JKIICE Journal of the Korea Institute of Information and Communication Engineering

한국정보통신학회논문지(J. Korea Inst. Inf. Commun. Eng.) Vol. 19, No. 2 : 365~371 Feb. 2015

검색 성능 향상과 동적 환경을 위한 HCB 트리의 개선

김성완*

Enhancement of HCB Tree for Improving Retrieval Performance and Dynamic Environments

Sung Wan Kim*

Division of Computer, Sahmyook University, Seoul 139-800, Korea

요 약

이진 트라이를 이진 비트열로 압축하여 표현하는 CB 트리는 키가 늘어남에 따라 이진 비트열이 길어지게 되어 검색 시간이 증가하며 잦은 키 삽입/삭제 연산에 비효율적이다. 작은 분할 트라이들을 계층적 구조로 표현한 HCB 트리가 제안되었으나 비트열 시프트 처리를 근본적으로 해결할 수 없으며 자식 혹은 부모 트리 참조를 위해 별도의 자료구조를 탐색해야 하는 부담이 있다. 본 논문에서는 각 분할 트리를 포화 이진 트라이 형태로 표현하고 레벨 순위에 따라 분할 트리 번호를 할당하여 검색 성능을 향상 시키는 한편 키의 삽입/삭제 시에 시프트 연산이 발생하지 않도록 하였다. 시·공간 복잡도를 사용한 성능 평가에서 검색 시에는 제안 방법과 HCB 트리 방법이 CB 트리에 비해 우수한 것으로 나타났으며, 키 삽입/삭제는 제안 방법이 가장 높은 성능을 보여주었다. 공간 사용량은 제안 방법이 CB 트리 방법에 비해 71～89%의 공간만을 요구하여 가장 좋은 성능을 보였다.

ABSTRACT

CB tree represents the binary trie by a compact binary sequence. However, retrieval time grows fast since the more keys stored in the trie, longer the binary sequences are. In addition it is inefficient for frequent key insertion/deletion. HCB tree is a hierarchical CB tree consisting of small binary tries. However it can not avoid shift operations and have to scan an additional table to refer child or parent trie. In order to improve retrieval performance and avoid shift operations when keys are inserted or deleted, we in this paper represent each separated trie by a full binary trie and then assign the unique identifier to it. Finally the theoretical evaluations show that both the proposed approach and HCB tree provides better than CB tree for key retrieval. The proposed approach shows the highest performance in case of key insertion/deletion and moreover requires only 71%~89% of storage as compared with CB tree.

키워드 : 이진 트라이, CB 트리, HCB 트리, 정보 검색, 인덱스

Key word : Binary Trie, Compact Binary Tree, Hierarchical CB tree, Information Retrieval, Index

접수일자 : 2014. 12. 12 심사완료일자 : 2015. 01. 05 게재확정일자 : 2015. 01. 19

* **Corresponding Author** Sung Wan Kim(E-mail:swkim@syu.ac.kr, Tel:+82-2-3399-1781)
Division of Computer, Sahmyook University, Seoul 139-800, Korea

Open Access http://dx.doi.org/10.6109/jkiice.2015.19.2.365 print ISSN: 2234-4772 online ISSN: 2288-4165

Ⅰ. 서 론

신속한 정보 검색을 위한 인덱스 구조는 현대 정보 검색 시스템의 핵심 요소이다[1]. 특히, 문자열을 키로 하는 탐색을 위한 인덱스 구조로 트라이 (trie)가 널리 사용되고 있다. 차수가 2인 이진 트라이 (binary trie)는 키를 이진 비트 단위로 분해하여 빠른 탐색에 활용한다. 그러나 이진 트라이를 인덱스 구조로 직접 구현할 경우 키 집합이 커지면 메모리에 유지할 수 없다.

이러한 문제점을 해결하기 위해 이진 트라이를 하나의 이진 비트열 형태로 압축하여 표현한 CB 트리가 제안되었으나 이진 비트열의 길이가 길어질 경우 키 검색 시간이 증가하게 되며, 키의 삽입 또는 삭제 시 이진 비트열에 대한 시프트 처리가 요구된다. 이를 완화하고자 작은 크기의 이진 트리들을 계층적 형태로 구성한 HCB 트리가 제안되었으나 시프트 처리 문제를 근본적으로 해결할 수 없으며 자식 혹은 부모 트리 참조를 위해 별도의 자료 구조를 탐색해야 한다.

본 논문에서는 HCB 트리의 검색 성능을 향상시키고 키의 잦은 삭제 및 삽입이 발생하는 동적인 환경에 효과적으로 대처하기 위한 개선 방안들을 제안하였다. 제안 방법들은 비트열에 대한 시프트 연산이 필요하지 않으며 자식 혹은 부모 트리를 참조 시 수식을 이용하여 상수 시간 내에 처리되도록 하였다. 2장에서는 CB 트리와 HCB 트리의 특징에 대해 설명하고 3장에서는 HCB 트리를 개선하기 위한 방안들을 제안한다. 4장에서는 제안 방법의 시·공간적 우수성을 평가하며 5장에서 결론을 맺는다.

Ⅱ. 관련연구

기수 탐색 트리 혹은 트라이는 가변 길이의 문자열을 키로 하는 검색에 가장 효율적인 방법이다[2, 3]. 키는 리프 노드에 유지되므로 키 비교는 리프 노드에서 1회만 수행된다. 이진 트라이는 차수가 2인 트라이로 키를 비트 단위로 분해하여 탐색 트리를 구성하며 각 키는 이진 시퀀스 형태로 표현된다. 이를 위해 키를 구성하는 각 문자는 이진 코드로 매핑 된다. 예를 들어 문자 'a'는 5비트의 이진 코드 '00000'으로 표현할 수 있으며, 키 'air'는 3개의 이진 코드를 합한 '00000 01000 10001' 형태의 이진 시퀀스로 변환 된다. <표 1>은 키 집합 K = {air, big, tea, try zoo}에 대한 이진 시퀀스를 나타낸 것이다.

표 1. 키 집합 K에 대한 이진 시퀀스
Table. 1 Binary Sequences for Ket Set K

키 값	이진 시퀀스
air	00000 01000 10001
big	00001 01000 00110
tea	10011 00100 00000
try	10011 10001 11000
zoo	11001 01110 01110

<그림 1>은 위 이진 시퀀스들을 높이가 5인 이진 트라이 구조에 모두 삽입한 것이다. 내부 노드는 분기를 위해 사용되며 리프 노드는 키가 저장된 버켓을 참조한다. 이진 트라이를 효과적으로 압축하는 한편 내부 노드의 개수 보다 리프 노드의 개수가 하나 더 많게 되는 이진 트리의 특성을 만족하게 하기 위해 버켓을 참조하지 않는 더미 리프를 추가한다.

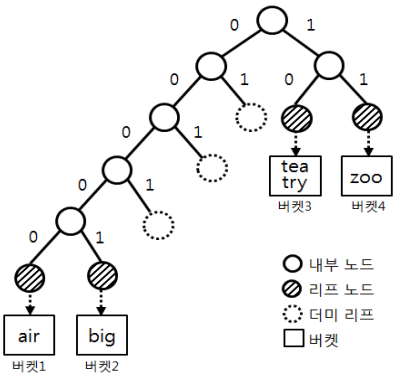

그림 1. 키 집합 K에 대한 이진 트라이
Fig. 1 Binary Trie for Key Set K

이진 트라이를 직접 구현 할 경우 키 집합이 커지면 저장 공간이 증가하여 메모리에 유지하기 어렵게 된다. 이러한 문제점을 해결 하고자 제안된 CB (compact binary) 트리는 이진 트라이를 밀집된 이진 비트 시퀀스 형태로 표현한다[4]. CB 트리는 treeMap, leafMap, bList로 구성 된다. treeMap은 이진 트라이의 각 노드를 전위 순회하여 얻게 되는 이진 비트 시퀀스이며, 내부 노드와 리프 노드를 방문할 때 각각 '0'과 '1'로 방문 순

서를 표현한다. leafMap은 리프 노드들만을 전위 순회하여 얻은 이진 비트 시퀀스로, 더미 리프는 '0'으로 일반 리프 노드는 '1'로 방문 순서를 표현한다. bList는 실제 키를 저장하는 버켓 주소를 유지하는 리스트이다. <그림 1>의 이진 트라이를 CB 트리로 표현한 것이 <그림 2> 이다. treeMap과 leafMap은 각각 13 비트와 7 비트를 사용하므로 이진 트리를 직접 구현할 때 보다 매우 압축된 형태로 표현이 가능하다.

treeMap	0	0	0	0	0	1	1	1	1	1	0	1	1
leafMap	1	1	0	0	0	1	1						

bList		
	1	버켓 1의 주소
	2	버켓 2의 주소
	3	버켓 3의 주소
	4	버켓 4의 주소

그림 2. 그림 1.에 대한 CB 트리
Fig. 2 CB Tree for Fig. 1

CB 트리를 사용한 키 탐색은 키에 대한 이진 시퀀스와 treeMap의 이진 시퀀스를 좌측에서 우측 방향으로 상호 비트별로 비교하면서 진행된다. 다만 키에 대한 이진 시퀀스의 특정 비트가 '1'일 경우 해당 비트를 루트로 하는 좌측 서브 트리의 탐색을 생략하고 우측 서브 트리를 탐색해야 한다. 예를 들어, <그림 1>에서 키가 '1'로 시작한다면 루트의 좌측 서브 트리는 탐색 범위에서 생략되어야 한다. 이 생략 과정은 이진 트라이의 특성을 이용하면 되는데 treeMap에서 '1' 비트들의 수가 '0' 비트들의 수보다 하나 클 때 까지 비트 탐색 위치를 우측으로 이동하여 처리하면 된다.

CB 트리에서는 키 집합이 커질수록 이진 시퀀스의 길이가 더 길어진다. 이진 시퀀스의 우측 끝부분에 위치한 키들을 찾을 경우 이진 시퀀스 대부분을 탐색해야 하므로 많은 검색 시간이 소요된다. 또한, 키 삽입/삭제의 경우 이진 시퀀스에 대한 시프트 연산이 요구되어 동적인 환경에서 부담이 크다. 예를 들어, <그림 1>의 버켓 1이 오버플로우 될 경우 두 개의 리프 노드를 자식으로 추가하고 버켓을 분할해야 하므로 삽입 위치를 기준으로 모든 오른쪽 시퀀스들의 이동이 요구된다. 또한, <그림 1>에서 키 'air'와 'big'이 연속해서 삭제될 경우 두 버켓이 공백이 되어 부모 노드와의 합병이 필요하며 연쇄적으로 루트의 좌측 노드까지 합병이 수행되므로 여러 번의 시프트 연산이 발생한다.

[5]에서는 CB 트리의 단점을 극복하기 위해 단일 이진 트라이를 <그림 3>과 같이 특정 깊이를 갖는 여러 개의 작은 이진 트라이들로 분할한 후 서로 연결된 계층적 구조를 갖도록 변형 하였다. 여기서, 분할된 각 이진 트라이를 '분할 트리'라 하며, 분할 트리를 생성하기 위해 사용된 깊이를 '분할 깊이'라 한다. <그림 3>에서 분할 깊이는 2이다. 또한, 각 분할 트리 식별을 위해 분할 트리 번호가 할당된다. 이러한 계층적 이진 트라이 구조를 이진 시퀀스 형태로 표현한 것을 계층적 CB 트리 (HCB)라 하며, 분할 트리 개수만큼의 treeMap, leafMap, bList를 각각 사용하여 표현된다. <그림 4>는 <그림 3>에 대한 HCB 표현을 나타낸 것이다.

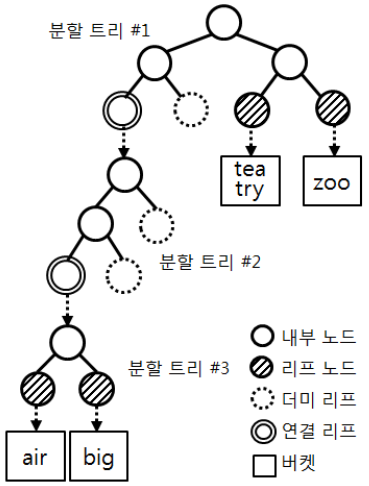

그림 3. 그림 1.에 대한 계층적 이진 트라이
Fig. 3 Hierarchical Binary Trie for Fig. 1

$treeMap_1$	0	0	1	1	0	1	1
$treeMap_2$	0	0	1	1	1		
$treeMap_3$	0	1	1				
$leafMap_1$	1	0	1	1			
$leafMap_2$	1	0	0				
$leafMap_3$	1	1					

$bList_1$	1	-2
	2	버켓 3의 주소
	3	버켓 4의 주소
$bList_2$	1	-3
$bList_3$	1	버켓 1의 주소
	2	버켓 2의 주소

그림 4. 그림 3.에 대한 HCB 트리
Fig. 4 HCB Tree for Fig. 3

한편, 자식 분할 트리를 포인트 하는 리프 노드를 '연결 리프'라 한다. 연결 리프를 위한 bList의 해당 엔트리에는 음수 형태의 자식 분할 트리 번호를 저장하며 하위 분할 트리를 연속하여 탐색할 때 활용된다.

<그림 3>에서 최우측에 위치한 키 'zoo'를 탐색할 경우 최상위 분할 트리만 탐색 범위가 되므로 CB 트리에 비해 검색 부담이 축소되는 장점이 있으나 여전히 다음과 같은 문제점을 포함하고 있다. 우선, 키 삽입 혹은 삭제로 인한 시프트 연산의 범위가 특정 분할 트리에 국한되지만 근본적으로 시프트 연산을 피할 수는 없다. 예를 들어 <그림 3>에서 첫 번째 버켓이 오버플로우 될 경우 CB 트리와 마찬가지로 시프트 연산이 요구된다.

또한, 부모 혹은 자식 분할 트리 참조 시 bList들을 스캐닝해야 한다. 이를 세 가지 경우로 나누어 살펴보면 첫째, 삭제 연산으로 인해 더미 리프만을 포함하게 된 분할 트리는 삭제되어야 하므로 이를 링크하고 있는 연결 리프와 합병되어야 한다. 즉, 부모 분할 트리의 연결 리프는 더미 리프로 수정되어야 한다. 이를 위해서는 부모 분할 트리 번호 i를 알아야 해당 $leafMap_i$을 접근하여 연결 리프에 대한 비트를 1 → 0으로 수정할 수 있다. 그러나 기존의 HCB 방법에서는 부모 분할 트리 번호를 알기 위해서는 모든 bList들을 스캐닝하여 삭제될 분할 트리 번호를 음수 형태로 가지고 있는 $bList_i$의 i값을 부모 분할 트리 번호로 취해야 한다.

둘째, 리프 노드에 연결된 버켓이 오버플로우 될 경우 새로운 자식 분할 트리를 생성해야 하며 해당 bList의 엔트리에 저장된 버켓 주소를 자식 분할 트리 번호로 변경해야 한다. 셋째, 키 검색 연산에서 자식 분할 트리 접근을 위해서는 해당 bList를 스캐닝 하여 음수 형태로 저장된 자식 분할 트리 번호를 추출해야 한다.

Ⅲ. 제안 표현 방법

본 장에서는 앞 장에서 설명한 HCB 트리의 문제점을 해결하기 위한 방안들을 제안한다. 첫째, <그림 5>와 같이 각 분할 트리를 분할 깊이가 2인 포화 이진 트라이(full binray trie) 구조로 표현한다. [6]의 논문에서는 키 집합의 모든 키를 등록 또는 검색 시 분할 깊이가 2인 경우가 가장 최적인 것으로 평가되었다. 포화 이진 트라이 구조로 표현한 이유는 분할 트리의 깊이가 낮을 경우 많은 키가 삽입 될수록 리프 레벨을 제외한 각 분할 트리는 대부분 포화 상태가 될 것이 예상되기 때문이다. 또한, 분할 깊이가 2인 포화 이진 트라이에 대한 treeMap의 이진 시퀀스는 '0011011'로 모두 동일하므로 하나의 treeMap만 유지해도 된다.

또한, 키 삽입에 대해서 시프트 연산이 필요하지 않게 된다. 즉, treeMap과 leafMap은 포화 이진 트라이에 포함된 모든 노드에 대한 비트 필드를 이미 포함하고 있으며 삽입 위치도 분할 트리 깊이에 해당하는 리프 레벨에서 진행되므로 비트 시퀀스의 해당 필드 값만을 수정하면 된다. 키 삭제의 경우에도 비트 시퀀스의 해당 필드 값만을 수정하면 된다. 예를 들어 <그림 5>에서 키 'air'가 삭제되더라도 해당 버킷을 참조하는 리프 노드를 더미 리프로 즉, 1 → 0으로 수정하기만 하면 된다.

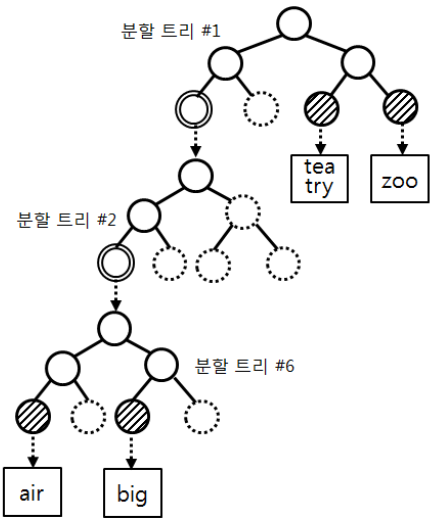

그림 5. 포화 이진 트라이 표현
Fig. 5 Full Binary Trie Representation

둘째, 좌측 서브 트리에 대한 이진 시퀀스 탐색을 생략하기 위해서 기존의 방법 처럼 treeMap 의 '1'과 '0' 비트들의 개수를 세지 않고 포화 이진 트리의 특성을 이용하여 다음 (1)번 수식을 통해 상수 시간에 처리하도록 한다. 여기서 d는 분할 트리의 높이를 나타내며 c는 현재 탐색 중인 비트가 분할 트리 상에서 위치한 레벨을 의미한다. t는 treeMap 이진 시퀀스에서 현재 탐색 중인 비트의 위치이다.

좌측 서브 트리의 끝 노드 위치 $2^{d-c} + t - 1$ (1)

n의 부모 분할 트리 번호 $\lfloor (n-2)/k + 1 \rfloor$ (2)

n의 i번째 자식 분할 트리 번호 $k(n-1) + i + 1$ (3)

셋째, 부모/자식 분할 트리 접근 시 모든 bList들에 대한 스캐닝 작업을 피하기 위해 분할 트리 번호를 레벨 순회 순서에 따라 할당하고, 위의 수식 (2)와 (3)을 이용하여 부모 및 자식 분할 트리 번호를 상수 시간에 각각 구할 수 있도록 한다. 여기서 n은 현재 분할 트리 번호를 의미한다. 각 분할 트리를 단일 노드로 간주하면 d=2일 경우 각 분할 트리는 최대 4개의 자식 분할 트리를 갖게 되므로 트리의 최대 차수 k는 4가 되며 자식 분할 트리의 개수 i의 범위는 $1 \le i \le 4$ 가 된다. <그림 5>에서 세 번째 분할 트리 번호는 6이 된다.

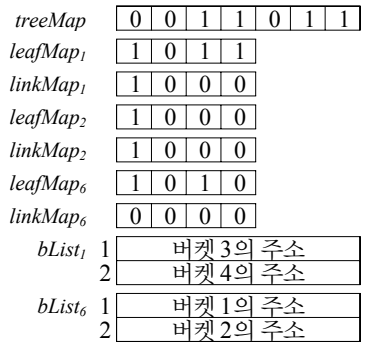

그림 6. 그림 5.에 대한 제안 트리
Fig. 6 The Proposed Tree for Fig. 5

키 검색 시 이진 시퀀스의 비트들을 카운트하여 bList로부터 자식 분할 트리 번호를 얻어내는 대신 (3)번 수식을 통해 구해낼 수 있다. 또한, 더미 노드만 포함하는 분할 트리를 삭제 시 이를 링크하는 부모 분할 트리의 해당 연결 리프와 합병하기 위해 얻어야 하는 부모 분할 트리 번호도 (2)번 수식 계산을 통해 빠르게 구해낼 수 있다.

넷째, 리프들은 더미 리프, 버켓을 참조하는 리프, 또는 자식 분할 트리를 참조하는 연결 리프 등 3가지가 존재한다. leafMap만으로는 3 종류의 리프 노드를 구별할 수 없다. 따라서, 본 논문에서는 leafMap과 동일한 크기를 가지는 linkMap을 추가하여 연결 리프를 식별할 수 있도록 하였다. linkMap에서 비트 '1'은 버켓을 참조하는 리프를, '0'은 더미 또는 연결 리프를 각각 나타낸다. 즉, 3가지 종류의 리프를 판별하기 위해서는 (leafMap[i], linkMap[i]) 쌍 값을 참조하여 (0, 0)인 경우는 더미 리프로, (1, 0)인 경우는 연결 리프로, (1, 1)인 경우는 버켓을 참조하는 리프로 판별하게 된다.

```
[1단계] 초기화
  k ← 1, n ← 1, t ← 1, i ← 1, c ← 1

[2단계] key[ ]의 각 비트 검증
  if key[k] == '1' then goto [3단계]
  else goto [4단계]

[3단계] 좌측 서브 트리 탐색 생략
  t ← 2^(d-c) + t - 1
  goto [4단계]

[4단계] 트리의 간선 탐색
  if treeMap_n[++t] == '0' then
     k ← k + 1, c ← c + 1
     goto [2단계]
  else
     goto [5단계]

[5단계] 리프 노드 유형 판별
  if t == 3 then i ← 1
  else if t == 4 then i ← 2
  else if t == 6 then i ← 3
  else  i ← 4  // if t == 7

  if leafMap_n[i] == '1' ∧ linkMap_n[i] == '1' then
     goto [6단계]
  else if leafMap_n[i] == '1' ∧ linkMap_n[i] == '0' then
     n ← 4(n-1) + i + 1 // 자식 분할 트리 번호
     k ← k + 1, t ← 1, c ← 1
     goto [2단계]
  else   // if leafMap_n[i] == '0'
     키 탐색 실패

[6단계] 해당 bList_n에서 버켓 주소 추출
  j ← linkMap_n[1 . . i]에서 '1' 비트의 개수
  addr ← bList_n[j]
  goto [7단계]

[7단계] 해당 버켓에서 키 포함여부 확인
  addr에 해당하는 버켓에서 키 포함 여부 확인
```

그림 7. 키 검색 알고리즘
Fig. 7 Key Retrieval Algorithm

<그림 6>은 <그림 5>에 대해 제안 방법을 이용하여 나타낸 개선된 HCB 트리 표현이다. treeMap은 하나만 유지하며, leafMap과 연계되어 활용되는 linkMap이 추

가 되었다. 또한, bList에는 자식 분할 트리 번호가 포함되지 않는다. <그림 7>은 제안 트리 구조를 이용한 키 탐색 알고리즘을 나타낸 것이다. 여기서 key는 키에 대한 이진 시퀀스를 나타내며 k, t, i는 key, treeMap, 그리고 leafMap과 linkMap에서의 현재 비교 위치를 각각 나타낸다. n은 분할 트리 번호를 의미하며 c는 현재 탐색 중인 분할 트리에서 접근 중인 노드의 레벨 값이다.

Ⅳ. 평 가

본 장에서는 [5]의 평가 방법을 참조하여 CB 트리, HCB 트리, 그리고 제안 방법을 비교 평가한다. 최악의 시·공간적 복잡도를 비교하기 위해 각 트라이는 포화 이진 트라이로 가정한다. n과 m은 이진 트라이의 높이와 분할 깊이를 각각 나타낸다. α는 ⌈n/m⌉으로 HCB 트리와 제안 방법 트리에서 레이어의 개수를 의미하며 예를 들어 <그림 1>에서 n은 5이며, <그림 5>에서 m과 α는 2와 3이 된다.

4.1. 시간 복잡도 평가

시간 복잡도 평가는 이진 비트열을 탐색하는 시간만을 고려하자. CB 트리의 최악의 검색 시간 복잡도는 이진 트라이 전체를 탐색해야 하므로 전체 노드 수에 해당하는 $O(2^n)$이 된다. HCB 트리와 제안 방법은 각 레이어당 한 개의 분할 트리씩 탐색하면 되므로 레이어 개수와 한 분할 트리의 전체 노드 수의 곱인 $O(\alpha 2^m)$이 되어 CB 트리 방법에 비해 우수한 검색 성능을 나타낸다.

삽입/삭제를 위한 시간 복잡도는 해당 위치까지 검색하는 시간도 포함되어야 하지만 시프트 연산이 필요한 비트열 길이만을 계산하자. 최악의 경우라도 트라이의 최 좌측 간선을 따라 위치한 노드들에 해당하는 비트들(이진 비트열의 최 좌측에서 '1'이 나오기 전까지 n개의 '0'비트들)은 시프트에서 제외되므로 CB 트리에서 복잡도는 $O(2^n-n)$이 된다. HCB 트리의 복잡도는 연산 범위가 특정 분할 트리 하나에만 국한되므로 $O(2^m-m)$이 되어 CB 트리 방법에 비해 우수하다. 제안 방법의 복잡도는 시프트 연산 없이 특정 비트 필드 값 하나만 수정하면 되므로 O(1)이 되어 가장 우수한 성능을 보이게 된다.

4.2. 공간 복잡도 평가

CB 트리에서 treeMap을 위한 비트 수는 포화 이진 트라이의 노드 수와 동일하므로 2^{n+1} - 1 비트가 된다. HCB에서 treeMap의 비트 수는 한 분할 트리의 모든 노드 개수와 분할 트리 수의 곱이 되므로 다음 수식과 같이 구할 수 있다.

$$\sum_{k=0}^{m} 2^k \times \sum_{k=1}^{\alpha} 2^{m(k-1)} = (2^{m+1} - 1) \times \frac{2^{m\alpha} - 1}{2^m - 1}$$

$$= \{2(2^m - 1) + 1\} \times \frac{2^{m\alpha} - 1}{2^m - 1} = (2^{n+1} - 1) + \frac{2^n - 1}{2^m - 1} - 1$$

제안 방법에서 treeMap의 비트 수는 분할 트리 하나의 모든 노드 수와 같으므로 2^{m+1} - 1 비트가 된다.

CB 트리에서 leafMap은 리프 노드 수와 같으므로 2^n 비트가 필요하다. HCB에서 leafMap을 위한 비트 수는 분할 트리의 리프 노드 수와 분할 트리 수의 곱이 되므로 다음 수식과 같이 구할 수 있다.

$$2^m \times \sum_{k=1}^{\alpha} 2^{m(k-1)} = 2^m \times \frac{2^{m\alpha} - 1}{2^m - 1}$$

$$= (2^m - 1 + 1) \times \frac{2^{m\alpha} - 1}{2^m - 1} = 2^n + \frac{2^n - 1}{2^m - 1} - 1$$

제안 방법의 leafMap 비트 수는 HCB에서와 같으며, linkMap 비트 수는 leafMap 비트 수와 같다. HCB의 treeMap과 leafMap 공간은 CB 트리의 공간과 비교하여 모두 $((2^n - 1)/(2^m-1)) -1$ 비트만큼 증가한다. 제안 방법의 treeMap공간은 $2^{m+1} - 1$ 이 되어 두 방법에 비해 공간 사용양이 현저하게 줄어든다. 제안 방법의 leafMap과 linkMap 공간은 CB 트리에 비해 $\{((2^n - 1)/(2^m-1)) -1\} \times 2$ 만큼의 비트가 소요된다.

<그림 8>은 위의 공간 복잡도 수식을 이용하여 m이 2일 때 n을 19, 22, 25로 증가시키면서 계산된 공간 사용량의 결과이다. y축은 로그 스케일로 나타내었다. 제안 방법의 treeMap 공간 사용량은 7 비트여서 그래프상에 표기되지 않았다. 제안 방법의 leafMap 공간 사용량은 linkMap 사용량을 합산한 결과이며, leafMap 공간 사용량만 보면 제안 방법이 가장 크지만 전체 공간량을 측정하면 HCB 트리 방법은 CB 트리 방법에 비해 약

122%의 공간을 필요로 하며 제안 방법은 CB 트리 방법에 비해 약 89%의 공간만을 필요로 한다. m이 4일 경우에도 동일한 방법으로 공간 사용량을 계산한 결과 <그림 8>과 유사한 결과가 측정되었으며 CB 트리 방법에 비해 HCB 트리는 104%, 제안 방법은 약 71%의 공간만을 사용하는 것으로 측정되었다.

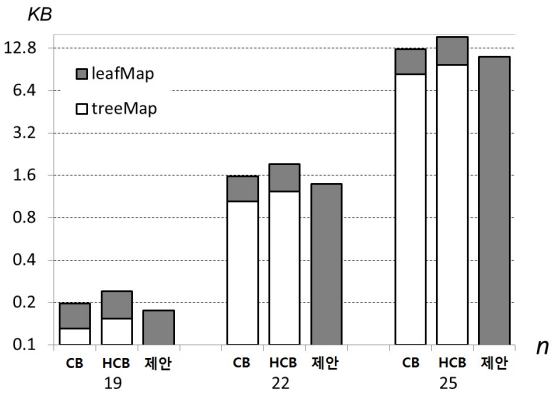

그림 8. 공간 사용량
Fig. 8 Space Usage

V. 결 론

본 논문에서는 이진 트라이를 압축된 이진 비트 시퀀스 형태로 표현하는 CB 트리와 HCB 트리의 검색 성능 향상 및 동적 환경에서의 문제점을 해결하기 위한 방안들을 제안하였다. 제안된 방안들을 적용할 경우 첫째, 키 검색에 있어 좌측 서브 트리의 탐색을 생략하거나 자식 분할 트리 번호를 구할 때 수식 계산을 통해 효과적으로 처리될 수 있다. 둘째, 키의 삽입/삭제 대해서도 이진 비트열의 시프트 연산이 필요치 않게 된다. 또한, 더미 노드만 포함하는 분할 트리를 삭제 시 접근해야 하는 부모 분할 트리 번호도 수식 계산을 통해 빠르게 구해낼 수 있다.

마지막으로 최악의 시·공간 복잡도 계산을 통해 성능을 평가하였다. 키 검색은 제안 방법과 HCB 트리가 CB 트리보다 우수하였으며 키 삽입/삭제는 제안 방법이 가장 우수하였다. 공간 사용량은 거의 leafMap과 linkMap 공간만이 필요 되는 제안 방법이 가장 우수하였고 HCB 방법이 제일 낮게 평가되었다. 향후 실험적 방법을 통해 제안 방법의 성능을 측정하고자 한다.

REFERENCES

[1] Ricardo B. and Berthier R., *Modern Information Retrieval,* 2nd ed., London:Addison Wesly, 2011.

[2] Wooseok Joo, *Data Structure in* C·C++, Seoul: Hanbit Media, Inc. 2004.

[3] Seokho Lee, *Data Structure in C*, Seoul: Kyobo Books, 2008.

[4] Jonge, D., et. al., “Two access methods using compact binary trees”, *IEEE Trans. Software Eng.*, vol. 13, no. 7, pp. 799-810, 1987.

[5] Masami Shishibori, et. al, “Two improved access methods on compact binary (CB) trees”, *Information Processing & Management*, vol. 36, no. 3, 2000.

[6] Minsoo Jung, et. al, “A Dynamic Construction Algorithm for the Compact Patricia Trie Using the Hierarchical Structure”, *Information Processing & Management*, vol. 38, no. 2, 2002.

김성완(Sung Wan Kim)

2003년 : 홍익대학교 전자계산학과 이학박사
1999년 ~ 2005년 : 삼육의명대학 컴퓨터정보과 조교수
2006년 ~ 현재 : 삼육대학교 컴퓨터학부 교수
※관심분야 : 웹데이터베이스, 정보검색, 모바일컴퓨팅