Lab 11: Sorting and Optimization

CS 0445: Data Structures

TAs: Jon Rutkauskas
Brian Nixon

http://db.cs.pitt.edu/courses/cs0445/current.term/

December 2, 2019
University of Pittsburgh, Pittsburgh, PA

Quicksort

- Standard quicksort is a (**practically** speaking) fast sorting algorithm
  - Average/Best case: **O(n log n)**
    - The array is split in half (or nearly in half) at each step
  - Worst case: **O(n^2)**
    - The result of poor pivot choice (max/min element)
- How can we improve quicksort?

Optimizing Quicksort

1. Pivot choice
   - randomized, median of threes, dual-pivot, etc.
2. Practical choice of base case (the goal of this lab)
   - Divide and conquer is less effective for small input size (**recursive overhead** can be costly)
   - Insertion sort can be used to optimize for small input
   - You will determine how *small* the input should be in this lab

How does insertion sort help us?

- Recursive overhead of quicksort is more costly than the benefits of divide and conquer on small arrays
- **Fix:** break down a size **n** array into base case of size **k** via quicksort, then pass the nearly sorted array to insertion sort
  - Property of insertion sort (**Adaptive**): If each element in an array is no more than **k** steps from its correct position, then the runtime complexity of insertion sort is **O(kn)**
    - i.e it is efficient for arrays that are substantially sorted
    - This is exactly the case if we chose a **smart** base case for quicksort!

Insertion Sort: Algorithm

- Builds the final sorted array **one item at a time**
- Considers the array as a union of **sorted/unsorted portions**
  - **Idea:**
    - On each iteration until sorted
      - Take an item from the **unsorted** portion
      - Place this item in its **correct** position in the **sorted** portion of the array

Insertion Sort: Example

6	4	9	5	3	1

- Consider the first element to be in the sorted portion (green)
- Consider the rest to be unsorted (red)
- Assume we are sorting in ascending order

Insertion Sort: Example

6	4	9	5	3	1

- Consider the first element in the unsorted portion (blue)
- Compare it to the element(s) of the sorted portion
  - Where should it be placed?
    - Potential rule to follow: Is the element to my left larger than me? If so, we swap.
    - Above: 4<6, so we swap.

4	6	9	5	3	1

Insertion Sort: Example

- Next iteration (repeat as before)

4	6	9	5	3	1

4	6	9	5	3	1

No need to swap

- Result

4	6	9	5	3	1

Insertion Sort: Example

- Next iteration (repeat as before)

4	6	9	5	3	1
4	6	9	5	3	1

No need to swap

Swap

Swap needed

- Result

4	5	6	9	3	1

Insertion Sort: Example

- Next iteration (repeat as before)

4	5	6	9	3	1

4	5	6	9	3	1
Swap	Swap	Swap	Swap needed		

- Result

3	4	5	6	9	1

Insertion Sort: Example

- Next iteration (repeat as before)

3	4	5	6	9	1
3	4	5	6	9	1
↑ Swap	↑ Swap	↑ Swap	↑ Swap	↑ Swap needed	

- Result – Sorted Array

1	3	4	5	6	9

Quicksort: Algorithm

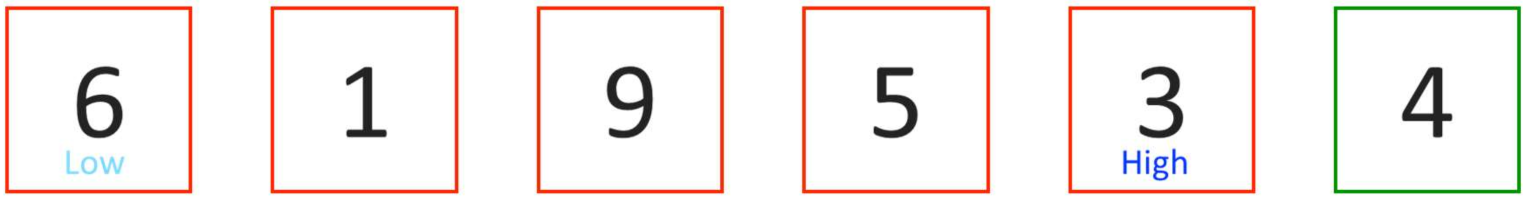
- Pick a pivot
- Partition the array into 2 sections: items higher than the pivot, and items lower than the pivot
  - This gives us two disjoint subarrays
- Place the pivot in its proper spot (between the two subarrays)
- Recurse on the two subarrays

Quicksort: Example

- Assume we sort in ascending order
- Assume we pick our pivot to be the *last* element

6	1	9	5	3	4

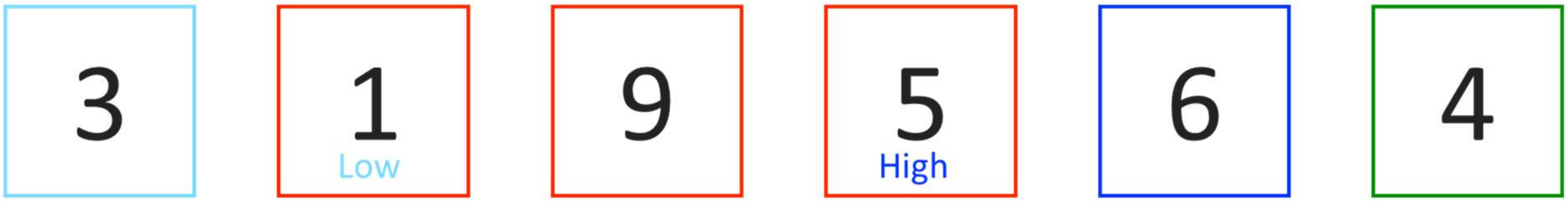
Quicksort: Example



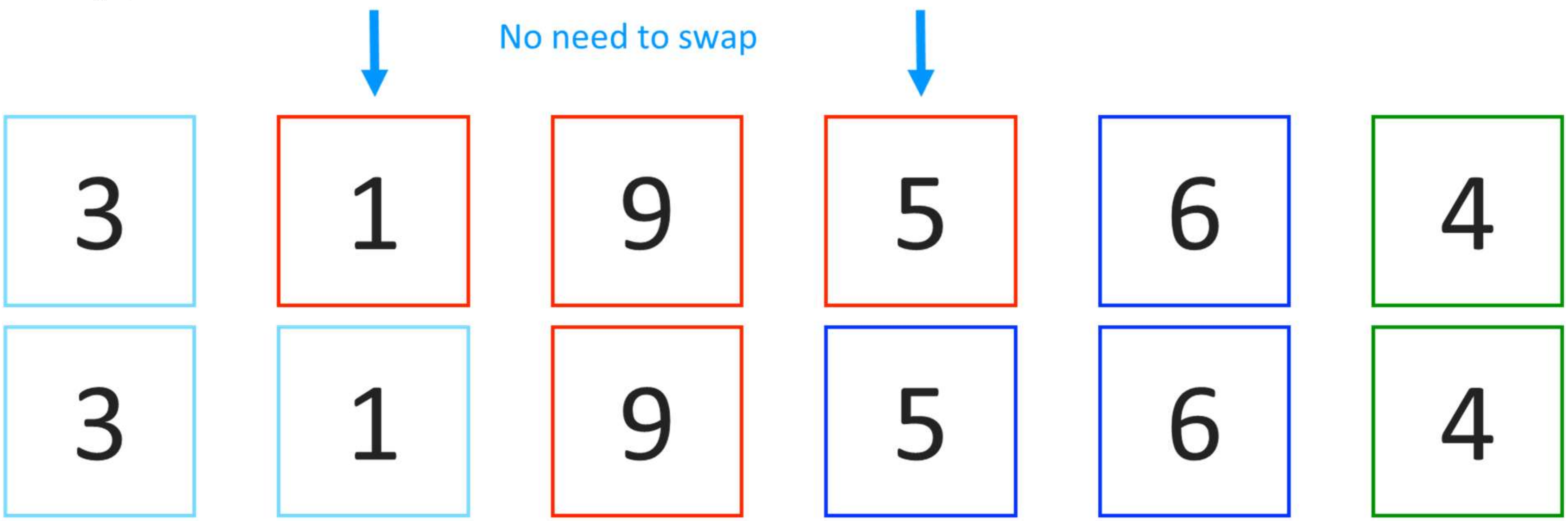
- Pivot is green element
- Compare the high-index and low-index items. Should they be swapped?

Quicksort: Example

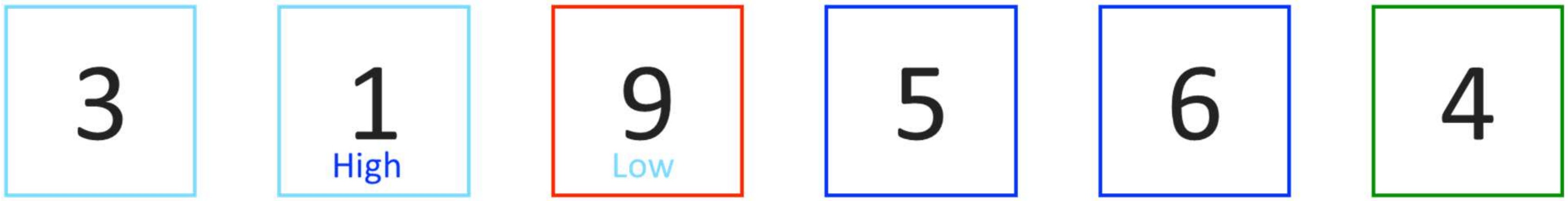
- Increment the low index, decrement the high index.



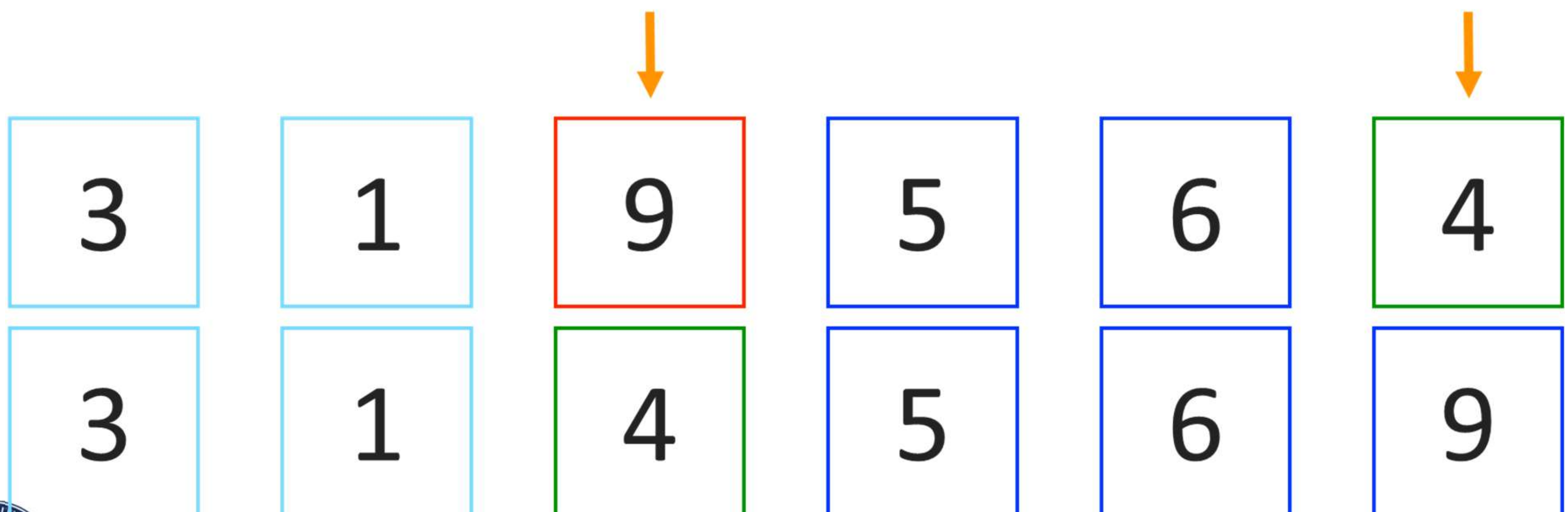
- Compare the new high item and new low item. Should they be swapped?



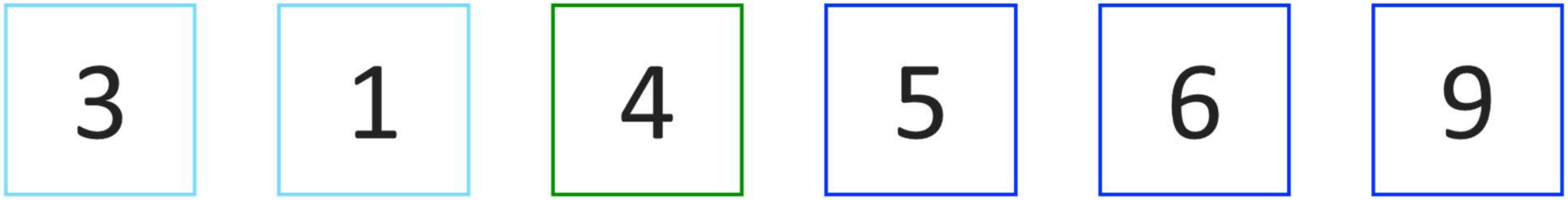
Quicksort: Example



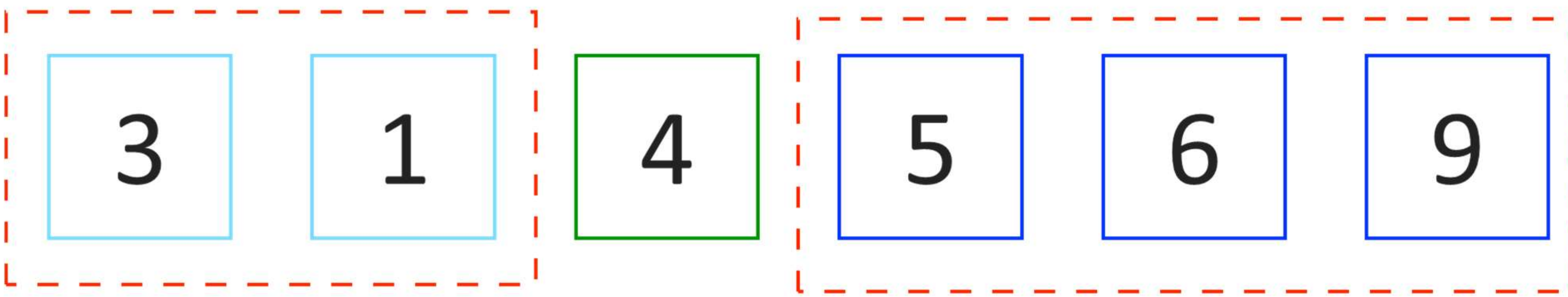
- On this iteration, our high-index and low-index will overlap. We will put the pivot element into place.



Quicksort: Example



- Now the pivot is in the correct position
- Recurse on the subarrays until base case is reached, then recombine



Repeat on low partition

Repeat on high partition

Your Tasks

- Download and read both the code and instructions carefully
  - http://db.cs.pitt.edu/courses/cs0445/current.term/
- Run **SortTiming** to determine for what **k** is insertion sort more efficient than quicksort (the specific number may depend on your individual computer)
- Implement **timeQuickSort2**
  - First, use a base case that is a clear optimization
  - Second, experiment with a larger range of base case sizes
- Draw and plot your results in Excel
  - Follow the instructions. The output will be a csv file

A note on JVM Optimizations

- Because of the way compiled Java bytecode is run, the JVM has the ability to optimize the execution of your code *as it runs*
  - That means your program can actually perform the same tasks *practically faster* as it continues to repeat the same sections of code.
- These optimizations are usually transparent to you and are usually just a pleasant surprise
- Here, because we are taking actual times, this *could* be a problem
  - You would potentially see a massive drop-off in runtime as the optimizations kick in

A note on JVM Optimizations

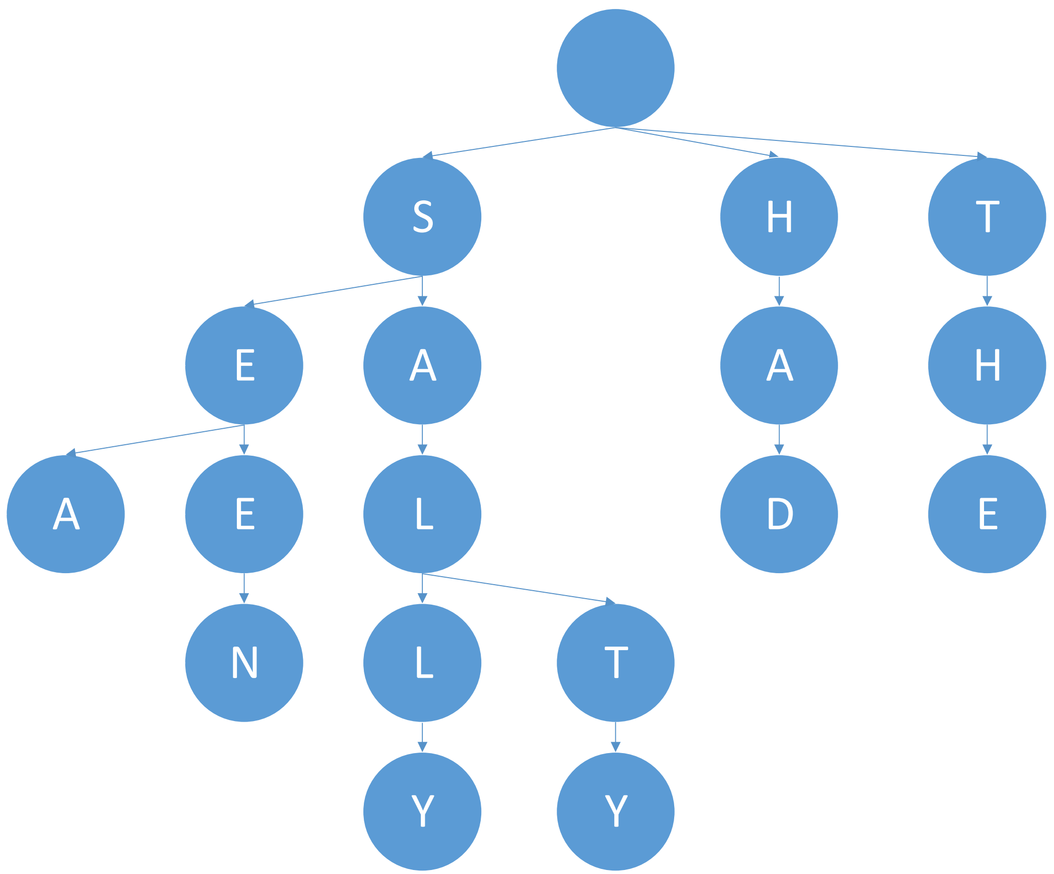
- If this happens to you, fear not; your timing data is not useless!
- All we need to do is run the sorts over and over *before* we actually start measuring timings
  - That way, the JVM optimizations will have already kicked in
- In code, a very simple fix.
  - Something along the lines of:

```
a = buildArray(250);
for (int i = 0; i < 1000; i++) {
    timeInsertionSort(numTrials);
    timeQuickSort(numTrials);
}
```

  - Because every individual computer will run differently, the actual time it takes for the optimizations to start and the actual benefit of them will differ from person-to-person

An Additional Note on Tries – A special type of tree

- Tries are a unique type of tree that stores the value **along the path down the tree** instead of storing the key in the tree.
- There are no values in the inner part of the tree, we only use them to build up the value at the end
- Example of a trie with the strings: ‘sally’ ‘had’ ‘seen’ ‘the’ ‘salty’ ‘sea’

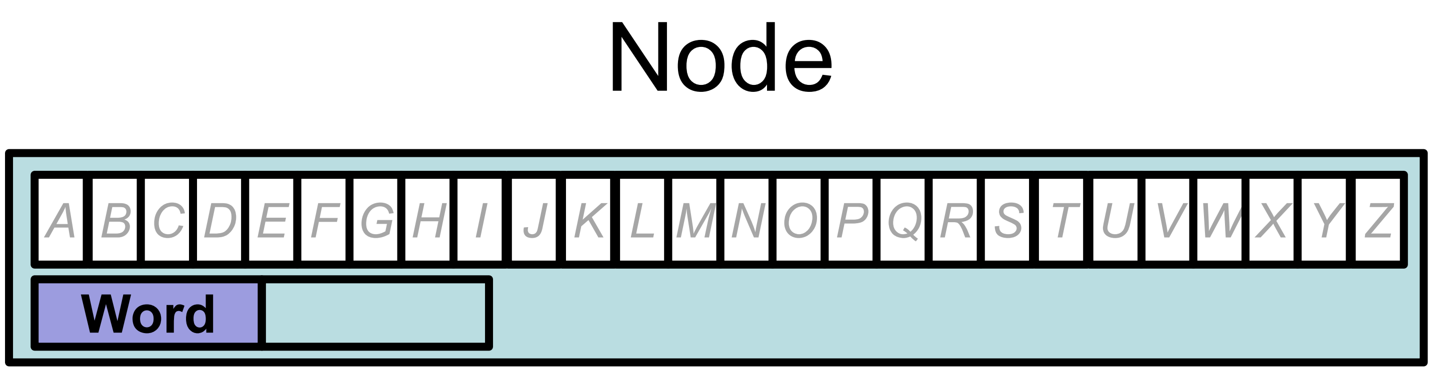


Implementing the trie

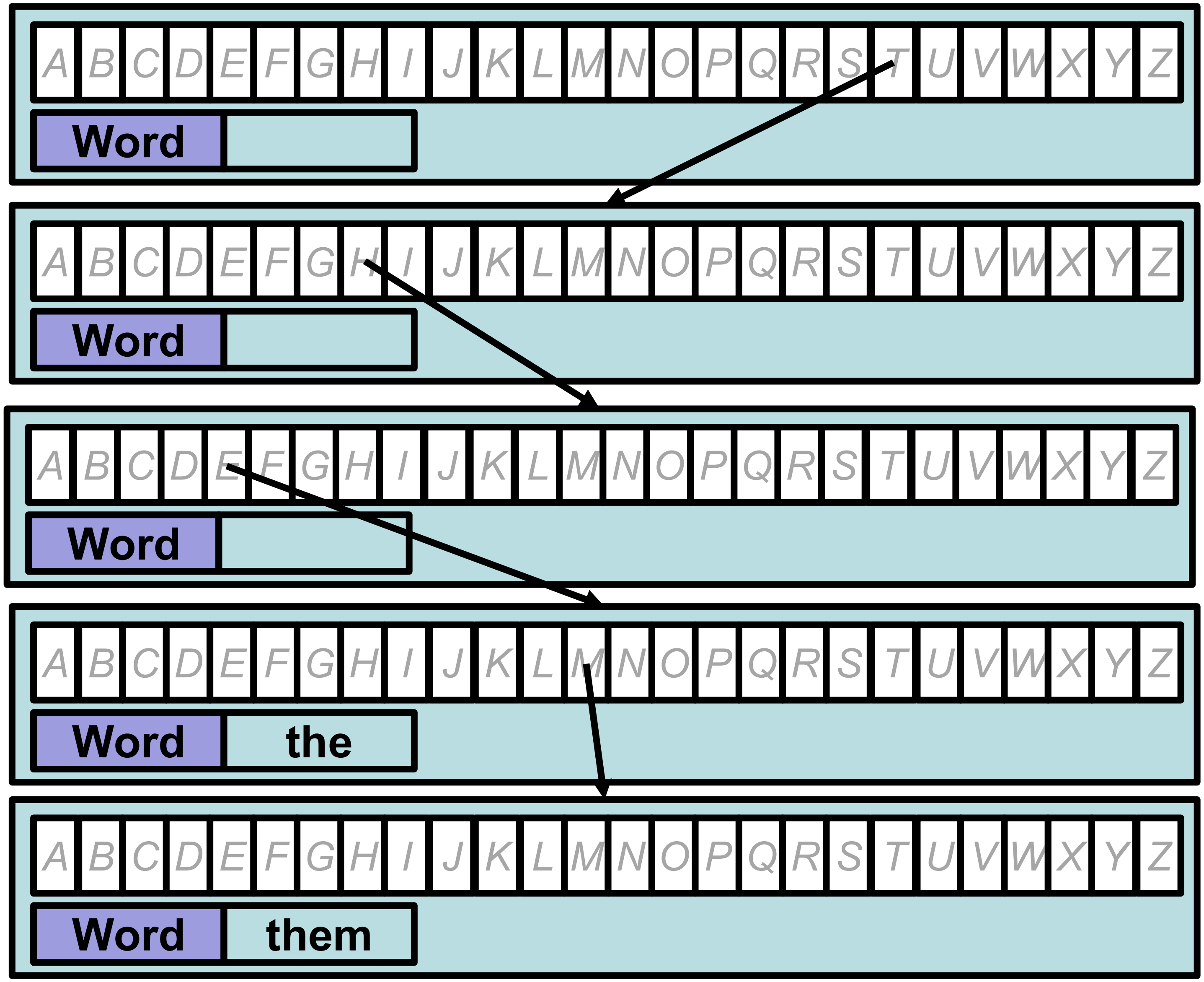
- One method of implementing is by having an array of Nodes of length 26 in *each* node (*Why 26?*)

```
private class Node {
        Node[] links = new Node[26];
        String word;
    }
```

- To specify that a node denotes the end of a word (and specify the last letter), we can set a special value to it: *the word we have found*
- Simple example: “the” “them”



“the” “them”


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
Word
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
Word
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
Word
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
Word the
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
Word them