공학석사 학위논문

# RFID 태그 추적을 위한 캐시와 메인 메모리 기반의 색인기법




2007년 8월

부경대학교 산업대학원

전 산 정 보 학 과

홍 진 숙

공학석사 학위논문

# RFID 태그 추적을 위한 캐시와 메인 메모리 기반의 색인기법

지도교수 윤 성 대

이 論文을 工學碩士 學位論文으로 提出함

2007년 8월

부경대학교 산업대학원

전 산 정 보 학 과

홍 진 숙

이 논문을 홍진숙의 공학석사
학위논문으로 인준함

2007년 8월 30일




주 심 이학박사 박 홍 복 (인)

위 원 공학박사 김 창 수 (인)

위 원 이학박사 윤 성 대 (인)

# <차례>

# <그림 차례>

Indexing Scheme based on the Cache & Main Memory
for RFID tag Tracing

Jin-Suk Hong

Department of Computer and Information
Graduate School of Industry
Pukyong National University

## Abstract

Over about the last 10 years, the speed of CPU has rapidly increased than the speed of the memory. As a result, main-memory access is bottlenecked by many other computer applications, including database systems. To reduce memory access latency, cache memory is incorporated in the memory subsystem, but cache memories can reduce the memory latency only when the requested data is found in the cache. This mainly depends on the memory access pattern of the application.

In main memory-based indexing techniques, Tmr-tree compared to R-tree has a defect takes too long to input. We proposed a CSTmr-tree(Cache Sensitive Tmr-tree) structure which we apply L2 cache this is an advantage to the existing Tmr-tree. We also applied the algorithm to like insert and delete at this structure. By simulation, we have known that the time of CSTmr-tree is faster at about 1.8 ~ 3 times more than the normal time of Tmr-tree.

# Ⅰ. 서론

유비쿼터스 컴퓨팅, 와이브로, 텔레매틱스, RFID 등의 이동체 데이터베이스들이 실시간 데이터 처리에 기반하고 있어 주기억 데이터베이스에 대한 관심과 수요는 폭발적으로 증가 할 것으로 전망된다. 최근 메모리 가격의 하락과 본격적인 64비트 운영체제로 인해 대용량의 램을 사용하는 주기억 데이터베이스 시스템의 구축이 실현 가능하게 되었다.

주기억 데이터베이스 접근 방법으로는 디스크 기반 색인 기법과 주기억 장치 색인 기법으로 제안 되어져 왔다. 기존의 디스크 기반 색인기법은 디스크 접근 시간만을 고려하기 때문에, 주기억 색인기법으로 디스크 기반 색인 기법을 직접적으로 적용시키기에는 어려움이 있다. 주기억 장치 색인 기법은 모든 색인 노드들이 주기억 장치에 상주하기 때문에 노드에 대한 접근 시간이 디스크 기반 기법에 비해 상당히 미미하다[2].

주기억 데이터베이스 시스템에서는 처리속도와 주기억 장치의 효율적인 사용이 성능에 중요한 요소로 제기된다. 이러한 주기억 데이터베이스 시스템을 위한 색인 기법으로는 AVL-트리, B-트리, T-트리, T*-트리 등이 제안되었으나[5][12][15], 기존의 주기억 데이터베이스 색인 기법은 정수, 실수, 문자 등과 같은 1차원 데이터이고 2차원 이상의 데이터를 포함하는 공간 데이터에는 적용이 불가능하다.

공간 데이터는 2차원 이상의 데이터를 포함하는 데이터로 가장 대표적인 R-트리색인 기법이 있고, 변형모델인 R*-트리, R+ -트리 등 색인기법들이 제안되었다. 그러나 가장 많이 쓰이는 R-트리 계열의 색인 구조 또한 디스크를 기반으로 한 데이터베이스 시스템으로 제안되었기 때문에, 디스크 접근의 최소화에 맞춰서 주기억 데이터베이스에서 처리속도 최적화와 효율적

인 메모리사용에 적합하지 못하다. 그러므로 주기억 데이터베이스에서 공간 데이터의 효율적인 색인 구조에 대한 연구가 요구된다[4][11][12].

그리고 지난 10년간 CPU의 속도는 메모리의 속도에 비해 급속한 속도로 발전하였다. 그래서 데이터베이스 시스템을 포함한 다른 컴퓨터 응용분야에서 메모리의 접근이 병목현상을 일으키게 되었다. 메모리의 접근 속도를 줄이기 위해 캐시 메모리가 도입되었다. 캐시 메모리는 원하는 데이터가 캐시에 옮겨져 있어야 메모리 접근 속도를 줄일 수 있다. 따라서 응용프로그램에서 데이터를 어떤 순서로 액세스 하느냐에 의해 캐시의 활용도가 달라지고 응용프로그램의 성능이 달라지게 된다[1].

CPU가 데이터를 처리하기 위해서 메모리에서 연산자와 데이터를 가져와야 한다. CPU의 클럭 스피드는 급격하게 빨라졌으나 반면 메인 메모리의 속도는 CPU만큼 빨라지지 않은 것이다. 때문에 메모리의 접근으로 인해 10배 정도 CPU 성능이 저하되는 병목 현상이 발생한다. 과거에는 느린 디스크 I/O 속도를 고려하여 자료구조와 알고리즘을 설계했듯이 최근의 컴퓨터에서 최적 성능을 내기 위해서는 메모리 액세스 속도가 느림을 고려하여 자료구조와 알고리즘을 설계하여야 한다.

느린 메모리 액세스 속도를 극복하기 위해서는 CPU와 메모리 사이의 속도 격차를 극복하기 위해 만들어진 캐시메모리도 성능을 고려하여 L1 캐시, L2 캐시 등과 같이 여러 계층으로 설계하고 있다. L2 캐시를 도입하게 되면 캐시가 없는 경우보다 약 2배 정도의 성능 향상을 가져올 수 있다. 캐시 메모리를 채택한 CPU의 성능은 캐시 메모리의 크기와 캐시 계층을 얼마나 최적화 하느냐에 달려 있다. 주기억 데이터베이스의 최우선 목표는 성능 향상이다. 모든 수단을 도입할 필요가 있으며 캐시의 특성이나 캐시의 대체(replacement) 정책을 고려하여 주의 깊게 프로그래밍을 하여도 캐시의 영향으로 성능이 높아진다[1][14].

본 논문은 Tmr-트리의 단점인 삽입 시 속도가 느려지는 점을 보안하여 캐시를 고려하여 Tmr-트리를 이진트리로 구성하여 캐시 블록 크기만큼 모아 노드블록을 만들어서 캐시 미스 없도록 만들어서 Tmr-트리의 단점을 보안하여 주기억 색인 기법인 Tmr-트리가 캐시를 효율적으로 사용하도록 새로운 색인구조를 제안하였다. 그리고 시스템의 L2 캐시를 최대한 활용하도록 기존 Tmr-트리의 장점을 가지는 새로운 CSTmr-트리(Cache Sensitive Tmr-트리) 색인기법을 개발하고, 실험을 통해 다른 인덱스 구조와 비교하여 CSTmr-트리의 우수성을 보인다.

본 논문의 구성은 2장에서 관련연구로 RFID 유비쿼터스 이동체 데이터베이스 데이터에 대한 특징과 주기억 색인 구조인 Tmr-트리와 캐시에 대해 알아본다. 3장에서는 본 논문에서 제시하는 캐시를 고려한 새로운 주기억 데이터베이스에서 효율적인 색인 구조인 CSTmr-트리를 제안하고, 4장에서는 실험 결과로 CSTmr-트리 색인 구조와 Tmr-트리와의 검색 및 삽입 시간을 비교하며, 5장에서 결론 및 향후 연구를 제시한다.

# Ⅱ. 관련연구

## 2.1 유비쿼터스 RFID 이동체 데이터베이스

기존이동객체의 위치추적은 시간이 지남에 따라 위치가 변화하는 이동객체 데이터모델로 이산적 모델을 사용하여 표현하며 이동객체의 샘플링 된 위치를 점(x,y)로 나타내고  궤적을 다중선(polyline)으로  표현하며 3DR-Tree, TB-Tree 이동체의 최근보고 이전의 이동궤적을 검색하고 3차원 MBB(Minimum Bounding Box)형태로 색인에 저장하며 질의 종류는 영역질의, 궤적질의, 복합질의처리가 있다[16][17].

전자태그(RFID)는 Radio Frequency Identification의 약자로 자동인식(AIDC) 기술의 한 종류이다. Micro-chip을 내장한 Tag등에 저장된 데이터를 무선주파수를 이용하여 비접촉으로 읽는 기술로 태그 반도체 칩과 안테나는 이러한 정보를 무선으로 수미터에서 수십미터까지 보내며 Reader는 이 신호를 받아 상품 정보를 해독한 후 컴퓨터로 보낸다[8].

그러므로 태그가 달린 모든 상품은 언제 어디서나 자동적으로 확인 또는 추적이 가능하며 태그는 메모리를 내장하여 정보의 갱신 및 수정이 가능한 것이다. 태그의 위치추적이나 현재의 상태를 감시하는 자동화 생산, 재고 관리, 공급망 관리 등 다양한 응용분야에서 사용되기 때문에 RFID태그 추적의 필요성이 높아지고 있다[6][7].

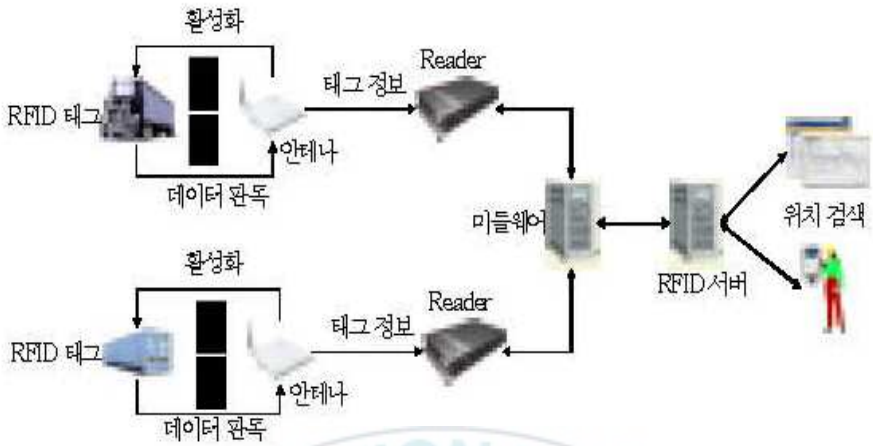
RFID 태그객체의 위치추적은 시간의 변화에 따라 위치가 변화하며 이동객체와 유사하지만 중요한 지점에 설치된 판독기에서만 위치를 보고하고 판독기의 인식영역의 크기와 수에 따라 위치의 정확성이 결정되어지므로  RFID 태그객체의 위치 추적을 위한 색인이 필요하고 새로운 과거 이력 표현 방법이 필요하며 순서와 시간을 기반으로 하는 데이터모델

과 색인구조가 필요하다[10][13].

그림1에서 RFID 시스템은 태그와 판독기, 그리고 안테나로 구성된다. 태그는 무선 주파수로 판독기와 통신할 수 있는 안테나와 객체 식별자(tid)를 저장하기 위한 메모리로 구성된다. 판독기는 응용 환경에 따라 전략적으로 중요한 지점에 설치되고, 고정 되어 있으며, 미리 계산된 위치 좌표(x,y,z)를 가질 수 있다. 또한, 판독기는 태그와 통신할 수 있는 3차원 공간상의 영역을 가지며 이것을 판독기의 호출 지역 이라고 한다.

RFID 태그 위치검색은 RFID 태그 객체가 호출 지역 안으로 들어가서 처음으로 인식되면 enter 이벤트가 발생하고, 호출 지역 밖으로 나가서 인식되지 않으면 leave이벤트가 발생한다. 이벤트가 발생할 때마다 판독기는 자신의 위치와 태그 객체의 식별자를 서버로 전송한다.

예를 들어서, 식별자 tid를 가지는 태그 객체가 판독기와 연결된 안테나의 호출 지역(loc)으로 이동할 때, 시간 t에서 인식 영역 안으로 들어가면 enter 이벤트가 발생하고 (tid, loc, tenter)가 서버로 전송된다. 반대로 객체가 호출 지역을 벗어나면 leave 이벤트가 발생하며 (tid, loc, tleave)가 서버로 전송된다. 이때 두 이벤트의 tid와 loc는 동일하지만 tleave는 tenter보다 항상 크다. 이렇게 수집된 데이터는 하나의 다차원 색인에 저장되고 검색된다[18].



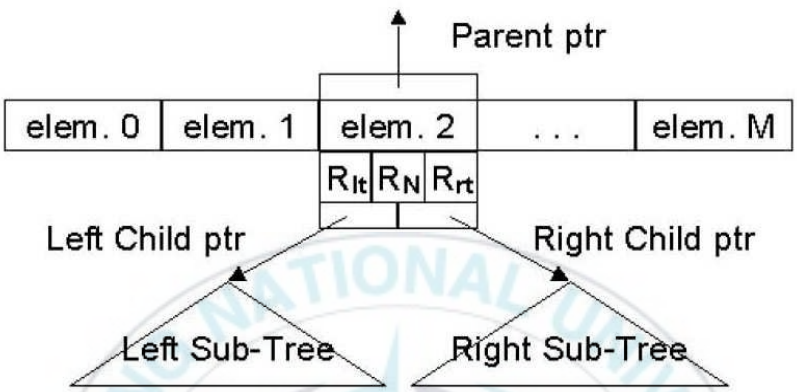

(그림 1) RFID 시스템 구성도

## 2.2 메인메모리 기반 Tmr-트리

주기억 데이터베이스에서 공간 데이터의 색인 구조로 주기억 데이터 색인구조인 T-tree 구조에 공간 데이터 색인 구조인 R-tree의 개념 중에 장점들을 포함하여 만든 색인 구조이다[9].

공간 데이터를 둘러싸는 MBR로 공간 데이터를 표현 할 때 메인메모리 기반 Tmr-트리 색인 구조는 높이 균형 트리인 T-트리 구조를 기본으로 사용하며 각 트리 노드의 구조는 그림 2와 같이 색인 노드의 구조는 T-트리 노드와 같이 부모노드에 대한 포인터(Parent ptr), 왼쪽 자식 노드에 대한 포인터(Left child ptr), 오른쪽 자식 노드에 대한 포인터(Right child ptr), 여러 개의 엔트리들, 그리고 새롭게 정의되는 왼쪽 서브트리의

MBR(Rlt), 자신 노드 N의 엔트리들에 대한 MBR(RN), 오른쪽 서브트리의 MBR(Rrt)으로 구성된다.




(그림 2) 공간 색인 구조

삽입시간에서 R-트리에 비해서 Tmr-트리가 약간 나쁜 성능을 보여주는데, 이것은 Tmr-트리 경우 노드 분할 발생 시 로테이션 수행과 같은 처리 지연 효과를 그 원인으로 설명할 수 있다. Tmr-트리 색인 구조는 R-트리와 달리 이진 검색에 의해 내부 노드에서 완료될 수 있기 때문에 빠른 검색이 가능하다.

2차원 이상의 공간 데이터의 경우, 삽입 및 삭제와 같은 연산이 자주 발생하지 않으므로 색인 구조는 주기억 장치 데이터베이스에서 공간 데이터를 다루는 데에 효율성을 가지고 있다고 볼 수 있다. Tmr-트리 색인 구조를 질의 연산인 조인과 같은 질의처리에 적용하는 방법이 R-트리에 비해서 삽입시간이 오래 걸린다는 단점이 있다[19].

## 2.3 캐시

캐시메모리는 적은 용량이지만 빠른 속도를 가지는 메모리로 최근에 사용되었던 데이터를 저장해서 프로그램의 성능을 높이는데 이용된다[4]. 캐시는 크게 용량(capacity), 연관성(associativity), 블록크기(block size) 세 가지 요소를 가지고 있다. 용량은 캐시메모리의 크기이며 블록크기는 메모리에서 캐시메모리로 전송하는 기본 블록 크기이다.

캐시 히트(cache hit)는 CPU가 필요로 하는 데이터가 캐시메모리에 있을 경우이며 캐시 미스(cache miss)는 캐시메모리에 데이터가 없을 경우를 말한다. 캐시 미스가 발생할 경우 메모리로부터 데이터를 캐시로 전송해야 하기 때문에 시간 지연이 발생한다.

최근의 컴퓨터는 대부분 L1, L2라 불리는 두 개의 캐시메모리를 가지고 여러 계층으로 설계한다. L1 캐시는 CPU의 클럭과 속도가 비슷하게 동작하지만 캐시메모리의 용량과 블록 크기가 매우 적다. L2 캐시는 메모리보다 속도가 빠르나 L1캐시보다 약간 느린 속도로 L2 캐시를 도입하게 되면 2배 성능 향상을 가져 올 수 있도록 동작한다.

캐시 히트가 빈번하게 발생해야 성능이 좋아진다. 캐시 히트의 발생빈도는 캐시의 대체 정책, 프로그램의 자료구조, 프로그램이 데이타를 액세스하는 방법에 밀접한 관련이 있고, 세심한 주의가 요구되는 프로그램을 작성하지 않으면 캐시메모리를 효율적으로 사용하지 못한다.

현대 컴퓨터에서 CPU와 메모리의 성능차이에 의해 메모리가 데이타베이스의 새로운 병목현상의 주된 원인이 됨을 보이고 캐시를 고려한 프로그램 설계 구현으로 메모리의 병목 현상을 완화시켜 성능 향상을 보였다.

CPU의 클럭이 높아질수록 메모리 접근으로 인한 속도 격차가 늘어나는 만큼 캐시는 중요하다. CPU 성능은 캐시메모리 크기와 캐시 계층을 얼마나 최적화 하느냐에 달려 있다[3].

# Ⅲ. CSTmr-트리의 제안

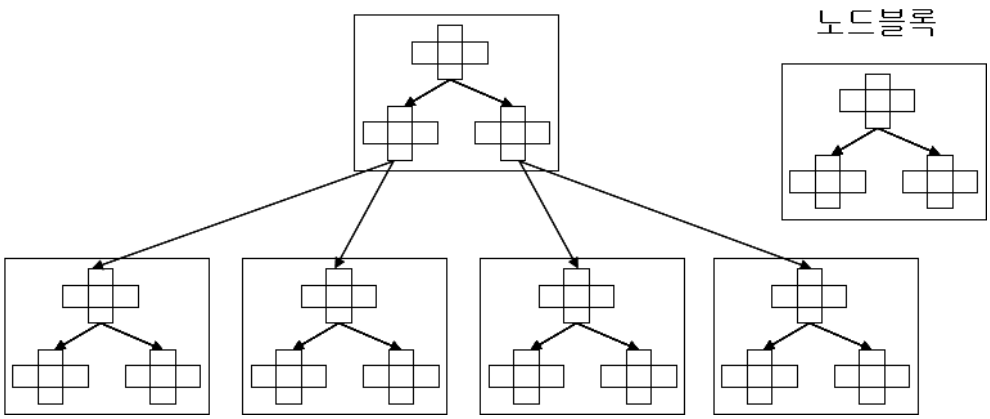
## 3.1 CSTmr-트리 구조

CSTmr-트리(Cache Sensitive Tmr-tree)는 캐시를 고려한 Tmr-트리를 설계한 인덱스 구조이다. 기존의 T-tree와 R-tree의 장점을 통합하여 만든 개념인 Tmr-트리에 캐시의 개념을 이용하여 만든 구조이다.

캐시 히트(Cache Hit)는 CPU가 필요로 하는 데이타가 캐시메모리에 있을 경우를 말하며, 캐시 미스(Cache Miss)는 캐시메모리에 데이타가 없을 경우를 말한다. 캐시 미스가 발생할 경우 메모리로부터 데이타를 캐시로 읽어오기 때문에 CPU의 성능이 저하되는 현상이 발생한다. 성능향상을 위해 캐시 미스를 줄이는 것이다.

캐시 미스를 줄이기 위해 캐시크기 만큼의 Tmr-트리를 모아 이진트리를 구성한다. Tmr-트리 노드를 캐시 블록 크기만큼 모아 만든 이진트리 블록을 '노드 블록'이라 부르기로 한다. 노드블록 사이즈가 캐시 블록 사이즈 일 때 가장 효율적이다. 노드 블록 내에서는 캐시 미스 없이 자식 노드를 읽을 수 있다. 노드 블록과 노드 블록 사이의 연결은 포인터를 이용한다.

그림 3은 CSTmr-트리의 논리적 구조이며 하나의 노드 블록에 연결된 자식 노드의 개수는 4개이다. 이 자식 노드의 공간을 할당 받을 때 각각을 할당 받게 되면 4개의 포인터가 필요하지만, 포인터가 하나만 있어도 인덱스의 개수가 4인 노드 블록의 배열로 공간을 할당 받으면 하나의 포인터와 배열의 인덱스로 각각의 자식 노드 블록을 액세스할 수 있다.




(그림 3) CSTmr-트리의 구조

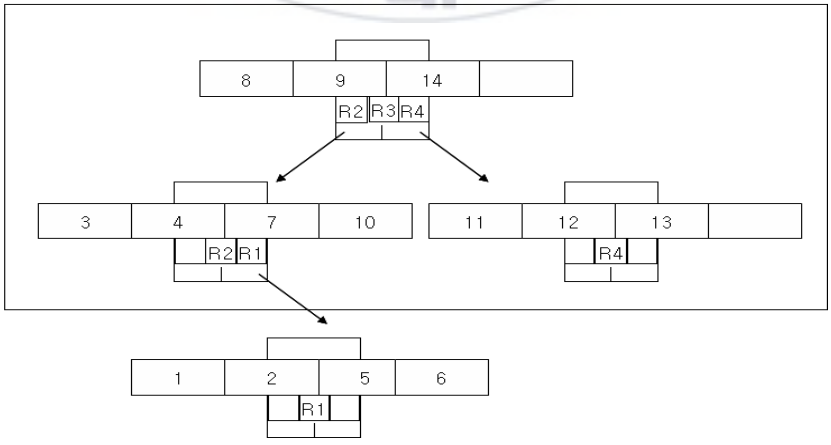
### 3.2 CSTmr-트리 검색 알고리즘

그림 4는 CSTmr-트리 검색 알고리즘으로 색인 구조를 사용한 검색을 위해서는 노드 블록을 B, 루트 노드는 N, 자신노드의 엔트리들에 대한 nt, 왼쪽 서브트리 lt, 오른쪽 서브트리 rt를 이용하여 객체 식별자(tid)를 검색 시, 자신 노드와 교차되는 지점이 있는지 검색하여 존재하면 자신 노드 엔트리들을 검색한다. 왼쪽 서브트리 교차점이 있는지 검색 후, 존재하면 왼쪽 자식 노드를 검색하고, 같은 방식으로 오른쪽 서브트리의 검색작업을 수행한다.

```
search(B, N) // B : node block, N : root node of CSTmr-tree
  if overlap(B, nt of N),
   for (each entry e of node N) if overlap(B, e), result=e
  if overlap(B, lt of N),
   result += search(B, N.left_child_ptr)
  if overlap(B, rt of N),
   result += search(B, N.right_child_ptr)
```

(그림 4) CSTmr-트리 검색 알고리즘

그림 5에서 만약 5지역 안의 한 지점을 검색할 경우, 우선 루트 노드에서 자신노드 R3과의 교차 여부를 확인 후, 왼쪽 서브트리 R2와의 교차 여부 확인, 오른쪽 서브트리 R4와의 교차 여부를 검색한다. 그러면, 해당 지점은 왼쪽 서브트리와 교차점이 있는 것을 알 수 있고, 왼쪽 자식 노드 R2를 루트노드로 하여 다시 검색 루틴을 수행해 오른쪽 서브트리 R1을 방문한다. 방문한 오른쪽 자식 노드에서 원하는 엔트리 5를 검색해 낼 수 있다.




(그림 5) 검색 시 트리구조

### 3.3 CSTmr-트리 삽입 알고리즘

그림 6의 CSTmr-트리 삽입 알고리즘(insert)은 검색을 하여 삽입할 노드를 찾아 삽입하고 삽입으로 인해 노드의 오버플로가 발생할 경우, 분할하여 엔트리들을 기존의 노드와 새롭게 생성한 노드에 분산 배치한다. 그림 7의 알고리즘으로 생성된 노드는 분할이 발생된 노드를 중심으로 거리가 최소가 되는 왼쪽 서브트리 또는 오른쪽 서브트리로 검색해서 들어간 후, 리프 혹은 반-리프 노드의 자식 노드가 되도록 배치한다.

```
insert(E, N) // E : object, N : root node of CSTmr-tree
if (minA(E,N)=nt of N),{
 if (N has empty),
     insert E into N, and adjust nt of N
 else {
     N' = split(N, E)
     insertN(N', N)
   }
  else if(minA(E,N)=lt of N),{
     insert(E, N.left_child_ptr), and adjust lt of N
     if (N.left_child_ptr is unbalanced),
         N.left_child_ptr = balance(N.left_child_ptr)
     }
    else {
     insert(E, N.right_child_ptr), and adjust rt of N
     if (N.right_child_ptr is unbalanced),
         N.right_child_ptr=balance(N.right_child_ptr)
  }
 }
```

(그림 6) CSTmr-트리 삽입 알고리즘(insert)

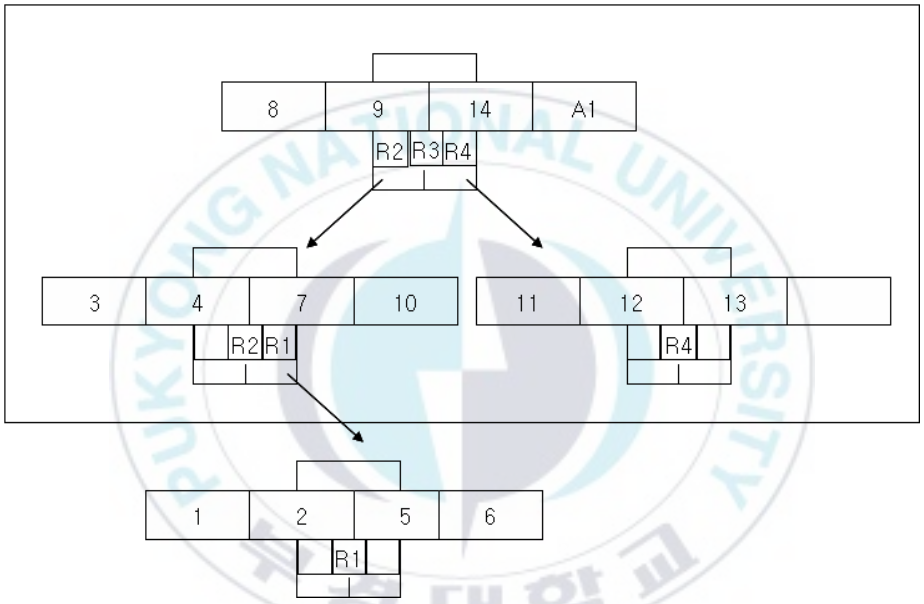
```
insertN(N', N) // N' : new node, N : node of CSTmr-tree
 if (N.left_child_ptr is NULL) N.left_child_ptr = N';
 else if (N.right_child_ptr is NULL) N.right_child_ptr = N';
 else {
   if (minA(N',N)=lt of N), {
     insertN(N', N.left_child_ptr);
   if (N.left_child_ptr is unbalanced),
      N.left_child_ptr = balance(N.left_child_ptr);
   } else {
     insertN(N', N.right_child_ptr);
   if (N.right_child_ptr is unbalanced),
      N.right_child_ptr = balance(N.right_child_ptr);
  }
 }
```

(그림 7) CSTmr-트리 삽입 알고리즘 2(insertN)

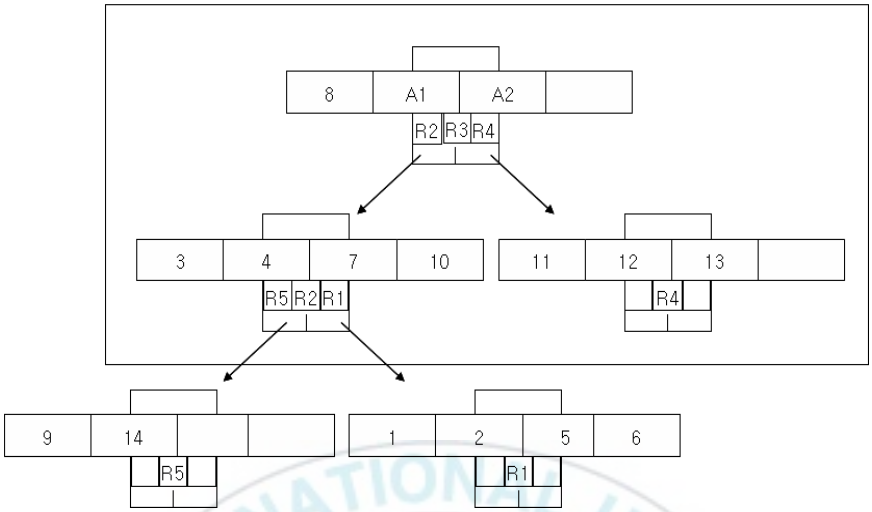
CSTmr-트리 삽입 알고리즘은 그림 5에서 제시된 검색 시 트리 구조에 새롭게 A1, A2를 삽입 할 경우 트리 구조는 아래와 같은 변환 과정을 거친다. A1이 삽입될 경우 우선 루트노드를 시작으로 자신노드 R3에 삽입할 경우와 왼쪽 서브트리 R2에 삽입할 경우, 오른쪽 서브트리 R4에 삽입할 경우를 비교해 거리가 최소가 되는 위치를 선택한다. 새로 삽입된 A1은 루트에 삽입되고 그림 8과 같은 트리 구조를 생성한다.

그림 9는 그림 8에서 A2를 삽입할 경우에는 위와 같은 방법으로 최적의 노드 R3을 선택한다. 그러나 A2를 R3에 삽입하기 위한 빈 엔트리가 존재하지 않기 때문에 오버플로가 발생한다. R-트리나 R*-트리의 분할 알고리즘을 이용해 노드 분할을 해서 새로운 노드 R5를 생성한 다음, 새로 생성된 노드 R5의 연결 위치를 결정한다.

분할이 발생된 루트 노드 R3은 새로 생성된 노드 R5를 포함할 경우 거리가 최소가 되는 왼쪽 R2, 오른쪽 R4 서브트리를 검색해 들어간다. 선택한 서브트리의 자식 노드 R2가 리프 또는 반-리프 일 경우, 트리의 높이가 높이균형이 되도록 생성된 노드 R5를 선택한 노드 R2의 자식노드가 되도록 연결한다.




(그림 8) A1 삽입 시 구조




(그림 9) A2 삽입 시 구조

## 3.4 CSTmr-트리 삭제 알고리즘

그림 10의 CSTmr-트리 삭제 알고리즘(delete)은 노드 안의 엔트리를 삭제를 할 경우에는 삭제 할 엔트리를 포함하는 노드를 검색해 삭제한다. CSTmr-트리의 트리구조는 리프와 모든 위치의 노드에 데이터를 저장하기 때문에 삭제 또한 모든 노드에서 이루어질 수 있다. 그림11의 알고리즘처럼 중간노드에서 삭제로 인해 언더플로가 발생할 경우에는 최소 직사각형의 거리를 비교하여 엔트리들 중 가장 거리가 작은 노드 엔트리를 검색하여 삭제 발생 노드로 이동시킨다.

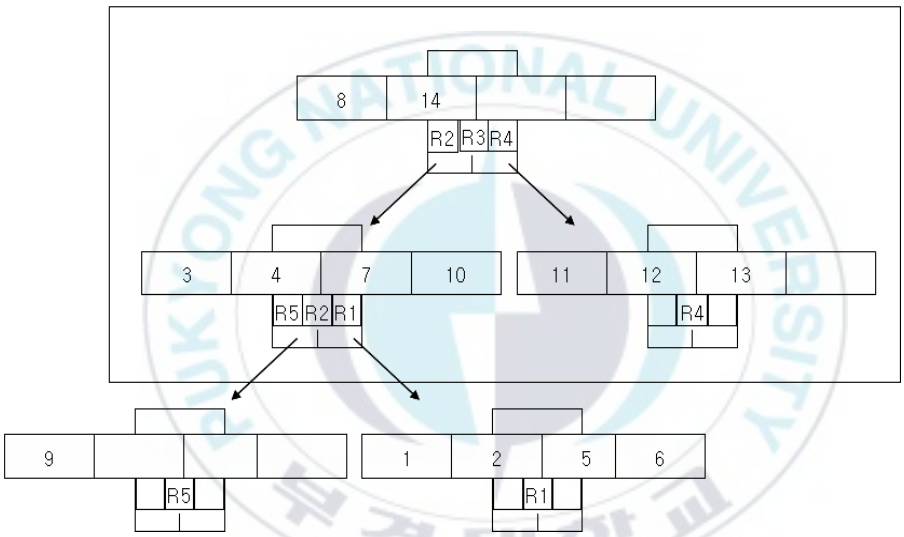
```
delete(E, N) // E: object, N : root node of CSTmr-tree
 if (N is underflow), {
  if (N is a leaf node), return
  else if(minA(E,N)=lt of N), {
       E = choose(nt, N.left_child_ptr)
       if (left sub-tree is unbalanced),
          N.left_child_ptr = balance(N.left_child_ptr)
       } else {
       E = choose(nt, N.right_child_ptr)
       if(right sub-tree is unbalanced),
         N.right_child_ptr = balance(N.right_child_ptr)
    }
   insert E into node N
 }
```

(그림 10) CSTmr-트리 삭제 알고리즘(delete)

```
choose(E, N)
// E: object of underflow node, N : node of CSTmr-tree
  if (minA(E,N)=nt of N), {
  search entry e in node N;
  result = remove e from N;
  If (N is underflow), delete(e's MBR, N);    return result;
  } else if (minA(E,N)=lt of N),
  result = choose(E, N.left_child_ptr);
  else result = choose(E, N.right_child_ptr);
  return result;
```

(그림 11) CSTmr-트리 삭제 알고리즘 2(choose)

그림 9에서 A1, A2를 삭제할 경우의 트리 구조는 다음 그림 12와 같다. A1이 삭제될 경우에는 노드에서 해당 엔트리를 삭제한 후 알고리즘이 간단하게 종료되지만, A2를 삭제할 경우에는 루트 노드에서 언더플로가 발생한다. 언더플로가 발생하면 루트 안의 엔트리 8과 최소의 거리를 유지하는 노드 R5를 검색한 후, 노드 R5에서 최소의 직사각형을 이루는 엔트리 14를 선택해 루트로 이동시킨다.




(그림 12) A1, A2 삭제 시 구조

## 3.5 CSTmr-트리 깊이 감축 알고리즘

그림 13은 CSTmr-트리는 검색 과정에서 성능에 가장 크게 영향을 미치는 것은 캐시 미스이다. 따라서 캐시 미스를 발생하게 하는 노드 블록 간의 높이균형에 더욱 신경을 쓸 필요가 있다. 노드 블록 간의 트리 깊이 감축은 Tmr-트리의 알고리즘을 변형하여 제안하였다. 노드 블록 간의 깊이 감축만큼이나 노드 블록 내의 이진트리 깊이 감축 알고리즘도 중요하다. 한 노드 블록 안에 최대한 많은 수의 노드가 포함되어 있어야 검색이 빨라지기 때문이다.

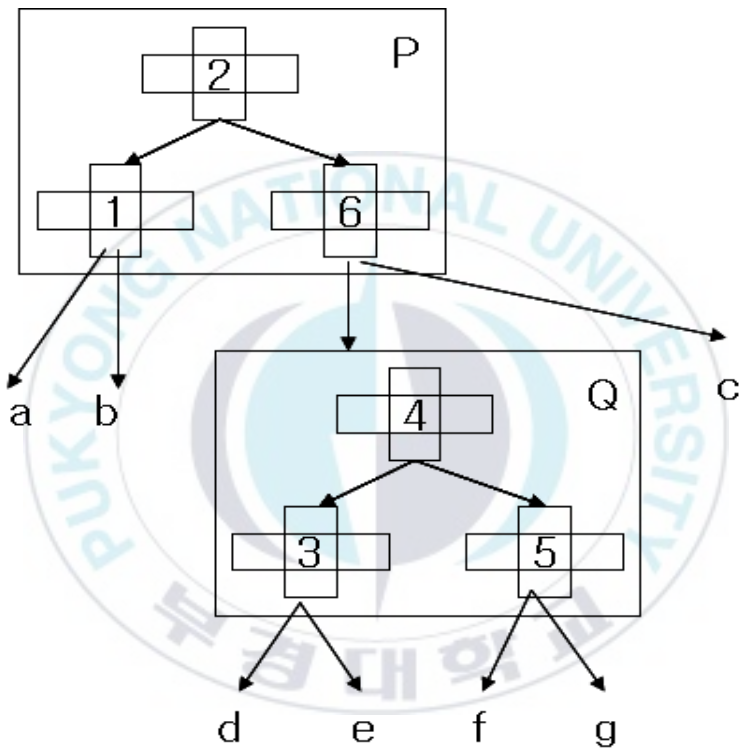
노드 블록 내의 이진트리 깊이 감축 알고리즘은 노드 블록 내의 이진트리는 캐시 블록 크기에 의해 크기가 제한되어 있으므로 균형이 깨질 때 마다 인덱스를 재배열하는 방법으로 높이균형을 조절한다. 노드 블록 내의 이진트리는 한 번에 캐시 블록으로 복사가 되므로 높이 균형 작업을 하는 중에 추가적인 캐시 미스는 발생하지 않는다.

노드 블록 간의 트리 깊이 감축 알고리즘은 삽입, 삭제 시에 높이 균형이 깨진 트리일 때 실행되어진다. 이것은 Tmr-트리에다 AVL트리의 알고리즘을 변형하여 만든 알고리즘이다.

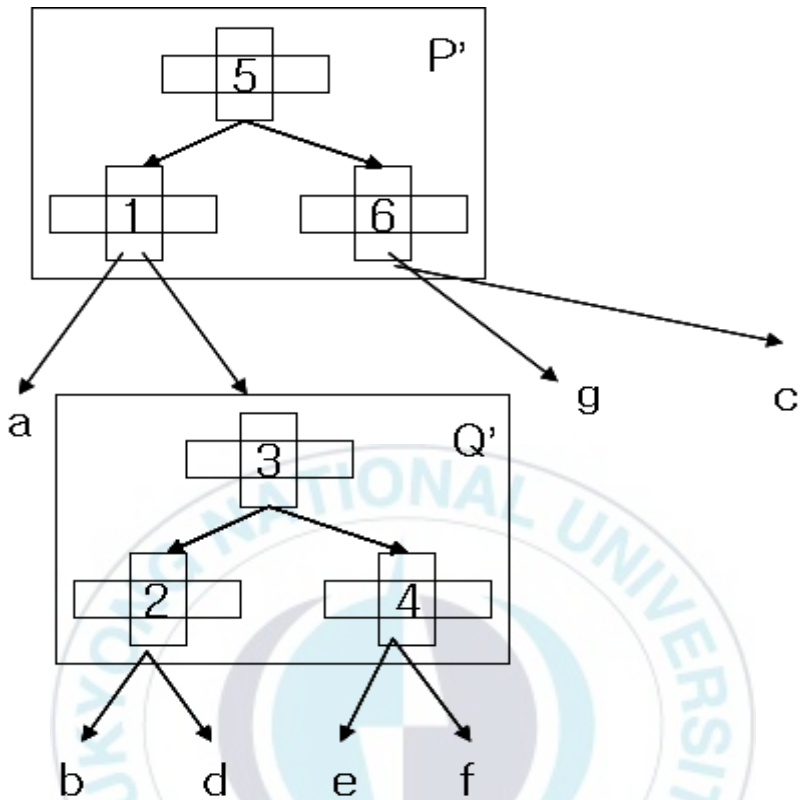
```
balance(B) // B : node block
if maxH(B) - minH(B) > 1 {
 if maxI(B) > minI(B)  {
    B' = maxI(B)
    if maxI(B) - minI(B) != 1 {
       e = maxI(B) -1
       s = minI(B)
       for ( i = s ; i < e; i++ ) {
         rotation(B, i+ 1, i)
       }
     }
    if maxI(B') != child {
       for ( i = maxI(B') ; i < child ; i++ )
         rotation(B' , i , i+ 1)
       }
    }
       rotation(B , maxI(B) , minI(B) )
else   //생략
  }
}
/* minH(B) : min Height(child of B),
maxH(B) : max Height(child of B),
minI(B) : min Index(child of B),
maxI(B) : max Index(child of B) */
```

(그림 13) 노드 블록 트리의 깊이 감축 알고리즘

CSTmr-트리 깊이 감축알고리즘의 예로서, 그림 14는 깊이5인 그래프는 노드 블록 P의 세 번째 노드 블록을 그림15의 노드블록 P의 두 번째로 옮기는 과정으로 회전이 일어나기 전과 회전이 일어난 후의 노드 블록의 변화를 보여 주고 있다.

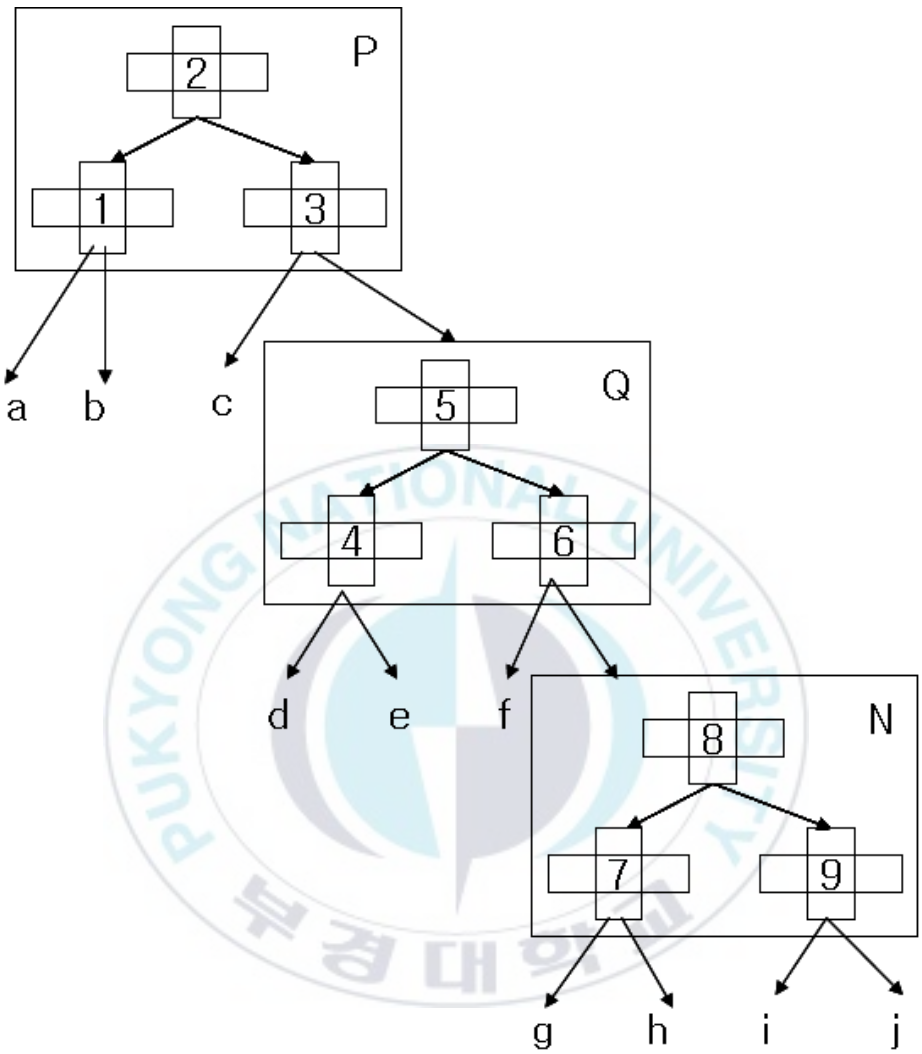



(그림 14) 깊이가 5인 그래프의 회전 전

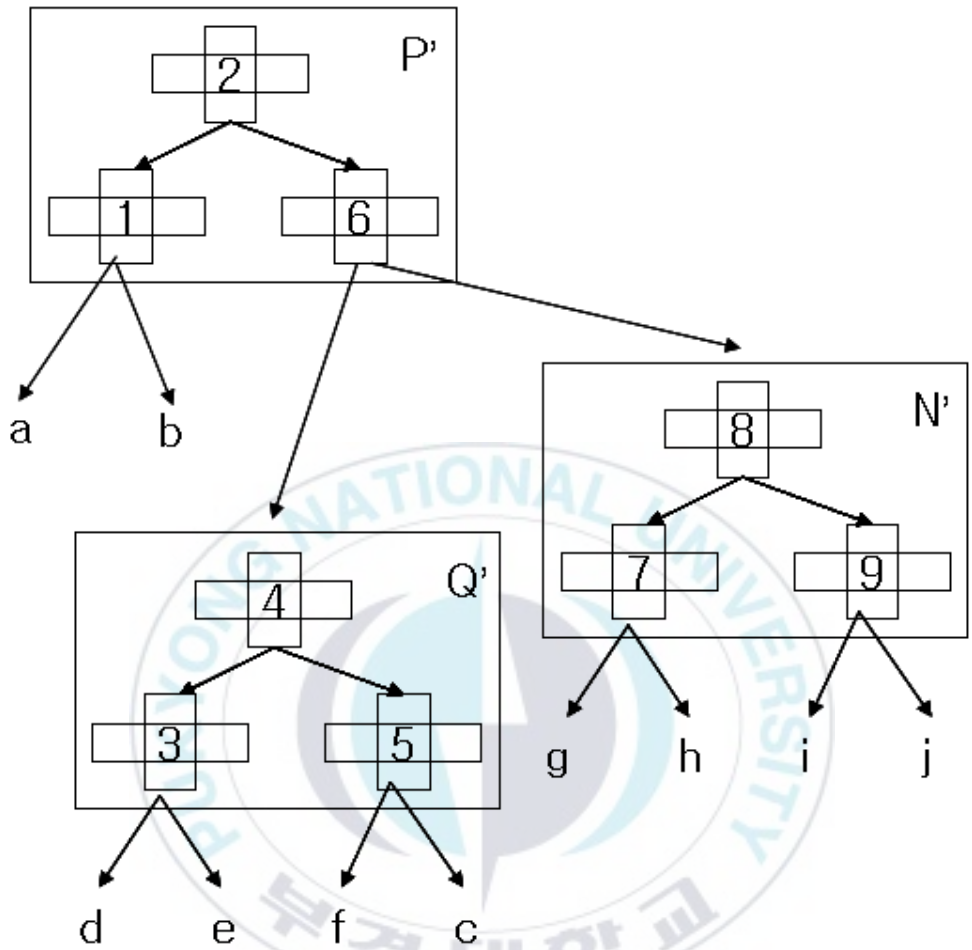



(그림 15) 깊이가 5인 그래프의 회전 후

그림 16은 깊이가 7인 그래프의 오른쪽 트리가 노드 블록 N 삽입으로 인해 Q의 높이가 1증가된 상태로 높이균형이 깨진 트리라고 가정하자. 이때, 그림 17처럼 노드블록 P의 네 번째 노드블록을 노드블록 P의 세 번째로 옮기는 회전 알고리즘을 적용하면 높이가 감축된다.




(그림 16) 깊이가 7인 그래프의 회전 전




(그림 17) 깊이가 7인 그래프의 회전 후

# Ⅳ. 성능 평가

## 4.1 실험 환경

CSTmr-트리는 C 프로그래밍 언어로 구현되었고 Linux 운영체제에서 GNU gcc컴파일러로 컴파일 했다. 실험을 한 컴퓨터는 Intel Pentium(R) IV 3.0Ghz CPU와 1.0GB DDR RAM의 컴퓨터로 1M의 L2 캐시 메모리를 가지고 실험하였다.

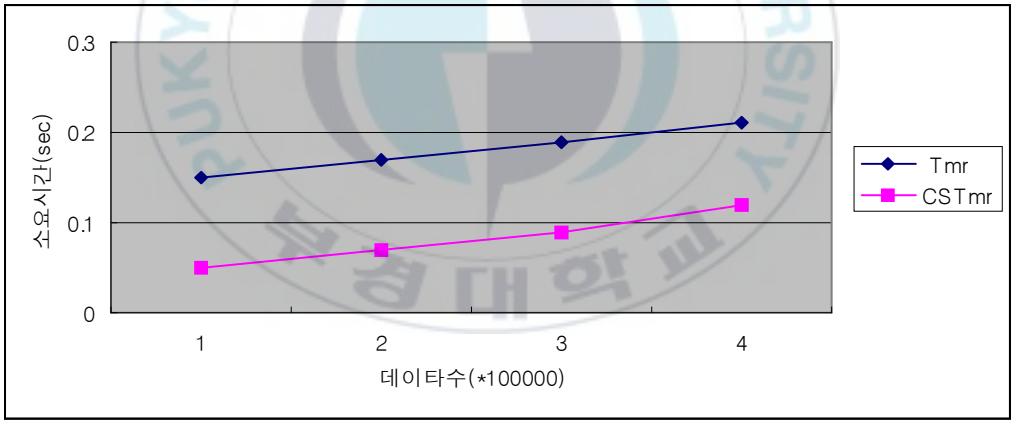
## 4.2 실험 방식

이동체 데이터베이스를 위한 데이터 셋을 이용하여 각각 100,000, 200,000, 300,000, 400,000 개의 임의의 키를 가지고 있을 때 200,000건의 임의 검색을 수행하였고, 10,000, 20,000, 30,000개의 키를 임의 생성하여 삽입 수행을 하였으며, 또한 200,000개의 키를 임의 생성하여 10,000, 20,000, 30,000개의 키를 임의로 삭제 수행을 하였다.

## 4.3 실험 결과

### 4.3.1 검색(Search) 성능

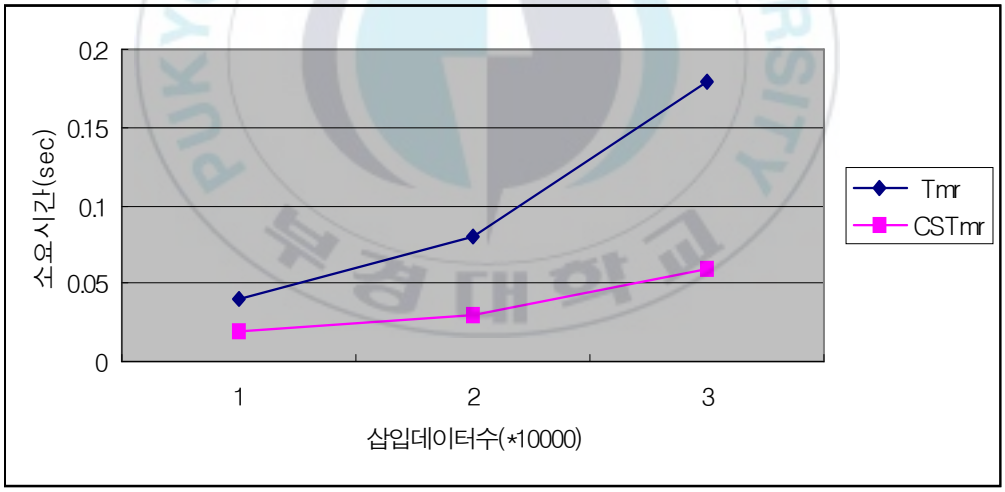
그림 18은 트리가 각각 100,000, 200,000, 300,000, 400,000 개의 키를 가지고 있을 때 200,000 건의 임의검색(Random Search)을 수행했을 때 걸린 시간을 그래프로 표시한 것이다. 트리가 데이터수에 따라 검색 (Random Search) 수행했을 때 걸린 시간이 CSTmr-트리가 Tmr-트리보다 1.8~3배 성능의 효과가 있었다.




(그림 18) 검색의 수행 시간

4.3.2 삽입(Insertion) 성능

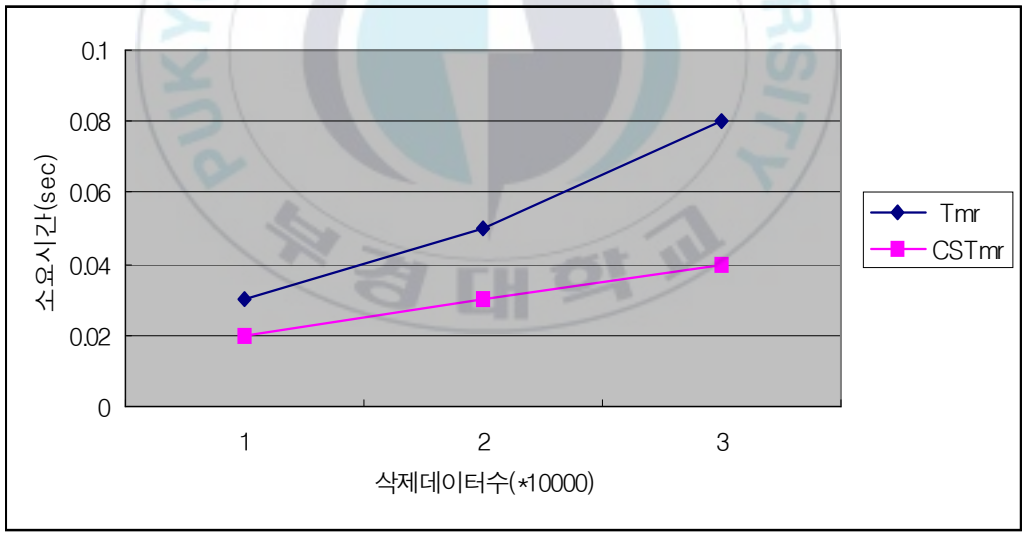
그림 19는 10,000, 20,000, 30,000 개의 키를 임의로 생성하여 삽입했을 때 걸린 시간을 그래프로 표시한 것이다. 트리가 데이터 수에 따라 삽입 수행했을 때 걸린 시간이 CSTmr-트리가 Tmr-트리보다 2~3배 성능의 효과가 있었다. 또한 Tmr-트리는 키가 증가 할수록 속도가 현저히 늦어지는 것이 보이는데 Tmr-트리의 단점이 삽입 시 속도가 느려진다. 그래서 단점을 보안하여 캐시를 고려한 CSTmr-트리를 제안하여 삽입 수행속도를 높였다.




(그림 19) 삽입의 수행 시간

### 4.3.3 삭제(Deletion) 성능

그림 20은 200,000 개의 키를 임의 생성하여 트리를 구축한 후, 10,000, 20,000, 30,000 개의 키를 임의로 삭제했을 때 걸린 시간을 그래프로 표시한 것이다. 트리가 데이터 수에 따라 삭제 수행했을 때 걸린 시간이 CSTmr-트리가 Tmr-트리보다 1.5~2배 성능의 효과가 있었다. 삭제 시에도 Tmr-트리가 속도가 느려지는 단점이 있어 그것을 고려하여 캐시기반 삭제기법을 제안하여 CSTmr-트리의 속도가 빨라진 것을 볼 수 있다.




(그림 20) 삭제의 수행 시간

# Ⅴ. 결론 및 향후연구

본 논문에서는 캐시를 고려한 주기억 데이터베이스에서 이동체 데이터를 위한 효율적인 색인 구조인 CSTmr-트리를 제안하였다. 캐시를 고려한 인덱스 설계로 주기억 데이터베이스 색인 성능을 높이고자 Tmr-트리에 적용하여 만들어졌다. 지금까지 이동체 데이터에 대한 색인 구조는 디스크 기반의 데이터베이스를 위한 구조이기 때문에 디스크 접근 횟수의 감소와 디스크 저장 공간의 낭비 없는 사용에 그 초점이 맞춰져 있었다. 그러나 주기억 데이터베이스에서는 디스크 기반 데이터베이스와는 달리 처리 속도 빠름과 효율적인 메모리 공간 사용이라는 목적을 두고 있다.

또한 메인 메모리의 속도는 CPU 만큼 빨라지지 않았기 때문에 메모리 액세스 속도 차를 극복하기 위해서는 CPU와 메모리 사이의 속도 격차를 줄이기 위해 만들어진 캐시메모리를 이용하여 캐시의 특성이나 캐시의 대체(replacement) 정책을 바탕으로 프로그래밍을 하므로 L2 캐시의 성능을 높일 수 있다.

시스템 L2 캐시 활용도를 높일 수 있도록 Tmr-트리의 구조를 새로이 설계하고 검색 알고리즘을 제안하였으며, 삽입, 삭제 시에는 노드블록 높이균형을 위해 제안한 구조에 맞게 새로운 트리 깊이 감축 알고리즘을 제안하였다. 제안한 알고리즘의 타당성을 위해 실험을 하였다. 실험의 결과에 의하면, Tmr-트리보다 CSTmr-트리가 검색, 삽입, 삭제 시에 성능이 우수함을 알 수 있었다.

본 논문에서 제안한 새로운 CSTmr-트리에 대해 다음의 향후 추가 연구가 필요하다. DBMS의 새로운 대안으로 하이브리드 MMDBMS 등장 시점에 맞추어 캐시를 고려한 주기억 데이터베이스 색인에서 더 발전한

캐시를 고려한 주기억 데이터베이스 색인과  디스크 기반의 데이터베이스 색인의 혼용 구성 연구가 필요하다.

# 참고문헌


[1] Jun Rao, et al, “Cache Conscious Indexing for Decision-Support in Main Memory”, VLDB 1999.

[2] David J. Randy H. Katz, Frank Olken, Leonard D. Shapiro, Micheal R. Stonebraker, David Wood “Implementation Techniques for Main Memory Database Systems”, In Proc. of ACM SIGMOD, Boston, 1984.

[3] Peter Boncz, et al, “Database Architecture Optimized for the new Bottleneck : Memory Access”, VLDB 1999.

[4] Guttman, A. “R-tree: A dynamic index structure for spatial searching”, In Proceedings of the ACM SIGMOD International conference on Management of data, 1984.

[5] Hongjun Lu, Yuet Yeung, Ng Zengping, “T-Tree or B-Tree : Main Memory Database Index Structure Revisited”, Australasian Database Conference, 2000.

[6] S. E. Sarma, S. A. Weis, and D. W. Engels, “RFID Systems and Security and Privacy Implications ,” Springer-Verlag, pp454-469, 2002.

[7] K. Romer, T. Schoch, “ Infrastructure Concepts for Tag-Based Ubiquitous Computing Applications ,” Workshop on Concepts and Models for Ubiquitous Computing, 2002.

[8] EPC Tag Data Standard Work Group, "EPC Tag Data Standards Version 1.23," EPC Global, 2005.

[9] Tobin J. Lehman, “A Study of Index Structures for Main Memory Database Management System”, VLDB 1986.

[10] Voler Gaede, Oliver Gunther, “Multidimensional access methods”, ACM Computing Surveys, Vol.30, No.2, June, 1998.

[11] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, “The R*-tree: An Efficient and Robust Access method for Points and Rectangles”, In Proc. ACM SIGMOD International Conference on management of Data, pp.332-331, 1990.

[12] T. Sellis, N. Roussopoulos, and C. Faloutsos, “The R+-tree: A dynamic index for multidimensional objects,” In Proc. 13th Int. conference on Very Large Databases, pp.507-518, 1987.

[13] P. Bones, S. Manegold, and M. Kersten, “Database Architecture Optimized for the New Bottleneck: Memory Access”, Proceedings of VLDB Conference, pp.54-56, 1999.

[14] Jun Rao, et al, “Making B+ Trees Cache Conscious in Main Memory”, SIGMOD 2000 .

[15] A. Aho, J. Hopcroft and J. D. Ullman, “The Design and Analysis of Computer Algorithm”, Addison-Wesley Publising Companay, 1974.

[16] D. Pfoser, C. S. Jensen, and Y. Theodoridis, “Novel Approaches to the Indexing of Moving Objects ,” Proc. of the 26th VLDB Conf., pp395-406,2000.

[17] Y. Theodoridis, M. Vassilakopoulos, and T. Sellis, “Spatio-temporal indexing for large multimedia applications ,” Proc. of the 3rd IEEE.

[18] 김동현,이기형,홍봉희,반재훈, “RFID 태그 객체의 위치 추적을 위한 색인 구조의 설계 및 구현”, 한국공간정보시스템학회논문지 제2권 제2호, 2004.

[19] 윤석우, 김경창 ,“Tmr-트리:주기억 데이터베이스에서 효율적인 공간 색인 기법”, 정보처리학회논문지D 제12-권 제 4 호, 2005.