IP Route Lookups

Introduction

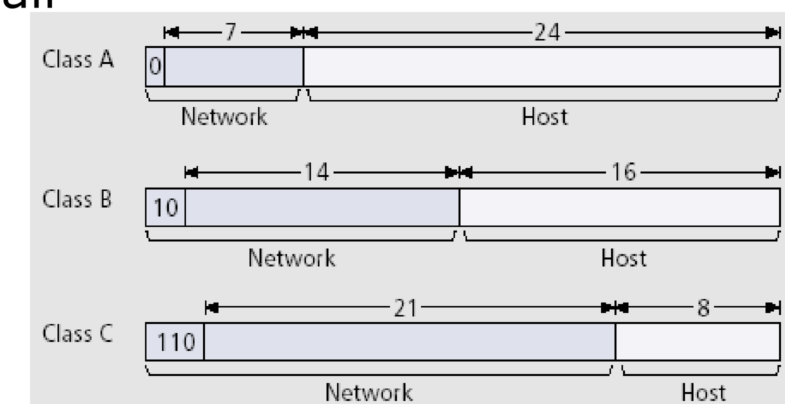
- Growth of the Internet
- Network capacity: A scarce resource
- Good Service
  - Large-bandwidth links -> Readily handled (Fiber optic links)
  - High router data throughput -> Readily handled (Switching technology)
  - High packet forwarding rates -> Key factor

IP Routing

- Packet forwarding tasks
  - Packet header encapsulation and decapsulation
  - Updating TTL field
  - Checking for errors
  - …
  - IP route lookup -> Dominates the processing time

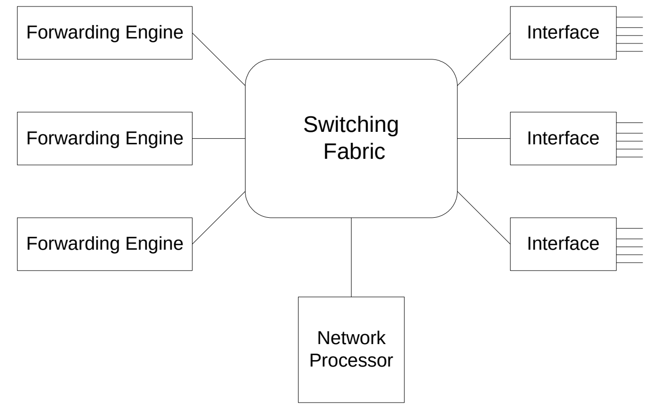
IP Routing – Classful and Classless

- Classful
  - 3 Classes: A, B, and C
  - 2 Levels of hierarchy
  - Wastes address space
- Classless Interdomain Routing (CIDR)
  - Arbitrary aggregation
  - Arbitrary length for host and network fields
  - Routing entry: <prefix/length> pair
    - <12.0.54.8/32>
    - <12.0.54.0/24>
    - <12.0.0.0/16>
  - Efficient routing table size
  - Needs to find the longest match
    - Packet destination: 12.0.54.2
      - Matches: <12.0.54.0/24>, <12.0.0.0/16>
      - <12.0.54.0/24> is used
    - Makes IP route lookup a bottleneck

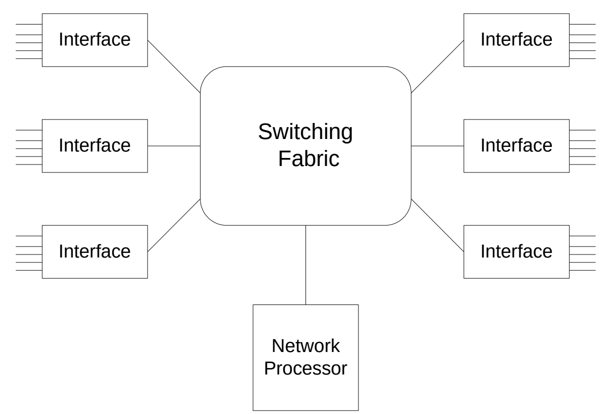


Architecture of generic routers

- With forwarding engines
  - Packet headers go to the forwarding engines
  - Forwarding engines determine the output interface to send the packet



- With processing power on interface
  - Input interfaces determine the output interface to send the packet



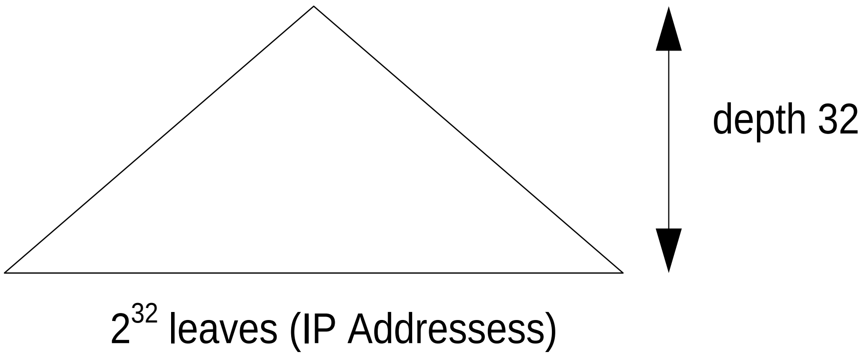
- Forwarding tables
  - Forwarding engine and input interfaces
    - Need not be dynamic
    - Optimized for fast lookups
  - Network processor
    - Dynamic and up-to-date

IP route lookup design

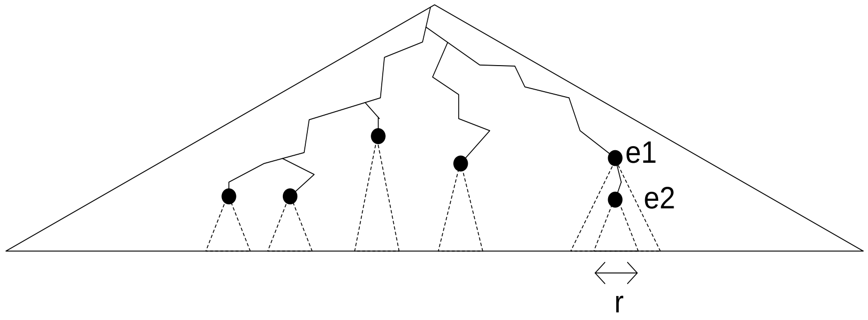
- Goals
  - Minimize time (primary goal)
    - Minimize the number of memory accesses
    - Minimize the size of the data structure
  - Minimize instructions needed
  - Aligned data structures

Route lookup structure

- IP address space
  - A binary tree with depth 32
  - 232 leaves



- <prefix/length> pair
  - Prefix defines a path in the tree
  - Length says how deep the path goes in the tree
  - All IP addresses in the subtree are routed according that entry
  - Longest matching concept
    - Subtrees of entries e1 and e2 overlap
    - e1 is hidden by e2 for addresses in the range r

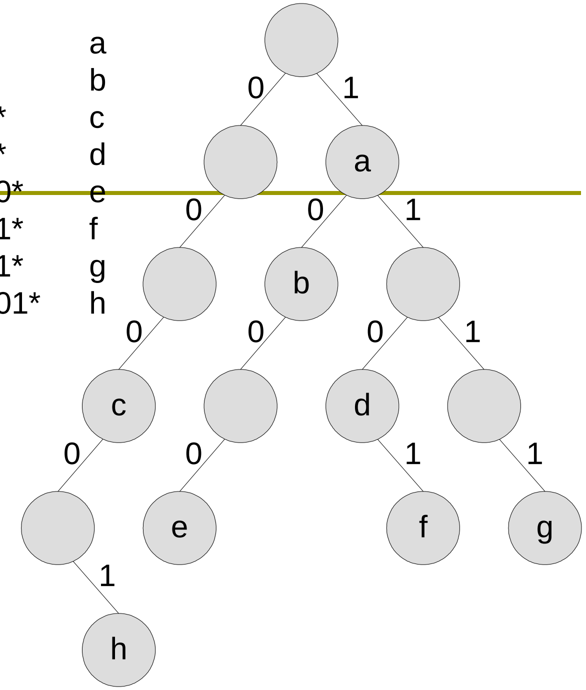


IP route lookup and caching

- Using caching techniques for IP route lookup
  - Relies on locality of destination address stream
  - There is not enough locality for backbone routers
  - Not a good solution for current backbone routers

Trie structure

1*	a
10*	b
000*	c
110*	d
1000*	e
1101*	f
1111*	g
00001*	h



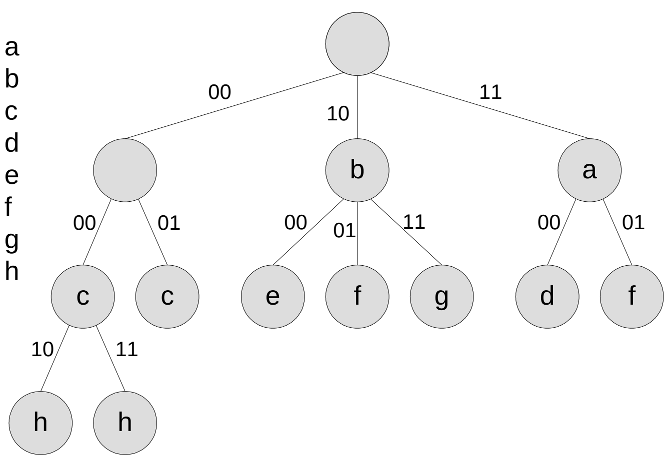
- Represents prefixes of different lengths
- 1-bit trie
  - Left link: 0
  - Right link: 1
  - Search
    - Start from root, move to left or right if the current bit of the address is 0 or 1 respectively
    - If a node containing a prefix mark (*) is seen, store it somewhere as the longest match up to now
  - Addition
    - Follow the path and create new nodes if needed and finally mark the last node as a prefix
  - Deletion
    - Follow the path and delete the last node and its parents until a marked node or a node with another child is seen

Trie level compression

- 1-bit trie: worst case of 32 memory accesses
- Multibit trie (n-bit trie)
  - n bits is checked at each level
  - 2n children for each node
  - Prefix expansion -> more memory usage

1*	(a)	10* 11*	(b) (a)
10*	(b)		
000*	(c)	0000* 0001*	(c) (c)
110*	(d)	1100* 1101*	(d) (f)
1000*	(e)		
1101*	(f)		
1111*	(g)		
00001*	(h)	000010* 000011*	(h) (h)

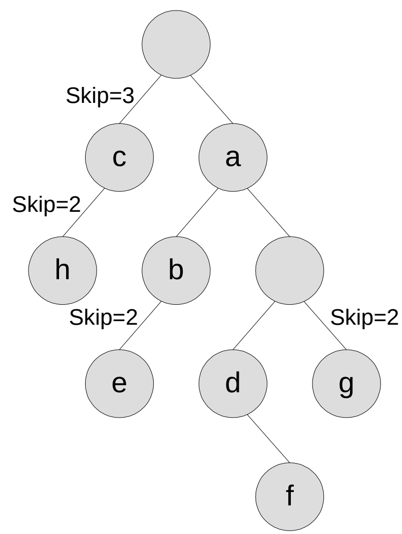
1*	a
10*	b
000*	c
110*	d
1000*	e
1101*	f
1111*	g
00001*	h



Trie path compression

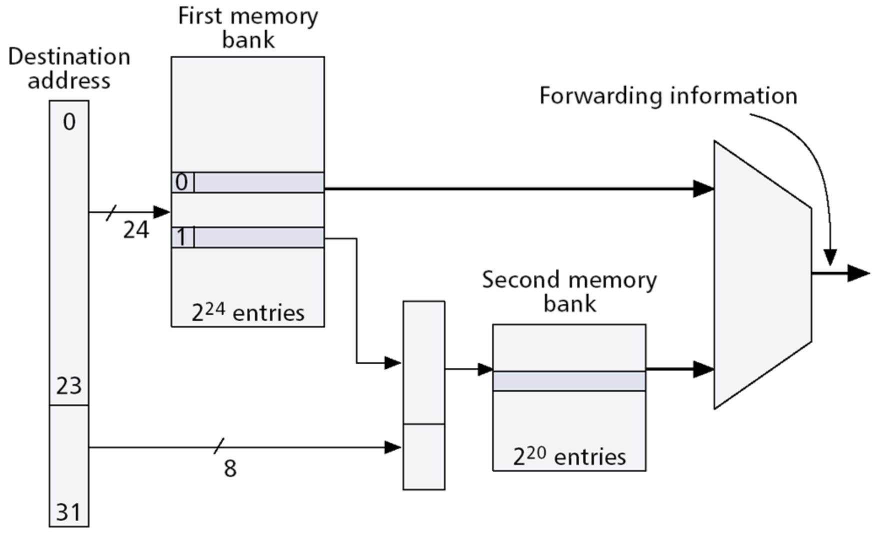
- PATRICIA trie
  - Remove nodes with one child without prefix
  - Store the number of removed nodes (Skip values)
  - Only useful in sparse tries, not backbone routing tables

1*	a
10*	b
000*	c
110*	d
1000*	e
1101*	f
1111*	g
00001*	h



DIR-24-8 implementation

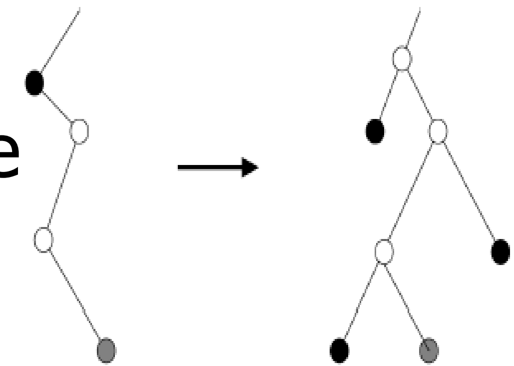
- Gupta et al.
- Two levels
  - First memory bank: 24 bits of address
  - Second memory bank: 8 bits of address
- Performance
  - Two pipelined memory accesses per lookup
  - DRAM delay of 50ns => 20 mlps
  - 33 Mbytes of DRAM



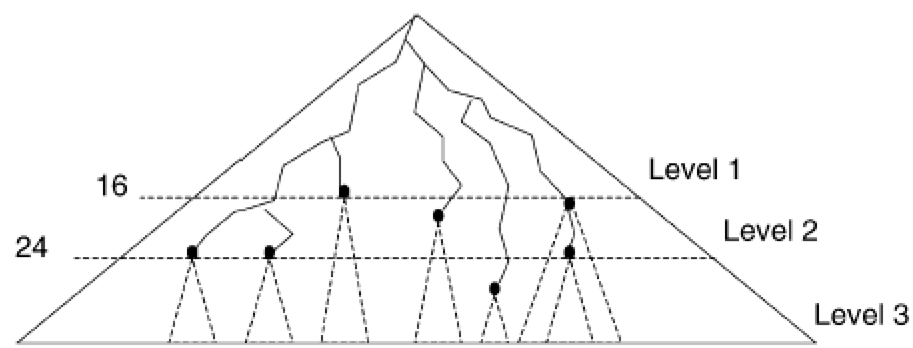
- Drawbacks
  - High memory usage
  - Many memory places may need to change for an update

Degermark et al. scheme

- Degermark et al. scheme
  - Large routing table in a small data structure
  - Small enough to fit in cache
  - Fast lookup in software
- Prefix tree needs to be complete
  - Each node: 0 or 2 children
  - Expanding the tree

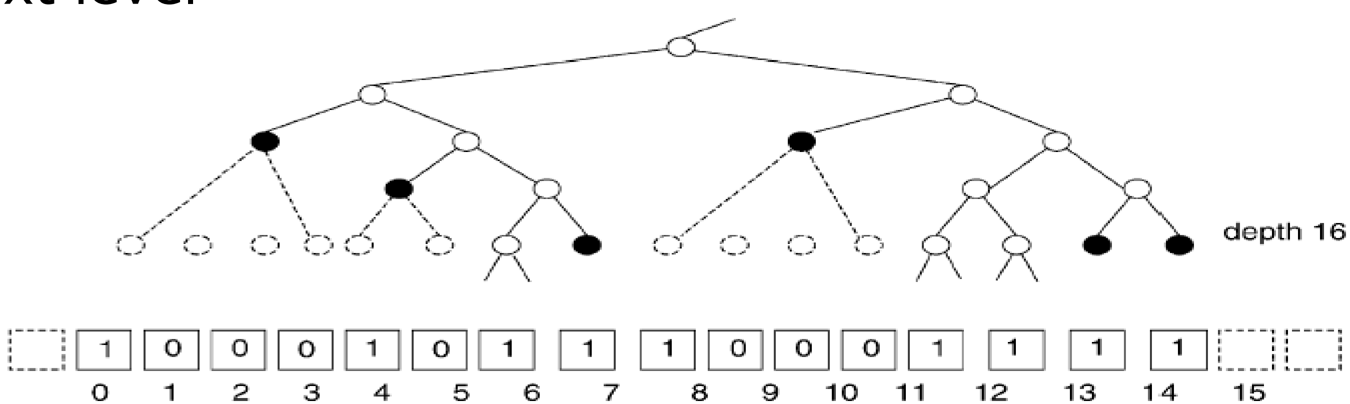


- Three levels
  - Level 1: depth 1-16
  - Level 2: depth 15-24
  - Level 3: depth 25-32



Degermark et al. scheme

- Level 1 of the tree
  - A bit vector
  - Representing a cut in depth 16
    - If tree continues below the cut => bit=1 (root head)
    - If a leaf is located in depth 16 or less
      - A range is spanned by that leaf in depth 16
      - The least significant bit of the range is set to 1 (genuine head)
    - Other bits are set to zero
  - For root head we store an index to NHP table
  - For genuine head we store an index to a subtree in the ext level

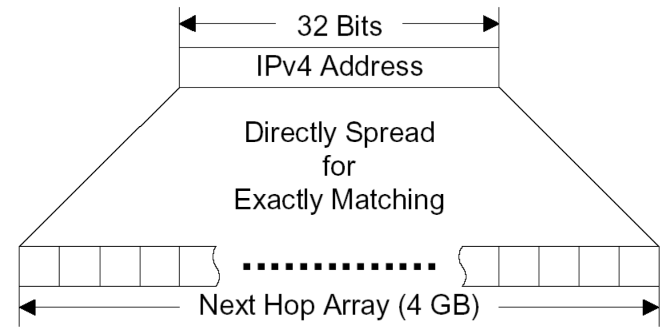


Degermark et al. scheme

- Search algorithm for level 1
  - Some bit extractions, array references and additions
  - 7 bytes of accesses to the memory
  - 10 Kbytes of memory usage
  - (A 2D array of 5.3 Kbytes is also used, but it is shared among all levels)
- Level 2 and 3
  - Some chunks indexed from the previous level
  - Each chunk: Depth of 8 (Possible 256 heads)
    - Sparse: 1-8 heads
    - Dense: 9-64 heads
    - Very dense: 65-256 heads
  - Dense and very dense chunks are searched like level 1
  - Sparse chunks are stored sorted and searched with at most 7 memory accesses

Huang et al. scheme

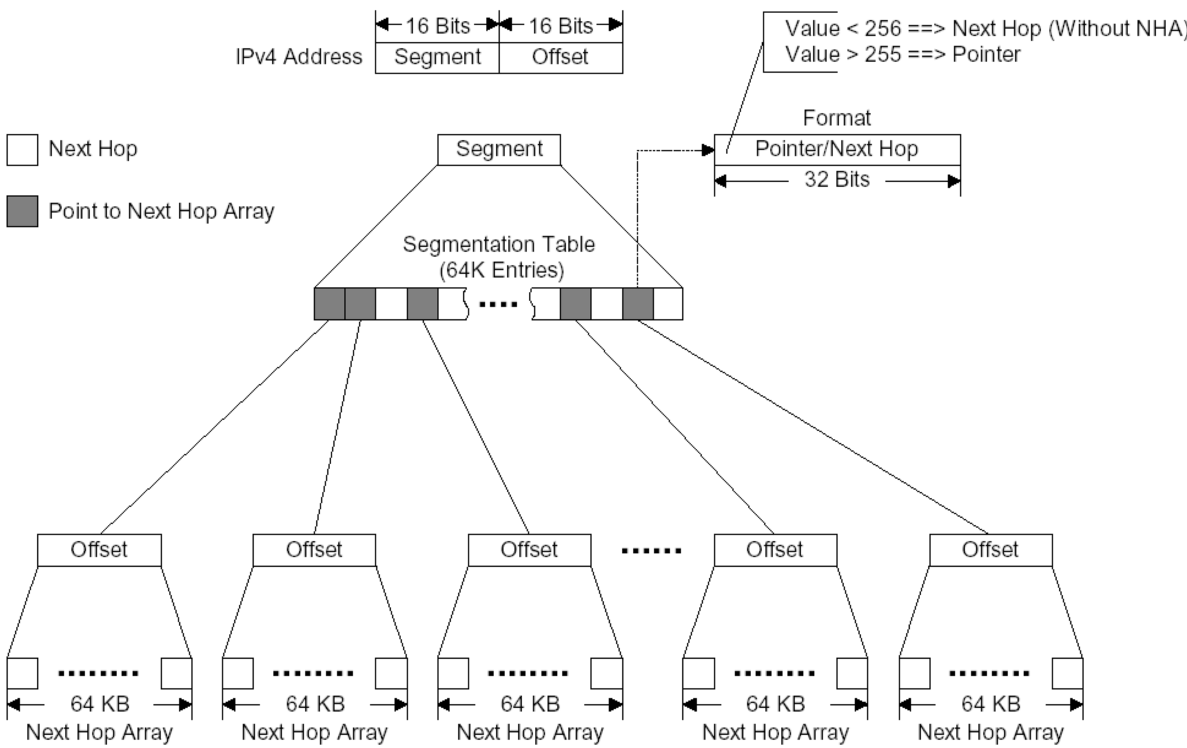
- Huang et al. scheme
  - Same as DIR-24-8, but
    - Uses variable length offsets to consider prefix distribution
    - Compresses routing data
  - Worst case of 3 memory accesses per lookup
  - 450-470 Kbytes memory usage



- Simplest case: Direct lookup
  - Expand all prefixes to 32 bits
  - 1 memory access per lookup
  - 4 GBytes memory usage

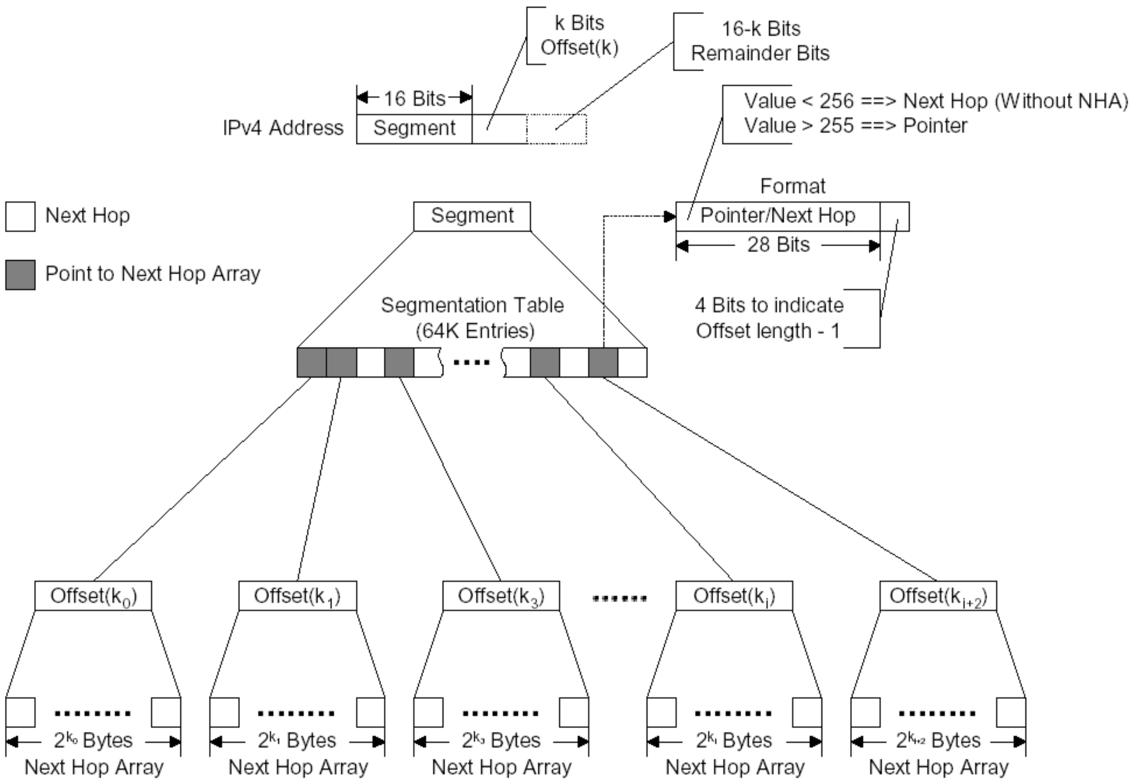
Huang et al. scheme

- Indirect Lookup
  - Break the address space to two levels
  - Same idea as DIR-24-8



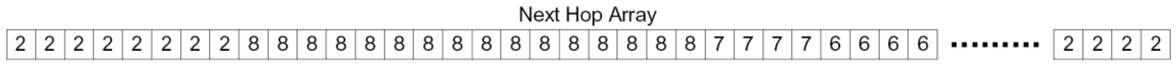
Huang et al. scheme

- Indirect lookup with variable length offsets
  - Reduces NHA sizes

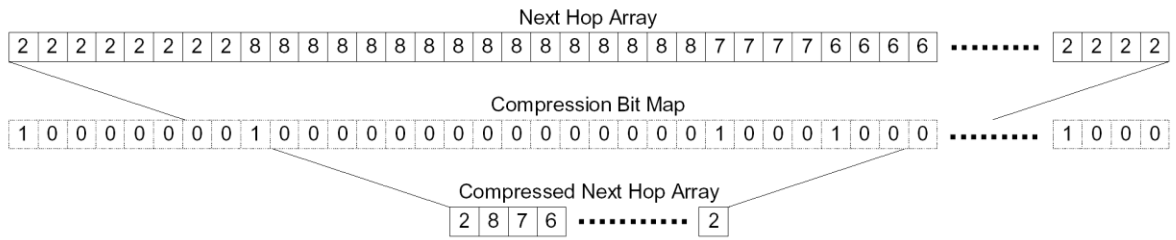


Huang et al. scheme

- NHA data compression
  - Redundancy

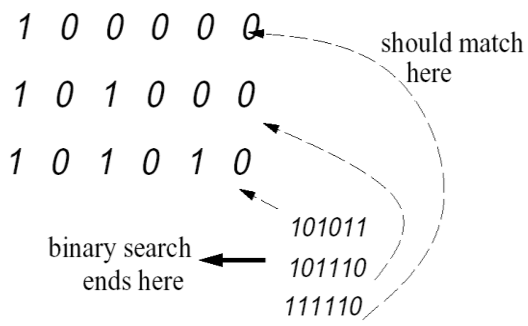


  - Compression



Multiway Search (Lampson et al.)

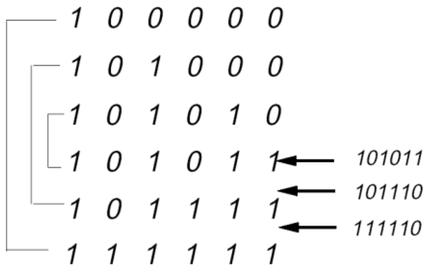
- Standard multiway search
  - Useful for exact matching
  - Needs modification for longest ma



- Basic idea
  - Consider 1*, 101*, 10101* prefix
    - Pad them to become of same length
    - Binary search incorrectly fails for these addresses
      - 101011, 101110, 111110
  - Two problems
    - Search may end up far away from the correct answer
    - Multiple addresses with different matching prefixes may end up in the same region

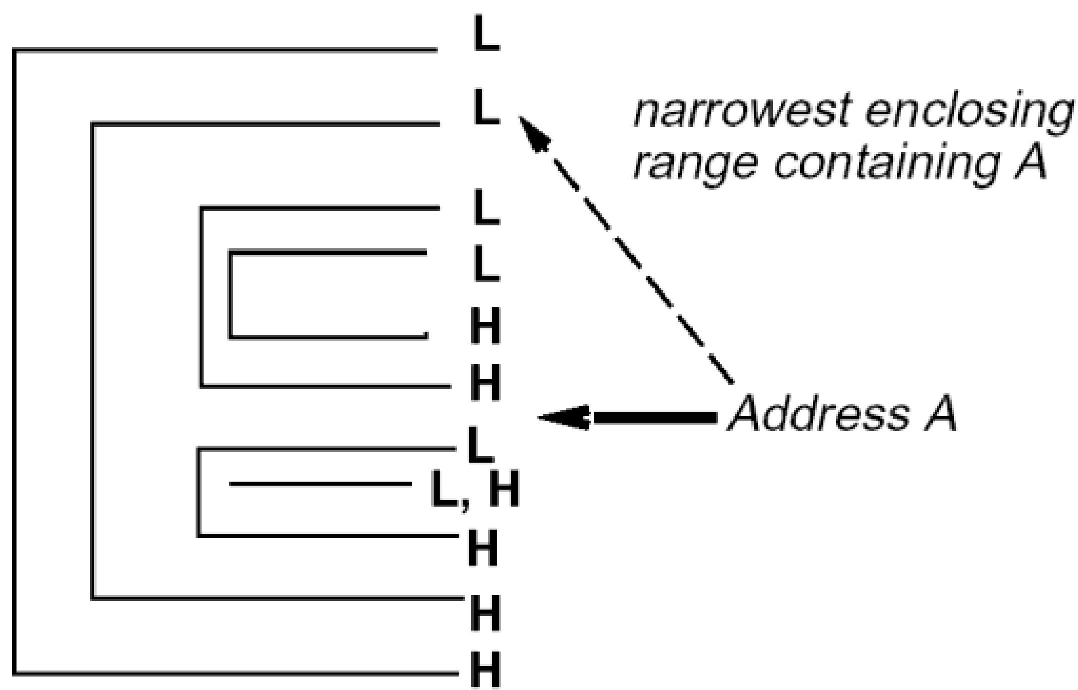
Multiway Search (Lampson et al.)

- Considering prefixes as ranges
  - Consider each prefix as a range
  - Expand each prefix to start and end of the range
    - 1* becomes 100000 and 111111
  - Solves the second problem



Multiway Search (Lampson et al.)

- The first problem
  - A linear search is needed to find the correct match



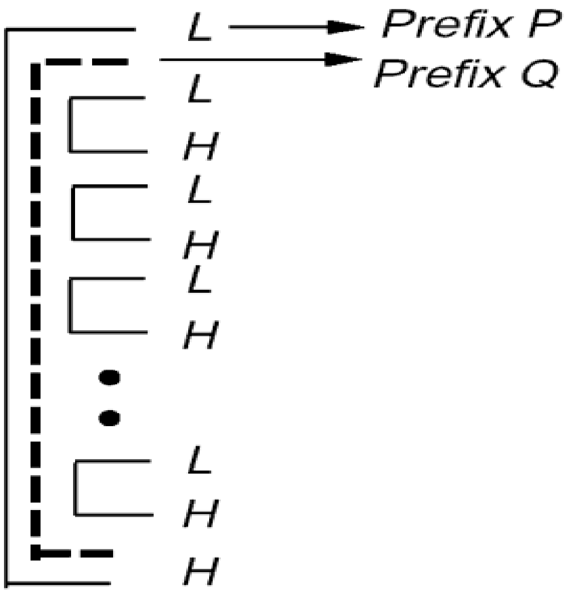
Multiway Search (Lampson et al.)

- How to solve the first problem
  - Precomputed pointers
  - For each row:
    - A pointer for when the binary search finishes with a hit (= pointer)
    - A pointer for when the binary search finishes with a fail (> pointer)

							>	=
P1)	1	0	0	0	0	0	P1	P1
P2)	1	0	1	0	0	0	P2	P2
P3)	1	0	1	0	1	0	P3	P3
	1	0	1	0	1	1	P2	P3
	1	0	1	1	1	1	P1	P2
	1	1	1	1	1	1	–	P1

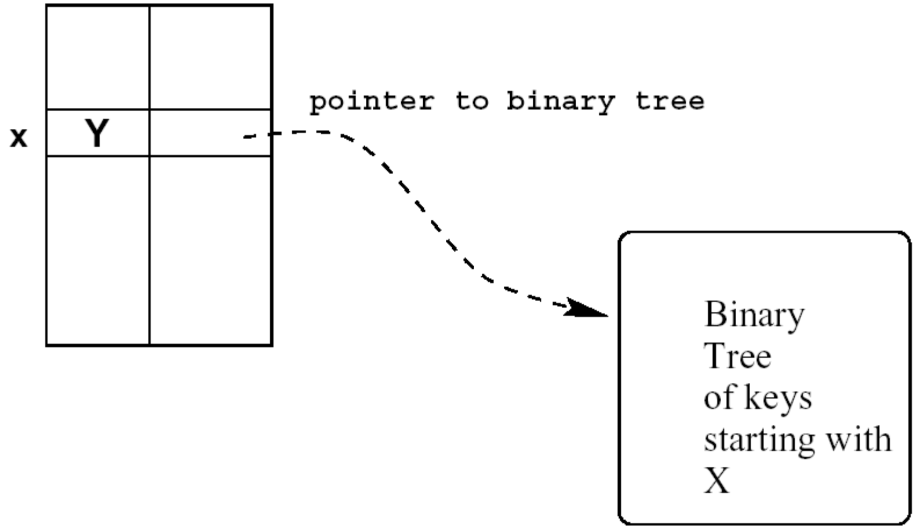
Multiway Search (Lampson et al.)

- Table construction
  - A push/pop algorithm to calculate pointers
- Prefix insertion and deletion
  - Many pointers may become invalid
  - High overhead
  - Batching may help



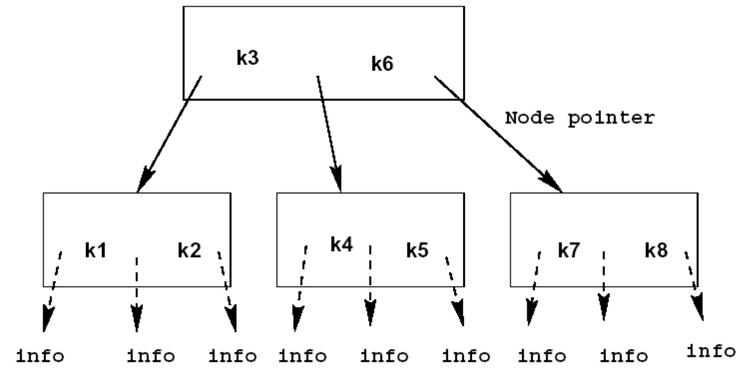
Multiway Search (Lampson et al.)

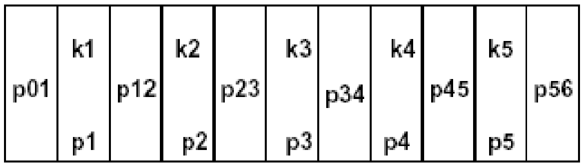
- Partitioning the problem
  - Inspect first Y bits of the address directly
  - This points us to one of 2Y subtables



Multiway Search (Lampson et al.)

- Multiway search
  - k keys
  - 2k+1 pointers per node
  - Logk+1 N comparisons (w
  - As large k as possible that fits in the CPU cache line
  - For Pentium Pro: k=5





Multiway search (Lampson et al.)

- Results
  - For 30000 entries
  - Considering 16 bits initial array
  - Worst case subtable: 336 entries
  - => Worst case of 4 memory accesses
  - On Pentium Pro 200 Mhz
    - 490ns worst case search time per lookup
    - 130ns average time per lookup
    - 1.7MB memory usage

Two- trie structure

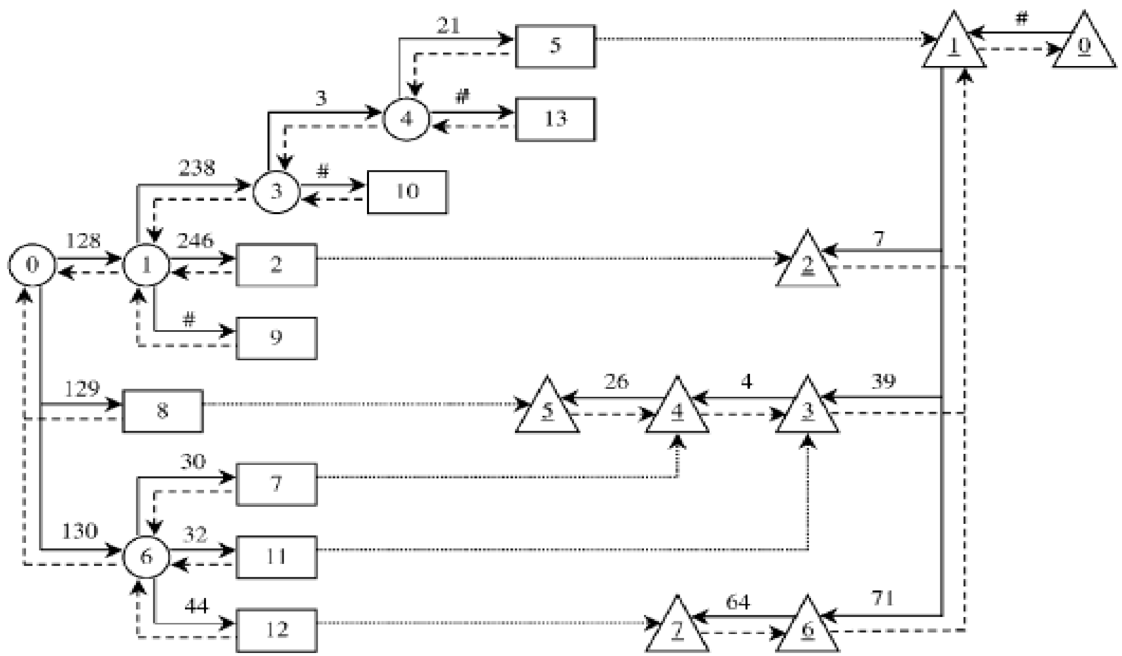
- Two- trie structure
  - Nodes representing front and rear part of the prefix are shared
  - Originally by Aoe et al. (general)
  - New version by Kijkanjanarat et al. for IP lookup
- K-bit two trie
  - Consists of two K-bit tries
    - Front trie
    - Rear trie
  - Joining leaf nodes in the middle
  - Both trie can be traversed in both direcrtions
    - Forward direction : from root to child
    - Backward direction: from child to root

Two- trie structure

- Assume prefix X of length Y bits
  - X is represented as <x(0).x(1)…x(N)>
  - x(i), (i between 0 and N-1) is the K-bit part of prefix X
  - x(N) is a special symbol # , N = [Y/K]
  - If Y is not a multiple of K, the prefix will be expanded to a set of prefixes

Two- trie structure

- An example
  - Triangles: Nodes of the rear trie
  - Circles: Nodes of the front trie
  - Rectangles: Separate nodes (Leaf nodes of the front trie



Two- trie structure

Algorithm IPLookup (X)

1. Let Z be the variable that stores the next hop of the longest matching prefix. Initially Z is the default next hop.
2. Start to do an IP lookup from the root node of the front trie by matching each K-bit part of the destination address X of the packet with prefixes in the two-trie structure.
3. If there is a match, the traversal is moved to the child node at the next level of the front trie.
4. Whenever a new front node is arrived at, the algorithm first looks for its child node corresponding to the symbol # (which must be the separate node). If the node is found, it means that the two-trie structure contains a longer matching prefix, so the variable Z is updated with the next hop value of this prefix retrieved from the separate node.
5. When the separate node is reached, matching continues to the rear trie by using a pointer at the separate node (shown as a dashed line in Fig. 13.29). Matching on the rear trie is done in the backward direction.
6. The algorithm stops whenever
   (a) a mismatch is detected somewhere in the structure (in such a case, the current value of Z is returned as the next hop), or
   (b) the traversal reaches the root node of the rear trie (no mismatch is detected). This means that the destination address X of the packet is actually stored as a prefix in the structure. The variable Z is updated with the next hop value of the prefix stored at the separate node we previously visited and returned as the output of the function.

Two- trie structure

- Performance
  - Memory accesses

Structure	Bits for Each Level	Average Case
Two-trie	8, 8, 8, and 8	3.6
Two-trie	1, 6, 8, and 8	1.6
Standard trie	8, 8, and 8	2.1

  - Memory usage

Structure	Bits for Each Level	Memory Requirements (Mbyte)
Two-trie	8, 8, 8, and 8	11.6
Two-trie	16, 8, and 8	11.6
Standard trie	8, 8, and 8	16.0