Matching Problems

(문자열) 부합 알고리즘

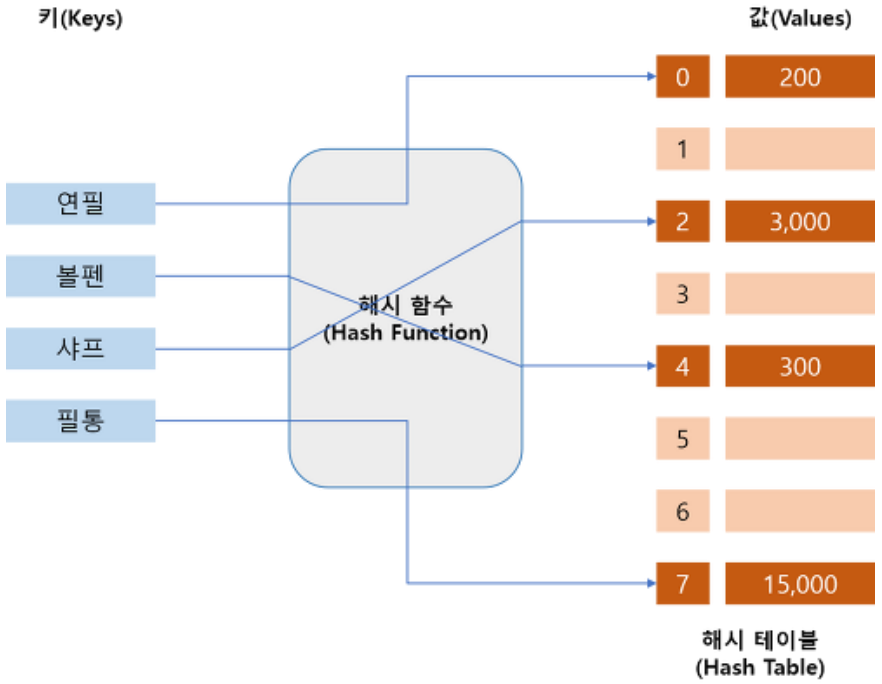
2020.07.26

CONTENS

- Hash-Table
    - Brute-force
    - KMP Algorithm
    - Karp-Rabin Algorithm
    - Boyer-Moore Algorithm
        - Suffix Array
        - Aho-corasick Algorithm
        - Z Algorithm

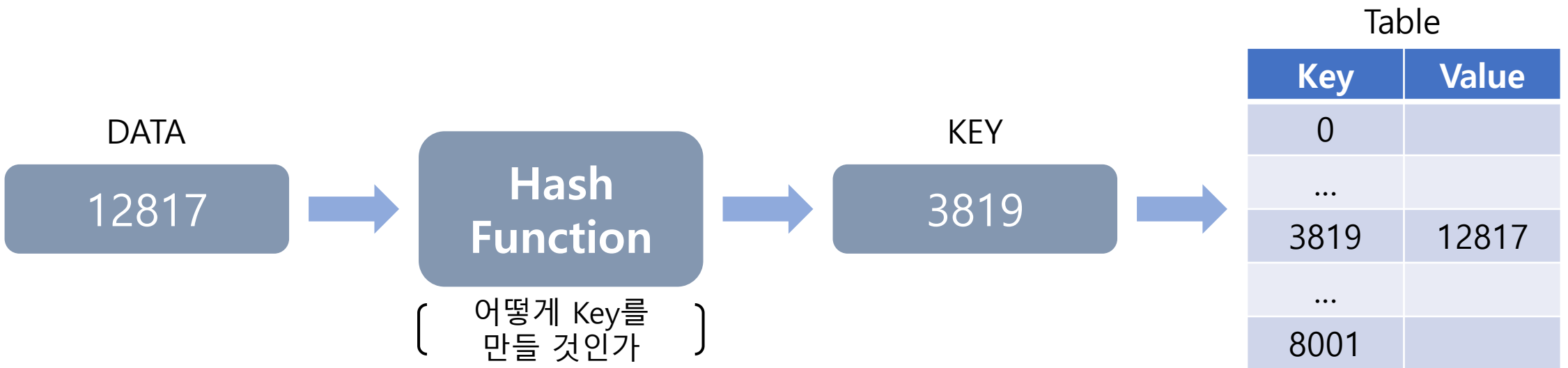
Hash-Table

- 배열의 성능을 극대화하는 자료구조, 이진 탐색보다 빠른 자료구조
- 데이터를 잘게 자른다는 의미, 잘게 자른 데이터는 Key라 부르고, 원본 데이터의 위치는 key 가 된다. key를 통해 Table에 저장된 원본 데이터에 바로 접근할 수 있다.
- 고정된 메모리를 차지한다.
- 시간 복잡도는 O(1)



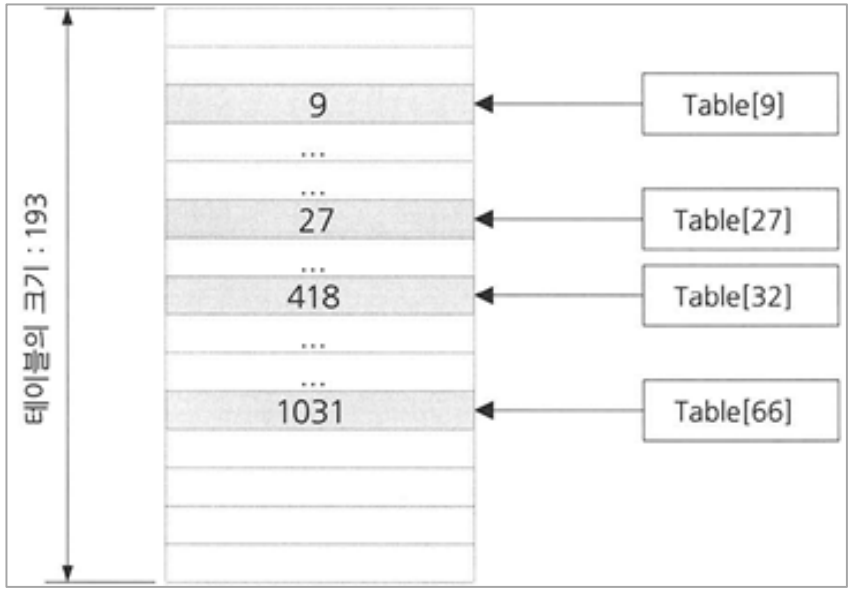
Hash-Table

- 배열의 성능을 극대화하는 자료구조, 이진 탐색보다 빠른 자료구조
- 데이터를 잘게 자른다는 의미, 잘게 자른 데이터는 Key라 부르고, 원본 데이터의 위치는 key 가 된다. key를 통해 Table에 저장된 원본 데이터에 바로 접근할 수 있다.
- 고정된 메모리를 차지한다.
- 시간 복잡도는 O(1)

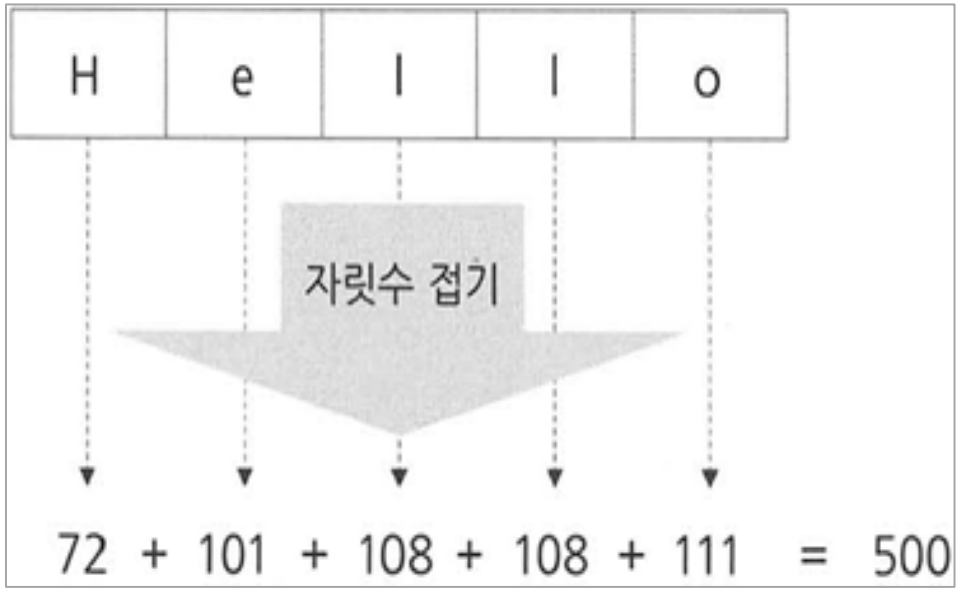


Hash-Table

- 나눗셈법 : 원본 데이터를 정수로 바꾼 다음 저장공간의 크기만큼 나눈 나머지로 key를 구하는 방법
- 자릿수 접기 : 원본 데이터의 자릿수 끼리 연산하여 key를 구하는 방법



주소 = 입력값 % 테이블 크기
어떤 값이든 테이블의 크기로 나누면 그 나머지는 절대 테이블의 크기(n)을 넘지 않는다.

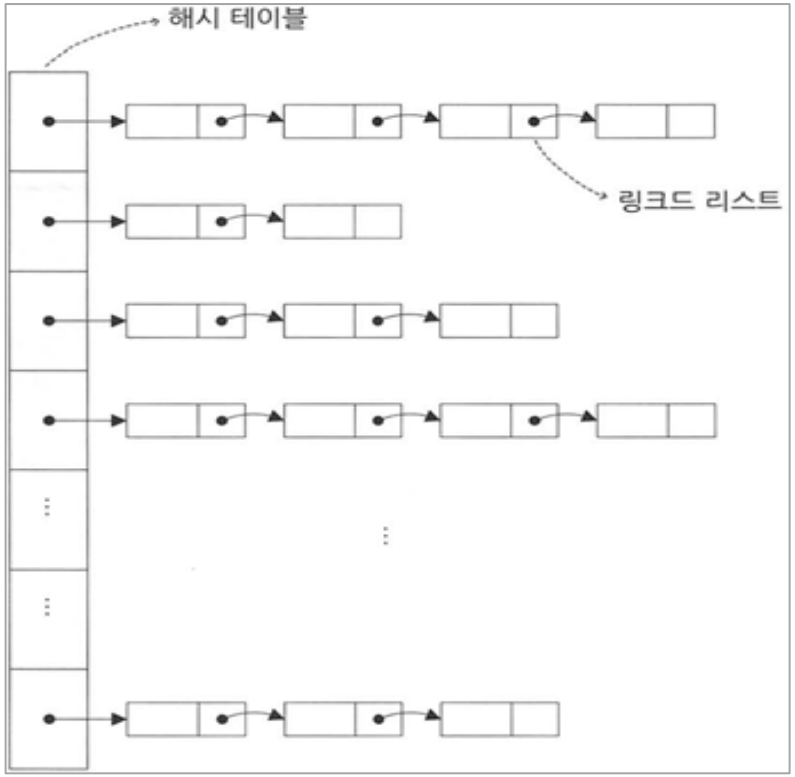


ASCII 코드를 기준으로 문자를 정수화

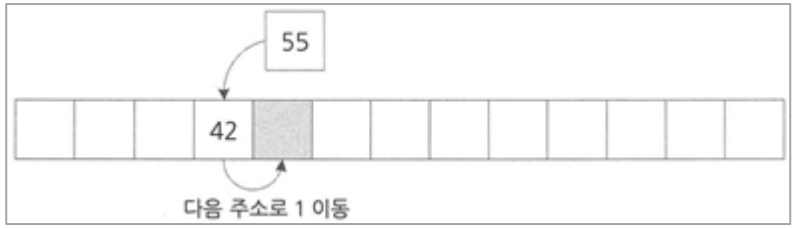
Hash-Table

- 해시 충돌 : 다른 데이터가 같은 key가 만들어지는 경우
- 클러스터 : 집중적으로 같은 key가 만들어지는 경우
- 회피 방법 : 체이닝, 선형탐색, 제곱탐색, 재해싱

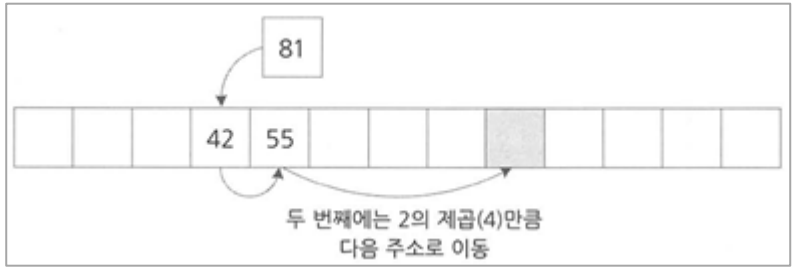
체이닝



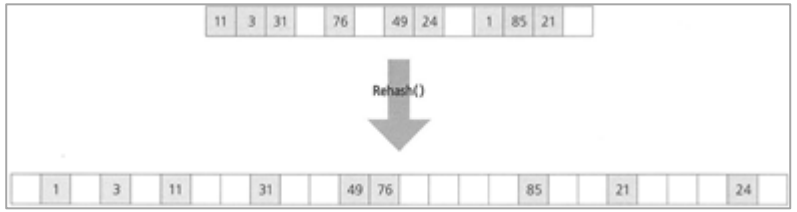
선형탐색



제곱탐색

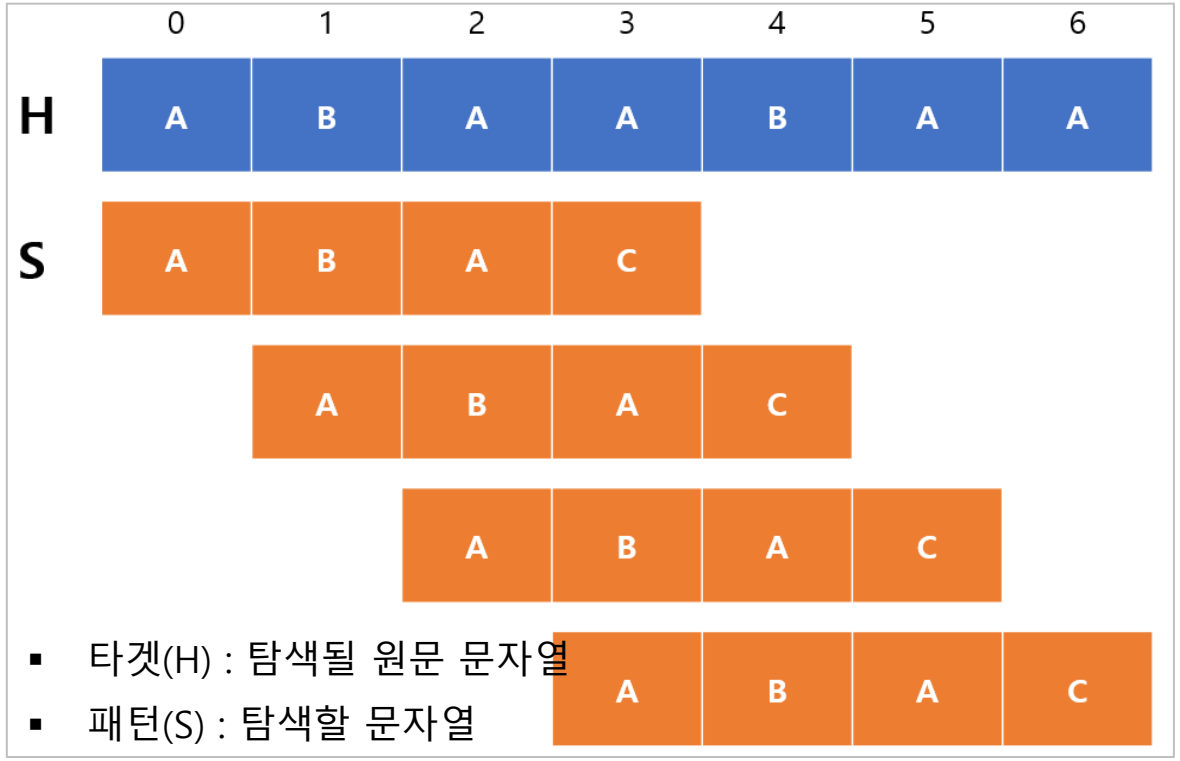


재해싱



Brute-force

- 시간 복잡도 : **O(P * T)**
- 구현이 간단하지만 이동거리 효율성이 떨어짐
  - 이동거리를 줄여서 시간 복잡도를 줄이는 알고리즘이 필요



Karp-Rabin Algorithm

해시 함수를 이용한 문자열 탐색

- 패턴을 해시 값으로 바꾸어 비교 연산을 획기적으로 줄일 수 있다.
- 해시 값과 해시 값을 비교한다. (H = H)

- 시간 복잡도 : **O(T) (T는 본 문자열의 길이)**
- **해시 충돌**이 일어날 수 있으므로 문자열 부합도 맞춰야 한다.

문자열 : "abacdab"

$= 97 * 2^6 + 98 * 2^5 + 97 * 2^4 + 99 * 2^3 + 100 * 2^2 + 97 * 2^1 + 98 * 2^0$

= 12,380

부모 해시 값: 12,398

패턴 해시 값: 12,380

부모 문자열	a	c	a	b	a	c	d	a	b	a	c
패턴 문자열	a	b	a	c	d	a	b				

라빈 카프 문자열 매칭

부모 해시 값: 12,477

패턴 해시 값: 12,380

부모 문자열	a	c	a	b	a	c	d	a	b	a	c
패턴 문자열	a	b	a	c	d	a	b				

라빈 카프 문자열 매칭

부모 해시 값: 12,380 => **문자열 매칭 발생!**

패턴 해시 값: 12,380

부모 문자열	a	c	a	b	a	c	d	a	b	a	c
패턴 문자열	a	b	a	c	d	a	b				

Karp-Rabin Algorithm

< S[n]은 본 문자열의 n번째 문자 / H[n]은 문자열의 해시 값 >

H(0) = S[0]*2^6 + S[1]*2^5 + S[2]*2^4 + S[3]*2^3 + S[4]*2^2 + S[5]*2^1 + S[6]*2^0

H(1) = S[1]*2^6 + S[2]*2^5 + S[3]*2^4 + S[4]*2^3 + S[5]*2^2 + S[6]*2^1 + S[7]*2^0

H(2) = S[2]*2^6 + S[3]*2^5 + S[4]*2^4 + S[5]*2^3 + S[6]*2^2 + S[7]*2^1 + S[8]*2^0

H(3) = S[3]*2^6 + S[4]*2^5 + S[5]*2^4 + S[6]*2^3 + S[7]*2^2 + S[8]*2^1 + S[9]*2^0

정리 : 다음 해시 값 = 2 * (현재 해시 값 - 가장 앞에 있는 문자의 수치) + 새 문자의 수치

$$H_i = \begin{cases} (S[i]\times 2^{m-1} + S[i+1]\times 2^{m-2} + \cdots + S[i+m-2]\times 2^1 + S[i+m-1]\times 2^0 \bmod q, & i=0 \\ (2(H_{i-1} - S[i-1]\times 2^{m-1}) + S[i+m-1]) \bmod q, & i>0 \end{cases}$$

문자열 : "abacdab"

$= 97 * 2^6 + 98 * 2^5 + 97 * 2^4 + 99 * 2^3 + 100 * 2^2 + 97 * 2^1 + 98 * 2^0$

= 12,380

부모 해시 값: 12,398

패턴 해시 값: 12,380

부모 문자열	a	c	a	b	a	c	d	a	b	a	c
패턴 문자열	a	b	a	c	d	a	b				

라빈 카프 문자열 매칭

부모 해시 값: 12,477

패턴 해시 값: 12,380

부모 문자열	a	c	a	b	a	c	d	a	b	a	c
패턴 문자열	a	b	a	c	d	a	b				

라빈 카프 문자열 매칭

부모 해시 값: 12,380 => **문자열 매칭 발생!**

패턴 해시 값: 12,380

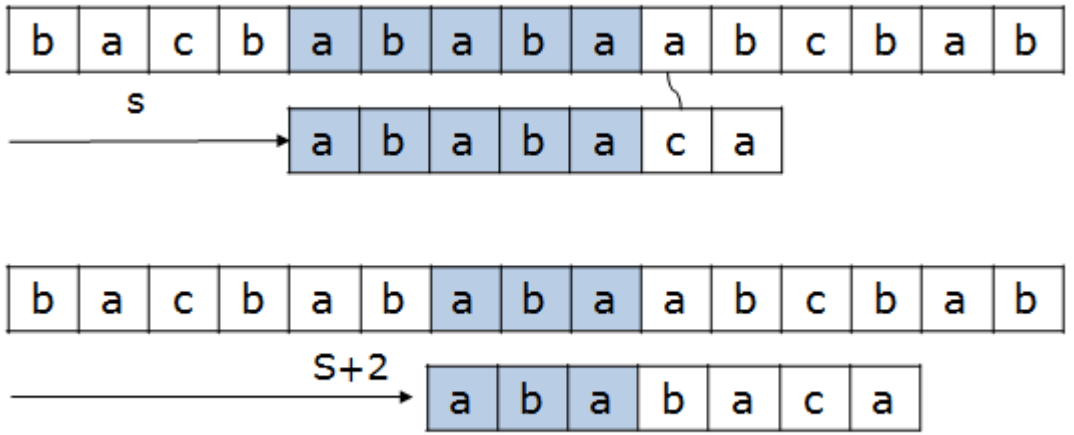
부모 문자열	a	c	a	b	a	c	d	a	b	a	c
패턴 문자열	a	b	a	c	d	a	b				

KMP Algorithm

접두사와 접미사를 활용해 탐색 이동거리를 최소화하는 알고리즘

- 경계 : 접두사와 접미사가 쌍인 것
- 탐색 불일치 시 이동 거리 = 일치한 문자열 수 - [불일치 위치 이전의 문자열 경계너비]
- 성능 : **O(P + T)**
- 패턴에 대한 경계 해시 테이블이 필요함

패턴	최대 경계너비
a	-1
ab	0
aba	1
abab	2
ababa	3
ababac	0
ababaca	1



- 접두사 : 문자열의 머리
- 접미사 : 문자열의 꼬리

Boyer-Moore Algorithm

- 패턴의 접미사에서 비교를 시작하는 문자열 탐색 알고리즘 **O(n) , 최악 O(MN)**
- [**공통**] 접미사에서 패턴과 본 문자열이 일치 하지 않을 시 이동거리가 패턴의 길이 만큼 이동한다.

- 나쁜 문자 이동

- 불일치 시 패턴의 접미사에서부터 탐색하여 **일치하지 않는 문자 (Bad character)**가 있을 시 패턴에서 동일한 나쁜 문자를 찾아낸다.
- 찾은 패턴의 나쁜 문자와 본 문자열의 나쁜 문자의 위치를 동일하게 맞추어 문자열을 배치한다.
- 문제점 : 본 문자열의 나쁜 문자가 패턴의 나쁜 문자보다 보다 앞에 있을 경우

Suffix Table

- 문자열 문제들을 해결할 수 있는 자료구조
- 한 문자열의 모든 접미사들을 모아둔 배열
- 1) 불일치 시 본 문자열의 접미사를 찾아 패턴에서 동일한 접미사의 위치를 본 문자열의 접미사 위치에 이동시킨다.
  => 없을 경우 (2)
- 2) 문자열의 접미사가 패턴에 없을 시 접미사의 접미사로 탐색을 한다.
  => 없을 경우 패턴의 길이만큼 이동

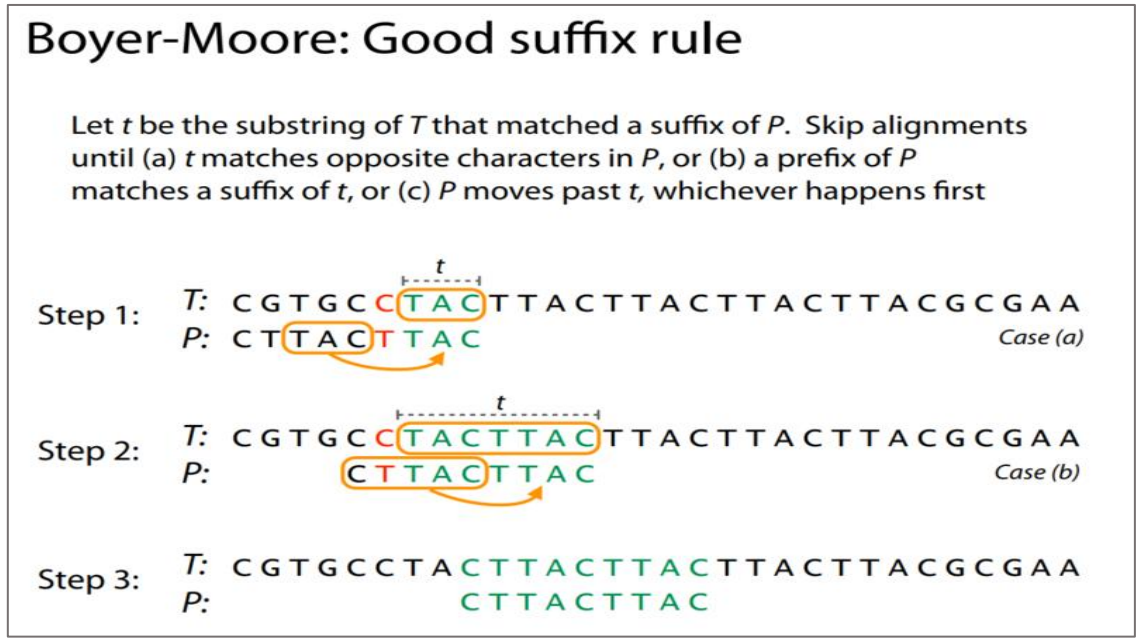
이동 거리 : 오른쪽에서 왼쪽으로 읽은 접미사의 시작 위치 – 이 접미사의 경계로 가지는 패턴 내 가장 큰 접미사의 시작 위치

위치	1	2	3	4	5	6	7	8	9	10	이동 거리
패턴	C	T	T	A	C	T	T	A	C	$	**8**
		T	**T**	**A**	**C**	**T**	**T**	**A**	**C**	$	**4**
			T	**A**	**C**	T	**T**	**A**	**C**	$	**4**
				A	C	T	T	**A**	**C**	$	**4**
					C	T	T	A	**C**	$	**4**
						T	T	A	C	$	**8**
							T	A	C	$	**8**
								A	C	$	**8**
									C	$	**8**
										$	1

Boyer-Moore Algorithm

- 패턴의 접미사에서 비교를 시작하는 문자열 탐색 알고리즘 **O(n) , 최악 O(MN)**
- [**공통**] 접미사에서 패턴과 본 문자열이 일치 하지 않을 시 이동거리가 패턴의 길이 만큼 이동한다.

- 착한 문자 이동

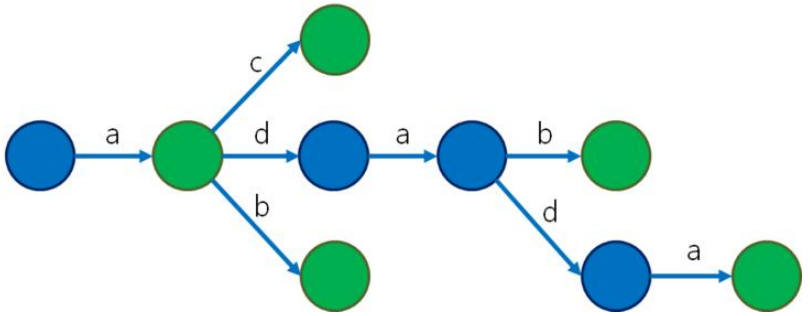


- 패턴의 **접미사 테이블(Suffix Table)**을 만들어야 한다.
- 불일치 시 본 문자열의 접미사를 찾아 패턴에서 동일한 접미사의 위치를 본 문자열의 접미사 위치에 이동시킨다.
- 문자열의 접미사가 패턴에 없을 시 접미사의 접미사로 탐색을 한다.

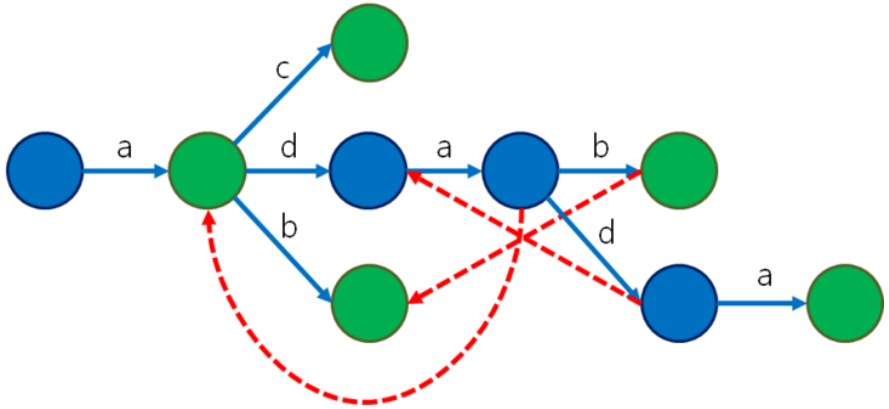
Aho-corasick Algorithm

- **아호코라식**(Aho-Corasick)은 현재 광범위하게 알려진 거의 유일한 **일대다 패턴매칭** 알고리즘.
- 문자열 하나 안에 여러 각각의 문자열이 존재하는지를 다 판별하는 것.
- 문자열 S에서, 찾을 단어 집합 W의 각 단어 w1, w2, ... , wk를 찾는다고 하면 S의 길이를 N, 각 단어의 길이를 m1, m2, ... mk라 하면 **O(N+m1+m2+...+mk)**

W = {a, ab, ac, adab, adada}



W = {a, ab, ac, adab, adada}



Z Algorithm

문자열 내에서 패턴의 위치를 모두 탐색하는 알고리즘

- 선형탐색으로 O(n)의 성능을 가짐
- Z배열을 생성하여 Z[i]의 값이 패턴의 길이와 같다면 패턴은 Z[i]에 있다.
- 패턴과 문자열의 접두사의 너비(길이)를 구한다.

Index:	0	1	2	3	4	5	6	7	8
Text:	a	b	c	a	b	g	a	b	c
Pattern:	a	b	c						

Press x to search for pattern in text using Z algorithm.

Concatenate the pattern and the text with a marker character between and compute Z for the new string.

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12
Z Text:	a	b	c	$	a	b	c	a	b	g	a	b	c
Z Values:		0	0	0	3	0	0	2	0	0	3	0	0

15 comparisons used to construct Z values.

Now the pattern can be matched by looping through the Z array after element 3 and selecting Z = 3.

Z algorithm pattern match has completed with 2 match(es) and 24 comparisons.

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12
Z Text:	a	b	c	$	a	b	c	a	b	g	a	b	c
Z Values:		0	0	0	3	0	0	2	0	0	3	0	0