Lesson 6: Pattern Matching Algorithms

CSC325 – ADVANCED DATA STRUCTURES & ALGORITHMS | SPRING 2022

DR. ANDREY TIMOFEYEV

OUTLINE

- Introduction.
- Brute force algorithm.
- Boyer-Moore algorithm.
- Knuth-Morris-Pratt algorithm.
- Trie data structure.

INTRODUCTION

- **Notations:**
  - Text is modeled as **character strings**.
    - `S = "CGTAAACTGCTTTAATCAAACGC"`, `T = "https://www.latech.edu/"`
  - Characters of a string come from an **alphabet ∑**.
    - $\sum = \{A, C, G, T\}, \sum = \{a, b, c, \ldots, x, y, z, A, B, C, \ldots, x, y, z\}$.
  - Breaking **large strings** into **smaller strings**. String *S* with length *n*.
    - **Indexing** = `S[j]` – character at index j, $0 \leq j \leq n\text{-}1$.
    - **Slicing** = `S[j:k]`, $0 \leq j \leq k \leq n$. **Substring** = characters `S[j]` up to `S[k-1]` but not `S[k]`.
    - **String prefix** = `S[0:k]` or `S[:k]` for $0 \leq k \leq n$.
    - **String suffix** = `S[j:n]` or `S[j:]` for $0 \leq j \leq n$.
- **Pattern-matching problem statement:**
  - Given **text string T** of length *n* and **pattern string P** of length *m*, find whether **P** is a **substring** of **T**.
    - Find lowest (or all) index of **T** at which **P** begins, thus **T[j:j+m] = P**

PATTERN MATCHING ALGORITHMS: BRUTE FORCE (1)

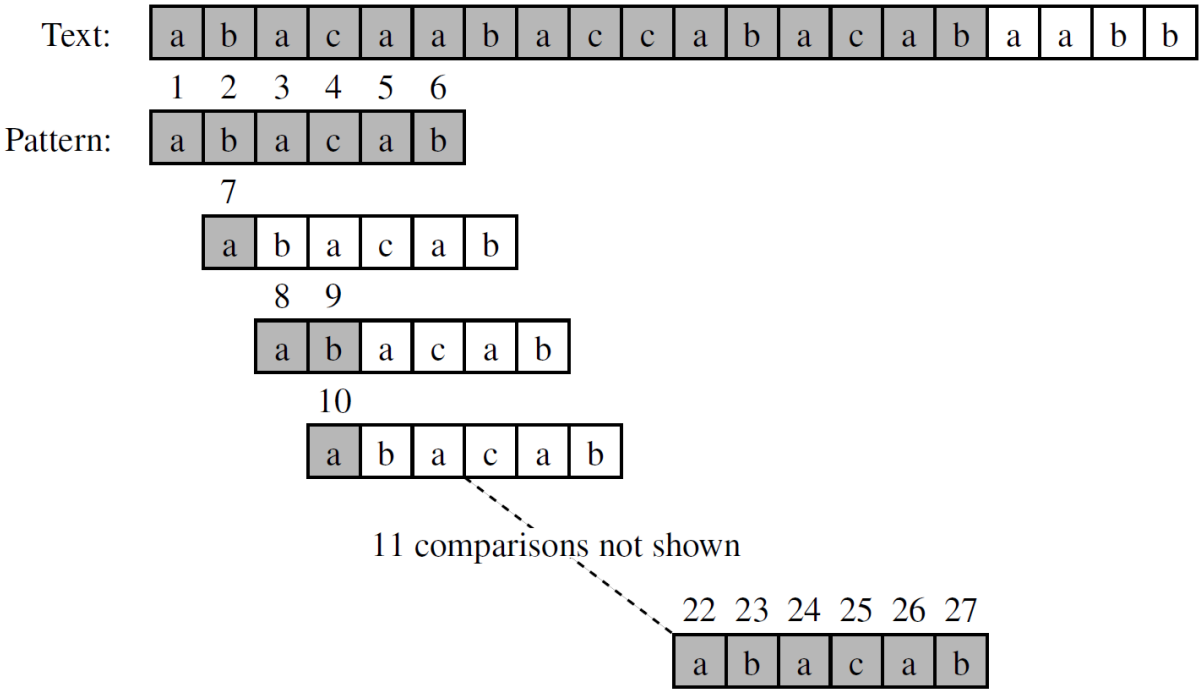
- **Brute-force algorithm design pattern:**
  - Enumerate **all possible configurations** of inputs involved & pick the **best** of **all enumerated configurations**.
- **Brute-force algorithm** for **pattern-matching problem**:
  - Test **all possible placements** of **pattern P** relative to **text T**.
  - Return **lowest index** of **T** at which **P** begins or -1 otherwise.
- **Algorithm steps:**

```
For every potential starting index of P in T
  Try matching every character of P
  If reached the end of P (pattern matched) -> return index
Return -1 (pattern not matched)
```

PATTERN MATCHING ALGORITHMS: BRUTE FORCE (2)

- **Brute force algorithm performance.**
  - **Two nested loops**.
    - **Outer** – indexing through all possible starting indexes of pattern in text.
      - Executes at most $n-m+1$ times.
    - **Inner** – indexing though each character of the pattern, comparing to characters in text.
      - Executes at most m times.
  - **Overall complexity is O(nm).**

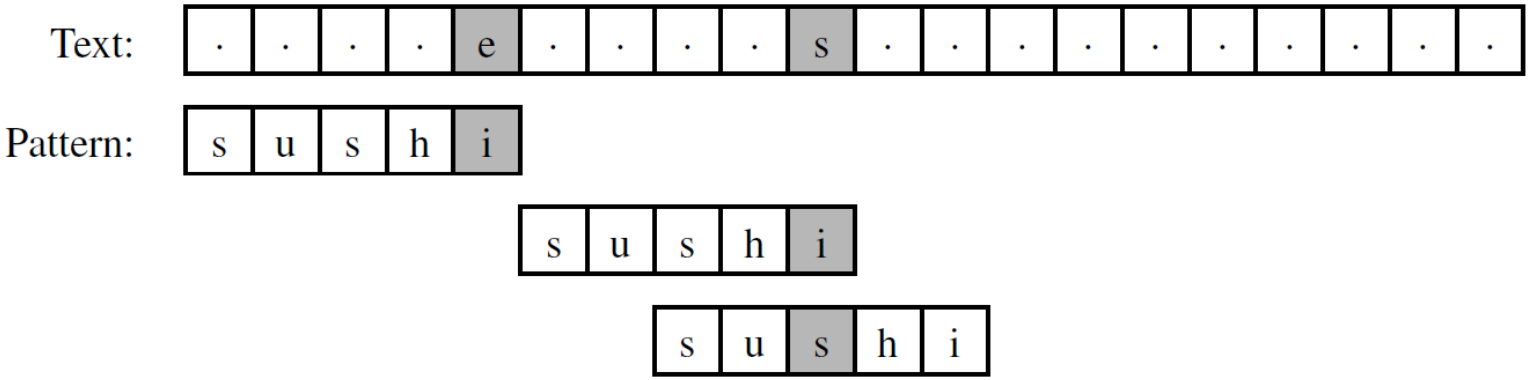
- **Example:**
  - `T = "abacaabaccabacabaabb"`
  - `P = "abacab"`



Brute-force example

PATTERN MATCHING ALGORITHMS: BOYER-MOORE (1)

- **Boyer-Moore algorithm** improves brute-force algorithm by adding **two heuristics**:
  - **Looking-glass heuristics.**
    - When testing if P in T, begin the comparisons from the end of P and move backward to the front of P.
  - **Character-jump heuristics.**
    - Mismatch of T[i] = c with P[k] is handled as follows:
      - If c not in P, then shift P completely past T[i].
      - Otherwise, shift P until an occurrence of c in P gets aligned with T[i].



Boyer-Moore heuristics example

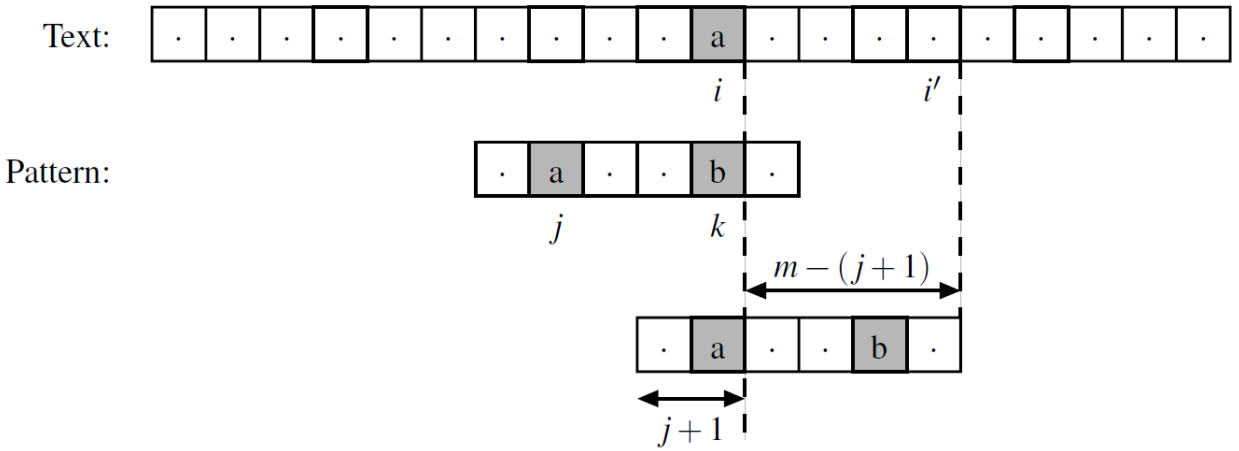
PATTERN MATCHING ALGORITHMS: BOYER-MOORE (2)

- **Boyer-Moore character-jump heuristics.**
  - If last character **matched**, keep checking to **extend** the **match**.
    - Either whole pattern matched, or there is a mismatch in the interior position.
  - If **mismatch** found in the **interior** position:
    - If **mismatched** character **not** in P, then **shift P** completely **past** mismatched character.
    - If **mismatched** character **elsewhere** in P, then **two cases**:
      - **Last occurrence** of **mismatched** character is **BEFORE** pattern character aligned with the **mismatched.**
      - **Last occurrence** of **mismatched** character is **AFTER** pattern character aligned with the **mismatched.**

PATTERN MATCHING ALGORITHMS: BOYER-MOORE (3)

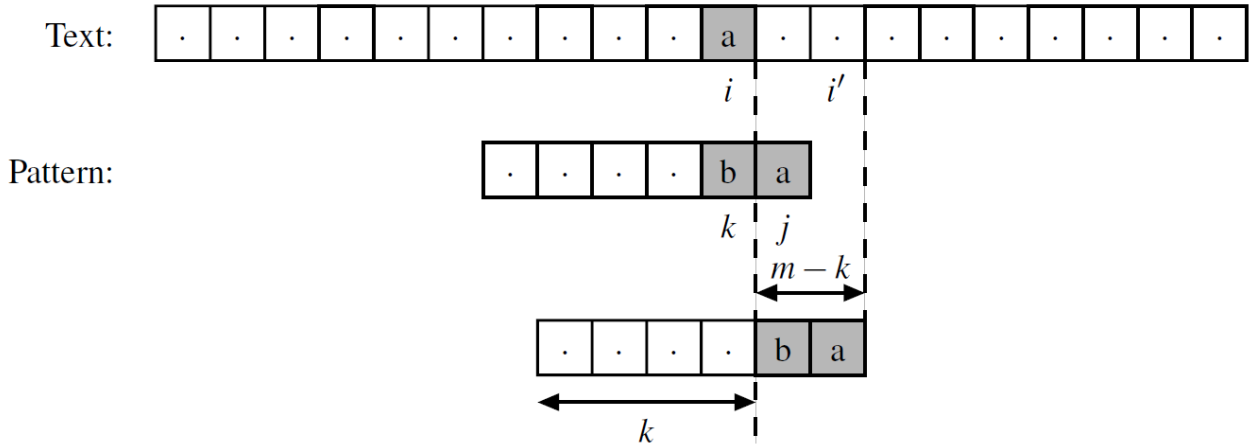
- **Boyer-Moore character-jump heuristics (cont).**
  - i – index of the mismatched character in the text,
  - k – corresponding index in the pattern,
  - j – index of the last occurrence of T[i] within the pattern.

- **Mismatched BEFORE aligned: $j < k$.**
  - Shift pattern by $k-j$; index *i* advances by $m-(j+1)$.



Mismatch before aligned

- **Mismatched AFTER aligned: $j > k$.**
  - Shift pattern by 1; index *i* advances by $m-k$.



Mismatch after aligned

PATTERN MATCHING ALGORITHMS: BOYER-MOORE (4)

- **Boyer-Moore algorithm steps.**
  - Let **pattern P** be of size **m** and **text T** be of size **n**.

```
Create a dictionary with last occurrences of characters in P.
Aling pattern P and text T at index equal to m-1
Go through aligned characters in pattern P and text T from right to left
  If found matching character
    If at the beginning of pattern P, then return index from text T (pattern matched)
    Otherwise, examine previous character of pattern P and text T
  If found mismatch character
    Check if mismatch character is in pattern P and its last occurrence
    Align text T appropriately (if mismatch before or after aligned)
    Align pattern P at index equal to m-1
Return -1 (pattern not matched)
```

PATTERN MATCHING ALGORITHMS: BOYER-MOORE (5)

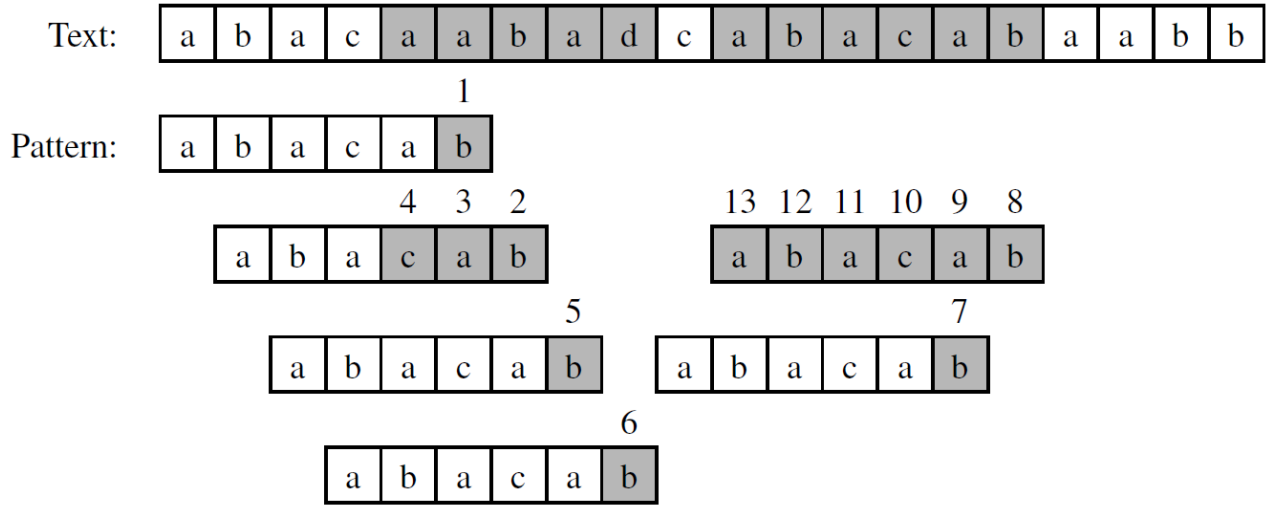
- **Boyer-Moore algorithm performance.**
  - **Finding if/where mismatch occurs in the pattern.**
    - Takes *O(m)*.
  - **Search for the pattern.**
    - Takes *O(nm)*.
  - **Overall complexity is O(nm).**
    - Worst case is unlikely to be achieved for English text.

$$T = \overbrace{aaaaaa\cdots a}^{n}$$

$$P = b\overbrace{aa\cdots a}^{m-1}$$

Worst-case scenario

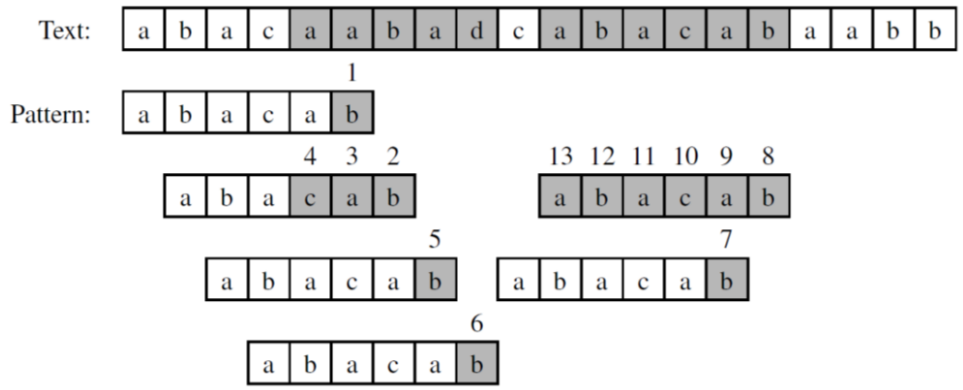
c	a	b	c	d
last(c)	4	5	3	−1



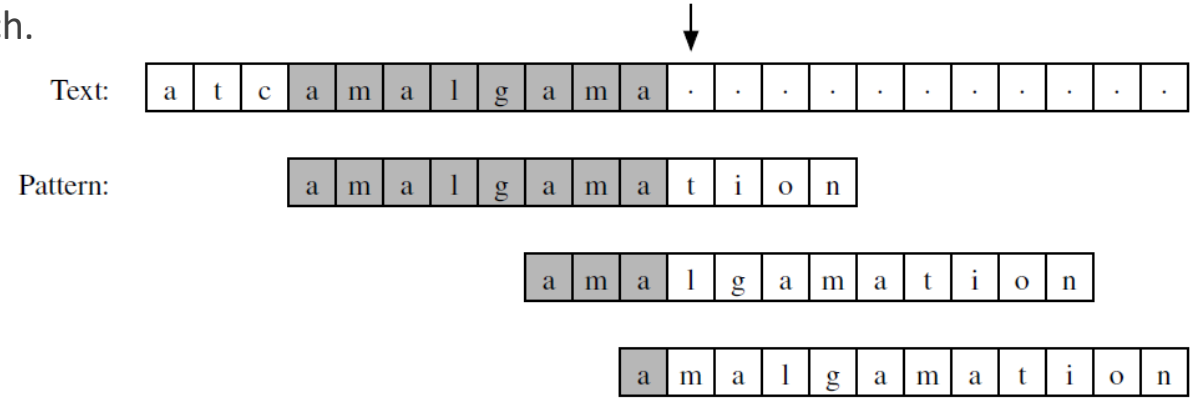
Boyer-Moore example

PATTERN MATCHING ALGORITHMS: KNUTT-MORRIS-PRATT (1)

- **Knutt-Morris-Pratt (KMP) algorithm.**
  - Avoids **information waste**.
    - Compared to brute-force & Boyer-Moore algorithms.
  - Achieves **O(n + m)** complexity.
    - Examines all characters of text & pattern at least once in the worst-case scenario.
- **KMP algorithm improvement:**
  - Pre-computes **self-overlaps** between portions of the pattern.
    - If **mismatch** -> know the **maximum** amount to shift before next search.
- **KMP** is based on pre-computed **failure function.**



Inefficiency of Boyer-Moore algorithm



Example of KMP

PATTERN MATCHING ALGORITHMS: KNUTT-MORRIS-PRATT (2)

- **KMP failure function f(k).**
  - Calculates a **proper shift** of P upon **failed comparison**.
  - **Defined** as a **length** of the **longest prefix** of P that is a **suffix** of P[1:k+1].
  - **If mismatch** at P[k+1] -> function shows how many **immediately preceding** characters can be reused to restart the pattern.
- **Failure function example:**
  - `P = "amalgamation"`

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

PATTERN MATCHING ALGORITHMS: KNUTT-MORRIS-PRATT (3)

- **KMP algorithm steps.**
  - Let **pattern P** be of size **m** and **text T** be of size **n**.

```
Pre-compute failure function list on the pattern
Align text and pattern at the beginning
Go through aligned characters in pattern P and text T, left to right
   If found matching character
     If at the end of pattern P
       Return index from text T (pattern matched)
     Move to the next characters to extend the match
   If found mismatch character
     If not at the beginning of pattern P
       Shift based on the entry of previous character in failure function list
     If at the beginning of pattern P
       Go to the next character in text T
Return -1 (pattern not matched)
```

PATTERN MATCHING ALGORITHMS: KNUTT-MORRIS-PRATT (4)

- **KMP failure function steps.**
  - Let **pattern P** be of size **m**.

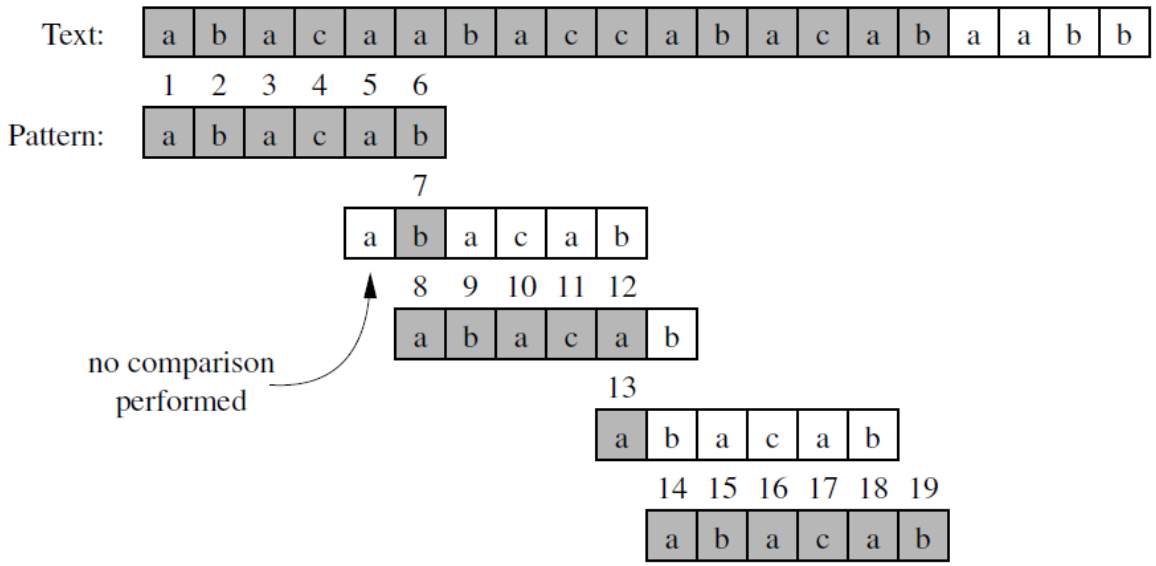
```
Initialize list of failures of the same size as pattern
Initialize indexes of first and next characters in pattern
Go through each character starting at next character
  If next character matches previous character
    Increase failure count and update
    Move on to the next two characters
  If characters do not match
     If not at the beginning
       Update index to the previous character
     If at the beginning
       Move to the next character
Return list of failures
```

PATTERN MATCHING ALGORITHMS: KNUTT-MORRIS-PRATT (5)

- **KMP algorithm performance.**
  - **Computing failure function list.**
    - Pattern P of length m is compared to itself in-place.
    - Takes *O(m)*.
  - **Pattern-matching algorithm.**
    - Iterates of over characters of text T and shift pattern P.
    - Amount by which pattern P is shifted is increased by at least 1 at each iteration.
    - Total number of iteration is at most 2n, thus *O(n)*.
  - **Overall complexity is O(n + m).**

The failure function:

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$f(k)$	0	0	1	0	1	2

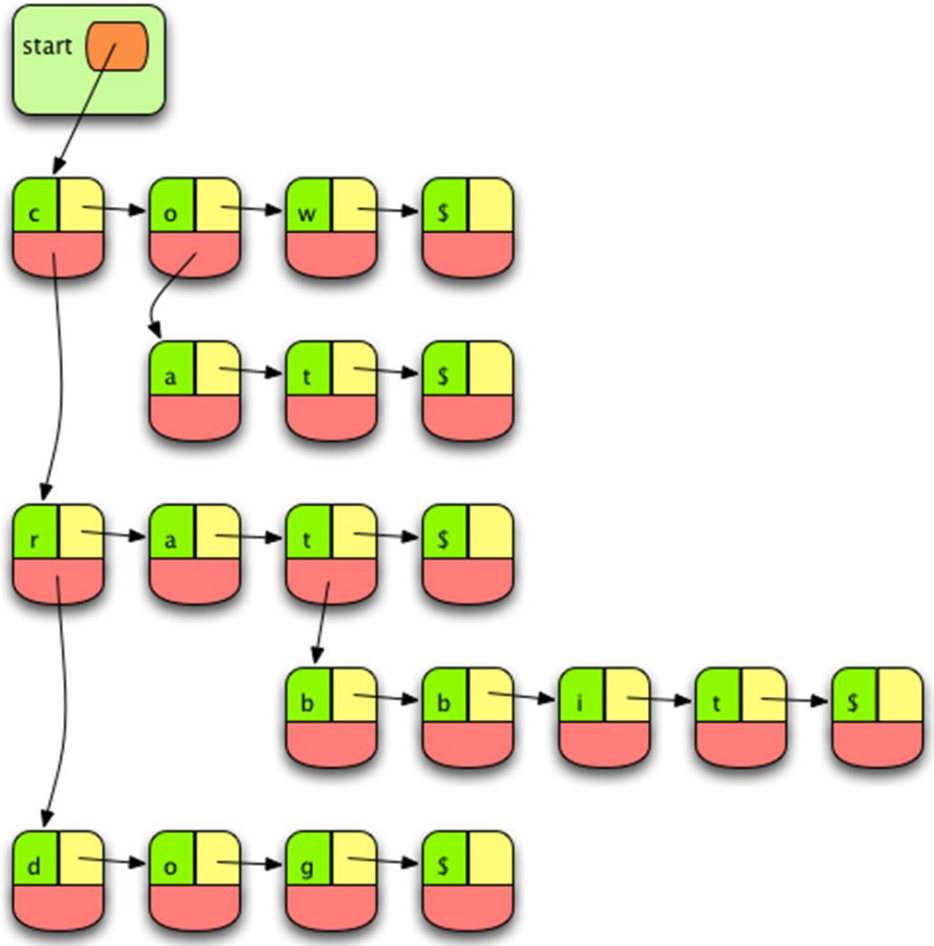


PATTERN MATCHING ALGORITHMS: TRIE DATA STRUCTURE (1)

- **Trie data structure.**
  - Designed specifically for **reTRIEvals**.
    - **Does not** support **deletions**, only retrievals based on the **key** value.
  - **Keys** are made up of **more than one unit** & **individual units overlap** with other items keys.
    - More overlaps = more compact structure.
  - Implemented as a **series of linked lists** making up a matrix.

PATTERN MATCHING ALGORITHMS: TRIE DATA STRUCTURE (2)

- **Each trie node consists of three values:**
  - **Unit of a key.**
  - **Follows pointer.**
    - Points to node that contains **next unit** within **same key**.
  - **Next pointer.**
    - Points to **next node** that contains **other unit** appearing in the **same position** within a **key**.
    - Used when there is **more than one** possible **next item** in sequence.
- Every inserted **sequence** is appended with **sentinel**.
  - Helps distinguishing between **prefix** & **sequence end**.
- **Keys** with **common prefix share** that **prefix** and are not **repeated**.



Trie data structure

PATTERN MATCHING ALGORITHMS: TRIE DATA STRUCTURE (3)

- **Insertion into trie:**

```
If empty key (no units left in key)
  Return None
If node is None
  Create new node with unit of the key
  Insert rest of the key into follows link
If first unit of key matches unit of current node
  Insert rest of the key into follows link of current node
Otherwise
  Insert key into next link of current node
```

- **Membership in trie:**

```
If key length is 0
  Return True (success)
If node is None
  Return False (failure)
If first unit of key matches unit of current node
  Check membership of rest of the key starting with follows node
Otherwise
  Check membership of key starting with next node
```

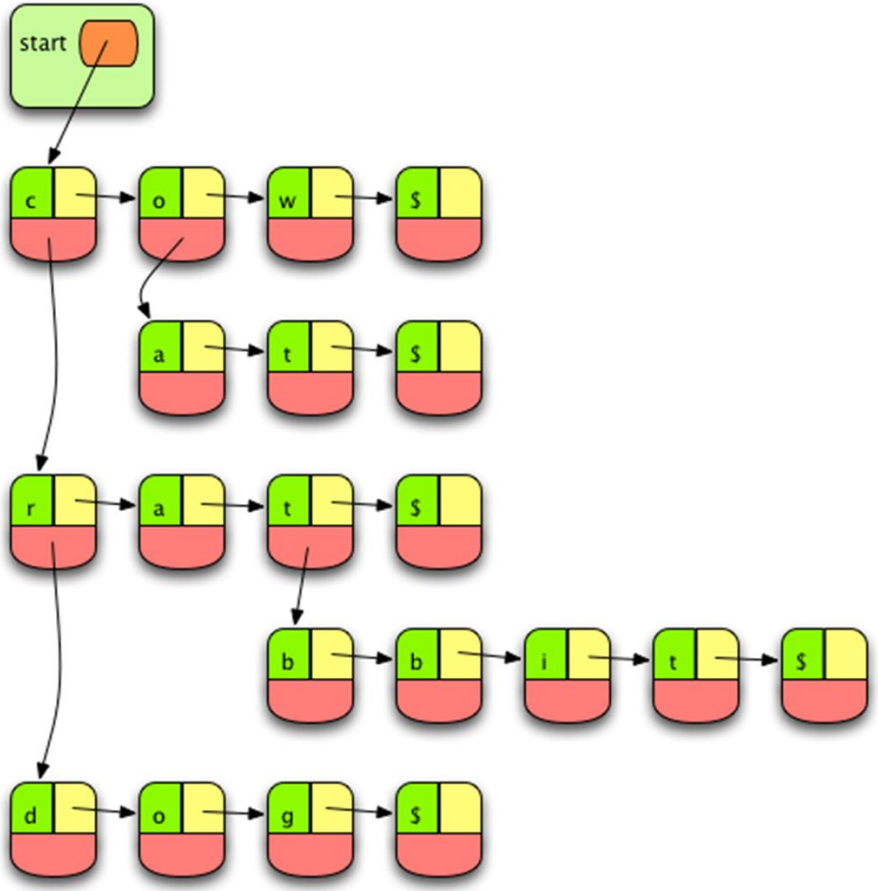
PATTERN MATCHING ALGORITHMS: TRIE DATA STRUCTURE (3)

- **Insertion into trie:**

```
If empty key (no units left in key)
  Return None
If node is None
  Create new node with unit of the key
  Insert rest of the key into follows link
If first unit of key matches unit of current node
  Insert rest of the key into follows link of current node
Otherwise
  Insert key into next link of current node
```

- **Membership in trie:**

```
If key length is 0
  Return True (success)
If node is None
  Return False (failure)
If first unit of key matches unit of current node
  Check membership of rest of the key starting with follows node
Otherwise
  Check membership of key starting with next node
```



Trie data structure