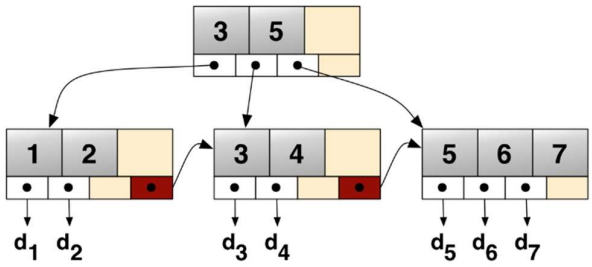
Homework4 보고서

Disk-based B+ Tree Insertion / Deletion

과목명	데이터베이스시스템및응용
담당 교수님	차재혁 교수
제출일	2021년 11월 15일(월요일)
소속	한양대학교 공과대학 컴퓨터소프트웨어학부
학번	**이름**
2019009261	최가온(CHOI GA ON)

Ⅰ. Overview

수업 시간에 Disk에 접근하는 I/O는 cost가 메모리에 접근할 때에 비해 몇 백 배, 많게는 몇 천배의 차이가 난다는 것을 배웠다. 따라서, Disk I/O를 줄이면서도 효율적으로 데이터를 읽고 쓸 수 있는 자료구조의 필요성이 대두되었다. 이러한 요구사항을 만족할 수 있는 자료구조가 바로 B+ Tree이다.



Order = M인 B+ Tree에 대해 아래의 명제가 성립한다.

1) Root Node가 Leaf Node인 경우를 제외하면 항상 최소 2개의 children을 가져야 한다.

2) Root Node와 Leaf Node를 제외한 모든 노드들은 최소 ceil(M/2)개, 최대 M개의 노드들을 가진다.

3) 모든 Leaf Node는 같은 level에 존재한다.

   이에 따라, 어떤 element에 접근하든 depth가 같기 때문에 같은 Disk I/O가 소모된다는 특징이 있다. B Tree와 구별되는 특성 중 하나이다.

4) Leaf Node는 최소 ceil(M/2)-1개의 key를 가지고 있어야 한다.

   (용어 정리: 하나의 Node에는 다수의 key가 존재하는 구조이다.)

이번 과제에서는 Deletion 알고리즘을 이해하고, 그것을 바탕으로 설계하는 과정을 거쳤다. B+ Tree 에서의 Deletion 과정은 아래와 같다.

Case 1)

가장 간단한 경우라고 할 수 있다. 지우고자 하는 키가 오직 LeafNode 에만 있는 경우이다. 즉, InternalNode 에 없는 경우를 의미하는 것이다.

1) 노드에 minimum number 보다 많은 개수의 키가 있는 경우

간단하게 지우고자 하는 해당 키만 지우고 종료하면 된다.

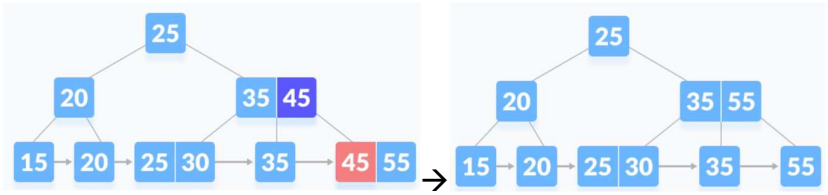
2) 현재 노드에 정확하게 minimum number 만큼 키가 존재하는 경우

옆의 노드(sibling)에서 키를 가져온다. (Redistribute 함수에 해당한다.) 이때, 이번 프로젝트에서는 오른쪽→왼쪽의 방향으로 가져오는 것을 원칙으로 하였다. 오른쪽에서 가장 작은 키를 하나 골라서 왼쪽으로 가져오는 Policy 를 선택하였다. 다만, 예외적으로 해당 키가 최우측에 존재하는 경우에 한해서, 그리고 왼쪽 노드에 키가 충분히 있다면 그때에만 왼쪽 노드에서 키를 빌려오는 것으로 결정하였다.

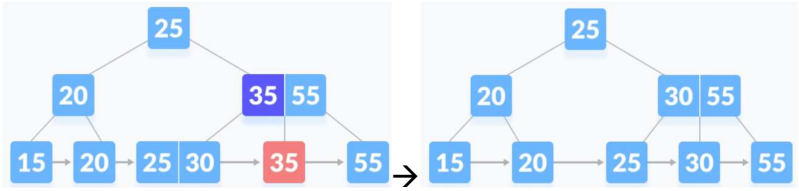
Case 2)

지우고자 하는 Key 가 LeafNode 뿐만 아니라 InternalNode 에도 존재하는 경우에 해당한다. 당연히 Key 가 삭제되는 과정에서 InternalNode 의 일부도 수정되어야 한다.

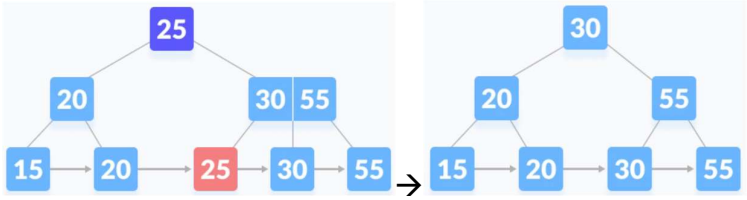
1) 해당 노드에 minimum number 보다 많은 키가 존재하는 경우, 간단하게 해당 키를 지운다. 이후에 InternalNode 에서 해당 키를 삭제한 후 그 비어있는 자리를 inorder successor 가 차지한다. (즉, 지운 키 바로 오른쪽에 있는 키가 InternalNode 로 들어간다고 보면 된다.) (같은 노드에서 가져옴)



2) 해당 노드에 정확하게 minimum number 만큼의 키가 존재하는 경우, 해당 키를 지우고 같은 부모에 해당하는 sibling 노드에서 키를 하나 빌려온다. 그리고 그 빈 자리를 빌려온 키로 채운다. (옆의 sibling 노드에서 가져옴)

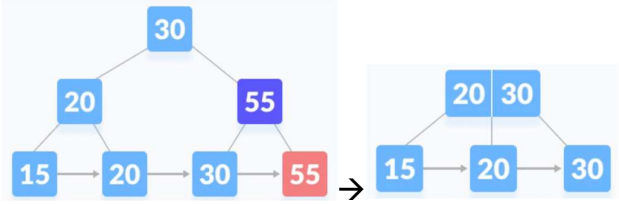


3) 1) 과 비슷하지만, RootNode 에 빈 자리가 생기는 경우이기 때문에 따로 빼서 구성하였다. key 를 지운 후 이웃하는 sibling 과 merge 한다. grandparent(level 2 개 올라간 부분)에 생긴 빈 자리를 inorder successor 로 교체한다. (예시에서는 오른쪽 sibling 의 가장 작은 원소)



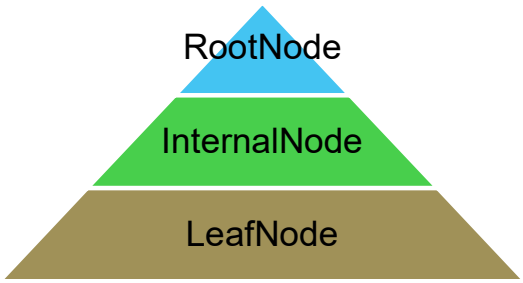
Case 3)

Deletion 에서 중요한 부분 중 하나는 merge 의 결과로 트리의 전체적인 높이가 낮아질 수 있다는 점이다.



Ⅱ. Implementation

먼저, 여러 개의 .py 파일에서 각각의 요소들(Node, RootNode, Entry 등)이 어떻게 대응되는지를 살펴보았다. node.py를 분석한 결과, 가장 크게 Node를 기준으로 B+ Tree를 바라보면 아래의 구조를 띠고 있었다.



위의 그림에서 RootNode와 LeafNode는 각각 1개의 level에 해당하며(충분히 element가 insert되었다는 가정하에) InternalNode는 1개 이상의 level로 구성될 수 있다. InternalNode는 ReferenceNode를 inherit받는 구조이다.

1. delete 함수

```
def delete(self, key):
    print(f"*** {key} delete ***")
    with self._mem.write_transaction:
        node = self._search_in_tree(key, self._root_node)
        print("delete--node key", node.entries) # for test
        if isinstance(node, LonelyRootNode):  # Root Node & Leaf Node
            print("In LonelyRootnode, delete entry");
            node.remove_entry(key)
            self._mem.set_node(node)
        else:  # Leaf Node
            print("In LeafNode, delete entry")
            parent = node.parent
            if node.can_delete_entry:
                print("case1: simple case, key is at only leaf, not internal") # test
                node.remove_entry(key)
                self._mem.set_node(node)
            else:
                print("case2: key is at leaf")  # test
                if node.smallest_key > parent.biggest_key:
                    # impossible case if it is well-implemented
                    assert ("Tree Error, parnet's biggest key < child's smallest key")
                elif node.smallest_key == parent.biggest_key:
                    print("case2-1: node is at the rightmost")  # test
                    src_node = self._mem.get_node(parent.biggest_entry.before)
                    src_node.parent = parent
                else:
                    print("case2-2")    # test
                    src_node = self._mem.get_node(node.next_page)
                    src_node.parent = parent
```

```
            if src_node.can_delete_entry:
                self._redistribute_leaf(src_node, node)
            else:
                node = self._merge_leaf(src_node, node)

            node.remove_entry(key)
            self._mem.set_node(node)

        parent = node.parent
        # change Interior Node

        if key < node.smallest_key and key >= parent.smallest_key:
            parent.get_entry(key).key = node.smallest_key
            self._mem.set_node(parent)

    print(f"delete complete\n\n")
```

`if node.can_delete_entry:` 은 Case1-1에 해당하는 가장 간단한 케이스를 의미한다. 각 노드에서 can_delete_entry는 현재 들어있는 key의 수를 minimum number와 비교하여 True/False를 반환하는 부분이다.

`elif node.smallest_key == parent.biggest_key:` 은 현재 지우고자 하는 키가 속한 노드의 가장 작은 키가 부모 노드의 가장 큰 키인 경우이다. 이때에는 부모 노드 중 가장 큰 엔트리의 바로 옆에 있는 노드를 src_node로 결정한다. 이 과정에서는 src_node를 구하는 것이 핵심인데, 이 과정을 통해 redistribute/merge 연산의 피대상자를 결정하게 된다.

else: 부분에서는 오른쪽 페이지(#로 시작하는 번호를 보면 된다.)를 src_node로 결정한다.

위에서 정한 src_node에 minimum number 이상의 키가 존재하는 상황이라면 그냥 키를 빌려올 수 있는 상황이다. 따라서, 이때에는 redistribute_leaf 함수를 호출한다.(→ 이때에는 양측 노드 모두 생존한다.) 만약, src_node도 빌려줄(src_node 입장에서는 제거당할) 상황이 아닐 경우에는 merge_leaf 함수가 호출된다.(→ 이때에는 양측 모두 생존할 수 없는 상황이므로 양 노드를 합치는 과정인 것이다.)

이후에 노드에서 키를 지운 후, 이를 _mem.set_node를 통해 실제 Disk에 반영한다.

2. _redistribute_leaf 함수

```
def _redistribute_leaf(self, src_node, dst_node):
    print("Leaf Node redistribute")

    # To do
    parent = src_node.parent

    # borrow one from sibling
    dst_node.insert_entry(src_node.smallest_entry)
    # delete if it is borrowed
    src_node.pop_smallest()

    for elem in parent.entries:
        if elem.after == src_node.page: elem.key = src_node.smallest_key
    print("redis_leaf fin:", parent.entries)
    return dst_node
```

다시 말하자면, 이번 프로젝트에서는 오른쪽→왼쪽 노드에서 key를 borrow하는 것을 기본 원칙으로 삼았다. src_node는 borrow의 피대상자이고, dst_node는 대상자이다. src_node의 키 하나를 dst_node로 넘겨야 하는 상황이다. 따라서, dst_node에 src_node의 최소값 키를 하나 삽입하고, 반대로 src_node 측에서는 그 키를 지워주면 “이동”의 효과를 얻을 것이다.

기본적으로, src_node.parent == dst_node.parent임을 인지해야 한다. parent.entries에서(type은 Reference의 List이다.) 어떤 Entry의 after가 src_node이면, 즉 src_node 바로 오른쪽 노드의 최좌측 엔트리이면 그것의 키를 src_node의 최소 키로 바꾸는 것이다.

3. _merge_leaf 함수

_merge_leaf 함수가 하는 기능을 거시적으로 설명하면 src_node와 dst_node를 하나로 합치는 것이다. 두 노드가 합쳐지는 과정에서 어떤 것이 달라질까? 우선 parent 입장에서는 각 children이 존재하고 그것들을 나누고 있는 InternalNode들이 존재한다. 즉 InternalNode가 N개 존재하는 parent의 입장에서는 N+1개의 children이 있는 것이다. 두 개의 자식 노드가 merge되는 상황이라면 부모 입장에서는 자신의 InternalNode 하나가 없어지는 것이다.

따라서, 먼저 우리는 그 없어지는 InternalNode를 구해야한다. 코드에서는 _to_be_deleted가 이에 해당한다고 볼 수 있다.

```
def _merge_leaf(self, src_node, dst_node):
    print("Leaf Node merge")
    parent = dst_node.parent

    # To do
    if dst_node.smallest_key == parent.biggest_key:
        for elem in parent.entries:
            if elem.before == src_node.page:
                _to_be_deleted = parent.get_entry(elem.key)
    else:
        for elem in parent.entries:
            if elem.after == src_node.page:
                _to_be_deleted = parent.get_entry(elem.key)
```

그 InternalNode를 구한 후에는 그 내부 노드를 지웠을 때 parent의 상황이 어떻게 될지를 보아야 한다. 가장 간단한 경우는 parent에 충분한 개수가 있어서 지워도 아무런 조치가 필요없는 경우일 것이다. src_node에 있던 모든 것들을 dst_node에 옮겨 담는다. 아래의 코드는 우선 parent가 RootNode일 때에 해당하는데, 나머지는 이어서 서술하였다.

```
""" Merge src_node and dst_node
    Delete parent entry if needed """
if isinstance(parent, RootNode):
    if parent.can_delete_entry:
        # To do
        for elem1 in src_node.entries:
            dst_node.insert_entry(elem1)

        for elem2 in src_node.entries:
            src_node.remove_entry(elem2)

        self._mem.set_node(dst_node)
        parent.remove_entry(_to_be_deleted.key); self._mem.set_node(parent)
```

parent에서 minimum number of keys 조건을 만족하지 못하는 경우 우선 src_node를 dst_node에 옮겨 담는 작업만 한다.

```
else:
    # To do
    for elem1 in src_node.entries:
        dst_node.insert_entry(elem1)

    for elem2 in src_node.entries:
        src_node.remove_entry(elem2)
```

parent가 RootNode가 아니라고 하더라도, 바로 삭제하고 끝날 수 있는 경우라면 맨 위의 과정을 그대로 따라할 수 있을 것이다.

```
elif parent.can_delete_entry:
    # To do
    for elem1 in src_node.entries:
        dst_node.insert_entry(elem1)

    for elem2 in src_node.entries:
        src_node.remove_entry(elem2.key)
    # update for dst_node, src_node
    self._mem.set_node(dst_node); self._mem.set_node(src_node)

    parent.remove_entry(_to_be_deleted.key)
    self._mem.set_node(parent)
```

else문까지 온 경우는, parent가 RootNode가 아니면서 바로 delete가 가능하지도 않은 가장 복잡한 경우가 되는 것이다. src_node의 parent를 grand_parent 단에서 탐색을 한다. 만약 src_parent에 대해 마찬가지로 minimum number of keys 조건을 만족하는지에 따라 분기해야 한다. 그 조건을 만족한다는 것은 parent level에서 sibling끼리 엔트리를 빌려올 수 있다는 것이고(Leaf에서와 마찬가지인 상황), 만족하지 않는다는 것은 parent 단에서의 Merge 과정이 필요하다는 것이다. (이 또한 Leaf 단에서와 마찬가지인 상황)

```
else:
    # To do
    grand_parent = parent.parent
    for elem in grand_parent.entries:
        if (parent == elem) and (elem == grand_parent.smallest_entry):
            src_parent =
self._mem.get_node(grand_parent.biggest_entry.after)
        elif (parent == elem) and (elem == grand_parent.biggest_entry):
            src_parent =
self._mem.get_node(grand_parent.biggest_entry.before)
        else:
            src_parent =
self._mem.get_node(grand_parent.biggest_entry.after)
```

```
    if src_parent.can_delete_entry:
        self._redistribute_parent(src_parent, parent)
    else:
        # Merge between two lists
        dst_node.entries = dst_node.entries + src_node.entries

        for elem1 in parent.entries:
            if elem1.after == src_node.page:
                parent.remove_entry(elem1.key)

        for elem2 in parent.entries:
            if elem2.before == src_node.page:
                elem2.before = dst_node.page
        parent = self._merge_parent(src_parent, parent)
```

만약 parent 단에서 merge를 해야하는 상황이라면 Leaf 단에서 논의한 _to_be_deleted와 같이 지워져야 할 entry 하나를 탐색하여 그것을 지우고 _merge_parent를 호출한다.

이 함수는 결과적으로 Merge하는 것이므로 Merge한 결과에 해당하는 dst_node를 반환하고 함수는 끝이 난다. (이 부분은 생략)

4. _redistribute_parent 함수

이 함수의 proto-type은 def _redistribute_parent(self, src_node, dst_node) 이다. 위에서 이 함수를 부를 때 src_node에는 src_parent가, dst_node에는 parent가 들어가 호출됨을 상기한다.

즉, parent 단의 같은 level에 있는 두 개의 InternalNode 사이에서 entry를 exchange하는 과정인 것이다. 이 함수에서 선언된 parent는 사실상 이전 레벨에서 이야기하면 grand-parent에 해당된다. dst_node의 최소 키 > parent의 최대 키인 경우 삭제되어야 할 키는 src_parent의 최대 키로 지정될 것이다. 이 src_parent의 최대 키가 포함된 entry를 dst_node ←→ src_node 사이에서 exchange한다.

```
def _redistribute_parent(self, src_node, dst_node):
    print("Parent(interior) Node redistribute")

    # To do
    parent = dst_node.parent
    if (dst_node.smallest_key > parent.biggest_key):
        _to_be_moved = src_node.entries[-1].key
        dst_node.insert_entry(src_node.get_entry(_to_be_moved))
        src_node.remove_entry(_to_be_moved)

        self._mem.set_node(dst_node); self._mem.set_node(src_node)

        for elem in parent.entries:
            if (elem.before == src_node.page):
                elem.key = src_node.biggest_key
        self._mem.set_node(parent)
```

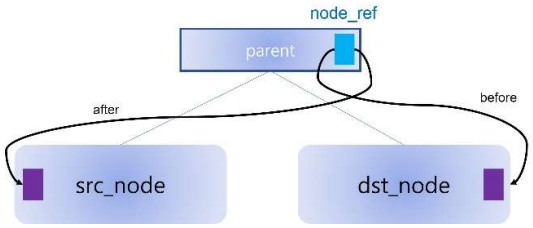
이외의 경우라면 src_node의 최대 키를 지우고 그것을 dst_node의 말단부에 삽입하면 된다.

```
else:
    dst_node.insert_entry_at_the_end(src_node.pop_smallest())
    self._mem.set_node(dst_node); self._mem.set_node(src_node)

    for elem in parent.entries:
        if (elem.after == src_node.page):
            elem.key = src_node.smallest_key
    self._mem.set_node(parent)
```

5. _merge_parent 함수

_split_parent에서 ref를 만든 것에 착안하여, dst_node의 parent에서 entries에서 가장 마지막 component를 의미하는 node_ref를 선언하였다.




```
def _merge_parent(self, src_node, dst_node):
    # print("src_node: ", str(src_node.entries))
    # print("dst_node: ", str(dst_node.entries))
    print("Parent(interior) Node merge")
    parent = dst_node.parent

    # To do
    node_ref = parent.entries[-1]
    node_ref.before, node_ref.after = dst_node.biggest_entry.after,
src_node.smallest_entry.before
```

이제 위의 node_ref를 dst_node에 삽입한 후, Leaf에서와 같은 방식으로 redistribute를 진행한다.

```
dst_node.insert_entry(node_ref)
print("*****test1")
print(src_node.entries)
print(dst_node.entries)
print(src_node.parent.entries)
for elem1 in src_node.entries: dst_node.insert_entry(elem1)
for elem2 in src_node.entries: src_node.remove_entry(elem2.key)
print("*****test2")
for elem in parent.entries:
    if elem.before == src_node.page: elem.before = dst_node.page
print("*****test3")
```

parent가 RootNode일 때를 고려한다. parent가 can_delete_entry, 즉 minimum number of keys의 특성을 만족하느냐의 여부에 따라 분기하는 것은 merge_leaf에서와 동일한 방식이다.

```
""" Merge src_node and dst_node
    Delete parent entry if needed """
if isinstance(parent, RootNode):
    # case: parent is the most-upper(ROOT)
    if parent.can_delete_entry:
        # the root is enough
        print("*****if")
```

```
            parent.remove_entry(node_ref.key)
            self._mem.set_node(parent)
            # return dst_node
        else:
            # To do
            print("*****else")
            parent.remove_entry(node_ref.key)
            # list concatenating
            parent.entries = parent.entries + dst_node.entries
            # update all nodes to disk
            self._mem.set_node(src_node)
            self._mem.set_node(dst_node)
            self._mem.set_node(parent)

            print("Root Node Delete")
```

마찬가지로, else문까지 도달한 경우는 parent가 RootNode가 아니면서 .can_delete_entries 조건도 만족하지 않는 가장 복잡한 경우이다. 동일하게 grand_parent, 즉 parent의 윗 레벨(level)에서 src_parent를 찾아 redistribute나 merge를 상황에 맞게 재귀적으로 호출한다.

이 과정은 위로 한 level씩 올라가면서 can_delete_entries 조건을 만족하는 노드가 나올 때까지 재귀적으로 호출되며, 만약 이것이 RootNode까지 전파될 경우 B+ Tree의 높이가 낮아지는 가능성이 생긴다. (merge의 경우)

```
elif parent.can_delete_entry:
    # To do
    parent.remove_entry(node_ref.key)
    self._mem.set_node(parent)

else:
    # To do
    parent.remove_entry(node_ref.key)
    self._mem.set_node(parent)
    grandparent = parent.parent
    for g_elem in grandparent.entries:
        if g_elem.before == parent.page:
            src_parent = self._mem.get_node(g_elem.after)
    if src_parent.can_delete_entry:
        self._redistribute_parent(src_parent, parent)
    else:
        parent = self._merge_parent(src_parent, parent)
```

Ⅲ. Execution

여기에는 Homework4.py를 실행한 결과를 표시하였다.

1) zk까지 insert가 모두 끝난 후의 트리의 모습

[#27: s,]

[#9: g, m, <parent #27>] [#26: y, ze, <parent #27>]

[#3: c, e, <parent #9>] [#8: i, k, <parent #9>] [#13: o, q, <parent #9>] [#17: u, w, <parent #26>] [#21: za, zc, <parent #26>] [#25: zg, zi, <parent #26>]

[#1: a, b, <parent #3>] [#2: c, d, <parent #3>] [#4: e, f, <parent #3>] [#5: g, h, <parent #8>] [#6: i, j, <parent #8>] [#7: k, l, <parent #8>] [#10: m, n, <parent #13>] [#11: o, p, <parent #13>] [#12: q, r, <parent #13>] [#14: s, t, <parent #17>] [#15: u, v, <parent #17>] [#16: w, x, <parent #17>] [#18: y, z, <parent #21>] [#19: za, zb, <parent #21>] [#20: zc, zd, <parent #21>] [#22: ze, zf, <parent #25>] [#23: zg, zh, <parent #25>] [#24: zi, zj, zk, <parent #25>]

2) j의 delete가 끝난 후의 트리의 모습

[#27: g, s, y, ze,]

[#3: c, e, <parent #27>] [#8: i, m, o, q, <parent #27>] [#17: u, w, <parent #27>] [#21: za, zc, <parent #27>] [#25: zg, zi, <parent #27>]

[#1: a, b, <parent #3>] [#2: c, d, <parent #3>] [#4: e, f, <parent #3>] [#5: g, h, <parent #8>] [#6: i, k, l, <parent #8>] [#10: m, n, <parent #8>] [#11: o, p, <parent #8>] [#12: q, r, <parent #8>] [#14: s, t, <parent #17>] [#15: u, v, <parent #17>] [#16: w, x, <parent #17>] [#18: y, z, <parent #21>] [#19: za, zb, <parent #21>] [#20: zc, zd, <parent #21>] [#22: ze, zf, <parent #25>] [#23: zg, zh, <parent #25>] [#24: zi, zj, zk, <parent #25>]

3) n의 delete가 끝난 후의 트리의 모습

[#27: g, s, y, ze,]

[#3: c, e, <parent #27>] [#8: i, m, q, <parent #27>] [#17: u, w, <parent #27>] [#21: za, zc, <parent #27>] [#25: zg, zi, <parent #27>]

[#1: a, b, <parent #3>] [#2: c, d, <parent #3>] [#4: e, f, <parent #3>] [#5: g, h, <parent #8>] [#6: i, k, l, <parent #8>] [#10: m, o, p, <parent #8>] [#12: q, r, <parent #8>] [#14: s, t, <parent #17>] [#15: u, v, <parent #17>] [#16: w, x, <parent #17>] [#18: y, z, <parent #21>] [#19: za, zb, <parent #21>] [#20: zc, zd, <parent #21>] [#22: ze, zf, <parent #25>] [#23: zg, zh, <parent #25>] [#24: zi, zj, zk, <parent #25>]

4) m의 delete가 끝난 후의 트리의 모습

[#27: g, s, y, ze,]

[#3: c, e, <parent #27>] [#8: i, o, q, <parent #27>] [#17: u, w, <parent #27>] [#21: za, zc, <parent #27>] [#25: zg, zi, <parent #27>]

[#1: a, b, <parent #3>] [#2: c, d, <parent #3>] [#4: e, f, <parent #3>] [#5: g, h, <parent #8>] [#6: i, k, l, <parent #8>] [#10: o, p, <parent #8>] [#12: q, r, <parent #8>] [#14: s, t, <parent #17>] [#15: u, v, <parent #17>] [#16: w, x, <parent #17>] [#18: y, z, <parent #21>] [#19: za, zb, <parent #21>] [#20: zc, zd, <parent #21>] [#22: ze, zf, <parent #25>] [#23: zg, zh, <parent #25>] [#24: zi, zj, zk, <parent #25>]

5) zg의 delete가 끝난 후의 트리의 모습

[#27: g, s, y, ze,]

[#3: c, e, <parent #27>] [#8: i, o, q, <parent #27>] [#17: u, w, <parent #27>] [#21: za, zc, <parent #27>] [#25: zh, zj, <parent #27>]

[#1: a, b, <parent #3>] [#2: c, d, <parent #3>] [#4: e, f, <parent #3>] [#5: g, h, <parent #8>] [#6: i, k, l, <parent #8>] [#10: o, p, <parent #8>] [#12: q, r, <parent #8>] [#14: s, t, <parent #17>] [#15: u, v, <parent #17>] [#16: w, x, <parent #17>] [#18: y, z, <parent #21>] [#19: za, zb, <parent #21>] [#20: zc, zd, <parent #21>] [#22: ze, zf, <parent #25>] [#23: zh, zi, <parent #25>] [#24: zj, zk, <parent #25>]

6) zk의 delete가 끝난 후의 트리의 모습

결론적으로 이 부분은 제대로 된 결과가 나오지 않았다. 우선 zk가 삭제되는 경우는 rightmost entry인 case에 해당한다. 이때에는 예외적으로 왼쪽의 sibling에서 entry를 borrow할 수 있다는 policy를 세웠다. 우선 zk가 LeafNode 단에서 삭제가 되기는 하는데 그 이후의 과정에서 문제가 있는 것으로 추정된다.

버그를 테스트하기 위해 “*****”를 넣어뒀는데

```
for elem1 in src_node.entries: dst_node.insert_entry(elem1)
for elem2 in src_node.entries: src_node.remove_entry(elem2.key)
print("*****test2")
```

위의 2개의 for loop에서 무한정 대기하는 상황이다.