CSCI 104
Tries

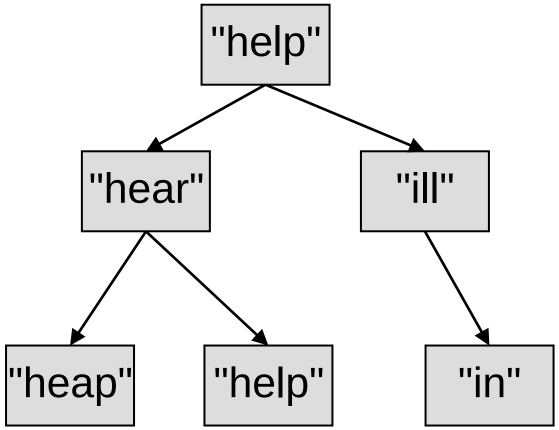
Mark Redekopp

David Kempe

TRIES

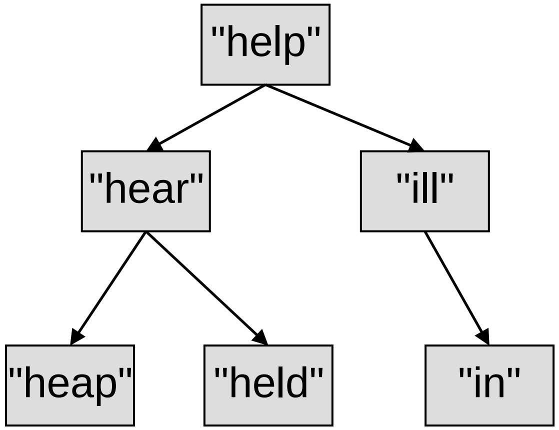
Review of Set/Map Again

- Recall the operations a set or map performs…
  - Insert(key)
  - Remove(key)
  - find(key) : bool/iterator/pointer
  - Get(key) : value ***[Map only]***
- We can implement a set or map using a binary search tree
  - Search = O(__________)
- But what work do we have to do at each node?
  - Compare (i.e. string compare)
  - How much does that cost?
    - Int = O(1)
    - String = O(k) where k is length of the string
  - Thus, search costs O(_____________)



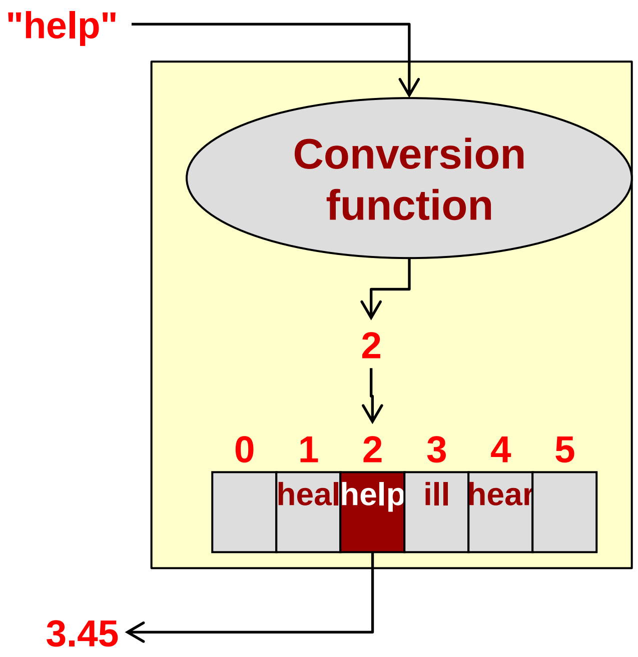
Review of Set/Map Again

- Recall the operations a set or map performs…
  - Insert(key)
  - Remove(key)
  - find(key) : bool/iterator/pointer
  - Get(key) : value ***[Map only]***
- We can implement a set or map using a binary search tree
  - Search = O(log(n))
- But what work do we have to do at each node?
  - Compare (i.e. string compare)
  - How much does that cost?
    - Int = O(1)
    - String = O(k) where k is length of the string
  - Thus, search costs O(k * log(n))



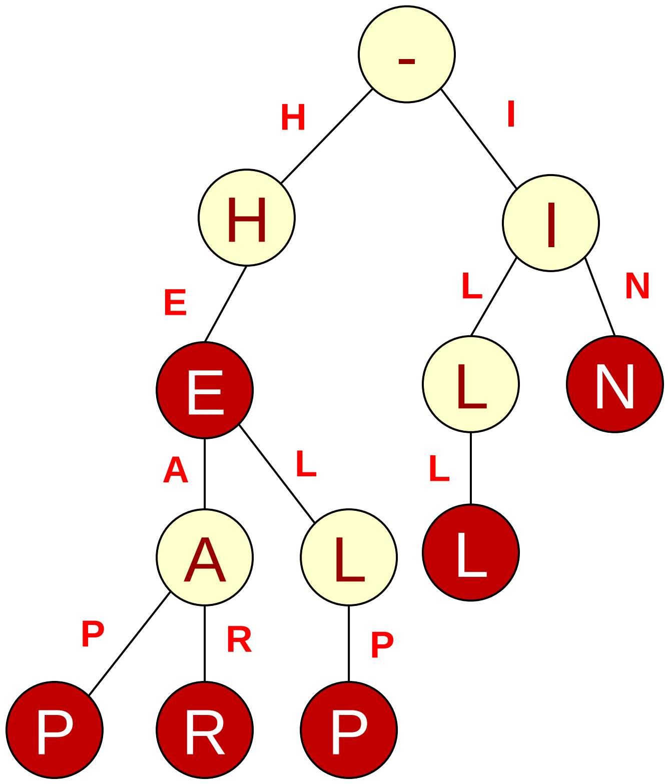
Review of Set/Map Again

- We can implement a set or map using a hash table
  - Search = O(1)
- But what work do we have to do once we hash?
  - Compare (i.e. string compare)
  - How much does that cost?
    - Int = O(1)
    - String = O(k) where k is length of the string
  - Thus, search costs O(k)



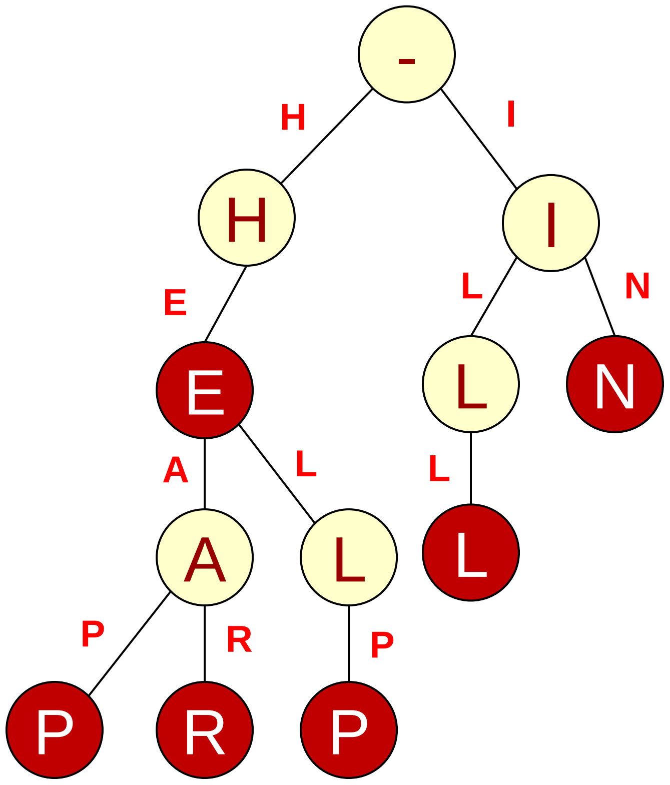
Tries

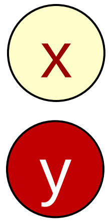
- Assuming unique keys, can we still achieve O(k) search but not have collisions?
  - O(k) means the time to compare is *independent* of how many keys (i.e. n) are being stored and only depends on the length of the key
- Trie(s) (often pronounced "try" or "tries") allow O(k) (i.e. constant time) retrieval
  - Sometimes referred to as a radix tree or **prefix tree**
- Consider a trie (prefix tree) for the keys
  - "HE", "HEAP", "HEAR", "HELP", "ILL", "IN"



Tries

- Rather than each node storing a full key value, each node represents a prefix of the key
- Highlighted nodes indicate terminal locations
  - For a map we could store the associated value of the key at that terminal location
- A key is represented by a path through the tree
  - Notice we "share" paths for keys that have a common prefix
- To search for a key, start at the root consuming one unit (bit, char, etc.) of the key at a time
  - If you end at a terminal node, SUCCESS
  - If you end at a non-terminal node, FAILURE



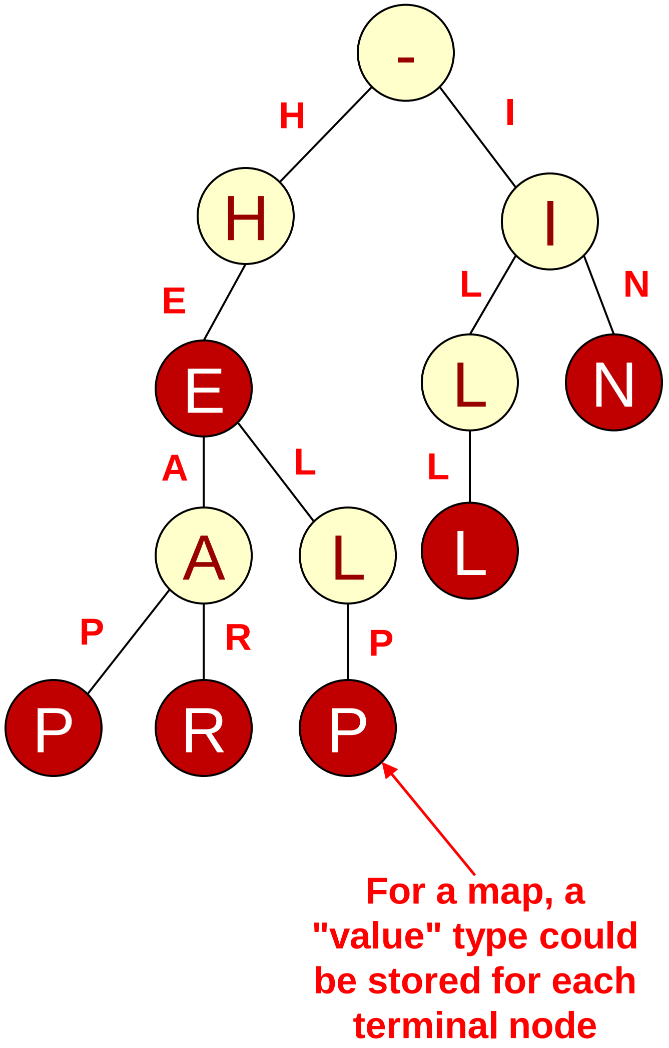


Internal node (prefix is not a key)

Terminal node (prefix is an inserted key)

Tries

- To search for a key, start at the root consuming one unit (bit, char, etc.) of the key at a time
  - If you end at a terminal node, SUCCESS
  - If you end at a non-terminal node, FAILURE
- Examples:
  - Search for "He"
  - Search for "Help"
  - Search for "Head"
- Search takes O(k) where k = length of key
  - Notice this is the same as a hash table

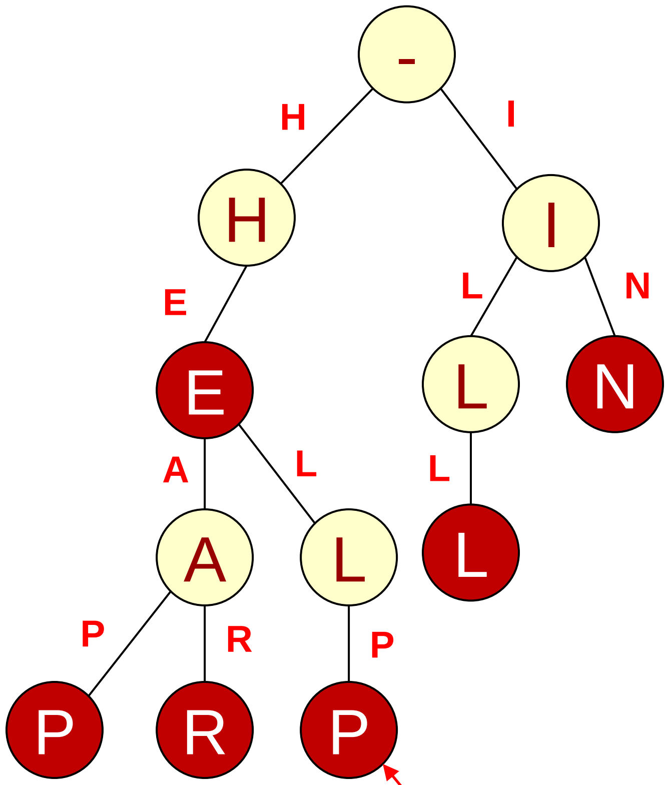

-
H
I
H
I
E
L
N
E
L
N
A
L
L
A
L
L
P
R
P
P
R
P
For a map, a "value" type could be stored for each terminal node

Your Turn

- Construct a trie to store the set of words
  - Ten
  - Tent
  - Then
  - Tense
  - Tens
  - Tenth

Thinking Exercise: Removal

- How would removal of a key work in a trie and what are the cases you'd have to worry about?
  - Does removal of a key always mean removal of a node?
  - If we do remove a node, would it only be one node in the trie?

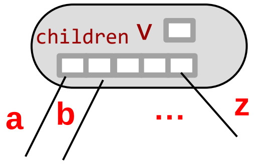


A "value" type could be stored for each non-terminal node

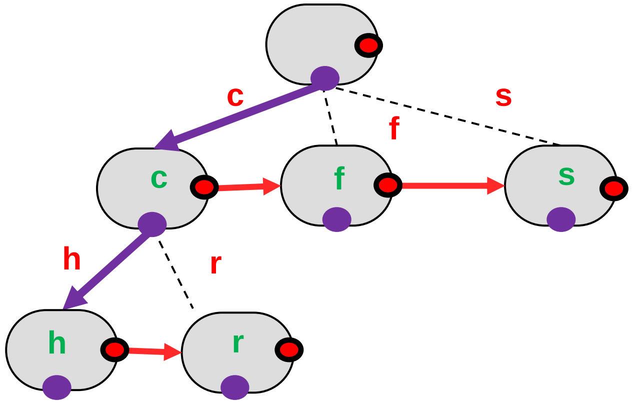
Structure of Trie Nodes

- What do we need to store in each node?
- Depends on how "dense" or "sparse" the tree is?
- Dense (most characters used) or small size of alphabet of possible key characters
  - Array of child pointers
  - One for each possible character in the alphabet
- Sparse
  - (Linked) List of children
  - Node needs to store ______

```
template < class V >
struct TrieNode{
  V* value; // NULL if non-terminal
  TrieNode<V>* children[26];
};
```



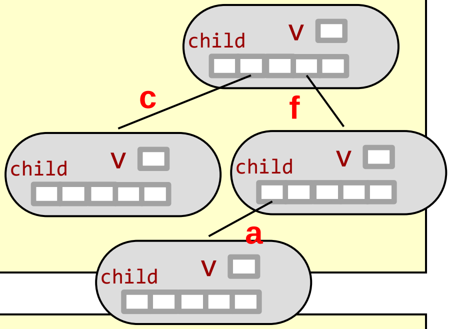
```
template < class V >
struct TrieNode{
  char key;
  V* value;
  TrieNode<V>* next; // sibling
  TrieNode<V>* children; // head ptr
};
```



Search

- Search consumes one character at a time until
  - The end of the search key
    - If value pointer exists, then the key is present in the map
  - Or no child pointer exists in the TrieNode
- Insert
  - Search until key is consumed but trie path already exists
    - Set v pointer to value
  - Search until trie path is NULL, extend path adding new TrieNodes and then add value at terminal

```
V* search(string key)
{
  TrieNode<V>* node = root;
  size_t i=0;
  while(i != key.size() && node != NULL){
    node = node->children[key[i] - 'a'];
    i++;
  }
  if(node) return node->v;
  else     return NULL;
}
```



```
void insert(string key, const V& v)  {
  TrieNode<V>* node = root;
  size_t i=0;
  while(i != key.size() && node != NULL){
    node = node->children[key[i] - 'a'];  i++;
  }
  if(node){
    node->v = new V(v);
  }
  else {
    // create new nodes in trie to extend path
    // updating root if trie is empty
  }
}
```

Application: IP Lookups

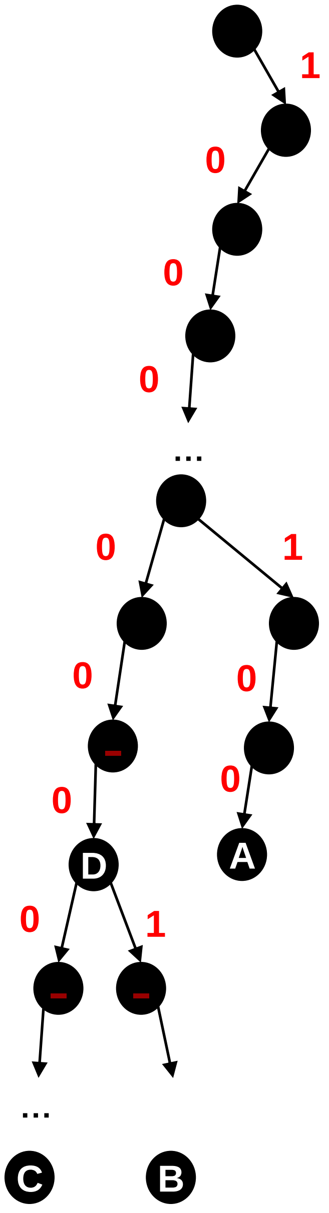
- Network routers form the backbone of the Internet
- Incoming packets contain a destination IP address (128.125.73.60)
- Routers contain a "routing table" mapping some prefix of destination IP address to output port
  - 128.125.x.x => Output port C
  - 128.209.32.x => Output port B
  - 128.x.x.x => Output port D
  - 132.x.x.x => Output port A
- Keys = Match the longest prefix
  - Keys are unique
- Value = Output port

Octet 1	Octet 2	Octet 3	Port
10000000	01111101		C
10000000	11010001	00100000	B
10000000			D
10000100			A

IP Lookup Trie

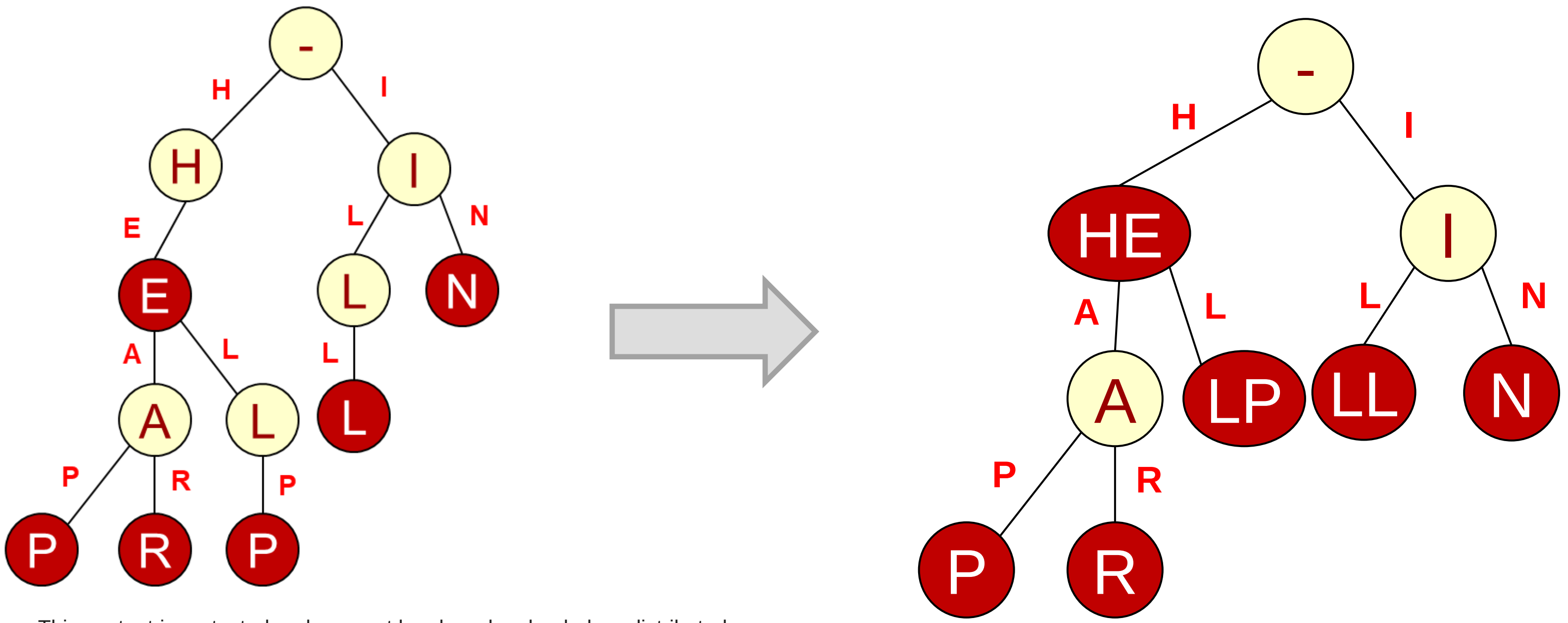
- A binary trie implies that the
  - Left child is for bit '0'
  - Right child is for bit '1'
- Routing Table:
  - 128.125.x.x => Output port C
  - 128.209.32.x => Output port B
  - 128.209.44.x => Output port D
  - 132.x.x.x => Output port A

Octet 1	Octet 2	Octet 3	Port
10000000	01111101		C
10000000	11010001	00100000	B
10000000			D
10000100			A



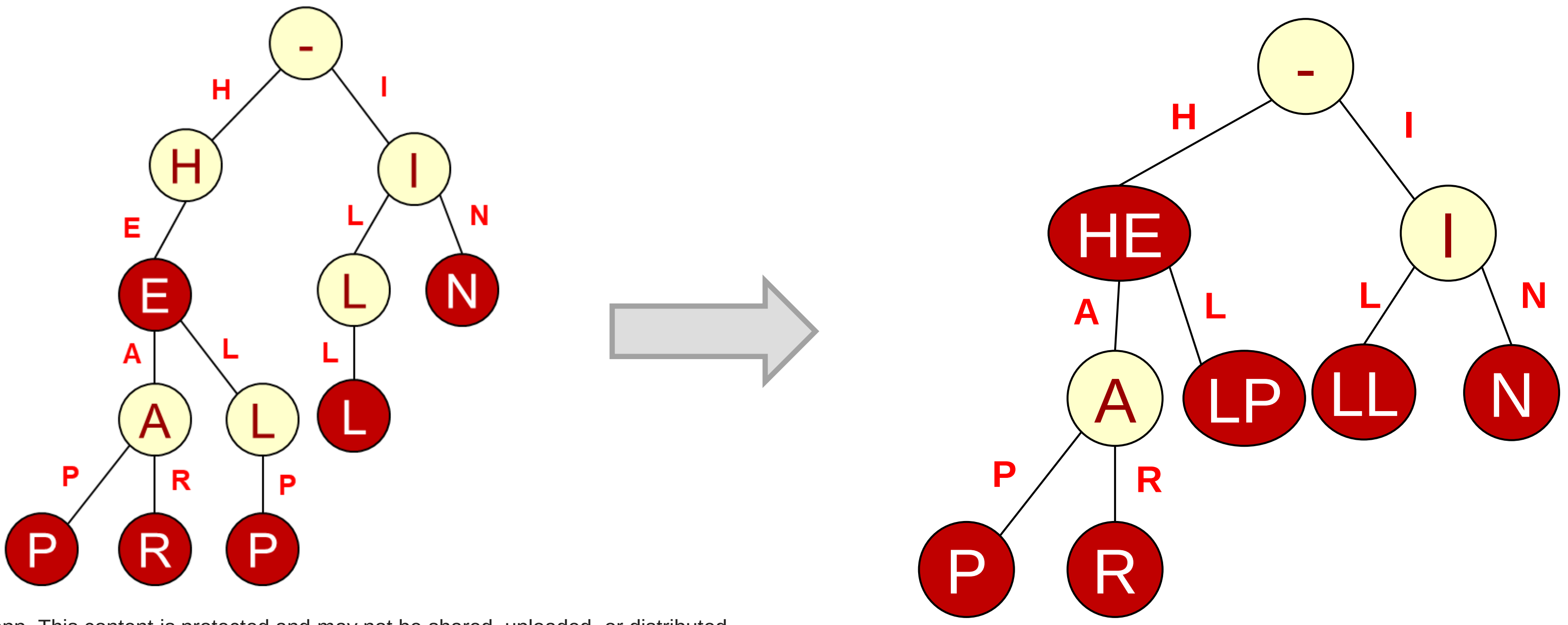
Compressed Prefix Tree

- We can reduce the number of nodes and thus storage, by storing substrings in each node
  - If a node has only one child, combine with child storing concatenation of characters
  - https://www.cs.usfca.edu/~galles/visualization/RadixTree.html



Compressed Prefix Tree

- Walk key string based on the length of the substring in the current node and then use the next key string character to choose the child node
- Key is not present if key string characters are exhausted before substring in node or no corresponding child entry
- Examples: 'H', 'HERD'



Practice

- Construct a compressed trie to store the set of words
  - Ten
  - Tent
  - Then
  - Tense
  - Tens
  - Tenth

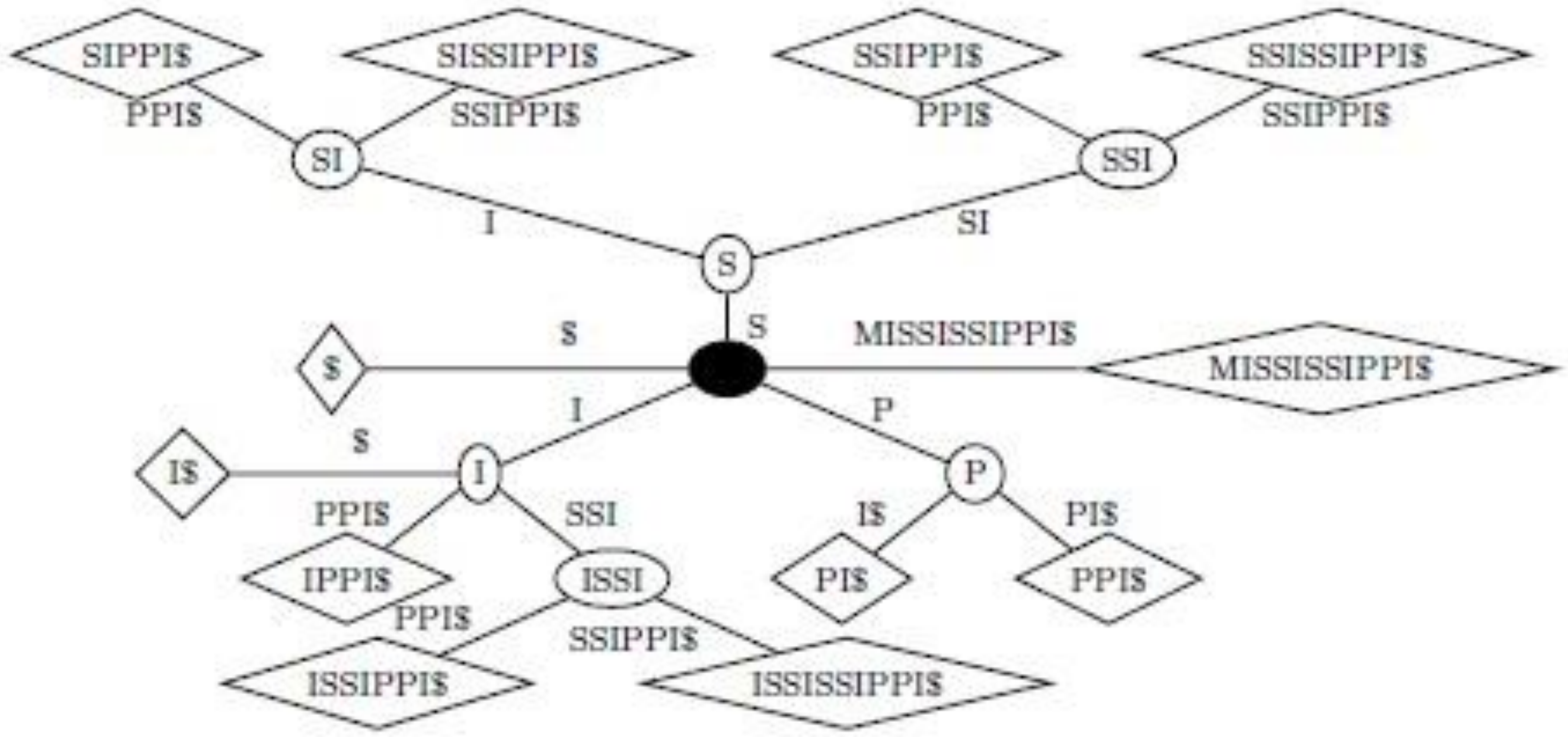
Prefix Trees (Tries) Review

- What problem does a prefix tree solve
  - Lookups of keys (and possible associated values)
- A prefix tree helps us match 1-of-n keys
  - "He"
  - "Help"
  - "Hear"
  - "Heap"
  - "In"
  - "Ill"
- Here is a slightly different problem:
  - Given a large text string, T, can we find certain substrings or answer other queries about patterns in T
  - A suffix tree (trie) can help here

SUFFIX TREES

Suffix Trees

A **suffix tree** of a string W is a compressed trie consisting of all possible suffixes of W.



Are \`issip\` or \`sipi\` substrings?

Suffix Trees

- When W has n characters (indexed 0 to n), the suffix tree has:
  - n leaves, each one representing a single suffix
    $$W[i:(n-1)], 0 \le i \le (n-1)$$
  - Every non-leaf node has at least two children
  - Each edge is labelled with a substring of W
  - If e and e’ are edges out of the same node, then their labels start with different letters.
  - For any root-leaf path, the concatenation of their edge labels is equal to $W[i:(n-1)]$
  - < n internal nodes
  - $O(n)$ total nodes
- There is an algorithm (Ukkonen’s Algorithm) which can build a suffix tree in linear time.

What Have We Learned

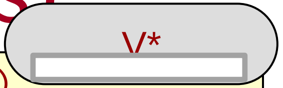
- **Key Point**: Think about all the data structures we've been learning
  - There is almost always a trade-off of memory vs. speed (Space vs. time)
  - Most data structures just exploit different points on that time-space tradeoff continuum
  - Often we build a data structure that replicates data and takes a lot of memory space…
  - …so that we can find data faster

Suffix Trie Slides

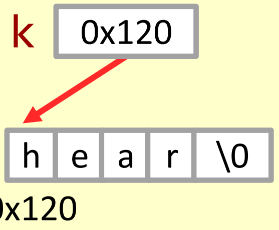
- http://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/suffixtrees.pdf

Search (Using C-Strings / Char arrays)

- Search consumes one character at a time until
  - The end of the search key
    - If value pointer exists, then the key is present in the map
  - Or no child pointer exists in the TrieNode
- Insert
  - Search until key is consumed but trie path already exists
    - Set v pointer to value
  - Search until trie path is NULL, extend path adding new TrieNodes and then add value at terminal



```
V* search(char* k, TrieNode<V>* node)
{
  while(*k != '\0' && node != NULL){
    node = node->children[*k – 'a'];
    k++;
  }
  if(node) return node->v;
  else     return NULL;
}
```



```
void insert(char* k, const Value& v)
{
  TrieNode<V>* node = root;
  while(*k != '\0' && node != NULL){
    node = node->children[*k – 'a'];  k++;
  }
  if(node){
    node->v = new Value(v);
  }
  else {
    // create new nodes in trie
    // to extend path
    // updating root if trie is empty
  }
}
```