Data Structure Project 2

컴퓨터정보공학부 2019202023 강병준

과목	데이터구조실습(수) 데이터구조설계(월 5 수 6)
교수	실습-공진흥 교수님 설계-이기훈 교수님
학번 및 이름	2019202023 강병준
전공	컴퓨터정보공학부
제출일자	2022/11/20

Introduction

데이터구조설계 과목의 2 차 프로젝트에 대한 설명이다. 자료구조인 FP-Growth 와 B+-Tree 를 구축하고, 필요한 명령어를 사용해 데이터 마이닝의 결과를 얻는다. 2 차 프로젝트에 대한 Introduction 은 구축해야 하는 2 개의 자료구조를 기준으로 진행할 것이다.

FP-Growth

FP-Growth 는 FP-Tree 구축에 앞서 상품 구매 목록(transaction)들이 저장되어 있는 market.txt 로부터 줄 단위로 읽어 Header Table 을 구성한다. Header Table 은 indexTable 과 dataTable 로 구분된다. FP-Tree 에서는 threshold 이상인 상품들만 저장하는데, indexTable 은 threshold 와는 관계없이 모든 상품과 빈도를 저장한다. indexTable 은 크게 2 가지 용도로 사용될 것이다. 해당 파일 내부에서의 각 item 들의 개수 확인 용도 및 각 transaction 을 Frequency 가 큰 순으로 정렬하여 FP-Tree 에 삽입할 때 Frequency 크기를 비교 가능하다. 이어서 dataTable 은 FP-Tree 의 map 들 속 FPNode* 즉, 각 FP-Tree 의 상품들과 table 과 연결하기 위한 Pointer 를 가지고 있는 table 이다. Fp-Tree 는 NULL 값을 갖는 root 와 함께 이와 연결되어 있는 자식 노드들이 구축되어야 한다. 후술할 Flowchart 나 Algorithm 에서 자세히 작성하겠지만, 자식 노드들의 삽입의 경우, indexTable 에서의 내림차순 Frequency 로 정렬된 transaction 의 삽입이다. FP-Tree 내부에서 같은 상품들은 pointer 로 연결되어 있으며, 해당 상품들 중 하나는 dataTable 의 Pointer 가 가르킬 것이다. 즉, dataTable 에서 chilcken 이라는 상품이 있을 때, chicken 에 대한 Pointer 를 getNext()함으로써 FP-Tree 내부의 모든 chicken 에 대한 node 에 접근이 가능하게 된다. 즉 Header Table 과 FP-Tree 를 연결되는 구조로 FP-Growth 자료 구조 구축을 완료한다
.

B+-Tree

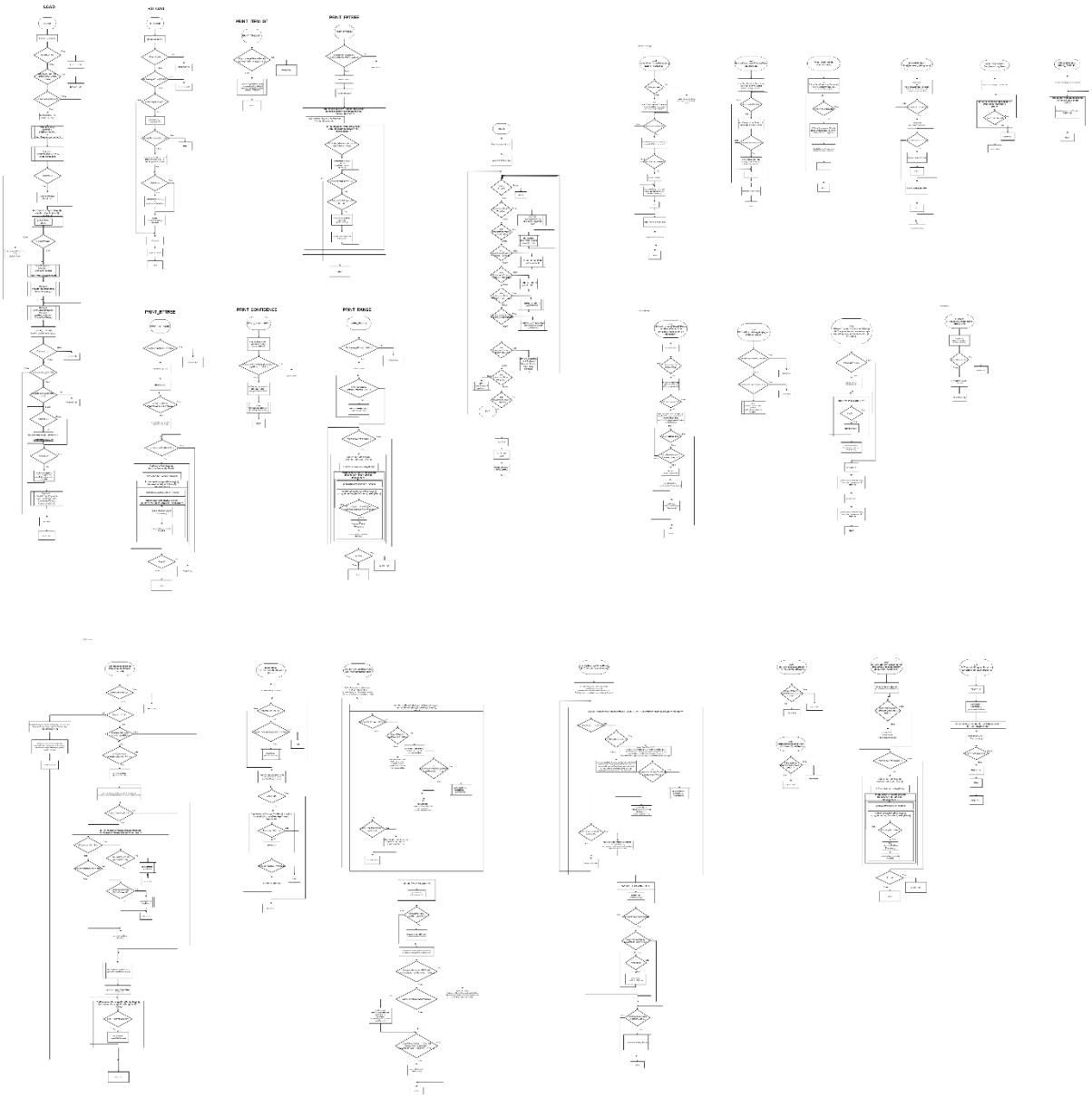
B+-Tree 는 FP-Growth 로부터 추출한 Frequent Pattern 이 저장되어 있는 result.txt 로부터 줄 단위로 읽는다. Frequent Pattern 앞에 빈도수가 주어지는데, 해당 빈도수로 dataNode 를 구성할 것이다. dataNode 의 구성은 처음 나온 빈도수에 대해서만 이루어진다. 만약 첫 번째 line 의 빈도가 3 이고 두 번째 line 의 빈도가 3 이라면, 첫 번째 line 을 읽었을 때만 dataNode 가 새로 생성되고 두 번째 line 을 읽었을 때는 dataNode 가 새로 생기지는 않지만 빈도가 3 인 dataNode 의 multimap 에 삽입된다. 또한 멤버 변수로 작성한 order 에 의해 같은 map 에 대한 dataNode 의 index 로부터 dataNode 에 대한 split 을 통해 indexNode 를 구성한다. indexNode 의 경우도 order 에 대해 같은 map 에 대한 indexNode 의 index 로부터 split 이 일어나 새로운 indexNode 를 만든다. 각 indexNode 는 가장 왼쪽 자식 노드를 가리키는 pointer 가 있기 때문에 tree 의 높이가 아무리 높아져도 dataNode 가 저장되어 있는 leaf 까지 접근이 가능하다. 이러한

알고리즘을 구축하여 B+-Tree 를 구성하고 필요한 data 를 B+-Tree 에 접근하여 dataNode 로 이동함으로써 원하는 빈도에 대한 상품 목록을 추출하는 등 원하는 data 를 mining 할 수 있다. 프로젝트에 대해 간단히 정리하자면, 저장된 상품 목록을 이용하여 FP-Growth(indexTable+ dataTable+Fp-Tree)를 구성하고, 이로부터 Frequent pattern 을 얻는다. 얻은 Frequent pattern 을 바탕으로 B+-Tree 를 구축하여 데이터 마이닝 프로그램을 구현하는 것이다.

+구현 관련

1. gitHub issue 의 답변에 따라 command.txt 에 대해 **tab 을 구분자**로 사용하였습니다.
2. gitHub issue 의 답변에 따라 같은 transaction 에 중복되는 아이템에 대한 중복된 값을 **제거**하고 진행하였습니다.

Flowchart

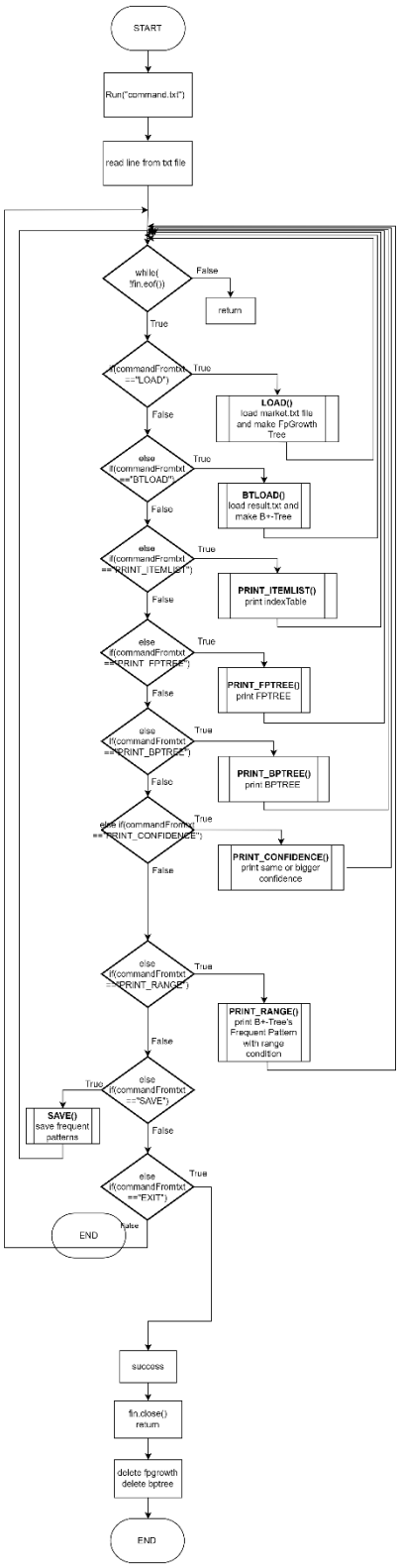


(위 이미지는 draw.io 로 직접 그린 전체 Flowchart 입니다. 모든 함수를 포함해 세로로 flow 를 만들면 매우 길어지므로 명령어 flowchart 및 함수 서브루틴 기호를 사용하여 flowchart 를 작성했습니다. 또한 flowchart 를 상세하게 작성하였기 때문에 몇몇 flowchart 는 보고서 상에서 확대를 하면 확인이 가능합니다.)

아래는 제가 직접 작성한 draw.io flowchart 링크입니다. flowchart 에 대해 확인할 수 있습니다.

https://drive.google.com/file/d/1MsjvuKB8hUGYF9t0J2lqlJDePkqh6878/view?usp=sharing

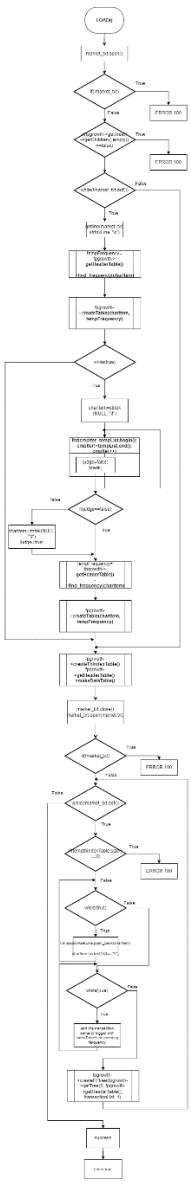
전체 흐름도-Run()



main.cpp 파일에서 제일 먼저 실행되는 Manager 클래스의 Run() 함수이다. Run() 함수는 txt 파일을 인자로 가지고 있기 때문에 command.txt 파일을 인자로 전달하여 실행한 것을

flowchart 에서 확인할 수 있다. command.txt 파일에서 getline 으로 읽어온 정보를 바탕으로 어떤 명령어가 실행될지 알 수 있다. command.txt 파일 안의 명령어는 하나만 존재하는 것이 아니므로 루프문을 통해 지속적으로 명령어의 기능을 수행하게 되는 것이다. 각 명령어마다 서브루틴 기호로 그렸기 때문에 flowchart 상에서 각 명령어에 대한 함수와 연결된다. 이를 바탕으로 마지막으로 EXIT() 함수가 실행되면 종료되도록 flowchart 의 구성을 완료하였다.

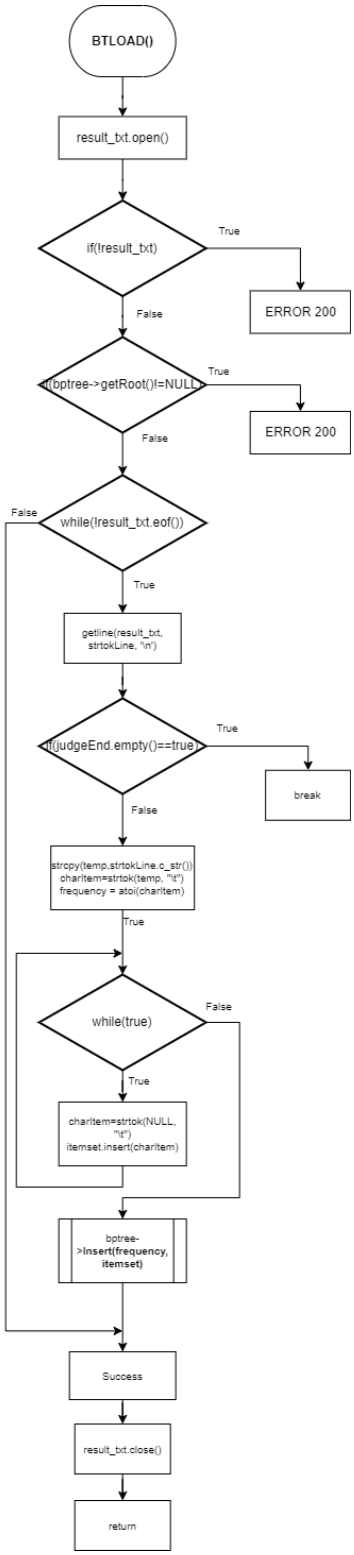
LOAD()



market.txt 를 읽어 dataTable, indexTable, FP-Tree 를 구축하는 LOAD() 명령어이다. 파일이 존재하지 않을 때의 예외처리와 함께 자료구조에 이미 데이터가 들어가 있을 때의 에러 코드 또한 출력하는 것을 flowchart 를 통해 확인할 수 있다. 전체적인 flow 는 market.txt 파일을 2 번 읽는다. 처음 읽을 때는 indexTable 와 dataTable 을 생성한다. 2 번째로 읽을 때는 처음부터

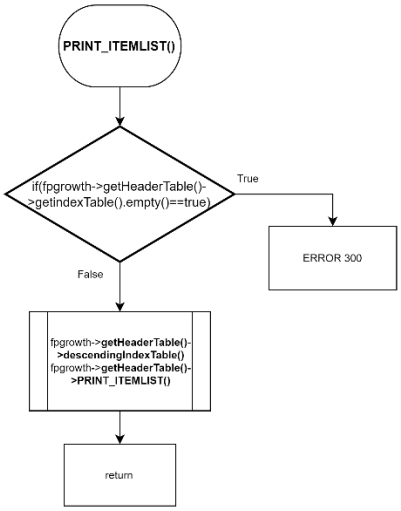
transaction 을 다시 읽으면서 앞서 처음 읽을 때의 indexTable 의 frequency 내림차순에 맞춰 transaction 을 정렬하여 fptree 에 삽입하여 fptree 를 구성하는 것을 확인할 수 있다.

BTLOAD()



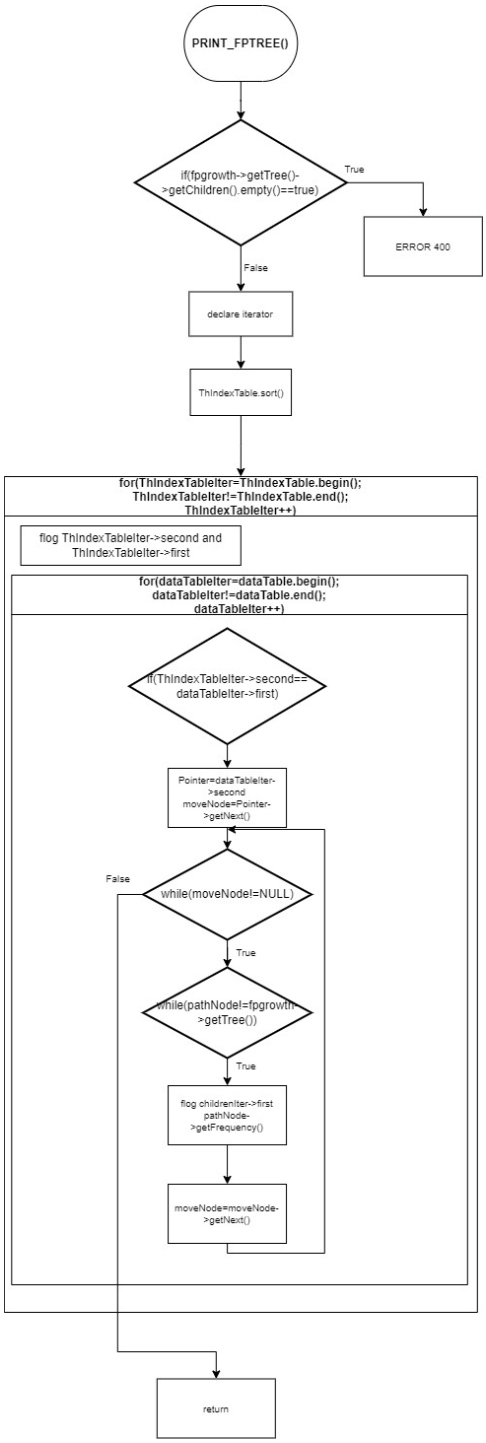
result.txt 를 읽어 B+-Tree 를 구축하는 LOAD() 함수이다. 파일이 존재하지 않을 때의 예외처리와 함께 자료구조에 이미 데이터가 들어가 있을 때의 에러 코드 또한 출력하는 것을 flowchart 를 통해 확인할 수 있다. 전체적인 flow 는 파일을 읽으면서 frequent pattern 은 set<string>에 저장하고 frequency 와 함께 bptree->Insert()로 삽입 서브루틴을 진행한다. while()문 안에서 result_txt 파일의 eof() 전까지 실행하므로 모든 line 을 읽을 때까지 getline 및 strtok 로 인자 자르기를 실행하여 같은 과정을 반복함으로써 B+-Tree 에 대한 Insert 를 진행하여 구축함으로써 BTLOAD()가 완료되게 된다.market.txt 파일을 2 번 읽는다. 처음 읽을 때는 indexTable 와 dataTable 을 생성한다. 2 번째로 읽을 때는 처음부터 transaction 을 다시 읽으면서 앞서 처음 읽을 때의 indexTable 의 frequency 내림차순에 맞춰 transaction 을 정렬하여 fptree 에 삽입하여 fptree 를 구성하는 것을 확인할 수 있다.

PRINT_ITEMLIST()



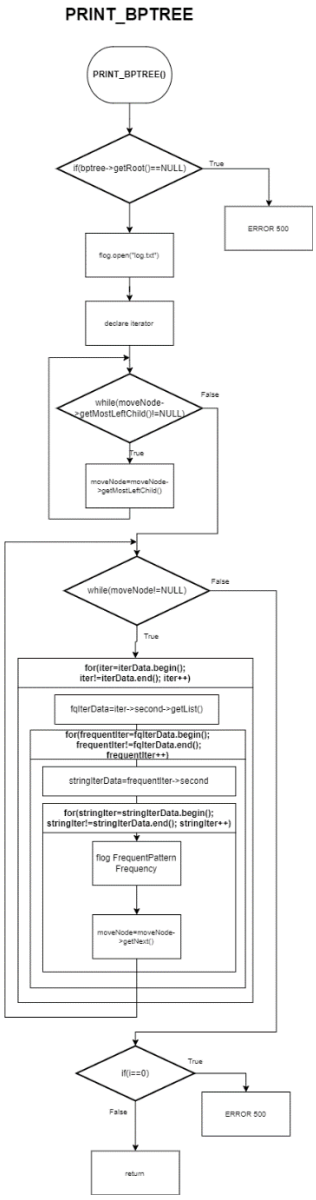
header table 에 저장된 상품을 frequency ascending 으로 출력하는 함수이다. 따라서 indexTable 에 대한 출력이다. indexTable 이 비어 있으면 에러를 출력하고, indexTable 이 존재한다면 fpgrowth 의 PRINT_ITEMLIST() 함수를 실행한다. 해당 함수에 대한 세부 구현 내용은 Algorithm 에 작성하였다. ascending 으로 출력 후 종료를 함으로써 flow 가 완료된다.

PRINT_FPTREE()



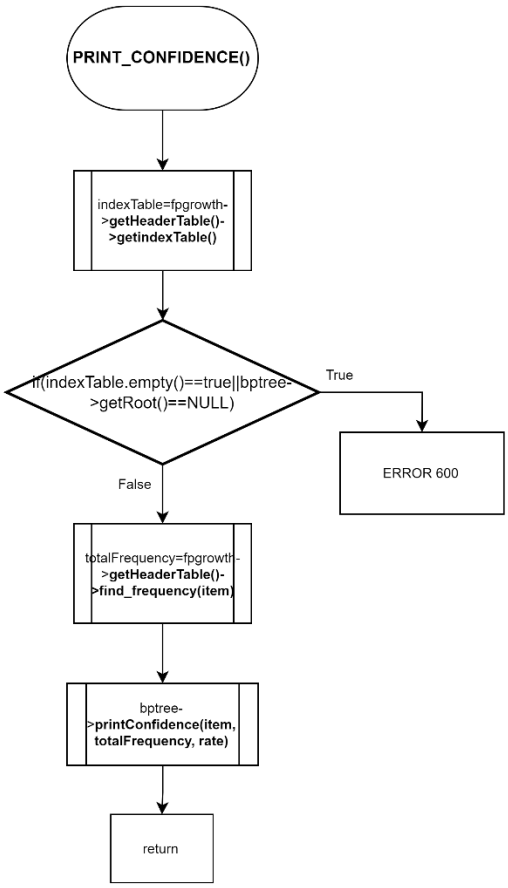
LOAD()로 구축한 FP-Tree 에 저장된 데이터를 출력하는 PRINT_FPTREE 명령어이다. indexTable 을 오름차순으로 정렬함과 동시에 iterator 반복자를 각 자료형을 매칭해줌으로써 위로 올라가면서 출력하기 위한 pathNode 와 pathNode 의 위치를 저장해주기 위해 getNext()로 이동하는 moveNode 를 사용하여 저장된 모든 데이터를 출력하면 종료되는 것을 flowchart 를 통해 확인할 수 있다. 이에 대한 세부 구현 내용은 후에 기술하겠다.

PRINT_BPTREE(char* item, int min_frequency)



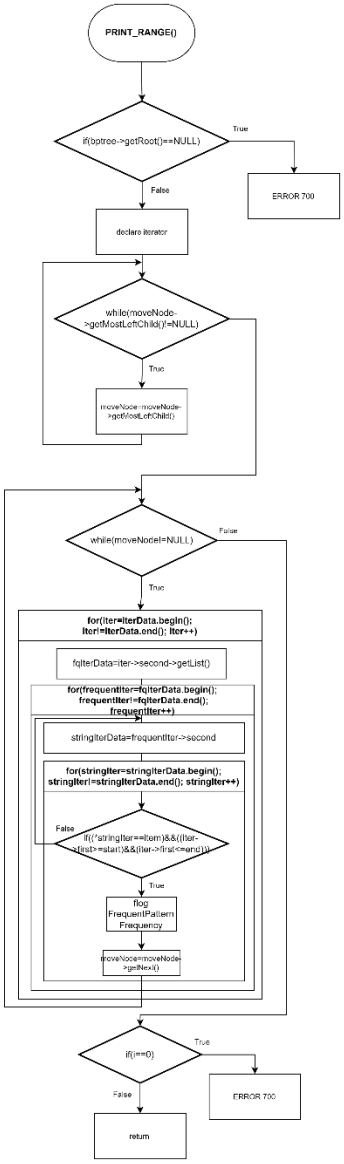
BTLOAD() 함수에 의해 구축된 B+-Tree 에 저장된 데이터들을 출력하는 PRINT_BPTREE() 함수이다. B+-Tree 가 비어 있으면 에러를 출력을 진행하며, moveNode 를 선언하여 root 를 저장해준다음, getMostLeftChild() 함수를 반복하여 가장 왼쪽의 dataNode 를 얻었다. 해당 노드부터 시작하여 모든 dataNode 가 출력될 수 있도록 노드를 이동시키면서 출력을 해주는 flow 로 구성되어 있는 것을 확인할 수 있다. 마지막에 if(i==0)이 있는 이유는, i 변수는 출력 시마다 +1 씩 증가하게 되는데 마지막에 i 가 0 이라면 frequent pattern 을 한 번도 출력한 적이 없다는 의미이므로 출력할 frequent pattern 이 없는 것에 대한 에러 출력을 위한 flow chart 를 작성한 것이다.

PRINT_CONFIDENCE(char* item, double rate)



상품명과 confidence 를 같이 입력 받아 B+-Tree 에서 해당 item 에 대해 입력받은 confidence 보다 높은 confidence 를 갖는 frequent pattern 을 출력하는 PRINT_CONFIDENCE 함수이다. B+-Tree 가 비어 있는 경우에 대한 에러 처리를 flow 에서 확인할 수 있다. 연관율은 부분집합의 빈도수/해당 상품의 총 빈도수이기 때문에 해당 상품의 총 빈도수를 얻기 위해 find_frequency() 함수를 이용하여 얻고, 이 값을 bptree 의 printConfidence 함수로 전달하여 해당 함수 안에서 출력을 진행하는 것을 flowchart 를 통해 확인할 수 있다.

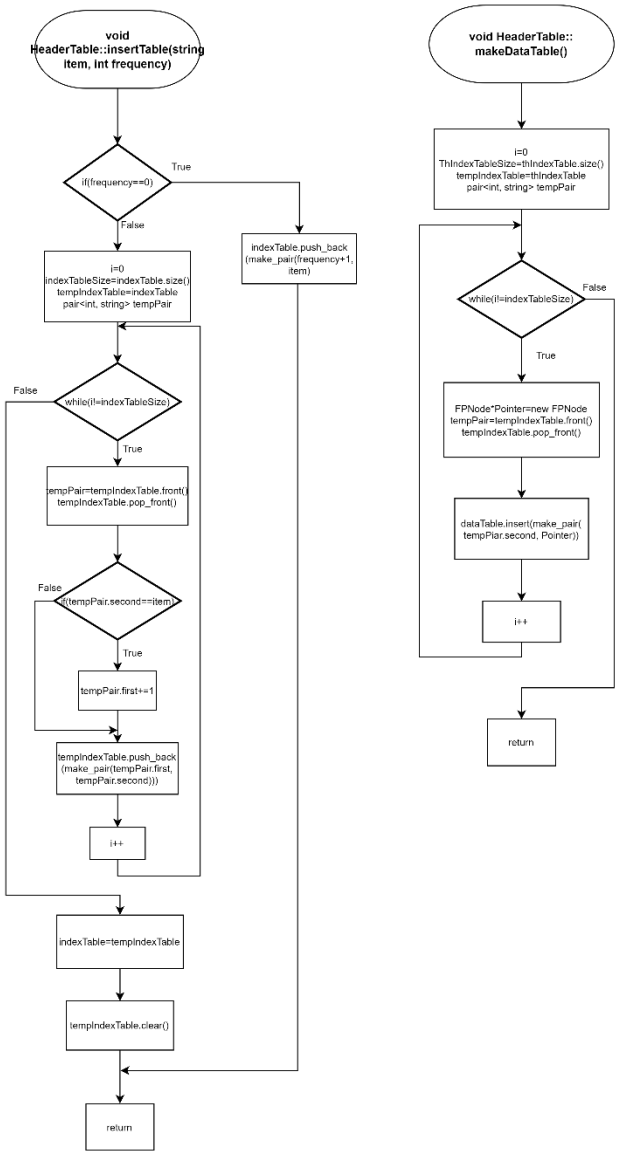
PRINT_RANGE()



구축되어 있는 B+-Tree 에 대해 원하는 빈도수 범위에 들어오는 frequent pattern 을 출력하는 PRINT_RANGE 명령어이다. 처음에 getRoot() 함수를 통해 B+-Tree 가 비어있는 확인한 후, 해당 에러를 처리한다. 출력 알고리즘은 PRINT_BPTREE 함수와 거의 유사하며, 입력 받은 빈도수 범위만 조건문으로 작성함으로써 빈도수 범위에 들어오지 않는 frequent pattern 은 출력하지 않도록 처리하고, 들어오는 frequent pattern 만 출력하는 것을 flowchart 를 통해 확인할 수 있다.

void HeaderTable::InsertTable(string item, int frequency)

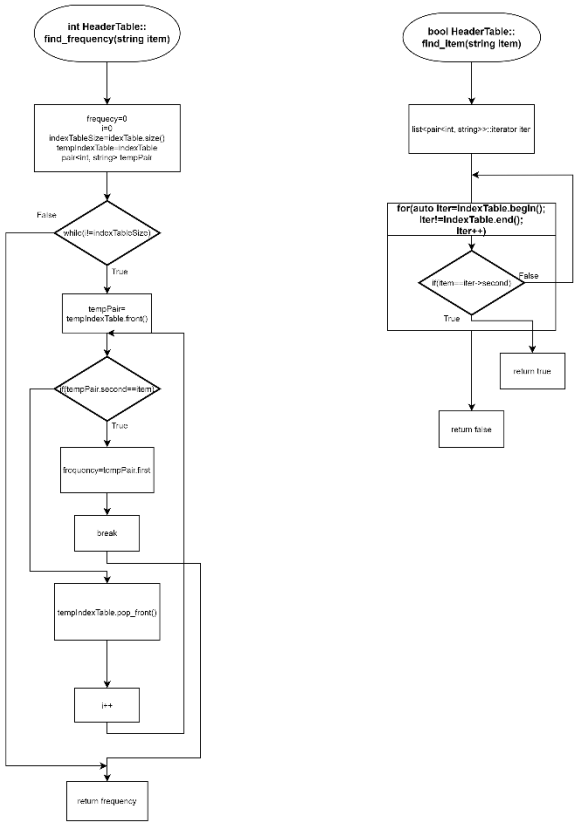
void HeaderTable::makeDataTable()



HeaderTable.cpp 에서의 indexTable 생성 함수 및 dataTable 생성 함수이다. 두 함수의 flow 가 유사하기 때문에 같이 작성하도록 하겠다. indexTable 생성의 경우 기존 테이블의 size 를 비교하면서 이미 있는 경우 frequency 를 증가시킨다. 없는 경우는 처음 frequency 가 0 인 if 조건문에 의해 indexTable 에 push_back 되는 것을 flowchart 를 통해 확인할 수 있다. dataTable 생성의 경우 구축된 indexTable 에서 threshold 이상인 item 에 대해서만 삽입을 진행한다. 동시에 pointer 도 생성하도록 flow 을 만들었다. 추후에 fp-tree 에 아이템이 들어오면 해당 pointer 와 연결될 것이다.

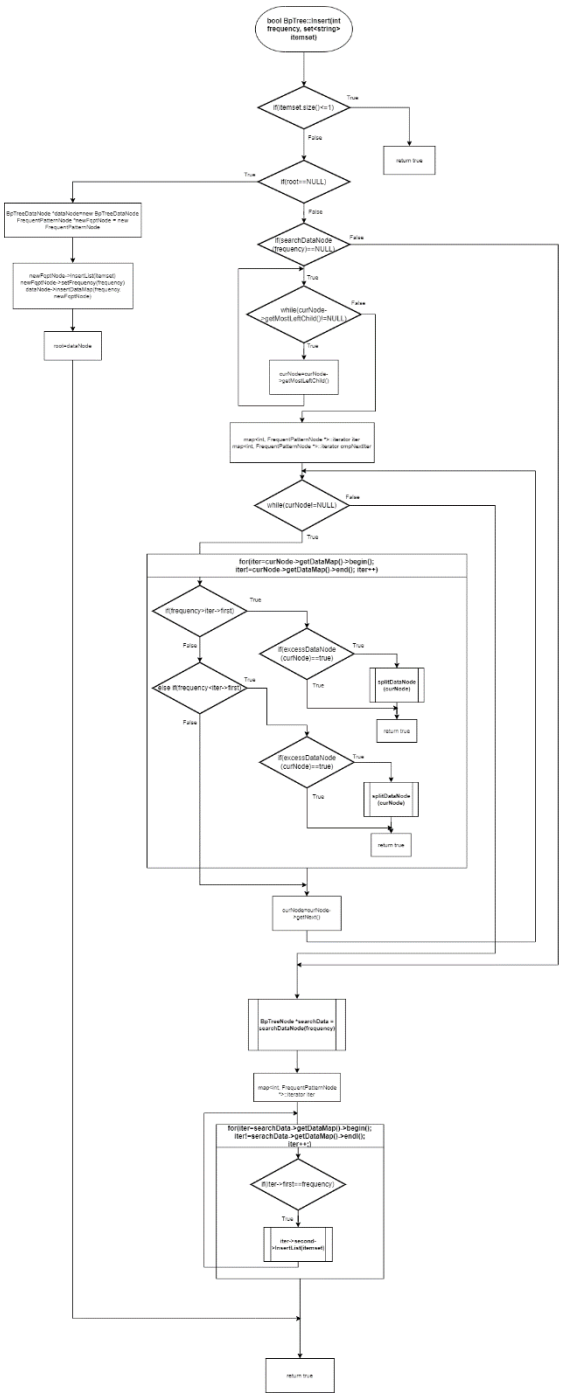
int HeaderTable::find_frequency()

bool HeaderTable::find_item()



headerTable 에서 특정 아이템에 대한 frequency 를 찾는 find_frequency 함수와 특정 아이템의 존재 유무를 반환하는 find_Item 함수이다. 탐색하여 찾는다는 관점에서 동일한 flow 이기 때문에 같이 flow 를 작성하도록 하겠다. find_frequency() 함수는 indexTable 과 indexTable 의 size 를 얻어 indexTable 에서 해당 아이템의 이름과 동일한 아이템을 탐색한다. 만약 일치한다면 indexTable 에서의 해당 아이템에 대한 frequency 를 반환한다. find_Item() 함수는 indexTable 의 처음부터 끝까지 탐색하여 해당 아이템이 있으면 true 를 반환하고, 끝까지 탐색했는데 해당 아이템이 존재하지 않는다면 false 를 반환하도록 flow chart 를 작성하였다.

bool BpTree::Insert(int frequency, set<string> itemset)

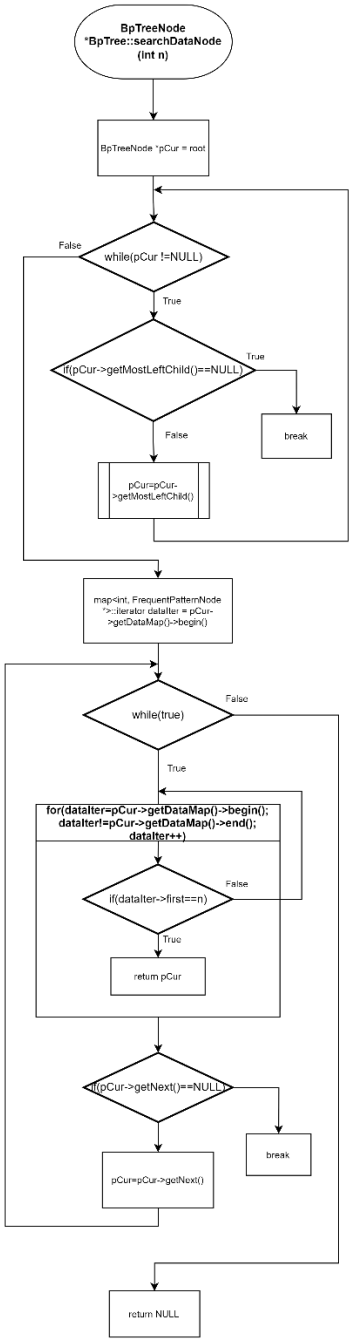


BpTree 를 구축할 때 사용하는 Insert 함수이다. 공집합이거나 원소의 개수가 하나일 경우 삽입하지 않는 것을 확인 가능하다. root 가 있는 경우와 없는 경우로 나누어 없는 경우 삽입 후 종료하고, 있는 경우에는 해당 frequency 를 가진 노드가 있는지 search 한다.

해당 frequency 를 가진 노드가 있으면 새로운 frequency 를 생성하지 않지만 없다면 새로운 frequency 를 생성한다. 이때 한 dataNode 내부의 frequency 가 order 와 같아지면 dataNode 에

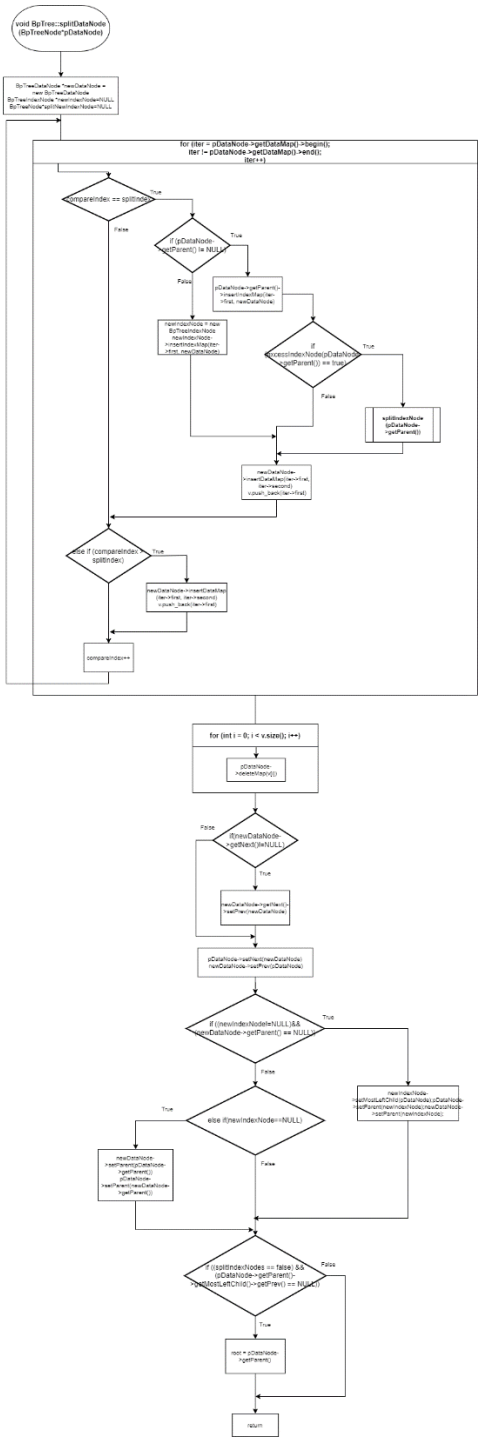
대한 split 을 진행해야 한다. 해당 flow 는 바로 다음 flowchart 에서 설명하겠다. 이로써 B+-Treed insert 에 대한 flow chart 를 작성 완료하였다.

BpTreeNode*BpTree::searchDataNode(int n)



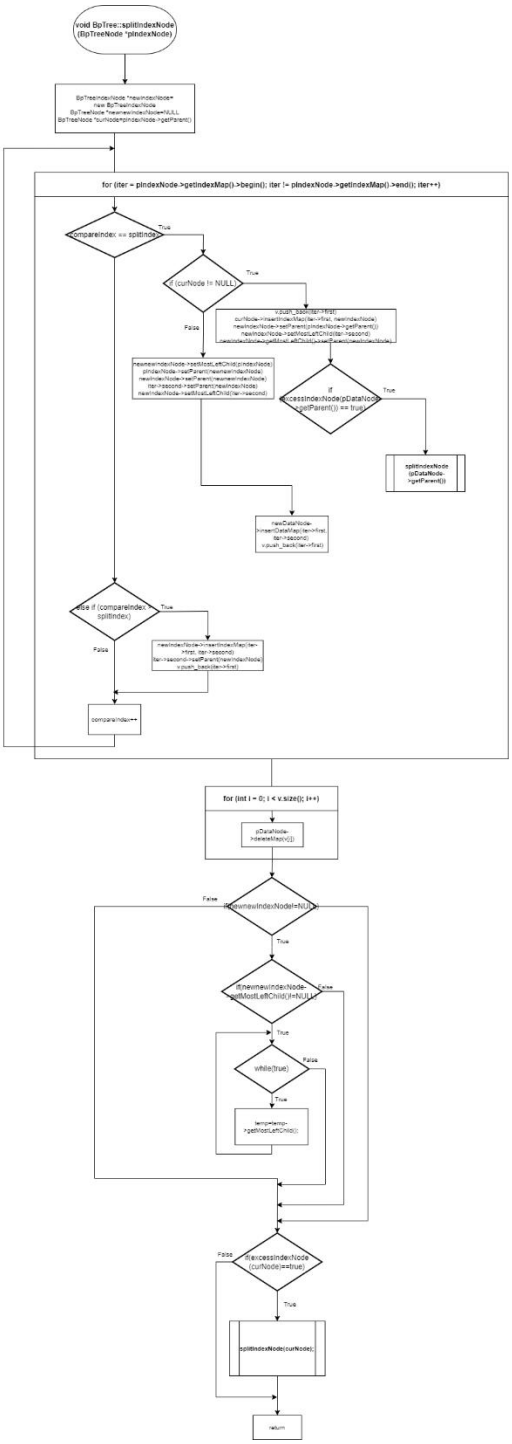
BpTree 에 insert() 함수 실행 시 이미 해당 frequency 를 가지고 있는지 탐색하고, 있다면 해당 노드를 반환하는 함수이다. 현재 노드인 pCur 을 초기값으로 root 로 설정하고 getMostLeftChild()를 반복하여 첫번째 dataNode 를 얻는다. 해당 dataNode 부터 시작하여 getNext()로 모든 dataNode 를 탐색하여 해당 frequency 를 가지고 있는지 확인하도록 flow 를 구성하였다. 만약 해당 frequency 를 가지고 있다면 해당 dataNode 를 반환한다.

void BpTree::splitDataNode(BpTreeNode*pDataNode)



앞서 flowchart 를 구성한 BpTree insert 함수에서 삽입 시에 일어나는 dataNode 에 대한 split 에 대한 flowchart 이다. 자세한 내용은 후에 기술할 Algorithm 에서 별도로 작성하였다. 따라서 간단하게 flowchart 에 대해 설명하자면, dataNode 가 split 될 때 새로운 dataNode 를 생성하여 데이터를 분할하고, 경우에 따라 indexNode 가 생성될 수도, 기존에 존재하는 indexNode 에 삽입만 진행하도록 flowchart 를 작성하였다.

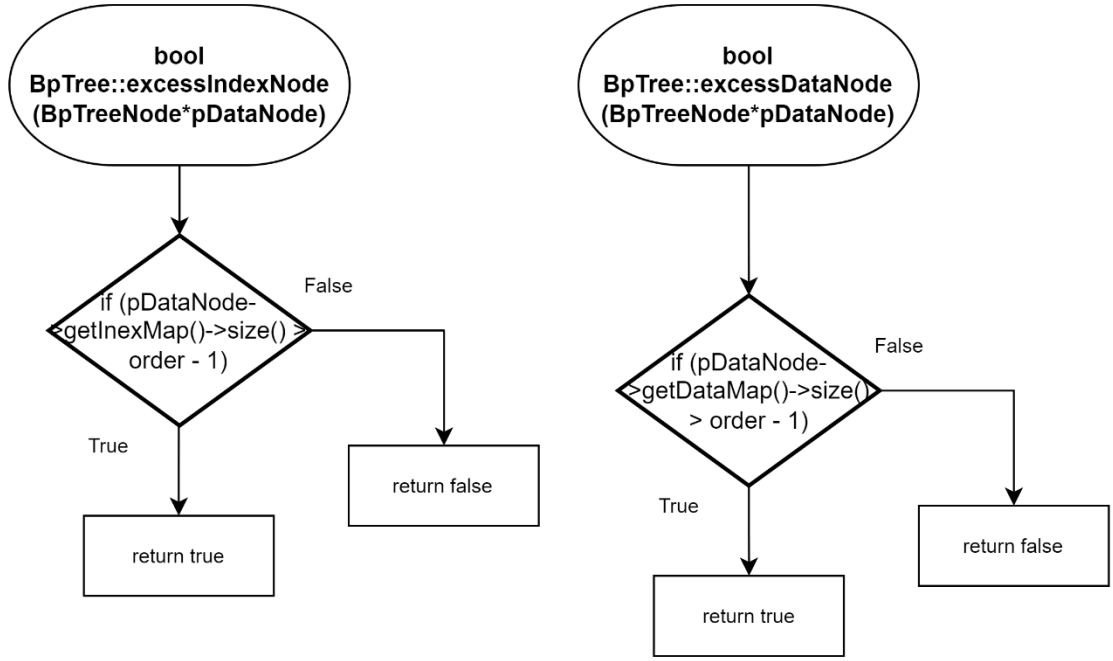
void BpTree::splitIndexNode(BpTreeNode*pIndexNode)



dataNode split 과정에서 일어나게 된 indexNode 의 split 에 대한 flowchart 이다. 자세한 내용은 후에 기술할 Algorithm 에서 별도로 작성하였다. 따라서 간단하게 flowchart 에 대해 설명하자면, indexNode 가 split 될 때 새로운 indexNode 를 생성하여 데이터를 분할하고, 경우에 따라 indexNode 의 부모 노드로 새로운 indexNode 가가 생성될 수도, 기존에 존재하는 부모 indexNode 에 삽입만 진행될 수 있다. 또한 indexNode 의 split 이 일어나면서 연속적으로 위의 indexNode 들 또한 split 이 반복될 수 있기에 마지막에 recursive 하게 flowchart 를 작성하였다.

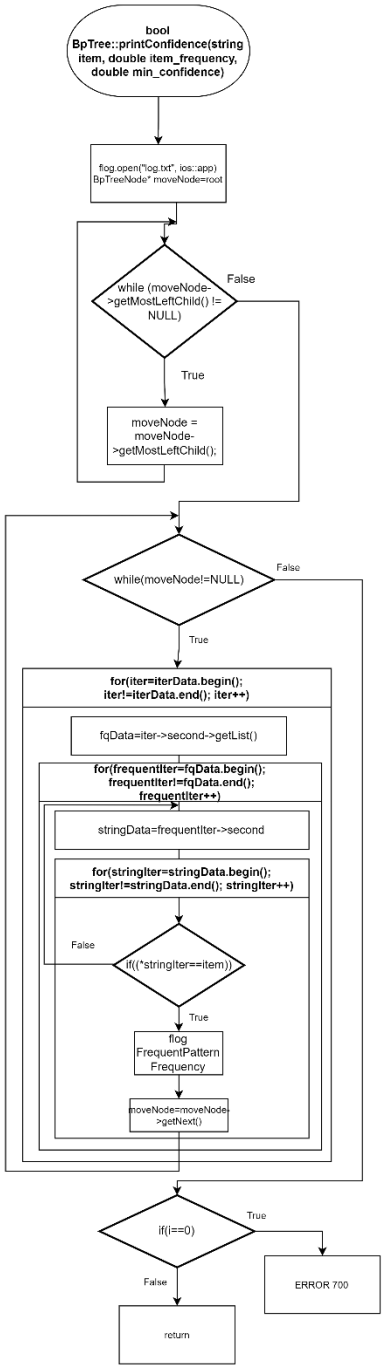
bool BpTree::excessIndexNode(BpTreeNode*pDataNode)

bool BpTree::excessDataNode(BpTreeNode*PDataNode)



dataNode 혹은 indexNode 에서 split 을 진행해야 하는지 판단하는 함수이다. order 의 수만큼 frequency 가 삽입되면 split 이 일어나야 하므로 true 나 false 를 반환한다.

bool BpTree::printConfidence(string item, double item_frequency, double min_confidence)



PRINT_CONFIDENCE 명령어를 수행하기 위해 manager.cpp 에서 호출하는 함수이다. 각 STL 의 자료형에 맞는 iterator 를 저장 후에 dataNode 를 탐색하여 해당 item 을 가지는 frequent pattern 을 찾는다. 찾은 후 indexTable 에서의 해당 item 의 총 빈도수로 나눔으로써 confidence 를 구해 만약 인자로 부여한 confidence 이상이라면 출력하는 flow 를 작성하였다.

Algorithm

파일을 읽어 명령어 및 인자들을 구분하는 Algorithm

command.txt 파일로부터 getline 으로 읽어 오기 때문에 string 으로 저장이 가능하다. 명령와 인자들을 구분하기 위해 해당 string 을 토큰화하여 자르는 방법을 사용하기 위해 strtok 함수를 사용하였다. 이번 2 차 프로젝트의 command.txt 파일은 gitHub issue 대로 탭(tab)으로 구분되어야 하기 때문에 토큰화 되는 문자는 ₩t 로 설정해주었다. 또한 strtok 함수는 char* 자료형에만 사용이 가능해 string 형에 .c_str()를 적용함으로써 char*형으로 변환해 strtok 를 " " 공백으로 구분해서 각 변수들에 저장해 인자들을 구분하였다. 이를 이용하여 인자가 모자를 경우 에러 코드 출력이 가능하게 추가적인 알고리즘을 사용 가능하다.

B+-Tree 에서 삽입 Algorithm

B+-Tree 의 삽입은 먼저 project spec 에 따라 Frequent Pattern 중 공집합 이거나 원소가 하나인 잡합은 저장하지 않는다. 따라서 Insert 함수 내부에서 itemset 의 size 가 1 이하이면 return 하도록 구성하였다. itemset 의 size 가 2 이상이라면 본격적으로 삽입을 진행하는데, root 가 없다면 처음 삽입되는 dataNode 를 root 로 설정해주었다.
그 이후 과정은 root 가 존재하는 case 로 들어가게 되는데 B+-Tree 의 삽입은 기본적으로 dataNode 에서 시작해야 한다. 따라서 searchDataNode 함수를 이용해 해당 frequency 를 가진 dataNode 가 있는지 확인한다. 해당 frequency 를 가진 dataNode 가 있으면 해당 dataNode 에 itemset 을 삽입한다. 반대로 해당 frequency 를 가진 dataNode 가 없으면 frequency 삽입을 진행해야 한다. 먼저, 현재 노드인 curNode 를 root 로 설정해주고, getMostLeftChild() 함수를 이용해 frequency 가 가장 적은, 제일 처음의 dataNode 의 위치로 이동한다.
여기서 새로 들어온 frequency 가 현재 dataNode 의 frequency 보다 큰 경우 다음 dataNode 가 존재하는 경우에 대해 다음 dataNode 의 첫 번째 frequency 와 비교한다. 만약 다음 dataNode 의 첫 번째 frequency 가 더 크다면 현재 dataNode 에 새로운 frequency 를 삽입한다.
다음 dataNode 가 존재하지 않으면 현재 dataNode 에 frequency 삽입을 진행한다. 삽입이 이루어지게 되면 order 와 현재 삽입된 수와 비교한다. 만약 order 의 수와 같다면 dataNode 에 대한 split 을 진행한다. dataNode split 이나 indexNode split 은 후술할 algorithm 에 작성하였다.

B+-Tree 에서 dataNode split Algorithm

fptree 의 dataNode 에 대한 frequency 삽입 시 order 에 따라 split 위치가 달라지게 된다. split 위치를 얻는 공식은 데이터구조실습 강의에서 배운대로 int splitIndex = ceil((order - 1) / 2.0) + 1;로 진행하였다.

dataNode 에 대한 split 이 일어난다는 것은 새로운 dataNode 와 새로운 indexNode 가 생성된다는 의미이다. 따라서 두 노드에 대한 생성을 한 후, 새로 생긴 dataNode 에 split 할 index 이상의 요소들을 삽입한다. 여기서 dataNode split 과 indexNode split 의 차이점이 있는데, dataNode split 은 split 할 index 이상의 요소를 삽입하지만 indexNode split 은 split 할 index 의 다음 요소부터 삽입한다.
다시 돌아와서 dataNode 에 split 할 index 이상의 요소들을 삽입하는 과정에서 split 할 노드의 parent(indexNode)가 이미 존재하는 경우 그 노드에 frequency 와 포인팅하기 위한 새로 생긴 dataNode 로 insert 한다. 만약 indexNode 의 excessIndexNode() 함수가 true 를 반환하는 경우, 즉 indexNode 에 frequency 수가 order 수가 되면 indexNode 에 대한 split 을 진행한다.
indexNode 에 대한 split 은 후술할 algorithm 에 작성하였다. 만약 split 할 노드의 parent(indexNode)가 존재하지 않는 경우에는 새로운 indexNode 를 할당하여 insertIndexMap()을 진행한다. 앞서 기존의 dataNode 에서 새로 할당한 dataNode 로 정보들을 다 옮겼으면, 기존의 dataNode 에 대한 deleteMap()을 진행하여 삭제한다. 이어서 새로 생긴 dataNode 와 기존의 dataNode 와 연결을 해주고 새로운 indexNode 가 생성되었으면 두 dataNode 가 해당 indexNode 를 setParent()하도록 한다.
새로운 indexNode 에 대해서는 기존 dataNode 를 setMostLeftChild()를 통해 연결해준다.
마지막으로 indexNode 가 새로 생겼다면, 새로운 root 로 설정해주어야 할지에 대한 알고리즘을 구현해야 하는데, 이는 마지막에 indexNode 가 split 된 적 없고, dataNode 위 indexNode 의 가장 왼쪽 dataNode 의 preveNode 가 없으면 root 로 설정하도록 조건문을 작성함으로써 B+-Tree 에서 dataNode split Algorithm 구현을 완료하였다.

B+-Tree 에서 indexNode split Algorithm

indexNode 에 대한 split 은 앞서 dataNode split 에서 작성한 것처럼 차이점이 존재한다. 바로 indexNode 는 split 할 index 이상의 요소들을 삽입하는 것이 아니라, split 할 index 의 다음 요소부터 삽입한다는 것이다. 마찬가지로 split 위치는 datNode split 과 동일하게 데구실 수업 시간에 배운 ceil((order-1)/2.0)+1 을 통해 얻어 진행하였다.
먼저, split 할 indexNode 의 부모 노드가 존재하는 경우 그 노드에 frequency 와 포인팅하기 위한 새로 생긴 indexNode 에 대해 insertIndexMap()을 진행한다. 기존의 indexNode 가 부모 노드가 존재하므로 새로 생긴 indexNode 의 setParent()를 기존의 indexNode 의 getParent()로 설정한다.
반면에 split 할 노드의 부모 노드가 존재하지 않는 경우에는 새로운 부모 노드(인덱스 노드)를 생성하고 기존 indexNode 의 첫 split 위치에 해당하는 정보를 해당 노드에 insertIndexMap()을 진행한다. 새로운 부모 노드로 indexNode 가 생겼으므로 setMostLeftChild()로 기존 indexNode 를 가리키게 하고 새로 생긴 indexNode 도 새로 생긴 부모 indexNode 를 가리키게 한다. 새로운 indexNode 에 대해서는 새로 생긴 부모 인덱스 노드의 다음 데이터부터 삽입을 진행해주면 된다.

완료 후 dataNode split 과 마찬가지로 기존 indexNode 의 데이터를 분배 완료하였으므로, deleteMap()을 진행한다. 또한 루트를 설정하기 위한 조건으로 새로 생긴 부모 인덱스 노드의 getMostLeftChild()를 반복하여 dataNode 를 얻었을 때, getPrev()가 존재하지 않다면 새로 생긴 부모 indexNode 를 root 로 설정한다. indexNode 의 경우 높이가 지속적으로 증가할 수 있기 때문에 마지막에 다시 한 번 현재 curNode 가 excessIndexNode()에 넣었을 때 true 를 반환하는지 확인하고, true 라면 다시 splitIndexNode()를 실행하여 재귀적으로 split 을 진행한다. 만약 excessIndexNode()가 false 를 반환한다면 indexNode 에 대한 split 이 종료되게 된다.

market.txt 에서 같은 transaction 의 중복된 item 제거 Algorithm

해당 알고리즘은 github DSLDataStorage/DS_Project_2_2022_2 의 issue #8 에 의해 구현하게 되었다. 한 라인(transaction)에서
burgers ham eggs whole wheat rice ham french fries cookies green tea 가 있다면 ham 이 2 개가 있으므로 아래와 같이 중복된 값은 제거해야 한다.
burgers ham eggs while wheat rice french fries cookies green tea
먼저 list<string> tempList 로 임시로 저장할 list 를 선언해주었다. 해당 list 에는 해당 transaction 에서 테이블에 삽입된 아이템들을 저장해준다. strtok 로 다음 아이템을 읽고, 그 아이템이 tempList 안에 없으면, 중복된 아이템이 아니므로 테이블에 삽입한다. 하지만 읽은 아이템이 tempList 안에 있으면 해당 transaction 에서 해당 item 이 중복된 것이다. 따라서 이 경우를 무시하기 위해 한번 더 strtok 로 다음 아이템을 읽는다. 만약 다음 아이템이 없으면 종료시킴으로써 market.txt 에서 같은 transaction 의 중복된 item 을 제거하여 table 에 삽입하는 algorithm 을 구현하였다.

FP-Tree 에서 저장된 정보 출력 Algorithm

개인적으로 FP-Tree 는 구축보다 출력이 더 어려웠어서 해당 algorithm 에 대한 설명을 작성하도록 하겠다. FP-Tree 의 모든 정보를 출력하기 위해 indexTable, dataTable 이 모두 필요하다. indexTable 은 빈도수를 기준으로 오름차순으로 정렬하여 threshold 이상의 상품들을 출력하기 위해 사용하며, dataTable 은 pointer 를 가지고 있기 때문에 해당 item 과 FP-Tree 와 연결되어 있으므로 참조하기 위해 필요하다. 먼저 iterator 들을 선언하여 여러 STL 들의 반복문 사용 용도로 사용할 것이다. 정렬된 indexTable 에서 threshold 이상인 item 이름에 대해 dataTable 의 아이템 이름과 동일하게 되면 해당 아이템에 대한 dataTable 의 Pointer 를 이용할 것이다. dataTable 의 Pointer 로 시작하여 getNext()로 연결된 해당 상품들로 이동하기 위해 moveNode 를 선언해주었고, 해당 moveNode 에서 루트노드 직전까지의 경로를 출력해야 하므로 추가적으로 움직이기 위한 노드인 pathNode 를 선언해주었다. moveNode 가 이동하여 pathNode 를 moveNode 로 설정하고, pathNode 는 getParent()를 하여 path 를 따라 위로

이동하면서 정보를 출력한다. pathNode 의 parent 가 root 라면 종료하고 moveNode 는 getNext()로 같은 상품에 대한 다음 위치로 이동하게 되고 같은 과정을 반복하게 된다. 한 아이템에 대해서 출력이 끝나게 되면 마찬가지로 threshold 를 만족하는 indexTable 에서의 상품의 이름과 dataTable 에서의 상품의 이름을 만족하게 되면 pointer 를 가져오는 작업을 반복함으로써 FP-Tree 내부에 구축되어 있는 모든 정보를 출력하도록 algorithm 을 구현하였다.

Result Screen

작성에 앞서 result screen 을 통한 검증을 위해 market.txt 는 github repository 의 testcase1 으로 사용하였으며, result.txt 는 result1 을 사용하였습니다.

threshold 는 3, order 는 3 으로 진행하였습니다.

command.txt

```
LOAD
BTLOAD
PRINT_ITEMLIST
PRINT_FPTREE
PRINT_BPTREE    eggs    2
PRINT_CONFIDENCE    eggs    0.4
PRINT_RANGE eggs    2   3
EXIT
```

log.txt 에 출력되는 결과를 얻기 위해 작성한 command.txt 는 위와 같다.

LOAD Result Screen (LOAD)

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

위는 command.txt 의 line 1 을 실행한 LOAD 명령어의 Result Screen 이다.

market.txt 텍스트 파일이 제대로 존재하기 때문에 ERROR 100 이 출력되지 않고, LOAD 의 기능을 성공적으로 구현하여 Success 가 출력된 것을 확인할 수 있다. 그리고 FP-Growth 가 제대로 생성되었는지는 후에 기술할 PRINT_FPTREE 명령어를 통해 검증할 수 있다.

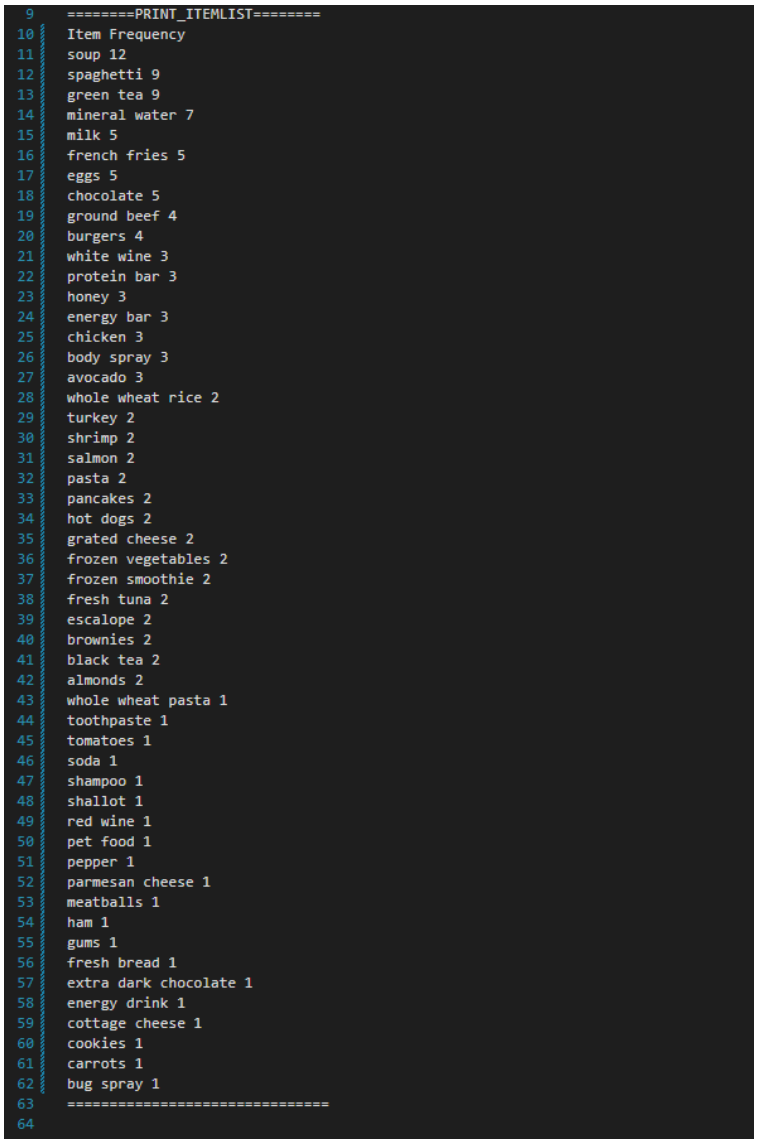
BTLOAD Result Screen (BTLOAD)

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

위는 command.txt 의 line 2 를 실행한 BTLOAD 명령어의 Result Screen 이다.

result.txt 텍스트 파일이 제대로 존재하기 때문에 ERROR 200 이 출력되지 않고, BTLOAD 의 기능을 성공적으로 구현하여 Success 가 출력된 것을 확인할 수 있다. 그리고 B+-Tree 가 제대로 생성되었는지는 후에 기술할 PRINT_BPTREE 명령어를 통해 검증할 수 있다.

PRINT_ITEMLIST Result Screen (PRINT_ITEMLIST)



```
========PRINT_ITEMLIST========
Item Frequency
soup 12
spaghetti 9
green tea 9
mineral water 7
milk 5
french fries 5
eggs 5
chocolate 5
ground beef 4
burgers 4
white wine 3
protein bar 3
honey 3
energy bar 3
chicken 3
body spray 3
avocado 3
whole wheat rice 2
turkey 2
shrimp 2
salmon 2
pasta 2
pancakes 2
hot dogs 2
grated cheese 2
frozen vegetables 2
frozen smoothie 2
fresh tuna 2
escalope 2
brownies 2
black tea 2
almonds 2
whole wheat pasta 1
toothpaste 1
tomatoes 1
soda 1
shampoo 1
shallot 1
red wine 1
pet food 1
pepper 1
parmesan cheese 1
meatballs 1
ham 1
gums 1
fresh bread 1
extra dark chocolate 1
energy drink 1
cottage cheese 1
cookies 1
carrots 1
bug spray 1
==============================

```

위는 command.txt 의 line 3 을 실행한 PRINT_ITEMLIST 명령어의 Result Screen 이다.

FP-Growth 의 Header Table 중 index Table 에 저장된 상품과 해당 상품의 빈도를 출력이 성공적으로 수행된 것을 확인할 수 있다. threshold 보다 적은 item 들도 성공적으로 출력이 진행되고 있다. 빈도 또한 내림차순으로 정렬이 된 것을 확인할 수 있다. 이로써 PRINT_ITEMLIST 명령어에 대한 검증이 완료되었다.

PRINT_FPTREE Result Screen (PRINT_FPTREE)

```
========PRINT_FPTREE========
{StandardItem,Frequency} (Path_Item,Frequency)
{avocado, 3}
(avocado, 1) (honey, 1) (white wine, 1) (burgers, 1)
(avocado, 1) (milk, 1) (spaghetti, 4) (soup, 12)
(avocado, 1) (body spray, 1) (french fries, 2) (green tea, 4) (soup, 12)
{body spray, 3}
(body spray, 1) (mineral water, 1) (green tea, 2) (spaghetti, 5)
(body spray, 1) (french fries, 2) (green tea, 4) (soup, 12)
(body spray, 1) (chicken, 1) (green tea, 4) (soup, 12)
{chicken, 3}
(chicken, 1) (eggs, 1) (mineral water, 3) (spaghetti, 5)
(chicken, 1) (chocolate, 1) (eggs, 1) (french fries, 1) (mineral water, 1) (soup, 12)
(chicken, 1) (green tea, 4) (soup, 12)
{energy bar, 3}
(energy bar, 1) (milk, 1) (mineral water, 1) (green tea, 1)
(energy bar, 1) (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)
(energy bar, 1) (protein bar, 1) (soup, 12)
{honey, 3}
(honey, 1) (protein bar, 1) (french fries, 1) (milk, 1)
(honey, 1) (white wine, 1) (burgers, 1)
(honey, 1) (mineral water, 1) (spaghetti, 4) (soup, 12)
{protein bar, 3}
(protein bar, 1) (french fries, 1) (milk, 1)
(protein bar, 1) (green tea, 4) (soup, 12)
(protein bar, 1) (soup, 12)
{white wine, 3}
(white wine, 1) (burgers, 1)
(white wine, 1) (chocolate, 1) (green tea, 2) (spaghetti, 5)
(white wine, 1) (ground beef, 1) (mineral water, 3) (spaghetti, 5)
{burgers, 4}
(burgers, 1)
(burgers, 1) (french fries, 1) (milk, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
(burgers, 1) (eggs, 1) (french fries, 2) (green tea, 4) (soup, 12)
(burgers, 1) (ground beef, 1) (chocolate, 2) (eggs, 2) (soup, 12)
{ground beef, 4}
(ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)
(ground beef, 1) (mineral water, 3) (spaghetti, 5)
(ground beef, 1) (chocolate, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
(ground beef, 1) (chocolate, 2) (eggs, 2) (soup, 12)
```

```
{chocolate, 5}
(chocolate, 2) (eggs, 2) (soup, 12)
(chocolate, 1) (eggs, 1) (french fries, 1) (mineral water, 1) (soup, 12)
(chocolate, 1) (green tea, 2) (spaghetti, 5)
(chocolate, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
{eggs, 5}
(eggs, 1) (mineral water, 3) (spaghetti, 5)
(eggs, 2) (soup, 12)
(eggs, 1) (french fries, 1) (mineral water, 1) (soup, 12)
(eggs, 1) (french fries, 2) (green tea, 4) (soup, 12)
{french fries, 5}
(french fries, 1) (milk, 1)
(french fries, 1) (mineral water, 1) (soup, 12)
(french fries, 2) (green tea, 4) (soup, 12)
(french fries, 1) (milk, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
{milk, 5}
(milk, 1) (mineral water, 1) (green tea, 1)
(milk, 1)
(milk, 1) (spaghetti, 4) (soup, 12)
(milk, 1) (mineral water, 3) (spaghetti, 5)
(milk, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
{mineral water, 7}
(mineral water, 1) (green tea, 1)
(mineral water, 3) (spaghetti, 5)
(mineral water, 1) (green tea, 2) (spaghetti, 5)
(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)
========================

```

위는 command.txt 의 line 4 을 실행한 PRINT_FPTREE 명령어의 Result Screen 이다. HeaderTable 의 오름차순으로, threshold 인 3 보다 작은 상품은 넘어가는 것을 확인할 수 있으며 FP-Tree 의 path 또한 정상적으로 출력을 하는 것을 확인할 수 있다. 개수 검증은 위에서 검증한 PRINT_ITEMLIST 와 일치하므로, 완료되었다. 또한 FP-Growth 를 생성하는 LOAD 의 기능 또한 증인 완료되었다.

PRINT_BPTREE Result Screen (PRINT_BPTREE eggs 2)

```
========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
==========================
```

위는 command.txt 의 line 5 을 실행한 PRINT_BPTREE 명령어의 Result Screen 이다. B+-Tree 에 저장된 Frequent Pattern 중 입력된 상품인 eggs 가 있으며, 최소 빈도수인 2 를 만족하는 Frequent Pattern 을 성공적으로 출력하는 것을 확인할 수 있다. Frequent Pattern 과 함께 Frequency 또한 정상적으로 잘 출력되는 것을 확인할 수 있다. 이로써 PRINT_BPTREE 에 대한 검증이 완료되었으며, B+-Tree 가 제대로 구축이 완료되었으므로 BTLOAD 명령어의 B+-Tree 구축에 대한 기능이 잘 수행되었으므로 BTLOAD 명령어의 구축 기능 또한 검증이 완료되었다.

PRINT_CONFIDENCE Result Screen (PRINT_CONFIDENCE eggs 0.4)

```
========PRINT_CONFIDENCE========
FrequentPattern Frequency Confidence
{almonds, eggs} 2 0.4
{burgers, eggs} 2 0.4
{chicken, eggs} 2 0.4
{eggs, french fries} 2 0.4
{eggs, mineral water} 2 0.4
{eggs, turkey} 2 0.4
{almonds, burgers, eggs} 2 0.4
{almonds, eggs, soup} 2 0.4
{burgers, eggs, soup} 2 0.4
{chicken, eggs, mineral water} 2 0.4
{eggs, french fries, soup} 2 0.4
{almonds, burgers, eggs, soup} 2 0.4
{chocolate, eggs} 3 0.6
{chocolate, eggs, soup} 3 0.6
{eggs, soup} 4 0.8
================================
```

위는 command.txt 의 line 6 을 실행한 PRINT_CONFIDENCE 명령어의 Result Screen 이다. B+-Tree 에 저장된 Frequent Pattern 중 eggs 와 confidence 0.4 이상의 confidence 값을 가지는 Frequent Pattern 과 해당 Frequent Pattern 의 Frequency 및 각각의 confidence 가 성공적으로 출력된 것을 확인할 수 있다. 이로써 PRINT_CONFIDENCE 명령어에 대한 검증이 완료되었다.

PRINT_RANGE Result Screen (PRINT_RANGE eggs 2 3)

```
182   ========PRINT_RANGE========
183   FrequentPattern Frequency
184   {almonds, eggs} 2
185   {burgers, eggs} 2
186   {chicken, eggs} 2
187   {eggs, french fries} 2
188   {eggs, mineral water} 2
189   {eggs, turkey} 2
190   {almonds, burgers, eggs} 2
191   {almonds, eggs, soup} 2
192   {burgers, eggs, soup} 2
193   {chicken, eggs, mineral water} 2
194   {eggs, french fries, soup} 2
195   {almonds, burgers, eggs, soup} 2
196   {chocolate, eggs} 3
197   {chocolate, eggs, soup} 3
198   ============================
199
```

위는 command.txt 의 line 7 을 실행한 PRINT_RANGE 명령어의 Result Screen 이다.

B+-Tree 에 저장된 Frequent Pattern 을 출력하되, frequency 범위에 해당한 Frequent Pattern 에 대해서만 출력을 진행해야 한다. frequency 범위를 2 와 3 으로 주었으므로 eggs 가 포함된 frequent pattern 중 frequency 가 2 이상 3 이하인 frequent pattern 만 성공적으로 출력이 완료된 것을 확인할 수 있다. 이로써 PRINT_RANGE 명령어에 대한 검증이 완료되었다.

EXIT Result Screen (EXIT)

```
200   ========EXIT========
201   Success
202   ====================
```

위는 command.txt 의 line 8 을 실행한 EXIT 명령어의 Result Screen 이다.

소멸자를 이용하여 할당된 메모리에 대한 해제를 진행 후 Success 출력되고 프로그램이 종료되었으므로, EXIT 명령어에 대한 검증 또한 완료되었다.

이로써 모든 명령어에 대한 성공적인 구현 및 검증이 완료되었다.

Consideration

이번 DS 2 차 프로젝트를 통해 FP-Growth, B+-Tree 자료 구조를 직접 구현하고 이를 연결함으로써 상품 추천 프로그램을 과제 spec 에 만족하도록 성공적으로 모든 명령어를 구현 완료 및 예외 처리를 하였다. 이미 구축해본 적 있는 자료구조가 대부분이었던 1 차 프로젝트에 비해 2 차 프로젝트의 난이도는 훨씬 어려웠다고 개인적인 생각이 들었다.

우선, 스켈레톤 코드의 경우 구축이 잘 되어 있어서 알고리즘 flow 를 짜는데 훨씬 수월하였으나 여러 STL 의 사용으로 헷갈리는 것이 많았다. FP-Growth 의 경우, indexTable, dataTable, FP-Tree 를 구축하는 것은 오래걸리지 않았으나 FP-Tree 를 출력하는 PRINT_FPTREE 명령어를 구현하는 것이 어려웠다. 앞서 PRINT_FPTREE 에 대한 상세 구현 내용과 같이 dataTable 과 연결된 포인터를 시작으로 getNext()로 이동할 때마다 root 전까지 path 를 따라 올라가서 출력하며 getNext()가 NULL 일 때까지 반복 후, 다른 item 들에게도 이러한 작업을 진행하여야 했었다. 처음에는 pathNode(root 직전까지 올라가며 출력하기 위한 노드)와 moveNode(dataTable 을 따라 getNext()로 이동하여 pathNode 를 정해주기 위한 노드)를 분리하지 않고, 하나의 단일 노드로 이동하는 방식을 구상하였다. 하지만 root 까지 출력 후, 다시 dataTable 의 첫 Pointer 로 가서 getNext() 다시 이동시켜 주어야 하는 방법이었는데 이는 코드 상으로도, time complexity 상으로도 상당히 비효율적이라는 것을 깨닫게 되었다. 따라서 옆으로 이동하기 위한 노드, 위로 이동하기 위한 노드를 분리함으로써 위로 이동하기 위한 노드를 뽑아내는데 매우 효율적이도록 만들어줌으로써 구현을 완료하였다.

반면에 이번 프로젝트에서 가장 큰 시간을 소요하고 제일 힘들었던 이슈는 B+-Tree 의 구축이었다. 처음에 구현을 시작할 때, 데이터구조설계 수업 시간에 배운 내용을 그대로 적용하여 구현을 해야 한다고 생각하니까 굉장히 막막하였다. order 에 따라서 split 되는 위치는 ceil 로 쉽게 구할 수 있었지만, dataNode 스플릿, indexNode 스플릿에서 어려움을 겪었다.

dataNode 가 스플릿 될 때, 이미 parent 가 존재하는 경우 해당 부모가 이미 있는지, 이미 있다면 또 다시 스플릿이 일어나는지 등 여러 상황을 동시에 고려해야 했다. 만약 parent 가 존재하지 않는다면 기존의 dataNode 를 제외하고 새로 생성되는 dataNode 와 새로 생성되는 indexNode 까지 생성되므로 생성되는 노드의 개수 또한 달라지게 되는 것이었다.

또한 새로 dataNode 가 생성될 때, 기존의 dataNode 에서 데이터를 옮겨 주는 것, 그리고 옮긴 후에 기존의 dataNode 에서 옮겨준 데이터를 제거하는 것과 동시에 dataNode 와 indexNode 의 STL 이 각각 multimap 과 map 으로 다르기까지 하여 매우 어렵게 느껴졌다.

이에 대해 프로젝트 spec 에 주어진 그림대로 똑같이 만들어보겠다는 생각으로 접근하였다. B+-Tree visualization 사이트를 이용하여 하나씩 삽입되었을 때의 변화를 확인하기 위해 디버깅을 통해 result.txt 의 한 transaction 이 삽입될 때의 흐름을 하나하나 직접 파악해 나갔다. 그렇게 시간을 들여서 데구실 강의자료 속 result.txt 예시와 프로젝트 github 에서 개수가 제일 적었던

result1 을 적용하여 올바른 결과를 출력하는 것을 확인하는데까지는 그리 오랜 시간이 걸리지 않았다.
하지만 검증을 위해 개수가 제일 많은 result3 를 적용하고 PRINT_BPTREE eggs 2 를 실행하니 30 번대 까지의 빈도밖에 출력이 되지 않았다. 이 상황에서는 본능적으로 개수가 많아질 때 끊기는 것을 보면 스플릿 과정을 잘못 진행했겠구나라는 생각이 들었었다. 하지만 오랜 시간을 들이고 검증을 진행하는데 스플릿 시의 새로운 노드 생성도 성공적으로 되었고, parent 연결까지 잘 되는 것을 확인하였다. 하지만 여전히 결과는 같아 이상하다는 생각에 root 부터 시작해 getMostLeftChild()를 dataNode 에 도착할 때까지 진행하여 getNext()로 dataNode 들이 잘 확인되었는지 최후의 수단으로 확인하기 시작하였다.
그런데 예상 지점에서 getNext()가 NULL 이 되어서 어느 부분에서 놓치고 있었는지 깨닫게 되었다. 스플릿 과정에서는 문제가 없었지만 스플릿 후에 생성된 새로운 dataNode 와 기존의 dataNode 를 연결하는 과정에서 놓쳤던 부분이 있었던 것이었다. 즉, 개수가 적을 때는 dataNode 가 충분히 생성되지 않아 오류를 발견하지 못하였지만 개수가 많을 때 dataNode 가 충분히 생성되었으므로 이러한 오류를 발견할 수 있었던 것이다. 분기문으로 case 를 하나 더 나누어 setNext()와 setPrev()를 꼼꼼하게 연결해줌으로써 result3 또한 성공적으로 결과를 출력하는데 성공하였다.
이렇게 B+-Tree 를 구축하고 PRINT_CONFIDENCE, PRINT_RANGE 명령어를 구현하는 것은 오래 걸리지 않았지만 가산점이 부여되는 SAVE 명령어 구현에 대한 욕심이 생겨 진행하였으나 레퍼런스도 많지 않아 데구실 강의자료와 데구설 강의자료의 그림대로 진행하면서 powerset 함수까지는 적용을 성공하였으나 이후의 과정을 완성하기에는 deadline 이 다 되어 중단하였다. 이 부분에서 개인적으로 굉장한 아쉬움이 들어 종강한 이후에 별도로 개인적으로 시도해볼 예정이다. market.txt 의 transaction 에 중복되는 상품도 존재하고 있는데, 이는 파일 제작자가 의도한 바인 것인지는 모르겠지만 github issue 상에서 중복이 존재한다면 중복 제거 후 해당 transaction 을 삽입하는 것으로 안내를 받아 중복을 제거하는 코드를 작성함으로써 데이터의 무결성을 높인 프로젝트를 구현할 수 있었던 것 같다.
이번 프로젝트를 통해 수많은 데이터에 대해 원하는 데이터를 뽑을 수 있는, data mining 에 대한 지식도 어느 정도 쌓을 수 있게 되었다. 많은 방대한 데이터를 모두 다 저장하는 것보다 중요한 것은, 이러한 데이터들을 어떻게 효율적으로 저장하고 관리할 것인지가 더 중요하다는 것을 깨닫게 되었다. 아무리 다 저장을 완료하여도 찾는데 시간이 오래 걸린다면 아무도 해당 데이터베이스를 사용하지 않을 것이다. height 를 최대한 낮추어 검색하기에 최적화되어 있는 B+-Tree 구축을 통해 실제 메모리 접근 빈도나, 탐색 시간에 있어서 확실히 효율적일 것 같다는 생각이 들었다.
마지막으로 list, map, multimap 등 여러 STL 라이브러리를 사용함으로써 각 라이브러리의

차이점과 장단점을 파악할 수 있었고, 이후에 있을 프로젝트 구현에 있어 효율적으로 알고리즘을 작성할 수 있는 계기가 될 수 있을 것 같다.