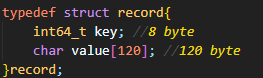
2021058995 황인혁

**개발 환경**

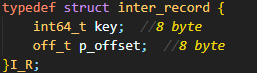
Gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.00

GNU Make 4.2.1

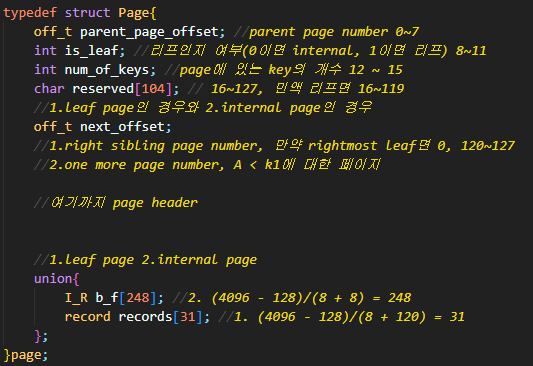
**구조체 설명**



**Leaf page**의 경우 사용하는 구조체이다.



**Internal page**의 경우 사용하는 구조체이다.



**Leaf page, internal page, free page**인 경우를 한 구조체로 표현하였다.

만약 free page인 경우 **parent\_page\_ofsset**은 다음 free page를 나타낸다.

**-Page header 부분**

**parent\_page\_offset**은 부모 페이지의 offset을 나타낸다.

**Is leaf**는 0이면 internal page, 1이면 leaf page를 나타낸다

**Num\_of keys**는 해당 페이지에 존재하는 key의 개수를 나타낸다.

**Reserved**는 페이지의 크기를 맞추기 위해서 존재하는 padding과 같은 역할이다.

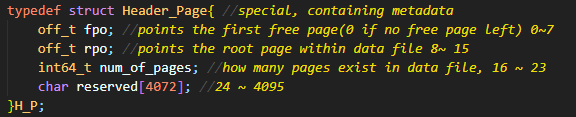
**Next\_offset**은 leaf page의 경우 **right sibling page number**를 의미하며 만약 rightmost leaf인 경우 0값을 가진다.

만약 internal page인 경우 **Next\_offset**은 **one more page number**를 의미하며 가장 작은 key값보다 작은 값들에 대한 page의 offset을 담고있다.

**-나머지 부분**

Internal page의 경우 **b\_f**를 사용하며 **b\_f**는 key와 page number에 대한 offset을 가진 구 조체의 배열이다.

Leaf page의 경우 **records**를 사용하며 **records**는 key와 value에 대한 구조체의 배열이다.



헤더 페이지의 구조체이다.

**Fpo**는 free page의 첫번째 offset을 나타낸다. Fpo 값이 0이라면 남아있는 free page가 없 다는 의미이다.

**Rpo**는 root page의 offset을 나타낸다.

**Num\_of\_pages** 는 현재 data file에 존재하는 사용된 page의 개수를 의미한다.

**Reserved**는 페이지의 크기를 맞추기 위해서 존재하는 padding과 같은 역할이다.

**함수 설명**



Pathname을 매개변수로 받아와 해당 파일이 존재하지 않다면 새로 만들고, 이미 존재한다면 해당 파일을 연다

.

Header page에 대한 offset(항상 0)을 인자로 넘기면 헤더페이지 구조체 포인터를 반환한다.



Load하고싶은 page에 대한 offset을 매개변수로 넘기면 해당 offset에 대한 페이지 구조체 포인터를 반환한다.



New\_page() 함수에서 더 이상 free page가 존재하지 않을 때 lseek(fd,0, SEEK\_END)를 통해 파일의 끝으로 이동시킨 뒤 reset함수를 통해 calloc으로 새로 페이지를 만든다.



Free page가 있다면 freetouse함수를 통해 해당 free page를 초기화시켜준 뒤 offset을 반환한다. Free page가 없다면 reset함수를 호출한 뒤 파일의 가장 끝 offset을 반환한다.



매개변수로 넘긴 key가 존재할 수 있는 leaf page의 offset을 반환한다.



Db에 키가 존재하는지 확인한 뒤 존재하지 않는다면 null을, 존재한다면 key에 대한 value값을 반환한다.



New\_page()함수에서 호출되며 페이지를 사용하기 이전에 페이지를 초기화시켜주는 역할을 수행한다.



매개변수로 넘어온 length의 절반에 해당하는 인덱스를 반환해준다.



최초로 insert가 일어나는 경우 실행되는 함수이다. 루트 페이지를 생성한 뒤 넘어온 레코드 매개변수를 해당 페이지에 넣어준다. 글로벌 루트 포인터 rt에 대한 정보를 갱신한다.



먼저 해당하는 key가 leaf page에 이미 존재하는지 확인한 뒤 만약 존재한다면 -1을 반환, 존재하지 않는다면 key가 들어갈 해당 페이지가 꽉 찼는지 여부에 따라 다른 함수를 호출한다.



만약 페이지가 꽉 차지 않아 key가 잘 들어갈 수 있다면 호출되는 함수이다. 해당 키를 정렬된 순서에 맞게 넣어준다.

 //새로 추가한 함수

스플릿이 발생하기 이전에 호출되는 함수이다.

우선 해당 페이지가 루트라면 바로 insert\_into\_leaf\_as 함수를 호출한다. (부모 페이지가 없기 때문에 key\_rotation이 발생할 수 없다)

1.만약 왼쪽 페이지에 공간이 존재한다면 해당 페이지(꽉찬 페이지)의 가장 작은 값을 왼쪽 이웃으로 옮기고 부모의 key를 해당 페이지(꽉찬 페이지)의 새로운 가장 작은 값으로 갱신

2.만약 오른쪽 페이지에 공간이 존재한다면 해당 페이지(꽉찬 페이지)의 가장 큰 값을 오른쪽 이웃으로 옮기고 부모의 key를 이웃 페이지의 가장 작은 값(옮겨준 값)으로 갱신

만약 왼쪽과 오른쪽 모두 공간이 존재하지 않는다면 insert\_into\_leaf\_as함수를 호출한다.



스플릿을 발생시키는 함수이다. 기존 페이지보다 사이즈가 하나 더 큰 temp라는 레코드 배열을 만든 뒤 추가되는 레코드인 inst까지 포함시켜 정렬한다. Temp와 cut함수를 활용하여 두 페이지로 분리시킨다(split). 부모 페이지로 올라가는 값을 매개변수로 insert\_into parent함수를 호출한다.



만약 parent page offset이 0이라면(루트라면) insert\_into\_new\_root함수를 호출하여 새로운 루트를 만든다. 부모로 올라온 key의 인덱스를 확인하기 위하여 get\_left\_index를 호출한다. 만약 부모의 스플릿이 발생하지 않는다면 insert\_into\_internal함수를 호출하고, 스플릿이 발생한다면 insert\_into\_internal\_as를 호출한다.



왼쪽 자식으로 연결되는 부모의 인덱스를 구해준다. Insert into\_parent 함수에서 호출되며 insert\_into\_internal과 insert\_into\_internal\_as로 해당 값을 넘겨준다.



새로운 루트 page를 만들고 자식 정보, 자식들에 대한 부모 정보, 루트 글로벌 포인터 등 필요한 값들을 갱신해준다.



Internal page가 꽉 차지 않은 상태에서 호출되며 해당 internal page로 추가된 레코드에 대한 정보(key값과 자식에 대한 offset 값)를 갱신한다.



Internal page가 꽉 찬 상태에서 레코드 삽입이 이루어지려는 상태이다. leaf에서 split이 일어나는 상황과 동일하게 temp라는 임시 레코드 배열을 활용한다. 트리의 위 level로 올라가는 상황은 루트까지 도달하기 전까지는 insert\_into\_parent가 호출된 뒤 insert\_into\_internal\_as가 재귀적으로 호출되며 스플릿이 일어나지 않는 level까지 반복된다(insert\_into\_parent의 insert\_into\_internal이 호출되고 종료됨). 만약 루트까지 가서 루트가 스플릿하게 되는 상황이 발생한다면 insert\_into\_parent의 insert\_into\_new\_root가 호출되어 새로운 루트가 만들어지고 recursion이 종료된다.



먼저 루트의 key 개수를 통해 해당 트리가 비어 있는 트리인지 확인한다. 그 다음 지우고자 하는 키가 리프에 존재하는지 확인한다. 존재하지 않다면 delete\_entry를 호출한다.



Remove\_entry\_from\_page를 호출하여 해당 레코드를 삭제한다. 루트 페이지의 경우 adjust\_root를 호출하여 루트에 대한 정보를 갱신한다.



Merge가 일어나기 이전 만약 이웃 노드로부터 키를 가져올 수 있다면 key rotation을 진행하는 함수이다. 본 코드(수정한 코드)에서는 거의 호출될 일이 없으며 internal page에서만 이웃 페이지가 꽉 찰 경우 발생한다. merge과정을 조금 늦추는 효과가 있다.



현재 페이지의 키가 부족할 경우 이웃 페이지와 merge를 진행해주는 함수이며, 재귀적으로 delete\_entry를 호출하여 tree의 level을 올라가 페이지들을 알맞게 배치시킨다. 본 코드(수정한 코드)에서는 부족함의 기준을 페이지 최대 키 개수의 절반이 아닌, 비어 있는 상태(empty page)의 경우이며 따라서 부족할 경우 해당 page를 free 시키고 재귀적으로 올라가 페이지들을 정비해준다. 이렇게 개선한 이유는 아래 상세하게 설명 되어있다.



루트의 키의 개수가 0일 때 만약 루트가 리프가 아니라면 자식을 새로운 루트로 만들고, 리프라면 빈 트리가 된다. Delete\_entry에서 쓰인다.



Delete\_entry에서 호출되며, 해당 페이지로부터 레코드를 삭제한다. 만약 루트페이지라면 루트를 갱신한다.



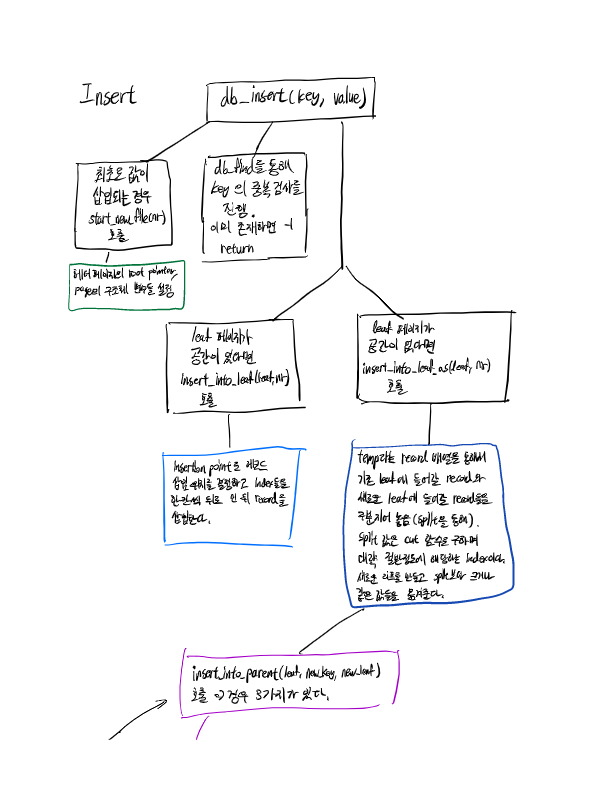
Coalesce\_page와 adjust\_root에서 쓰이며 페이지를 초기화 시킨 뒤 free page로 만들어주는 함수이다.



Find\_leaf에서 호출되며 이진 탐색을 통해 internal node의 index를 찾아 나간다. Internal page의 order가 leaf에 비하여 압도적으로 많기 때문에 internal에 대해서만 binary search를 진행하였다. 기존 binary search와는 조금 다르게 구현하였으며, 이유는 아래 설명에 자세하게 나와있다.

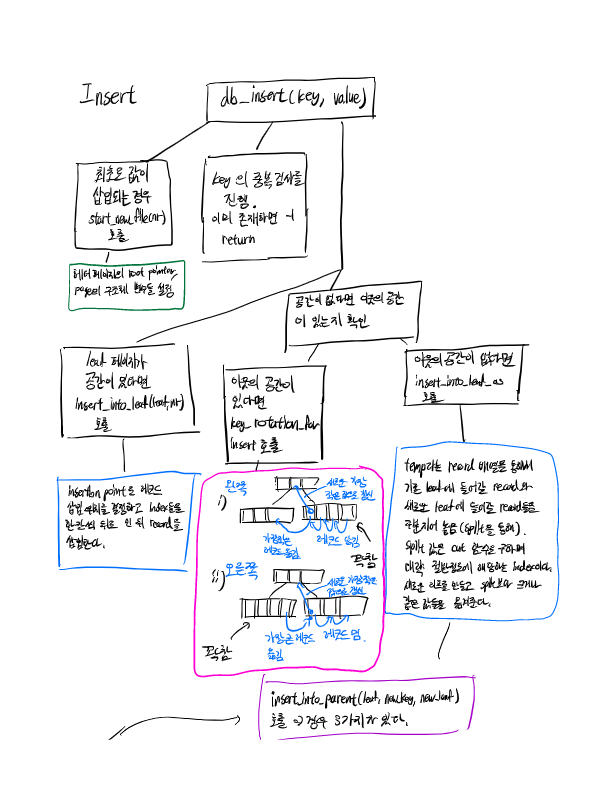
**Detail flow of the structure modification, DESIGN**

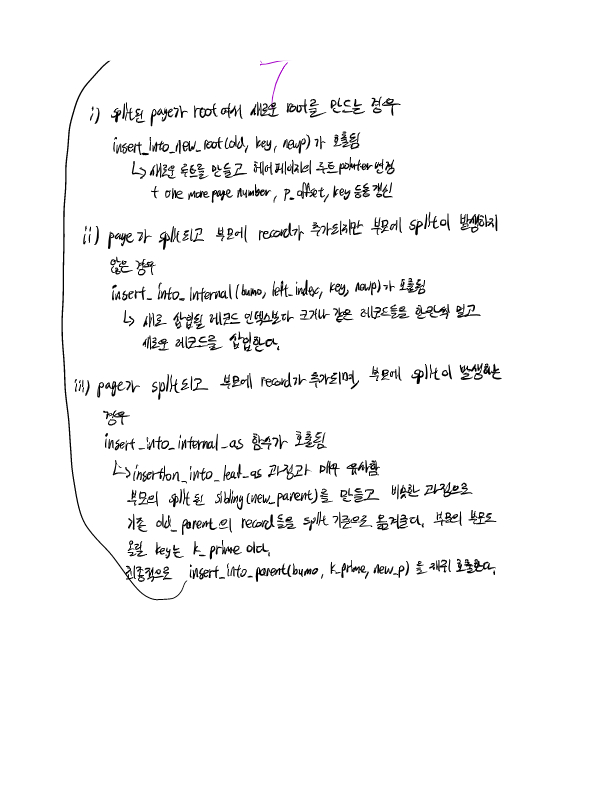
* **Insert/split ORIGINAL**

텍스트이(가) 표시된 사진

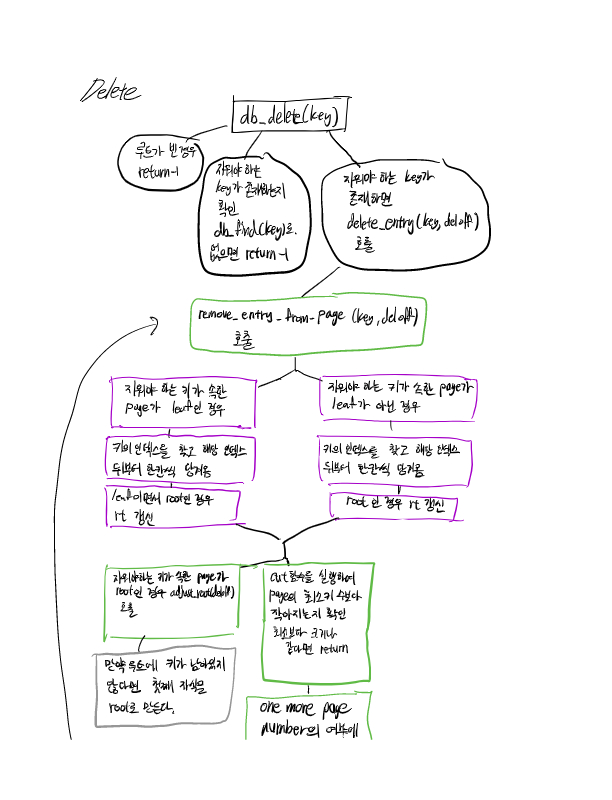
자동 생성된 설명

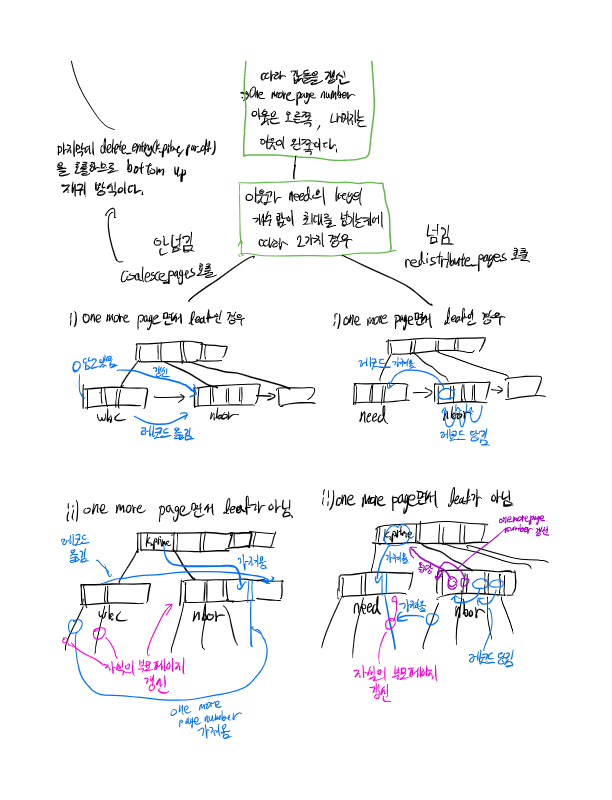
**-insert/split MODIFIED**

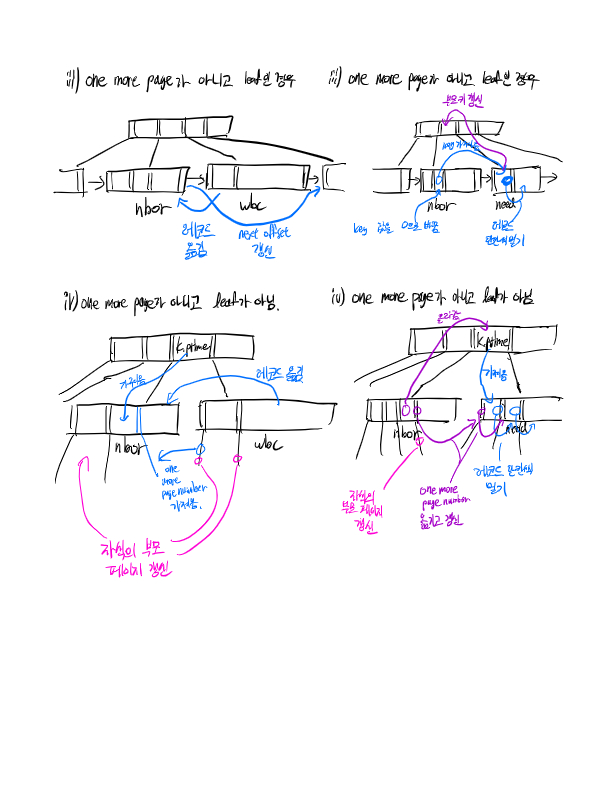




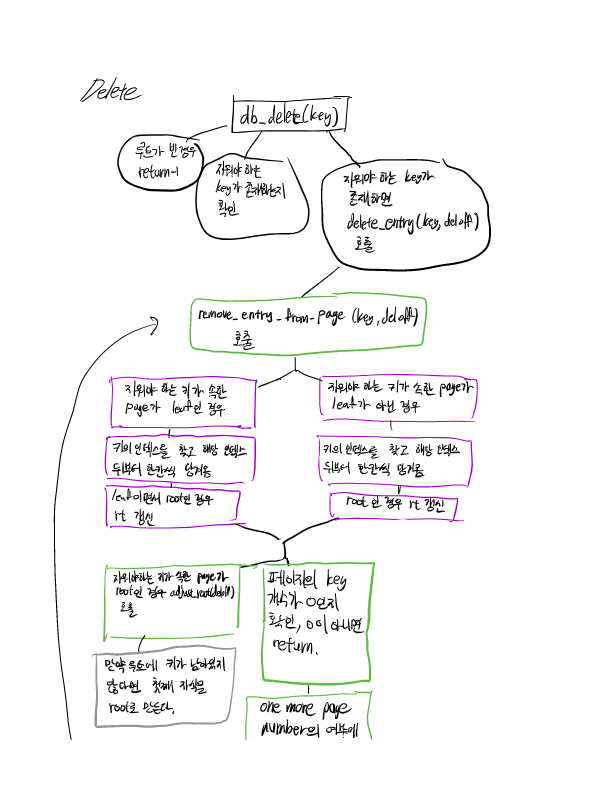
**-delete/merge ORIGINAL**

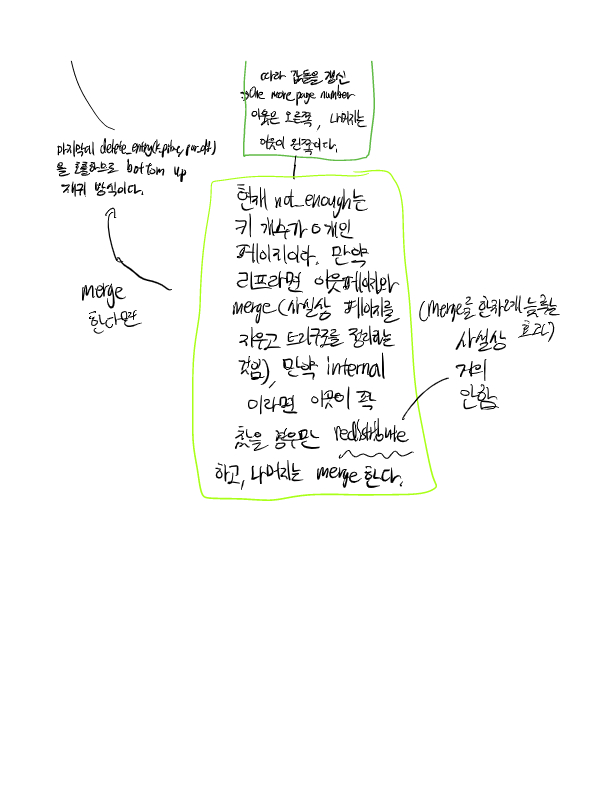






**-delete/merge MODIFIED**

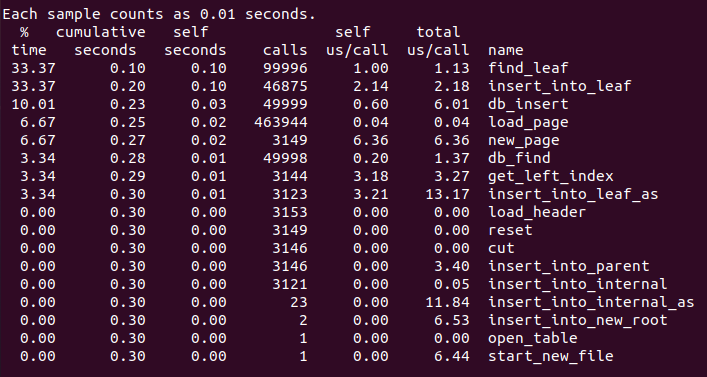
****



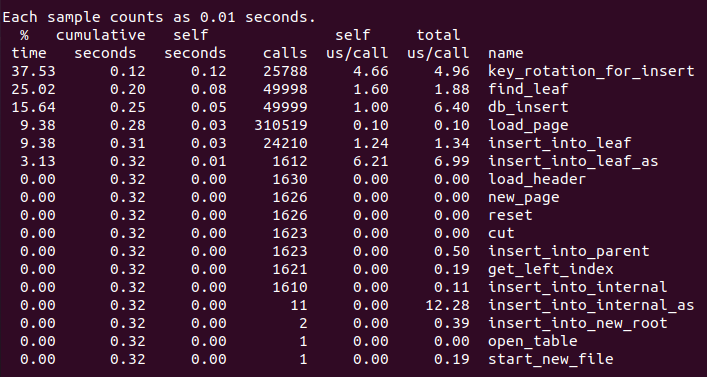
**성능검사를 위한 c언어 프로파일러, gprof**

-리눅스에는 내장 되어있는 프로파일러인 gprof가 있다. 만약 설치되어 있지 않다면 Sudo apt-get install binutils를 통해 설치가 가능하다. 컴파일 할 때 -pg옵션을 준 뒤 컴파일 된 바이너리 파일을 실행하고 gprof ~~ gmon.out > profOutput.txt 를 하면 profOutput.txt를 통해서 해당 c 파일에 속한 함수들의 호출 횟수와 시간, 시간 비율 등을 구할 수 있다.

**Design required changes for reducing overhead in B+tree, IMPLEMENT, RESULT**



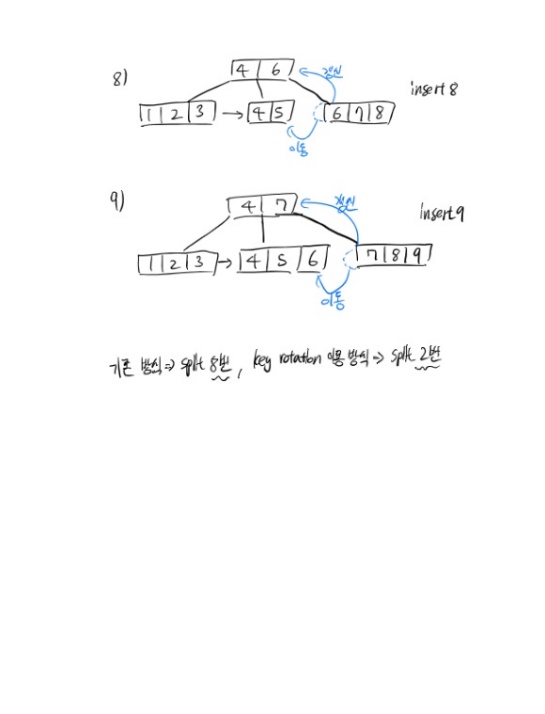
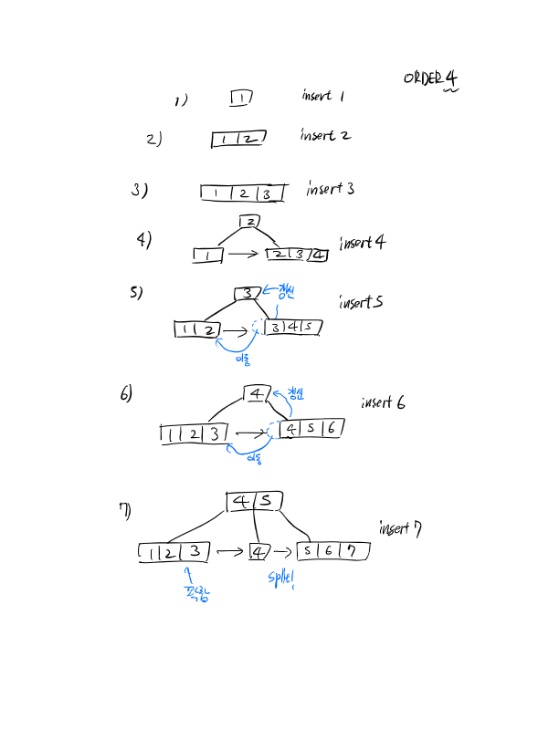
original b+tree **INSERTION** with 50000 inputs



Improved(db\_insert 함수 개선, key rotation 적용) b+tree **INSERTION** with 50000 inputs

Insertion과정에서의 split을 줄이기 위해서 만약 insert를 진행한 이후 해당 페이지가 꽉 찼다면, split을 진행하기 이전에 이웃 페이지(왼쪽, 오른쪽)를 확인해보고 왼쪽 페이지에 공간이 있다면 가장 작은 값을, 오른쪽 페이지에 공간이 있다면 가장 큰 값을 이동 시킨 뒤 부모의 키 값을 갱신한다. 양쪽 이웃 모두 공간이 부족한 경우에만 split을 진행하면 split연산의 횟수는 월등히 감소한다**(insert\_into\_leaf\_as의 호출 횟수는 3123번에서 1612번으로 대폭 감소하였다).** 게다가 Reconstructuring(split, merge)는 concurrent한 b+tree 알고리즘에서 병목(serialization bottleneck)을 만들기 때문에 되도록이면 안 발생 하는 것이 좋다.

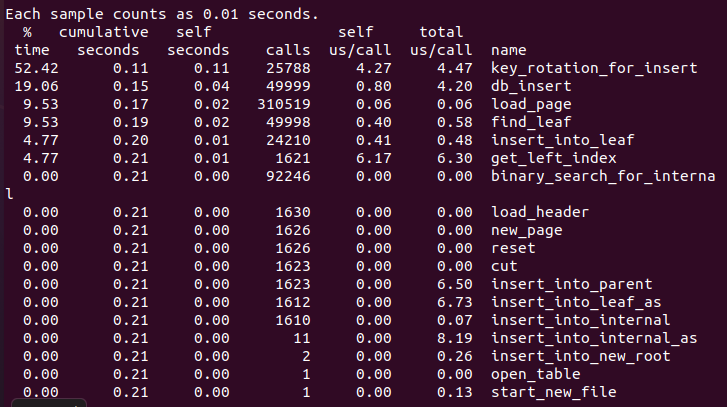
sorted order로 insert를 진행한다면 계속 tree의 오른쪽으로만 page의 변화가 생기며 왼쪽에 있는 page들은 공간이 많이 생겨 페이지의 낭비가 생긴다. 따라서 key로테이션을 지속적으로 하며 대부분의 페이지들을 거의 채운 상태로 유지하는 것이 트리의 높이를 낮게 유지하여 disk io를 줄일 수 있다. **결과 비교 사진에서도 new\_page의 호출 횟수가 절반으로 감소한 것을 확인 할 수 있다.**



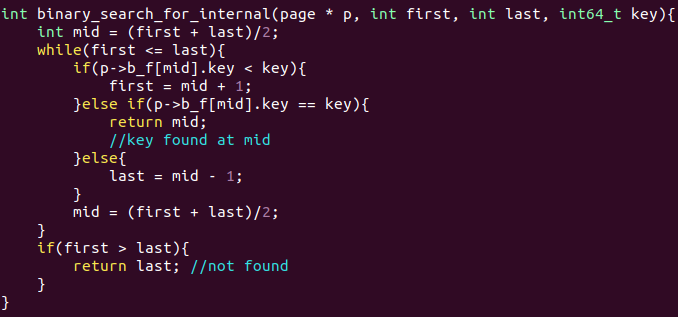
Db\_insert함수를 관찰해보면 insert하고자 하는 key가 leaf에 존재하는지 확인하기 위하여 db\_find를 호출하고 만약 존재하지 않는다면 다시 find\_leaf함수를 호출하여 key가 속할 수 있는 leaf\_page의 offset을 구한다. Db\_find 과정에서 이미 find\_leaf 함수를 호출하므로 불필요한 load\_page의 오버헤드가 발생한다. Insert 과정에서 중복검사를 하는 과정 이전에 미리 find\_leaf를 하고 db\_find를 안 쓰는 방법으로 바꾸었다. **(결과적으로 load\_page 횟수를 463944번에서 310519번으로 낮출 수 있었다)**

Db\_delete함수도 마찬가지의 방법으로 key가 leaf에 존재하는지 확인하였기 때문에 db\_insert와 같은 방식으로 load\_page 횟수를 줄일 수 있었다. **(아래 사진 참고, 470771번에서 307038번으로 낮추었음)**

Internal page의 order가 무려 248이기 때문에 linear search를 통해 자식의 offset을 찾아 나가는 과정은 줄일 필요가 있다. Binary search를 find\_leaf 함수에 적용하면 탐색 시간은 O(n)에서 O(logn)으로 줄일 수 있다. 일반적인 binary search와는 코드가 조금 다른 것을 확인 할 수 있는데, leaf page 바로 위 level의 internal page의 경우는 찾고자 하는 키가 internal node에 존재할 수 있으면서 존재하지 않을 수도 있다. 존재하는 경우는 mid의 index를 반환하고 존재하지 않는 경우는 last를 반환하여 last에 속한 key보다 더 큰 key를 가진 자식 page쪽으로 내려갈 수 있도록 하였다. Leaf page 바로 위 level이 아닌 나머지 internal node에서는 찾고자 하는 key가 일치하여 존재할 수 없으므로 last를 반환하는 과정을 통해 level을 내려갔다. 함수들의 call 횟수에는 변화가 없지만 total us/call을 살펴보면 find\_leaf를 내부적으로 실행시키는 함수들의 시간이 전부 줄어든 모습을 확인할 수 있다.



more improved(db\_insert 함수 개선, key rotation 적용, binary search 적용) b+tree **INSERTION** with 50000 inputs



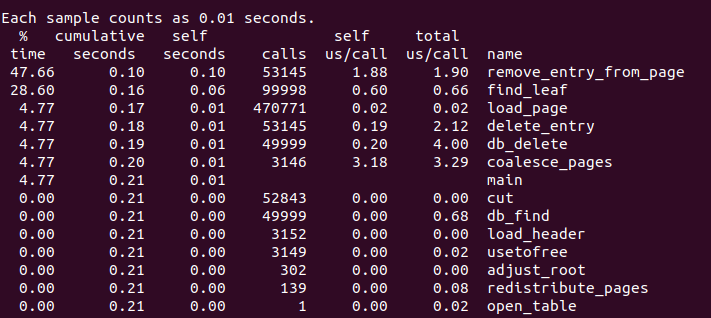
Modified Binary search code

Merge 연산을 줄이기 위하여 delay merge를 적용해볼 수 있다. 만약 insert가 여러 번 발생할 뒤에 양 옆 페이지까지도 전부 꽉 찬 상태라면 split이 일어난다. Split이 일어나면 해당 페이지는 새로운 이웃 페이지와 함께 절반정도 찬 상태가 된다. 만약 기존 b+트리의 상황이라면 절반정도 찬 상태의 page에 delete가 발생하면 underflow가 발생하여 merge를 수행하려고 한다. 즉 만약 insert와 delete가 교차되어 반복적으로 일어나면 split과 merge가 계속 발생하여 매우 큰 오버헤드를 만들 가능성이 있다. 따라서 우리는 해당 페이지의 merge 조건을 절반보다 적게 만들어 merge가 바로 일어나지 않도록 만들 수 있다. 하지만 이렇게 했을 경우 page의 space utilization lower bound가 낮아져서 b+tree의 성능을 오히려 저하시킬 수 있다. Insert와 delete가 어떤 순서로 얼만큼 일어나는지 여러가지 상황을 생각해보며 overhead를 생각해 보아야 하며 space utilization과 reconstructure rate(split, merge)의 trade-off를 고려해야 한다.

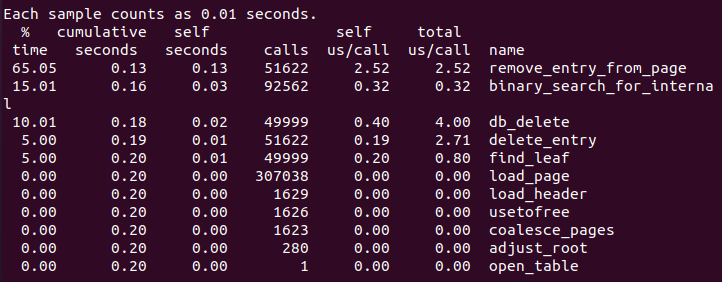
참고[1]을 보면 이와 관련된 논문이 있다. 이 논문의 저자는 극단적인 상황으로 key의 개수가 0이 되는 순간까지 merge를 안 하다가 0이 되면 해당 page를 free page로 만드는 방법(free-at-empty)을 선택하였고 이 방법이 merge-at-half 보다 성능이 ‘일반적으로’ 좋다는 것을 증명하였다. Merge-at-half 는 항상 50%의 space utilization lower bound을 가지며 insert와 delete의 횟수가 비슷할 경우 space utilization은 60%이다. 반면에 free-at-empty는 0% space utilization lower bound를 가지지만 insert와 delete의 횟수가 비슷할 경우 space utilization은 39%이다. 이렇게만 보면 free-at-empty가 안 좋아 보이지만 만약 insert가 delete보다 약간이라도 더 많아진다면 free-at-empty의 space utilization은 상당히 높아진다. Reconstructuring rate(split, merge)를 q라고 가정하면 어떠한 상황에서도 merge-at-half가 q>0.1를 만족하며 free-at-empty는 매우 작은 값을 유지한다. Reconstructuring은 또한 concurrent한 b+tree 알고리즘에서 병목(serialization bottleneck)을 만들기 때문에 되도록이면 안 발생 하는 것이 좋다. 따라서 이와 같은 모든 상황을 고려했을 때 free-at-empty b+tree가 성능이 좋은 것을 확인 할 수 있다. (자세한 증명은 논문 참고)

기존의 코드를 활용하되 논문의 아이디어를 반영하여 새로운 방식의 merge와 redistribute를 진행하였다. 원래 코드에 따르면 1. 현재 페이지에서 키 삭제로 인해 페이지의 키 개수가 절반 아래로 내려간다면 왼쪽 노드의 키 개수와 현재 페이지의 키 개수의 합을 구하여 merge가 진행 가능한 지 확인한다. 2. 가능하다면 merge를 진행하고 페이지의 최대 값보다 크다면 redistribute를 진행하였다.

**개선한 방식-** 1.key의 개수가 0이 아닌 상태까지는 merge나 redistribute가 일어나지 않는다. 2.만약 키 개수가 0이라면 왼쪽 노드의 키 개수를 구하여 merge 조건에 부합하는지 확인한다. Internal page의 merge 과정에서는 부모의 key값이 내려와서 merge를 수행하기 때문에 max값이 internal\_max-1이었고 leaf page의 경우에는 리프 페이지와 리프의 부모 페이지의 값 중복 때문에 max값이 leaf\_max였다. 따라서 merge가 발생하는 조건은 왼쪽 페이지가 internal\_page의 경우 internal\_max보다 작고, leaf\_page인 경우는 leaf\_max보다 작거나 같은 경우이다. Redistribute가 발생하는 조건은 internal\_page의 경우 왼쪽 페이지가 internal\_max보다 크거나 같고, leaf\_page의 경우 leaf\_max보다 큰 경우이다(leaf\_max보다 큰 경우는 존재할 수 없으므로 internal\_page에서만 redistribution이 발생한다). 사실 redistribution을 사용하거나 사용하지 않거나 성능차이는 매우 미미하고 redistribution의 발동 조건이 매우 까다롭다. 매우 특수한 경우에 대해서 merge를 한차례 늦춰준다는 장점과 이웃 internal 페이지의 스플릿 발생을 한차례 늦춰준다는 장점 때문에 사용하였다.



original b+tree **DELETION** with 50000 inputs



Improved(db\_delete함수 개선, delay merge 적용, binary search 적용) b+tree **DELETION** with 50000 inputs

Troubleshooting

-우분투에서 해당 zip 파일을 열기 위해 두가지 시도를 하였다. 첫 번째 시도는 윈도우에서 클론한 깃 폴더로 해당 zip 파일을 넣은 뒤 git bash로 푸쉬하고 우분투 상에서 pull하여 갖고 오는 방법이었다. 정상적으로 갖고 오는 데에 성공하였으나 실수로 main.c를 지워버렸고 다시 pull하려니까 everything up-to-date가 나오며 아무것도 변하지 않았다. 아직 깃의 작동 방식을 명확하게 알고 있지 않아서 그런 것 같다. 해당 zip파일을 그냥 지워버리고 두 번째 방법으로 시도하였다. 두 번째 방법은 우분투 상에서 zip 파일을 바로 다운 받는 방법이다. 다른 수업에서는 disk image를 서버로 사용하였기에 desk-top 버전의 disk image를 새로 깔았다. 우분투 상에서 파일을 다운 받고 압축 해제한 뒤 실행하니 정상적으로 작동하였다.

-b+tree와 관련된 성능에 대한 자료들을 검색하려고 하였으나 관련 자료가 많지 않았다. 따라서 관련 논문들을 찾기 위해 google scholar와 science direct를 이용하여 참고할 수 있을 만한 자료들을 얻었다. 논문의 구성 방식과 해당 논문에서 참고한 다른 논문들의 참고 표시 등 논문에 대해 자세하게 알 수 있었다.

-split 횟수를 줄이고 height를 줄이기 위한 목적으로 split 이전에 이웃 노드의 공간이 있다면 key rotation을 진행하였다. 처음에 5만개의 input을 넣고 비교를 해보려고 하였으나 tree의 order가 너무 커 height에서는 비교적 이득을 보지 못했다. sequential하게 1부터 50000까지 insert를 해본 결과 tree의 레벨이 3개밖에 나오지 않았는데, 키 로테이션을 적용한 상황과 적용하지 않은 상황 모두 3개로 동일 하게 나왔다. 생각해보니 한 internal page당 fanout이 무려 248개이고 leaf page의 수용 가능한 key의 개수가 31개여서 모든 페이지가 꽉 찬다 가정했을 경우 248^2\*31이라는 매우 큰 숫자가 나오기 때문에 height에 눈에 띄는 성능 변화를 주기 위해서는 최소 수억 개의 input 데이터가 필요하거나 order를 줄이는 방법 밖에는 없었다.

-gprof 사용할 때 컴파일 과정에서 pg옵션을 주는 방법을 몰랐었다. MakeFile을 열어보니 CFLAGS+= -g -fPIC -I가 있었고 여기의 -g를 -pg로 바꾸니 올바르게 gprof를 사용할 수 있었다. 사용한 뒤 다시 원상태로 돌려놨다.

참고

1. B-Trees with Inserts and Deletes: Why Free-at-Empty Is Better Than Merge-at\_Half\*

by THEODORE JOHNSON AND DENNIS SHASHA,

<https://www.sciencedirect.com/science/article/pii/002200009390020W>

1. <https://noanswercode.tistory.com/2> - gprof 사용 방법