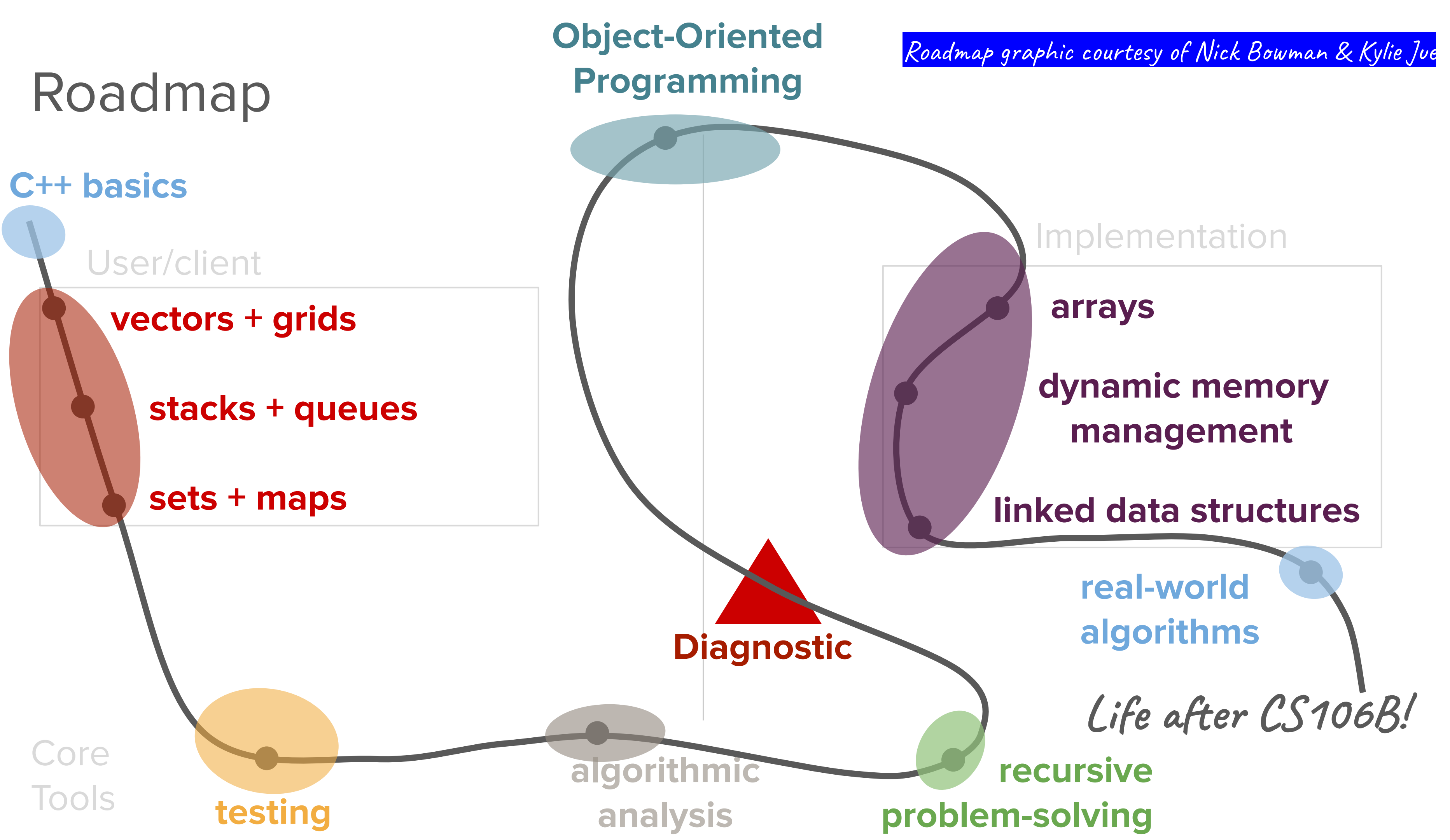
Using Abstractions: Breadth-First Search

What is a tradition that's special to you?

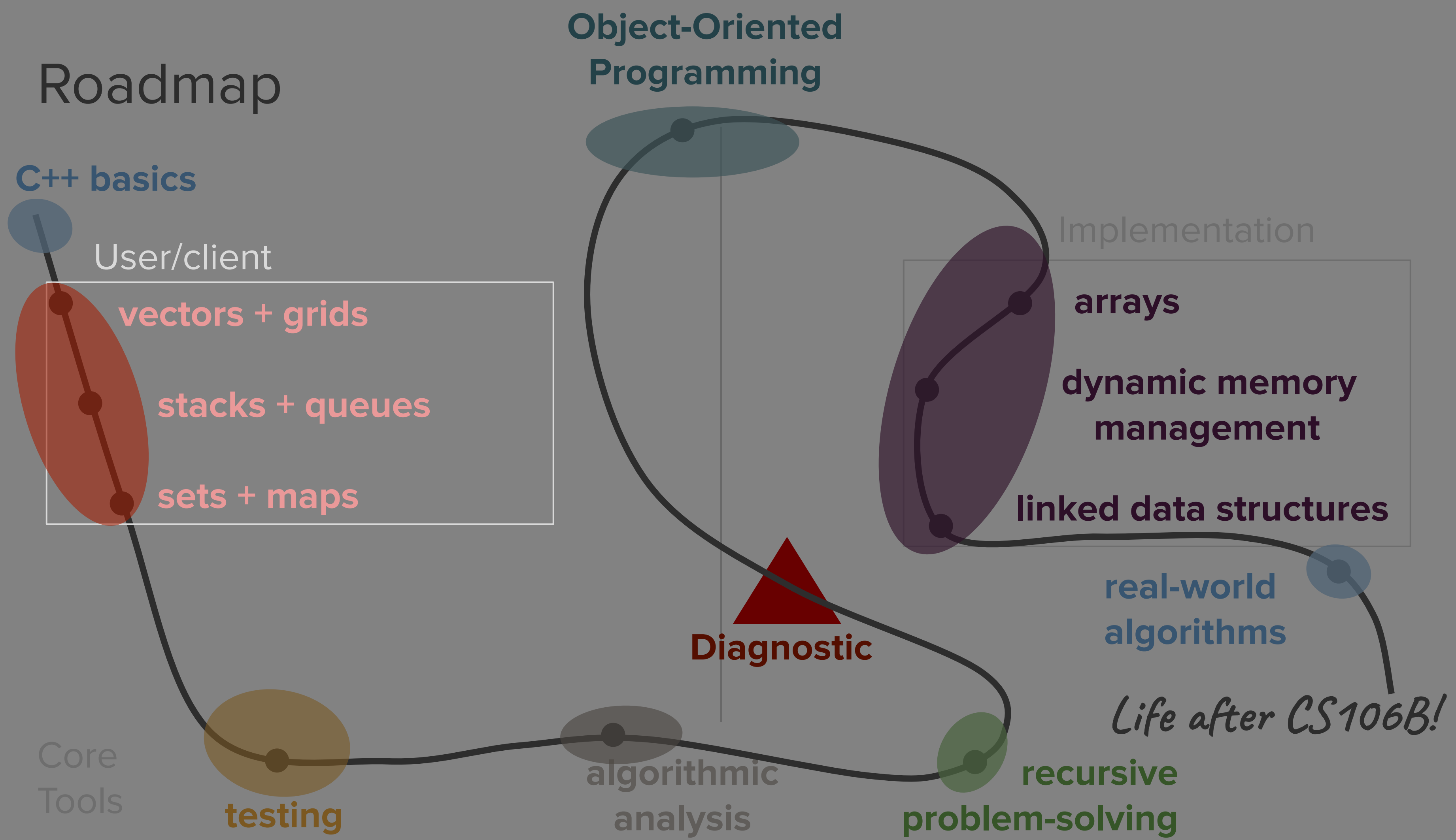
(put your answers in the chat)


STANFORD

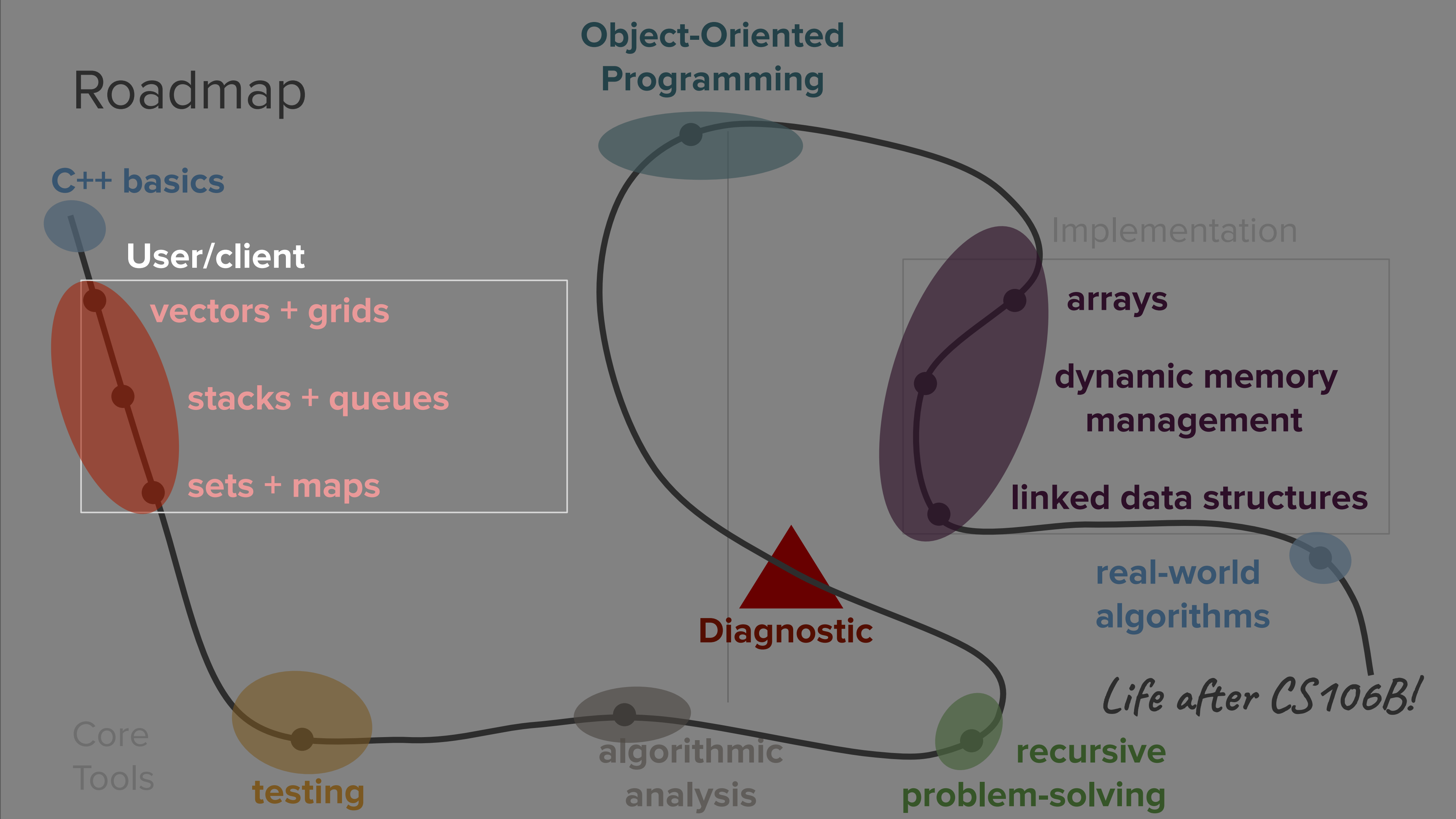
Roadmap


Roadmap graphic courtesy of Nick Bowman & Kylie Jue
Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap



Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving


Roadmap
Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

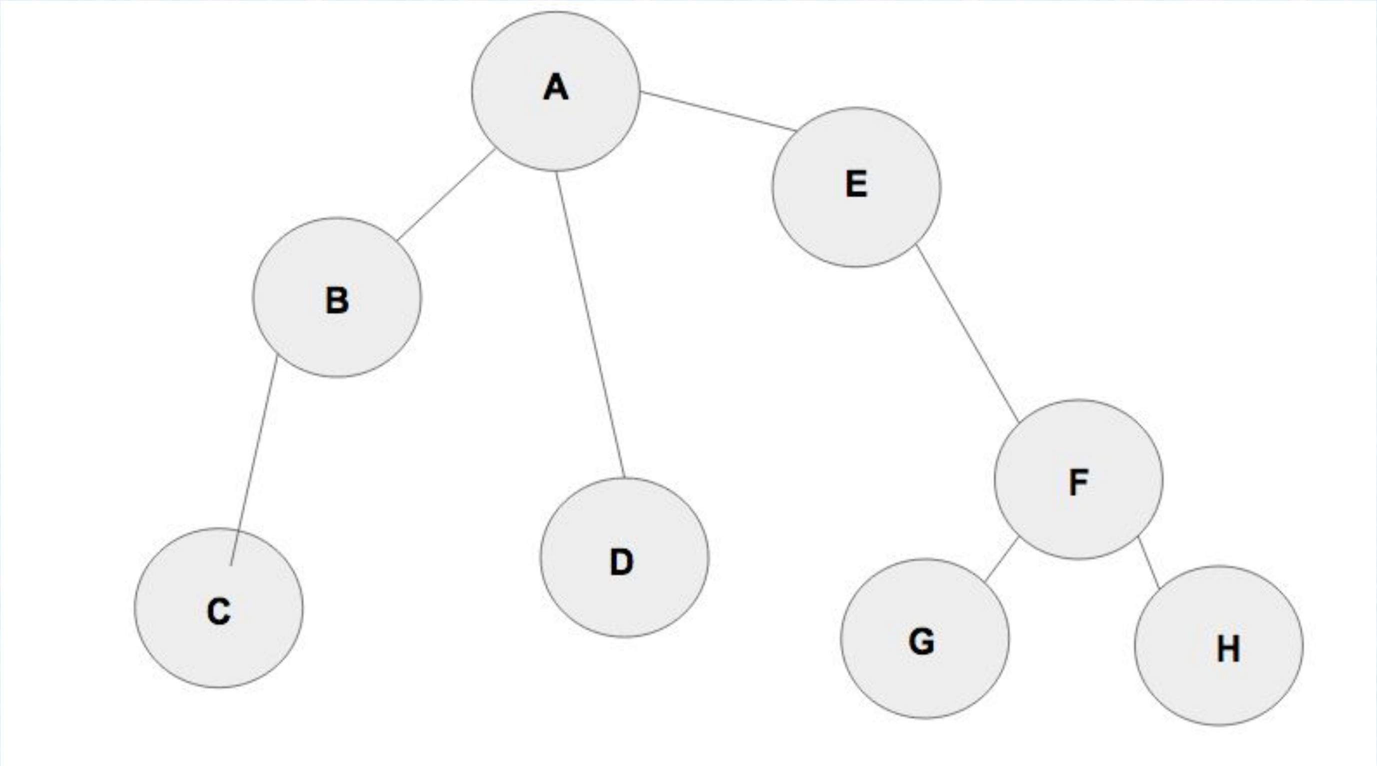
Today's question

How can we use the unique properties of different abstractions to solve problems?

Today's topics

1. Review
2. Implementing Counting Sort
3. Implementing Breadth-First Search

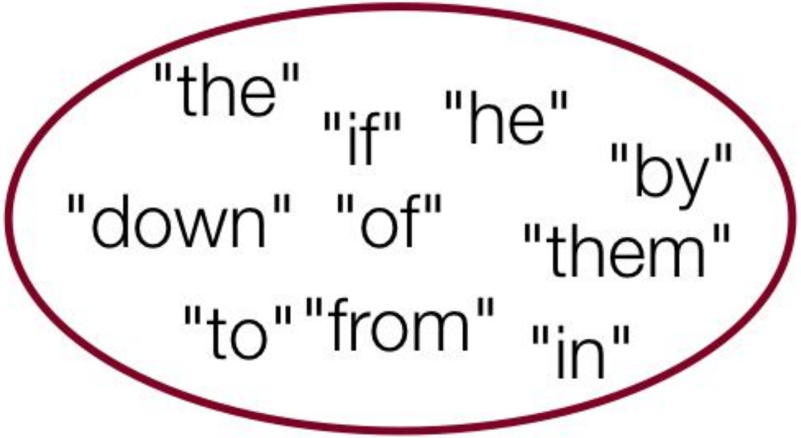
Breadth-First Search Algorithm



Review

sets and maps

What is a set?



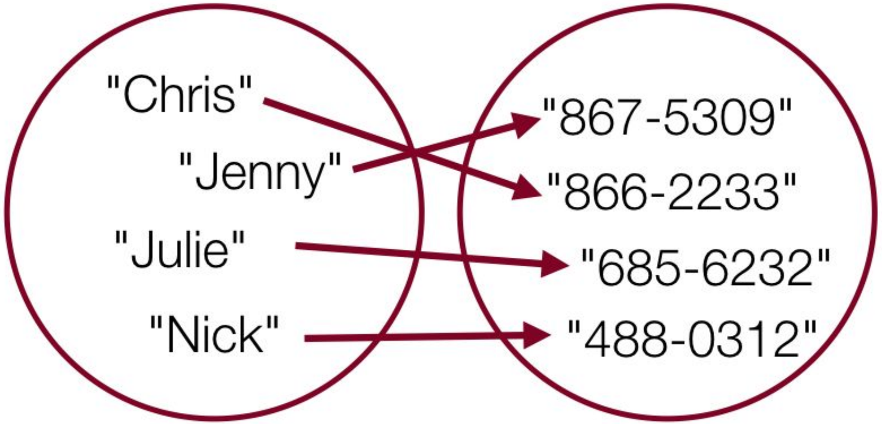
- A set is a collection of elements with no duplicates.

- Sets are faster than ordered data structures like vectors – since there are no duplicates, it's faster for them to find things.
    - (Later in the quarter we'll learn about the details of the underlying implementation that makes this abstraction efficient.)
    - We'll formally define "faster" on Thursday.

- Sets don't have indices!

What is a map?

- A map is a collection of key/value pairs, and the key is used to quickly find the value.



- A map is an alternative to an ordered data structure, where the “indices” no longer need to be integers.

Ordered ADTs

Elements accessible by indices:

- Vectors (1D)
- Grids (2D)

Elements not accessible by indices:

- Queues (FIFO)
- Stacks (LIFO)

Unordered ADTs

- Sets (elements unique)
- Keys (keys unique)

Useful when numerical ordering of data isn't optimal

Activity:

`Counting Sort`

Counting Sort

- Sorting is a fundamental topic in computer science and one that we will revisit in more depth later this quarter

Counting Sort

- Sorting is a fundamental topic in computer science and one that we will revisit in more depth later this quarter

- For now, let's consider this question: how would you efficiently sort all the letters in a word in alphabetical order?
  - How can we take advantage of some of the data structures we've recently learned about to meaningfully structure the data that we want to sort?

Counting Sort

- Sorting is a fundamental topic in computer science and one that we will revisit in more depth later this quarter
- For now, let's consider this question: how would you efficiently sort all the letters in a word in alphabetical order?
    - How can we take advantage of some of the data structures we've recently learned about to meaningfully structure the data that we want to sort?
- Idea: If we can tally up how many times each of the letters from 'a' to 'z' shows up, we can then build a new string composed of the correct number of 'a's, followed by the correct number of 'b's, ... etc.

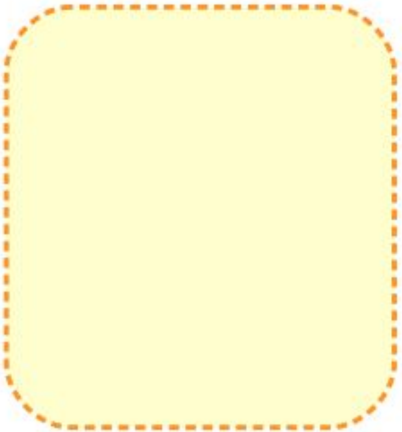
Counting Sort Example

Counting Sort Example

b	a	n	a	n	a

Counting Sort Example

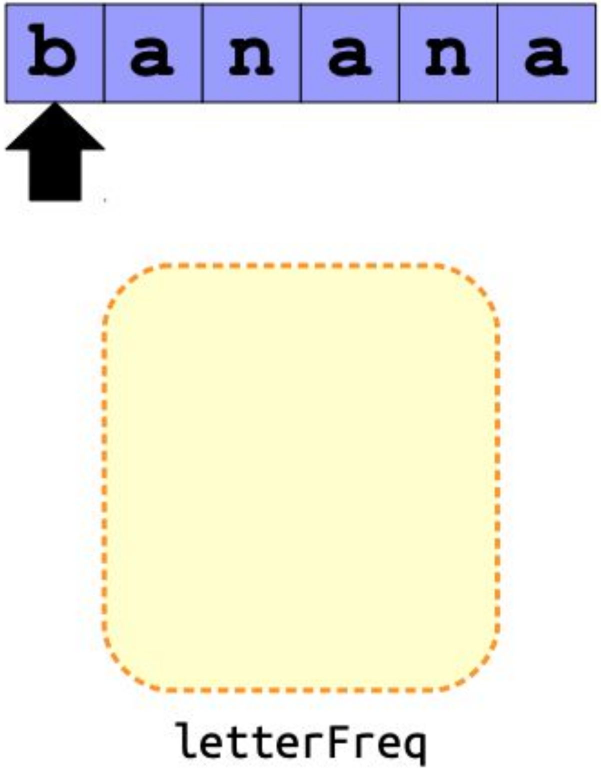
b	a	n	a	n	a



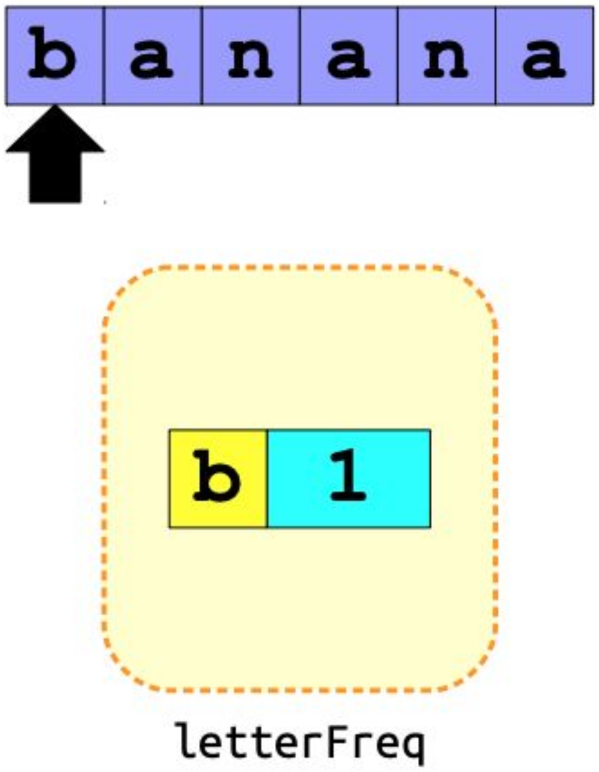
letterFreq

Example and slides taken from a slide deck by Keith Schwarz

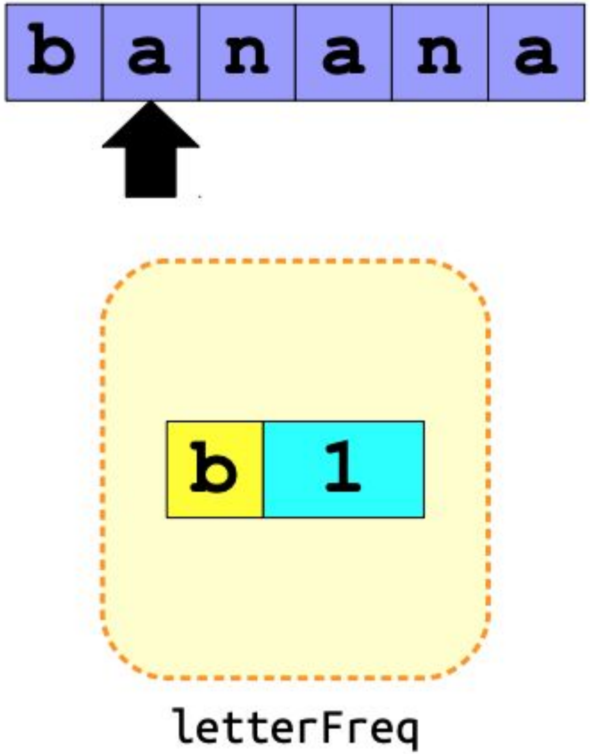
Counting Sort Example



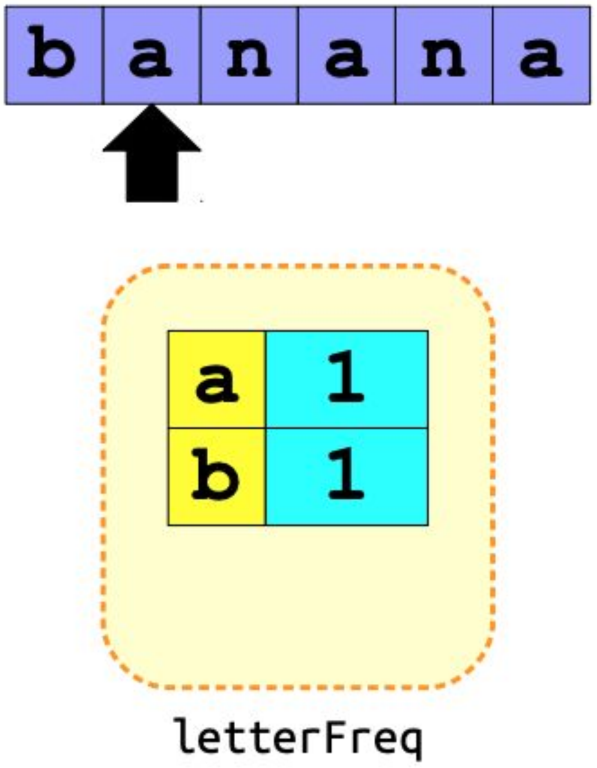
Counting Sort Example



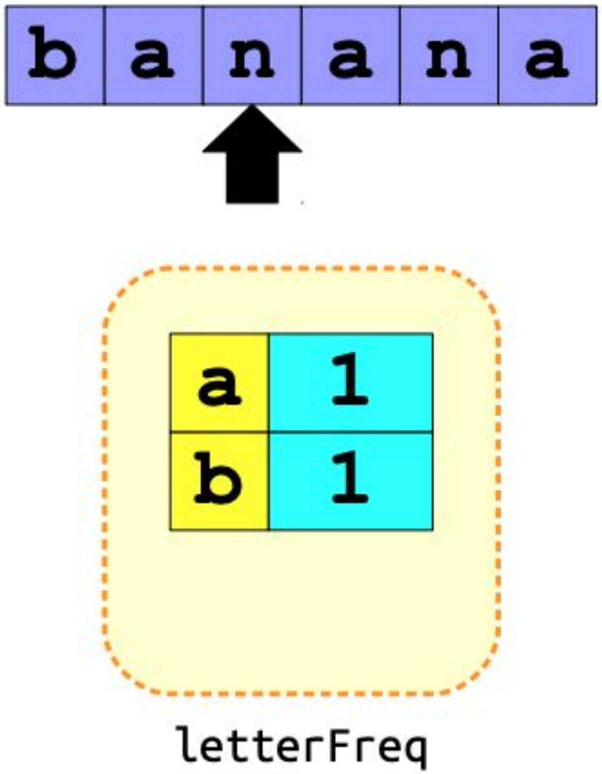
Counting Sort Example



Counting Sort Example

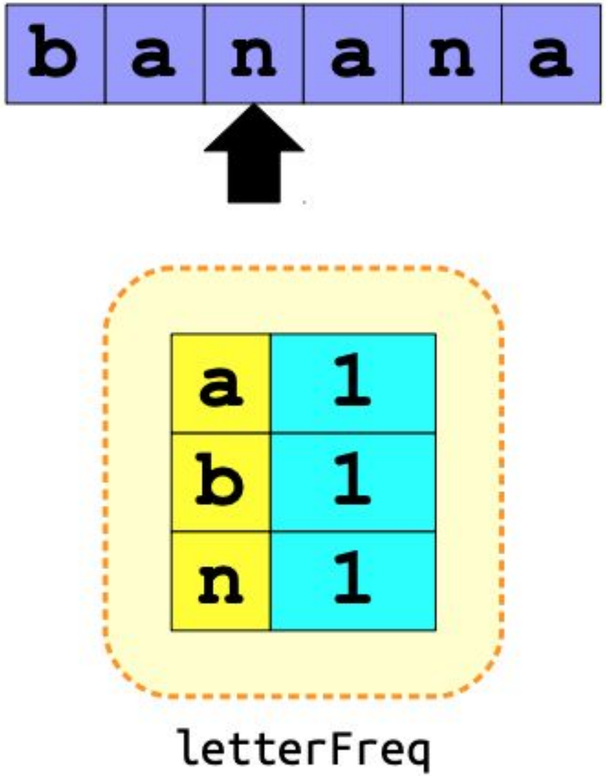


Counting Sort Example

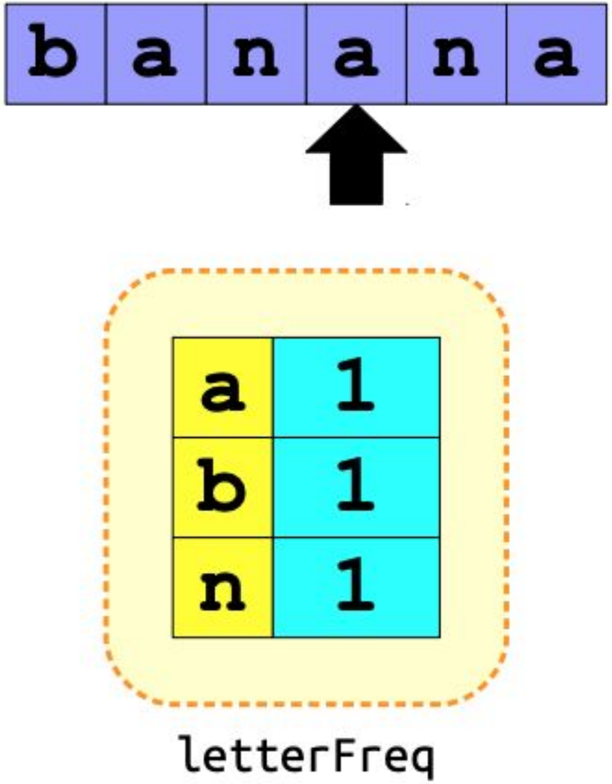


Example and slides taken from a slide deck by Keith Schwarz

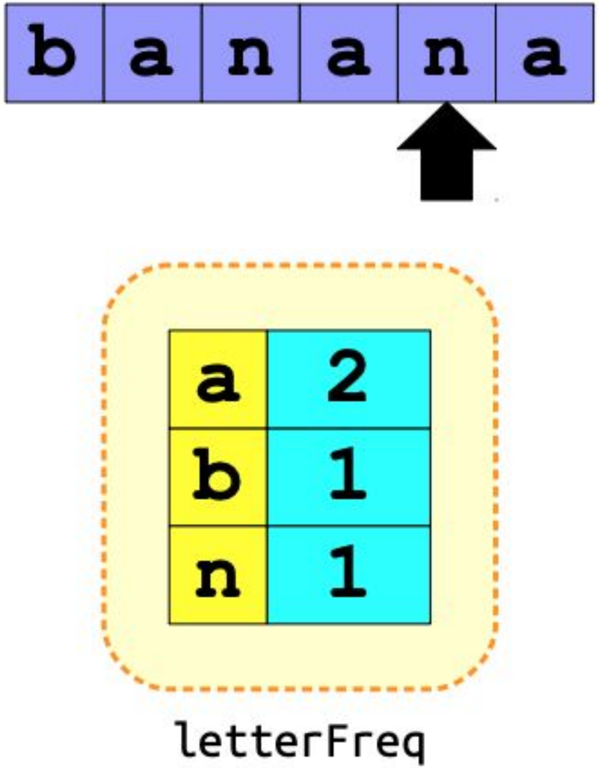
Counting Sort Example



Counting Sort Example

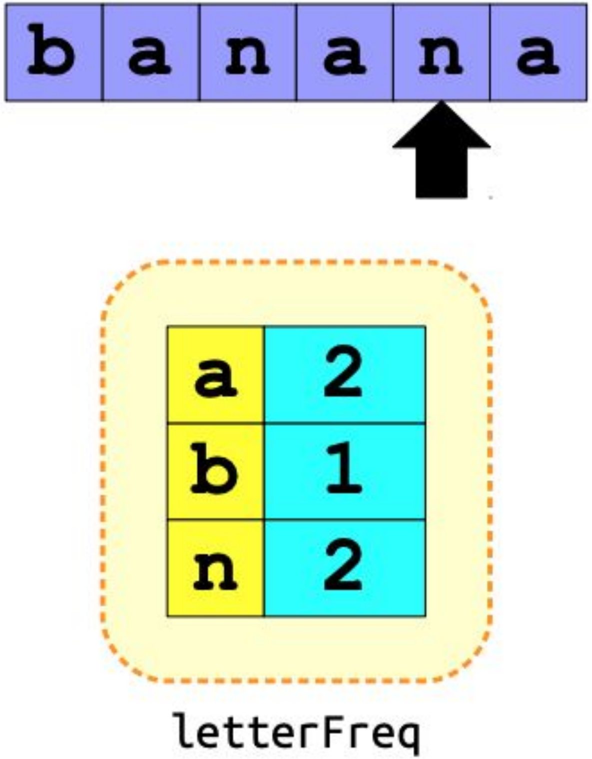


Counting Sort Example

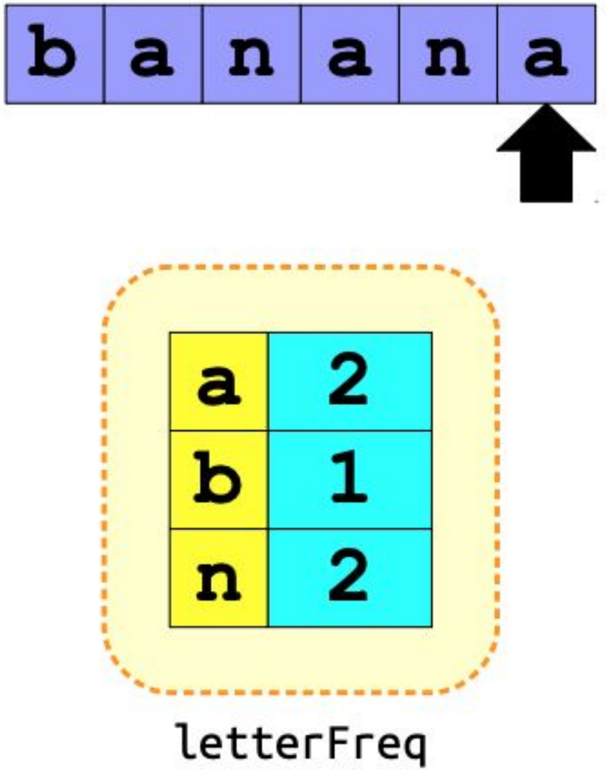


Example and slides taken from a slide deck by Keith Schwarz

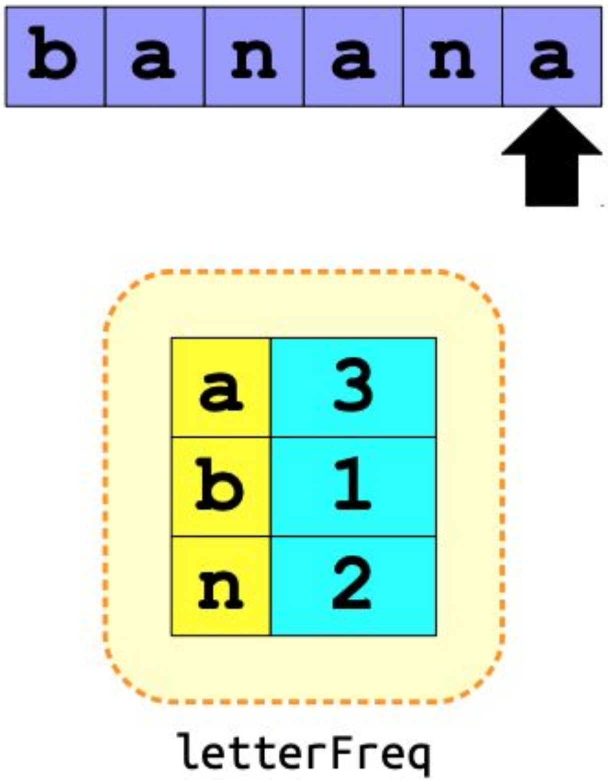
Counting Sort Example



Counting Sort Example



Counting Sort Example



Counting Sort Example

b	a	n	a	n	a

a	3
b	1
n	2

letterFreq

Counting Sort Example

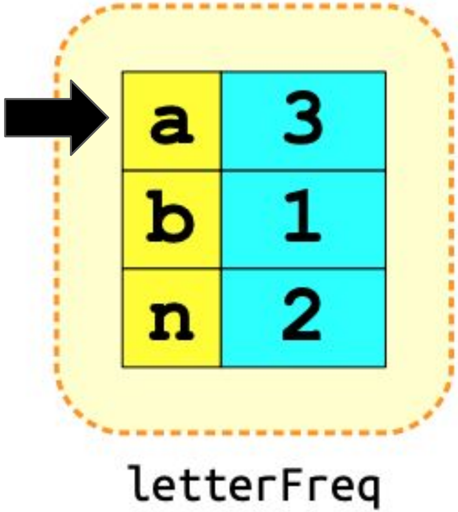
b	a	n	a	n	a

a	3
b	1
n	2

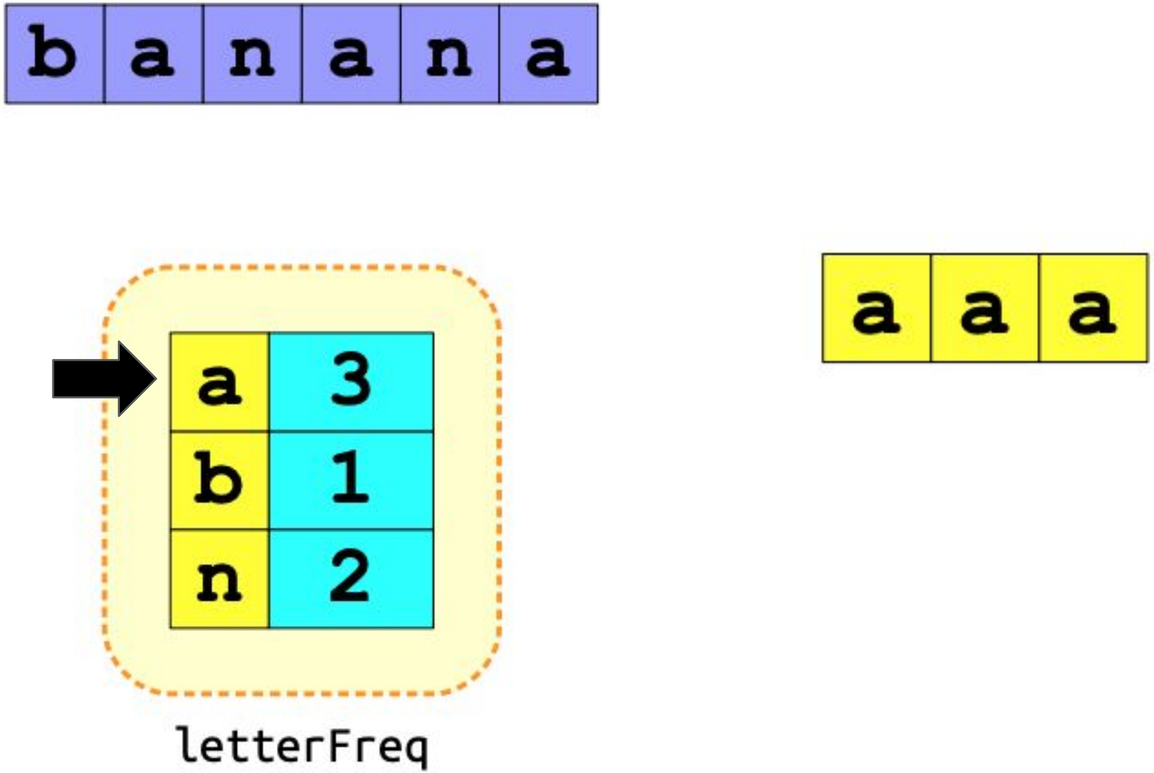
letterFreq

Counting Sort Example

b	a	n	a	n	a

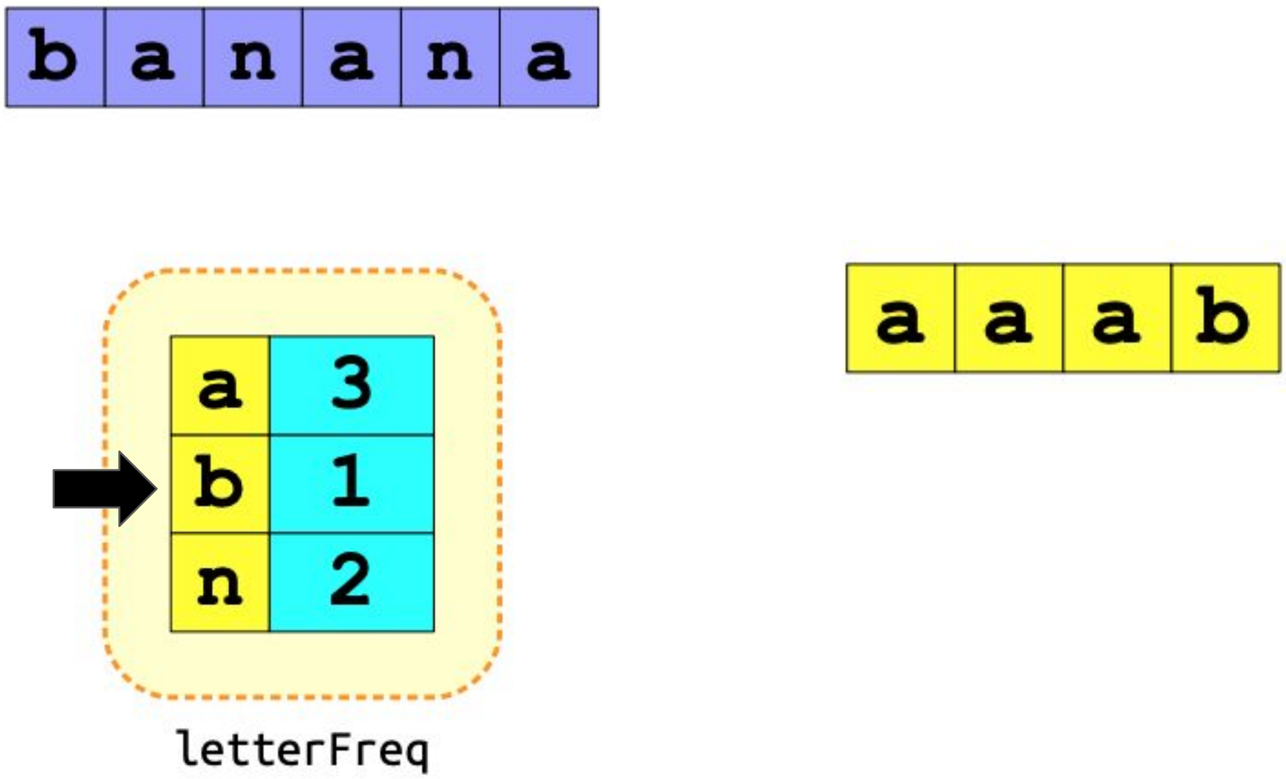


Counting Sort Example



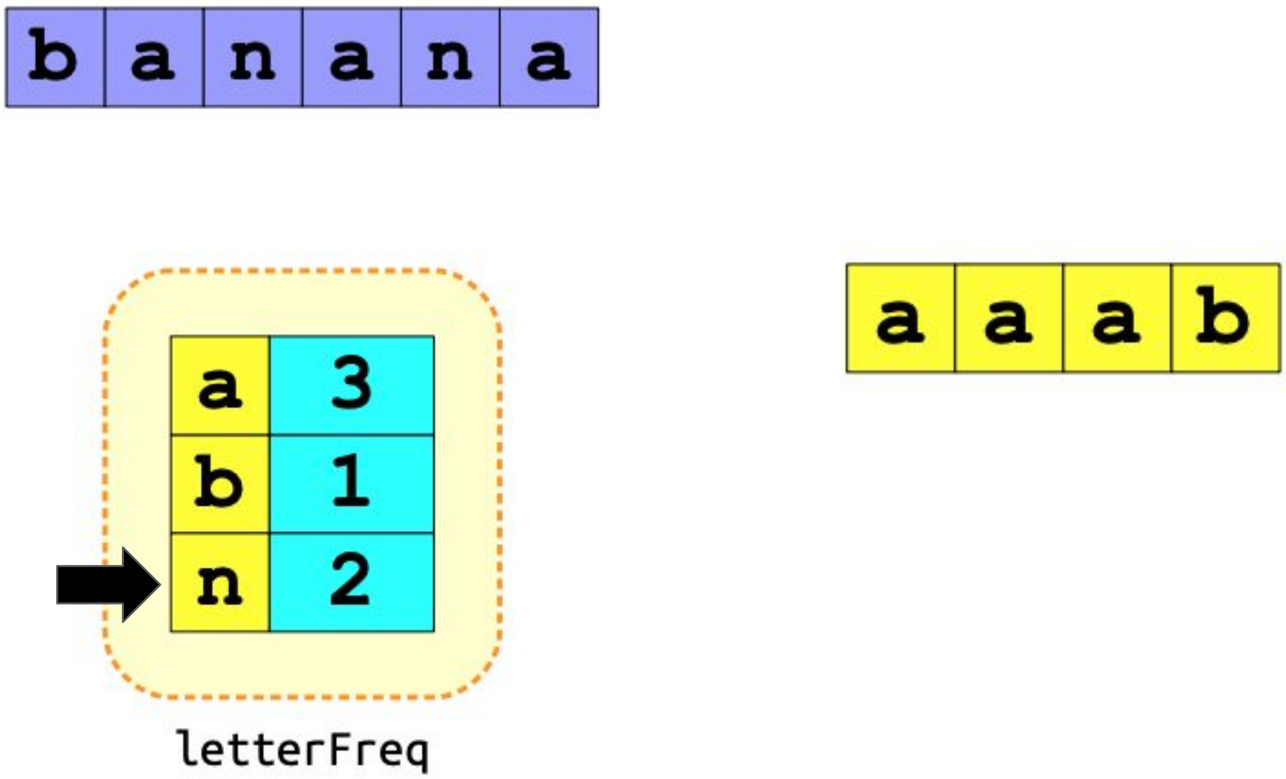
Example and slides taken from a slide deck by Keith Schwarz

Counting Sort Example

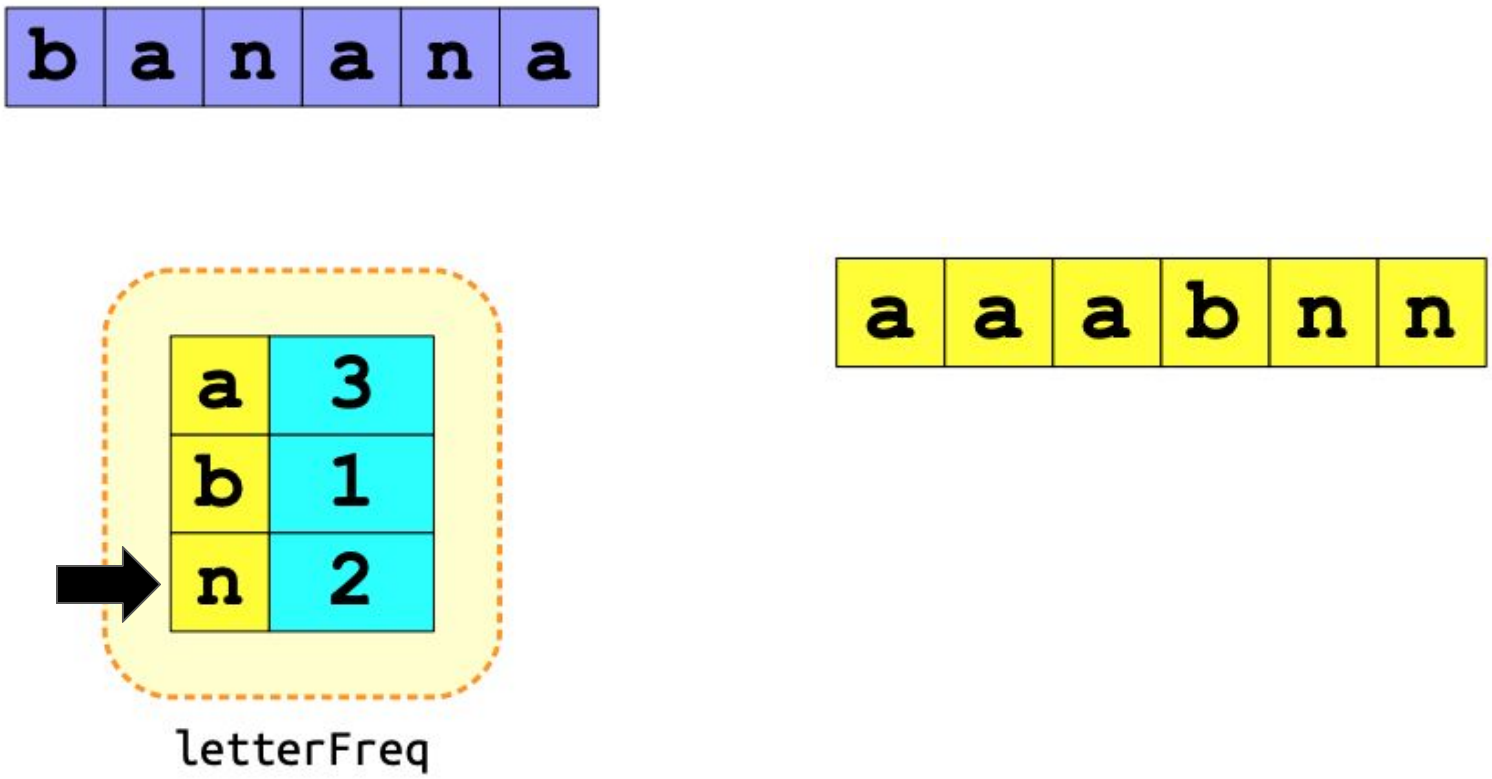


Example and slides taken from a slide deck by Keith Schwarz

Counting Sort Example



Counting Sort Example



Counting Sort Example

b	a	n	a	n	a

a	3
b	1
n	2

letterFreq

a	a	a	b	n	n

Mission Accomplished!

Counting Sort Pseudocode

- Loop over the word and build a frequency map of all letters that appear in the original string
- Loop through all letters from 'a' to 'z' and build up a new string with the right amount of each letter
- Return the newly generated string

pseudocode
before implementing
the algorithm

Counting Sort Pseudocode

- Loop over the word and build a frequency map of all letters that appear in the original string
- Loop through all letters from 'a' to 'z' and **build up a new string with the right amount of each letter**
- Return the newly generated string

Provided Code

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {

    }
    return sortedString;
}
```

- Loop over the word and build a frequency map of all letters that appear in the original string
- Loop through all letters from 'a' to 'z' and build up a new string with the right amount of each letter

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        /* TODO: Generate pseudocode to complete the algorithm! */
        /*
    }
    return sortedString;
}
```

Loop through all letters from 'a' to 'z' and build up a new string with the right amount of each letter

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        /* TODO: Generate pseudocode
           to complete the algorithm! */
    }
    return sortedString;
}
```

Loop through all letters from 'a' to 'z' and build up a new string with the right amount of each letter

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
     /* TODO: Generate pseudocode to complete the algorithm! */
     /* Use ch as key into the freq map & get associated value. */

    }
    return sortedString;
}
```

Loop through all letters from 'a' to 'z' and build up a new string with the right amount of each letter

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        /* TODO: Generate pseudocode to complete the algorithm! */
        /* Use ch as key into the freq map & get associated value. */
        /* Add ch to sortedString as many times as that value. */
    }
    return sortedString;
}
```

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        /* Use ch as key into the freq map & get associated value. */
        /* Add ch to sortedString as many times as that value. */
    }
    return sortedString;
}
```

Counting Sort Code

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        // taking advantage of map auto-insertion!
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        /* Use ch as key into the freq map and get associated value. */
        /* Add ch to sortedString as many times as that value. */
        for (int i = 0; i < freqMap[ch]; i++) {
           sortedString += charToString(ch);
        }
    }
    return sortedString;
}
```

Counting Sort Code

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        // taking advantage of map auto-insertion!
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        /* Use ch as key into the freq map and get associated value. */
        /* Add ch to sortedString as many times as that value. */
        for (int i = 0; i < freqMap[ch]; i++) {
           sortedString += charToString(ch);
        }
    }
    return sortedString;
}
```

Loop through all letters from 'a' to 'z' and build up a new string with the right amount of each letter

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        /* Check if the freq map contains the key ch */
        /* If so, get associated value for ch key from freq map. */
        /* Add ch to sortedString as many times as that value. */
    }
    return sortedString;
}
```

Counting Sort Code

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        // taking advantage of map auto-insertion!
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        /* Check if the freq map contains the key ch */
        if (freqMap.containsKey(ch)) {
            /* If so, get associated value for ch key from freq map. */
             /* Add ch to sortedString as many times as that value. */
             for (int i = 0; i < freqMap[ch]; i++) {
                sortedString += charToString(ch);
             }
        }
    }
    return sortedString;
}
```

Counting Sort Code

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        // taking advantage of map auto-insertion!
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        if (freqMap.containsKey(ch)) {
            for (int i = 0; i < freqMap[ch]; i++) {
                sortedString += charToString(ch);
            }
        }
    }
    return sortedString;
}
```

This check isn't strictly required, but it does avoid unnecessary things being added to the map via auto-insertion

Counting Sort Code

```
string countingSort(string s) {
    Map<char, int> freqMap;
    for (char ch: s) {
        // taking advantage of map auto-insertion!
        freqMap[ch] = freqMap[ch] + 1;
    }

    string sortedString;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        if (freqMap.containsKey(ch)) {
            for (int i = 0; i < freqMap[ch]; i++) {
                sortedString += charToString(ch);
            }
        }
    }
    return sortedString;
}
```

Challenge for home:

What other types of data could you efficiently sort in this manner?

How can we use the unique properties of different abstractions to solve problems?

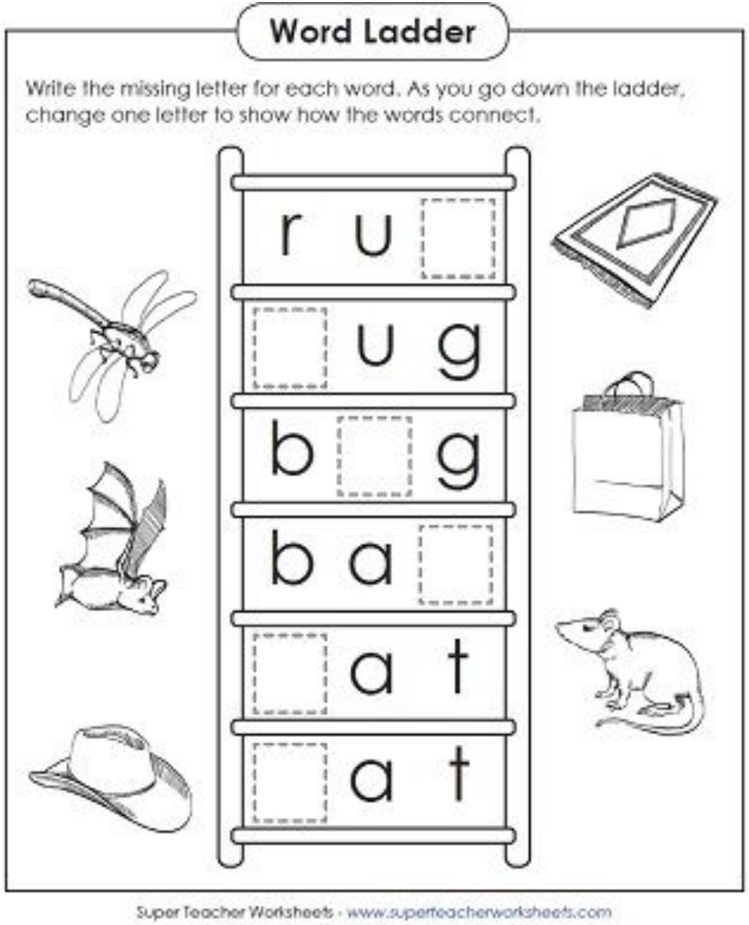
Examples of interesting problems to solve using ADTs

- 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 to help tourists plan their trip to the state
- Develop a ticketing management system for a stadium
- Aggregate and analyze reviews for an online shopping website
- Solve fun puzzles

Examples of interesting problems to solve using ADTs

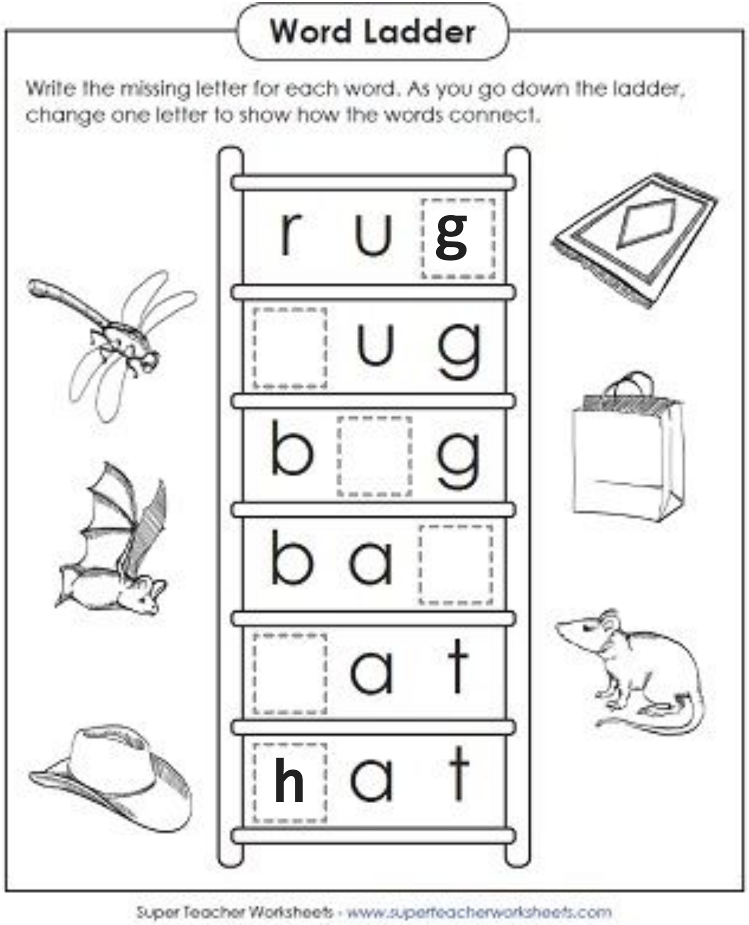
- Simulate potential impacts of flooding on a topographical landscape (how does water flow outwards from a source and settle into the surrounding areas)
- Generate simulated text in the style of a certain author. Similarly, do textual analysis to determine who the author of a provided piece of text was.
- 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**

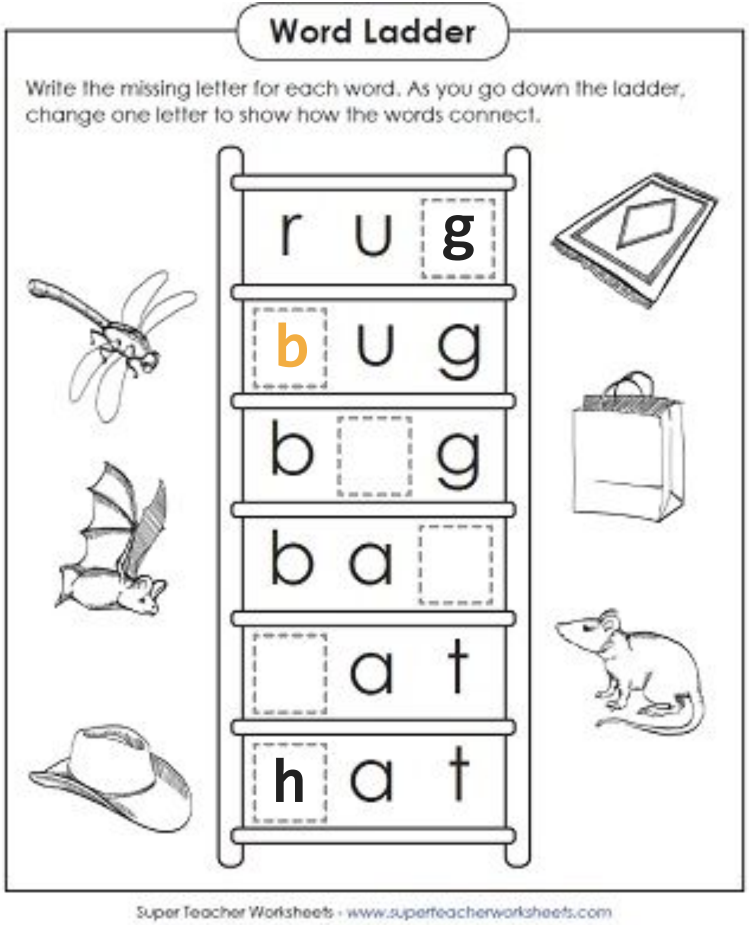
Word Ladders

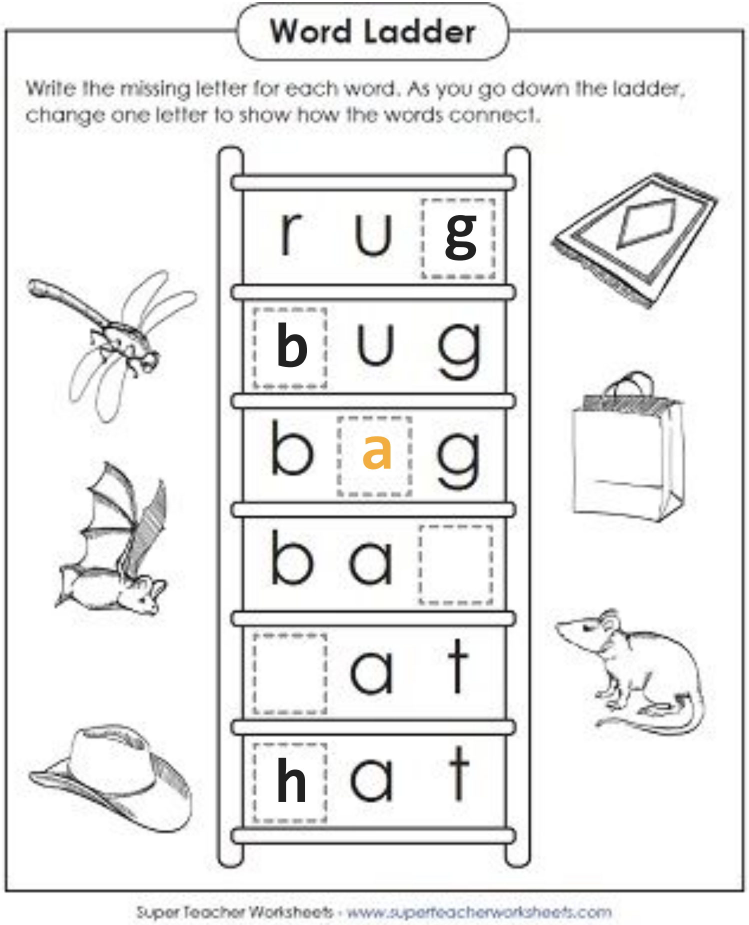


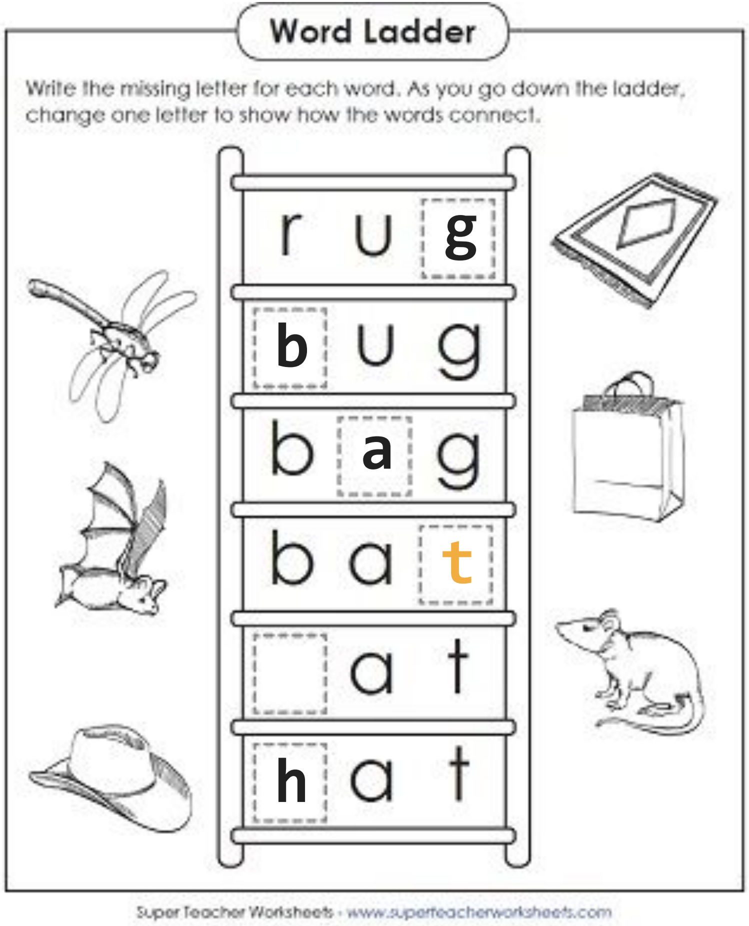
start word

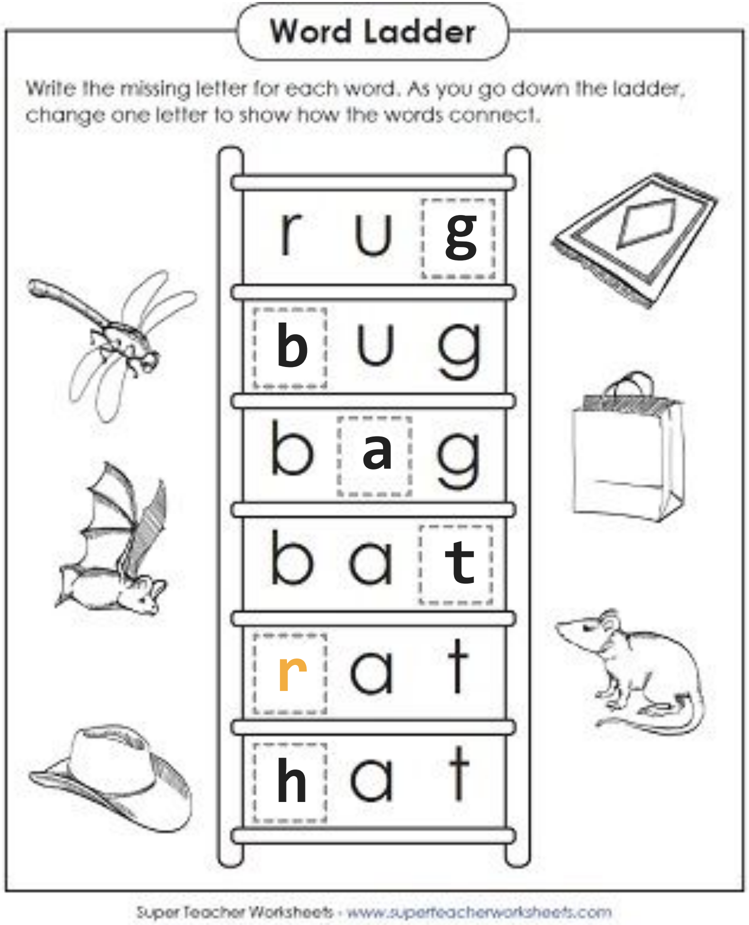
ending word

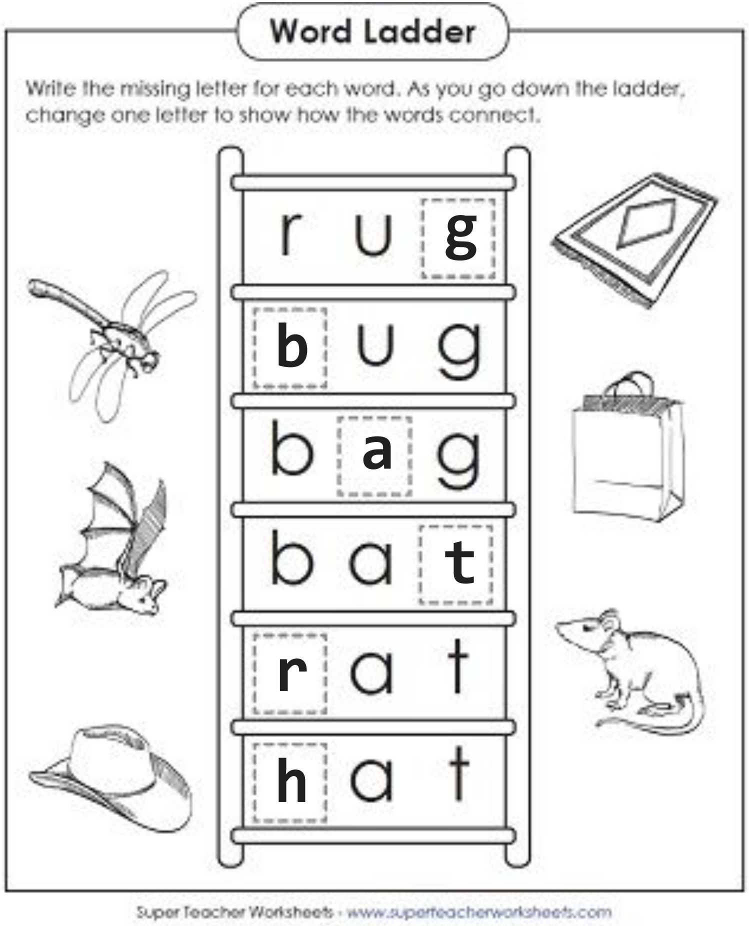

Word Ladder
Write the missing letter for each word. As you go down the ladder, change one letter to show how the words connect.
r u g
u g
b g
b a
a t
h a t
Super Teacher Worksheets - www.superteacherworksheets.com


Word Ladder
Write the missing letter for each word. As you go down the ladder, change one letter to show how the words connect.
r u g
b u g
b g
b a
a t
h a t
Super Teacher Worksheets - www.superteacherworksheets.com

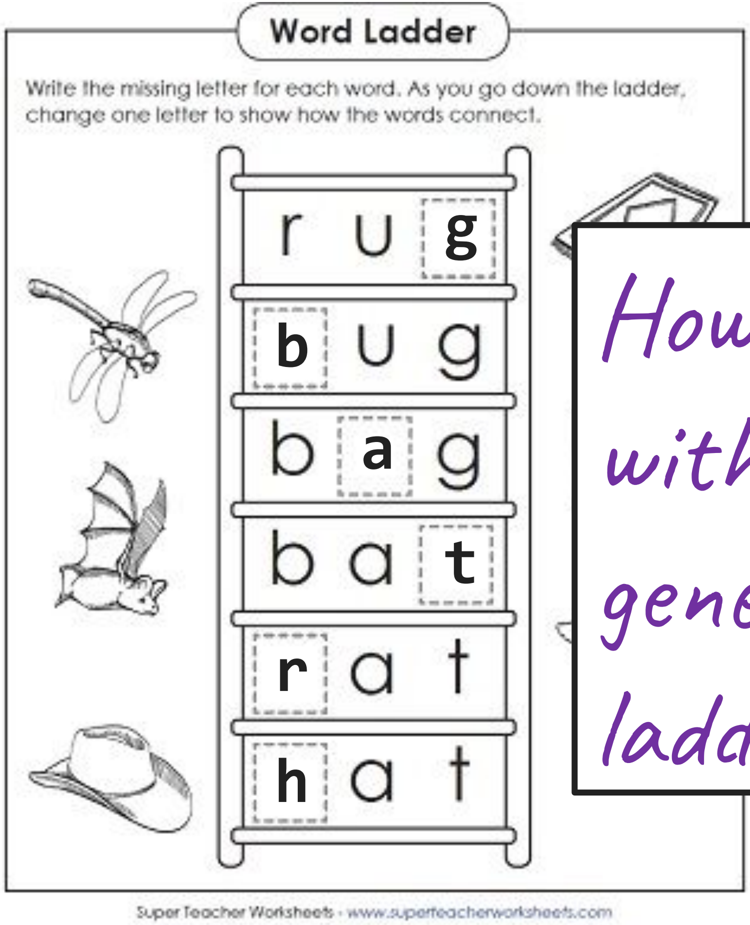

Word Ladder
Write the missing letter for each word. As you go down the ladder, change one letter to show how the words connect.
r u g
b u g
b a g
b a
a t
h a t
Super Teacher Worksheets - www.superteacherworksheets.com


Word Ladder
Write the missing letter for each word. As you go down the ladder, change one letter to show how the words connect.
r u g
b u g
b a g
b a t
a t
h a t
Super Teacher Worksheets - www.superteacherworksheets.com


Word Ladder
Write the missing letter for each word. As you go down the ladder, change one letter to show how the words connect.
r u g
b u g
b a g
b a t
r a t
h a t
Super Teacher Worksheets - www.superteacherworksheets.com



Word Ladder
Write the missing letter for each word. As you go down the ladder, change one letter to show how the words connect.
r u g
b u g
b a g
b a t
r a t
h a t
Super Teacher Worksheets - www.superteacherworksheets.com



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 (BFS)

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.**

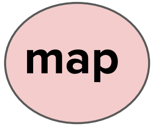
Breadth-First Search Example

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"

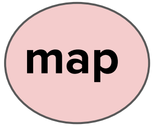
Breadth-First Search Example

start: map
destination: way



Breadth-First Search Example

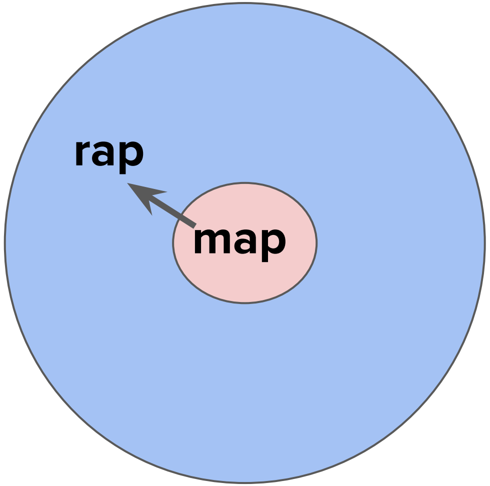
start: map
destination: way



0 steps away

Breadth-First Search Example

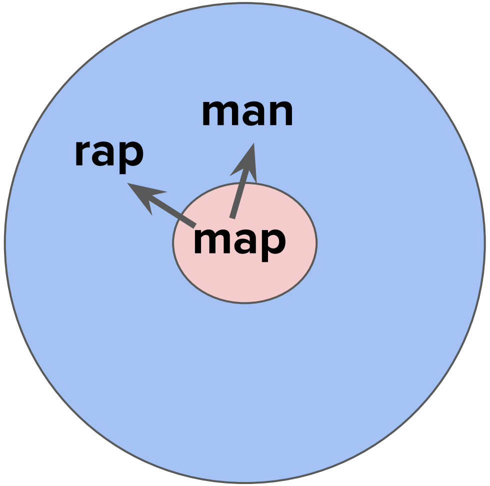
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

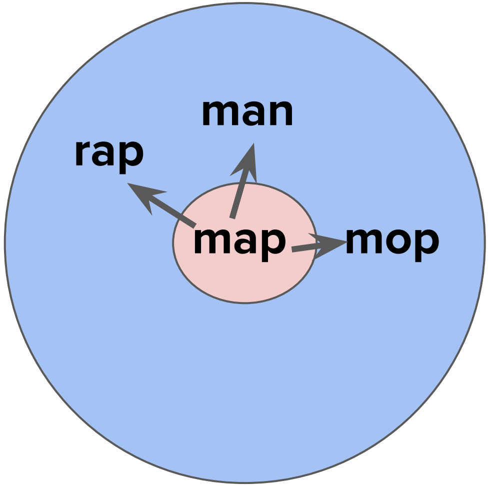
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

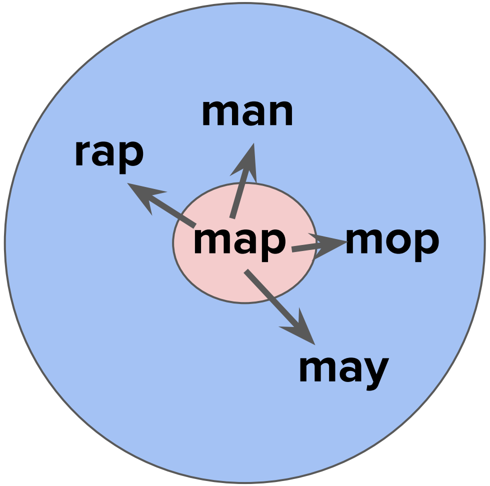
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

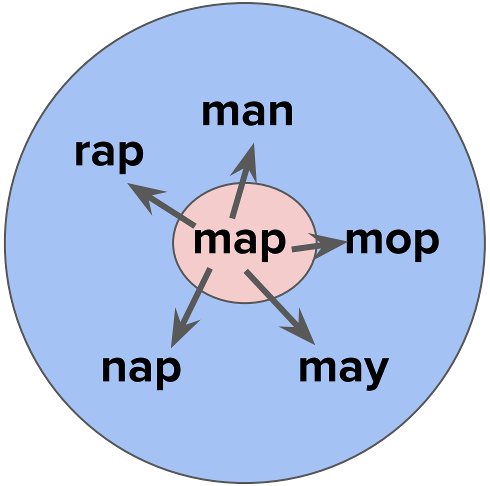
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

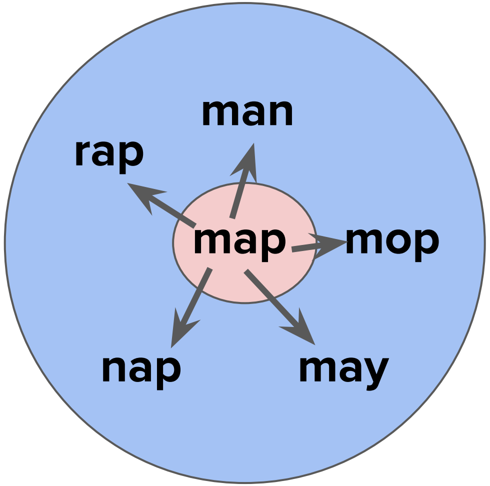
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

start: map
destination: way

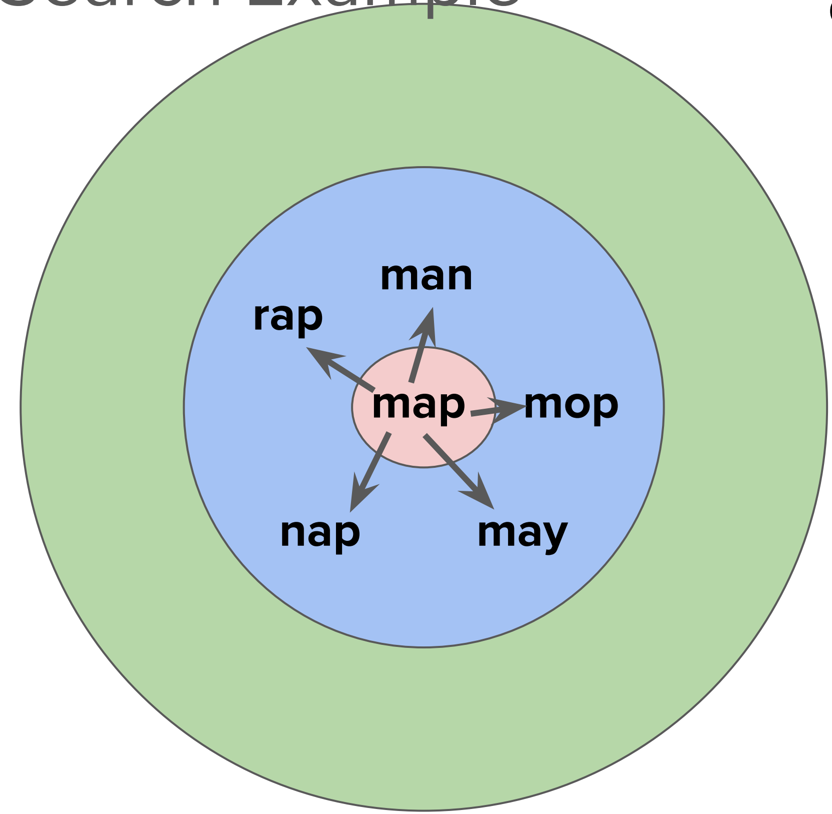


0 steps away
1 step away

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

Breadth-First Search Example

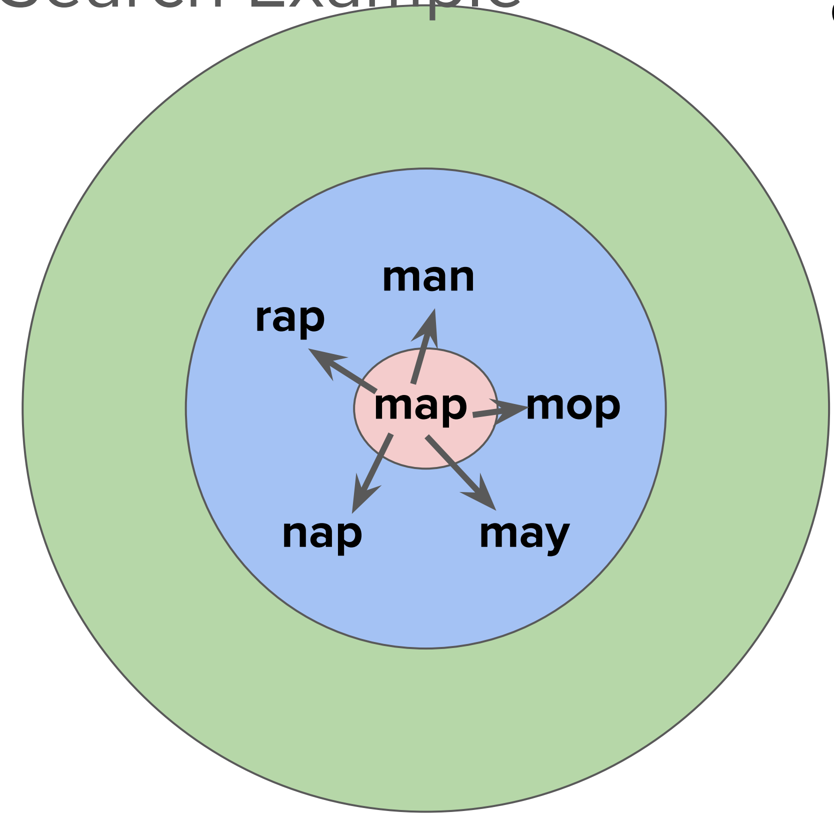
start: map
destination: way



0 steps away
1 step away

Breadth-First Search Example

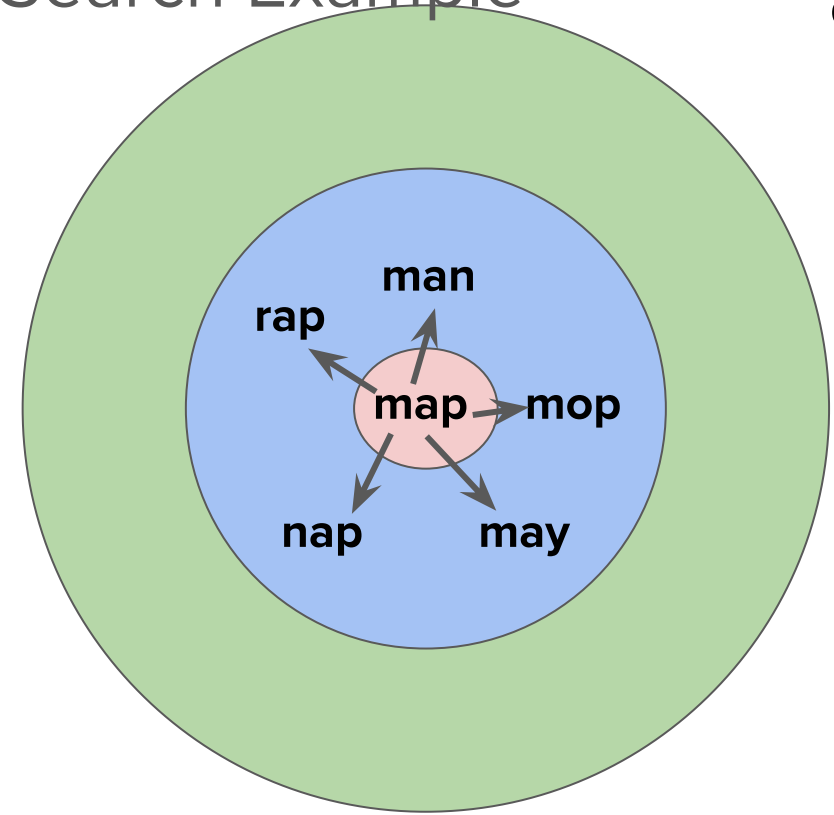
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

start: map
destination: way

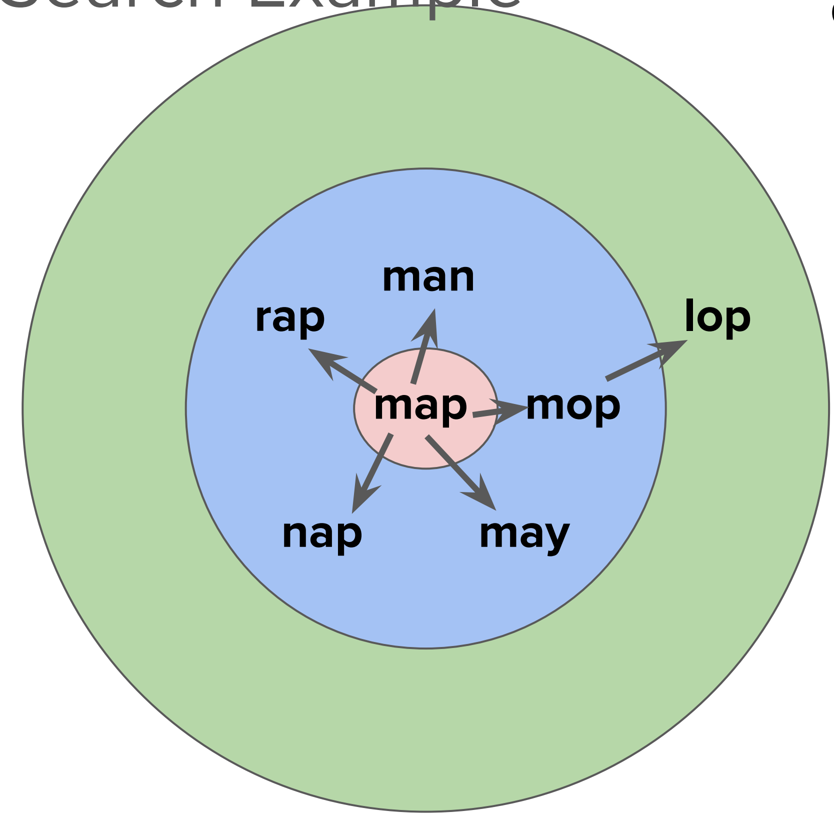


Observation: 2 steps away from "map" is really just 1 step away from any of its neighbors

0 steps away
1 step away
2 steps away

Breadth-First Search Example

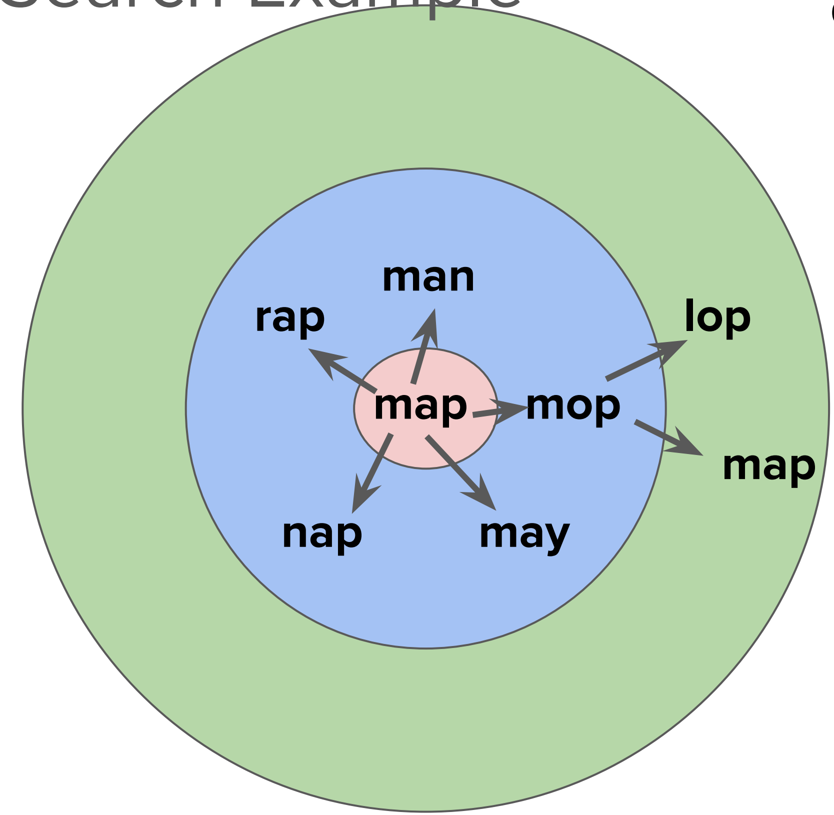
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

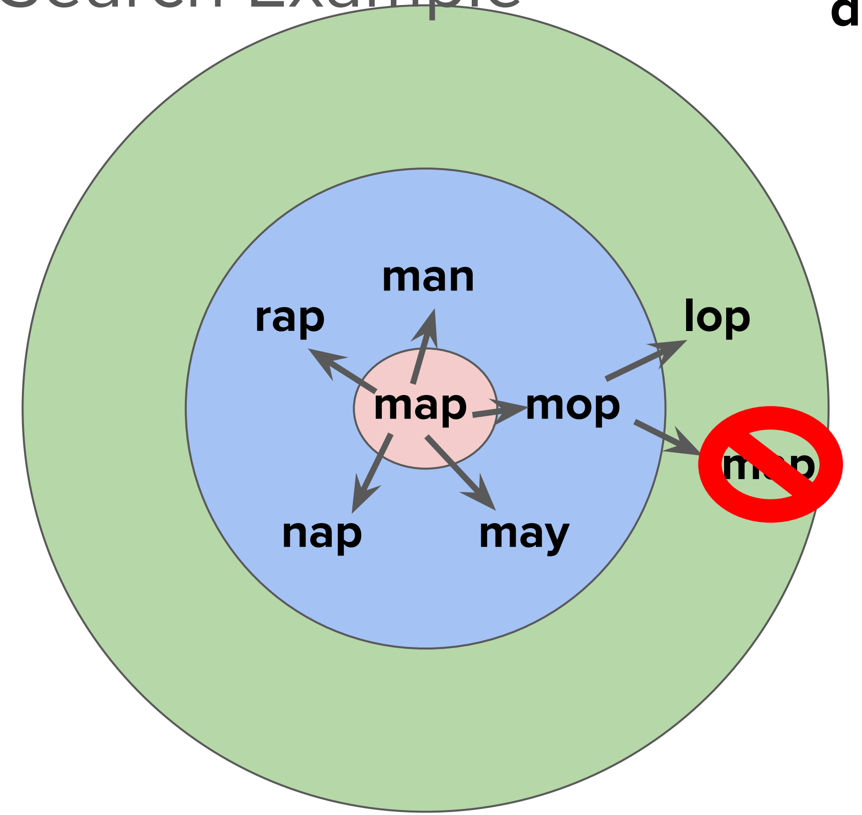
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

start: map
destination: way

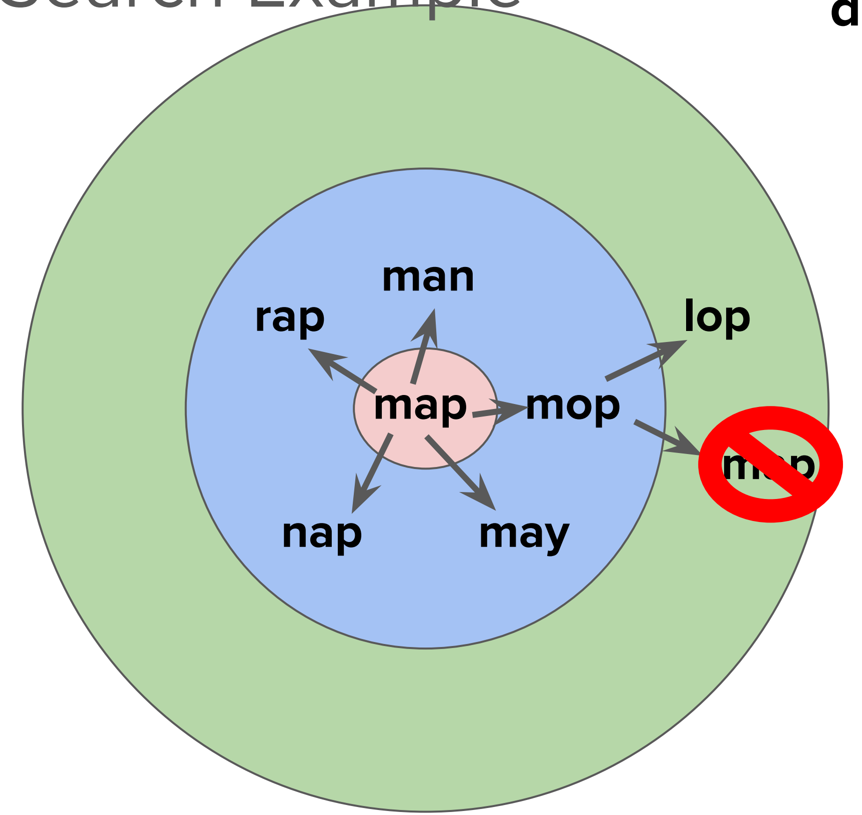


Visiting a word we've already been at before is basically like going backwards in our search. We want to avoid this at all costs!

0 steps away
1 step away
2 steps away

Breadth-First Search Example

start: map
destination: way

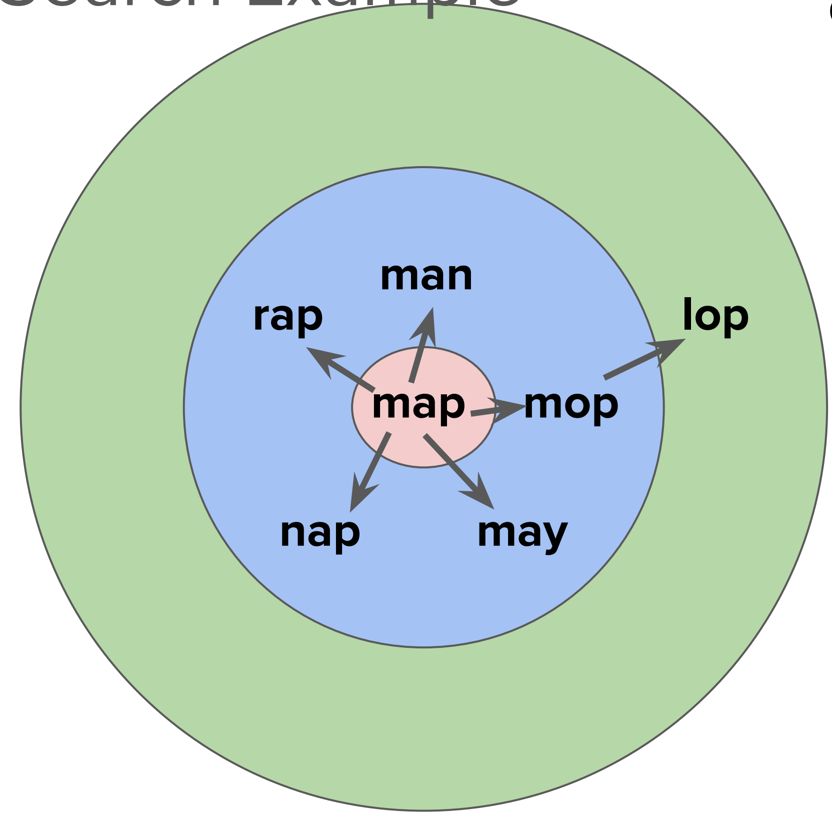


Idea: Keep track of a collection of visited words, and don't double visit

0 steps away
1 step away
2 steps away

Breadth-First Search Example

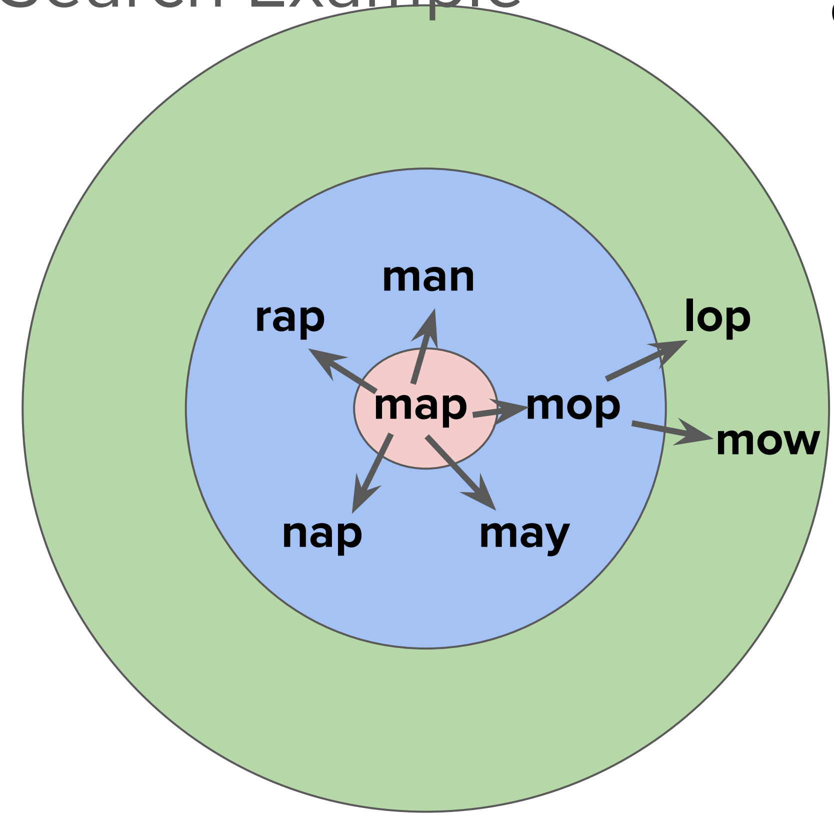
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

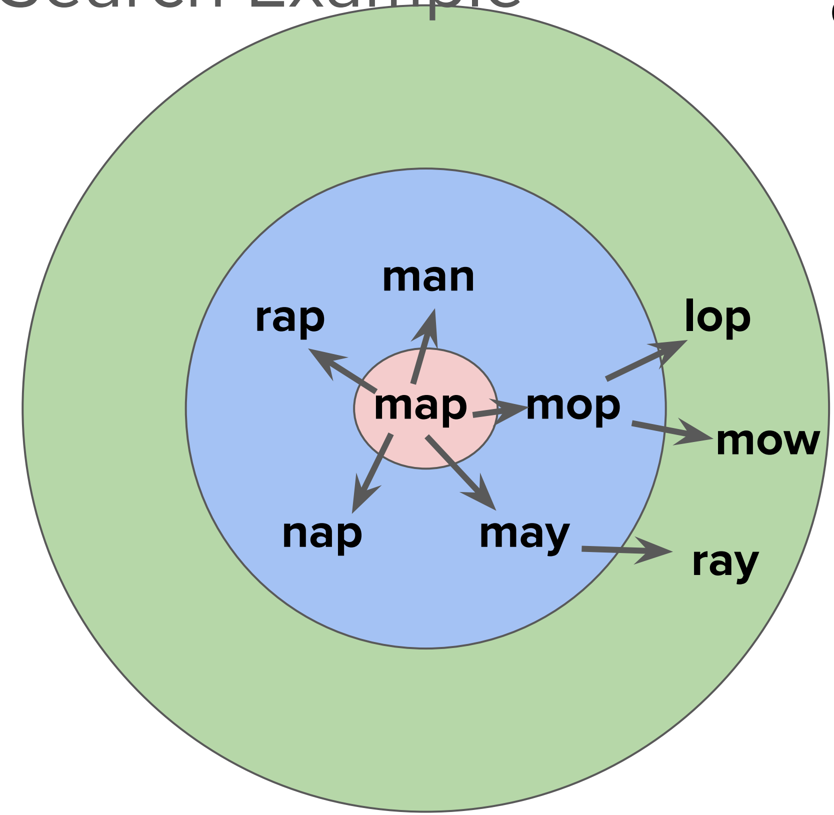
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

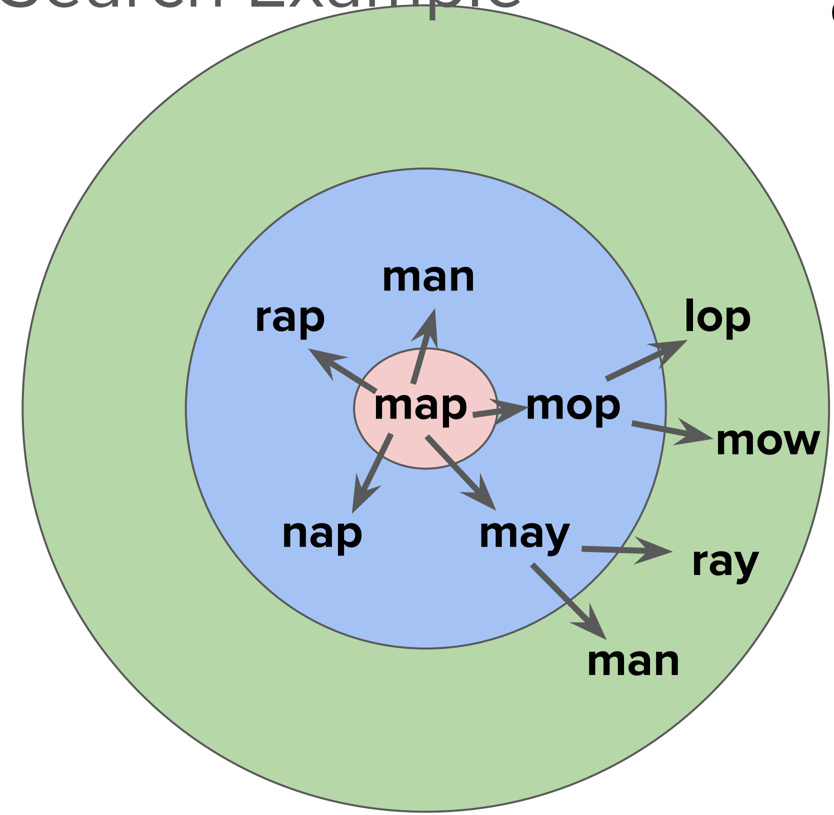
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

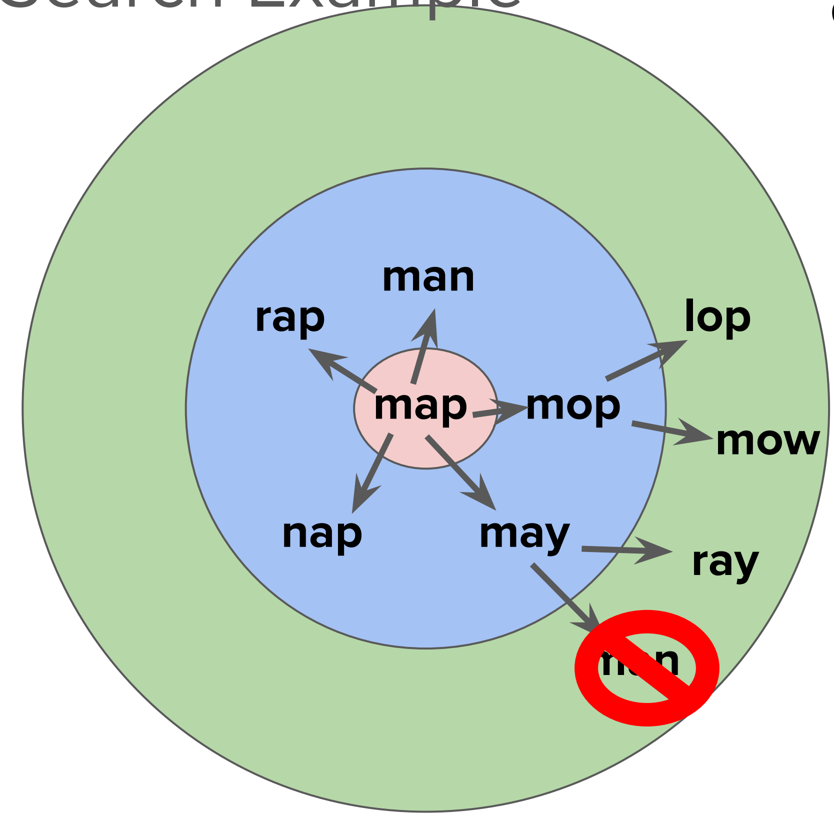
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

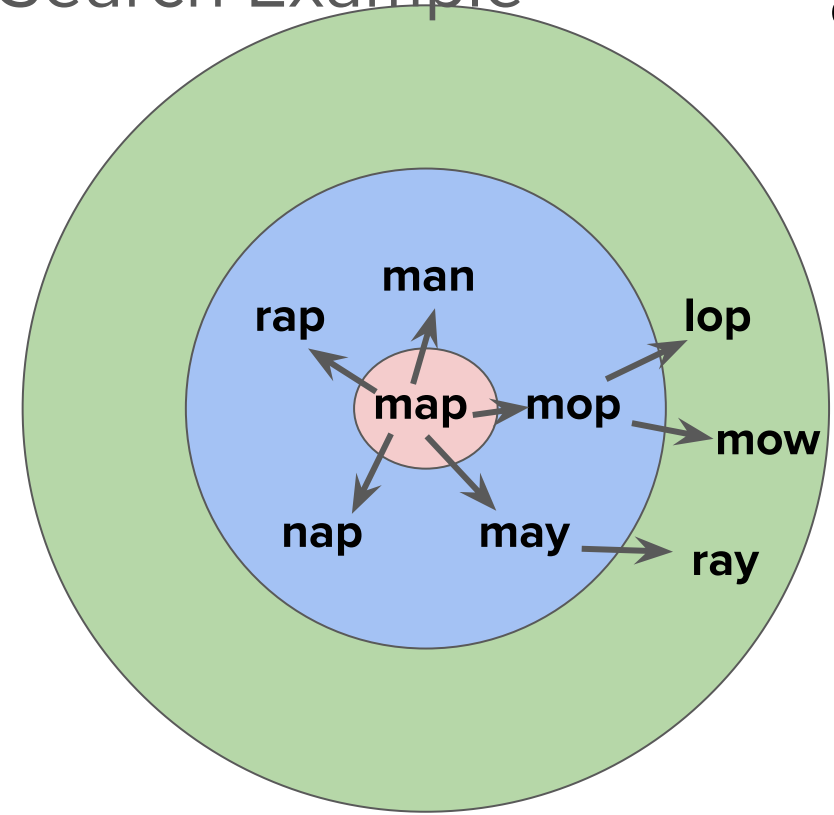
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

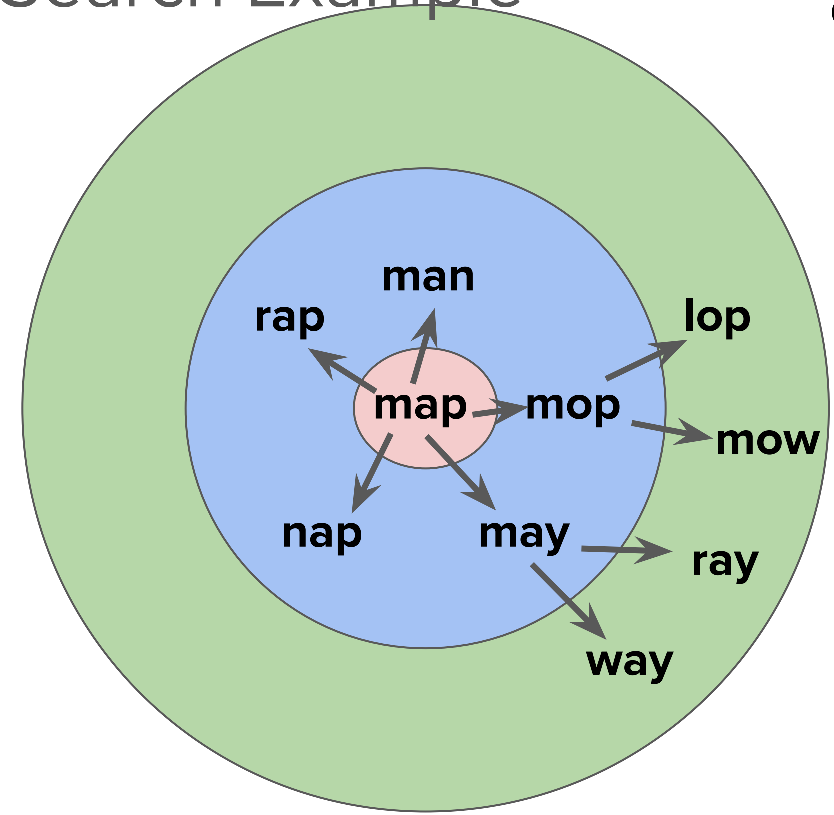
start: map
destination: way



0 steps away
1 step away
2 steps away

Breadth-First Search Example

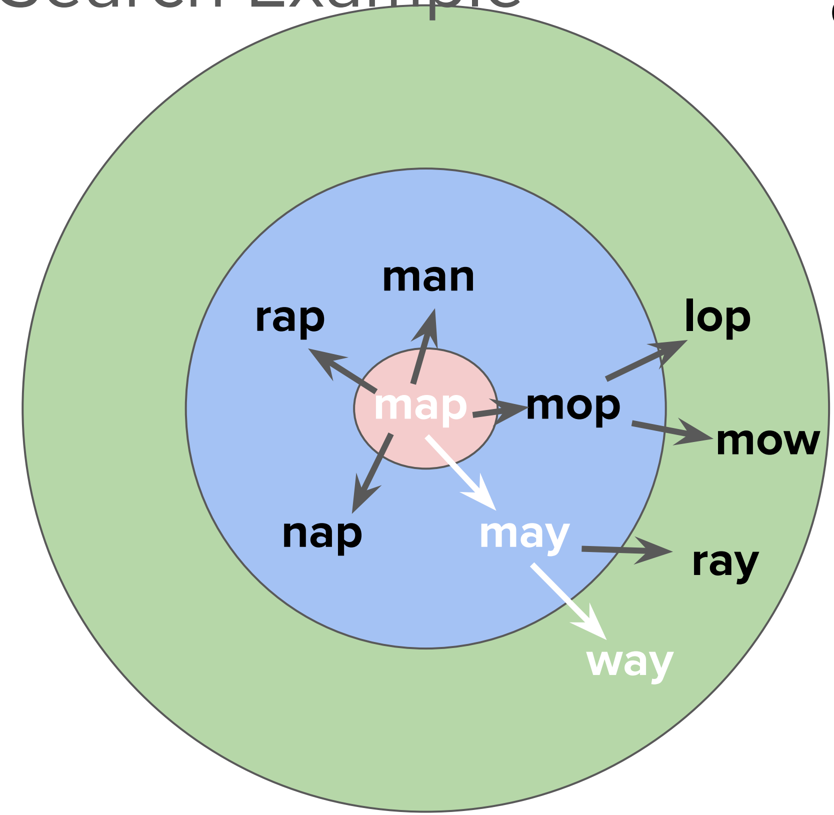
start: map
destination: way



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 Breadth-First Search (BFS)

Breadth-First Search Data Structures

We need...

1) A data structure to represent (partial word) ladders
    - Desired characteristics: can easily access the most recent word added to the word ladder

Breadth-First Search Data Structures

We need...

1) A data structure to represent (partial word) ladders
    - Desired characteristics: can easily access the most recent word added to the word ladder

2) A data structure to store all the partial word ladders that we have generated so far and have yet to explore
    - Desired characteristics: can maintain an ordering of partial word ladders so that all ladders of a certain length get explored before ladders of longer length get explored

Breadth-First Search Data Structures

We need...

1) A data structure to represent (partial word) ladders
   - Desired characteristics: can easily access the most recent word added to the word ladder
2) A data structure to store all the partial word ladders that we have generated so far and have yet to explore
   - Desired characteristics: can maintain an ordering of partial word ladders so that all ladders of a certain length get explored before ladders of longer length get explored
3) 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: can check quickly whether a word has been seen before

Breadth-First Search Data Structures

We need...

1) A data structure to represent (partial word) ladders
   - Desired characteristics: can easily access the most recent word added to the word ladder
2) A data structure to store all the partial word ladders that we have generated so far and have yet to explore
   - Desired characteristics: can maintain an ordering of partial word ladders so that all ladders of a certain length get explored before ladders of longer length get explored
3) 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: can check quickly whether a word has been seen before

Breadth-First Search Data Structures

We need...

1) A data structure to represent (partial word) ladders
   - **`Stack<string>`**
2) A data structure to store all the partial word ladders that we have generated so far and have yet to explore
   - Desired characteristics: can maintain an ordering of partial word ladders so that all ladders of a certain length get explored before ladders of longer length get explored
3) 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: can check quickly whether a word has been seen before

Breadth-First Search Data Structures

We need...

1) A data structure to represent (partial word) ladders
    - **`Stack<string>`**
2) A data structure to store all the partial word ladders that we have generated so far and have yet to explore
    - **`Queue<Stack<string>>`**
3) 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: can check quickly whether 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 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 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
```

Implementing Breadth-First Search

[Qt Creator]

Implementing Breadth-First Search

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.

Announcements

Announcements

- Assignment 2 was released last night. It will be due at the end of the day on **Wednesday, July 7**.
- YEAH will be tomorrow, 7/1 at 7pm PT. Link is on the course website on the zoom info page.
- Check out the A2 warmup to ensure that your Qt debugger works nicely with the Stanford C++ collections ***before*** starting on the assignment.
- This assignment is a step-up in complexity compared to A1 – get started early!

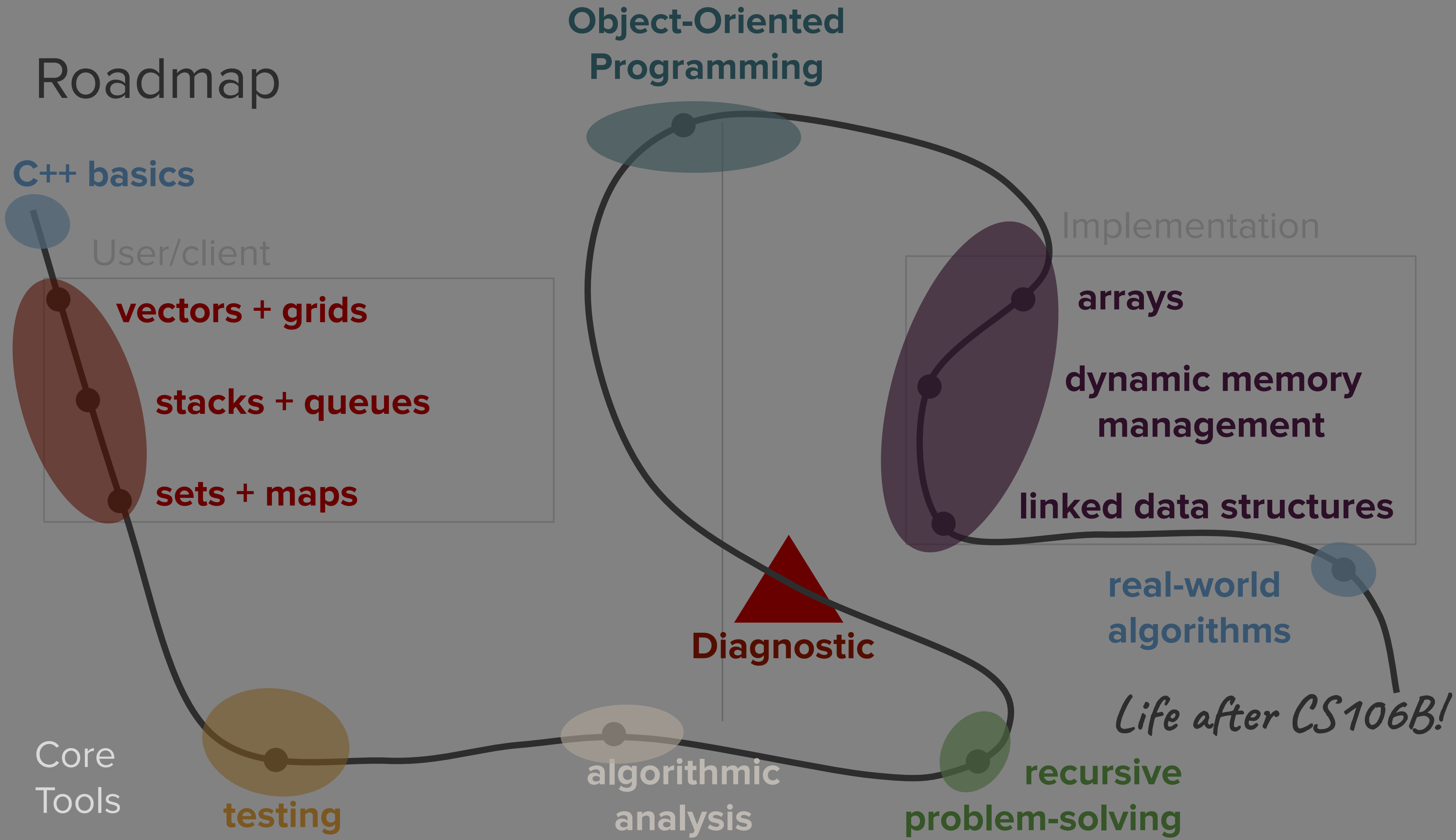
Goals for this Course

Learn how to model and solve complex problems with computers.

- Explore common abstractions for representing problems.
- Harness recursion and understand how to think about problems recursively.
- Quantitatively analyze different approaches for solving problems.

What’s next?

Roadmap



Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Big O and Algorithmic Analysis