CS 2124: DATA STRUCTURES
Spring 2024

- 7th Lecture
- Topics: **Introduction to Trees (Part – II)**

Topics

1. Introduction to Trees
   I. Binary Trees
      i. Types of Binary Trees
   II. Building A Binary Search Tree (BST)
      i. Insert into an empty BST
      ii. Duplicate Removal in BST
   III. Binary Tree Traversal
      i. Preorder Traversal
      ii. In order Traversal
      iii. Post order Traversal
2. Expressions as Trees
3. Building Trees
   I. Binary Trees: Dynamic Nodes
4. Traversal Implementation: Recursive
5. Traversal Implementation: Using Stacks
6. Applications

Duplicates Removal in Array using BST (Application of BST)

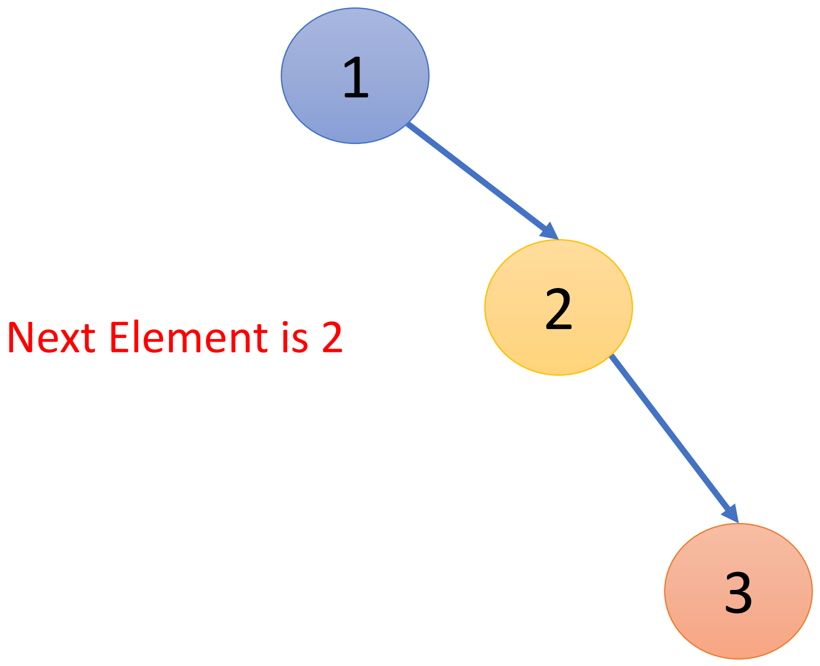
- Input: a[] = {1, 2, 3, 2, 5, 4, 4}

- The duplicates in the array can be removed using Binary Search Tree.

Duplicates Removal in Array using BST (Application of BST)

- Input: a[] = {1, 2, 3, 2, 5, 4, 4}

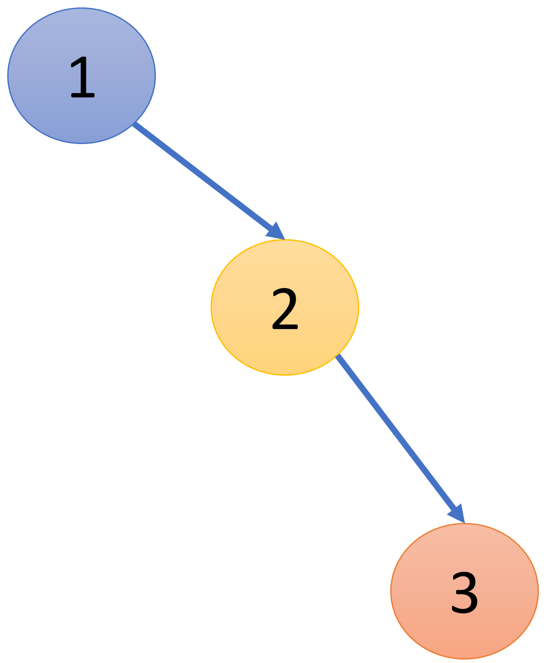
- The duplicates in the array can be removed using Binary Search Tree.



Duplicates Removal in Array using BST (Application of BST)

- Input: a[] = {1, 2, 3, 2, 5, 4, 4}

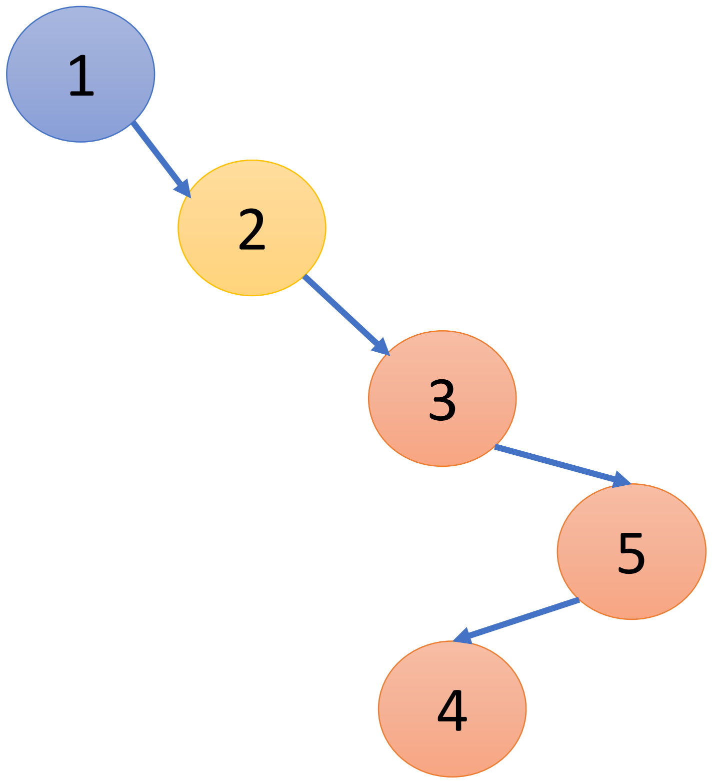
- The duplicates in the array can be removed using Binary Search Tree.



The idea is to create a BST using the array elements with the conditions:

1. First element is taken as the root(parent)
2. Element "less" than root = Left child
3. Element "greater" than root = Right child
4. Since no condition for "equal" exists the duplicates are automatically removed when we form a binary search tree from the array elements.

Duplicates Removal in Array using BST (Application of BST)



- Input: a[] = {1, 2, 3, 2, 5, 4, 4}

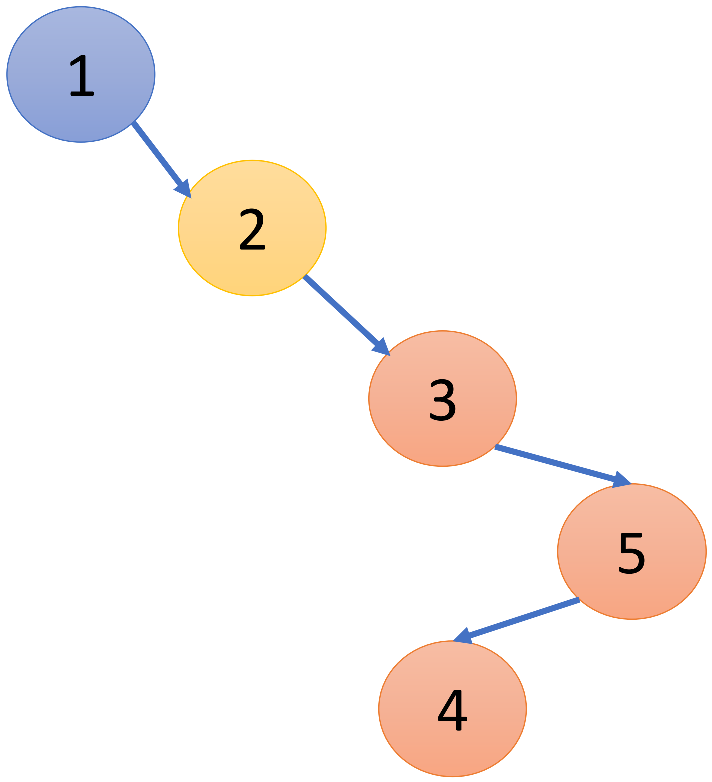
- The duplicates in the array can be removed using Binary Search Tree.

The idea is to create a BST using the array elements with the conditions:

1. First element is taken as the root(parent)
2. Element “less” than root = Left child
3. Element “greater” than root = Right child
4. Since no condition for “equal” exists the duplicates are automatically removed when we form a binary search tree from the array elements.

Duplicates Removal in Array using BST (Application of BST)

- Input: a[] = {1, 2, 3, 2, 5, 4, 4}
- Output: a[] = {1, 2, 3, 4, 5}



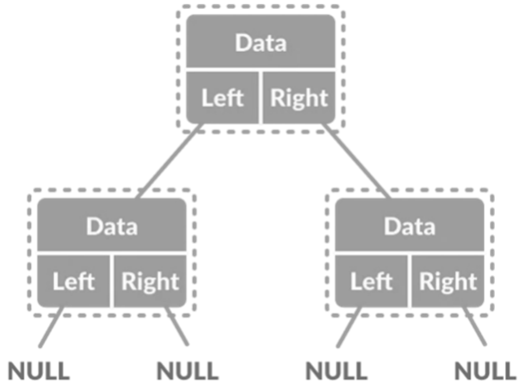
- The duplicates in the array can be removed using Binary Search Tree.

The idea is to create a BST using the array elements with the conditions:

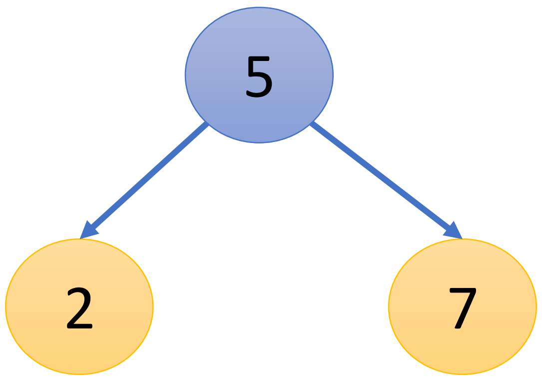
1. First element is taken as the root(parent)
2. Element "less" than root = Left child
3. Element "greater" than root = Right child
4. Since no condition for "equal" exists the duplicates are automatically removed when we form a binary search tree from the array elements.

Duplicates Removal in Array using BST

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  struct Node { // Struct declaration
4      int data;
5      struct Node* left;
6      struct Node* right;
7  };
8  struct Node* newNode(int data)
9  {  // Node creation
10     struct Node* nn
11         = (struct Node*)(malloc(sizeof(struct Node)));
12     nn->data = data;
13     nn->left = NULL;
14     nn->right = NULL;
15     return nn;
16 }
17 struct Node* insert(struct Node* root, int data)
18 {   // Function to insert data in BST
19     if (root == NULL)
20         return newNode(data);
21     else {
22         if (data < root->data)
23             root->left = insert(root->left, data);
24         if (data > root->data)
25             root->right = insert(root->right, data);
26         return root;
27     }
```



Root = 5



Line 22: 2 (Data) < 5(Root->Data)

Line 24: 7 (Data) > 5(Root->Data)

Duplicates Removal in Array using BST

```c
#include <stdio.h>
#include <stdlib.h>
struct Node { // Struct declaration
    int data;
    struct Node* left;
    struct Node* right;
};
struct Node* newNode(int data)
{  // Node creation
    struct Node* nn
        = (struct Node*)(malloc(sizeof(struct Node)));
    nn->data = data;
    nn->left = NULL;
    nn->right = NULL;
    return nn;
}
struct Node* insert(struct Node* root, int data)
{   // Function to insert data in BST
    if (root == NULL)
        return newNode(data);
    else {
        if (data < root->data)
            root->left = insert(root->left, data);
        if (data > root->data)
            root->right = insert(root->right, data);
        return root;
    }
}
void inOrder(struct Node* root)
{
if (root == NULL) // InOrder function to display value
    return;       // of array in sorted order
else {
        inOrder(root->left);
        printf(" %d, ", root->data);
        inOrder(root->right);
    }
}
int main()
{
    int arr[] = { 2, 0, 2, 3, 2, 0, 2, 3 };
    // Finding size of array arr[]
    int n = sizeof(arr) / sizeof(arr[0]);
    struct Node* root = NULL;
     printf("Initial Tree:");
for (int i = 0; i < n; i++) {
    printf(" %d, ", arr[i]);
        // Insert element of arr[] in BST
        root = insert(root, arr[i]);
} // Inorder Traversal to print nodes of Tree
    printf("\nInOrder (Duplicates removed):");
    inOrder(root);
    return 0;
}
```

In-Order, Pre-Order and Post-Order traversal will be discussed in upcoming slides

Duplicates Removal in Array using BST

```c
#include <stdio.h>
#include <stdlib.h>
struct Node { // Struct declaration
    int data;
    struct Node* left;
    struct Node* right;
};
struct Node* newNode(int data)
{  // Node creation
    struct Node* nn
        = (struct Node*)(malloc(sizeof(struct Node)));
    nn->data = data;
    nn->left = NULL;
    nn->right = NULL;
    return nn;
}
struct Node* insert(struct Node* root, int data)
{   // Function to insert data in BST
    if (root == NULL)
        return newNode(data);
    else {
        if (data < root->data)
            root->left = insert(root->left, data);
        if (data > root->data)
            root->right = insert(root->right, data);
        return root;
    }
}
void inOrder(struct Node* root)
{
if (root == NULL) // InOrder function to display value
    return;       // of array in sorted order
else {
        inOrder(root->left);
        printf(" %d, ", root->data);
        inOrder(root->right);
    }
}
int main()
{
    int arr[] = { 2, 0, 2, 3, 2, 0, 2, 3 };
    // Finding size of array arr[]
    int n = sizeof(arr) / sizeof(arr[0]);
    struct Node* root = NULL;
     printf("Initial Tree:");
for (int i = 0; i < n; i++) {
    printf(" %d, ", arr[i]);
        // Insert element of arr[] in BST
        root = insert(root, arr[i]);
} // Inorder Traversal to print nodes of Tree
    printf("\nInOrder (Duplicates removed):");
    inOrder(root);
    return 0;
}
```

Output ?

Binary Tree vs Binary Search Tree (BST)

	BINARY TREE	**BINARY SEARCH TREE**
Definition	BINARY TREE is a nonlinear data structure where each node can have at most two child nodes.	BINARY SEARCH TREE is a node based binary tree that further has right and left subtree that too are binary search tree.
Types	Full binary tree, Complete binary tree, Extended Binary tree and more	AVL tree, Splay Tree, T-trees and more
Structure	In BINARY TREE there is no ordering in terms of how the nodes are arranged	In BINARY SEARCH TREE the left subtree has elements less than the nodes element and the right subtree has elements greater than the nodes element.
Data Representation	Data Representation is carried out in a hierarchical format.	Data Representation is carried out in the ordered format.
Duplicate Values	Binary trees allow duplicate values.	Binary Search Tree does not allow duplicate values.

Tree Traversals

A **traversal** is an order for visiting all the nodes of a tree

- ***Pre-order***: root -> left subtree -> right subtree

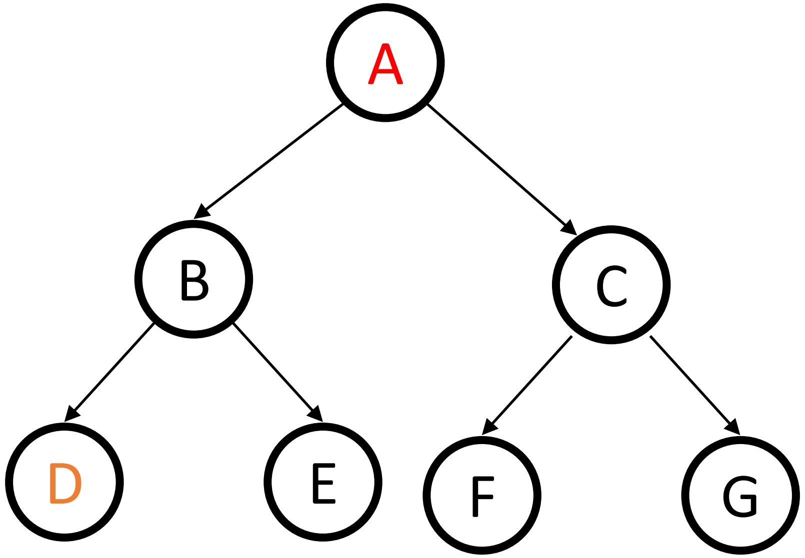
  A → B → D → E → C → F → G

- ***In-order***: left subtree -> root -> right subtree

  D → B → E → A → F → C → G

- ***Post-order***: left subtree -> right subtree -> root

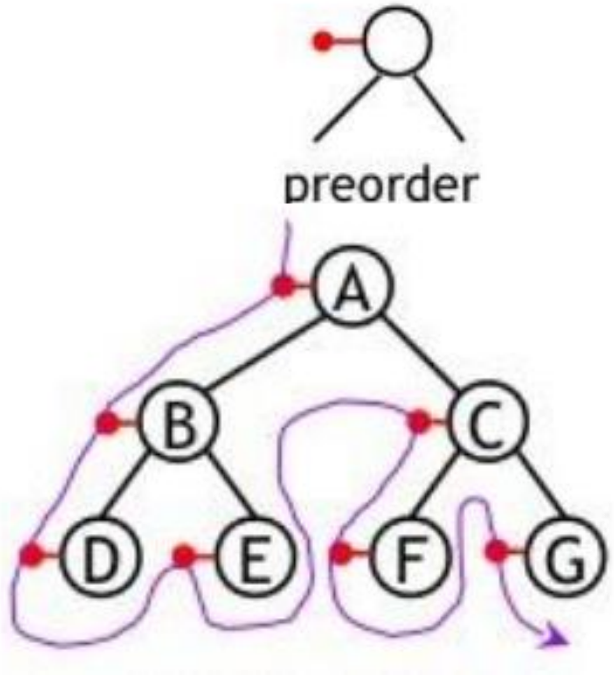
  D → E → B → F → G → C → A



Tree Traversals (A trick to remember)

Pre-order

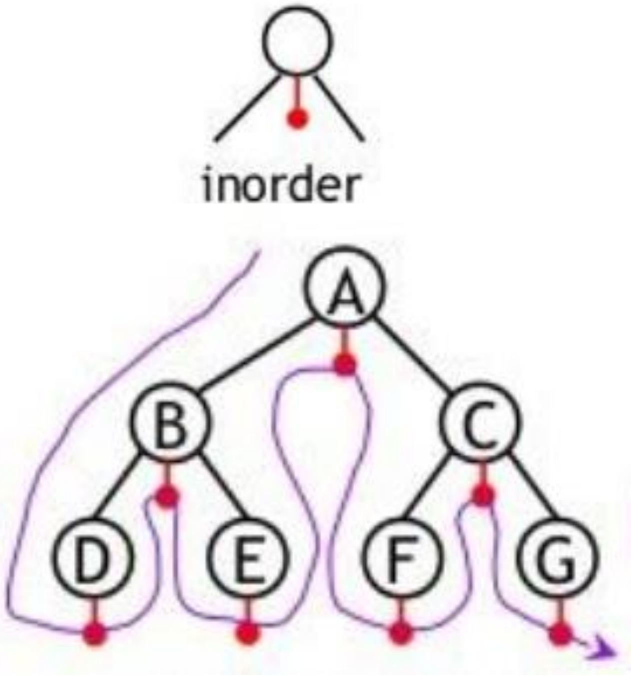
1. Root
2. Left Subtree
3. Right Subtree



A → B → D → E → C → F → G

In-order

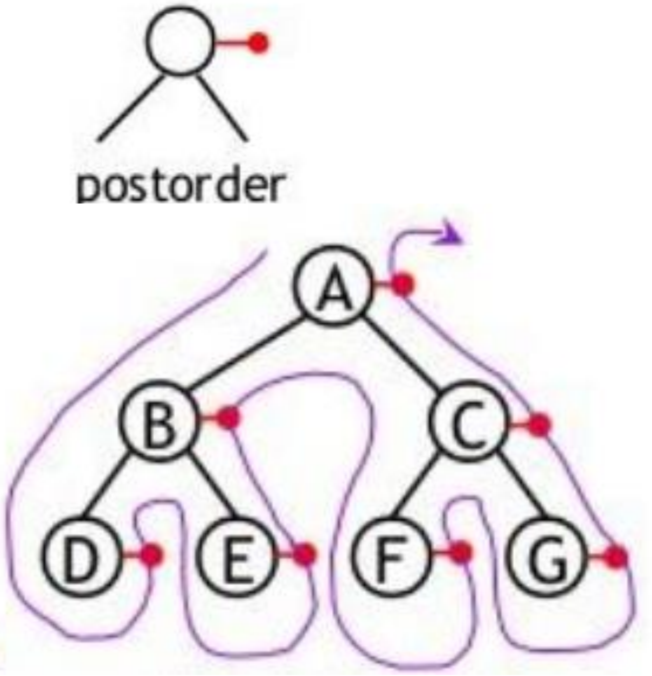
1. Left Subtree
2. Root
3. Right Subtree



D → B → E → A → F → C → G

Post-order

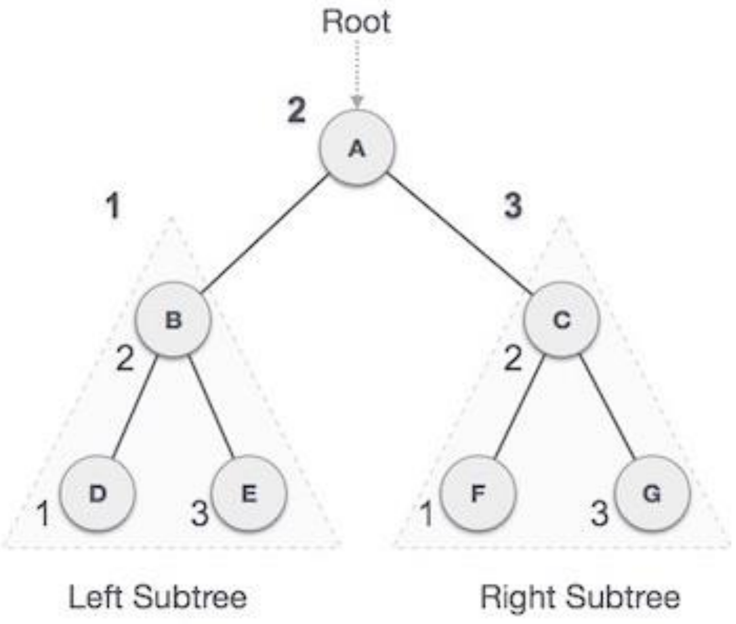
1. Left Subtree
2. Right Subtree
3. Root



D → E → B → F → G → C → A

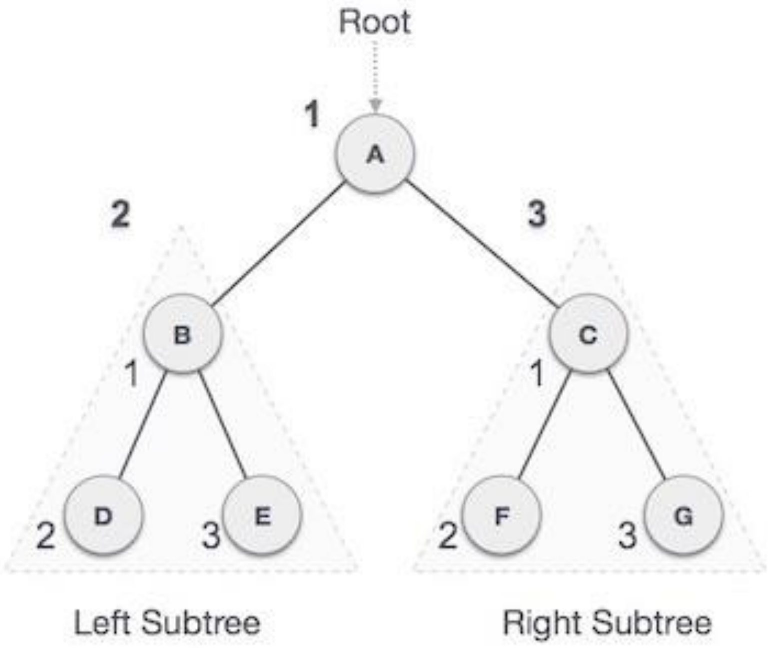
Tree Traversals (Another trick to remember the traversal order)

In-order Traversal



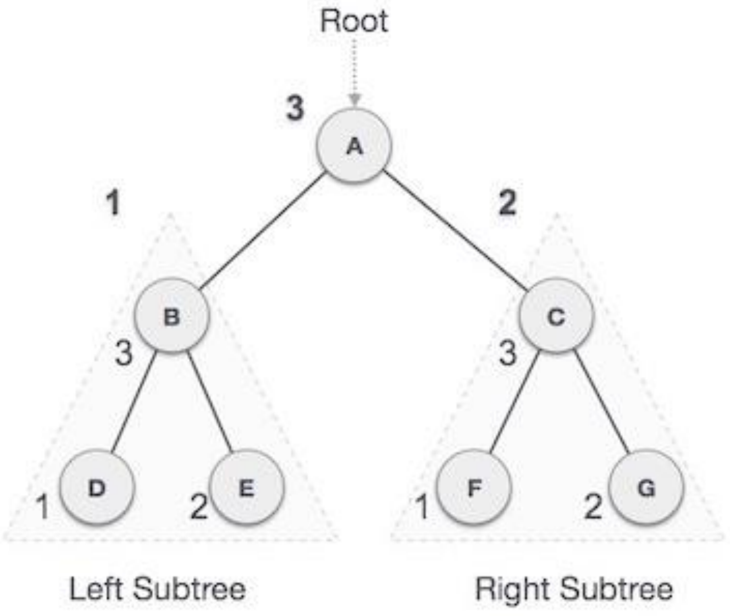
D → B → E → A → F → C → G

Pre-order Traversal



A → B → D → E → C → F → G

Post-order Traversal



D → E → B → F → G → C → A

Tree Traversals: Practice

Which one makes sense for evaluating this *expression tree?*

- ***Pre-order***: root, left subtree, right subtree

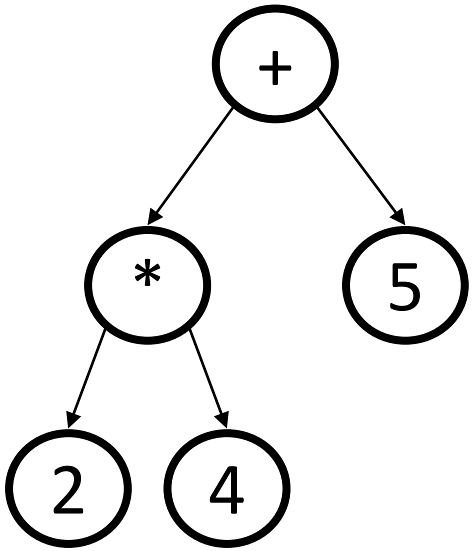
  + * 2 4 5

- ***In-order***: left subtree, root, right subtree

  2 * 4 + 5

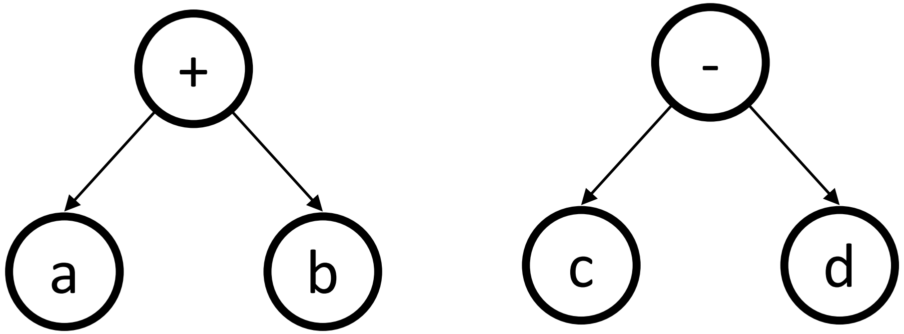
- ***Post-order***: left subtree, right subtree, root

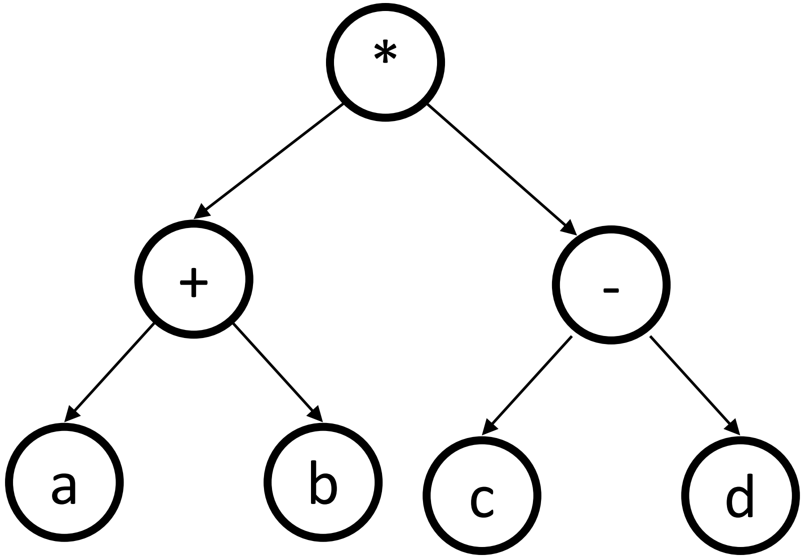
  2 4 * 5 +



Expressions as Trees

- We can also divide the tree into sub-trees and then traverse them
- (a+b)*(c-d)

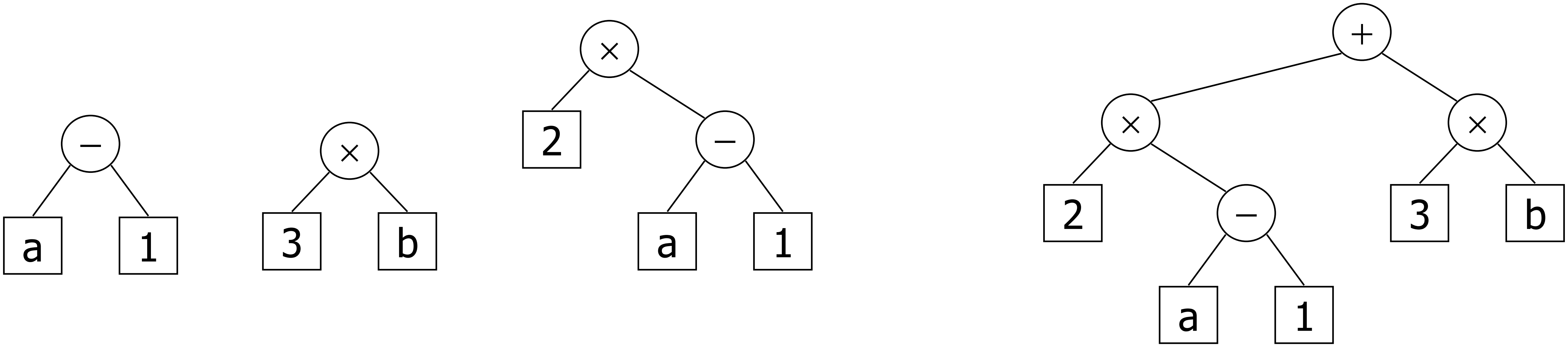




- ***Pre-order***: root, left subtree, right subtree * + a b – c d
- ***In-order***: left subtree, root, right subtree a + b * c - d
- ***Post-order***: left subtree, right subtree, root a b + c d - *

Expressions as Trees

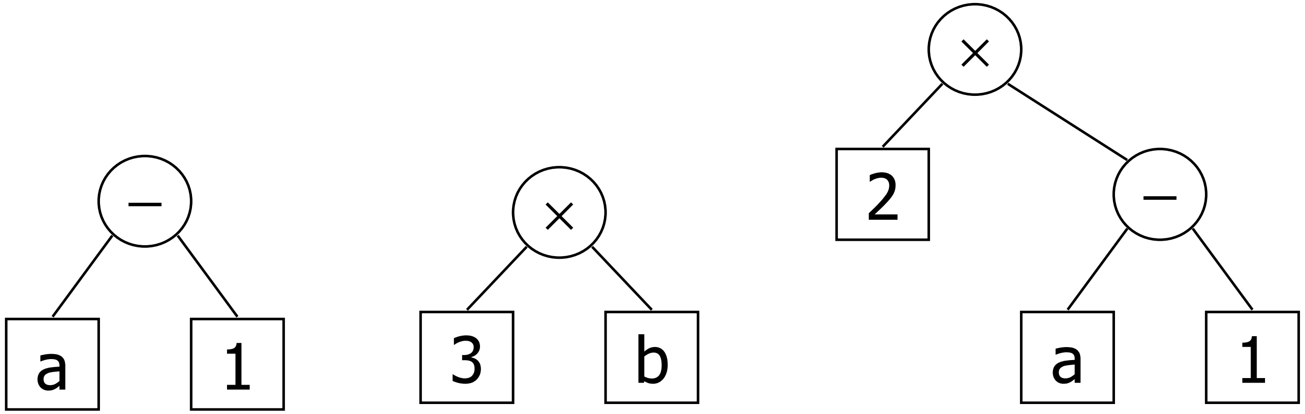
- $(2 \times (a - 1) + (3 \times b))$

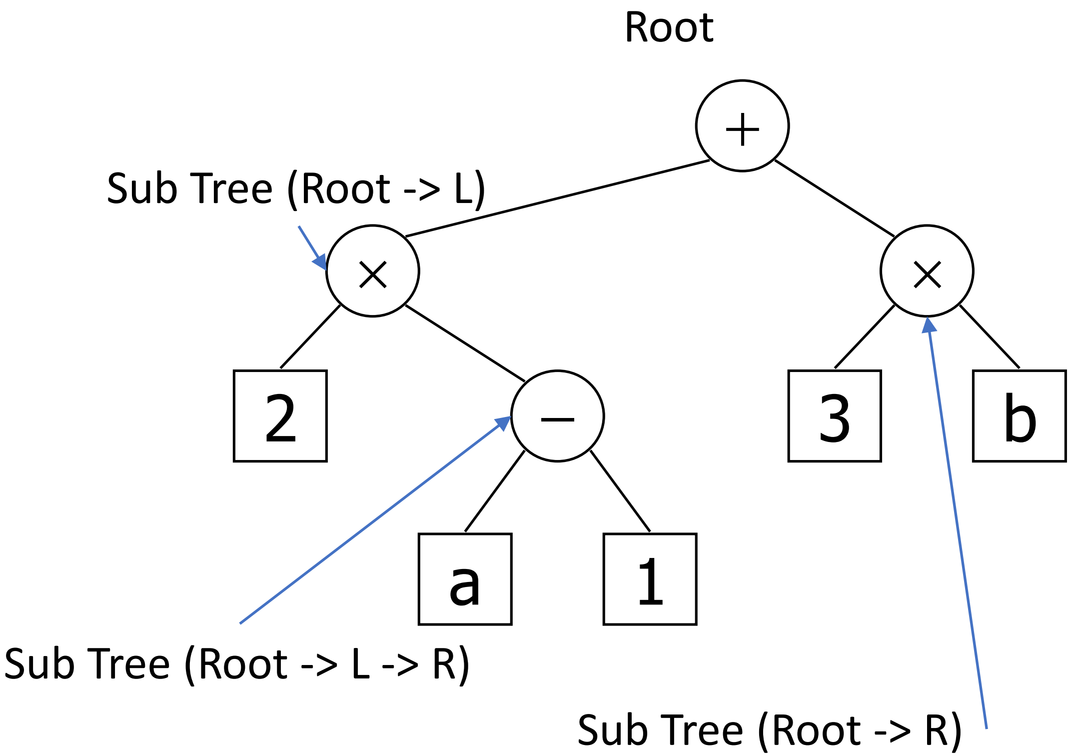


- ***Pre-order***: root, left subtree, right subtree
- ***In-order***: left subtree, root, right subtree
- ***Post-order***: left subtree, right subtree, root

Expressions as Trees

- $(2 \times (a - 1) + (3 \times b))$

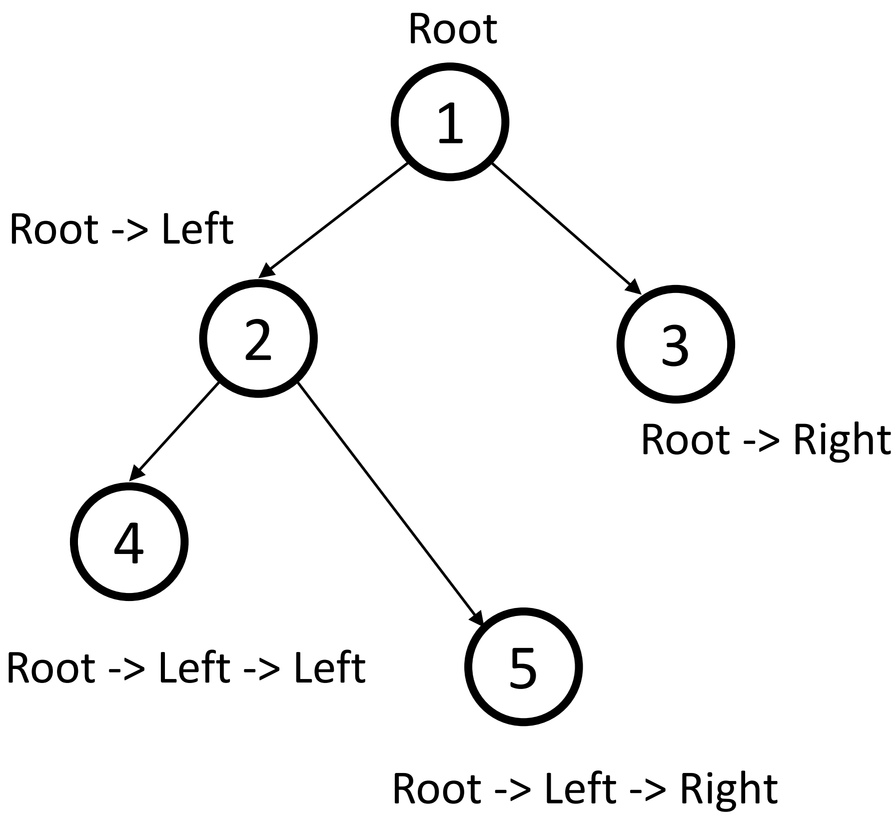




- ***Pre-order***: root, left subtree, right subtree
- ***In-order***: left subtree, root, right subtree
- ***Post-order***: left subtree, right subtree, root

Postfix : ?
Prefix : ?

Tree Traversals



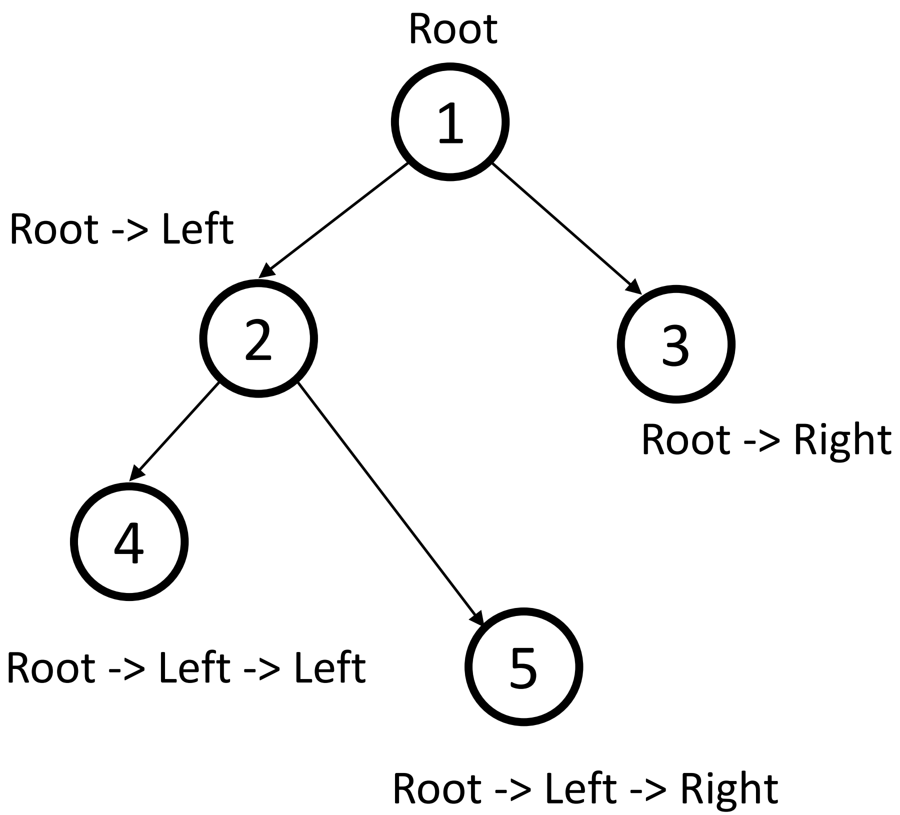
In-order

1. Left Subtree,
2. Root,
3. Right Subtree

```c
// Given a binary tree, print its nodes in inorder
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    // First recur on left child
    printInorder(node->left);
    // Then print the data of node
    printf("%d ", node->data);
    // Now recur on right child
    printInorder(node->right);
}
int main()
{
    struct node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    // Function call
    printf("Inorder traversal of binary tree is \n");
    printInorder(root);
}
```

Output?

Tree Traversals



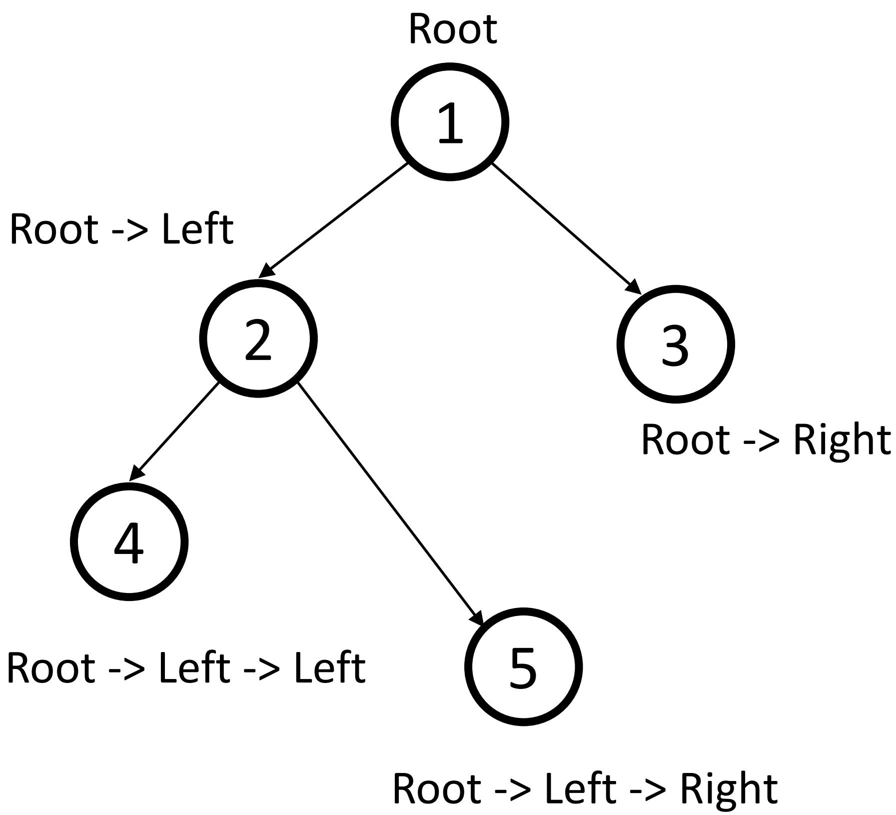
Pre-order

1. Root,
2. Left Subtree,
3. Right Subtree

```c
void printPreorder(struct node* node)
{
    if (node == NULL)
        return;
    // First print data of node
    printf("%d ", node->data);
    // Then recur on left subtree
    printPreorder(node->left);
    // Now recur on right subtree
    printPreorder(node->right);
}
int main()
{
    struct node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    // Function call
    printf("Preorder traversal of binary tree is \n");
    printPreorder(root);
}
```

Output?

Tree Traversals



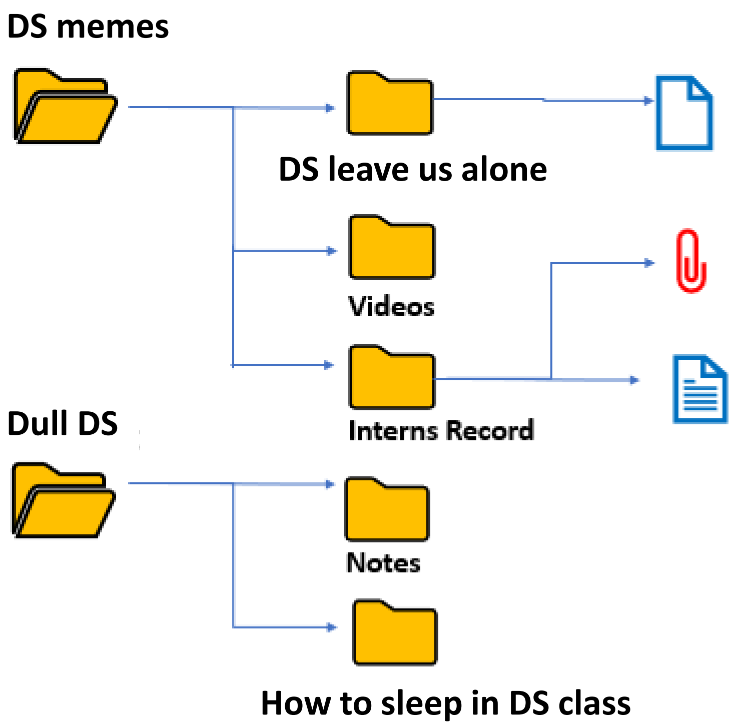
Post-order

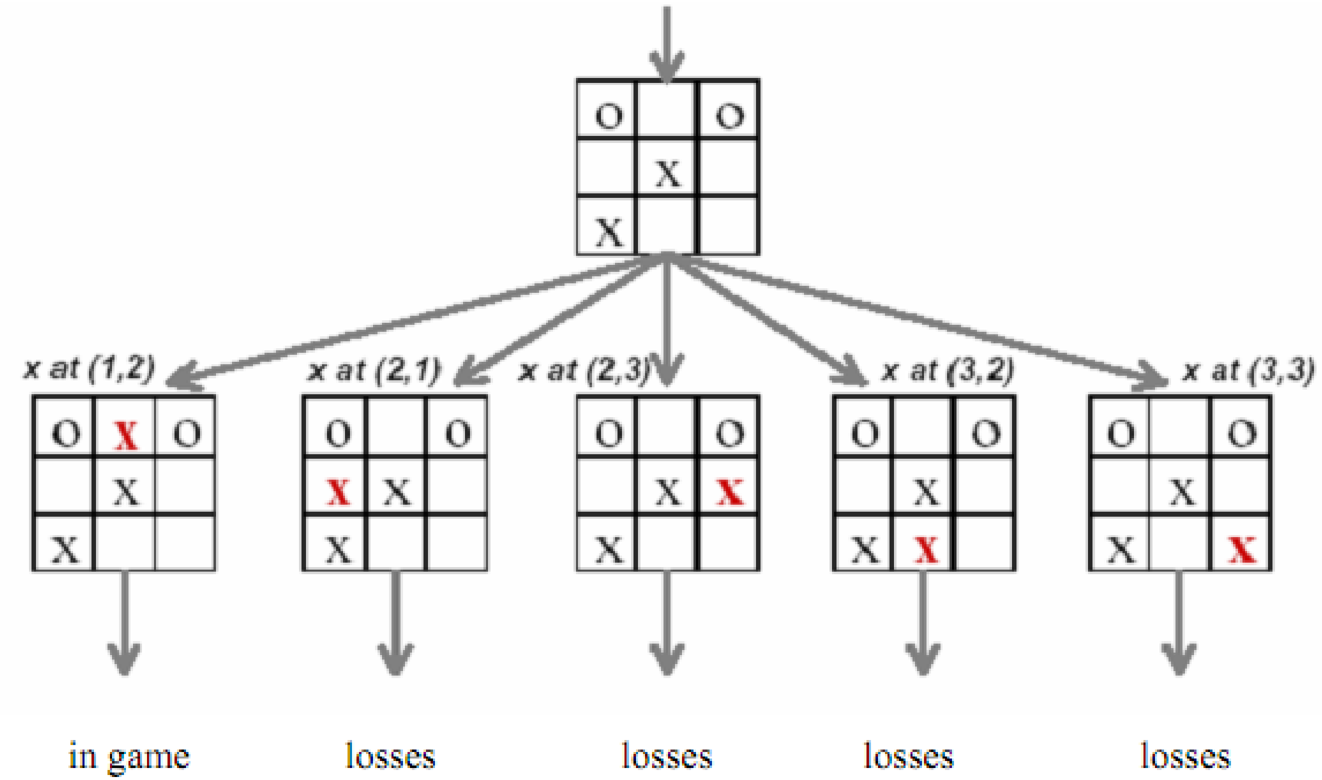
1. Left Subtree,
2. Right Subtree,
3. Root

```c
void printPostorder(struct node* node)
{
    if (node == NULL)
        return;
    // First recur on left subtree
    printPostorder(node->left);
    // Then recur on right subtree
    printPostorder(node->right);
    // Now deal with the node
    printf("%d ", node->data);
}
int main()
{
    struct node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    // Function call
    printf("Postorder traversal of binary tree is \n");
    printPostorder(root);
}
```

Output?

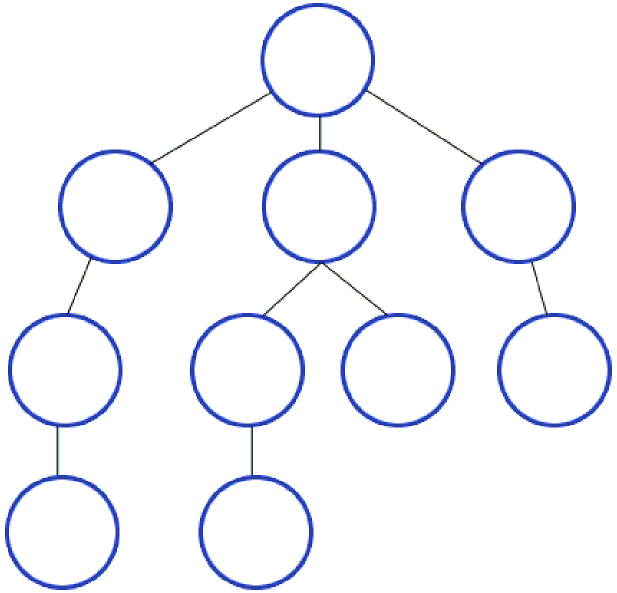
Applications

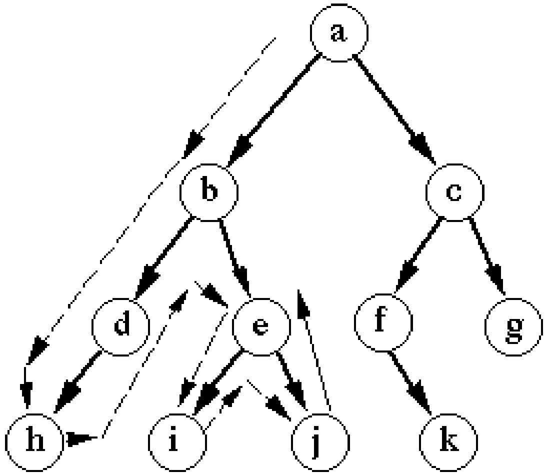




Depth-first search (DFS)

- DFS goes through a graph as far as possible in one direction before backtracking to other nodes. DFS is similar to the pre-order tree traversal, but you need to make sure you don't get stuck in a loop. To do this, you'll need to keep track of which Nodes have been visited.

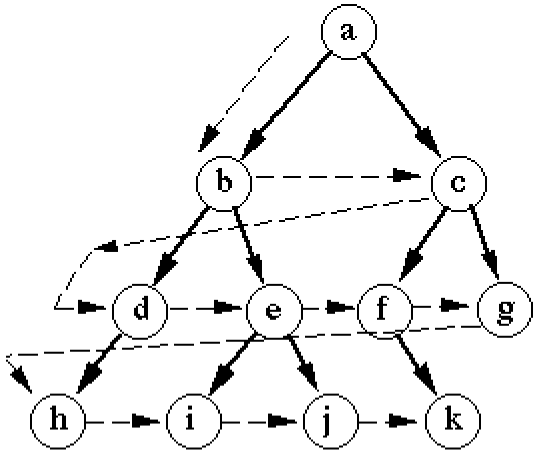




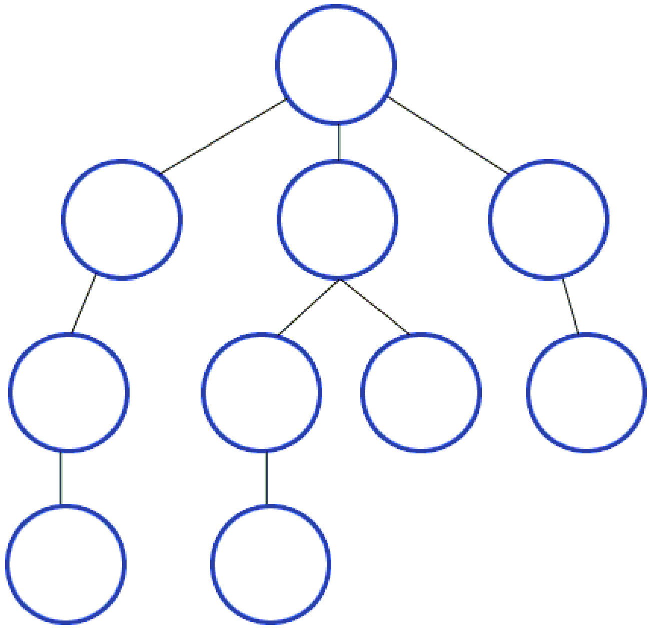
Depth-first search

Breadth-first search (BFS)

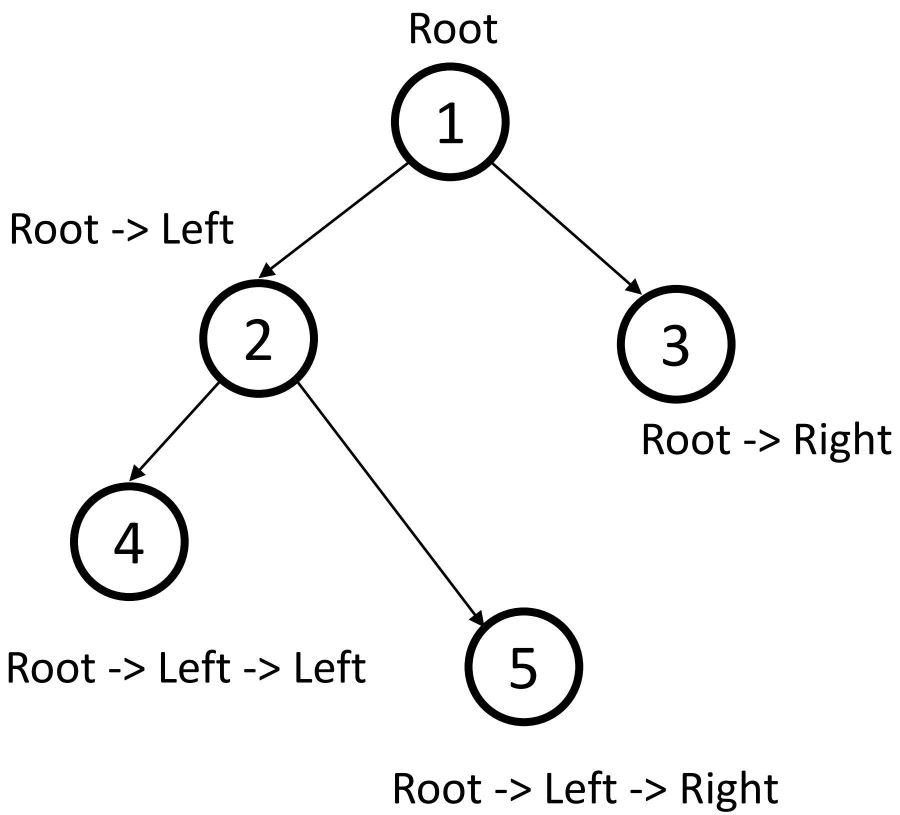
- BFS is a graph traversal algorithm that explores nodes in the order of their distance from the roots, where distance is defined as the minimum path length from a root to the node.



Breadth-first search



Tree Traversals (Using Stacks)



- End goal is to print the Tree in In-order
- 4 2 5 1 3

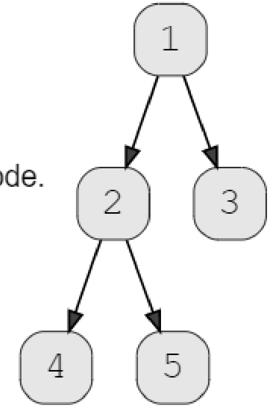
In-order

1. Left Subtree,
2. Root,
3. Right Subtree

1

The root is made to be the current node.
The stack is empty, for now.

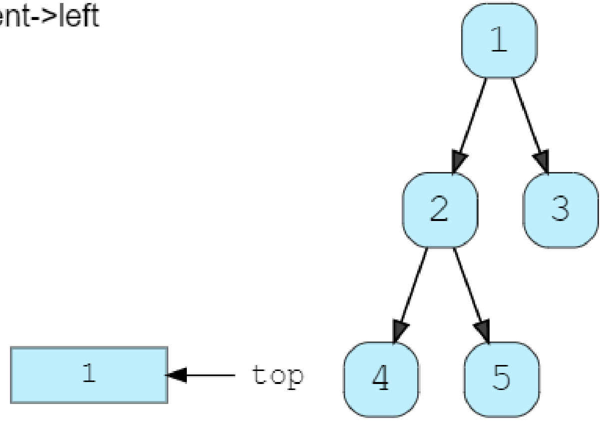
current = 1



Push the current node to the stack and and set
current = current->left

2

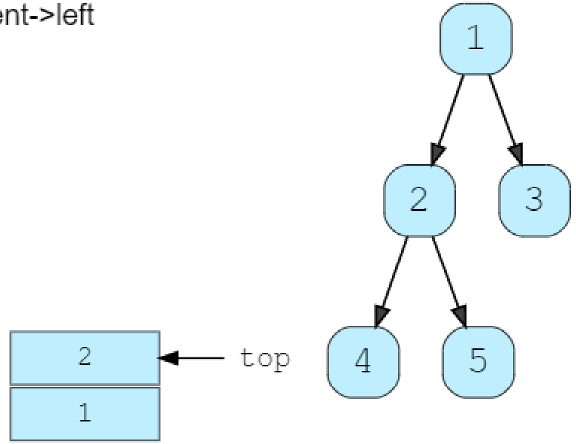
current = 1



Push the current node to the stack and and set
current = current->left

3

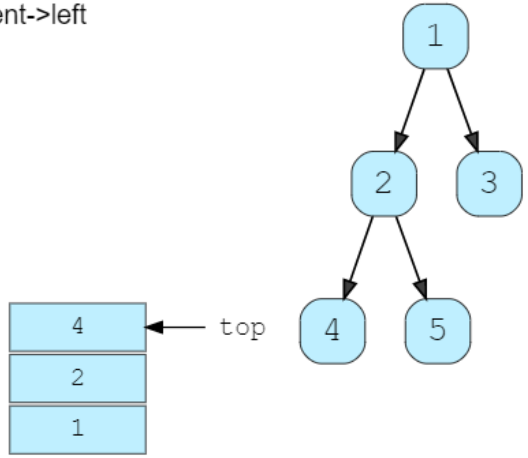
current = 2



Push the current node to the stack and and set
current = current->left

4

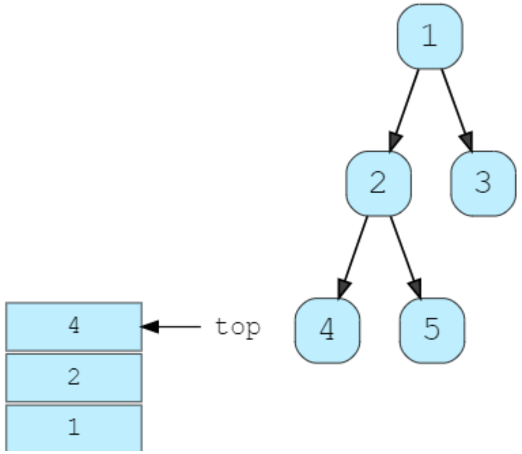
current = 4



Since current = NULL, we need to pop from the stack.

5

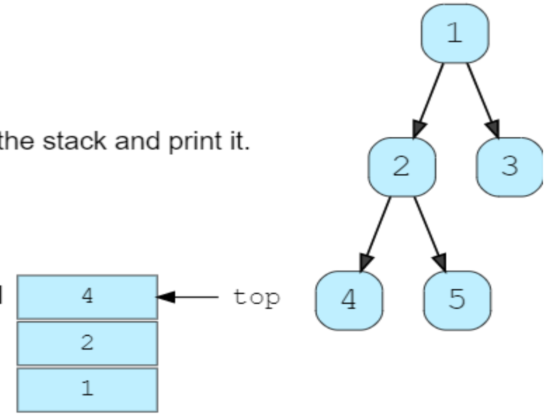
current = null



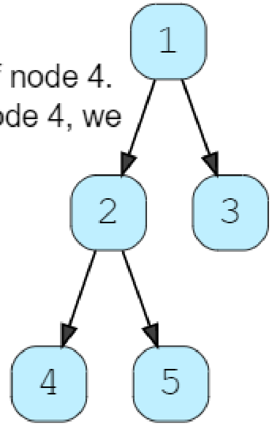
6

Pop '4' from the stack and print it.

current = null

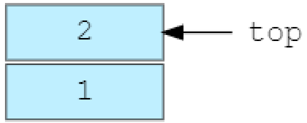


7 Pop '4' from the stack and print it.
Current is set to the node to the right of node 4.
Since there is no node to the right of node 4, we
continue to pop from the stack.

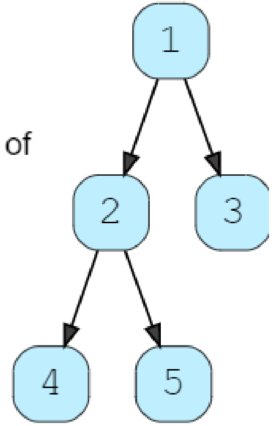


4

current = null

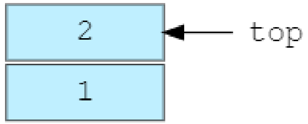


8 Pop '2' from the stack and print it.
Set current to be the node to the right of
node 2.

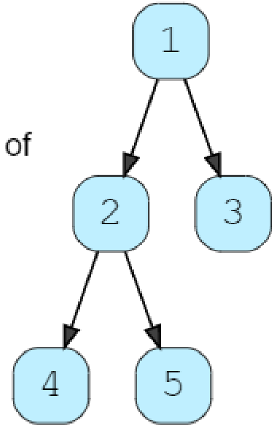


4, 2

current = null

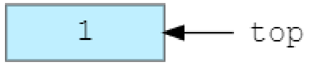


9 Pop '2' from the stack and print it.
Set current to be the node to the right of
node 2.



4, 2

current = 5

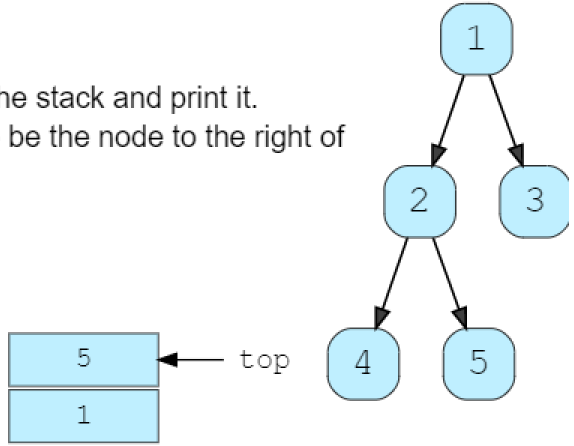


10

Pop '2' from the stack and print it.
Set current to be the node to the right of node 2.

4, 2

current = null

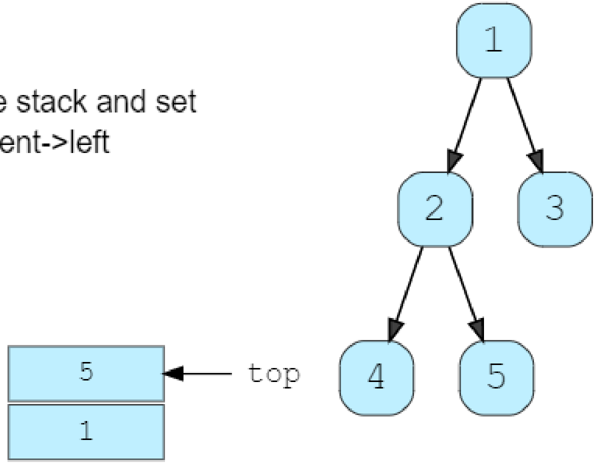


11

Push '5' to the stack and set current = current->left

4, 2

current = null



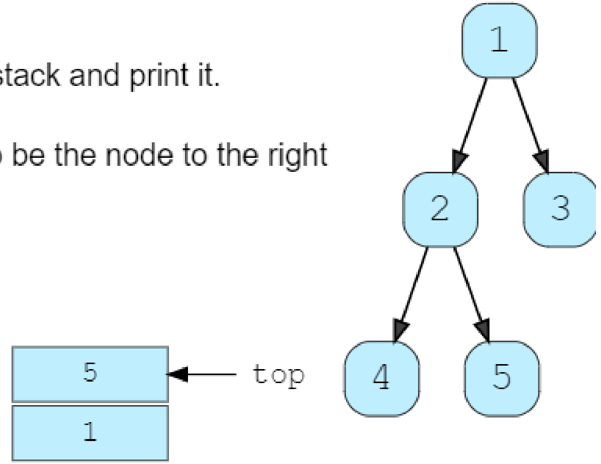
12

Pop '5' from stack and print it.

Set current to be the node to the right of node '5'.

4, 2, 5

current = null



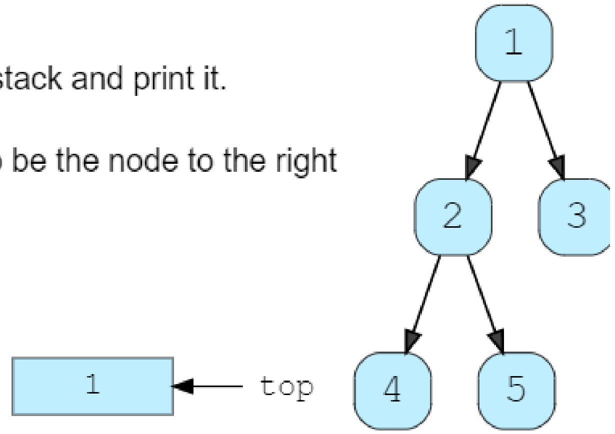
13

Pop '5' from stack and print it.

Set current to be the node to the right of node '5'.

4, 2, 5

current = null



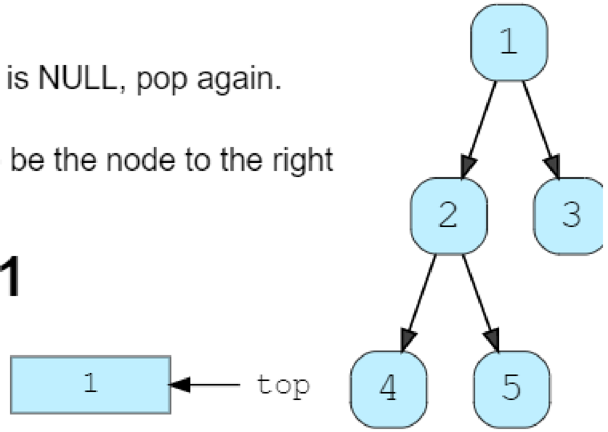
14

Since current is NULL, pop again.

Set current to be the node to the right of node '1'.

4, 2, 5, 1

current = null



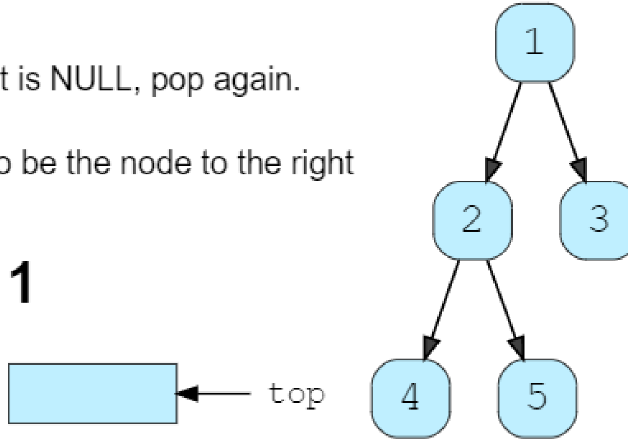
15

Since current is NULL, pop again.

Set current to be the node to the right of node '1'.

4, 2, 5, 1

current = 3



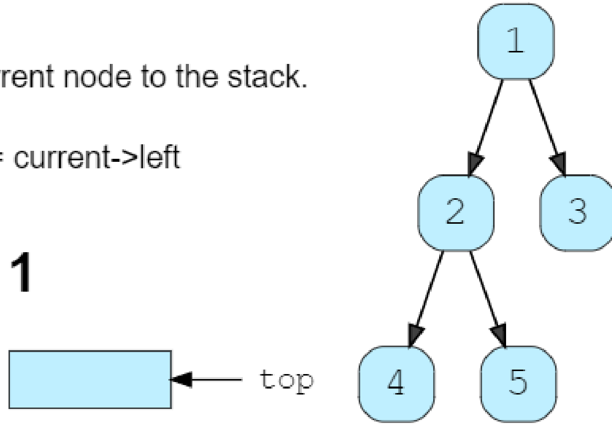
16

Push the current node to the stack.

Set current = current->left

4, 2, 5, 1

current = 3



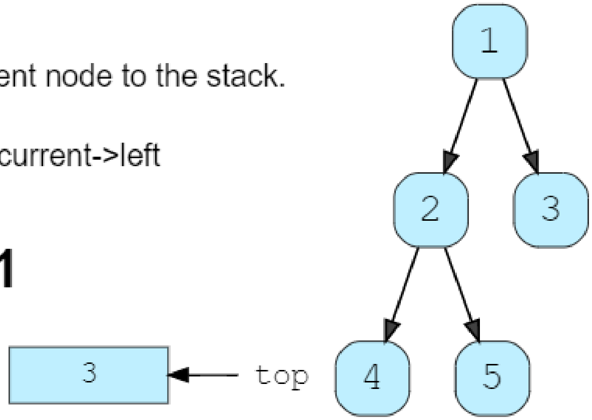
17

Push the current node to the stack.

Set current = current->left

4, 2, 5, 1

current = null



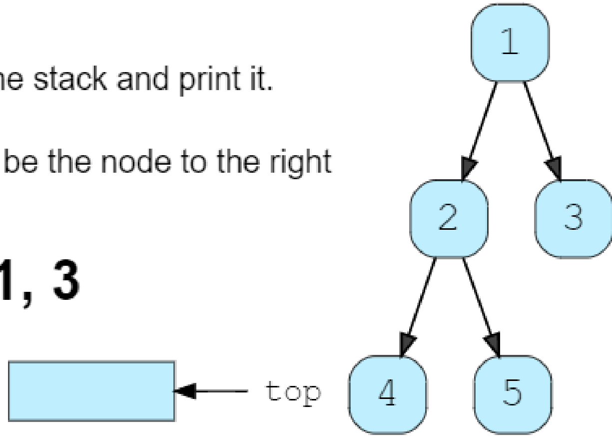
18

Pop '3' from the stack and print it.

Set current to be the node to the right of node '3'.

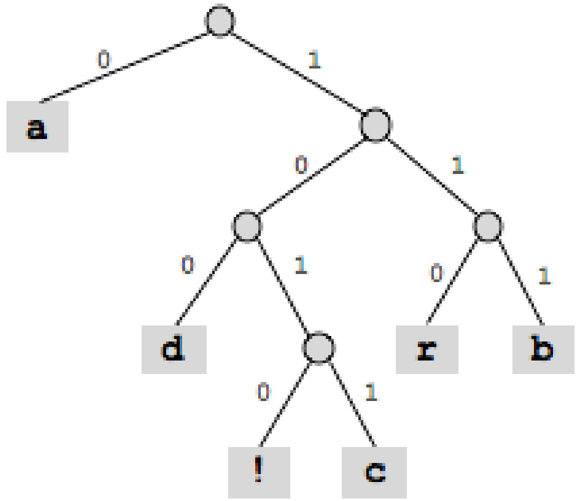
4, 2, 5, 1, 3

current = null



Applications (Huffman Encoding)

- The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters (i.e. more bits for rare letters, and fewer bits for common letters).
- The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character.
- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

Applications (Huffman Encoding)

13-character string "go go gophers" requires 13 * 8 = 104 bits

Character	ASCII code	8-bit binary value
Space	32	00100000
e	101	01100101
g	103	01100111
h	104	01101000
o	111	01101111
p	112	01110000
r	114	01110010
s	115	01110011

Table-1

0	0
0	1
1	0
1	1

8 bits = one character

Two Bits can represent 4 values

Applications (Huffman Encoding)

13-character string "go go gophers" requires 13 * 8 = 104 bits

Character	ASCII code	8-bit binary value
Space	32	00100000
e	101	01100101
g	103	01100111
h	104	01101000
o	111	01101111
p	112	01110000
r	114	01110010
s	115	01110011

Table-1

8 bits = one character

Since there are only 8 different characters in "go go gophers", it is possible to use only 3-bits to encode the 8 different characters.

1	g	000
2	o	001
3	p	010
4	h	011
5	e	100
6	r	101
7	s	110
8	\space	111

Applications (Huffman Encoding)

13-character string "go go gophers" requires 13 * 8 = 104 bits

Character	ASCII code	8-bit binary value
Space	32	00100000
e	101	01100101
g	103	01100111
h	104	01101000
o	111	01101111
p	112	01110000
r	114	01110010
s	115	01110011

Table-1

8 bits = one character

Since there are only 8 different characters in "go go gophers", it is possible to use only 3-bits to encode the 8 different characters.

Character	Code Value	3-bit binary value
g	0	000
o	1	001
p	2	010
h	3	011
e	4	100
r	5	101
s	6	110
Space	7	111

Table-2

13-character string "go go gophers" requires 13 * 3 = 39 bits

"go go gophers" would be encoded as:
000 001 111 000 001 111 000 001 010 011 100 101 110

Applications (Huffman Encoding)

Character	Code	Frequency	Total Bits
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	9
T	100	4	12
P	101	13	39
Newline	110	1	3

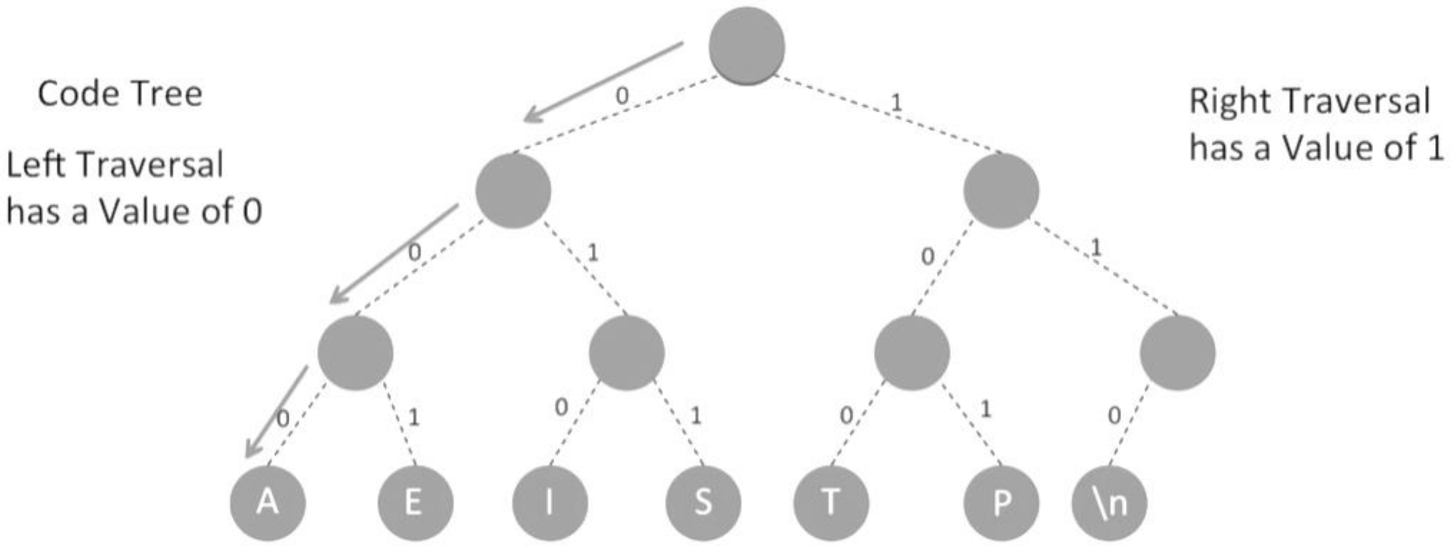
Code bit * Frequency = Total Bits = 174

Huffman Tree (Fix Bit Representation)

Applications (Huffman Encoding)

Character	Code	Frequency	Total Bits
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	9
T	100	4	12
P	101	13	39
Newline	110	1	3

Code bit * Frequency = Total Bits = 174



Huffman Tree (Fix Bit Representation)

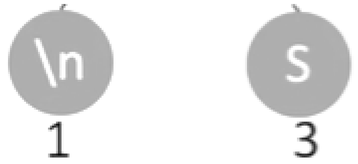
But we want to further reduce the number of bits i.e. less then 174 bits

Huffman Tree (Variable Bit Representation) to reduce the bits

Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1

Step 1: Take the 2 chars with the lowest frequency

Step: 1



Step 1: Take the 2 chars with the lowest frequency

Step 2: Make a 2 leaf node tree from them

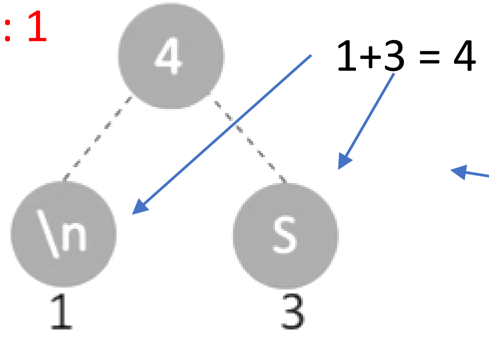
Step: 1

Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1

Step 1: Take the 2 chars with the lowest frequency

Step 2: Make a 2 leaf node tree from them

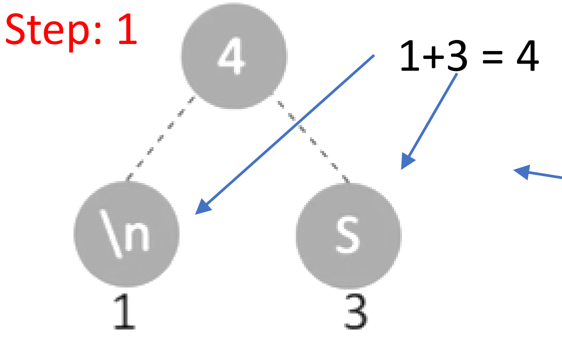
Step: 1



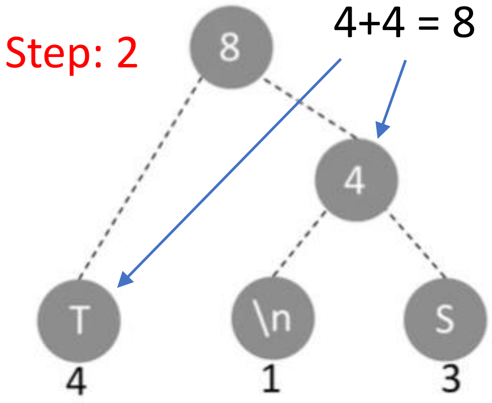
Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1

Step 1: Take the 2 chars with the lowest frequency

Step 2: Make a 2 leaf node tree from them



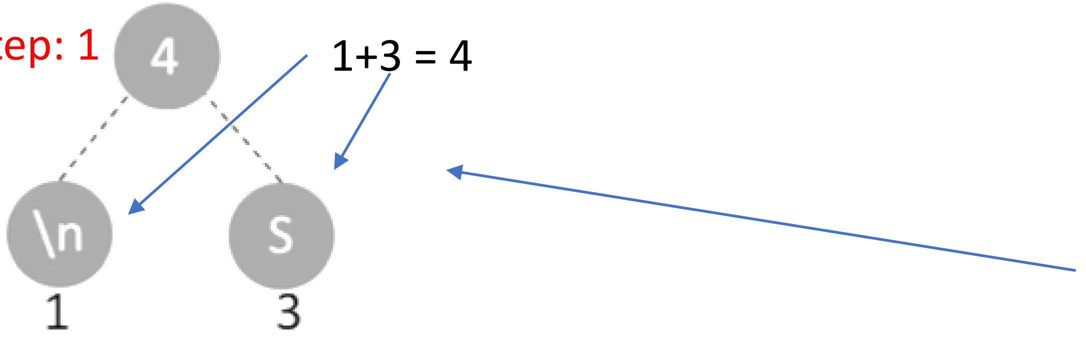
Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1



Step 1: Take the 2 chars with the lowest frequency

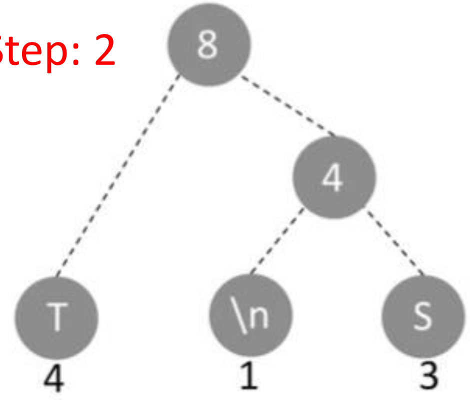
Step 2: Make a 2 leaf node tree from them

Step: 1



Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1

Step: 2

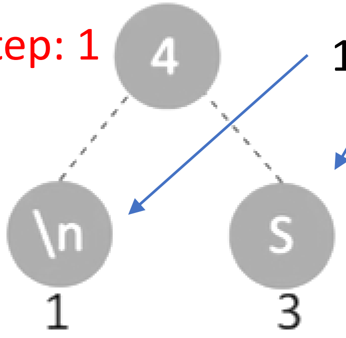


Step: 3

Step 1: Take the 2 chars with the lowest frequency

Step 2: Make a 2 leaf node tree from them

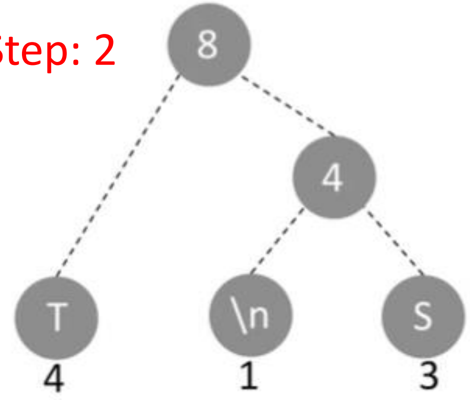
Step: 1



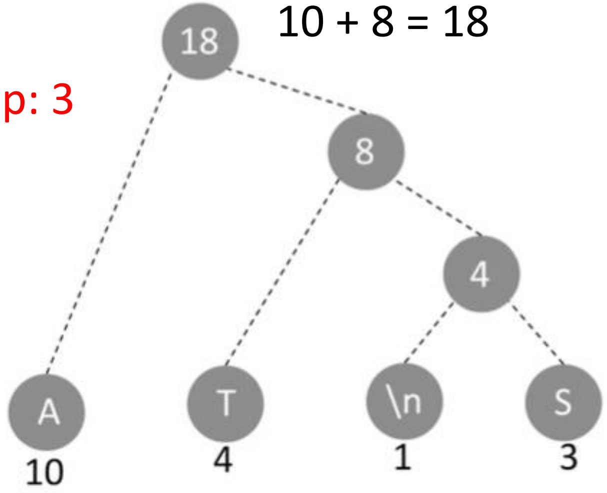
1+3 = 4

Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1

Step: 2



Step: 3

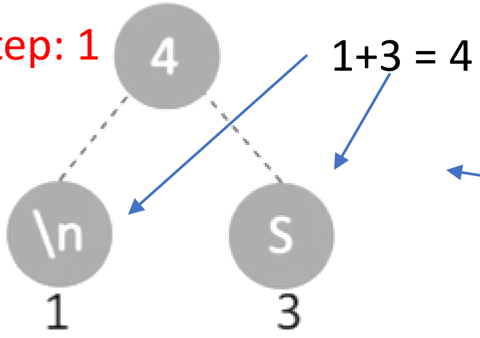


10 + 8 = 18

Step 1: Take the 2 chars with the lowest frequency

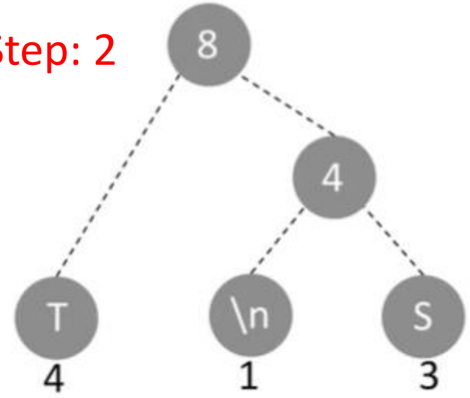
Step 2: Make a 2 leaf node tree from them

Step: 1

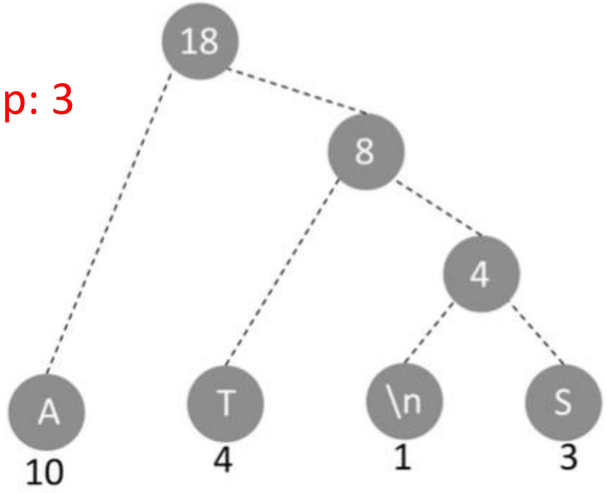


Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1

Step: 2



Step: 3

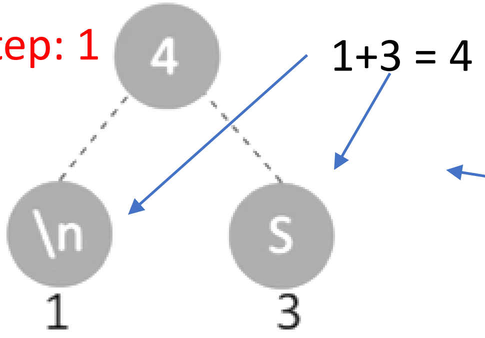


Step: 4

Step 1: Take the 2 chars with the lowest frequency

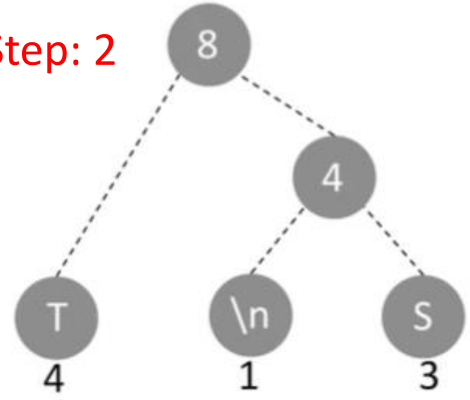
Step 2: Make a 2 leaf node tree from them

Step: 1

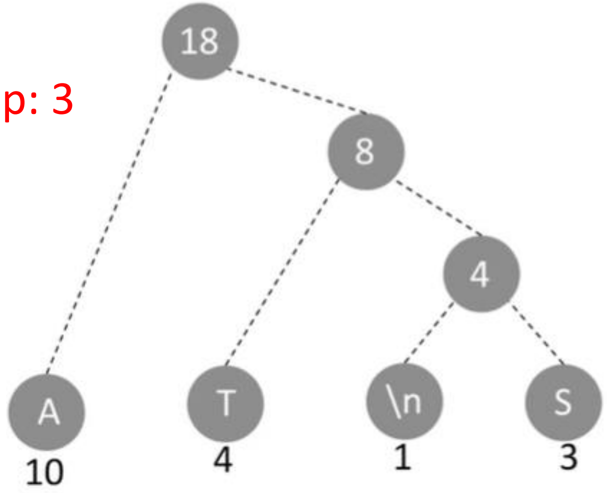


Char	Freq
A	10
E	15
I	12
S	3
T	4
P	13
\n	1

Step: 2

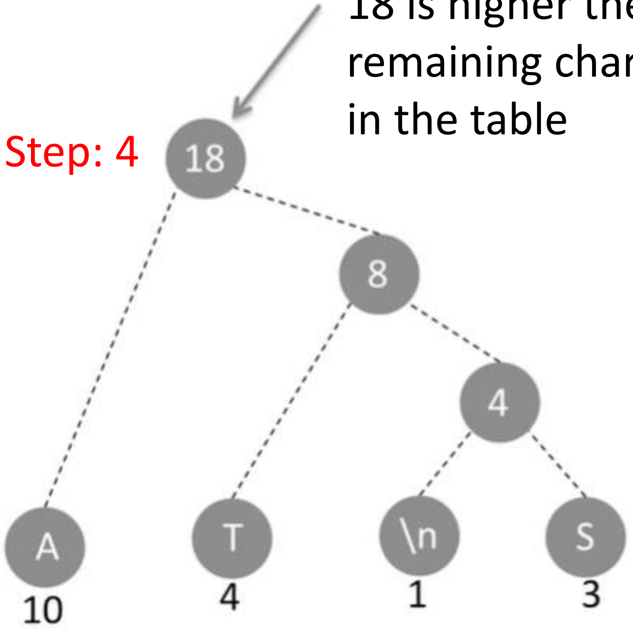


Step: 3



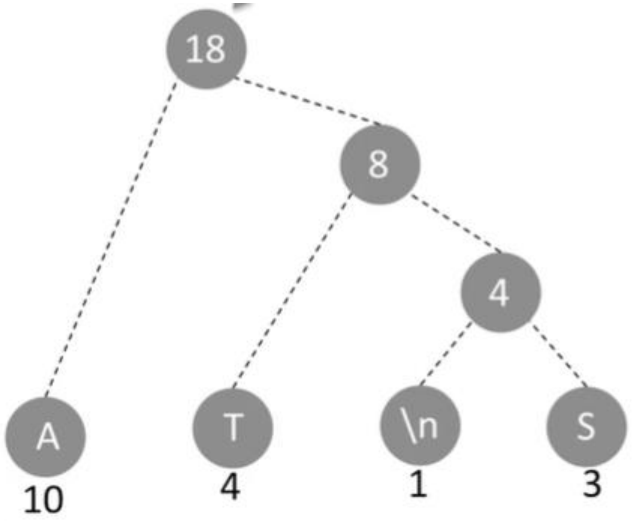
Step: 4

18 is higher then the remaining characters in the table

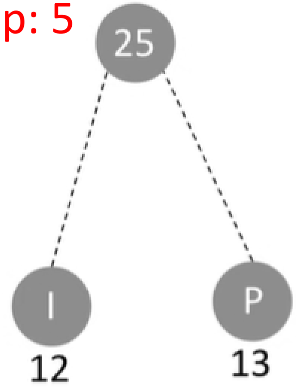


Char	Freq
~~A~~	~~10~~
E	15
I	12
~~S~~	~~3~~
~~T~~	~~4~~
P	13
~~\n~~	~~1~~

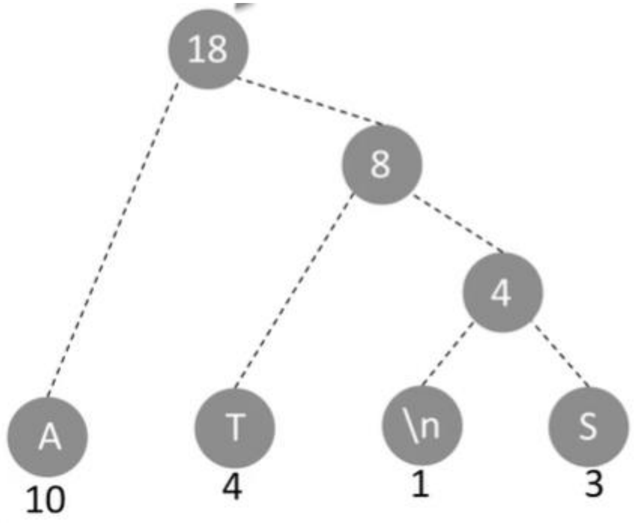
Char	Freq
~~A~~	~~10~~
E	15
I	12
~~S~~	~~3~~
~~T~~	~~4~~
P	13
~~\n~~	~~1~~



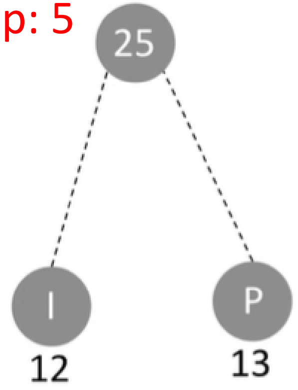
Step: 5



Char	Freq
~~A~~	~~10~~
E	15
~~I~~	~~12~~
~~S~~	~~3~~
~~T~~	~~4~~
~~P~~	~~13~~
~~\n~~	~~1~~

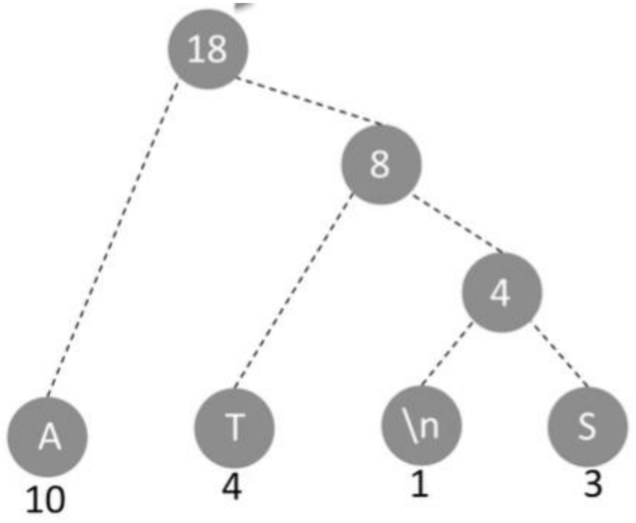


Step: 5

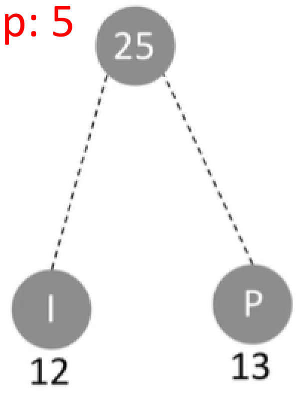


Step: 6

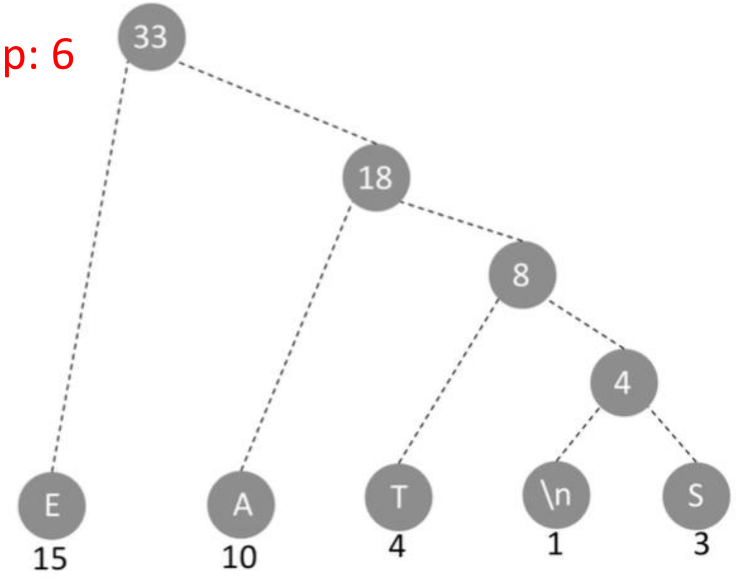
Char	Freq
~~A~~	~~10~~
~~E~~	~~15~~
~~I~~	~~12~~
~~S~~	~~3~~
~~T~~	~~4~~
~~P~~	~~13~~
~~\n~~	~~1~~



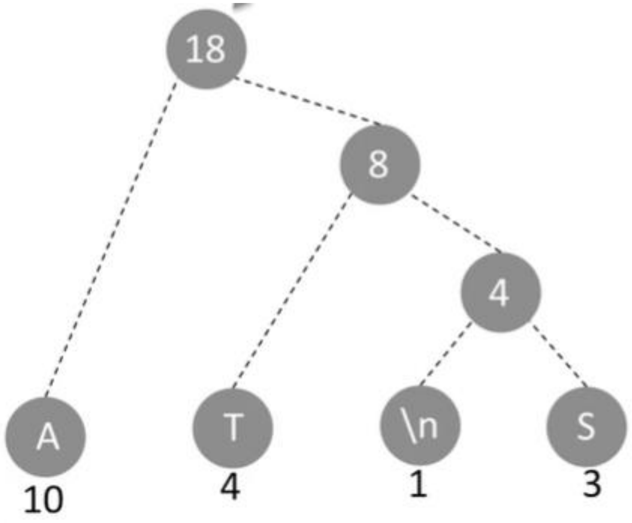
Step: 5



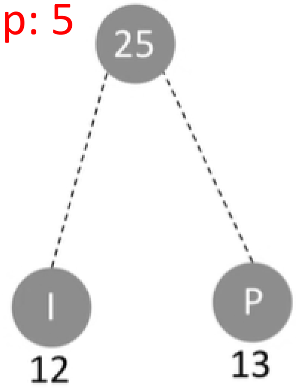
Step: 6



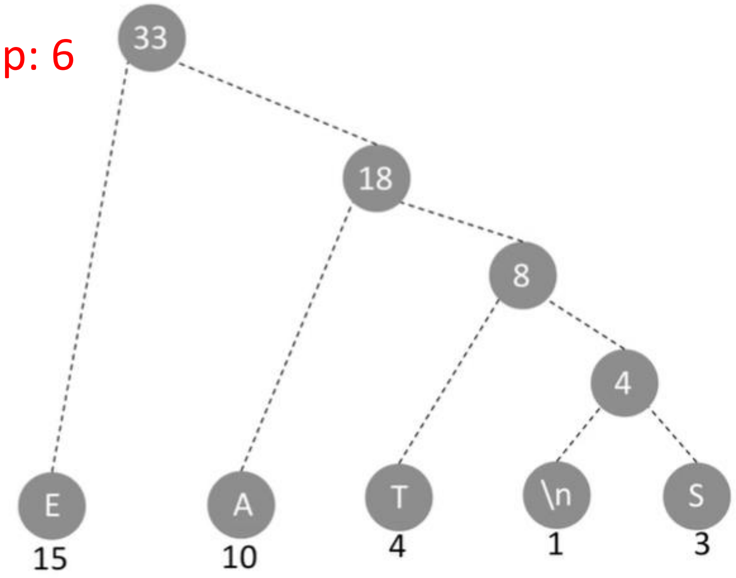
Char	Freq
~~A~~	~~10~~
~~E~~	~~15~~
~~I~~	~~12~~
~~S~~	~~3~~
~~T~~	~~4~~
~~P~~	~~13~~
~~\n~~	~~1~~



Step: 5

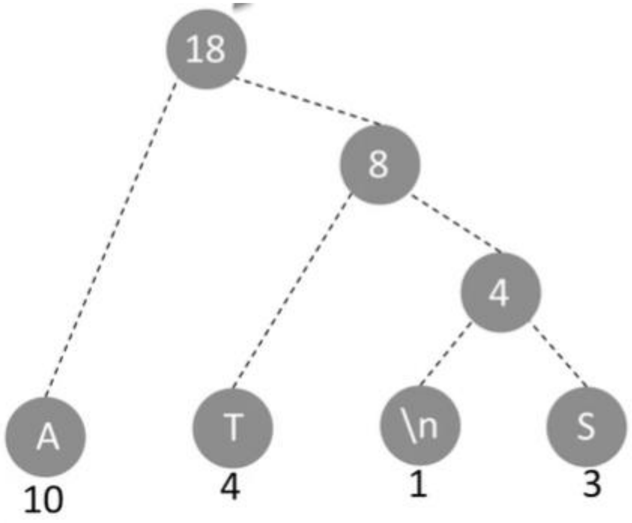


Step: 6

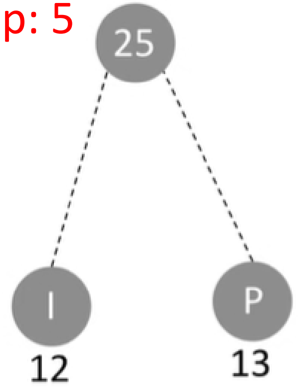


Step: 7

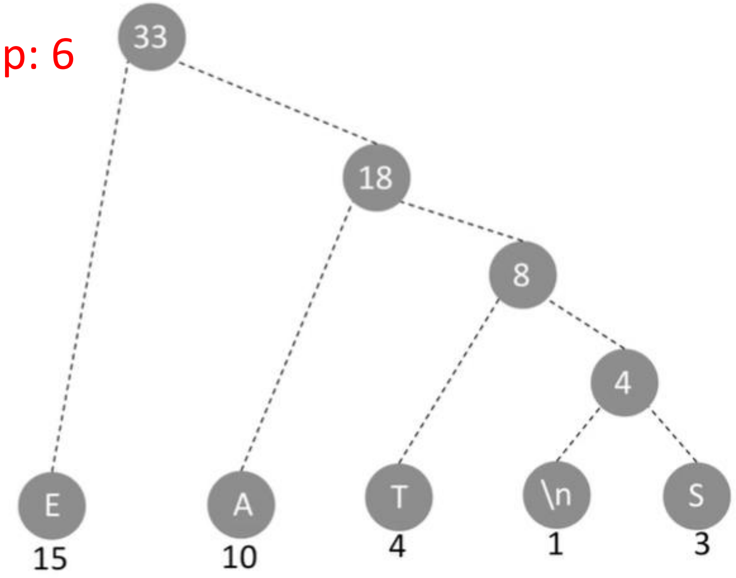
Char	Freq
~~A~~	~~10~~
~~E~~	~~15~~
~~I~~	~~12~~
~~S~~	~~3~~
~~T~~	~~4~~
~~P~~	~~13~~
~~\n~~	~~1~~



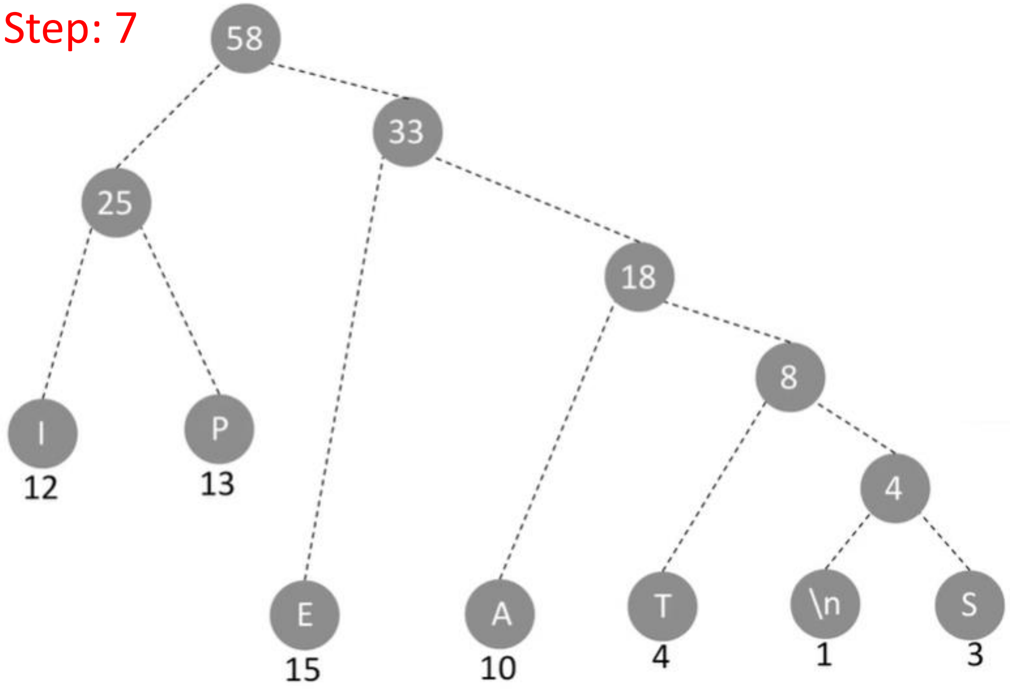
Step: 5



Step: 6

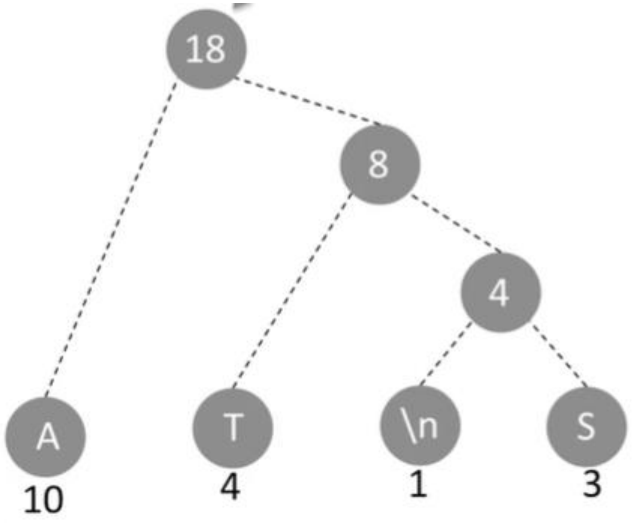


Step: 7

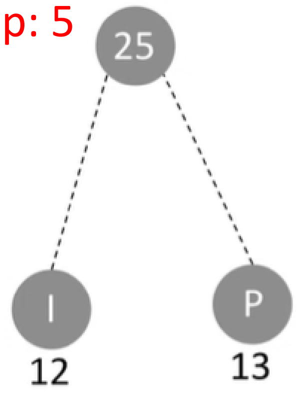


Step: 8 (Assign Bits to the tree)

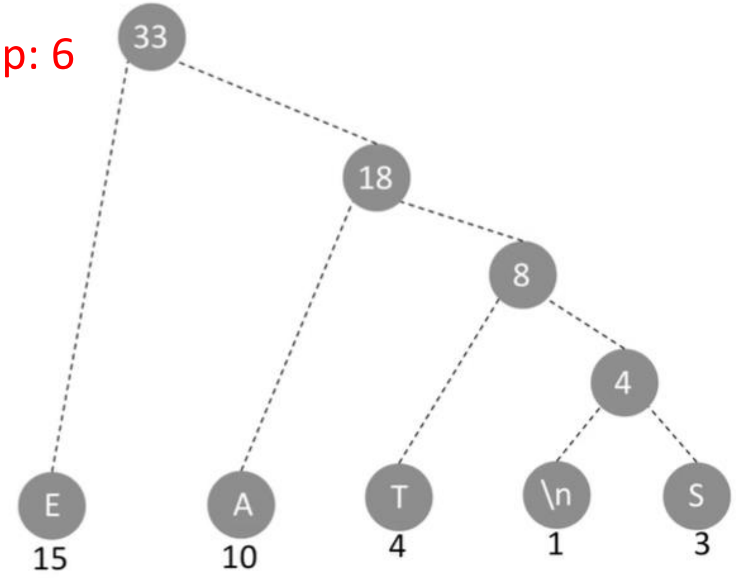
Char	Freq
~~A~~	~~10~~
~~E~~	~~15~~
~~I~~	~~12~~
~~S~~	~~3~~
~~T~~	~~4~~
~~P~~	~~13~~
~~\n~~	~~1~~



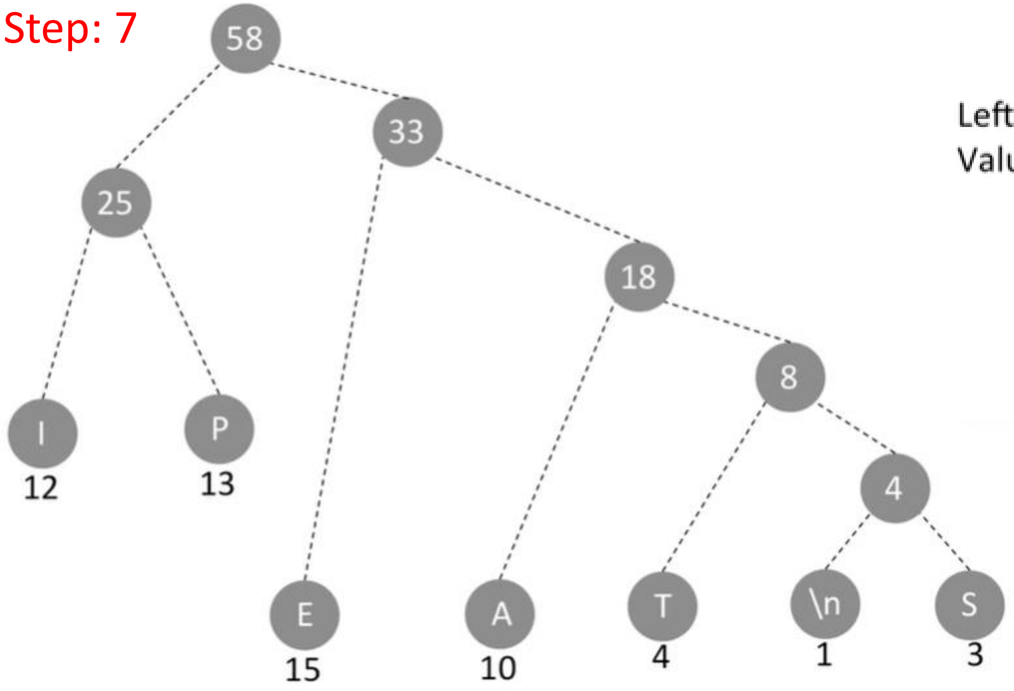
Step: 5



Step: 6

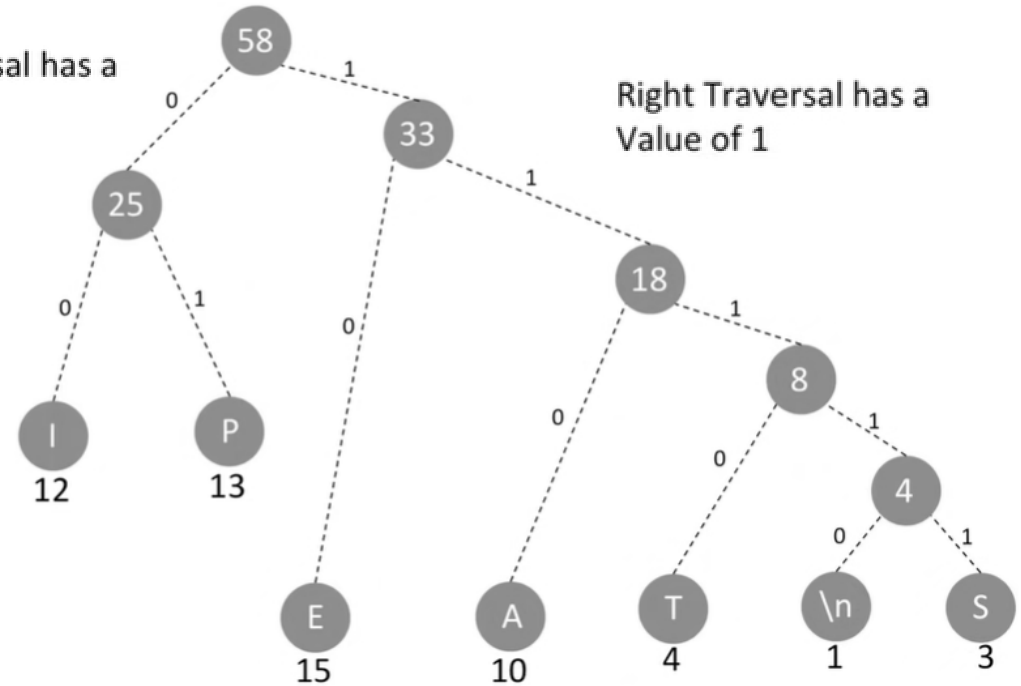


Step: 7



Left Traversal has a Value of 0

Step: 8



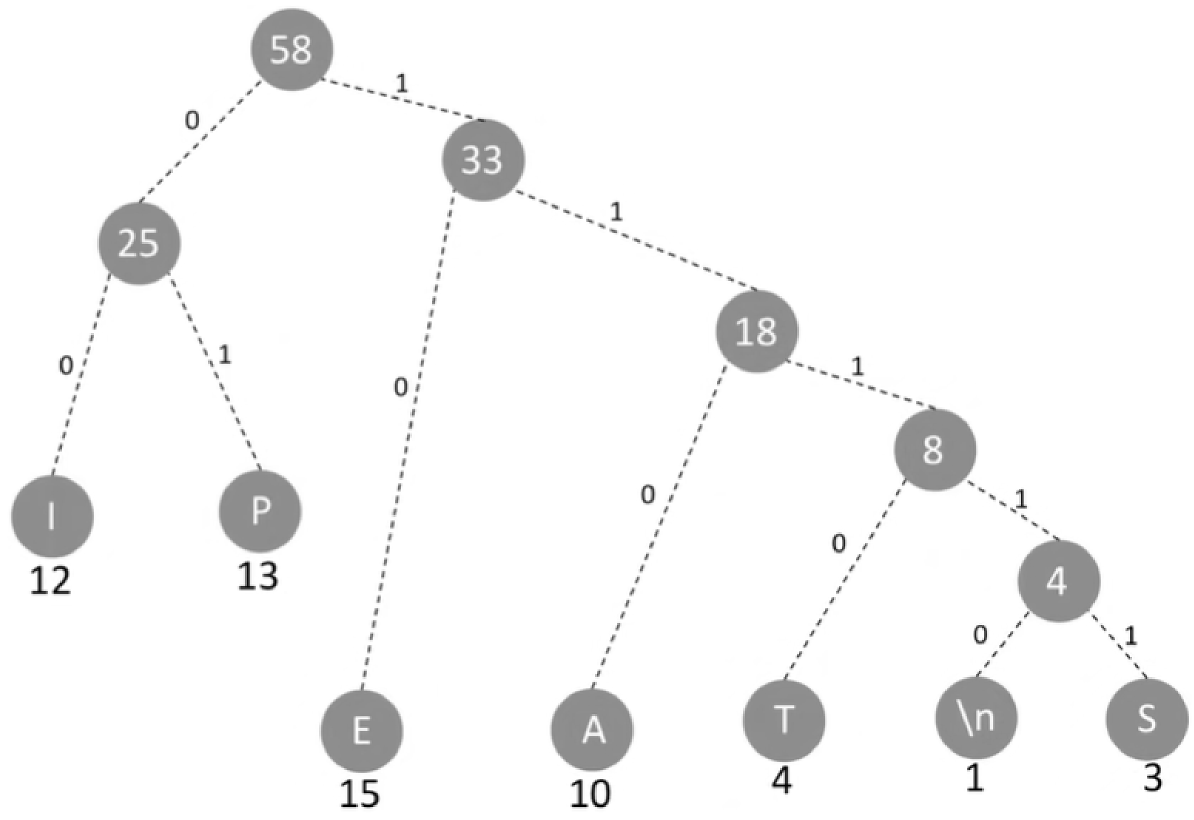
Right Traversal has a Value of 1

Applications (Huffman Encoding)

Char	Code	Freq	Total Bits
A	110	10	30
E	10	15	30
I	00	12	24
S	11111	3	15
T	1110	4	16
P	01	13	26
\n	11110	1	5

3*10=30
(i.e. 110 = 3 bits)

149



Character	Code	Frequency	Total Bits
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	12
T	100	4	12
P	101	13	39
Newline	110	1	3

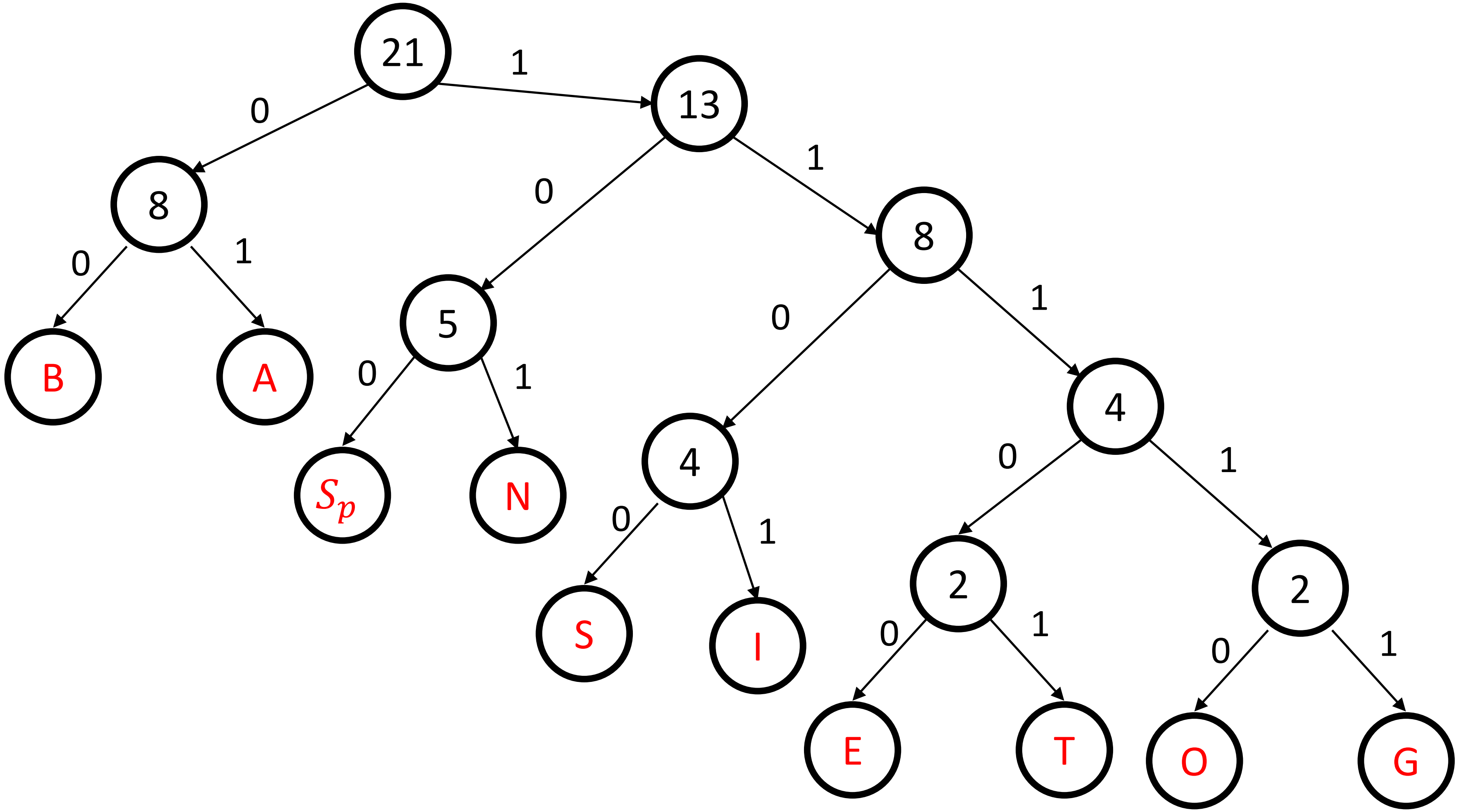
Fix bit (Total Bits) = 174 (Slide)

Applications (Huffman Encoding)

Char	Freq
B	5
I	2
G	1
S_p	3
O	1
T	1
E	1
S	2
A	3
N	2

Applications (Huffman Encoding)

Char	Freq	Code
B	5	00
I	2	1101
G	1	11111
S_p	3	100
O	1	11110
T	1	11101
E	1	11100
S	2	1100
A	3	01
N	2	101



Applications of Huffman Coding

Real-world examples of Huffman Coding in practice (Link)

- Lossless Image Compression
  - A simple task for Huffman coding is to encode images in a lossless manner. This is useful for precise and critical images such as medical images and Very High Resolution (VHR) satellite images, where it is important for the data to remain consistent before and after compression.

- Image with a diverse set of colors:
  - This image has a broad range of colors. It has many red pixels (in the horse), green pixels (in the grass), and blue pixels (in the sky). Intuition hints that this image may not be very compressible. The entropy of this image is calculated to be 5.39. The results of the image compression with Huffman coding are shown below:

Compression Ratio	Bits/Points after
1.02	7.81

The values that each number in the matrix can take on is an integer from 0 to 255. Encoding this range of numbers requires an 8-bit number.