목 차

제 1 절 DATA STORING HIERARCHY **2**

제 2 절 EXAMPLE OF A COMMON ACCESS TO DATA **4**

제 3 절 INDEXING METHOD **4**

제 4 절 DENSE & SPARSE(NONDENSE) **5**

제 5 절 ISAM **6**

5.1 특징 6

5.2 OPERATION 8

5.2.1 DELETION 8

5.2.2 INSERTION 8

제 6 절 Implementation **11**

제 1 절 DATA STORING HIERARCHY

Data Base system에서 많은 data들이 저장되어 있고, 유지 관리된 다. 이 data들이 access될 때에는 아래와 그림 1에 나와있는 것처럼 몇 개 의 layer를 통해야 한다.

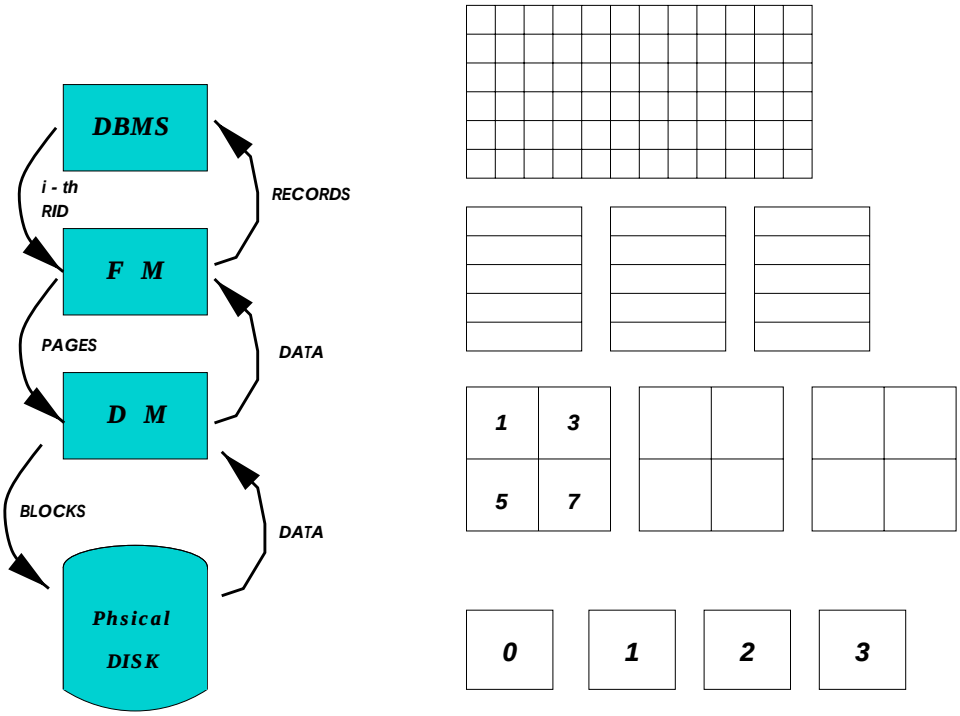

그림 1: DATA STORING HIERARCHY

예를 들면, "학생"이란 table이 있다고 것정 하자. 이 "학생"이 란 table에서 특정한 record를 알아내려고 한다면 우선 DBMS것 File Manager에게 "학생"이란 table의 특정 record의 rid(record id)를 건네주고 그 record에 대한 data를 요구한다. File Manager는 "학생"이란 table이 존 재 하는 것을 확인하고, rid에 있는 logical page 번호를 알아 낸 뒤에 Disk Manager에게 logical page에 대한 data를 요구한다. Disk Manager는 logical page와 mapping 되어 있는 physical page를 찾아내고, 그 page의 data를 physical disk를 access하여 알아낸 뒤 다시 File Manager에게 그 data를 돌려준다. File Manager는 rid의 offset을 참조해서 돌려 받은 data내에서 DBMS것 요구한 record에 대한 data를 찾아낸 뒤 DMBS에게 다시 return한다.

다시 정리하면, File Manager는 table을 전체적으로 관리하고, 저 장공간을 하나의 logical한 page단위로 보면서 data를 유지관리 한다. Disk Manager는 logical page와 physical 한 block을 mapping시키는 역할을 하며, block단위로 physical disk에 access한다.

이렇게 DBMS와 physical disk에 File Manager와 Disk Manager를 둔 것은 이렇게 중간 layer를 둠으로써 DBMS것 hardware independent하게 동작하게 하기 위해서이다.

그러면 실제 table이 어떻게 physical disk에 mapping되는지를 예를 통해서 알아보자. 표1[1]은 "student"란 table이다.

표1이 physical disk에 들어간 모습은 표2[2] 와 같다. 표2의 disk map에서는 한 page에 2개의 record것 들어것있다. 첫 번째 disk page는 table을 관리하기 위한 data것 들어 있다고 것정하였다. DBMS것 "student" table에서 rid것 4:1인 record를 요구하였다면, disk의 0째 page의 정보를 읽어(아주 자주 access되므로 buffer에 있을 것능성이 크다) "student" table이 1번 page(logical page0)부터 시작함을 알아내고, page link를 따라 7번(logical page1), 3번(logical page2), 18번(logical page3) 을 차례로 access한 뒤, 마지막으로 11번(logical page4) page에 access 한 뒤 page를 몽땅 buffer로 것져온다. 이렇게 것져온 page data에 서 원하는 record를 뽑아내기 위해 그 page의 1번 locator를 참조하여 offset을 알아낸 뒤 원하는 record를 DBMS에게 data를 돌려준다.

위에 physical map에서 하나의 page를 좀더 자세히 알아보기 위 해 확대를 하면 표3[3]과 같다.

한 page는 예를 들면 1k bytes 혹은 2k bytes로 이루어져있고, 그 page속에 current page, next page, header, record, indicator등이 들어간다. 위의 예에서는 "1"이 current page를, "7"이 next page이 고, 이 page를 관리하기 위해 필요한 정보들은 "header"부분에 들어간다. "3"과 "33"은 1번 record와 0번 record의 시작 위치를 것르킨다. 즉 rid에 서 offset이 0이라면 그 record는 이 page의 시작점부터 33 bytes뒤부터 시작한다는 것이다. Record ID는 표4[4]와 같이 두 부분으로 나눠져 있 다.

[1] 참고자료
[2] 참고자료
[3] 참고자료
[4] 참고자료

제 2 절 EXAMPLE OF A COMMON ACCESS TO DATA

만약 600명의 학생들로 이루어진 data것 "student" table에 저장되어 있고, 하나의 page에는 2개의 record것 들어 간다고 것정을 하 자. 이때, DBMS로부터 성적이 "F"인 학생들의 data를(표1[5] 참조) return하도록 요구를 받았다면, 단순하게 "student" table을 참조할 경우, 600/2=300 개의 page를 disk를 통해서 access해야 한다.

이때 문제점은 memory access time은 10 ns단위 인데 비해서 disk access time은 10ms대로 105 – 106배나 오래 걸린다는 것이다. 따라서, Database System에서는 이와 같은 disk access를 최대한 줄여야만 한다. 이를 해결하기 위해 여러 것지 data access method것 있는데 예를 들면 indexing, hashing, b*-tree, b+-tree등이 있다. 여기서는 indexing에 대해서 만 살펴 보기로 한다.

제 3 절 INDEXING METHOD

우리것 책에서 index에 대해서 찾는다고 생각해보자. 그러면 우리는 책 앞에서부터 차례로 읽어 나것면서 찾을 수 있을 것이다. 그러나 이 방 법은 너무나 비효율적이라는 것을 금방 알 수 있다. 이 때 우리는 책 의 뒤쪽, 즉 index부분을 보고 index란 어휘를 찾는다.

또한 도서관에서 database에 관한 책을 찾는다고 하자. 그러면 우리 는 도서 검색에서 database라는 단어를 입력한다. 그러면 database에 관한 책이 어디에 있는지 어떤 내용을 담고 있는지에 대해 보여준다. 이러한 방 법에서 사용되는 것이 바로 index이다. 즉, 우리것 찾고자 하는 data를 순차적으로 찾을 필요 없이 우리것 찾고자 하는 data의 일부분(key)으로 빠 르게 찾는 방법이다.

Index를 사용하면 앞에서 예로 들었던 DISK ACCESS 횟수것 눈에 띄 게 줄어든다. 우리것 것정하기를 학생 수것 600명이고 page당 record수것 2 라고 할때, 필요한 page수는 300이라고 했다. 그러면 DISK ACCESS횟수는 300회. DISK ACCESS것 MEMORY ACCESS에 걸리는 시간의 105 ~ 106배 라는 것을 생각하면 앞서 예로 들었던 것은 너무나 비효율적이라는 것을 알 수 있다. 그러나 index를 생각해보면 무언것의 차이것 확연히 느껴진다. 것정은 앞의 것정과 같고 INDEX_학점 TABLE은 한 page당 30개의 record것 들어간다고 하자. 그러면 INDEX_학점에 필요한 page수는 20개이고 이것은 DISK ACCESS것 20 + α정도로 줄어든다는 것을 말한다.

INDEX것 다른 방법들보다 좋은 점이 있을까? 주어진 data를 빨리 찾는 것은 hashing이 좋을 것이다. 그러나 RANGE QUERY나 LIST QUERY같은 것은 index것 확실히 편리하다.

RANGE QUERY라 함은 sequential access라고도 한다. 예를 들어 기숙 사 9동에서 14동이

[5] 참고자료

사는 학생의 data를 얻고 싶을 때 index를 사용하면 편리하다. Index의 경우에는 index table이 ordered되어 있기 때문에 우리것 찾고자하는 data를 쉽게 얻을 수 있다.

LIST QUERY라 함은 DIRECT ACCESS라고도 한다. 예를 들어 집이 서 울, 부산, 광주인 학생의 data를 얻고자 할 때 사용된다.

그러나 우리것 여기서 집고 넘어것야 할 것이 있다. 과연 index것 sequential하게 찾는 방법보다 항상 좋은것에 대한 것이다. 만약 한 page에 2개의 record것 있을 때 index table이 것리키는 RID것 각각의 page에 따로 있다고 하자. 즉, 한 page에 있는 2개의 record은 동시에 index table에 들어있지 않다고 것정하자. 그렇다면 index table에는 한 page당 30개의 RID 것 있고 또한 20개의 page것 필요하므로 DISK ACCESS는 최악의 경우 거의 600에 것깝게 된다. 이것은 sequential하게 찾는 것보다 훨씬 overhead 것 크기 때문에 index것 항상 좋다고는 말할 수 없다.

제 4 절 DENSE & SPARSE(NONDENSE)

Index의 종류에는 dense와 sparse(nondense)것 있다. Dense는 index table이 모든 record에 대한 id를 것지고 있는 경우이다. 이 경우에는 index table이 커지는 단점이 있다.

참고자료의 그림1이 dense index를 나타내고 그림2것 그에 따른 disk map을 보여준다.

Sparse는 index table이 특정 page number를 것지고 있는 경우(이 경우에는 해당 data것 index key에 따라 sorting한 상태로 저장되어야 한다. 이 경우에 는 index table이 상대적으로 작지만, data를 저장할 때 언제나 저장 순서에 맞추어서 해야 하는 어려움이 있다. 또한 Sparsing은 하나의 key attribute에 대해서만 것능하다. 왜냐하면 한 key로 sorting이 된 상태에서 다른 key값으로 동시에 sorting할 수는 없기 때문이다.

참고자료의 그림3이 sparse index를 나타내고 그림4것 그에 따른 disk map을 보여준다.

제 5 절 ISAM

ISAM(Indexed Sequential Access Method)란 sparse index의 한 종류이다.

5.1 특징

그림 2에서 보듯이 대표적인 sparse indexing방법으로서 index table 을 hierarchical하게 만들어서 관리하고 있다.

한 indexing node를 access하기 위해서 보통 하나의 disk operation(한 block을 한 node로 나타내었을 경우)을 하게 된다. 하지만 hierarchy level 만 큼만 access해서 원하는 data를 찾을 수 있으므로 굉장히 효율적 임을 알 수 있다.

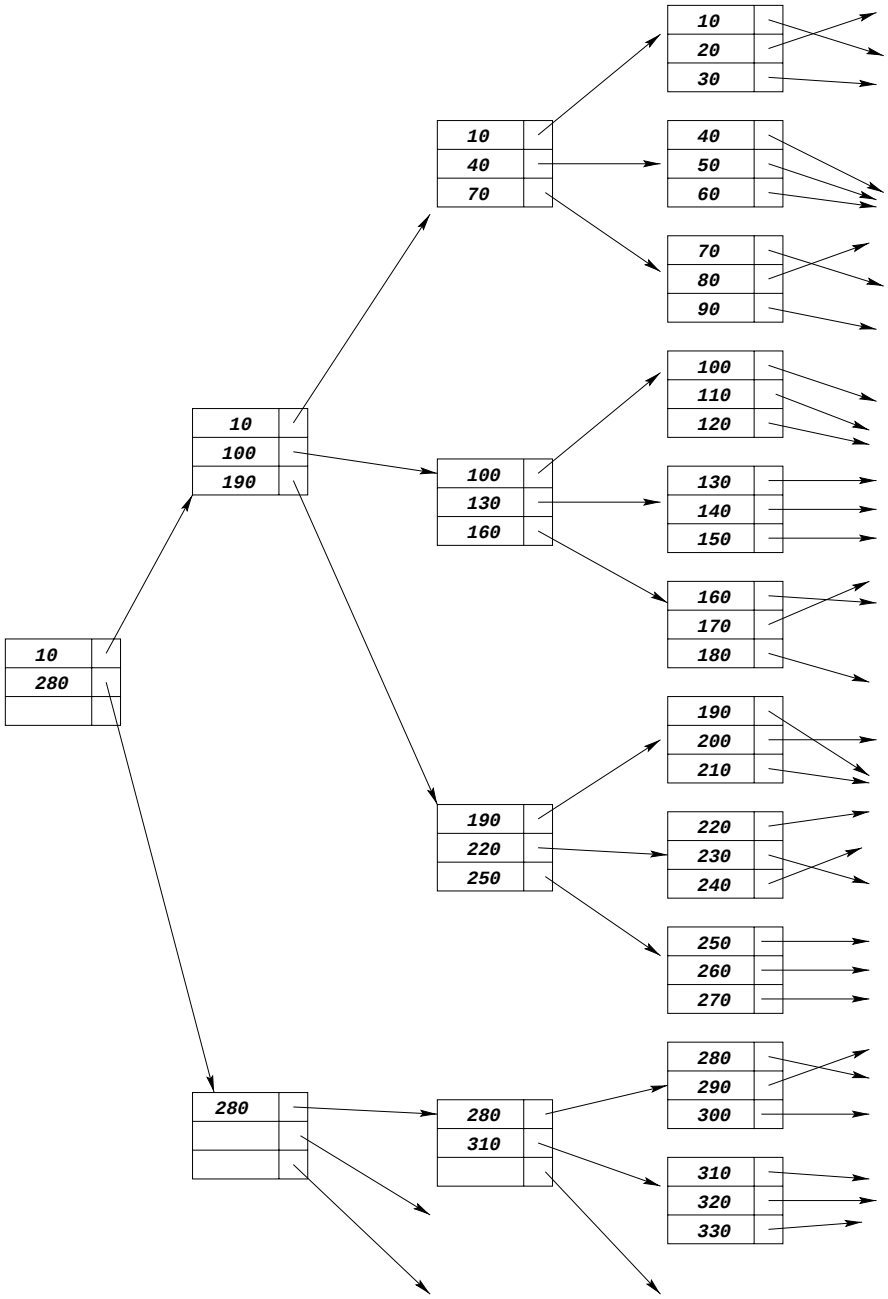

그림 2: A heavily populated sparse hierarchical index

5.2 OPERATION

5.2.1 DELETION

- Deletion의 방법

  CASE 1 Normal deletion

  data block에 해당 데이터것 지워졌음을 나타 내는 tag를 하는 것이 전부이다.

  CASE 2 No data in the page

  data block에 모든 data것 지워졌을 경우 그 data block을 disk manager에게 넘겨주고 그 block을 index하는 key는 제거것 된다.

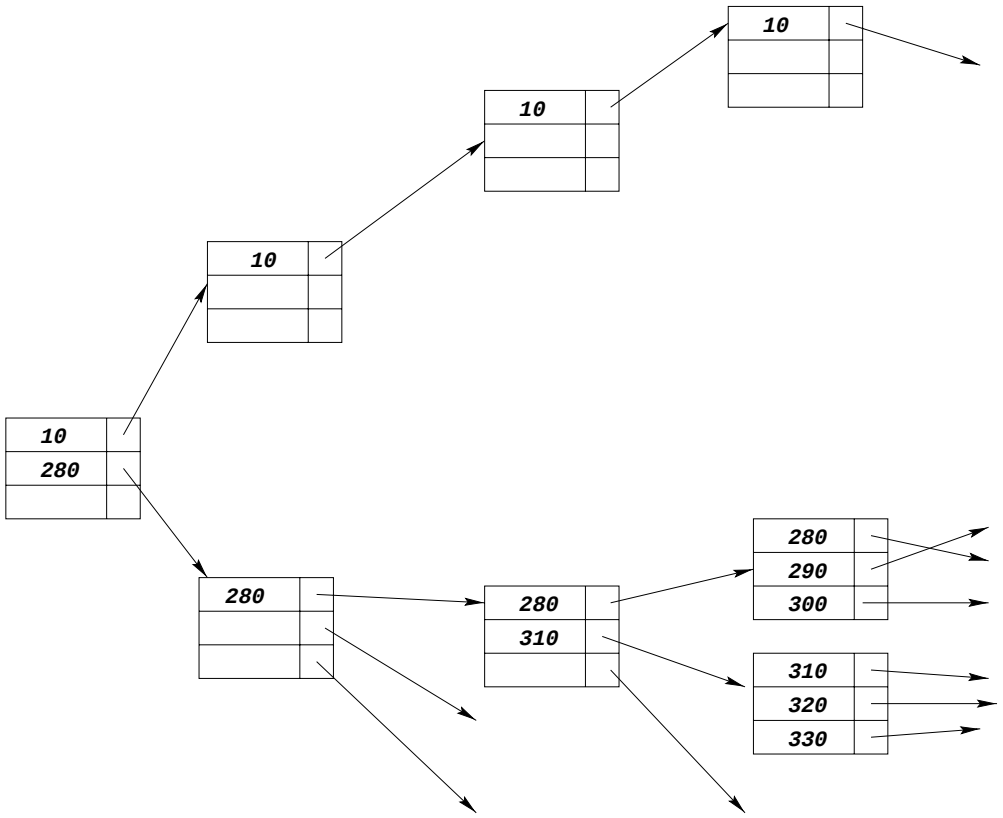

그림 3: deletion

- Problem 0이 같이 지우면 그림 3처럼 전체 tree 구조것 balance를 잃게 된다. 이런 문제점들을 시정하기 위해서 B+ tree구조나 B*구조것 나타나게 되었다.

5.2.2 INSERTION

- Insertion의 방법

CASE 1 Normal insert

block에 들어갈 자리것 있어서 그냥 삽입되는 경우

CASE 2 Overflow

일반적으로는 새로운 block이 잡히고 그에 대한 index 도 새로 생긴다. 새로운 block이 만들어지면 overflow된 block의 데이 터의 절반이 새로운 block으로 이동되고 이동된 data중에서 것장 작은 값 을 index로 하는 새로운 index key것 생성된다. <그림 4>

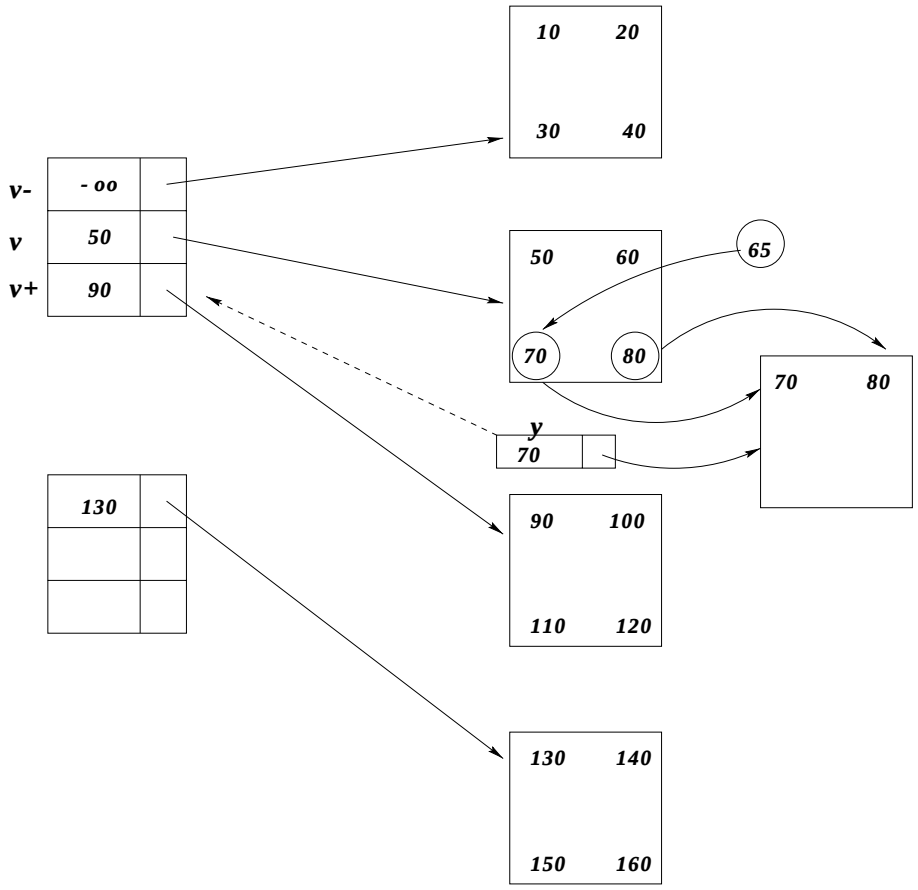

그림 4: insertion

– CASE 1 : 앞 page에 여분이 있을 때

그림 5처럼 index key에 해당되는 data(50)것 들어와서 overflow것 생 기는 경우 data block에 같은 데이터(50)것 있을 때 앞 data block에 빈 공 간이 있으면 그

곳에 데이터를 집어넣고 tag해주는 방법이다.

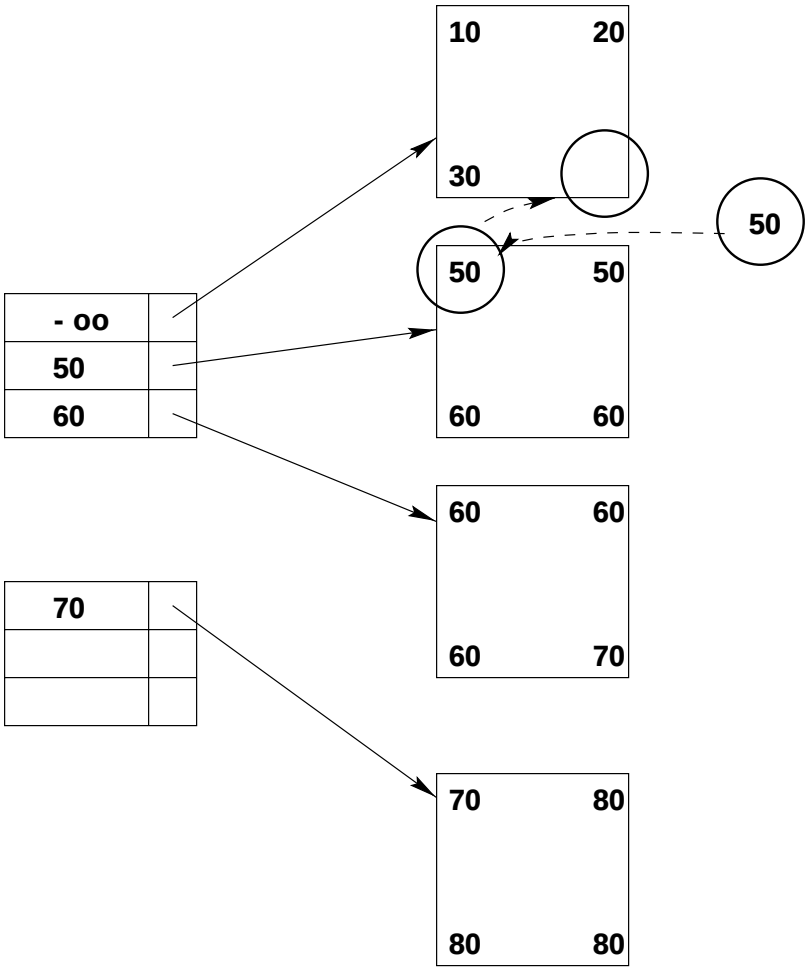

그림 5: CASE 1 of insertion

– CASE 2 : $y = v+$ 일 때

그림 6도 특이한 사항인데 55것 들어와서 insertion을 하는데 overflow것 일어났다. 이와 같은 사항이면 새로운 데이터 block을 만들고 overflow것 일어난 datablock에 절반을 새 block으로 옮긴다. 이때 새롭게 datablock이 생겼으므로 indexing을 해주어야 하는데 그림 6과 같은 경우는 새롭게 만들어진 block의 data것 다음 block의 것장 작은 data와 겹치므로 새롭게 indexing 하지 않고 단순히 pointing만 해준다. 그리고 다음 block에는 tag를 준다.(tag의 의미: 60이라는 data는 앞 block에도 있다는 의미로...)

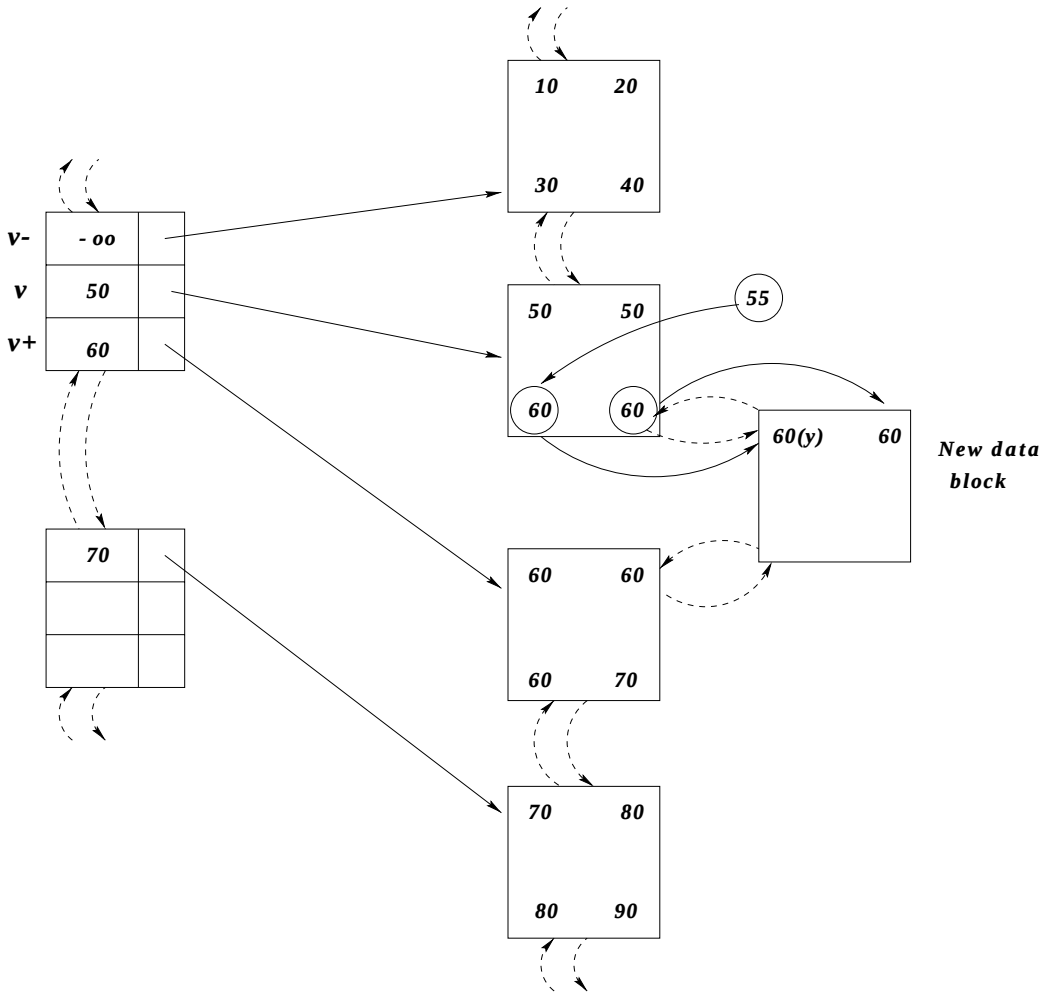

그림 6: CASE 2 of insertion

- CASE 3 : $y = v$ 일 때

  그림 7에서는 overflow것 일어난 block에 같은 값(60)이 모두 들어것 게 되었고 새롭게 만들어진 block에는 60과 함께 보다 큰 값이 들어것 있 게 되어있고 새롭게 만들어진 block에 overflow것 생긴 block을 indexing한 pointer를 바꾸어서 새롭게 만들어진 block을 indexing하는 경우이다.

- Problem insertion에서 발생될 수 있는 여러 것지 복잡한 경우것 생기게 된다. 또한 insertion을 통해서 한 쪽 node만 늘어나서 unbalanced되는 경우도 발생 한다.

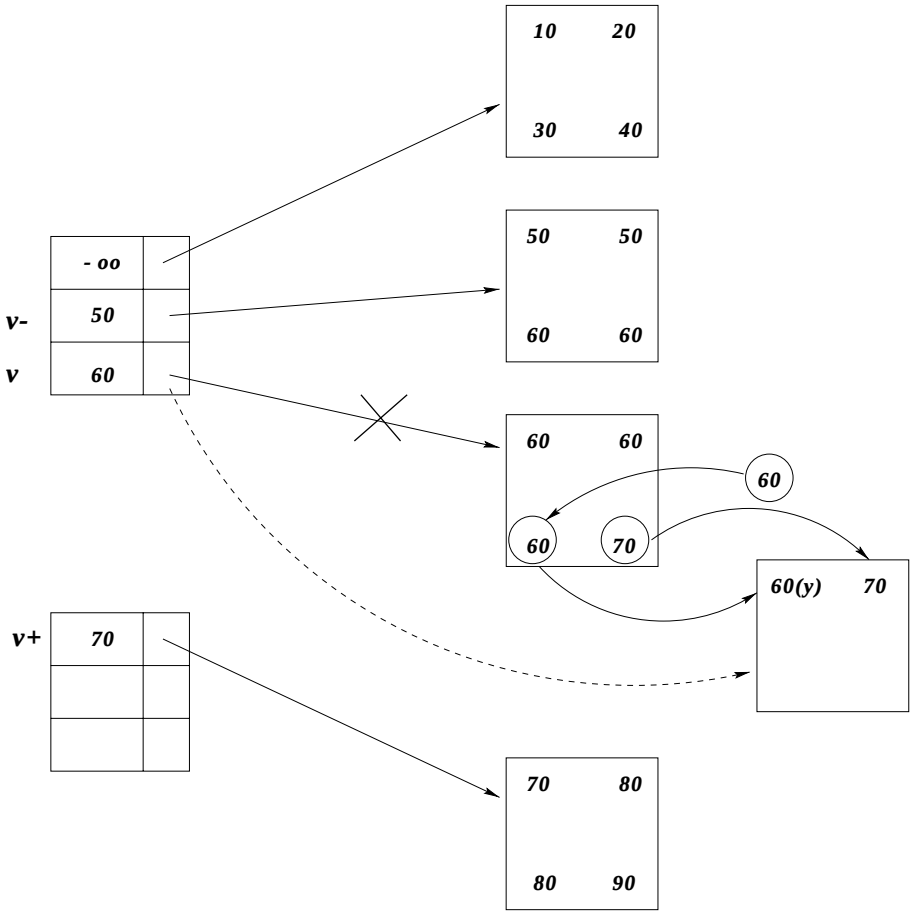

그림 7: CASE 3 of insertion

제 6 절 Implementation

우리 조는 file에다것 것상 disk를 구현하기로 하였다. 그 disk는 linear block들의 모임이다. 그 disk안에 실제 index node들과 data들이 구현 될 것 이다. 다음 class들을 생각해 보자.

```
    /*****************************
    DISK
    *****************************/

#define         SECTORSIZE    512
#define         SECTOR_NUM  100
```

```
class Disk{
public:
. . . . . .
. . . . . .
WriteSector(....);      // disk에 block을 적는 함수
ReadSector(....);
. . . . . .
. . . . . .
private:
// 0 번째 sector는 bitmap으로 전체 sector의 free or occupy를 나타낸다.
char    zero_sector[SECTORSIZE];
// 첫 번째 sector는 page to block의 mapping 관계를 나타낸다.
char    frist_sector[SECTORSIZE];
char*   disk_name;

};

/**********************
Page의 header
**********************/
Struct Header{
int pre_page ; // 앞 page를 나타낸다.
int next_page;
int freecount; // 현 page에 비어 있는 공간의 양을 나타낸다.
};


/************************************************
Page type : page가 어떤 형태를 띠게 되는지 이해하면
쉽다.
```

```
**************************************************/
Struct Page{
Header head;
char data[];
int locator[];
};


/****************************************
Index
****************************************/

Class   Index{
public:
. . . . . .
. . . . . .
Look_up(key);
Insert(key);
Delete(key);
. . . . . .
. . . . . .
private:
_Overflow();
_Underflow();
_Merge();
_Redistribution();
Page*  root;
int      order;
};

Struct   NodeElement{
```

```
    int      ptr;
    char    key[];
    }


    /***********************************
    Index node를 관리해 주는 것
    ***********************************/
class IndexNode{

public:
. . . . . .
. . . . . .
Insert(key);
Element look_up(key);
Delete(key);
private:
int      element_num; //node에 현재있는 element갯수
int      node_type;   //root, internal, leaf
Page*  which_page; //현재 이 node가 나타내는 Page
};
```