

데이터구조 설계 및 실습

2 차 프로젝트 보고서

제출 일자 : 2022 년 11 월 20 일 (일)

학과 : 컴퓨터정보공학부
학번 : 2021202043
이 름 : 이 은 서

1. Introduction

FP-Growth 와 B+ Tree 를 이용하여 상품을 추천하는 프로그램을 구현하는 과제이다. 구매한 상품 데이터들이 저장되어 있는 market.txt 파일을 불러와서 FP-Growth 를 구축한다. 구축한 FP-Growth 는 Tree 구조로 저장되어 있는 FP-Tree 와 상품별 빈도수 및 정보, Tree 와 연결된 노드를 관리하는 Header Table 로 구성되어 있다. 연관된 상품들을 묶은 Frequent Pattern 들은 save 명령어를 통해 result.txt 에 빈도수, 상품 순으로 저장이 되고, 저장된 result.txt 파일을 불러와서 B+Tree 에 저장한다. B+Tree 는 Index Node, Data Node 로 구성되어 있다.

프로젝트에서 요구하는 기능은 다음과 같다.

| 명령어 | 간단한 기능 설명 |
|-------------------------|---|
| LOAD | market.txt로부터 상품 정보를 받아와 FP-Growth 구축 |
| BTLOAD | result.txtf로부터 빈도수와 상품 정보를 받아와 B+Tree에 저장 |
| PRINT_ITEMLIST | 빈도수를 기준으로 Header Table에 저장된 상품을 내림차순으로 출력. 이때 Threshold 보다 작은 빈도수를 가진 상품도 출력한다. |
| PRINT_FPTREE | Header Table의 오름차순 순으로 FP-Tree의 path를 출력. |
| PRINT_BPTREE | B+ Tree에 저장된 Frequent Pattern 중 입력된 상품과 최소 빈도수 이상의 값을 가지는 Pattern을 찾아 출력한다. |
| PRINT_CONFIDENCE | B+ Tree에 저장된 Frequent Pattern 중 입력된 상품과 연관율 이상의 confidence 값을 가지는 Pattern을 찾아 출력한다. |
| PRINT_RANGE | B+ Tree에 저장된 Frequent Pattern 중 입력받은 최소 빈도수와 최대 빈도수 까지의 pattern을 찾아 출력한다. |
| SAVE | FP-Growth에서 상품들의 연관성 결과를 result.txt에 저장한다. |

1) LOAD

LOAD 명령어는 market.txt라는 지정된 파일에 접근하여 txt 파일의 데이터 정보를 불러와 FP-Growth 를 구축하는 명령어이다. 텍스트 파일에는 줄 단위로 상품명이 저장되어 있고 한 줄은 transaction 이다. White space 를 구분자로 하여 상품을 구분한다. 만약 텍스트 파일이 존재하지 않거나 이미 FP-Growth 가 구축된 상태라면 에러코드 100 을 출력한다. FP-Growth 를 구축하기 위해서는 Header Table 을 생성하고 저장할 클래스와 이를 토대로 path 를 생성하여 FP-Tree 를 만들 클래스가 필요하다.

2) BTLOAD

BTLOAD 명령어는 result.txt라는 지정된 파일에 접근하여 txt 파일의 데이터 정보를 불러와 B+ Tree에 정보를 저장하는 명령어이다. 텍스트 파일에는 하나의 줄에 빈도수와 상품명들이 순서대로 저장되어있고, white space 를 구분자로 하여 빈도수와 상품명들을 구분한다. 만약 텍스트 파일이 존재하지 않거나 이미 B+ Tree에 정보가

저장되어 있는 상태라면 에러코드 200 을 출력한다. B+ Tree 를 구축하기 위해서는 데이터 노드를 저장할 클래스와 인덱스 노드를 저장할 클래스가 필요하다.

3) PRINT_ITEMLIST

PRINT_ITEMLIST 명령어는 LOAD 를 통해 구축된 FP-Growth 의 Header Table 에 저장된 상품들을 빈도수를 기준으로 내림차순으로 출력하는 명령어이다. FP-Growth 를 구축할 때는 threshold 이상의 빈도수를 가지는 상품들만 사용하는데, header table 에는 threshold 보다 작은 빈도수를 가진 상품들도 저장되어 있으며, threshold 상관 없이 모든 상품과 빈도수를 출력한다. 만약 Header Table 이 비어있는 경우에는 에러코드 300 을 출력한다.

4) PRINT_FPTREE

PRINT_FPTREE 는 FP Tree 의 정보를 출력하는 명령어이다. Header Table 을 빈도수를 기준으로 오름차순으로 정렬한 이후, threshold 이상의 상품들을 출력한다. 오름차순이기 때문에 threshold 이상인 상품들 중 가장 적은 빈도수를 가진 상품부터 path 가 출력된다. 즉, FP-Tree 를 생각해보면 아래쪽에 있는 노드부터 차례대로 parent 에 접근하여 위로 올라가며 root 에 접근하기 전까지의 상품을 출력하는 것이다. 이때 선택된 노드는 {상품명, 빈도수}로 표시하고, root 노드 전까지 연결된 부모 노드를 출력할 때는 (상품명, 빈도수)의 형태로 출력한다. Root 노드 전까지의 부모 노드들을 모두 출력한 다음에는 선택한 노드의 next node 로 이동하여 이 상품으로부터 시작하는 또 다른 path 를 출력한다. {상품명, 빈도수}에 나타나는 빈도수는 해당 상품의 총 빈도수를 뜻하며, (상품명, 빈도수)에 나타나는 빈도수는 해당 경로에서 중복되는 상품의 빈도수이다. 즉, 모든 item list 를 출력하였을 때 {상품명, 빈도수}에 나타나는 상품명의 path 에서 () 안에 표시된 빈도수의 모든 합이 {} 안에 표시된 빈도수와 같아야 한다. FP-Tree 가 비어있는 경우에는 에러코드 400 을 출력한다.

5) PRINT_BPTREE

PRINT_BPTREE 는 B+ Tree 에 저장되어 있는 Frequent Pattern 중에서 입력된 상품과 최소 빈도수 이상의 값을 가지는 Frequent Pattern 만을 뽑아 출력하는 명령어이다. Command 인자로는 명령어'PRINT_BPTREE'와 찾을 '상품명', '최소 빈도수' 총 3 개의 인자를 순서대로 받으며 입력받은 최소 빈도수를 기준으로 B+ Tree 를 탐색한다. 최소 빈도수를 만족하며 입력된 상품명을 포함하는 Frequent Pattern 을 출력한다. 빠르게 찾기 위해 입력받은 최소 빈도수를 기준으로 Index Node 에서 Data Node 에 접근하고, Data Node 에서 next 로 갈수록 오름차순으로 정렬되어 있으므로 null 을 만나기 전까지 계속해서 next data node 로 접근하면 된다. 만약 출력할 Frequent Pattern 이 없거나 B+Tree 가 비어있는 경우에는 에러코드 500 을 출력한다.

6) PRINT_CONFIDENCE

PRINT_CONFIDENCE 명령어는 B+ Tree 에 저장된 Frequent Pattern 중에서 입력된 상품과 연관율 이상의 confidence 값을 가지는 Frequent Pattern 들을 뽑아 출력하는 명령어이다. Command 인자로는 'PRINT_CONFIDENCE'와 찾을 '상품명', '연관율' 총 3 개의 인자를 순서대로 받으며 입력받은 상품명과 연관율을 기준으로 B+ Tree 를 탐색한다.

$$\text{연관율} = \frac{\text{부분집합의 빈도수}}{\text{해당 상품의 총 빈도수}}$$

연관율은 다음과 같이 계산된다. 만약 B+ Tree 에 저장된 패턴 중 milk 상품이 총 40 개가 있다고 가정해보자. 총 100 개의 패턴이 있고, 그 중에 milk 가 포함된 패턴이 총 40 개이다. 그러면 구할 수 있는 연관율은 40 개가 될 것이다. 이 40 개 패턴에 각각 접근하여 연관율을 계산하고자 하는데, 하나의 패턴을 보니 3 의 빈도수를 가지고 있다면, 이 패턴의 연관율은 $3/4 = 0.75$ 인 것이다. 총 40 개의 패턴에 접근하여 입력받은 연관율보다 높은 연관율을 가진 패턴만 출력하면 된다. B+ Tree 의 탐색을 용이하게 하기 위해 data node 의 맨 처음부터 끝까지 접근하여 해당 상품의 총 빈도수를 구한 후, 입력받은 연관율과 곱하여 올림을 한 값을 구한다. 그리고 구한 이 값보다 크거나 같은 빈도수를 가진 패턴만 체크하여 출력하면 된다. 만약 입력받을 때 2 개의 인자가 모두 입력되지 않거나 형식이 다를 때, 혹은 출력할 frequent pattern 이 없거나 B+ Tree 가 비어 있는 경우에 에러코드 600 을 출력한다.

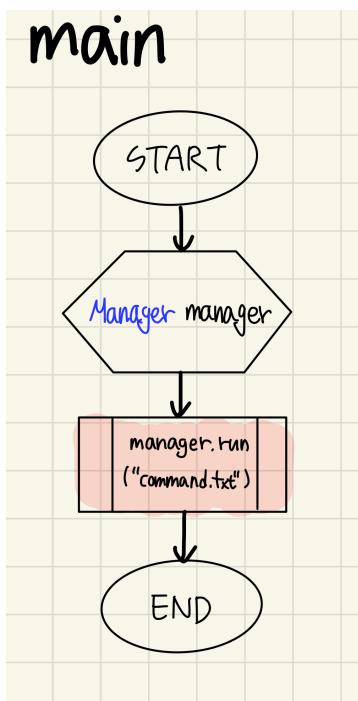
7) PRINT_RANGE

PRINT_RANGE 는 B+Tree 에 저장된 Frequent Pattern 중 최소 빈도수에서 최대 빈도수까지의 빈도수에 해당하는 패턴만 뽑아 출력하는 명령어이다. Command 인자로 'PRINT_RANGE', '상품명', '최소 빈도수', '최대 빈도수'를 순서대로 입력받고 각각은 white space 로 구분한다. B+ Tree 를 탐색하기 위해 최소 빈도수를 기준으로 Index Node 에서 Data Node 로 접근하고, 접근한 노드에서 해당하는 상품명이 있는 Frequent Pattern 을 출력하며 next data node 가 최대 빈도수가 될때까지 접근하고 출력하는 것을 반복한다. 만약 입력받을 인자 3 개가 모두 입력되지 않거나 형식이 다른 경우, 출력할 Frequent Pattern 이 없거나 B+ Tree 가 비어있는 경우 에러코드 700 을 출력한다.

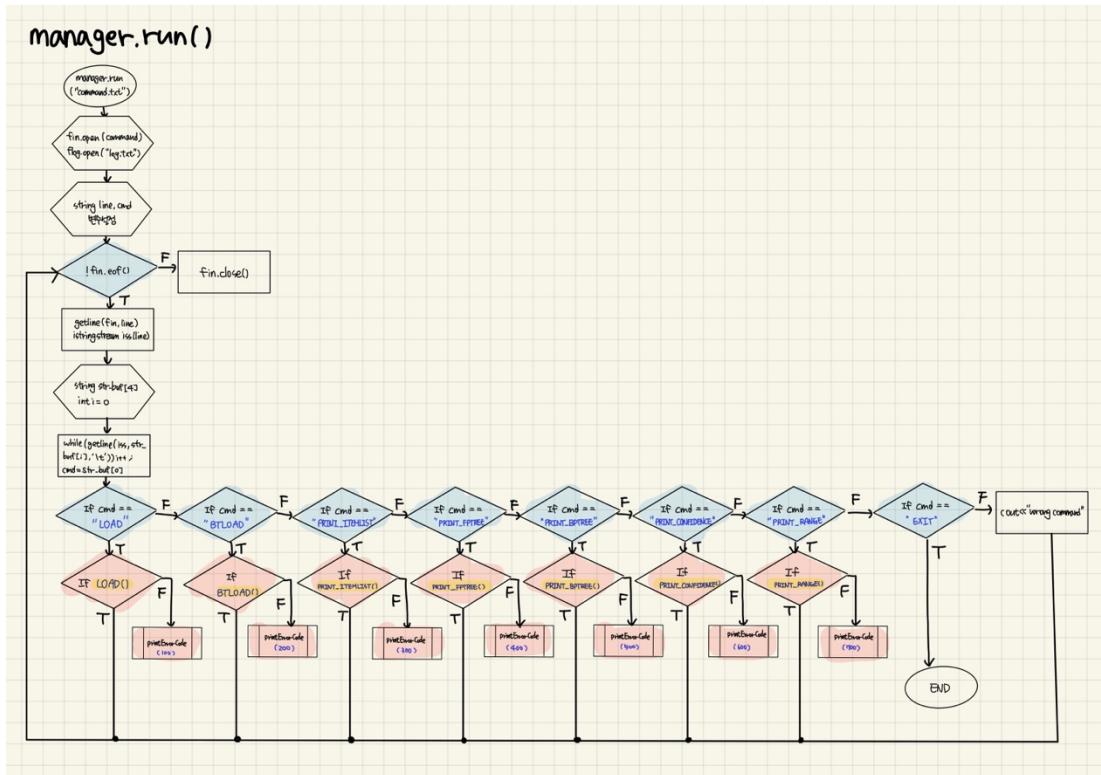
8) SAVE

SAVE 명령어는 FP-Growth 의 상품들의 연관성 결과를 저장하는 명령어이다. 연관성 결과를 토대로 Frequent Pattern 을 생성하고 이를 'result.txt'에 저장한다. White space 를 구분자로 하여 빈도수와 상품명들을 구분하고 하나의 줄은 frequent pattern 이 되는 것이다. 만약 FP-Growth 의 Frequent Pattern 이 비어있는 경우에는 에러코드 800 을 출력한다.

2. Flowchart

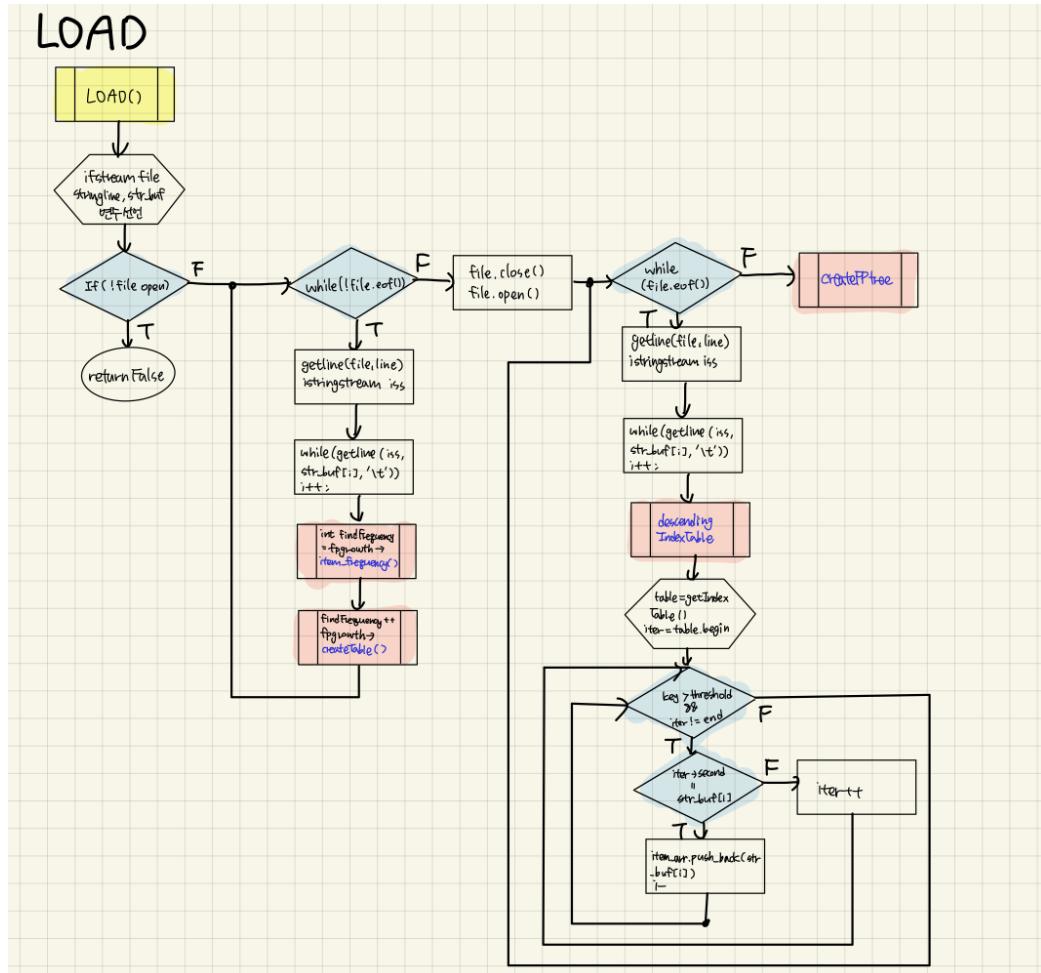


main.cpp 파일의 경우에는 다음과 같은 순서도를 지니고 있다. Manger 클래스의 인스턴스를 만든 다음 해당 인스턴스에 접근하여 command.txt 경로를 인자로 하는 Run 함수를 실행한다. 그리고 Run 함수를 실행한 후 종료 조건을 만나면 프로그램은 끝이 난다. 다음 사진은 Manger 클래스의 Run 함수를 실행할 때의 순서도이다.



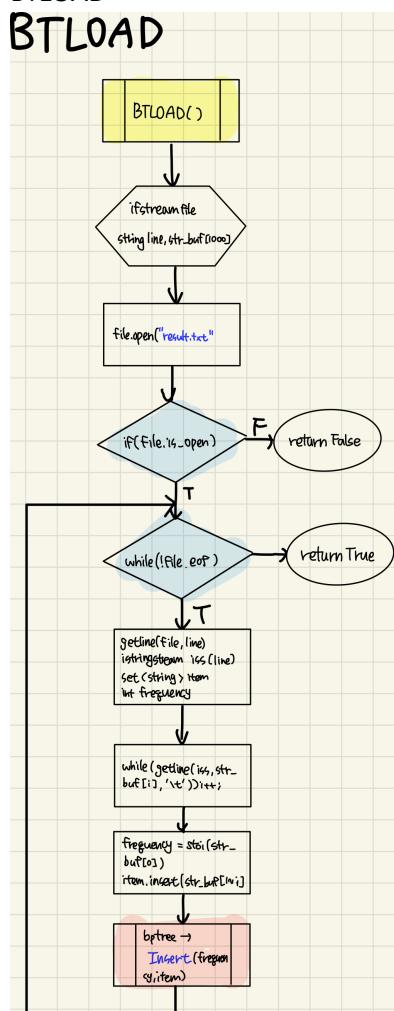
Run 함수를 실행하면 명령어가 적힌 "command.txt" 파일을 열고, log 를 기록할 "log.txt"를 선언한다. 그리고 앞으로 사용할 변수들을 선언한 다음 열린 command.txt 파일의 첫 줄부터 끝 줄까지 줄 단위로 읽어서 명령어와 인자를 구분한다. 파일이 열리지 않으면 출력되지 않았다는 메세지와 함께 프로그램이 종료된다. 한줄씩 읽을 때는 istringstream 과 getline 을 사용하여 white space 를 기준으로 문자열을 자르고 해당 변수에 넣어 명령을 수행한다. 모든 명령어는 white space 가 나오기 전 첫번째 문자열에 저장되므로 cmd 에는 항상 str_buf[0]이 저장된다. Cmd 를 기준으로 총 8 개의 명령어를 나눌 수 있고, 명령어는 LOAD, BTLOAD, PRINT_ITEMLIST, PRINT_FPTREE, PRINT_BPTREE, PRINT_CONFIDENCE, PRINT_RANGE, EXIT 이다. SAVE 명령어의 경우에는 구현을 하지 않았다. EXIT 의 경우에는 바로 프로그램이 종료되므로 따로 flow chart 를 그리지 않았고 나머지 7 개의 명령어는 if 문을 통해 실행이 정상적으로 진행되면 true, 여러 조건을 만나면 false 를 반환하여 바로 해당 error code 를 반환하도록 하였다. PRINT_BPTREE, PRINT_CONFIDENCE, PRINT_RANGE 함수는 몇가지 변수만 추가하여 출력하는 것 외에는 흐름이 똑같기 때문에 똑같은 코드를 사용하였고 따라서 대표적인 PRINT_BPTREE 함수만 플로우 차트를 보일 예정이다. 따라서 지금부터 5 개의 명령어들의 플로우차트를 차례대로 나타내었다. 더 자세하게 풀어쓴 설명은 3.Algorithm 부분에서 예시와 함께 설명하였다.

1. LOAD



LOAD 함수는 file.open 이 실패하면 false 를 반환하며 끝나고 그렇지 않으면 다음 단계로 넘어간다. File.eof 를 사용하여 파일의 끝줄까지 한줄씩 읽어온다. 한줄씩 읽어온 문자열은 line 에 저장되어 있는데 이를 istringstream 을 사용하여 'Wt'를 구분자로 하여 단어를 끊고 각각을 fprowth->item_frequency 함수에 넣어 해당 아이템의 현재 빈도수를 구하고 그 값을 증가시켜 다시 createTable 함수에 넣어 table 에 해당 아이템과 빈도수를 저장한다. 이를 계속 반복하고 끝났을 경우에 파일을 닫았다가 다시 열어서 처음부터 시작하도록 한다. 다시 한줄씩 읽고, istringstream 을 사용하여 white space 를 기준으로 문자열을 나눈 후 descending indextable 함수를 사용하여 indextable 을 내림차순으로 정렬한 후 이를 처음부터 끝까지 순회한다. 해당 key 값이 threshold 보다 크고 끝이 아닐 때만 계속 반복을 하며 아이템 명이 같으면 이를 item_arr 에 순차적으로 넣는다. 이 과정은 내림차순으로 정렬된 index table 을 기준으로 각각의 transection 에서 빈도수를 기준으로 상품들을 정렬하고자 하기 위해 진행되었다. 모든 과정이 끝나면 createFPTree 에 정렬된 transection 을 한줄씩 넣어서 fptree 를 구축한다.

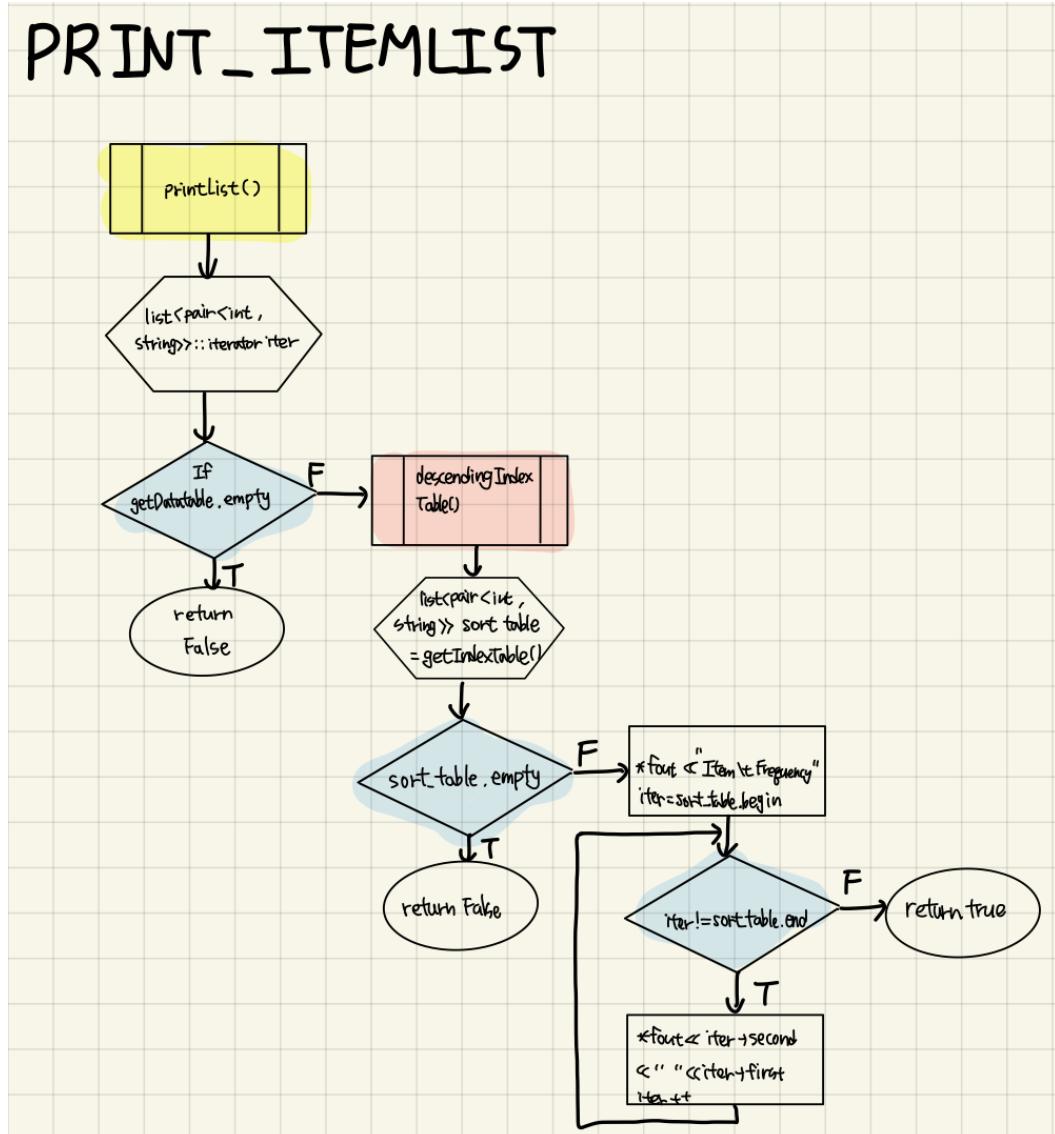
2. BTLOAD



BTLOAD 함수는 'result.txt'파일을 열고 한줄씩 읽어서 이를 bptree 에 Insert 함수를 사용하여 빈도수와 아이템을 넣어 b+ tree 를 구축하는 함수이다. 만약 파일을 열지

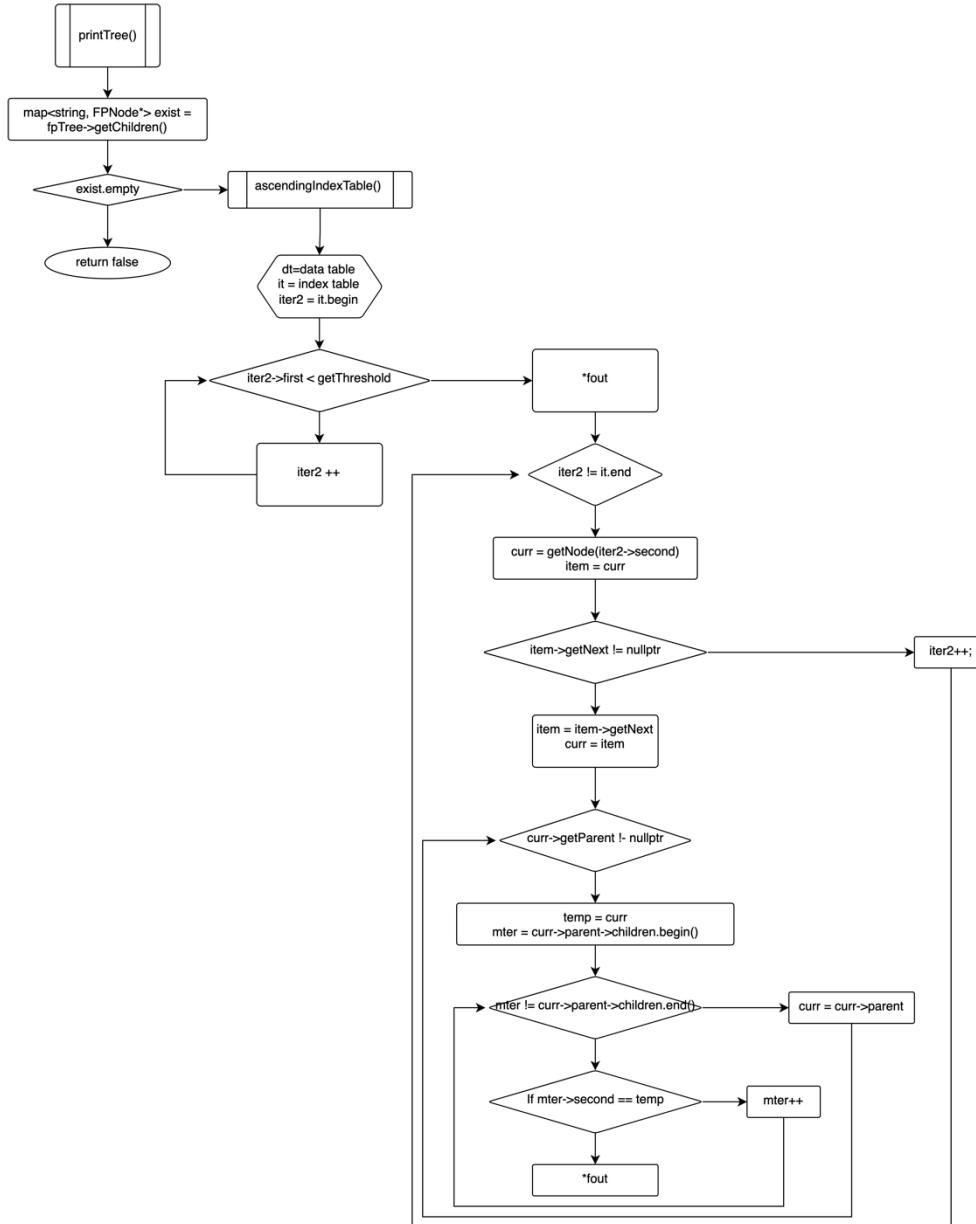
못한 경우에는 false 를 반환하고, 파일을 열고 끝줄까지 모두 읽은 다음에는 true 를 반환한다.

3. PRINT_ITEMLIST



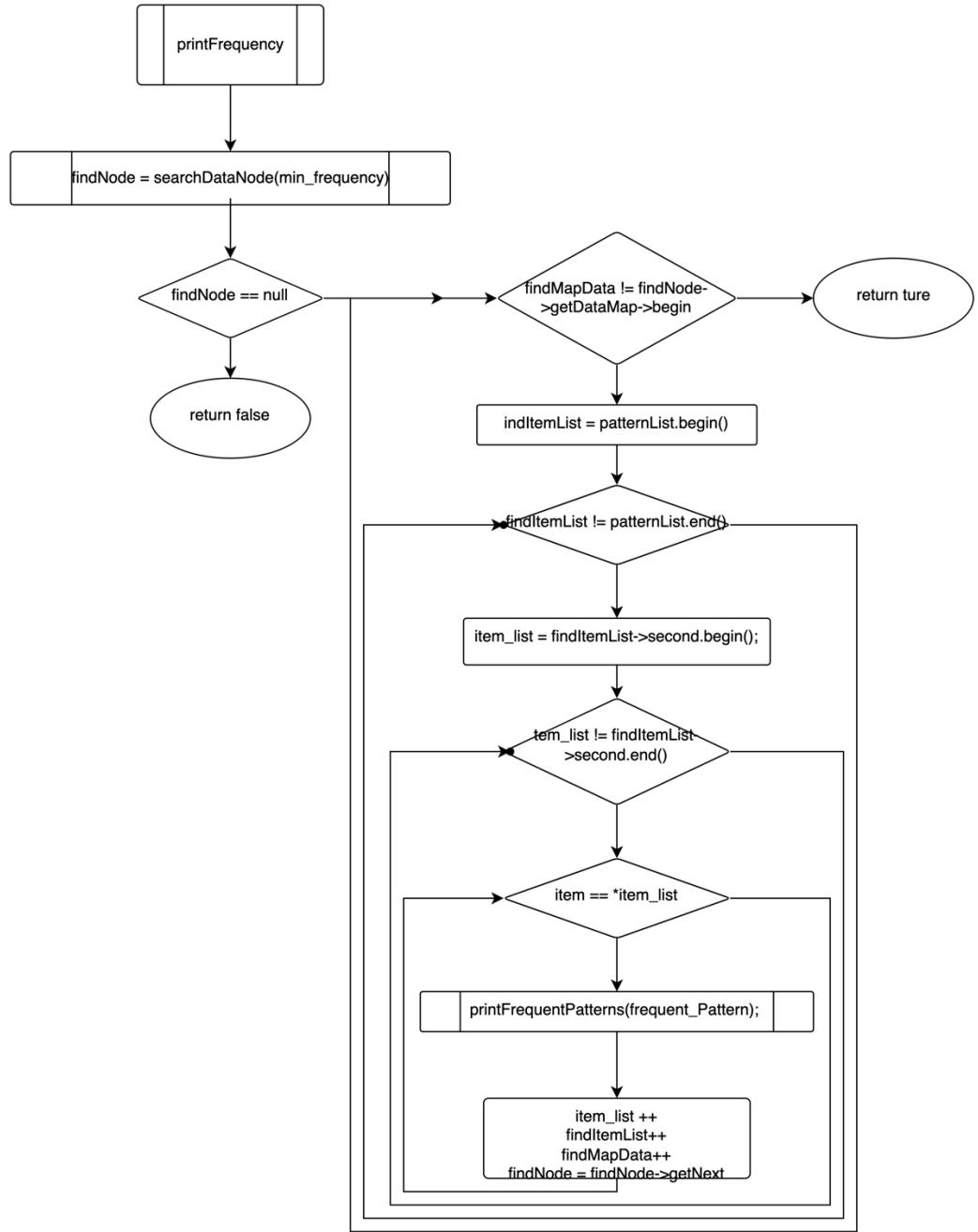
PRINT_ITEMLIST 는 fpgrowth 의 printList 함수를 수행하므로 해당 함수의 플로우차트로 대체한다. Datatable 이 비어있는 경우에는 false 를 반환하며 그렇지 않은 경우에는 descendingIndexTable 함수를 사용하여 indextable 을 내림차순으로 정렬한다. 그리고 내림차순을 정렬한 indextable 에 접근하여 sort_table 로 지정한다. 만약 인덱스 테이블이 존재하지 않는 상태이면 false 를 반환하고 그렇지 않은 경우에는 출력을 진행한다. 해당 인덱스 테이블의 처음부터 끝까지 상품명과 상품의 빈도수를 출력하는 것을 반복하고 모든 과정이 끝나면 true 를 반환한다.

4. PRINT_FPTREE



printFptree 는 empty 상태를 체크한 뒤 ascendingIndexTable 함수를 사용하여 indextable 을 오름차순으로 정렬한다. 그리고 정렬된 index table 과 data table 을 가지고 와서 순회를 반복한다. Index table 에 저장된 상품명들에 하나씩 접근하여 getNode를 통해 FPNode 를 불러오고 이를 next로 진행해가며 모든 노드에 접근한다. 그다음 그 노드에서부터 root 가 되기 전까지 경로를 출력한 다음 다시 getNode 로 돌아온 다음 next로 이동한다.

5. PRINT_BPTREE



PRINT_BPTREE 는 가장 먼저 min_frequency 의 값을 가지는 data Node 를 찾는다. 찾지못하면 false 를 반환한다. 찾은 경우에는 findNode 인 DataNode 에 저장된 데이터맵의 처음부터 접근하여 끝까지 while 문을 반복하는데, 이때 하나의 데이터맵에 저장된 여러개의 pattern List 를 출력하기 위해 다시 처음부터 끝까지 접근하는 while 문을 사용한다. 그리고 빈도수를 뽑아내고 해당 상품명이 존재하는 패턴이면 패턴을 출력하는 함수를 실행한다. 모든 while 문이 끝날때까지 반복한 다음 true 를 반환한다.

6. PRINT_CONFIDENCE, PRINT_RANGE

앞에서 설명했듯이 Print_confidence 와 print_range 함수는 print_bptree 와 구조가 비슷하다. Confidence 의 경우에는 연관율을 따로 검색하여 출력하는 형태이고,

`printrnage` 는 범위만 하나 추가한 것이다. 몇가지 변수를 제외한 나머지의 흐름은 비슷하기 때문에 5. PRINT_BPTREE 를 대표로 설명하였다. 코드 역시 같은 코드를 사용하였고 각각의 `max_frequency` 혹은 `confidence` 와 같은 변수를 추가한 것 외에는 큰 차이점이 없다. 더 자세한 함수의 흐름 설명은 3. Algorithm 파트(p.19)에서 자세하게 설명하였다.

3. Algorithm

1) FP-Growth

Header Table 과 FP-Tree 로 이루어진 class 이다. Header Table 은 `indexTable`, `DataTable` 로 구성되어 있고, `IndexTable` 에는 `frequency` 와 상품명에 대한 정보가 저장되어 있고, `Data Table` 에는 상품명과 fp-tree 를 가리키는 `pointer` 정보가 저장되어 있다. `Market.txt` 에 저장된 데이터를 불러와서 같이 구매한 물품을 출단위로 읽어와 헤더테이블을 구성한 후 이를 토대로 fp-tree 를 형성한다. 이때, fp-tree 는 주어진 `threshold` 값 이상의 빈도수를 지닌 상품명들로만構성을 한다.

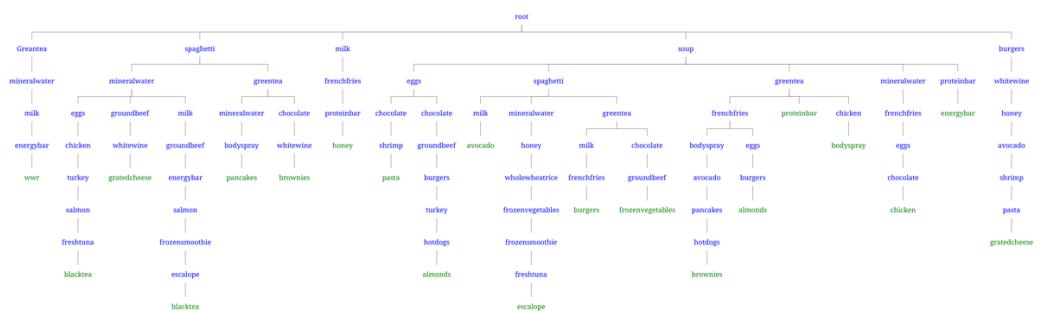
Header Table 은 `threshold` 보다 작은 상품들도 저장을 하며, `IndexTable`, `DataTable` 로 구성되어 있다. `IndexTable` 은 위에서 설명한대로 빈도수와 상품명을 저장하고 있는데, 이때 두개의 값을 저장하기 위해서 STL 의 `list` 와 `pair` 를 사용한다. `IndexTable` 은 빈도수를 기준으로 오름차순과 내림차순으로 정렬할 수 있는 함수가 구현되어 있는데, `sort` 함수를 사용한다. `Sort` 의 `default` 는 오름차순이므로 그대로 사용하여도 되고, 내림차순으로 정렬할 때는 `greater` 를 추가하여 내림차순으로 정렬이 가능케 한다. `DataTable` 은 상품명과 상품과 연결되는 포인터를 저장하고 있는데 이는 STL `map` 을 사용하여 컨테이너 형태로 `key` 와 `value` 에 각각 상품명과 상품 노드 포인터를 저장한다.

FP Tree 는 `FPNode` 들이 트리형태로 이루어져 있는 모양이다. `Root` 는 자식노드만을 가지고 있고, 따로 값을 가지지 않는다. 자식노드들은 `map` 형태로 저장하고, 부모 노드를 가리키는 노드가 존재한다. `Key` 와 `value` 에 각각 상품명과 해당상품의 빈도수 정보, 및 노드 정보를 가리키는 `FPNode*` 포인터를 저장한다. 저장된 노드들은 Header Table 에서 같은 상품노드끼리 연결되어야 한다.

다음은 testcase1, result1 파일의 테스트 케이스를 가지고 나타낸 그림이다. 각각의 상품명에 해당하는 노드들이 연결리스트로 계속해서 연결되어있다고 보면 된다.

IndexTable DataTable

| 번호 | 상품명 | Pointer |
|----|---------------|---------|
| 12 | Soup | |
| 9 | spaghetti | |
| 9 | greentea | |
| 7 | mineral water | |
| 5 | milk | |
| 5 | eggs | |



각각의 transection 을 index table 에 저장된 내림차순의 번호수 기준으로 상품을 정렬한다. 정렬한 transection 을 하나의 path 로 잡고 root 에서부터 시작하여 노드를 생성하고 트리를 만든다. 상품들을 번호수 기준으로 정렬한 다음 fp-tree 를 구축하는 이유는 root 바로 밑의 자식 노드 개수를 줄이기 위해서이다. 가장 많은 번호수를 가진 노드들이 root 의 자식노드가 되어야 복잡하지 않고 깔끔하게 정렬이 가능하다. 깔끔한 정렬은 후에 탐색에 있어서 시간 단축을 가능케 한다. 그리고 트리에 나와있는 각각의 노드들을 헤더테이블의 데이터노드에 저장되는 pointer 와 연결하고, 연결리스트 형태로 같은 상품명이 쭉 연결된다. 나중에 PRINT_FPTREE 기능을 구현할 때, threshold 이상의 값이면서 가장 적은 번호수부터 시작하여 root 노드 직전까지의 부모 노드를 출력한 후 다음 상품명으로 이동할 수 있다. 이때 이동할 수 있는 이유는 header table 의 data node 의 각 상품의 포인터 value 와 fp-tree 의 노드, 그리고 fp-tree 의 노드와 또다른 path 에 있는 같은 상품명의 노드를 연결하였기 때문이다. Fp-tree 의 children 에 저장된 데이터는 상품명과 상품노드이며, 상품 노드에는 번호수 정보, 부모노드 정보, 자식 노드 정보가 저장되어 있다.

FP-Growth 를 구축하기 위해서는 HeaderTable 과 FPNode 들로 이루어진 fptree 가 필요하다. 즉, 크게 HeaderTable class, FPNode class, FPgrowth class 가 필요하다. 각각의 class 에서 구현한 함수는 다음과 같다.

- HeaderTable class

| 반환 타입 | 함수명(인자) | 기능 설명 |
|--|---|---|
| void | insertTable(string item, int frequency) | 입력받은 상품명과 빈도수를 기준으로 table에 저장하는 함수이다. |
| list<pair<int, string>> | getIndexTable() | HeaderTable이 가지고 있는 IndexTable 정보를 반환하는 함수이다. 빈도수와 상품명을 값으로 가지고 있다. |
| map<string, FPNode*> | getDataTable() | HeaderTable이 가지고 있는 DataTable 정보를 반환하는 함수이다. 상품명과 FPNode를 가리키는 포인터를 값으로 가지고 있다. |
| FPNode* | getNode(string item) | 입력받은 상품명의 포인터를 dataTable에서 찾아 반환하는 함수이다. |
| void | descendingIndexTable() | IndexTable을 빈도수를 기준으로 내림차순으로 정렬하는 함수이다. |
| vlid | ascendingIndexTable() | IndexTable을 빈도수를 기준으로 오름차순으로 정렬하는 함수이다. |
| int | findFrequency(string item) | header table에 저장된 빈도수를 찾는 함수이다. |

ㄱ. insertTable

새로운 FPNode를 생성한다.

- 만약 indextable이 비어있는 경우, 모든 header table이 비어있는 것이므로 고려사항 없이 data table과 index table에 값을 저장한다. Index table에는 frequency와 item을 저장하고, data table에는 item과 nullptr을 저장한다. 왜냐하면 아직 fp tree를 구축하기 전이므로 상품명이 가리키고 있는 fpnode가 없기 때문이다.

- 만약 indexTable이 비어있지 않은 경우에는, 값이 저장되어 있는 상태이므로 상품명이 이미 존재하는지 아닌지를 확인해야 한다. 인덱스테이블의 처음부터 끝까지 순회하면서 second값, 상품명이 입력된 상품명과 같은지 아닌지 확인한다.

- 1) 만약 같은 상품이라면, 인덱스테이블과 데이터테이블에서 해당 상품명이 저장된 list와 Map을 삭제한 다음, push_back을 사용하여 다시 값을 넣는다.
- 2) 만약 같은 상품이 없다면, 곧바로 인덱스테이블에 값을 저장한다.

ㄴ. findFrequency

insertTable 함수에서 값을 넣을 때 indexTable이 비워있지 않고, 해당 인덱스테이블에 이미 같은 상품명의 정보가 저장되어 있을 때 frequency를 바꾸기 위하여 값을 탐색하는 함수이다. Indextable이 비어있는 경우에는 어떠한 값도 존재하지 않으므로 0을 return하고, indextable의 처음부터 끝까지 순회하면서 같은 이름의 값이 존재하는지 찾은 다음, 그 값의 frequency값을 return한다. 만약 찾지 못할 경우에는 0을 return한다.

경로로 설정하여 파일을 읽고 링크드리스트에 저장한다는 것이다. 그 외에는 LOAD의 기능과 동일하다.

- FPNode class

| 반환 타입 | 함수명(인자) | 기능 설명 |
|-----------------------------------|---|--|
| void | setParent(FPNode* node) | fp tree 에서 부모노드를 설정하는 함수이다.. |
| void | setNext(FPNode* node) | fp tree 에서 next 노드를 설정하는 함수이다. |
| void | pushchildren(string item, FPNode* node) | 상품명과 상품 정보가 저장되어 있는 children 노드를 저장하는 함수이다. |
| void | updateFrequency(int frequency) | 해당 노드의 frequency 에 입력받은 빈도수만큼 더하여 update 하는 함수이다. |
| int | getFrequency() | 해당 노드의 frequency 를 반환하는 함수이다. |
| FPNode* | getParent() | 해당 노드의 parent Node 를 반환하는 함수이다. |
| FPNode* | getNext() | 해당 노드의 next Node 를 반환하는 함수이다. |
| FPNode* | getChildrenNode() | 해당 노드의 children Node 를 반환하는 함수이다. |
| map<string, FPNode*> | getChildren() | 해당 노드의 children map 을 반환하는 함수이다. getChildrenNode() 함수와 다른 점은, 이는 상품명과 빈도수, 부모노드, 자식노드, next 의 값을 모두 가지고 있는 같은 레벨에 있는 여러개의 node 들을 반환하는 것이다. |

- FPGrowth class

| 반환 타입 | 함수명(인자) | 기능 설명 |
|---------------------|--|---|
| void | createTable(string item, int frequency) | 입력받은 item 과 frequency 를 가지고 테이블을 생성하는 함수이다. 여기서 HeaderTable class 의 함수인 insertTable 에 접근한다. |
| void | createFPTree(FPNode* root, HeaderTable* table, list<string> item_array, int frequency) | fp tree 를 생성하는 함수이다. 루트노드부터 시작하여 하나의 transection 을 path 로 저장하는 함수이다. |
| void | connectNode(HeaderTable* table, string item, FPNode* node) | fp tree 에 있는 같은 상품명을 header table 의 data table 과 연결하고, fp tree 에 있는 같은 상품끼리도 연결하여 연결리스트를 만든다. |
| int | item_frequency(string item) | HeaderTable calss 의 함수인 find_frequency 함수에 접근하여 해당 상품의 빈도수를 찾는 함수이다. |
| FPNode* | getTree() | 해당 fptree 를 반환하는 함수이다. |
| HeaderTable* | getHeaderTable() | 해당 fp tree 가 가진 header table 을 반환하는 함수이다. |
| int | getThreshold() | 해당 fp tree 에 주어진 threshold 를 반환하는 함수이다. |
| bool | printList() | 빈도수를 기준으로 내림차순으로 정렬된 index table 을 차례대로 접근하여 상품명과 빈도수를 출력하는 함수이다. |
| bool | printTree() | 빈도수를 기준으로 오름차순으로 정렬된 index table 중 threshold 이상의 값을 빈도수로 가진 상품에 차례대로 저장하여 root 직전의 부모노드까지의 경로를 출력하는 함수이다. |

ㄱ. createFPTree

curr = root 부터 밑으로 진행한다.

item_array 에 저장된 상품명들을 하나씩 불러와서 새로 생성한 FPNode 에 정보를 저장한다. 먼저 새로 만든 FPNode 의 next = nullptr 로 지정한다.

1. Curr 노드의 자식노드 중 해당 상품의 노드가 존재하지 않을 때

바로 자식노드에다가 새로만든 FPNode 를 가리키는 children node 를 만든 다음, 그 자식노드로 curr 를 이동시킨다. 그 다음 frequency 를 증가시킨다. 그리고 데이터테이블과 연결하기 위해 connectNode 함수를 실행한다.

2. Curr 노드의 자식노드 중 해당 상품의 노드가 존재할 때

이미 해당 상품의 노드가 자식중에 있으므로 그 노드로 이동한 다음 frequency 를 증가시킨다.

↳. connectNode

데이터 테이블에서 입력받은 상품명이 있는 map 컨테이너를 찾는다. 그리고 map 컨테이너의 두번째 값인 value, FPNode* 포인터를 가져와 해당 포인터의 next 가 널일때까지 계속 옆으로 이동한 다음 마지막 노드와 createFPTree 에서 새로 생성한 노드를 연결한다. 하나의 FPNode* 포인터는 단방향연결리스트이다.

▷. printList

만약 데이터테이블이 비어있으면 이는 headerTable 에 어떠한 값도 없는 것으로 false 를 반환한다. 그런 것이 아니라면 일단 indexTable 을 내림차순으로 정렬한 다음 정렬된 indexTable 을 다시 불러온다. 그리고 내림차순으로 정렬된 인덱스테이블에 처음부터 끝까지 접근하면서 상품명과 빈도수를 출력한다. 이는 PRINT_ITEMLIST 함수에 사용된다.

◁. printTree

만약 fptree 가 비어있으면 false 를 반환한다. 그런 것이 아니라면 일단 indexTable 을 오름차순으로 정렬한 다음, 인덱스테이블의 처음부터 끝까지 접근한다. 만약 threshold 보다 적은 값의 빈도수를 가지고 있으면 넘어가고, 그렇지 않으면 해당 상품명과 빈도수를 출력한 다음, 동일한 상품명을 데이터테이블에서 찾은 다음 데이터테이블의 포인터의 next 가 null 일때까지 계속 이동해가면서 root 직전까지의 부모 노드를 출력한다.

2) B+ Tree

B+ Tree 는 result.txt 에 저장된 데이터를 이용하여 구축하는 자료구조이다. 빈도수와 연관 상품들인 Frequent Pattern 들이 한줄씩 저장되어있고 이를 읽어서 Data Node, Index Node 를 구성하여 B+ Tree 를 구축한다. Index Node 가 있기 때문에 빈도수를 기준으로 사용해야하는 함수의 경우 빠른 접근이 가능하여 시간 단축의 장점이 있다. 데이터노드는 Frequent Pattern 이 저장되어있는 FrequentPatternNode 와 frequency 를 데이터로 가지고 있는 map 컨테이너 형식으로 되어있고, FrequentPatternNode 는 pattern 의 크기와 element 들을 multimap 형식으로 저장하고 있다. 그리고 데이터노드는 order 이상의 크기를 가질 수 없고, 새로운 데이터 노드를 저장할 때

order 와 같은 크기를 갖게 되면 split 을 진행하여 새로운 인덱스 노드를 만든다. 인덱스노드들도 저장할때 order 와 같은 크기를 갖게 되면 split 을 진행한다. 다음은 B+Tree 를 구축하기 위해 알아야 할 정보이다.

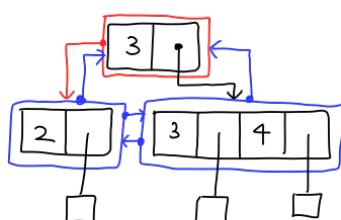
B+Tree

- ① result.txt에서 빈노드 & 상용형 & 상용형 &... 일시문제
- ② 데이터노드 BtreeDataNode 는 `map<int, FrequentPatternNode*> mapData` 를 이루고 있다.
 - 1) FrequentPatternNode 는 `multimap<int, string> FreqPatternList` 으로 같은 빈도로 이루어진 여러 패턴들을 `set` 형태로 저장한다.
 - 2) `multimap<int, string> FreqPatternList`
패턴별로
제한 횟수
 - 3) 맵의 `int <= 1`이며 저장 X
- ③ 맵 힙속 데이터노드 vs 삽입방법 데이터노드 비교
 - 1) 더 적으면 정렬에 쓰임
 - 2) 더 크면 다음 노드와 비교하여 삽입
 - 3) `mapData.size() <= order` 까지 많아야 하며, 맵은 힙에는
 - i) split을 하고 위치는 $\lfloor \frac{order-1}{2} + 1 \rfloor$ if $order=3 \rightarrow 2$, $order=4 \rightarrow 3$, $order=5 \rightarrow 3$
 - ii) split 위치 정기적인 데이터노드, 뒤의 데이터노드를 만든 다음 `*pNext`, `*pPrev`를 지정해준다.
 - iii) split 위치의 데이터노드의 frequency 를 가져와 새롭게 인덱스노드를 만든다.
 - iv) 만든 인덱스노드는 원래 `*pParent`가 처리하는 노드에 넣는다.
 - v) ii)에서 split한 노드 2개의 `*pParent`를 다시 연결해준다.

이 그림은 split 을 하는 과정을 설명한 그림이다.

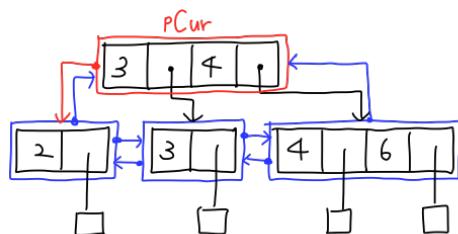
B+Tree < Insert & Split >

①



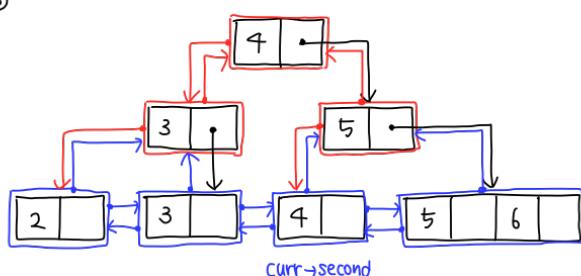
Insert 4

②



Insert 6

③



Insert 5

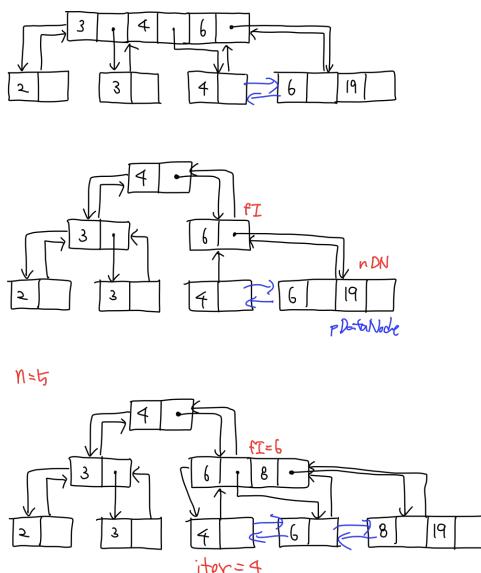
첫번째, Order = 3 이라고 가정했을 때, 2 와 3 의 key 를 가진 데이터노드가 있는 상태에서 4 를 입력하면 해당 데이터노드는 꽉차게 된다. 이때 split 을 진행하며 split 을 하는 위치에 대한 식은 다음과 같다.

$$split\ place = ceil(\frac{order - 1}{2.0} + 1)$$

다음과 같이 split 을 진행하면 기존 노드에는 2만 남아있고 새로만든 노드에 3과 4를 저장한 다음 3의 데이터를 뽑아 새로운 인덱스 노드를 만든다. 그리고 새로운 인덱스 노드와 기존의 데이터 노드, 새로만든 데이터노드를 서로 연결한다.

두번째, 6 을 집어넣었을 때 6 은 root 부터 인덱스노드의 key 값과 비교하여 삽입할 데이터노드에 접근한다. 인덱스노드는 3 이고, 6 은 3 보다 크므로 3 의 BpTreeNode* 포인터가 가리키는 데이터노드에 접근하여 삽입을 진행한다. 그러면 해당 데이터노드는 3, 4, 6 의 값이 들어가고 3의 사이즈를 가지게 되므로 split 을 진행하게 된다. 진행한 결과 기존 노드에 4, 6 가 삭제되고 3 만 남으며, 새로만든 데이터노드에는 4, 6 의 값이 들어가고 새로운 인덱스 노드는 4 의 값을 가지고 3 이 저장되어있는 map 에 저장된다.

세번째, 5 를 집어넣었을 때는 인덱스노드에 있는 3, 4 의 값과 비교하였을 때 마지막 값인 4 보다도 5 가 크기때문에 4 의 BpTreeNode* 포인터가 가리키는 값을 이동한다. 그러면 기존 노드인 4, 5 에 6 이 추가되고 꽉차게되면서 split 을 진행한다. 그리고 새로운 인덱스노드인 5 를 기준노드(4)가 가리키는 부모노드인 인덱스노드(3, 4)에 접근하여 새로운 인덱스값인 5 를 삽입한다. 그런데, 인덱스노드 역시 꽉차게되면서 split 이 진행된다. 가운데 값인 4 가 가리키던 most left child 노드에 접근한 다음 이의 부모노드로 indexnode 의 split 을 진행하여 새로 만들어진 인덱스노드(5)를 가리키게 한다. 그리고 위로 올라가는 인덱스노드(4)의 most left child 노드로는 원래의 인덱스노드(3)를 가리키고, 포인터로는 새로만들어진 인덱스노드(5)를 가리키게 한다.



이해를 돋기위해 가져온 새로운 split 예시이다.

BpTree class 에서 구현한 함수는 다음과 같다.

| 반환 타입 | 함수명(인자) | 기능 설명 |
|------------------------|---|--|
| bool | Insert(int key, set<string> set) | 입력받은 key 값과 상품리스트를 가지고 B+Tree 를 구축한다. |
| bool | excessDataNode(BpTreeNode* pDataNode) | 입력받은 DataNode 가 꽉 차있는지 아닌지를 판단하는 함수이다. |
| bool | excessIndexNode(BpTreeNode* pIndexNode) | 입력받은 IndexNode 가 꽉 차있는지 아닌지를 판단하는 함수이다. |
| void | splitDataNode | DataNode 가 꽉 찬 경우 split을 진행한다. |
| void | splitIndexNode | IndexNode 가 꽉 찬 경우 split을 진행한다 |
| BpTreeNode* de* | getRoot() | root 를 반환하는 함수이다. |
| BpTreeNode* de* | searchDataNode(int n) | 데이터노드중에서 n 의 값을 key 로 가지고 있는 노드를 찾는 함수이다. |
| void | printFrequentPatterns(set<string> pFrequentPattern) | 상품명을 정해진 형식대로 출력하는 함수 |
| bool | printFrequency(string item, int min_frequency) | 입력받은 최소빈도수 이상의 값을 빈도수로 가지고 있는 상품 frequent Pattern 들을 출력하는 함수이다. |
| bool | printConfidence(string item, double item_frequency, double min_frequency) | 해당상품의 총 빈도수와 최소 넘어야할 빈도수를 받아와서 연관율 이상의 상품 frequent pattern 들을 출력하는 함수이다. |
| bool | printRange(string item, int min, int max) | 빈도수가 min 보다 크거나 같고 Max 보다 작거나 같은 frequent pattern 들을 출력하는 함수이다. |

ㄱ. Insert

만약 루트가 비어있다면 아직 B+ tree 가 비어있는 상태인 것으로 새로운 데이터 노드와 새로운 패턴 노드를 생성한 다음 새로만든 데이터 노드에 데이터맵을 넣고, 이 데이터 노드를 루트로 지정한다.

만약 b+ tree 가 존재한다면, 가장 먼저 insert 할 key 값이 이미 있는지 아닌지를 확인해야 한다. 따라서 searchDataNode 함수를 실행하여 찾는다. 이미 있는 값이면 해당 노드의 주소를 반환하고, 그렇지않으면 널을 반환한다.

1. 이미 있는 값일때. 반환하는 주소는 해당 데이터맵이 아닌 데이터맵이 존재하는 데이터노드이므로 데이터노드를 다시 찾아야한다. 순회하다가 찾은 경우에는 그 데이터맵의 FrequentPatternNode 의 멀티맵에 새로운 pattern 을 insert 한다.
2. B+ tree 에 존재하지 않는 값일때, root 부터 시작하여 순회한다. 인덱스노드가 존재하면 그 인덱스노드의 key 값과 입력할 값의 key 를 비교하여 넣을 위치를 찾는다. 찾으면 새로운 패턴 노드를 만들고 이를 아까 찾은 넣을 위치에 새로운 데이터맵으로 삽입한다. 그리고 넣는 것이 끝났을 때 그 데이터노드가

꽉 차있는지 excessDataNode 함수로 판단을 한 다음, true 일 때 splitDataNode 함수를 실행하여 split 을 진행한다.

└. searchDataNode

root 부터 시작하여 순회할 예정이다. 만약 인덱스노드가 있는 상태라면 데이터노드가 나타날때까지 인덱스노드의 key 값과 찾는 n 의 값을 비교하여 이동한다. 찾은 데이터노드는 데이터맵이 아니므로 데이터노드 내에서 다시 찾아줘야하는데 만약에 똑같은 값이 있는 경우에는 그 데이터 노드를 return 한다. 없는 경우에는 null 을 반환한다. 만약 시작부터 인덱스노드가 없는 상태였다면, 데이터노드를 끝까지 순회해야하므로 next 로 이동하는 함수의 추가가 필요하다.

ㄷ. splitDataNode

위에서 설명한 split place 를 찾는 식을 사용하여 위치를 정하고, 기존 데이터 노드와 split 후 새로 생성될 데이터 노드, 그리고 새로 만들어질 인덱스 노드 이렇게 총 3 개가 필요하다. 기존노드가 필요한 이유는 기존노드가 원래 가리키고 있던 parent, prev, next 등의 포인터 정보가 필요하기 때문이다. 자세한 split 방법은 위에서 설명하였으므로 생략한다.

ㄹ. splitIndexNode

splitIndexNode 함수도 splitDataNode 와 비슷하다. Split 위치를 찾고 split 한 다음 새로운 인덱스노드 2 개를 만들어서 연결시킨다. 하지만 splitDataNode 와 다른 점은 parent 가 될 새로운 인덱스 노드를 만든 경우, 기존 인덱스노드에서 가리키고 있던 pointer 값을 새로운 인덱스 노드와 most left child 와 parent 관계로 연결해야 한다는 점이다. 그리고 parent 인 인덱스 노드에 새로운 인덱스 맵을 넣었는데 꽉 찬 경우에는 다시 한 번 split 을 진행해야하므로 일종의 재귀함수가 된다는 사실이다. 이를 주의하여 split 을 진행하면 된다.

ㅁ. printFrequency

최소 빈도수를 가진 데이터 노드를 searchDataNode 함수를 사용하여 찾은 다음 해당 노드부터 next 로 이동하며 모든 frequent pattern 들을 가져온다. 이중에서 입력받은 상품명과 같은 상품이 있는 pattern 들만 출력한다. 이는 PRINT_BPTREE 함수를 구현하는데 사용된다.

ㅂ. printConfidence

printConfidence 함수가 호출되기 전 미리 아이템의 전체 총 빈도수와 입력받은 연관률을 곱하여 계산하여 최소한의 빈도수를 구한다. 그리고 구한 최소한의 빈도수와 전체 총 빈도수를 함께 인자로 받아 함수를 실행하는데, PrintFrequency 와 비슷하게 최소 빈도수를 가진 데이터노드를 searchDataNode 함수를 사용하여 찾은 다음 해당 노드부터 next 로 이동해가며 모든 frequent pattern 들 중 입력받은 상품명과 같은 상품이 있는 pattern 을 가져온다. 그리고

각 pattern 의 빈도수와 총 빈도수를 계산하여 confidence 를 구한 다음 이 confidence 를 함께 출력한다. 이는 PRINT_CONFIDENCE 함수를 구현하는데 사용된다.

. printRange

printRange 함수도 마찬가지로 최소한의 빈도수를 가진 데이터 노드를 searchDataNode 함수를 사용하여 찾은 다음 해당 노드로 부터 next 로 이동해간다. 다만, 최대 빈도수보다 적은 빈도수를 가진 데이터노드여야 한다. 그리고 이 데이터노드의 요소에 모두 접근하여 해당 아이템이 존재하는 frequent pattern 들을 모두 출력한다. 이는 PRINT_RANGE 함수를 구현하는데 사용된다.

4. Result Screen

Command.txt 는 다음과 같은 명령어를 적었을 때 log.txt 는 다음과 같다. 결과 도출을 위해 사용한 result.txt 와 market.txt 는 각각 result1, testcase1 과 같다.

| | | | |
|---|--------------------------------|---|------------------|
| 1 | LOAD | 1 | =====LOAD===== |
| 2 | BTLOAD | 2 | Success |
| 3 | PRINT_ITEMLIST | 3 | ===== |
| 4 | PRINT_FPTREE | 4 | |
| 5 | PRINT_BPTREE eggs 3 | 5 | =====BTLOAD===== |
| 6 | PRINT_CONFIDENCE green tea 0.1 | 6 | Success |
| 7 | PRINT_RANGE spaghetti 3 4 | 7 | ===== |
| 8 | EXIT | | |

| | | | |
|----|--------------------------|----|------------------------|
| 9 | =====PRINT_ITEMLIST===== | 41 | |
| 10 | Item Frequency | 42 | black tea 2 |
| 11 | soup 12 | 43 | almonds 2 |
| 12 | spaghetti 9 | 44 | whole wheat pasta 1 |
| 13 | green tea 9 | 45 | toothpaste 1 |
| 14 | mineral water 7 | 46 | tomatoes 1 |
| 15 | milk 5 | 47 | soda 1 |
| 16 | french fries 5 | 48 | shampoo 1 |
| 17 | eggs 5 | 49 | shallot 1 |
| 18 | chocolate 5 | 50 | red wine 1 |
| 19 | ground beef 4 | 51 | pet food 1 |
| 20 | burgers 4 | 52 | pepper 1 |
| 21 | white wine 3 | 53 | parmesan cheese 1 |
| 22 | protein bar 3 | 54 | meatballs 1 |
| 23 | honey 3 | 55 | ham 1 |
| 24 | energy bar 3 | 56 | gums 1 |
| 25 | chicken 3 | 57 | fresh bread 1 |
| 26 | body spray 3 | 58 | extra dark chocolate 1 |
| 27 | avocado 3 | 59 | energy drink 1 |
| 28 | whole wheat rice 2 | 60 | cottage cheese 1 |
| 29 | turkey 2 | 61 | cookies 1 |
| 30 | shrimp 2 | 62 | carrots 1 |
| 31 | salmon 2 | 63 | bug spray 1 |
| 32 | pasta 2 | | ===== |
| 33 | pancakes 2 | | |
| 34 | hot dogs 2 | | |
| 35 | grated cheese 2 | | |
| 36 | frozen vegetables 2 | | |
| 37 | frozen smoothie 2 | | |
| 38 | fresh tuna 2 | | |
| 39 | escalope 2 | | |
| 40 | brownies 2 | | |

DS_project_2 · log.txt

```

HeaderTable.h x HeaderTable.cpp x FPNode.h x FPNode.cpp x FPGrowth.h x FPGrowth.cpp x BpTree.h x BpTree.cpp x command.txt x log.txt x
=====
65 =====PRINT_BTREE=====
66 {StandardItem, Frequency} (Path_Item, Frequency)
67 {almonds, 2}
68 {(almonds, 1) (burgers, 1) (eggs, 1) (french fries, 2) (green tea, 4) (soup, 12)}
69 {(almonds, 1) (hot dogs, 1) (burgers, 1) (ground beef, 1) (chocolate, 2) (eggs, 2) (soup, 12)}
70 {black tea, 2}
71 {(black tea, 1) (fresh tuna, 1) (salmon, 1) (turkey, 1) (chicken, 1) (eggs, 1) (mineral water, 3) (spaghetti, 5)}
72 {(black tea, 1) (escalope, 1) (frozen smoothie, 1) (salmon, 1) (energy bar, 1) (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)}
73 {brownies, 2}
74 {(brownies, 1) (hot dogs, 1) (pancakes, 1) (avocado, 1) (body spray, 1) (french fries, 2) (green tea, 4) (soup, 12)}
75 {(brownies, 1) (white wine, 1) (chocolate, 1) (green tea, 2) (spaghetti, 5)}
76 {escalope, 2}
77 {(escalope, 1) (frozen smoothie, 1) (salmon, 1) (energy bar, 1) (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)}
78 {(escalope, 1) (fresh tuna, 1) (frozen smoothie, 1) (frozen vegetables, 1) (whole wheat rice, 1) (honey, 1) (mineral water, 1) (spaghetti, 4) (soup, 12)}
79 {fresh tuna, 2}
80 {(fresh tuna, 1) (salmon, 1) (turkey, 1) (chicken, 1) (eggs, 1) (mineral water, 3) (spaghetti, 5)}
81 {(fresh tuna, 1) (frozen smoothie, 1) (frozen vegetables, 1) (whole wheat rice, 1) (honey, 1) (mineral water, 1) (spaghetti, 4) (soup, 12)}
82 {frozen smoothie, 2}
83 {(frozen smoothie, 1) (salmon, 1) (energy bar, 1) (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)}
84 {(frozen smoothie, 1) (frozen vegetables, 1) (whole wheat rice, 1) (honey, 1) (mineral water, 1) (spaghetti, 4) (soup, 12)}
85 {frozen vegetables, 2}
86 {(frozen vegetables, 1) (whole wheat rice, 1) (honey, 1) (mineral water, 1) (spaghetti, 4) (soup, 12)}
87 {(frozen vegetables, 1) (ground beef, 1) (chocolate, 1) (green tea, 2) (spaghetti, 4) (soup, 12)}
88 {grated cheese, 2}
89 {(grated cheese, 1) (pasta, 1) (shrimp, 1) (avocado, 1) (honey, 1) (white wine, 1) (burgers, 1)}
90 {(grated cheese, 1) (white wine, 1) (ground beef, 1) (mineral water, 3) (spaghetti, 5)}
91 {hot dogs, 2}
92 {(hot dogs, 1) (pancakes, 1) (avocado, 1) (body spray, 1) (french fries, 2) (green tea, 4) (soup, 12)}
93 {(hot dogs, 1) (turkey, 1) (burgers, 1) (ground beef, 1) (chocolate, 2) (eggs, 2) (soup, 12)}
94 {pancakes, 2}
95 {(pancakes, 1) (body spray, 1) (mineral water, 1) (green tea, 2) (spaghetti, 5)}
96 {(pancakes, 1) (avocado, 1) (body spray, 1) (french fries, 2) (green tea, 4) (soup, 12)}
97 {pasta, 2}

```

The image shows two side-by-side code editor windows, likely from the CLion IDE, displaying the contents of a file named `log.txt`. Both windows have identical titles and file lists at the top.

Top Editor (Left):

```
98 (pasta, 1) (shrimp, 1) (chocolate, 2) (eggs, 2) (soup, 12)
99 (pasta, 1) (shrimp, 1) (avocado, 1) (honey, 1) (white wine, 1) (burgers, 1)
100 {salmon, 2}
101 (salmon, 1) (turkey, 1) (chicken, 1) (eggs, 1) (mineral water, 3) (spaghetti, 5)
102 (salmon, 1) (energy bar, 1) (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)
103 {shrimp, 2}
104 (shrimp, 1) (chocolate, 2) (eggs, 2) (soup, 12)
105 (shrimp, 1) (avocado, 1) (honey, 1) (white wine, 1) (burgers, 1)
106 {turkey, 2}
107 (turkey, 1) (chicken, 1) (eggs, 1) (mineral water, 3) (spaghetti, 5)
108 (turkey, 1) (burgers, 1) (ground beef, 1) (chocolate, 2) (eggs, 2) (soup, 12)
109 {whole wheat rice, 2}
110 (whole wheat rice, 1) (energy bar, 1) (milk, 1) (mineral water, 1) (green tea, 1)
111 (whole wheat rice, 1) (honey, 1) (mineral water, 1) (spaghetti, 4) (soup, 12)
112 {avocado, 3}
113 (avocado, 1) (honey, 1) (white wine, 1) (burgers, 1)
114 (avocado, 1) (milk, 1) (spaghetti, 4) (soup, 12)
115 (avocado, 1) (body spray, 1) (french fries, 2) (green tea, 4) (soup, 12)
116 {body spray, 3}
117 (body spray, 1) (mineral water, 1) (green tea, 2) (spaghetti, 5)
118 (body spray, 1) (french fries, 2) (green tea, 4) (soup, 12)
119 (body spray, 1) (chicken, 1) (green tea, 4) (soup, 12)
120 {chicken, 3}
121 (chicken, 1) (eggs, 1) (mineral water, 3) (spaghetti, 5)
122 (chicken, 1) (chocolate, 1) (eggs, 1) (french fries, 1) (mineral water, 1) (soup, 12)
123 (chicken, 1) (green tea, 4) (soup, 12)
124 {energy bar, 3}
125 (energy bar, 1) (milk, 1) (mineral water, 1) (green tea, 1)
126 (energy bar, 1) (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)
127 (energy bar, 1) (protein bar, 1) (soup, 12)
128 {honey, 3}
129 (honey, 1) (protein bar, 1) (french fries, 1) (milk, 1)
130 (honey, 1) (white wine, 1) (burgers, 1)
```

Bottom Editor (Right):

```
131 (honey, 1) (mineral water, 1) (spaghetti, 4) (soup, 12)
132 {protein bar, 3}
133 (protein bar, 1) (french fries, 1) (milk, 1)
134 (protein bar, 1) (green tea, 4) (soup, 12)
135 (protein bar, 1) (soup, 12)
136 {white wine, 3}
137 (white wine, 1) (burgers, 1)
138 (white wine, 1) (chocolate, 1) (green tea, 2) (spaghetti, 5)
139 (white wine, 1) (ground beef, 1) (mineral water, 3) (spaghetti, 5)
140 {burgers, 4}
141 (burgers, 1)
142 (burgers, 1) (french fries, 1) (milk, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
143 (burgers, 1) (eggs, 1) (french fries, 2) (green tea, 4) (soup, 12)
144 (burgers, 1) (ground beef, 1) (chocolate, 2) (eggs, 2) (soup, 12)
145 {ground beef, 4}
146 (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)
147 (ground beef, 1) (mineral water, 3) (spaghetti, 5)
148 (ground beef, 1) (chocolate, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
149 (ground beef, 1) (chocolate, 2) (eggs, 2) (soup, 12)
150 {chocolate, 5}
151 (chocolate, 2) (eggs, 2) (soup, 12)
152 (chocolate, 1) (eggs, 1) (french fries, 1) (mineral water, 1) (soup, 12)
153 (chocolate, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
154 (chocolate, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
155 {eggs, 5}
156 (eggs, 1) (mineral water, 3) (spaghetti, 5)
157 (eggs, 2) (soup, 12)
158 (eggs, 1) (french fries, 1) (mineral water, 1) (soup, 12)
159 (eggs, 1) (french fries, 2) (green tea, 4) (soup, 12)
160 {french fries, 5}
161 (french fries, 1) (milk, 1)
162 (french fries, 1) (mineral water, 1) (soup, 12)
163 (french fries, 2) (green tea, 4) (soup, 12)
```

```

DS_project_2 > log.txt
HeaderTable.h HeaderTable.cpp FPNode.h FPNode.cpp FPNode.h FPNode.cpp FPNode.h FPNode.cpp BpTree.h BpTree.cpp command.txt log.txt
164 {French fries, 1} {milk, 1} {green tea, 2} {spaghetti, 4} {soup, 12}
165 {milk, 5}
166 {milk, 1} {mineral water, 1} {green tea, 1}
167 {milk, 1}
168 {milk, 1} {spaghetti, 4} {soup, 12}
169 {milk, 1} {mineral water, 3} {spaghetti, 5}
170 {milk, 1} {green tea, 2} {spaghetti, 4} {soup, 12}
171 {mineral water, 7}
172 {mineral water, 1} {green tea, 1}
173 {mineral water, 3} {spaghetti, 5}
174 {mineral water, 1} {green tea, 2} {spaghetti, 5}
175 {mineral water, 1} {soup, 12}
176 {mineral water, 1} {spaghetti, 4} {soup, 12}
177 {green tea, 9}
178 {green tea, 1}
179 {green tea, 2} {spaghetti, 5}
180 {green tea, 4} {soup, 12}
181 {green tea, 2} {spaghetti, 4} {soup, 12}
182 {spaghetti, 9}
183 {spaghetti, 5}
184 {spaghetti, 4} {soup, 12}
185 {soup, 12}
186 {soup, 12}
187 =====
188 ======PRINT_BTREE=====
189 FrequentPattern Frequency
190 {chocolate, eggs} 3
191 {chocolate, eggs, soup} 3
192 {eggs, soup} 4
193 =====
194 ======PRINT_CONFIDENCE=====
195 =====
196 ======PRINT_CONFIDENCE=====

DS_project_2 > log.txt
HeaderTable.h HeaderTable.cpp FPNode.h FPNode.cpp FPNode.h FPNode.cpp FPNode.h FPNode.cpp BpTree.h BpTree.cpp BpTree.cpp command.txt log.txt
196 ======PRINT_CONFIDENCE=====
197 FrequentPattern Frequency Confidence
198 {brownies, green tea} 2 0.1
199 {burgers, green tea} 2 0.1
200 {chocolate, green tea} 2 0.1
201 {green tea, milk} 2 0.1
202 {green tea, mineral water} 2 0.1
203 {green tea, pancakes} 2 0.1
204 {body spray, green tea, pancakes} 2 0.1
205 {body spray, green tea, soup} 2 0.1
206 {burgers, french fries, green tea} 2 0.1
207 {burgers, green tea, soup} 2 0.1
208 {chocolate, green tea, spaghetti} 2 0.1
209 {green tea, soup, spaghetti} 2 0.1
210 {burgers, french fries, green tea, soup} 2 0.1
211 {body spray, green tea} 3 0.2
212 {french fries, green tea} 3 0.2
213 {french fries, green tea, soup} 3 0.2
214 {green tea, spaghetti} 4 0.2
215 {green tea, soup} 6 0.3
216 =====
217 ======PRINT_RANGE=====
218 FrequentPattern Frequency
219 {ground beef, spaghetti} 3
220 {milk, spaghetti} 3
221 {green tea, spaghetti} 4
222 {soup, spaghetti} 4
223 =====
224 ======EXIT=====
225 Success
226 =====
227 =====
228 =====

```

LOAD 명령어와 BTLOAD 명령어가 잘 수행되었음을 확인할 수 있고, printItemList 명령어 실행 결과 index table 을 빈도수로 정렬한 다음 상품명과 빈도수를 threshold 상관 없이 모두 출력이 된 것을 확인할 수 있다. Print_fpTree 명령어에서는 하나의 상품명과 빈도수를 토대로 여러 경로를 확인 할 때 해당 상품의 빈도수를 모두 합한 결과가 {}안에 있는 빈도수와 같은지 확인하여 모든 경로가 출력되었는지 체크하였다. 그리고 앞서 보여준 testcase1 에 대한 fp tree 를 직접 그려가며 경로를 비교하였고, 정확하게 출력된 것을 확인하였다. Print BpTree 역시 최소 빈도수 이상의 패턴 중에서 해당 상품명을 가진 패턴들을 모두 출력하였고 print confidence 에서는 소수점 둘째자리까지의 연관율을 출력하는 것을 확인하였다. Print range 도 범위를 벗어나지 않고 해당 범위 내에서만 출력이 잘 진행되었고, 마지막으로 EXIT 명령어를 통해 프로그램을 종료하였다.

5. Consideration

LOAD 를 구현하는 중에는 큰 어려움을 겪지는 않았다. 하지만 LOAD 를 성공시키기 위해서는 fp growth 가 구축되었는지 확인되어야 하는데, 이를 확인하는 것이 난감하여 조금 고민을 하였다. 그래서 print itemlist 와 print fptree 를 차례대로 구현하였고, 단계별로 진행하다보니 fp growth 를 구축할 수 있었다. Header table 의 data table 에서 getNode 에 접근하여 경로를 출력하는 과정에서 루트 직전의 부모노드까지 출력을 하면서 이동을 하면 다음 Next node 로 이동하도록 하였는데 자꾸 다른 경로가 출력되는 문제점이 있었다. 이는 다시 출발점으로 돌아가 next node 를 해야했는데, 그러지 않고 해당 부모노드에서 next node 를 하였기 때문에 경로가 다른 상품의 것으로 바뀐 것이었다. 따라서 출발점을 저장해두는 temp node 를 하나 만들고 root 직전까지 갔다가 되돌아오는 방식으로 문제를 해결하였다. Fp growth 를 구축하는 과정이 조금 어렵긴 하였지만 쉽게 고칠 수 있었고, 단계별로 해나가는 느낌이 들어 만족스러웠다. 하지만 bp tree 는 달랐다. 과정이 너무 복잡하게 느껴졌고, 분명히 insert, split, search 함수를 완성하였는데도 그 안에서 여러 오류 케이스들이 발생하여 자꾸 실행이 멈추는 문제점이 발생하였다. 따라서 직접 노트에 그림을 그려가며 케이스들을 분류하고 여러 시행착오를 겪으면서 임시저장해둘 노드를 생성해보기도 하고, 디버깅을 통해서 어디서부터 주소가 날아가는지 체크하기도 하였다. 또한 주어진 result.txt 가 아닌 직접 frequency 를 바꿔보면서 값을 넣어 제대로 insert 가 되는지 확인하였다. 그러다보니 정말 많은 경우의 수를 놓친것을 알게되었는데, 이러한 모든 경우를 어떻게 판별해야하는지 잘 모르겠다. 대표적인 예로는 보통 맨 마지막에 있는 데이터노드에 값을 추가하고 split 되어 indexnode 가 올라가는 형식이었는데 갑자기 중간에 있는 데이터 노드에 데이터맵을 추가하였을 때, 기존 노드와 새로 만든 데이터노드간의 연결은 되어있었지만, 새로만든 데이터노드와 그 앞서 있던 원래 기존노드의 next 로 연결되어있던 데이터노드와의 연결은 끊기는 일이 있었다. 그래서 맨 앞 most left children 으로부터 next 로 이동해가며 모든 데이터노드를 출력하고자 할 때 갑자기 끊기면서 종료되는 현상이 일어났던 것이다. 또 다른 사례로는 indexnode 를 기준으로 key 값의 범위를 통해 most left children 또는 해당 데이터맵의 BpTreeNode* 포인터가 가리키는 곳으로 이동하여 거기서 또 값을 비교하고 이러는데, 이동하는 과정에서 오류가 났었다. 인덱스 노드가 split 되어 새로운 인덱스 노드를 생성하였을 때, split 된 위치의 bptreeNode*포인터가 가리키는 데이터노드를 새로운 인덱스 노드의 mostleftChild 로 설정은 해두었지만 반대로 parent 설정을 안해두었기 때문에 생긴 일이었다. 그리고 insert 와 search 함수에서도 해당 key 값이 존재하는 경우도 있고 존재하지는 않지만 해당 데이터 맵 사이에 들어가야 하는 경우도 있고 정말 많은 경우의 수가 존재하여서 골머리를 썩였다. 그러느라 함수가 많이 복잡하고 더러워지고, 중복되는 함수들도 많이 생겼다. 문제점을 해결하여 원하는 결과값을 얻을 수는 있었지만, 더 깔끔하고 중복되는 함수들이 많이 없도록 클린한 코드를 작성하는데에 대한 공부가 필요하다고 느꼈다. 그리고 모든 케이스들을 고려하여 설계할 수 있으려면 어떻게 해야하는지에 대해 찾아봐야할 의미를 느꼈다. 지금까지는 번호를 매기며 단계를 나누고 하나씩 해왔는데, 또 다른 방법이 있는지 찾아봐야겠다. 이번 과제를 통해 전체적인 상품 구매 데이터를 가지고 FP Growth 를 구축하고 이를 통해

얻은 frequent pattern 들로 b+ tree 자료구조를 구축하는 과정을 진행하면서 해당 자료구조에 대한 이해와 관심을 높일 수 있었다. 또한, 자료구조를 공부하고 직접 구현해보고 오류를 고쳐가면서 성장할 수 있어 매우 도움이 되었다.