과제

장유선

2023.08.23

1. 문제 정의

Node에 값을 4개까지 지닐 수 있고 자식 노드는 2개를 지니는 B-Tree를 만들어 봅시다.

중복된 값을 입력하는 경우는 가정하지 않습니다.

- Insert

1. Root Node부터 시작합니다.

2. 현재 노드가 자식 노드를 가지고 있지 않은 경우 현재 노드에 값을 추가합니다.

    a. 값이 추가 되었을 때 Node에 값이 4개 이하인 경우 해당 값을 저장합니다.

    b. 값이 추가 되었을 때 Node에 값이 5개가 된 경우 Node 내의 값을 정렬하여 중심 값을 기준값으로 설정하고 자식 노드를 생성하며 좌측에는 기준값보다 작은 값, 우측에는 기준값보다 큰 값을 입력합니다.

3. 현재 노드가 자식 노드를 지니고 있는 경우, 현재 노드의 기준값과 입력값을 비교하여 기준값보다 작은 경우 좌측 노드를, 큰 경우 우측 노드를 현재 노드로 선택하고 2.로 돌아갑니다.

- Get

1. Root Node부터 시작합니다

2. 현재 노드가 자식 노드를 지니고 있지 않은 경우

    a. 현재 노드에 입력 값을 지니고 있는지 여부를 리턴합니다.

3. 현재 노드가 자식 노드를 지니고 있는 경우

    a. 현재 노드의 기준값이 입력 값과 동일 한 경우에는 값이 존재한다고 리턴

    b. 현재 노드의 기준값과 입력 값이 다른 경우

        i. 입력 값이 기준 값보다 작은 경우 좌측 노드를 현재 노드로 선택하고 2.로 돌아갑니다.

        ii. 입력 값이 기준 값보다 큰 경우 우측 노드를 현재 노드로 선택하고 2.로 돌아갑니다.

테스트 코드

1. 0 ~ 1,000,000 사이의 임의의 값 100,000개를 중복 없이 추출합니다.

2. 0 ~ 1,000,000 사이의 임의의 값 10,000개를 중복 없이 추출합니다.

3. List에 1.에서 추출한 값을 삽입합니다.

4. B-Tree에 1.에서 추출한 값을 삽입합니다.

5. List의 in 연산자를 이용하여 2.에서 추출한 값이 3.에서 만든 List에 존재하는지 판단하고, 해당 과정이 소요 된 시간을 측정합니다.

6. B-Tree에서 2.에서 추출한 값이 3.에서 만든 List에 존재하는지 판단하고, 해당 과정이 소요 된 시간을 측정합니다.

Example

List 내에 특정 값이 존재하는 지 확인하는 방법

```
l = [1, 3, 5, 7, 9]
if 3 in l:
  print('3이 존재합니다.')
```

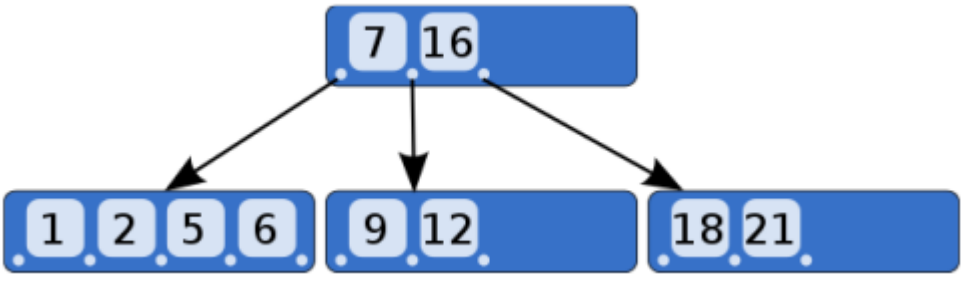
2. B-Tree

- 하나의 노드에 여러자료가 배치되는 트리구조

- 한 노드에 M개의 자료가 배치되면 M차 B-Tree

- 노드의 자료수가 N이라면, 자식의 수는 N+1이어야 한다.

- 각 노드의 자료는 정렬된 상태여야 한다.



3. Python Code Hard Copy

```
# 노드
class BTreeNode:
    def __init__(self):
        self.values = [] # 노드에 저장된 값
        self.children = [] # 자식 노드

# 각 레벨의 노드와 해당 노드의 값을 출력
def print_tree(node, level=0):
    if node is not None:
        print(f"Level {level}: {node.values}")
        for child in node.children:
            print_tree(child, level + 1)

# B-Tree
class BTree:
    def __init__(self):
        self.root = BTreeNode() # 루트 노드

    def insert(self, value): # 값 삽입
        self.insert_value(self.root, value)

    def insert_value(self, node, value):
        # 현재 노드의 children 리스트가 비어있는지를 확인
        if not node.children:  # 현재 노드가 리프 노드인 경우 -> 값 삽입
            node.values.append(value) # 삽입
            node.values.sort() # 정렬

            if len(node.values) > 4:  # 노드가 가득 찼을 경우 -> 분할
                left_node = BTreeNode() # 좌측 자식 노드
                right_node = BTreeNode() # 우측 자식 노드
                left_node.values = node.values[:2] # 중간값을 기준으로 분할
                right_node.values = node.values[3:]
                node.values = [node.values[2]] # 중간값으로 값 갱신
                node.children = [left_node, right_node] # 자식 노드

        else: # 현재 노드가 부모 노드인 경우 자식 노드로 이동
            i = 0
            while i < len(node.values): # value 와 현재 노드의 values 리스트를
비교
                # value 가 현재 노드의 values[i] 값보다 작다면,
                # 삽입할 value 는 현재 노드의 values[i] 값보다 왼쪽에 있어야 함을
의미
                # 이 경우, while 루프를 탈출하고 i 는 삽입할 value 가 들어가야
하는 인덱스가 된다.
                if value < node.values[i]:
```

```
                    break
                i += 1
            self.insert_value(node.children[i], value) # value를 자식 노드 중 i 인덱스에 해당하는 자식 노드로 이동

    def get(self, value):
        return self.get_value(self.root, value)

    def get_value(self, node, value):
        if not node.children:  # 현재 노드가 리프 노드인 경우
            return value in node.values
        else:  # 현재 노드가 부모 노드인 경우 자식 노드로 이동
            i = 0
            while i < len(node.values):
                if value == node.values[i]:
                    return True
                if value < node.values[i]:
                    return self.get_value(node.children[i], value)
                i += 1
            return self.get_value(node.children[i], value)

# B-Tree 생성
btree = BTree()

# 값 삽입
values_insert = [5, 2, 11, 6, 7, 8, 3, 10, 15]
print("[5, 2, 11, 6, 7, 8, 3, 10, 15]를 삽입합니다.")
for value in values_insert:
    btree.insert(value)

# 값을 검색
values_find = [3, 10, 6]
print("[3, 10, 6]값이 있는지 확인합니다.")
for value in values_find:
    found = btree.get(value)
    if found:
        print(f"{value}를 찾았습니다.")
    else:
        print(f"{value}를 찾지 못했습니다.")
print("\nB-Tree 노드와 레벨을 확인합니다.")
print_tree(btree.root)


import random
import time

# 1. 0 ~ 1,000,000 범위에서 중복 없는 100,000개의 임의의 값 추출
values_insert = random.sample(range(1000001), 100000)
```

```
# 2. 0 ~ 1,000,000 범위에서 중복 없는 10,000 개의 임의의 값 추출
values_find = random.sample(range(1000001), 10000)

# 3. List 에 값을 삽입
start_time = time.time()
l = []
for value in values_insert:
    l.append(value)
list_insert_time = time.time() - start_time

# 4. B-Tree 에 값을 삽입
start_time = time.time()
btree = BTree()
for value in values_insert:
    btree.insert(value)
btree_insert_time = time.time() - start_time

# 5. List 에서 값의 존재 확인 및 시간 측정
start_time = time.time()
for value in values_find:
    found = value in l
    '''
    if found:
        print(f"{value}를 찾았습니다.")
    else:
        print(f"{value}를 찾지 못했습니다.")
    '''
list_find_time = time.time() - start_time

# 6. B-Tree 에서 값의 존재 확인 및 시간 측정
start_time = time.time()
for value in values_find:
    found = btree.get(value)
btree_find_time = time.time() - start_time

print("\nList 와 B-Tree 의 소요시간을 출력합니다.")
print(f"List 소요 시간: {abs(list_find_time - list_insert_time)} 초")
print(f"B-Tree 소요 시간: {abs(btree_find_time - btree_insert_time)} 초")
```

4. Code 설명

4-1. BTreeNode 클래스

```
class BTreeNode:
    def __init__(self):
        self.values = []
```

```
        self.children = []
```

- B-Tree의 노드를 정의한다.
- Values는 현재 노드에 저장된 값을 담는 리스트
- Children은 현재 노드의 자식 노드를 나타내는 리스트

4-2. BTree 클래스

```
class BTree:
    def __init__(self):
        self.root = BTreeNode()

    def insert(self, value):
        self.insert_value(self.root, value)
```

- BTree 클래스는 B-Tree 자체를 정의
- self.root 속성은 B-Tree의 루트 노드를 나타낸다.
- insert 메서드는 값을 삽입하는 메서드로, 루트 노드부터 시작하여 값을 삽입

4-3. insert_value 메서드 (현재 노드에 자식노드가 존재하지 않는 경우)

```
    def insert_value(self, node, value):
        if not node.children:
            node.values.append(value)
            node.values.sort()

            if len(node.values) > 4:
                left_node = BTreeNode()
                right_node = BTreeNode()
                left_node.values = node.values[:2]
                right_node.values = node.values[3:]
                node.values = [node.values[2]]
                node.children = [left_node, right_node]
```

- 현재 노드의 children 리스트가 비어있는지를 확인한 후 현재 노드가 leaf 노드인 경우 값을 삽입한다.
- 노드가 가득 찼을 경우 (5개 이상일 경우) 분할한다.
- 중간값을 기준으로 좌측, 우측 노드를 생성한다.

4-4. insert_value 메서드 (현재 노드에 자식노드가 존재하는 경우)

```
        else:
            i = 0
            while i < len(node.values):
                if value < node.values[i]:
```

```
                    break
                i += 1
            self.insert_value(node.children[i], value)
```

- value와 현재 노드의 values 리스트를 비교
- value가 현재 노드의 values[i] 값보다 작다면, 삽입할 value는 현재 노드의 values[i] 값보다 왼쪽에 있어야 함을 의미
- while 루프를 탈출하고 i는 삽입할 value가 들어가야 하는 인덱스가 된다.
- Value를 자식 노드 중 i 인덱스에 해당하는 자식 노드로 이동한다.

4-5. get 메서드와 get_value 메서드

```
def get(self, value):
        return self.get_value(self.root, value)

    def get_value(self, node, value):
        if not node.children:
            return value in node.values
        else:
            i = 0
            while i < len(node.values):
                if value == node.values[i]:
                    return True
                if value < node.values[i]:
                    return self.get_value(node.children[i], value)
                i += 1
            return self.get_value(node.children[i], value)
```

- get 메서드는 값을 검색하는 메서드로, 루트 노드부터 시작하여 값을 검색
- get_value 메서드는 현재 노드와 자식 노드를 탐색하여 값을 검색
- 현재 노드가 리프 노드인 경우에만 값을 직접 확인하고, 그 외에는 자식 노드로 이동하여 값을 검색

4-6. print_tree 함수

```
def print_tree(node, level=0):
    if node is not None:
        print(f"Level {level}: {node.values}")
        for child in node.children:
            print_tree(child, level + 1)
```

- print_tree 함수는 B-Tree의 노드를 레벨별로 출력하는 함수
- node는 현재 노드
- level은 현재 노드의 레벨

- 노드와 레벨을 출력

4-7. B-Tree 코드 확인용 예제

```
btree = BTree()
values_to_insert = [5, 2, 11, 6, 7, 8, 3, 10, 15]
for value in values_to_insert:
    btree.insert(value)


values_to_check = [3, 10, 6]
for value in values_to_check:
    found = btree.get(value)
    if found:
        print(f"{value}를 찾았습니다.")
    else:
        print(f"{value}를 찾지 못했습니다.")

print_tree(btree.root)
```

- B-Tree를 생성하고 값을 삽입한 후, 값을 검색하고 모든 노드를 출력

4-8. 리스트와 B-Tree 삽입, 검색 속도 비교

```
import random
import time

values_insert = random.sample(range(1000001), 100000)
values_find = random.sample(range(1000001), 10000)

start_time = time.time()
l = []
for value in values_insert:
    l.append(value)
list_insert_time = time.time() - start_time

start_time = time.time()
btree = BTree()
for value in values_insert:
    btree.insert(value)
btree_insert_time = time.time() - start_time

start_time = time.time()
for value in values_find:
    found = value in l
    '''
    if found:
        print(f"{value}를 찾았습니다.")
    else:
```

```
        print(f"{value}를 찾지 못했습니다.")
    '''
list_find_time = time.time() - start_time

start_time = time.time()
for value in values_find:
    found = btree.get(value)
btree_find_time = time.time() - start_time

print(f"List 소요 시간: {abs(list_find_time - list_insert_time)} 초")
print(f"B-Tree 소요 시간: {abs(btree_find_time - btree_insert_time)} 초")
```

- random 모듈을 사용하여 임의의 값을 저장한다.
- Insert_time을 초기화하기 위해 각각의 데이터 구조에 값을 삽입하기 전에 현재 시간을 기록합니다.
- B-Tree에서 값을 검색하고 found 변수에 저장
- 각각의 작업을 수행하는 데 걸린 시간을 계산합니다.
- 리스트와 B-Tree 간에 값을 삽입하고 검색하는 데 걸린 시간 차이를 출력합니다.
- abs() 함수를 사용하여 절대값으로 출력한다.

5. 결과

```
[5, 2, 11, 6, 7, 8, 3, 10, 15]를 삽입합니다.
[3, 10, 6]값이 있는지 확인합니다.
3를 찾았습니다.
10를 찾았습니다.
6를 찾았습니다.

B-Tree 노드와 레벨을 확인합니다.
Level 0: [6]
Level 1: [2, 3, 5]
Level 1: [10]
Level 2: [7, 8]
Level 2: [11, 15]

List와 B-Tree의 소요시간을 출력합니다.
List 소요 시간: 58.519429206848145 초
```

B-Tree 소요 시간: 2.8104729652404785 초

6. 결과 화면