

DS Project2

이기훈 교수님

컴퓨터정보공학부

2018202076

이연걸

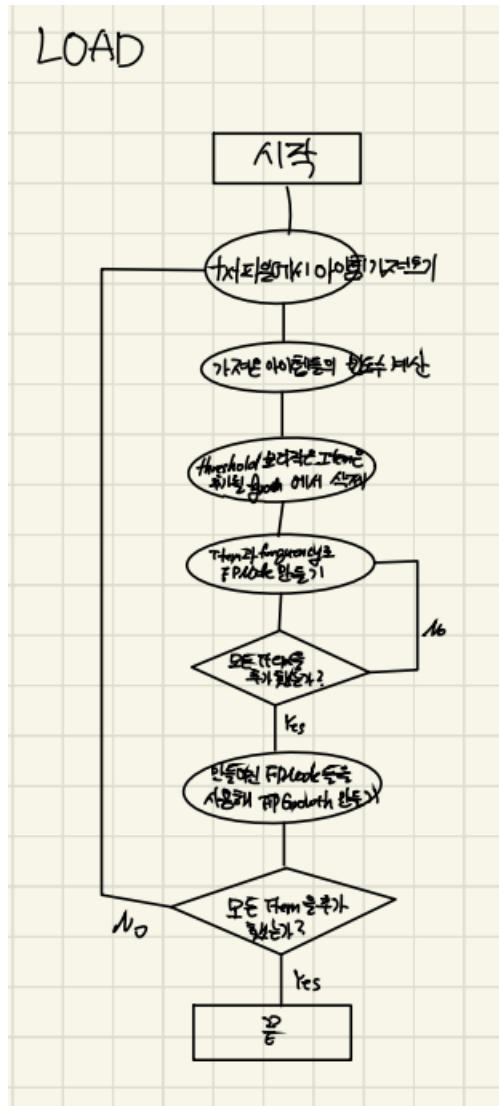
- Introduction

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

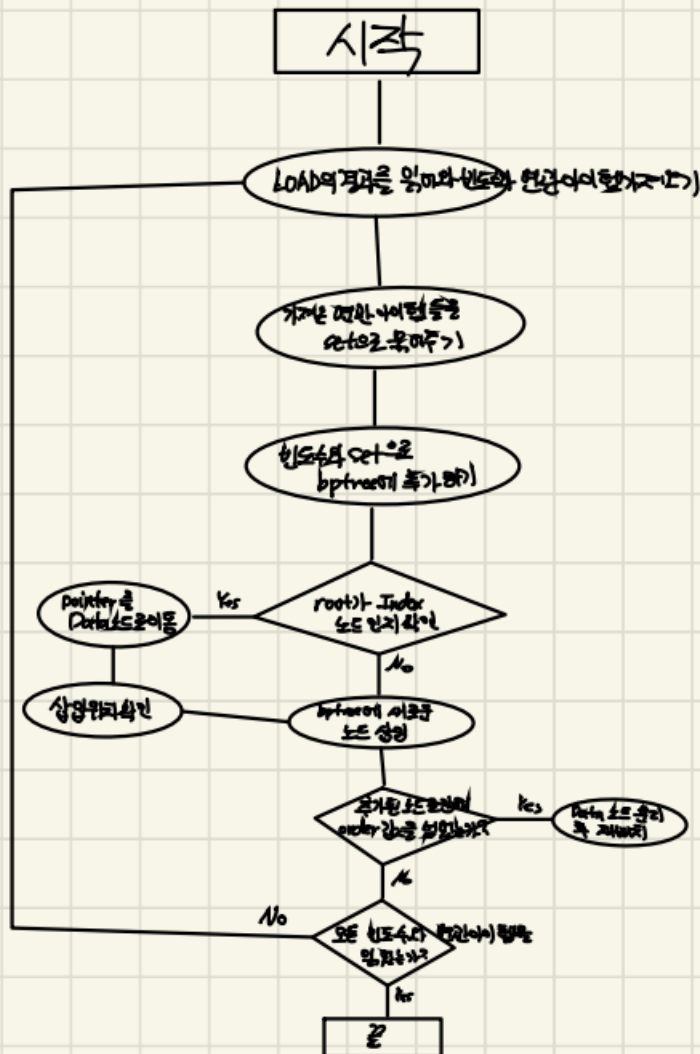
- LOAD



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.

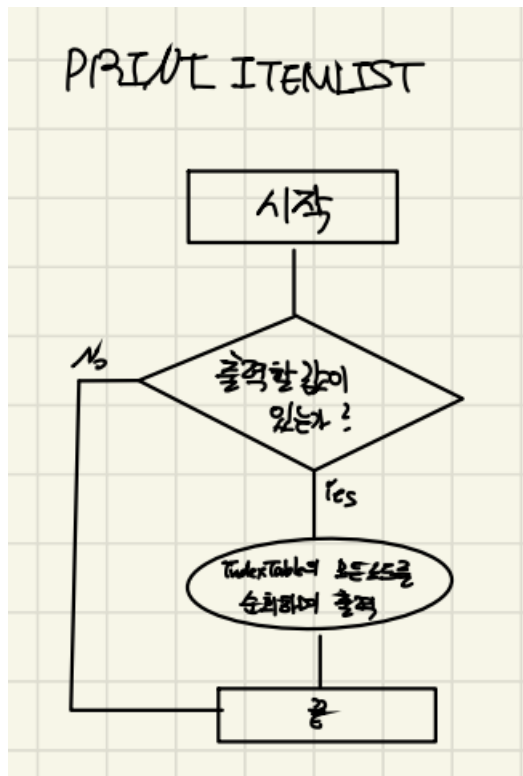
- BTLOAD

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.

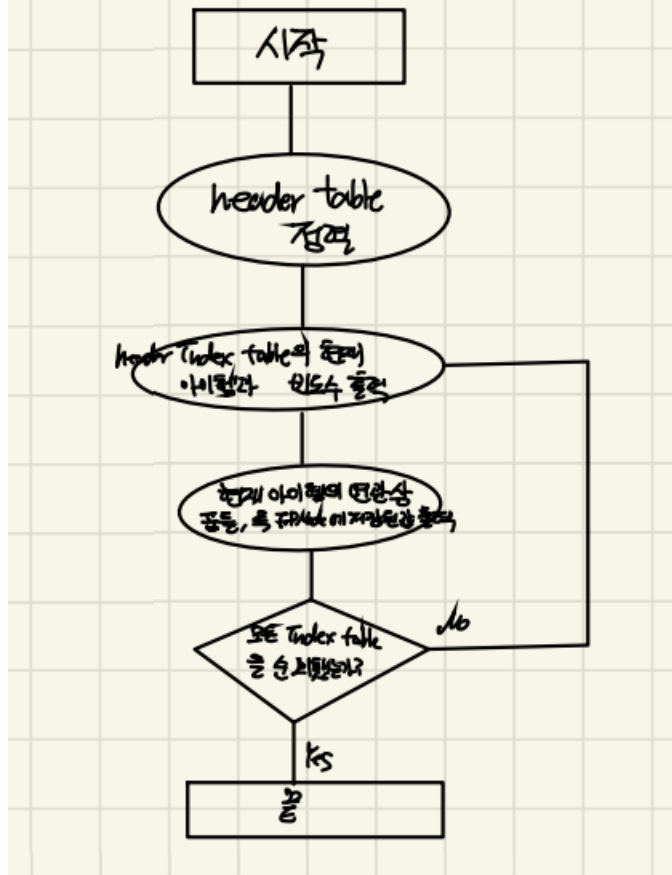
- PRINT_ITEMLIST



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

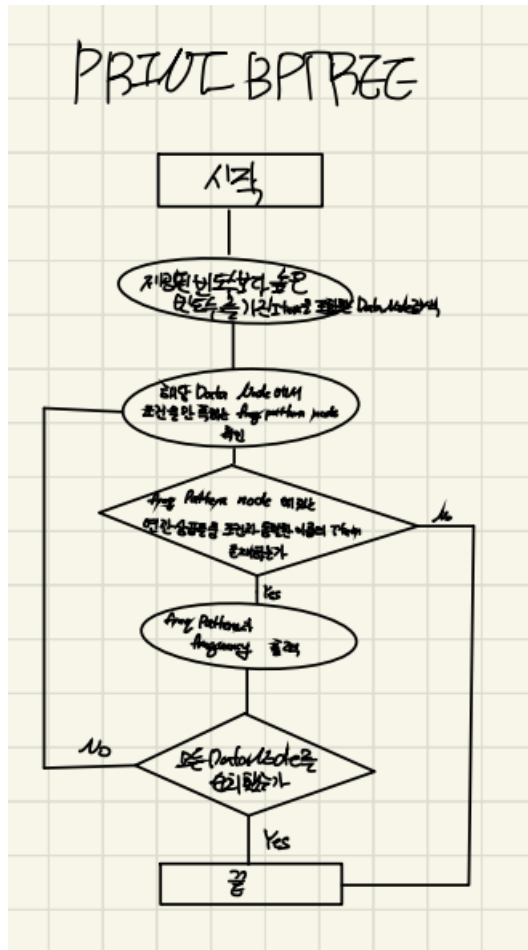
- PRINT_FPTREE

PRINT BPTREE



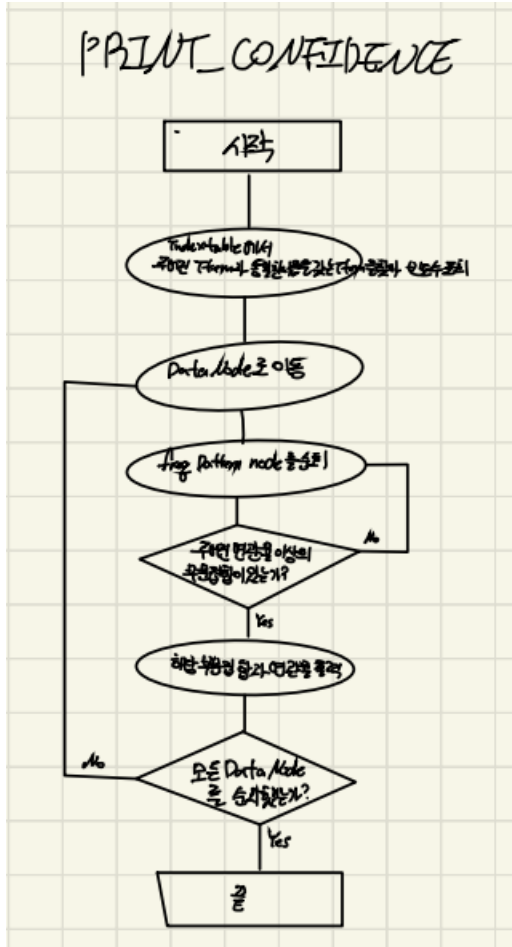
제시된 조건에 따라 headertable 을 정렬한다. 그 후 index table 를 순회하며 현재 index table 의 아이템과 빈도수를 부분 집합과 함께 출력한다. 모든 index table 을 순회했다면 함수를 종료한다.

- PRINT_BPTREE



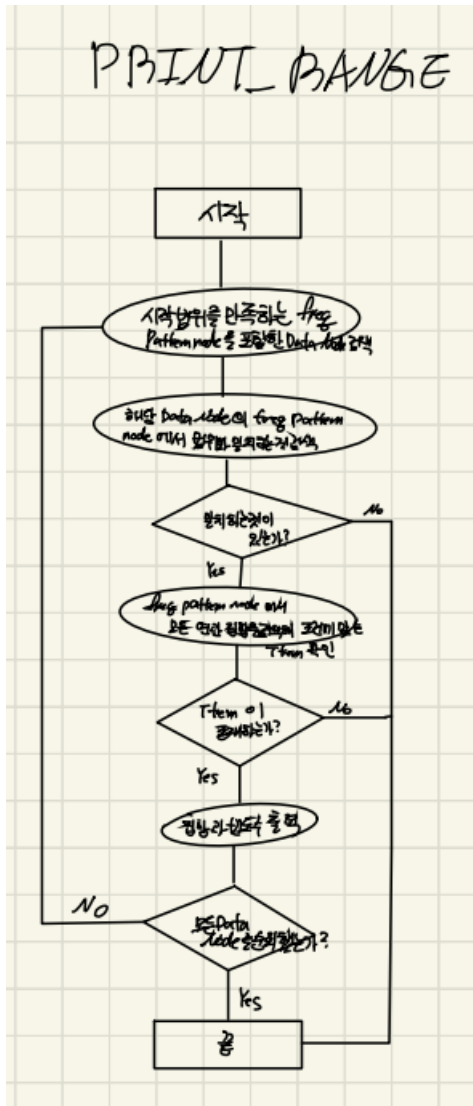
BTLOAD를 통해 만든 DataNode에서 조건을 만족하는 freq pattern node를 포함하고 있는 DataNode를 찾는다. 해당 DataNode에서 조건을 만족하는 freq pattern node가 어느 위치에 존재하는지 확인한다. 해당 위치의 frequent pattern node부터 시작해 모든 data node를 순회하면서 찾는 item과 동일한 이름을 갖는 item을 포함한 부분집합을 출력한다. 모든 DataNode를 순회했다면 함수를 종료한다.

- 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.

```

36 }
37 cur_fp_node.insert(make_pair(total_item[i][j], new_fp_node));
38 fpTree->fp_count++;
39 fp_node->children = cur_fp_node;
40 fp_node = new_fp_node;
41 cur_fp_node = new_fp_node->children;
42 new_fp_node = NULL;
43 delete new_fp_node;
44 }
45 }

```

Vector 인 total_item 에서 가져온 값을

```

37 cur_fp_node.insert(make_pair(total_item[i][j], new_fp_node));
38 fpTree->fp_count++;
39 fp_node->children = cur_fp_node;
40 fp_node = new_fp_node;
41 cur_fp_node = new_fp_node->children;
42 new_fp_node = NULL;
43 delete new_fp_node;
44 }
45 }

```

Fptree 에 삽입하고 있는 것을 확인할 수 있다.

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

```

53 else
54 {
55     FPNODE *new_fp_node = new FPNODE;
56     new_fp_node->frequency = 1;
57     new_fp_node->parent = fp_node;
58
59     auto cur_item = table->dataTable.find(total_item[i][j]);
60     if (cur_item == table->dataTable.end()) // Check current item already exists
61     {
62         table->dataTable.insert({total_item[i][j], new_fp_node});
63         fpTree->fp_count++;
64     }
65     else
66     {
67         par_node = cur_item->second;
68         temp_node = cur_item->second->next;
69     }
70 }

```

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

```
173 {
174     for (int j = 0; j < freq_item_total[i].size(); j++)
175     {
176         for (int k = 0; k < freq_item.size(); k++)
177         {
178             if (j >= freq_item_total[i].size())
179                 break;
180             if (freq_item_total[i][j] == freq_item[k].second && freq_i
181                 freq_item_total[i].erase(freq_item_total[i].begin() +
182             )
183         }
184     }
185     for (int i = 0; i < freq_item.size(); i++) // Create ftree Node with c
186         fpgrowth->createTable(freq_item[i].second, freq_item[i].first);
187     fpgrowth->table->descendingIndexTable();
188     for (int i = 0; i < freq_item_total.size(); i++) // Sort created lists
189     {
190         for (int j = freq_item_total[i].size() - 1; j > 0; j--) // Buble s
191         {
192             for (int k = 0; k < j; k++)
193             {
194                 for (item_prev = 0; item_prev < freq_item.size(); item_pre
195                 {
196                     if (freq_item_total[i][k] == freq_item[item_prev].seco
197                         break;
198                 }
199                 for (item_next = 0; item_next < freq_item.size(); item_nex
200                 {
201                     if (freq_item_total[i][k + 1] == freq_item[item_next].
```

다시 저 line 을 실행하면

```
173 {
174     for (int j = 0; j < freq_item_total[i].size(); j++)
175     {
176         for (int k = 0; k < freq_item.size(); k++)
177         {
178             if (j >= freq_item_total[i].size())
179                 break;
180             if (freq_item_total[i][j] == freq_item[k].second && freq_i
181                 freq_item_total[i].erase(freq_item_total[i].begin() +
182             )
183         }
184     }
185     for (int i = 0; i < freq_item.size(); i++) // Create ftree Node with c
186         fpgrowth->createTable(freq_item[i].second, freq_item[i].first);
187     fpgrowth->table->descendingIndexTable();
188     for (int i = 0; i < freq_item_total.size(); i++) // Sort created lists
189     {
190         for (int j = freq_item_total[i].size() - 1; j > 0; j--) // Buble s
191         {
192             for (int k = 0; k < j; k++)
193             {
194                 for (item_prev = 0; item_prev < freq_item.size(); item_pre
195                 {
196                     if (freq_item_total[i][k] == freq_item[item_prev].seco
197                         break;
198                 }
199                 for (item_next = 0; item_next < freq_item.size(); item_nex
200                 {
201                     if (freq_item_total[i][k + 1] == freq_item[item_next].
```

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.

The screenshot shows a debugger window with two panes. The left pane displays the state of a variable named `indexTable`, which is a vector of pairs. Each pair consists of a frequency (first) and a food item (second). The items are sorted in descending order of frequency. The right pane shows the corresponding C++ code, with line numbers 179 to 202. The code includes a `break;` statement at line 179, a `if` statement at line 180, and a `for` loop at line 185. The `indexTable` variable is expanded to show its contents:

- `indexTable: {...}`
- `[0]: {...}`
 - `first: 12`
 - `second: "soup"`
- `[1]: {...}`
 - `first: 9`
 - `second: "spaghetti"`
- `[2]: {...}`
 - `first: 9`
 - `second: "green tea"`
- `[3]: {...}`
 - `first: 7`
 - `second: "mineral water"`
- `[4]: {...}`
 - `first: 5`
 - `second: "milk"`
- `[5]: {...}`
 - `first: 5`
 - `second: "french fries"`
- `[6]: {...}`
 - `first: 5`
 - `second: "eggs"`
- `[7]: {...}`
 - `first: 5`
 - `second: "chocolate"`
- `[8]: {...}`
 - `first: 4`
 - `second: "ground beef"`

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

- FP-Tree

```

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
```

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.

```

6  bool BpTree::insertDataMap(const key, set<string> set)
7  {
8      FrequentPatternNode *cur_fpn = new FrequentPatternNode;
9
10     if (node_count < order) // Check root is data node
11     {
12         if (node_count == 0) // Check root is NULL
13         {
14             cur_fpn->InsertList(set);
15             root->insertDataMap(key, cur_fpn);
16         }
17         else
18         {
19             cur_fpn = root->getDataMap()->find(key)->second;
20             cur_fpn->InsertList(set);
21             root->insertDataMap(key, cur_fpn);
22         }
23     }
24     node_count++;
25 }
26

```

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

우선 분리할 위치를 찾는다.

```

80  BpTreeDataNode *new_dtn = new BpTreeDataNode;
81  auto item = pDataNode->getDataMap()->begin();
82  for (int i = 0; i < pDataNode->getDataMap()->size() / 2; i++) // Find dtn's middle position
83      item++;

```

그 후 새로운 Data Node 를 만든다.

```

74  }
75
76  void BpTree::splitDataNode(BpTreeNode *pDataNode)
77  {
78      int item_cnt;
79
80      BpTreeDataNode *new_dtn = new BpTreeDataNode;
81      auto item = pDataNode->getDataMap()->begin();
82      for (int i = 0; i < pDataNode->getDataMap()->size() / 2; i++) // Find dtn's middle position
83          item++;
84
85      for (item; item != pDataNode->getDataMap()->end(); item++) // Make new_dtn
86          new_dtn->insertDataMap(item->first, item->second);
87
88      item_cnt = pDataNode->getDataMap()->size();
89      for (int i = 0; i <= item_cnt / 2; i++) // Erase Data that used to new_dtn
90      {
91          item = pDataNode->getDataMap()->end();
92          item--;
93          pDataNode->getDataMap()->erase(item);
94      }
95
96      if (pDataNode->getNext() != NULL) // Set Prev & Next

```

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

새로운 DataNode 에 사용된 data 들은 본래의 DataNode 에서 삭제한다.

```
91     item = pDataNode->getDataMap()->end();
92     item--;
93     pDataNode->getDataMap()->erase(item);
94 }
95 pDataNode->getDataMap()->end();
96 if (pDataNode->getNext() != NULL) // Set Prev & Next
97 {
98     pDataNode->getNext()->setPrev(new_dtn);
99     new_dtn->setNext(pDataNode->getNext());
100 }
101
102 pDataNode->setNext(new_dtn);
103 new_dtn->setPrev(pDataNode);
104 if (pDataNode->getParent() == NULL) // Parent Node doesn't exist
105 {
106     BpTreeIndexNode *new_idn = new BpTreeIndexNode();
107     new_idn->insertIndexMap(new_dtn->getDataMap()->begin()->first,
```

```
91     item = pDataNode->getDataMap()->end();
92     item--;
93     pDataNode->getDataMap()->erase(item);
94 }
95 pDataNode->getDataMap()->end();
96 if (pDataNode->getNext() != NULL) // Set Prev & Next
97 {
98     pDataNode->getNext()->setPrev(new_dtn);
99     new_dtn->setNext(pDataNode->getNext());
100 }
101
102 pDataNode->setNext(new_dtn);
103 new_dtn->setPrev(pDataNode);
104 if (pDataNode->getParent() == NULL) // Parent Node doesn't exist
105 {
106     BpTreeIndexNode *new_idn = new BpTreeIndexNode();
107     new_idn->insertIndexMap(new_dtn->getDataMap()->begin()->first,
```

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

```
96     if (pDataNode->getNext() != NULL) // Set Prev & Next
97     {
98         pDataNode->getNext()->setPrev(new_dtn);
99         new_dtn->setNext(pDataNode->getNext());
100     }
101
102     pDataNode->setNext(new_dtn);
103     new_dtn->setPrev(pDataNode);
104     if (pDataNode->getParent() == NULL) // Parent Node doesn't exist
105     {
106         BpTreeIndexNode *new_idn = new BpTreeIndexNode();
107         new_idn->insertIndexMap(new_dtn->getDataMap()->begin()->first,
108                                new_idn->setMostLeftChild(pDataNode);
109
110         pDataNode->setParent(new_idn);
111         new_dtn->setParent(new_idn);
112         root = new_idn;
113     }
114     else
115     {
116         pDataNode->getParent()->insertIndexMap(new_dtn->getDataMap()->begin()->first,
117                                                  new_dtn->setParent(pDataNode->getParent());
118         if (excessIndexNode(new_dtn->getParent()))
119             splitIndexNode(pDataNode->getParent());
120     }
121     new_dtn = NULL;
122     delete new_dtn;
123 }
124
```

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


```
table->dataTable: {...}
  ▾ ["avocado"]: 0x555555599ea0 ⓘ
    fp_count: 0
    frequency: 1
    > parent: 0x555555599e00
    > next: 0x55555559a210
    > children: {...}
    > ["body spray"]: 0x555555599b30
    > ["burgers"]: 0x555555599e00
    > ["chicken"]: 0x555555599360
    > ["chocolate"]: 0x5555555997c0
    > ["eggs"]: 0x555555599220
    > ["energy bar"]: 0x555555598ff0
    > ["french fries"]: 0x555555599450
    > ["green tea"]: 0x555555594820
    > ["ground beef"]: 0x55555559a350
    > ["honey"]: 0x5555555994f0
    > ["milk"]: 0x555555598f00
    > ["mineral water"]: 0x555555598e...
    > ["protein bar"]: 0x5555555996d0
    > ["shrimp"]: 0x555555599fe0
    > ["soup"]: 0x555555599900
    > ["spaghetti"]: 0x555555599180
    > ["white wine"]: 0x555555599d10
```

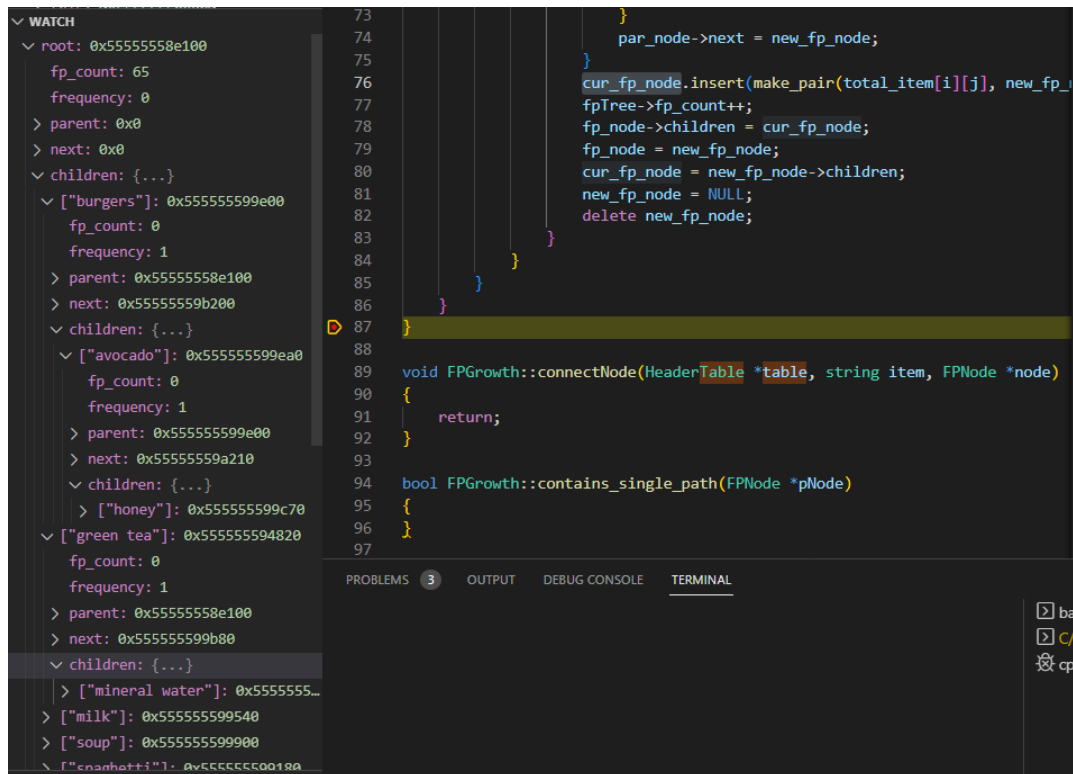
FP-Tree 를 만들면서 dataTable 이 완성된 것을 볼 수 있다.

```

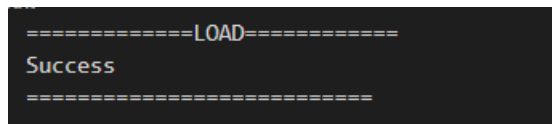
✓ table->indexTable: {...}
  ✓ [0]: {...}
    first: 12
    ✓ second: "soup"
  ✓ [1]: {...}
    first: 9
    > second: "spaghetti"
  ✓ [2]: {...}
    first: 9
    ✓ second: "green tea"
  > [3]: {...}
  ✓ [4]: {...}
    first: 5
    > second: "milk"
  > [5]: {...}
  > [6]: {...}
  > [7]: {...}
  > [8]: {...}
  ✓ [9]: {...}
    first: 4
    > second: "burgers"
  > [10]: {...}
  > [11]: {...}
  > [12]: {...}
  ✓ [13]: {...}
    first: 3
    > second: "energy bar"
  > [14]: {...}

```

Index table 도 완성된 것을 확인할 수 있다.

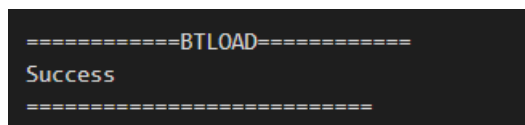


LOAD 작업이 종료되면 이렇게 FP-Tree 에 값이 채워진 것을 확인할 수 있다. 구현 방법은 Algorithm 목차에서 확인할 수 있다.



- BTLOAD

FrequentPatternNode 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.



- PRINT_ITEMLIST

```
> __for_range: {...}
__for_begin: {first = 1, second = "bug spray"}
__for_end: {first = 52, second = ""}
> this: 0x7fffffffdd2e0
> Registers

WATCH
  cur_bptree_node: -var-create: unable to create v...
  > bptree: 0x55555558e1a0

248         bptree->splitDataNode(cur_bptree_node, cur_bptree_node->left, cur_bptree_node->right);
249     }
250 }
251
252     return true;
253 }
254
255 bool Manager::PRINT_ITEMLIST()
256 {
257     if (fpgrowth->getHeaderTable()->getIndexTable() == nullptr)
258         return false;
259     fout << "Item\tFrequency\n";
260     for (auto i : fpgrowth->table->indexTable)
261         fout << i.second << '\t' << i.first << '\n';
262     return true;
263 }
```

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

```
Manager.cpp M log.txt M X result1 result3 BpTre
log.txt
5
6 Success
7 =====
8
9 =====PRINT_ITEMLIST=====
10 Item    Frequency
11 soup 12
12 spaghetti 9
13 green tea 9
14 mineral water 7
15 milk 5
16 french fries 5
17 eggs 5
18 chocolate 5
19 ground beef 4
20 burgers 4
21 white wine 3
22 protein bar 3
23 honey 3
24 energy bar 3
25 chicken 3
26 body spray 3
27 avocado 3
28 whole wheat rice 2
29 turkey 2
30 shrimp 2
31 salmon 2
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
41 black tea 2
42 almonds 2
43 whole wheat pasta 1
44 toothpaste 1
45 tomatoes 1
46 soda 1
47 shampoo 1
48 shallot 1
49 red wine 1
50 pet food 1
```

- PRINT_FPTREE

```
Locals
  cur_item: {first = <error rea...
  cur_fp_node: 0x5555555d0e0 <...
  standard_item: {...
    first: 3
    second: "avocado"
    __for_range: {...
    __for_begin: {first = 3, seco...
    __for_end: {first = 52, seco...
    this: 0x7fffffff2e0
  Registers
WATCH
  bptree: 0x5555558e1a0
  threshold: 3

271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291

fpgrowth->table->ascendingIndexTable();
for (auto standard_item : fpgrowth->table->indexTable) // Travelling repeat state
{
    if (standard_item.first >= threshold) // Check item threshold
    {
        fcout << "(" << standard_item.second << ", " << standard_item.first << ")";
        << "\n";
        auto cur_item = fpgrowth->table->dataTable.find(standard_item.second);
        FPNode *cur_fp_node = cur_item->second;
        while (cur_fp_node) // Check all nodes including the current item
        {
            FPNode *par_fp_node = cur_fp_node;
            while (par_fp_node != fpgrowth->fpTree)
            {
                fcout << "(";
                for (auto path_item : par_fp_node->parent->children)
                {
                    if (path_item.second == par_fp_node)
                        fcout << path_item.first << ", " << par_fp_node->frequency;
                }
            }
        }
    }
}
```

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

```
path_item: {...
  first: "burgers"
  second: 0x555555599e00 <...
  __for_range: {...
  __for_begin: {first = "burgers", second = 0x55555559...
  __for_end: {first = <error: Cannot access memory at ...
  par_fp_node: 0x555555599900
  cur_item: {first = "avocado", second = 0x555555599ea...
  cur_fp_node: 0x55555559a210
  standard_item: {...
  __for_range #2: {...
  __for_begin #2: {first = "burgers", second = 0x5555...
  __for_end #2: {first = <error: Cannot access memory ...
  this: 0x7fffffff2e0
  Registers

279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299

auto cur_item = fpgrowth->table->dataTable.find(standar...
FPNode *cur_fp_node = cur_item->second;
while (cur_fp_node) // Check all nodes including the cur...
{
    FPNode *par_fp_node = cur_fp_node;
    while (par_fp_node != fpgrowth->fpTree)
    {
        fcout << "(";
        for (auto path_item : par_fp_node->parent->child...
        {
            if (path_item.second == par_fp_node)
                fcout << path_item.first << ", " << par_f...
        }
        fcout << ") ";
        par_fp_node = par_fp_node->parent;
    }
    fcout << "\n";
    cur_fp_node = cur_fp_node->next;
}
```

```
path_item: {...
  first: "green tea"
  second: 0x555555594820 <...
  __for_range: {...
  __for_begin: {first = "green tea", second = 0x555555...
  __for_end: {first = <error: Cannot access memory at ...
  par_fp_node: 0x555555599900
  cur_item: {first = "avocado", second = 0x555555599ea...
  cur_fp_node: 0x55555559a210
  standard_item: {...
  __for_range #2: {...
  __for_begin #2: {first = "green tea", second = 0x555...
  __for_end #2: {first = <error: Cannot access memory ...
  this: 0x7fffffff2e0
  Registers

279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297

auto cur_item = fpgrowth->table->dataTable.find(standar...
FPNode *cur_fp_node = cur_item->second;
while (cur_fp_node) // Check all nodes including the c...
{
    FPNode *par_fp_node = cur_fp_node;
    while (par_fp_node != fpgrowth->fpTree)
    {
        fcout << "(";
        for (auto path_item : par_fp_node->parent->child...
        {
            if (path_item.second == par_fp_node)
                fcout << path_item.first << ", " << par...
        }
        fcout << ") ";
        par_fp_node = par_fp_node->parent;
    }
    fcout << "\n";
    cur_fp_node = cur_fp_node->next;
}
```

위처럼 조건을 만족하는 값들이 차례로 출력되는 것을 확인할 수 있다.

```
Manager.cpp M log.txt M X result1 result3 BpTree.cpp M BpT
log.txt
64
65 =====PRINT_FPTREE=====
66 {StandardItem.Frequency} (Path_Item.Frequency)
67 {avocado, 3}
68 (avocado, 1) (burgers, 1)
69 (avocado, 1) (milk, 1) (spaghetti, 4) (soup, 12)
70 (avocado, 1) (french fries, 1) (green tea, 4) (soup, 12)
71 {body spray, 3}
72 (body spray, 1) (mineral water, 1) (green tea, 2) (spaghetti, 5)
73 (body spray, 1) (avocado, 1) (french fries, 1) (green tea, 4) (soup, 12)
74 (body spray, 1) (chicken, 1) (green tea, 4) (soup, 12)
75 {chicken, 3}
76 (chicken, 1) (eggs, 1) (mineral water, 3) (spaghetti, 5)
77 (chicken, 1) (french fries, 1) (chocolate, 1) (eggs, 1) (mineral water, 1) (soup, 12)
78 (chicken, 1) (green tea, 4) (soup, 12)
79 {energy bar, 3}
80 (energy bar, 1) (milk, 1) (mineral water, 1) (green tea, 1)
81 (energy bar, 1) (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)
82 (energy bar, 1) (soup, 12)
83 {honey, 3}
84 (honey, 1) (french fries, 1) (milk, 1)
85 (honey, 1) (avocado, 1) (burgers, 1)
86 (honey, 1) (mineral water, 1) (spaghetti, 4) (soup, 12)
87 {protein bar, 3}
88 (protein bar, 1) (honey, 1) (french fries, 1) (milk, 1)
89 (protein bar, 1) (green tea, 4) (soup, 12)
90 (protein bar, 1) (energy bar, 1) (soup, 12)
91 {white wine, 3}
92 (white wine, 1) (honey, 1) (avocado, 1) (burgers, 1)
93 (white wine, 1) (chocolate, 1) (green tea, 2) (spaghetti, 5)
94 (white wine, 1) (ground beef, 1) (mineral water, 3) (spaghetti, 5)
95 {burgers, 4}
96 (burgers, 1)
97 (burgers, 1) (french fries, 1) (milk, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
98 (burgers, 1) (french fries, 1) (eggs, 1) (green tea, 4) (soup, 12)
99 (burgers, 1) (eggs, 1) (chocolate, 1) (soup, 12)
100 {ground beef, 4}
101 (ground beef, 1) (milk, 1) (mineral water, 3) (spaghetti, 5)
102 (ground beef, 1) (mineral water, 3) (spaghetti, 5)
103 (ground beef, 1) (chocolate, 1) (green tea, 2) (spaghetti, 4) (soup, 12)
104 (ground beef, 1) (burgers, 1) (eggs, 1) (chocolate, 1) (soup, 12)
105 {chocolate, 5}
106 (chocolate, 1) (eggs, 1) (soup, 12)
107 (chocolate, 1) (eggs, 1) (mineral water, 1) (soup, 12)
```

- 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


```
Locals
cur_item: "eggs"
> item_set: {...}
res: "eggs"
freq_pat_node: {first = 2, second = std_
> cur_freq_pat_node: {...}
cur_data_node: {first = 2, second = 0x5...
> data_node: 0x55555558e1d8
flag: 0
item_name: "eggs"

auto item_set = freq_pat_node->second;
auto res = item_set.find(item_name);
if (res != item_set.end())
{
    fcout << "{";
    for (auto cur_item = item_set.begin
    {
        fcout << *cur_item++;
        if (cur_item != item_set.end())
            fcout << ", ";
        cur_item--;
    }
}
```

```
=====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
```

```
Manager.cpp M log.txt M X result1 result3 BpTree.cpp M
log.txt
133 (mineral water, 1) (soup, 12)
134 (mineral water, 1) (spaghetti, 4) (soup, 12)
135 {green tea, 9}
136 (green tea, 1)
137 (green tea, 2) (spaghetti, 5)
138 (green tea, 4) (soup, 12)
139 (green tea, 2) (spaghetti, 4) (soup, 12)
140 {spaghetti, 9}
141 (spaghetti, 5)
142 (spaghetti, 4) (soup, 12)
143 {soup, 12}
144 (soup, 12)
145 =====
146
147 =====PRINT_BPTREE=====
148 FrequentPattern Frequency
149 {almonds, eggs} 2
150 {burgers, eggs} 2
151 {chicken, eggs} 2
152 {eggs, french fries} 2
153 {eggs, mineral water} 2
154 {eggs, turkey} 2
155 {almonds, burgers, eggs} 2
156 {almonds, eggs, soup} 2
157 {burgers, eggs, soup} 2
158 {chicken, eggs, mineral water} 2
159 {eggs, french fries, soup} 2
160 {almonds, burgers, eggs, soup} 2
161 {chocolate, eggs} 3
162 {chocolate, eggs, soup} 3
163 {eggs, soup} 4
164 =====
165
166 =====PRINT_CONFIDENCE=====
167 FrequentPattern Frequency Confidence
168 {milk, mineral water} 4 0.8
169 =====
170
171 =====PRINT_RANGE=====
172 FrequentPattern Frequency Confidence
173 {milk, spaghetti} 3
174 {milk, soup, spaghetti} 3
175 {milk, mineral water} 4
176 =====
177
```

이렇게 출력된 것을 확인할 수 있다.

- PRINT_CONFIDENCE

```
cur_item: {first = 5, second = "milk"}
item_freq: 5
flag: 0
item_name: "milk"
> data_node: 0x5555555a4710
> this: 0x7ffffffd2e0
> item: 0x7ffffffd261 "milk"
rate: 0.5999999999999998
> Registers

350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
{
    if (item == NULL || bptree->node_count == 0)
        return false;

    fout << "FrequentPattern Frequency Confidence\n";

    int item_freq = 0, flag = 0;
    string item_name = item;
    BpTreeNode *data_node = bptree->root;

    for (auto cur_item = fpgrowth->getHeaderTable()->indexTable.begin();
        {
            if (cur_item->second == item_name)
            {
                item_freq = cur_item->first;
                break;
            }
        }
    }
```

우선 찾는 item 의 빈도수를 먼저 구한다.

그 후 DataNode 를 찾는다.

```
data_node: 0x55555558e1c0
> pParent: 0x5555555a0220
pMostLeftChild: 0x0
> pParent
> pMostLeftChild
> this: 0x7ffffffd2e0
> item: 0x7ffffffd261 "milk"
rate: 0.5999999999999998
> Registers

353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
{
    fout << "FrequentPattern Frequency Confidence\n";

    int item_freq = 0, flag = 0;
    string item_name = item;
    BpTreeNode *data_node = bptree->root;

    for (auto cur_item = fpgrowth->getHeaderTable()->indexTable.begin();
        {
            if (cur_item->second == item_name)
            {
                item_freq = cur_item->first;
                break;
            }
        }

    while (data_node->getMostLeftChild()) // Find Data Node
        data_node = data_node->getMostLeftChild();

    while (data_node->getNext()) // Travelling All Bp Tree Node
        data_node = data_node->getNext();
}
```

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

```
Locals
> cur_freq_pat_node: {...}
freq_pat_node: {first = 4, second = 0x5555555a0960}
> cur_data_node: 0x5555555a4228
item_freq: 5
flag: 0
item_name: "milk"
data_node: 0x5555555a4210
> pParent: 0x5555555a0220
pMostLeftChild: 0x0
> pParent
> pMostLeftChild
> this: 0x7ffffffd2e0
> item: 0x7ffffffd261 "milk"
rate: 0.5999999999999998
> Registers

359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
{
    for (auto cur_item = fpgrowth->getHeaderTable()->indexTable.begin(); cur_item
    {
        if (cur_item->second == item_name)
        {
            item_freq = cur_item->first;
            break;
        }
    }

    while (data_node->getMostLeftChild()) // Find Data Node
        data_node = data_node->getMostLeftChild();

    while (data_node->getNext()) // Travelling All Bp Tree Node
    {
        auto cur_data_node = data_node->getDataMap();
        for (auto freq_pat_node = cur_data_node->begin(); freq_pat_node != cur_data_node->end(); freq_pat_node++)
        {
            if ((double)freq_pat_node->first / item_freq > rate) // Check that the
            {
                auto cur_freq_pat_node = freq_pat_node->second->getList();
                for (auto item_set = cur_freq_pat_node->begin(); item_set != cur_freq_pat_node->end(); item_set++)
                {
                    if (item_set->second == item_name)
                    {
                        // ...
                    }
                }
            }
        }
    }
}
```

찾고 난 후 이름이 조건과 동일한 부분집합을 찾아 출력한다.

The screenshot shows a debugger interface. On the left, the 'Locals' window displays the following variables and values:

- `cur_item`: "milk"
- `cur_item_set`: {...}
- `item_set`: {first = 2, second = std::set with 2 elements = {[0] = "milk", [1] = "mineral wat..."}
- `cur_freq_pat_node`: {...}
- `freq_pat_node`: {first = 4, second = 0x5555555a0960}
- `cur_data_node`: 0x5555555a4228
- `item_freq`: 5
- `flag`: 0
- `item_name`: "milk"
- `data_node`: 0x5555555a4210
- `this`: 0x7fffffff2e0
- `item`: 0x7fffffff261 "milk"
- `rate`: 0.5999999999999998

On the right, the source code window shows a loop structure. The current execution point is at line 387, which is highlighted in yellow.

The screenshot shows the same debugger interface as above, but the 'Locals' window now displays variables for 'mineral water':

- `cur_item`: "mineral water"
- `cur_item_set`: {...}
- `item_set`: {first = 2, second = std::set with 2 elements = {[0] = "milk", [1] = "mineral wat..."}
- `cur_freq_pat_node`: {...}
- `freq_pat_node`: {first = 4, second = 0x5555555a0960}
- `cur_data_node`: 0x5555555a4228
- `item_freq`: 5
- `flag`: 0
- `item_name`: "milk"
- `data_node`: 0x5555555a4210
- `this`: 0x7fffffff2e0
- `item`: 0x7fffffff261 "milk"
- `rate`: 0.5999999999999998

The source code window shows the same loop structure, but the current execution point is now at line 389, which is highlighted in yellow.

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

```

166  =====PRINT_CONFIDENCE=====
167  FrequentPattern Frequency Confidence
168  {milk, mineral water} 4 0.8
169  =====
170

```

- PRINT_RANGE

VARIABLES

- Locals**
 - flag: 0
 - item_name: "milk"
 - condition_pos: 0x555555a0a40
 - this: 0x7fffffffd2e0
 - item: 0x7fffffffd25c "milk"
 - start: 3
 - end: 4
- Registers**

WATCH

- condition_pos->getDataMap()->begin()->frequency: 0
- FrequentPatternList: {...}
 - [2]: {...}
 - [0]: "body spray"
 - [1]: "green tea"
 - [2] #2: {...}
 - [0]: "burgers"
 - [1]: "soup"
 - [2] #3: {...}
 - [0]: "chocolate"
 - [1]: "eggs"
 - [2] #4: {...}
 - [0]: "french fries"
 - [1]: "green tea"
 - [3]: {...}
 - [0]: "chocolate"
 - [1]: "eggs"
 - [2]: "soup"
 - [3] #2: {...}
 - [0]: "french fries"
 - [1]: "green tea"
 - [2]: "soup"

CALL STACK

result1

Index	Count	Item Name
11	2	black tea
12	2	black tea
13	2	black tea
14	2	black tea
15	2	black tea
16	3	body spray
17	2	body spray
18	2	body spray
19	2	body spray
20	2	body spray
21	2	brownies
22	2	burgers
23	2	burgers
24	2	burgers
25	2	burgers
26	2	burgers
27	2	burgers
28	2	burgers
29	2	burgers
30	3	burgers
31	2	chicken
32	2	chicken
33	2	chicken
34	2	chicken
35	3	chocolate
36	3	chocolate
37	2	chocolate
38	2	chocolate
39	2	chocolate
40	2	chocolate
41	4	chocolate

우선 조건에 해당하는 DataNode 를 찾는다.

VARIABLES

- Locals**
 - item: 0x7fffffffd25c "milk"
 - start: 3
 - end: 4
- Registers**

WATCH

- condition_pos->getDataMap()->begin()->frequency: 0

Code Snippet

```

410 // FrequentPattern Frequency Confidence
411
412 int flag = 0;
413 string item_name = item;
414 BpTreeNode *condition_pos = bptree->searchDataNode(start);
415 condition_pos->getDataMap()->begin()->second;
416 while (condition_pos->getNext()) // Travelling all data nodes
417 {
418     auto data_node = condition_pos->getDataMap();
419     for (auto cur_data_node = data_node->begin(); cur_data_node
  
```

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

VARIABLES

- Locals**
 - cur_item_set: {...}
 - [0]: "milk"
 - [1]: "spaghetti"
 - ch: "milk"
 - item_set: {first = 2, second = std::set with 2 elements = {[0] = "milk", [1] = "spaghetti"}}
 - cur_freq_pat_node: {...}
 - cur_data_node: {first = 3, second = 0x555555a40e0}
 - data_node: 0x555555a3928
 - flag: 0
 - item_name: "milk"
 - condition_pos: 0x555555a3910
 - this: 0x7fffffffd2e0
 - item: 0x7fffffffd25c "milk"
 - start: 3
 - end: 4
- Registers**

Code Snippet

```

406 bool Manager::PRINT_RANGE(char *item, int start, int end)
407 {
408     if (item == NULL || bptree->node_count == 0)
409         return false;
410     fout << "FrequentPattern Frequency Confidence\n";
411
412     int flag = 0;
413     string item_name = item;
414     BpTreeNode *condition_pos = bptree->searchDataNode(start);
415     condition_pos->getDataMap()->begin()->second;
416     while (condition_pos->getNext()) // Travelling all data nodes
417     {
418         auto data_node = condition_pos->getDataMap();
419         for (auto cur_data_node = data_node->begin(); cur_data_node
420             {
421             if (cur_data_node->first >= start && cur_data_node->first
422                 {
423                     auto cur_freq_pat_node = cur_data_node->second->getL
424                     for (auto item_set = cur_freq_pat_node.begin(); item
425                         {
426                             auto cur_item_set = item_set->second;
427                             auto ch = cur_item_set.find(item_name);
428                             if (ch != cur_item_set.end())
429                             {
430                                 flag = 1;
431                                 fout << "(";
432                                 for (auto cur_item = cur_item_set.begin(); c
433                                     {
434                                         fout << " " << cur_item->first;
  
```

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 multimap 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.