Using Nested ADTs: Breadth-First Search

Preparing you for Assignment 2!

Today's question

How can we use abstractions (ADTs) to solve problems?

Nested Data Structures

Nested Data Structures

- Nesting data structures (using one ADTs as the data type inside of another ADT) is a great way of organizing data with complex structure.

Nested Data Structures

- Nesting data structures (using one ADTs as the data type inside of another ADT) is a great way of organizing data with complex structure.

- You will thoroughly explore nested data structures (specifically nested Sets and Maps) in Assignment 2!

An example

- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo.

An example

- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo.

- Requirements: We need to be able to quickly look up the feeding times associated with an animal if we know it's name. We need to be able to store multiple feeding times for each animal. The feeding times should be stored in the order in which the feedings should happen.

An example

- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo

- Requirements: We need to be able to quickly look up the feeding times associated with an animal if we know it's name. We need to be able to store multiple feeding times for each animal. The feeding times should be stored in the order in which the feedings should happen.

- Data Structure Declaration
  - `Map<string, Vector<string>>`

An example

- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo

- Requirements: We need to be able to quickly look up the feeding times associated with an animal if we know it's name. We need to be able to store multiple feeding times for each animal. The feeding times should be stored in the order in which the feedings should happen.

- Data Structure Declaration
  - `Map<string, Vector<string>>`

Quick lookup by animal name

An example

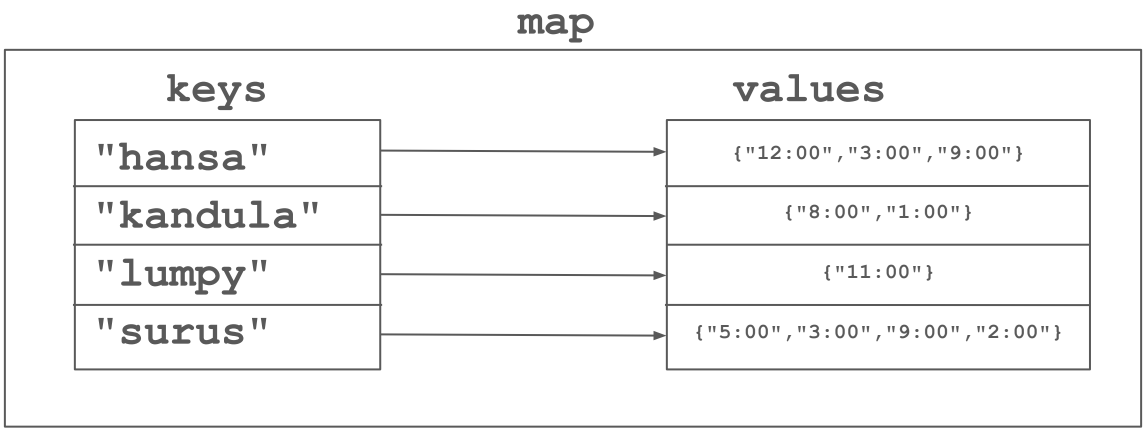
- Imagine we are designing a system to keep track of feeding times for the different animals at a zoo

- Requirements: We need to be able to quickly look up the feeding times associated with an animal if we know it's name. We need to be able to store multiple feeding times for each animal. The feeding times should be stored in the order in which the feedings should happen.

- Data Structure Declaration
    - ```
      Map<string, Vector<string>>
      ```

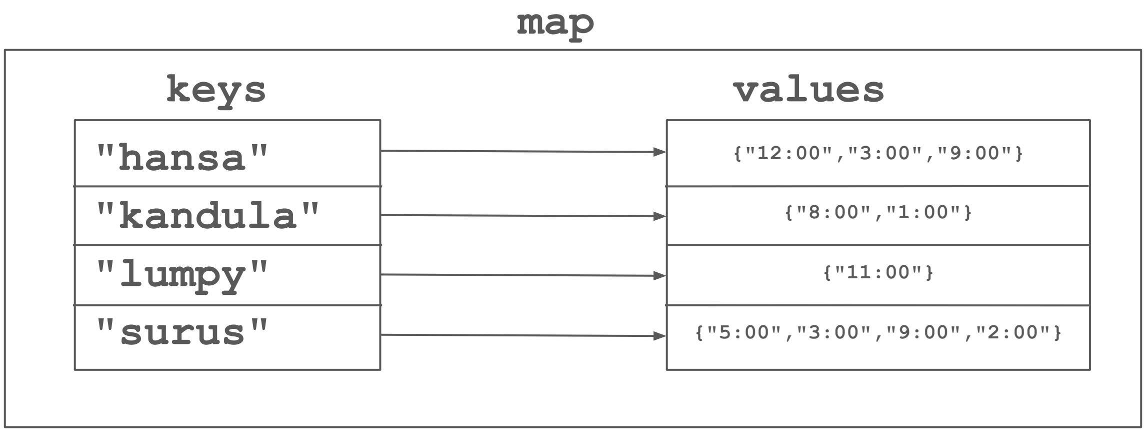
Store multiple, ordered feeding times per animal
      ```

Nested Data Structures Example



Wonderful diagram and animal naming borrowed from Sonja Johnson-Yu

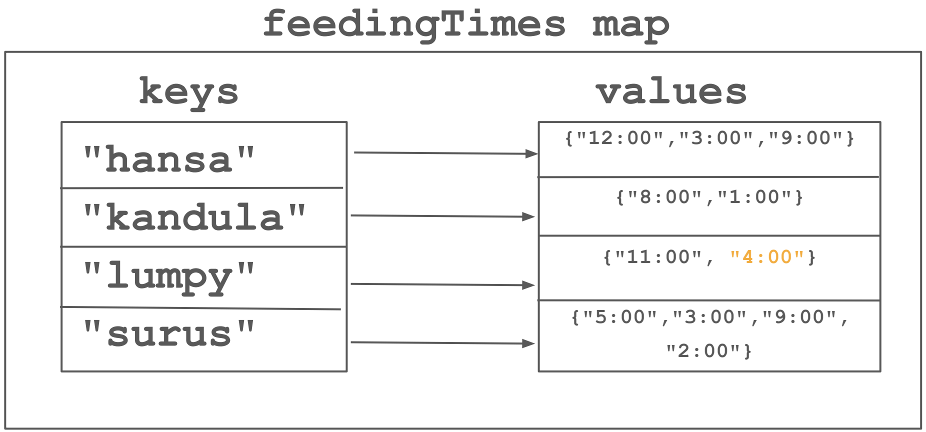
Nested Data Structures Example



How do we use modify the internal values of this map?

Nested Data Structures Example

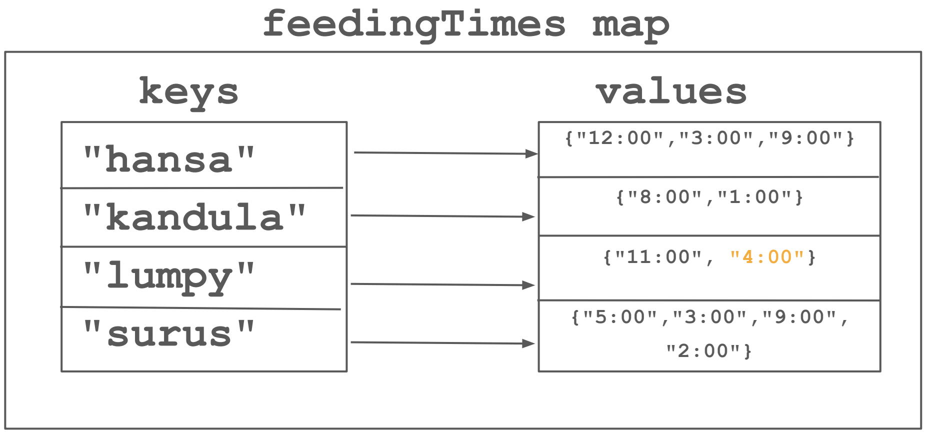
Goal: We want to add a second feeding time of 4:00 for "lumpy".



Nested Data Structures Example

Goal: We want to add a second feeding time of 4:00 for "lumpy".

Which of the following three snippets of code will correctly update the state of the map?



1.
```
feedingTimes["lumpy"].add("4:00");
```

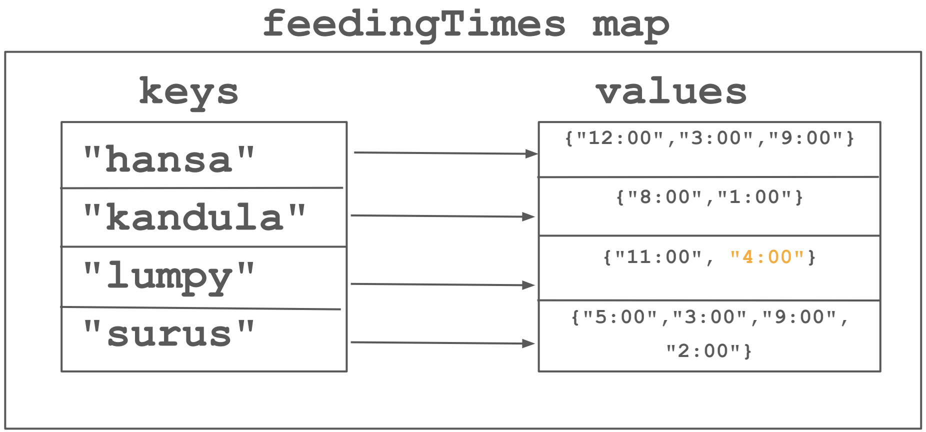
2.
```
Vector<string> times = feedingTimes["lumpy"];
times.add("4:00");
```

3.
```
Vector<string> times = feedingTimes["lumpy"];
times.add("4:00");
feedingTimes["lumpy"] = times;
```

Nested Data Structures Example

Goal: We want to add a second feeding time of 4:00 for "lumpy".

Which of the following three snippets of code will correctly update the state of the map?




```
1.  feedingTimes["lumpy"].add("4:00");

2.  Vector<string> times = feedingTimes["lumpy"];
    times.add("4:00");

3.  Vector<string> times = feedingTimes["lumpy"];
    times.add("4:00");
    feedingTimes["lumpy"] = times;
```

`[]` Operator and `=` Operator Nuances

- When you use the [] operator to access an element from a map, you get a reference to the map, which means that any changes you make to the reference will be persistent in the map.

```
feedingTimes["lumpy"].add("4:00");
```

`[]` Operator and `=` Operator Nuances

- When you use the [] operator to access an element from a map, you get a reference to the map, which means that any changes you make to the reference will be persistent in the map.

- However, when you use the = operator to assign the result of the [] operator to a variable, you get a copy of the internal data structure.

```
// makes and modifies a copy, not the actual map value:
Vector<string> times = feedingTimes["lumpy"];
times.add("4:00");
```

`[]` Operator and `=` Operator Nuances

- When you use the [] operator to access an element from a map, you get a reference to the map, which means that any changes you make to the reference will be persistent in the map.

- However, when you use the = operator to assign the result of the [] operator to a variable, you get a copy of the internal data structure.

- If you choose to store the internal data structure in an intermediate variable, you must do an explicit reassignment to get your changes to persist.

```
// would store the modified `times` copy in the map
feedingTimes["lumpy"] = times;
```

Using Nested ADTs

- Powerful
  - Can express highly structured and complex data
  - Used in many real-world systems
- Tricky
  - With increased complexity comes increased cognitive load in differentiating the information stored at each level of the nesting.
  - Specifically in C++, working with nested data structures can be tricky due the use of references and copies. Follow the correct paradigms to stay on track!

Examples of interesting problems to solve using ADTs

- Simulate potential impacts of flooding on a topographical landscape (how does water flow outwards from a source and settle into the surrounding areas)
- Generate simulated text in the style of a certain author. Similarly, do textual analysis to determine who the author of a provided piece of text was.
- Spell check and autocomplete for a word document editor
- Manage information about the natural landmarks and state parks in California to help tourists plan their trip to the state
- Develop a ticketing management system for Stanford Stadium
- Aggregate and analyze reviews for an online shopping website
- Solve fun puzzles

Examples of interesting problems to solve using ADTs

- Simulate potential impacts of flooding on a topographical landscape (how does water flow outwards from a source and settle into the surrounding areas)
- Generate simulated text in the style of a certain author. Similarly, do textual analysis to determine who the author of a provided piece of text was.
- Spell check and autocomplete for a word document editor
- Manage information about the natural landmarks and state parks in California to help tourists plan their trip to the state
- Develop a ticketing management system for Stanford Stadium
- Aggregate and analyze reviews for an online shopping website
- **Solve fun puzzles**

Word Ladders

Word Ladders

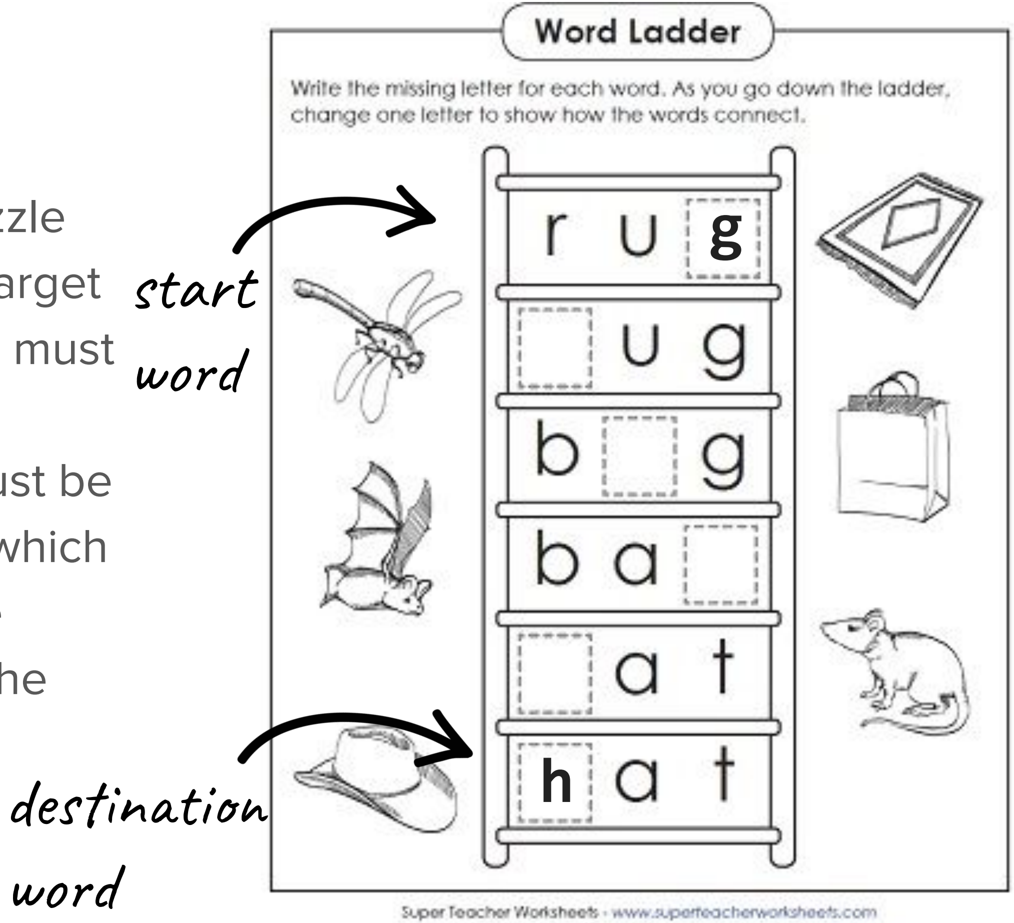
- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.

Word Ladders

- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.
- A common tool for teaching kids English vocabulary!

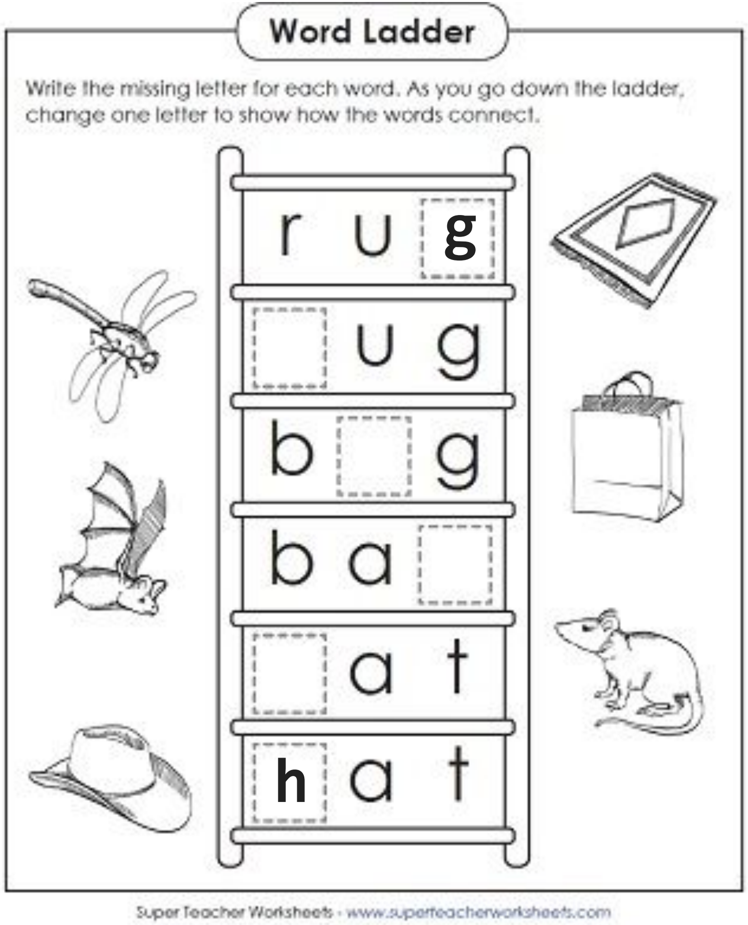
Word Ladders

- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.
- A common tool for teaching kids English vocabulary!



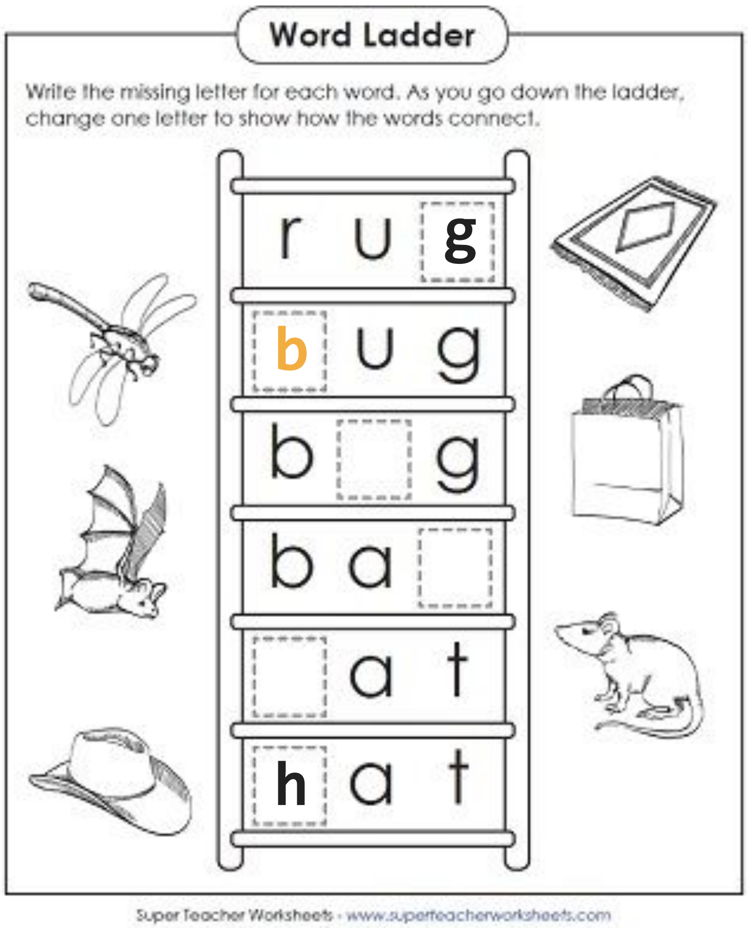
Word Ladders

- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.
- A common tool for teaching kids English vocabulary!



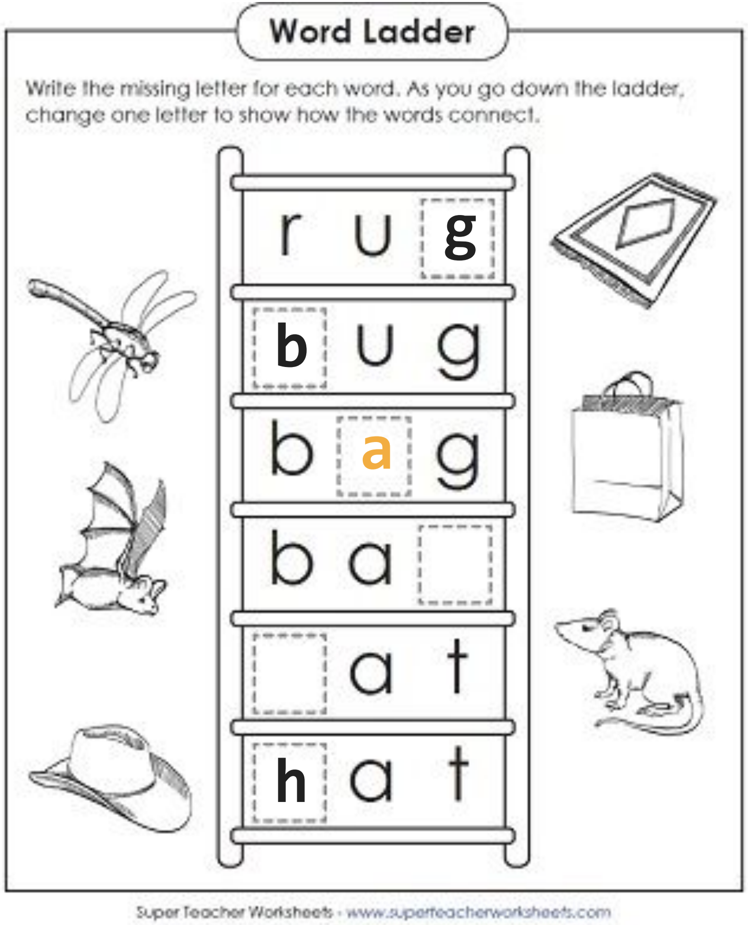
Word Ladders

- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.
- A common tool for teaching kids English vocabulary!



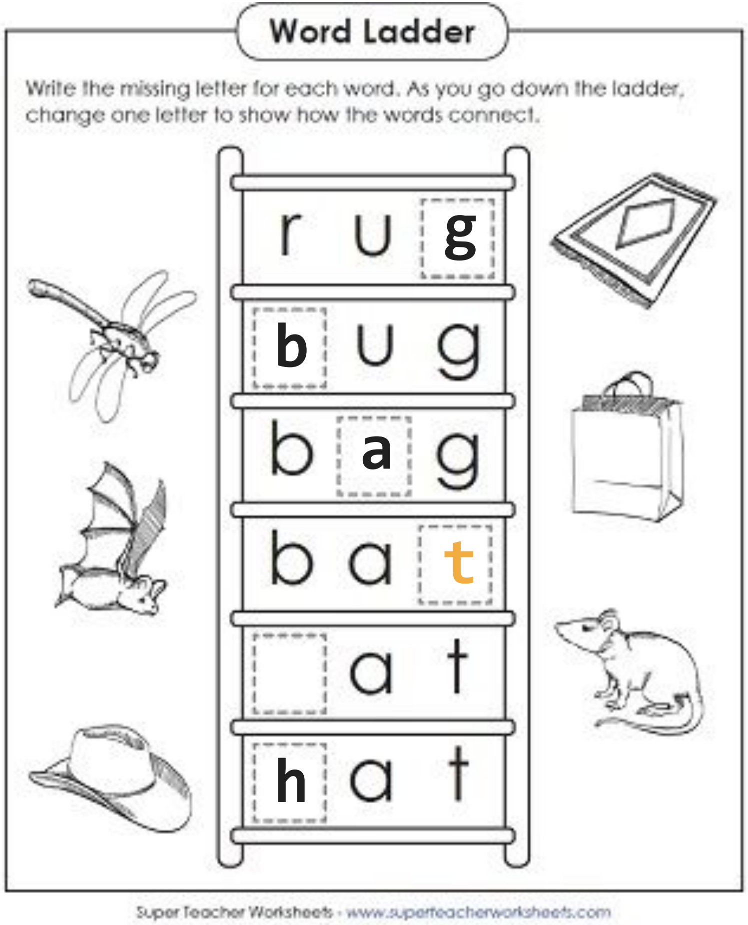
Word Ladders

- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.
- A common tool for teaching kids English vocabulary!



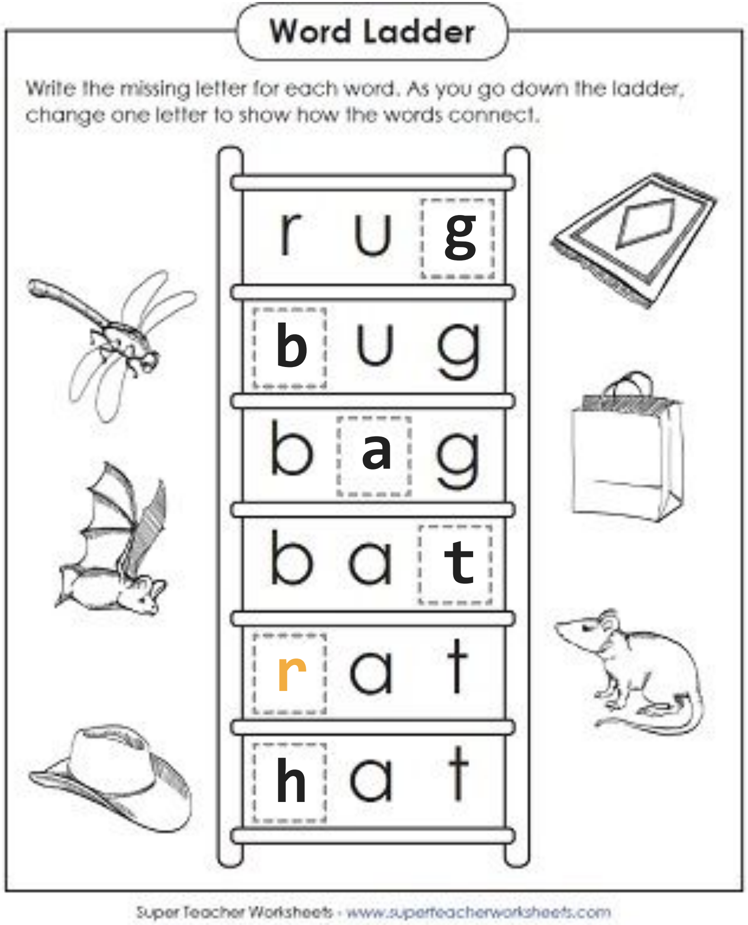
Word Ladders

- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.
- A common tool for teaching kids English vocabulary!



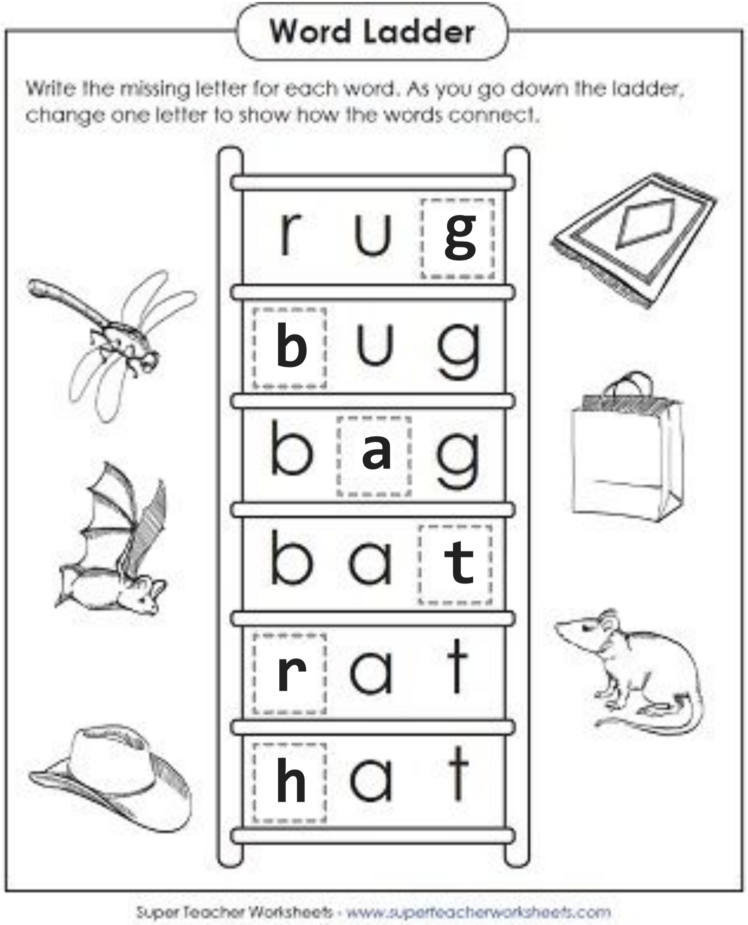
Word Ladders

- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.
- A common tool for teaching kids English vocabulary!



Word Ladders

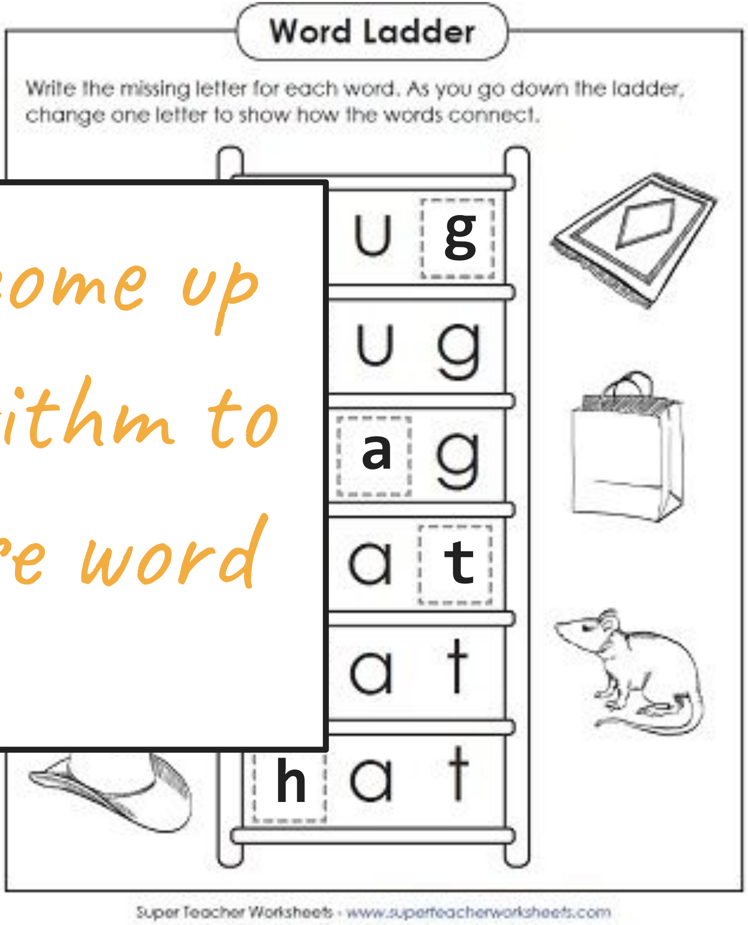
- A word ladder is a type of puzzle based on a start word and a target word. To solve the puzzle you must generate a sequence of intermediate words (which must be valid English words), each of which is one letter different from the previous one, that gets from the start word to the target word.
- A common tool for teaching kids English vocabulary!



Word Ladders

- A word ladder is
  based on a start
  word. To solve th
  generate a sequ
  intermediate wo
  valid English wo
  is one letter diffe
  previous one, th
  start word to the
- A common tool for teaching kids English vocabulary!

How can we come up with an algorithm to generate these word ladders?



Word Ladder Generation First Attempt

- Given a start word and a target word, a natural place to start would be to model how a human might attempt to solve this problem

Word Ladder Generation First Attempt

- Given a start word and a target word, a natural place to start would be to model how a human might attempt to solve this problem
  - Start at the start word
  - Make an educated guess about what letter to change first
  - Modify that letter to get to a new English word
  - From there, make another educated guess about which letter to change and modify that letter
  - Keep repeating this process until you reach the target word (unlikely) or hit a dead end (likely)
  - If you hit a dead end, start over again, taking a different first step

Word Ladder Generation First Attempt

- Given a start word and a target word, a natural place to start would be to model how a human might attempt to solve this problem
  - Start at the start word
  - Make an educated guess about what letter to change first
  - Modify that letter to get to a new English word
  - From there, make another educated guess about which letter to change and modify that letter
  - Keep repeating this process until you reach the target word (unlikely) or hit a dead end (likely)
  - If you hit a dead end, start over again, taking a different first step
- What are the issues with this approach?
  - Requires intuition – does a computer have intuition?
  - Unorganized – no organized strategy for the exploration
  - No guarantee that you'll ever find a solution!

Breadth-First Search

Breadth-First Search

- We need a structured way to explore words that are "adjacent" to one another (one letter difference between the two of them)

Breadth-First Search

- We need a structured way to explore words that are "adjacent" to one another (one letter difference between the two of them)
- What's the simplest possible word ladder we could find?
    - If the words are only one letter different from one another (pig and fig), then finding the word ladder is relatively easy – we look at all words that are one letter away from the current word

Breadth-First Search

- We need a structured way to explore words that are "adjacent" to one another (one letter difference between the two of them)
- What's the simplest possible word ladder we could find?
  - If the words are only one letter different from one another (pig and fig), then finding the word ladder is relatively easy – we look at all words that are one letter away from the current word
- What's the next simplest possible word ladder we could find?
  - If the word ladder requires two steps, then we can break down the problem into the problem of exploring one step away from all the words that are one step away from the starting word

Breadth-First Search

- We need a structured way to explore words that are "adjacent" to one another (one letter difference between the two of them)
- What's the simplest possible word ladder we could find?
  - If the words are only one letter different from one another (pig and fig), then finding the word ladder is relatively easy – we look at all words that are one letter away from the current word
- What's the next simplest possible word ladder we could find?
  - If the word ladder requires two steps, then we can break down the problem into the problem of exploring one step away from all the words that are one step away from the starting word
- **Important observation: In order to keep our search organized, we first explore all word ladders of "length" 1 before we explore any word ladders of "length" 2, and so on.**

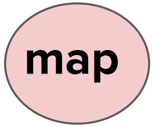
BFS Example

Breadth-First Search Example

- Let's try to apply this approach to find a word ladder starting at the word "map" and ending at the word "way"

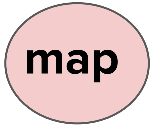
Breadth-First Search Example

start: map
destination: way



Breadth-First Search Example

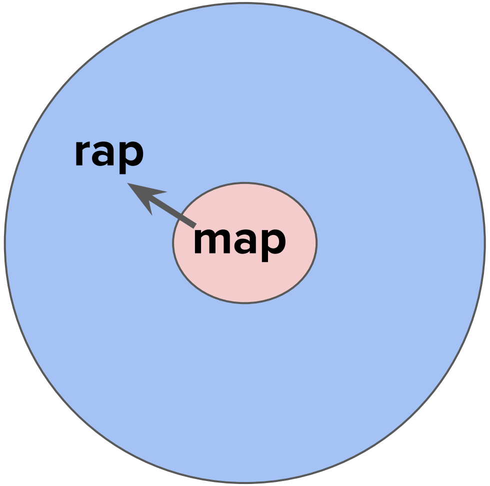
start: map
destination: way



0 steps away

Breadth-First Search Example

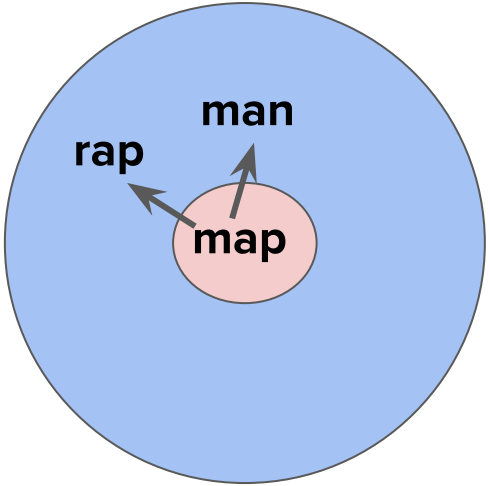
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

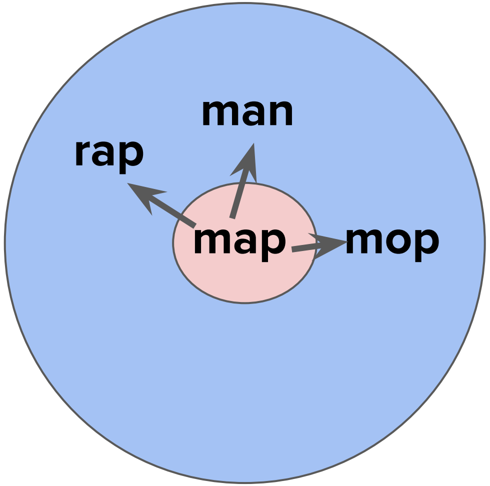
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

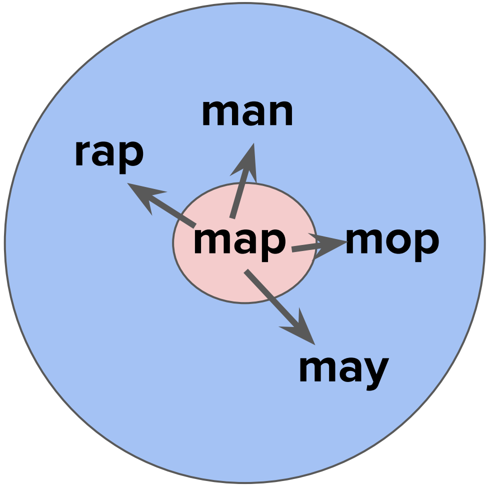
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

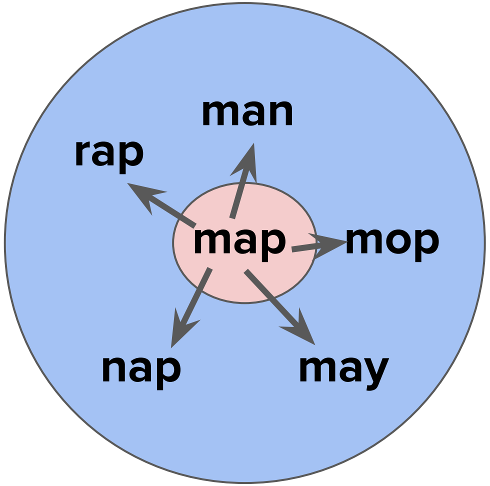
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

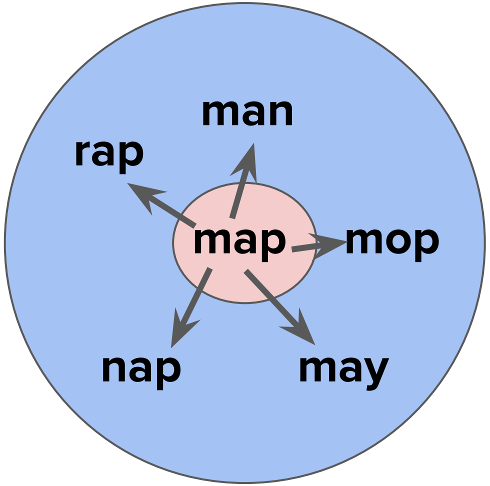
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

start: map
destination: way

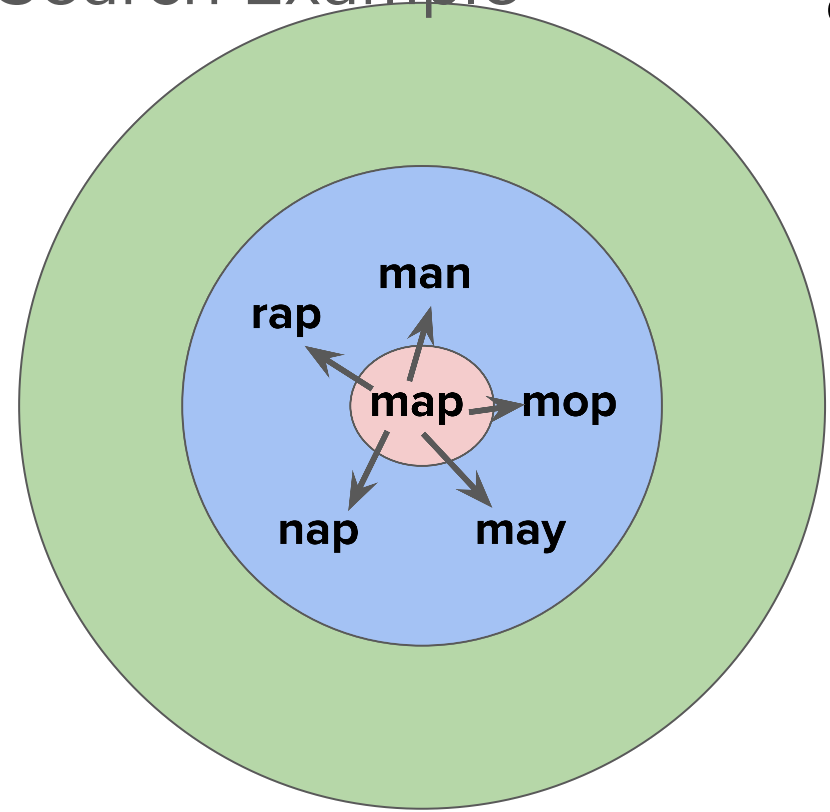


0 steps away
1 step away

Note: For the sake of brevity/demonstration, we will not enumerate all possible words that are 1 step away

Breadth-First Search Example

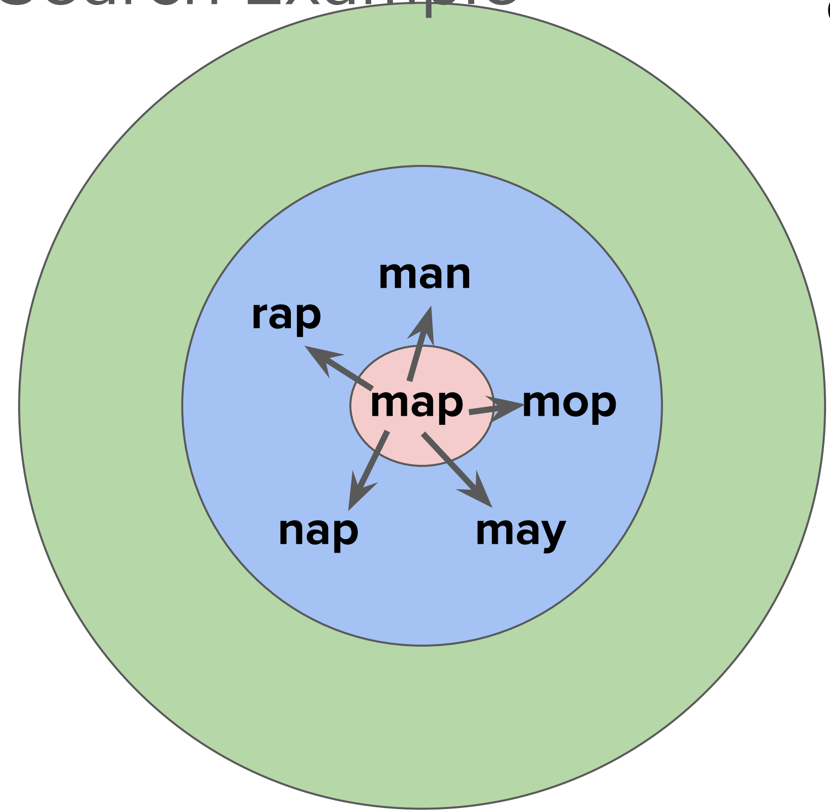
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

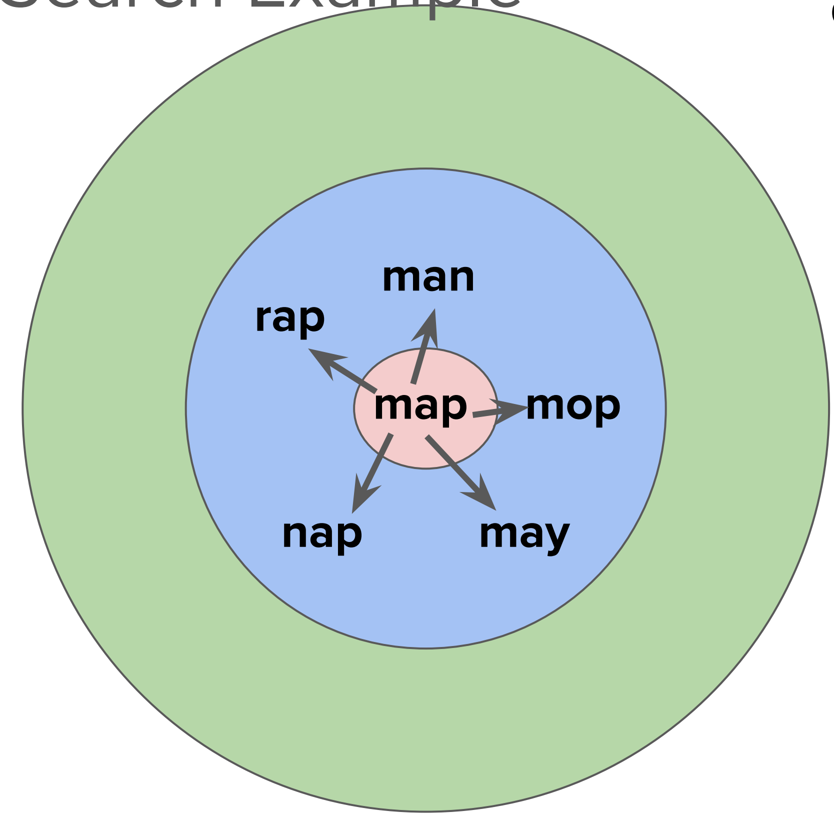
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

start: map
destination: way

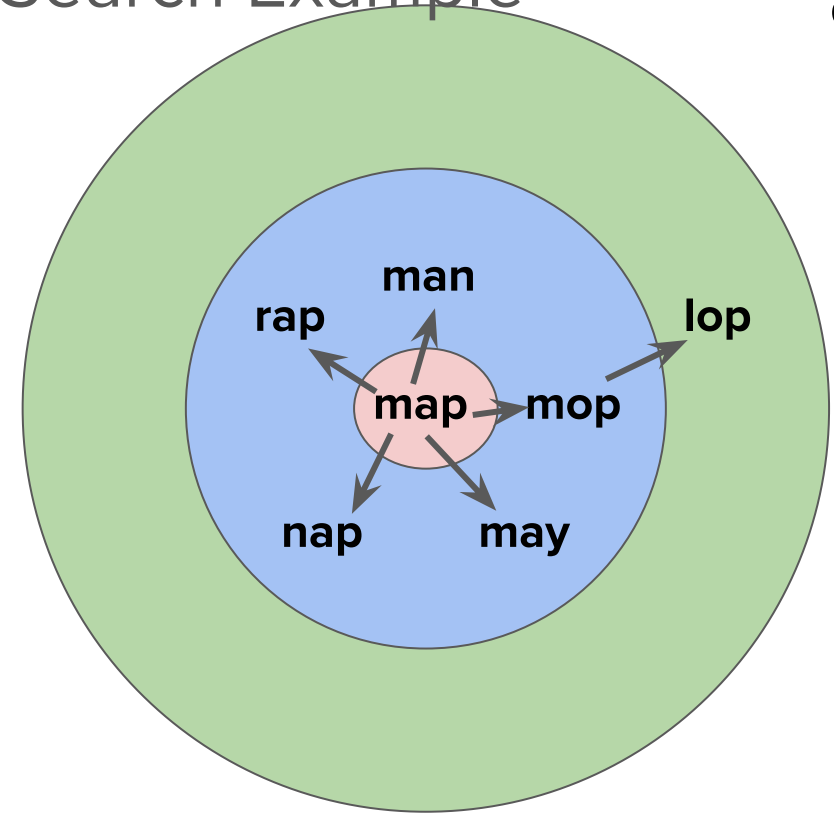


Observation: 2 steps away from "map" is really just 1 step away from any of its neighbors

0 steps away
1 step away
2 steps away

Breadth-First Search Example

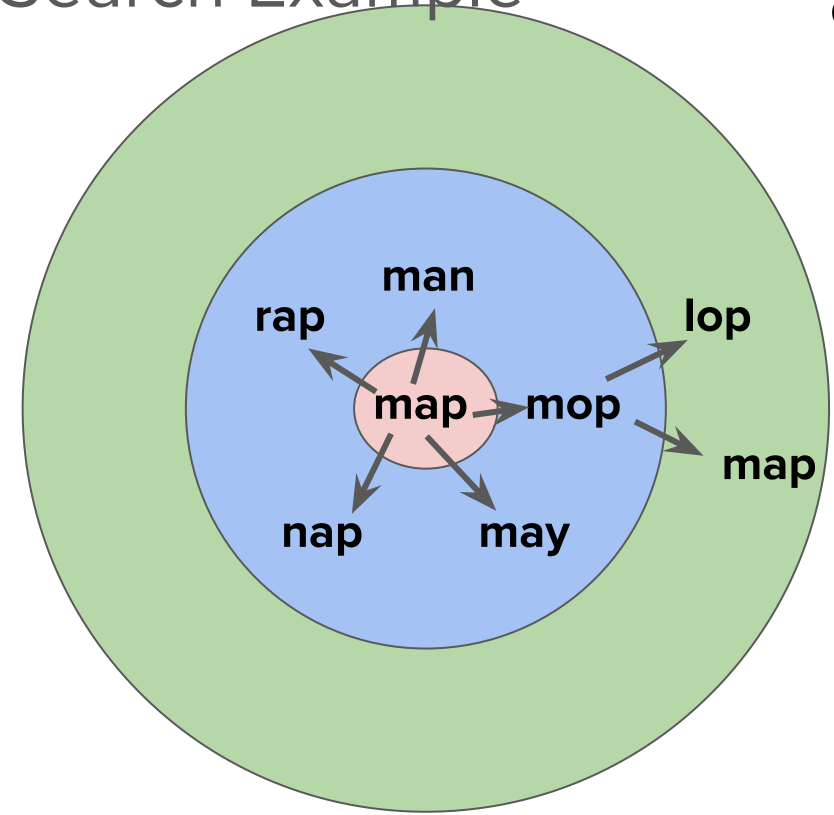
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

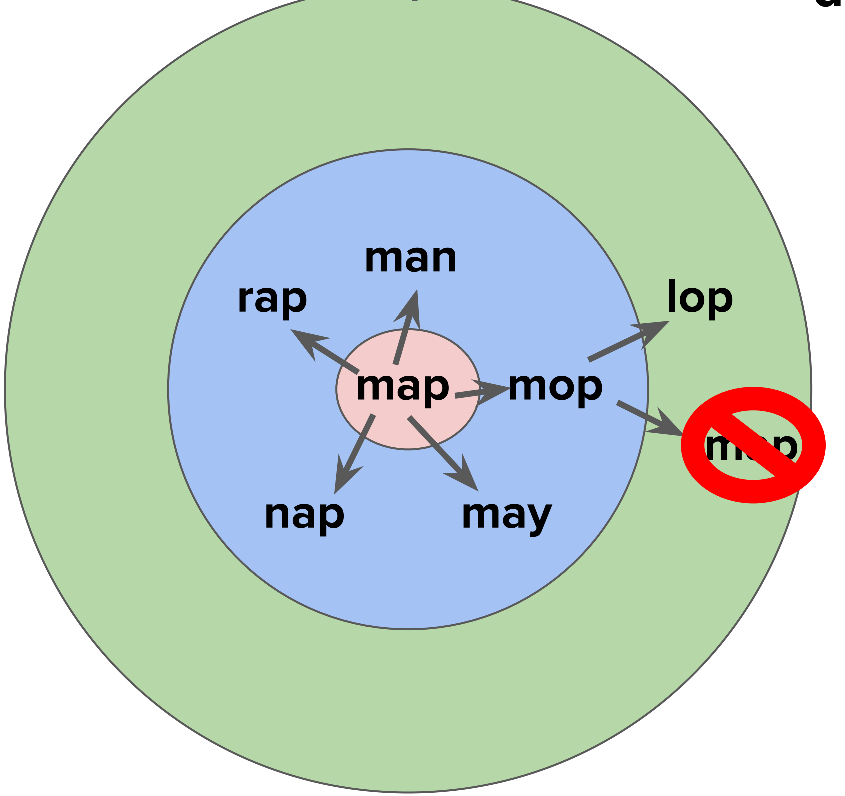
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

start: map
destination: way

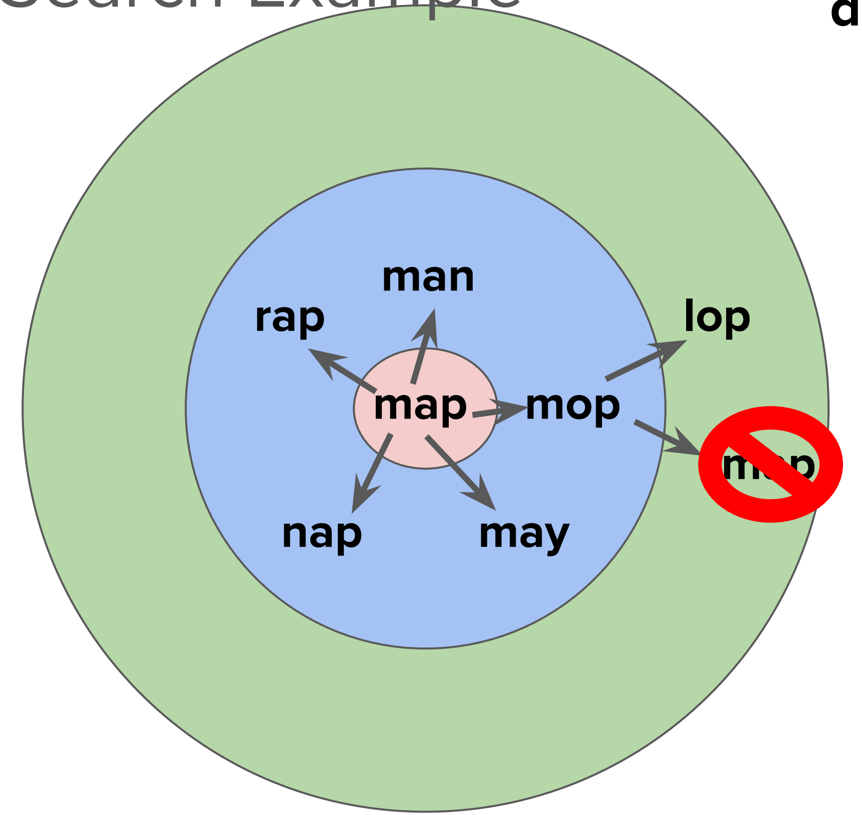


Visiting a word we've already been at before is basically like going backwards in our search. We want to avoid this at all costs!

0 steps away
1 step away
2 steps away

Breadth-First Search Example

start: map
destination: way

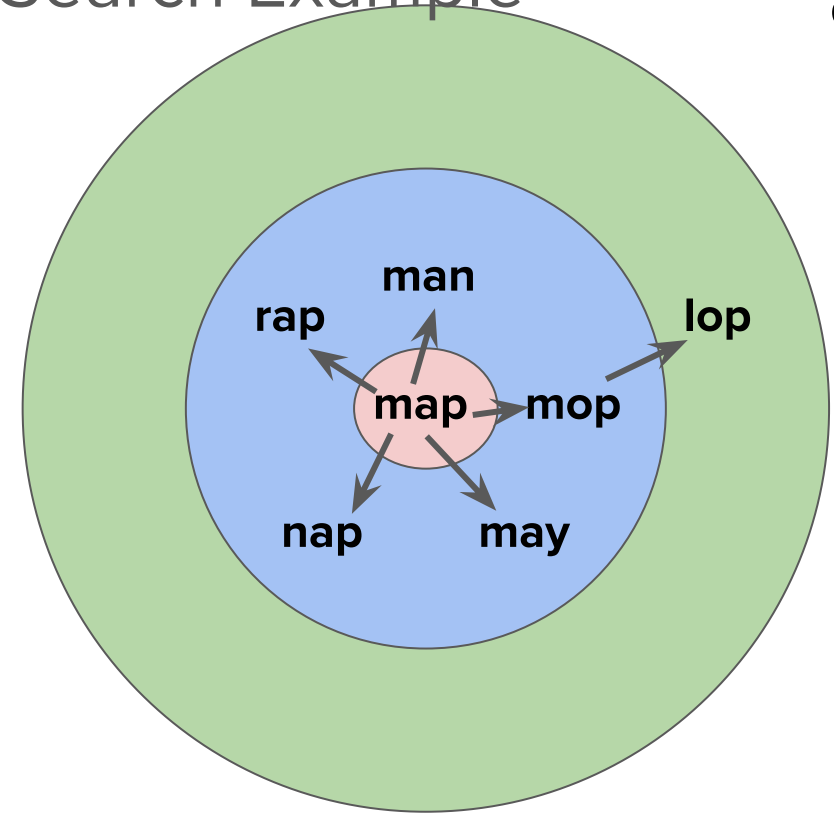


Idea: Keep track of a collection of visited words, and don't double visit

0 steps away
1 step away
2 steps away

Breadth-First Search Example

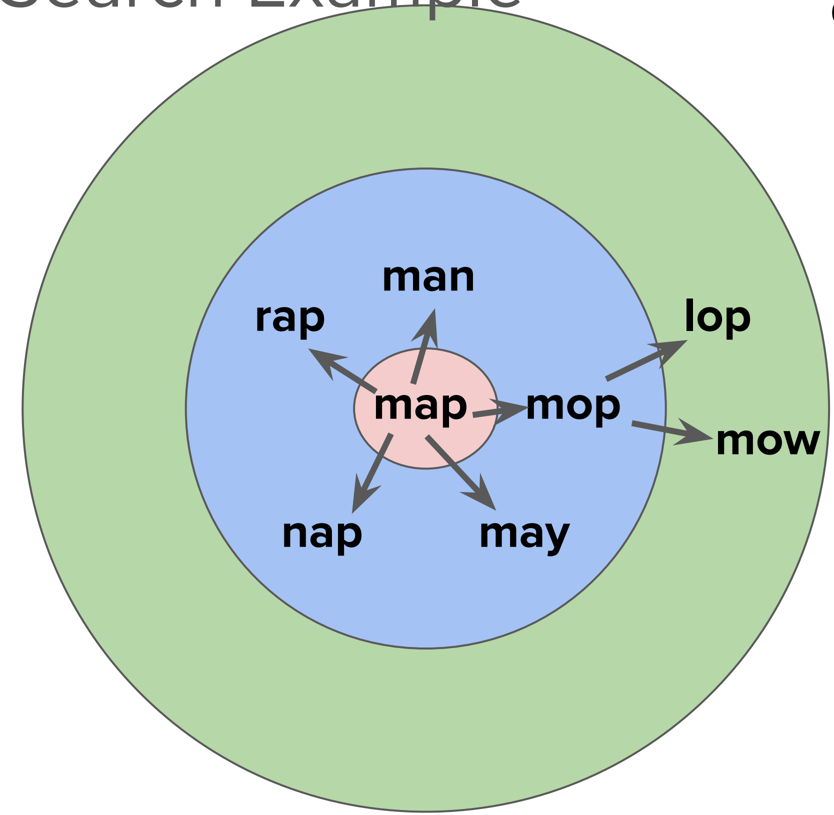
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

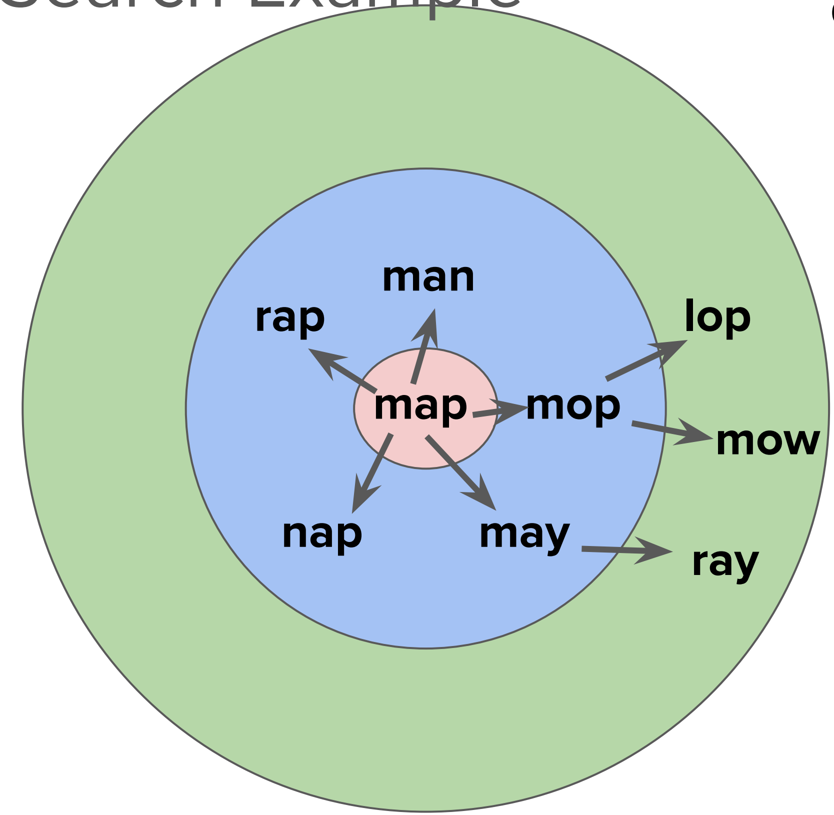
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

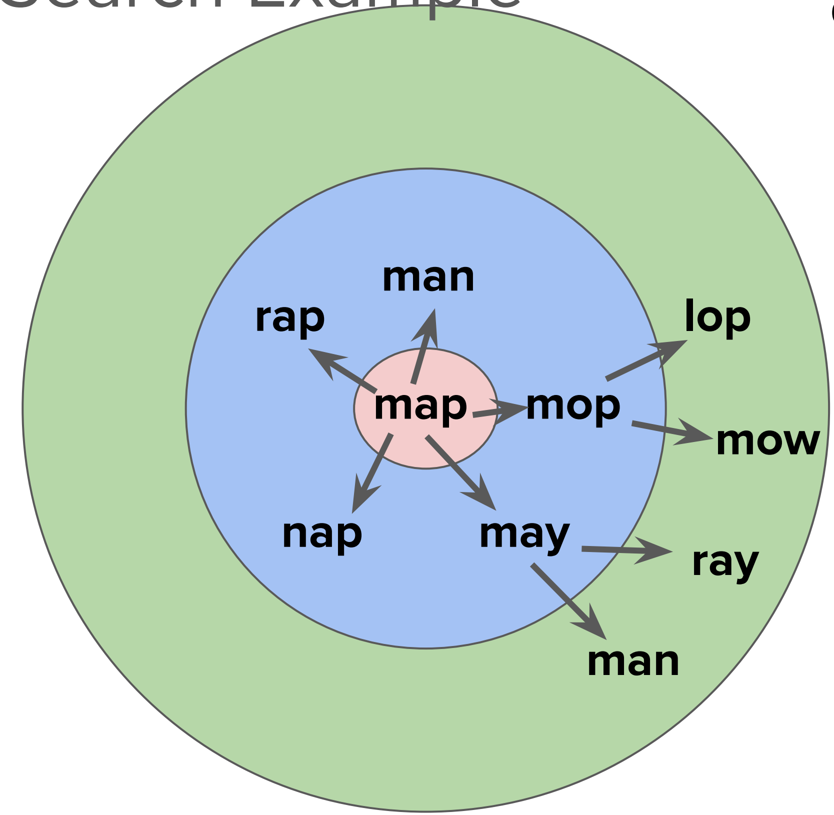
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

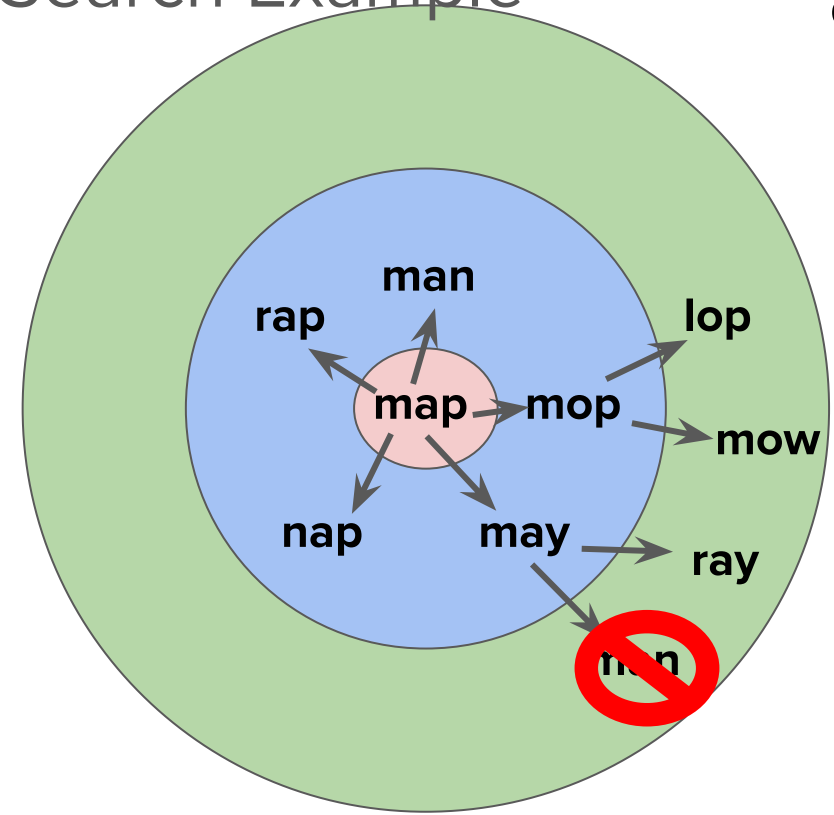
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

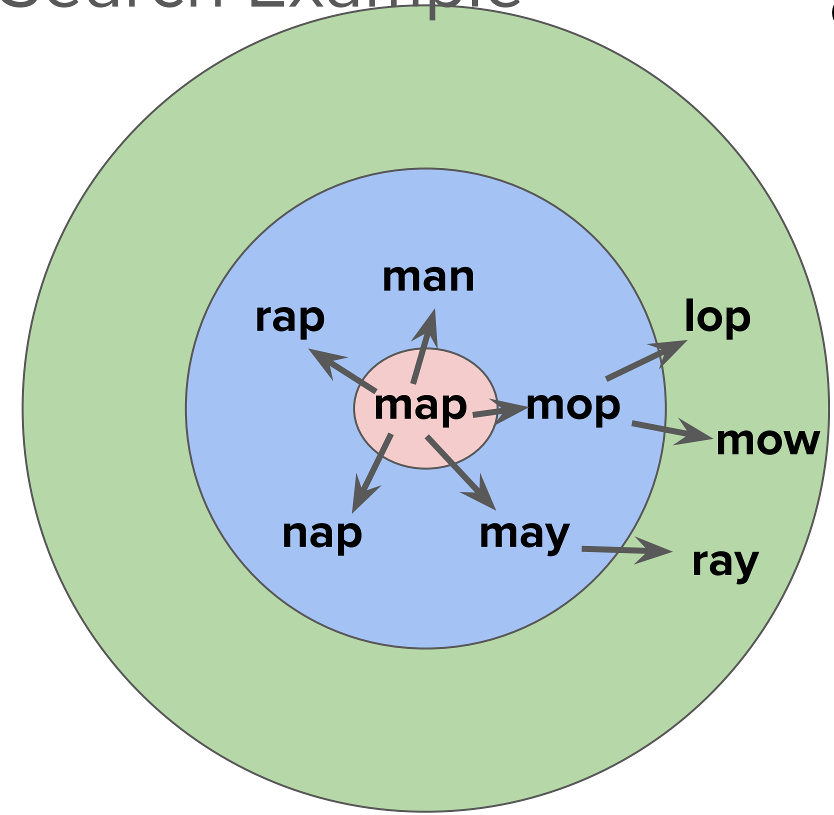
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

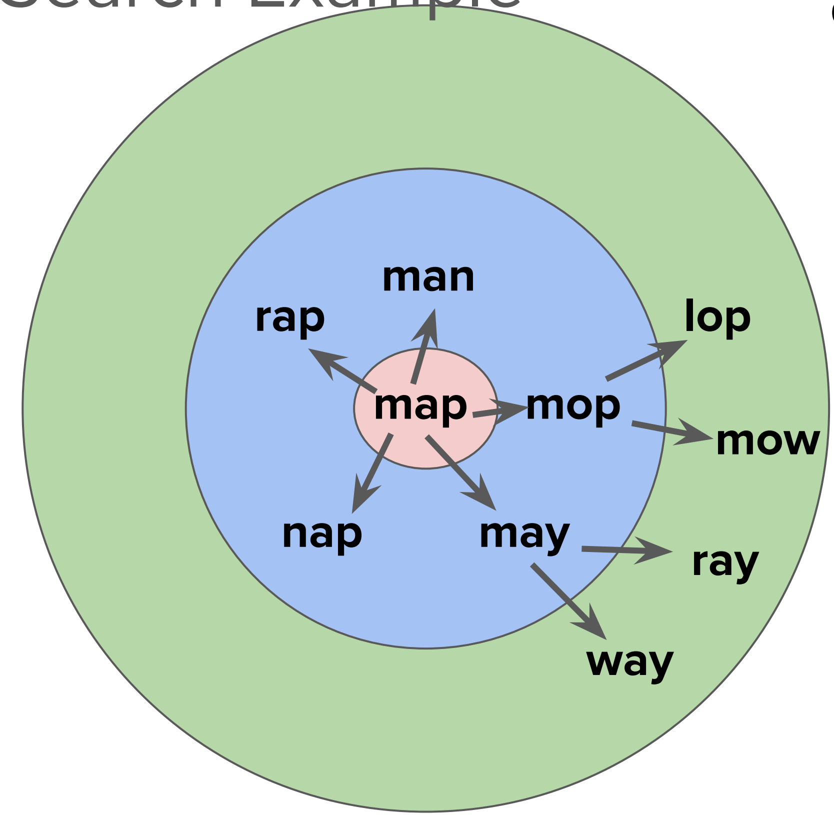
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

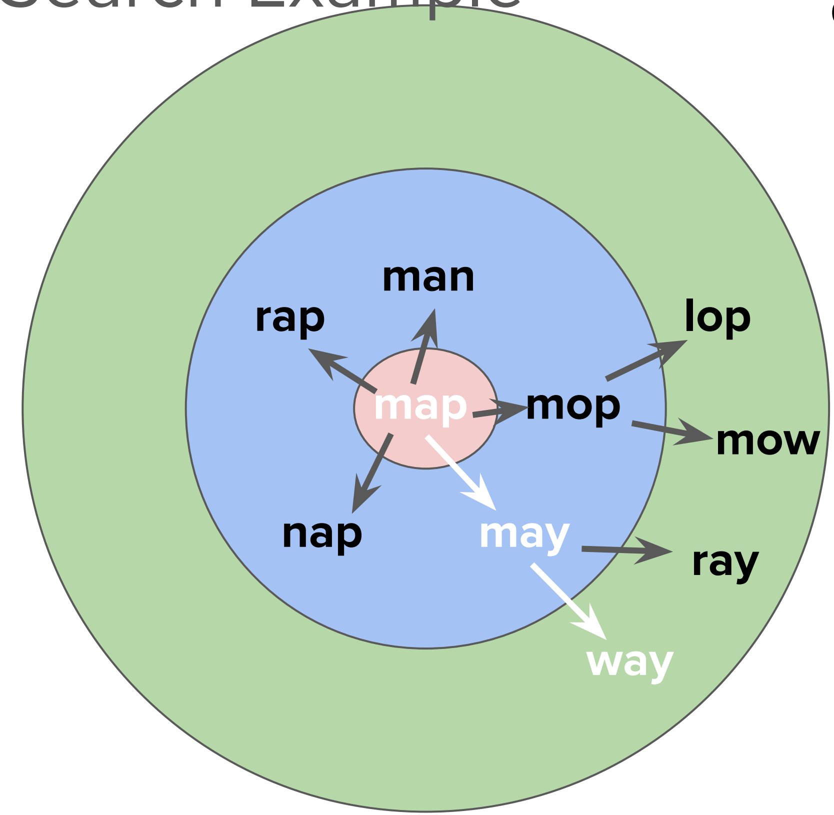
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

start: map
destination: way



0 steps away
1 step away
2 steps away

Success! We have found a valid word ladder
map -> may -> way

Formalizing BFS

Breadth-First Search Data Structures

We need...

- A data structure to represent (partial word) ladders
  - Desired characteristics: We should be able to easily access the most recent word added to the word ladder

Breadth-First Search Data Structures

We need...

- A data structure to represent (partial word) ladders
  - Desired characteristics: We should be able to easily access the most recent word added to the word ladder
- A data structure to store all the partial word ladders that we have generated so far and have yet to explore
  - Desired characteristics: We want to maintain an ordering of ladders such that all ladders of a certain length get explored before ladders of longer length get explored

Breadth-First Search Data Structures

We need...

- A data structure to represent (partial word) ladders
  - Desired characteristics: We should be able to easily access the most recent word added to the word ladder
- A data structure to store all the partial word ladders that we have generated so far and have yet to explore
  - Desired characteristics: We want to maintain an ordering of ladders such that all ladders of a certain length get explored before ladders of longer length get explored
- A data structure to keep track of all the words that we've explored so far, so that we avoid getting stuck in loops
  - Desired characteristics: We want to be able to quickly decide whether or not a word has been seen before.

Breadth-First Search Data Structures

We need...

What data structures should we use for each of these components?

- A data structure
  - Desired charac... recent word added to the word ladder
- A data structure ...e have generated so far and have yet...
  - Desired charac... uch that all ladders of a certain length g... red
- A data structure ... xplored so far, so that we avoid getting stuck in loops
  - Desired characteristics: We want to be able to quickly decide whether or not a word has been seen before.

Breadth-First Search Data Structures

We need...

- A data structure to represent (partial word) ladders
  - **`Stack<string>`**
- A data structure to store all the partial word ladders that we have generated so far and have yet to explore
  - Desired characteristics: We want to maintain an ordering of ladders such that all ladders of a certain length get explored before ladders of longer length get explored
- A data structure to keep track of all the words that we've explored so far, so that we avoid getting stuck in loops
  - Desired characteristics: We want to be able to quickly decide whether or not a word has been seen before.

Breadth-First Search Data Structures

We need…

- A data structure to represent (partial word) ladders
  - **`Stack<string>`**
- A data structure to store all the partial word ladders that we have generated so far and have yet to explore
  - **`Queue<Stack<string>>`**
- A data structure to keep track of all the words that we've explored so far, so that we avoid getting stuck in loops
  - Desired characteristics: We want to be able to quickly decide whether or not a word has been seen before.

Breadth-First Search Data Structures

We need...

- A data structure to represent (partial word) ladders
  - **`Stack<string>`**
- A data structure to store all the partial word ladders that we have generated so far and have yet to explore
  - **`Queue<Stack<string>>`**
- A data structure to keep track of all the words that we've explored so far, so that we avoid getting stuck in loops
  - **`Set<string>`**

Breadth-First Search Pseudocode

Breadth-First Search Pseudocode

```
Create an empty queue and an empty set of visited locations
Create an initial word ladder containing the starting word and add it to the
queue
```

Breadth-First Search Pseudocode

```
Create an empty queue and an empty set of visited locations
Create an initial word ladder containing the starting word and add it to the queue
While the queue is not empty
```

Breadth-First Search Pseudocode

```
Create an empty queue and an empty set of visited locations
Create an initial word ladder containing the starting word and add it to the queue
While the queue is not empty
    Remove the next partial ladder from the queue
    Set the current search word to be the word at the top of the ladder
    If the current word is the destination, then return the current ladder
```

Breadth-First Search Pseudocode

```
Create an empty queue and an empty set of visited locations
Create an initial word ladder containing the starting word and add it to the queue
While the queue is not empty
    Remove the next partial ladder from the queue
    Set the current search word to be the word at the top of the ladder
    If the current word is the destination, then return the current ladder
    Generate all "neighboring" words that are valid English words and one
    letter away from the current word
    Loop over all neighbor words
```

Breadth-First Search Pseudocode

```
Create an empty queue and an empty set of visited locations
Create an initial word ladder containing the starting word and add it to the
queue
While the queue is not empty
    Remove the next partial ladder from the queue
    Set the current search word to be the word at the top of the ladder
    If the current word is the destination, then return the current ladder
    Generate all "neighboring" words that are valid English words and one
    letter away from the current word
    Loop over all neighbor words
        If the neighbor word hasn't yet been visited
```

Breadth-First Search Pseudocode

```
Create an empty queue and an empty set of visited locations
Create an initial word ladder containing the starting word and add it to the
queue
While the queue is not empty
    Remove the next partial ladder from the queue
    Set the current search word to be the word at the top of the ladder
    If the current word is the destination, then return the current ladder
    Generate all "neighboring" words that are valid English words and one
    letter away from the current word
    Loop over all neighbor words
        If the neighbor word hasn't yet been visited
            Create a copy of the current ladder
            Add the neighbor to the top of the new ladder and mark it visited
            Add the new ladder to the back of the queue of partial ladders
```

Live Coding:
Implementing BFS

[Qt Creator]

Live Coding: Implementing BFS

[Qt Creator]

We hope that you find this to be a helpful resource when working on Assignment 2. However, we do not encourage trying to copy the code as a starting point. The problems are distinctly different, and you will benefit from explicitly developing your own problem-specific pseudocode first.

A final note: **`const`** reference

- Passing a large object (e.g. a million-element Vector) by value makes a copy, which is inefficient in time and space.
- Passing parameters by reference avoids making a copy, but creates risk that a function may modify a piece of data that you don't want it to edit.

- Solution: **`const`** reference!
  - The "by reference" part avoids a copy.
  - The "const" (constant) part means that the function can't change that argument.

```
void proofreadLongEssay(const string& essay) {
    /* can read, but not change, the essay. */
}
```