DataBase System

B+Tree Wiki

담당 교수 : 김상욱 교수님

2022076062

컴퓨터소프트웨어학부

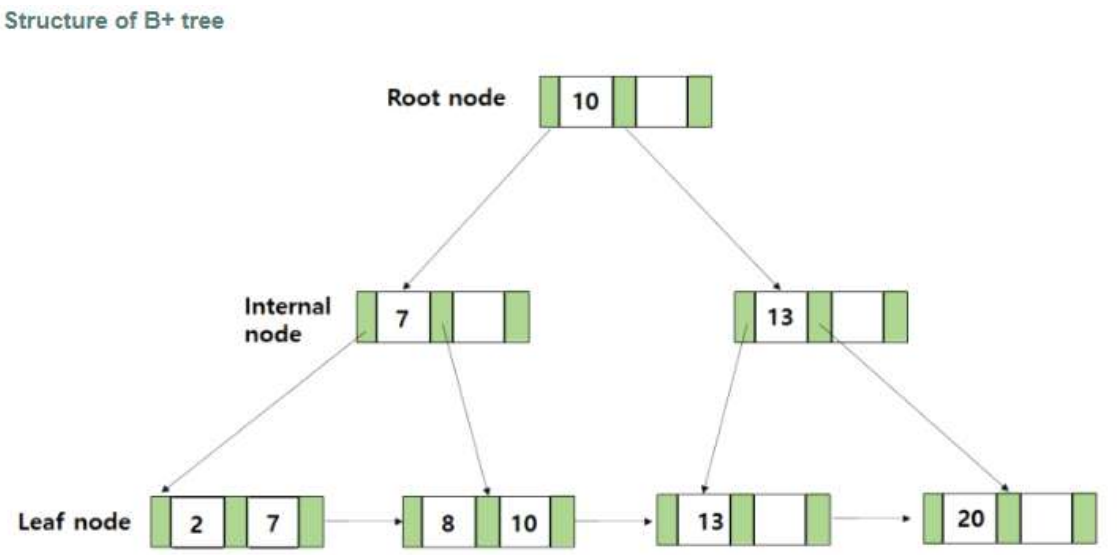
김유찬

개요

B+트리란 B-트리의 응용버전으로 삽입 삭제에 어려움이 있지만 데이터 검색에 효율적인 알고리즘이다. 우선 아래는 B+트리의 차수에 따른 노드의 개수에 대한 정보이다.

- **내부 노드**
  - * 최대 키 수 : m-1
  - * 최소 키 수 : ceil(m/2) -1
- **리프노드**
  - * 최대 키 수 : m-1 or m
  - * 최소 키 수 : ceil(m/2) - 1

B+트리의 대략적인 구조는 아래 사진과 같다
B-트리와 다른점은 리프노드끼리 연결리스트로 연결되어 있고 내부 노드에 리프 노드의 값이 들어있다는 것이다.



출처 : https://blog.naver.com/jokercsi1/222405659083

코드설명

전체적인 구조

Node 클래스

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.leftchild = None
        self.rightchild = None
```

key : 각 Node의 키

value : key에 따른 value값

leftchild : Node의 왼쪽 자식

rightchild : Node의 오른쪽 자식

이후에 많이 이용하는 핵심 부분은 leftchild와 rightchild가 None일 때는 리프노드에만 있는 것이고 leftchild와 rightchild가 값을 가지면 내부 노드에도 있다는 뜻이다. leftchild와 rightchild는 노드의 왼쪽 Nodelist와 오른쪽 Nodelist를 가리키는 변수이다.

Nodelist 클래스

```
class NodeList:
    def __init__(self, size, file):
        self.head = None
        self.parent = None
        self.isleaf = True
        self.size = size
        self.file = file
        self.nodelist = []
        self.next = None # 리프 노드일 때 다음 노드리스트 가리킴
        # print("bptree 생성 size : " + size + " file명 : " + file)
```

head : bptree의 head가 되는 부분이다(root)

parent : 해당 Nodelist의 부모 Nodelist를 가리킨다

isleaf : 현재 리프노드인지 판단(없어도 됐었다)

size : Nodelist가 가질 수 있는 자식의 수

file : bptree가 들어있는 파일

nodelist : Node가 들어있는 리스트

next : 리프 노드일 때 다음 리프 노드를 가리키는 변수

파일 생성

```
# data file 만들기
if sys.argv[1] == "-c":
    with open(sys.argv[2], 'w') as file:
        node_size = sys.argv[3]
        file.write("node_size : " + node_size)
        create_bptree(node_size, sys.argv[2])
```

명령어 예시 : python b+tree.py -c index.dat 4

argv[2]에 입력받은 파일에 bptree를 생성시키는 효과를 보이는 역할

파일 맨 윗 줄에 node_size 출력

```
# bptree 생성
def create_bptree(node_size, file):
    bptree = NodeList(node_size, file)
    bptree.head = bptree # 자기 자신을 가리킴
    # pickle 파일에 추가
    # print("pickle 파일에 추가")
    update_bptree_list(bptree)
```

def create_bptree(node_size, file)

: Nodelist를 새로 만들고 head 설정 후 pickle에 업데이트

```
# pickle파일에서 bptree list 불러오기
def load_bptree_list():
    try:
        with open('bptree_list.pkl', 'rb') as file:
            # print("bplist 불러오기")
            return pickle.load(file)
    # 파일이 없으면 빈 리스트 반환
    except FileNotFoundError:
        # print("빈 bplsit 생성")
        return []
```

def load_bptree_list()

: pickle 파일에 저장된 리스트 불러오기

리스트에 저장된 건 bptree의 루트이다

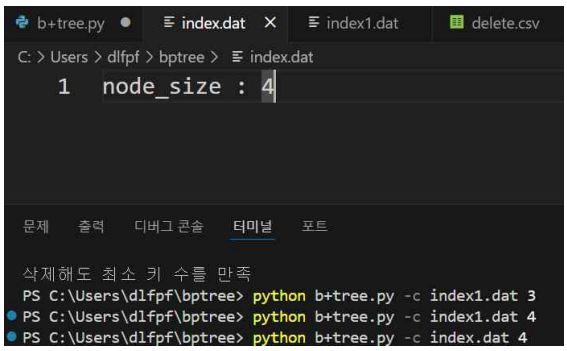
리스트가 없다면 빈 리스트를 생성한다

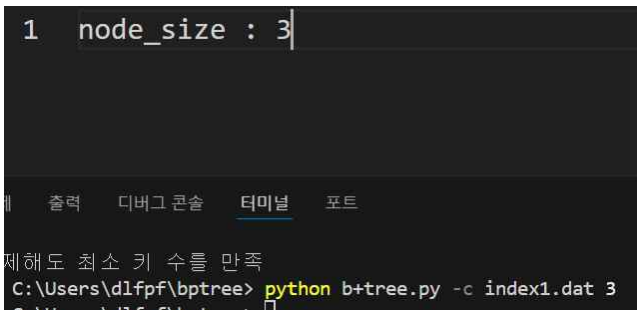
```python
# pickle 파일에 bptree 추가하기(업데이트)
def update_bptree_list(bptree):
    bptree_list = load_bptree_list()
    # print("업데이트 전 : ", end = '')
    # print(bptree_list)
    updated = False
    # 기존 pickle 파일에 bptree 있으면 업데이트
    for i in range(len(bptree_list)):
        if bptree_list[i].file == bptree.file:
            bptree_list[i] = bptree
            updated = True
            # print("기존 bptree 덮음 : " + bptree.file)
            break
    # 기존 pickle에 없으면 bptree 추가
    if not updated:
        bptree_list.append(bptree)
        # print("bptree 새로 추가")
    # print("업데이트 후 : ", end = '')
    # print(bptree_list)
    # 업데이트 된 리스트를 다시 pickle 파일에 저장
    with open('bptree_list.pkl', 'wb') as file:
        pickle.dump(bptree_list, file)
    # print("bptree_list 업데이트")
```

def update_bptree_list(bptree)

: bptree list를 업데이트 하는 함수 후에 삽입, 삭제 등 과정을 했을 때 bptree의 형태가 변하므로 그 bptree로 업데이트 해주는 역할

출력결과





index.dat에는 node_size:4 index1.dat에는 node_size3이 저장된 모습이다.

삽입하기

```
# data file에 삽입하기
elif sys.argv[1] == "-i":
    node_list = []
    bptree_list = load_bptree_list()  # bptree_list 불러오기
    bptree = load_bptree(bptree_list, sys.argv[2])
    with open(sys.argv[2], 'r') as file:
        node_size = int(file.readline()[11:])

    with open(sys.argv[3], 'r') as file:
        reader = csv.reader(file)
        for row in file:
            key, value = map(int, row.strip().split(','))
            node = Node(key, value)
            node_list.append(node)
    # print("넣어야할 노드들 : ", end = '')
    # print(node_list)

    for node in node_list:
        fnodelist = bptree.head.find_position(node.key)
        # print("삽입시작" + str(node.key))
        bptree.insert(fnodelist, node)
    update_bptree_list(bptree)
    bptree.head.write_bptree_to_file(sys.argv[2])
```

명령어 : python b+tree.py index.dat input.csv

: 우선 bptree list를 불러온 후 삽입하고 쓸 file 정보와 비교하여 bptree를 찾아낸다. input.csv 파일을 한 줄씩 읽고 Nodelist에 key, value를 가진 Node를 생성 후 append 해준다. fnodelist는 Node가 들어갈 Nodelist이고 find_position 함수를 이용한다. bptree.head에서 시작하면 루트부터 시작하여 Node가 들어갈 leaf node를 발견하고 리턴해준다. 이후 반복문을 돌며 bptree에 하나씩 삽입하고 bptree list를 업데이트 해준다 그리고 bptree를 input.dat 파일에 출력해주는 함수를 호출한다.

```
#삽입할 노드 리스트를 찾는 함수
def find_position(self, key):
    current = self
    if current.isleaf:
        return current

    for i in range(len(current.nodelist)):
        # 왼쪽 자식으로 이동
        if key < current.nodelist[i].key:
            current = current.nodelist[i].leftchild
            break
        # 맨 마지막이면 오른쪽 자식으로 이동
        elif i == len(current.nodelist) - 1:
            current = current.nodelist[i].rightchild
        # 다음 노드로 이동
        else:
            continue
    # 자식 노드에서 다시 위치 찾기
    return current.find_position(key)
```

def find_position(self, key)

자기가 들어갈 Nodelist를 찾는 함수 head부터 시작해서 현재 Nodelist가 리프 노드가 아니면 어디로 내려가야할지 찾는다. Nodelist에 저장된 nodelist의 노드들의 key값을 비교하며 이동해준다.

```
# 삽입 함수
def insert(self, fnodelist, node):
    # 리프 노드 리스트에 넣고 오름차순 정렬
    print("일단 노드 리스트에 넣고 정렬")
    print("넣을 노드")
    for i in fnodelist.nodelist:
        print(i.key)
    fnodelist.nodelist.append(node)
    fnodelist.nodelist.sort(key = lambda x : x.key)
    if not fnodelist.isleaf:
        fnodelist.rearrange()
    print("노드 리스트 길이 : " + str(len(fnodelist.nodelist)))
    # key가 size만큼 있으면 분리해야함
    if len(fnodelist.nodelist) == int(fnodelist.size):
        print("분리 시작")
        self.split(fnodelist)
```

알고리즘

일단 nodelist에 넣고 node들을 key값에 따라 정렬시킨다. 만약 fnodelist에 삽입했는데 size와 같아진다면 분리를 시작한다.

```
65      # 분리하는 함수
66      def split(self, fnodelist):
67          # 부모가 없다면 새로운 부모 노드 리스트 생성
68          if fnodelist.parent is None:
69              parentnodelist = NodeList(self.size, self.file)
70              parentnodelist.isleaf = False
71              self.head = parentnodelist # 트리의 루트 바꿔주기
72          # 부모가 있다면 그거 가져오기
73          else:
74              parentnodelist = fnodelist.parent
75          # 리프 노드 리스트를 분리할 때
76          if fnodelist.isleaf:
77              newnodelist = NodeList(self.size, self.file)
78              split_point = int(self.size) // 2
79              # 슬라이싱을 사용해 절반 분리
80              newnodelist.nodelist = fnodelist.nodelist[split_point:]
81              fnodelist.nodelist = fnodelist.nodelist[:split_point]
```

```
            # 리프 노드 리스트끼리 연결
            if fnodelist.next:
                newnodelist.next = fnodelist.next
                fnodelist.next = newnodelist
            else:
                fnodelist.next = newnodelist
            # 중간 값을 부모로 올림
            mid = newnodelist.nodelist[0]
            # print("중간 키 : " + str(mid.key))
            mid.leftchild = fnodelist
            mid.rightchild = newnodelist
            fnodelist.parent = parentnodelist
            newnodelist.parent = parentnodelist
            # parentnodelist에 mid 삽입
            self.insert(parentnodelist, mid)
        # 부모 노드 리스트를 분리할 때
        else:
            print("부모 노드 분리")
            newnodelist = NodeList(self.size, self.file)
            newnodelist.isleaf = False
            split_point = int(self.size) // 2 + 1
            # 부모를 가리키는 변수 재조정
            for i in range(split_point - 1, len(fnodelist.nodelist)):
                fnodelist.nodelist[i].rightchild.parent = newnodelist
            print("분리 지점 : " + str(split_point))
            # 슬라이싱을 사용해 절반 분리
            newnodelist.nodelist = fnodelist.nodelist[split_point:]
            print("오른쪽 노드")
            for i in newnodelist.nodelist:
                print(i.key)
            fnodelist.nodelist = fnodelist.nodelist[:split_point]
            mid = fnodelist.nodelist.pop()
            print("왼쪽 노드")
            for i in fnodelist.nodelist:
                print(i.key)
            # 중간값을 없애고 부모로 올림
            print("중간 키 : " + str(mid.key))
            mid.leftchild = fnodelist
            mid.rightchild = newnodelist
            fnodelist.parent = parentnodelist
            newnodelist.parent = parentnodelist
            # parentnodelist에 mid 삽입
            self.insert(parentnodelist, mid)
```

split(self, fnodelist)

부모가 없다면 새로운 부모 노드 리스트를 생성해주고 트리의 루트를 바꿔준다. 부모가 있다면 그걸 가져온다. 리프노드를 분리할 때와 부모노드를 분리하는 경우를 따로 작성한다. 슬라이싱의 차이가 있어 split_point가 다르기 때문이다. 각자 잘 분리해주며 leftchild, rigthchild, parent, next 값을 적절히 넣는다. 두 경우 모두 parent에 node를 새로 넣는 insert 함수를 호출한다. 위에서 insert함수를 보면 리프 노드가 아닐 때 rearrange함수를 호출하여 정리를 해준다.

```
# 부모의 자식들 재조정
def rearrange(self):
    for i in range(len(self.nodelist) - 1):
        self.nodelist[i+1].leftchild = self.nodelist[i].rightchild
```

rearrange(self)

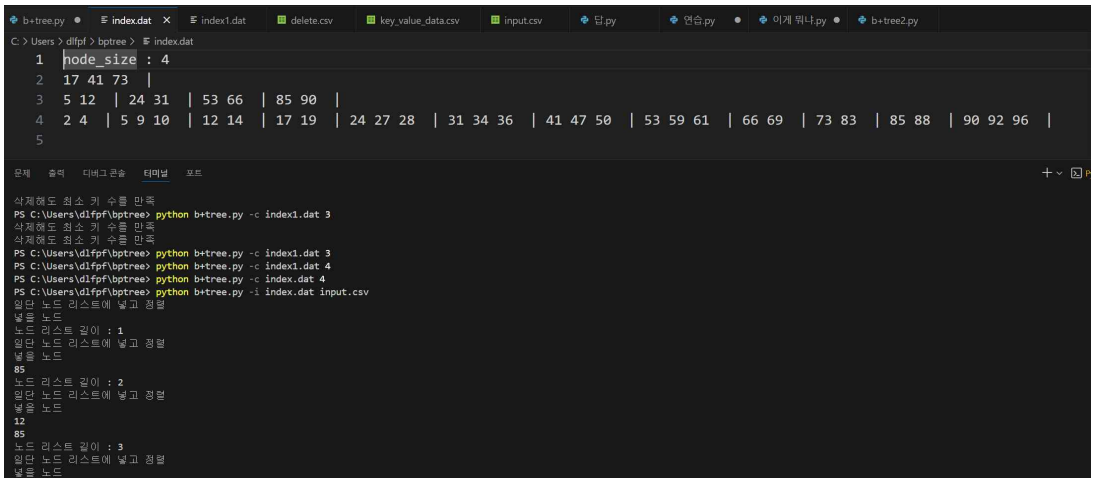
현재 중간에 Node를 새로 넣어 Nodelist의 자식 관계가 엉켜 있으므로 자식을 맞춰주는 함수이다.

```python
# bptree를 파일에 쓰기
def write_bptree_to_file(self, ifile):
    size = 1
    new_size = 0
    dq = deque()
    dq.append(self)
    with open(ifile, 'r') as file:
        node_size = file.readline()
    with open(ifile, 'w') as file:
        file.write(node_size + '\n')
        # BFS 방식으로 트리 순회
        while dq:
            for i in range(size):
                current = dq.popleft()
                # 노드의 key 기록
                for j in range(len(current.nodelist)):
                    file.write(str(current.nodelist[j].key) + ' ')
                    # 자식 노드를 큐에 추가(리프 노드일 때는 추가하지 않음)
                    if not current.isleaf:
                        if current.nodelist[j].leftchild:
                            dq.append(current.nodelist[j].leftchild)
                            new_size += 1
                        if j == len(current.nodelist) - 1 and current.nodelist[j].rightchild:
                            dq.append(current.nodelist[j].rightchild)
                            new_size += 1
                file.write(' | ')
            file.write('\n')
            size = new_size
            new_size = 0
```

write_bptree_to_file(self, ifile)

bptree에 삽입이 끝났으면 index.dat 파일에 bptree의 형태를 출력해주는 함수이다. head에서 시작해서 큐에 넣는 방식으로 BFS방법을 통해 구현했다. Nodelist의 구분은 '|'이며 각 노드는 띄어쓰기로 구분했다. Nodelist의 레벨은 줄바꿈 형태로 표현했다.

출력 결과



찾기

single search

```
450  # single key 찾기
451  elif sys.argv[1] == '-s':
452      bptree_list = load_bptree_list()  # bptree_list 불러오기
453      bptree = load_bptree(bptree_list, sys.argv[2])
454      search_key = int(sys.argv[3])
455      # print(str(search_key)+"찾기")
456      bptree.head.single_search(search_key)
```

명령어 : python b+tree.py -s index.dat 41

bptree.head에서 시작해서 single_search를 하도록 구현했다. fnodelist를 찾는 find_position과 유사하게 구현했다.

```
29  class NodeList:
173     # single key 찾기
174     def single_search(self, search_key):
175         current = self
176         # leaf노드 도달하면 여기서 value 구하기
177         if current.isleaf:
178             found = False
179             print("leaf노드 도달")
180             for i in current.nodelist:
181                 if i.key == search_key:
182                     print("found : " + str(i.value))
183                     found = True
184                     break
185             if not found:
186                 print("Not found")
187             return
188         # internal 노드면 해당 노드 리스트 키값 전부 출력
189         else:
190             for i in self.nodelist: # 출력
191                 print(i.key, end = ' ')
192             print('')
193             for i in range(len(current.nodelist)): # 이동
194                 # 왼쪽 자식으로 이동
195                 if search_key < current.nodelist[i].key:
196                     current = current.nodelist[i].leftchild
197                     break
198                 # 맨 마지막이면 오른쪽 자식으로 이동
199                 elif i == len(current.nodelist) - 1:
200                     current = current.nodelist[i].rightchild
201                 else: # 다음 노드로 이동
202                     continue
203             # 자식 노드에서 다시 위치 찾기
204             return current.single_search(search_key)
```

single_search(self, search_key)

리프 노드에 들어가면 Nodelist를 돌아 key를 검색한다. key를 찾으면 key에 해당하는 value를 출력하고 없으면 Not found를 출력한다. 리프노드가 아닐 때는 해당 Nodelist의 nodelist에 들어있는 Node.key를 모두 출력한다
다음 노드로 가는 방법은 find_position과 동일하다

출력결과

```
PS C:\Users\dlfpf\bptree> python b+tree.py -s index.dat 41
17 41 73
53 66
leaf노드 도달
found : 431142
PS C:\Users\dlfpf\bptree> python b+tree.py -s index.dat 100
17 41 73
85 90
leaf노드 도달
Not found
PS C:\Users\dlfpf\bptree>
```

ranged search

```
458  # 범위 찾기
459  elif sys.argv[1] == '-r':
460      bptree_list = load_bptree_list()  # bptree_list 불러오기
461      bptree = load_bptree(bptree_list, sys.argv[2])
462      search_key_front = int(sys.argv[3])
463      search_key_rear = int(sys.argv[4])
464      # print(str(search_key_front) + "에서" + str(search_key_rear) + "까지 찾기")
465      bptree.head.ranged_search(search_key_front, search_key_rear)
```

명령어 : python b+tree.py -r input.dat 30 80

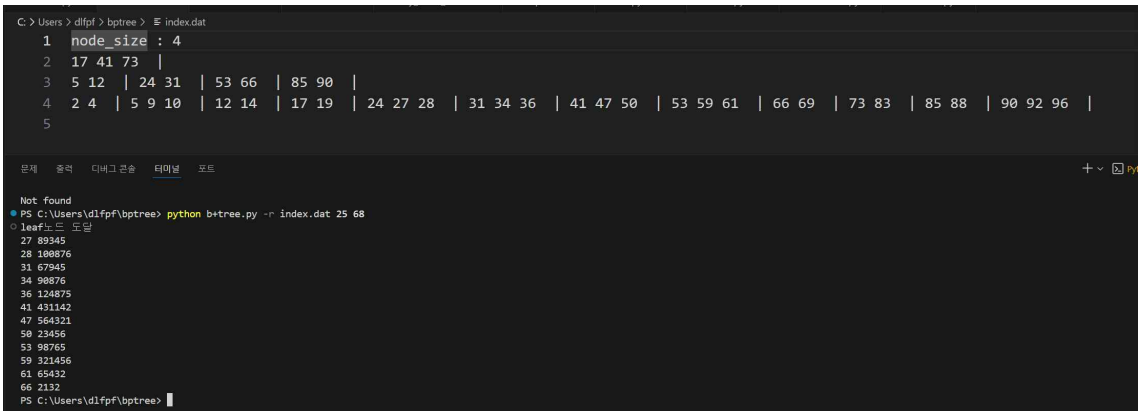
single search와 마찬가지로 head에서부터 찾아준다

```
206      # 범위 찾기
207      def ranged_search(self, front, rear):
208          current = self
209          check = False
210          # leaf노드 도달하면 여기서 범위 key, value 구하기
211          if current.isleaf:
212              print("leaf노드 도달")
213              while current:
214                  for i in current.nodelist:
215                      if i.key >= front and i.key <= rear:
216                          print(i.key, i.value)
217                      if i.key > rear:
218                          check = True
219                          break
220                  current = current.next
221                  if check:
222                      break
223              return
224          else: # front 가 있을 리프 노드로 이동하기
225              for i in range(len(current.nodelist)): # 이동
226                  # 왼쪽 자식으로 이동
227                  if front < current.nodelist[i].key:
228                      current = current.nodelist[i].leftchild
229                      break
230                  # 맨 마지막이면 오른쪽 자식으로 이동
231                  elif i == len(current.nodelist) - 1:
232                      current = current.nodelist[i].rightchild
233                  # 다음 노드로 이동
234                  else:
235                      continue
236              # 자식 노드에서 다시 위치 찾기
237              return current.ranged_search(front, rear)
```

ranged_search(self, front, rear)

single search 와 비슷하지만 rear가 있다는 것이 다르다. single search를 하듯이 front를 기준으로 제일 왼쪽 front보다 크거나 같은 Nodelist를 찾는다. 이후에 Nodelist 안에 nodelist를 돌며 key, value를 호출한다. Nodelist의 nodelist가 끝에 도달했다면 next로 넘어가고 rear보다 작을 때까지 계속한다

출력결과



삭제하기(일부분 미완성)

```python
# data file에 삭제하기
elif sys.argv[1] == '-d':
    key_list = []
    bptree_list = load_bptree_list()  # bptree_list 불러오기
    bptree = load_bptree(bptree_list, sys.argv[2])
    with open(sys.argv[2], 'r') as file:
        node_size = int(file.readline()[11:])

    with open(sys.argv[3], 'r') as file:
        reader = csv.reader(file)
        for row in file:
            key = int(row)
            key_list.append(key)
    # print("삭제할 키들 : ", end = '')
    # print(key_list)

    for key in key_list:
        fnodelist = bptree.head.find_position(key)
        print("삭제" + str(key))
        bptree.delete(fnodelist, key)
    update_bptree_list(bptree)
    bptree.head.write_bptree_to_file(sys.argv[2])
```

명령어 :python b+tree.py -d index.dat delete.csv

insert함수와 비슷하게 nodelist에 key, value를 가진 node를 넣고 반복문을 돌렸다. 이후 bptree를 업데이트하고 데이터파일에 출력했다.

```
    # 삭제 함수
    def delete(self, fnodelist, key):
        index = 0
        print(str(key)+"삭제")
        # 노드리스트에서 삭제할 노드의 인덱스 구하기
        for i in range(len(fnodelist.nodelist)):
            if fnodelist.nodelist[i].key == key:
                index = i
                print("leaf 노드에서 인덱스" + str(index))
                break

        # 삭제해도 최소 키 수를 만족할 때
        if len(fnodelist.nodelist) > math.ceil(int(fnodelist.size)/2) - 1:
            print("삭제해도 최소 키 수를 만족")
            # internal node이면 다음 노드랑 internal 노드랑 교체
            if fnodelist.nodelist[index].leftchild:
                pindex = fnodelist.nodelist[index].find_my_key_in_internal(key)
                internal_nodelist = fnodelist.nodelist[index].leftchild.parent
                internal_nodelist.nodelist[pindex] = fnodelist.nodelist[index+1]
                # 바꿔줄 때 child 조정
                if fnodelist.isleaf: # 리프 노드 일때
                    internal_nodelist.nodelist[pindex].leftchild = fnodelist.nodelist[index].leftchild
                    internal_nodelist.nodelist[pindex].rightchild = fnodelist.nodelist[index].rightchild
                else: # 부모 노드 일때
                    fnodelist.nodelist[index+1].leftchild = fnodelist.nodelist[index].leftchild
            fnodelist.nodelist.pop(index)
        # 삭제하면 최소 키 수를 만족하지 못할 때
        else:
            # 형제에서 빌려온다
            print("형제에서 빌려온다")
            lnodelist, rnodelist, k = fnodelist.find_l_r_i_nodelist(key)
            # 왼쪽 형제가 None이 아니고 빌려줄 수 있을 때
            if lnodelist and len(lnodelist.nodelist) > math.ceil(int(fnodelist.size)/2) - 1:
                if lnodelist.isleaf: # 리프 노드일때
                    print("왼쪽에서 빌려온다")
                    node = lnodelist.nodelist.pop()
                    print("빌릴 노드 키")
                    print(node.key)
                    fnodelist.nodelist.pop(index)
                    fnodelist.nodelist.append(node)
                    fnodelist.nodelist.sort(key = lambda x : x.key)
                    # 부모 노드 리스트 재조정 및 자식 관리
                    node.leftchild = fnodelist.parent.nodelist[k].leftchild
                    node.rightchild = fnodelist.parent.nodelist[k].rightchild
                    fnodelist.parent.nodelist[k].leftchild = None
                    fnodelist.parent.nodelist[k].rightchild = None
                    fnodelist.parent.nodelist[k] = node
                else: # 부모 노드일 때
                    lnode = lnodelist.nodelist.pop()
                    pnode = fnodelist.parent.nodelist.pop()
                    tmp = lnode.rightchild
                    fnodelist.nodelist.append(pnode)
                    fnodelist.parent.nodelist.append(lnode)
                    lnode.leftchild = pnode.leftchild
                    lnode.rightchild = pnode.rigthchild
                    pnode.leftchild = tmp
                    pnode.rightchild = fnodelist.nodelist[0].leftchild
                    fnodelist.nodelist.sort(key = lambda x : x.key)
                    fnodelist.parent.nodelist.sort(key = lambda x : x.key)
```

```
            # 오른쪽 형제가 None이 아니고 빌려줄 수 있을 때
            elif rnodelist and len(rnodelist.nodelist) > math.ceil(int(fnodelist.size)/2) - 1:
                if rnodelist.isleaf: # 리프 노드일때
                    print("오른쪽에서 빌려온다")
                    # 오른쪽 형제에서 첫 번째 노드를 빌려옴
                    node = rnodelist.nodelist.pop(0)
                    print("빌릴 노드 키: " + str(node.key))
                    fnodelist.nodelist.append(node)
                    # 오른쪽 형제의 첫 번째 노드의 자식 노드를 부모 노드와 연결
                    rnodelist.nodelist[0].leftchild = fnodelist.parent.nodelist[k].leftchild
                    rnodelist.nodelist[0].rightchild = fnodelist.parent.nodelist[k].rightchild
                    fnodelist.parent.nodelist[k] = rnodelist.nodelist[0]  # 부모 노드를 빌려온 노드로 교체
                    # Internal 노드가 있을 경우 자식 노드의 참조를 수정
                    if fnodelist.nodelist[index].leftchild:
                        print("Internal 노드가 있다면")

                        # 부모에서 올바른 위치에 해당하는 노드를 찾음
                        pindex = fnodelist.nodelist[index].find_my_key_in_internal(key)
                        internal_nodelist = fnodelist.nodelist[index].leftchild.parent

                        # 자식 노드 재배치 : 빌려온 노드를 부모에 추가
                        fnodelist.nodelist[index+1].leftchild = internal_nodelist.nodelist[pindex].leftchild
                        fnodelist.nodelist[index+1].rightchild = internal_nodelist.nodelist[pindex].rightchild

                        # 부모 노드 갱신
                        internal_nodelist.nodelist[pindex] = fnodelist.nodelist[index+1]
                    # 삭제 후 노드 재배치
                    fnodelist.nodelist.pop(index)
                else: # 부모 노드일 때
                    rnode = rnodelist.nodelist.pop()
                    pnode = fnodelist.parent.nodelist.pop()
                    tmp = rnode.leftchild
                    fnodelist.nodelist.append(pnode)
                    fnodelist.parent.nodelist.append(rnode)
                    rnode.leftchild = pnode.leftchild
                    lnode.rightchild = pnode.rigthchild
                    pnode.rightchild = tmp
                    pnode.lefttchild = fnodelist.nodelist[-1].rightchild
                    fnodelist.parent.nodelist.sort(key = lambda x : x.key)
            # 형제한테 못빌리고 부모한테 도움을 받아야할 때
            # 이때 못하겠어서 리프노드에서 값만 삭제하도록 구현
            # 리프 노드가 최소 키 개수를 만족하지 못함
            else:
                print("부모한테 도움 받기")
                fnodelist.nodelist.pop(index)
```

delete(self, fnodelist, key)

알고리즘

1. 우선 fnodelist에서 key가 삭제될 index를 구한다
2. 삭제해도 최소 키 수를 만족하는 지 판단한다

2-1) 삭제해도 최소 키 수를 만족할 때

원래는 그냥 삭제해도 되지만 internal node가 있다면 자식 연결을 재조정해 준 후 삭제한다 (종료)

2-2) 삭제해도 최소 키 수를 만족하지 못할 때는 3번으로 넘어간다

3-1) 왼쪽 nodelist가 있고 왼쪽 nodelist가 빌려줄 수 있을 때

3-2) 오른쪽 nodelist가 있고 오른쪽 nodelist가 빌려줄 수 있을 때

3-3) 부모에게 도움을 받는다

```python
# 삭제해도 최소 키 수를 만족할 때
if len(fnodelist.nodelist) > math.ceil(int(fnodelist.size)/2) - 1:
    print("삭제해도 최소 키 수를 만족")
    # internal node이면 다음 노드랑 internal 노드랑 교체
    if fnodelist.nodelist[index].leftchild:
        pindex = fnodelist.nodelist[index].find_my_key_in_internal(key)
        internal_nodelist = fnodelist.nodelist[index].leftchild.parent
        internal_nodelist.nodelist[pindex] = fnodelist.nodelist[index+1]
        # 바꿔줄 때 child 조정
        if fnodelist.isleaf: # 리프 노드 일때
            internal_nodelist.nodelist[pindex].leftchild = fnodelist.nodelist[index].leftchild
            internal_nodelist.nodelist[pindex].rightchild = fnodelist.nodelist[index].rightchild
        else: # 부모 노드 일때
            fnodelist.nodelist[index+1].leftchild = fnodelist.nodelist[index].leftchild
    fnodelist.nodelist.pop(index)
```

우선 삭제해도 최소 키 수를 만족할 때는 무조건 삭제할 index 옆에 index+1의 인덱스를 가진 노드가 있을 수 밖에 없다. 그래서 internal node를 그걸로 대체하고 자식 연결을 재조정 해주면 2-1) 경우의 수는 끝이 난다

```python
# 왼쪽 형제가 None이 아니고 빌려줄 수 있을 때
if lnodelist and len(lnodelist.nodelist) > math.ceil(int(fnodelist.size)/2) - 1:
    if lnodelist.isleaf: # 리프 노드일때
        print("왼쪽에서 빌려온다")
        node = lnodelist.nodelist.pop()
        print("빌릴 노드 키")
        print(node.key)
        fnodelist.nodelist.pop(index)
        fnodelist.nodelist.append(node)
        fnodelist.nodelist.sort(key = lambda x : x.key)
        # 부모 노드 리스트 재조정 및 자식 관리
        node.leftchild = fnodelist.parent.nodelist[k].leftchild
        node.rightchild = fnodelist.parent.nodelist[k].rightchild
        fnodelist.parent.nodelist[k].leftchild = None
        fnodelist.parent.nodelist[k].rightchild = None
        fnodelist.parent.nodelist[k] = node
    else: # 부모 노드일 때
        lnode = lnodelist.nodelist.pop()
        pnode = fnodelist.parent.nodelist.pop()
        tmp = lnode.rightchild
        fnodelist.nodelist.append(pnode)
        fnodelist.parent.nodelist.append(lnode)
        lnode.leftchild = pnode.leftchild
        lnode.rightchild = pnode.rigthchild
        pnode.leftchild = tmp
        pnode.rightchild = fnodelist.nodelist[0].leftchild
        fnodelist.nodelist.sort(key = lambda x : x.key)
        fnodelist.parent.nodelist.sort(key = lambda x : x.key)
```

왼쪽 형제가 있을 때는 왼쪽에서 빼온 것을 자기 nodelist에 삽입하고 자식 연결 재조정을 해주면 된다. 밑에 부모 노드일 때에 대한 코드는 이후에 설명

```python
# 오른쪽 형제가 None이 아니고 빌려줄 수 있을 때
elif rnodelist and len(rnodelist.nodelist) > math.ceil(int(fnodelist.size)/2) - 1:
    if rnodelist.isleaf: # 리프 노드일때
        print("오른쪽에서 빌려온다")
        # 오른쪽 형제에서 첫 번째 노드를 빌려옴
        node = rnodelist.nodelist.pop(0)
        print("빌릴 노드 키: " + str(node.key))
        fnodelist.nodelist.append(node)
        # 오른쪽 형제의 첫 번째 노드의 자식 노드를 부모 노드와 연결
        rnodelist.nodelist[0].leftchild = fnodelist.parent.nodelist[k].leftchild
        rnodelist.nodelist[0].rightchild = fnodelist.parent.nodelist[k].rightchild
        fnodelist.parent.nodelist[k] = rnodelist.nodelist[0]  # 부모 노드를 빌려온 노드로 교체

        # Internal 노드가 있을 경우 자식 노드의 참조를 수정
        if fnodelist.nodelist[index].leftchild:
            print("Internal 노드가 있다면")

            # 부모에서 올바른 위치에 해당하는 노드를 찾음
            pindex = fnodelist.nodelist[index].find_my_key_in_internal(key)
            internal_nodelist = fnodelist.nodelist[index].leftchild.parent

            # 자식 노드 재배치 : 빌려온 노드를 부모에 추가
            fnodelist.nodelist[index+1].leftchild = internal_nodelist.nodelist[pindex].leftchild
            fnodelist.nodelist[index+1].rightchild = internal_nodelist.nodelist[pindex].rightchild

            # 부모 노드 갱신
            internal_nodelist.nodelist[pindex] = fnodelist.nodelist[index+1]
        # 삭제 후 노드 재배치
        fnodelist.nodelist.pop(index)
```

```
    else: # 부모 노드일 때
        rnode = rnodelist.nodelist.pop()
        pnode = fnodelist.parent.nodelist.pop()
        tmp = rnode.leftchild
        fnodelist.nodelist.append(pnode)
        fnodelist.parent.nodelist.append(rnode)
        rnode.leftchild = pnode.leftchild
        lnode.rightchild = pnode.rigthchild
        pnode.rightchild = tmp
        pnode.lefttchild = fnodelist.nodelist[-1].rightchild
        fnodelist.parent.nodelist.sort(key = lambda x : x.key)
```

오른쪽 형제가 있고 도움을 줄 수 있을 때는 왼쪽과 비슷하지만 internal 노드가 바로 위 parent에 있는 왼쪽 형제와 달리 따로 처리를 해줘야함.

```
# 형제한테 못빌리고 부모한테 도움을 받아야할 때
# 이때 못하겠어서 리프노드에서 값만 삭제하도록 구현
# 리프 노드가 최소 키 개수를 만족하지 못함
else:
    print("부모한테 도움 받기")
    fnodelist.nodelist.pop(index)
```

마지막으로 형제한테 도움을 받지 못해 부모의 도움이 필요한 경우를 구현해야 했는데 못함. internal node에 대한 삭제를 하기 위해 원래는 merge라는 함수를 만들었다. 우선 아래 사진은 위의 fnodelist.nodelist.pop(index) 한줄 대신에 작성했던 코드 전체이다.

```
480             # 리프 노드가 최소 키 개수를 만족하지 못함
481             else:
482                 print("부모한테 도움 받기")
483                 fnodelist.nodelist.pop(index)
484 
485                 # 트리 구조 세팅을 먼저 하고 merge로 싹싹
486                 print("부모한테 빌려온다 = 노드 합체")
487                 # internal node에 대한 처리 먼저 해주기
488                 if fnodelist.nodelist[index].leftchild:
489                     # internal nodelist 위치 찾기
490                     print("internal node 있음")
491                     pindex = fnodelist.nodelist[index].find_my_key_in_internal(key)
492                     internal_nodelist = fnodelist.nodelist[index].leftchild.parent
493                     # 다음 index가 있다면 거기서 갖고 온다
494                     # 다음 index는 internal node가 없어서 바로 처리해줄 수 있음
495                     if index + 1 < len(fnodelist.nodelist):
496                         fnodelist.nodelist[index+1].leftchild = fnodelist.nodelist[index].leftchild
497                         fnodelist.nodelist[index+1].rightchild = fnodelist.nodelist[index].rightchild
498                         fnodelist.nodelist[index].leftchild = None
499                         fnodelist.nodelist[index].rightchild = None
500                         internal_nodelist.nodelist[pindex] = fnodelist.nodelist[index+1]
```

```
중단점을 추가하려면 클릭합니다.
502                     # 다음 index는 없고 rnodelist가 있으면 그 첫번째 값만 복사해옴
503                     elif rnodelist:
504                         print("rnode에서 갖고옴")
505                         internal_nodelist.nodelist[pindex].key = rnodelist.nodelist[0].key
506                         # internal_node에 다음거랑 같아질 수가 있음 이때 key를 -1로 설정한다
507                         if pindex + 1 < len(internal_nodelist.nodelist):
508                             if internal_nodelist.nodelist[pindex].key == internal_nodelist.nodelist[pindex+1].key:
509                                 print(internal_nodelist.nodelist[pindex].key)
510                                 print("근데 값이 같아서 이걸 -1로 바꿈")
511                                 internal_nodelist.nodelist[pindex].key = -1
512 
513                     # 다 없으면 그냥 없앰 key에 -1로 표현(트리 형태를 유지하기 위해)
514                     else:
515                         print("맨 오른쪽이라 -1 변경")
516                         internal_nodelist.nodelist[pindex].key = -1
517                         for i in internal_nodelist.nodelist:
                            print(i.key)
```

```python
# 리프노드에서 일단 제거 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
if fnodelist.isleaf:
    fnodelist.nodelist.pop(index)
    print("리프 노드에서 일단 제거")

# 맨 왼쪽이나 오른쪽 노드면 부모의 k인덱스의 왼쪽 오른쪽 합치기
if (not lnodelist and rnodelist) or (lnodelist and not rnodelist):
    print("맨 왼쪽이나 오른쪽 노드")
    self.merge(fnodelist, k)
# 나머지는 부모의 k-1인덱스의 왼쪽과 오른쪽 합치기(왼쪽 위주로 합치기)
else:
    print("중간")
    self.merge(fnodelist, k-1)
```

우선 merge 함수에 넘어가기 전에 tree 구조에 대해 먼저 세팅을 하고 이 tree를 merge에 넣으려고 했다. 우선 find_my_key_in_internal 함수로 internal nodelist에 있는 삭제할 키 값의 index를 리턴하도록 구현했다. 그리고 nodelist[index].leftchild.parent는 internal_nodelist를 가리킨다. 여기서 경우의 수를 3가지로 나눈다.

1. 같은 Nodelist에 다음 index가 있는 경우
이땐 다음 인덱스(index+1)은 internal node가 없어서 internal nodelist에 있는 삭제할 key를 index+1노드로 바꿔줄 수 있다.
2. 같은 트리 안에 rnodelist가 있는 경우
이땐 다음 인덱스 대신 rnodelist에 있는 첫 번째 node의 key값을 복사해서 넣는다. 이때 internal node에서 삭제할 키 값이 옆에 있는 rnodelist의 첫 번째 key 값과 같다면 -1을 넣는다. -1을 넣는 경우는 이따 자식에 대한 재조정을 위해 노드는 남겨놓기 위함이다.
3. 작은 트리 안에 맨 오른쪽 node인 경우
이땐 그냥 internal node값을 -1로 변경한다.

이렇게 트리 구조에 대해 세팅을 하고 이 트리를 merge함수로 보내서 정리하려고 했다. 아래는 merge 함수이다. 우선 merge는 왼쪽 - 부모 - 오른쪽 이런 구조의 트리라면 무조건 왼쪽으로 넣는 구조로 만들었다. 그래서 delete함수에서 merge를 호출할 때 부모에서 삭제될 키 값을 기준으로 merge로 넘겼다. 그래서 merge가 호출되면 lnodelist, rnodelist, parent에 대한 값을 기준으로 merge를 하게 되는 것이다.

```python
    # 삭제에서 필요한 합치기 함수
    def merge(self, fnodelist, index):
        # 합치는건 무조건 부모의 값을 왼쪽에 삽입하고 오른쪽 nodelist를 붙인다
        lnodelist = fnodelist.parent.nodelist[index].leftchild
        rnodelist = fnodelist.parent.nodelist[index].rightchild
        parent = fnodelist.parent
        print("부모랑 밑에 왼 오 합칠거임 아래는 부모 키들")
        for i in parent.nodelist:
            print(i.key, end = ' ')
        print(' ')
        for i in lnodelist.nodelist:
            print(i.key, end = ' ')
        print(' ')

        # 합치다가 루트노드를 만났을 때(루트는 최소 키 개수 영향 x)
        if parent == self.head:
            print("부모가 루트일 때")
            node = parent.nodelist.pop(index)
            node.leftchild = lnodelist.nodelist[-1].rightchild
            node.rightchild = rnodelist.nodelist[0].leftchild
            # 왼쪽에 추가
            lnodelist.nodelist.append(node)
            for i in rnodelist.nodelist:
                lnodelist.nodelist.append(i)
            if len(parent.nodelist) > 0: # 루트에 다른 키가 남아있으면 자식 연결만 바꿔줌
                parent.nodelist[index].leftchild = lnodelist
            else: # 루트에 다른 키가 없으면 루트를 바꿔주고 리턴
                self.head = lnodelist
                return

        if lnodelist.isleaf:
            lnodelist.next = rnodelist.next
        # 일단 옮겨
        # -1 이거나 rnode랑 같으면 넣지 않고 합체
        if parent.nodelist[index].key == -1 or parent.nodelist[index].key == rnodelist.nodelist[0].key:
            print("-1 삭제 또는 같은 값 삭제")
            # 리프가 아니면 자식 재조정
            if not lnodelist.isleaf:
                lnodelist.nodelist[-1].rigthchild = rnodelist.nodelist[0].leftchild
            for i in rnodelist.nodelist:
                lnodelist.nodelist.append(i)
                print(str(i.key) + "를 옮김")
            # 선을 하나로 압축
            parent.nodelist[index].rightchild = lnodelist
            if parent.nodelist[index].key == -1:
                self.delete(parent, -1) # -1 삭제
            else:
                self.delete(parent, parent.nodelist[index].key) # 같은 값 삭제
        # 부모 노드에 있는 값 넣고 합체
        else:
            node = parent.nodelist[index]
            # 리프가 아니면 자식 재조정
            if not lnodelist.isleaf:
                if len(lnodelist.nodelist) > 0:
                    node.leftchild = lnodelist.nodelist[-1].rightchild
                if len(rnodelist.nodelist) > 0:
                    node.rightchild = rnodelist.nodelist[0].leftchild
            # 왼쪽에 추가
            lnodelist.nodelist.append(node)
            for i in rnodelist.nodelist:
                lnodelist.nodelist.append(i)
            parent.nodelist[index].leftchild = lnodelist
            if len(parent.nodelist) >= math.ceil(int(fnodelist.size)/2) - 1:
                return
            else:
                glnodelist, grnodelist, gk = parent.find_l_r_i_nodelist(node.key)
                # 맨 왼쪽이나 오른쪽 노드면 부모의 k인덱스의 왼쪽 오른쪽 합치기
                if (not glnodelist and grnodelist) or (glnodelist and not grnodelist):
                    self.merge(fnodelist, gk)
                # 나머지는 부모의 k-1인덱스의 왼쪽과 오른쪽 합치기(왼쪽 위주로 합치기)
                else:
                    self.merge(fnodelist, gk-1)
```

우선 부모 노드가 key를 빌려주거나 빌려주지 않는 경우의 수 2가지로 나눴다. 아까 넣은 -1은 삭제된거고 key값이 rnodelist의 첫 번째와 같다면 이것 또한 한 Nodelist에 같은 값이 있는거라 -1과 같은 취급을 할 수 있게 된다. 이 경우에 합칠 때 왼쪽 Nodelist에 부모의 Node가 들어가지 않는다. 그리고 부모에서 삭제할 key를 delete에 다시 넣어 delete - merge - delete - merge ... 이런 구조를 생각했다. 다음으로 -1 취급이 없는 경우 왼쪽 Nodelist에 부모의 Node가 들어갈 수 있다. 이때 부모의 Nodelist 길이가 최소 길이보다 길다면 return하고 아니라면 다시 merge를 재귀호출하여 좁혀가는 방식이다. 마지막으로 root에 대한 처리로 root는 최소 키 개수에 영향을 받지 않는다. 그래서 그냥 합쳐주고 self.head를 바꿔준 후 return 해준다. 정상 작동 됐다면 여기 root 처리에서 재귀가 멈췄을 것이다.

대체 방법

internal node에 대한 처리를 하지 않고 그냥 leaf node에 있는 값만 삭제했다. 이 경우 leaf node도 최소 키 개수에 영향 받지 않도록 구현했다.

출력 결과

```
1  node_size : 3
2  68  |
3  26  | 86  |
4  10 15  | 37 42  | 75  | 87  |
5  9  | 10  | 15 20  | 26  | 37  | 42 52  | 68  | 75 84  | 86  | 87 99  |
```

현재 아무것도 삭제하지 않고 input만 한 상태이다

여기서 최소 키 수를 만족하는 15, 42, 75를 삭제하면

```
1  node_size : 3
2
3  68  |
4  26  | 86  |
5  10 20  | 37 52  | 84  | 87  |
6  9  | 10  | 20  | 26  | 37  | 52  | 68  | 84  | 86  | 87 99  |
7  |
```

internal node까지 완벽하게 없어지는 모습이다.

하지만 37을 삭제하면

```
1  node_size : 3
2
3  68  |
4  26  | 86  |
5  10 20  | 37 52  | 84  | 87  |
6  9  | 10  | 20  | 26  |  | 52  | 68  | 84  | 86  | 87 99  |
7  |
```

internal node가 없어지지 않고 leaf노드도 없어지지 않는다
여기서 40, 42, 44를 넣는다면

```
node_size : 3

68  |
26 42  | 86  |
10 20  | 37  | 52  | 84  | 87  |
9  | 10  | 20  | 26  | 40  | 42 44  | 52  | 68  | 84  | 86  | 87 99  |
```

어찌저찌 tree 구조는 유지 되는 것 같다.
모든 경우에서 이렇게 유지되는 것이 아니고 특수한 경우이기에
삭제는 최소 키 개수를 유지할 수 있을 때, 왼쪽, 오른쪽에서 빌려올 수 있을 때만 정상적으로 tree구조를 유지할 수 있게 된다.

다른 testcase
node_size : 7
key, value : 1000개 넣기

```
node_size : 7
21878  |
5223 10717 13666 17237  | 25536 29420 34489 39049 41969 46590  |
867 1443 2112 2425 3205 4069  | 6242 7050 7729 8210 9011 9711  | 11396 12287 12946  | 14686 15261 16051  | 17657 18089 18919 20119 2076
188 354 419 537 724  | 1092 1195 1276 1373  | 1509 1553 1766 1830 1898 2002  | 2204 2297 2348  | 2454 2602 2712 2767 2877 3022  | 3423
33 90 113  | 188 208 209 348  | 354 359 362 368 402  | 419 424 464 512 531  | 537 699 722  | 724 786 810 860  | 867 871 882 935 966 104

PS C:\Users\dlfpf\bptree> python b+tree.py -s index.dat 1564
21878
5223 10717 13666 17237
867 1443 2112 2425 3205 4069
1509 1553 1766 1830 1898 2002
leaf노드 도달
found : 1564
PS C:\Users\dlfpf\bptree>

PS C:\Users\dlfpf\bptree> python b+tree.py -s index.dat 1565
21878
5223 10717 13666 17237
867 1443 2112 2425 3205 4069
1509 1553 1766 1830 1898 2002
leaf노드 도달
Not found
```

```
PS C:\Users\dlfpf\bptree> python b+tree.py -r index.dat 400 1000
leaf노드 도달
402 402
419 419
424 424
464 464
512 512
531 531
537 537
699 699
722 722
724 724
786 786
810 810
860 860
867 867
871 871
882 882
935 935
966 966
PS C:\Users\dlfpf\bptree>
```

(key와 value값을 같게 넣었습니다)

삭제가 되는 경우 골라서 삭제

```
7  | 39771 40461 41055  | 42674 43202 43862
 | 7251 7297 7563  | 7829 7921 8172  | 8356
1514 1533  | 1553 1564 1573 1599 1762  | 17
```

```
7563  | 7829 7921 8172  | 8356 845
| 1553 1564 1573 1599  | 1766 1787
```

```
PS C:\Users\dlfpf\bptree> python b+tree.py -d index.dat delete.csv
삭제1762
1762삭제
leaf 노드에서 인덱스4
삭제해도 최소 키 수를 만족
```