2-3, 2-3-4 트리 구현 및 실험

1. 서론
2-3, 2-3-4 트리를 구현하고 비교 실험을 진행하였다.

2. 구현
2-3, 2-3-4 트리는 각각 order 가 3, 4 인 B-tree 이므로, 일반적인 B-tree 를 구현하였다. 홀수 order 를 지원하기 위해, insert 와 delete 에서 split/merge 는 non-preemtive 방식으로 구현하였다.

2.1. btree.py

```
구현에서 사용할 유틸리티 함수
#-------------------------------------------------------------
def is_empty(coll): 입력받은 collection(list, tuple, ..)이 비어있으면 True 반환.
def index(a, x): list나 tuple a에서 x가 있는 인덱스를 반환한다. 없으면 None 반환.
def tup_insert(tup, idx, *val): tup의 idx에 *val을 삽입한 새 tuple을 반환하다.
def tup_update(tup, idx, *val): tup의 idx를 *val로 변경한 새 tuple을 반환하다.
def tup_omit(tup, idx): tup에서 idx 위치의 원소를 제거한 새 튜플을 반환한다.

Btree 정의와 생성
#-------------------------------------------------------------------
Node = namedtuple('Node', 'is_leaf keys children', defaults=[()])
불변타입인 namedtuple을 이용하여 Btree의 노드를 정의하였다. 빈 트리는 None으로 정의한다.
def leaf(*keys): 빈 leaf 노드를 반환한다.
def btree(max_n, *keys): *keys를 모두 포함하는 b-tree를 반환한다. max_n은 b-tree 노드가
가질 수 있는 최대 key의 수로, 2이면 2-3트리, 3이면 2-3-4트리가 된다.

find 구현
#-------------------------------------------------------------
def find(tree, key):
    if key in tree.keys:
        return tree
    else:
        idx = bisect(tree.keys, key)
        if is_empty(tree.children):
            return None
        else:
            return find(tree.children[idx], key)

트리를 재귀적으로 순회하여 노드의 keys에 key가 있다면 그 노드를 반환한다. 트리에 key가
존재하지 않으면 None을 반환한다.

insert 구현
#-------------------------------------------------------------
def insert(tree, key, max_n):
    if tree is None:
        return leaf(key)
    idx = bisect(tree.keys, key) # 이진 탐색으로 key가 삽입될 자식을 찾는다.
    if tree.is_leaf:
        leaf에 key를 넣고 반환한다. 만일 key 수가 max_n 이상이면 split 한다.
    else:
        old_child = tree.children[idx]
        child = insert(tree.children[idx], key, max_n)
```

```
        insert를 재귀 호출하여 key가 삽입된 자식을 구한다.
        if not has_split_child:
        child가 split되지 않았으면 현재 노드를 child로 업데이트하고 반환(트리 구성)한다.
        else:
        child split되었다면, split된 노드의 key값을 현재 노드(tree)에 merge하고,
        split된 노드의 자식을 현재 노드(tree)에 붙여서 merged를 생성한다.
        merged의 key 수가 max_n 이상이면 split 하고 반환한다.

up_idx = 1 # (2-3, 2-3-4 just same as 1)
def split_node(node, max_n):
max_n 개의 key를 가진 노드를 3개의 노드로 나눈다.
2-3 트리의 경우 | 2-3-4 트리의 경우
a-b-c            a-b-c-d
=>  b            =>  b
   a c              a c-d

delete 구현
#------------------------------------------------------------
def delete(tree, key):
    tree가 비었을 경우 경우 빈 트리를 반환한다.

    key가 tree의 어느 노드에 있는지 확인한다.
    nodes, path, founds = get_path(tree, key)
    if not any(founds):
        return tree
        key가 tree에 존재하지 않으면 tree를 그대로 반환한다.
    if founds[-1] is False: # not leaf
        key가 leaf가 아닌 중간 노드인 n에 존재한다면, n에서 key를 삭제한다.
        삭제한 key 다음에 오는 자식 subtree에서 가장 작은 key인 mv_key로
        삭제된 key를 대체한다.
        mv_key가 있던 노드에서는 mv_key를 삭제한다.

        변경된 tree와 n을 이용하여 _delete를 수행한다.
        위 과정에서 만일 root가 변경될 경우, _delete는 key 대신 mv_key로 호출한다.
        key = mv_key

    ret = _delete(tree, key)
    반환된 트리의 ret.keys가 비어 있다면 빈 루트를 제거한다.
    빈 루트가 자식이 없다면 빈 트리이므로 None을 반환한다.
    트리가 비어있지 않으면 ret를 그대로 반환한다

def _delete(node, key): key 삭제를 위한 재귀 함수이다. 트리에 존재하는 key에 대해서만 정의.
    if node.is_leaf:
        leaf 노드라면 key를 제거한다.
        제거된 keys로 현재 노드를 업데이트하여 반환한다.
    else:
        leaf가 아니라면 이진 탐색으로 key가 존재하는 자식을 찾는다.
        _delete를 재귀호출하여 자식의 목록 children을 생성한다.
        idx = bisect(node.keys, key)
        children = tup_update(
            node.children, idx, _delete(node.children[idx], key))

        자식 목록에서 빈 노드(최소 수 이하의 key를 가진 노드)가 없다면
        현재 노드를 children으로 업데이트하여 반환한다.
        empty_idx = empty_node_idx(children)
        if empty_idx is None: # no steal, no merge
            return node._replace(children = children)

        자식 중에 빈 노드가 있다면, theft_victim함수를 이용하여
```

```
        merge를 할지 steal을 할지 결정한다.
        empty_node = children[empty_idx]
        new_keys = node.keys

        sib_idxes = sibling_idxes(children, empty_idx)
        victim, victim_idx, up_key, stolen_child = theft_victim(
            children, sib_idxes, empty_idx)

        if victim: # steal
            steal이 가능하면 한다.
            victim에서 key를 끌어와 현재 노드의 keys를 바꾸고,
            현재 노드의 keys에서 key를 끌어내려 빈 노드를 업데이트한다.
            현재 노드에서 key가 빠진 keys를 new_keys로 선언한다.
            업데이트된 빈 노드와 victim으로 children을 업데이트한다.
            # build new keys
            new_keys = tup_update(new_keys, key_idx, up_key)
            # build new children
            children = tuple(child_lst)
        else: # merge
            steal이 불가능하면 merge를 한다.
            현재 노드에서 key를 꺼내 빈 노드를 채우고, 빈 노드와 sibling을 병합한다.
            현재 노드에서 key가 빠진 keys를 new_keys로 선언한다.
            병합된 노드를 이용하여 children을 업데이트한다.
            # build new keys
            new_keys = tup_omit(new_keys, key_idx)
            # build new children
            children = tup_update(children, empty_idx, new_empty_node)
            children = tup_omit(children, sib_idx)

        생성된 new_keys와 children으로 현재 노드를 업데이트하여 반환한다.
        return node._replace(
            keys = new_keys,
            children = children
        )

def get_path(tree, key): # Get path root to leaf
key가 있는 노드로 향하는 path를 반환한다. 다음 3가지를 튜플로 반환한다.
nodes: path에 존재하는 노드를 트리의 깊이에 따라 반환. path의 정점에 해당한다.
idxes: 각 노드의 부모 노드.children에서의 인덱스. path의 간선에 해당한다. nodes보다 길이가
1 작다.
founds: key가 존재하는 경우 True, 그렇지 않은 경우 False를 반환. nodes와 길이가 같다.
delete에서 호출됨

def update(tree, idxes, new_node):
tree를 받아 new_node로 업데이트한다. idxes가 정의하는 path를 따라 트리를 순회하다가, idxes
가 끝나는 노드를 new_node로 대체하여 트리를 업데이트한다. delete에서 호출됨

def empty_node_idx(children):
자식 노드 목록에서 keys의 길이가 0인 빈 노드의 인덱스를 반환한다. _delete에서 호출됨.

def sibling_idxes(children, idx):
자식 노드 목록에서 idx에 위치하는 노드 n의 sibling을 반환한다. _delete에서 호출됨.

def theft_victim(children, sibling_idxes, target_idx):
자식 노드 목록, sibling_idxes에서 생성한 노드 목록, victim을 찾는 target노드의 자식
목록에서의 인덱스 target_idx를 이용하여 steal에서 사용할 victim(key를 뺏기는 노드), victim
의 인덱스, victim에서 부모로 올라가는 key, victim이 뺏긴 자식 노드를 반환한다. _delete에서
호출된다.
```

테스트와 디버깅을 위한 함수
```
#--------------------------------------------------------------
```
```
def all_keys(root):
```
dfs로 트리를 순회하여 트리에 존재하는 모든 key의 리스트를 반환한다. 올바른 B-tree라면 key는 오름차순으로 정렬되어 있어야 한다.

```
def all_nodes(root, nodes=()):
```
dfs로 트리를 순회하여 트리에 존재하는 모든 노드의 리스트를 반환한다.

```
def intersect_seq(children, keys):
```
두 리스트를 받아 서로 교차하는 리스트를 반환한다. [a,b,c], [1,2] => [a,1,b,2,c].
all_keys에서 호출됨.

```
def is_invalid(node, max_n, min_n=1):
```
node가 올바른 B-tree 노드인지 판단한다. 올바른 노드라면 빈 문자열 ''을 반환한다. node가 invalid할 경우 노드가 B-tree 노드가 아닌 이유를 문자열로 반환한다. valid node 조건으로는 다음 6가지를 확인한다:
- key 수는 max_n 이하여야 한다.
- key 수는 min_n 이상이어야 한다.
- 자식 수는 key 수보다 1 커야 한다.
- leaf 노드라면 자식은 없어야 한다.
- internal 노드라면 자식이 있어야 한다.
- keys가 오름차순으로 정렬되어 있어야 한다.

assert_valid 에서 호출됨.

```
def assert_valid(tree, max_n, input_keys):
```
tree가 올바른 B-tree인지 assert한다. invalid B-tree의 경우 AssertionError가 발생한다.
B-tree의 valid 조건으로는 다음 4가지를 확인한다:
- 트리에 key가 max_n보다 많다면, 루트 노드는 leaf가 아니어야 한다.
- all_keys(tree)의 결과는 오름차순으로 정렬된 input_keys와 같다.
- 트리에 존재하는 모든 노드는 올바른 B-tree 노드여야 한다.
- 모든 leaf 노드의 깊이는 동일해야 한다(children에 leaf가 있다면 모든 자식이 leaf여야 한다).

2.2. unit test: btree_test.py

B-tree 구현을 위해 다양한 경우를 테스트하였다.

```
def test_insert_no_split():
def test_insert_split():
def test_insert_explicit_sequence():
```
insert를 위한 유닛 테스트

```
def test_sibling_idxes(xs_idx):
def test_delete_explicit_sequence():
```
delete를 위한 유닛 테스트

```
def test_tuple_update(tup_idx_new):
def test_tup_omit(tup_idx_new):
def test_btree(max_n, keys):
```
보조 함수를 위한 유닛 테스트

```
def test_find_prop_test(keys_included_excluded_max_n):
def test_insert_prop_test(keys):
def test_delete_prop_test(keys_shuffled):
```
B-tree 알고리즘은 매우 복잡하기 때문에, 정확한 구현을 위해서 property-based test를 적용하였다. keys를 무작위로 생성한 후, insert, find, delete를 수행한다. 각 단계마다 이전에 정의했던 함수 assert_valid를 호출하여 트리가 B-tree인지 체크하였다.

3. 실험
3.1. 실험 구성

구현한 B-tree 를 이용하여 2-3 트리와 2-3-4 트리 구현할 수 있다. 2-3 트리와 2-3-4 트리를 생성, key 탐색/삭제를 동일한 key 시퀀스에대해서 수행하고 수행 시간을 측정하여 비교했다. 데이터는 1000*N 까지의 정수를 데이터를 무작위로 섞어서 삽입하고, 이렇게 삽입한 정수를 다시 무작위 순서로 탐색하고 삭제하였다.

N = 1~100 로 실험을 진행하였으며 탐색, 삭제의 경우 50%의 정수는 B-tree 에 존재하지 않는 정수를 찾거나 제거하였다.

4. 실험 결과

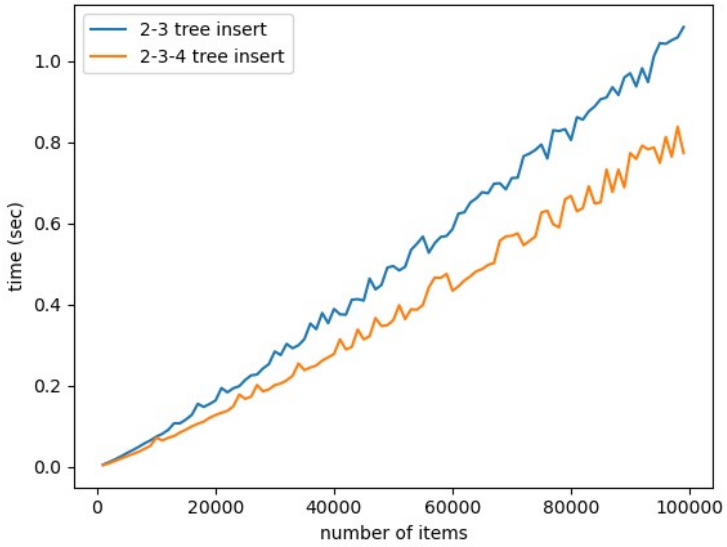

그림 1. 데이터에 대한 2-3 트리, 2-3-4 트리의 insert 수행 시간

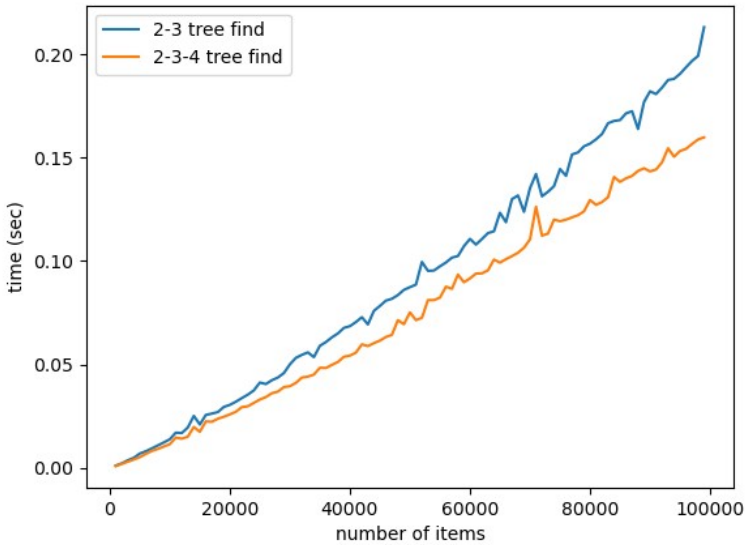

그림 2. 데이터에 대한 2-3 트리, 2-3-4 트리의 find 수행 시간

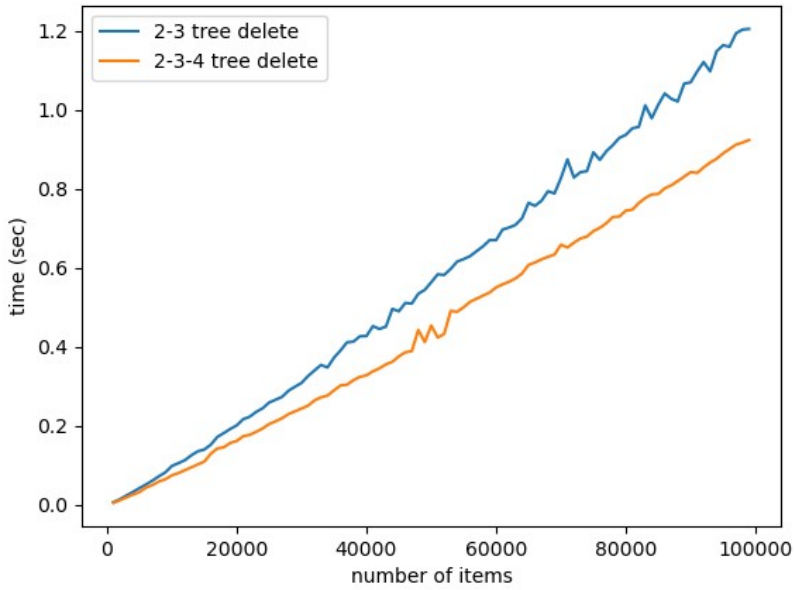

그림 3. 데이터에 대한 2-3 트리, 2-3-4 트리의 delete 수행 시간

실험 결과 2-3-4 트리가 2-3 트리에 비해 더 나은 수행 시간을 보이는 것을 확인할 수 있었다. 이는 2-3-4 트리의 경우 노드에 더 많은 수의 key 를 저장할 수 있어 트리의 높이가 2-3 트리보다 작기 때문이다.

5. 결론

이번 과제에서는 2-3 트리와 2-3-4 트리를 구현하고 insert, find, delete 성능을 비교하였다. 2-3-4 트리가 2-3 트리보다 더 나은 수행 시간 성능을 보이는 것을 확인할 수 있었다.