외부 메모리에서 문자열을 효율적으로 탐색하기 위한 인덱스 자료 구조

(An Index Data Structure for String Search in External Memory)

나 중 채 † 박 근 수 ††
(Joong Chae Na) (Kunsoo Park)

요 약 본 논문에서는 새로운 외부 메모리 인덱스 자료 구조인 접미사 B-tree를 제안한다. 접미사 B-tree는 String B-tree와 마찬가지로 문자열을 키로 가지는 B-tree이다. String B-tree의 노드는 복잡한 Patricia trie로 구현된 반면, 접미사 B-tree의 노드는 일반적인 B-tree처럼 배열로 구현되어 보다 간단하고 구현하기 쉽다. 그럼에도 불구하고 접미사 B-tree에서 배열을 이용하여 String B-tree만큼 효율적으로 분기를 찾을 수 있다. 결과적으로 문자열 알고리즘 분야에서 기본적이고 중요한 문제인 문자열 매칭을 String B-tree와 동일한 디스크 접근을 사용하여 수행할 수 있다.

키워드 : 문자열 알고리즘, 외부 메모리 알고리즘, 인덱스 자료 구조, 패턴 매칭

Abstract We propose a new external-memory index data structure, the Suffix B-tree. The Suffix B-tree is a B-tree in which the key is a string like the String B-tree. While the node in the String B-tree is implemented with a Patricia trie, the node in the Suffix B-tree is implemented with an array. So the Suffix B-tree is simpler and easier to be implemented than the String B-tree. Nevertheless, the branching algorithm of the Suffix B-tree is as efficient as that of the String B-tree. Consequently, the Suffix B-tree takes the same worst-case disk accesses as the String B-tree to solve the string matching problem, which is fundamental and important in the area of string algorithms.

Key words : String algorithms, external memory algorithms, index data structures, pattern matching

1. 서 론

길이 N인 문자열 T와 길이 M인 패턴 P가 주어졌을 때, 문자열 매칭은 T안에 발생하는 모든 P의 위치를 찾는 문제이다. 많은 응용에서 T는 고정되어 있는 반면, P는 다양하게 변하게 된다[1]. 이러한 경우 이 문제에 대한 효율적인 해결책은 T에 대한 인덱스를 만들어 놓고, 이 인덱스를 이용해 P가 발생하는 위치를 찾는 것이다. 이렇게 인덱스를 저장하고 탐색을 쉽게 하기 위한 자료 구조를 인덱스 자료 구조라 한다. 지금까지 접미사 트리[2-4], 접미사 배열[5], 접미사 오토마타[6], Patricia tries[7], 접미사 이진 탐색 트리[8] 등 많은 인덱스 자료구조들이 개발되어 왔다.

많은 컴퓨터 응용 프로그램들은 주 메모리보다 훨씬 많은 데이타를 저장하고 처리한다. 이러한 응용에서는 필요한 데이타를 디스크와 같은 외부 메모리에 저장하고 사용하는데, 이러한 외부 저장 장치의 입출력 속도는 주 메모리에 비해서 매우 느려서 프로그램 성능에 가장 큰 병목 원인이 된다. 따라서 이런 경우에는 디스크 입출력의 회수와 효율 등을 고려하여 알고리즘을 개발하여야 한다. 이렇게 외부 데이타의 위치, 이동 등을 명시적으로 고려하는 알고리즘과 자료 구조를 각각 외부 메모리 알고리즘, 외부 메모리 자료 구조라 한다[9].

인덱스 자료 구조의 경우에도 inverted files[10], B-trees[11], Prefix B-trees[12,13], String B-trees[14]처럼 외부 메모리를 고려한 자료 구조들이 개발되었다. 앞서 언급한 접미사 트리와 같은 자료구조들은 주 메모리 안에서의 성능에 초점이 맞추어져 개발되어 외

· 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 육성·지원사업의 연구결과로 수행되었음
† 비 회 원 : 서울대학교 전기.컴퓨터공학부
jcna@theory.snu.ac.kr
†† 종신회원 : 서울대학교 전기.컴퓨터공학부 교수
kpark@theory.snu.ac.kr
논문접수 : 2003년 10월 7일
심사완료 : 2005년 8월 2일

부 메모리를 사용할 경우 성능이 크게 저하되어 거의 사용할 수가 없게 된다. 하지만 이들 외부 메모리 인덱스 자료구조는 처음부터 외부 메모리를 고려하여 설계되었기 때문에 좋은 성능을 보인다. 그러나 String B-trees를 제외한 외부 메모리 인덱스 자료구조들은 처리할 문자열의 길이에 제약이 없는 일반적인 상황을 잘 처리하지 못한다는 단점을 가지고 있다.

String B-tree[14]는 기본적으로 B-tree의 구조를 가지고 있지만, 임의 길이의 문자열을 처리할 수 있고, 문자열 알고리즘에서 발생하는 다양한 문자열 탐색을 효율적으로 수행할 수 있다. B-tree의 탐색 과정에는 원하는 키 값이 속해있는 subtree를 찾는 분기 과정이 필요한데 String B-tree에서는 노드의 내부를 Patricia tries를 이용하여 구현함으로써 분기를 효율적으로 찾을 수 있도록 하였다. String B-tree는 이론적으로 기존의 자료구조보다 좀더 효율적인 인덱스와 탐색을 지원하는 반면, 자료 구조가 너무 거대하고, 복잡하다는 단점을 가지고 있다.

본 논문에서는 "*접미사 B-tree*"라 불리는 새로운 자료구조를 제안한다. 접미사 B-tree는 String B-tree보다 간단하고 구현하기 쉬우면서 문자열 알고리즘에서 가장 기본적이고 중요한 패턴 탐색은 효율적으로 지원한다. 접미사 B-tree는 노드 내부의 자료 구조로서 Patricia tries 대신 구현이 쉽고 필요한 공간이 일정한 "배열"을 사용하였다. 그럼에도 불구하고 배열보다 복잡한 trie 구조를 사용한 String B-tree만큼 분기를 효율적으로 할 수 있다. 디스크의 한 블록의 크기를 B, 패턴의 발생 회수를 occ라 하면, 본문에서 설명할 분기 알고리즘을 이용하여 String B-tree와 동일한 $O\left(\log_B N+\frac{(M+occ)}{B}\right)$번의 디스크 접근을 사용하여 패턴을 탐색할 수 있다. 이 분기 알고리즘과 String B-tree의 생성 알고리즘을 결합하면 String B-tree와 동일한 디스크 접근을 사용하여 접미사 B-tree를 생성할 수 있다.

논문은 다음과 같이 구성된다. 2절에서는 접미사 B-tree의 정의와 특징에 대해서 설명한다. 3절에서는 접미사 B-tree를 이용하여 패턴을 탐색하는 방법과 탐색의 핵심 알고리즘인 분기 알고리즘에 대해 설명한다. 그리고 접미사 B-tree의 생성 방법에 대해서 간략히 설명하고, 4절에서 결론을 내린다.

2. 접미사 B-tree

문자열 T를 알파벳 Σ로 이루어진 길이 N인 문자열이라 하면, T는 $\lceil N/B \rceil$ 개의 디스크 블록에 저장된다. T의 i번째 문자를 $T[i]$, i번째부터 j번째까지의 부분 문자열을 $T[i..j]$로 표기한다. 알고리즘의 단순화를 위해 T의 마지막 문자는 T의 다른 위치에서는 나타나지 않는 특수 문자 '$'라고 가정한다. 문자 '$'는 Σ-'$'의 어떤 문자보다 사전 순으로 작다. 임의의 $1 \leq i \leq N$에 대하여 $T[1..i]$과 $T[i..N]$을 각각 T의 접두사와 접미사라 부르고, 특별히 접미사 $T[i..N]$을 $suffix(i)$라 표기하고 i를 접미사 번호라 칭한다.

여러 문자열이 있을 때 다음과 같이 순서 관계를 정의한다. 만약 문자열 α가 문자열 β보다 사전 순으로 앞에 있으면, $\alpha < \beta$라 표기하고, α는 β보다 작고, β는 α보다 크다라고 말한다. 문자열 α가 문자열 β보다 짧고 α가 β의 접두사이면 사전 순으로 α가 β보다 먼저 나오므로, $\alpha < \beta$이다.

두 문자열 α와 β의 가장 긴 공통 접두사를 lcp (longest common prefix)라 하고 이의 길이를 $lcp(\alpha, \beta)$로 표기한다. 즉, $\alpha[1..i] = \beta[1..i]$이고 $\alpha[i+1] \neq \beta[i+1]$이면, $lcp(\alpha, \beta)$는 i이고, 역도 성립한다. 정렬된 문자열 $\alpha_h < \alpha_{h+1} < \ldots < \alpha_j$에 대해서, $lcp(\alpha_h, \alpha_j) = \min_{h < i \leq j} lcp(\alpha_{i-1}, \alpha_i)$이다[5,15].

이 성질로부터 다음 보조 정리를 얻을 수 있다.

보조정리 1. 서로 다른 세 문자열 α, β_1, β_2에 대해서 $\beta_1 < \beta_2$, $lcp(\beta_1, \beta_2) = l$이고 $lcp(\alpha, \beta_1) = l' > l$이면, $\alpha < \beta_2$이고 $lcp(\alpha, \beta_2) = l$이다.

증명. α의 순서에 따라서 다음 세 가지 경우로 나누어 생각해보자.

i) $\alpha < \beta_1 < \beta_2$인 경우:

$lcp(\alpha, \beta_2) = \min(lcp(\alpha, \beta_1), lcp(\beta_1, \beta_2)) = \min(l', l) = l$

이다.

ii) $\beta_1 < \alpha < \beta_2$인 경우:

$lcp(\beta_1, \beta_2) = \min(lcp(\beta_1, \alpha), lcp(\alpha, \beta_2))$이다. 정리의 조건에 의해서 이 식은 $l = \min(l', lcp(\alpha, \beta_2))$과 동치이다. $l' > l$이므로, 이 식이 성립하려면 $lcp(\alpha, \beta_2)$는 l이어야 한다.

iii) $\beta_1 < \beta_2 < \alpha$인 경우:

$\min(lcp(\beta_1, \beta_2), lcp(\beta_2, \alpha)) = \min(l, lcp(\beta_2, \alpha)) \leq l$

이므로, $lcp(\alpha, \beta_1)$은 최대 l이다. 이는 조건 $lcp(\alpha, \beta_1) = l' > l$에 모순. 따라서 이 경우는 발생할 수 없다.

i), ii), iii)에 의해서 위 정리가 성립함을 알 수 있다. □

보조정리 2. 세 문자열 α, β_1, β_2에 대해서 $\beta_1 < \beta_2$, $lcp(\beta_1, \beta_2) = l$일 때,

A. $\beta_2[l+1] < \alpha[l+1]$이거나 $\beta_2[l+1] = \alpha[l+1]$이면, $lcp(\alpha, \beta_1) \leq lcp(\alpha, \beta_2)$이고,

B. $\beta_2[l+1] > \alpha[l+1]$이면, $lcp(\alpha,\beta_1) \geq lcp(\alpha,\beta_2)$이다.

증명. $\beta_1 < \beta_2$, $lcp(\beta_1, \beta_2) = l$이므로 $\beta_1[1..l] = \beta_2[1..l]$, $\beta_1[l+1] < \beta_2[l+1]$이다. 다음 4 가지 경우로 나누어서 생각해 보자.

i) $\beta_1[1..l] \neq \alpha[1..l]$인 경우:
$lcp(\alpha,\beta_1)$을 l_1이라 할 때, $l_1 < l$이다. 즉, $\alpha[1..l_1] = \beta_1[1..l_1] = \beta_2[1..l_1]$이고 $\alpha[l_1+1] \neq \beta_1[l_1+1] = \beta_2[l_1+1]$이다. 따라서 이 경우에는 $l+1$번째 문자의 순서에 관계없이 $lcp(\alpha,\beta_1) = lcp(\alpha,\beta_2) < l$이다.

ii) $\beta_1[1..l] = \alpha[1..l]$이고, $\beta_2[l+1] < \alpha[l+1]$인 경우:
$\beta_1[l+1] < \beta_2[l+1] < \alpha[l+1]$이므로 $lcp(\alpha,\beta_1) = lcp(\alpha,\beta_2) = l$이다.

iii) $\beta_1[1..l] = \alpha[1..l]$이고, $\beta_2[l+1] = \alpha[l+1]$인 경우:
$\beta_1[l+1] < \beta_2[l+1] = \alpha[l+1]$이므로 $lcp(\alpha,\beta_1) = l$이고, $lcp(\alpha,\beta_2) > l$이다.

iv) $\beta_1[1..l] = \alpha[1..l]$이고, $\beta_2[l+1] > \alpha[l+1]$인 경우:
$\beta_2[l+1] > \alpha[l+1]$이므로 $lcp(\alpha,\beta_2) = l$이다. 하지만, $lcp(\alpha,\beta_1)$은 $\beta_1[l+1] = \alpha[l+1]$일 경우 l보다 크고, $\beta_1[l+1] \neq \alpha[l+1]$인 경우에는 l이 된다. 즉, $lcp(\alpha,\beta_1) \geq l$이다. 따라서 $lcp(\alpha,\beta_1) \geq lcp(\alpha,\beta_2)$이 성립한다.

i), ii), iii)에 의해서 A가 성립함을 알 수 있고, i), iv)에 의해 B가 성립함을 알 수 있다. 이 정리의 역은 성립하지 않는다. □

문자열 T에 대한 접미사 B-tree는 T의 모든 접미사를 key로 저장하고 있는 B-tree로 다음과 같은 성질을 만족한다.

1. 각 노드 x는 다음과 같은 데이타를 저장한다.
   a. $n[x]$: 현재 노드 x에 저장되어 있는 접미사 번호의 개수
   b. $n[x]$개의 접미사 번호: 접미사 번호는 자신이 나타내는 접미사의 오름차순으로 저장된다. 즉, 노드의 i번째에 저장된 접미사 번호를 $k_i[x]$라 하면 다음 식이 성립한다.

$$suffix(k_1[x]) < suffix(k_2[x]) < \cdots < suffix(k_{n[x]}[x])$$

* 알고리즘의 기술을 간단히 하기 위해 $suffix(k_0[x])$와 $suffix(k_{n[x]+1}[x])$를 다음과 같이 정의한다. 하지만 $k_0[x]$와 $k_{n[x]+1}[x]$는 노드 x에 저장되어 있지는 않다.

노드 x의 부모노드를 z라 할 때. 다음을 만족시키는 인덱스 j가 존재한다.

$$suffix(k_j[z]) < suffix(k_1[x]) < , \ 0 \leq j \leq n[z]$$
$$\cdots < suffix(k_{n[x]}[x]) < suffix(k_{j+1}[z])$$

이 때, $suffix(k_0[x])$는 $suffix(k_j[z])$이고 $suffix(k_{n[x]+1}[x])$는 $suffix(k_{j+1}[z])$이다.

root 노드의 경우, $suffix(k_0[root])$는 빈 문자열, $suffix(k_{n[root]+1}[root])$는 Σ^*의 어떤 문자열 보다 큰 가상의 문자열이라고 간주한다.

   c. $n[x]+1$개의 lcp의 길이 $lcp_i[x]$:

$$lcp_i[x] = lcp(suffix(k_{i-1}[x]), suffix(k_i[x])) \quad \text{for} 1 \leq i \leq n[x]+1$$

   d. $n[x]$개의 문자 $lnc_i[x]$(lcp next character): $lnc_i[x]$는 $suffix(k_i[x])$의 $lcp_i[x]+1$번째 문자를 나타낸다. 이 값은 $1 \leq i \leq n[x]$에 대해서 정의된다.

2. x는 자식들을 가리키는 $n[x]+1$개의 포인터 $c_1[x]$, $c_2[x], \cdots, c_{n[x]+1}[x]$를 포함한다. 만약 x가 leaf이면, 이 포인터는 NIL값을 가진다.
3. $suffix(k_i[x])$는 각 subtree에 저장되는 접미사의 범위를 나눈다:
   만약 suf_i가 노드 $c_i[x]$를 root로 하는 subtree에 저장된 임의의 접미사라면,
   $suf_1 < suffix(k_1[x]) < suf_2 < suffix(k_2[x]) < \cdots < suffix(k_{n[x]}[x]) < suf_{n[x]+1}$이다.
4. 모든 leaf는 같은 depth를 가지고, 이 depth는 tree의 높이 즉, $O(\log N)$이다.
5. 한 노드에는 $t-1$개 이상, $2t-1$개 이하의 키가 저장되어 있다. 따라서 적어도 t개, 많아야 $2t$개의 자식을 가진다.

표기를 간단히 하기 위해 앞으로 노드를 표시하는 $[x]$는 혼동이 없는 범위에서 생략한다.

그림 1은 하나의 디스크 페이지에 저장되는 노드 x의 자료구조이다. 그림 2에서는 각 k_i가 나타내는 접미사가 그림과 같을 때 lcp_i와 lcs_i의 의미를 각각 직사각형과 원으로 나타내고 그 값을 표시하였다. 예를 들어

lcp_1	k_1	lcp_2	k_2	$\cdots$	k_i	$\cdots$	k_{n-1}	lcp_n	k_n	lcp_{n+1}
c_1	lnc_1	c_2	lnc_2	$\cdots$	lnc	$\cdots$	lnc_{n-1}	c_n	lnc_n	c_{n+1}

그림 1 node x의 구조

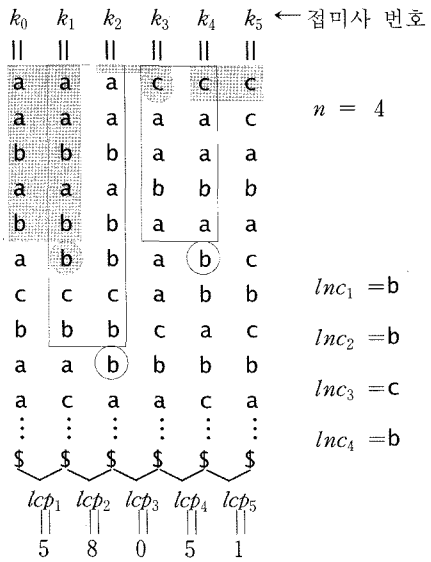

그림 2 노드에 저장되는 데이타의 예

가장 왼쪽의 회색 직사각형은 $suffix(k_0)$와 $suffix(k_1)$의 lcp를 나타내고, 가장 왼쪽의 회색 원은 lnc_1을 나타낸다. 따라서 lcp_1은 5이고, lnc_1은 'b'임을 알 수 있다.

접미사 B-tree는 실수를 키로 가지는 일반적인 B-tree와 다음과 같은 차이를 가진다.

첫째, 수가 아닌 접미사라는 문자열을 키로 가진다. 접미사는 길이가 다양하고 매우 길 수 있으므로, 접미사 그 자체가 아닌 접미사 번호만을 노드에 저장한다.

둘째, 탐색을 효율적으로 하기 위해 보조 데이타 lcp_i, lnc_i를 노드에 저장한다. 일반적인 B-tree에서는 키 자체가 노드에 저장되어 있고, 키를 컴퓨터의 한 word로 표시할 수 있으므로 키의 비교를 쉽게 할 수 있지만, suffix B-tree에서는 접미사의 문자 단위로 비교하기 위해 실제 키인 접미사를 디스크로부터 읽어들여야 하므로, 문자 단위의 비교 회수와 그에 필요한 디스크 액세스 횟수를 줄여야한다. 다음 절에서 위의 보조 데이타를 이용해 탐색을 효율적으로 하는 방법을 설명한다.

3. 탐색 알고리즘

이번 절에서는 접미사 B-tree를 이용해 문자열 매칭 문제를 어떻게 풀 수 있는 지를 설명한다. 문자열 알고리즘에서 매칭 문제는 텍스트 $T[1..N]$과 패턴 문자열 $P[1..M]$이 주어졌을 때, 텍스트 T에서 P가 나타나는 모든 발생 위치(occurrence)를 찾는 것이다. 좀 더 간단한 버전으로 패턴이 나타나는지 나타나지 않는지 만을 결과로 보이는 결정 문제(decision problem)도 있다. 접미사 B-tree를 이용하여 간단한 버전인 결정 문제를 푸는 방법을 우선 설명하고, 이를 확장하여 원래 문제를 푸는 방법을 설명한다.

3.1 탐색 알고리즘 개요

접미사 B-tree에서 패턴을 찾기 위해서 트리의 root 부터 leaf 방향으로 패턴 P로 시작하는 T의 접미사를 찾는다. 패턴 P가 T의 i번째 위치에서 발생하면 P는 $suffix(i)$의 접두사가 되고, 역도 역시 성립한다. P로 시작하는 접미사를 P-접미사라고 칭하자. 임의의 P-접미사를 α라 하고, P-접미사가 아니면서 P-접미사보다 작은 T의 임의의 접미사를 β라 하면 $\beta < P < \alpha$의 관계가 성립한다. 따라서 일반적인 이진 tree나 B-tree에서처럼 root 노드에서부터 leaf 노드 방향으로 P가 위치할 수 있는 분기를 찾아나간다.

그림 3은 탐색 알고리즘의 전체적인 구조를 나타내는 pseudo code이다. 알고리즘의 기술을 간단히 하기 위해서 P의 끝에 P와 T에는 나타나지 않는 특수 문자 '#'을 붙인다. 이 특수 문자 '#'은 Σ의 어떤 문자보다 사전 순으로 작다고 간주한다. 즉, 문자 '#'은 '$'보다도 작다. 아래 코드에서FindBranch(x, l, mode, P)는 $suffix(k_{j-1}) < P < suffix(k_j)$를 만족하는 j 즉, 다음 탐색을 할 분기 $c_j[x]$를 찾는 프로시저이다. 이 프로

```
1: SearchPattern (B, P)
2:    B: 텍스트 T의 접미사 B-tree
3:    P[1..M]: 찾고자 하는 패턴
4: begin
5:    P[M+1] ← '#'
6:    x ← root[B];  l ← 0;  mode ← left;
7:    while x ≠ NIL and l < M do
8:       (i, l, mode) ← FindBranch(x, l, mode, P)
9:       if l = M then    // P occurs in T
10:         FindAllOcc(c_i[x], right, M)
11:         do
12:            print k_i[x]
13:            FindAllOcc(c_{i+1}[x], left, M)
14:            i ← i+1;
15:         while i ≤ n[x] and lcp_i[x] ≥ M
16:      x ← c_i[x];
17:
18:   if l < M then
19:      print "P does not occur in T"
20: end
```

그림 3 SearchPattern 함수

시저의 반환 값 중 l 은 노드 x 의 접미사들과 P의 lcp 길이 중 최대 값이다. 따라서 이 값이 M과 같으면 노드 x에 P-접미사가 존재한다는 것을 뜻한다. 다른 인자와 반환 값은 분기 알고리즘을 기술할 때 자세히 설명한다. 9~15 번째 줄은 모든 발생 위치를 찾는 부분으로 3.4 절에서 설명한다.

3.2 분기 알고리즘

FindBranch를 구현하는 효율적인 알고리즘을 설명하기 전에 우선 단순한 알고리즘을 설명한다. 노드 x 에서 분기를 찾는 가장 간단한 방법은 패턴 P를 노드 x 의 첫 번째 접미사부터 차례로 접미사 $suffix(k_i)$와 P를 비교하면서, 처음으로 P보다 큰 접미사를 찾는 것이다. 일반적인 B-tree에서는 키가 하나의 수이므로 이 방법이 간단하고 좋으나, 접미사 B-tree에서는 접미사와 패턴을 비교하기 위해 접미사를 디스크로부터 읽어 들여야 하므로 최악의 경우 $n[x]$개의 접미사를 디스크로부터 읽어야 한다. 또한 하나의 접미사와 패턴을 비교할 때 패턴의 맨 마지막 문자만 다르다면, 하나의 접미사와 비교를 하기 위해 길이 M의 문자열을 디스크로부터 읽어야 하므로 $\lceil M/B \rceil$ 번의 디스크 접근이 필요하다. 따라서 하나의 노드에서 분기를 찾는데 최대 $n[x] \times \lceil M/B \rceil$ 번의 디스크 접근이 필요하다.

그러나 보조 데이타 lcp_i, lnc_i 를 이용하면 분기를 찾기 위해 필요한 디스크 접근 횟수를 줄일 수 있다. 이 개선된 알고리즘은 크게 3 단계로 구성되어 있다.

- 단계 A : lcp_i, lnc_i 를 이용하여 노드 x 에 저장된 접미사 중 P와의 lcp 길이가 가장 긴 접미사를 찾는다. 이 후보 접미사와 P의 실제 lcp 길이는 이 단계에서 알 수는 없으나, 노드 x 의 다른 접미사와 P의 lcp 길이는 후보 접미사와 P의 lcp 길이보다 크지 않다는 것을 안다.
- 단계 B : 선택된 접미사를 디스크로부터 읽어들여 mismatch가 발생할 때까지 문자 단위로 P와 비교한다.
- 단계 C : 앞의 비교결과를 바탕으로 실제 분기 위치 c_j 을 찾는다. 또한 그 결과로 우리는 노드 x 에서 P와 lcp가 가장 긴 접미사가 $suffix(k_j)$인지 $suffix(k_{j-1})$인지를 알 수 있다. 이 때 lcp의 길이를 l 이라 했을 때, 노드 x 의 다른 접미사와 P의 lcp 길이는 l 보다 같거나 작다. 이 정보는 다음 노드에서 분기를 찾을 때 필요한 정보가 된다.

다음 소절에서 각 단계에서 하는 일을 좀더 상세히 설명한다.

프로시저 FindBranch의 입력으로 패턴과 노드 이외에 다음 2가지가 필요하다.

1) $l = \max(lcp(P, suffix(k_0)),\ lcp(P, suffix(k_{n+1})))$

2) $\text{mode} = \begin{cases} left & \text{if } l = lcp(P,\ suffix(k_0)) \\ right & \text{otherwise} \end{cases}$

이는 P가 $suffix(k_0)$, $suffix(k_{n+1})$ 중 어느 접미사와 더 긴 lcp를 가지는지를 나타내는 정보로서, 노드 x 의 부모 노드에서 분기를 찾는 과정에서 얻어질 수 있다. 처음 분기 시작노드인 root에서는 l 은 0이고, $mode$는 $left$ 이다.

3.2.1 단계 A

단계 A에서는 노드 x 에 저장된 접미사 중 P와의 lcp 길이가 가장 긴 접미사를 찾는다. 이 단계에서는 접미사가 어떤 문자열인지 알지 못하기 때문에 P와의 lcp 길이는 알지 못하지만 노드 x 에 저장된 lcp_i, lnc_i 와 보조정리 2의 결과를 이용하면 이 lcp 길이들의 상대적인 크기를 알 수 있고, 이 상대적인 크기를 이용해 단계 A에서는 다음 조건을 만족하는 j를 찾는다.

$$lcp(P,\ suffix(k_j)) \geq lcp(P,\ suffix(k_i)) \text{ for } 0 \leq i \leq n+1.$$

단계 A에서는 $mode$에 따라 필요한 일이 약간 달라지는데 우선 $mode$가 $left$ 인 경우를 설명한다.

$mode$가 $left$ 인 경우:

노드에 저장된 가장 작은 접미사인 $suffix(k_1)$부터 최대 $suffix(k_n)$까지 오름차순으로 접미사와 P를 비교한다. 이 비교는 루프로 구성되고, lcp_i 와 lnc_i 만을 이용하여 이루어지며 $suffix(k_i)$를 디스크에서 읽어들이지는 않는다. $suffix(k_i)$와 비교하는 부분을 i 번째 반복(iteration)이라 하자. 루프를 실행하는 동안에 단계 A에서는 두 변수 $cand\_idx$, $mlcp$ 를 유지하고, 다음과 같은 loop invariant가 성립한다.

loop invariant :

i 번째 iteration 실행 전에 다음과 같은 조건을 만족한다.

1) $lcp(suffix(k_{i-1}), P) \geq l$

2) 두 변수 $cand\_idx$, $mlcp$ 는 다음 값을 유지한다.

- $cand\_idx$: 현재까지 비교된 접미사 중 P와의 lcp가 가장 긴 접미사 번호를 저장하는 원소의 index. 즉, 다음 식이 성립된다.

$$lcp(P,\ suffix(k_{cand\_idx})) \geq lcp(P,\ suffix(k_j)) \text{ for } 0 \leq j < i.$$

- $mlcp$: $suffix(k_{cand_idx})$와 $suffix(k_{i-1})$의 lcp 길이. 즉,

$$mlcp = \begin{cases} \min_{cand\_idx < j < i} \{lcp_j\} & \text{if } i > cand\_idx + 1. \\ -1 & \text{otherwise} \end{cases}$$

첫 번째 iteration 전에는 $cand_idx$은 0, $mlcp$ 는 -1

로 초기화한다. 이는 위의 invariant와 부합한다.

다음은 P와 $suffix(k_i)$의 lcp 길이를 P와 $suffix(k_{cand_idx})$의 lcp 길이와 비교하는 i번째 iteration으로 조건에 따라 크게 4가지의 경우로 나누어진다. 아래의 알고리즘이 옳다는 것은 보조정리 1과 2에 의해서 보여질 수 있다.

경우 1) $lcp_i \geq l$ 이고, $cand\_idx = i-1$.

P, $suffix(k_{cand\_idx})$, $suffix(k_i)$, lcp_i가 각각 보조정리 2의 α, β_1, β_2, l에 해당한다.

A) $lnc_i < P[lcp_i+1]$ 또는 $lnc_i = P[lcp_i+1]$인 경우:

보조정리 2-A에 의해서 $lcp(P, suffix(k_{cand_idx})) \leq lcp(P, suffix(k_i))$이므로 $cand_idx$는 i가 된다. 그리고 $mlcp$의 값은 -1이 된다. 아래 그림 4(a)는 $lnc_i = P[lcp_i+1]$인 경우이다. 그림에서 $lcp(P, suffix(k_0)) = 2$, $lcp(P, suffix(k_1)) = 4$임을 알 수 있다. 그림 4(b)는 $lnc_i < P[lcp_i+1]$인 경우의 예이다. 이 경우 $lcp(P, suffix(k_1)) = lcp(P, suffix(k_2)) = 4$임을 알 수 있다.

B) $lnc_i > P[lcp_i+1]$인 경우:

보조정리 2-B에 의해서 $lcp(P, suffix(k_{cand_idx})) \geq lcp(P, suffix(k_i))$이므로 $cand_idx$의 값은 변화 없다. 그리고 $mlcp$는 lcp_i가 된다. 그림 4(c)는 이 경우의 예로서 $lcp(P, suffix(k_2)) = lcp(P, suffix(k_3)) = 4$이고, $cand_idx$는 변하지 않고, $mlcp$는 7이 된다.

경우 2) $lcp_i \geq l$, $cand_idx < i-1$이고, $mlcp \geq lcp_i$.

P, $suffix(k_{cand\_idx})$, $suffix(k_i)$가 각각 보조정리 2의 α, β_1, β_2에 해당하고,

$$lcp(suffix(k_{cand\_idx}),\ suffix(k_i)) = \min\{\min_{cand\_idx < j < i}\{lcp_j\},\ lcp_i\} = \min\{mlcp,\ lcp_i\} = lcp_i$$

이므로 lcp_i가 보조정리 2의 l에 해당함을 알 수 있다. 즉, lnc_i가 보조정리 2의 $\beta_2[l+1]$에 해당하므로 경우 1의 결과를 똑같이 적용할 수 있다. 그림 4(d)는 $mlcp$는 7이고, lcp_i는 4이므로 이 경우에 해당하고, c>b이므로 조건 B에 해당한다. 따라서 $cand_idx$는 변하지 않고, $mlcp$는 4가 된다.

경우 3) $lcp_i \geq l$, $cand_idx < i-1$이고, $mlcp < lcp_i$.

P, $suffix(k_{cand\_idx})$, $suffix(k_i)$가 각각 보조정리 2의 α, β_1, β_2에 해당한다고 하면, $lcp(suffix(k_{cand_idx}), suffix(k_i)) = mlcp$이므로 보조정리 2의 결과를 이용하기 위해 $P[mlcp+1]$과 $suffix(k_i)[mlcp+1]$의 순서 관계를 조사한다.

P와 $suffix(k_h)$를 비교하는 h번째 iteration의 상황을 생각해보자. 여기서 h는 $lcp_h = mlcp$, $cand_idx < h < i$를 만족하는 인덱스이다. $lcp_h = mlcp$이므로 경우 1 또는 2에 해당하고, $cand_idx$의 값이 변하지 않았으므로, 조건 B에 해당했음을 의미한다. 즉, $P[mlcp+1] < suffix(k_h)[mlcp+1]$이다. 또, $lcp(suffix(k_h), suffix(k_i)) \geq mlcp$이므로, $suffix(k_h)[mlcp+1] < suffix(k_i)[mlcp+1]$이거나 $suffix(k_h)[mlcp+1] = suffix(k_i)[mlcp+1]$이다.

그러므로 $P[mlcp+1] < suffix(k_i)[mlcp+1]$이고, 보조정리 2-B에 의해서 $lcp(P, suffix(k_{cand_idx})) \geq lcp(P, suffix(k_i))$이다. 따라서 $cand_idx$의 값은 변화 없고, $mlcp \leq lcp_i$이므로 $mlcp$도 변화 없다. 그림 4(e)는 이 경우의 예이다. 이 예에서 $h=4$, $mlcp=4$이다.

경우 4) $lcp_i < l$.

$lcp(suffix(k_i), P) = lcp_i < l \leq lcp(suffix(k_{i-1}), P)$

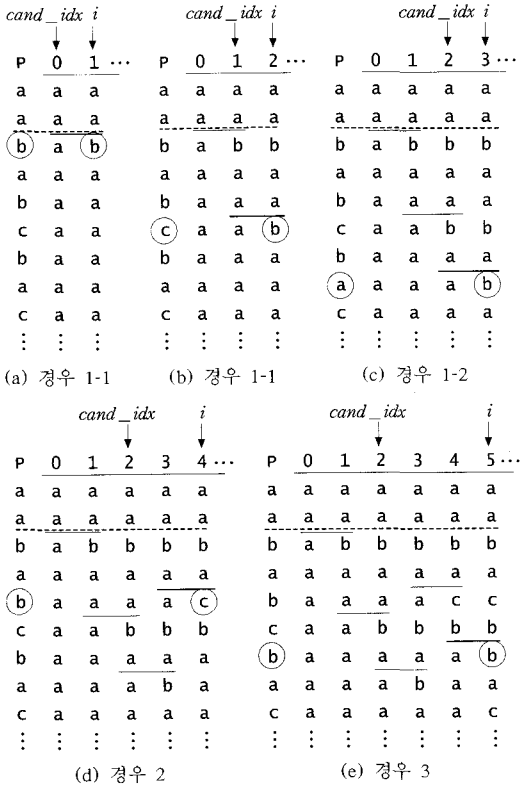

그림 4 단계 A의 예

이므로 보조정리 1에 의해 $cand\_idx$ 의 값에는 변화가 없고, $P < suffix(k_i)$ 이다. 따라서 loop를 종료하고 단계 B로 넘어간다.

경우 4의 발생은 loop의 종료를 의미하고, 경우 1, 2, 3의 결과는 다음 $i+1$번째 iteration 시작 전의 loop invariant 조건을 만족시킴을 알 수 있다.

*mode*가 *right* 인 경우:

이 경우에도 *mode*가 *left* 인 경우와 대칭적으로 노드에 저장된 가장 큰 접미사인 $suffix(k_n)$부터 $suffix(k_1)$까지 내림차순으로 접미사와 P를 비교할 수 있지만, 전체 알고리즘을 단순하게 하기 위해 위의 모듈을 그대로 사용하는 방법을 설명한다. *mode*가 *left* 인 경우 $lcp(P, suffix(k_i))$가 l보다 같거나 큰 접미사 $suffix(k_i)$ 만을 조사한다. 그리고 그 중에서 P와의 lcp가 가장 긴 접미사를 위의 알고리즘을 통해 찾는다. 마찬가지로 *mode*가 *right* 인 경우에도 $lcp(P, suffix(k_i))$가 l보다 같거나 큰 접미사 $suffix(k_i)$들을 가장 작은 접미사부터 위의 알고리즘을 사용하여 P와 비교하면 된다. 즉,

1. lcp_{n+1}부터 인덱스의 내림차순으로 노드에 저장된 lcp_i를 읽으면서 그 값이 처음으로 l보다 작은 인덱스 j를 찾는다. 이 j보다 작은 i에 대해서 $lcp(P, suffix(k_i)) < l$이고, j보다 같거나 큰 i에 대해서 $lcp(P, suffix(k_i)) \geq l$이다. 즉, $suffix(k_j)$가 노드에 저장된 접미사들 중에서 P와의 lcp가 l보다 같거나 큰 가장 작은 접미사이다.
2. $cand_idx$는 j, $mlcp$는 -1로 초기화하고, 위의 loop를 $j+1$번째 iteration부터 수행한다.

3.2.2 단계 B

단계 B에서는 단계 A에서 선택된 접미사 $suffix(k_{cand_idx})$를 P와 비교한다. 이 때 이미 둘 사이의 lcp 길이가 l보다 같거나 길다는 것을 알기 때문에 $l+1$번째 문자부터 비교하면 된다. 단계 B의 알고리즘은 다음과 같다.

1. 디스크로부터 문자 $suffix(k_{cand_idx})[l+1]$이 저장된 page를 읽어들인다.
2. 문자 $suffix(k_{cand_idx})[l+1]$와 $P[l+1]$ 부터 비교를 수행한다.
3. 중간에 mismatch가 발생하면 종료한다. 그렇지 않고 읽어들인 page의 끝 문자까지 비교가 진행되면, 다음 page를 읽어들여서 비교하는 일을 반복한다. P의 끝에 특수 문자 '#'을 붙였기 때문에 반드시 mismatch가 발생한다.

단계 B에서 우리는 P와 $suffix(k_{cand_idx})$의 크기 비교 결과와 $lcp(P, suffix(k_{cand\_idx}))$ 값을 알 수 있다. 비교 결과에 따라 *mode*와 l을 다음과 같이 재정의한다.

$mode = \{$ if $suffix(k_{cand\_idx}) < P$ $\}$

$l = lcp(P, suffix(k_{cand\_idx}))$

앞의 예에서 $suffix(k_{cand_idx})$인 $suffix(k_2)$의 비교 결과 *mode* = *left* 이고, $l = 4$이다.

앞에서 제시한 알고리즘은 $cand_idx$가 0이나 $n+1$이 아닌 일반적인 경우이다. 문자 단위로 비교를 하지 않더라도 $cand_idx$가 0인 경우에는 P와의 lcp 길이가 l이고, *mode*는 *left* 이고, $cand_idx$가 $n+1$인 경우에는 P와의 lcp 길이가 l이고, *mode*는 *right* 임을 알 수 있다.

3.2.3 단계 C

단계 C에서는 실제 분기 위치를 찾기 위해 $suffix(k_{j-1}) < P < suffix(k_j)$를 만족하는 j를 찾는다.

*mode*가 *left* 인 경우:

$suffix(k_{cand_idx}) < P$이므로, $suffix(k_{cand_idx}) < suffix(k_i)$인 접미사 $suffix(k_i)$와 P의 순서 관계를 조사한다.

(a) $lcp(suffix(k_i), suffix(k_{cand_idx})) > l$이 면, $suffix(k_i)$, $suffix(k_{cand\_idx})$, P가 각각 보조정리 1의 α, β_1, β_2의 조건에 부합하므로 $suffix(k_i) < P$이다.

(b) $lcp(suffix(k_i), suffix(k_{cand_idx})) < l$이면, P, $suffix(k_{cand\_idx})$, $suffix(k_i)$가 각각 보조정리 1의 α, β_1, β_2의 조건에 부합하므로 $P < suffix(k_i)$임을 알 수 있다.

(c) $lcp(suffix(k_i), suffix(k_{cand_idx})) = l$인 경우, 단계 A의 경우 3에서와 비슷하게, $P[l+1] < suffix(k_i)[l+1]$이라는 것을 알 수 있다. 따라서 $P < suffix(k_i)$가 성립한다.

따라서 다음을 만족하는 c_j가 찾고자 하는 분기이다. ①$j > cand_idx$이고, ②$lcp_j \leq l$이면서, ③$cand_idx$ 보다 크고 j보다 작은 모든 i에 대해서 $lcp_i > l$이다.

아래 그림 5는 앞의 예에 대한 단계 C의 결과를 나타낸다. 이 예에서 위 식을 만족하는 j는 4이고, P의 위치는 세 번째와 네 번째 접미사 사이임을 알 수 있다.

*mode*가 *right* 인 경우:

위와 비슷하게 다음을 만족하는 c_j가 찾고자 하는 분기이다. ①$j \leq cand_idx$이고, ②$lcp_j \leq l$이면서, ③

*cand_idx*와 같거나 작고 j보다 큰 모든 i에 대해서 $lcp_i > l$ 이다.

따라서 프로시저 FindBranch의 반환 값은 ($j, l, mode$) 가 된다.

P	0	1	2	3 ▼	4	5	6 …
a	a	a	a	a	a	a	a
a	a	a	a	a	a	a	b
b	a	b	b	b	b	b	b
a	a	a	a	a	a	a	b
b	a	a	a	a	c	c	a
c	a	a	b	b	b	b	a
b	a	a	a	a	a	b	a
a	a	a	a	b	a	a	a
c	a	a	a	a	a	c	a
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

그림 5 단계 C

3.3 시간 복잡도

탐색 알고리즘을 수행하는데 필요한 디스크 접근 횟수를 살펴보자. FindBranch에서 필요한 디스크 접근은 노드를 저장한 페이지를 디스크로부터 읽어들이는 것과 단계 B에서 비교를 위해 접미사를 읽어 들이는 것이다. 노드를 읽어 들이는 총 횟수는 tree의 높이이므로 $O(\log_B N)$이다. SearchPattern의 while 루프에서 i 번째 iteration의 FindBranch 반환 값 l 을 l_i라 하면 i 번째 iteration에서 접미사를 읽어들이기 위한 디스크 접근 횟수는 $\lceil (l_i - l_{i-1})/B \rceil$ 이 되고, H를 트리의 높이라 할 때 탐색 과정 전체에서 $\Sigma_{i=1}^{H} \lceil (l_i - l_{i-1})/B \rceil = O(M/B)$이다. 따라서 전체 디스크 접근 횟수는 최악의 경우에 $O(\log_B N + M/B)$이다. 이는 Patricia trie를 이용해 분기를 찾는 String B-tree에서 패턴을 탐색하기 위해 필요한 디스크 접근 횟수와 같다.

3.4 모든 발생 위치 찾기

이번 소절에서는 위의 알고리즘을 확장하여 텍스트에서 패턴이 나타나는 모든 발생 위치를 찾는 방법을 기술한다. SearchPattern 함수에서 loop를 수행하면서 P-접미사를 가장 처음 찾은 노드를 x라 하면, 다른 P-접미사들은 x를 root로 하는 subtree에 존재한다. Subtree의 한 노드 x에서 한 접미사 $suffix(k_i[x])$가 P-접미사일 때, x 안에 있는 다른 접미사 $suffix(k_j[x])$는 $lcp(\ suffix(k_i),\ suffix(k_j)\ ) \geq M$이면 P-접미사이고 그렇지 않으면 P-접미사가 아니다. 또 역도 성립한다. 따라서 노드에 저장된 lcp_i만을 이용해 노드 x의 다른 P-접미사를 찾을 수 있다.

SearchPattern 함수의 9~15번째 줄은 P-접미사를 처음 찾은 노드 x에서 다른 P-접미사를 찾고, Subtree를 탐색하는 부분이다. 노드 x의 접미사 중 $suffix(k_i[x])$부터 $suffix(k_j[x])$까지의 접미사가 P-접미사라 했을 때 탐색해야할 자식 노드들은 $c_i[x]$부터 $c_{j+1}[x]$이다.

FindAllOcc 함수는 노드 x를 root로 하는 subtree에서 모든 occurrence를 찾는 함수이다. 함수의 인자 *mode*는 $suffix(k_0[x])$가 P-접미사인지 $suffix(k_{n[x]+1}[x])$가 P-접미사인지 나타낸다. 다음 조건을 만족하는 $suffix(k_j[x])$가 P-접미사가 된다.

i) $suffix(k_0[x])$가 P-접미사인 경우: (*mode*가 *left*)

$$\min_{0 < i \leq j} \{ \mathrm{lcp}_i \} \geq \mathrm{M}$$

ii) $suffix(k_{n[x]+1}[x])$가 P-접미사인 경우: (*mode*가 *right*)

$$\min_{j < i \leq n[x]+1} \{ \mathrm{lcp}_i \} \geq \mathrm{M}$$

그리고, $suffix(k_{j-1}[x])$나 $suffix(k_j[x])$가 P-접미사이면 재귀 호출을 이용해 $c_j[x]$를 탐색한다.

모든 발생 위치를 찾기 위해 부가적으로 필요한 디스크 접근은 발생 위치의 개수에 비례한다. FindAllOcc 함수는 재귀 호출을 이용한다. 즉, 노드 x에서 하나의 자식노드를 탐색한 후에는 다른 자식을 탐색하거나 부모 노드로 돌아가기 위해서 다시 노드 x의 데이타를 필요로 한다. 따라서 메모리에 하나의 디스크 페이지만을 로드할 수 있다면, 하나의 노드를 여러 번 디스크로부터 읽어야 하므로 디스크 접근 회수가 증가한다. 하지만 디스크 페이지에 비해서 메모리는 매우 크다. 하나의 노드를 처리할 때 다시 접근해야 하는 노드들은 그 노드의 선조 노드이다. 즉, 동시에 메모리에 상주하는 디스크 페이지 개수는 많아야 트리의 높이이다. B-tree의 높이는 매우 작기 때문에 한 노드의 디스크 페이지는 자식 노드를 전부 순회할 때까지 메모리에 상주할 수 있다. 따라서 모든 발생 위치를 찾기 위해 부가적으로 필요한 디스크 액세스는 $O(occ/B)$이다. 여기서 occ는 T에서 발생하는 패턴 P의 개수이다.

정리 1. 텍스트 T의 접미사 B-tree에서 패턴 P의 모든 occurrence를 찾기 위해 필요한 디스크 접근 회수는 $O\left(\log_B N + \frac{(M+occ)}{B}\right)$이다.

3.5 생성 알고리즘

접미사 B-tree의 생성하는 방법에는 크게 3가지 정도가 있다. 첫 번째는 다른 인덱스 자료구조를 중간 결과

로 이용하는 방법이고, 두 번째는 일반적인 B-tree처럼 생성하는 방법이고, 세 번째는 string B-tree [14]의 생성 알고리즘을 그대로 사용하는 방법이다.

접미사 트리, 접미사 배열, String B-tree와 같은 인덱스 자료 구조들은 접미사들을 사전 순으로 배열할 수 있는 공통점을 가지고 있다. 이 접미사들의 순서를 이용하면, 어떤 접미사가 접미사 B-tree의 어느 위치에 저장되어야 하는 지를 알 수 있다. 따라서 B-tree의 병합(merge), 분할(split) 연산 없이 바로 접미사 B-tree를 만들 수 있다. 그러나 이 방법은 다른 자료 구조를 중간에 이용하므로 효율이 가장 뒤떨어진다.

두 번째 방법은 B-tree에서처럼 T의 접미사를 tree에 하나씩 삽입하는 방법이다. 위의 탐색 알고리즘을 이용해 접미사가 삽입되어야 할 위치를 찾은 후 필요한 데이타를 노드에 삽입한다. 이 때 B-tree에서처럼, 노드의 병합, 분할 연산이 발생할 수 있다. 하나의 접미사를 삽입하는데 $O(\log_B N+N/B)$번의 디스크 접근이 필요하므로, 접미사 B-tree의 생성 즉, T의 접미사를 모두 삽입하는데 필요한 디스크 접근은 $O(N\log_B N+N^2/B)$번이다.

String B-tree에서는 위의 B-tree처럼 접미사를 하나씩 삽입하는데, 전체 복잡도를 줄이기 위해서 추가 정보를 사용한다. 접미사 B-tree도 String B-tree와 똑같은 부가 정보를 사용하면 좀더 적은 디스크 접근을 사용해 생성할 수 있다. 다만 생성 알고리즘에서도 노드 안에서 분기를 찾는 부분이 필요한데 그 부분에서는 [14]의 분기 알고리즘 대신 위에서 제시된 분기 알고리즘을 사용한다. 분기 부분을 제외하고는 [14]의 생성 알고리즘과 동일하므로 알고리즘의 자세한 기술은 생략한다. 자세한 생성 알고리즘에 대해서는 [14]를 참조하기 바란다. 이 생성 알고리즘은 $O(N\log_B N)$번의 디스크 접근을 사용한다.

4. 결 론

본 논문에서는 기존의 String B-tree라는 외부 메모리 자료구조보다 간단하지만 효율적으로 문자열을 탐색할 수 있는 접미사 B-tree를 제안하고, 탐색 알고리즘의 핵심인 분기 알고리즘을 제시하였다. 접미사 B-tree는 노드 내부의 자료 구조에 배열을 사용하였기 때문에 trie 구조를 사용한 String B-tree보다 구현이 간단하다는 장점을 가지고 있다. 접미사 B-tree를 이용하면 문자열 알고리즘의 기본적인 연산인 문자열 탐색을 String B-tree에서와 같이 $O\left(\log_B N+\frac{(M+occ)}{B}\right)$의 외부 메모리에 접근만을 사용하여 수행할 수 있다.

참 고 문 헌

[1] D. Gusfield, Algorithms on Strings, Tree, and Sequences, Cambridge University Press, Cambridge, 1997.

[2] E. M. McCreight, "A space-economical suffix tree construction algorithms," J. ACM 23, pp. 262-272, 1976.

[3] E. Ukkonen, "On-line construction of suffix trees," Algorithmica 14, pp. 353-364, 1993.

[4] P. Weiner, "Linear pattern matching algorithms," Proceedings of the 14th IEEE symposium on Switching and Automata Theory, pp. 1-11, 1973.

[5] U. Manber, G. Myers, "Suffix arrays: a new method for on-line string searches," SIAM J. Computing 22, pp. 935-948, 1993.

[6] E. Ukkonen, D. Wood, "Approximate string matching with suffix automata," Algorithmica 10, pp. 353-364, 1993.

[7] D. R. Morrison, "PATRICIA: Practical algorithm to retrieve information coded in alphanumeric," J. ACM 15, pp. 514-534, 1968.

[8] R. W. Irving, L. Love, "The suffix binary search tree and suffix AVL tree," to appear in Journal of Discrete Algorithms.

[9] J. S. Vitter, "External memory algorithms and data structures: dealing with massive data," ACM Computing Surveys 33, pp. 209-271, 2001.

[10] N. Prywes, H. Gray, "The organization of a Multilist-type associative memory," IEEE Trans. on Communication and Electronics 68, pp. 488-492, 1963.

[11] R. Bayer, C. McCreight, "Organization and maintenance of large ordered indexes," Acta Informatica 1, 3. pp. 173-189, 1972.

[12] R. Bayer, K. Unterauer, "Prefix B-trees," ACM Trans. Database System 2, 1, pp. 11-26, 1977.

[13] D. Comer, "The ubiquitous B-trees," Computing Surveys 11, pp. 121-137, 1979.

[14] P. Ferragina, R. Grossi, "The string B-tree: a new data structure for string search in external memory and its applications," J. ACM 46(2), pp. 236-280, 1999.

[15] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," 12th Symposium on Combinatorial Pattern Matching, pp. 181-192, 2001.

나 중 채
1998년 서울대학교 컴퓨터공학과 학사
2000년 서울대학교 컴퓨터공학과 석사
2000년~현재 서울대학교 컴퓨터공학부 박사과정. 관심분야는 컴퓨터이론, 알고리즘, 생물정보학

박 근 수
정보과학회논문지 : 시스템 및 이론
제 32 권 제 5 호 참조