Implementation of

a B+ tree index instruction

- Course name: Database Systems (ITE2038)
- Professor: Sang-Wook Kim
- Student: 이민준
- Student ID: 2018008859
- Environment: OS-Windows, Language-Python

Table of Contents

1. Instructions for Compiling Source Codes

- Data file Creation

- **Command**: python bptree.py -c index_file b
- This command creates a new index file containing an empty index with node size b
- index_file: name of a new index file

```
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>python bptree.py -c index.dat 10
```

- Insertion

- **Command**: python bptree.py -i index_file data_file
- This command inserts all the key-value pairs inside the data_file

  into the index in the index_file
- data_file: name of the input data file that has a number of key-value pairs to be inserted
- The data file is provided as a .csv file

```
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>python bptree.py -i index.dat input.csv
```

- Deletion

- **Command**: python bptree.py -d index_file data_file
- This command deletes all the key-value pairs inside the input data file from the index
- data_file: name of the input data file that has a number of keys to be deleted
- The data file is provided as a .csv file

```
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>python bptree.py -d index.dat delete.csv
```

- Single Key Search

- **Command**: python bptree.py -s index_file key
- This command returns a value of a pointer to a record with the key
- While searching, the program prints each non-leaf node that the search passes through

```
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>
WB-tree_Assignment>python bptree.py -s index.dat 506666
```

```
237209,518029,804199
278424,317875,349560,398579,435808,462362,485945
490682,495889,499195,502554,509595,513789
503214,503705,504764,505278,505678,506666,507440,508818
613427
```

Print the value matched with the search key in the last line

```
237209,518029,804199
39116,81454,110339,133205,164405,202631
6103,9742,15165,21098,26190,33535
710,1041,1414,1839,2598,3185,3604,5182
NOT FOUND
```

If the key doesn't exist in the leaf node, print "NOT FOUND"

- Range Search

- Command: python bptree.py -r index_file start_key end_key
- start_key: lower bound of the range search
- end_key: upper bound of the range search
- This command returns the values of pointers to records having the keys within the range

```
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>
₩B-tree_Assignment>python bptree.py -r index.dat 710 1044
```

```
710,965357
874,53992
924,404453
953,811683
1031,58326
1041,391259
```

2. Summary of Algorithm &

Detailed Description of Codes

- Outline

• Tree

```
22  class bPlusTree:
23      root: Node
24      degree: int
25      key_values: dict[int, dict[int, int]]
26      # 큰 dict 의 key 는 포인터 주소의 역할,
27      # 큰 dict 의 value 에 key_value pairs 를 저장
28
29      def __init__(self, degree=0):
30          self.root = Node(degree)
31          self.degree = degree
32          self.key_values = dict()
```

Line 22 ~ 32

- root node
- degree
- <key,value> pair list

• Node

```python
class Node:
    m: int  # number of keys
    p: dict  # data_node 이면 [int, dict], index_node 이면 [int, Node]
    r: Node  # data_node 이면 right sibling node, index_node 이면 leftmost child node 를 가리킴
    degree: int
    is_index_node: bool
    is_data_node: bool

    def __init__(self, degree=0):
        self.m = 0
        self.p = dict()
        self.r = None
        self.degree = degree
        self.is_index_node = True
        self.is_data_node = True
```

Line 5 ~ 19

- m: number of keys
- p: data node의 경우 array of <key,pointer to the value> pairs
- p: index node의 경우 array of <key,child node> pairs
- r: data node의 경우 right sibling node
- r: index node의 경우 해당 노드의 leftmost child node
- degree
- is_index_node
- is_data_node

- Insertion

• tree에 data node만 존재하는 경우의 insertion

```
235  def insertion(tree: bPlusTree, key: int, value: int) -> bPlusTree:
236      mid = tree.degree // 2
237
238      # tree 에 data node 만 존재하는 경우
239      if tree.root.is_data_node:
240          tree.root.is_index_node = False
241          tree.key_values[key] = {key: value}
242          tree.root.p[key] = tree.key_values[key]
243          tree.root.p = sort_dictionary(tree.root.p)  # The keys in a node are stored in an ASCENDING order
244          tree.root.m += 1
245
246          # node 에 공간이 없을 경우 split 시행
247          if tree.root.m == tree.degree:
248              tree.root = split(tree, tree.root, mid)
```

Line 235 ~ 248

1. tree의 <key,value> pair list에 input을 저장

2. tree의 root node의 p에 input의 key와

해당 key의 <key,value> pair를 가리키는 pointer key를 key 기준 오름차순으로 삽입

2-1. tree의 root node의 m이 1 증가

3. node에 공간이 없을 경우(node.m == node.degree), split 시행

4. split 완료 시, split한 노드가 tree의 root node

• tree에 index node와 data node 모두 존재하는 경우의 insertion

```
250         # tree 에 index node 와 node node 모두 존재하는 경우
251         else:
252             tmp_node = tree.root
253 
254             # data node 에 도착할 때까지 진행
255             while not tmp_node.is_data_node:
256 
257                 # index node 탐색 도중 key 가 발견 되었다면 그 index node 가 가리키는 child 로 이동
258                 if key in list(tmp_node.p.keys()):
259                     tmp_node = tmp_node.p[key]
260 
261                 # index node 의 number of keys 가 1 일 때
262                 elif len(tmp_node.p.keys()) == 1:
263 
264                     # key 가 index node 에 존재하는 key 보다 작으면 index node 의 left child 로 이동
265                     if key < list(tmp_node.p.keys())[0]:
266                         tmp_node = tmp_node.r
267 
268                     # key 가 index node 에 존재하는 key 보다 크면 index node 의 right child 로 이동
269                     else:
270                         tmp_node = tmp_node.p[list(tmp_node.p.keys())[0]]
```

```
272                 # index node 에 여러개의 key 들이 있을 때
273                 else:
274 
275                     # split_key 가 tmp_node 의 모든 key 들보다 작을 경우 tmp_node 의 leftmost child 로 이동
276                     if key < list(tmp_node.p.keys())[0]:
277                         tmp_node = tmp_node.r
278 
279                     # split_key 가 tmp_node 의 모든 key 들보다 클 경우 tmp_node 의 rightmost child 로 이동
280                     elif key > list(tmp_node.p.keys())[-1]:
281                         tmp_node = tmp_node.p[list(tmp_node.p.keys())[-1]]
282 
283                     # split_key 가 tmp_node 의 keys[i] keys[i + 1] 사이의 범위에 존재하면 keys[i]가 가리키는 child 로 이동
284                     else:
285                         # for loop 를 돌며 if 문을 만족하면 tmp_node 이동 후 바로 break for loop
286                         for i in range(len(list(tmp_node.p.keys())) - 1):
287                             if list(tmp_node.p.keys())[i] < key < list(tmp_node.p.keys())[i + 1]:
288                                 tmp_node = tmp_node.p[list(tmp_node.p.keys())[i]]
289                                 break
```

Line 250 ~ 289

1. input의 key가 들어가야 할 data node 탐색

1-1. key가 index node의 모든 key들보다 작으면 index node의 leftmost child로 이동

1-2. key가 index node의 key들 사이 범위에 존재하면

오름차순으로 정렬되어 있는 node의 p를 한 칸씩 돌며

key가 속하는 범위에 해당하는 child로 이동

1-3. key가 index node의 모든 key들보다 크면 index node의 rightmost child로 이동

1-4. key가 탐색 도중 index node에 존재하면 그 key가 가리키는 node로 이동

1-4. 탐색 도중 data node에 도달하면 탐색 종료

```
291         # 도착한 data node 에 key 삽입
292         tree.key_values[key] = {key: value}
293         tmp_node.p[key] = tree.key_values[key]
294         tmp_node.p = sort_dictionary(tmp_node.p)  # The keys in a node are stored in an ASCENDING order
295         tmp_node.m += 1
296
297         # node 에 공간이 없을 경우 split 시행
298         # split 완료 후의 node 에 또 공간이 없을 경우, 다시 split 시행
299         while tmp_node.m == tree.degree:
300             tmp_node = split(tree, tmp_node, mid)
```

Line 291 ~ 300

2. 탐색한 node의 p에 input의 key와

해당 key의 <key,value> pair를 가리키는 pointer key를 key 기준 오름차순으로 삽입

2-1. 삽입한 node의 m이 1 증가

3. node에 공간이 없을 경우(node.m == node.degree), split 시행

3-1. split한 node에 또 공간이 없을 경우, 다시 split 시행

3-2. 3-1의 상황이 계속 반복된다면, 그렇지 않을 때까지 3-1 시행

- Split

- 공통 사항

```
112 # node 를 받아 split 하여 반환
113 # parent node 가 있는 data node
114 # parent node 가 없는 data node
115 # parent node 가 있는 index node
116 # parent node 가 없는 index node
117 # 총 4가지 경우로 나누어서 진행
118 def split(tree: bPlusTree, node: Node, mid: int) -> Node:
119     # split_key 선택
120     split_key = list(node.p.keys())[mid]
```

Line 118 ~ 120

√. degree가 홀수인 경우: split_key == {(degree – 1) / 2}번째 key (0번째 시작 기준)

√. degree가 짝수인 경우: split_key == (degree / 2)번째 key (0번째 시작 기준)

√. data node의 split은 child node가 split_key를 포함하도록 split 진행

√. index node의 split은 child node가 split_key를 포함하지 않도록 split 진행

• parent node가 있는 data node의 split

```
123     if node.is_data_node:
124
125         # parent node 가 있는 data node
126         if isinstance(find_data_node_parent(tree, split_key), Node):
127             # data node 이므로 child node 가 split_key 를 포함하도록 split 진행
128             right = Node(node.degree)
129             right.p = dict(list(node.p.items())[mid:])
130             right.m = len(right.p)
131             right.r = node.r
132             right.is_index_node = node.is_index_node
133             right.is_data_node = node.is_data_node
134
135             # parent node 가 있는 경우이므로 parent node 를 찾아 split_key 추가
136             parent = find_data_node_parent(tree, split_key)
137             parent.p[split_key] = right
138             parent.p = sort_dictionary(parent.p)
139             parent.m += 1
140
141             # 기존 node 에서 split_key 와 right child 노드의 key 삭제
142             for k in list(right.p.keys()):
143                 del node.p[k]
144             node.m -= len(list(right.p.keys()))
145             node.r = right
```

```
147             # split_key 가 parent node 에서 제일 작은 key 이면
148             # 기존 node 를 parent node 의 leftmost child 로 지정
149             if split_key <= list(parent.p.keys())[0]:
150                 parent.r = node
```

Line 123 ~ 150

1. right node에 split_key보다 크거나 같은 key들 저장
2. 기존 node에서 right node의 key 삭제 후 left node로 지정

2-1. 2에서 삭제한 key의 수만큼 node의 m 감소

3. parent node가 있는 경우이므로 parent node를 찾아 split_key 삽입

• parent node가 없는 data node의 split

```
152         # parent node 가 없는 data node
153         else:
154 
155             # data node 이므로 child node 가 split_key 를 포함하도록 split 진행
156             right = Node(node.degree)
157             right.p = dict(list(node.p.items())[mid:])
158             right.m = len(right.p)
159             right.r = node.r
160             right.is_index_node = node.is_index_node
161             right.is_data_node = node.is_data_node
162 
163             # 기존 node 에서 split_key 와 right child 노드의 key 삭제
164             for k in list(right.p.keys()):
165                 del node.p[k]
166             node.m -= len(list(right.p.keys()))
167             node.r = right
168 
169             # parent node 가 없는 경우이므로 새로운 parent node 에 split_key 추가 후 root node 로 지정
170             parent.is_data_node = False
171             parent.is_index_node = True
172             parent.p[split_key] = right
173             parent.r = node
174             parent.m += 1
175             tree.root = parent
```

Line 152 ~ 175

1. right node에 split_key보다 크거나 같은 key들 저장

2. 기존 node에서 right node의 key 삭제 후 left node로 지정

2-1. 2에서 삭제한 key의 수만큼 node의 m 감소

3. parent node가 없는 경우이므로 새로운 parent node에 split_key 추가 후 root node로 지정

• parent node가 있는 index node의 split

```
177     elif node.is_index_node:
178
179         # parent node 가 있는 index node
180         if isinstance(find_index_node_parent(tree, split_key), Node):
181
182             # index node 이므로 child node 가 split_key 를 포함하지 않도록 split 진행
183             right = Node(node.degree)
184             right.p = dict(list(node.p.items())[mid + 1:])
185             right.m = len(right.p)
186             right.r = node.p[list(node.p.keys())[mid]]
187             right.is_index_node = node.is_index_node
188             right.is_data_node = node.is_data_node
189
190             # parent node 가 있는 경우이므로 parent node 를 찾아 split_key 추가
191             parent = find_index_node_parent(tree, split_key)
192             parent.p[split_key] = right
193             parent.p = sort_dictionary(parent.p)
194             parent.m += 1
195
196             # 기존 node 에서 split_key 와 right child 노드의 key 삭제
197             for k in list(right.p.keys()):
198                 del node.p[k]
199             del node.p[split_key]
200             node.m -= (len(list(right.p.keys())) + 1)
```

```
202             # split_key 가 parent node 에서 제일 작은 key 이면
203             # 기존 node 를 parent node 의 leftmost child 로 지정
204             if split_key <= list(parent.p.keys())[0]:
205                 parent.r = node
```

Line 177 ~ 205

1. right node에 split_key보다 큰 key들 저장

2. 기존 node에서 split_key와 right node의 key 삭제 후 left node로 지정

2-1. 2에서 삭제한 key의 수만큼 node의 m 감소

3. parent node가 있는 경우이므로 parent node를 찾아 split_key 삽입

• parent node가 없는 index node의 split

```
207         # parent node 가 없는 index node
208         else:
209 
210             # index node 이므로 child node 가 split_key 를 포함하지 않도록 split 진행
211             right = Node(node.degree)
212             right.p = dict(list(node.p.items())[mid + 1:])
213             right.m = len(right.p)
214             right.r = node.p[list(node.p.keys())[mid]]
215             right.is_index_node = node.is_index_node
216             right.is_data_node = node.is_data_node
217 
218             # 기존 node 에서 split_key 와 right child 노드의 key 삭제
219             for k in list(right.p.keys()):
220                 del node.p[k]
221             del node.p[split_key]
222             node.m -= (len(list(right.p.keys())) + 1)
223 
224             # parent node 가 없는 경우이므로 새로운 parent node 에 split_key 추가 후 root node 로 지정
225             parent.is_data_node = False
226             parent.is_index_node = True
227             parent.p[split_key] = right
228             parent.r = node
229             parent.m += 1
230             tree.root = parent
```

Line 207 ~ 230

1. right node에 split_key보다 큰 key들 저장

2. 기존 node에서 split_key와 right node의 key 삭제 후 left node로 지정

2-1. 2에서 삭제한 key의 수만큼 node의 m 감소

3. parent node가 없는 경우이므로 새로운 parent node에 split_key 추가 후 root node로 지정

- Single Key Search

• 공통 사항

√. while loop로 node들을 규칙에 알맞게 방문하며 방문한 노드들 출력

√. node 방문 시, node 내 key의 존재 여부에 따라 동작이 다름

• key가 node에 존재하는 경우

```
447         # key 가 node 에 존재하는 경우: 그 node 가 index_node 인 경우와 data_node 인 경우로 분할
448         if key in list(tmp_node.p.keys()):
449
450             # index_node 인 경우
451             if tmp_node.is_index_node:
452
453                 # node 의 key 값들 출력
454                 for k in list(tmp_node.p.keys()):
455                     if k == list(tmp_node.p.keys())[-1]:
456                         print(k)
457                     else:
458                         print(k, end=',')
459
460                 # 해당 key 가 가리키는 child node 로 이동
461                 tmp_node = tmp_node.p[key]
462
463             # data_node 인 경우
464             elif tmp_node.is_data_node:
465
466                 # key 에 해당하는 value 출력
467                 print(tmp_node.p[key][key], end='')
468                 # 종료
469                 break
```

Line 447 ~ 469

- index node의 경우: node의 key값들 출력 후 해당 key가 가리키는 child node로 이동

- data node의 경우: key에 해당하는 value 출력 후 while loop 종료

• key가 node에 존재하지 않는 경우: 4가지 경우로 분할

1. index node: key가 node의 모든 key들보다 작은 경우

```
473             # key 가 node 의 모든 key 들 보다 작은 경우
474             elif list(tmp_node.p.keys())[0] > key:
475
476                 # node 의 key 값들 출력
477                 for k in list(tmp_node.p.keys()):
478                     if k == list(tmp_node.p.keys())[-1]:
479                         print(k)
480                     else:
481                         print(k, end=',')
482
483                 # node 의 leftmost child 로 이동
484                 tmp_node = tmp_node.r
```

Line 473 ~ 484

- node의 key값들 출력 후 해당 node의 leftmost child로 이동

2. index node: key가 node의 key들의 제일 작은 값과 제일 큰 값 사이에 있는 경우

```
499             # key 가 node 의 key 들의 제일 작은 값과 제일 큰 값 사이에 있는 경우
500             else:
501
502                 # for loop 를 돌며 if 문을 만족하면 tmp_node 이동 후 바로 break for loop
503                 for i in range(len(list(tmp_node.p.keys())) - 1):
504
505                     # keys[i] keys[i + 1] 사이의 범위에 존재하면
506                     if list(tmp_node.p.keys())[i] < key < list(tmp_node.p.keys())[i + 1]:
507
508                         # node 의 key 값들 출력
509                         for k in list(tmp_node.p.keys()):
510                             if k == list(tmp_node.p.keys())[-1]:
511                                 print(k)
512                             else:
513                                 print(k, end=',')
514                         break
515
516                 # keys[i]가 가리키는 child 로 이동
517                 tmp_node = tmp_node.p[list(tmp_node.p.keys())[i]]
```

Line 499 ~ 517

- node 내부에서 for loop를 돌며 keys[i]와 keys[i + 1] 사이의 범위에 존재하면 node의 key값들 출력 후 keys[i]가 가리키는 child로 이동

3. index node: key가 node의 모든 key들보다 큰 경우

```
486             # key 가 node 의 모든 key 들 보다 큰 경우
487             elif key > list(tmp_node.p.keys())[-1]:
488 
489                 # node 의 key 값들 출력
490                 for k in list(tmp_node.p.keys()):
491                     if k == list(tmp_node.p.keys())[-1]:
492                         print(k)
493                     else:
494                         print(k, end=',')
495 
496                 # node 의 rightmost child 로 이동
497                 tmp_node = tmp_node.p[list(tmp_node.p.keys())[-1]]
```

Line 486 ~ 497

- node의 key값들 출력 후 해당 node의 rightmost child로 이동

4. data node

```
440             # data_node 에 도달했는데 key 가 없을 경우
441             if tmp_node.is_data_node and key not in list(tmp_node.p.keys()):
442                 # print 'NOT FOUND'
443                 print("NOT FOUND", end='')
444                 # 종료
445                 break
```

Line 440 ~ 445

- "NOT FOUND" print 후 while loop 종료

- Range Search

- 기본 사항

  - leaf node인 start node와 end node를 찾은 후 그 사이의 모든 <key,value> 값들 출력

- start node와 end node 지정

```
527         start_node = tree.root
528         # start_node 에 start_key 이상이면서 제일 작은 key 를 가진 leaf node 를 저장
529         while not start_node.is_data_node:
530             if start_key in list(start_node.p.keys()):
531                 start_node = start_node.p[start_key]
532 
533             elif len(start_node.p.keys()) == 1:
534                 if start_key < list(start_node.p.keys())[0]:
535                     start_node = start_node.r
536                 else:
537                     start_node = start_node.p[list(start_node.p.keys())[0]]
538 
539             else:
540                 if start_key < list(start_node.p.keys())[0]:
541                     start_node = start_node.r
542                 elif start_key > list(start_node.p.keys())[-1]:
543                     start_node = start_node.p[list(start_node.p.keys())[-1]]
544                 else:
545                     for i in range(len(list(start_node.p.keys())) - 1):
546                         if list(start_node.p.keys())[i] < start_key < list(start_node.p.keys())[i + 1]:
547                             start_node = start_node.p[list(start_node.p.keys())[i]]
548                             break
```

Line 527 ~ 571

(start node와 end node의 지정 알고리즘이 같으므로 코드는 start node의 경우만 첨부)

* p.8의 "tree에 index node와 data node 모두 존재하는 경우의 insertion" 경우와 동일

  - key가 index node의 모든 key들보다 작으면 index node의 leftmost child로 이동

  - key가 index node의 key들 사이 범위에 존재하면

    오름차순으로 정렬되어 있는 node의 p를 한 칸씩 돌며

    key가 속하는 범위에 해당하는 child로 이동

  - key가 index node의 모든 key들보다 크면 index node의 rightmost child로 이동

  - key가 탐색 도중 index node에 존재하면 그 key가 가리키는 node로 이동

  - 탐색 도중 data node에 도달하면 탐색 종료

  √. start node와 end node는 모두 leaf node

• start node부터 end node까지 leaf node 출력 시작

```
573         # start_node 와 end_node 가 같아질 때까지 start_(key,value) 부터 출력 시작
574         # start_node 와 end_node 가 처음부터 같을 경우는 while 문 실행하지 않음
575         while start_node is not end_node:
576             real_start_key = 0
577             start_key_in_node = False
578             count = 0
579             for i in range(len(list(start_node.p.keys()))):
580                 if start_key > list(start_node.p.keys())[i]:
581                     real_start_key = list(start_node.p.keys())[i]
582                     count += 1
583                 elif start_key == list(start_node.p.keys()):
584                     real_start_key = start_key
585                     start_key_in_node = True
```

Line 573 ~ 585

- start node의 keys 중 입력받은 start key 이상이며 최소인 key를 real start key로 지정

```
586             # start_key 가 node 의 모든 key 들보다 작은 경우
587             if real_start_key == 0:
588                 for k in list(start_node.p.keys()):
589                     print(str(k) + "," + str(start_node.p[k][k]))
590                 start_node = start_node.r
591             else:
592                 # start_key 가 node 안에 존재하는 경우
593                 if start_key_in_node:
594                     for i in range(count, len(list(start_node.p.keys()))):
595                         k = list(start_node.p.keys())[i]
596                         print(str(k) + "," + str(start_node.p[k][k]))
597                     start_node = start_node.r
598                 else:
599                     # start_key 가 node 의 모든 key 들보다 큰 경우
600                     if count == tree.degree - 1:
601                         start_node = start_node.r
602                     # start_key 가 node 안에는 존재하지 않지만 key 들 사이에 존재하는 경우
603                     else:
604                         for i in range(count, len(list(start_node.p.keys()))):
605                             k = list(start_node.p.keys())[i]
606                             print(str(k) + "," + str(start_node.p[k][k]))
607                         start_node = start_node.r
```

Line 586 ~ 607

- real start key가 node의 모든 key들보다 작으면 node의 모든 <key,value> pairs 출력 후 다음 node로 이동

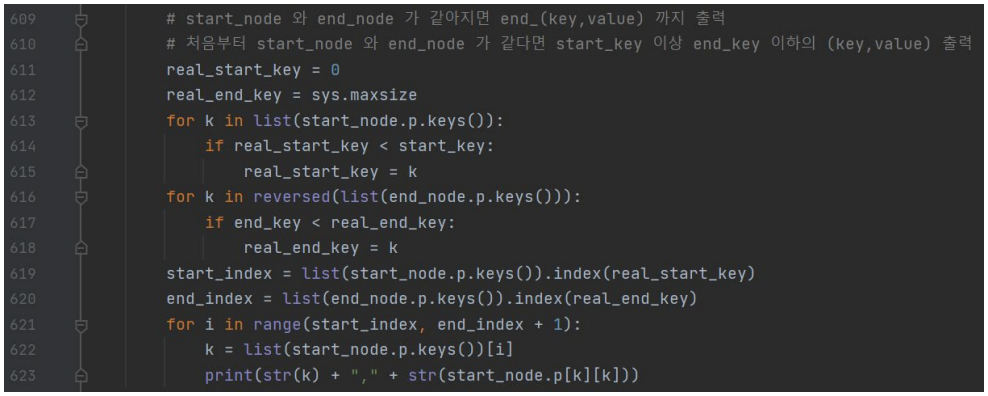
- real start key가 node의 key 값들 사이에 존재하면

real start key부터 node의 마지막 <key,value> pairs까지 출력 후 다음 node로 이동

- real start key가 node의 모든 key들보다 크면 출력하지 않고 다음 node로 이동

√. 위 코드는 start node와 end node가 같지 않을 때까지(또는 같지 않을 때)만 작동

• start node와 end node가 같아졌을 때(또는 처음부터 같았을 때)



```
609         # start_node 와 end_node 가 같아지면 end_(key,value) 까지 출력
610         # 처음부터 start_node 와 end_node 가 같다면 start_key 이상 end_key 이하의 (key,value) 출력
611         real_start_key = 0
612         real_end_key = sys.maxsize
613         for k in list(start_node.p.keys()):
614             if real_start_key < start_key:
615                 real_start_key = k
616         for k in reversed(list(end_node.p.keys())):
617             if end_key < real_end_key:
618                 real_end_key = k
619         start_index = list(start_node.p.keys()).index(real_start_key)
620         end_index = list(end_node.p.keys()).index(real_end_key)
621         for i in range(start_index, end_index + 1):
622             k = list(start_node.p.keys())[i]
623             print(str(k) + "," + str(start_node.p[k][k]))
```

Line 609 ~ 623

- start node의 key값들 중 입력 받은 start key 이상이며 최소인 key를 real start key로 지정

- end node의 key 값들 중 입력 받은 end key 이하이며 최대인 key를 real end key로 지정

- real start key부터 real end key까지의 <key,value> pairs 출력

3. Other Functions

- sort_dictionary

```
40  # node 에 key 삽입 시 ASCENDING order 로 저장하기 위해 사용
41  def sort_dictionary(dictionary: dict) -> dict:
42      sorted_tuple = sorted(dictionary.items())
43      sorted_dict = dict((key, value) for key, value in sorted_tuple)
44      return sorted_dict
```

Line 40 ~ 44

- node에 key 삽입 시 ASCENDING order로 저장하기 위해 사용
- dictionary의 sort 함수는 sorting 후 tuple의 형태로 반환
- sorting 된 tuple을 다시 dictionary 형태로 변형 후 반환

- creation

```
35  def creation(degree: int) -> bPlusTree:
36      new_tree = bPlusTree(degree)
37      return new_tree
```

Line 35 ~ 37

- new_tree를 degree에 맞게 생성 후 반환

- main function: Data File Creation

```
355 # Data File Creation
356 if sys.argv[1] == "-c":
357     index_file = sys.argv[2]
358     degree = int(sys.argv[3])
359     tree = creation(degree)
360     with open(index_file, 'w') as f_index:
361         f_index.write("B+ tree with node size:" + str(degree) + "\n")
```

Line 355 ~ 361

- degree를 입력받아 tree 생성 후 index_file에 degree 입력

- main function: Insertion of Keys

```
363     # Insertion of Keys
364     elif sys.argv[1] == "-i":
365         index_file = sys.argv[2]
366         data_file = sys.argv[3]
367 
368         # index_file 에 저장되어있던 tree 호출
369         tree = index_file_to_tree(index_file)
370 
371         # key,value pairs 저장
372         keys = list()
373         values = list()
374 
375         with open(data_file, 'r') as f_data:
376             while True:
377                 line = f_data.readline()
378                 if not line:
379                     break
380                 key = int(line.split(',')[0])
381                 value = int(line.split(',')[1])
382                 keys.append(key)
383                 values.append(value)
```

```
385         for i in range(len(keys)):  # keys 와 values 의 길이는 반드시 같음
386             tree = insertion(tree, keys[i], values[i])
387 
388         with open(index_file, 'w') as f_index:
389             f_index.write("B+ tree with node size:" + str(tree.degree) + "\n")
390 
391         with open(index_file, 'a') as f_index:
392             f_index.write("\n")
393             f_index.write("key,value pairs")
394             f_index.write("\n")
395 
396         tree_to_index_file(tree, index_file)
```

Line 363 ~ 396

- data_file에서 한 줄마다 comma(,)로 split해

앞의 숫자는 keys(list)에 뒤의 숫자는 values(list)에 저장

- <key,value> pairs insertion 진행
- index_file에 degree 입력
- tree의 <key,value> pairs를 index_file에 저장

- find_data_node_parent

```python
def find_data_node_parent(tree: bPlusTree, split_key: int) -> Node:
    tmp_node = tree.root
    parent_node = None

    # tmp_node 가 data_node 가 될 때까지 반복
    while not tmp_node.is_data_node:
        parent_node = tmp_node

        # split_key 가 tmp_node 의 모든 key 들보다 작을 경우 tmp_node 의 leftmost child 로 이동
        if list(tmp_node.p.keys())[0] > split_key:
            tmp_node = tmp_node.r

        # split_key 가 tmp_node 의 모든 key 들보다 클 경우 tmp_node 의 rightmost child 로 이동
        elif split_key > list(tmp_node.p.keys())[-1]:
            tmp_node = tmp_node.p[list(tmp_node.p.keys())[-1]]

        # split_key 가 tmp_node 의 keys[i] keys[i + 1] 사이의 범위에 존재하면 keys[i]가 가리키는 child 로 이동
        else:
            # for loop 를 돌며 if 문을 만족하면 tmp_node 이동 후 바로 break for loop
            for i in range(len(list(tmp_node.p.keys())) - 1):
                if list(tmp_node.p.keys())[i] < split_key < list(tmp_node.p.keys())[i + 1]:
                    tmp_node = tmp_node.p[list(tmp_node.p.keys())[i]]
                    break

    return parent_node
```

Line 49 ~ 73

* p.8의 "tree에 index node와 data node 모두 존재하는 경우의 insertion"

  p.17의 "start node와 end node 지정"의 두 경우와 동일한 알고리즘

- key가 index node의 모든 key들보다 작으면 index node의 leftmost child로 이동
- key가 index node의 key들 사이 범위에 존재하면

  오름차순으로 정렬되어 있는 node의 p를 한 칸씩 돌며

  key가 속하는 범위에 해당하는 child로 이동
- key가 index node의 모든 key들보다 크면 index node의 rightmost child로 이동
- key가 탐색 도중 index node에 존재하면 그 key가 가리키는 node로 이동
- 탐색 도중 data node에 도달하면 탐색 종료

√. while loop의 진행에 따라 tmp_node가 data node가 될 시에 종료되고,

parent_node는 tmp_node가 data node가 되기 직전의 node

- find_index_node_parent

```python
# index_node 의 parent_node 반환
# 해당 node 가 root node 일 경우 None 반환
def find_index_node_parent(tree: bPlusTree, split_key: int) -> Node:
    tmp_node = tree.root
    parent_node = None
    while True:
        # while loop 진행 도중 split_key 가 tmp_node 에 존재하는 경우 break while loop
        if split_key in list(tmp_node.p.keys()):
            break

        # 다음 while loop 가 line 82 ~ 83 에 의해 break 된다면 반환해줄 parent node 지정
        parent_node = tmp_node

        # split_key 가 tmp_node 의 모든 key 들보다 작을 경우 tmp_node 의 leftmost child 로 이동
        if list(tmp_node.p.keys())[0] > split_key:
            tmp_node = tmp_node.r
        # split_key 가 tmp_node 의 모든 key 들보다 클 경우 tmp_node 의 rightmost child 로 이동
        elif split_key > list(tmp_node.p.keys())[-1]:
            tmp_node = tmp_node.p[list(tmp_node.p.keys())[-1]]

        # split_key 가 tmp_node 의 keys[i] keys[i + 1] 사이의 범위에 존재하면 keys[i]가 가리키는 child 로
        else:
            # for loop 를 돌며 if 문을 만족하면 tmp_node 이동 후 바로 break for loop
            for i in range(len(list(tmp_node.p.keys())) - 1):
                if list(tmp_node.p.keys())[i] < split_key < list(tmp_node.p.keys())[i + 1]:
                    tmp_node = tmp_node.p[list(tmp_node.p.keys())[i]]
                    break

    # 처음부터 split_key 가 tmp_node 에 존재해 while loop 가 바로 종료되었다면
    # split_key 가 root node 에 존재하는 것이므로 None 반환
    if tmp_node is tree.root:
        return None

    return parent_node
```

Line 76 ~ 109

* 위 find_data_node_parent와 거의 동일한 알고리즘

- key가 index node의 모든 key들보다 작으면 index node의 leftmost child로 이동

- key가 index node의 key들 사이 범위에 존재하면

오름차순으로 정렬되어 있는 node의 p를 한 칸씩 돌며

key가 속하는 범위에 해당하는 child로 이동

- key가 index node의 모든 key들보다 크면 index node의 rightmost child로 이동

- key가 탐색 도중 index node에 존재하면 while loop 종료

√. while loop의 진행에 따라 tmp_node에 key가 존재할 시에 종료되고,

parent_node는 tmp_node에 key가 존재하기 직전의 node

- index_file_to_tree

```python
# index_file 을 tree 로 옮겨주는 함수
def index_file_to_tree(index_file: str) -> bPlusTree:
    existed_tree = bPlusTree()
    with open(index_file, 'r') as f_index:

        # 기존 노드와 degree 가 같은 빈 tree 생성
        first_line = f_index.readline()
        degree = int(first_line.split(':')[1])
        existed_tree.degree = degree
        existed_tree.root = Node(degree)

        # key,value pairs 저장
        dummy = "dummy string"
        keys = list()
        values = list()
        while True:
            dummy = f_index.readline()
            if dummy == "key,value pairs\n" or not dummy:
                break
```

```python
        while True:
            line = f_index.readline()
            if not line:
                break
            key = int(line.split(',')[0])
            value = int(line.split(',')[1])
            keys.append(key)
            values.append(value)

        for i in range(len(keys)):  # keys 와 values 의 길이는 반드시 같음
            existed_tree = insertion(existed_tree, keys[i], values[i])

    return existed_tree
```

Line 309 ~ 341

- 기존의 index_file에서 degree를 읽어 tree의 degree로 지정
- <key,value> pairs가 나올 때까지 dummy로 readline()
- 한 줄마다 comma(,)로 split해 앞의 숫자는 keys(list)에 뒤의 숫자는 values(list)에 저장
- <key,value> pairs insertion 진행
- existed tree 반환

- tree_to_index_file

```
344  # insert 완료한 tree 를 index_file 에 저장
345  def tree_to_index_file(tree: bPlusTree, index_file: str) -> None:
346      tmp_tree = tree
347      with open(index_file, 'a') as f_index:
348          for k in list(tmp_tree.key_values.keys()):
349              f_index.write(str(k) + "," + str(tmp_tree.key_values[k][k]) + "\n")
```

Line 344 ~ 349

- tree의 <key,value> pairs를 index_file에 저장