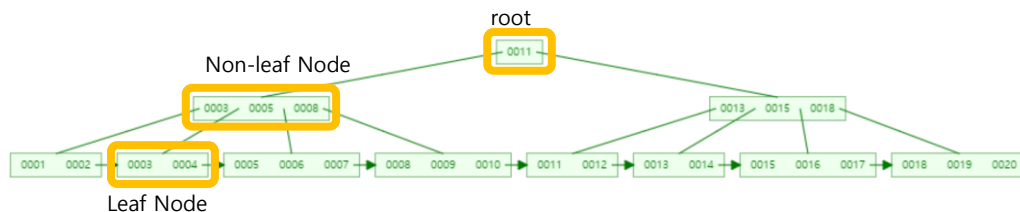


Summary of B+tree Assignment

컴퓨터소프트웨어학부 이해민

Summary of algorithm



datafile에서 input된 정보에 따라 key-value를 b+tree에 insert하거나 key(와 그에 따른 value를)를 b+tree에서 삭제한다. 업데이트된 b+tree 정보는 indexfile에 저장된다.

(1) insertion

삽입할 key와 value 값을 받는다.

해당하는 key가 들어가면 되는 위치의 leaf node를 탐색한다.

삽입할 leaf node의 현재 키 개수가 최대 키 개수를 넘지 않으면 바로 삽입해준다. 이때 노드의 key-value쌍은 오름차순으로 정렬된다.

삽입할 leaf node의 현재 키 개수가 최대 키인 경우 노드를 분할해준다. 중간 값을 기준으로 분할하면 되는데, degree가 홀수일 경우 $\text{ceil}(\text{degree} / 2) - 1$, 짝수일 경우 $\text{degree} / 2$ 번째 키를 기준으로 분할해준다. 중간 값은 parent node로 올려준다.

만약 parent node에 child의 중간 값을 넣었을 때 최대 키 개수를 초과하면 parent node도 분할한다. 이때 parent node가 root라면 parent node를 분할하고 위로 올린 중간 값을 가진 새로운 노드를 만들고 이를 root로 설정한다. root가 아니라면 parent에 중간 값을 삽입해도 최대 키를 넘지 않을 때까지 계속해서 split 해준다. 재귀적으로 실행된다.

(2) deletion

삭제할 key를 받아서 탐색한다. 트리가 비었거나 트리에 해당하는 노드가 없는 경우 예외 처리된다.

삭제할 key가 트리에 존재하고 키가 존재하는 노드의 현재 키 개수가 최소 키 개수를 초과할 경우, 바로 삭제해준다. internal node에서도 해당 키가 존재한다면 삭제해준다.

키가 존재하는 노드의 현재 키 개수가 최소 키 개수일 때, 형제 노드에서 빌려올 수 있는지 확인해본다. 빌려올 수 있다면 형제 노드에서 키 하나를 빼내서 키가 존재했던 노드에

넣어준다.

형제 노드에서도 빌려올 수 없다면 형제 노드와 merge 한다. merge하고나서 parent node의 키 개수가 최소 키 개수 미만일 경우 parent node의 형제 노드에서 키를 빌려올 수 있는지 확인해보고 빌려올 수 있으면 빌려오고 못 빌려온다면 형제 노드와 merge한 뒤 다시 부모를 확인하는 재귀적인 과정을 거친다.

최종적으로 root에 도달하는 경우, root에 key가 2개 이상이라면 root에서 key 하나를 가져오고, root에도 key가 1개만 존재하는 경우 병합한 root의 child node가 새로운 root가 되며 전체적으로 트리의 height이 하나 낮아지게 된다.

(3) single key search

search할 key를 받는다. root 부터 시작해서, 탐색하는 도중의 노드의 키 값과 key를 비교한다. 예를 들어 노드의 i번째 키 값보다 같거나 크다면 i + 1번째 child node로 이동하고, 작으면 i번째 child node로 내려가면서 탐색한다. 모든 노드를 탐색하지 않으므로 약 $\log n$ 정도의 효율을 보인다.

(4) range search

range search할 key1와 key2를 받아서 key1이상 key2 이하에 해당하는 key-value 쌍을 모두 출력한다. key1을 single key search 한 다음 해당하는 leaf node에서부터 다음 노드로 이동한다. b+tree의 경우 leaf 노드가 linked list로 이어져있기 때문에 range search가 쉽다. key2에 도달할 때까지 계속해서 오른쪽 leaf node로 이동한다.

1. Main.java

- main 함수가 위치한 클래스로, command line argument를 받는다.
- 명령어에 따른 동작을 수행한다. 동작을 수행한 뒤 필요하다면 indexfile과 datafile에 변경 내용을 업데이트한다.
- insertion시: datafile에서 key-value pair를 읽어서 b+tree에 insert한다. b+tree는 indexfile에 update된다.
- delete시: datafile에서 key를 읽어서 해당되는 key-value 쌍을 b+tree로부터 삭제한다. b+tree는 indexfile에 update된다.
- search: single key를 검색할 경우, key를 받아서 해당하는 key가 있다면 그에 따른 value를, 없다면 NOT FOUND를 출력한다. range를 search할 경우, 해당하는 범위에 key가 있다면 start_key와 end_key를 포함한 key-value쌍을 순서대로 출력한다. leaf node들이 linked list로 연결되어 있으므로 해당하는 start_key와 end_key가 있는 리프 노드만을 각각 한번씩 탐색한

뒤 순차적으로 linked list를 따라 이동한다. 각 key의 대소가 역전되거나 해당 범위에 key가 없으면 NOT FOUND를 출력한다.

- 빈 노드를 새로 만드는 경우를 제외하고는 이전에 만든 노드를 불러와야 하므로 Main.java의 readTree에서 index.dat파일을 읽는다.

2. Node.java

- b+tree를 구성하는 node를 만드는 클래스
- 멤버 변수
 - leaf인지 아닌지 판별하는 boolean 변수
 - 현재 노드에 저장된 key개수
 - key를 저장하는 key배열
 - value를 저장하는 value배열
 - 자식 노드(혹은 leaf인 경우 오른쪽 노드)를 저장하는 Node 배열
- 멤버변수에 대한 getter와 setter, key-value 혹은 key를 넣으면 노드 내에 순서대로 배열해주는 push_back 함수, 현재 node의 key를 모두 보여주는 함수가 있다.

3. BPlusTree.java

- 코드 상에서 B+tree에 대한 정보가 처리 및 저장되는 클래스
- singleKeySearch, rangeSearch, insert, delete, saveTree가 public으로 구현되어 있다.

4. indexfile, datafile

- indexfile: datafile에서 불러온 정보를 BPlusTree에서 정리하여 내보내면 그 정보를 저장하는 파일 (index.dat)
- datafile: 삽입하거나 삭제하려는 정보가 담겨진 파일 (input.csv 혹은 delete.csv)

Detailed description of codes (for each function)

Main.java

1. `main(String[] args)`: command line argument를 받아서 명령을 수행한다. (명령어는 instructions for compiling source에 설명되어 있습니다!)
2. `readTree(String indexFile)`: index_file에 저장되어 있는 tree를 읽어온다. degree만

저장되어 있다면 빈 트리를 반환한다. 트리 구조 자체를 저장(BPlusTree.java의 saveTree 함수가 이 역할을 합니다)하기 때문에 그대로 불러와서 트리를 만들어준다. 이때, index_file에 저장된 정보를 적절하게 파싱해서 트리를 복사해서 반환한다.

3. `readNode(BufferedReader br, Node parentNode, int indexInParentNode, int totalNumberOfKeys)`: 저장되어 있는 index_file을 파싱해서 트리를 복사한다. DFS 기반으로 재귀적으로 실행된다. 즉 leaf node에 도달하면 다시 다른 정보를 복사한다.

Node.java

1. (생성자) `Node(int totalNumberOfKeys, boolean isLeaf, Node parent)`: 빈 노드를 생성한다.
2. `isLeaf()`: 해당 노드가 leaf인지(true) 아닌지(false)에 따라 boolean값을 반환한다.
3. `getCurrentNumberOfKeys()`: 현재 노드에 key가 몇 개나 있는지 int값을 return한다.
4. `setCurrentNumberOfKeys(int currentNumberOfKeys)`: 현재 노드에 있는 key의 개수를 함수 인자로 설정한다.
5. `getKey(int i)`: 노드의 i번째 key값(int)을 반환한다.
6. `setKey(int key, int i)`: 노드의 i번째 key값을 key로 변경한다.
7. `getChildNode(int i)`: 노드의 i번째 childNode를 받아온다. (객체를 받아오기 때문에 주소를 받아오는 것과 비슷함)
8. `setChildNode(Node leftNode, int i)`: 노드의 i번째 childNode를 함수인자인 leftNode로 설정한다.
9. `getValue(int i)`: leaf노드에서, i번째 key에 맞는 i번째 value를 가져온다.
10. `setValue(int value, int i)`: leaf노드에서, i번째 key에 맞는 i번째 value를 value로 설정한다.
11. `getParent()`: node의 parent를 반환한다.
12. `setParent(Node parent)`: node의 parentNode를 parent로 설정한다.
13. `findIndexOfKeyInKeyArray(int input)`: input이 keys에서 몇번째에 위치하는지, 혹은 input이 keys안에 없다면 몇번째 사이인지 반환한다. b+tree의 insert에서 받은 키가 노드의 키 배열에서 몇번째에 삽입되면 되는지 반환할 때 쓰인다.
14. `findSmallIndexOfKeyInKeyInKeys(int input)`: input이 해당 노드의 key 배열에

서 몇번째 노드 이상인지 순서를 반환한다.

15. `findIndexOfChild(int input)`: `input`이 해당 노드의 `key` 배열에서 몇번째 `child node` 순서 이상인지를 반환한다.
16. `push(int key, int value)`: (leaf node일 때) `key-value`쌍을 노드의 `keys`와 `values`에 넣어준다. 이때, `key`를 넣으면서 다시 `keys`를 오름차순으로 정렬한다. `value`는 해당하는 `key`순서로 알아서 따라간다. (leaf node가 아닐 때, internal node를 쪼갤 때 사용) 함수 인자로 받은 `key`를 `keys`에 넣어준다. 이때 오름차순으로 정렬한다. 이 `key` 삽입으로 인해 배열이 하나씩 밀리는 것을 처리한다. `child node`의 순서가 밀리는 것도 정상적으로 처리한다.
17. `push_out(int key)`: `key`를 노드에서 지운다. 만약 해당 노드가 leaf가 아니라면 `child node`를 향하는 포인터도 적절하게 바꿔준다.
18. `showKeys()`: 현재 노드에 있는 `key`값을 `'`로 구분해서 보여준다. `single key search`시 이용된다.

BPlusTree.java

1. (생성자) `BPlusTree(int degree, Node root)`: `root`를 `root`로 가지며, `degree`가 `degree`인 빈 `b+tree`를 생성한다.
2. `singleKeySearchNode(int target, boolean showNodeKey)`: `target`이 위치한 노드, 혹은 가장 유사하게 위치한 노드를 반환한다. 예를 들어 `target`이 노드의 `i`번째 키 값보다 같거나 크다면 `i + 1`번째 `child node`로 이동하고, 작으면 `i`번째 `child node`로 내려가면서 탐색한다.
3. `singleKeySearch(int target)`: 해당하는 `target`을 탐색하며(leaf node 탐색까지는 2.의 함수로 진행) 탐색하는 도중 지나가는 모든 internal 노드의 모든 키를 출력한다. 최종적으로 leaf node에 `target`이 존재하면 해당하는 `value`, 없으면 `NOT FOUND`를 출력한다.
4. `rangeSearch(int startTarget, int endTarget)`: `startTarget`이상, `endTarget` 이하에 해당하는 키-밸류값을 모두 출력한다. `startTarget`을 `singleKeySearchNode`로 탐색한뒤 해당하는 leaf node에서 linked list를 따라가며 `endTarget`에 도달할 때까지 출력한다. 다만 해당하는 키가 없거나 `startTarget`과 `endTarget`이 역전되는 등의 경우에는 `NOT FOUND`를 출력한다.
5. `insert(int inputKey, int inputValue)`: 중복 키가 들어오지 않는다고 가정한다(명세서). 삽입할 leaf node를 탐색해서 leaf node의 키가 최대가 아니라면 바로 `key-value`를 삽입해준다. 키 개수가 최대라면 해당 키를 split한다. `virtualNode`에 `inputKey`까지 들어온 leaf node를 넣은 다음 다시 각각 분할해주고 중간값인 `divide_i`에 해당하는 값은 부모 노드로 올려서 삽입해준다. 부모 노드에도 키가 최대인 상태라면 부모 노드도 다시 분할한다(->6.) 부모

노드가 root라면 6.으로 넘어가지 않고 새로운 root를 만들어주고 함수를 종료한다.

6. `internalNodeInsert(Node parentNode, Node childNode, int inputKey)`: 부모 노드도 다시 분할해야 하는 상황에 실행된다. 마찬가지로 virtual node에 자식으로부터 올라온 중간값과 기존의 부모 노드 값을 삽입한 뒤 최대가 아니라면 그대로 종료하고, 최대라면 다시 노드를 분할하고 부모의 부모 노드에 다시 중간 값을 삽입하는 식의 재귀 함수 형태로 구성되어있다. 재귀적으로 호출되는 도중 root가 포화되면 새로운 root를 만들고 함수를 종료한다.
7. `void delete(int inputDeleteKey)`: 트리가 비었다면 The tree is empty를 출력한다. 트리에 해당하는 노드가 없다면 There is no (inputDeleteKey) in the b+tree를 출력한다. 그 외에 삭제 가능한 경우, 해당하는 leaf node 탐색 결과 leaf node에 최소 키 개수보다 키가 많다면 그냥 바로 삭제해준다. 그런데 key 개수가 딱 최소 개수라면 먼저 형제 노드로부터 빌려올 수 있는지 확인해본다. 안된다면 재귀함수 merge를 실행한다.
8. `borrowFromSiblingNode(Node mainNode, int deleteKey)`: 지우려는 key가 있는 노드의 형제 노드에서 key를 빌려올 수 있는지 확인한다. 기본적으로 왼쪽에서 빌려오되 부모의 가장 왼쪽 노드이거나 왼쪽 형제 노드에 키 값이 부족한 경우 오른쪽 형제 노드에서 빌려온다. 형제 노드에도 key가 부족하다면 delete로 돌아가 merge를 실행한다.
9. `merge(Node mainDeleteNode, int deleteKey)`: mainDeleteNode와 형제 노드를 merge한다. 빌리는게 안되는 상황에서 넘어온 것이므로 바로 merge한다. 만약 mainDeleteNode가 parentNode의 0번째 자식인 경우 오른쪽으로 merge하고, 아니라면 일반적으로는 왼쪽으로 merge한다. merge하게되면 부모 노드에서 키가 하나 사라지게 될텐데, 사라지고 나서도 부모 노드의 키 개수가 최소 개수 이상이면 재귀를 그만둔다. 부모 노드에 여전히 키가 부족하다면 redistribute로 넘어간다.
10. `redistribute(Node mainNode, int deleteKey)`: 아래에 merge한 노드가 있는 상황에 실행된다. 8과 비슷하게, internal node에서 형제 노드에서 키를 빌려올 수 있는 경우 빌려온 뒤 재귀를 종료한다. 빌려올 수 없는 경우 internal Merge를 실행한다. 11과 재귀적으로 반복해서 실행되는 함수다.
11. `internalMerge(Node mainDeleteNode, int leastkey)`: internal node들끼리 merge해주는 함수이다. 마찬가지로 형제 노드와 merge한 뒤, 부모 노드에 키가 적다면 또다시 redistribute를 실행해준다. 그런데 부모 노드가 root였다면, root에 키가 2개 이상이었을 경우 root에서 키를 하나 빼오고, root에 키가 하나 밖에 없었다면 전체 트리의 높이가 낮아지며 병합한 노드가 새로운 root가 된다.
12. `saveTree(String indexFile)`: indexFile에 tree를 저장한다. degree를 먼저 저장하고, 나머지는 saveNode에서 재귀적으로 저장한다. 트리 구조 자체를 복사하듯 저장한다.

13. saveNode(FileWriter fw, Node node, int parentNodeNumber): 각 노드를 순회하면서 트리를 저장하는데 일조한다. DFS 기반으로 순회한다. 개행문자 '\n'으로 각 노드를 구별한다.

Instructions for compiling source

- 주의 사항: 시스템 환경 변수 편집

고급 → 환경 변수 → CLASSPATH를 "%JAVA_HOME%\lib;"로 편집하고 저장해주시길 바랍니다.

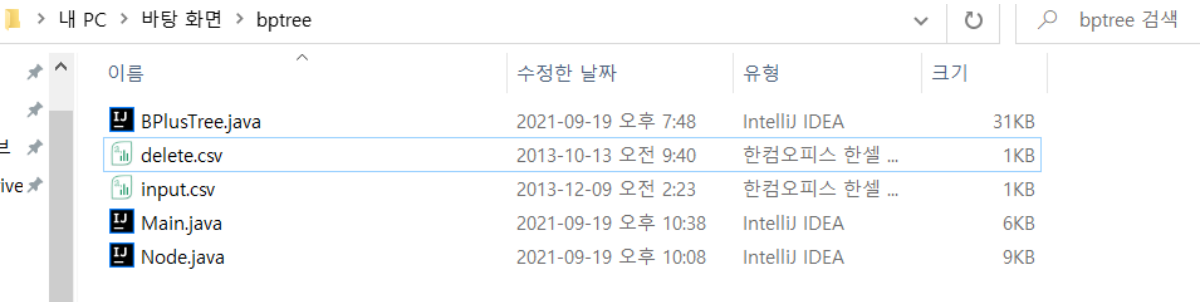
- 주의 사항: 인코딩

javac (컴파일할 .java 파일) -encoding UTF-8

주석에 한글이 있으므로 적절하게 인코딩을 해주기 위해 컴파일시 위와 같이 인코딩을 UTF-8로 설정해주시길 바랍니다.

- cmd에서 컴파일하기

(1) 준비물



내 PC > 바탕 화면 > bptree				▼	↺	🔍 bptree 검색
이름	수정한 날짜	유형	크기			
BPlusTree.java	2021-09-19 오후 7:48	IntelliJ IDEA	31KB			
delete.csv	2013-10-13 오전 9:40	한컴오피스 한셀 ...	1KB			
input.csv	2013-12-09 오전 2:23	한컴오피스 한셀 ...	1KB			
Main.java	2021-09-19 오후 10:38	IntelliJ IDEA	6KB			
Node.java	2021-09-19 오후 10:08	IntelliJ IDEA	9KB			

- ① 세개의 소스코드(Main.java, BPlusTree.java, Node.java)
- ② 삽입 시 필요한 input.csv 파일과 삭제 시 필요한 delete.csv

(2) 컴파일하기(Windows10 기준)

```
명령 프롬프트
Microsoft Windows [Version 10.0.19042.1237]
(c) Microsoft Corporation. All rights reserved.

C:\Users\loveg>cd C:\Users\loveg\Desktop\bptree
C:\Users\loveg\Desktop\bptree>javac *.java -encoding UTF-8
C:\Users\loveg\Desktop\bptree>java Main -c index.dat 4
C:\Users\loveg\Desktop\bptree>java Main -i index.dat input.csv
C:\Users\loveg\Desktop\bptree>java Main -s index.dat 26
20,68,86
1290832
C:\Users\loveg\Desktop\bptree>java Main -r index.dat 10 37
10,84382
20,57455
26,1290832
37,2132
C:\Users\loveg\Desktop\bptree>java Main -d index.dat delete.csv
C:\Users\loveg\Desktop\bptree>java Main -r index.dat 10 37
37,2132
C:\Users\loveg\Desktop\bptree>_
```

- ① (1)의 준비물을 모두 한 폴더에 준비합니다. 해당 폴더를 bptree라고 하면, bptree 폴더까지의 경로에 있는 모든 폴더의 이름은 영어로 해주시길 바랍니다.
- ② 명령 프롬프트(cmd)를 실행하고 bptree 폴더까지의 경로를 복사해서 명령 프롬프트에 다음과 같이 입력합니다.

cd (bptree폴더까지의 경로)

- ③ 세 개의 자바 소스 코드를 컴파일 해줍니다. 인코딩에 주의합니다. 실행 후 세 개의 .class 파일(Main.class, BPlusTree.class, Node.class)이 잘 생성되었는지 확인해주시길 바랍니다.

javac *.java -encoding UTF-8

- ④ 각 명령어는 다음과 같습니다. 형식이 맞지 않으면 익셉션을 발생시킬 수도 있어서 주의해주시길 바랍니다. 또한 실행 파일의 이름이 bptree가 아닌 Main임에 주의해주시길 바랍니다.

- data file creation: java program -c index_file b

degree가 b인 빈 트리를 생성합니다. 생성한 트리를 index_file에 저장합니다. 이 명령어로 index_file을 재구성하실 경우 기존에 insert 및 delete 했던 정보가 전부 사라지고 degree만이 저장됩니다.

java Main -c index.dat 8

- insertion: java program -i index_file data_file

data_file의 key-value 쌍을 index_file에서 지정한 degree의 tree에 저장합니다. 이때

data_file은 .csv파일의 형태여야 합니다. 또한, key-value를 ';'로 구분해야 하며, 한 줄에는 하나의 key-value쌍 만이 존재해야 합니다.

```
java Main -i index.dat input.csv
```

- deletion: java program -d index_file data_file

data_file의 key를 index-file에 저장된 트리로부터 삭제하고 이를 다시 index_file에 덮어씁니다. 이때 datafile은 .csv형태의 파일이어야 하며 한 줄에는 하나의 key가 존재해야 합니다.

```
java Main -d index.dat delete.csv
```

- Single Key Search: java program -s index_file key

index_file에 저장된 트리에서 key 값을 검색합니다. 검색 과정에서 지나치는 internal node의 키 값들을 한 줄에 한 노드씩 출력합니다. 한 노드 안에서 각 키는 ';'로 구별하여 출력합니다. 마지막 leaf node의 key는 출력하지 않으며, 해당하는 key 값이 tree에 있으면 key에 해당하는 value를, 없다면 NOT FOUND를 출력합니다.

```
java Main -s index.dat 125
```

- Range Search: java program -r index_file start_key end_key

index_file에 저장된 트리에서 start_key값 이상, end_key값 이하의 key 값을 검색해서 해당하는 값이 있다면 전부 출력합니다. 이때, key-value 쌍을 'key,value'의 형태로 한 줄에 한쌍씩 출력합니다.

```
java Main -r index.dat 100 200
```

감사합니다😊