1.

Insert

- 根據key值和global_depth去放到哪個hash_bucket
  hash_index = key & ((1 << global_depth)-1);
  然後找bucket裡面有沒有key值相同的
  key值相同就用value取代原本的
  最後看如果bucket還沒放滿就直接放到第一個
  滿了就**extend**
  然後再重新**insert**一次

extend

1. 如果local_depth == global_depth 就擴展table
   然後把新增的映射到原本的bucket
2. 重新分類原bucket裡的entry
3. 把所有應該指到new_bucket的table都重新指向

2.

remove

- 根據key值和global_depth去找到hash_bucket
  hash_index = key & ((1 << global_depth)-1);
- 接著從bucket找要刪的key值刪掉並把數量減1
  最後檢查需不需要shrink

shrink

1. 找到對應的另一個桶 & 檢查是不是同樣的depth(if not => return)
2. local_depth-1 & 重新把應該映射到pair的table重新映射 & 刪除原桶

3. 檢查pair的pair是不是因為depth比較淺，空著沒有shrink，現在相同可以**shrink**了
4. 確認需不需要half_table

half_table

- 檢查global_depth是不是大於1
- 檢查全部的bucket的local_depth是不是都比global_depth小
  make table half , global_depth-1;

[紀錄用的hackmd]

3.

b+tree

- 每個節點，最多只能有 M 個子樹。
- 每個節點中的內部節點，最多只能有 M - 1 個。
- 每個非葉子節點(就是非最底層的節點)，只儲放索引用。
- 每個葉節點，儲放了實際資料。
- 每個葉節點，會包含一個指針，指向右邊的葉節點。

extendible hashing

- 能夠更快找到要找的資料
- 不過資料可能會比較不平衡，可能有bucket的depth特別大
- 在插入的時候，需要做的事比較多，耗的時間可能會比較長
- 做範圍搜尋沒有效率

如果沒有用到範圍搜尋，我覺得extendible hashing比較好用，能更快找到要的值，雖然在插入的時候耗的時間會比較長。
會用到範圍搜尋的話，只能用b+tree。