

# **CSE-203: Data Structures & Algorithms I**

## **Topic: Graph Searching**

Nafiz Imtiaz Khan  
Lecturer, Department of CSE, MIST  
Email: nafiz@cse.mist.ac.bd

# Graph Searching

- Given: a graph  $G = (V, E)$ , directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected
- There are two standard graph traversal techniques:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

# Breadth-First Search

- “Explore” a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find (“discover”) its children, then their children, etc.

# Breadth-First Search

- Again will associate vertex “colors” to guide the algorithm
  - **White vertices** have not been discovered
    - All vertices start out white
  - **Grey vertices** are discovered but not fully explored
    - They may be adjacent to white vertices
  - **Black vertices** are discovered and fully explored
    - They are adjacent only to black and grey vertices
- Explore vertices by scanning **adjacency list** of grey vertices

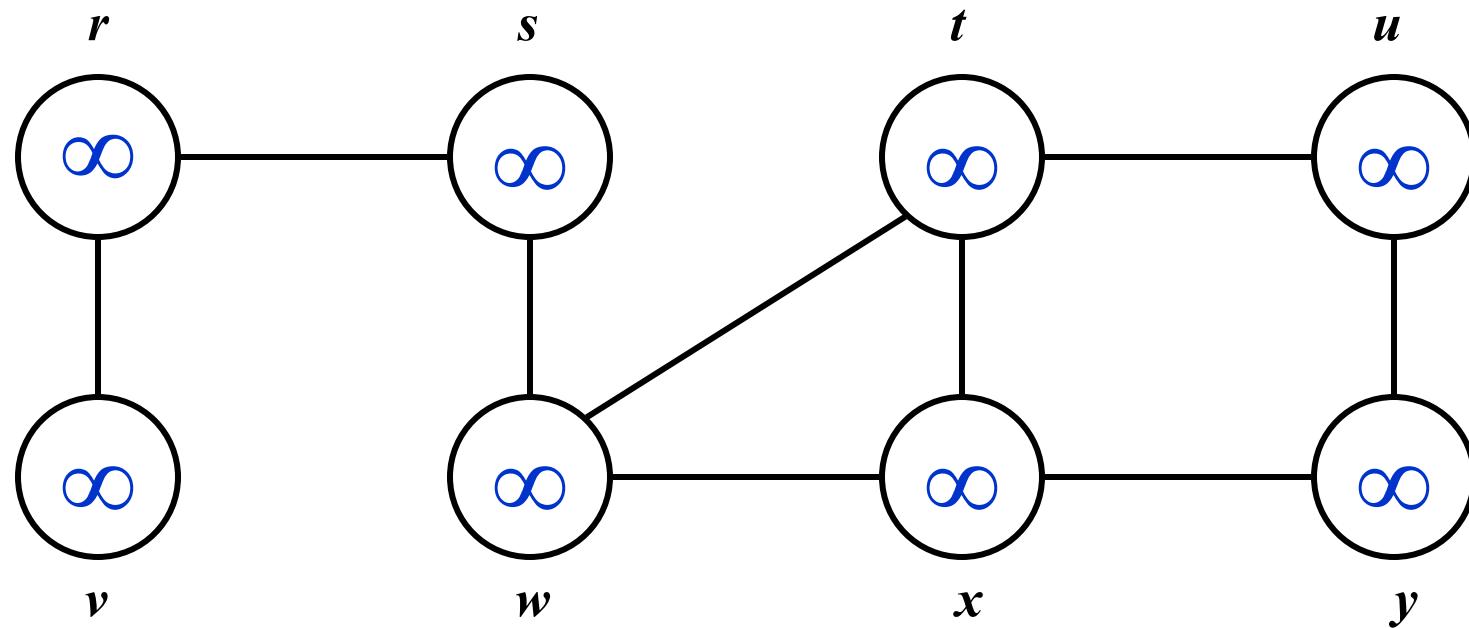
## BFS( $G, s$ )

```
1  for each vertex  $u \in V[G] - \{s\}$ 
2      do  $color[u] \leftarrow \text{WHITE}$ 
3           $d[u] \leftarrow \infty$ 
4           $\pi[u] \leftarrow \text{NIL}$ 
5   $color[s] \leftarrow \text{GRAY}$ 
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow \text{NIL}$ 
8   $Q \leftarrow \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11     do  $u \leftarrow \text{DEQUEUE}(Q)$ 
12         for each  $v \in Adj[u]$ 
13             do if  $color[v] = \text{WHITE}$ 
14                 then  $color[v] \leftarrow \text{GRAY}$ 
15                      $d[v] \leftarrow d[u] + 1$ 
16                      $\pi[v] \leftarrow u$ 
17                     ENQUEUE( $Q, v$ )
18      $color[u] \leftarrow \text{BLACK}$ 
```

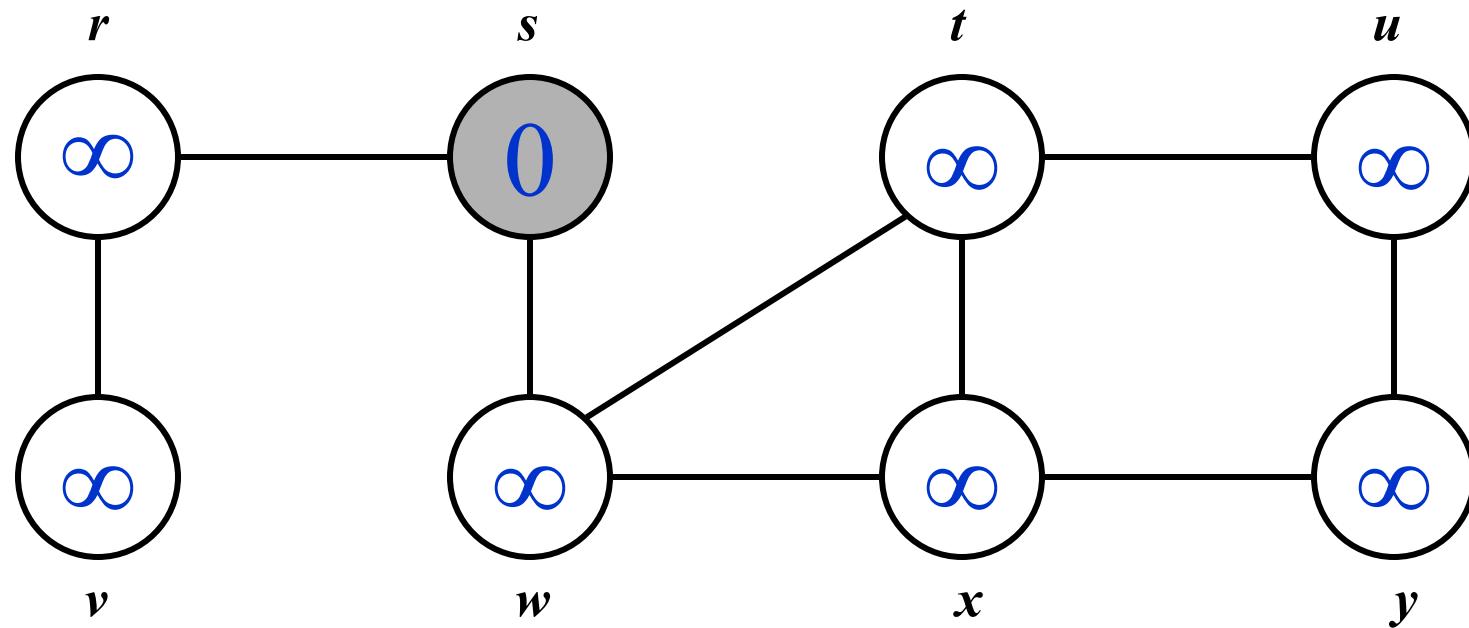
# Breadth-First Search - Pseudo-Code

1. Declare and initialize the necessary variables and data structures:
  - An array to store vertex values.
  - An array to track the color of each vertex during BFS.
  - A 2D array representing the adjacency matrix of the graph.
  - Initialize all elements of the arrays.
2. Define the graph connections:
  - Set specific edges in  $g[x][y]$  to 1, indicating the presence of connections between  $x$  and  $y$  vertices.
3. Initialize a queue ( $q$ ) and enqueue the starting vertex:
  - Create a queue ( $q$ ) to store vertices for BFS traversal.
  - Enqueue the starting vertex (source vertex) into the queue.
  - Set the color of the vertex to grey, indicating that vertex 1 is visited.
4. Perform BFS traversal:
  - While the queue is not empty, repeat the following steps:
    - Dequeue a vertex from the front of the queue and assign it to a variable.
    - Print the value of the dequeued element, indicating that the vertex has been visited.
    - If the color of a vertex (let's say  $k$ ) is grey (indicating the vertex is visited but not processed):
      - Find the adjacent vertices of the vertex  $k$ .
      - Enqueue the adjacent vertices of  $k$  into the queue.
      - Set the color of the adjacent vertices of  $k$  to grey.
      - As all the adjacent vertices of  $k$  are visited, set the color of  $k$  to black
5. Continue the BFS traversal until the queue becomes empty.
6. Terminate the program.

# Breadth-First Search: Example

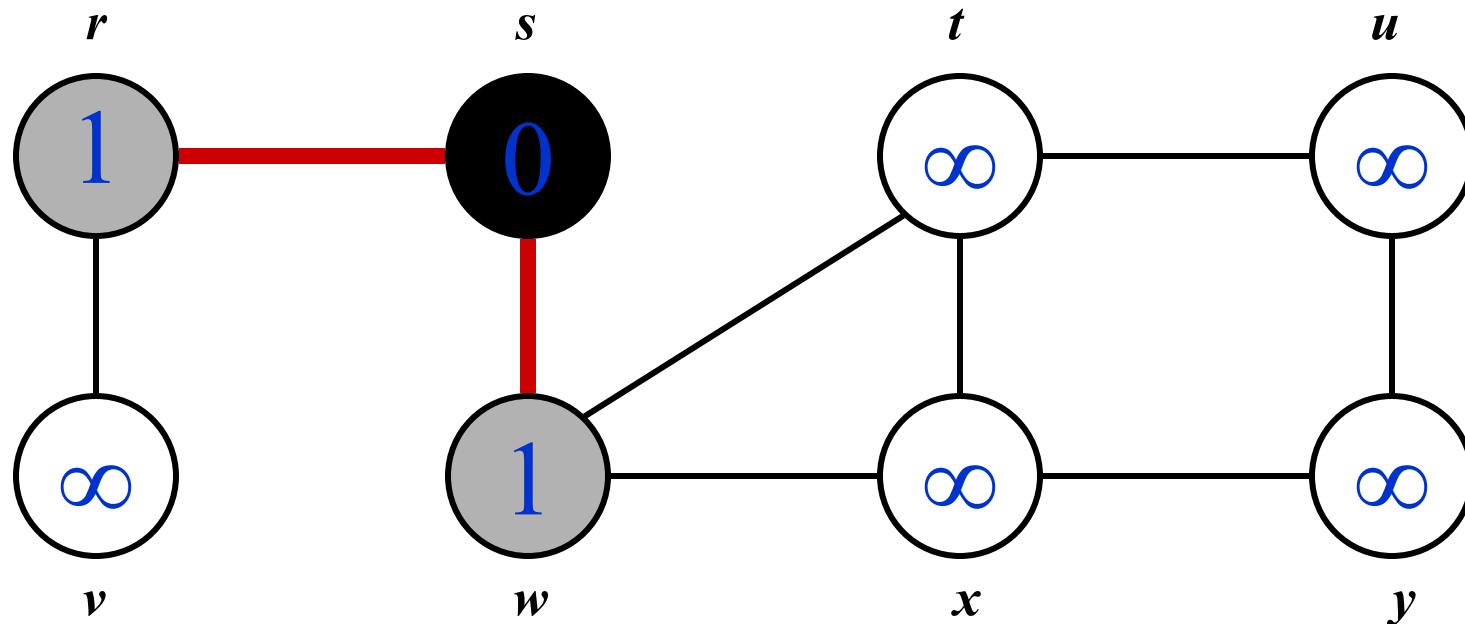


# Breadth-First Search: Example



$Q:$    $s$

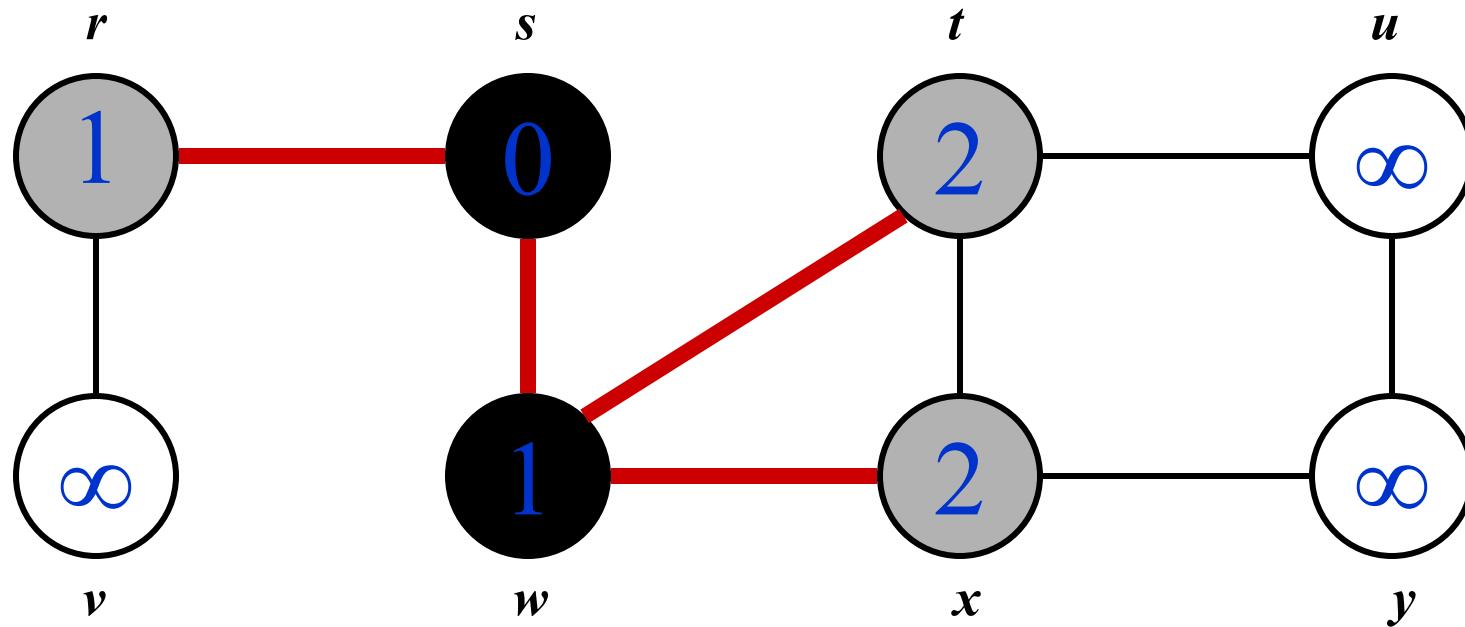
# Breadth-First Search: Example



$Q:$ 

$w$	$r$
-----	-----

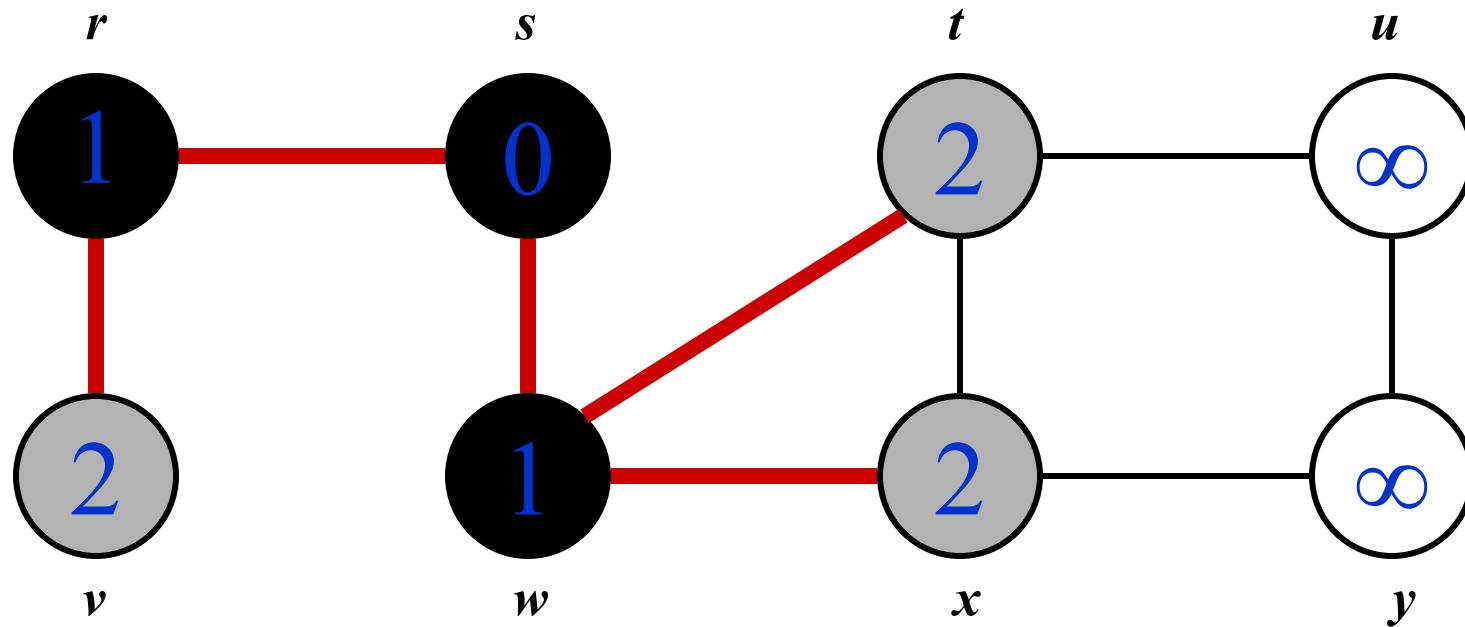
# Breadth-First Search: Example



$Q:$ 

$r$	$t$	$x$
-----	-----	-----

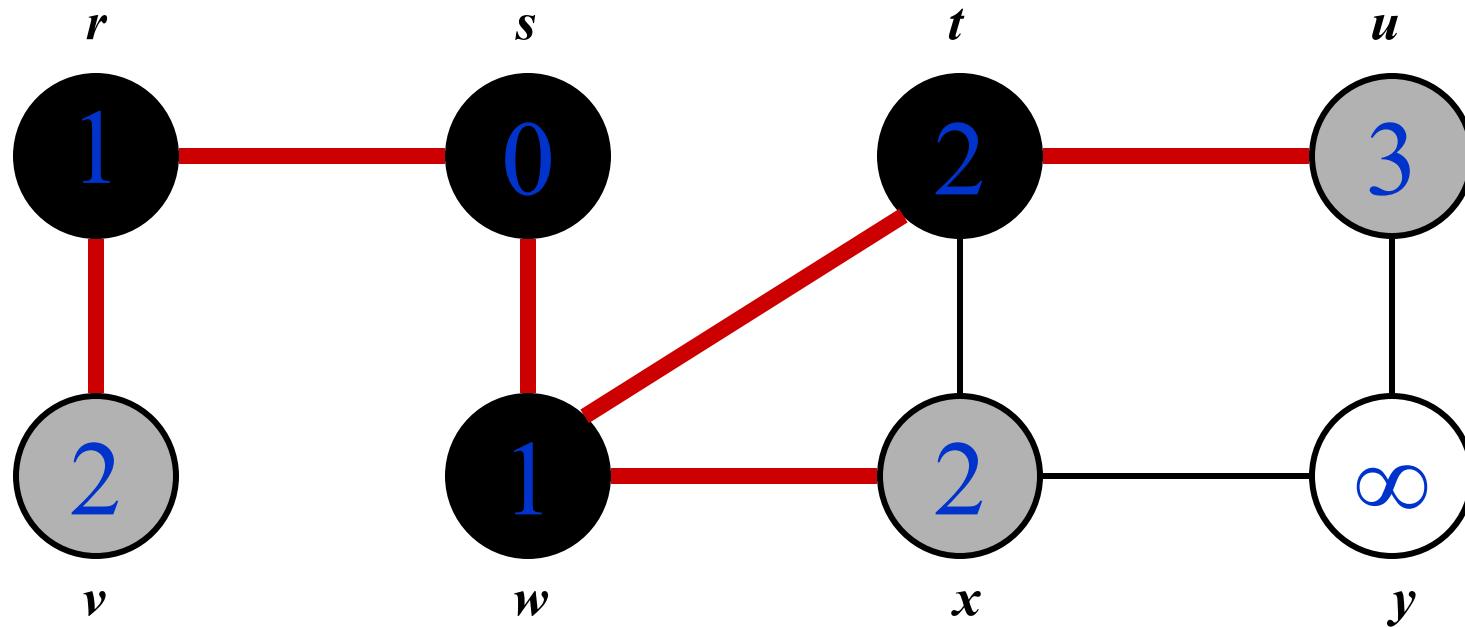
# Breadth-First Search: Example



$Q:$ 

$t$	$x$	$v$
-----	-----	-----

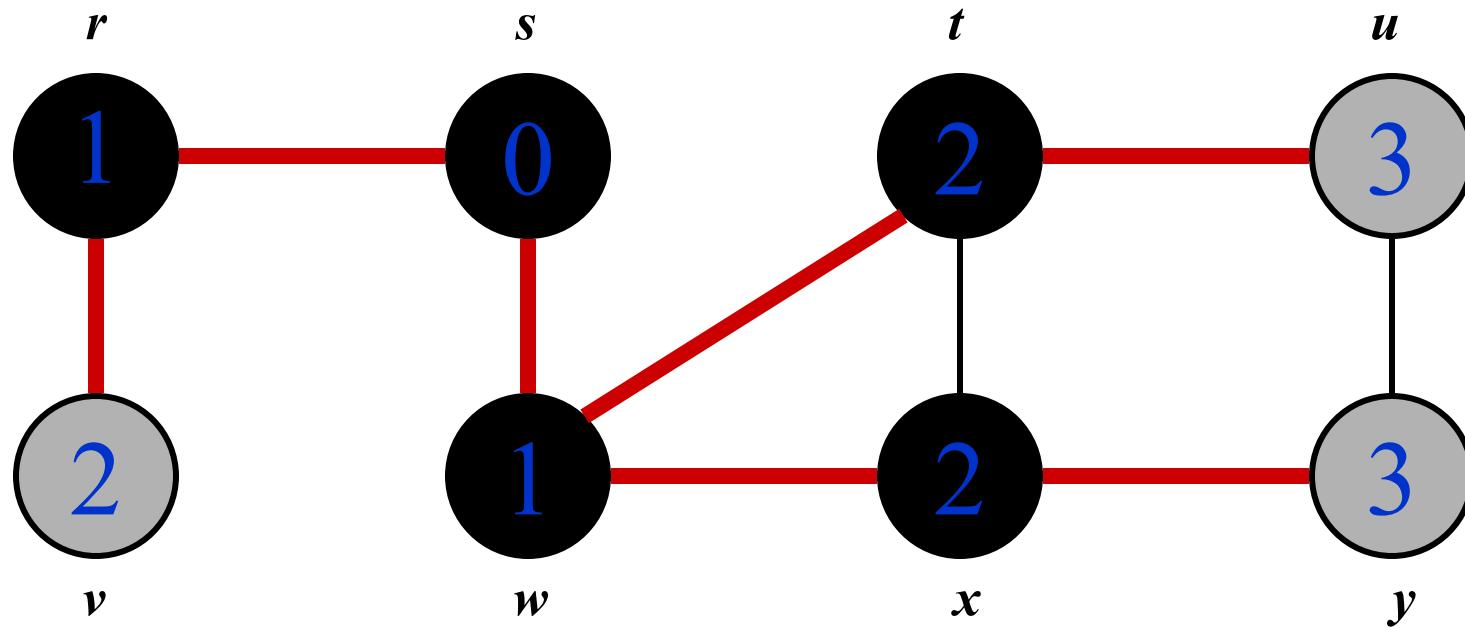
# Breadth-First Search: Example



$Q:$ 

$x$	$v$	$u$
-----	-----	-----

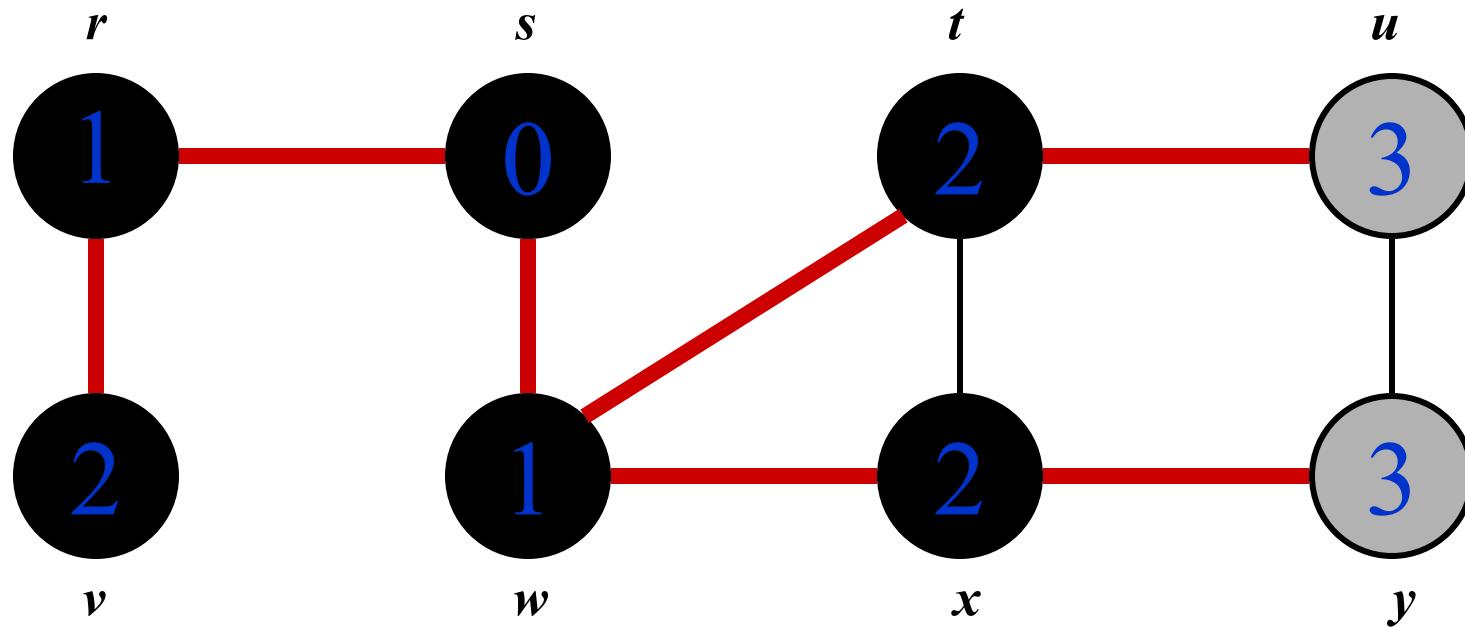
# Breadth-First Search: Example



$Q:$ 

$v$	$u$	$y$
-----	-----	-----

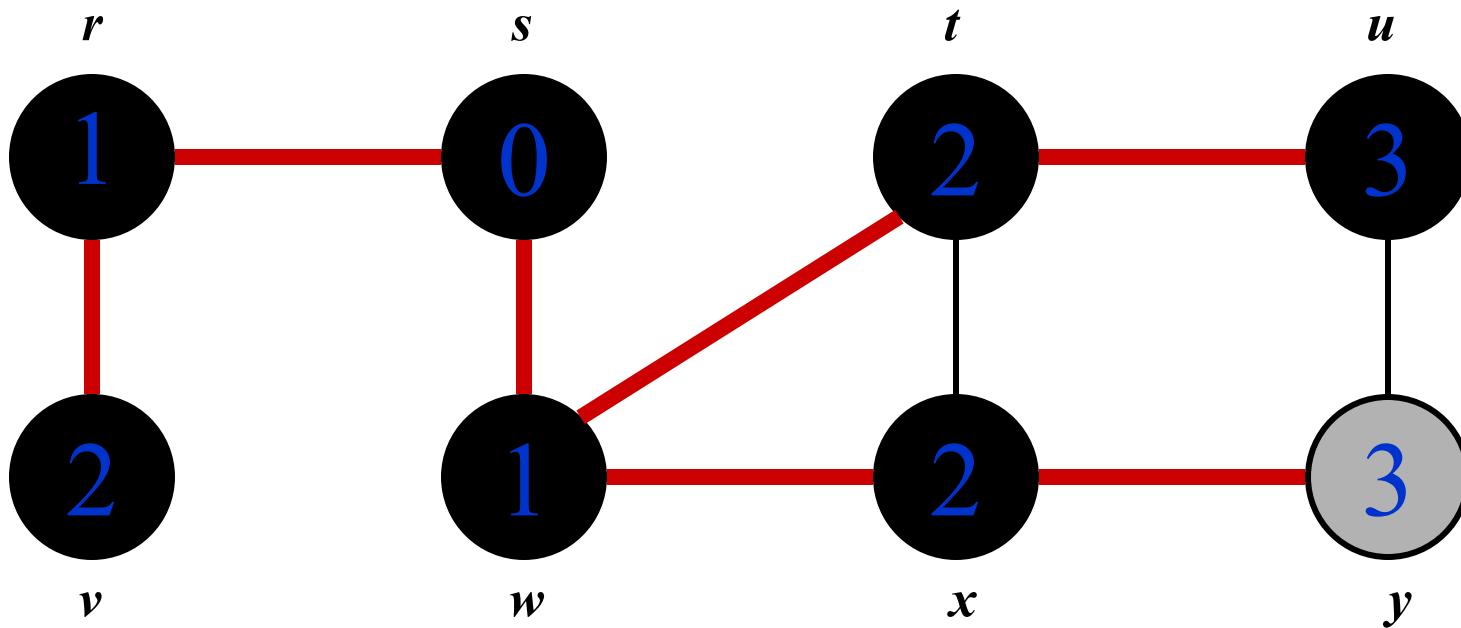
# Breadth-First Search: Example



$Q:$ 

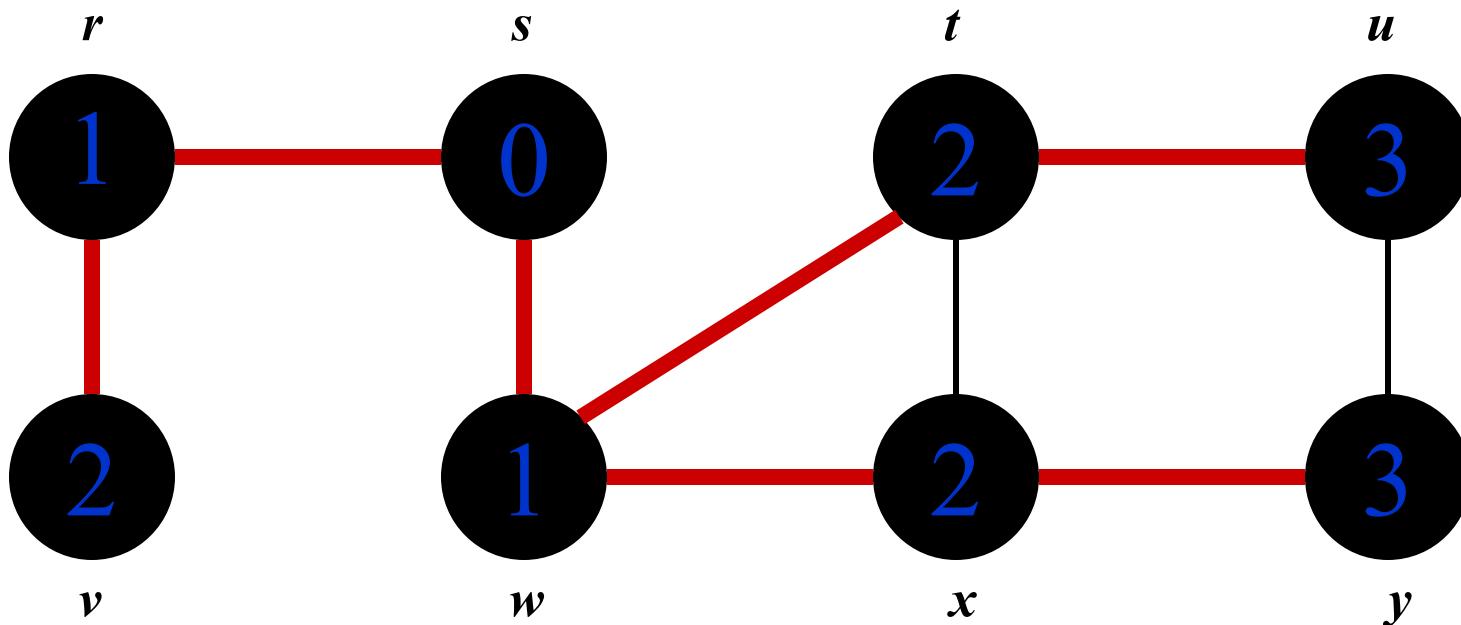
$u$	$y$
-----	-----

# Breadth-First Search: Example



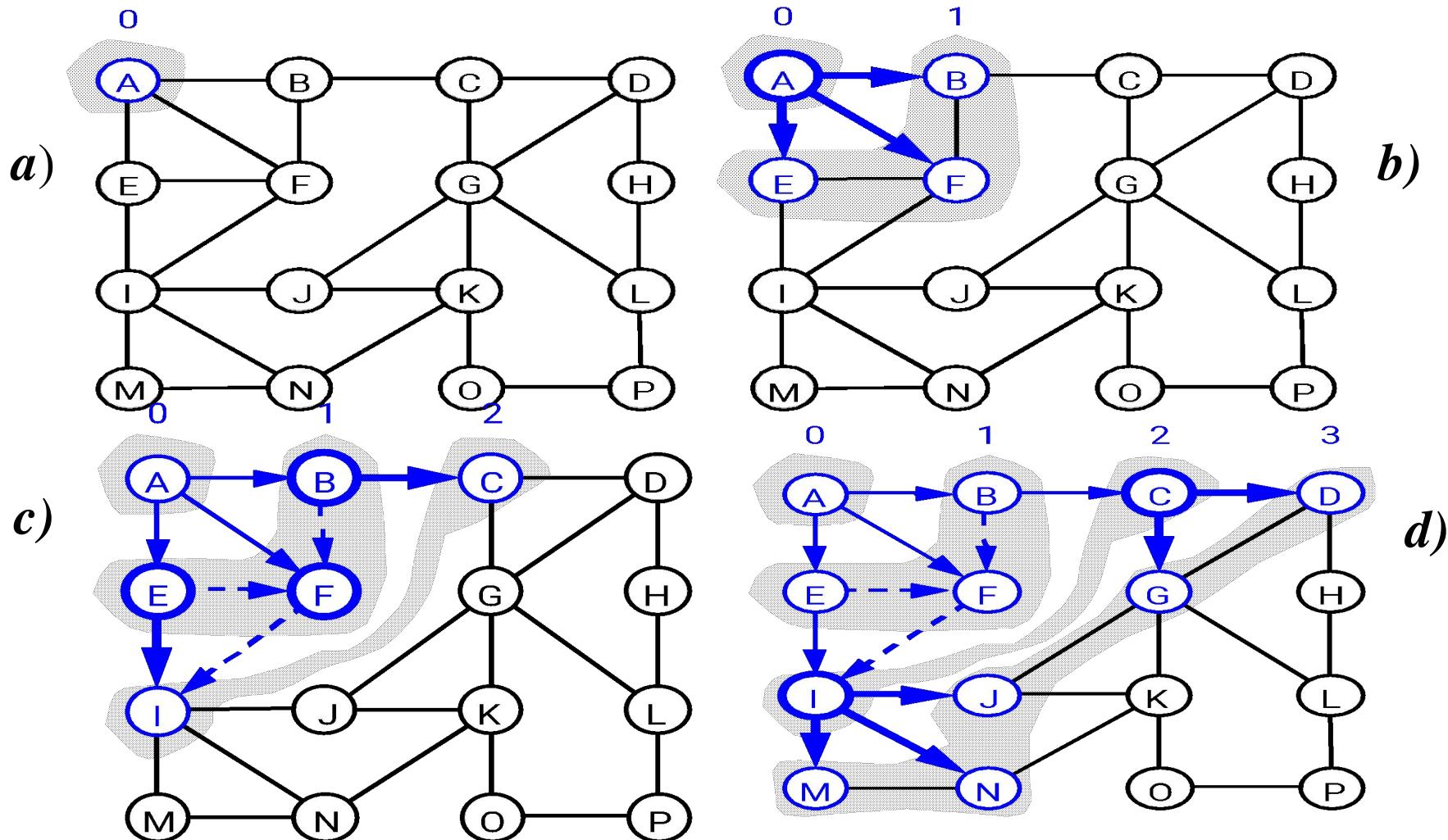
$Q:$    $y$

# Breadth-First Search: Example

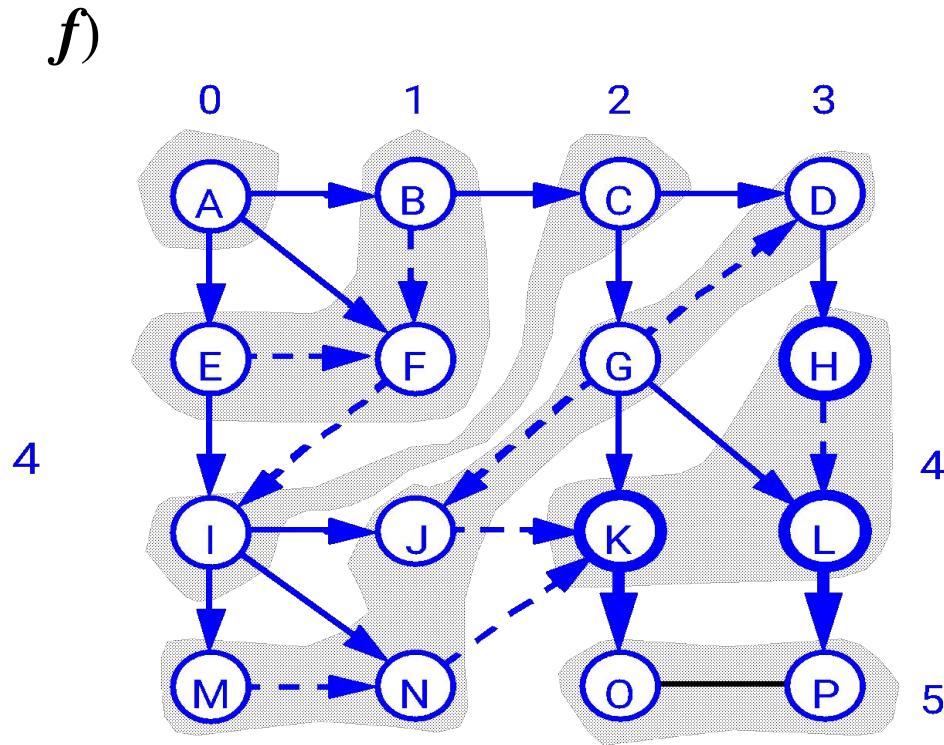
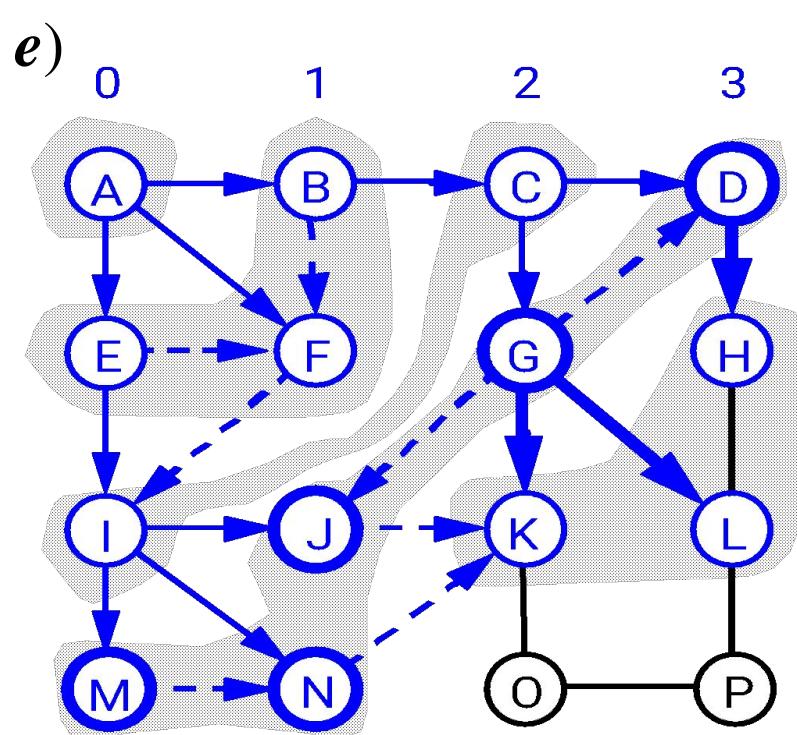


$Q:$   $\emptyset$

# BFS - A Graphical Representation



# BFS - A Graphical Representation



# BFS: The Code Again

```
BFS(G, s) {  
    initialize vertices;           ← Touch every vertex:  $O(V)$   
    Q = {s};  
    while (Q not empty) {  
        u = RemoveTop(Q);         ← u = every vertex, but only once  
        for each v ∈ u->adj {  
            if (v->color == WHITE)  
                v->color = GREY;  
                v->d = u->d + 1;  
                v->p = u;  
                Enqueue(Q, v);  
        }  
        u->color = BLACK;  
    }  
}
```

*So v = every vertex that appears in some other vert's adjacency list*

*(Why?)*

*What will be the running time?*

Total running time:  $O(V+E)$

# BFS: The Code Again

```
BFS(G, s) {  
    initialize vertices;  
    Q = {s};  
    while (Q not empty) {  
        u = RemoveTop(Q);  
        for each v ∈ u->adj {  
            if (v->color == WHITE)  
                v->color = GREY;  
                v->d = u->d + 1;  
                v->p = u;  
                Enqueue(Q, v);  
        }  
        u->color = BLACK;  
    }  
}
```

*What will be the storage cost  
in addition to storing the tree?*

Total space used:  
 $O(V + \Sigma(\text{degree}(v))) = O(V + E)$

# Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance  $\delta(s,v)$  = minimum number of edges from  $s$  to  $v$ , or  $\infty$  if  $v$  not reachable from  $s$
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in  $G$ 
  - Thus can use BFS to calculate shortest path from one vertex to another in  $O(V+E)$  time in an unweighted tree

# Depth-First Search

- *Depth-first search* is another strategy for exploring a graph
  - Explore “deeper” in the graph whenever possible
  - Edges are explored out of the most recently discovered vertex  $v$  that still has unexplored edges
  - When all of  $v$ ’s edges have been explored, backtrack to the vertex from which  $v$  was discovered
- Vertices initially colored white
- Then colored grey when discovered
- Then black when finished

# Depth-First Search: The Code

```
DFS (G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj []
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

# Depth-First Search: The Code

```
DFS (G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj []
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

*What does u->d represent?*

# Depth-First Search: The Code

```
DFS (G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj []
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

*What does u->f represent?*

# Depth-First Search: The Code

```
DFS (G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj []
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

*Will all vertices eventually be colored black?*

# Depth-First Search: The Code

```
DFS (G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj []
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

*What will be the running time?*

# Depth-First Search: The Code

```
DFS (G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj []
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

*Running time:  $O(n^2)$  because call  
DFS\_Visit on each vertex,  
and the loop over Adj[] can run as  
many as  $|V|$  times*

# Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

*BUT, there is actually a tighter bound.  
How many times will DFS\_Visit()  
actually be called?*

# Depth-First Search: The Code

```
DFS (G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj []
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

*So, running time of DFS =  $O(V+E)$*

# Depth-First Search Analysis

function DFS(graph, start\_vertex):

    create an array to keep track of the visited vertices

    Initialize the vertices with white

    call DFS-Visit(source vertex)

function DFS-Visit(vertex):

    print vertex

    mark vertex as visited/change the color to grey

    for each neighbor in graph[vertex]:

        if neighbor is not visited:

            call DFS-Visit(neighbour))

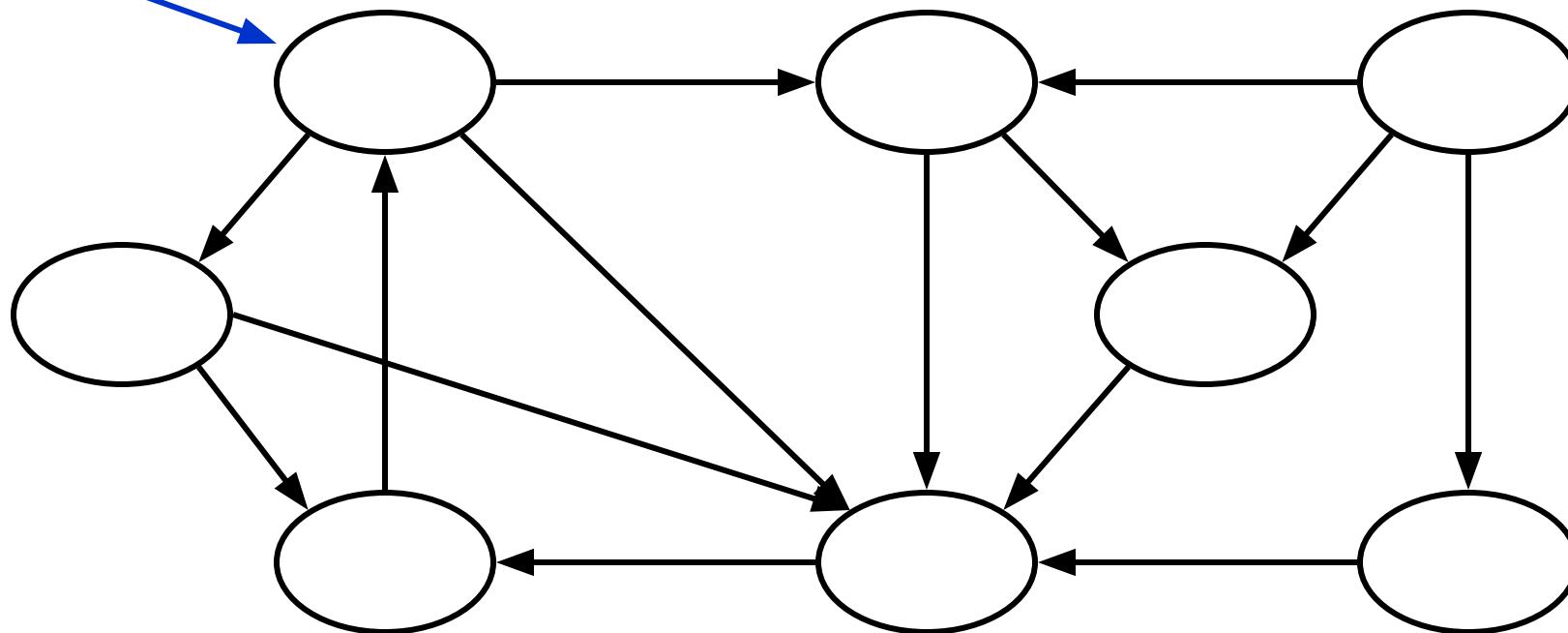
        change the color of the vertex to black

# Depth-First Search Analysis

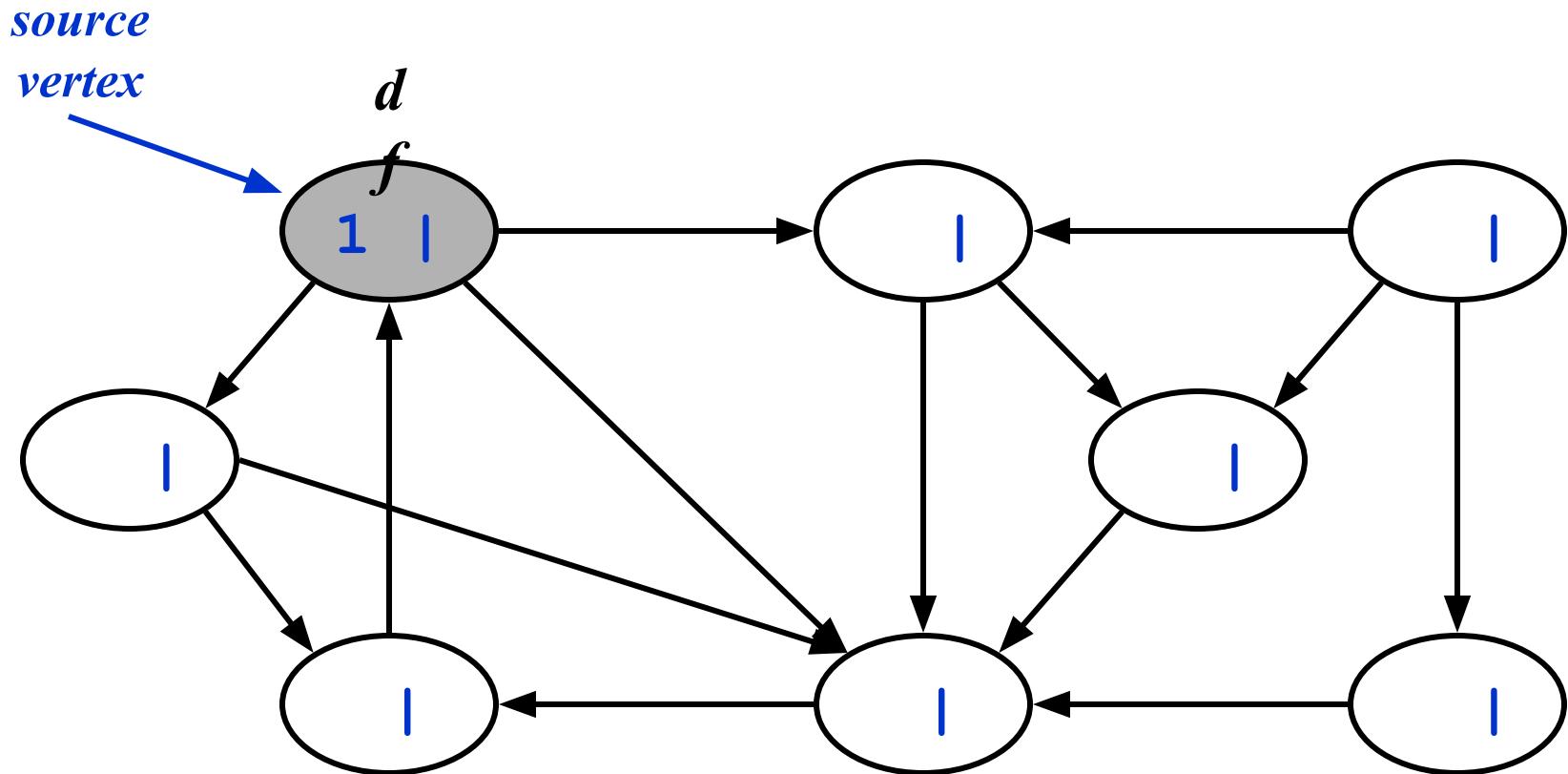
- This running time argument is an informal example of *amortized analysis*
  - “Charge” the exploration of edge to the edge:
    - Each loop in DFS\_Visit can be attributed to an edge in the graph
    - Runs once/edge if directed graph, twice if undirected
    - Thus loop will run in  $O(E)$  time, algorithm  $O(V+E)$
  - Storage requirement is  $O(V+E)$ , since adj list requires  $O(V+E)$  storage

# DFS Example

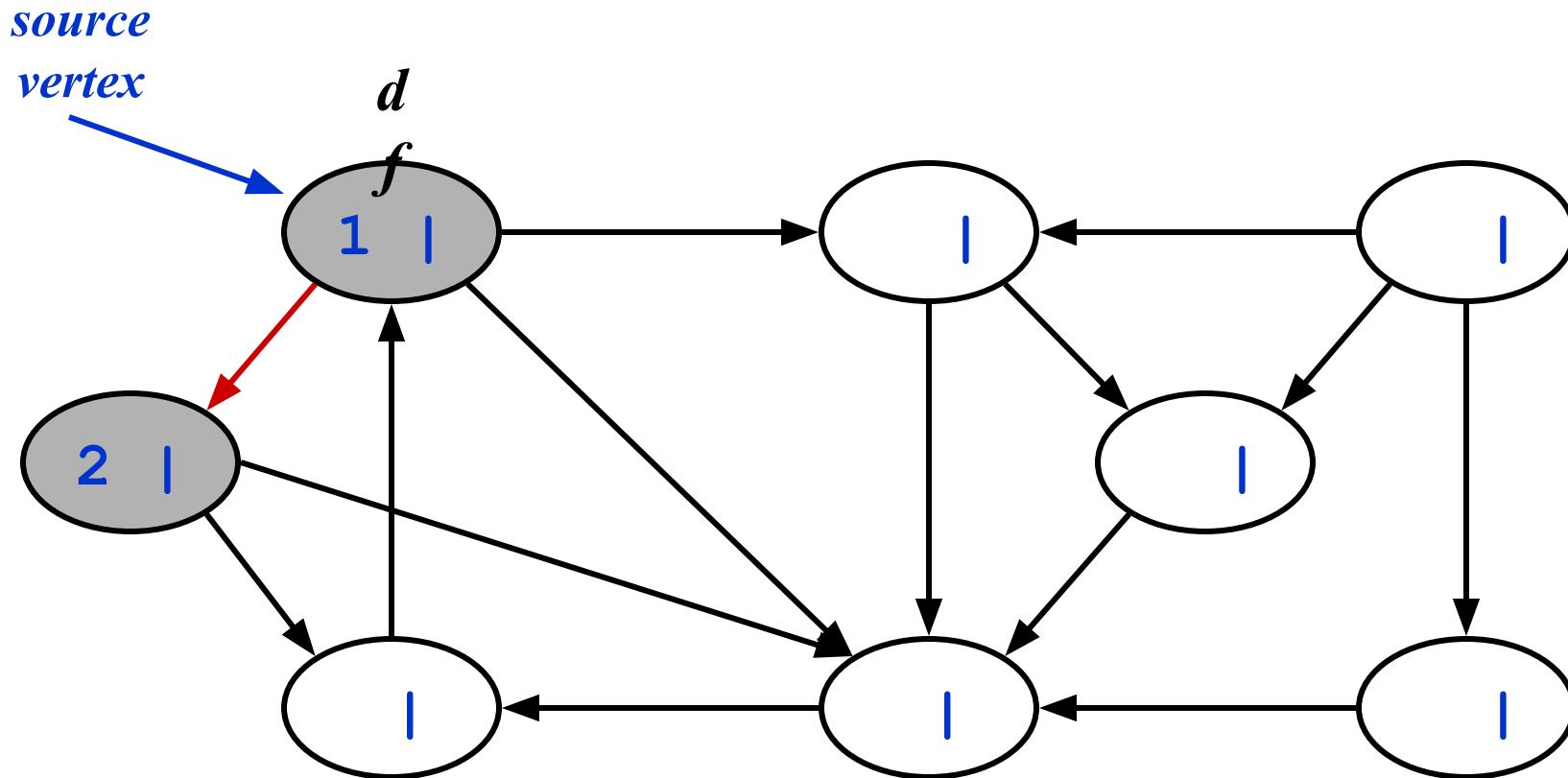
*source  
vertex*



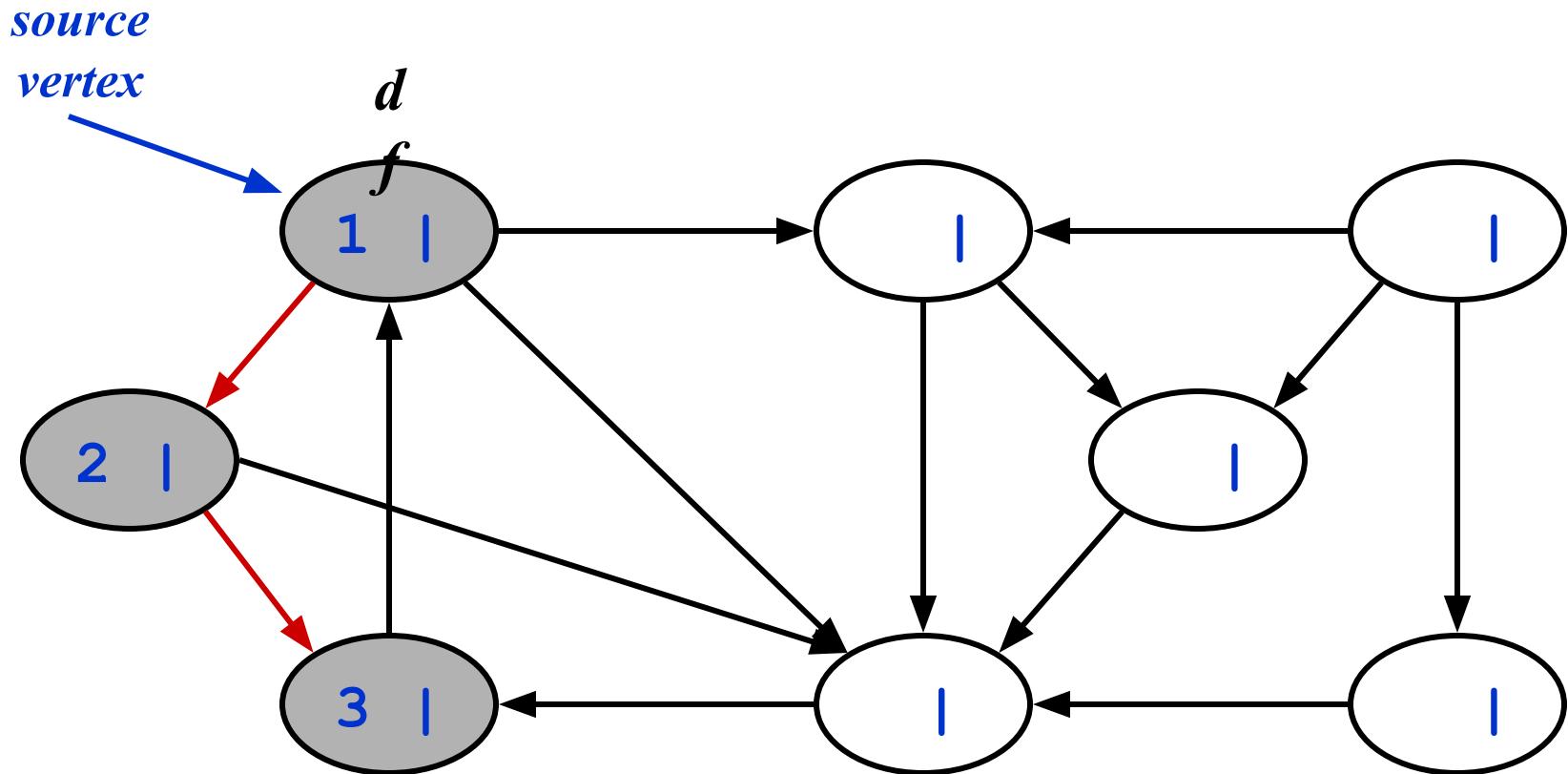
# DFS Example



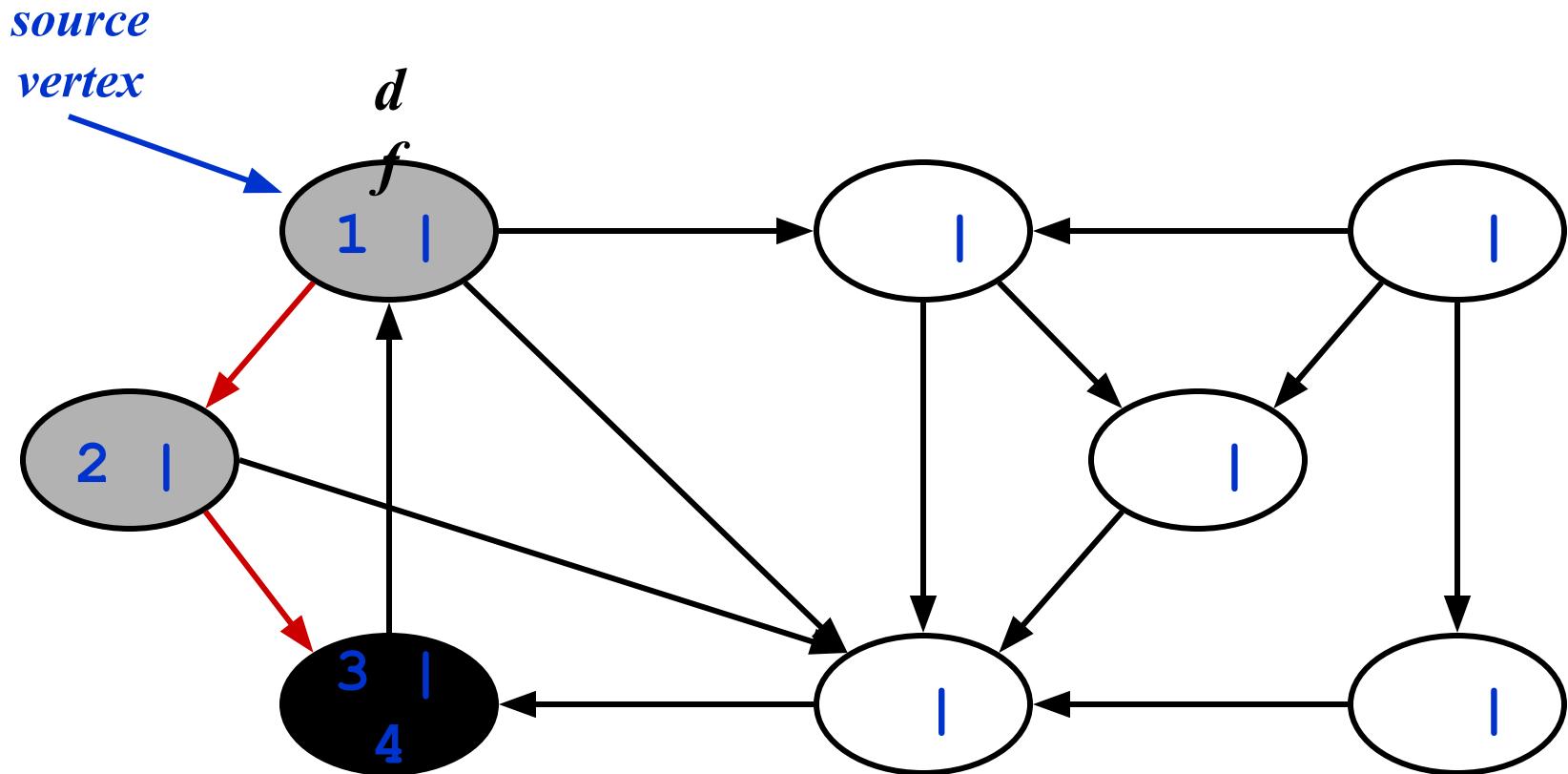
# DFS Example



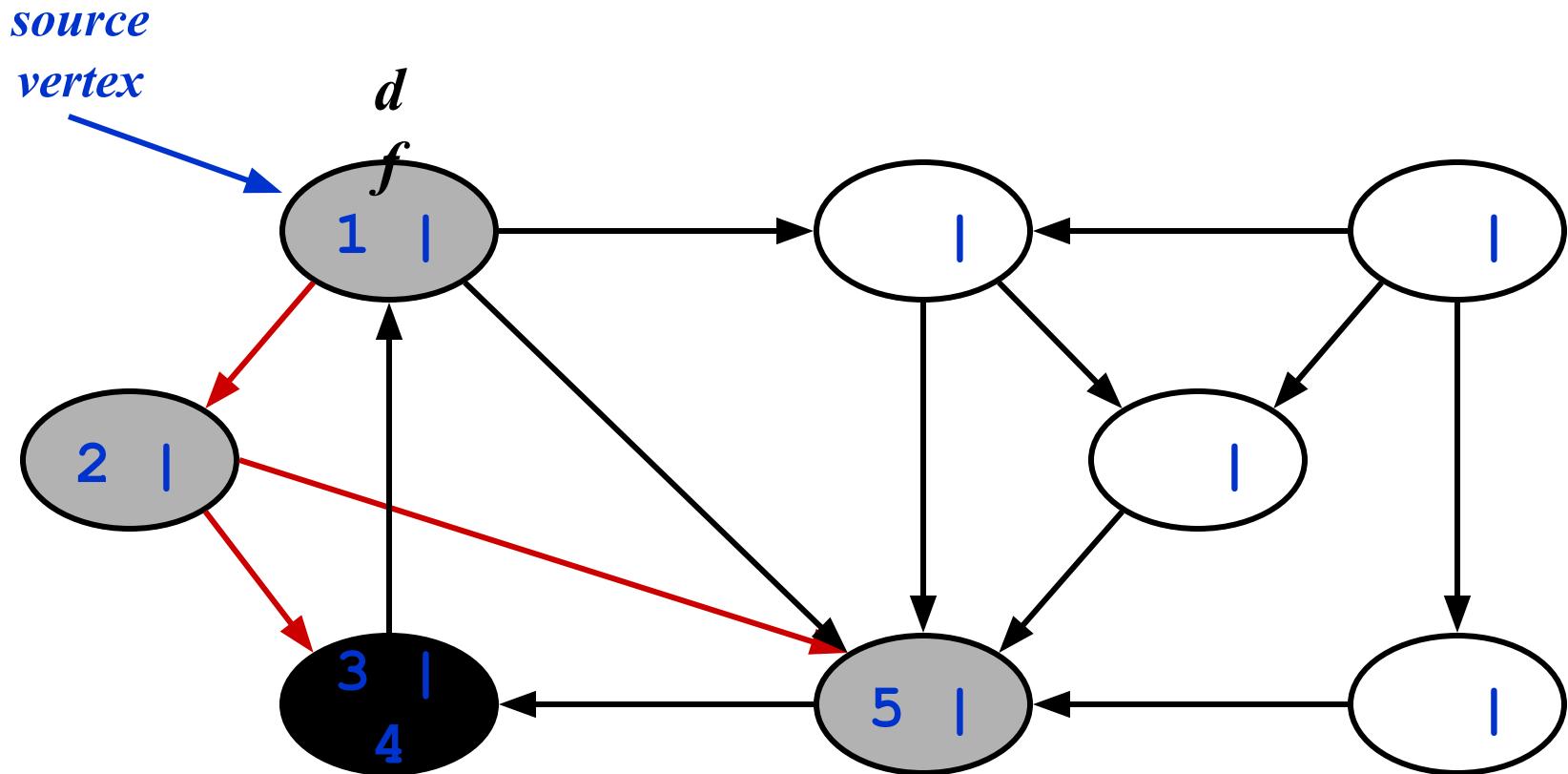
# DFS Example



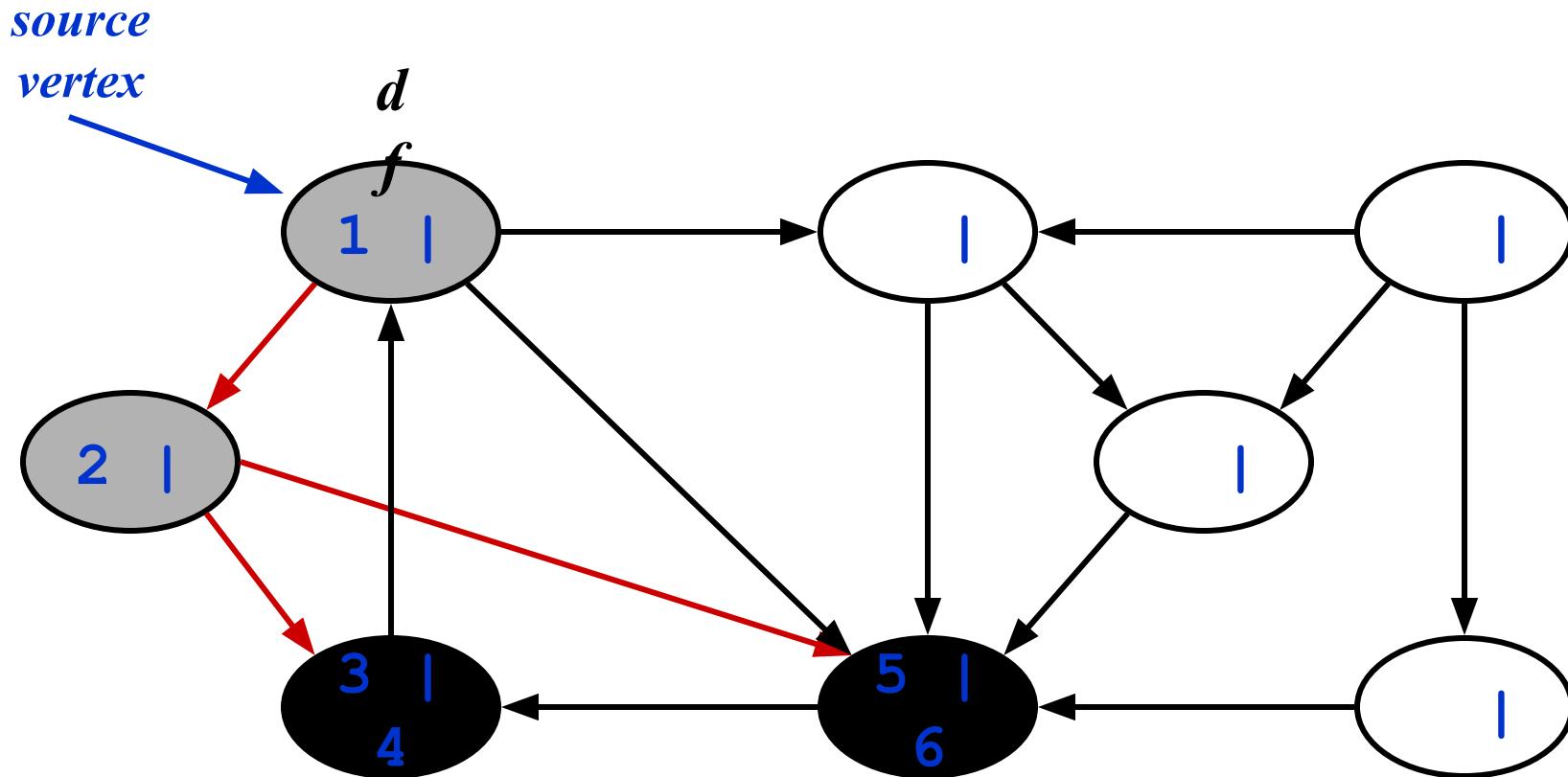
# DFS Example



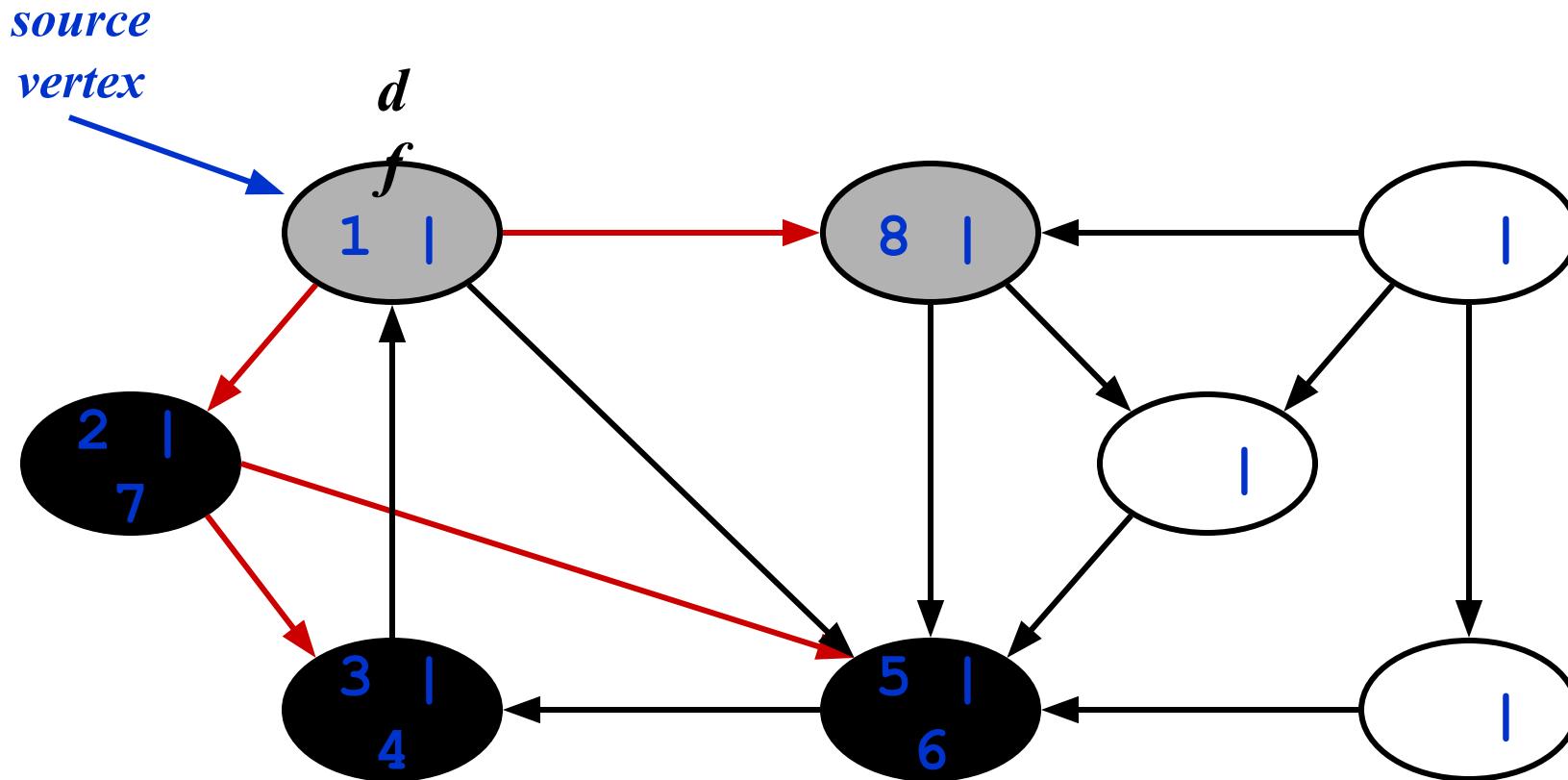
# DFS Example



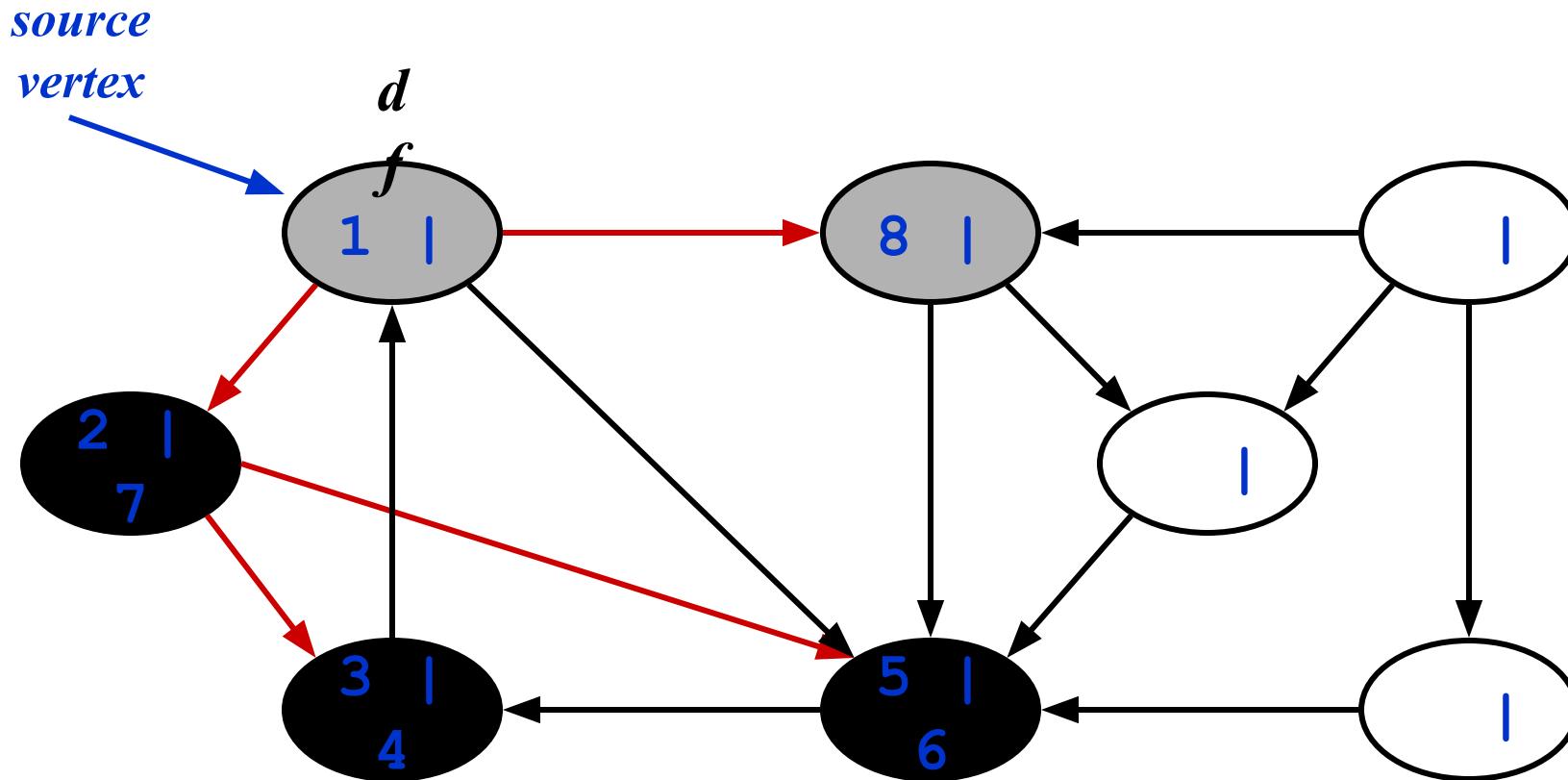
# DFS Example



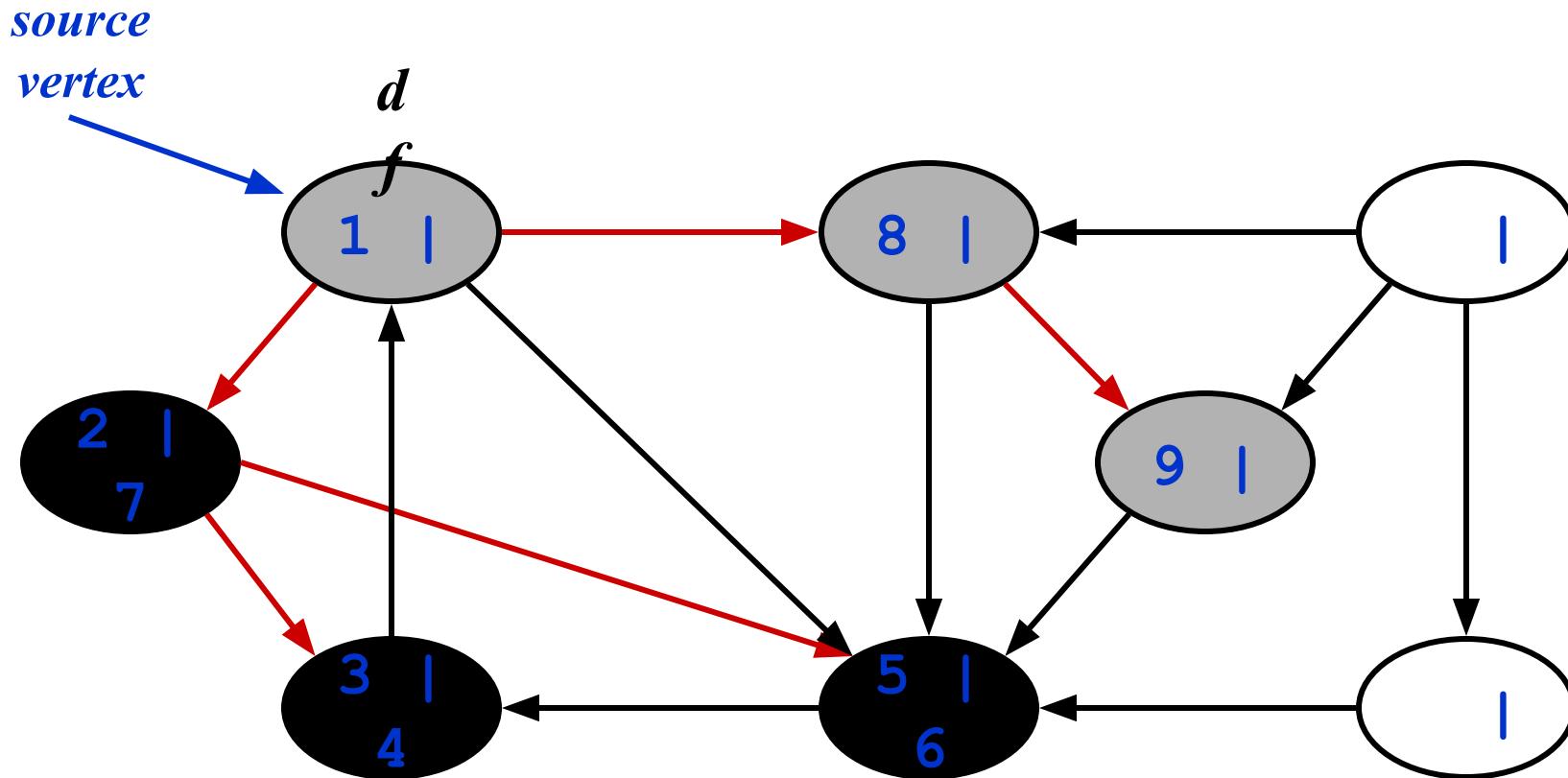
# DFS Example



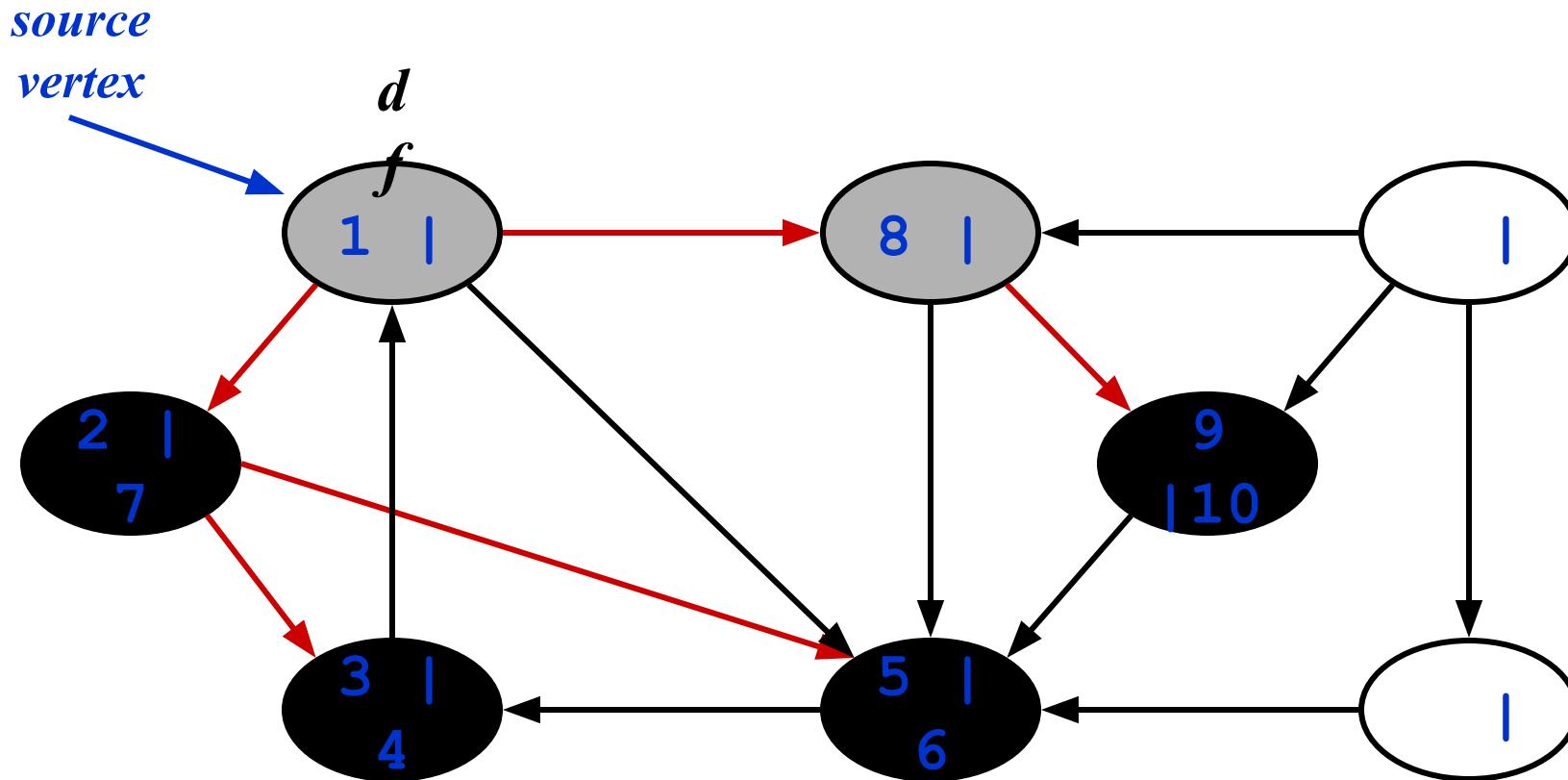
# DFS Example



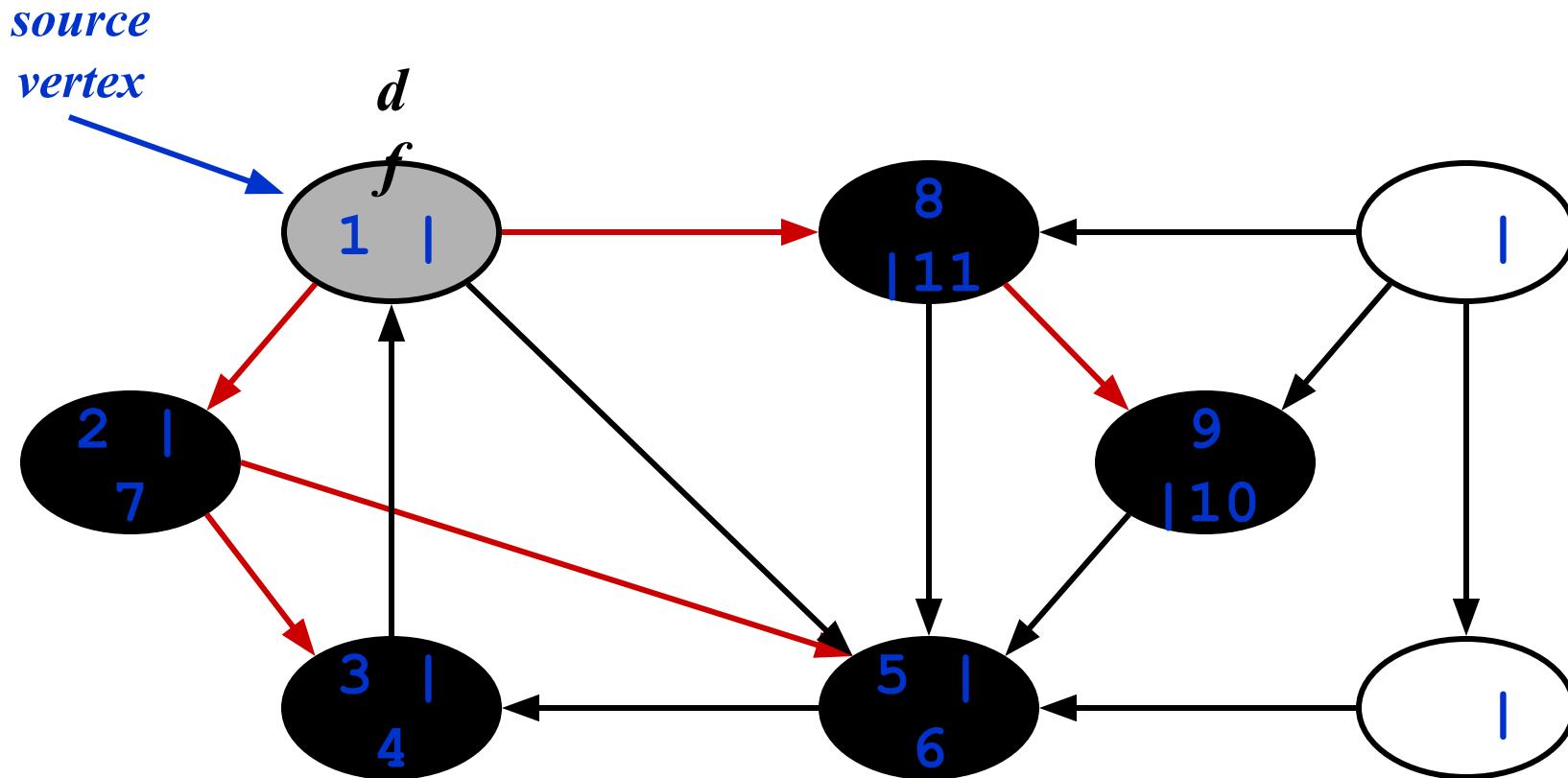
# DFS Example



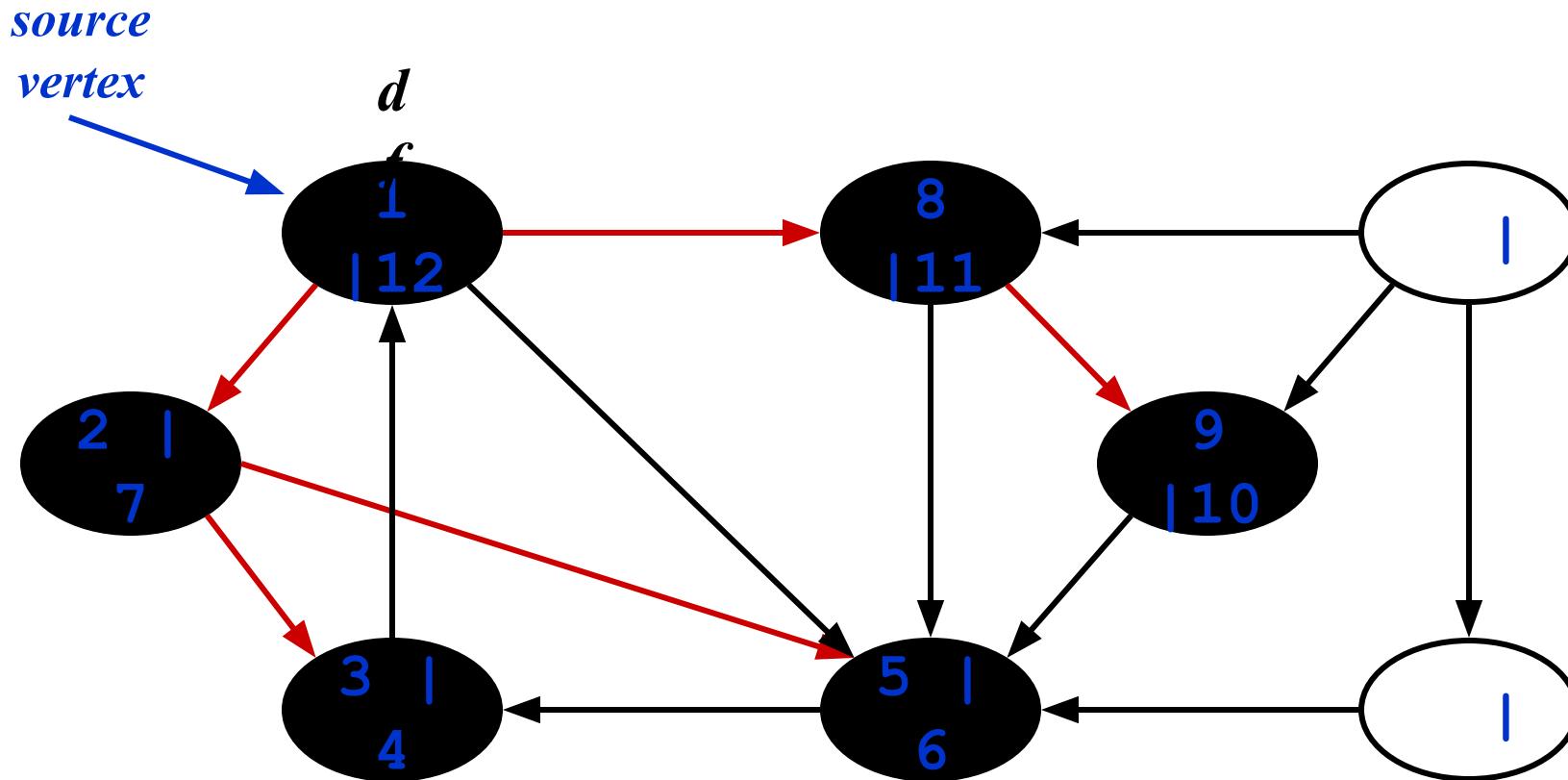
# DFS Example



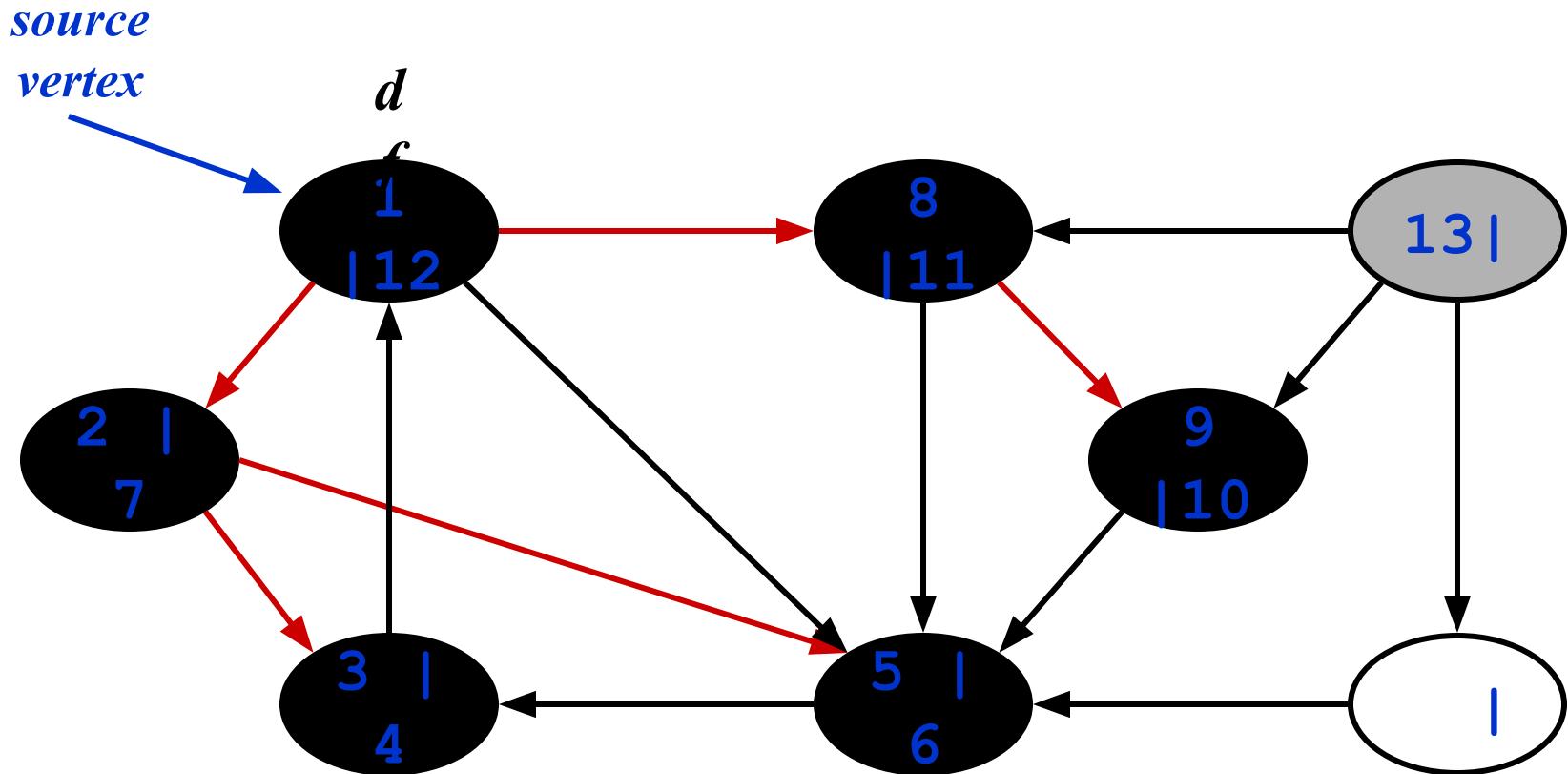
# DFS Example



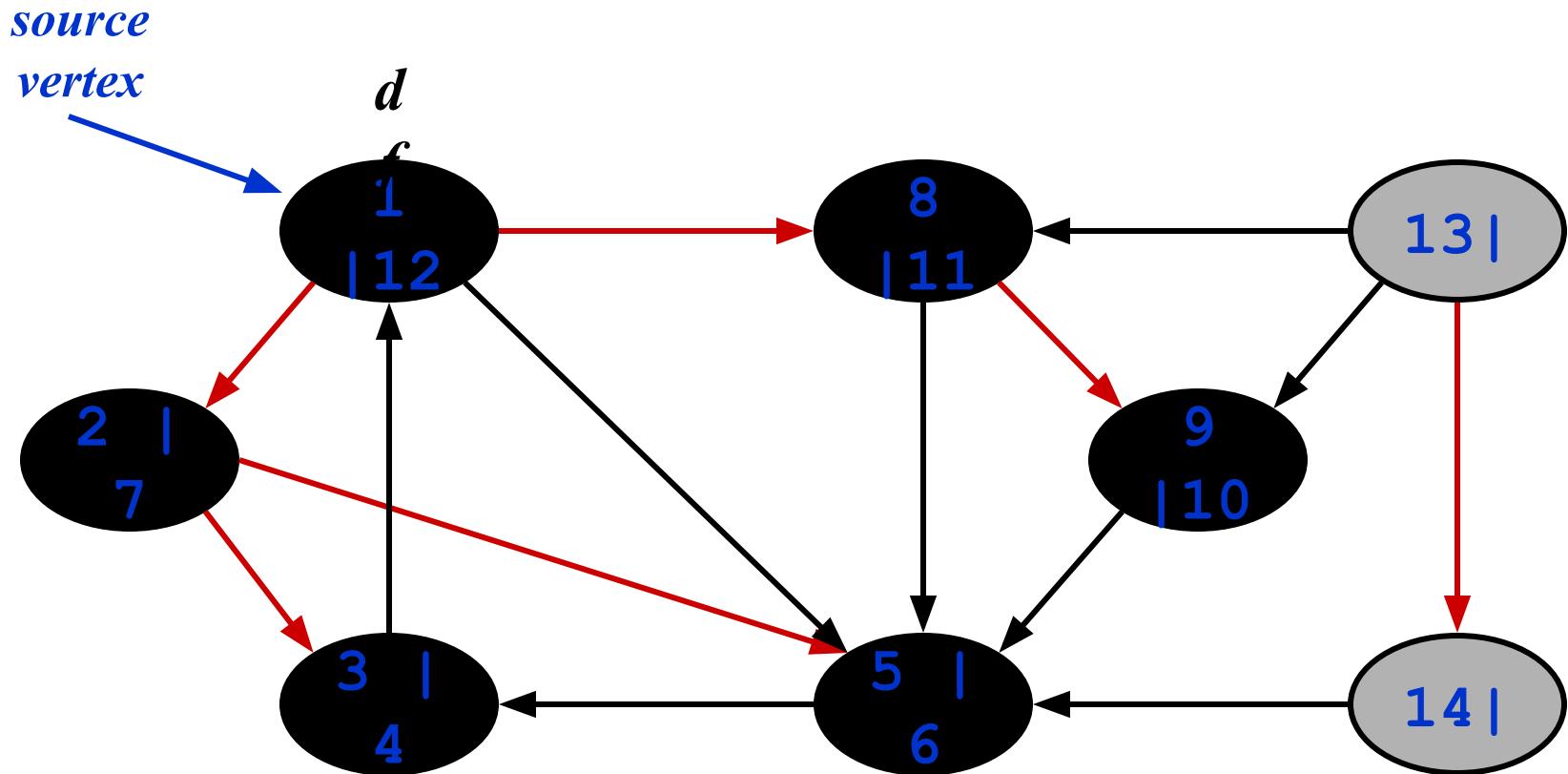
# DFS Example



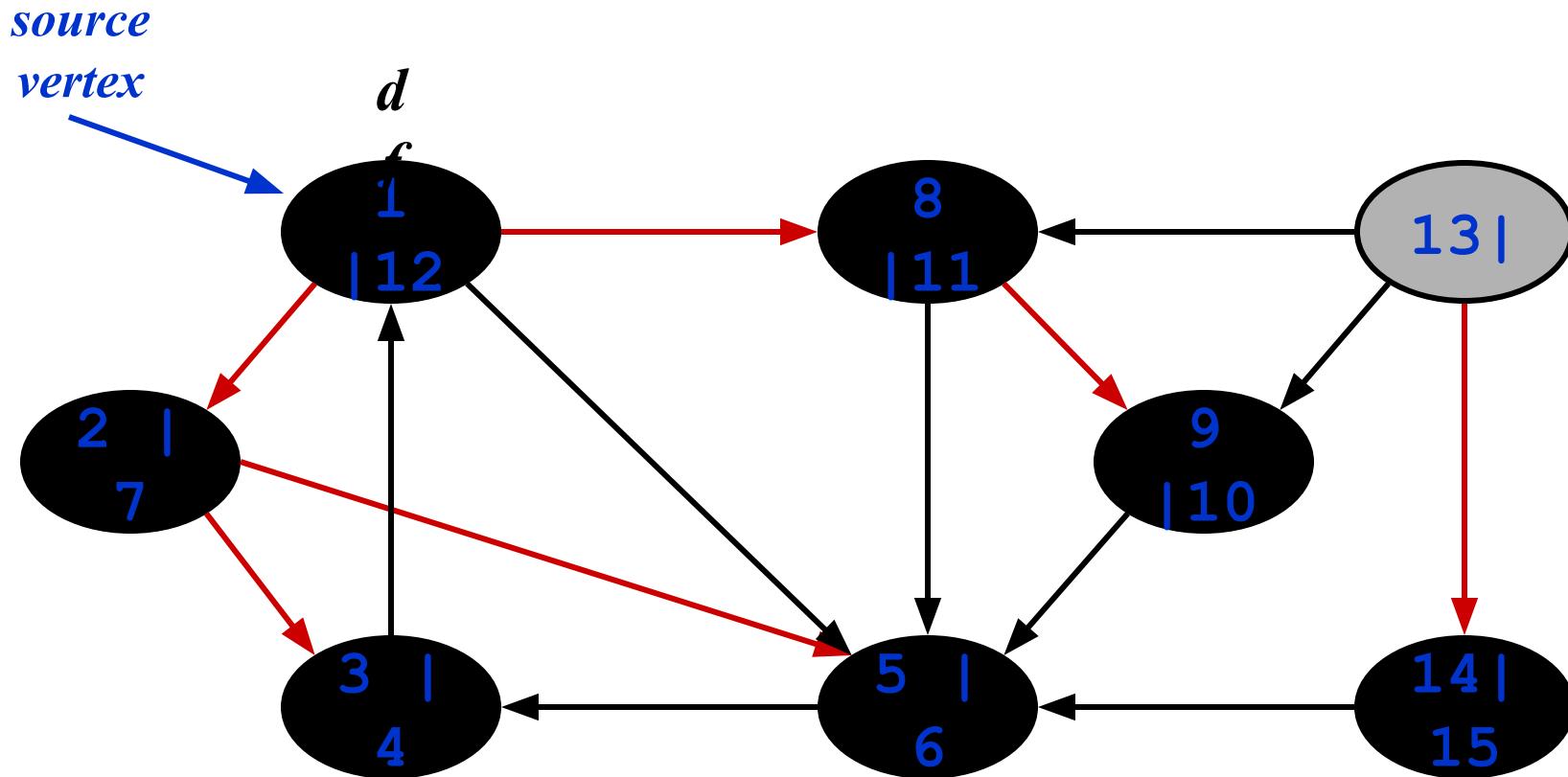
# DFS Example



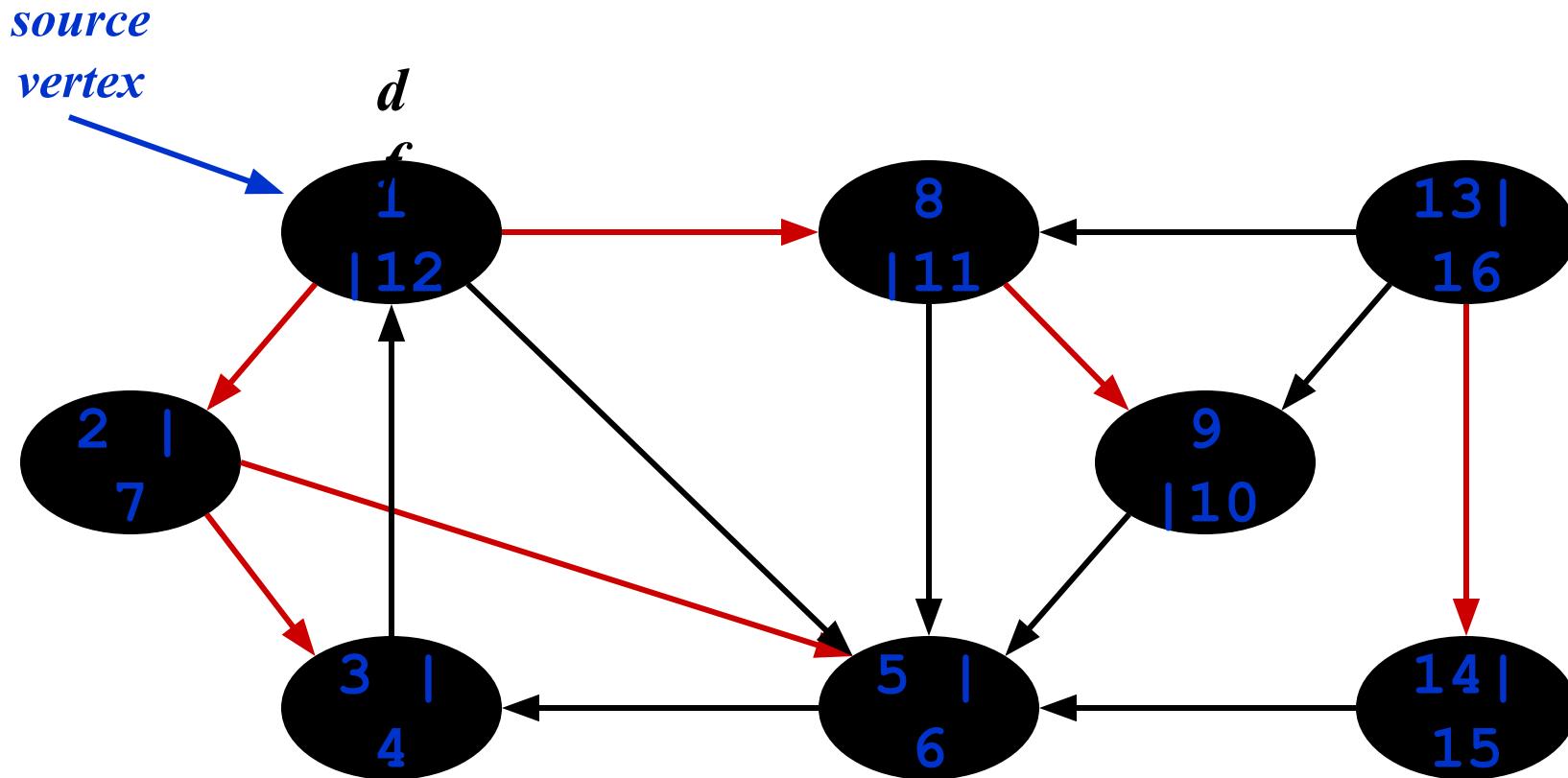
# DFS Example



# DFS Example



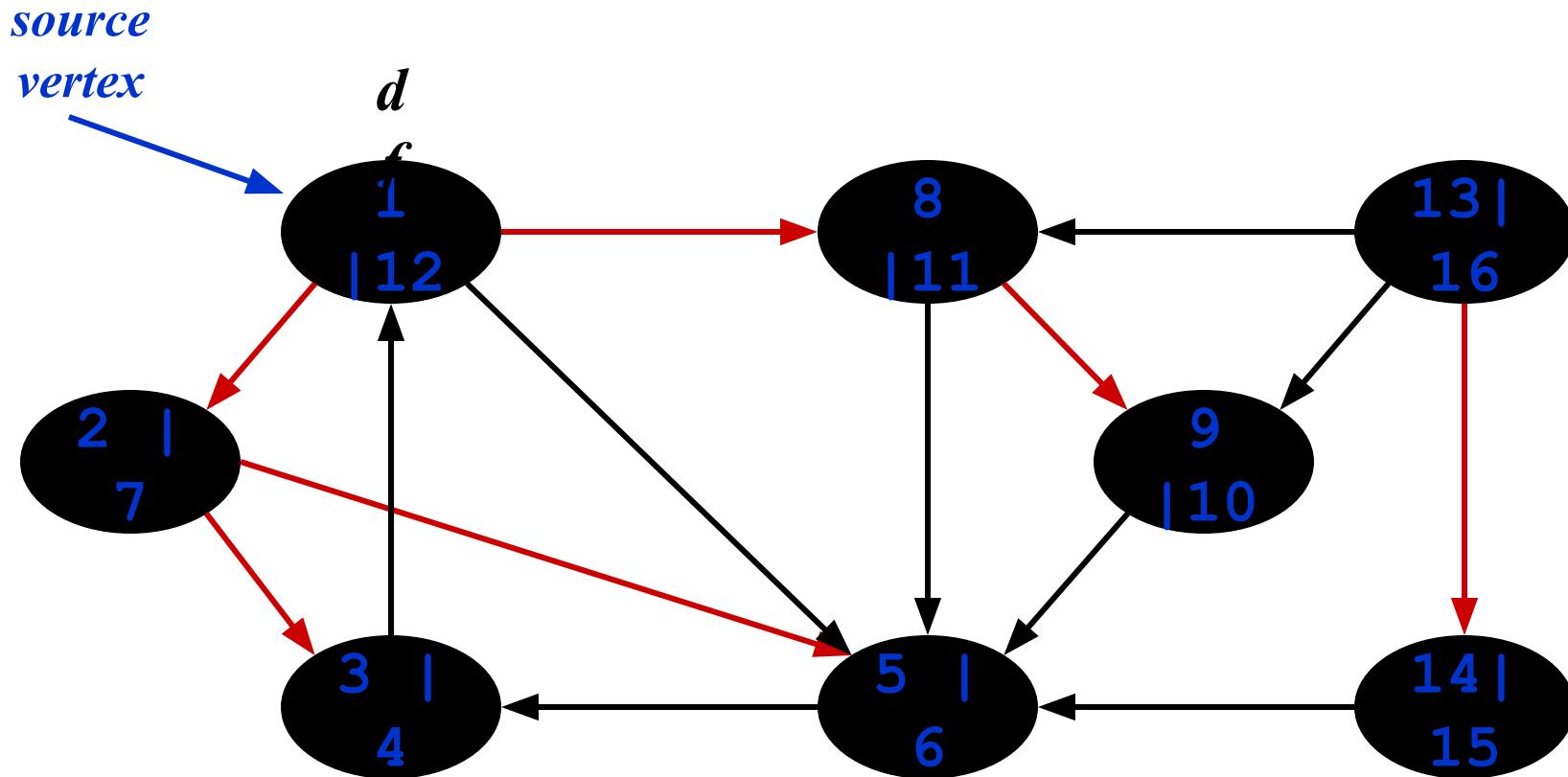
# DFS Example



# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
    - The tree edges form a spanning forest
    - *Can tree edges form cycles? Why or why not?*

# DFS: Kinds of edges

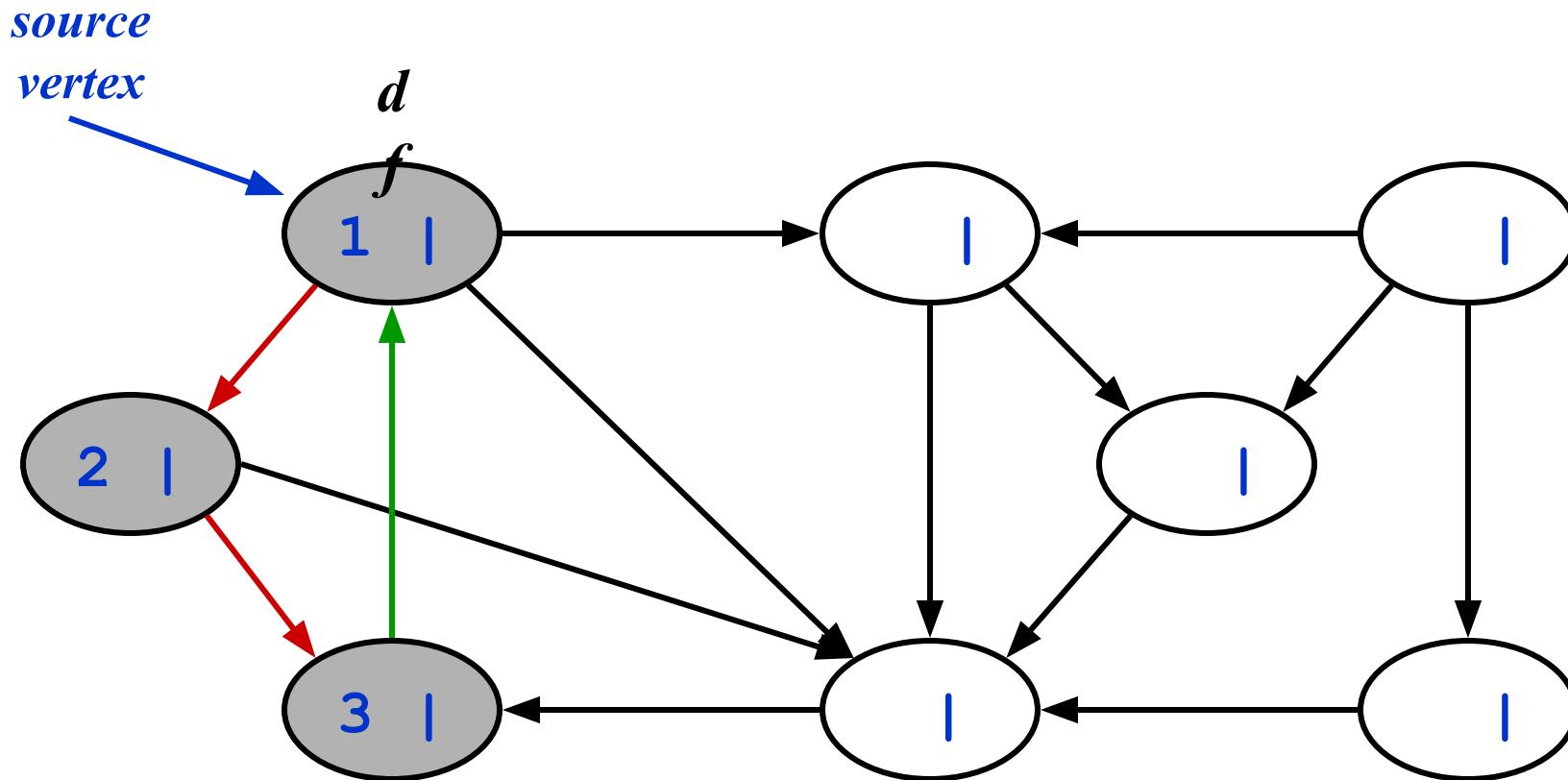


*Tree edges*

# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
    - Encounter a grey vertex (grey to grey)

# DFS: Kinds of edges

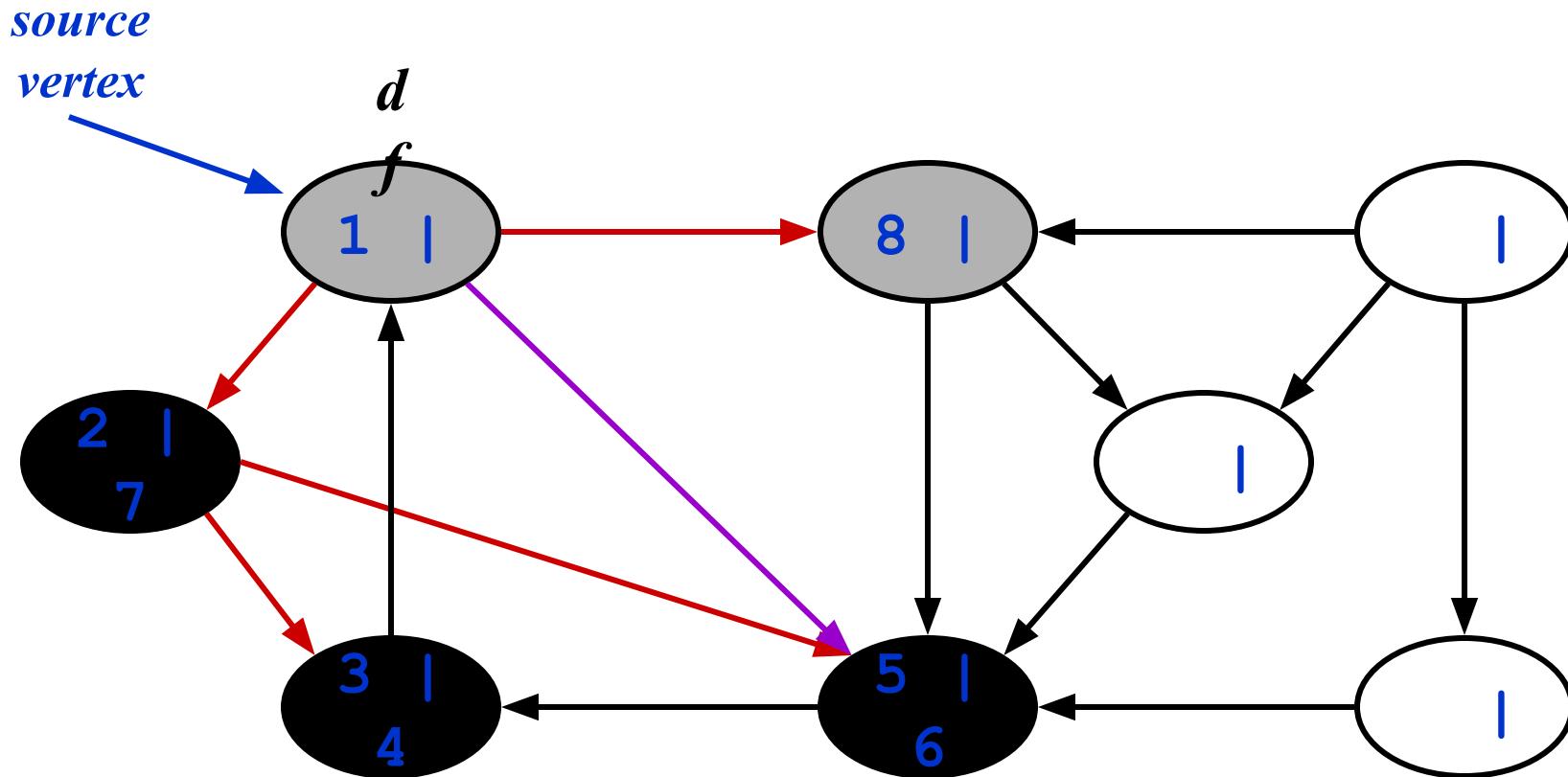


*Tree edges*   *Back edges*

# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
    - Not a tree edge, though
    - From grey node to black node

# DFS: Kinds of edges

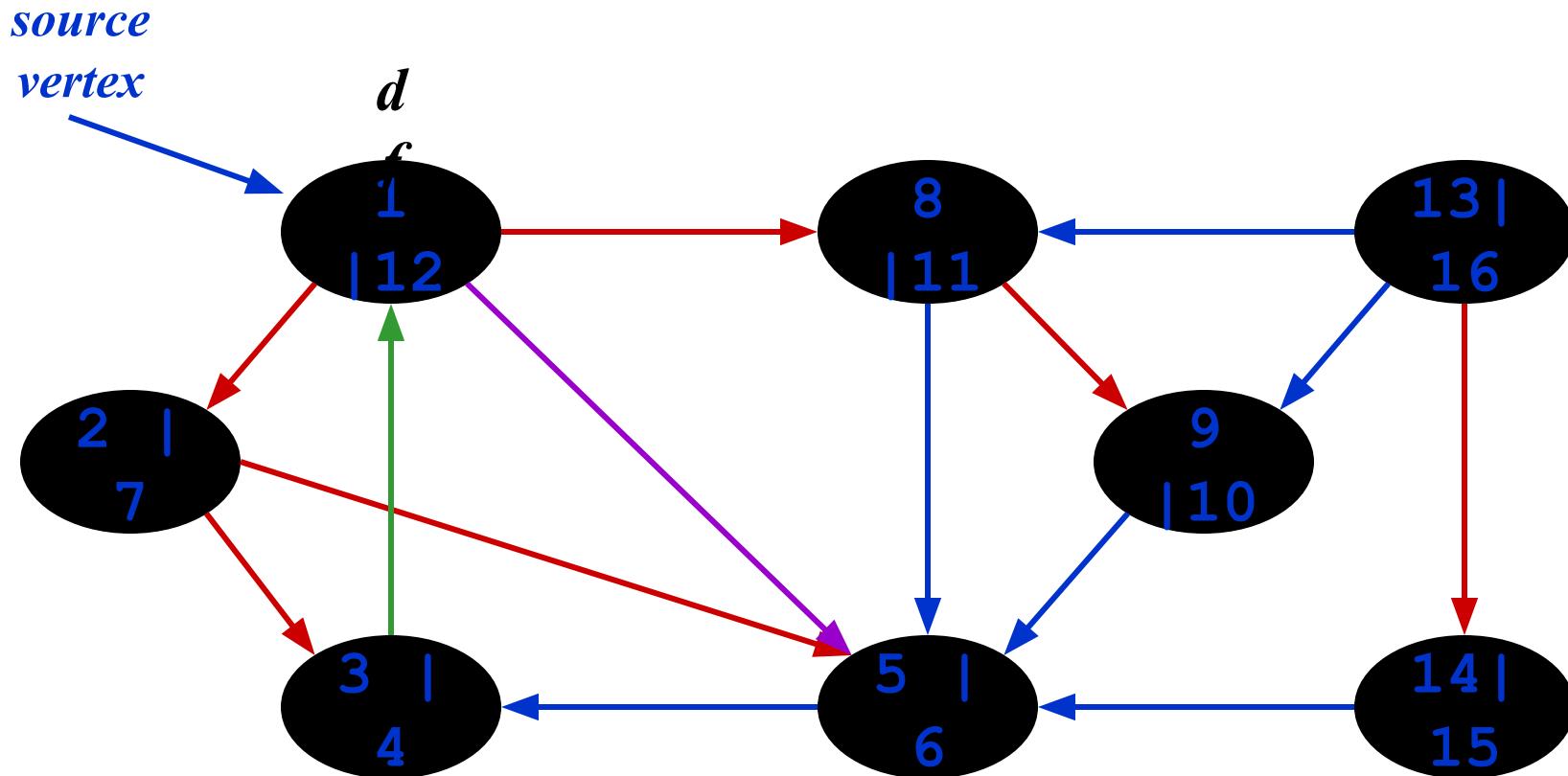


*Tree edges   Back edges   Forward edges*

# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
    - From a grey node to a black node

# DFS: Kinds of edges



*Tree edges    Back edges    Forward edges    Cross edges*

# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
- Note: tree and back edges are very important; some algorithms use forward and cross edges



# Thanks

Do you have any questions?

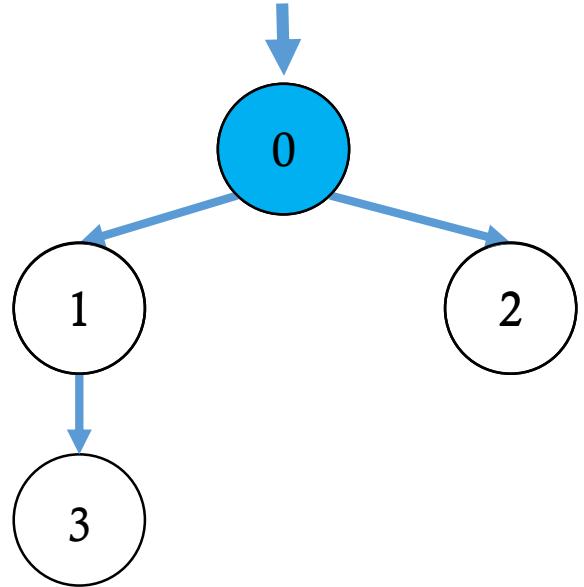
# Breadth First Search

# *Story Behind BFS*

- A monarchy maintains a hierarchical plan
- The first king is decided by people
- Then the rule is simple
- When anyone becomes king he nominates all his children for next king
- When any king departs, the person who stands in front of the nomination list becomes the king
- No person can be king for the second time

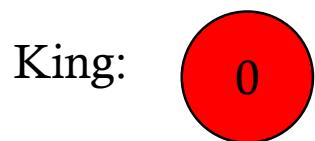
# *Story Behind BFS*

- Lets follow the hierarchy
- The first king decided by people is **0**
- Now, find the order of the king

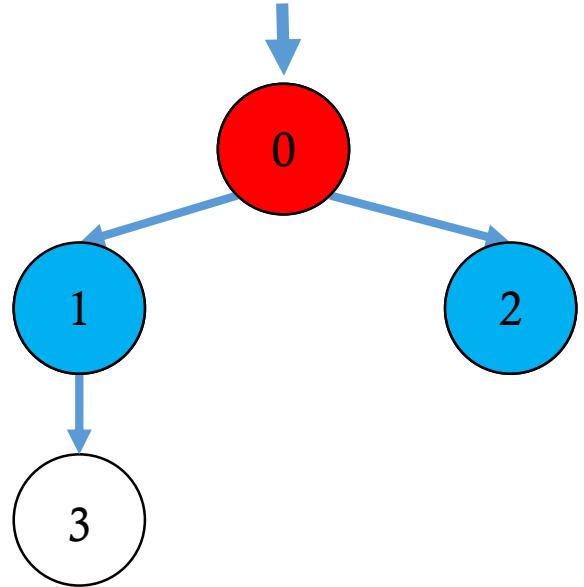
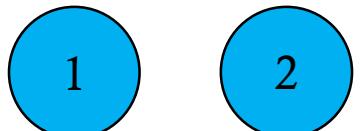


# *Story Behind BFS*

- Lets follow the hierarchy
- The first king decided by people is **0**
- Now, find the order of the king

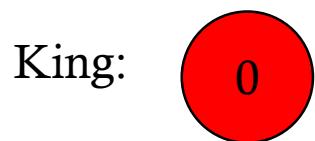


Nomination List:

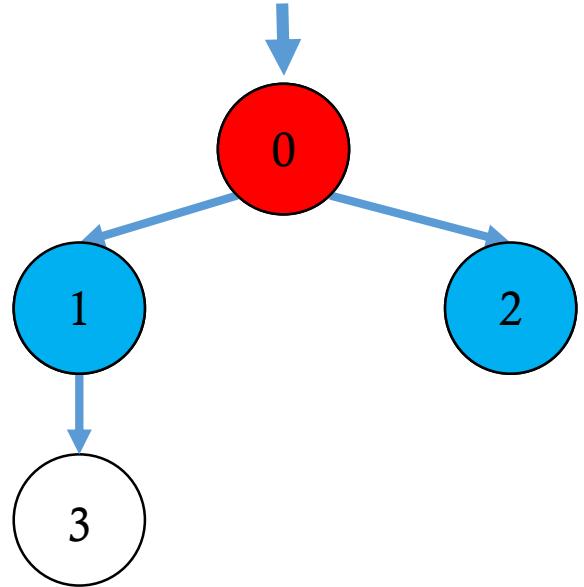
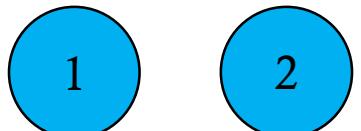


# *Story Behind BFS*

- Lets follow the hierarchy
- The first king decided by people is **0**
- Now, find the order of the king



Nomination List:

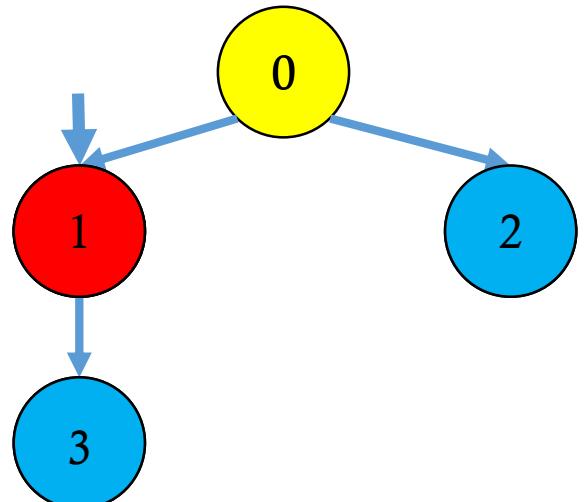


# *Story Behind BFS*

- Lets follow the hierarchy
- The first king decided by people is **0**
- Now, find the order of the king

King:      1

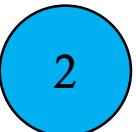
Nomination List:      1      3

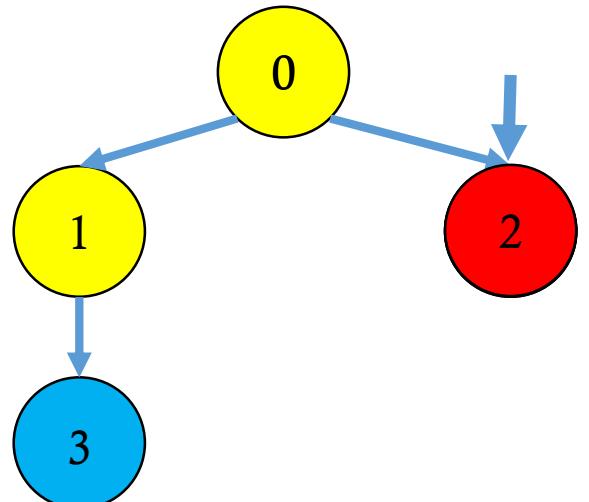


# *Story Behind BFS*

- Lets follow the hierarchy
- The first king decided by people is **0**
- Now, find the order of the king

King: 

Nomination List:  

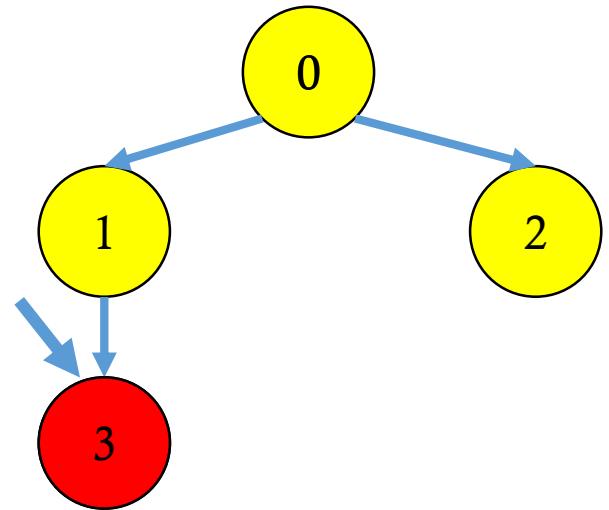


# *Story Behind BFS*

- Lets follow the hierarchy
- The first king decided by people is **0**
- Now, find the order of the king

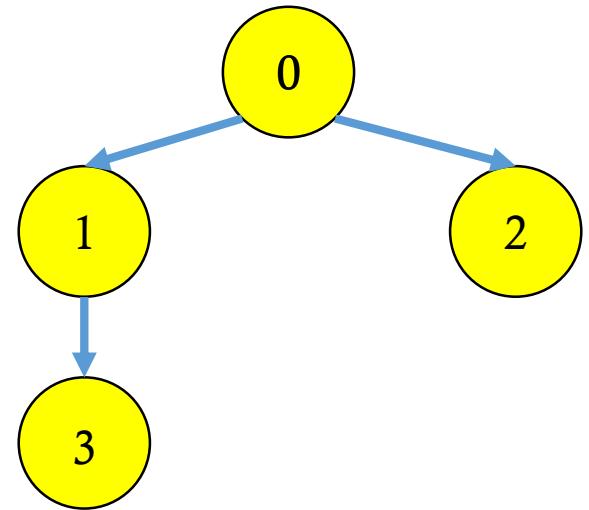
King: 

Nomination List: 



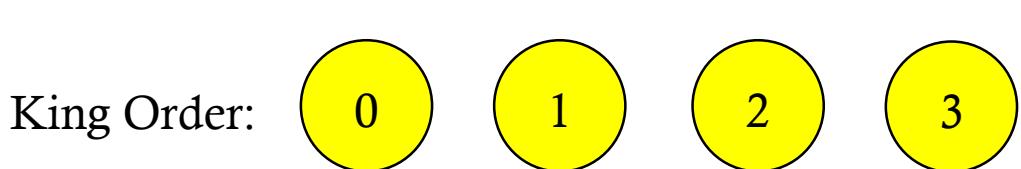
# *Story Behind BFS*

- Lets follow the hierarchy
- The first king decided by people is **0**
- Now, find the order of the king



King:

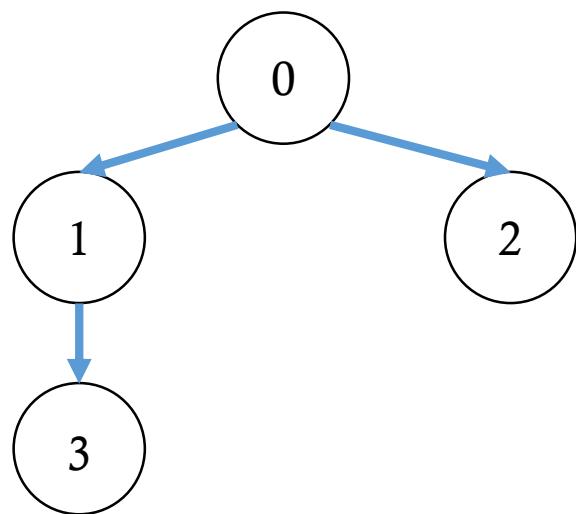
Nomination List:



# *Required Data Structures*

- 2D array for adjacency matrix
- Queue for nomination list

# *Adjacency Matrix*

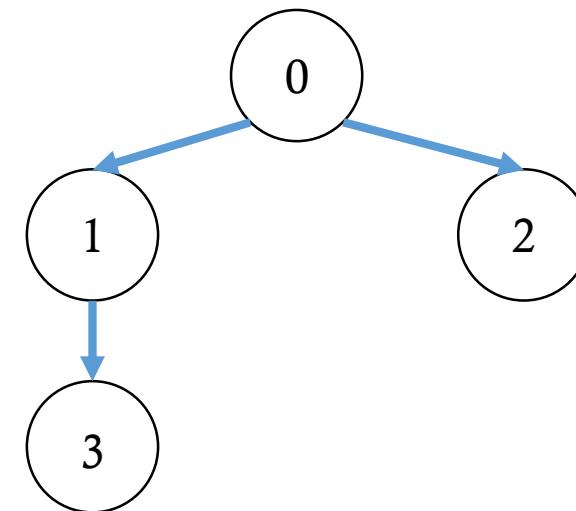


	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	0	0	0
3	0	0	0	0

# *Adjacency Matrix*

- Sometimes it is needed to declare a large size adjacency matrix
- Just bound it at the time of using

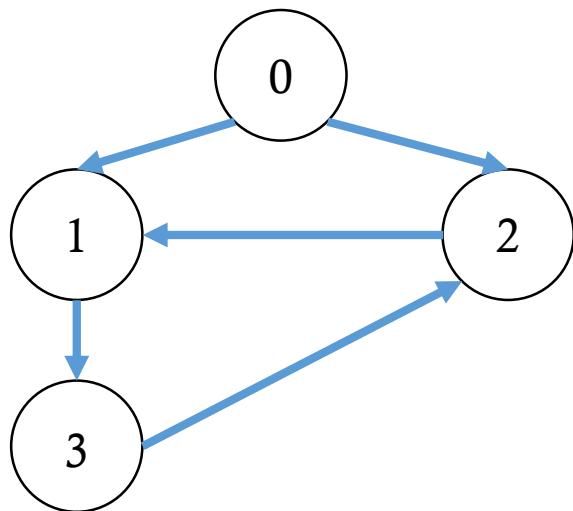
	0	1	2	3	4	5
0	0	1	1	0	0	0
1	0	0	0	1	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0



*Lets Do Some Coding*

# *What If*

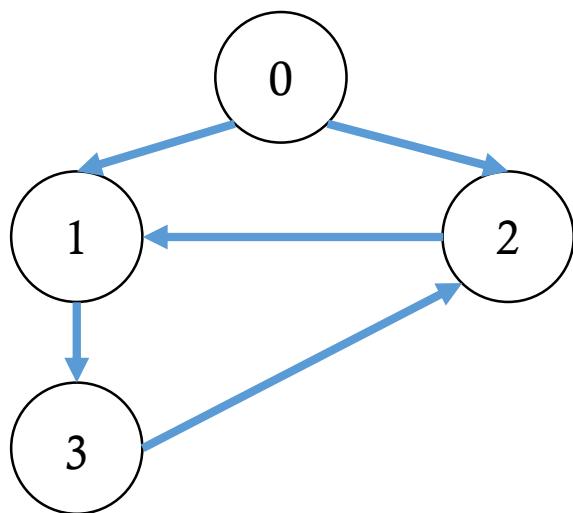
- There is an edge from Vertex-2 to Vertex-1 & Vertex-3 to Vertex-2
- Find the King Order this time (Starting from 0)



King:  
Nomination List: 0

# *What If*

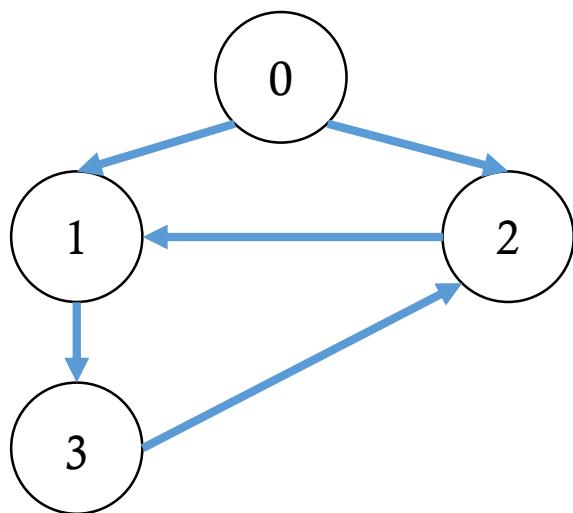
- There is an edge from Vertex-2 to Vertex-1 & Vertex-3 to Vertex-2
- Find the King Order this time (Starting from 0)



King: 0  
Nomination List: 1 2

# *What If*

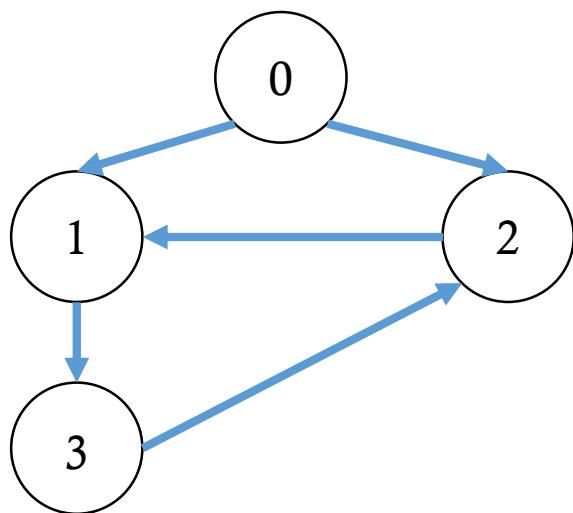
- There is an edge from Vertex-2 to Vertex-1 & Vertex-3 to Vertex-2
- Find the King Order this time (Starting from 0)



King: 1  
Nomination List: 2 3

# *What If*

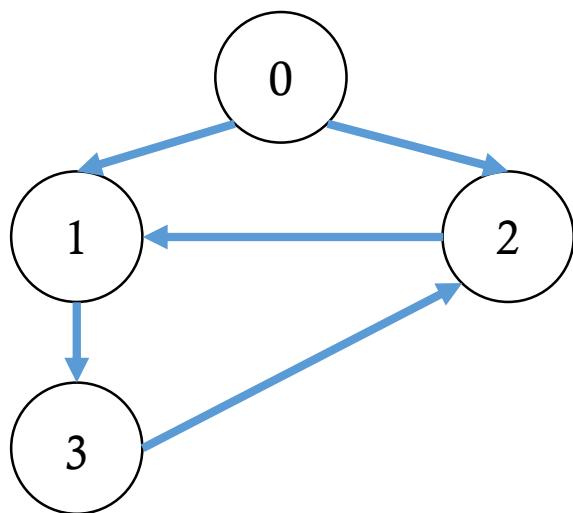
- There is an edge from Vertex-2 to Vertex-1 & Vertex-3 to Vertex-2
- Find the King Order this time (Starting from 0)



King: 2  
Nomination List: 3 1

# *What If*

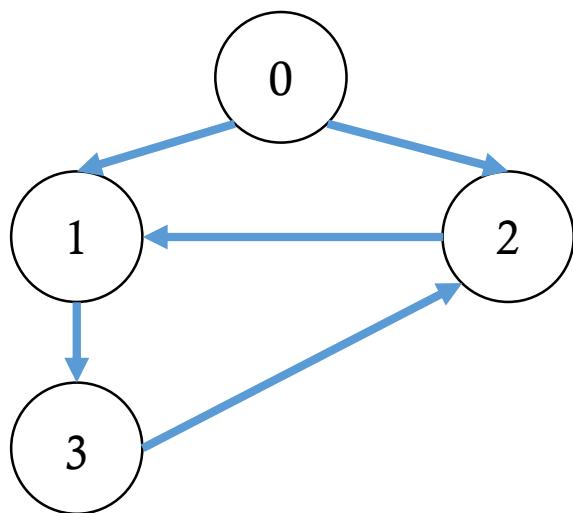
- There is an edge from Vertex-2 to Vertex-1 & Vertex-3 to Vertex-2
- Find the King Order this time (Starting from 0)



King: 3  
Nomination List: 1 2

# *What If*

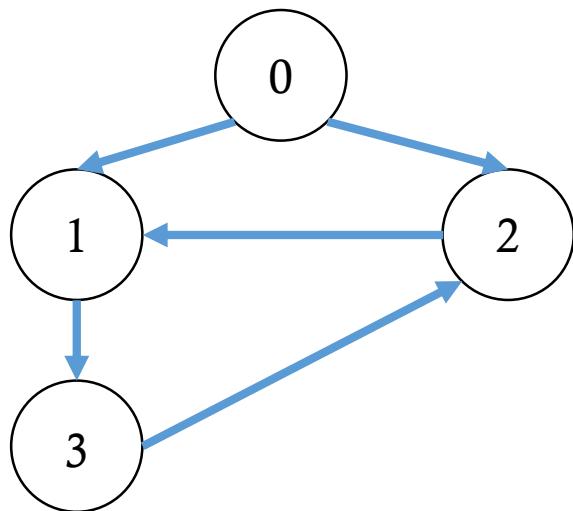
- There is an edge from Vertex-2 to Vertex-1 & Vertex-3 to Vertex-2
- Find the King Order this time (Starting from 0)



King: 1  
Nomination List: 2 3

# *What If*

- There is an edge from Vertex-2 to Vertex-1 & Vertex-3 to Vertex-2
- Find the King Order this time (Starting from 0)



King: 2

Nomination List: 3 1

This becomes an infinite process!

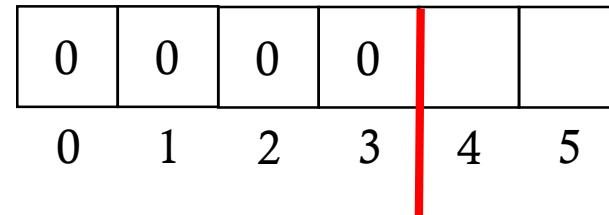
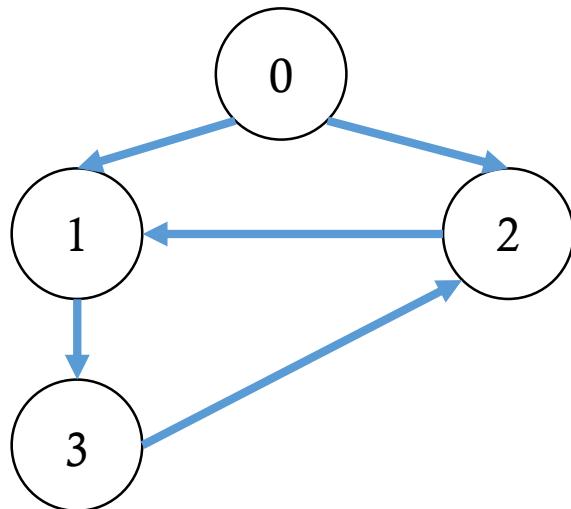
Because same element is entering into the queue for multiple times!

# *Solution*

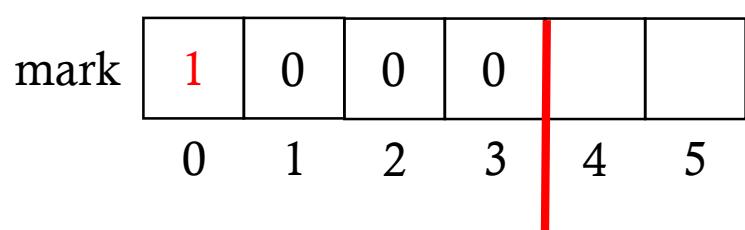
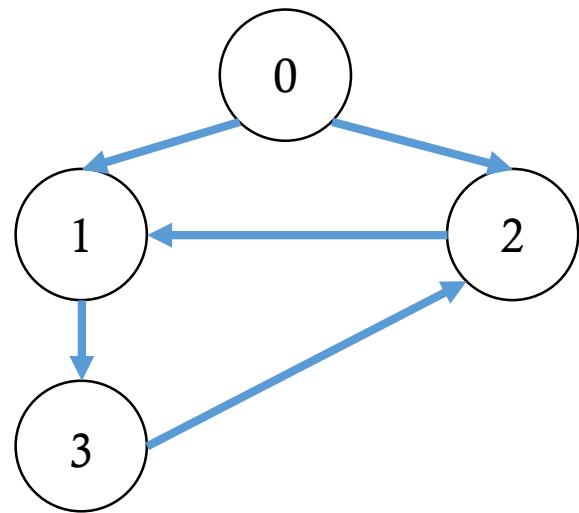
- Mark a vertex at the time of pushing in the queue
  - Checking the mark of a vertex before pushing in the queue
  - Real time example: Election Center
- 
- Required Data Structure
    - 1D Array of dimension **n**

# *Implementation Idea*

- Treat the vertices as the index of the array
- Initially set “No Mark” for all vertices
- A vertex will be pushed in the queue if it has “No Mark”
- Change the mark of a vertex after pushing it to the queue



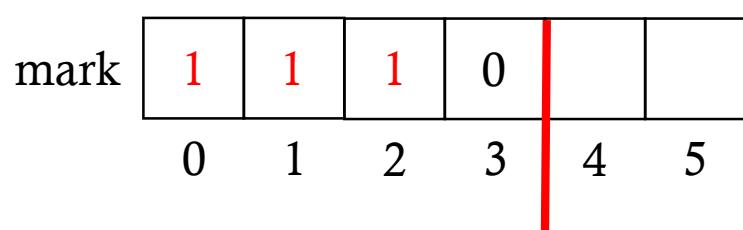
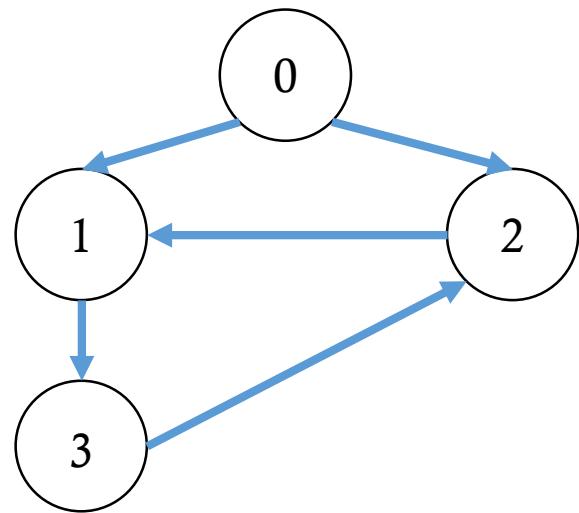
# *Implementation Idea*



mark[0] = 1

King:  
Nomination List: 0

# *Implementation Idea*

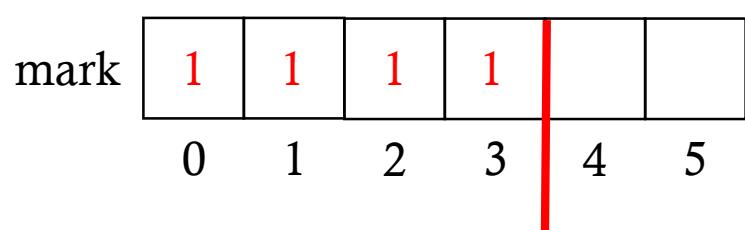
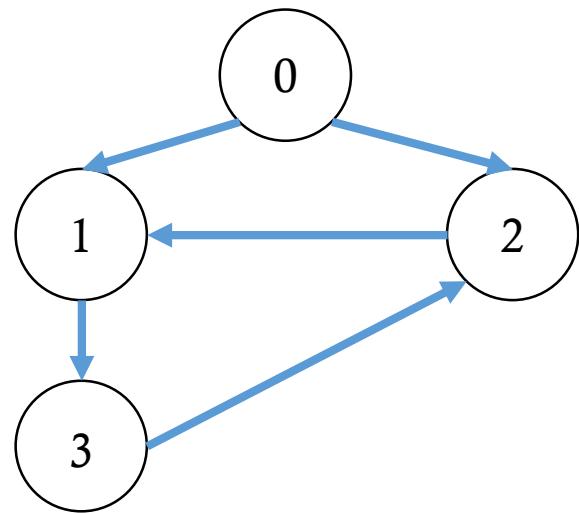


mark[1] = 1

King: 0  
Nomination List: 1 2

mark[2] = 1

# *Implementation Idea*

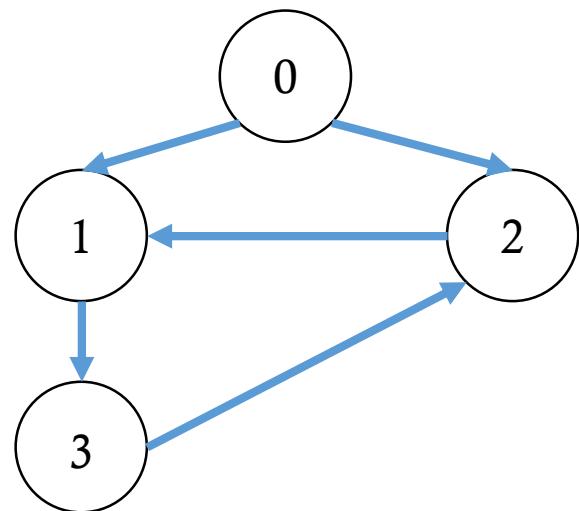


$\text{mark}[3] = 1$

King: 1

Nomination List: 2 3

# *Implementation Idea*



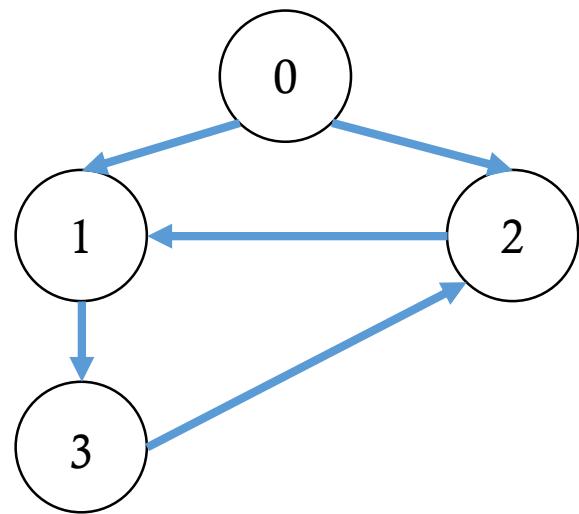
King: 2

Nomination List: 3

mark

1	1	1	1		
0	1	2	3	4	5

# *Implementation Idea*



King: 3

Nomination List:

mark

1	1	1	1		
0	1	2	3	4	5

*Lets Do Some Coding*

# *QUESTIONS*

# Graph

---

## Traversal

DEPTH FIRST SEARCH

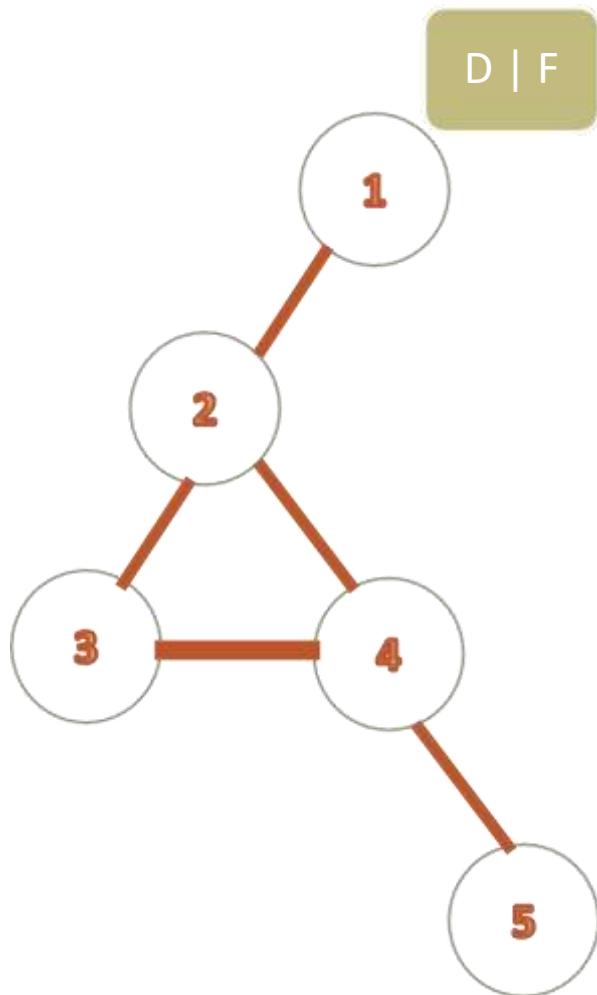
# Depth-First Search: The Code

DFS(G)

```
{  
    for each vertex u ∈ G-  
    >v  
    {  
        u->color = WHITE;  
    }  
    time = 0;  
    for each vertex u ∈ G-  
    >v  
    {  
        if (u->color == WHITE)  
            DFS_Visit(u);  
    }  
}
```

DFS\_Visit(u)

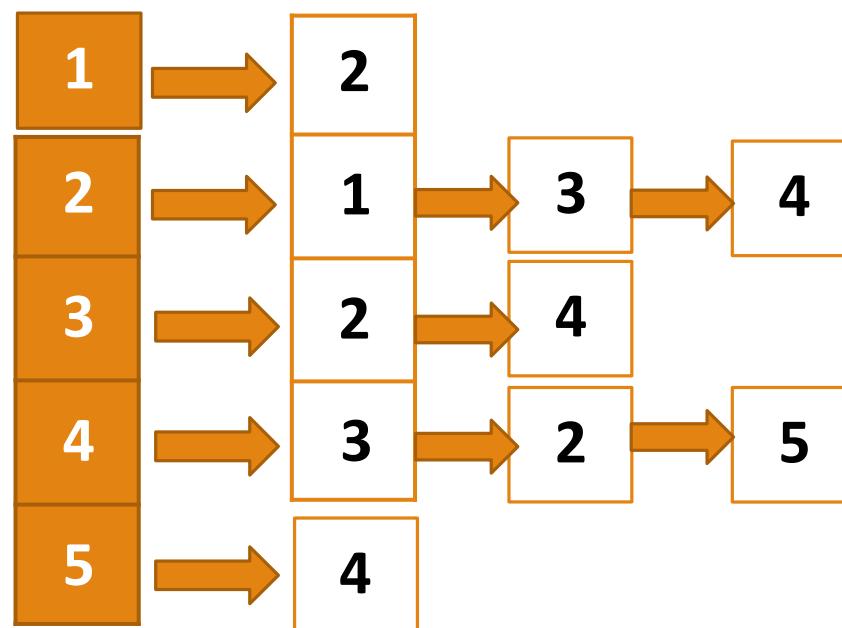
```
{  
    u->color = GREY;  
    time = time+1;  
    u->d = time;  
    for each v ∈ u->Adj[]  
    {  
        if (v->color == WHITE)  
            DFS_Visit(v);  
    }  
    u->color = BLACK;  
    time = time+1;  
    u->f = time;
```

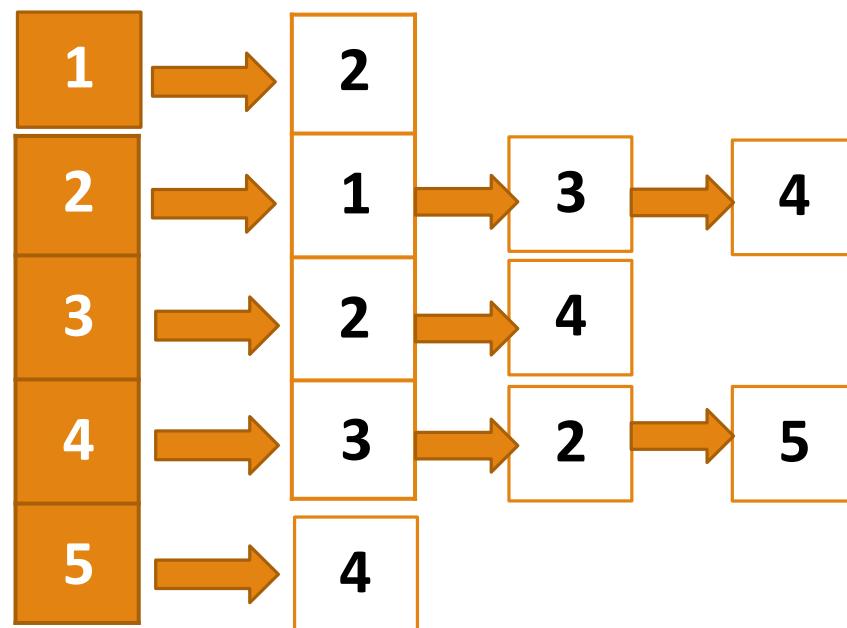
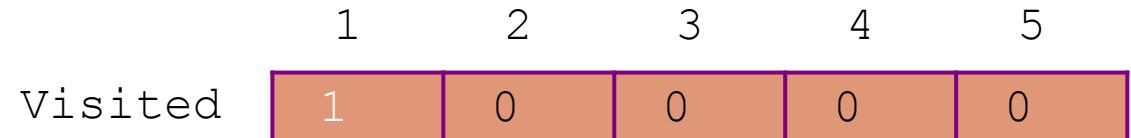
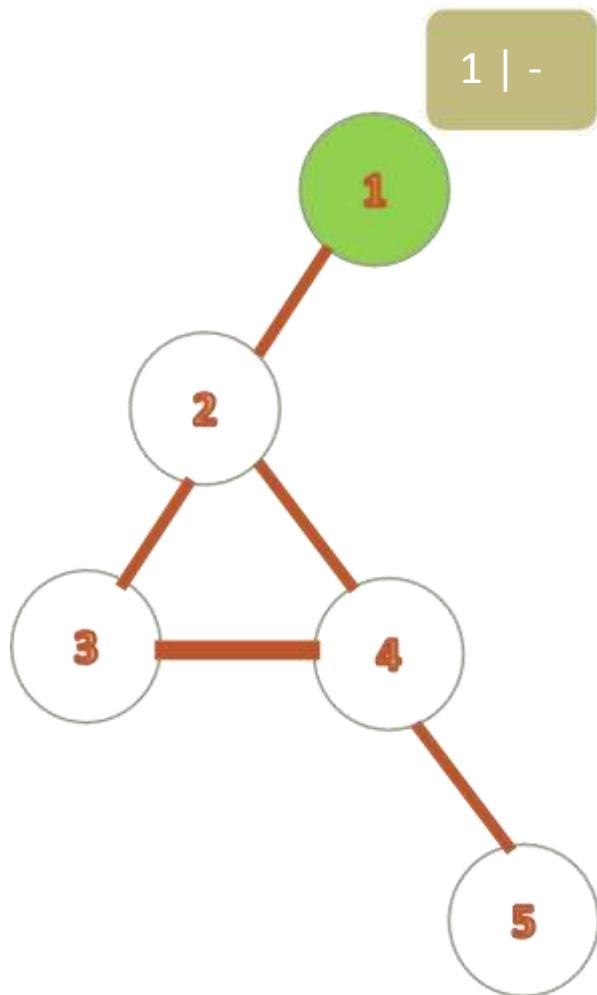


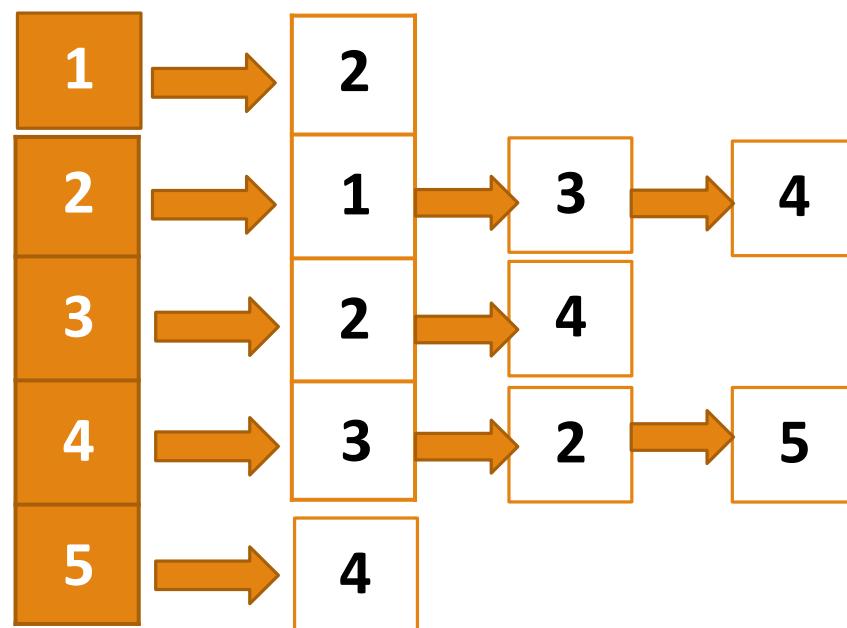
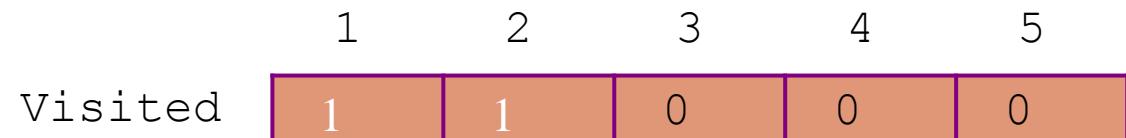
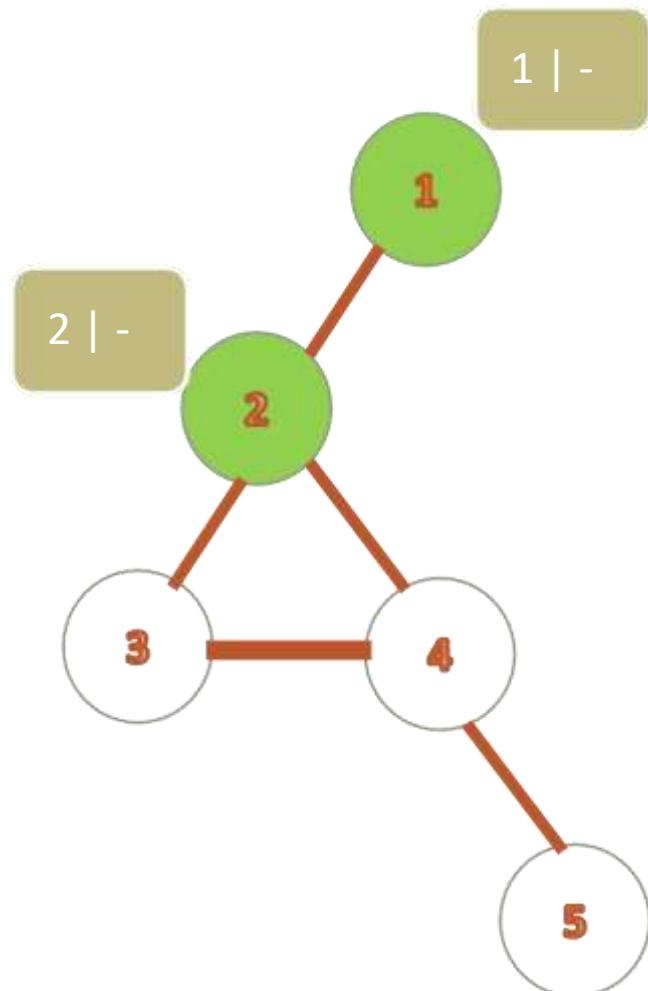
Visited

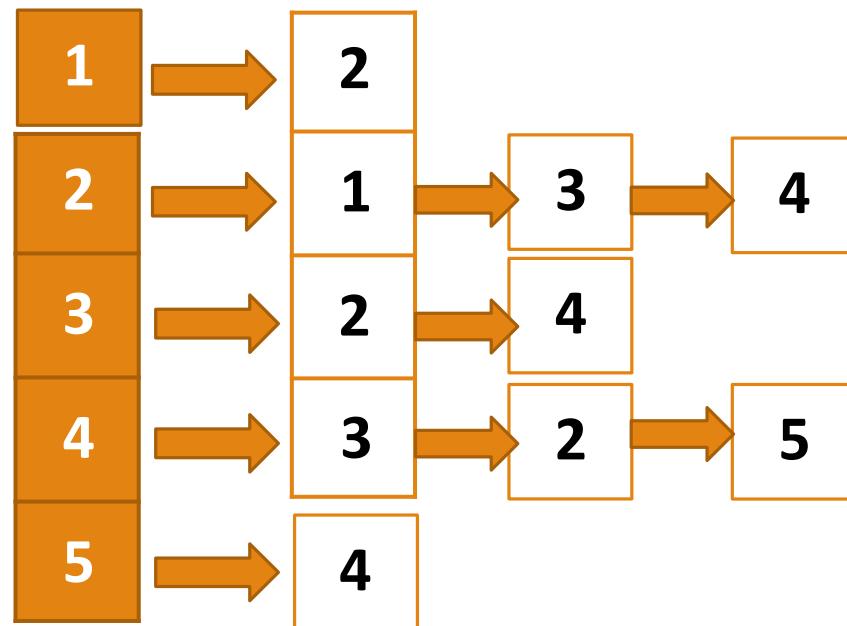
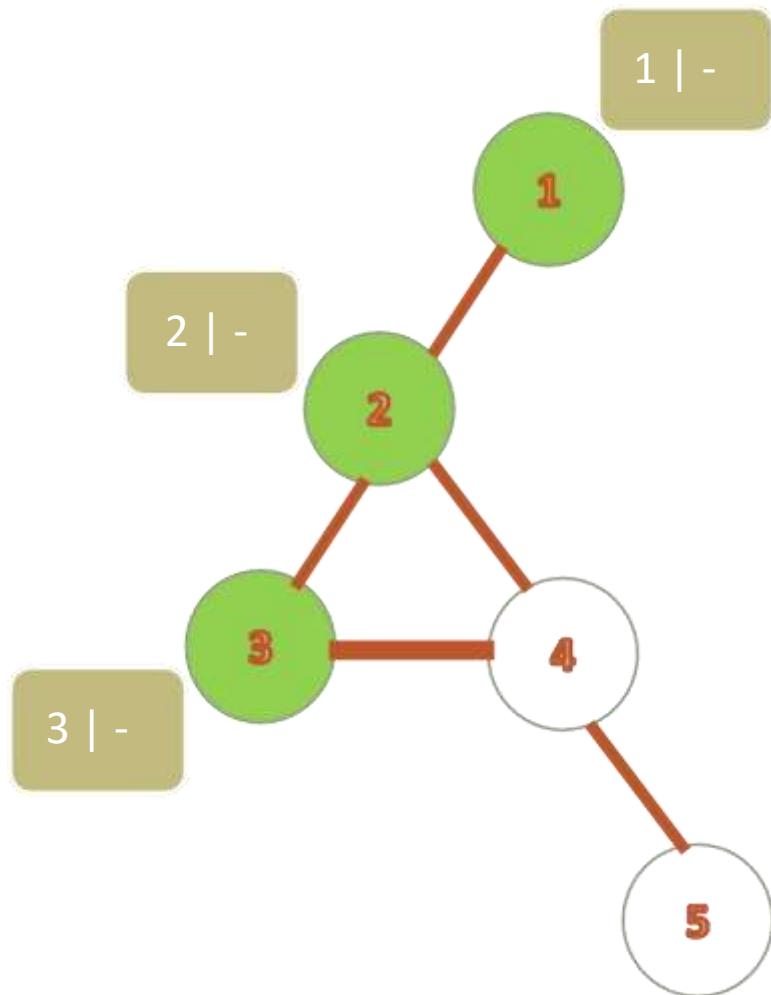
1	2	3	4	5
0	0	0	0	0

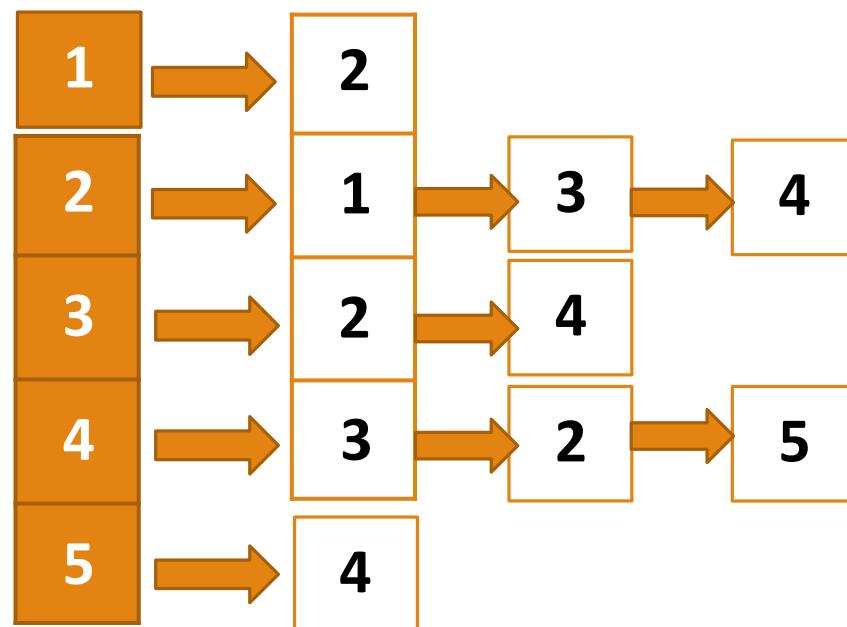
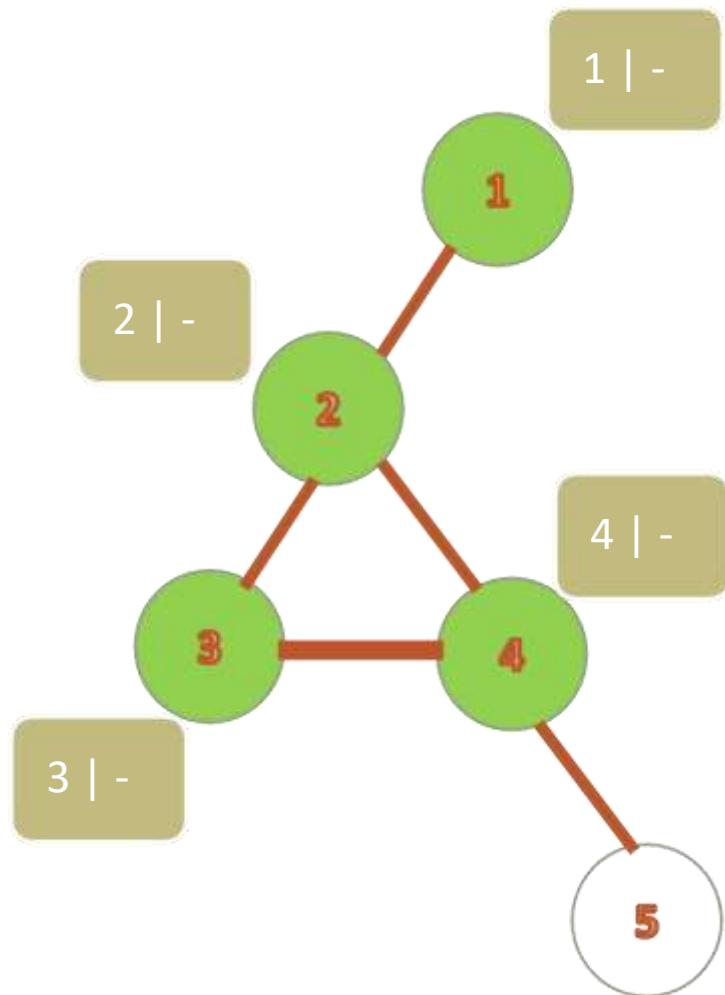
D = Discover Time  
F = Finishing Time

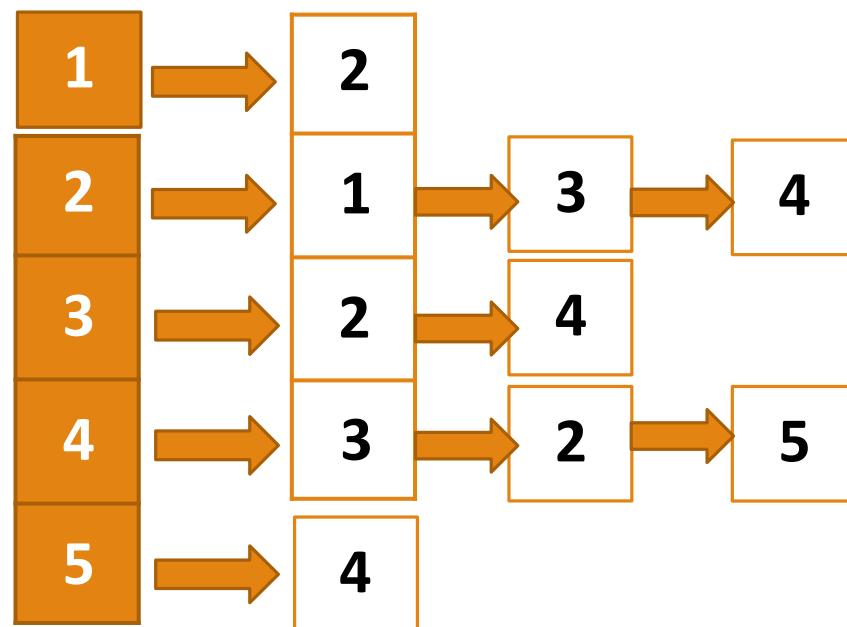
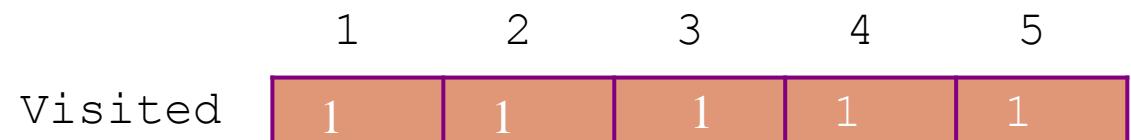
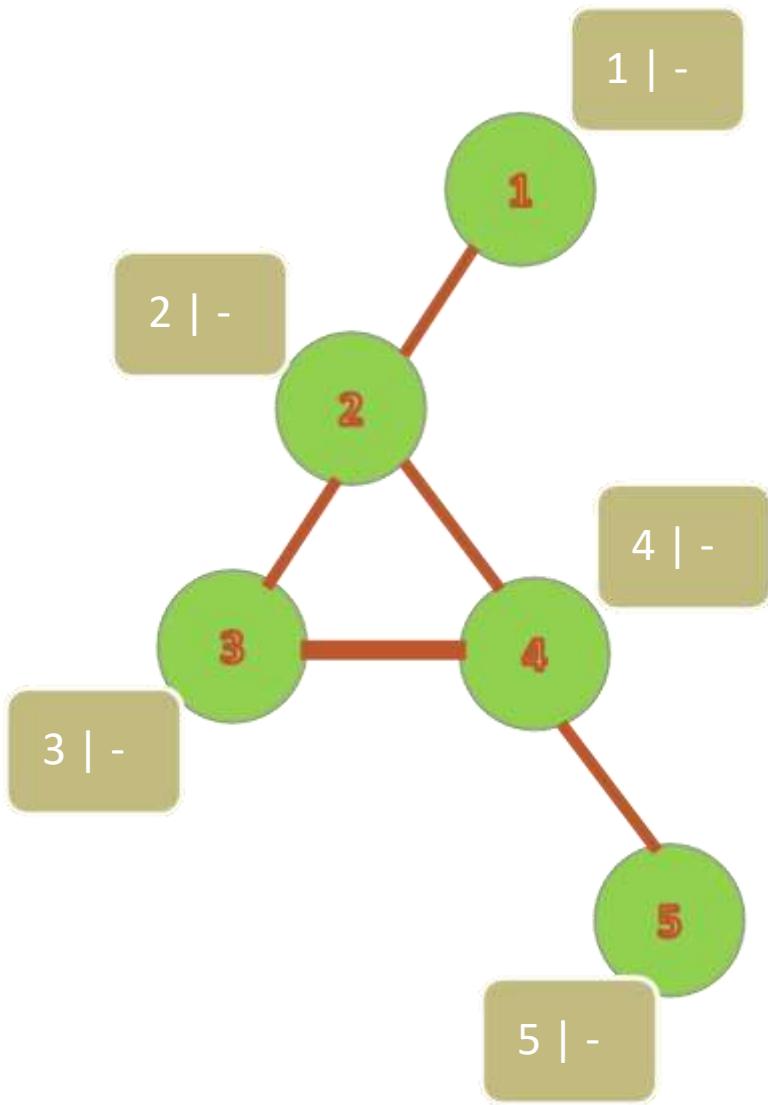


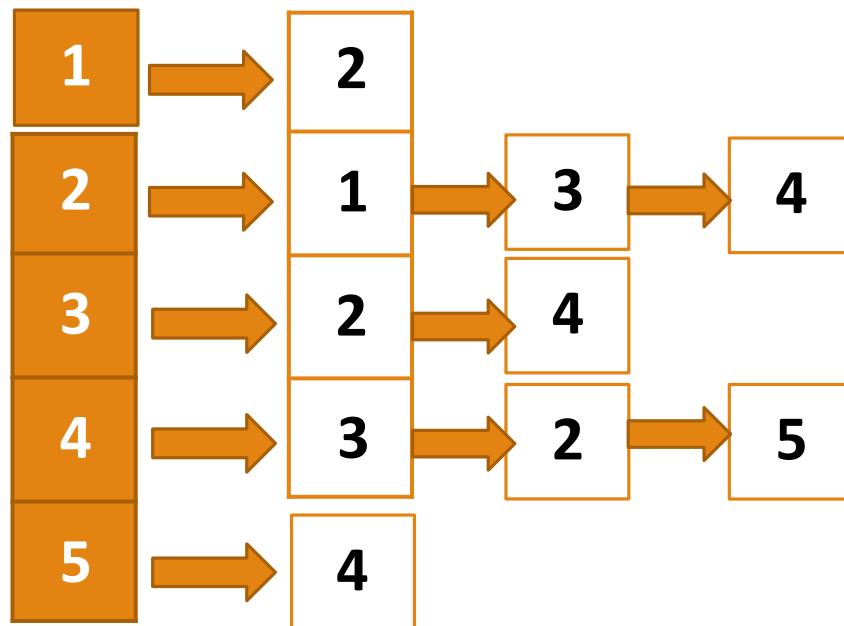
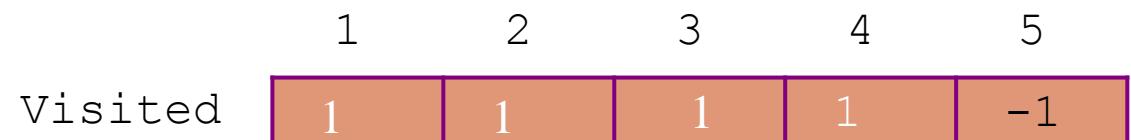
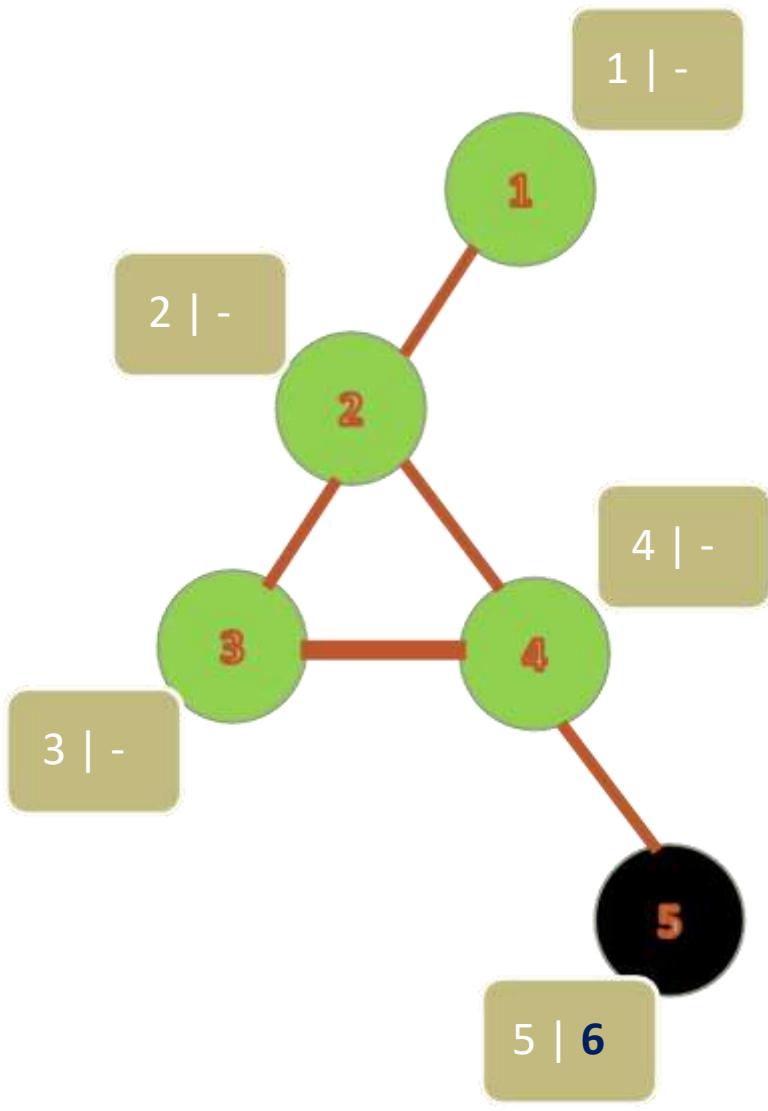


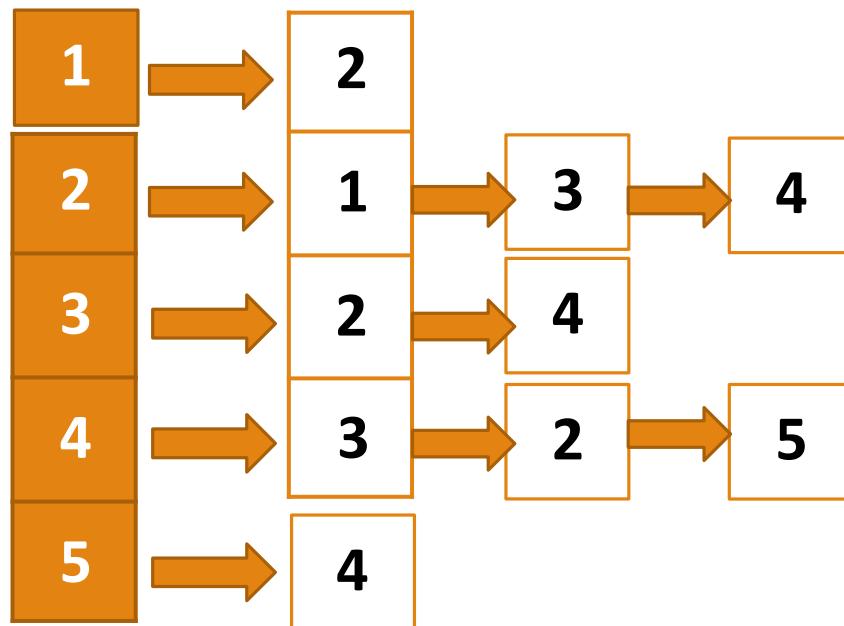
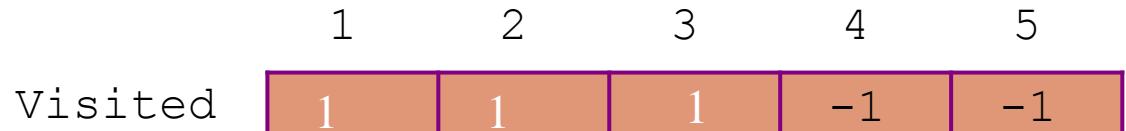
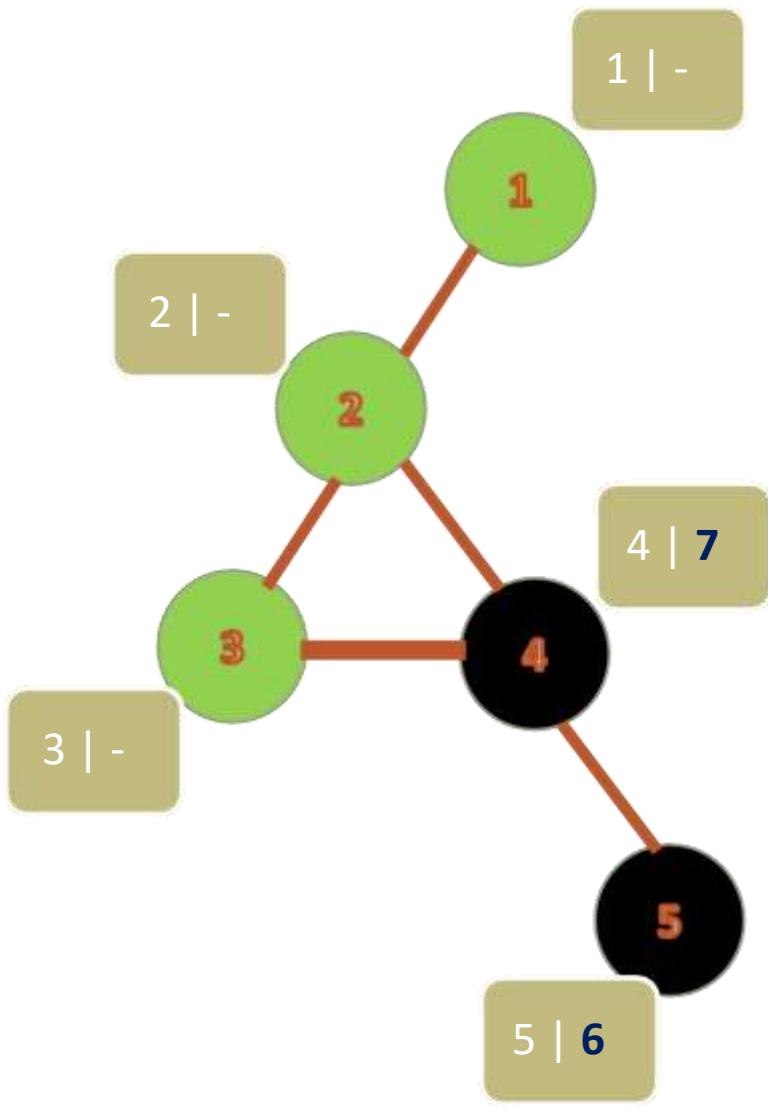


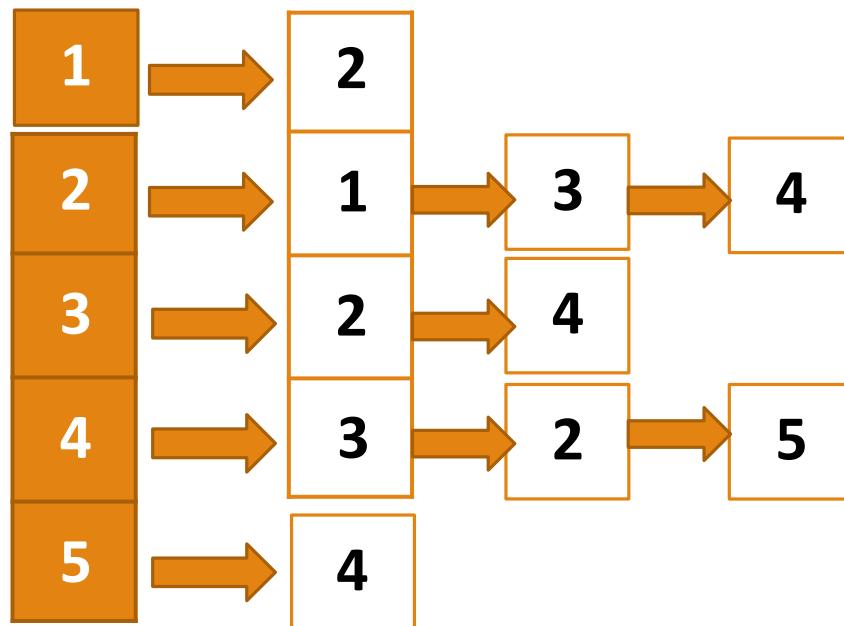
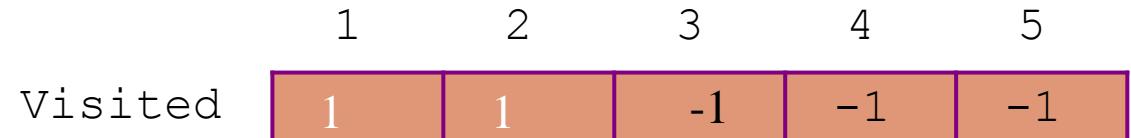
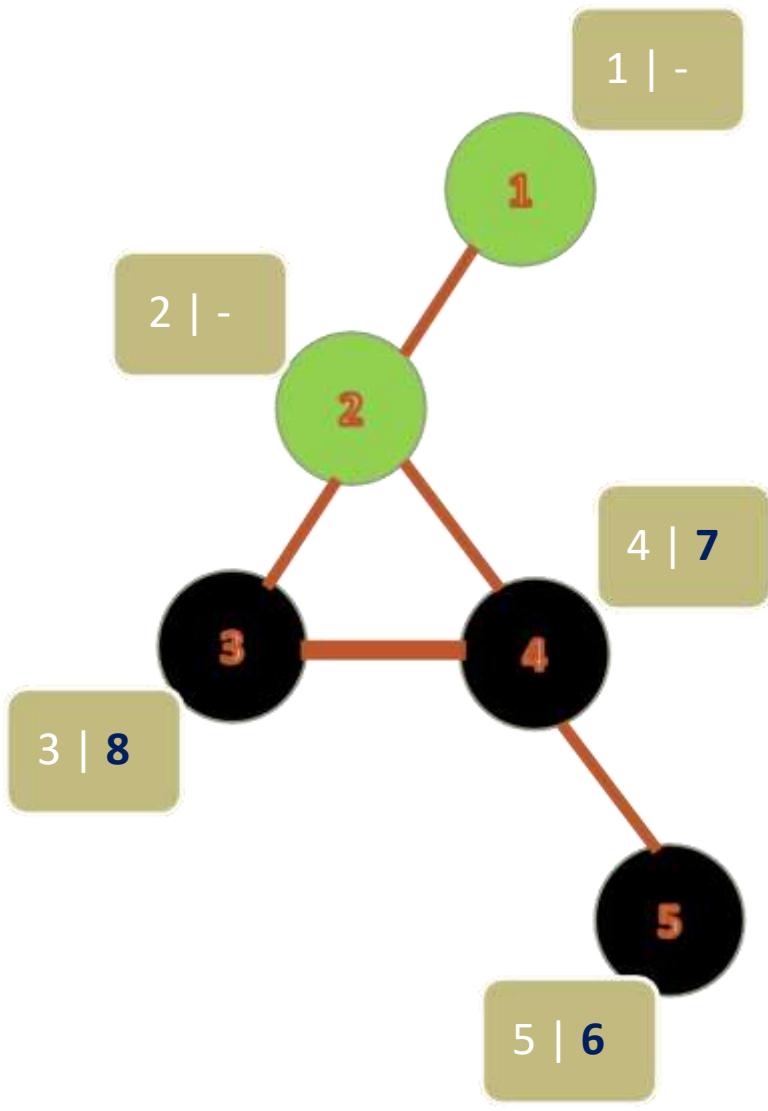


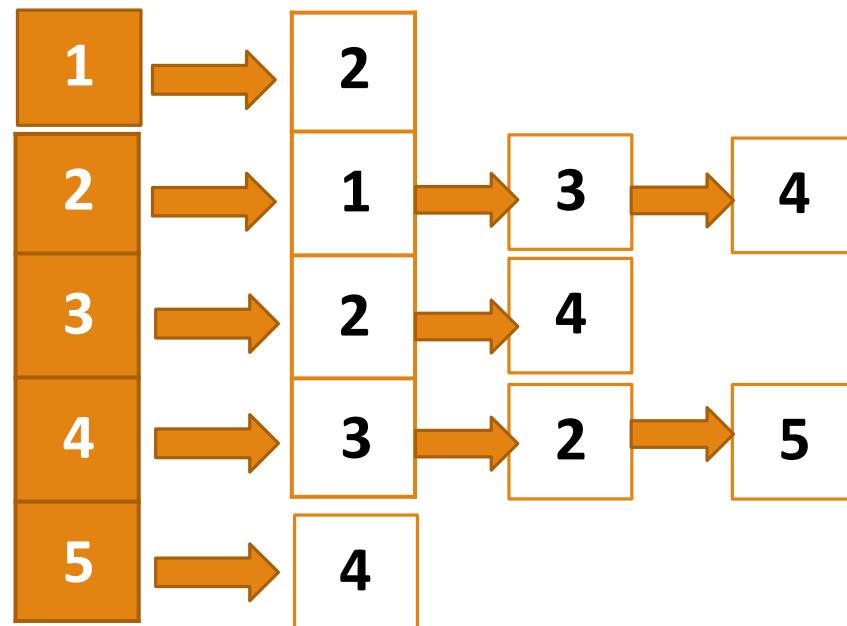
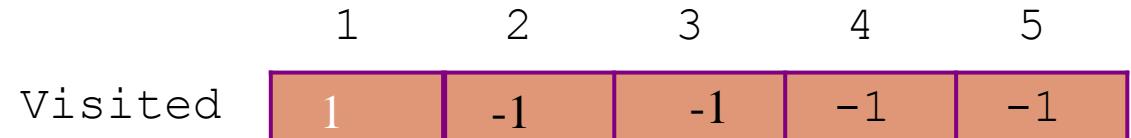
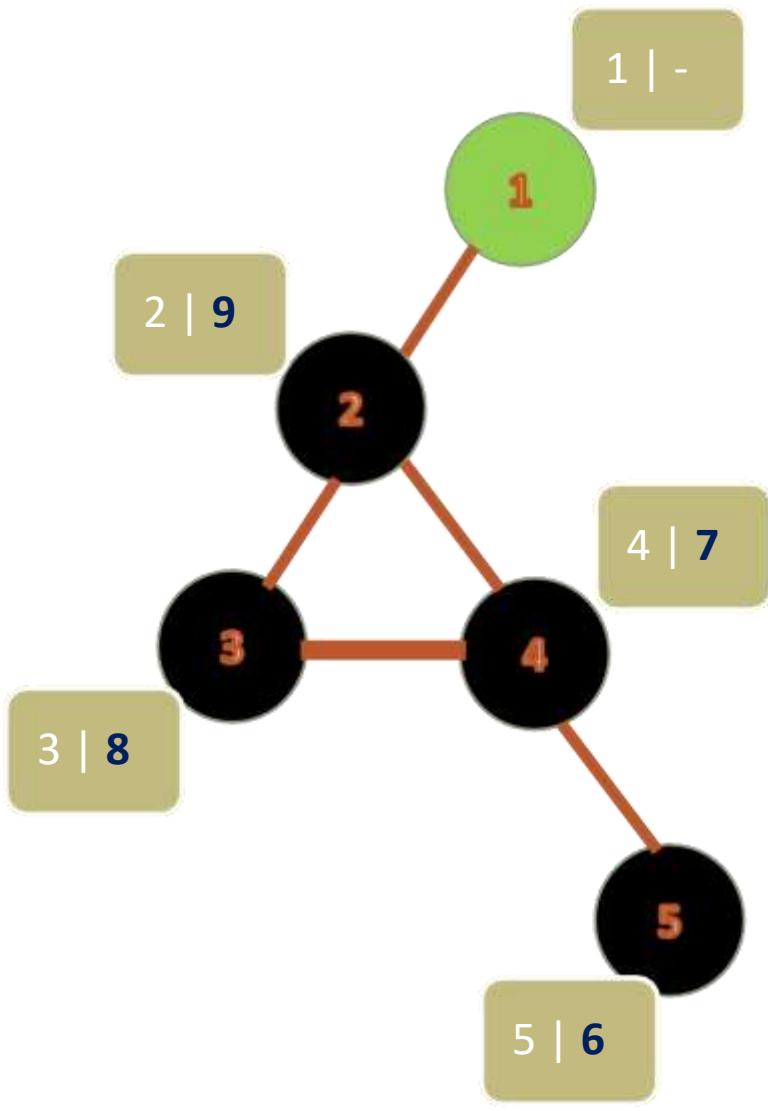


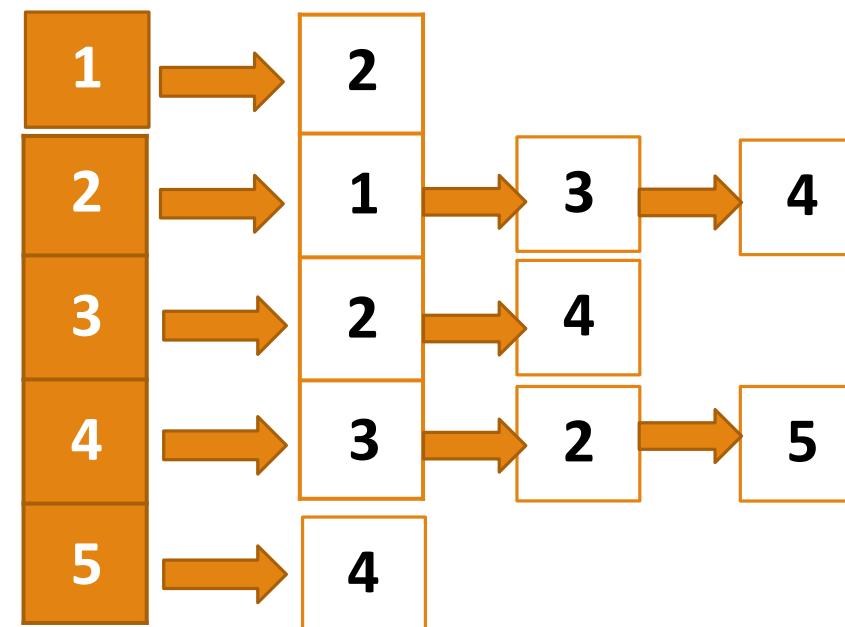
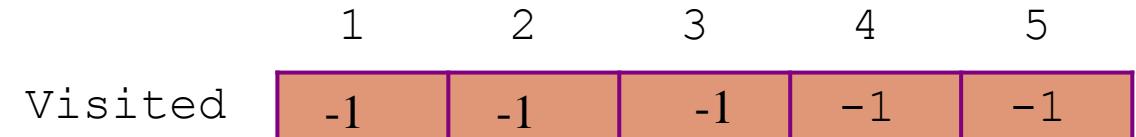
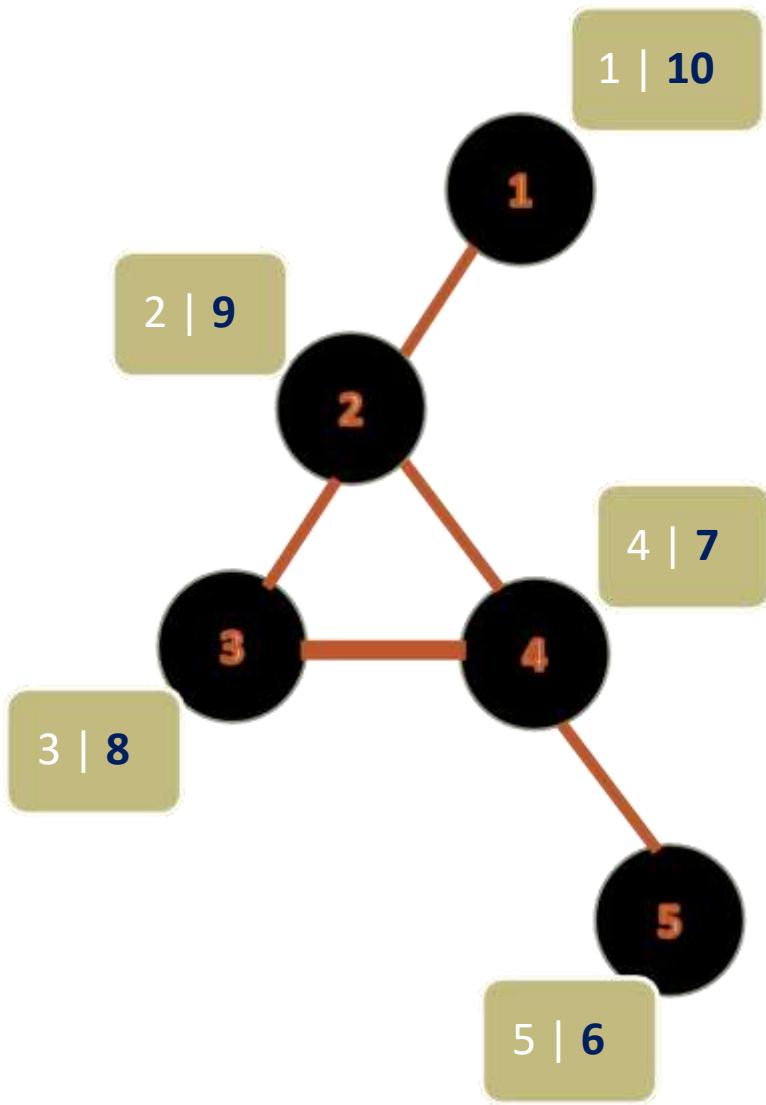












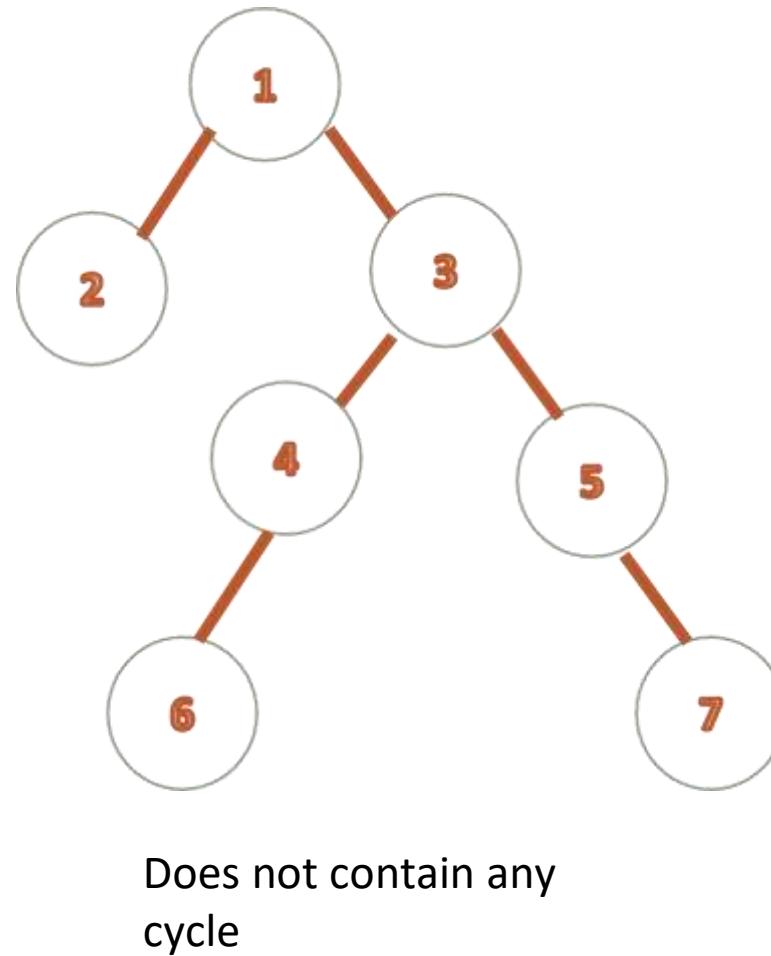
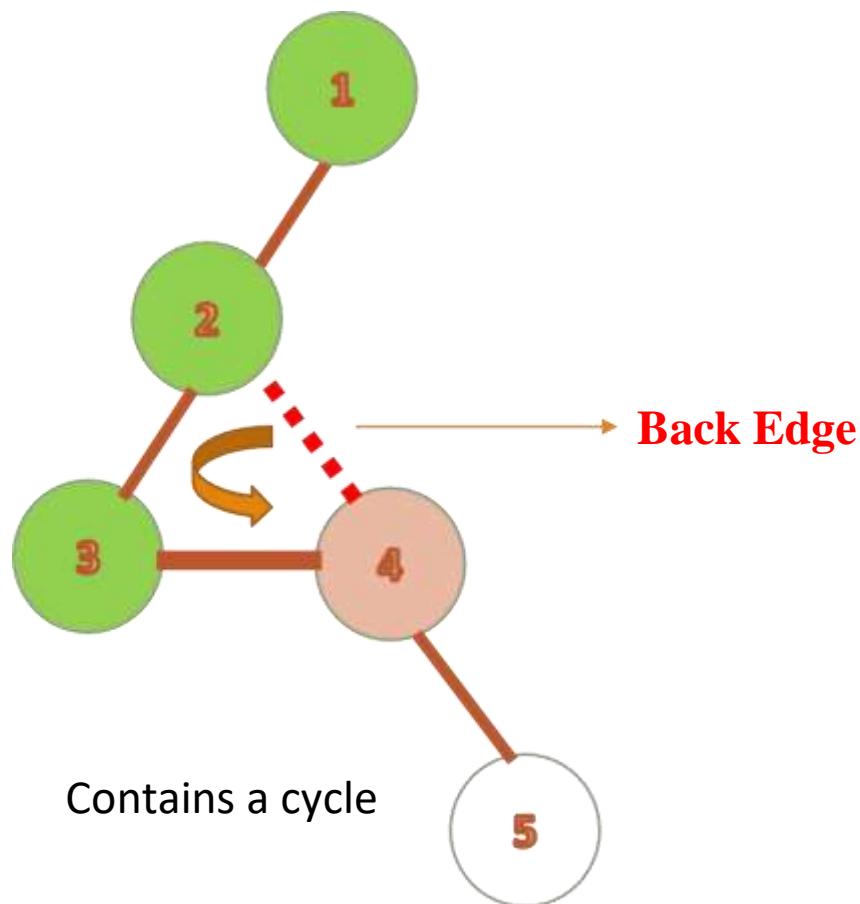
# DFS Implementation

- ***Vector in C++ STL:*** Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- ***memset()*** is used to fill a block of memory with a particular value.

# DFS Application

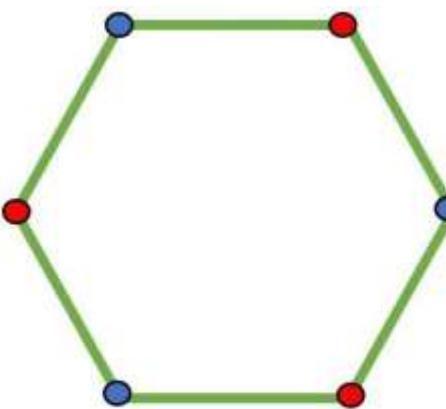
- 1. Minimum Spanning Tree:** If we perform DFS on unweighted graph, then it will generate minimum spanning tree.
- 2. Detecting a Cycle in a Graph:** A graph has a cycle if we found a back edge during DFS. Therefore, we should run DFS for the graph and verify for back edges.
- 3. Path Finding:** We can specialize in the DFS algorithm to search a path between two given vertices  $u$  and  $v$ .
- 4. Bipartite graph checking:** A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. So, to check if the graph is bipartite or not we can use DFS.

# Detecting Cycle

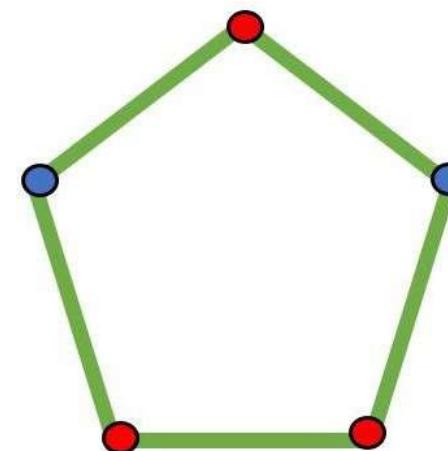


# Bi-coloring

Given a graph, want to give each node a color such that no two neighbors have the same color. Goal is to do color a graph with two colors if possible, else say impossible. Easy !!



Cycle graph of length 6



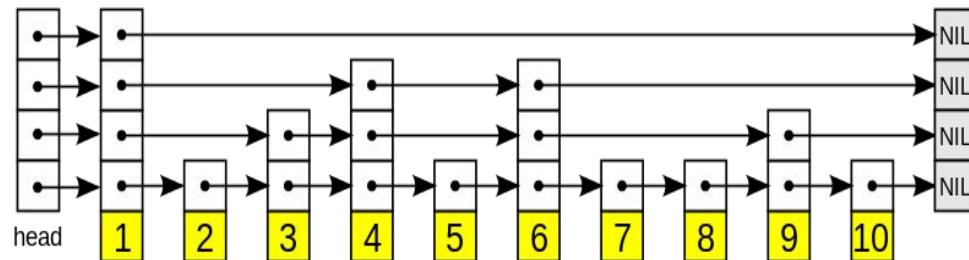
Cycle graph of length 5

---

# Thank You

# CSE-205: Data Structures & Algorithms I

## Topic: Skip Lists



Nafiz Imtiaz Khan  
Lecturer, Department of CSE, MIST  
Email: nafiz@cse.mist.ac.bd



# Skip List

---

- ❑ Invented around 1990 by Bill Pugh
- ❑ Generalization of sorted linked lists – so simple to implement
- ❑ Expected search time is  $O(\log n)$
- ❑ *Randomized* data structure:
  - ❑ - use random coin flips to build the data structure



# Why Skip List?

- The worst case search time **for a sorted linked list** is  $O(n)$  as we can only linearly traverse the list **and cannot skip nodes while searching.**
- For a Balanced Binary Search Tree, we skip almost half of the nodes after one comparison with root.
- For a sorted array, we have random access and we can apply Binary Search on arrays.
- **Can we augment sorted linked lists to make the search faster?**
- **The answer is Skip List.**
- The idea is simple, we create multiple layers so that we can skip some nodes.



# Why Skip List?

- ❑ Linked list does not support binary search.
- ❑ Skip list allows fast search and insertion.
  - ❑ **Search:**  $O(\log n)$  time complexity on average.
  - ❑ **Insertion:**  $O(\log n)$  time complexity on average.



# Outline

---

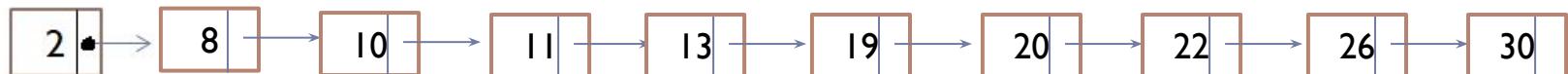
- ❑ Building a skip list.
- ❑ Search.
- ❑ Insertion.
- ❑ Deletion



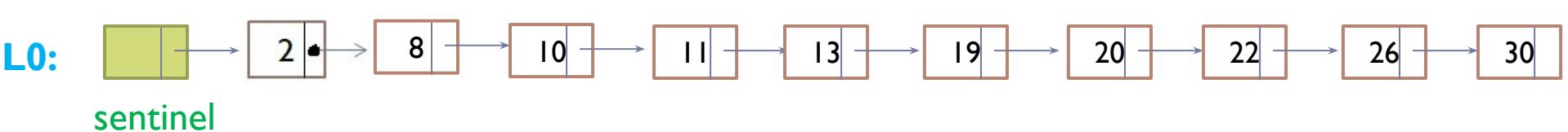
# Building a Skip List

# Initial State

- Initially we have a linked list containing n elements in ascending order.

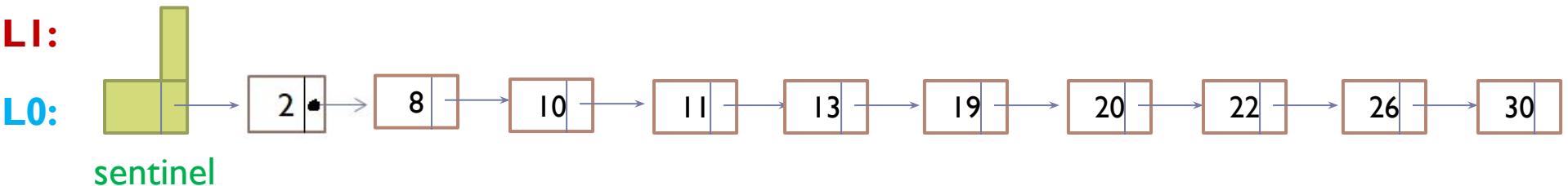


# Add sentinel in the front



# Iteration 1

- Build L1 Linked list

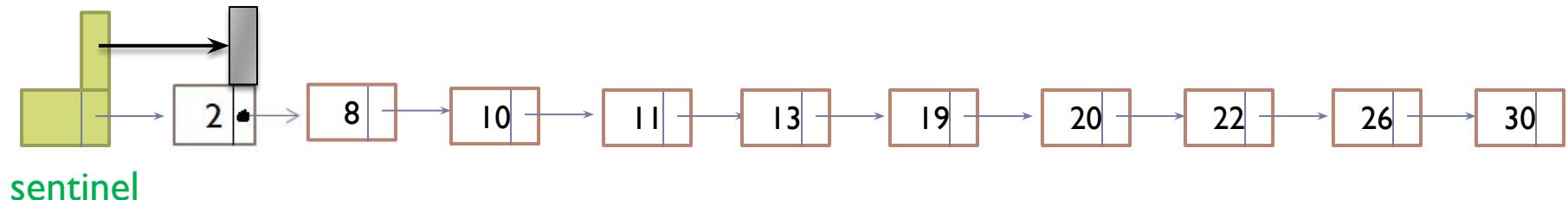


# Iteration 1(A)

Flip a coin



L1:  
L0:

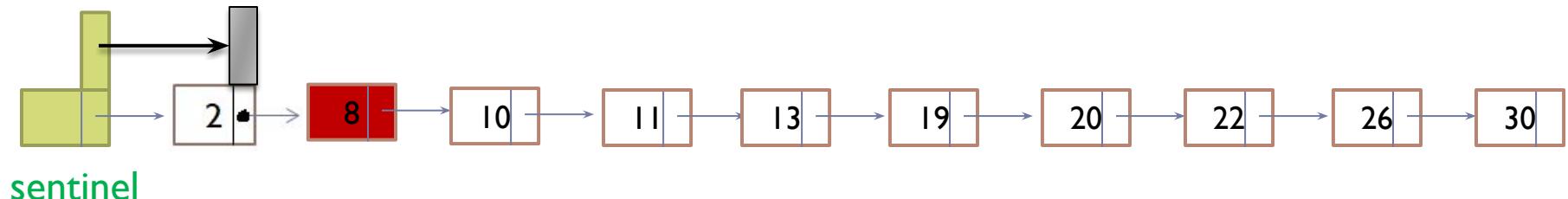


# Iteration 1(B)

Flip a coin



L1:  
L0:

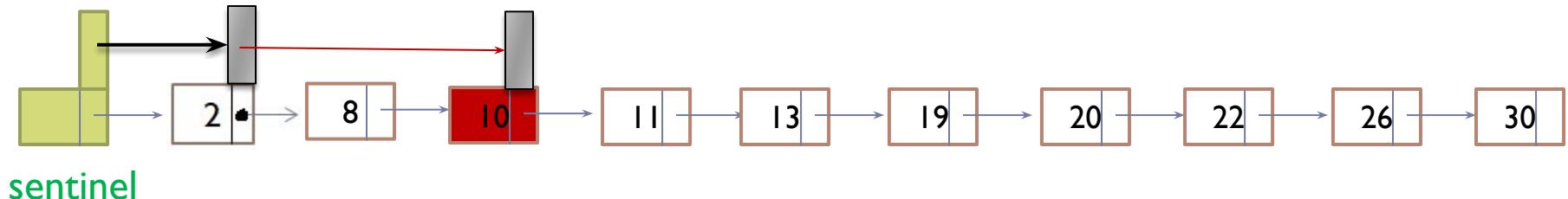


# Iteration 1(C)

Flip a coin



L1:  
L0:

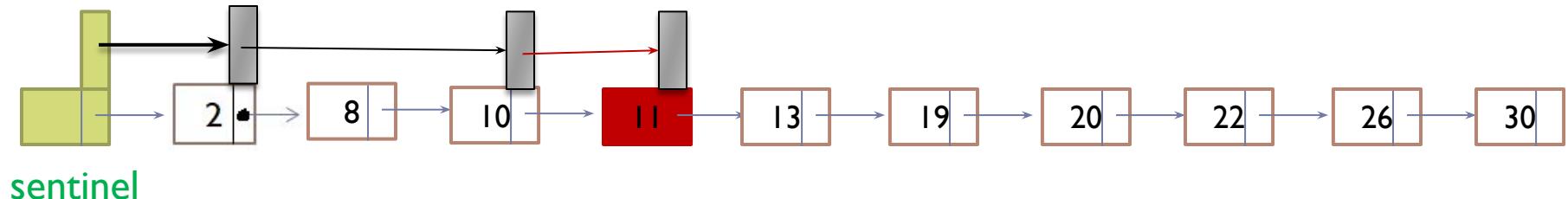


# Iteration 1(D)

Flip a coin



L1:  
L0:

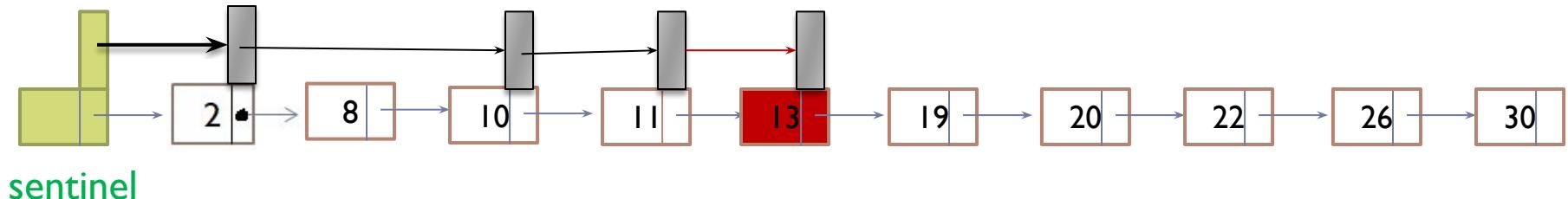


# Iteration 1(E)

Flip a coin



L1:  
L0:



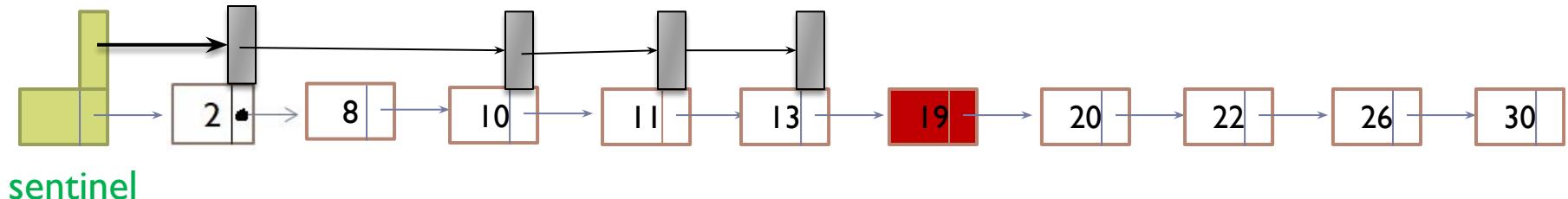
sentinel

# Iteration 1(F)

Flip a coin



L1:  
L0:

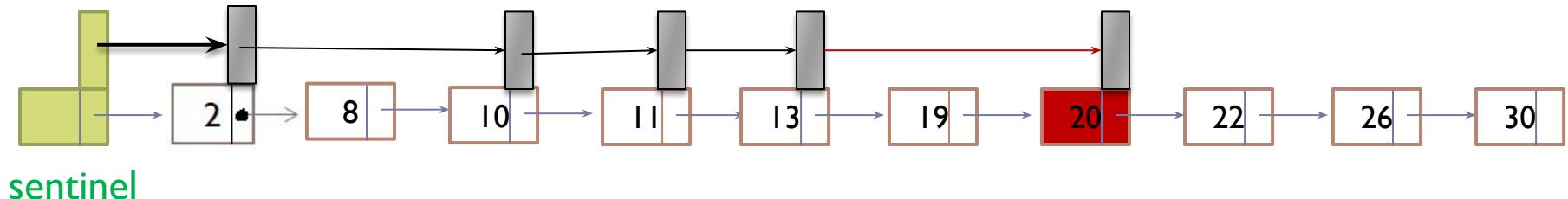


# Iteration 1(G)

Flip a coin



L1:  
L0:



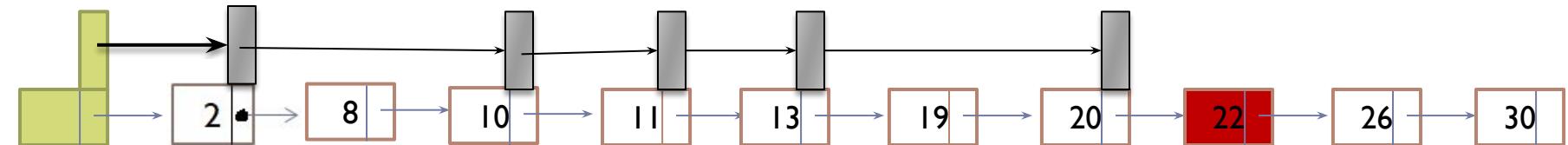
sentinel

# Iteration 1(H)

Flip a coin



L1:  
L0:

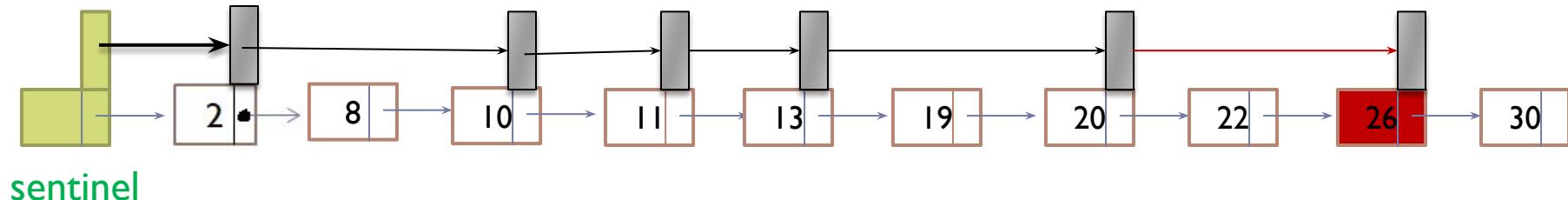


# Iteration 1(I)

Flip a coin



L1:  
L0:

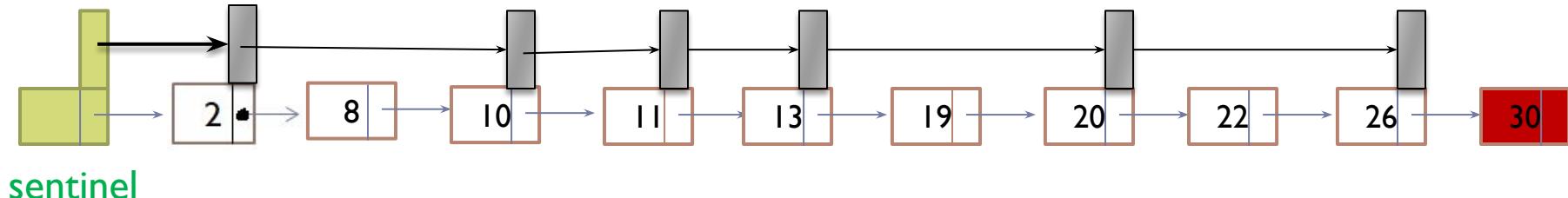


# Iteration 1(J)

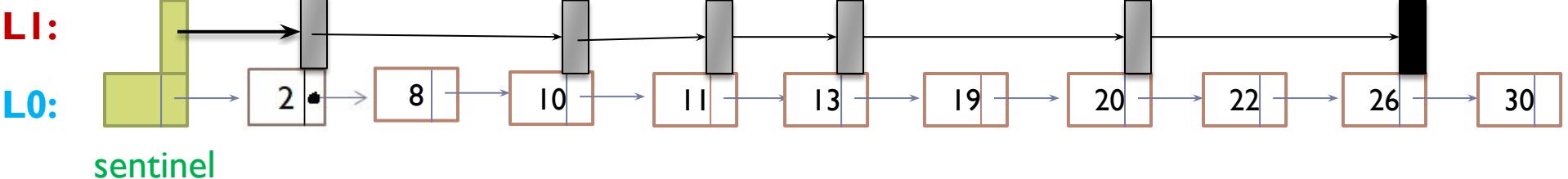
Flip a coin



L1:  
L0:



# Iteration 1(end)

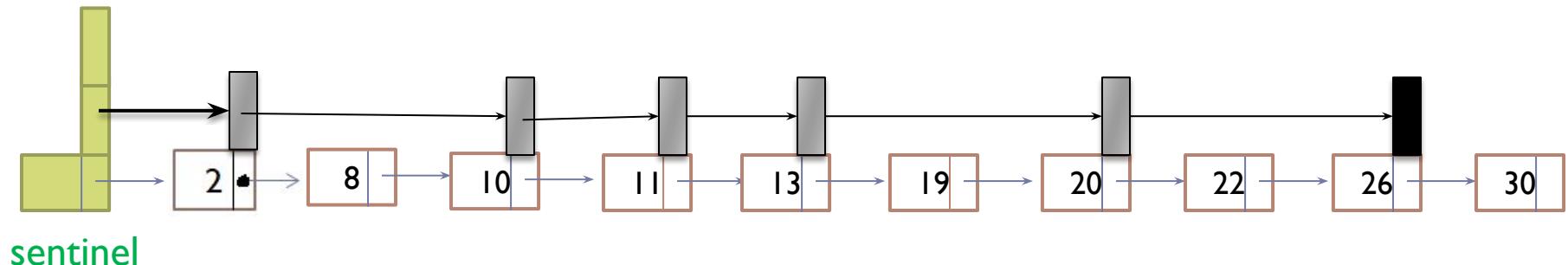


# Iteration 2

L2:

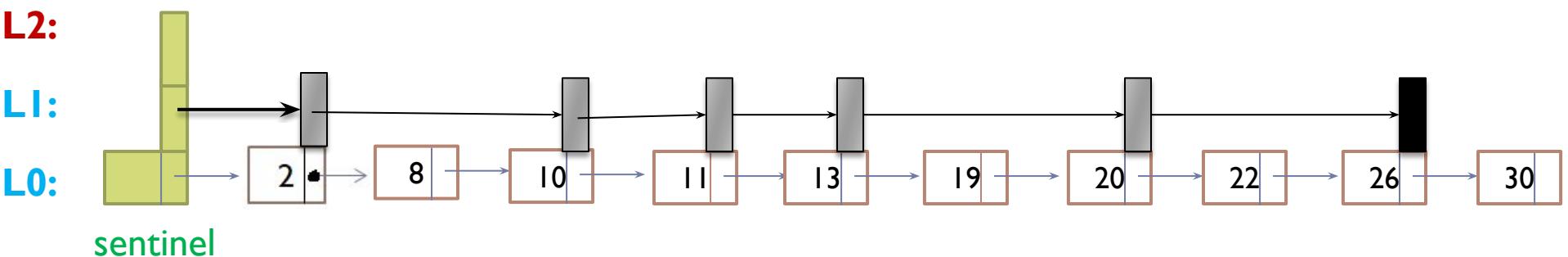
L1:

L0:



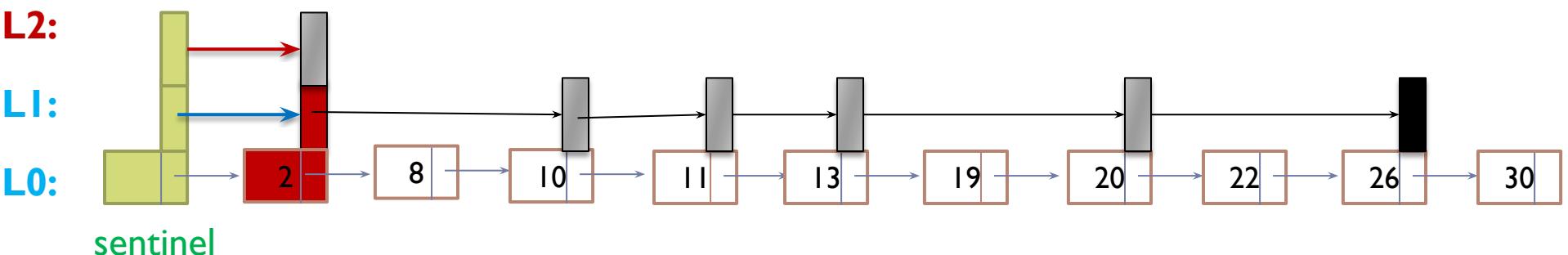
# Iteration 2

- ❑ Build L2 Linked list



# Iteration 2 (A)

Flip a coin



# Iteration 2 (B)

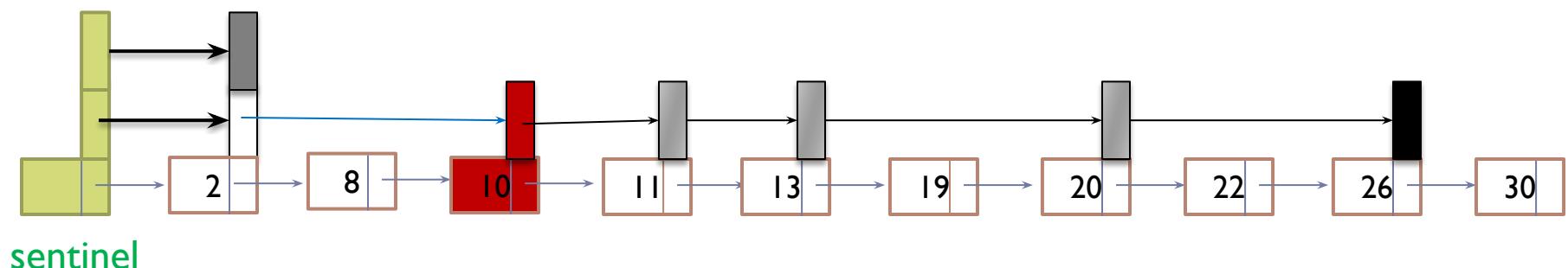
Flip a coin



L2:

L1:

L0:



# Iteration 2 (C)

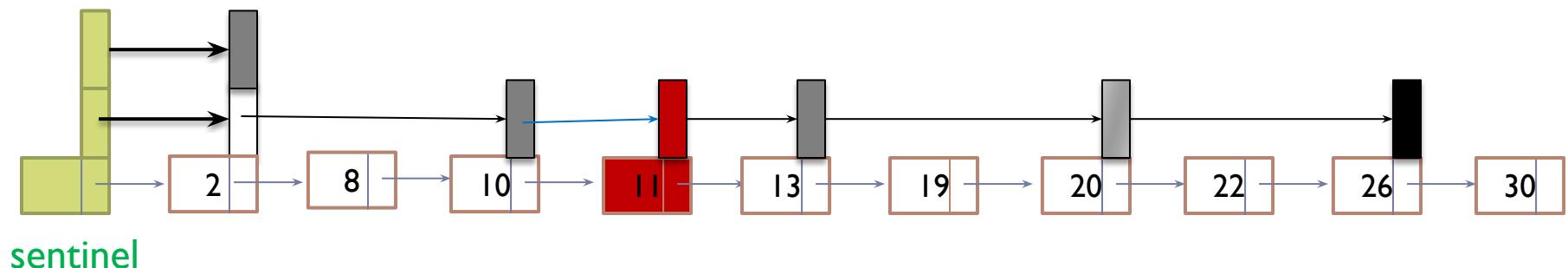
Flip a coin



L2:

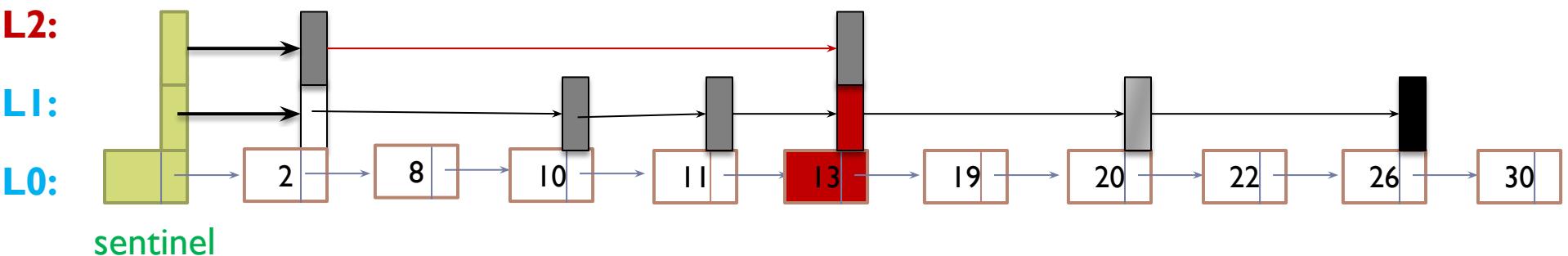
L1:

L0:



# Iteration 2 (E)

Flip a coin



# Iteration 2 (F)

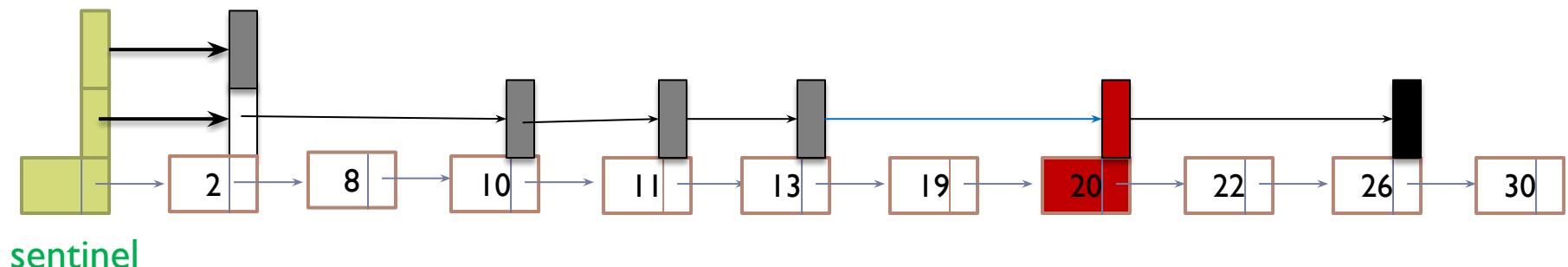
Flip a coin



L2:

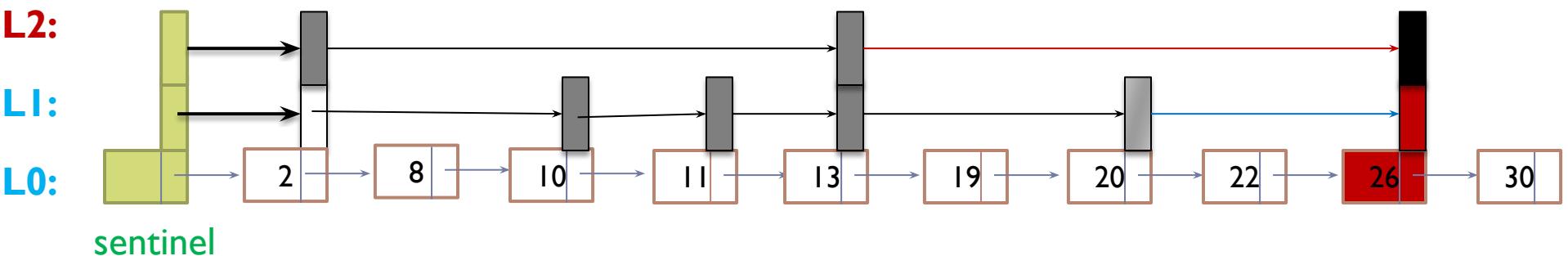
L1:

L0:

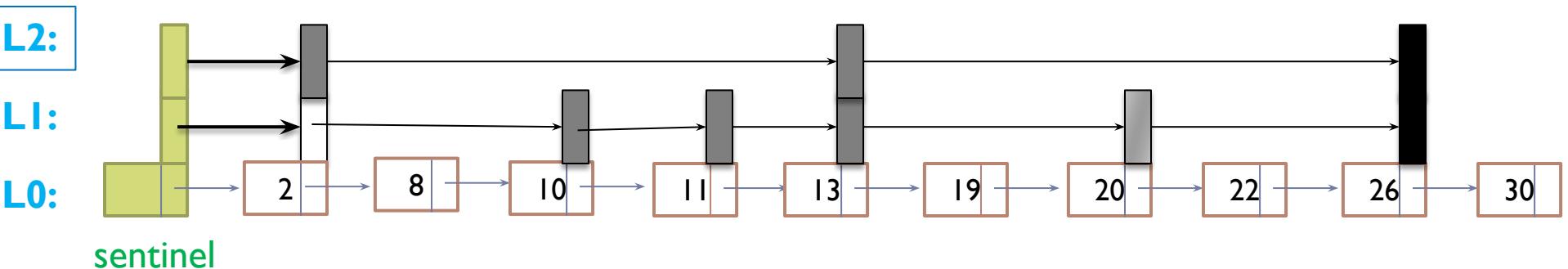


# Iteration 2 (G)

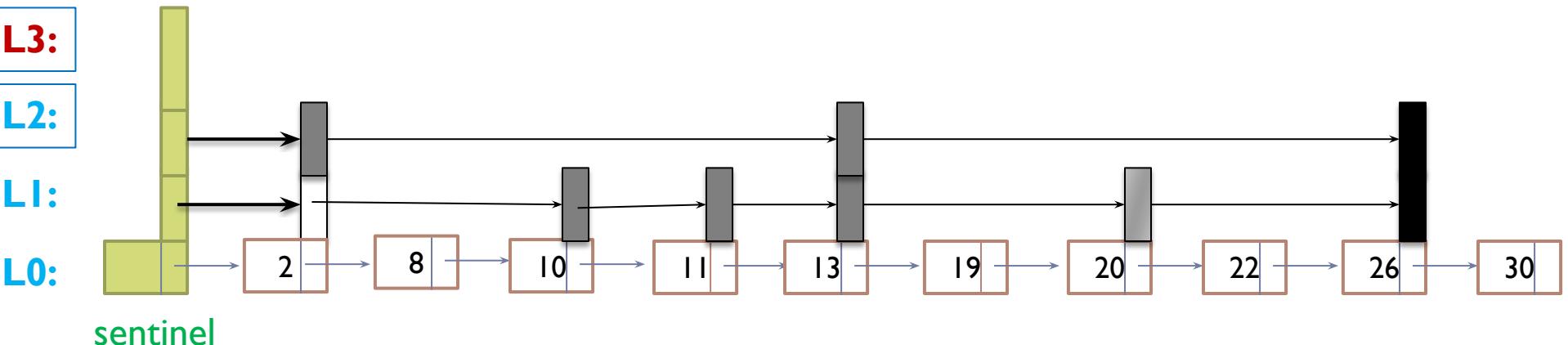
Flip a coin



# Iteration 2 (end)

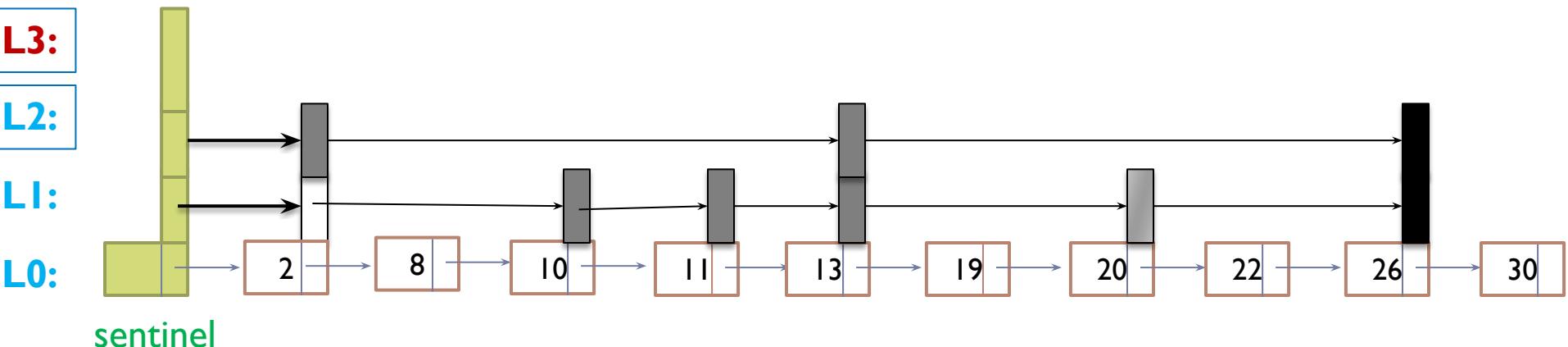


# Iteration 3



# Iteration 3

- ❑ Build L3 Linked list



# Iteration 3(A)

Flip a coin

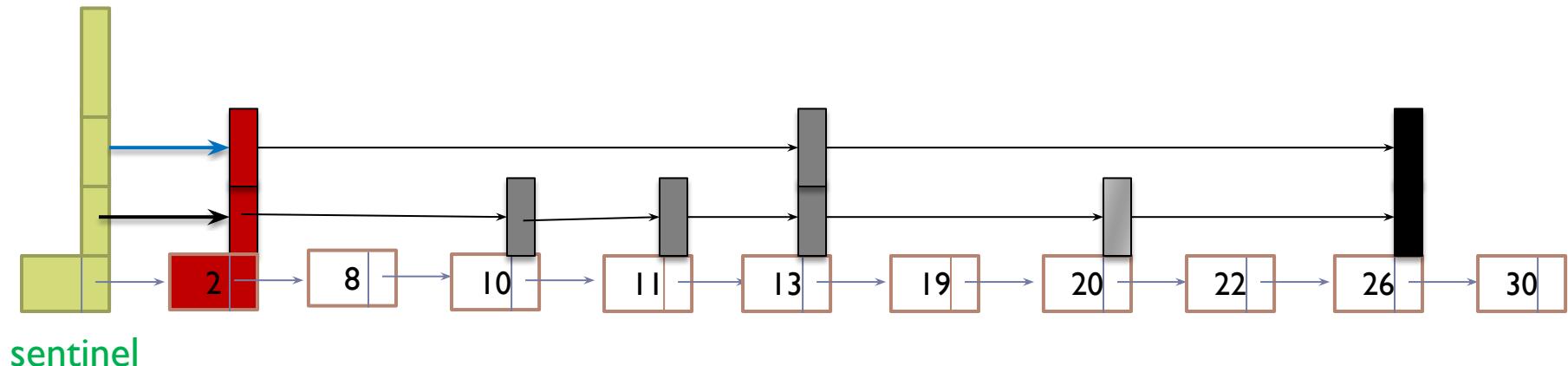


L3:

L2:

L1:

L0:

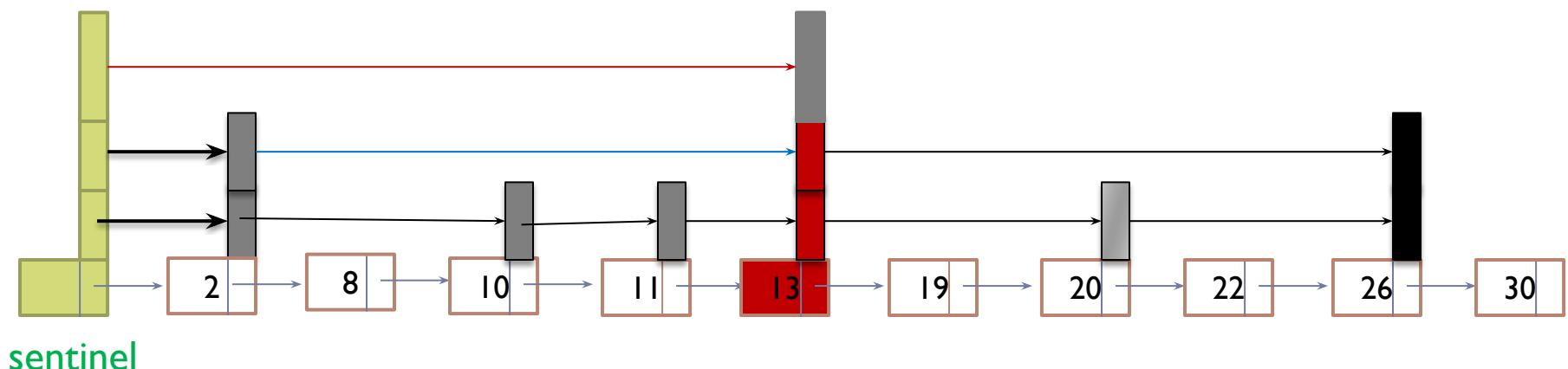


# Iteration 3(B)

Flip a coin



L3:  
L2:  
L1:  
L0:



# Iteration 3(C)

Flip a coin

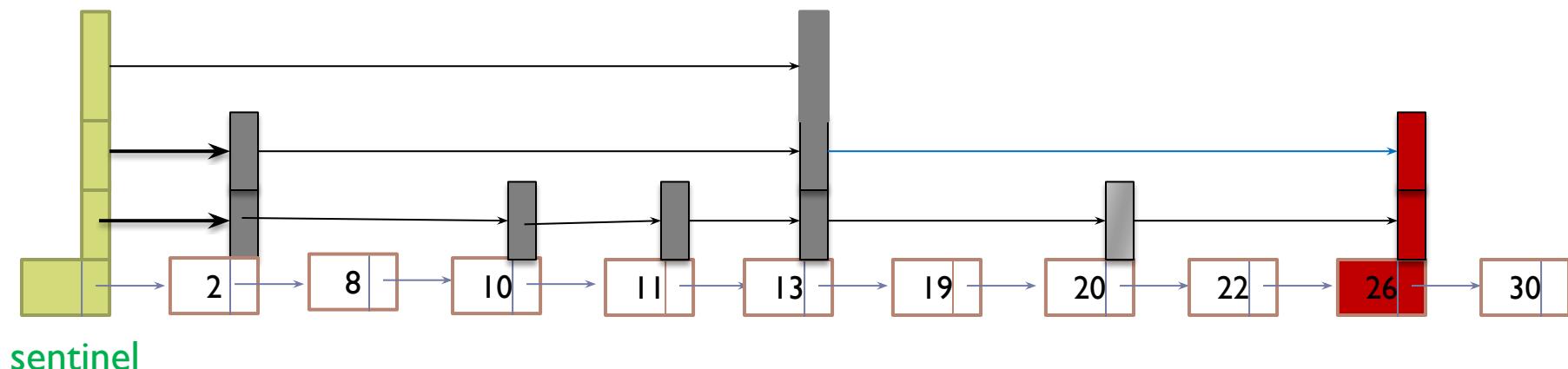


L3:

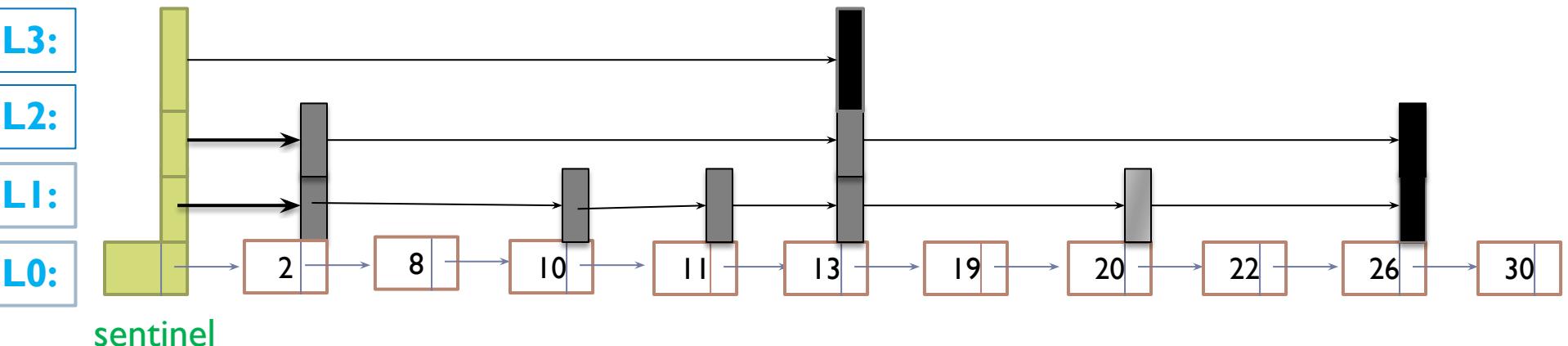
L2:

L1:

L0:

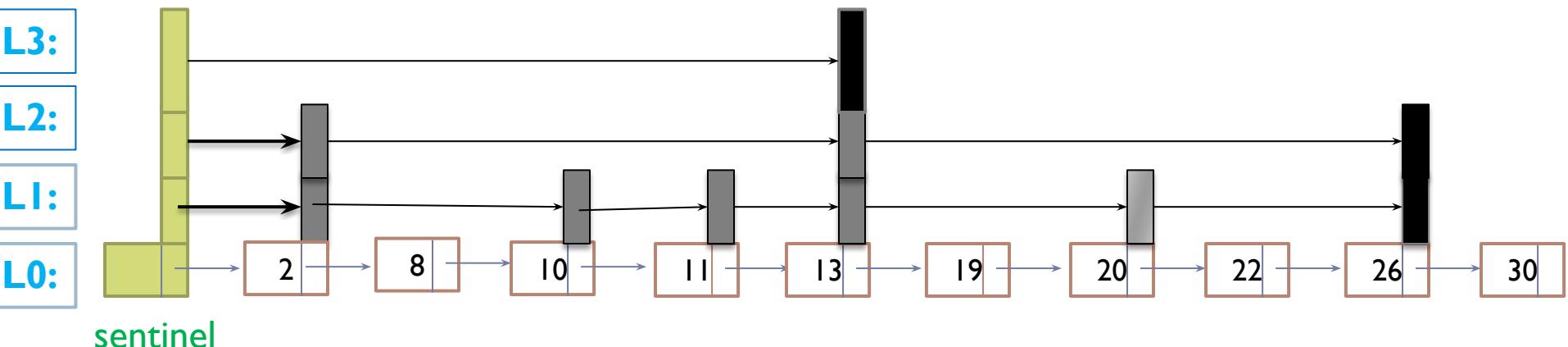


# Iteration 3(end)



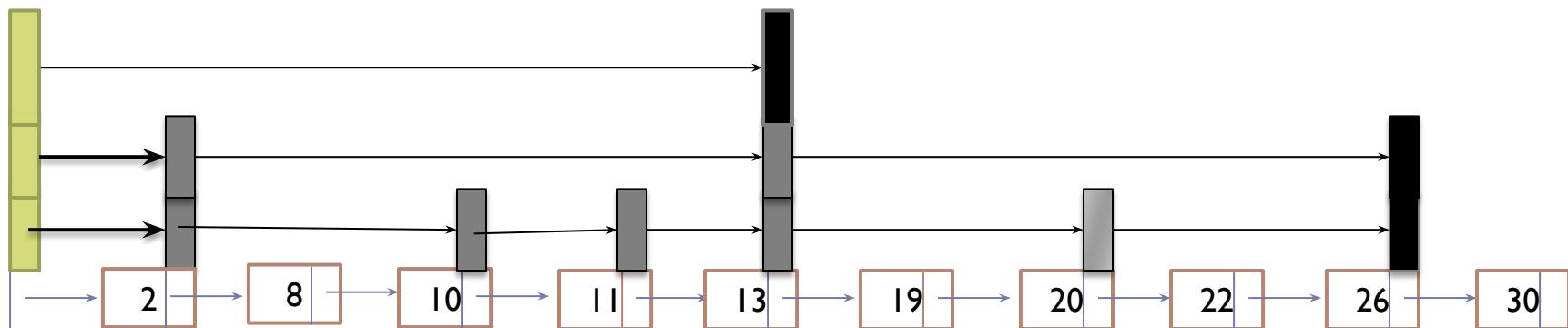
# End of Procedure

- ❑ The number of layers is up to the user; we used 4 layers
- ❑ The number of layers should be  $\log n$ .



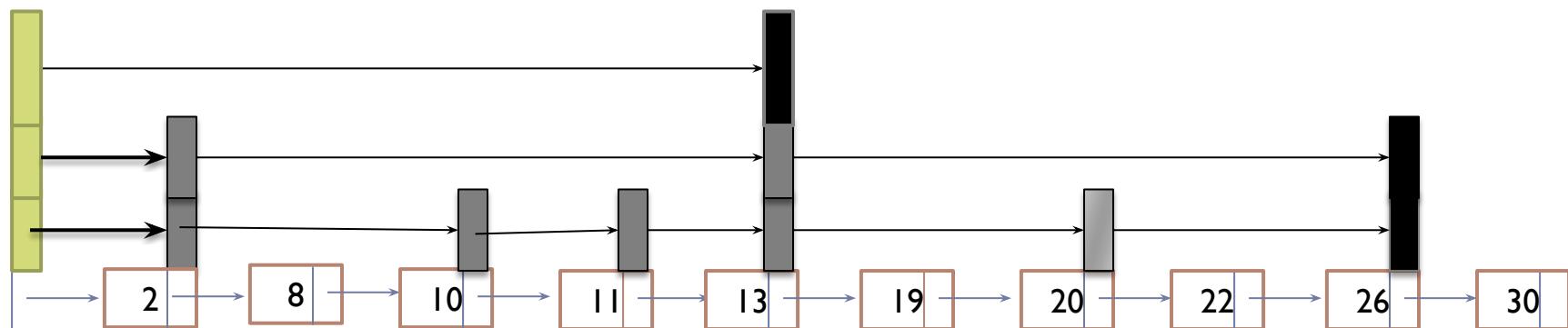
# Perfect Skip List

- ❑ Keys in sorted order.
- ❑  $O(\log n)$  *levels*
- ❑ Each higher level contains  $1/2$  the elements of the level below it.
- ❑ Header & sentinel nodes are in every level



# Perfect Skip List Continued

- ❑ Nodes are of variable size:
  - contain between 1 and  $O(\log n)$  pointers
- ❑ Pointers point to the start of each node  
(picture draws pointers horizontally for visual clarity)
- ❑ Called *skip lists* because higher level lists let you skip over many items





---

# Search

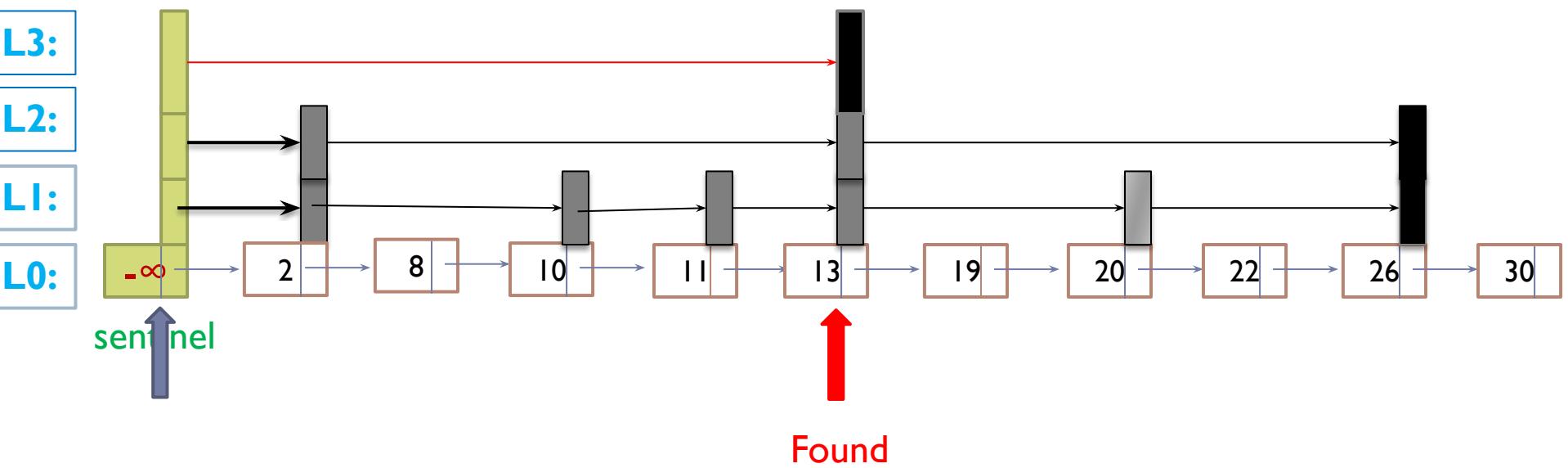


# Search

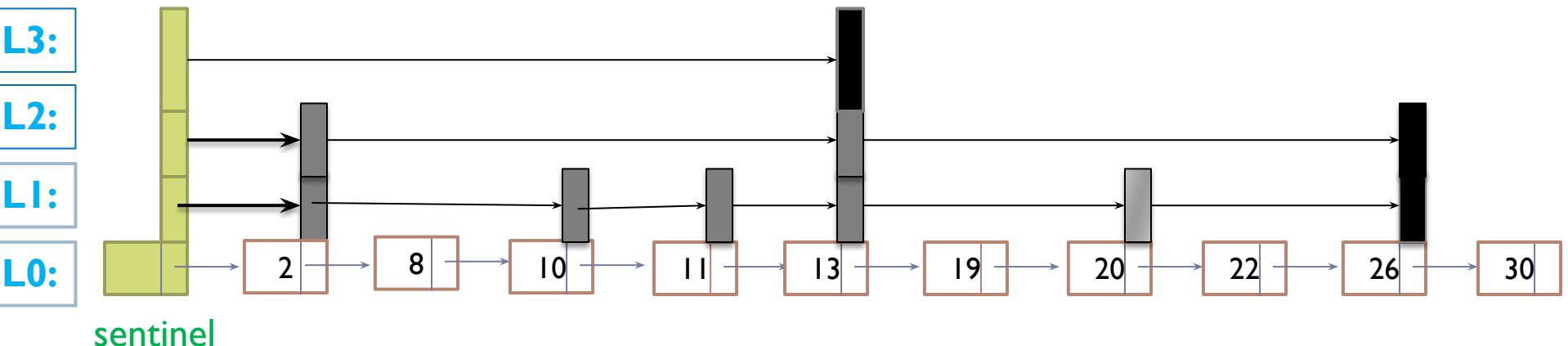
---

- ❑ When search for  $k$ :
  - If  $k = \text{key}$ , done!
  - If  $k < \text{next key}$ , go down a level
  - If  $k \geq \text{next key}$ , go right

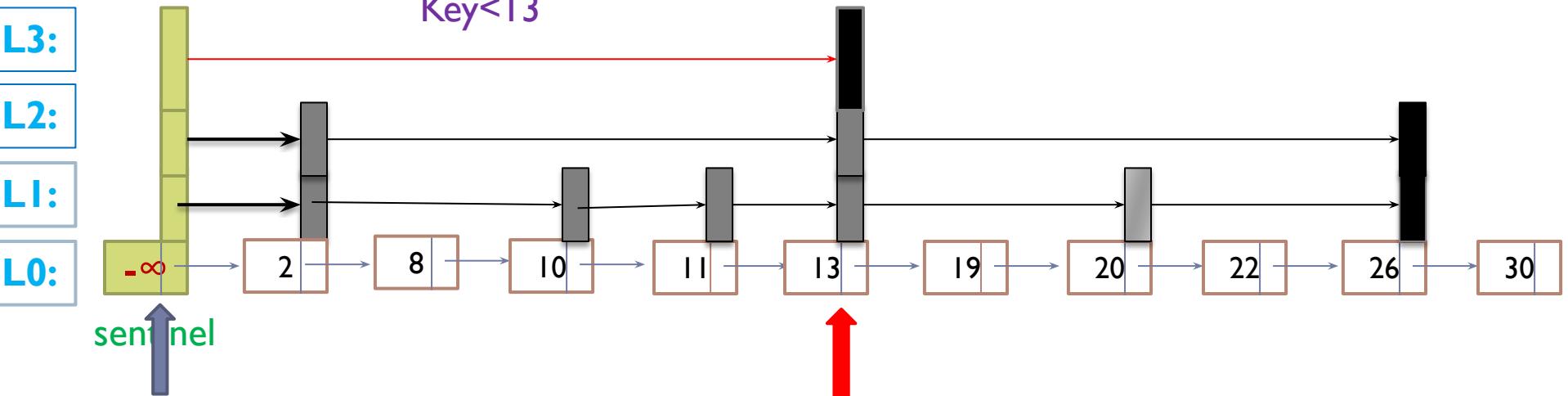
# Search: key=13



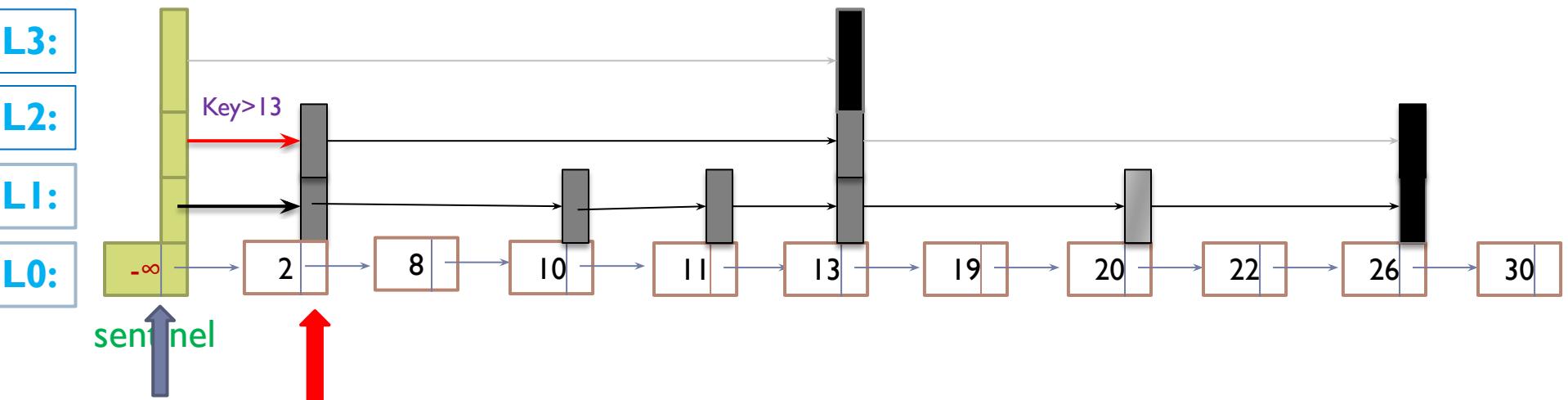
# Search: key=8



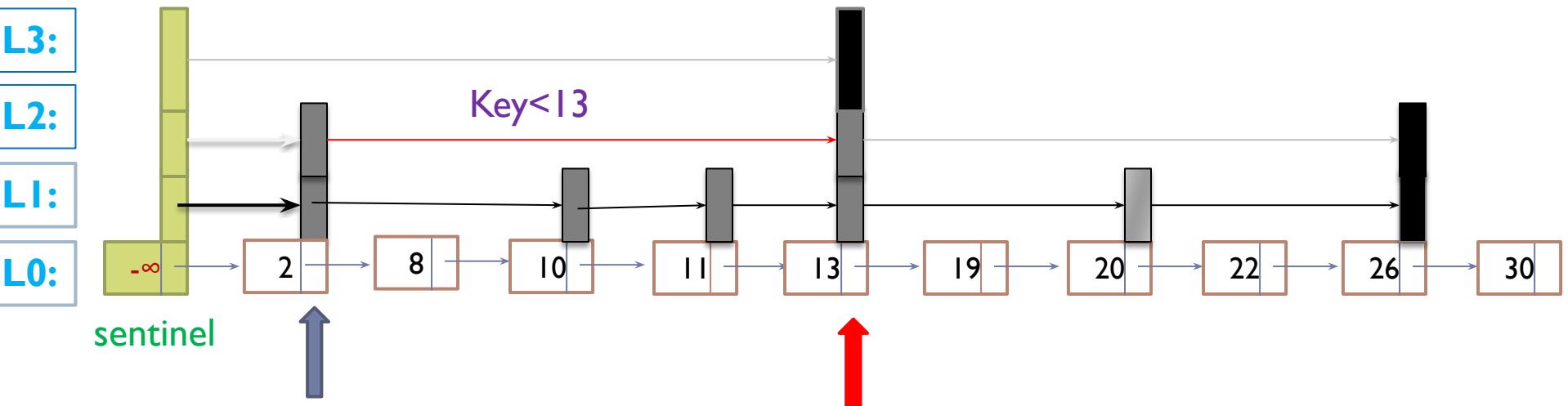
# Search: key=8



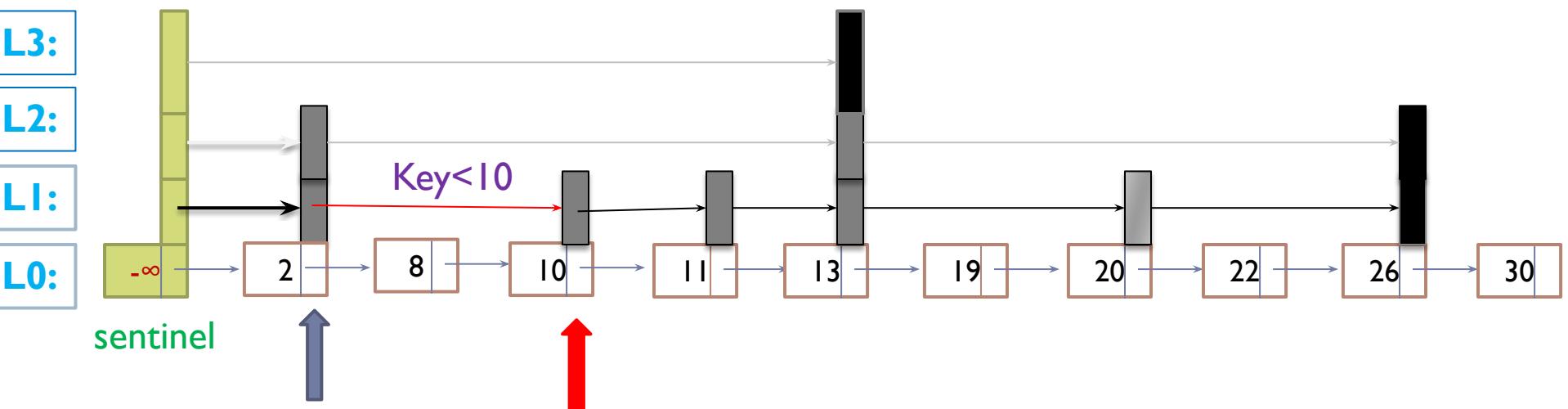
# Search: key=8



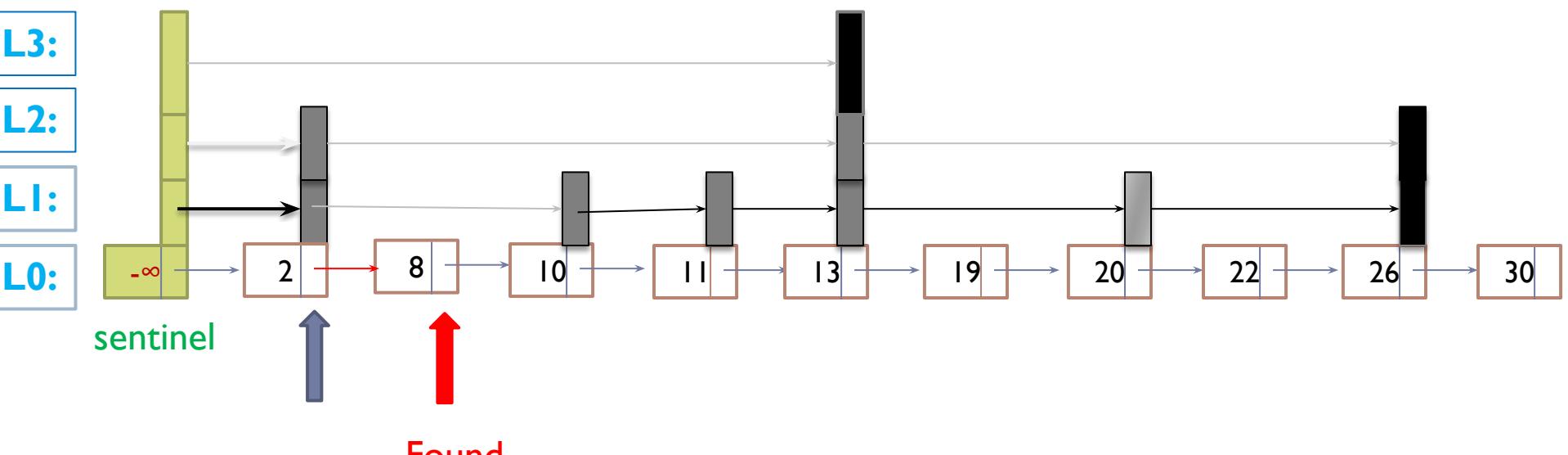
# Search: key=8



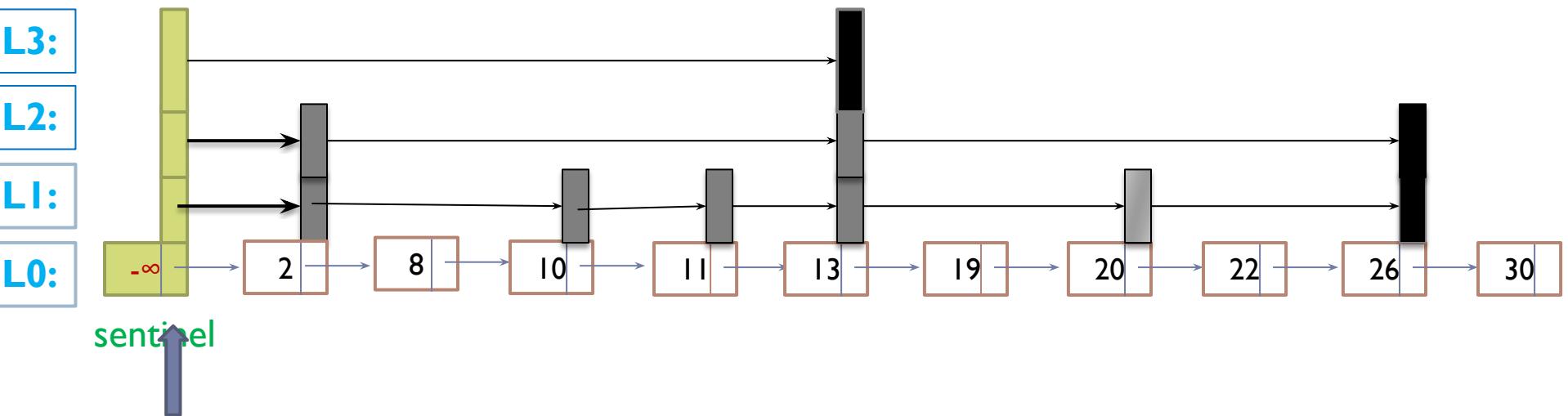
# Search: key=8



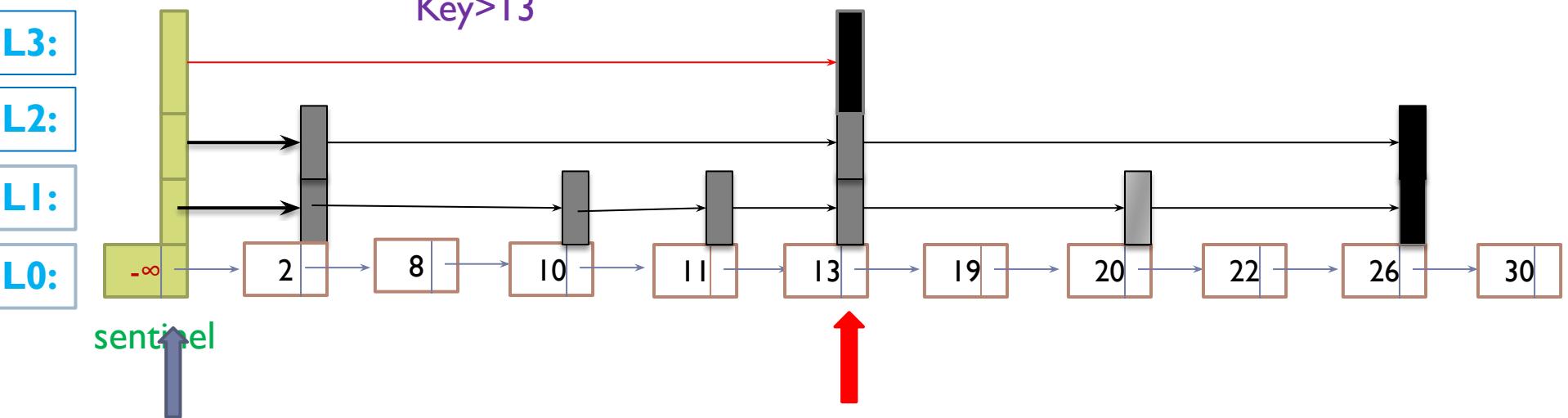
# Search: key=8



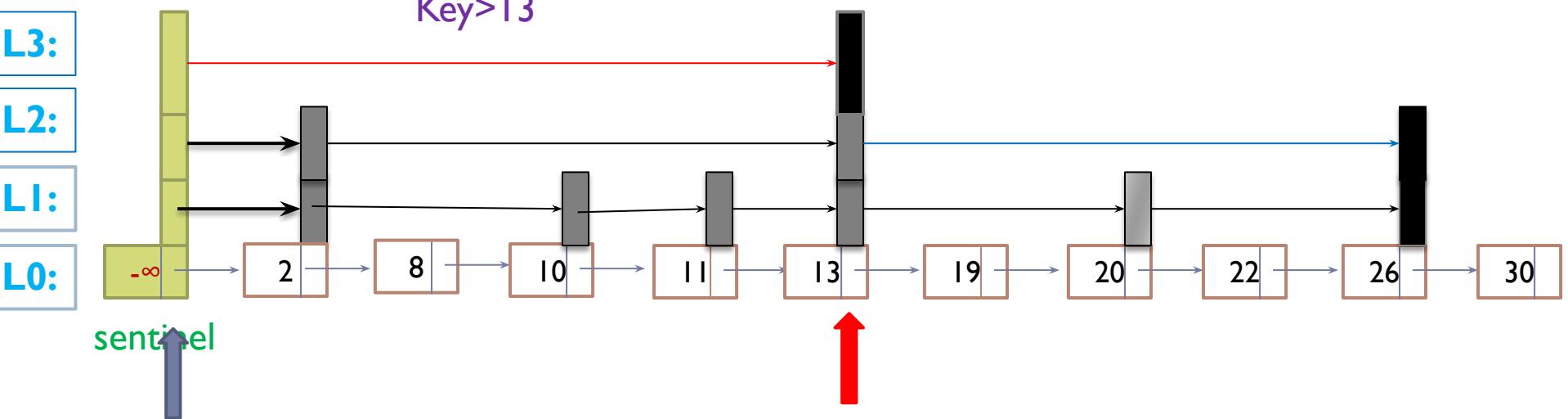
# Search: key=20



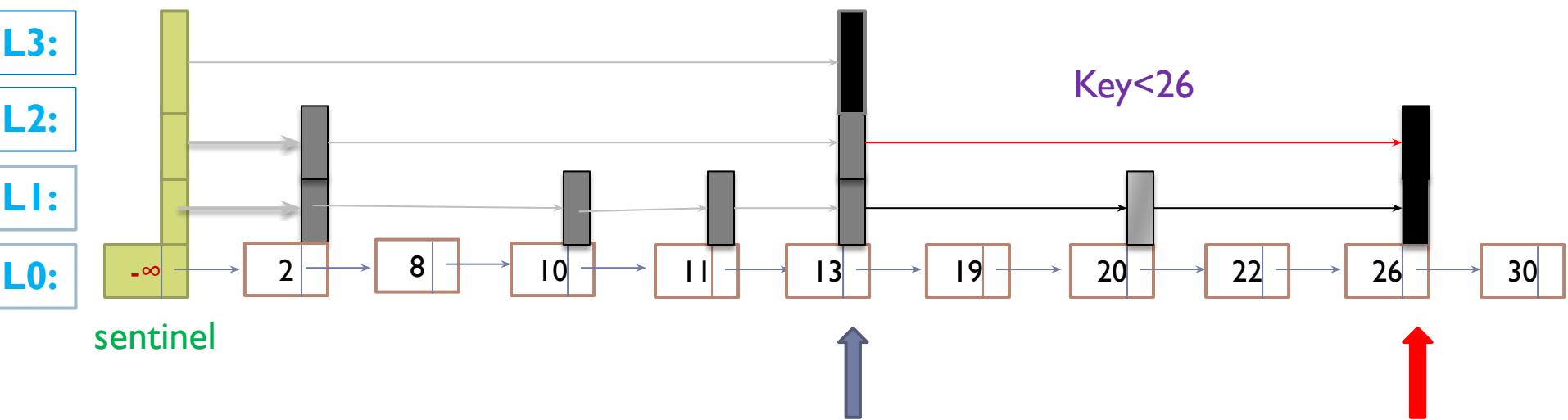
# Search: key=20



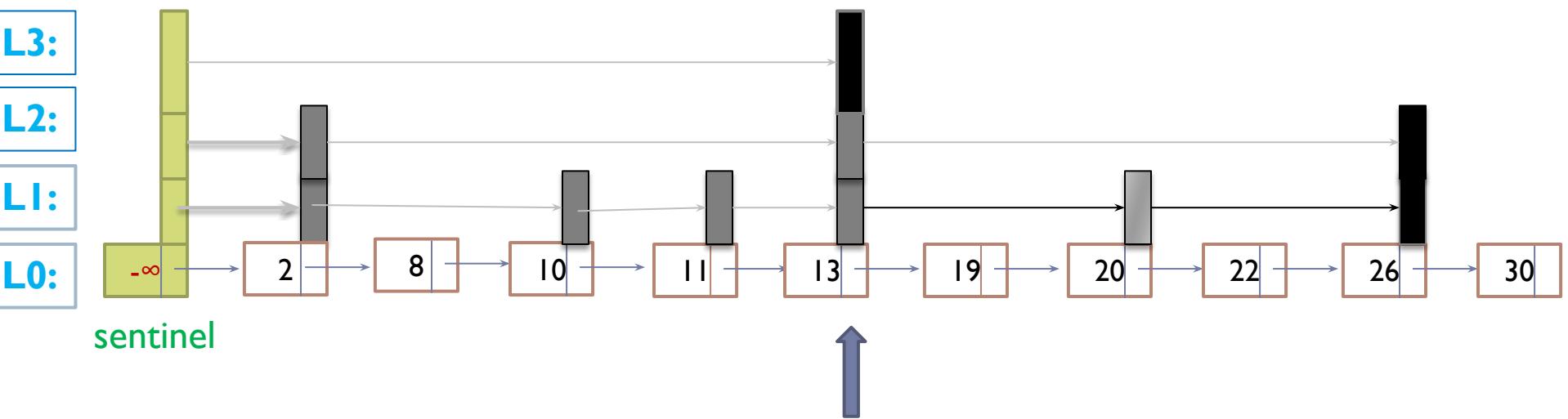
# Search: key=20



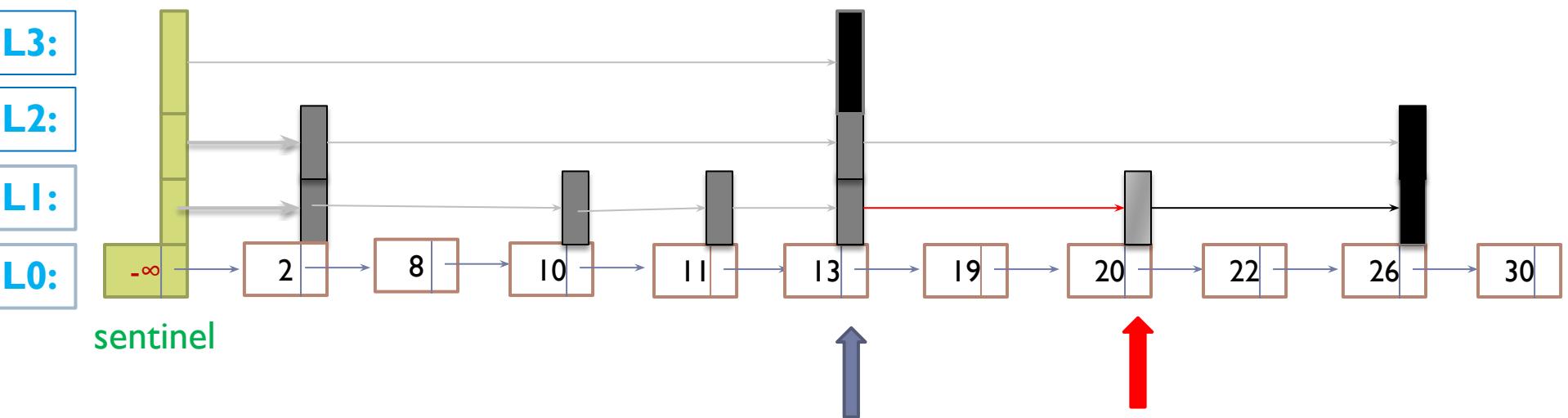
# Search: key=20



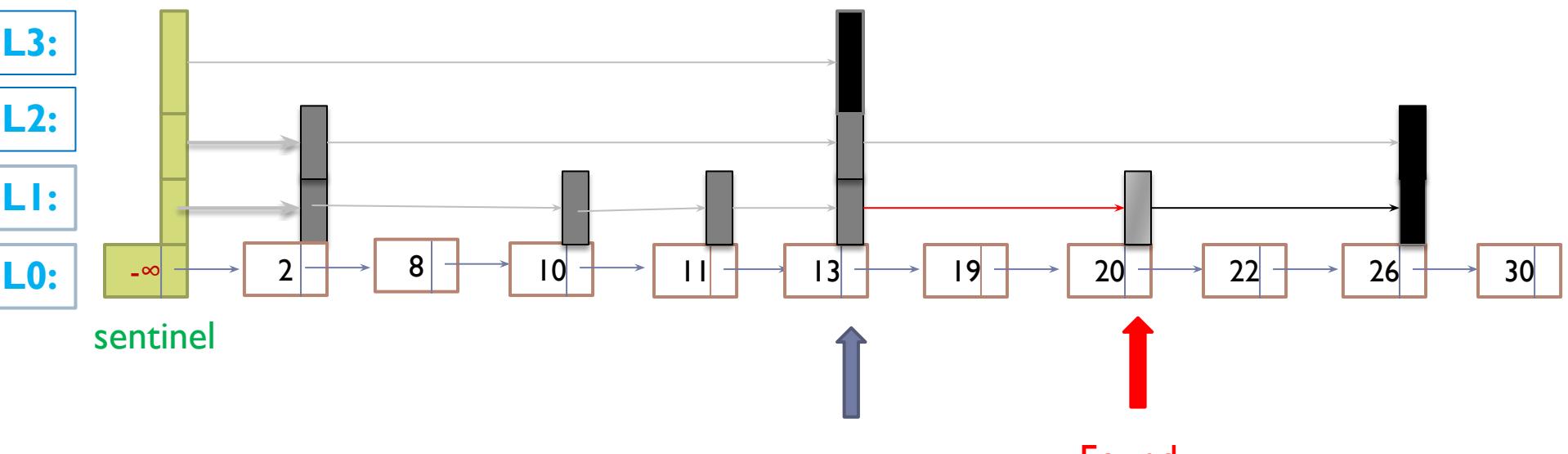
# Search: key=20



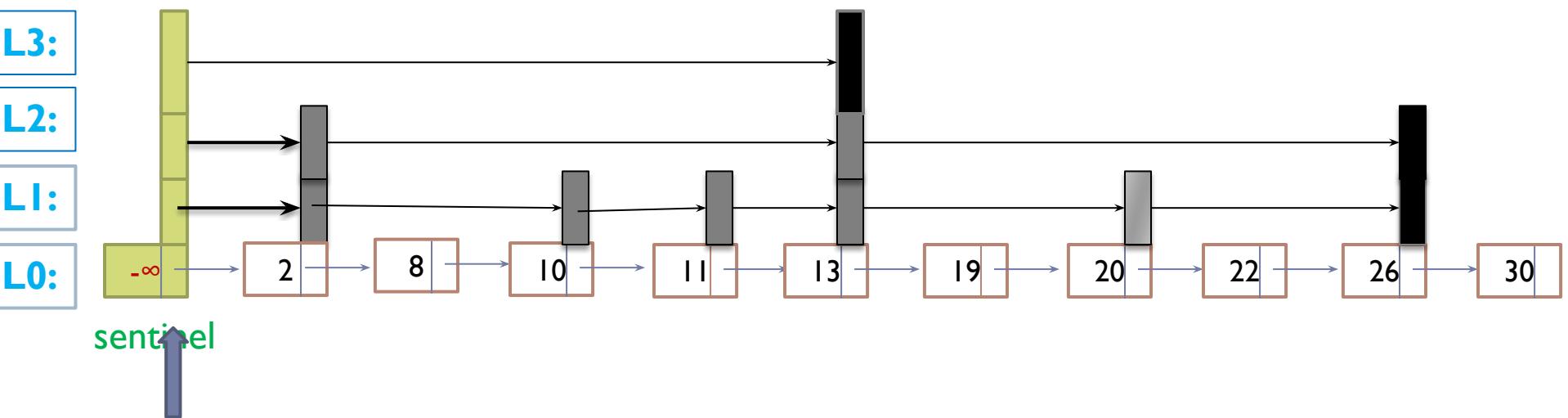
# Search: key=20



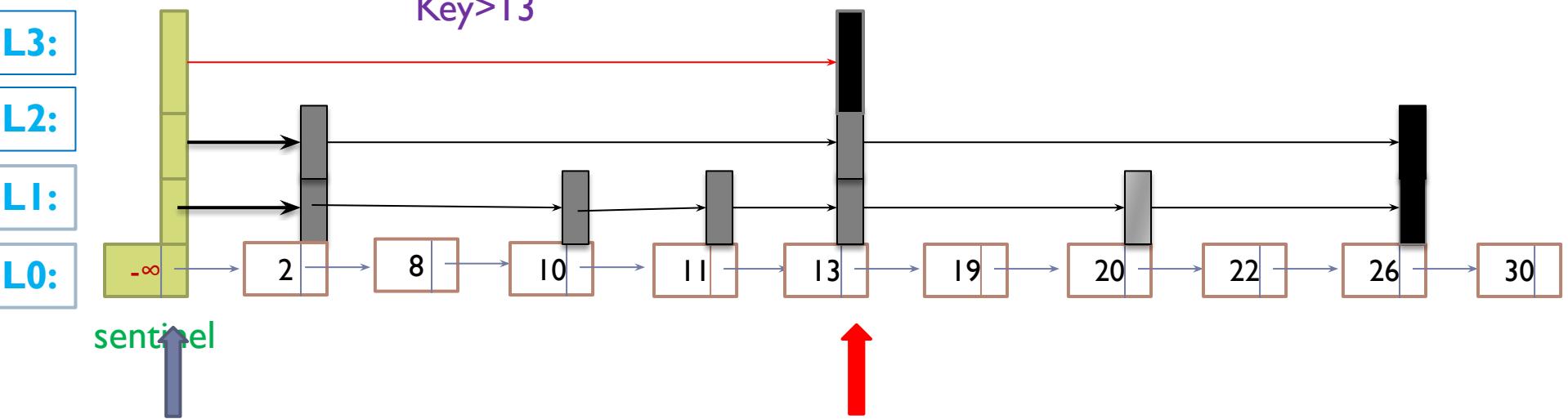
# Search: key=20



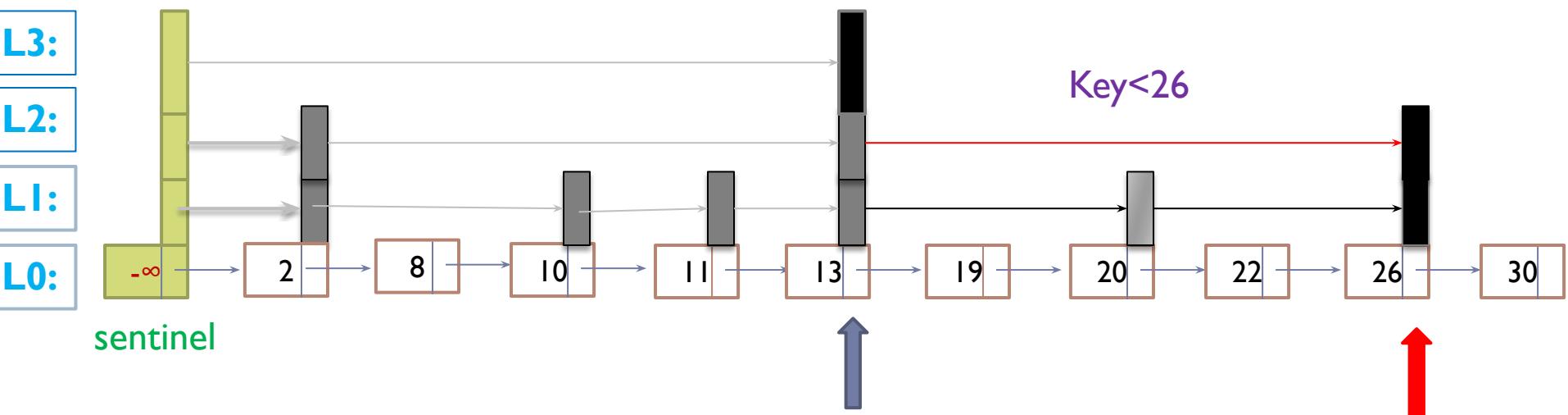
# Search: key=21



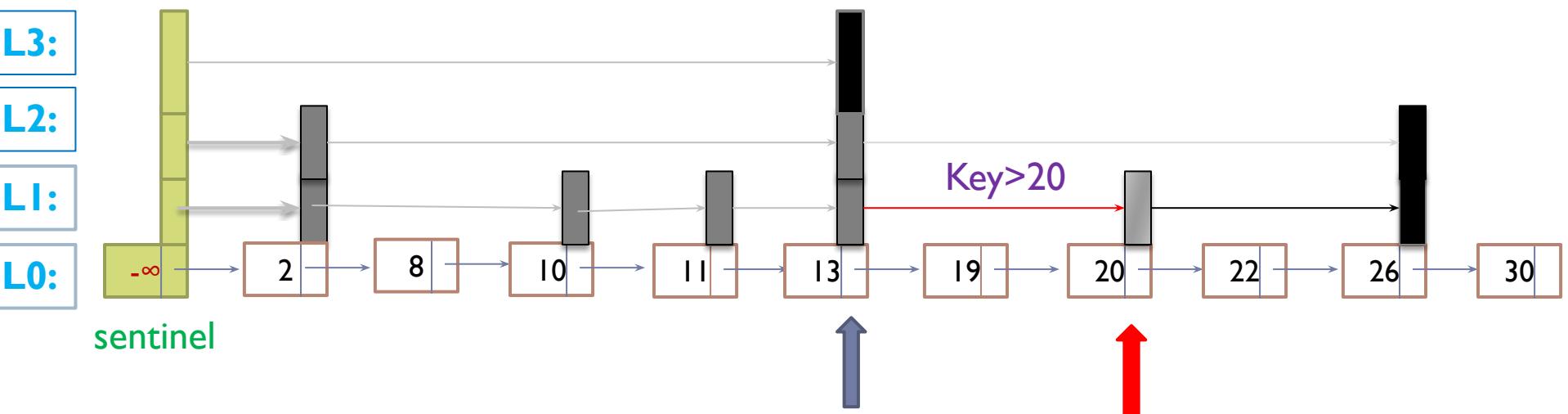
# Search: key=21



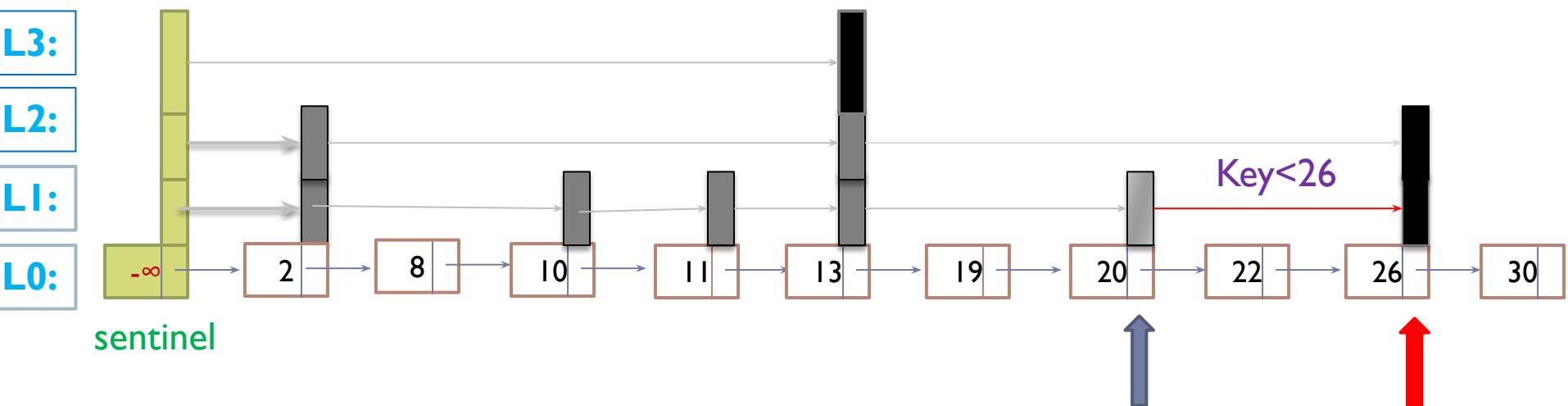
# Search: key=21



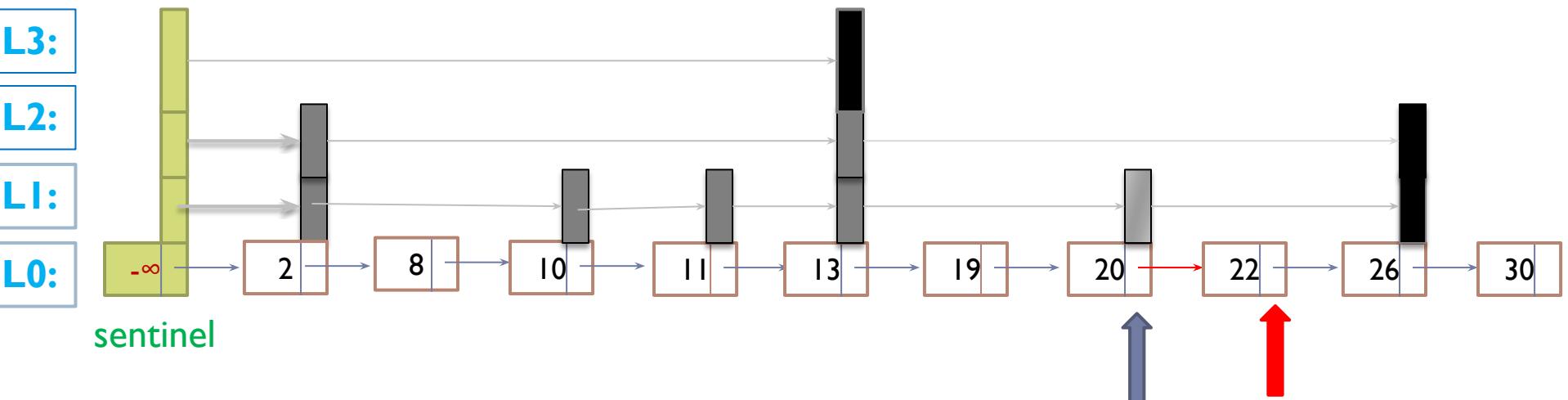
# Search: key=21



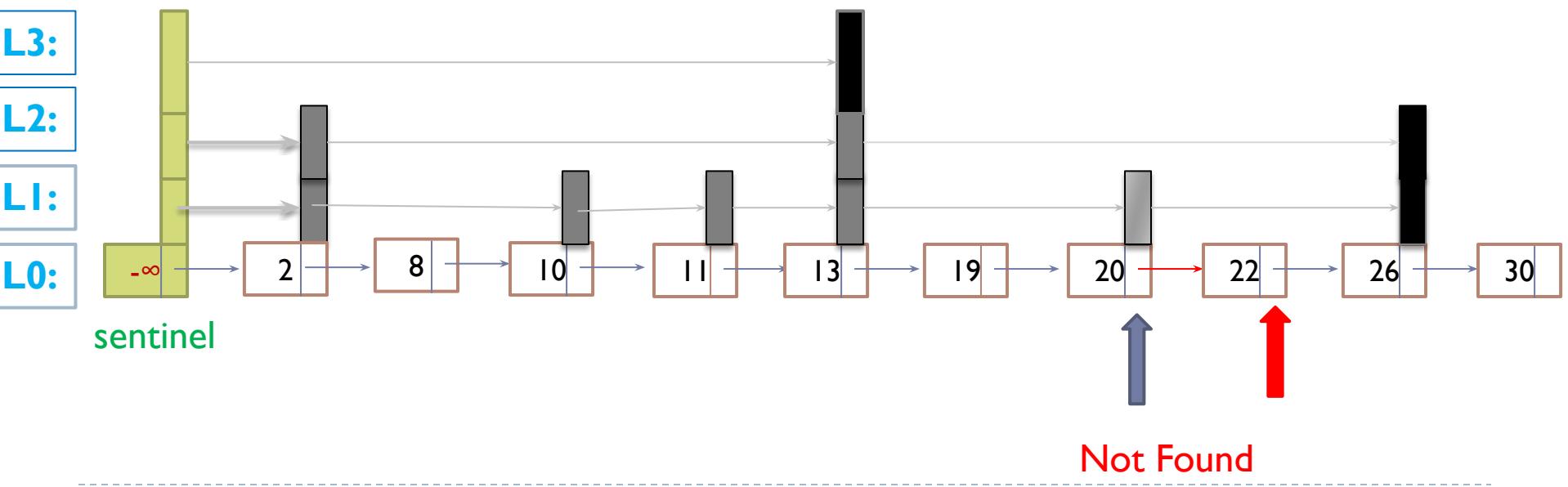
# Search: key=21



# Search: key=21



# Search: key=21





# Search Time

- ❑  $O(\log n)$  levels --- because you cut the # of items in half at each level
- ❑ Will visit at most 2 nodes per level:
  - ❑ If you visit more, then you could have done it on one level higher up.
- ❑ Therefore, search time is  $O(\log n)$ .

# Insert & Delete

- ❑ Insert & delete might need to rearrange the entire list
- ❑ Like Perfect Binary Search Trees, Perfect Skip Lists are *too* structured to support efficient updates.
- ❑ Idea:
  - ❑ - Relax the requirement that each level have exactly half the items of the previous level
  - ❑ - Instead: design structure so that we *expect* 1/2 the items to be carried up to the next level
  - ❑ - Skip Lists are a *randomized* data structure: the same sequence of inserts / deletes may produce different structures depending on the outcome of random coin flips.



# Randomization

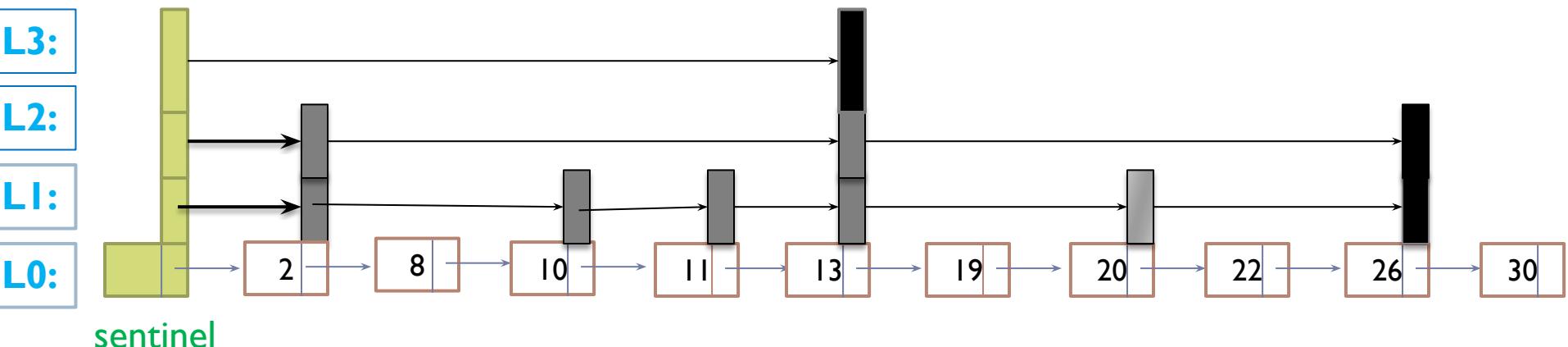
- ❑ Expected behavior (over the random choices) remains the same as with perfect skip lists.
- ❑ Idea: Each node is promoted to the next higher level with probability  $1/2$ 
  - Expect  $1/2$  the nodes at level 1
  - Expect  $1/4$  the nodes at level 2
  - ... \
- ❑ Therefore, expect # of nodes at each level is the same as
  - ❑ with perfect skip lists.
  - ❑ Also: expect the promoted nodes will be well distributed across the list



# Insertion

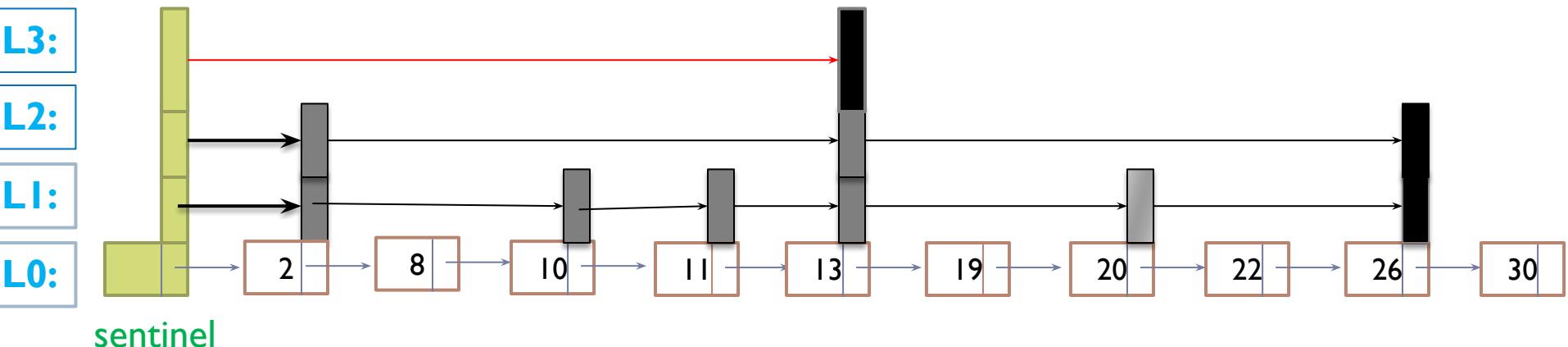
# Insert: key=9

- First search key 9 and record the path
- Start from Level 3 and step down to Level 0



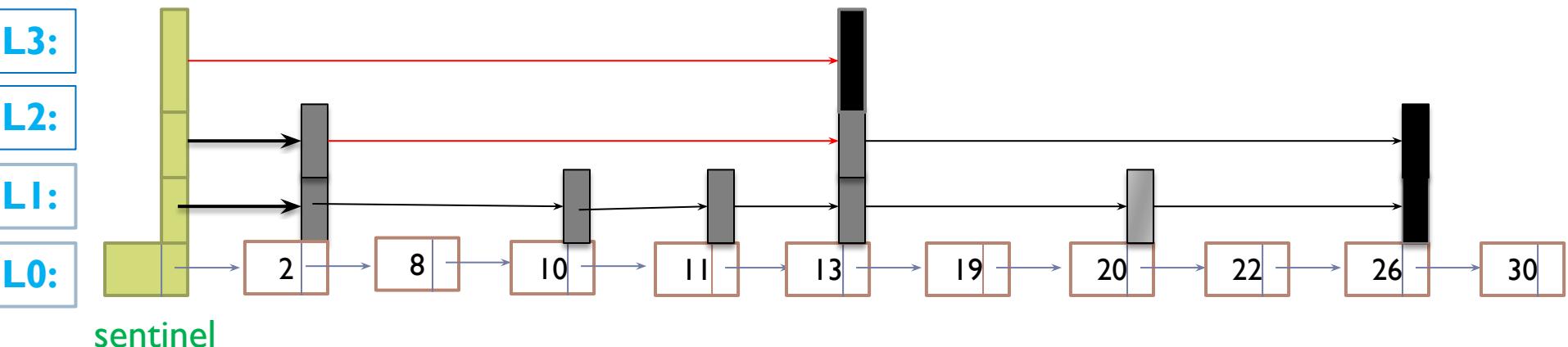
# Insert: key=9

- ❑ First search key 9 and record the path



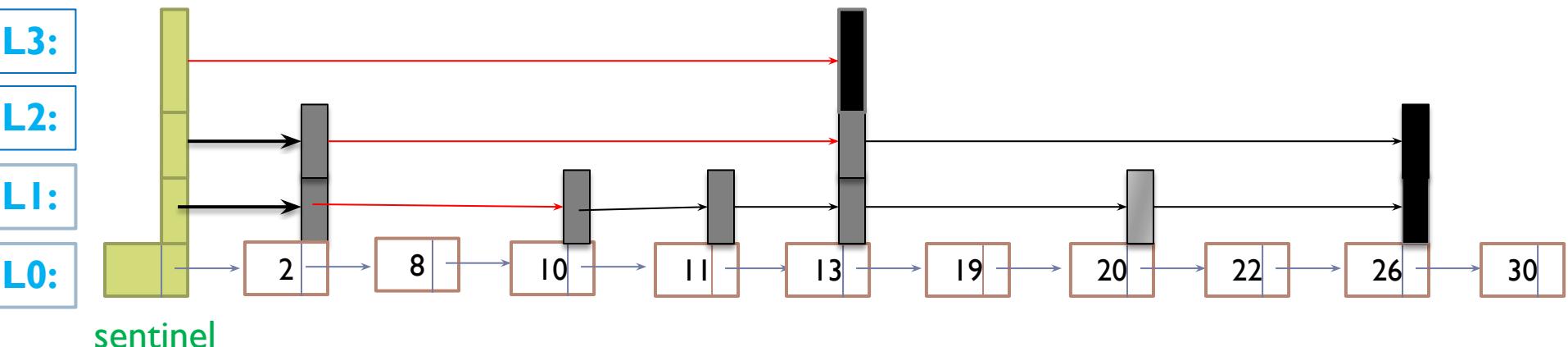
# Insert: key=9

- ❑ First search key 9 and record the path
- ❑ Start from Level 3 and step down to Level 0



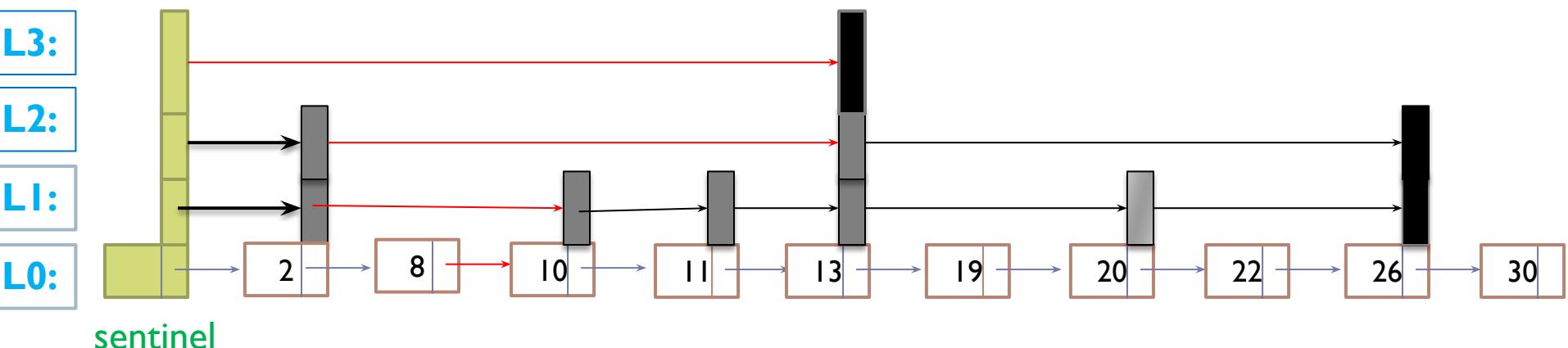
# Insert: key=9

- ❑ First search key 9 and record the path
- ❑ Start from Level 3 and step down to Level 0



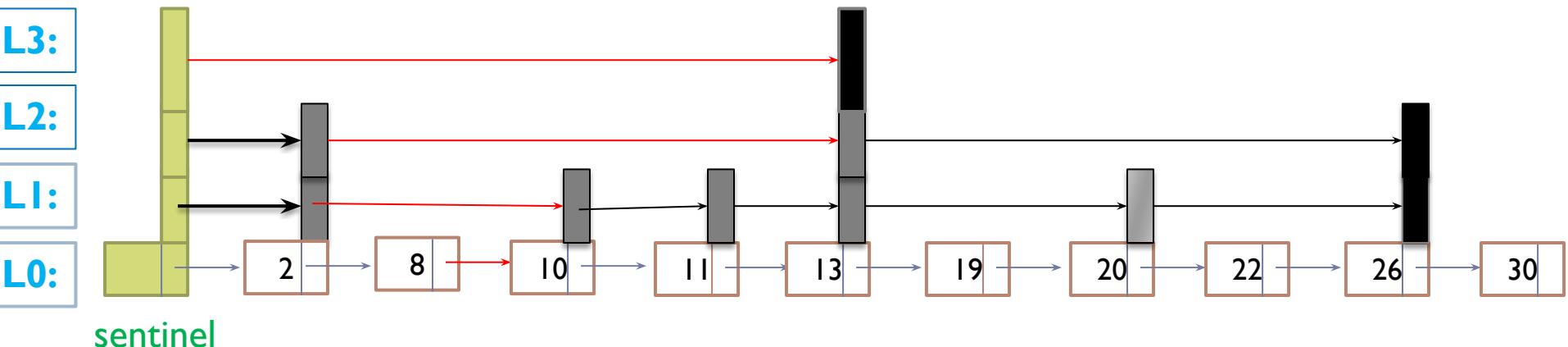
# Insert: key=9

- ❑ First search key 9 and record the path
- ❑ Start from Level 3 and step down to Level 0



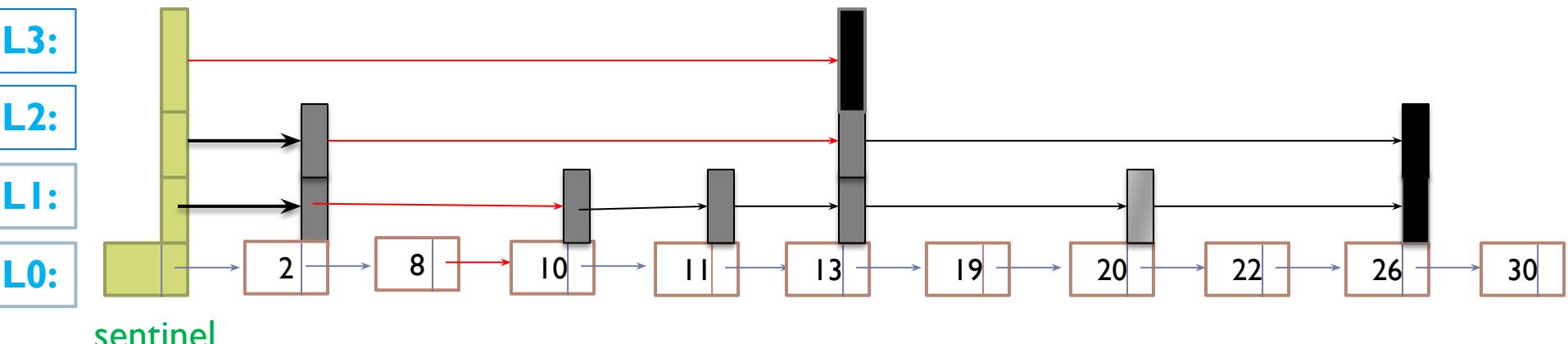
# Insert: key=9

- ❑ First search key 9 and record the path
- ❑ Start from Level 3 and step down to Level 0
- ❑ Red arrow means the key is within the interval



# Insert: key=9

- First search key 9 and record the path
- Start from Level 3 and step down to Level 0
- Red arrow means the key is within the interval
- Use a table to record the two ends of red arrow.

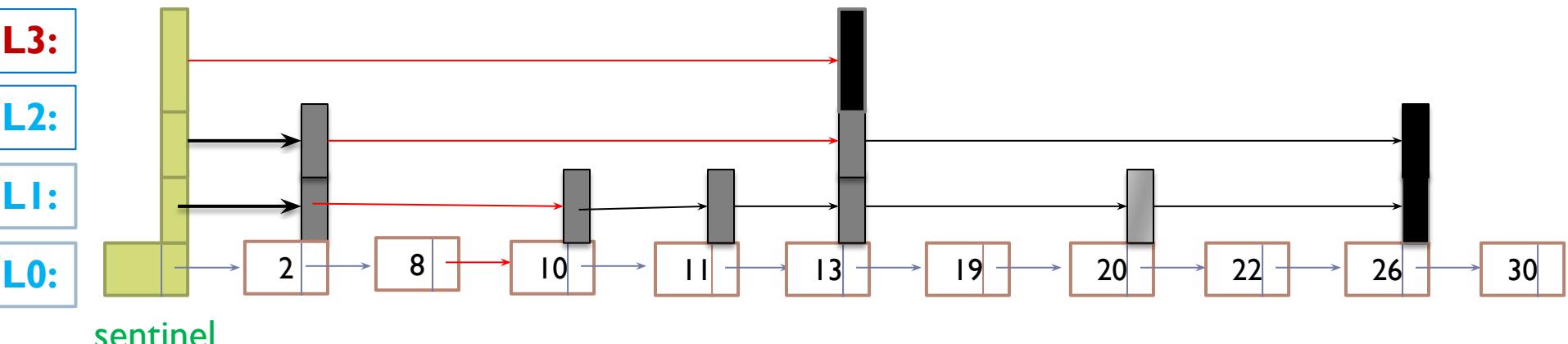


Start From

L3	
L2	
L1	
L0	

# Insert: key=9

- First search key 9 and record the path
- Start from Level 3 and step down to Level 0
- Red arrow means the key is within the interval
- Use a table to record the two ends of red arrow.

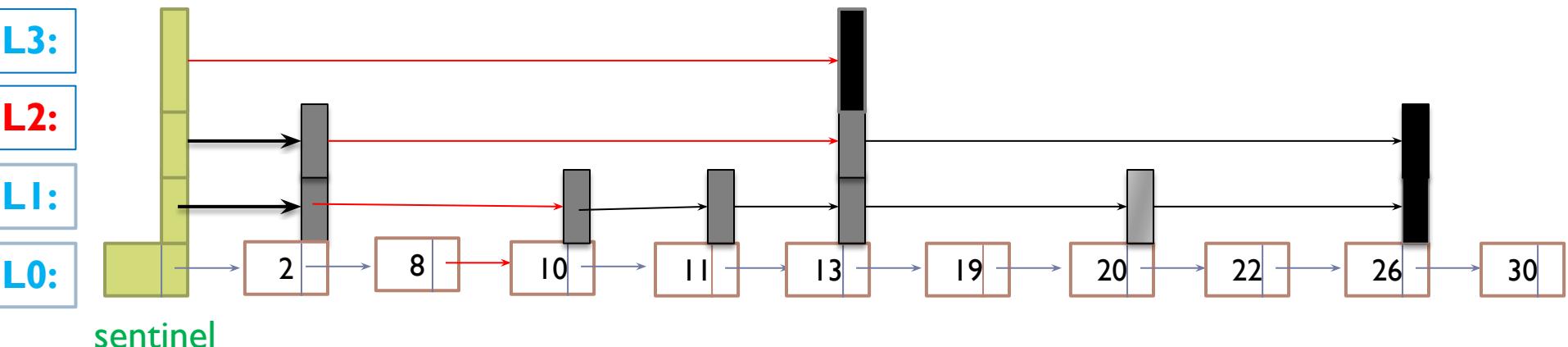


Start From

L3	sentinel
L2	
L1	
L0	

# Insert: key=9

- First search key 9 and record the path
- Start from Level 3 and step down to Level 0
- Red arrow means the key is within the interval
- Use a table to record the two ends of red arrow.

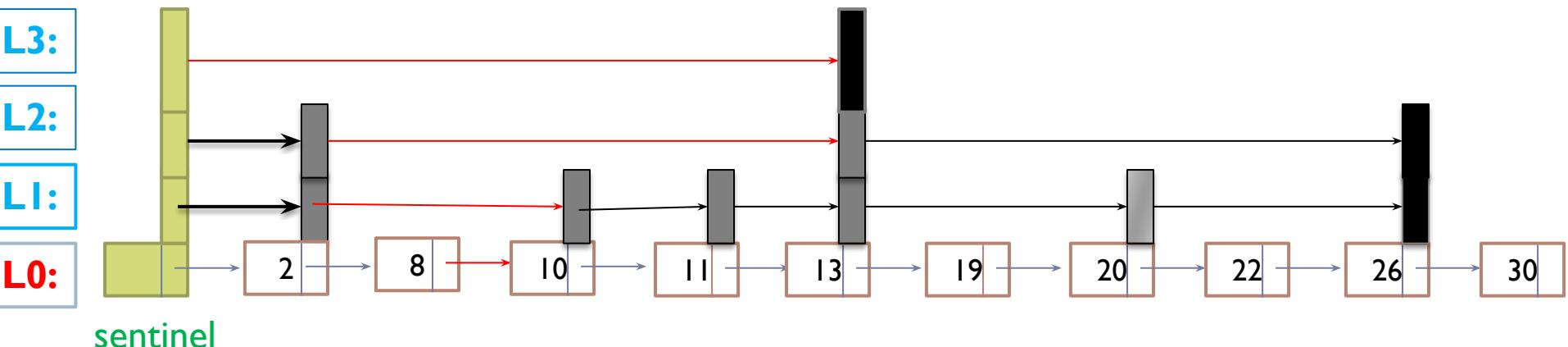


Start From

L3	sentinel
L2	Node 2
L1	
L0	

# Insert: key=9

- First search key 9 and record the path
- Start from Level 3 and step down to Level 0
- Red arrow means the key is within the interval
- Use a table to record the two ends of red arrow.

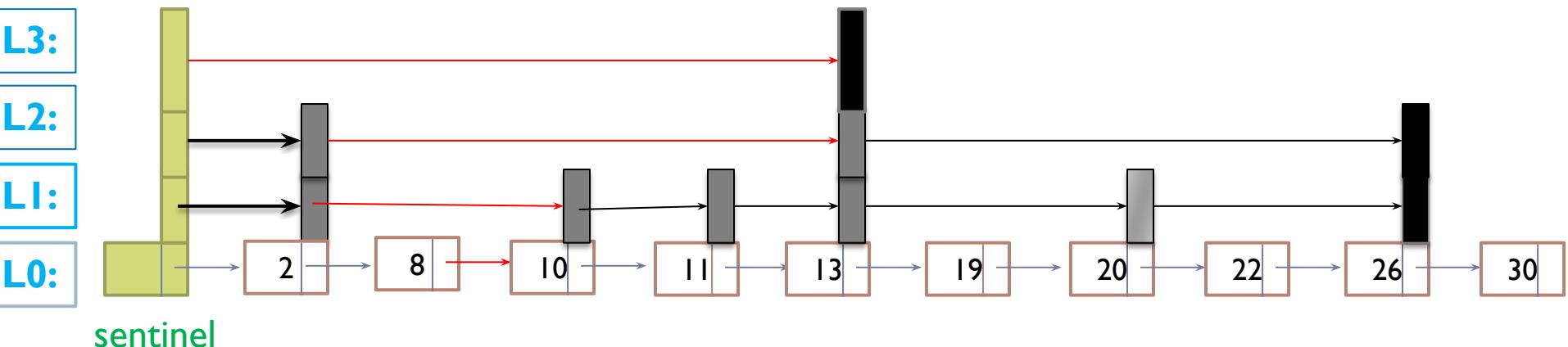


Start From

L3	sentinel
L2	Node 2
L1	Node 2
L0	Node 8

# Insert: key=9

- Second, create a node whose level is random e.g. height=2



Create a node:

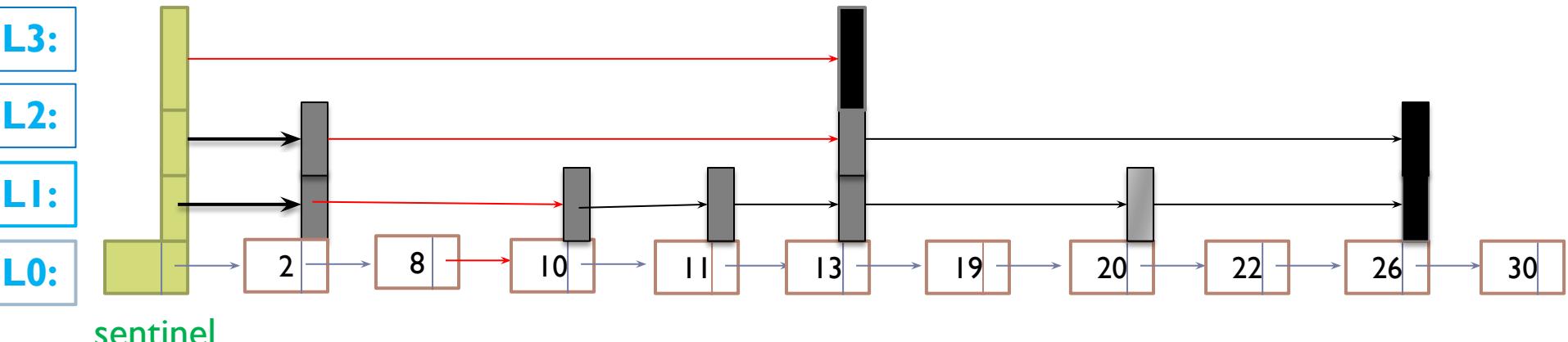


Start From

L3	sentinel
L2	Node 2
L1	Node 2
L0	Node 8

# Insert: key=9

- ❑ Second, create a node whose level is random e.g. height=2
- ❑ Initially the height is 0



Create a node:



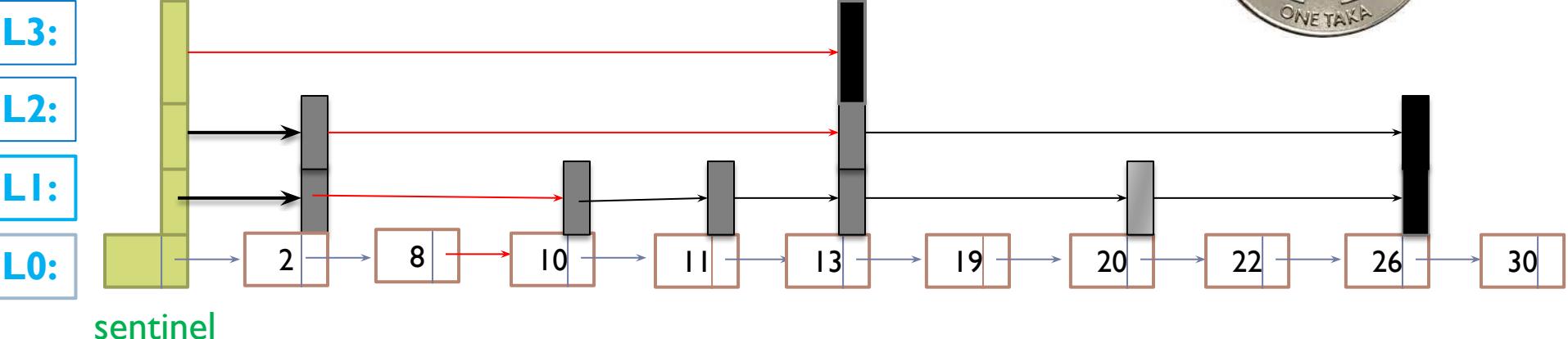
Start From

<b>L3</b>	sentinel
<b>L2</b>	Node 2
<b>L1</b>	Node 2
<b>L0</b>	Node 8

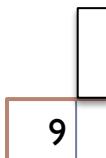
# Insert: key=9

- Second, create a node whose level is random e.g. height=?
- Initially the height is 0

Flip a coin



Create a node:



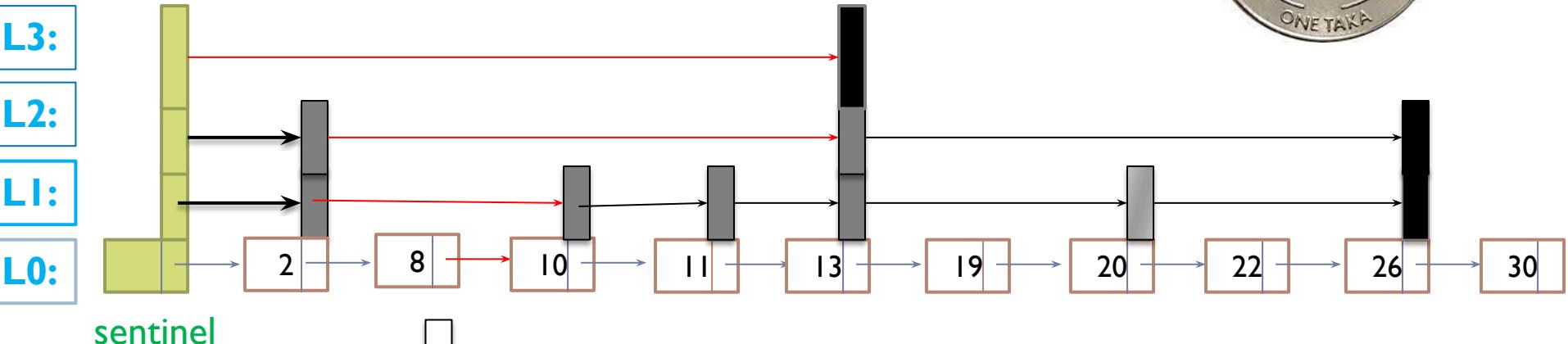
Start From

L3	sentinel
L2	Node 2
L1	Node 2
L0	Node 8

# Insert: key=9

- ❑ Second, create a node whose level is random e.g. height=?
- ❑ the height is 1

Flip a coin



Create a node:



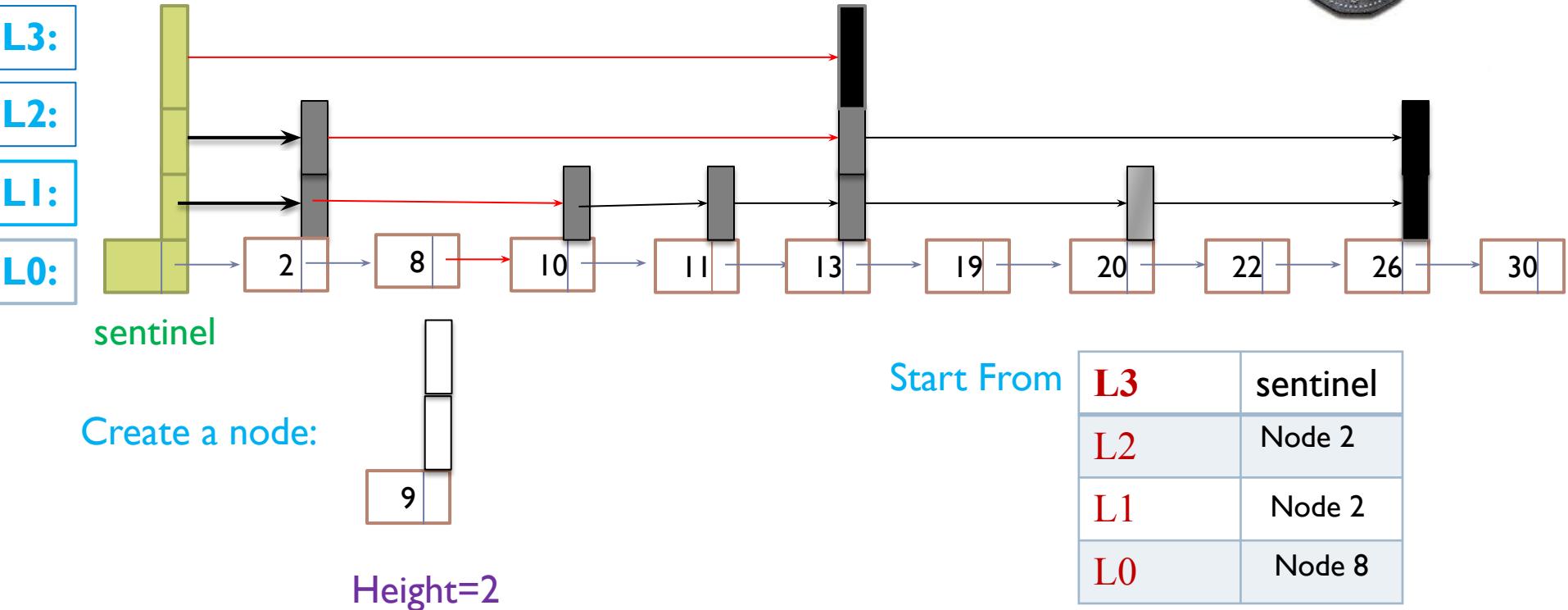
Start From

L3	sentinel
L2	Node 2
L1	Node 2
L0	Node 8

# Insert: key=9

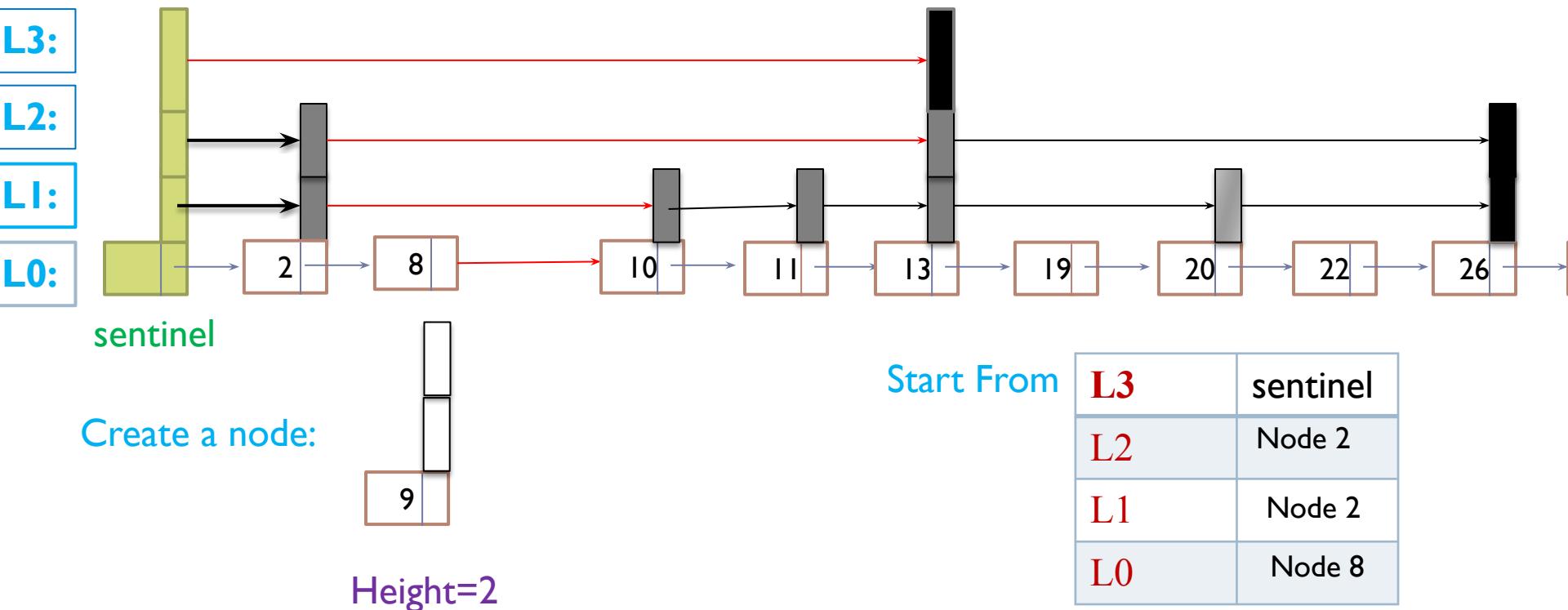
- ❑ Second, create a node whose level is random e.g. height=2
- ❑ the height is 2

Flip a coin



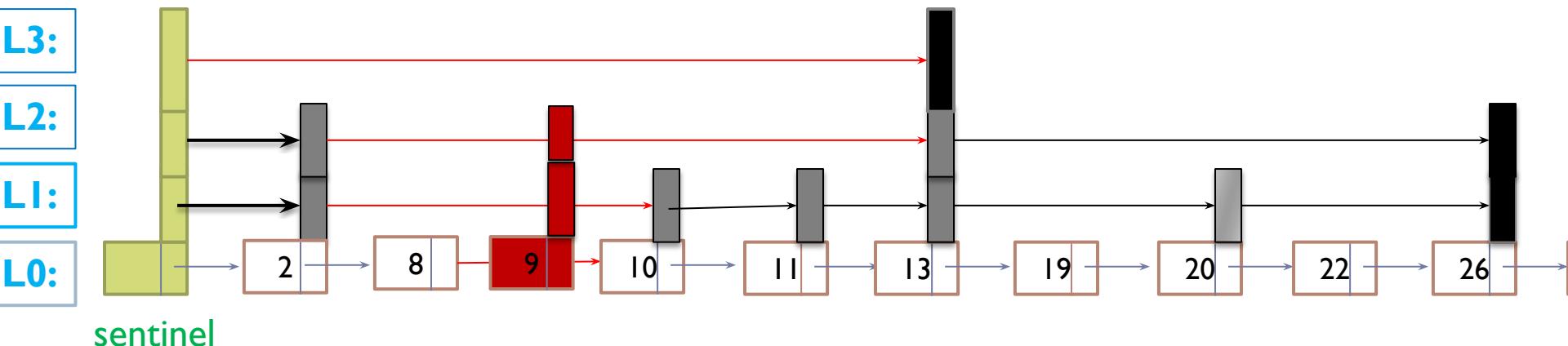
# Insert: key=9

- ❑ Second, create a node whose level is random e.g. height=2
- ❑ the height is 2



# Insert: key=9

- ❑ Second, create a node whose level is random e.g. height=2
- ❑ the height is 2

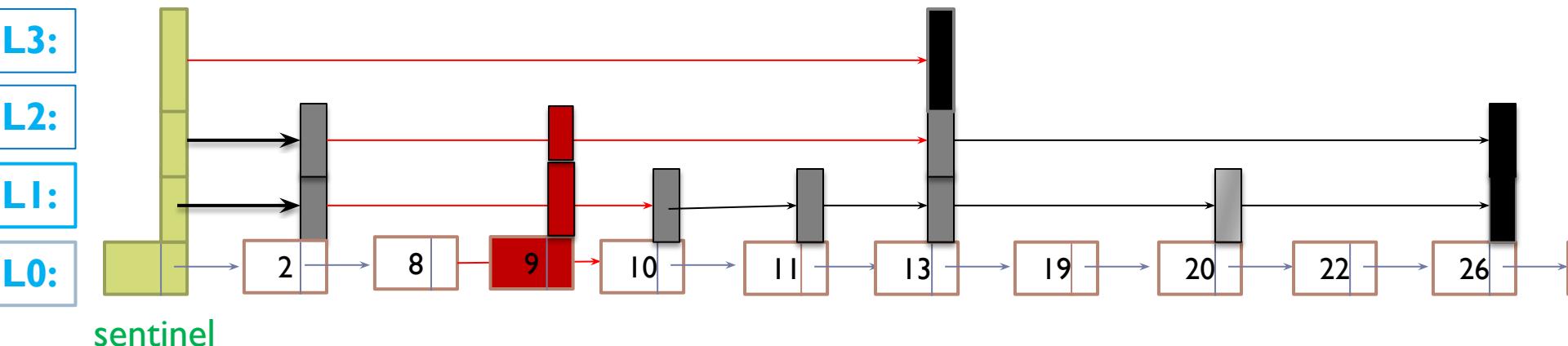


Start From

<b>L3</b>	sentinel
<b>L2</b>	Node 2
<b>L1</b>	Node 2
<b>L0</b>	Node 8

# Insert: key=9

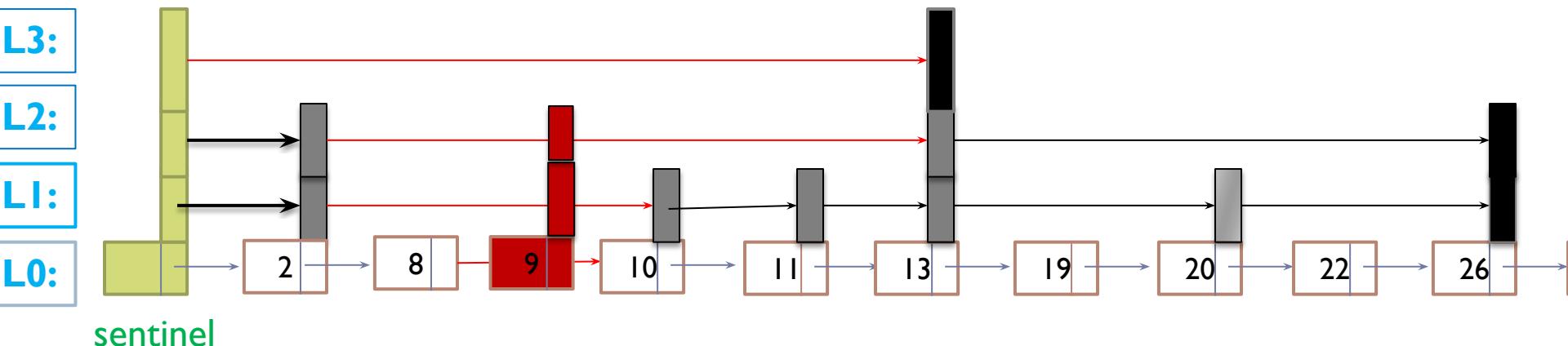
- Third, link the new node to the skip list



Start From	L3	sentinel
L2	Node 2	
L1	Node 2	
L0	Node 8	

# Insert: key=9

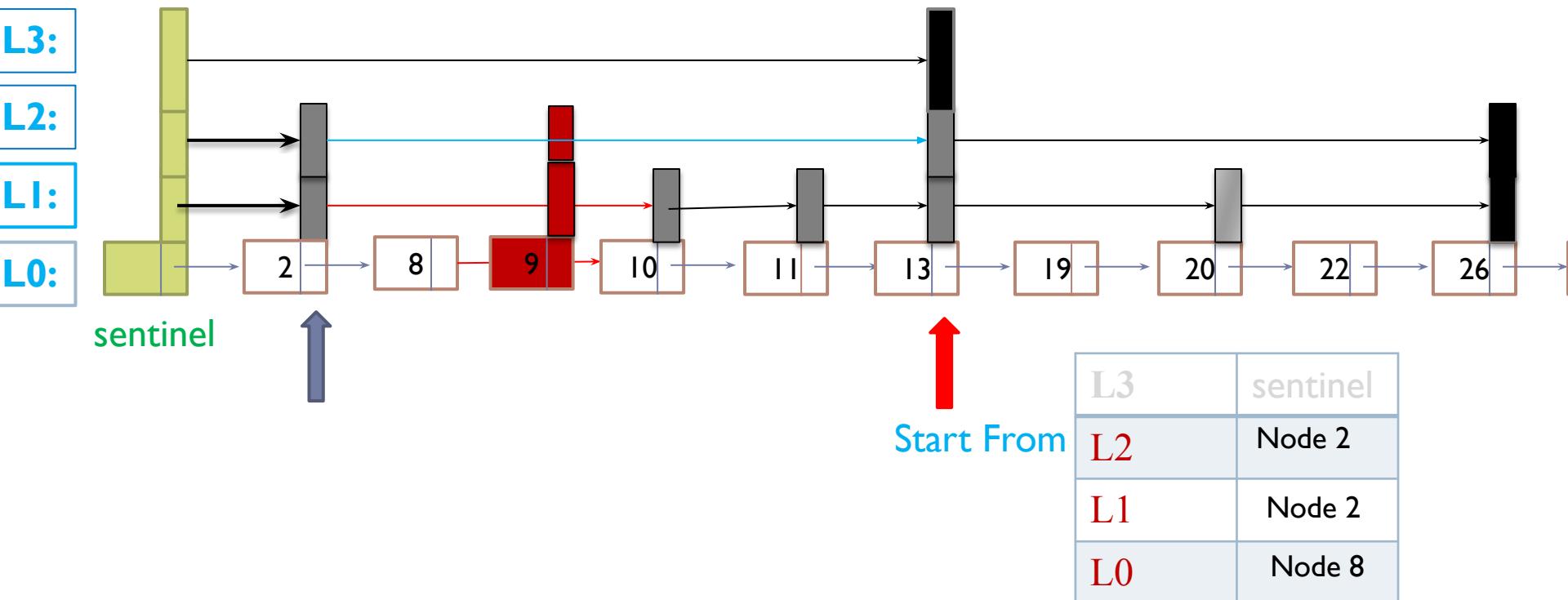
- Third, link the new node to the skip list



Start From	L3	sentinel
L2	Node 2	
L1	Node 2	
L0	Node 8	

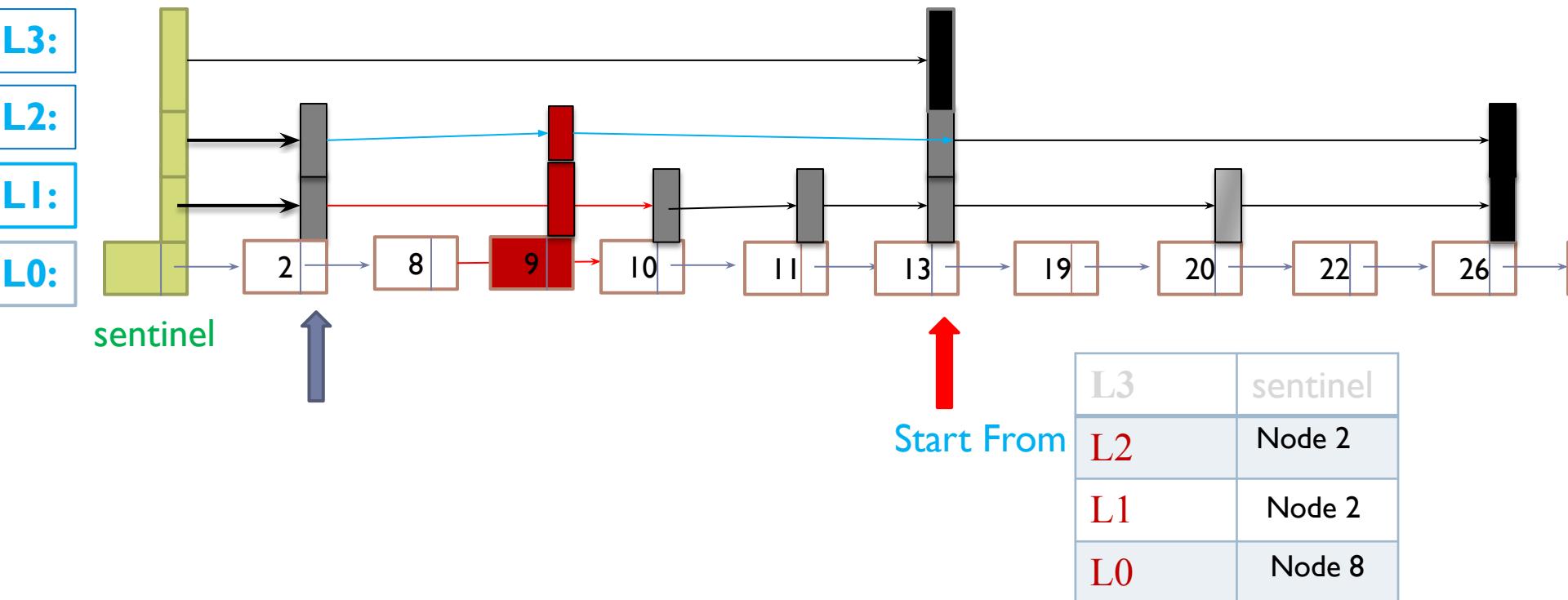
# Insert: key=9

- Third, link the new node to the skip list



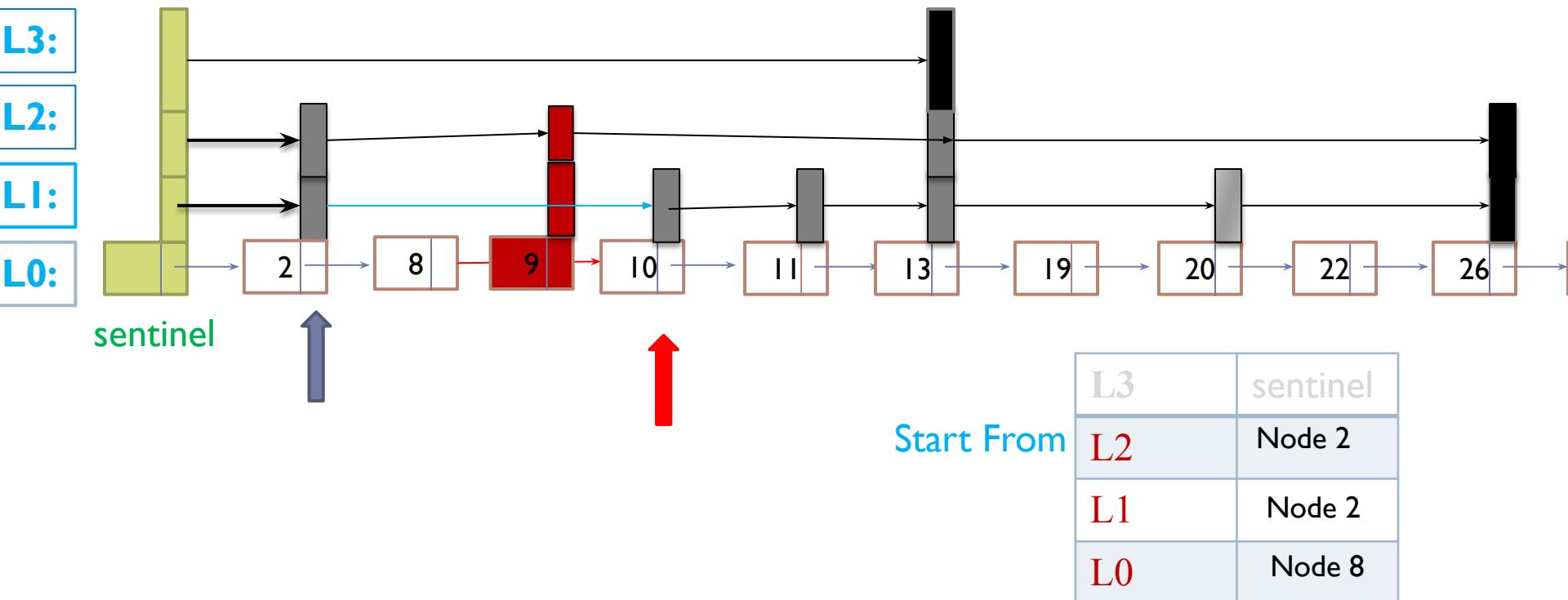
# Insert: key=9

- Third, link the new node to the skip list



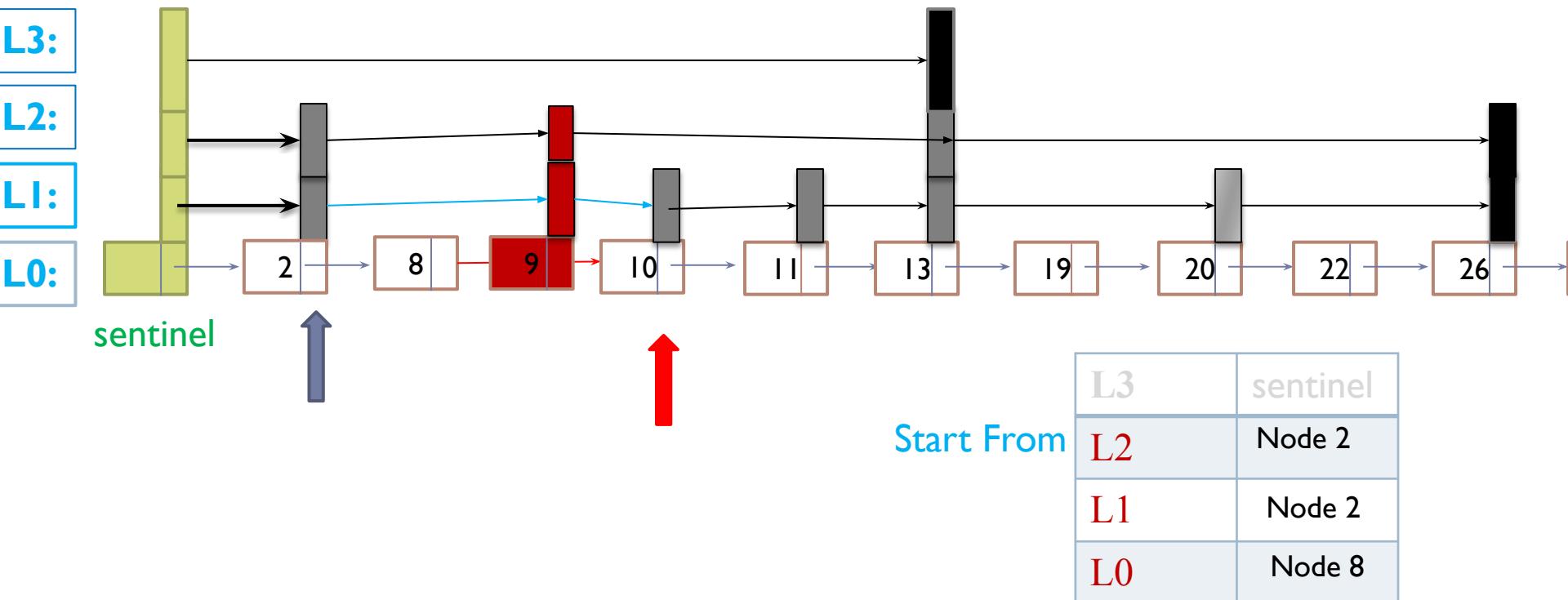
# Insert: key=9

- Third, link the new node to the skip list



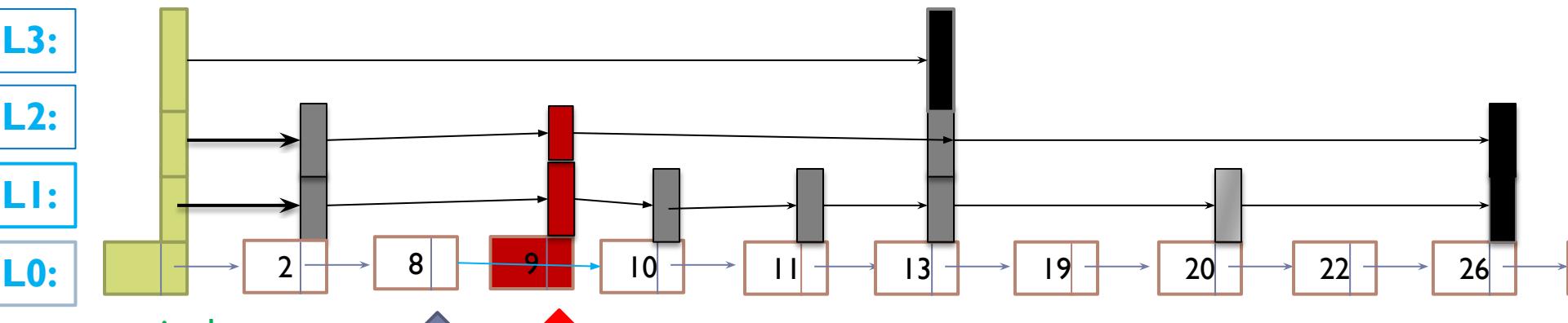
# Insert: key=9

- Third, link the new node to the skip list



# Insert: key=9

- Third, link the new node to the skip list

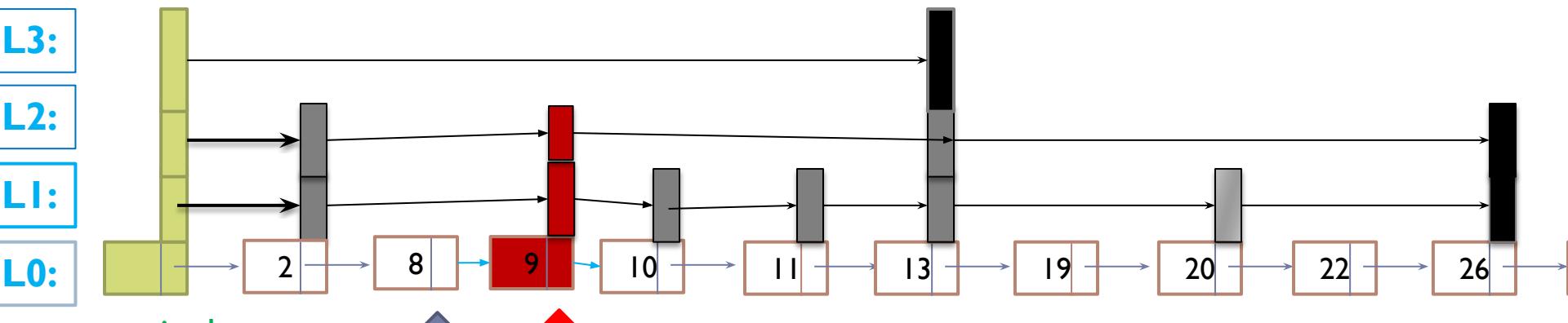


L3	sentinel
L2	Node 2
L1	Node 2
L0	Node 8

Start From

# Insert: key=9

- Third, link the new node to the skip list

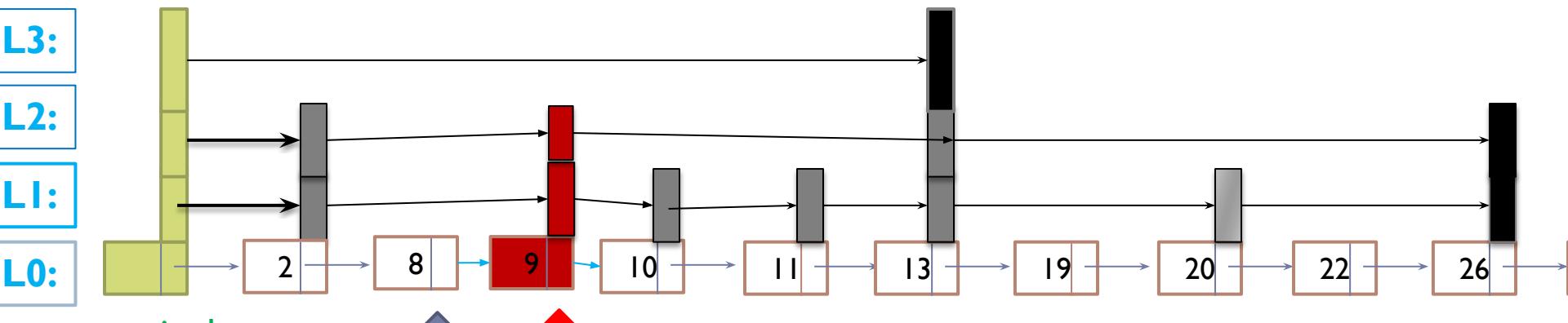


L3	sentinel
L2	Node 2
L1	Node 2
L0	Node 8

Start From

# Insert: key=9

- Third, link the new node to the skip list



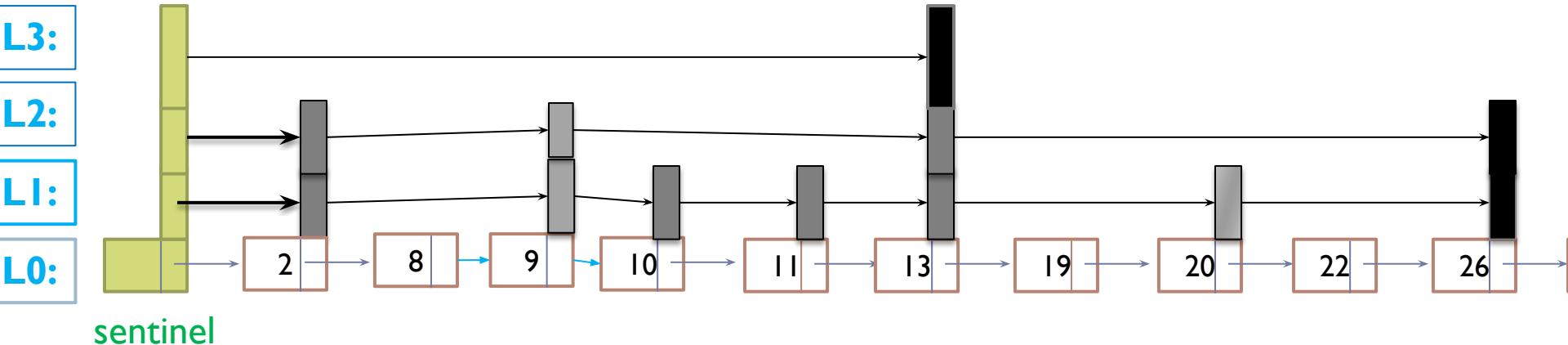
L3	sentinel
L2	Node 2
L1	Node 2
L0	Node 8

Start From

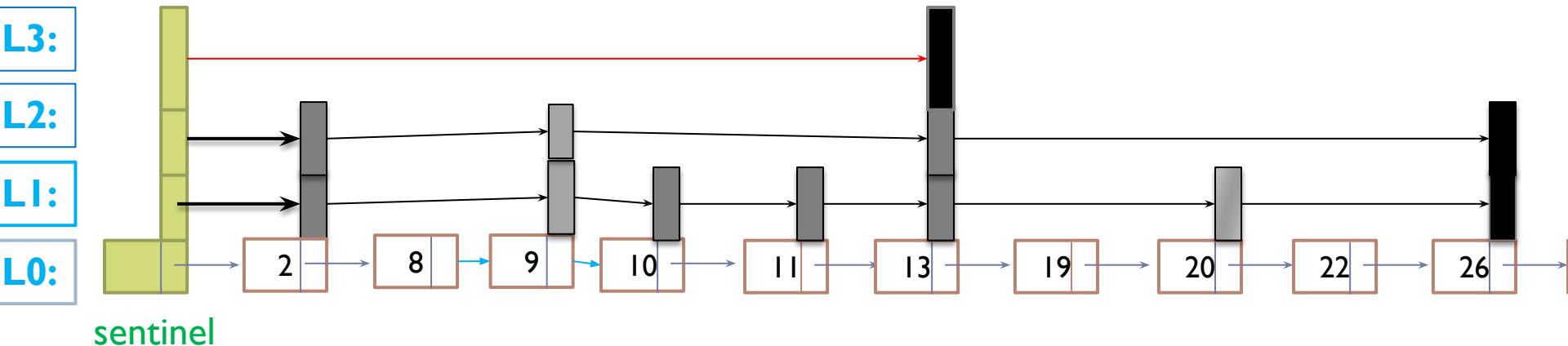


# Deletion

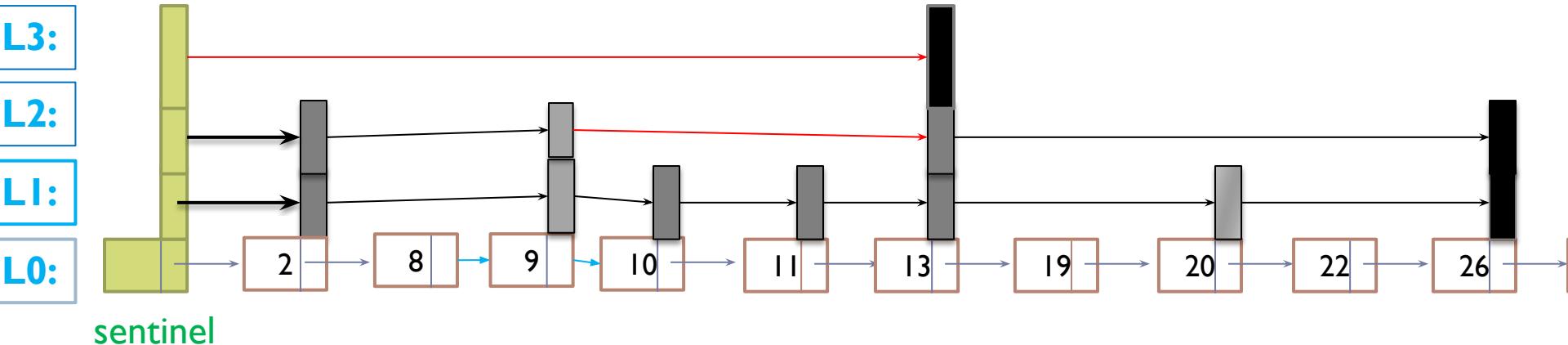
# Delete: key=13



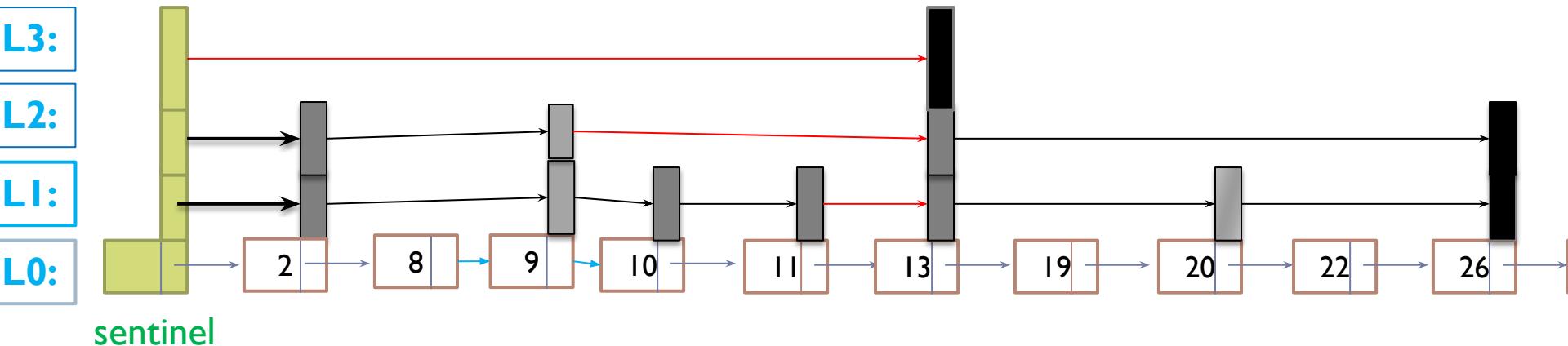
# Delete: key=13



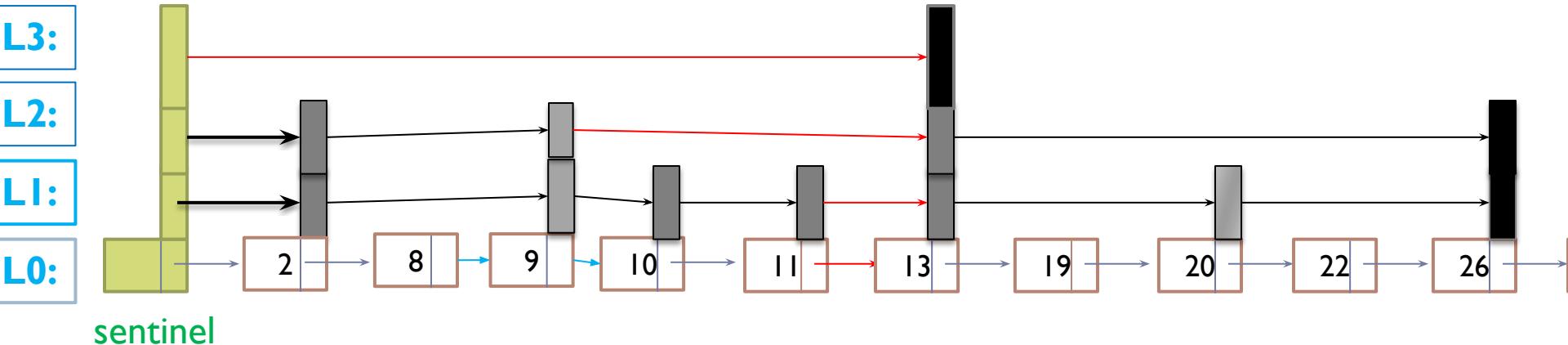
# Delete: key=13



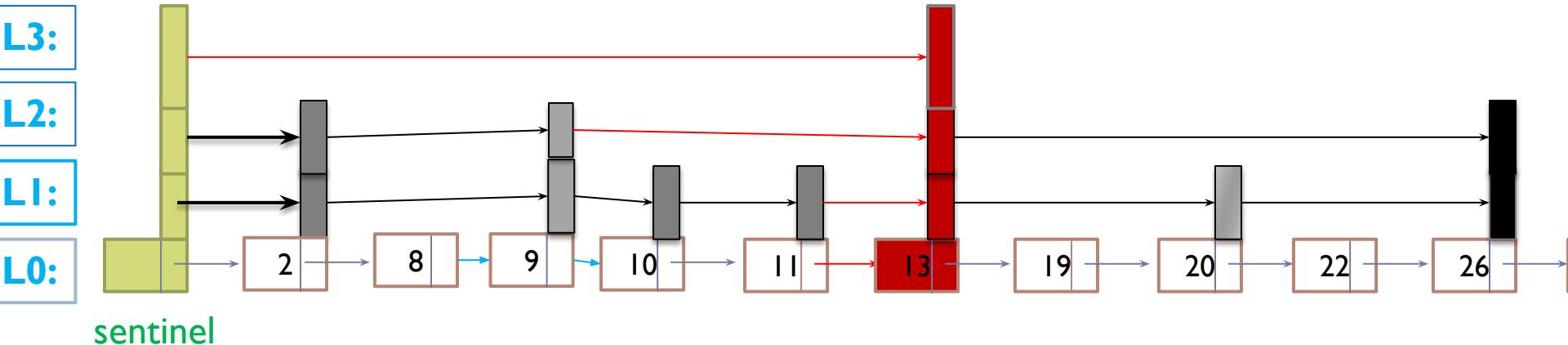
# Delete: key=13



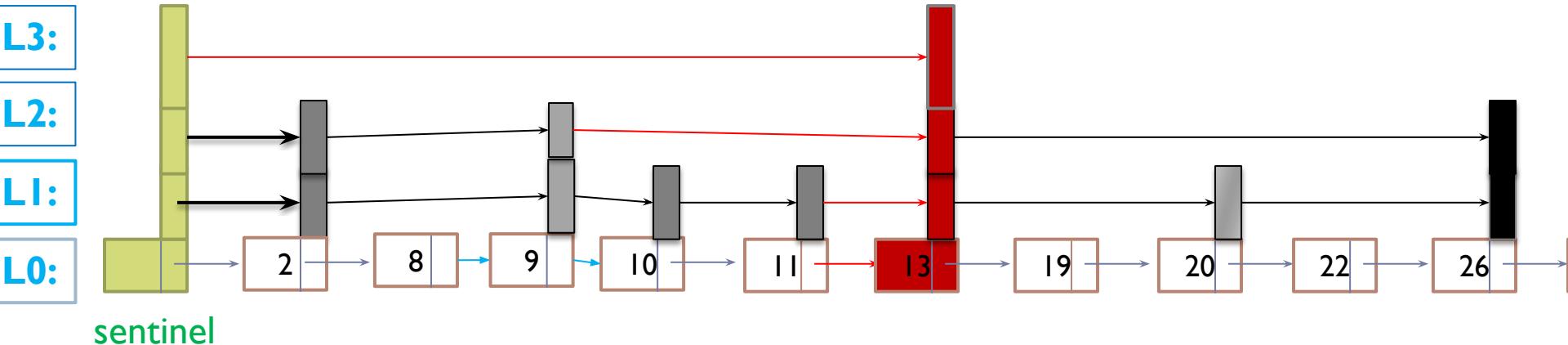
# Delete: key=13



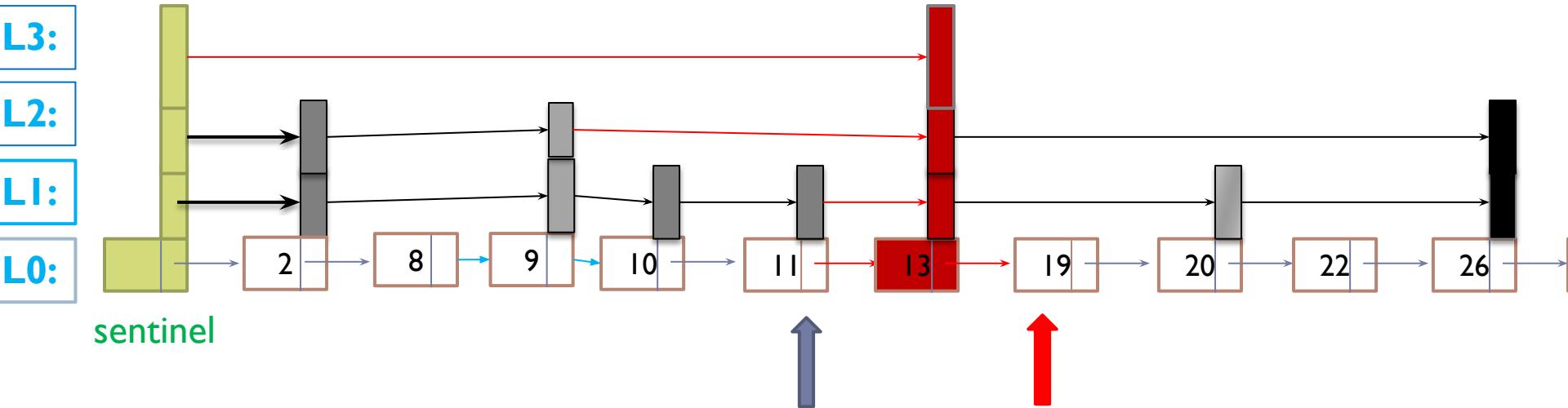
# Delete: key=13



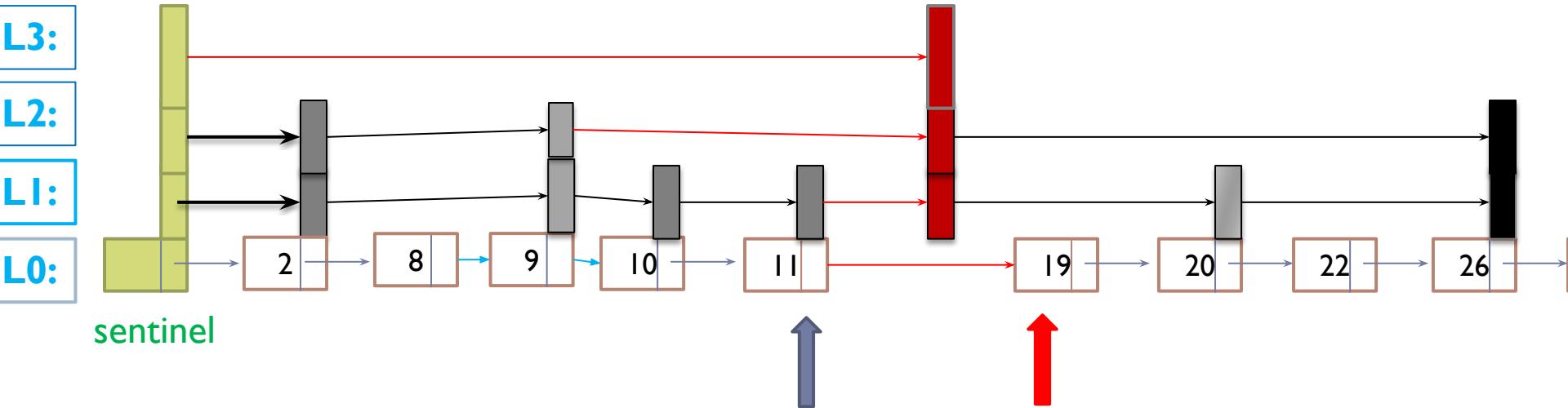
# Delete: key=13



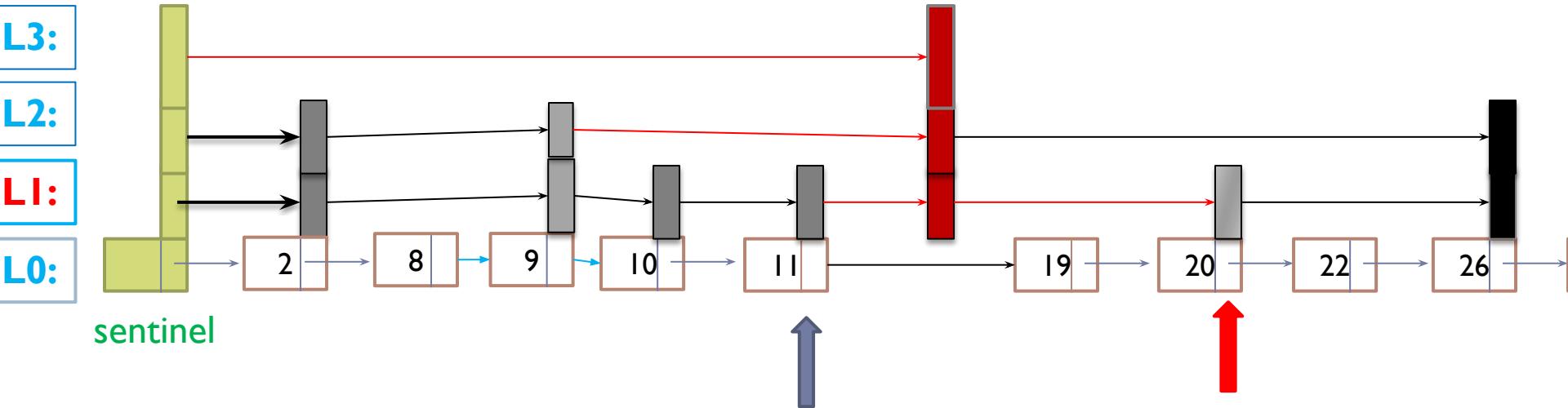
# Delete: key=13



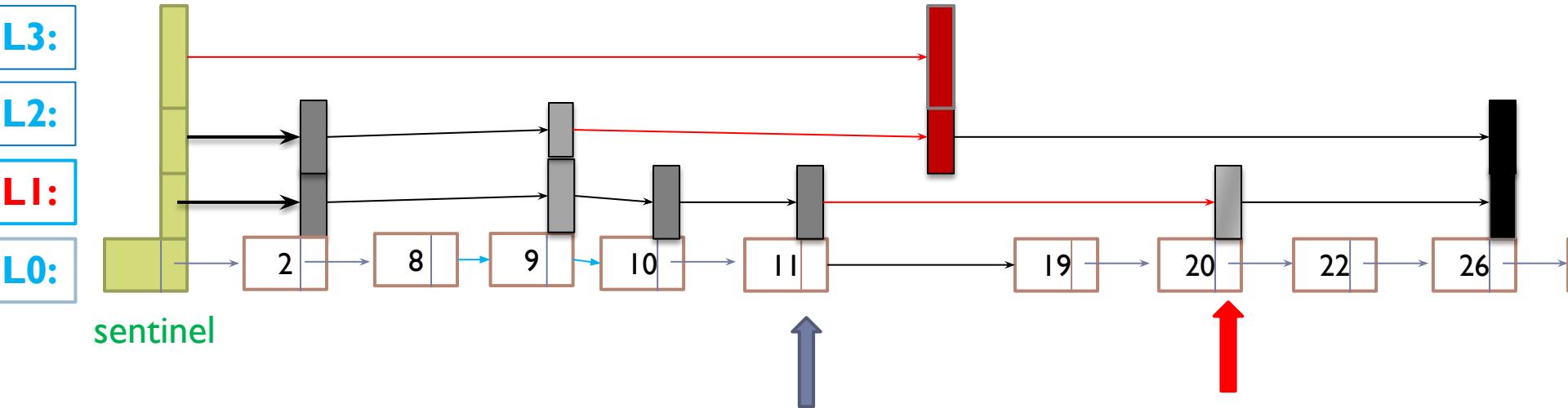
# Delete: key=13



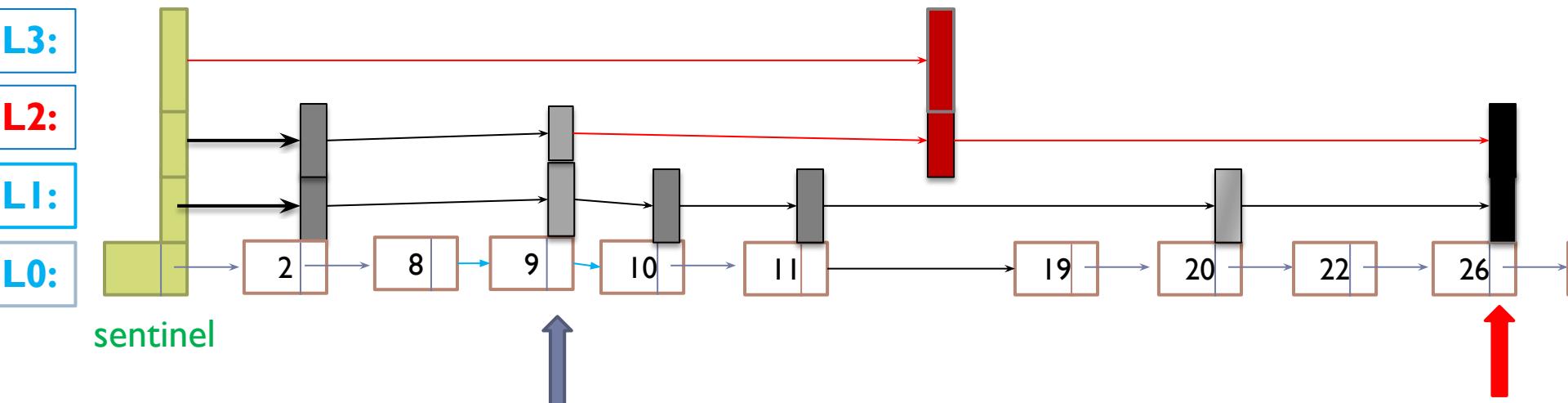
# Delete: key=13



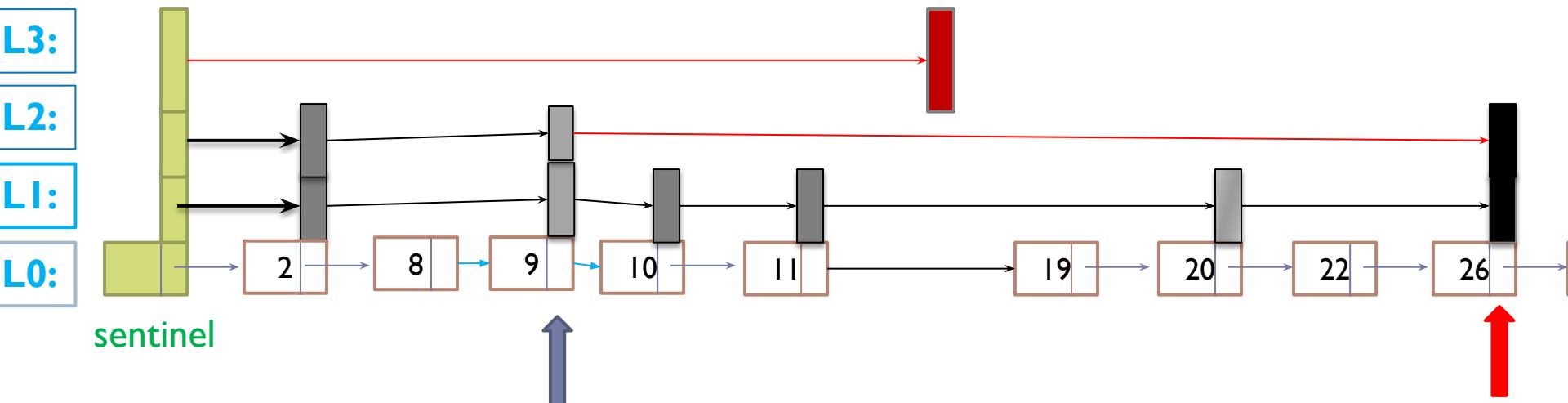
# Delete: key=13



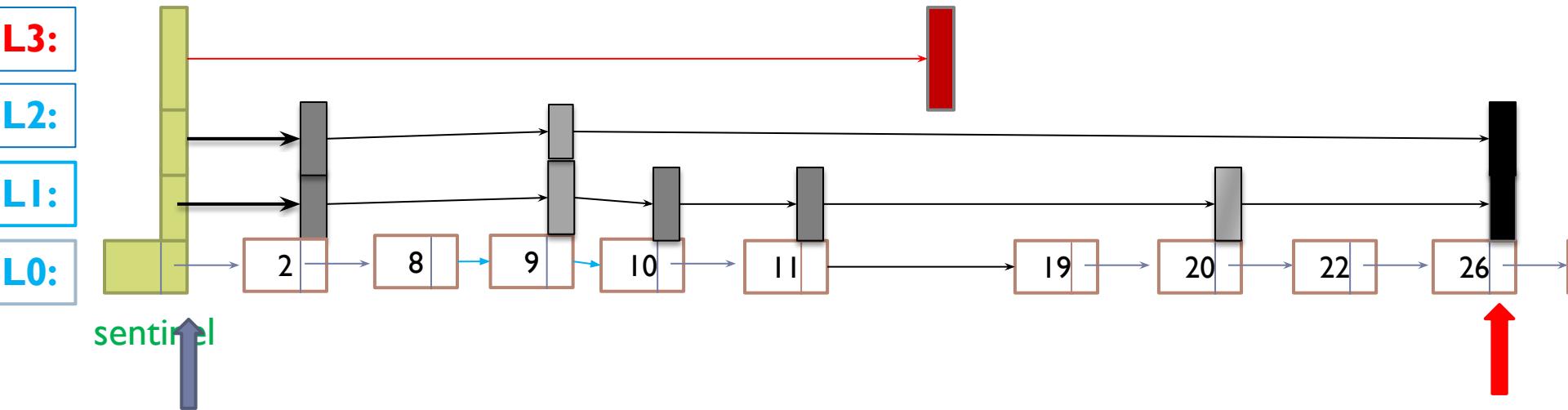
# Delete: key=13



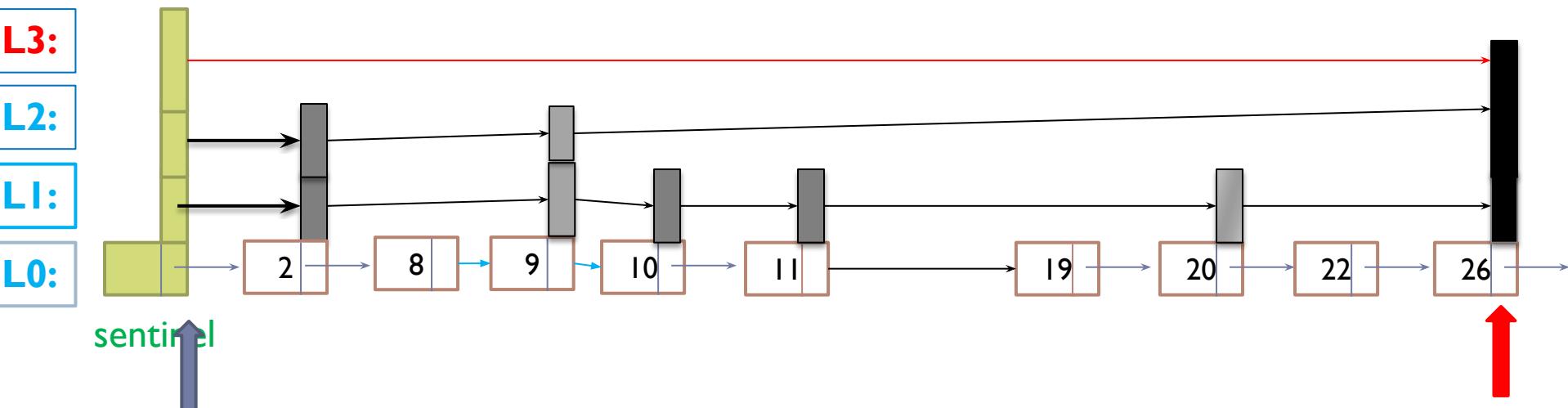
# Delete: key=13



# Delete: key=13

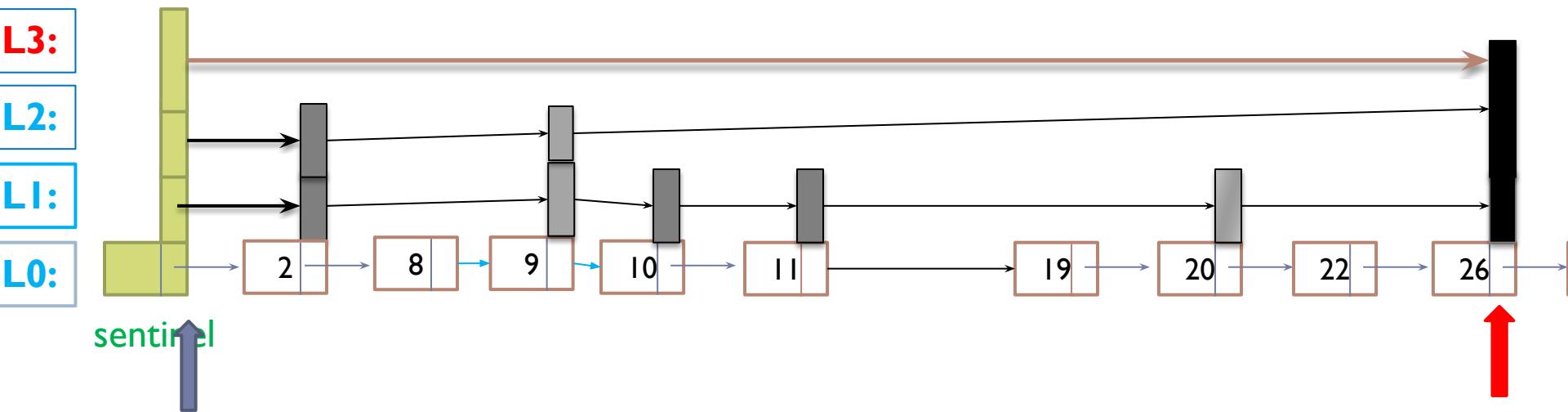


# Delete: key=13

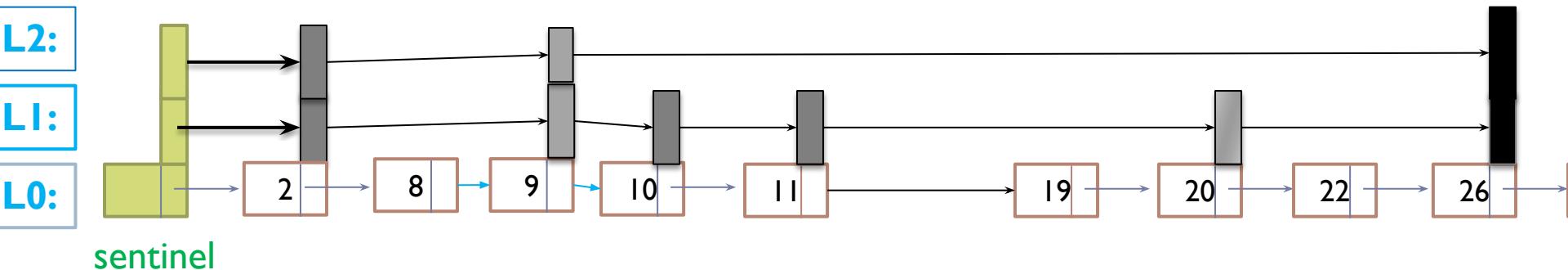


# Delete: key=13

In level 3 no node except sentinel and Null  
 So this level should be deleted



# Delete: key=13





# There are no “bad” sequences:

- ❑ We expect a randomized skip list to perform about as well as a perfect skip list.
- ❑ With some *very* small probability,
  - ❑ - the skip list will just be a linked list, or
  - ❑ - the skip list will have every node at every level
  - ❑ - These *degenerate* skip lists are very unlikely!
- ❑ Level structure of a skip list is independent of the keys you insert.
- ❑ Therefore, there are no “bad” key sequences that will lead to degenerate skip lists



# Skip List Analysis

- ? Expected number of levels =  $O(\log n)$ 
  - $E[\# \text{ nodes at level 1}] = n/2$
  - $E[\# \text{ nodes at level 2}] = n/4$
  - ...
  - $E[\# \text{ nodes at level } \log n] = 1$
- ? Still need to prove that # of steps at each level is small.



# Backward Analysis

---

- ❑ Consider the *reverse* of the path you took to find  $k$ :
  - Note that you *always* move up if you can.
  - (because you always enter a node from its topmost level when doing a find)*

# Analysis Continued

- ❑ What's the probability that you can move up at a give step of the reverse walk?  
**0.5**
- ❑ Steps to go up  $j$  levels =  
Make one step, then make either  
 $C(j-1)$  steps if this step went up [Prob = 0.5]  
 $C(j)$  steps if this step went left [Prob = 0.5]
- ❑ Expected # of steps to walk up  $j$  levels is:  
$$C(j) = 1 + 0.5C(j-1) + 0.5C(j)$$

# Analysis Continued 2

- ❑ Expected # of steps to walk up  $j$  levels is:

$$C(j) = 1 + 0.5C(j-1) + 0.5C(j)$$

So:

- ❑  $2C(j) = 2 + C(j-1) + C(j)$
- ❑  $C(j) = 2 + C(j-1)$

Expected # of steps at each level = 2

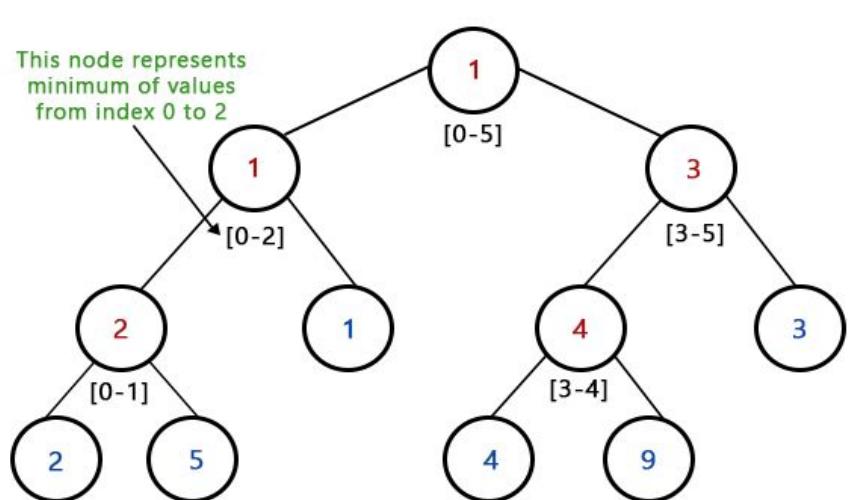
- ❑ Expanding  $C(j)$  above gives us:  $C(j) = 2j$
- ❑ Since  $O(\log n)$  levels, we have  $O(\log n)$  steps, expected

---

# Thank You

# CSE-203: Data Structures & Algorithms I

## Topic: Segment Tree



Nafiz Imtiaz Khan  
Lecturer, Department of CSE, MIST  
Email: nafiz@cse.mist.ac.bd



# SUM OF ARRAY

---

- An array  $a$  is given of size  $n$  with  $q$  number of queries
- Each query involves  $i$  and  $j$  that indicates a range  $[i,j]$
- For each query it is asked to print the sum of the elements from  $a[i]$  to  $a[j]$
- What is the worst case complexity to answer all the queries?
  - In the worst case  $i$  can be 0 and  $j$  can be  $n-1$  for each query
  - That means the entire array need to be searched for each query
  - So in the worst case the time complexity is  $O(qn)$
- If  $q = 10^6$  and  $n = 10^3$  then the required time in worst case is  $10^9$  that surely exceeds 1 second
- Segment tree deals here to answer each query in  $\log n$  amount of time in worst case
- As a result it will take  $10^6 \times \log 10^3 \approx 10^7$  time that is less than 1 second
- Segment tree is widely used for answering the range queries



# CONCEPT OF SEGMENT TREE

---

- ❑ The main idea is to build a tree of height  $\log n$  from the input data before answering the queries
- ❑ For each query answer from the tree rather than searching the array from  $i$  to  $j$
- ❑ The tree maintains the following properties
  - The root is located at index-1
  - Left child of index- $i$  is located at index- $2i$  and right child is located as index- $(2i+1)$
  - Parent of index- $i$  is located at index- $\text{floor}(i/2)$
- ❑ It implies Segment Tree as a complete binary tree
- ❑ Each node of the tree deals with 3 indices  $(i, j, t_i)$
- ❑  $(i, j)$  together defines a range and a node stores the sum of the elements from  $i$  to  $j$  of the given array
- ❑  $t_i$  indicates the index of the node in the tree
- ❑ Segment tree follows divide and conquer approach to build the tree



# CONSTRUCTION OF TREE

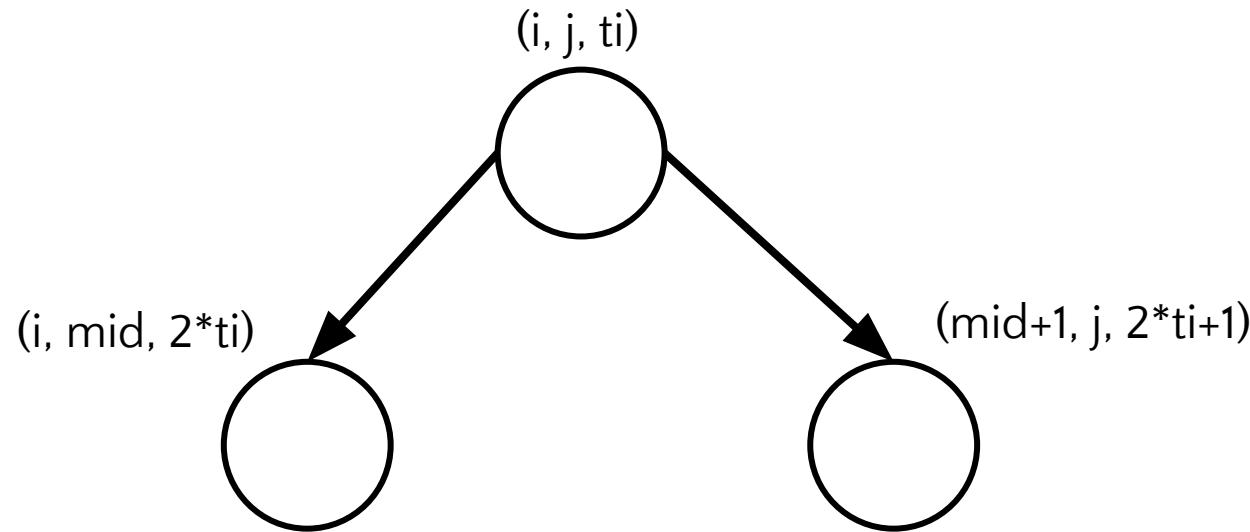
---

- ❑ Each segment is denoted as  $(i, j, ti)$  where
  - $(i, j)$  defines the range of the segment
  - $ti$  defines the tree index of the segment
- ❑ If the size of the input array is  $n$  then the segment tree starts dividing from  $(i=0, j=n-1, ti=1)$
- ❑ At each iteration the current range  $(i, j)$  is divided into 2 segments
- ❑ Range of left segment:  $(i, mid)$
- ❑ Range of right segment:  $(mid+1, j)$
- ❑  $mid = \text{floor}((i+j)/2)$
- ❑ Node for the left segment is stored into index- $(2*ti)$  in the tree
- ❑ Node for the right segment is stored into index- $(2*ti+1)$  in the tree
- ❑ Segment tree follows divide and conquer approach to build the tree



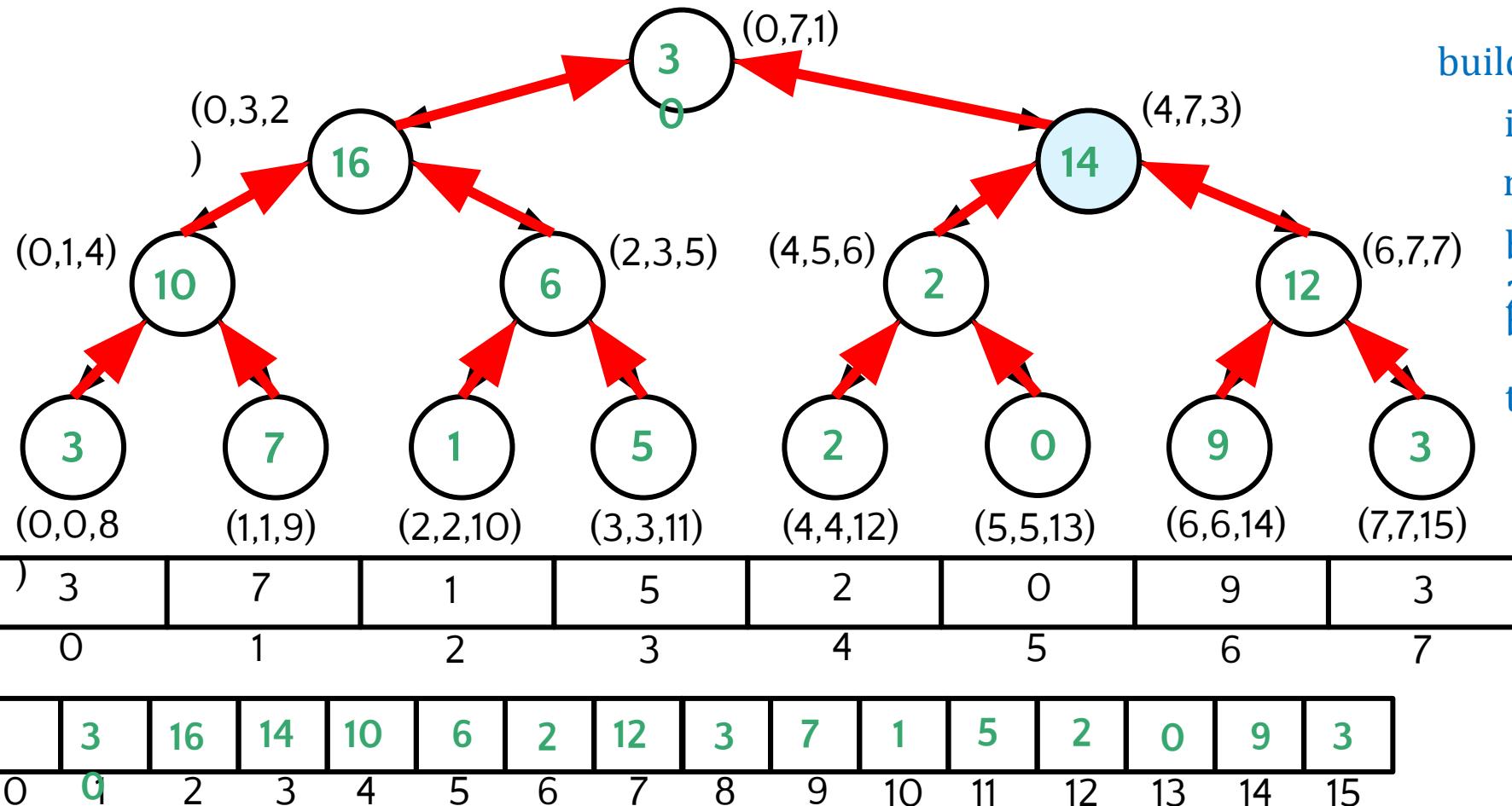
# CONSTRUCTION OF TREE

---



- When  $i$  and  $j$  becomes the same in that case it refers a single element and treated as base case

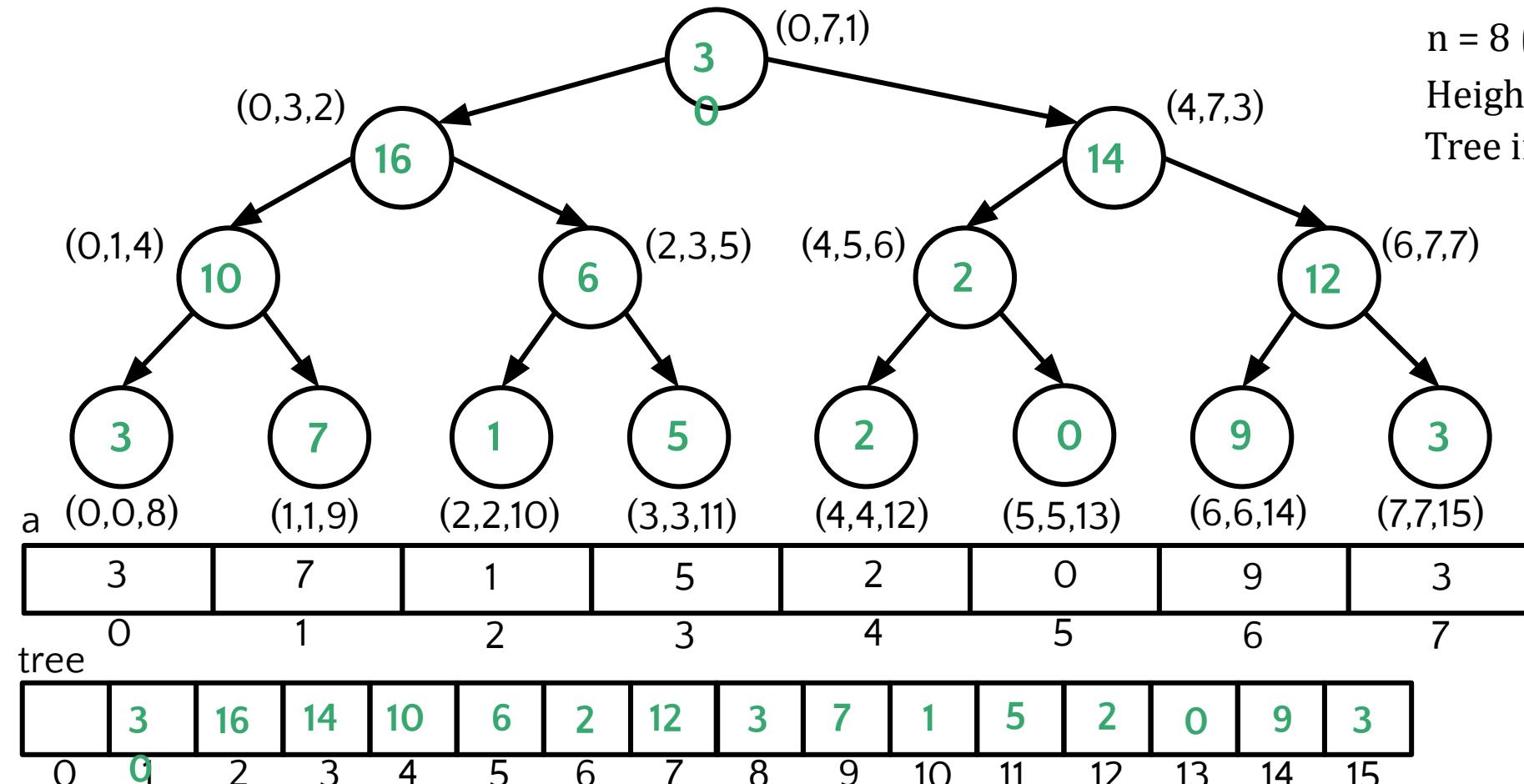
# CONSTRUCTION OF TREE



buildTree( $i=0, j=n-1, ti=1$ )  
 if( $i==j$ )  $tree[ti] = a[i]$   
 $mid = (i+j)/2$   
 $buildTree(i, mid, 2*ti)$   
 $buildTree(mid+1, j, 2*ti+1)$   
 $tree[ti]=tree[2*ti]+tree[2*ti+1]$



# CONSTRUCTION OF TREE



$n = 8$  (Number of elements in  $a$ )

Height of tree,  $h = 3$  ( $\log_2 n$ )

Tree index of the last node in tree is  $N$

$$N = 2^0 + 2^1 + 2^2 + \dots + 2^h \dots \text{(i)}$$

$$N - 2^0 = 2^1 + 2^2 + \dots + 2^h$$

$$N - 1 = 2^1 + 2^2 + \dots + 2^h$$

$$N - 1 = 2(2^0 + 2^1 + \dots + 2^{h-1})$$

$$N - 1 = 2(2^0 + 2^1 + \dots + 2^{h-1} + 2^h - 2^h)$$

$$N - 1 = 2(2^0 + 2^1 + \dots + 2^{h-1} + 2^h) - 2 \cdot 2^h$$

$$N - 1 = 2N - 2 \cdot 2^h$$

$$N = 2x(2^{\log_2 n}) - 1$$

$$N = 2 \cdot 2^h - 1$$

$$N = 2n - 1$$

So the required memory for the tree is  $2n$ , as it is considered that the array used to store the tree is indexed from 0 to  $N-1$ .

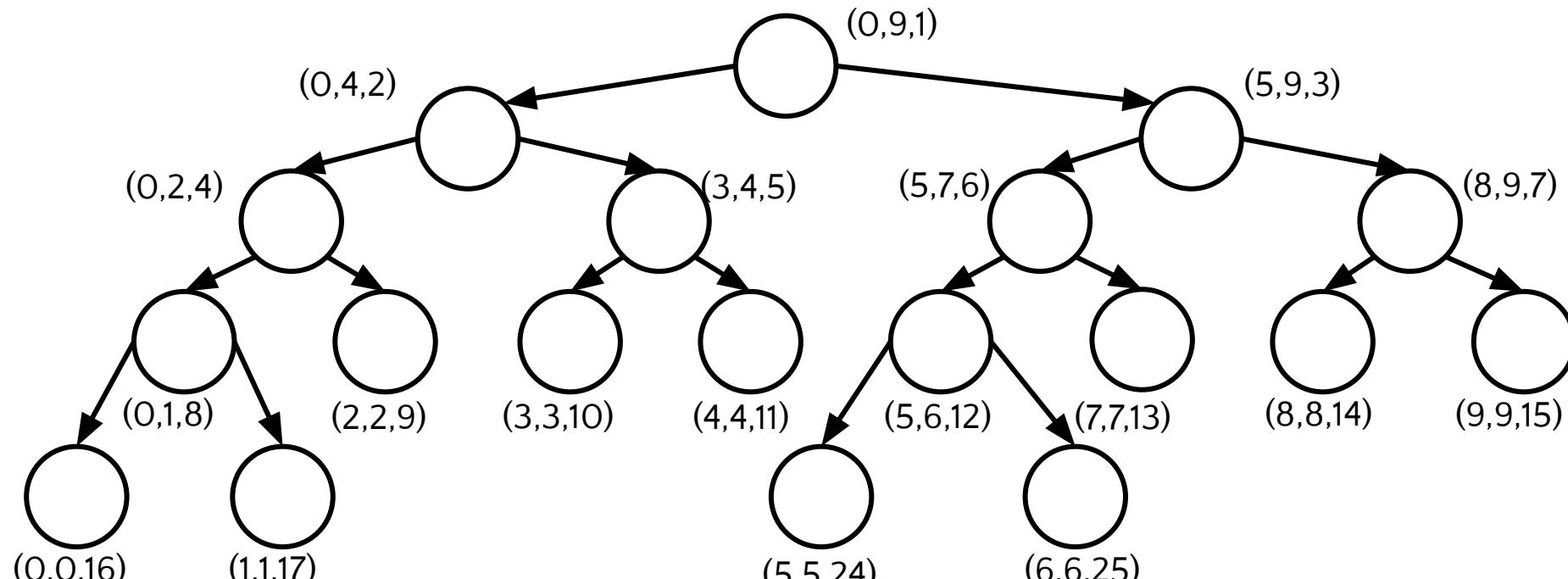
Here, size of the given array,  $n = 8$

So, size of the tree is,  $N = 2 \times 8 = 16$

So the theorem is proved correct

# CONSTRUCTION OF TREE

- Will the tree index of the last node (N) will be always  $2n$ ?  **NO**
- Draw a segment tree for  $n=10$  showing the range and tree index for each node



tree	[ ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ X ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ X ]	[ X ]	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	2	21	22	23	24	25



# CONSTRUCTION OF TREE

---

- Why  $N=2n$  didn't work for  $n=10$ ?
- Because  $N=2n$  comes from  $N = 2 \cdot 2^h$  where  $h = \log_2 n$
- So from algebra, if  $n = 10$  then  $h = 3.32$
- But in the previous tree for  $n=10$ , height of the tree is 4
- Means there can be at most  $2^4$  number of nodes at the last level
- But as  $h=3.32$  is taken for derivation so it is considered that there can be at most  $2^{3.32}$  number of nodes at the last level which is less counting
- Assuming  $h=3.32$  is leading to  $N=2n$
- But if  $h=\text{ceil}(\log_2 n)$  would taken then it would lead  $N>2n$
- It can be proved that if  $h=\text{ceil}(\log_2 n)$  is taken then  $N$  will never exceed  $4n$
- So, for a segment tree the memory complexity is  $O(4n)$  or  $O(n)$
- Note that, when  $n$  is a power of 2 then  $N=2n$  works properly as in that case  $\log_2 n$  and  $\text{ceil}(\log_2 n)$  provides same value

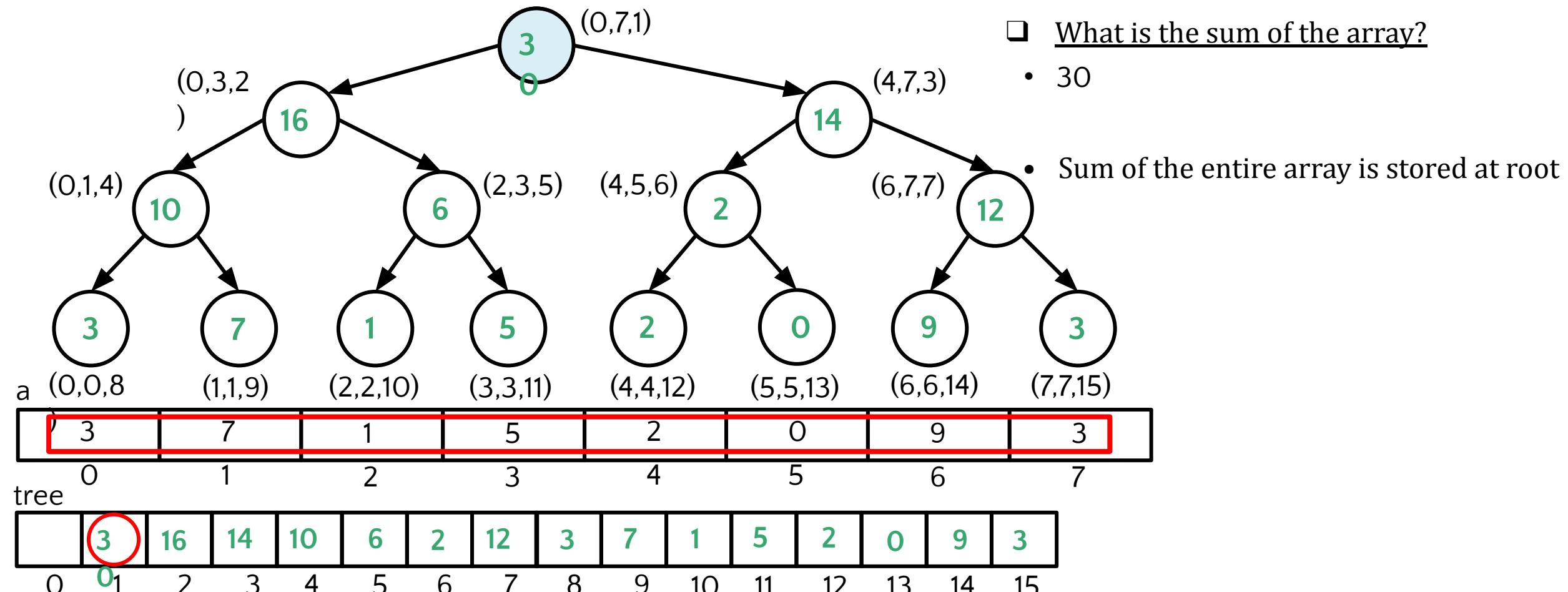


# CONSTRUCTION OF TREE

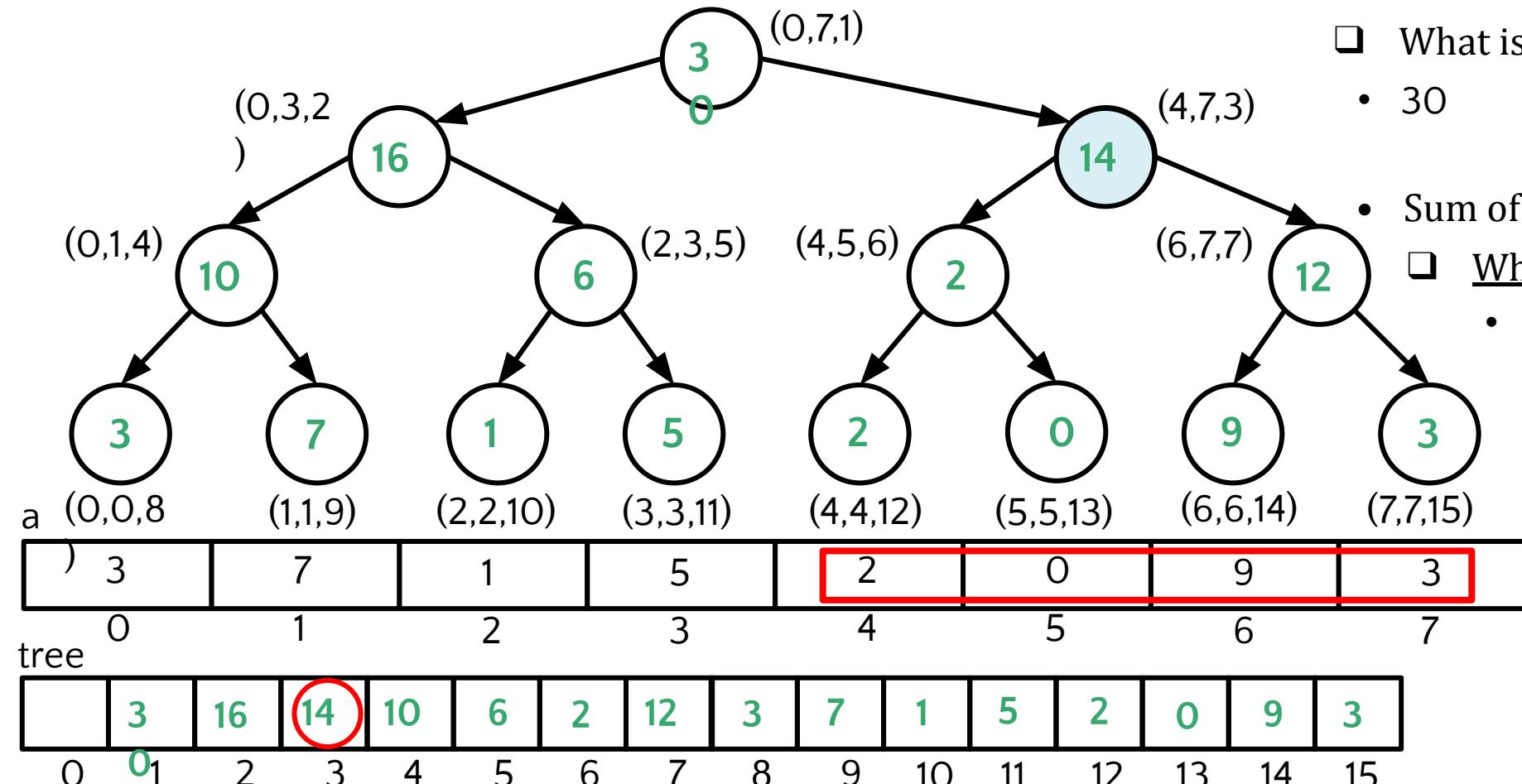
---

- ❑ Time complexity for construction of tree
  - Construction of tree is nothing but setting the values of an array of size at most  $4n$
  - Means there can be at most  $4n$  number of recursive calls to build the tree
  - So, the worst case time complexity is  $O(4n)$  or  $O(n)$

# QUERY

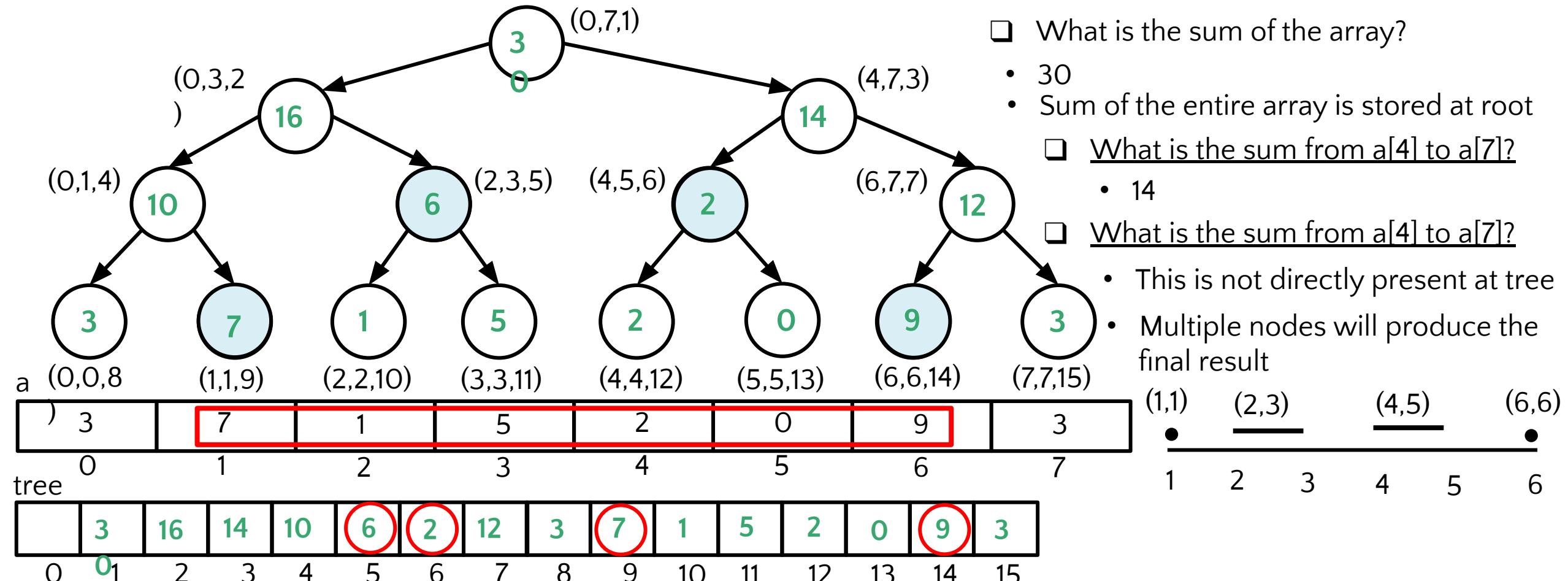


# QUERY

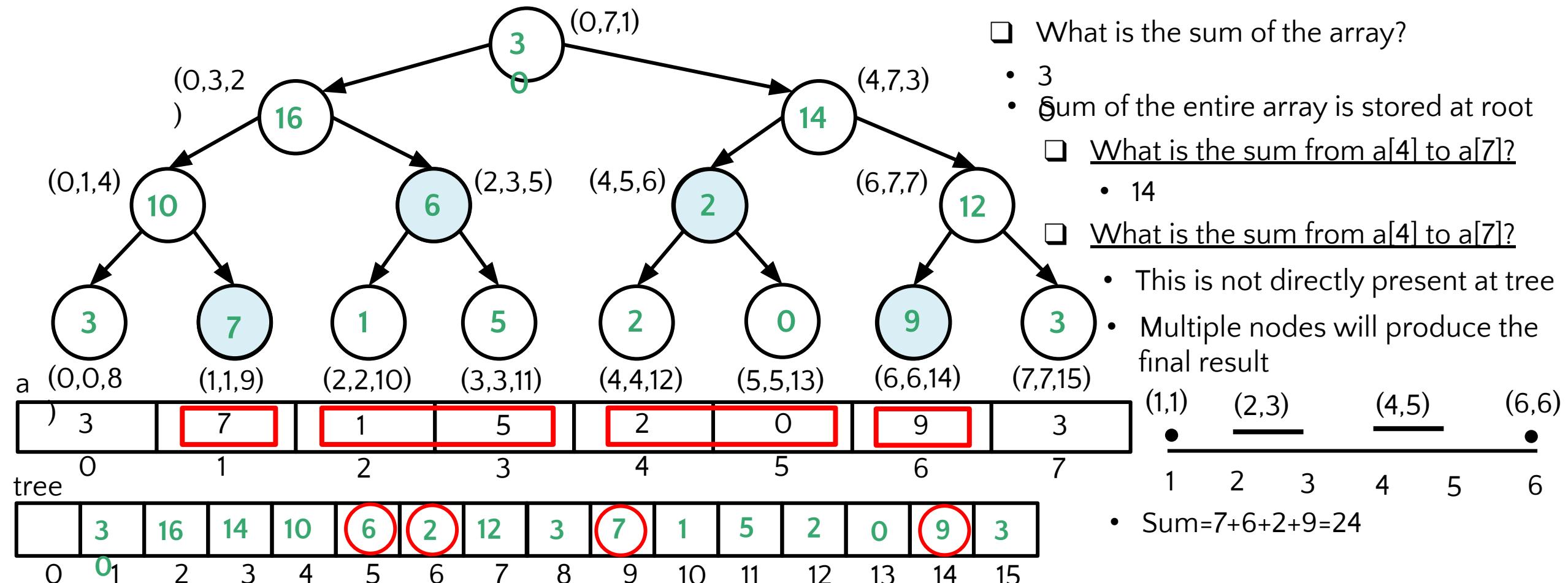


- What is the sum of the array?
  - 30
- Sum of the entire array is stored at root
  - What is the sum from  $a[4]$  to  $a[7]$ ?
    - 14

# QUERY



# QUERY





# QUERY

- In last query the sum of the elements from of  $a$  from 1 to 6 has been asked
- The left point of the query is denoted as  $qi$  and the right point of the query is denoted as  $qj$
- Here,  $qi = 1$  and  $qj = 6$
- If the range of a tree node completely falls on  $(qi \ qj)$  then the node is considered for contribution
- The range of a tree is denoted as  $(i \ j)$  and the corresponding tree index is  $ti$
- 3 cases**
  - If the range of a tree node completely falls on  $(qi \ qj)$  then that node is a contributor

$qi \xrightarrow{i \text{ --- } j} qj \quad \text{if}(i \geq qi \ \&\& \ j \leq qj) \quad \text{return tree}[ti];$

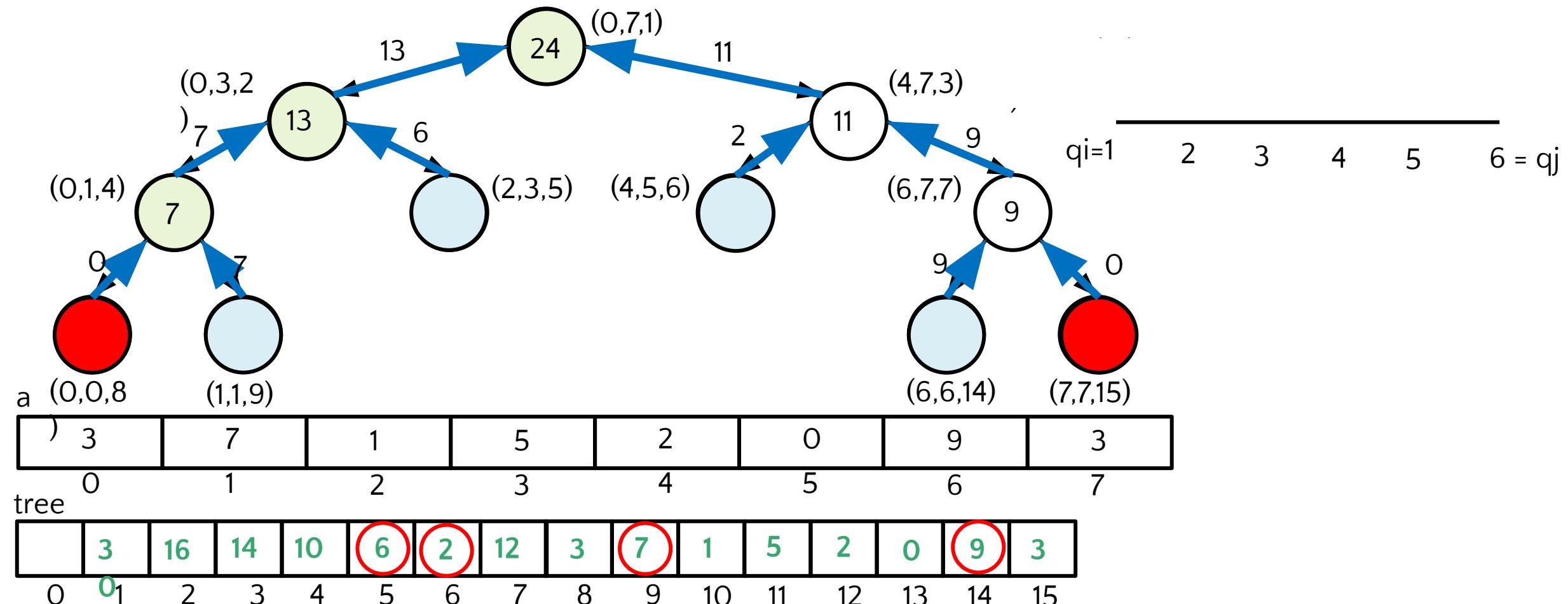
- If the range of a tree node completely falls out of  $(qi \ qj)$  then that node is ignored

$i \text{ --- } j \quad qi \xrightarrow{\quad\quad\quad} qj \quad i \text{ --- } j \quad \text{else if}(j < qi \ || \ i > qj) \quad \text{return NULL};$

- Otherwise the node is explored for both left and right (partial overlap)

$i \text{ --- } j \quad i \text{ --- } j \quad \text{else return query}(i, mid, qi, qj, 2 * ti) + query(mid+1, j, qi, qj, 2 * ti+1);$

# QUERY





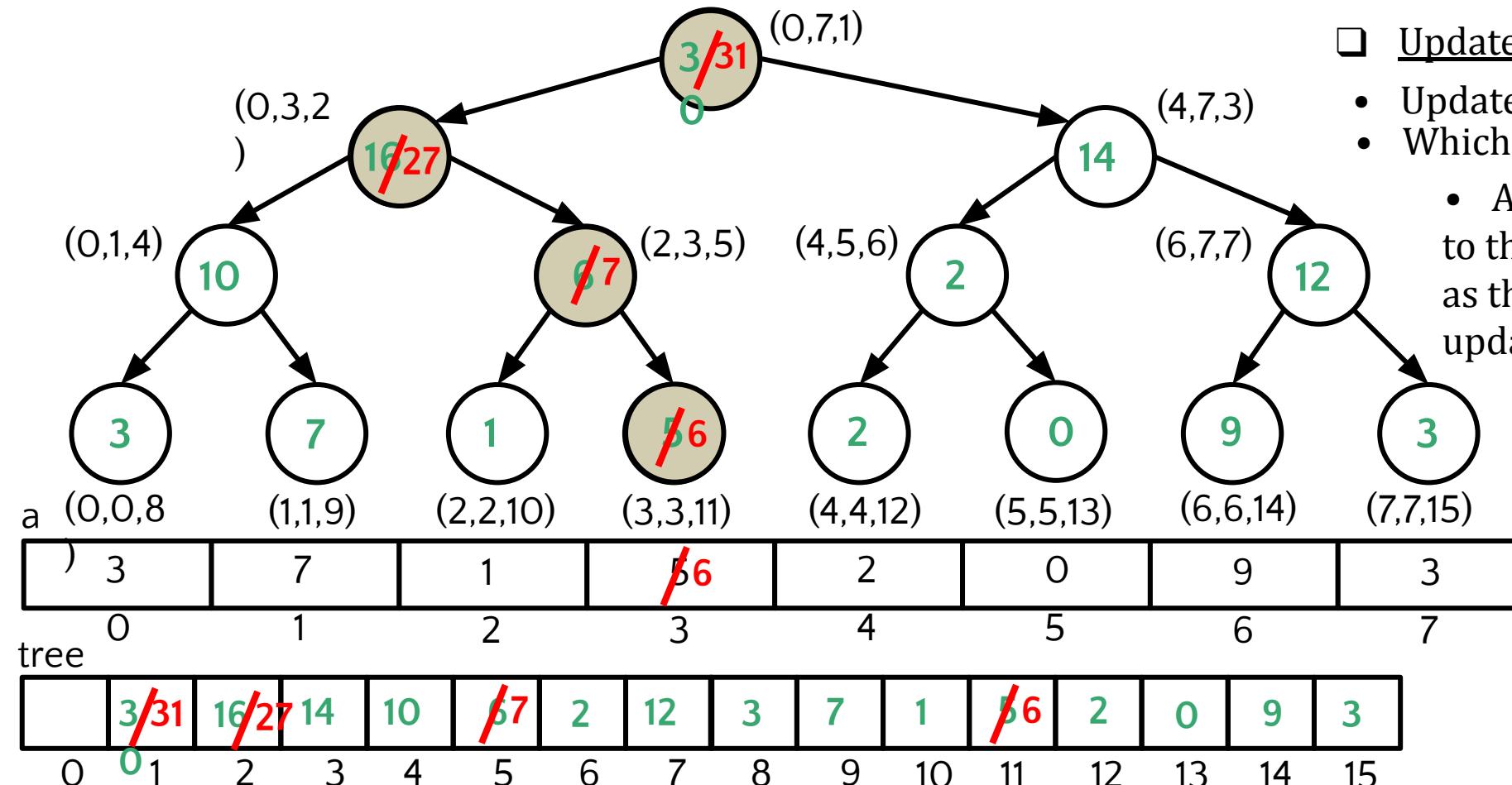
# QUERY

---

```
int query(int tL, int tR, int qL, int qR, int ti=1)

{
    if(tR<qL || tL>qR)      return 0;                      //Out of bound
    if(tL>=qL && tR <= qR)    return tree[ti];           //The full range is taken
    int mid = (tL + tR)/2;
    int L = query(tL, mid, qL, qR, 2*ti);
    int R = query(mid+1, tR, qL, qR, 2*ti+1);
    return L+R
}
```

# UPDATE



- Update the  $k$ -th index of  $a$  with  $v$ 
  - Update 3<sup>th</sup> index of  $a$  with 6
  - Which nodes will be updated in tree?
    - All the nodes in the path from root to that leaf node which indicates  $[k \ k]$  as the range gets updated in each update query
      - The leaf node gets updated with the updated value of  $a[k]$  Means  $tree[ti]=a[k]$ ;
      - The other nodes in the path get updated with the sum of the value of it's left child and right child means  $tree[ti]=tree[2*ti]+tree[2*ti+1]$ ;



# UPDATE

---

```
void update(int k, int i=0, int j=n-1, int ti=1)    //k must be a valid index means k must be from 0 to n-1  
    if(i==j)  tree[ti]=a[k];  
    int mid = (i+j)/2;  
    if(k<=mid)  update(k, i, mid, 2*ti);  
    else  update(k, mid+1, j, 2*ti+1);  
    tree[ti] = tree[2*ti] + tree[2*ti+1];  
}
```

} From the current node we choose either left or right tree where [k k] falls in

- We traverse a single path from root to leaf for updating
- Length of a path can be at most  $h$  ( $\log_2 n$ )
- So the complexity for a update query is  $\log_2 n$

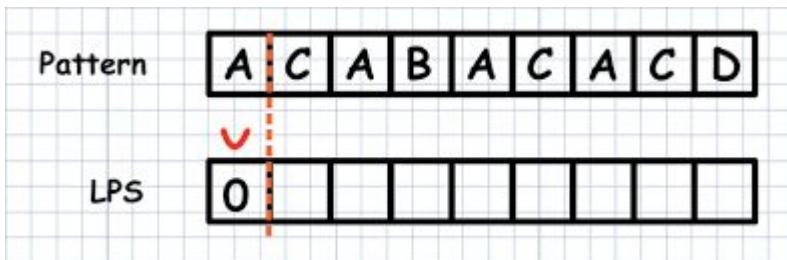


# Thanks

---

# CSE-203: Data Structures & Algorithms I

## Topic: KMP (Knuth-Morris-Pratt)



Nafiz Imtiaz Khan  
Lecturer, Department of CSE, MIST  
Email: nafiz@cse.mist.ac.bd



# String Matching Algorithm

---

**String-matching algorithms**, are an important class of string algorithms that try to find a place where one or several strings (also called **patterns**) are found within a larger string or text.



# Uses

---

- ❑ Database Schema
- ❑ Network Systems



# Classification

---

Exact String Matching Algorithms

Approximate String Matching Algorithms



# Exact String Matching Algorithms

---

Exact string matching algorithms is to find one, several, or all occurrences of a defined string (**pattern**) in a large string (**text** or **sequences**) such that each matching is perfect. All alphabets of patterns must be matched to corresponding matched subsequence.

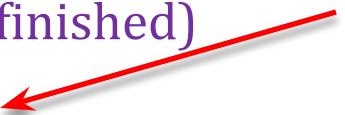


# Categories

- Algorithms based on character comparison
  - **Naïve Algorithm:** It slides the pattern (**length m**) over text (**length n**) one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches. (**already known**)
    - Time Complexity:  $O(m * n)$
  - **Using The TRIE Data Structure:** It is used as an efficient information retrieval data structure. It stores the keys in form of a balanced BST. (**already finished**)
  - **KMP (Knuth-Morris-Pratt) Algorithm:** The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.



# Categories

- Algorithms based on character comparison
  - **Naïve Algorithm:** It slides the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches. (already known)
  - **Using The TRIE Data Structure:** It is used as an efficient information retrieval data structure. It stores the keys in form of a balanced BST. (already finished)  

  - **KMP (Knuth-Morris-Pratt) Algorithm:** The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.



# Pattern Searching

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

A	A	B	A													
A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

Pattern Found at 0, 9 and 12



# Pattern Searching

## **Input:**

txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

## **Output:**

Pattern found at index 10

## **Input:**

txt[] = "AABAACACAADAABAABA"

pat[] = "AABA"

## **Output:**

Pattern found at index 0

Pattern found at index 9

Pattern found at index 12



# Limitations of Naïve Pattern Searching

The *Naïve Pattern Searching Algorithm* doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

`txt[] = "AAAAAAAAAAAAAAAAB"`

`pat[] = "AAAAB"`

`txt[] = "ABABABCABABABCABABABC"`

`pat[] = "ABABAC"` (not a worst case, but a bad case for Naive)



# KMP Algorithm

The *KMP matching algorithm* uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to  $O(n)$ .

Let's consider the following example:

text [ ]= "AAAAAABAAABA"

pattern [ ] = "AAAA"



# Lets' Start

Iterator i for text, j for pattern.

0    |    2    3    4    5    6    7    8    9    10

text:    

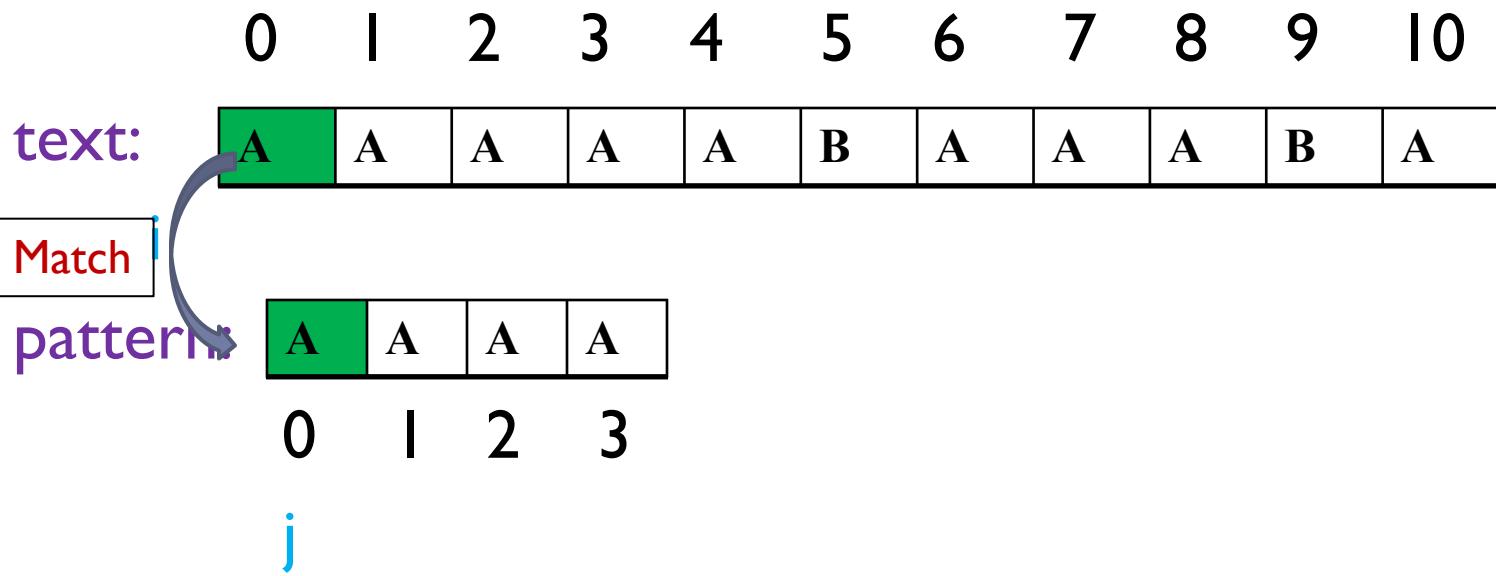
A	A	A	A	A	B	A	A	A	B	A
---	---	---	---	---	---	---	---	---	---	---

pattern:    

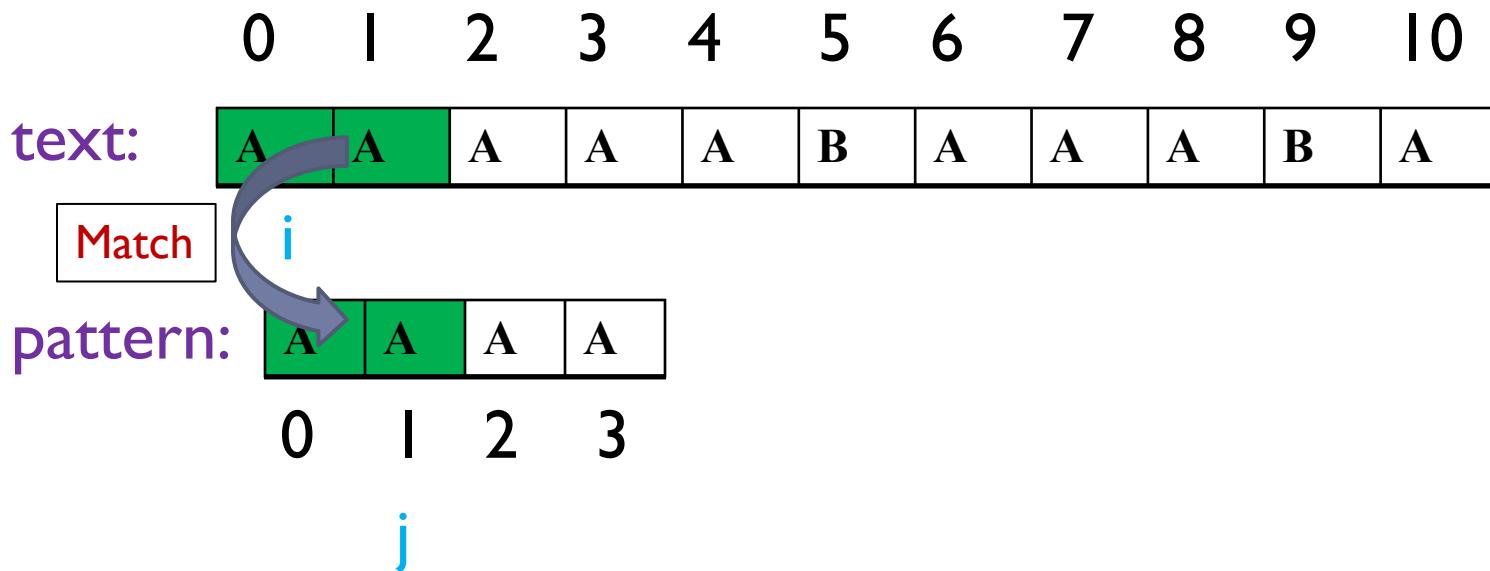
A	A	A	A
---	---	---	---

0    |    2    3

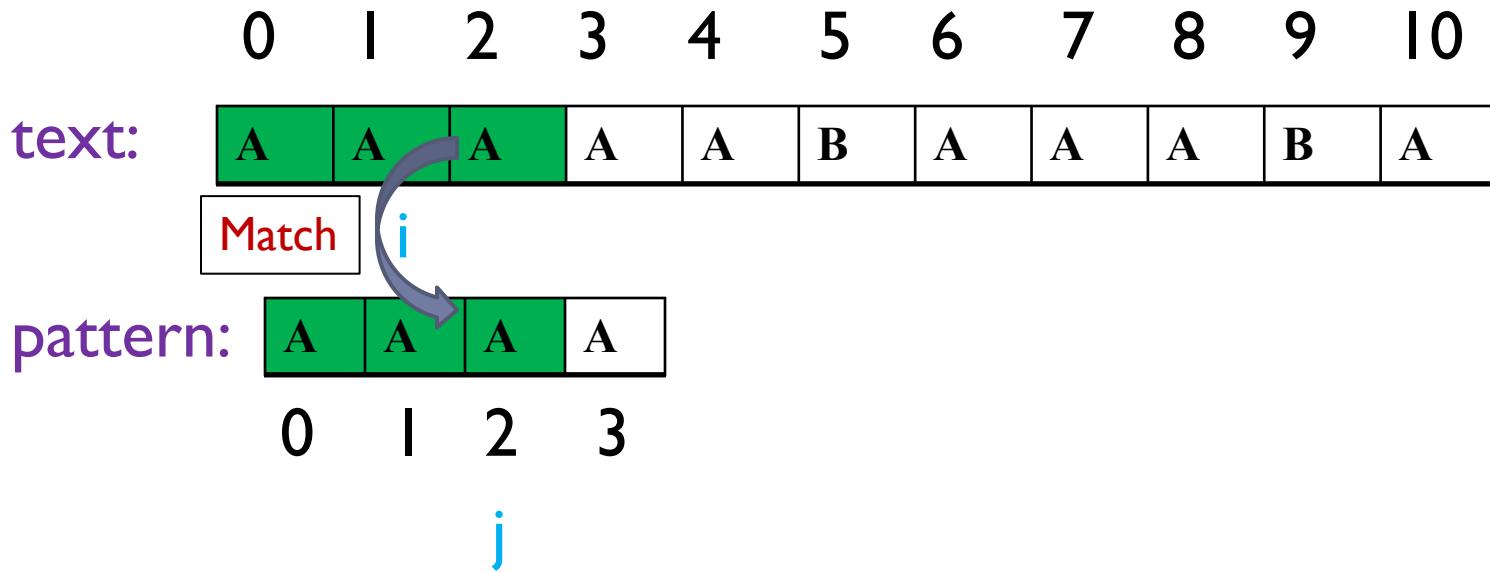
# Simulation



# Simulation

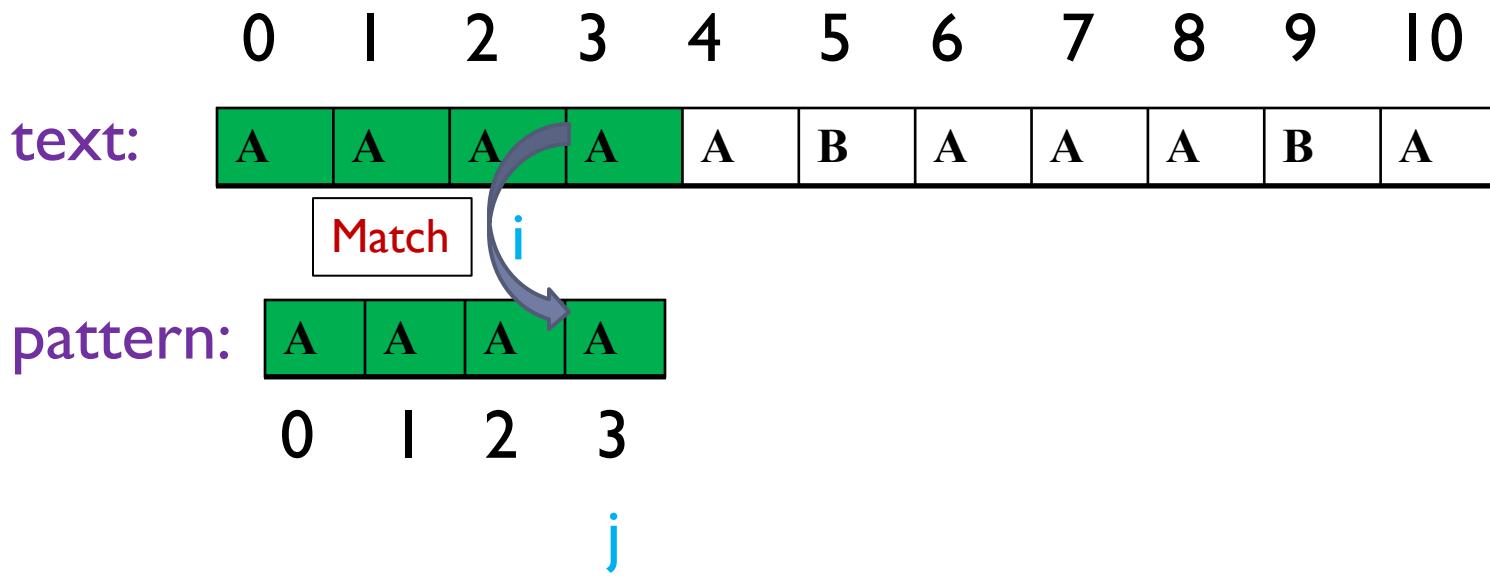


# Simulation



# Simulation

Till now simply Naïve process continues.

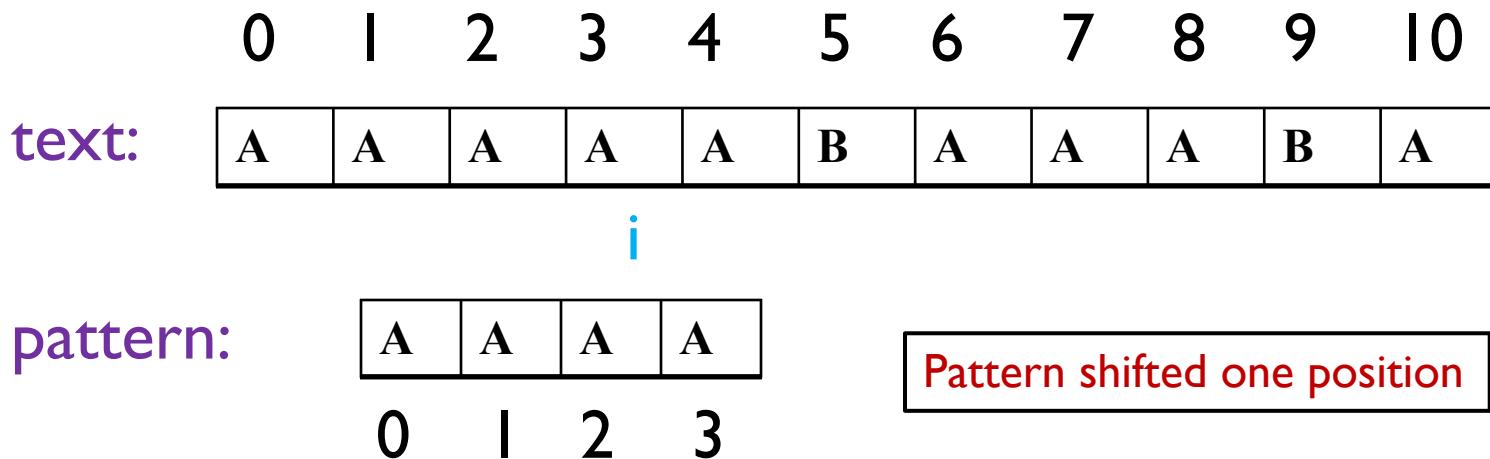


Pattern is matched from  $0^{\text{th}}$  position of text. But searching is still not finished.



# Simulation

This is where KMP does optimization over Naive.

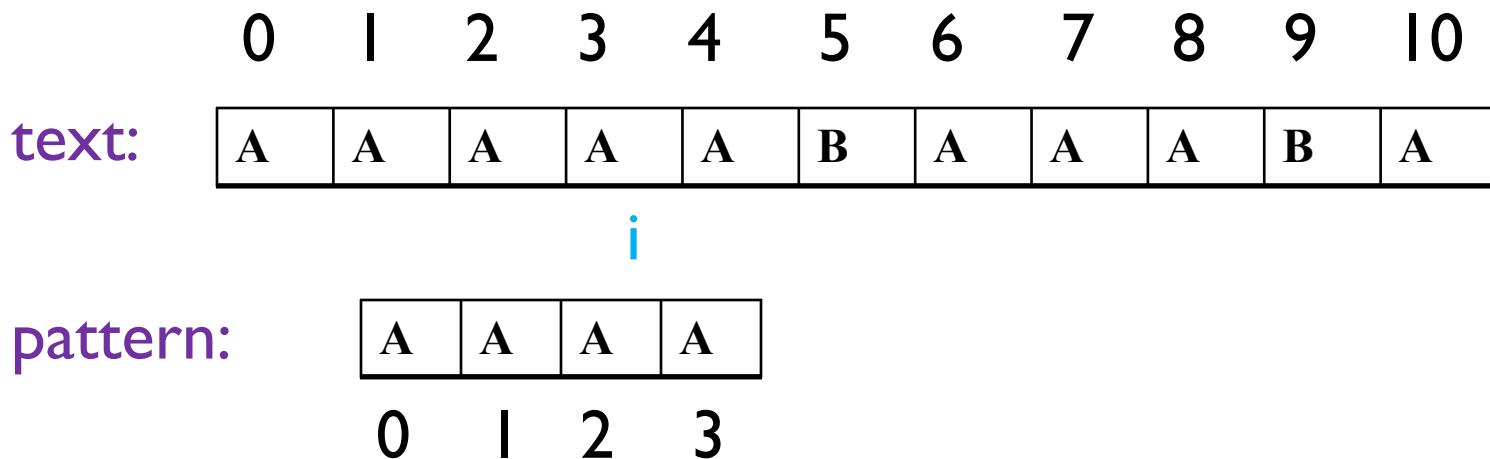


The question is from which position  $j$  will start?



# Simulation

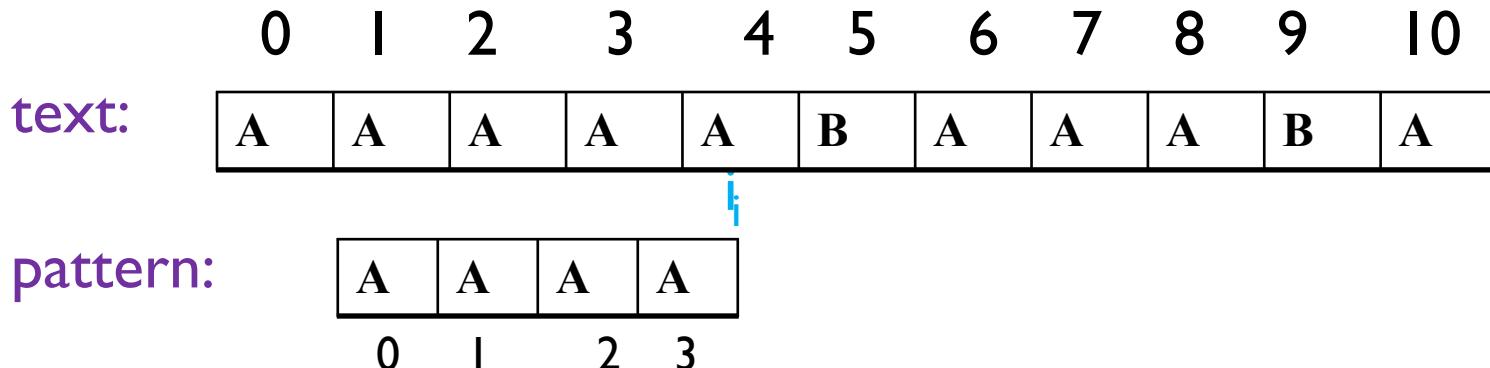
This is where KMP does optimization over Naive.



Whether j will start from 0 position or not?

# Simulation

In this second window, we only compare fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.





# Need of Preprocessing?

An important question arises from the above explanation, how to know how many characters to be skipped. To know this, pre-process pattern and prepare an integer **array lps[]** that tells the count of characters to be skipped.

To do it first, need to know the idea of prefix and suffix  
So,

If the pattern is “ABAC”

Then,

Prefix: A, AB, ABA, ABAC (**not proper prefix as it is same as the pattern**)

Suffix: C, AC, BAC, ABAC (**not proper suffix as it is same as the pattern**)



# Preprocessing Overview

- KMP algorithm preprocesses `pat[]` and constructs an auxiliary `lps[]` of size  $m$  (**same as size of pattern**) which is used to **skip** characters while matching.
- **name lps indicates longest proper prefix which is also suffix..** A proper prefix is prefix with whole string **not** allowed.
- Search for `lps` in sub-patterns. More clearly focus on **sub-strings** of patterns that are either **prefix and suffix**.
- For each sub-pattern  $\text{pat}[0..i]$  where  $i = 0$  to  $m-1$ , `lps[i]` stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern `pat[0..i]`.

`lps[i]` = the longest proper prefix of  $\text{pat}[0..i]$   
which is also a suffix of  $\text{pat}[0..i]$ .

**Note :** `lps[i]` could also be defined as longest prefix which is also proper suffix. Need to use properly at one place to make sure that the whole substring is not considered.



# Preprocessing Algorithm

In the preprocessing part, calculate values in `lps[]`.

To do that, keep track of **the length of the longest prefix suffix** value (use `len` variable for this purpose) for the previous index.

Initialize `lps[0]` and `len` as **0**.

If `pat[len]` and `pat[i]` match,

**increment** `len` by **1**

**assign** the incremented value to `lps[i]`.

If `pat[i]` and `pat[len]` do not match

`len` is not **0**

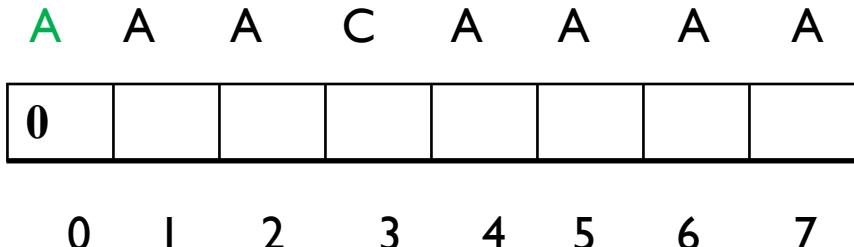
**update** `len` to `lps[len-1]`.

# Illustration

$\text{pat}[] = \text{"AAACAAAAA"}$

$\text{len} = 0, i = 0$ .  **$\text{lps}[0]$  is always 0**,

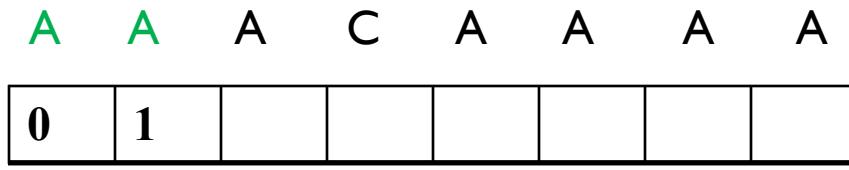
move to  $i = 1$



$\text{len} = 0, i = 1$ .

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[i]$  match, do  $\text{len}++$ ,  
store it in  $\text{lps}[i]$  and do  $i++$ .

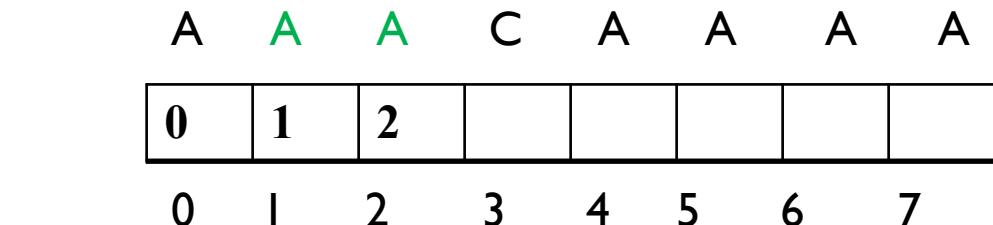
$\text{len} = 1, \text{lps}[1] = 1, i = 2$



$\text{len} = 1, i = 2$ .

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[i]$  match, do  $\text{len}++$ ,  
store it in  $\text{lps}[i]$  and do  $i++$ .

$\text{len} = 2, \text{lps}[2] = 2, i = 3$



# Illustration continues...

$\text{len} = 2, \text{i} = 3.$

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[\text{i}]$  do not match, and  $\text{len} > 0$ ,  
set  $\text{len} = \text{lps}[\text{len}-1] = \text{lps}[1] = 1$

A A A C A A A A

0	1	2					
---	---	---	--	--	--	--	--

$\text{len} = 1, \text{i} = 3.$

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[\text{i}]$  do not match and  $\text{len} > 0$ ,  
 $\text{len} = \text{lps}[\text{len}-1] = \text{lps}[0] = 0$

0 I 2 3 4 5 6 7

A A A C A A A A

0	1	2					
---	---	---	--	--	--	--	--

$\text{len} = 0, \text{i} = 3.$

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[\text{i}]$  do not match and  $\text{len} = 0$ ,  
Set  **$\text{lps}[3] = 0$**  and  $\text{i} = 4.$

4 5 6 7

A A A C A A A A

0	1	2	0				
---	---	---	---	--	--	--	--

# Illustration continues...

$\text{len} = 0, \text{i} = 4.$

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[\text{i}]$  match, do  $\text{len}++$ ,  
store it in  $\text{lps}[\text{i}]$  and do  $\text{i}++.$

$\text{len} = 1, \text{lps}[4] = 1,$

$\text{i} = 5 \text{ len} = 1, \text{i} = 5.$

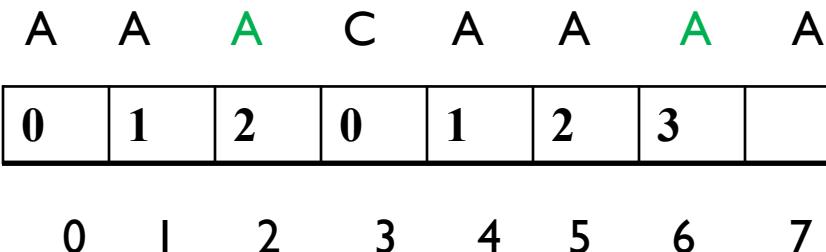
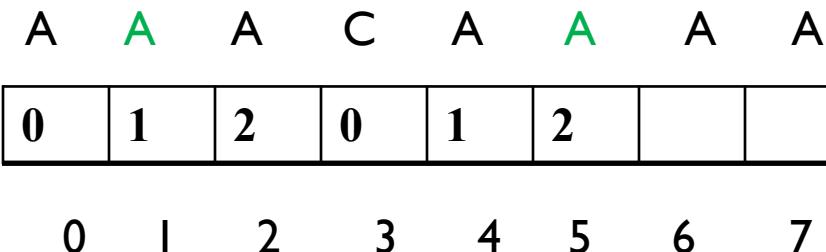
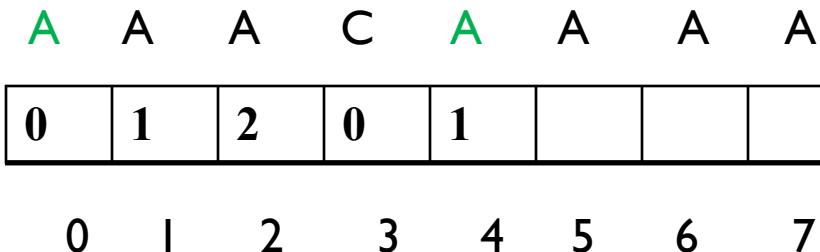
Since  $\text{pat}[\text{len}]$  and  $\text{pat}[\text{i}]$  match, do  $\text{len}++$ ,  
store it in  $\text{lps}[\text{i}]$  and do  $\text{i}++.$

$\text{len} = 2, \text{lps}[5] = 2,$

$\text{i} = 6 \text{ len} = 2, \text{i} = 6.$

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[\text{i}]$  match, do  $\text{len}++$ ,  
store it in  $\text{lps}[\text{i}]$  and do  $\text{i}++.$

$\text{len} = 3, \text{lps}[6] = 3,$



## Illustration continues...

$i = 7$   $\text{len} = 3$ ,  $i = 7$ .

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[i]$  do not match and  $\text{len} > 0$ ,  
set  $\text{len} = \text{lps}[\text{len}-1] = \text{lps}[2] = 2$

A A A C A A A A

0	1	2	0	1	2	3	
---	---	---	---	---	---	---	--

0 1 2 3 4 5 6 7

$\text{len} = 2$ ,  $i = 7$ .

Since  $\text{pat}[\text{len}]$  and  $\text{pat}[i]$  match, do  $\text{len}++$ ,  
store it in  $\text{lps}[i]$  and do  $i++$ .

A A A C A A A A

0	1	2	0	1	2	3	3
---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7

**i = 8** We stop here as we have constructed the whole  $\text{lps}[]$ .



# Example of lps[] construction

---

pattern “AAAA”,

lps[]: [0, 1, 2, 3]

pattern “ABCDE”,

lps[]: [0, 0, 0, 0, 0]

pattern “AABAACAAABAA”,

lps[]: [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

pattern “AACACAAAAAC”,

lps[]: [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

pattern “AAABAAA”,

lps[]: [0, 1, 2, 0, 1, 2, 3]



# Now Searching

---

Unlike *Naïve Algorithm*, where slide the pattern by one and compare all characters at each shift,

use a value from `lps[]` to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

- How to use `lps[]` to decide next positions (or to know a number of characters to be skipped)?



# Searching Algorithm

- Start comparison of  $\text{pat}[j]$  with  $j = 0$  with characters of current window of text.
- Keep matching characters  $\text{txt}[i]$  and  $\text{pat}[j]$  and keep incrementing  $i$  and  $j$  while  $\text{pat}[j]$  and  $\text{txt}[i]$  keep **matching**.
- When a **mismatch found**
  - We know that characters  $\text{pat}[0..j-1]$  match with  $\text{txt}[i-j\dots i-1]$  (Note that  $j$  starts with 0 and increment it only when there is a match).
  - We also know (from above definition) that  $\text{lps}[j-1]$  is count of characters of  $\text{pat}[0\dots j-1]$  that are **both** proper prefix and suffix.
  - From above two points, we can conclude that we **do not need to match** these  $\text{lps}[j-1]$  characters with  $\text{txt}[i-j\dots i-1]$  because we know that these characters will anyway match. Let us consider mentioned example to understand this.



# Searching Illustration

`txt[] = "AAAAAABAAAB"`

`pat[] = "AAAA"`

`lps[] = {0, 1, 2, 3}`

`i = 0, j = 0`

`txt[] = "AAAAAABAAABA"`

`pat[] = "AAAA"`

`txt[i] and pat[j] match, do i++, j++`

`i = 1, j = 1`

`txt[] = "AAAAAABAAABA"`

`pat[] = "AAAA"`

`txt[i] and pat[j] match, do i++, j++`



# Searching Illustration continues..

i = 2, j = 2

txt[] = "AAAAABAABA"

pat[] = "AAAA"

pat[i] and pat[j] match, do i++, j++

i = 3, j = 3

txt[] = "AAAAAABAABA"

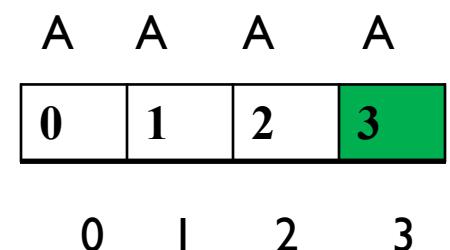
pat[] = "AAAA"

txt[i] and pat[j] match, do i++, j++

i = 4, j = 4

Since j == M, print **pattern found** and reset j,

j = lps[j-1] = lps[3] = 3





# Searching Illustration continues..

Here unlike Naive algorithm, do not match first three characters of this window. Value of  $\text{lps}[j-1]$  (in above step) gave us index of next character to match.

$i = 4, j = 3$

$\text{txt}[] = \text{"AAAAAABAABA"}$

$\text{pat}[] = \text{"AAA A"}$

$\text{txt}[i]$  and  $\text{pat}[j]$  match, do  $i++, j++$

$i = 5, j = 4$  Since  $j == M$ , print **pattern found** and reset  $j$ ,

$j = \text{lps}[j-1] = \text{lps}[3] = 3$

A	A	A	A
0	1	2	3

0    |    1    2    3



# Searching Illustration continues..

Again unlike Naive algorithm, we do not match first three characters of this window. Value of  $\text{lps}[j-1]$  (in above step) gave us index of next character to match.

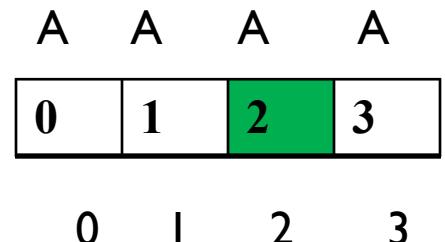
$i = 5, j = 3$

$\text{txt}[] = \text{"AAAAA}\text{\color{red}B}\text{AAABA"}$

$\text{pat}[] = \text{"AAA}\text{\color{red}A"}$

$\text{txt}[i]$  and  $\text{pat}[j]$  do NOT match and  $j > 0$ , change only  $j$

$j = \text{lps}[j-1] = \text{lps}[2] = 2$



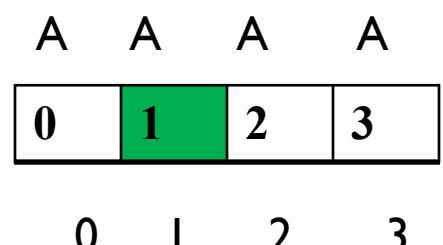
$i = 5, j = 2$

$\text{txt}[] = \text{"AAAAA}\text{\color{red}B}\text{AAABA"}$

$\text{pat}[] = \text{"AAA}\text{\color{red}A"}$

$\text{txt}[i]$  and  $\text{pat}[j]$  do NOT match and  $j > 0$ , change only  $j$

$j = \text{lps}[j-1] = \text{lps}[1] = 1$



# Searching Illustration continues..

i = 5, j = 1

txt[] = "AAAAA**B**AAABA"

pat[] = "**AAAA**"

txt[i] and pat[j] do NOT match and j > 0, change only j

j = lps[j-1] = lps[0] = 0

A	A	A	A
0	1	2	3
0		2	3

i = 5, j = 0

txt[] = "AAAAA**B**AAABA"

pat[] = "**AAAA**"

txt[i] and pat[j] do NOT match and j is 0, do i++.

i = 6, j = 0

txt[] = "AAAAAB**AAABA**"

pat[] = "**AAAA**"

txt[i] and pat[j] match, do i++ and j++



# Searching Illustration continues..

i = 7, j = 1

txt[] = "AAAAAABAAABA"

pat[] = "AAAA"

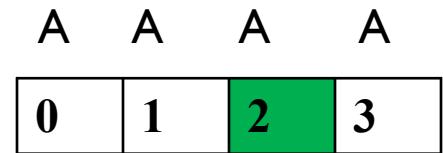
txt[i] and pat[j] match, do i++ and j++

i = 8, j = 2

txt[] = "AAAAAABAAABA"

pat[] = "AAAA"

txt[i] and pat[j] match, do i++ and j++



i = 9, j = 3

txt[] = "AAAAAABAAABA"

pat[] = "AAAA"

txt[i] and pat[j] do not match, match and j > 0, change only j

j = lps[j-1] = lps[3-1] = 2

0 1 2 3



# Searching Ends

---

$i = 10, j = 2$

$txt[] = "AAAAAABAAAB\textcolor{red}{A}"$

$pat[] = "\textcolor{red}{AAAA}A"$

$txt[i]$  and  $pat[j]$  match, do  $i++$  and  $j++$



# Outcome

---

As the text length has finished, searching will be stopped here.

## Result:

Pattern “AAAA” has been found in “AAAAAABAAABA”

at 0 position

at 1 position



# Time Complexity

---

- LPS table formation:  $O(m)$
- And searching :  $O(n)$
- So, complexity is  $O(n + m)$

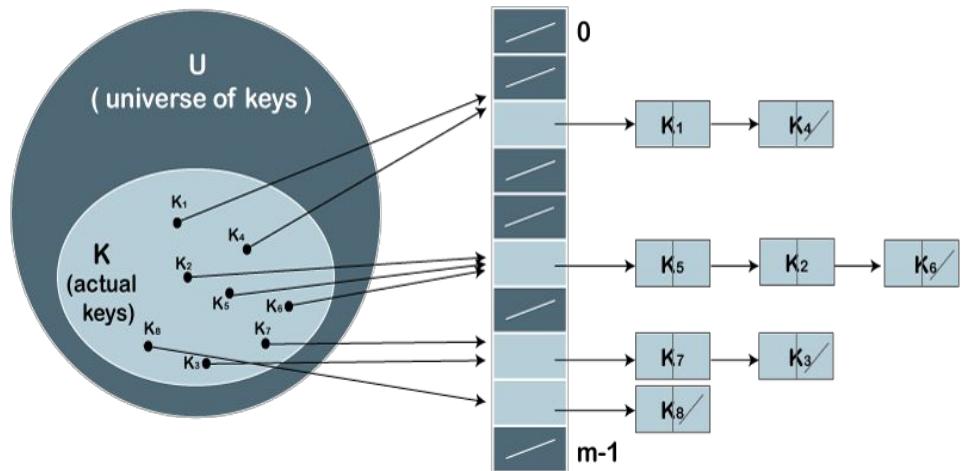


---

# THANK YOU

# CSE-203: Data Structures & Algorithms I

## Topic: Hash Tables



Nafiz Imtiaz Khan  
Lecturer, Department of CSE, MIST  
Email: nafiz@cse.mist.ac.bd



# Dictionary

- **Dictionary:**

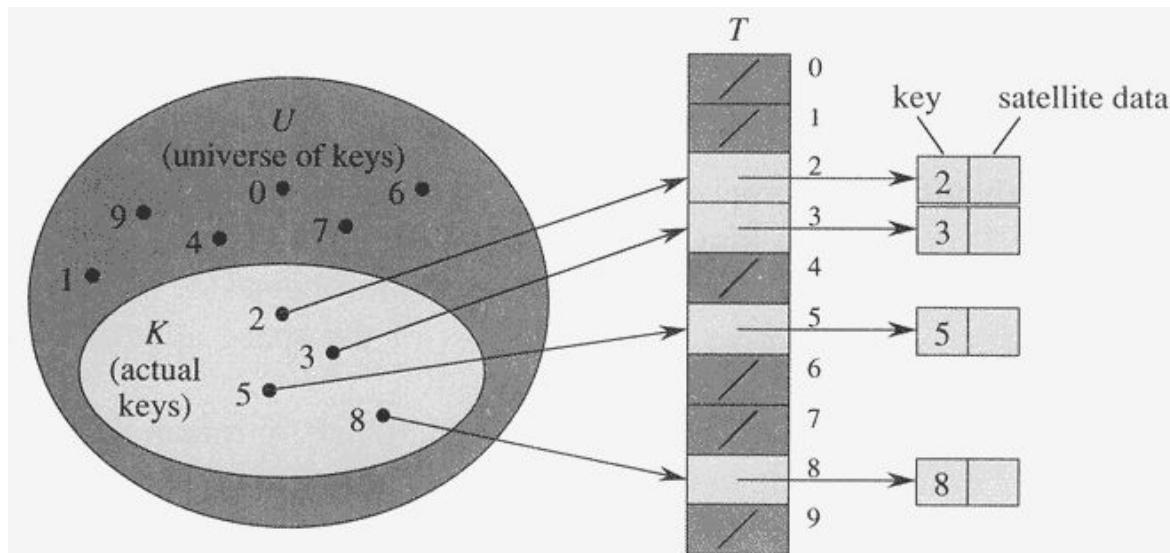
- Dynamic-set data structure for **storing** items indexed using *keys*.
- Supports **operations** Insert, Search, and Delete.
- **Applications:**
  - Symbol table of a compiler.
  - Memory-management tables in operating systems.
  - Large-scale distributed systems.

- **Hash Tables:**

- Effective way of implementing dictionaries.
- Generalization of ordinary arrays.

# Direct-Address Tables

- Direct-address Tables are **ordinary arrays**
- **Facilitate direct addressing**
  - Element whose key is  $k$  is obtained by indexing into the  $k^{\text{th}}$  position of the array
- **Applicable** when we can afford to allocate an array with one position for every possible key
  - i.e. when the universe of keys  $U$  is small
- **Dictionary operations** can be implemented to take  $O(1)$  time





# Direct-Address Tables

- Suppose:
  - The range of keys is  $0..m-1$
  - Keys are distinct
- The idea:
  - Set up an array  $T[0..m-1]$  in which
    - $T[i] = x$  if  $x \in T$  and  $\text{key}[x] = i$
    - $T[i] = \text{NULL}$  otherwise
  - This is called a *direct-address table*
    - Operations take  $O(1)$  time !
    - *So what's the problem?*



# Direct-Address Tables

- Direct addressing works well when the range  $m$  of keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have  $2^{32}$  entries, more than 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range  $0..m-1$
- This mapping is called a *hash function*



# Hash Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We want these to be fast, but don't care about sorted order
- The structure we will use is a *hash table*
  - Supports all the above in  $O(1)$  expected time !



# Hash Tables

- **Notation:**
  - $U$  : Universe of all possible keys.
  - $K$  : Set of keys actually stored in the dictionary.
  - $|K| = n$ .
- When  $U$  is very large,
  - Arrays are not practical.
  - $|K| \ll |U|$ .
- Use a table of size proportional to  $|K|$  – **The hash tables.**
  - However, we lose the direct-addressing ability.
  - Define functions that map keys to slots of the hash table.



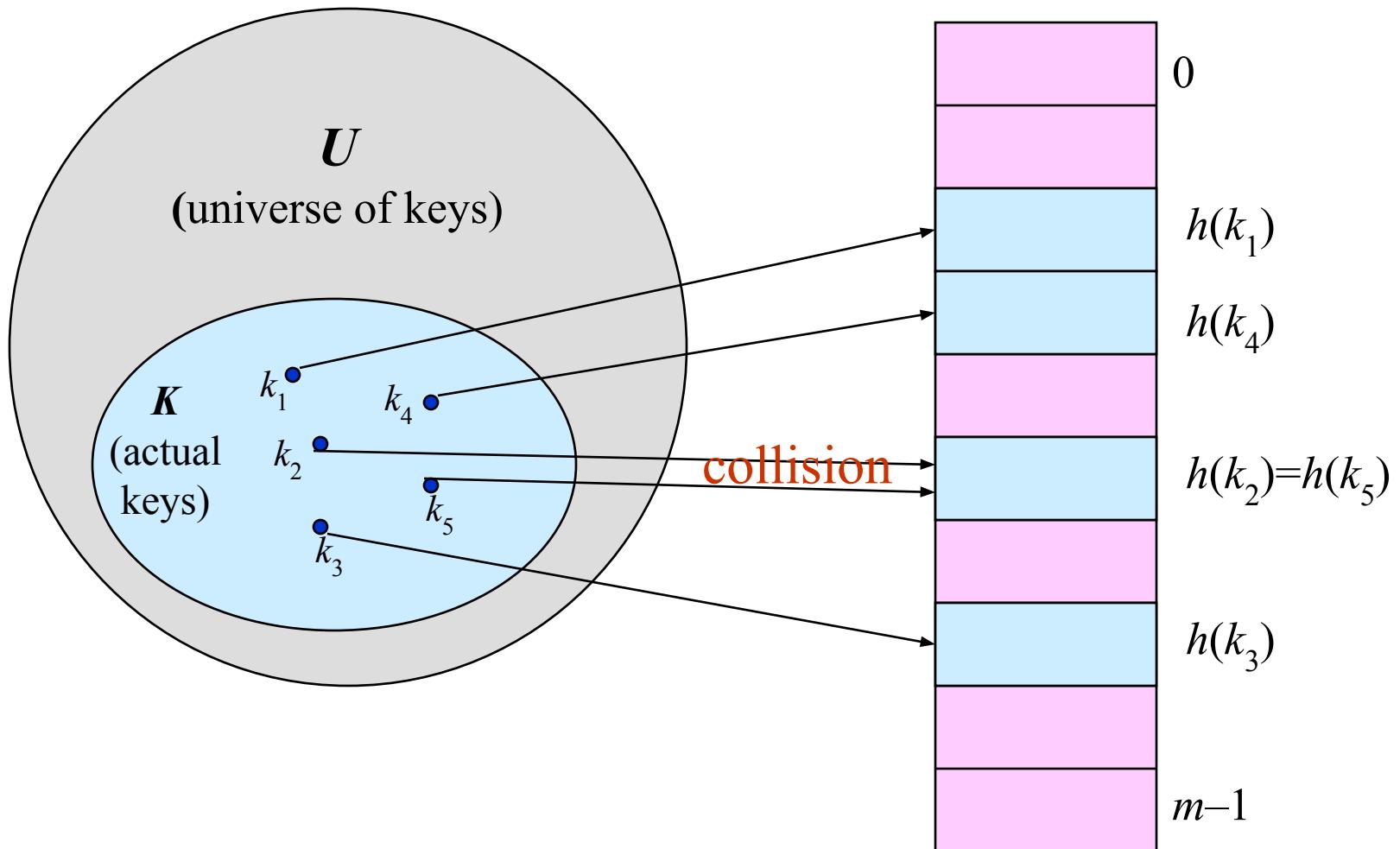
# Hashing

- Hash function  $h$ : Mapping from  $U$  to the slots of a hash table  $T[0..m-1]$ .

$$h : U \rightarrow \{0, 1, \dots, m-1\}$$

- With arrays, key  $k$  maps to slot  $A[k]$ .
- With hash tables, key  $k$  maps or “hashes” to slot  $T[h[k]]$ .
- $h[k]$  is the *hash value* of key  $k$ .

# Hashing



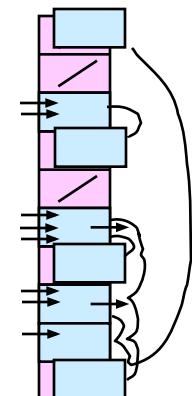
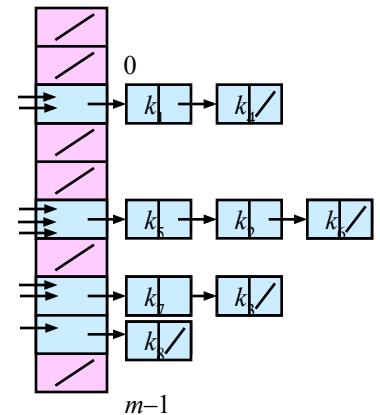


# Issues with Hashing

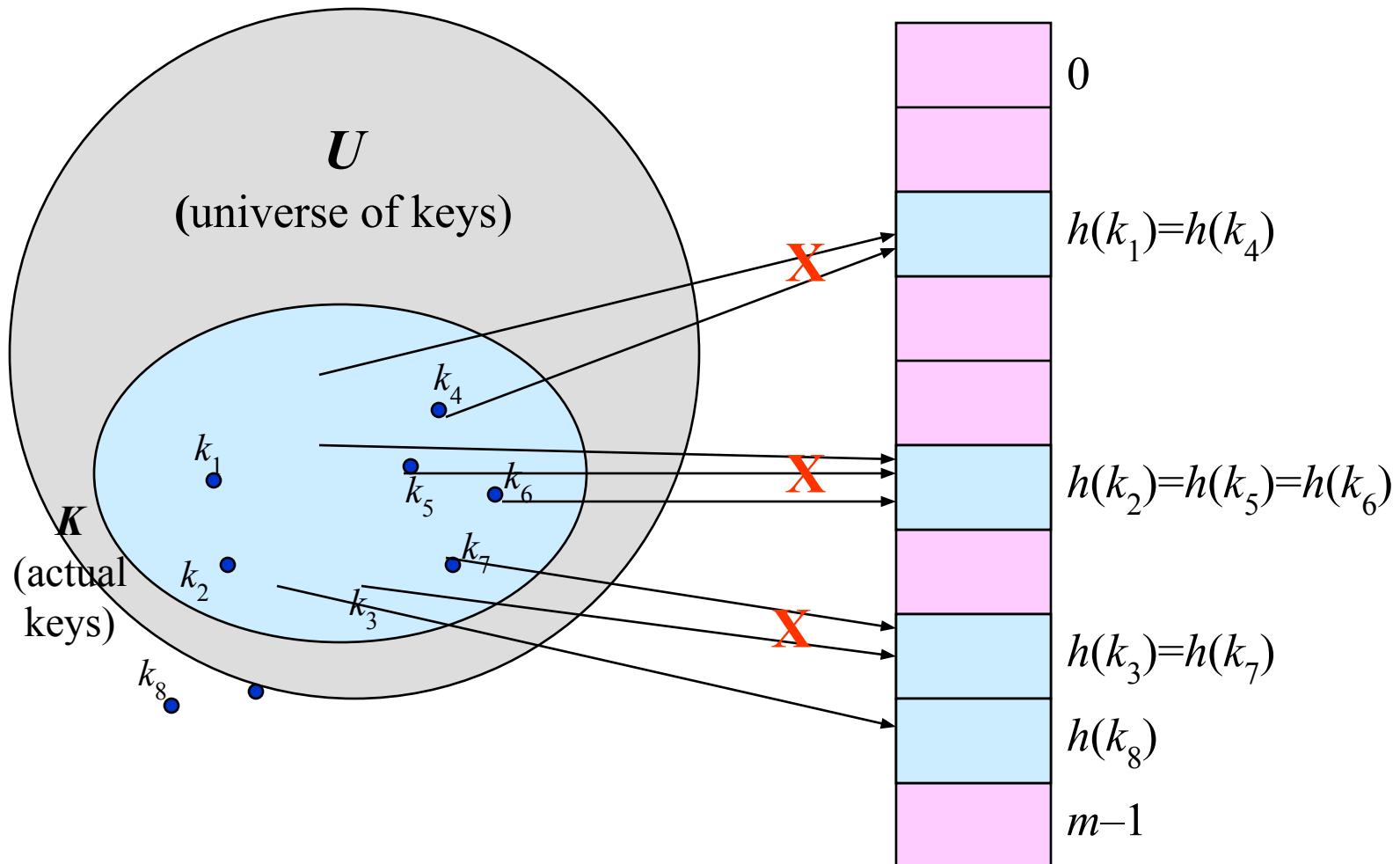
- Multiple keys can hash to the same slot – **collisions are possible.**
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost  $\Theta(n)$  time in the worst case.
  - However, all operations can be made to have an expected complexity of  $\Theta(1)$ .

# Methods of Resolution

- **Chaining:**
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.
- **Open Addressing:**
  - All elements are stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.

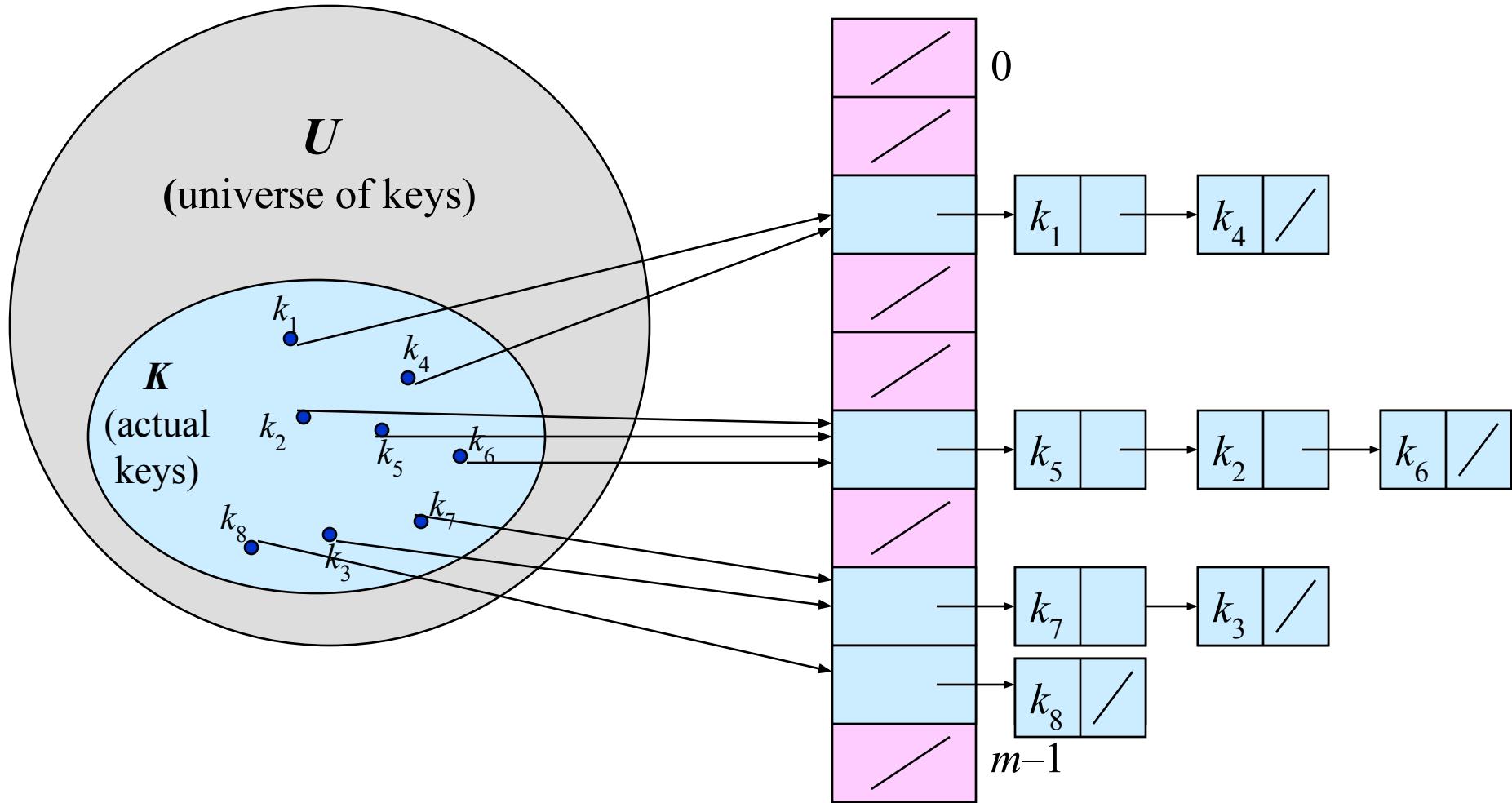


# Collision Resolution by Chaining



# Collision Resolution by Chaining

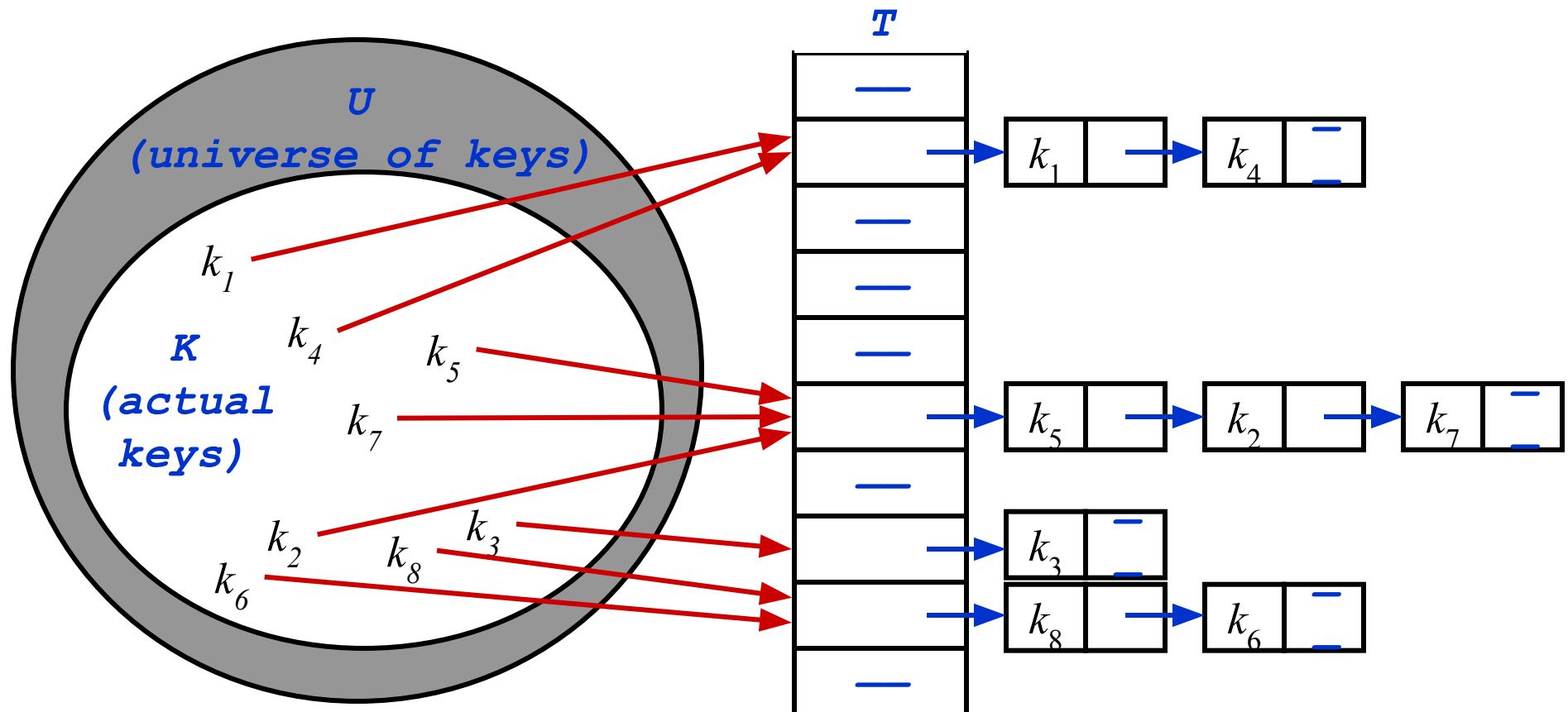
- Chaining puts elements that hash to the same slot in a linked list:



# Chaining

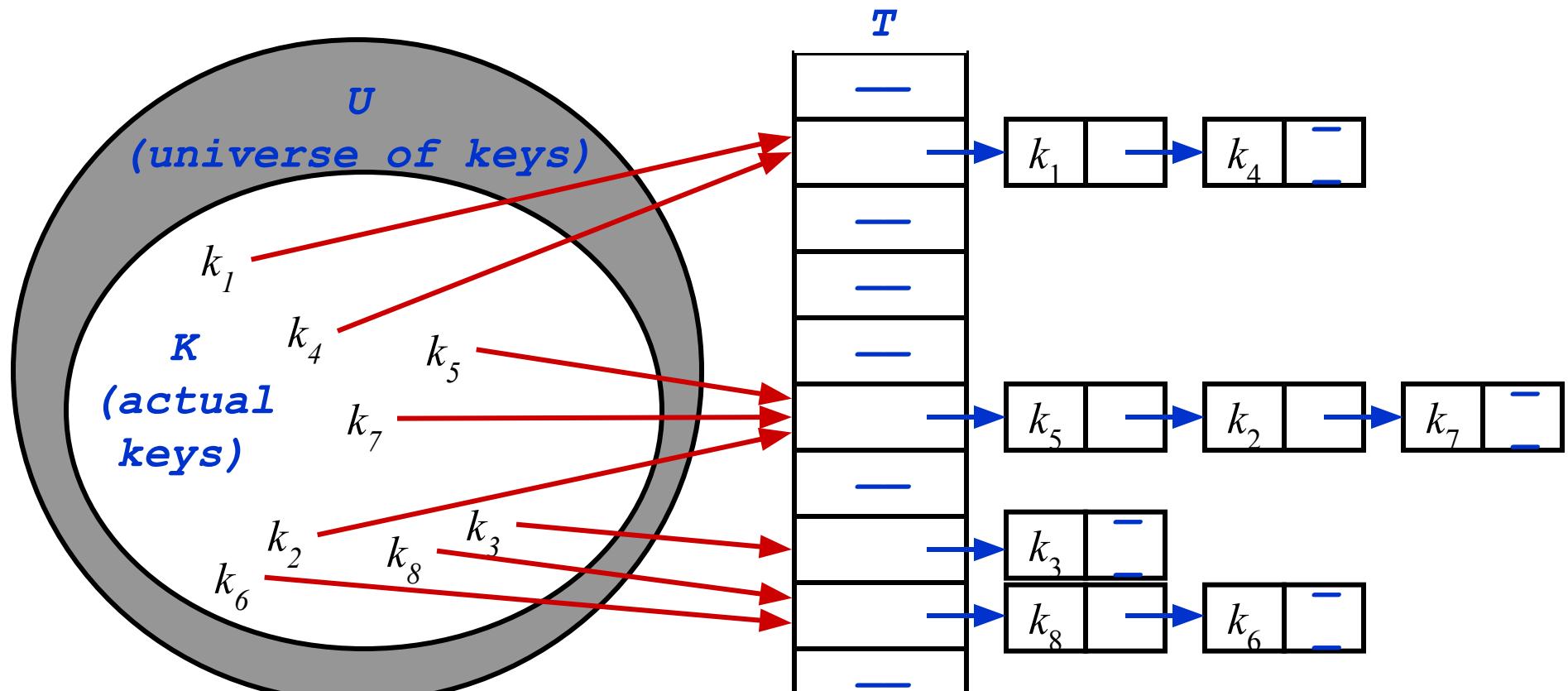


# *How do we insert an element?*



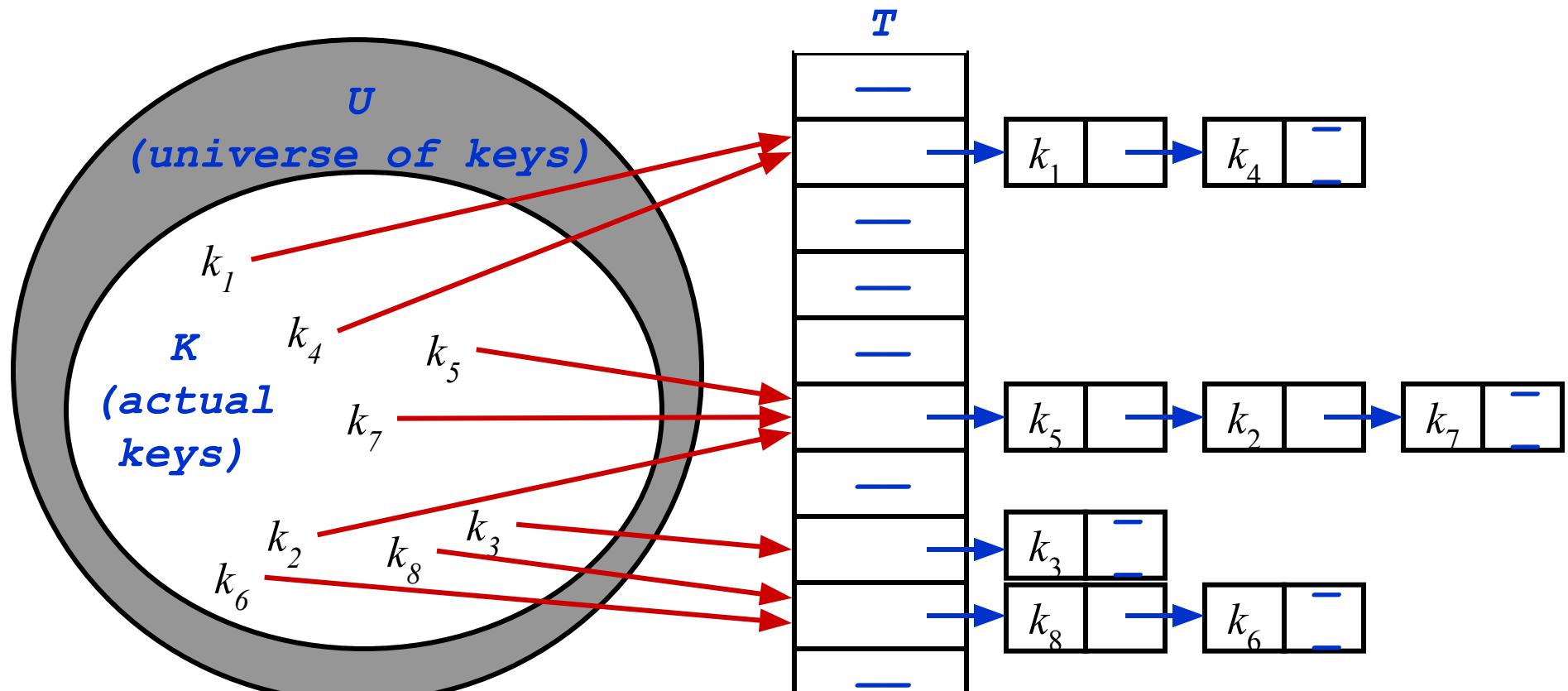
# Chaining

- How do we delete an element?



# Chaining

- How do we search for an element with a given key?





# Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given  $n$  keys and  $m$  slots in the table: the *load factor*  $\alpha = n/m$  = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?*



# Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given  $n$  keys and  $m$  slots in the table, the *load factor*  $\alpha = n/m$  = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A:  $O(1+\alpha)$



# Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given  $n$  keys and  $m$  slots in the table, the *load factor*  $\alpha = n/m$  = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A:  $O(1+\alpha)$
- *What will be the average cost of a successful search?*



# Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given  $n$  keys and  $m$  slots in the table, the *load factor*  $\alpha = n/m$  = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A:  $O(1+\alpha)$
- *What will be the average cost of a successful search?* A:  $O(1 + \alpha/2) = O(1 + \alpha)$



# Analysis of Chaining

Draw the 11-item hash table that results from using the hash function  $h(k) = (2k + 5) \text{ mod } 11$ , to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.



# Open Addressing

- Basic idea:
  - To insert: if slot is full, try another slot, ..., until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking
- Table needn't be much bigger than  $n$

# Probe Sequence

- Sequence of slots examined during a key search constitutes a ***probe sequence***.
- Probe sequence must be a permutation of the slot numbers.
  - We examine every slot in the table, if we have to.
  - We don't examine any slot more than once.
- The hash function is extended to:
  - $$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$
  
$$\underbrace{\hspace{10em}}_{\text{probe number}} \qquad \underbrace{\hspace{10em}}_{\text{slot number}}$$
- $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$  should be a permutation of  $\langle 0, 1, \dots, m - 1 \rangle$ .

# Computing Probe Sequences

- The ideal situation is ***uniform hashing***:
  - Generalization of simple uniform hashing.
  - Each key is equally likely to have any of the  $m!$  permutations of  $\langle 0, 1, \dots, m - 1 \rangle$  as its probe sequence.
- It is **hard to implement** true uniform hashing.
  - **Approximate** with techniques that at least guarantee that the probe sequence is a permutation of  $\langle 0, 1, \dots, m - 1 \rangle$ .
- Some techniques:
  - Use ***auxiliary hash functions***.
    - Linear Probing.
    - Quadratic Probing.
    - Double Hashing.
  - Can't produce all  $m!$  probe sequences.



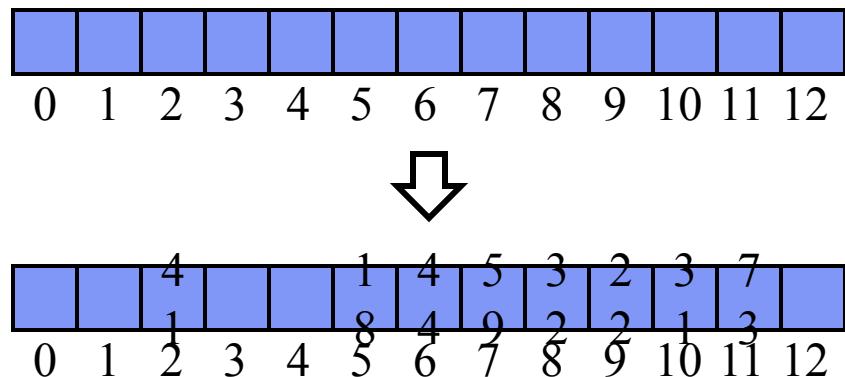
# Linear Probing

- $$h(k, i) = (h'(k) + i) \text{ mod } m.$$

ke      Probe      Auxiliary hash function  
      y      number
- The initial probe determines the entire probe sequence.
  - $T[h'(k)], T[h'(k)+1], \dots, T[m-1], T[0], T[1], \dots, T[h'(k)-1]$
  - Hence, only  $m$  distinct probe sequences are possible.
- Suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by  $i$  full slots gets filled next with probability  $(i+1)/m$ .
  - Hence, average search and insertion times increase.

# Ex: Linear Probing

- Example:
  - $h'(k) = k \bmod 13$
  - $h(k, i) = (h'(k) + i) \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



# Operation Insert

- Act as though we were searching, and insert at the first NIL slot found.

Hash-Insert( $T, k$ )

1.  $i \leftarrow 0$
2. **repeat**  $j \leftarrow h(k, i)$
3.       **if**  $T[j] = \text{NIL}$
4.           **then**  $T[j] \leftarrow k$
5.           **return**  $j$
6.       **else**  $i \leftarrow i + 1$
7. **until**  $i = m$
8. **error** “hash table overflow”



# Pseudo-code for Search

## Hash-Search ( $T, k$ )

1.  $i \leftarrow 0$
2. **repeat**  $j \leftarrow h(k, i)$
3.       **if**  $T[j] = k$
4.           **then return**  $j$
5.         $i \leftarrow i + 1$
6. **until**  $T[j] = \text{NIL}$  **or**  $i = m$
7. **return**  $\text{NIL}$



# Deletion

- Cannot just turn the slot containing the key we want to delete to contain NIL. Why?
- Use a special value **DELETED** instead of NIL when marking a slot as empty during deletion.
  - **Search** should treat DELETED as though the slot holds a key that does not match the one being searched for.
  - **Insert** should treat DELETED as though the slot were empty, so that it can be reused.
- **Disadvantage:** Search time is no longer dependent on  $\alpha$ .
  - Hence, chaining is more common when keys have to be deleted.

# Quadratic Probing

- $$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad c_1 \neq c_2$$

key
Probe number
Auxiliary hash function
- The initial probe position is  $T[h'(k)]$ , later probe positions are offset by amounts that depend on a quadratic function of the probe number  $i$ .
- Must constrain  $c_1$ ,  $c_2$ , and  $m$  to ensure that we get a full permutation of  $\langle 0, 1, \dots, m-1 \rangle$ .
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same.

# Double Hashing

- $$h(k, i) = (h_1(k) + i h_2(k)) \text{ mod } m$$

key
Probe
Auxiliary hash functions
- **Two auxiliary hash functions.**
  - $h_1$  gives the initial probe.  $h_2$  gives the remaining probes.
- Must have  $h_2(k)$  relatively prime to  $m$ , so that the probe sequence is a full permutation of  $\langle 0, 1, \dots, m - 1 \rangle$ .
  - Choose  $m$  to be a power of 2 and have  $h_2(k)$  always return an odd number. Or,
  - Let  $m$  be prime, and have  $1 < h_2(k) < m$ .
- **$\Theta(m^2)$  different probe sequences.**
  - One for each possible combination of  $h_1(k)$  and  $h_2(k)$ .
  - Close to the ideal uniform hashing.



# THANK YOU