CS 7260 Lecture 8 Notes

Notes by **Daewon Lee**

Quick review of the previous lecture

Completely expanding a TRIE at a node corresponds to a lookup table at the said node, which contains $2^{height(r)}$, where r is the said root node. This can be expressed as following;

$$opt(r,1) = 2^{height(r)},$$

where r is root node, 1 is the number of strides left.

Variable-Stride Approach (to IP address lookup)

- Solution to optimal TRIE can be solved by dynamic programming with following optimization function

$$opt(r,m) = \min_{1 \le j \le 32 - level(r)} \left\{ 2^j + \sum_{t \in forest(r,j)} opt(t, m-1) \right\},$$

where "forest(r,j)" is the number of leaf nodes with more TRIEs required to perform all IP lookup (i.e. Not a stump)

- We can solve by the dynamic programming by following pseudo code. ε is empty prefix (of an IP address), k is the maximum stride allowance.

```
main {
        opt( ε, k )
}
```

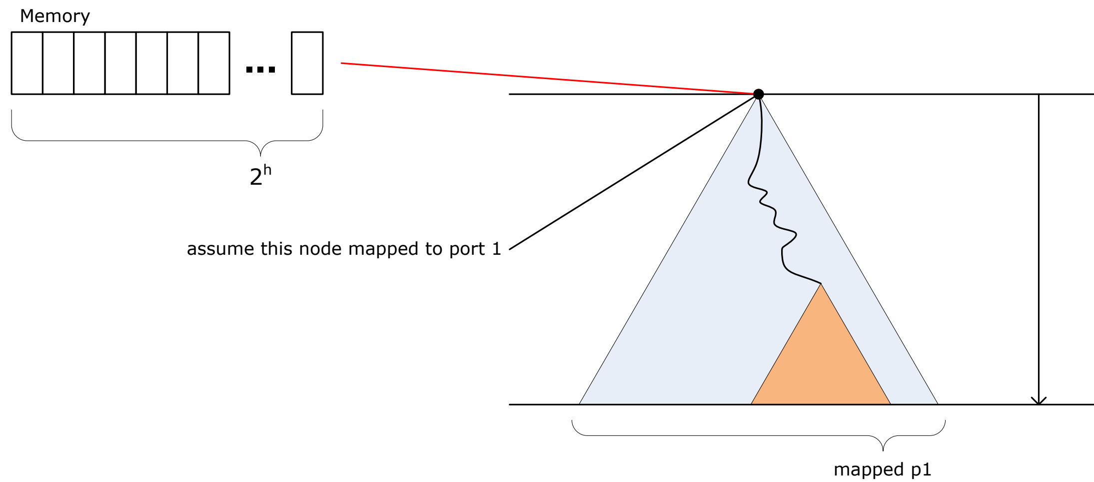
- The goal of the dynamic programming is to get "small memory footprint" (small required memory size).

Multi-bit TRIE Cost

- Multi-bit TRIE potential reduces require memory footprint
- There exist some cost to implementation of multi-bit TRIE due to IP address to output port mapping changes

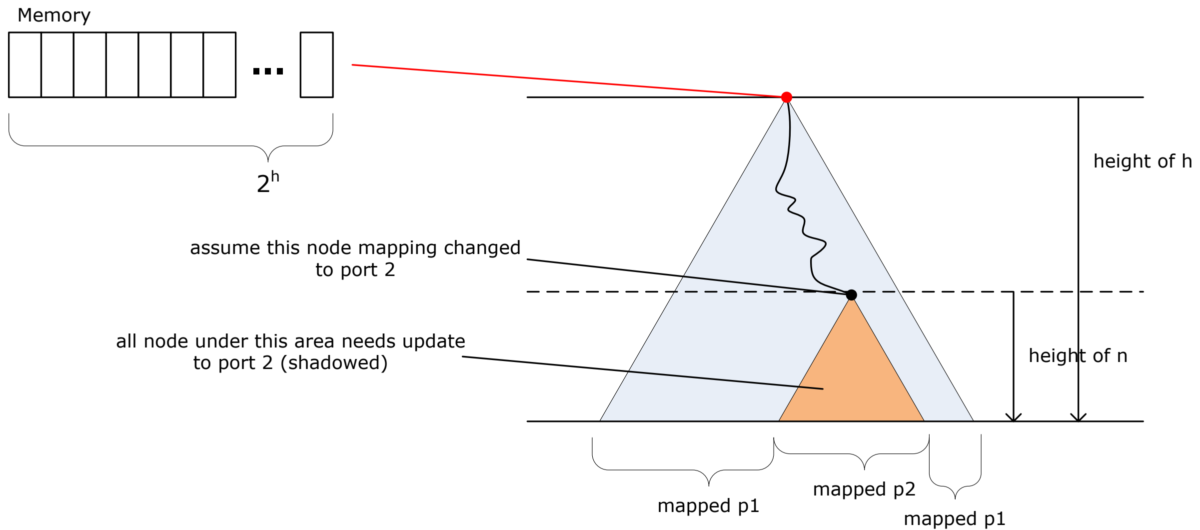
Example description of the multi-bit TRIE cost to dynamic changes of the mapping of the ports:

The following shows data structure of the multi-bit TRIE



The node with height of h, requires 2^h of memory space.

If we change a mapping of the node within the TRIE (e.g. figure below), then we would need to change all corresponding memory entries below the changed node.



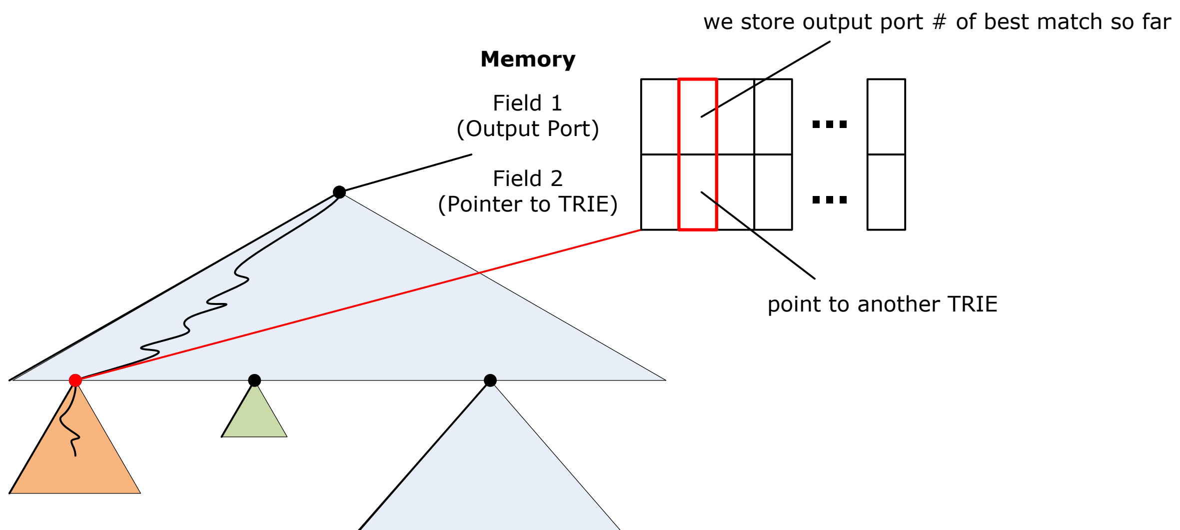
In the example 2^n entries out of 2^h entries needs to be updated due to change. The leaf nodes under the changed node are "shadowed" and are also changed to output port 2.

Data structure for the multi-bit TRIE

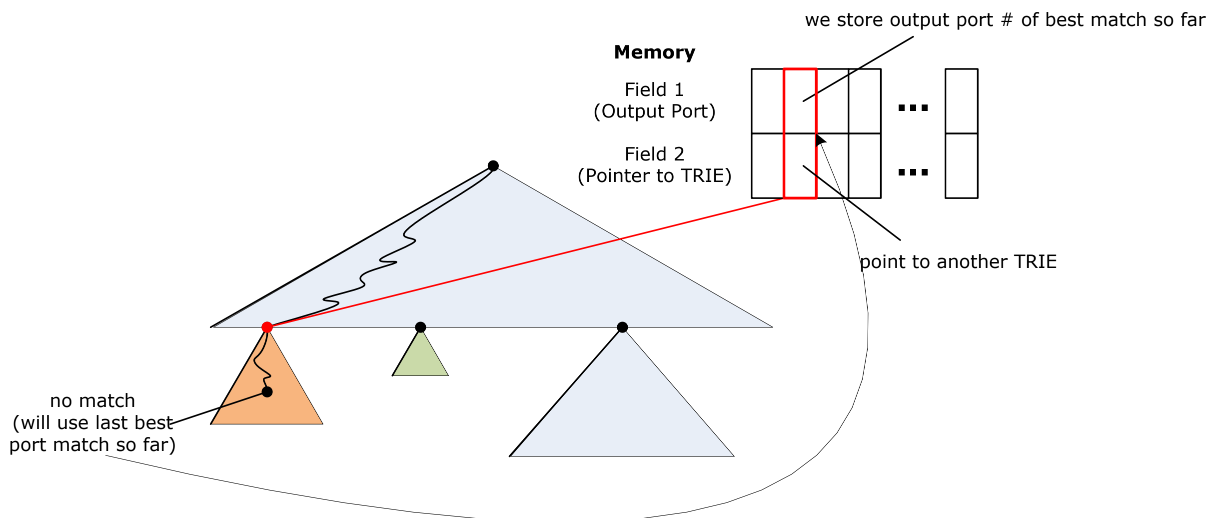
We actually do not have TRIE's structurally. We will implement TRIE in forms of memory vectors with output port as 1st field, and pointer to next TRIE as the 2nd field. There also small additional space needed to store depth of the TRIE, etc. (This is very small and will not be considered as cost).

For each TRIE, we store 2 fields per node.

- Field 1 : the output port number for given node
- Field 2 : the pointer to the next TRIE. Note that not all nodes will contain pointers, if the node is a stump (a node which does not require a longer prefix matching and thus does not require a further TRIE lookup) it will correspond to a null pointer. (a stump : will have a null pointer)



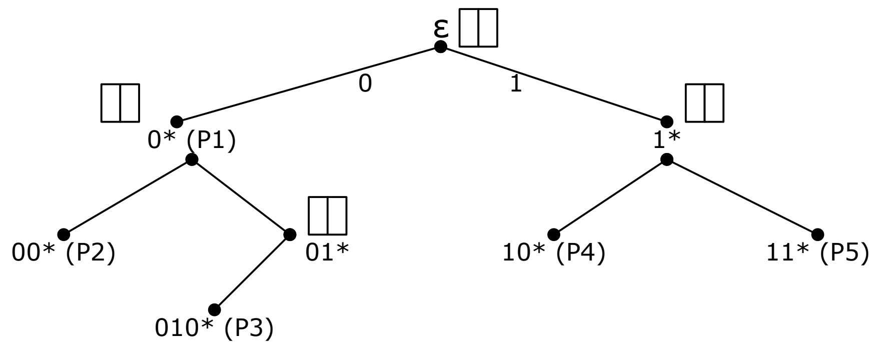
If a lookup results in no exact match, then it will use the best match up to the last lookup.



Example of data structure implementation

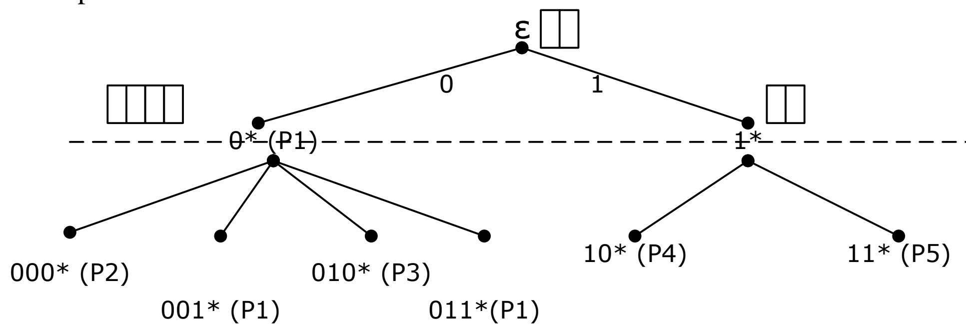
IP Prefix	Output Port
0*	P1
00*	P2
010*	P3
10*	P4
11*	P5

Binary TRIE implementation



- This implementation will cost maximum of 3 lookups, with 8 memory entries

Multi-bit TRIE implementation



- This implementation has maximum of 2 lookups, with 8 memory entries

Binary Search (Prefix) Ranges

- Instead of searching making prefix exact match lookup, we can search on the ranges of the IP prefix

Example of IP prefix range search: (assume that IP address is only 4 bits long)

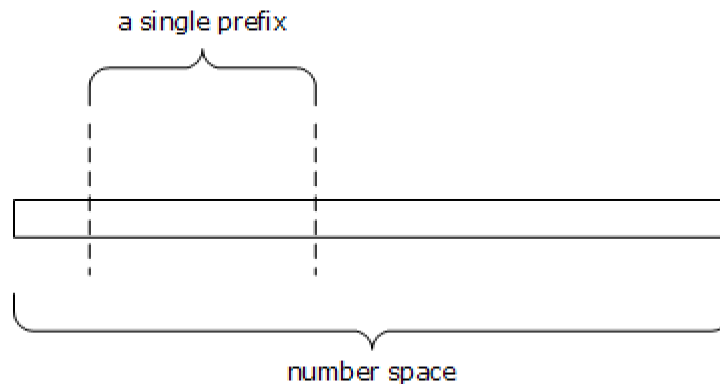
IP Prefix	Output Port
*	(default) P1
1*	P2
101*	P3

We can convert the IP prefix to range numbers
$(1000)_2 = 8$
$(1010)_2 = 10$
$(1011)_2 = 11$

Logic of the algorithm is

$$\text{If IP address is in } \begin{cases} [0,8) & \text{, then P1} \\ (8,10) & \text{, then P2} \\ [10,11] & \text{, then P3} \\ [11,15] & \text{, then P1} \end{cases}$$

- Every prefix divides a number range into two parts (called cuts)
- Each prefix cuts into the number space and divide them
- If we have N prefix, then the maximum of parts is 2N + 1



- Sort out the range boundaries, given a number we can find the where the number fits inside which range

- this is not good enough, because the number of operations depends on number of prefixes.
- - a better algorithm would be where we have log of something, where it is dependent on number of IP address bits

Binary search on Prefix Length

- first we assume there exist a magic data structure, we will describe an algorithm to perform on the data structure

Example of the binary search on prefix length

- x : an IP address number (assume 32 bit address)
- *f16*(·) : denote the first 16 bits of the address

```
check if f16( x )  is a prefix[1] at least one IP prefix[2] rule then

    check if f24( x ) is a prefix is at least one IP prefix rule then
        …
    else
        …
    end

else

    check if f8( x ) is …

end
```

[1] : the bit mask
[2] : aggregation of bunch of IP address

- the algorithm goes on searching the number space by *bisection*
- *f16*(·) is the half of 32 bit number space, *f24*( ) is the upper quarter of the 32 bit number space, *f8*(·) is the lower quarter of the 32 bit number space
- We still need a data structure which allows this algorithm to work. This will be discussed in the next lecture