UNIT 1
Advance Trees

DR. SUYASH BHARDWAJ

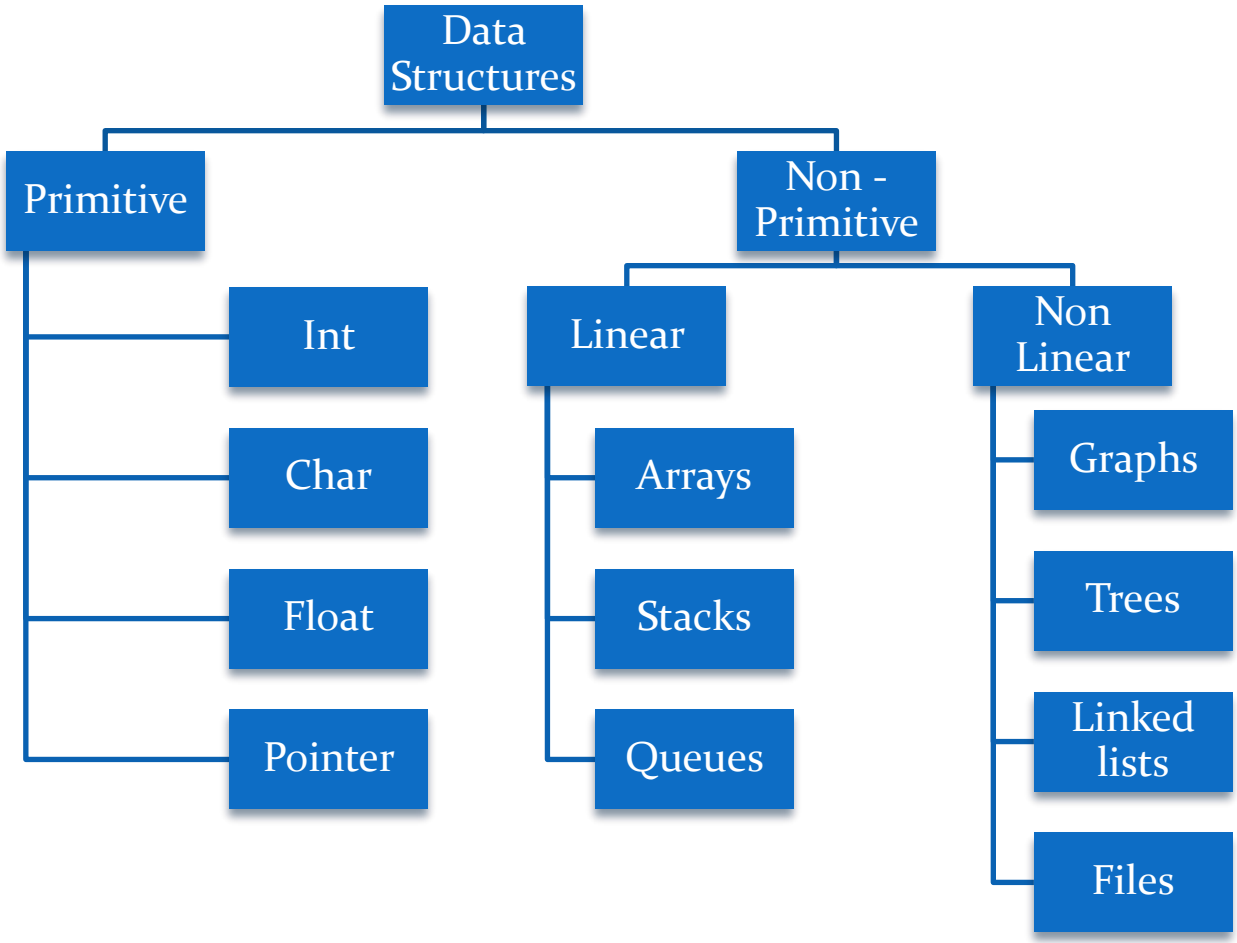
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

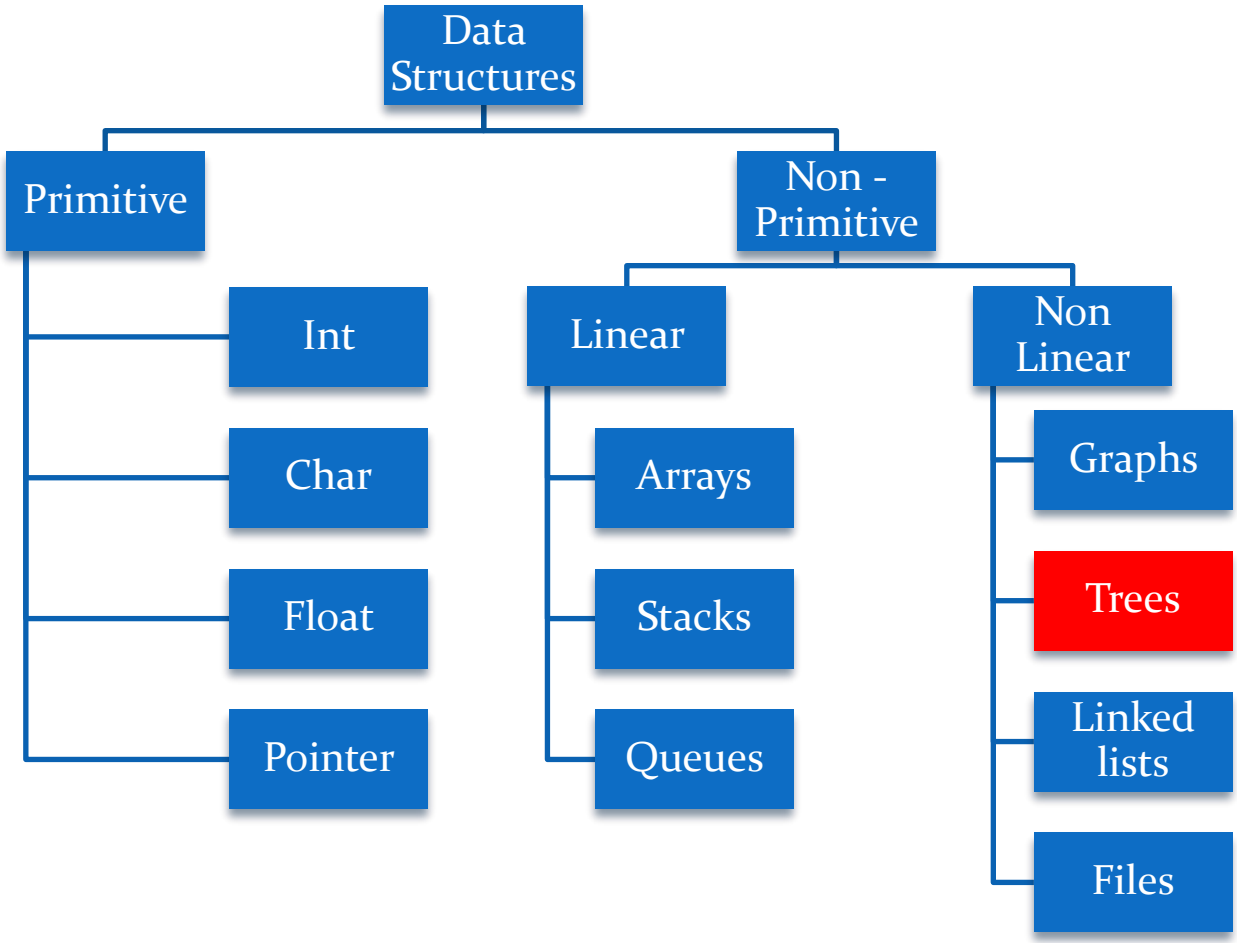
FACULTY OF ENGINEERING AND TECHNOLOGY

GURUKUL KANGRI VISHWAVIDYALAYA, HARIDWAR

In this unit

- Introduction to Binary tree, Construction of Binary Tree, AVL Tree
- Advanced Trees:
  - Threaded Binary trees,
  - Traversing Threaded Binary trees,
  - recursive and non recursive traversal of binary tree,
  - Efficient non recursive tree traversal algorithms,
  - B+ Tree,
  - B* Tree,
  - Weight Balanced Trees (Huffman Trees)


Data Structures
Primitive
Non - Primitive
Int
Char
Float
Pointer
Linear
Non Linear
Arrays
Stacks
Queues
Graphs
Trees
Linked lists
Files

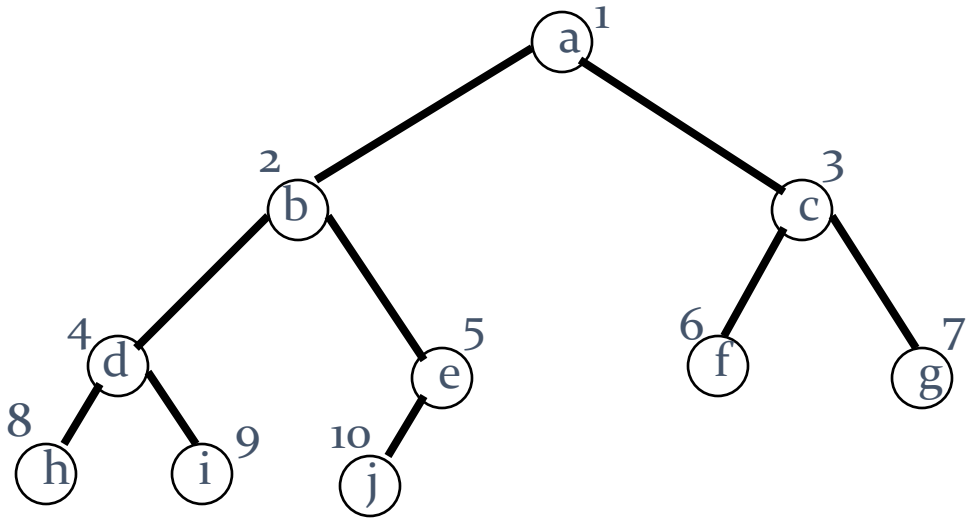

Data Structures
Primitive
Non - Primitive
Int
Char
Float
Pointer
Linear
Non Linear
Arrays
Stacks
Queues
Graphs
Trees
Linked lists
Files

Binary Tree Representation

- Array representation.
- Linked representation.

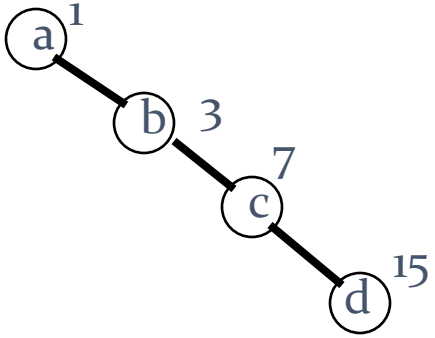
Array Representation

- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].



tree[]		a	b	c	d	e	f	g	h	i	j

Right-Skewed Binary Tree



tree[]		a	-	b	-	-	-	c	-	-	-	-	-	-	-	d

- An n node binary tree needs an array whose length is between n+1 and 2^n.

Linked Representation

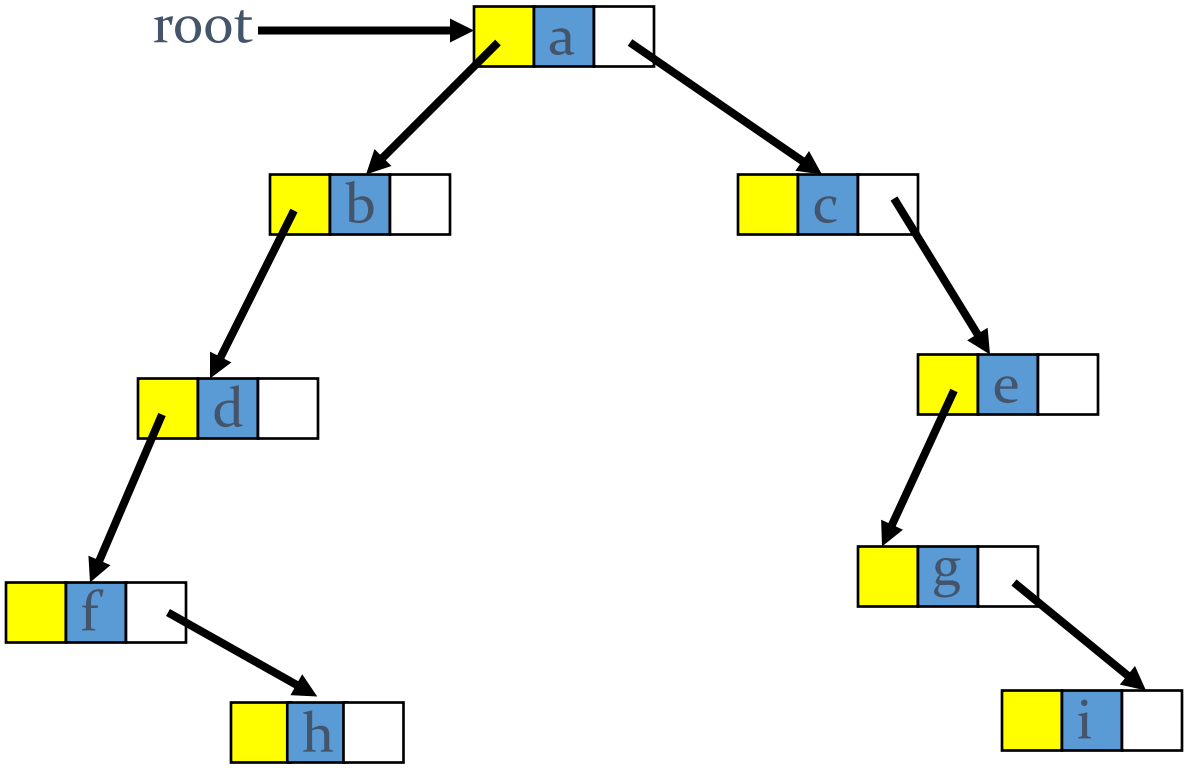
- Each binary tree node is represented as an object whose data type is BinaryTreeNode.
- The space required by an **n** node binary tree is

  n * (space required by one node).

Link Representation of Binary Tree

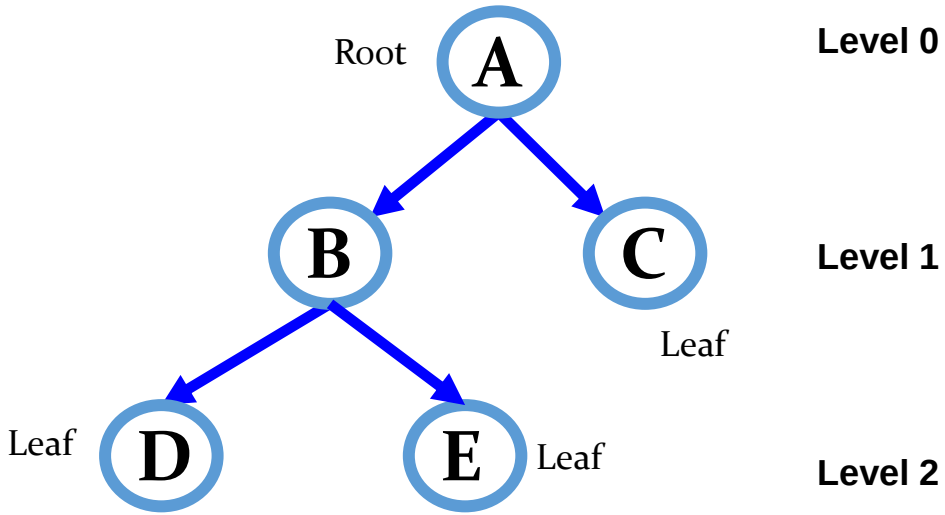
```
typedef struct node
{
    int data;
    struct node *lc,*rc;
};
```

Linked Representation Example



TREE

- A **tree** is a hierarchical representation of a finite set of one or more data items such that:
  - There is a special node called the root of the tree.
  - The nodes other than the root node form an ordered pair of disjoint subtrees.



Basic Terminology

- **Root Node** : the root node R is the topmost node in the tree, if R=NULL, then it means the tree is empty

- **Sub Trees :** if the root node R is not NULL, then the disjoints sub sets, ie right part and left part are known as left and right sub trees

- **Path :** a sequence of consecutive edges is called a path.

- **Level Number :** every node in the tree is assigned a level number in such a way that the root node is at level 0.

Basic Terminology

- **Degree :** degree of a node is equal to the number of children that a node has, the degree of leaf node is zero.

- **In degree :** the in degree of a node is the number of edges arriving at that node.

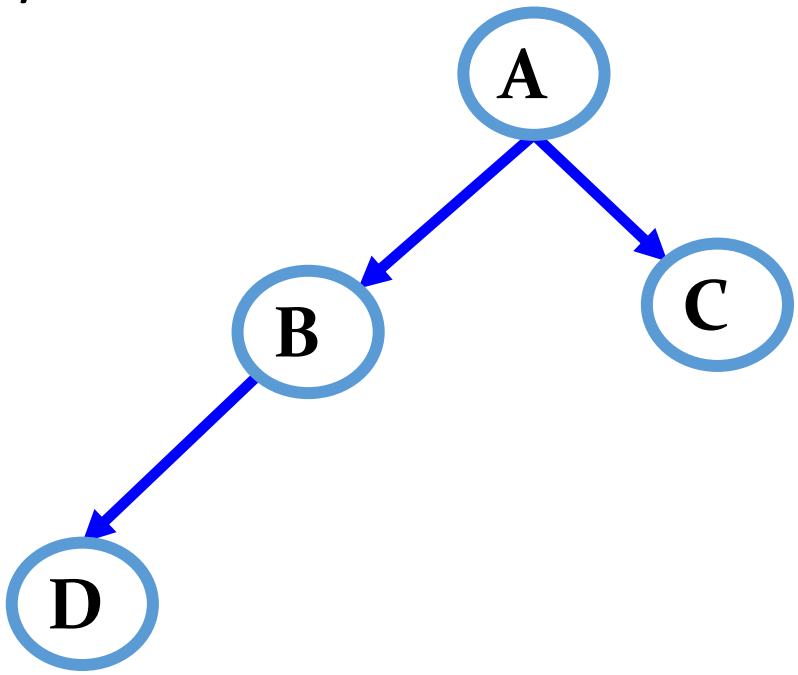
- **Out degree :** the out degree of a node is the number of edges leaving that node.

Types of Trees

- **Binary Tree :** a tree in which any node can have maximum of 2 children, i.e either 0, 1 or 2 children.
- **Strict Binary Tree :** is a Binary Tree in which any node can have 2 children or no children at all.
- **Complete Binary Tree :** is a Binary Tree in which any node can have 2 children or no children with all leaf nodes at same level.
- **Extended Binary Tree :** is a Binary tree in which every empty sub tree is replaced by an extended node. The original nodes of the tree are known as internal nodes, and the new added nodes are known as external nodes

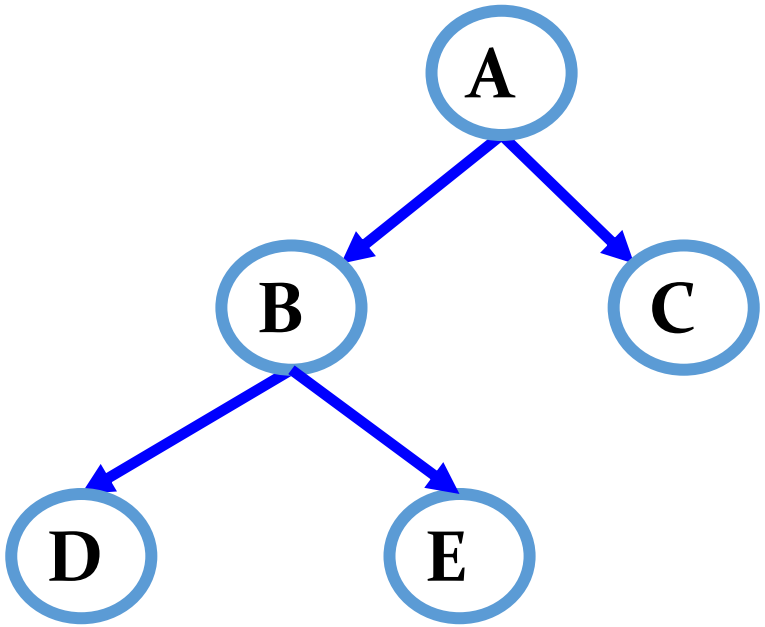
Binary Tree

Binary Tree : a tree in which any node can have maximum of 2 children, i.e either 0, 1 or 2 children.

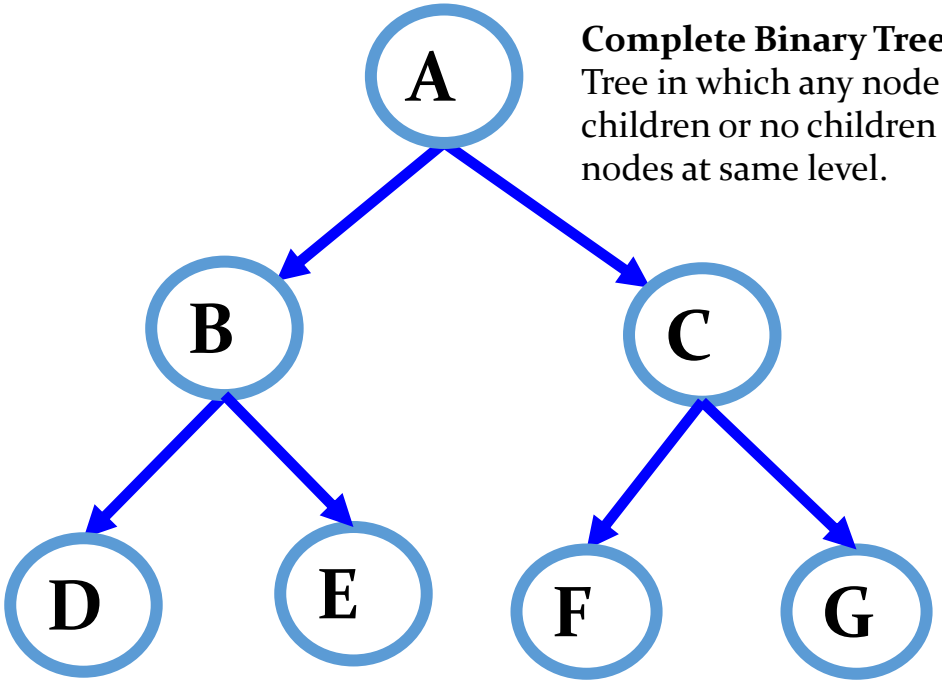


Strict Binary Tree

Strict Binary Tree : is a Binary Tree in which any node can have 2 children or no children at all.

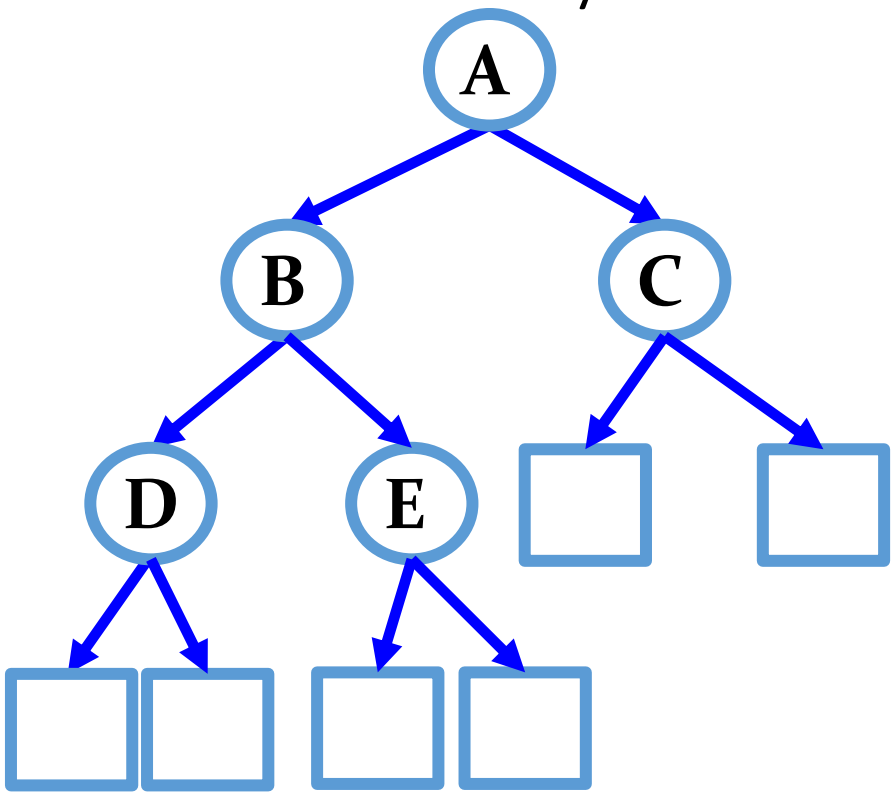


Complete Binary Tree



Complete Binary Tree : is a Binary Tree in which any node can have 2 children or no children with all leaf nodes at same level.

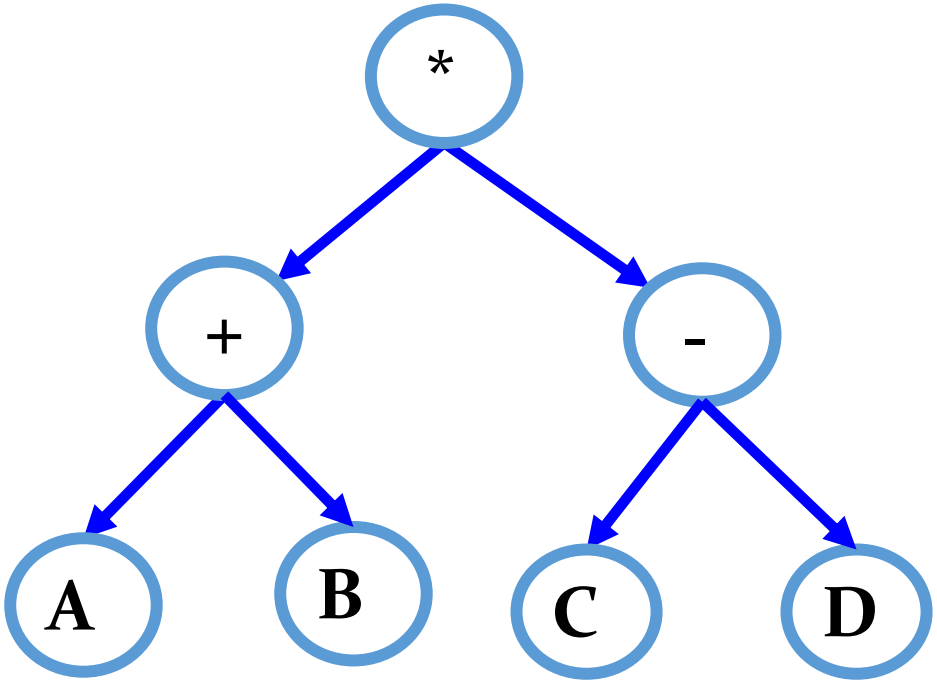
Extended Binary Tree



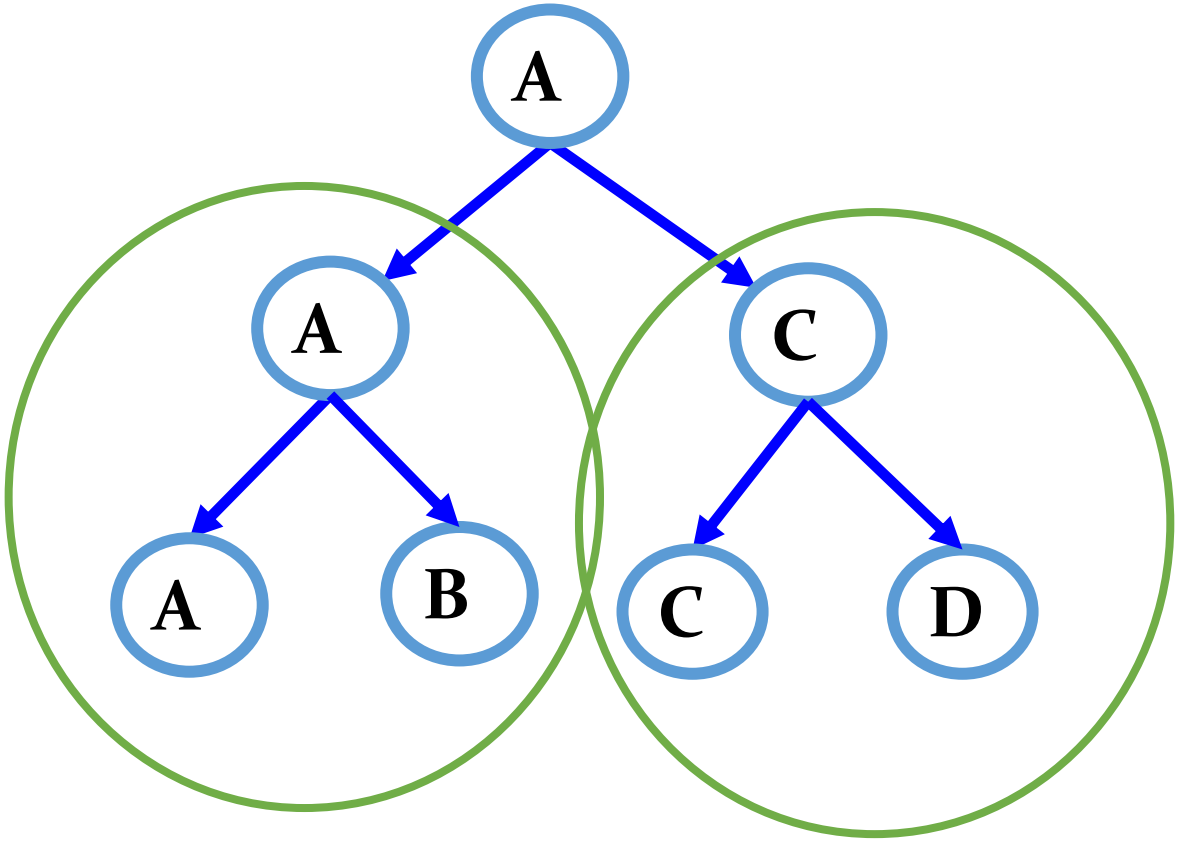
Extended Binary Tree : is a Binary tree in which every empty sub tree is replaced by an extended node.
The original nodes of the tree are known as internal nodes, and the new added nodes are known as external nodes

Expression Trees

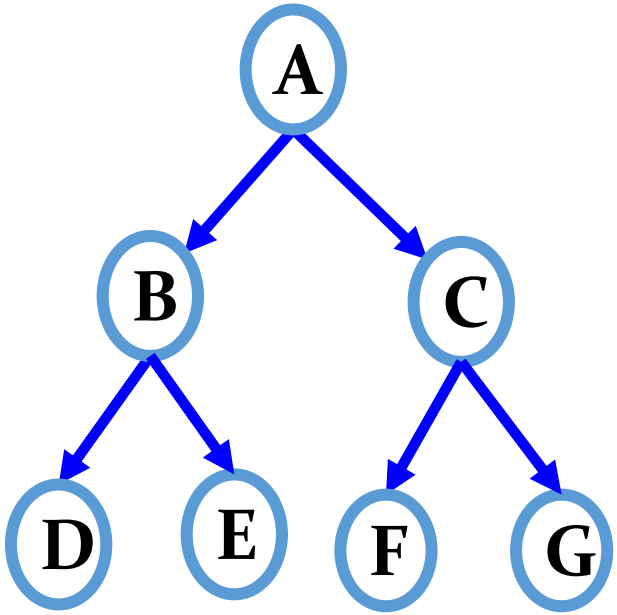
Exp = (A+B)*(C-D)



Tournament Trees

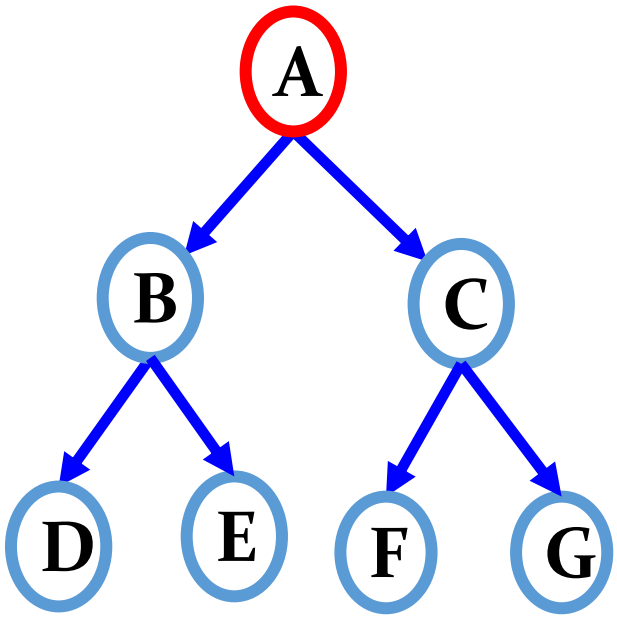


Tree Traversal : Pre order



- Pre order (N L R)

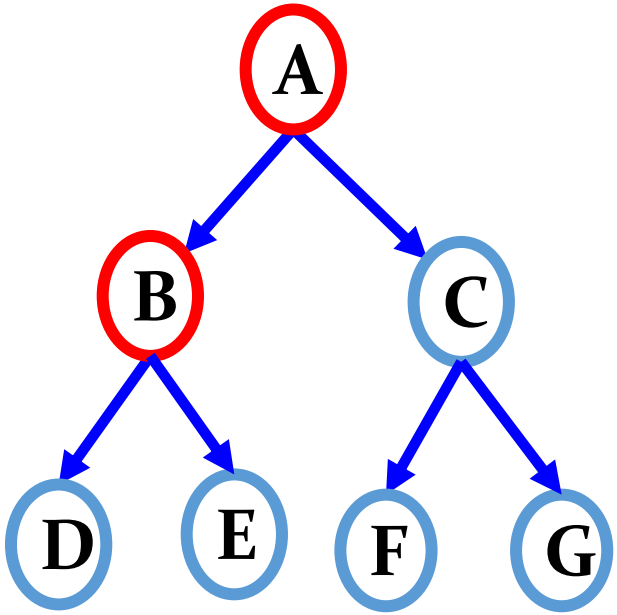
Tree Traversal : Pre order



- Pre order (N L R)

  A

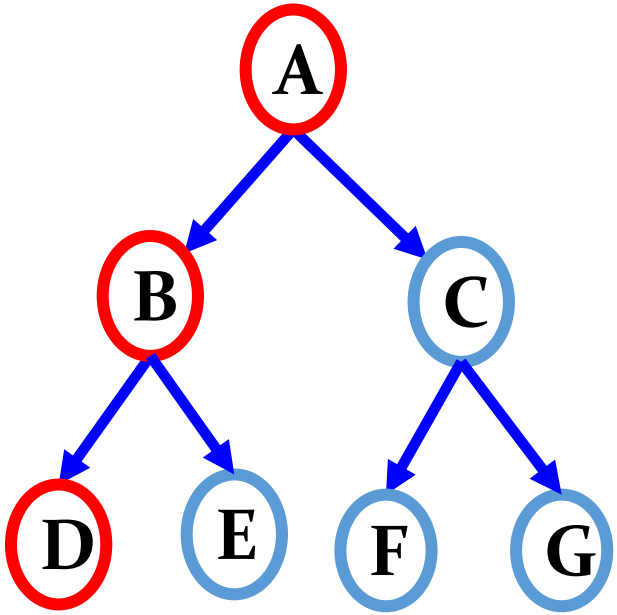
Tree Traversal : Pre order



- Pre order (N L R)

  A, B

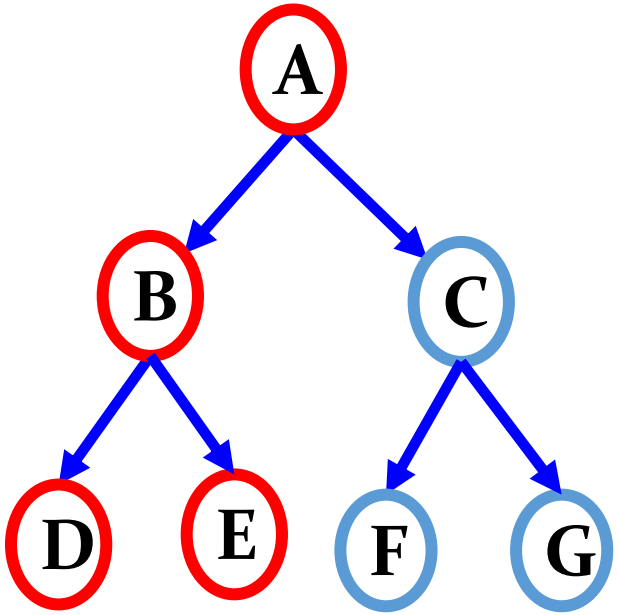
Tree Traversal : Pre order



- Pre order (N L R)

  A, B, D

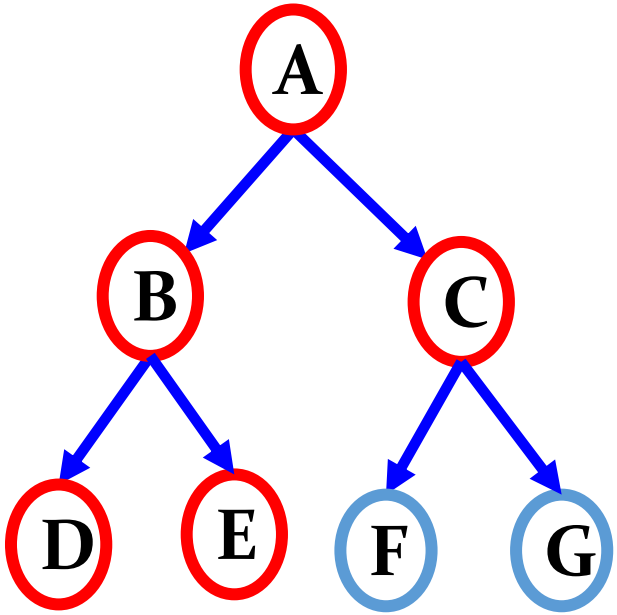
Tree Traversal : Pre order



- Pre order (N L R)

  A, B, D, E

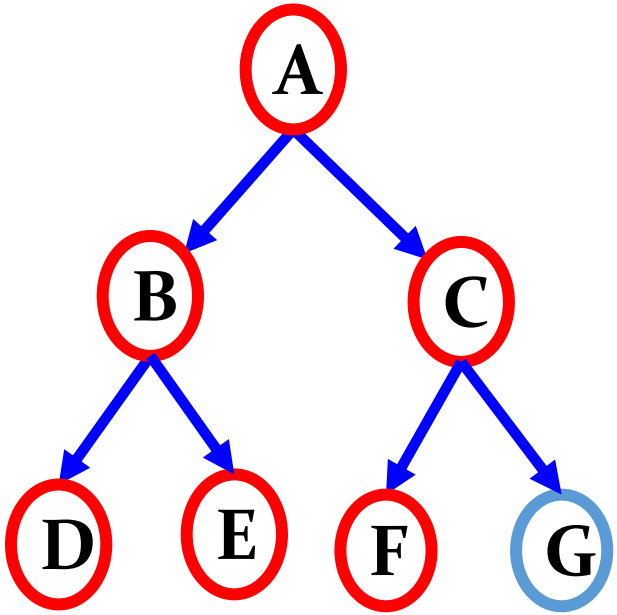
Tree Traversal : Pre order



- Pre order (N L R)

  A, B, D, E, C

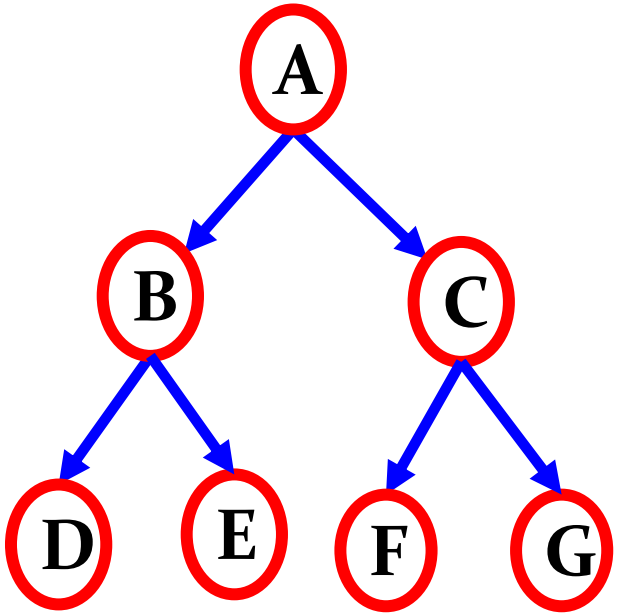
Tree Traversal : Pre order



- Pre order (N L R)

  A, B, D, E, C, F

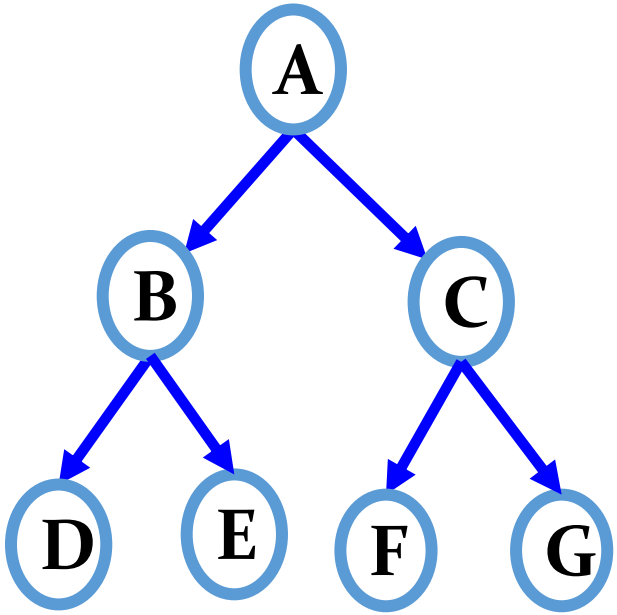
Tree Traversal : Pre order



- Pre order (N L R)

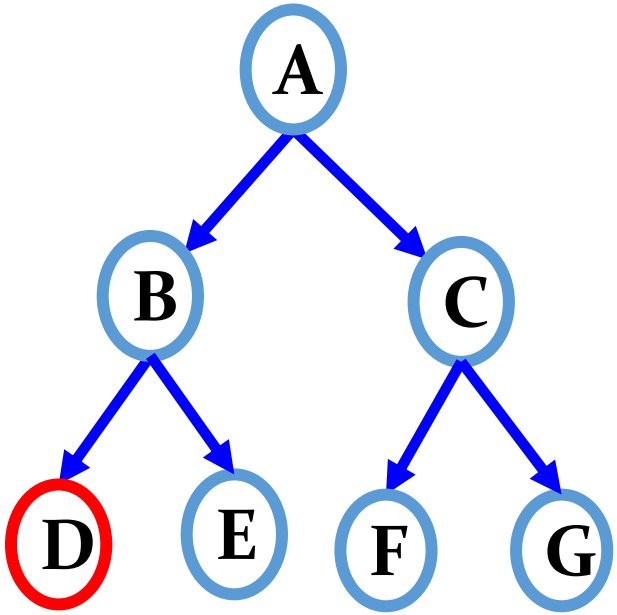
  A, B, D, E, C, F, G

Tree Traversal : In order



- In order (L N R)

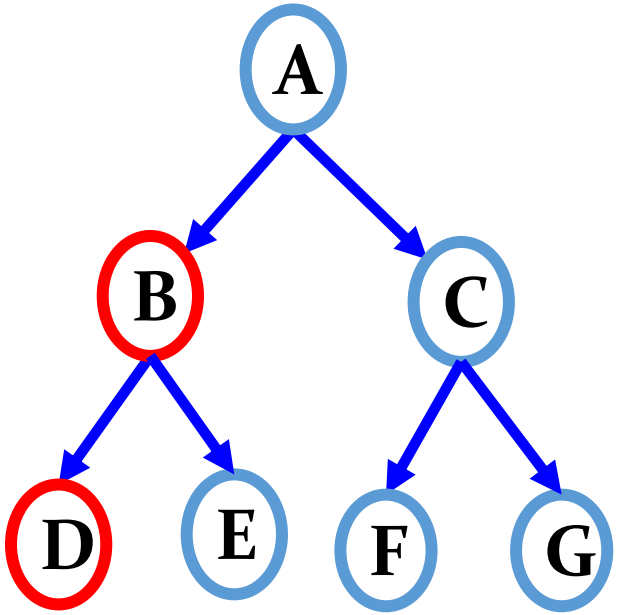
Tree Traversal : In order



- In order (L N R)

  D

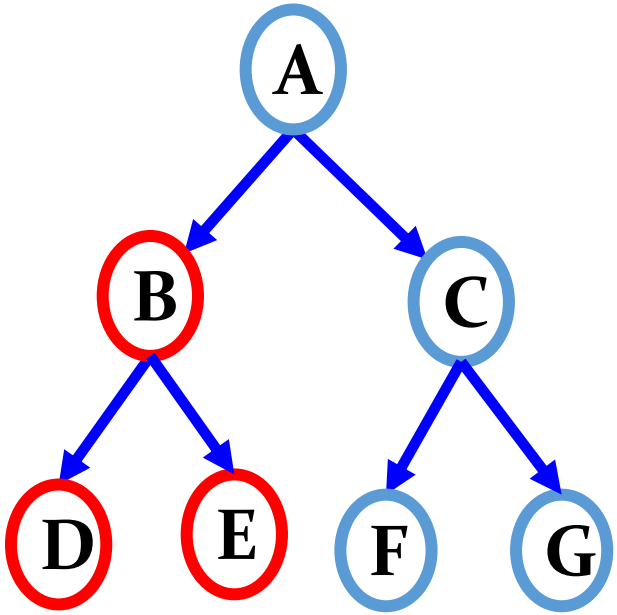
Tree Traversal : In order



- In order (L N R)

  D, B

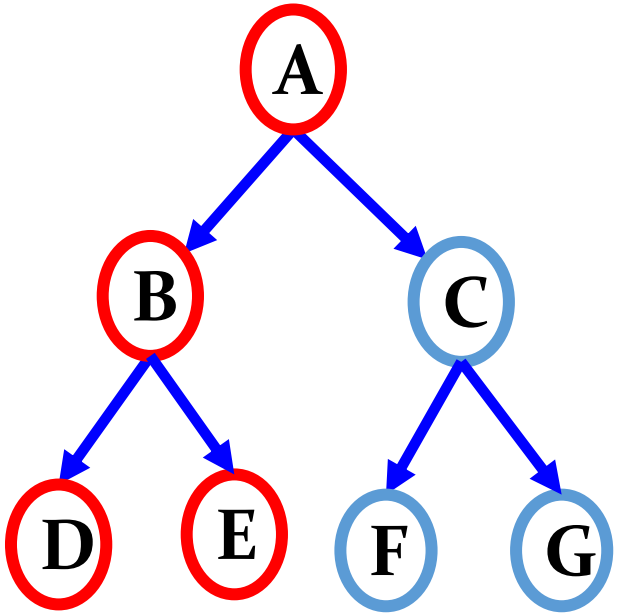
Tree Traversal : In order



- In order (L N R)

  D, B, E

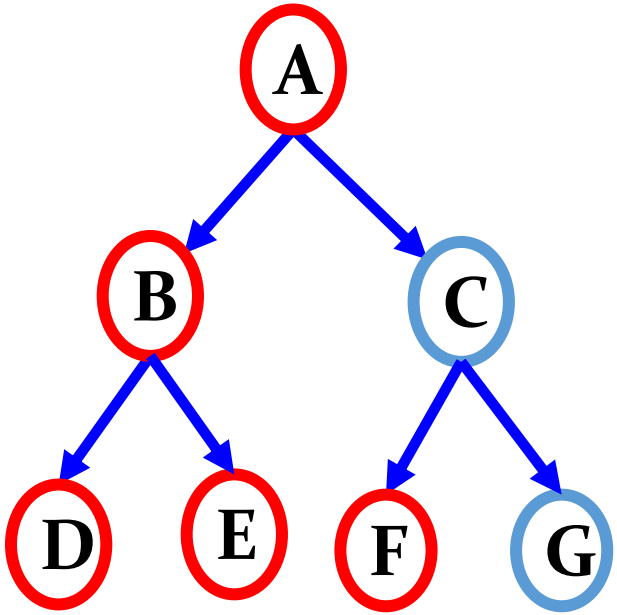
Tree Traversal : In order



- In order (L N R)

  D, B, E, A

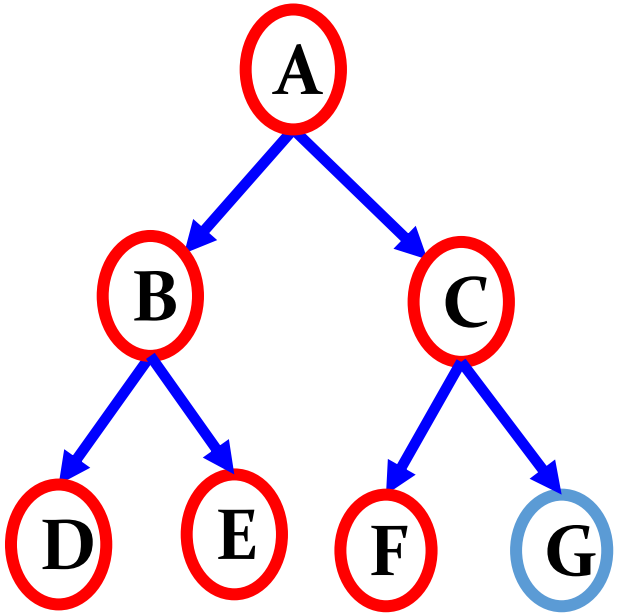
Tree Traversal : In order



- In order (L N R)

  D, B, E, A, F

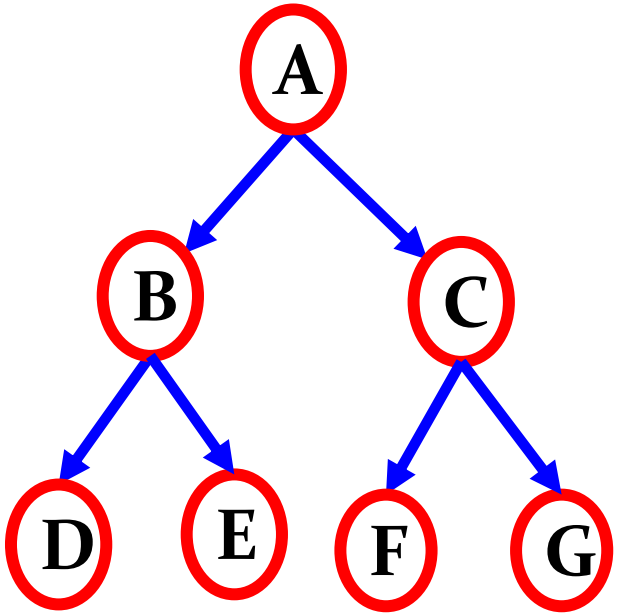
Tree Traversal : In order



- In order (L N R)

  D, B, E, A, F, C

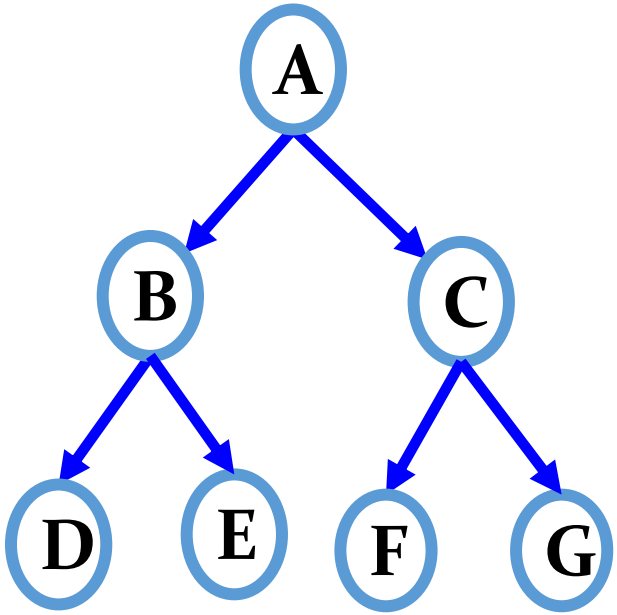
Tree Traversal : In order



- In order (L N R)

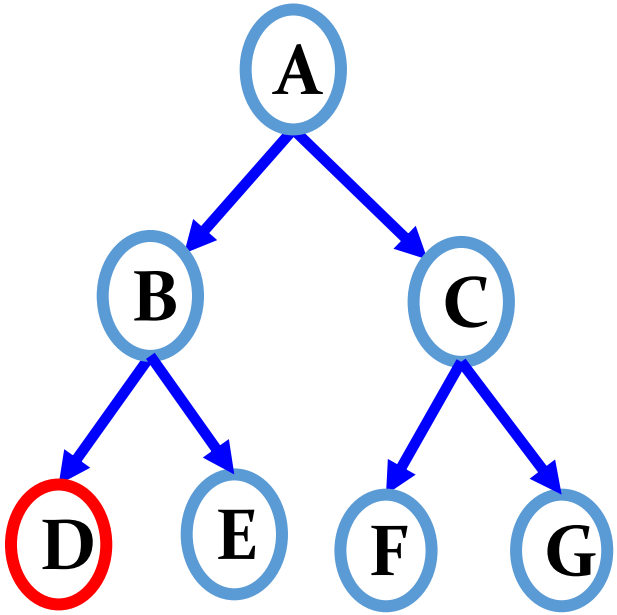
  D, B, E, A, F, C, G

Tree Traversal : Post order



- Post order (L R N)

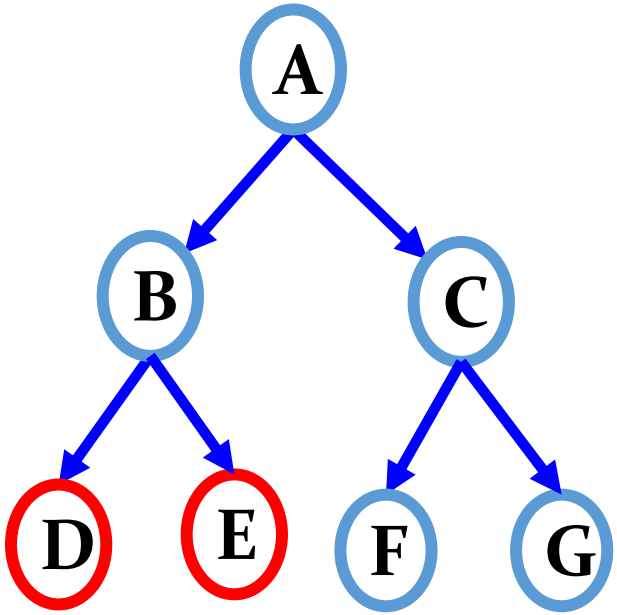
Tree Traversal : Post order



- Post order (L R N)

  D

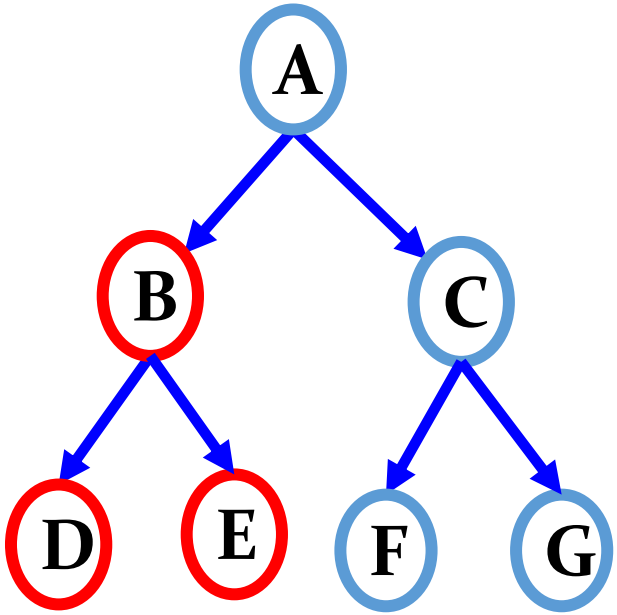
Tree Traversal : Post order



- Post order (L R N)

  D, E

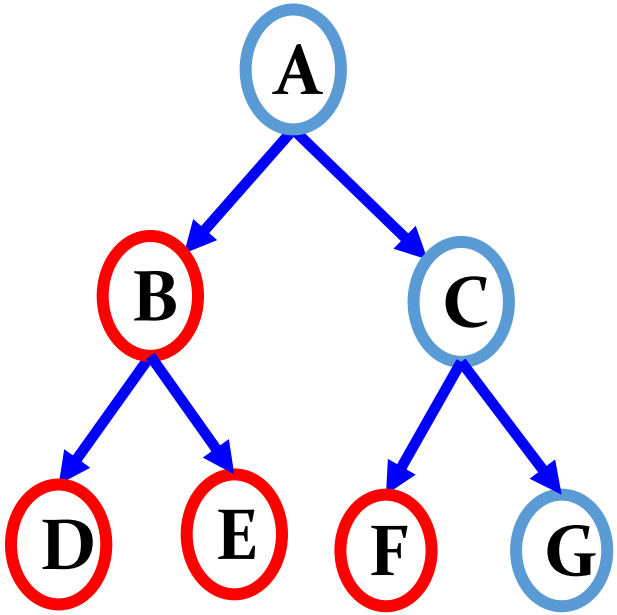
Tree Traversal : Post order



- Post order (L R N)

  D, E, B

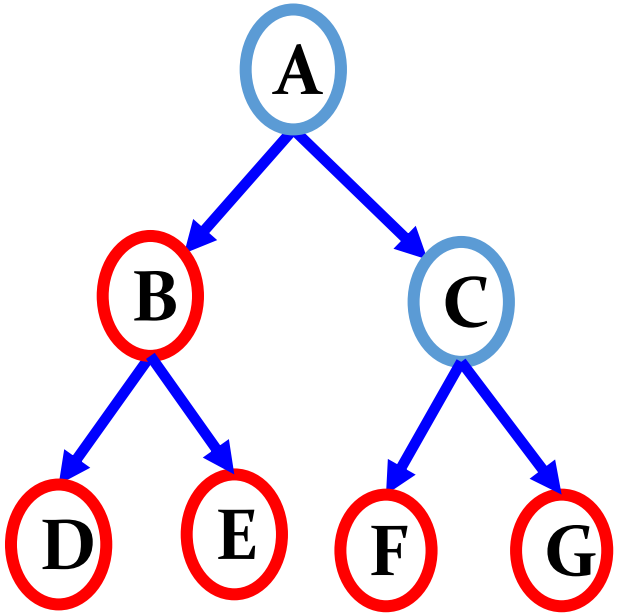
Tree Traversal : Post order



- Post order (L R N)

  D, E, B, F

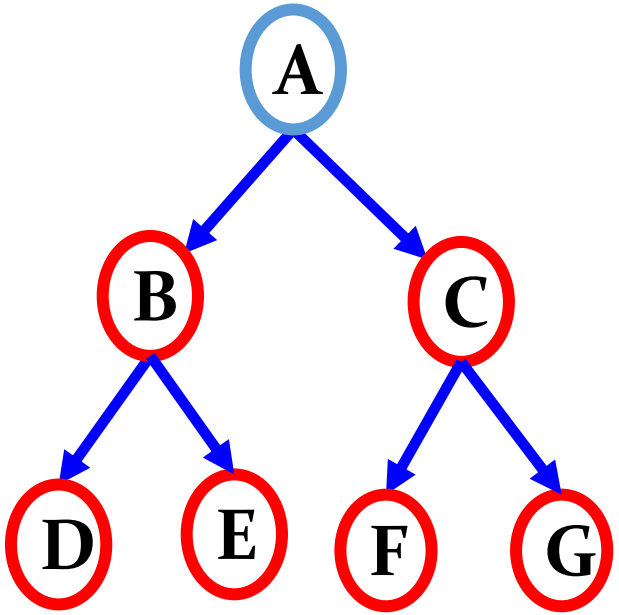
Tree Traversal : Post order



- Post order (L R N)

  D, E, B, F, G

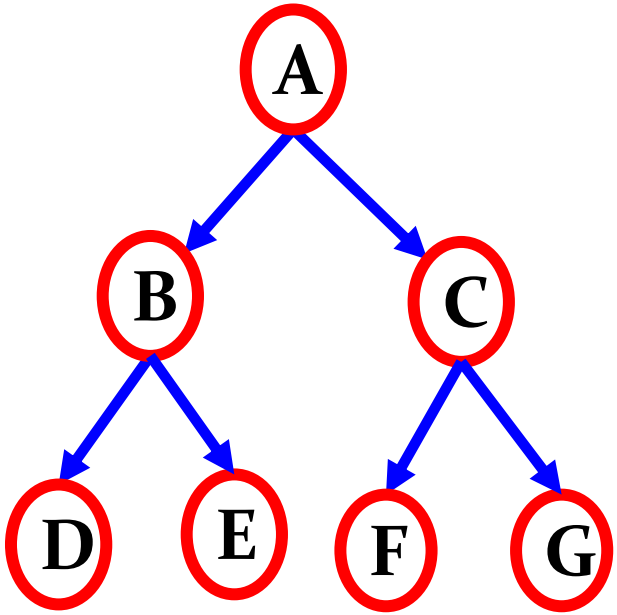
Tree Traversal : Post order



- Post order (L R N)

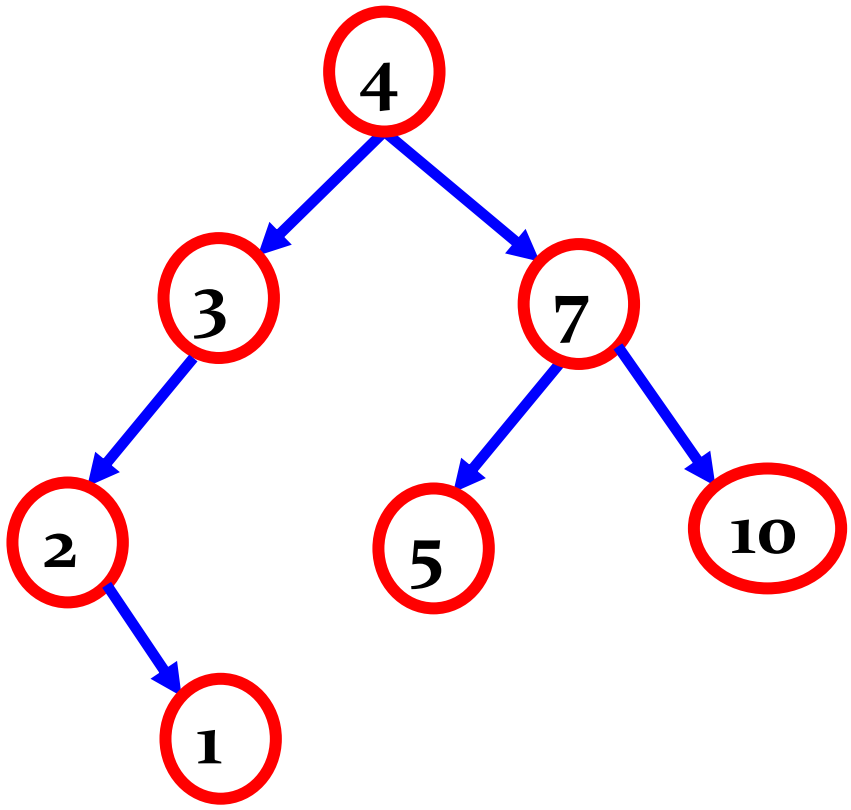
  D, E, B, F, G, C

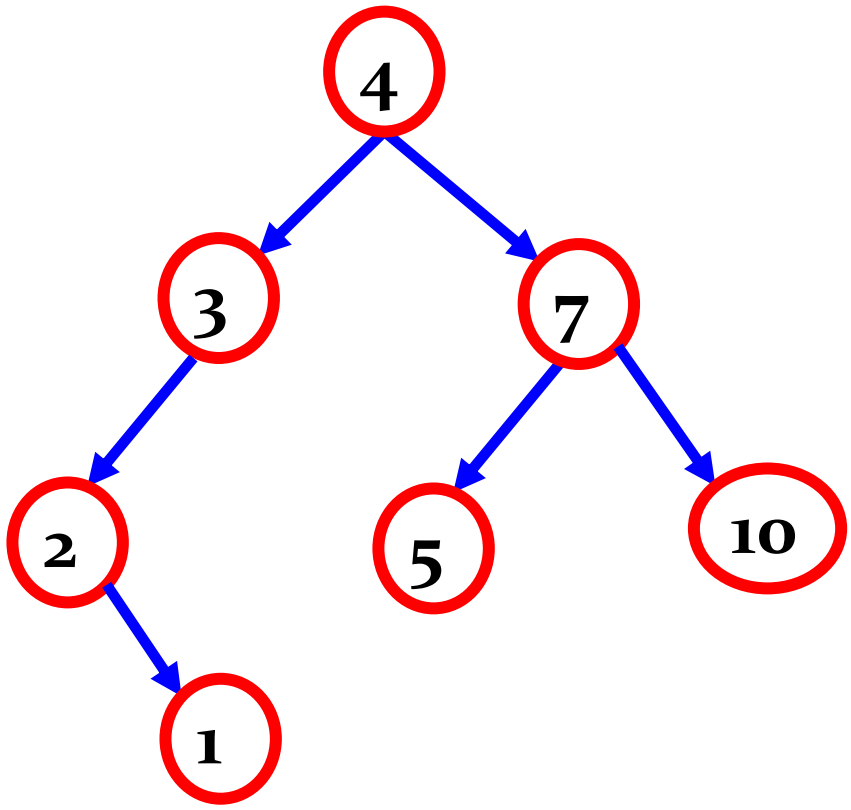
Tree Traversal : Post order



- Post order (L R N)

  D, E, B, F, G, C, A


4
3
7
2
5
10
1



PRE – N L R
4 3 2 1 7 5 10

POST – L R N
1 2 3 5 10 7 4

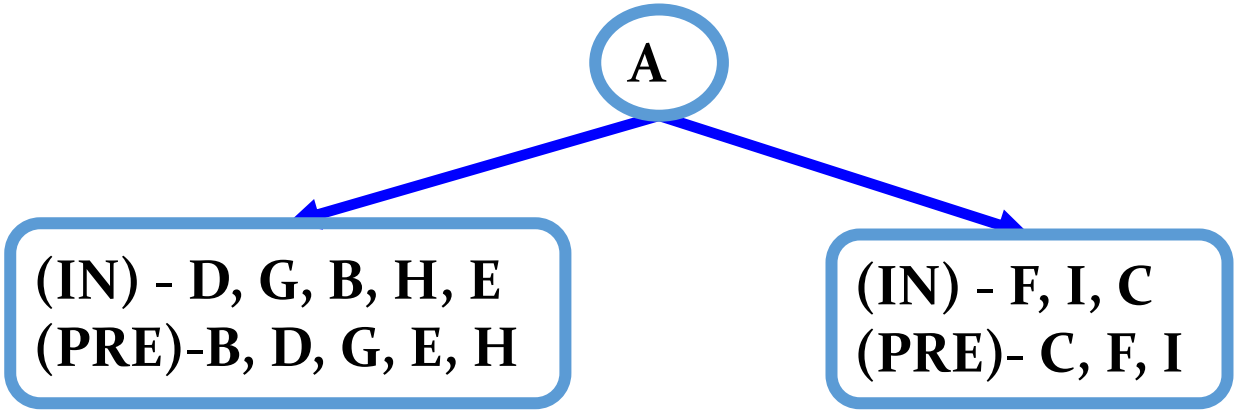
IN – L N R
2 1 3 4 5 7 10

Constructing Binary Tree

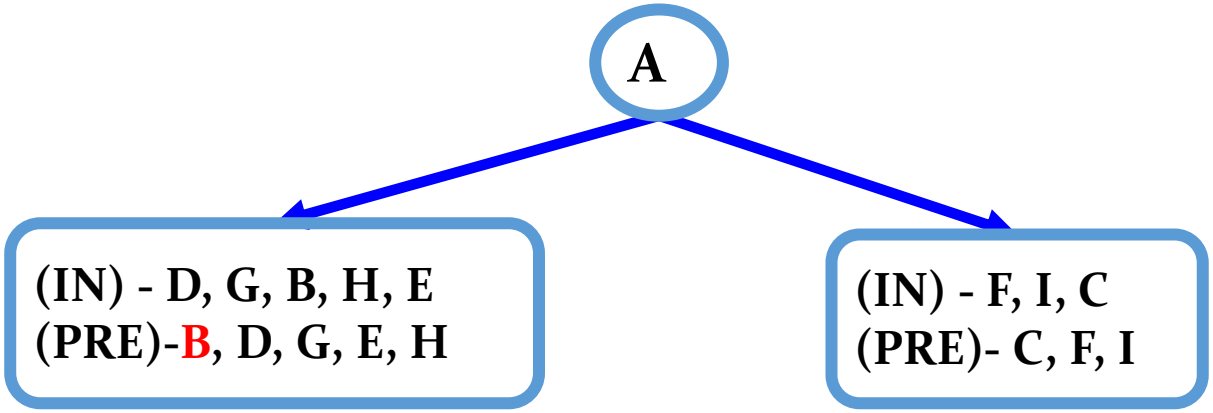
- In order – D, G, B, H, E, A, F, I, C
- Pre order – A, B, D, G, E, H, C, F, I

- Step 1 : finding the root
  - Root Node – A (from pre order)
- Step 2 : Find left and right part of the root
  - Left part - (IN) - D, G, B, H, E
    (PRE)-B, D, G, E, H
  - Right part - (IN) - F, I, C
    (PRE)- C, F, I

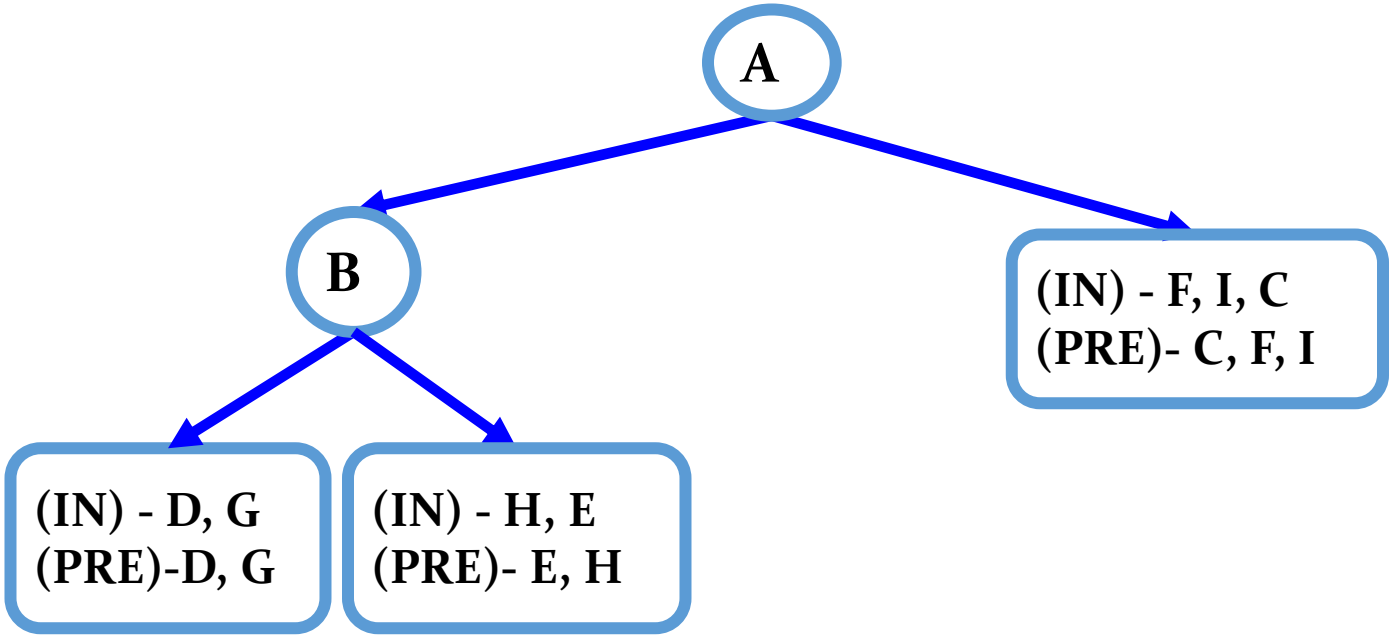
Constructing Binary Tree



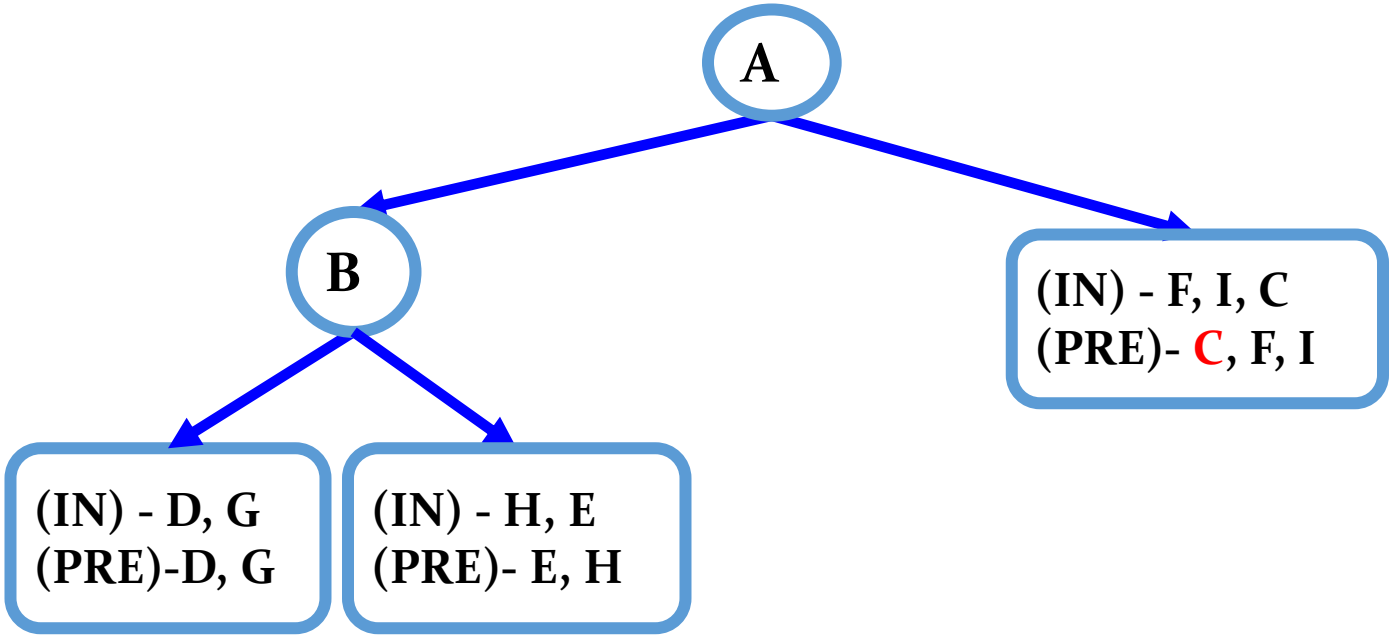
Constructing Binary Tree



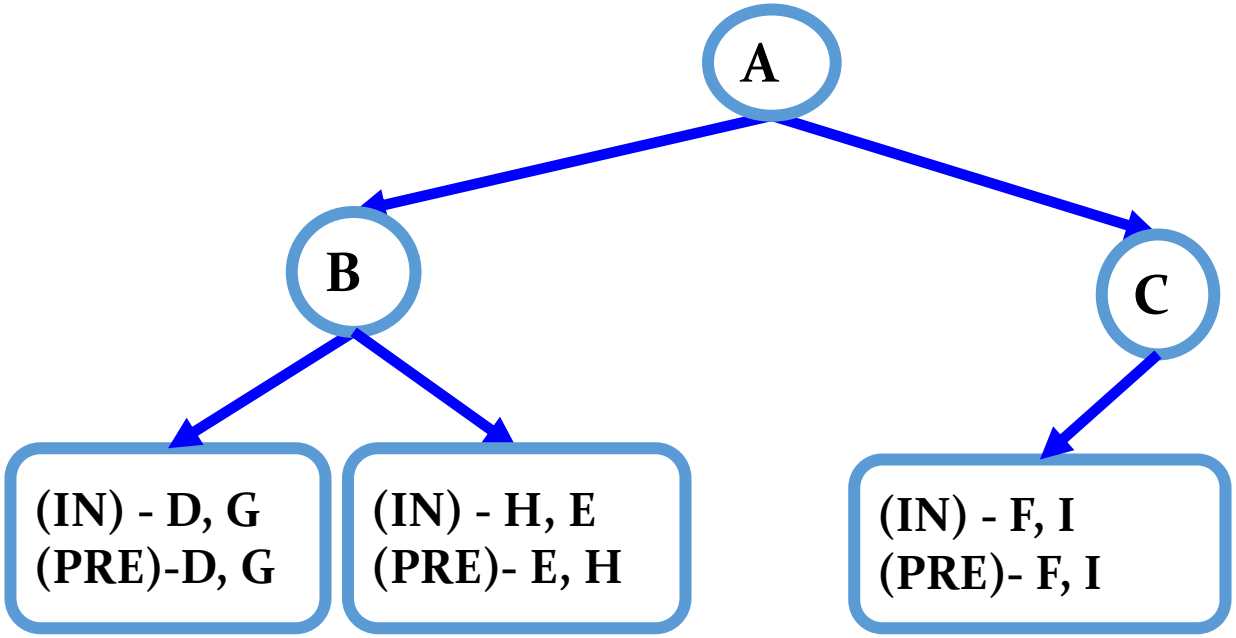
Constructing Binary Tree



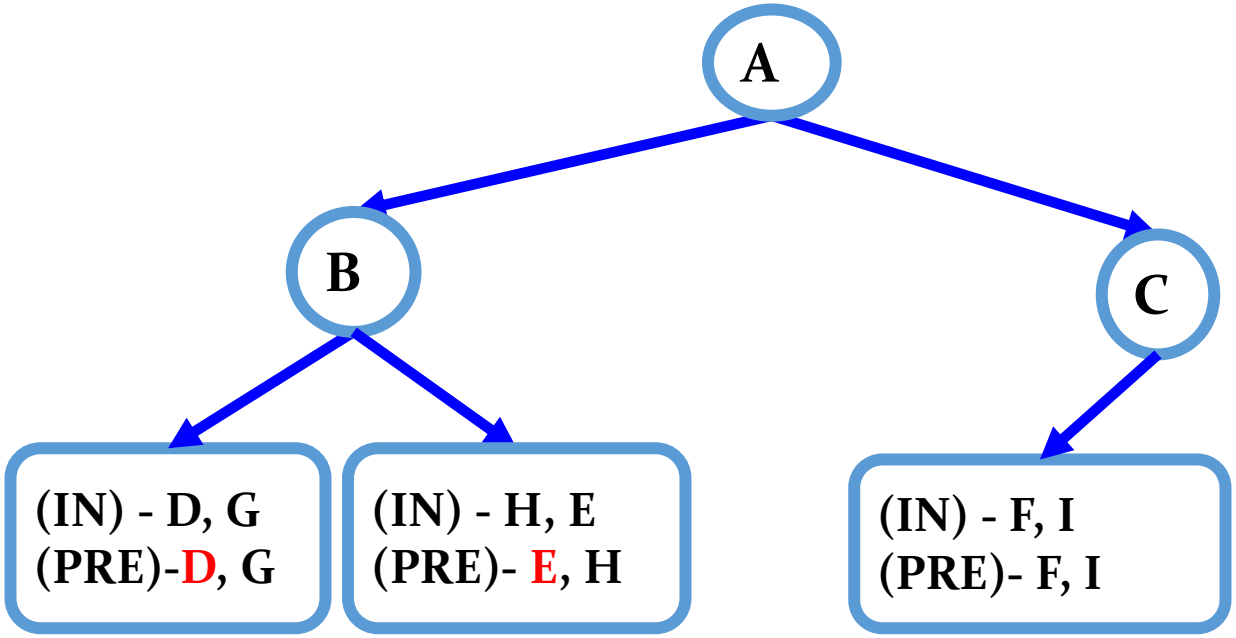
Constructing Binary Tree



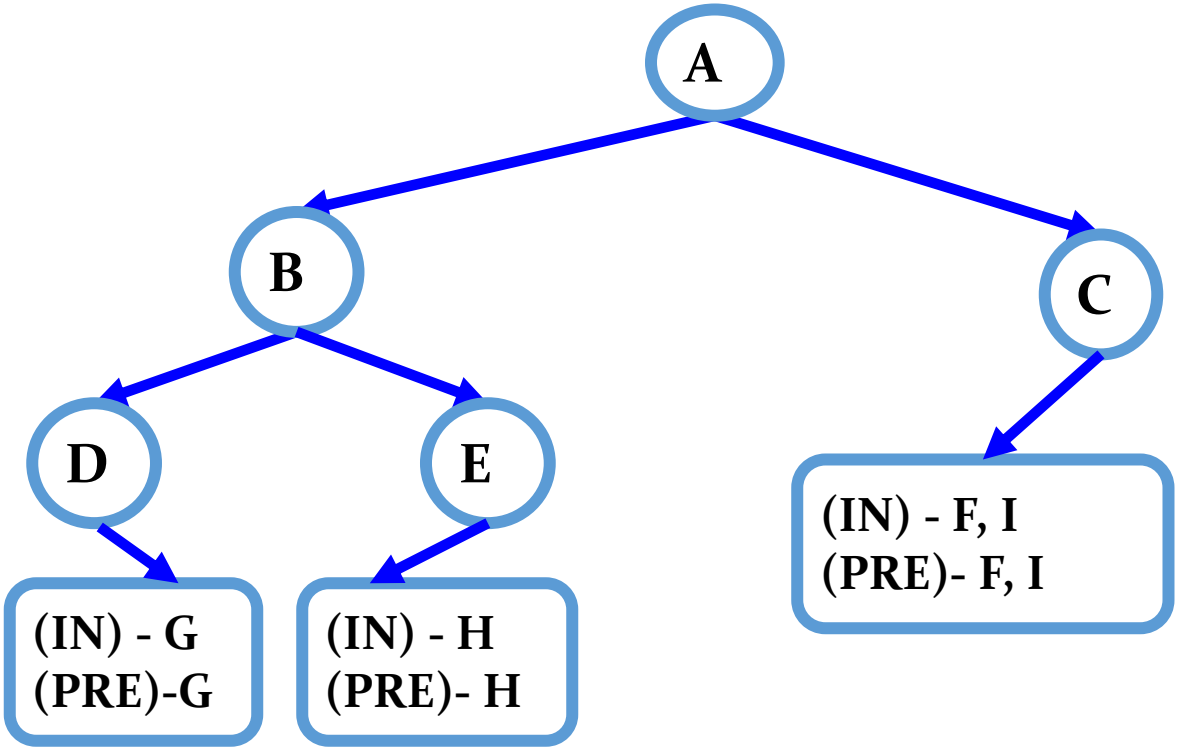
Constructing Binary Tree



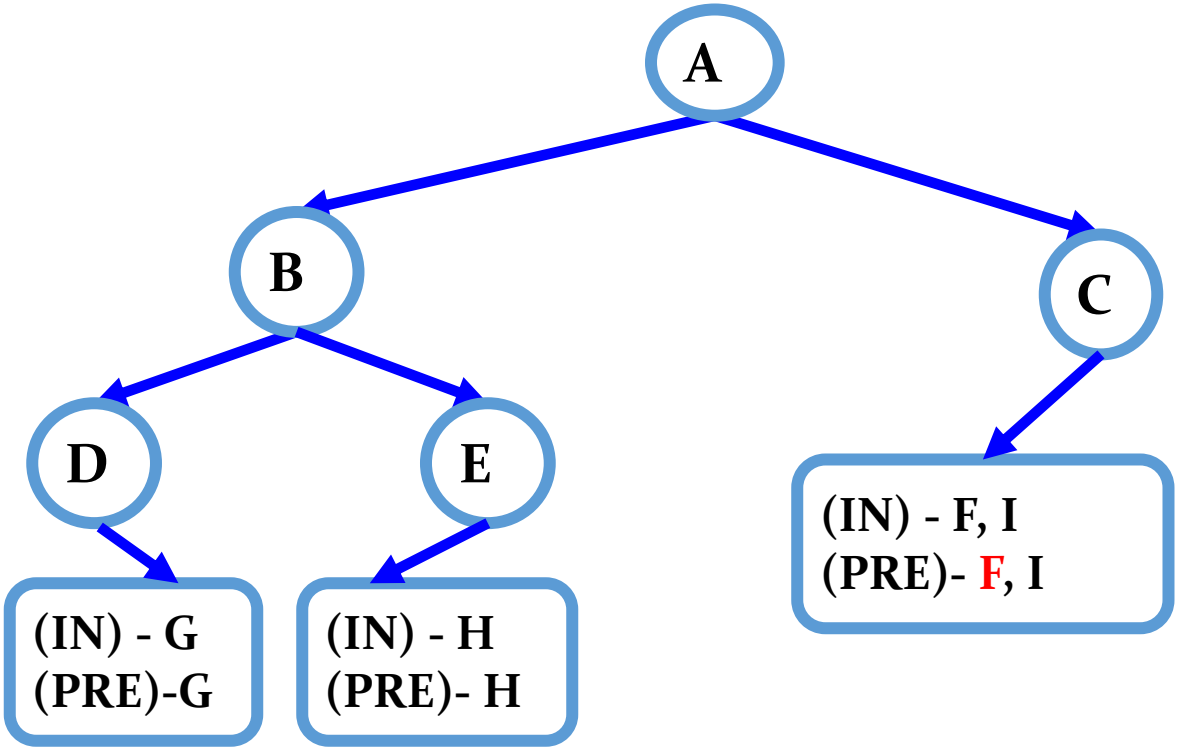
Constructing Binary Tree



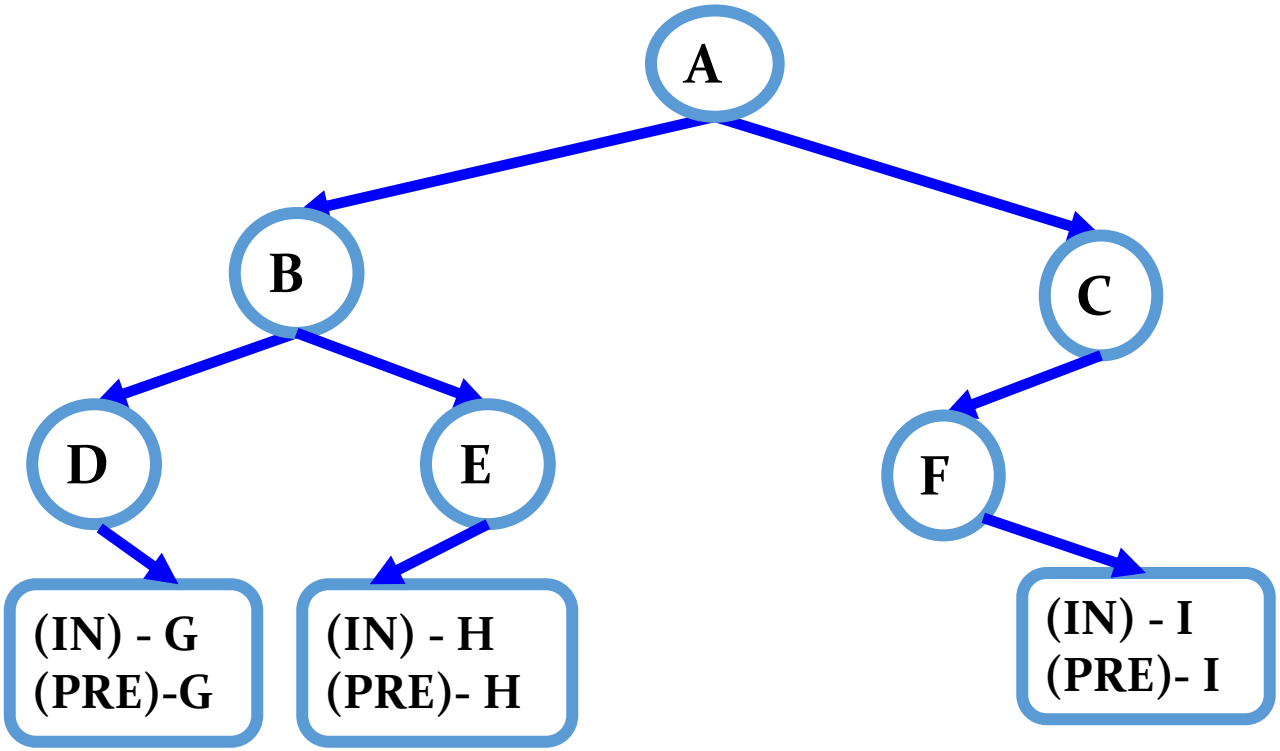
Constructing Binary Tree



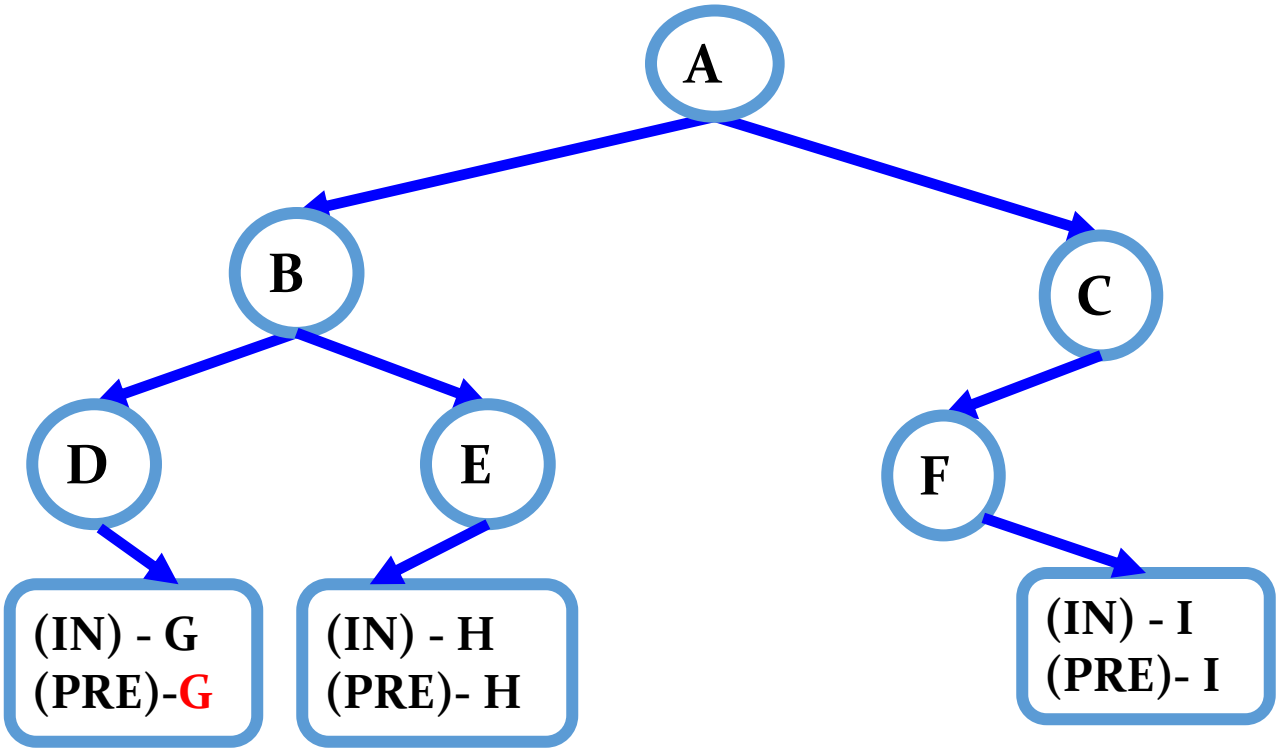
Constructing Binary Tree



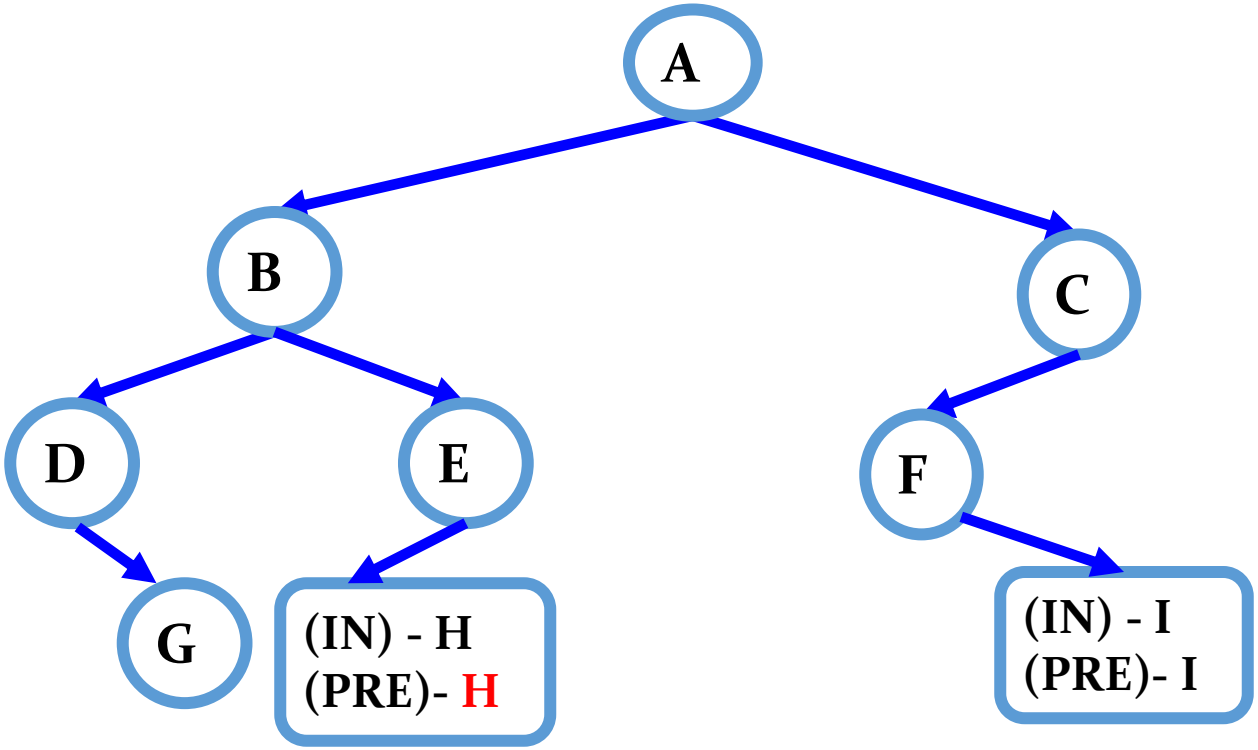
Constructing Binary Tree



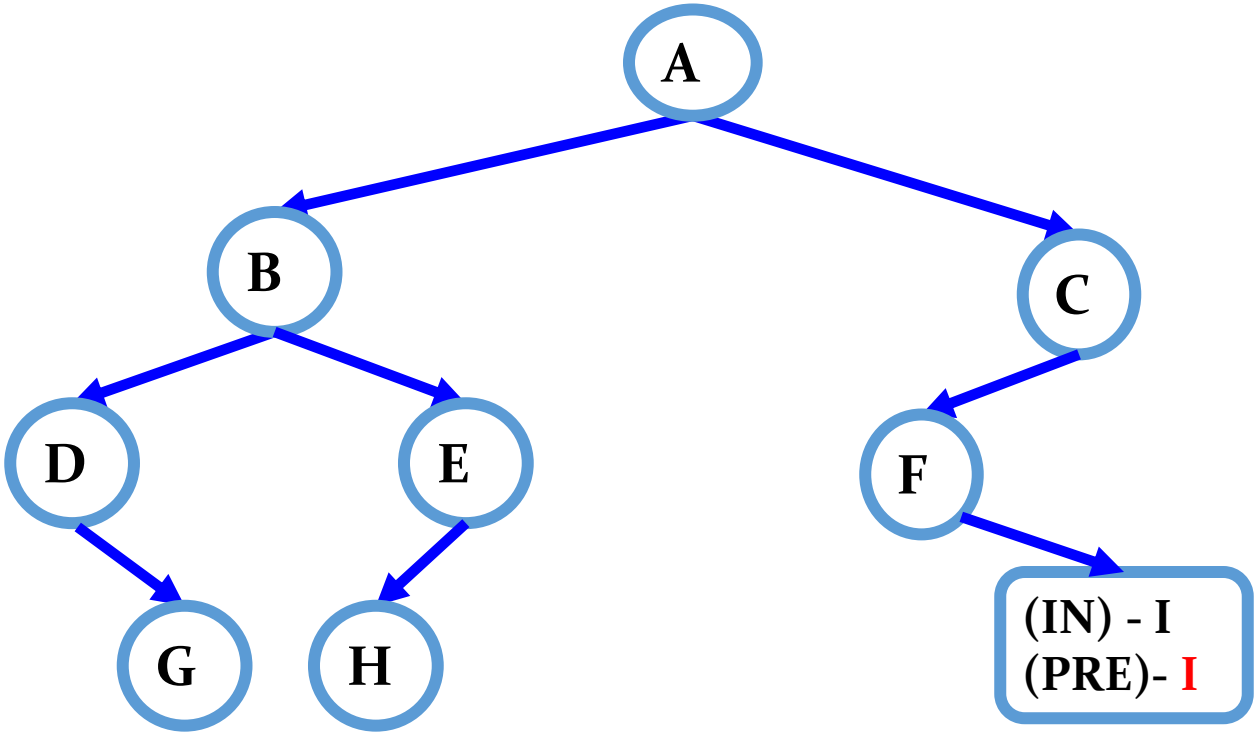
Constructing Binary Tree



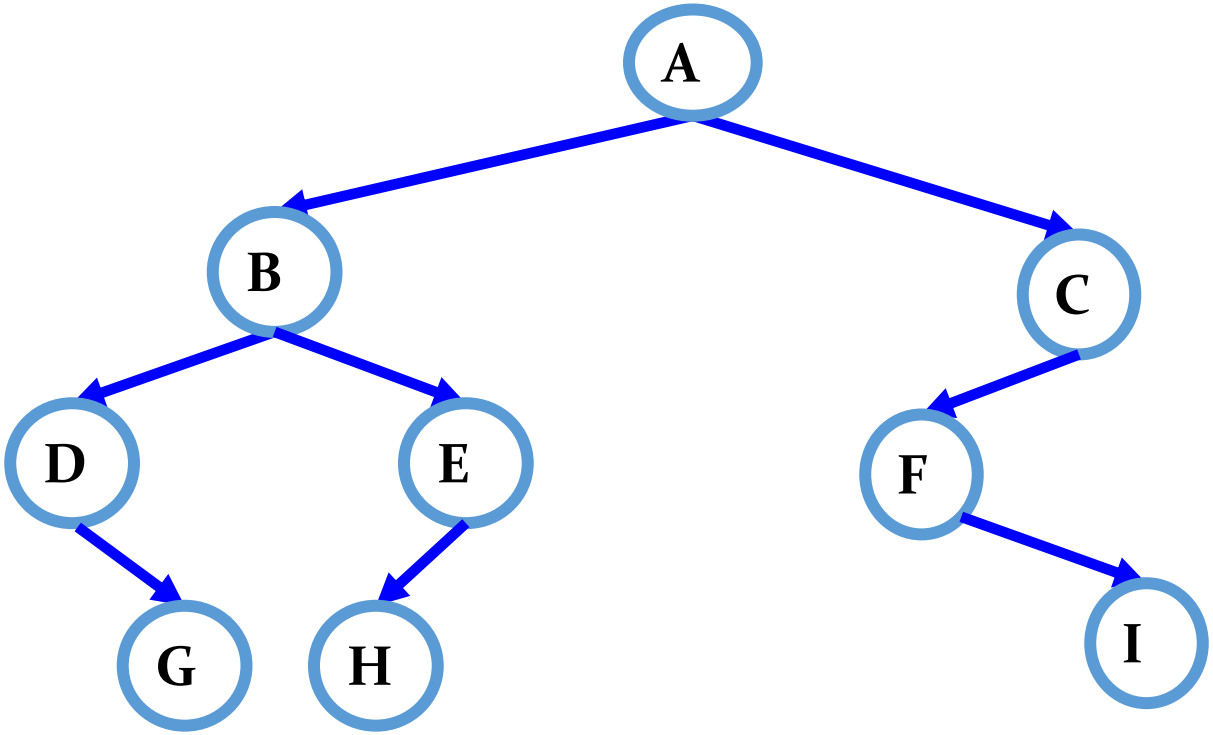
Constructing Binary Tree



Constructing Binary Tree

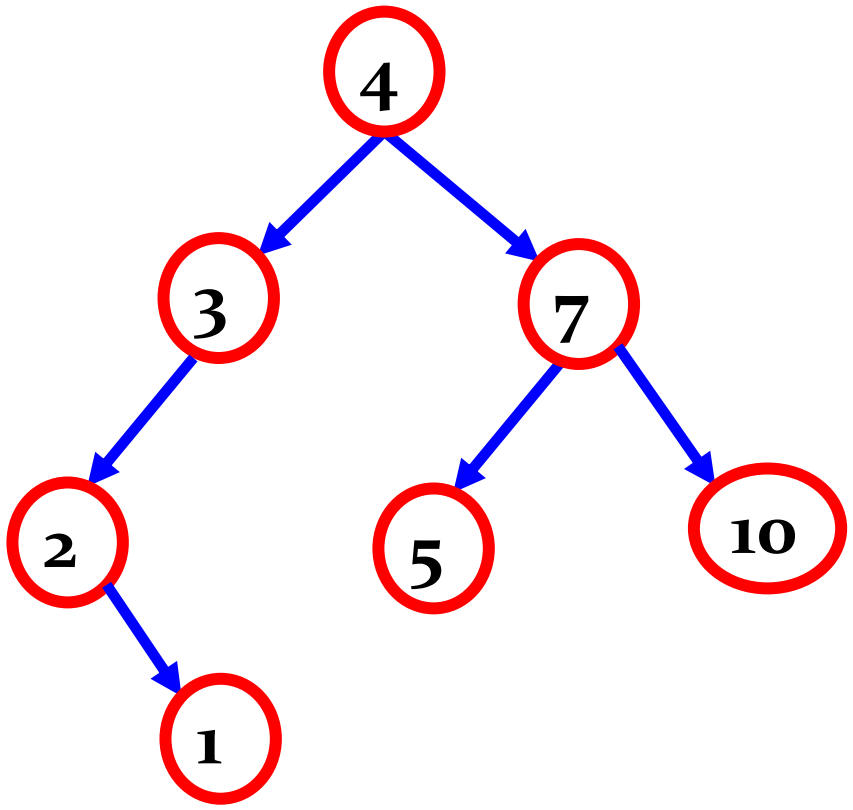


Constructing Binary Tree



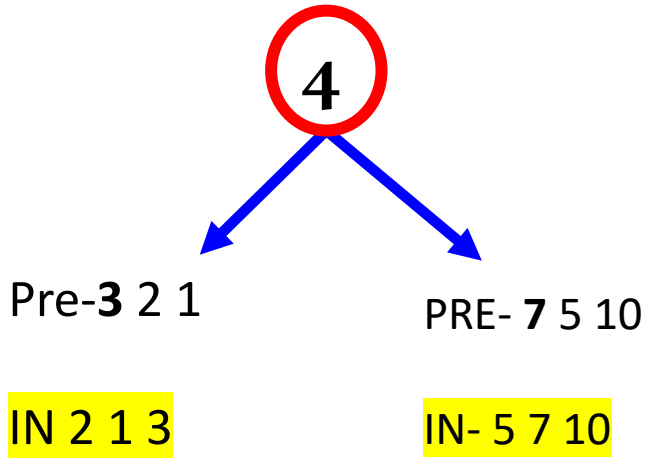
PRE – N L R
4 3 2 1 7 5 10

IN – L N R
2 1 3 4 5 7 10



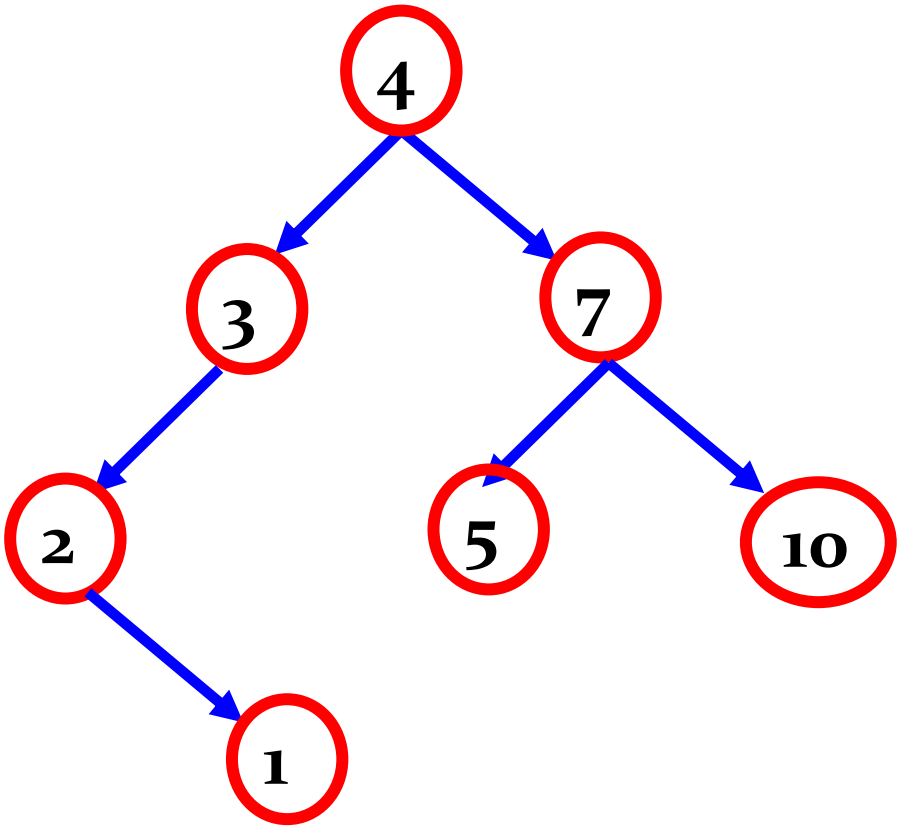
PRE – N L R
4 3 2 1 7 5 10

IN – L N R
2 1 3 4 5 7 10



PRE – N L R
4 3 2 1 7 5 10

IN – L N R
2 1 3 4 5 7 10



PRE – N L R
4 3 2 1 7 5 10

IN – L N R
2 1 3 4 5 7 10

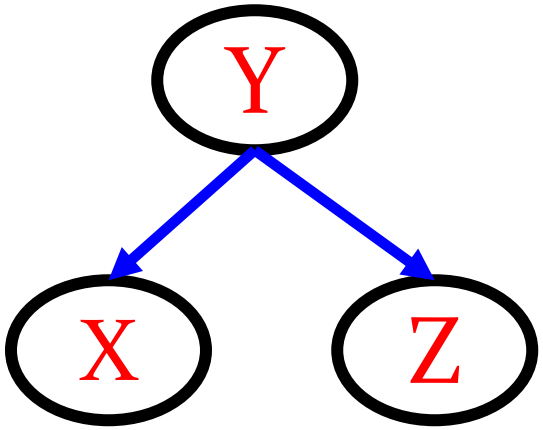
Binary Search Tree

The value at any node,

- Greater than every value in left subtree
- Smaller than every value in right subtree

– Example

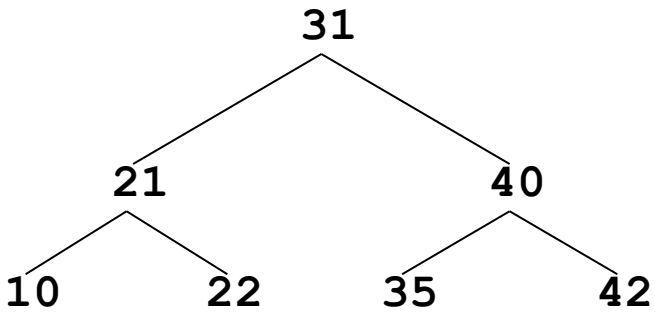
- Y > X
- Y < Z



Binary Search Tree

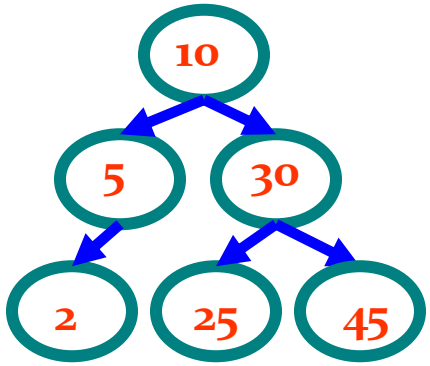
- Values in left sub tree less than parent
- Values in right sub tree greater than parent
- Fast searches in a Binary Search tree, maximum of log n comparisons

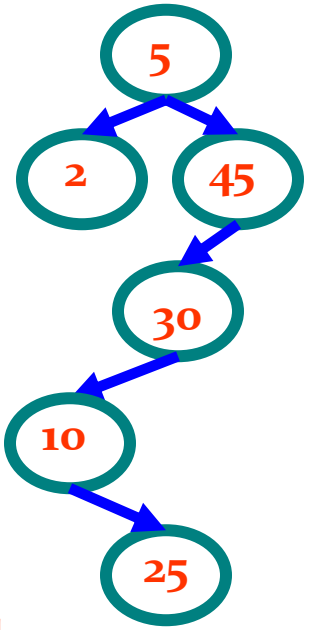
10	21	22	31	35	40	42



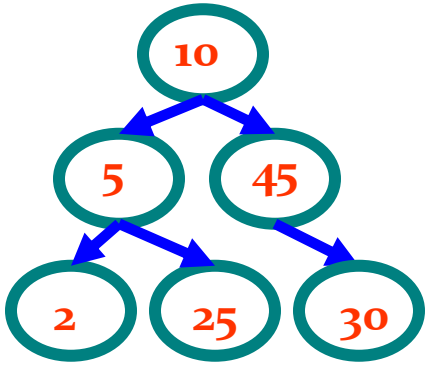
Binary Search Trees

- Examples





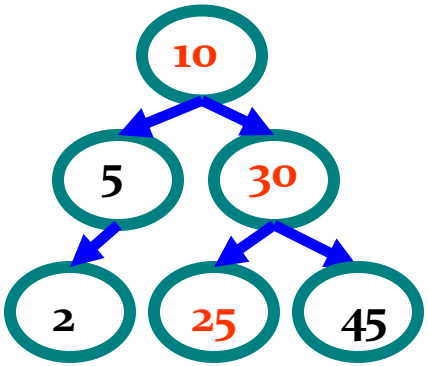
Binary search trees



Not a binary search tree

Example Binary Searches

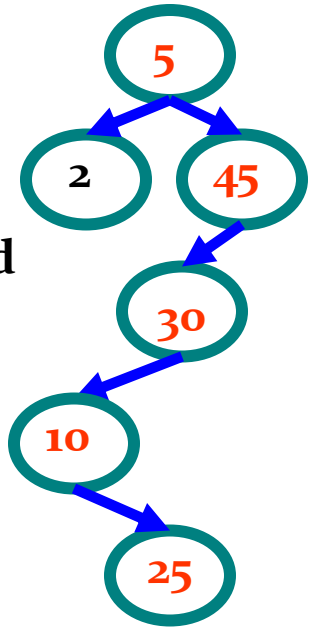
- search (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

30 > 25, left

10 < 25, right

25 = 25, found

Algorithm for Binary Search Tree

- A) compare ITEM with the root node N of the tree
  - 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, i.e. search is successful.
  - ii) we meet an empty sub tree, i.e. the search is unsuccessful.

Binary Tree Implementation

```
  typedef struct node
{
    int data;
    struct node *lc,*rc;
};
```

Iterative Search of Binary Search Tree

```
search()
{
      while (n != NULL)
      {
      if (n->data == item)       // Found it
                  return n;
            if (n->data > item)              // In left subtree
                  n = n->lc;
            else                             // In right subtree
                  n = n->rc;
      }
      return null;
}
```

Recursive Search of Binary Search Tree

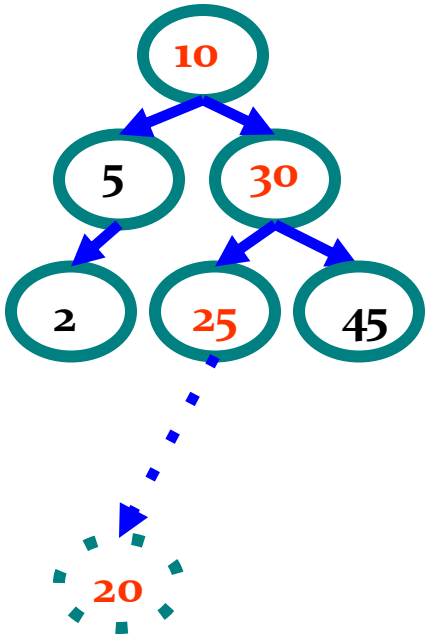
```
Search(node *n, info)
{
    if (n == NULL)                  // Not found
        return( n );
    else if (n->data == item)       // Found it
        return( n );
    else if (n->data > item)        // In left subtree
        return search( n->left, item );
    else                            // In right subtree
        return search( n->right, item );
}
```

Insertion in a Binary Search Tree

- Algorithm
  1. Perform search for value X
  2. Search will end at node Y (if X not in tree)
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y

Insertion in a Binary Search Tree

- Insert (20)



10 < 20, right

30 > 20, left

25 > 20, left

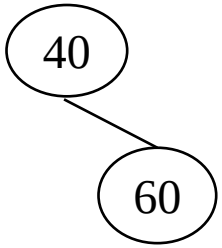
Insert 20 on left

Insertion in a Binary Search Tree

40, 60, 50, 33, 55, 11

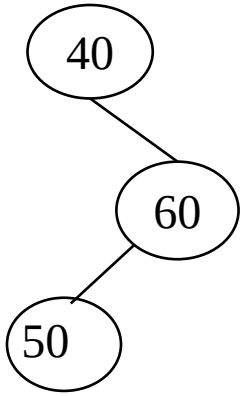
Insertion in a Binary Search Tree

40, 60, 50, 33, 55, 11



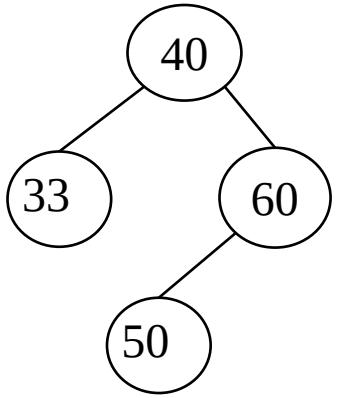
Insertion in a Binary Search Tree

40, 60, 50, 33, 55, 11



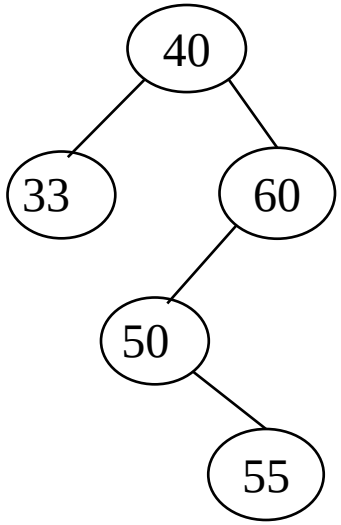
Insertion in a Binary Search Tree

40, 60, 50, 33, 55, 11



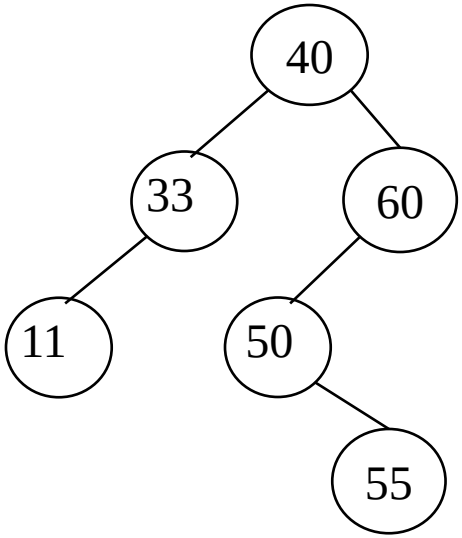
Insertion in a Binary Search Tree

40, 60, 50, 33, 55, 11



Insertion in a Binary Search Tree

40, 60, 50, 33, 55, 11



Deletion in Binary Search Tree

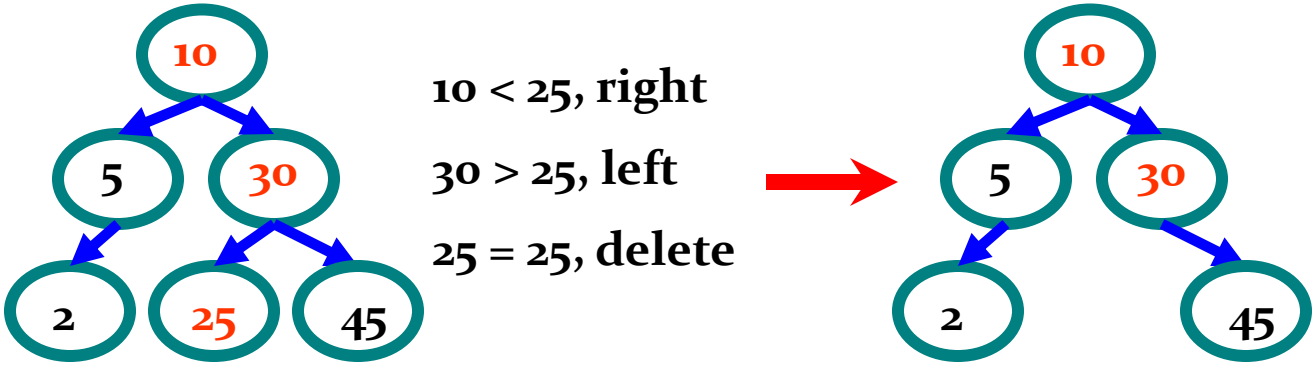
- **Algorithm**
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else //we must delete internal node
     a) Replace with largest value Y on left subtree
        OR smallest value Z on right subtree
     b) Delete replacement value (Y or Z) from subtree

Note :-

- Deletions may unbalance tree

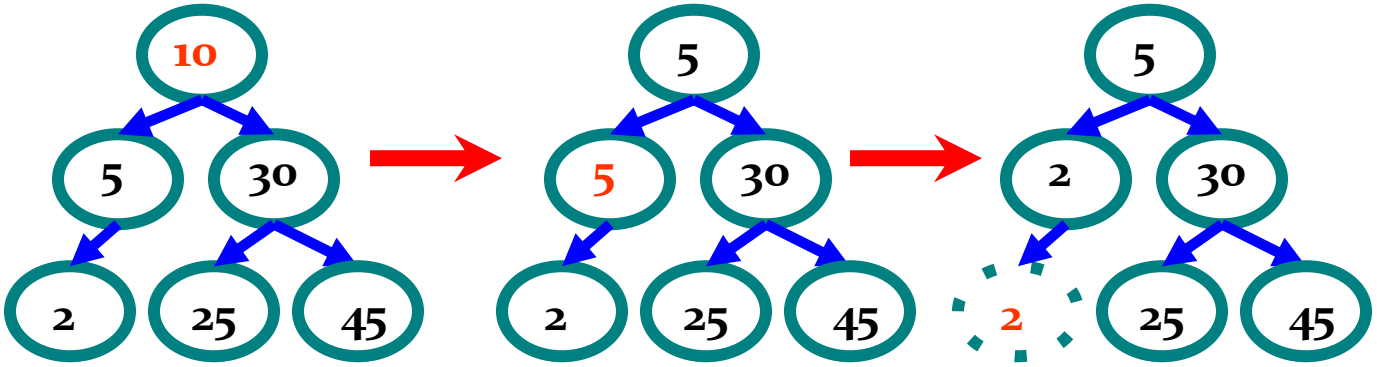
Example Deletion (Leaf)

- Delete (25)



Example Deletion (Internal Node)

- Delete (10)



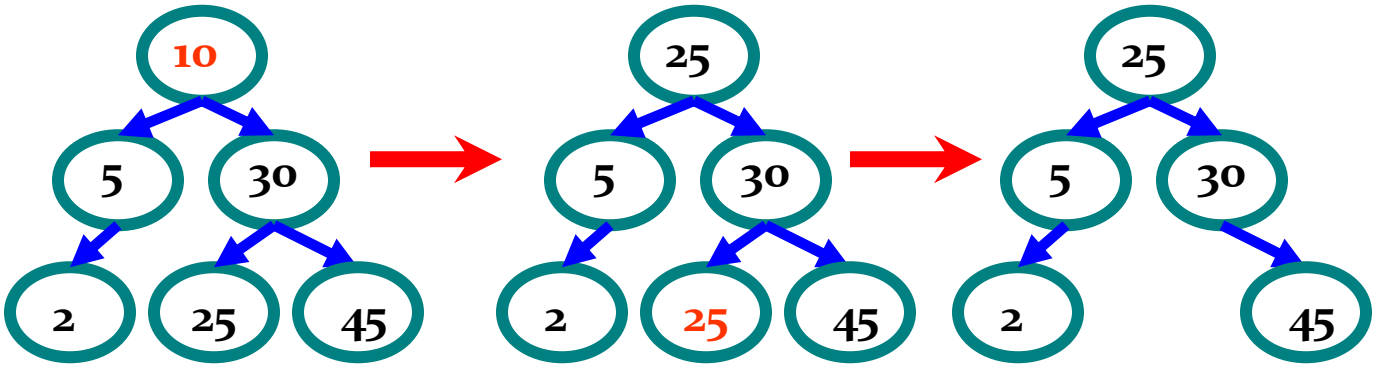
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

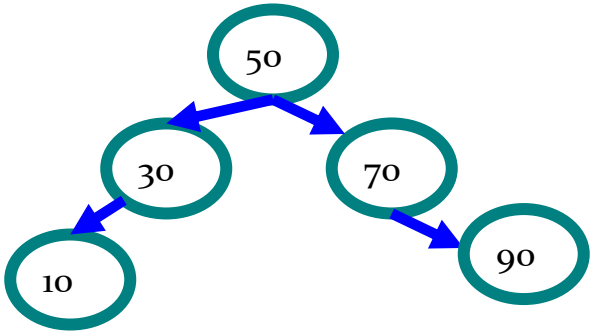
Resulting tree

Insert 50, 60, 70, 30, 40, 10, 90

Delete 60, 40

Insert 50, 60, 70, 30, 40, 10, 90

Delete 60, 40



Binary Search Properties

- Time of search
  - Proportional to height of tree
  - Balanced binary tree
    - O(log(n)) time
  - Degenerate tree
    - O(n) time
    - Like searching linked list / unsorted array

AVL Tree

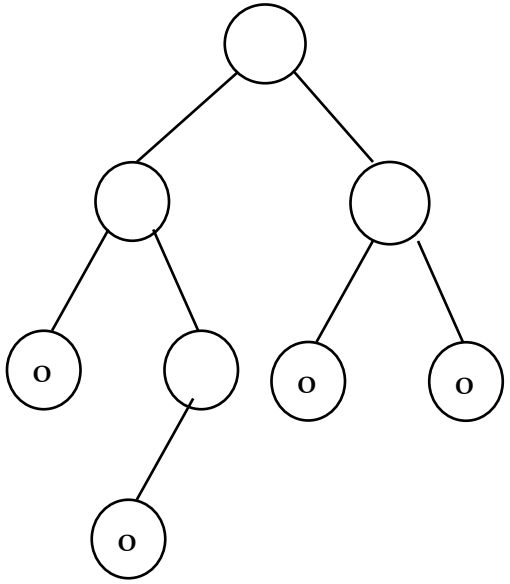
- AVL trees are height-balanced binary search trees
- Balance factor of a node=

  height(left sub tree) - height(right sub tree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right sub tree can differ by no more than 1
  - Store current heights in each node

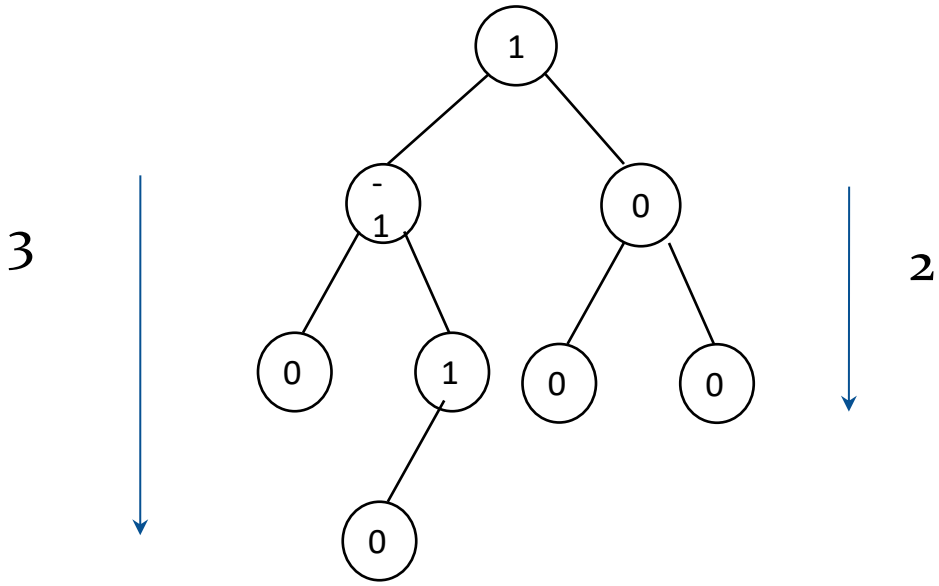
AVL Tree

- A binary tree in which the difference of height of the right and left sub tree of any node is less than or equal to 1 is known as AVL Tree.
- Height of left subtree – height of right subtree
  can be either -1,0,1

AVL Tree

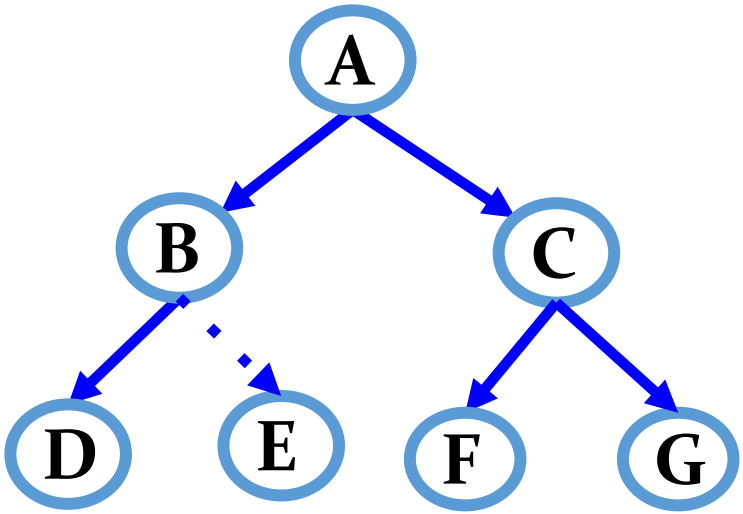


AVL Tree



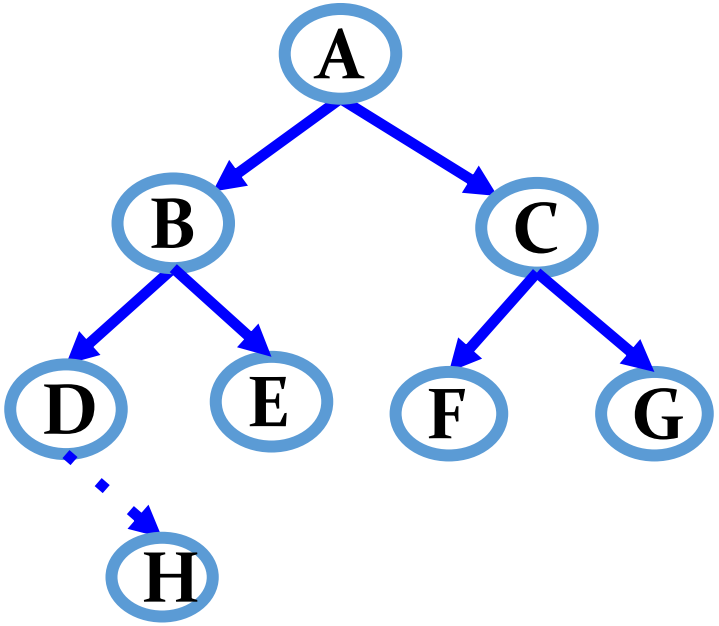
Balanced as LST-RST=1

Insertion in AVL Tree



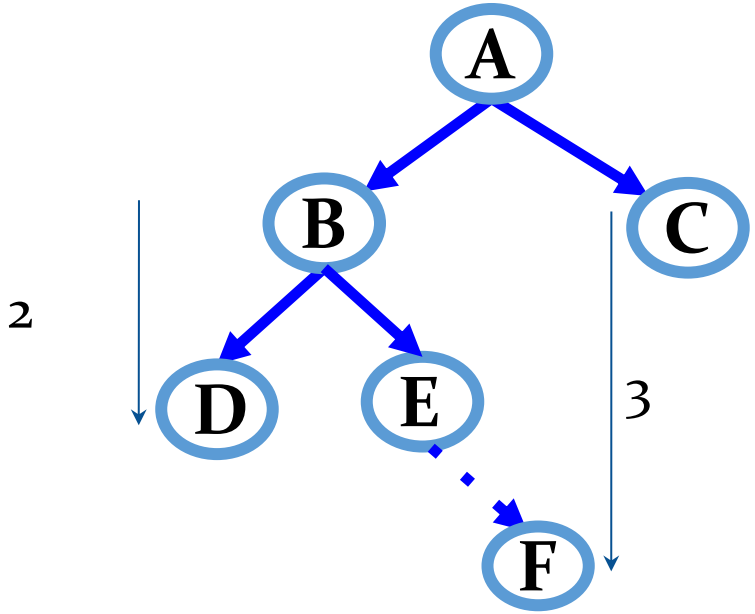
Case 1: the node was either left heavy or right heavy and has become balanced

Insertion in AVL Tree



Case 2: the node was balanced and has now become left or right heavy

Insertion in AVL Tree



Case 3: the node was heavy and the new node has been inserted in the heavy sub tree thus creating an unbalanced sub tree

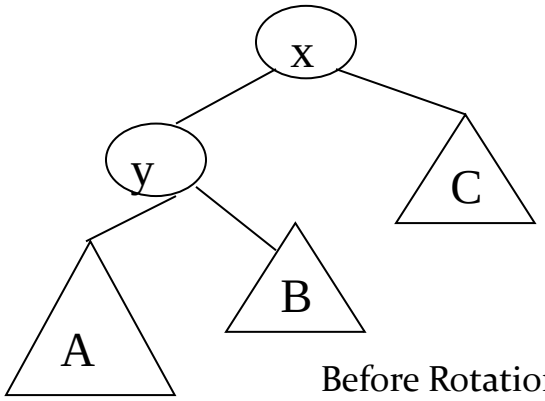
Rebalancing

- When the tree structure changes (e.g., insertion or deletion), we need to transform the tree to restore the AVL tree property.
- This is done using single rotations or double rotations.

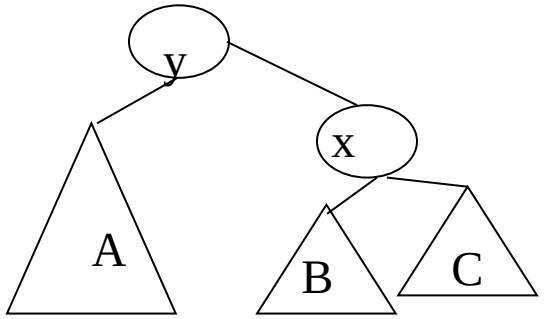
Rotations

- single rotations

e.g. Single Rotation



Before Rotation



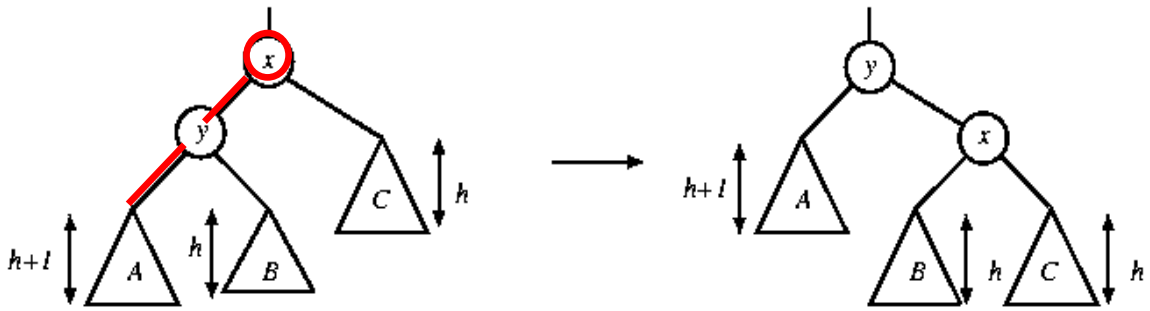
After Rotation

Rotations

- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node x, it means that the heights of left(x) and right(x) differ by exactly 2.
- Rotations will be applied to x to restore the AVL tree property.

Single Rotation

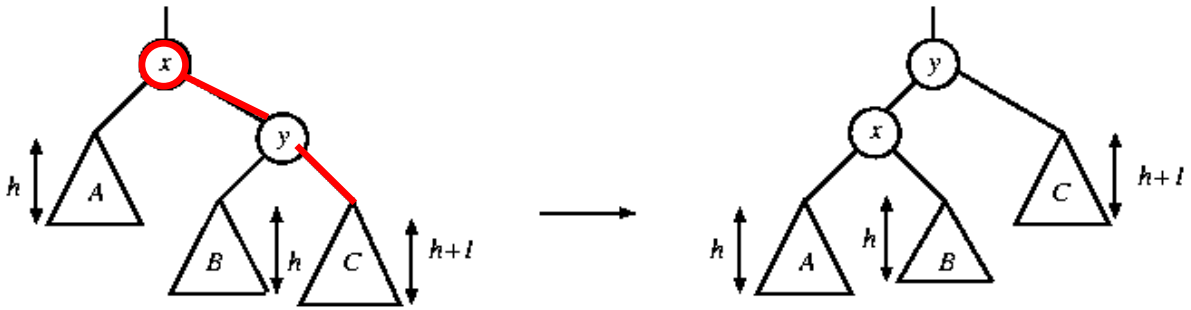
The new item is inserted in the subtree A.
The AVL-property is violated at x
height of left(x) is h+2
height of right(x) is h.



Rotate with left child

Single Rotation

The new item is inserted in the subtree C.
The AVL-property is violated at x.

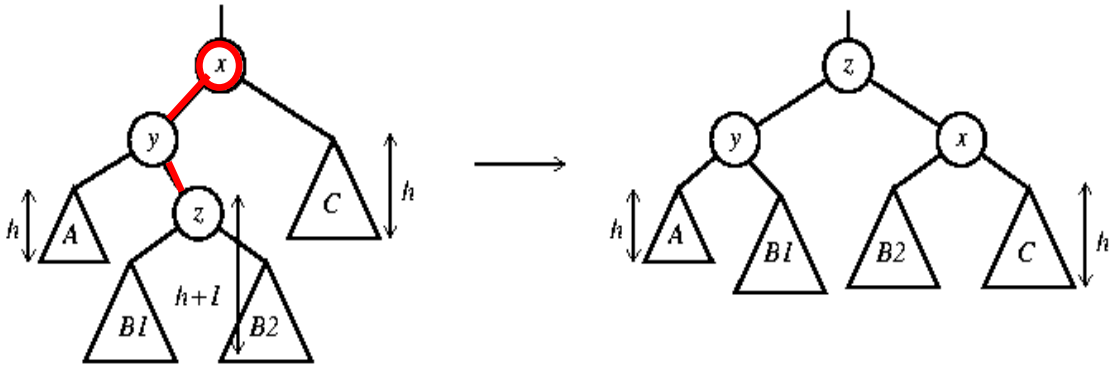


Rotate with right child

Single rotation takes O(1) time.
Insertion takes O(log N) time.

Double Rotation

The new key is inserted in the subtree B1 or B2.
The AVL-property is violated at x.
x-y-z forms a zig-zag shape

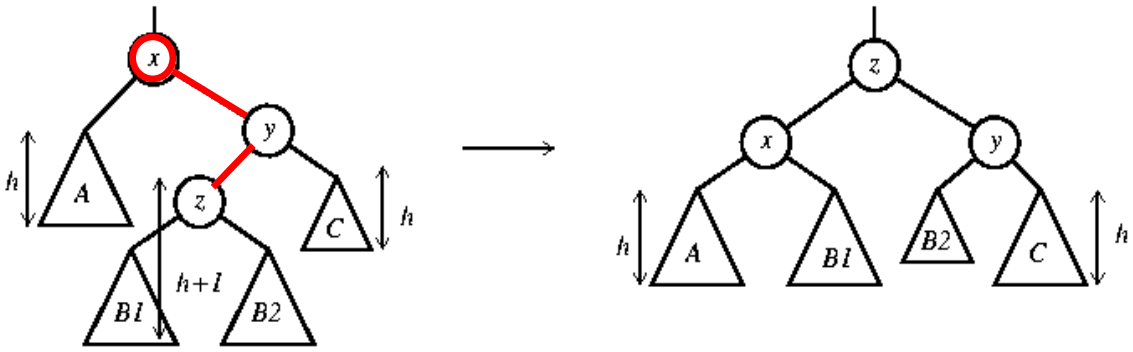


Double rotate with left child

also called left-right rotate

Double Rotation

The new key is inserted in the subtree B1 or B2.
The AVL-property is violated at x.

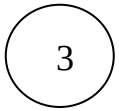


Double rotate with right child

also called right-left rotate

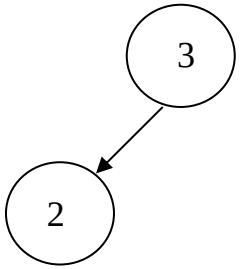
Example

Insert 3,2,1,4,5,6,7



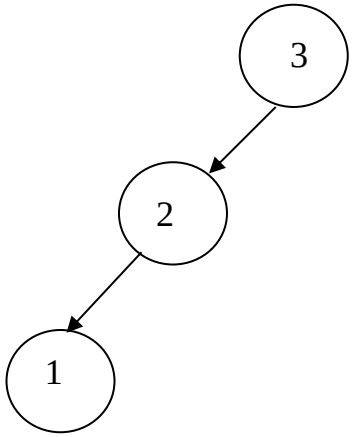
Example

Insert 3,2,1,4,5,6,7



Example

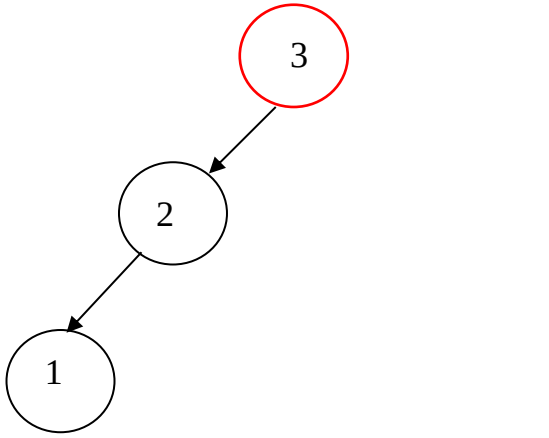
Insert 3,2,1,4,5,6,7



Example

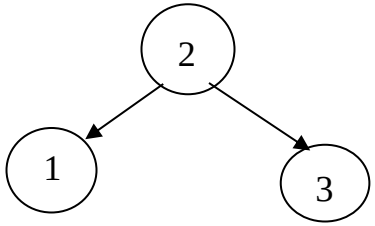
Insert 3,2,1,4,5,6,7

Single rotation



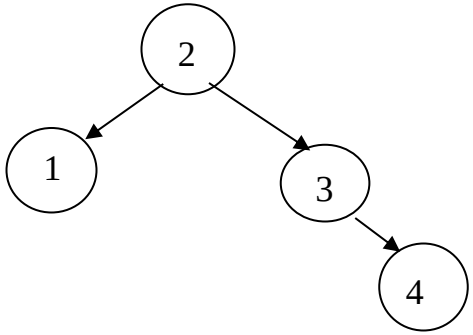
Example

Insert 3,2,1,4,5,6,7



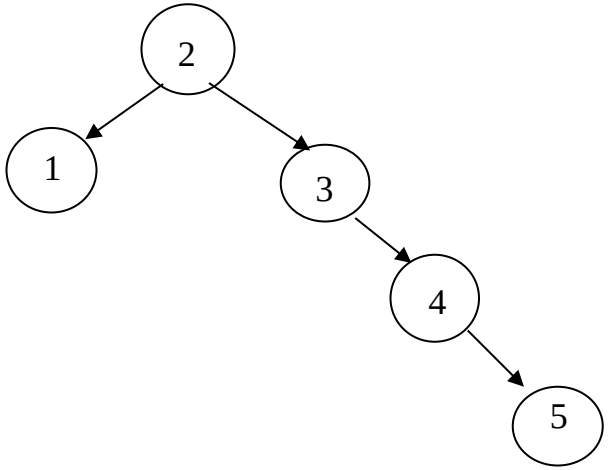
Example

Insert 3,2,1,4,5,6,7



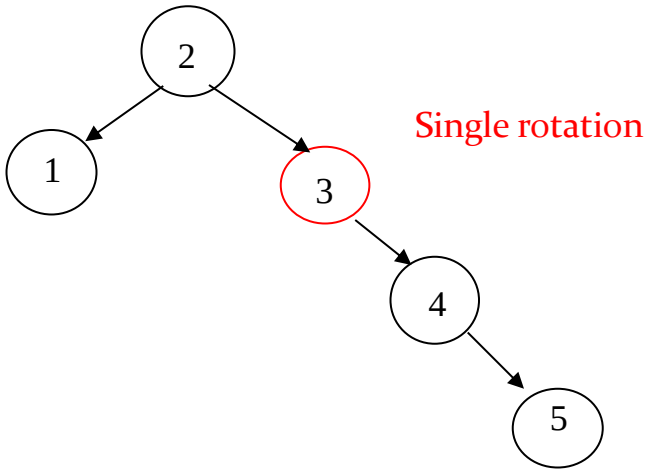
Example

Insert 3,2,1,4,5,6,7



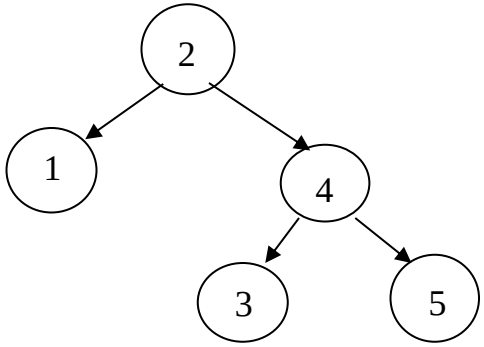
Example

Insert 3,2,1,4,5,6,7



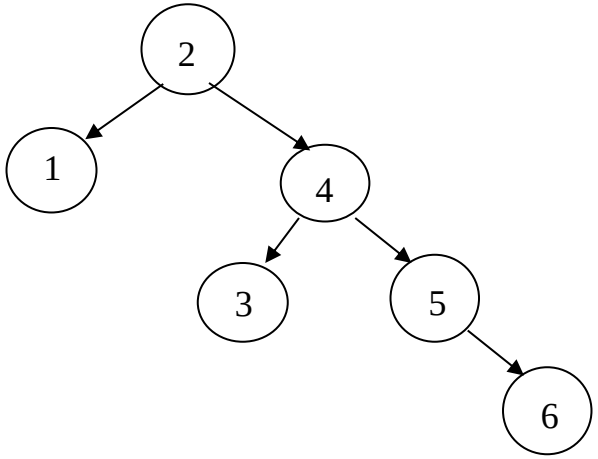
Example

Insert 3,2,1,4,5,6,7



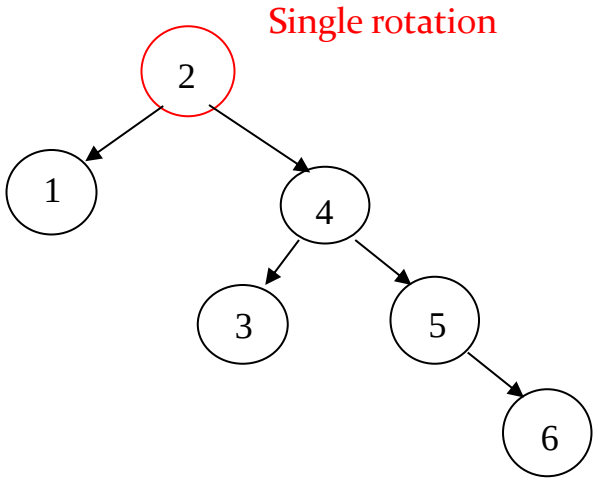
Example

Insert 3,2,1,4,5,6,7



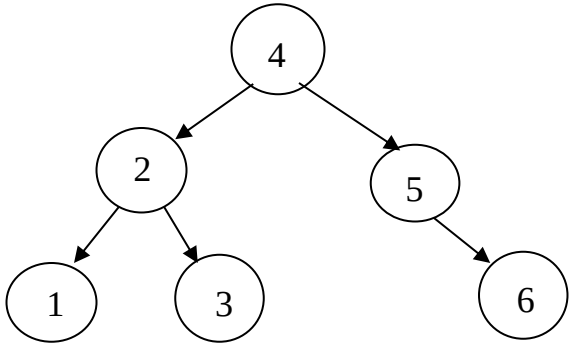
Example

Insert 3,2,1,4,5,6,7



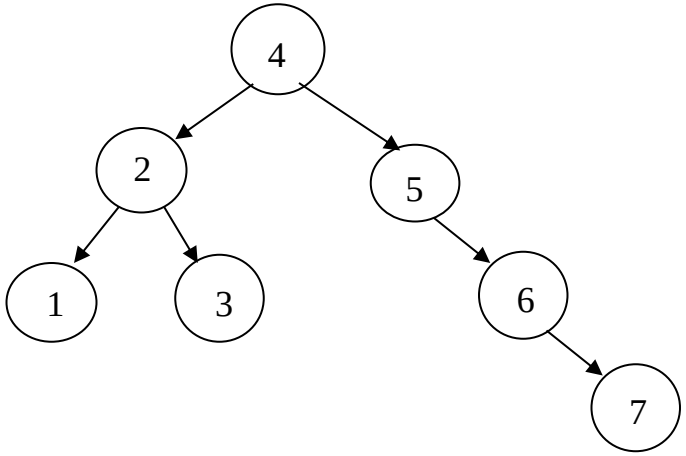
Example

Insert 3,2,1,4,5,6,7



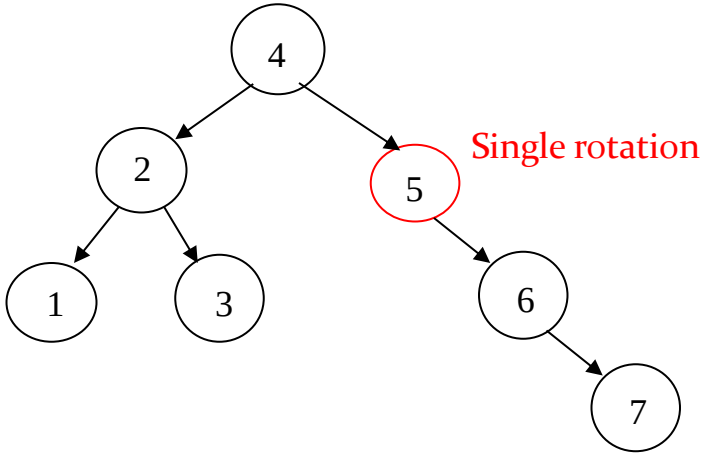
Example

Insert 3,2,1,4,5,6,7



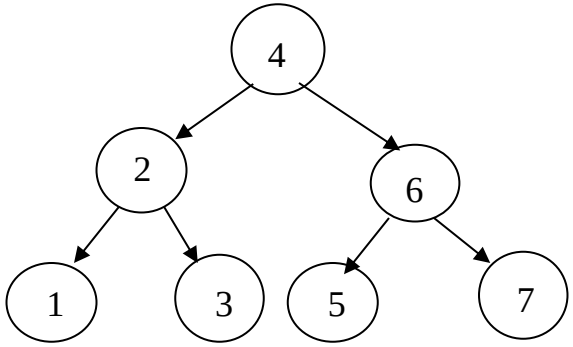
Example

Insert 3,2,1,4,5,6,7



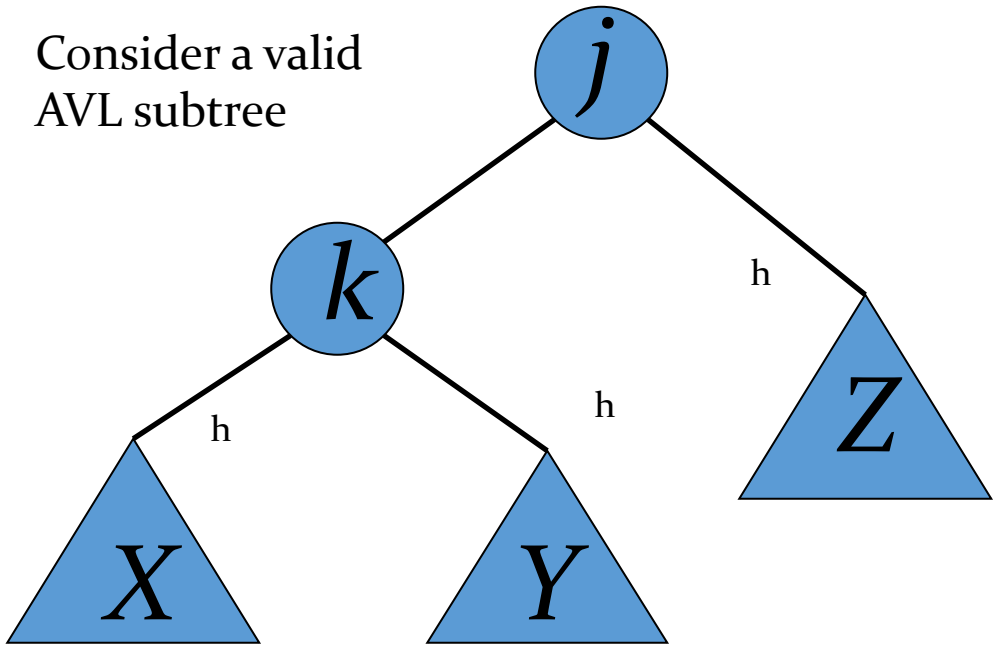
Example

Insert 3,2,1,4,5,6,7

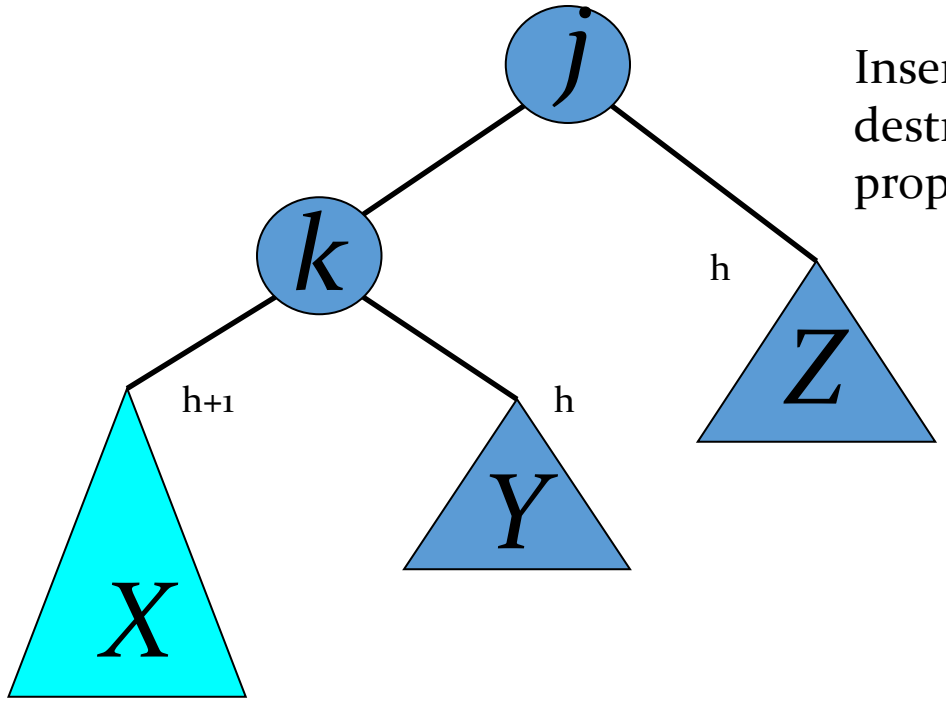


AVL Insertion: Outside Case

Consider a valid AVL subtree

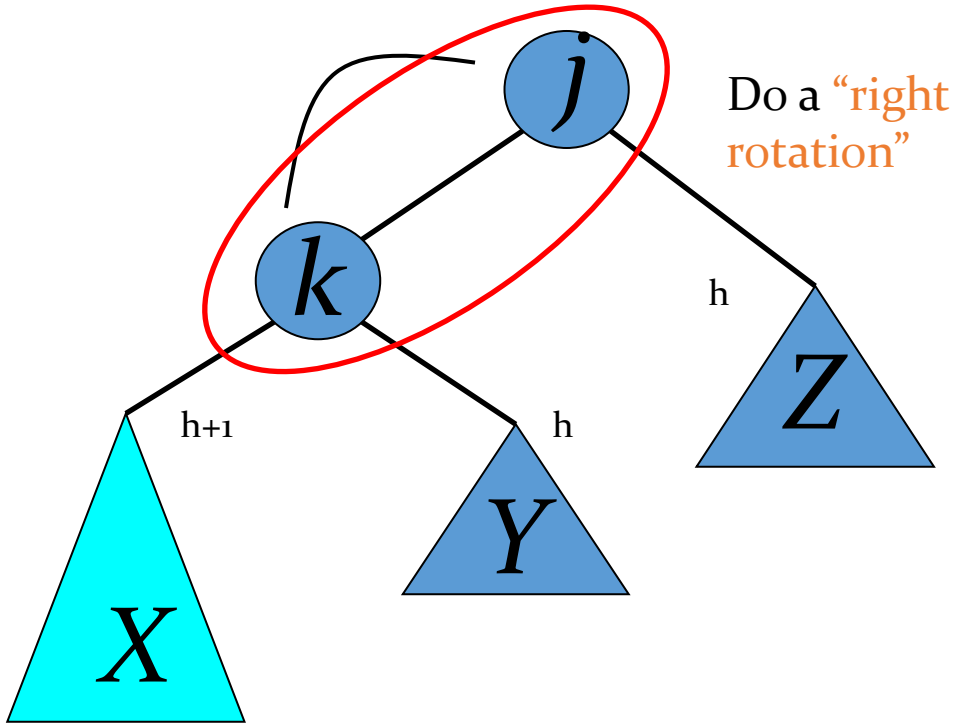


AVL Insertion: Outside Case



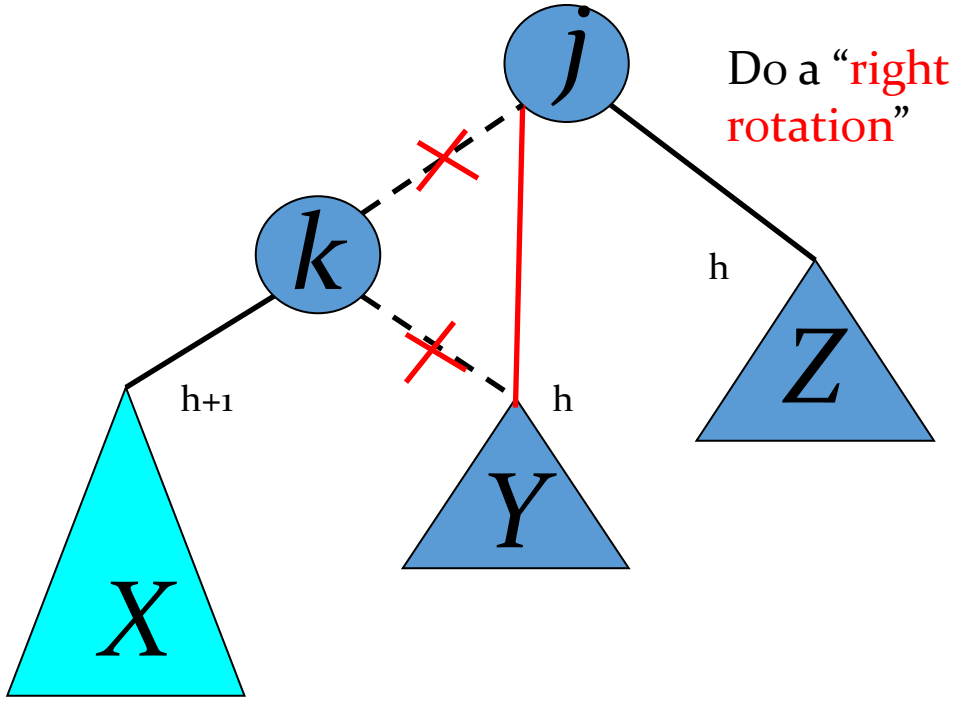
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case

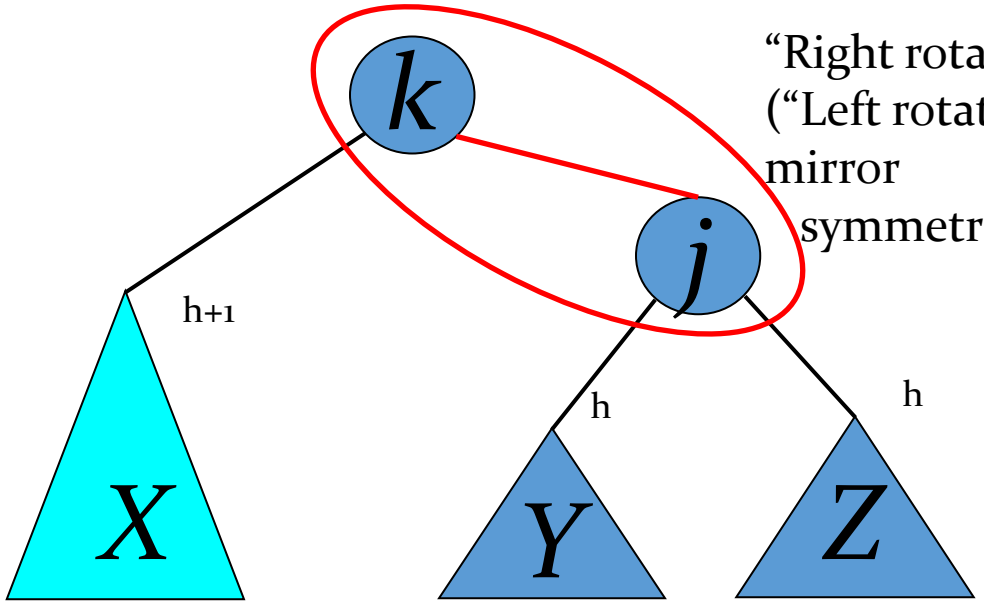


Do a "right rotation"

Single right rotation



Outside Case Completed

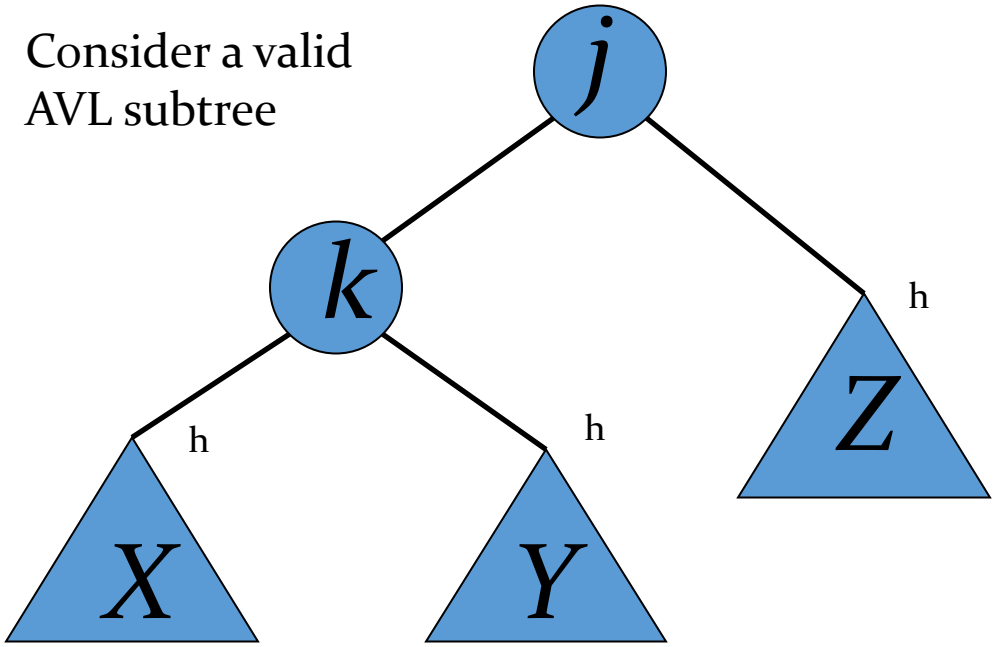


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

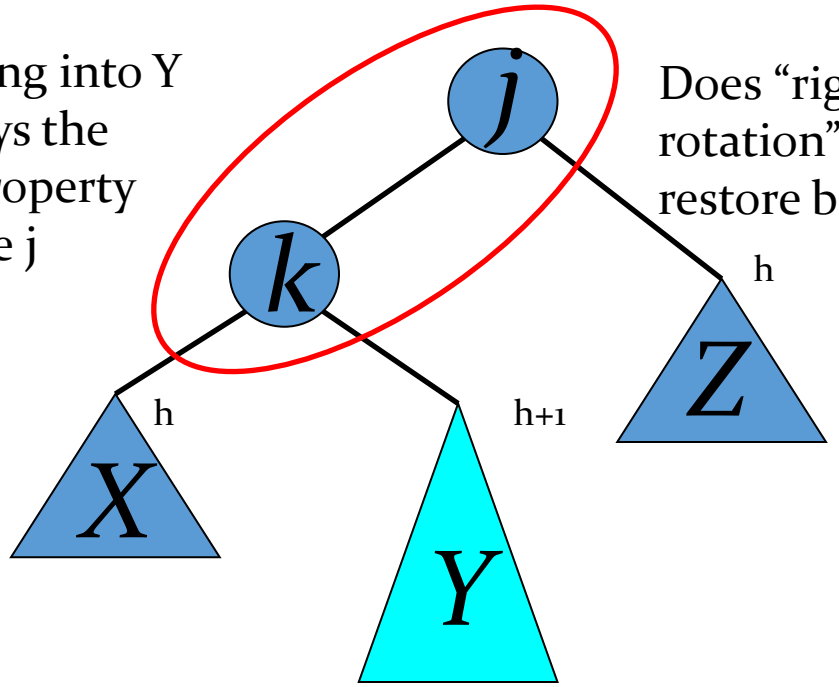
AVL Insertion: Inside Case

Consider a valid AVL subtree



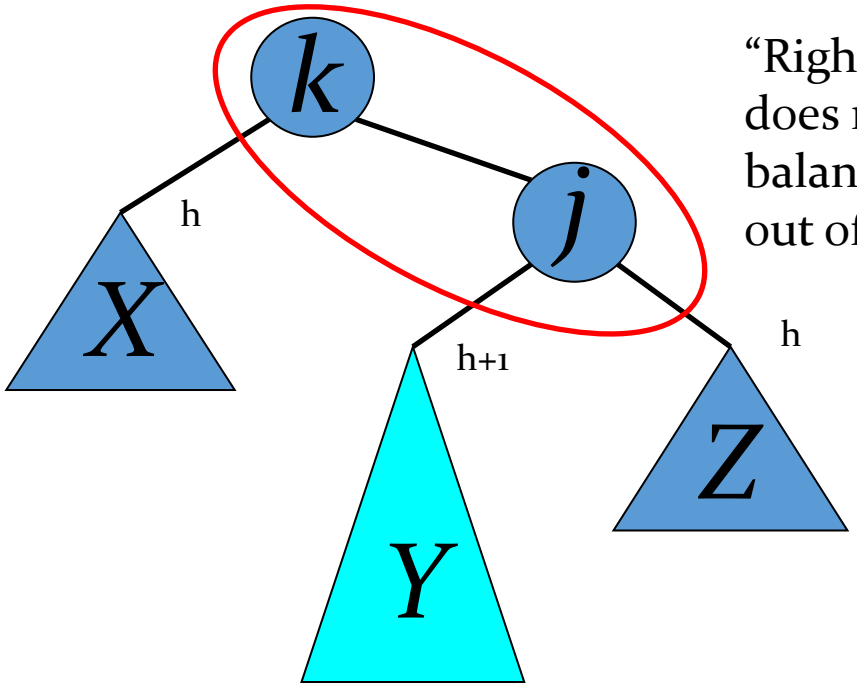
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does "right rotation" restore balance?

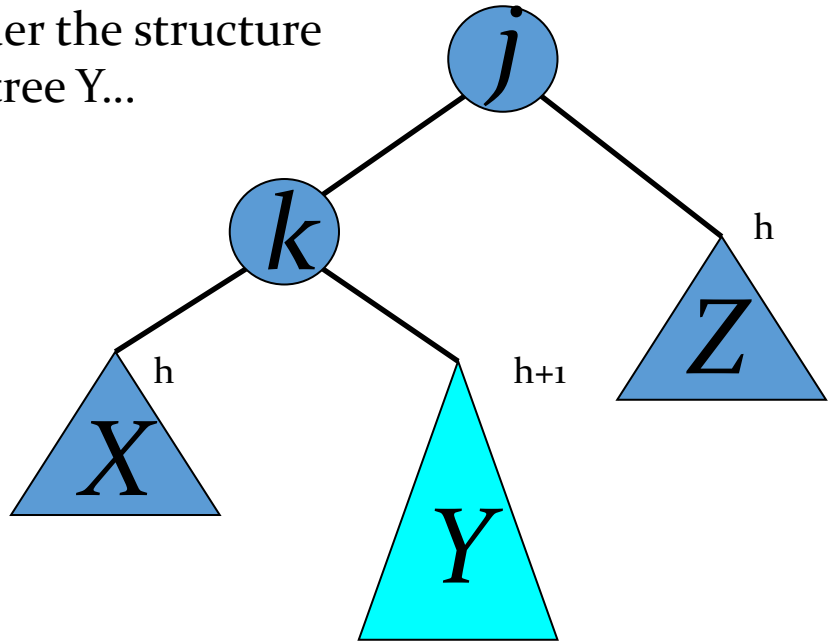
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now k is out of balance

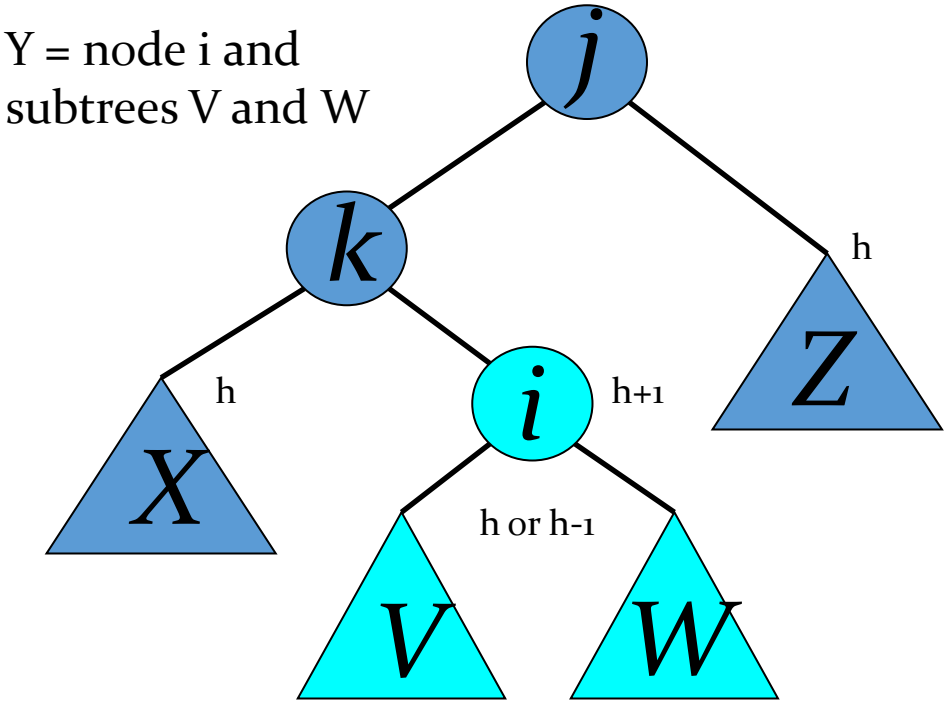
AVL Insertion: Inside Case

Consider the structure of subtree Y...

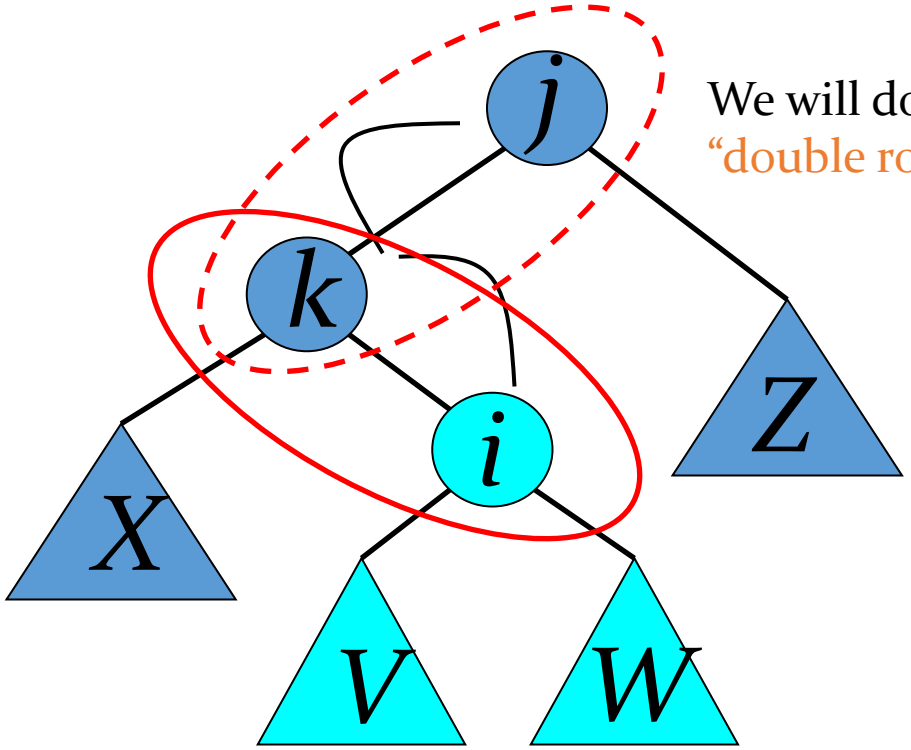


AVL Insertion: Inside Case

Y = node i and
subtrees V and W

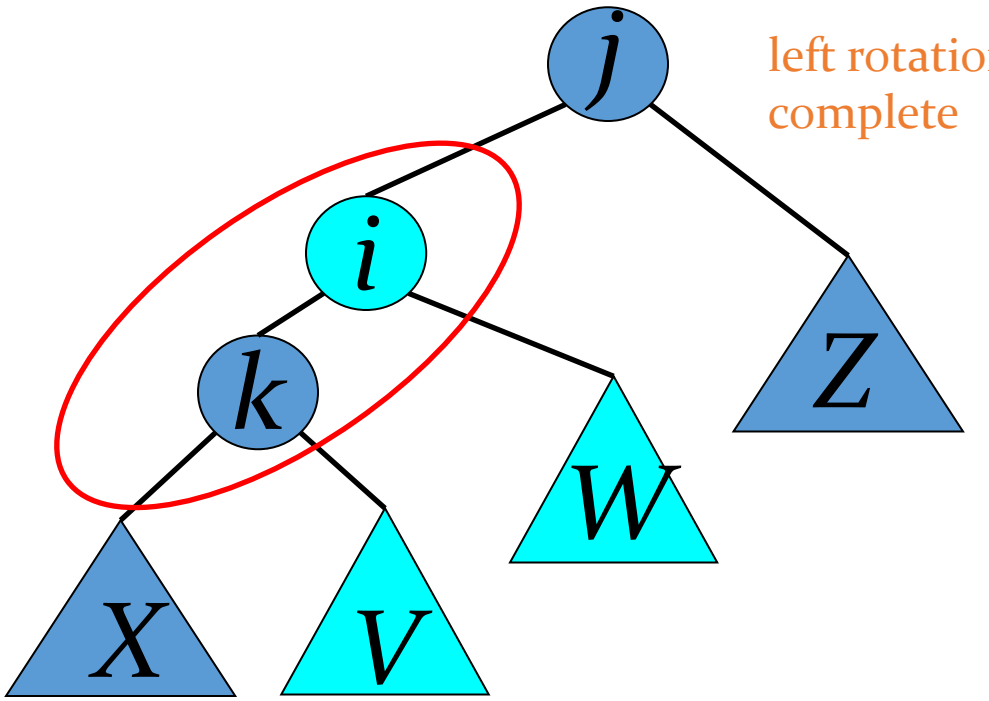


AVL Insertion: Inside Case



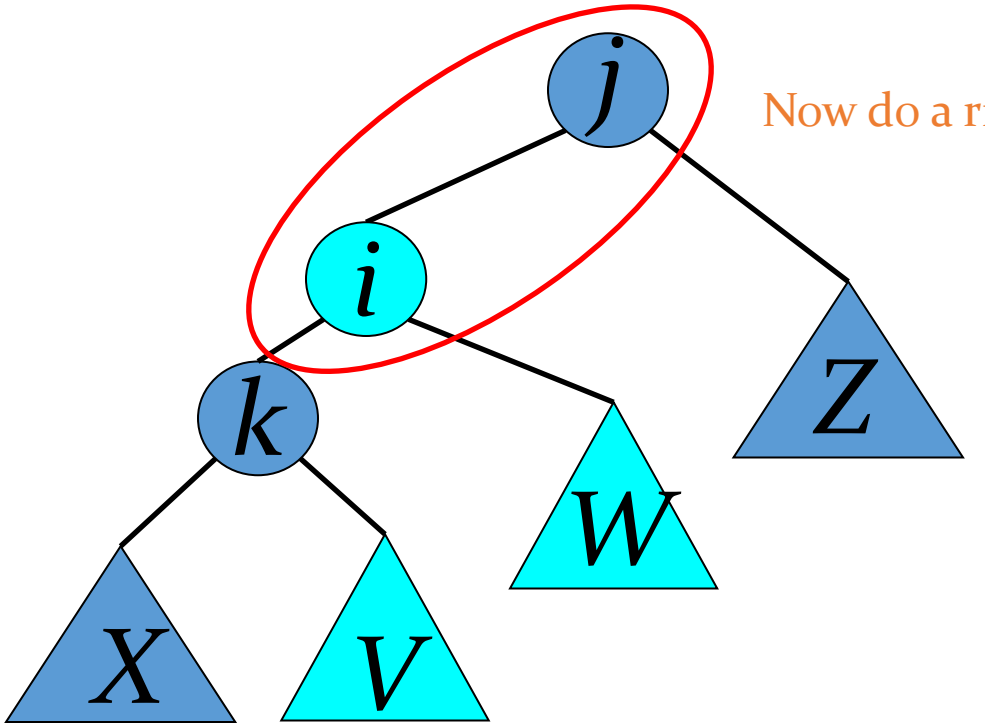
We will do a left-right "double rotation" . . .

Double rotation : first rotation



left rotation
complete

Double rotation : second rotation

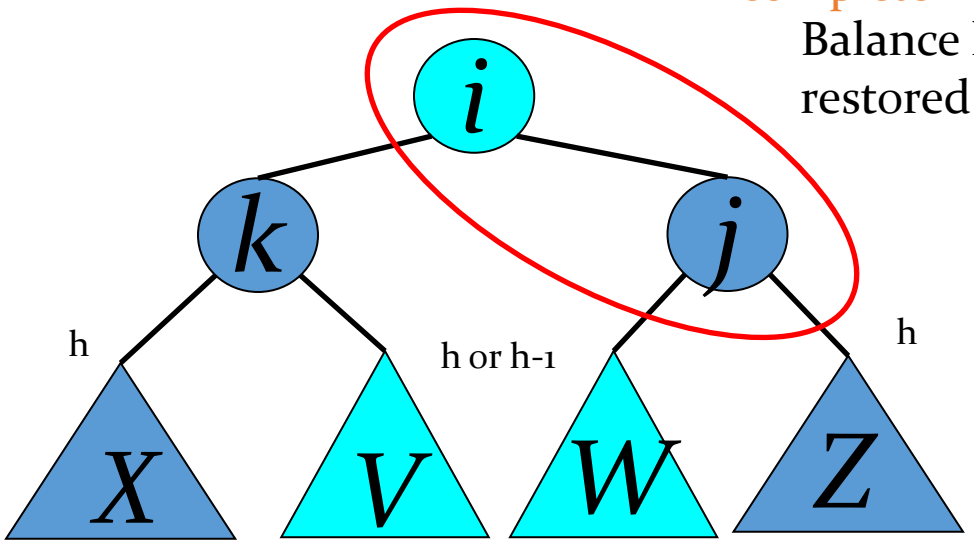


Now do a right rotation

Double rotation : second rotation

right rotation complete

Balance has been restored



Threaded Trees

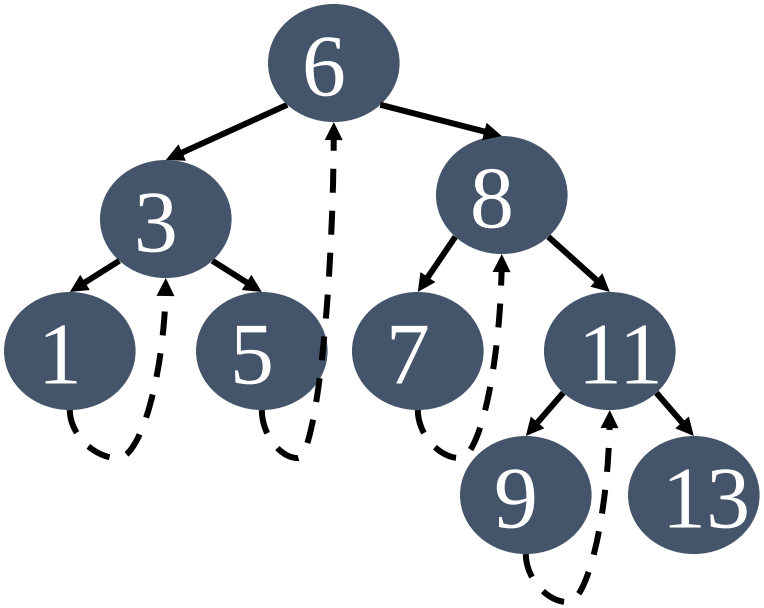
- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers
- We can use these pointers to help us in inorder traversals
- We have the pointers reference the next node in an inorder traversal; called *threads*
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer

Threaded Tree Code

- Example code:

```
class Node {
   Node left, right;
   boolean leftThread, rightThread;
}
```

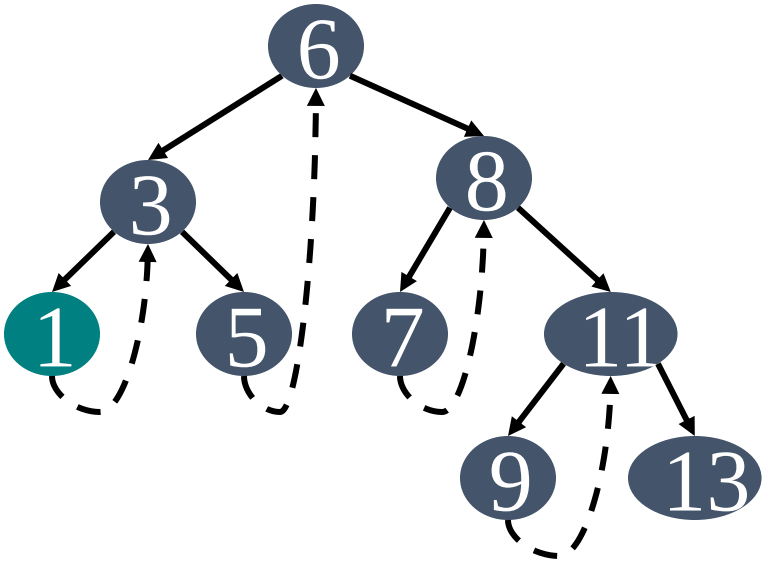
Threaded Tree Example



Threaded Tree Traversal

- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right
- If we follow a link to the right, we go to the leftmost node, print it, and continue

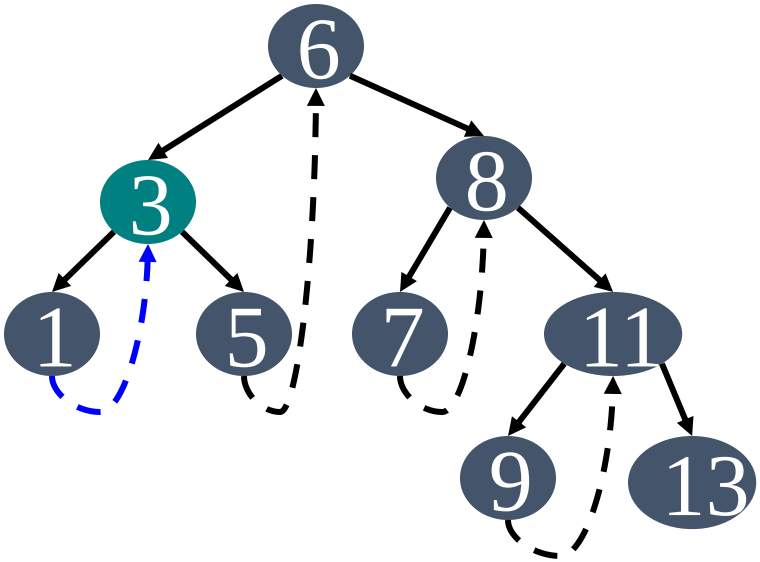
Threaded Tree Traversal



Output
1

Start at leftmost node, print it

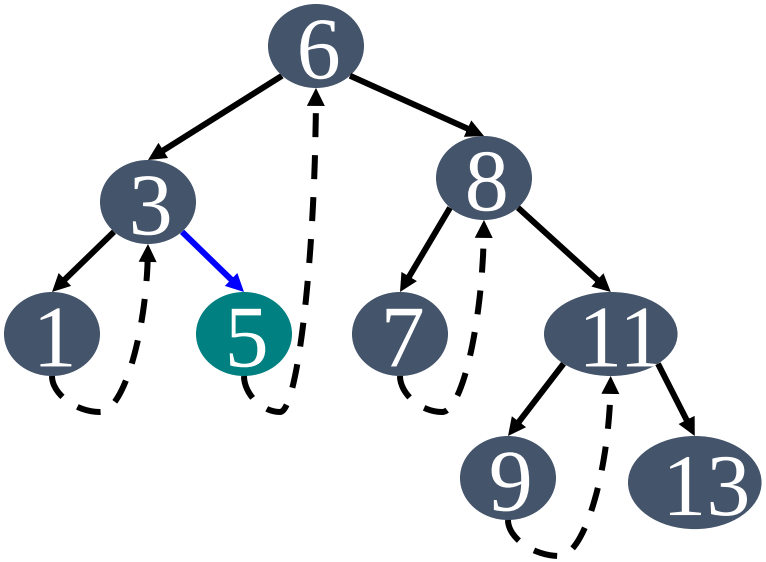
Threaded Tree Traversal



Output
1
3

Follow thread to right, print node

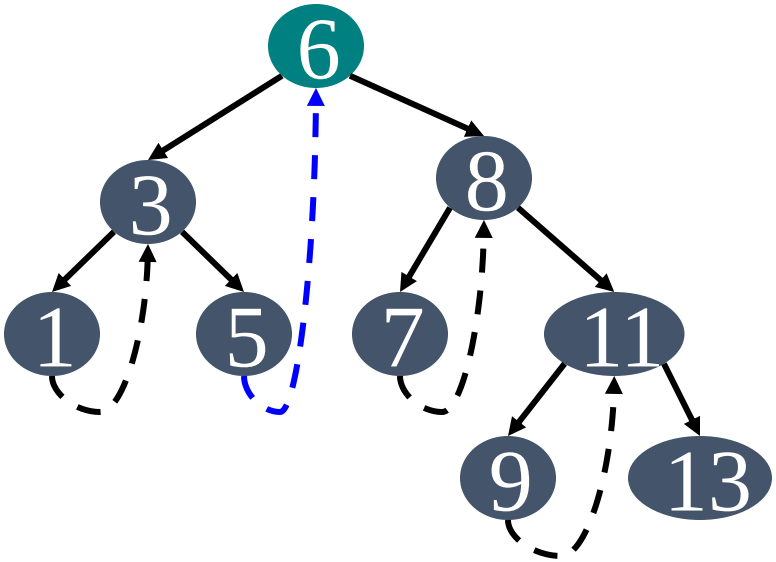
Threaded Tree Traversal



Output
1
3
5

Follow link to right, go to leftmost node and print

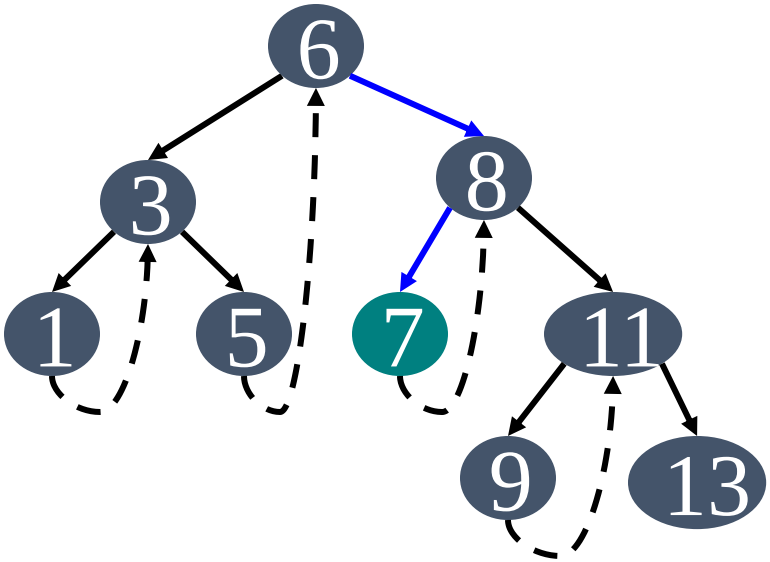
Threaded Tree Traversal



Output
1
3
5
6

Follow thread to right, print node

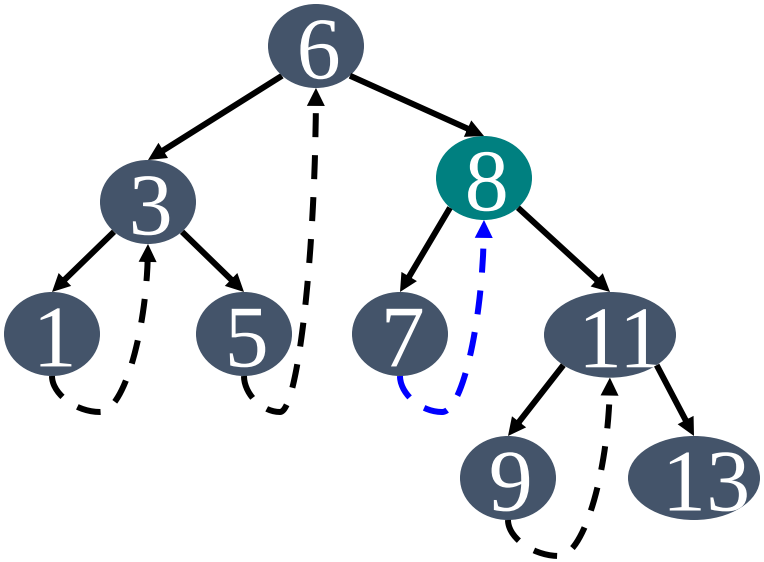
Threaded Tree Traversal



Output
1
3
5
6
7

Follow link to right, go to leftmost node and print

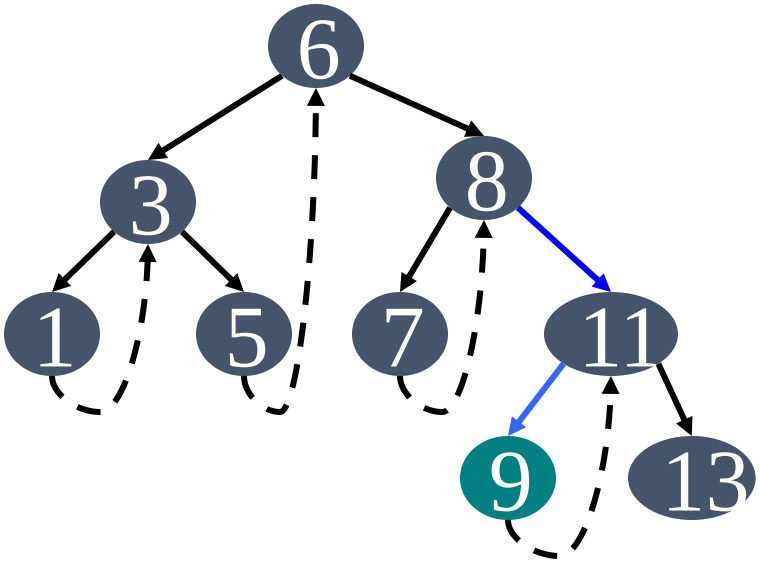
Threaded Tree Traversal



Output
1
3
5
6
7
8

Follow thread to right, print node

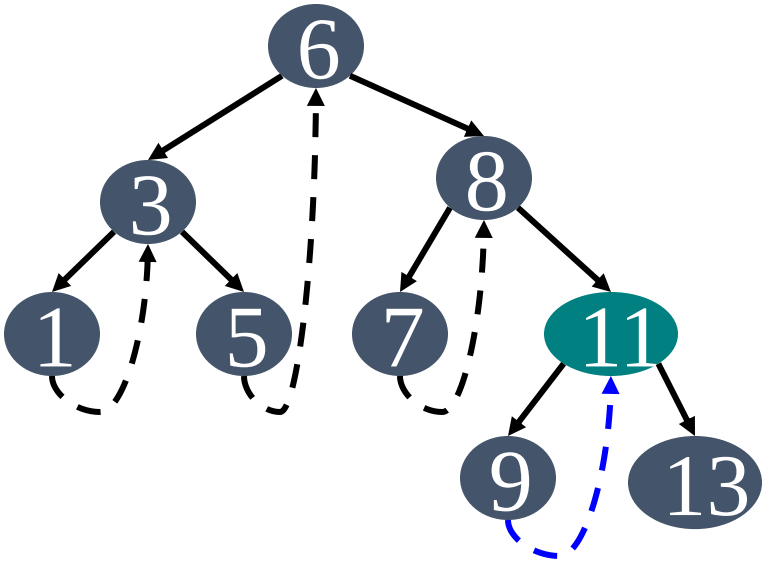
Threaded Tree Traversal



Output
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

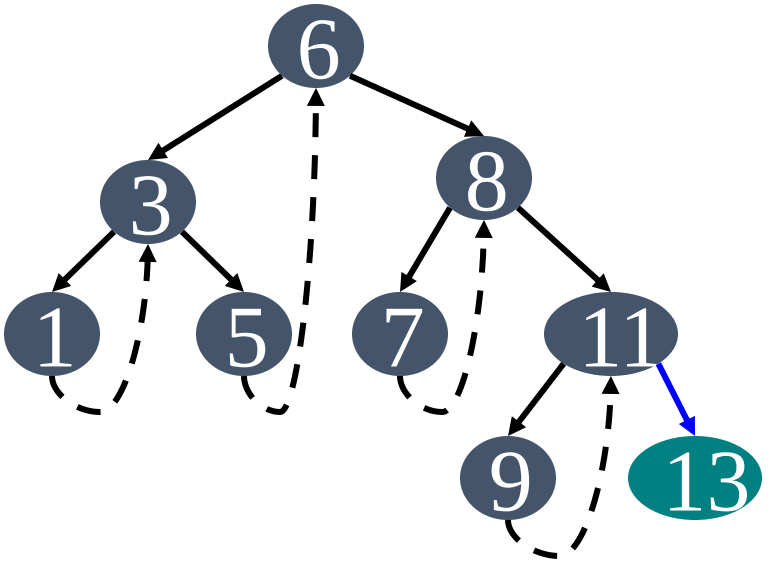
Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11

Follow thread to right, print node

Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11
13

Follow link to right, go to leftmost node and print

Threaded Tree Traversal Code

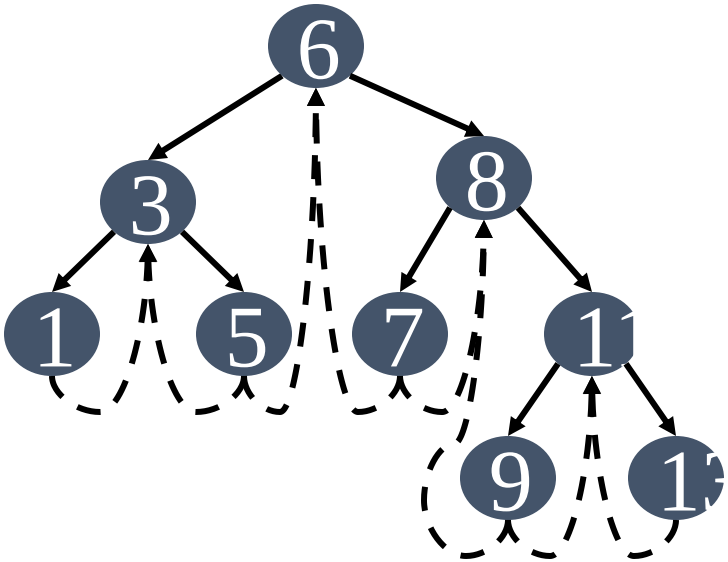
```
Node leftMost(Node n)
  {
   Node ans = n;
   if (ans == null) {
       return null;
   }
   while (ans.left != null)
  {
      ans = ans.left;
   }
   return ans;
}
```

```
void inOrder(Node n) {
   Node cur = leftmost(n);
   while (cur != null) {
      print(cur);
      if (cur.rightThread) {
         cur = cur.right;
      } else {
         cur = leftmost(cur.right);
      }
   }
}
```

Threaded Tree Modification

- We’re still wasting pointers, since half of our leafs’ pointers are still null
- We can add threads to the previous node in an inorder traversal as well, which we can use to traverse the tree backwards or even to do postorder traversals

Threaded Tree Modification



B Tree

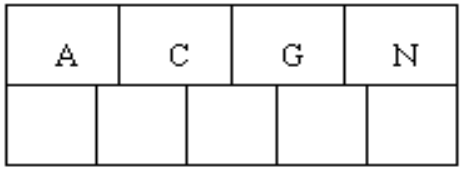
B-tree is a fairly well-balanced tree.

- All leaves are on the bottom level.
- All internal nodes (except perhaps the root node) have at least cell(m / 2) (nonempty) children.
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node (other than the root node if it is a leaf) must contain at least ceil(m / 2) - 1 keys.

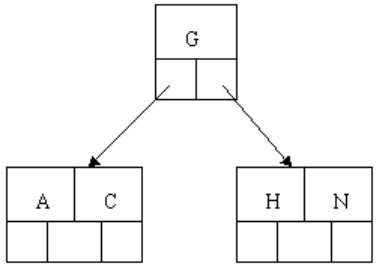
Let's work our way through an example similar to that given by Kruse. Insert the following letters into what is originally an empty B-tree of **order 5**:

C N G A H E K Q M F W L T Z D P R X Y S

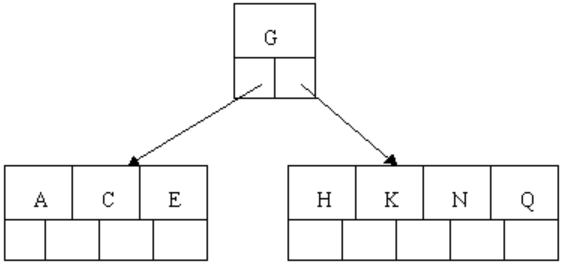
Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.

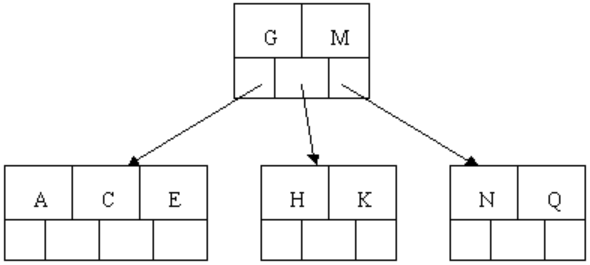


Inserting E, K, and Q proceeds without requiring any splits:

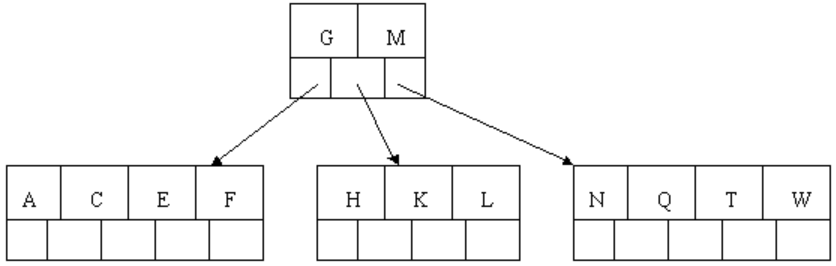


H E K Q M F W L T Z D P R X Y S

Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.

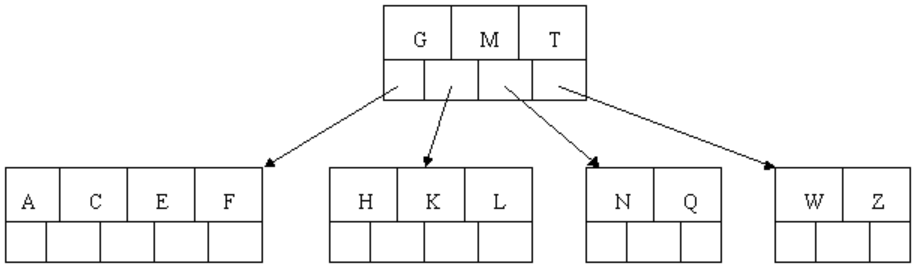


The letters F, W, L, and T are then added without needing any split.

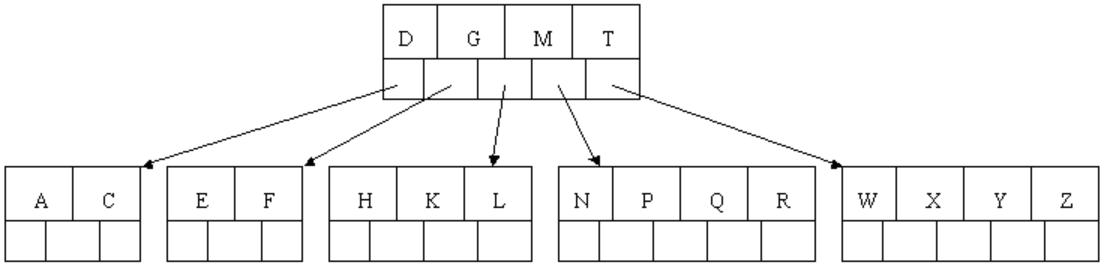


M F W L T Z D P R X Y S

When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.

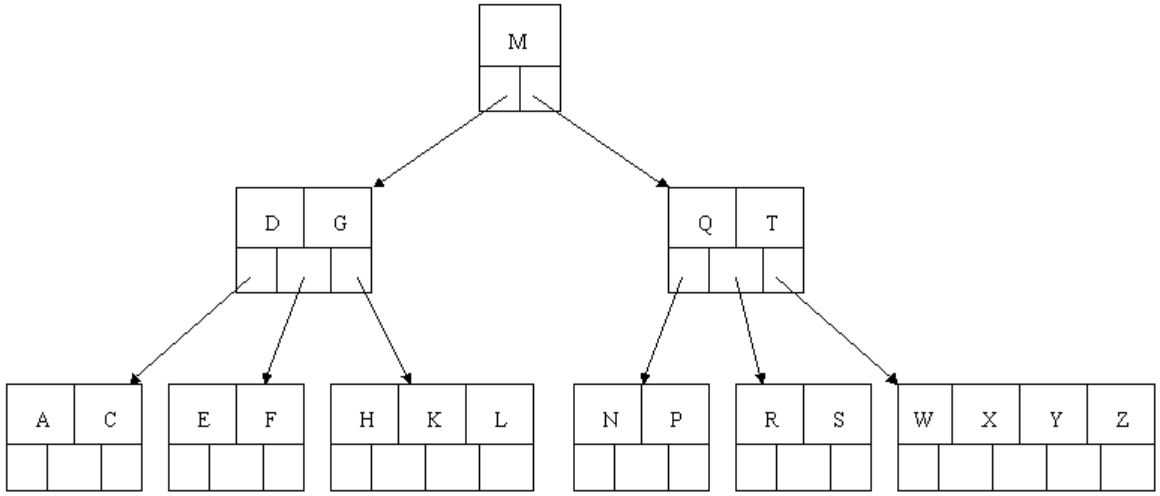


The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



Z D P R X Y S

Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



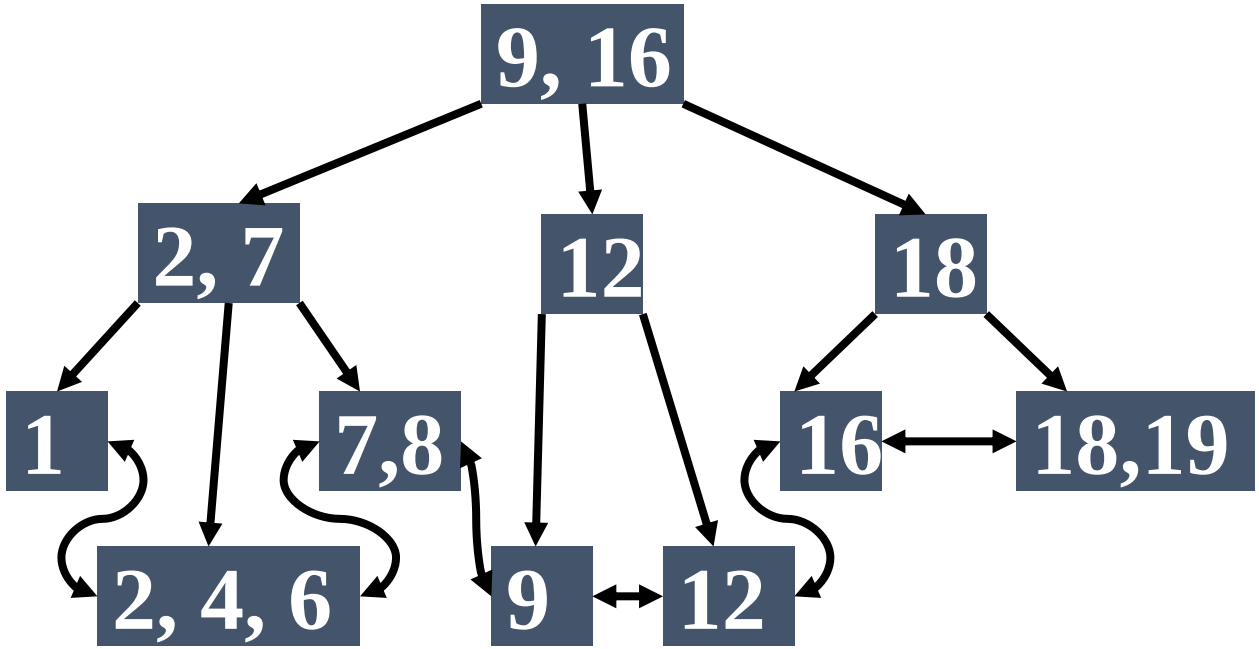
B+ Tree & B- Tree

- In the **B+ tree**, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access.
- a **B - tree** stores keys in its internal nodes but need not store those keys in the records at the leaves.
- **B - trees** can be turned into order statistic trees to allow rapid searches for the Nth record in key order, or counting the number of records between any two records, and various other related operations.

B+ Trees

- Similar to B trees, with a few slight differences
- All data is stored at the leaf nodes (*leaf pages*); all other nodes (*index pages*) only store keys
- Leaf pages are linked to each other
- Keys may be duplicated; every key to the right of a particular key is >= to that key

B+ Tree Example

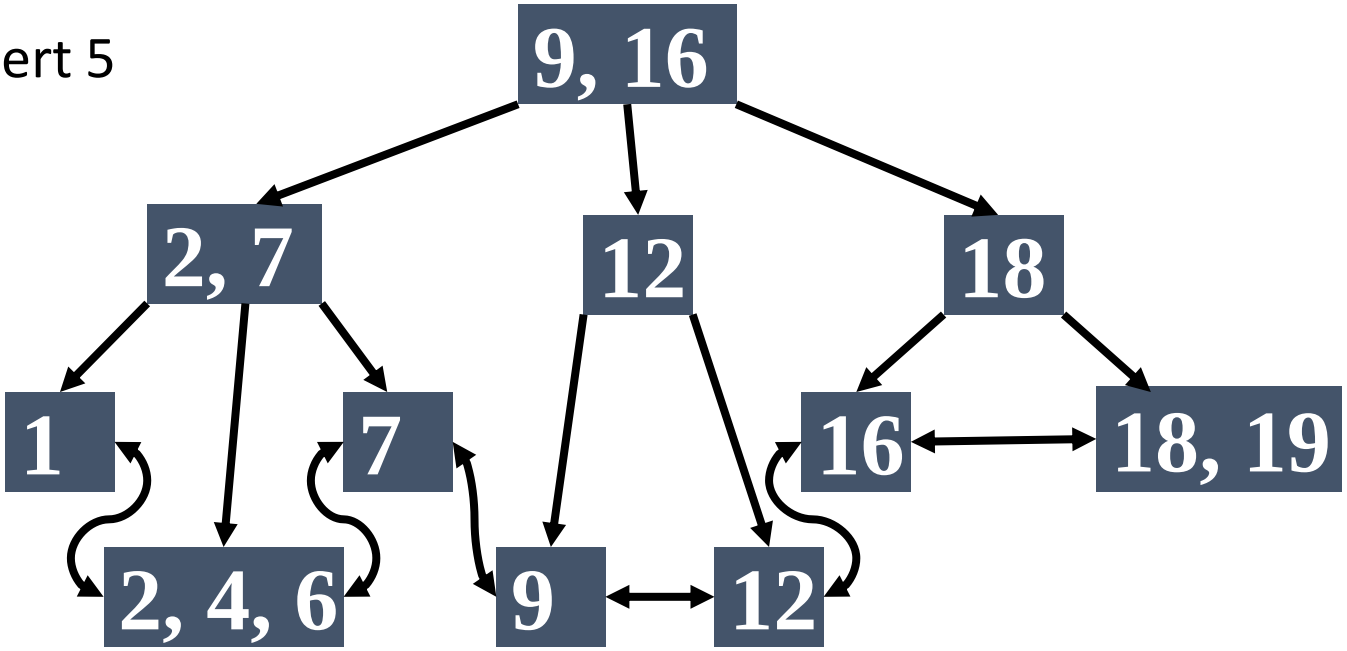


B+ Tree Insertion

- Insert at bottom level
- If leaf page overflows, split page and copy middle element to next index page
- If index page overflows, split page and move middle element to next index page

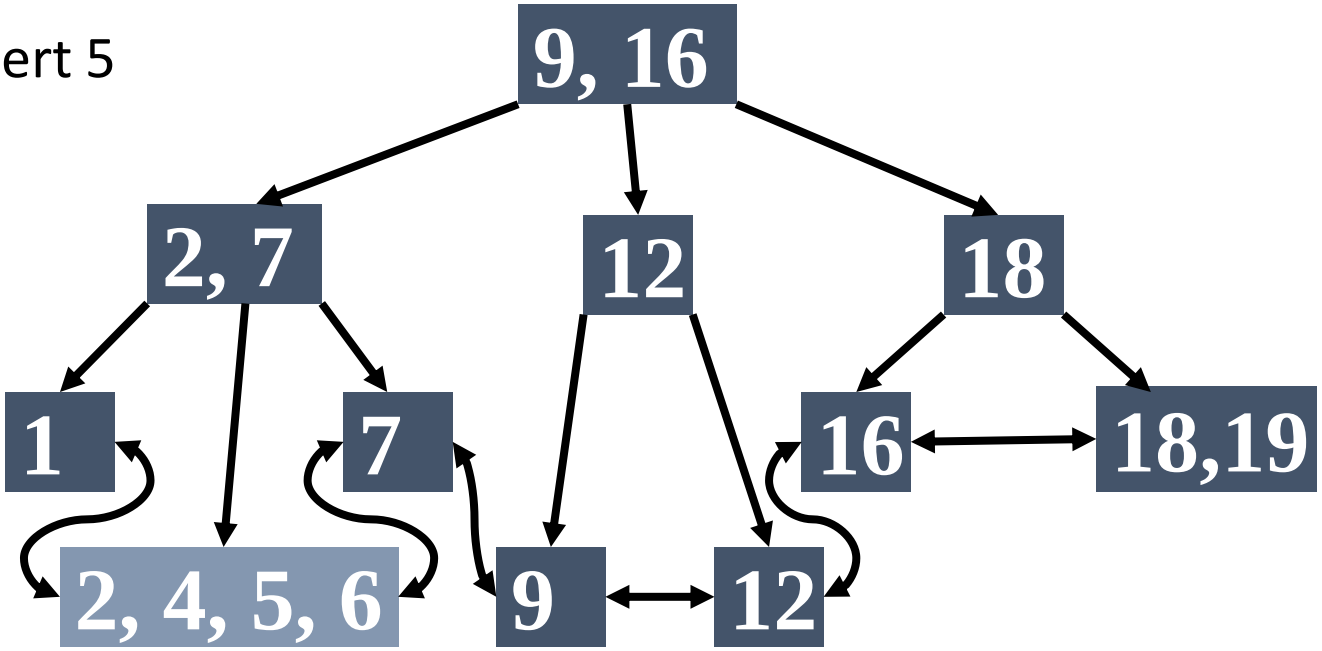
B+ Tree Insertion Example

Insert 5



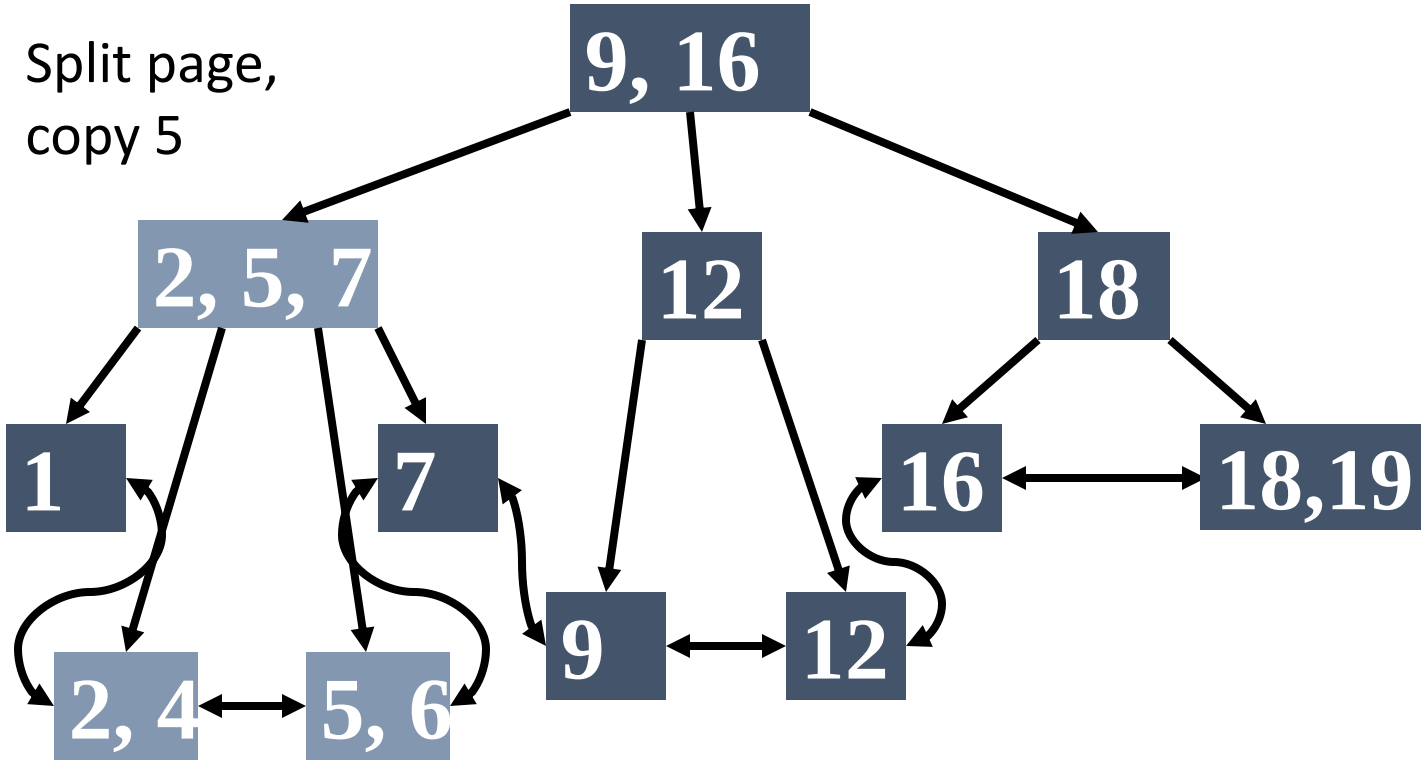
B+ Tree Insertion Example

Insert 5



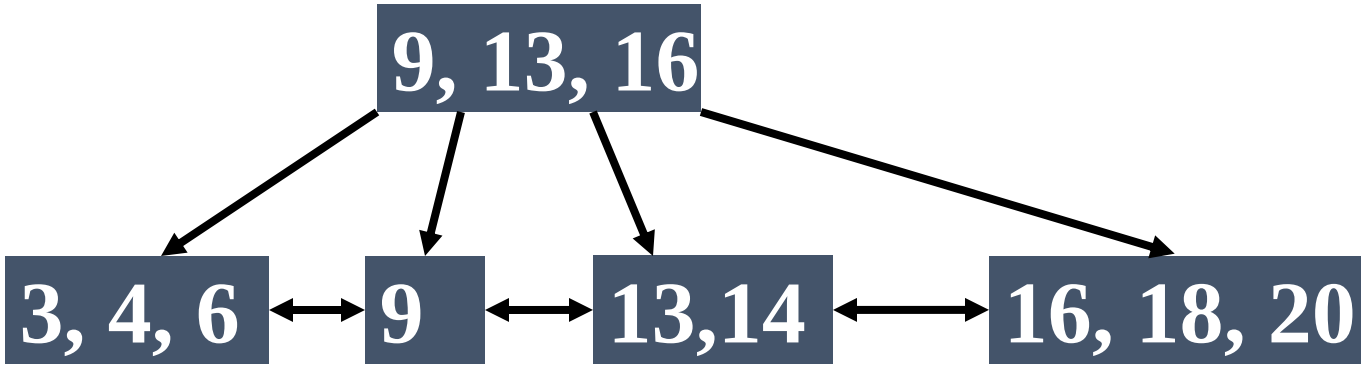
B+ Tree Insertion Example

Split page,
copy 5



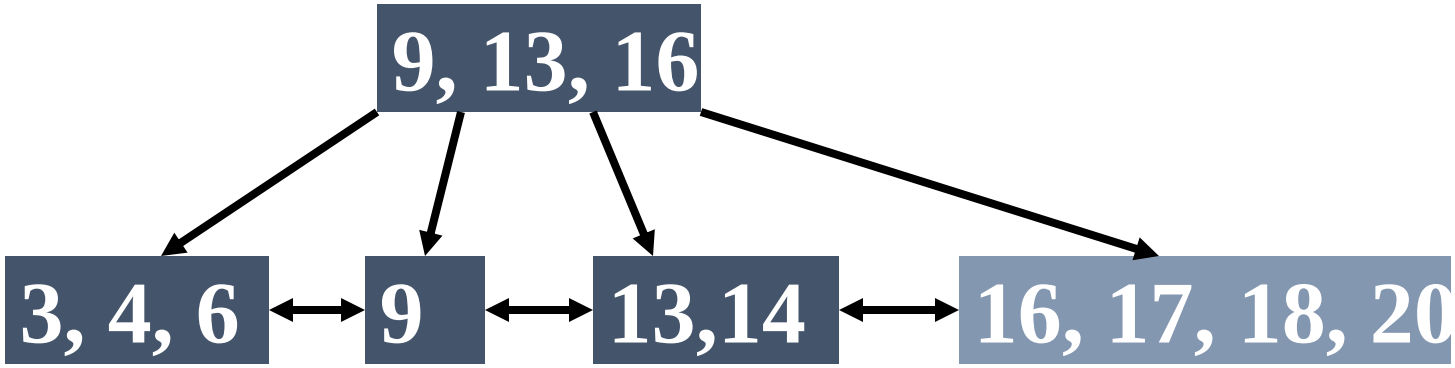
B+ Tree Insertion Example 2

Insert 17



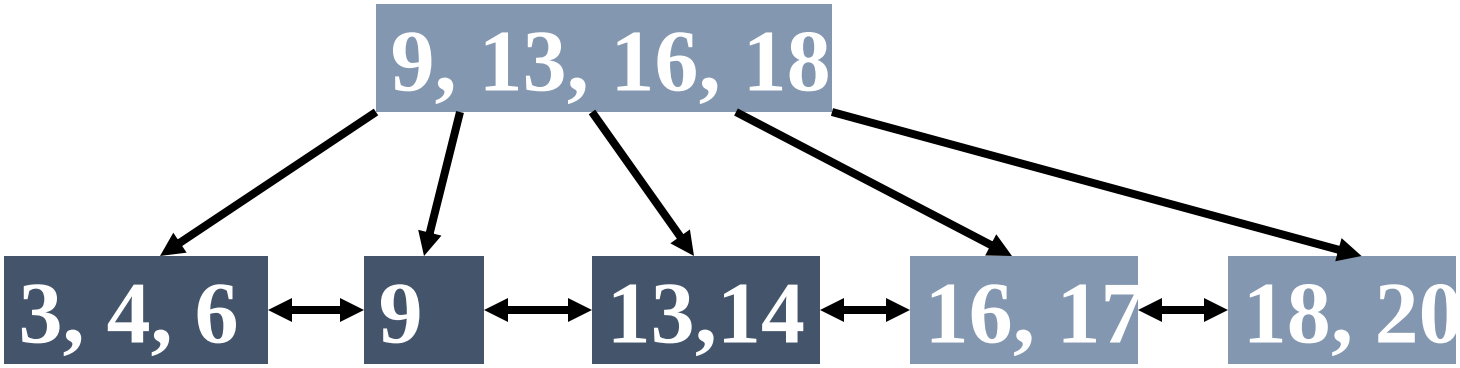
B+ Tree Insertion Example 2

Insert 17

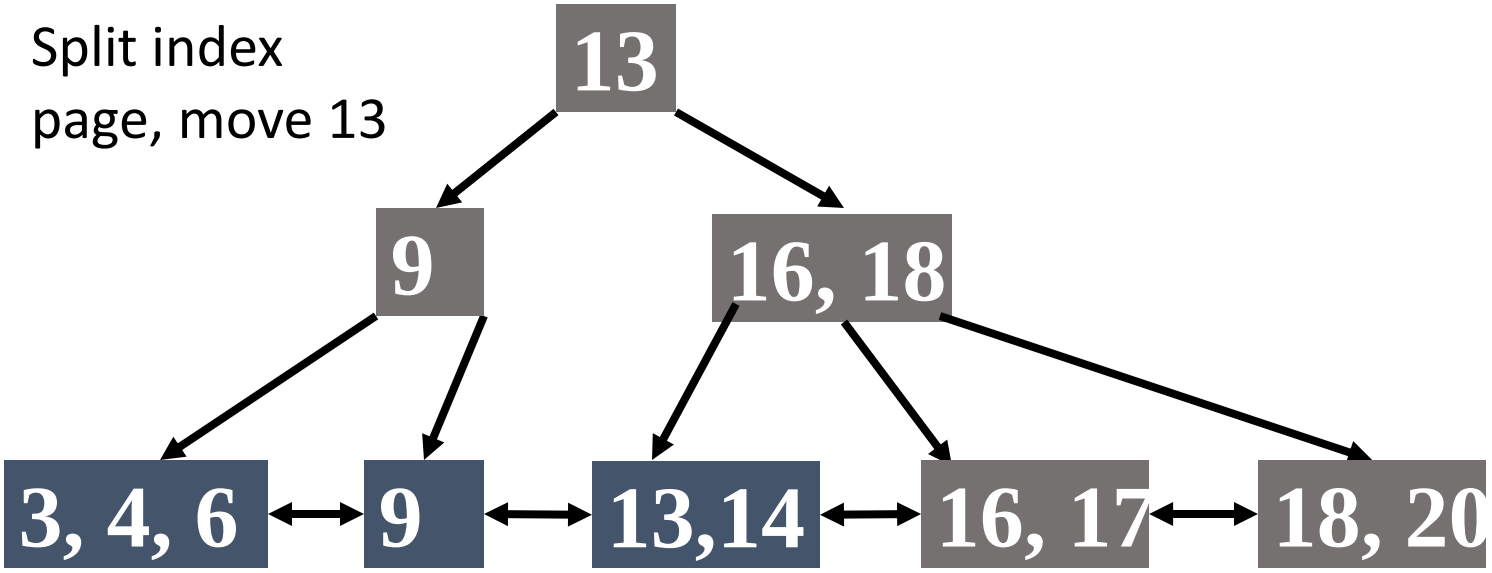


B+ Tree Insertion Example 2

Split leaf page,
copy 18



B+ Tree Insertion Example 2

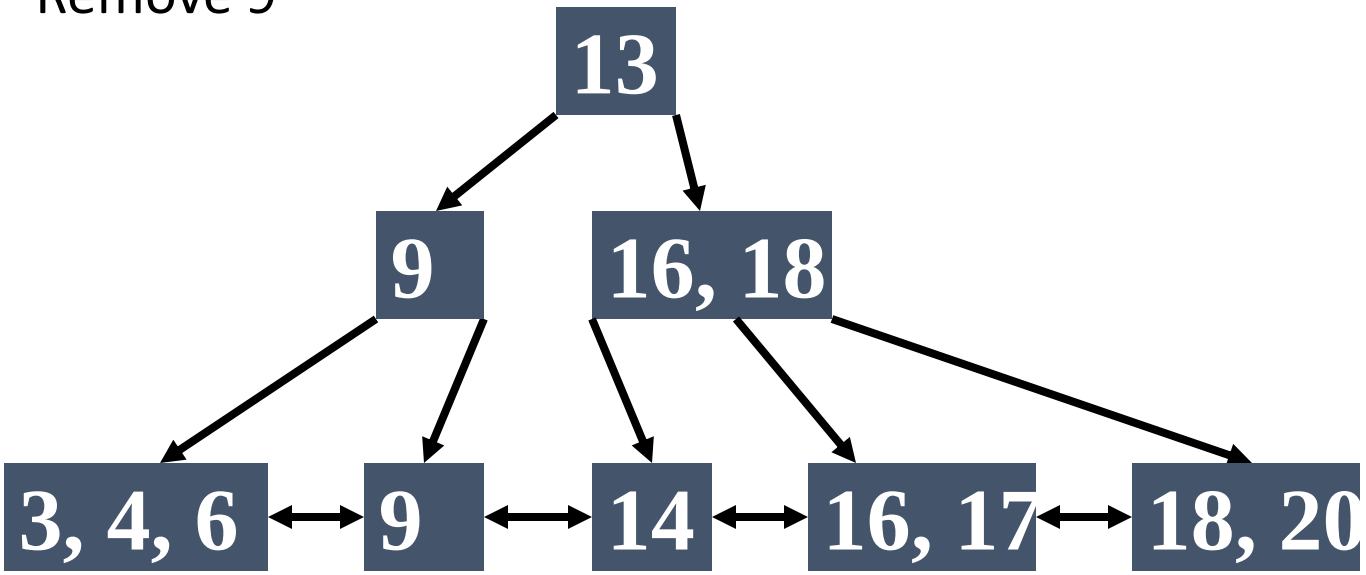


B+ Tree Deletion

- Delete key and data from leaf page
- If leaf page underflows, merge with sibling and delete key in between them
- If index page underflows, merge with sibling and move down key in between them

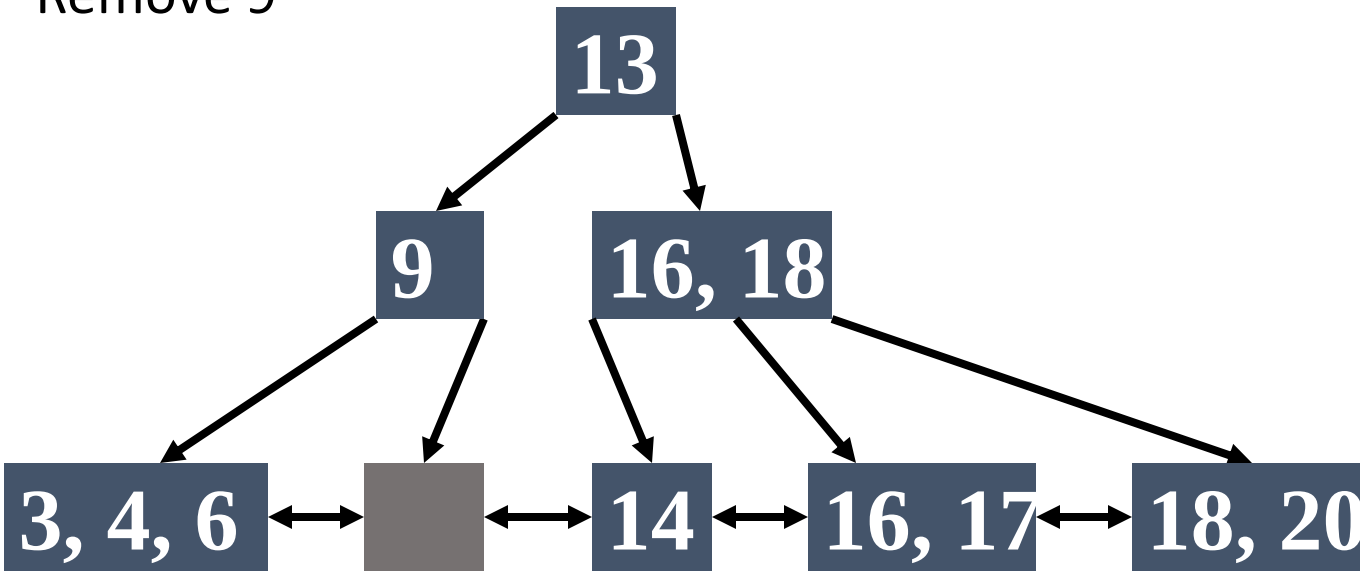
B+ Tree Deletion Example

Remove 9

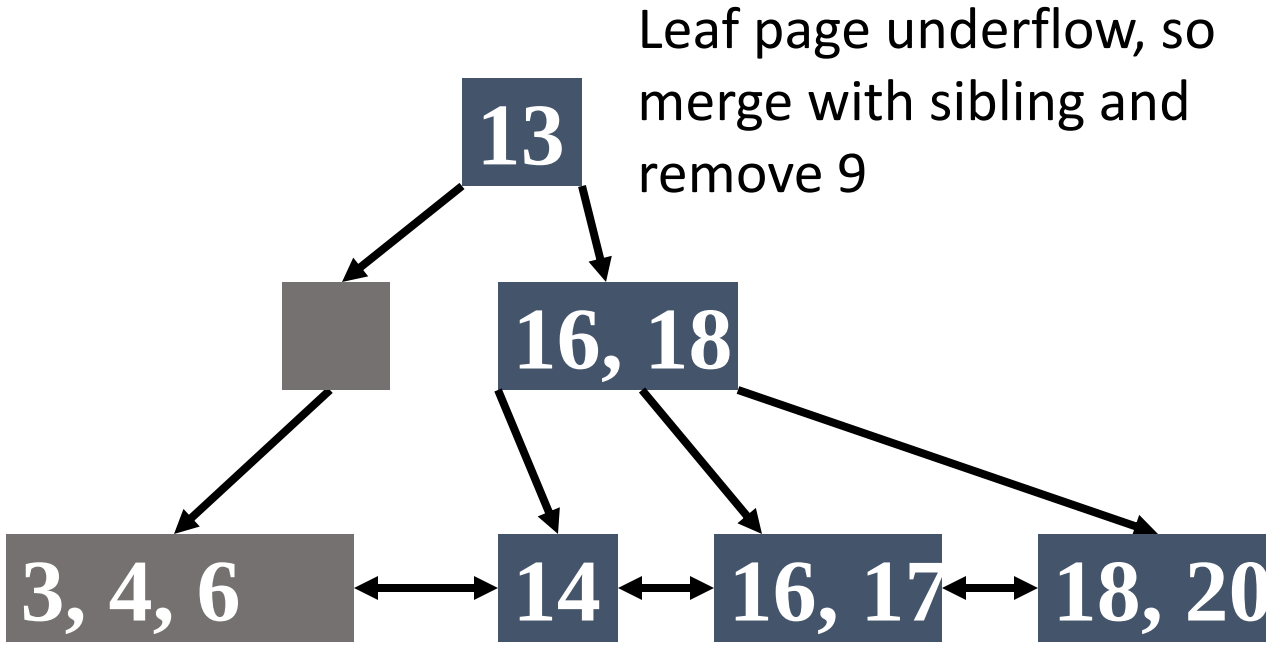


B+ Tree Deletion Example

Remove 9

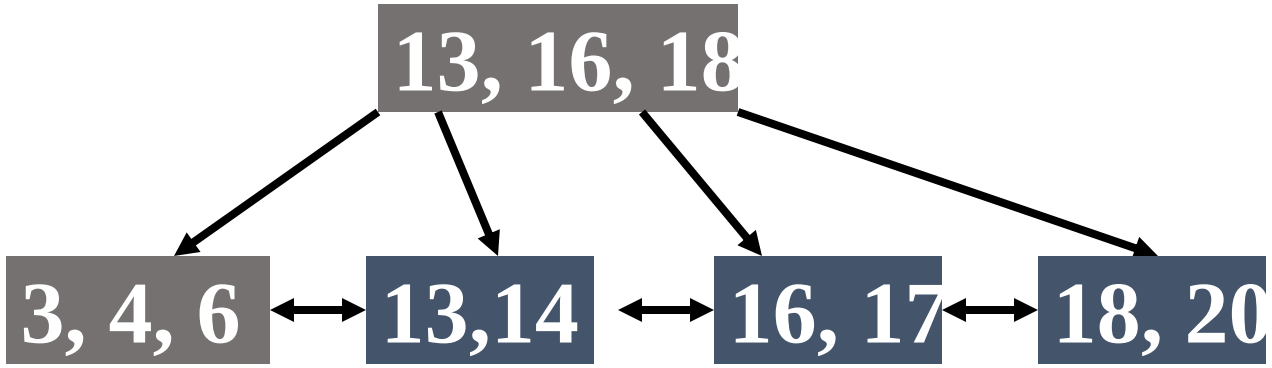


B+ Tree Deletion Example



B+ Tree Deletion Example

Index page underflow,
so merge with sibling
and demote 13



B* Tree by Donald Knuth

I. Every node except the root has at most m children.
II. Every node, except for the root and the leave , has at least (2m-1)/3 children.
III. The root has at least 2 and at most 2[(2m-2)/3]+1 children.
IV. All leaves appear on the same level.
V. A nonleaf node with k children contains k-1 keys

- The important change is condition II, which asserts that we utilize at least two third of the available space in every node.

B* Tree

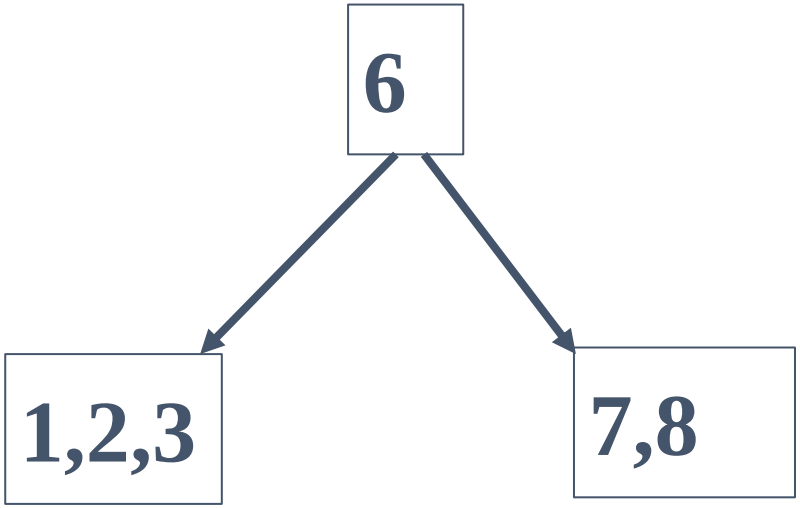
- The **B*** **tree** balances more neighbouring internal nodes to keep the internal nodes more densely packed.
  - This variant requires non-root nodes to be at least 2/3 full instead of 1/2.
  - To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a node next to it.
  - When both nodes are full, then the two nodes are split into three. Deleting nodes is somewhat more complex than inserting however.

B* Tree

- The complication in the implementation arises, since instead of combining two nodes into one on deletes, and splitting one node into two on inserts, you have to combine three nodes into two, and split two nodes into three.
- The Star-ness of a B-Tree can be extended to any fraction, forcing nodes to be 3/4 full, or 9/10 full of 99/100 full.

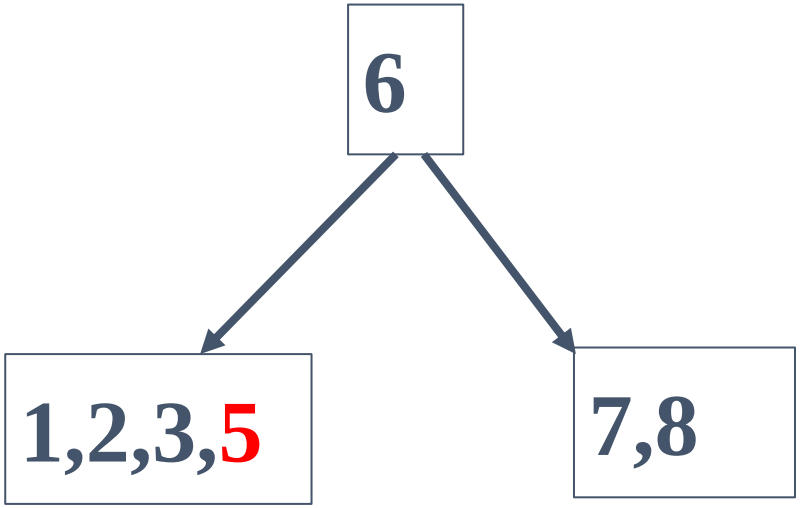
Order m= 5
Keys = 4
Minimum fill (2m-1)/3 = 3

ADD 5



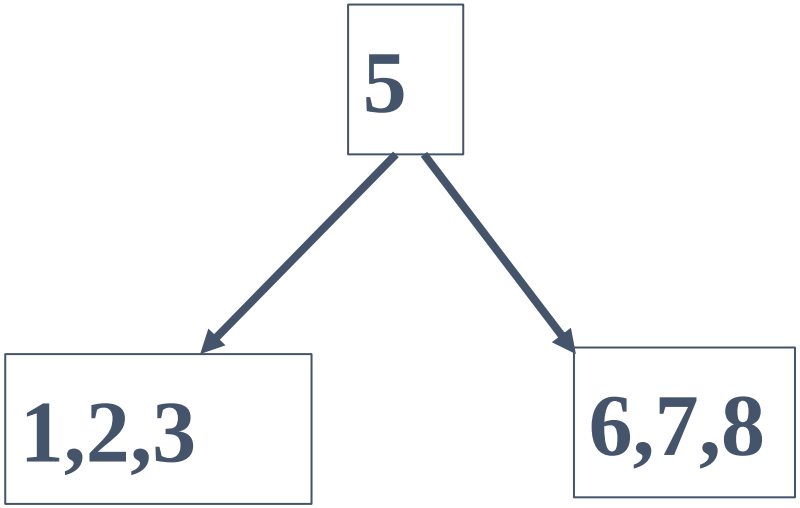
Order m= 5
Keys = 4
Minimum fill (2m-1)/3 = 3

ADD 5



Order m= 5
Keys = 4
Minimum fill (2m-1)/3 = 3

ADD 5



Recursive Traversals

- Inorder
  - Traverse the left subtree
  - Visit the node
  - Traverse the right subtree
- Preorder
  - Visit the node
  - Traverse the left subtree
  - Traverse the right subtree

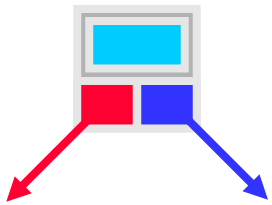
Recursive Traversals

- Postorder
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the node

Non recursive Traversal of Binary Tree

The Scenario

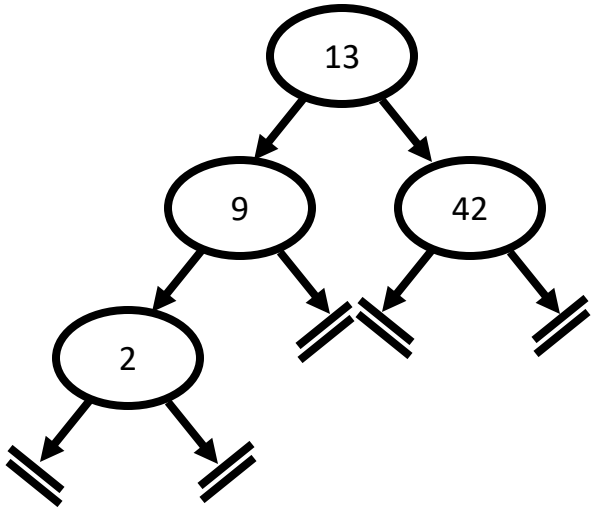
- **Imagine we have a binary tree**
- **We want to traverse the tree**
  - **It's not linear**
  - **We need a way to visit all nodes**



- **Three things must happen:**
  - **Deal with the entire left sub-tree**
  - **Deal with the current node**
  - **Deal with the entire right sub-tree**

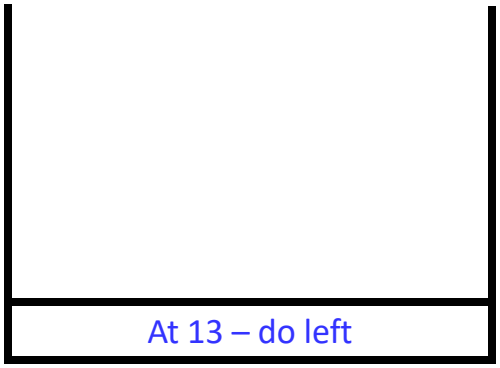
Use the Activation Stack

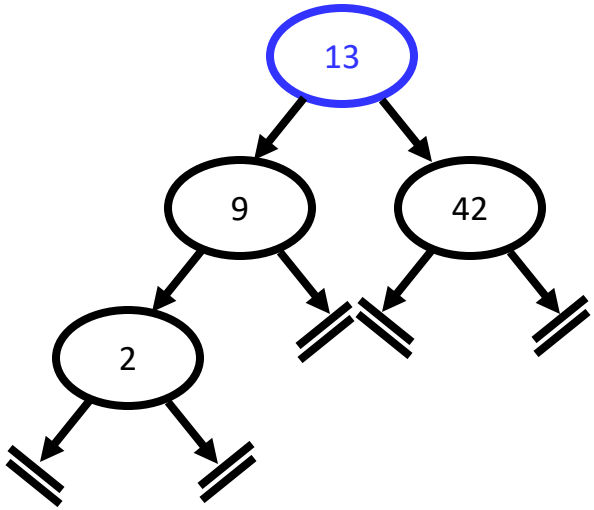
- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**



Use the Activation Stack

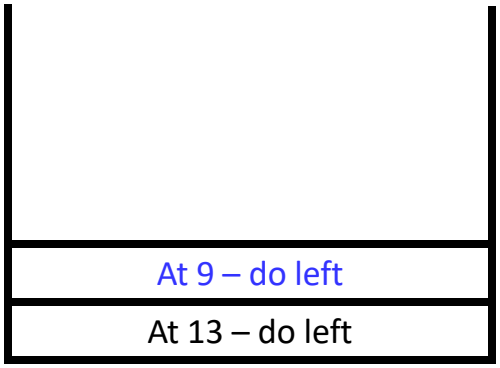
- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

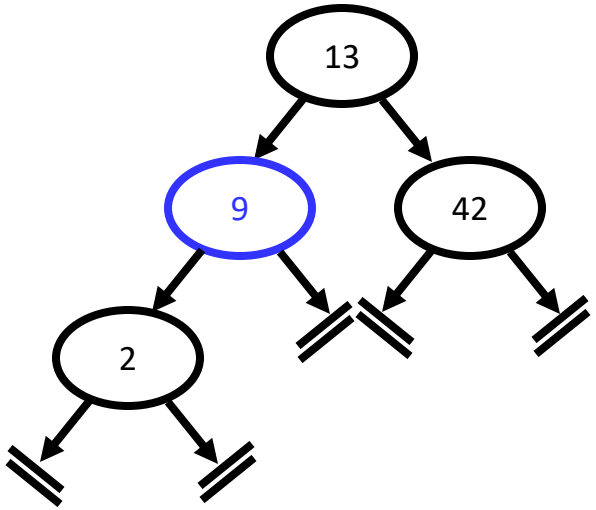




Use the Activation Stack

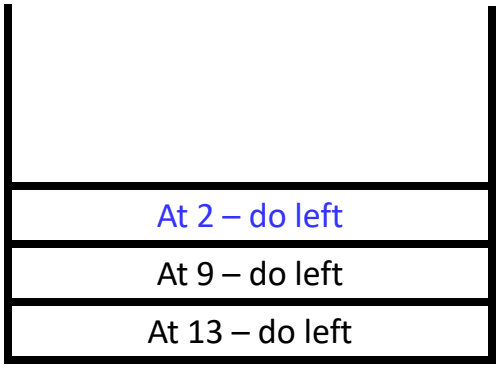
- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

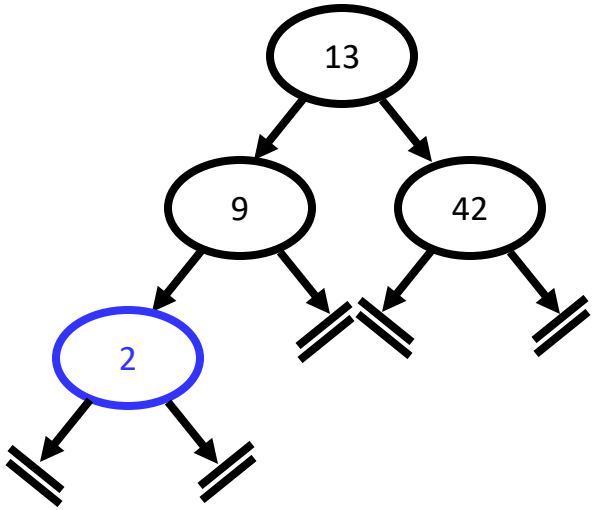




Use the Activation Stack

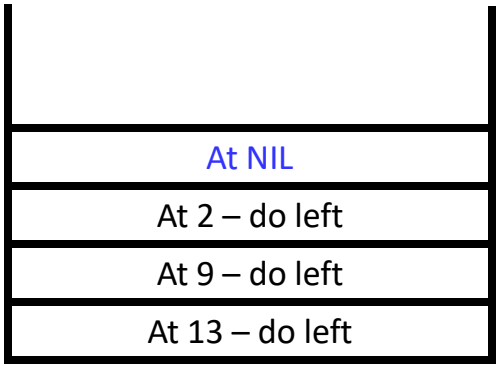
- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

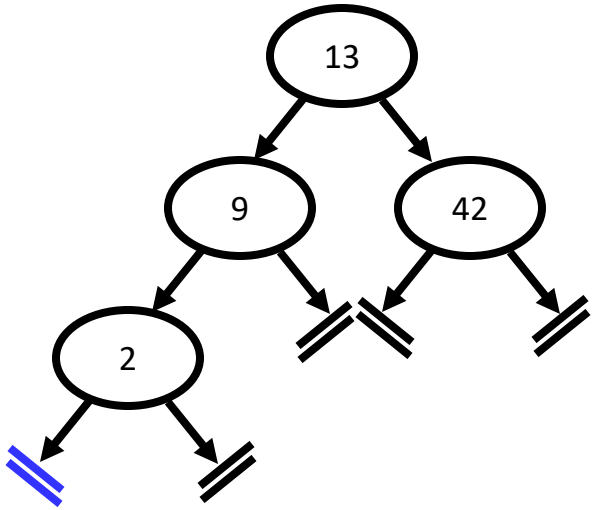




Use the Activation Stack

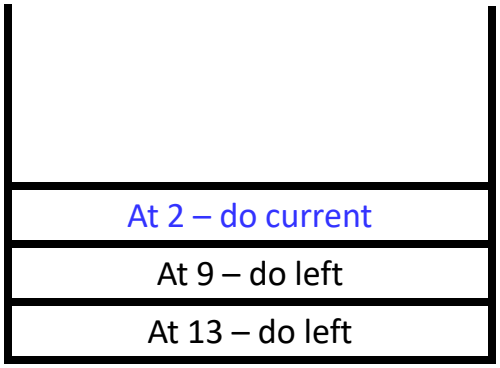
- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

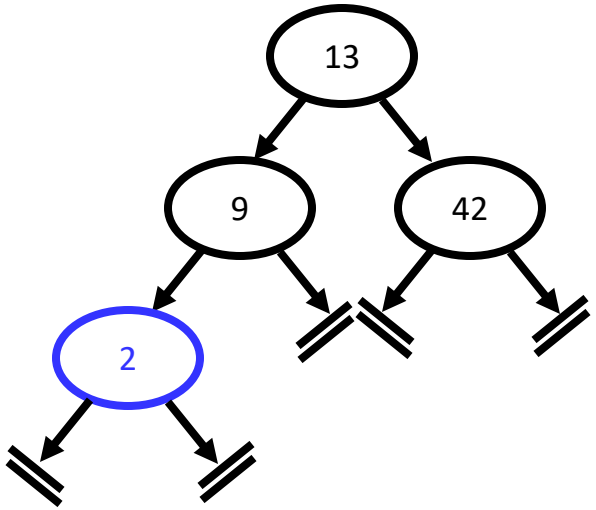




Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

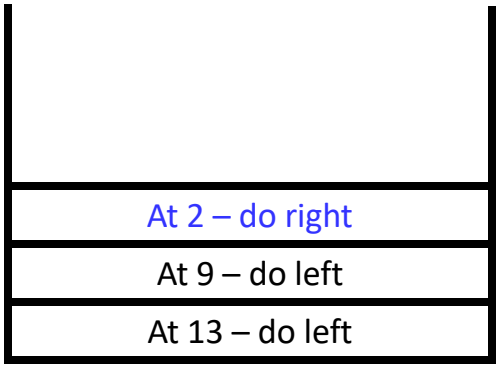


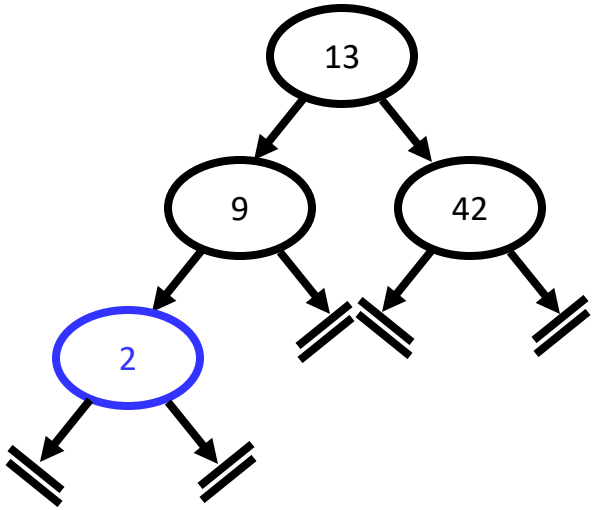


2,

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

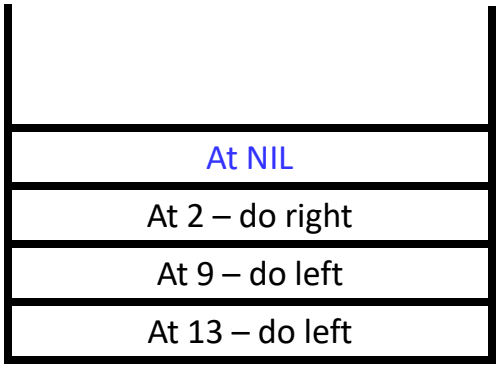


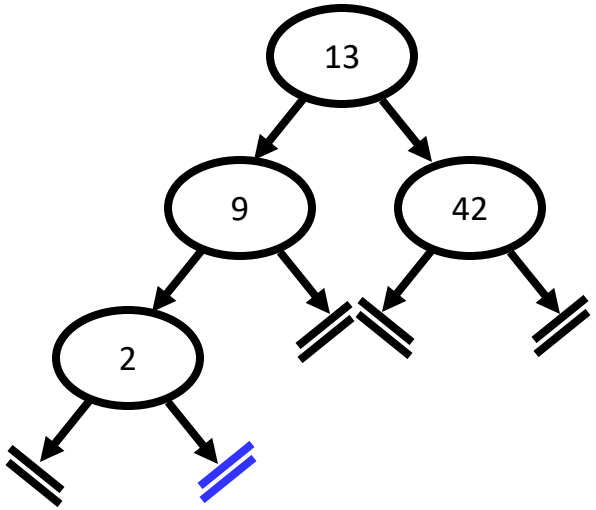


2,

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

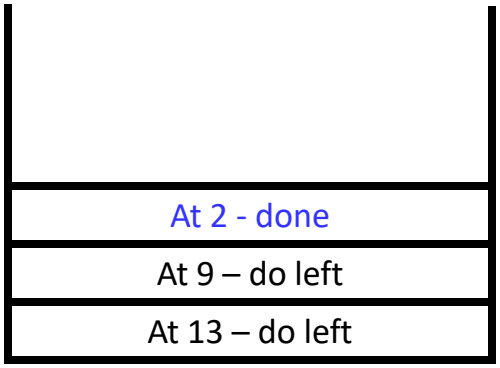


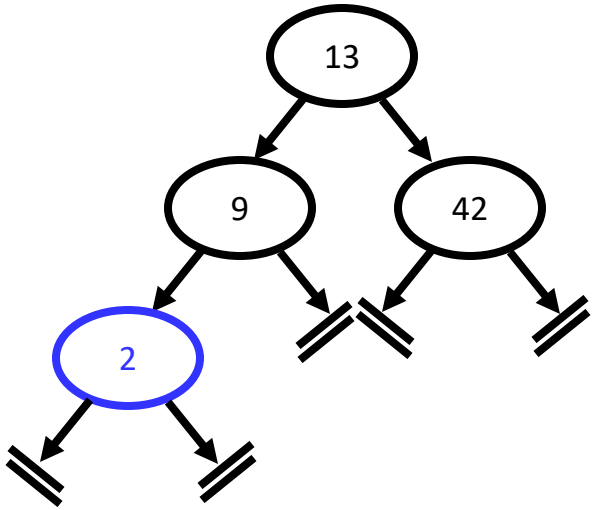


2,

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

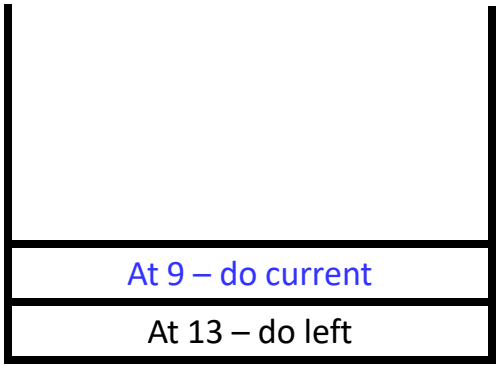


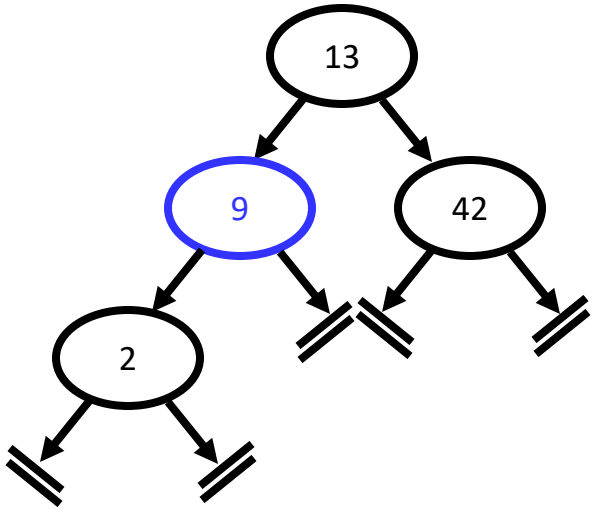


2,

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

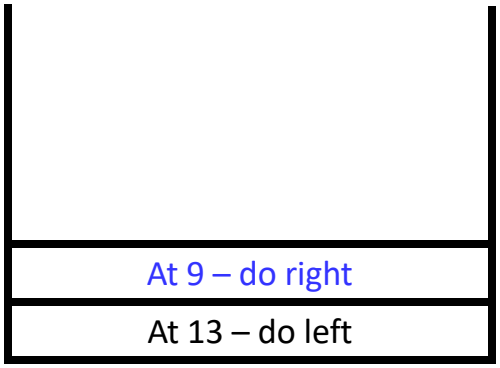


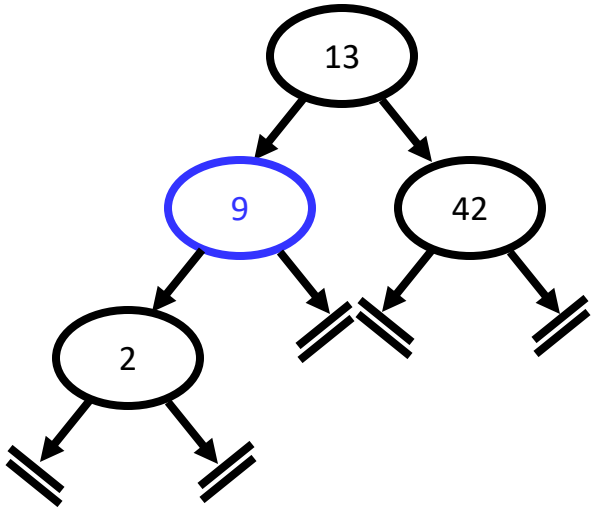


2,9

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

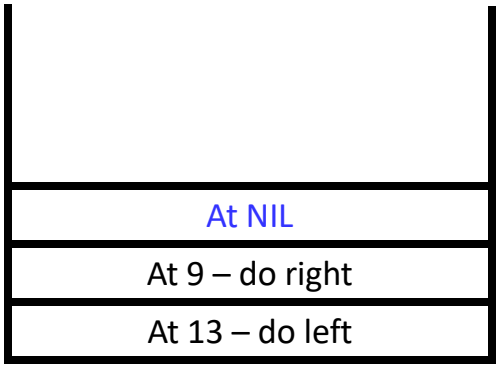


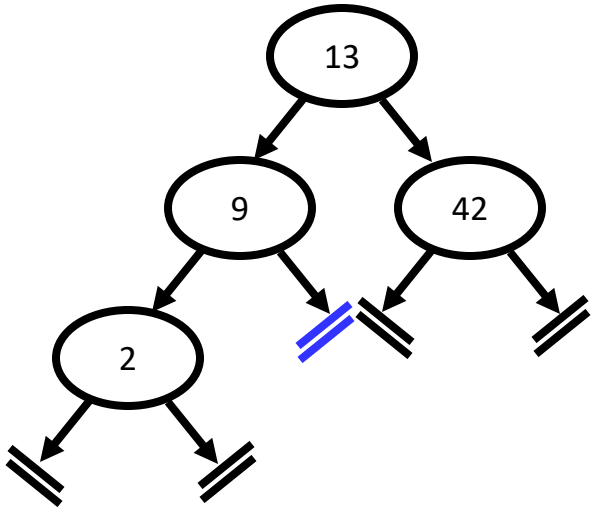


2,9

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

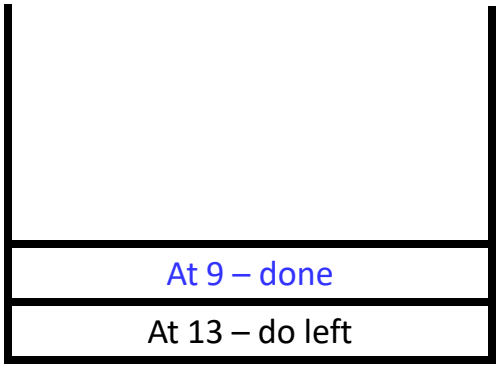


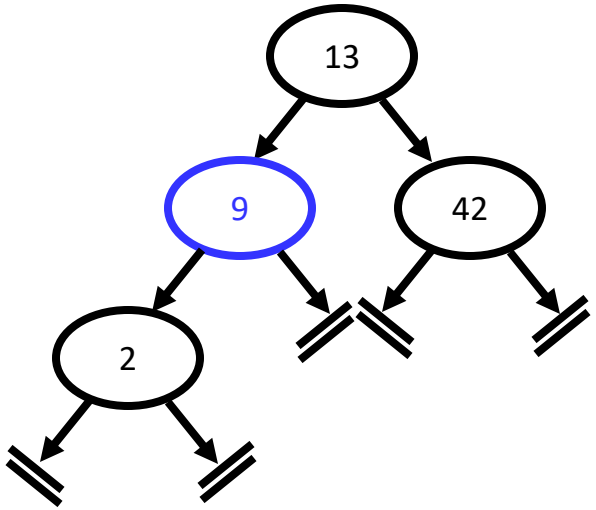


2,9

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

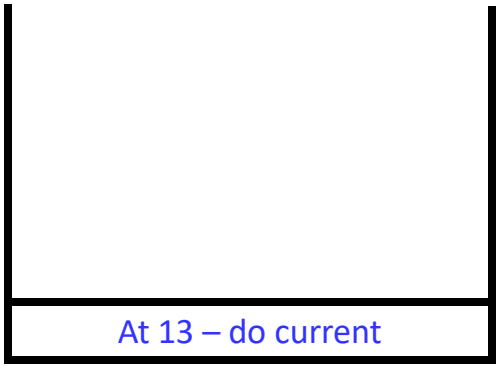


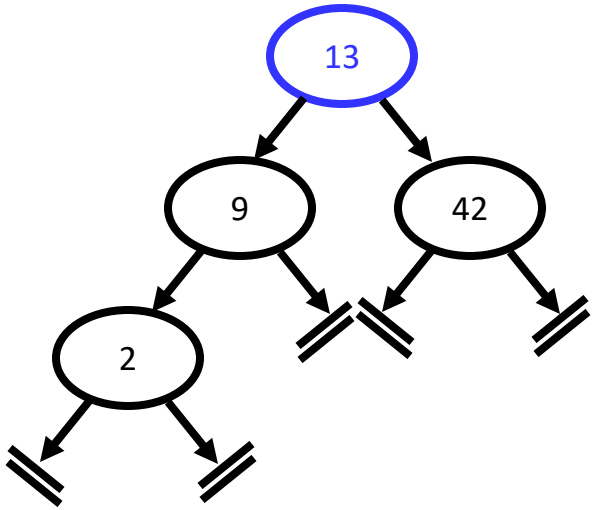


2,9

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

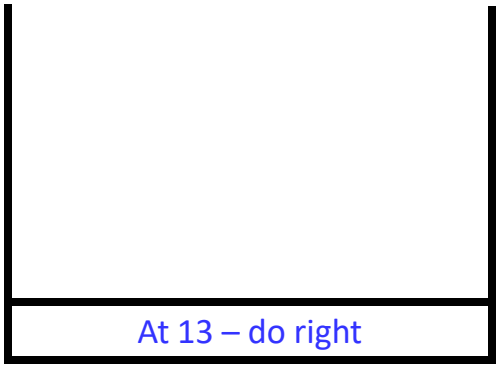


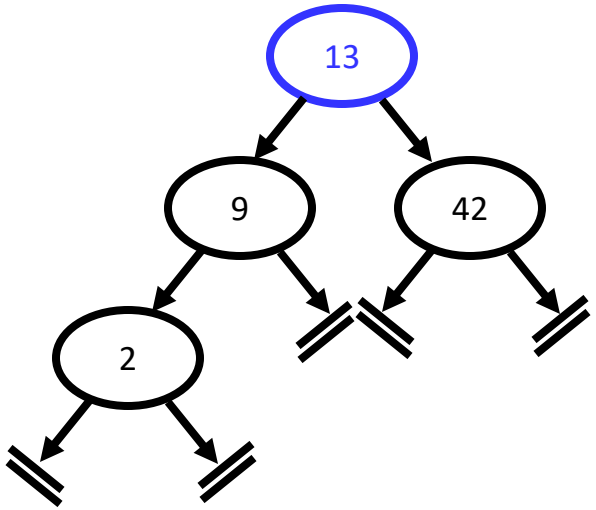


2,9,13

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

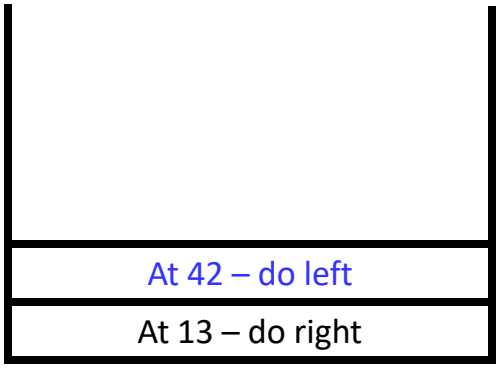


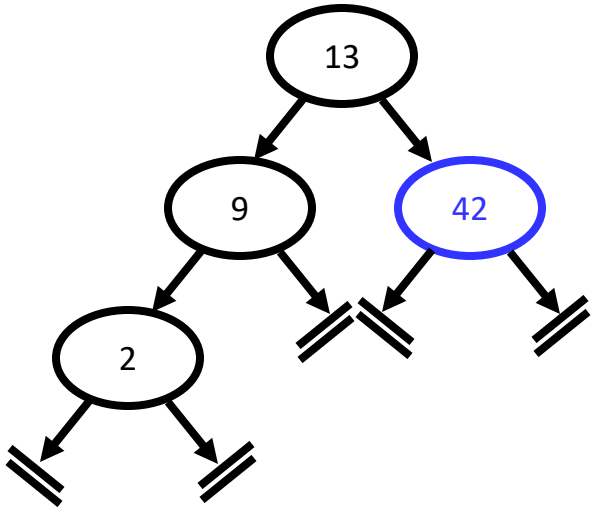


2,9,13

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

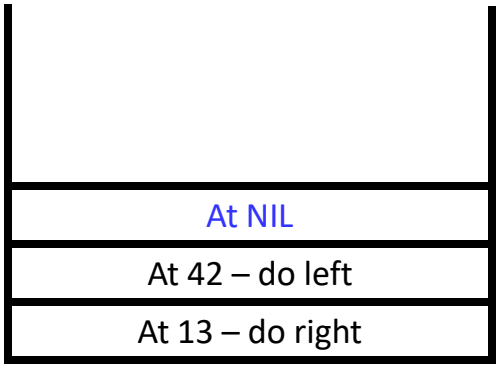


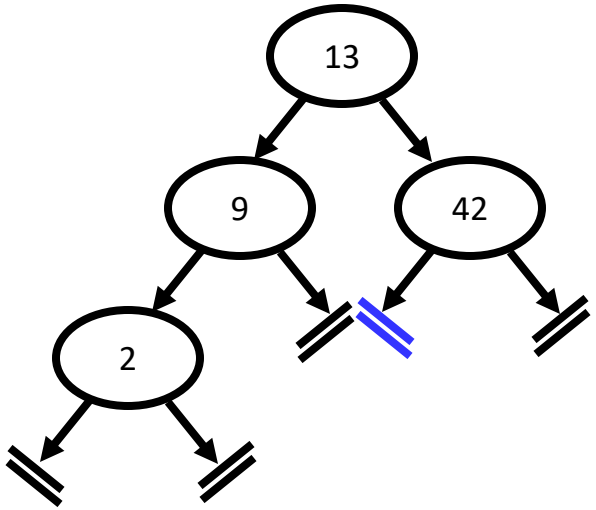


2,9,13

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

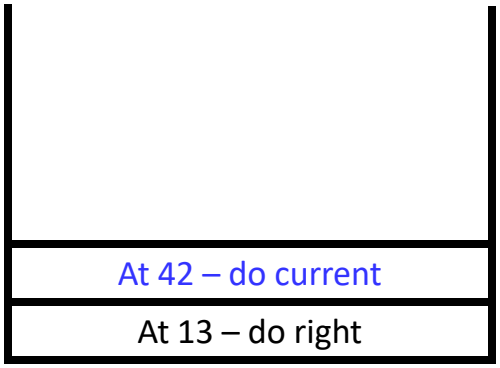


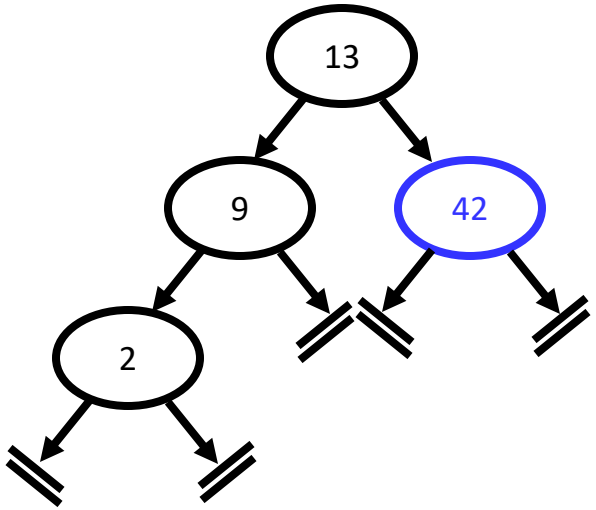


2,9,13

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

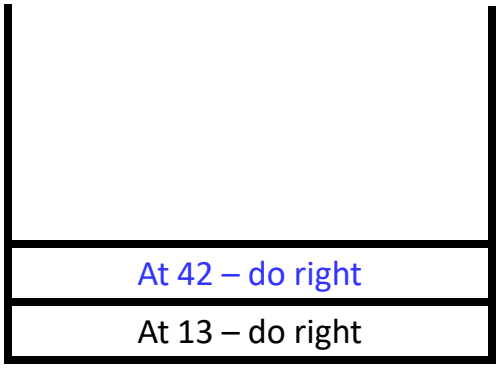


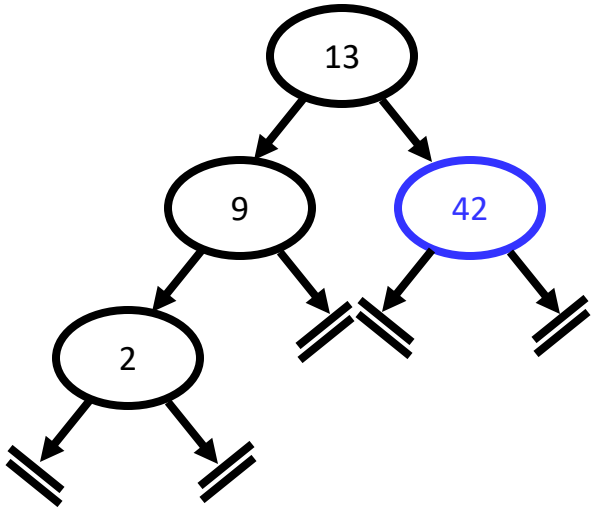


2,9,13,42

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

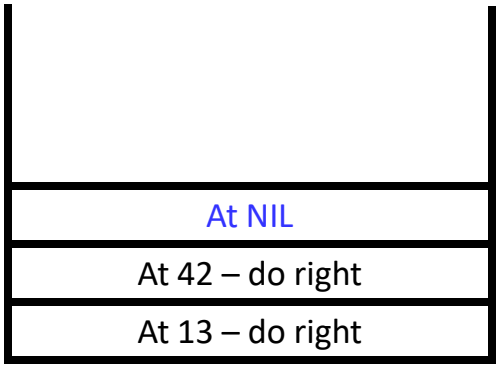


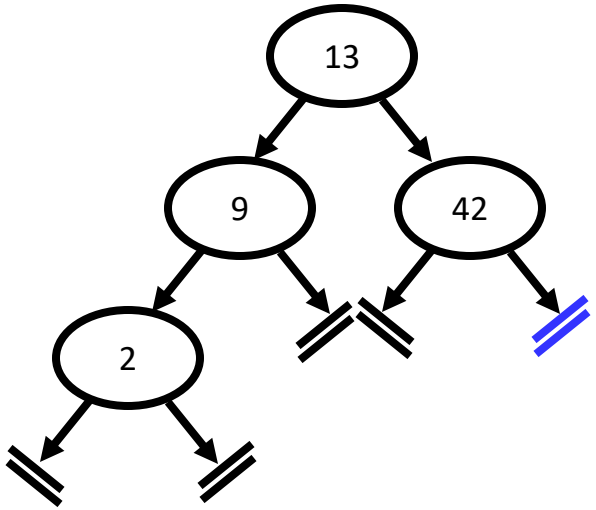


2,9,13,42

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

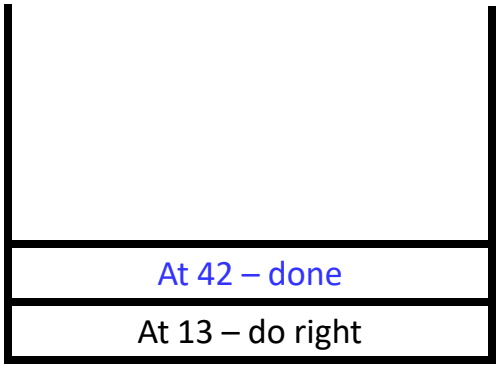


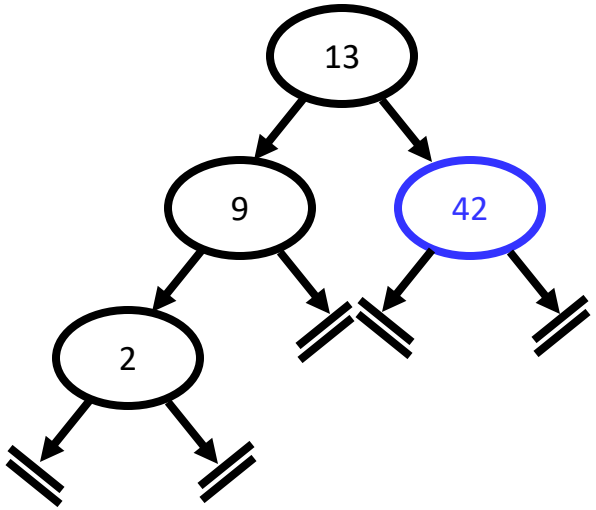


2,9,13,42

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**

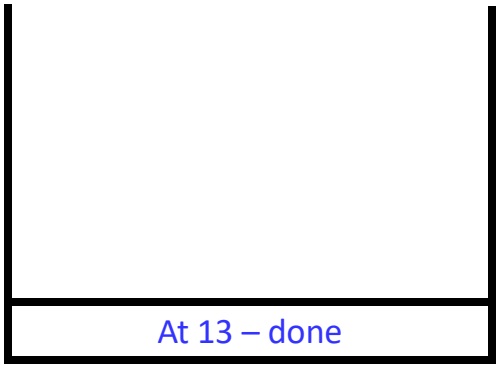


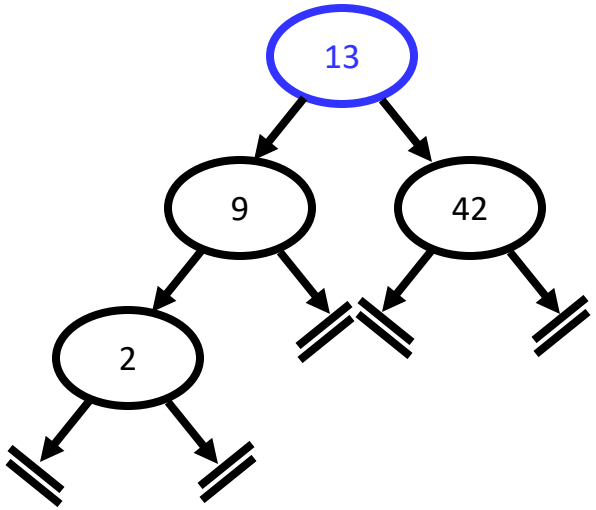


2,9,13,42

Use the Activation Stack

- **With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.**





2,9,13,42

Outline of In-Order Traversal

- **Three principle steps:**
  - **Traverse Left**
  - **Do work (Current)**
  - **Traverse Right**

- **Work can be anything**
- **Separate work from traversal**

Nonrecursive Inorder Traversal

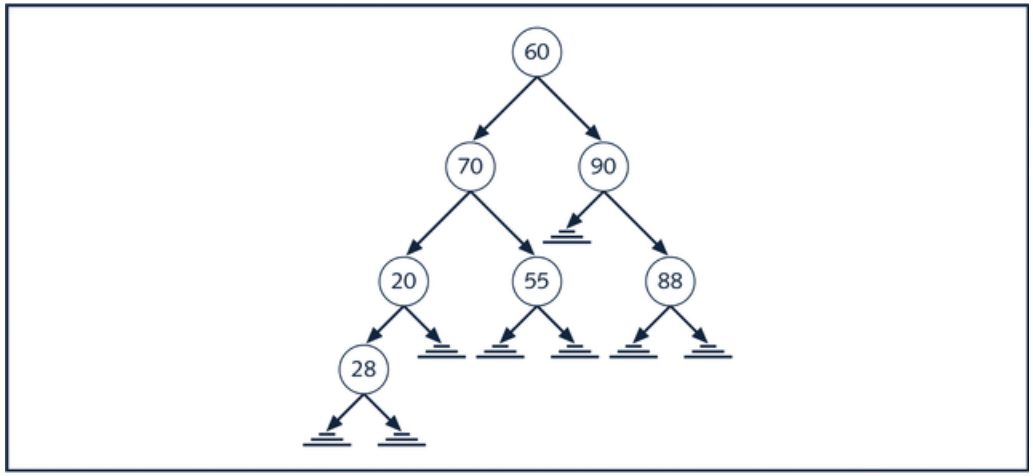


Figure 11-17 Binary tree; the leftmost node is 28

Nonrecursive Inorder Traversal: General Algorithm

```
current = root;  //start traversing the binary tree
at
                          // the root node
while(current is not NULL or stack is nonempty)
       if(current is not NULL)
       {
          push current onto stack;
          current = current->llink;
       }
       else
       {
          pop stack into current;
          visit current;    //visit the node
          current = current->rlink;     //move to the
                                        //right child
       }
```

Nonrecursive Preorder Traversal: General Algorithm

```
current = root;  //start the traversal at the root
node
while(current is not NULL or stack is nonempty)
    if(current is not NULL)
    {
       visit current;
       push current onto stack;
       current = current->llink;
    }
    else
    {
       pop stack into current;
       current = current->rlink;    //prepare to
visit
                                    //the right
subtree
    }
```

Nonrecursive Postorder Traversal

```
current = root;  //start traversal at root node
v = 0;
if(current is NULL)
          the binary tree is empty
if(current is not NULL)
push current into stack;
push 1 onto stack;
current = current->llink;
while(stack is not empty)
     if(current is not NULL and v is 0)
     {
               push current and 1 onto stack;
               current = current->llink;
              }
```

Nonrecursive Postorder Traversal (Continued)

```
    else
{
          pop stack into current and v;
          if(v == 1)
          {
                push current and 2 onto stack;
                current = current->rlink;
                v = 0;
    }
    else
                visit current;
 }
```

Tree Categorization based on balance factor

- Height Balanced Trees
- Weight Balanced Trees

Height-Balanced Trees

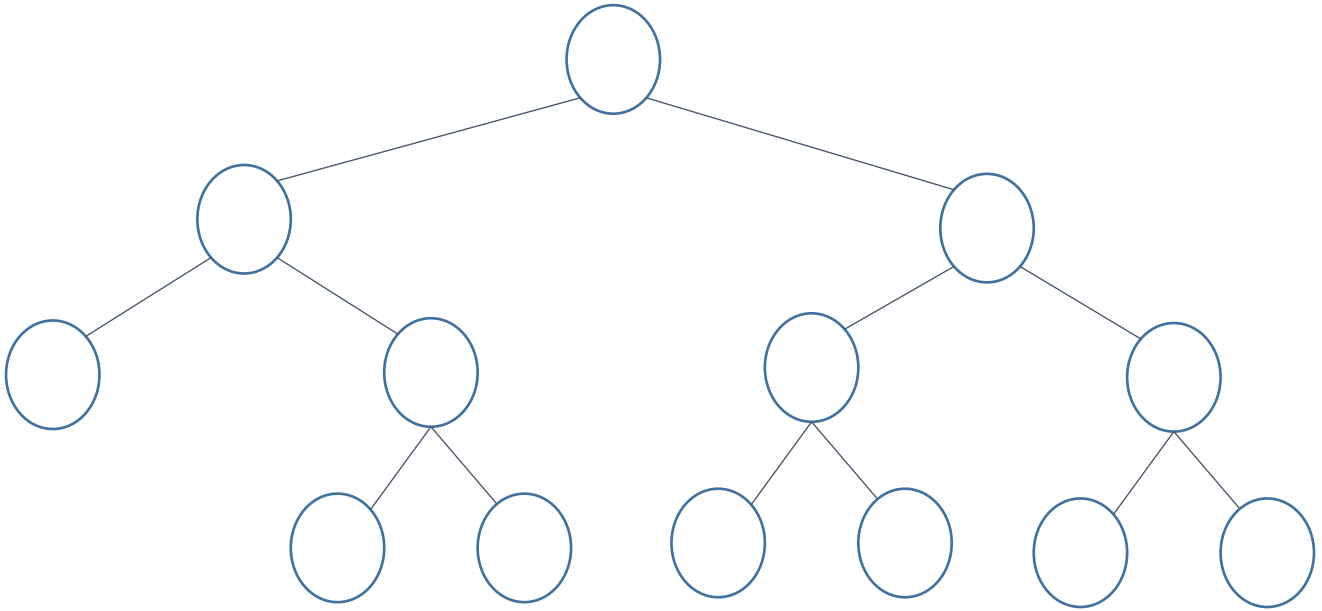
Height – The maximum length of any path from the root to a leaf.

Height-Balanced Tree –

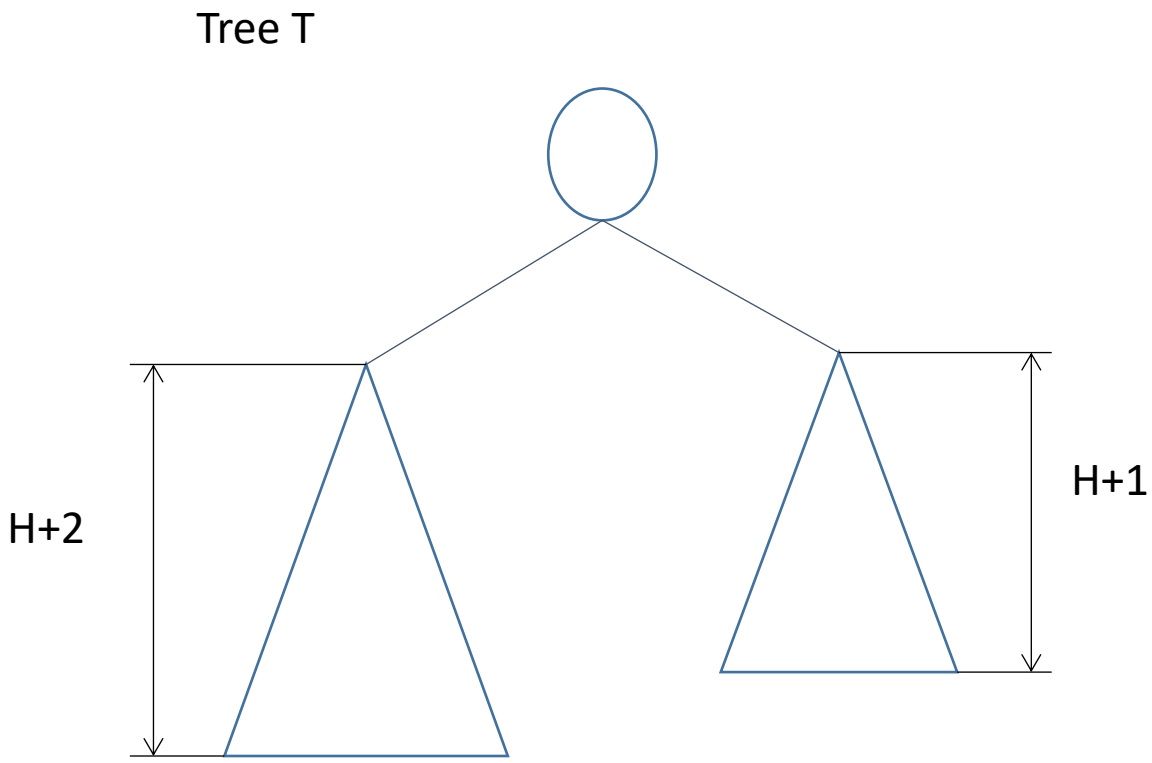
In each interior node, the height of the right subtree and left subtree differ by at most one.

Height-Balanced Trees

Example:



Height-Balanced Trees



HEIGHT Balanced Trees

Abs(depth(*leftChild*) – depth(*rightChild*)) <= 1
Depth of a tree is it's longest path length

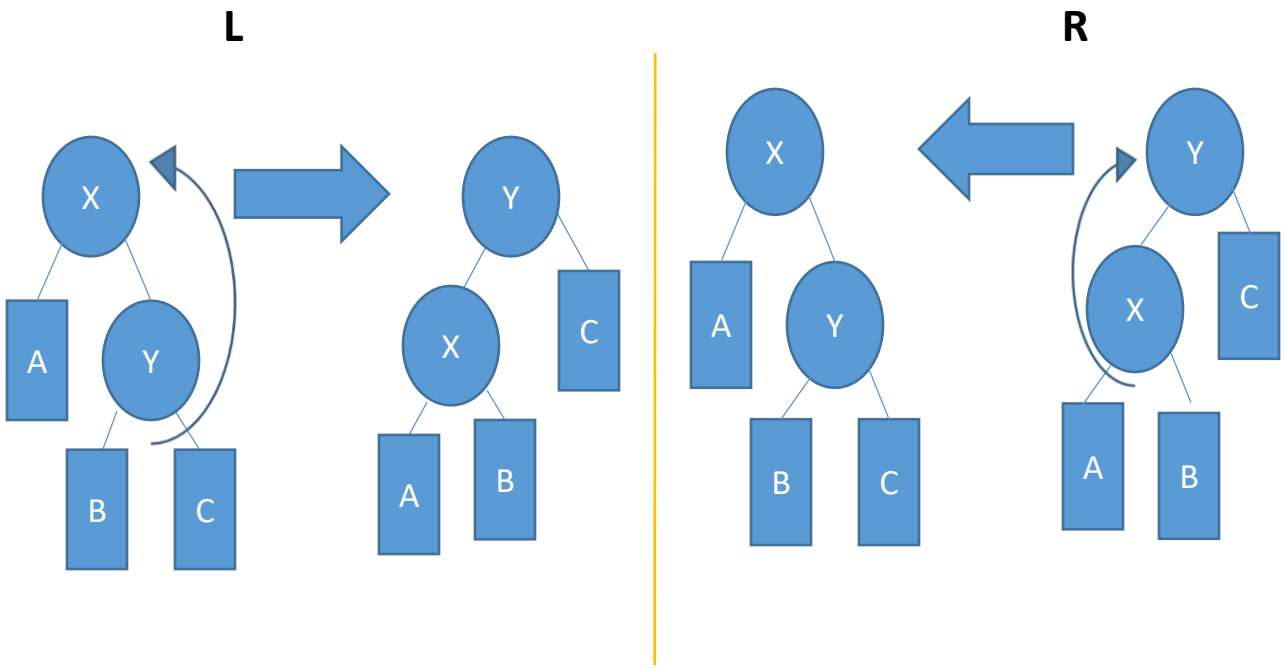
- Red-black trees – Restructure the tree when rules among nodes of the tree are violated as we follow the path from root to the insertion point.
- AVL Trees – Maintain a three way flag at each node (-1,0,1) determining whether the left sub-tree is longer, shorter or the same length. Restructure the tree when the flag would go to –2 or +2.
- Splay Trees – Don't require complete balance. However, N inserts and deletes can be done in NlgN time. Rotations are done to move accessed nodes to the top of the tree.

Rotations

Analyze possible tree depths after rotation

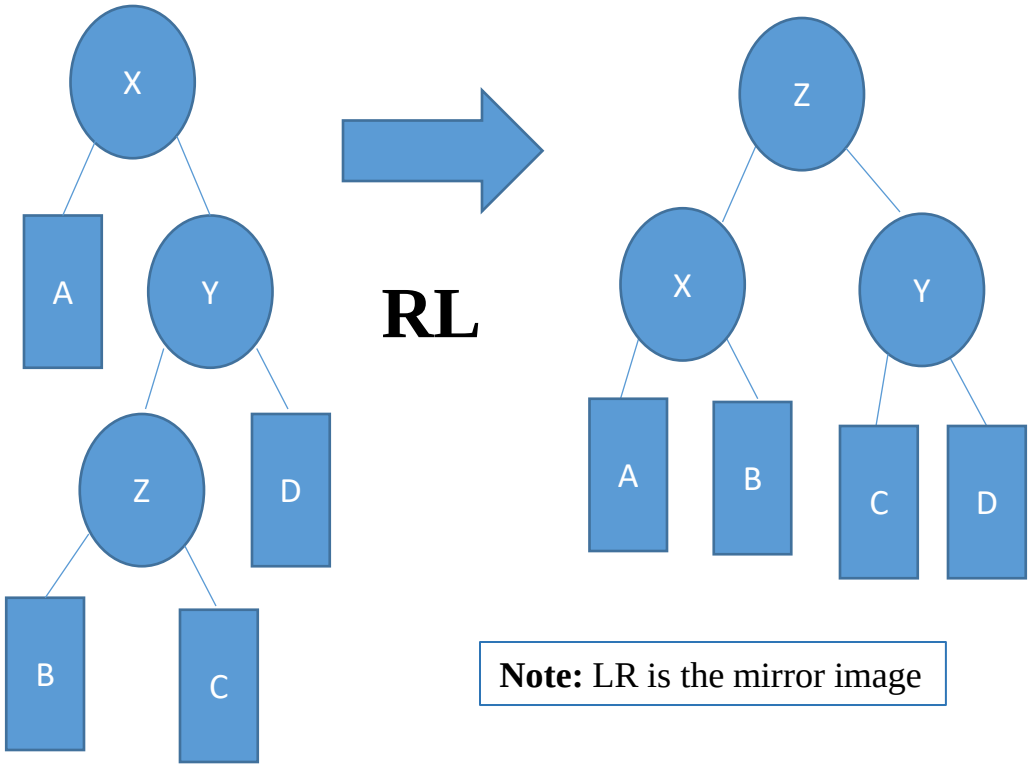
- LL, RR Rotation
  - Child node is raised one level
- RL, LR Rotation
  - Child node is raised two levels in two steps
- Splay Tree Rotation
  - Outer nodes of grandparent nodes are raised two levels.

Outer R and L rotation



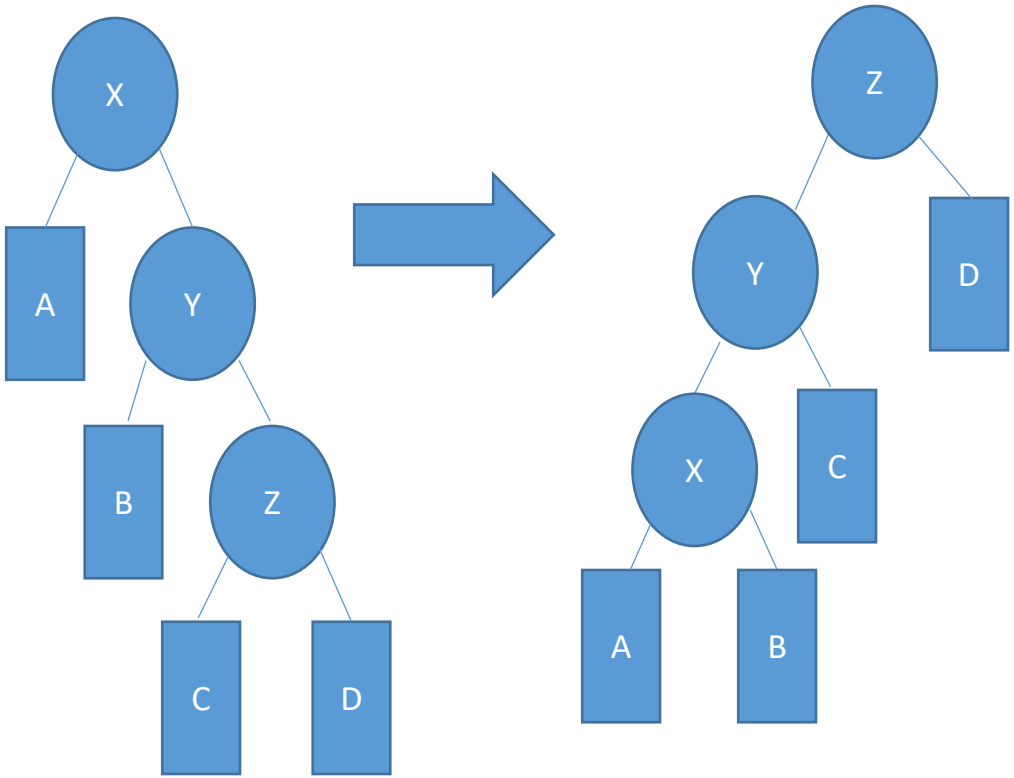
Note: The squares are subtrees, the circles are nodes

Inner LR and RL rotation



Note: LR is the mirror image

Outer Splay Rotation



Note: There is also a rotate for the mirror image

Splay Tree

- A **splay tree** is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality.
- A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. Splay trees can provide good performance in this situation.

Splay Tree

- A splay tree is a binary search tree. But is slightly different from a simple BST.
- whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant.
- If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.

Weight-Balanced Trees

Weight –

The number of leaves of a tree.

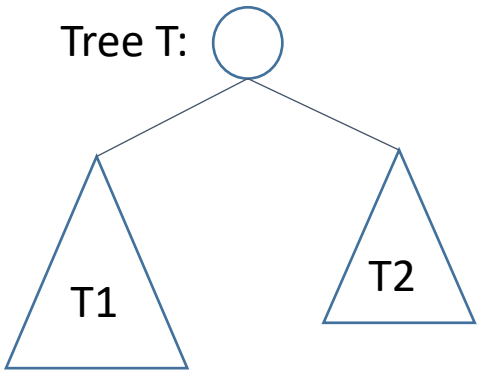
Weight-Balanced Tree –

The weight of the right and left subtree in each node differ by at most one.

Weight-Balanced Trees

- α-weight-balanced trees

  For each subtree, the left and right sub-subtrees has each at least a fraction of α of total weight of the subtree.



$$\alpha W_T \le W_{T1} \le (1-\alpha) W_T$$

$$\alpha W_T \le W_{T2} \le (1-\alpha) W_T$$

Weight-Balanced Trees

The weight-balanced tree structure supports search, insert, and delete in O(logn) time.

Search => O(logn)

Insert => search + insert + rebalance => O(logn)

O(logn) O(1) O(logn)

Delete => search + delete + rebalance => O(logn)

O(logn) O(1) O(logn)

Weight Balanced Trees

- **weight-balanced binary trees** (**WBTs**) are a type of self-balancing binary search trees that can be used to implement dynamic sets, dictionaries (maps) and sequences.

- These trees were introduced by Nievergelt and Reingold in the 1970s as **trees of bounded balance**, or **BB[α] trees**. Their more common name is due to Knuth.

Weight Balanced Trees

- A weight-balanced tree is a binary search tree that stores the sizes of subtrees in the nodes. That is, a node has fields
  - *key*, of any ordered type
  - *value* (optional, only for mappings)
  - *left*, *right*, pointer to node
  - *size*, of type integer.
- By definition, the size of a leaf (typically represented by a NULL pointer) is zero. The size of an internal node is the sum of sizes of its two children, plus one (size[n] = size[n.left] + size[n.right] + 1). Based on the size, one defines the weight as weight[n] = size[n] + 1.[a]

Weight Balanced Trees

- Operations that modify the tree must make sure that the weight of the left and right subtrees of every node remain within some factor α of each other, using the same rebalancing operations used in AVL trees: rotations and double rotations. Formally, node balance is defined as follows:
- A node is α-weight-balanced if

  weight[n.left] ≥ α·weight[n] ≥weight[n.right]

Here, α is a numerical parameter to be determined when implementing weight balanced trees.

Weight Balanced Trees

- Lower values of α produce "more balanced" trees, but not all values of α are appropriate; Nievergelt and Reingold proved that

- is a necessary condition for the balancing algorithm to work $\alpha < 1 - \frac{1}{\sqrt{2}}$
- Applying balancing correctly guarantees a tree of *n* elements will have height

$$h \leq \log_{\frac{1}{1-\alpha}} n = \frac{\log_2 n}{\log_2 \left(\frac{1}{1-\alpha}\right)} = O(\log n)$$

Huffman Coding:
An Application of Binary Trees and Priority Queues

Why Compression of Data is required?

- BCD 4 bit
- EBCDIC 6 bit
- ASCII-7 7 bit
- ASCII-8 8 bit
- UTF -16, 32, 64
- bit fixed length code

Purpose of Huffman Coding

- Proposed by Dr. David A. Huffman in 1952
  - *"A Method for the Construction of Minimum Redundancy Codes"*
- Applicable to many forms of data transmission
  - Our example: text files

Huffman Tree

- A *Huffman tree* represents *Huffman codes* for characters that might appear in a text file
- As opposed to ASCII or Unicode, Huffman code uses different numbers of bits to encode letters
- More common characters use fewer bits
- Many programs that compress files use Huffman codes

The Basic Algorithm

- Huffman coding is a form of statistical coding
- Not all characters occur with the same frequency!
- Yet all characters are allocated the same amount of space
  - 1 char = 1 byte, be it e or x

The Basic Algorithm

- Any savings in tailoring codes to frequency of character?
- Code word lengths are no longer fixed like ASCII.
- Code word lengths vary and will be shorter for the more frequently used characters.

The (Real) Basic Algorithm

1. Scan text to be compressed and tally occurrence of all characters.

2. Sort or prioritize characters based on number of occurrences in text.

3. Build Huffman code tree based on prioritized list.

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
Prioritize characters

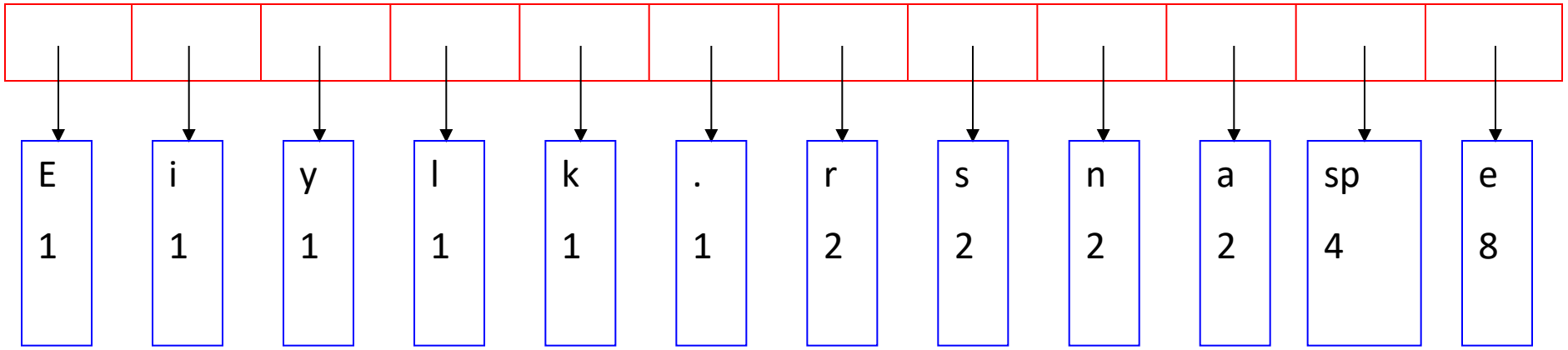
- Uses binary tree nodes

```
public class HuffNode
{
   public char myChar;
   public int myFrequency;
   public HuffNode myLeft, myRight;
}

priorityQueue myQueue;
```

Building a Tree

- The queue after inserting all nodes

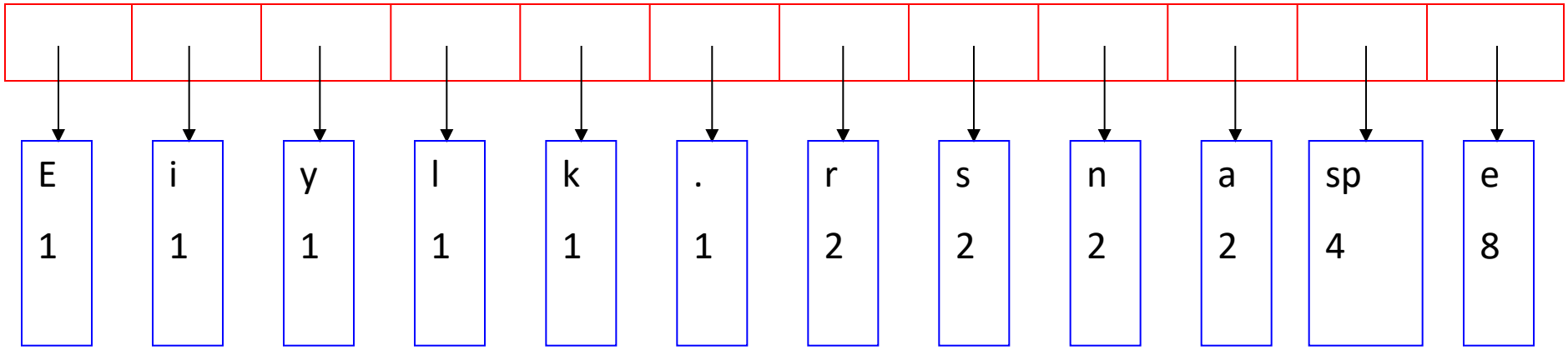


- Null Pointers are not shown

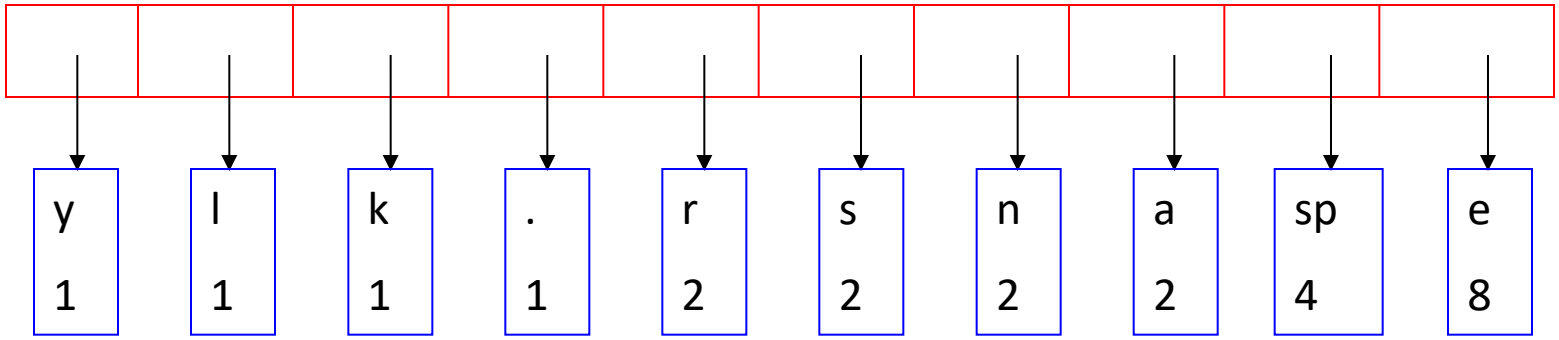
Building a Tree

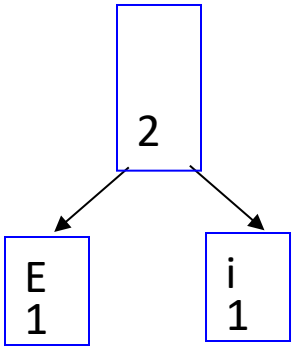
- While priority queue contains two or more nodes
  - Create new node
  - Dequeue node and make it left subtree
  - Dequeue next node and make it right subtree
  - Frequency of new node equals sum of frequency of left and right children
  - Enqueue new node back into queue

Building a Tree

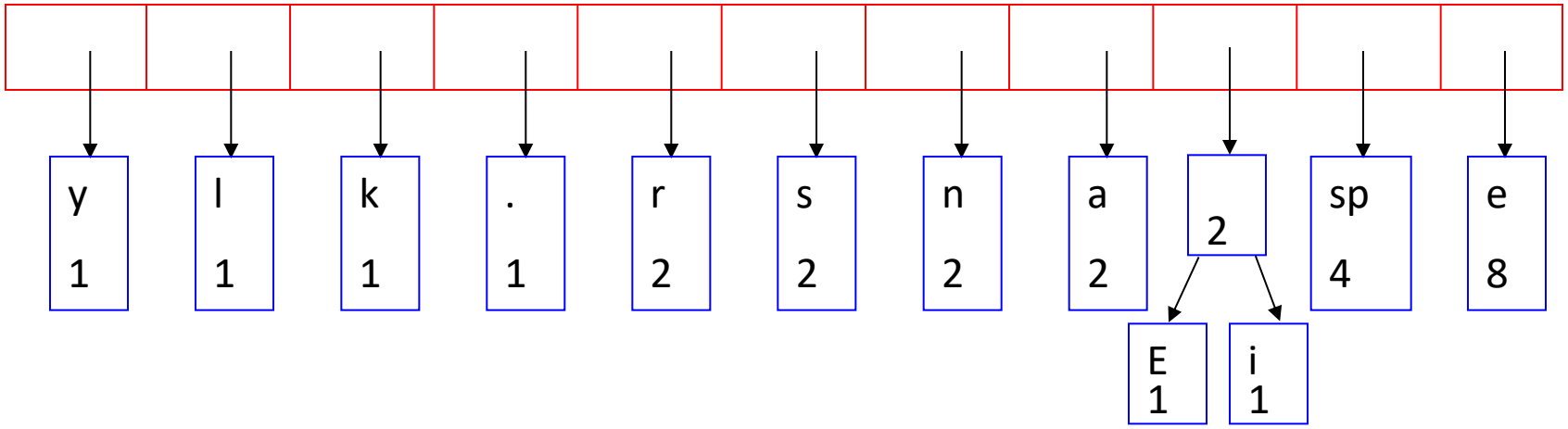


Building a Tree

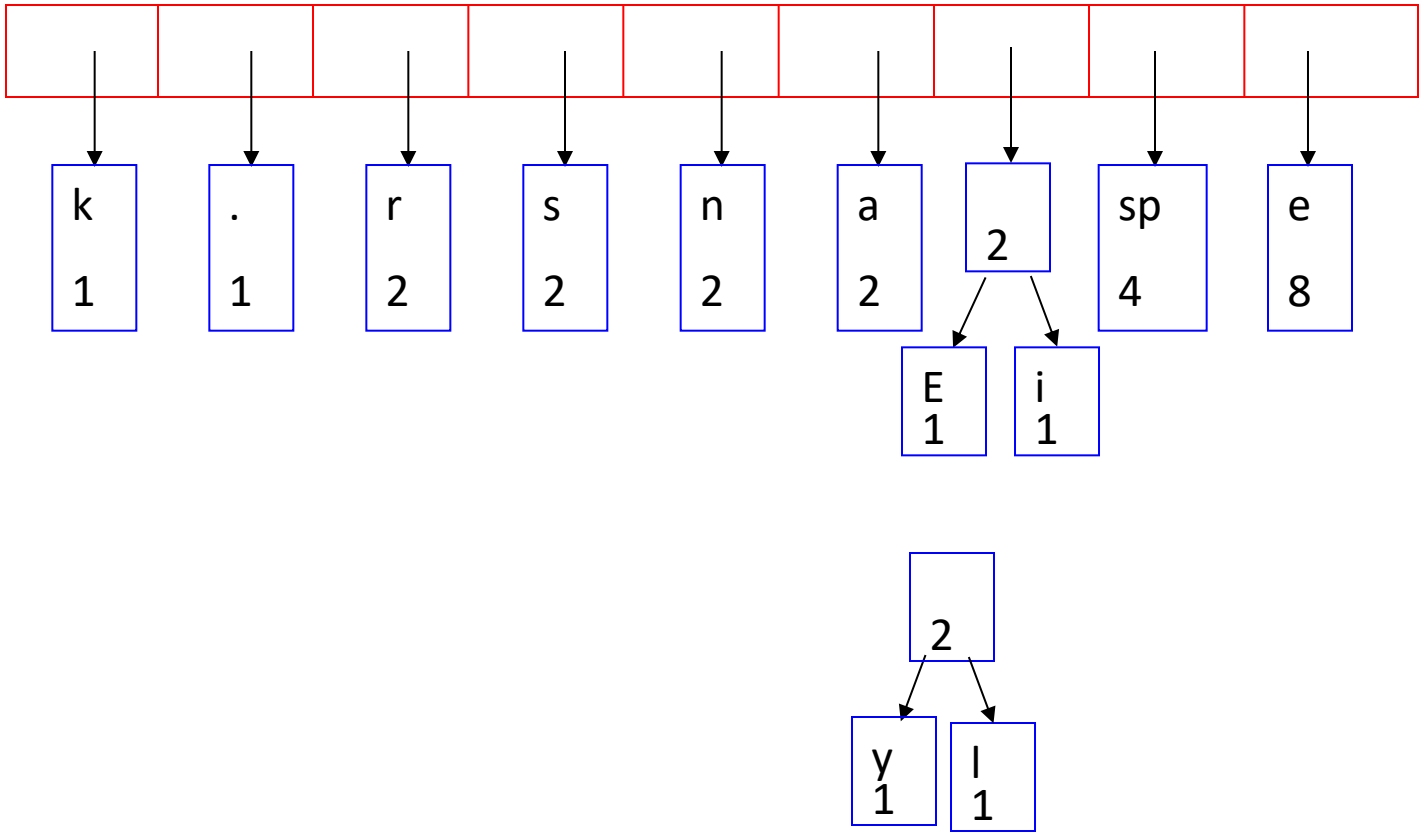




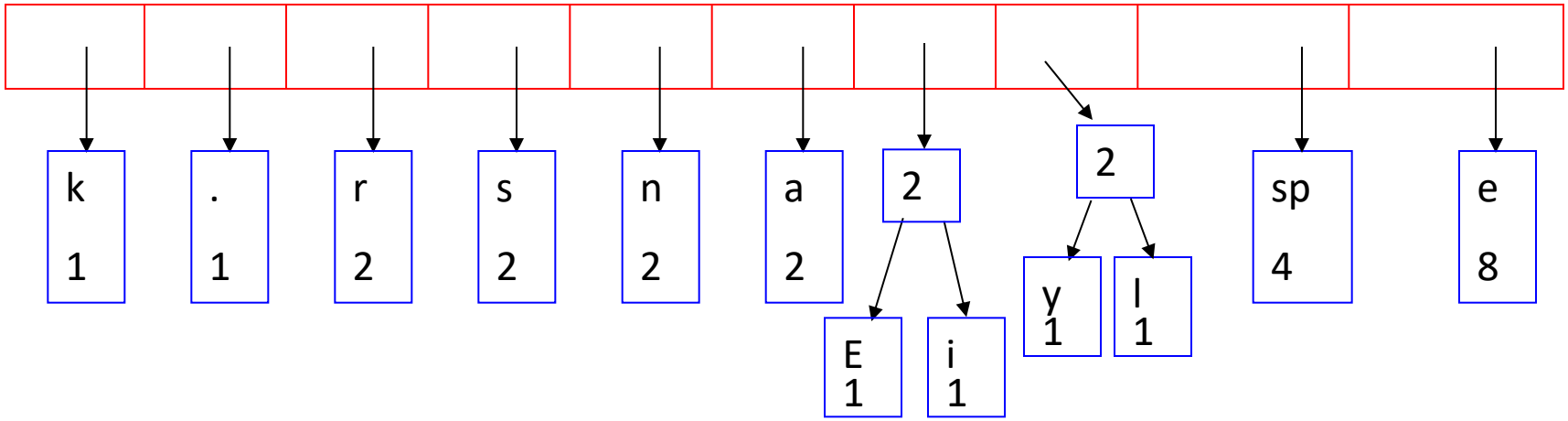
Building a Tree



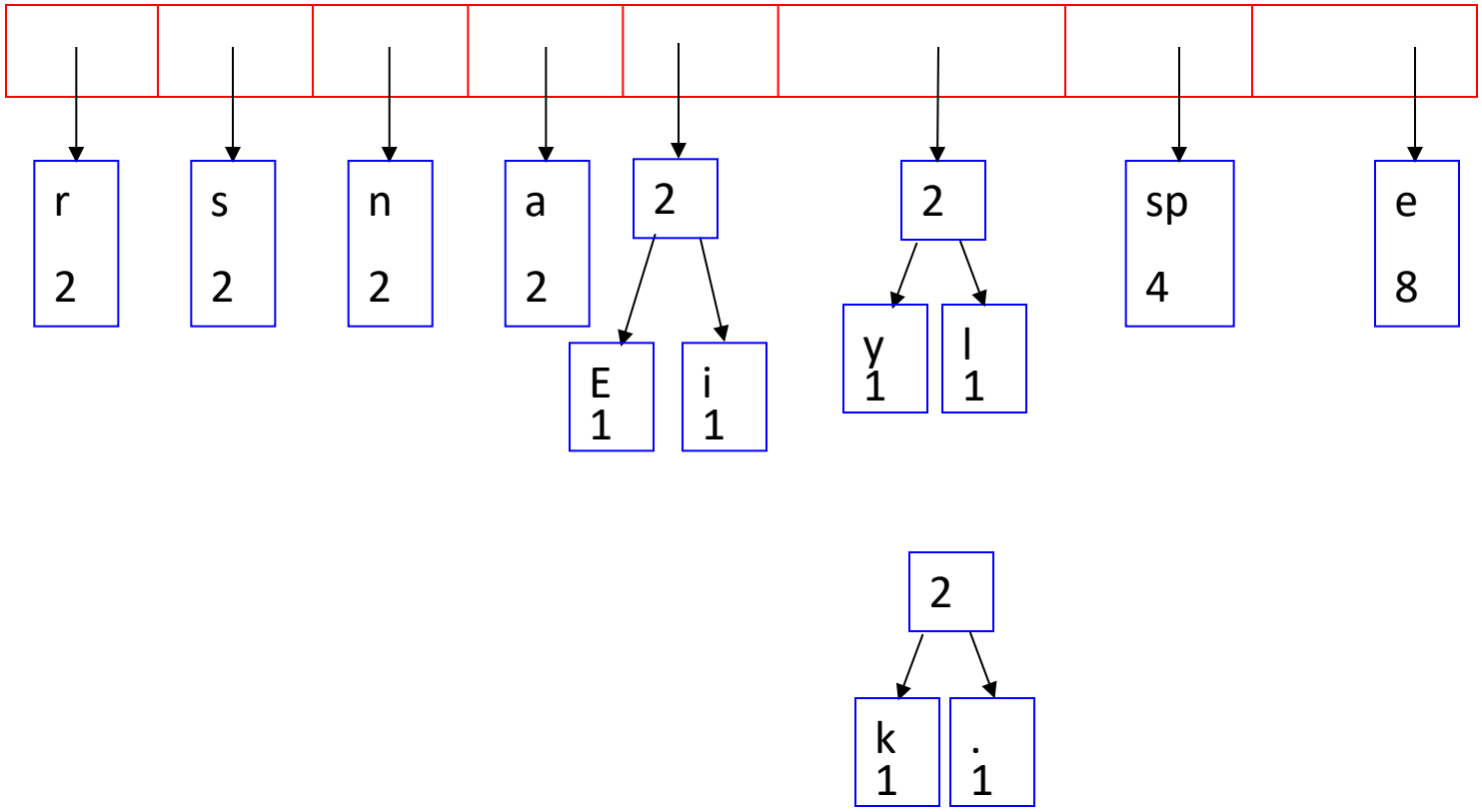
Building a Tree



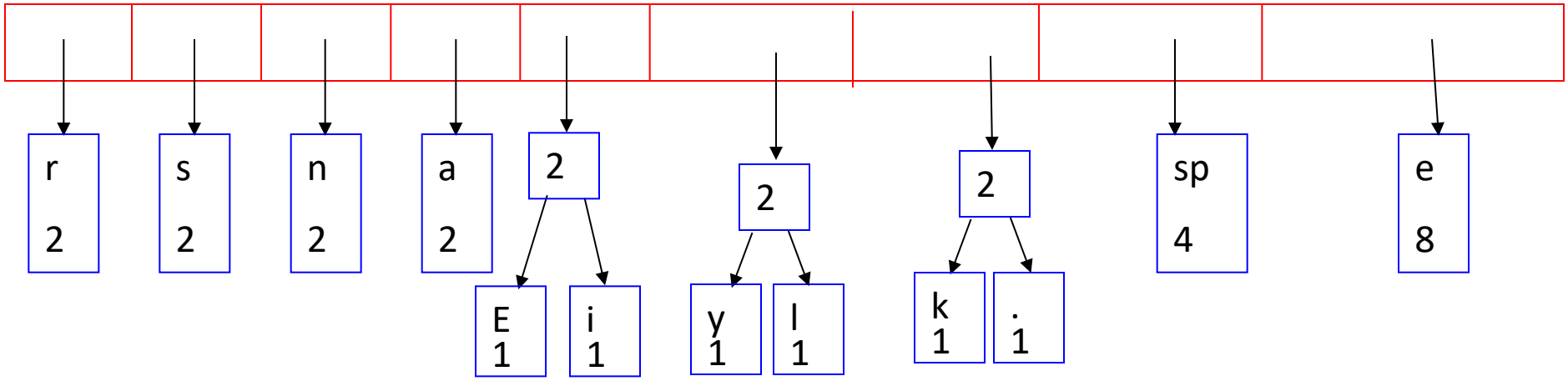
Building a Tree



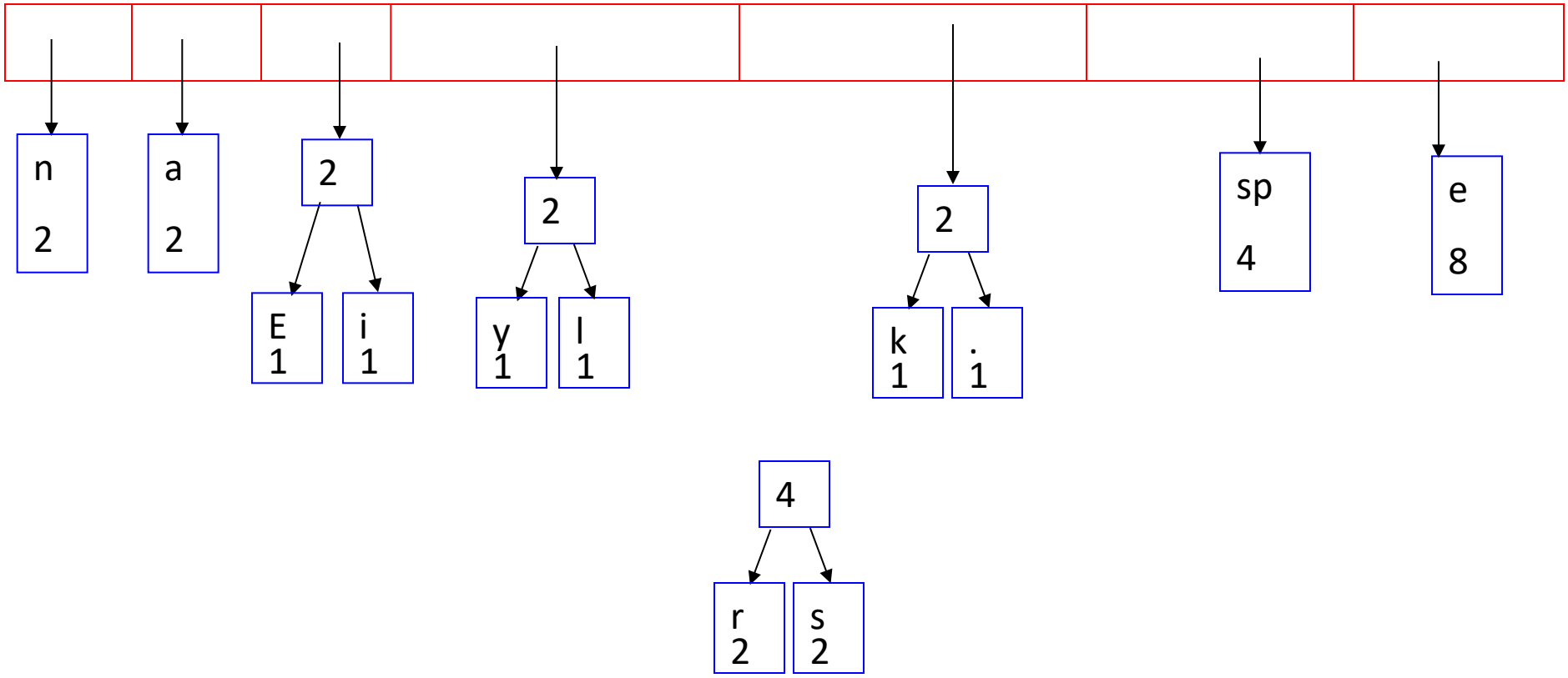
Building a Tree



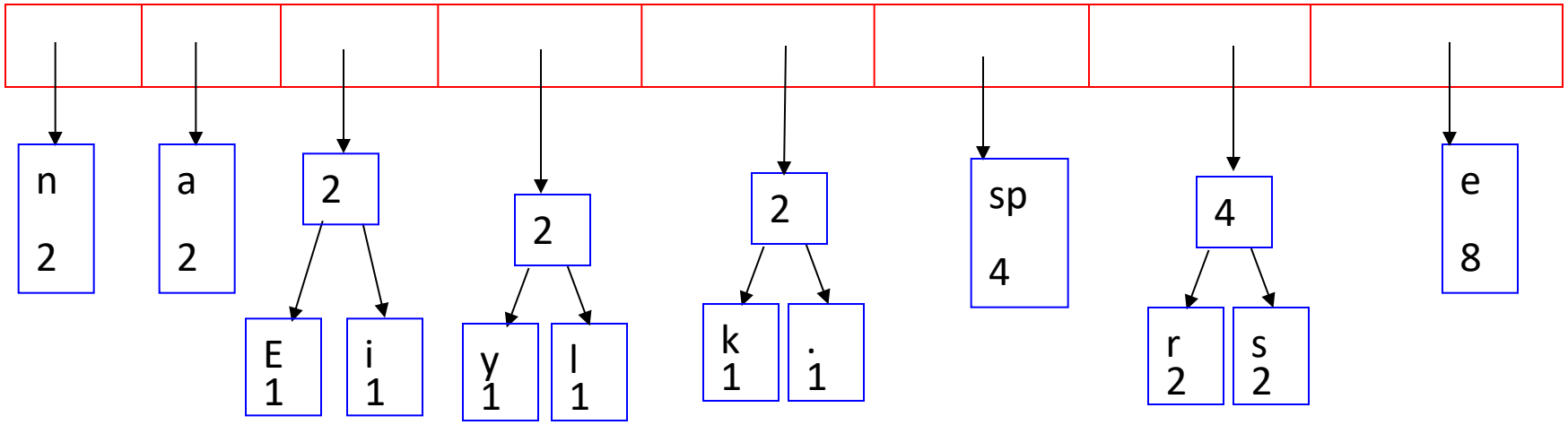
Building a Tree



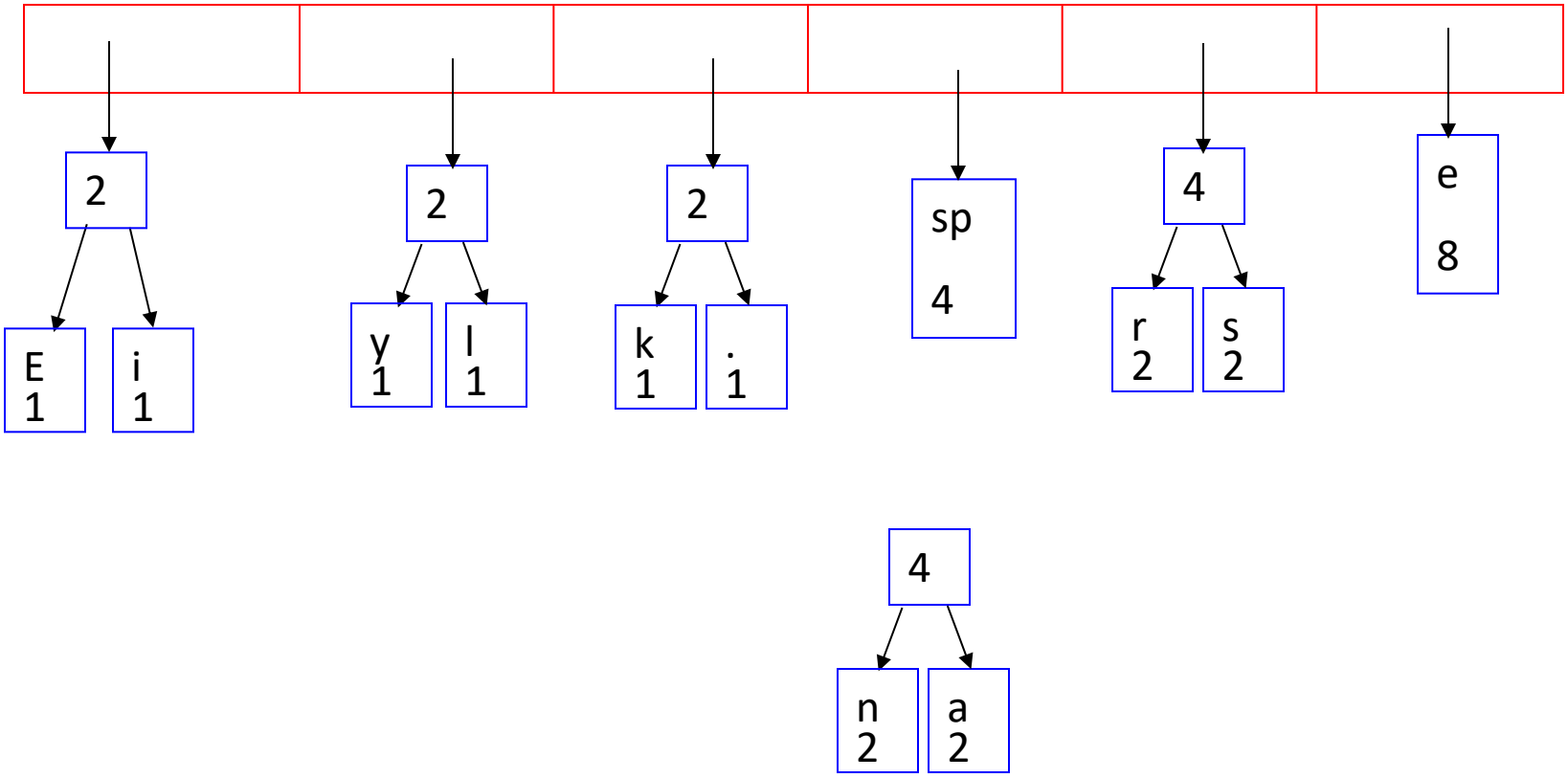
Building a Tree



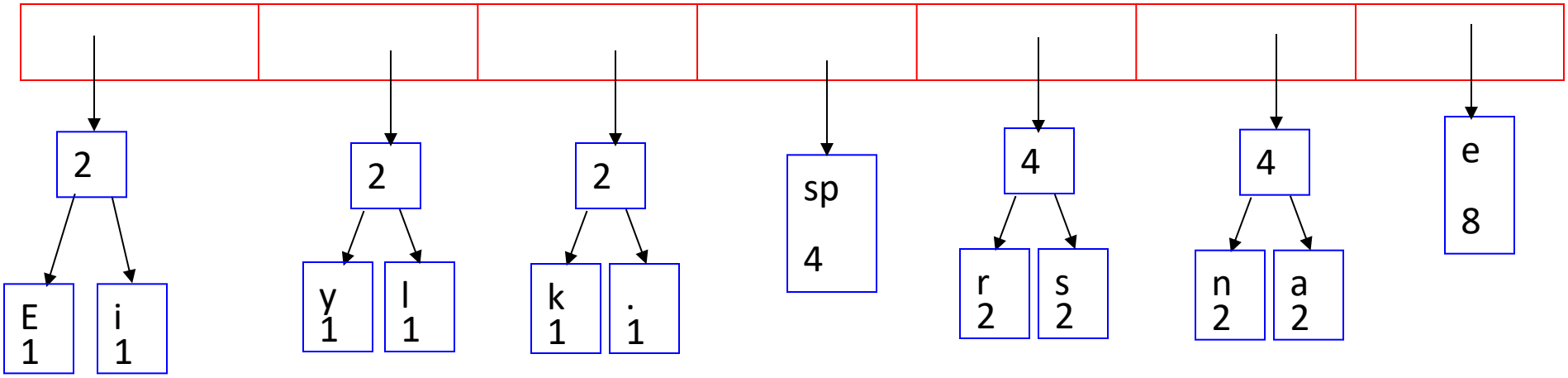
Building a Tree



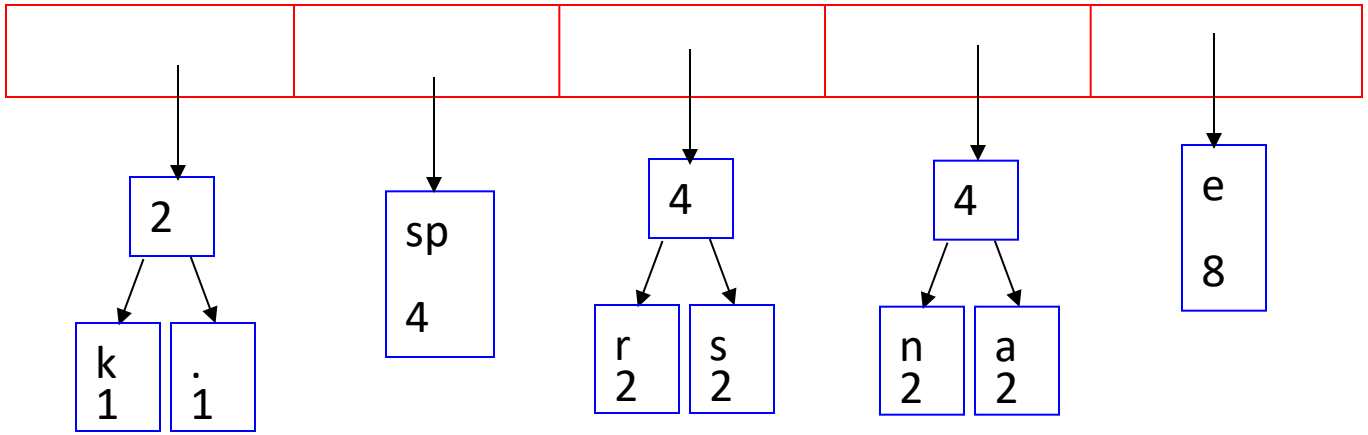
Building a Tree

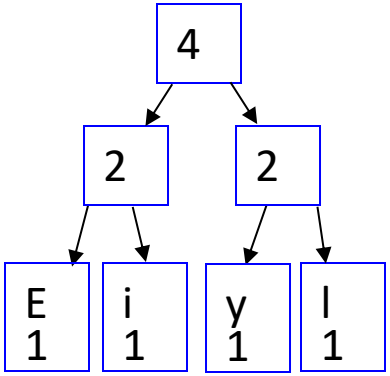


Building a Tree

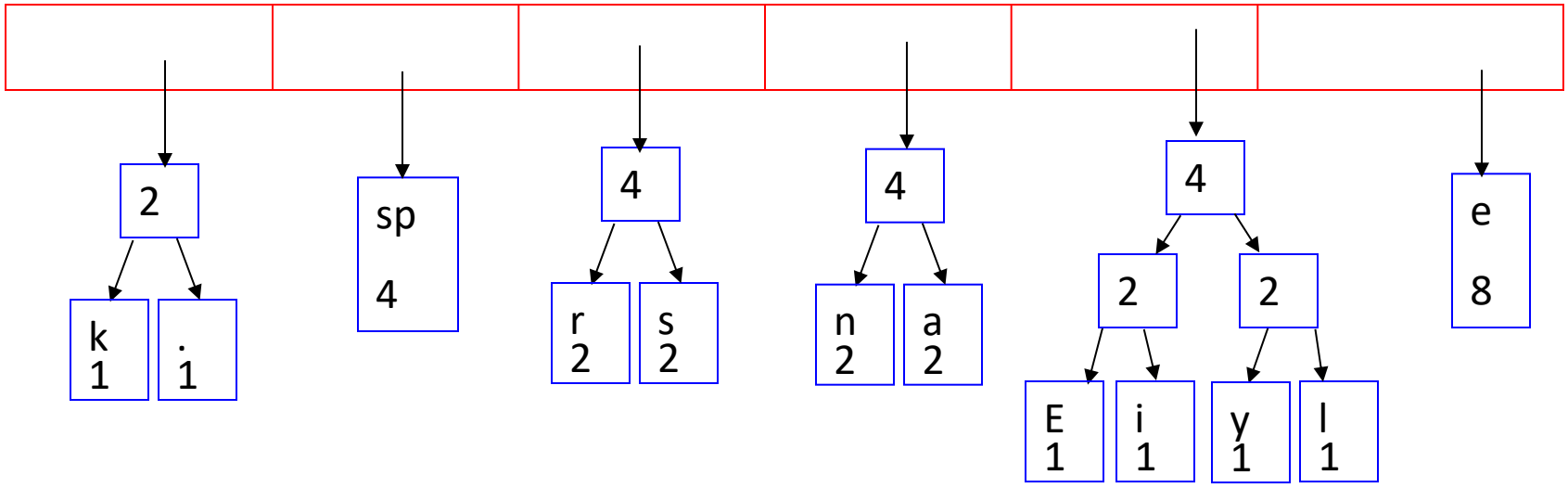


Building a Tree

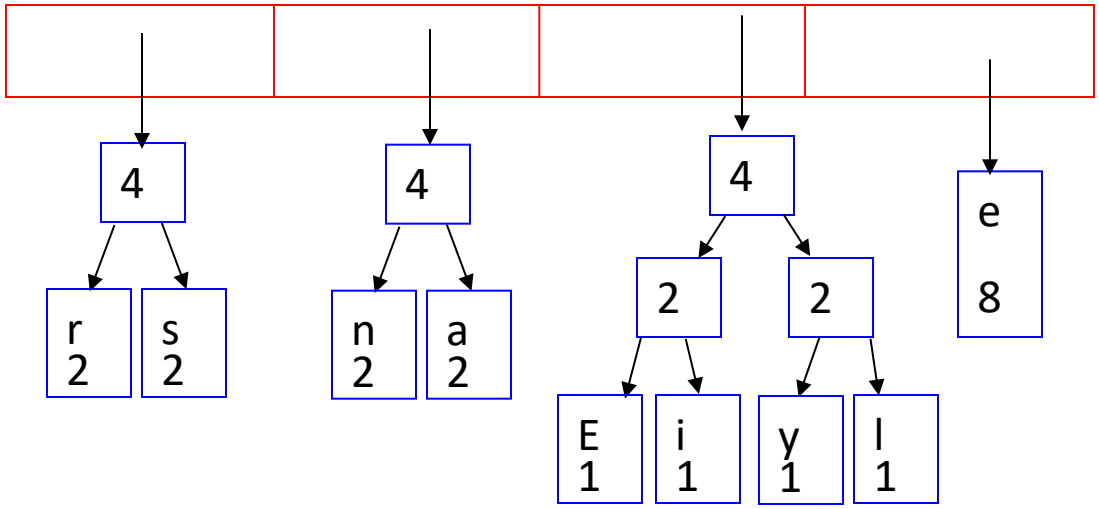


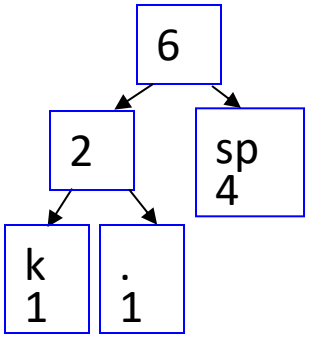


Building a Tree

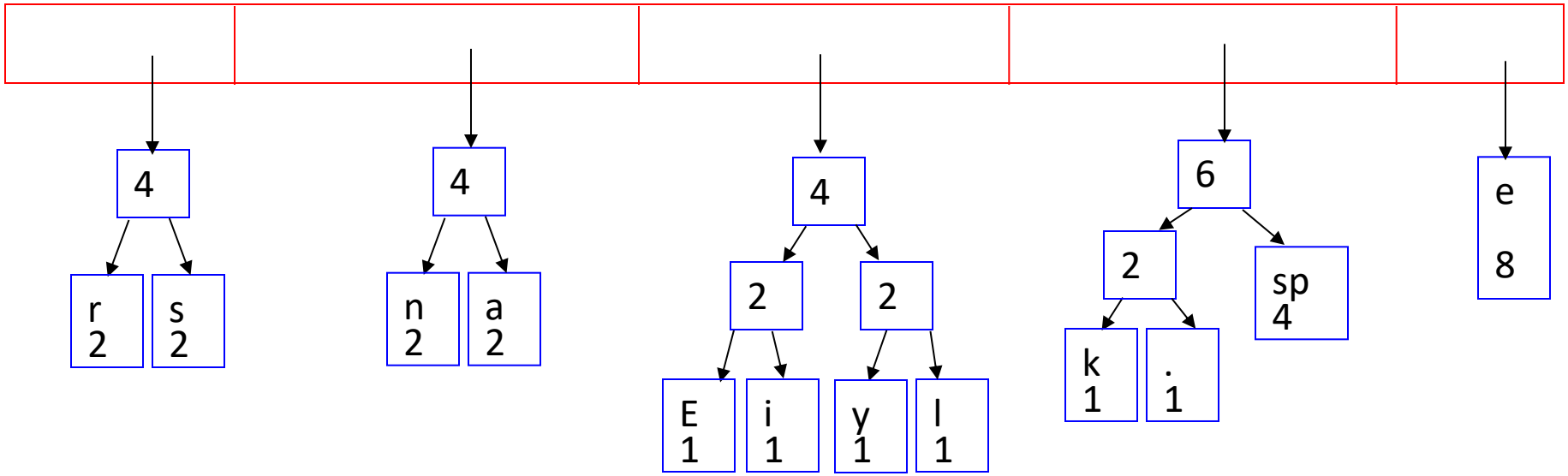


Building a Tree



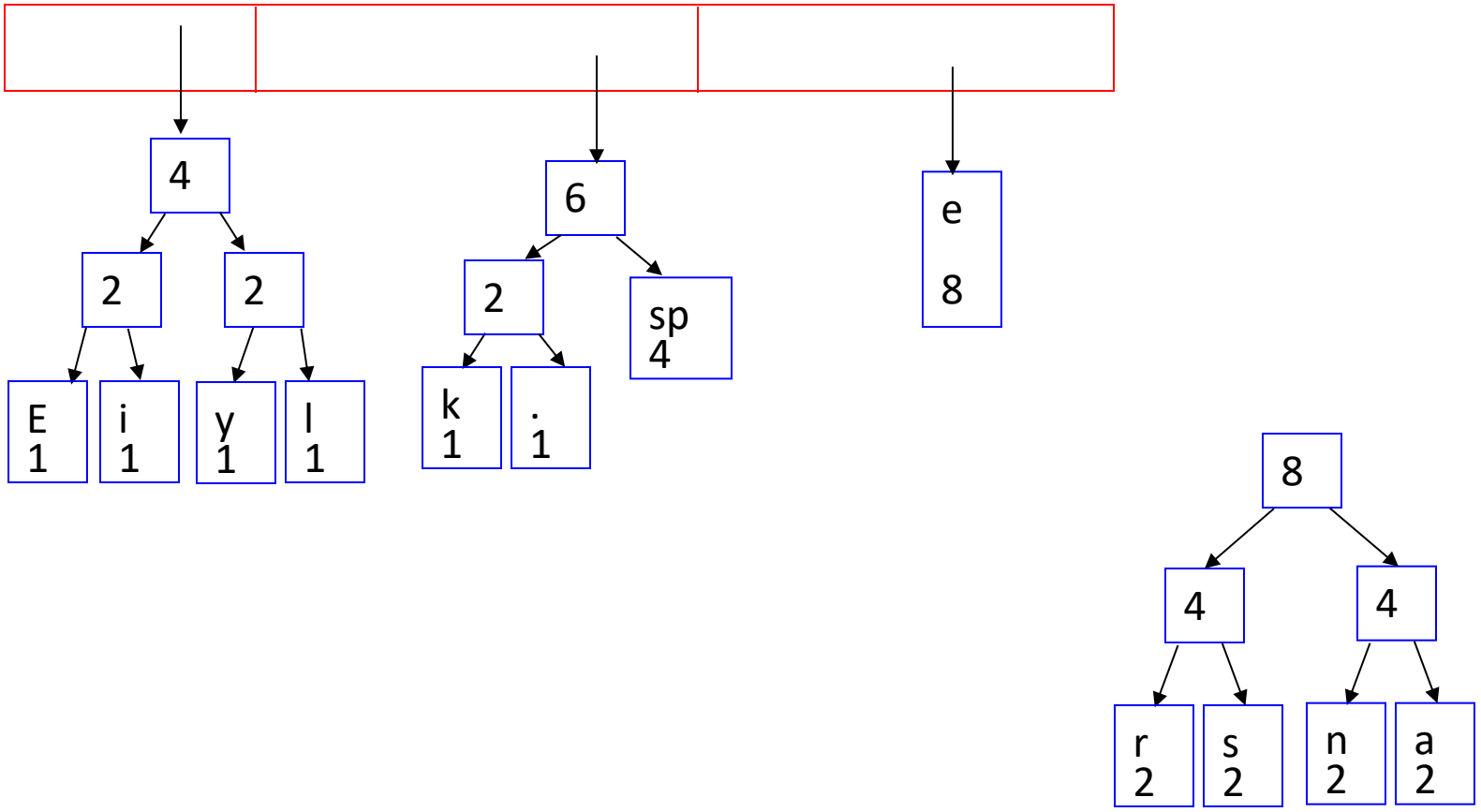


Building a Tree

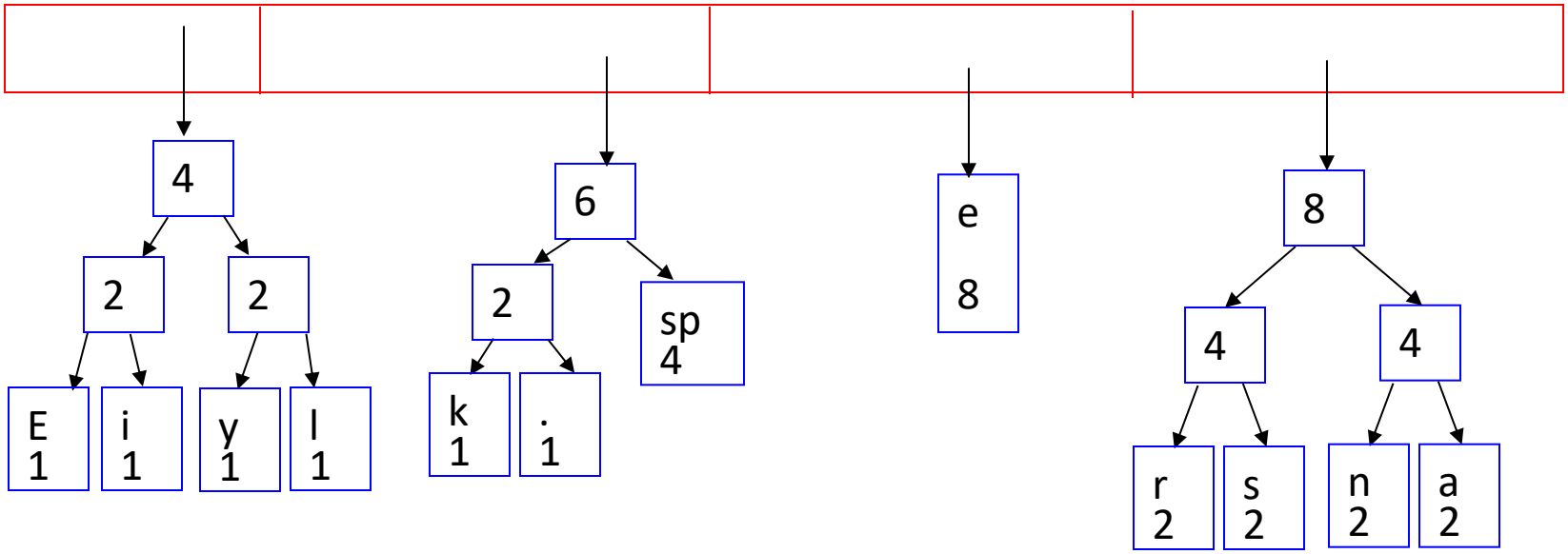


What is happening to the characters with a low number of occurrences?

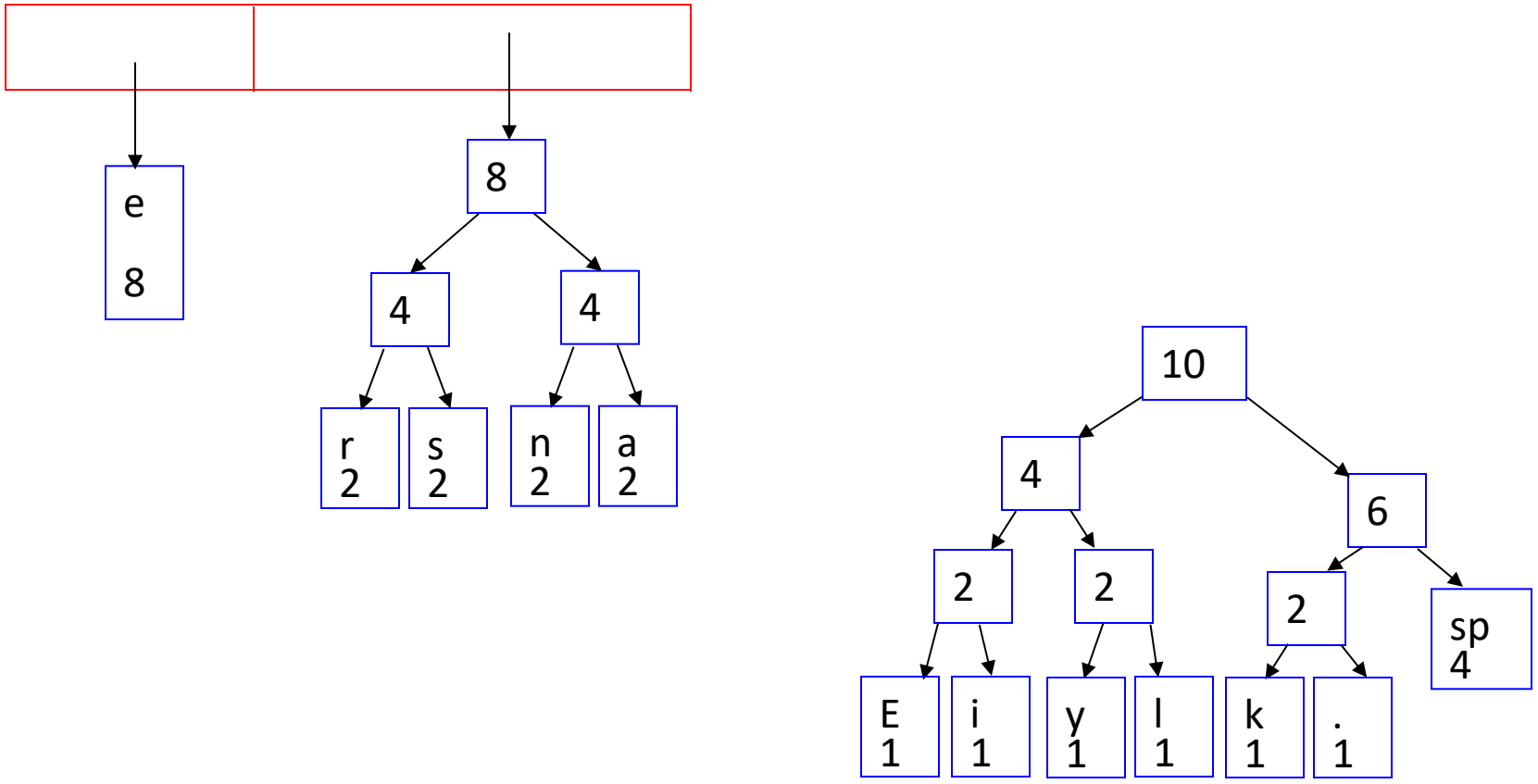
Building a Tree



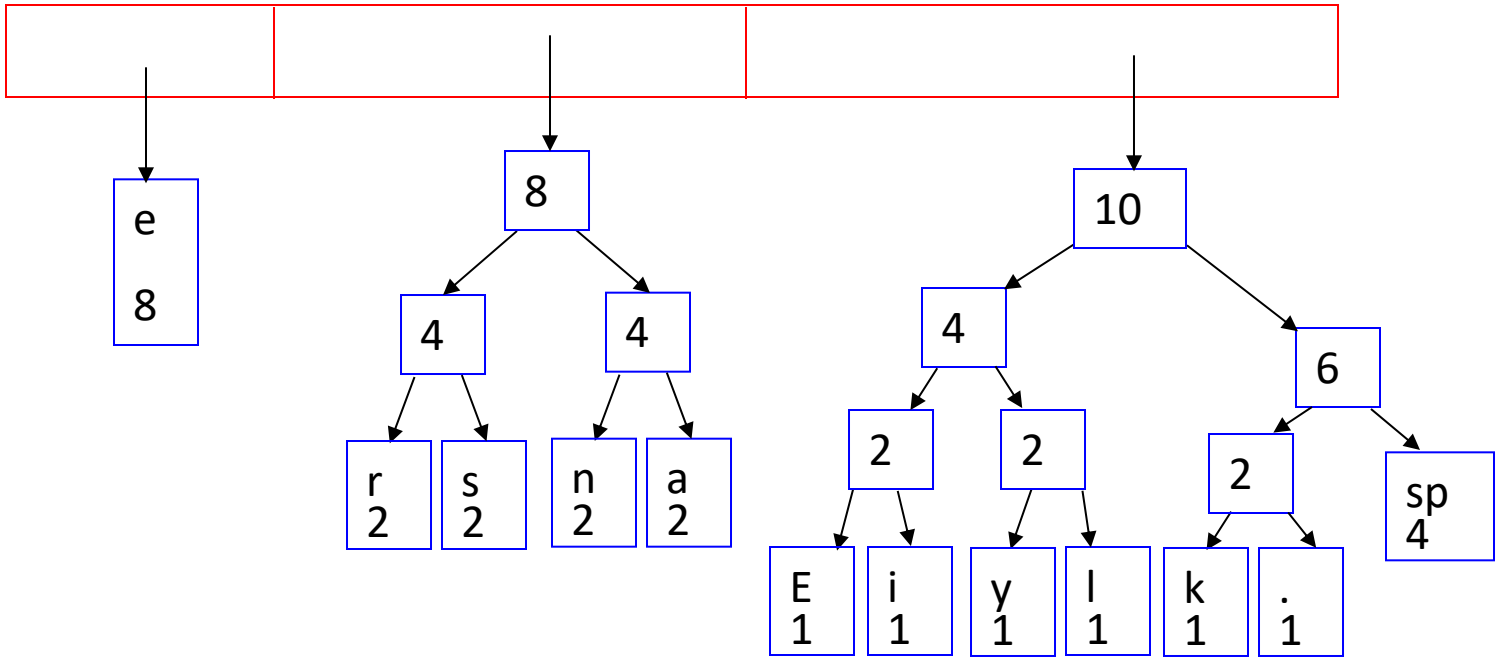
Building a Tree



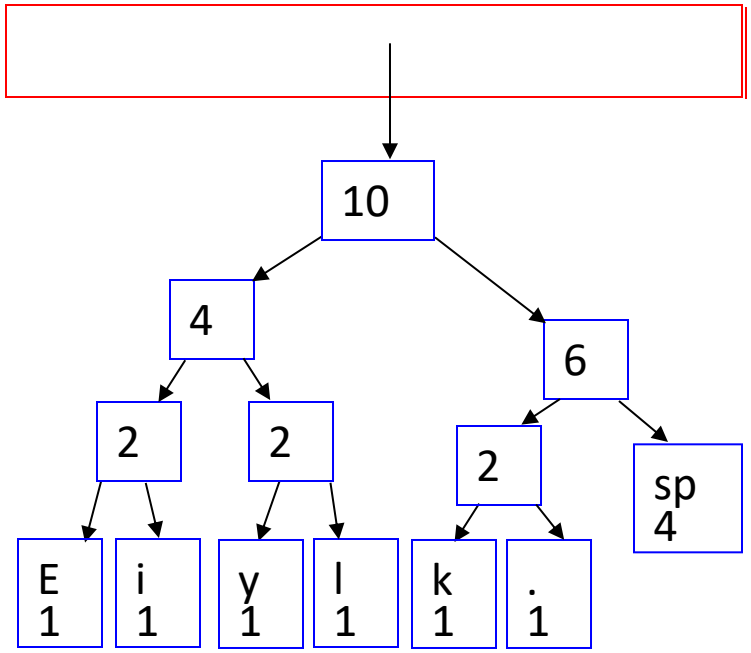
Building a Tree

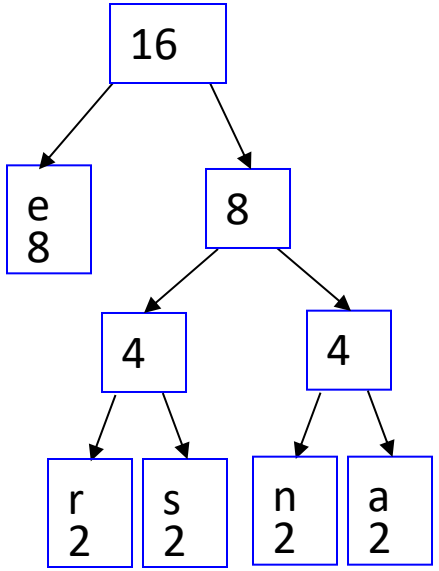


Building a Tree

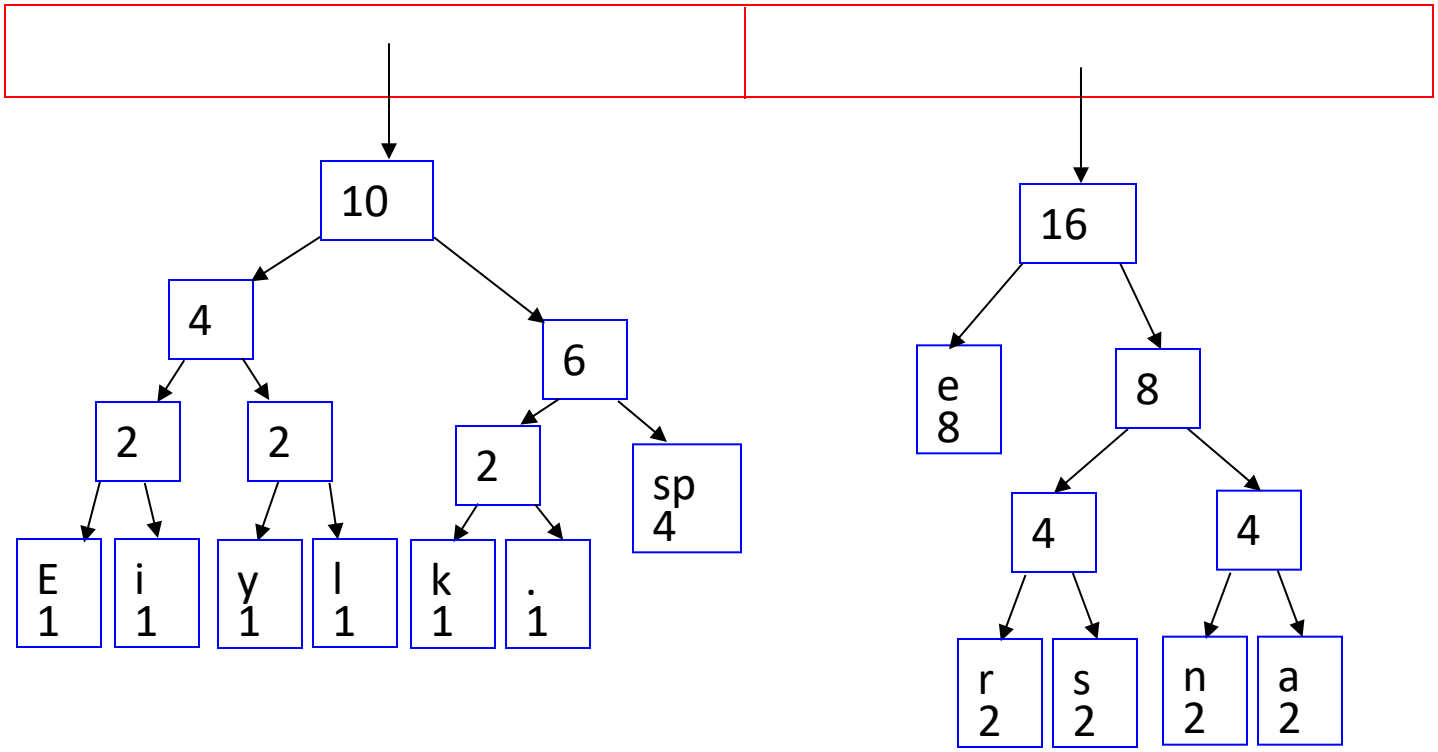


Building a Tree

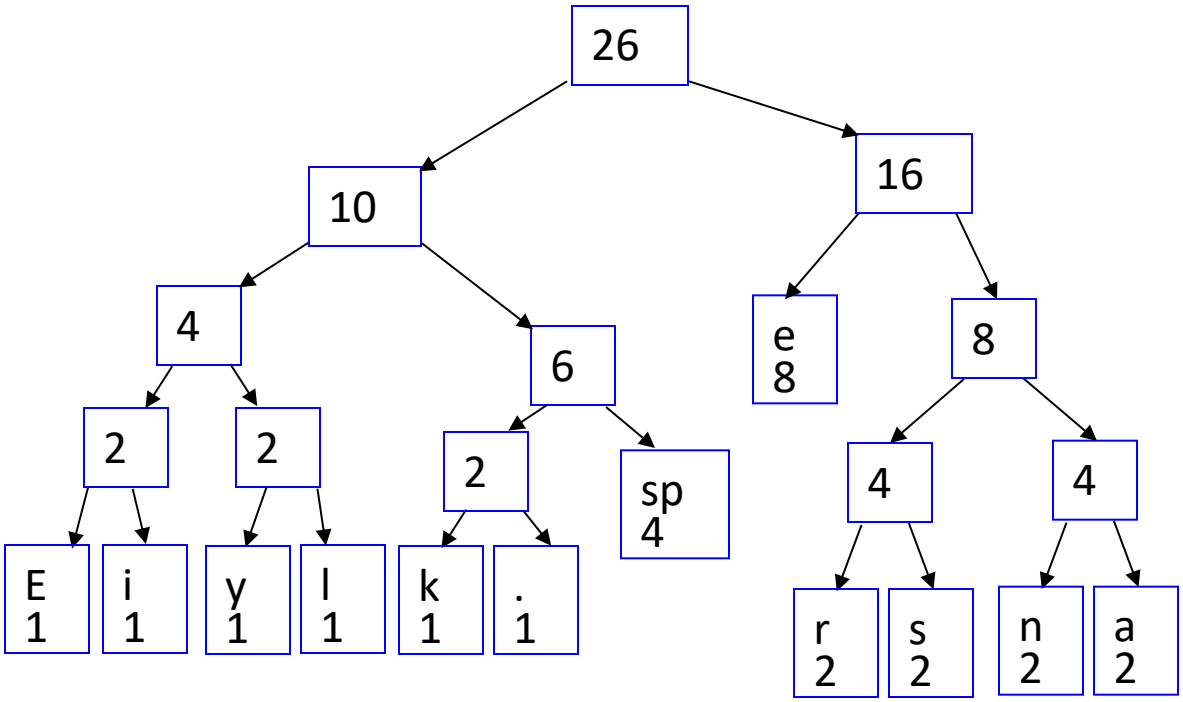




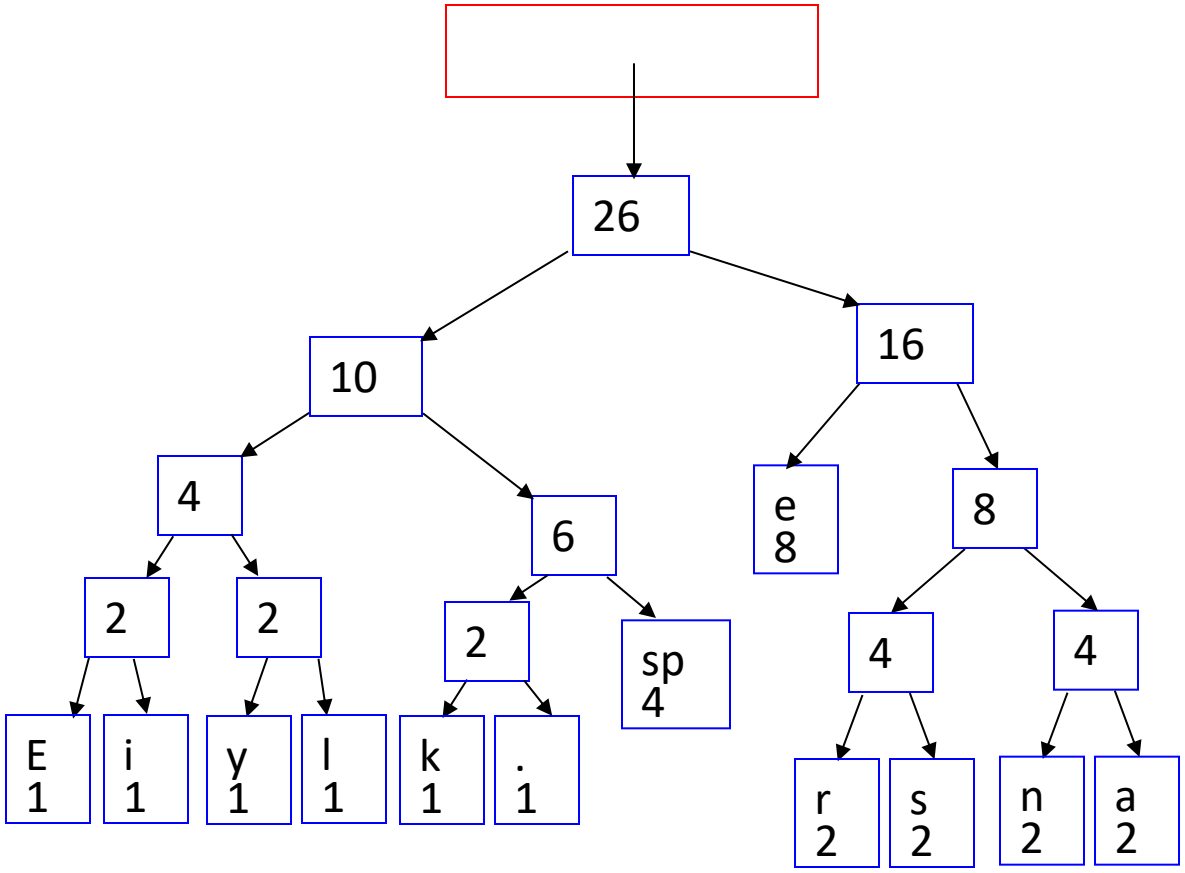
Building a Tree



Building a Tree



Building a Tree



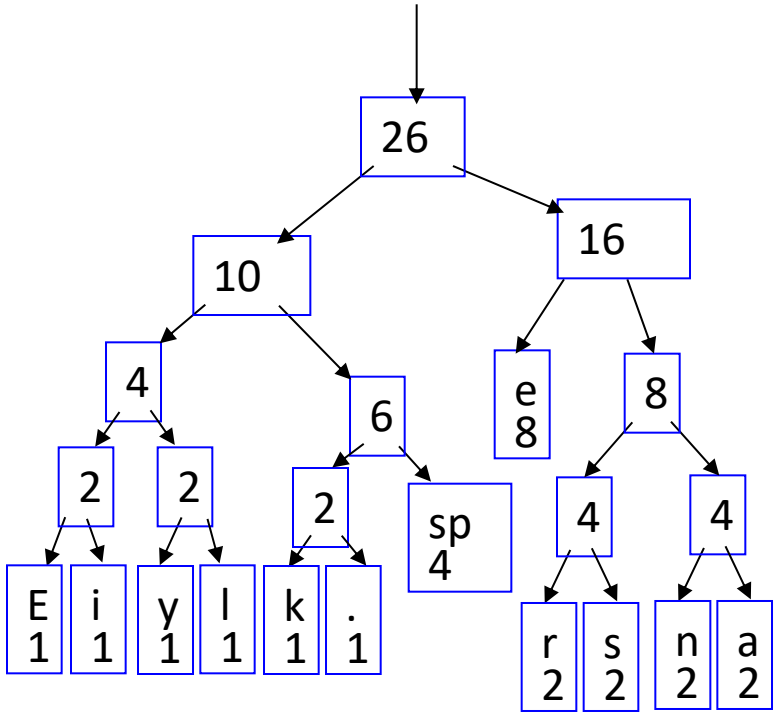
After enqueueing this node there is only one node left in priority queue.

Building a Tree

Dequeue the single node left in the queue.

This tree contains the new code words for each character.

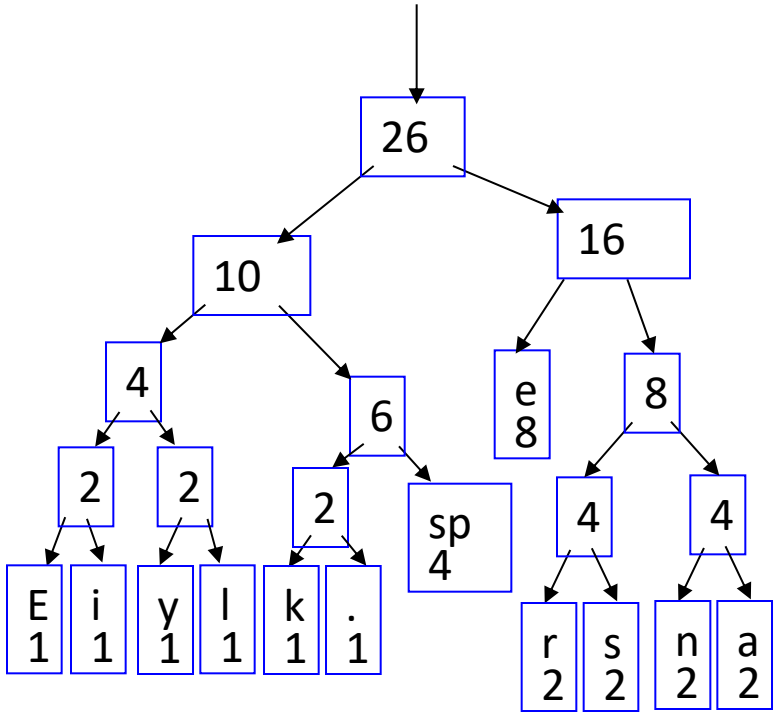
Frequency of root node should equal number of characters in text.



Eerie eyes seen near lake. 26 characters

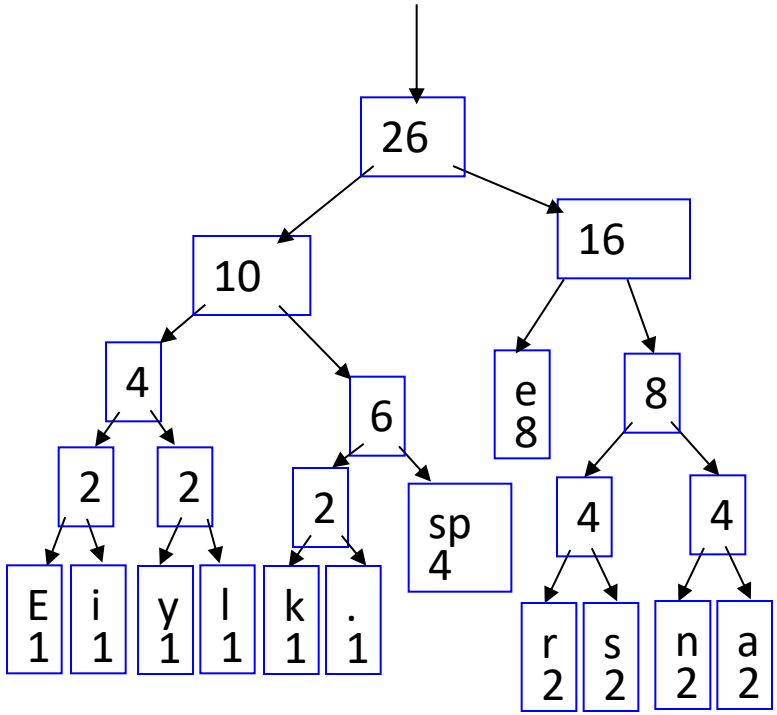
Encoding the File
Traverse Tree for Codes

- 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

- Rescan text and encode file using new code words

Eerie eyes seen near lake.

```
000010110000011001110001010110110100111
101011111100011001111110100100101
```

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

- Why is there no need for a separator character?

.

Encoding the File Results

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

```
0000101100000110011100010101101101001111101011111100011001111111010100101
```

- If modified code used 4 bits per character are needed. Total bits 4 * 26 = 104. Savings not as great.

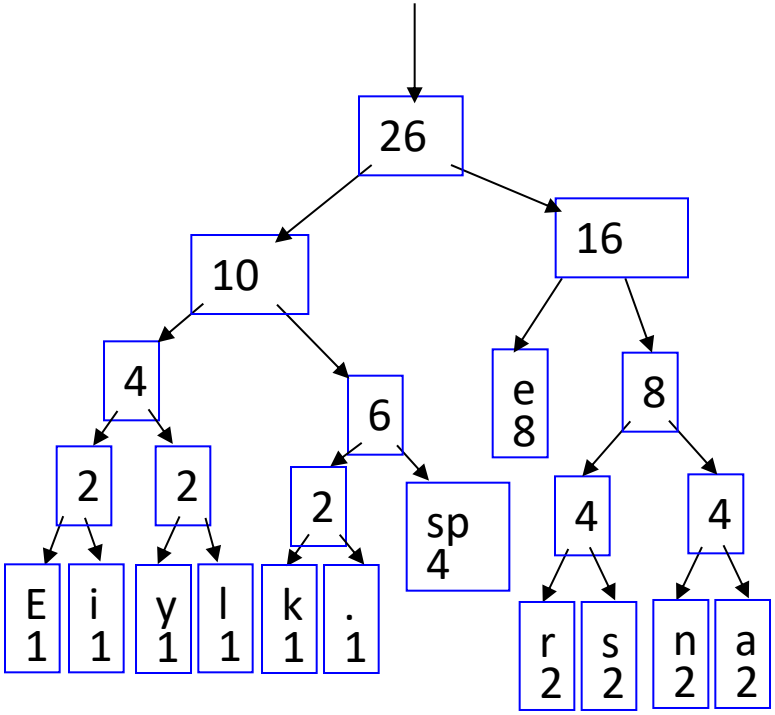
Decoding the File

- How does receiver know what the codes are?
  - We have to send the code table or tree to the receiver so that he an reconvert the code into text.
- Tree constructed for each text file.
  - Considers frequency for each file
  - Big hit on compression, especially for smaller files
- Tree predetermined
  - based on statistical analysis of text files or file types
- Data transmission is bit based versus byte based

Decoding the File

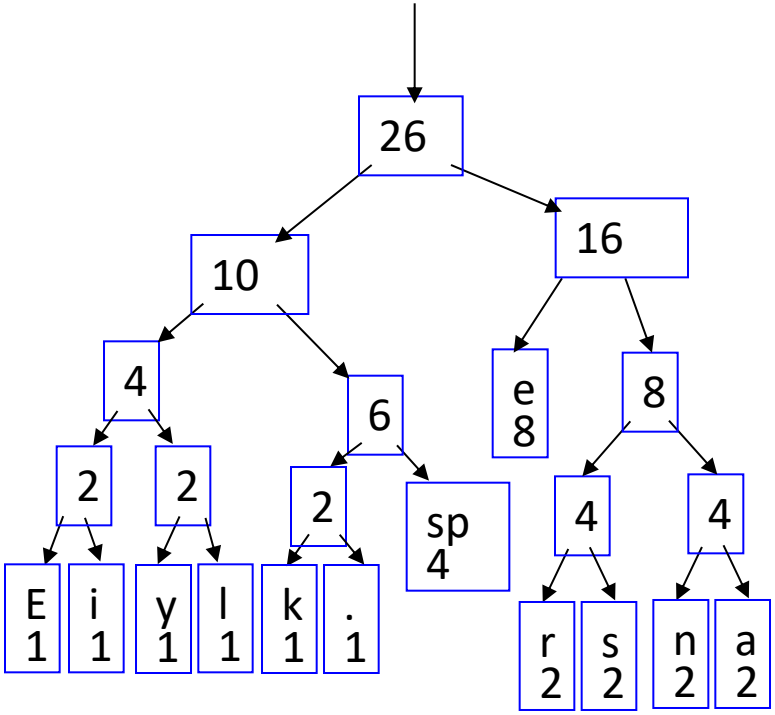
- Once receiver has tree it scans incoming bit stream
- 0 $\Rightarrow$ go left
- 1 $\Rightarrow$ go right

```
00001011000001100111000101011011010011111101011111100011001111100100101
```



Decoding the File

- Once receiver has tree it scans incoming bit stream
- 0 ⇒ go left
- 1 ⇒ go right



000010110000011001110001010
110110100111110101111110001
100111111010010010 1

Eerie eyes seen near lake.

Example

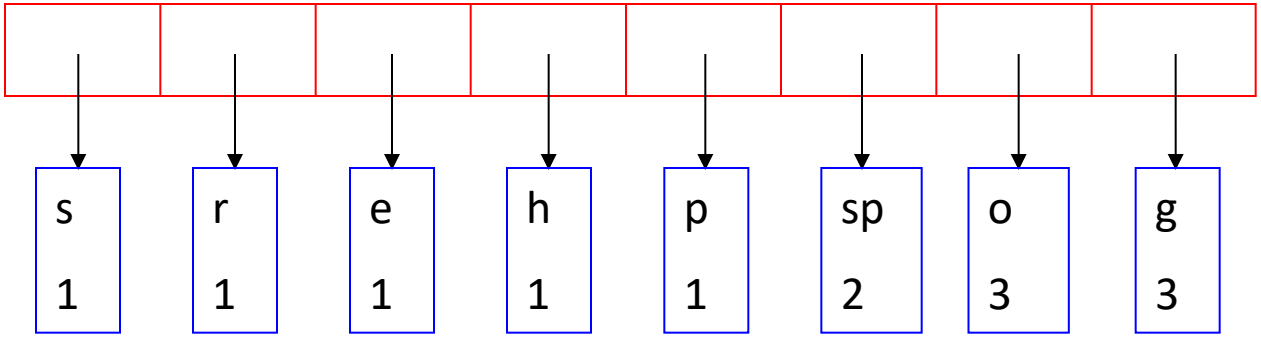
- Encode
  - “go go gophers” using Huffman coding
- Steps
  1. Create a leaf node for each unique character
  2. Calculate the frequency for each charater
  3. Build a min heap of all leaf nodes
  4. Extract two nodes with the minimum frequency from the min heap.
  5. Create a new internal node with a frequency equal to the sum of the two nodes frequencies.
  6. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
  7. Repeat steps until the heap contains only one root node.
  8. The remaining node are arranged as child and the tree is complete.

Frequency Count

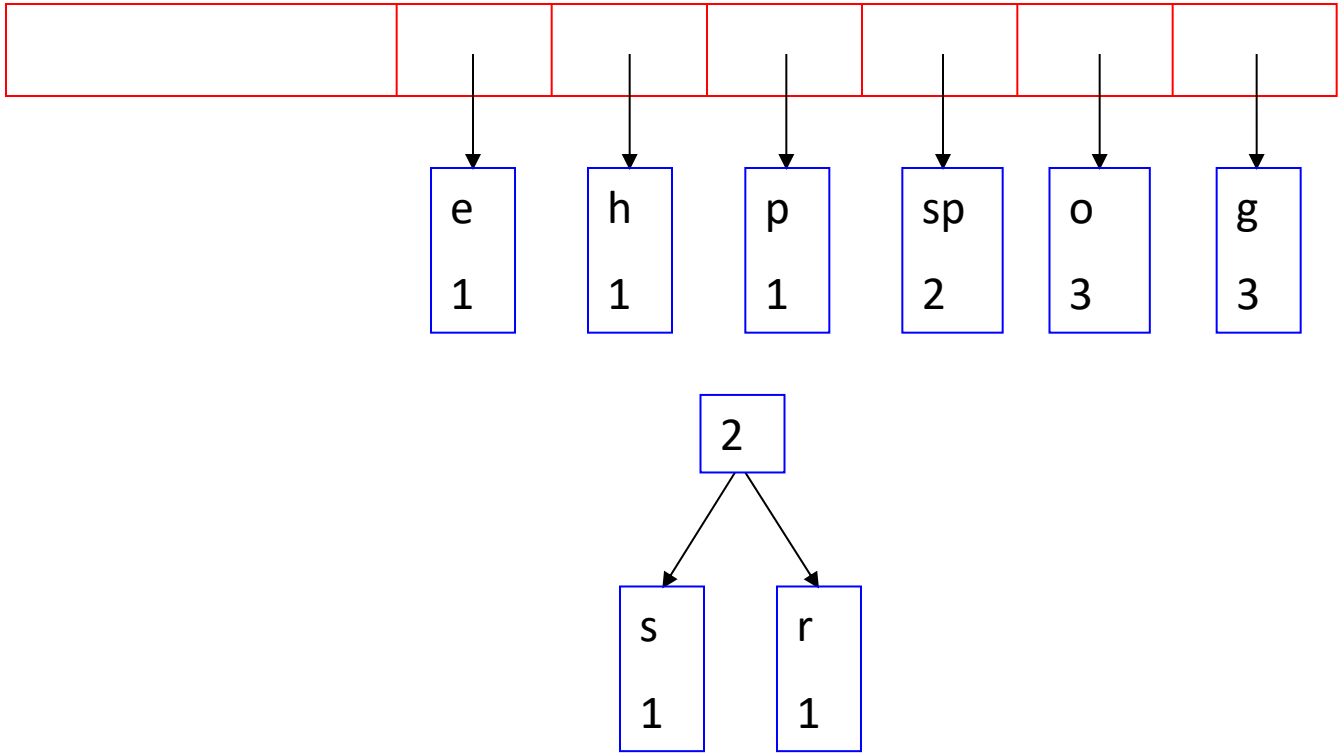
- go go gophers

char	frequency
'g'	3
'o'	3
sp	2
'p'	1
'h'	1
'e'	1
'r'	1
's'	1

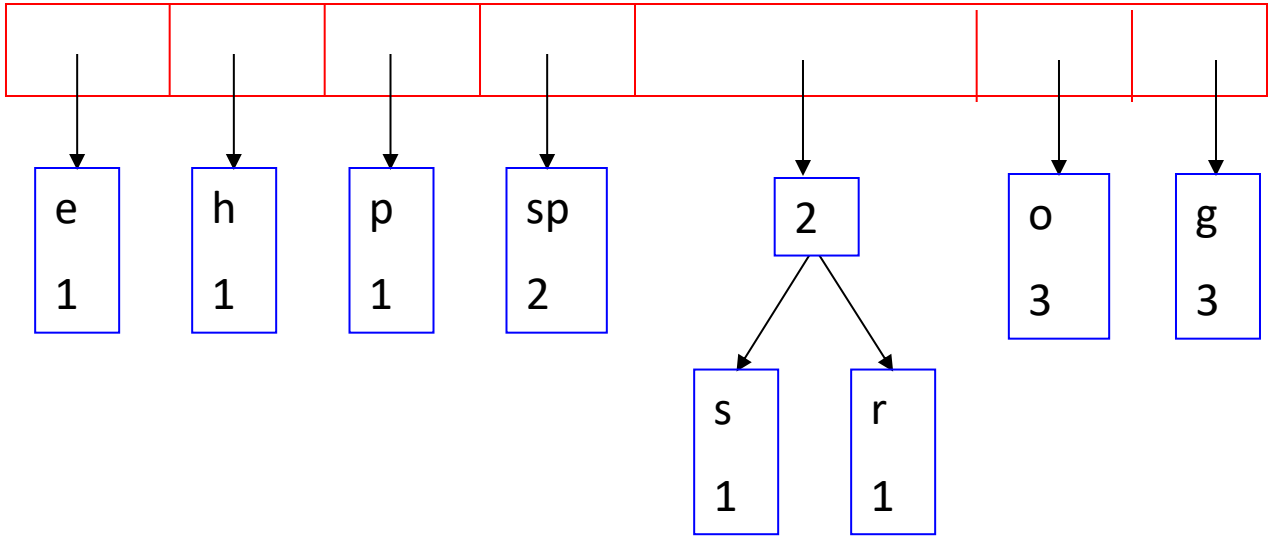
Building the Coding Tree



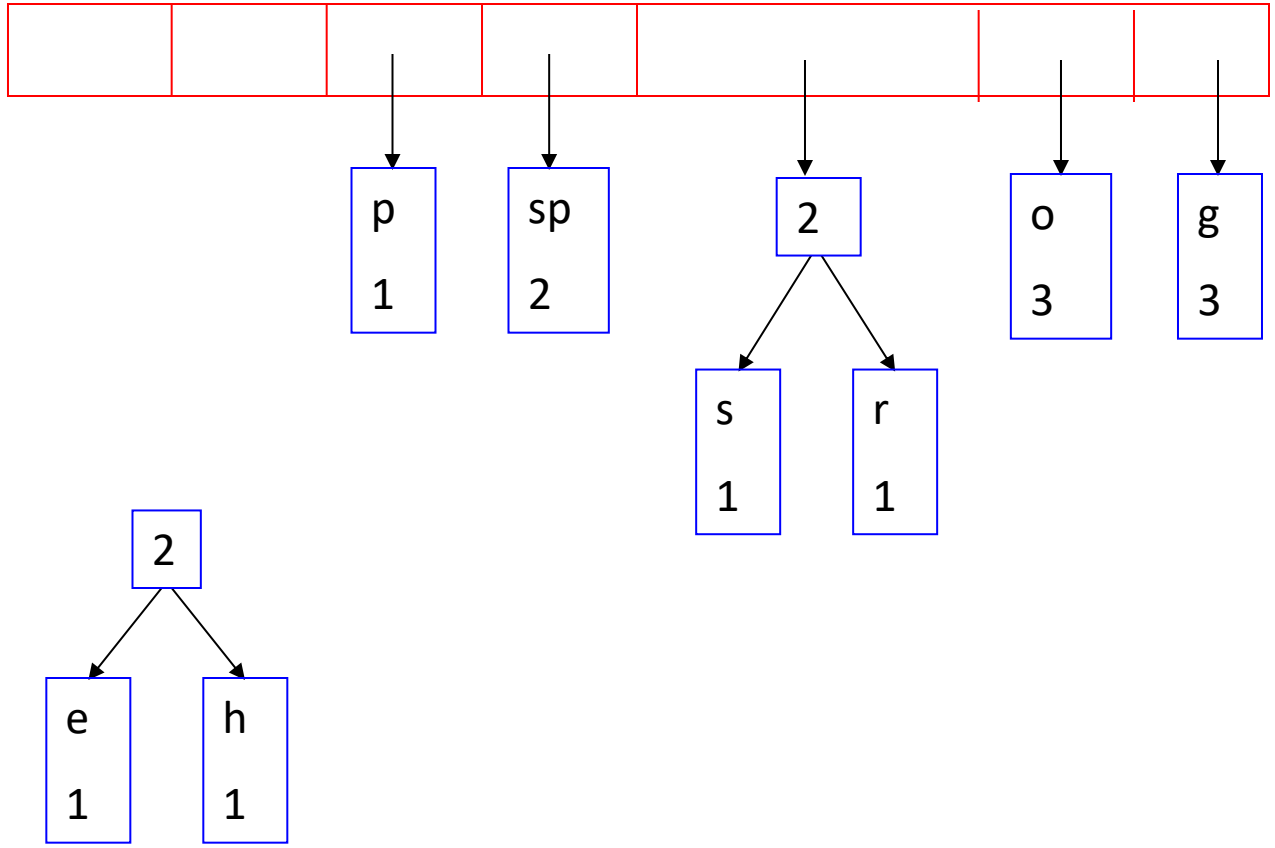
Building the Coding Tree



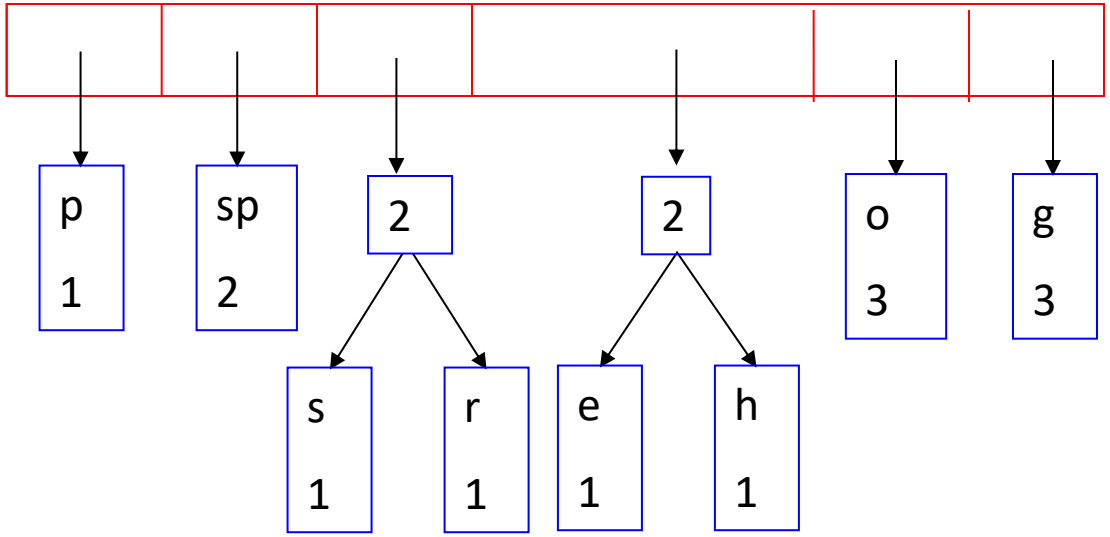
Building the Coding Tree



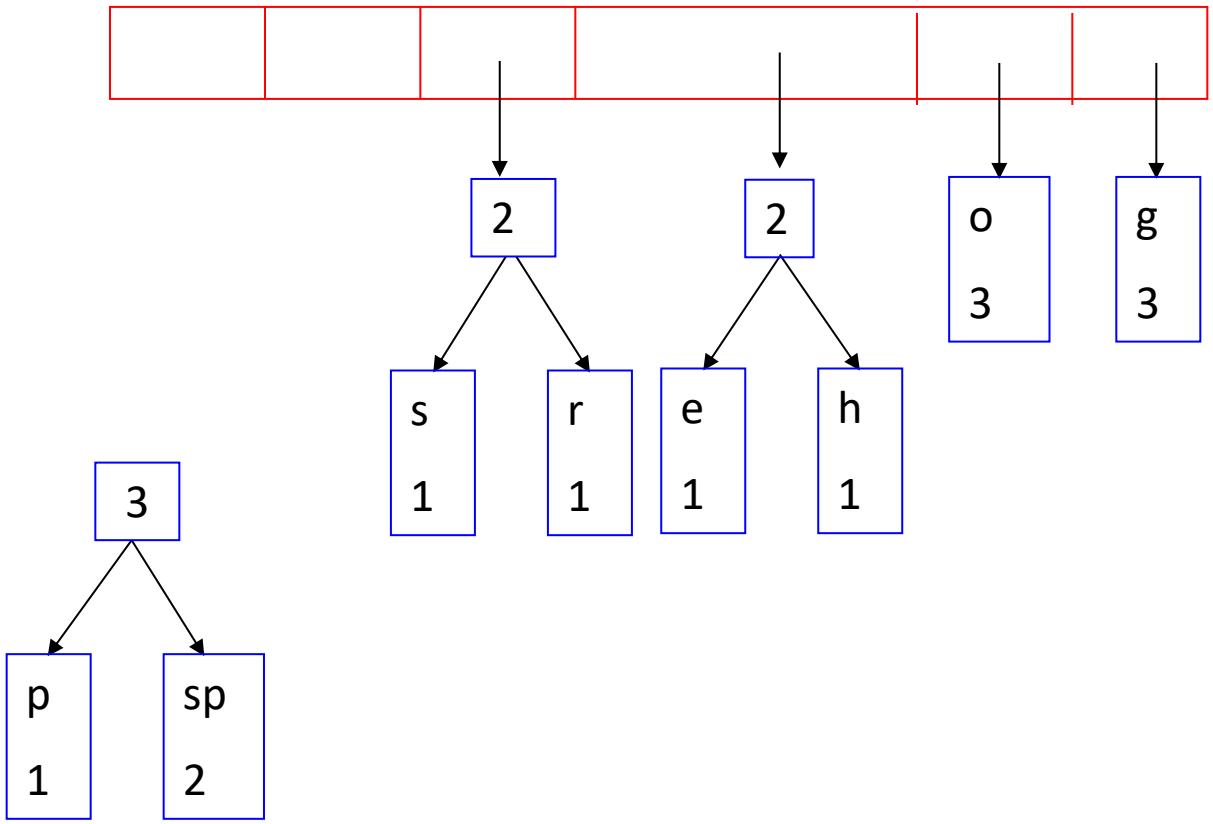
Building the Coding Tree



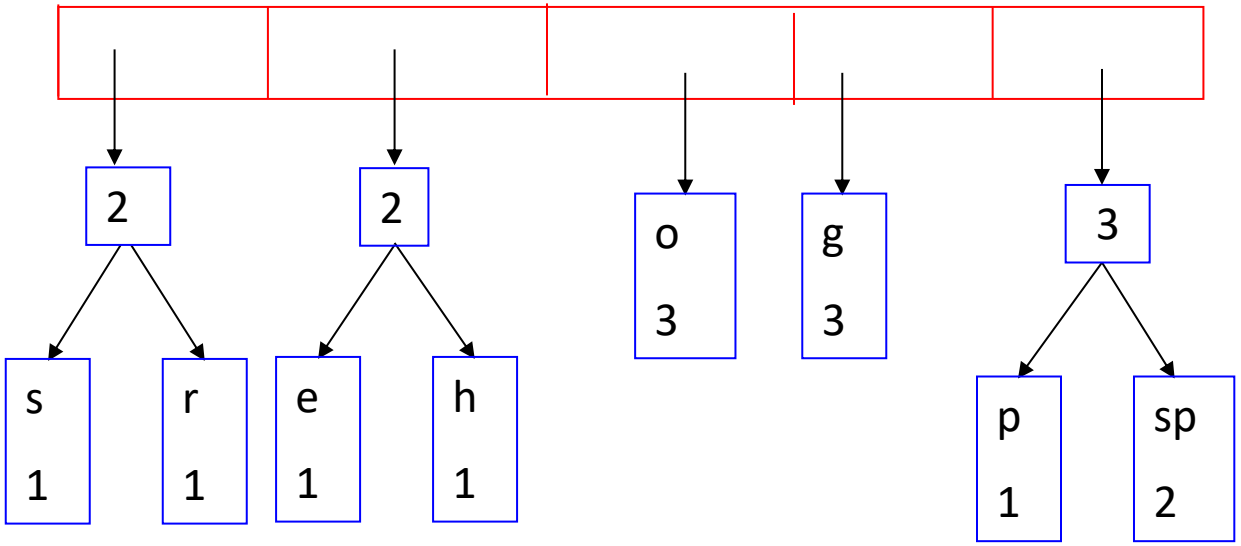
Building the Coding Tree



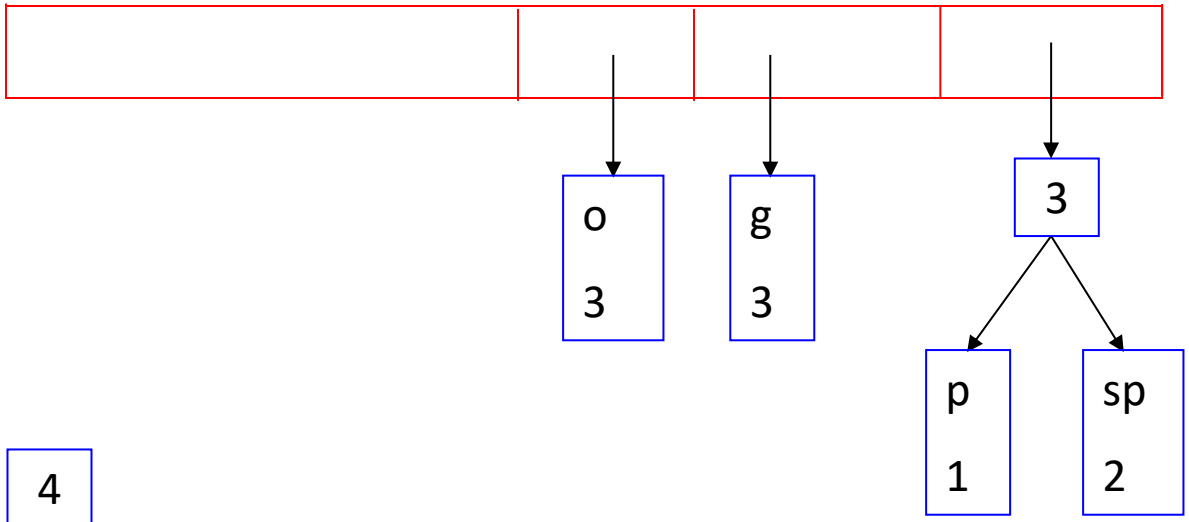
Building the Coding Tree

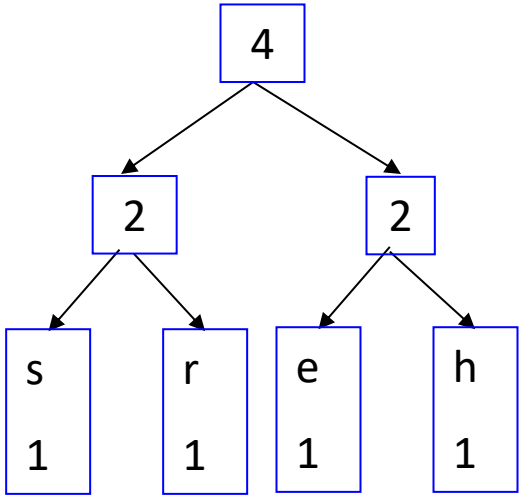


Building the Coding Tree

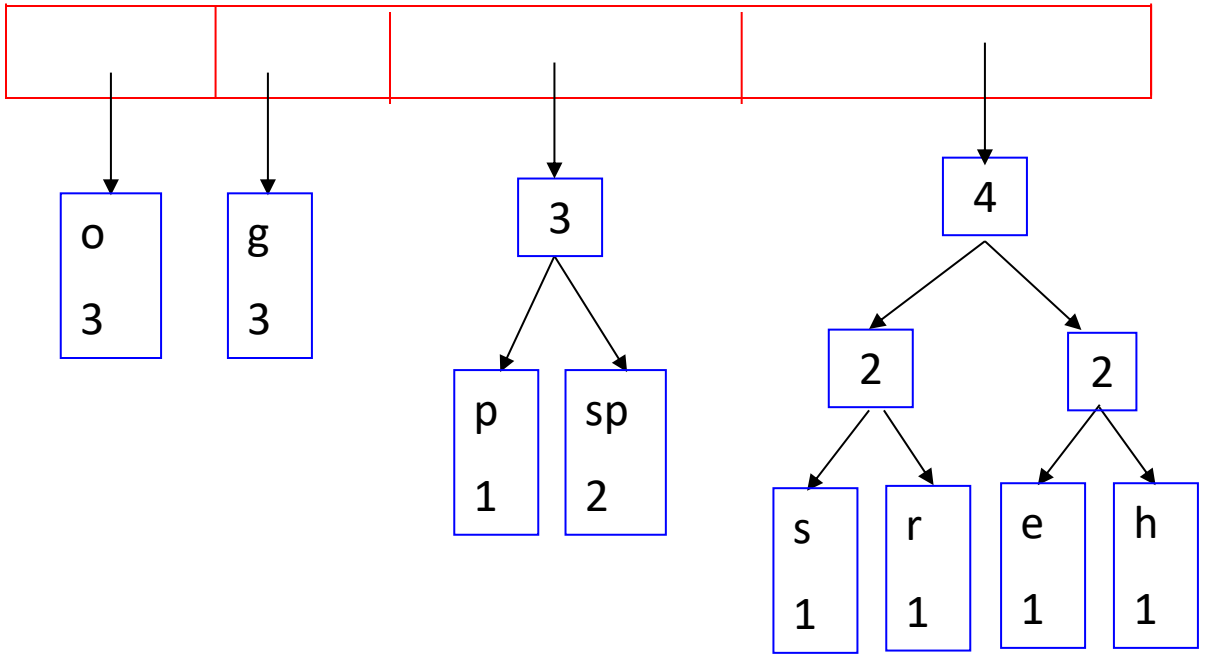


Building the Coding Tree

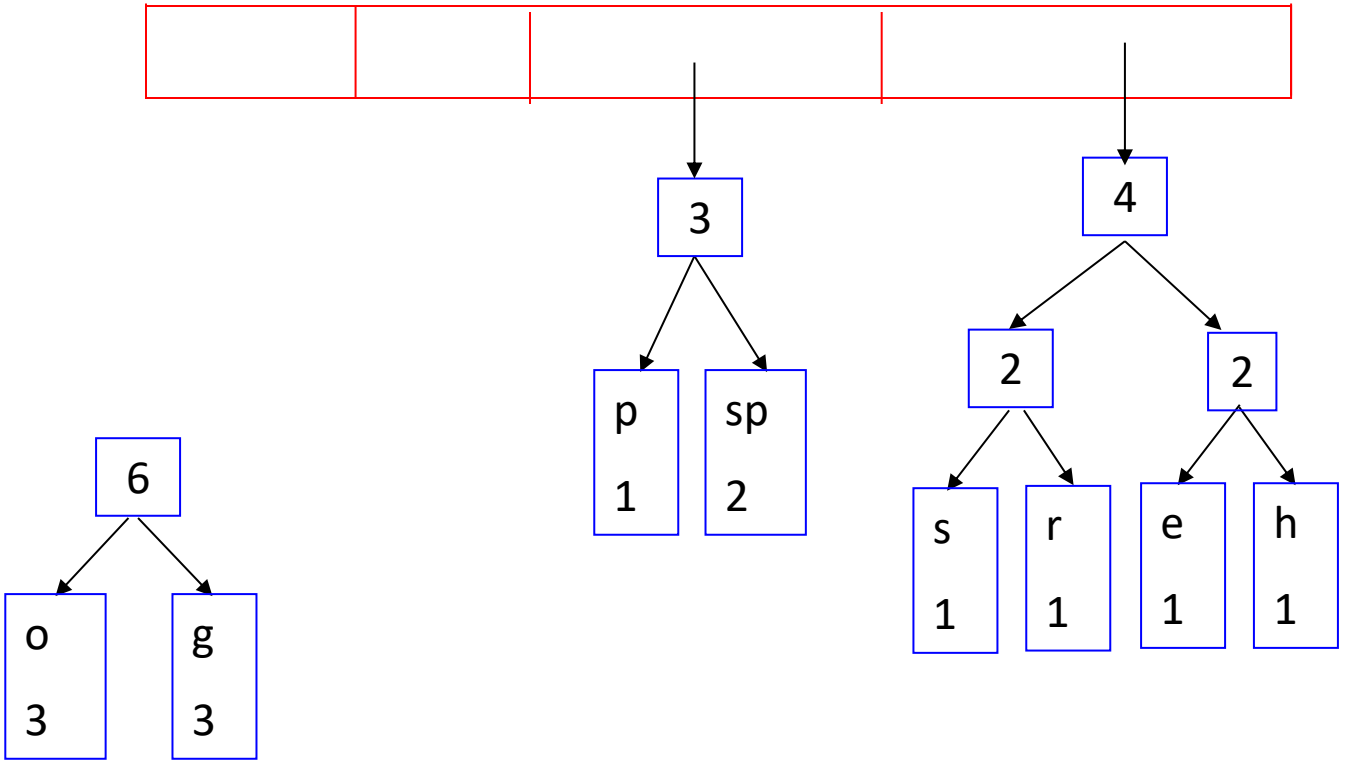




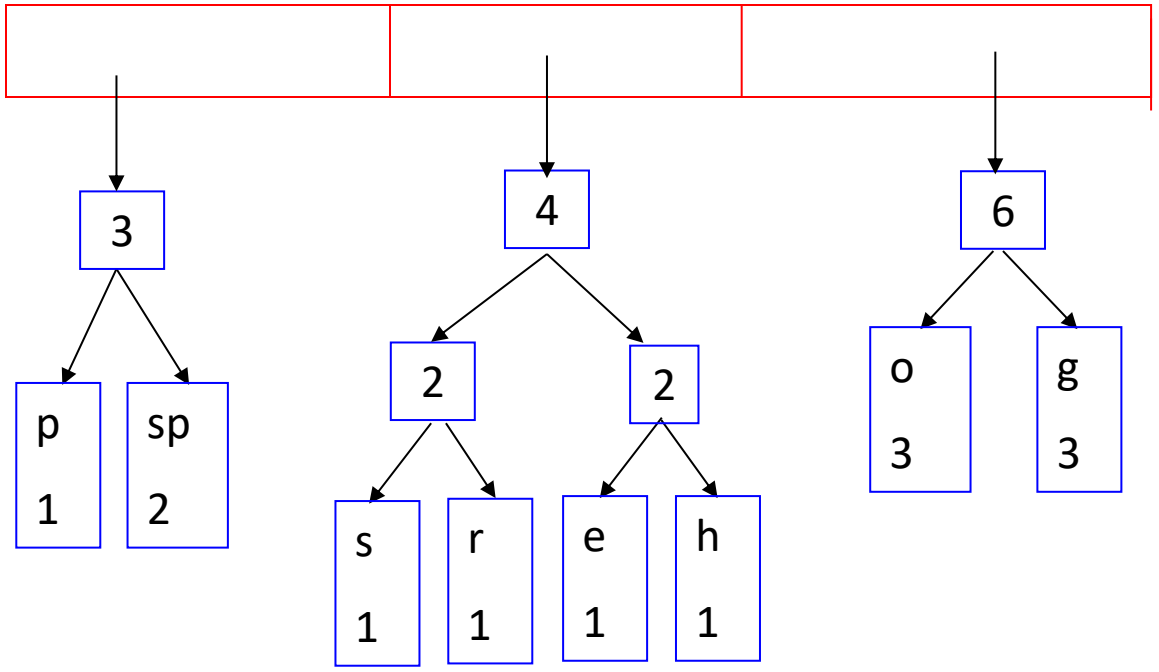
Building the Coding Tree



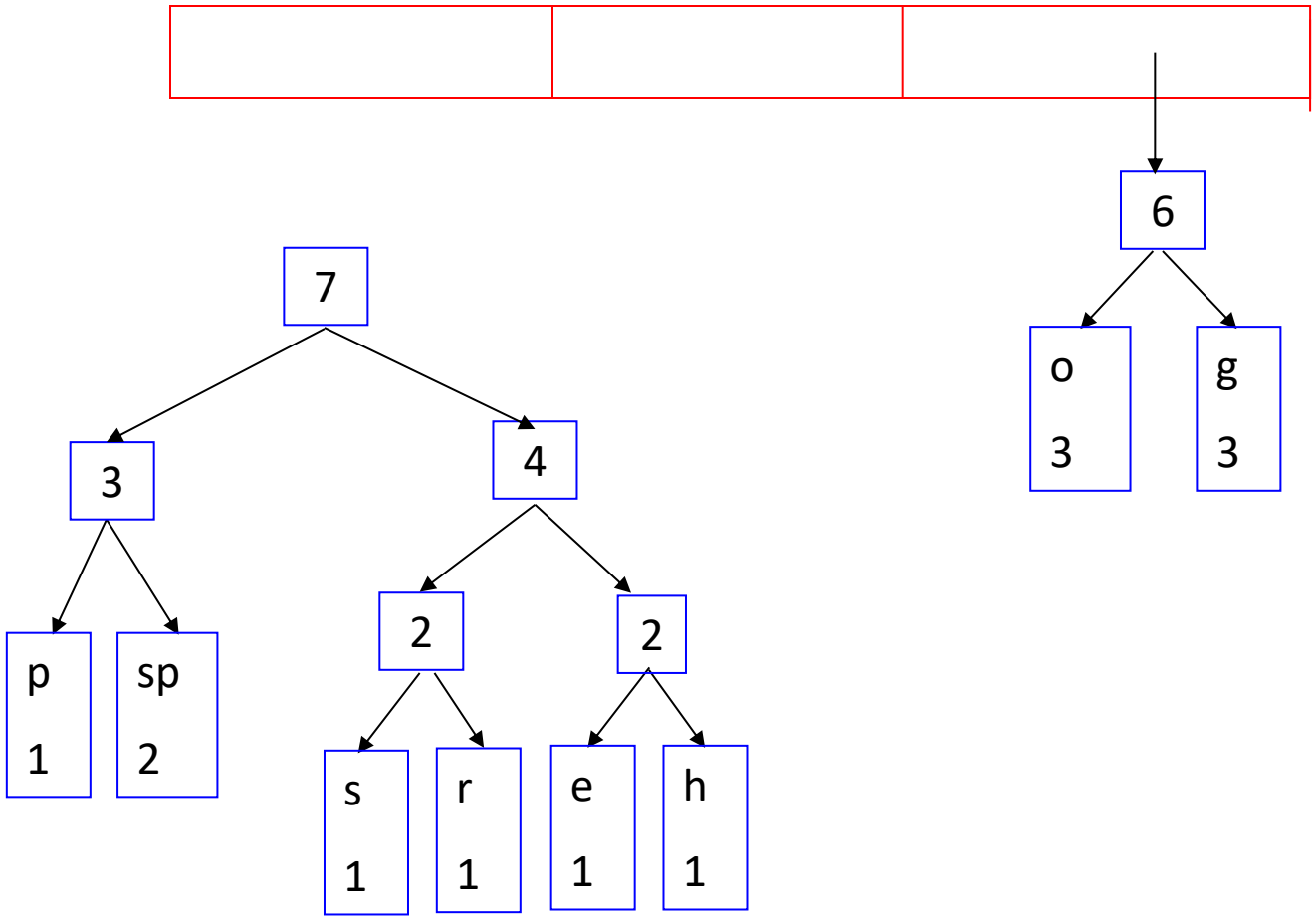
Building the Coding Tree



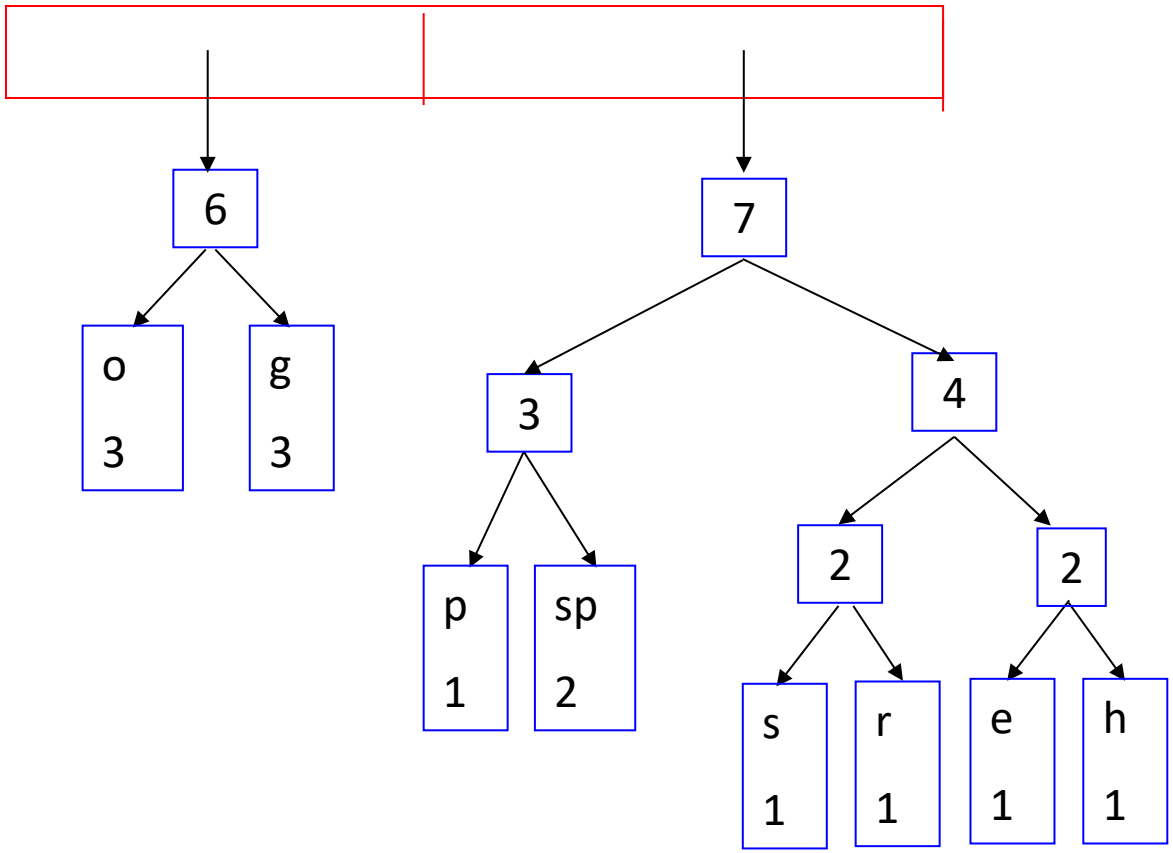
Building the Coding Tree



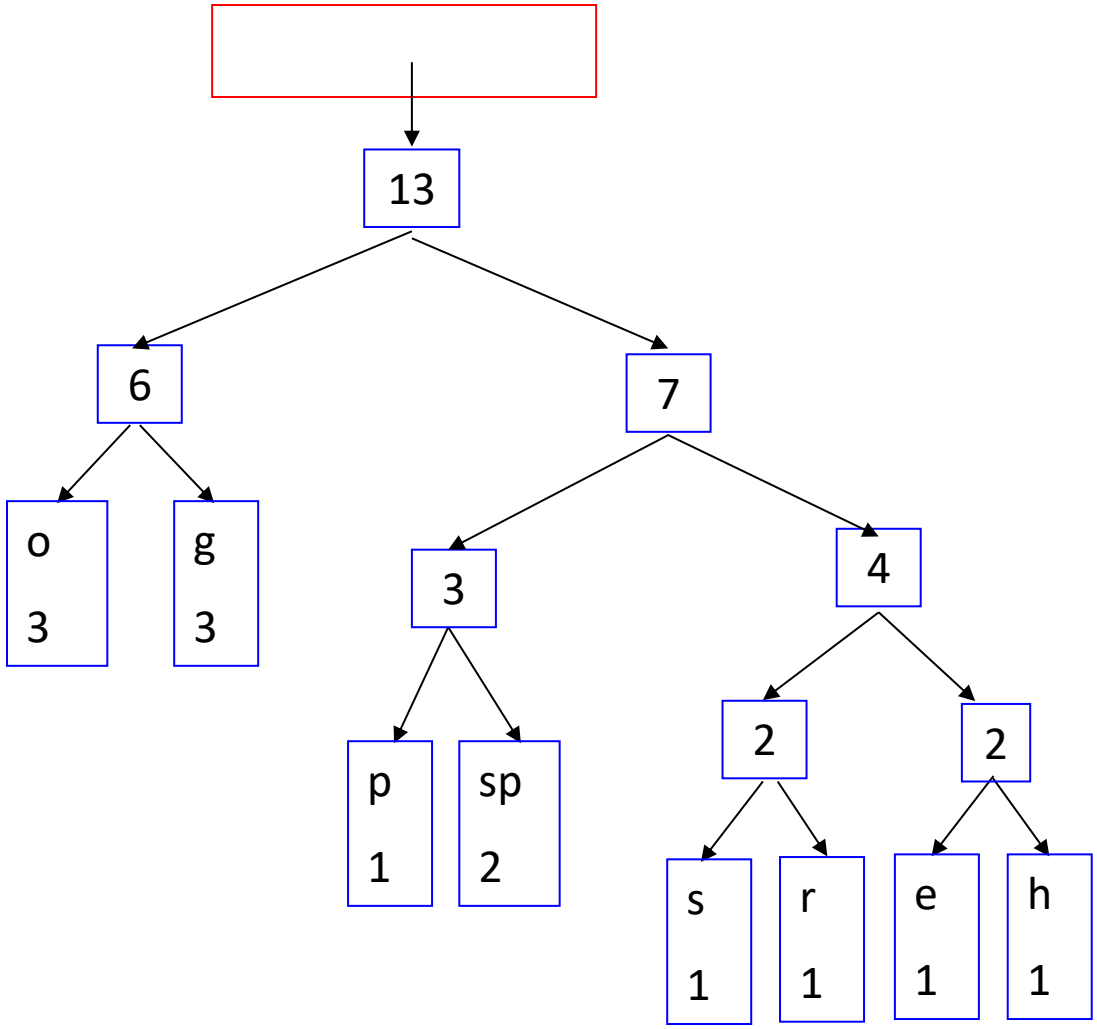
Building the Coding Tree



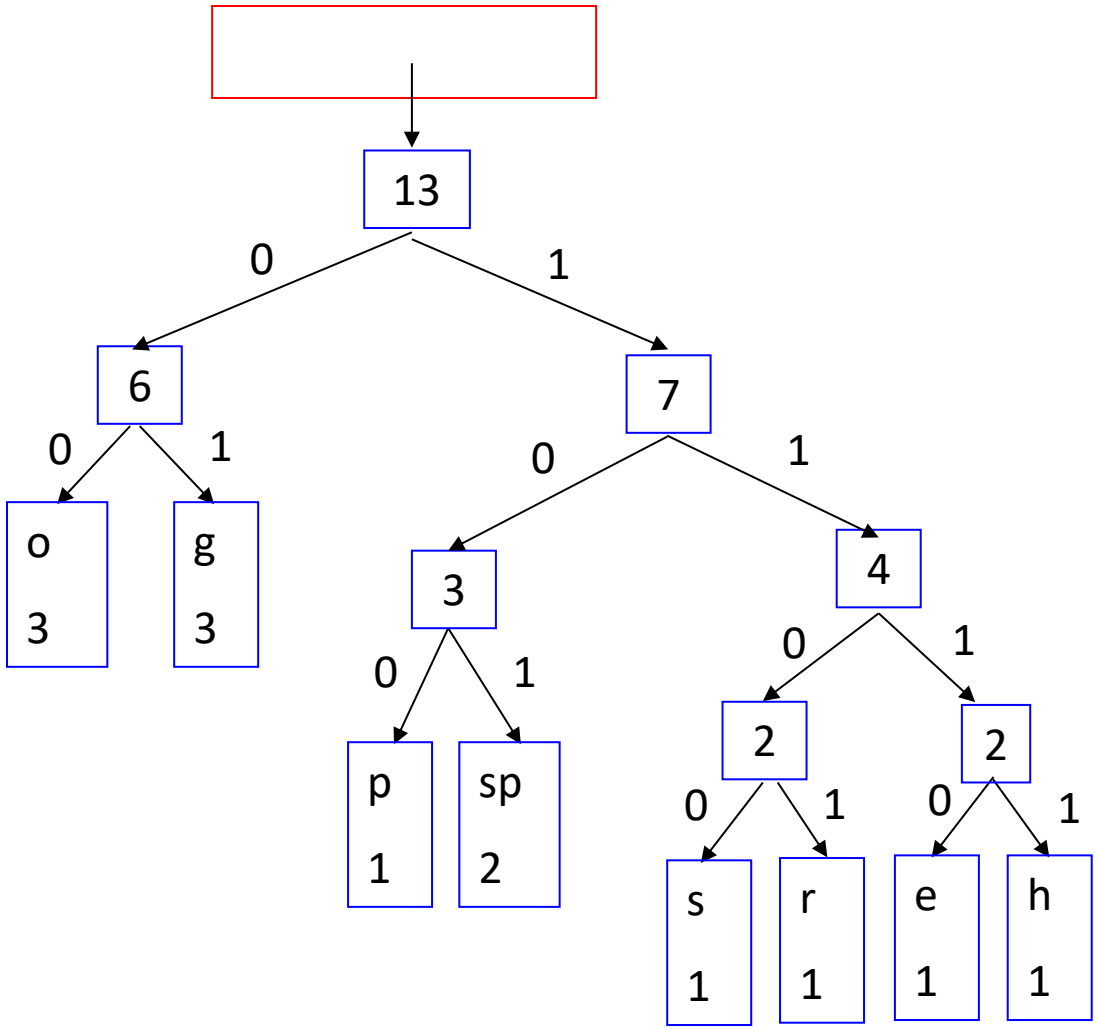
Building the Coding Tree



Building the Coding Tree



Building the Coding Tree



char	binary
'g'	01
'o'	00
‘sp’	101
'p'	100
'h'	1111
'e'	1110
'r'	1101
's'	1100

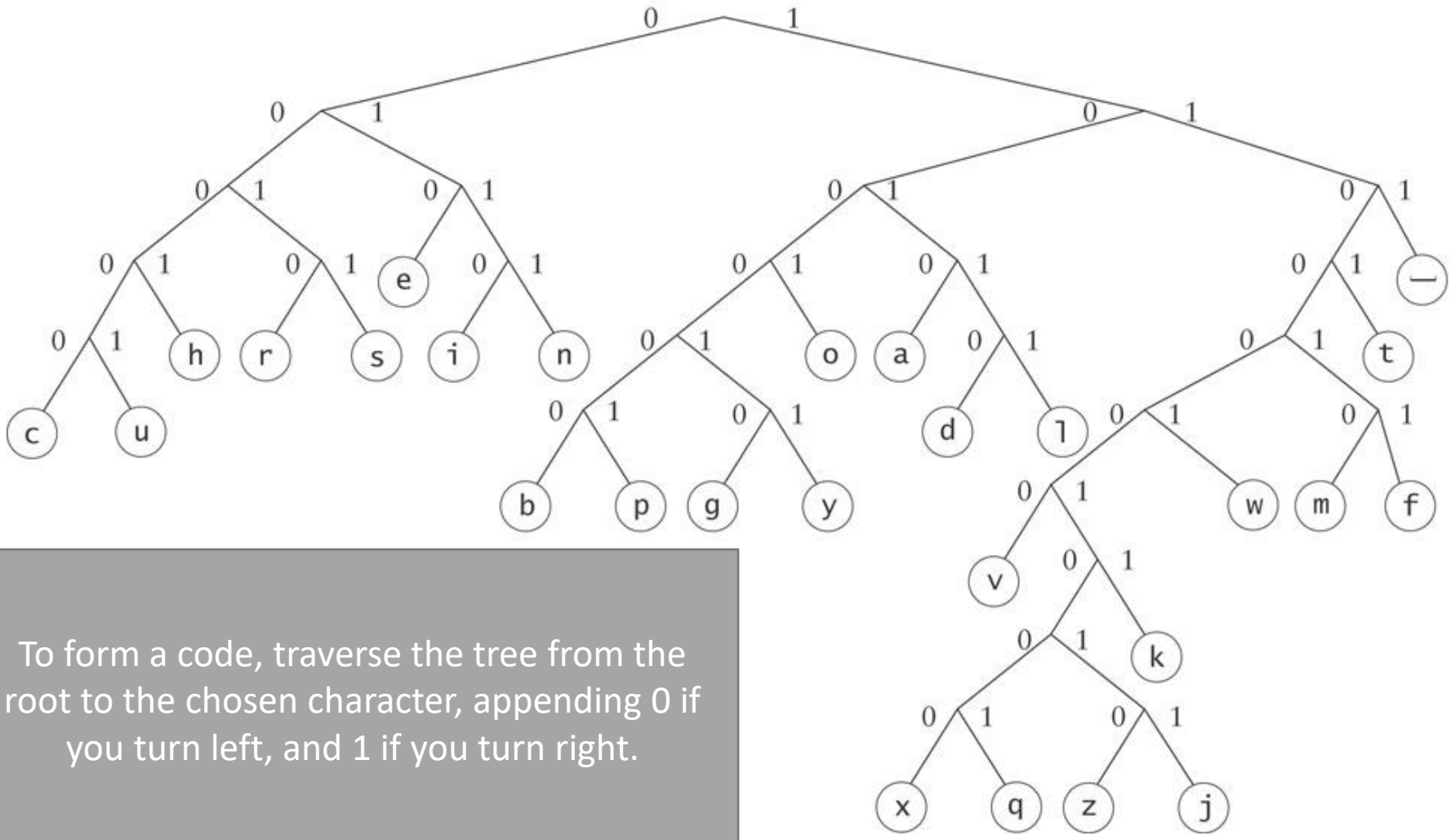
Advantages and disadvantages

- Advantages
  - Code is compressed
  - Size of message is reduced
  - Works best for text compression
  - Easy to encode and decode
- Disadvantages
  - We have to send the tree or table for decoding
  - Different coding scheme for different text
  - More difficult to decode if tree or table is not available

Solution

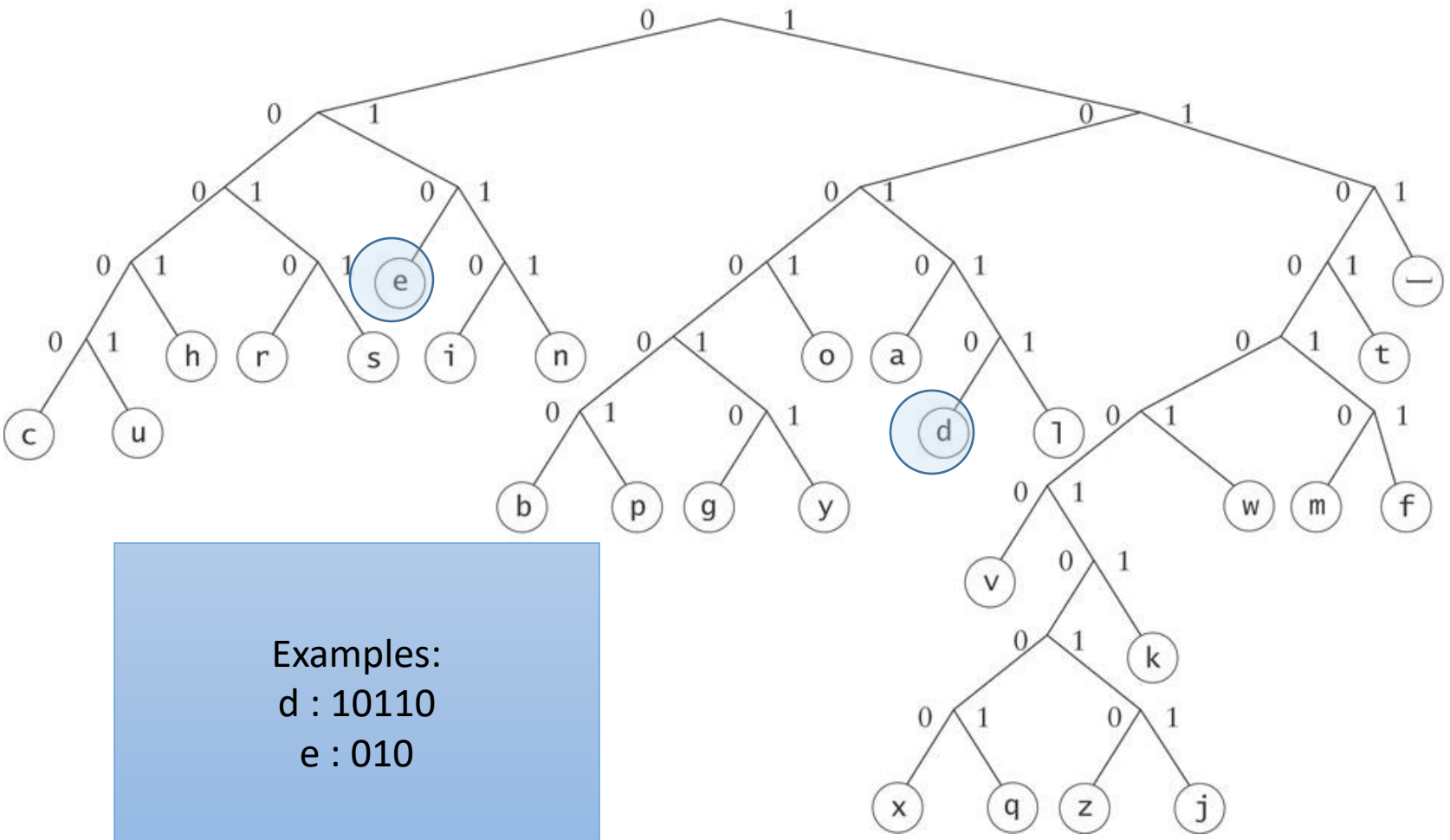
- A common encoding decoding tree for all text compressions
- This tree will be standard and will be used for all type of text compressions.
- This tree will remove the necessity of sending the encoding tree with the text
- This tree will also solve the problem of different coding scheme for different text

Huffman Tree (cont.)



To form a code, traverse the tree from the root to the chosen character, appending 0 if you turn left, and 1 if you turn right.

Huffman Tree (cont.)



Examples:
d : 10110
e : 010

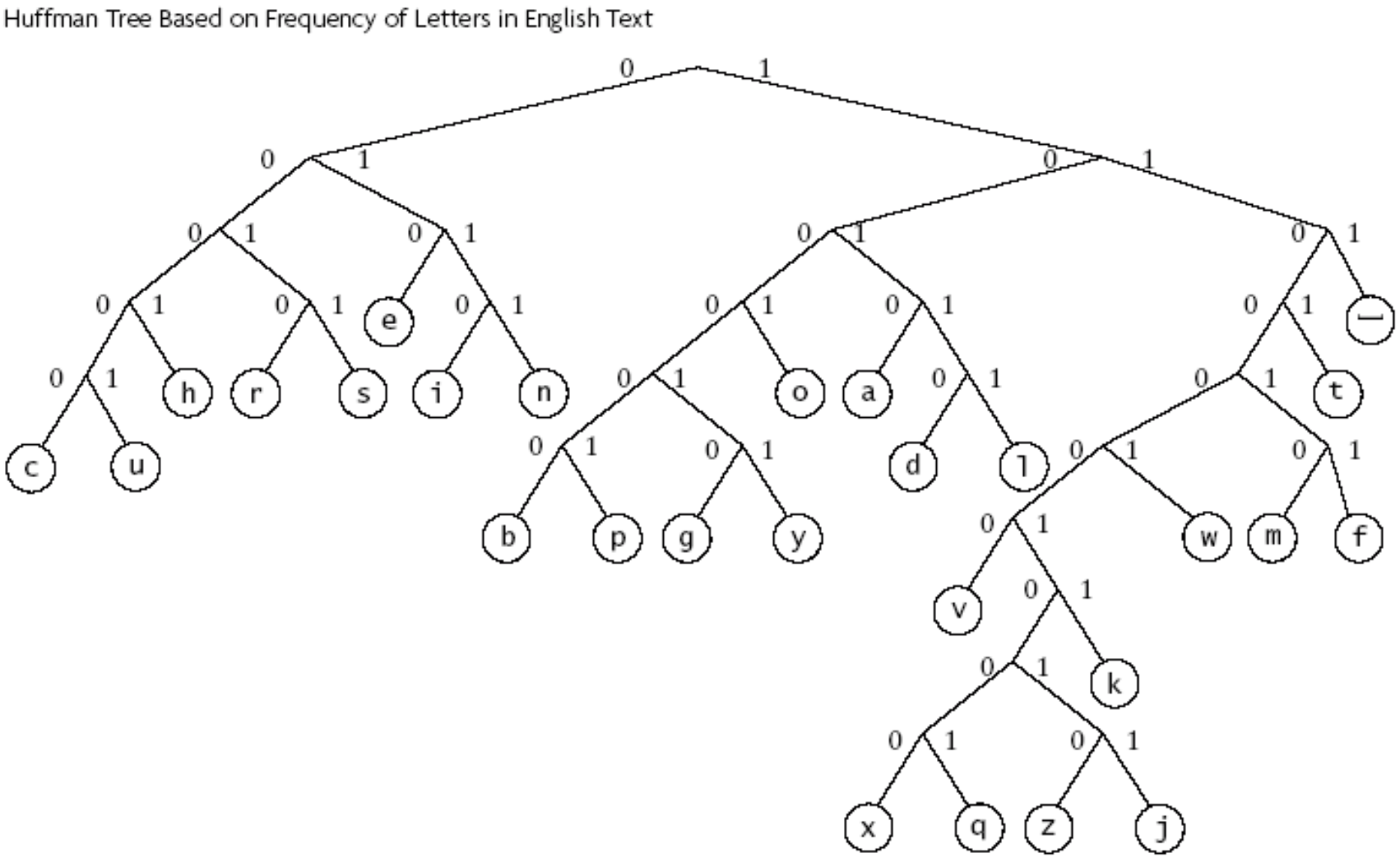
Huffman Trees

- Implemented using a binary tree and a PriorityQueue
- Unique binary number to each symbol in the alphabet
  - Unicode is an example of such a coding
- The message “go eagles” requires 144 bits in Unicode but only 38 bits using Huffman coding

Huffman Trees (cont.)

Symbol	Frequency	Symbol	Frequency	Symbol	Frequency
_	186	h	47	g	15
e	103	d	32	p	15
t	80	l	32	b	13
a	64	u	23	v	8
o	63	c	22	k	5
i	57	f	21	j	1
n	57	m	20	q	1
s	51	w	18	x	1
r	48	y	16	z	1

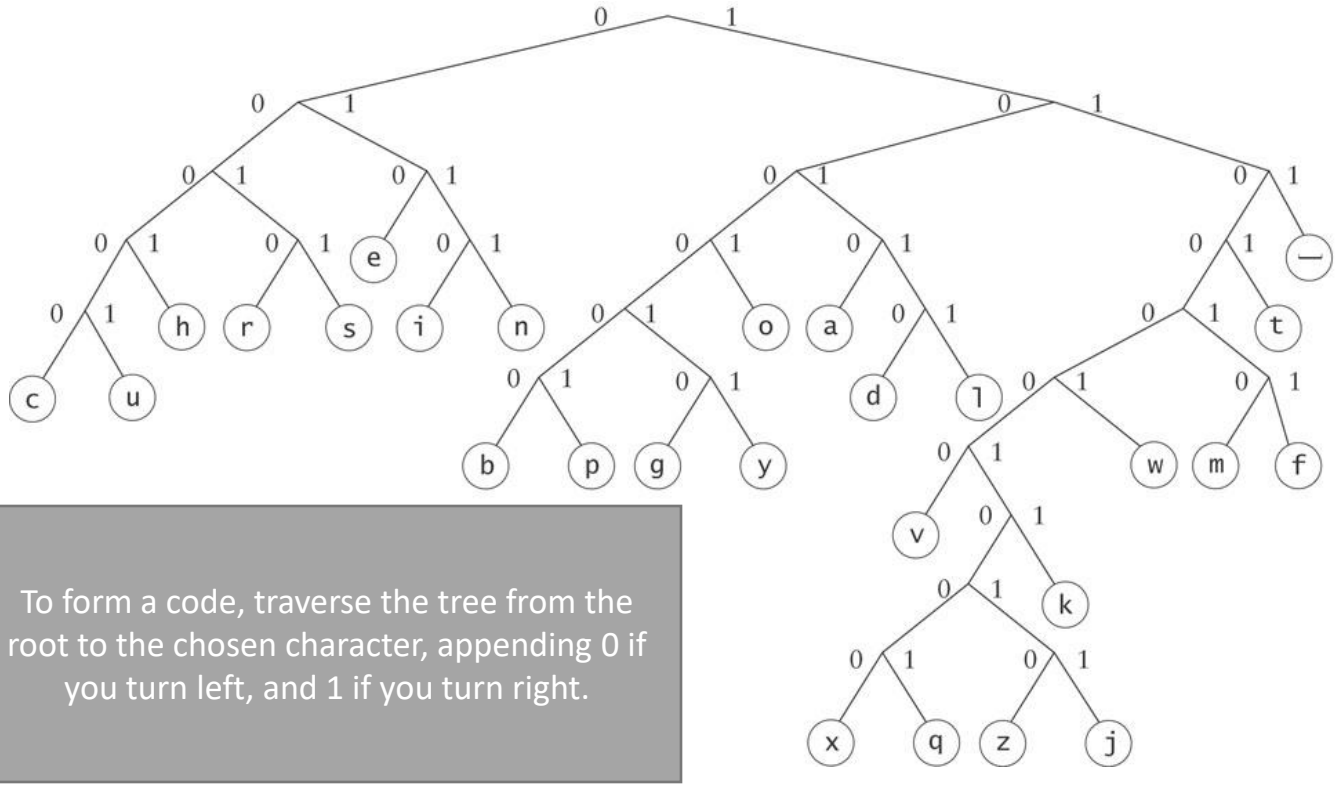
Huffman Trees (cont.)



Huffman Tree

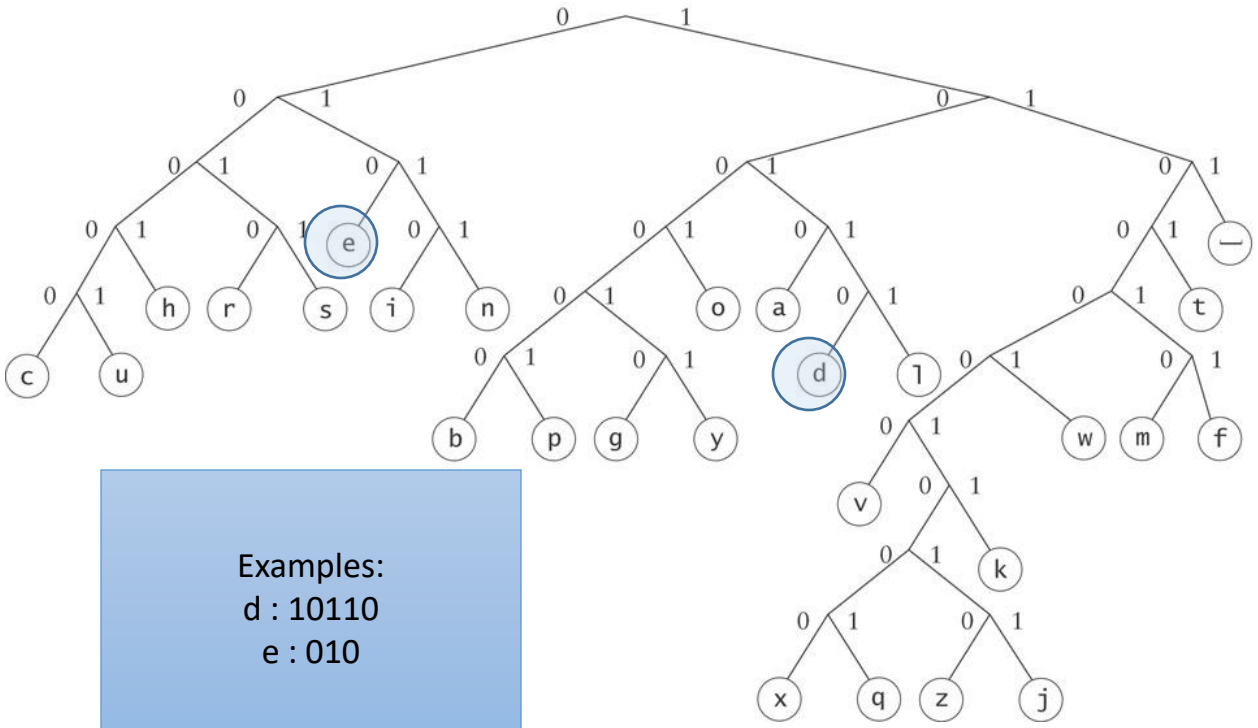
- A *Huffman tree* represents *Huffman codes* for characters that might appear in a text file
- As opposed to ASCII or Unicode, Huffman code uses different numbers of bits to encode letters
- More common characters use fewer bits
- Many programs that compress files use Huffman codes

Huffman Tree (cont.)



To form a code, traverse the tree from the root to the chosen character, appending 0 if you turn left, and 1 if you turn right.

Huffman Tree (cont.)



Examples:
d : 10110
e : 010

Huffman Trees

- Implemented using a binary tree and a PriorityQueue
- Unique binary number to each symbol in the alphabet
  - Unicode is an example of such a coding
- The message “go eagles” requires 144 bits in Unicode but only 38 bits using Huffman coding

Huffman Trees (cont.)

Symbol	Frequency	Symbol	Frequency	Symbol	Frequency
_	186	h	47	g	15
e	103	d	32	p	15
t	80	l	32	b	13
a	64	u	23	v	8
o	63	c	22	k	5
i	57	f	21	j	1
n	57	m	20	q	1
s	51	w	18	x	1
r	48	y	16	z	1

Huffman Trees (cont.)