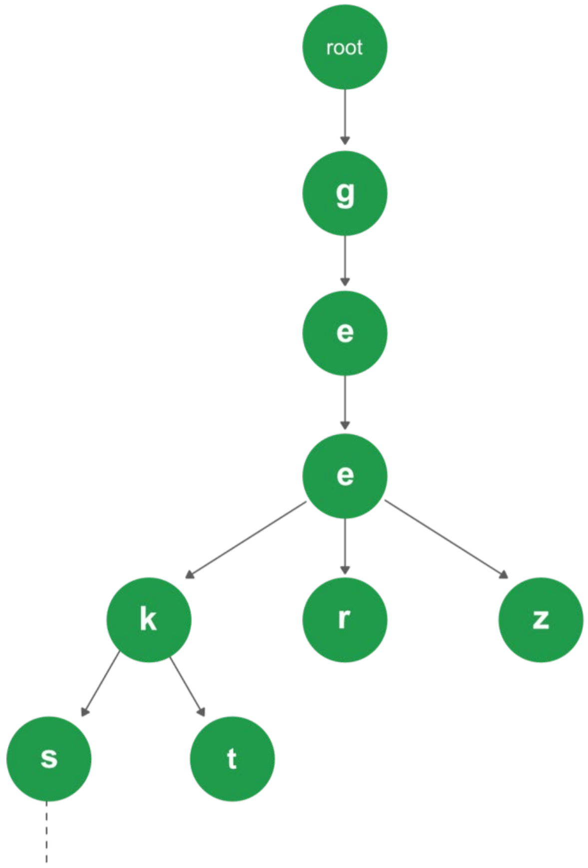
CS2040S – Data Structures and Algorithms

Lecture 14 – *Trie

chongket@comp.nus.edu.sg

NUS
National University of Singapore
School of Computing

Trie – A special ordered map for strings



Motivation for using Trie

Cons of using hash table for Strings	Cons of using AVL for Strings
High overhead: Requires additional O(K) time to convert string of length K into integer value before hash function can be applied	High overhead: Even though keys are ordered lexicographically, time complexity of insertion/deletion/search operations take O(KlgN) time where K is longest key length. Up to O(lgN) nodes must be compared and each comparision takes O(K) time
Does not maintain lexicographical ordering of the String keys	need to do linear search through the string to check for correct lexicographical order

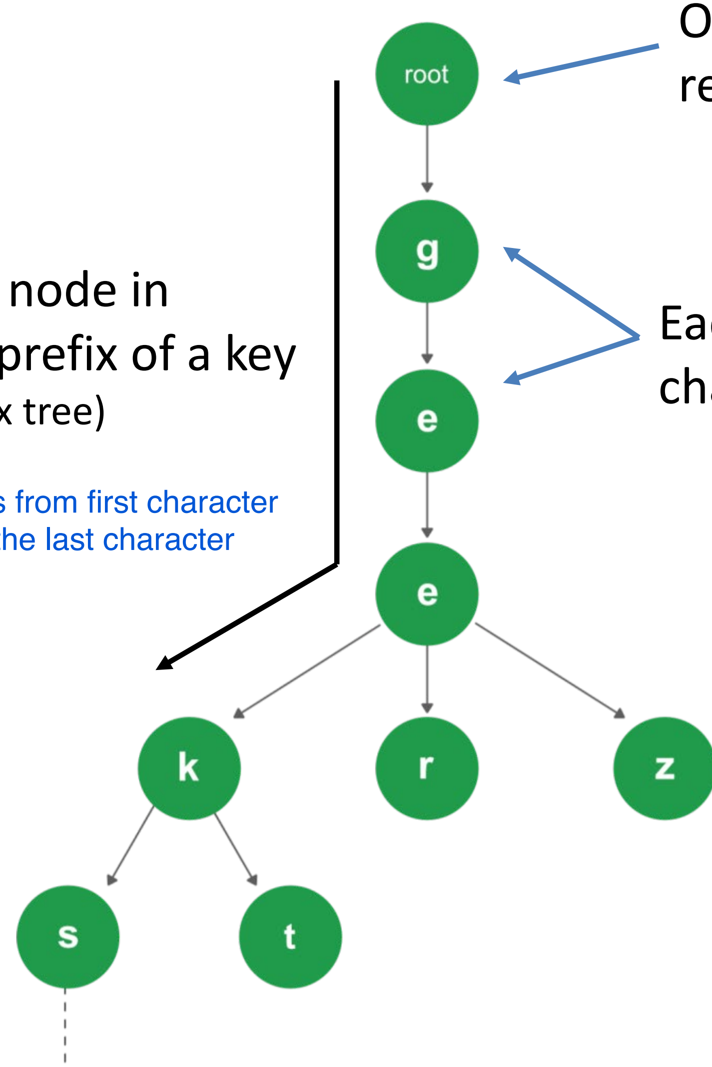
- A Trie addresses the weaknesses of both hash table and AVL for storing String keys

Anatomy of a Trie (1)

Only root does not represent a character

Path from root to some node in the Trie represents the prefix of a key
(thus Trie also known as prefix tree)

prefix- any given substring that starts from first character
suffix - any substring that ends with the last character

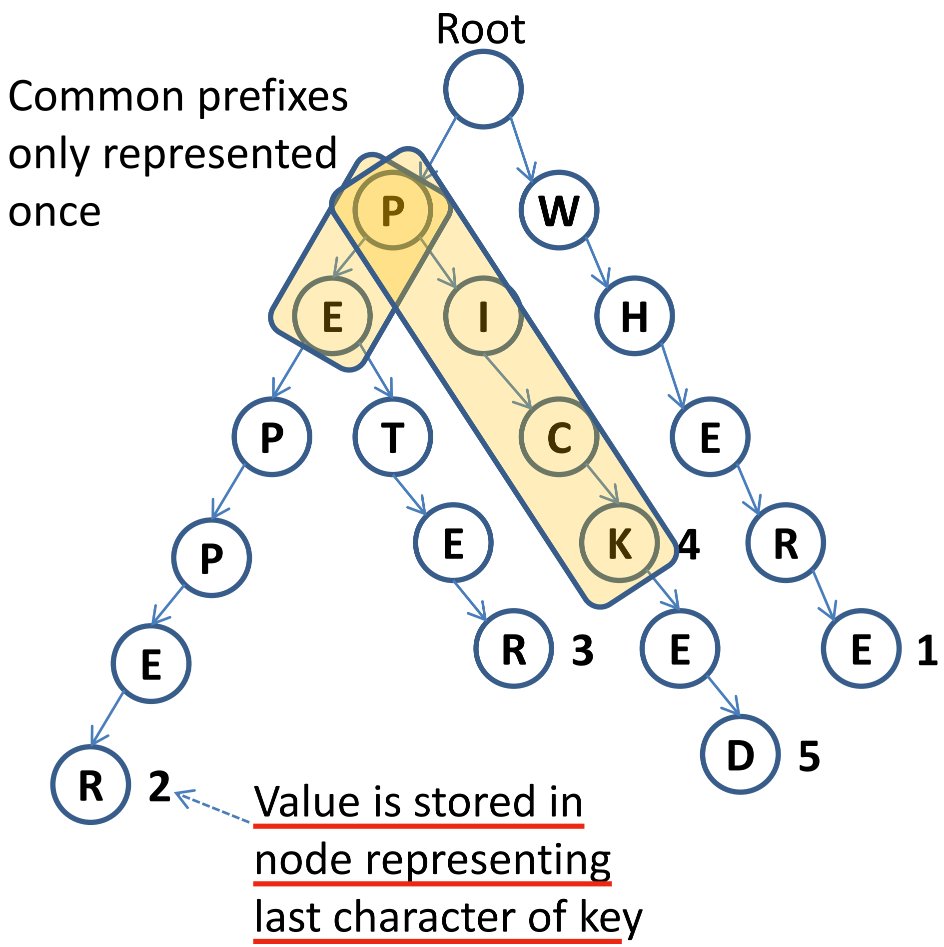


Each node represents a character of a String key

Anatomy of a Trie (2)

Key/Value Pairs

WHERE 1
PEPPER 2
PETER 3
PICK 4
PICKED 5



Operations on Trie

- 3 basic operations
  - Retrieval
  - Insertion
  - Removal

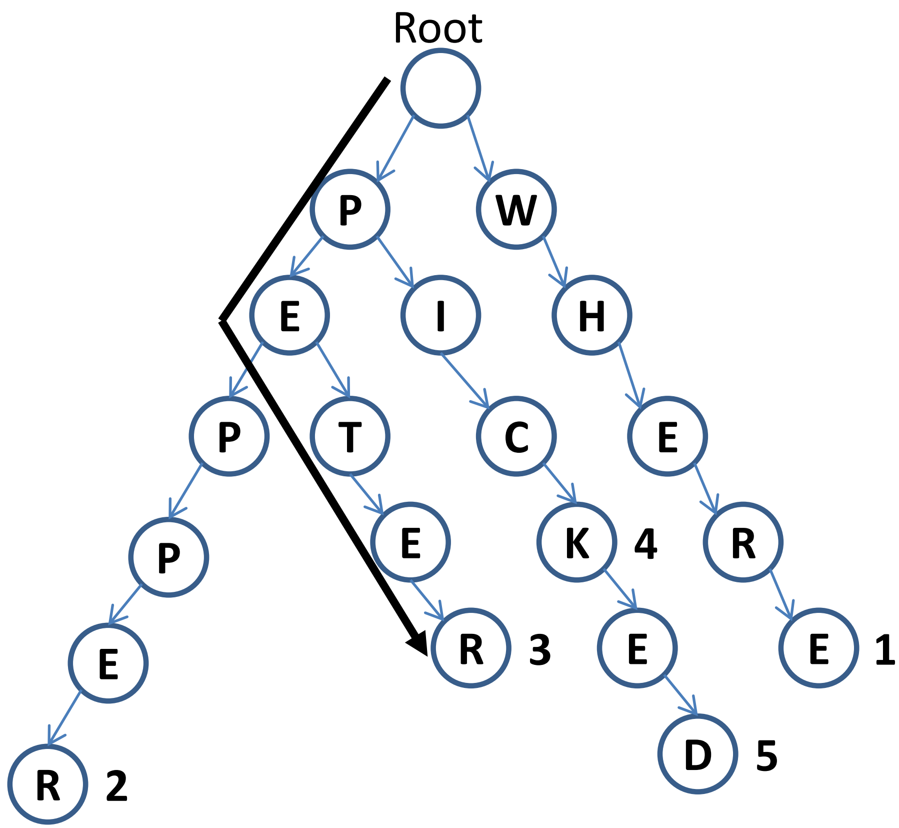
Retrieval Algorithm

Given a String key S to retrieve

1. Matching process
   i. Start from the root (level 0), check if it has a child node with character matching S[0], if it has move to child node and repeat the matching for S[1] and so on.
   ii. In general for a node at level M, the matching process check if the node has a child matching the character at index M of S
2. Terminating condition
   i. hit a node with no children node matching current character of the key ← return a miss
   ii. hit a node matching last character of key and there is a valid value for that node ← return the value (a hit)
   iii. hit a node matching last character of key and there is no valid value for that node ← return a miss

Retrieval Examples

- Search for
  - PETER
  - PICK
  - PEP
  - THE



Insertion Algorithm

- Perform a search for the key similar to retrieval operation until

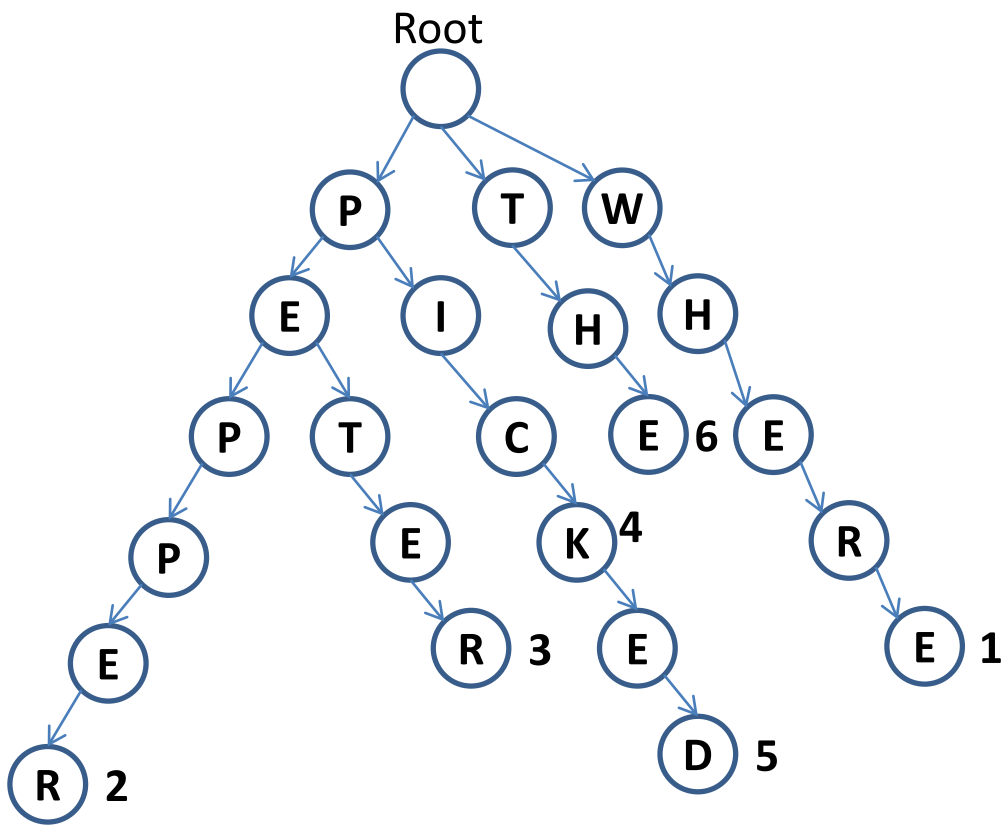
1. Hit a node matching the last character of the key and there is a valid value for the node (existing key/value pair)
   ← update the value with the new value.

2. Hit a node matching the last character of the key and there is no valid value for the node
   ← store the value in this node.

3. We hit a node with no children matching the current character of the string
   ←start inserting the remaining characters as descendent nodes in a linked list like fashion and put the value in the last node.

Insertion Examples

- Inserting
  - THE 10
  - PICKLED 7
  - PEPP 8



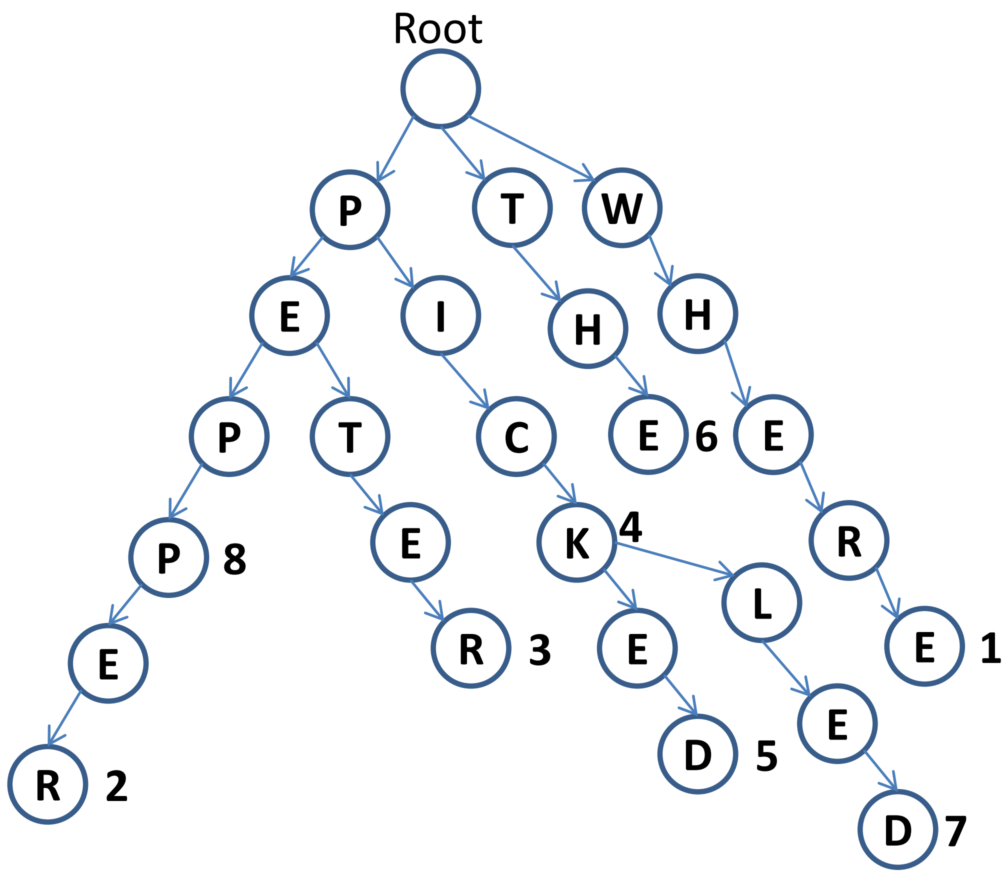
Deletion Algorithm

- Again perform a search for the key

1. Hit a node with no children matching current character of key ← do nothing
2. Hit a node matching the last character of the key and there is no valid value for the node ← do nothing
3. Hit a node matching the last character of the key and there is a valid value for the node ← remove the value
   i. If it has children nodes do nothing else (it is both a key and also the prefix for other existing keys)
   ii. If it has no children nodes then move back towards the root and start removing nodes. Stop when we hit a node that has at least 1 child (a prefix of some key(s)) or it has no children but it has a valid value (so it cannot be deleted)

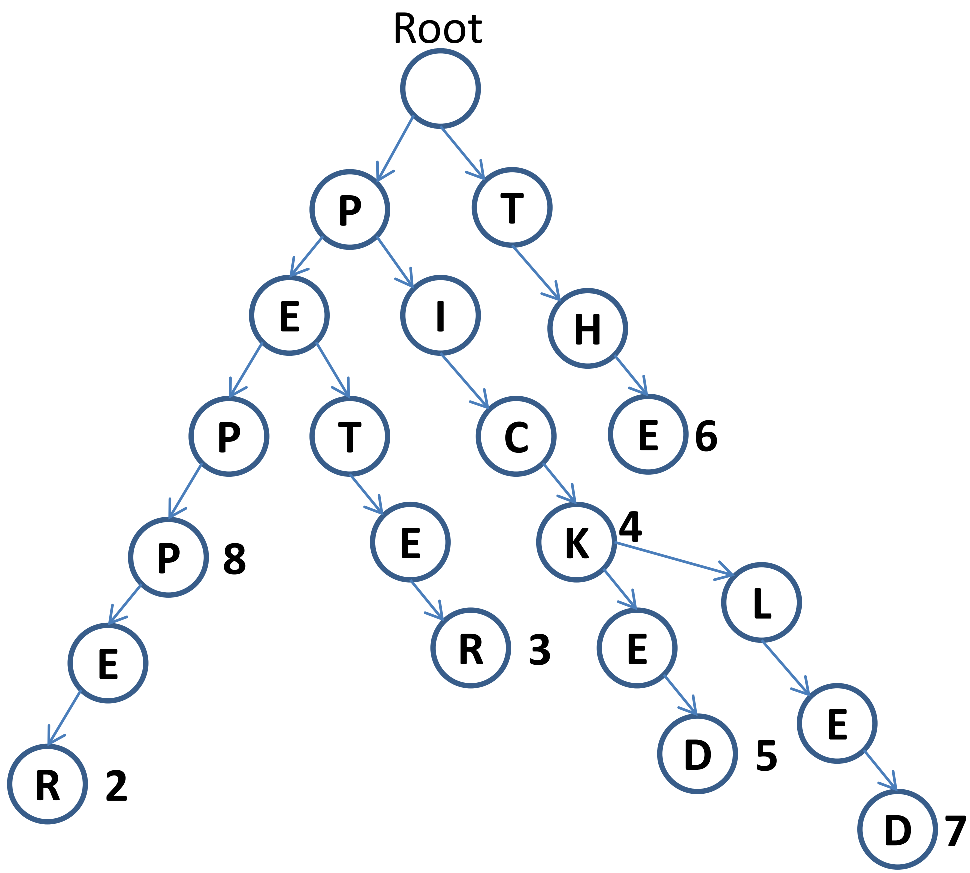
Deletion Examples (1)

- Deleting
  - WHERE
  - PICKLED
  - TO
  - PEPP



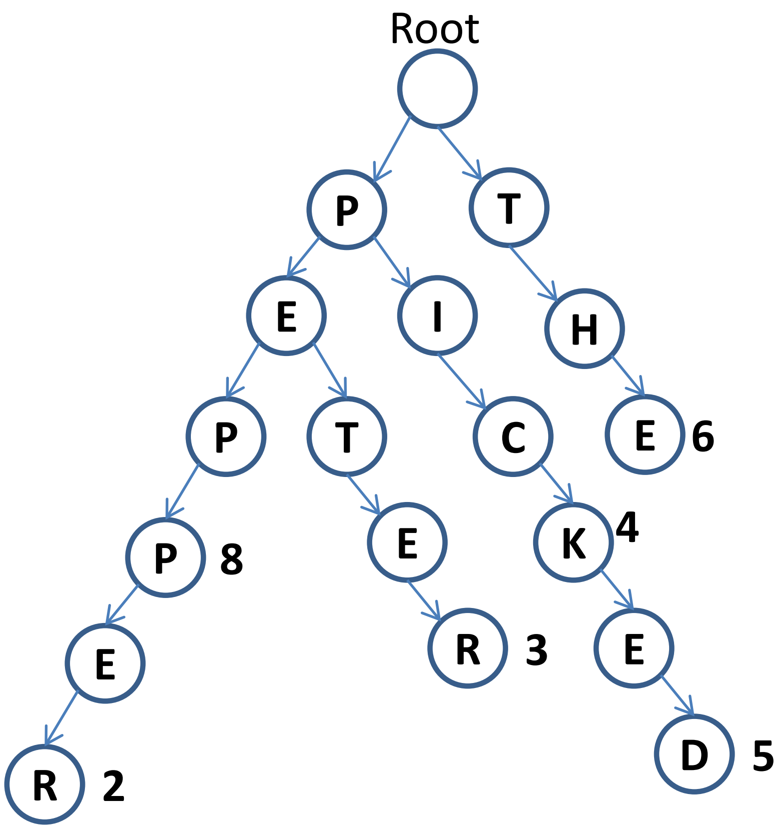
Deletion Examples (2)

- Deleting
  - WHERE
  - PICKLED
  - TO
  - PEPP



Deletion Examples (3)

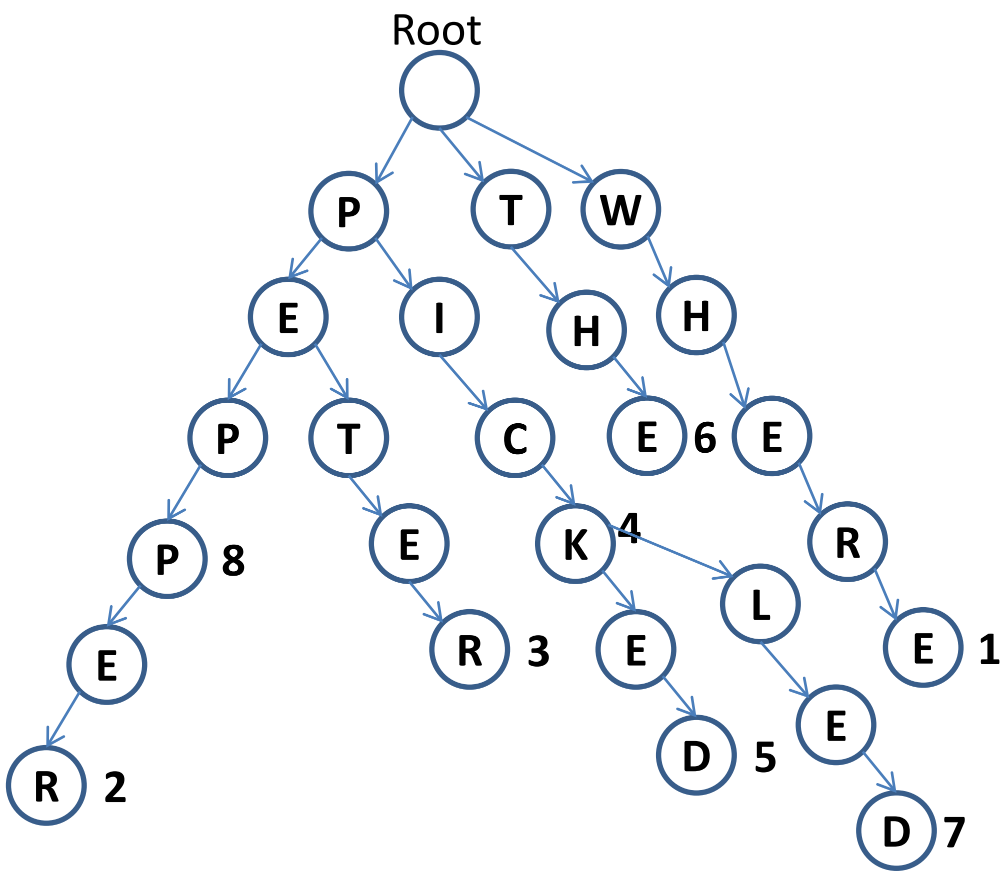
- Deleting
  - WHERE
  - PICKLED
  - TO
  - PEPP



Sorting Keys in Trie

- Simply perform an pre-order traversal of the Trie (visit children in lexicographical ordering of their characters)
- Keep track of the characters along the path from root to the current node
- If current node has a valid value then output the characters as a string key

Sorting Example

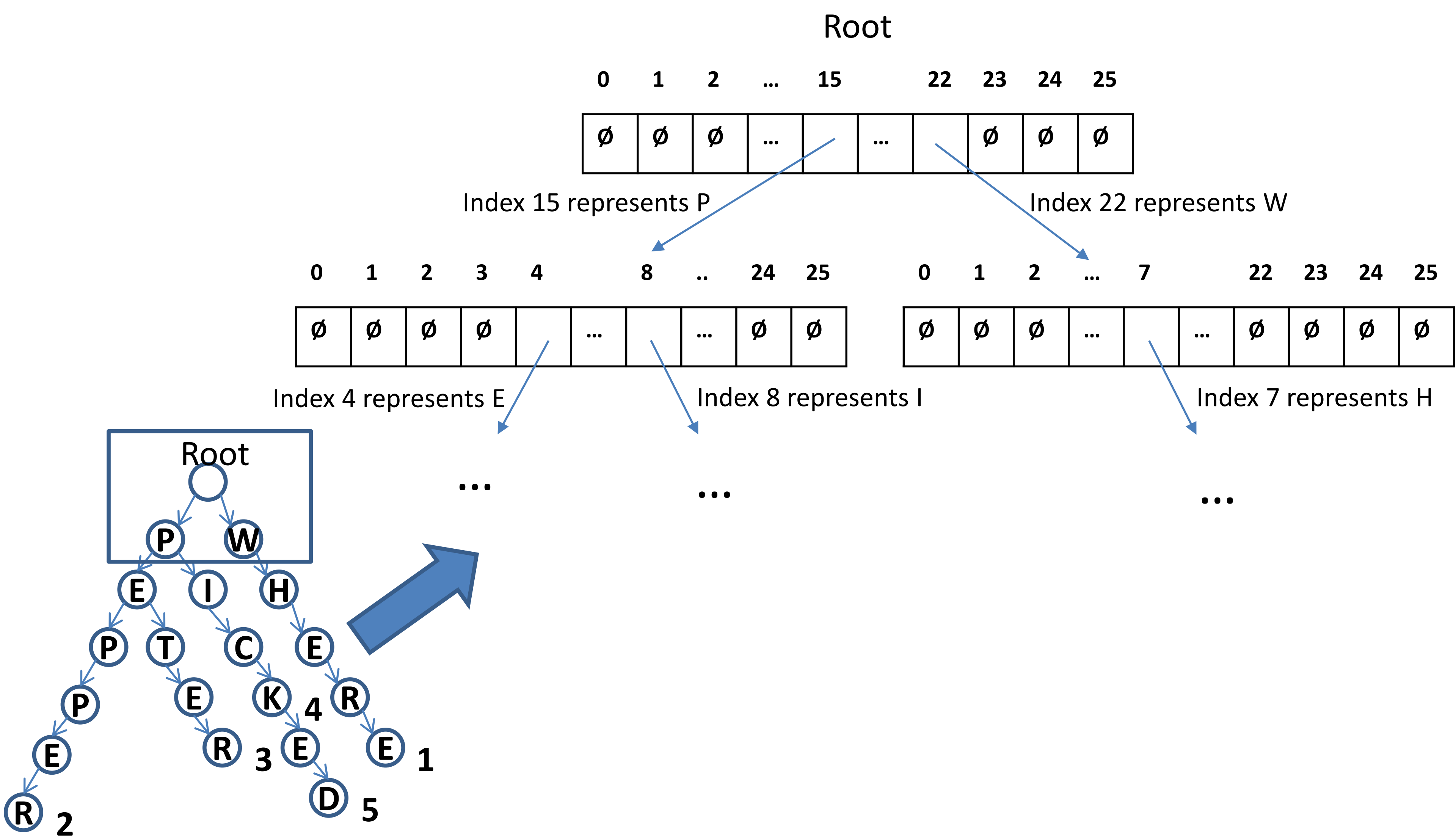


PEPP, PEPPER, PETER, PICK, PICKED, PICKLED, THE, WHERE

Representing Trie in code (1)

- Use nodes and references like BST/AVL
  - Unlike BST/AVL with at most 2 children per node, there is up to R children per node for a Trie for a language with R alphabets (e.g 26 for English)
  - Instead of using a variable sized array of references for the children, simply use an array of R references where a child node of a particular character is directly mapped to the array index (A == index 0, B == index 1 …)

Representing Trie in code (2)



Representing Trie in code (3)

- Additional attributes required in a node
  - **count** ← integer value to keep track of number of children in a node
  - **value** ← attribute to store the value of the key/value pair. Can use a reference type so if it is null means the string represented by the path from the root to this node is not a valid key. If not null means this is a valid key.

Time complexity analysis of Trie operations

- Worst case for Insertion, Retrieval (successful) and Deletion
  - $O(L)$ where L is the length of the key.

    if key is found, searching for it takes O(L) time but if its not found, its O(log N)

- Average time unsuccessful Retrieval (missing key) in a Trie of N keys
  - Assuming keys are randomly distributed over an alphabet of size R. Average time taken if key is not found is $O(log_R N)$ = $O(logN)$ ← independent of key length!

- Worst case for Sorting Keys
  - $O(L' * N)$ where L' is the length of the longest key and N is the number of keys in the Trie

Space complexity of Trie

- The space required for a Trie containing N keys is at least $O(NR)$ since each node in the Trie has an array of references of size R (alphabet size) and each key is at least length 1
- On average the number of nodes required is then $O(wNR)$ where w is the average key length
- This is more then the space required for hash tables which is $O(wN)$ ← total length of keys
- This is also more then the space required for AVL which is only $O(N)$+$O(wN)$ = $O(wN)$

N nodes for the N keys   Sum of total space required to store the N keys