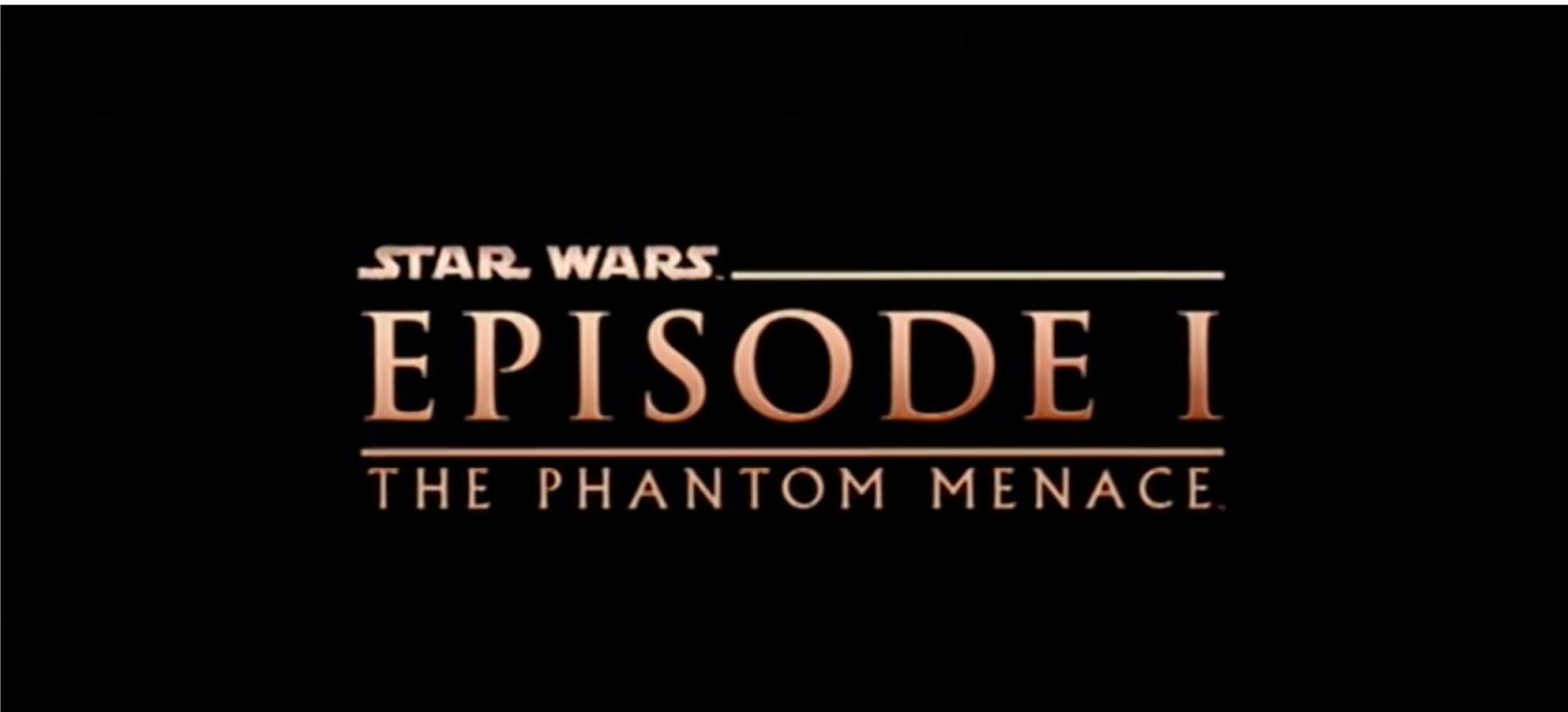


Lecture 9

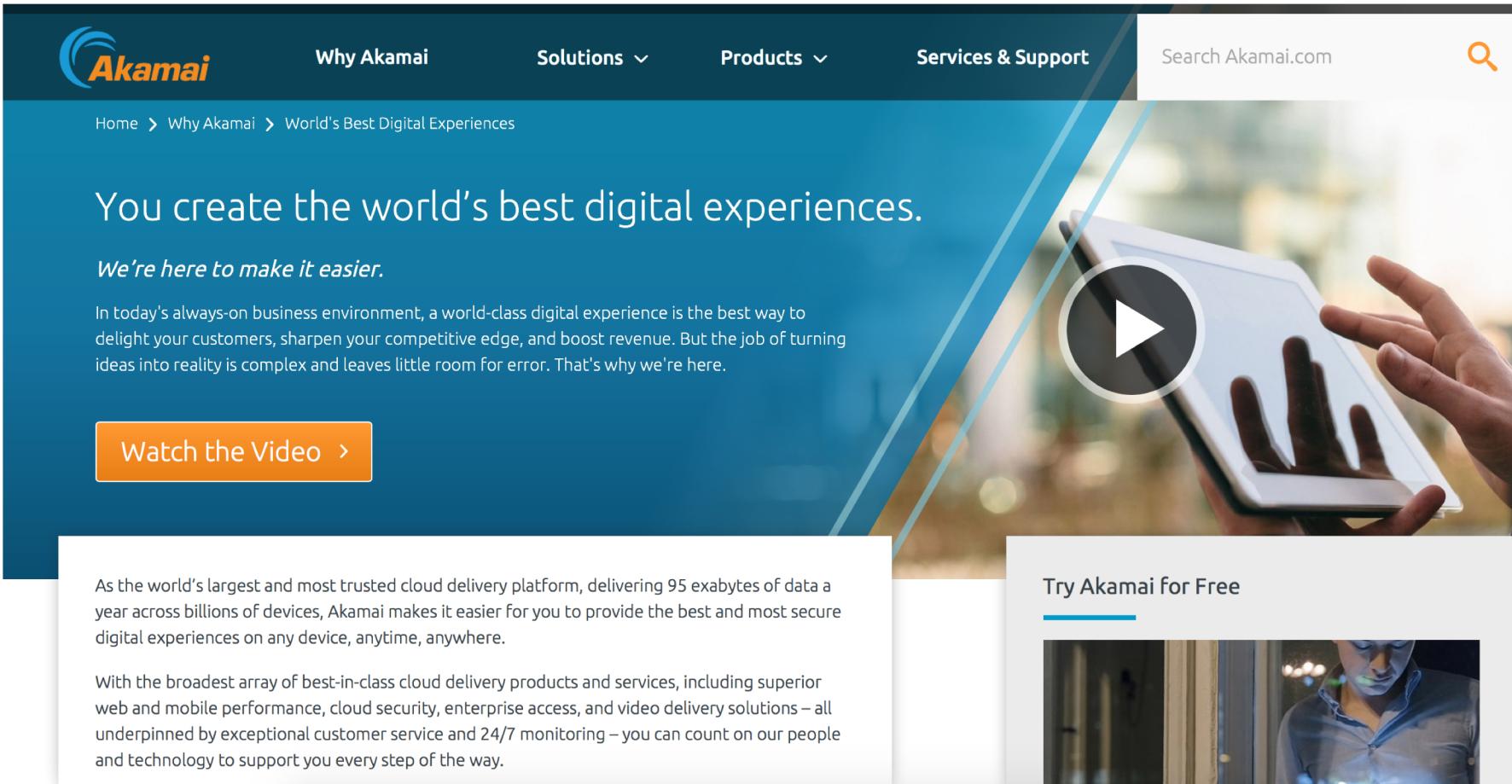
Consistent Hashing & Chord Algorithm

Wilson Rivera

Consistent Hashing!!



Consistent Caching: Akamai



The image shows a screenshot of the Akamai website's homepage. At the top, there is a dark navigation bar with the Akamai logo, "Why Akamai", "Solutions", "Products", "Services & Support", a search bar, and a magnifying glass icon. Below the navigation bar, the URL "Search Akamai.com" is visible. The main content area has a blue background with white text. It features the headline "You create the world's best digital experiences." followed by the subtext "We're here to make it easier." and a paragraph about the company's mission. A large orange button labeled "Watch the Video >" is prominently displayed. To the right of the text, there is a large image of a person's hands interacting with a tablet screen, which also displays a play button icon. A white callout box contains text about Akamai's role in providing secure digital experiences. Another callout box below it discusses the company's broad array of products and services. At the bottom right, there is a section titled "Try Akamai for Free" with a small image of a person looking at a screen.

Akamai

Why Akamai

Solutions

Products

Services & Support

Search Akamai.com

Home > Why Akamai > World's Best Digital Experiences

You create the world's best digital experiences.

We're here to make it easier.

In today's always-on business environment, a world-class digital experience is the best way to delight your customers, sharpen your competitive edge, and boost revenue. But the job of turning ideas into reality is complex and leaves little room for error. That's why we're here.

Watch the Video >

As the world's largest and most trusted cloud delivery platform, delivering 95 exabytes of data a year across billions of devices, Akamai makes it easier for you to provide the best and most secure digital experiences on any device, anytime, anywhere.

With the broadest array of best-in-class cloud delivery products and services, including superior web and mobile performance, cloud security, enterprise access, and video delivery solutions – all underpinned by exceptional customer service and 24/7 monitoring – you can count on our people and technology to support you every step of the way.

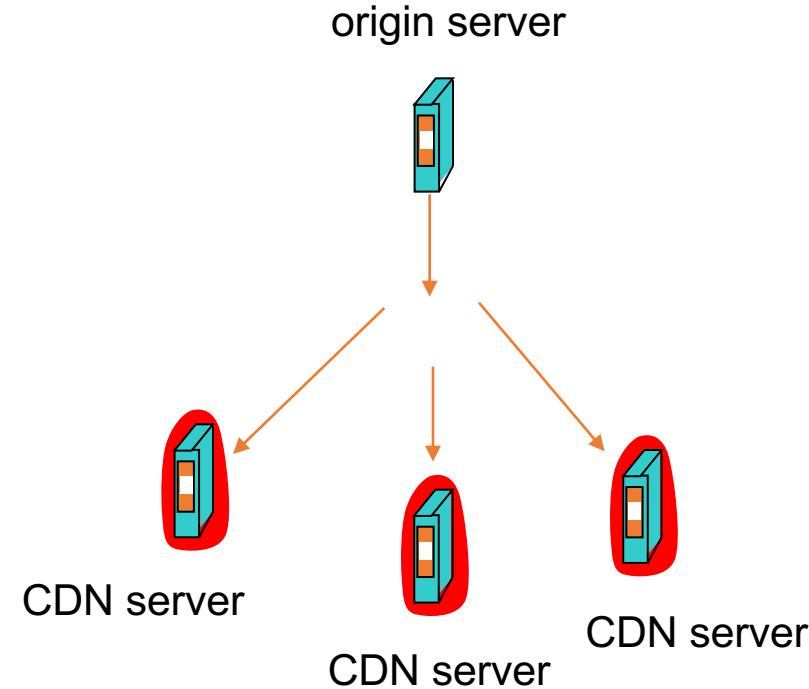
Try Akamai for Free

Consistent Caching

- Web caching
 - Faster response, less traffic
- Spread caching over multiple machines
- Consistent hashing
 - 1997: Research paper
 - 1998: Akamai is funded
 - 1999: start wars trailer (apple)

Scalable algorithms for discovery

- If many nodes are available to cache, which one should file be assigned to?
- If content is cached in some node, how can we discover where it is located, avoiding centralized directory or all-to-all communication?



Partitioning Problem

- Consider problem of data partition:
 - Given data X, choose one of k servers to use
- Suppose we use modulo hashing
 - Number servers 1..k
 - Place X on server $i = \text{hash}(X) \bmod k$
 - Problem? What happens if a server fails or joins ($k \rightarrow k \pm 1$)?
 - Answer: All entries get remapped to new nodes!

Chord Algorithm – Consistent Hashing

- Supports just one operation: given a key, it maps the key onto a node.
- Consistent Hashing
 - Assign each node and key an m-bit identifier using a base hash function such as SHA-1
 - Identifiers are ordered in an identifier circle modulo 2^m (Chord ring)
 - Key k is assigned to the first node whose identifier is equal to or follows k.

I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan.
Chord: A scalable Peer-To-Peer lookup service for internet
applications. In Proceedings of the 2001 ACM SIGCOMM
Conference, pages 149–160, 2001.

Chord Algorithm – Consistent Hashing

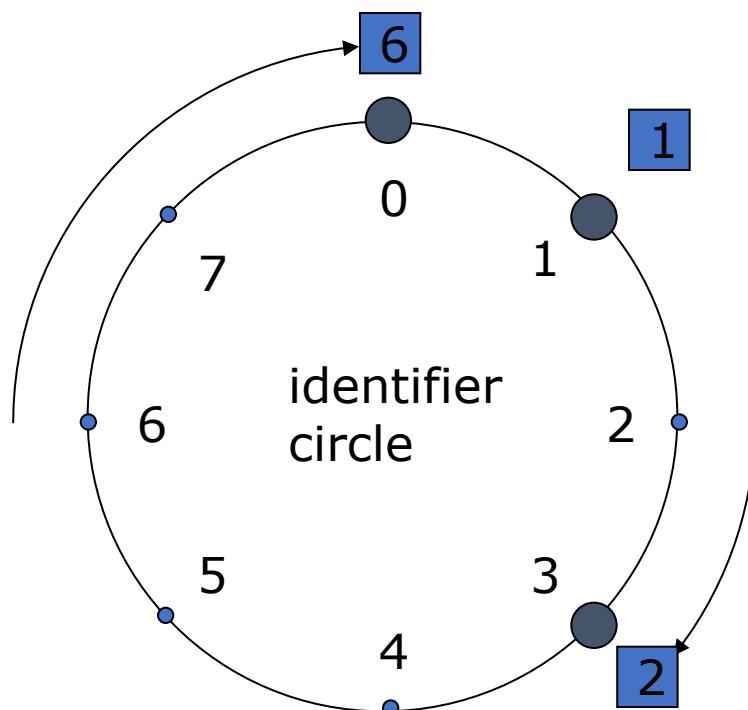
identifier space : $m=3$

node key

0 1

1 2

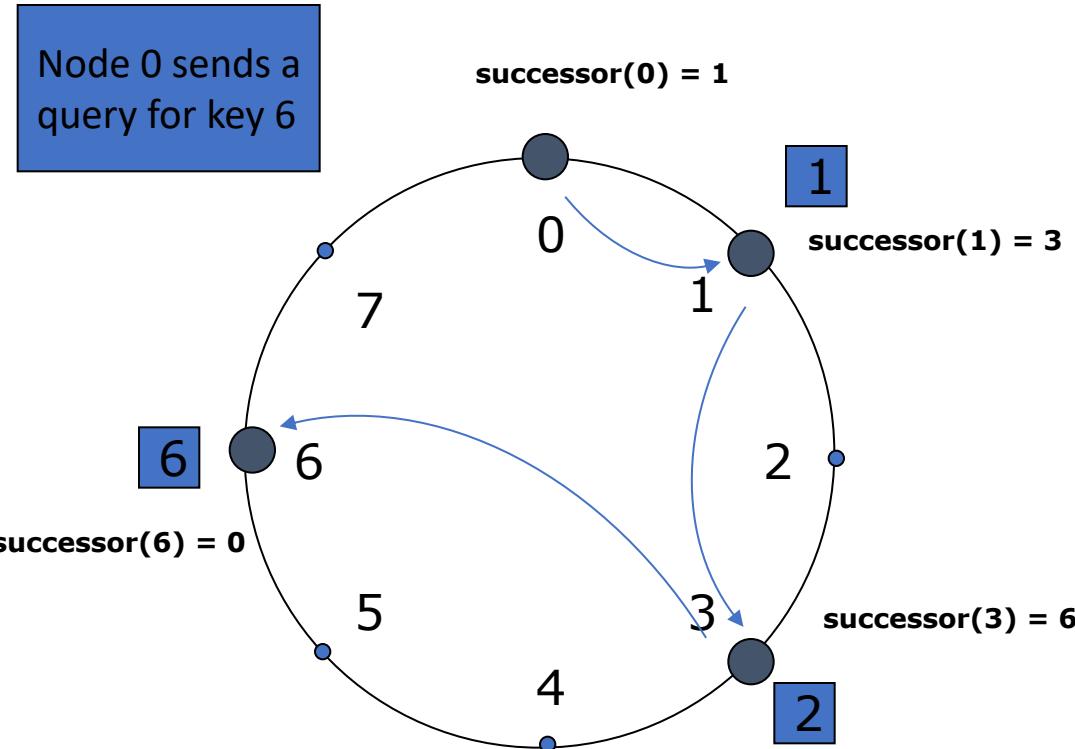
3 6



Chord Algorithm – Simple Key Lookup

- Simple Key Lookup

- Queries are passed around the circle via successor pointers
- Requires traversing all Nodes to find the appropriate mapping



Chord Algorithm – Scalable Key Location

- Finger Table

- Each node n maintains a routing table with up to m entries
- The i^{th} entry in the table at node n contains the identifier of the first node s that succeeds n by at least 2^{i-1} on the identifier circle.
 - ($s = \text{successor}(n+2^{i-1})$).
- s is called the i^{th} finger of node n .

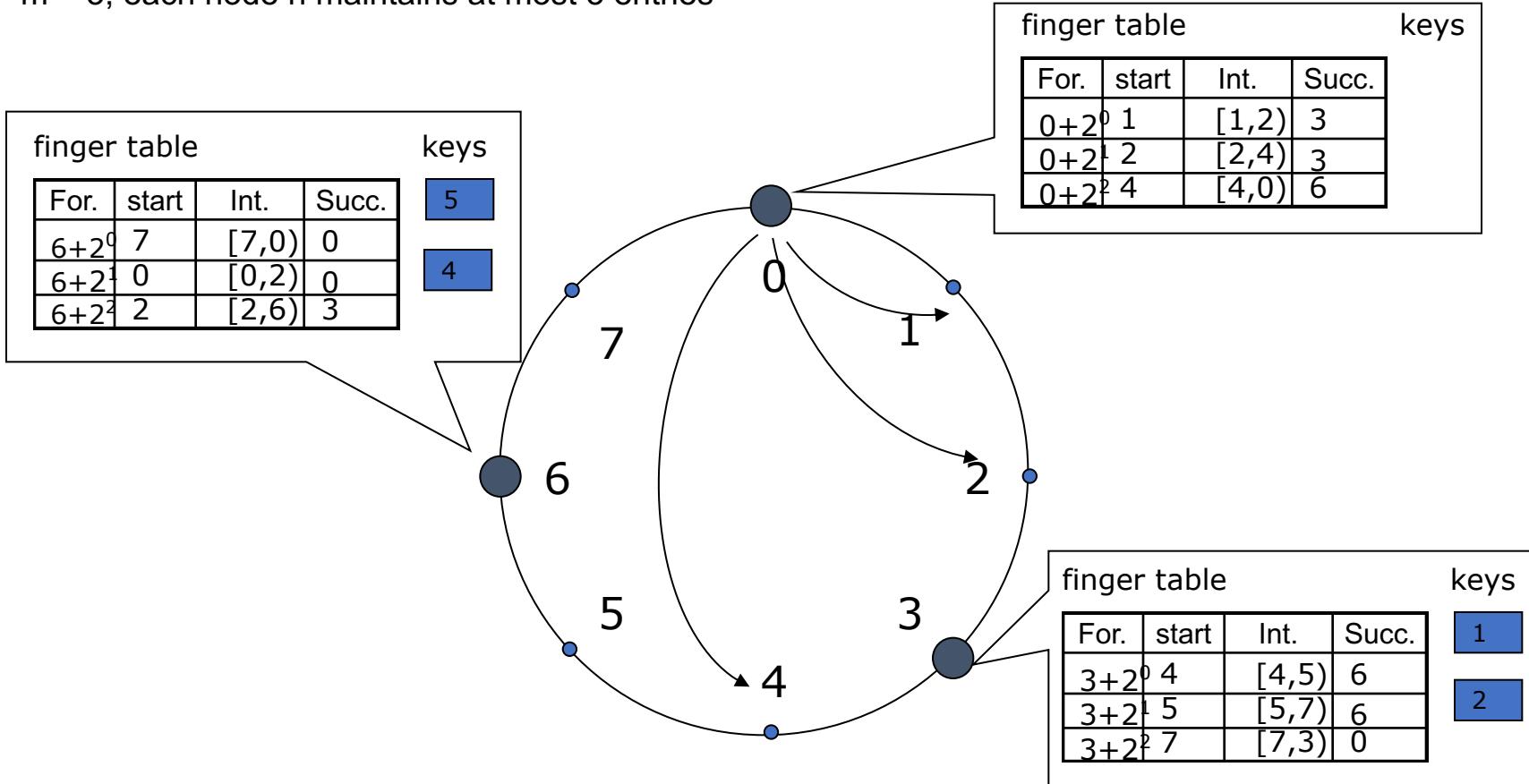
▷ Definition of variables for node n

Notation	Definition
$\text{finger}[k].start$	$(n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
$.interval$	$[\text{finger}[k].start, \text{finger}[k+1].start)$
$.node$	first node $\geq n$, $\text{finger}[k].start$
successor	the next node on the identifier circle; $\text{finger}[1].node$
predecessor	the previous node on the identifier circle

Chord Algorithm – Scalable Key Location

Finger table

$m = 3$, each node n maintains at most 3 entries



Chord Algorithm – Scalable Key Location

```
// ask node n to find id's successor
```

```
n.find_successor(id)  
n' = find_predecessor(id);  
return n'.successor;
```

Find id's successor by finding the immediate predecessor of the id

```
// ask node n to find id's predecessor
```

```
n.find_predecessor(id)  
n' = n;  
while (id  $\notin$  (n', n'.successor])  
    n' = n'.closest_preceding_finger(id);  
return n';
```

Walk clockwise to find the node which precedes id and whose successor succeeds id

```
// return closest finger preceding id
```

```
n.closest_preceding_finger(id)  
for i = m downto 1  
    if (finger[i].node  $\in$  (n, id))  
        return finger[i].node;  
return n;
```

Start with the m^{th} finger of node n
See if it comes between node n and the id, if not, check the $m-1^{\text{th}}$ finger until we find one which does.
This is the closest node preceding id among all the fingers of n

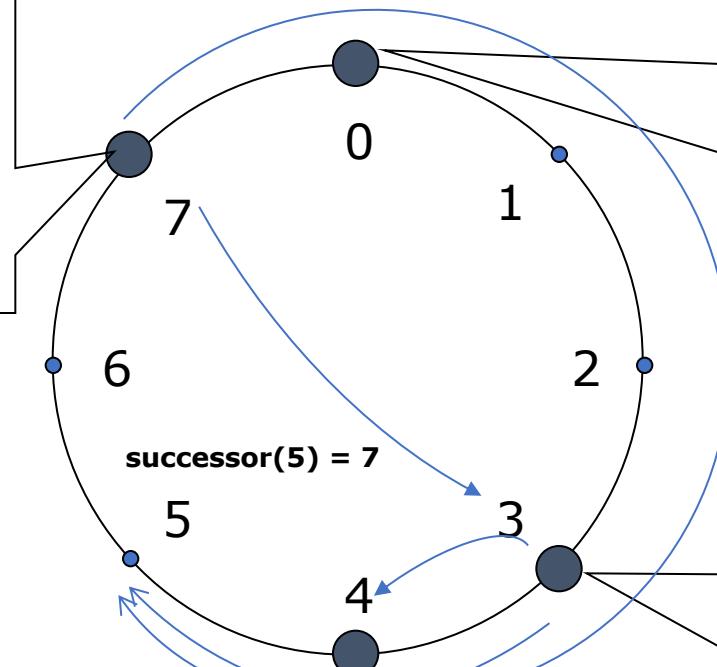
Chord Algorithm – Scalable Key Location

$\text{id}=5$

$n=7$

finger table			keys
start	Int.	Succ.	6
0	[0,1)	0	
1	[1,3)	0	
3	[3,7)	3	

Successor 0
Predecessor 4



finger table			keys
start	Int.	Succ.	3
1	[1,2)	3	
2	[2,4)	3	
4	[4,0)	4	

Successor 3
Predecessor 7

finger table			keys
start	Int.	Succ.	4
5	[5,6)	7	
6	[6,0)	7	
0	[0,4)	0	

Successor 7
Predecessor 3

finger table			keys
start	Int.	Succ.	1
4	[4,5)	4	
5	[5,7)	7	
7	[7,3)	7	

Successor 4
Predecessor 0

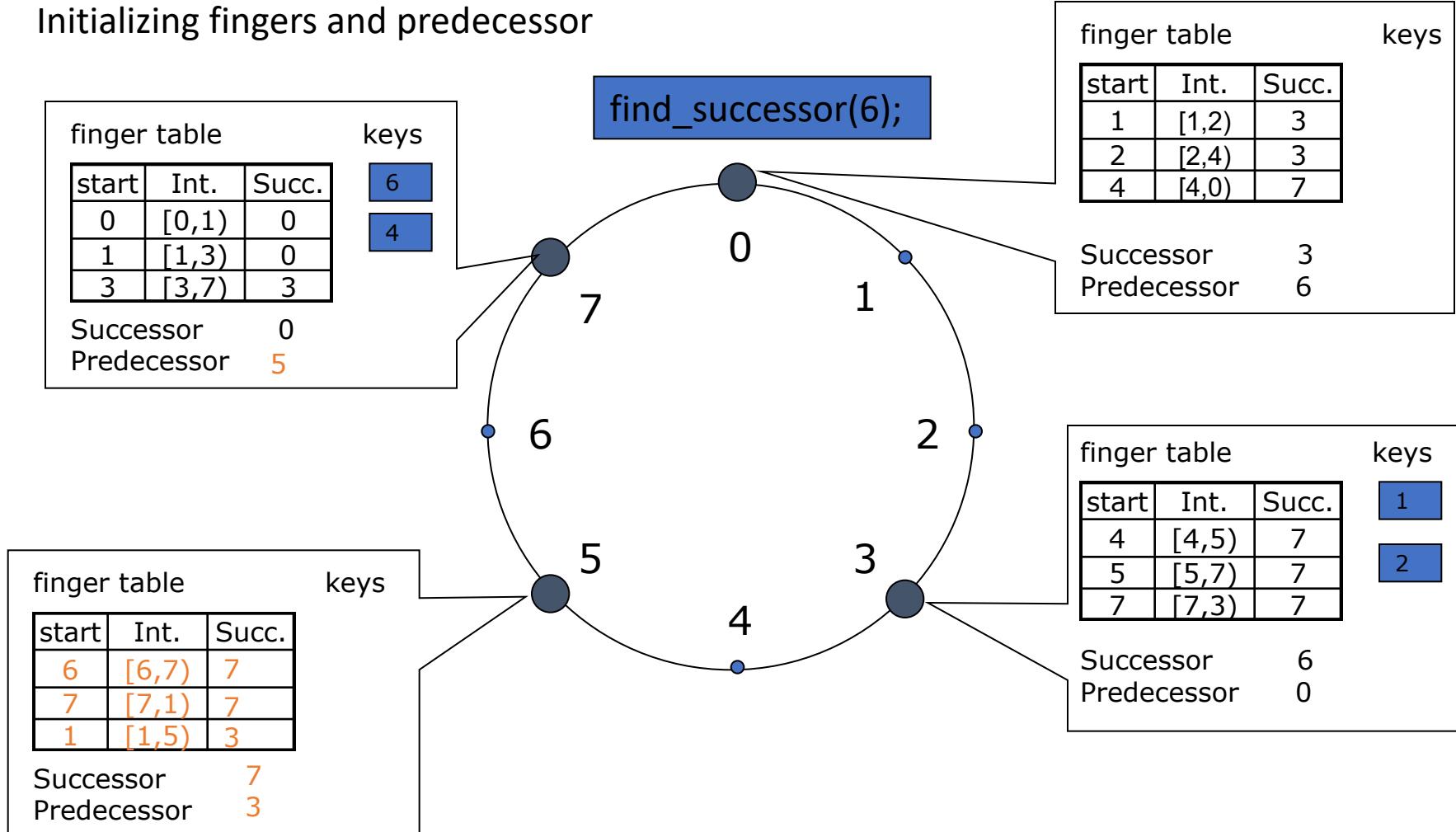
$O(\log N)$

Chord Algorithm - Node joins

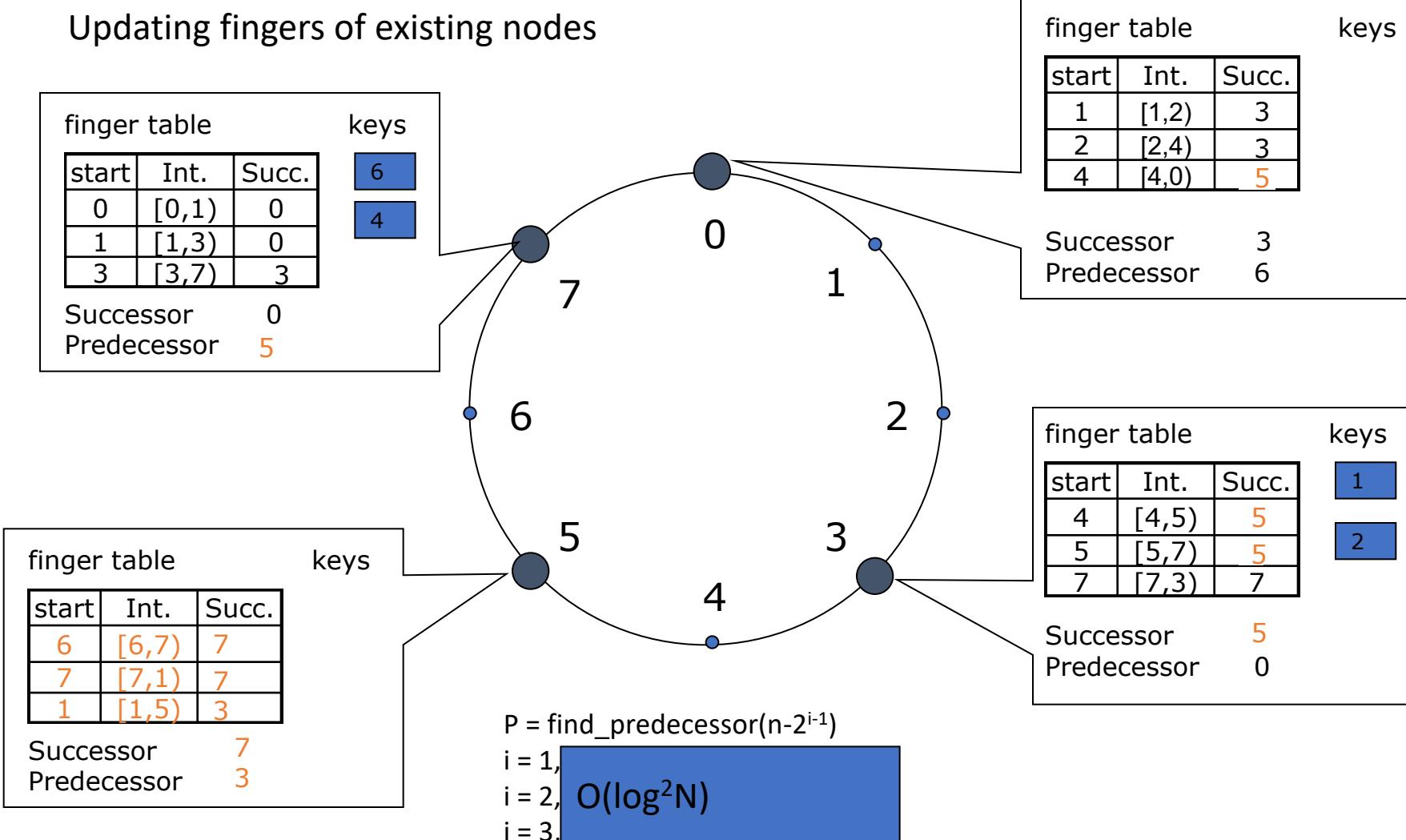
- Invariants to be preserve
 - Each node's successor is correctly maintained
 - For every key k , node $\text{successor}(k)$ is responsible for k
- It is desirable for the finger tables to be correct
- Tasks to be performed by Chord
 - Initialize the predecessor and fingers of node n
 - Update the fingers and predecessor of existing nodes to reflect the addition of n
 - Notify the higher layer software so that it can transfer state associated with keys that node n is now responsible for

Chord Algorithm - Node joins

Initializing fingers and predecessor

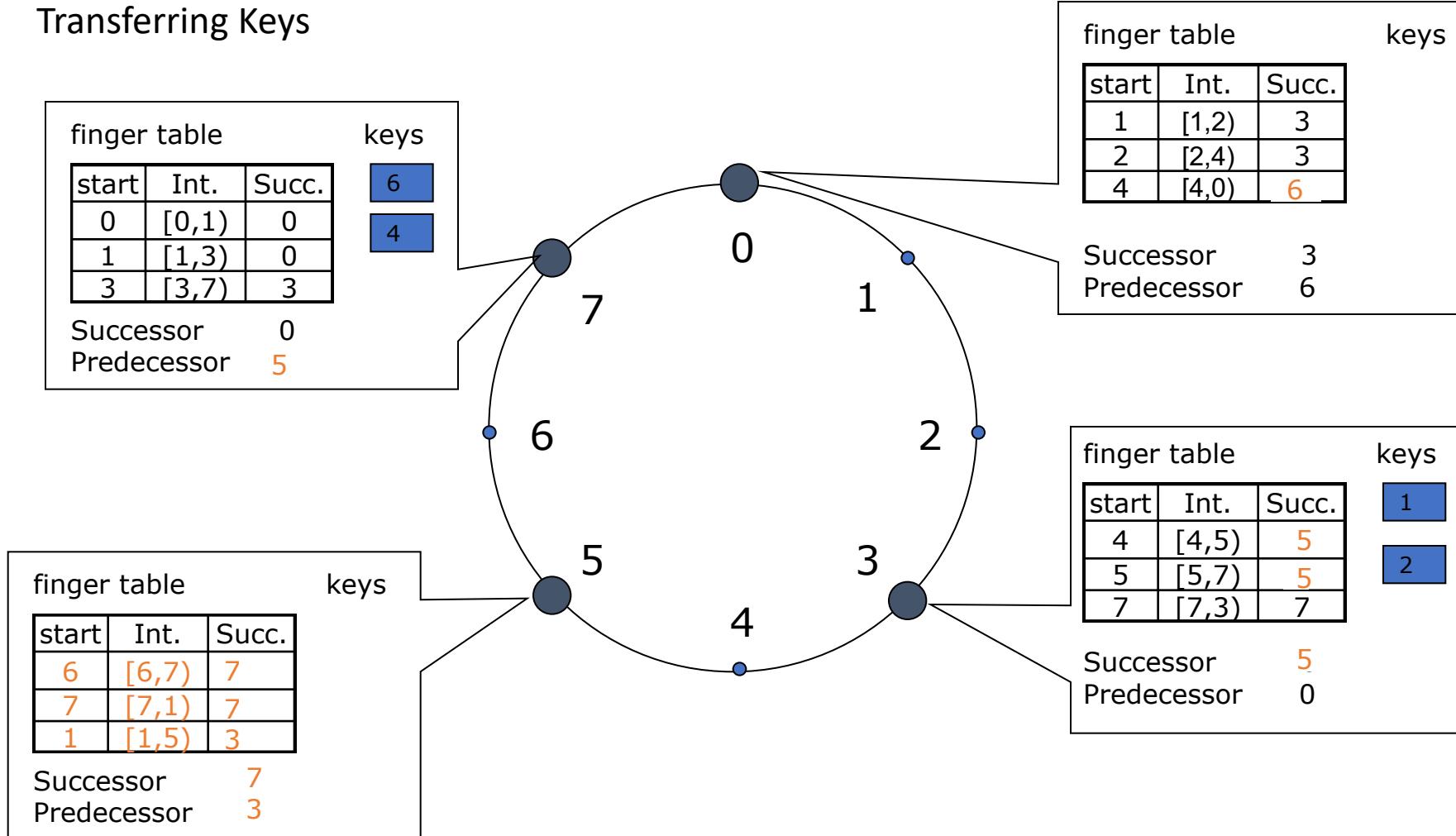


Chord Algorithm - Node joins



Chord Algorithm - Node joins

Transferring Keys



Chord Algorithm – node joins

```
#define successor finger[1].node  
  
// node n joins the network;  
// n' is an arbitrary node in the network  
n.join(n')  
if (n')  
    init_finger_table(n');  
    update_others();  
    // move keys in (predecessor, n] from successor  
else // n is the only node in the network  
    for i = 1 to m  
        finger[i].node = n;  
        predecessor = n;  
  
// initialize finger table of local node;  
// n' is an arbitrary node already in the network  
n.init_finger_table(n')  
    finger[1].node = n'.find_successor(finger[1].start);  
    predecessor = successor.predecessor;  
    successor.predecessor = n;  
    for i = 1 to m - 1  
        if (finger[i + 1].start ∈ [n, finger[i].node])  
            finger[i + 1].node = finger[i].node;  
        else  
            finger[i + 1].node =  
                n'.find_successor(finger[i + 1].start);  
  
// update all nodes whose finger  
// tables should refer to n  
n.update_others()  
for i = 1 to m  
    // find last node p whose ith finger might be n  
    p = find_predecessor(n - 2i-1);  
    p.update_finger_table(n, i);  
  
// if s is ith finger of n, update n's finger table with s  
n.update_finger_table(s, i)  
if (s ∈ [n, finger[i].node])  
    finger[i].node = s;  
    p = predecessor; // get first node preceding n  
    p.update_finger_table(s, i);
```

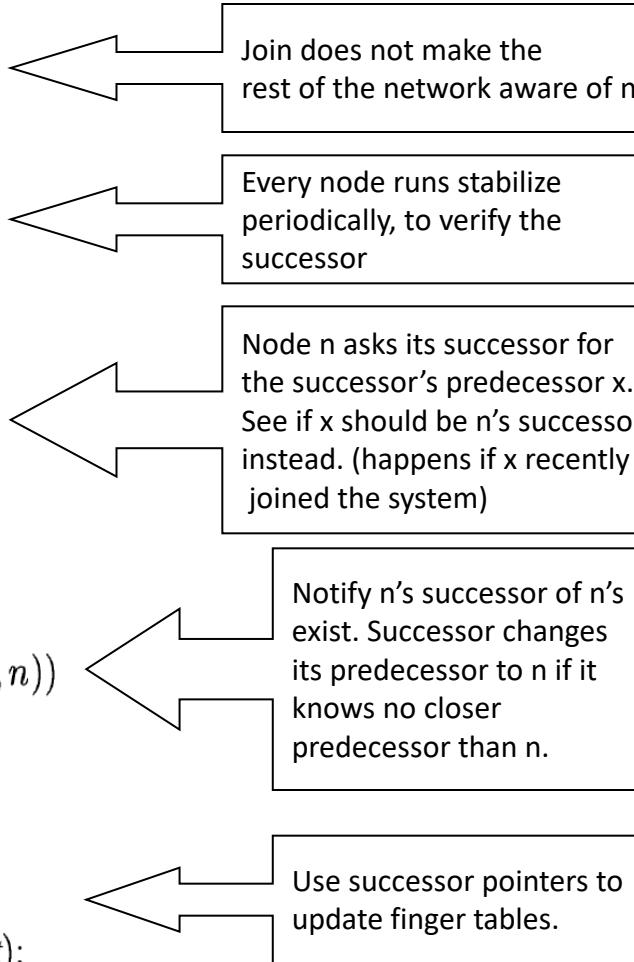
Pseudocode for the node join operation

Chord Algorithm - Stabilization

- Stabilization
 - Correctness and performance
 - Keep node's successor pointers up to date
 - Use successor pointers to verify correct finger table entries

Chord Algorithm - Stabilization

```
n.join( $n'$ )
  predecessor = nil;
  successor =  $n'.find\_successor(n)$ ;  
  
// periodically verify n's immediate successor;
// and tell the successor about n.
n.stabilize()
  x = successor.predecessor;
  if ( $x \in (n, successor)$ )
    successor = x;
  successor.notify( $n$ );  
  
//  $n'$  thinks it might be our predecessor.
n.notify( $n'$ )
  if (predecessor is nil or  $n' \in (predecessor, n)$ )
    predecessor =  $n'$ ;  
  
// periodically refresh finger table entries.
n.fix_fingers()
  i = random index > 1 into finger[];
  finger[i].node = find_successor(finger[i].start);
```



Chord Algorithm – Node Failure

- Node Failure
 - Successor-list
 - If successor fails, replaces it with the first live entry in the list
 - Later run *stabilize* to correct finger table and successor-list

Summary

- The routing table at each node contains $O(\log(n))$ entries
- Inserting and deleting nodes requires $O(\log(n)^2)$ messages
- The hash for the node and key is generated by using the SHA-1 algorithm.