File Processing

Programming Project #3

담당 교수 : 박석

학번 : 20161566

이름 : 권형준

목차

0. 프로젝트 필요하다고 생각되는 가정

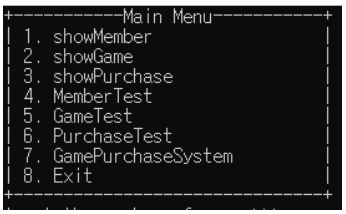
- Visual Studio 2019환경에서 프로젝트를 진행하였다.
- 프로젝트 1, 2의 가정을 모두 포함한다.
- Purchase에 대한 index를 생성할 때 인덱스 키는 문자열이 아닌 하나의 문자를 사용하게 된다. B-tree로 index를 구현하게 됨으로 존재할 수 있는 key의 개수는 2^8개인 최대 256개 밖에 되지 않는다. 또한 모든 char은 키보드로 사용할 수 있는 것이 아니다. 따라서 사용할 수 있는 key의 값은 최대 100개로 제한한다는 조건을 활용하여 key의 값의 범위를 정하였다.

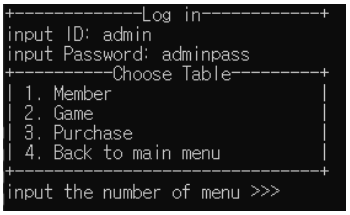
  Key: 24~126, 단(26, 27은 사용하지 않는다)

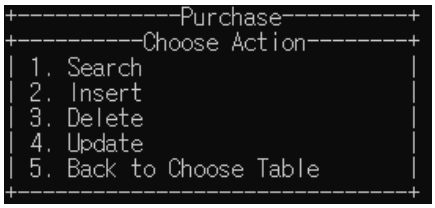
  이외의 char을 사용하는 경우 data파일을 처음 만들 때, txt파일을 읽는 부분에서 오류가 발생할 가능성도 있었기 때문이다.
- B-tree의 file은 한번 닫히면 그 Height를 file에 저장하지 않게 되어 다시 계산하기 어려워진다. 따라서 프로젝트3에 맞게 초기에는 4로 설정하였다.

1. 프로그램 실행 설명

- 실행화면 캡처







- 동작 방법

  이번 프로젝트에서 사용되는 부분에 관한 동작설명만 하자면 다음과 같다. 우선 프로그램을 실행하면 main menu가 <화면1>과 같이 등장한다. 이때 1,2,3은 단순히 text file을 읽어오는 show 함수들을 호출하고, 4,5,6은 text file을 data file로 변형시키는 함수들을 호출한다. 따라서 우선 프로그램을 실행하면 4,5,6은 실행한 뒤 7(Game Purchase System)을 실행하는 것이 안전하다. 이후 Game Purchase System이 실행됨과 동시에 각 data file들의 접근을 위한 index file이 자동으로 생성된다. 이미 존재하는 index file이 있더라도 data file이 외부에서 변형되었을 수 있기 때문에 매번 새로 만들어주는 것이 더 reliability를 높일 수 있다고 생각하였다. Login 화면에서는 user mode와 admin mode 둘 중 하나로 login을 하며(프로젝트3의 경우 admin mode로 로그인을 해야 되기 때문에 위와 같이 아이디와 비밀번호를 입력하면 된다.)이후 접근을 원하는 list들이 등장한다. 여기서 프로젝트3에 해당하는 Purchase를 선택하면 <화면3>과 같은 화면이 나온다. Search의 경우 game id, member id로의 search모두 호환이 되야 되기 때문에 index를 사용하는 접근은 옳지 않다고 생각하였다. 어차피 index를 사용하더라도 모든 record의 탐색이 필요하기 때문이다. 따라서 추가로 수정하지 않고, 단 한 개의 record data만을 찾아서 조작하는 insert, delete, update함수들만 수정하게 되었다. 원하는 기능을 선택하고(2, 3, 4) 이후부터는 insert를 원하는 purchase의 내용, delete를 원하는 purchase의 Id, update를 원하는 purchase의 내용과 ID등을 선택적으로 입력할 수 있다.

2. Purchase List

Purchase list에는 다음과 같은 attribute들이 담겨있다. Purchase의 고유한 key를 뜻하는 purchase ID, 해당하는 구매 내역의 대응하는 game, member를 identify하는 game ID, Member ID그리고 구매한 날짜, 마지막으로 index에서 사용되는 key. Purchase list는 member와 game list에 대한 종속성을 가지고 있다. Game이 삭제되거나 member가 삭제되면 대응하는 구매 내역도 삭제돼야 된다. 또한 새로운 구매가 추가될 때, 함부로 game, member ID를 추가하면 안 되고 기존의 game, member list를 탐색하여 존재하는지 확인한 이후 추가할 수 있다.

기존의 프로젝트 조건은 purchase ID가 character 12개로 이루어져 있다고 하였지만 이번 프로젝트에서 index를 구분하는 key값을 character하나로 한다고 하였기 때문에 character 12개의 중 가장 처음 character를 unique하게 설정하여 key로 사용하였다. Txt, dat, 파일 등에서 읽고 쓰는데 문제가 발생하기 때문에 사용하는 character의 범위도 설정하였다. 따라서 위에 프로젝트 가정에 해당하는 key값들만 삭제하고 추가할 수 있다.

3. B-tree

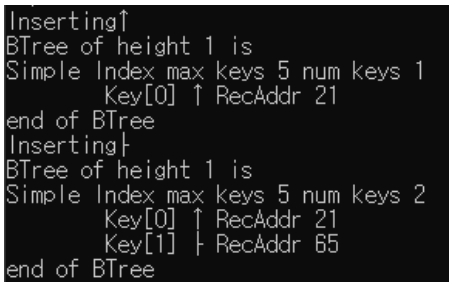
우선 프로젝트의 조건대로 index에 대한 purchase list index를 생성하는데, b-tree index를 사용하였다. 프로젝트2에서 사용한 index와 마찬가지로 검색을 b-tree로 사용하여 해당하는 위치에 삽입, 삭제, 수정을 진행하였다. 한가지 다른 점은 프로젝트2에서의 index file안에는(TextIndexedFile) 대응하는 datafile도 포함되어 있었기 때문에 바로 수정이 가능했지만, b-tree는 2단계의 수정이 필요하였다. 한번은 b-tree에서의 수정, 한번은 대응하는 record내부에서 수정이 필요하였다. BTree구조체 내부에 프로젝트2처첨 datafile을 추가하는 방법이 조금 더 효율을 높일 수 있을 것 같지만, 이는 b-tree index를 사용하는 기본적인 concept를 해친다고 생각하였다. (파일의 크기가 커질 경우) 따라서 B-tree를 통해 search를 빠르게 할 뿐 record의 삽입과 삭제는 이전과 같은 방식으로 이루어졌다.

① 생성

- 설명

  새로운 B-tree를 생성하는 것은 교재에 나온 것처럼 bottom-up으로 성장한다. 예를 들어서 차수가 4인 node에 key가 5개가 들어올 경우 split을 한 뒤 새로운 상위 노드가 생기는 것이다. B-tree를 create할 때, 기존의 data file을 하나씩 읽어서 차례대로 insert하는 과정을 거치면 자동으로 생성된다. 코드를 살펴보면 Btree는 Btree node를 사용하고 있으며, Btree node는 simple index를 상속하여 사용하고 있다. 따라서 simple index class에 구현되어 있는 search등의 method를 그대로 상속하여 사용하는 것을 살펴볼 수 있다. 생성하는 과정은 Insert함수를 반복하여 사용하는 것이기 때문에 뒤에 삽입에서 추가로 설명하겠다.

- 화면 캡처



Data file의 data를 하나씩 읽어오면서 insert되는 시작 과정을 Print method를 사용하여 표현한 것이다. index에는 최대 4개의 key가 들어갈 수 있다. Key[0]에 들어가는 char값은 24에 해당하는 ASCII코드이다. 그 다음 들어오는 값은 25로 Key[0]보다 뒤에 삽입되는 것을 볼 수 있다. 이러다가 하나의 node가 최대 key값을 초과하게 되면 다음과 같이 분리된다.

```
InsertingL
BTree of height 2 is
Simple Index max keys 5 num keys 2
        Key[0] L RecAddr 110
        Key[1] ~ RecAddr 86
Node at level 2 address 110 Simple Index max keys 5 num keys 3
        Key[0] ↑ RecAddr 21
        Key[1] ├ RecAddr 65
        Key[2] L RecAddr 202
Node at level 2 address 86 Simple Index max keys 5 num keys 2
        Key[0] } RecAddr 113
        Key[1] ~ RecAddr 160
end of BTree
```

L이라는 key를 입력하게 되면 level 1에 key가 5개가 되어 두개의 node로 분할되고 상위에 node가 새로 생겨 level 2가 된다. Level 1의 key[0], key[1]이 level 2의 각 node의 최대 값인 key[2]와 key[1]에 해당함을 알 수 있다. 이런 식으로 모든 data들이 입력된다.

- 소스코드 캡처

```
BTree<char> bt(BTreeSize);
result = bt.Create(index_filename, ios::in | ios::out | ios::trunc);
if (!result) {
    cout << "Please delete fileOfPurchase.ind" << endl;
    return;
}
PurchaseFile.Open(data_filename, ios::in);

while (1) {
    Purchase p;
    int recaddr = PurchaseFile.Read(p);
    //cout << "Inserting" << p.Key() << endl;
    if (recaddr == -1)
        break;
    result = bt.Insert(p.Key(), recaddr);
    //bt.Print(cout);
}
```

Insert method를 반복적으로 사용하여 data file의 key값들을 삽입하는 것을 볼 수 있다. 위에 결과를 출력하기 위한 Print는 주석처리를 했다. Insert의 경우 밑에서 더 자세하게 설명할 것임으로 여기에는 추가하지 않았다. 이렇게 사용할 수 있는 것은 Purchase member에 Key()라는 method를 추가하였으며 이는 character하나로 구성되어 있고 unique함이 이미 보장되어 있기 때문이다.

- 결과 화면

B-tree의 최악의 깊이와 최선의 깊이는 계산할 수 있다. 만약 차수가 4인 B-tree에서 총 100개의 data를 넣게 된다면 최선의 경우 모든 tree node가 꽉 차 있을 것임으로 4level이면 100개의 data를 단말 node들에 모두 포함시킬 수 있다. 하지만 각 node가 반만 차 있는 경우, 즉 2개밖에 없는 경우, 7Level까지 깊어질 수 있다. 이때 사용한 data file이 key를 기준으로 거의 정렬되어 있기 때문에 또한 split이 발생하면 한쪽에는 3개, 다른 쪽에는 2개의 key값을 가지고 있기 때문에 하나의 node에 평균적으로 3개의 key를 담고 있어서 4~5의 Level이 형성될 것이다. Tree가 형성된 이후 결과를 살펴보면 최대 깊이가 4인 B-tree가 생성된 것을 알 수 있다.

```
end of level 3
Node at level 2 address 1190 Simple Index max keys 5 num keys 2
        Key[0] s RecAddr 1070
        Key[1] ~ RecAddr 1166
Node at level 3 address 1070 Simple Index max keys 5 num keys 3
        Key[0] m RecAddr 1022
        Key[1] p RecAddr 1046
        Key[2] s RecAddr 1094
Node at level 4 address 1022 Simple Index max keys 5 num keys 3
        Key[0] k RecAddr 3730
        Key[1] l RecAddr 3779
        Key[2] m RecAddr 3821
Node at level 4 address 1046 Simple Index max keys 5 num keys 3
        Key[0] n RecAddr 3867
        Key[1] o RecAddr 3911
        Key[2] p RecAddr 3953
Node at level 4 address 1094 Simple Index max keys 5 num keys 3
        Key[0] q RecAddr 3997
        Key[1] r RecAddr 4040
        Key[2] s RecAddr 4084
end of level 4
Node at level 3 address 1166 Simple Index max keys 5 num keys 3
        Key[0] v RecAddr 1118
        Key[1] y RecAddr 1142
        Key[2] ~ RecAddr 1214
Node at level 4 address 1118 Simple Index max keys 5 num keys 3
        Key[0] t RecAddr 4130
        Key[1] u RecAddr 4179
        Key[2] v RecAddr 4227
Node at level 4 address 1142 Simple Index max keys 5 num keys 3
        Key[0] w RecAddr 4274
        Key[1] x RecAddr 4317
        Key[2] y RecAddr 4362
Node at level 4 address 1214 Simple Index max keys 5 num keys 4
        Key[0] z RecAddr 4407
        Key[1] { RecAddr 4453
        Key[2] } RecAddr 113
        Key[3] ~ RecAddr 160
end of level 4
end of level 3
```

Root node이후부터 오는 각 Node의 Print는 simple index의 print가 아닌 B-tree node의 print method가 호출되어 node가 몇 Level에 있는지, B-Tree file의 어느 주소에 있는지도 출력해준다.

- 파일 내용 캡처

```
00001080  31 35 7C 00 2C 00 76 30  30 30 30 30 30 30 30 30  15|.,.v0000000000
00001090  30 30 7C 37 35 31 30 37  38 32 38 7C 69 6B 6E 6F  00|7510782 8|ikno
000010a0  6F 6D 31 31 30 37 7C 32  30 32 30 2F 30 37 2F 31  om1107|2020/07/1
000010b0  36 7C 00 28 00 77 30 30  30 30 30 30 30 30 30 30  6|.(.w0000000000
```

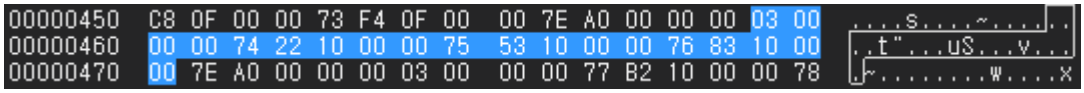
바로 위에서 결과화면에 나온 Index를 가지고 index file이 제대로 형성되었는지 살펴보자. 우선 v라는 값의 record address는 <화면 1>에서 볼 수 있듯이 1083이다. 이 값이 index file에 제대로 저장되어 있는지 확인해야 된다. 우선 level 2의 node 1190(4a6)이라는 주소로 가면

```
000004a0  00 7E A0 00 00 00 02 00  00 00 73 2E 04 00 00 7E  .~....[....s....~
000004b0  8E 04 00 00 6A CE 03 00  00 7E A0 00 00 00 04 00  ....j....~......
```

<화면 2>에서 결과를 확인할 수 있다. 처음 4byte는 node의 수를 표시하였고 다음 1byte는 key 값인 's', '~'를 표시하고 뒤에 각각 4byte씩 다음 주소를 가리키는 pointer가 저장되어 있다. 우리가 찾는 것은 v임으로 s보다 크고 ~보다 작은 값임으로 '~'가 가리키는 pointer 1166(48e)로 이동하면 다음과 같은 <화면 3>을 찾을 수 있다.

```
00000480  DD 10 00 00 79 0A 11 00  00 7E A0 00 00 00 03 00  ....y....~......
00000490  00 00 76 5E 04 00 00 79  76 04 00 00 7E BE 04 00  ..v^...yv...~...
000004a0  00 7E A0 00 00 00 02 00  00 00 73 2E 04 00 00 7E  .~........s....~
```

아직 level이 3이기 때문에 한 단계 더 들어가야 되는 것을 알 수 있으며, 우리가 찾는 값은 'v' 임으로 v보다 작거나 같은 key들을 가리키고 있는 1118(45e)로 이동한다.



드디어 Level 4에 도달하였다. 이제 v에 해당하는 주소는 index file에서 다음 node로의 주소가 아닌 record file로의 주소임을 알 수 있고. 1083으로 우리가 찾던 주소가 올바르게 저장된 것을 알 수 있다. B-tree 구조체 내부에 level을 항상 저장하고 있어, 찾은 주소가 다른 node로의 주소인지 record data로의 주소인지 항상 확인해야 된다.

이제 올바르게 B-tree가 생성되었다는 것을 알 수 있다. 그렇다면 생성된 B-tree를 가지고 삽입, 삭제, 수정은 어떻게 이루어지는지 알아보자.

② 삽입(insert)

- 우선 삽입하고자 하는 Purchase의 내용을 읽어야 된다. 입력 받은 Game ID, Member ID가 존재하는 ID이며, 입력 받은 Purchase ID는 존재하지 않는 ID라면 입력을 시작하게 된다. 앞의 조건을 만족한다면 data file에 추가되고 뒤따라서 B-tree에도 삽입된다. 이때, B-tree를 생성할 때 사용하였던 함수인 insert 함수를 그대로 사용한다. Insert함수는 <btree.hpp 65~116>까지의 함수이다. 우선 key에 해당하는 leaf node를 찾는다. 이는 simple index에서 사용하는 search함수를 사용한다. FindLeaf함수는 <btree.hpp 232~243>까지며, level=1부터 Height까지 Search를 사용하여 key에 해당하는 leaf를 찾는다. 새롭게 삽입된 key가 node에서 새로운 최대값이 되었다면 이는 상위 Node에 전파가 되어야 된다. B-tree에서 각각의 자식 node의 최대값을 부모가 가지고 있기 때문이다. 이를 전파해주는 과정을 거치면 overflow가 일어났는지 확인한다. 이 부분이 86번째 줄~102번째 줄 사이의 while문을 통해서 이루어진다. 우선 초과한 node를 split한다. 다음 split하여 생성된 2개의 node를 삽입한 뒤, 부모의 node에 이를 넣어준다. 만약 부모도 꽉 차게 된다면 thisnode의 자리에 parentnode를 넣어 부모도 같은 과정을 반복하게 된다. 마지막으로 level이 -1까지 올라갔다면, 즉 root까지 overflow가 발생했다면 root도 split을 하게 되는데. Root는 이미 while문 안에서 split이 되었기 때문에 단순히 새로운 node를 생성하여 이의 index로 넣어주면 완료되는 것이다.
- Insert가 정상적으로 이루어졌는지에 대한 결과는 위에 생성에서 보였기 때문에 생략한다.

③ 삭제(delete, 9장 13번 연습문제)

- 우선 삭제하고자 하는 Purchase에 대응하는 ID가 이미 있는지 찾아야 된다. 존재한다면 datafile에서 삭제를 하고, B-tree에서도 삭제한다. 우선 datafile에서 삭제하는 것은 이전에 구현하였다. 하지만 탐색을 하는 과정을 index를 사용하면 더 빠를 것이기 때문에 해당하는 key의 address만 index를 사용하여 접근하도록 구현하였다. 다음 index file에서 삭제를 하는 것을 구현해야 된다. 이는 9장 13번 연습문제에 해당하는 Delete method를 구현하는 것인데, 여기서는 Remove method로 대체하여 구현하였다. Insert와 비슷하게 해당하는 key의 leaf node를 찾아서 하나를 삭제하고 under flow가 발생하면 merge를 하면서 root까지 갔다가, root에 자식이 하나밖에 남지 않는다면 root의 자식을 새로운 root로 만드는 방식이다. 이러한 과정을 <btree.hpp 118~227>까지 구현을 했다. 이 코드는 기존에 주어진 것이 아니기 때문에 더 자세하게 설명을 해보면 다음과 같다.

- Insert와 비슷하게 key가 담겨있는 node를 FindLeaf를 사용하여 찾는다. 해당하는 leaf node의 가장 큰 값을 저장해 놓는다. 이 값이 만약 삭제하려는 값과 같다면, 삭제한 이후에 이 값이 다른 값으로 대체되어야 되기 때문이다. 이후 삭제를 실행한다. 삭제가 완료된 이후 underflow가 발생하지 않는다면 정상적임으로 insert와 마찬가지로 기존의 최대 키를 새로운 최대 키로 바꿔주는 과정만 처리하면 된다.

- 하지만 문제는 under flow가 발생하면서부터 시작된다. Insert는 단순히 하나의 node를 두개로 나눠주면 끝이었지만, underflow는 merge를 같이 할 sibling이 존재하는 지부터 확인해야 된다. 또한 merge를 할 수 있는 sibling은 오직 바로 옆에 붙은 sibling 밖에 되지 않으며, 해당 sibling과 merge시에 overflow가 또 발생할 수 있음을 주의해야 된다. 우선 현재 node thisNode가 parent의 몇 번째 자식인지 확인한다. 이때 k번째 자식이라고 하면(이를 구하는 부분은 <btree.hpp 155~160), 바로 앞과 뒤의 자식을 비교하여 merge를 시도해본다. 이때 merge가 되었다면 이제 부모에게서 사라진 하나의 자식을 가리키던 key를 삭제해주면 된다. 이는 206~215줄 사이에 구현을 해 놓았다. 오른쪽과 왼쪽 둘 중 어느 sibling과 merge하느냐에 따라서 삭제해야 되는 parent의 key값은 달라진다.

- 이러한 과정을 insert와 마찬가지로 root까지 반복해야 됨으로 마지막에 thisNode=parentNode를 하여 전파시킨다.

- 이제 merge하였을 때 overflow가 발생하는 경우를 살펴보자. Overflow가 발생하는 경우에는 다행히 부모의 key가 변하지 않아 전파는 하지 않아도 된다. 2개의 merge는 오직 2개로만 분리된다. 따라서 이때도 앞의 자식과 merge할지, 뒤의 자식과 merge할지 정한 뒤, 한쪽에 모든 key를 몰아넣은 뒤 insert에서 한 것처럼 split을 실행한다. 다음 split한 2개의 node에 바뀐 최대 key값을 parent에서 수정해주면서 완료된다.

④ 수정(update)

- 수정의 경우, key의 값이 수정된다면 그것은 완전히 다른 record가 되는 것임으로 삭제 후 삽입의 절차가 수행되어야 된다. 다른 말로는 key의 수정은 불가능하다. 하지만 그 외의 attribute의 수정이 발생하여 record의 길이가 달라져 시작 주소도 달라지게 된다면 이는 index file의 결과에 적용되어야 한다. 하지만 이때, 시작 주소의 field 크기는 정해져 있기 때문에 수정은 단순히 search이후 가리키는 address의 위치만 바꿔주면 되는 것이다. 하지만 이를 구현하기 위하여 추가 method를 구현하여야 되기 때문에 그냥 Remove이후 Insert를 호출해주는 방식을 선택한다.

4. B-tree를 통한 data record 접근

B-tree에서 key를 사용하여 data record의 주소를 찾는 방법은 B-tree생성에서 예시를 통하여 설명하였다. key값을 level1부터 비교하여 다음 node로, 다음 node로 옮겨 가서 마지막 level의 node에서 key 값에 대응하는 주소가 바로 data record의 주소이다. 없는 경우, 해당 data-record는 존재하지 않는다고 할 수 있다. 그렇다면 여기서는 조금 더 자세하게 코드의 관점에서 그 과정을 설명해보자. 이는 삭제하기 위해 해당 record의 주소를 search하는 부분을 살펴본다.

➔ 우선 main.cpp의 993번째 줄을 보면 BTree 파일을 오픈하여 구조체에 저장한다. 이제 BTree에서 해당하는 key의 주소를 탐색하게 된다.

➔ Bt.Search를 따라가면 btree.hpp의 236번째 줄로 가게 된다. 이 함수 안에는 FindLeaf와 다시 한번

search함수가 호출이 된다. 우선 FindLeaf를 따라가면

➔ 같은 파일(btree.hpp)의 275번째 줄로 가게 된다. 이는 해당하는 key를 담고 있는 leaf Node를 찾는 동시에 그 길을 Node라는 배열에 저장하는 것이다. 이때 사용하는 Search라는 method는 Btnode구조체 안에 있다. Fetch라는 함수도 사용하는데 우선 Search부터 들어가보자.

➔ btnode구조체는 simple index를 상속하여 사용한다. 따라서 Search method를 따로 선언하지 않아도 simple index안에 미리 구현되어 있는 method를 사용하면 된다. 이는 simpind.hpp의 63번째 줄에 가면 있다. Key에 해당하는 index를 찾아서 반환하는 것이다. 즉, 하나의 node안에 담고 있는 자식들 중 key가 가리키는 자식을 찾는 것이다. Find함수를 간단하게 살펴보면 key를 기준으로 해당 node에 담긴 key들을 비교하는 과정을 거친다. 이때, exact 비교와 inexact 비교 둘 다 가능한 flag도 담고 있다. 이는 같은 파일(simpin.hpp의 82번째 줄에 있다)

➔ 이제 다시 FindLeaf로 돌아가서 Fetch를 알아보자. Fetch는 btree.hpp의 297번째 줄에 있다. 해당하는 주소에 저장되어 있는 btree node를 읽어온다. 모든 btree의 크기는 같기 때문에 정확하게 읽어올 수 있다.

➔ 이제 key를 담고 있는 leaf node와 root부터 이어지는 모든 node가 저장되어 있따. 이를 가지고 Search method를 다시 호출한다. 이 search method는 이전에 simple index안에 구현되어 있는 것이다. Leaf node에서 search를 하게 되면 node안에서 key에 해당하는 주소, 즉 실제 데이터 레코드의 주소를 반환한다.

➔ 마지막으로 이 값을 가지고 data file을 연 다음 해당 주소로 direct access가 가능하다. 이전에는 처음부터 순차 탐색을 하던 것에 비하여 속도가 훨씬 증가한 것을 알 수 있다.

➔ 이러한 자세한 과정은 위에 data, index 파일의 그림을 통해 확인할 수 있다.

5. Simple Prefix B⁺-tree

앞에서 B-tree를 구현하였고 이제는 simple prefix B⁺-tree를 구현할 차례이다. 이 둘의 차이점은 무엇일까? 앞에서 사용한 B-tree와 같은 형태를 index로 가지면서 record들이 하나의 큰 block에 담겨서 연결되는 형태이다. 즉, B-tree는 마지막 leaf node에 연결된 것들이 각 data record였지만, 여기서는 block에 대한 pointer이며, 각 block의 내부의 record들은 정렬된 상태를 유지한다. 또한 각 block들은 서로 연결되어 있다. B⁺-tree는 이렇게 유지함으로써 임의 접근과 순차 접근 모두에게 유리한 자료구조를 유지하게 된다. 또한 차수와 key의 수가 같았던 B-tree와는 다르게 key의 값이 하나 줄어들게 된다. 기존의 B-tree class를 사용하여 B⁺-tree를 비슷하게 구현할 수 있지만, 완전한 B⁺-tree를 만들기 위해서는 새로운 class를 정의해야 된다. 만약 기존의 B-tree class를 사용하여 구현하려고 한다면 다음과 같이 차이점들을 극복할 수 있을 것이다.

➔ 임의로 block의 크기를 정한다. 우리가 사용하는 것은 variable length record임으로 해당 block에 들어가 있는 record의 수는 상관없이 크기로 설정한다. 임의로 설정한 block의 시작 주소를 B⁺-tree leaf node가 가리키게 된다.

➔ 기존에 사용하였던 key들은 그대로 사용하여도 된다. character하나만을 사용하였기 때문에 그것 만으로도 prefix이며 separator가 될 수 있다.

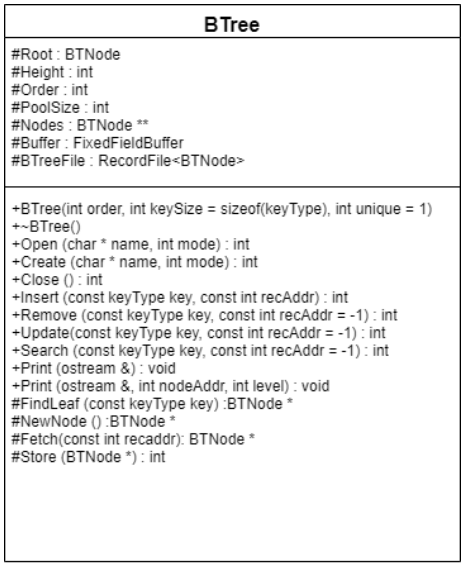
➔ Simple index class에서 key의 수를 조절해야 된다. 기존에는 key의 값과 자식의 수가 같아서 search,

find등의 함수에서 해당하는 key값을 찾는 방식이 달랐다. 하지만 이제는 key의 값이 하나 줄어들기 때문에 마지막 pointer의 경우 최대 key와 비교하여 "그 값보다 크다면" 다음 자식 node로 연결한다.

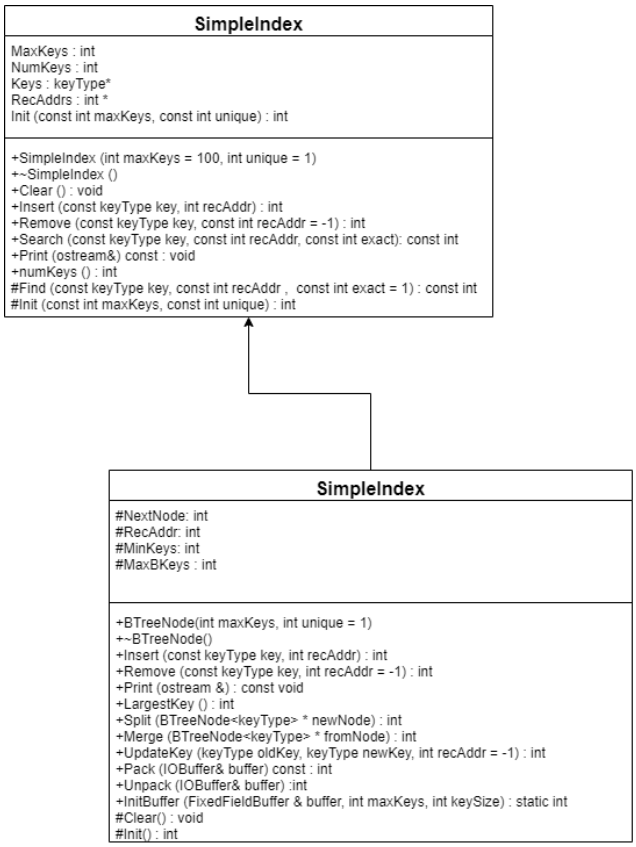
위의 조건들을 어느정도 절충하여 data file을 순차적으로 유지하면 어느정도 B^+-tree와 유사한 구조가 나옴을 알 수 있다. 순차적으로 정렬되어 있기 때문에 block단위 뿐만 아니라 전체 단위로도 정렬이 되어 있게 된다. 이는 프로젝트3에서 전체 입력의 수가 100개로 제한되기 때문에 가능한 것이며 이 점이 바뀌게 된다면 매우 비효율적이고 구현하기 힘들게 될 것이다. 따라서 수정할 것은, 삽입, 수정, 삭제가 발생하게 될 경우 fileOfPurchase를 처음부터 다시 써서 항상 정렬되게 유지하는 것이다.

6. (새로) 구현한 클래스의 다이어그램

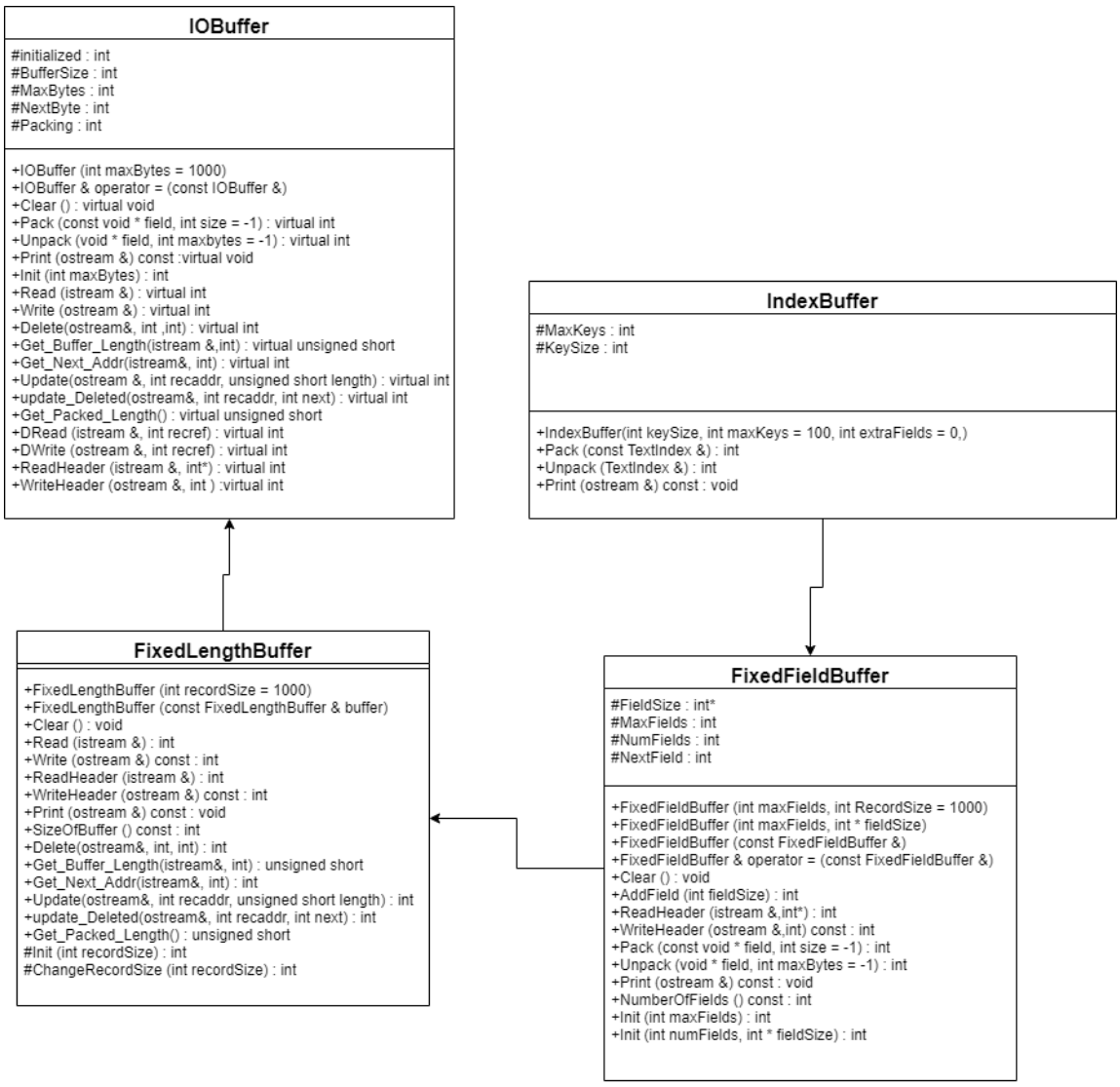
- Btree



- btreeNode, SimpleIndex



- IndexBuffer



7. (새로) 사용된 소스, 헤더 파일

- Btree.h, btree.hpp

  Btree 하나에 대응하는 구조체, 안에 여러 개의 btree node등 btree에 대한 attribute 포함

- Btnode.h, Btnode.hpp

  Btree node한 개에 대응하는 구조체, 안에 저장된 key와 recaddress등을 포함

- Simpind.h, simpind.hpp

  Btree node가 상속하는 부모 class, maxkeys, numkeys, search method등을 제공