CSE 100:
TRIES
isOnBoard(G,v, word) (v is the vertex where the search starts, word is the word we are looking for)

Stack S := {} (start with an empty stack), pos := 0

for each vertex u, set visited[u] := false; push S, v;

while (S is not empty) do
    u := pop S;
    if u.revisit == true
        u.revisit := false
        u.visited := false;
        pos := pos - 1
    else if (NOT u.visited AND u.letter == word[pos])
        u.visited := true;
        pos := pos + 1
    if pos == length(word)
        return
    u.revisit := true;
    push S, u
    for each unvisited neighbour w of u
        push S, w;
end if
end while

END DFS()
Whole-key search

- Consider the usual find operation in a linked list
  - Nodes in the list hold keys
  - The key you are looking for is compared to the key in each node in sequence, until found or until reaching the end of the list
- Consider the usual find operation in a binary search tree
  - Nodes in the tree hold keys
  - The key you are looking for is compared to the key in the root node; depending on the result of the comparison, the search continues recursively in the left or right subtree, until found or reaching a null reference
- In each case, when a comparison is done, the entire search key is compared to the key in the current node. (This comparison is usually assumed to take $O(1)$ time...)  
  - For example, if keys are Strings, all the characters in the search key String may be compared to all the characters in the current node key String
  - ... or if keys are double precision floating point variables, all 64 bits in both keys are used for the comparisons
- Another approach is possible: each comparison only uses a small piece of the keys
- This leads to the idea of **radix search**
Radix search

- A key can be considered as a sequence of smaller segments
- Call each segment a *digit*
- Each digit can take on values from some set
- The size of this set of possible digit values is called the *base* or *radix*

**Examples:**
- An int can be considered as a sequence of 32 1-bit digits
  - The radix is 2: each digit can have one of two values
- An int can be considered as a sequence of 8 4-bit digits
  - The radix is 16: each digit can have one of 16 values
- A Java String can be considered as a sequence of 16-bit chars
  - The radix is 65536: each digit can have one of 65536 values

- The idea of radix search is: search is guided by comparisons that involve only one digit of the keys at a time
- This will have some advantages, but requires somewhat different data structures and algorithms: digital search trees, multiway tries, ternary tries
Digital search tree: example

- Consider the following 6 5-bit keys with values as shown:

  - A 00001
  - S 10011
  - E 00101
  - R 10010
  - C 00011
  - H 10100

- We will insert them in that order into an initially empty DST.
- (In this example, the 0th bit is the leftmost bit)
Digital search tree properties

- In a DST, each key is somewhere along the path specified by the bits in the key (this guarantees that the find and insert algorithms will work)

- Suppose keys each contain no more than B bits. Then:
  - The worst case height of a DST containing N keys is $\frac{B}{2}$
  - Compare: worst case height in a regular BST containing N keys is $N$, which with B bit keys can be as much as $2^B$

- So, when N is large and B is comparable to $\log_2 N$, DST’s give worst-case guarantees comparable to balanced BST’s, and are much easier to implement

- However, DST’s do not have a strong key ordering property:
  - The key in a node X can be larger or smaller than keys in either of its subtrees
  - So, an ordinary traversal of a DST is not guaranteed to visit keys in sorted order

- So consider tries:
  - Give strong key ordering property
  - Preserve the nice worst-case properties of DST’s
  - Do true radix search, avoiding repeated full-key comparisons
Two searches in PA4

Board

Dictionary

race
radiant
rain
ran
rat
rats

...
Two searches in PA4

Parallel search allows you to minimize time by ruling out unneeded paths. But how to represent the tree.
Tries: Efficient way to store/find keys that are sequences of digits

Create the code trie for the following code:

A: 00001
S: 10011
E: 00101
R: 10010
C: 0001
H: 101
W: 00
Does the structure of the trie depend on the order in which keys are inserted?

A. Yes  B. No
Multi-way tries: Efficient finding of keys by their sequence

Build the trie which holds the following number keys:

Assuming your trie could potentially hold any decimal number, how many children does each node (potentially) have?

A. 2  B. 8  C. 10  D. Other
Multi-way tries: Efficient finding of keys by their sequence

Build the trie to store the following numbers:

Which node in the trie represents 1234?

A. Red
B. Purple
C. Blue
D. Green
E. Other
Multi-way tries: Efficient finding of keys by their sequence

Build the trie to store the following numbers:

Is there a way to find whether all keys contained in a sequence of digits are present in the trie?

A. Yes
B. No
Properties of tries

Build the trie to store the following numbers:

Is there a “strong ordering” property in a trie? That is, are smaller keys always to the left of larger keys?
A. Yes  B. No
Properties of tries

Build the trie to store the following numbers:

Suppose your keys are a sequence of at most D digits, and N is the maximum number of keys you will store. What is the worst case height of this trie (i.e., for large N)?

A. D  B. lg(D)  C. N  D. DxN  E. Other
Properties of tries

Build the trie to store the following numbers:

If you stored the same $N$ $D$-digit keys in a Binary Search Tree, what would be the worst case height of the tree?

A. $N$  B. $\lg(10^D)$  C. $\lg(N)$  D. $\lg(D)$  E. Other
Properties of tries

Build the trie to store the following numbers:

Consider storing the full $10^D$ keys. We know that on average the height of a BST will be $\log(10^D)$. Which is smaller: $D$ or $\log(10^D)$?

A. $D$  B. $\log(10^D)$  C. They are the same
Properties of tries

Build the trie to store the following numbers:

8
1234
59
123
8775
80

So what is the main drawback of tries?
A. They are difficult to implement
B. They (usually) waste a lot of space
C. They are slow
D. There is no drawback of tries
Ternary search trees to the rescue!

- Tries combine binary search trees with tries.
- Each node contains the following:
  - A key digit for search comparison
  - Three pointers:
    - left and right: for when the digit being considered is less than and greater than (respectively) the digit stored in the node (the BST part)
    - middle: for when the digit being considered is equal to the digit stored in the node (the trie part)
  - An end bit to indicate we’ve completed a key stored in the tree.
List all the words (strings) you can find in this TST

Are the following in the tree? (A=yes, B=no)
- get
- if
- gif
- its
- gacar
- tsem
- tis
- cag
Draw the ternary tree for the following (in this order)

i
just
met
this
is
crazy
call
me
maybe

Does the structure of the tree depend on the order in which keys were inserted? A. Yes  B. No
Algorithms for insert and find (in TSTs and MWTs)

• In your reading and/or in Paul Kube’s slides