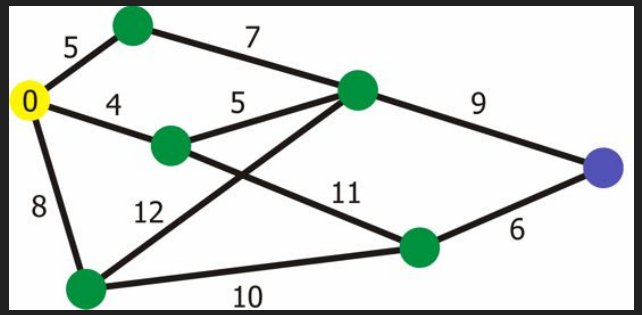
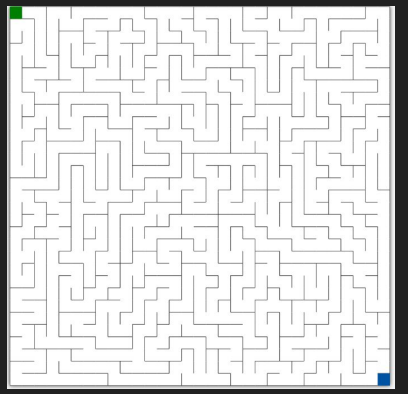
Please do not redistribute these slides without prior written permission





CS 5008/5009

Data Structures, Algorithms, and Their Applications Within Computer Systems

Dr. Mike Shah

Pre-Class Warmup

- Take a look at the study guide for Algorithms!
- We have learned quite a bit!

Module 14 - Final Review and Course Wrap up | Video • Lecture outline ◦ TRACE for CS 5008 ◦ Make sure you have a gameplan of when to take the exam befor April 26, 11:59 pm EST ◦ Review ◦ Bonus Topics (Not on the exam) ◦ Wrap-up and open discussion ◦ Begin Module!	Online Exam opens April 20 at 12:00am EST on Canvas. (Study Guide). Exam must be completed by **April 27 at 11:59 pm EST**.	Last day of class :(

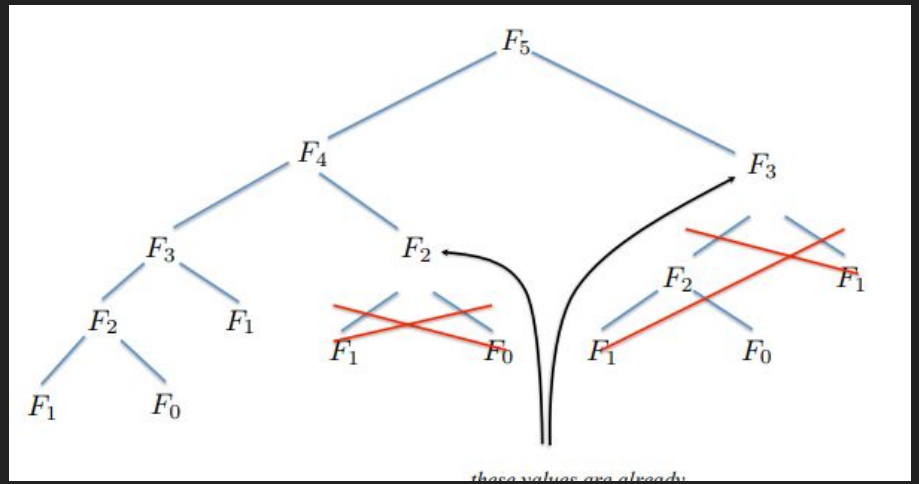
Note to self: Start audio recording of lecture :)
(Someone remind me if I forget!)

Course Logistics

- HW11 part 1 and part 2 due
  - (Just submit assignment in part 1 repository)
- Will start class with trace evaluation (10-15 minutes)
- ~~Then class picture~~
- No lab today
- Last in-class activity
- Make sure you find a time to take the exam!

Last Time

- Greedy Algorithms
  - Dijkstra's Shortest Path
- Dynamic Programming
  - Example with Fibonacci sequence
  - Can use memoization to avoid recomputing values.



Today

- Trace Evaluation
  - I'll leave the room for about 15 minutes for you to fill out the evaluation (someone come get me)
- Homework Discussion
- ~~Class Picture~~
  - ~~You do not have to take part--but it is a tradition (picture will go on course website so I remember you!)~~
- Exam Review
- Mini-Topics
  - Trie
  - String Matching

TRACE Evaluations

TRACE

- Please take a few moments to complete your TRACE evaluations of this course
- They are very helpful to me to improve as an instructor, improve this course, and improve other courses that I may be your instructor in the future!

hw structure discussion

live drawing/coding/discussion

- ~~lab 12 memoization~~
- print_path
- has_cycle
  - tip: ‘add visited attribute’
  - General algorithm
    - Use DFS (Can also use BFS)
- Additional help for part 2
  - Last weeks Vidoje’s lecture (for is_reachable) link
- ike online better or on campus better

Previously

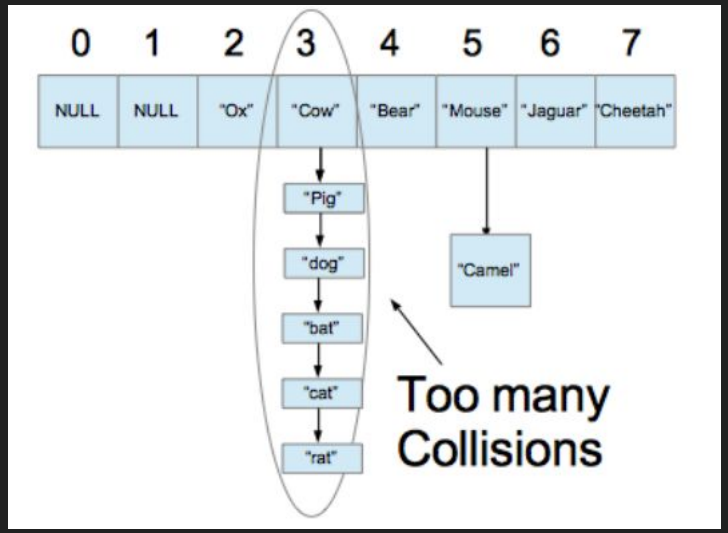
Key-Value Pairs

Key-Value Data Structures

- We have looked at a few data structures that have key-value pairs
  - Arrays
    - Key: Unsigned integer to an index
    - Value: Whatever data type is being stored
  - Hash map
    - Key: Gets hashed to a positive integer value--the more unique the better
    - Value: Whatever data type is being stored
- Both of these data structures could give us constant time lookup
  - arrays in the worse-case give O(1)
  - hashmaps in the average-case give Θ(1)
    - (given some reasonable hash function)

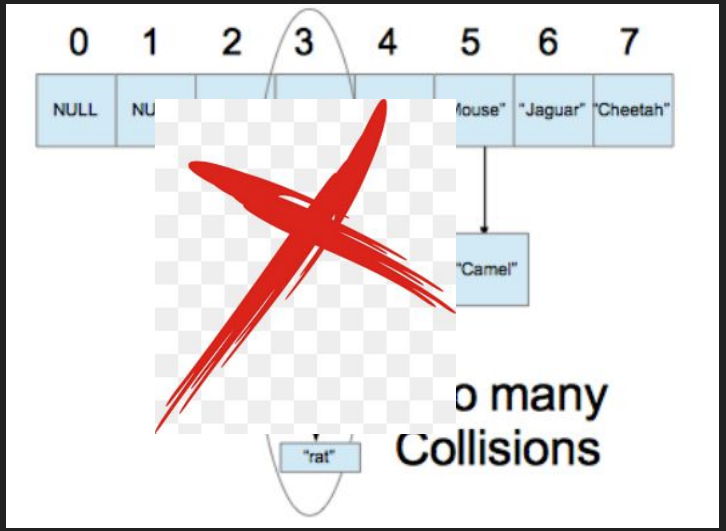
Key-Value Data Structures - Problems

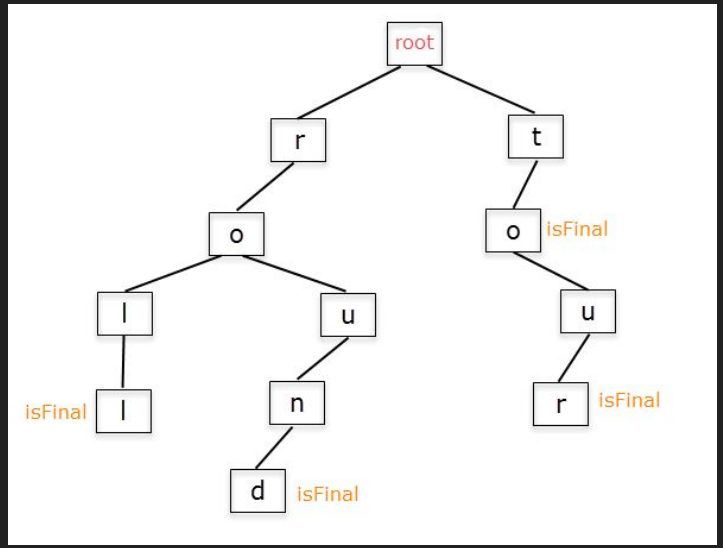
- One of the problems we potentially had with hash maps, was there could be possible collisions of data.
  - This means the lookup time in the worse-case could be O(n)
    - (Everything lands in the same bucket)
  - And if we know we are going to do lots of lookups to test the ‘existence’ of some key, this could be very slow!



A new data structure

- In order to avoid collisions, instead we will build a data structure completely with the data.
- This strategy works well when we are storing ‘unique’ entries as keys
  - (No duplicates--though we could track them if necessary)





Tries

Trie (pronounced ‘try’)

- A data structure used to store information
  - Typically this information is ‘strings’
- It is optimized for very fast for doing ‘lookup’ of keys.
  - Typically we use them with working with ‘string’ data
- Can think of ‘re**trie**val’ as the origin of trie, meaning it is useful for retrieving information.
  - (Again--the pronunciation seems a little bit off, but it is ‘try’)

Trie definition

- ∑ (Capital Sigma)
  - represents a **fixed** alphabet of symbols in our trie.
  - An example would be an alphabet which consists of the set of all lowercase letters in the alphabet
    - {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}
- The alphabet is the data we can work with in our ‘trie’ data structure.
- So any ‘keys’ may consist of these 26 lowercase letters.
- We will label keys as ‘k’.
  - The length of k is 26
    - i.e. |k| = 26

Trie structure

- We structure tries in code as the following:
  - An array of pointers the length of our alphabet.
    - i.e. $|\sum| = 26$
  - A boolean indicating whether we are finished with our word
    - (mini-optimization could be to use a ‘char’ to save some space)

```
typedef struct TrieNode{
        // The branches for the size of a node are the size of the alphabet.
        struct TrieNode* alphabet[ALPHABET_SIZE];
        int end;        // 1 if it is the end of a word, otherwise 0.
}TrieNode_t;
```

Trie is a tree (1/2)

- So if you note the ‘alphabet’ is an array of pointers to other nodes.
- This indicates we have a tree structure
  - We have ‘26’ children in our example
- Previously in binary search tree, we had a left and right node.
  - (2 children)
- Now we will have as many as $|\sum|$ nodes (i.e. 26 in our running example) per level in our tree

```
typedef struct TrieNode{
        // The branches for the size of a node are the size of the alphabet.
        struct TrieNode* alphabet[ALPHABET_SIZE];
        int end;        // 1 if it is the end of a word, otherwise 0.
}TrieNode_t;
```

Trie is a tree (2/2)

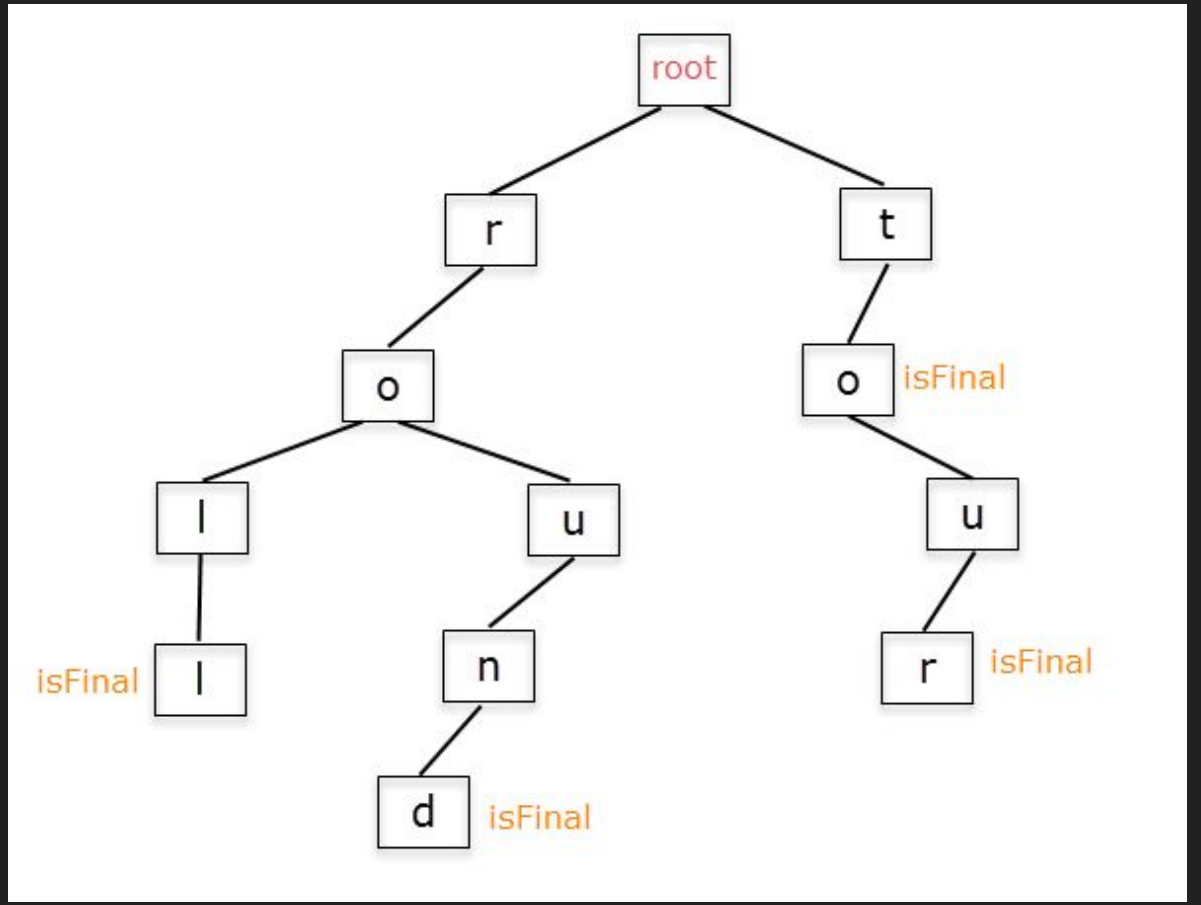
- So if you note the ‘alphabet’ is an array of pointers to other nodes.
- This indicate
  - We have “
- Previously in ode.
  - (2 children
- Now we will xample) per level in our tree

Let’s see an example!

```
typedef struct Tr
    // The branches for the size of a node are the size of the alphabet.
    struct TrieNode* alphabet[ALPHABET_SIZE];
    int end;          // 1 if it is the end of a word, otherwise 0.
}TrieNode_t;
```

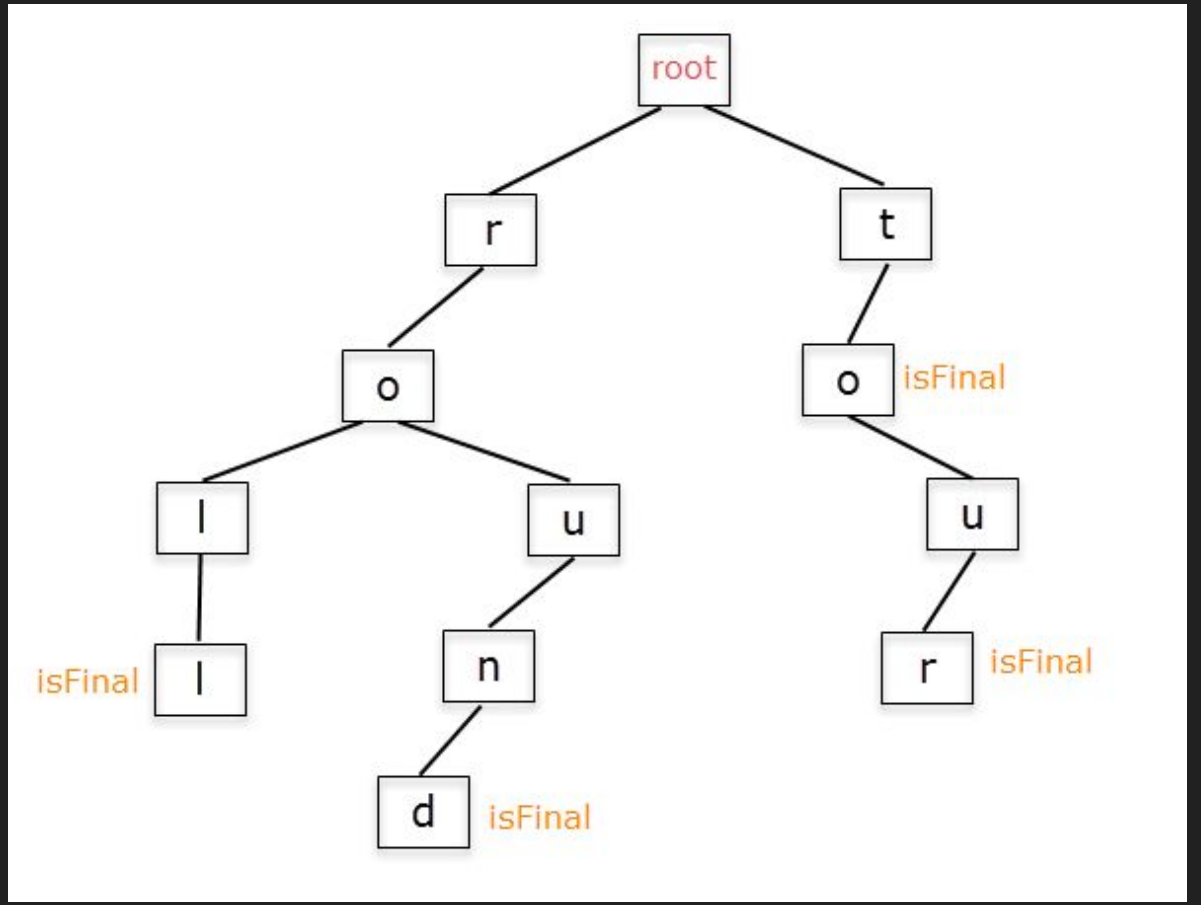
Trie Example (1/2)

- So here's an example trie
  - If we traverse from the root we have some words that begin with 'r' and 't'
  - If they have been added to our dictionary, we mark them 'isFinal'
- Question to audience: Can you identify any words stored?



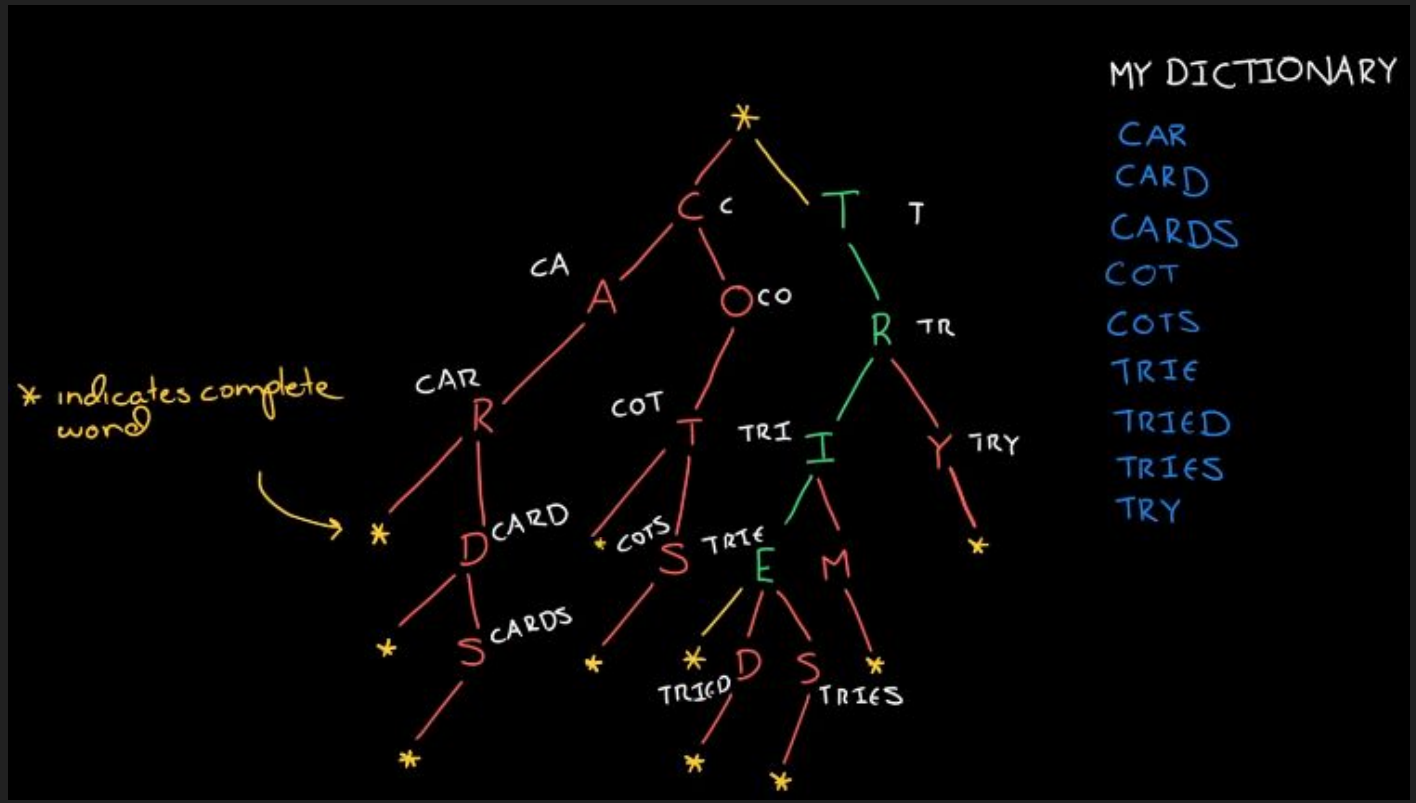
Trie Example (2/2)

- So here's an example trie
  - If we traverse from the root we have some words that begin with 'r' and 't'
  - If they have been added to our dictionary, we mark them 'isFinal'
- Question to audience: Can you identify any words stored?
  - roll, round, to, tour



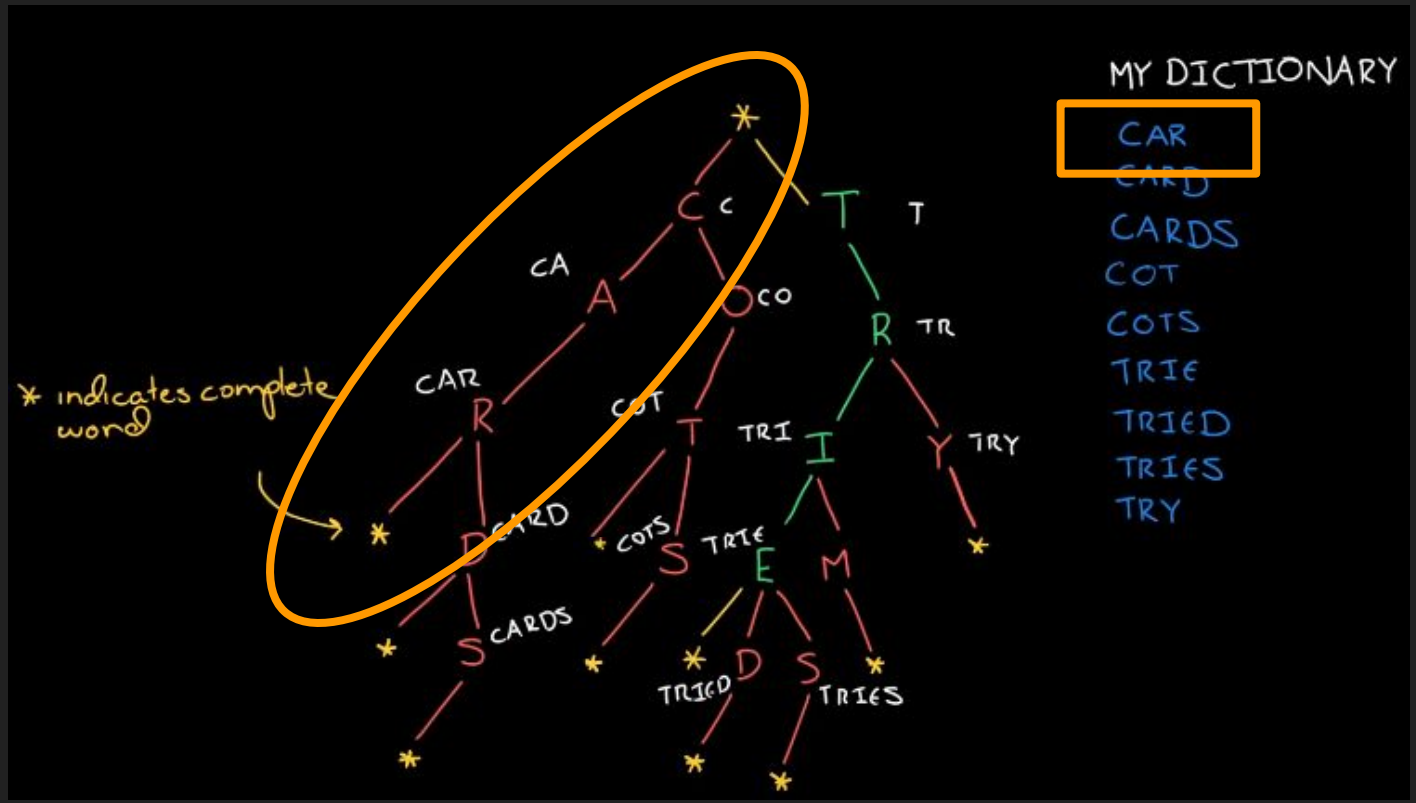
Trie example

Here is an example trie



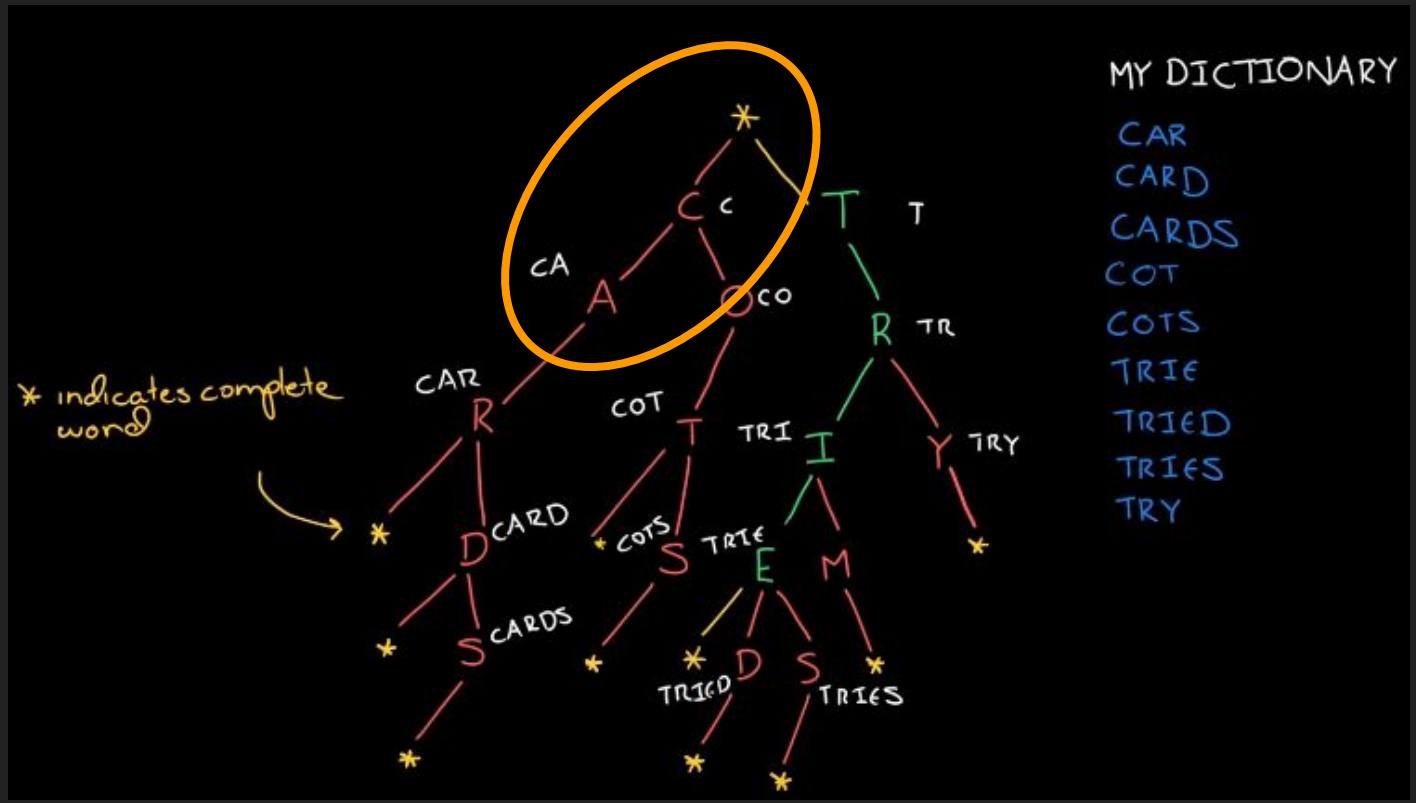
Trie example

We can navigate the trie and find different pieces of data.
Navigating left down to the * finds ‘car’



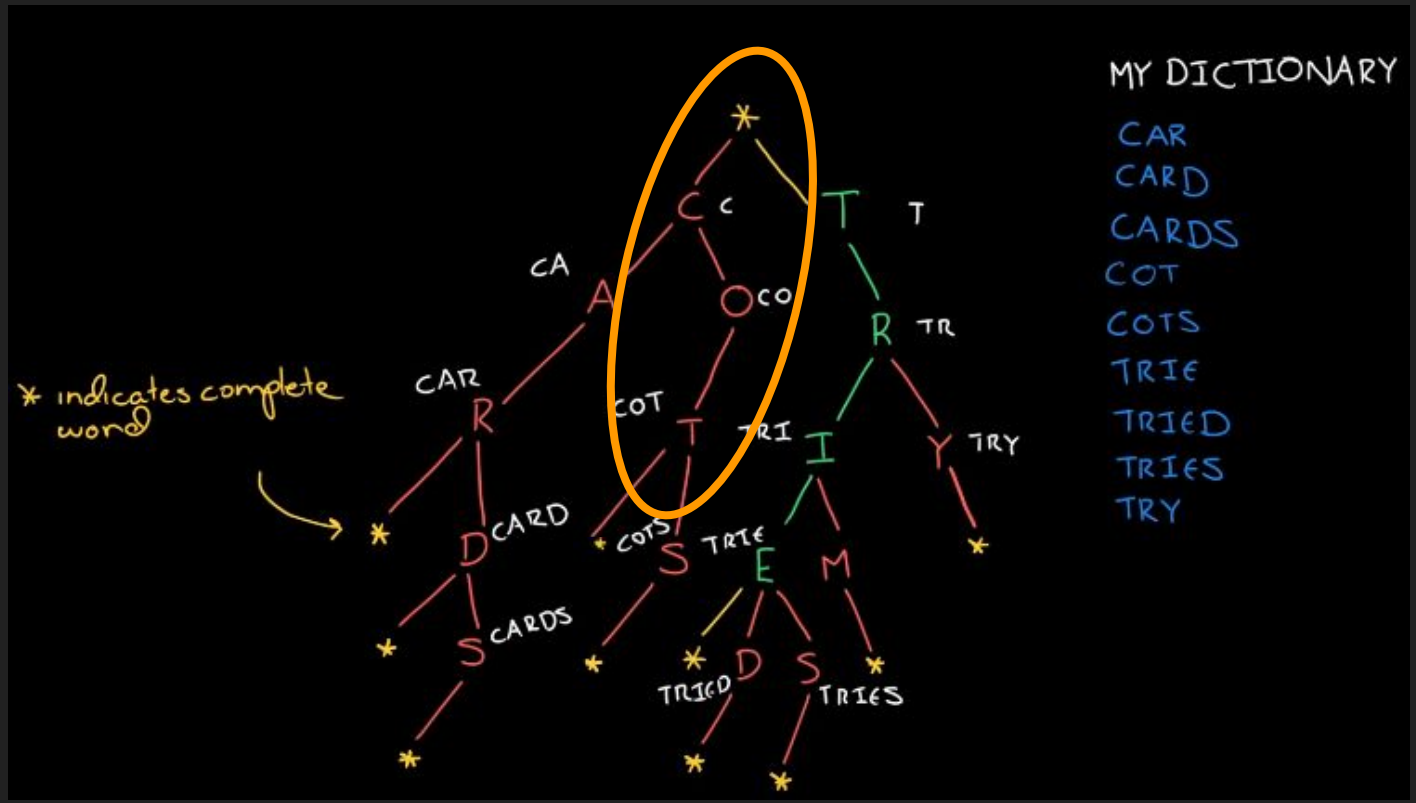
Trie example

‘CA’ however is not a terminal node in our tree



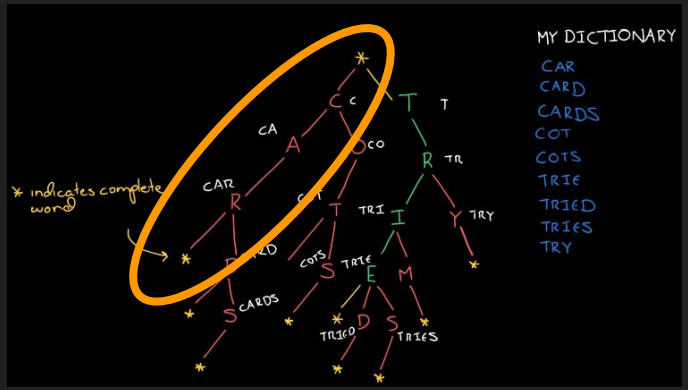
Trie example

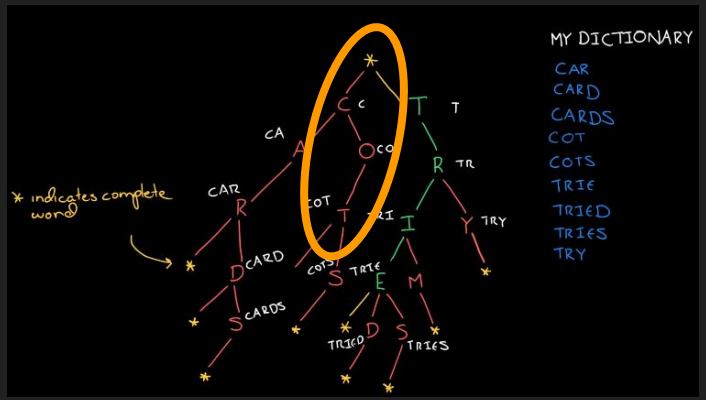
‘COT’ is, and there are more branches to explore



Key Insight - fast lookup!

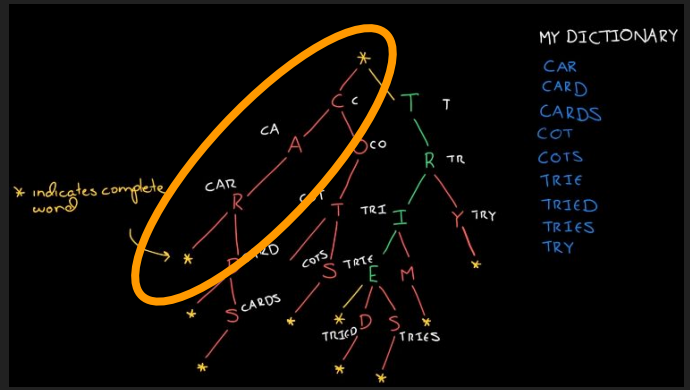
- In each of these examples, notice that the **lookup** time is independent of the amount of data ‘n’ in our data structure.
- The lookup time is the length of our string.
  - e.g. O(|string|)
- So even if we have a dense trie, the lookup time does not change.

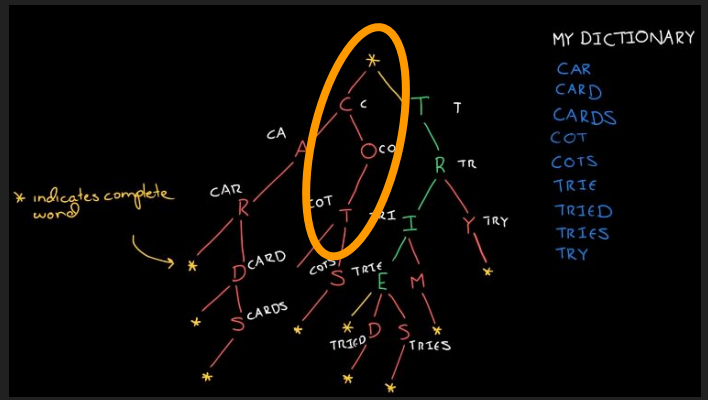




Key Insight - fast insert? (1/2)

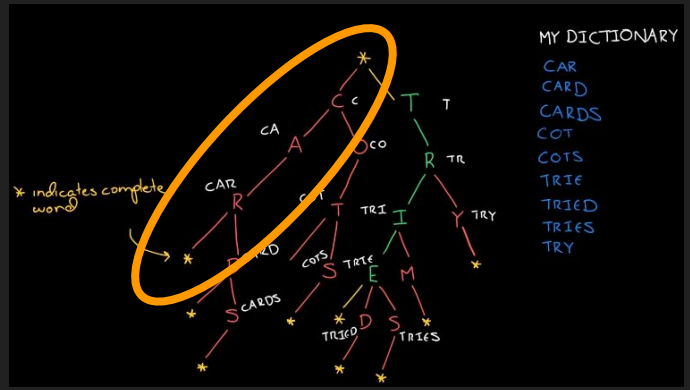
- What do folks think about the insertion time?

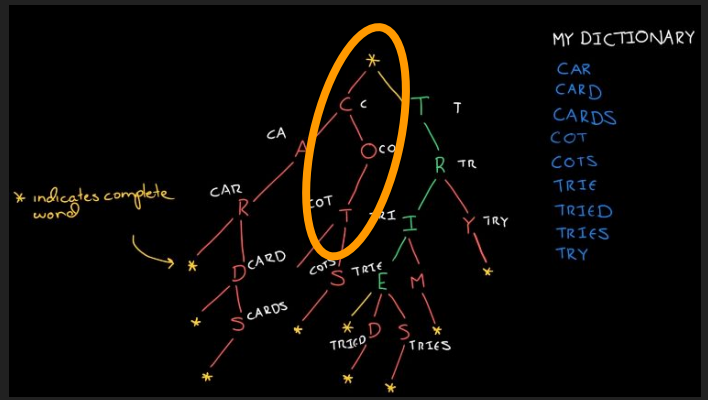




Key Insight - fast insert! (2/2)

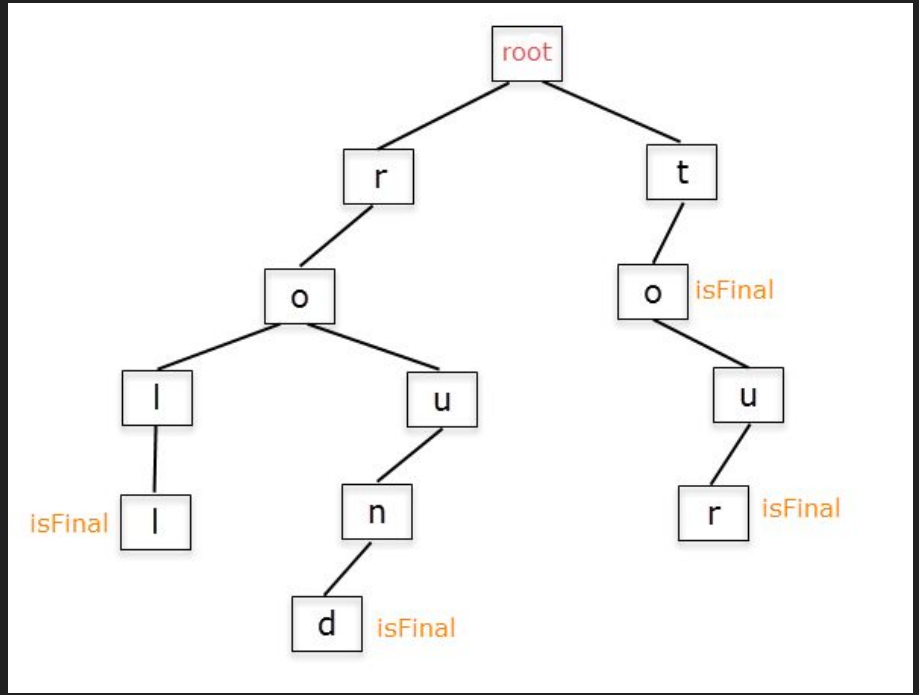
- What do folks think about the insertion time?
- Same insight -- insertion is proportion to the length of the string
  - O(|string|)





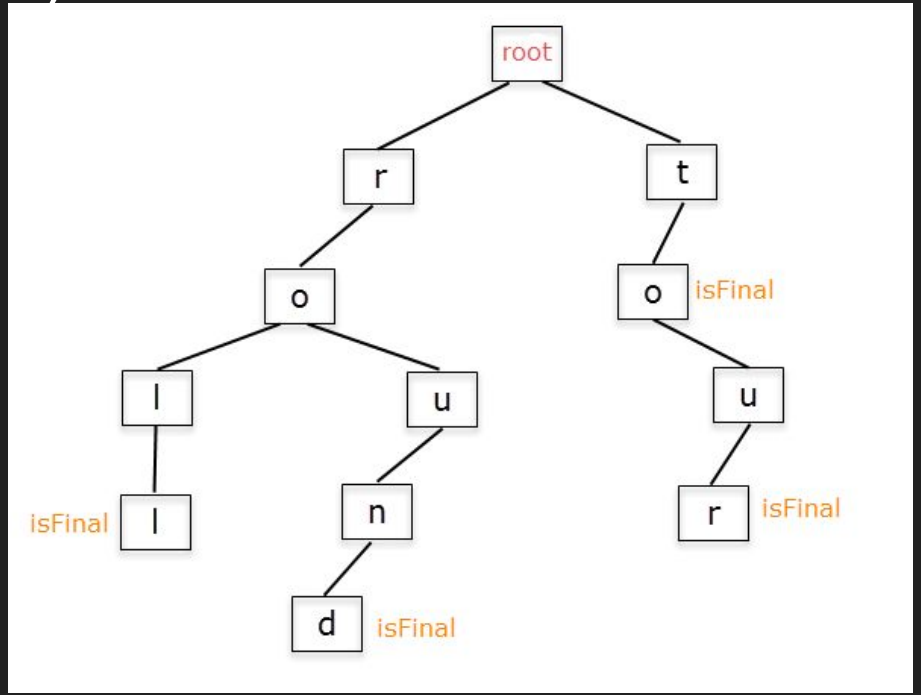
Trie removal

- When removing from a trie, we have two cases
  - If it is a leaf node
    - delete the node
      - Then keep deleting nodes 1 level above until we reach a node marked ‘isFinal’
  - If it is not a leaf
    - Simply flip the ‘isFinal’ bit.



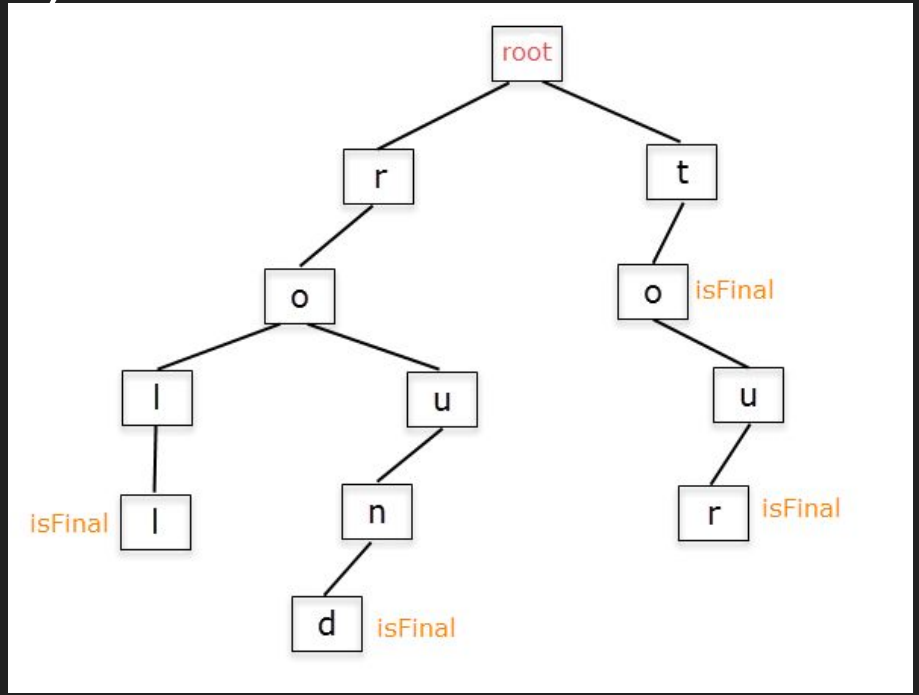
Space complexity of trie (1/2)

- One problem with the trie is that per node, we need to store the entire alphabet
  - So even if we have a sparse tree (i.e. very few words we are storing), we are storing lots of data!
  - So O(|∑|*|k|*n) is the space complexity
    - How could we fix this?



Space complexity of trie (2/2)

- One probablem with the trie is that per node, we need to store the entire alphabet
  - So even if we have a sparse tree (i.e. very few words we are storing), we are storing lots of data!
  - So O(|∑|*|k|*n) is the space complexity
    - How could we fix this?
      - Eliminate unused nodes occasionally?
      - Use a succinct trie (more advanced encoding which compresses bits!)
        - Succinct data structure



When to Use?

- Typically used for validation tasks--especially of words.
- Spell Checkers
  - As soon as you have an invalid word (i.e. something not in your trie), then flag that word.
- Password History
  - Can store a users past passwords in a ‘trie’ structure
  - (Probably a little more secure than just a plain list as well)

Not limited to just ‘characters’ however

- Our key can still be abstracted as any piece of data.
- Let’s look at another example where we use four-digits to store when a university was founded.
- So our key will be in the form (YYYY) where ‘y’ is digit 0-9
  - e.g.
    - 1980
    - 1990
    - 2000
    - 0001 (A very old school!)

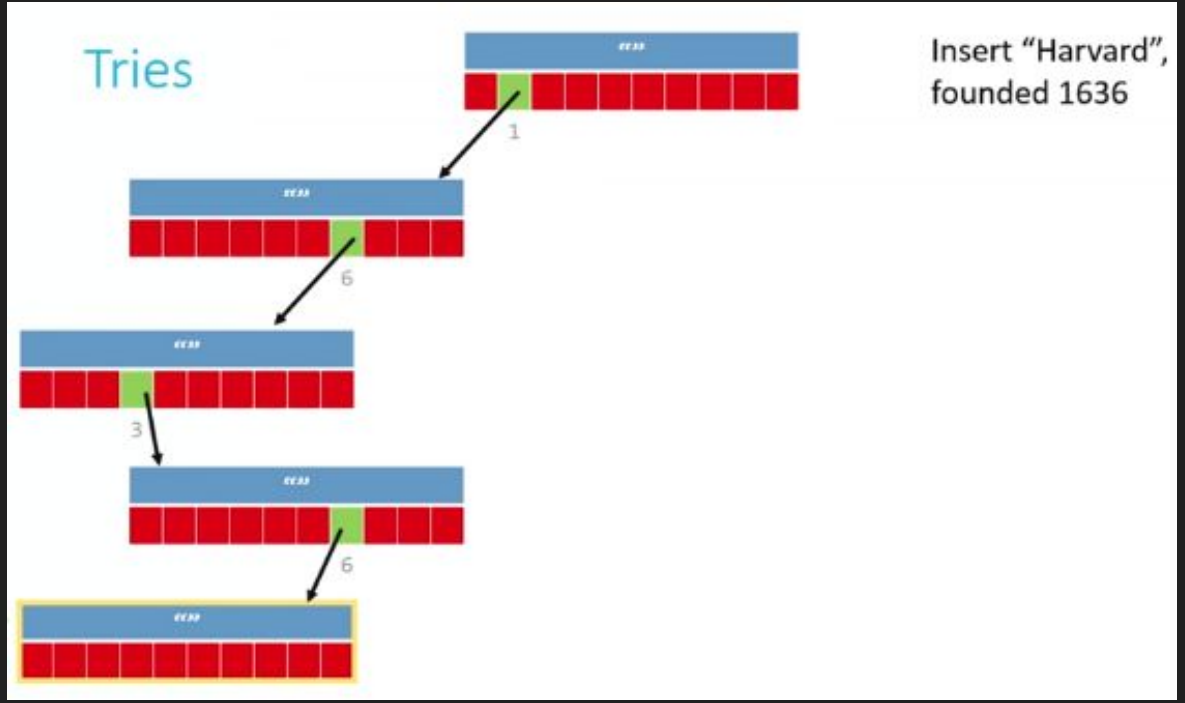
Trie node

- Again, we have the node
- Then at every junction we have 10 other pointers representing the digits for where we can go
  - (i.e. the array of paths)

```
typedef struct _trie
{
    char university[20];
    struct _trie* paths[10];
}
trie;
```

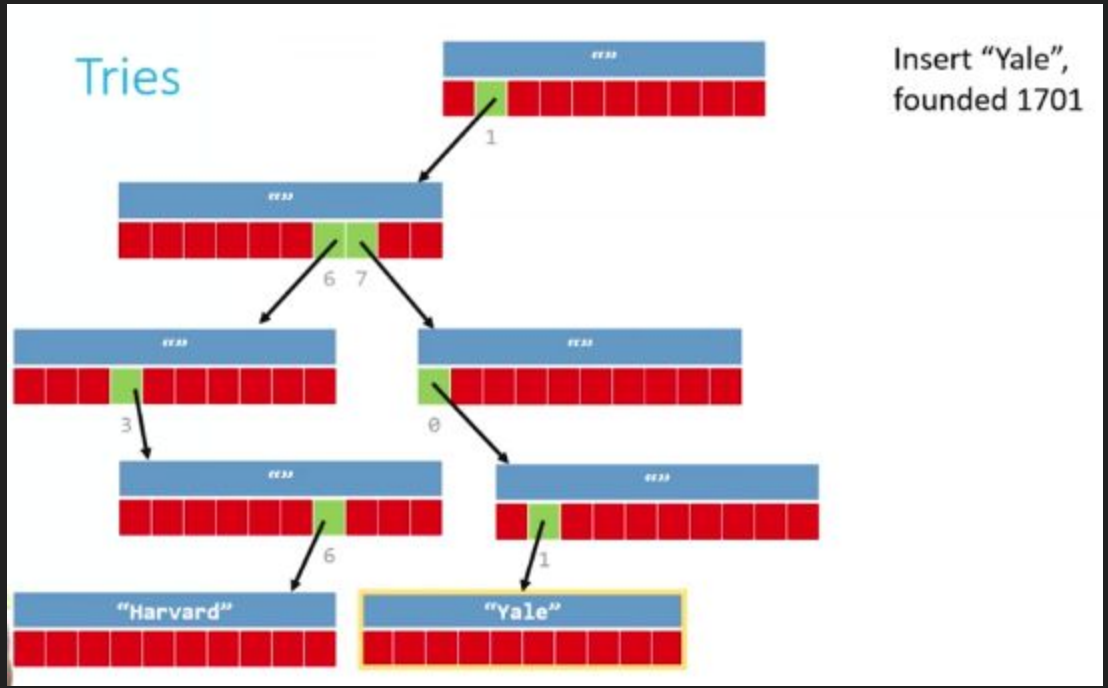
Inserting our first node

- To start again we have a root node.
  - So let’s insert Harvard at 1636
- At each path, we have to ‘malloc’ a new node that has not been explored



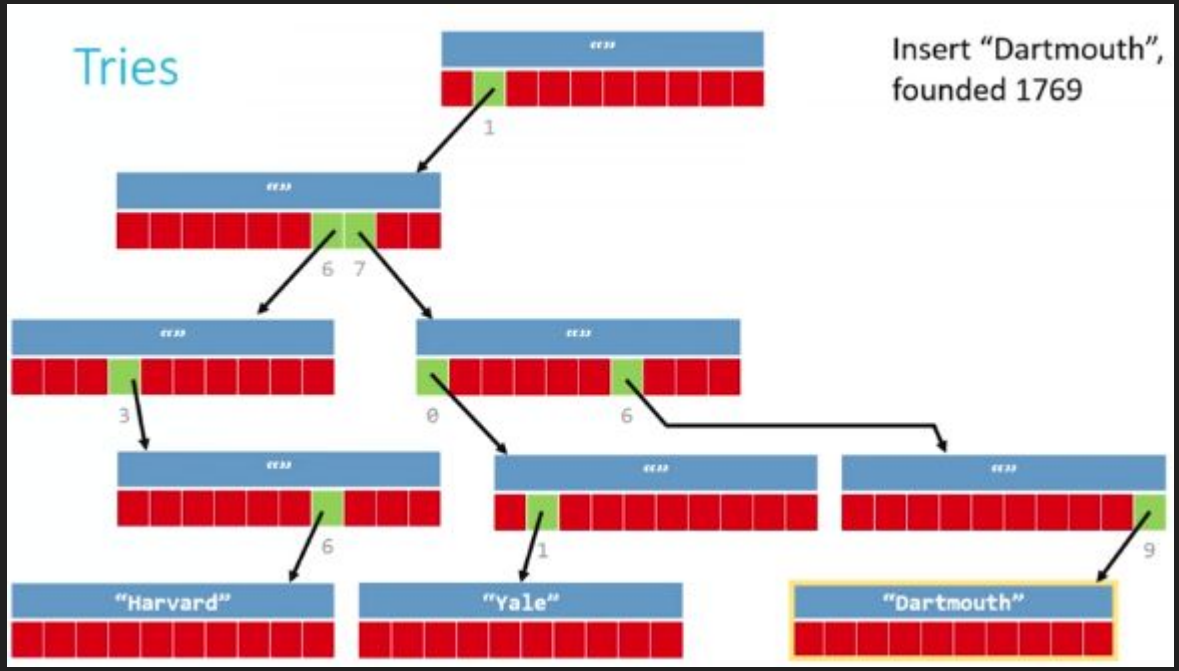
Inserting a second node

- We again start with our root
  - There is no work at ‘1’ that we need to do
  - When we hit the next path, we build a new path from ‘7’
  - Then we continue on.



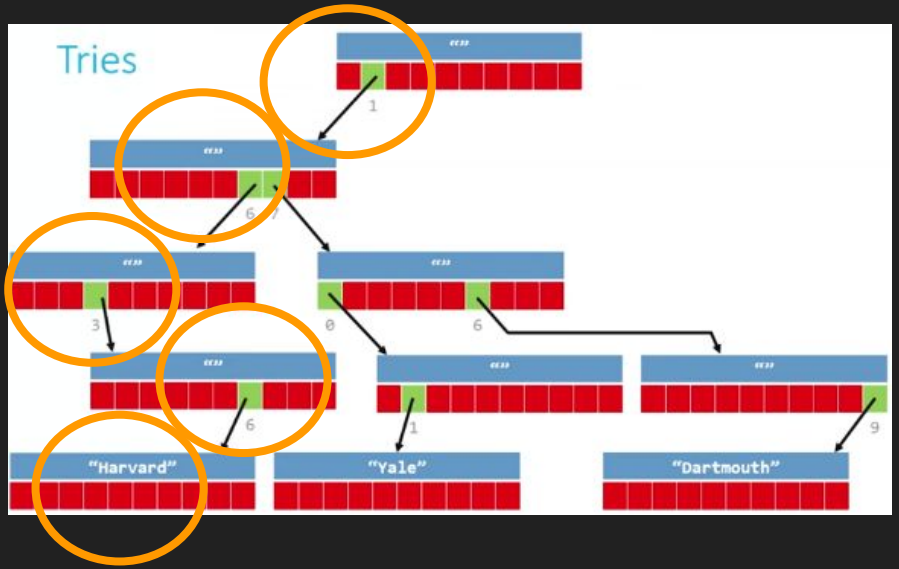
Inserting a third node

- Repeat the procedure



Searching our trie

- Almost the same idea
- This time, we just look through our trie until we find either:
  - NULL (a path does not exist)
  - the end of our string
    - (And if it matches our key)
    - In this case: “Harvard”



Is this a good use for a trie?

- *Maybe* -- it depends on our data set (perhaps Ivy League schools are okay)
- However, if two universities are founded in the same year we get a collision.
- So we have to be careful, or otherwise be willing to replace the data
- Trie's are generally best used when data is unique (i.e., you are representing a set of data)
    - So Social Security numbers, ISBN numbers, words in a dictionary, etc. are probably a better use case here.

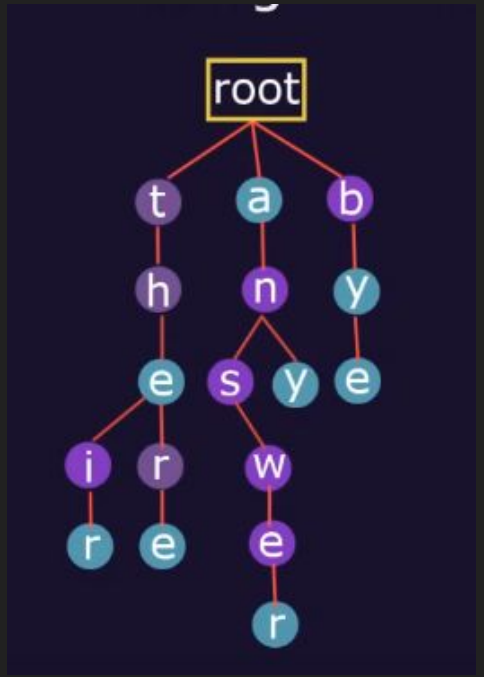
Complexity Analysis

- We are doing constant time lookup ‘4’ lookups each time
- We are doing constant time insertion ‘4’ at every time
- So ‘k’ is the depth of our search tree--it’s fixed in our last example at 4,
  - Or in other words, the length of k. (|k|)
- Thus we are O(|k|) for lookup and search regardless of n
  - (It does not matter how many entries, ‘n’ we have)
  - ^ Think about this and understand that we cannot get more optimal!
- However, the space complexity remains quite large!
  - We could be O(|∑|*|k|*n)

Computer Systems Feed

YOUR DAILY FEED

- (An article/image/video/thought injected in each class!)
- https://www.youtube.com/watch?v=dUBkaqrcYT8



“I try to design things that someone like myself would like to use which is that it just works, and you don't have to think about it at all”

Radia Perlman

“I try to design things that someone like myself would like to use which is that it just works, and you don't have to think about it at all”

Try is our keyword of the day. But this is also just fantastic design advice.

Who is Radia Perlman? [wiki]

- MIT Graduate
- National Inventors Hall of Fame (2016)
- Part of the Internet Hall of Fame (2014)
  - Many other honors
- Most known for her contributions to different networking protocols
  - (See algorhyme)

Algorhyme

I think that I shall never see
A graph more lovely than a tree.

A tree whose crucial property
Is loop-free connectivity.

A tree which must be sure to span
So packets can reach every LAN.

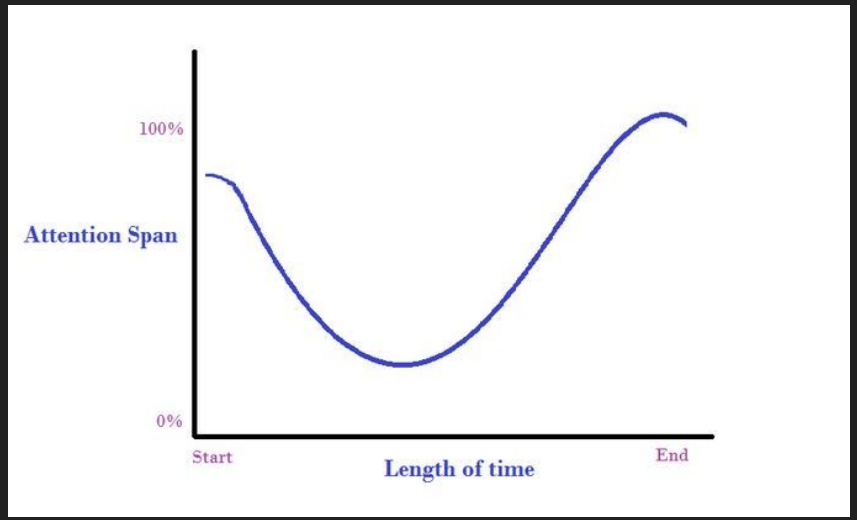
First the Root must be selected.
By ID it is elected.

Least cost paths from Root are traced.
In the tree these paths are placed.

A mesh is made by folks like me
Then bridges find a spanning tree.

Short 5 minute break

- 3 hours is a long time.
- I will try to never lecture for more than half of that time without some sort of ‘break’ or transition to an in-class activity/lab.
- Use this time to stretch, check your phones, eat/drink something, etc.



String Matching

String Matching Problem (1/2)

- With the introduction of ‘tries’ that is a good introduction to the idea of finding strings
- With tries we can ‘lookup’ to see if a word exists.
- But often times, we want to see if a string occurs within some larger text.

String Matching Problem (1/2)

- With the introduction of ‘tries’ that is a good introduction to the idea of finding strings
- With tries we can ‘lookup’ to see if a word exists.
- But often times, we want to see if a string occurs within some larger text.
- The string matching problem:
  - Given a text string T and a nonempty string P
  - Find all occurrences of P in T.

Solutions to String Matching

- The string matching problem:
  - Given a text string T and a nonempty string P
  - Find all occurrences of P in T.
- Question to the audience:
  - So how would you solve this problem?
  - *Hint* Give yourself some example
    - T = “Hello my name is Mike”
    - P = “Mike”

Brute Force Approach (A solution)

- Check every possible position of the string, and see if it matches our pattern
  - Return the index ‘i’ of where the pattern starts

```
1 function NaiveSearch(string s[1..n], string pattern[1..m])
2    for i from 1 to n-m+1
3       for j from 1 to m
4          if s[i+j-1] ≠ pattern[j]
5             jump to next iteration of outer loop
6       return i
7    return not found
```

Brute-Force String match complexity

- We can see two inner loops ranging from (n-m+1) and m.
- Assuming our ‘text’ is length n, and n >> m (i.e. n is much greater than m)
  - We get O(n*m).

```
1 function NaiveSearch(string s[1..n], string pattern[1..m])
2    for i from 1 to n-m+1
3       for j from 1 to m
4          if s[i+j-1] ≠ pattern[j]
5             jump to next iteration of outer loop
6       return i
7    return not found
```

Real Implementation

- See module page for a test case.

```
 5 // T - Our Text
 6 // p - The pattern we want to search
 7 // Returns the index(0 or greater) of the matched string position.
 8 // Returns a negative value if there are no occurrences found
 9 int bruteForceStringMatch(char* T, char* p){
10     unsigned int i, j;
11
12     unsigned int n = strlen(T);
13     unsigned int m = strlen(p);
14     // If we are searching for a string longer than our text
15     // return false.
16     if(m > n){
17         return -2; // '-2'tells us pattern 'p' was greater than the text.
18     }
19     // Otherwise do a character by character search
20     for(i=0; i < n-m+1; i++){
21         j=0;// Start at the start of our pattern each iteration
22         while(j<m && p[j]==T[i+j]){
23             j++;
24             if(j==m){
25                 return i; // Return index of match
26             }
27         }
28     }
29     // If we reach this point, then we never got a full match, return -1.
30     return -1;
31 }
```

Why to care about string matching? (1/2)

- Classic examples like tools like ‘grep’ we have been using
- Anytime you press ‘Ctrl+f’ and search for a word
- In genetics you may be looking for a pattern that indicates a disease
  - e.g. ATGCATGCGATCGTGAGCTGA....(10 million other characters)
    - Perhaps pattern AATTGC is meaningful to find.
    - (And in this case--performance really matters with the large scale!)
- For antivirus databases
  - Typically there is some ‘binary signature’ that indicates a malicious program

Why to care about string matching? (2/2)

- Classic examples like tools like ‘grep’ we have been using
- Anytime you
- In genetics y es a disease
  - e.g. ATGC)
    - Perh
    - (And cale!)
- For antivirus
  - Typically tl program

So let’s improve on our brute-force solution

Knuth-Morris-Pratt (KMP) Algorithm [wiki]

- Knuth and Pratt (and independently Morris) discovered a faster string matching algorithm which they published together in 1977
- The key idea is they can pre-compute a table of the *next possible match to search in a string*.
  - This improves upon looking at every single character like in our brute-force approach
- We can think of this as a ‘sliding window’ technique.
- Let’s visualize it first!

Problem

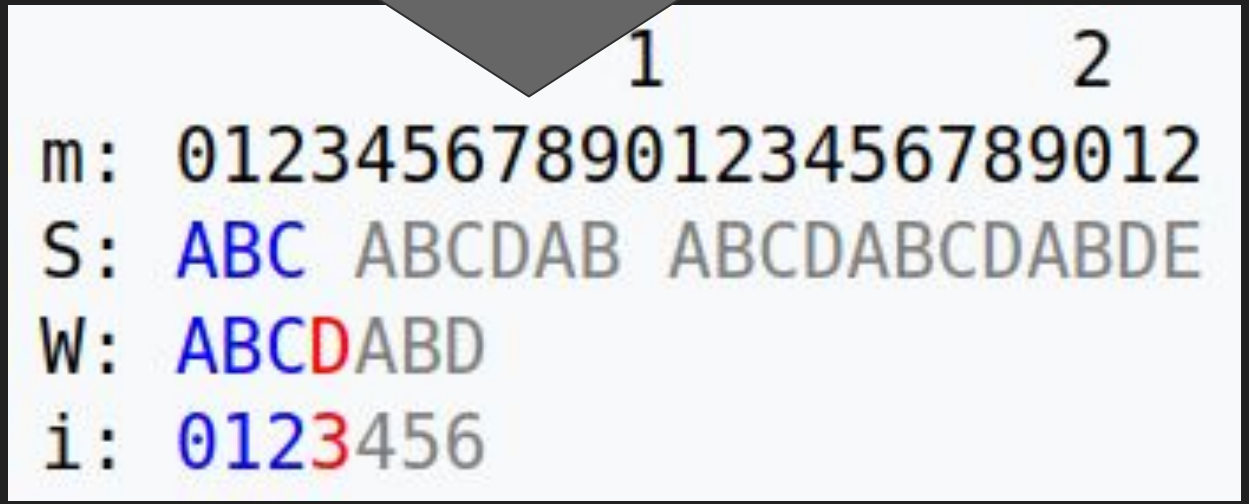
- Given text ‘S’: = `ABC ABCDAB ABCDABCDABDE`
- Searching for pattern ‘W’: `ABCDABD`

Problem

- Given text ‘S’: =
- Searching for pattern ‘W’: ABC

m is the index, i is how many characters we have serached in ‘W’

Narration:

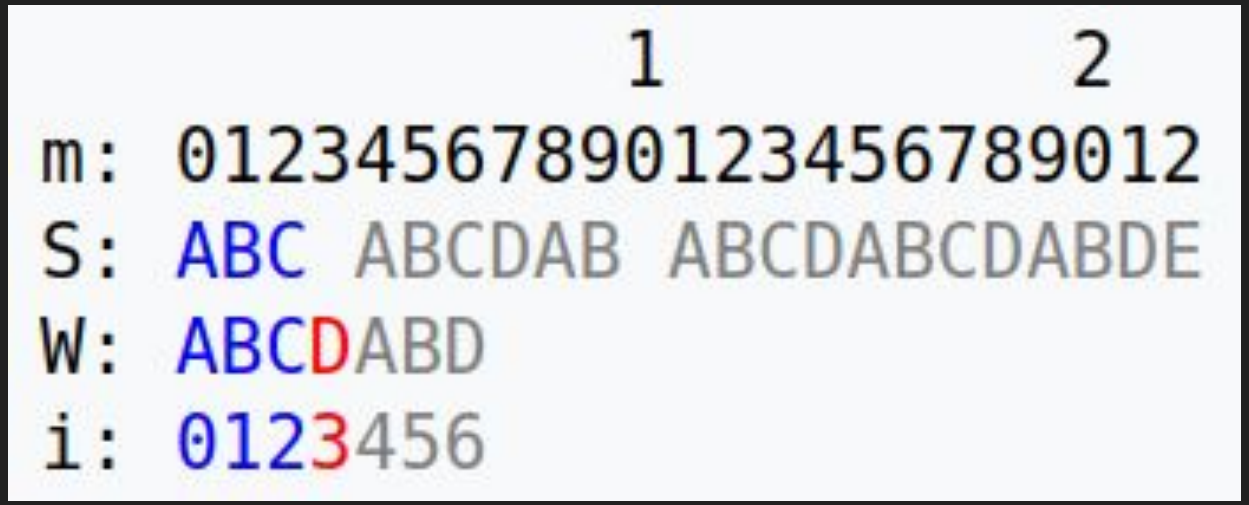


Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

We start out by checking each matching character

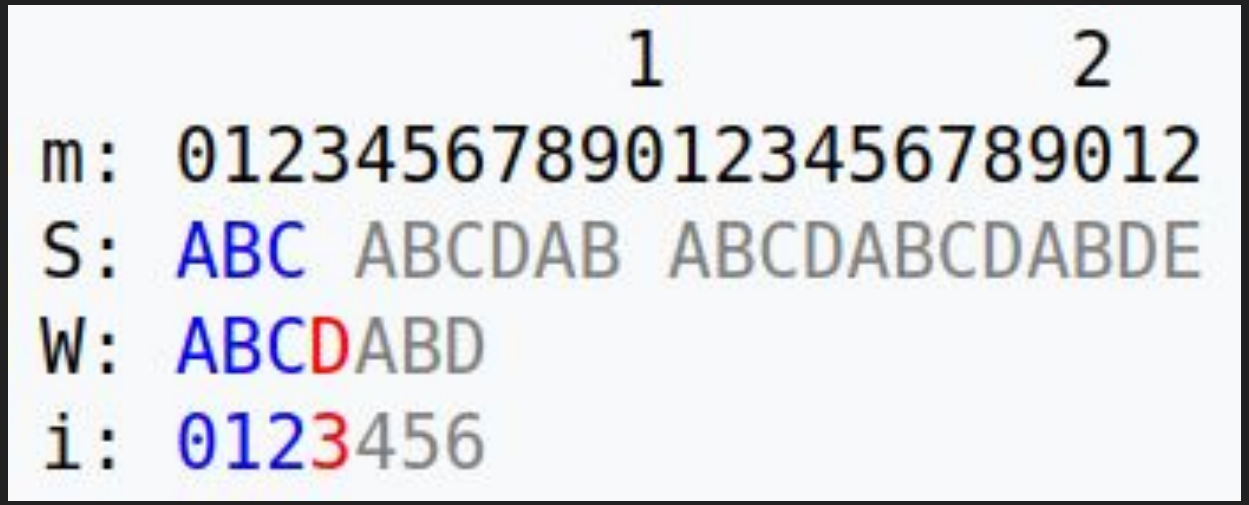


Problem

- Given text ‘S’: = `ABC ABCDAB ABCDABCDABDE`
- Searching for pattern ‘W’: `ABCDABD`

Narration:

We happen to find the first ‘3’ that match, but the 4th character ‘D’ does not match the space.



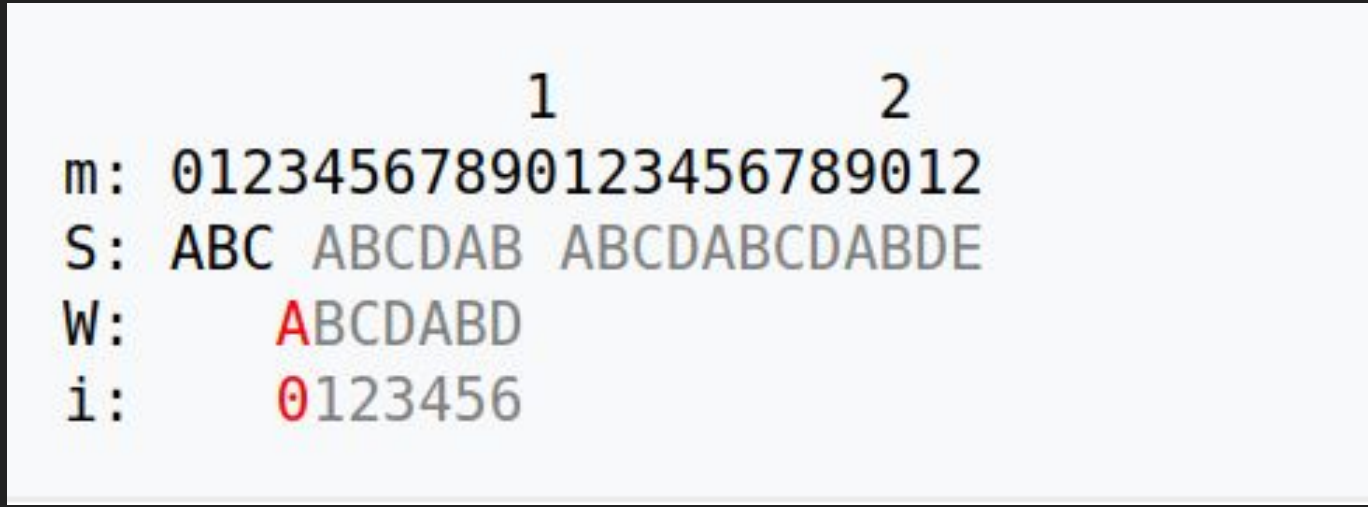
Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

So now we move to the next characters we have not yet checked.

We immediately fail.

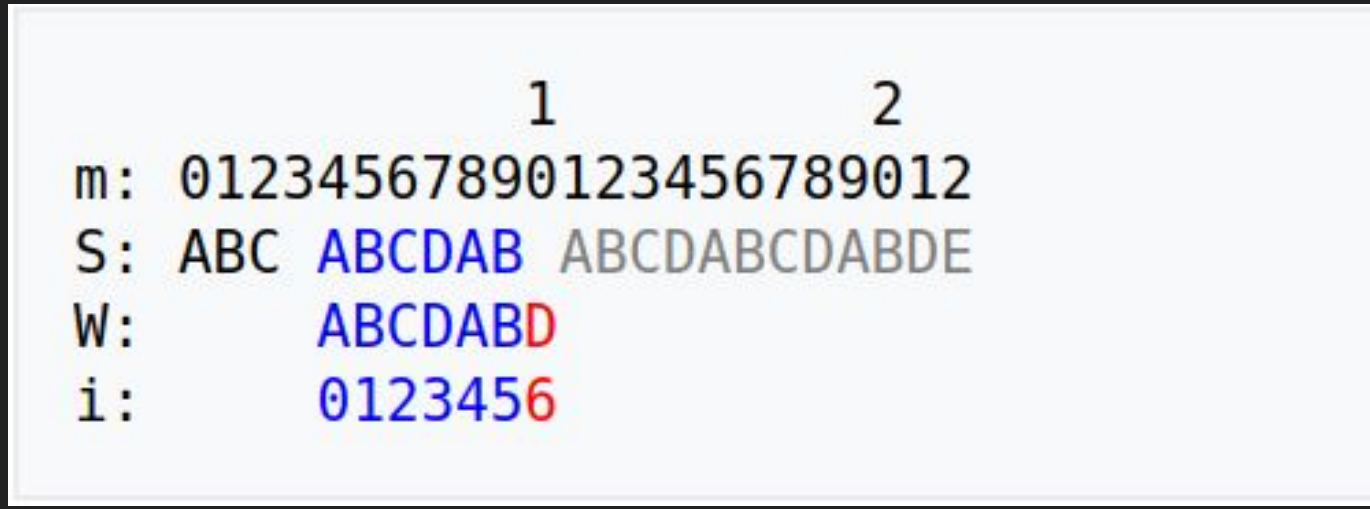


Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

We try again, and we find 6 of 7 matches.

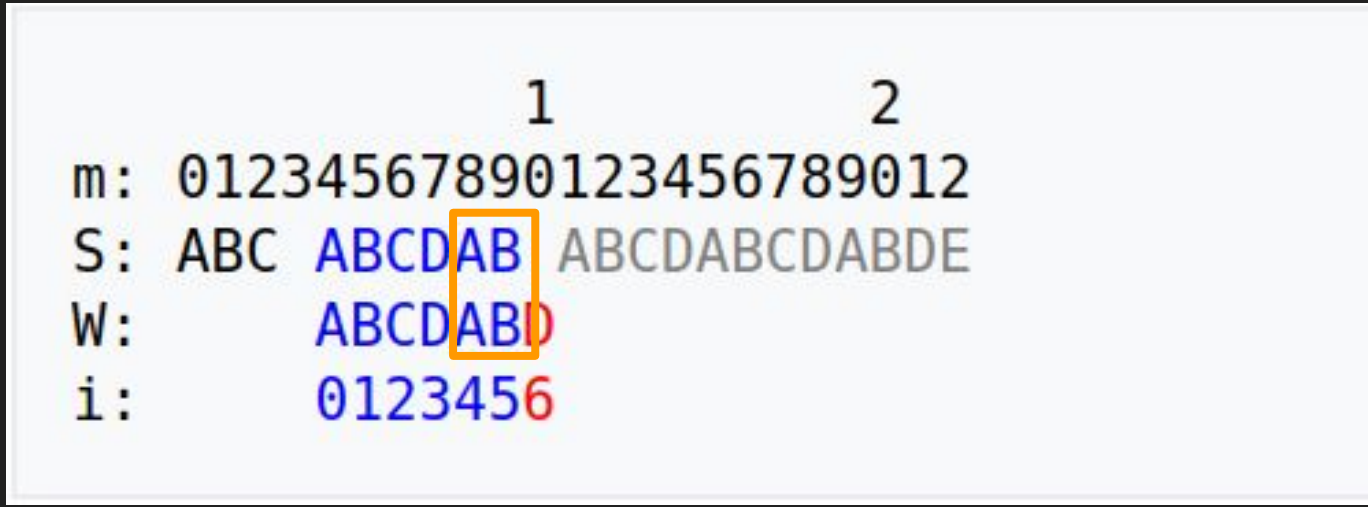


Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

We again fail, but note that we find a ‘substring’ ‘AB’ that could be the start of a potential match.



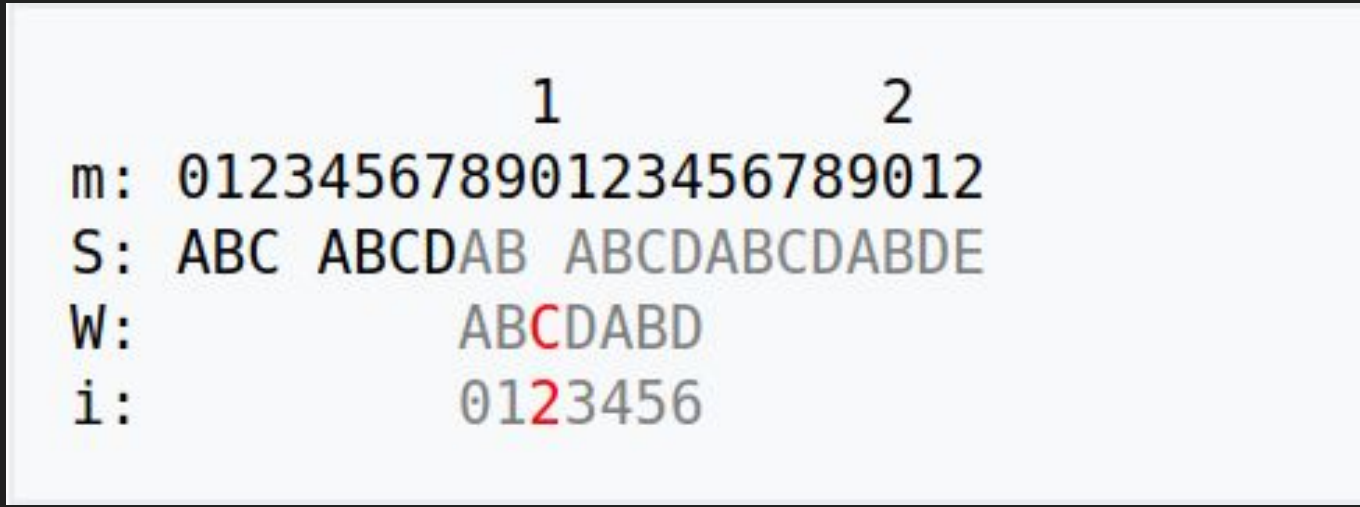
Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

So we actually start from that substring, and continue.

Unfortunately--no match again!

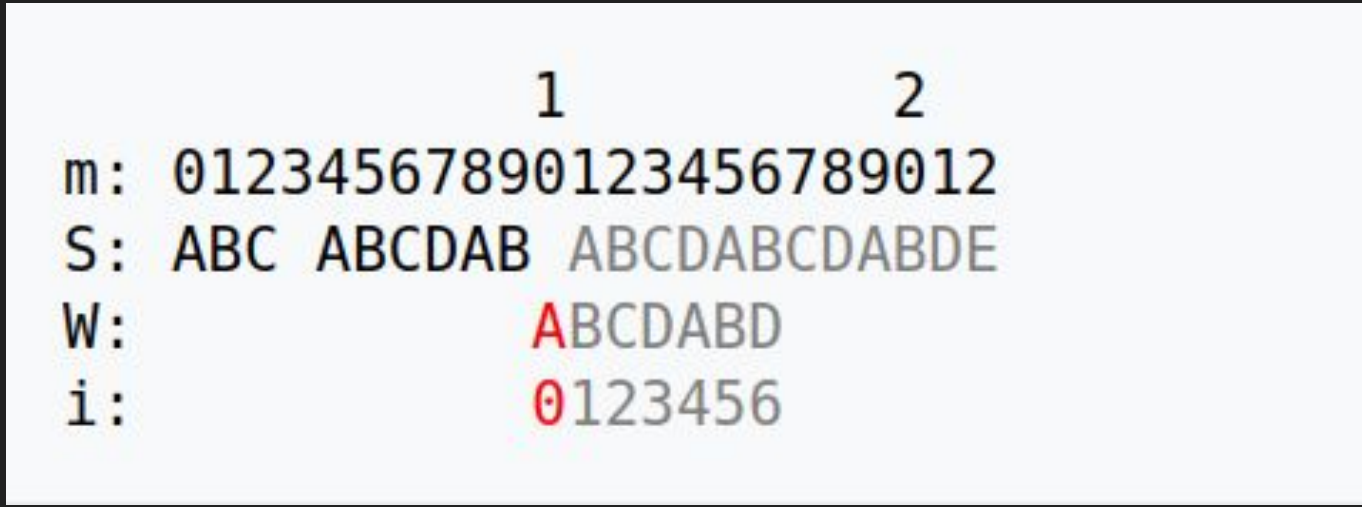


Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

No potential substrings to search from, so we start from the furthest index we have made progress from.

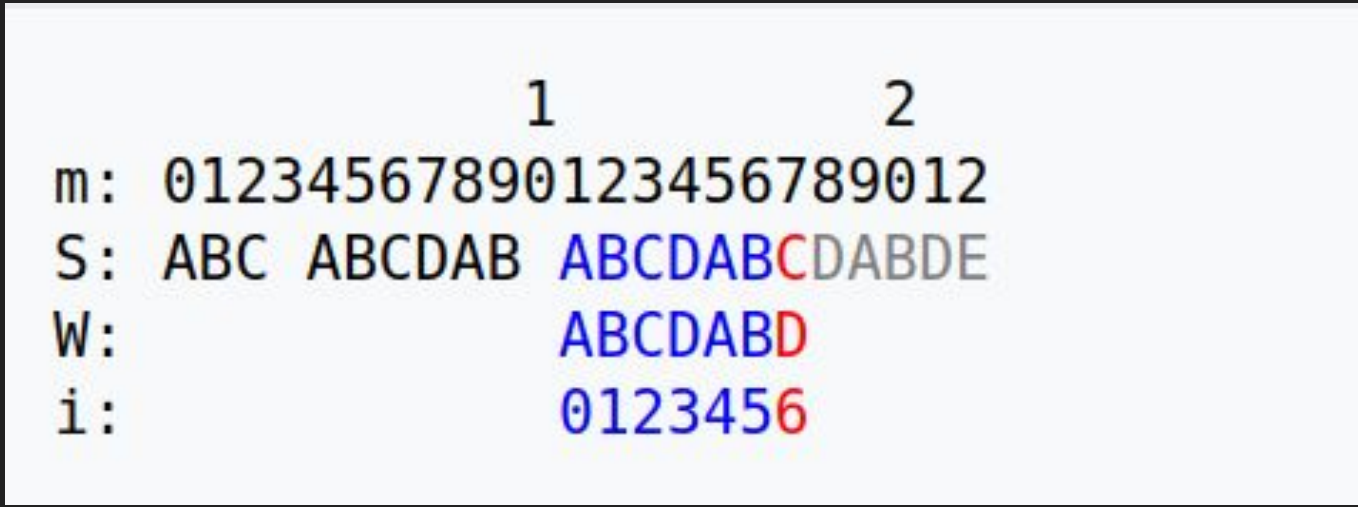


Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

We try again! Unfortunately we do not match all 7 characters

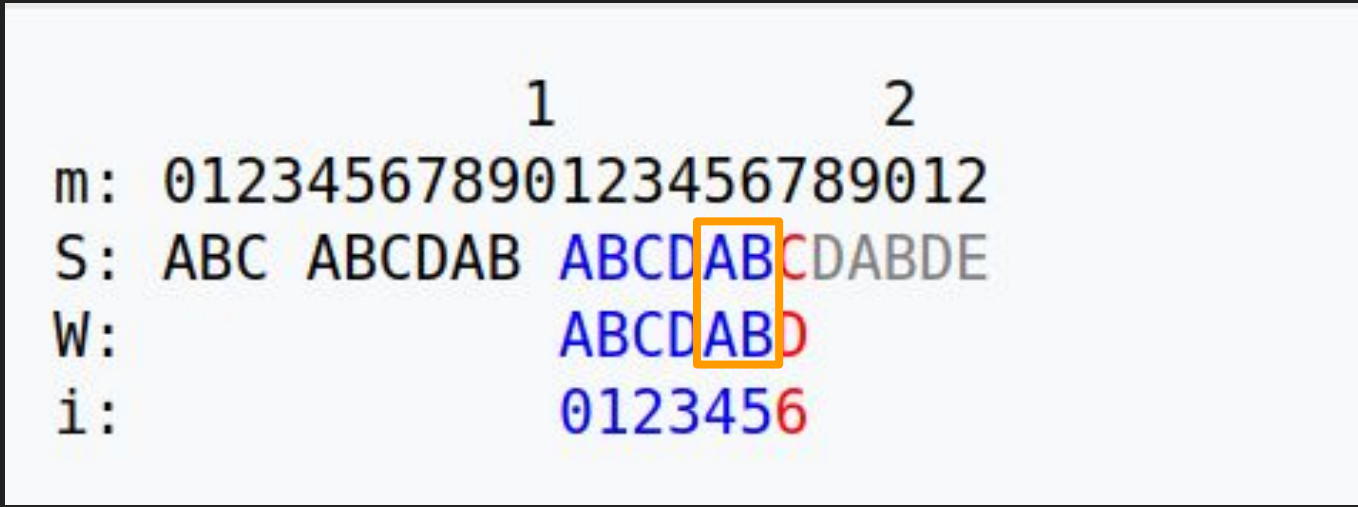


Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

But we find another substring, so we start from here



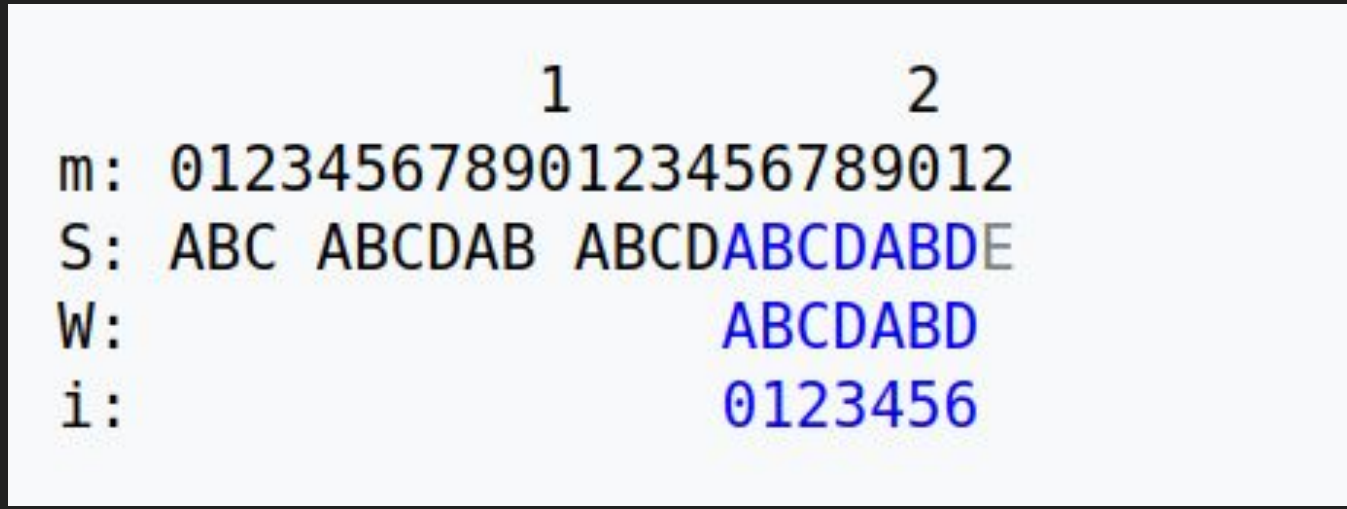
Problem

- Given text ‘S’: = ABC ABCDAB ABCDABCDABDE
- Searching for pattern ‘W’: ABCDABD

Narration:

Ah, eventually find a perfect match.

No more occurrences to search.



KMP Pseudocode

- Our input:
  - The text we will search
  - The pattern we want to find

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an array of integers, P (positions in S at which W is found)
        an integer, nP (number of positions)

    define variables:
        an integer, j ← 0 (the position of the current character in S)
        an integer, k ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    let nP ← 0

    while j < length(S) do
        if W[k] = S[j] then
            let j ← j + 1
            let k ← k + 1
            if k = length(W) then
                (occurrence found, if only first occurrence is needed, m ← j - k  may be returned here)
                let P[nP] ← j - k, nP ← nP + 1
                let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

KMP Pseudocode

- Output:
  - Bookkeeping structures that report where we found pattern ‘W’ in ‘S’

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an array of integers, P (positions in S at which W is found)
        an integer, nP (number of positions)

    define variables:
        an integer, j ← 0 (the position of the current character in S)
        an integer, k ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    let nP ← 0

    while j < length(S) do
        if W[k] = S[j] then
            let j ← j + 1
            let k ← k + 1
            if k = length(W) then
                (occurrence found, if only first occurrence is needed, m ← j - k  may be returned here)
                let P[nP] ← j - k, nP ← nP + 1
                let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

KMP Pseudocode

- Where we are searching
- (magic table ‘T’)
  - to be continued

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an array of integers, P (positions in S at which W is found)
        an integer, nP (number of positions)

    define variables:
        an integer, j ← 0 (the position of the current character in S)
        an integer, k ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    let nP ← 0

    while j < length(S) do
        if W[k] = S[j] then
            let j ← j + 1
            let k ← k + 1
            if k = length(W) then
                (occurrence found, if only first occurrence is needed, m ← j - k  may be returned here)
                let P[nP] ← j - k, nP ← nP + 1
                let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

KMP Pseudocode - Partial match table

- Goal is to find fall back positions by finding ‘suffix’ that could occur.
  - In our previous case, that was ‘AB’ where we see that it repeats.
  - At the second AB, note at ‘D’ that if we reach that point we start our next search at index 2 and save some work.

i	0	1	2	3	4	5	6	7
W[i]	A	B	C	D	A	B	D	
T[i]	-1	0	0	0	-1	0	2	0

```
algorithm kmp_search:
    input:

    outp

    defi

    let

    whil

            (occurrence found, if only first occurrence is needed, m ← j - k  may be returned here)
            let P[nP] ← j - k, nP ← nP + 1
            let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

KMP Pseudocode

- The comparison

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an array of integers, P (positions in S at which W is found)
        an integer, nP (number of positions)

    define variables:
        an integer, j ← 0 (the position of the current character in S)
        an integer, k ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    let nP ← 0

    while j < length(S) do
        if W[k] = S[j] then
            let j ← j + 1
            let k ← k + 1
            if k = length(W) then
                (occurrence found, if only first occurrence is needed, m ← j - k  may be returned here
                let P[nP] ← j - k, nP ← nP + 1
                let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

KMP Visualization

- Note the ‘sliding’ effect
- Occasionally we are able to hop a few positions

KMP Complexity

- The complexity ends up being O(n)
- https://youtu.be/HBI-7Hs0Lx4?t=357
  - This video shows the construction of the prefix table once more.

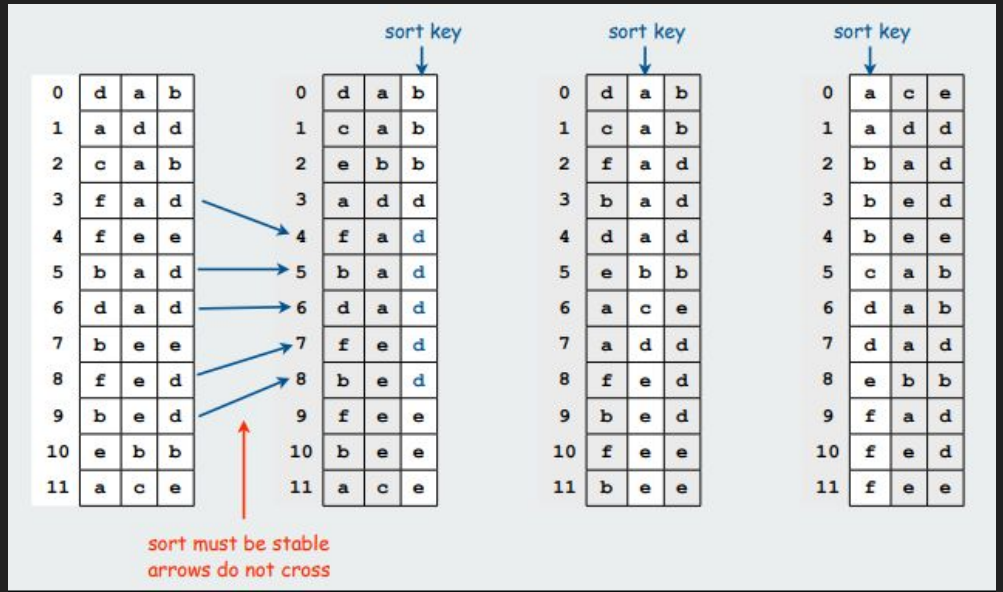
Short Topics

A few short topics

- Here are a few fun topics in a slide that you will likely see later in CS 5800.
- Unfortunately we do not have time to cover them in a 7 week class
  - (Think of all of the stuff we have already learned and are building on--wow!)
- Anyway, here are some topics in a slide!

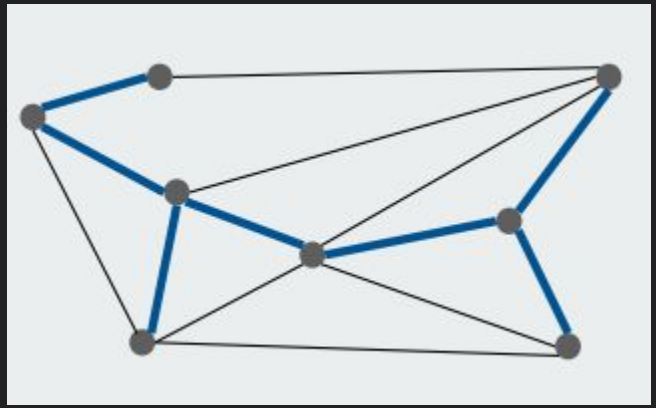
Radix Sort

- In a Sentence:
  - Considers the least-significant character of a key. Sorts based on those keys, then move forward.
- Resource to learn more: https://www.cs.princeton.edu/~rs/AlgsDS07/18RadixSort.pdf



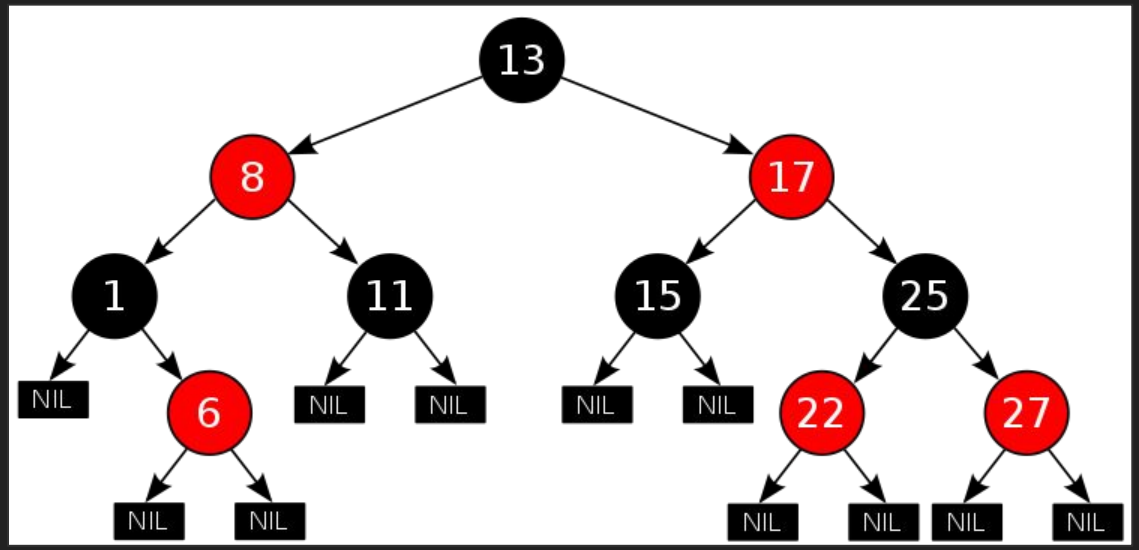
Minimum Spanning Tree

- In a Sentence:
  - Computes the minimum weight set of edges that connect all vertices
- Resource to learn more: https://www.cs.princeton.edu/~rs/AlgsDS07/14MST.pdf
  - Search Kruskal's
  - Search Prim's



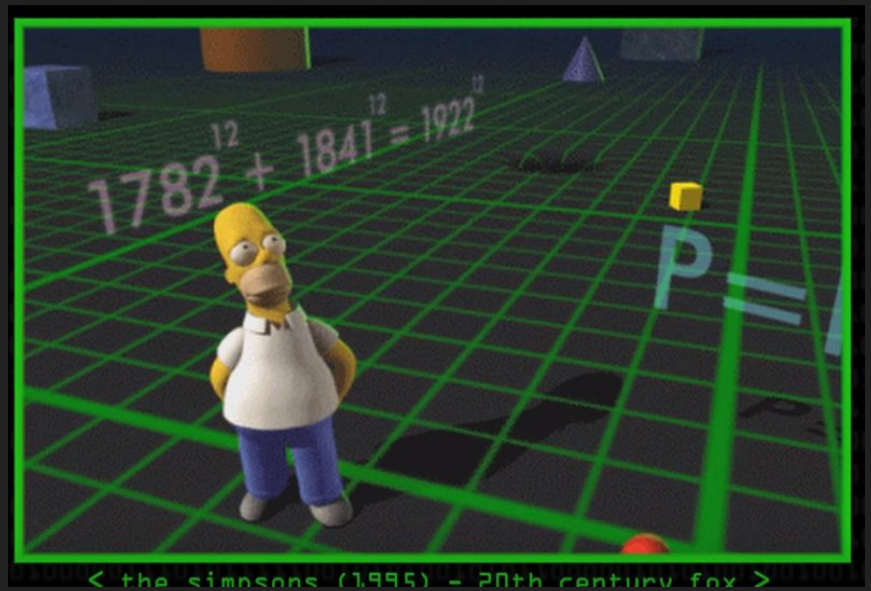
More Data Structures - Red Black Trees

- In a Sentence:
  - Our binary search trees can become unbalanced and potentially find O(n) lookup time, red-black trees ‘rotate’ and rebalance the tree to ensure this does not happen.
- Resource to learn more: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree



Tractability - P vs NP

- In a Sentence:
  - “An equivalent definition of NP is the set of decision problems solvable in polynomial time by a non-deterministic Turing machine.”
- Resource to learn more:
  - https://www.youtube.com/watch?v=dJUEkjxylBw



Exam Review

- Questions?
- Topics?

In-Class Activity

1. Complete the in-class activity from the schedule
   a. (Do this during class, not before :))
2. Please take 2-5 minutes to do so
3. These make up a total of 5% of your grade
   a. We will review the answers shortly

In-Class Activity or Lab (Enabled toward the end of lecture)

- In-Class Activity link
  - (This is graded)
  - This is an evaluation of what was learned in lecture.

This lecture in summary

- We have explored
  - Tries
  - String Matching
  - Some additional topics

Algorithm, Data Structure, and Proof Toolboxes

For this course, I want you to be able to see how each data structure and algorithm is different.

- For data structures learn how each restriction on how we organize our data causes tradeoffs
- For algorithms, think about the higher level technique

Algorithm Toolbox: Searches and Sorts

Comparison Sorts

Bubble Sort - $O(n^2)$

6 5 3 1 8 7 2 4

Swap adjacent elements and 'bubble' up element

Selection Sort - $O(n^2)$

5 3 4 1 2

Selection Sort

Search for minimum element and place in ordered position amongst unordered elements

Insertion Sort - $O(n^2)$

6 5 3 1 8 7 2 4

Select each element and place in its sorted position amongst all elements that have been previously placed

Heap Sort- $O(n*\log_2(n))$

Divide and Conquer Sorts

Merge Sort- $O(n*\log_2(n))$

Randomized Algorithms

QuickSort- $\Theta(n*\log_2(n))$ ("theta")

$O(n^2)$ - in the worst case

Searches

Linear Search - $O(n)$

Search and compare each element one at a time

Binary Search - $\log_2(n)$

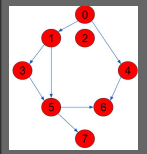
Search sorted data from midpoint, eliminate values less than or greater than 'element' you are search for each step until we match the mid

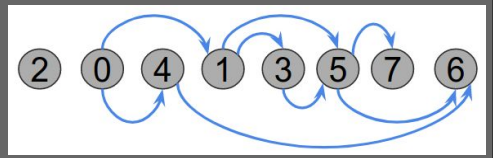
Algorithm Toolbox: Trees and Graphs

Tree Sorts

Topological Sort

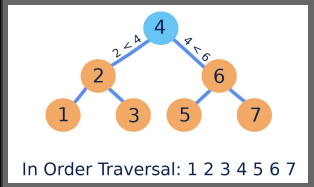
Generates a possible linear ordering of nodes from a Tree by performing a DFS on each unvisited node. O(|V|+|E|)





Tree and/or Graph Searches

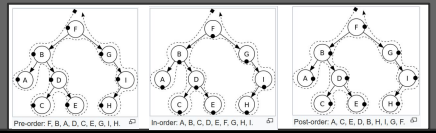
Binary Search Tree



Data structure containing a left and right child
$\Theta(\log_2(n))$ for search, insertion, and deletion

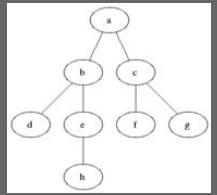
Depth-First Search (DFS)

Traverse graph (or tree) as far as possible in a direction, storing nodes in a stack. O(|V|+|E|)



Breadth-first search (BFS)

Traverse graph (or tree) as widely as possible (level-order traversal) storing each nodes neighbors in a queue.
O(|V|+|E|)



Shortest Path

Dijkstra's Shortest Path

Greedily take the shortest path from each node.

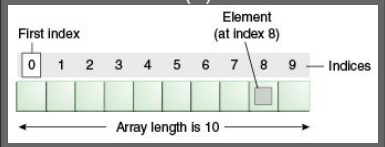
And many more! https://en.wikipedia.org/wiki/List_of_data_structures

Data Structure Toolbox: Fundamental Structures

Associative Containers

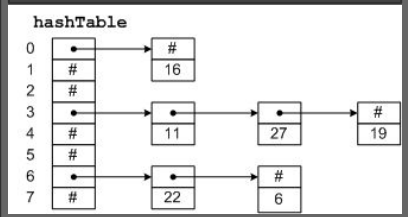
Arrays -

A contiguous block of memory, random access O(1)



Hashmap (chained implementation) -

Associative Data Structure with key/value pairs and a 'hash function'



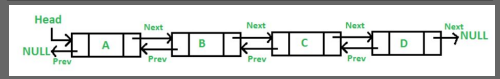
Sequence Containers

Linked Lists -

A 'chain' of nodes, can traverse in one direction

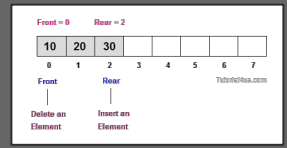
Doubly Linked Lists -

A 'chain' of nodes, can traverse in both directions



Queues -

A First in, First out data structure (FIFO)

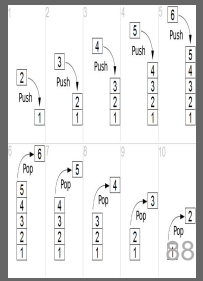


Priority Queues -

Uses a min-heap or max-heap and promotes min or max element to front of queue.

Stacks -

A Last in, last out data structure (LIFO)

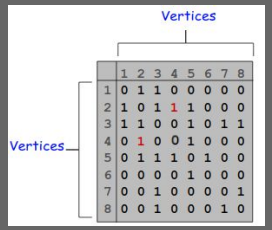


And many more! https://en.wikipedia.org/wiki/List_of_data_structures

Data Structure Toolbox: Tree and/or Graphs

Graph and Tree Data Structures

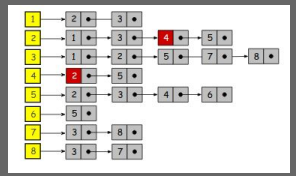
Adjacency Matrix -



2D array holding edge weights (or 0 if unconnected)

Space complexity of O(|V|*|V|)

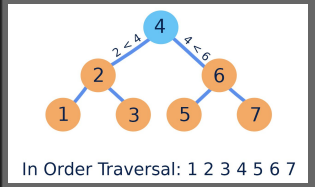
Adjacency List -



An array of lists or list of lists where each list indicates connectivity

Space complexity of O(|V|+|E|)

Binary Tree



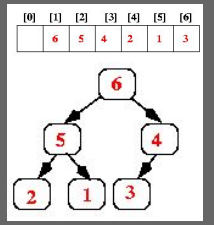
Data structure containing a left and right child

$\Theta(\log_2(n))$ for search, insertion, and deletion

Heaps

Binary Heaps

Array-based structure to hold a complete binary tree



Proof Toolbox

- Our tools so far!

Notation

```
∀n - "for all"
| - "such that"
n ∈ ℤ - "n is an element of the integers"
```

Proof Techniques

- Proof by Case
  - Enumerate or test all possible inputs
- Proof by Induction
  - Show that two cases hold
- Proof by Invariant
  - Step through 4 steps of algorithm
- Big-O Analysis
  - Prove run-time complexity

- Recurrence
  - Can be solved with Subsitution Method
- Recurrence Tree
  - "A Visual Proof" (Somewhat informal)
- Master Theorem
  - Proven by definition
- Substitution Method
  - (Works for any recurrence)

Building Blocks

Definition - Something given, we can assume is true

e.g. let x = 7

Proposition - A true or false statement

e.g.
1+7 = 7 FALSE
2+7 = 9 TRUE

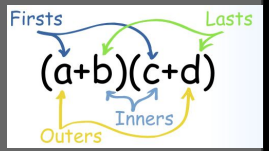
Predicate - A proposition whose truth depends on its input. It is a function that returns true or false.

"P(n) ::= "n is a perfect square"
P(4) thus is true, because 4 is a perfect square
P(3) is false, because it is not a perfect square.

Math Toolbox

Mathematics

Multiplication



$$(a+b)(c+d) = ac + ad + bc + bd$$

Logs

Logs -

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Usually we work in log base 2, i.e. $\log_2(n)$. The change of base formula is given below.

In this course we think about logs as 'halving' the number of our sub-problems (or search space).

Notation

Pi Production Notation

$$n! = \prod_{i=1}^{n} i.$$

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

Big-O: O(n) - "Worse Case Analysis or upper bound"
Big-Theta: Θ - "Average Case Analysis"
Big-Omega Ω - "Best Case or lower bound"

Factorial (!)

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Summation ("sigma")

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$$

i = index of summation
n = upper limit of summation

Good luck going forward, you can do it!