# DS Project2

이기훈 교수님

**컴퓨터정보공학부**

**2018202076**

**이연걸**

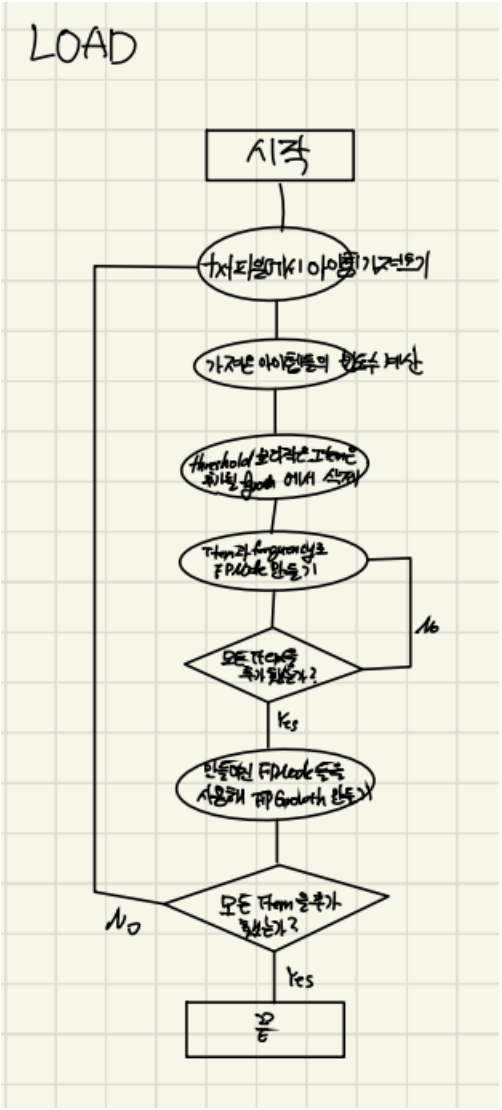
- Introduction

  본 프로젝트는 FP-Growth 와 B+Tree 를 사용해 상품 추천 프로그램을 구현하는 것이다. 텍스트 파일에서 데이터를 읽어와 HeaderTable 로 구성된 FP-Tree 에 삽입해야 하며 연관 상품을 나타내는 Frequent pattern 을 사용해 결과 값을 저장한다. 이 때 FP-Tree 에 대한 이해가 필요하다. FP-Growth 의 threshold 값을 사용해 필요한 데이터만 저장해야 하는 기능이 필요하기에 threshold 에 대해 이해해야 한다.

  위에서 도출된 결과를 다시 읽어와 B+Tree 에 삽입한다. IndexNode 와 DataNode 로 구성되는 B+Tree 를 이해해야 한다. IndexNode 는 DataNode 를 찾기 위해 사용되며 DataNode 는 위에서 설명한 Frequent pattern 정보를 갖고 있기에 B+Tree 라는 하나의 node 를 Index, Data 이 두가지 노드로 상속해 코드의 길이를 줄이고 같은 기능을 다르게 구현한다. 그렇기 때문에 B+Tree 와 상속에 대한 이해가 필요하다. 위의 threshold 와 달리 order 값을 사용해 필요한 데이터만 저장해야 하는 기능이 필요하기에 order 에 대해서도 이해해야 한다.

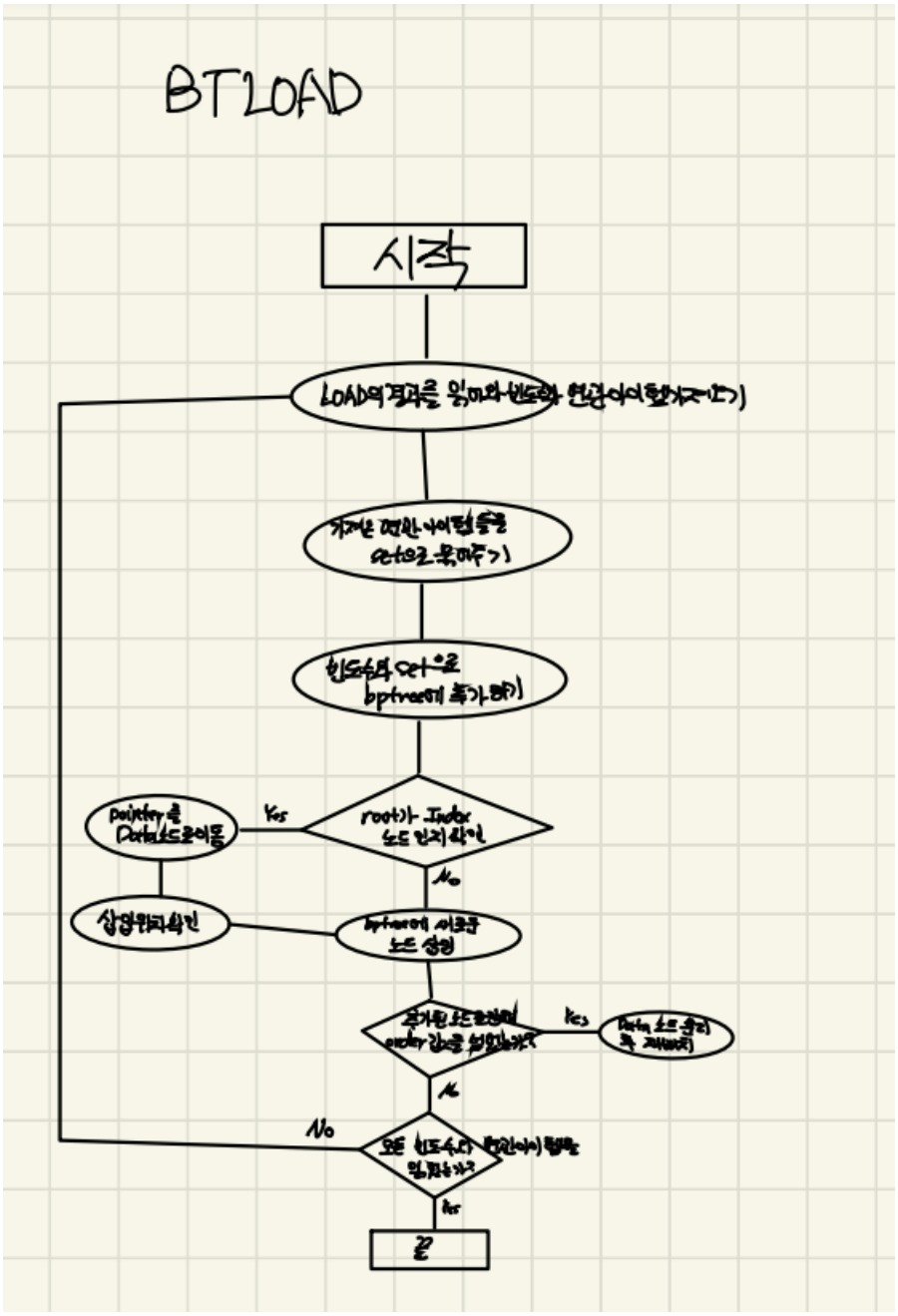
- Flowchart

  - LOAD



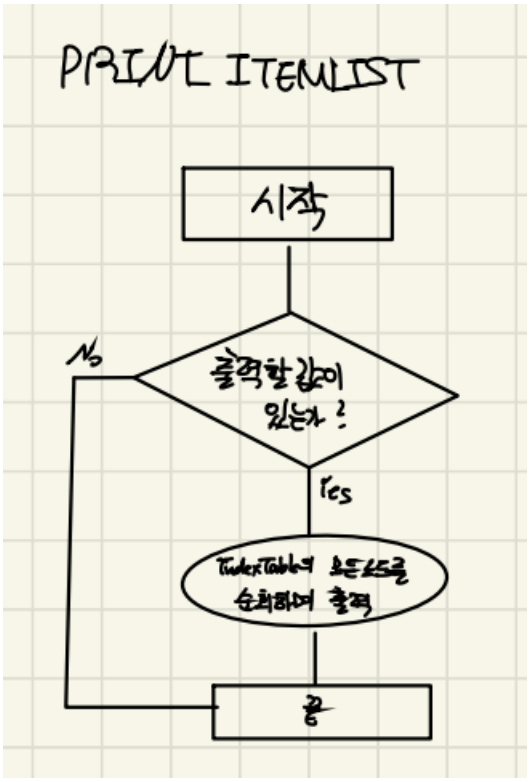

우선 제공된 txt 파일에서 아이템들을 가져와 빈도수를 계산한다. 모든 item 을 frequent patten 에 담지만 threshold 보다 작은 item 을 fp tree 에 삽입되지 않는다. 이제 가져온 item 과 frequency 를 사용해 FPNode 를 만든다. 이렇게 모든 item 을 추가하고 만들어진 FP Node 를 FPGrowth 에 추가하고 이 과정을 txt 파일을 다 읽을 때 까지 반복한다.

- BTLOAD



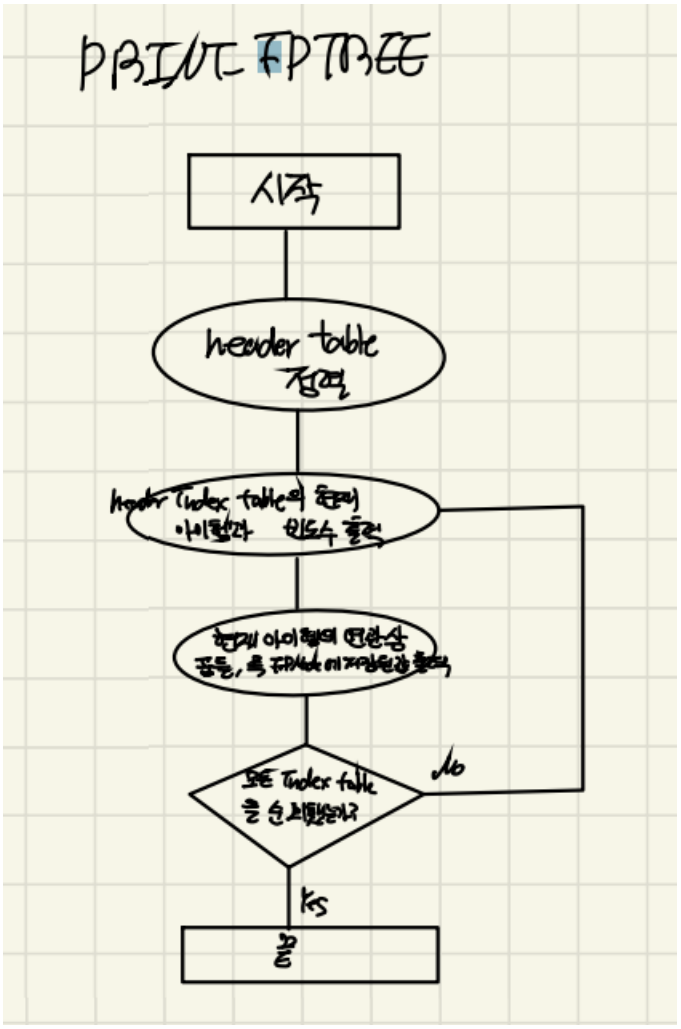

LOAD 의 결과를 읽어온다. 이는 SAVE 를 구현했을 경우지만 이번 과제에서는 구현하지 않았기 때문에 testcase 파일의 읽어온다. 읽어온 데이터를 set 을 사용해 묶어준다. Frequency 와 set 을 사용해 BpTreeNode 를 만든다. 삽입은 현재 root 가 DataNode 인지 IndexNode 인지에 따라 달라지기 때문에 order 값을 이용해 판단하였고, root 가 indexNode 라면 삽입 위치를 결정 한 후에 b+tree 에 삽입해 준다. 삽입한 노드로 인해 order 값을 초과했을 경우 DataNode 를 분리해주는 B+ tree 의 작업을 실행한다. 모든 데이터를 읽어왔다면 함수를 종료한다.

- PRINT_ITEMLIST



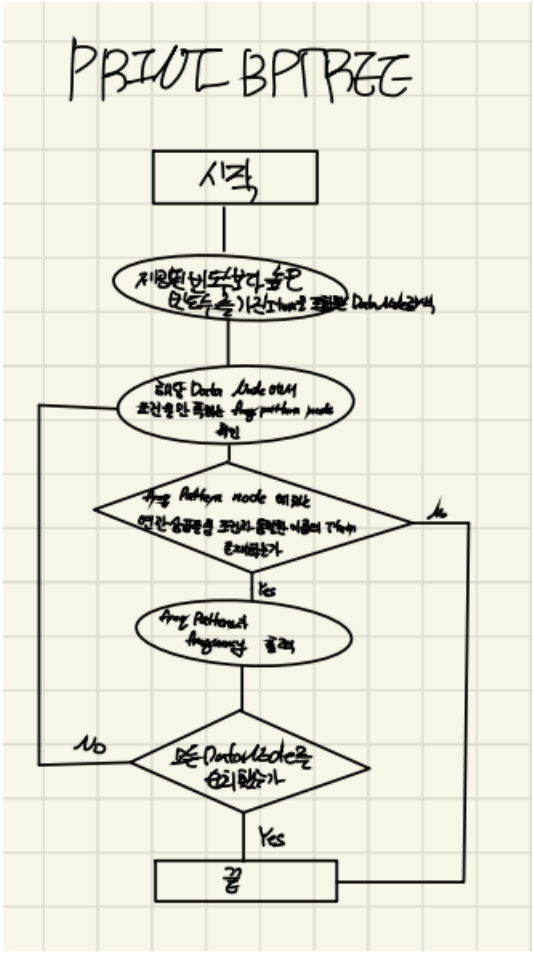

FPGROWTH 가 비어 있는지 확인해 예외 처리를 한 후에 indexTable 에 저장된 값을 item name 과 frequency 로 분리해 출력한다.

- PRINT_FPTREE



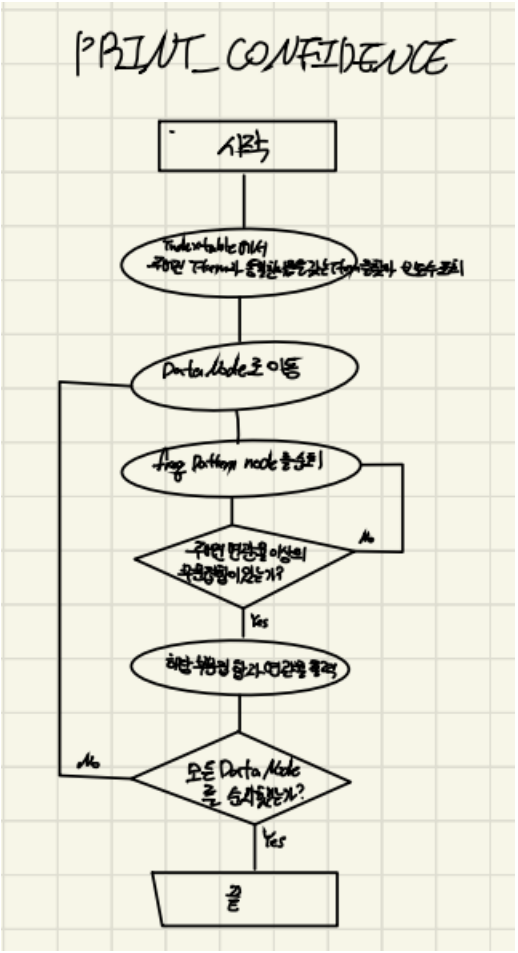

제시된 조건에 따라 headertable 을 정렬한다. 그 후 index table 를 순회하며 현재 index table 의 아이템과 빈도수를 부분 집합과 함께 출력한다. 모든 index table 을 순회했다면 함수를 종료한다.

- PRINT_BPTREE



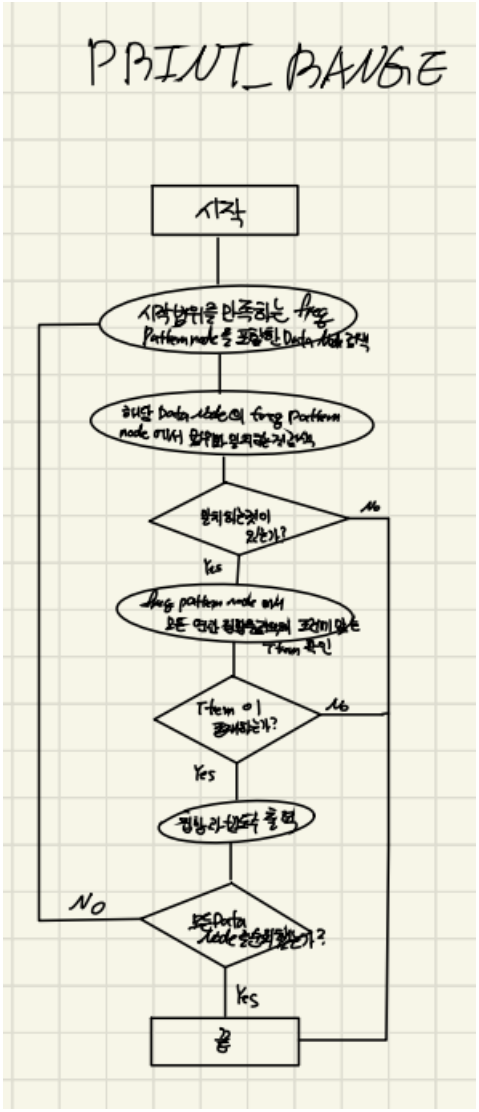

BTLOAD를 통해 만든 DataNode에서 조건을 만족하는 freq pattern node를 포함하고 있는 DataNode 를 찾는다. 해당 DataNode 에서 조건을 만족하는 freq patten node 가 어느 위치에 존재하는지 확인한다. 해당 위치의 frequent pattern node 부터 시작해 모든 data node 를 순회하면서 찾는 item 과 동일한 이름을 갖는 item 을 포함한 부분집합을 출력한다. 모든 DataNode를 순회했다면 함수를 종료한다.

- PRINT_CONFIDENCE




Indextable 을 검사해 제공된 item 과 동일한 이름을 가진 item 의 빈도수를 가져온다. 그 후 DataNode 로 이동해 모든 DataNode 를 검사한다. DataNode 들의 frequenct pattern node 를 검사하면서 제공된 item 과 검사하는 item_set 의 비율, 즉 연관율이 제공된 값을 넘는지 확인한다. 만약 연관율 조건을 만족한다면 해당 item_set 을 연관율과 함께 출력한다. 만약 그 어떤 부분집합, 즉 item_set 도 조건을 만족하지 못했다면 Error 를 발생시키고, 함수를 종료한다.

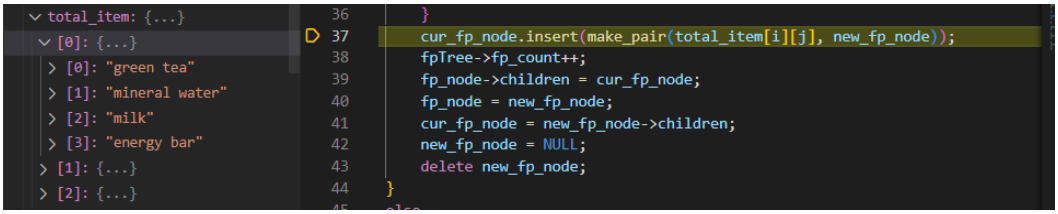
- PRINT_RANGE



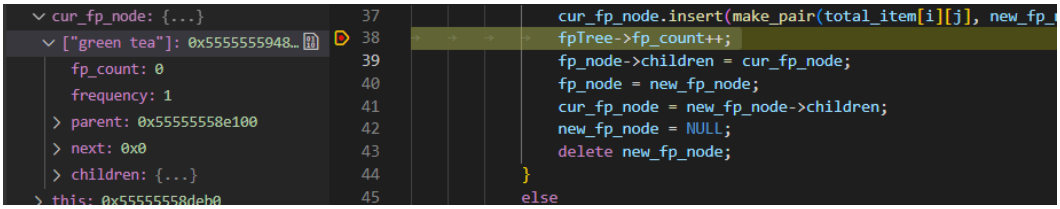

PRINT_BPTREE 와 유사한 동작을 한다. 역시 제시된 start 값 이상인 빈도수를 가진 frequent pattern node 를 찾고 해당 frequent pattern node 를 검사해 제공된 item 이름과 동일한 item 집합을 찾는다. 여기서는 start 와 end 를 확인해 start 이상, end 이하의 빈도수를 가진 부분 집합만을 출력하고 그 외의 동작은 PRINT_BPTREE 와 동일하다.

- Algorithm
  - FP-Growth

    본 과제의 FP-Growth 는 FP-Tree 와 FP-Tree 의 상품 노드를 관리하는 Header Table 로 구성되어 있다. FP-growth 알고리즘을 적용하기 위해서는 FP-Tree 를 완성할 필요가 있다. 때문에 vector 에 담아둔 data 를 createFPtree 에서 옮기는 작업을 수행한다.

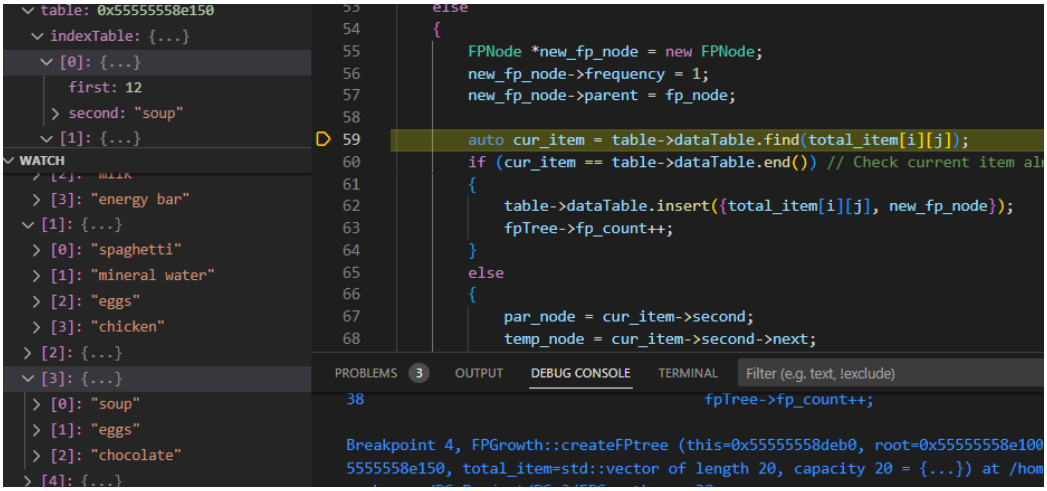



Vector 인 total_item 에서 가져온 값을



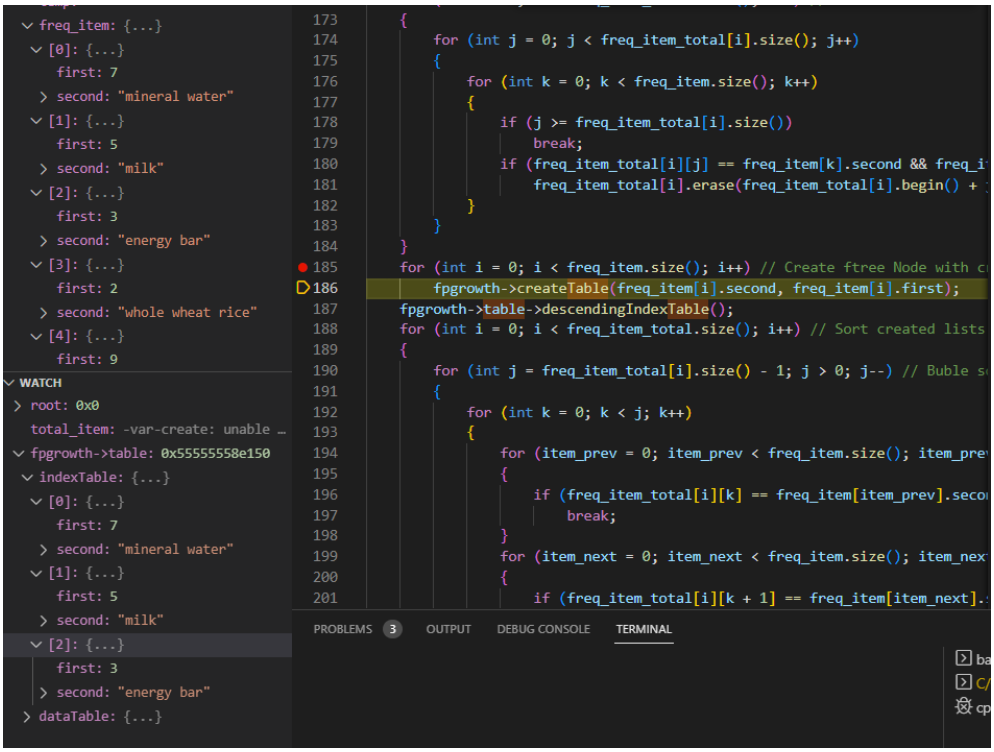

Fptree 에 삽입하고 있는 것을 확인할 수 있다.

위의 작업에는 중복 제거를 위해 다음과 같이 header table 의 정보가 필요한 경우가 있다.




Header table 은 text 파일에서 가져온 데이터를 아래의 코드에서 추가하고 있는 것을 확인할 수 있다.


다시 저 line 을 실행하면



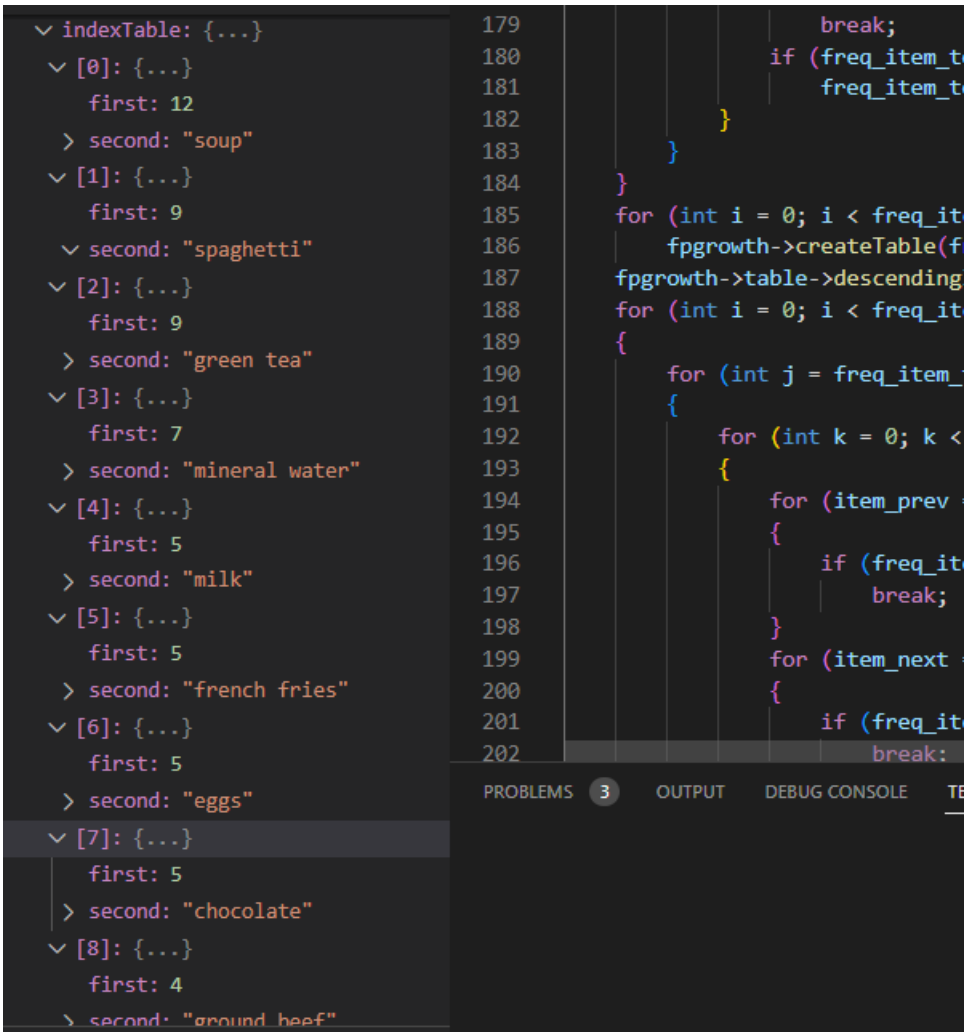

이렇게 값이 추가되고 있는 것을 확인할 수 있다. 만들어진 Header Table 을 사용해 현재 item 이 이미 존재하는지 확인할 수 있다.

- Header Table

Header Table 은 indexTable 과 dataTable 로 구성된다. Index 테이블에는 빈도수를 기준으로 오름차순, 내림차순으로 정렬을 하는 함수를 구현하는 것이 조건이다.

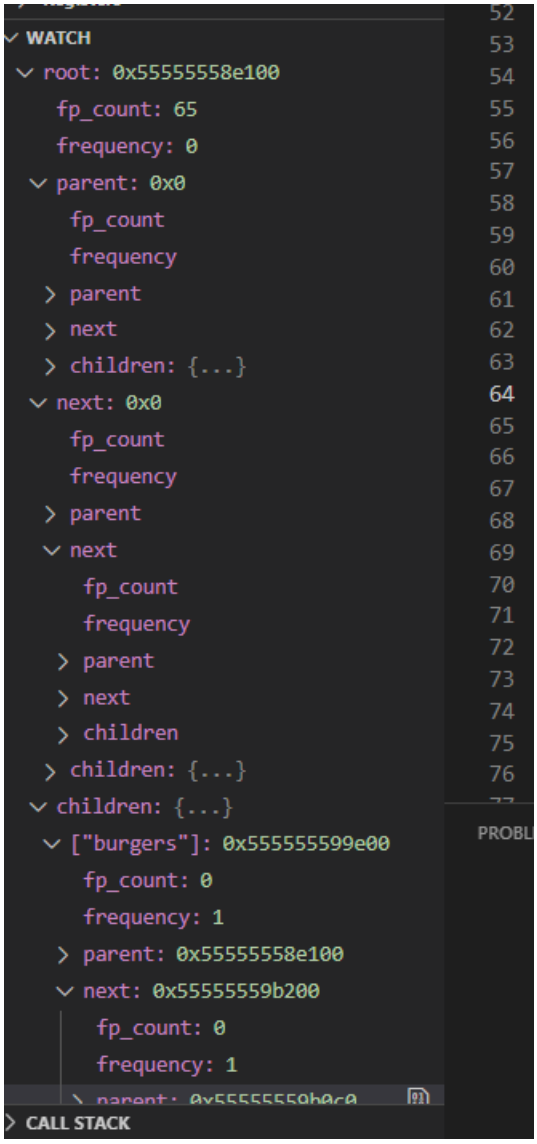
```
19      FPNode *getNode(string item) { return dataTable.find(item)->second; }
20      void descendingIndexTable() { indexTable.sort(greater<pair<int, string>>()); }
21      void ascendingIndexTable() { indexTable.sort(); }
22      int find_frequency(string item);
```

이 함수들이 위의 기능을 실행하는 함수들이다.




내림차순 정렬을 잘 실행한 것을 볼 수 있다. STL 의 sort 함수를 사용해 손쉽게 구현되어 있었다.

- FP-Tree



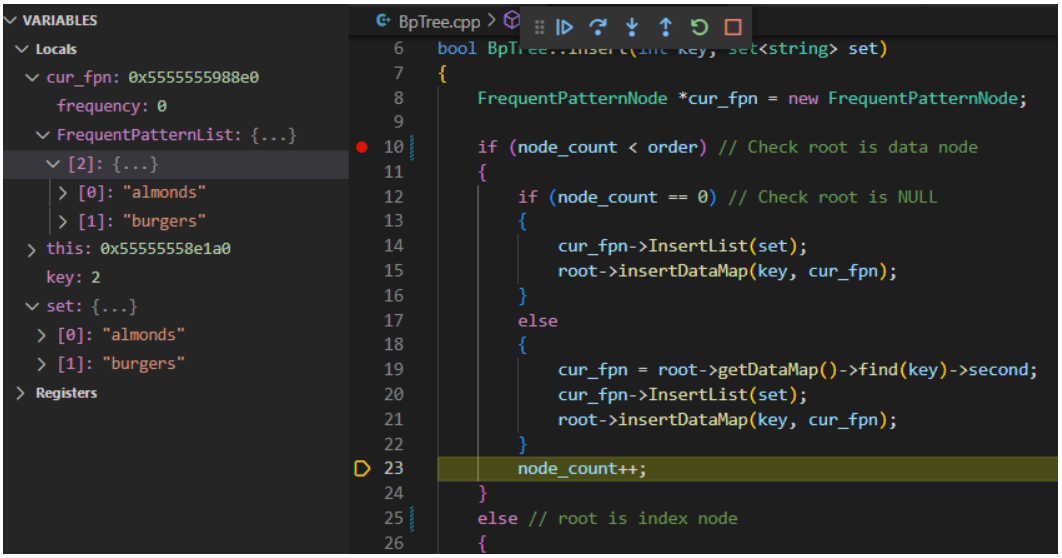

root 에서 자식 노드를 제외한 변수들은 모두 NULL 값을 갖고 key 에는 상품명, value 에는 빈도수 정보가 저장된 것을 확인할 수 있다.

- B+ Tree

B+ tree 란 b-tree 의 확장개념이다. B-tree 는 한 노드가 갖을 수 있는 자식 노드가 2 개 이상 가능한 노드들로 이루어진 tree 다. B+Tree 는 여기서 각 Node 를 indexNode, dataNode 로 나눈다. indexNode 에는 dataNode 의 위치를 찾기 위한 값, dataNode 에는 데이터들이 저장되어 있어 key 값을 이용해 원하는 data 를 빠르게 검색할 수 있다.

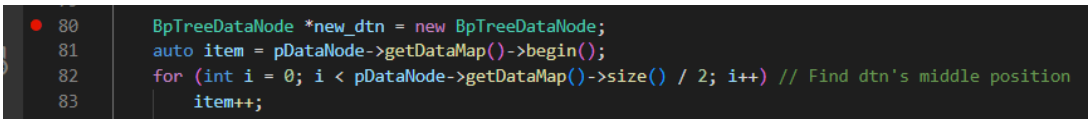
본과의 B+Tree 는 key 값, 그리고 frequentPatternNode 로 이루어져 있다.

초기에는 dataNode 와 indexNode 의 구분이 없기 때문에 dataNode 에 값이 저장되는 것을 바로 확인할 수 있다.

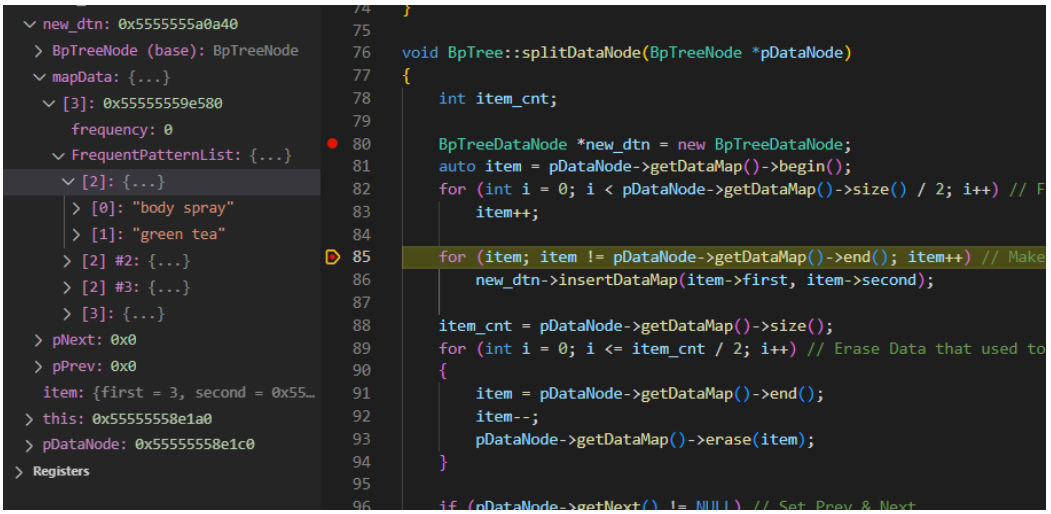



dataNode 가 채워지면서 order 값을 넘는 순간이 발생한다. 이때 splitDataNode()함수로 dataNode 를 분리한다.

우선 분리할 위치를 찾는다.

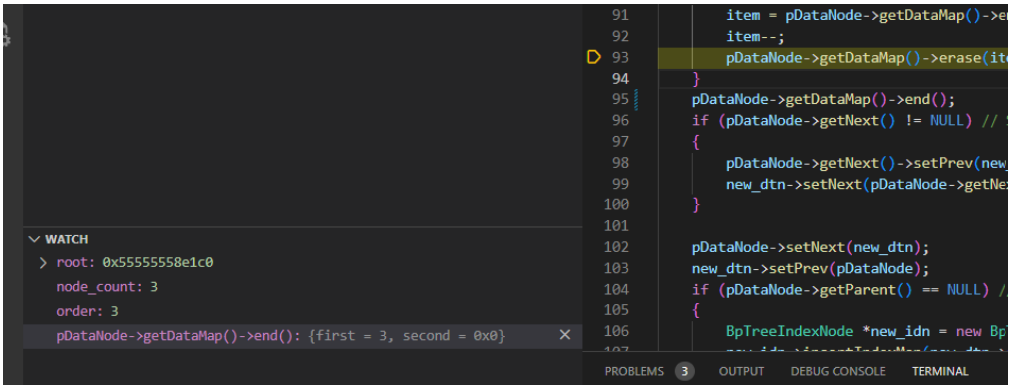


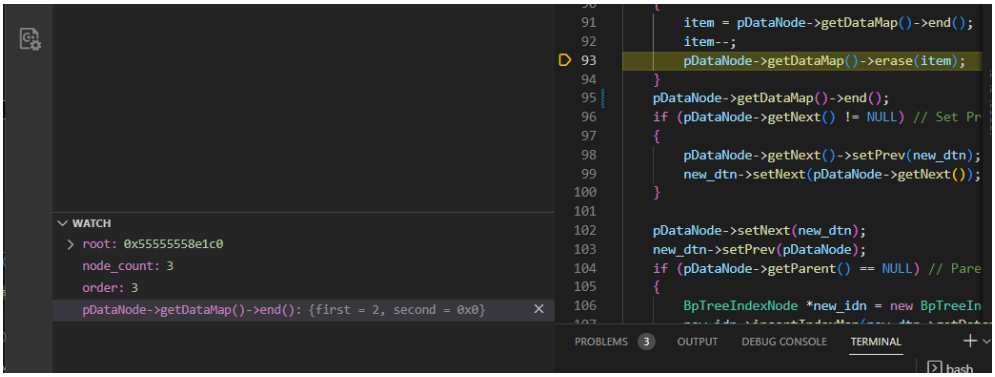

그 후 새로운 Data Node 를 만든다.



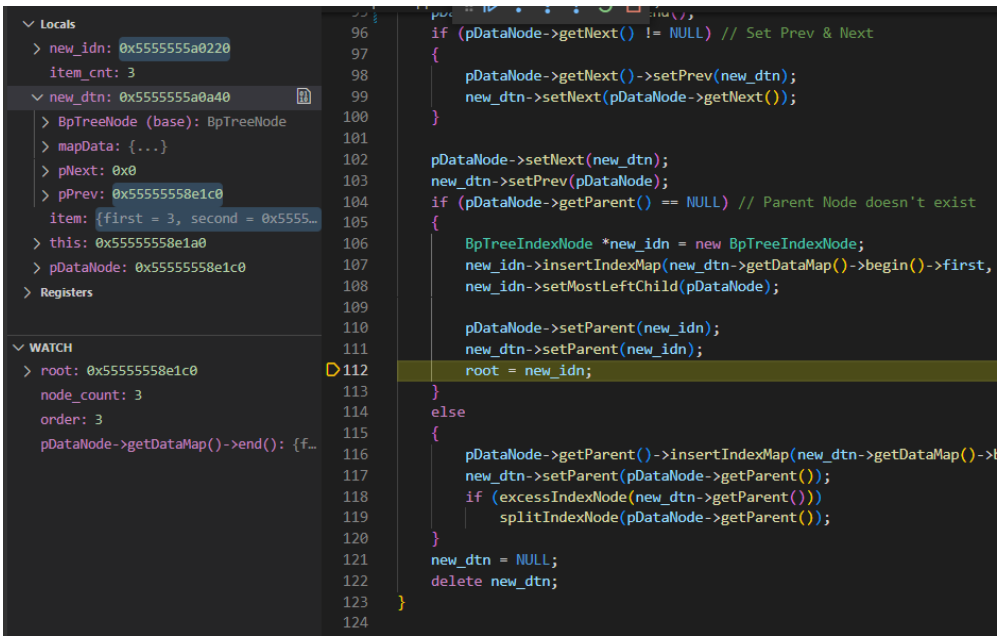

새로운 DataNode 인 new_dtn 에 값이 들어가고 있는 것을 확인할 수 있다.

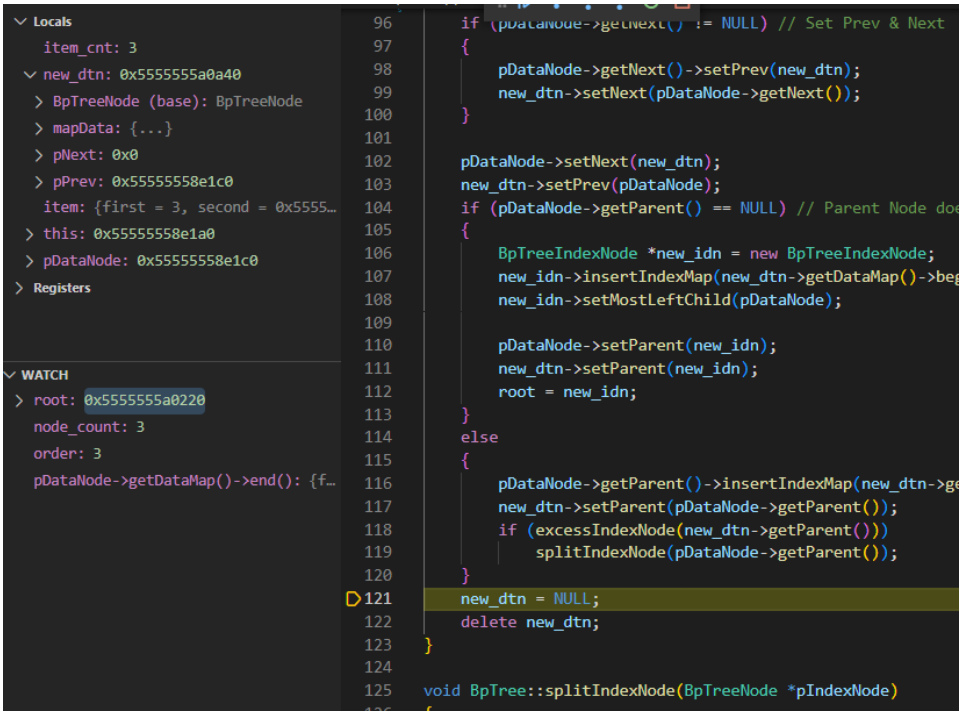
새로운 DataNode 에 사용된 data 들은 본래의 DataNode 에서 삭제한다.




Data 가 삭제되고 있는 것을 확인할 수 있다. 그 후에는 부모 노드 새롭게 생성된 노드로 바꿔준다.



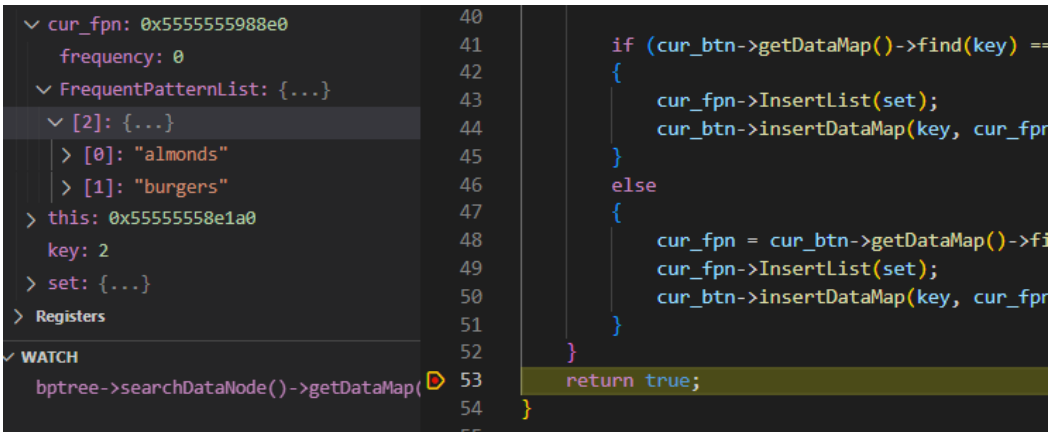



이 경우는 dataNode 개수가 order 를 넘었을 때지만 IndeNode 의 경우는 따로 있다. 차이점은 Next 와 Prev 를 설정하지 않는다는 것이고 그 외에는 dataNode 와 유사하다.

이렇게 DataNode 를 쌓아가면서 order 를 넘으면 split 하며 B+Tree 를 완성할 수 있다.

- FrequentPatternNode

B+Tree 를 구성하는 FrequentPatternNode 는 key 와 value 로 frequent pattern 길이와 원소 정보를 저장한다.

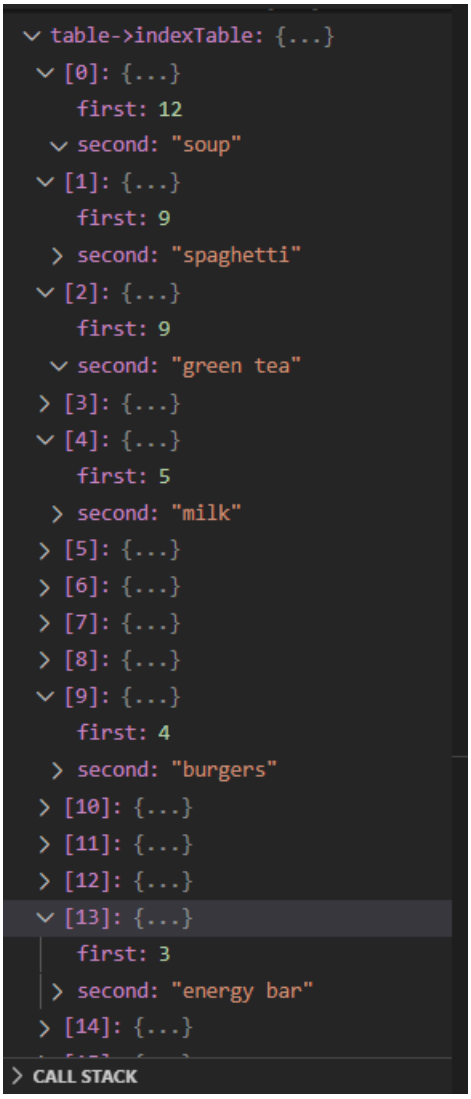



원소 정보가 set 컨테이너 형태로 저장된 것을 위에서 확인할 수 있다.

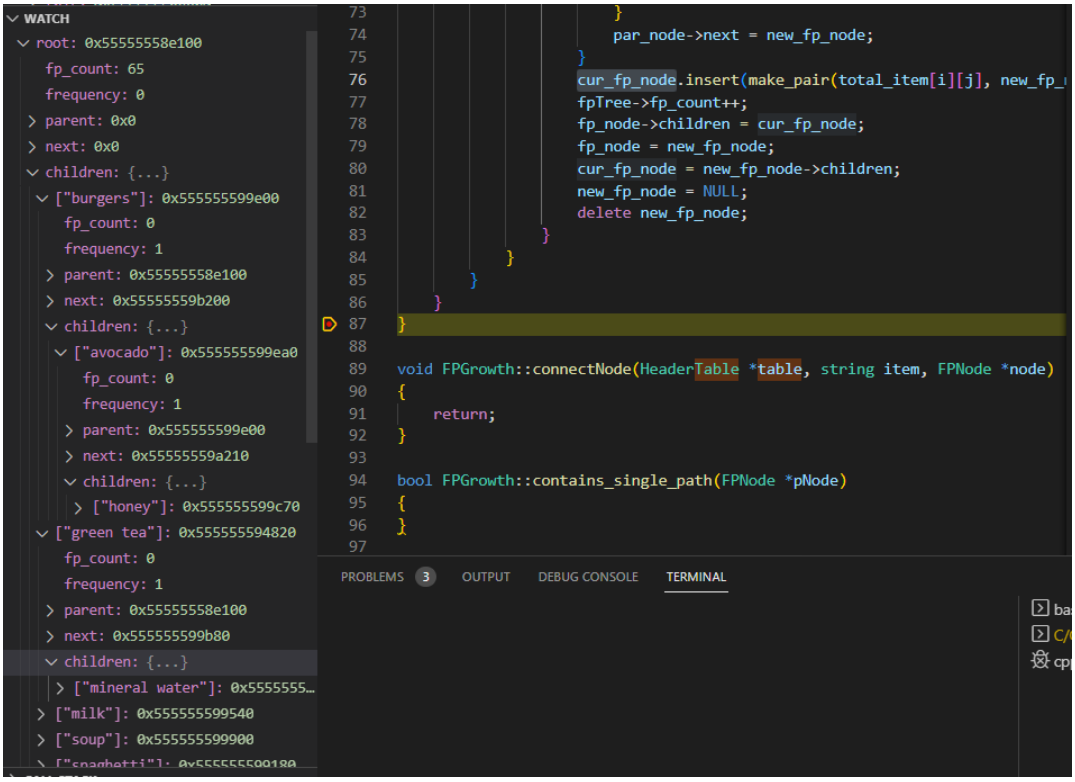
- Result Screen

  - LOAD


FP-Tree 를 만들면서 dataTable 이 완성된 것을 볼 수 있다.




Index table 도 완성된 것을 확인할 수 있다.




LOAD 작업이 종료되면 이렇게 FP-Tree 에 값이 채워진 것을 확인할 수 있다. 구현 방법은 Algorithm 목차에서 확인할 수 있다.

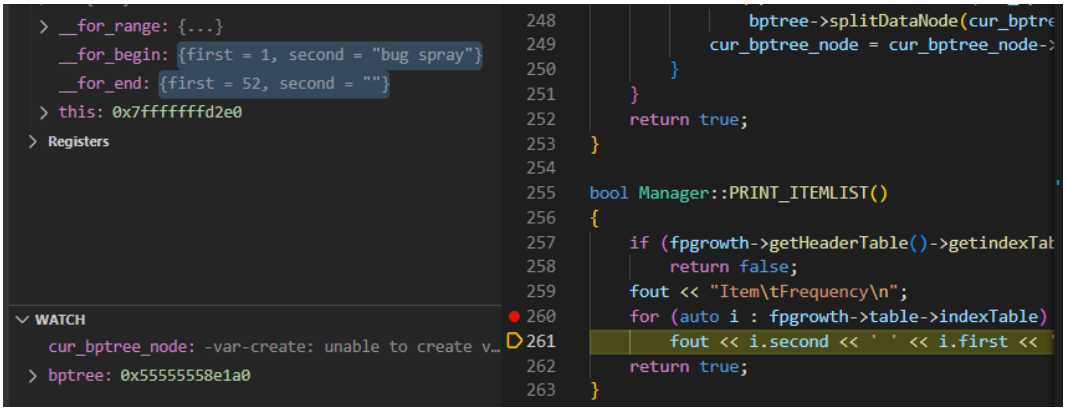
```
============LOAD============
Success
============================
```

- BTLOAD

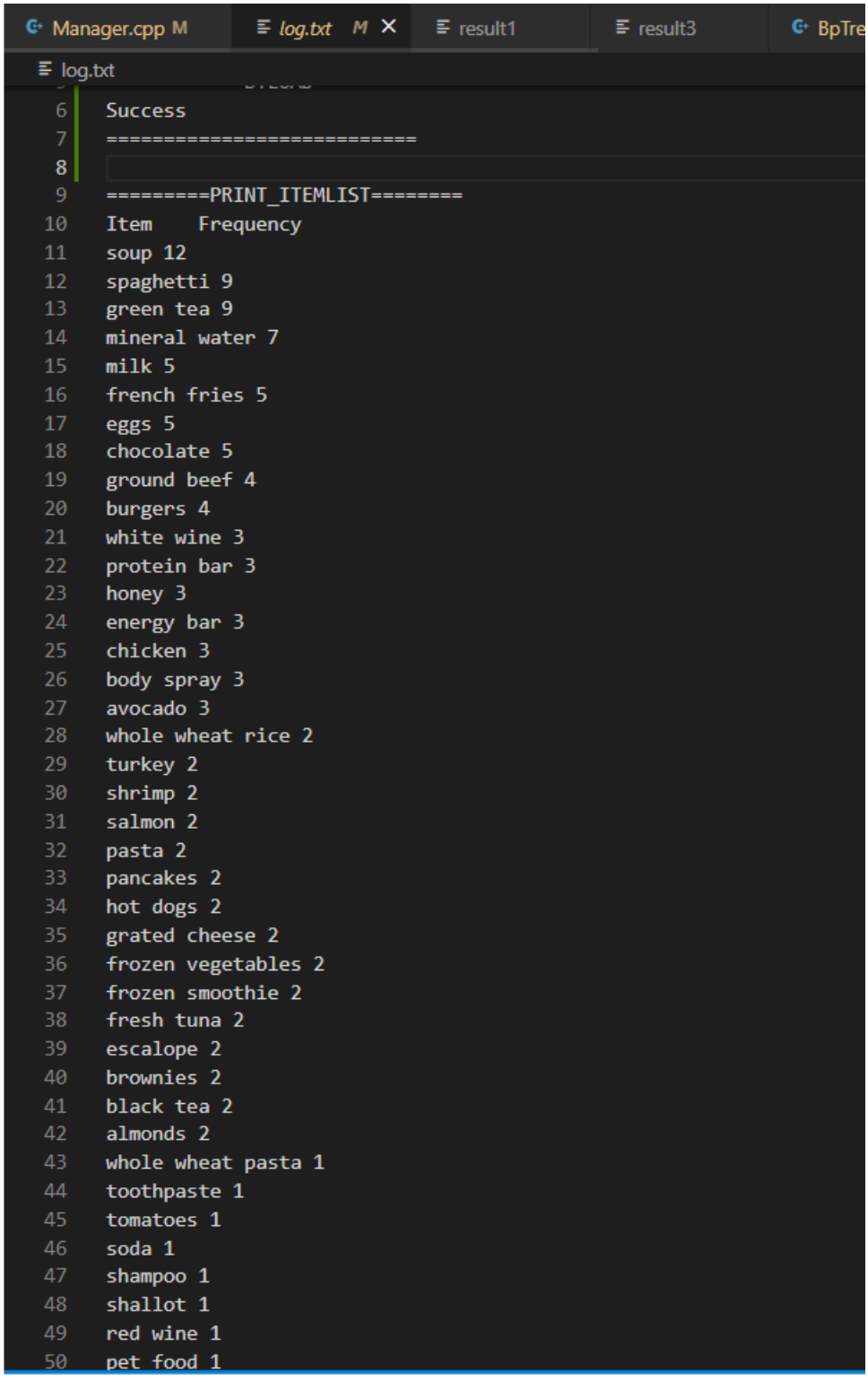
FrequenctPatternNode 가 과제에서 제시된 것과 동일하게 저장된 것, B+tree Node 에 값이 채워지는 것을 Algorithm 목차에서 확인할 수 있고 PRINT_BPTREE 에서 전체 구현 결과를 볼 수 있다.

```
============BTLOAD============
Success
============================
```

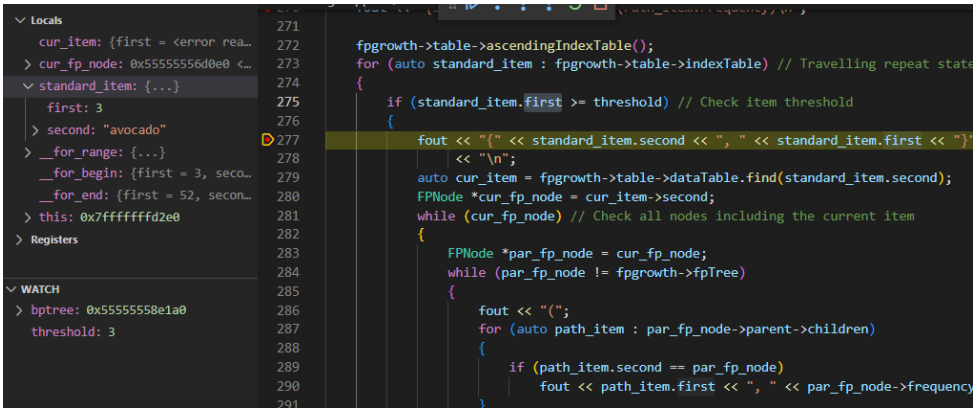
- PRINT_ITEMLIST



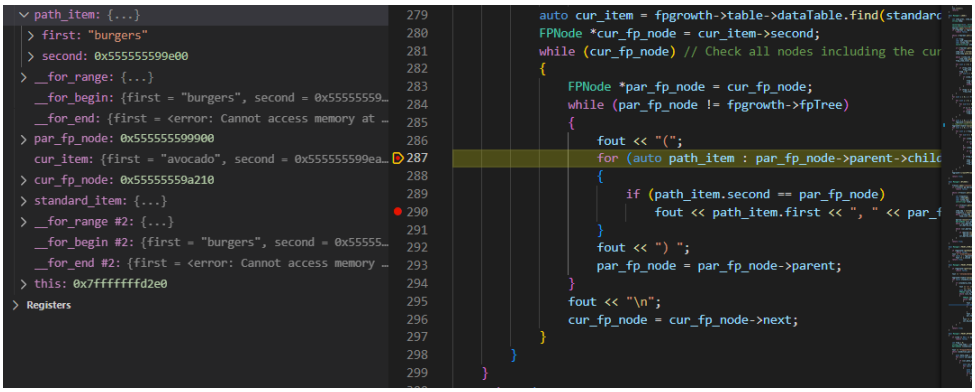

indexTable 에 담긴 data 를 순회하면서 item 이름과 빈도수를 출력하는 것을 확인할 수 있다. 여기서 auto 를 사용했는데 c++ 에서 제공하는 타입 추론 기능이다.

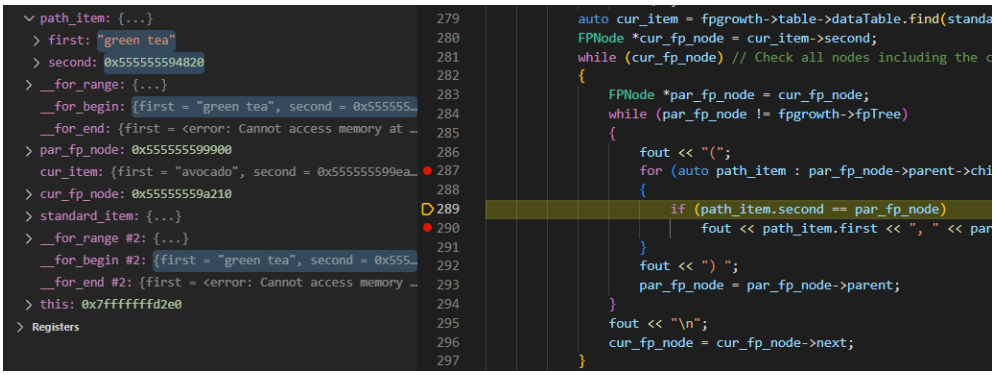



- PRINT_FPTREE

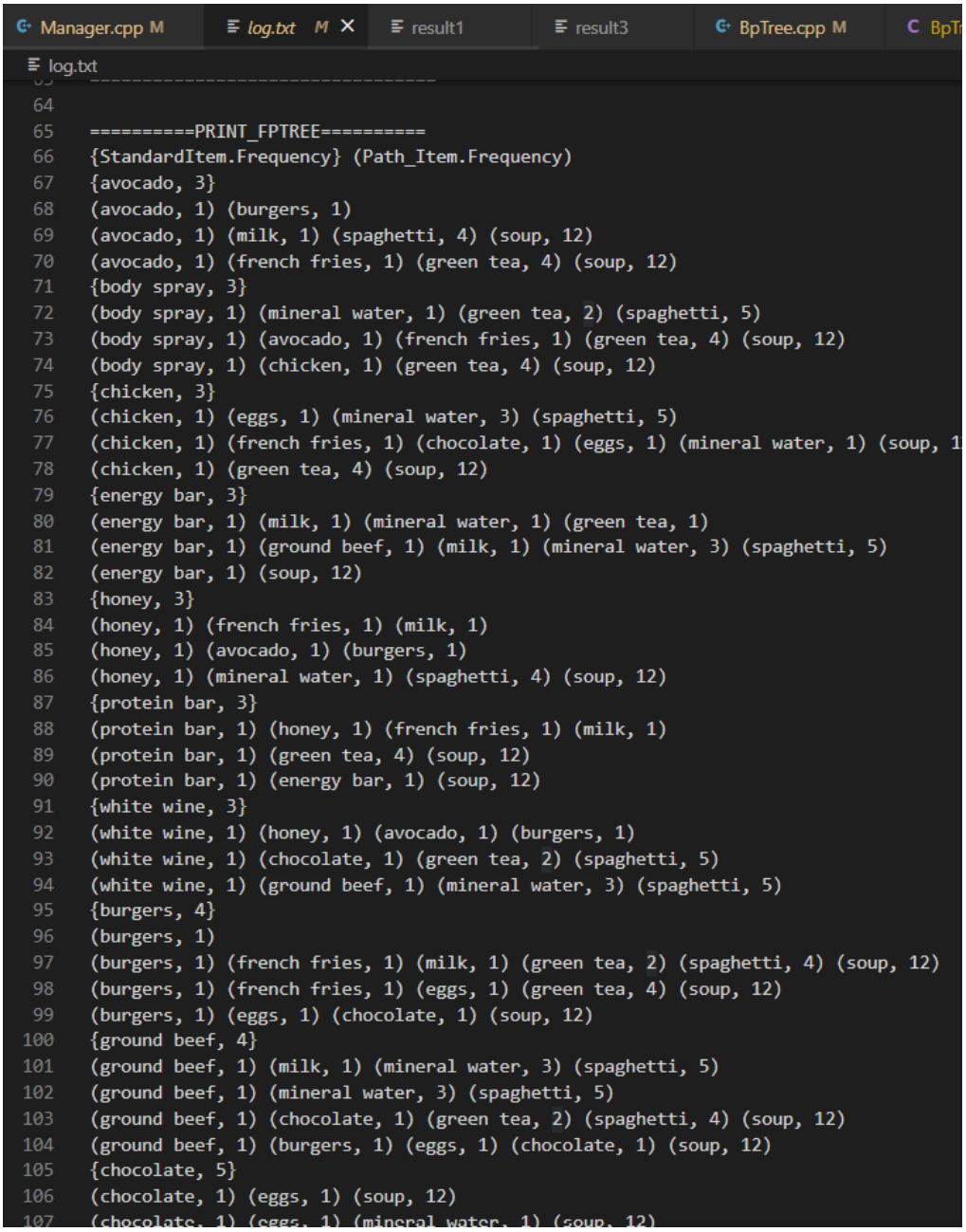



Standard_item 의 빈도수가 threshold 이상일 때 조건문 안으로 들어오는 것을 확인할 수 있다.

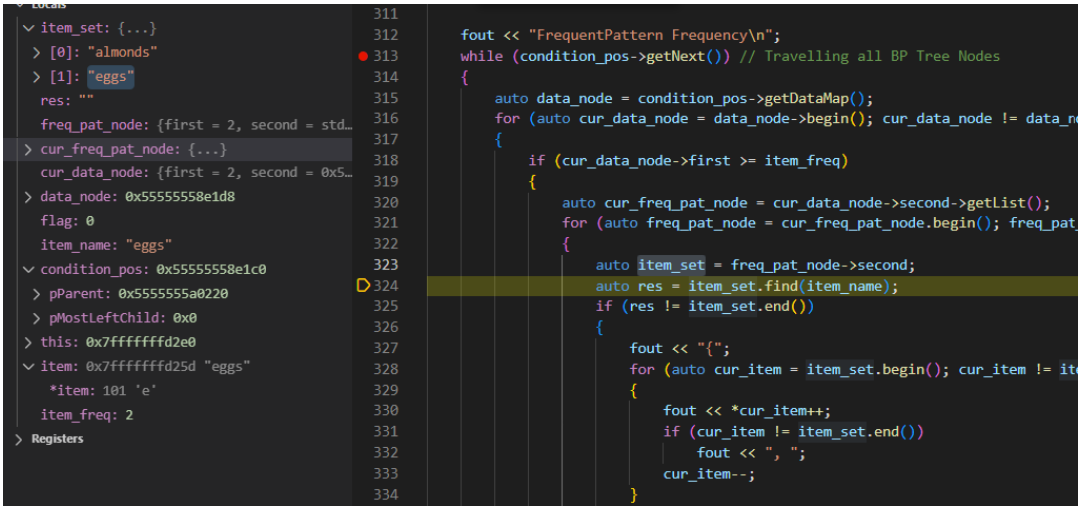


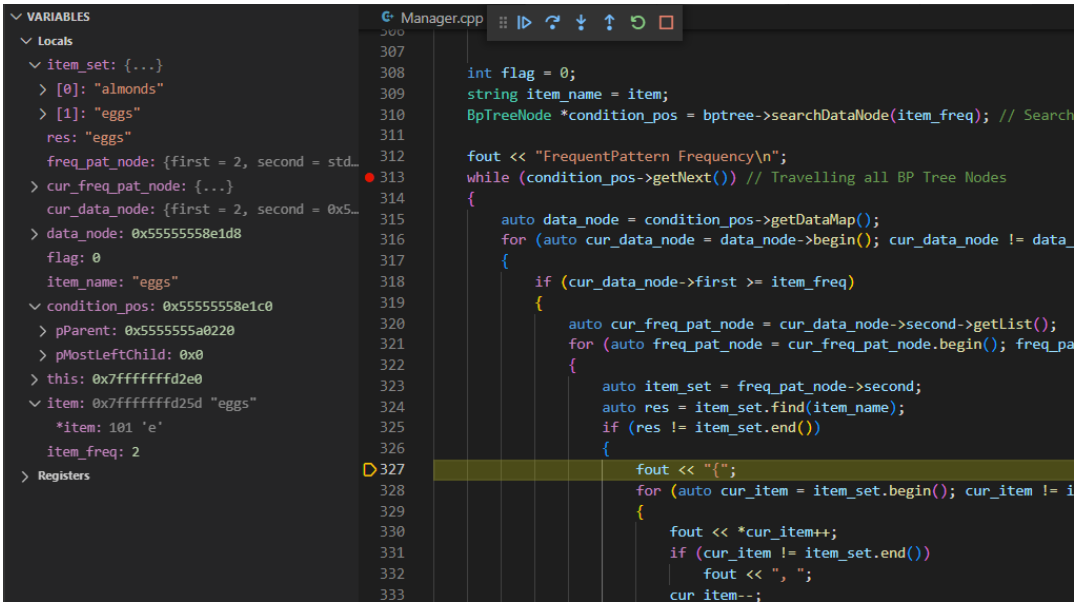

위처럼 조건을 만족하는 값들이 차례로 출력되는 것을 확인할 수 있다.




- PRINT_BPTREE



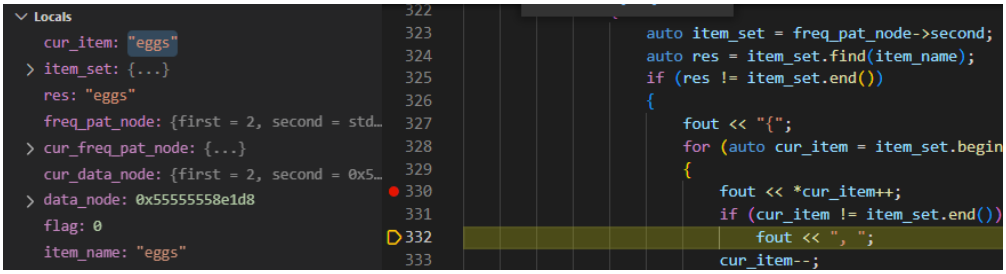

이 사진을 보면 cur_data_node 의 frequency 가 제공된 frequency 이상이고 item_set 중에서 제공된 이름과 동일한 element 를 갖는 값이 있음을 확인할 수 있다. 바로 다음으로 내려오면




이렇게 조건문을 통과했다.

빈도수 조건을 만족하는 값인 almonds 와


Eggs 가




```
==========PRINT_BPTREE==========
FrequentPattern Frequency
{almonds, eggs} 2
{burgers, eggs} 2
{chicken, eggs} 2
{eggs, french fries} 2
```

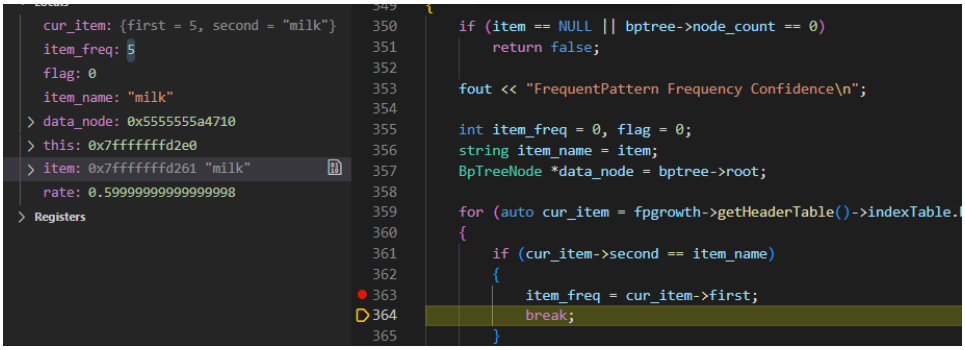
Manager.cpp M | log.txt M | result1 | result3 | BpTree.cpp M

log.txt

```
(mineral water, 1) (soup, 12)
(mineral water, 1) (spaghetti, 4) (soup, 12)
{green tea, 9}
(green tea, 1)
(green tea, 2) (spaghetti, 5)
(green tea, 4) (soup, 12)
(green tea, 2) (spaghetti, 4) (soup, 12)
{spaghetti, 9}
(spaghetti, 5)
(spaghetti, 4) (soup, 12)
{soup, 12}
(soup, 12)
================================

==========PRINT_BPTREE==========
FrequentPattern Frequency
{almonds, eggs} 2
{burgers, eggs} 2
{chicken, eggs} 2
{eggs, french fries} 2
{eggs, mineral water} 2
{eggs, turkey} 2
{almonds, burgers, eggs} 2
{almonds, eggs, soup} 2
{burgers, eggs, soup} 2
{chicken, eggs, mineral water} 2
{eggs, french fries, soup} 2
{almonds, burgers, eggs, soup} 2
{chocolate, eggs} 3
{chocolate, eggs, soup} 3
{eggs, soup} 4
================================

=========PRINT_CONFIDENCE=========
FrequentPattern Frequency Confidence
{milk, mineral water} 4 0.8
================================

==========PRINT_RANGE==========
FrequentPattern Frequency Confidence
{milk, spaghetti} 3
{milk, soup, spaghetti} 3
{milk, mineral water} 4
================================
```

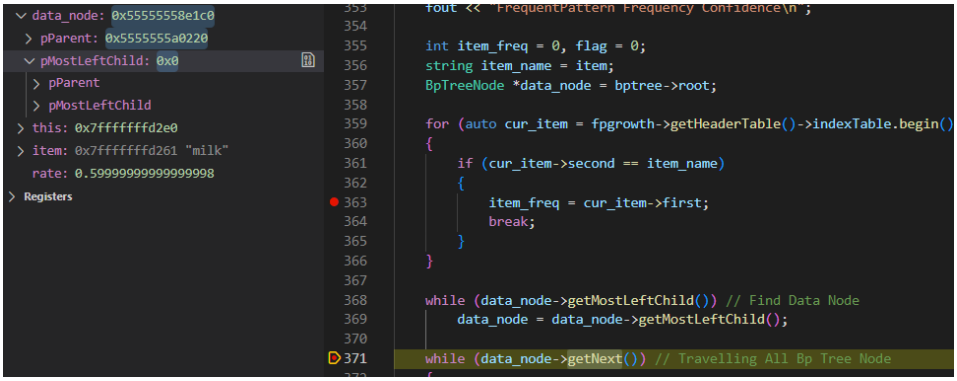
이렇게 출력된 것을 확인할 수 있다.

- PRINT_CONFIDENCE

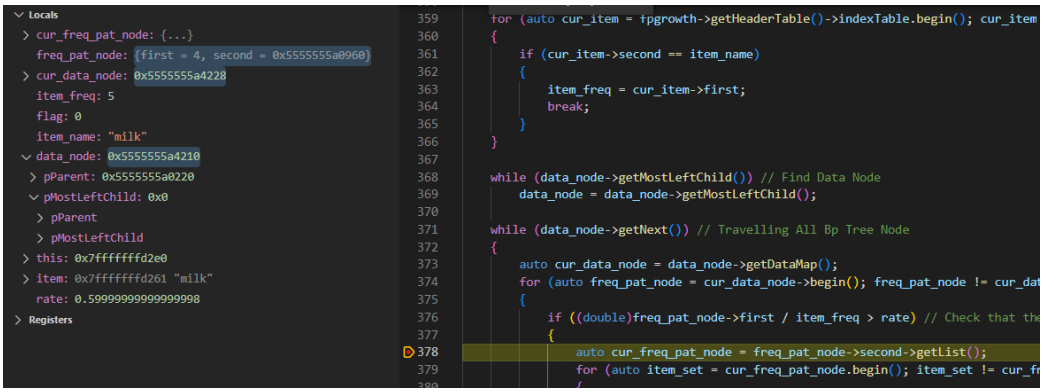



우선 찾는 item 의 빈도수를 먼저 구한다.

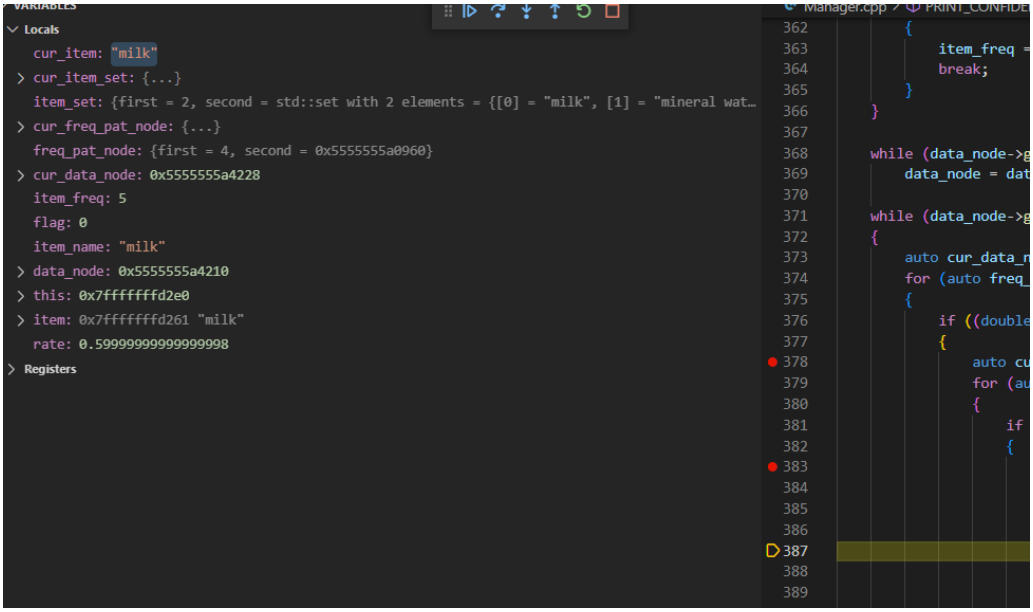
그 후 DataNode 를 찾는다.

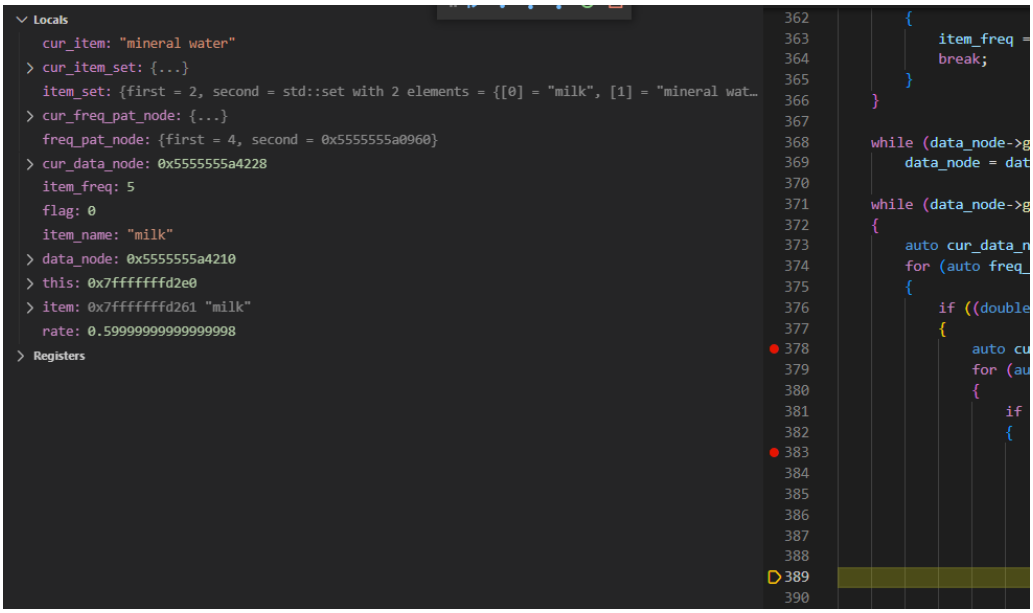



자식 노드가 없는 DataNode 에 갔음을 확인할 수 있다. 그 후 조건을 만족하는 부분집합을 찾는다.

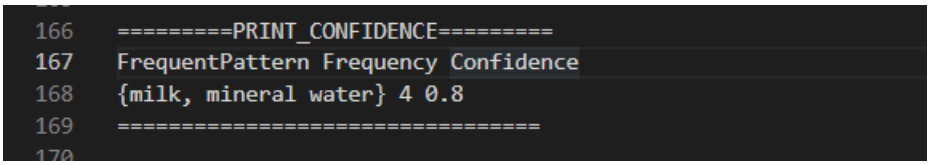



찾고 난 후 이름이 조건과 동일한 부분집합을 찾아 출력한다.

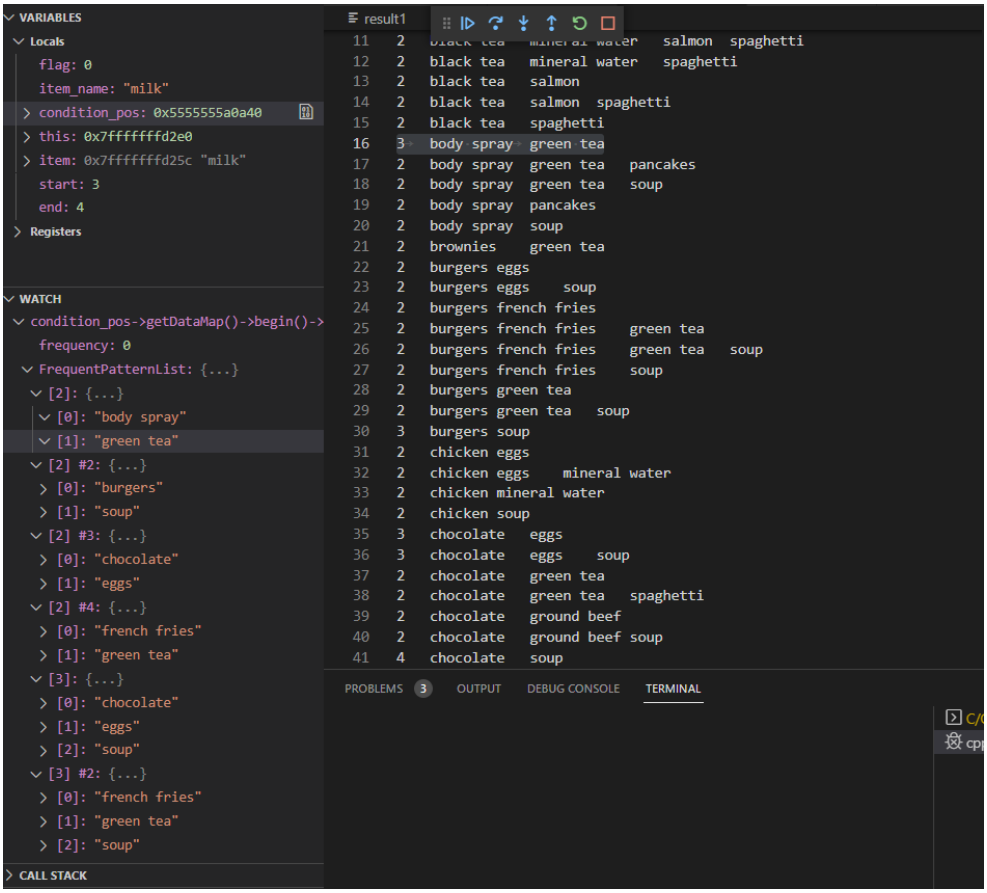


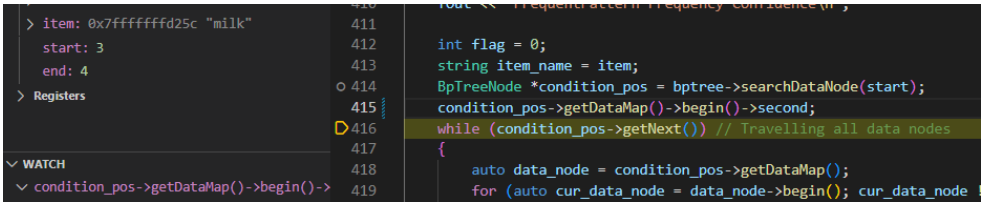

이렇게 milk 와 mineral water 를 갖는 부분집합이 출력된 것을 확인할 수 있다.



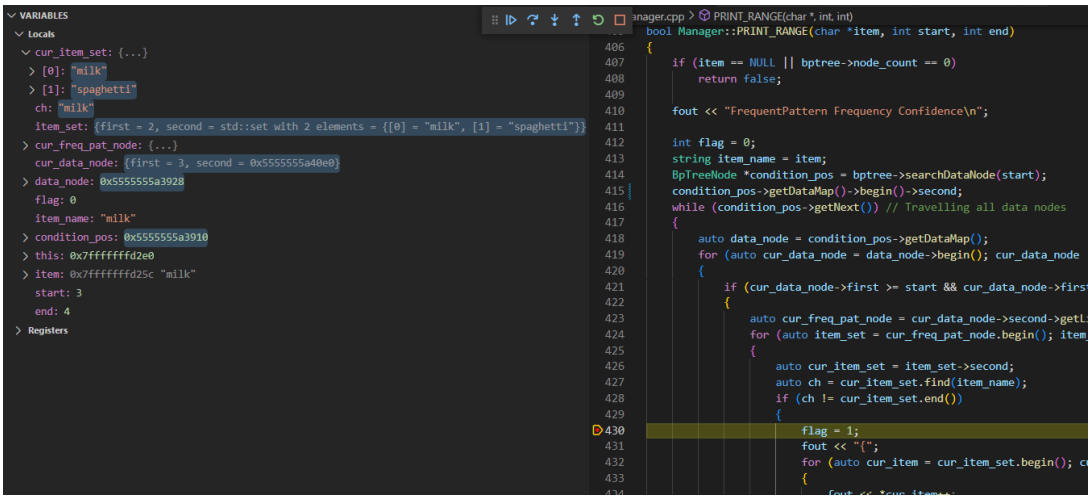

- PRINT_RANGE




우선 조건에 해당하는 DataNode 를 찾는다.




위에서 start 가 3 이상인 부분집합들을 가져오는 것을 확인할 수 있다.



여기서는 빈도수와, 이름 조건을 만족한 값들이 출력되는 것을 확인할 수 있다. 대부분이 PRINT_BPTREE 와 동일하다. 마지막으로 출력값을 확인하면 milk spaghetti 를 포함한 값이 출력된 것을 확인할 수 있다.

```
==========PRINT_RANGE==========
FrequentPattern Frequency Confidence
{milk, spaghetti} 3
{milk, soup, spaghetti} 3
{milk, mineral water} 4
================================
```

- Consideration

이번 프로젝트를 진행하면서 많은 우여곡절이 있었다.

무엇보다 B+Tree 개념을 확실히 숙지하지 못해서 구현하는데 엄청난 시간이 소요되었다. B+Tree 는 IndexNode, DataNode 가 다르고 여기서 indexNode 는 Key 값뿐만 아니라 item 인 set<string>값 또한 저장한다. 이 부분을 해당 과제에서는 상속을 통해 구현하도록 유도하였다. 우선 BpTreeNode 를 BpTreeDataNode, BpTreeIndexNode 가 상속받아 이름은 동일하지만 기능이 다른 함수를 구현해야 했다. 처음에는 이부분을 무시하고 각각 다른 노드로 구현하려다 B+Tree 의 DataNode 는 indexNode 가 될 수 있고 그 반대도 가능하다는 것을 알게 되었다. order 값을 넘어가는 순간부터 DataNode 는 order 값에 따라 분리가 되어 대략 중간 값이 indexNode 로도 올라가는 기능이 필요하다. 각각 다른 class 로 만들게 된다면 이 기능을 구현할 수가 없어서 다시 과제에서 제시된 스켈레톤 코드를 충실히 따라가서 해결할 수 있었다.

다른 사항은 map 과 multimap 의 차이였다. 과제에서 map 과 mutliamp 이 동시에 쓰였기 때문에 잘 구분해야 했다. BTLOAD 를 구현하고 난 후 디버거를 사용해 테스트를 하는데 분명 로직 상의 문제는 없지만 전체 길이가 늘어나지 않는 오류가 있었다. 이를 해결하기 위해서 따로 카운터 변수를 두거나, 모든 insert 마다 동적 할당을 해 여러 개의 map 을 사용하는 방법 또한 사용해 봤지만 기능이 동작하지 않거나 segmentation 오류를 뱉는 경우가 대부분이었다. 다시 과제 PDF 를 꼼꼼히 읽어보니 map 과 multimap 의 사용 이유가 그림으로 분명히 게시되어 있었다. 중복된 값이 발생하면은 FrequentPatternList 에 저장하는 것이었다. 그래서 map 으로 선언된 변수는 FrequentPattenNode 자료형을 가진 mapData 고 multimap 으로 선언된 변수는 FrequentPatternList 였던 것이다. 이를 확인해 값이 덮어 쓰여지던 오류를 해결할 수 있었다.

B+Tree 에 값이 끝없이 저장되는 오류도 있었다. 이 문제는 생각보다 간단했지만 구현을 전부 같은 방식으로 했기 때문에 수정하는데 오래걸렸다. 문제는 추가하는 map 의 size 를 조건으로 사용한 for 문이다. 글로 하면 어렵기 때문에 바로 다음 그림을 보면

```
for (int i =0; i < pDataNode->getDataMap()->size(); i++) // Make New Data Node
    new_dtn->insertDataMap(item->first, item++->second);
```

Map 의 최대값만큼 반복문을 돌리고 싶은데 최대값이 반복문이 돌때마다 증가한다는 것이다.

```
for (item; item != pDataNode->getDataMap()->end(); item++) // Make New Data Node
    new_dtn->insertDataMap(item->first, item->second);
```

이렇게 iterator 와 end 값을 사용해 해결할 수 있었다.

FP-Growth 와 B+Tree 에 대한 이해 없이는 진행할 수 없는 과제였다. FP-Growth 를 만들 때도 먼저 FP-Tree 를 만들고 해당 값을 토대로 FP-Growth 알고리즘을 적용하는 것을 몰라 구현에 어려움이 있었고 PRINT 기능들도 first, second 로 실수하기 쉬웠다.

PRINT 기능을 구현할 때 중복되는 코드가 상당히 많았는데, 중복되는 부분을 인자를 5 개 정도 받는 함수로 구현한다면 코드 길이를 훨씬 줄일 수 있을 것으로 예상된다.