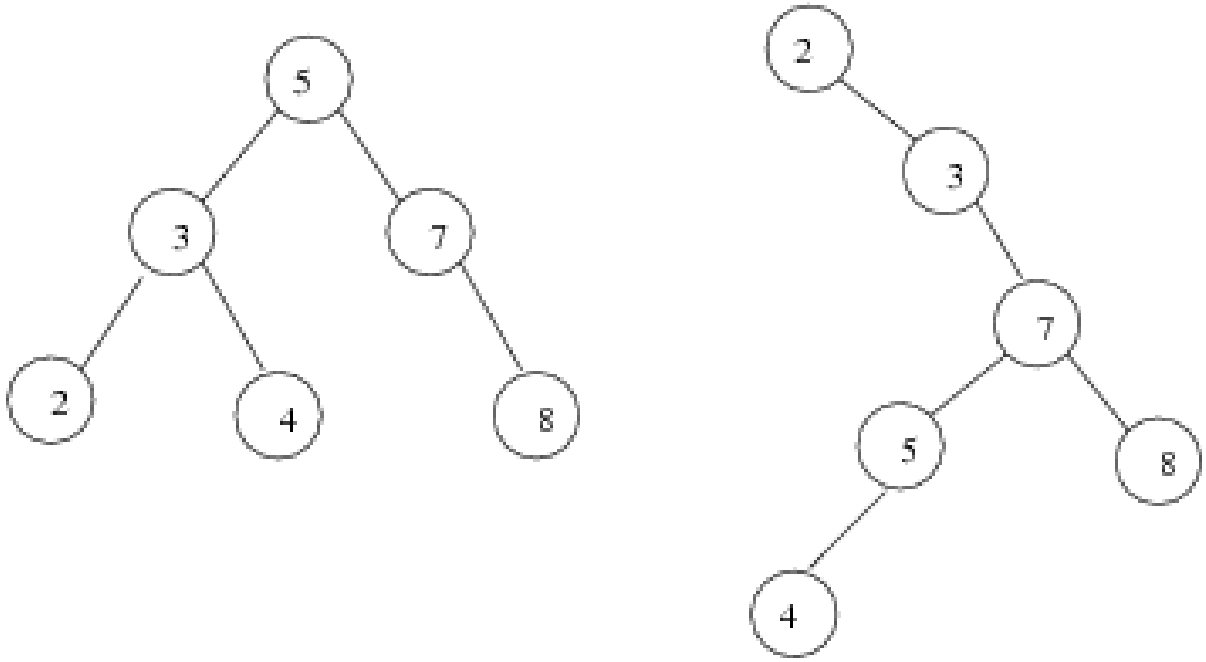

A binary search tree




Not a binary search tree

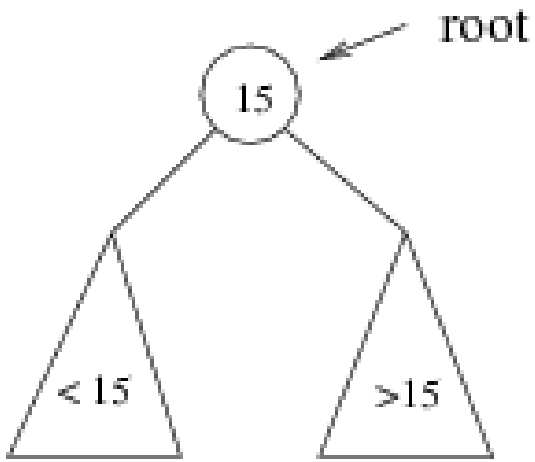
# BST Examples

Two binary search trees representing the same set:



# Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.

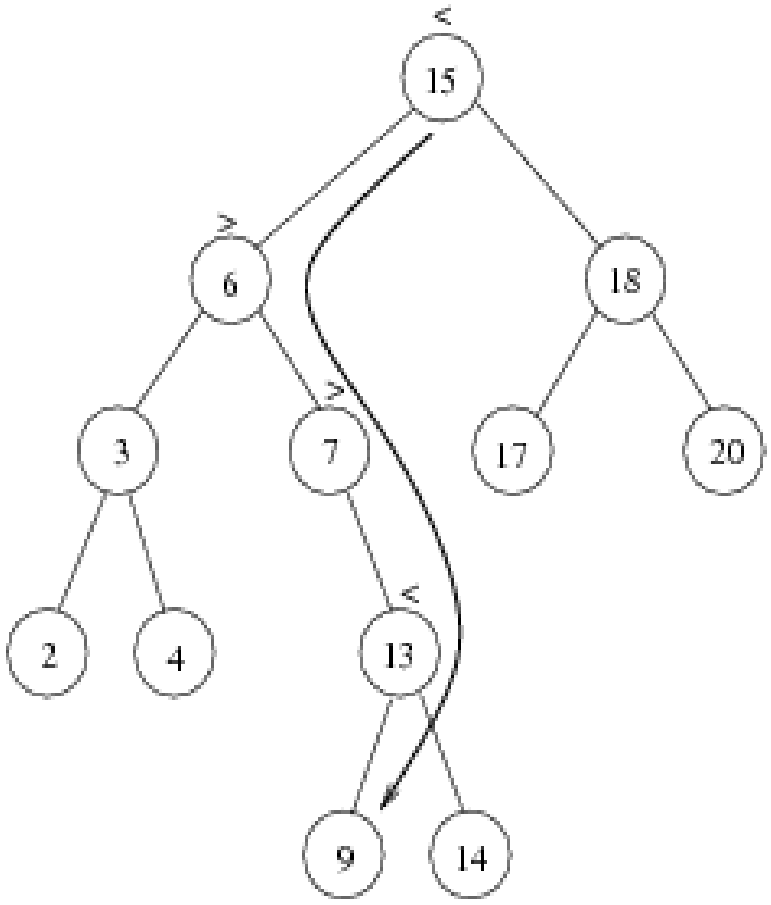


# Searching BST

*Example:* Search for 9 ...

Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!



# Searching a node in BST

***Algorithm Search_BST(ROOT , Key):*** This algorithm searches Key in a BST . This procedure finds the location LOC of Key in tree. ROOT is variable pointing to root of tree. This algorithm also finds the parent location PAR of Key , Initially LOC =NULL and PAR=NULL.

```
1.  Set LOC =NULL and PAR=NULL
2.  If ROOT == NULL   // Empty Tree
3.          Exit
4.  If ROOT->Data == Key   // Tree with only root node
5.          LOC = ROOT   //  Location of  searched element
6.          PAR = NULL     // location of the parent of the searched element
7.          Exit
8.  Temp = ROOT  //Tree has more than one nodes
10. While (Temp !=NULL)
11.         PAR = Temp
12.         If  Key< (Temp->Data)
13.                 Temp = Temp ->lc
14.         Else        Temp = Temp ->rc
15.         If Temp->Data == Key
16.                 LOC = Temp
17.                 BREAK
18. If LOC == NULL
19.         Print Search Unsuccessful.
```

# Insertion in BST

***Algorithm Insert_BST(ROOT, Key):*** This algorithm inserts a new leaf node with value Key in BST at appropriate location. ROOT is variable pointing to root of tree. This algorithm uses sub algorithm Search_BST to find the Key location LOC and parent location PAR of Key.

```
1.  Call Search_BST(ROOT, Key)
2.  If LOC != NULL  // exit if Key is already present
3.          Exit
4.  Node * New_Node= Allocate memory
5.  New_Node ->Data = Key
6.  New_Node > lc = NULL
7.  New_Node -> rc = NULL
8.  If ROOT == NULL  // if Tree is Empty previously
9.          ROOT = New_Node
10. Else If (Key<(PAR->Data) )
11.         PAR->lc = New_Node // make left child of parent
12. Else
13.         PAR->rc = New_Node // make right child of parent
```

# Deletion in BST

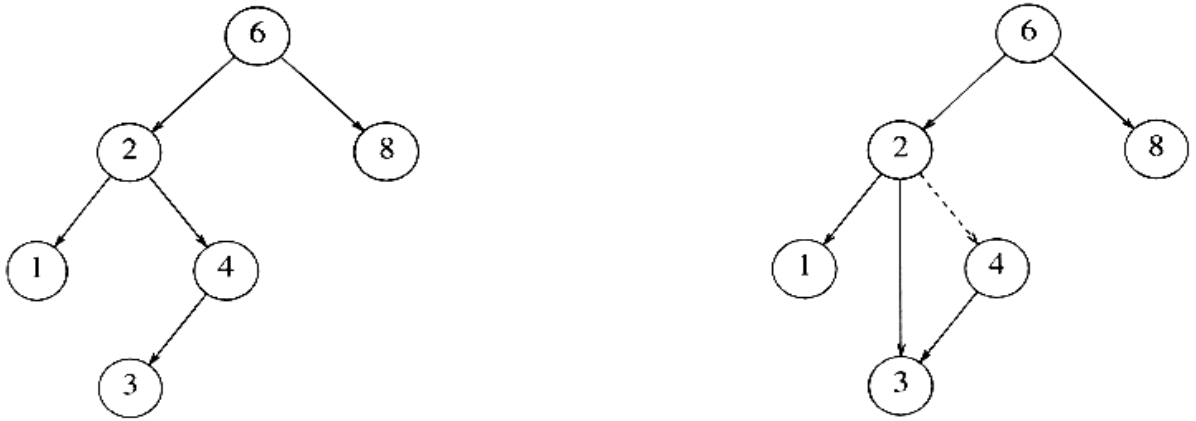
- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the BST is maintained.
  - Three cases to be consider for deletion

# Deletion in BST

Three cases in deletion –Suppose we want to delete a node X from BST,

***Case 1:*** X has no child, then delete X from tree and make its location in parent node null

***Case 2:*** X has exactly one child, then X will be deleted by replacing the location of X in its parent node by the location of its only child




***Case 3:*** X has two children

# Case A (case 1 and case 2)

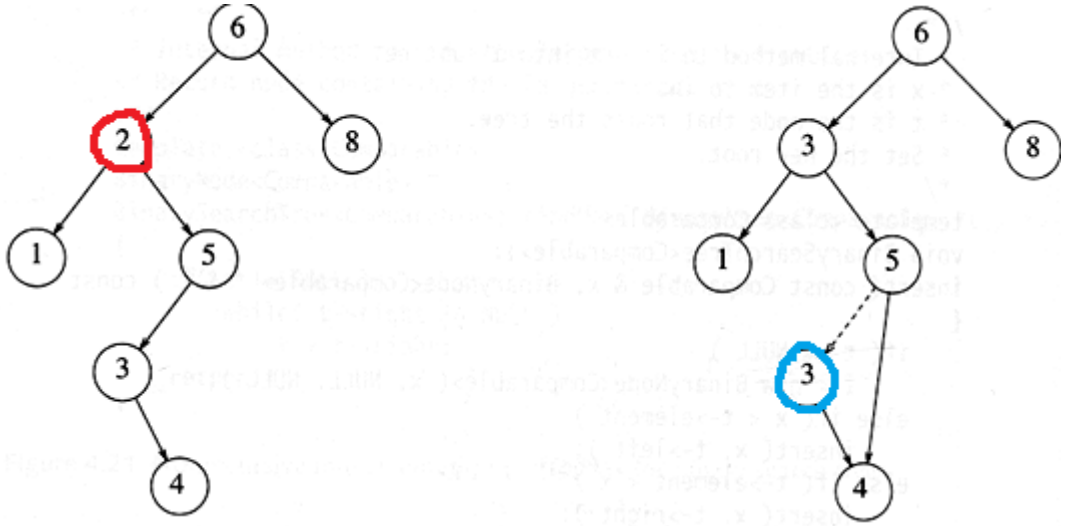
***Algorithm CaseA_delete(ROOT, Key):*** This algorithm deletes a node X with value **Key** from tree, if this node is not having two children. This algorithm uses sub algorithm Search_BST to find the Key location LOC and parent location PAR of Key. CHILD is a pointer variable pointing to child of node X.

```
1.  Call Search_BST(ROOT, Key)
2.  Node *CHILD
3.  If LOC == NULL  // Node with Key  does not exist
4.          Exit
5.  If LOC ->lc ==NULL
6.          CHILD = LOC ->rc
7.  Else
8.          CHILD = LOC ->lc
9.  If PAR != NULL  //  node to be deleted is not the root node
10.         If LOC == PAR->lc
11.                 PAR ->lc = CHILD
12.         Else
13.                 PAR ->rc = CHILD
14. Else     //  delete root
15.         ROOT = NULL
16. Free LOC
```

# Deletion in BST (case 3)

X has two children

- Replace the data value of node X with the minimum element at the right subtree (i.e. replace with in-order successor of X)
- Delete in-order successor
  - Note : in-order successor always has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.



# Case B (Delete Node which is having two children)

***Algorithm CaseB_delete(ROOT, Key):*** This algorithm deletes a node X with Key from tree, if this node is having two children. This algorithm uses sub algorithm Search_BST to find the Key location LOC and parent location PAR of Key. SUC is a pointer variable pointing in order successor of node X.

```
1. Call Search_BST(ROOT, Key)
2. Node *Temp,*SUC
3. Temp = LOC ->rc
4. While Temp->lc != NULL
5.          Temp = Temp->lc
6. SUC = Temp
7. SUC_Data= SUC->data
8. CaseA_delete(ROOT, SUC->Data)
9. LOC->data= SUC_Data
```

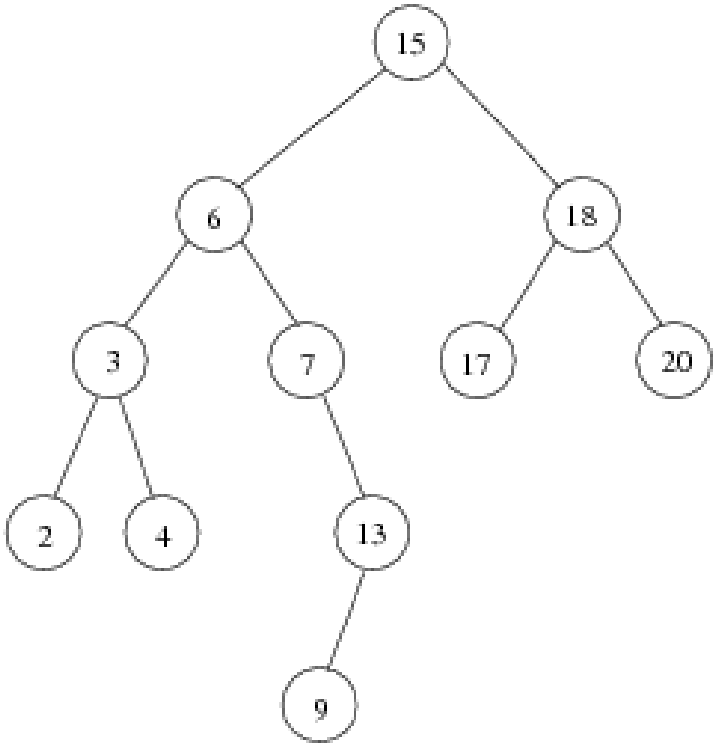
# Deletion in BST (Cont…)

***Algorithm Delete_BST(T, Key):*** This algorithm deletes a node X with Key from BST. ROOT is variable pointing to root of tree T. This algorithm uses sub algorithm Search_BST to find the Key location LOC and parent location PAR of Key.

```
1. Call Search_BST(ROOT, Key)
2. If LOC == NULL
3.         Exit
4. If LOC ->lc !=NULL and LOC ->rc !=NULL // If both children exist
5.         Call CaseB_delete(ROOT,Key)
6. Else
7.         Call CaseA_delete(ROOT,Key)
```

# Inorder traversal of BST

- Inorder traversal of BST always gives sorted order of the elements

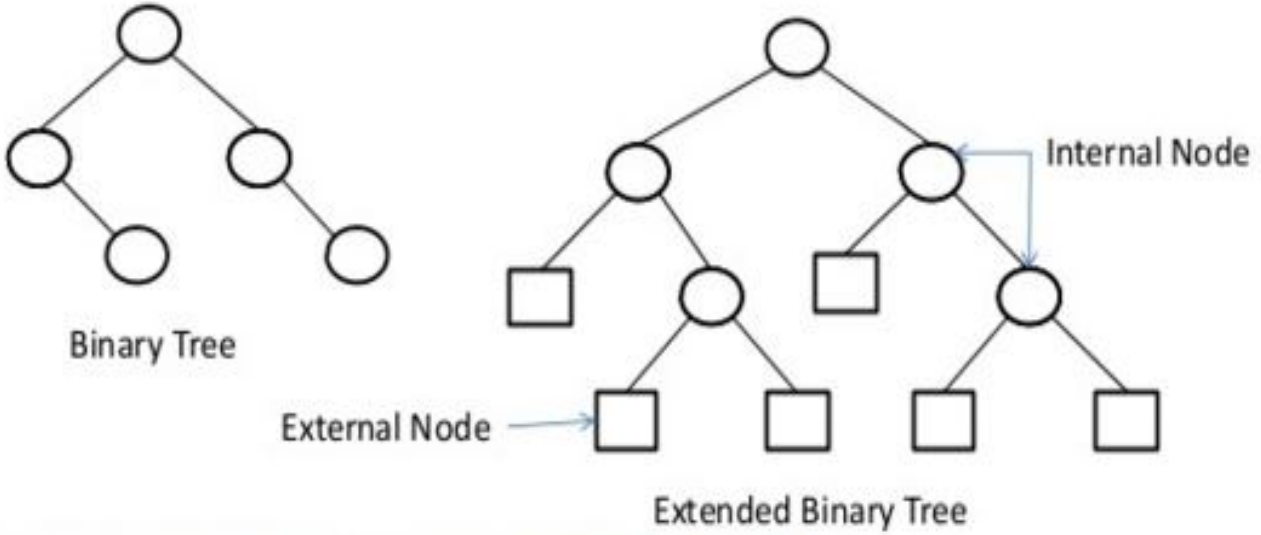



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

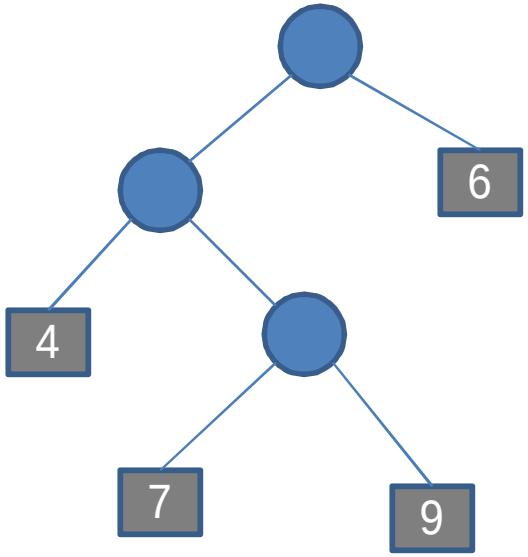
# Extended Binary Tree

•An *extended binary tree* is a transformation of any binary tree into a full binary tree. This transformation consists of replacing every null subtree of the original tree with "special nodes."

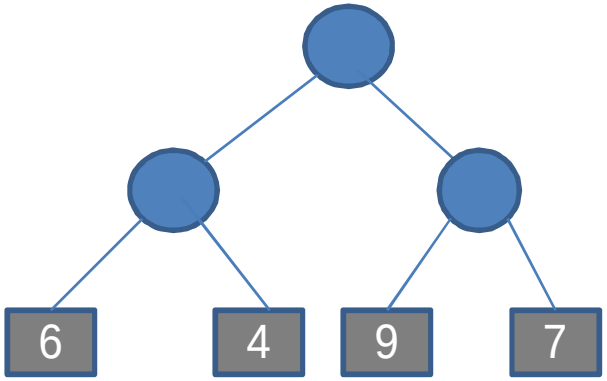
•**The nodes from the original tree are then become *internal nodes*, while the "special nodes" are *external nodes*.**



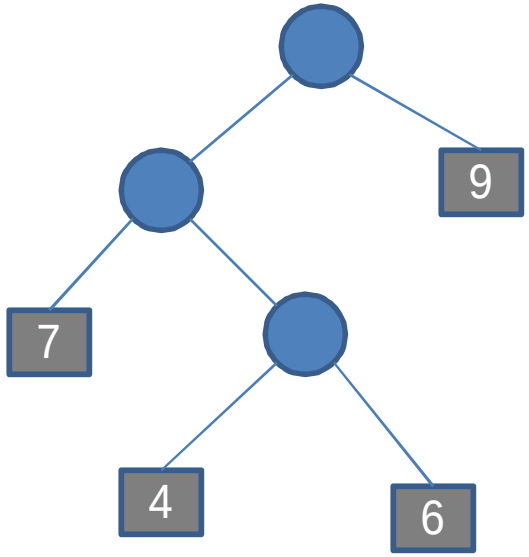
# Extended Binary Tree




(A)




(B)




(C)

(A) Total path length of Tree A= 4*2 + 7*3 + 9*3 + 6*1 = 62

(B) Total path length of Tree B= 6*2 + 4*2 + 9*2 + 7*2 = 52

(C) Total path length of Tree C= 7*2 + 4*3 + 6*3 + 9*1 = 53

# Huffman Coding

- An Application of Binary Trees and Priority Queues.
- Huffman coding is a lossless data compression algorithm.
- In this algorithm, a variable-length code is assigned to different characters.

# Huffman Coding : The Basic Idea

- Not all the characters occur with the same frequency!
- Yet all characters are allocated the same amount of space
  - 1 char = 1 byte, whether it is a or x
- In Huffman encoding 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.

# Huffman Coding Algorithm

1. Scan text to be compressed and count occurrence of all characters.
2. Sort characters based on number of occurrences in text.
3. Build Huffman code tree based on sorted list using the priority queue.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

# Building a Tree

## Scan the original text

- **Consider the following short text:**

  ***Eerie eyes seen near lake.***

- **Count up the occurrences of all characters in the text**

# Building a Tree

## Scan the original text

***Eerie eyes seen near lake.***

- **What characters are present?**

E e r i space
y s n a r l k .

# Building a Tree

## Scan the original text

**Eerie eyes seen near lake.**

- What is the frequency of each character in the text?

| Char | Freq. | Char | Freq. | Char | Freq. |
|---|---|---|---|---|---|
| E | 1 | y | 1 | k | 1 |
| e | 8 | s | 2 | . | 1 |
| r | 2 | n | 2 | | |
| i | 1 | a | 2 | | |
| space | 4 | l | 1 | | |

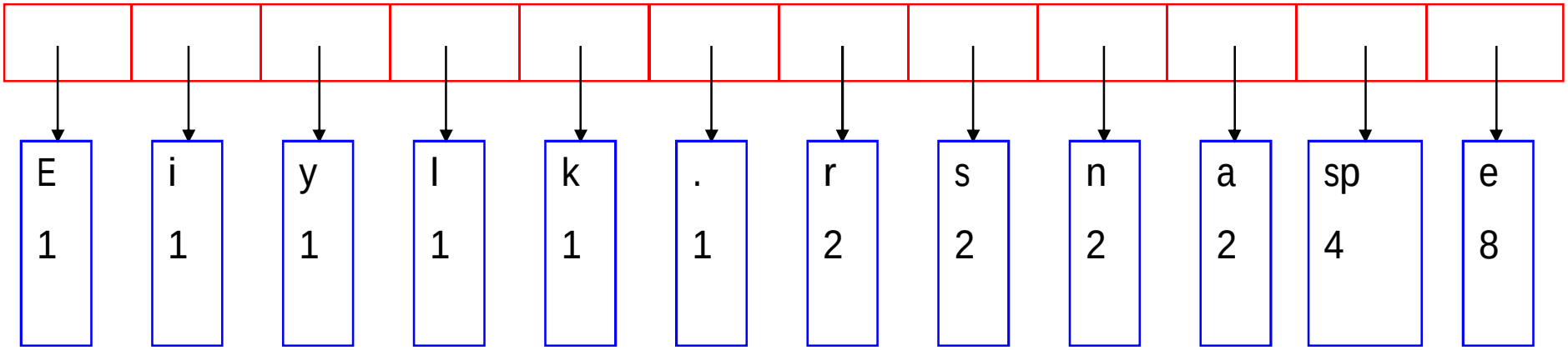
# Building a Tree
## Prioritize characters

- Create binary tree nodes with character and frequency of each character
- Place nodes in a priority queue
  - The lower the occurrence, the higher the priority in the queue

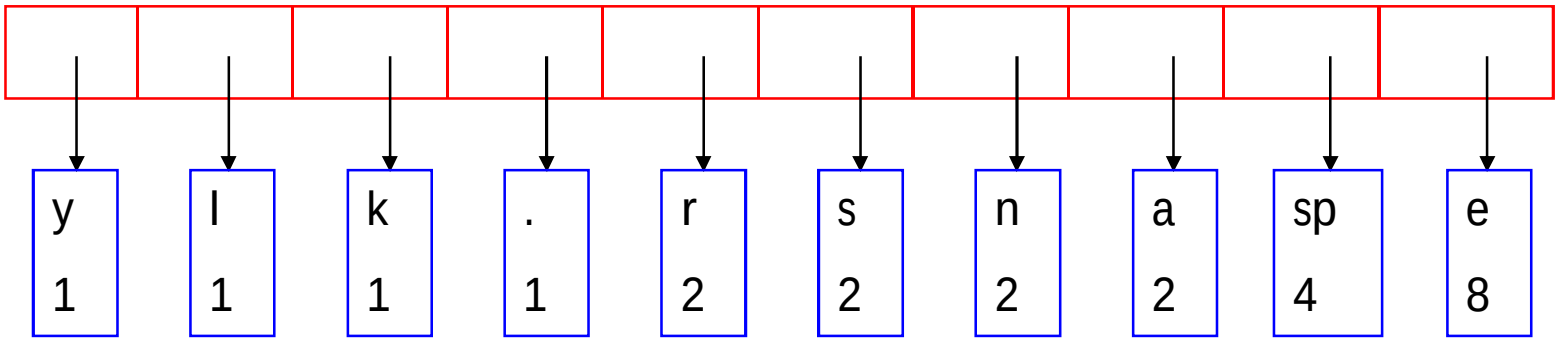
# Building a Tree

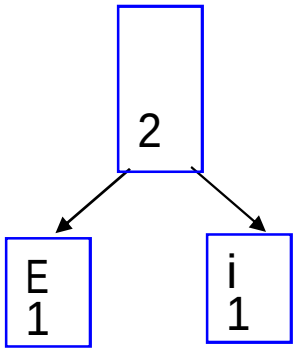
- While priority queue contains two or more nodes
  - Create new node
  - Dequeue node and make it left subtree
  - Dequeue next node and make it right subtree
  - Frequency of new node equals sum of frequency of left and right children
  - Enqueue new node back into queue

# Building a Tree

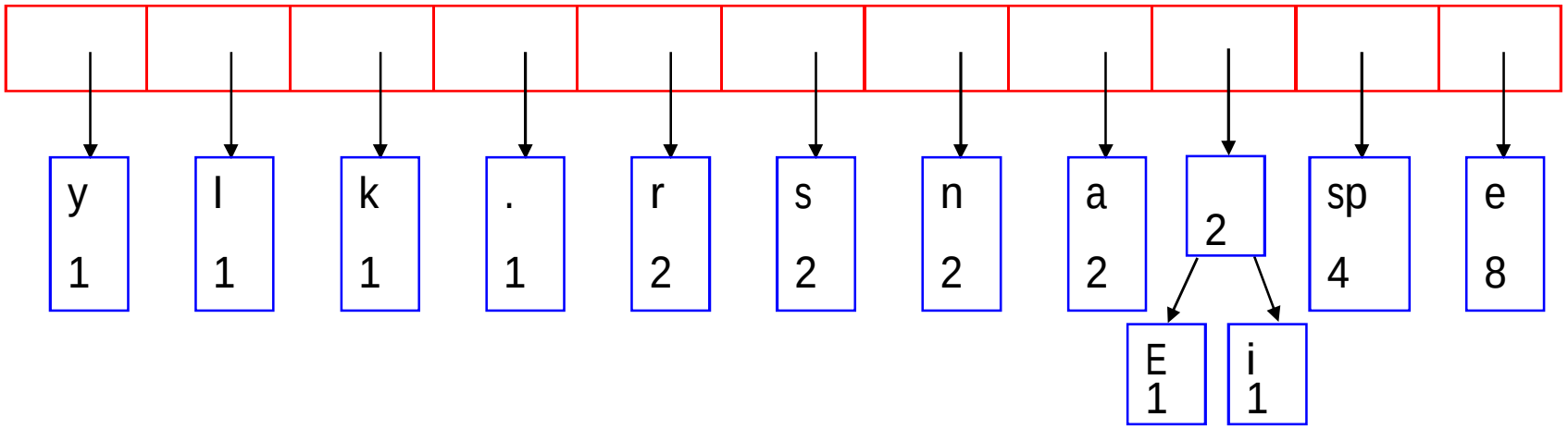


# Building a Tree

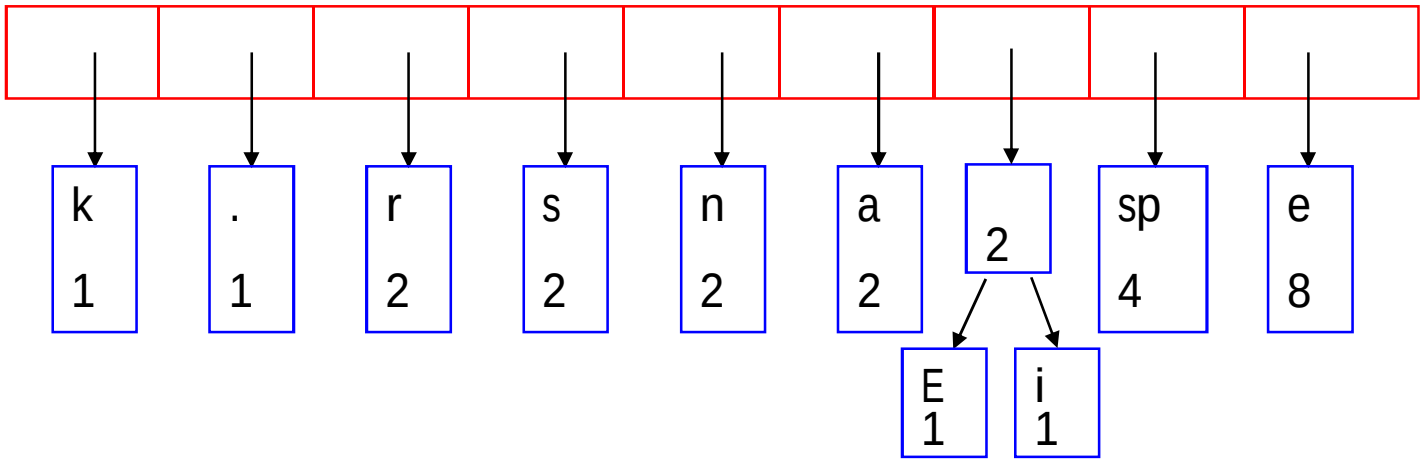


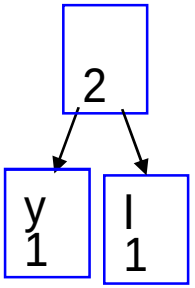



# Building a Tree

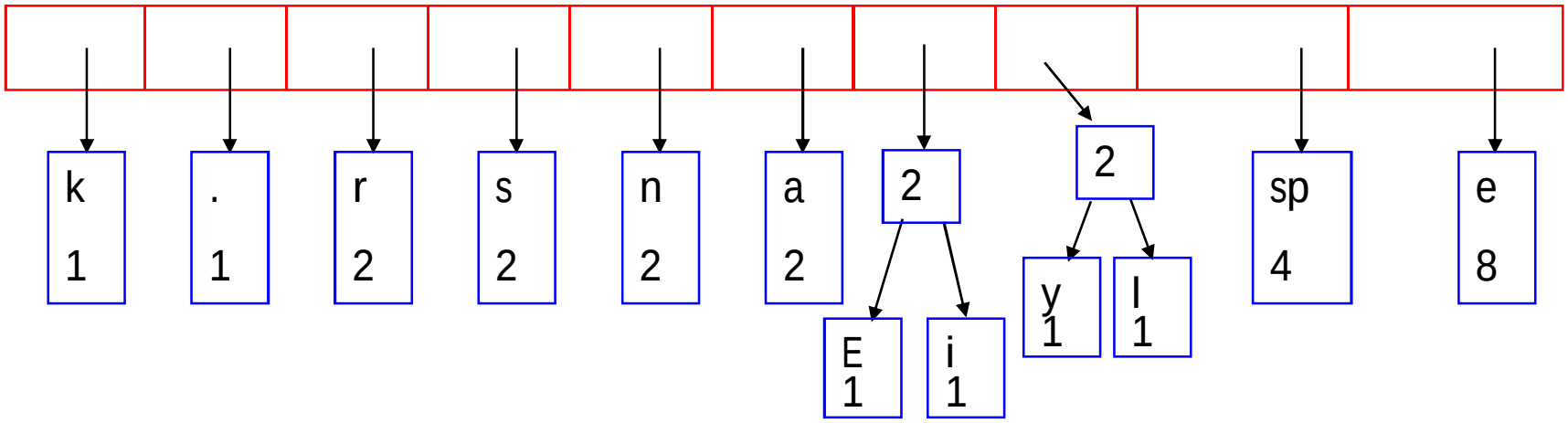


# Building a Tree

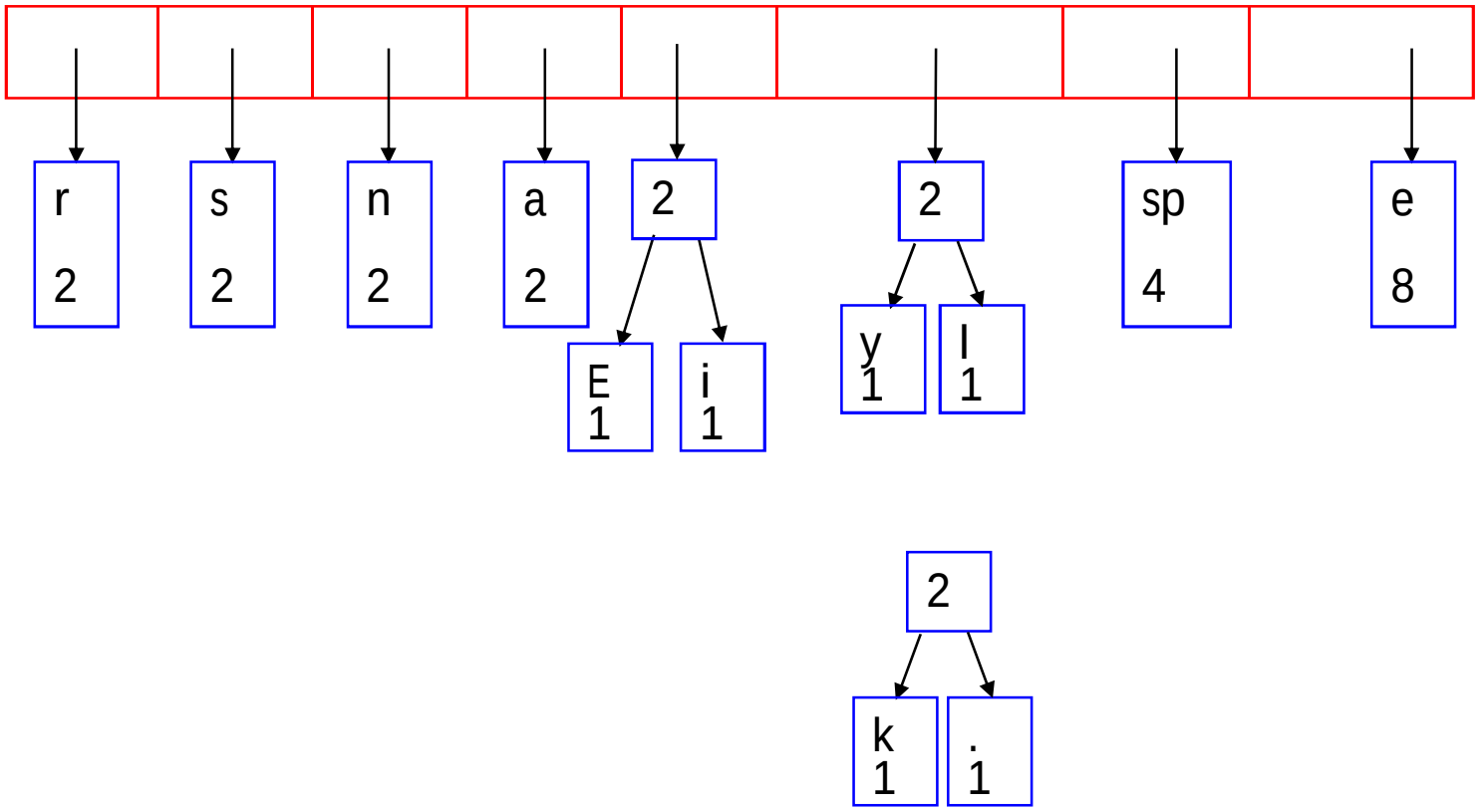


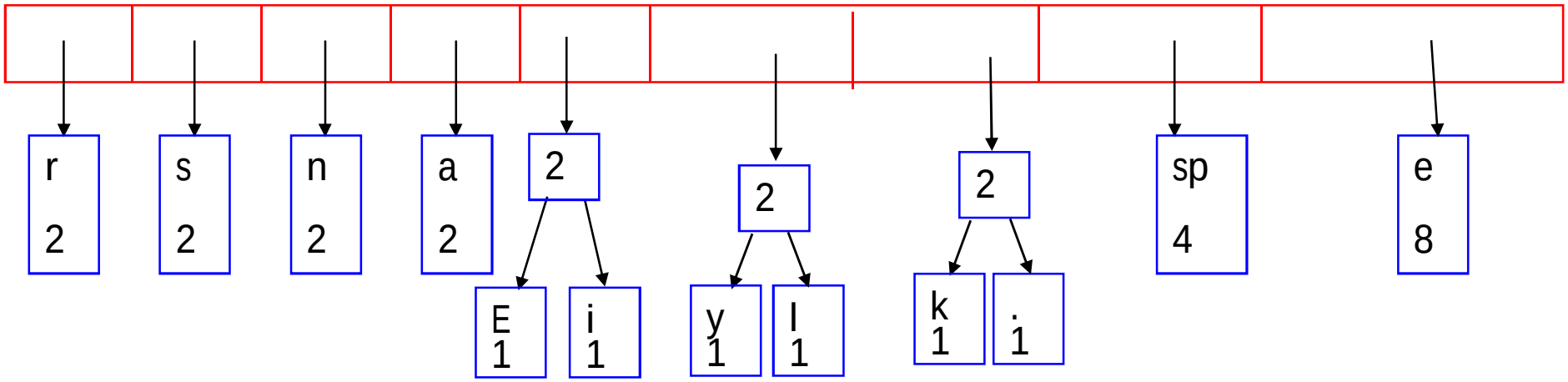
# Building a Tree



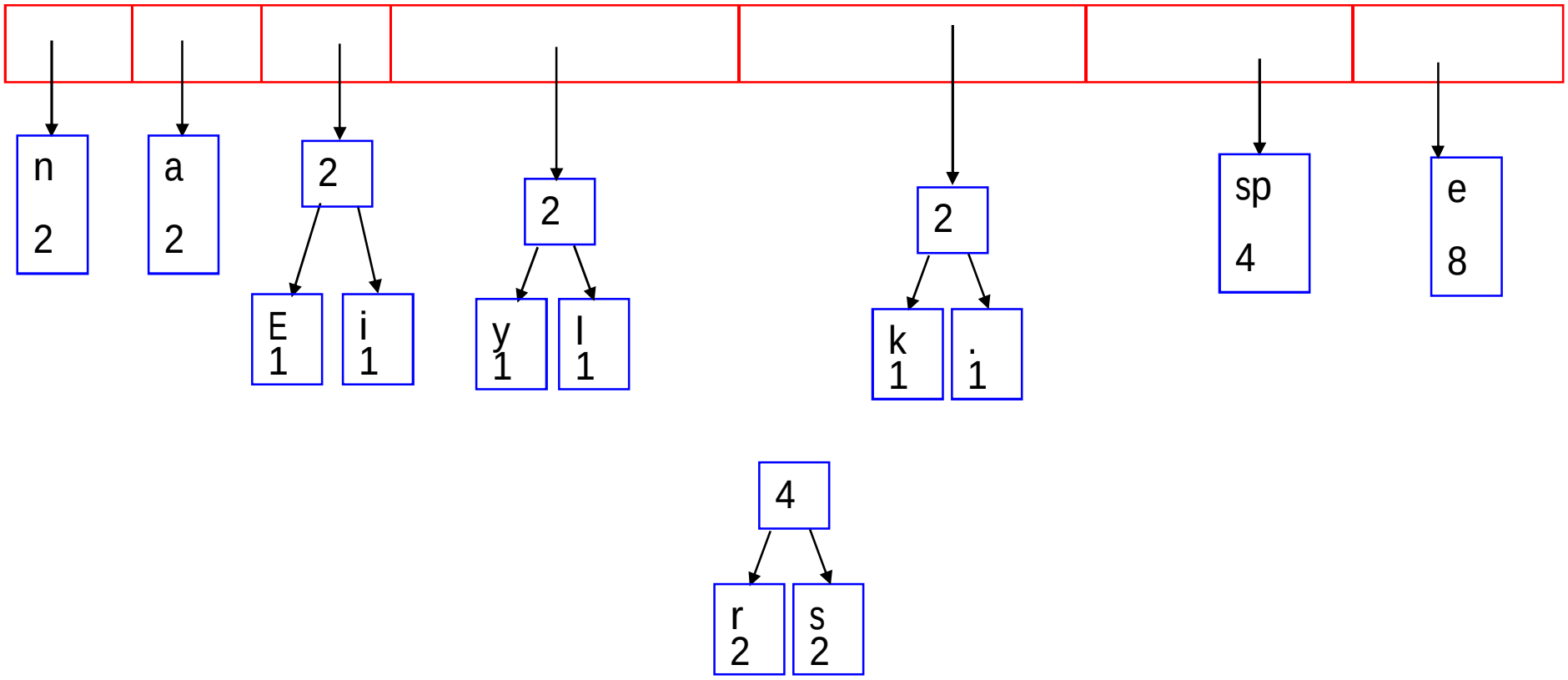
# Building a Tree



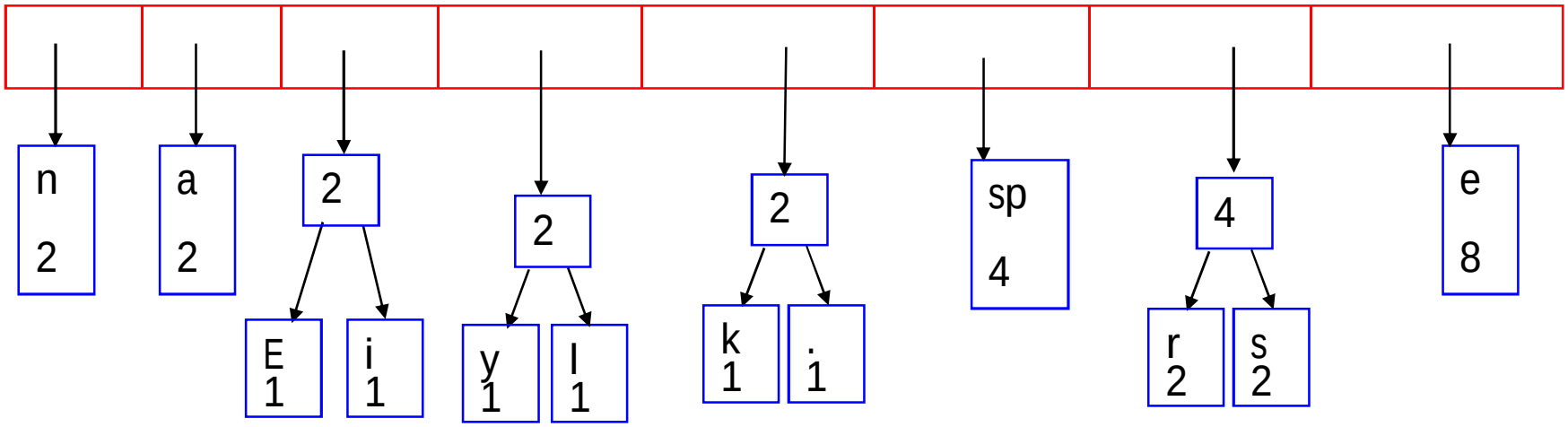
# Building a Tree



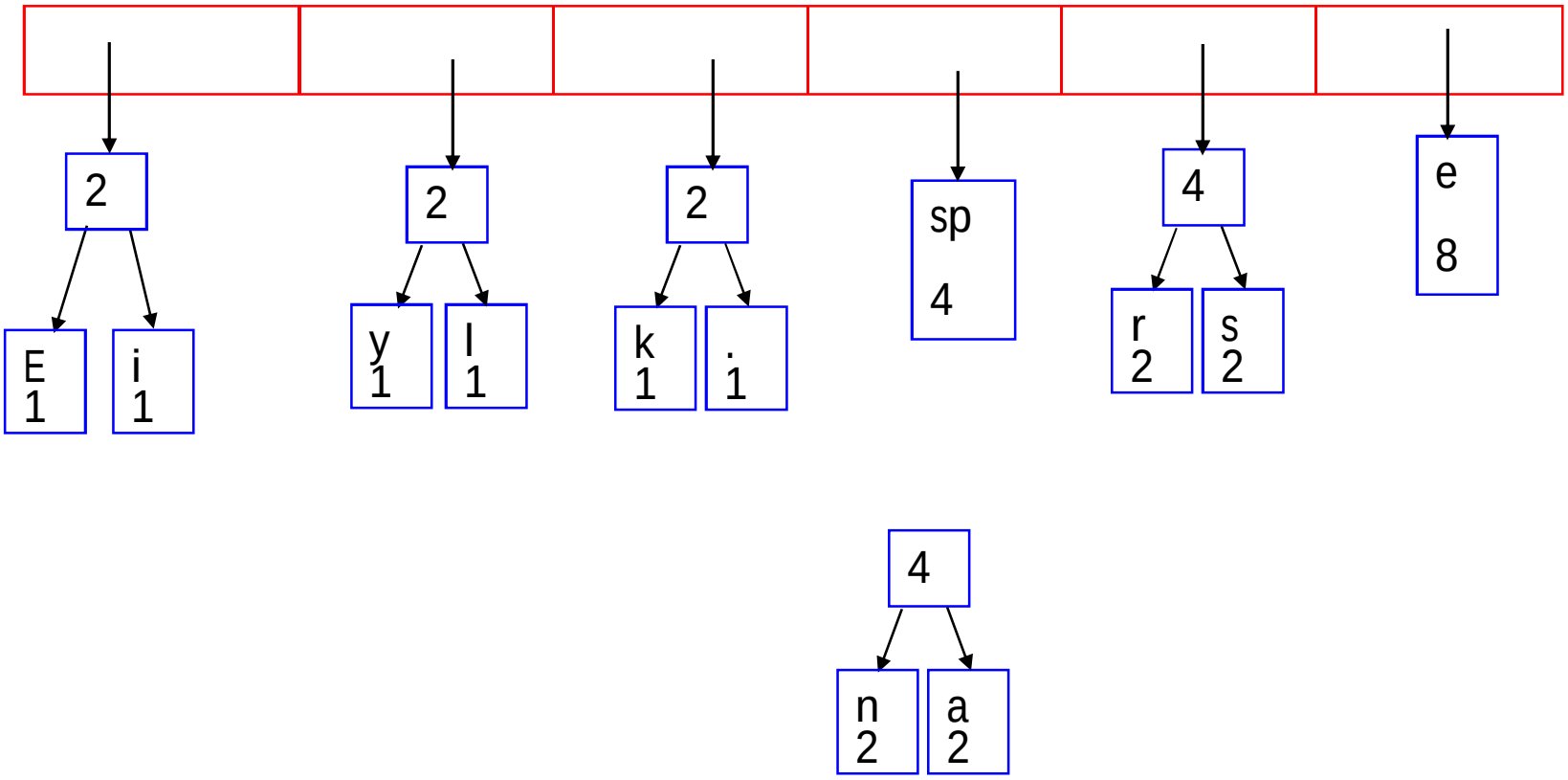
# Building a Tree



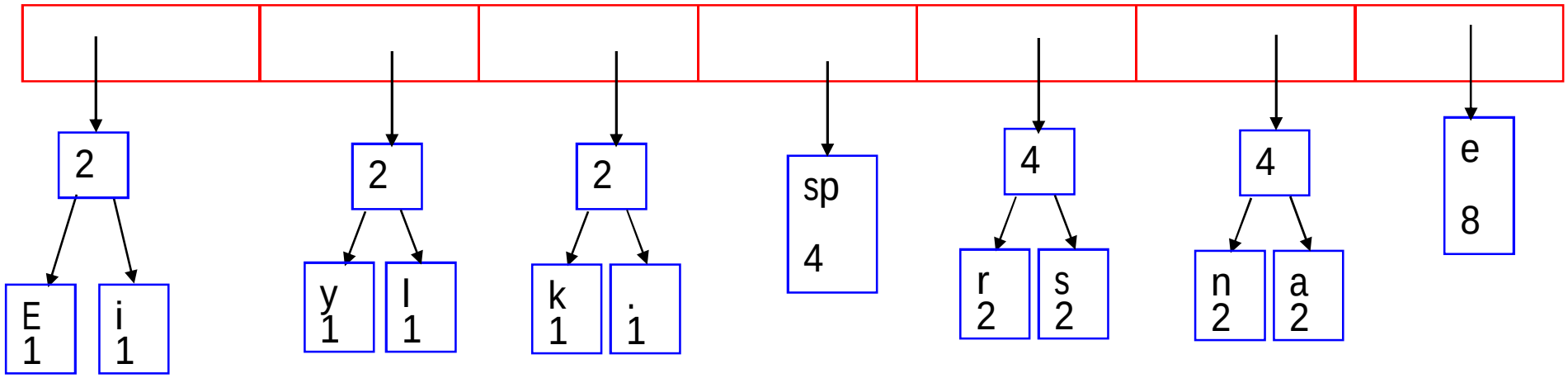
# Building a Tree



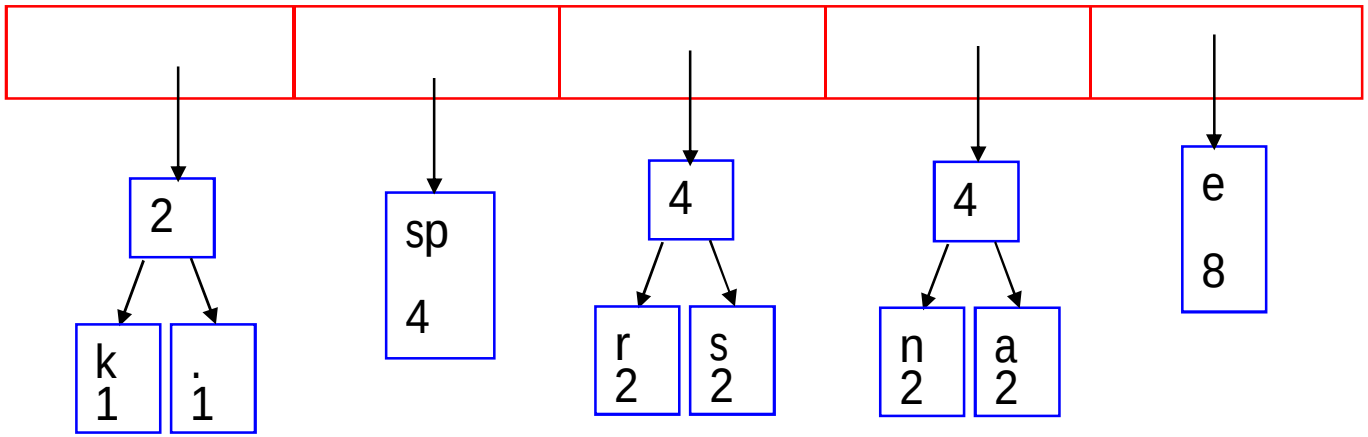
# Building a Tree

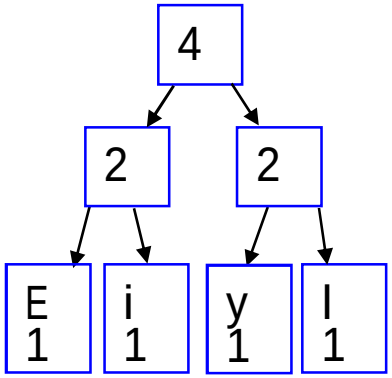


# Building a Tree

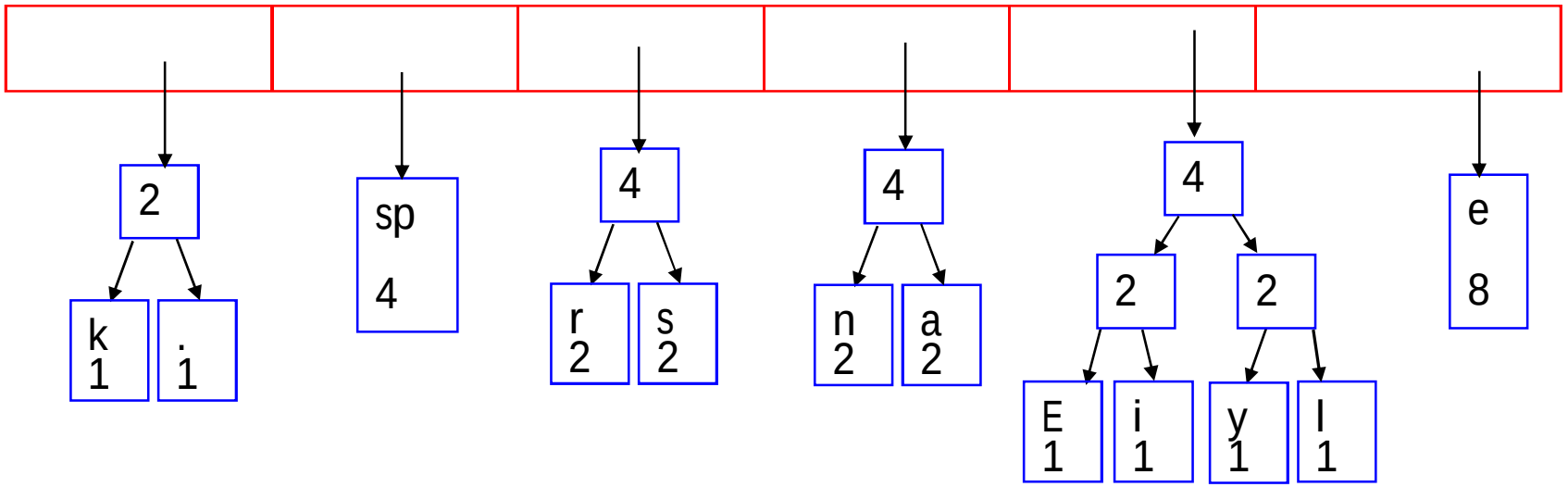


# Building a Tree

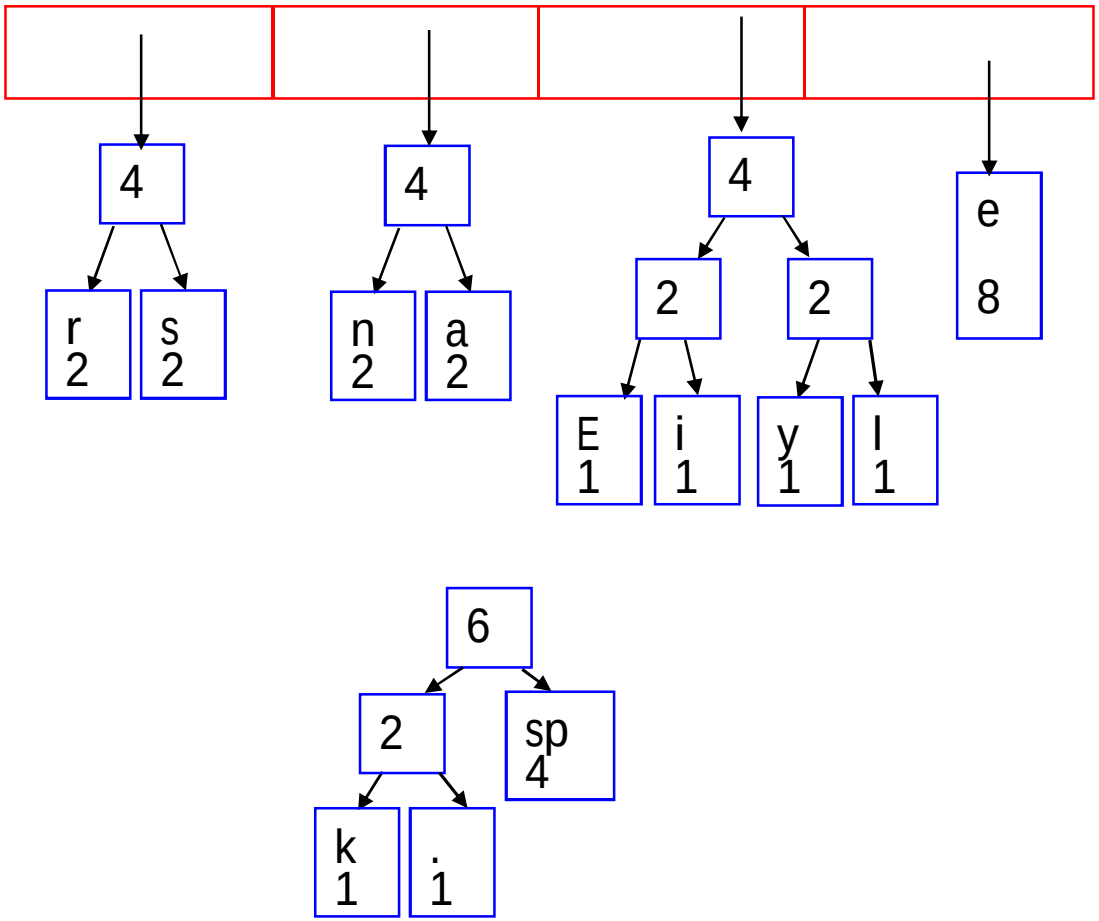


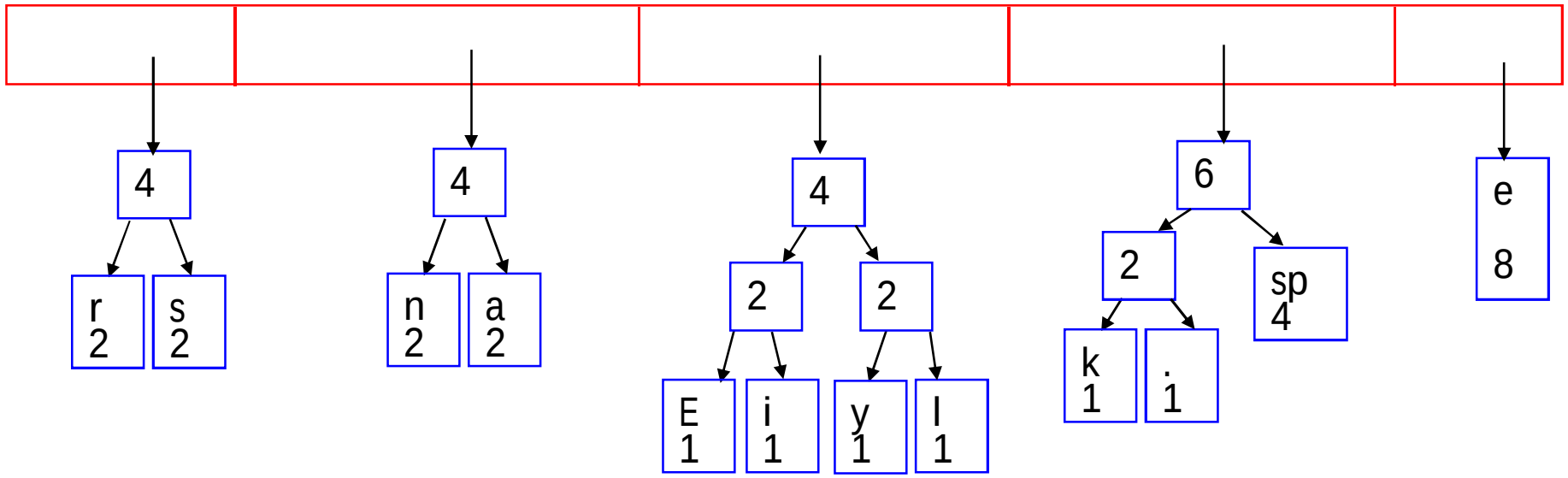
# Building a Tree



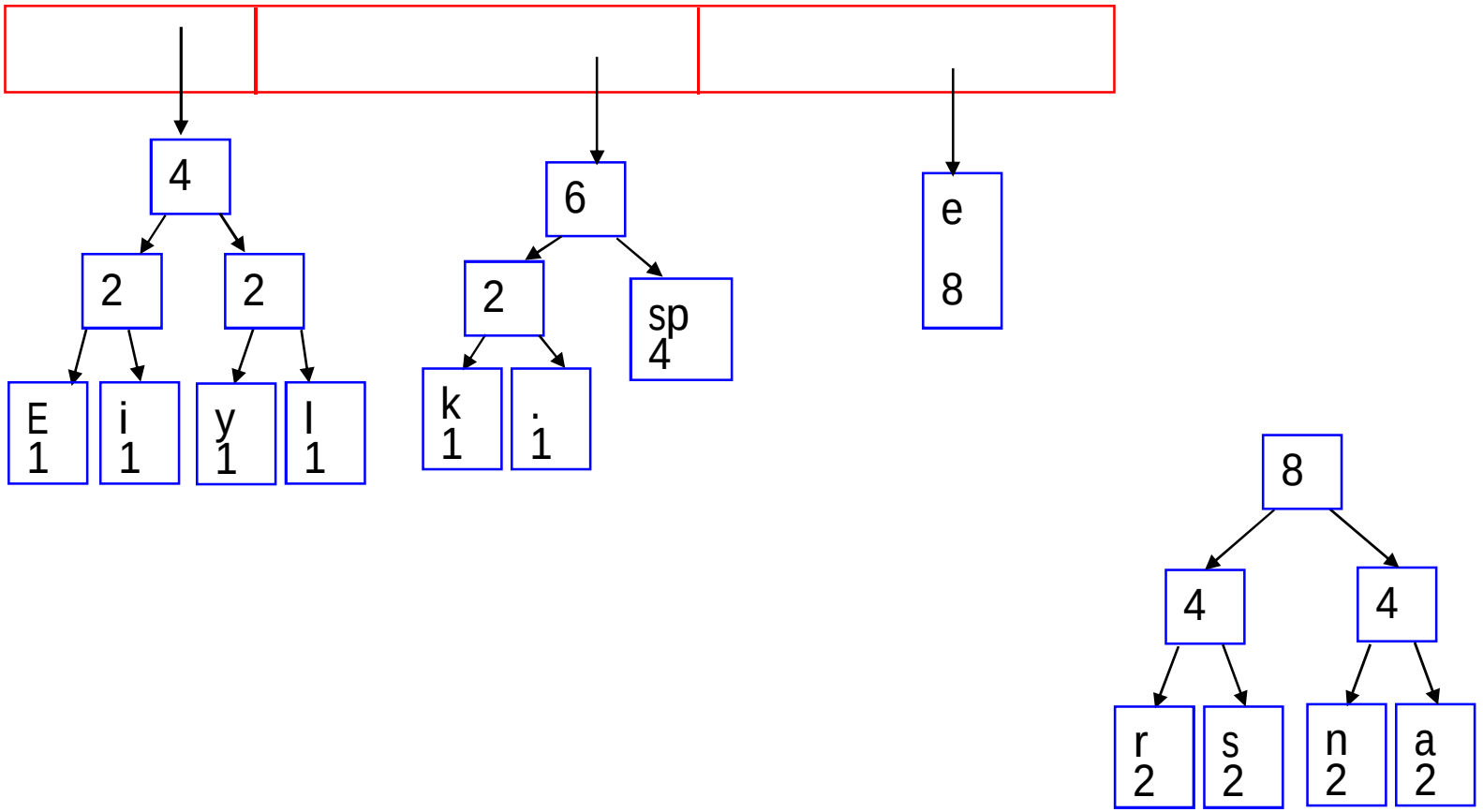
# Building a Tree



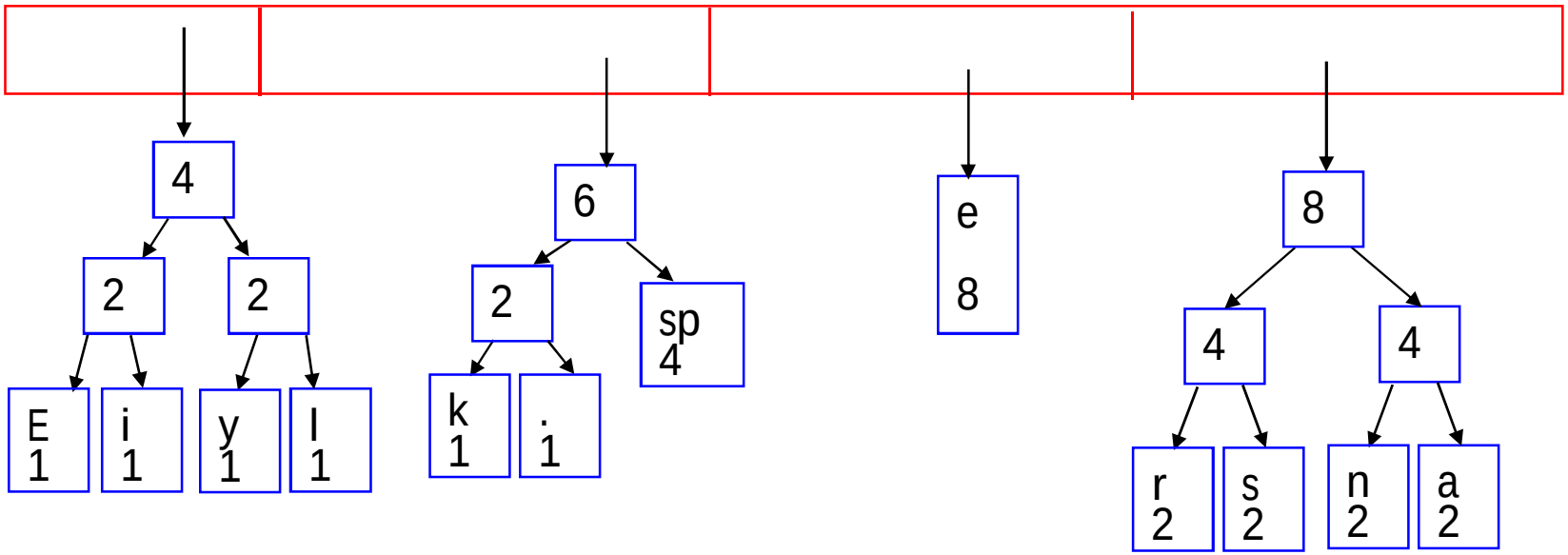
# Building a Tree



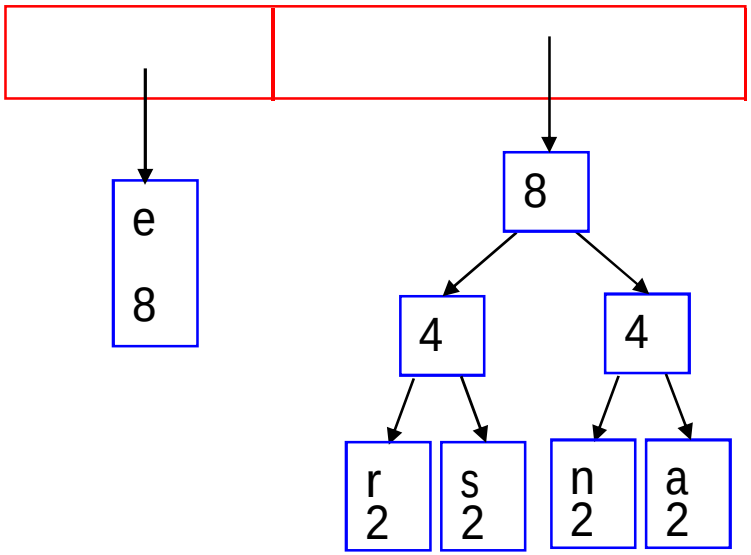
# Building a Tree

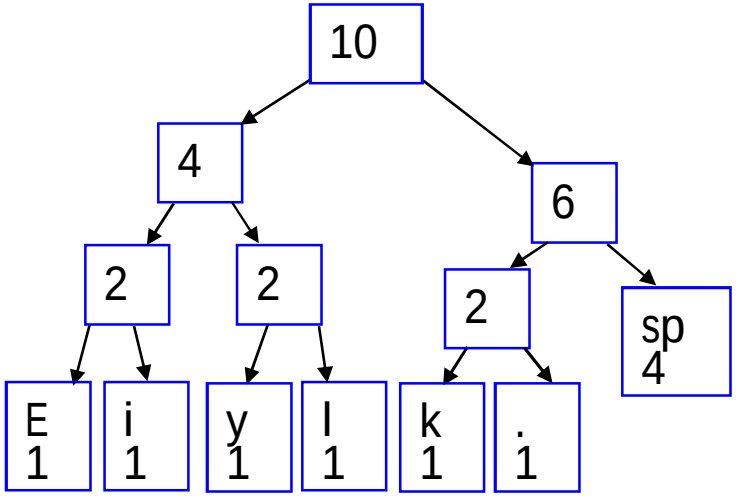


# Building a Tree

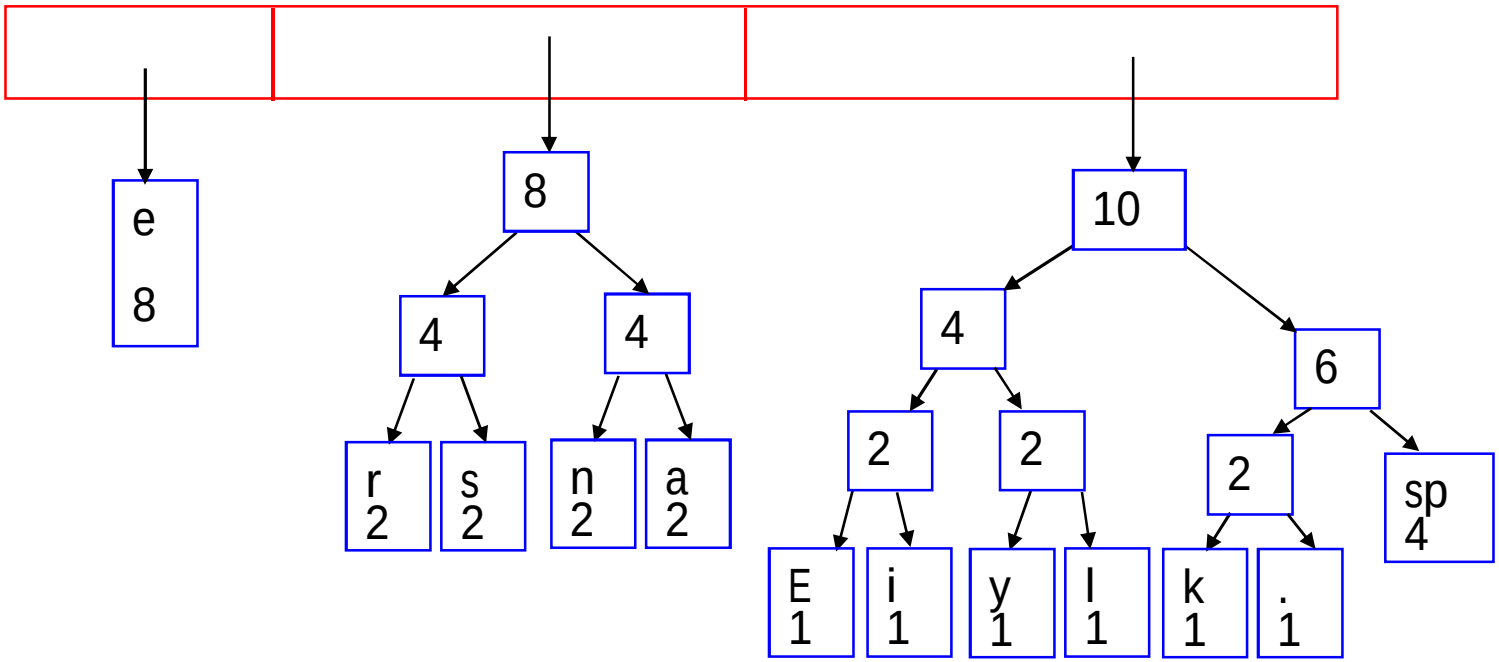


# Building a Tree

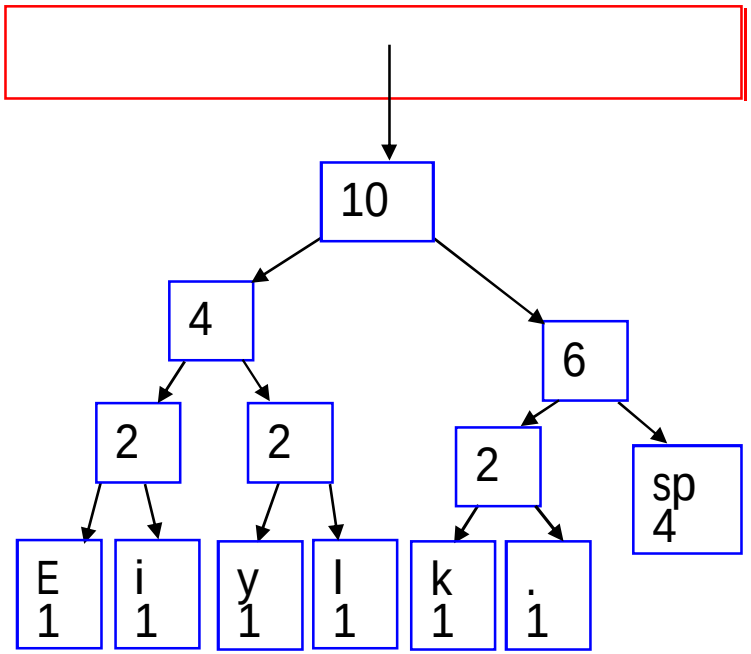


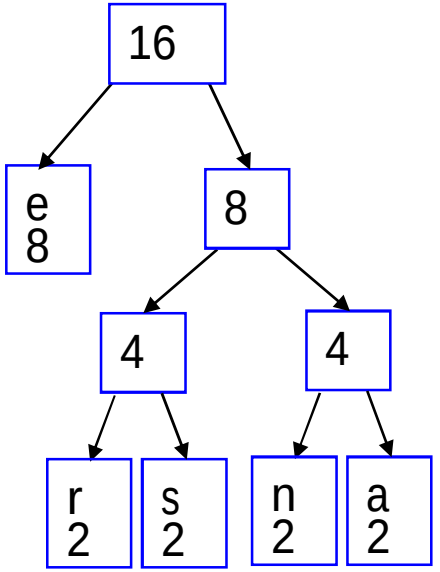



# Building a Tree

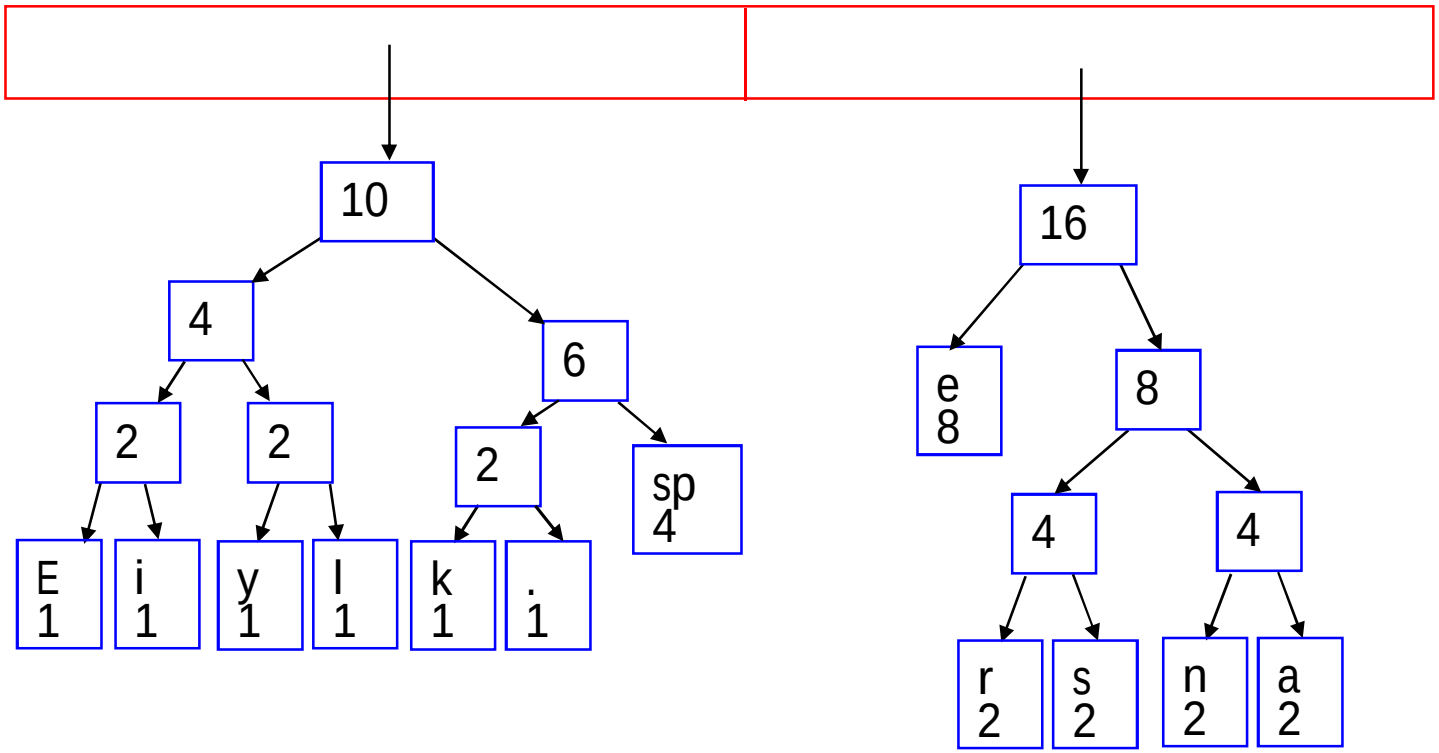


# Building a Tree

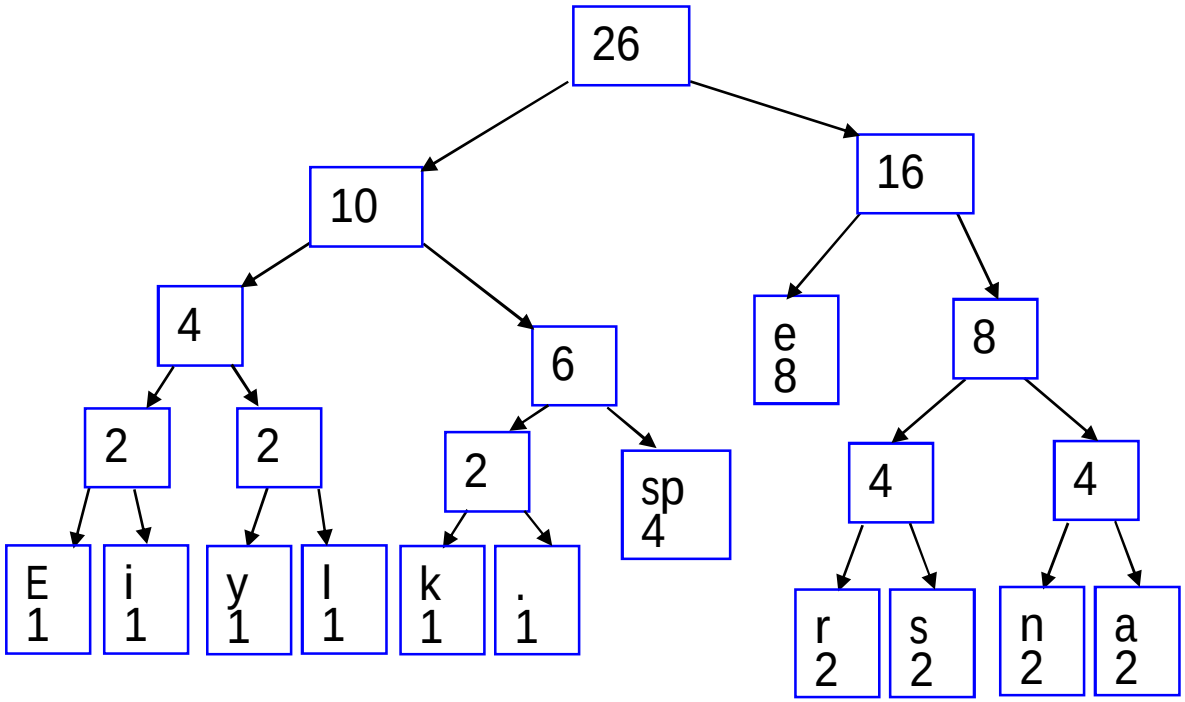


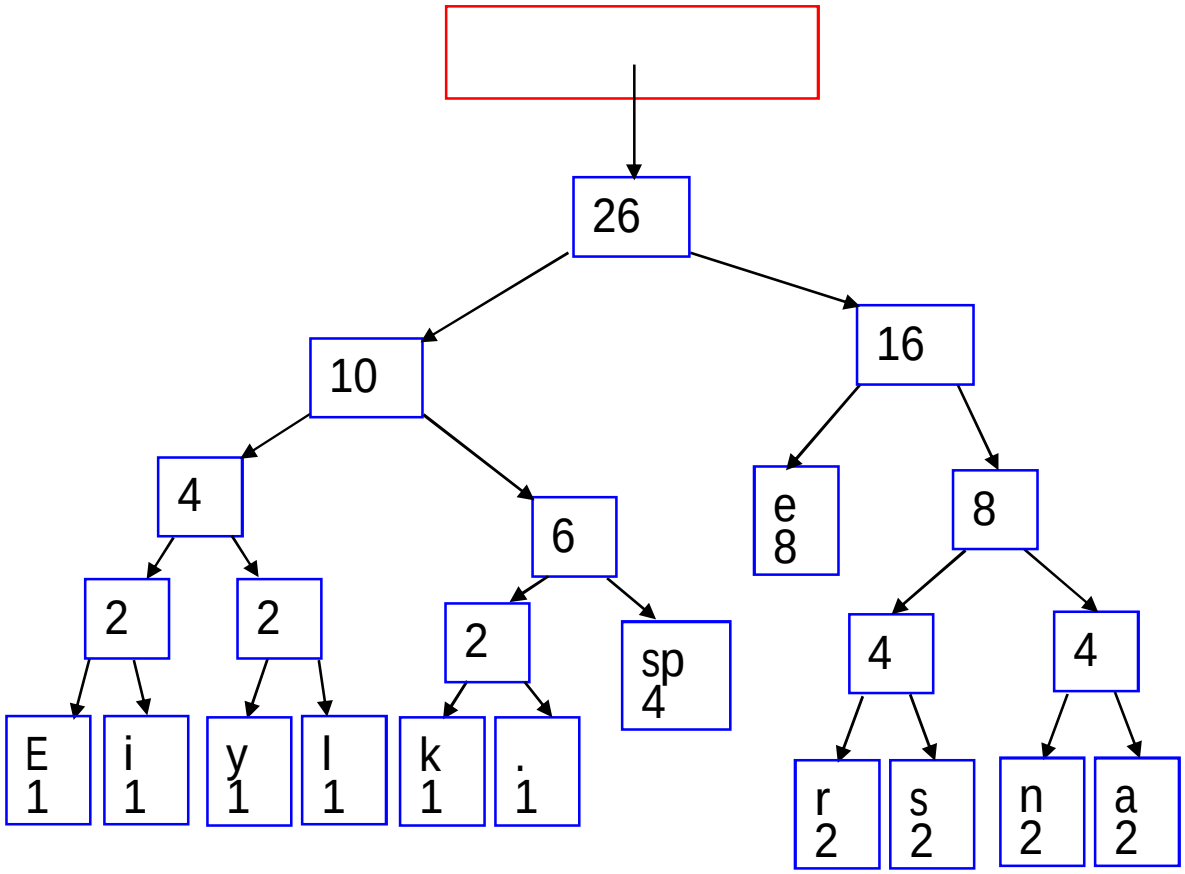
# Building a Tree



# Building a Tree



# Building a Tree



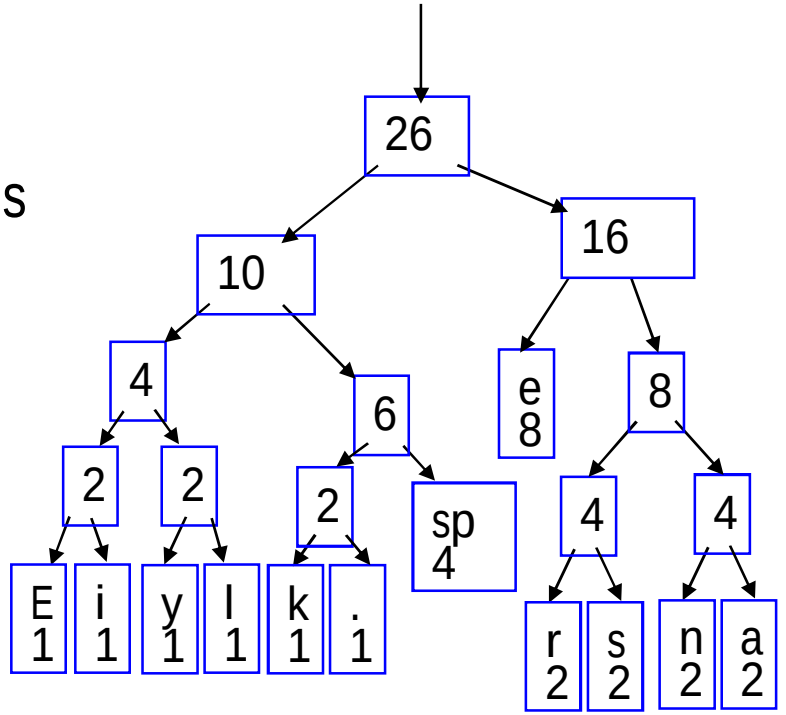

After enqueueing this node there is only one node left in priority queue.

# Building a Tree

Dequeue the single node left in the queue.

This tree contains the new code-words for each character.

**Frequency of root node should equal number of characters in text**.



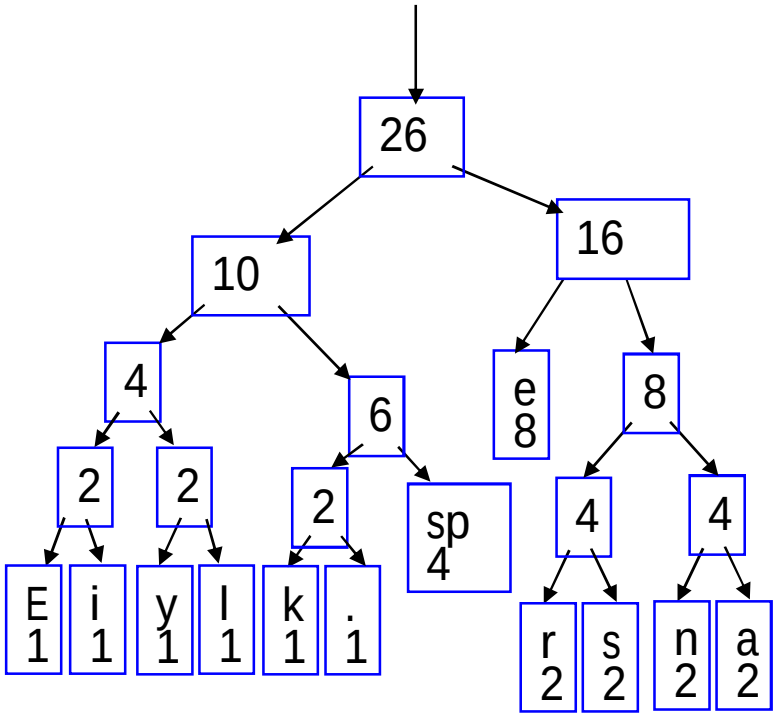

Eerie eyes seen near lake.

26 characters

# Encoding the File

## Traverse Tree for Codes

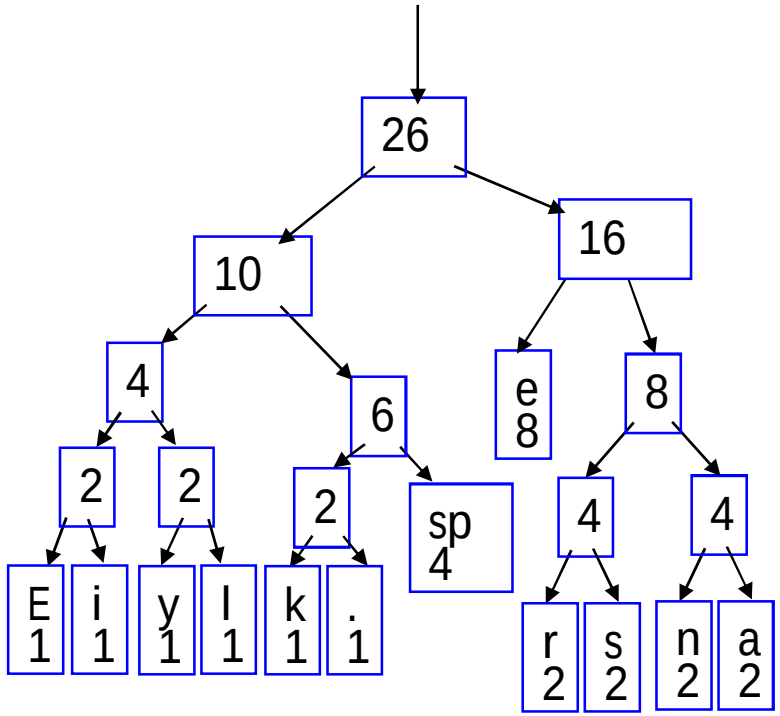
- Perform a traversal of the tree to obtain new code words
- Going left is a 0 going right is a 1
- code word is only completed when a leaf node is reached



# Encoding the File

## Traverse Tree for Codes

| Char | Code |
|---|---|
| E | 0000 |
| i | 0001 |
| y | 0010 |
| l | 0011 |
| k | 0100 |
| . | 0101 |
| space | 011 |
| e | 10 |
| r | 1100 |
| s | 1101 |
| n | 1110 |
| a | 1111 |



# Encoding the File

## Results

- Have we made things any better?
- 73 bits to encode the text
- ASCII would take 8 * 26 = 208 bits

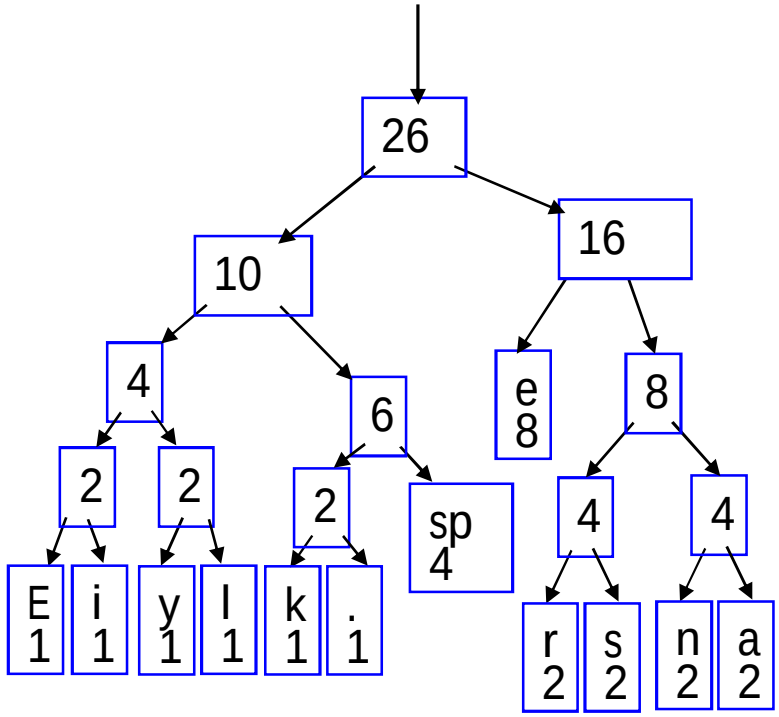
```
0000101100000110011100010101101101001111110101111110001100111111010010010101
```

# Decoding the File

- How does receiver know what the codes are?
- Tree constructed for each text file are then saved 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)
-

# Decoding the File

- Once receiver has tree it scans incoming bit stream
- 0 $\Rightarrow$ go left
- 1 $\Rightarrow$ go right
- Print character




01001100000111011111011111011100011100

krisna sir

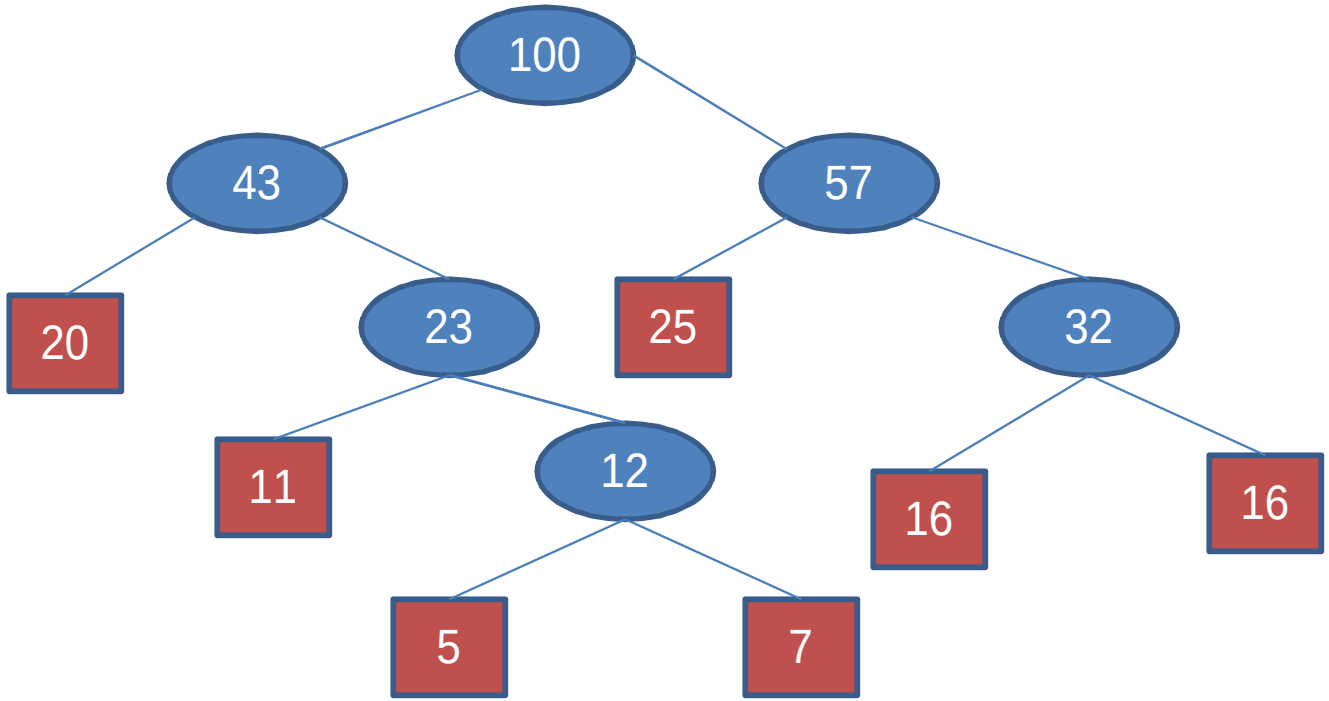
# Important Facts:

1. It is an example of a greedy algorithm: It's called greedy because the two smallest nodes are chosen at each step, and this local decision results in a globally optimal encoding tree.
2. Huffman tree is an optimal tree: there are no other trees with the same characters that use fewer bits to encode the same string.
3. **Huffman codes are prefix free codes (prefix codes) i.e the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.**
4. **Huffman coding is used to assign variable size code for symbol. In huffman coding, less size code is assigned to frequently occurring symbols and greater size code is assign to rarely occurring symbols.**

# Huffman Tree

Let following information is given –

| **Letter** | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Frequency** | 16 | 11 | 7 | 20 | 25 | 5 | 16 |



# Application of Huffman Tree

Huffman tree is used to compress the text(message) if we want to send a message on network.

- JPEG image format uses Huffman algorithm for compression.
- GZIP, PKZIP (winzip etc) and BZIP2,
- Image format PNG also uses huffman coding
- used in Video format MPEG

# Exercise

Draw the huffman tree and find the codes for the following statements –

1. the quick brown fox jumps over the lazy dog
2. Eerie eye seen near the lake.
3. A cat may look at a king
4. Fortune favours the brave

# Problem

In delete operation of BST, we need inorder successor (or predecessor) of a node when the node to be deleted has both left and right child as non-empty. Which of the following is true about inorder successor needed in delete operation?

**(A)** Inorder Successor is always a leaf node

**(B)** Inorder successor is always either a leaf node or a node with empty left child

**(C)** Inorder successor may be an ancestor of the node

**(D)** Inorder successor is always either a leaf node or a node with empty right child

**Answer: (B)**

# Problem

1. A networking company uses a compression technique to encode the message before transmitting over the network. Suppose the message contains the following characters with their frequency:

| character | Frequency |
|---|---|
| a | 5 |
| b | 9 |
| c | 12 |
| d | 13 |
| e | 16 |
| f | 45 |

If the compression technique used is Huffman Coding, how many bits will be saved in the message?

**(A)** 224 **(B)** 800 **(C)** 576 **(D)** 324

**Answer: (C)** Total number of characters in the message = 100. Each character takes 1 byte. So total number of bits needed = 800.

After *Huffman Coding*, the characters can be represented with: f: 0, c: 100, d: 101, a: 1100, b: 1101, e: 111

Total number of bits needed = 224

Hence, number of bits saved = 800 - 224 = 576

# Problem

1. **Postorder traversal of a given binary search tree, T produces the following sequence of keys**
   **10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29**
   **Which one of the following sequences of keys can be the result of an in-order traversal of the tree T?**
   (A) 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95
   (B) 9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29
   (C) 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95
   (D) 95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29

2. **Construct a binary tree for the following preorder and inorder traversals:**
   Inorder sequence: DIBHJEAFLKCGM
   Preorder sequence: ABDIEHJCFKLGM

# Problem

**Level of a node is distance from root to that node. For example, level of root is 1 and levels of left and right children of root is 2. The maximum number of nodes on level i of a binary tree is**

**(A)** 2^(i-1)
**(B)** 2^i
**(C)** 2^(i+1)
**(D)** 2^[(i+1)/2]

**Answer: (A)**

**The height of a binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is:**
**(A)** 2^h -1
**(B)** 2^(h-1) – 1
**(C)** 2^(h+1) -1
**(D)** 2*(h+1)

**Answer: (C)**

# Problem

**In a binary tree with n nodes, every node has an odd number of descendants. Every node is considered to be its own descendant. What is the number of nodes in the tree that have exactly one child?**

(A) 0
(B) 1
(C) (n − 1) /2
(D) n-1

**Answer: (A)**

**The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are**

**(A)** 63 and 6, respectively
**(B)** 64 and 5, respectively
**(C)** 32 and 6, respectively
**(D)** Can not determine

**Answer: (A)**

# Problem

**1.** In a complete k-ary tree, every internal node has exactly k children or no child. The number of leaves in such a tree with I internal nodes is:
**(A)** I*k **(B)** (I – 1) k+ 1 **(C)** I*( k – 1) + 1 **(D)** I*(k – 1)

**Answer: (C)**

2. The number of leaf nodes in a rooted tree of n nodes, with each node having 0 or 3 children is:
**(A)** n/2 **(B)** (n-1)/3 **(C)** (n-1)/2 **(D)** (2n+1)/3

**Answer: (D)**
L = (3-1)I + 1 = 2I + 1Total number of nodes(n) is sum of leaf nodes and internal nodes
n = L + I After solving above two, we get L = (2n+1)/3

# Problem

**1.Which of the following is/are correct inorder traversal sequence(s) of binary search tree(s)?**

**1.** 3, 5, 7, 8, 15, 19, 25 **2.** 5, 8, 9, 12, 10, 15, 25
**3.** 2, 7, 10, 8, 14, 16, 20 **4.** 4, 6, 7, 9, 18, 20, 25
(A) 1 and 4 only (B) 2 and 3 only (C) 2 and 4 only (D) 2 only

**Answer: (A)**
In-order traversal of a BST gives elements in increasing order. So answer A is correct.

2. While inserting the elements 71,65,84,69,67,83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is
(A)65 (B) 67 (C) 69 (D) 83

3. Level of a node is distance from root to that node. For example, level of root is 0 and levels of left and right children of root is 1. The maximum number of nodes on level i of a binary tree (the operator '^' indicates power).
**(A)** 2^(i-1) **(B)** 2^i **(C)** 2^(i+1) **(D)** 2^[(i+1)/2]

# Problem

**1.** There are 8, 15, 13, 14 nodes were there in 4 different trees. Which of them could have formed a full binary tree?

A. 15 B. 8 C.14 d. NONE

**Answer:**

2. Evaluation of postfix 7 8 + 3 2 + / will give :

3 b.4 c. 10 d. none

3. **A binary search tree is generated by inserting in order the following integers:**
**50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24**

The number of the node in the left sub-tree and right sub-tree of the root, respectively, is

a) (4, 7)
b) (7, 4)
c) (8, 3)
d) (3, 8)