TRIE

Prepared By

Lec Swapnil Biswas

TRIE

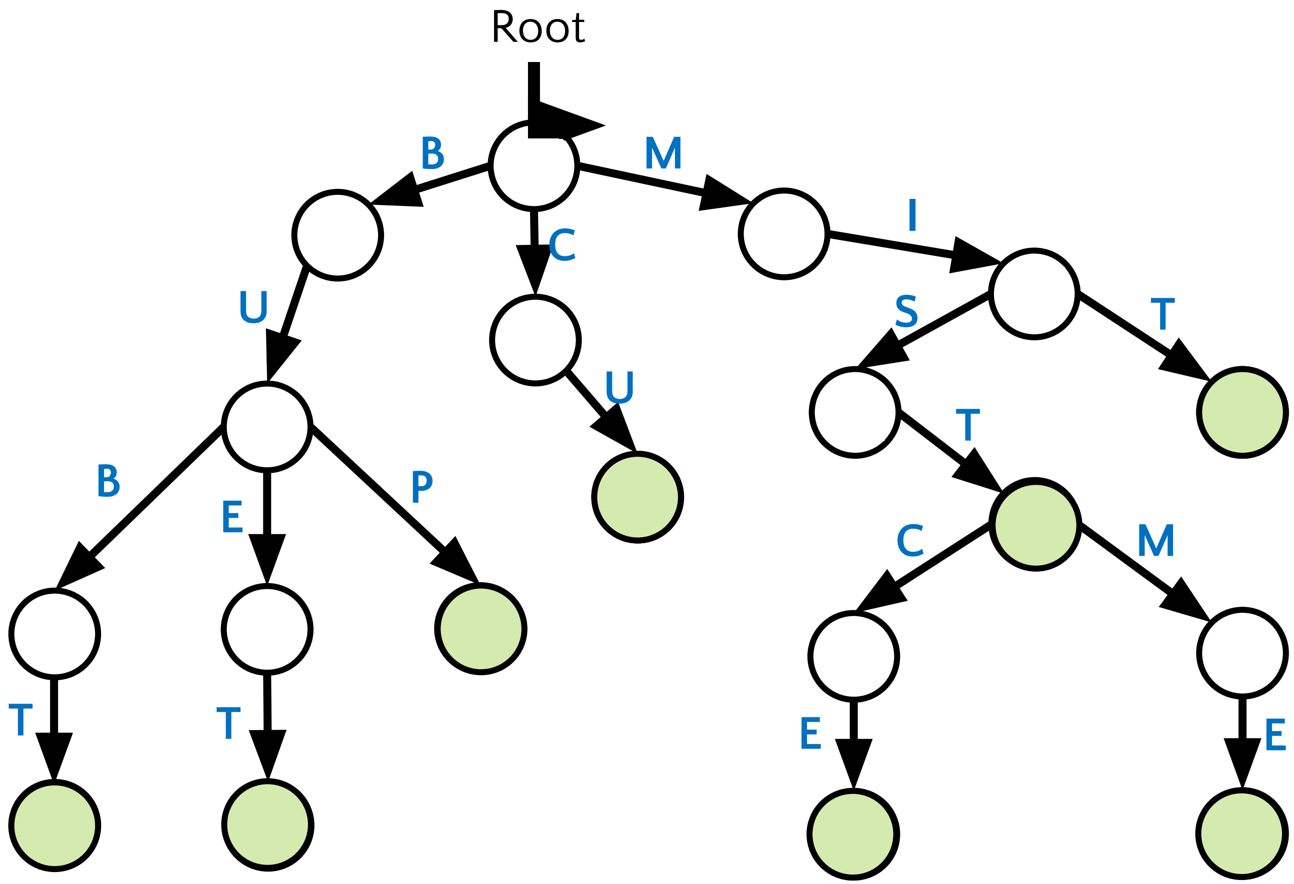
- A tree based data structure (k-ary tree)
- Root is an empty node.
- (k=26) Each node will have 26 children (Each child represents a alphabetic letter)
- Implemented by linked data structure
- It allows for very fast searching and insertion operations
- The word TRIE comes from the word Re**trie**val
- It refers to the quick retrieval of strings
- Used for storing strings, string matching, lexicographical sorting etc.

WHY TRIE?

- Consider a database of strings
- Number of strings in the database is ***n***
- Now what is the complexity to find a given string ***x*** whether ***x*** exists in the database or not
- Ans: O(***n*** x ***m***) where ***m*** is the average length of the strings
- Now if the database is too big, then finding a string from the database will be time consuming
- Goal is to find a string ***x*** without the dependency of ***n***
- TRIE will solve this issue to find a string ***x*** in O(length(***x***)) complexity
- So doesn't matter how long the database is, time complexity of finding a string x will remain length(x)

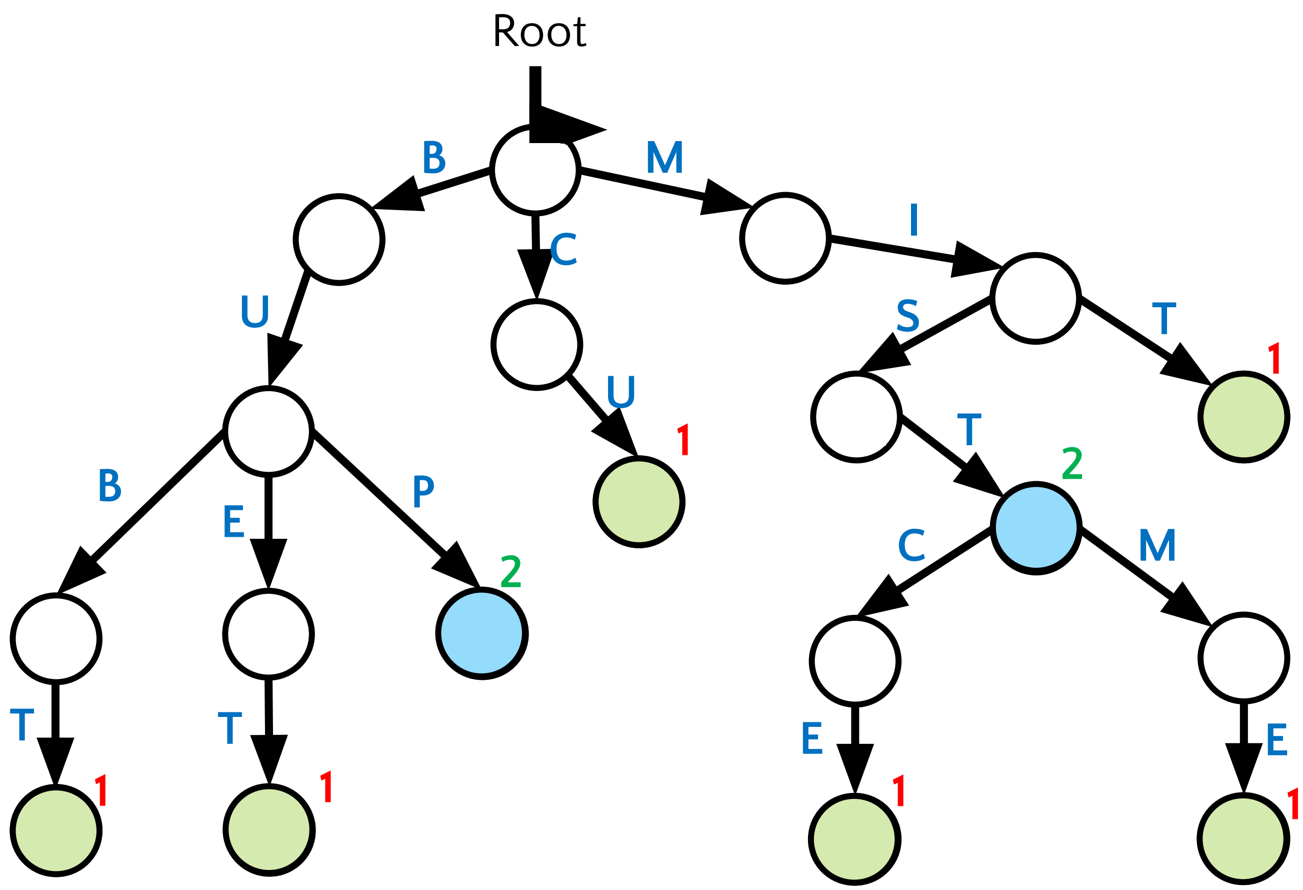
INSERT IN TRIE

- ❑ insert("MIT")
- ❑ insert("MIST")
- ❑ insert("BUET")
- ❑ insert("MISTCE")
- ❑ insert("BUBT")
- ❑ insert("MISTME")
- ❑ insert("BUP")
- ❑ insert("CU")
- ❑ insert("MIST")
- ❑ Is it possible to know the frequency of any string in the TRIE?
  - NO
- ❑ But keeping a counter variable at each node can address this issue



INSERT IN TRIE (WITH COUNTER)

- ❑ insert(“MIT”)
- ❑ insert(“MIST”)
- ❑ insert(“BUET”)
- ❑ insert(“MISTCE”)
- ❑ insert(“BUBT”)
- ❑ insert(“MISTME”)
- ❑ insert(“BUP”)
- ❑ insert(“CU”)
- ❑ insert(“MIST”)
- ❑ insert(“BUP”)

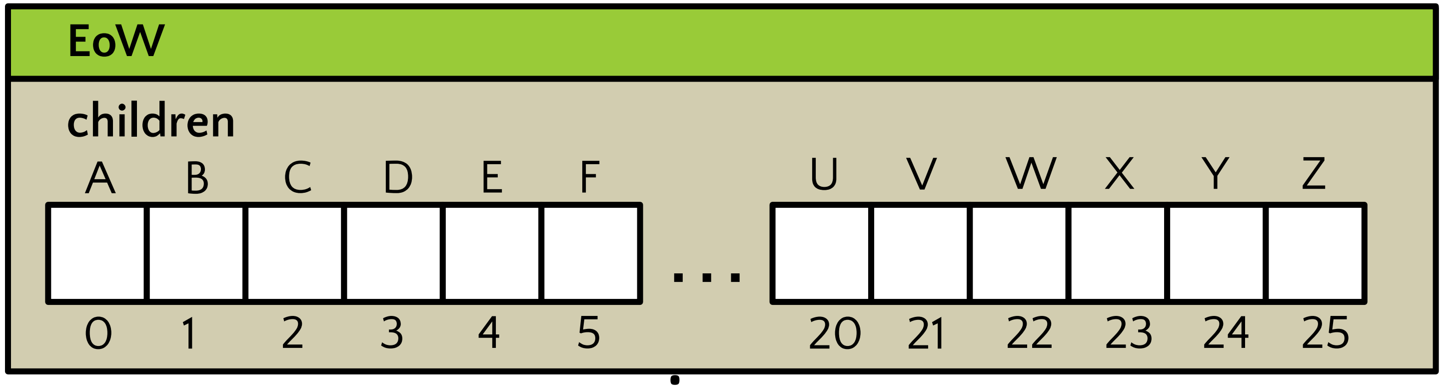


INSERT IN TRIE (WITH COUNTER)

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- insert("BUP")

NODE REPRESENTATION

```
struct Node{
    int EoW;
    Node  *children[26];
}
```



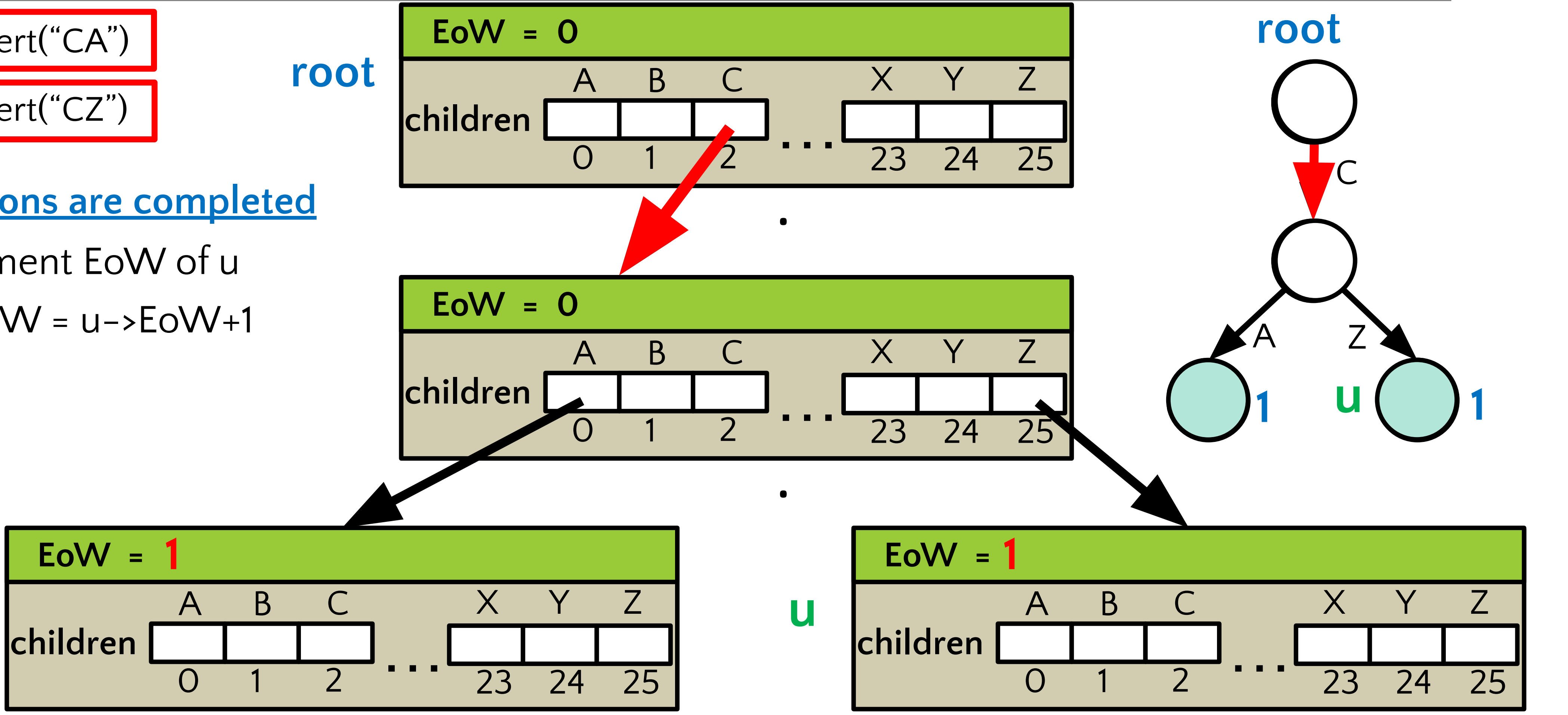
NODE REPRESENTATION

- ❑ insert("CA")
- ❑ insert("CZ")

<u>Iterations are completed</u>

Increment EoW of u

u->EoW = u->EoW+1



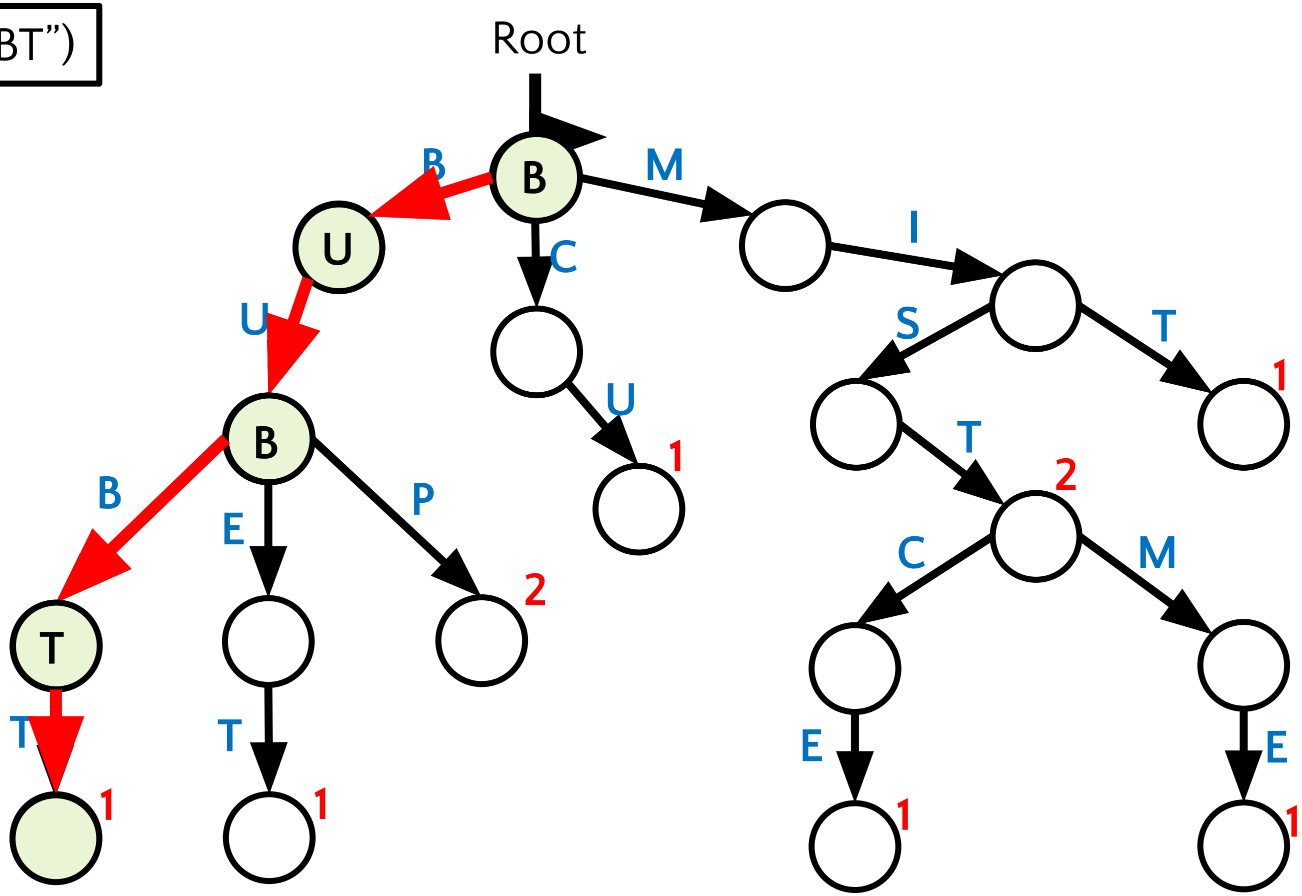
INSERT IN TRIE

```
insert(x)
    Node pointer u ← root                  Initially pointing u at the root
    for k ← 0 to size(x) - 1               Iterates for size(x) number of times
        r ← x[k] - 65                      r is the relative position of current char
O(|x|)
        if u->children[r] is NULL          No children condition
            u->children[r] ← new Node( )   Creates new node under children[r]
        u ← u->children[r]                 Pushes u down for next iteration
    u->EoW ← u->EoW + 1;                   Increments u->EoW after completing iteration
```

SEARCH IN TRIE

- ❑ insert("MIT")
- ❑ insert("MIST")
- ❑ insert("BUET")
- ❑ insert("MISTCE")
- ❑ insert("BUBT")
- ❑ insert("MISTME")
- ❑ insert("BUP")
- ❑ insert("CU")
- ❑ insert("MIST")
- ❑ insert("BUP")

❑ search("BUBT")



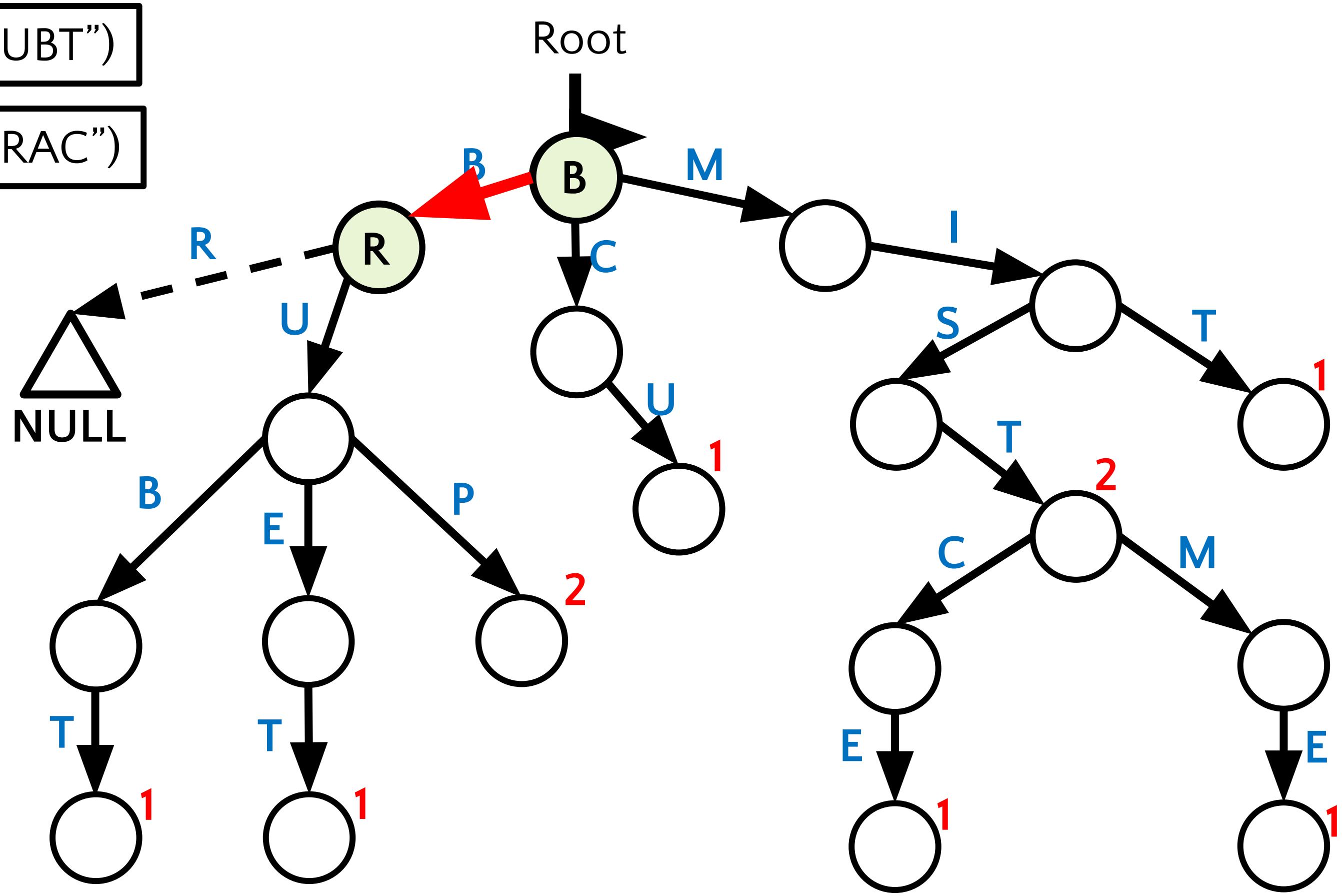
We reach a vertex with counter >0

Means "BUBT" exists

SEARCH IN TRIE

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- insert("BUP")

- search("BUBT")
- search("BRAC")



We reach to NULL

Means "BRAC" doesn't exist

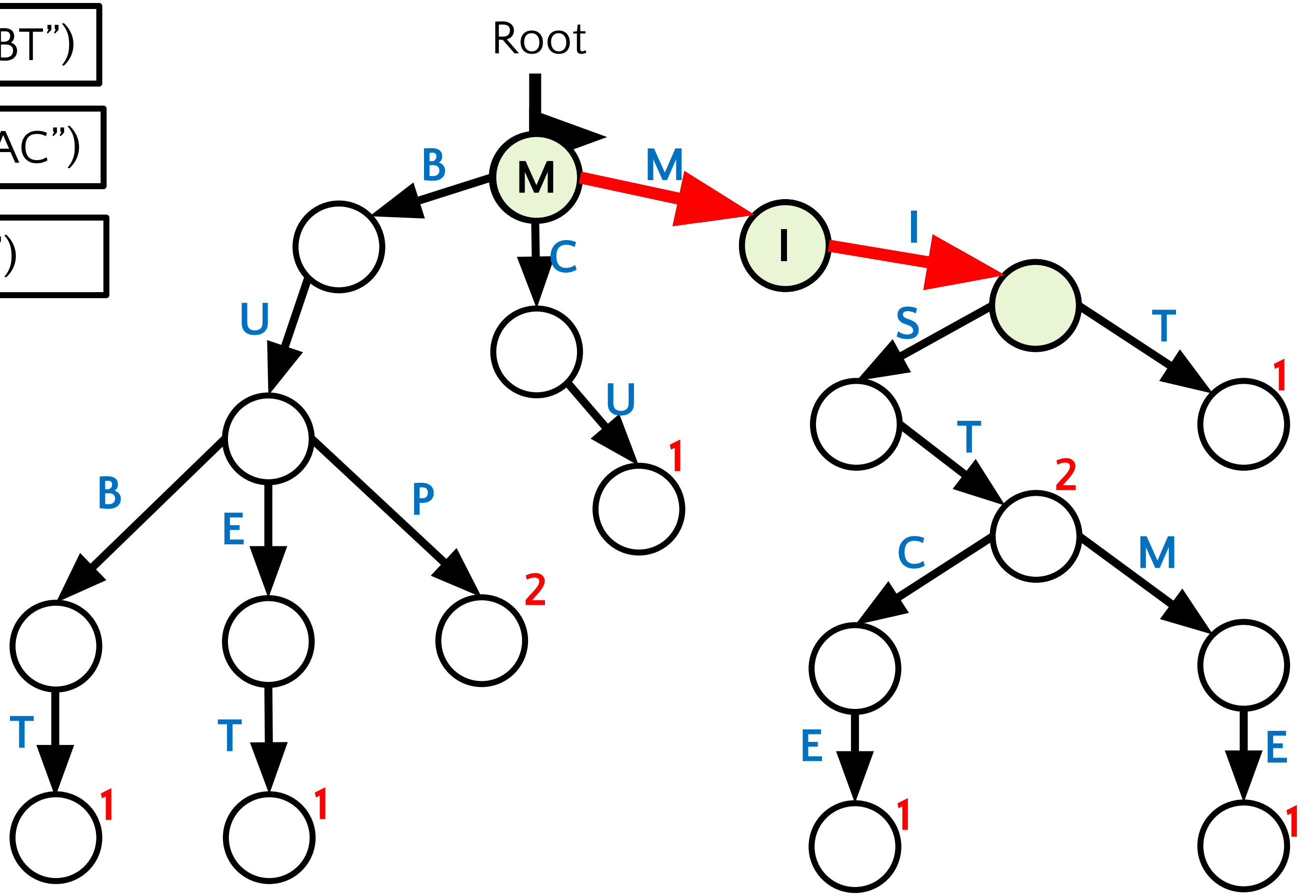
SEARCH IN TRIE

- ❑ insert("MIT")
- ❑ insert("MIST")
- ❑ insert("BUET")
- ❑ insert("MISTCE")
- ❑ insert("BUBT")
- ❑ insert("MISTME")
- ❑ insert("BUP")
- ❑ insert("CU")
- ❑ insert("MIST")
- ❑ insert("BUP")

❑ search("BUBT")

❑ search("BRAC")

❑ search("MI")



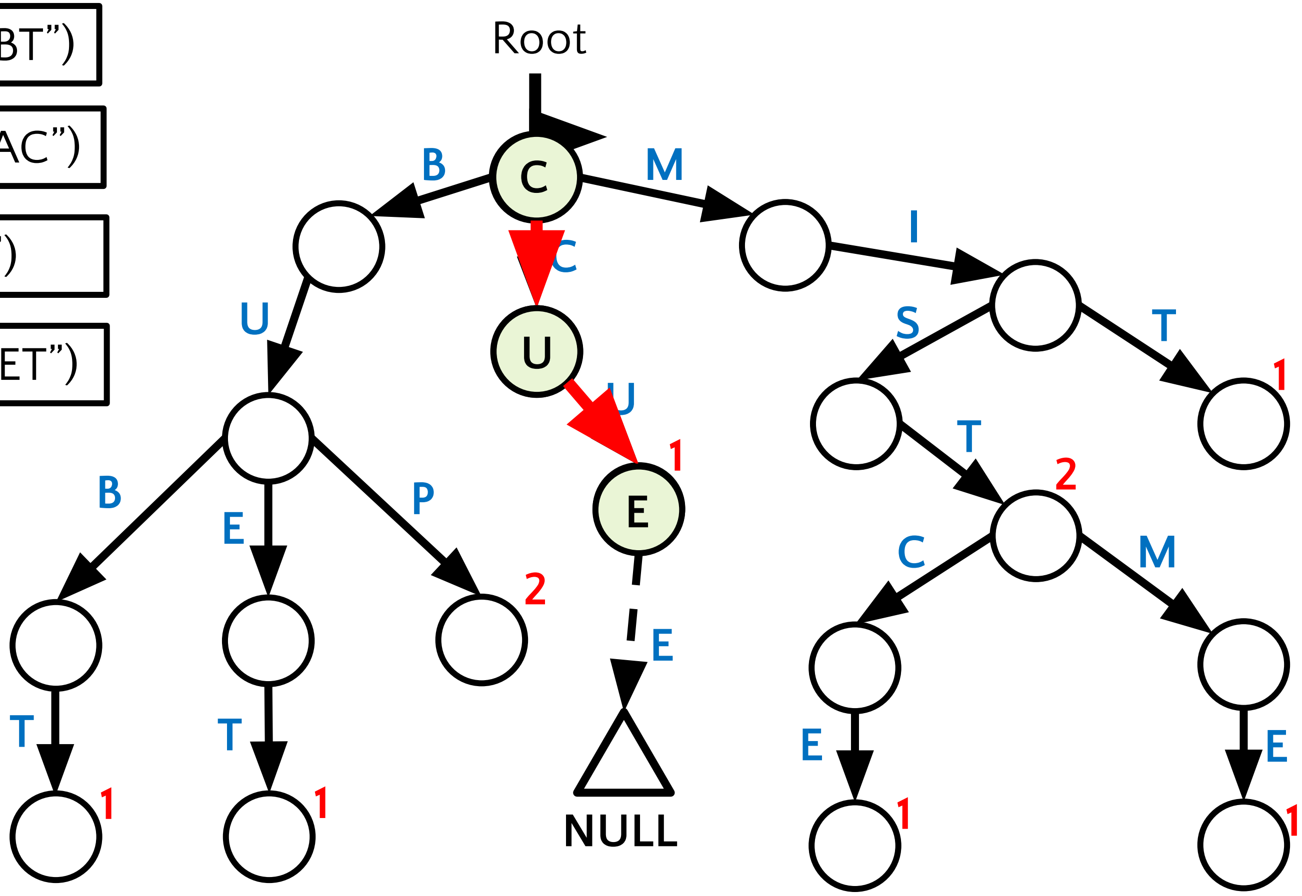
We can't reach a node with counter=0

Means "MI" doesn't exist

SEARCH IN TRIE

- insert("MIT")
- insert("MIST")
- insert("BUET")
- insert("MISTCE")
- insert("BUBT")
- insert("MISTME")
- insert("BUP")
- insert("CU")
- insert("MIST")
- insert("BUP")

- search("BUBT")
- search("BRAC")
- search("MI")
- search("CUET")

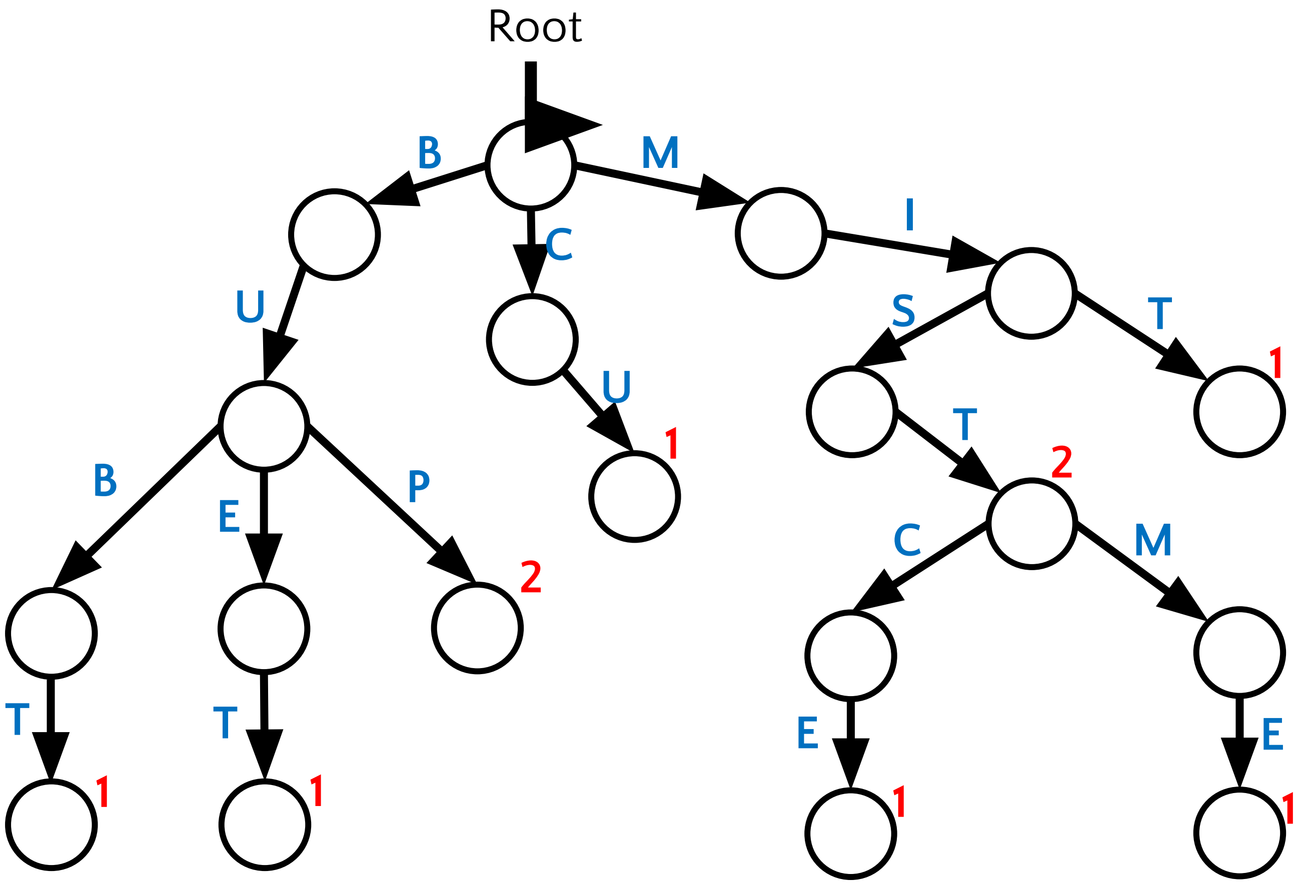


We reach to NULL

Means "CUET" doesn't exist

SEARCH IN TRIE

- We don't find a string in TRIE if
  - The search ends to a NULL
  - The search ends to a node with counter = 0 (Not the end of a word)



METHODS

- void insert(string x)
- int search(string x)
- bool delete(string x)
- void lexSort( )

RELATIVE POSITION OF A CHARACTER

- Consider the strings can only contain uppercase letters
- The relative position of a character is obtained by subtracting 65 from it

Character	Relative Position
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8

Character	Relative Position
I	9
J	10
K	11
L	12
M	13
N	14
O	15
P	16
Q	17

Character	Relative Position
R	18
S	19
T	20
U	21
V	22
W	23
X	24
Y	25

RELATIVE POSITION OF A CHARACTER

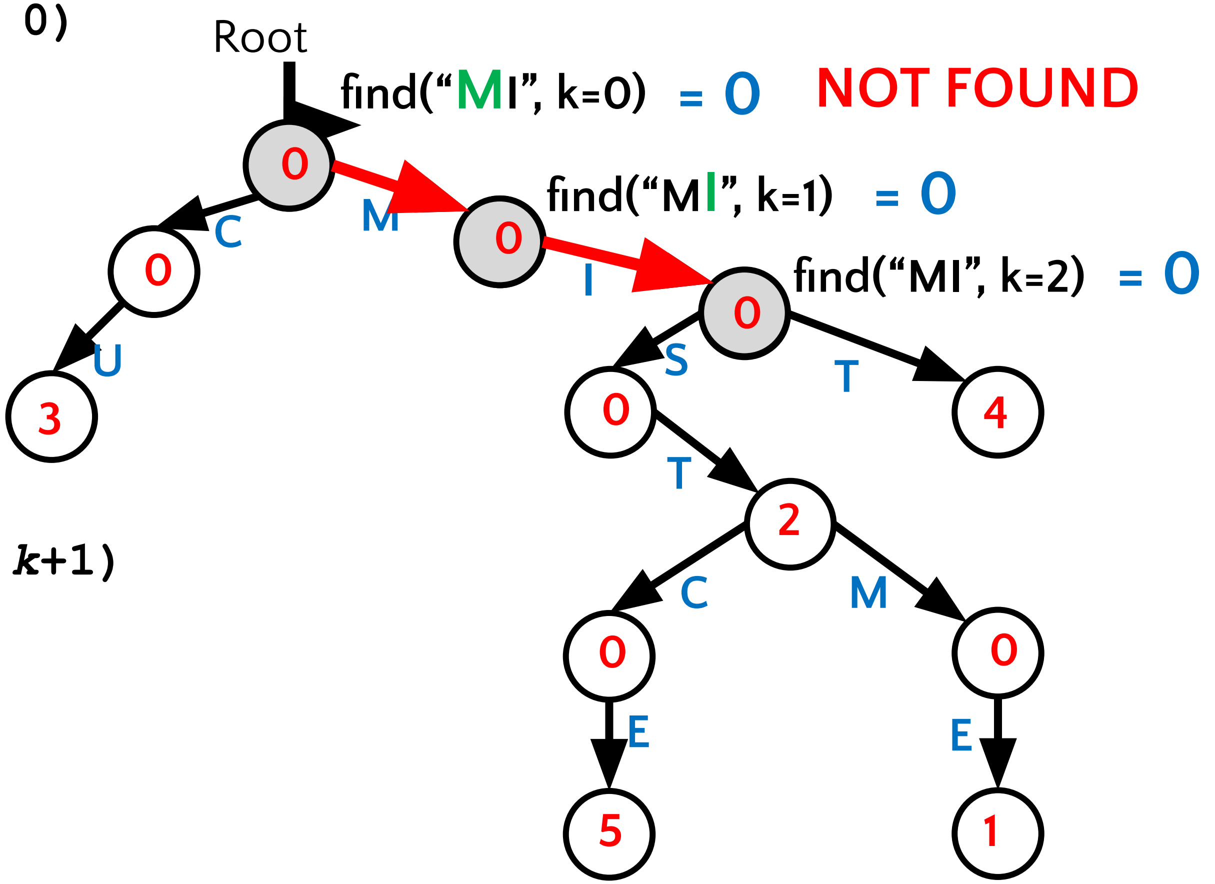
```
int relPos(char c){
    int ascii = (int) c;
    return ascii – 65;
}
```

SEARCH IN TRIE

```
find(x, Node pointer cur ← root, k ← 0)
    if cur is NULL
        return 0
    if k equals size(x)
        return cur->EoW

    r ← x[k] - 65
    return find(x, cur->children[r], k+1)
```

- find("MI")

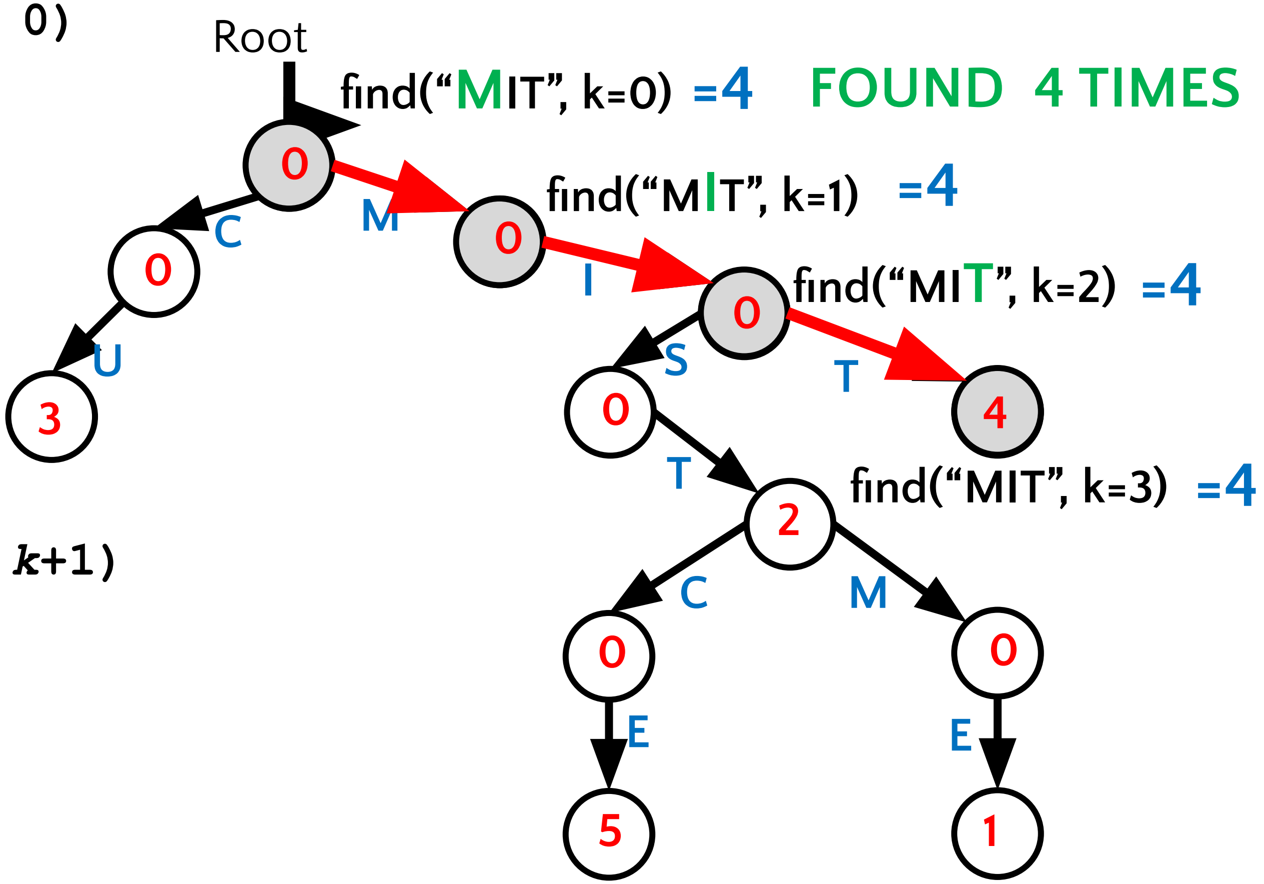


SEARCH IN TRIE

```
find(x, Node pointer cur ← root, k ← 0)
    if cur is NULL
        return 0
    if k equals size(x)
        return cur->EoW

    r ← x[k] - 65
    return find(x, cur->children[r], k+1)
```

- ❑ find("MI")
- ❑ find("MIT")

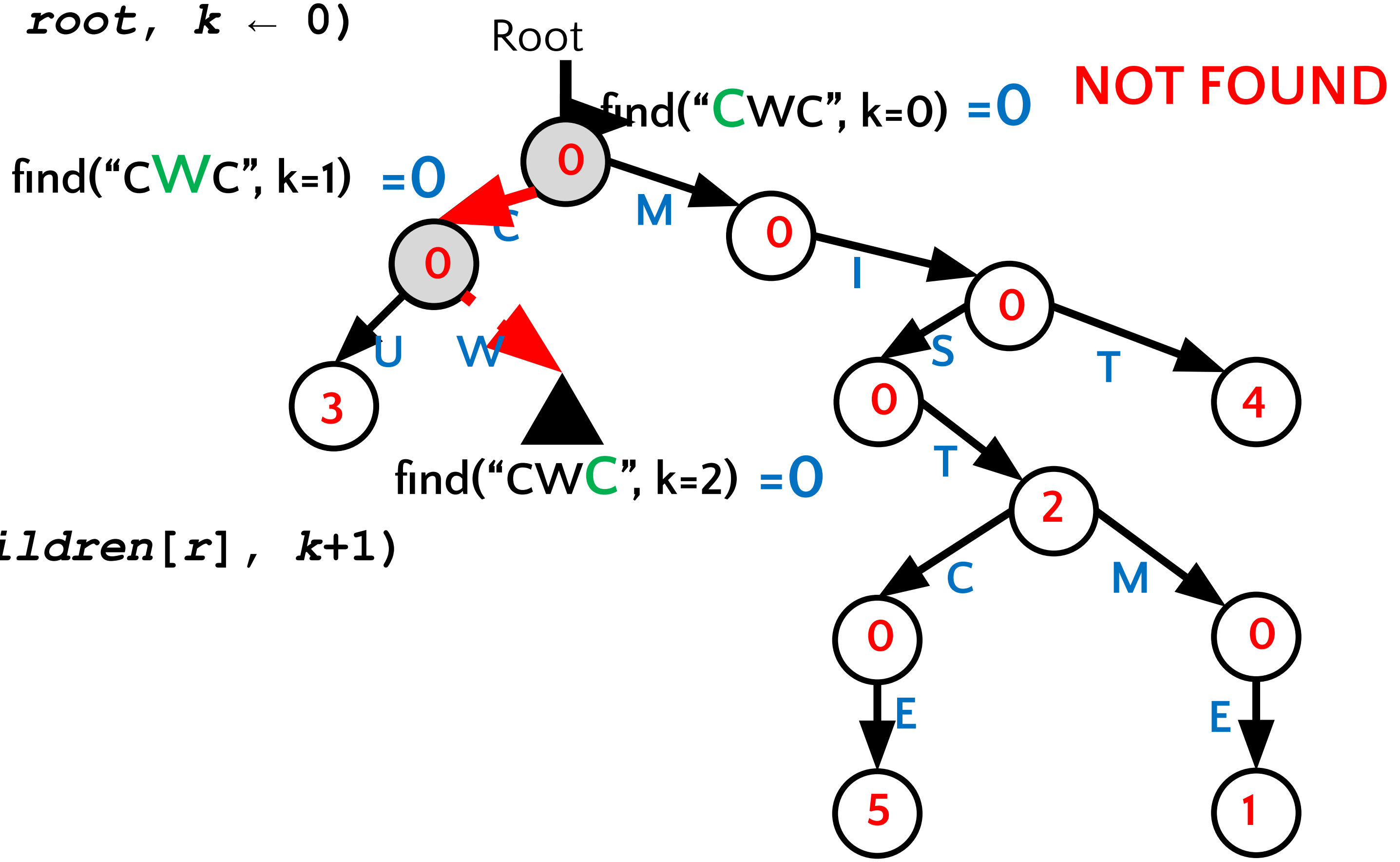


SEARCH IN TRIE

```
find(x, Node pointer cur ← root, k ← 0)
    if cur is NULL
        return 0
    if k equals size(x)
        return cur->EoW

    r ← x[k] - 65
    return find(x, cur->children[r], k+1)
```

- ❑ find(“MI”)
- ❑ find(“MIT”)
- ❑ find(“CWC”)



SEARCH IN TRIE (COMPLEXITY)

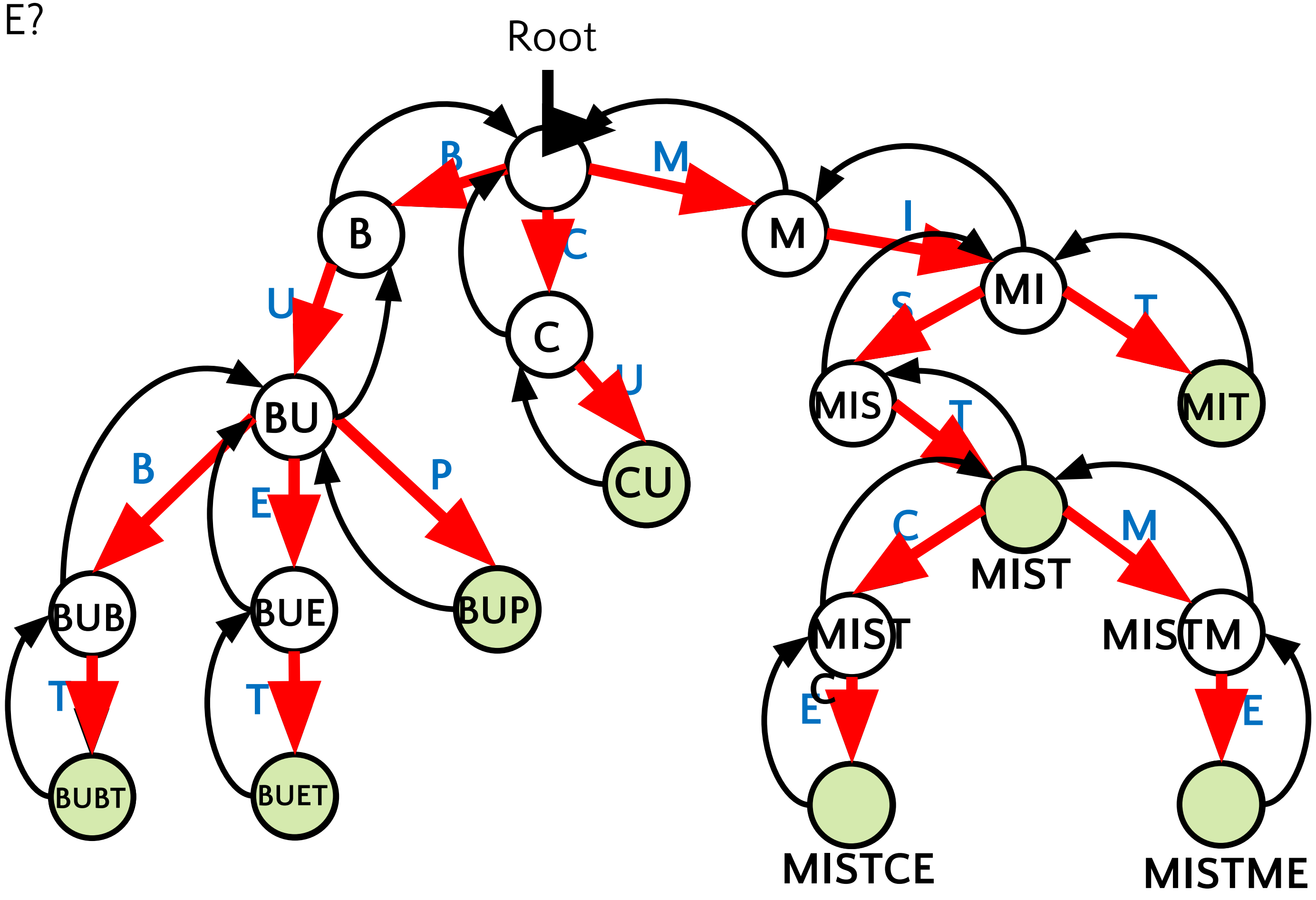
- Number of recursive call can not exceed the length of longest string in the TRIE
  - Let the longest string in the TRIE is ***s***
  - So the time complexity of searching is O(|***s***|)

LEXICOGRAPHICAL ORDER

- What are the strings stored in the TRIE?

BUBT
BUET
BUP
CU
MIST
MISTCE
MISTME
MIT

- Strings are sorted lexicographically
- Left to Right approach
  (Merging with parent)



LEXICOGRAPHICAL ORDER

```
void printTRIE(Node *cur = root, string s="")
{
	if(cur->EoW>0)
	{
		cout<<s<<endl;
	}
	for(int i=0; i<26; i++)
	{
		if(cur->children[i]!=NULL)
		{
			char c = char(i + 65);
			printTRIE(cur->children[i], s+c);
		}
	}
}
```

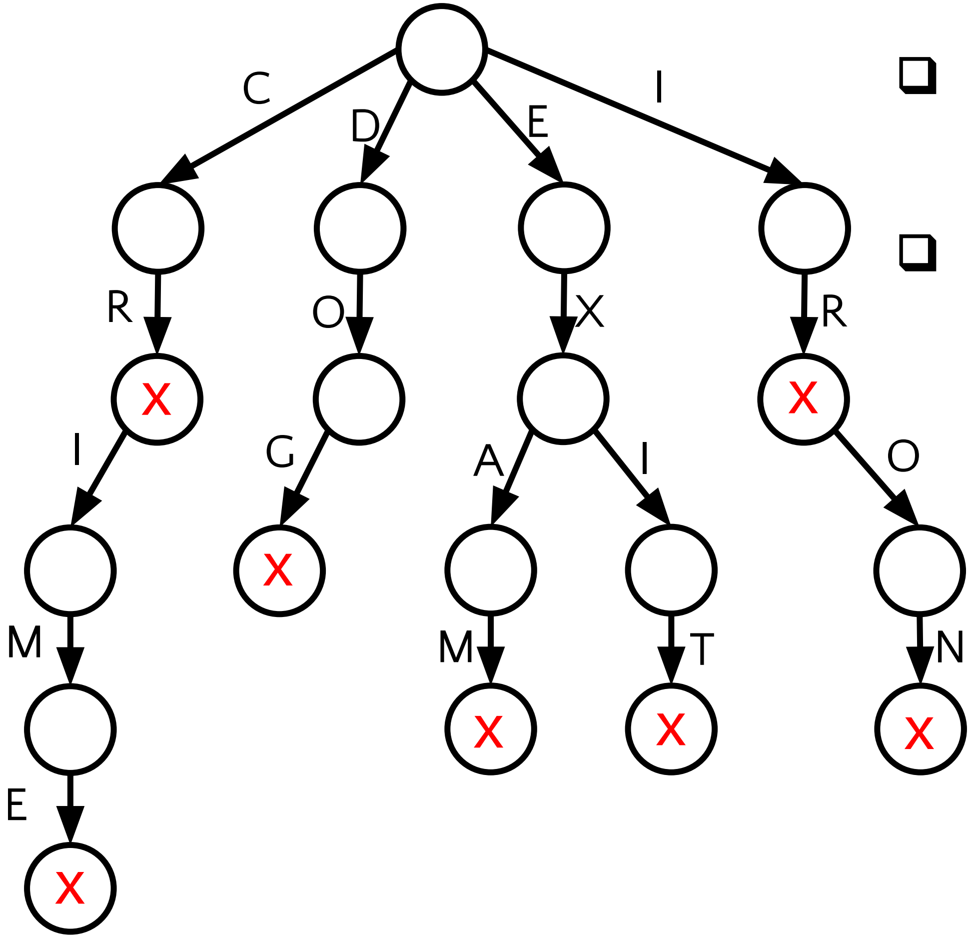
Base case:
If the pointer reaches to the end of a word
Then the word is printed

Traversing all the edges of a node from left to right

Calling the function recursively for those nodes
Having at least one child(edge).

So for leaf node: No recursive call is made

DELETE FROM TRIE



- List down the words

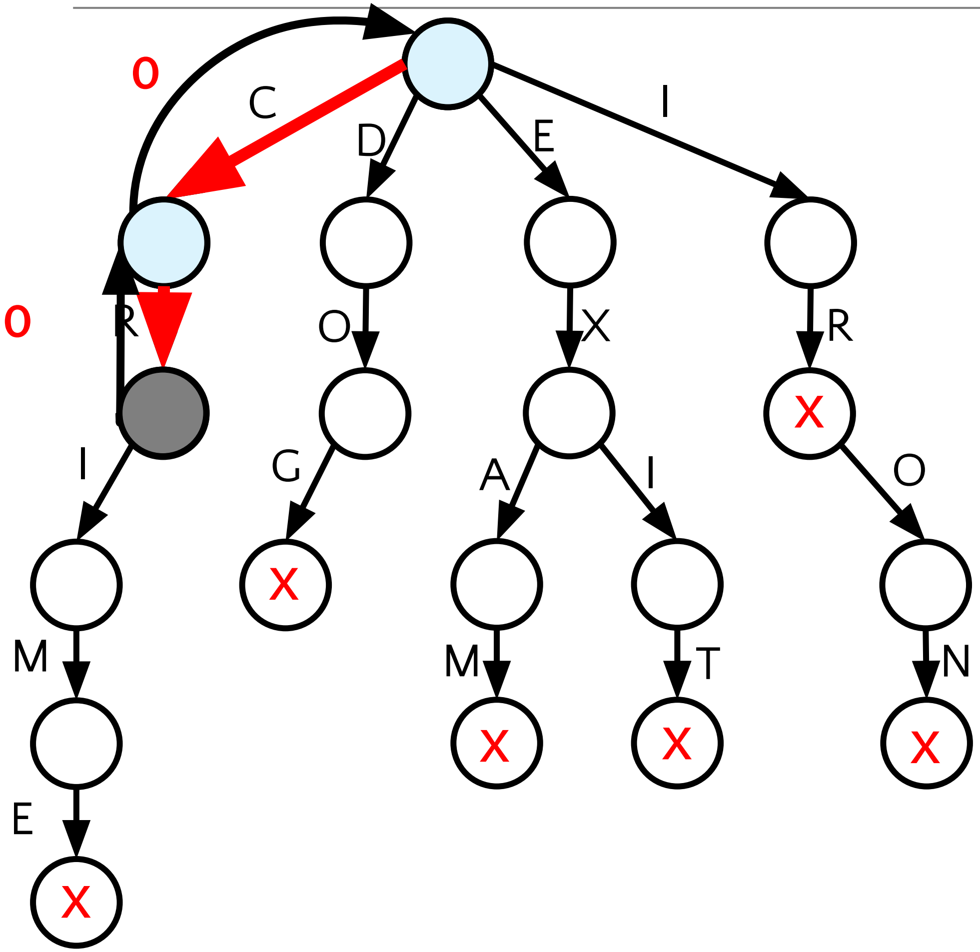
  CR CRIME DOG EXAM EXIT IR IRON

- 2 Cases for deletion

  - The word is a prefix of other words
  - Ex : CR IR
  - The word is not a prefix of any other words
  - Ex : CRIME DOG EXAM EXIT IRON

But it is to be checked that whether the word exists in the TRIE or not before deletion

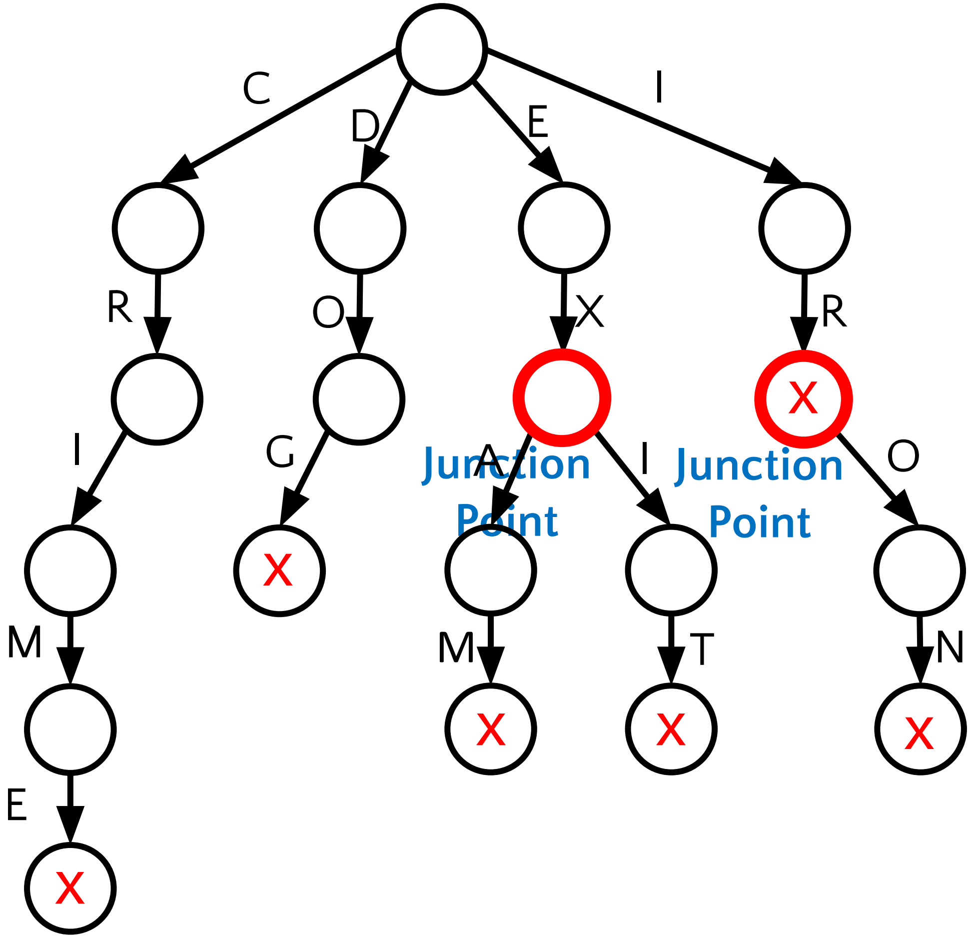
DELETE FROM TRIE (CASE-1)



- **The word is a prefix of other words**
  - Simply remove the EoW mark from the final node of the string in TRIE
  - delete("CR")
- **How did we understand that "CR" is a prefix of other words?**
  - Because the final node of CR in TRIE is not a leaf
- **How to check that whether a node is a leaf or not?**
  - Leaf: If a node having no child or all the child point to NULL

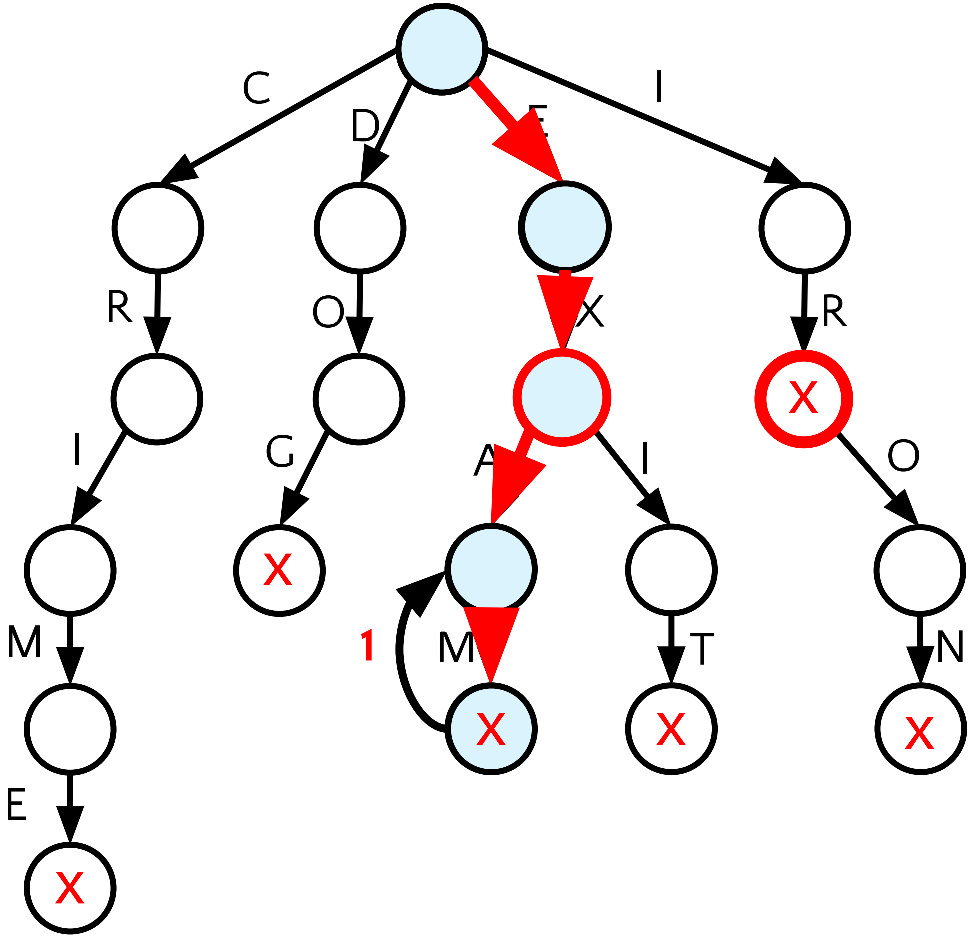
```
bool isLeaf(Node *u){
    for(int i=0; i<26; i++)    if(u->children[i]!=NULL)    return false;
    return true;
}
```

DELETE FROM TRIE (CASE-2)



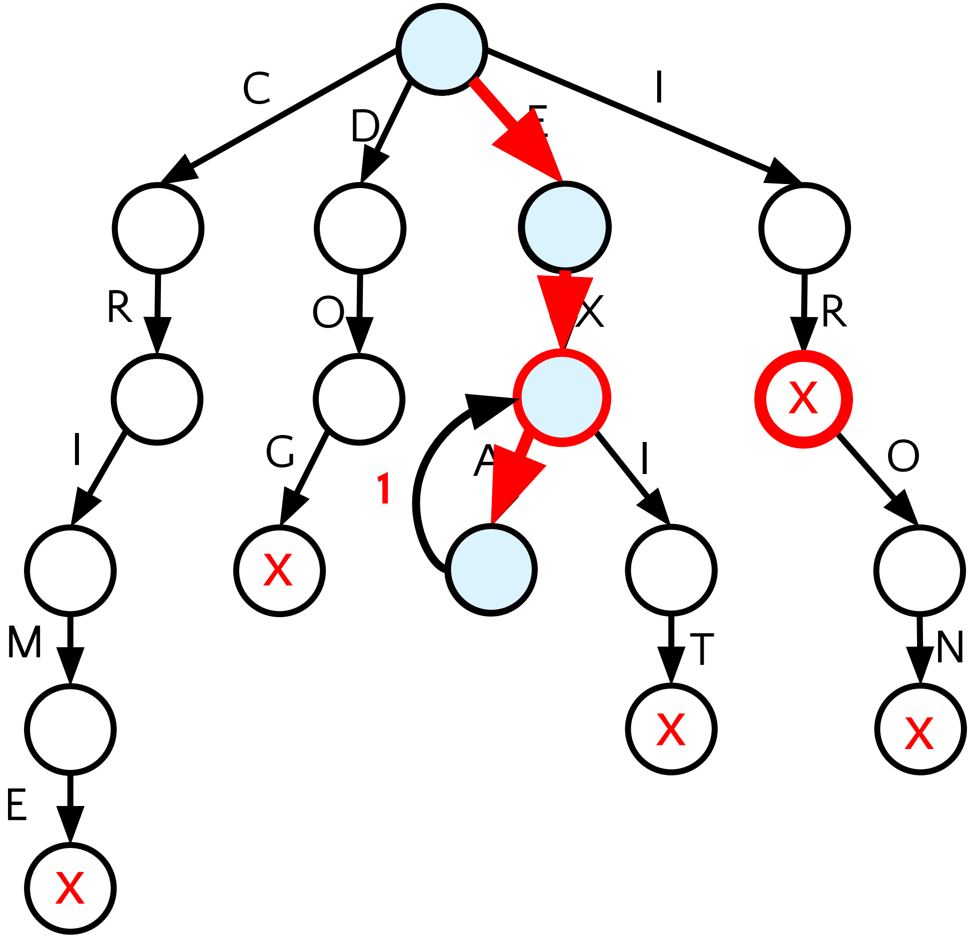
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")
  - delete("IRON")

DELETE FROM TRIE (CASE-2)



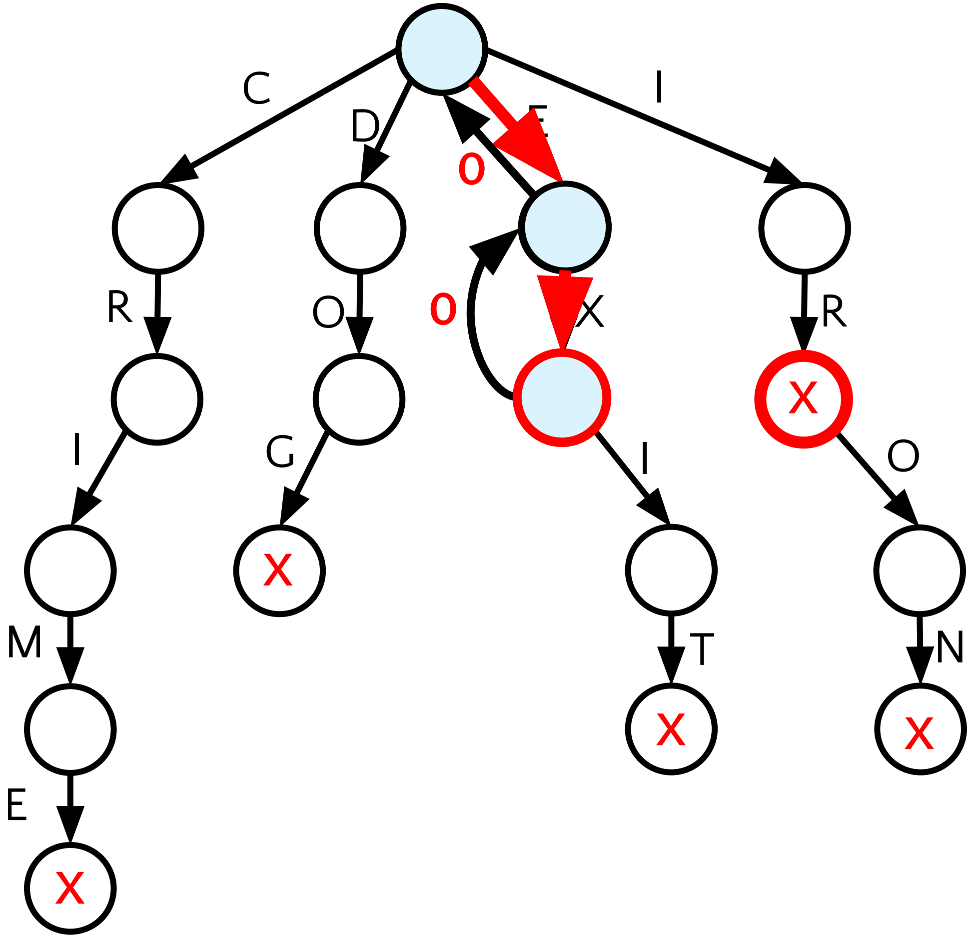
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")

DELETE FROM TRIE (CASE-2)



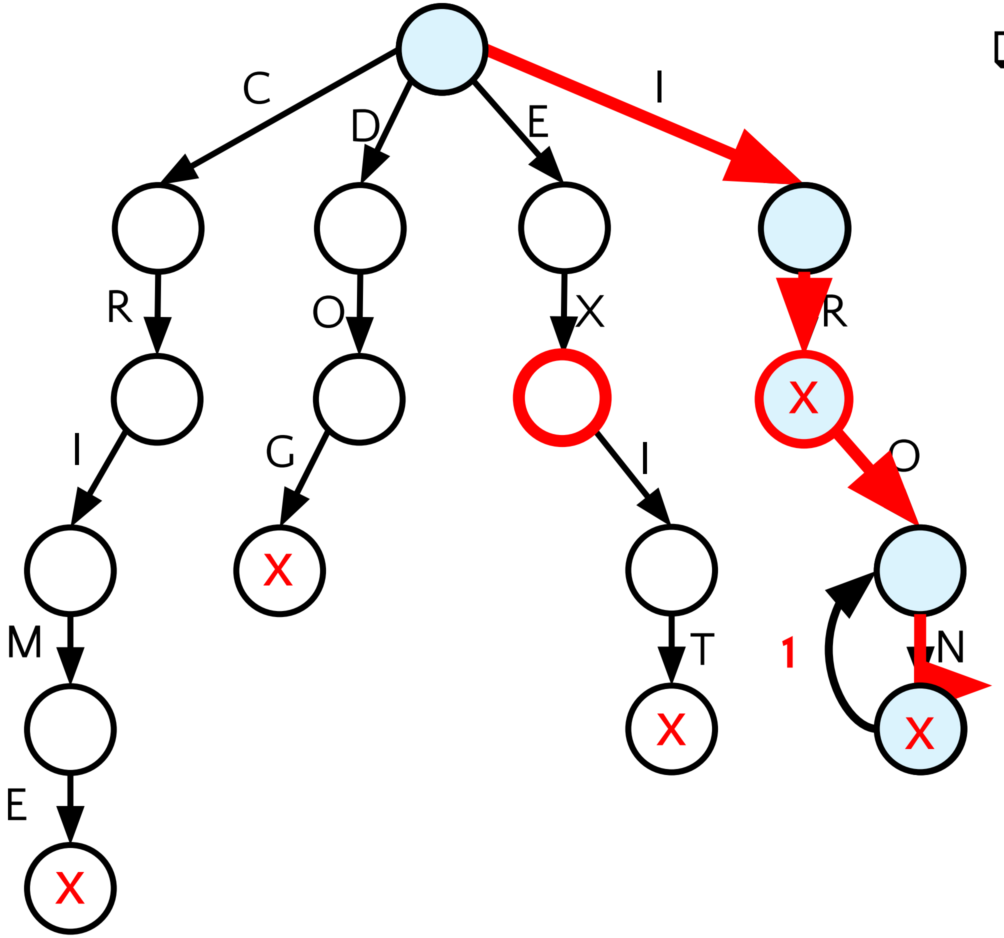
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete(“EXAM”)

DELETE FROM TRIE (CASE-2)



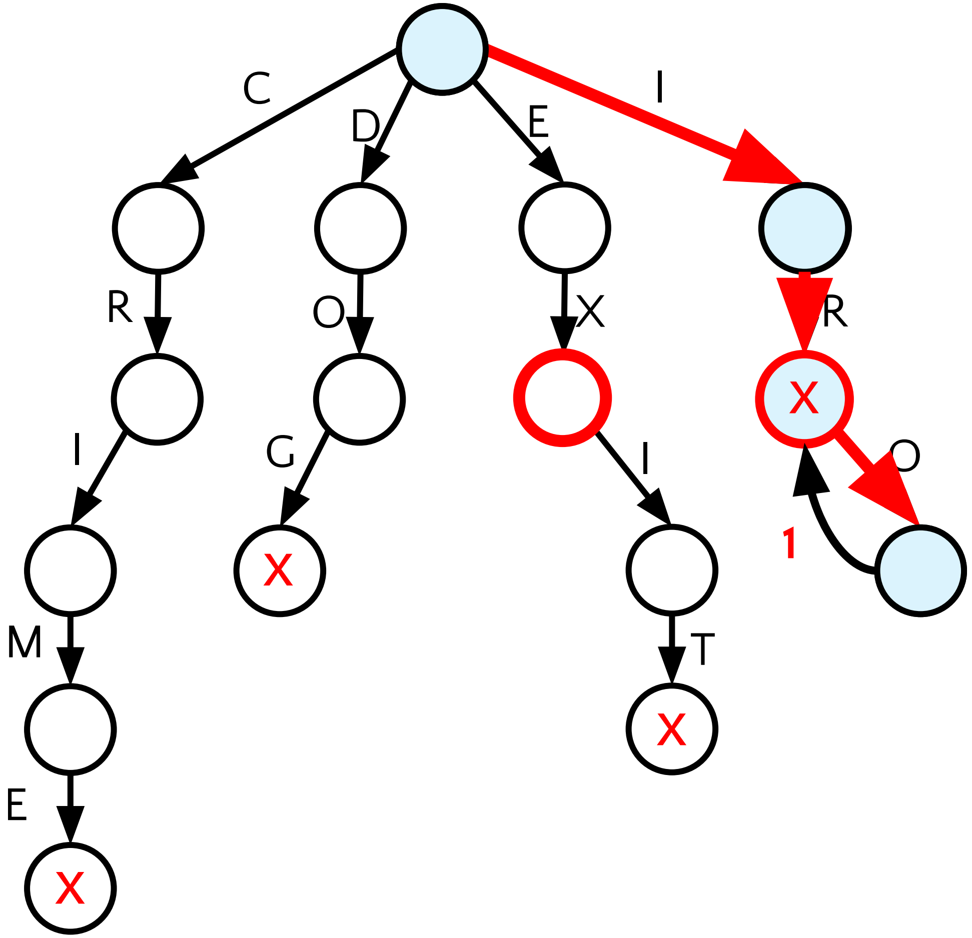
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")

DELETE FROM TRIE (CASE-2)



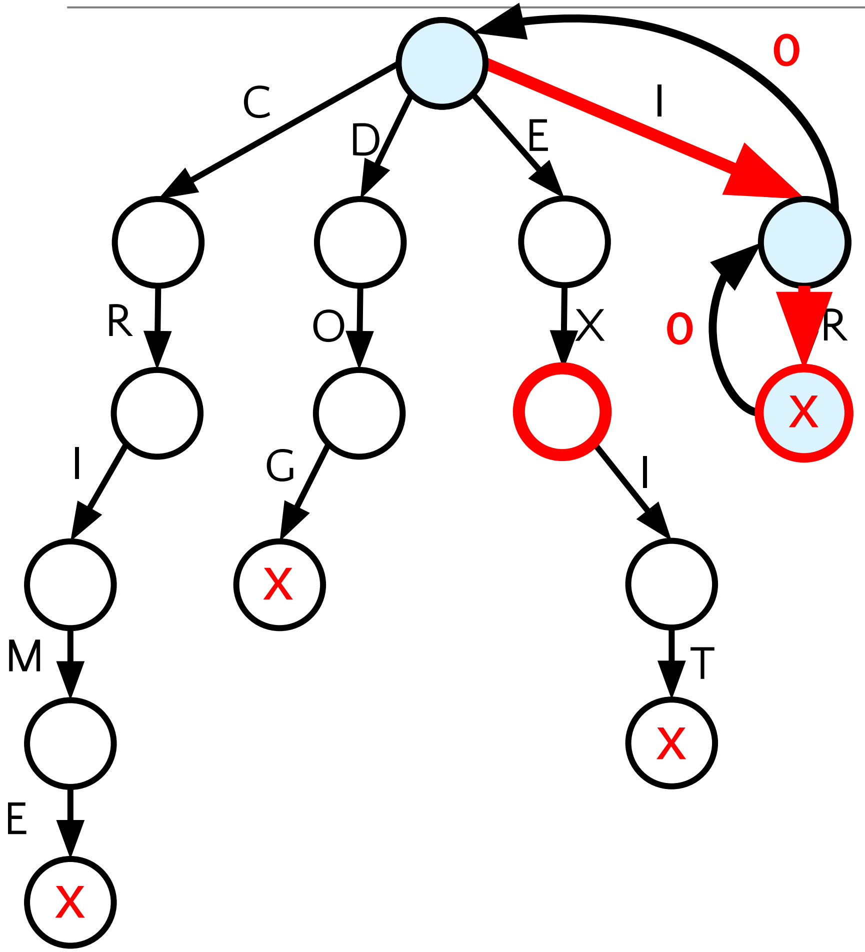
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete("EXAM")
  - delete("IRON")

DELETE FROM TRIE (CASE-2)



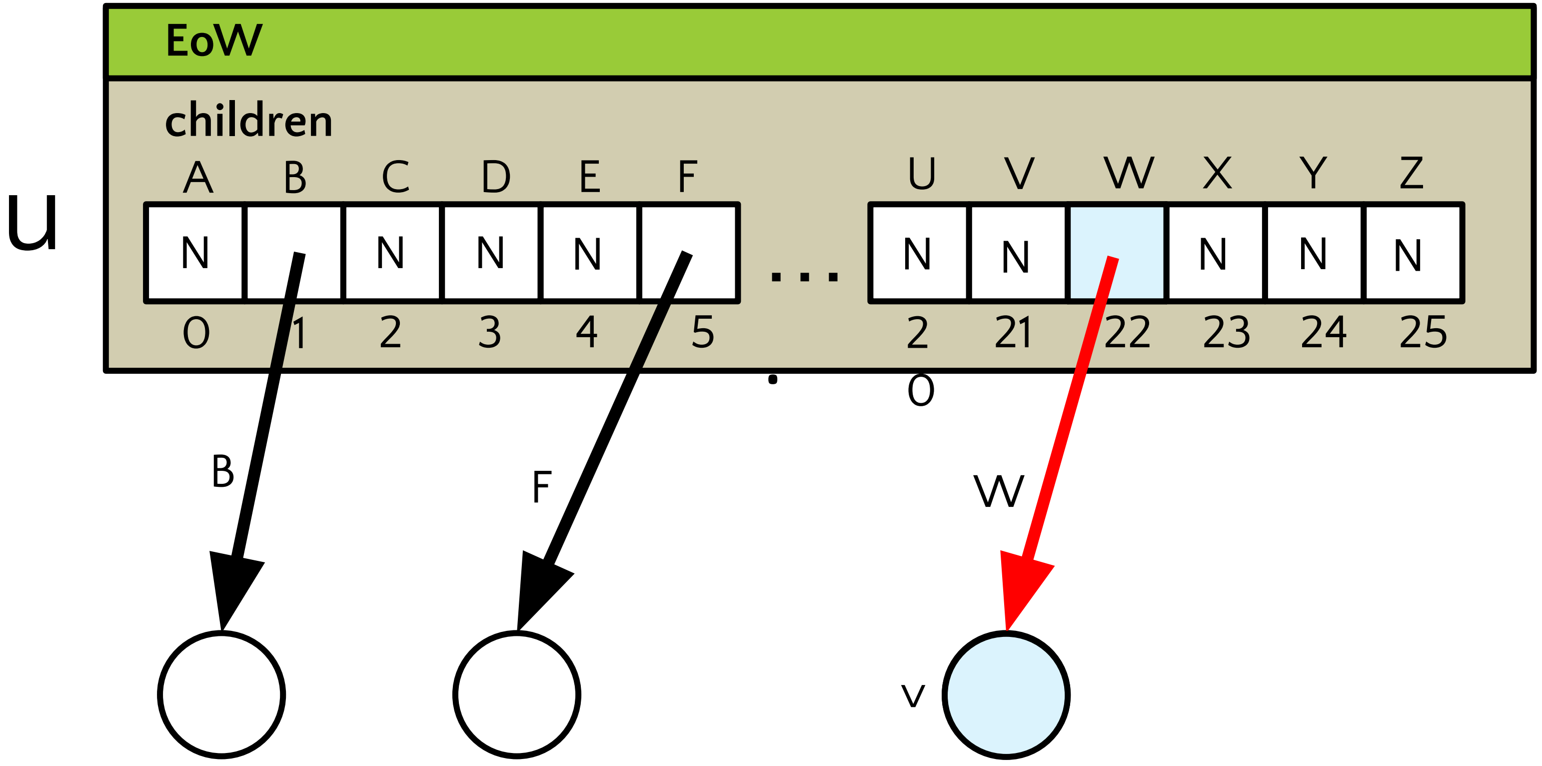
- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete(“EXAM”)
  - delete(“IRON”)

DELETE FROM TRIE (CASE-2)



- **The word is not a prefix of other words**
  - Remove all the nodes from leaf node to the first junction node associated with the string along with the edges
  - delete(“EXAM”)
  - delete(“IRON”)

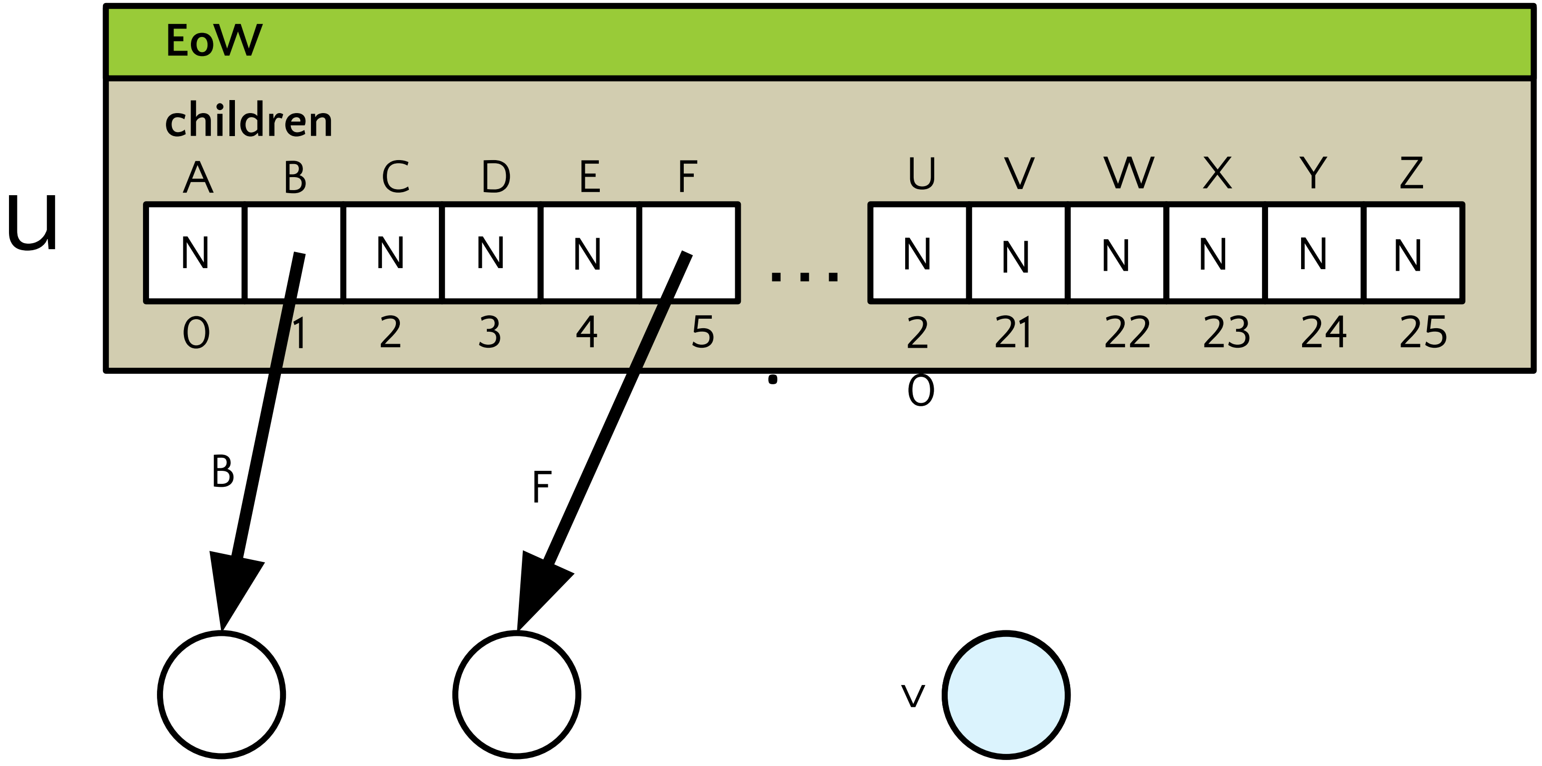
DELETION OF AN EDGE




```
r ← 22
Node *v ← u->children[r]
u->children[r] = NULL
```

DELETE THE RED MARKED EDGE

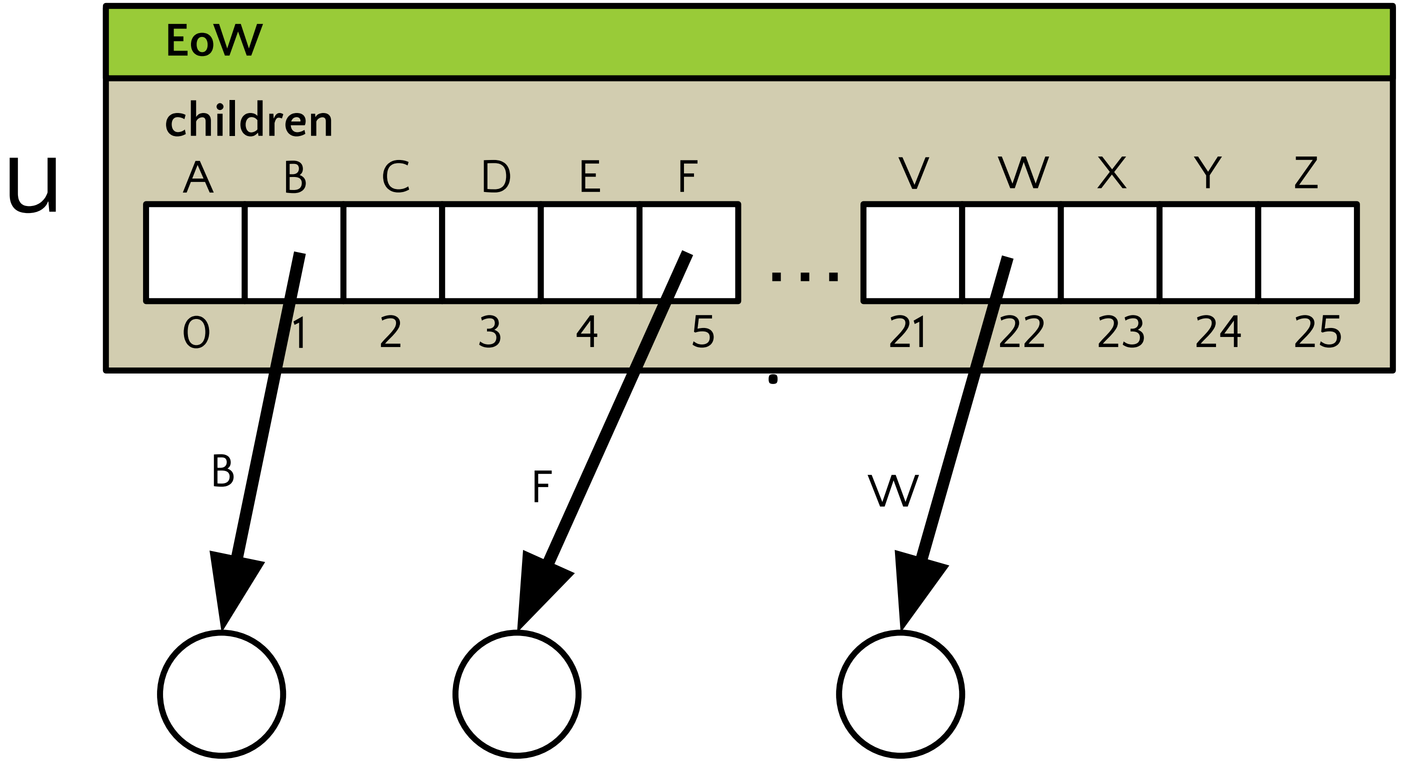
DELETION OF AN EDGE




```
r ← 22
Node *v ← u->children[r]
u->children[r] = NULL
delete v
```

DELETE THE RED MARKED EDGE

DELETION OF AN EDGE




```
deleteEdge(Node *u, char c, int d)
    if d is 0
        return without doing anything

    r ← c-65
    Node *v ← u->children[r]
    u->children[r] ← NULL
    delete v
```

DELETE IN TRIE

```
delete(string x, Node *u ← root, k ← 0)
    if u is NULL
        return 0
    if k equals size(x)
        if u->EoW is 0
            return 0
        if isLeaf(u) is false
            u->EoW = 0
            return 0
        return 1
    r ← x[k]-65
    d ← delete(x, u->children[r],k+1)
    j <- isJunction(u)
    removeEdge(u, x[k], d)
    if j is 1
        d ← 0
    return d
```

Traversing of x is not complete

r becomes the relative position of k-th character in x

d becomes 1 if the next node is removable

Otherwise d becomes 0

If u is a junction then set j variable to 1.

Removes the k-th edge of u if d permits

Then if u was a junction before removing the edge then sets the permission as 0

Then sends the permission to it's parent

JUNCTION POINT

- A node containing an EoW=1 mark
- A node having at least 2 child

```
bool isJunction(Node *u)
   count<-0
   for(int i=0;i<26;i++)
        if(u->children[i]!=NULL)    count++;
   if(u->EoW>0 or count>1)    return true;

   return false;
```

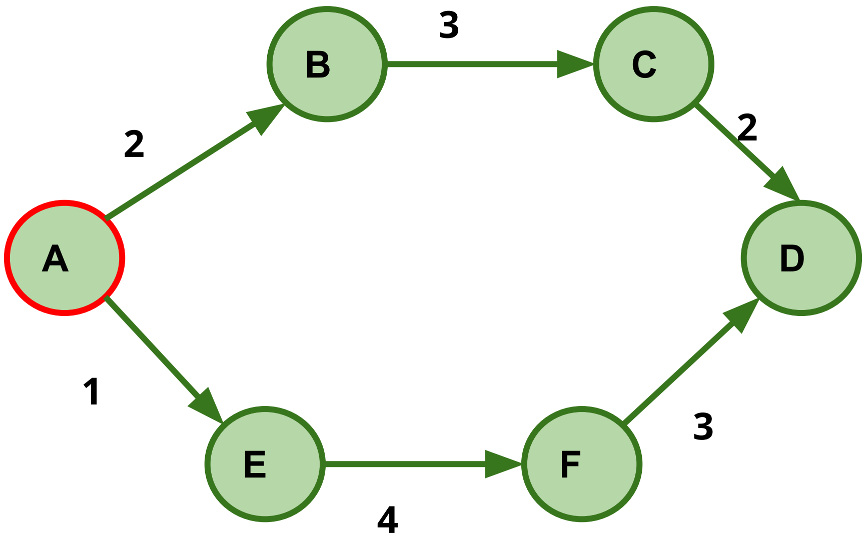
Thank You!

CSE 215: Data Structures & Algorithms II

Single Source Shortest Paths

Lec Sumaiya Nuha Mustafina
Dept Of CSE
sumaiyamustafina@cse.mist.ac.bd

Single Source Shortest Paths



Source = **A**

All possible paths:
A->B
A->B->C
A->B->C->D
A->E
A->E->F
A->E->F->D

Minimum distance from A to D is 2+3+2 = 7.
So shortest path from A to D : A->B->C->D

Shortest Path Problem

A Shortest Path Problem includes :

- ❑ A directed graph G = (V,E)
- ❑ Weight (w) associated with each edge
- ❑ Weight associated with each path

If path p = v0 -> v2 -> ... -> vk, then the weight of path p is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

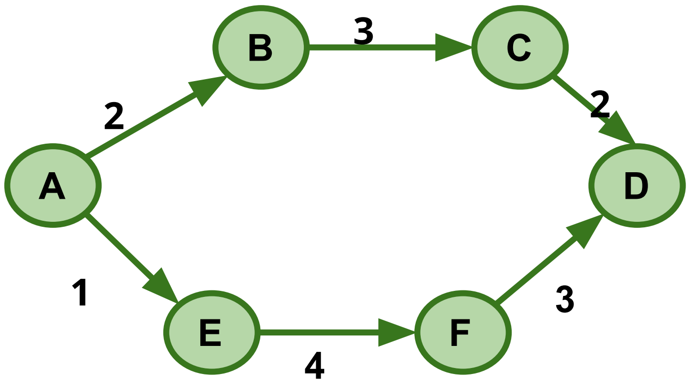
Weight of Path

The weight of path p = < v0 -> v2 -> ... -> vk > is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \,.$$

If, p = < A -> B -> C-> D>
w(p) = w (A,B) + w(B,C) +w(C,D)
= 2+3+2 =7

If, p = <A->E->F->D>
w(p) = w (A,E) + w(E,F) +w(F,D)
= 1+4+3 =8



Shortest Path Weight

The ***shortest-path weight*** δ (u, v) from u to v is defined by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v\,, \\ \infty & \text{otherwise}\,. \end{cases}$$

A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight δ (u, v)

Single-source shortest-paths problem : Variants

Single-source shortest-paths problem:
Given a graph G (V, E), we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.

Variants:

- ***Single-destination shortest-paths problem***: Find a shortest path to a given destination vertex t from each vertex .
- ***Single-pair shortest-path problem***: Find a shortest path from u to v for given vertices u and v .
- ***All-pairs shortest-paths problem***: Find a shortest path from u to v for every pair of vertices u and v.

Optimal substructure property of shortest path

Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.

****Optimal substructure is one of the key indicators that dynamic programming and the greedy method might be applied.*

Optimal substructure property of shortest path

Subpaths of shortest paths are shortest paths

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

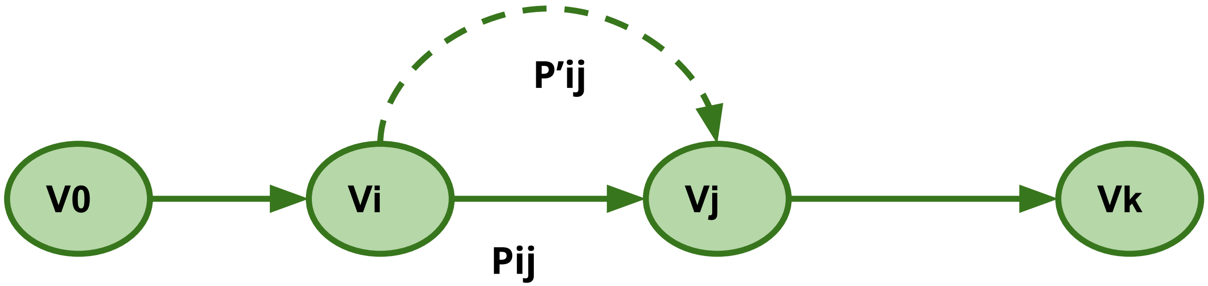
Proof!!!!

Optimal substructure property of shortest path

Subpaths of shortest paths are shortest paths

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof!!!!

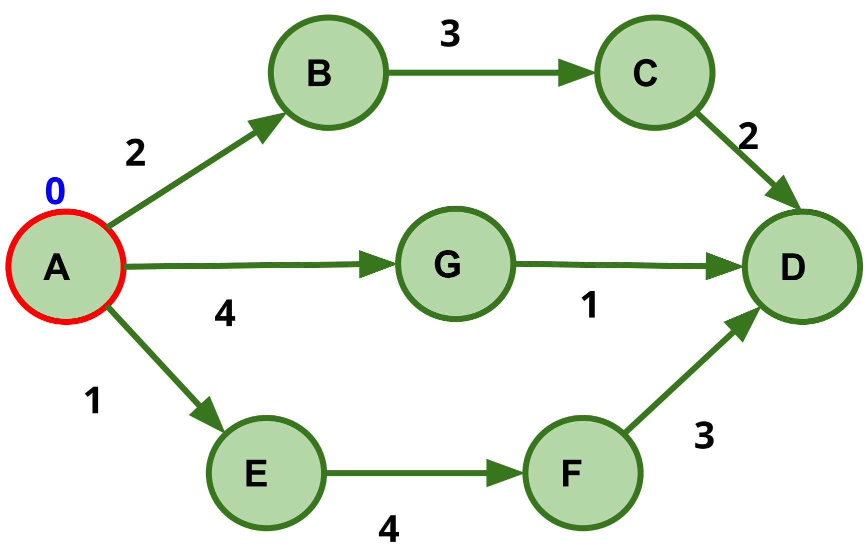


Single-source shortest-paths problem : Algorithms

- ***Dijkstra Algorithm (Greedy Method)***
  O((V+E)log V) using Binary Heap as priority queue

- ***Bellman Ford Algorithm (Dynamic Problem)***
  O(VE)

Shortest Path Problem Concept



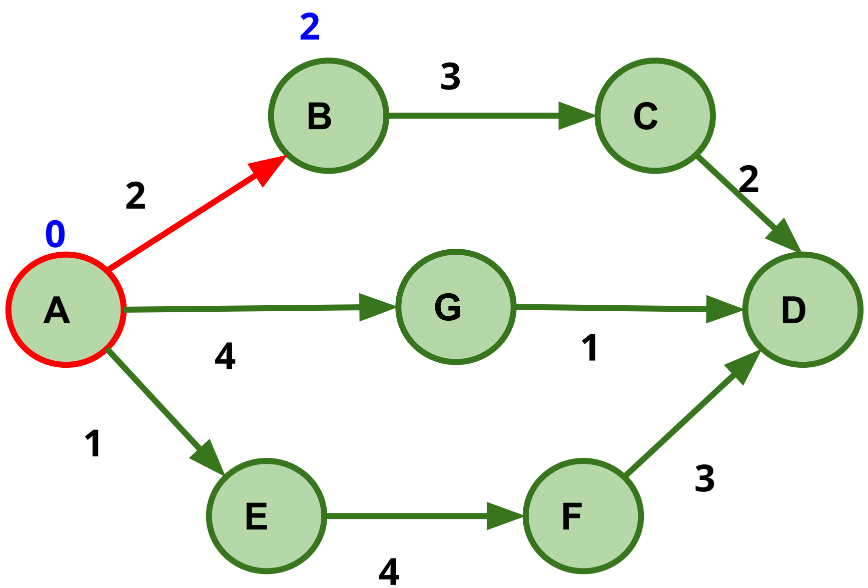
d[v] : Distance from source to vertex v

d[v] : shortest path estimate

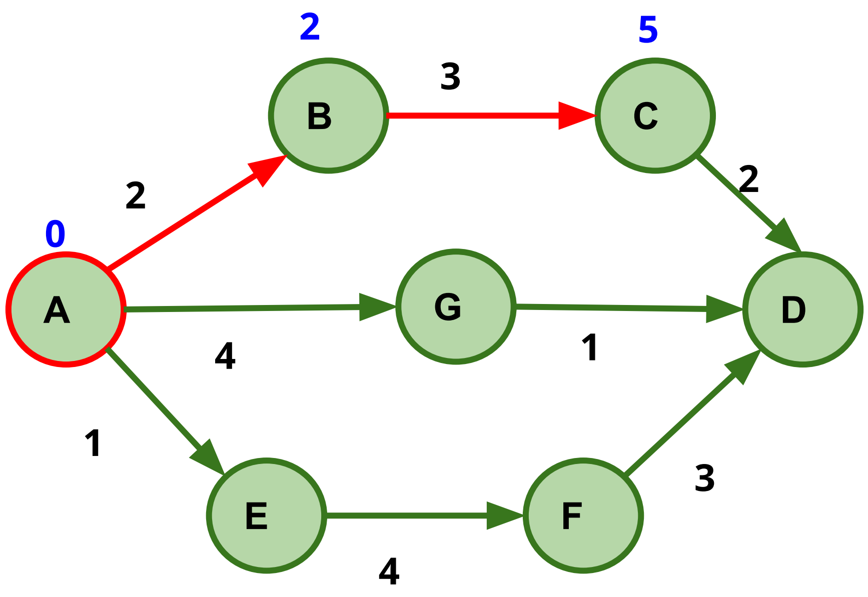
d[v] >= δ (s, v)

If d[v] = δ (s, v), it never changes

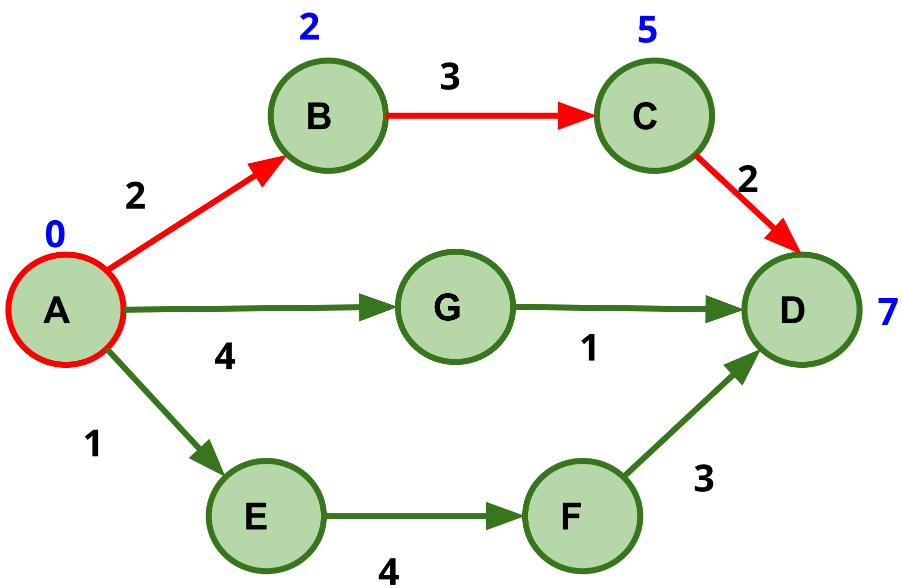
Shortest Path Problem Concept



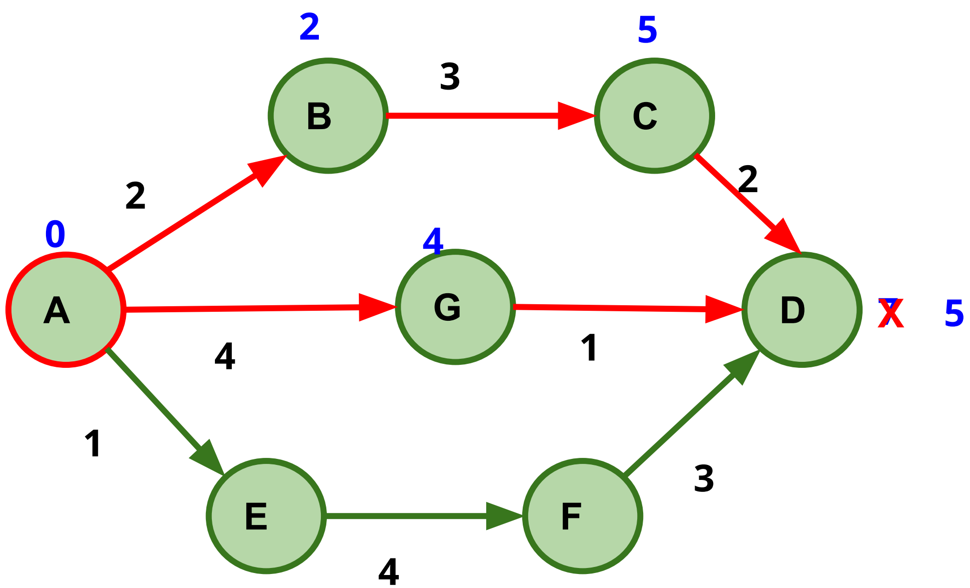
Shortest Path Problem Concept



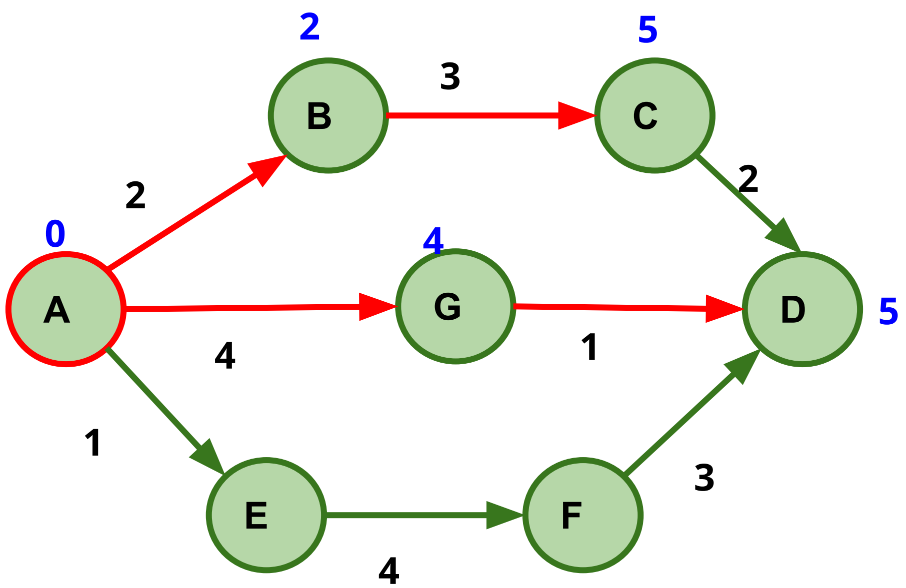
Shortest Path Problem Concept



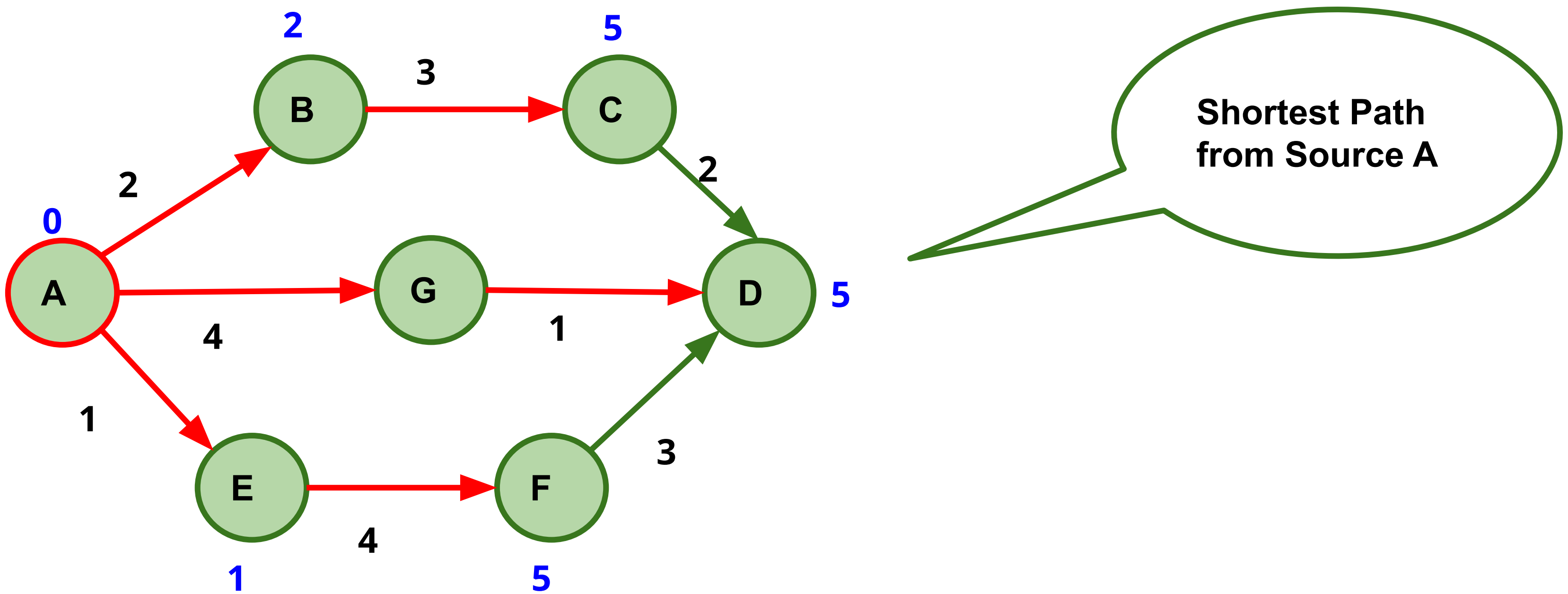
Shortest Path Problem Concept



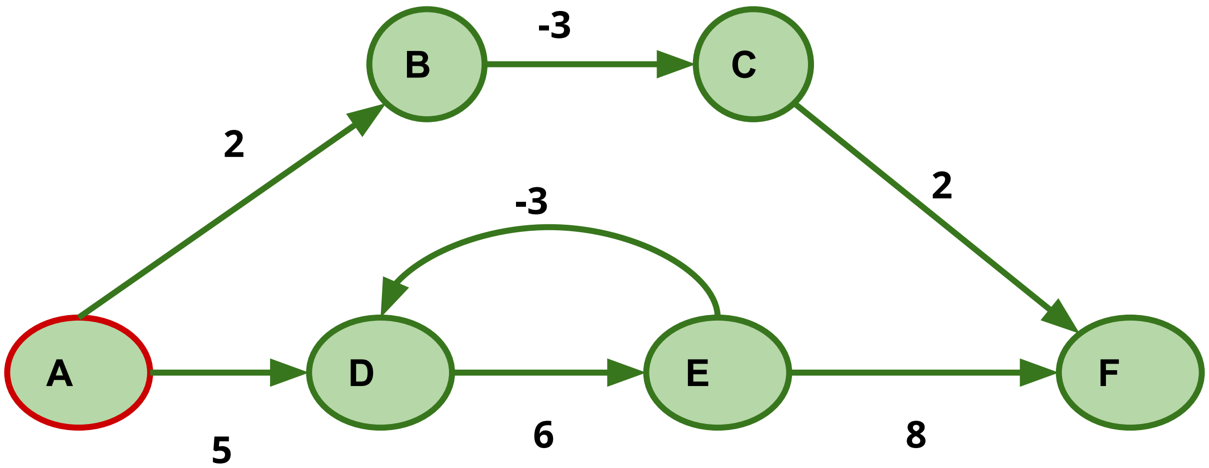
Shortest Path Problem Concept



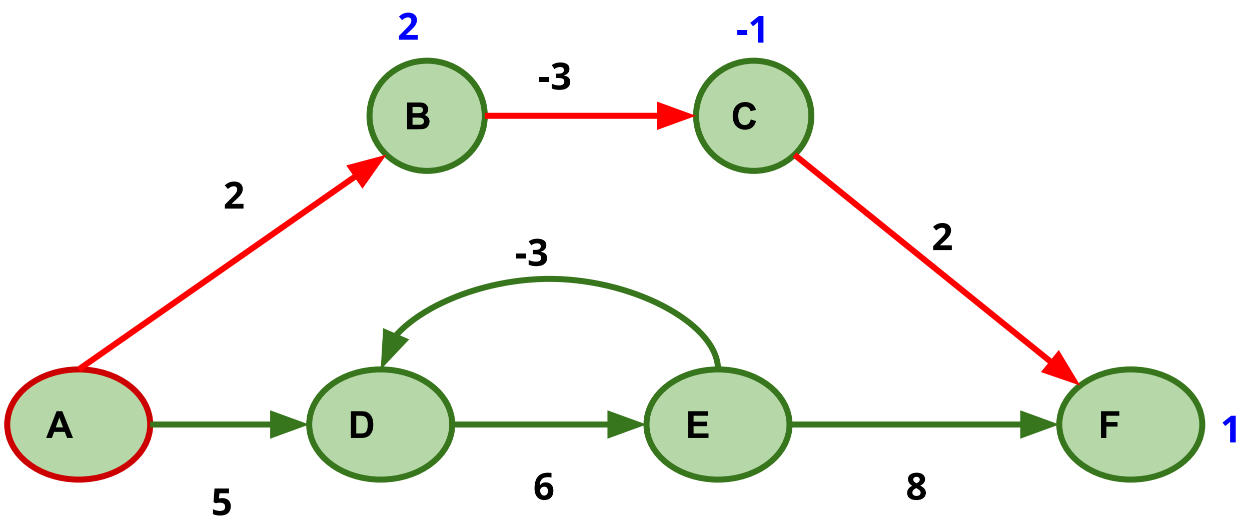
Shortest Path Problem Concept



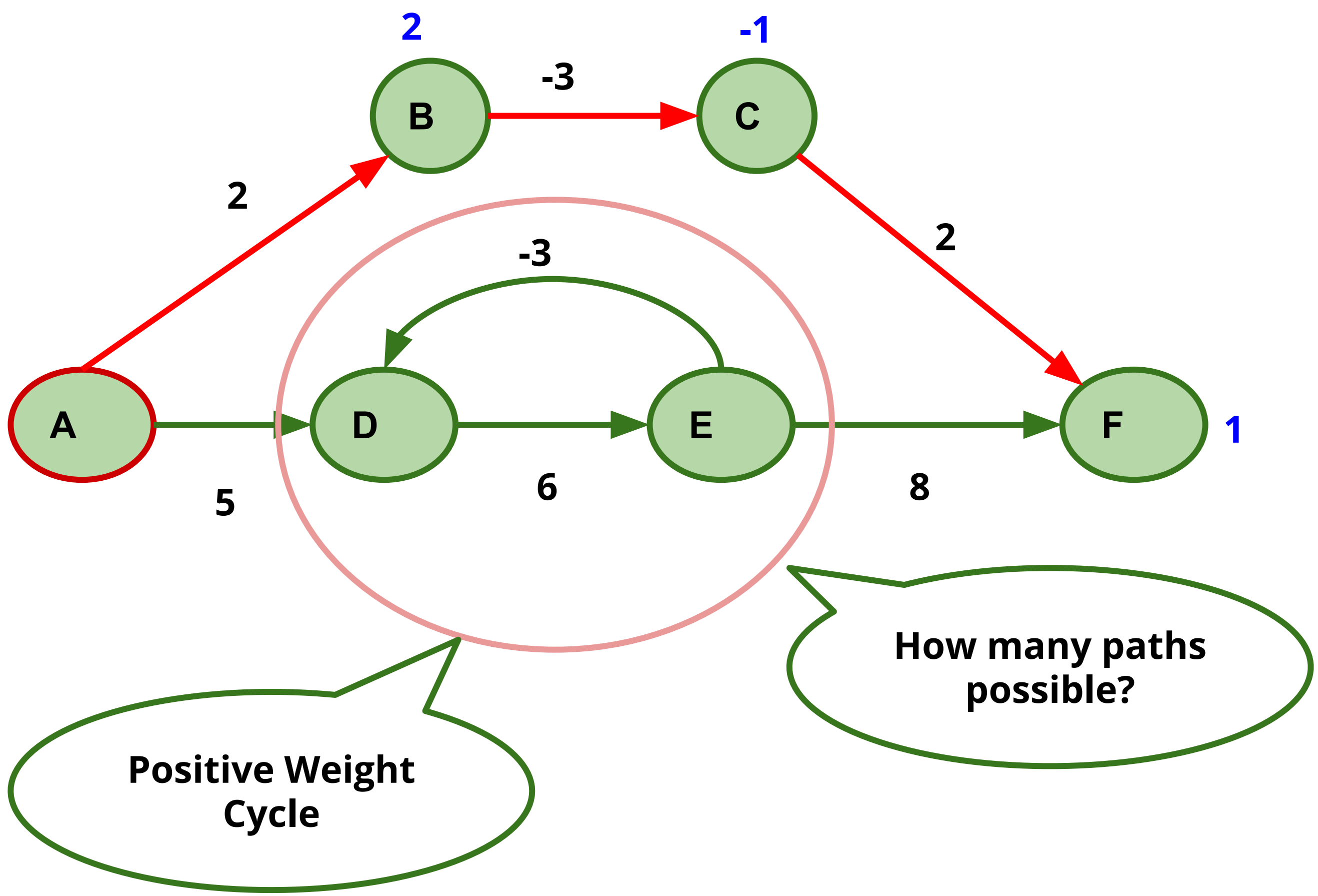
Negative Weight Edge



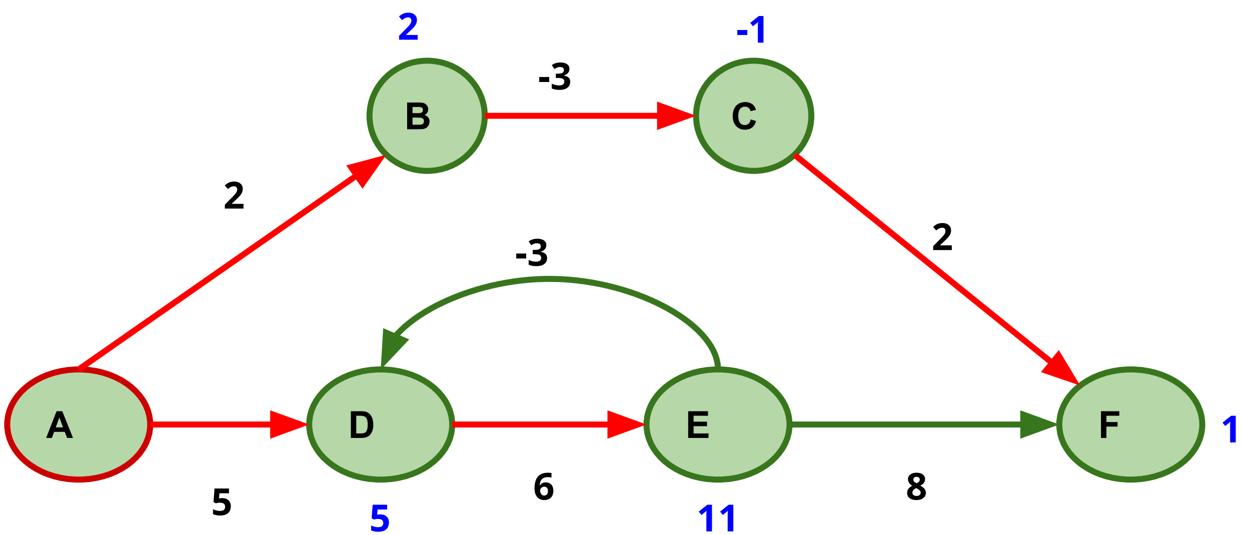
Negative Weight Edge



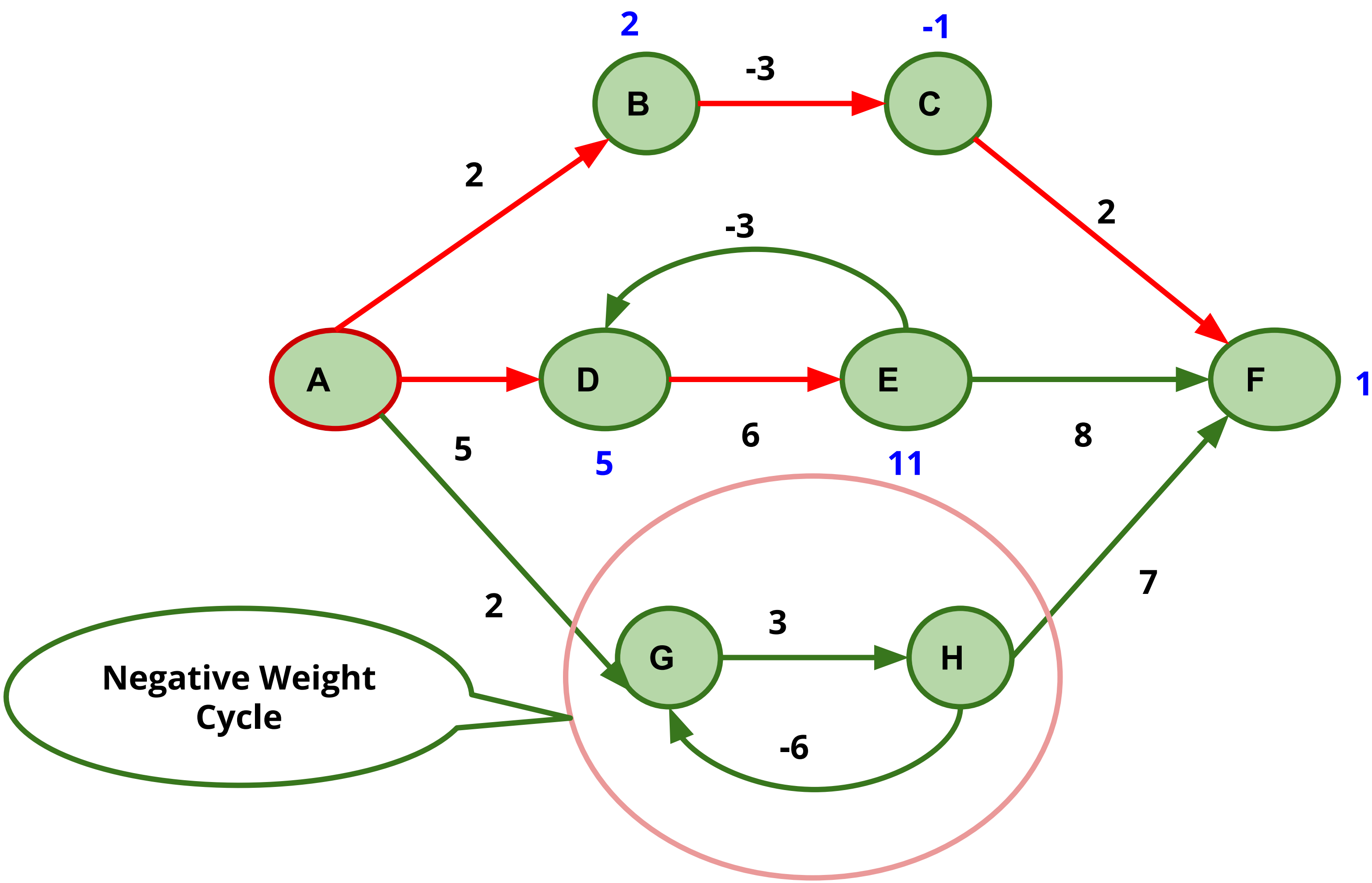
Negative Weight Edge



Negative Weight Edge

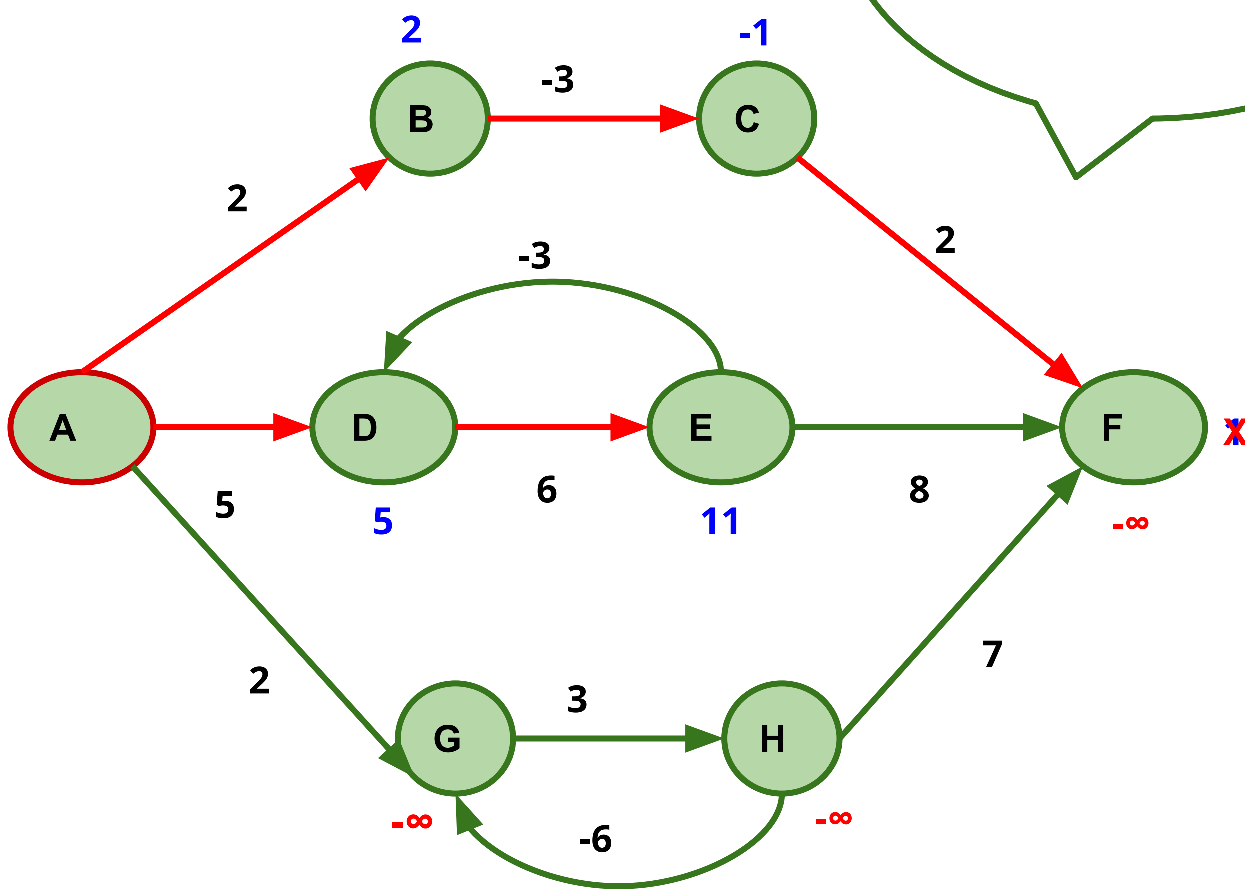


Negative Weight Edge



Negative Weight Edge

If there is a negative weight cycle on any path from s to v, we define δ (s, v) = -∞



Negative Weight Edge

- If the graph G=(V, E) contains no negative weight cycles reachable from the source s, then for all vertex v , the shortest-path weight δ (s, v) remains well defined, even if it has a negative value.

- If the graph contains a negative-weight cycle reachable from s, then, shortest-path weights are not well defined.

- If there is a negative weight cycle on any path from s to v, then δ (s, v) = **-∞**

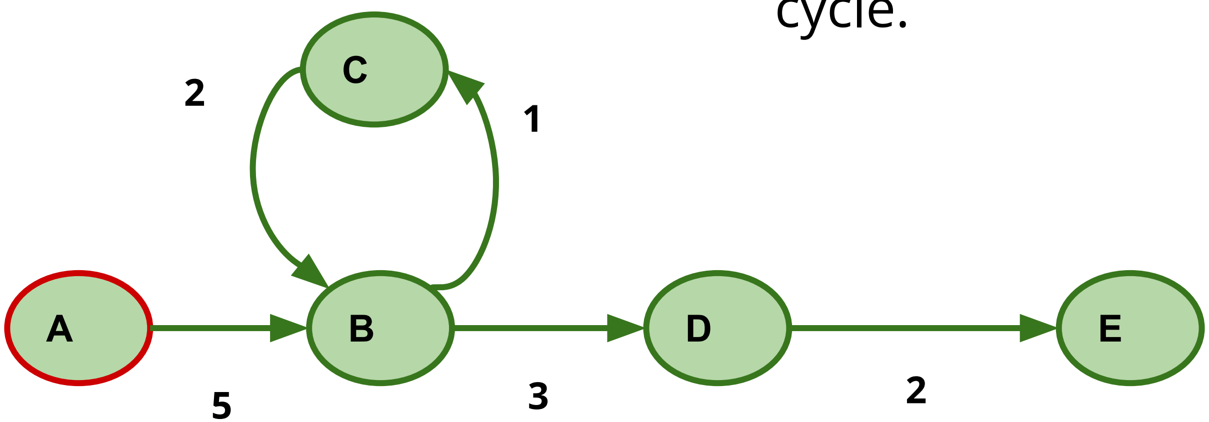
Negative Weight Edge

- ***Dijkstra Algorithm***
  Consider all edges' weights are nonnegative

- ***Bellman Ford Algorithm (Dynamic Problem)***
  - Allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source.

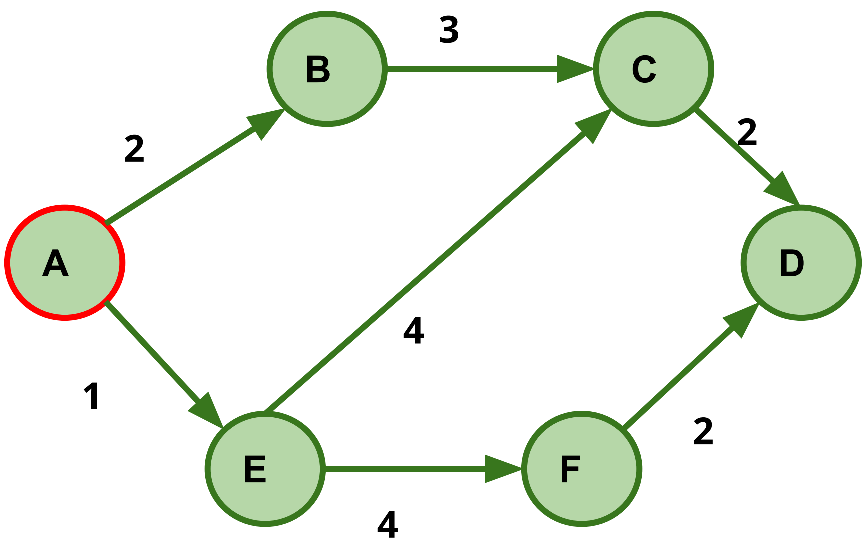
  - Can detect negative-weight cycles

Cycle in shortest path!!

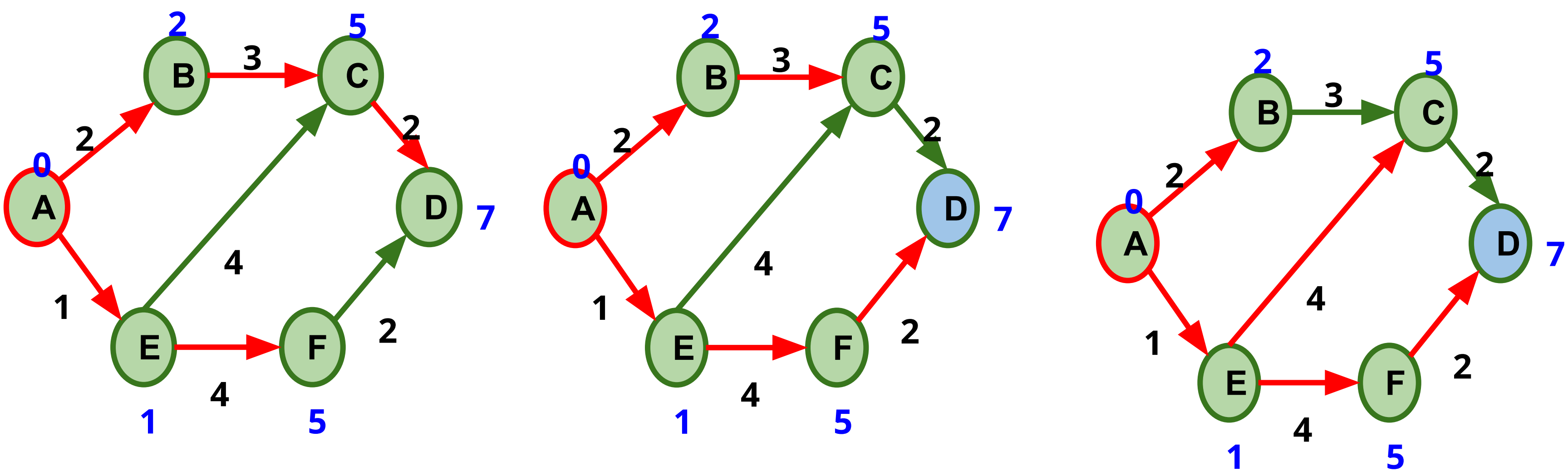


Shortest paths are simple paths with no weighted cycle.

Are Shortest Paths Unique?



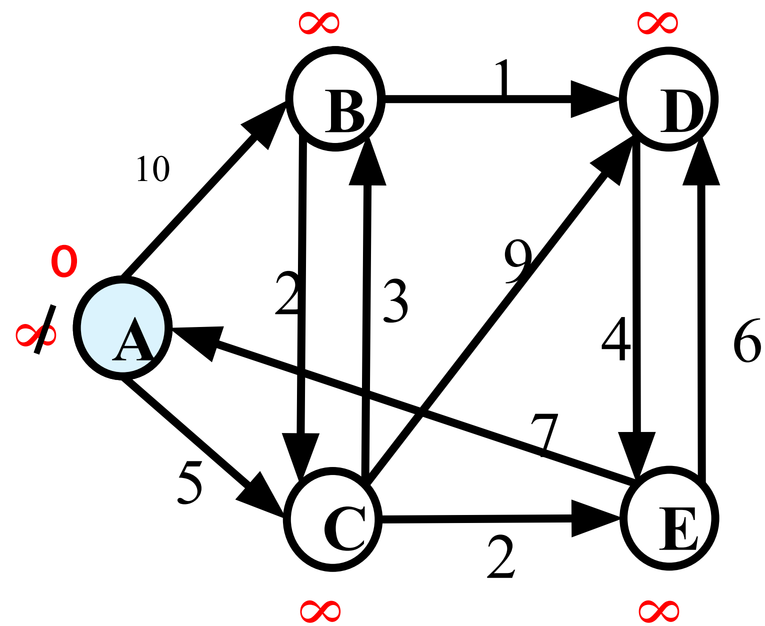
Is Shortest Path Unique?



Shortest paths are not unique.

Shortest Path Concepts

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```



Shortest Path Concepts

Relaxation:

The process of relaxing an edge (u,v) consists of
testing whether we can improve the shortest path to v found so far by going through u
and, if so, updating v.d and v.π

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.π = u
```

Properties of shortest paths and relaxation

Triangle inequality

Upper-bound property

No-path property

Convergence property

Path-relaxation property

Predecessor-subgraph property

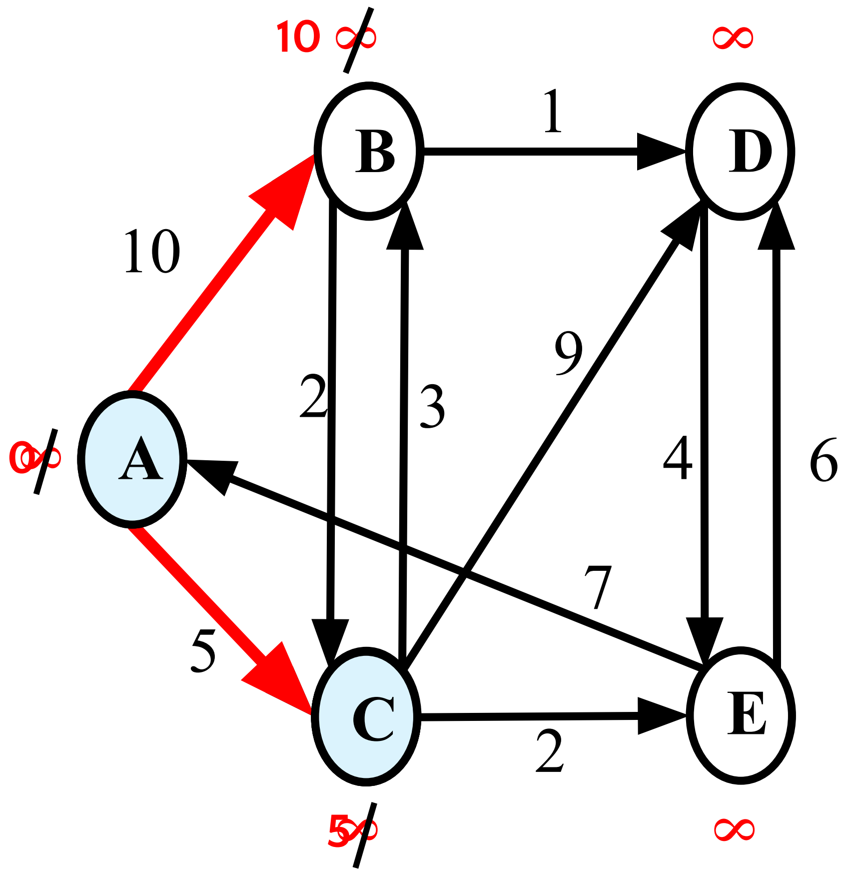
Dijkstra Algorithm

```
DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

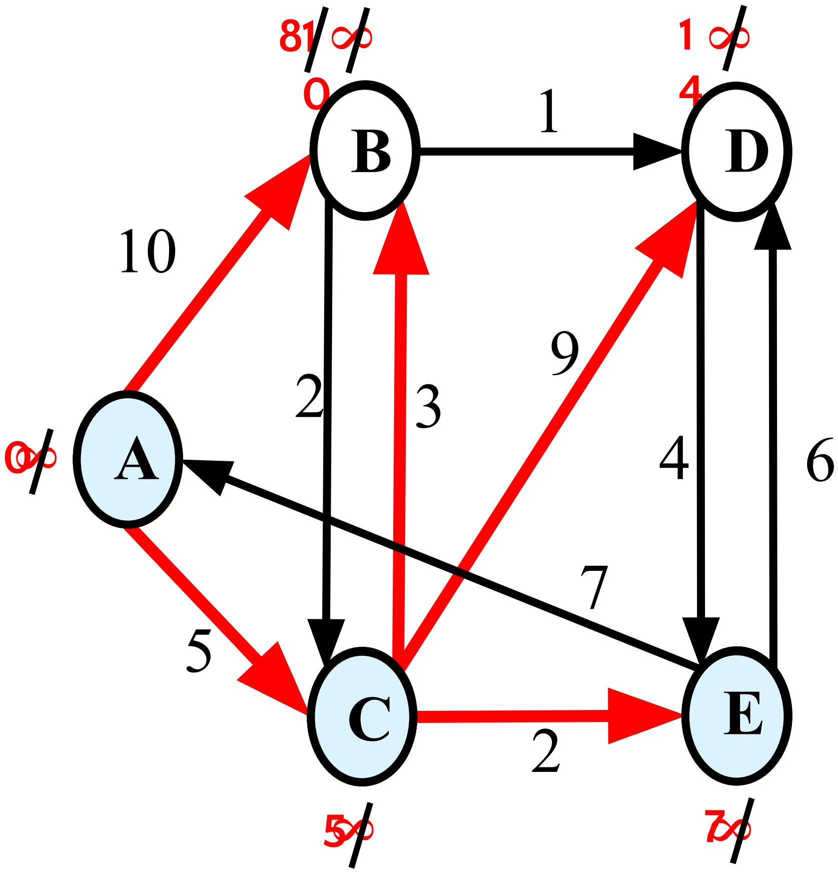
Q : Min Priority Queue with priority value based on d.

S : set of vertices whose final shortest-path weights from the source s have already been determined

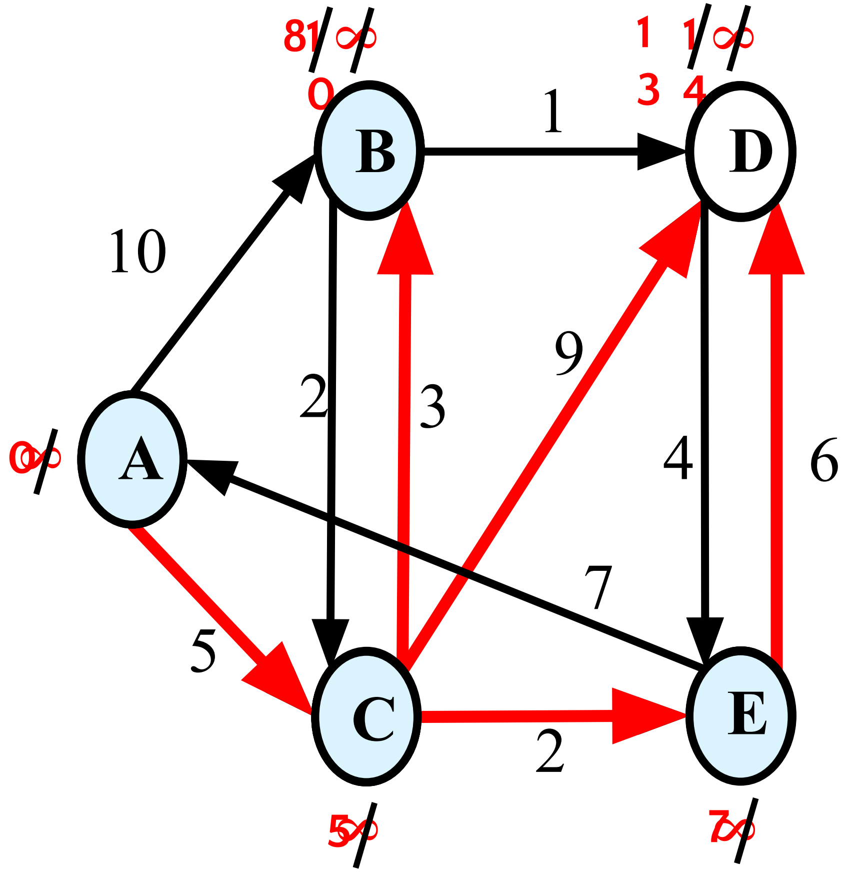
DIJSKTRA's SIMULATION



DIJSKTRA's SIMULATION

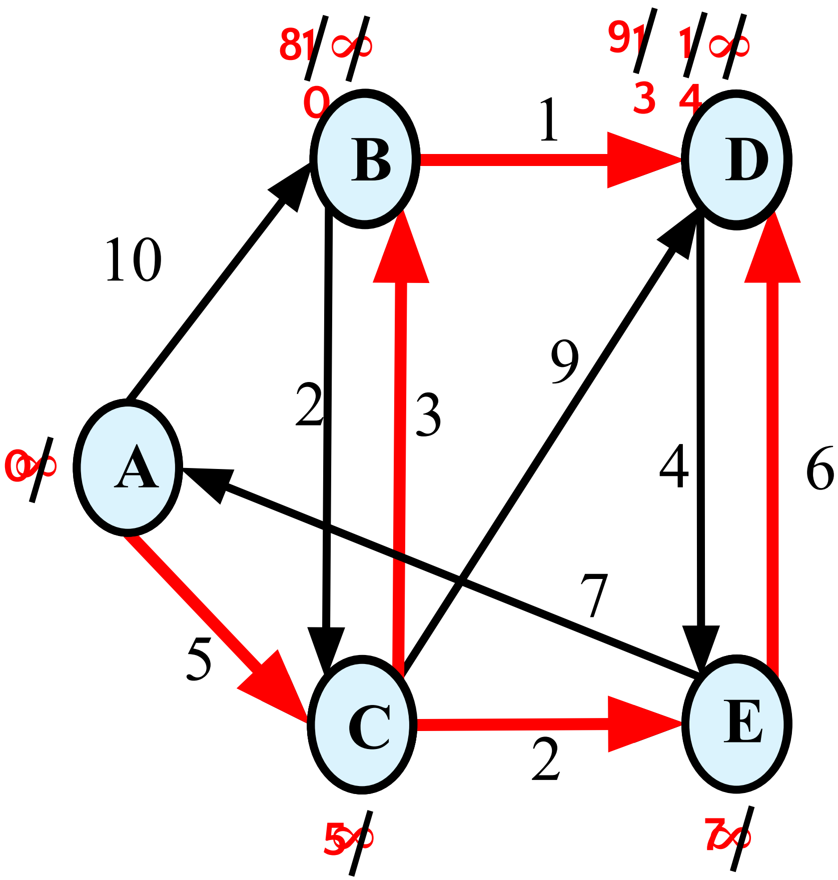


DIJSKTRA’s SIMULATION

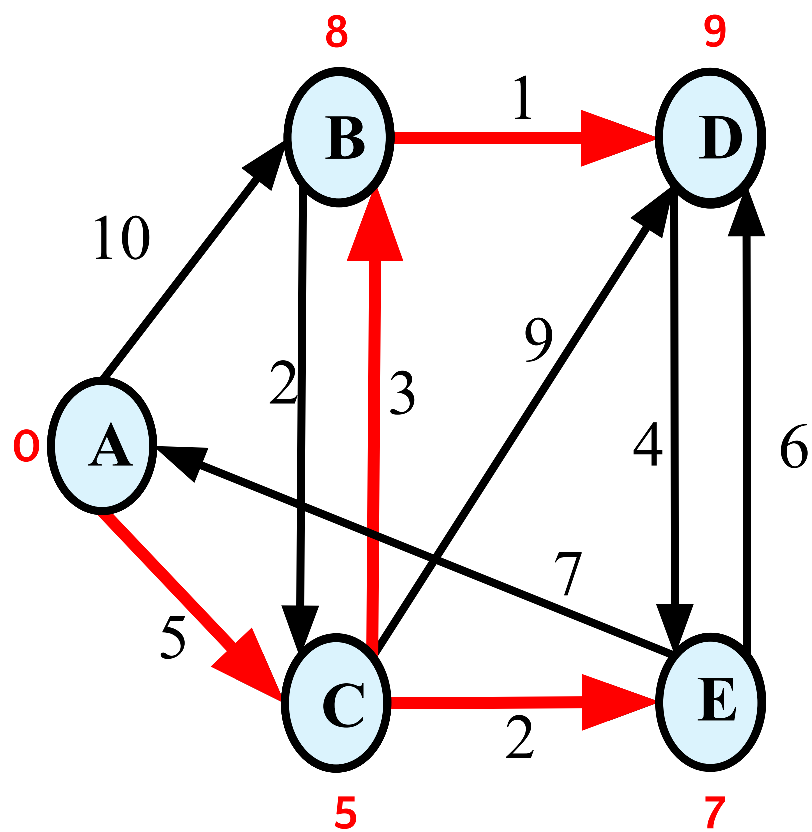


Special thanks to: Lec Swapnil Biswas

DIJSKTRA's SIMULATION

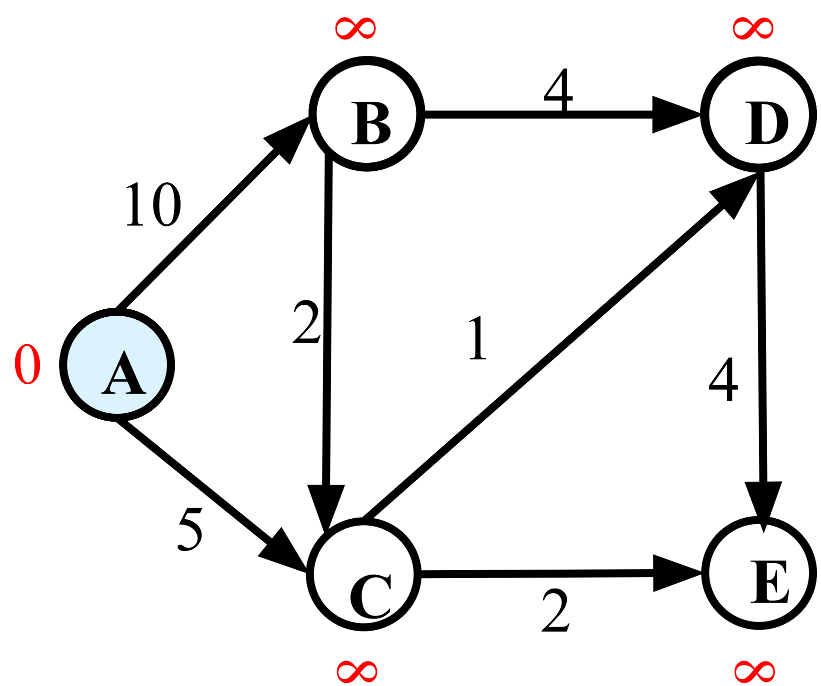


DIJSKTRA's SIMULATION



Bellman Ford Algorithm

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```



Bellman Ford Algorithm

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Finds the shortest path and distance from source

Detects Negative Weighted Cycle

Correctness of Bellman Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

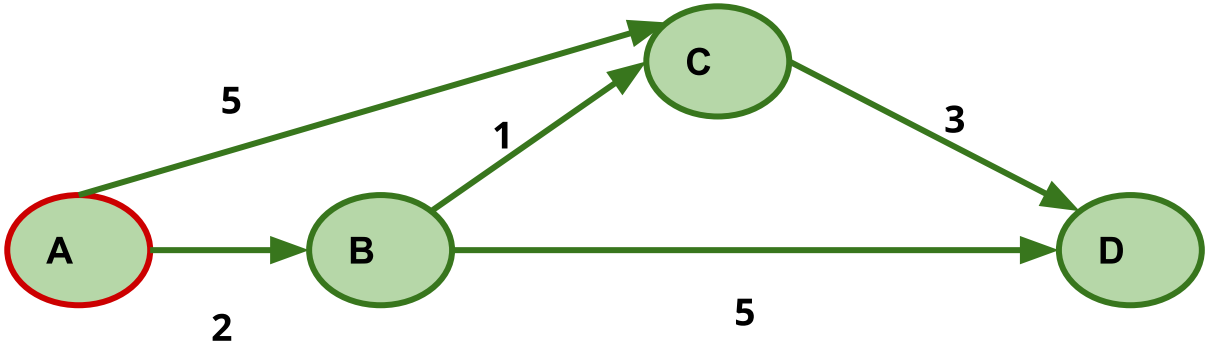
How does this part find the shortest path and distance from source ? / Correctness of Line 2-4 !!!

Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

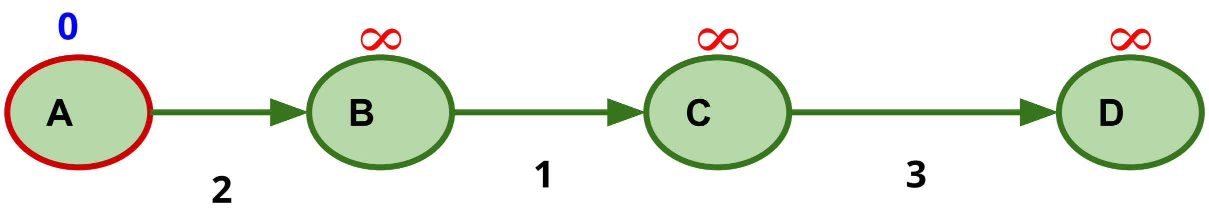


Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.



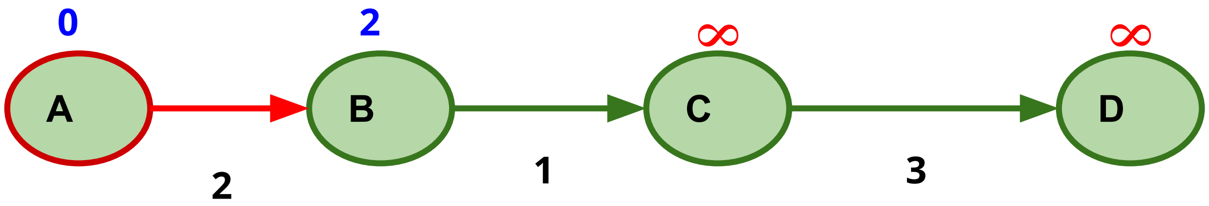
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

```
relax( A, B, 2)
```



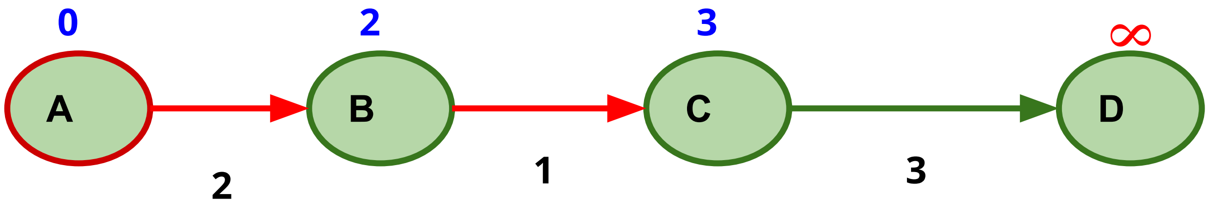
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

```
relax( B, C, 1)
```



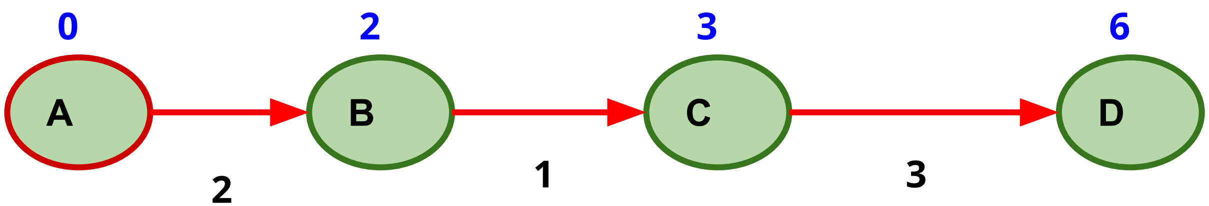
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

A.C.T path-relaxation property:

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

```
relax( C, D, 3)
```



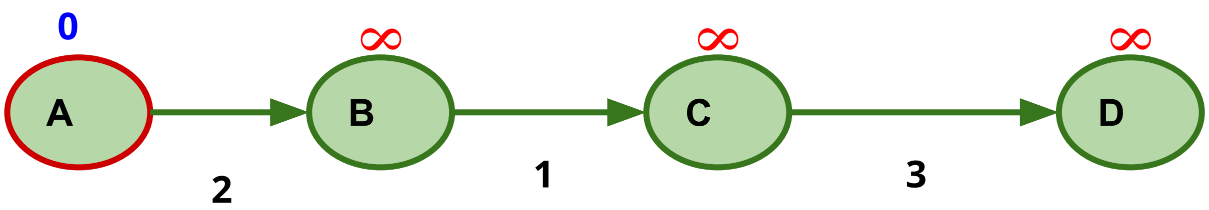
For each vertex, we found the shortest distance from the source A.
<u>Observation :</u>
If the edge relaxation order is maintained then each edge needs to be relaxed once to get the shortest distance of all vertices.

Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

What if the edge relaxation order is done differently ????

Let's assume : E= { CD, BC, AB }

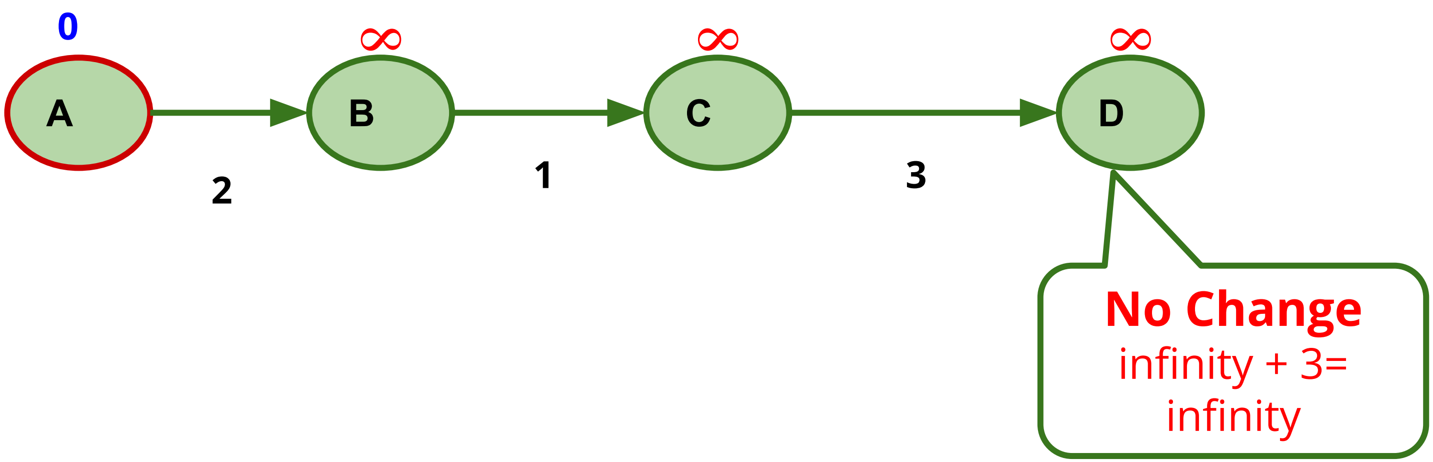


Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

E= { CD, BC, AB }

```
relax( C, D, 3)
```

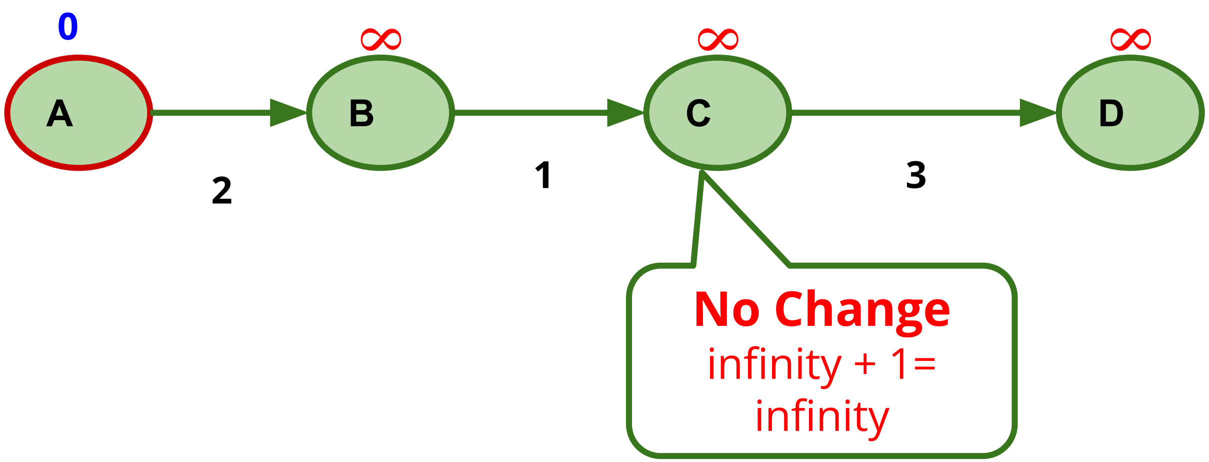


Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

E= { CD, BC, AB }

```
relax( B, C, 1)
```

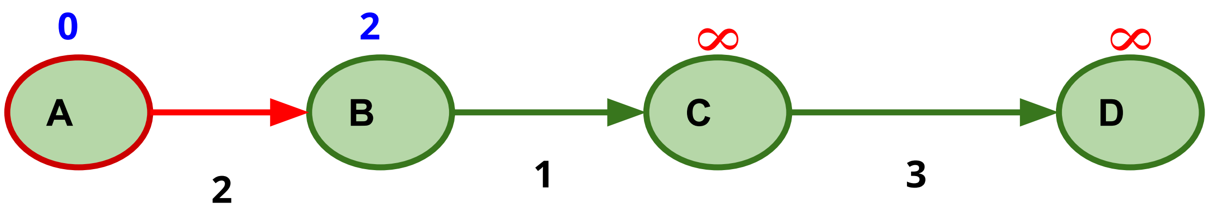


Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

E= { CD, BC, AB }

```
relax( A, B, 2)
```



After relaxing each edge once , we found the shortest distance of only one vertex.

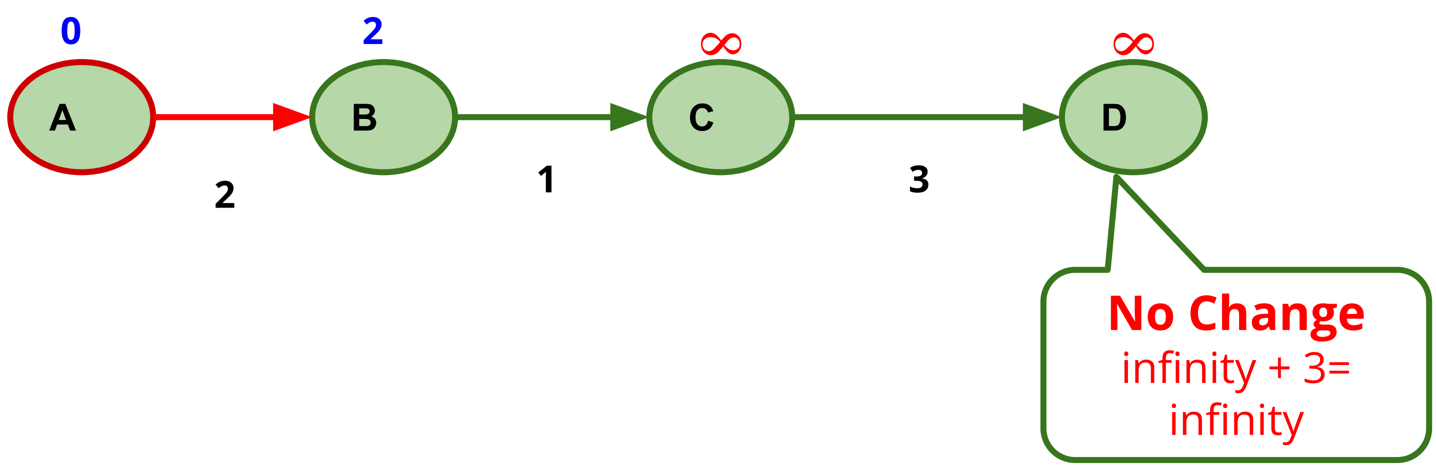
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **2nd** time**)**

E= { CD, BC, AB }

```
relax( C, D, 3)
```



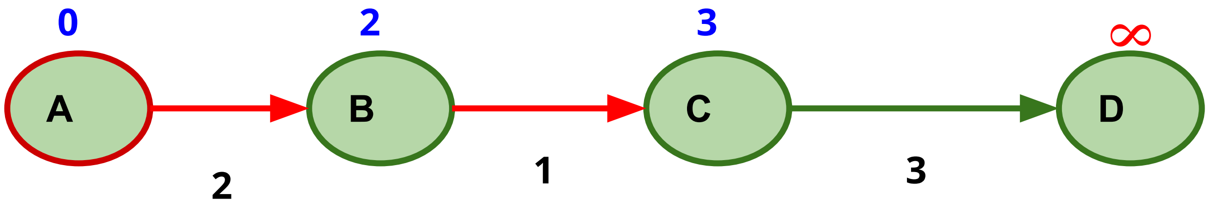
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **2nd** time**)**

E= { CD, BC, AB }

```
relax( B, C, 1)
```



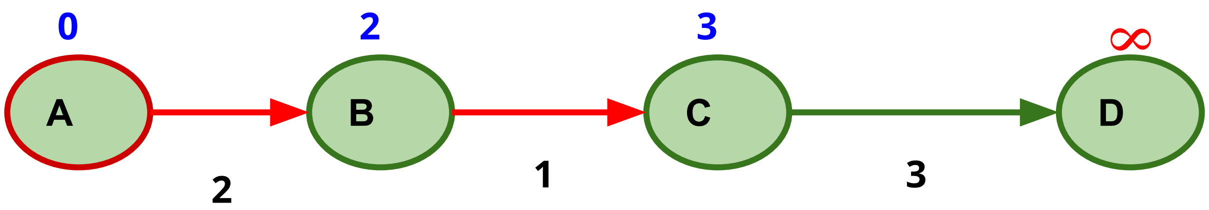
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **2nd** time**)**

E= { CD, BC, AB }

```
relax( A, B, 2)
```



*After relaxing each edge **twice** , we found the shortest distance of two vertices.*

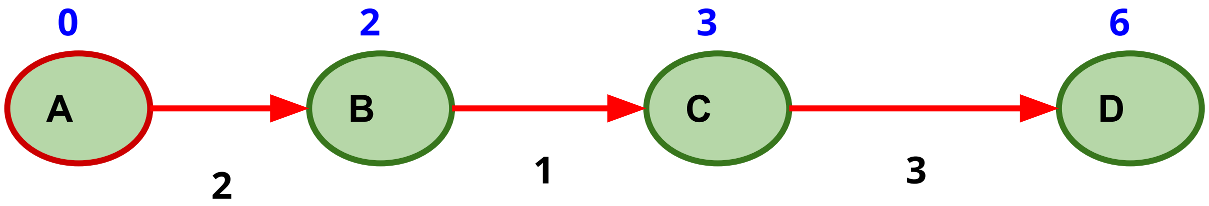
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **3rd** time**)**

E= { CD, BC, AB }

```
relax( C, D, 3)
```



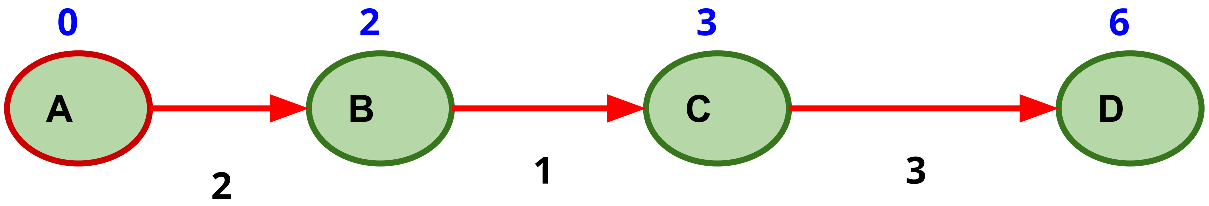
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **3rd** time**)**

E= { CD, BC, AB }

```
relax( B, C, 1)
```



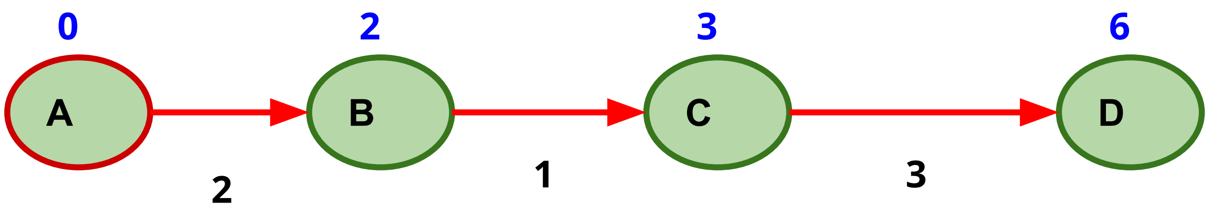
Correctness of Bellman Ford Algorithm

Correctness of Line 2-4 !!!

Let's repeat the process: (Relax the edges for **3rd** time**)**

E= { CD, BC, AB }

```
relax( A, B, 2)
```



After relaxing each edge ***thrice*** *, we found the shortest distance of three vertices.*

Correctness of Bellman Ford Algorithm

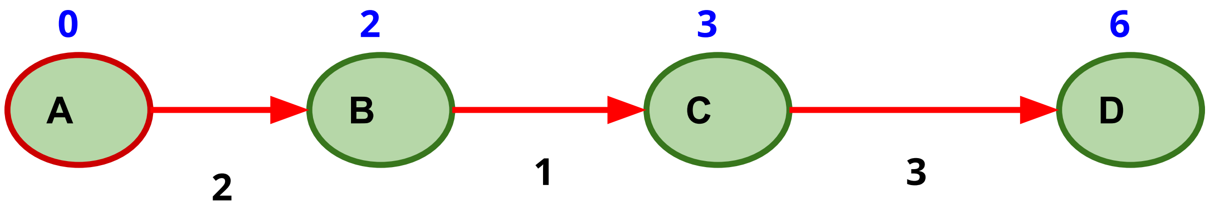
Correctness of Line 2-4 !!!

Maximum Number of edges in a shortest path = V-1

In the example shown, V= {A,B,C,D} and *E= { CD, BC, AB }*

*And after relaxing each edge **thrice (V-1 times)** , we found the shortest distance of all vertices.*

Thus, lines 2-4 find the v.d = δ(s,v) for each vertices v ∈ V after relaxing each edge V-1 times.



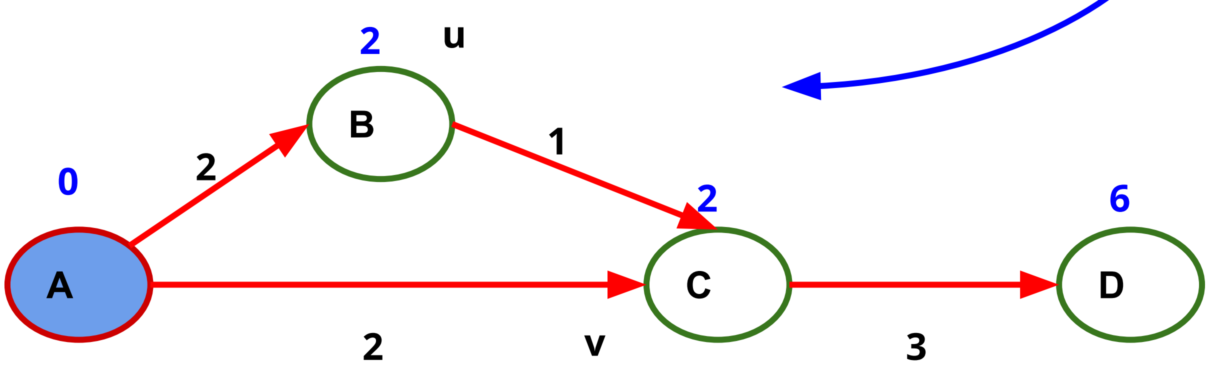
Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning true

Let's assume graph has no negative weight cycle reachable from source.

A.C.T triangle inequality property,

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.



Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning true

Let's assume graph has no negative weight cycle reachable from source.

Lines 2-4 find the v.d = δ(s,v) for each vertices v ∈ V after relaxing each edge V-1 times.

A.C.T triangle inequality property,

For any edge $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.

$$
\begin{aligned}
v.d &= \delta(s, v) \\
&\le \delta(s, u) + w(u, v) \quad \text{(by the triangle inequality)} \\
&= u.d + w(u, v) ,
\end{aligned}
$$

So none of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning False

Let’s assume graph has negative weight cycle C reachable from source.

Let, C = <V0,V1,.....,Vk>, where V0 =Vk.
So,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 .$$

Let’s assume, for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE.

Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning False

Let's assume, for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. So,

$$v_i.d \le v_{i-1}.d + w(v_{i-1}, v_i) \text{ for } i = 1, 2, \ldots, k.$$

Summing the inequalities around cycle c gives us

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\le \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) . \end{aligned}$$

Now, in circle, V0=Vk. So,

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d .$$

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Correctness of Bellman Ford Algorithm

Correctness of the algorithm returning False

But,

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i) \quad \xleftrightarrow{\text{Contradicts}} \quad \sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 .$$

We conclude that the Bellman-Ford algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise.

Correctness of Dijkstra's algorithm

Self Study:
Correctness of Dijkstra Algorithm (Theorem 24.6)

Reference:
Introduction to ALGORITHMS: THOMAS H. CORMEN (3rd Edition)


Thank you

CSE 215: Data Structures & Algorithms II

Dynamic Programming : LCS

Lec Sumaiya Nuha Mustafina
Dept Of CSE
sumaiyamustafina@cse.mist.ac.bd

Dynamic Programming

Dynamic programming :

- Applied to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

- Solves every subproblems once and stores it in a table.

- Dynamic programming refers to a tabular method, not computer code.

Dynamic Programming

Two key characteristics that a problem must have for dynamic programming to be a viable solution technique.

- Optimal substructure property.
- Overlapping subproblems property.

*Recall: *Optimal substructure is one of the key indicators that dynamic programming and the greedy method might be applied.*

Dynamic Programming

Optimal substructure property.

- The optimal substructure property states that an optimal solution to a problem contains **optimal solutions to its sub problems**.
- In simpler terms, if you can solve a larger problem by breaking it down into smaller subproblems, and the solution to the larger problem relies on the solutions to those sub problems, then the problem exhibits optimal substructure.

Overlapping subproblems property.

- Overlapping subproblems occur when a problem can be broken down into sub problems which are reused several times.
- This means that the same sub problem is solved multiple times in the process of solving the larger problem.
- Dynamic programming takes advantage of this property by solving each sub problem **only once** and storing the results (usually in a table or array) so that when the same subproblem is encountered again, it can be quickly retrieved from the table instead of being recalculated.

Dynamic Programming

There are usually ***two equivalent ways*** to implement a dynamic-programming approach.

The first approach is ***top-down with memoization***.

- ❑ Memoization is derived from the Latin word "memorandum" ("to be remembered").
- ❑ In this approach, the procedure is written recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).
- ❑ Recursive code + Memoization code
- ❑ The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner.
- ❑ The recursive procedure is said to be memoized as it "remembers" what results it has computed previously.

Dynamic Programming

There are usually ***two equivalent ways*** to implement a dynamic-programming approach.

The second approach is ***the bottom-up method (Tabulation)***.

- This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems.
- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.

Dynamic Programming

The development of dynamic-programming, is broken into a sequence of ***four steps:***

1. Characterize the structure of an optimal solution.

1. Recursively define the value of an optimal solution.

1. Compute the value of an optimal solution, typically in a bottom-up fashion.

1. Construct an optimal solution from computed information.

Longest common subsequence (LCS)

LCS:

- is defined as the longest subsequence that is common to all the given sequences.
- the elements of the subsequence need not be consecutive.
- but the sequence or order will be maintained.

Let's say, X = ABCD Y= JBAGHCED

AB**CD** JB**A**GH**C**E**D**

length of LCS : 3

longest common subsequence is: **ACD**

LCS is a optimization problem as we are try to perform maximization here.

Longest common subsequence (LCS)

X = **ABCBDAB**
Y= **BDCABA**

Find the **LCS** of X and Y.

One of the LCS of X and Y = **BCBA**

Longest common subsequence (LCS)

X = **ABCBDAB**
Y= **BDCABA**

Find the **LCS** of X and Y.
One of the LCS of X and Y = **BCBA**

***Brute Force Approach of finding LCS*:**

- Enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find.
- Each subsequence of X corresponds to a subset of the indices {1,2,....,m} of X.
- Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

Longest common subsequence (LCS)

Solving LCS using Dynamic Programming:
Step 1: Characterizing a longest common subsequence

The LCS problem has an optimal-substructure property.

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof : Theorem 15.1 (3rd Edition)
Self Study

Prefix: defined as **i**th prefix of X, for i=0, 1,.... ,m, as $\mathbf{X_i = \langle x_1, x_2, \ldots, x_i \rangle}$.
For example, if X=<A,B,C,B,D> , then $\mathbf{X_2}$=<A,B> and $\mathbf{X_0}$ is the empty sequence.

Longest common subsequence (LCS)

Solving LCS using Dynamic Programming:
Step 2: A recursive solution

The optimal-substructure property implies that we should examine either one or two subproblems when finding an LCS of
$\mathbf{X=\langle x_1, x_2, \ldots, x_m\rangle}$ **and** $\mathbf{Y=\langle y_1, y_2, \ldots, y_n\rangle}$**.**

- If $\mathbf{x_m=y_n}$:
  we must find an LCS of $\mathbf{X_{m-1}}$ **and** $\mathbf{Y_{n-1}}$. *Appending* $x_m = y_n$ to this LCS yields an LCS of X and Y .

- If $\mathbf{x_m \neq y_n}$:
  then we must solve two subproblems:
  finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}.
  Whichever of these two LCSs is longer is an LCS of X and Y .

Longest common subsequence (LCS)

Solving LCS using Dynamic Programming:
Step 2: A recursive solution

Let, ***c[i , j]*** be the length of an LCS of the sequences $\mathbf{X}_i$ and $\mathbf{Y}_j$
Thus optimal substructure of the LCS problem gives the recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS

X = **ABCBDAB**
Y= **BDCABA**

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Bottom-Up)

*It computes the entries in **row-major order**. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.)*

The procedure also maintains the table b=[1..m; 1..n] to help in constructing an optimal solution.

X = **ABCBDAB**
Y= **BDCABA**
LCS-LENGTH(X,Y)

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

O(mn)

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Bottom-Up)

X = **ABCBDAB**
Y= **BDCABA**

LCS-LENGTH(X,Y)

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
        c[i,0] = 0
for j = 0 to n
        c[0, j] = 0
for i = 1 to m
        for j = 1 to n
                if x_i == y_j
                        c[i, j] = c[i - 1, j - 1] + 1
                        b[i, j] = "↖"
                elseif c[i - 1, j] ≥ c[i, j - 1]
                        c[i, j] = c[i - 1, j]
                        b[i, j] = "↑"
                else c[i, j] = c[i, j - 1]
                        b[i, j] = "←"
return c and b
```

in code x[i-1] == y[j-1]

Longest common subsequence (LCS)

Step 4: Constructing an Optimal Solution / Constructing an LCS

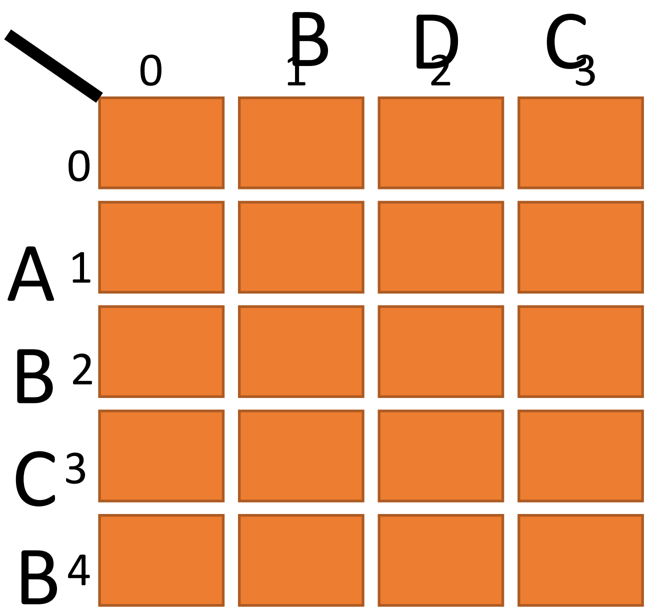
```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

O(m+n)

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Top-Down : Memoization)

Recursive code + Memoization code



X= ABCB
Y=BDC

Memo

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Top-Down : Memoization)

	0	B 1	D 2	C 3
0	-1	-1	-1	-1
A 1	-1	-1	-1	-1
B 2	-1	-1	-1	-1
C 3	-1	-1	-1	-1
B 4	-1	-1	-1	-1

X= ABCB
Y=BDC

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Top-Down : Memoization)

		B	D	C
	0	1	2	3
0	-1	-1	-1	-1
A 1	-1	-1	-1	-1
B 2	-1	-1	-1	-1
C 3	-1	-1	-1	-1
B 4	-1	-1	-1	-1

As it is a Top-Down approach, we will start from the top i.e. we will consider the bigger problem first.

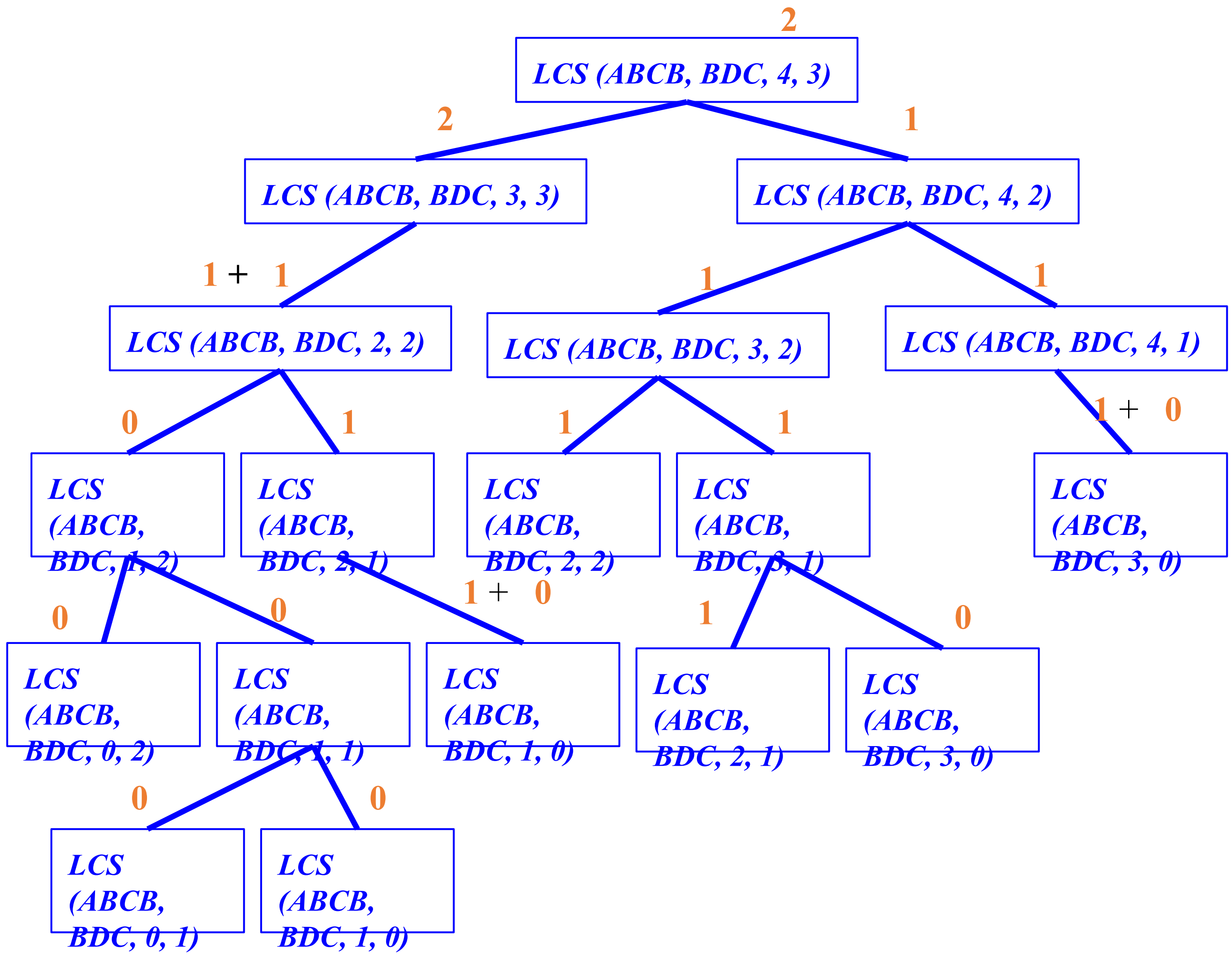
X= ABCB , Y=BDC

LCS (X, Y, 4, 3)

Longest common subsequence (LCS)

Step 3: Computing the length of an LCS (Top-Down : Memoization)

			B	D	C
		0	1	2	3
	0	-1	0	0	-1
A	1	0	0	0	-1
B	2	-1	1	1	-1
C	3	0	1	1	2
B	4	-1	1	1	2



Longest common subsequence (LCS)

Time Complexity of LCS:

m and n *are* ***length*** *of sequence* ***X and Y.***

Bottom-Up Approach : O(mn)

Top-Down Approach : O(mn)

Brute Force Approach : Exponential time

Longest common subsequence (LCS)

Exercises:

15.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

15.4-3

Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.


Thank You