CSx55: Distributed Systems [DHTs]

Routing in DHTs

So many, many
nodes and items
But the mapping's unambiguous
Deterministic to boot

Each node in the know
only about a few others
Messages relayed closer and closer
in a few bounded hops

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Is a prefix hop done for every hex value in the GUID?
- Would matching prefixes go all the way towards the end of the GUID?
- Where is the final determination made that a key needs to be stored a given node in Pastry?

Topics covered in this lecture

- Pastry [wrap-up]
- Tapestry
- Chord

Pastry: Host failure or departure

Detection and coping with node failures

- When a node's immediate neighbors (in the GUID space) cannot communicate with it?
  - The node is considered failed

- Necessary to **repair** leaf sets and routing tables that contain the failed GUID
  - Leaf sets are repaired *proactively*
  - Routing tables at the other nodes are updated on a "*when discovered basis*"

Repairing leaf sets

- Node that discovers the failure
  - Looks for a live node close to the failed node, and requests copy of that node's leaf set, L'
  - This should contain GUIDs that partly overlap those in the node that discovered failure
    - Include one that should replace the failed node
- Other neighboring nodes are informed
  - They perform a similar procedure

How many nodes must update their leaf sets?

- With a leaf set size of 4 (2 smaller, 2 larger), a node x appears only in the leaf sets of its 2 immediate predecessors and 2 immediate successors
  - If x fails, those 4 neighbors must update their leaf sets

- To generalize: A node x appears in the leaf sets of its ℓ predecessors and ℓ successors on the GUID ring
  - If node x fails, exactly $\ell + \ell = 2\ell$ neighbors must update their leaf sets

Locality

- Pastry's routing structure is redundant
  - Multiple routes between pairs of nodes
- Construction of routing tables tries to take advantage of this redundancy
  - Reduce message transmission times by exploiting locality properties of underlying network

Routing table: Exploiting locality. [1/2]

- In the routing table, each row contains 16 entries
  - Entries in the i^{th} row give addresses of 16 nodes with GUIDs with $i-1$ initial hexadecimal digits
  - I^{th} digit takes each of the possible hexadecimal values
- Well-populated Pastry system contains more nodes than can be contained in an individual routing table

Routing table: Exploiting locality. [2/2]

- When routing table is constructed, a choice is made for each position
  - Between multiple candidates
  - Based on *proximity* neighbor selection
- Locality metric
  - IP hops or measured latency

Performance of exploiting locality

- Since the information in the routing table is not comprehensive
  - Mechanism does not produce globally optimal routing
- Simulations show that
  - On average, the routing is 30-50% longer than the optimum

Coping with malicious nodes

- Small degree of *randomness* is introduced into route selection
- Randomized to yield a common prefix that is less than the maximum length
  - With a certain probability
- Routes are taken from an earlier row
  - Less optimal, but different than standard version
  - Client transmission succeeds in the presence of small numbers of malicious nodes

Into this house, we're born
Into this world, we're thrown
Like a dog without a bone
An actor out on loan
Riders on the storm

Riders on the Storm; Morrison, Densmore, Manzarek, Krieger; The Doors

TAPESTRY

Tapestry

- Routes messages to nodes based on GUIDs associated with the resources
  - Uses **prefix routing** in a manner similar to Pastry
- **160-bit** identifiers are used
  - To refer to both objects and nodes that perform routing actions
- For any resource with GUID G, there is a unique root node, with GUID R_G
  - R_G is *numerically closest* to G

Tapestry Routing [Summary]

- Uses local routing tables, which they also call **neighbor maps**, to route messages
- Routing is digit-by-digit
  - 4***→ 42** → 42A* → 42AD
- This longest prefix routing is also used by classless interdomain routing (CIDR)

Tapestry: Routing messages

- Each node maintains a **routing table**
  - Entries include nodeIDs and IP addresses

- This routing table has multiple levels
  - Each level contains links to nodes matching a prefix up to a digit position in the ID
  - The i^{th} entry in the j^{th} level at node **N**?
    - Location of the closest node which begins with the *prefix*(**N**, j-1) + i
    - E.g., 9th entry of the 4th level for node 325AE is ?
      - 3259

Tapestry Routing

- The router for the n^{th} hop
  - Shares a prefix of length $\geq n$ with the destination ID
  - Looks in its $(\mathrm{n+1})^{th}$ level map for entry matching the next digit in the destination ID

- Guarantees that any node in the system can be reached in at most $log\ N$ logical hops
  - N is the size of the ID space i.e. $\mathrm{N} = 2^{160}$

When a digit cannot be matched?

- Looks for a "close" digit in the routing table
- This approach is called **surrogate routing**
  - Results in mapping every identifier **G** to a unique root node G_R

Managing a dynamic environment

- Route reliably even when intermediate links are changing or faulty
- Exploit network **path diversity**
  - Via *redundant* routing paths
- Primary links are augmented by **backup-links**
  - Each sharing the same prefix

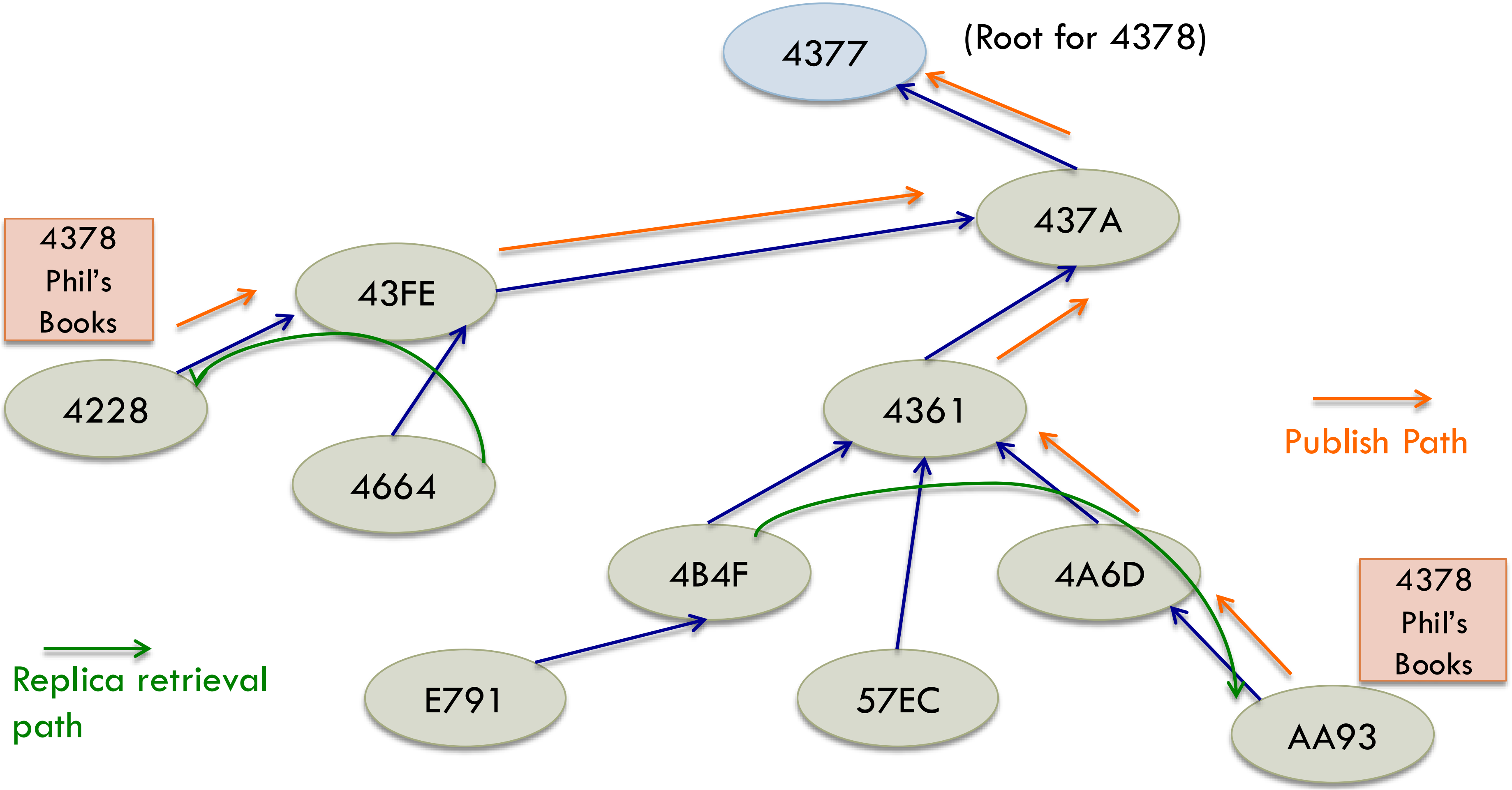
Managing multiple copies of the resource [1/2]

- Hosts H holding replicas of **G** periodically invoke *publish(G)*
  - Ensures that newly arrived hosts become aware of the existence of **G**
- On each invocation of *publish(G)*
  - Message is routed from invoker towards node R_G
  - On receipt of a publish message R_G enters (G, IP_H)
    - The mapping between **G** and IP address of H
  - Each node in the publication path caches the same mapping

Managing multiple copies of the resource [2/2]

- When nodes hold multiple (**G**, IP) mappings for the same GUID?
  - They are **sorted** by network distance to the IP address
- Results in *selection of nearest* available replica of the object

An example of managing replicas using Tapestry



My room is round when I lay down, when I wake up it's square
When I go outside it's on a spiral set of stairs
The people that surround me are waiting out there
In a round room they can't find me anywhere

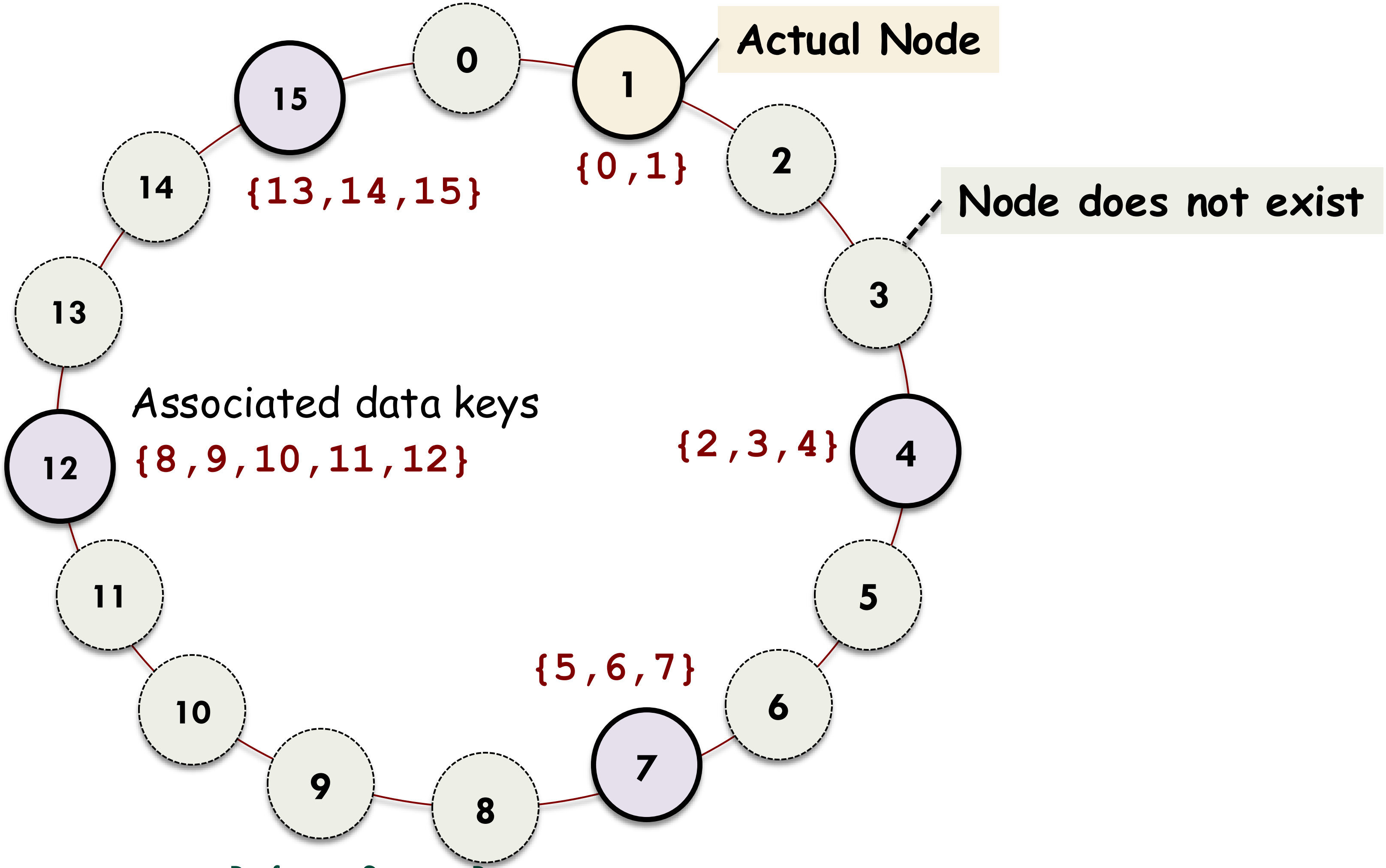
The Round Room; Mike Gordon; Phish

CHORD

The Chord System

- Assigns IDs to keys and nodes from the same 1-dimensional ID space
- Nodes are organized into a **ring**
- Data item with key **`k`** is mapped to a node with the **smallest** **`id`** ≥ **`k`**
  - Also referred to as `successor(`**`k`**`)`

Mapping of data items to nodes in Chord



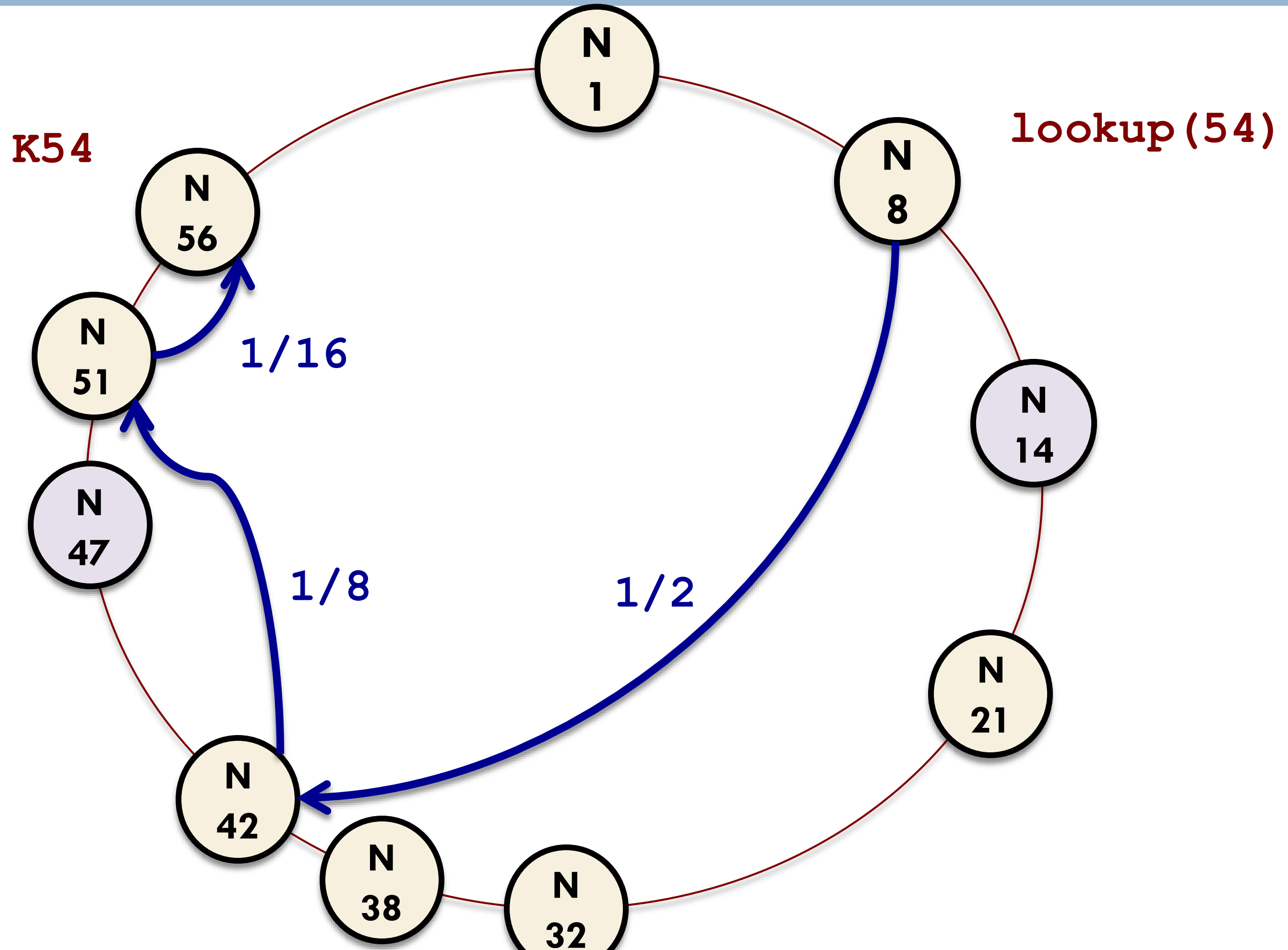
Chord lookups

- **N** is the number of possible nodes in the system
- Each node maintains a **finger table**
  - With log N entries
  - Entries contains IP addresses of nodes
    - Half-way around the ID space from it
    - $1/4^{th}$, $1/8^{th}$, … **in powers of two**
  - Ensures node can forward lookup query to at least ½ of the remaining ID-space distance to key
    - Lookups in $O(\log N)$

Storing keys and forwarding lookups

- An entity with key k falls under the ***jurisdiction*** of node with the *smallest identifier* id
  - $id >= k$
  - Referred to as the successor of k or $succ(k)$
- A node ***forwards*** query for key k to node (in its FT) with highest ID $\leq k$
  - The exception is ONLY when the first entry is greater than k
    - In this case, that node is responsible for storing that element

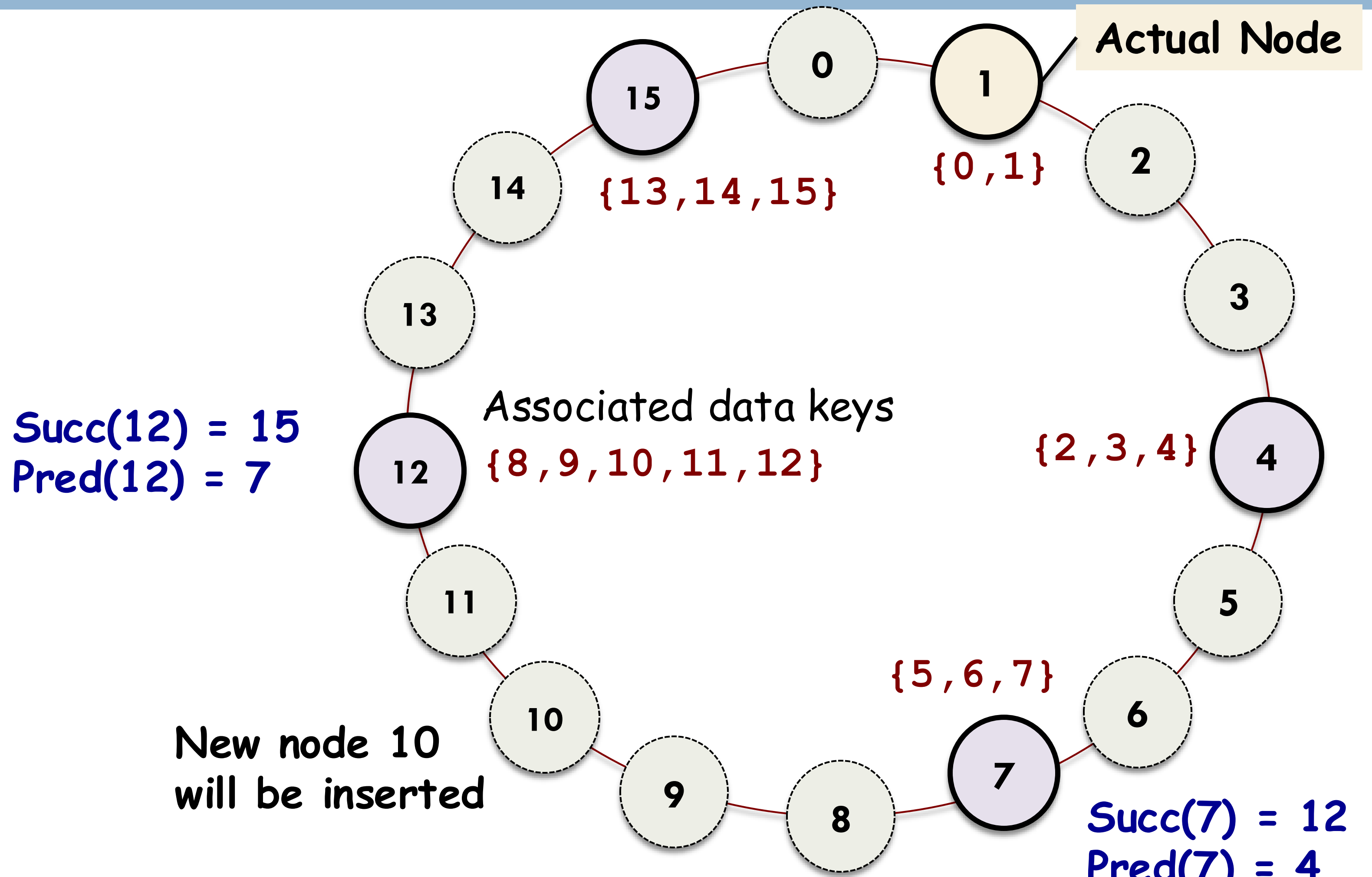
Chord lookup example for k=54



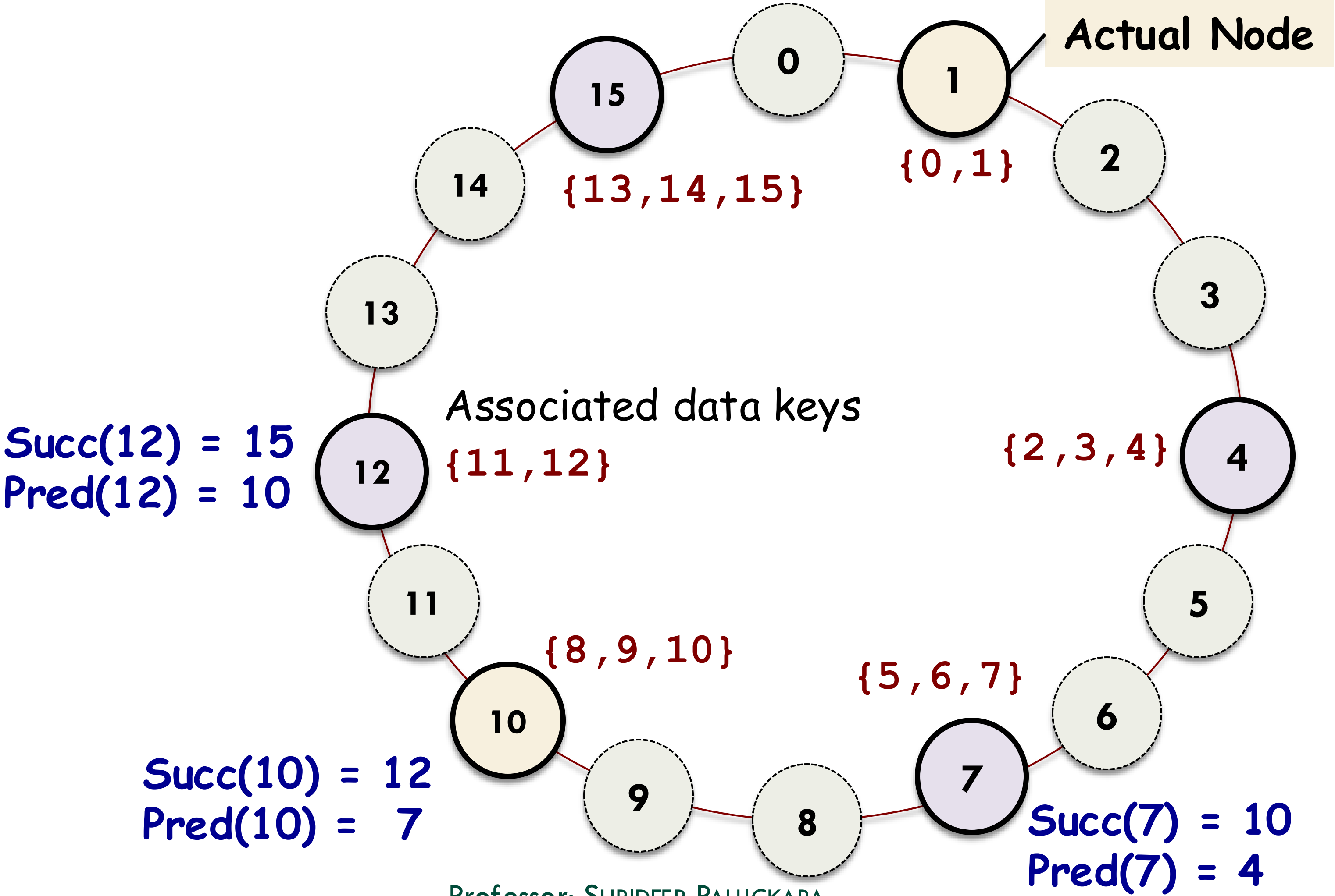
When a node wants to join

- Generate a random id
  - Probability of collisions is low
- **`lookup(id)`**
  - Will return successor(id)
- Contact *successor(id)* and its predecessor
  - Insert self in the ring
  - **Transfer** data items
    - All keys must be fetched from the new node's successor

An example of inserting a new node



An example of inserting a new node



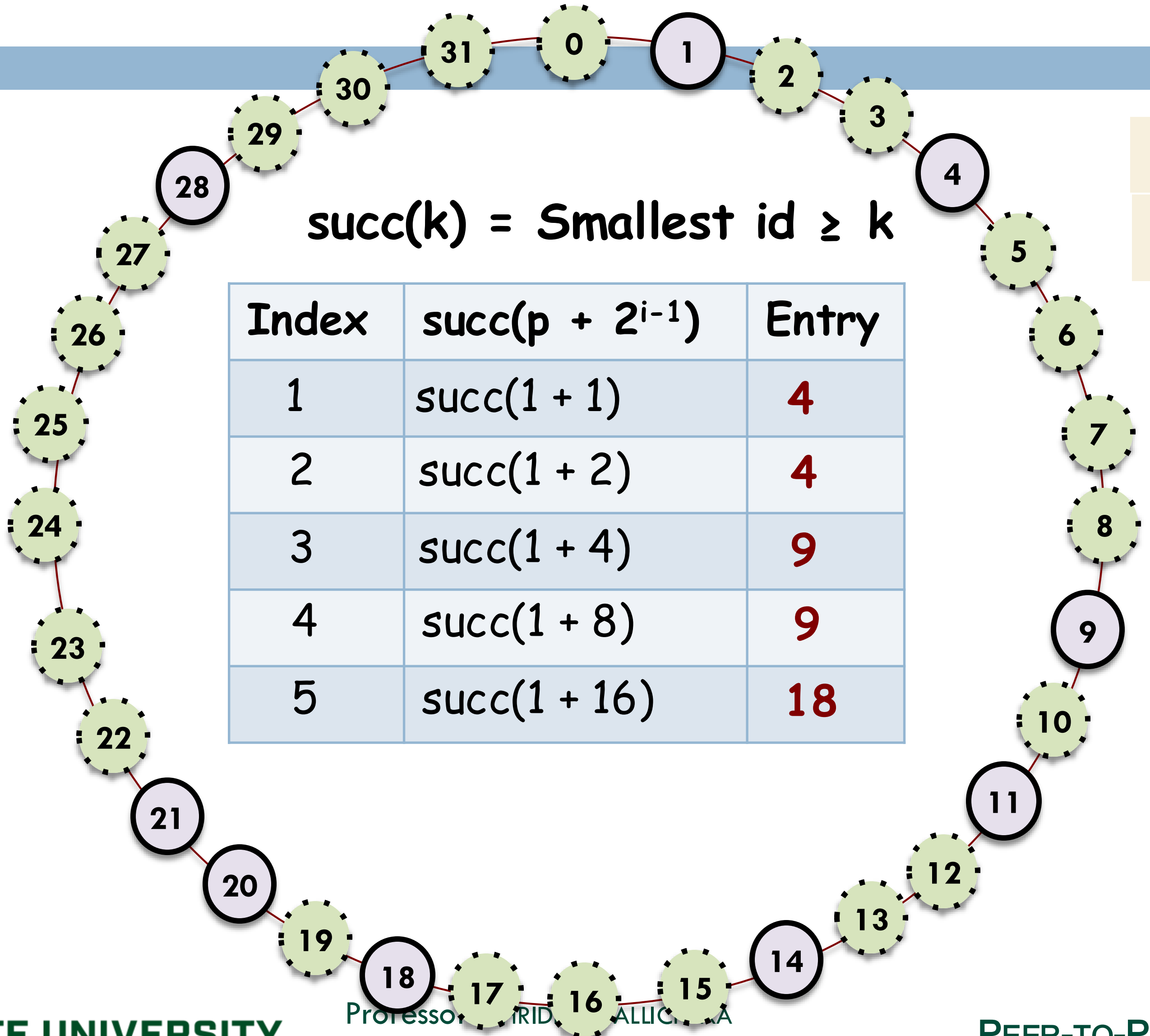
Finger Table in Chord

- Chord uses an m-bit identifier space
  - 2^m possible peers

- Each node, p, in Chord maintains a Finger Table with m-entries
  - $\mathbf{FT_p[i] = succ(p + 2^{i-1})}$

Note: This is when you count your indices from 1.
When you code, and we are counting from 0 this would be
$\mathbf{FT_p[i] = succ(p + 2^{i})}$

Constructing the Finger Table: Node 1



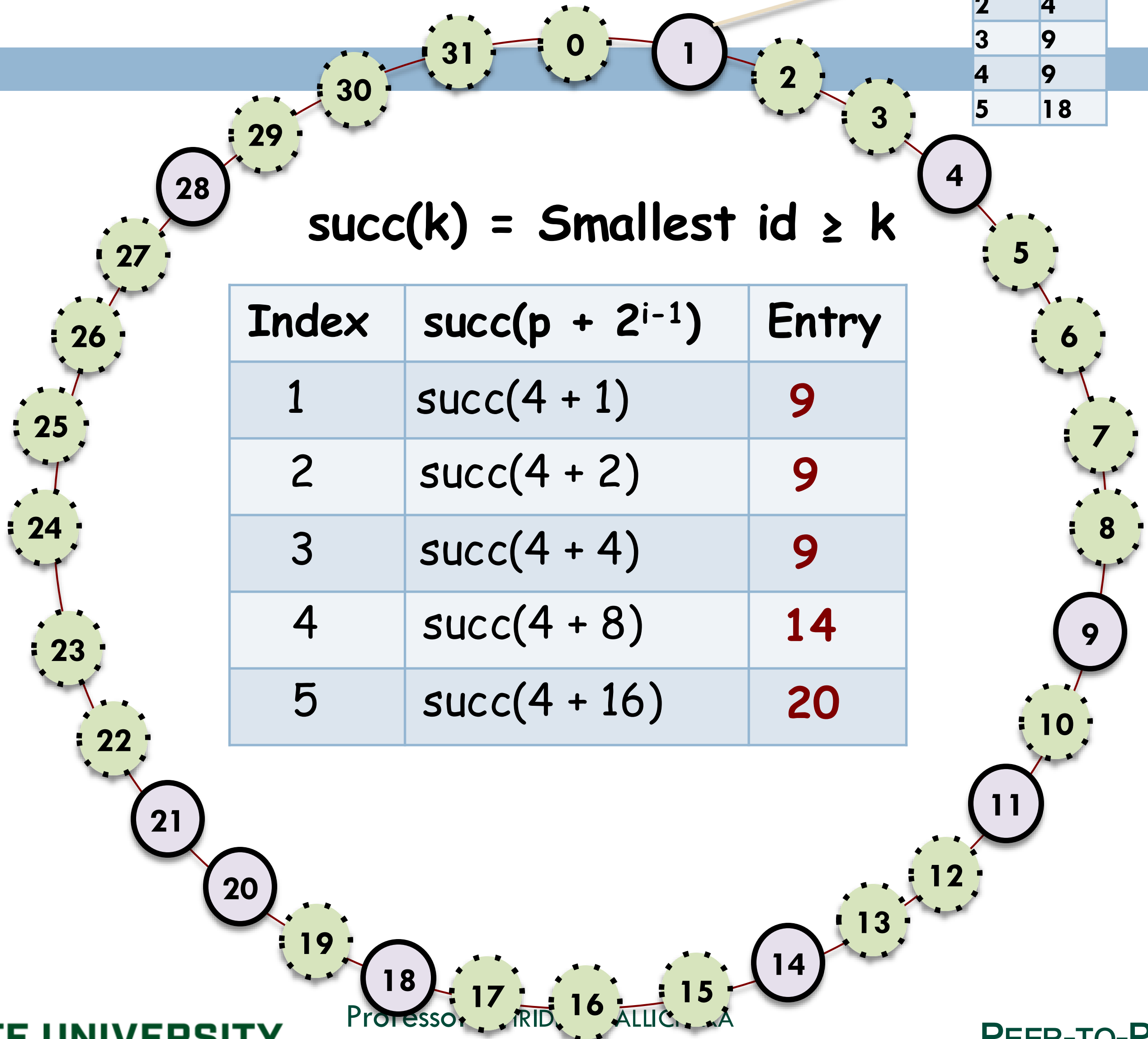
succ(k) = Smallest id ≥ k

Index	succ(p + 2^{i-1})	Entry
1	succ(1 + 1)	4
2	succ(1 + 2)	4
3	succ(1 + 4)	9
4	succ(1 + 8)	9
5	succ(1 + 16)	18

We are looking at a 5-bit ID space.

IDs go from 0 through (2^5 – 1)

Constructing the Finger Table: Node 4

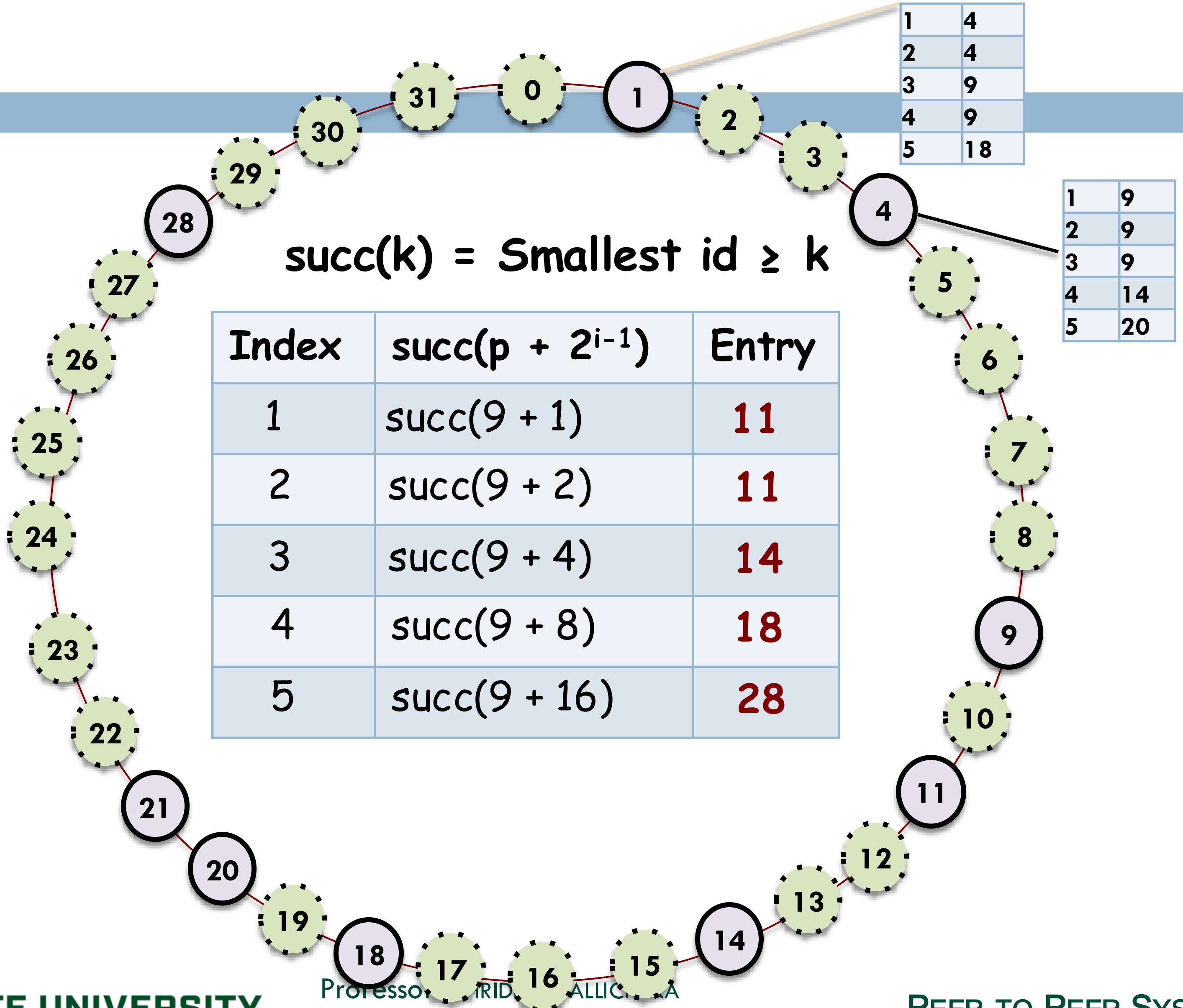


1	4
2	4
3	9
4	9
5	18

succ(k) = Smallest id ≥ k

Index	succ(p + 2^{i-1})	Entry
1	succ(4 + 1)	9
2	succ(4 + 2)	9
3	succ(4 + 4)	9
4	succ(4 + 8)	14
5	succ(4 + 16)	20

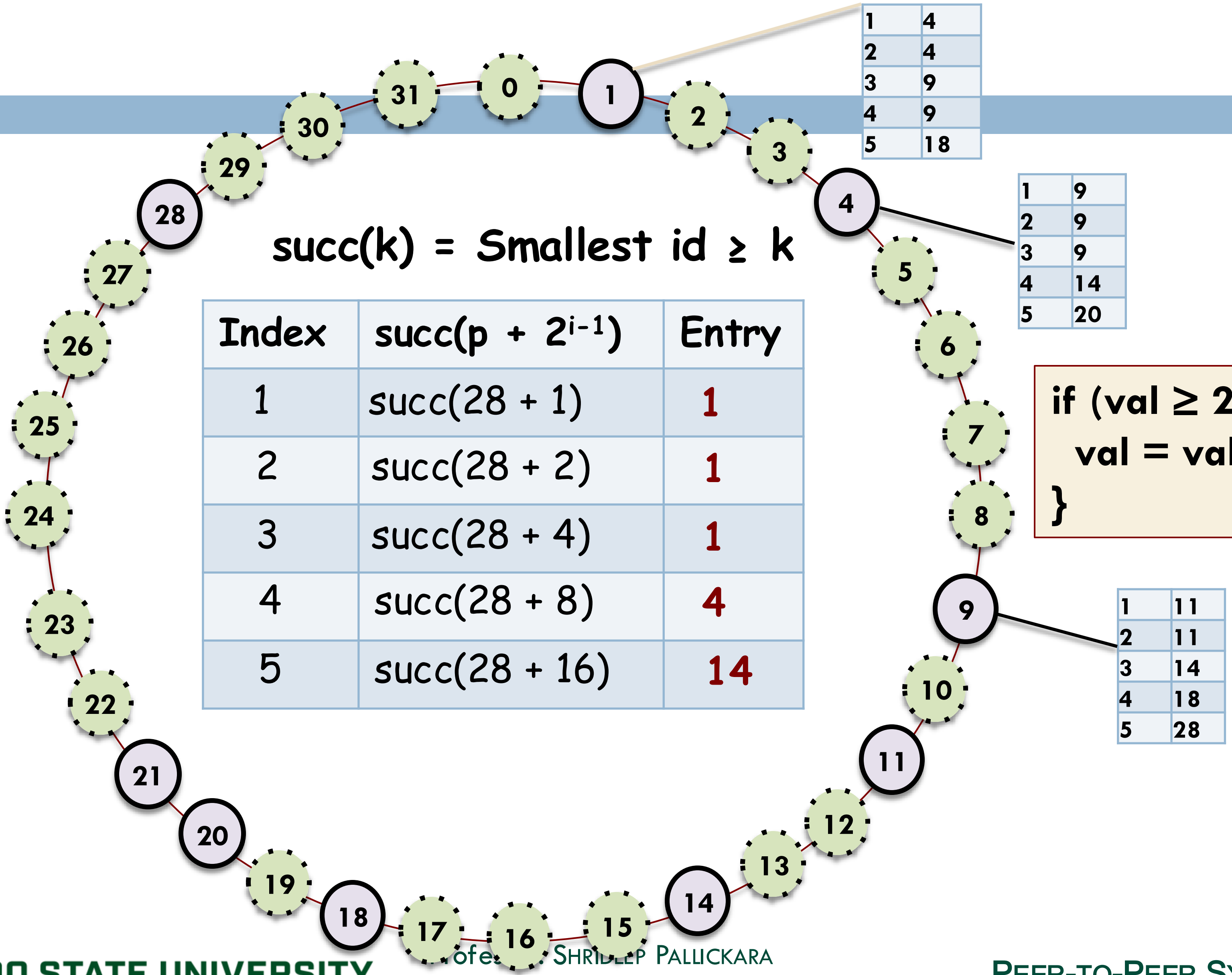
Constructing the Finger Table: Node 9



succ(k) = Smallest id ≥ k

Index	succ(p + 2^{i-1})	Entry
1	succ(9 + 1)	11
2	succ(9 + 2)	11
3	succ(9 + 4)	14
4	succ(9 + 8)	18
5	succ(9 + 16)	28

Constructing the Finger Table: Node 28



succ(k) = Smallest id ≥ k

Index	succ(p + 2^{i-1})	Entry
1	succ(28 + 1)	1
2	succ(28 + 2)	1
3	succ(28 + 4)	1
4	succ(28 + 8)	4
5	succ(28 + 16)	14

```
if (val ≥ 2^m) {
  val = val (mod 2^m)
}
```

Using the finger table to route queries: Make sure you don't overshoot

- To lookup a key k, node p will forward query to node q with index j in p's FT where:

Node with
greatest ID less than or equal to k

$$q = FT_p[j] \leq k < FT_p[j+1]$$

OR

$$q = FT_p[1] \text{ when } p < k < FT_p[1]$$

First entry ONLY if its ID is greater than k

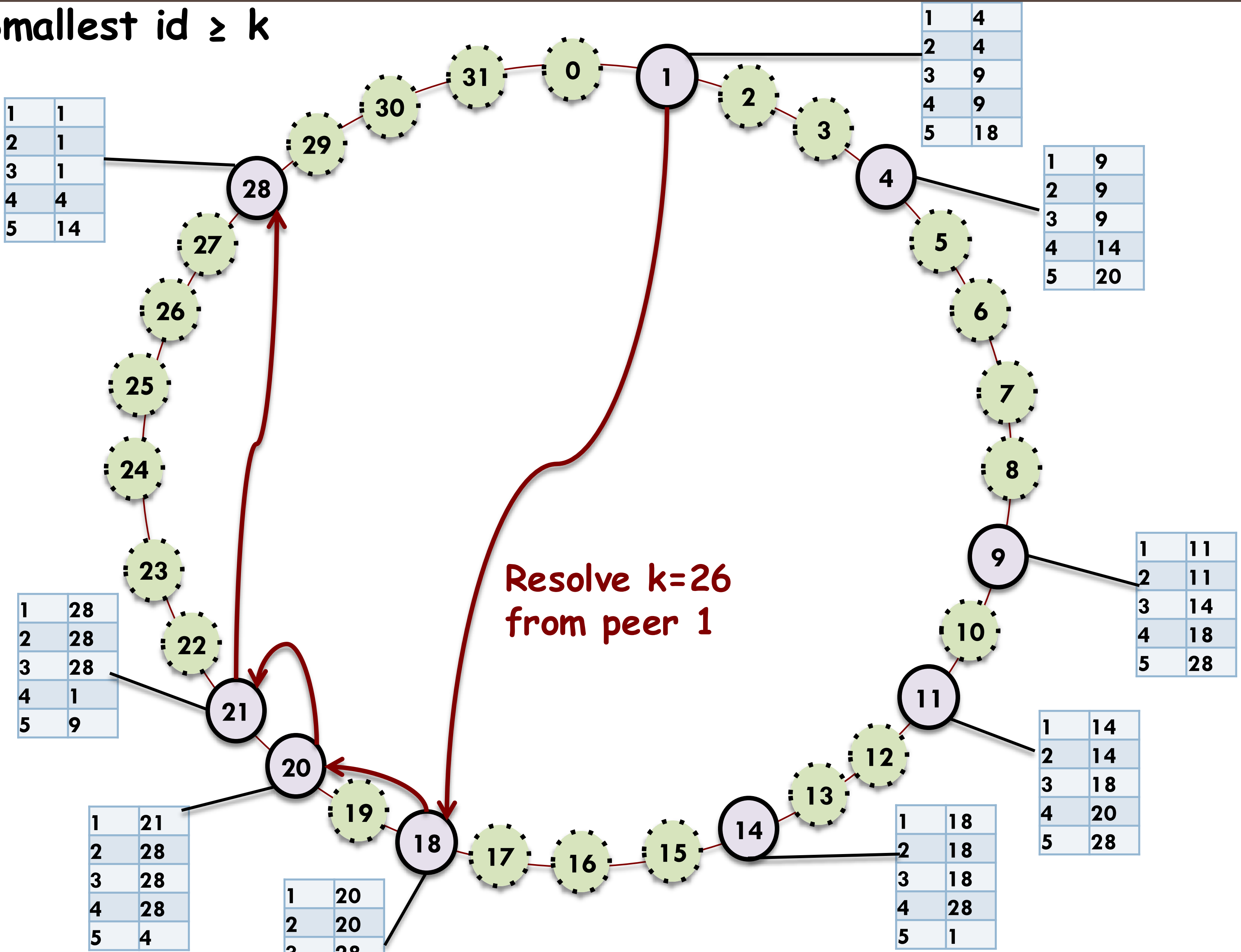
Stop forwarding the query when you are the target node

- A node is ***responsible*** for keys that fall in the range

  key > predecessor

  key <= self

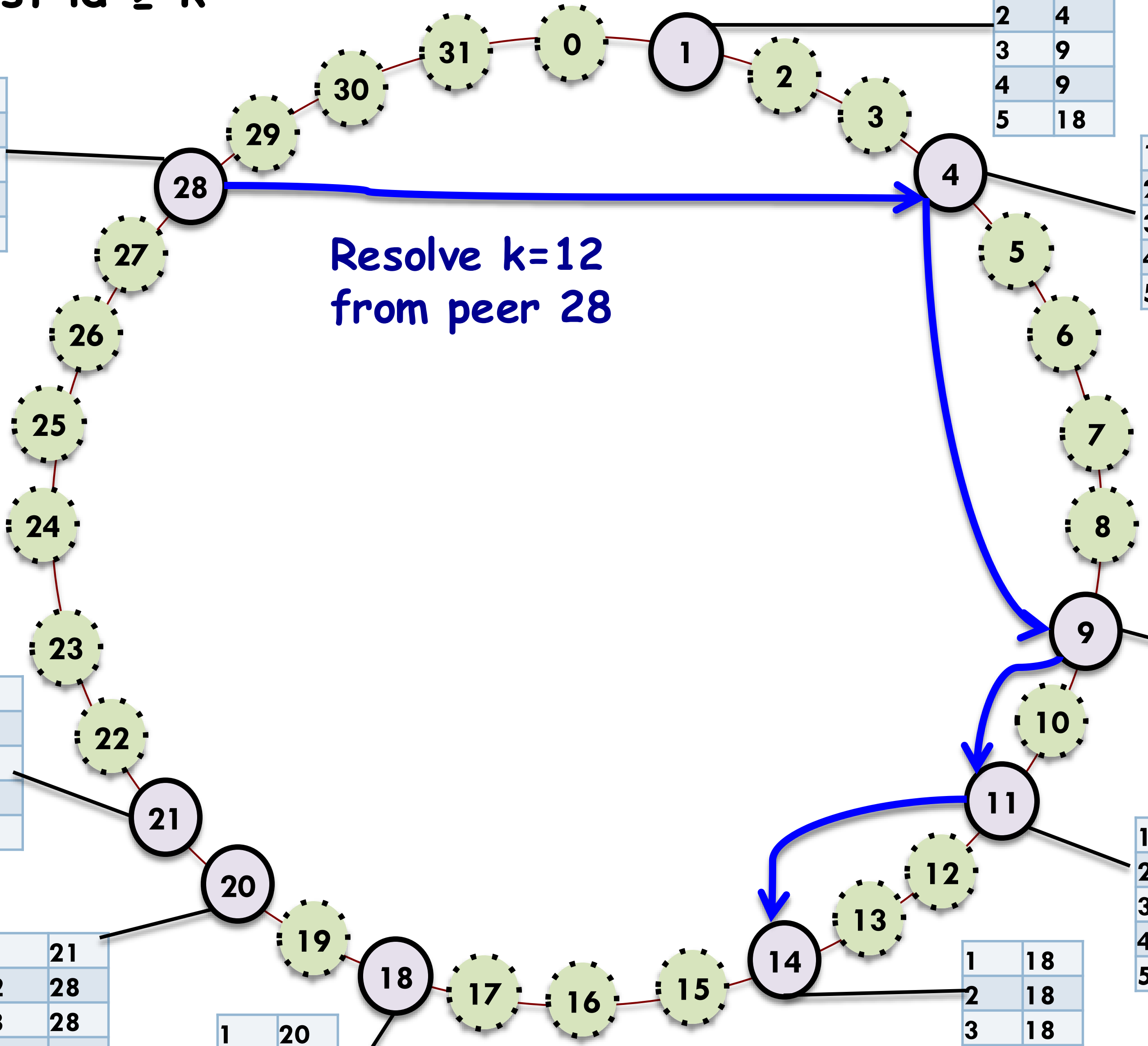
Smallest id ≥ k



$$q = FT_p[j] \leq k < FT_p[j+1]$$

$$q = FT_p[1] \text{ when } p < k < FT_p[1]$$

Smallest id ≥ k



1	4
2	4
3	9
4	9
5	18

1	9
2	9
3	9
4	14
5	20

1	11
2	11
3	14
4	18
5	28

1	14
2	14
3	18
4	20
5	28

1	18
2	18
3	18
4	28
5	1

1	20
2	20
3	28
4	28
5	4

1	21
2	28
3	28
4	28
5	4

1	28
2	28
3	28
4	1
5	9

1	1
2	1
3	1
4	4
5	14

$q = FT_p[j] \le k < FT_p[j+1]$

$q = FT_p[1]$ when $p < k < FT_p[1]$

Keeping the finger table up-to-date: At node q, $FT_q[1]$ must be accurate

① Contact $succ(q+1)$ {This is $FT_q[1]$}

- Have it return its predecessor

② If $q = pred(\ succ(q+1)\ )$

- Everything is fine

③ Otherwise:

- There is a new node p such that $q < p \leq succ(q+1)$
- $FT_q[1] = p$
- Check if p has recorded q as its predecessor
  No? Go to step (1)

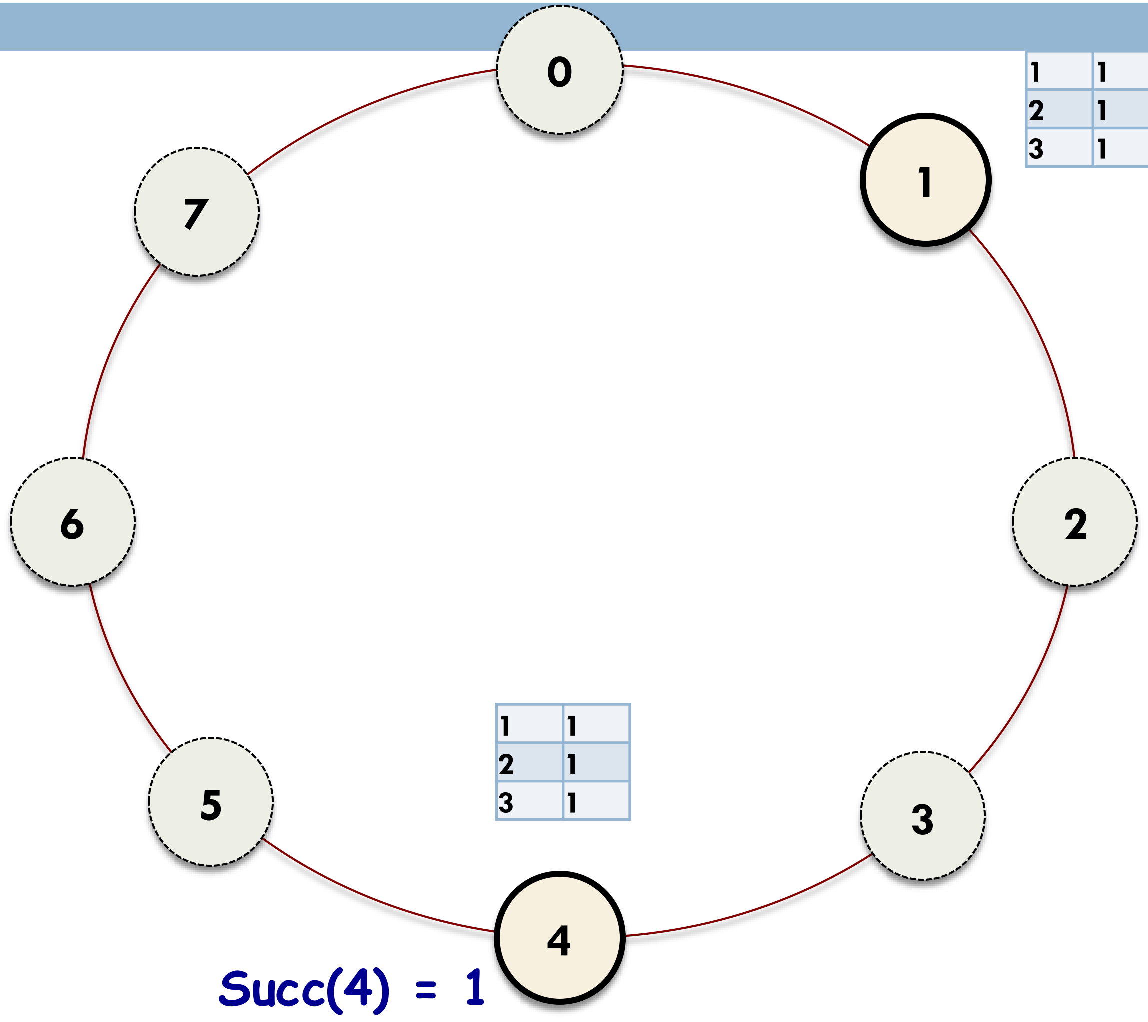
Before you can ever reach your destination, you must travel halfway there, always leaving another half.

Zeno's Paradox.

N.B: Also referred to as the *Dichotomy paradox* in a recounting of Zeno's Paradox by Aristotle.

An example of nodes joining in Chord

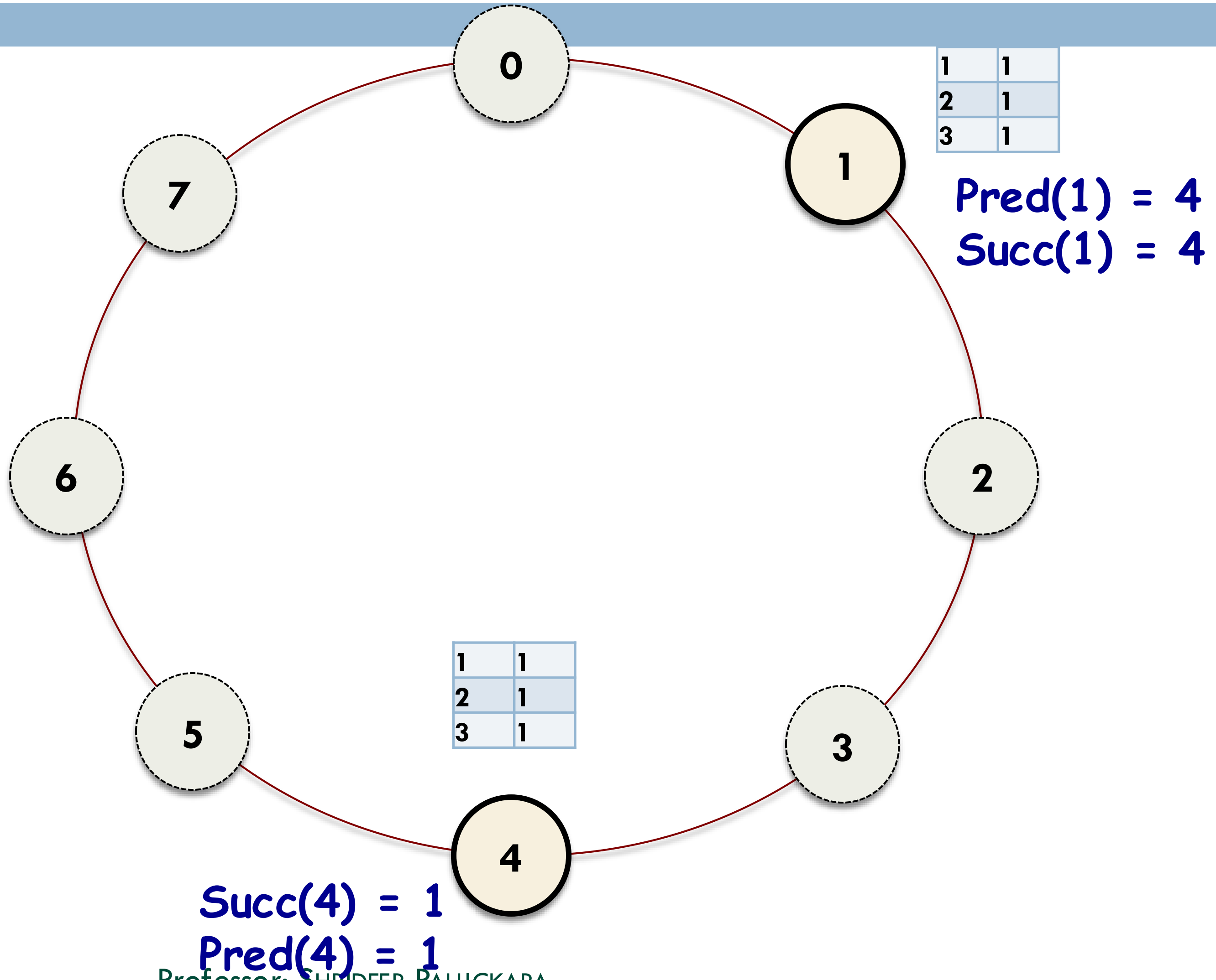
An example of inserting a new node N-4: Node-4 comes in and contacts Node-1



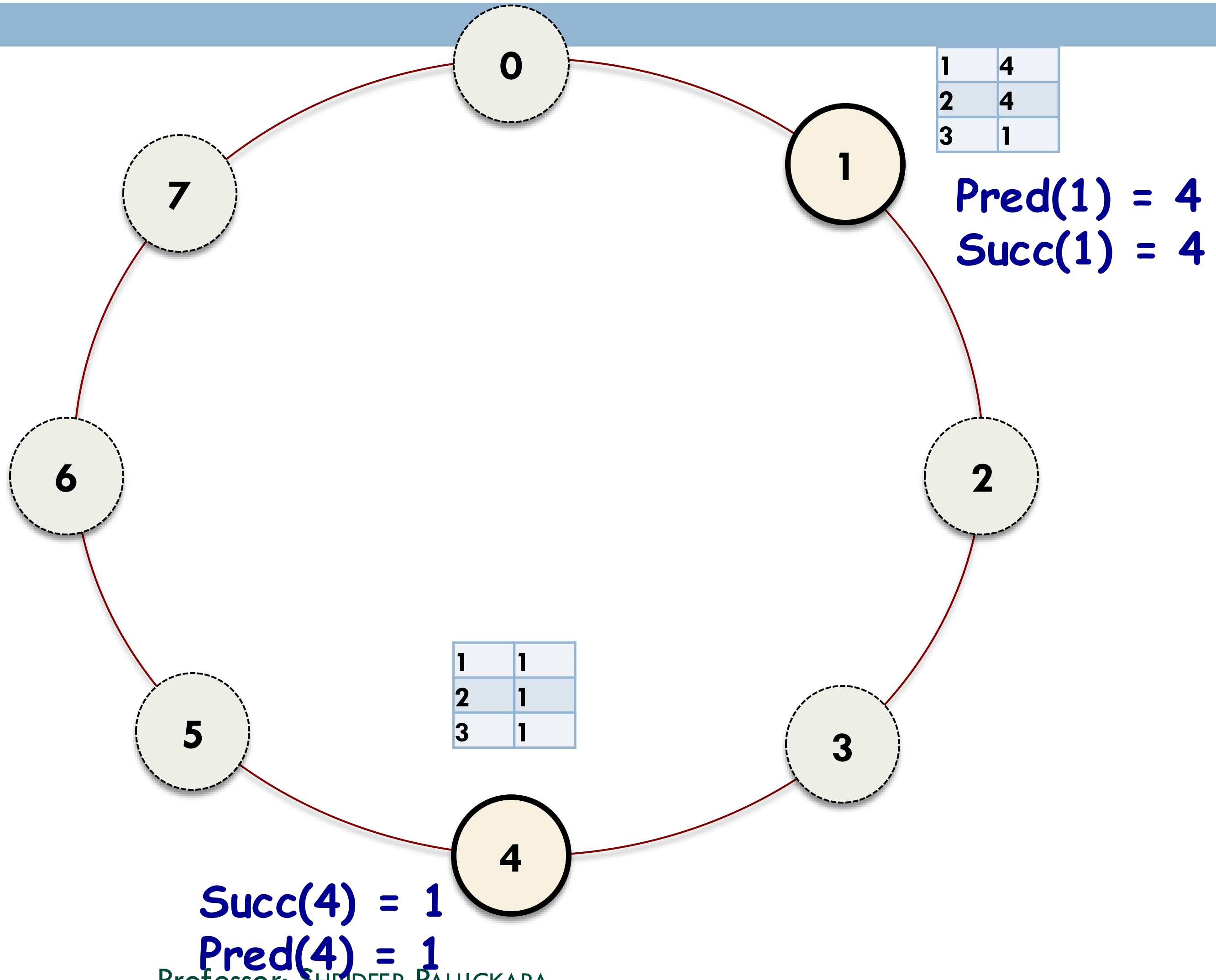
1	1
2	1
3	1

1	1
2	1
3	1

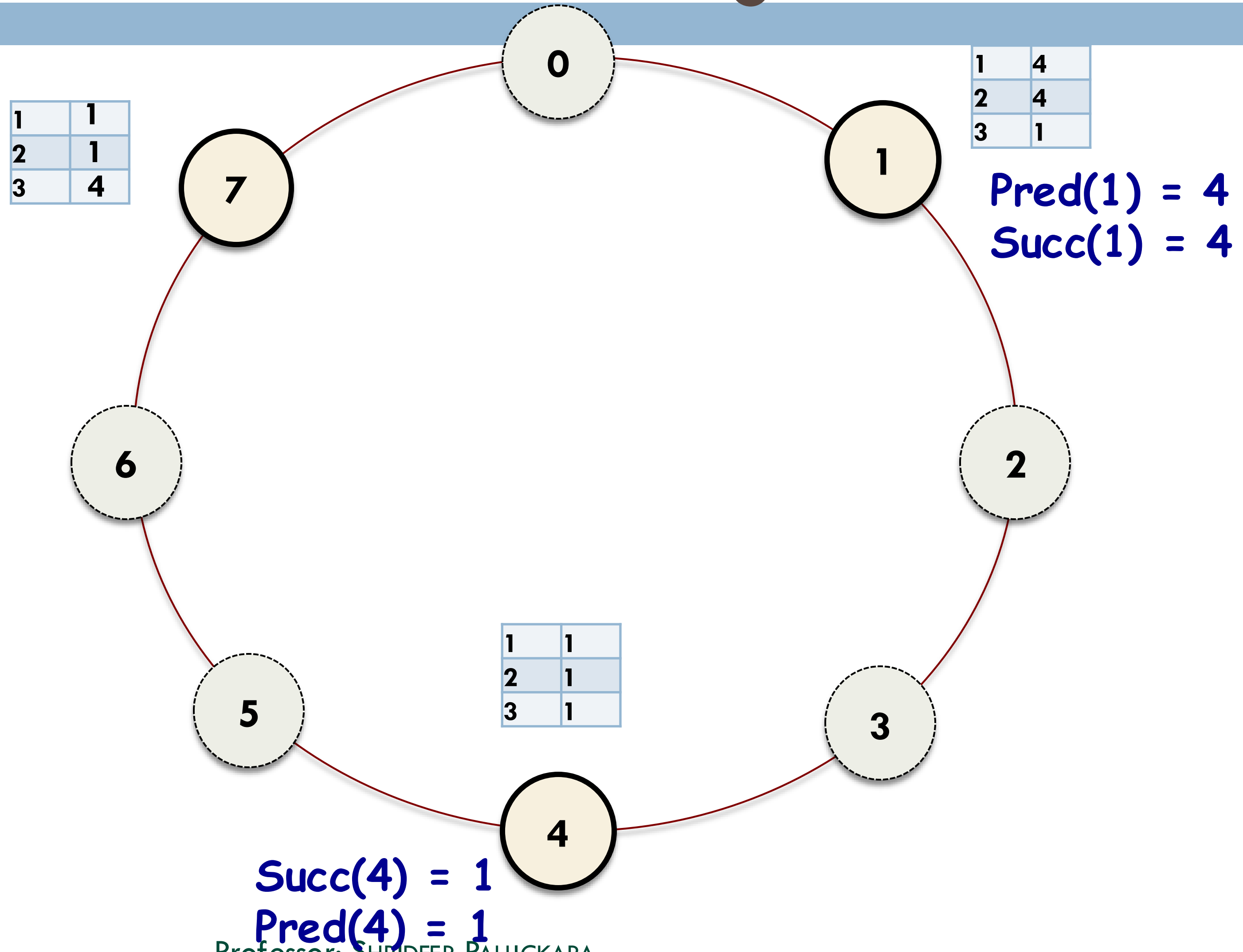
Installing successor at Node-1



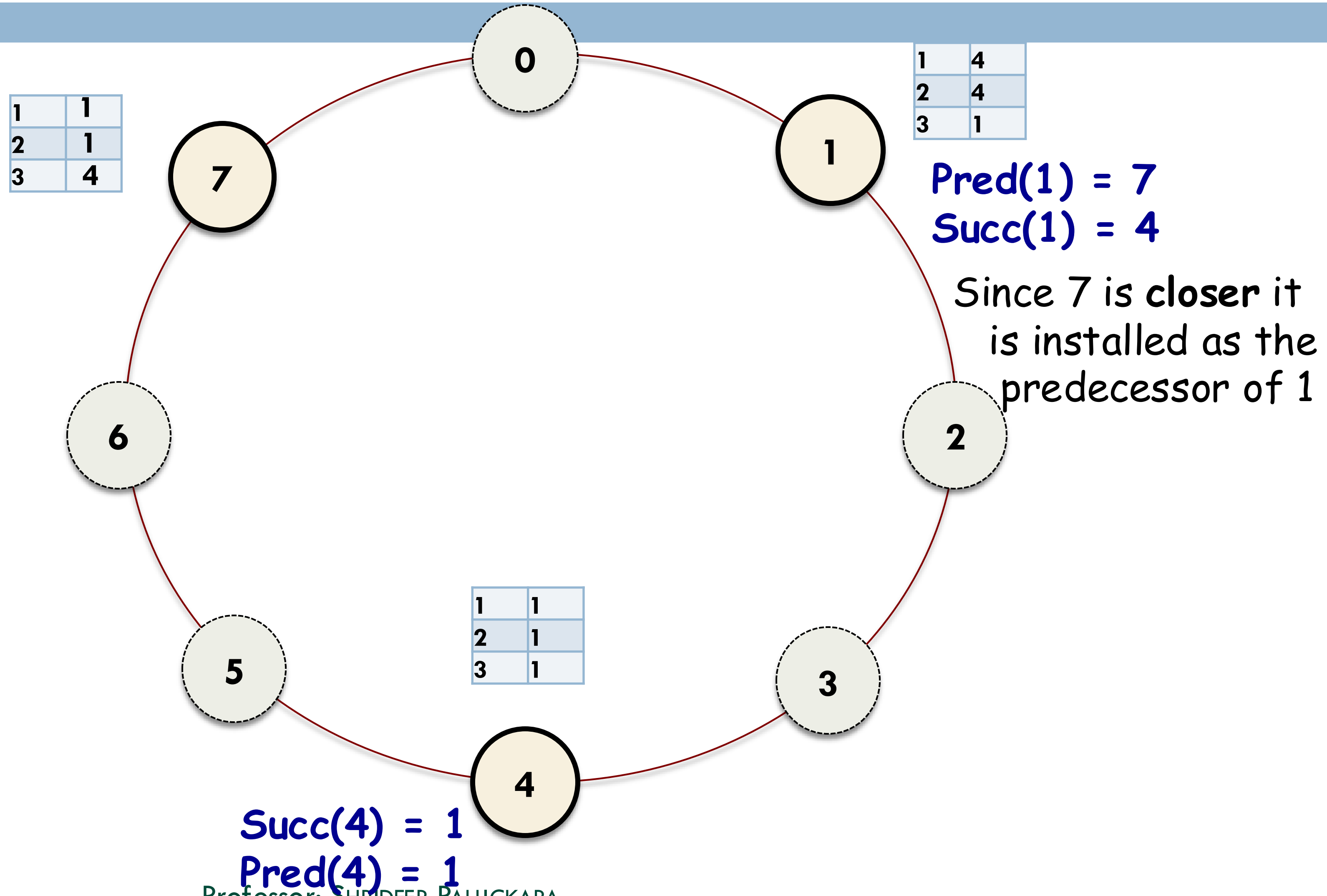
Updating the FT at N-1



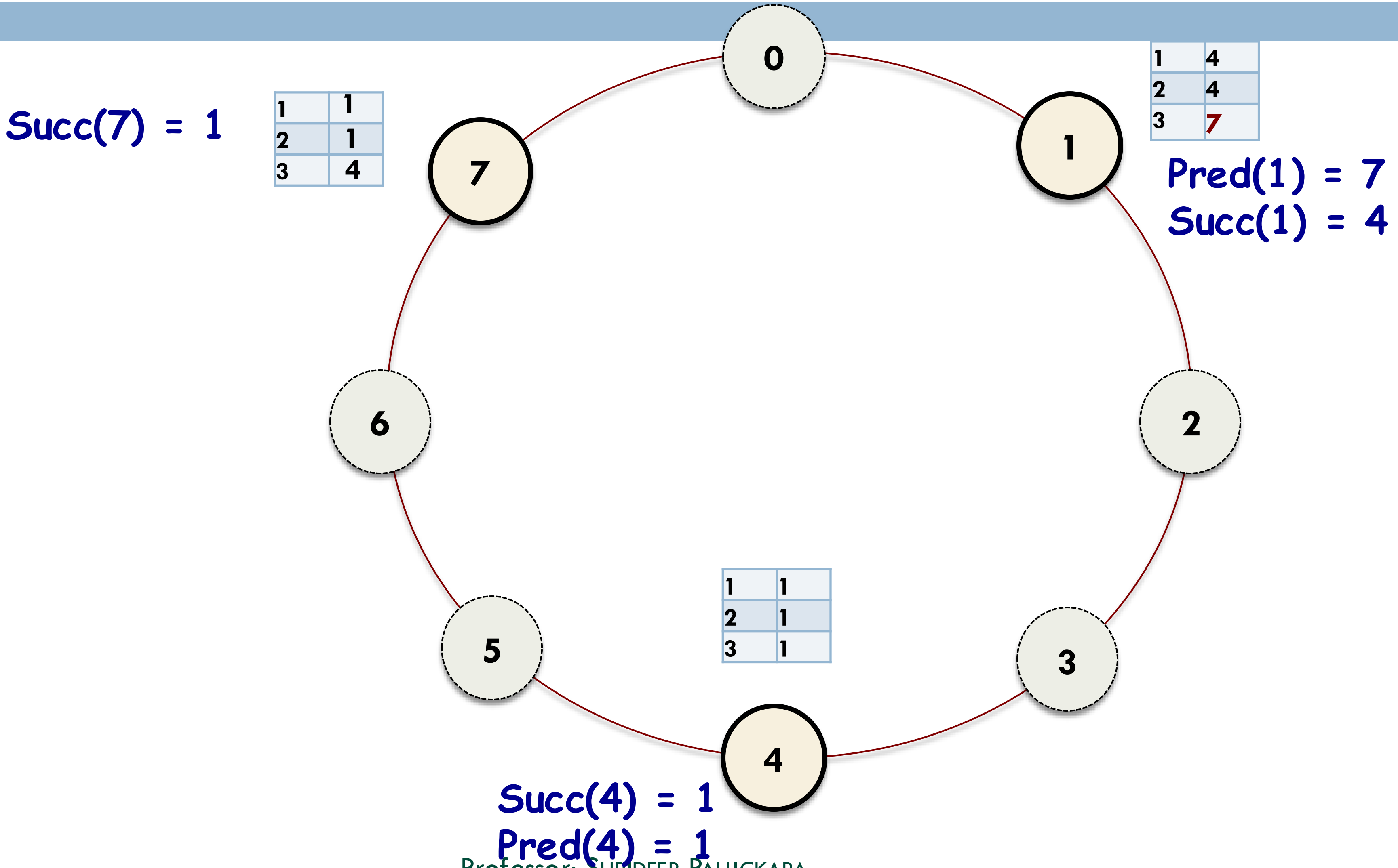
An example of inserting a new node N-7: N-7 contacts N-1 for filling its FT



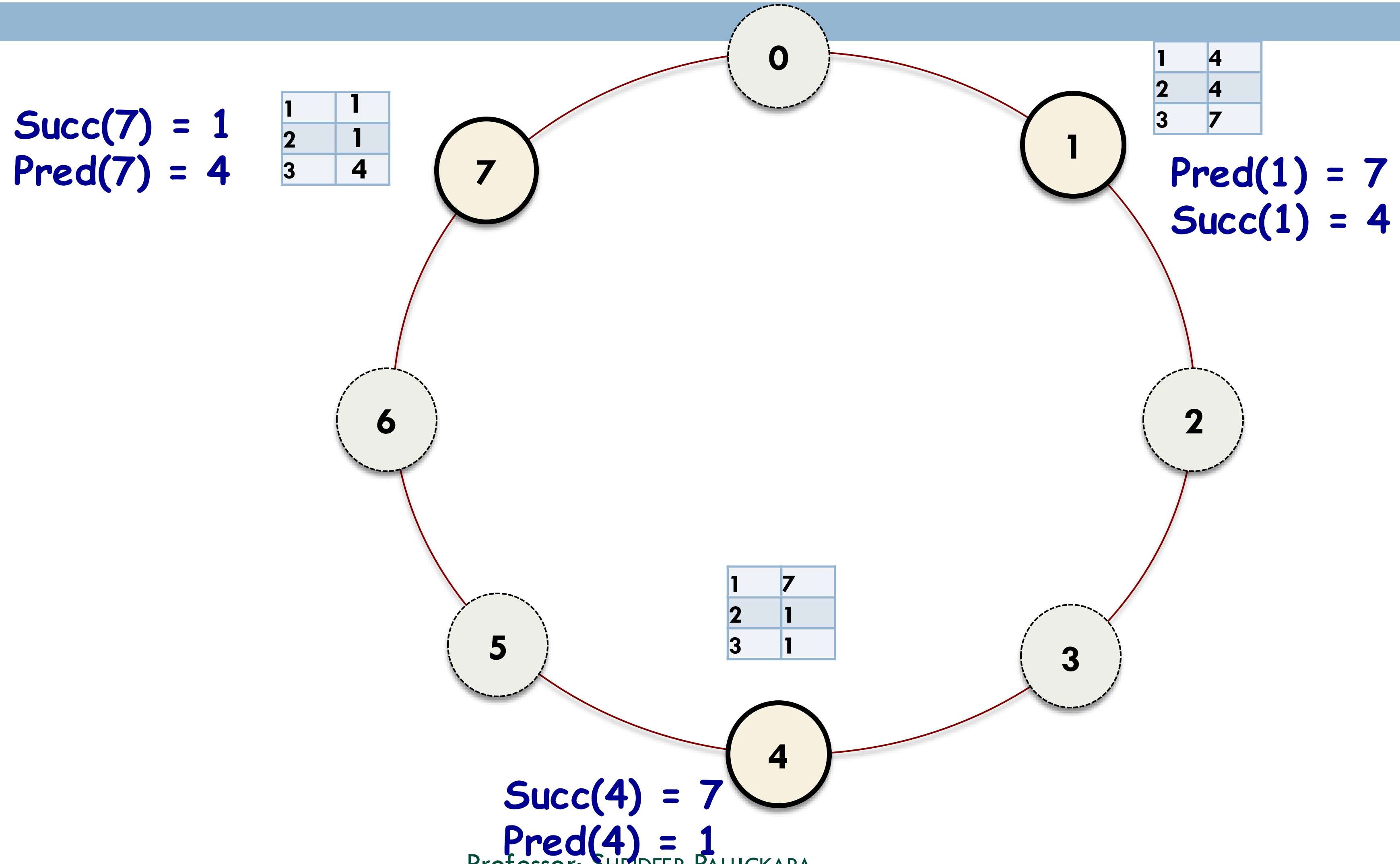
N-7 informs N-1 that it (N-7) is now N-1's predecessor



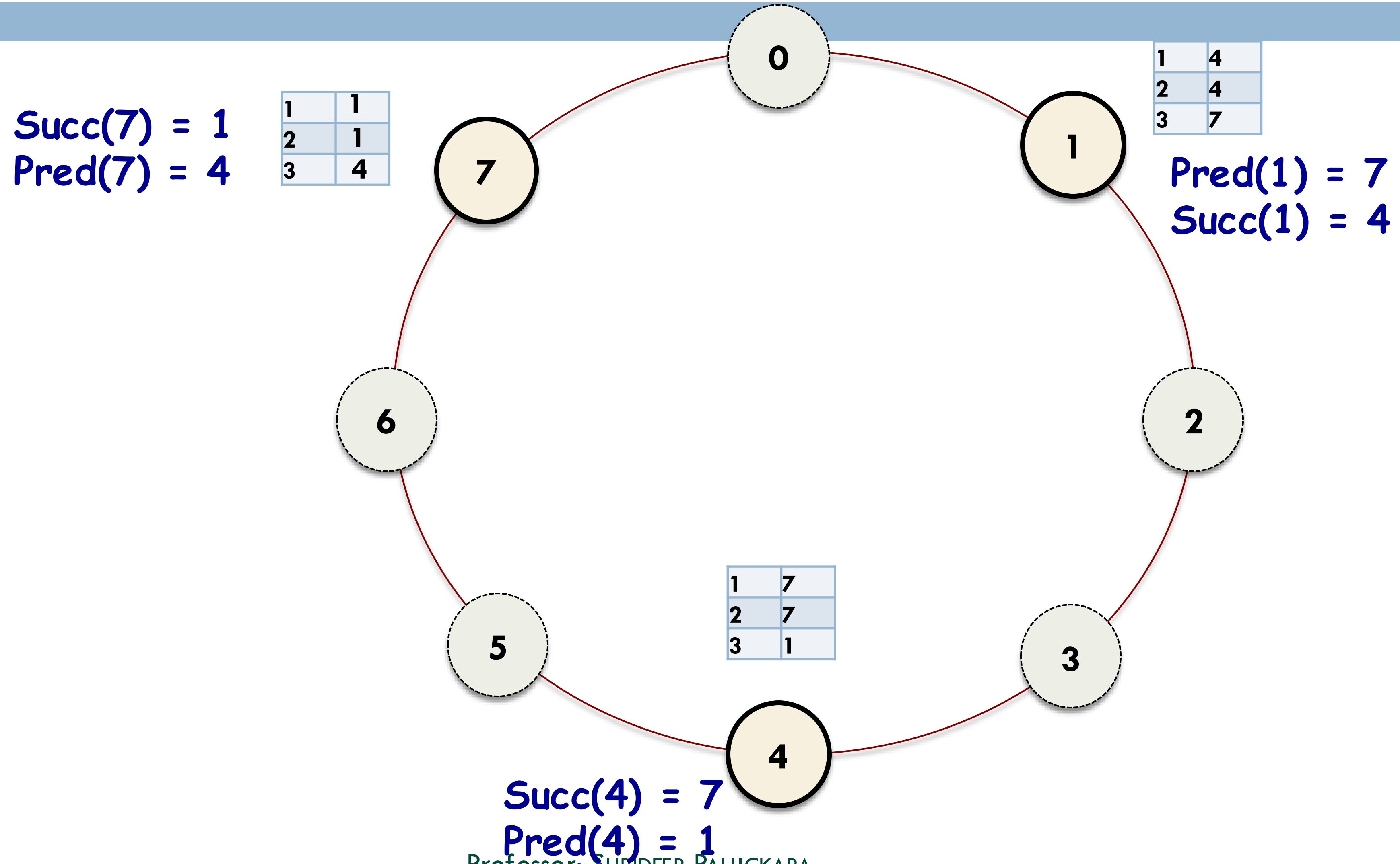
When N-1 updates its FT later on …



N-4 contacts N-1 to see if it is still its predecessor … and installs N-7 as its successor



When the FT at N-4 is updated …



The contents of this slide-set are based on the following references

- *Distributed Systems: Principles and Paradigms. Andrew S. Tanenbaum and Maarten Van der Steen. 2nd Edition. Prentice Hall. ISBN: 0132392275/978-0132392273.* [Chapter 5]
- *Distributed Systems: Concepts and Design. George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair. 5th Edition. Addison Wesley. ISBN: 978-0132143011.* [Chapter 10]