Announcements

- Devin Balkcom (Undergraduate Advisor) — visiting on Monday
- Last lab hours — Sunday, March 8th
- Next week's recitation sections are optional. Your recitation section will be used for extra time to answer questions, review practice problems, etc.
- Practice problems/solutions for Final Exam — to be posted later today or tomorrow.
- More announcements coming on Canvas regarding…
  - Updated office hours (next week)
  - Exam review time
  - Final exam

But first, I'll teach you some maths that you've likely never seen before…

Prove 1 = .9999…

```
Let        x = 0.9999…
          10x = 9.9999…
        -  x = 0.9999…
           ------------
           9x = 9 ==>  x = 1  ==>  x = 1 = 0.9999…
```

□

Prove 9999… = -1

```
Let          x = 9999…
       (1/10)x = (1/10)*9999…
       (1/10)x = 9999… .9
          -  x = 9999…
             ------------
      (-9/10)x = .9
      (-9/10)x = (9/10)
             x = (9/10)(-10/9)
             x = -1
```

□

String Finding

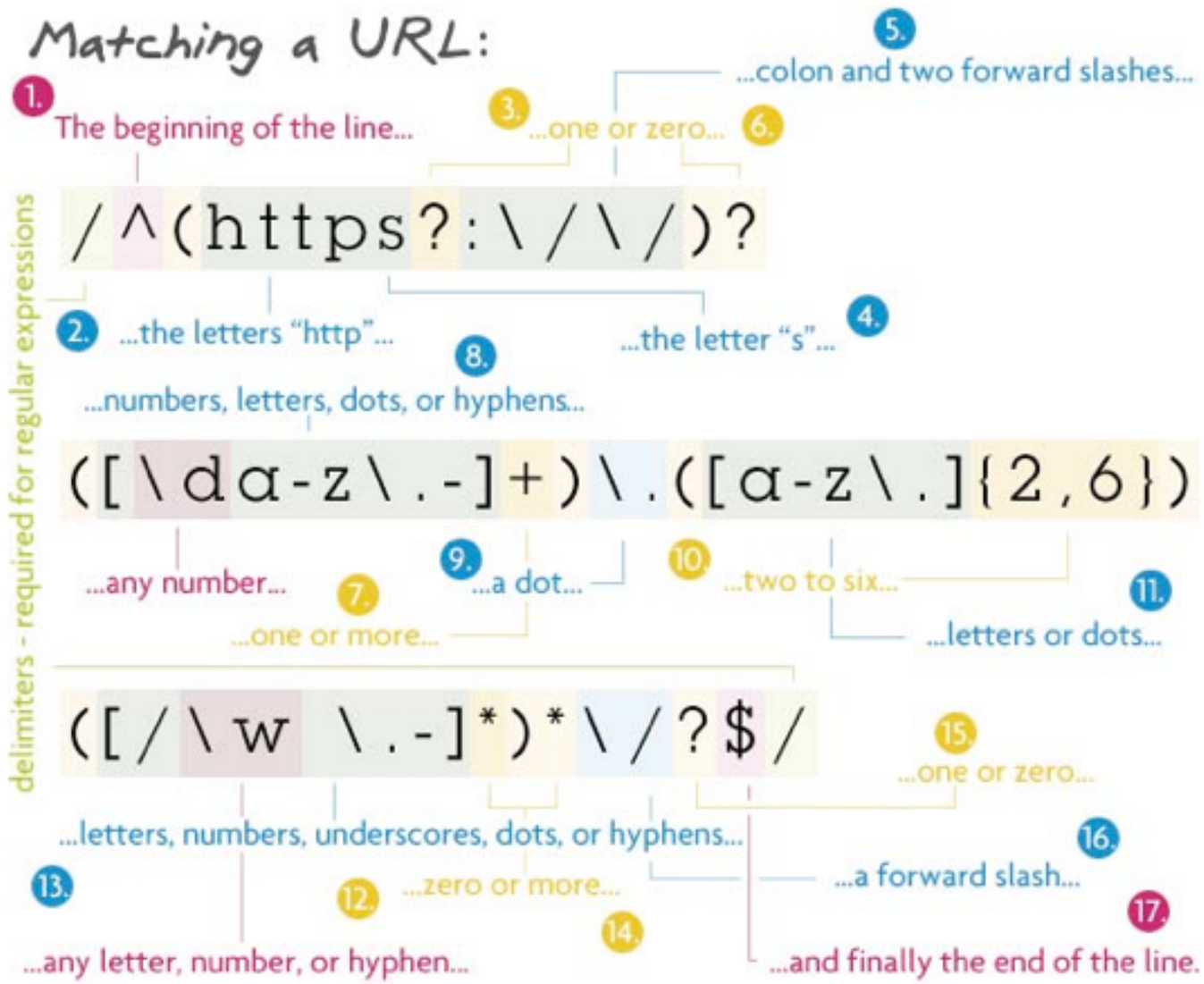
Matching Strings and Substrings

String Finding: Overview

- Matching/recognizing patterns in sequences is a very relevant problem in CS

String Finding: Overview

- Matching/recognizing patterns in sequences is a very relevant problem in CS
- **Regular Expressions** — “regex” — used to find seq. of characters in larger text
    - ex. think “find/replace” in text editors/word documents



String Finding: Overview

- Matching/recognizing patterns in sequences is a very relevant problem in CS
- **DNA Sequencing**
  - ex. find GAGATGCTCCAGAAC in

AGGACGCCGCATTGACCATCTATGAGATGCTCCAGAACATCTTTGCTATTTTCAG
ACAAGATTCATCTAGCACTGGCTGGAATGAGACTATTGTTGAGAACCTCCTGGCT
AATGTCTATCATCAGATAAACCATCTGAAGACAGTCCTGGAAGAAAAACTGGAGA
AAGAAGATTTCACCAGGGGAAAACTCATGAGCAGTCTGCACCTGAAAAGATATTA
ATGACCAACAAGTGTCTCCTCCAAATTGCTCTCCTGTTGTGCTTCTCCACTACAG
CTCTTTCCATGAGCTACAACTTGCTTGGATTCCTACAAAGAAGCAGCAATTTTCA
GTGTCAGAAGCTCCTGTGGCAATTGAATGGGAGGCTTGAATACTGCCTCAAGCAC
AGGATGAACTTTGACATCCCTGAGGAGATTAAGCAGCTGCAGCAGTTCCAGAAGG
ATGACCAACAAGTGTCTCCTCCAAATTGCTCTCCTGTTGTGCTTCTCCACTACAG
CTCTTTCCATGAGCTACAACTTGCTTGGATTCCTACAAAGAAGCAGCAATTTTCA
GTGTCAGAAGCTCCTGTGGCAATTGAATGGGAGGCTTGAATACTGCCTCAAGCAC
AGGATGAACTTTGACATCCCTGAGGAGATTAAGCAGCTGCAGCAGTTCCAGAAGG
AGGACGCCGCATTGACCATCTATGAGATGCTCCAGAACATCTTTGCTATTTTCAG
ACAAGATTCATCTAGCACTGGCTGGAATGAGACTATTGTTGAGAACCTCCTGGCT
AATGTCTATCATCAGATAAACCATCTGAAGACAGTCCTGGAAGAAAAACTGGAGA
AAGAAGATTTCACCAGGGGAAAACTCATGAGCAGTCTGCACCTGAAAAGATATTA
TGGGAGGATTCTGCATTACCTGAAGGCCAAGGAGTACAGTCACTGTGCCTGGACC
ATAGTCAGAGTGGAAATCCTAAGGAACTTTTACTTCATTAACAGACTTACAGGTT
AGGACGCCGCATTGACCATCTATGAGATGCTCCAGAACATCTTTGCTATTTTCAG
ACAAGATTCATCTAGCACTGGCTGGAATGAGACTATTGTTGAGAACCTCCTGGCT
AATGTCTATCATCAGATAAACCATCTGAAGACAGTCCTGGAAGAAAAACTGGAGA
AAGAAGATTTCACCAGGGGAAAACTCATGAGCAGTCTGCACCTGAAAAGATATTA
TGGGAGGATTCTGCATTACCTGAAGGCCAAGGAGTACAGTCACTGTGCCTGGACC
ATAGTCAGAGTGGAAATCCTAAGGAACTTTTACTTCATTAACAGACTTACAGGTT

String Finding: Overview

- Matching/recognizing patterns in sequences is a very relevant problem in CS
- We will look at different ways to solve the string/substring matching problem:
  - Boyer-Moore (algorithm)
  - Tries (data structure)
  - Suffix Trees (data structure)

Finding Substrings: Brute-Force

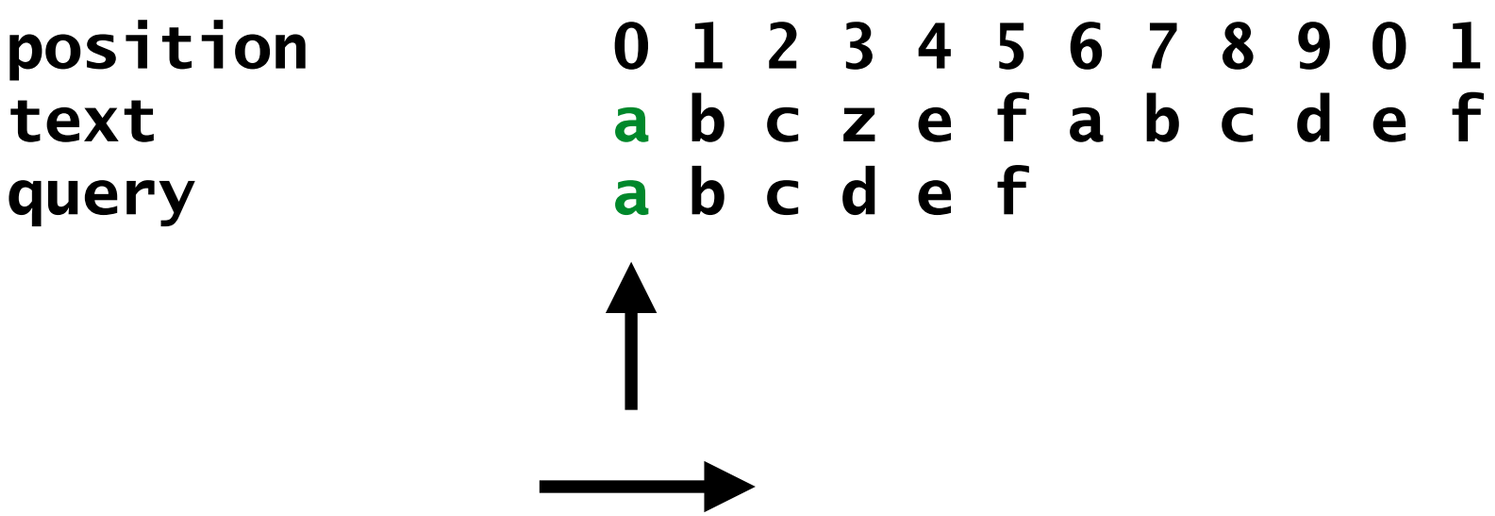
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***

```
position        0 1 2 3 4 5 6 7 8 9 0 1
text            a b c z e f a b c d e f
query           a b c d e f
```

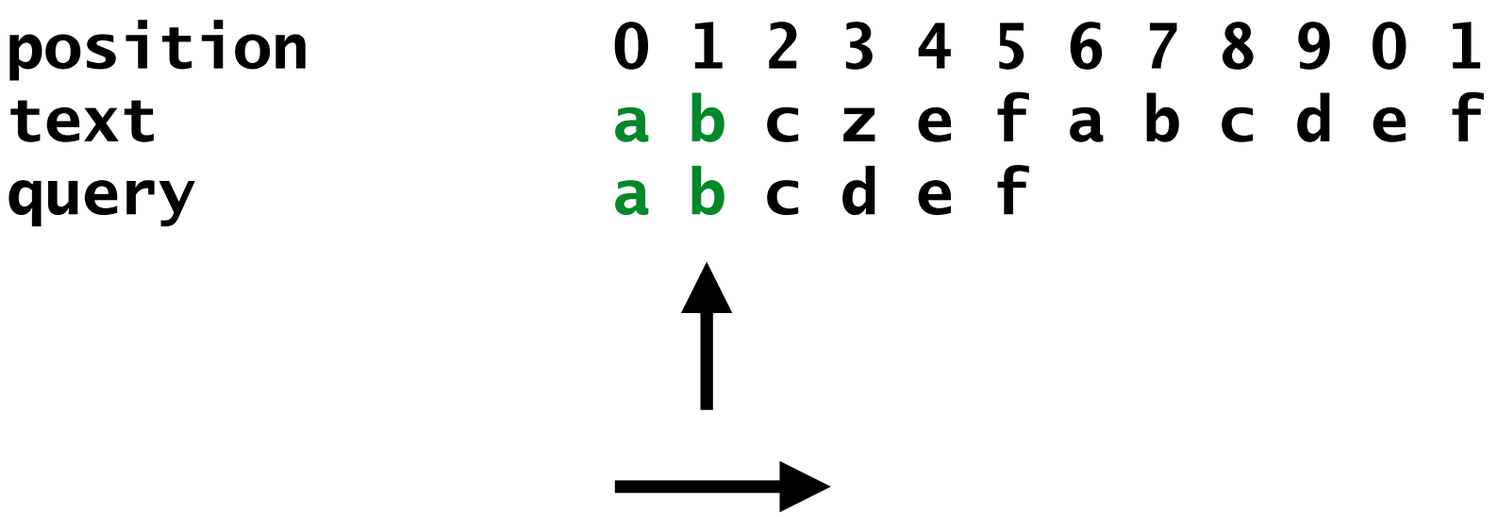
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



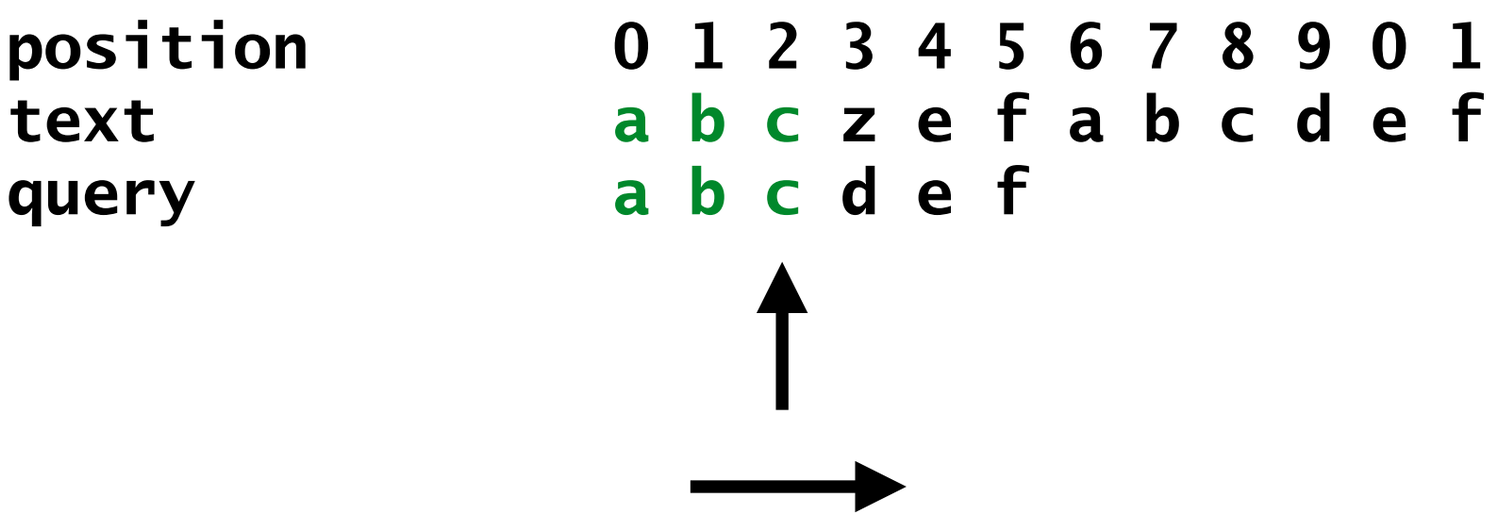
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



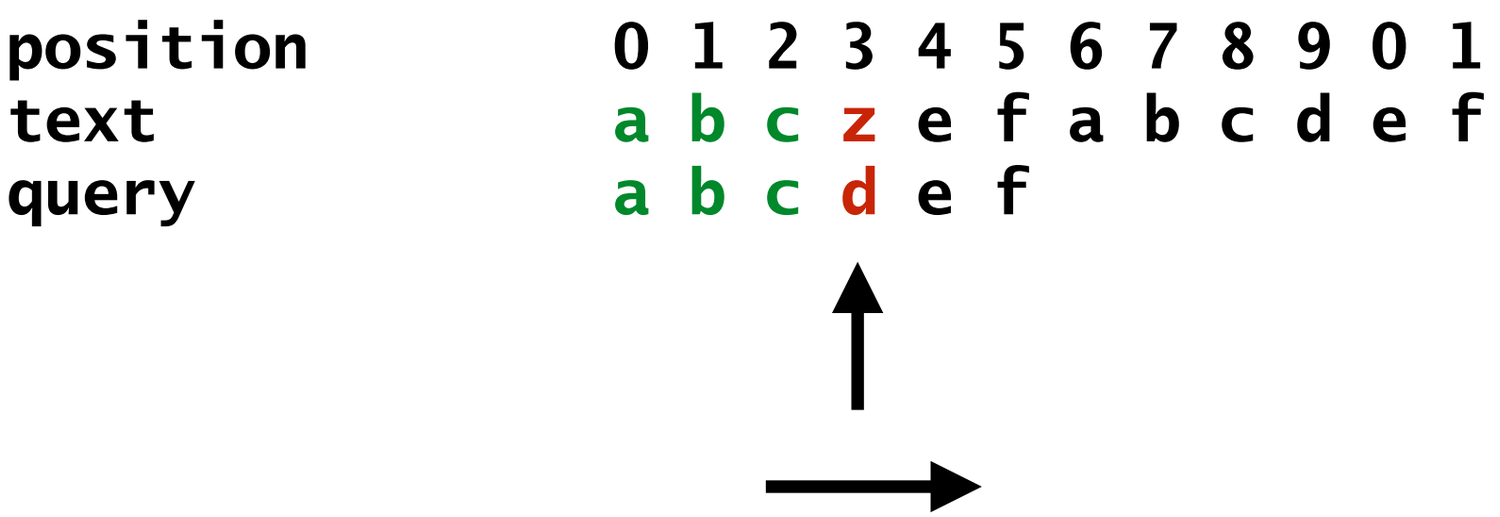
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



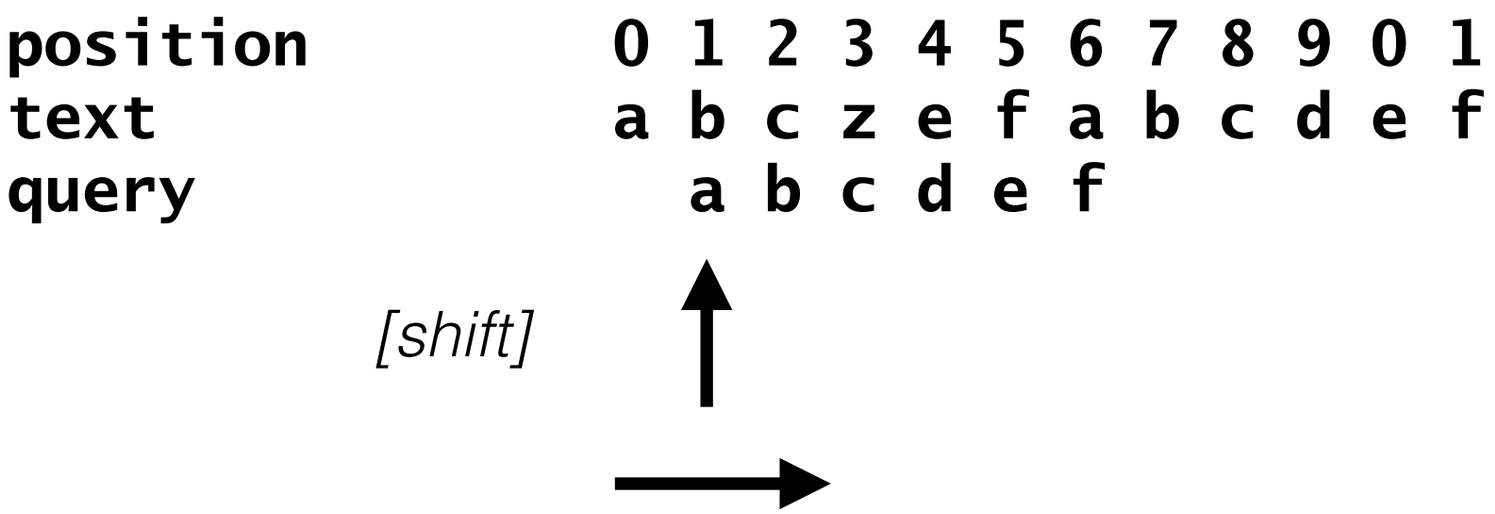
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



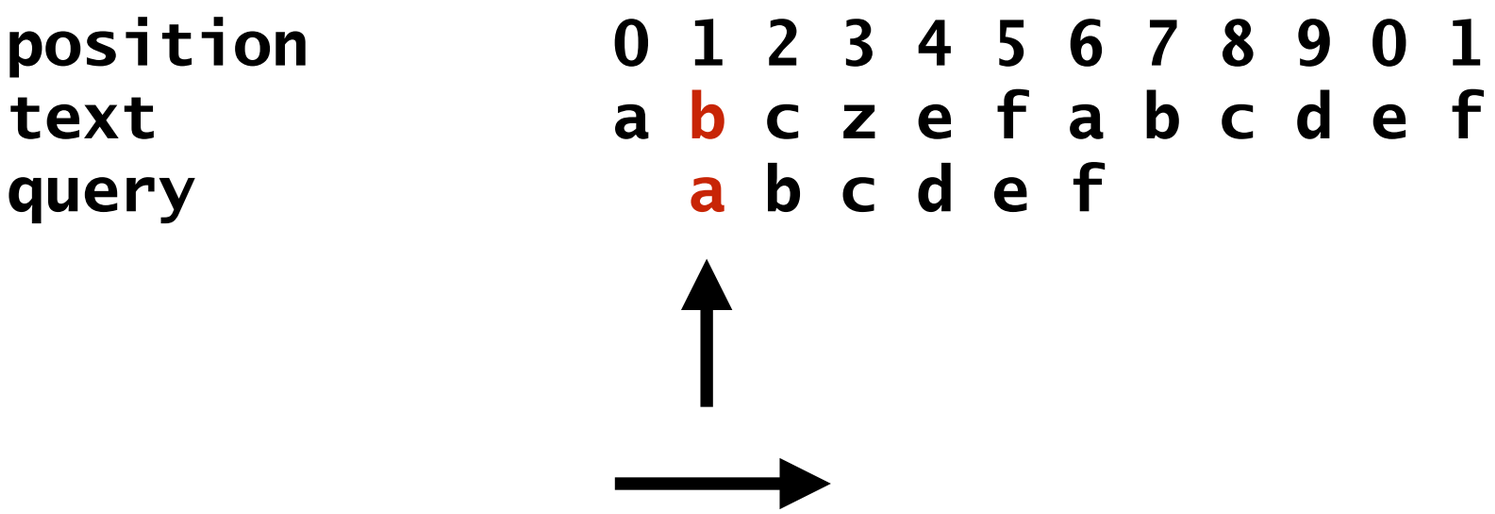
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



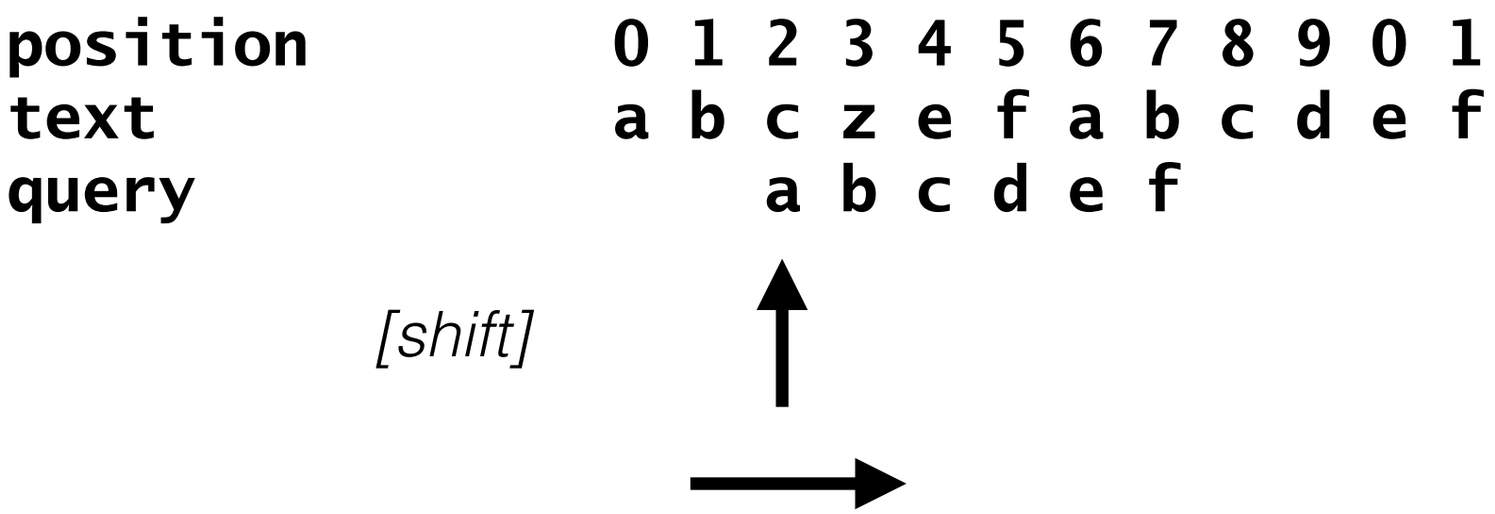
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



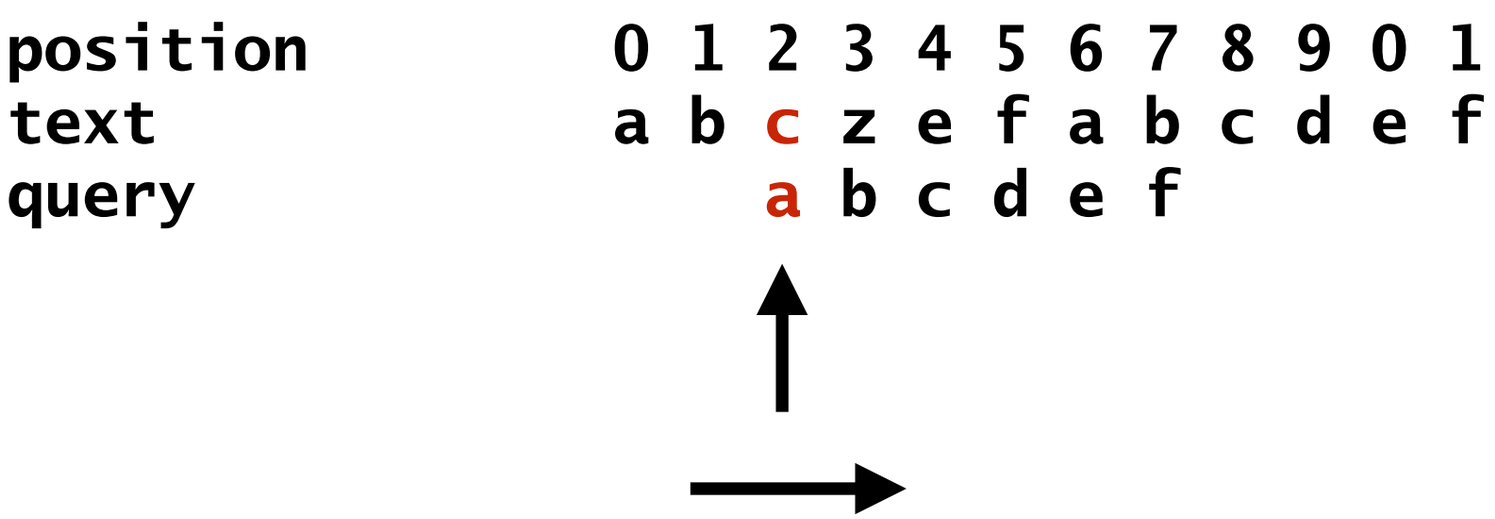
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



Finding Strings: Brute-Force

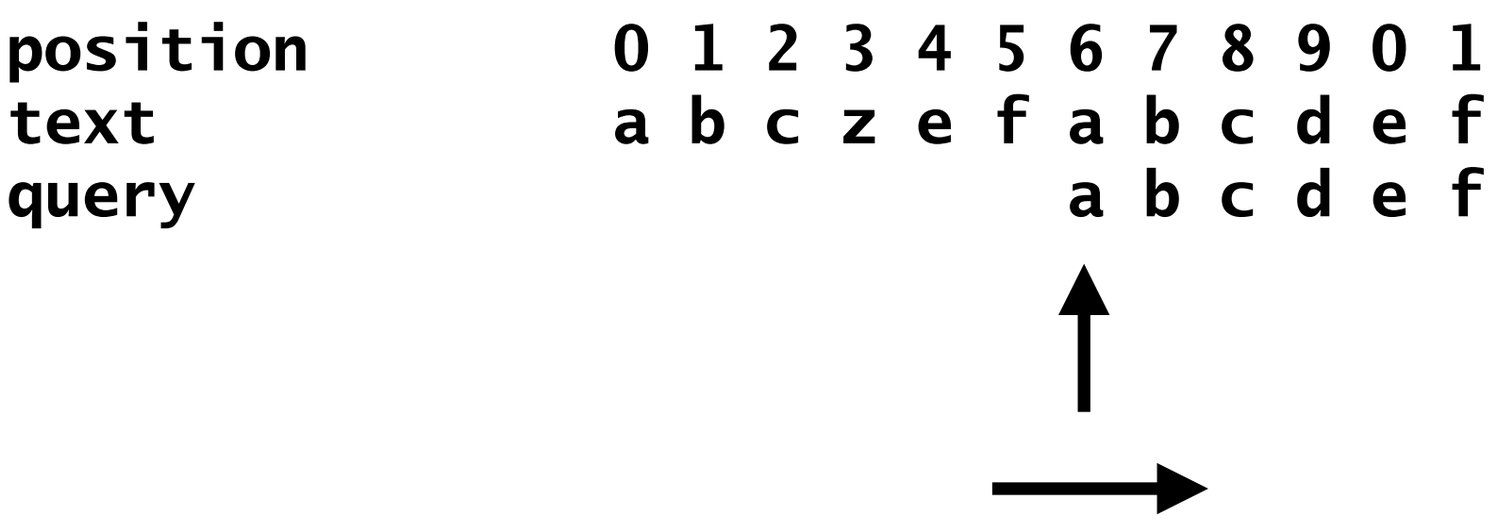
- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***

```
position        0 1 2 3 4 5 6 7 8 9 0 1
text            a b c z e f a b c d e f
query                 ...
```

[more mismatches & shifting]

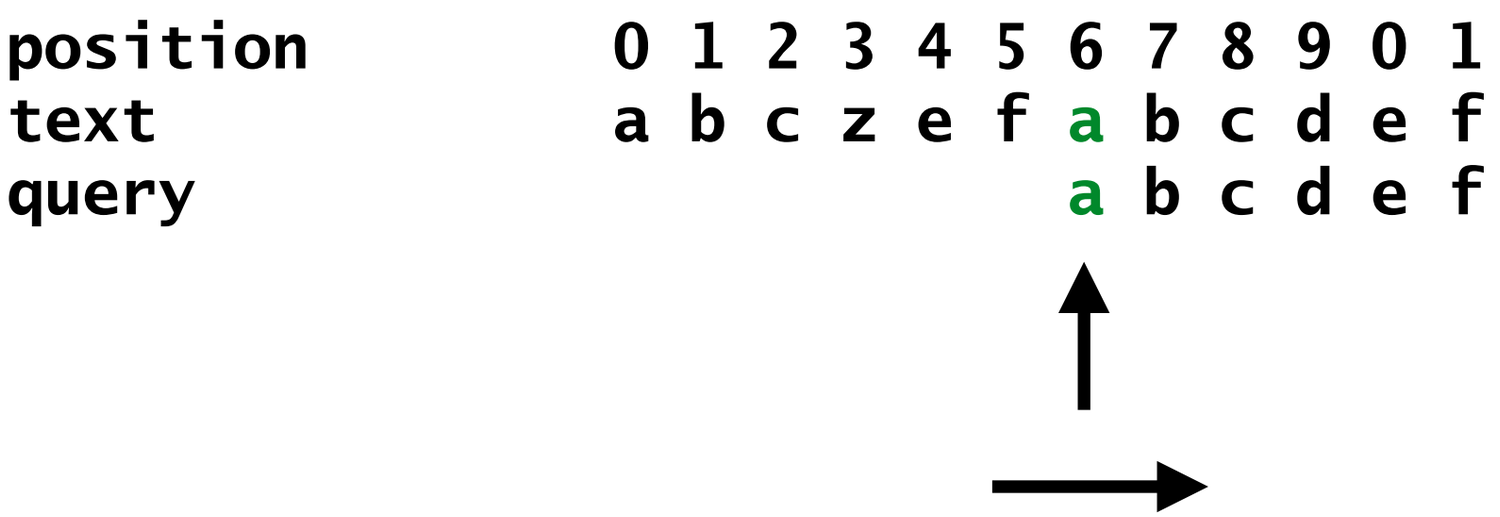
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



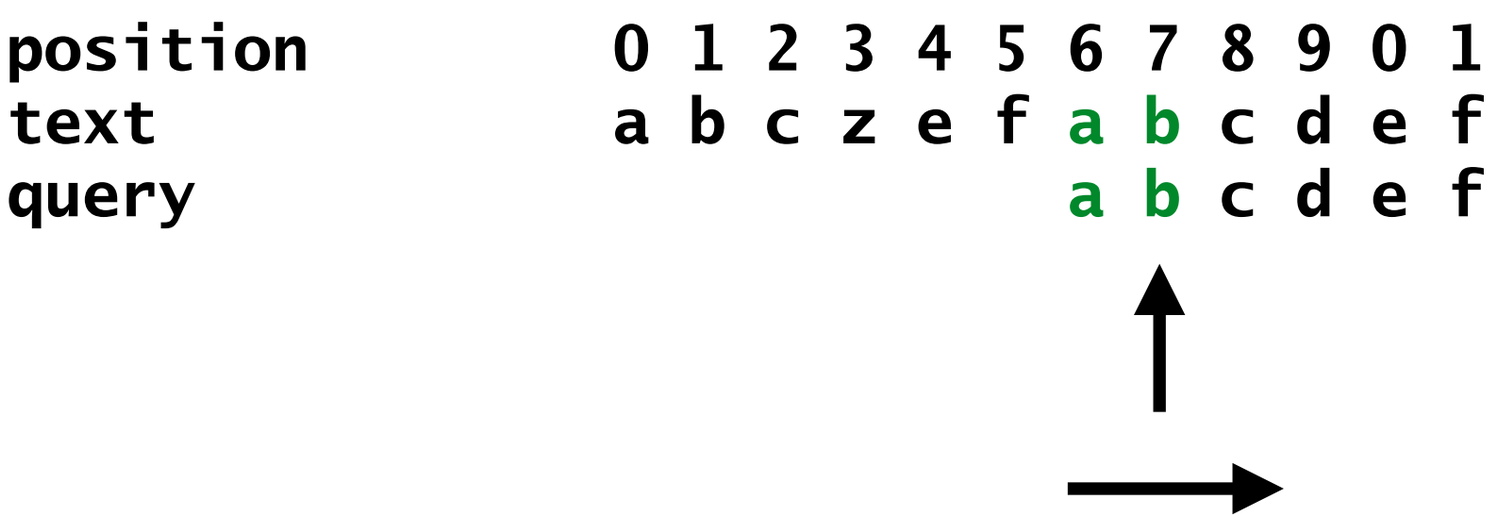
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



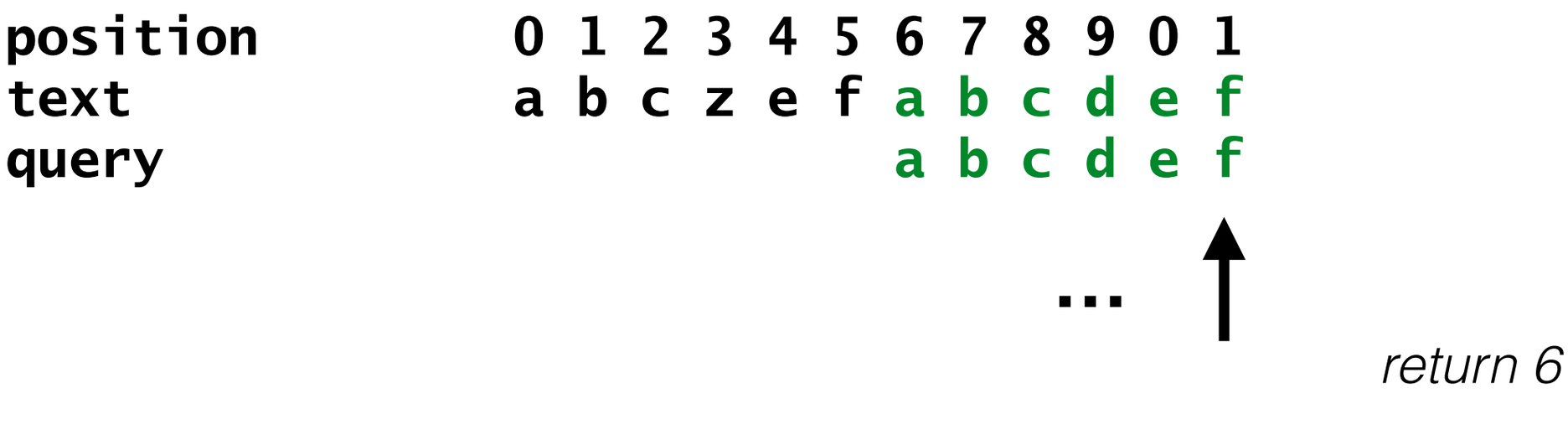
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



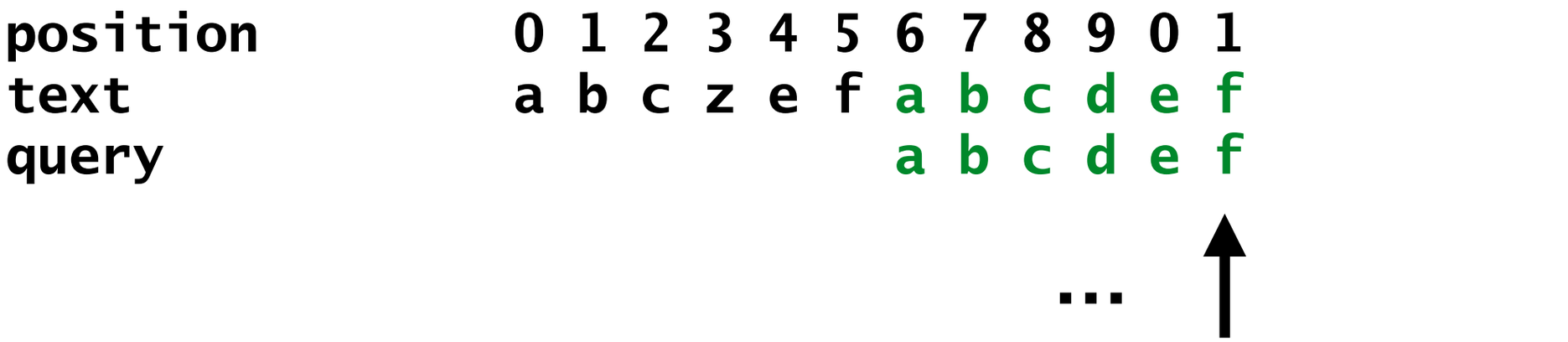
Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***



Finding Strings: Brute-Force

- Given two strings:
  - Text — generally a *large* string
  - Query — generally a *shorter* string
- **Question:** Is the query string somewhere in the text string?
- Java's **indexOf()** method does this…
  - ex. "abcdef".indexOf("cde") ==> 2
  - ex. "abcdef".indexOf("xyz") ==> -1 (not found)
- … using a (shameful) naive, brute-force approach…
  - at each position in *text,* try to match the *query* there.
  - ***[demo]***
- If text has length *n* and query has length *m* — running time *O(mn)*
  - *try matching all m query characters starting at all n - m + 1 position in text*
  - *we assume n > m*



Finding Substrings: Boyer-Moore Algorithm

Finding Strings: Boyer-Moore Alg.

- A whole mess of diff. algorithms have been developed to do better than brute-force.
- Boyer-Moore is once such alg. that we will look at today.
  - More efficient — *O(m + n) = O(n)* (assuming n > m)
  - Pretty awesome!
- We will look at a more basic version that doesn't achieve linear running time, but it works quite well *and* it gets at some of the intuitions we need to improve our approach to performing string/substring matching.

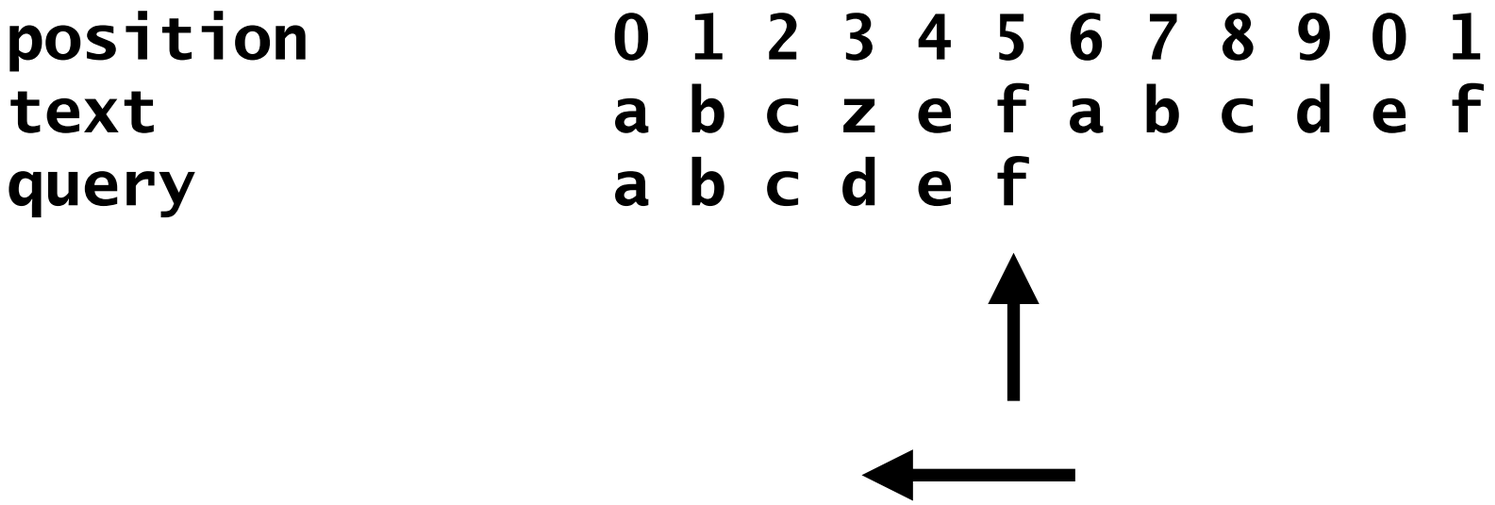
Finding Strings: Boyer-Moore Alg.

- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.

```
position        0 1 2 3 4 5 6 7 8 9 0 1
text            a b c z e f a b c d e f
query           a b c d e f
```

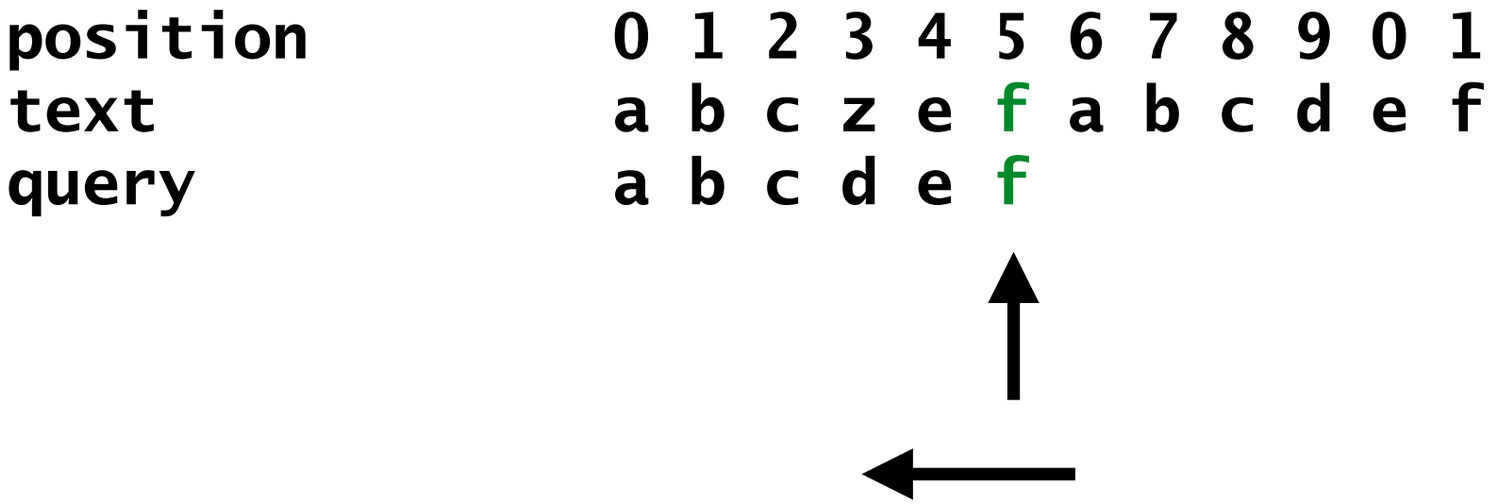
Finding Strings: Boyer-Moore Alg.

- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - Work *backwards* through the query when trying to match in the text



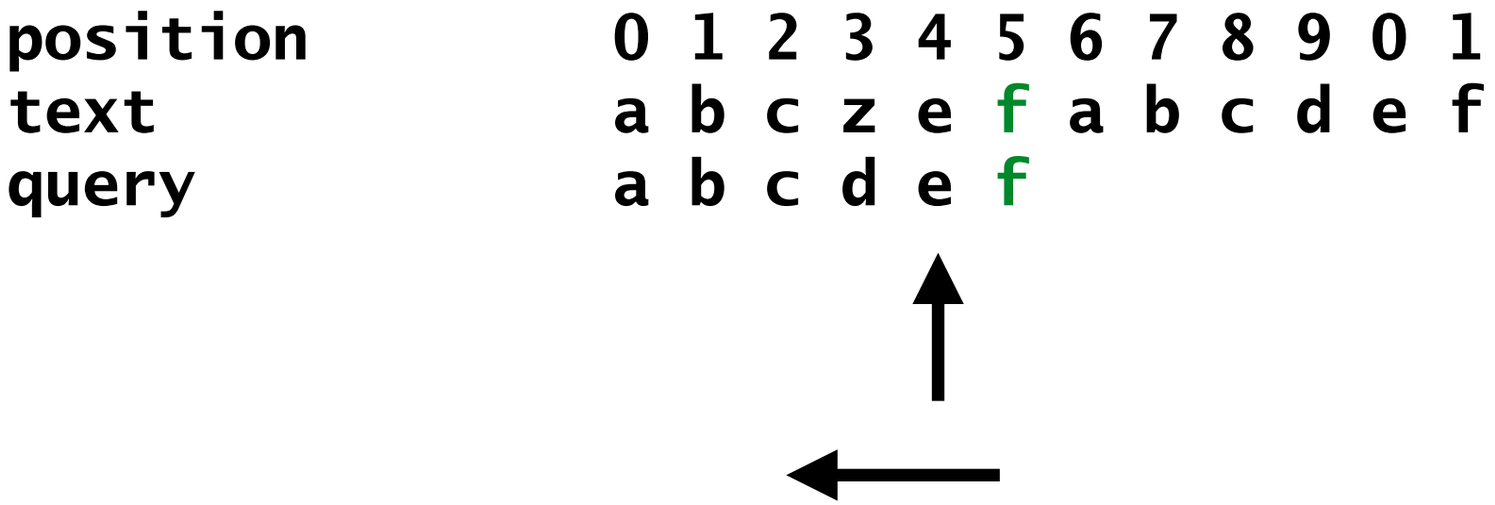
Finding Strings: Boyer-Moore Alg.

- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - Work *backwards* through the query when trying to match in the text



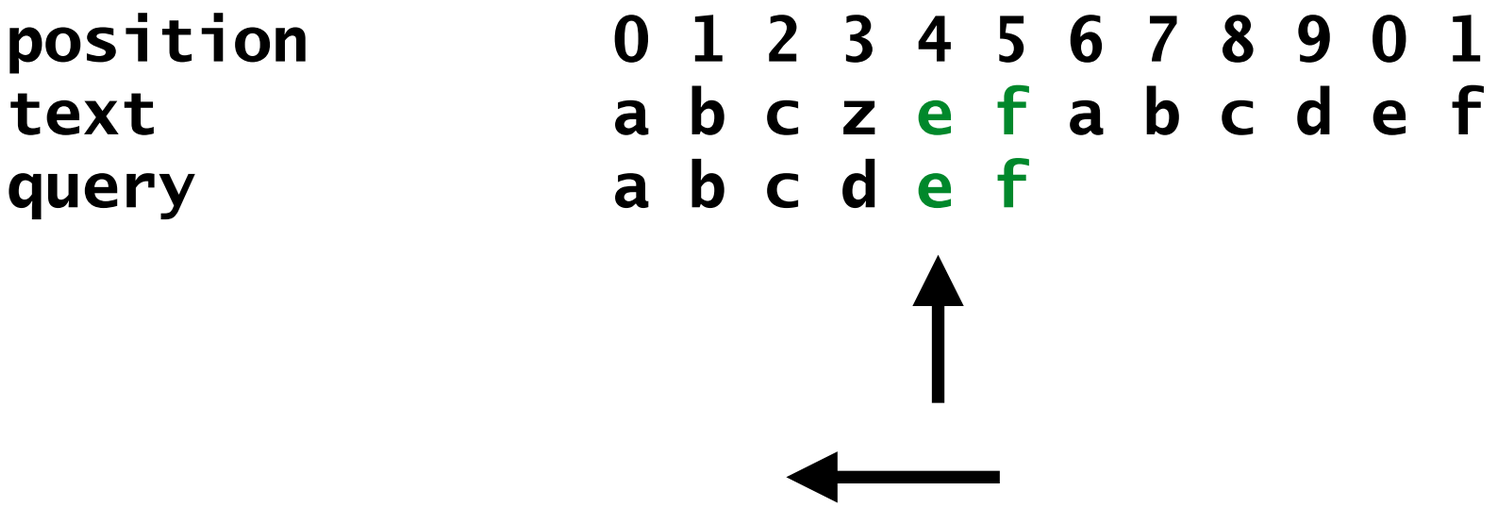
Finding Strings: Boyer-Moore Alg.

- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - Work *backwards* through the query when trying to match in the text



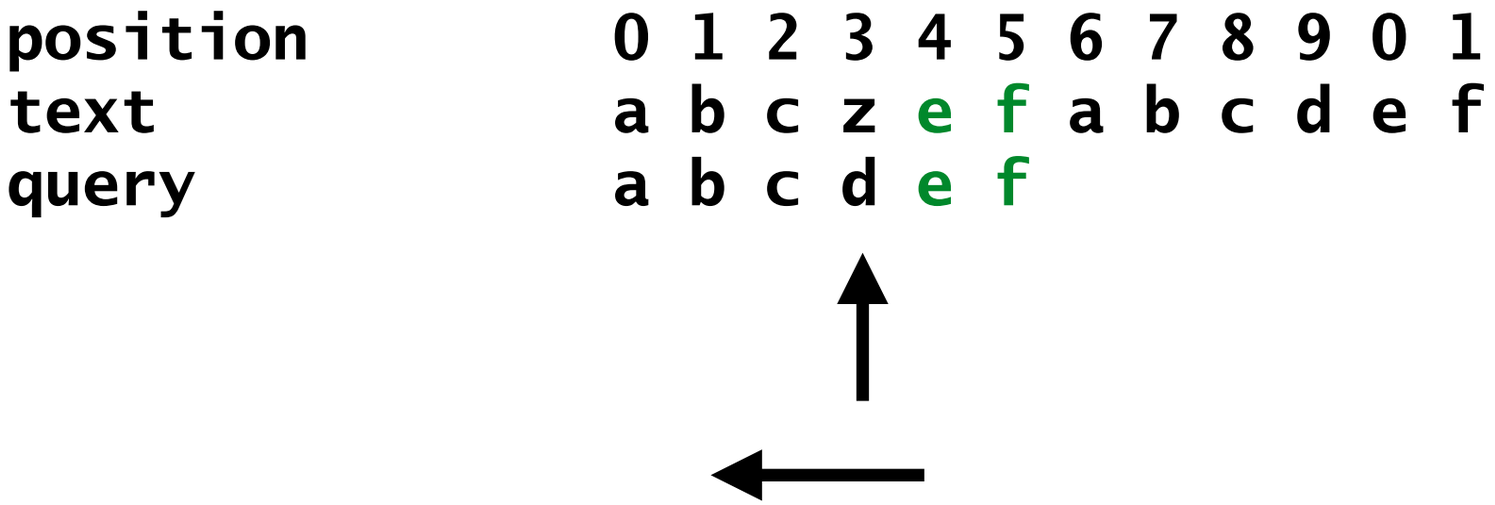
Finding Strings: Boyer-Moore Alg.

- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - Work *backwards* through the query when trying to match in the text



Finding Strings: Boyer-Moore Alg.

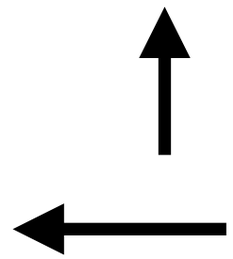
- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - Work *backwards* through the query when trying to match in the text



Finding Strings: Boyer-Moore Alg.

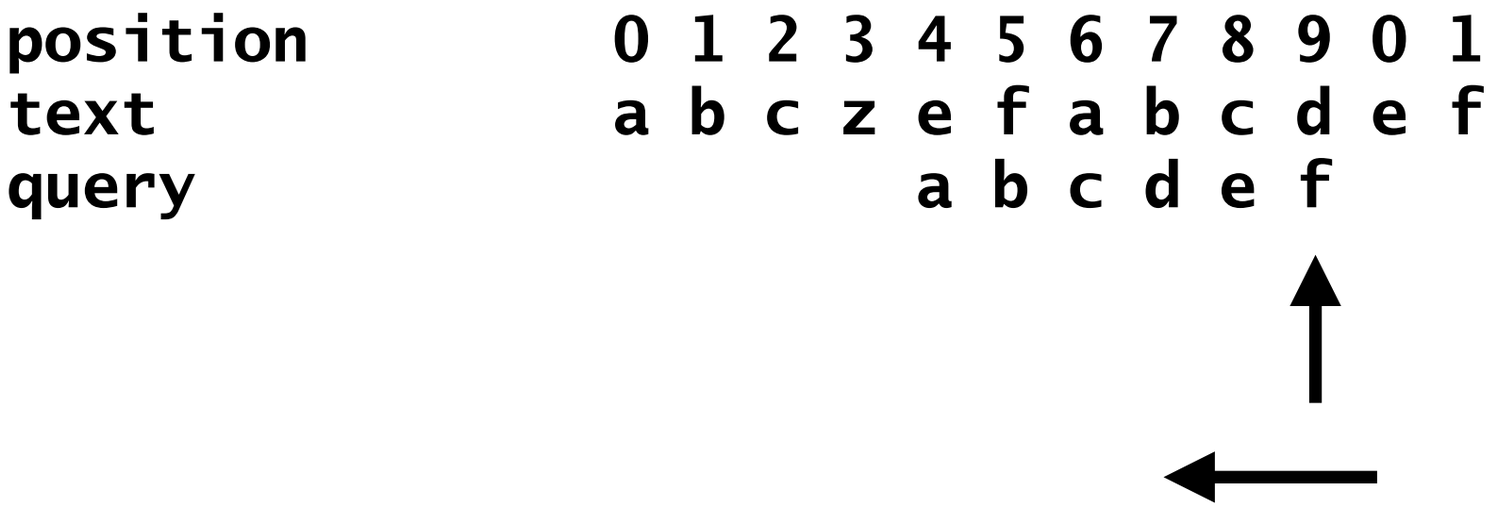
- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - Work *backwards* through the query when trying to match in the text
  - We find a mismatch at position 3
    - Our query doesn't contain a "z" though…
    - It wouldn't make sense to shift the query over by 1 and try again since this will lead to a mismatch (as will when we sift and start at position 2 and position 3).
    - Shift all the way past the "bad" letter and start again (e.g., position 4).

```
position        0 1 2 3 4 5 6 7 8 9 0 1
text            a b c z e f a b c d e f
query           a b c d e f
```



Finding Strings: Boyer-Moore Alg.

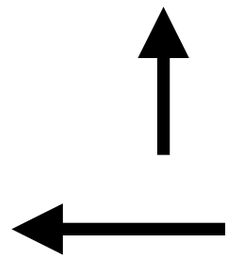
- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - That the “extreme” case.
  - **Question:** what about the case where it’s a mismatch with a letter that the query *does* have?



Finding Strings: Boyer-Moore Alg.

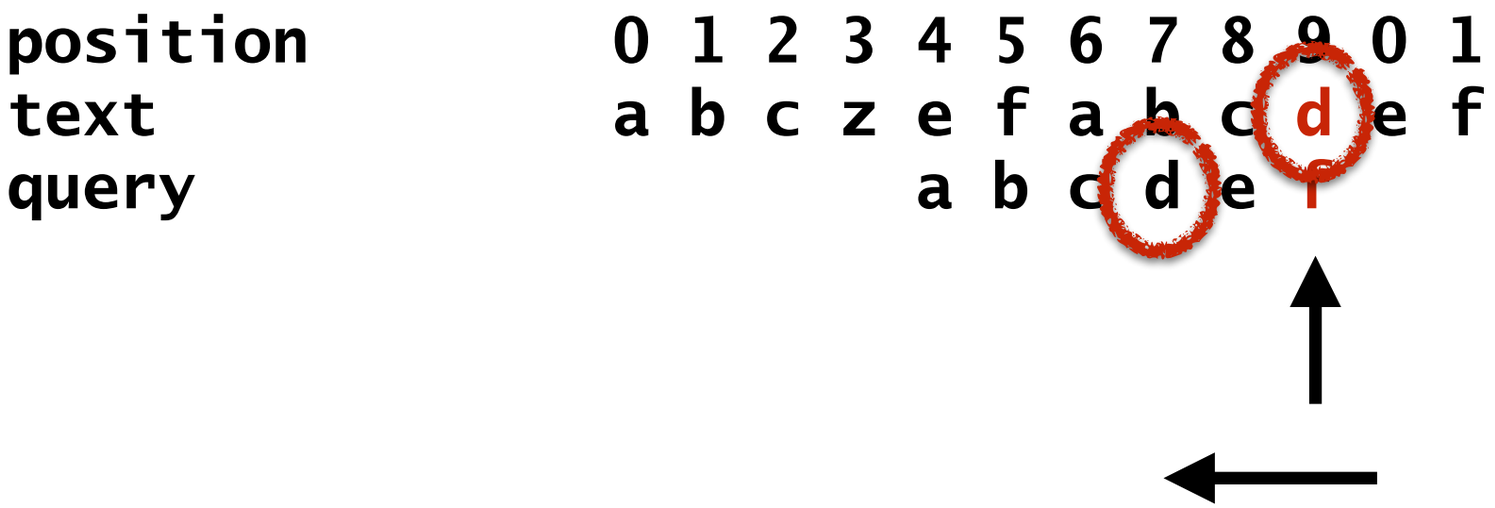
- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - That the “extreme” case.
  - **Question:** what about the case where it’s a mismatch with a letter that the query *does* have?
    - ex. picking up after the shift past “z”…
      - first try to match “f” in query against “d” in text — mismatch!

```
position        0 1 2 3 4 5 6 7 8 9 0 1
text            a b c z e f a b c d e f
query                   a b c d e f
```



Finding Strings: Boyer-Moore Alg.

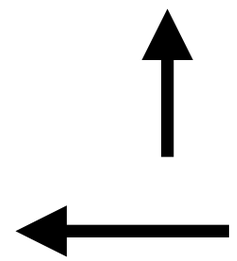
- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - That the “extreme” case.
  - **Question:** what about the case where it’s a mismatch with a letter that the query *does* have?
    - ex. picking up after the shift past “z”…
      - first try to match “f” in query against “d” in text — mismatch!
      - since we are working backwards, the next possible match is where “d” in query is aligned with “d” in text…



Finding Strings: Boyer-Moore Alg.

- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - That the “extreme” case.
  - **Question:** what about the case where it’s a mismatch with a letter that the query *does* have?
    - ex. picking up after the shift past “z”…
      - first try to match “f” in query against “d” in text — mismatch!
      - since we are working backwards, the next possible match is where “d” in query is aligned with “d” in text…
  - doing this shift may or may not work (it does in this case)
  - test as usual (starting from the end).

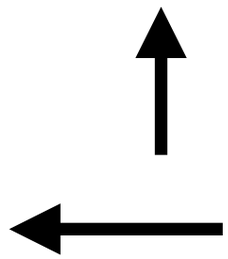
```
position    0 1 2 3 4 5 6 7 8 9 0 1
text        a b c z e f a b c d e f
query               a b c d e f
```



Finding Strings: Boyer-Moore Alg.

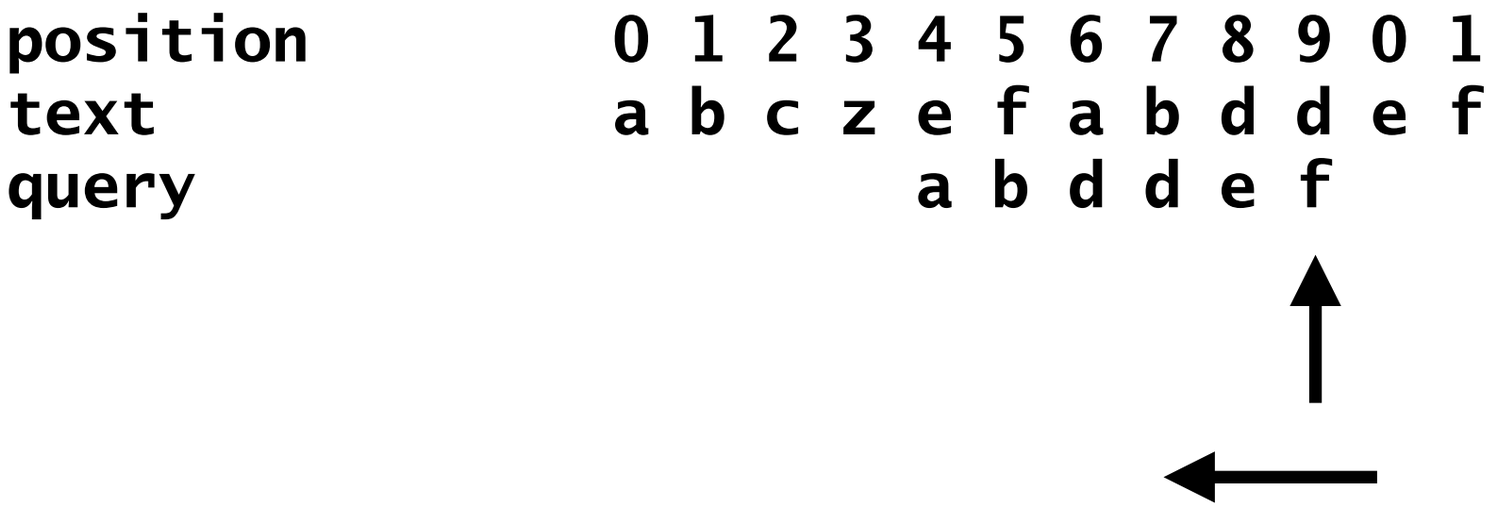
- **Key Insight:** make use of the work we did in partially matching the query to the text at a given position, before discovering that not all of the query matched there.
- Now:
  - That the “extreme” case.
  - **Question:** what about the case where it’s a mismatch with a letter that the query *does* have?
    - ex. picking up after the shift past “z”…
      - first try to match “f” in query against “d” in text — mismatch!
      - since we are working backwards, the next possible match is where “d” in query is aligned with “d” in text…
  - doing this shift may or may not work (it does in this case)
  - test as usual (starting from the end).

```
position    0 1 2 3 4 5 6 7 8 9 0 1
text        a b c z e f a b c d e f
query               a b c d e f
```



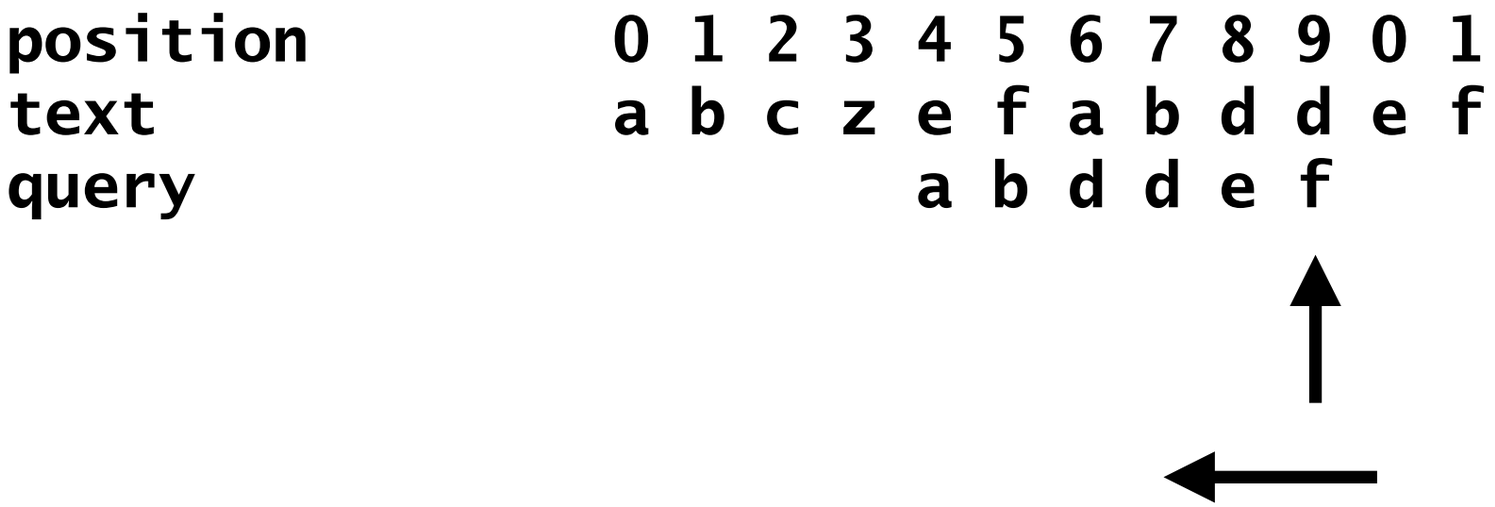
Finding Strings: Boyer-Moore Alg.

- **Question:** what if there were multiple “d”s in the query?



Finding Strings: Boyer-Moore Alg.

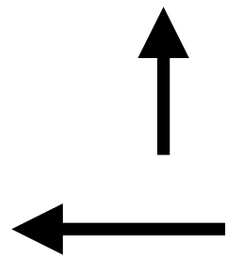
- **Question:** what if there were multiple “d”s in the query?
- *A slight modification…*
  - *change text to have 2 “d”s*
  - *change query to have 2 “d”s*
  - *put query back to position before the “d” shift.*



Finding Strings: Boyer-Moore Alg.

- **Question:** what if there were multiple “d”s in the query?
- *A slight modification…*
  - *change text to have 2 “d”s*
  - *change query to have 2 “d”s*
  - *put query back to position before the “d” shift.*
- *Question*: should we shift s.t. the 1st or 2nd “d” in query lines up with the originally mismatched “d” (at position 9) in text?

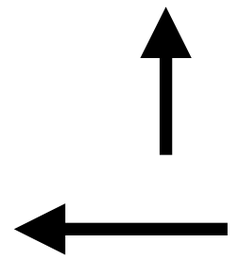
```
position        0 1 2 3 4 5 6 7 8 9 0 1
text            a b c z e f a b d d e f
query                   a b d d e f
```



Finding Strings: Boyer-Moore Alg.

- **Question:** what if there were multiple “d”s in the query?
- *A slight modification…*
  - *change text to have 2 “d”s*
  - *change query to have 2 “d”s*
  - *put query back to position before the “d” shift.*
- *Question*: should we shift s.t. the 1st or 2nd “d” in query lines up with the originally mismatched “d” (at position 9) in text?
  - if we shift s.t. the 1st “d” lined up, we’d scoot right by the correct match!

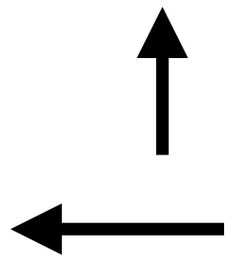
```
position        0 1 2 3 4 5 6 7 8 9 0 1
text            a b c z e f a b d d e f
query                       a b d d e f
```



Finding Strings: Boyer-Moore Alg.

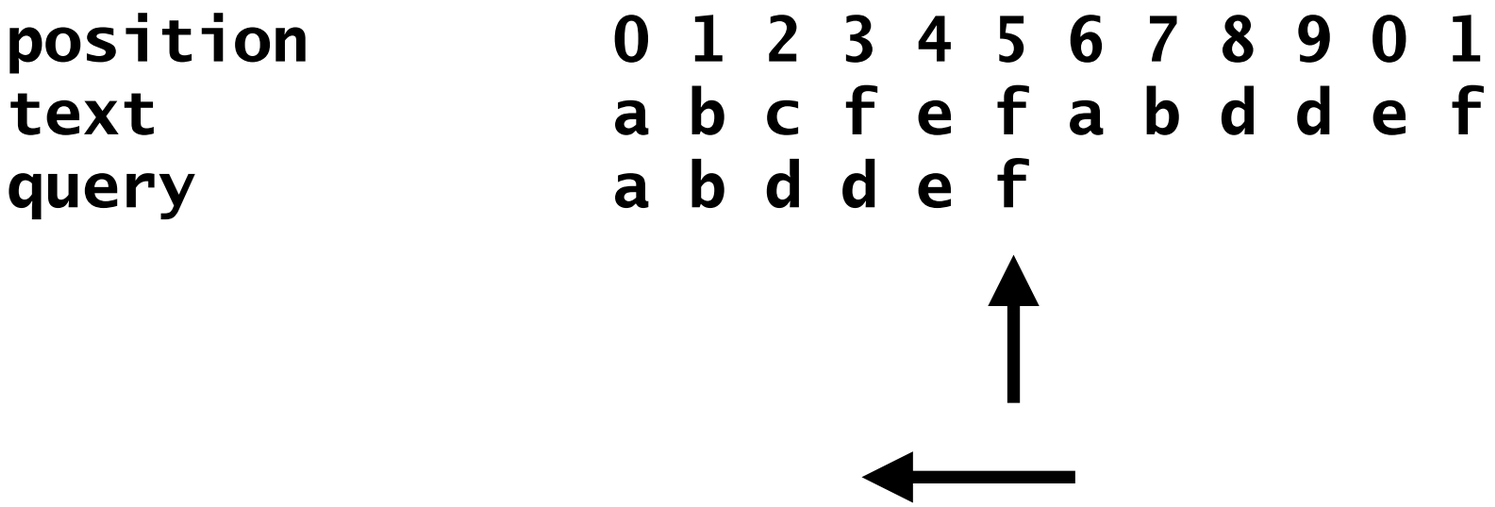
- **Question:** what if there were multiple “d”s in the query?
- *A slight modification…*
  - *change text to have 2 “d”s*
  - *change query to have 2 “d”s*
  - *put query back to position before the “d” shift.*
- *Question*: should we shift s.t. the 1st or 2nd “d” in query lines up with the originally mismatched “d” (at position 9) in text?
  - if we shift s.t. the 1st “d” lined up, we’d scoot right by the correct match!
  - So, in general, we shift based on the *last* occurrence in the query of the mismatched character in the text that caused the mismatch.

```
position      0 1 2 3 4 5 6 7 8 9 0 1
text          a b c z e f a b d d e f
query                   a b d d e f
```



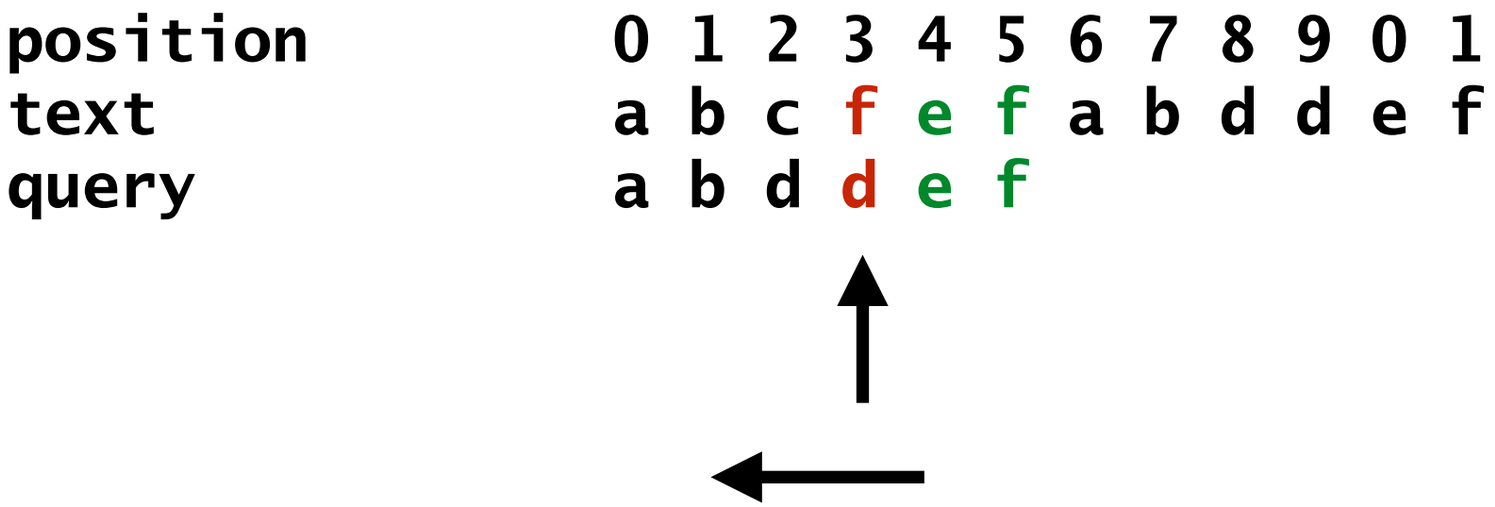
Finding Strings: Boyer-Moore Alg.

- **Question:** what if the last occurrence in the query of the mismatched character is already after the mismatch?
- *A slight modification to text*
  - *change "z" at pos. 3 to "f".*



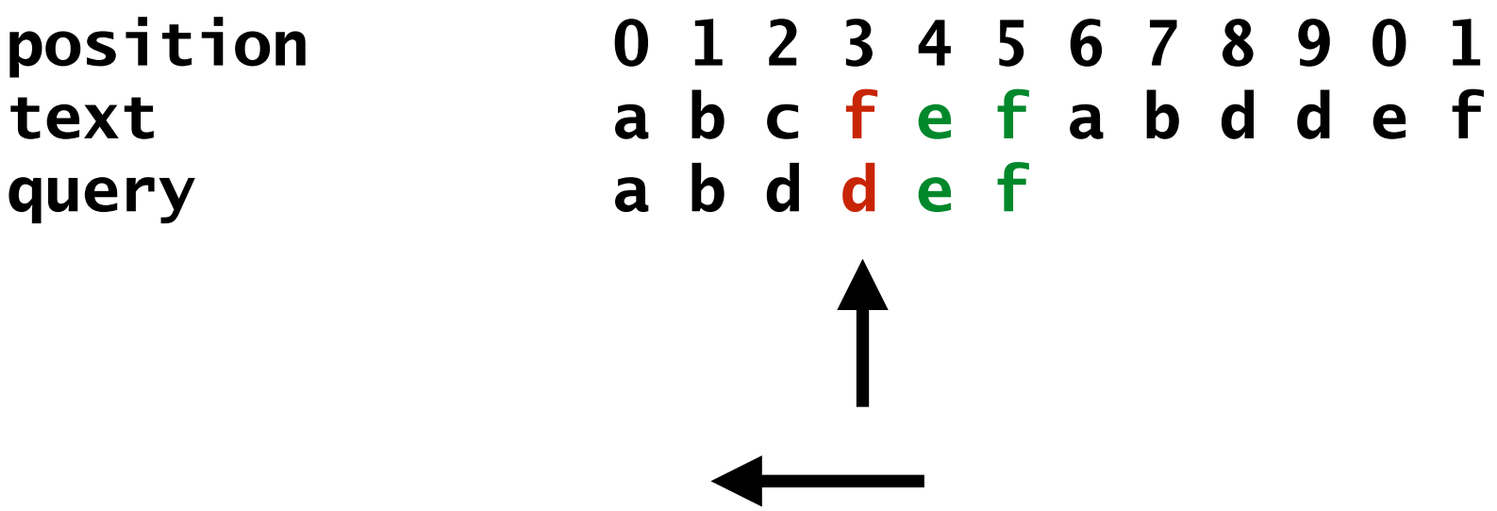
Finding Strings: Boyer-Moore Alg.

- **Question:** what if the last occurrence in the query of the mismatched character is already after the mismatch?
- *A slight modification to text*
  - *change "z" at pos. 3 to "f".*
- We don't want to move backwards…



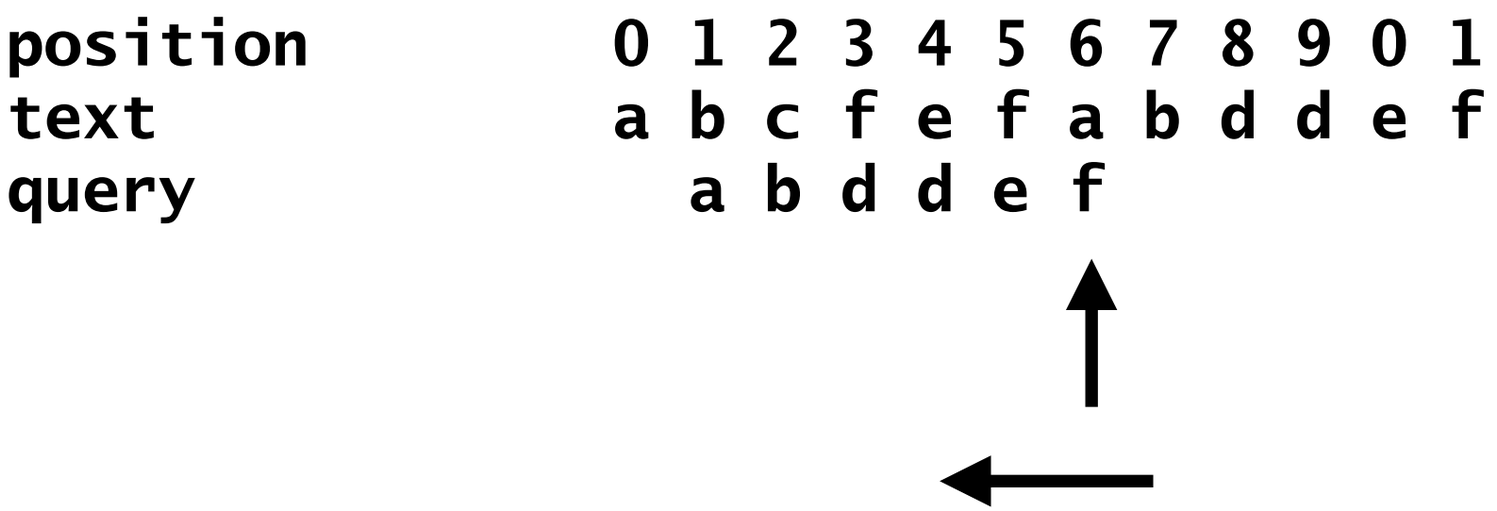
Finding Strings: Boyer-Moore Alg.

- **Question:** what if the last occurrence in the query of the mismatched character is already after the mismatch?
- *A slight modification to text*
  - *change "z" at pos. 3 to "f".*
- We don't want to move backwards…
- So we can't make use of the partial match information — just advance by 1 as in the naive algorithm.



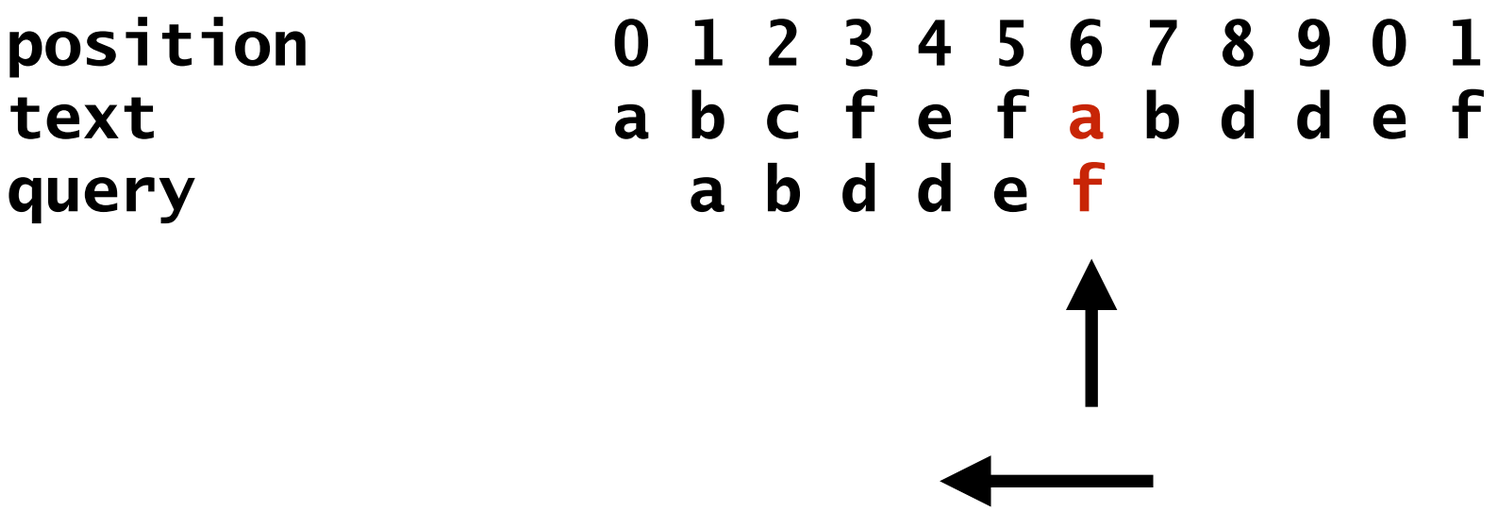
Finding Strings: Boyer-Moore Alg.

- **Question:** what if the last occurrence in the query of the mismatched character is already after the mismatch?
- *A slight modification to text*
  - *change "z" at pos. 3 to "f".*
- We don't want to move backwards…
- So we can't make use of the partial match information — just advance by 1 as in the naive algorithm.



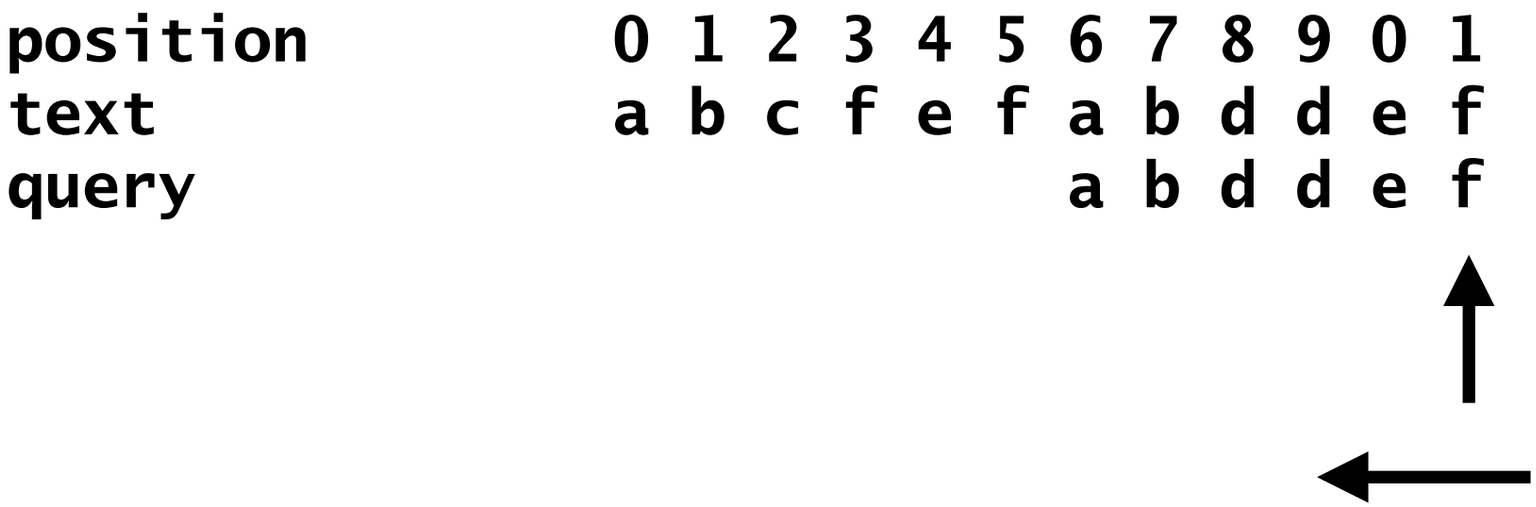
Finding Strings: Boyer-Moore Alg.

- **Question:** what if the last occurrence in the query of the mismatched character is already after the mismatch?
- *A slight modification to text*
  - *change "z" at pos. 3 to "f".*
- We don't want to move backwards…
- So we can't make use of the partial match information — just advance by 1 as in the naive algorithm.



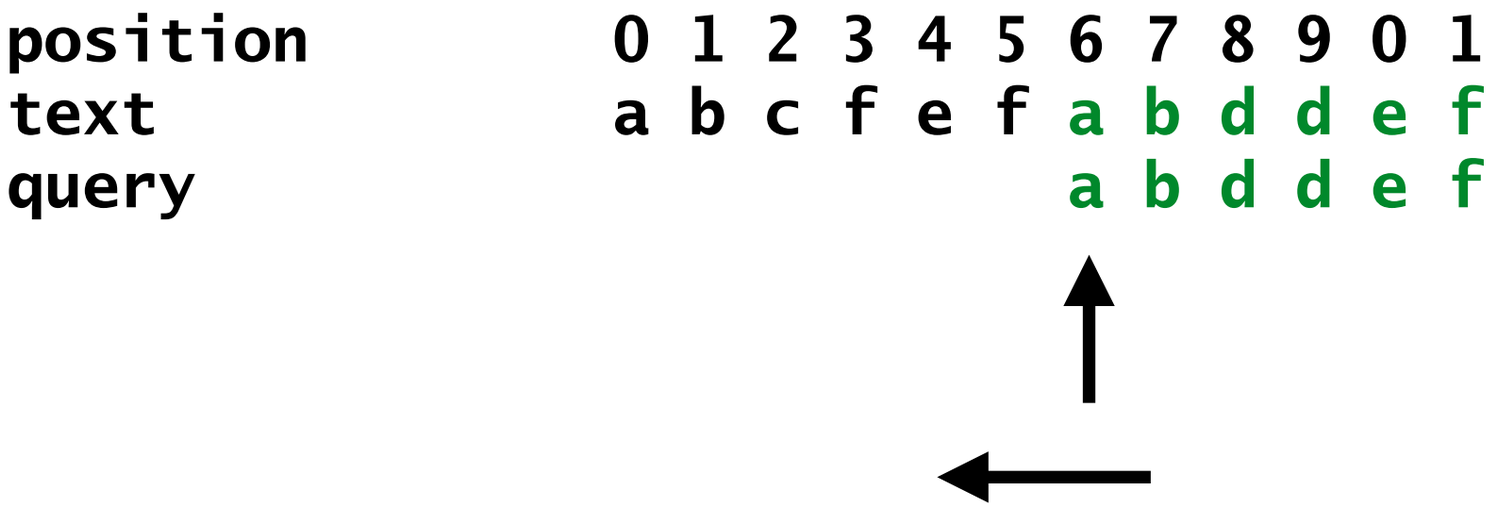
Finding Strings: Boyer-Moore Alg.

- **Question:** what if the last occurrence in the query of the mismatched character is already after the mismatch?
- *A slight modification to text*
  - *change "z" at pos. 3 to "f".*
- We don't want to move backwards…
- So we can't make use of the partial match information — just advance by 1 as in the naive algorithm.



Finding Strings: Boyer-Moore Alg.

- **Question:** what if the last occurrence in the query of the mismatched character is already after the mismatch?
- *A slight modification to text*
  - *change "z" at pos. 3 to "f".*
- We don't want to move backwards…
- So we can't make use of the partial match information — just advance by 1 as in the naive algorithm.



Finding Strings: Boyer-Moore Alg.

- We do have one big question left to address…
- **Question:** how do we efficiently compute the *last* occurrence of the character in the query string?
- We can do this as a preprocessing step!
  - store a HashMap of Character => Integer.

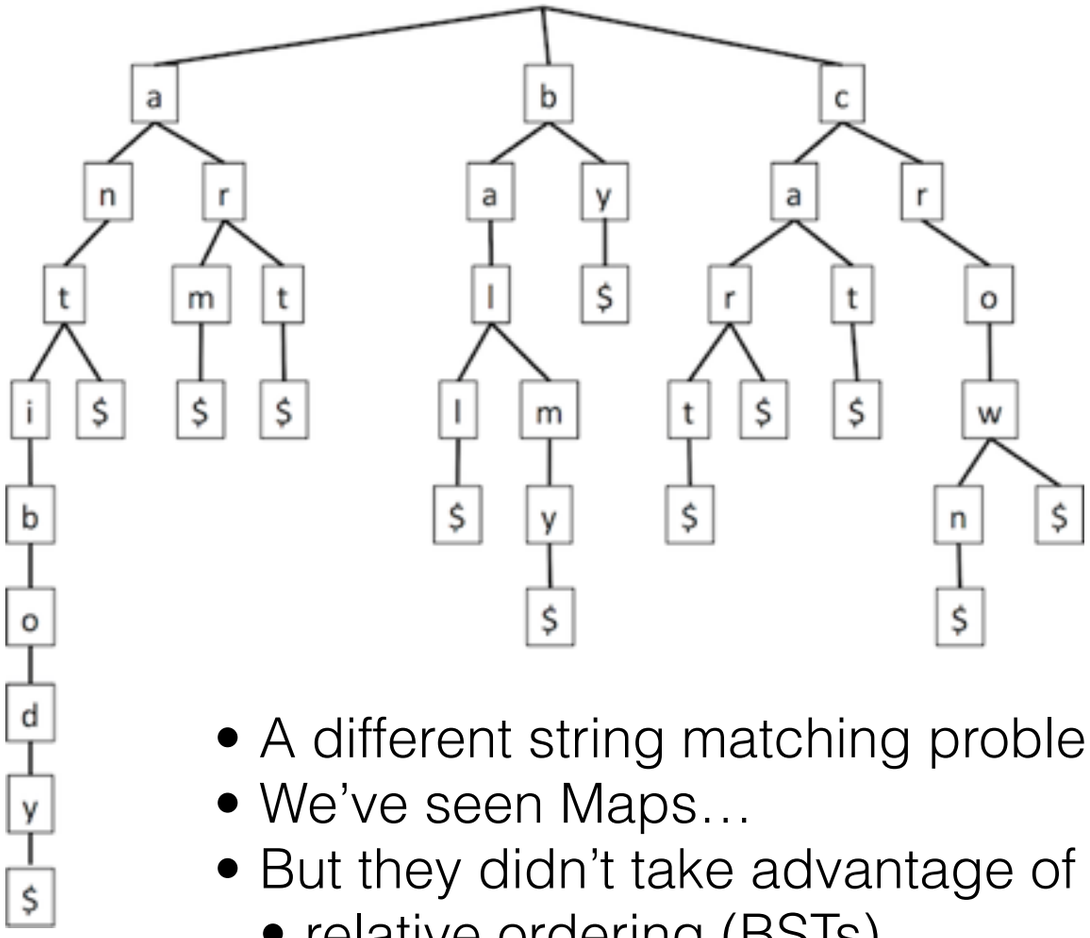
```
// Initialization.
Map<Character, Integer> last = new HashMap<>();
for (int i = 0; i < tLen; i++) {
    last.put(text[i], -1);   // set all chars, by default, to -1
}
for (int i = 0; i < pLen; i++) {
    last.put(pattern[i], i); // update last seen positions
}
```

```
position        0 1 2 3 4 5 6 7 8 9 0 1
text            a b c f e f a b d d e f
query             a b d d e f
```

Finding Strings: Boyer-Moore Alg.

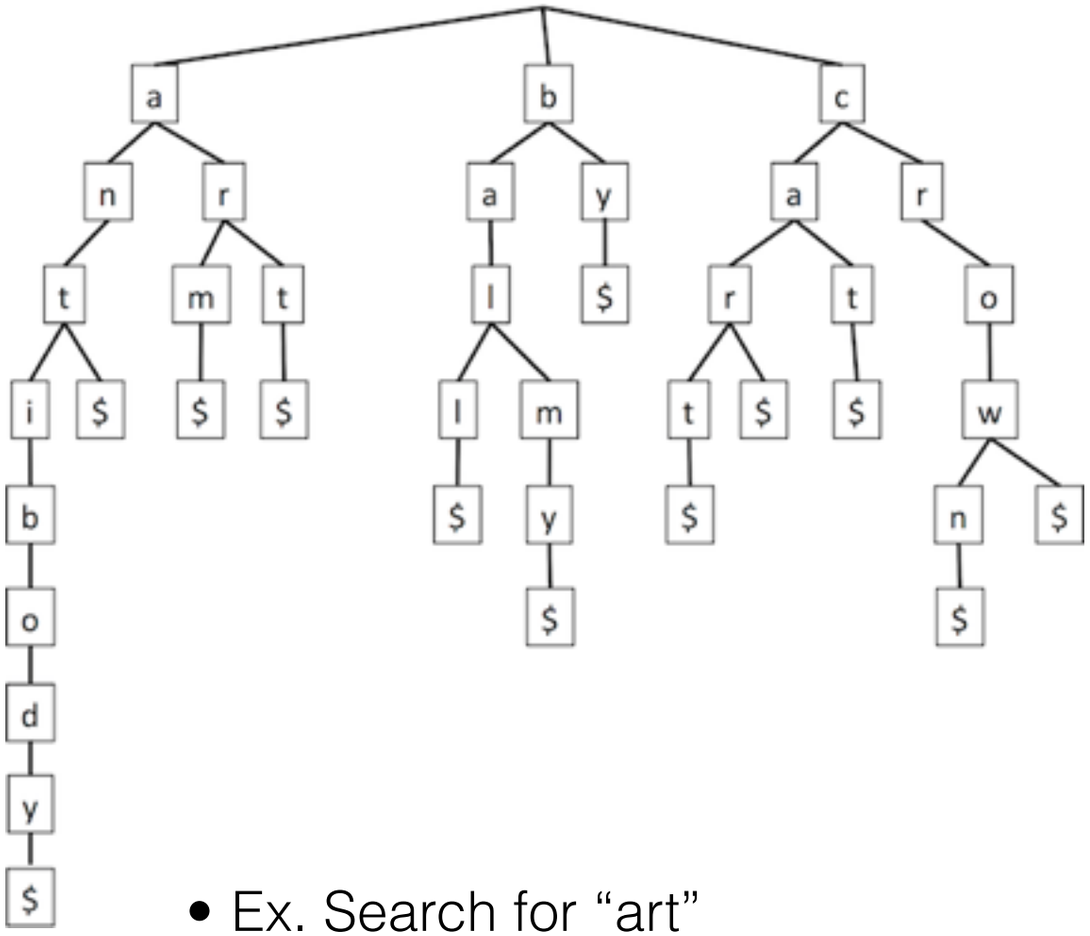
```java
public static int findBoyerMoore(char[] text, char[] pattern) {
    int tLen = text.length;
    int pLen = pattern.length;

    // trivial search for empty string.
    if (pLen == 0)
        return 0;

    // Initialization.
    Map<Character, Integer> last = new HashMap<>();
    for (int i = 0; i < tLen; i++) {
        last.put(text[i], -1);   // set all chars, by default, to -1
    }
    for (int i = 0; i < pLen; i++) {
        last.put(pattern[i], i); // update last seen positions
    }

    // Start with the end of the pattern aligned at index pLen-1 in the text.
    int tIdx = pLen - 1;  // index into the text
    int pIdx = pLen - 1;  // index into the pattern
    while (tIdx < tLen) { // match! return tIdx if complete match; otherwise, keep checking.
        if (text[tIdx] == pattern[pIdx]) {
            if (pIdx == 0)
                return tIdx; // done!
            tIdx--;
            pIdx--;
        } else { // jump step + restart at end of pattern
            tIdx += pLen - Math.min(pIdx, 1 + last.get(text[tIdx]));
            pIdx = pLen - 1;
        }
    }
    return -1; // not found.
}
```

Tries



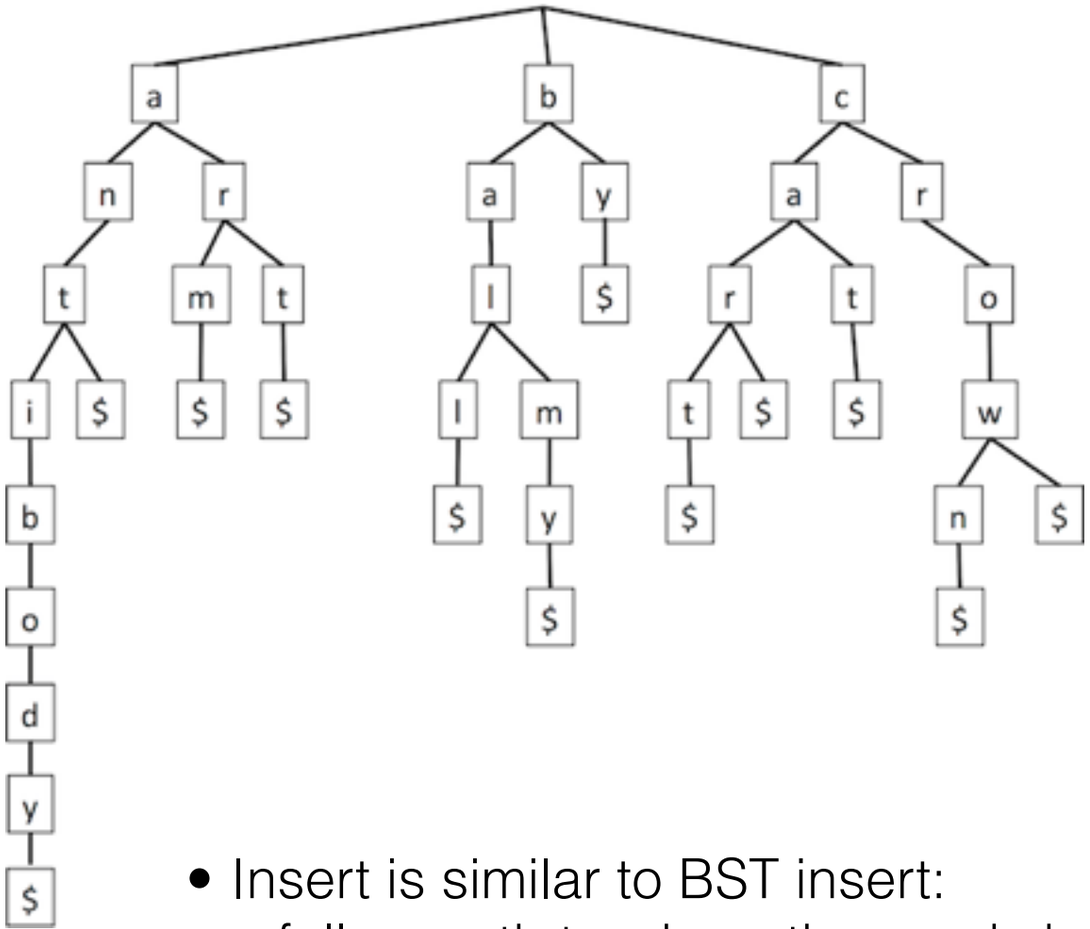
Tries

- A different string matching problem: looking up words in a dictionary.
- We've seen Maps…
- But they didn't take advantage of the *content* of the words, just
  - relative ordering (BSTs),
  - some funky numeric function of their characters (hashing)
- A ***Trie*** provides an alternative data structure that allows efficient lookup based on the strings in the map.
  - *Note: **Trie** comes from re**trie**val and is pronounced "try" not "tree".*
- A trie is a multi-way tree
- Each node (other than the root) has a character
- Often assumed that each word ends in $ to clearly distinguish where a word ends.
  - This is useful to know when a child could further extend a particular node that is a complete word itself (i.e., a prefix).
- To match, start at the root —> go to child with the first letter, then go to child with second letter, etc.



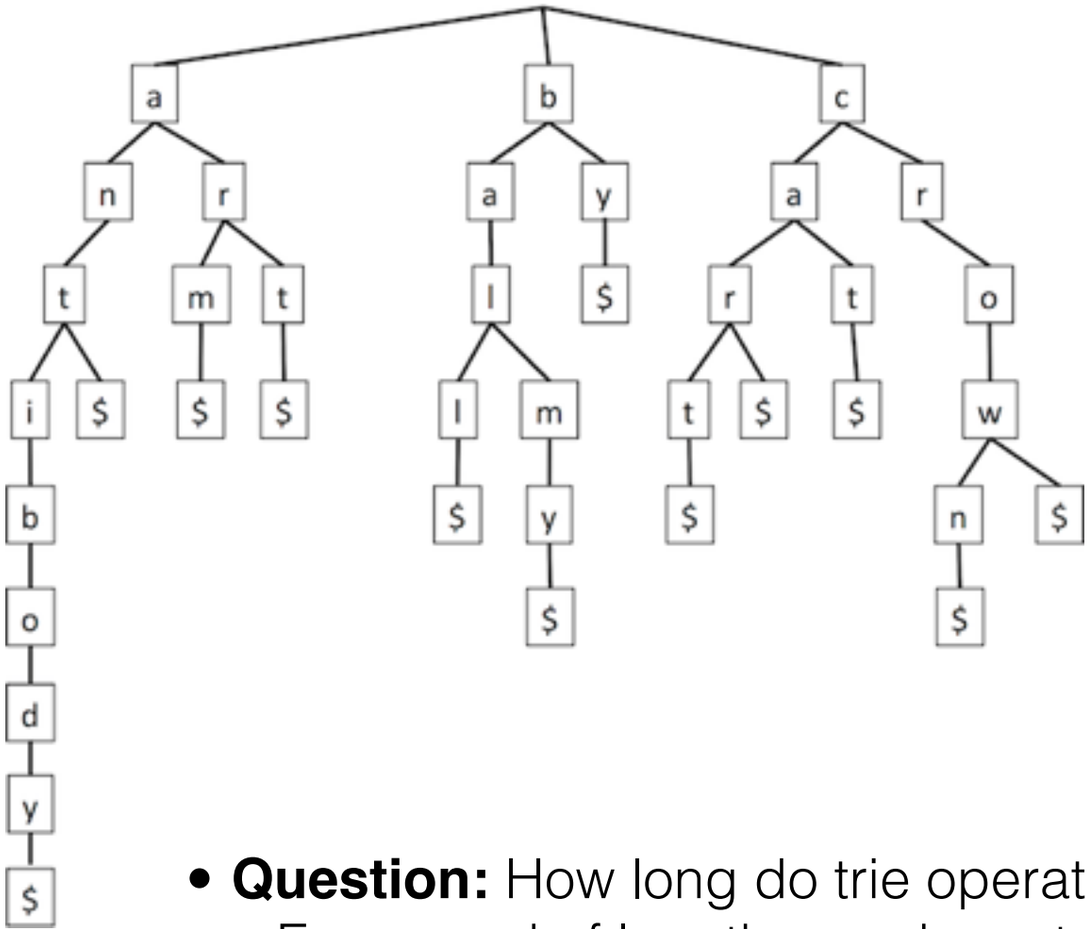
Tries

- Ex. Search for “art”
  - start at root
  - go to “a”
  - then go to “r”
  - then go to “t”
  - then end-of-word “$”



Tries

- Insert is similar to BST insert:
  - follow path to where the word should be
  - add nodes/edges to complete the word.
- Ex. insert “artistic”
  - pick up at “t” of “art”
  - add another child to “t” — “i”.
  - keep inserting…



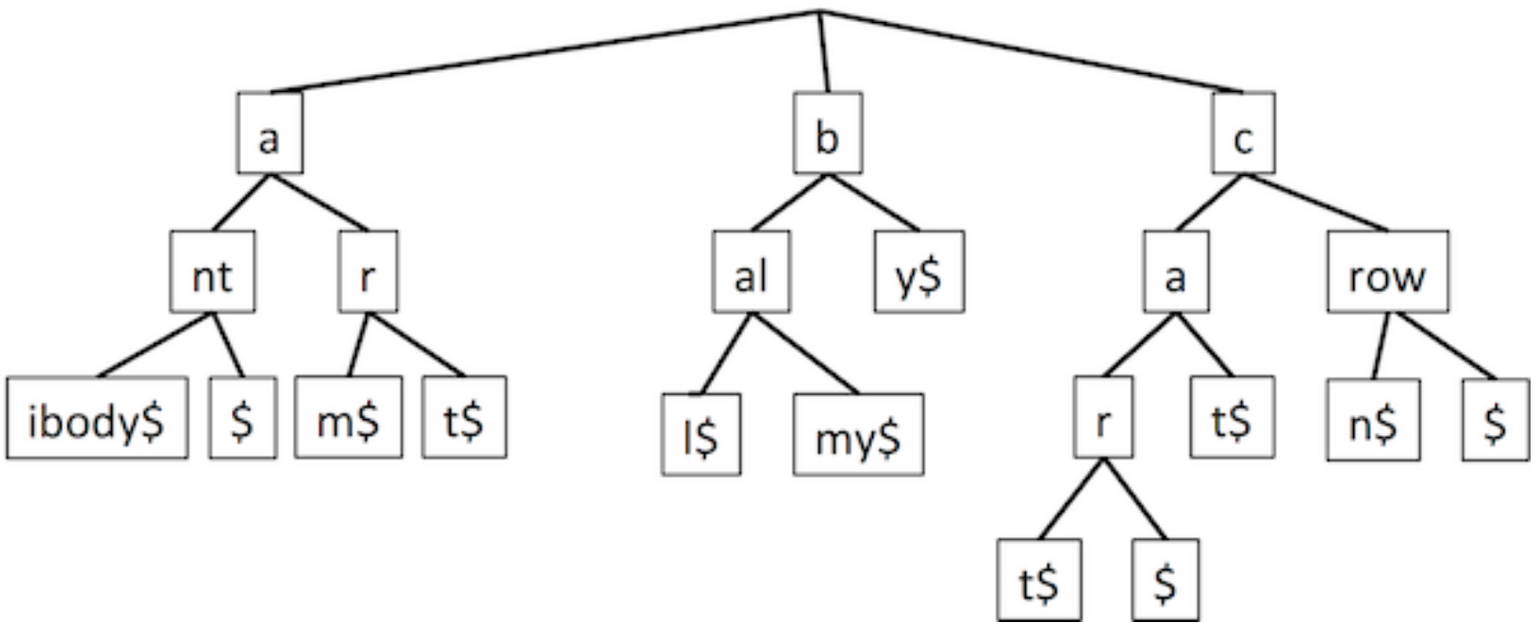
Tries

- **Question:** How long do trie operations take?!
  - For a word of length *n* we have to take *n* steps down the trie
  - At each step, we have to decide which child to “visit”
  - If our alphabet has *d* letters, then there is at most *d* children
  - total —> *O(dn)*
  - There are ways to speed this up (think about it — check out the book, too)
  - Big take-away: runtime is proportional to the length of the query word

Compressed Tries

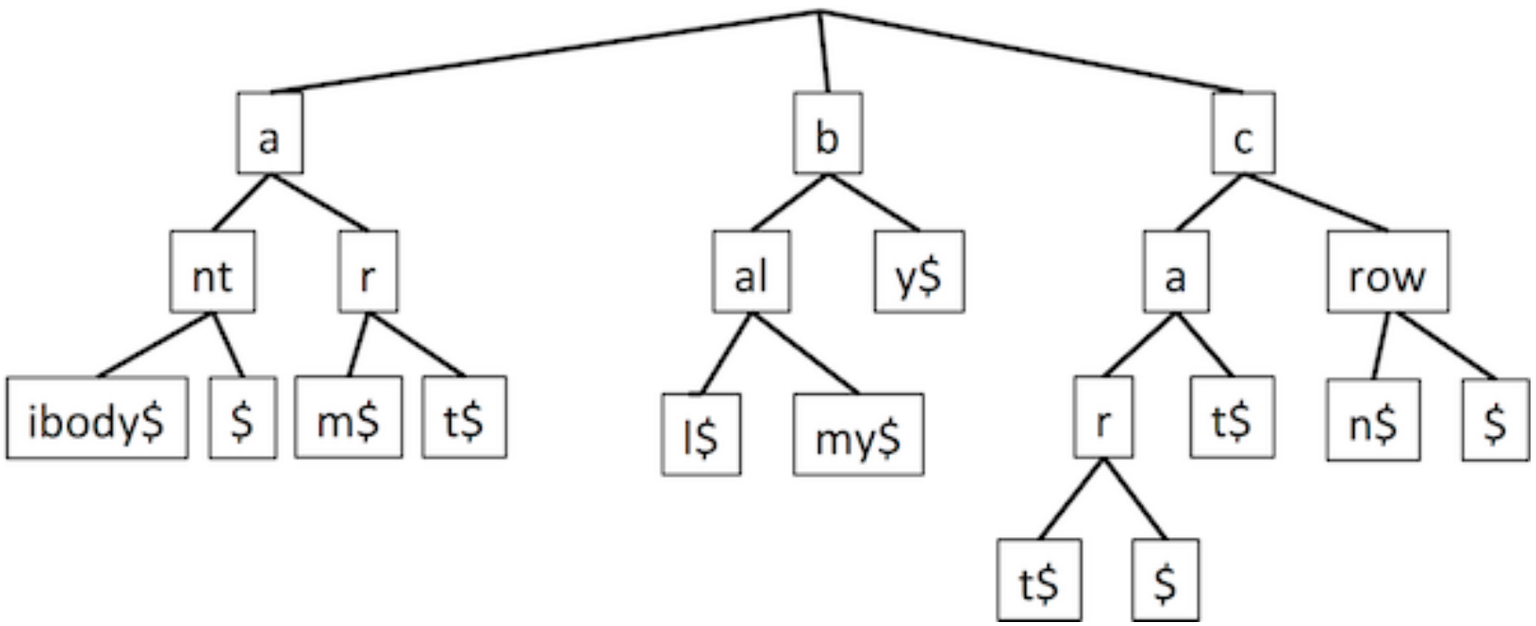
Compressed Tries

- A trie can be compressed…
  - no need to store a node w/ only one child
  - in fact, you can group a whole substring that has no branches inside it!
- # nodes reduced from O(n) (the # of letters) to O(s) (the # of strings).
- Also, could store indices to a string in set of strings rather than storing the string itself in each node.



(Compressed) Tries: Final Notes

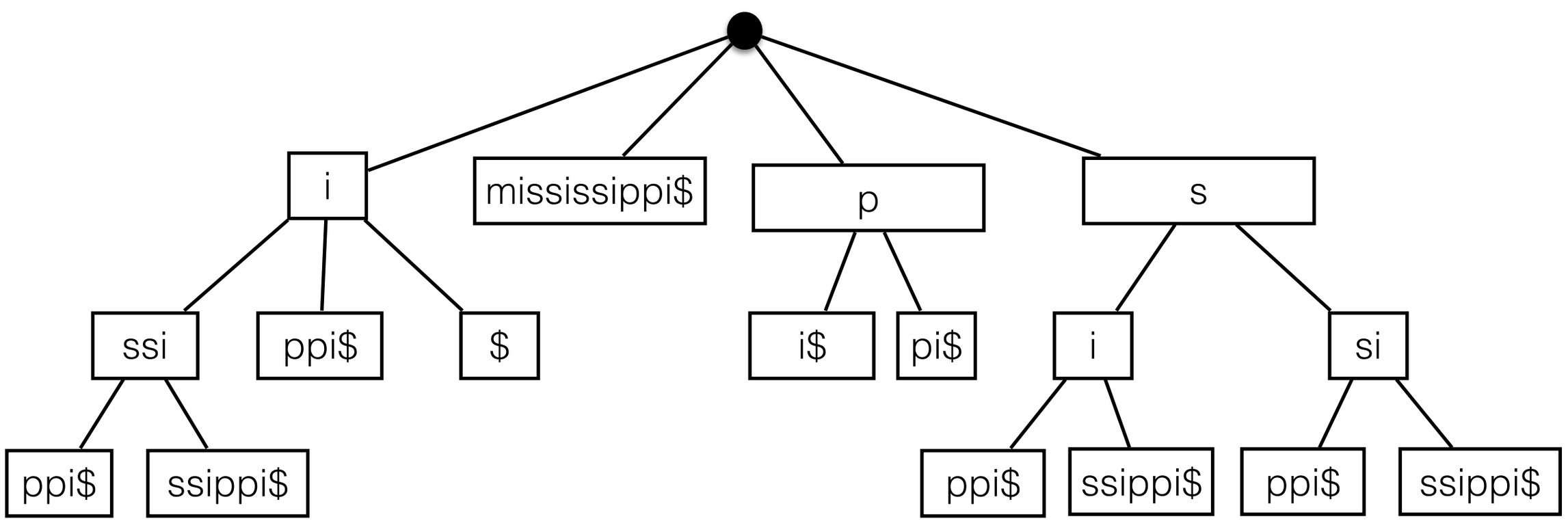
- A trie can also…
- be used for sorting!
  - Insert all words into a trie — pre-order traversal of tree
- be used for auto-complete!
  - after you've typed a few characters, you've gone part-way down the tree, and the leaf nodes under that node are possible completions.
  - each node could store some sort of "score" (e.g., usage frequency).
  - you could update this over time based on a particular user's habits.



Suffix Trees

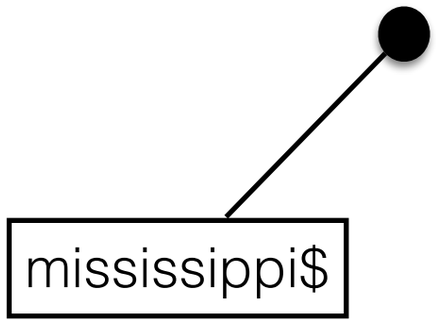
Suffix Trees

- Let's return to the problem of finding substrings.
- We could: preprocess the *text* into a trie containing all of its *suffixes.*
- We would call such a thing a ***Suffix Trie*** or a ***Suffix Tree***.
- The canonical example — "Mississippi" — is shown below (as a *compressed* suffix trie).
  - ***[Talk about construction]***
  - ***[searching — just like a regular trie!]***
  - see: online notes and textbook for applications of suffix trees (there are many!)
- You get to explore (non-compressed) Suffix Trees in SA11 (due Monday).

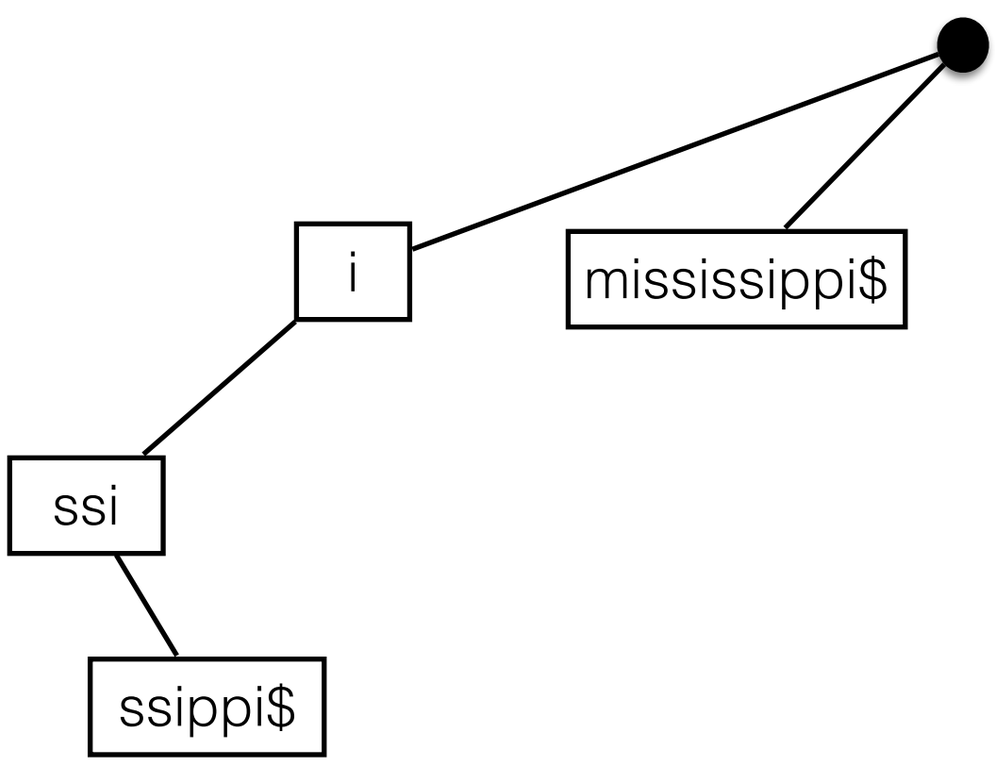


Suffix Trees

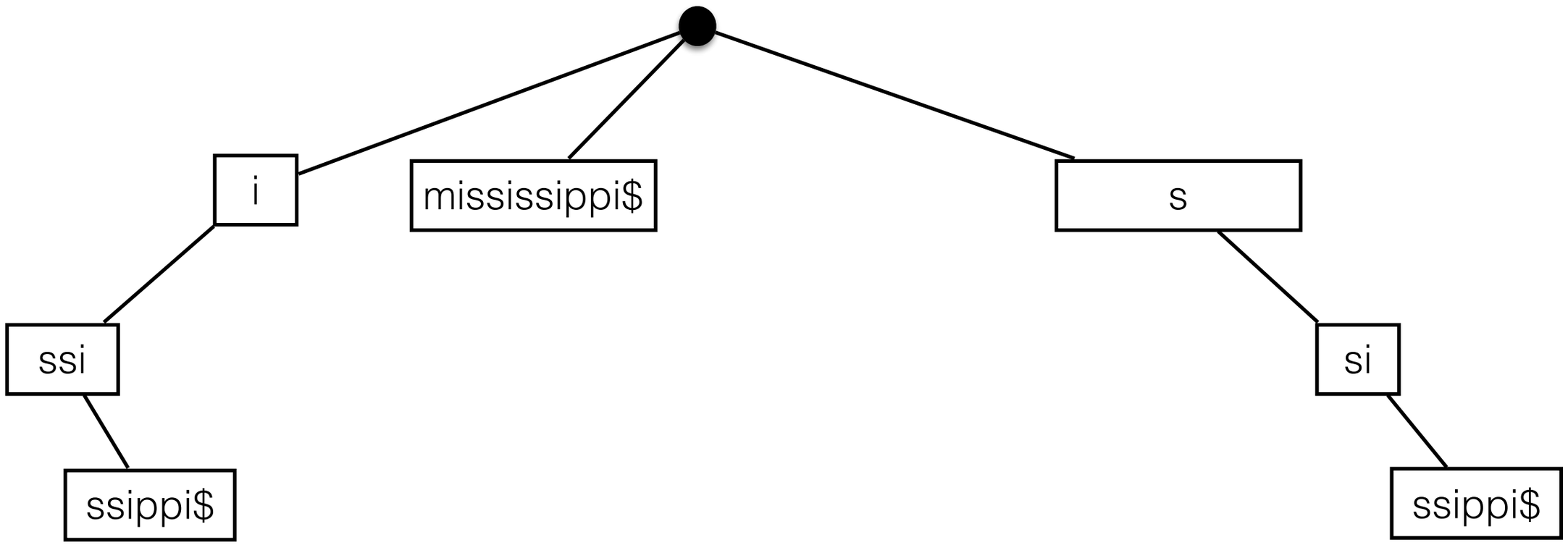
Suffix Trees



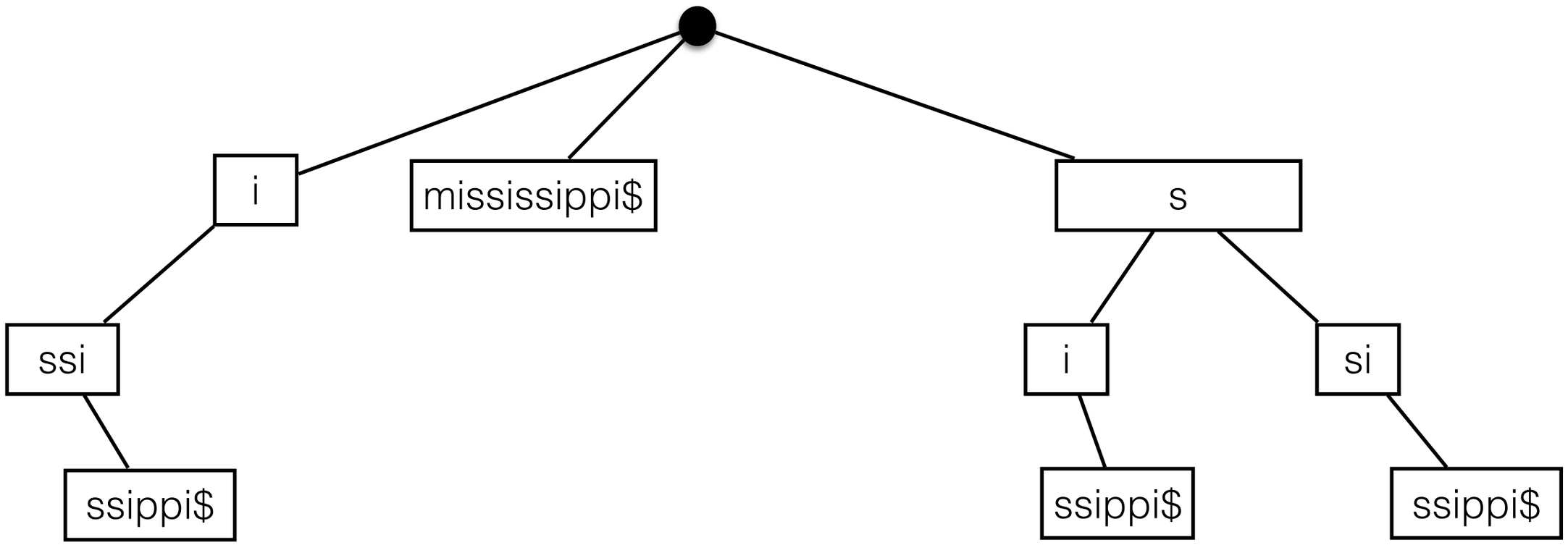
Suffix Trees



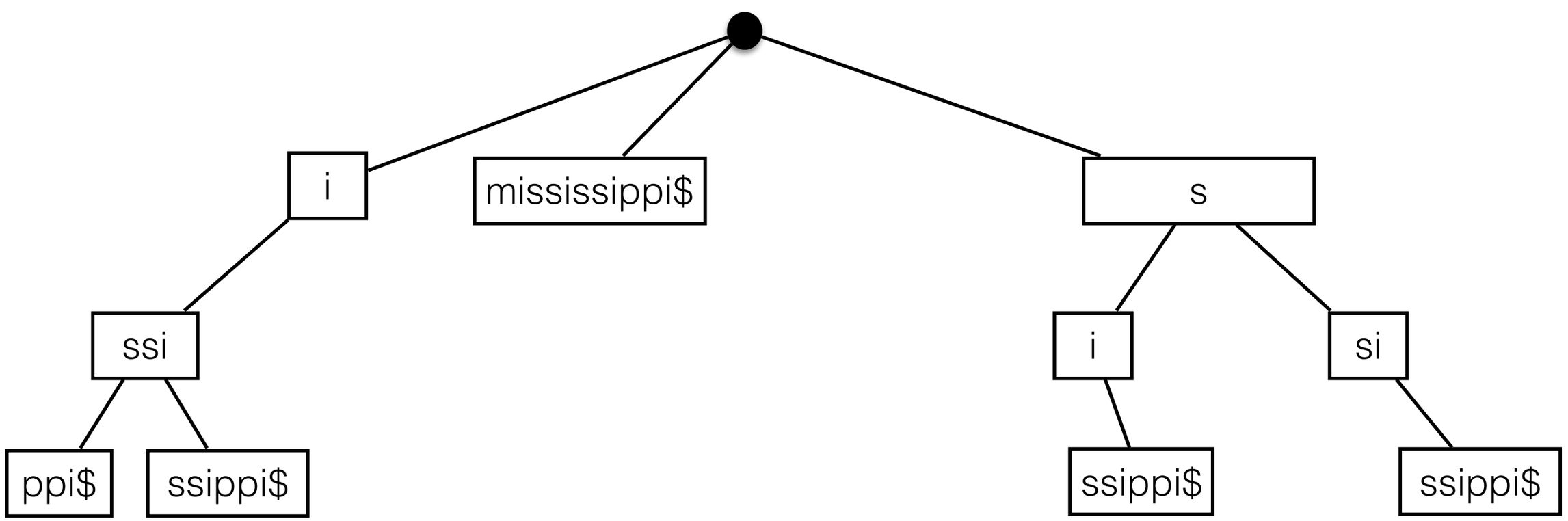
Suffix Trees



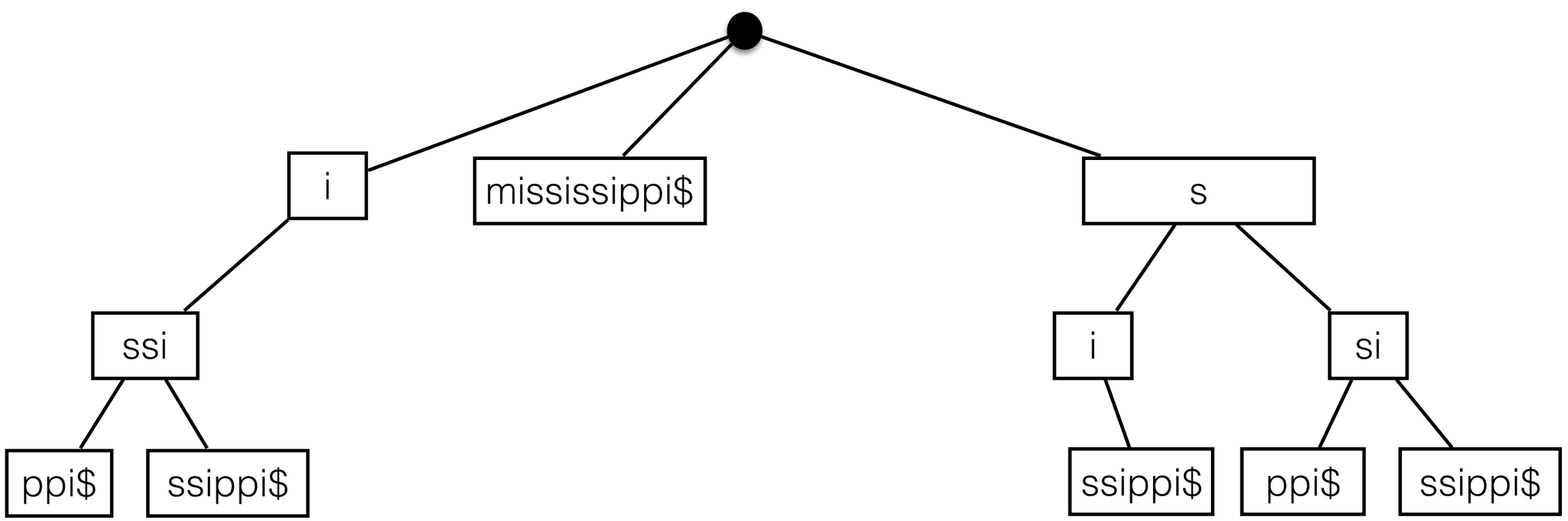
Suffix Trees



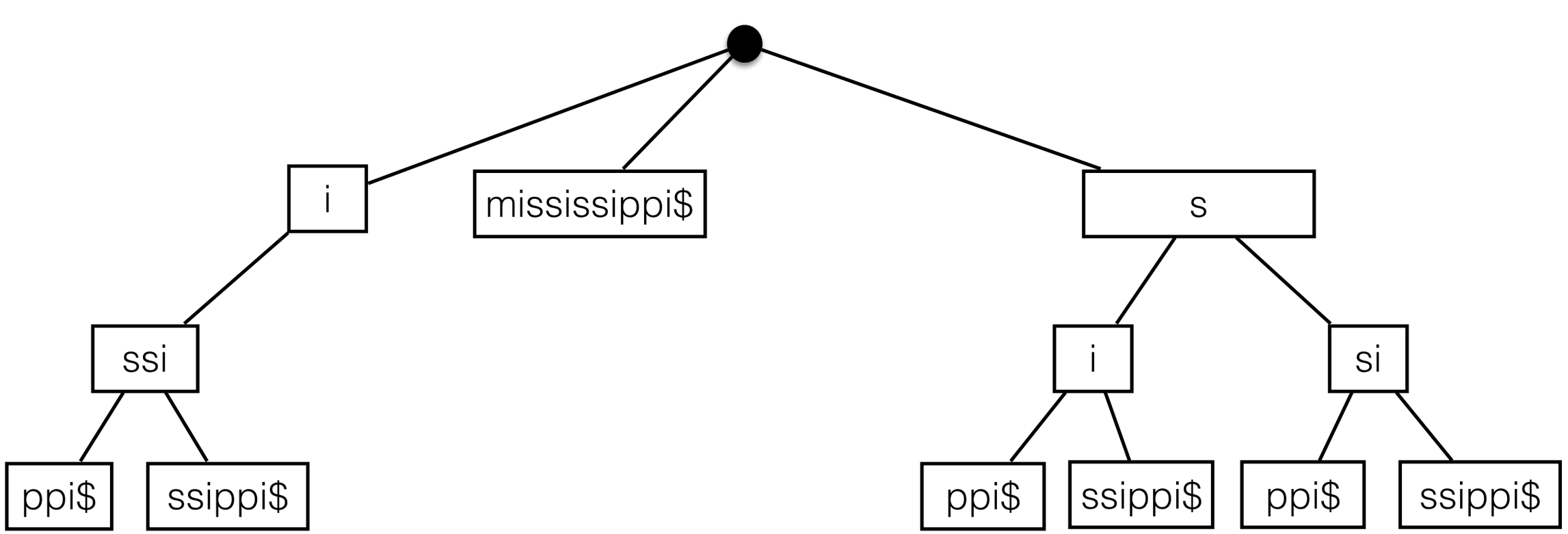
Suffix Trees



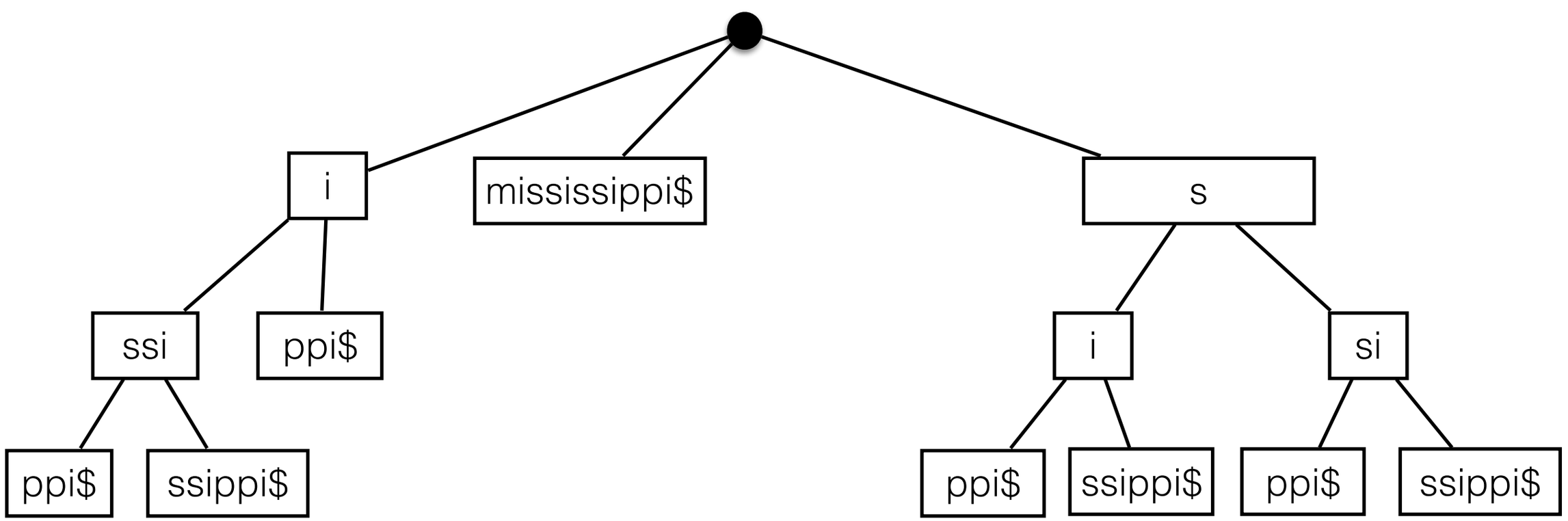
Suffix Trees



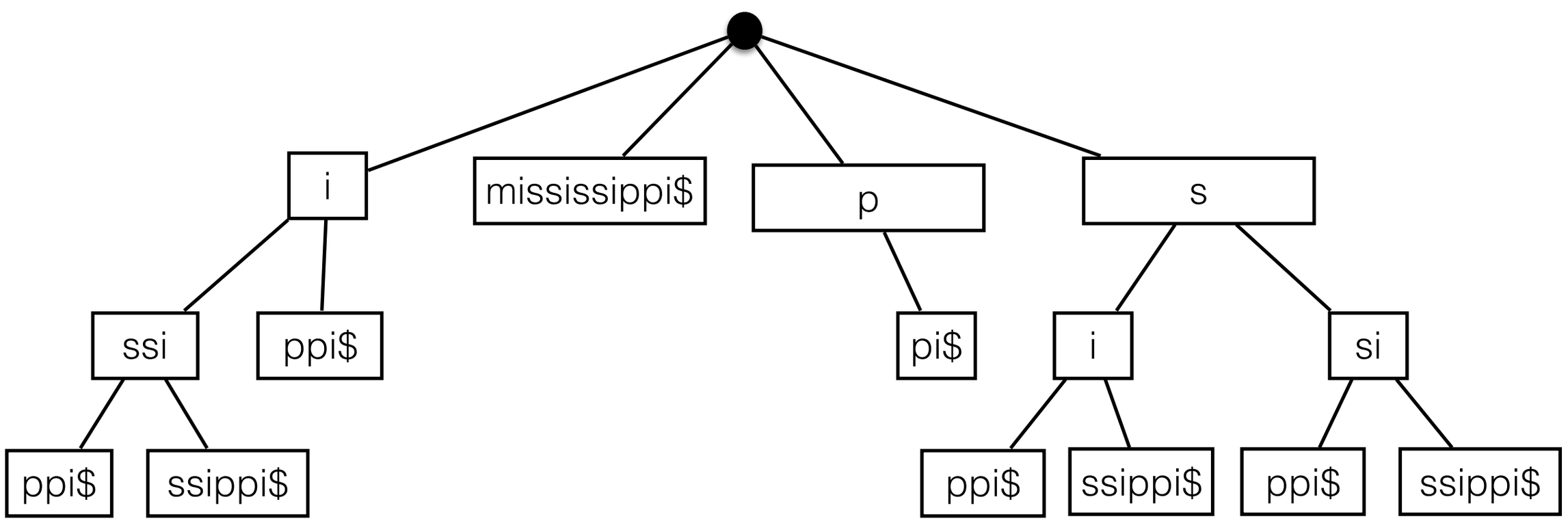
Suffix Trees



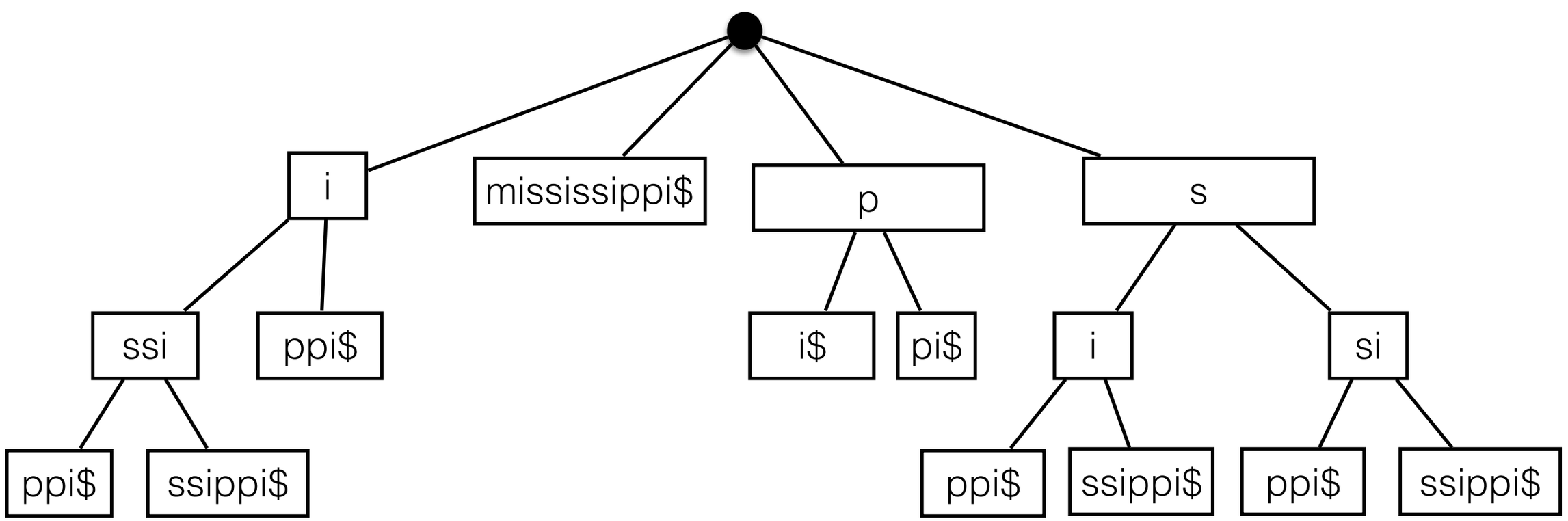
Suffix Trees



Suffix Trees



Suffix Trees



Suffix Trees