CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

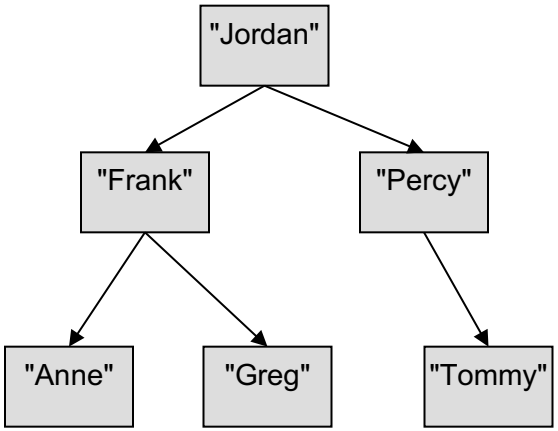
Slides adapted from: Mark Redekopp and David Kempe

An imperfect set…

BLOOM FILTERS

Set Review

- Recall the operations a set performs…
  - Insert(key)
  - Remove(key)
  - Contains(key) : bool (a.k.a. find())
- We can think of a set as just a map without values…just keys
- We can implement a set using
  - List
    - O(n) for some of the three operations
  - (Balanced) Binary Search Tree
    - O(log n) insert/remove/contains
  - Hash table
    - O(1) insert/remove/contains



Bloom Filter Idea

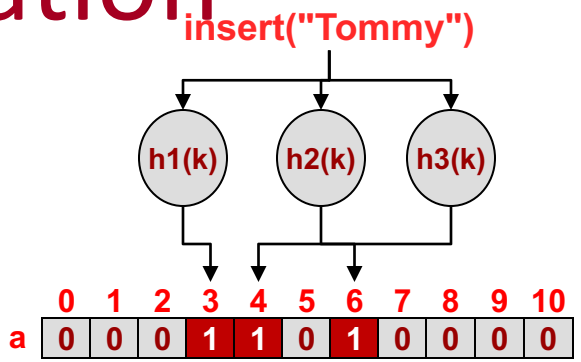
- Suppose you are looking to buy the next hot consumer device. You can only get it in stores (not online). Several stores who carry the device are sold out. Would you just start driving from store to store?
- You'd probably call ahead and see if they have any left.
- If the answer is "NO"…
  - There is no point in going…it's not like one will magically appear at the store
  - You save time
- If the answer is "YES"
  - It's worth going…
  - Will they definitely have it when you get there?
  - Not necessarily…they may sell out while you are on your way
- But overall this system would at least help you avoid wasting time

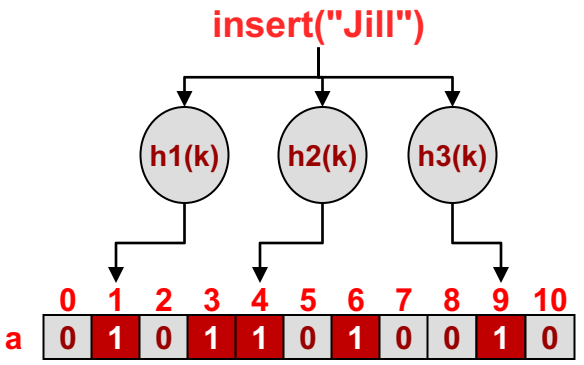
Bloom Filter Idea

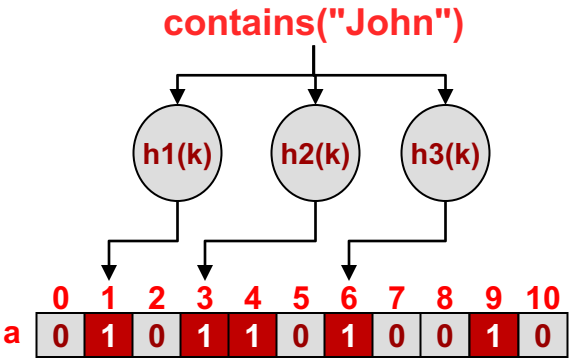
- A Bloom filter is a set such that "contains()" will *quickly* answer…
  - "No" correctly (i.e. if the key is not present)
  - "Yes" with a chance of being incorrect (i.e. the key may not be present but it might still say "yes")
- Why would we want this?
  - A Bloom filter usually sits in front of an actual set/map
  - Suppose that set/map is EXPENSIVE to access
    - Maybe there is so much data that the set/map doesn't fit in memory and sits on a disk drive or another server as is common with most database systems
      - Disk/Network access = ~milliseconds
      - Memory access = ~nanoseconds
  - The Bloom filter holds a "duplicate" of the keys but uses FAR less memory and thus is cheap to access (because it can fit in memory)
  - We ask the Bloom filter if the set contains the key
    - If it answers "No" we don't have to spend time search the EXPENSIVE set
    - If it answers "Yes" we can go search the EXPENSIVE set

Bloom Filter Explanation

- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \ldots h_s(k)\}$
- Insert()
  - Apply each $h_i(k)$ to the key
  - Set $a[h_i(k)]$ = True
- Contains()
  - Apply each $h_i(k)$ to the key
  - Return True if ***all*** $a[h_i(k)]$ = True
  - Return False otherwise
  - In other words, answer is "Maybe" or "No"
    - May produce "false positives"
    - May NOT produce "false negatives"
- We will ignore removal for now

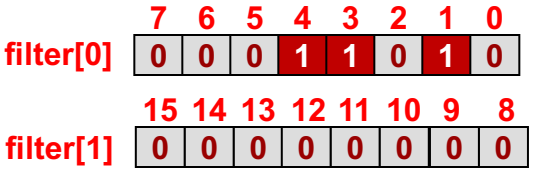






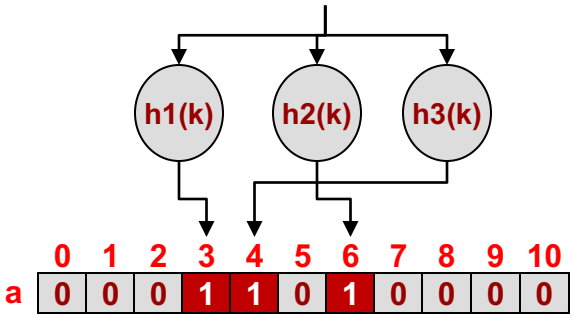
Implementation Details

- Bloom filter's require only a bit per location, but modern computers read/write a full byte (8-bits) at a time or an int (32-bits) at a time
- To not waste space and use only a bit per entry we'll need to use bitwise operators
- For a Bloom filter with N-bits declare an array of N/8 unsigned char's (or N/32 unsigned ints)
  - unsigned char filter8[ceil(N/8)];
- To set the k-th entry,
  - filter[k/8] |= (1 << (k%8));
- To check the k-th entry
  - if (filter[k / 8] & (1 << (k%8)))



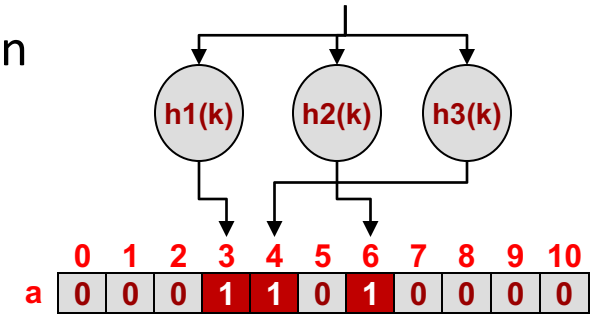
Probability of False Positives

- What is the probability of a false positive?
- Let's work our way up to the solution
  - Probability that one hash function selects or does not select a location x assuming "good" hash functions
    - $P(h_i(k) = x) = 1/m$
    - $P(h_i(k) \neq x) = [1 - 1/m]$
  - Probability that all j hash functions don't select a location
    - $[1 - 1/m]^{j}$
  - Probability that all s-entries in the table have not selected location x
    - $[1 - 1/m]^{sj}$
  - Probability that a location x HAS been chosen by the previous n entries
    - $1 - [1 - 1/m]^{nj}$
  - Math factoid: For small y, $e^{y} = 1+y$ (substitute y = -1/m)
    - $1 - e^{-nj/m}$
  - Probability that all of the j hash functions find a location True once the table has n entries
    - $(1 - e^{-nj/m})^{j}$



Probability of False Positives

- Probability that all of the j hash functions find a location True once the table has s entries
  - $(1 - e^{-nj/m})^j$
- Define $\alpha = n/m$ = loading factor
  - $(1 - e^{-\alpha j})^j$
- First "tangent": Is there an optimal number of hash functions (i.e. value of j)
  - Use your calculus to take derivative and set to 0
  - Optimal # of hash functions, $j = \ln(2) / \alpha$
- Substitute that value of j back into our probability above
  - $(1 - e^{-\alpha \ln(2)/\alpha})^{\ln(2)/\alpha} = (1 - e^{-\ln(2)})^{\ln(2)/\alpha} = (1 - 1/2)^{\ln(2)/\alpha} = 2^{-\ln(2)/\alpha}$
- Final result for the probability that all of the j hash functions find a location True once the table has s entries: $\mathbf{2^{-\ln(2)/\alpha}}$
  - Recall $0 \leq \alpha \leq 1$



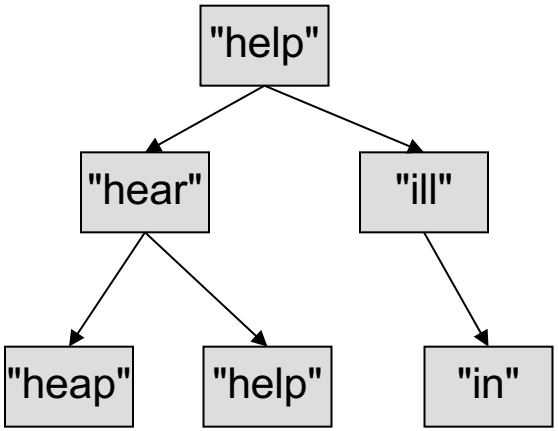
Sizing Analysis

- Can also use this analysis to answer or a more "useful" question…
- …To achieve a desired probability of false positive, what should the table size be to accommodate n entries?
  - Example: I want a probability of p=1/1000 for false positives when I store n=100 elements
  - Solve $2^{-m*\ln(2)/n} < p$
    - Flip to $2^{m*\ln(2)/n} \geq 1/p$
    - Take log of both sides and solve for m
    - $m \geq [n*\ln(1/p)] / \ln(2)^2 \approx 2n*\ln(1/p)$ because $\ln(2)^2 = 0.48 \approx ½$
  - So for p=.001 we would need a table of m=14*n since ln(1000) ≈ 7
    - For 100 entries, we'd need 1400 bits in our Bloom filter
  - For p = .01 (1% false positives) need m=9.2*n (9.2 bits per key)
  - Recall: Optimal # of hash functions, j = ln(2) / α
    - So for p=.01 and α = 1/(9.2) would yield j ≈ 7 hash functions

TRIES

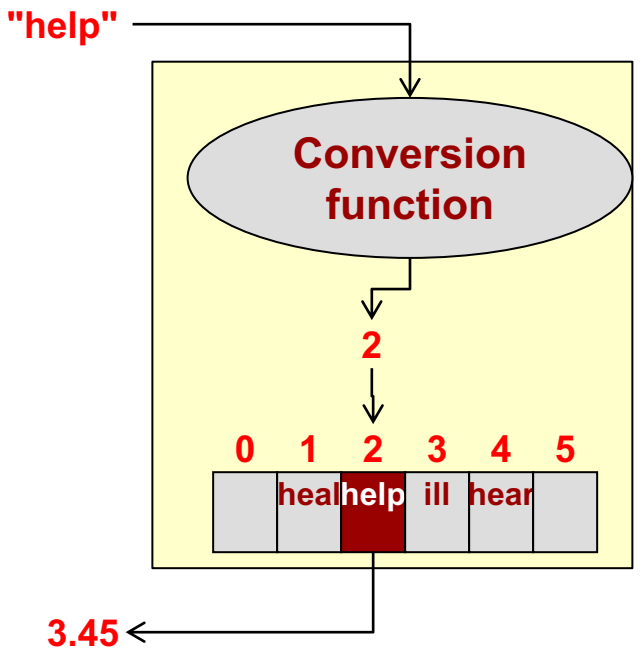
Review of Set/Map Again

- 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(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(m) where m is length of the string
  - Thus, search costs O(m * log(n))



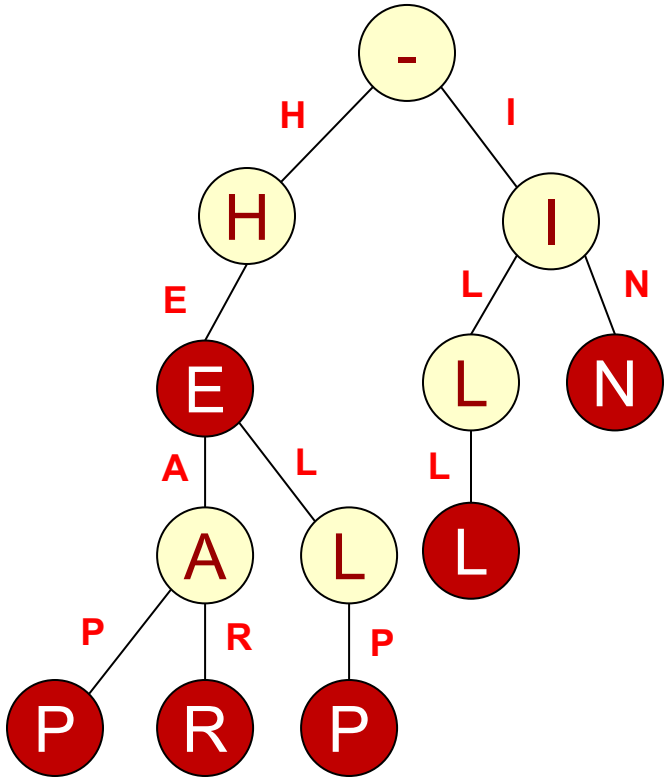
Review of Set/Map Again

- 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(m) where m is length of the string
  - Thus, search costs O(m)



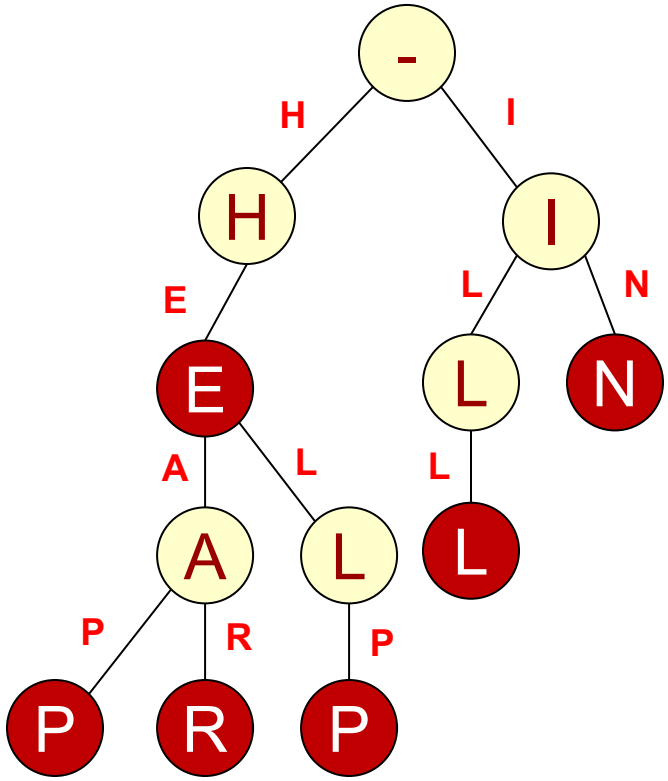
Tries

- Assuming unique keys, can we still achieve O(m) search but not have collisions?
  - O(m) 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(m) retrieval
  - Sometimes referred to as a radix tree or prefix tree
- Consider a trie for the keys
  - "HE", "HEAP", "HEAR", "HELP", "ILL", "IN"



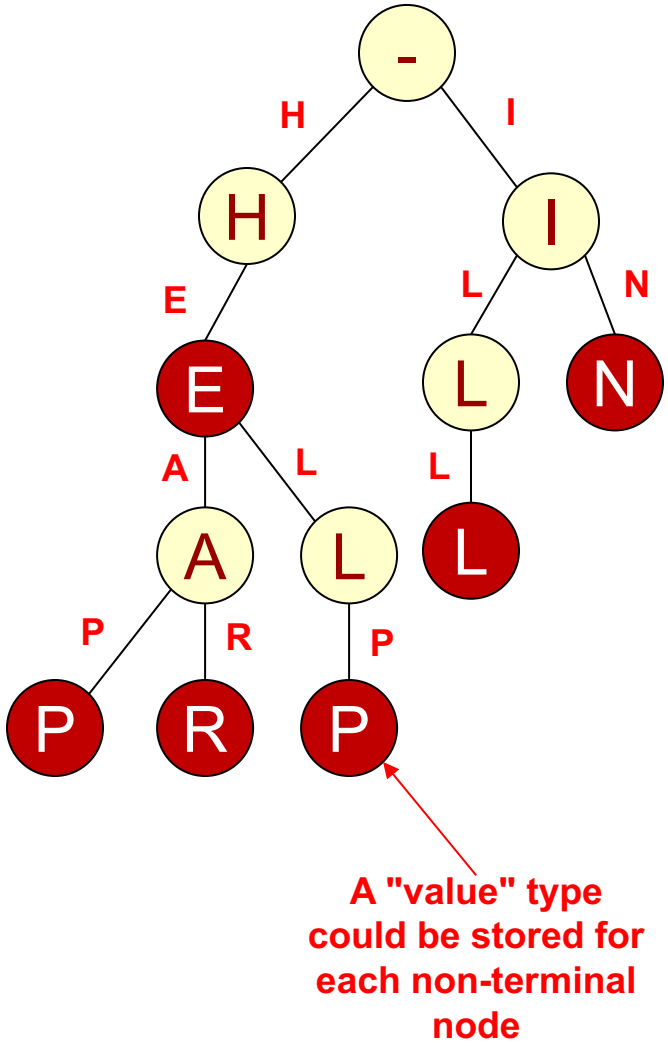
Tries

- 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
- 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



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(m) where m = length of key
  - Notice this is the same as a hash table

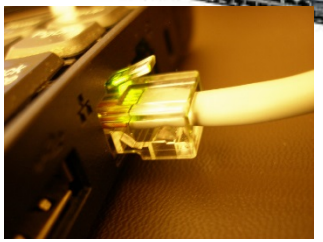


Your Turn

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

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

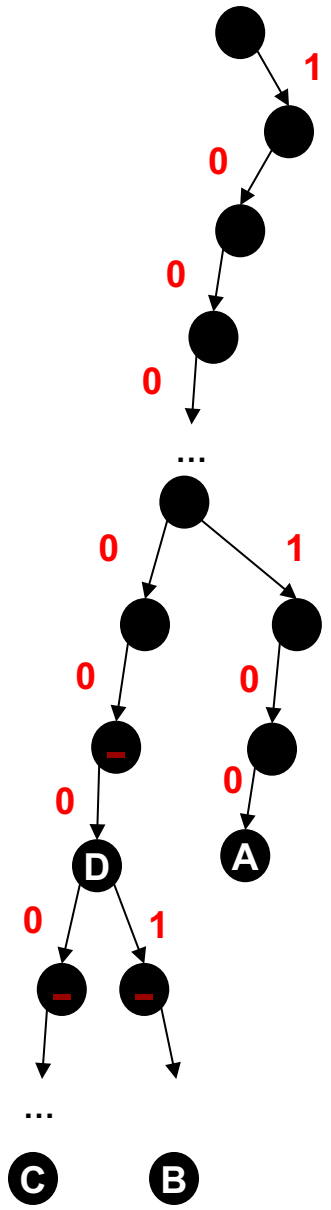


Octet 1	Octet 2	Octet 3	Port
10000000	01111101		C
10000000	11010001	00100000	B
10000000			D
10000100			A

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.x.x.x => Output port D
  - 132.x.x.x => Output port A

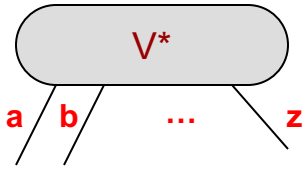
Octet 1	Octet 2	Octet 3	Port
10000000	01111101		C
10000000	11010001	00100000	B
10000000			D
10000100			A



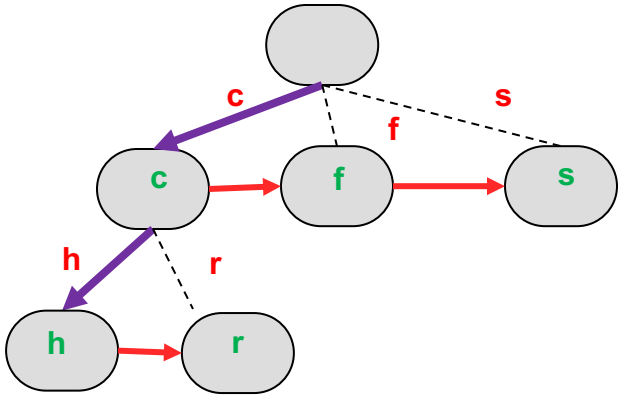
Structure of Trie Nodes

- 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;
  TrieNode<V>* children;
};
```



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

```
void insert(char* k, 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
  }
}
```

SUFFIX TREES (TRIES)

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 Trie Slides

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

Suffix Trie Wrap-Up

- How many nodes can a suffix trie have for text, T, with length |T|?
  - $|T|^2$
  - Can we do better?
- Can compress paths without branches into a single node
- Do we need a suffix trie to find substrings or answer certain queries?
  - We could just take a string and search it for a certain query, q
  - But it would be slow => O(|T|) and not O(|q|)

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
    - i.e. Space vs. time
  - Most data structures just exploit different points on that time-space tradeoff continuum
  - Think about searches in your project…Do we need a map?
  - No, we could just search all items each time a keyword is provided
    - But think how slow that would be
  - So we build a data structure (i.e. a map) that replicates data and takes a lot of memory space…
  - …so that we can find data faster