BLM5106- Advanced Algorithm Analysis and Design

H. İrem Türkmen

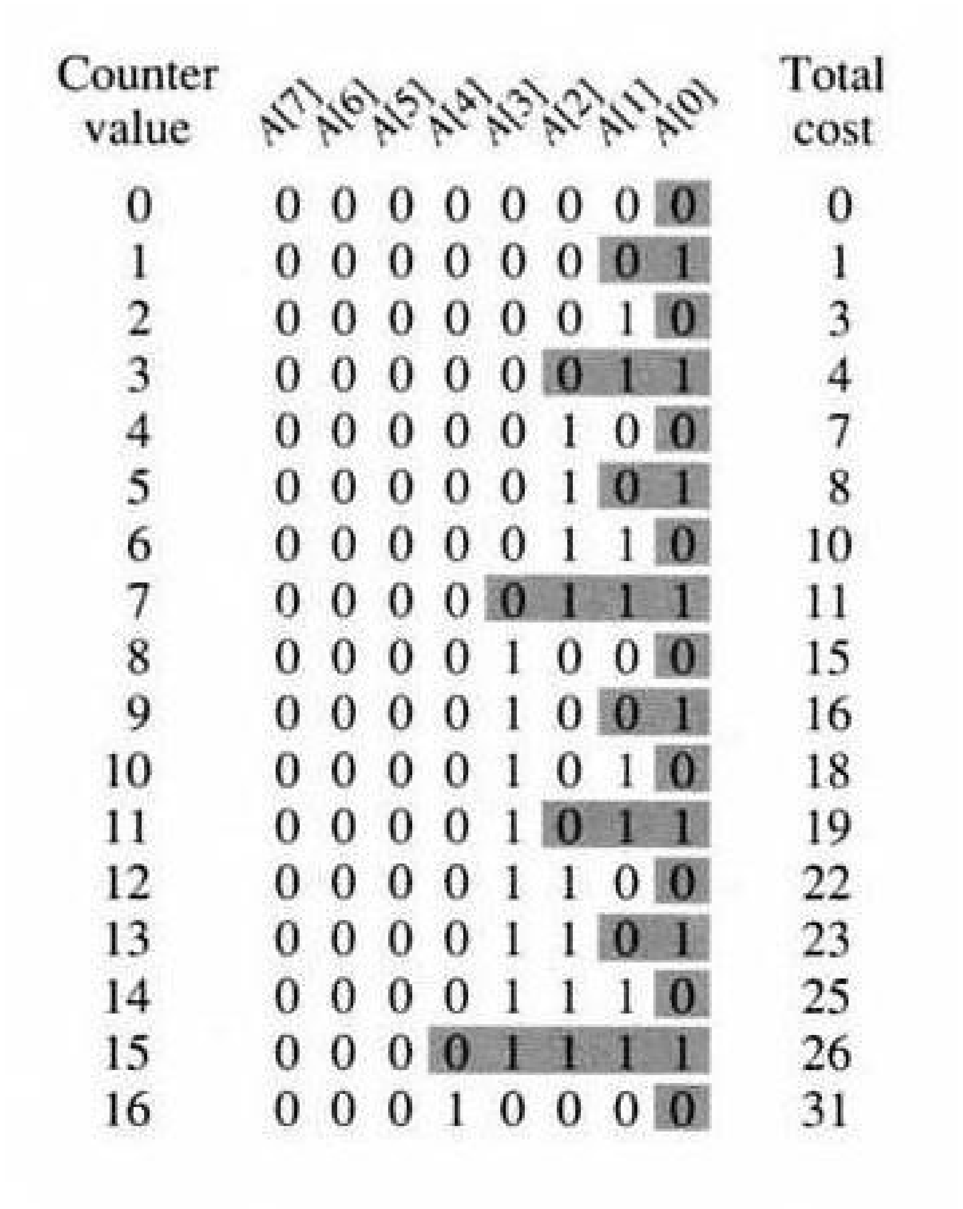
Amortized Analysis

- **Aggregate analysis**, in which we determine an upper bound T(n) on the total cost of a sequence of n operations. The average cost per operation is then

T (n)/n

- **The accounting method**
- **Potential method**

Amortized Analysis



Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Trees

- Dr. B. Prabhakaran, University of Texas at Dallas, Department of Computer Science
- Dr. Bina Ramamurthy, University at Buffalo (UB)
- Dr. Carl Kingsford, Carnegie Mellon University, School of Computer Science

Application Examples

- Useful for locating items in a list
- Used in Huffman coding algorithm
- Study games like checkers and chess to determine winning strategies
- Weighted trees used to model various categories of problems

Trees

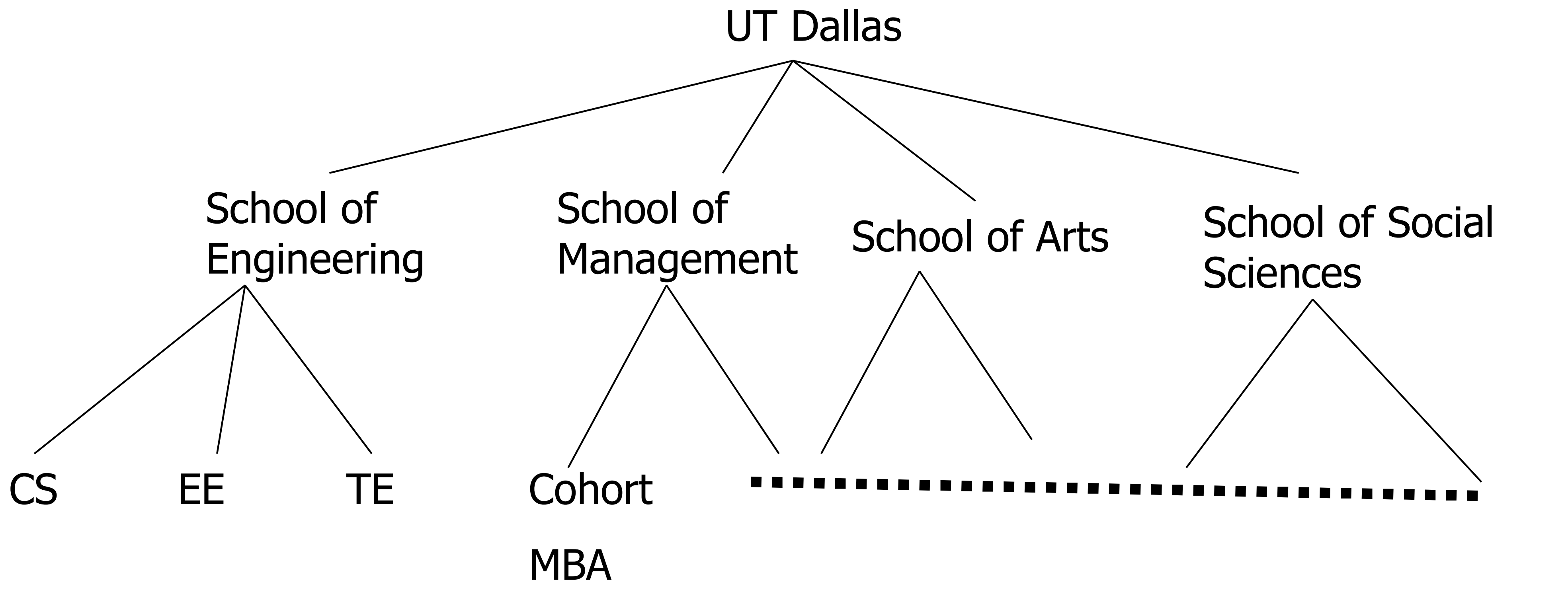
Definition: A connected undirected graph with no simple circuits

Characteristics

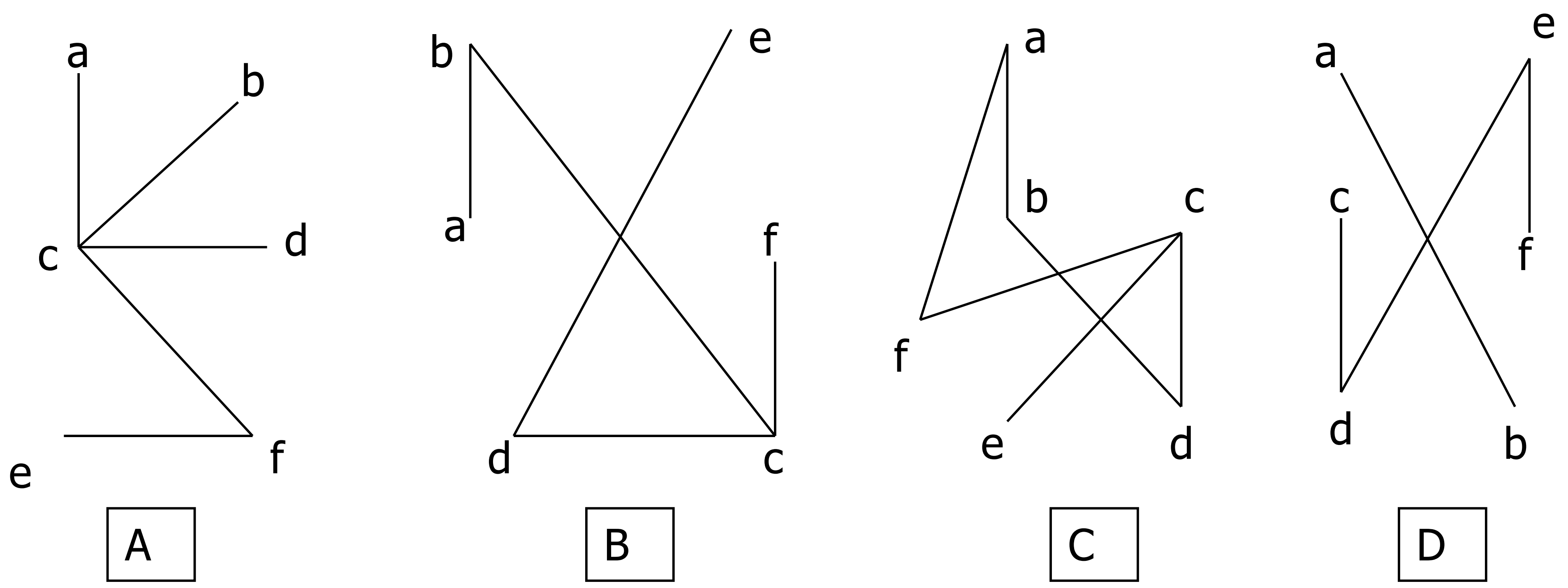
- No multiple edges

- No loops

Example


UT Dallas
School of Engineering
School of Management
School of Arts
School of Social Sciences
CS
EE
TE
Cohort
MBA

Which of the following graphs are trees?



A & B : Trees

C: Not a Tree (Cycle abdcfa)

D: Not a Tree (Not connected). However it is a *forest* i.e. unconnected graphs

Tree Theorem

Theorem 1 : An undirected Graph is a tree if and only if there is a unique simple path between any two of its vertices

Proof (=>) The graph is a Tree and it does not have cycles. If there were cycles then there will be more than one simple path between two vertices. Hence, there is a unique simple path between every pair of vertices

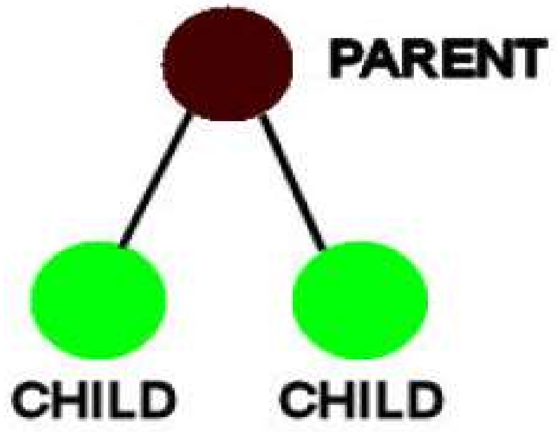
Tree terminology

Rooted tree: one vertex designated as root and every edge directed away

Parent: *u* is the parent of *v* iff (if and only if) there is an edge directed from *u* to *v*

Child: *v* is called the child of *u*

Every non-root node has a unique parent (?)



Tree terminology

Siblings: vertices with same parent

Ancestors: all vertices on path from the root to this vertex, excluding the vertex

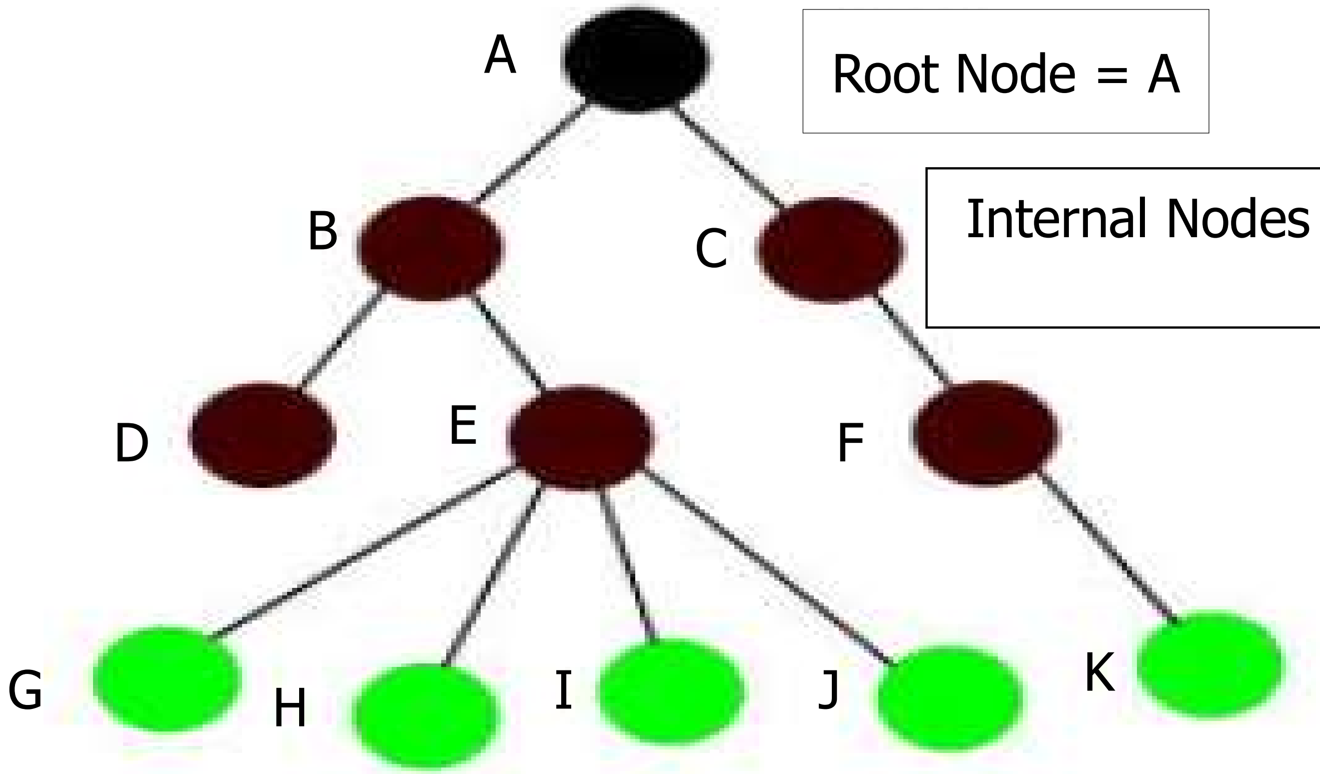
Descendants: Descendants of vertex v are all vertices that have v as an ancestor

Internal Nodes: Nodes that have children

External or Leaf Nodes: Nodes that have no children

Given : Tree rooted at A

Find: Descendants (B), Ancestor (A), Siblings (G) ?

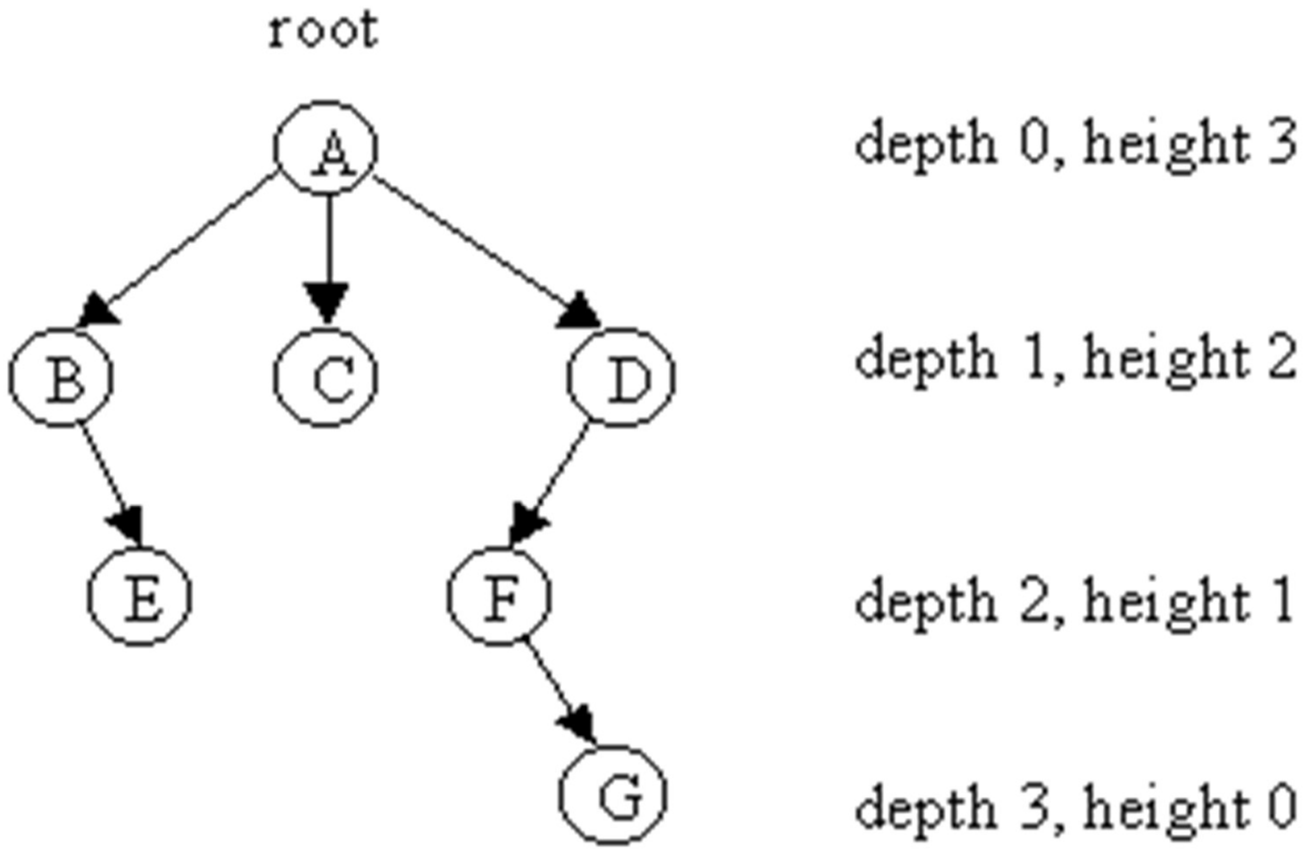


Root Node = A

Internal Nodes = B, C, E and F

External Nodes = D,G, H ,I, J and K

Definition – Height/Depth(Level)

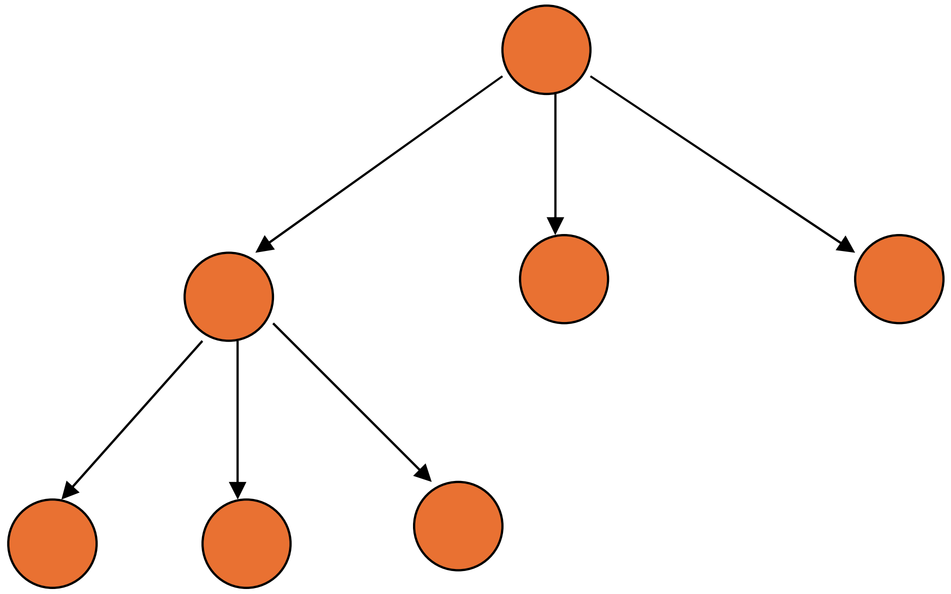


A tree of height 3

Definition: m-ary trees

- Rooted tree where every vertex has no more than ‘m’ children
- Full m-ary if every internal vertex has exactly ‘m’ children (i.e., except leaf/external vertices).
- m=2 gives a binary tree

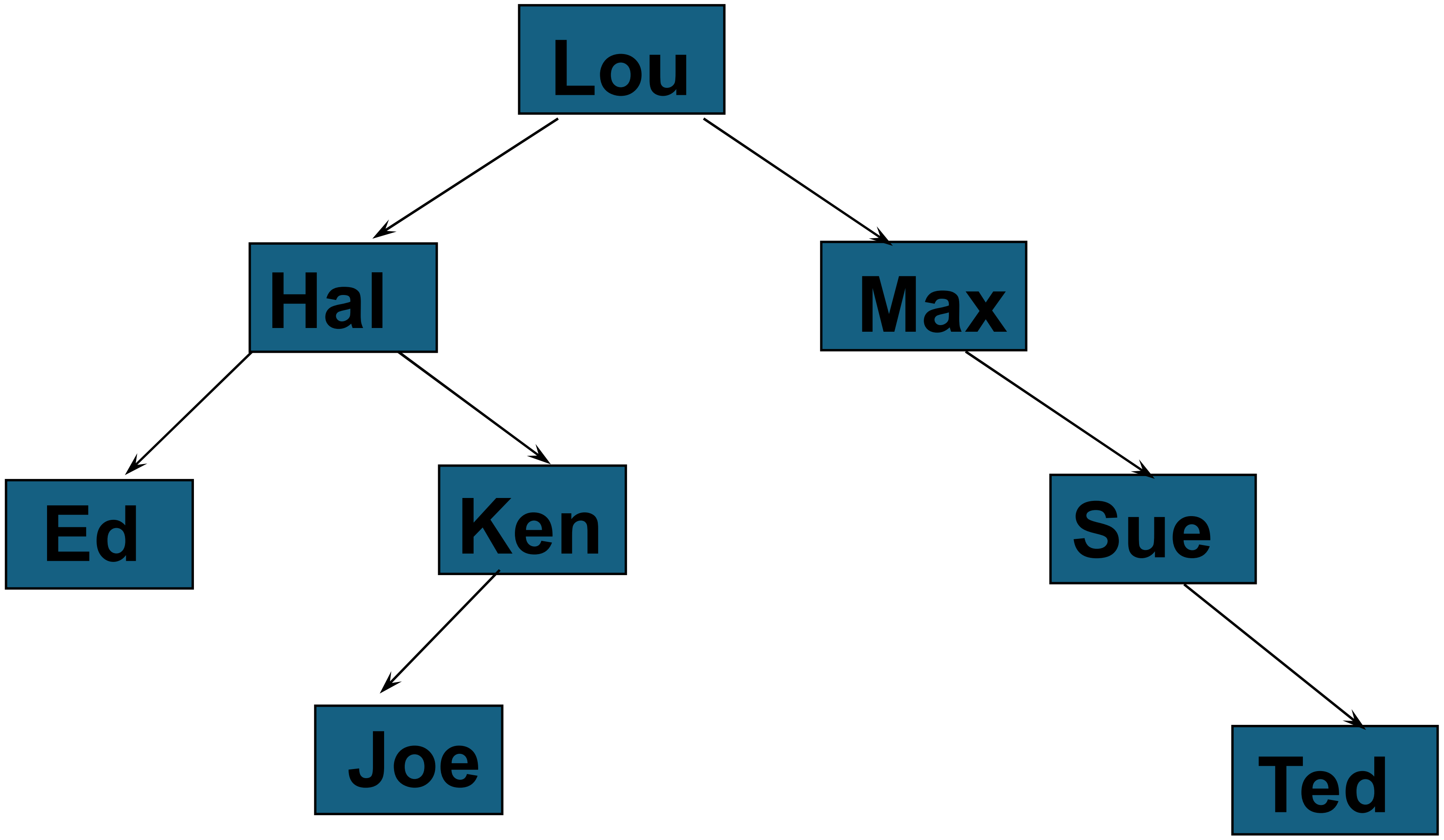
Example: 3-ary tree



Definition: Binary Tree

- Every internal vertex has a maximum of 2 children
- An ordered rooted tree is a rooted tree where the children of each internal vertex are ordered.
- In an ordered binary tree, the two possible children of a vertex are called the left child and the right child, if they exist.

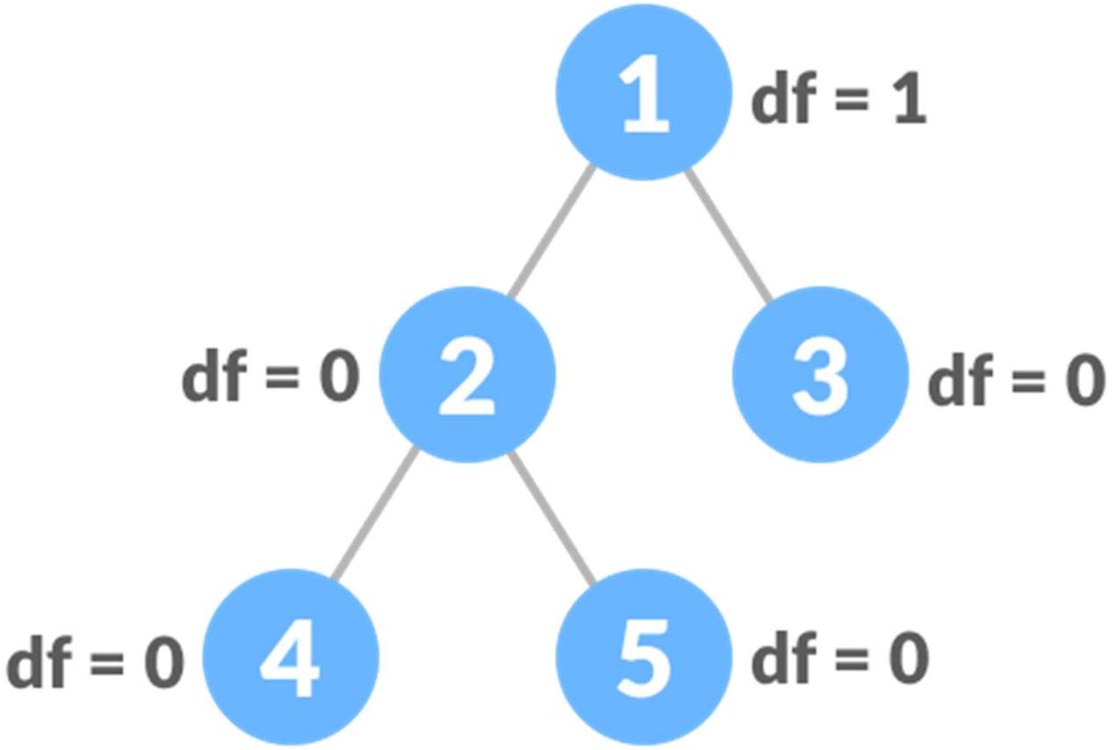
An Ordered Binary Tree

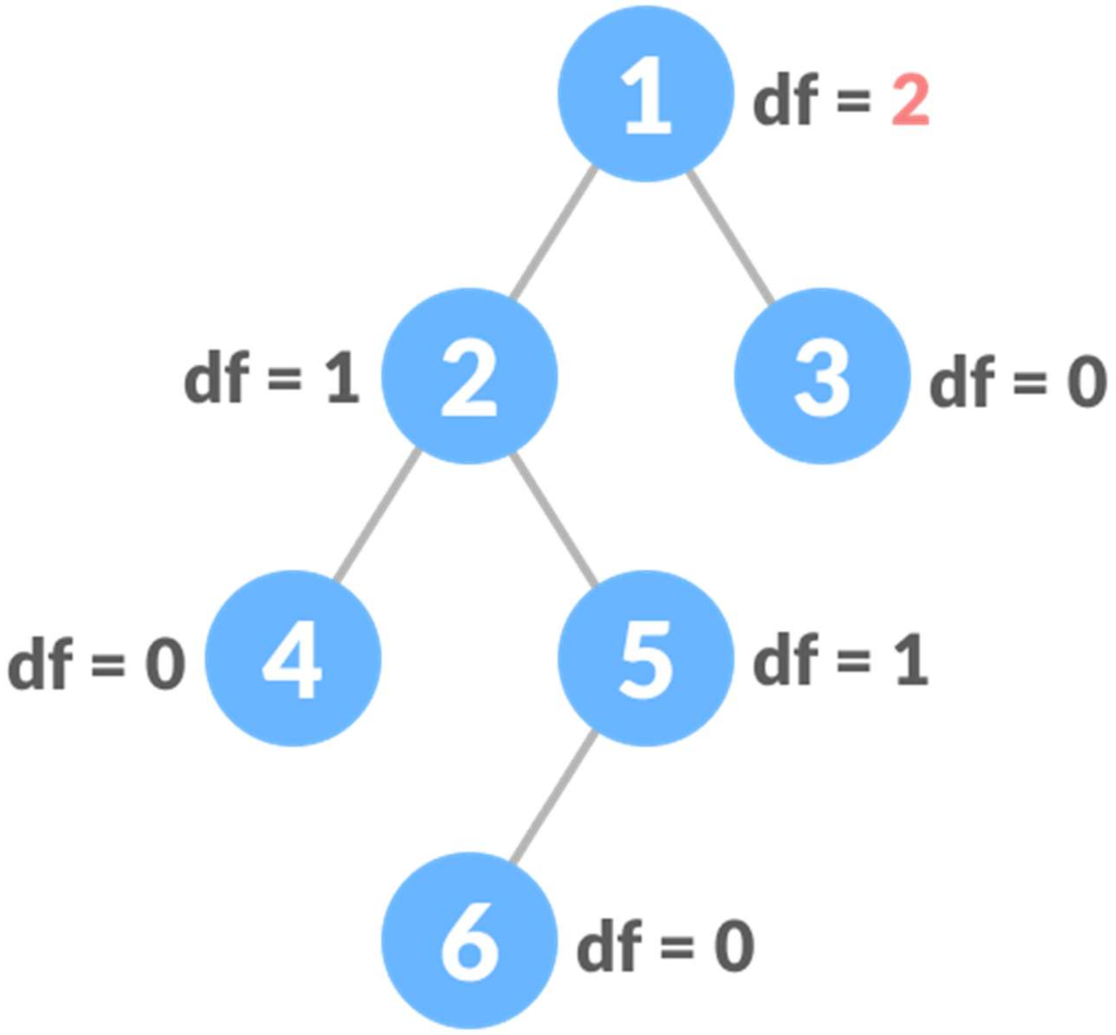


Definition: Balanced

- A balanced binary tree is a binary tree in which the height of the left and right subtrees of any node differs by no more than one.
- Formally, a binary tree is balanced if, for every node in the tree:

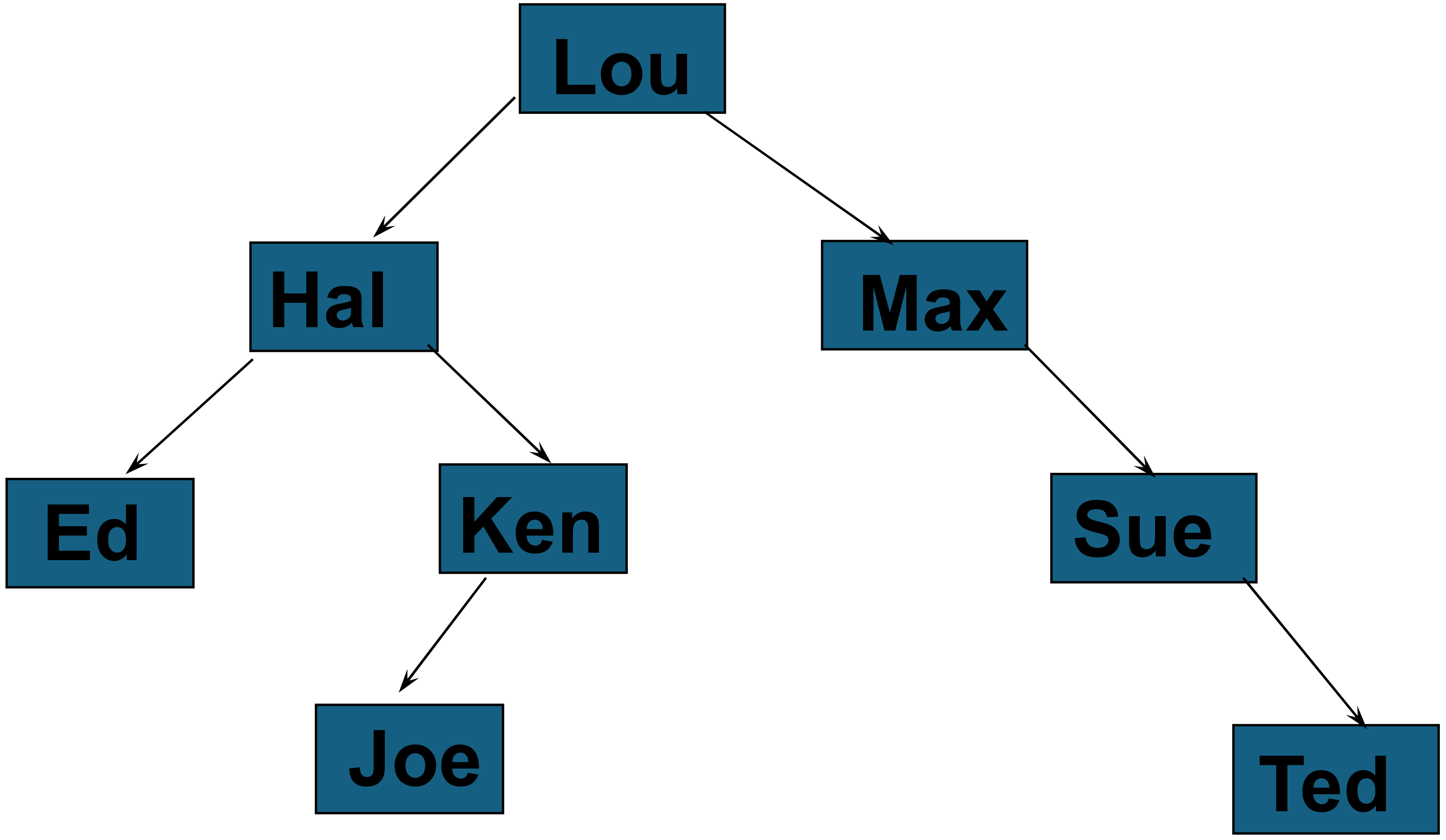
$|\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})| \leq 1$$|\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})| \leq 1$


1
df = 1
2
df = 0
3
df = 0
4
df = 0
5
df = 0



df = |height of left child - height of right child|

Is this tree balanced?



Tree Property 1

Theorem 2 : A tree with n vertices has n-1 edges

Proof: By induction

Basis Step: for n = 1 there are (1 – 1) 0 edges

Inductive Step: We assume that n vertices has n – 1 edges . We need to prove that n + 1 vertices have n edges.

A tree with n + 1 vertices can be achieved by adding an edge from a vertex of the tree to the new vertex (why 1 edge ?). The total number of edges is (n – 1 + 1 = n).

Tree Property 2

Theorem 3: A full m-ary tree with i internal vertices contains n = mi +1 vertices

Proof: i internal vertices have m children. Therefore, we have mi vertices. Since the root is not a child we have n = mi + 1 vertices.

Tree Property 3

Theorem 4: There are at most m^h leaves in a m-ary tree of height h

Proof: By Induction

Basic Step: m-ary tree of height 1 => leaves = m which is true since the root has m-ary children

l0 ->1 =m^0

l1 -> m =m^1

l2 -> m*m =m^2

l3 -> m*m *m = m^3

Applications of trees

- Binary Search Trees
- Decision Trees
- Prefix Codes
- Game Trees

Binary Search Trees (BST)

Applications

- A Binary Search Tree can be used to store items in its vertices
- It enables efficient searches

Definition: BST

A special kind of binary tree in which:

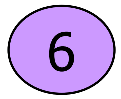
- Each vertex contains a distinct key value,
- The key values in the tree can be compared using “greater than” and “less than”, and
- The key value of each vertex in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree.

Shape of a binary search tree

Depends on its key values and their order of insertion.

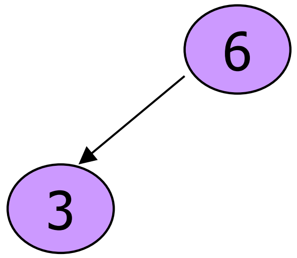
Example: Insert the elements 6, 3, 4, 2, 10 in that order.

The first value to be inserted is put into the root



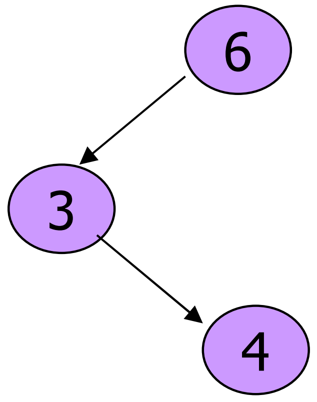
Shape of a binary search tree

Inserting 3 into the BST



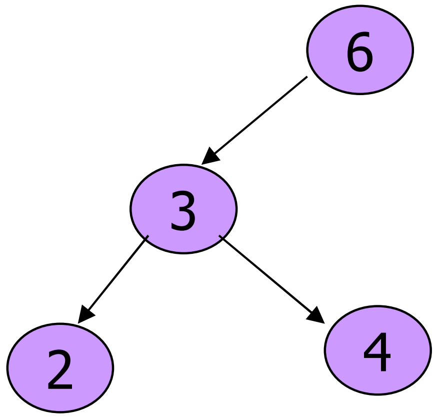
Shape of a binary search tree

Inserting 4 into the BST



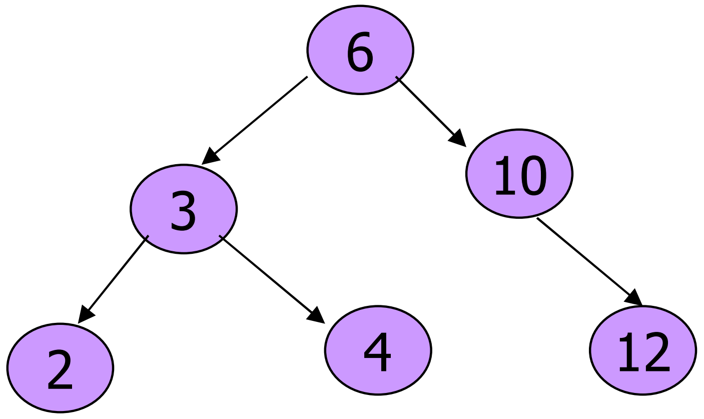
Shape of a binary search tree

Inserting 4 into the BST


$$\begin{tikzcd}
 & & 6 \arrow[dl] \\
 & 3 \arrow[dl] \arrow[dr] & \\
2 & & 4
\end{tikzcd}$$

Shape of a binary search tree

Inserting 10 into the BST



Codes

Text Encoding

Our next goal is to develop a code that represents a given text as compactly as possible.

A standard encoding is ASCII, which represents every character using 7 bits:

“An English sentence” = 133 bits ≈ 17 bytes

1000001 (A) 1101110 (n) 0100000 ( ) 1000101 (E) 1101110 (n) 1100111 (g)1101100 (l) 1101001 (i) 1110011 (s) 1101000 (h)

0100000 ( ) 1110011 (s) 1100101 (e) 1101110 (n) 1110100 (t) 1100101 (e) 1101110 (n) 1100011 (c)
1100101 (e)

Codes

Text Encoding

Of course, this is wasteful because we can encode 12 characters in 4 bits:

‹space› = 0000   A = 0001   E = 0010   c = 0011   e = 0100 g = 0101   h = 0110
i = 0111   l = 1000   n = 1001   s = 1010   t = 1011

Then we encode the phrase as

0001 (A)   1001 (n)   0000 ( )   0010 (E)   1001 (n)   0101 (g)   1000 (l)   0111 (i)
1010 (s)   0110 (h)   0000 ( )   1010 (s)   0100 (e)   1001 (n)   1011 (t)   0100 (e)
1001 (n)   0011 (c)   0100 (e)

This requires 76 bits ≈ 10 bytes

Codes

Text Encoding

An even better code is given by the following encoding:

‹space› = 000   A = 0010   E = 0011   s = 010   c = 0110   g = 0111
h = 1000 i = 1001   l = 1010   t = 1011   e = 110   n = 111

Then we encode the phrase as

0010 (A)   111 (n)   000 ( )   0011 (E)   111 (n)   0111 (g)   1010 (l)
1001 (i) 010 (s)   1000 (h)   000 ( ) 010 (s)   110 (e)   111 (n)   1011
(t)   110 (e)   111 (n) 0110 (c)   110 (e)

This requires 65 bits ≈ 9 bytes

Codes

Codes That Can Be Decoded

Fixed-length codes:

- *Every character is encoded using the same number of bits.*
- To determine the boundaries between characters, we form groups of *w* bits, where *w* is the length of a character.
- **Examples:**
  - ASCII
  - Our first improved code

Prefix codes:

- *No character is the prefix of another character.*
- **Examples:**
  - Fixed-length codes
  - Huffman codes

Why Prefix Codes ?

- Consider a code that is not a prefix code:

  a = 01   m = 10   n = 111   o = 0   r = 11   s = 1   t = 0011

- Now you send a fan-letter to your favorite movie star. One of the sentences is “You are a star.”

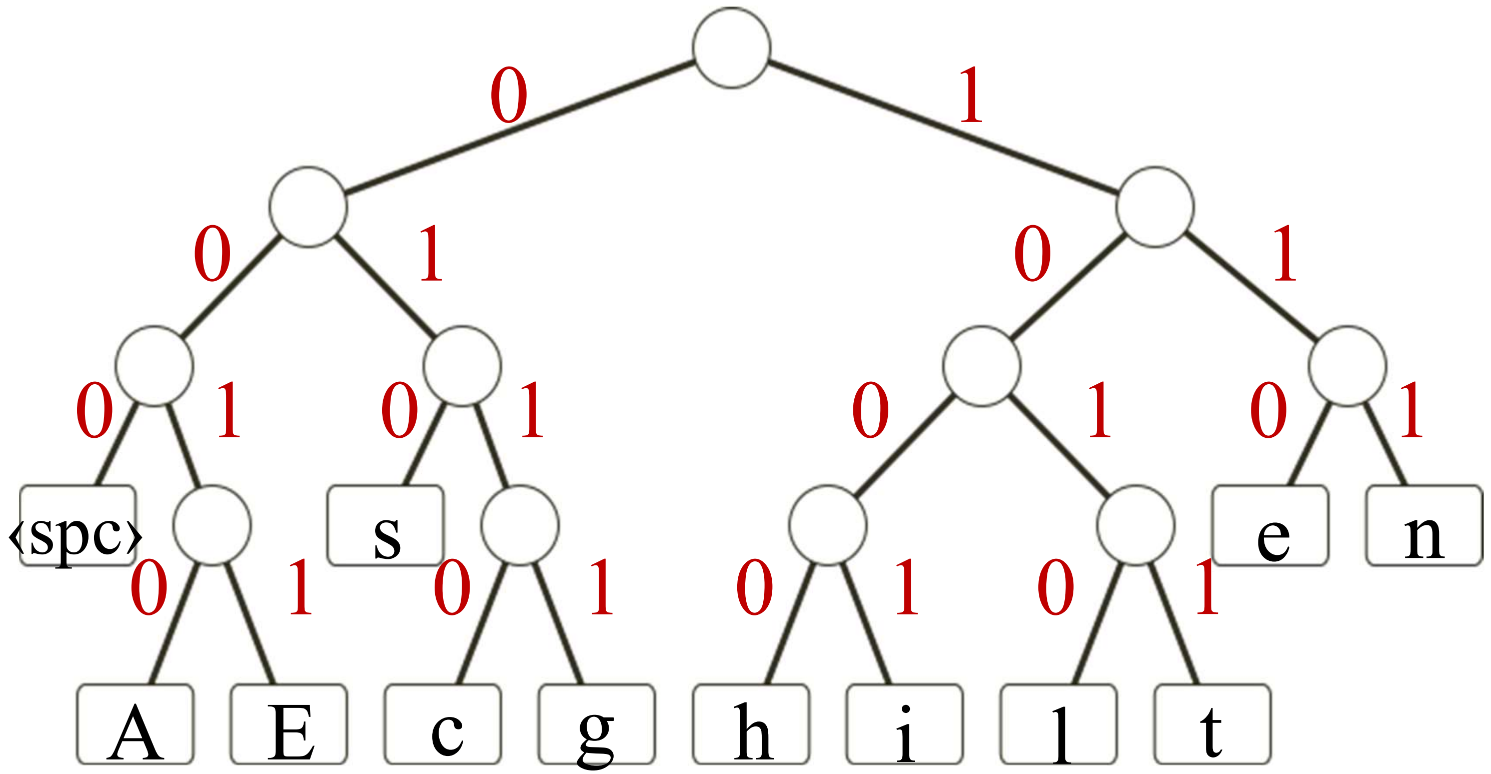
  You encode “star” as “1 0011 01 11”.

- Your idol receives the letter and decodes the text using your coding table:

  100110111 = 10 0 11 0 111 = “moron”

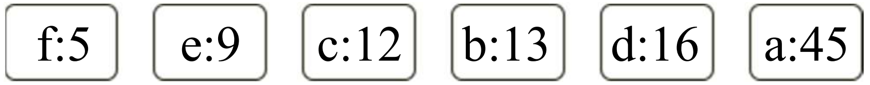
Representing a Prefix-Code Dictionary

Our example: ‹space› = 000 A = 0010 E = 0011 s = 010 c = 0110 g = 0111 h = 1000 i = 1001 l = 1010 t = 1011 e = 110 n = 111



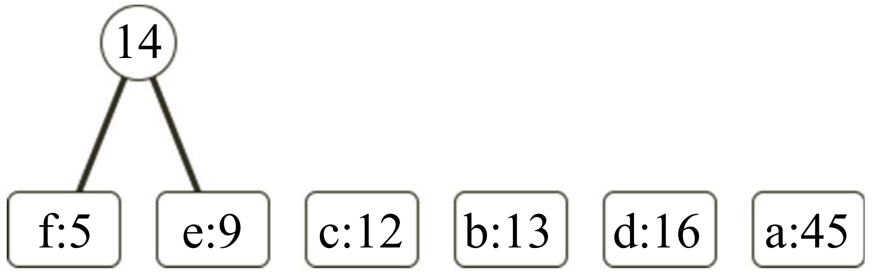
Huffman’s Algorithm

```
Huffman($C$)
$n \leftarrow |C|$
$Q \leftarrow C$
for $i = 1..n - 1$
    do allocate a new node $z$
    left[$z$] $\leftarrow x \leftarrow$ Delete-Min($Q$)
    right[$z$] $\leftarrow y \leftarrow$ Delete-Min($Q$)
    $f[z] \leftarrow f[x] + f[y]$
    Insert($Q$, $z$)
return Delete-Min($Q$)
```



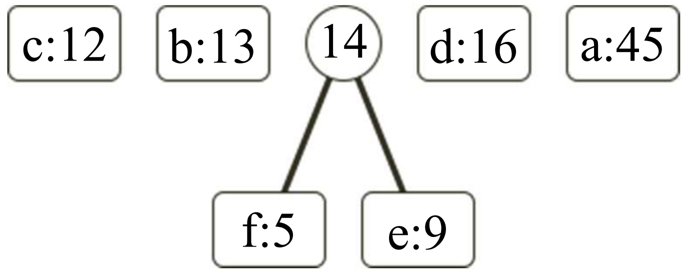
Huffman's Algorithm

```
Huffman($C$)
1   $n \leftarrow |C|$
2   $Q \leftarrow C$
3   for $i = 1..n-1$
4   do allocate a new node $z$
5       left[$z$] $\leftarrow x \leftarrow$ Delete-Min($Q$)
6       right[$z$] $\leftarrow y \leftarrow$ Delete-Min($Q$)
7       $f[z] \leftarrow f[x] + f[y]$
8       Insert($Q$, $z$)
9   return Delete-Min($Q$)
```



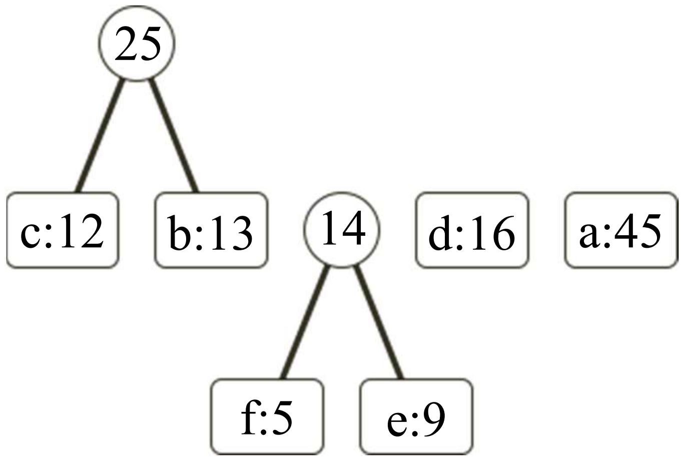
Huffman's Algorithm

```
Huffman(C)
1    $n \leftarrow |C|$
2    $Q \leftarrow C$
3    for $i = 1..n - 1$
4    do allocate a new node $z$
5    left[$z$] $\leftarrow x \leftarrow$ Delete-Min($Q$)
6    right[$z$] $\leftarrow y \leftarrow$ Delete-Min($Q$)
7    $f[z] \leftarrow f[x] + f[y]$
8    Insert($Q$, $z$)
9    return Delete-Min($Q$)
```



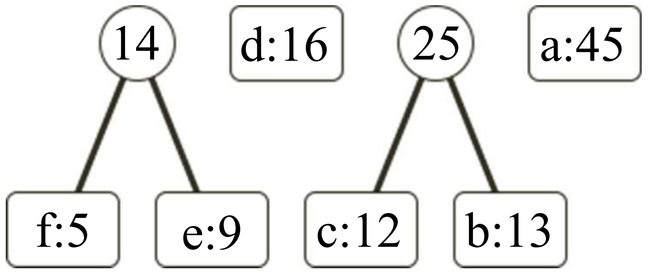
Huffman's Algorithm

```
Huffman($C$)
1    $n \leftarrow |C|$
2    $Q \leftarrow C$
3    for $i = 1..n - 1$
4    do allocate a new node $z$
5    left[$z$] $\leftarrow x \leftarrow$ Delete-Min($Q$)
6    right[$z$] $\leftarrow y \leftarrow$ Delete-Min($Q$)
7    $f[z] \leftarrow f[x] + f[y]$
8    Insert($Q$, $z$)
9    return Delete-Min($Q$)
```



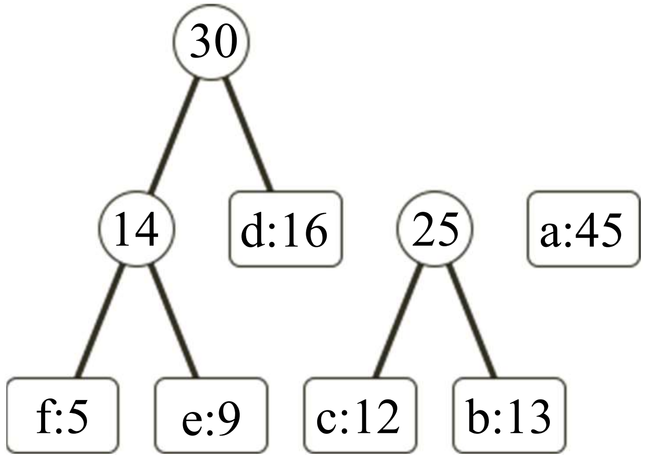
Huffman’s Algorithm

```
Huffman(C)
1    $n \leftarrow |C|$
2    $Q \leftarrow C$
3    for $i = 1..n - 1$
4    do allocate a new node $z$
5       left[$z$] $\leftarrow x \leftarrow$ Delete-Min($Q$)
6       right[$z$] $\leftarrow y \leftarrow$ Delete-Min($Q$)
7       $f[z] \leftarrow f[x] + f[y]$
8       Insert($Q$, $z$)
9    return Delete-Min($Q$)
```



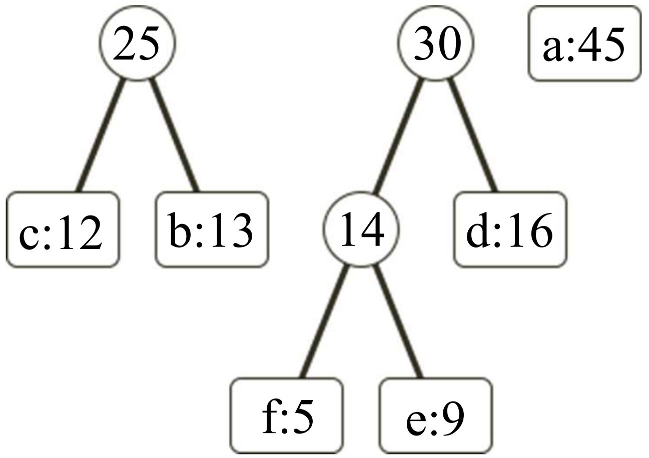
Huffman's Algorithm

```
Huffman(C)
1    n ← |C|
2    Q ← C
3    for i = 1..n − 1
4    do allocate a new node z
5        left[z] ← x ← Delete-Min(Q)
6        right[z] ← y ← Delete-Min(Q)
7        f[z] ← f[x] + f[y]
8        Insert(Q, z)
9    return Delete-Min(Q)
```



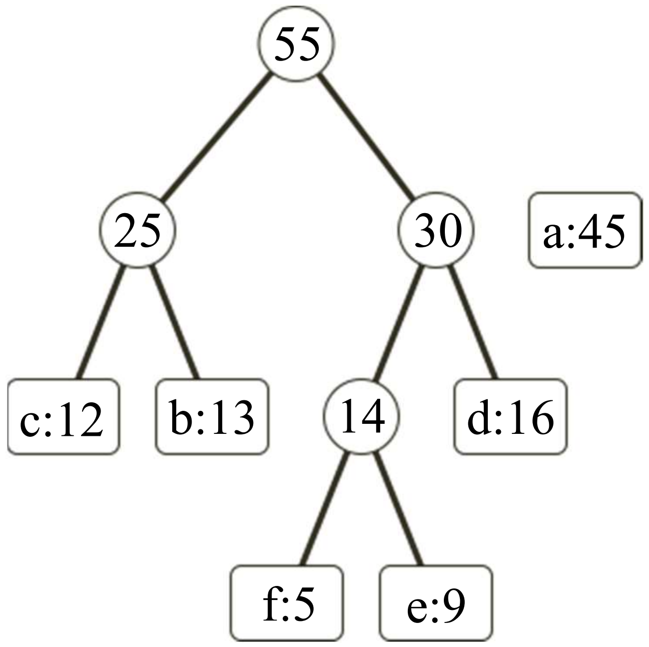
Huffman's Algorithm

```
Huffman($C$)
1     $n \leftarrow |C|$
2     $Q \leftarrow C$
3     for $i = 1..n - 1$
4     do allocate a new node $z$
5     left[$z$] $\leftarrow x \leftarrow$ Delete-Min($Q$)
6     right[$z$] $\leftarrow y \leftarrow$ Delete-Min($Q$)
7     $f[z] \leftarrow f[x] + f[y]$
8     Insert($Q$, $z$)
9     return Delete-Min($Q$)
```



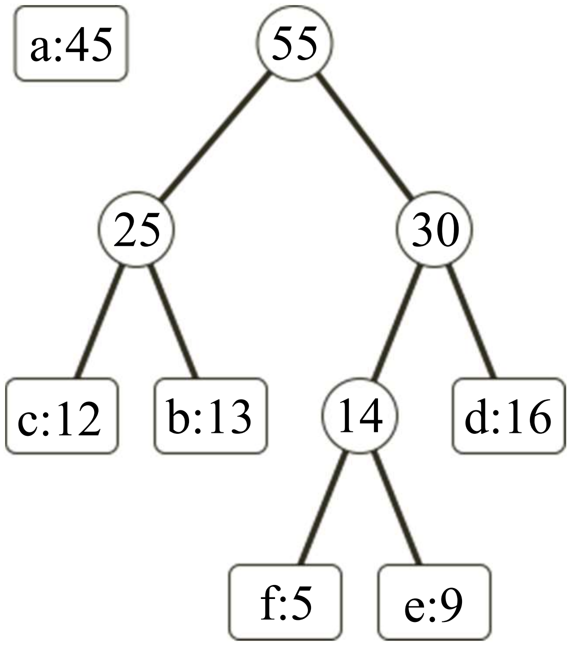
Huffman's Algorithm

```
Huffman($C$)
1    $n \leftarrow |C|$
2    $Q \leftarrow C$
3    for $i = 1..n - 1$
4    do allocate a new node $z$
5       left[$z$] $\leftarrow x \leftarrow$ Delete-Min($Q$)
6       right[$z$] $\leftarrow y \leftarrow$ Delete-Min($Q$)
7       $f[z] \leftarrow f[x] + f[y]$
8       Insert($Q$, $z$)
9    return Delete-Min($Q$)
```



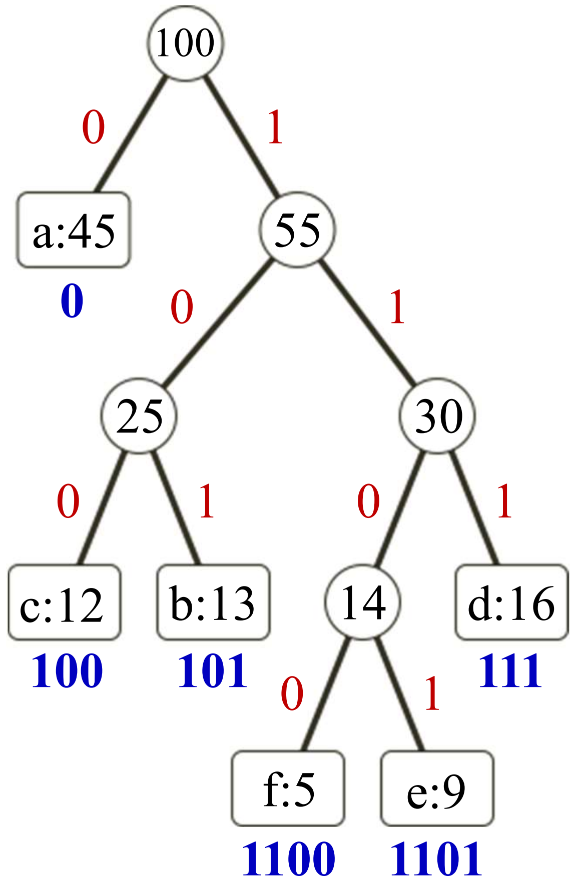
Huffman's Algorithm

```
Huffman(C)
1    n ← |C|
2    Q ← C
3    for i = 1..n − 1
4    do allocate a new node z
5    left[z] ← x ← Delete-Min(Q)
6    right[z] ← y ← Delete-Min(Q)
7    f[z] ← f[x] + f[y]
8    Insert(Q, z)
9    return Delete-Min(Q)
```

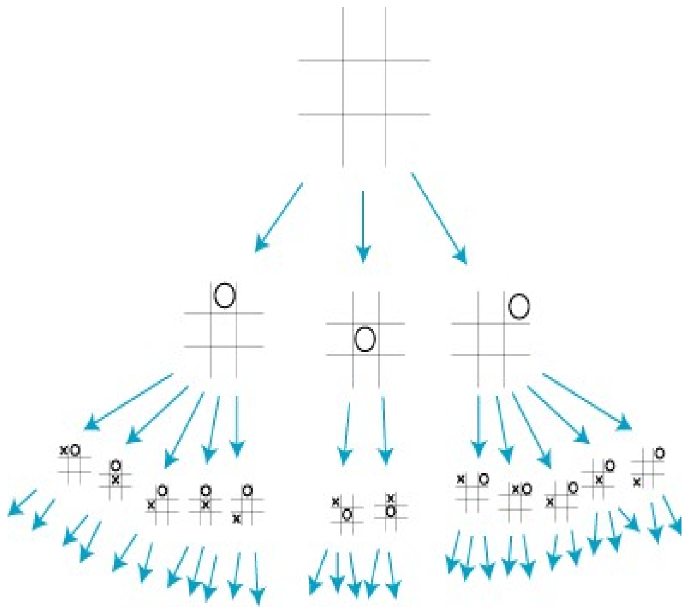


Huffman's Algorithm

```
Huffman(C)
1    $n \leftarrow |C|$
2    $Q \leftarrow C$
3    for $i = 1..n - 1$
4    do allocate a new node $z$
5    left[$z$] $\leftarrow x \leftarrow$ Delete-Min($Q$)
6    right[$z$] $\leftarrow y \leftarrow$ Delete-Min($Q$)
7    $f[z] \leftarrow f[x] + f[y]$
8    Insert($Q$, $z$)
9    return Delete-Min($Q$)
```

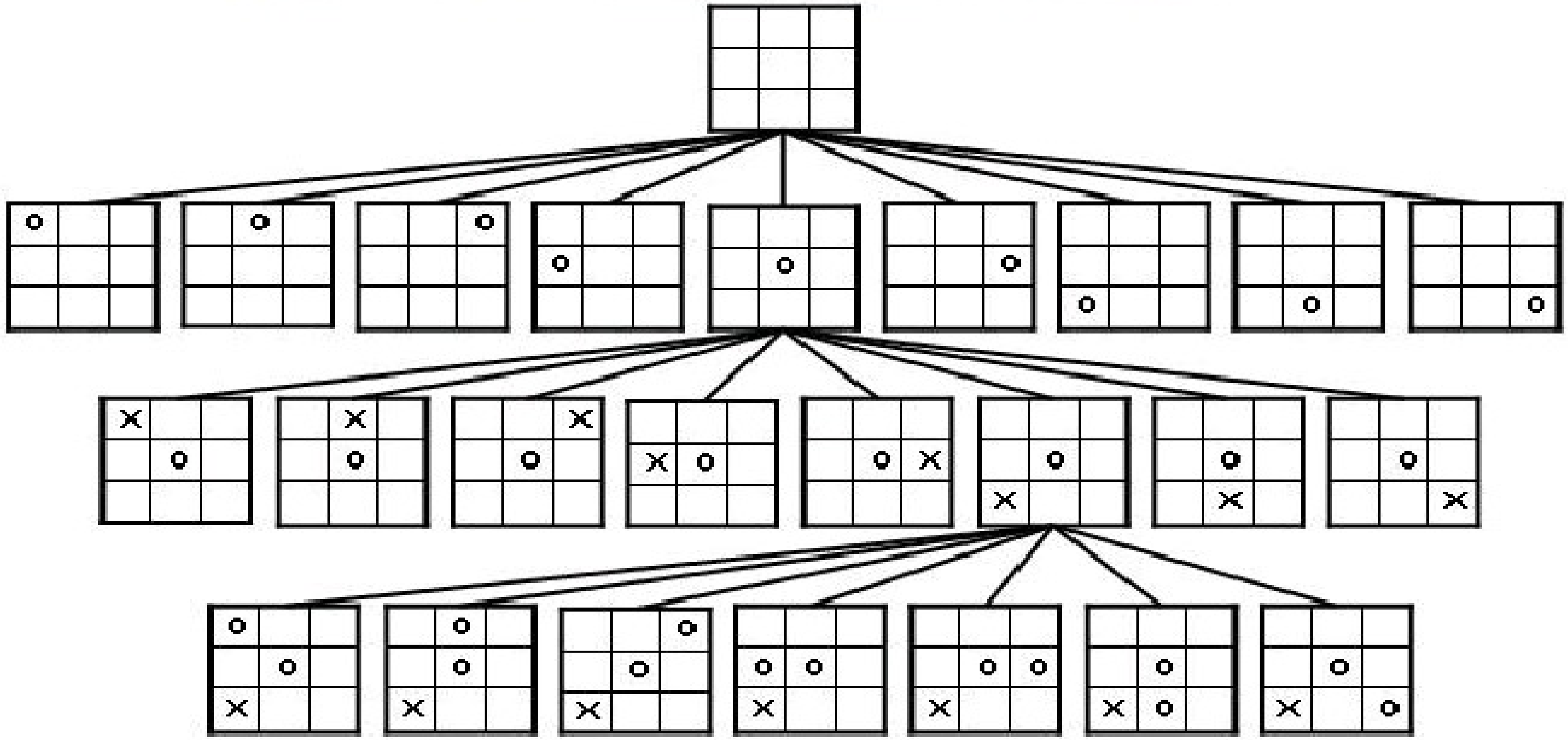


Game Trees



- Writing down all the possible moves by both players down onto a tree
- Root node represents the current status
- **Internal nodes at even levels** correspond to positions in which the first player is to move (MAX nodes)
- **Internal nodes at odd levels** correspond to positions in which the second player is to move (Min nodes)
- **Leaves** correspond to positions at which the game has ended. One player or the other has won, or perhaps the game is drawn.

Game Trees: Partial Game Tree for Tic-Tac-Toe



Here's a little piece of the game tree for Tic-Tac-Toe, starting from an empty board. Note that even for this trivial game, the search tree is quite big.

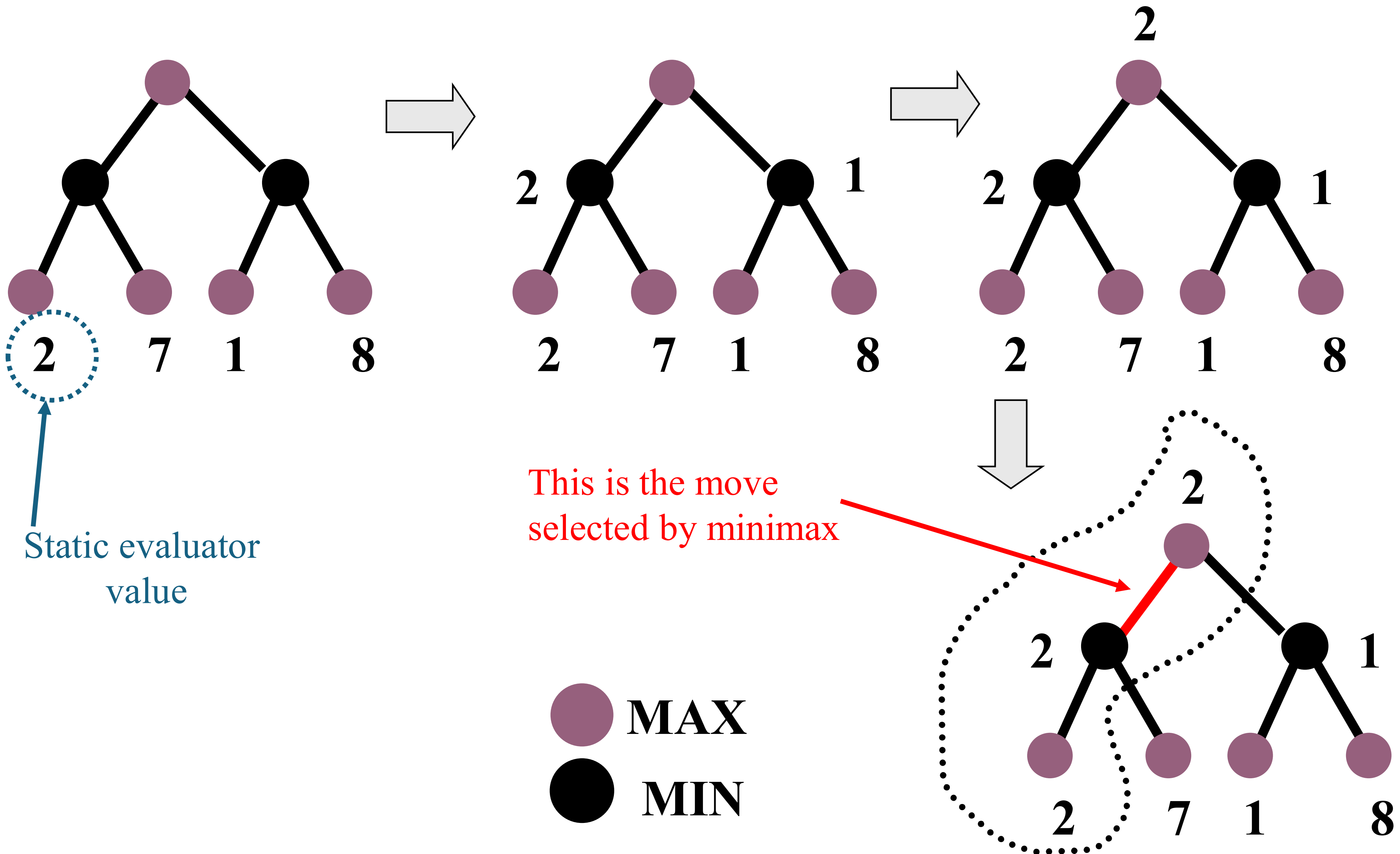
Game Trees

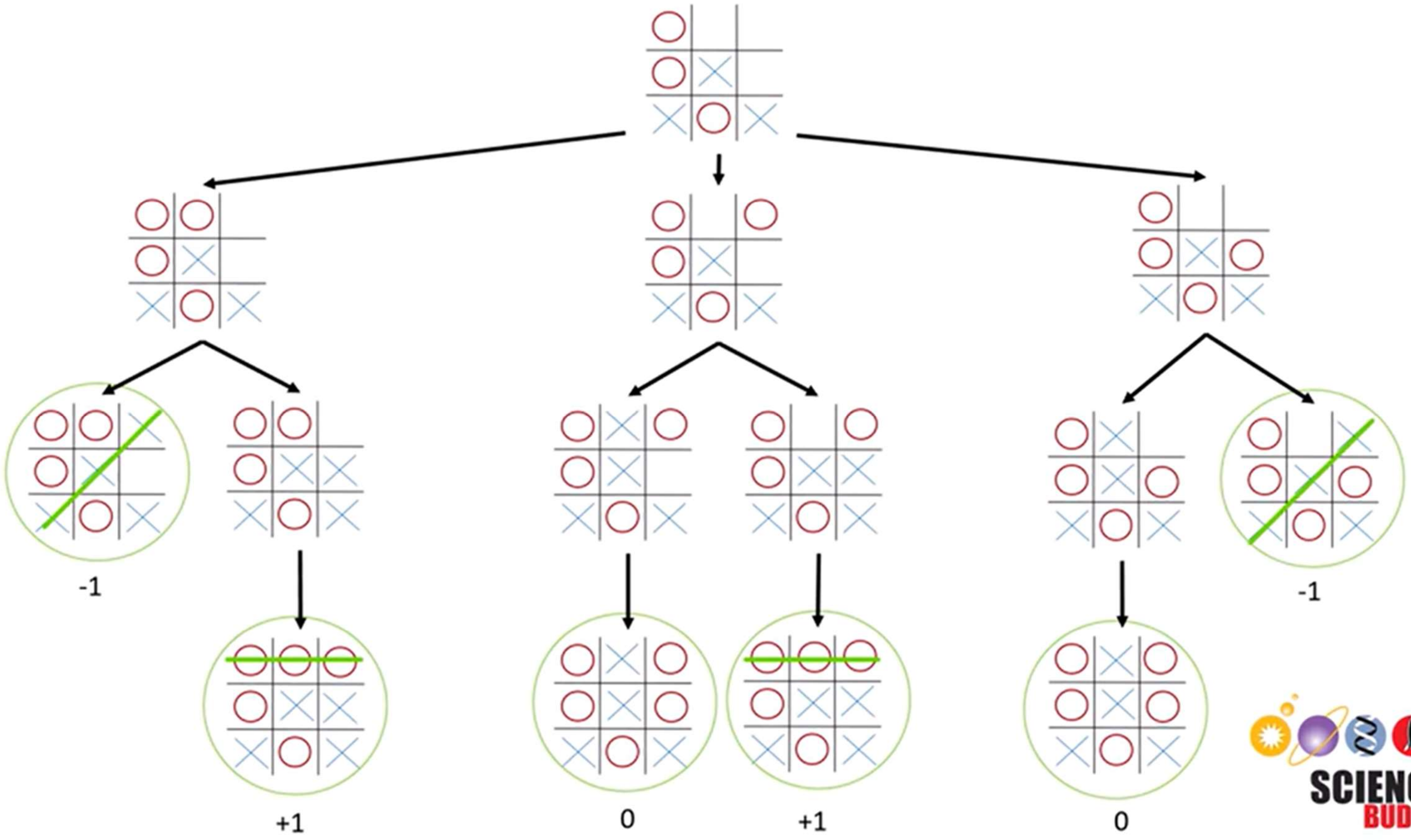
- In case of TIC TAC TOE, there are 9 choices for the first player
- Next, the opponent have 8 choices, ..., and so on
- Number of nodes = 1 + 9 + 9*8 + 9*8*7 + ... +9*8*7*6*5*4*3*2*1 = ???
- Exponential order of nodes
- If the depth of the tree is m and there are b moves at each node, the time complexity of evaluating all the nodes are $O(b^m)$
- Even in case of TICTACTOE, there are so many nodes

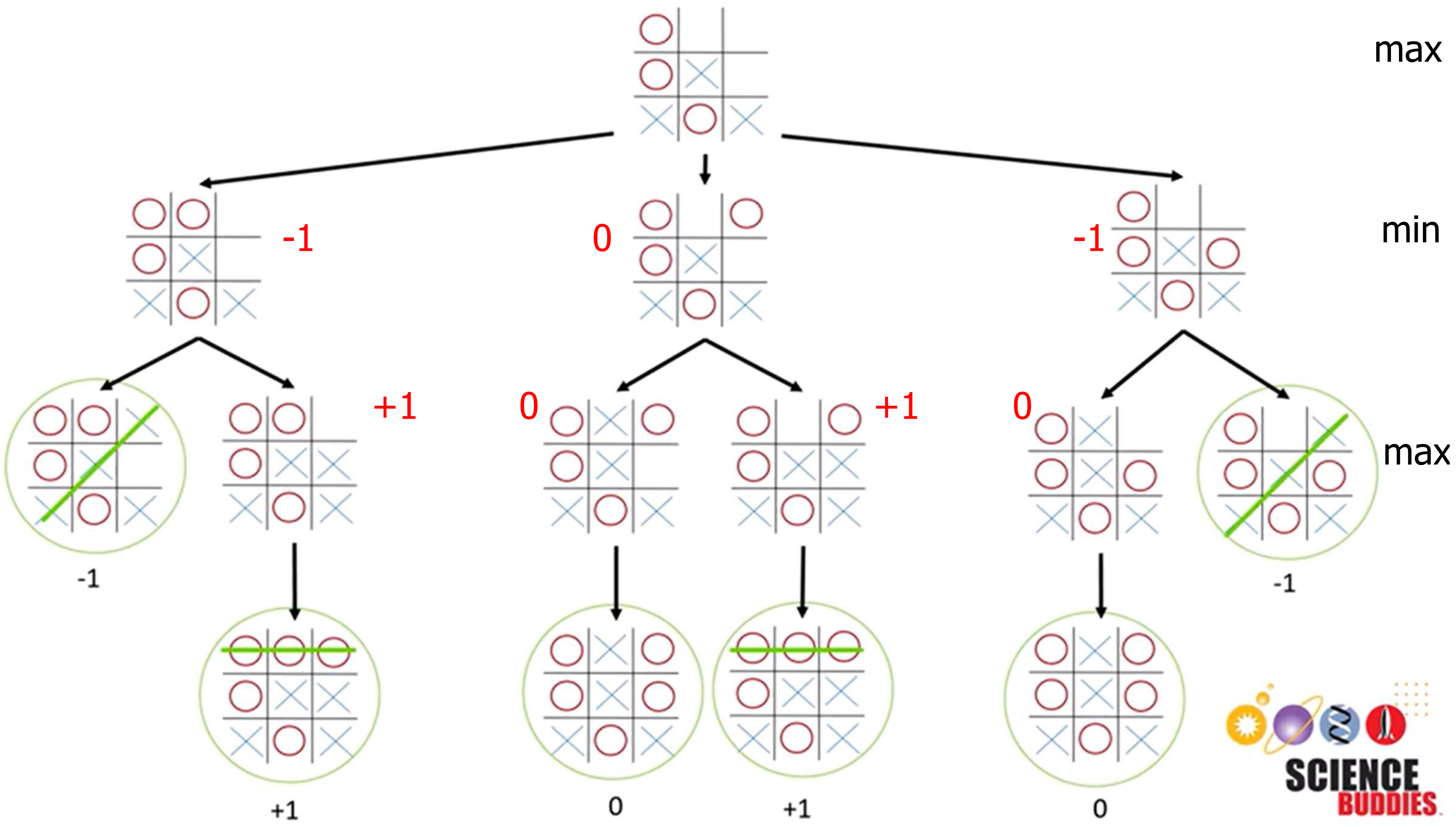
Game Trees: Min-Max Procedure

- So instead of expanding the whole tree, we use a min-max strategy
- Starting from the current game position as the node, expand the tree to some depth
- Apply the evaluation function at each of the leaf nodes
- “Back up” values for each of the non-leaf nodes until a value is computed for the root node
    - At MIN nodes, the backed-up value is the **minimum** of the values associated with its children.
    - At MAX nodes, the backed-up value is the **maximum** of the values associated with its children.
- Pick the edge at the root associated with the child node whose backed-up value determined the value at the root

Game Trees: Minimax Search




-1
+1
0
+1
0
-1


max
min
max
-1
0
-1
+1
0
+1
0
-1
+1
0
+1
-1
0

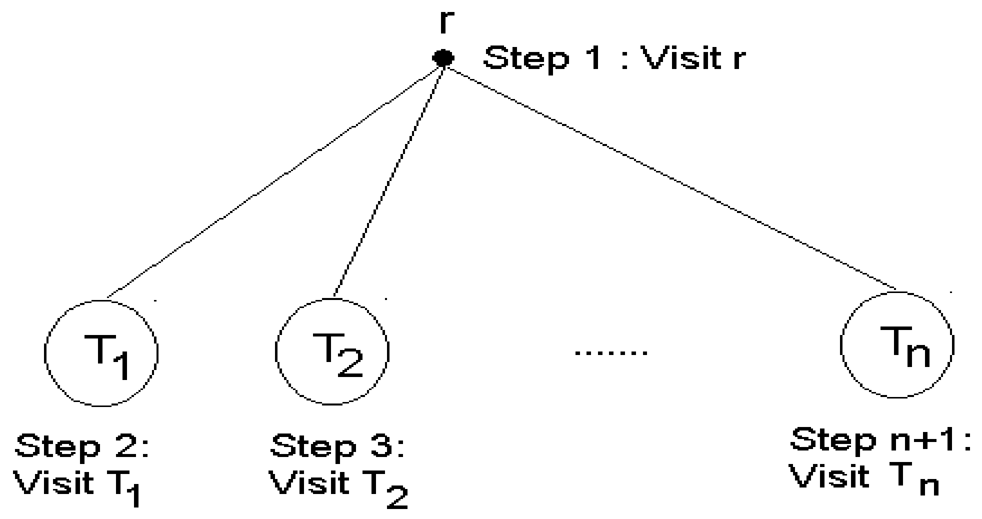
Tree Traversal

- Information is stored in Ordered Rooted tree
- Methods are needed for visiting each vertex of the tree in order to access data
- Three most commonly used traversal algorithms are:
    1. Preorder Traversal
    2. Inorder Traversal
    3. Postorder Traversal

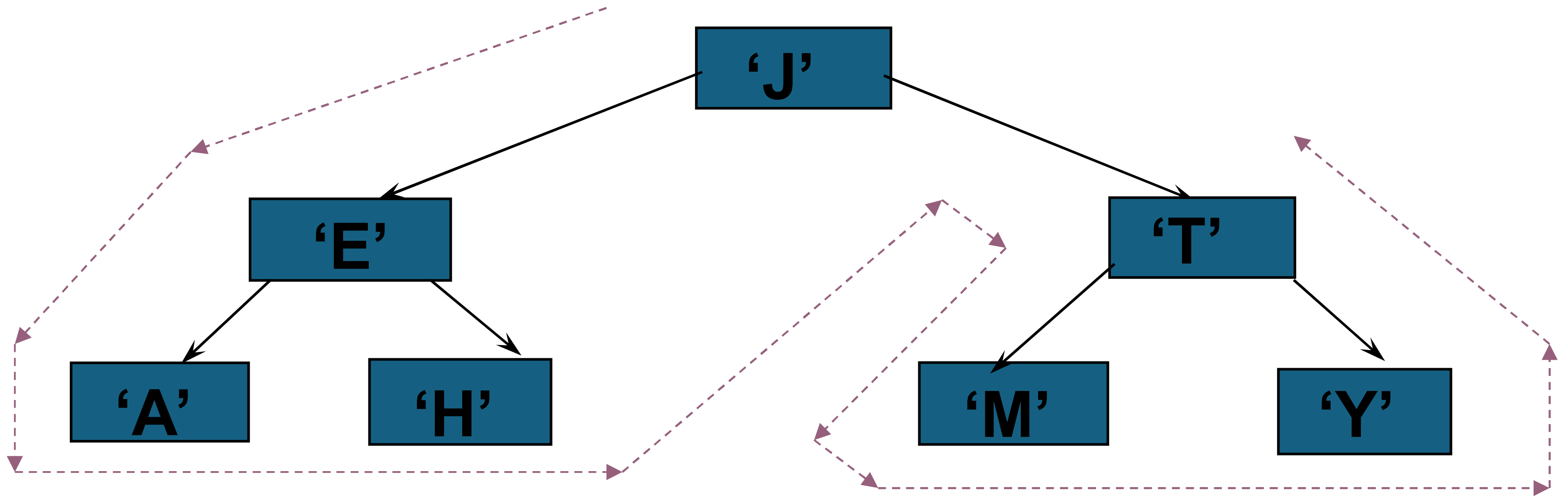
Preorder Traversal

Let T be an ordered binary tree with root r

If T has only r, then r is the preorder traversal. Otherwise, suppose T1, T2 are the left and right subtrees at r. The preorder traversal begins by visiting r. Then traverses T1 in preorder, then traverses T2 in preorder.



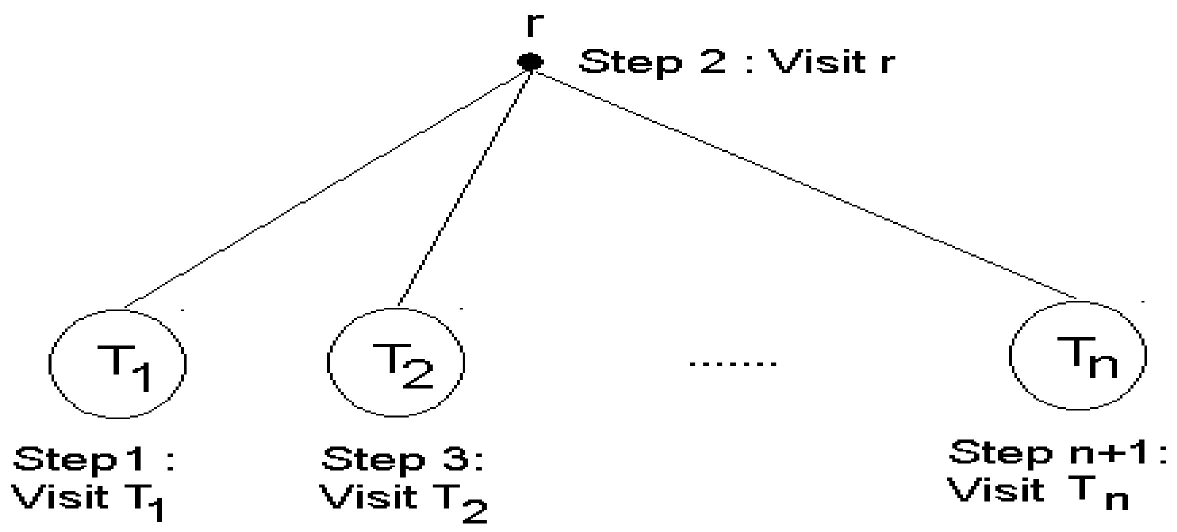
Preorder Traversal Example: J E A H T M Y



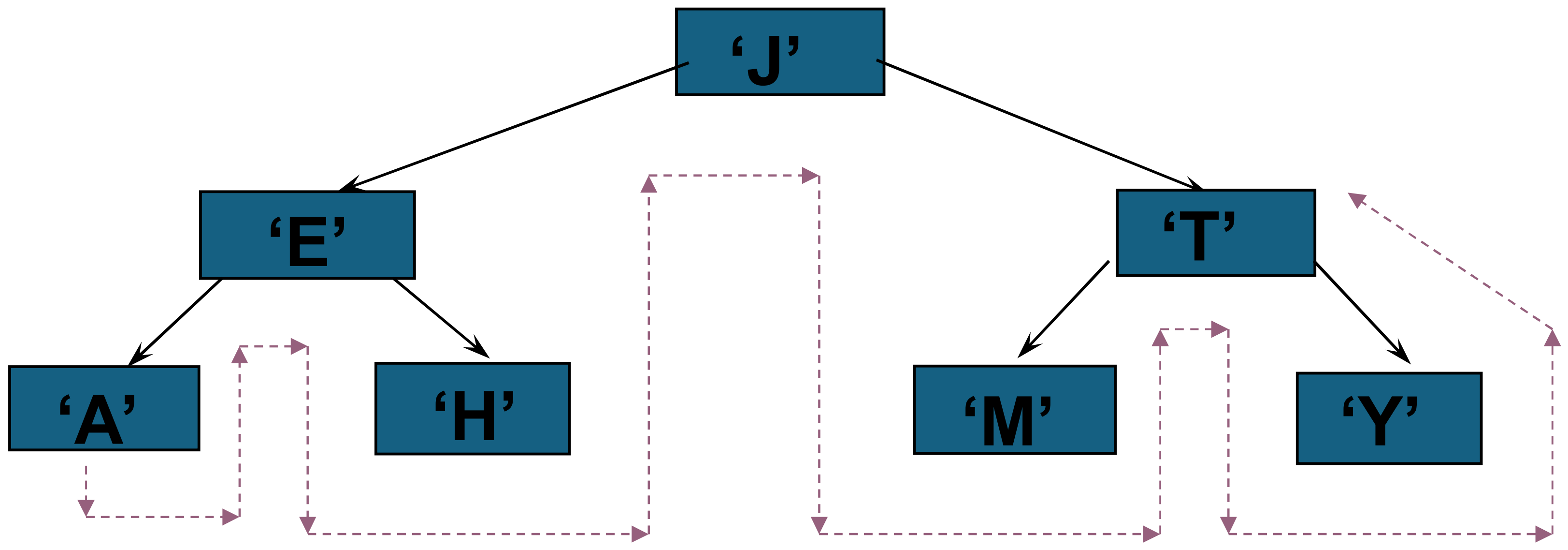
Inorder Traversal

Let T be an ordered binary tree with root r.

If T has only r, then r is the inorder traversal. Otherwise, suppose T1, T2 are the left and right subtrees at r. The inorder traversal begins by traversing T1 in inorder. Then visits r, then traverses T2 in inorder.



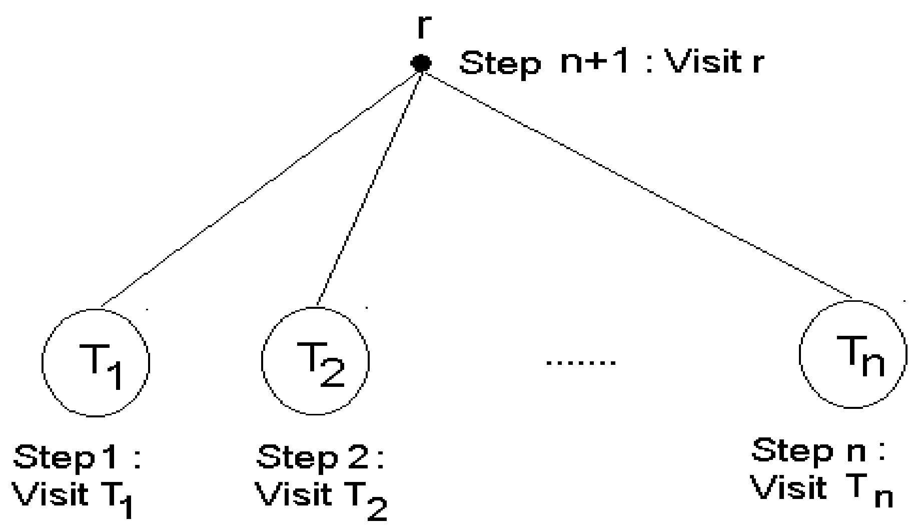
Inorder Traversal: A E H J M T Y



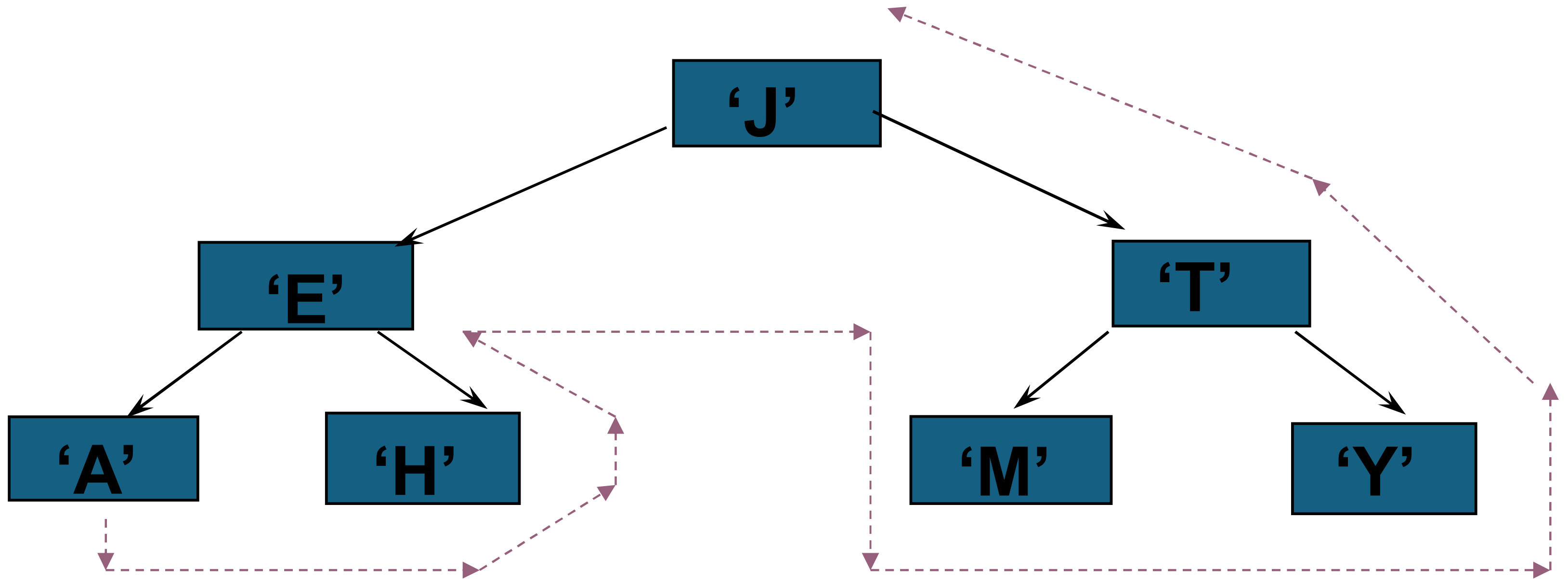
Postorder Traversal

Let T be an ordered binary tree with root r.

If T has only r, then r is the postorder traversal. Otherwise, suppose T1, T2 are the left and right subtrees at r. The postorder traversal begins by traversing T1 in postorder. Then traverses T2 in postorder, then ends by visiting r.



Postorder Traversal: A H E M Y T J



Binary Expression Tree

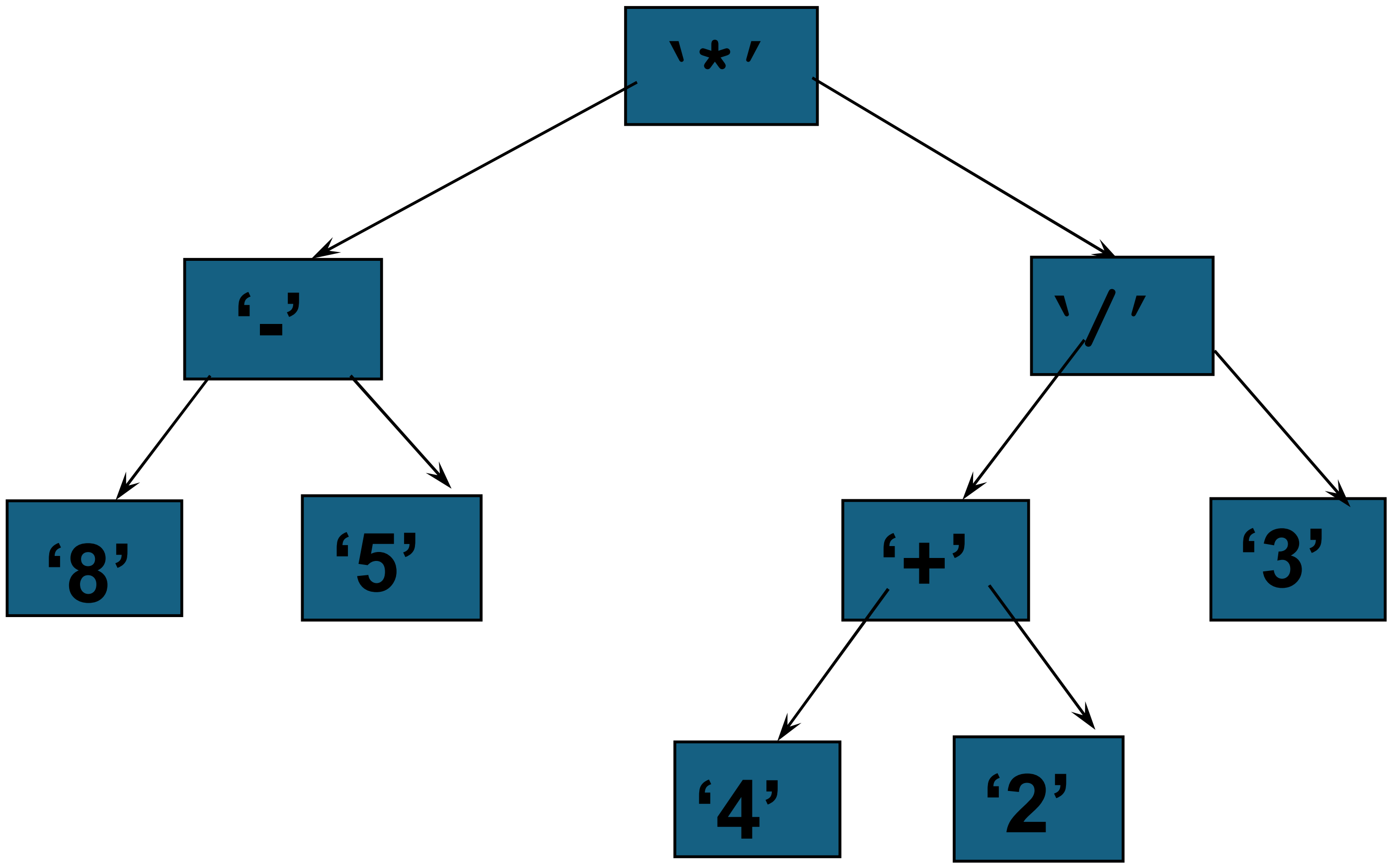
- Ordered rooted tree representing complicated expression such compound propositions, combinations of sets, and arithmetic expression are called Binary Expression Tree.
- In this special kind of tree,

  1. Each leaf node contains a single operand,

  2. Each nonleaf node contains a single binary operator, and

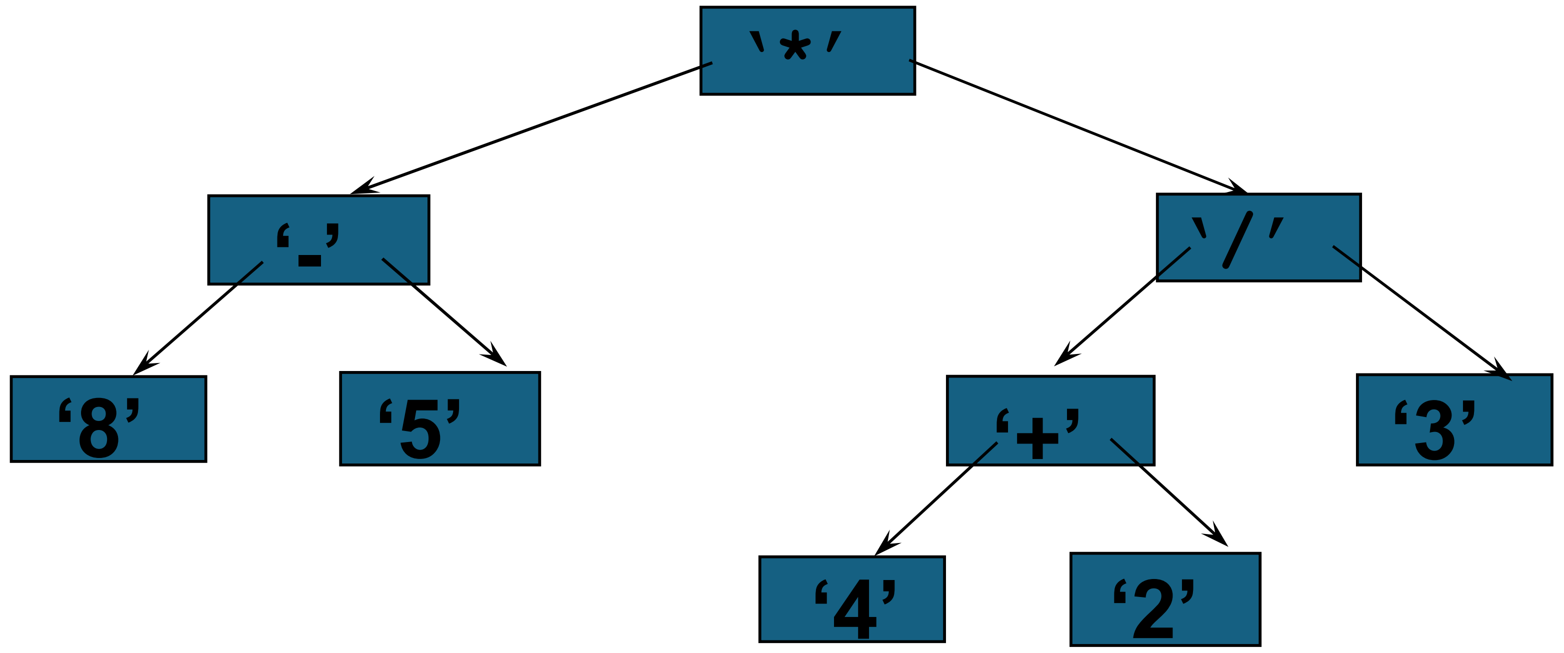
  3. The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

Infix, Prefix, and Postfix Expressions



What infix, prefix, postfix expressions does it represent?

Infix, Prefix, and Postfix Expressions

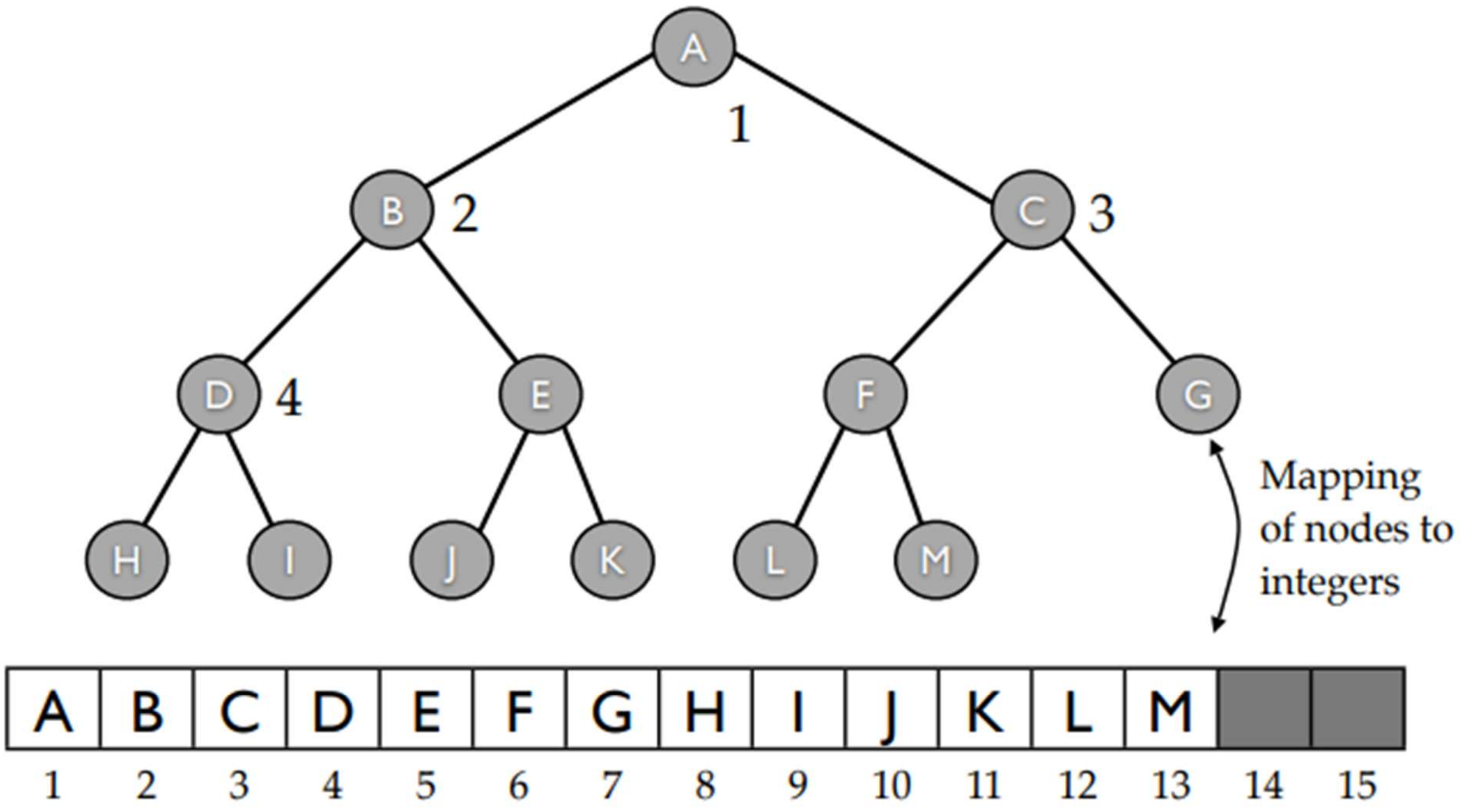


Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / *

Array Implementation for Complete Binary Trees

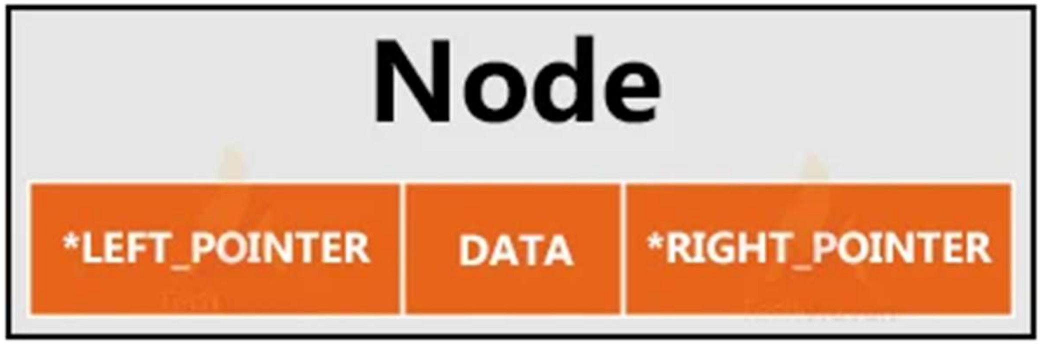


left(i): $2i$ **if** $2i \leq n$ **otherwise** 0

right(i): $(2i + 1)$ **if** $2i + 1 \leq n$ **otherwise** 0

parent(i): $\lfloor i/2 \rfloor$ **if** i ≥ 2 **otherwise** 0

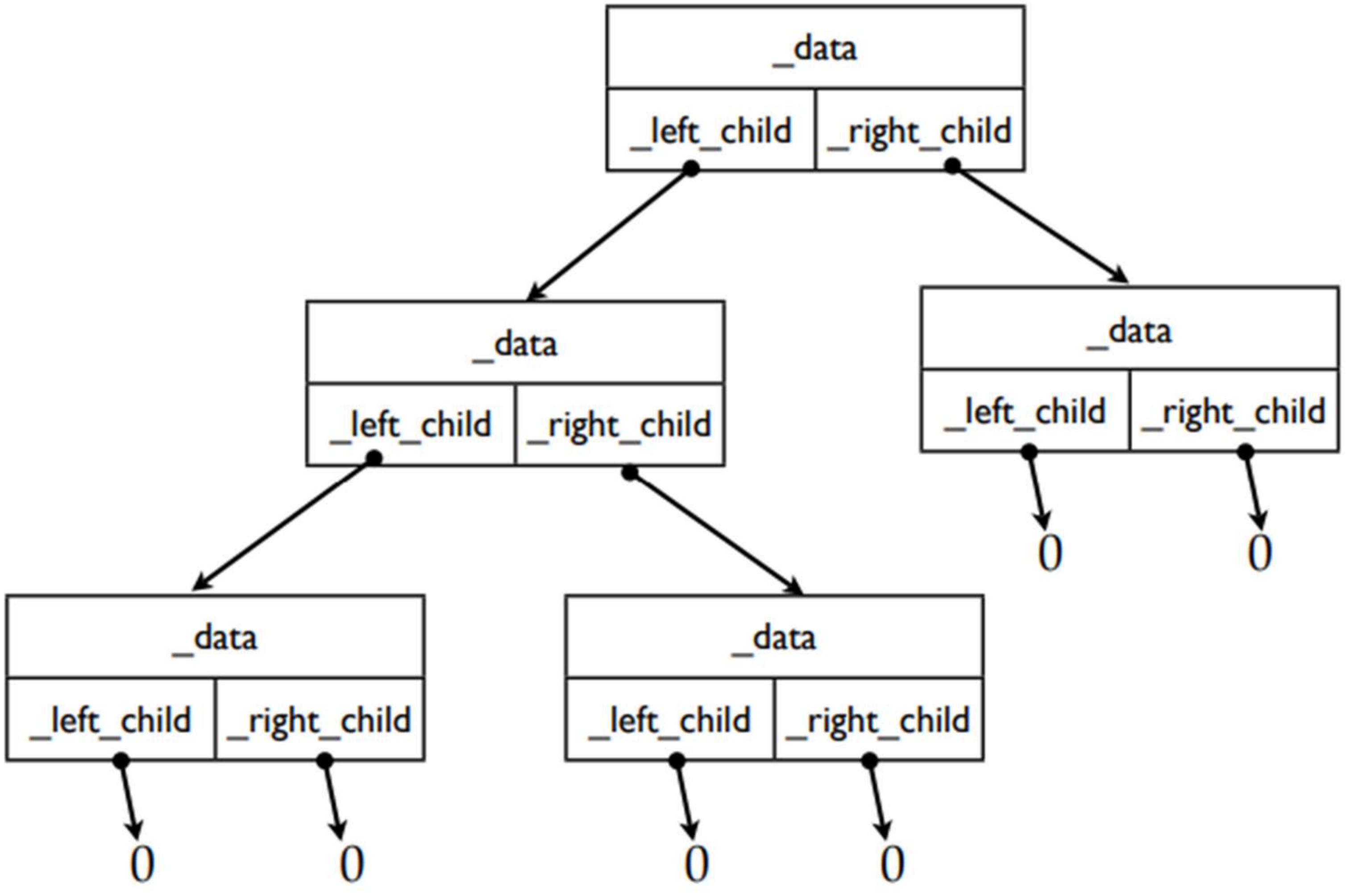
Linked Binary Tree Implementation



Example:- A tree node with integer data

```
struct node
{
    int data;
    struct node *left_child;
    struct node *right_child;
};
```

Linked Binary Tree Implementation



Binary Search Tree

Lets see search, insert, delete..

Radix Trees

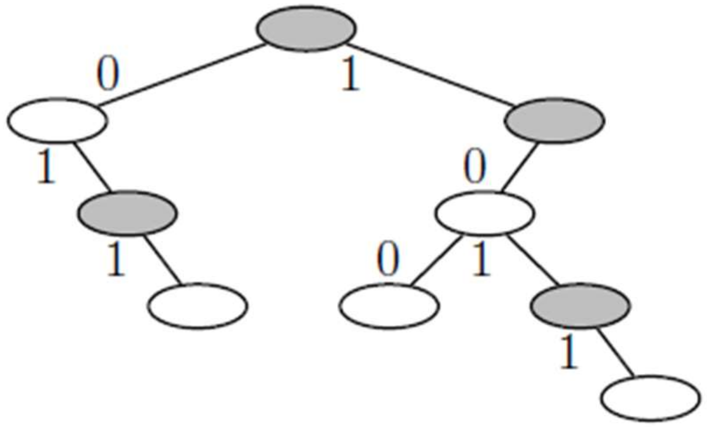


Figure 1: A radix tree storing the numbers `0, 011, 10, 100, 1011`.

Radix Trees

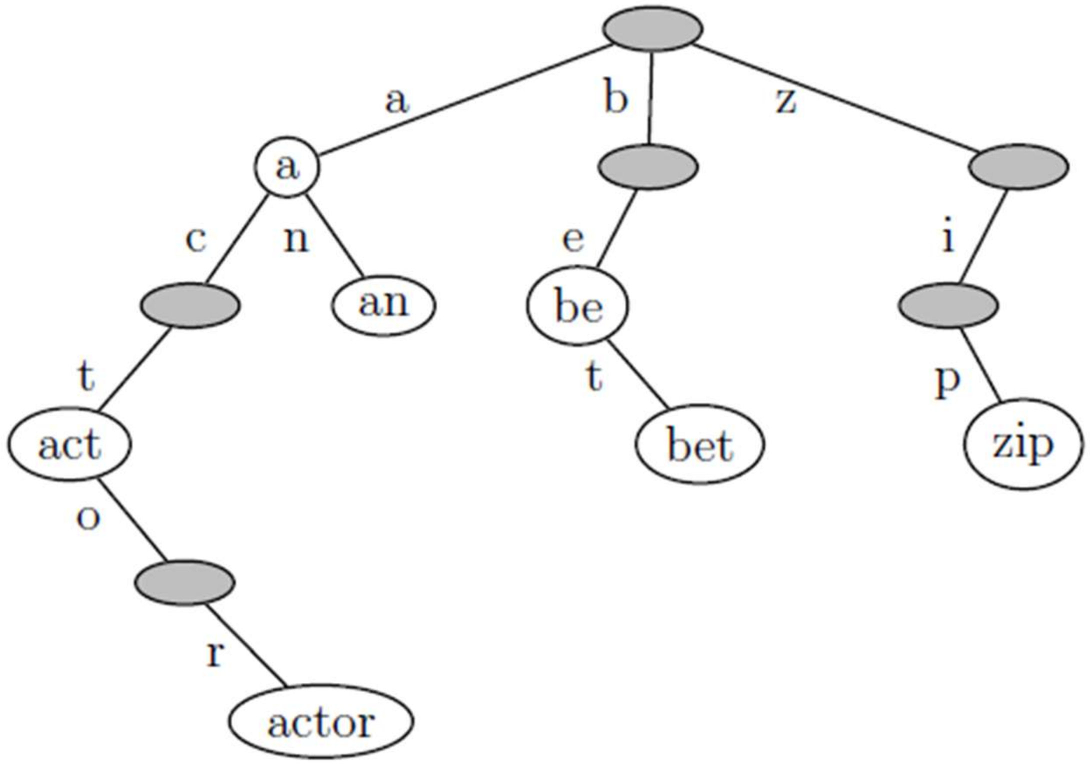


Figure 2: A radix tree storing the set of strings { `a`, `act`, `actor`, `an`, `be`, `bell`, `bet`, `zip`}. Note that the **branching factor** of the tree, i.e the max number of children of a node is equal to the size of the alphabet. In the figure, the null pointers are not shown.

Radix Trees - Search

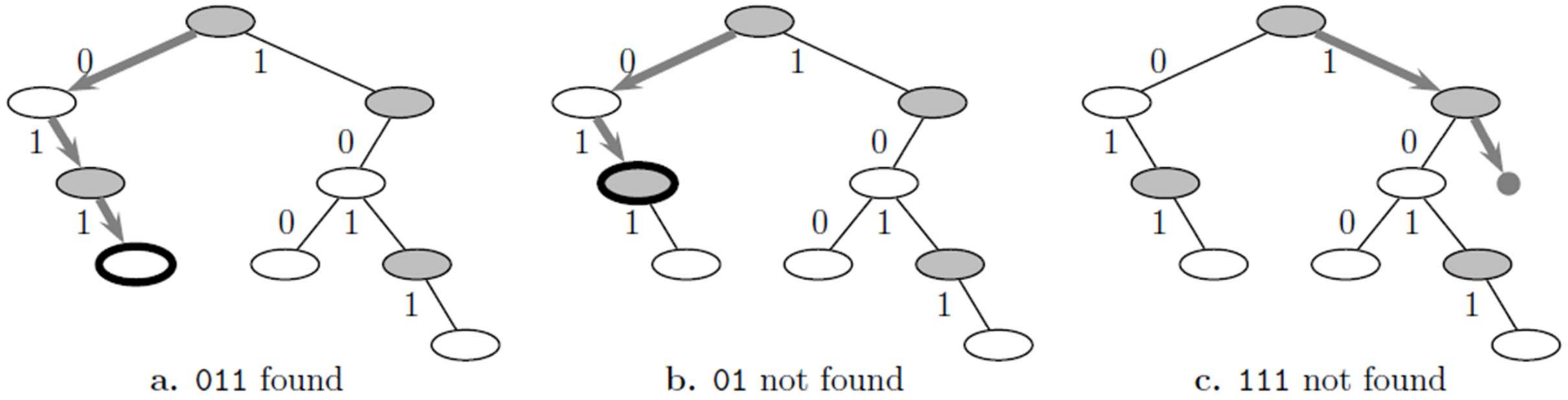
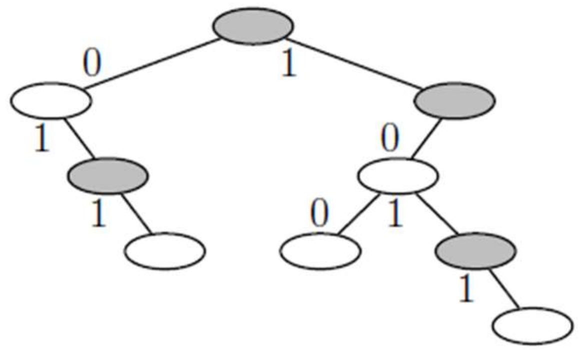


Figure 3: Searching in a radix tree.

Radix Trees - Insert



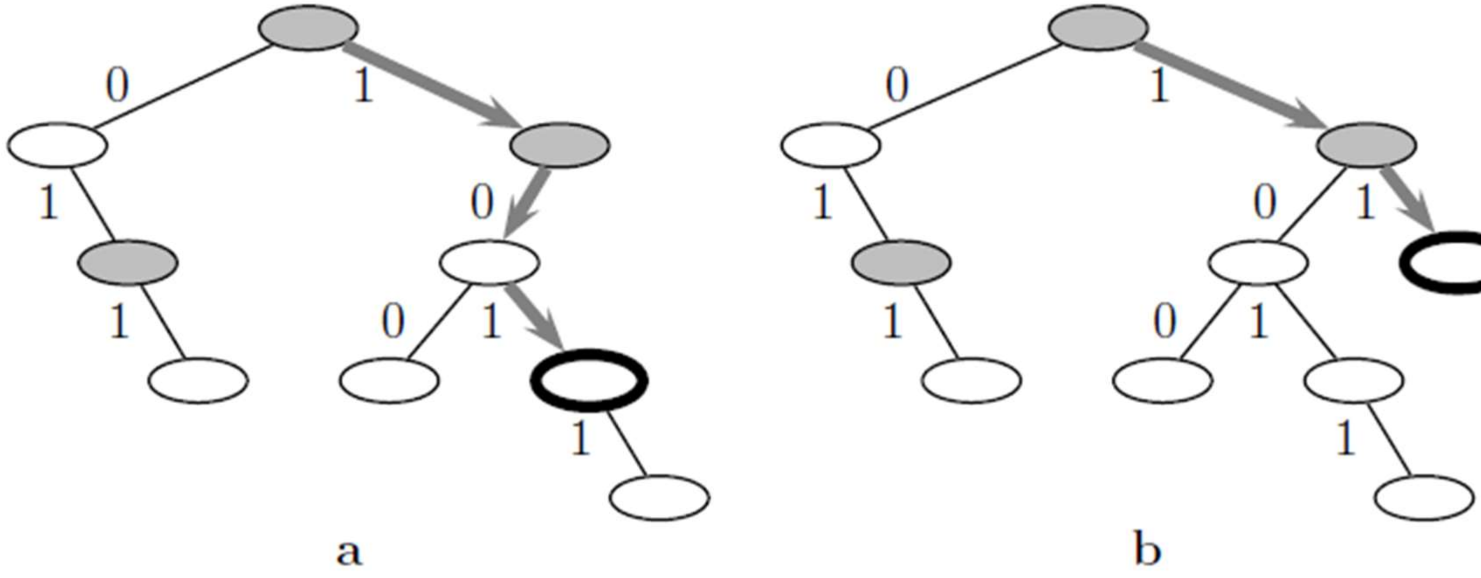
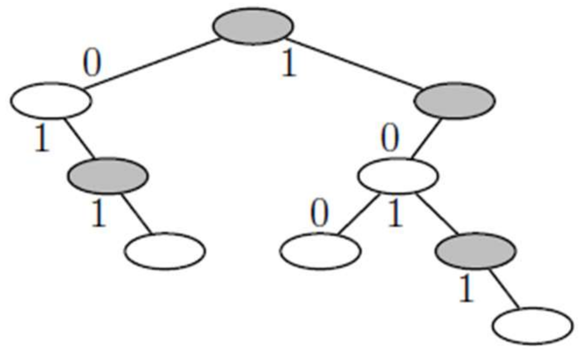


Figure 4: Inserting 101 (a), then 11 (b) in the radix tree from figure 1. In (a), the node already exists and it is colored in white; in (b), a new leaf has to be created.

Radix Trees - Delete



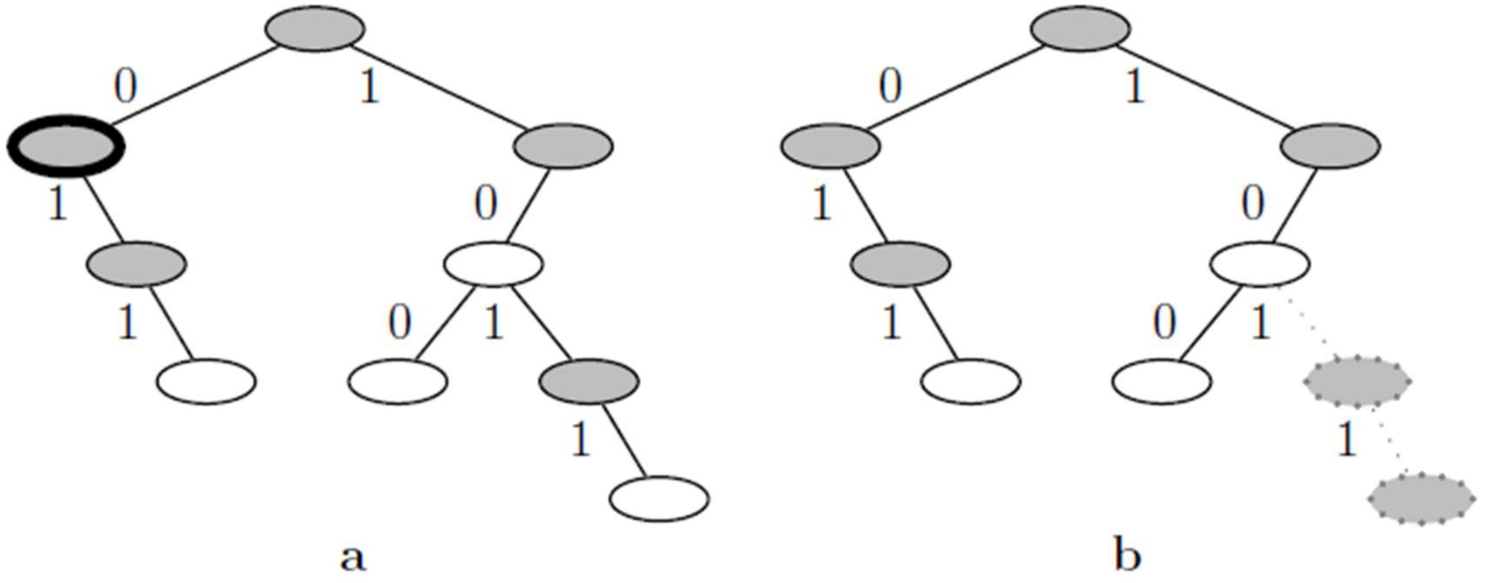


Figure 5: Deleting 0 (a) and 1011 (b) from the radix tree in figure 1. In (a), the deleted string is at an interior node; the node is colored gray. In (b) the deleted string is at a leaf; the leaf is deleted, then recursively all the other resulting gray leaves.