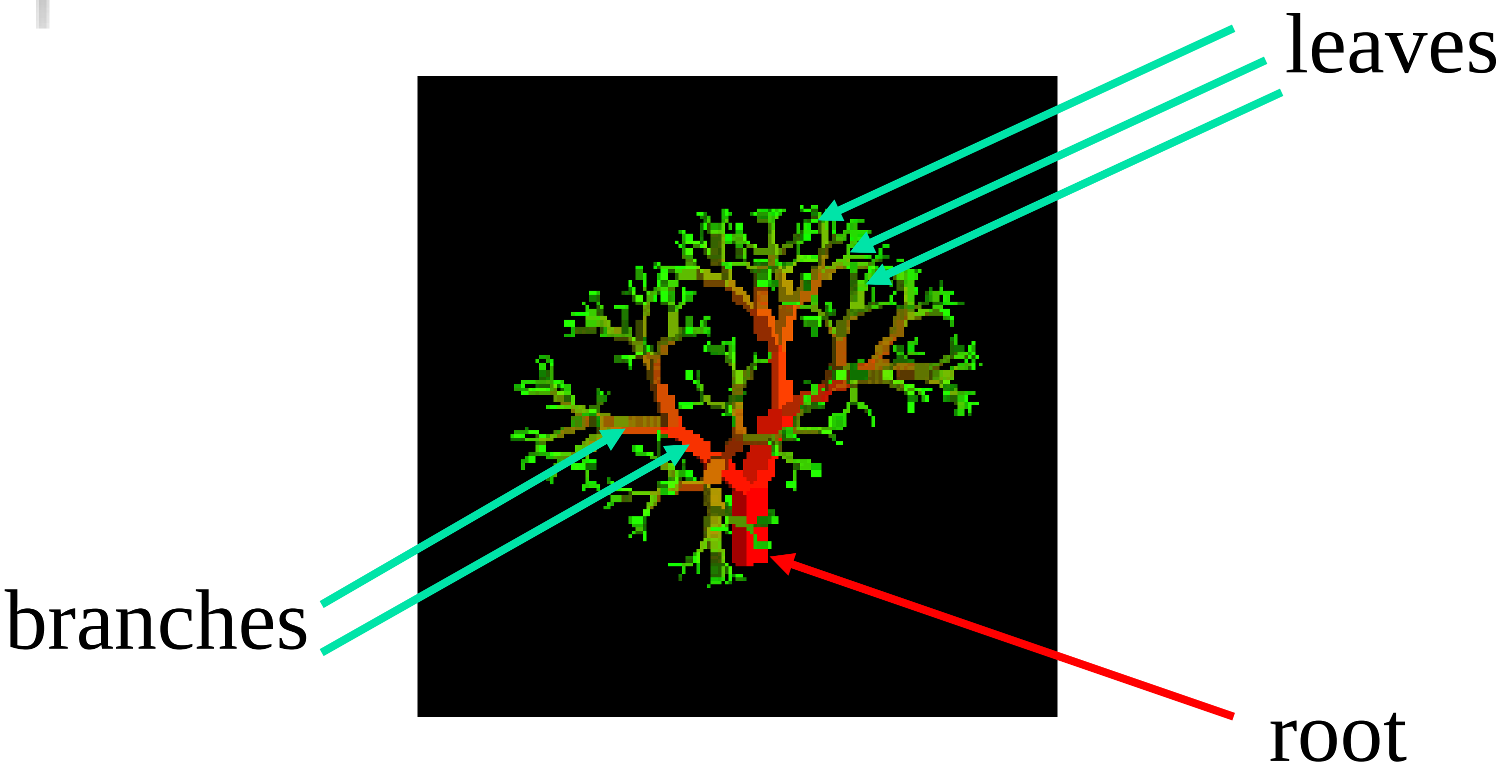
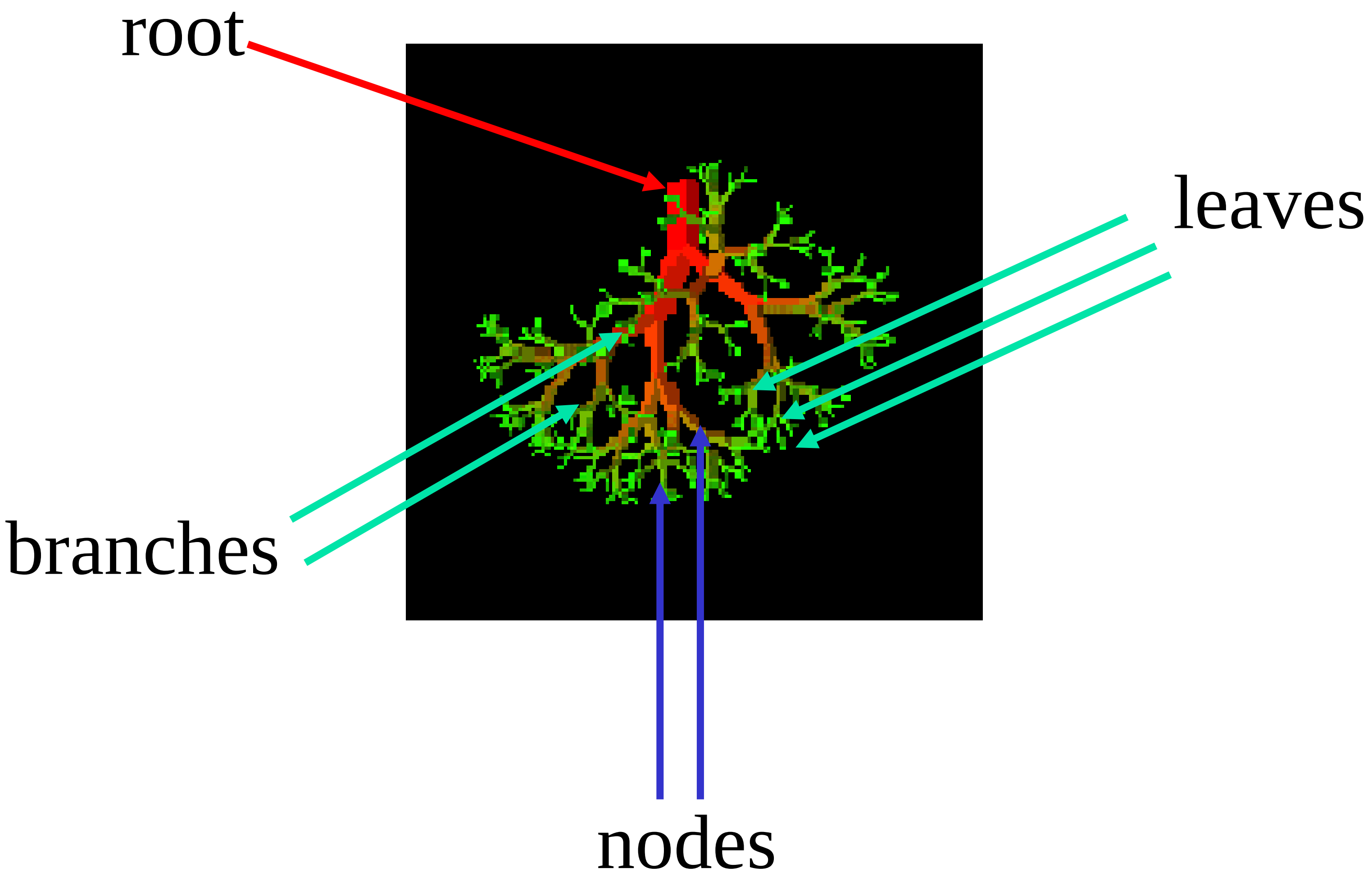
Trees

Abu Said Md. Rezoun
Lecturer, Dept. of CSE
Sonargaon University
E-mail: abusaid.rezoun@gmail.com
Contact: 01788210077

Nature View of a Tree



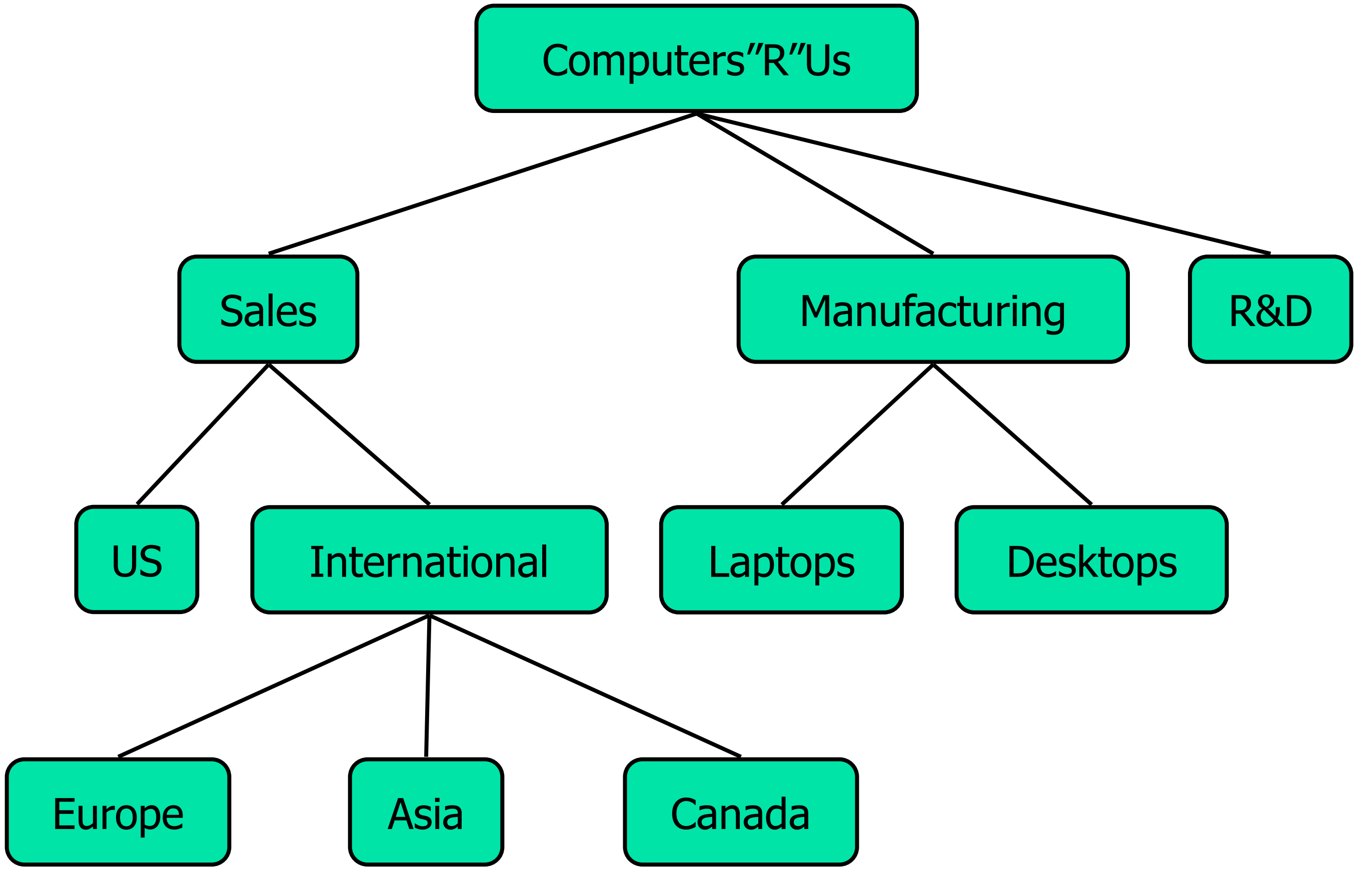
Nature View of a Tree



Trees

- Tree is a nonlinear data structure
- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- Consists of nodes with a parent-child relation.

Trees



Computers"R"Us
Sales
Manufacturing
R&D
US
International
Laptops
Desktops
Europe
Asia
Canada

Characteristics of Trees

- Non-linear data structure
- Combines advantages of an ordered array
- Searching as fast as in ordered array
- Insertion and deletion as fast as in linked list
- Simple and fast

Application of Trees

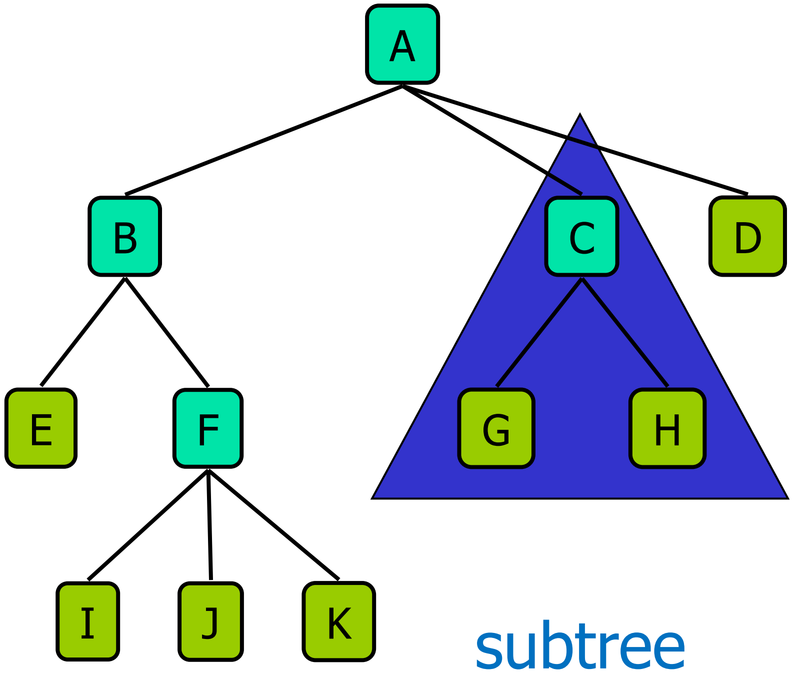
- Directory structure of a file store
- Structure of an arithmetic expressions
- Used in almost every 3D video game to determine what objects need to be rendered.
- Used in almost every high-bandwidth router for storing router-tables.
- Used in compression algorithms, such as those used by the .jpeg and .mp3 file formats.

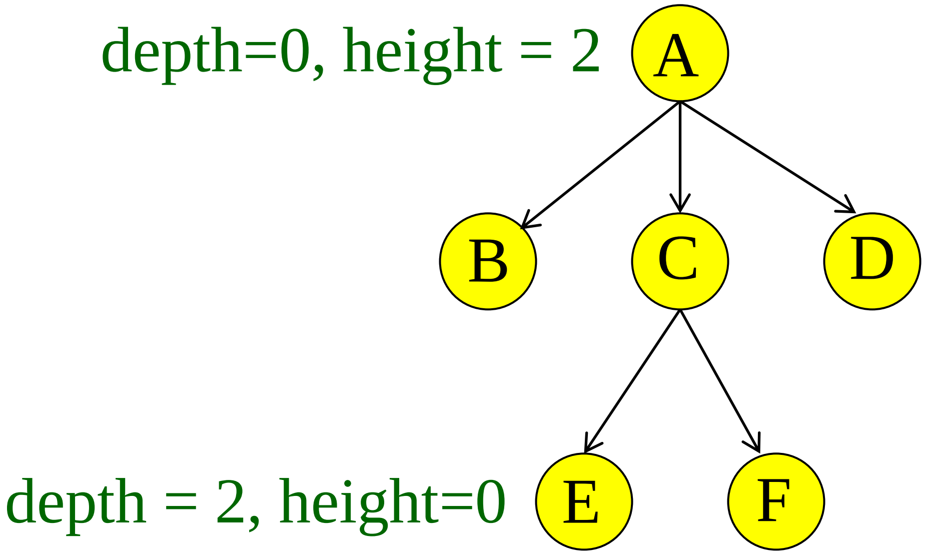
Tree Terminology

- **Root:** node without parent (A)
- **Siblings(brothers):** nodes share the same parent
- **Internal node:** node with at least one child (A, B, C, F)
- **External node (leaf / terminal node):** node without children (E, I, J, K, G, H, D)
- **Ancestors of a node:** parent, grandparent, grand-grandparent, etc.
- **Descendant of a node:** child, grandchild, grand-grandchild, etc.
- **Depth of a node:** number of ancestors

Tree Terminology

- **Height of a tree:** maximum depth of any node (3)
- **Degree of a node:** the number of its children
- **Degree of a tree:** the maximum number of its node. Ex: 3
- **Subtree**: tree consisting of a node and its descendants





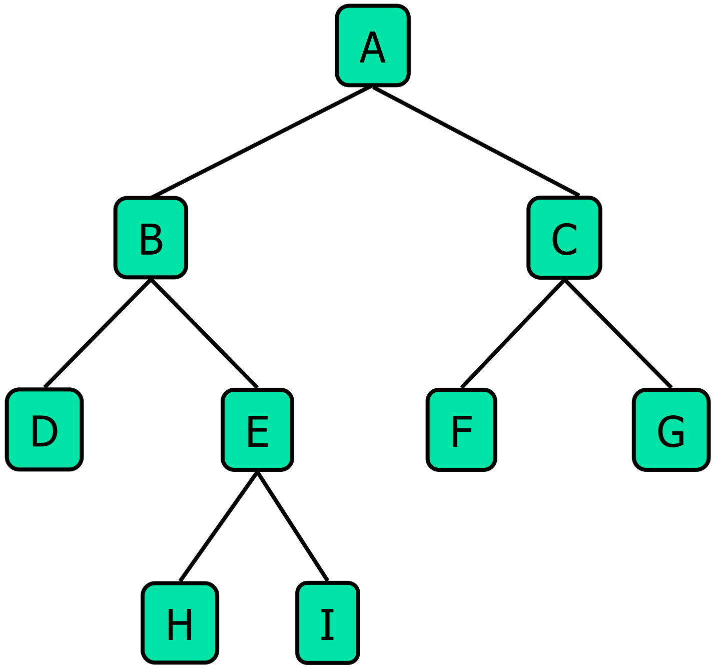
Binary Trees

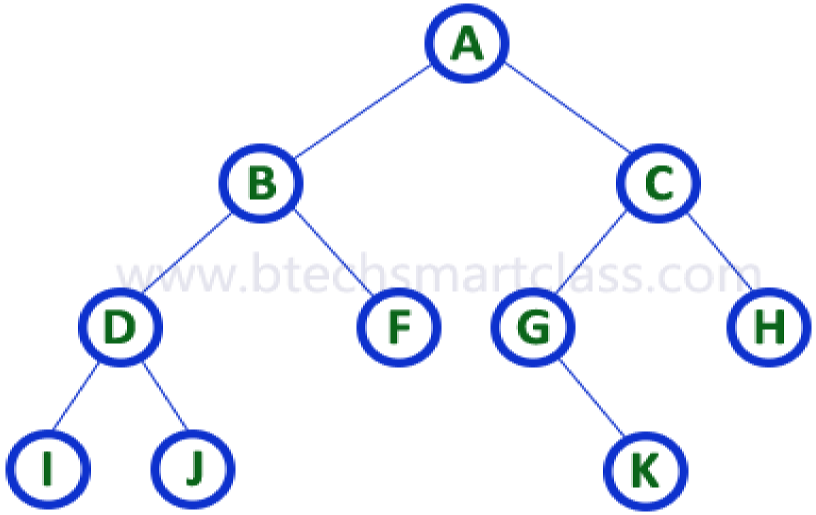
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (degree of two)
  - The children of a node are an ordered pair
- We call the children of an internal node left child (left successor) and right child (right successor)
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, OR
  - a tree whose root has an ordered pair of children, each of which is a binary tree

Binary Trees

- A binary tree T is defined as a finite set of elements, called node, such that –
  - T is empty(called the null tree or empty tree) or
  - T contains a distinguished node R, called the root of T, and the remaining nodes of T from an ordered pair of disjoint binary tree T_1 and T_2
- Applications
  - arithmetic expressions
  - decision processes
  - searching

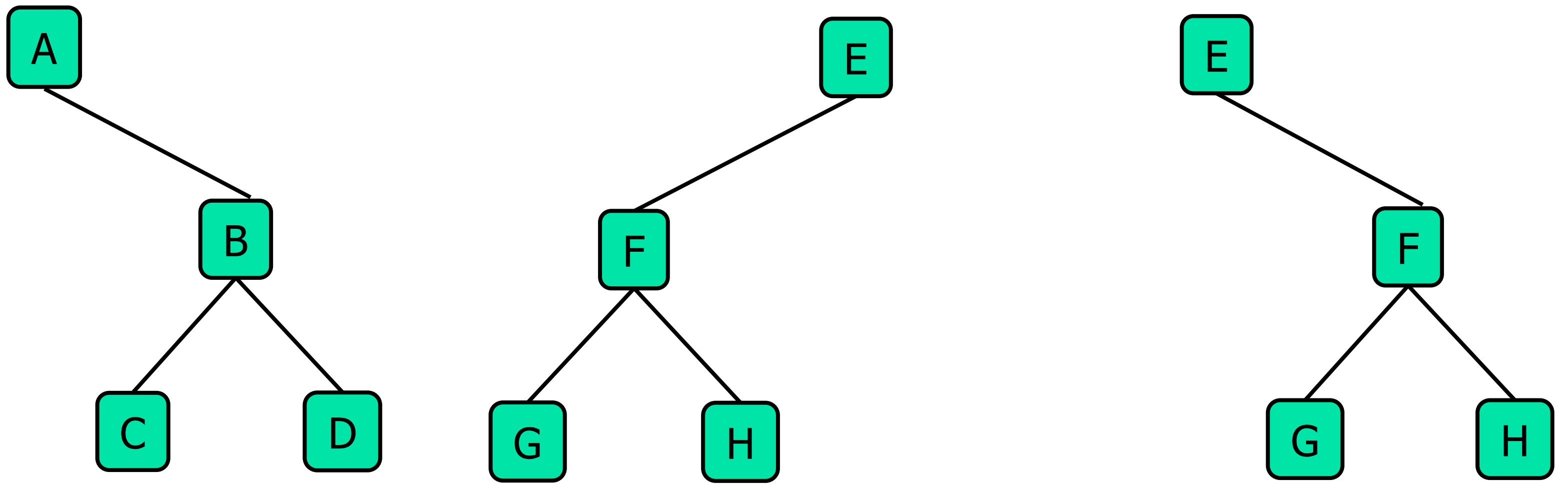
Binary Trees





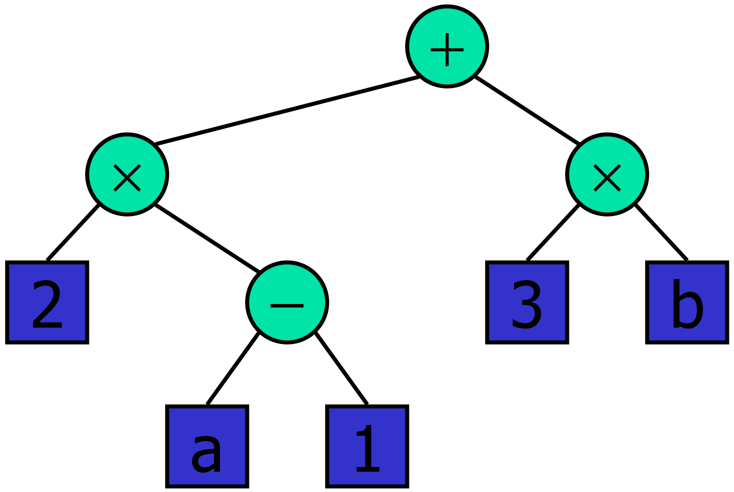
Binary Trees

- Binary trees T and T' are similar if they have the same structure or if they have same shape.
- First and third tree are similar but second tree is not similar to the rests.



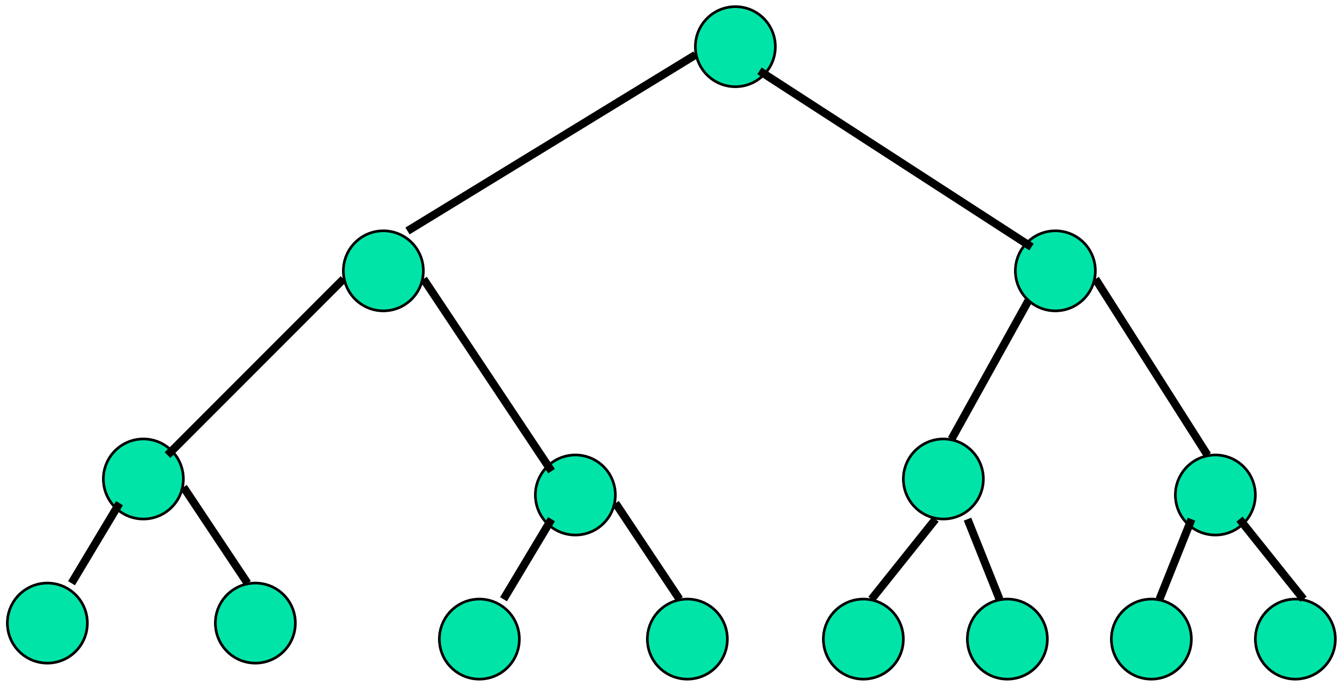
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Full Binary Tree or 2-tree or Extended Binary Tree

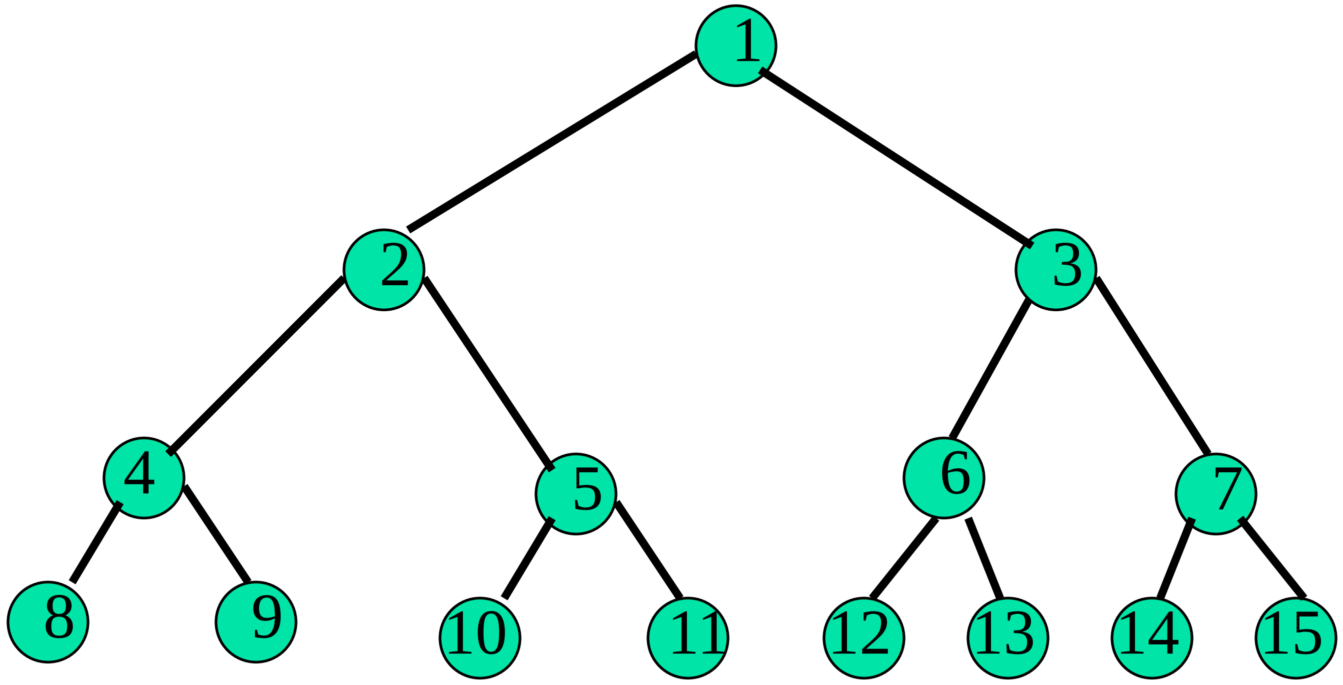
- A binary tree in which every node has either two or zero number of children is called Full Binary Tree
- A full binary tree of a given height k has $2^{k+1}–1$ nodes.



Height 3 full binary tree.

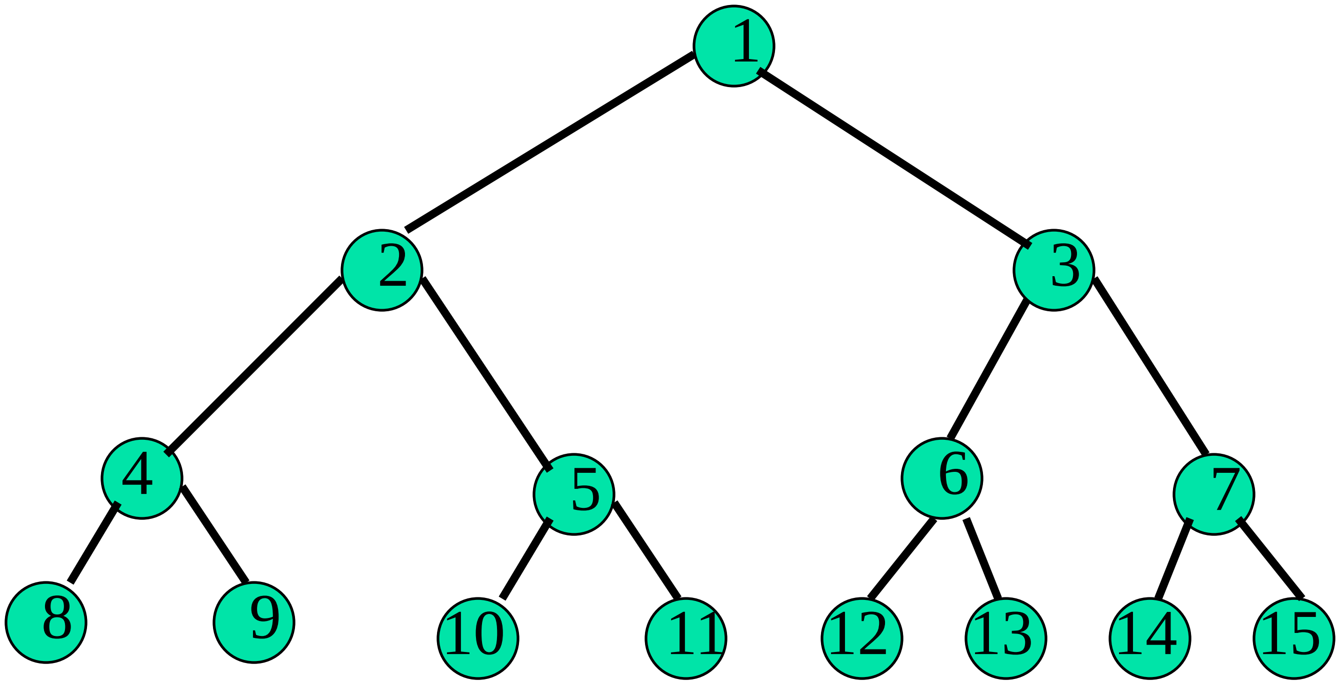
Labeling Nodes In A Full Binary Tree

- Label the nodes 1 through $2^{k+1} - 1$.
- Label by levels from top to bottom.
- Within a level, label from left to right.



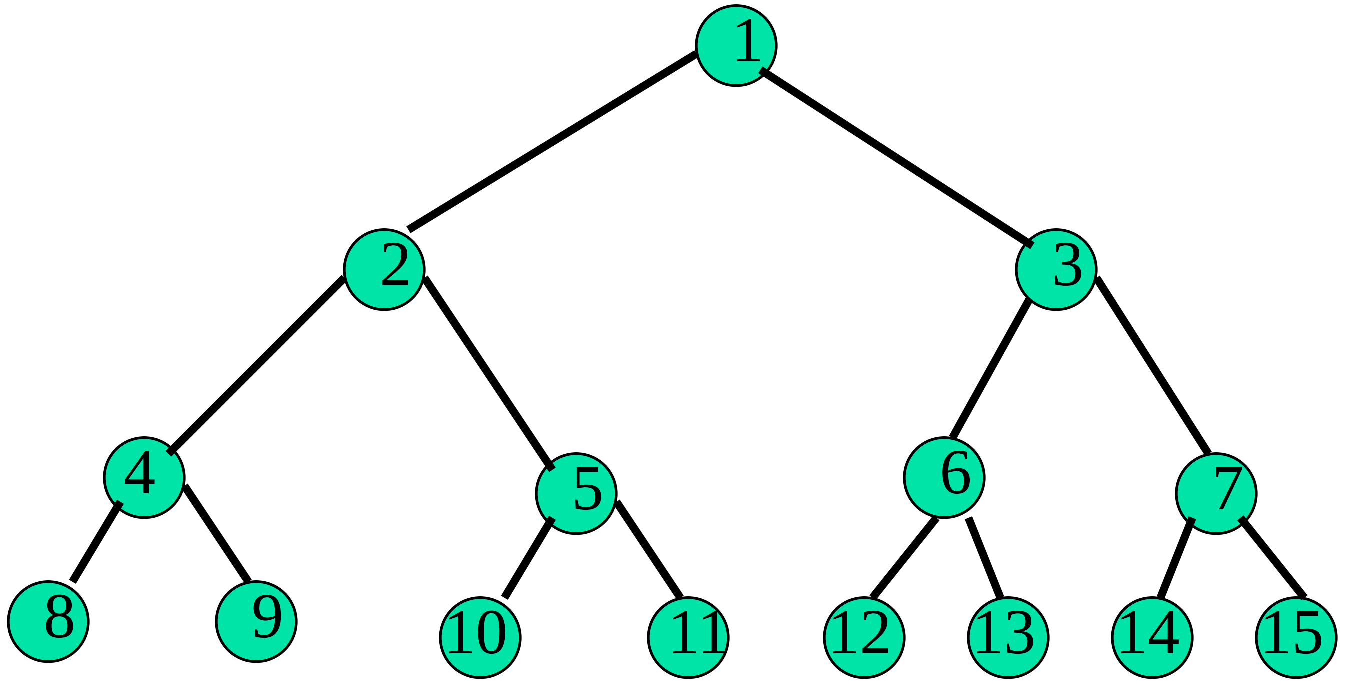
Node Number Properties

- Parent of node i is node i / 2, unless i = 1.
- Node 1 is the root and has no parent.



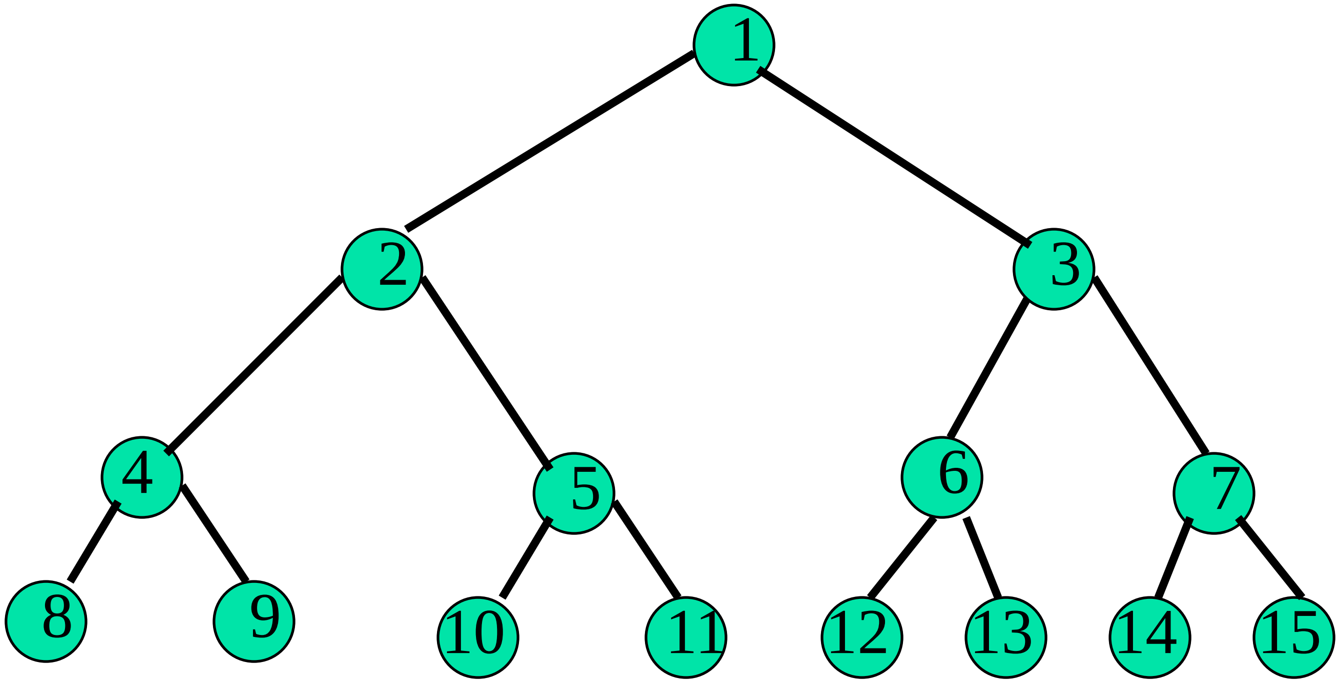
Node Number Properties

- Left child of node i is node 2i, unless 2i > n, where n is the number of nodes.
- If 2i > n, node i has no left child.



Node Number Properties

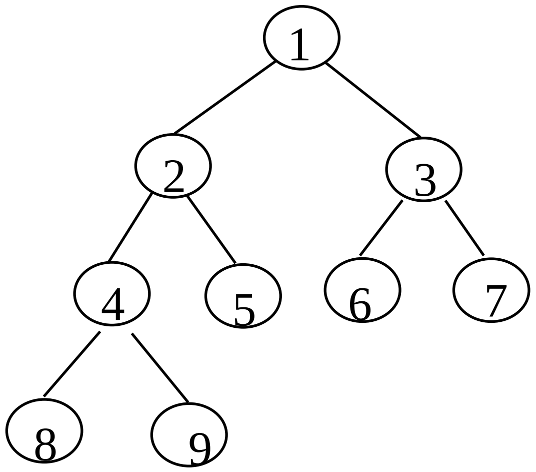
- Right child of node i is node 2i+1, unless 2i+1 > n, where n is the number of nodes.
- If 2i+1 > n, node i has no right child.



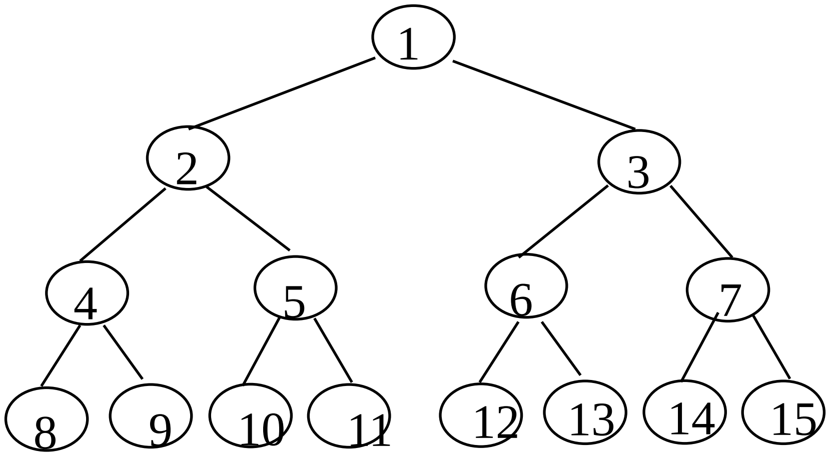
Complete Binary Trees

- A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree. Complete binary tree is also called as **Perfect Binary Tree**
- **Number of nodes = $2^{d+1} – 1$**
- **Number of leaf nodes = 2^{d}**
- Where, d – Depth of the tree

Complete Binary Trees



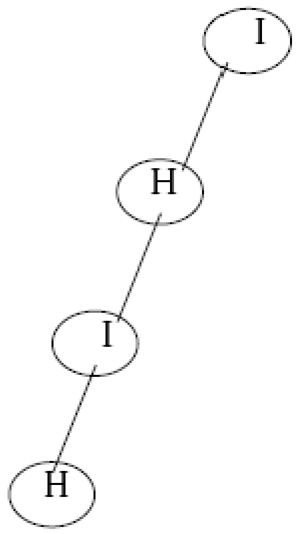
Complete binary tree

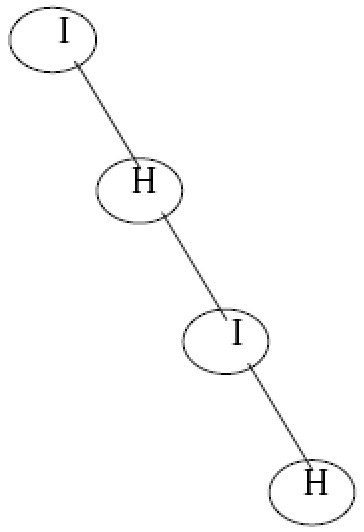


Full binary tree of depth 3

Left Skewed and Right Skewed Trees

- Binary tree has only left sub trees - Left Skewed Trees
- Binary tree has only right sub trees - Right Skewed Trees





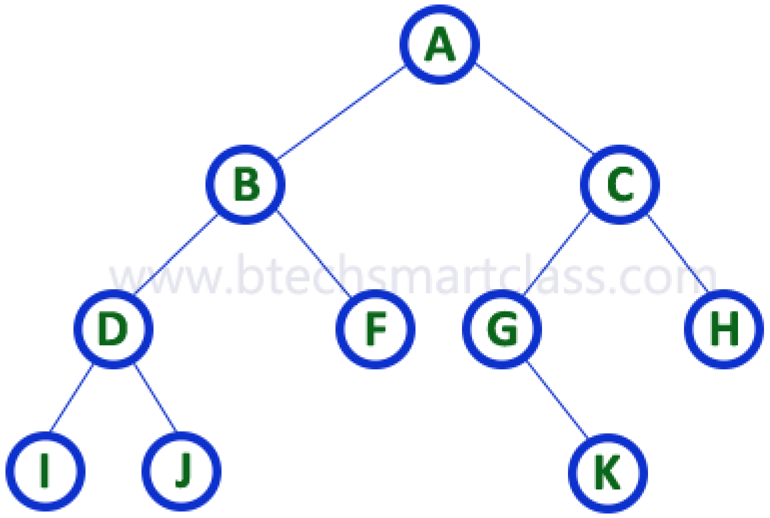
Binary Tree Representation

- Two way of representation:
  1. Sequential representation using arrays
  2. List representation using Linked list
- The main requirement of any representation of T is that one should have direct access to the root R of T and given any node N of T.

Sequential representation

- To represent a binary tree of depth **‘d'** using array representation, we need one dimensional array with a maximum size of $\mathbf{2^{d+1} - 1}$.
- If TREE is a single linear array:
  - The root R of T is stored in TREE[1]
  - If a node N occupies TREE[K], then its left child is stored in TREE[2*K] and its right child is stored in TREE[2*K + 1]

Sequential representation



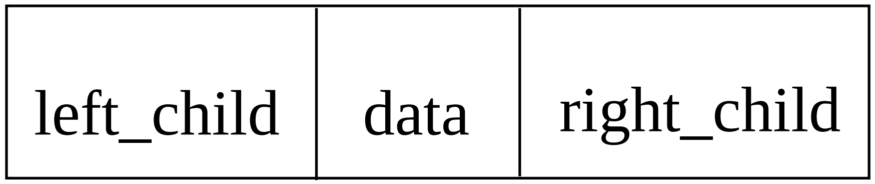
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-

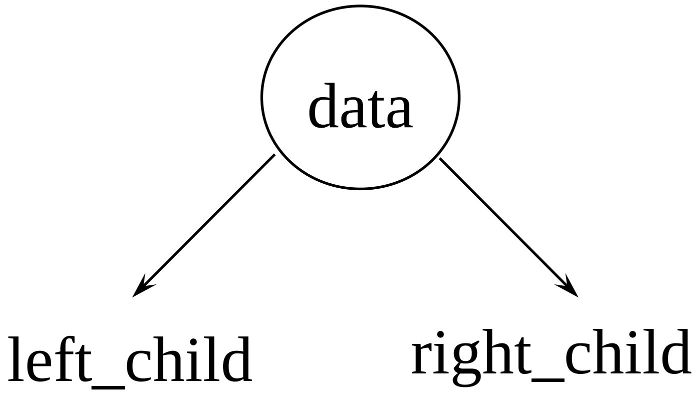
Sequential representation

- Advantages:
  - Direct access to all nodes (Random access)
- Disadvantages:
  - Height of tree should be known
  - Memory may be wasted
  - Insertion and deletion of a node is difficult

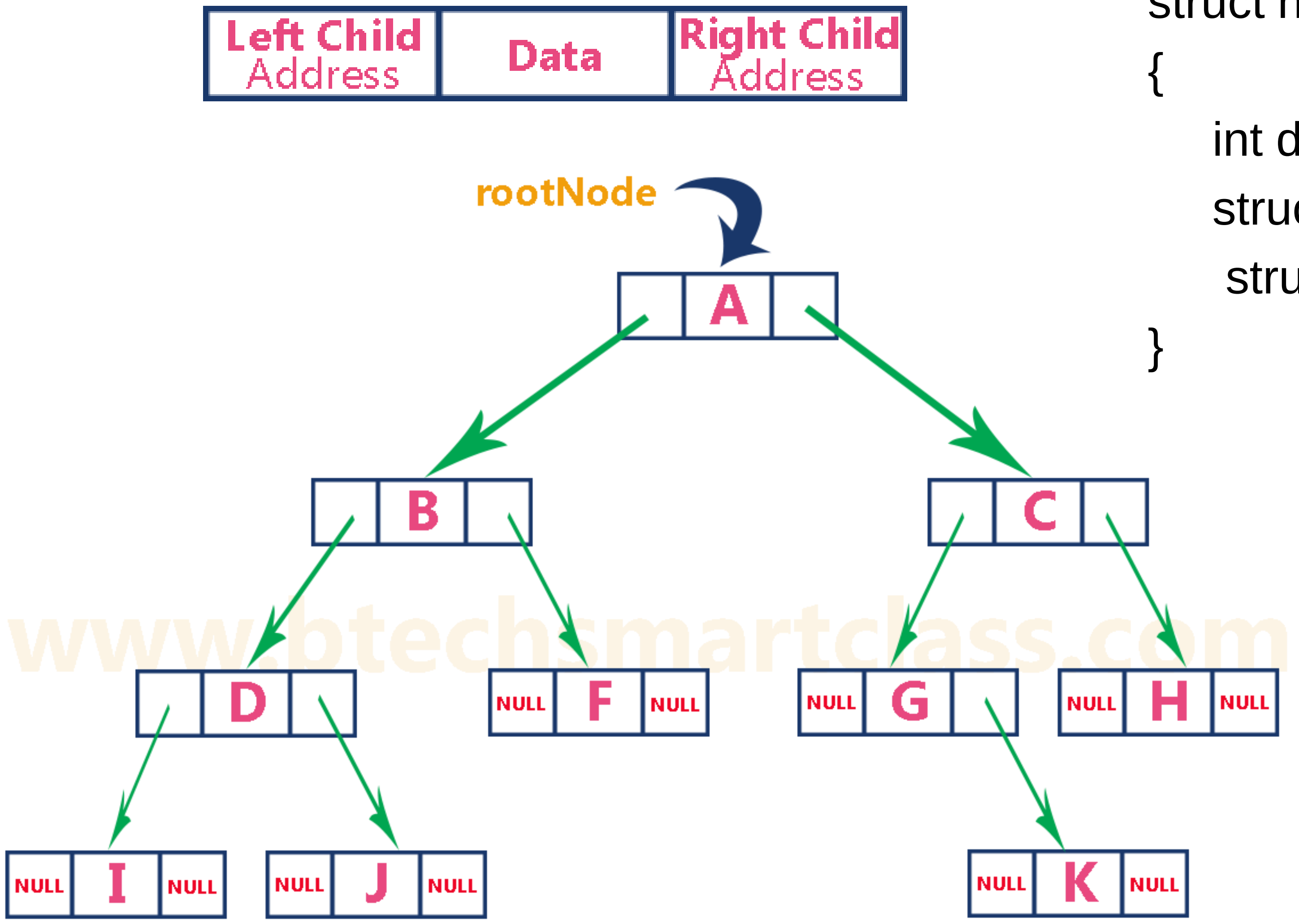
List representation

- Linked representation uses three parallel arrays: INFO, LEFT and RIGHT and a pointer variable ROOT as follows.
  - INFO[K] contains the data at the node N
  - LEFT[K] contains the location of the left child of node N
  - RIGHT[k] contains the location of the right child of node N
- ROOT contains the location of the root R.





List representation



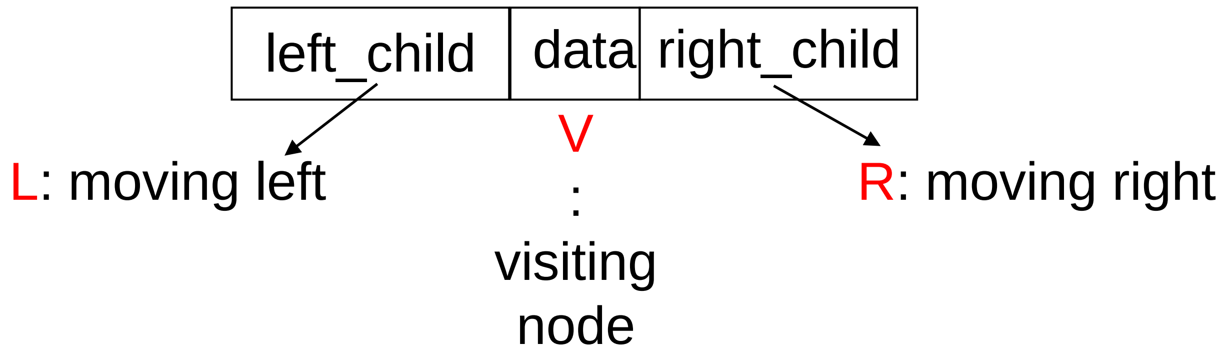
```
struct node
{
    int data;
    struct node *left;
     struct node *rifgt;
}
```

List representation

- Advantages:
  - Height of tree need not be known
  - No memory wastage
  - Insertion and deletion of a node is done without affecting other nodes
- Disadvantages:
  - Direct access to node is difficult
  - Additional memory required for storing address of left and right node

Traversing Binary Trees

- Three main methods:
  - Preorder (VLR)
  - Inorder (LVR)
  - Postorder (LRV)



Traversing Binary Trees

- Preorder:
  - Process the root R
  - Traverse the left subtree of R in preorder
  - Traverse the right subtrees of R in preorder
- Inorder
  - Traverse the left subtree of R in inorder
  - Process the root R
  - Traverse the right subtrees of R in inorder
- Postorder
  - Traverse the left subtree of R in postorder
  - Traverse the right subtrees of R in postorder
  - Process the root R

Traversing Binary Trees

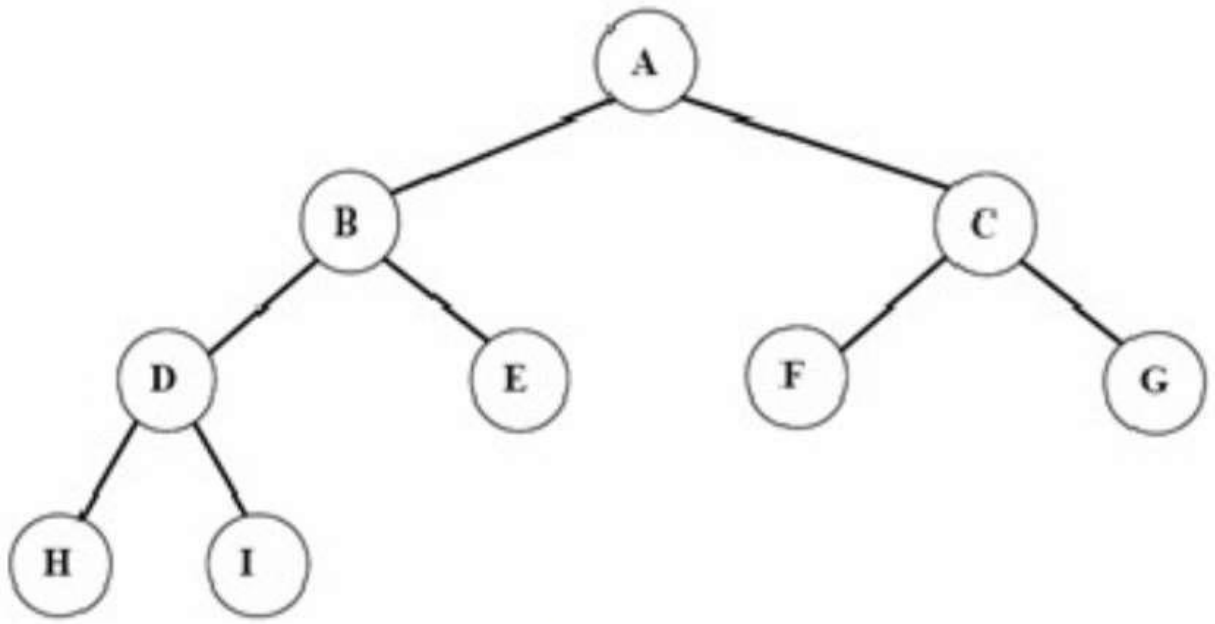


fig Binary tree

The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

Traversing Binary Trees

The in-order traversal output of the given tree is
H D I B E A F C G

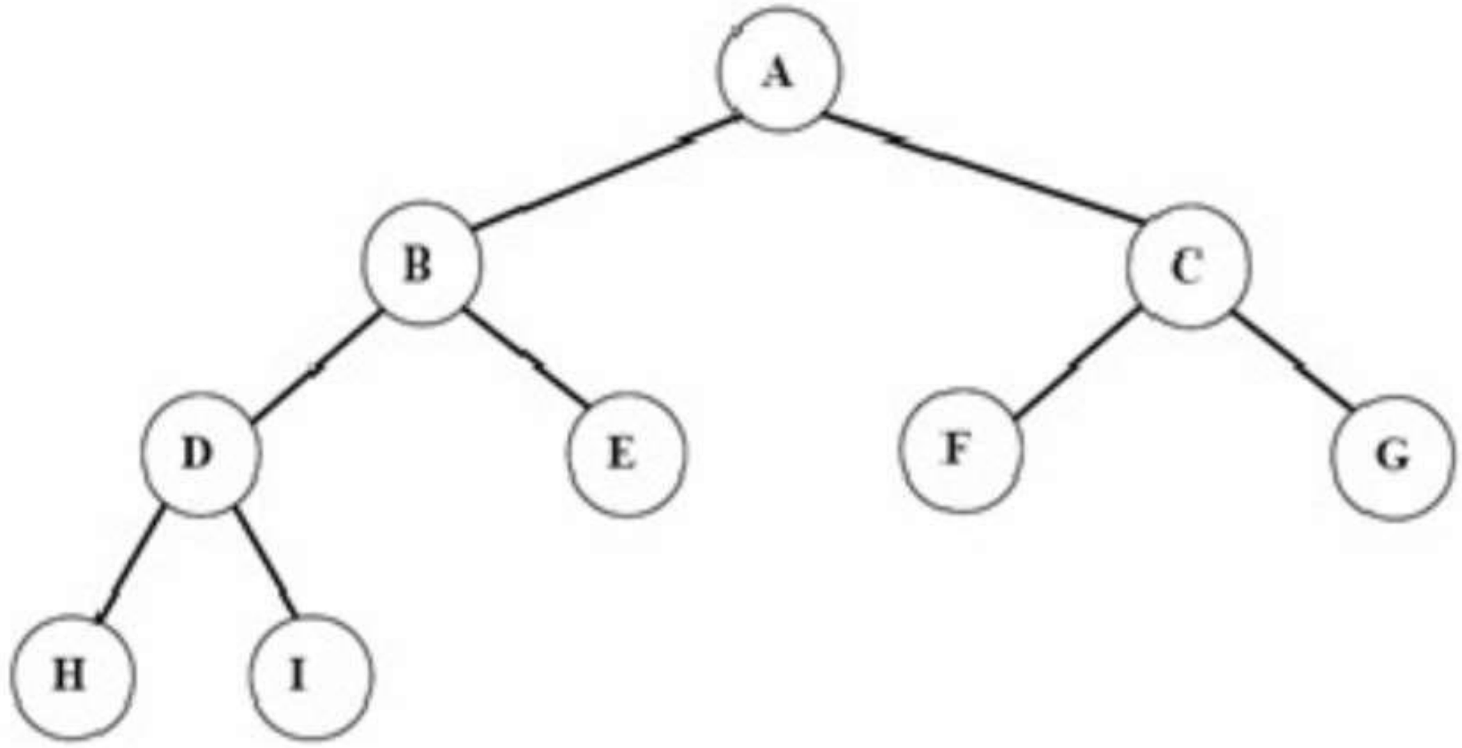


fig Binary tree

Traversing Binary Trees

The in-order traversal output of the given tree is
H I D E B F G C A

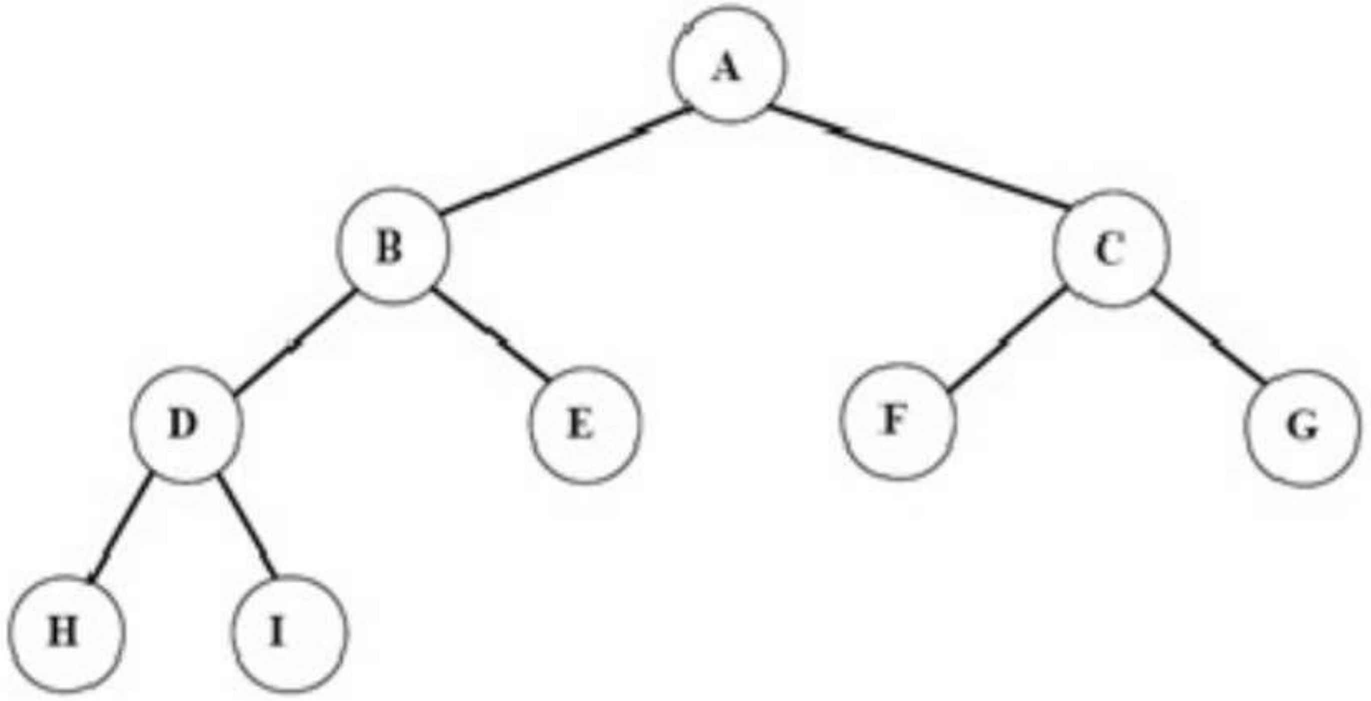
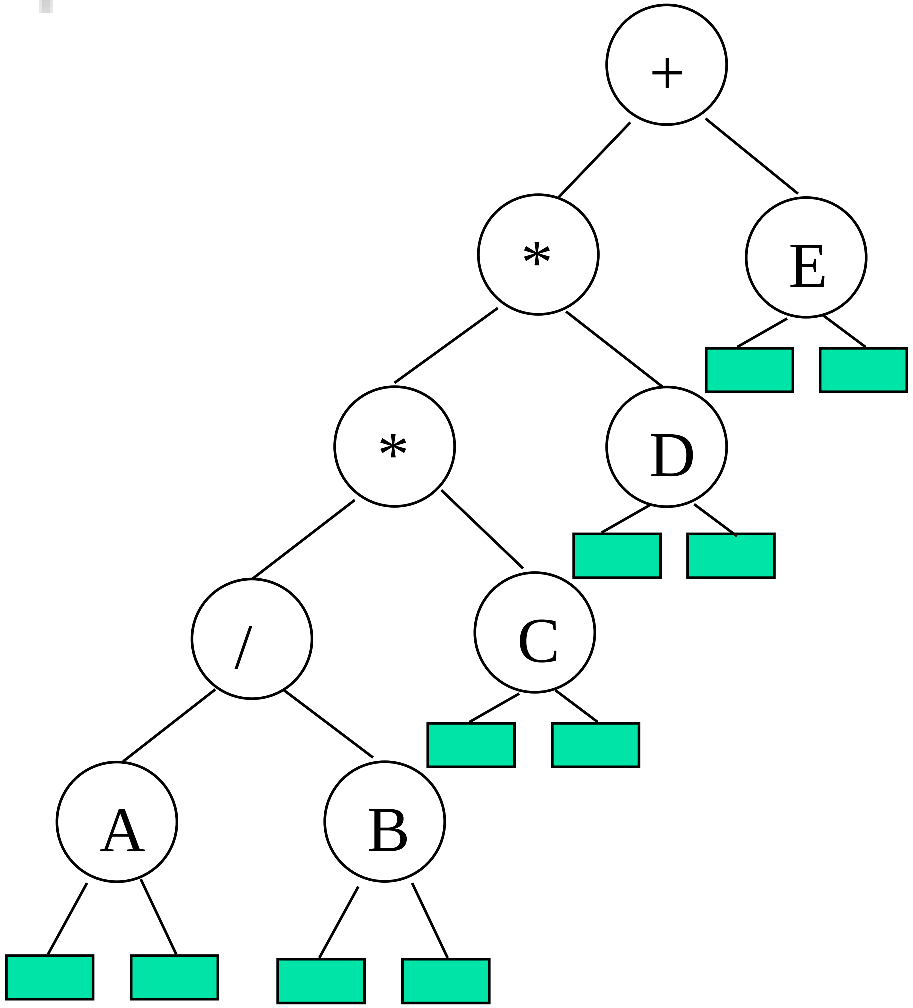


fig Binary tree

Traversing Binary Trees



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

postfix expression

level order traversal

+ * E * D / C A B

Traversal Algorithm Using Stacks

- **Pre-order Tree traversal algorithm:**

  Initially push NULL onto STACK and then set PTR:= ROOT. Then repeat the following steps until PTR = NULL or equivalently while PTR ≠ NULL.

  a) Proceed down the left-most path rooted at PTR, processing each node N on the path and pushing each child R(N), if any, onto STACK. The traversing ends after a node N with no left child L(N) is processed. [thus PTR is updated using the assignment PTR:= LEFT[PTR] and the traversing stops when LEFT[PTR] = NULL]

  b) [Backtracking] Pop and assign to PTR the top element on STACK. If PTR ≠ NULL, then return to Step(a); otherwise Exit.

Traversal Algorithm Using Stacks

- **PREORD (INFO, LEFT, RIGHT, ROOT)**

A binary tree T is in memory. The algorithm does a pre-order traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1.  [Initially push NULL onto STACK and initialize PTR.]
    Set TOP:=1, STACK[1]:=NULL and PTR:= ROOT
2.  Repeat Steps 3 to 5 while PTR ≠ NULL
3.        Apply PROCESS to INFO[PTR]
4.        [Right child?]
           If RIGHT[PTR] ≠ NULL, then: [Push on STACK]
              Set TOP:= TOP + 1, and STACK[TOP]: = RIGHT[PTR]
              [end of If structure]
5.         [Left child?]
           If LEFT[PTR] ≠ NULL, then:
                  Set PTR:= STACK[TOP] and TOP:= TOP - 1
              [end of If structure]
     [End of Step 2 loop]
6.  Exit
```

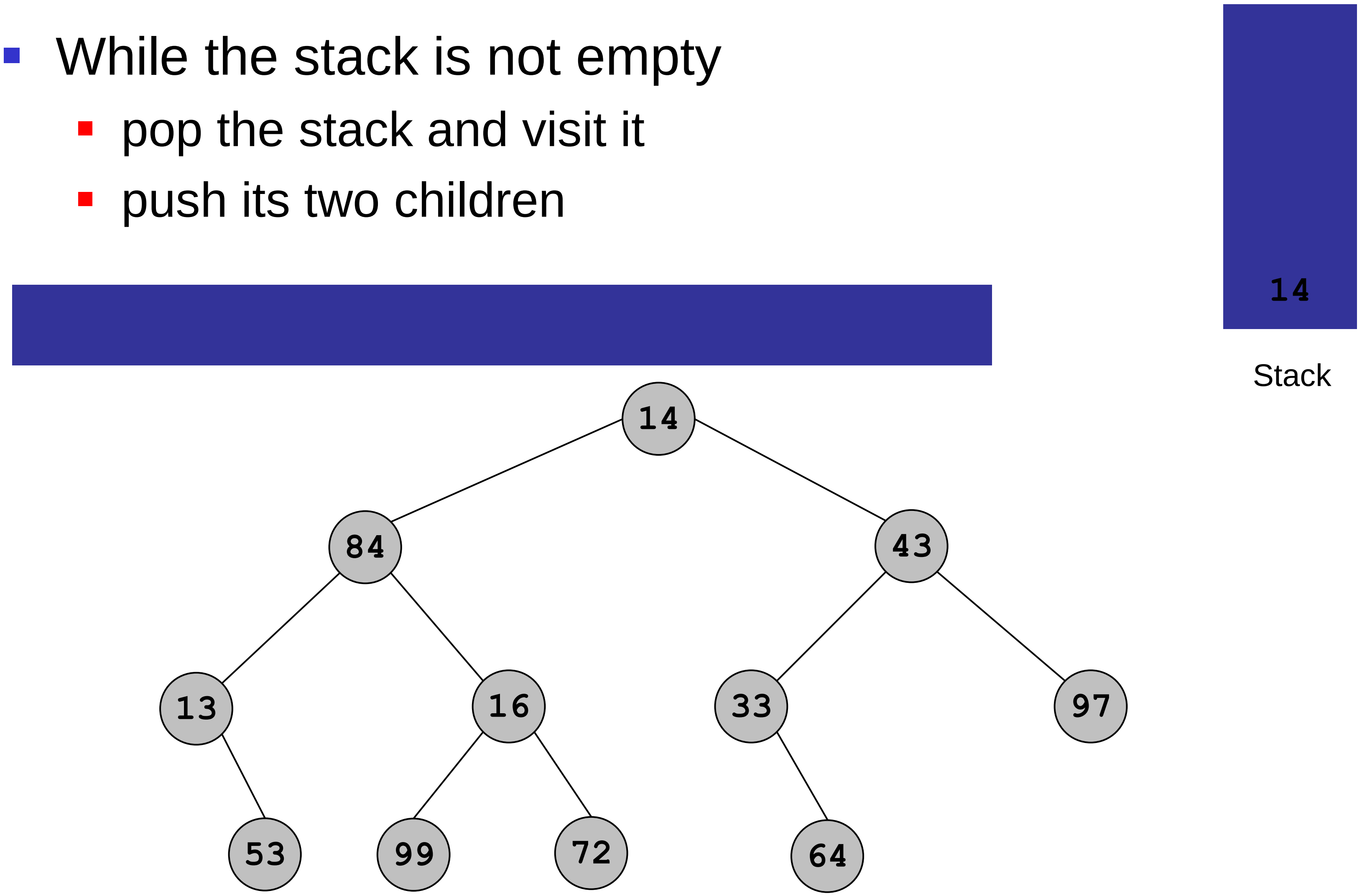
Traversal Algorithm Using Stacks

- **Pre-order Tree traversal algorithm:**

```
public void preorder (BinaryTreeNode<T> r) {
    if (r != null) {
        visit(r);
        preorder (r.getLeftChild());
        preorder (r.getRightChild());
    }
}
```

Traversal Algorithm Pre-order

- Push the root onto the stack.
- While the stack is not empty
  - pop the stack and visit it
  - push its two children

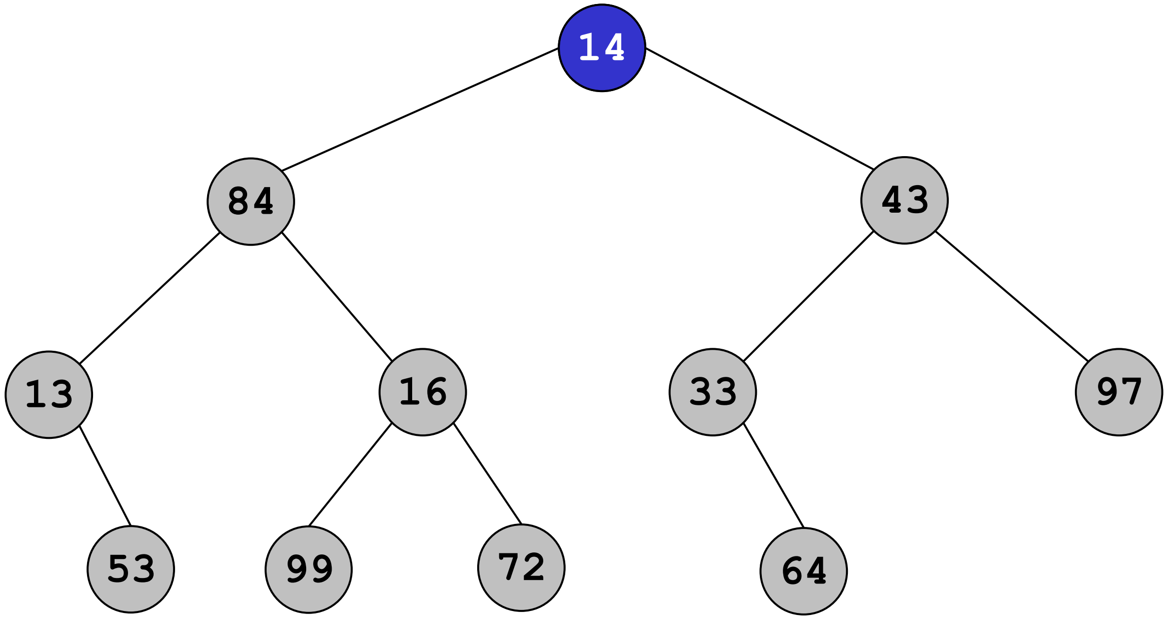


Traversal Algorithm Pre-order

14

84
43

Stack

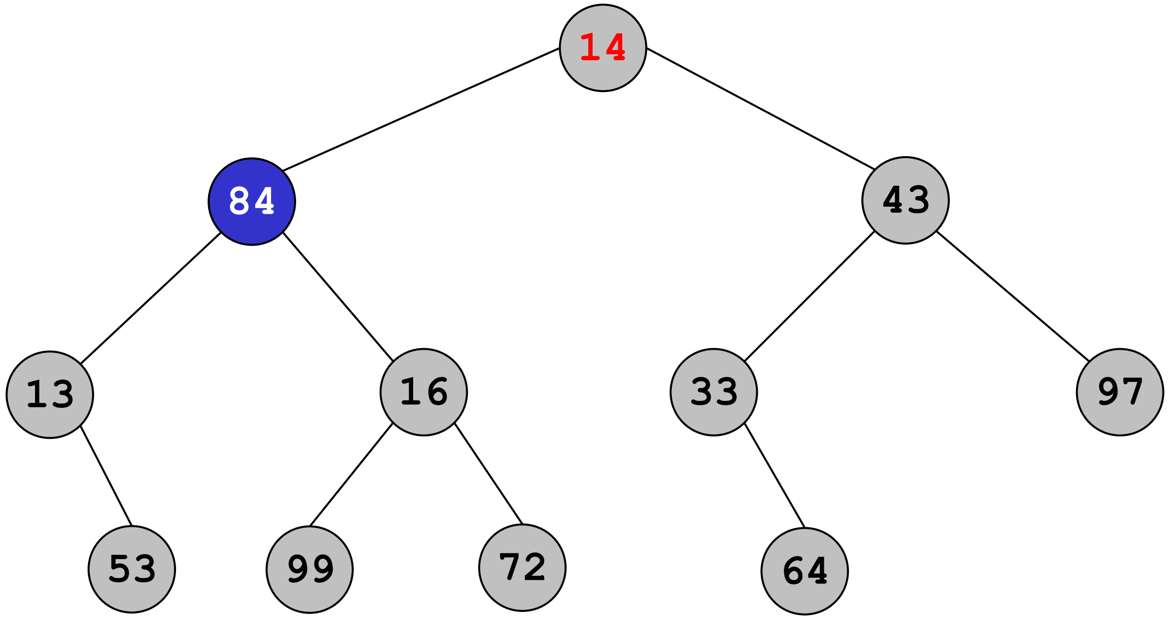


Traversal Algorithm Pre-order

14 84

13
16
43

Stack

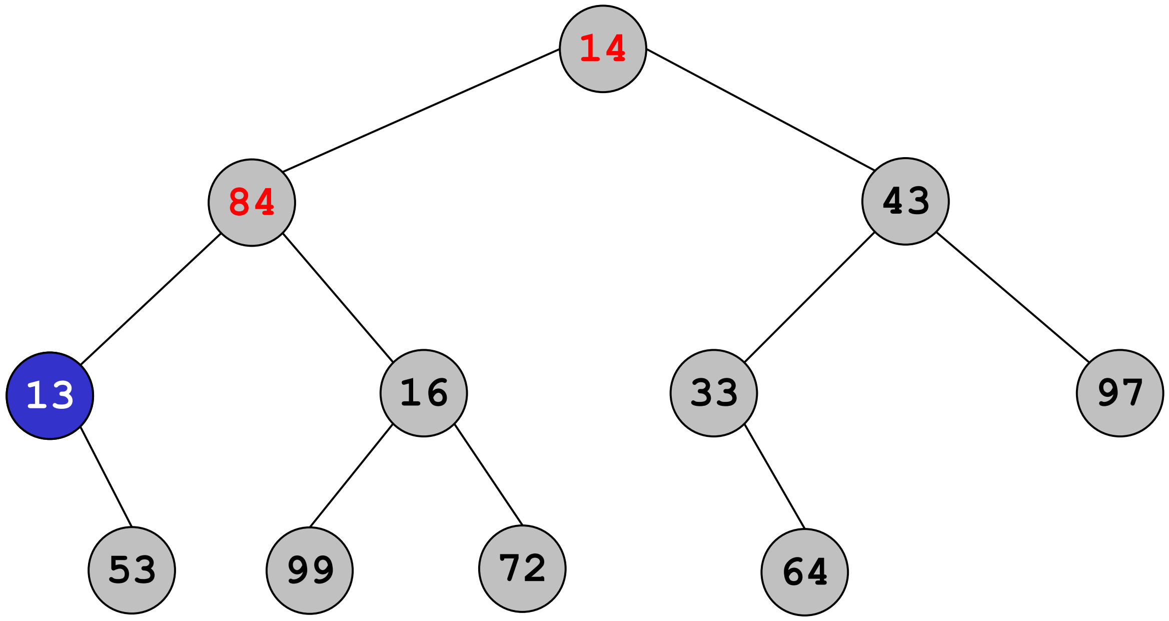


Traversal Algorithm Pre-order

14 84 13

53
16
43

Stack

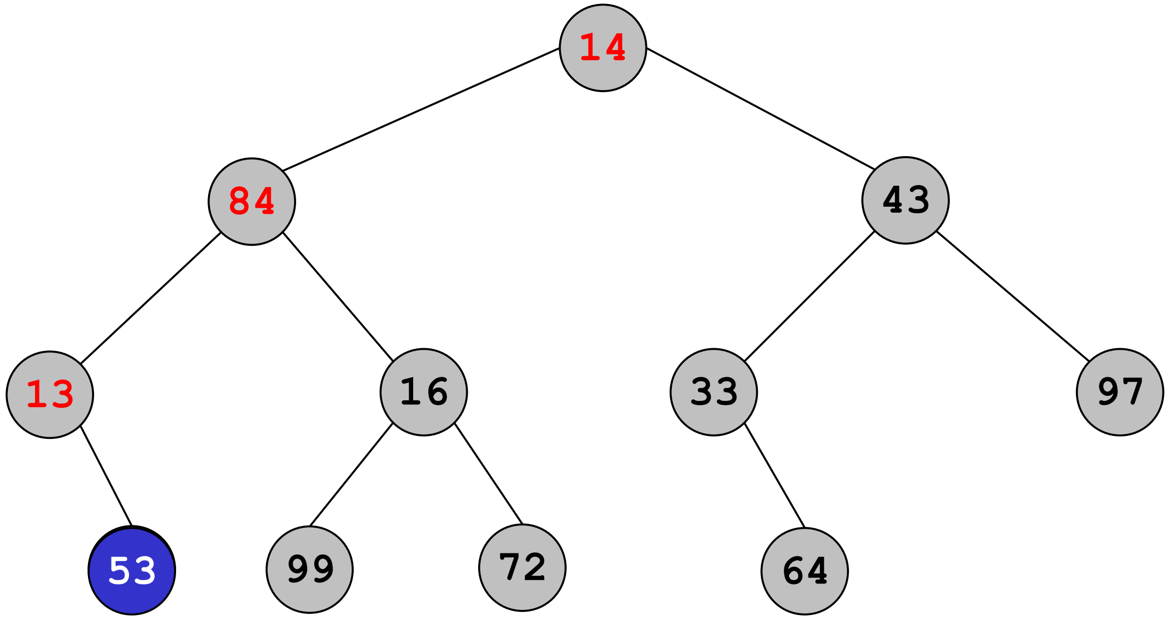


Traversal Algorithm Pre-order

14 84 13 53

16
43

Stack

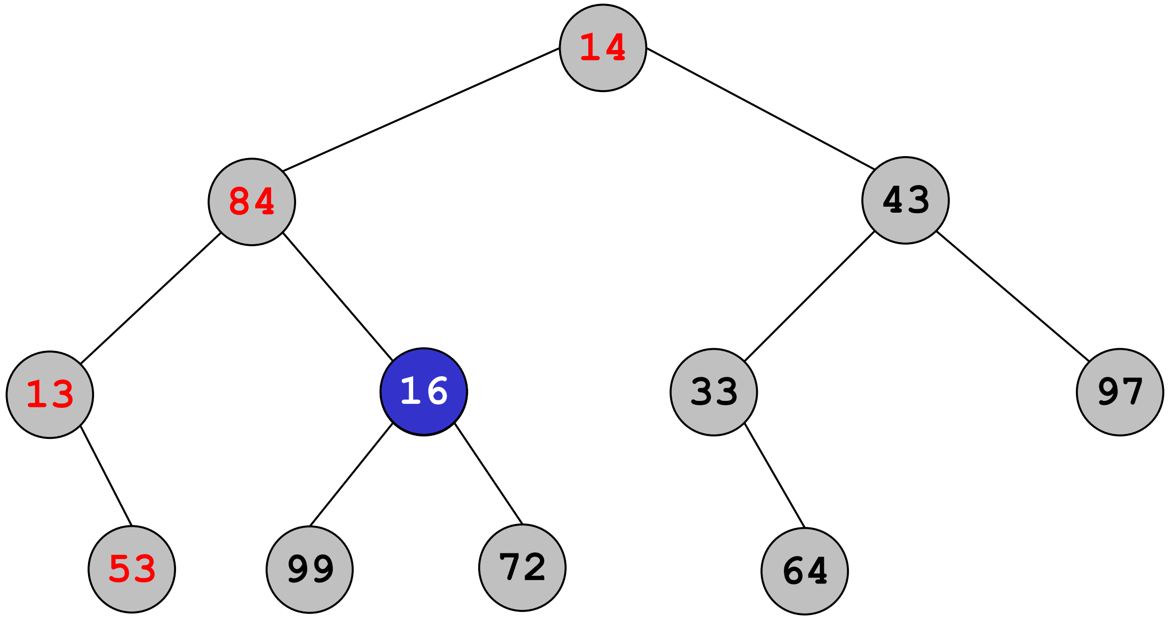


Traversal Algorithm Pre-order

14 84 13 53 16

99
72
43

Stack

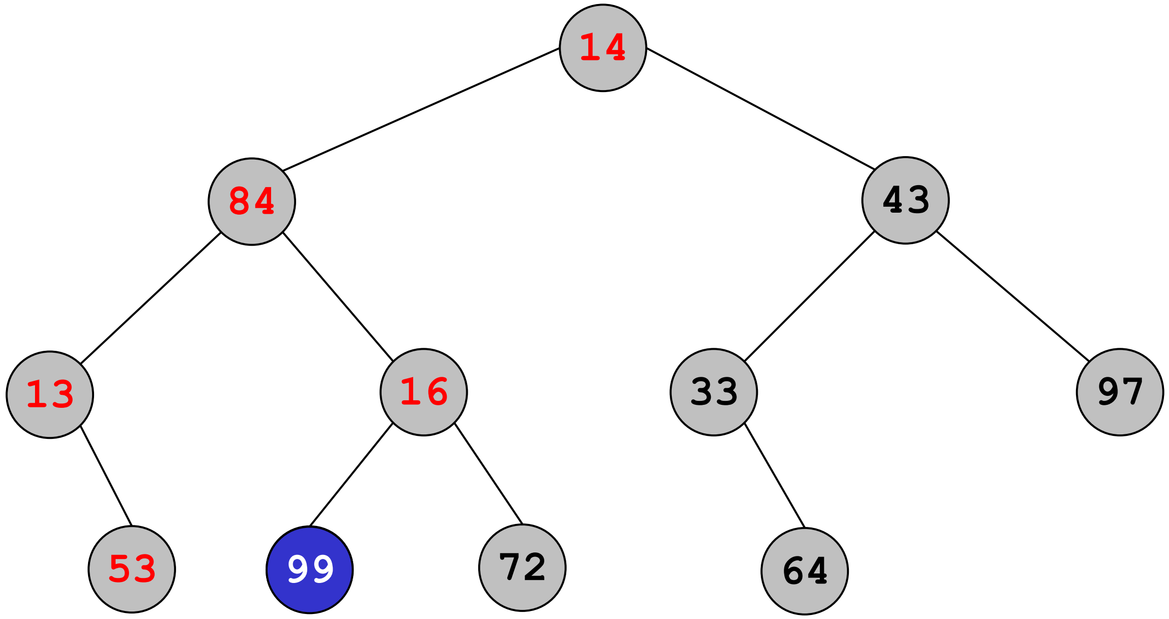


Traversal Algorithm Pre-order

14 84 13 53 16 99

72
43

Stack

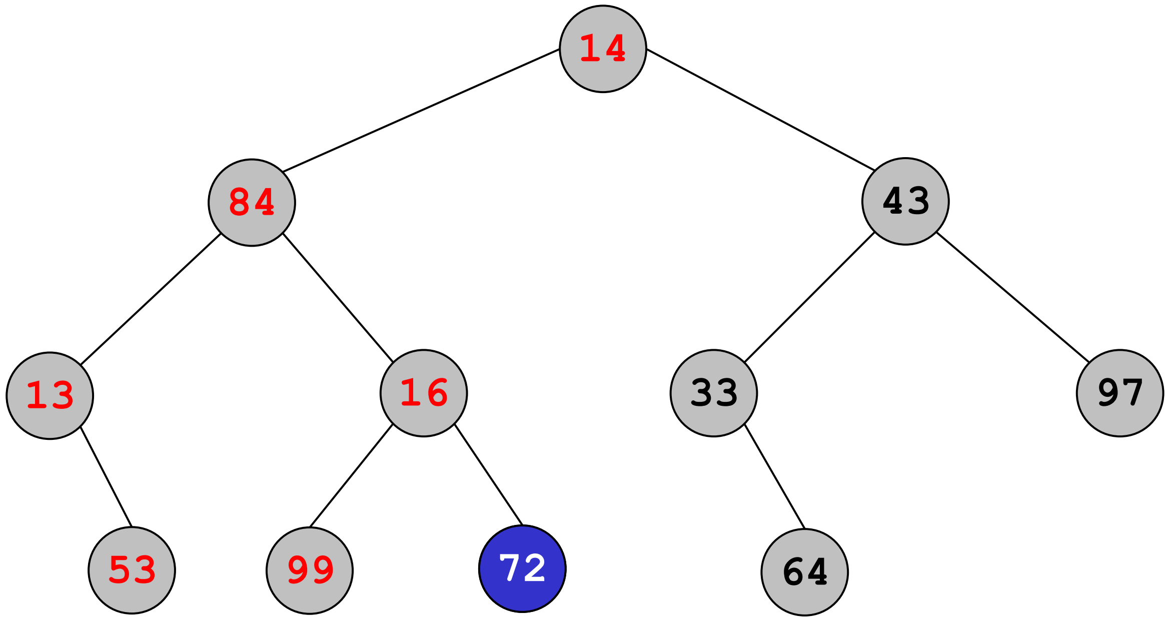


Traversal Algorithm Pre-order

14 84 13 53 16 99 72

43

Stack

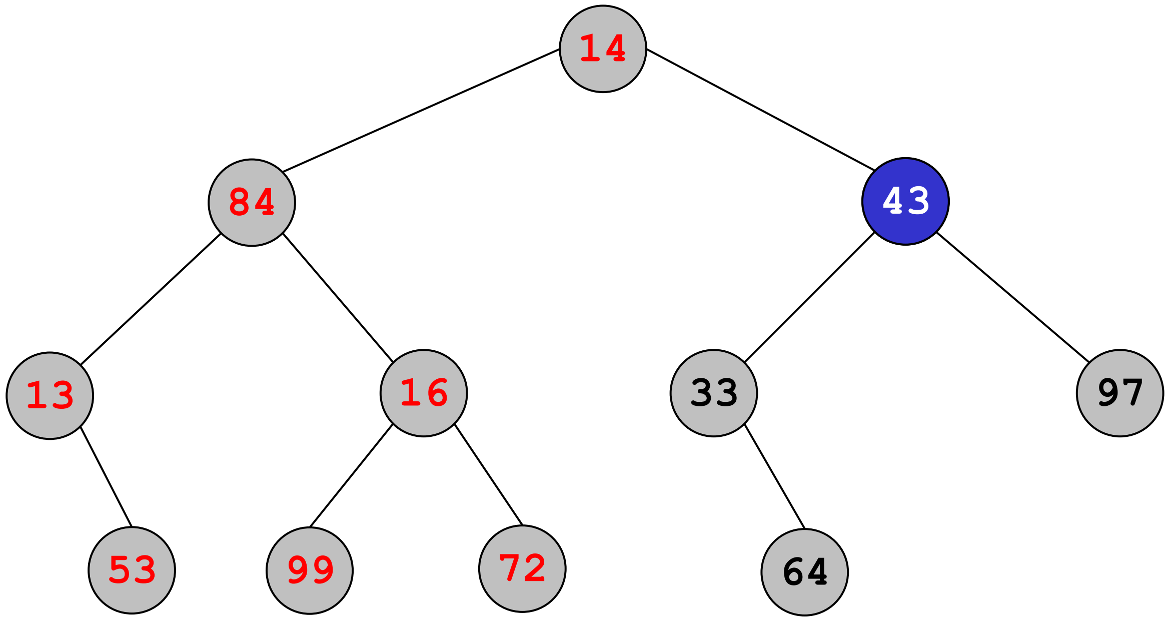


Traversal Algorithm Pre-order

14 84 13 53 16 99 72 43

33
97

Stack

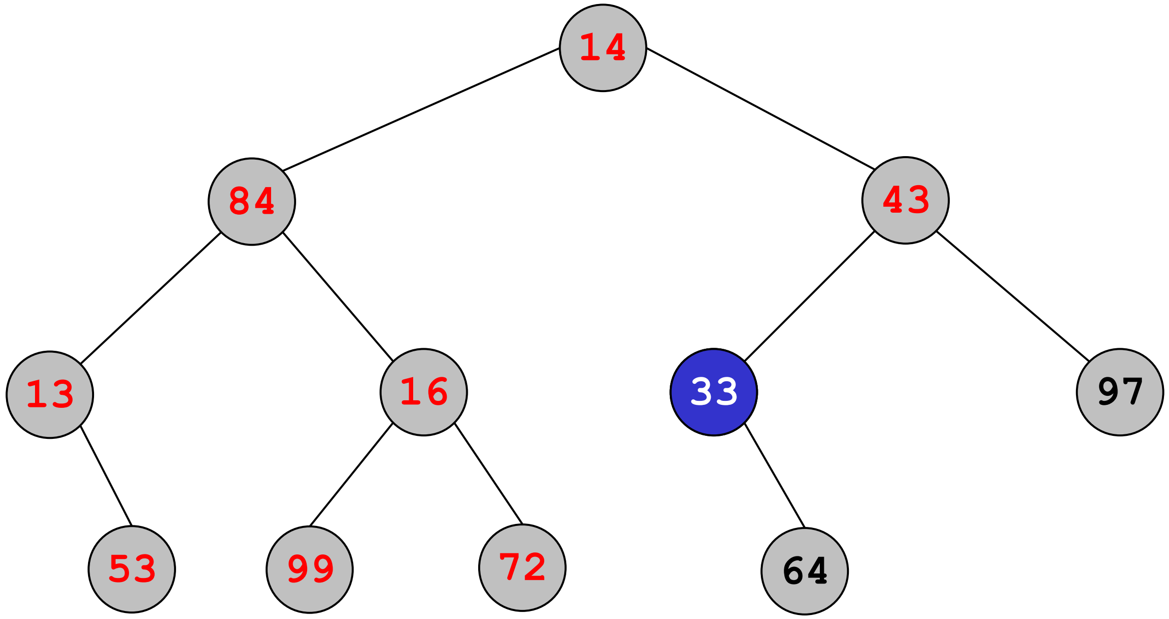


Traversal Algorithm Pre-order

14 84 13 53 16 99 72 43 33

64
97

Stack

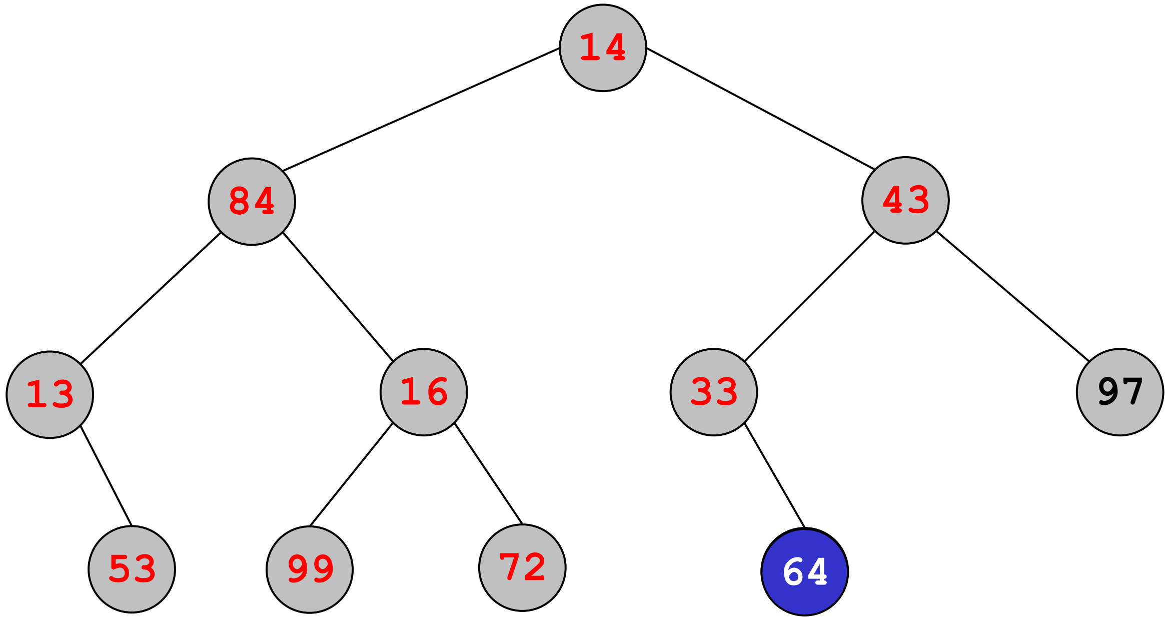


Traversal Algorithm Pre-order

14 84 13 53 16 99 72 43 33 64

97

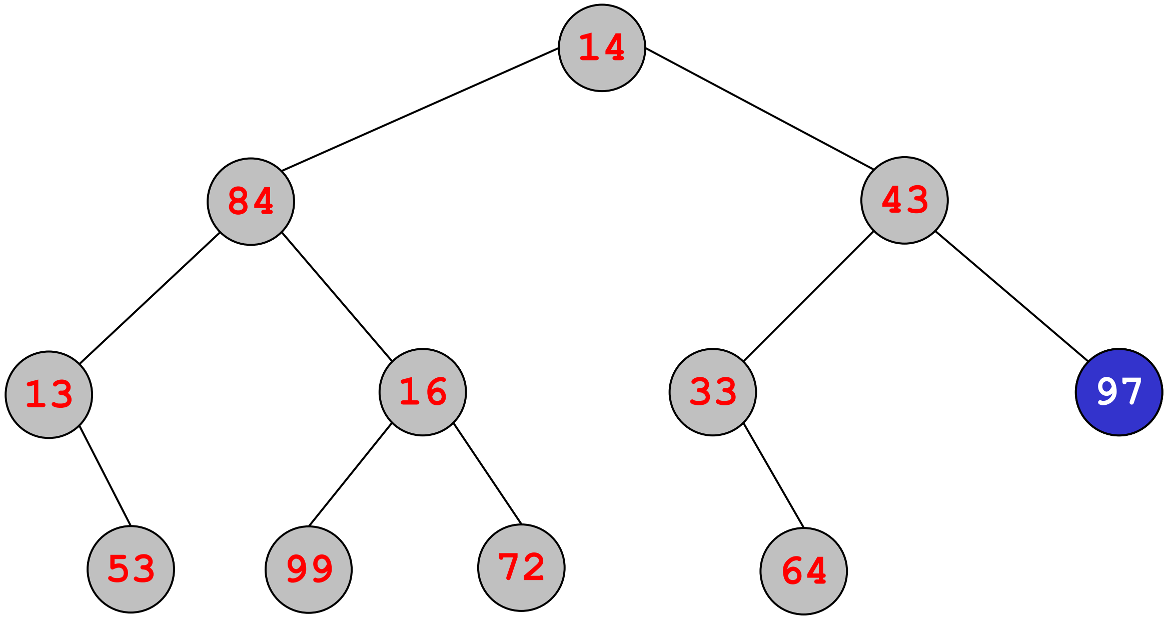
Stack



Traversal Algorithm Pre-order

14 84 13 53 16 99 72 43 33 64 97

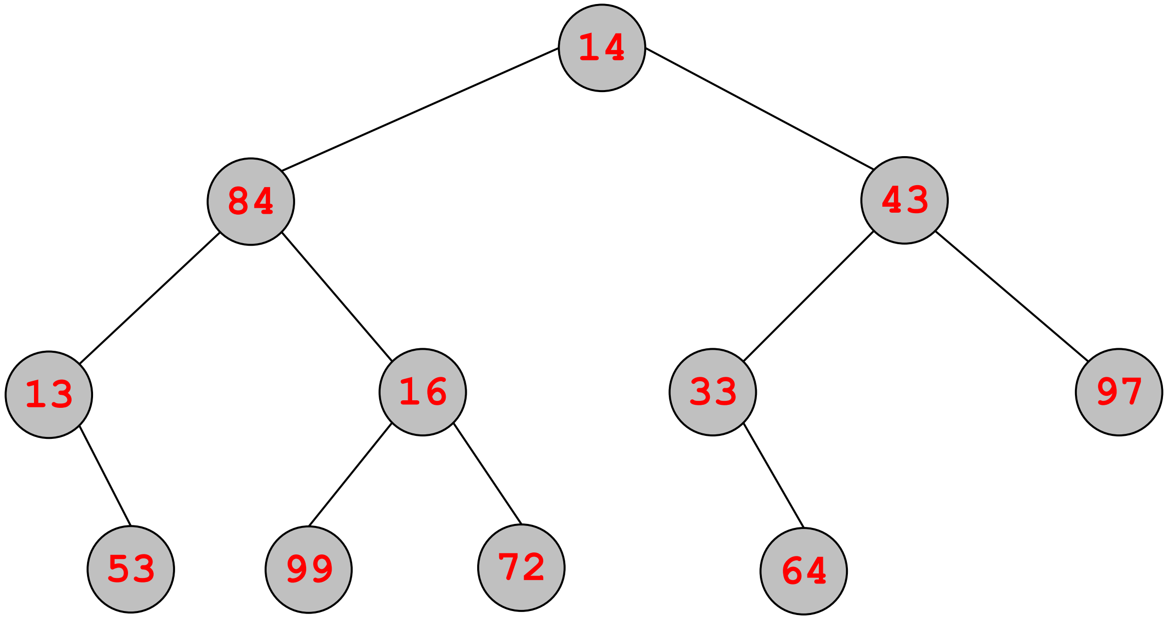
Stack



Traversal Algorithm Pre-order

```
14 84 13 53 16 99 72 43 33 64 97
```

Stack



Traversal Algorithm Using Stacks

Inorder Tree traversal algorithm:

Initially push NULL onto STACK and then set PTR:= ROOT. Then repeat the following steps until NULL is popped from STACK.

a) Proceed down the left-most path rooted at PTR, pushing each node N on STACK and stopping when a node N with no left child is pushed onto STACK.

b) [Backtracking] Pop and process the node on STACK. If NULL is popped, then Exit. If a node N with a right child R(N) is processed, set PTR = R(N). (by assigning PTR:= RIGHT[PTR]) and return to Step (a).

Traversal Algorithm Using Stacks

- **INORD (INFO, LEFT, RIGHT, ROOT)**

  A binary tree T is in memory. This algorithm does a inorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1.  [Push NULL onto STACK and initialize PTR.]
    Set TOP:=1, STACK[1]:=NULL and PTR:= ROOT
2.  Repeat while PTR ≠ NULL: [Pushes left-most path onto STACK]
    a)  Set TOP:= TOP + 1 and STACK[TOP]:= PTR [saves node]
    b)  Set PTR:= LEFT[PTR].  [update PTR]
      [end of loop]
3.  Set PTR:= STACK[TOP] and TOP := TOP – 1 [pops node from
    STACK]
4.  Repeat steps 5 to 7while PTR ≠ NULL: [Backtracking]
5.        Apply PROCESS to INFO[PTR]
6.        [Right child?] If RIGHT[PTR] ≠ NULL then:
            a)  Set PTR:= RIGHT[PTR]
            b)  Go to step 3
     [end of If structure]
```

Traversal Algorithm Using Stacks

- **INORD (INFO, LEFT, RIGHT, ROOT)**

```
7.    Set PTR:= STACK[TOP] and TOP:= TOP-1. [pop node]
      [End of step 4 loops]
8.    Exit
```

Traversal Algorithm Using Stacks

- **Inorder Tree traversal algorithm:**

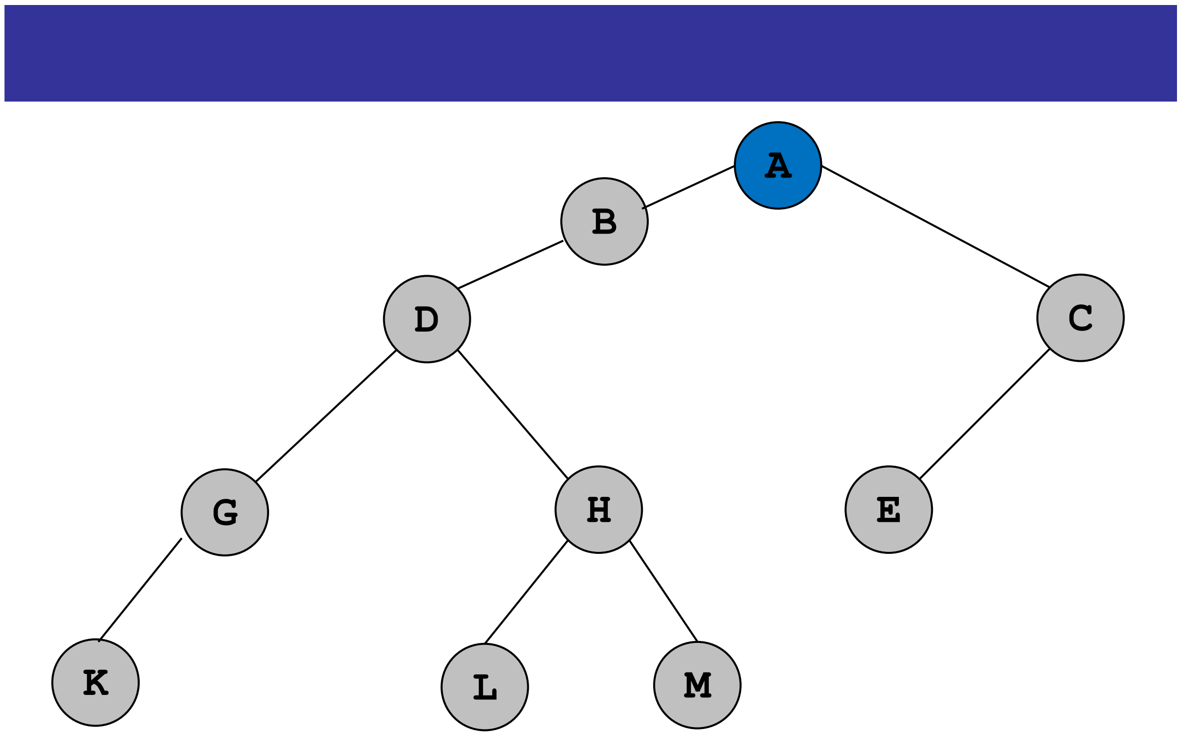
```
public void inorder (BinaryTreeNode<T> r) {
    if (r != null) {
        inorder (r.getLeftChild());
        visit(r);
        inorder (r.getRightChild());
    }
}
```

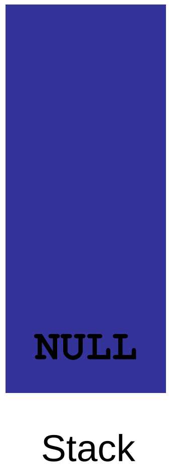
Traversal Algorithm Inorder

- Initially push NULL onto the stack.

  STACK: NULL

  then set PTR: = A, the root of T



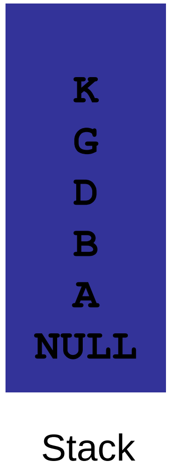


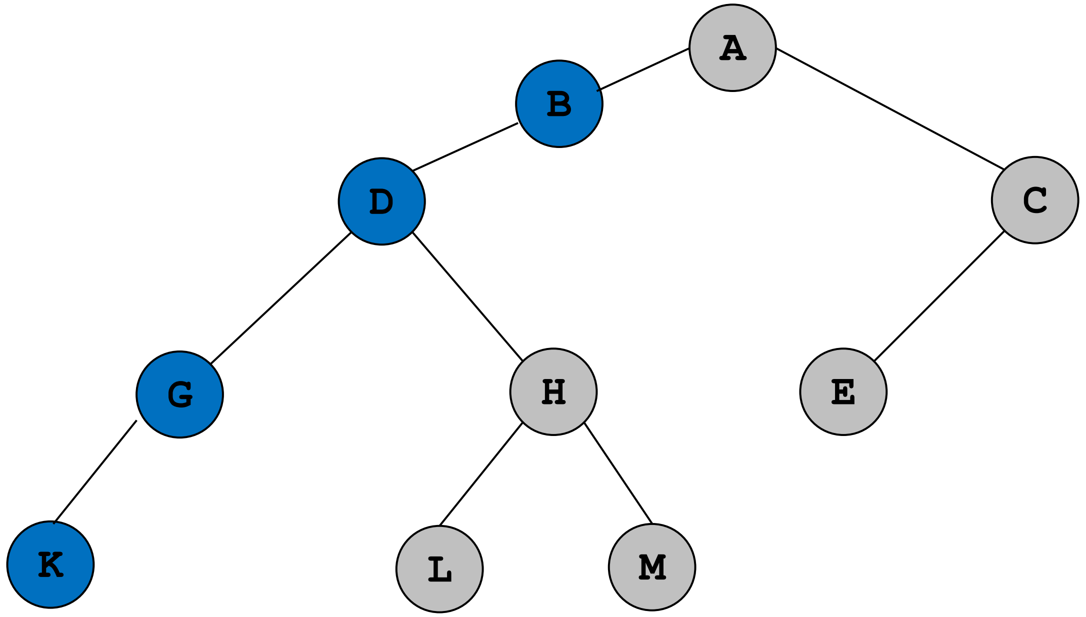
Traversal Algorithm Inorder

- Processed down the left-most path rooted at PTR = A, pushing the nodes A, B, D, G, K onto the stack.

  STACK: NULL, A, B, D, G, K

  (No other node is pushed onto STACK, since K has no left child)





Traversal Algorithm Inorder

- [Backtracking] nodes K, G, D are popped and processed.

  STACK: NULL, A, B

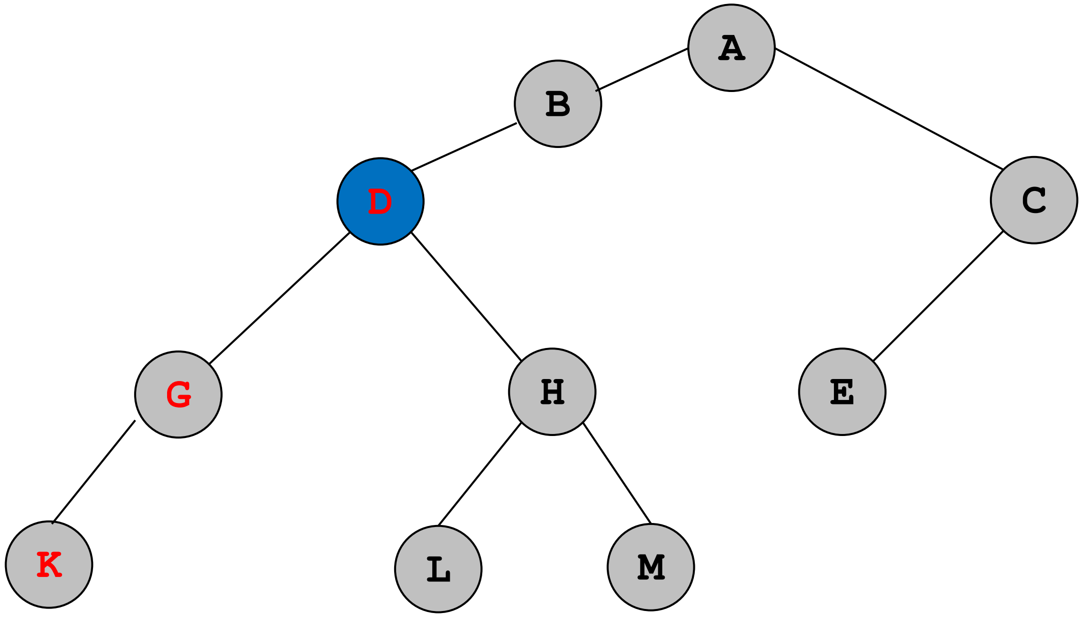
  (we stop processing at D, since D has a right child)

  then PRT:=H

K G D

B
A
NULL

Stack



Traversal Algorithm Inorder

- Processed down the left-most path rooted at PRT:=H, pushing the nodes H and L onto STACK.

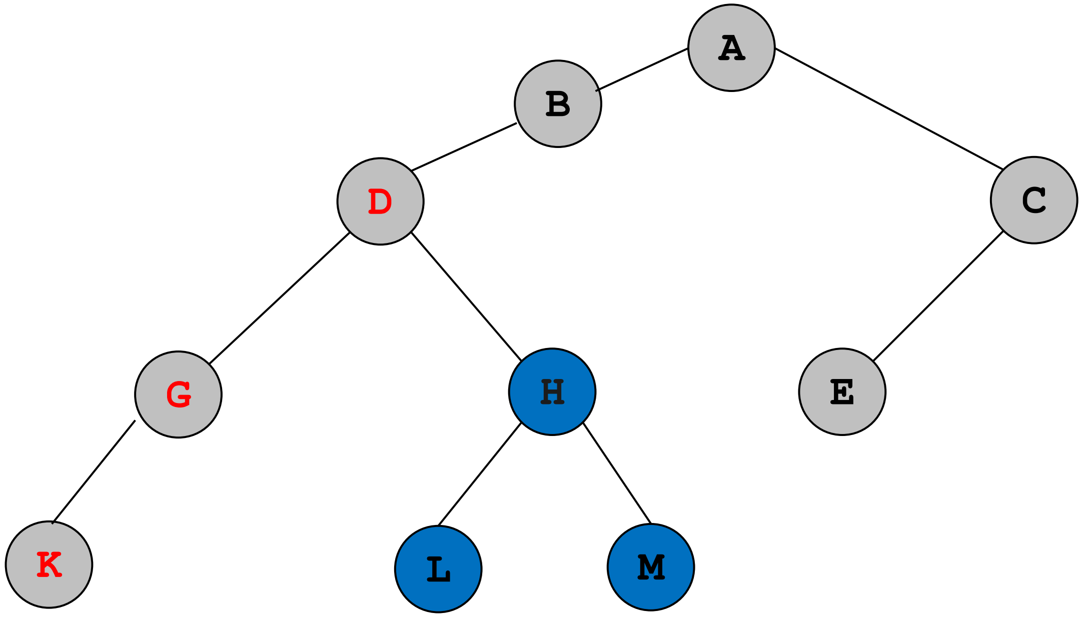
  STACK: NULL, A, B, H, L

(No other node is pushed onto STACK, since L has no left child)

K G D

L
H
B
A
NULL

Stack



Traversal Algorithm Inorder

- [Backtracking] nodes L and H are popped and processed.

  STACK: NULL, A, B

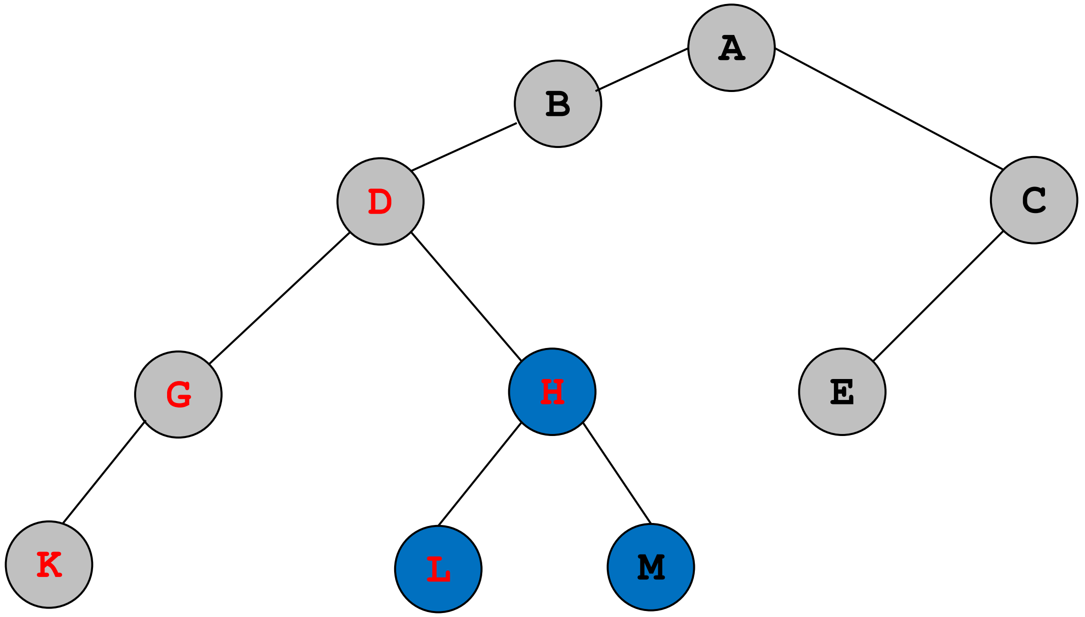
  (we stop processing at H, since D has a right child)

  then PRT:=M

K G D L H

B
A
NULL

Stack



Traversal Algorithm Inorder

- Processed down the left-most path rooted at PRT:=M, pushing the nodes M onto STACK.

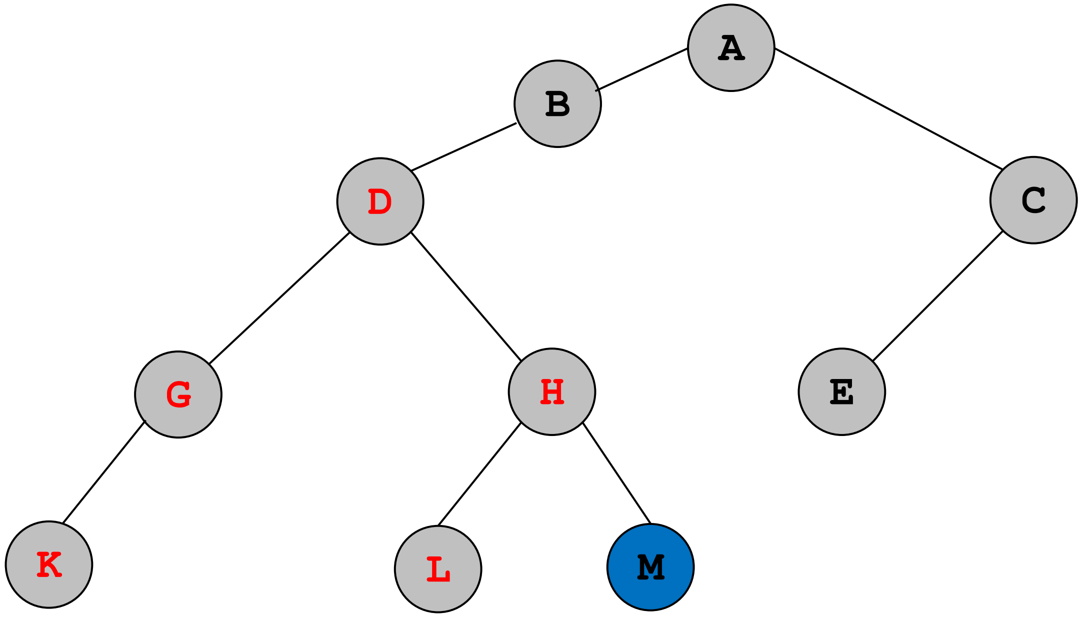
  STACK: NULL, A, B, M

(No other node is pushed onto STACK, since M has no left child)

K G D L H

M
B
A
NULL

Stack



Traversal Algorithm Inorder

- [Backtracking] nodes M, B and A are popped and processed.

  STACK: NULL

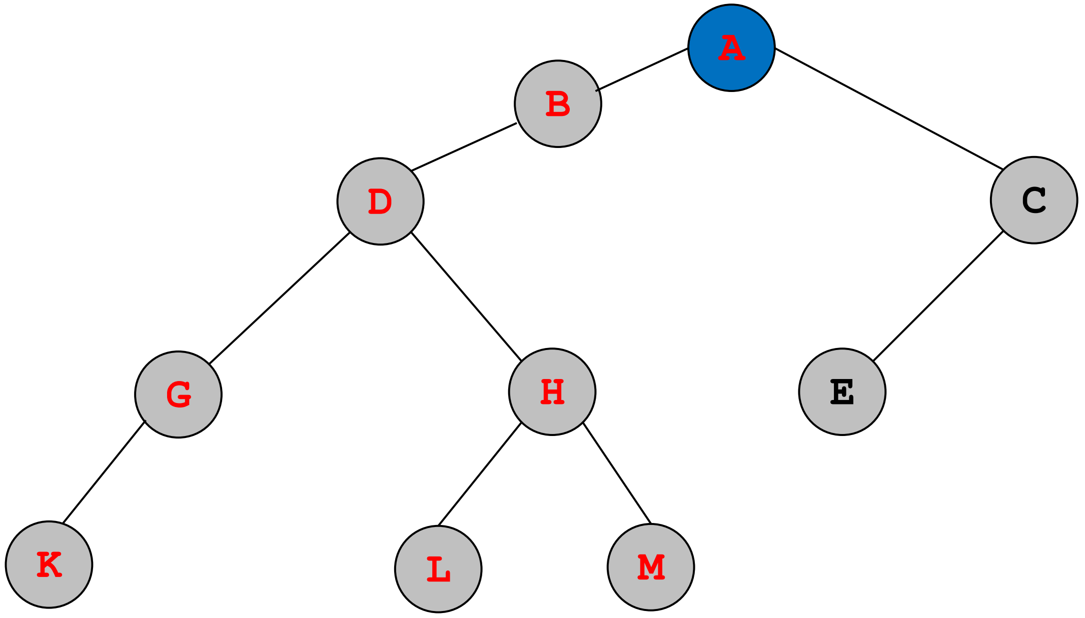
(no other element of STACK is popped, since A has a right child)

then set PRT:= C

K G D L H M B A

NULL

Stack



Traversal Algorithm Inorder

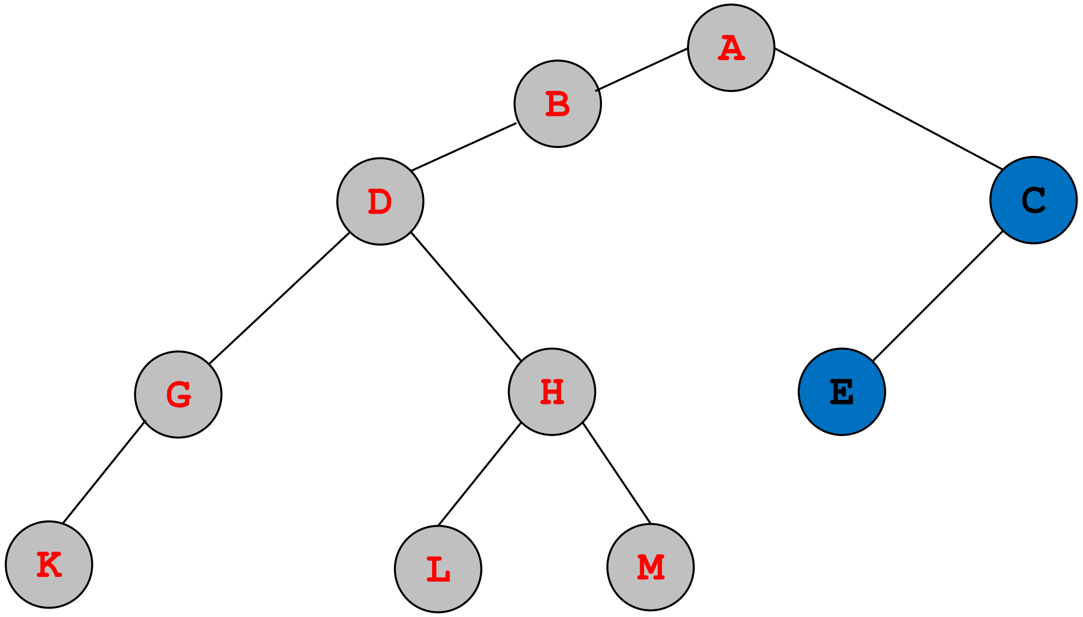
- Processed down the left-most path rooted at PRT:=C, pushing the nodes C and E onto STACK.

  STACK: NULL, C, E

K G D L H M B A

E
C
NULL

Stack



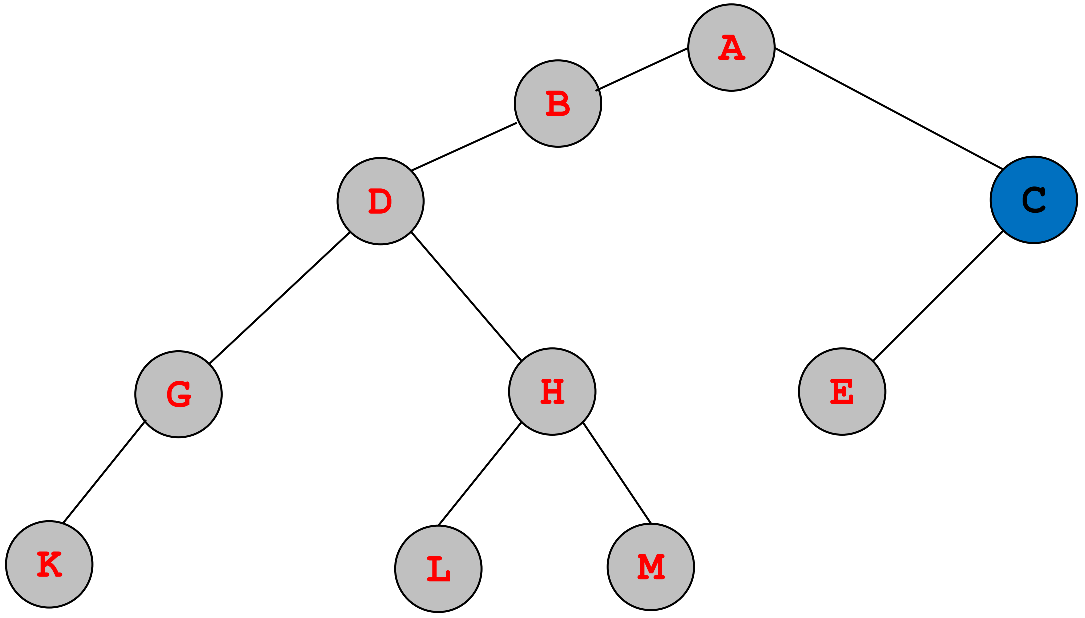
Traversal Algorithm Inorder

- [Backtracking] nodes E is popped and processed. Science E has no right child, node C is popped and processed. Science C has no right child, the next element is NULL, is popped from STACK.

K G D L H M B A E

C
NULL

Stack



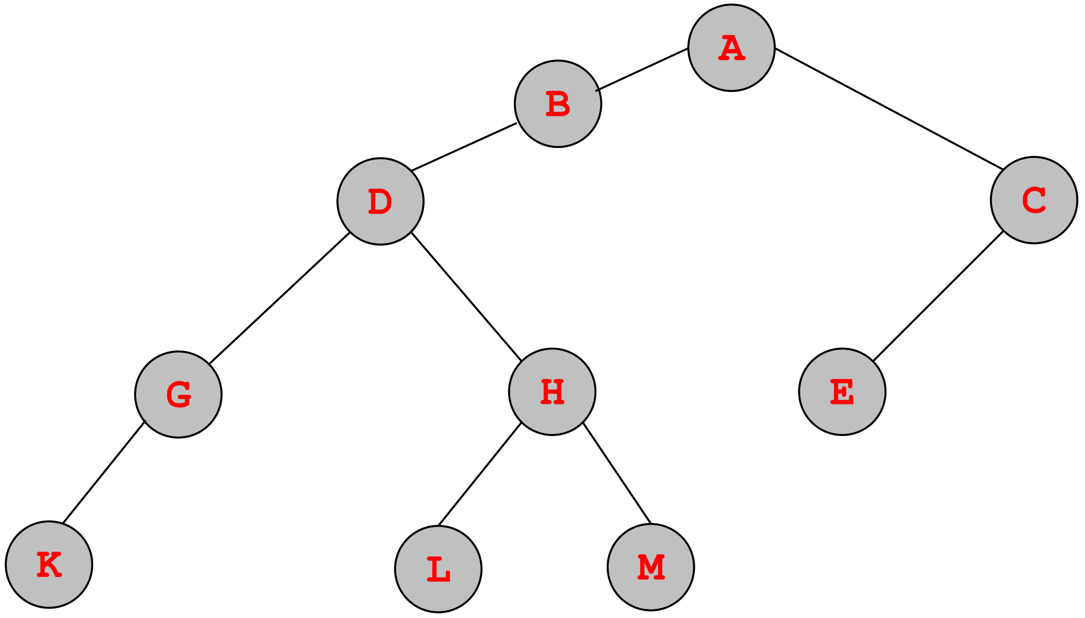
Traversal Algorithm Inorder

- [Backtracking] nodes E is popped and processed. Science E has no right child, node C is popped and processed. Science C has no right child, the next element is NULL, is popped from STACK.

K G D L H M B A C

NULL

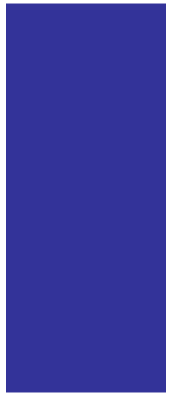
Stack

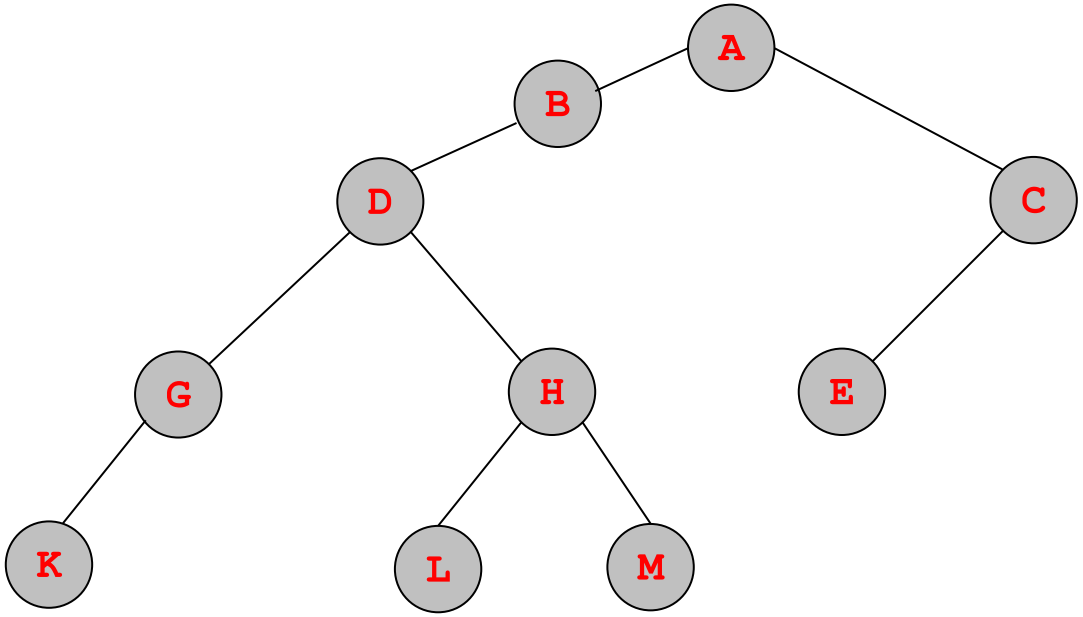


Traversal Algorithm Inorder

- [Backtracking] nodes E is popped and processed. Science E has no right child, node C is popped and processed. Science C has no right child, the next element is NULL, is popped from STACK.

K G D L H M B A C


Stack



Traversal Algorithm Using Stacks

Post-order Tree traversal algorithm:

Initially push NULL onto STACK and then set PTR:= ROOT. Then repeat the following steps until NULL is popped from STACK.

a) Proceed down the left-most path rooted at PTR. At each node N of the path, push N onto STACK and if N has a right child R(N), push -R(N) onto STACK.

b) [Backtracking] Pop and process positive nodes on STACK. If NULL is popped, then Exit. If a negative node is popped, that is, if PTR = -N for some node N, set PTR = N (by assigning PTR:= -PTR) and return to Step (a).

Traversal Algorithm Using Stacks

- **POSTORD (INFO, LEFT, RIGHT, ROOT)**

  A binary tree T is in memory. This algorithm does a postorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1. [Push NULL onto STACK and initialize PTR.]
   Set TOP:=1, STACK[1]:=NULL and PTR:= ROOT
2. [Push left-most path onto STACK]
   Repeat steps 3 to 5 while PTR ≠ NULL
3.      Set TOP := TOP + 1 and STACK[TOP] := PTR
   [Pushes PTR on STACK]
4.      If  RIGHT[PTR] ≠ NULL then:   [Pushes on STACK]
            Set TOP := TOP + 1 and STACK[TOP] := -RIGHT[PTR]
        [end of If structure]
5.      Set PTR:= LEFT[PTR] [Updates pointer PTR]
   [End of step 2 loops]
6. Set PTR:= STACK[TOP] and TOP := TOP – 1 [pop node]
```

Traversal Algorithm Using Stacks

- **POSTORD (INFO, LEFT, RIGHT, ROOT)**

```
7.  Repeat while PTR>0
    a)  Apply PROCESS to INFO[PTR]
    b)  Set PTR:= STACK[TOP] and TOP:= TOP-1.
         [Pops node from STACK]
    [End of loop]
8.  If PTR < 0, then:
    a)  Set PTR := - PTR
    b)  Go to Step 2.
    [End of If structure]
9.  Exit
```

Traversal Algorithm Using Stacks

- **Post-order Tree traversal algorithm:**

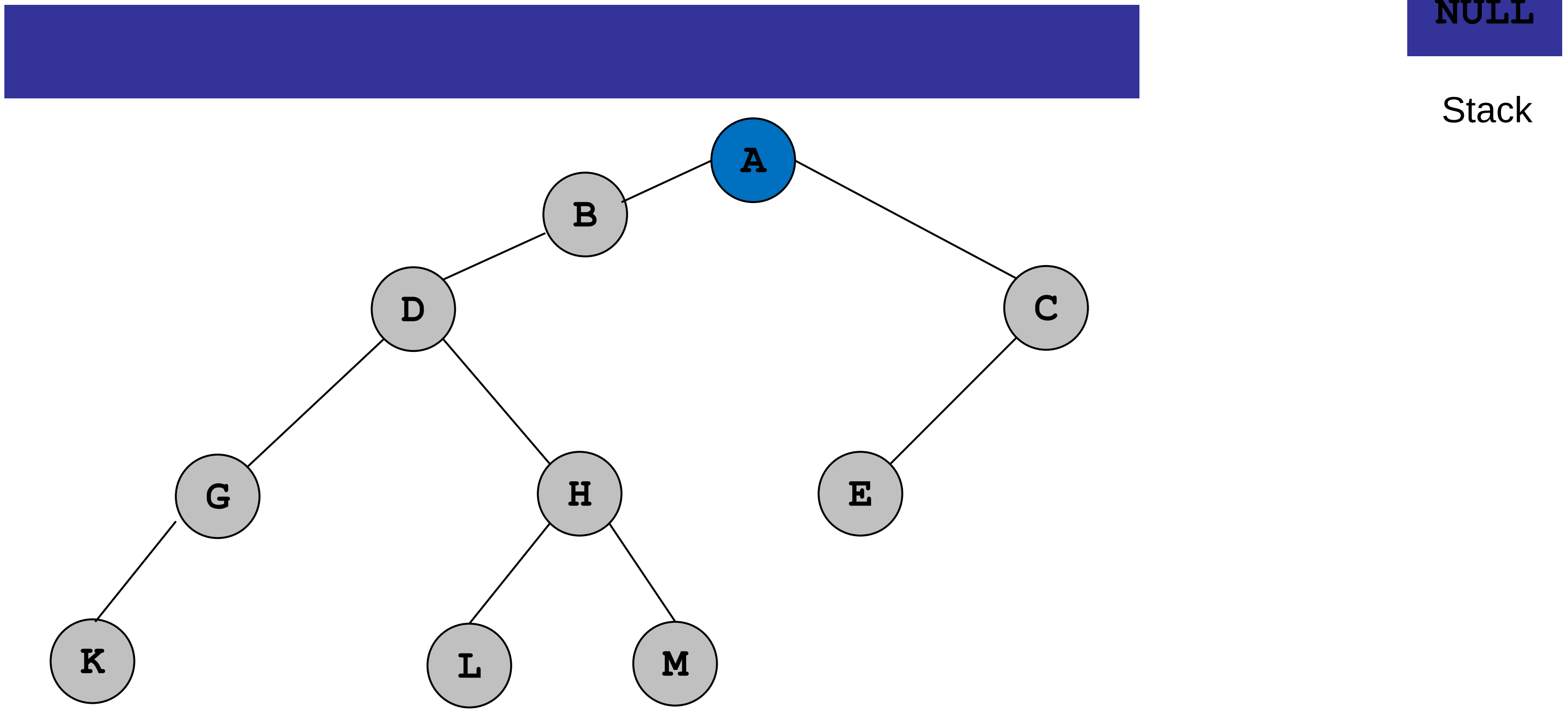
```
public void inorder (BinaryTreeNode<T> r) {
    if (r != null) {
        inorder (r.getLeftChild());
        visit(r);
        inorder (r.getRightChild());
    }
}
```

Traversal Algorithm Postorder

- Initially push NULL onto the stack.

  STACK: NULL

  then set PTR: = A, the root of T



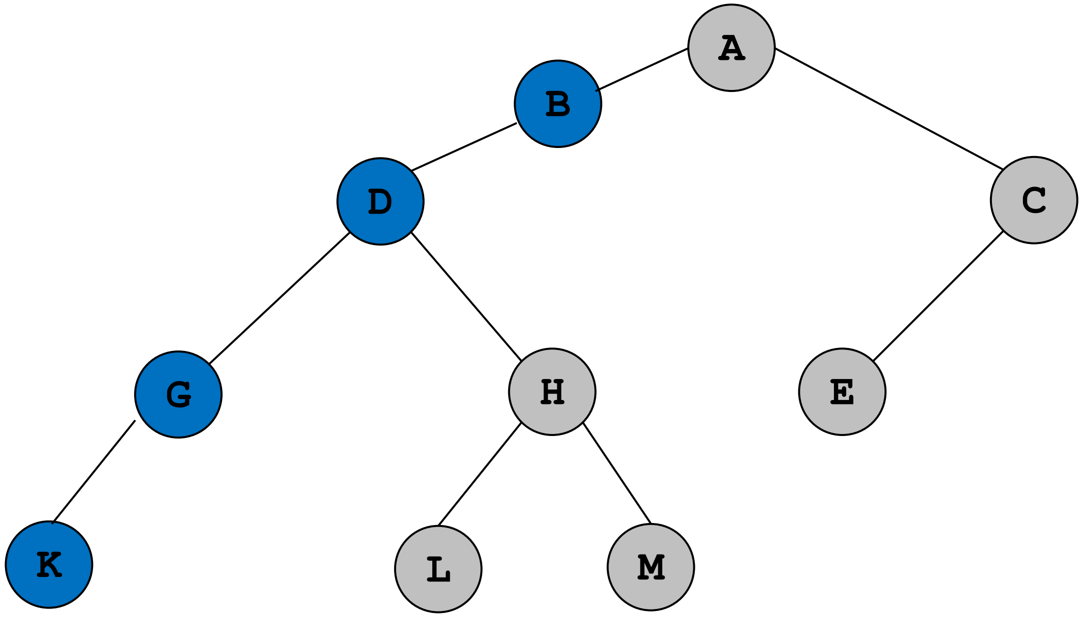
Traversal Algorithm Postorder

- Processed down the left-most path rooted at PTR = A, pushing the nodes A, B, D, G, K onto the stack. Furthermore, since A has write child C push –C onto STACK after A before B. Same for H.

  STACK: NULL, A, -C, B, D, -H, G, K

K
G
-H
D
B
-C
A
NULL

Stack



Traversal Algorithm Postorder

- [Backtracking] Pop and process K, G. Since –H is negative, only pop -H.

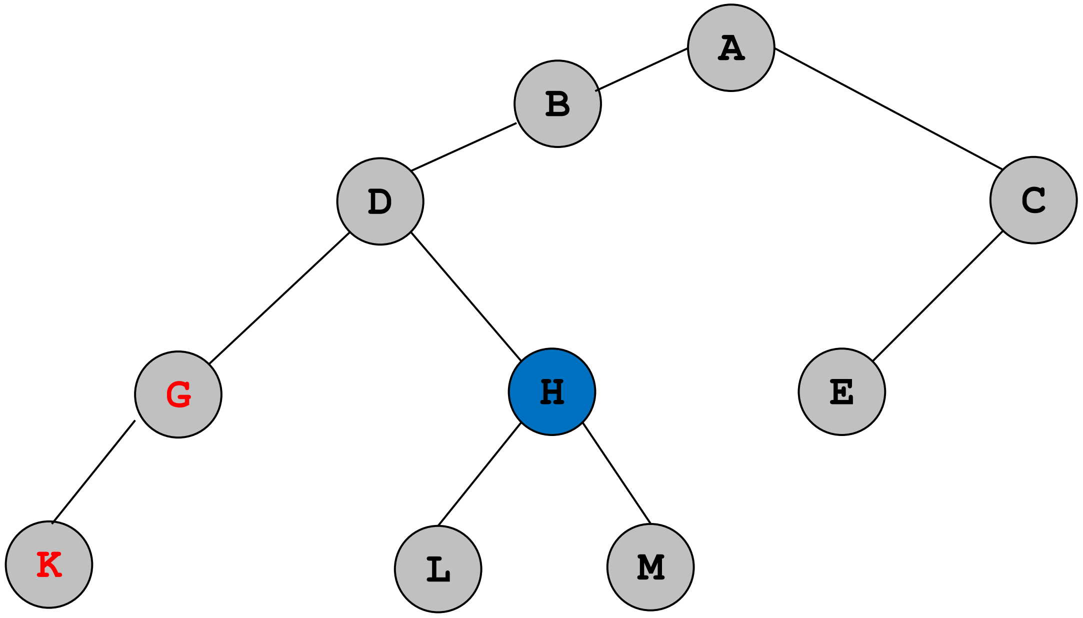
    STACK: NULL, A, -C, B, D

Now PRT:= - H. Reset PRT:=H

K G

D
B
-C
A
NULL

Stack



Traversal Algorithm Postorder

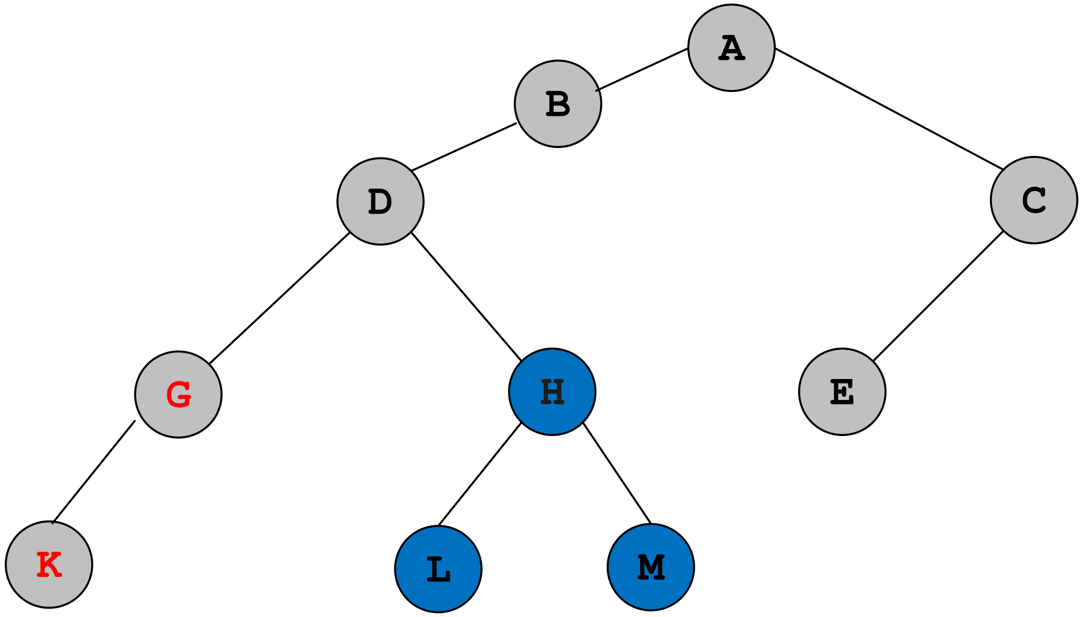
- Processed down the left-most path rooted at PRT:=H, first push H onto STACK. Since H has a right child M, push –M onto STACK after H. Last, push L onto STACK.

  STACK: NULL, A, -C, B, D, H, -M, L

L
-M
H
D
B
-C
A
NULL

Stack

K G



Traversal Algorithm Postorder

- [Backtracking] Pop and process L, only pop -M.

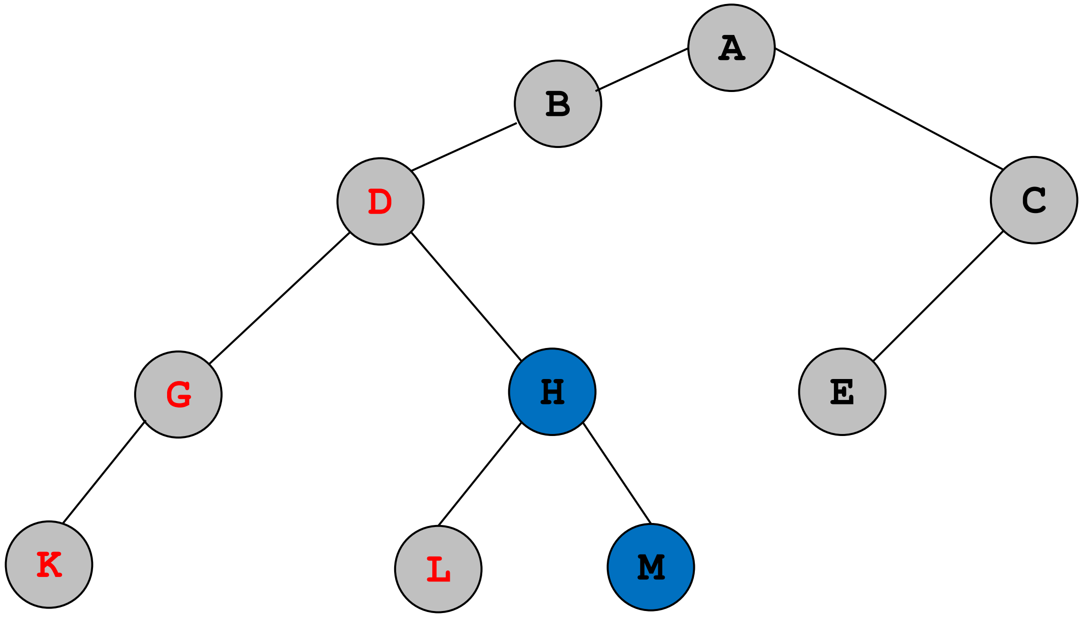
  STACK: NULL, A, -C, B, D, H

  Now PRT:= - M. Reset PRT:=M

K G L

H
D
B
-C
A
NULL

Stack



Traversal Algorithm Postorder

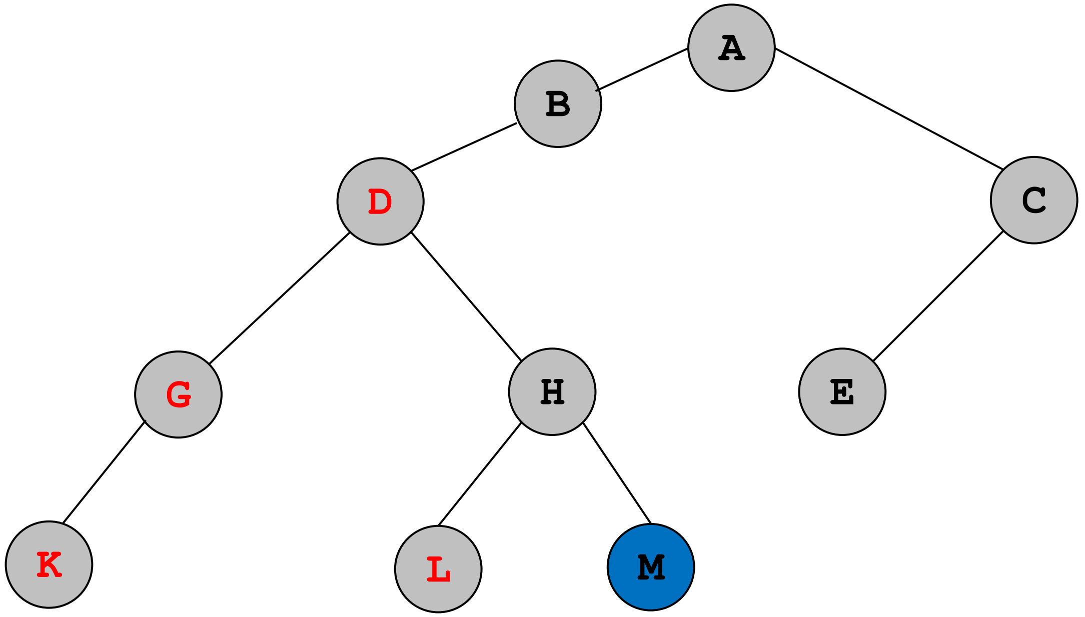
- Processed down the left-most path rooted at PRT:=M, now only M is pushed onto STACK.

  STACK: NULL, A, -C, B, D, H, M

M
H
D
B
-C
A
NULL

Stack

K G L

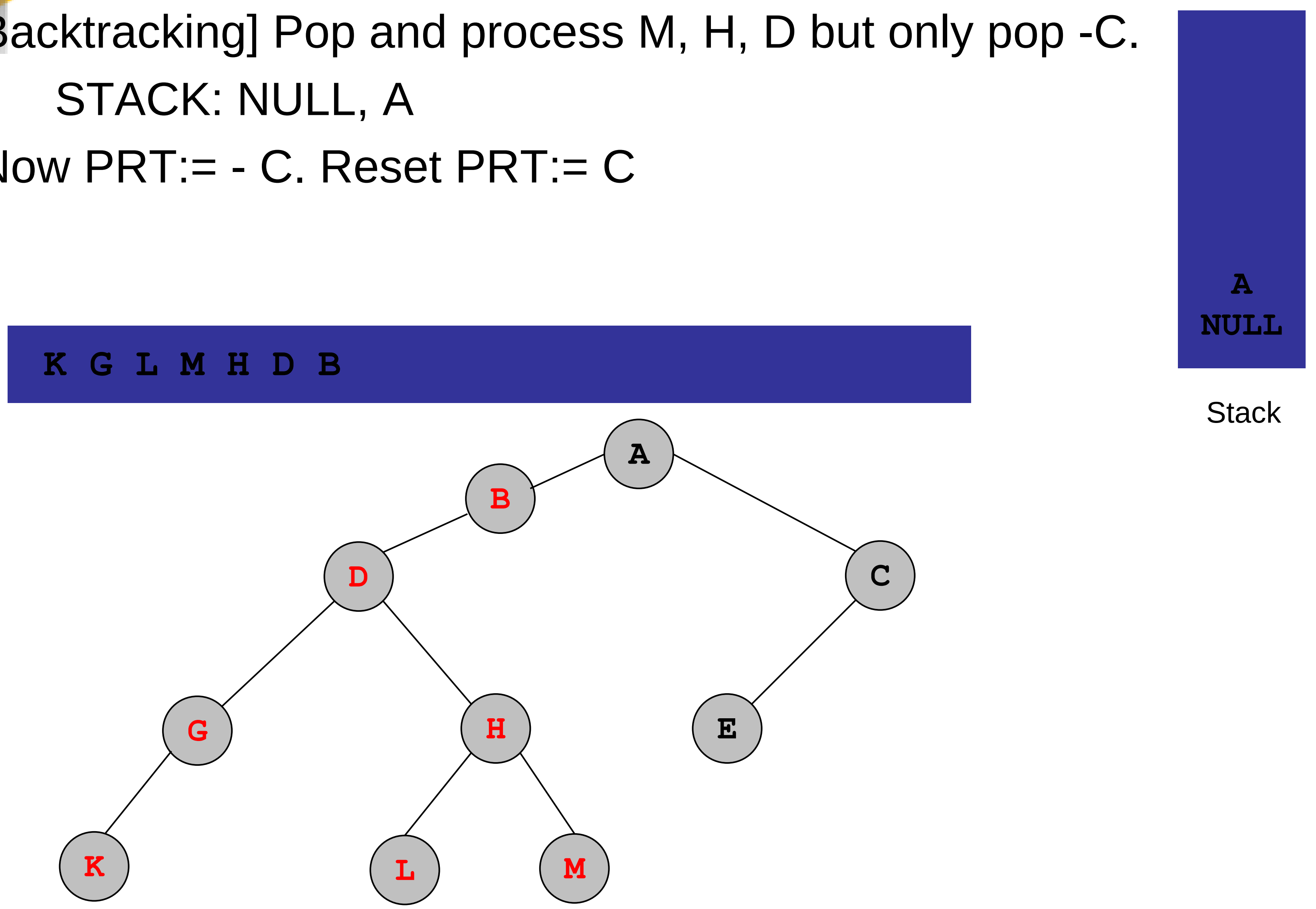


Traversal Algorithm Postorder

- [Backtracking] Pop and process M, H, D but only pop -C.

  STACK: NULL, A

  Now PRT:= - C. Reset PRT:= C



Traversal Algorithm Postorder

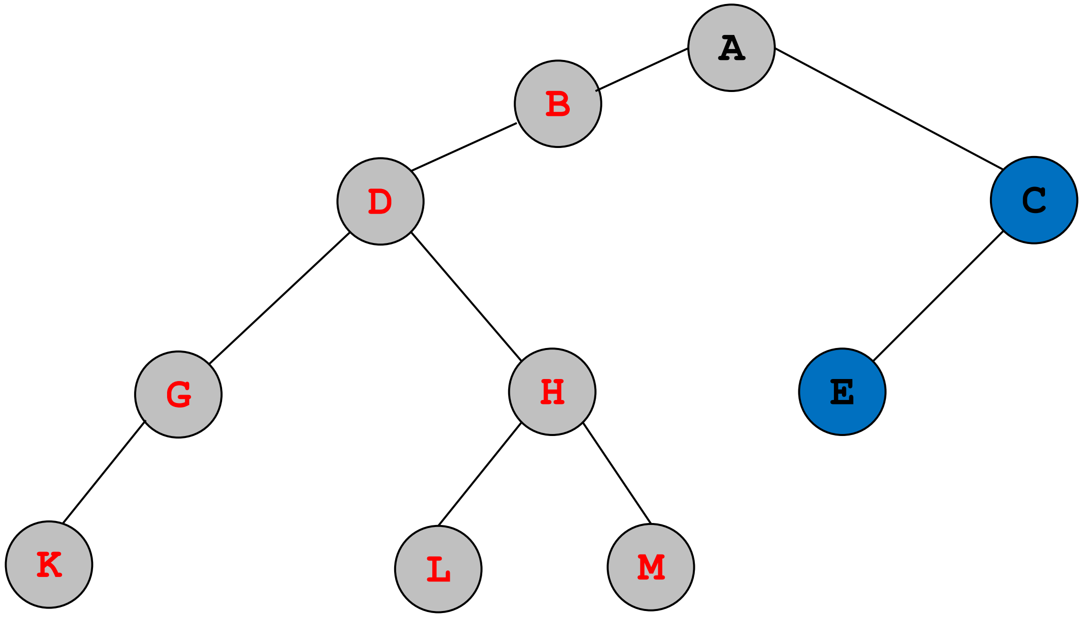
- Processed down the left-most path rooted at PRT:=C, first, C is pushed onto STACK and then E.

  STACK: NULL, A, C, E

K G L M H D B

E
C
A
NULL

Stack

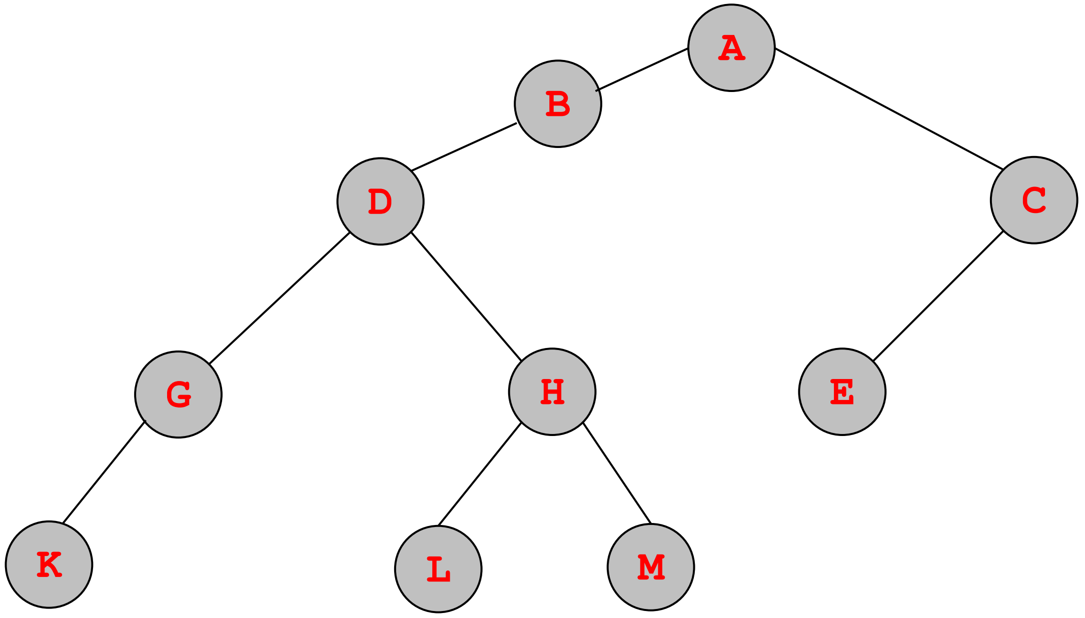


Traversal Algorithm Postorder

- [Backtracking] Pop and process E, C, A. When NULL is popped, STACK is empty and algorithm is completed.

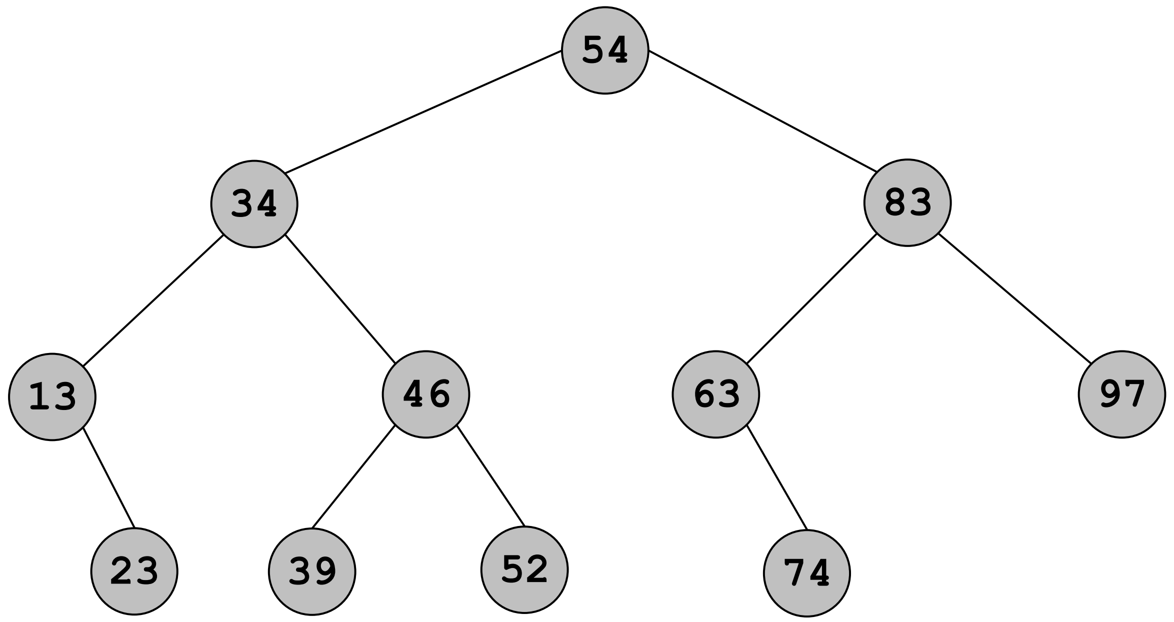
K G L M H D B E C A

Stack



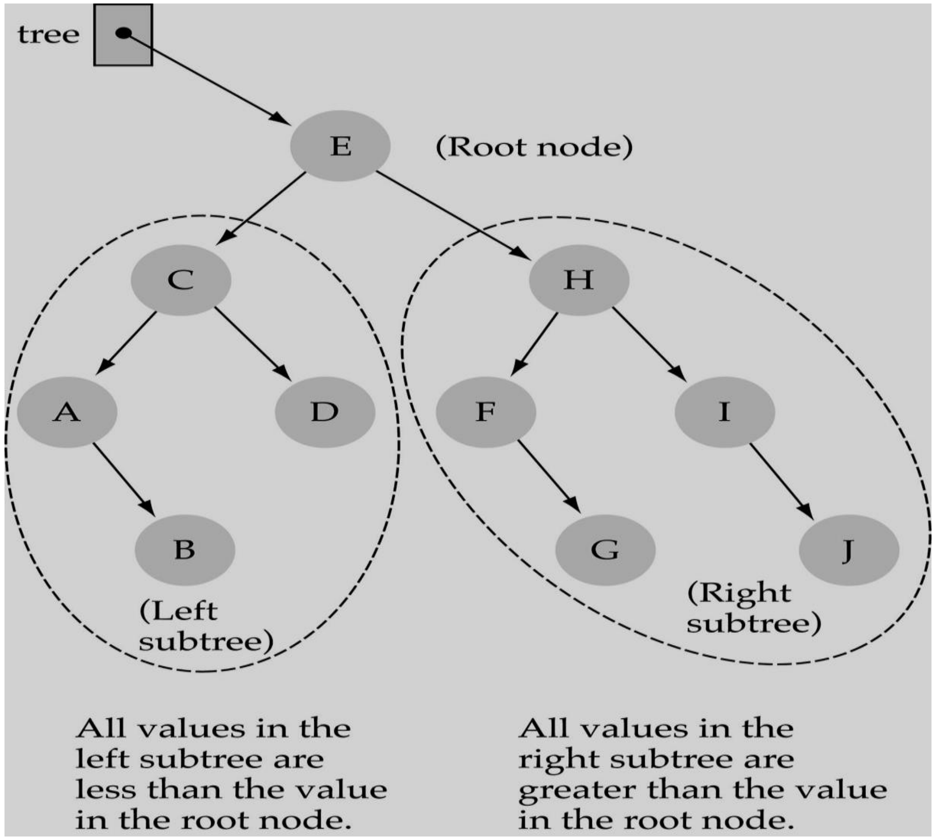
Binary Search Tree

- How to search a binary tree?
  - Start at the root
  - Search the tree level by level, until you find the element you are searching for or you reach a leaf.



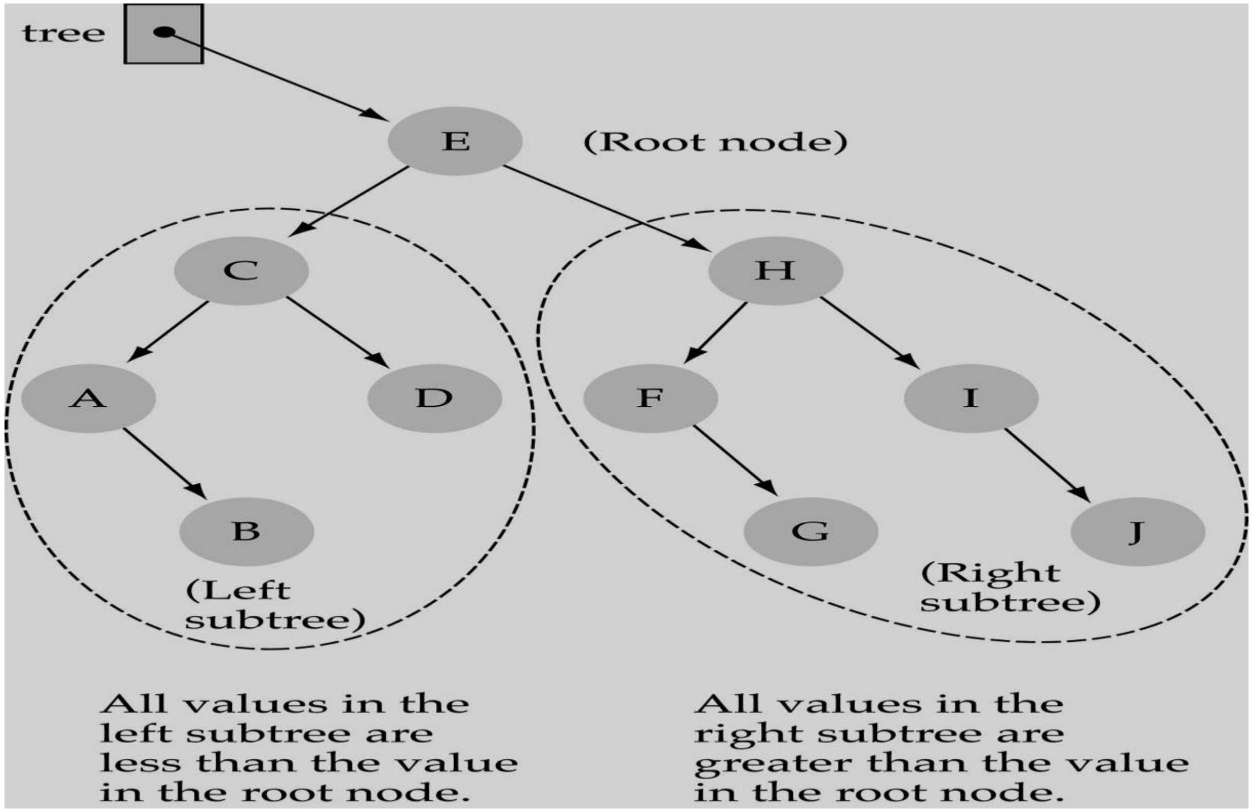
Binary Search Tree

- Binary Search Tree Property:
  - The value stored at a node is greater than the value stored at its left child and less than the value stored at its right child.


tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

Binary Search Tree (BST)

- Binary Search Tree Property:
  - In a BST, the value stored at the root of a sub-tree is greater than any value in its left sub-tree and less than any value in its right sub-tree!


tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

How to search a binary search tree?

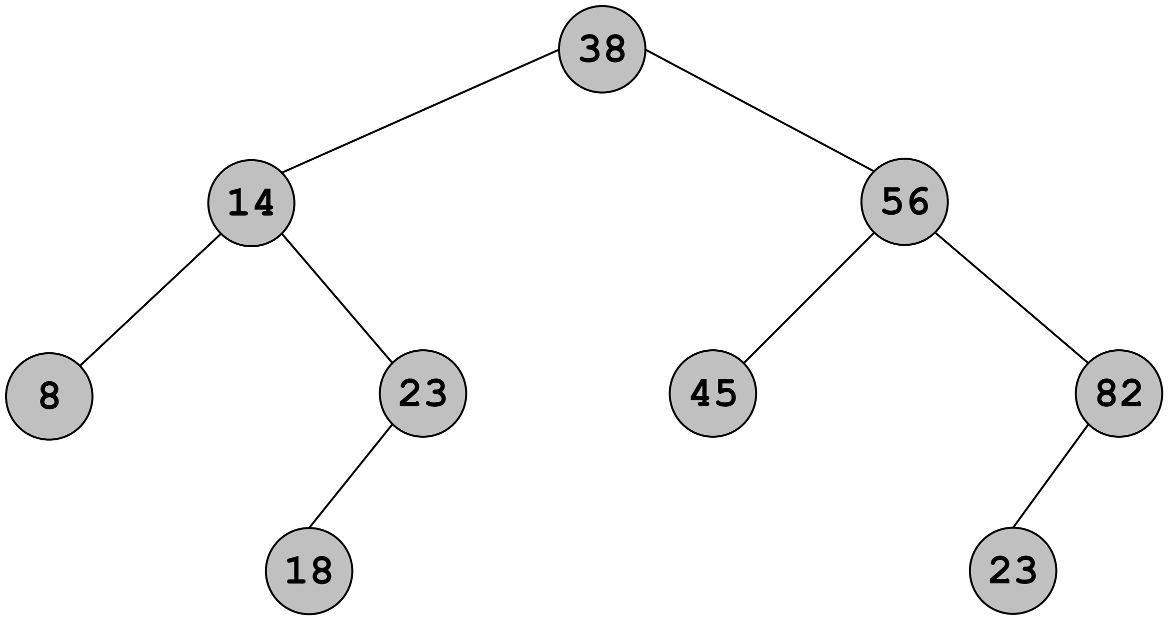
1. Start at the root node (N)
2. Compare the value of the item you are searching for (ITEM) with the value stored at the root
   - If ITEM < N, proceed to the left child of N
   - If ITEM > N, proceed to the right child of N
3. Repeat step 2 until one of the following occurs:
   - we meet a node N such that ITEM = N. In this case search is successful.
   - We meet an empty sub tree, which indicates that the search is unsuccessful.

- Is this better than searching a linked list?

  Yes !! ---> O(logN)

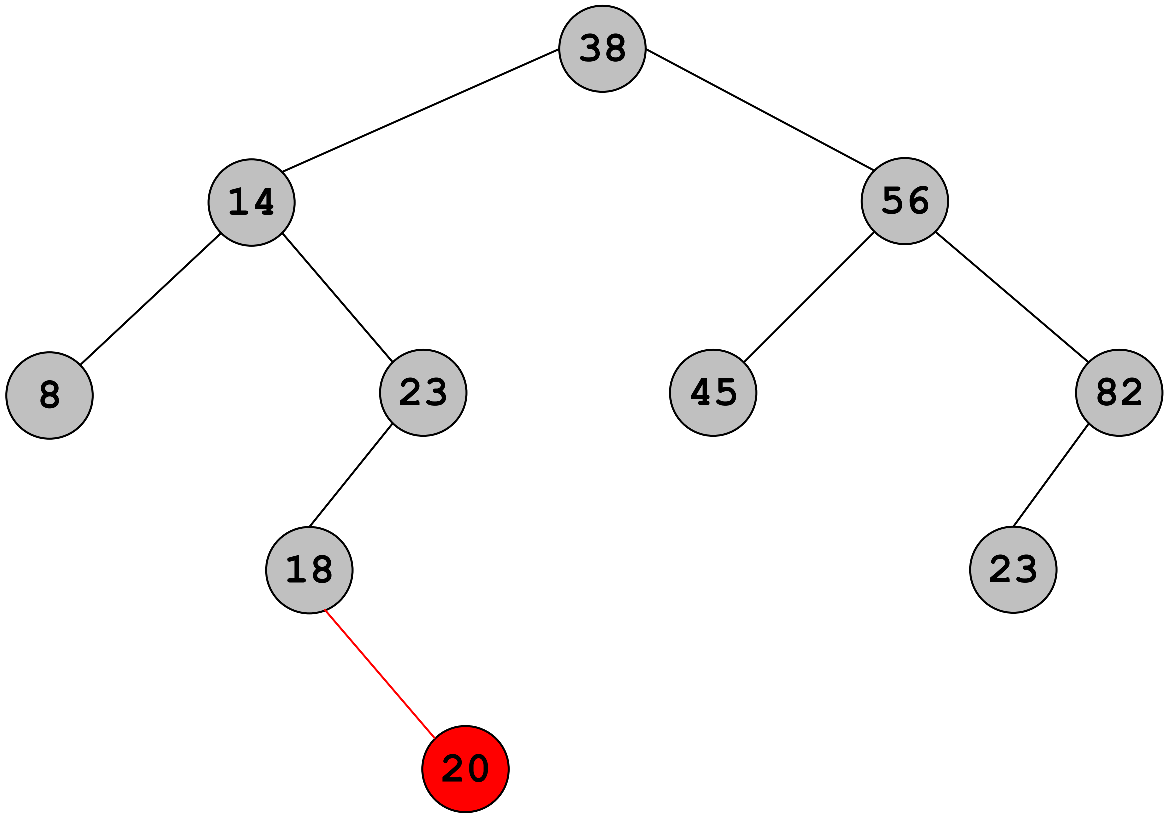
How to search a binary search tree?

- Suppose ITEM = 20 is given. For searching we get:
  - Compare ITEM = 20 with the root, 38. 20<38, proceed to the left child of 38 which is 14.
  - Compare ITEM = 20 with 14. 20>14, proceed to the right child of 14 which is 23.
  - Compare ITEM = 20 with 23. 20<23, proceed to the left child of 23 which is 18.



How to search a binary search tree?

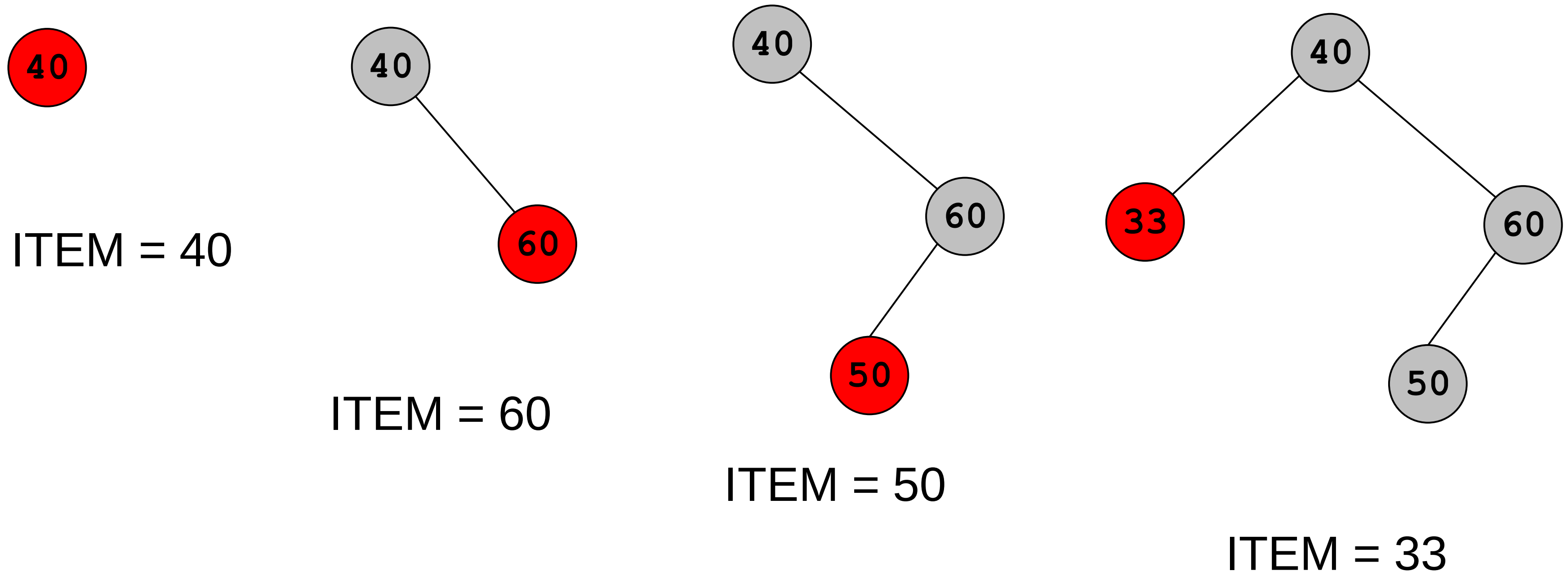
- Suppose ITEM = 20 is given. For searching we get:
  - Compare ITEM = 20 with 18. 20>18, and 18 does not have a right child, insert 20 as the right child of 18.



How to search a binary search tree?

- Suppose the following six numbers are inserted in order into an empty binary search tree:

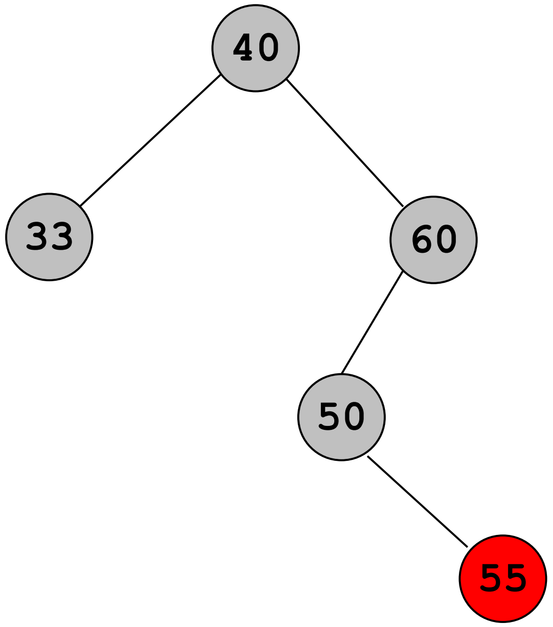
40, 60, 50, 33, 55, 11



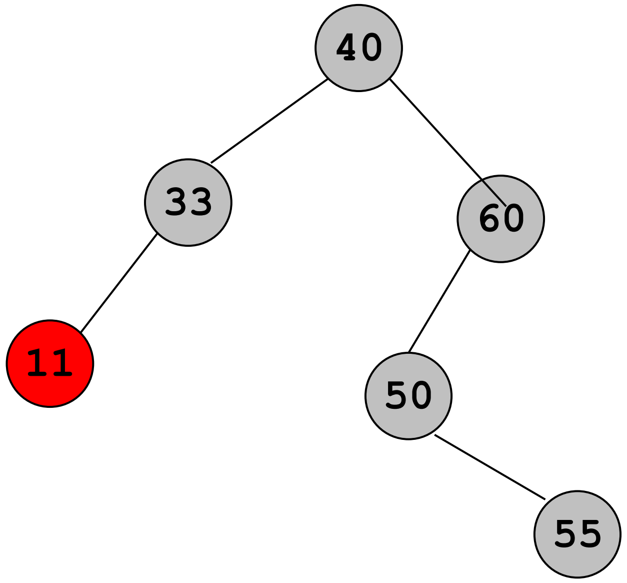
How to search a binary search tree?

- Suppose the following six numbers are inserted in order into an empty binary search tree:

40, 60, 50, 33, 55, 11



ITEM = 55



ITEM = 11

Search Algorithm in a Binary Search Tree

- FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

A binary search tree T is in memory and ITEM is given. Find the location LOC of ITEM in tree and also the location PAR of the parent of ITEM. There are three special cases:

i. LOC = NULL and PAR = NULL will indicate that the tree is empty

ii. LOC ≠ NULL and PAR = NULL will indicate that ITEM is the root of T

iii. LOC = NULL and PAR ≠ NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.

Search Algorithm in a Binary Search Tree

- FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

```
1.  [Tree empty?]
      If ROOT = NULL then: set LOC := NULL and PAR := NULL and RETURN
2.  [ITEM at root?]
      If ITEM=INFO[ROOT] then: set LOC:=ROOT and PAR:=NULL and RETURN
3.  [initialize pointer PTR and SAVE]
      If ITEM < INFO[ROOT] then: set PTR := LEFT[ROOT] and SAVE := ROOT
       Else: set PTR := RIGHT[ROOT] and SAVE := ROOT
          [End of If structure]
4.  Repeat step 5 and 6 while PTR ≠ NULL
5.  [ITEM found?]
      If ITEM = INFO[ROOT] then: set LOC := PTR, PTR := SAVE and RETURN
6.  If ITEM < INFO[ROOT] then: set SAVE := PTR and PTR := LEFT[PTR]
      Else: set SAVE := PTR and PTR := RIGHT[PTR] [End of If structure]
       [End of step 4 loop]
7.  [search unsuccessful] Set LOC := NULL and PTR := SAVE
8.  Exit
```

Insert Algorithm in a Binary Search Tree

- INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

```
1. Call  FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. If LOC := NULL and Exit
3. [copy ITEM into new node in AVAIL list]
   a) If AVAIL = NULL then: Write: OVERFLOW and Exit
   b) Set NEW := AVAIL, AVAIL := LEFT[AVAIL] and INFO[NEW] := ITEM
   c) Set LOC := NEW, LEFT[NEW] := NULL and RIGHT[NEW] := NULL
      [End of If structure]
4. [add ITEM to tree]
   If PAR = NULL then: set ROOT := NEW
   Else if ITEM < INFO[PAR] then: set LEFT[PAR] := NEW
   Else: set RIGHT[PAR] := NEW
   [End of If structure]
5. Exit
```

Deletion in Binary Search Tree

- First, find the item; then, delete it
- Binary search tree property must be preserved!!
- We need to consider three different cases:

  (1) Deleting a leaf

  (2) Deleting a node with only one child

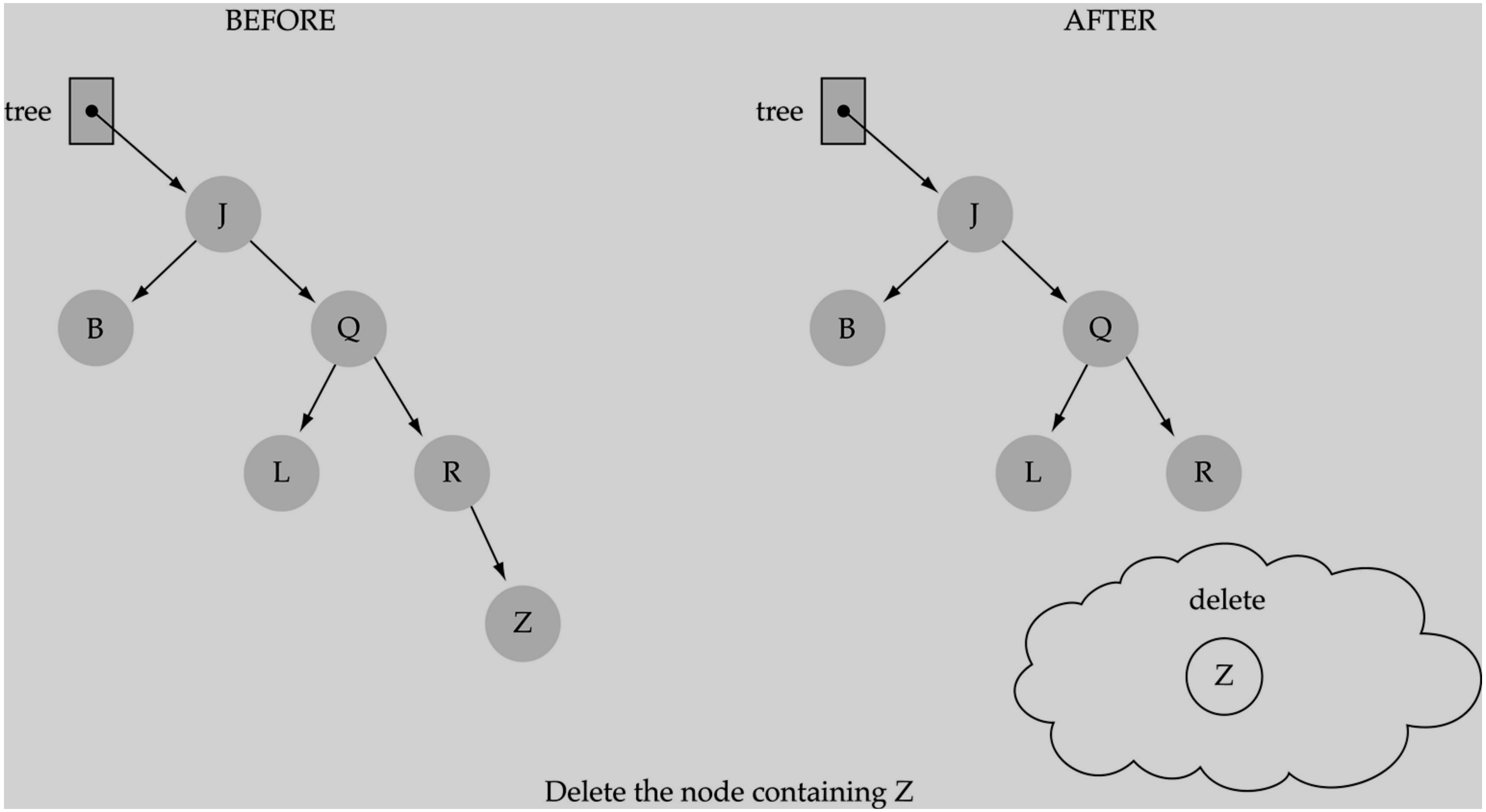
  (3) Deleting a node with two children

Deletion in Binary Search Tree

- Suppose T is a binary search tree and an ITEM of information is given. The deletion algorithm first use procedure to find the location of the node N which contains ITEM and also the location of the parent node P(N). N is deleted from the tree depends primarily on the number of children of node N. There are three cases:
  - **Case 1**: N has no child. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the NULL pointer.
  - **Case 2:** N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.
  - **Case 3:** N has two children. S(N) denote the inorder successor of N. Then N is deleted from T by first deleting S(N) from T and then replacing node N in T by S(N).

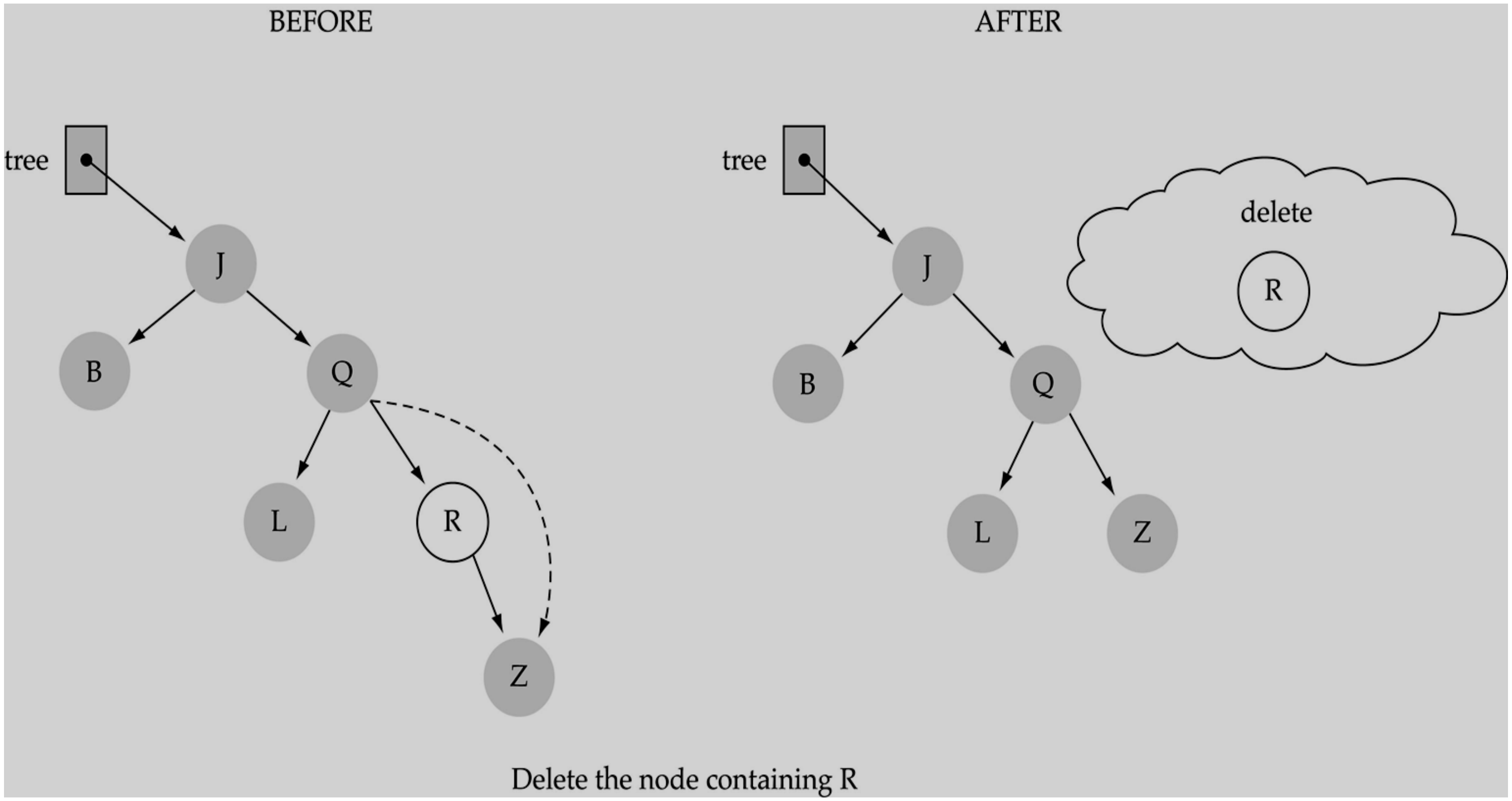
Deletion in Binary Search Tree

- Deleting a Leaf (first case)



Deletion in Binary Search Tree

- Deleting a node with only one child (second case)



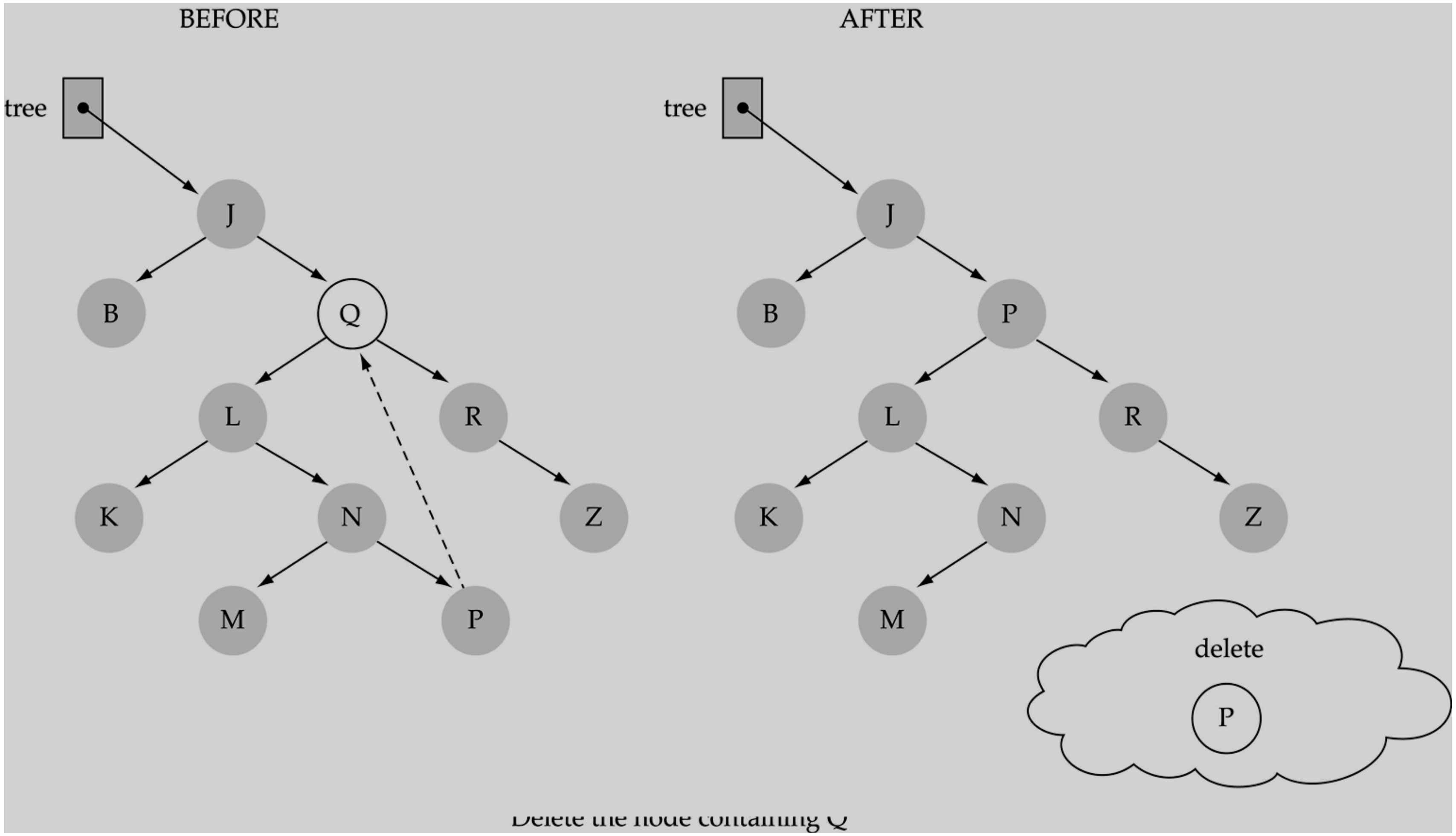
Delete the node containing R

Deletion in Binary Search Tree

- Deleting a node with two children (third case)
    - Find predecessor (i.e., rightmost node in the left sub-tree)
    - Replace the data of the node to be deleted with predecessor's data
    - Delete predecessor node

Deletion in Binary Search Tree

- Deleting a node with two children (third case)



Delete the node containing Q

Deletion in Binary Search Tree Algorithm

- CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

Delete the node N at location LOC, where N does not have two children. Pointer PAR gives the location of the parent of N, or PAR = NULL means N is root. Pointer CHILD gives the location of only child of N, or else CHILD = NULL means N no child.

```
1.  If LEFT[LOC] = NULL and RIGHT[LOC] = NULL then: [initialize CHILD]
            set CHILD := NULL
    Else if LEFT[LOC] ≠ NULL then: set CHILD := LEFT[LOC]
    Else set CHILD := RIGHT[LOC]
    [End of If structure]
2.  If PAR ≠ NULL then:
            If LOC = LEFT[PAR] set LEFT[PAR] := CHILD
    Else set RIGHT[PAR] := CHILD        [End of If structure]
    Else set ROOT := CHILD
    [End of If structure]
3.  Return
```

Deletion in Binary Search Tree Algorithm

- CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

N has two children. Pointer PAR gives the location of the parent of N, or PAR = NULL means N is root. Pointer SUC gives the location of inorder successor of N, and PARSUC gives the location of the parent of inorder successor.

```
1.  [find SUC and PARSUC]
    a)  set PTR := RIGHT[LOC] and SAVE := LOC
    b)  Repeat while LEFT[PTR] ≠ NULL
            set SAVE := PTR and PTR := LEFT[PTR]    [End of loop]
    c)  Set SUC := PTR and PARSUC := SAVE
2.  Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)
3.  [Replace node N by its inorder successor]
    a)  If PTR ≠ NULL then:
            if LOC : = LEFT[PTR] then: set LEFT[PTR]: = SUC
            Else: set RIGHT[PTR]: = SUC    [End of If structure]
        Else set ROOT := SUC    [End of If structure]
    b)  set LEFT[SUC] := LEFT[LOC] and RIGHT[SUC] := RIGHT[LOC]
4.  Return
```

Deletion in Binary Search Tree Algorithm

- DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

A binary search tree T is in memory and ITEM is given. Delete ITEM from the tree.

```
1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR) [find the
   location of ITEM and its parent in the tree]
2. [ITEM in tree?]
    If LOC = NULL, then: Write: ITEM  not in tree and EXIT
3. [Delete note containing ITEM]
   If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL then:
            Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
   Else: Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
   [End of If structure]
4. [Return deleted node to the AVAIL list]
    Set LEFT[LOC] := AVAIL and AVAIL := LOC
5. Return
```

m-Way Search Tree

- Each internal node of a multi-way search tree T:
  - has at least two children
  - contains d - 1 items, where d is the number of children → d-nodes
  - “contains” 2 pseudo-items: $\mathbf{k_0} = -\infty$, $\mathbf{k_d} = \infty$
- Children of each internal node are “between” items
- all keys in the subtree rooted at the child fall between keys of those items

m-Way Search Tree

- An M-way search tree:
  - Between 1 to (M-1) values in each node
  - At most M children per node
- An m-way search tree, either is empty or satisfies the following properties:
  - For some integer m, known as order of the tree, each node is of degree which can reach a maximum of m, in other orders each node has al most m child nodes. A node may be represent as A_0, (K_1, A_1), (K_2, A_2), …, (K_{m-1}, A_{m-1})

    where each A_i, $0 \leq i \leq m-1$, is a pointer to a sub-tree, and each K_i, $1 \leq i \leq m-1$, is a key value.

m-Way Search Tree

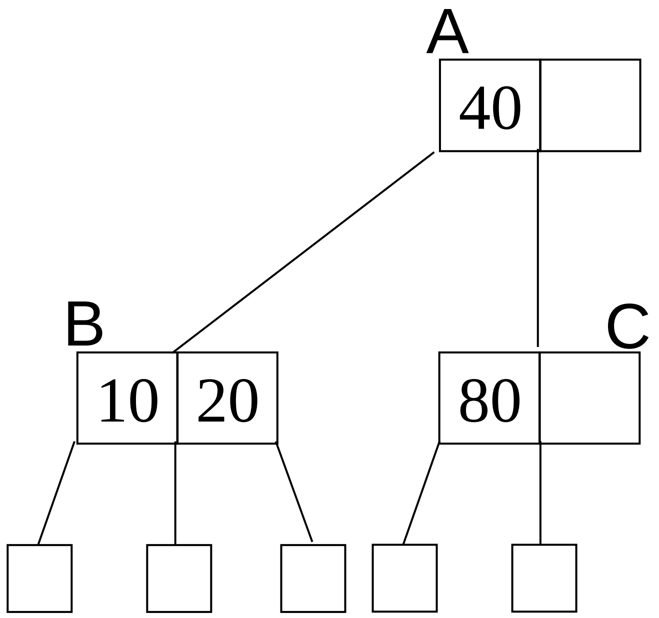
- An m-way search tree, either is empty or satisfies the following properties:
  - If a node has k child nodes where k ≤ m, then the node can have only (k-1) keys K_1, K_2, … , K_{k-1} , contained on the node such that $K_i < K_{i+1}$ and each of keys partitions all the keys in the sub-trees into k subsets.
  - For a node A_0, (K_1, A_1), (K_2, A_2), …, (K_{m-1}, A_{m-1}), all key values in the sub-tree pointed to by A_i are less than the key K_{i+1}, $0 \le i \le m-2$, and all key values in the sub-tree pointed to by A_{m-1} are greater than K_{m-1}.
  - Each of the sub-tree A_i, $0 \le i \le m-1$, are also m-way search tree.

B Tree

- A B tree of order m, if non empty, is a m-way tree in which:
  - The root has at least two child nodes and at most m child nodes
  - The internal nodes except the root have at least $\left\lceil \frac{m}{2} \right\rceil$ child nodes and at most m child nodes
  - The number of keys in each internal node is one less than the number of child nodes and these keys partition the keys in the sub-trees of the node in a manner similar to that of m-way search tree.
  - All leaf nodes are on the same level.

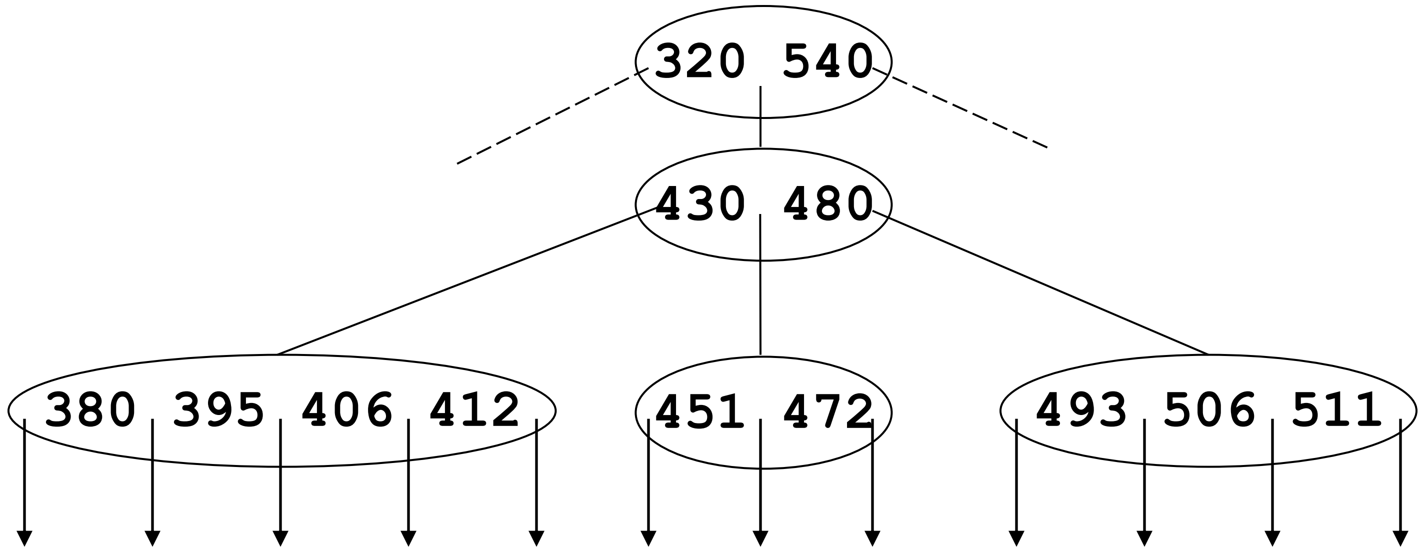
B Tree

2-3 tree: B-tree of order 3



B Tree

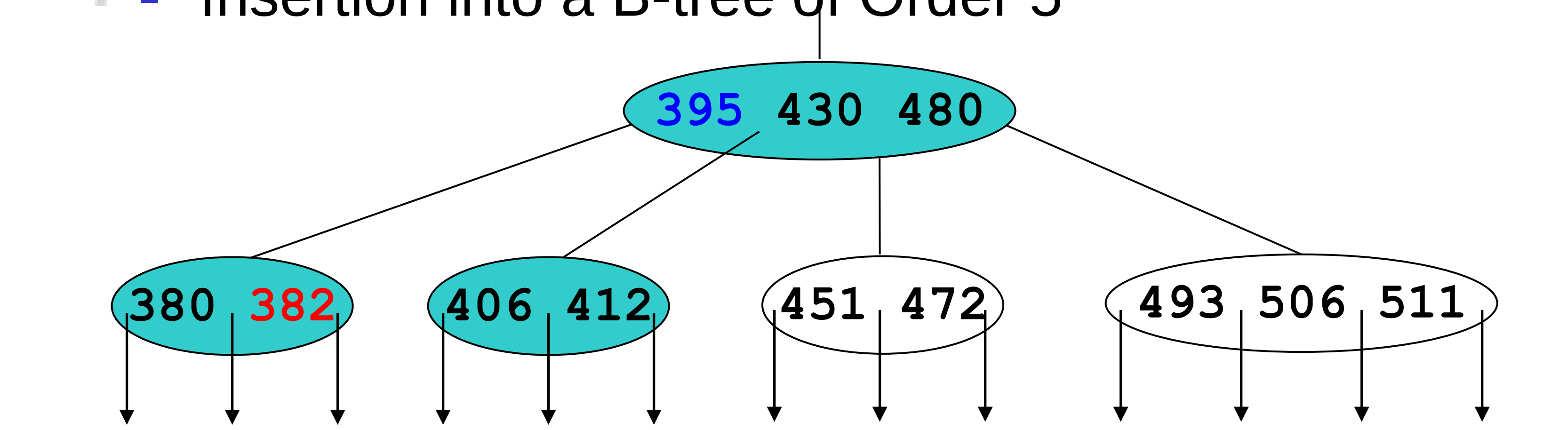
- Insertion into a B-tree of Order 5



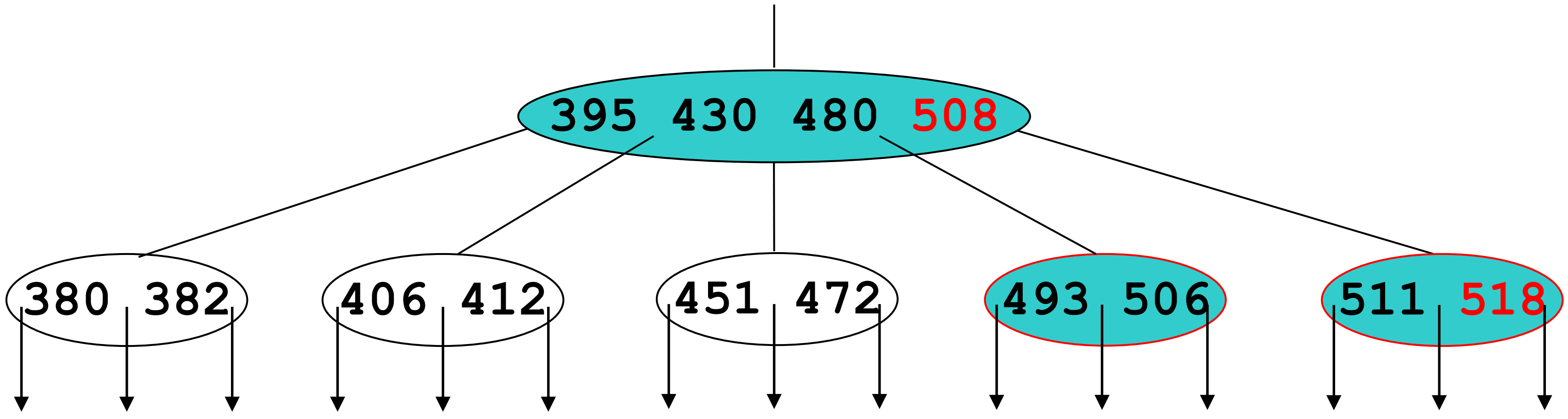
(a) Initial portion of a B-tree

B Tree

- Insertion into a B-tree of Order 5



(b) After inserting 382 (Split the full node)



(c) After inserting 518 and 508

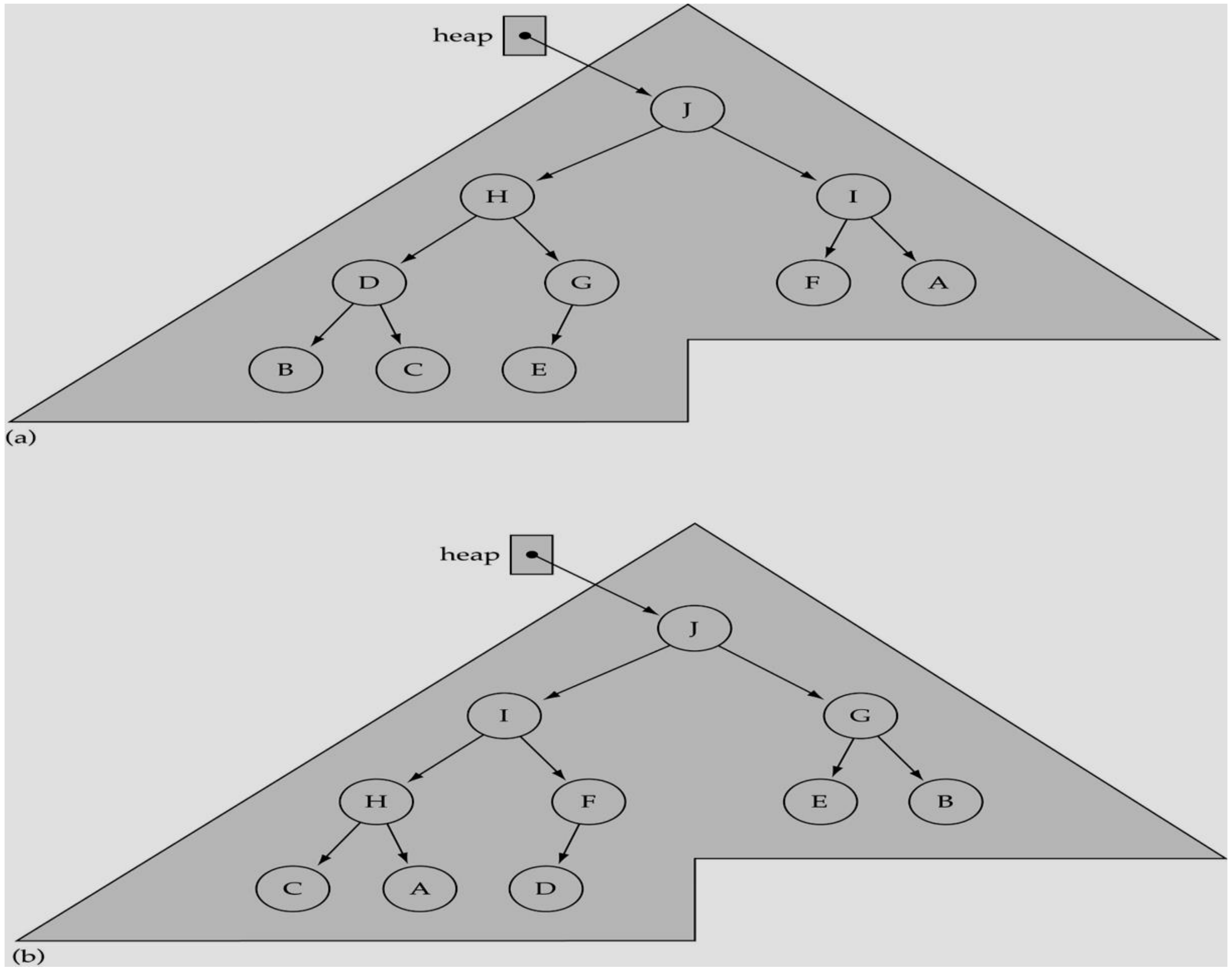
What is a heap?

- A heap tree is a complete binary tree in which data values stored in any node is greater than or equal to the value of search of its children(if any).
- It is a binary tree with the following properties:
  - Property 1: it is a complete binary tree
  - Property 2: the value stored at a node is greater or equal to the values stored at the children (heap property)

What is a heap?

- There is only restriction between parent and its children and not within the left or right sub-tree as in case of binary search trees.
- The value stored in the root node of a heap tree is always the largest value in the tree. Such a heap tree is called a max-heap.
- We can also choose to orient a heap tree in such a way that data values stored in any node is less than or equal to the value of that node's children.
- In min-heap tree, the value stored in the root is guaranteed to be the smallest.

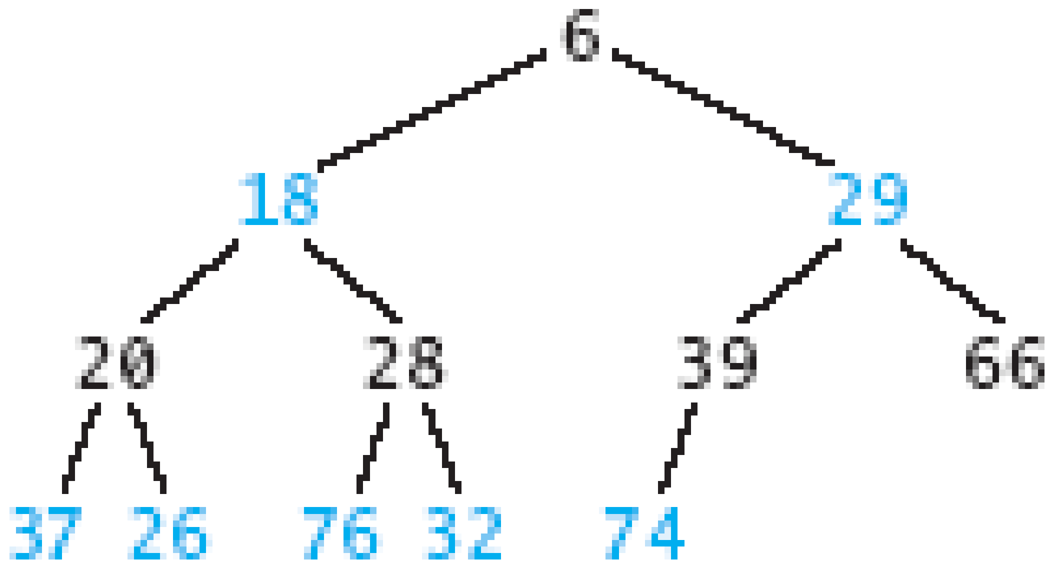
What is a heap?



What is a heap?

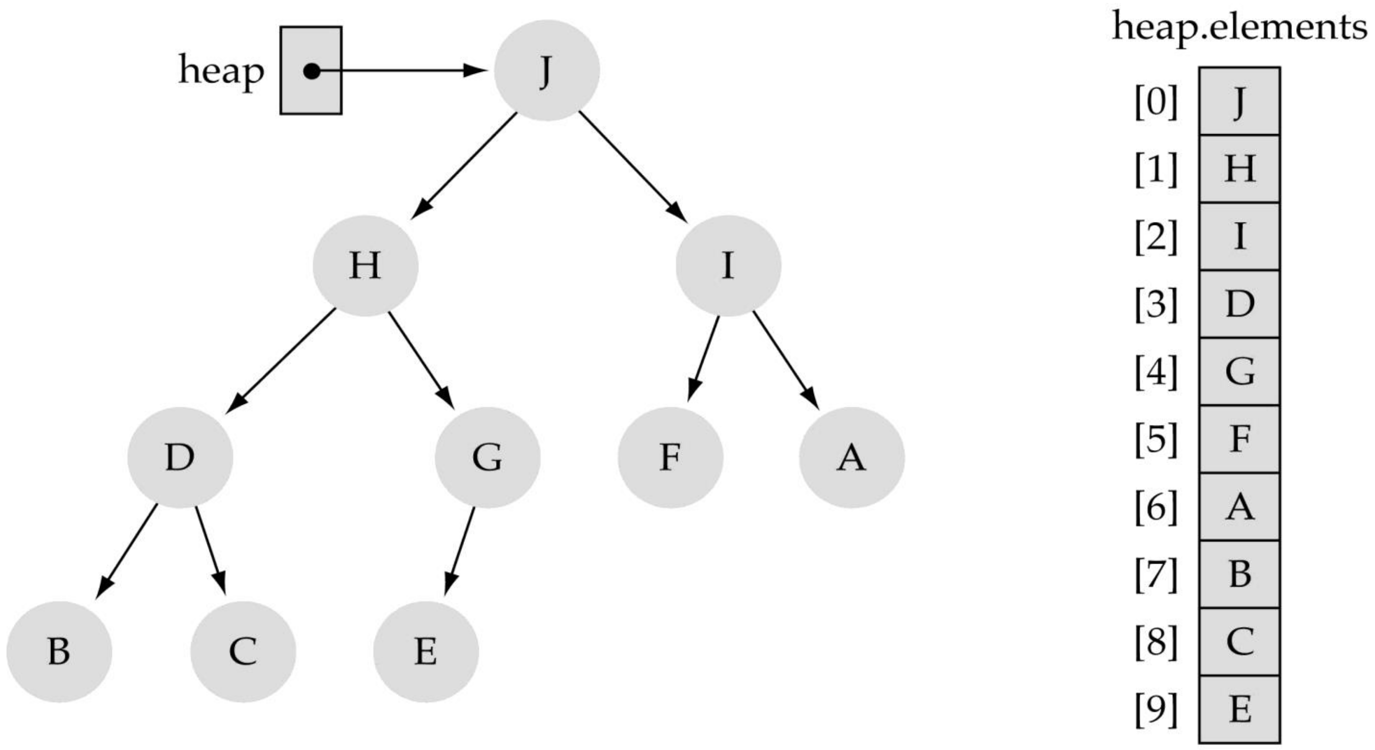
FIGURE 8.20

Example of a Heap



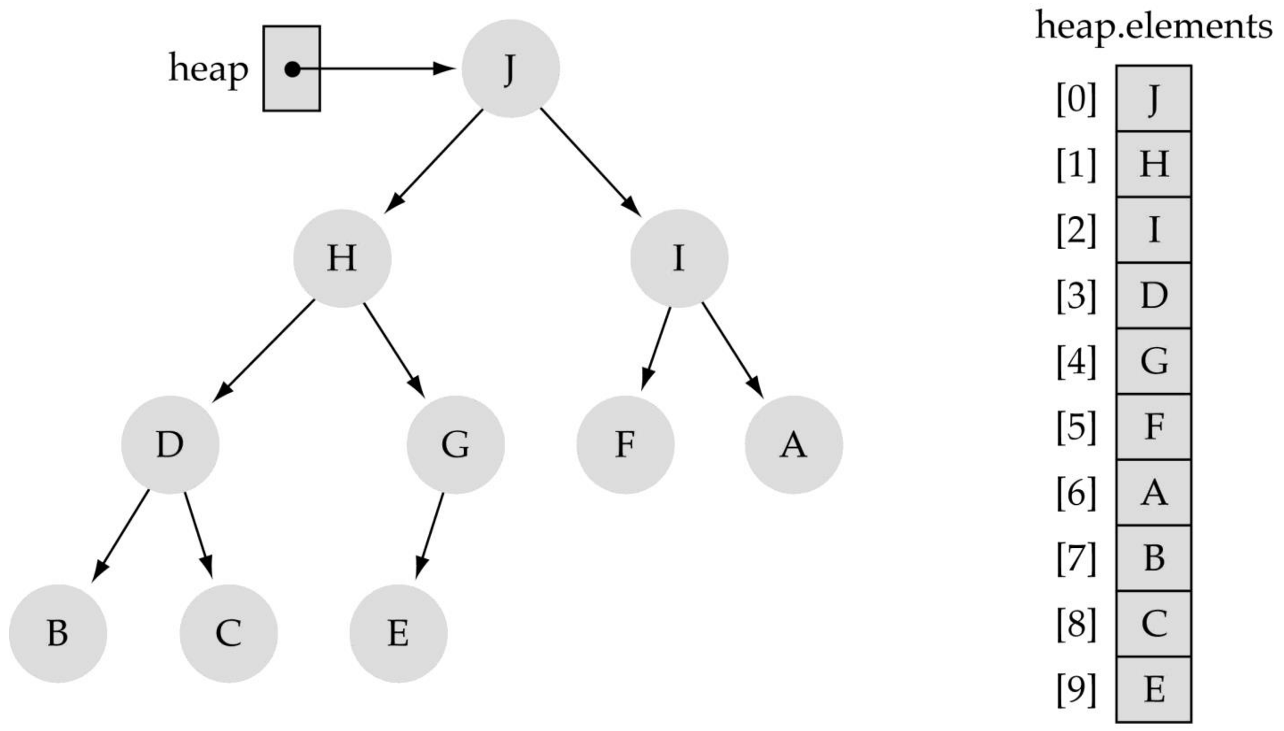
Largest heap element

- From Property 2, the largest value of the heap is always stored at the root



Heap Implementation using Array Representation

- A heap is a complete binary tree, so it is easy to be implemented using an array representation



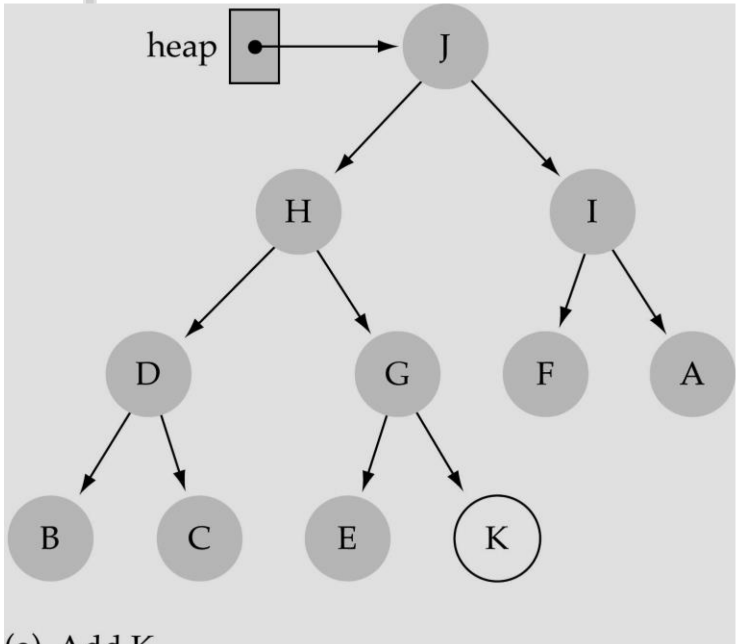
Inserting into a Heap

- Inserting an item into a heap tree is relatively straight forward.
- We begin the insertion by first placing the node containing the given item immediately after the last node in the tree.
- It is done so that the new tree still remains a complete binary tree. Although it satisfies the structural property but may violate the ordering property of heap tree.
- If so, the new node is bubbled up the tree by exchanging the child and parent node that are out of order. We repeat the process until either new node reaches its correct location or root of heap is reached.
- This operation that repairs the heap tree structure so that the new node is bubbled up to its correct position is the reheap up operation

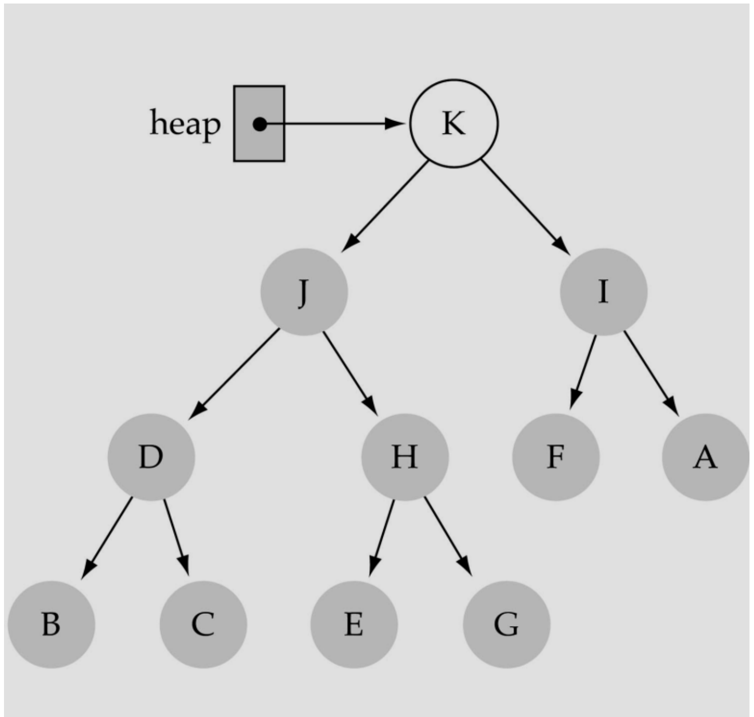
Inserting into a Heap

- Suppose H is a heap with N elements and an ITEM of information is given. We insert ITEM into the heap as follows:
  1) First adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.
  2) Then let ITEM rise to its “appropriate place” in H so that H is finally a heap.

Inserting into a Heap

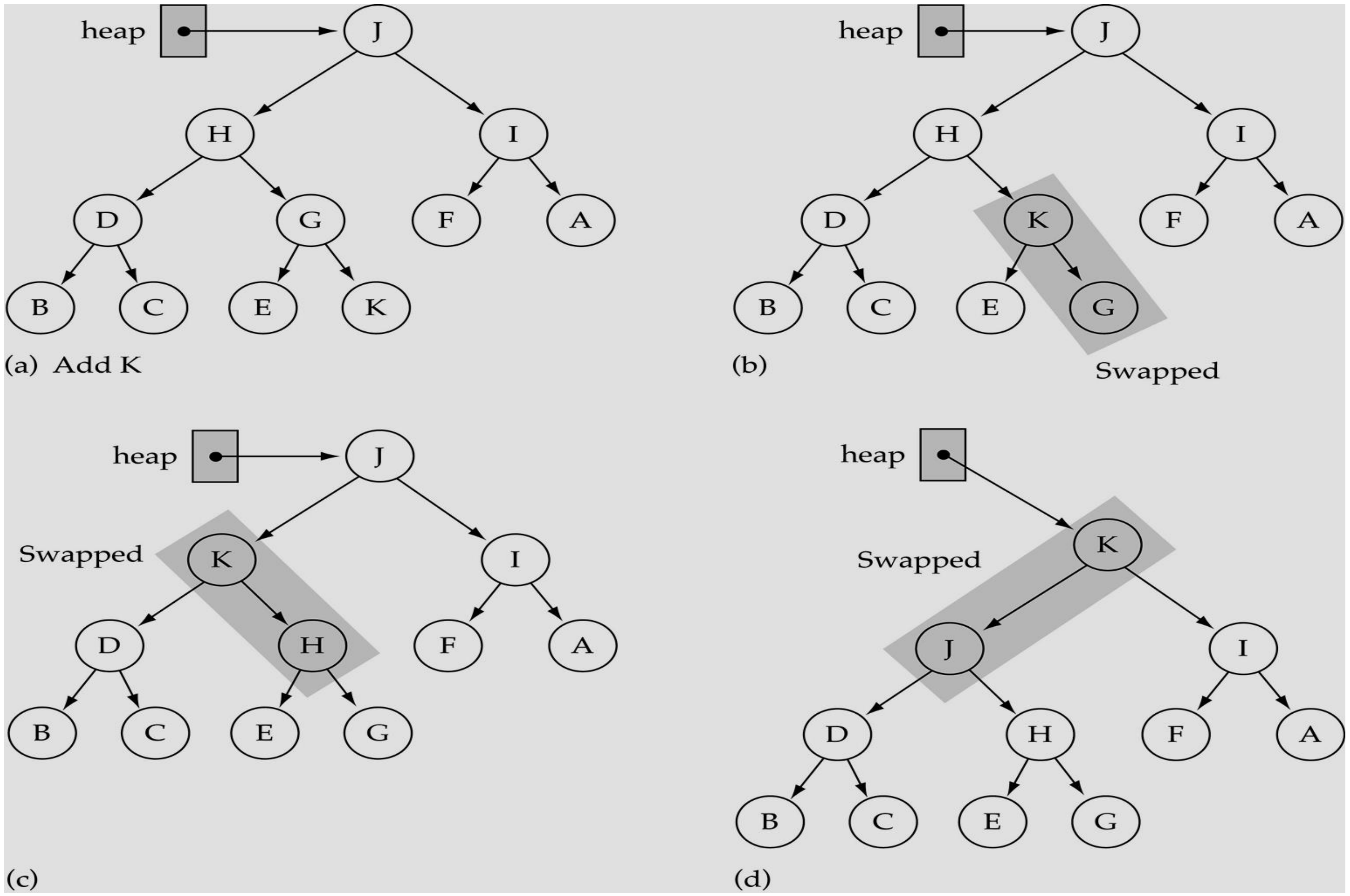


(a) Add K



(b) ReheapUp

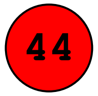
Inserting into a Heap



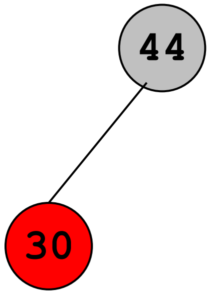
Inserting into a Heap

- Suppose we want to built a heap from the following list of numbers:

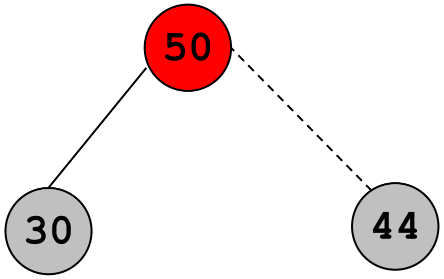
44, 30, 50, 22, 60, 55, 77, 55



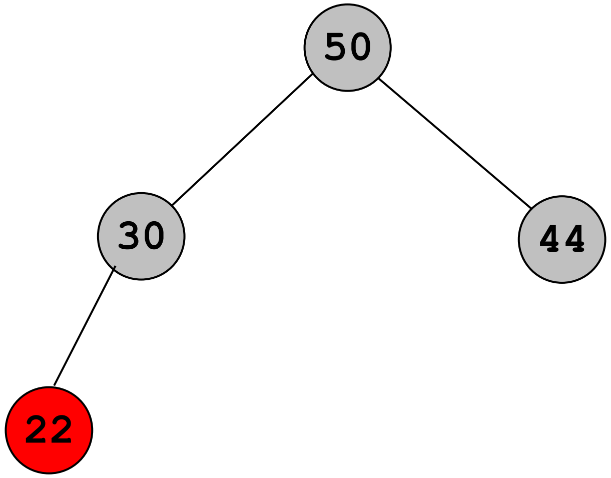
ITEM = 44



ITEM = 30



ITEM = 50

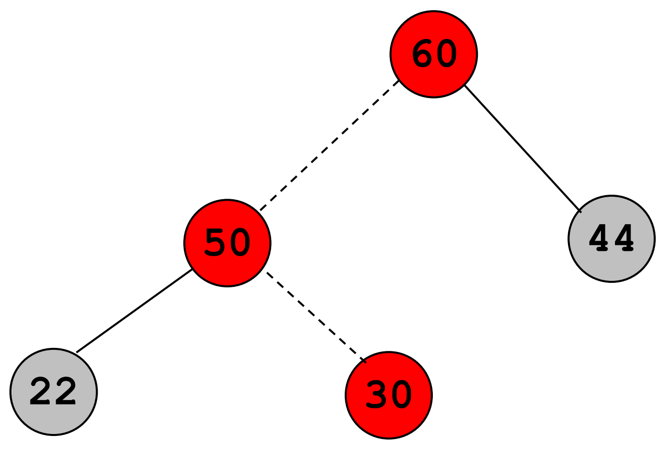


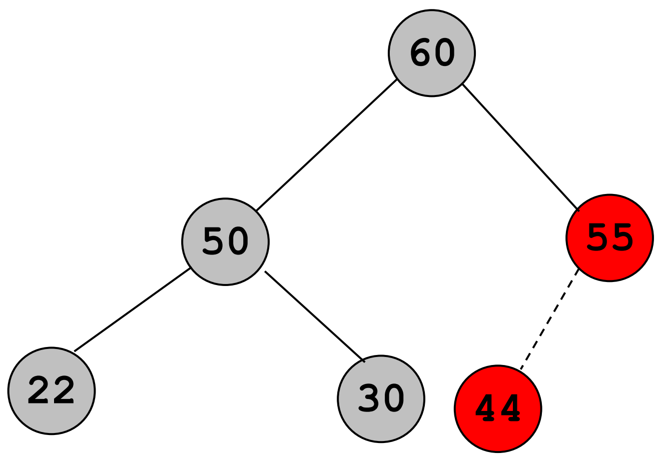
ITEM = 22

Inserting into a Heap

- Suppose we want to built a heap from the following list of numbers:

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





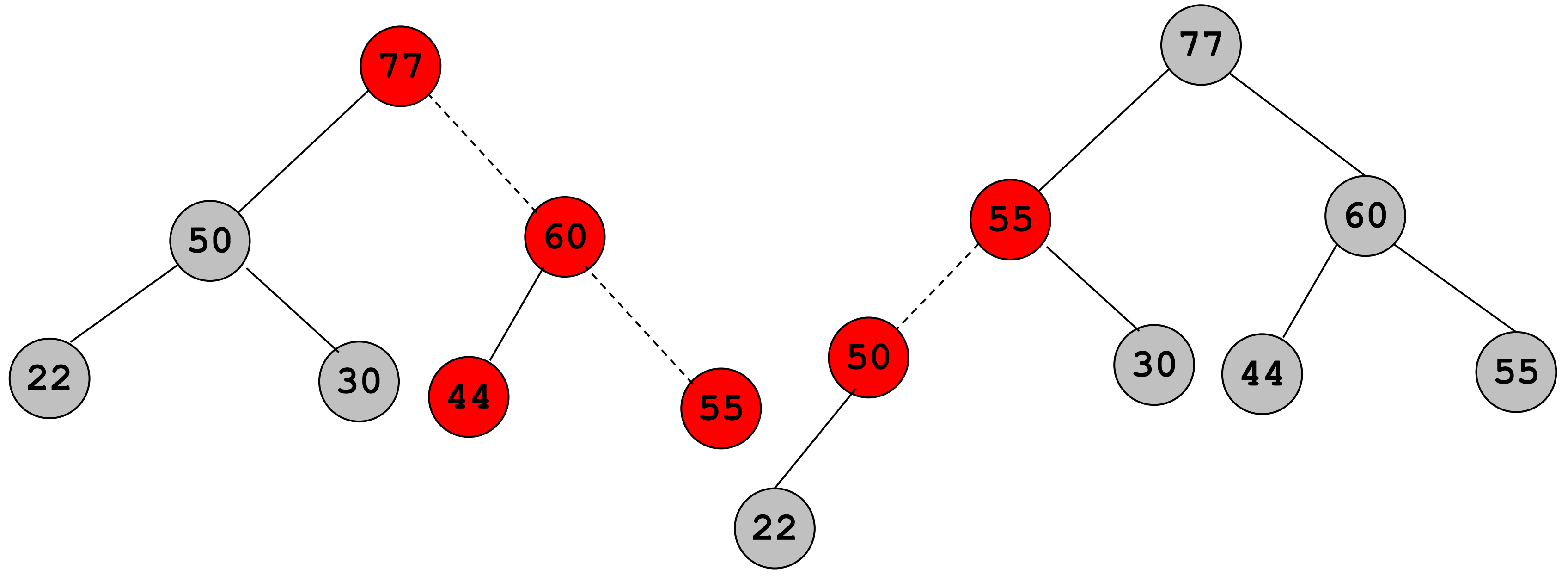
ITEM = 60

ITEM = 55

Inserting into a Heap

- Suppose we want to built a heap from the following list of numbers:

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



ITEM = 77

ITEM = 55

Inserting into a Heap

- INSHEAP(TREE, N, ITEM)

A heap H with N elements stored in array TREE and ITEM is given. Insert ITEM in H. PTR is the location ITEM as it rises in the tree and PAR is the location of parent of ITEM.

```
1. [add new node to H and initialize PTR] set N := N+1 and PTR := N
2. [find location to insert ITEM] Repeat steps 3 to 6 while PTR < 1
3.          Set PAR := $\left\lfloor \frac{PTR}{2} \right\rfloor$ [location of parent node]
4.          If ITEM ≤ TREE[PAR] then: set TREE[PTR] := ITEM and Return
   [end of If structure]
5.          Set TREE[PTR] := TREE[PAR] [moves node down]
6.          Set PTR := PAR [update PTR]
   [end of step 2 loop]
7. [Assign ITEM as the root of H] set TREE[1] := ITEM
8. Return
```

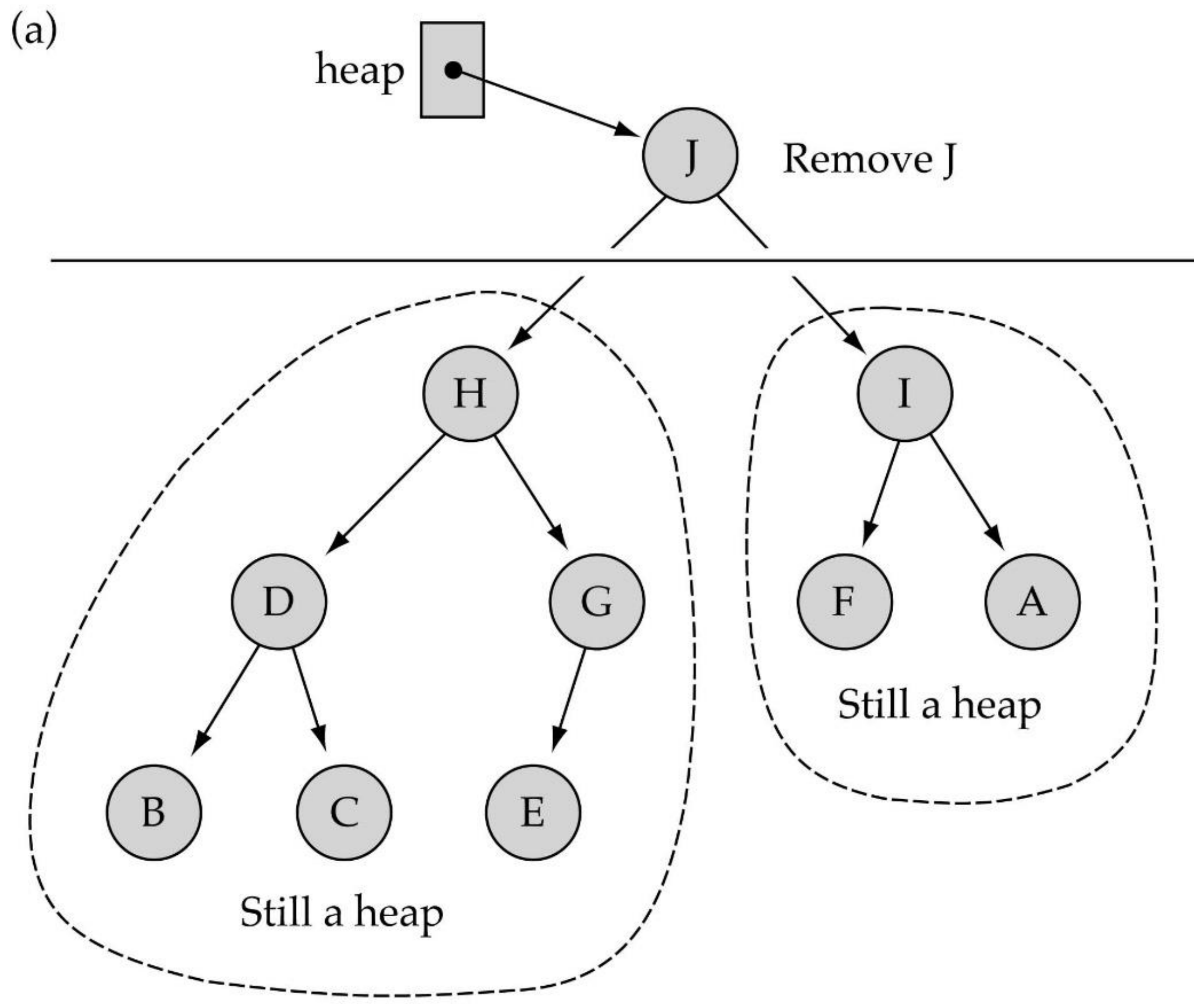
Deleting the Root of a Heap

- The root node is deleted from the heap
- After deleting the root we are left with two sub-trees without a root.
- So we must reconstruct the tree by moving the data from the last node to the root node.
- And then replace the root node with its child node and swap with large child and repeat until the node reaches its correct location.

Deleting the Root of a Heap

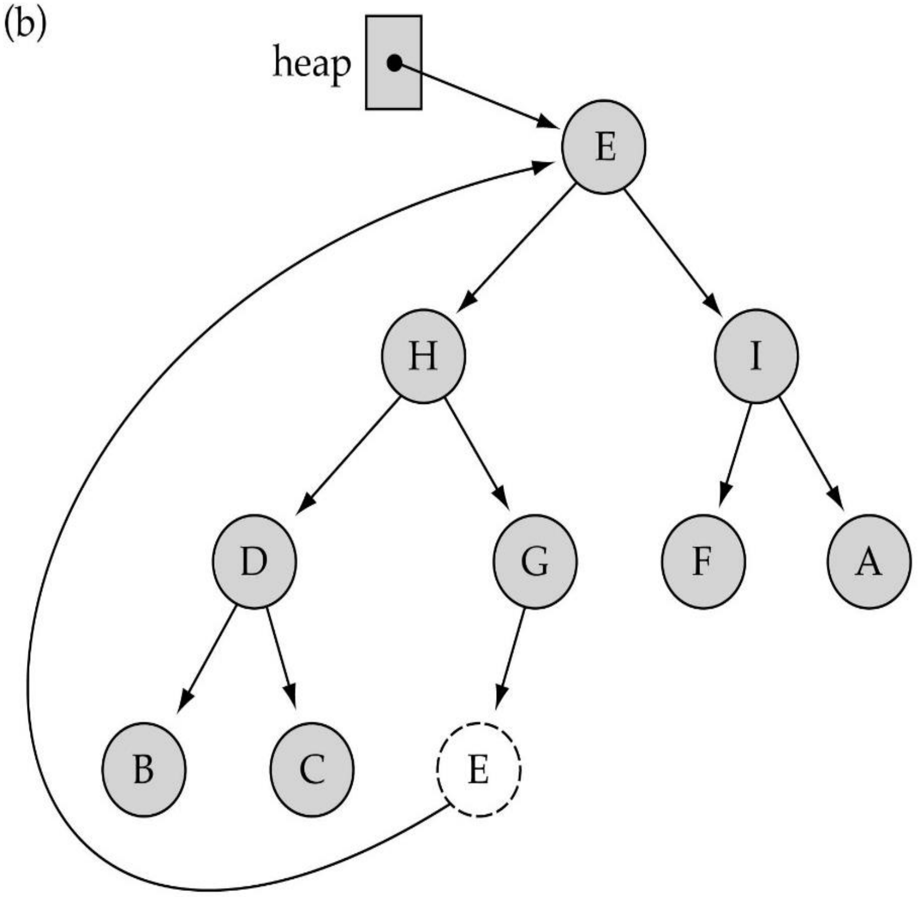
- Suppose H is a heap with N elements. We want to delete the root R of H. This is accomplished as follows:
  1) Assign the root R to some variable ITRM.
  2) Replace the deleted node R by the last node L of H so that H is still a complete tree, but not necessarily a heap.
  3) (reheap) Let L sink to its appropriate place in H so that H is finally a heap.

Deleting the Root of a Heap

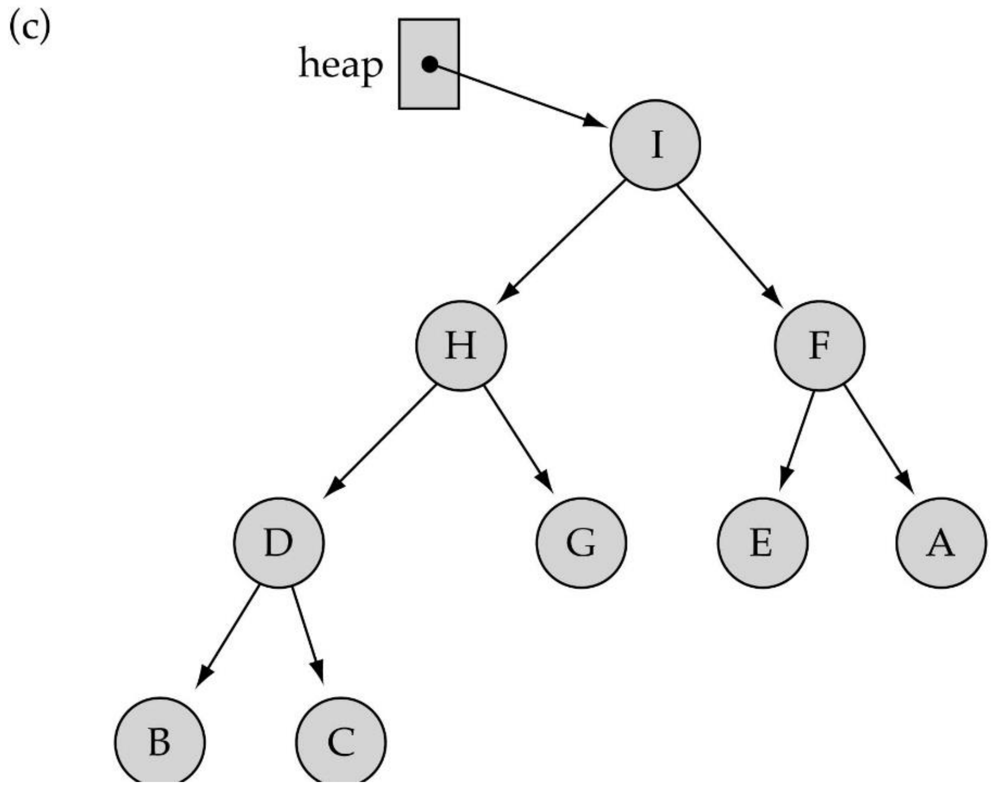


Deleting the Root of a Heap

(b)



Deleting the Root of a Heap



Deletion into a Heap

- DELHEAP(TREE, N, ITEM)

A heap H with N elements stored in array TREE. Assign the root TREE[1] of H to the variable ITEM and then reshapes the remaining elements. LAST saves the value of original last node. Pointer PTR, LEFT and RIGHT give the location of LAST and its left and right children.

```
1.  set ITEM := TREE[1] [remove root]
2.  Set LAST := TREE[N] and N := N-1 [remove last node of H]
3.  Set PTR := 1, LEFT := 2 and RIGHT := 3 [initialize the pointers]
4.  Repeat steps 5 to 7 while RIGHT ≤ N
5.      If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT] then:
            set TREE[PTR] := Last and Return      [end of If structure]
6.      If TREE[RIGHT] ≤ TREE[LEFT] then:
            set TREE[PTR] := TREE[LEFT] and PTR := LEFT
        Else: Set TREE[PTR] := TREE[RIGHT] and PTR := RIGHT [end of If]
7.      Set LEFT := 2*PTR and RIGHT := LEFT + 1
    [end of step 4 loop]
8.  If LEFT = N and if LAST < TREE[LEFT] then: set PTR := LEFT
9.  Set TREE[PTR] : = LAST
10. Return
```

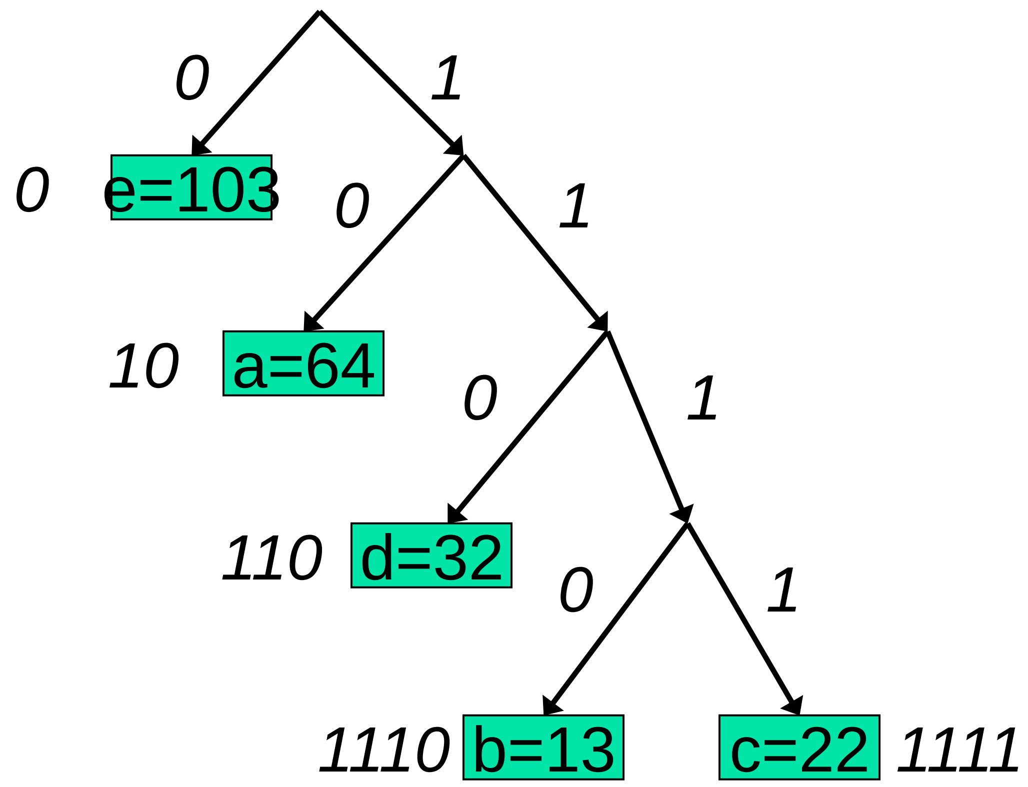
Huffman Trees

- **Problem Input:** A set of symbols, each with a frequency of occurrence.
- **Desired output:** A Huffman tree giving a code that minimizes the bit length of strings consisting of those symbols with that frequency of occurrence.
- **Strategy:** Starting with single-symbol trees, repeatedly combine the two lowest-frequency trees, giving one new tree of frequency = sum of the two frequencies. Stop when we have a single tree.

Huffman Trees

- **Implementation approach:**
  - Use a priority queue to find lowest frequency trees
  - Use binary trees to represent the Huffman (de)coding trees
- Example: b=13, c=22, d=32 a=64 e=103
  - Combine b and c: bc=35
  - Combine d and bc: d(bc)=67
  - Combine a and d(bc): a(d(bc))=131
  - Combine e and a(d(bc)): e(a(d(bc)))=234 ... done

Huffman Trees



Huffman Trees: Prioritize characters

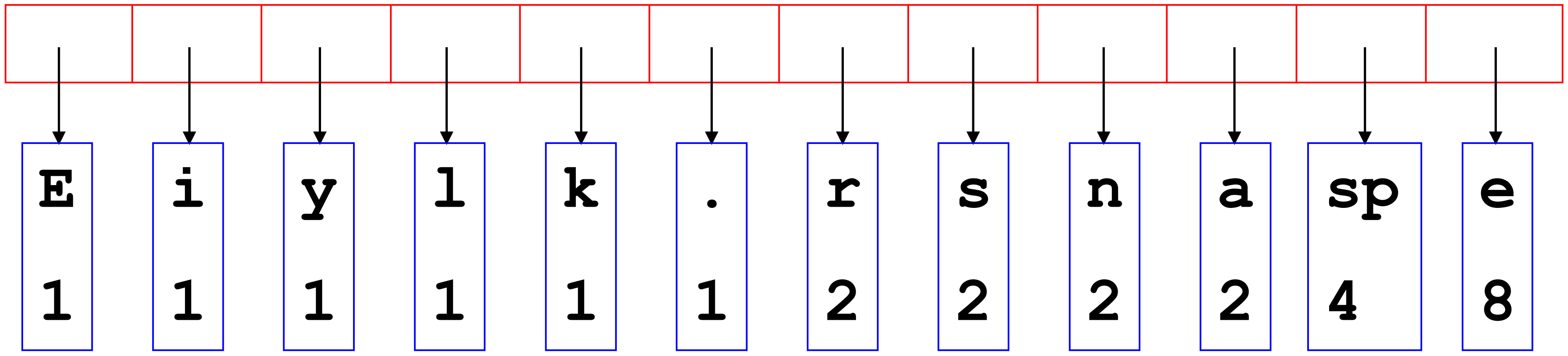
- What is the frequency of each character in the text?

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		

- 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

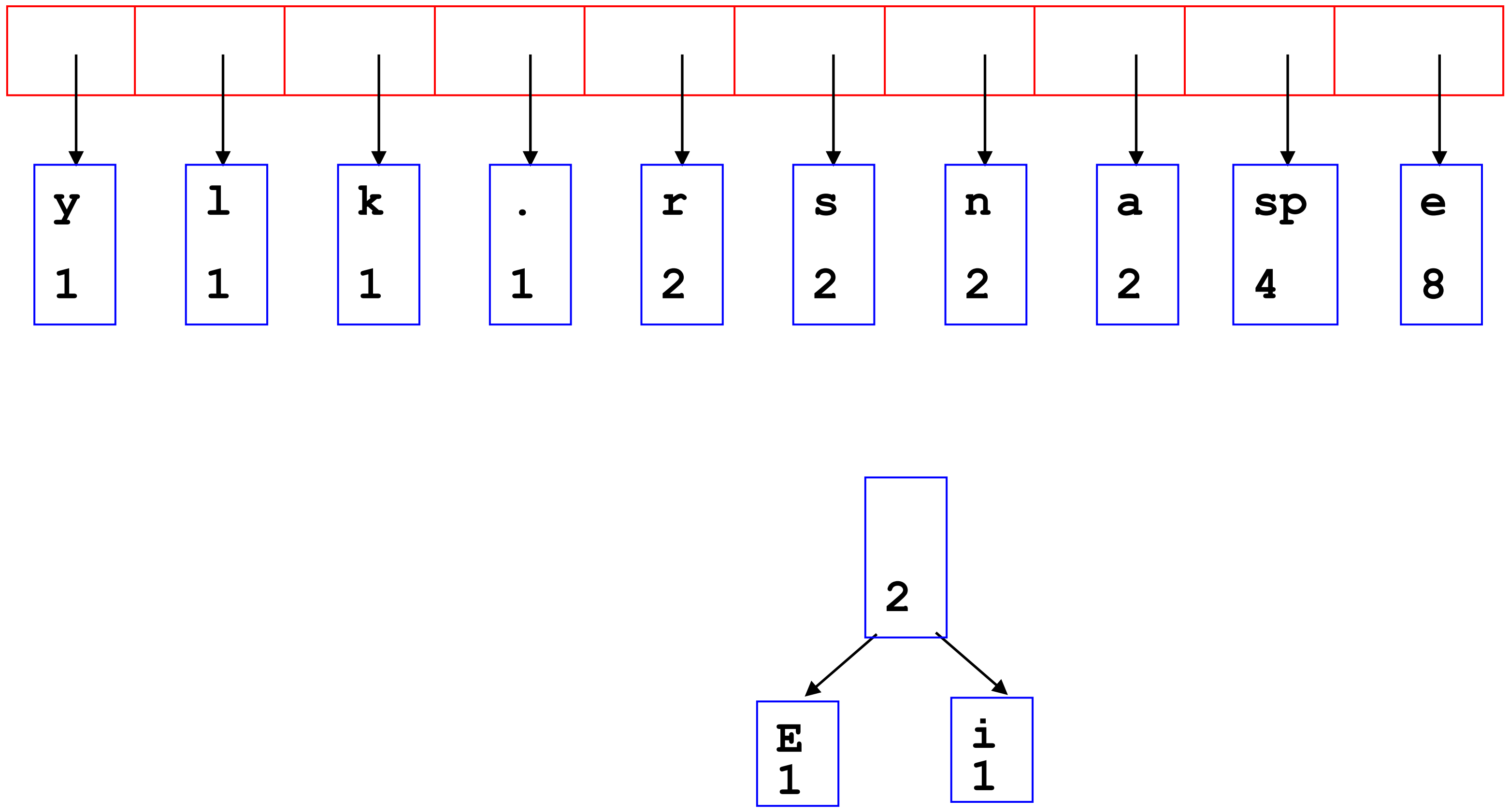
Huffman Trees: Prioritize characters

- The queue after inserting all nodes

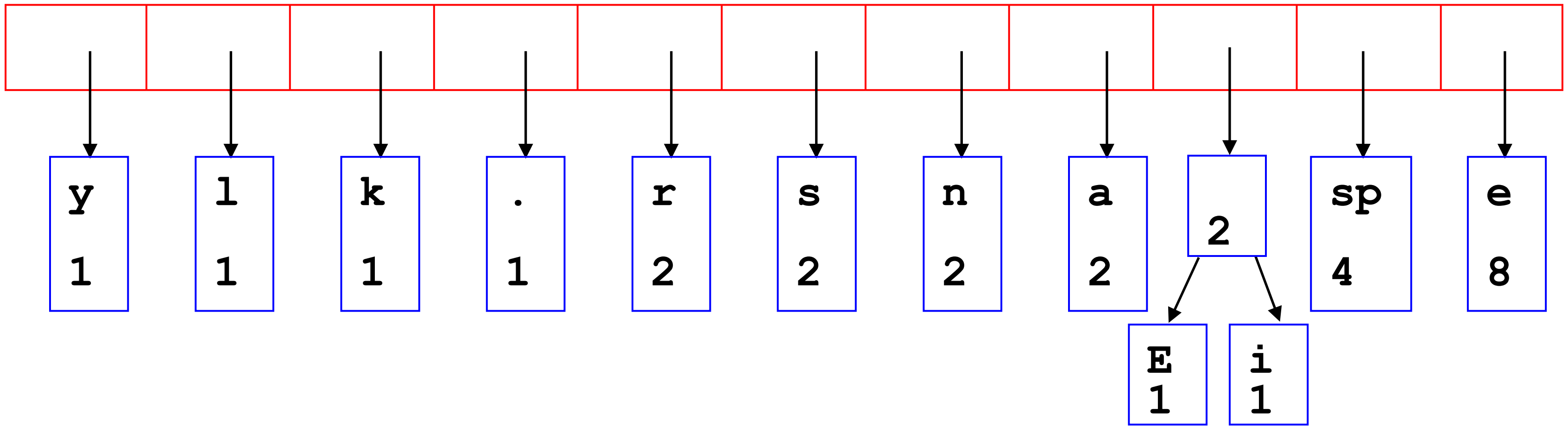


- Null Pointers are not shown

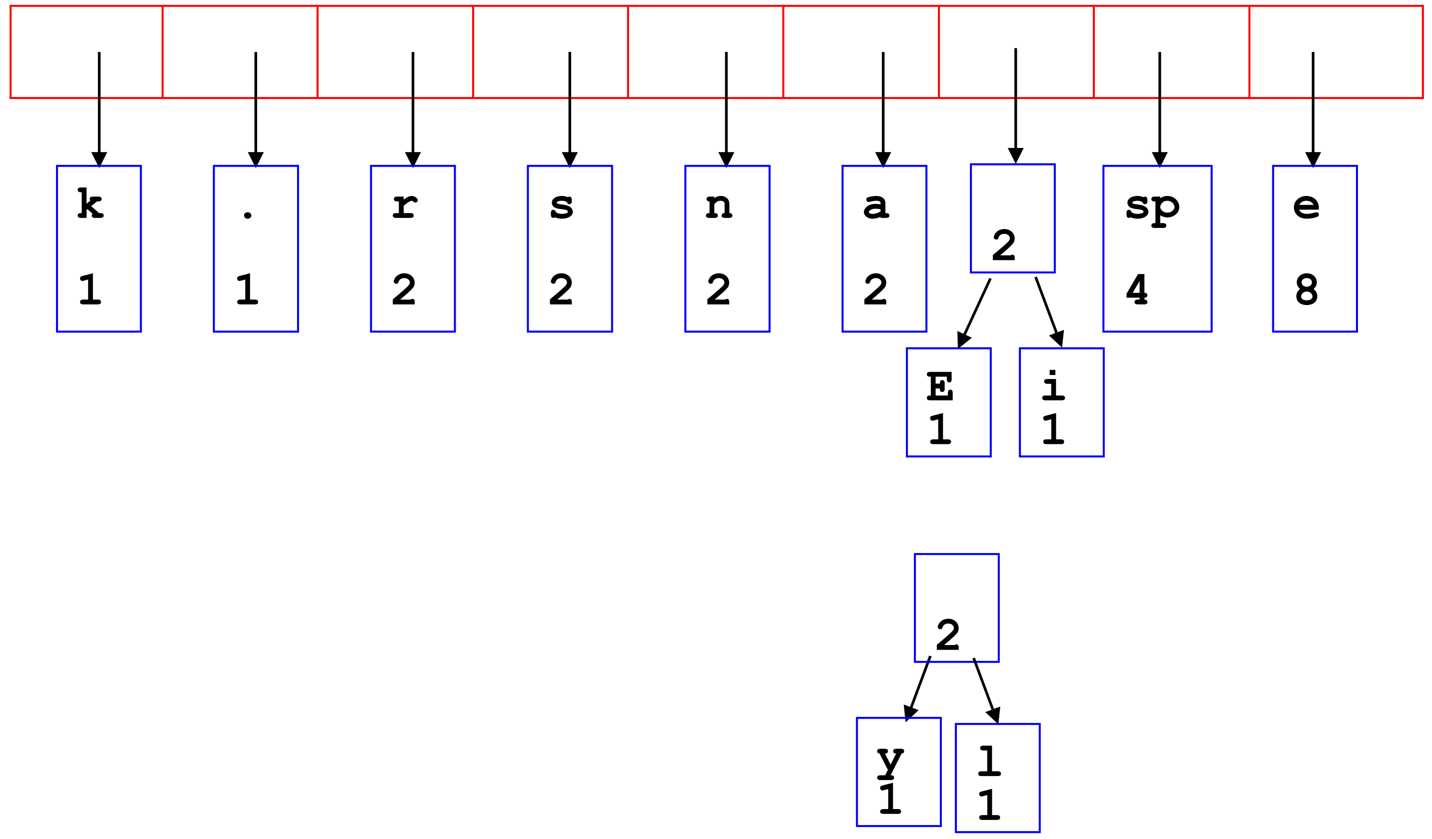
Huffman Trees: Prioritize characters



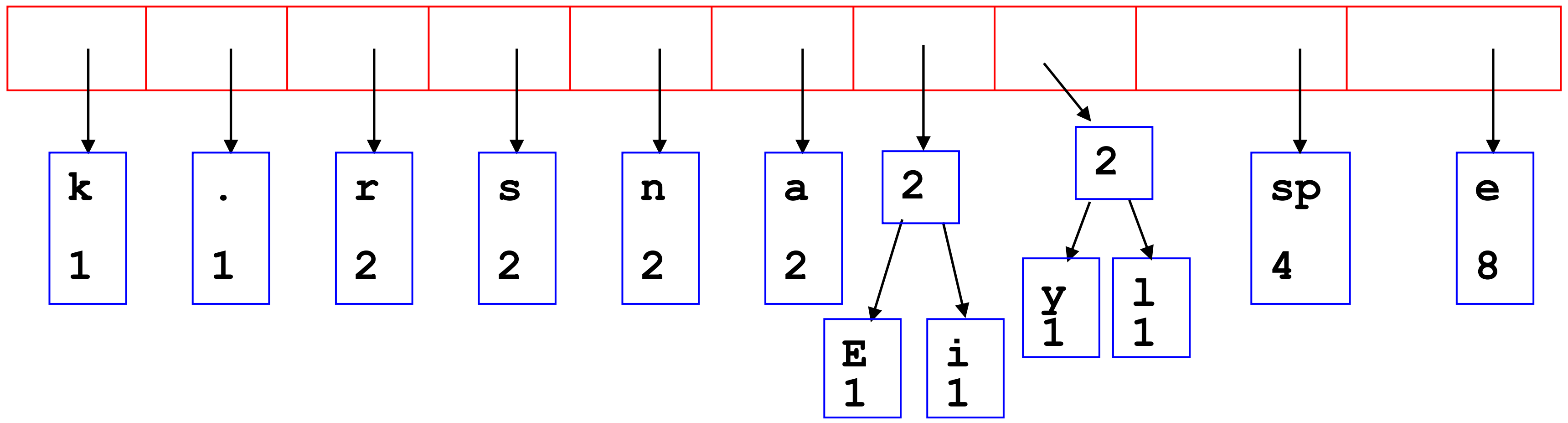
Huffman Trees: Prioritize characters



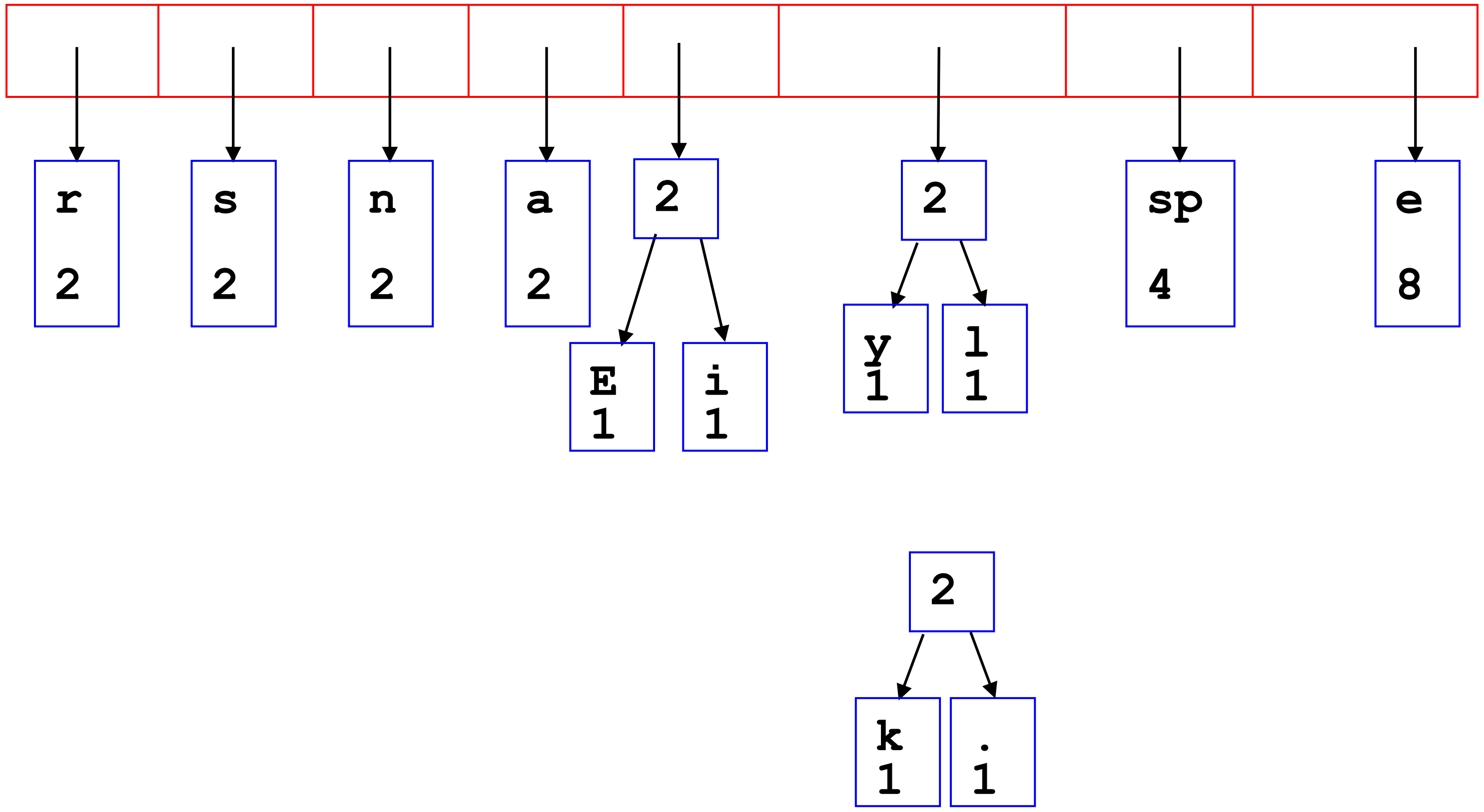
Huffman Trees: Prioritize characters



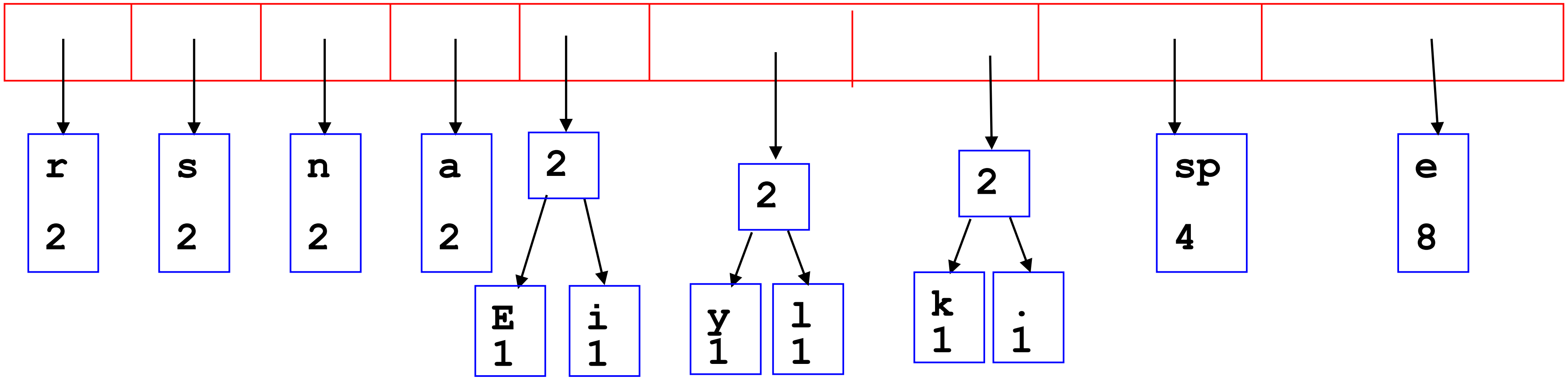
Huffman Trees: Prioritize characters



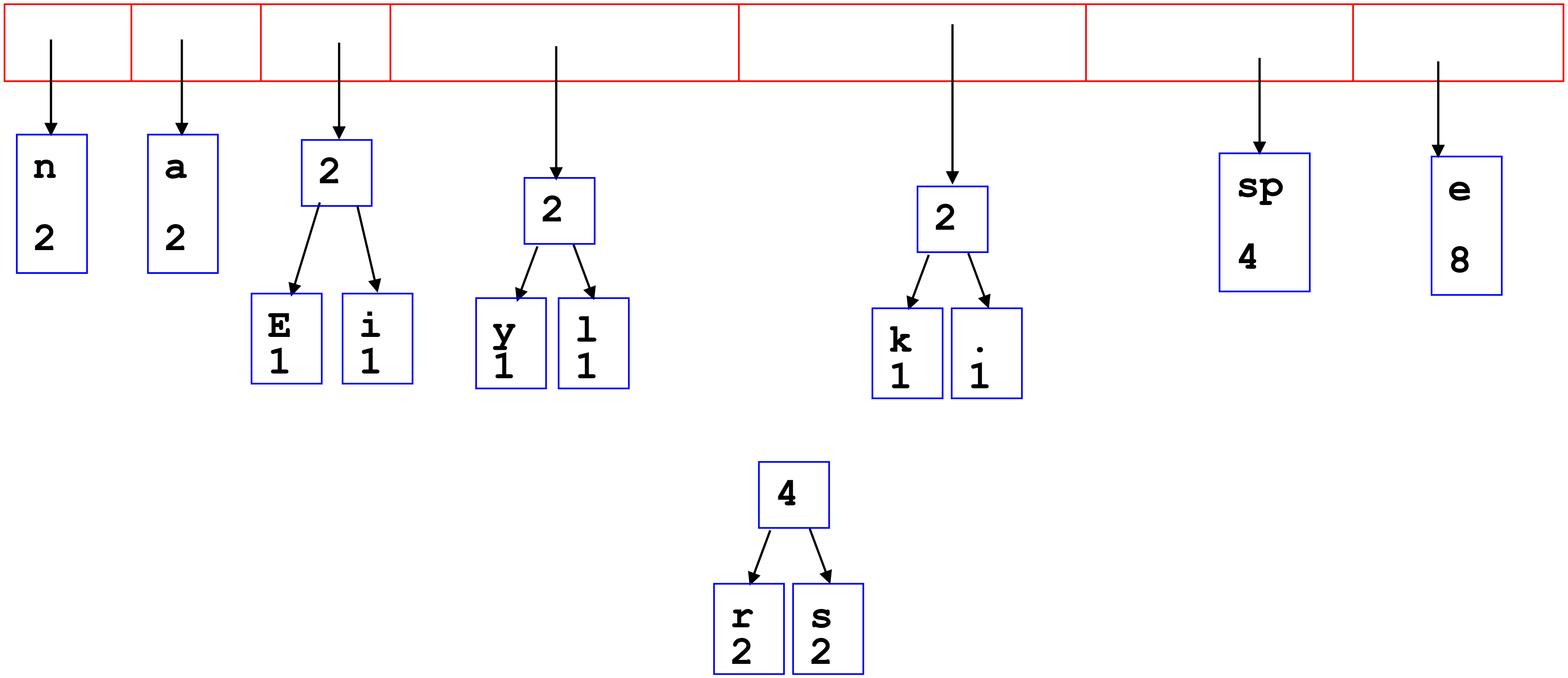
Huffman Trees: Prioritize characters



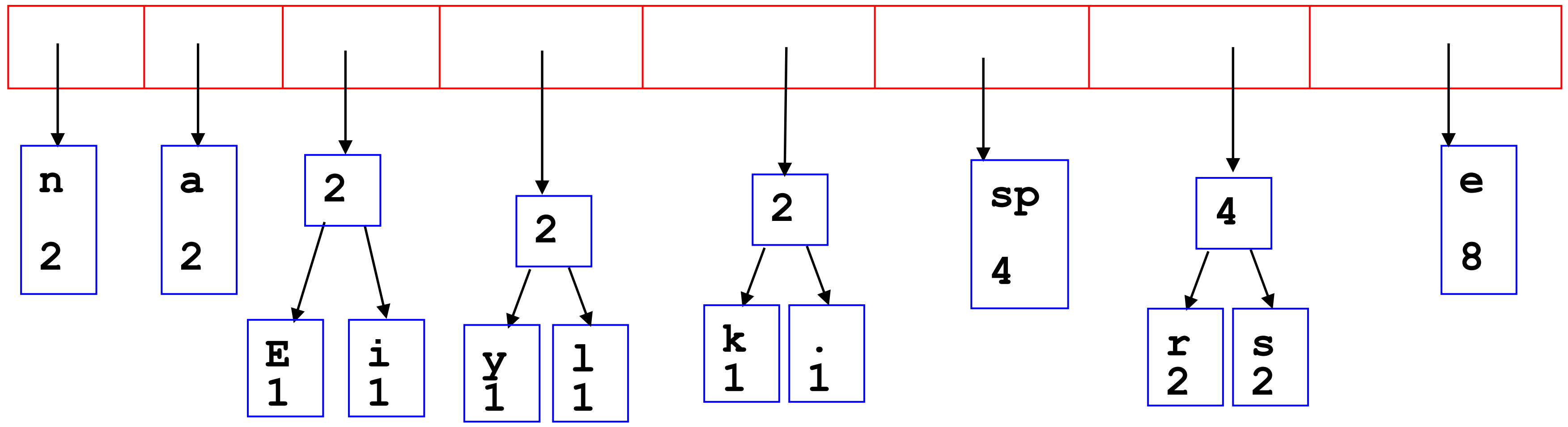
Huffman Trees: Prioritize characters



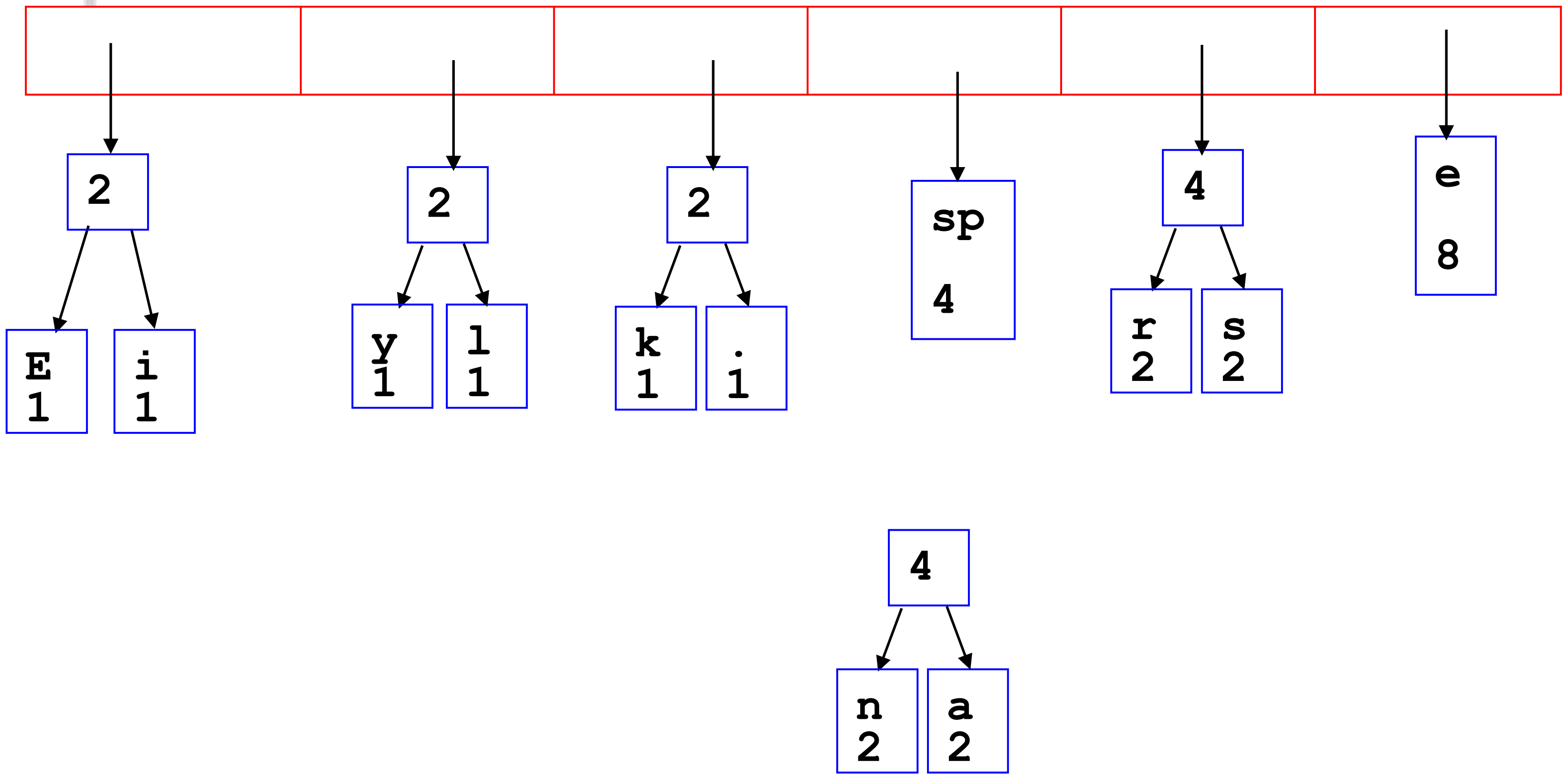
Huffman Trees: Prioritize characters



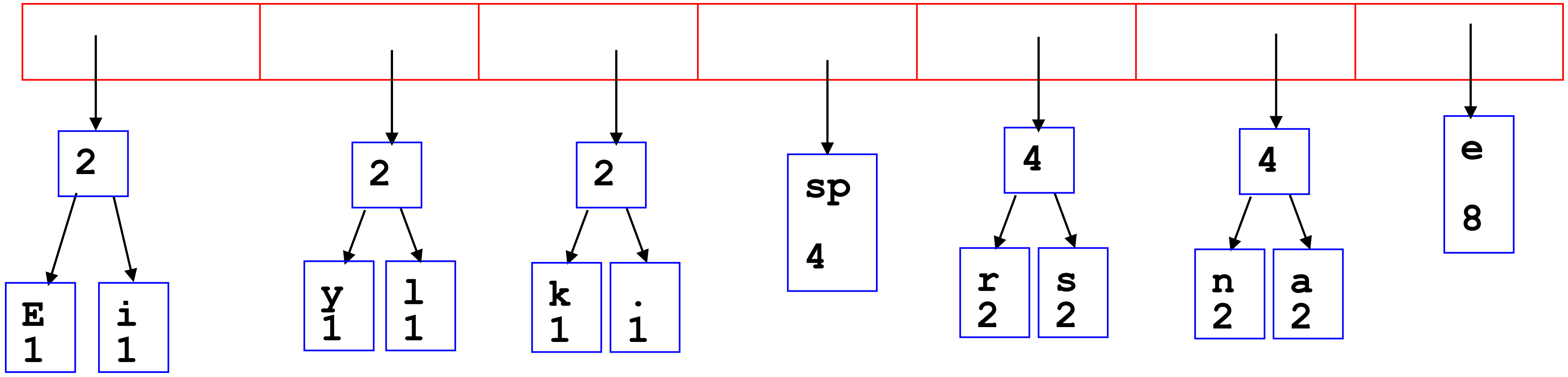
Huffman Trees: Prioritize characters



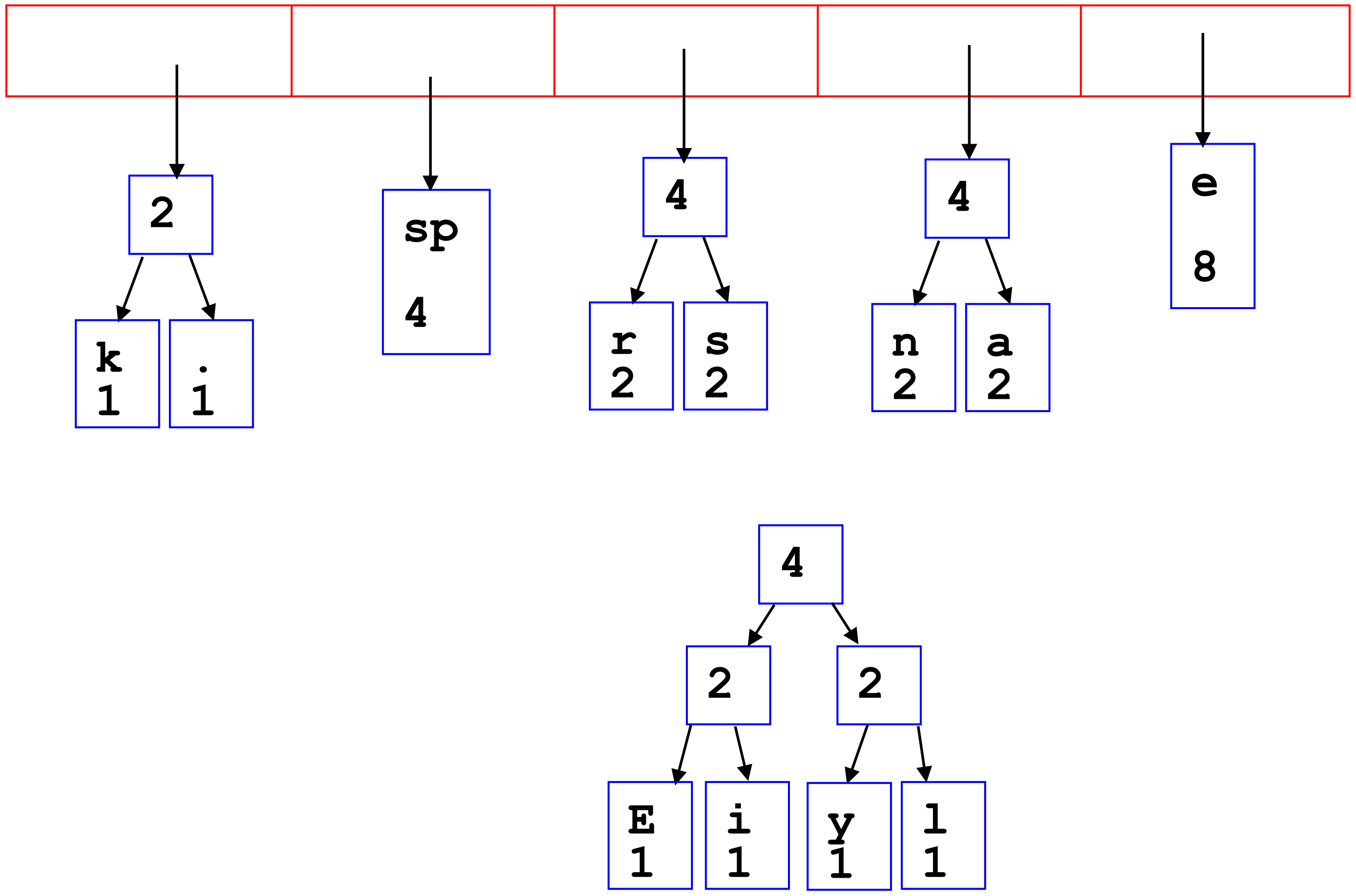
Huffman Trees: Prioritize characters



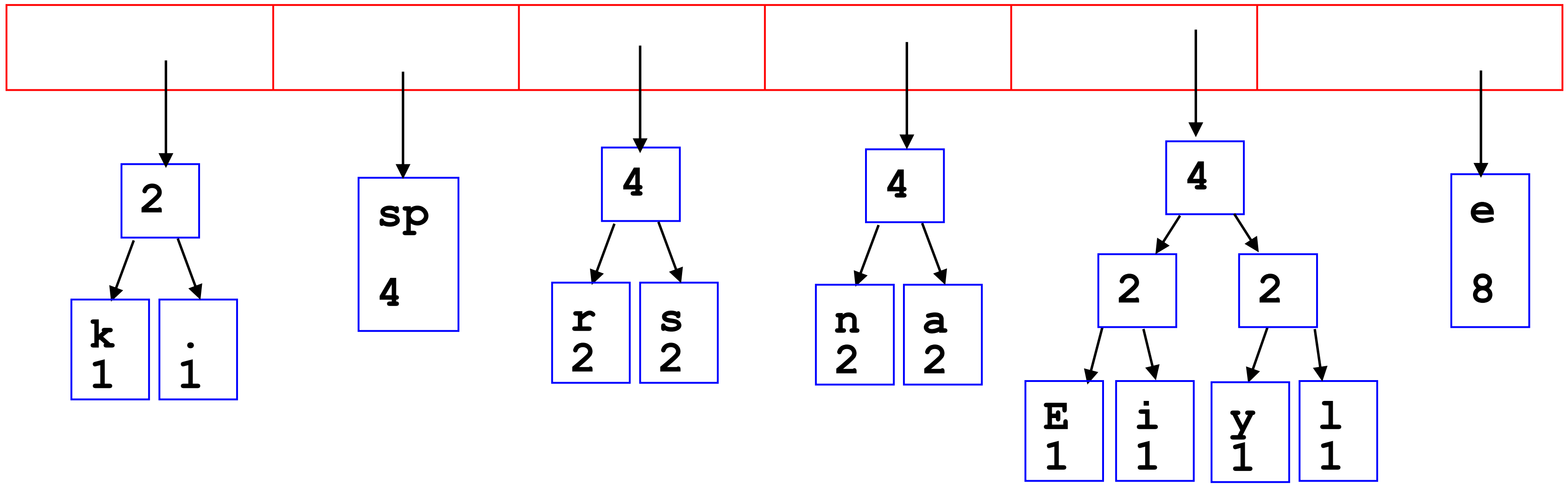
Huffman Trees: Prioritize characters



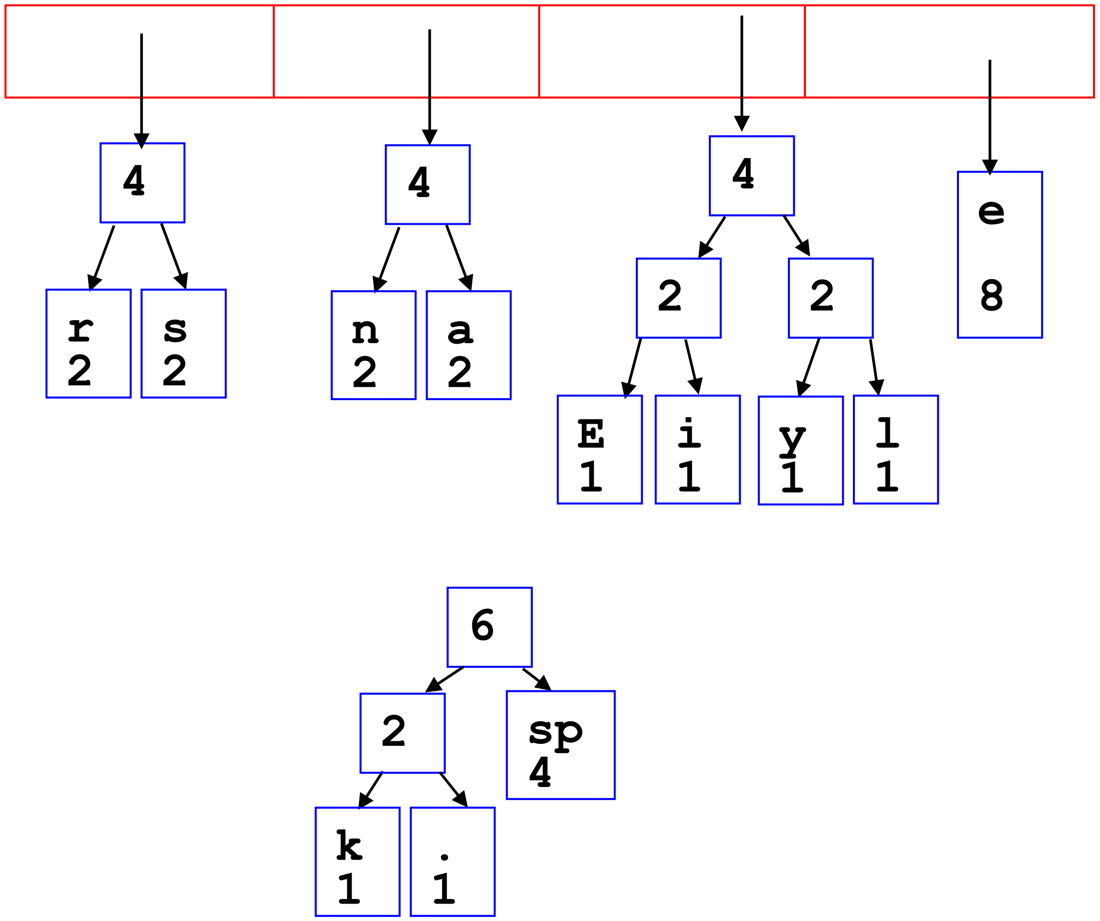
Huffman Trees: Prioritize characters



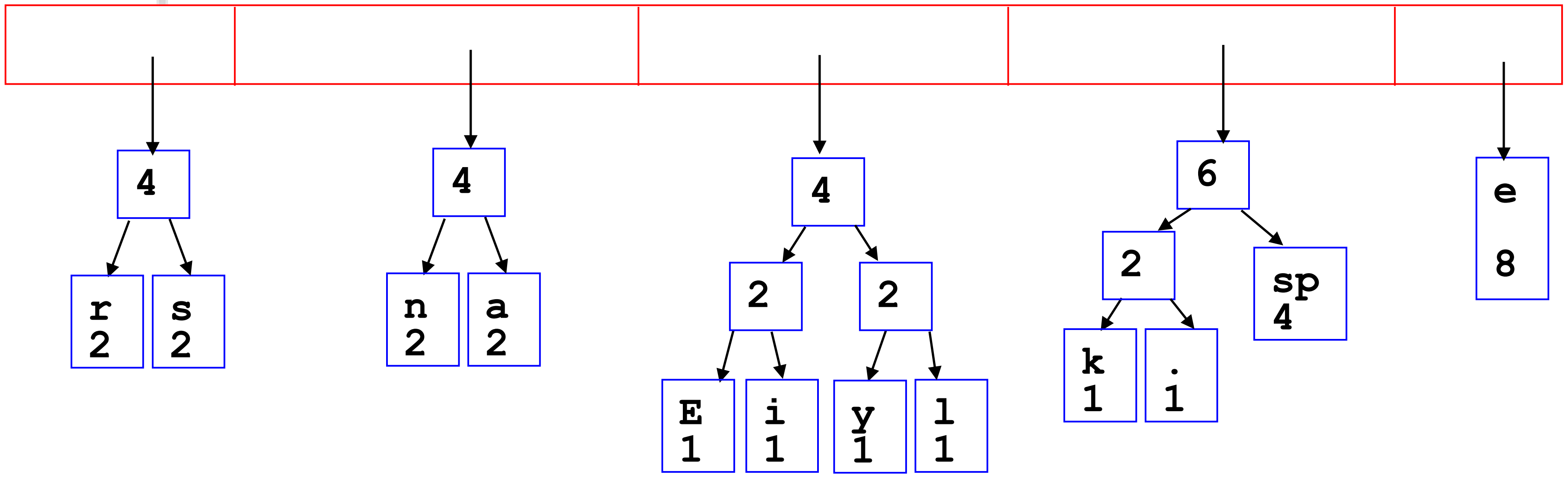
Huffman Trees: Prioritize characters



Huffman Trees: Prioritize characters

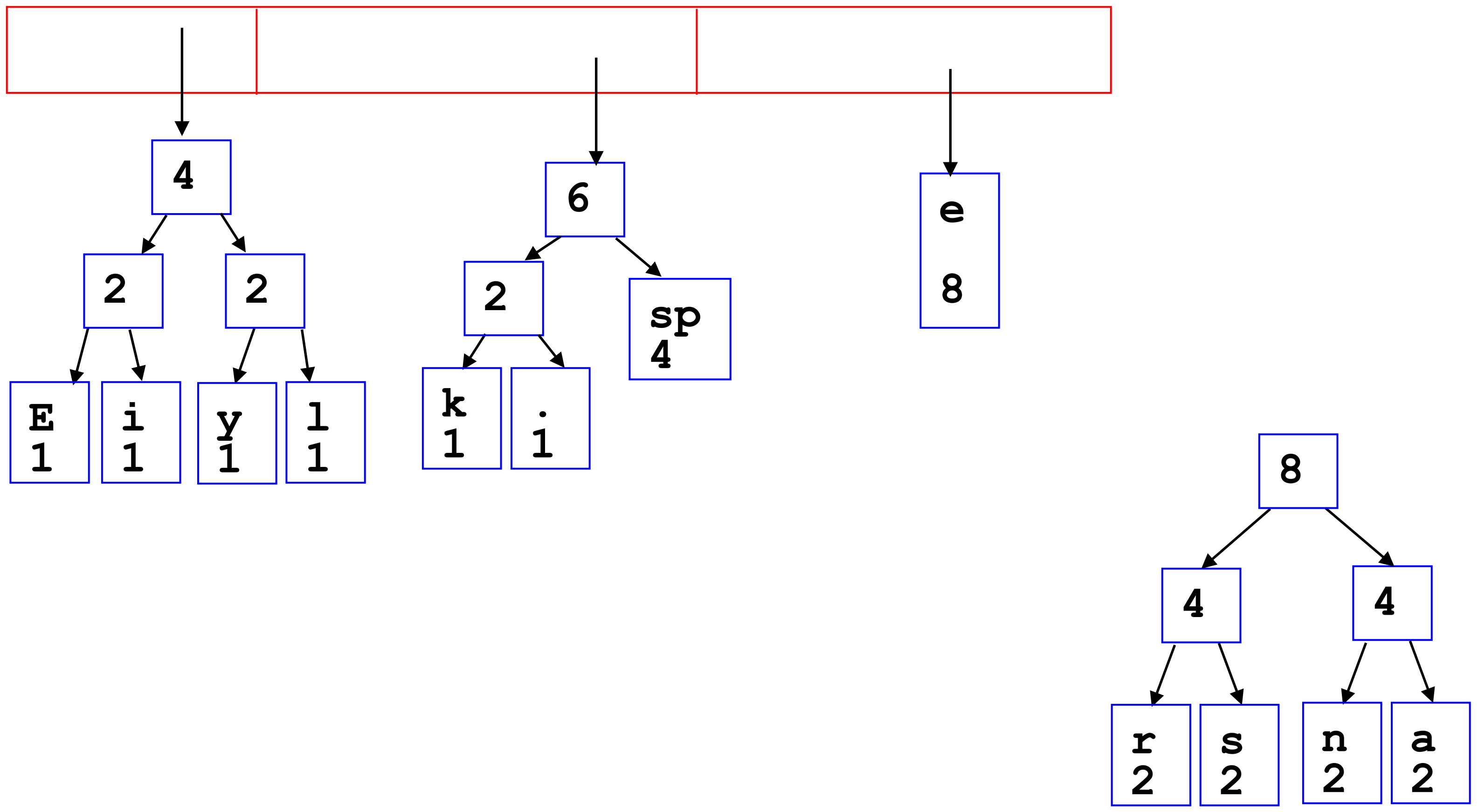


Huffman Trees: Prioritize characters

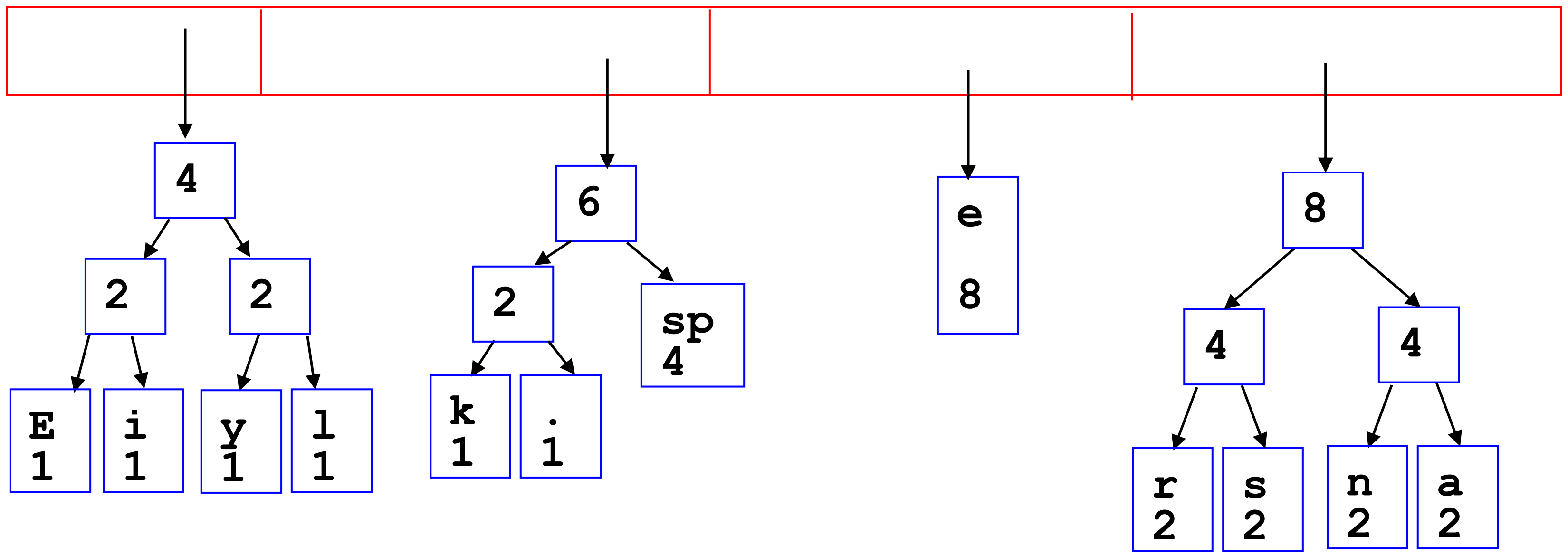


What is happening to the characters with a low number of occurrences?

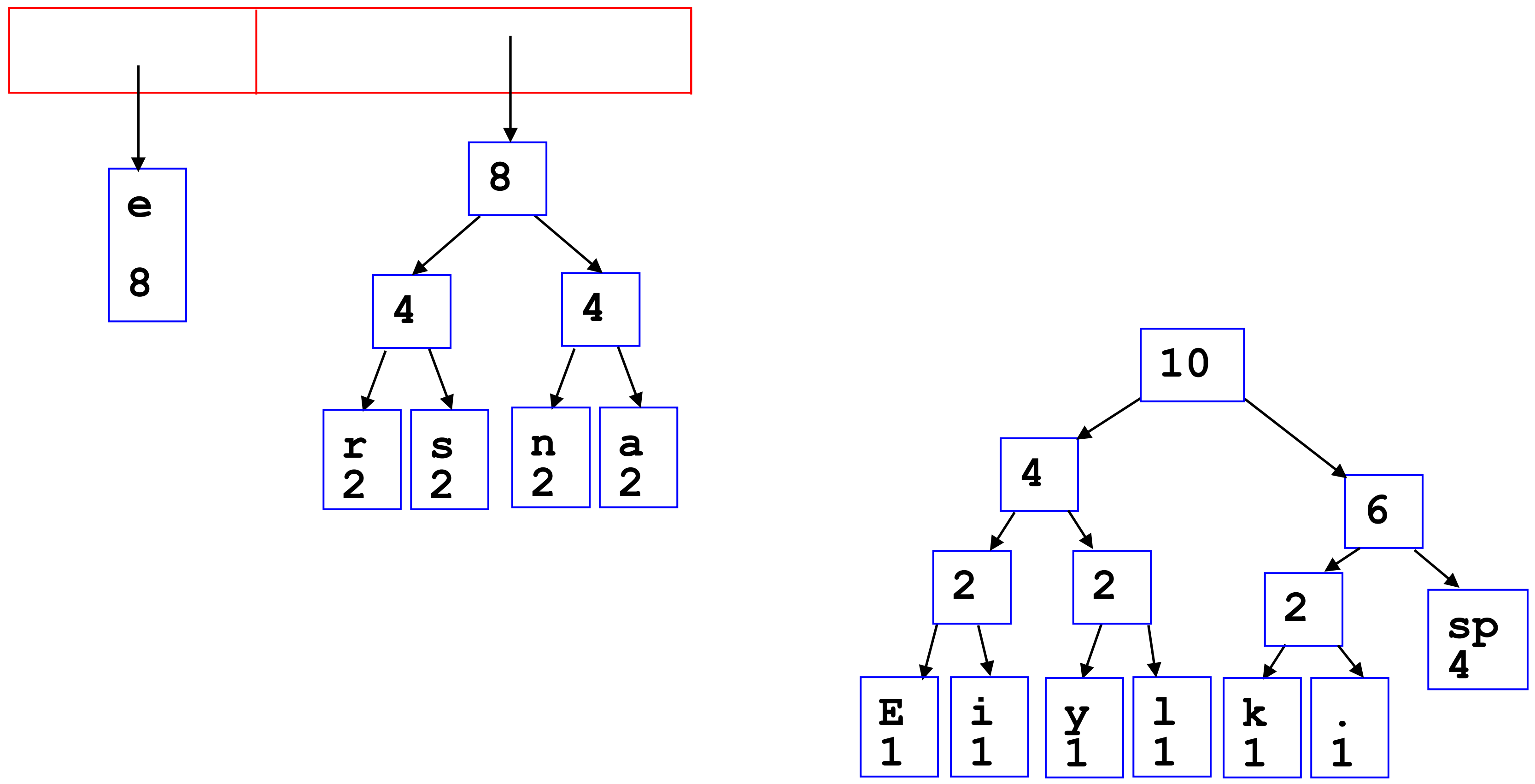
Huffman Trees: Prioritize characters



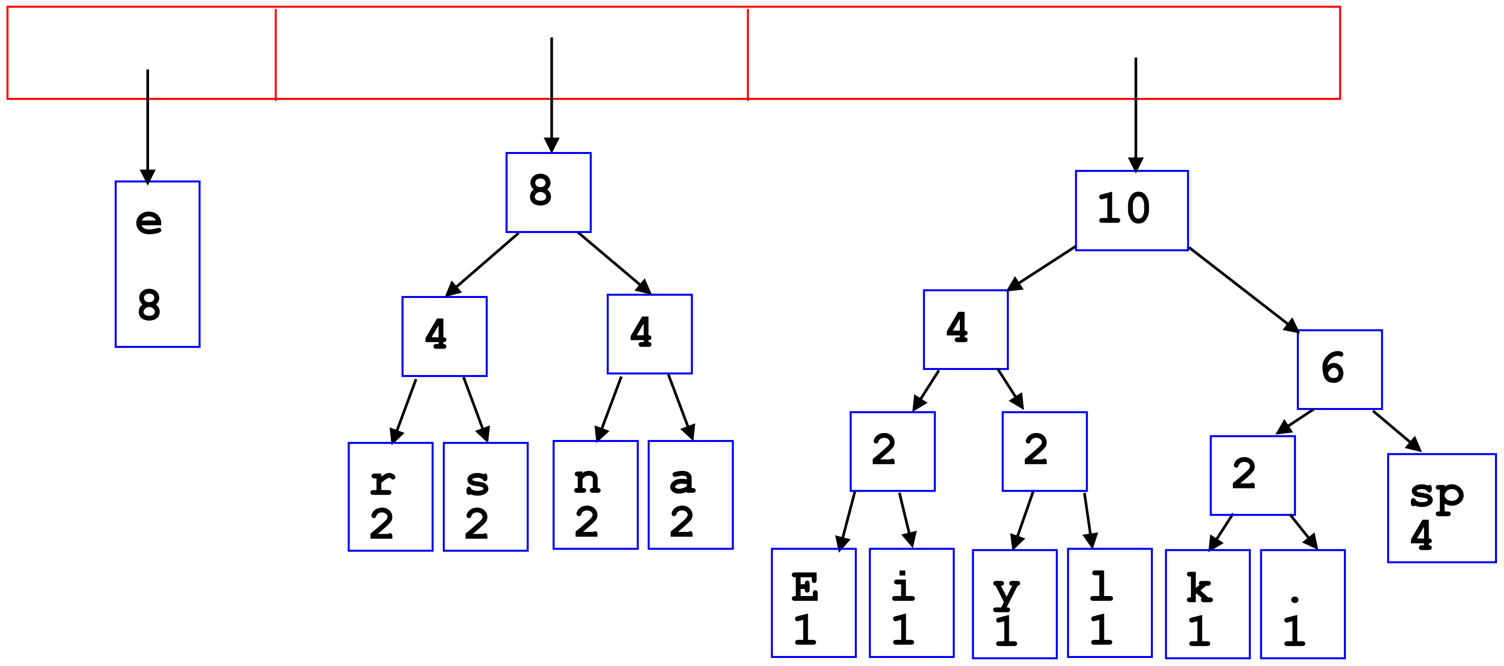
Huffman Trees: Prioritize characters



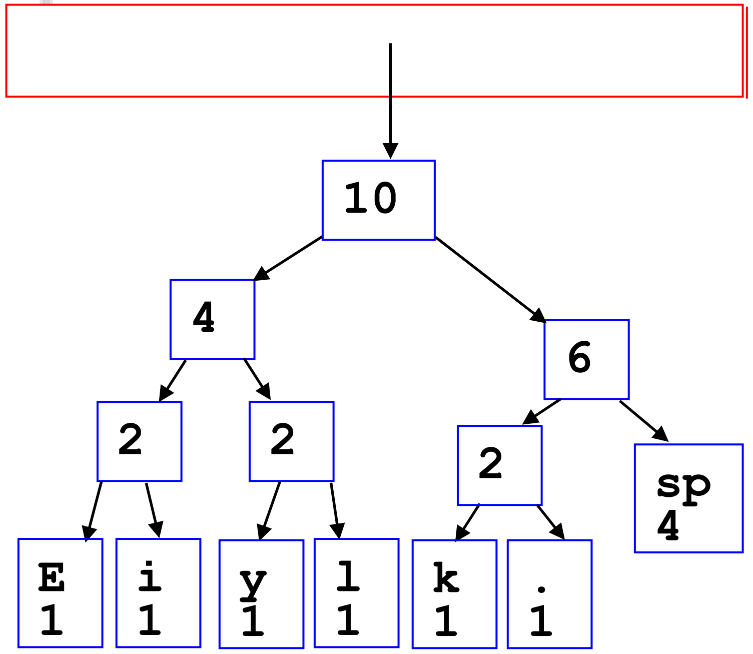
Huffman Trees: Prioritize characters

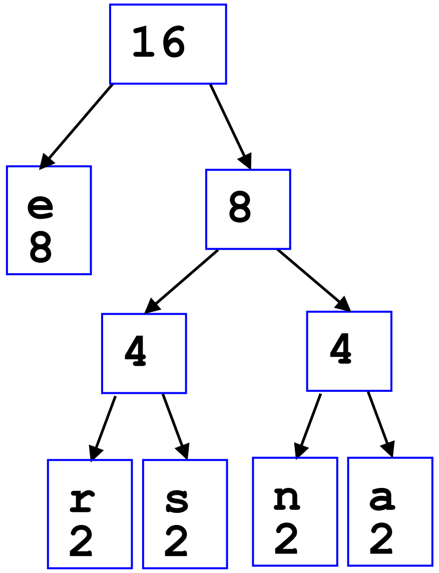


Huffman Trees: Prioritize characters

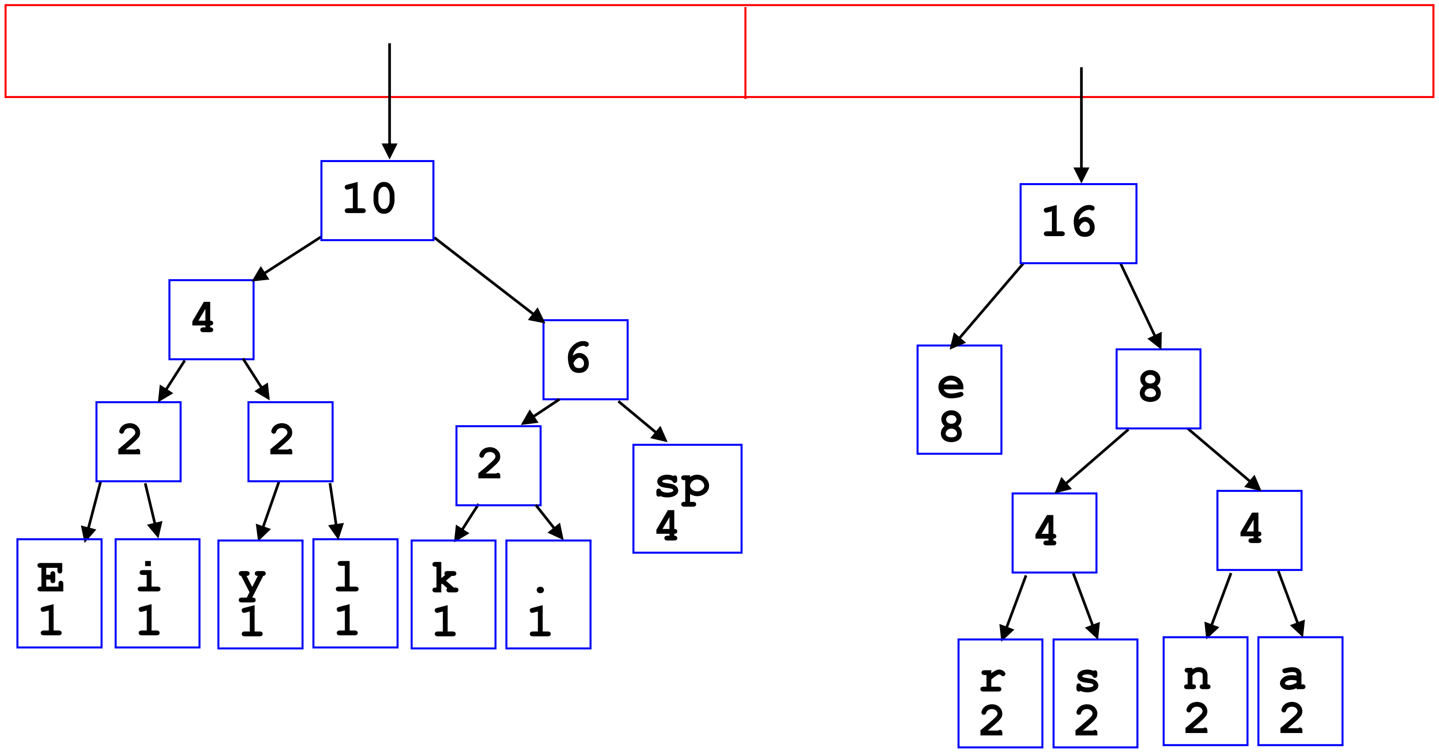


Huffman Trees: Prioritize characters

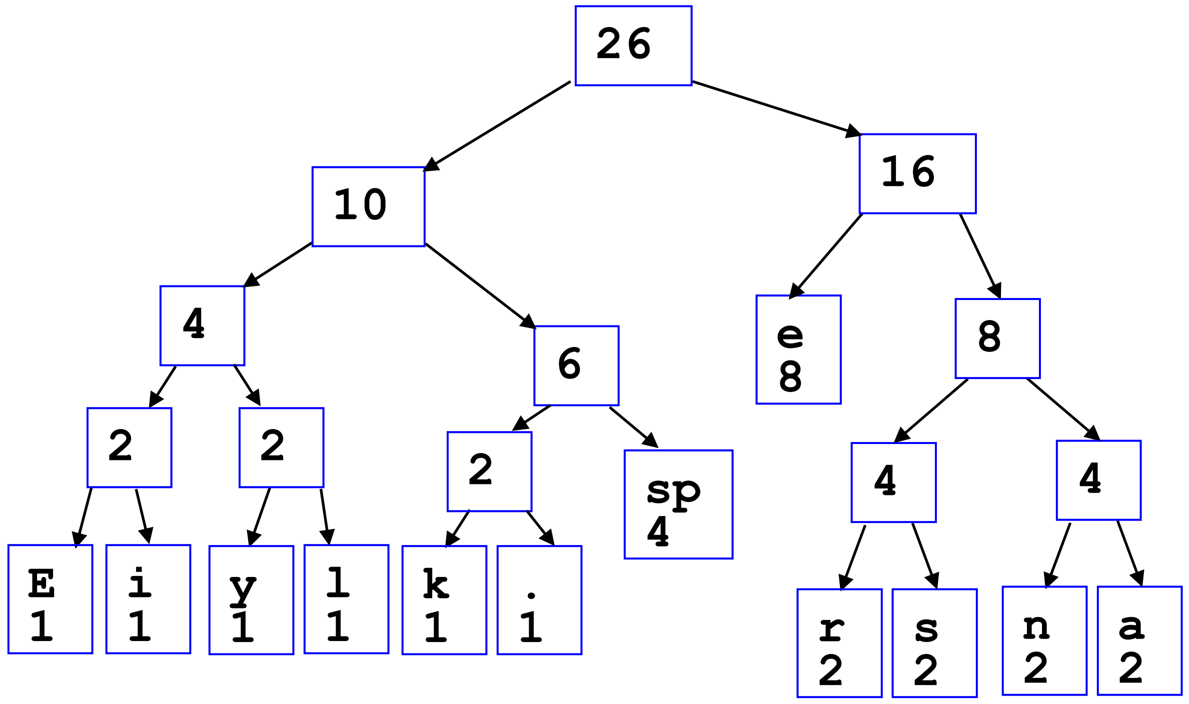




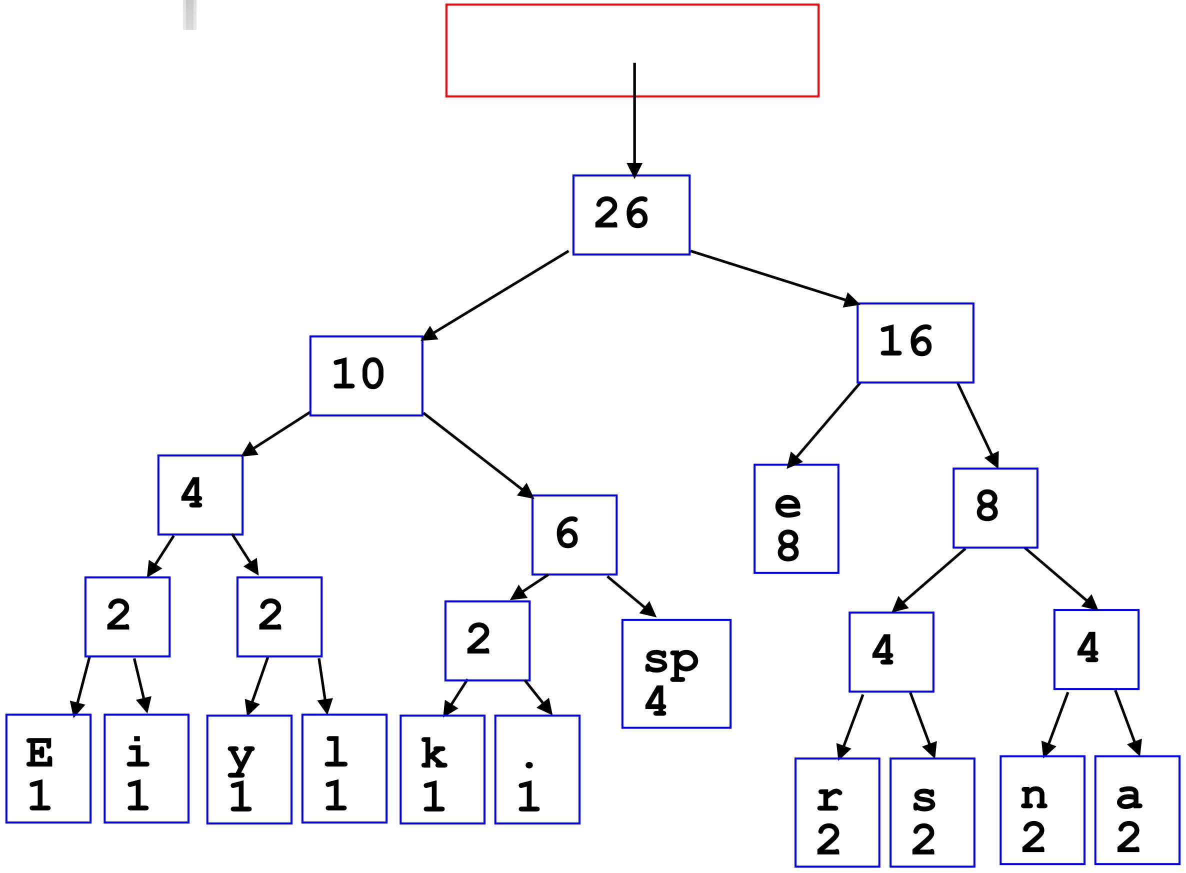
Huffman Trees: Prioritize characters



Huffman Trees: Prioritize characters



Huffman Trees: Prioritize characters



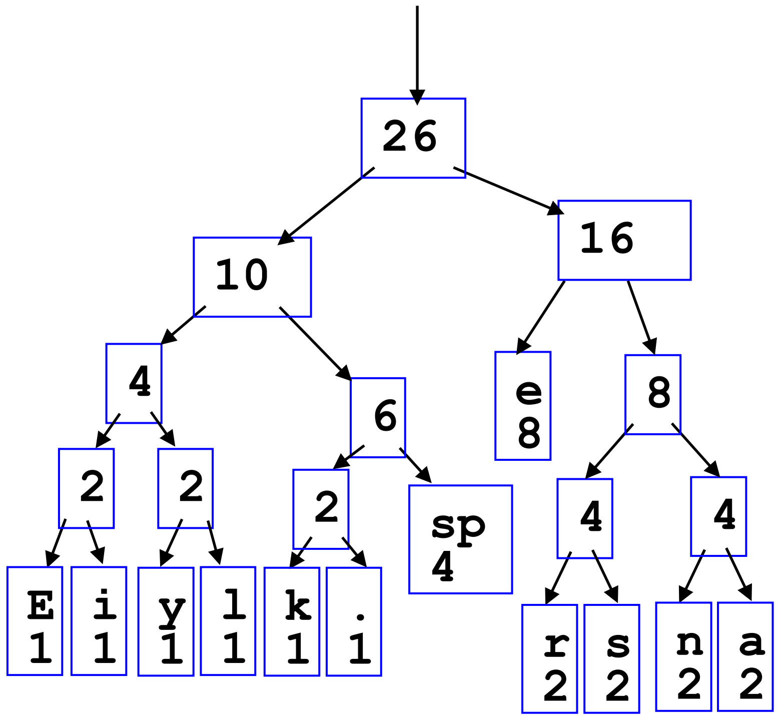
- **After enqueueing this node there is only one node left in priority queue.**

Huffman Trees: Prioritize characters

Dequeue the single node left in the queue.

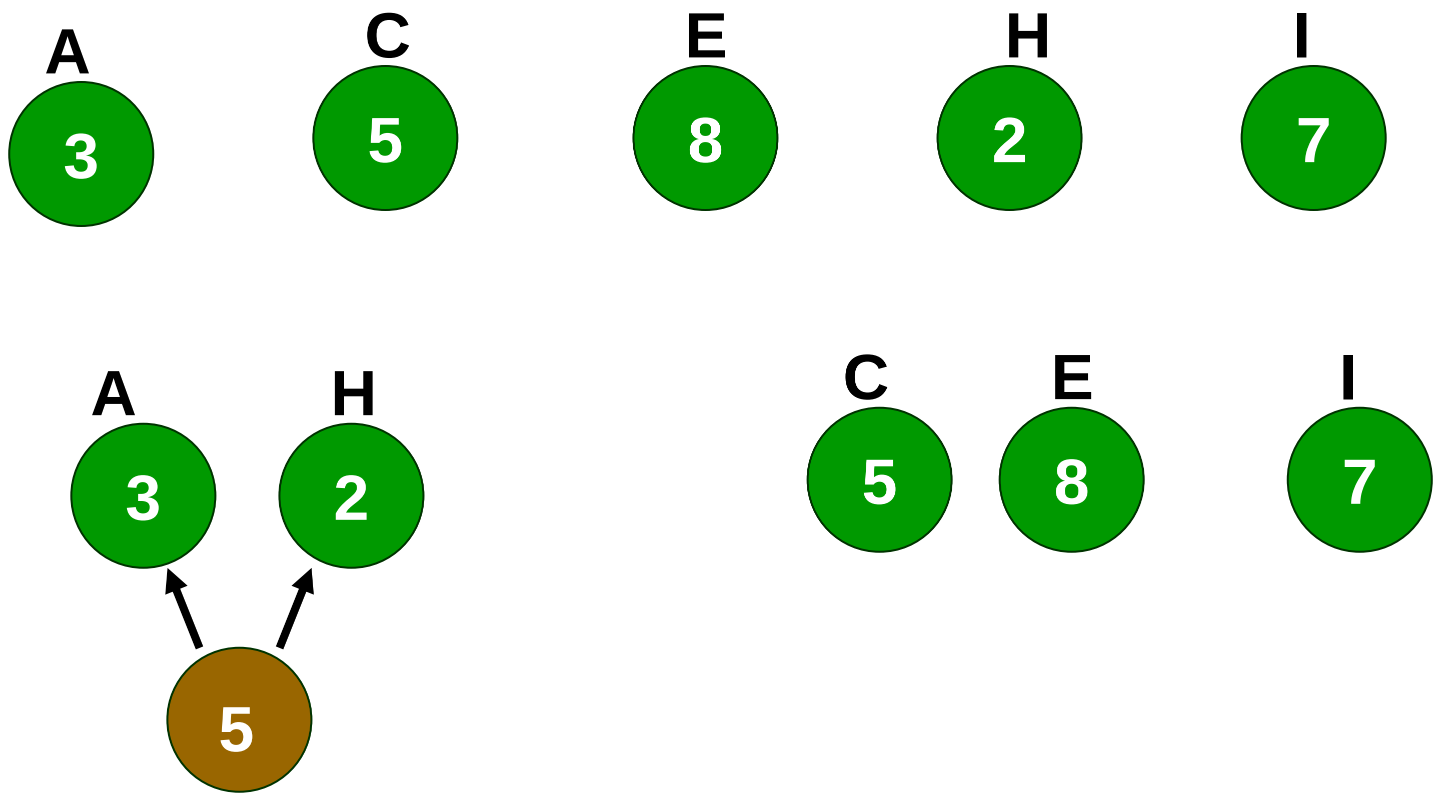
This tree contains the new code words for each character.

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

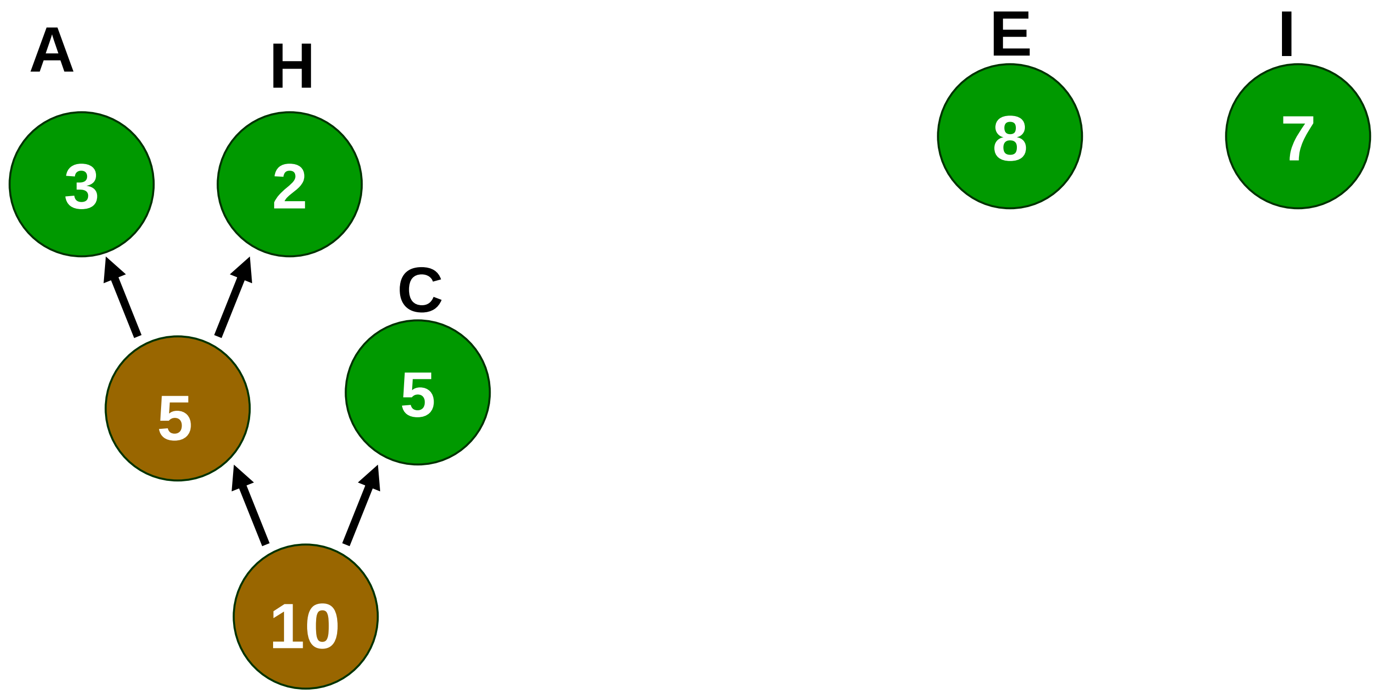


Eerie eyes seen near lake. → 26 characters

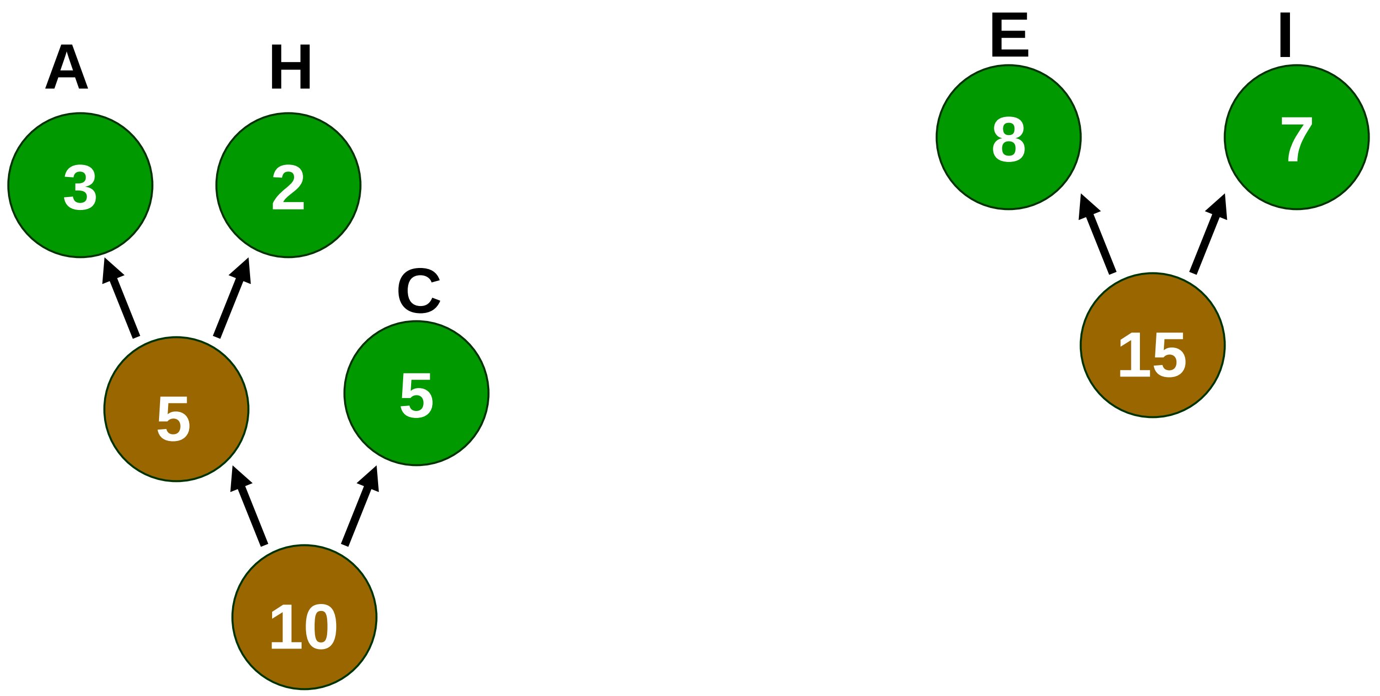
Huffman Trees Application to Coding



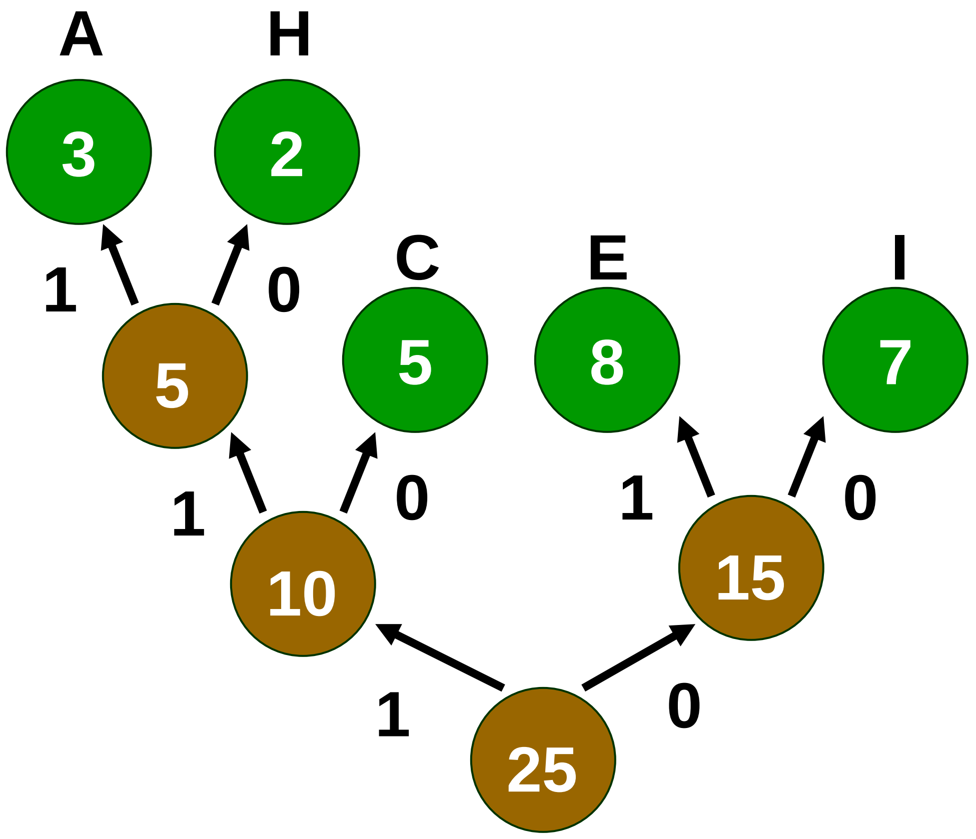
Huffman Trees Application to Coding



Huffman Trees Application to Coding



Huffman Trees Application to Coding



E	=	01
I	=	00
C	=	10
A	=	111
H	=	110

Huffman Trees Application to Coding

- Huffman code

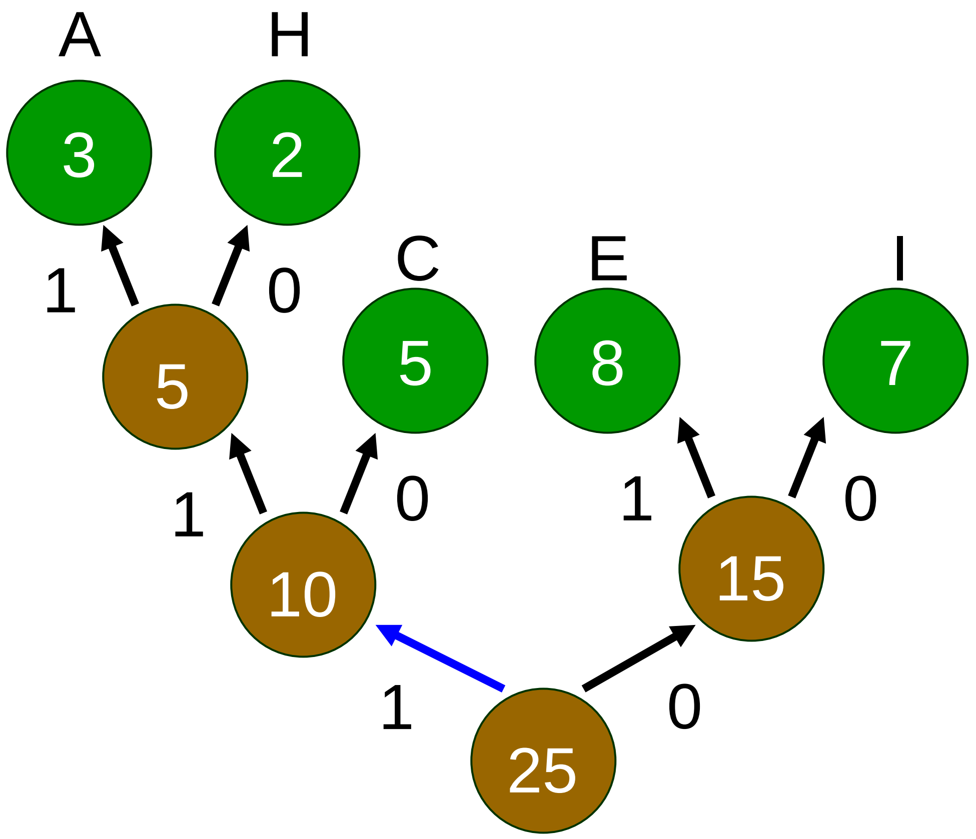
E	=	01
I	=	00
C	=	10
A	=	111
H	=	110

- Input
  - ACE
- Output
  - (111)(10)(01) = 1111001

Huffman Trees

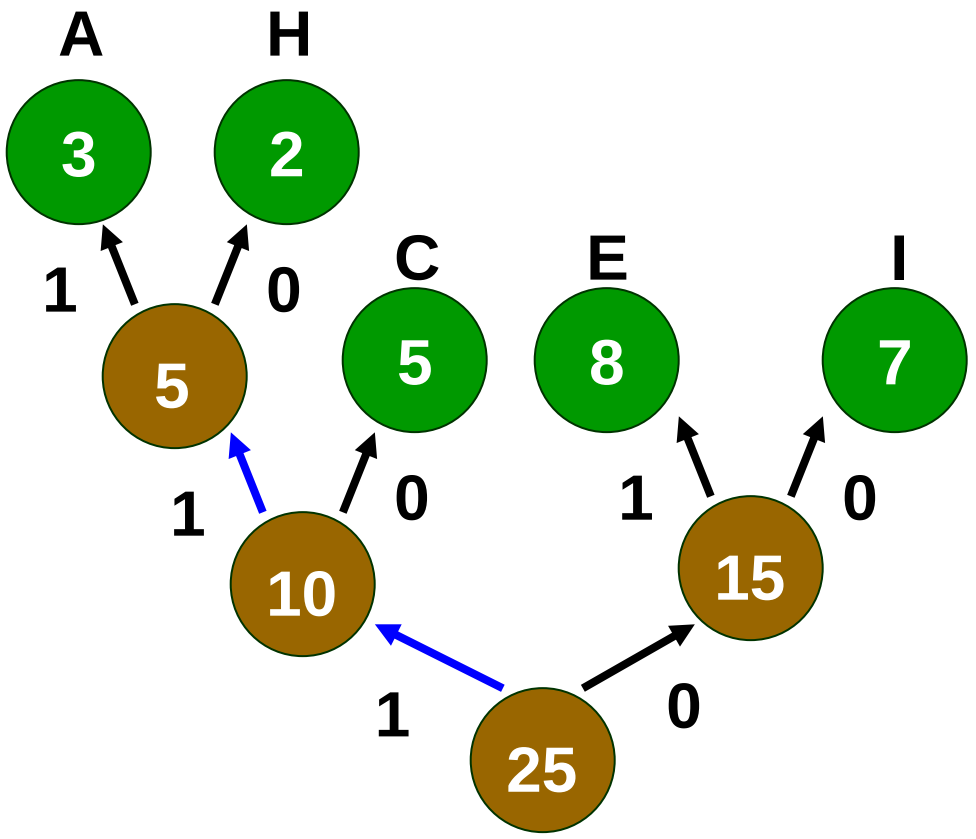
- Decoding
    - Read compressed file & binary tree
    - Use binary tree to decode file
        - Follow path from root to leaf

Huffman Trees Application to Coding



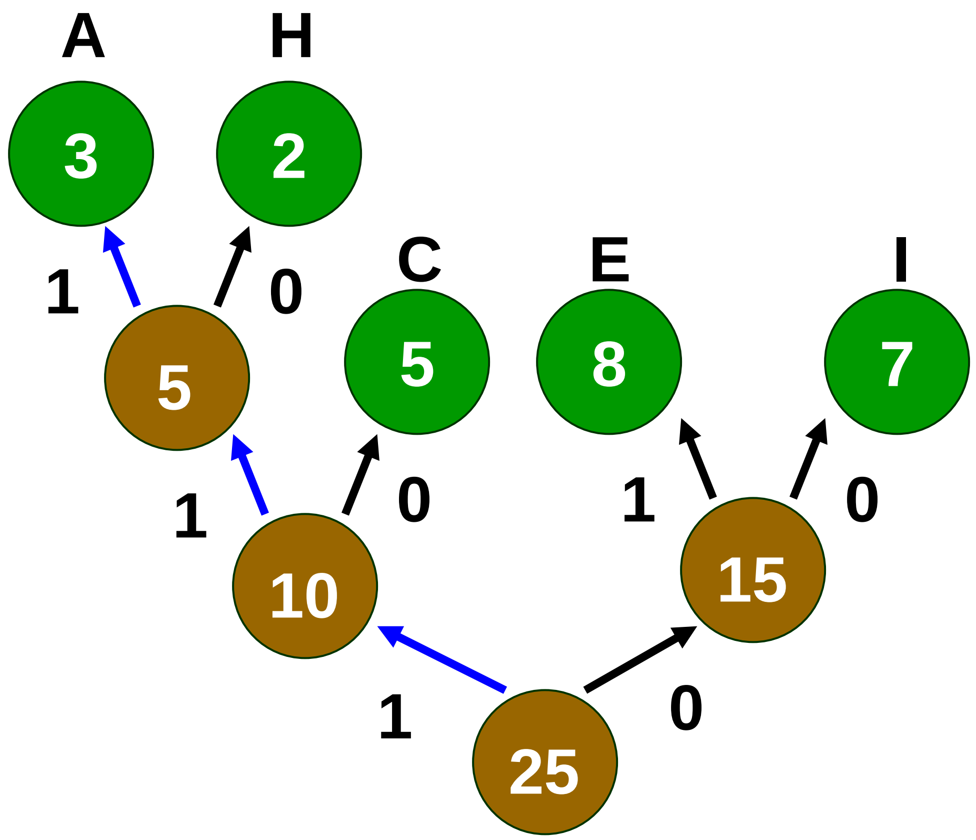
1111001

Huffman Trees Application to Coding



1111001

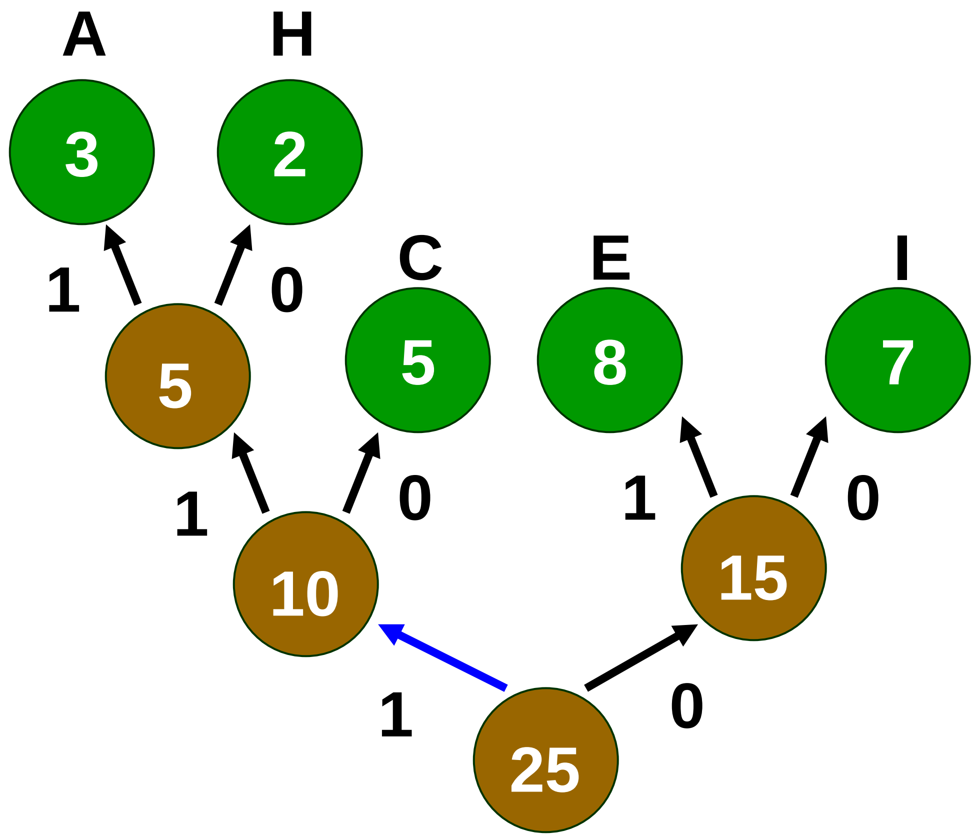
Huffman Trees Application to Coding



1111001

A

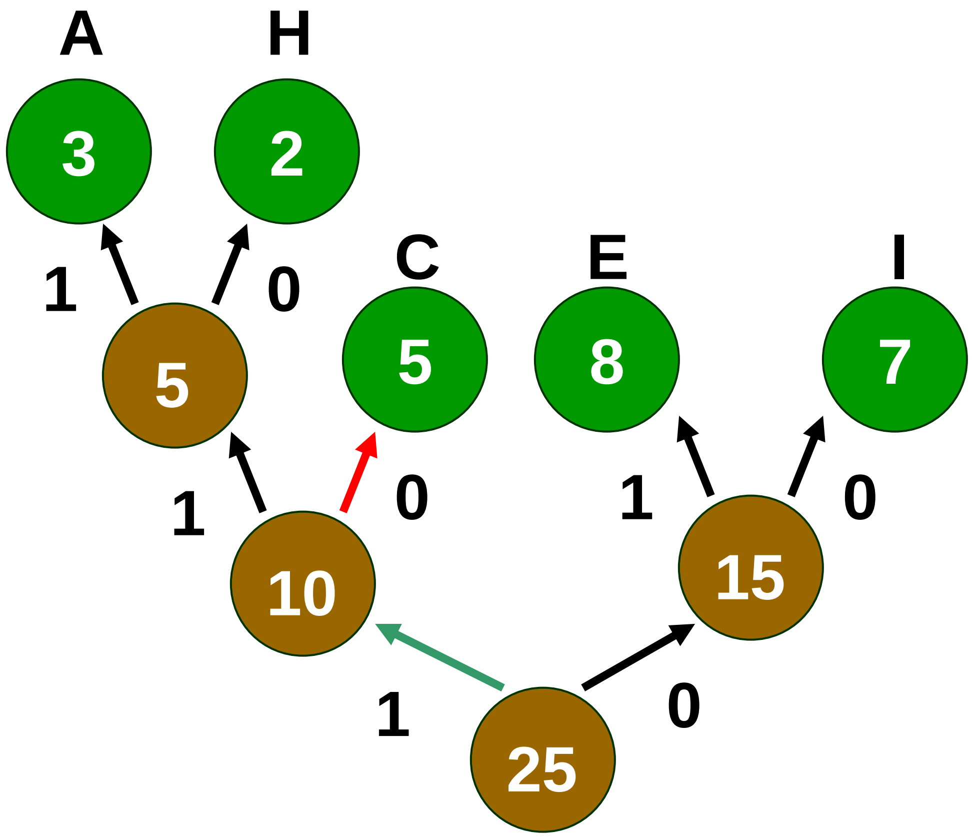
Huffman Trees Application to Coding



1111001

A

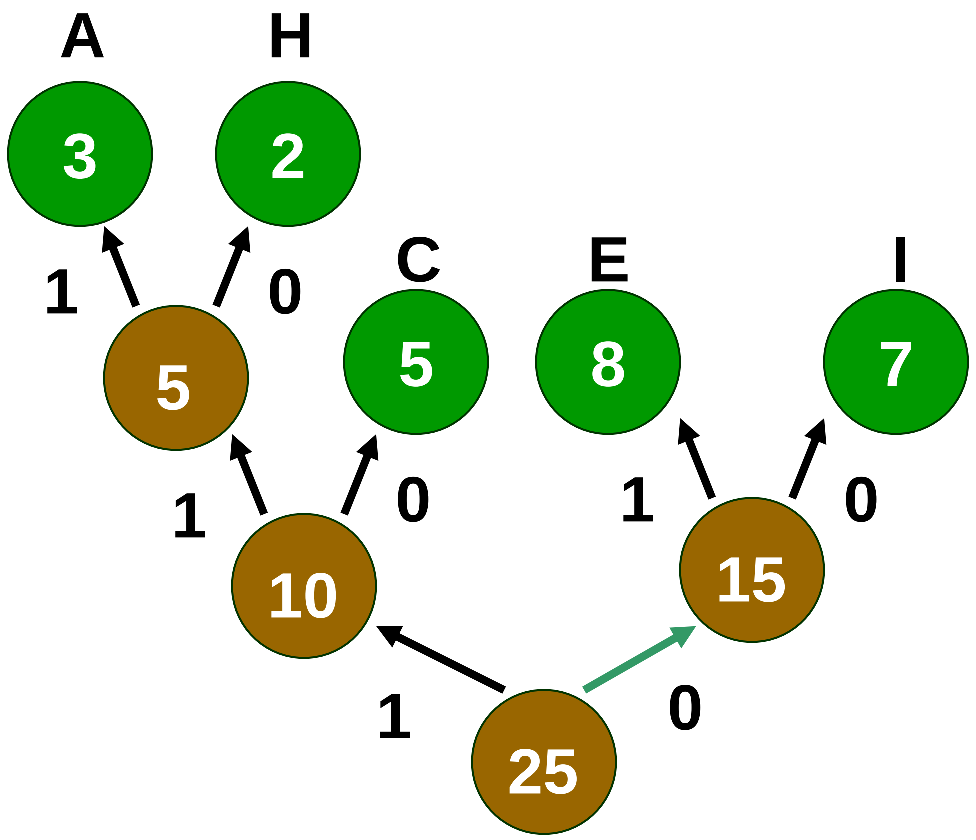
Huffman Trees Application to Coding



1111001

AC

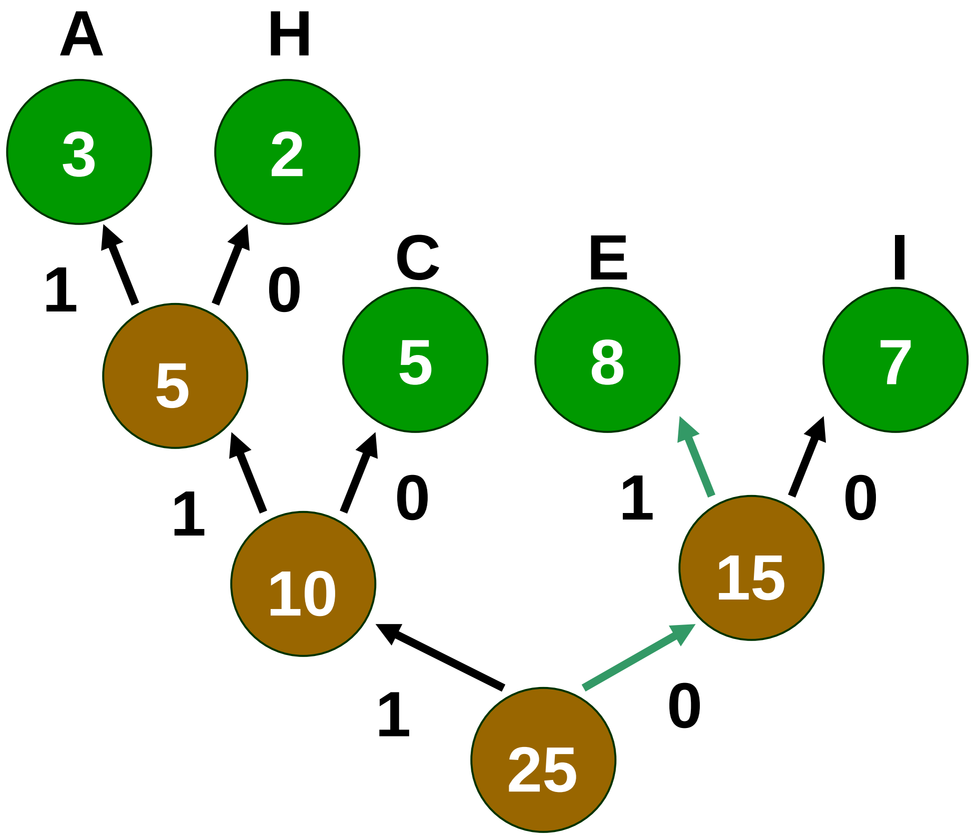
Huffman Trees Application to Coding



1111001

AC

Huffman Trees Application to Coding



1111001

ACE

General Trees

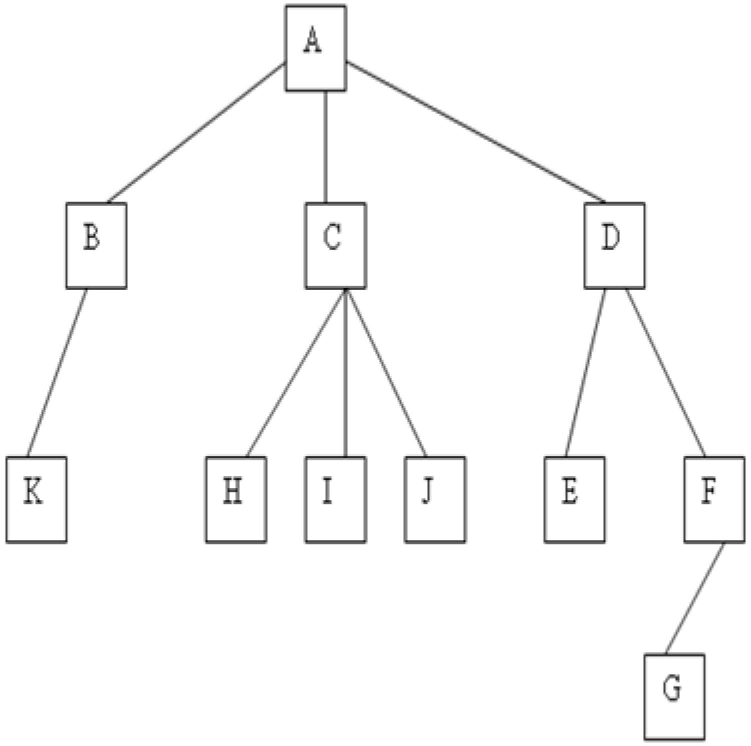
- A general tree is defined to be a nonempty finite set T of elements called nodes, such as:
  - T contains a distinguished elements R, called the root of the tree
  - The remaining elements of T from an ordered collection of zero or more disjoint trees.

$$T_1, T_2 , \dots , T_m$$

  The trees $T_1, T_2 , \dots , T_m$ are called sub-trees of the root R.

- The number of sub-trees for any node may be variable.
  - Some nodes may have 1 or no sub-trees, others may have 3, some 4, or any other combination

General Trees



References

1. Data Structures, Seymour Lipschutz, Schaum's Outline Series.
2. Fundamentals of Data Structures, E. Horowitz and S. Sahni.
3. Julia Rahman, Assistant Professor, Dept. of CSE, RUET