An implementation of on-disk concurrent skip list for an alternative of B-tree Index

- Author: Ryo Kanbayashi (ryo.contact@gmail.com)

Introduction

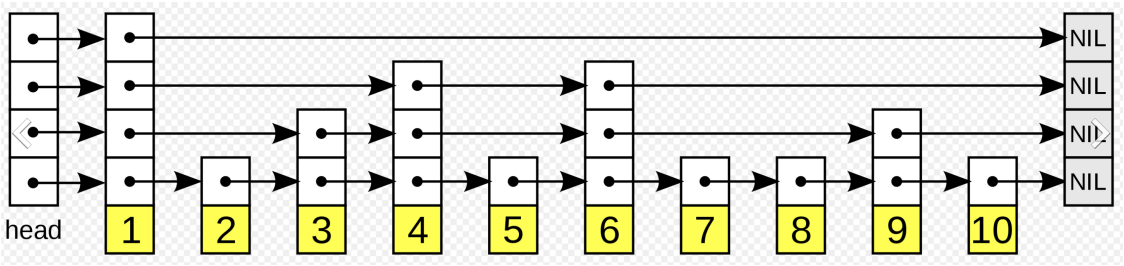
- The purpose of this document is to share the knowledge and findings that I have gained through the design and implementation of on-disk concurrent Skip Lists
- I couldn't find any web pages and books that provide same kind of information shared in this document in a summarized form, so this document should be useful to some people as valuable knowledge and insight

About Skip List

What is Skip List?

- A probabilistic data structure that realizes Key-Value store
    - "probabilistic" data structure is unique compared to other data structures that provide similar functionality
    - Data can be searched (got), inserted, and deleted in O (log N)
        - The base of logarithm can be varied by design, but most are 2 or 4; the author designed it to be 2
- stored data can be retrieved efficiently in sorted order based on the definition of the size of the data used as the Key as same as B+ tree
    - e.g.) Given Key-A and Key-B where A < B, it is possible to retrieve the values stored with x satisfying A <= x <= B in ascending order of the key
- Overview of Data Structure
    - Basically, it is just a concatenated list, and a linear search can be performed to reach the node that holds the desired entry
    - However, as the name "Skip" implies, when creating a node, a path is created that allows skipping some nodes with a certain probability
        - As a result, it is like moving from the first station on the train to the destination while taking the limited express, express, local, and local train in that order
        - To use the above analogy, creating a path is like recognizing each different train service as a stop (among the services from each stop to express, the probability of recognizing a station as a stop is determined)
        - Each operation is distinguished by its level, and the number of stops is smaller for those with higher levels. The level at which a train stops at each station is referred to as level 1 in this document
    - For details, please refer to Wikipedia's skip list page

- The image below is from the above Wkipedia page



Good points of Skip List

- It can provide the same level of functionality and access performance as B+ tree, yet is relatively simple to implement.
- B-tree variants
  - The data structure of the B-tree variants are a balanced tree, so if the distribution of data in the tree structure becomes unbalanced, a process called rebalancing is performed
  - Internal processes called node split and merge (or another type of process that can achieve the same purpose) are performed as needed
  - Basically, rebalancing occurs when inserting and deleting data, which are update operations, but the implementation of merge processing for deletion tends to be particularly complex
  - Unless a special design is adopted, the implementation should take into account the case where the rebalancing process is not a local update in a tree, but a wide-ranging update
- Skip List
  - Skip List is relatively simple to implement because its base is just a concatenated list and no rebalancing is performed
    - Node split and delete are necessary, but merge is not necessary

What is on-disk concurrent Skip List in this document?

- On-disk
  - Take on-memory XXX is basically operated while data is still in memory
  - On-disk XXX changes the location of the sub-element (node in the B-tree variants and Skip List) that handles stored data, such as being written to non-volatile storage, or conversely being loaded into memory
  - It may be said that a program which is designed to efficiently handle a group of data whose total size does not fit in the memory capacity by setting up a cache area in memory and drowing on it
- **Concurrent Accessible**
  - The design of partial locking inside the data structure allows multiple threads to access the data structure at the same time (as much as possible) while maintaining the integrity of the data
  - <=> If the entire data structure is controlled by a single lock, only one thread can access it at the same time
- Skip List

Good points of Skip List (from the perspective of handling concurrent access)

- It is relatively simple to deal with concurrent access
- In this document, locking for exclusive control is done on a per-node granularity

    - (In B-tree variants, tree locking may be used in some cases, but the basic principle is the same)
- B-tree variants
    - Implemantaton of code to perform rebalancing with appropriate exclusive control is complex (difficult to understand)
    - It is necessary to deal with cases where the update range for rebalancing is extensive
    - Locks must be acquired while taking care to avoid deadlocks, but the sequence itself tends to be difficult to understand
- Skip List
    - Lower complexity than B-tree variants
        - In both programs, most update operations search for nodes to insert or delete data while performing exclusive control, and then update the contents of the found nodes as needed to complete the process
        - The hardest part of the implementation is when updating one node requires updating other nodes as well...
            - In the B-tree variants, the need for rebalancing also causes node splits and merges
            - In the Skip List, a split occurs when there is no free space on a node to hold an entry, and a node deletion occurs when the number of entries in a node reaches zero, but since rebalancing is not performed, the process is relatively simple
            - The necessary processing includes updating the connection information between nodes and moving the entries between nodes
            - The base of Skip List is a simple linked list, and even if an entry is added or deleted, nodes position is never changed

Bad points of (naive) Skip List

- Because of its probabilistic desing and lack of rebalancing, there are not few cases where access cannot be performed in log N steps, compared to the B-tree variants
- Efficient concurrent access is harder to achieve than B-tree variants
    - (You had better to read the section about concurrent implementation before read here...)
    - On Skip List and B-tree variants, the threads start their search from the same starting point, and if one of the threads that goes along the same route acquires a W-lock of a node first, it will cause a contentions with the other threads and the throughput of concurrent access decreases due to these
    - However, in the Skip List, if a thread acquires a W-lock of a node, other threads that want to pass through the locked node is blocked at the node on all levels up to the level of the node
        - Fundamental difference may be that branch and leaf (node) locks are not separated in Skip List unlike B-tree variants
        - (I think it is necessary to take into account that there are areas that become inaccessible while rebalance is processed in the case of B-tree variants)
- Disk cache efficiency is poor than B-tree variants
    - "Why are skip lists not preferred over B+-trees for databases? - stackoverflow.com"
- Range scan (≒iterating by specifying a range) of entries can only be performed in the direction decided at the time of data structure design
    - Although it may be possible to do it in both directions, the complexity of the logic (especially for concurrent access) may increase significantly if you want to achieve it without reducing the processing efficiency too much

Explanation of the specifications and design of the on-disk concurrent Skip List created by the author

Background of development

- In the process of developing RDB, author thought of implementing B+tree indexes which are widely used in major RDBs, but as author have explained so far, I found that it is very difficult especially to the point where concurrent access supporting. So, In order to create a Skip List index, I implemented a Skip List container
    - For this reason, the implementation exists in the code base of SamehadaDB, of which author is the main developer
    - [Reference] LevelDB (on-disk KVS), which is also used in Google Chrome and other applications, uses Skip List in its index (on-memory) to improve access efficiency to entry data in memory (not on disk)
- Therefore, some of the explanations and design in this document may be influenced by the purpose described above
- Nevertheless, it is designed to be general-purpose usable, and it should not be difficult for viewers to change or extend almost all of parts

External Specifications

- Notes
    - Some specifications are not intentional, but simply because the detailed design has not been completed
    - Some of the author's implementations are not as described
        - This document is intended to share knowledge about design of on-memory concurrent Skip List, which does not necessarily have to be consistent with the author's implementation
- Assumptions
    - Data is handled as a mapping of Key: number, string, etc -> Value: number, string, etc, and this pair is called an entry
        - It is assumed (for the sake of explanation) that there exists a type (referred to as KV type in this document) that can represent multiple data formats and provides comparison functions (optional) and {"",de} serialization with byte array for each of them, and that Key and Value are treated uniformly as KV type
        - Any data format can be supported as possible as it is expressible in KV type, but the data format to be used as Key must be implemented with large/small comparison
    - Key data format is fixed to the type specified at the time of Skip List instantiation
    - Keys are processed as unique. Multiple entries with keys that are recognized as same value by the comparison function cannot coexist. For example, Inserting an entry with the same key overwrites the Value (Insert = UPSERT).
        - Aside: As a container used in RDB index, it is not usable unless it is available in the form Key -> [Value], not Key -> Value
            - Combine the original Key and Value in a format called "Comparable Format" and treat combined data as Key'. Then Key' -> Value mapping can be used to support Key -> [Value] mapping

            - To retrieve a set of values corresponding to an original key, generate an Iterator by specifying Key ++ **lower limit of the value range** to Key ++ **higher limit of the value range** as the key
            - To delete a specific Key-Value combination, use Remove specifying Key'
            - To delete all entries for an original Key, use the above method to extract all of them and then delete them one by one
            - However, as mentioned below, the Iterator function is not atomic to other operations, and the case of using the value obtained by Iterator to perform some operation (such as deleting all entries of a specific Key above) is also not atomic if done naively with the design in this document. This is a point of concern
- Provided operations
    - Retrieving an entry
        - SkipList::Get(key *KV) *KV
            - If the retrieval succeeds, the return value is the content of Value. If it fails, null is returned
    - Inserting (updating) an entry
        - SkipList::Insert(key *KV, val *KV) bool
            - Returns boolean value of success or failure of the operation
    - Removing an entry
        - SkipList::Remove(key *KV) bool
            - The return value is the truth value of success or failure of the operation
    - Getting the iterator and doing range scan with the iterator
        - SkipLisr::Iterator(start *KV, end *KV) *SLItr
            - start is the lower limit of the scan range, end is the upper limit. if start and end are passed null, that side is treated as unspecified
            - SLItr type is an iterator that returns one value at a time, sorted in ascending order. The return value is the value obtained by scanning the Skip List in the specified range when the Iterator function is called.
                - However, since the scan is not atomic to other operations, it is not guaranteed to be a snapshot of the moment the Iterator function is called

Internal design

- On-disk support
    - Nodes are associated with blocks (fixed length) in a file for on-disk persistence and these blocks are called pages
        - For example, locking/unlocking a node for exclusivity control is basically the same as doing so for a page
        - Each page has a unique page ID
    - Data for a page is obtained via a component called the page manager. Thread notifies the page manager when the thread finished accessing the page and when obtained page is no longer needed
        - The page manager has an internal buffer in memory (cache area for pages), and individual pages are loaded onto the buffer as needed, but are written to disk if necessary to allocate space for other pages, and so on
    - Page (Note: this is different from "page" in OS memory management mechanism)
        - Page type

        - Holds the page ID of the corresponding page in a member variable
            - When an instance corresponding to a page A (page ID=>A) exists in memory, it is controlled so that only one instance corresponding to page A exists, and when page A is manipulated in a program, it is accessed by sharing a reference to the same instance
        - Read-write locking is possible (with the corresponding Mutex which is a member variable)
        - Has a counter called "pin count" as a member, and if it is 0, it means that there is no thread using the page, and the page manager recognizes it as a page that can be cached out to disk
        - Has a reference to the area of page data loaded in memory
- Page Manager (hereafter referred to as PM)
    - The PM shall provide the following I/F (assumed behaviors are only described below)
        - PageManager::NewPage(newPage *Page) int32
            - Generates a new page and returns ID of the allocated page and a reference to the Page type entity.
            - The pin count of the page is 1
        - PageManager::DeallocatePage(pageId int32)
            - The page corresponding to the given ID is added to the list of reusable pages that can be returned by NewPage
                - The corresponding area of the DB file is made available for reuse
            - Once a page ID is passed to DeallocatePage, the ID is not reused
                - FetchPage with a page ID that has already been deallocated returns null
        - PageManager::FetchPage(int32 pageId) *Page
            - When a page ID is passed, a Page type instance is returned with reference to the page data if it is cached in memory, or if not, to the page data after reading data from disk in the buffer
            - If a page ID that does not exist in memory or on disk is passed, null is returned
            - Page is returned with a pin count of +1
        - PageManager::UnpinPage(pageId int32, isDirty bool)
            - Notifies the PM that access to the page corresponding to the passed page ID is finished
            - The isDirty flag is set to true if the page content has been updated
                - hint information for efficient caching
            - Set the pin count of the corresponding page to -1. PM recognizes pages with a pin count of 0 as pages that can be cached out
                - When multiple threads are accessing a page, the pin count does not always become 0 just because one thread calls UnpinPage the page
    - Reference
        - If you are interested in the internal design of PageManager, please refer to the slides of CMU's open courseware lecture below
            - Note: In this lecture, Page Manager is called "Buffer Pool"
            - For more information about the lecture, please see the top of the repository which PDFs are stored at
            - 03-storage1.pdf
            - 04-storage2.pdf

        - 05-bufferpool.pdf
- Node
    - Node type
        - Has a reference to a Page type instance, through which a Read-Write lock is possible
        - It also has access to the page data area and provides getter/setter equivalent I/F groups for metadata and entry data stored in/on the page
        - Since there is only one member mentioned above and other data exists in the page data area, they are accessed via {"",de} serialization performed by the I/Fs mentioned above
        - In most cases, multiple entries are stored

Other important points

- Constraints
    - The size limit of an entry is the page size minus the size of the page header area and the metadata area required when serializing the KV type
        - However, it is necessary to consider the case where two new nodes must be created when splitting, as described below, and this would complicate the process. In reality, it would be safe to set the maximum size at a little less than half of the page size
- Additional points that should be considered for a datastore
    - Does it support transactions?
        - => Not supported
        - (Because it was originally developed as a container for RDB indexing, it is possible to use the transaction mechanism of RDB itself when it is embedded in RDB)
    - Does it support Logging/Recovery?
        - => Not supported
        - Therefore, if the program terminates without writing all the dirty pages in the PM to disk due to a crash or some other reason, there may be entries that have not been persisted

Why is it so difficult to make on-disk concurrent Skip List?

Let me summarize before continuing.

- Because one node contains multiple entries
    - On-memory implementations are usually implemented in a way that one node corresponds to one entry (please see later section)
    - In contrast, the structure described above requires storeing multiple entries.
        - Sequential disk I/O considering OS page size (more efficient than random access), and data caching to hide slow disk I/O compared to main memory as much as possible, data needs to be handled in page units
        - The page size needs to be larger than 4KB based on the default settings of recent OSes, and as a result, it is reasonable to design a page that contains multiple entries
        - However, a structure that includes multiple entries may simply require logic expansion, and in many cases, processing that would be simple if handled on a one-to-one basis is no longer so
- There was information on a method to enable concurrent access to on-memory Skip Lists (please see later section), but as mentioned above, the data structure was different, so additional considerations were required when applying the method

Logic design, etc.

Algorithm for on-memory Skip List

- The logic explained in the winning team's slides at ACM ICPC Maraton Prague 2015 was used as a basis
  - **slide**
  - However, the one described in this slide is an on-memory implementation and each node is associated with one entry
  - Additionaly, the implementation is not accessible concurrently
- The following explanation assumes an understanding of the implementation presented in the "ACM ICPC Maraton Prague 2015 Winning Team's Slides"
  - The following points are of particular importance for understanding what follows
    - How the code traverses the nodes from the first node to the node to be explored
    - What information is stored in the process of traversing a node?
      - => List of references to the node at which location transfered
    - What is needed to add or delete a node?
      - => processing connection info of nodes described above

Extension to a form where one node holds multiple entries (vs. on-memory implementation, regarding data structure)

- The on-disk version of this document is designed to have multiple entries because one node corresponds to one page
- Each node (Node type) should provide the following (including those that are necessary in the one-to-one correspondence)
  - (Assuming that the entries are still sorted by Key even within a node)
  - Retrieving, deleting, and adding entries by Key (and Value, if necessary)
    - The fact that entries are sorted by Key enables bisection search, and the overall processing time can be reduced to O (log N), even if it includes processing after node search.
    - The retrieval operation returns a entry with the closest value among the entries with keys smaller than the specified Key, even if the entry which has specified Key does not exist
      - Necessary to realize Iterator
  - Retrieving the entry with the smallest Key in a node
    - Note that even if an entry is returned, only the Key is used
  - Obtaining the level of a node
    - The value set when the node was created
  - Obtains and sets the page ID of the node connected as the next node at the specified level
    - The acquisition and setting must be possible from level 1 to the level of the node

[Reference] Data layout of a node in SamehadaDB

For convenience of use, Value is a fixed length of 4 bytes. LSN is currently used only as an update counter as

described in later section

```
// Slotted page format:
//  ------------------------------------------------------
//  | HEADER | ... FREE SPACE ... | ... INSERTED ENTRIES ... |
//  ------------------------------------------------------
//                               ^
//                               free space pointer
//
//  Header format (size in bytes):
//  ----------------------------------------------------------------------------------------------------------
//  | PageId (4)| LSN (4) | level (4)| entryCnt (4)| forward (4 * MAX_FOWARD_LIST_LEN) | FreeSpacePointer(4) |
//  ----------------------------------------------------------------------------------------------------------
//  ------------------------------------------------------------
//  | Entry_0 offset (2) | Entry_0 size (2) | ..................|
//  ------------------------------------------------------------
//  offsetEntryInfos (=sizeBlockPageHeaderExceptEntryInfos)
//
//  Entries format (size in bytes):
//  ------------------------------------------------------------------------------------------------------------------------------------------
//  | HEADER | ... FREE SPACE ... | ....more entries....| Entry1_key (1+x) | Entry_1 value (4) | Entry0_key (1+x) | Entry_0 value (4)|
//  ------------------------------------------------------------------------------------------------------------------------------------------
//                               ^                  ^ <-------------- size ---------------> ^ <-------------- size ---------------->
//                               freeSpacePointer     offset(from page head)                  offset(...)
//
// Note:
//  placement order of entry location data on header doesn't match with entry data on payload
//  for entry insertion cost is keeped lower
//
//  Entry_key format (size in bytes)
//    = Serialized types.Value
//  ------------------------------------------------
//  | isNull (1) | data according to KeyType (x) |
//  ------------------------------------------------
```

Extension to a form where one node holds multiple entries (vs. on-memory implementation, for logic)

- Code example in Go language is also used here
- The code described here is at a stage where concurrent access is not considered
- The description is omitted where the extension method is self-explanatory
- Node search process
    - The code includes a comment [number], each of which is described below
    - At the end of the call of FindNode, the pin count of the return value **foundNode** is incremented (+1)

```
// Utility function to cast a pointer of type Page to a pointer of type Node
func FetchAndCastToNode(pm *PageManager, pageId int32) *Node {
    fetchedPage := pm.FetchPage(pageId)
    return (*Node)(unsafe.Pointer(fetchedPage))
}

// Arguments
// key: Key of the entry to be searched
// opType: a constant indicating what the FindNode function was called for. Pass a
constant corresponding to one of GET, INSERT, or REMOVE
// Return value
// isSuccess: Whether the search was successful (only true is returned in this
code example)
// foundNode: Pointer to the node found as a result of the search. It only means
```

```
that the foundNode node may contain the entry corresponding to key argument.
existence is not guaranteed
// predOfCorners_: page ID of the previous node connected to the node when it was
"switched" at each level. The index is (level - 1).
// corners_: Page ID of the node that was "switched" at each level. The value of
level 1 corresponds to foundNode. The index is (level - 1)
func (sl *SkipList) FindNode(key *KV, opType SkipListOpType) (isSuccess bool,
foundNode *Node, predOfCorners_ []int32, corners_ []int32) {
    pred := FetchAndCastToNode(sl.pm, sl.HeadNodeID)

    predOfPredId := int32(-1)
    predOfCorners := make([]int32, MAX_FOWARD_LIST_LEN)
    // entry of corners is corner node or target node
    corners := make([]SkipListCornerInfo, MAX_FOWARD_LIST_LEN)
    var curr *Node = nil
    for ii := (MAX_FOWARD_LIST_LEN - 1); ii >= 0; ii-- { //[1]
        for { //[2]
            curr = FetchAndCastToNode(sl.pm, pred.GetForwardEntry(int(ii)))
            if !curr.GetSmallestKey(key.ValueType()).CompareLessThanOrEqual(*key)
{ // ! (key >= curr.GetSmallestKey())
                // (ii + 1) level's corner node or target node has been identified
(= pred)
                break
            } else {
                // keep moving foward
                predOfPredId = pred.
                UnpinPage(pred.GetPageId(), false)
                pred = curr
            }
        }
        if opType == SKIP_LIST_OP_REMOVE && ii ! = 0 && pred.GetEntryCnt() == 1 &&
key.CompareEquals(pred.GetSmallestKey(key.ValueType()))) {
                // [3]

            // pred is already reached goal, so change pred to appropriate node
            predOfCorners[ii] = int32(-1)
            corners[ii] = predOfPredId

            sl.pm.UnpinPage(curr.GetPageId(), false)
                        UnpinPage(pred.GetPageId(), false)

                        // go backward for gathering appropriate corner nodes info
            pred = FetchAndCastToNode(sl.pm, predOfPredId)
        } else { //[4]
            UnpinPage(curr.GetPageId(), false)
            predOfCorners[ii] = predOfPredId,
            corners[ii] = pred.GetPageId()
        }
    }

    return true, pred, predOfCorners, corners
}
```

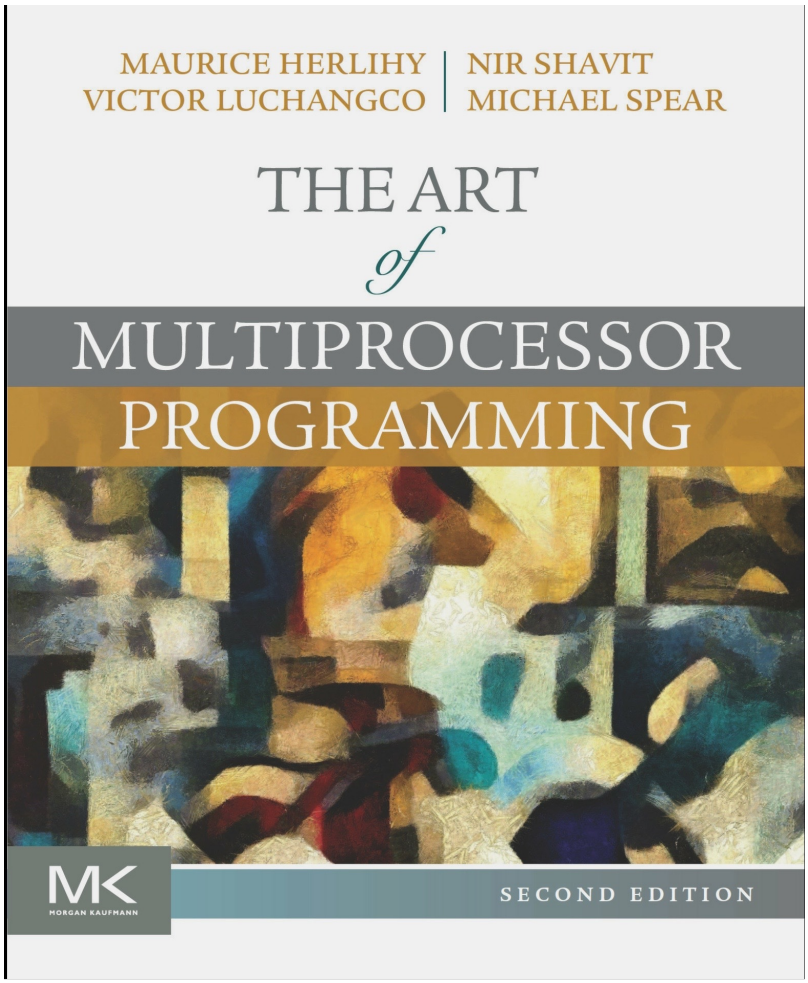
- [1] Loop for traversing from the level of the upper limit (MAX_FOWARD_LIST_LEN) in the overall Skip List level toward level 1
    - In the author's implementation, the upper limit was set to 20
        - The probability **P** in the function that determines the level of a node is set to 0.5
    - In the on-memory implementation, the "highest level" of the entire Skip List was maintained and updated, and the search was started at that level. However, when concurrent access is supported, it is difficult to always update the "highest level" appropriately and make it available for reference. Therefore, the search is started from the upper level, even though wasteful processing runs in sequential execution
- [2] A loop that linearly searches for nodes to be used for transfer at level ii
    - The keys to be searched are compared with the minimum key of each node. Since the entries are designed to be kept in ascending order, when a node with a minimum key greater than the key to be searched for is encountered, a node one before the node is selected for transafer, and the loop ends
        - If reached a node which is too far, a node before the node is a node should be used to transfer to the next level
- [3] Consideration when node deletion is found to occur
    - Processing to be performed after discovering a transfer node
    - The route to be taken if the operation to be performed is "remove", the level traversed is greater than 1, the number of entries held by the node to transfer transfer at is 1, and the key of the entry matches the key to be searched for
        - This route is taken into account in cases where node deletion is to be performed as a result, and the node to be searched for has been reached at a level above 1
        - When deleting a node, the connection relationship between nodes is updated, but if processing continues without consideration in this case, location transfering continues at the node to be deleted until reaching level 1 and nodes which should be updated connection relationship are not stored in the **predOfCornes** list appropriately. this is problem
        - Therefore, the route in [3] changes the node at which location transfers at the current level to the previous node, so that the above problem does not occur
- [4] Processing at the end of one loop in the normal case (the same thing is done in [3])
    - UnpinPage the node **curr** that has gone too far, since it will no longer be accessed
    - Store the page ID of the transfer node in **corners** and the page ID of the node before the transfer node in **predOfCorners**
- Subsequent operations after finding target node
    - Get
        - Basically, it only calls the I/F of the entry acquisition provided by the node indicated by the return value **foundNode** and returns the result
            - Search within a node by binary search (similar for other operations)
            - There may be cases where matching entry does not exist

        - Finally, PageManager::UnpinPage is called with the node indicated by **foundNode** as the argument (dirty flag=false)
- Insert
    - Basically, it just calls the I/F for adding an entry provided by the node indicated by the **foundNode** in the return value and returns the result
    - In most cases, the entry is inserted at the appropriate position in the node and the process is completed, but if there is not enough free space in the node, a split is performed
    - split
        - Creates a new node (page) with PageaManager::NewPage
            - The level of the new node is the value obtained after the call of NewPage using the function to determine the level (please see code of on-memory version)
            - The new node is assumed to be the next (adjacent) node (at level 1) of the node that was originally to be added (in this case, author calls it "parent node")
        - Updating node connectivity
            - Update the connection relationship up to the level of the new node
            - At level 1, the node to which the parent node is connected is the new node, and the node to which the new node is connected is the node to which the parent node was connected
            - For other levels, it is sufficient to do the same by referring to the transfer nodes which are stored in the return value of FindNode, **corners**_
                - Since it is the page ID that is stored, PageManager::FetchPage is used to retrieve the node, and PageManager::UnpinPage is called (dirty flag=true) when the access, including updating the connection destination, is complete
        - Moving entries
            - Basically, move the back half of the entries existing in the parent node to the new node
                - The entry to be moved is determined so that it is as close as possible to half the size of the data area
            - If moving half of the entry does not fit the entry to be added, calculate and move the entry accordingly
        - Adding entries
            - Add an entry to the parent node or the new node, whichever is appropriate
        - Call PageManager::UnpinPage with the new node as argument (dirty flag=true)
    - Finally, call PageManager::UnpinPage with the parent node as argument (dirty flag=true)
- Remove
    - Basically, the I/F of entry removal provided by the node indicated by the return value **foundNode** is called, and the result is the return value.
        - There are cases where the entry specified by Key does not exist
    - In most cases, the entry corresponding to the specified Key is deleted to complete the process, but if the number of entries in the node becomes zero as a result of the deletion, node deletion is performed
    - Node deletion
        - Node connection relationship update

                - For deletion, processes the node connected to the target node up to the height of the level of the target. processing changes next node setting to nodes to which target node connected
                - What is done is generally the same as when a split occurs in Insert, except that the node corresponding to the parent node is the node whose page ID is stored in index 0 of FindNode's return value **predOfCorners_**
            - Call PageManager::UnpinPage with the node to be deleted as an argument (dirty flag=true)
            - Deletion of the page used by the node
                - Call PageManager::DeallocatePage
        - If node deletion did not occur, PageManager::UnpinPage is called with the node indicated by **foundNode** that has finished accessing (dirty flag=true)
    - Iterator
        - Cases in which FindNode is called are at least the cases in which a starting point is specified
        - Generate SLItr type using the resulting entries
            - Iteration can be done by traversing the next node at level 1
            - Based on this, the SLItr type can be designed as you desired
                - However, it is necessary to consider whether or not it will work according to the specification if an operation such as updating the Skip List is performed before the iterator returns all entries in the specified range, even if the iterator is used on not concurrent execution
        - Notes
            - In SLItr type implementations, it should be considered not to return a temporary starting point if there is no entry that matches the key specified as the starting point of the range

A method to make Skip List concurrently accessible

- THE ART of MULTIPROCESSOR PROGRAMMING - SECOND EDITION" by Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear
    - Commonly known as the TAoMP book. One of the bible of multi thread programming



- In chapter14 section3 "A lock-based concurrent skiplist" of this book, there is an example of how to construct a concurrent skip list
    - This method was originally proposed in the following paper
        - M. Herlihy, Y. Lev, V. Luchangco, N. Shavit, A provably correct scalable skiplist (brief announcement), Proc. of the 10th International Conference on Principles of Distributed Systems. OPODIS 2006. 2006.
- The following is an example implementation (in Java) of the code shown in the book

```
1  public final class LazySkipList<T> {
2    static final int MAX_LEVEL = ...;
3    final Node<T> head = new Node<T>(Integer.MIN_VALUE);
4    final Node<T> tail = new Node<T>(Integer.MAX_VALUE);
5    public LazySkipList() {
6      for (int i = 0; i < head.next.length; i++) {
7        head.next[i] = tail;
8      }
9    }
10   ...
11   private static final class Node<T> {
12     final Lock lock = new ReentrantLock();
13     final T item;
14     final int key;
15     final Node<T>[] next;
16     volatile boolean marked = false;
17     volatile boolean fullyLinked = false;
18     private int topLevel;
19     public Node(int key) { // sentinel node constructor
20       this.item = null;
21       this.key = key;
22       next = new Node[MAX_LEVEL + 1];
23       topLevel = MAX_LEVEL;
24     }
25     public Node(T x, int height) {
26       item = x;
27       key = x.hashCode();
28       next = new Node[height + 1];
29       topLevel = height;
30     }
31     public void lock() {
32       lock.lock();
33     }
34     public void unlock() {
35       lock.unlock();
36     }
37   }
38 }
```

```
39   int find(T x, Node<T>[] preds, Node<T>[] succs) {
40     int key = x.hashCode();
41     int lFound = -1;
42     Node<T> pred = head;
43     for (int level = MAX_LEVEL; level >= 0; level--) {
44       volatile Node<T> curr = pred.next[level];
45       while (key > curr.key) {
46         pred = curr; curr = pred.next[level];
47       }
48       if (lFound == -1 && key == curr.key) {
49         lFound = level;
50       }
51       preds[level] = pred;
52       succs[level] = curr;
53     }
54     return lFound;
55   }
```

```
56    boolean add(T x) {
57      int topLevel = randomLevel();
58      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
59      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
60      while (true) {
61        int lFound = find(x, preds, succs);
62        if (lFound != -1) {
63          Node<T> nodeFound = succs[lFound];
64          if (!nodeFound.marked) {
65            while (!nodeFound.fullyLinked) {}
66            return false;
67          }
68          continue;
69        }
70        int highestLocked = -1;
71        try {
72          Node<T> pred, succ;
73          boolean valid = true;
74          for (int level = 0; valid && (level <= topLevel); level++) {
75            pred = preds[level];
76            succ = succs[level];
77            pred.lock.lock();
78            highestLocked = level;
79            valid = !pred.marked && !succ.marked && pred.next[level]==succ;
80          }
81          if (!valid) continue;
82          Node<T> newNode = new Node(x, topLevel);
83          for (int level = 0; level <= topLevel; level++)
84            newNode.next[level] = succs[level];
85          for (int level = 0; level <= topLevel; level++)
86            preds[level].next[level] = newNode;
87          newNode.fullyLinked = true; // successful add linearization point
88          return true;
89        } finally {
90          for (int level = 0; level <= highestLocked; level++)
91            preds[level].unlock();
92        }
93      }
94    }
```

```
95    boolean remove(T x) {
96      Node<T> victim = null; boolean isMarked = false; int topLevel = -1;
97      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
98      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
99      while (true) {
100       int lFound = find(x, preds, succs);
101       if (lFound != -1) victim = succs[lFound];
102       if (isMarked ||
103            (lFound != -1 &&
104            (victim.fullyLinked
105            && victim.topLevel == lFound
106            && !victim.marked))) {
107         if (!isMarked) {
108           topLevel = victim.topLevel;
109           victim.lock.lock();
110           if (victim.marked) {
111             victim.lock.unlock();
112             return false;
113           }
114           victim.marked = true;
115           isMarked = true;
116         }
117         int highestLocked = -1;
118         try {
119           Node<T> pred, succ; boolean valid = true;
120           for (int level = 0; valid && (level <= topLevel); level++) {
121             pred = preds[level];
122             pred.lock.lock();
123             highestLocked = level;
124             valid = !pred.marked && pred.next[level]==victim;
125           }
126           if (!valid) continue;
127           for (int level = topLevel; level >= 0; level--) {
128             preds[level].next[level] = victim.next[level];
129           }
130           victim.lock.unlock();
131           return true;
132         } finally {
133           for (int i = 0; i <= highestLocked; i++) {
134             preds[i].unlock();
135           }
136         }
137       } else return false;
138     }
139   }
```

```
140   boolean contains(T x) {
141       Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
142       Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
143       int lFound = find(x, preds, succs);
144       return (lFound != -1
145            && succs[lFound].fullyLinked
146            && !succs[lFound].marked);
147    }
```

- The prefix "lazy" is not limited to Skip List, but is used to refer to exclusive control methods such as the type in this example, and should be taken to mean "delayed"
- In the book, the keyword "optimistic" is also used, but it seems more appropriate in the context of exclusive control methods
- Although a detailed explanation of the implementation example is omitted, the point to keep in mind is the method of exclusive control, which is realized in the following manner
    - The locks on the nodes to be updated are not held in advance. These are obtained only when a node needs to be added or deleted

        - The target nodes here are all the nodes corresponding to the stations that were transfered at during the process of traversing (to use the analogy in past section)
    - The nodes to be updated are locked one by one, and checked to see if they have been updated by other threads
    - If a node is found that has been updated during the process, all locks acquired are released and the process is canceled. Then traversing for desired operation is retried from start
    - If none of the nodes have been updated, the desired update process (node addition/deletion) is performed while retaining all locks acquired
- Simply put, if this method is applied to an extended Skip List (not supported concurrent access = sequential version) in which each node holds multiple entries, it can support concurrent access
    - (Note that the implementation example cited above has only one entry per node)

Apply the concurrent access method described in the TAoMP book to the multiple-entry per node version

- This section focuses on the processing required to support concurrent access for the sequential version and points to be noted when supporting concurrent access
- Fundamentals of exclusive control design
    - Methods of exclusive control
        - Read-write locking is used.
            - Mutexes that are reentrant (the same thread does not block even if it acquires the same lock multiple times without releasing it) should not be used, as performance will be degraded
    - Order in which locks are acquired
        - To avoid deadlocks, the order in which locks are acquired should be the same among threads
            - For example, if the mutex is A, B, and C, and the order is A->B->C, A->C is OK, but B->A, C->B is not
            - In Skip List, the nodes are connected from the head node to the last node, so you only need to follow that order
    - Lock retention
        - After starting access to the Skip List, always hold the lock of at least one node, except when split and some processes for node deletion are being performed, and release the lock after the access is completed. When retrying (see below), all locks are released also
- Node search process
    - The code contains comments [number], each of which is explained below.
    - At the end of FindNode, if the return value **isSuccess** is true, the thread has acquired the R or W-lock of the returned **foundNode** Also, the pin count of the node is increased by +1
        - W-lock for update operations, R-lock for reference operations
        - In the code, the term "latch" is used instead of "lock" as is customary in the world of database systems

```
type SkipListCornerInfo struct { // [1]
    PageId int32
    UpdateCounter int32
}
```

```
// utility function to cast a pointer of type Page to a pointer of type Node
func FetchAndCastToNode(pm *PageManager, pageId int32) *Node {
    fetchedPage := pm.FetchPage(pageId)
    return (*Node)(unsafe.Pointer(fetchedPage))
}

// Utility function to acquire and release the latch (≒lock) of a node according
to its argument
func latchOpWithOpType(node *Node, getOrUnlatch LatchOpCase, opType
SkipListOpType) {
    switch opType {
    case SKIP_LIST_OP_GET:
        if getOrUnlatch == SKIP_LIST_UTIL_GET_LATCH {
            node.RLatch()
        } else if getOrUnlatch == SKIP_LIST_UTIL_UNLATCH {
            node.RUnlatch()
        }
    default: // SKIP_LIST_OP_INSERT or SKIP_LIST_OP_REMOVE
        if getOrUnlatch == SKIP_LIST_UTIL_GET_LATCH {
            WLatch()
        } else if getOrUnlatch == SKIP_LIST_UTIL_UNLATCH {
            node.WUnlatch()
        }
    }
}

// Arguments
// key: Key of the entry to be searched.
// opType: a constant indicating what the FindNode method was called for. Pass a
constant corresponding to one of GET, INSERT, or REMOVE
// Return value
// isSuccess: Whether the search was successful or not (false indicates that a
retry is necessary)
// foundNode: Pointer to the node found as a result of the search. It only means
that the foundNode node may contain the entry corresponding to key argument.
existence is not guaranteed
// predOfCorners_: page ID of the previous node connected to the node when it was
"switched" at each level. The index is (level - 1)
// corners_: Page ID of the node that was "switched" at each level. The value of
level 1 corresponds to foundNode. The index is (level - 1)
func (sl *SkipList) FindNode(key *KV, opType SkipListOpType) (isSuccess bool,
foundNode *Node, predOfCorners_ []SkipListCornerInfo, corners_ []
SkipListCornerInfo) {
    pred := FetchAndCastToNode(sl.pm, sl.HeadNodeID)
    latchOpWithOpType(pred, SKIP_LIST_UTIL_GET_LATCH, opType)

    predOfPredId := int32(-1)
    predOfPredCounter := int32(-1)
    predOfCorners := make([]SkipListCornerInfo, MAX_FOWARD_LIST_LEN)
    // entry of corners is corner node or target node
    corners := make([]SkipListCornerInfo, MAX_FOWARD_LIST_LEN)
    var curr *Node = nil
    for ii := (MAX_FOWARD_LIST_LEN - 1); ii >= 0; ii-- {
        for { // [2]
```

```
            curr = FetchAndCastToNode(sl.pm, pred.GetForwardEntry(int(ii)))
            latchOpWithOpType(curr, SKIP_LIST_UTIL_GET_LATCH, opType)
            if !curr.GetSmallestKey(key.ValueType()).CompareLessThanOrEqual(*key)
{
                // (ii + 1) level's corner node or target node has been identified
(= pred)
                break
            } else {
                // keep moving foward
                predOfPredId = pred.
                predOfPredCounter = pred.
                UnpinPage(pred.GetPageId(), false)
                latchOpWithOpType(pred, SKIP_LIST_UTIL_UNLATCH, opType)
                pred = curr
            }
        }
        if opType == SKIP_LIST_OP_REMOVE && ii ! = 0 && pred.GetEntryCnt() == 1 &&
key.CompareEquals(pred.GetSmallestKey(key.ValueType())) {
            // [3]

            // pred is already reached goal, so change pred to appropriate node
            predOfCorners[ii] = SkipListCornerInfo{types.InvalidPageID, -1}
            corners[ii] = SkipListCornerInfo{predOfPredId, predOfPredLSN}

            sl.pm.UnpinPage(curr.GetPageId(), false)
            latchOpWithOpType(curr, SKIP_LIST_UTIL_UNLATCH, opType)
            sl.pm.UnpinPage(pred.GetPageId(), false)
            latchOpWithOpType(pred, SKIP_LIST_UTIL_UNLATCH, opType)

            // go backward for gathering appropriate corner nodes info
            pred = FetchAndCastToNode(sl.pm, predOfPredId)
            latchOpWithOpType(pred, SKIP_LIST_UTIL_GET_LATCH, opType)

            // check updating occurred or not
            afterCounter := pred.
            // check update state of beforePred (pred which was pred before
sliding)
            if predOfPredCounter ! = afterCounter {
                // updating exists
                UnpinPage(pred.GetPageId(), false)
                latchOpWithOpType(pred, SKIP_LIST_UTIL_UNLATCH, opType)
                return false, nil, nil, nil, nil
            }
        } else { // [4]
            sl.pm.UnpinPage(curr.GetPageId(), false)
            latchOpWithOpType(curr, SKIP_LIST_UTIL_UNLATCH, opType)
            predOfCorners[ii] = SkipListCornerInfo{predOfPredId,
predOfPredCounter}
            corners[ii] = SkipListCornerInfo{pred.GetPageId(),
pred.GetUpdateCounter()}
        }
    }
```

```
    return true, pred, predOfCorners, corners
}
```

- [1] Structure for storing transfer node information passed in the search process
    - In the sequential version, only the page ID, but in the concurrent access version, the update counter must be stored also
- [2] A loop that linearly searches for nodes at which location transfers at level ii
    - What is done is basically the same as the sequential version
    - However, the difference is that it proceeds while acquiring and releasing locks to support concurrent access
        - If there is no node with a lock, there is a possibility that the search process will not be executed correctly if a node is updated by another thread, so the search always proceeds with one node holding a lock at least
        - However, if the search proceeds while acquiring a new lock without releasing the held lock, other threads' access to the node holding the lock will be interfered with (= lock contention), and the throughput of multiple threads in concurrent access will be reduced, so only the minimum necessary amount of locks are retained and proceed
            - In order to keep locks to the minimum necessary, a new node's lock is acquired while the prior lock is retained, and the prior lock is released
            - The exception is when a node with a newly acquired lock is found to be unnecessary for the continuation of the search process (the case referred to as "too much" in the sequential version), in which case the newly acquired lock is released and the search process continues
        - In the case of an update operation, the node proceeds while acquiring the W-lock, and in the case of a reference operation, the node proceeds while acquiring the R-lock
            - Since it is basically impossible to replace the R-lock with a W-lock, so update operations require a W-lock for the final update target will proceed by acquiring the W-lock, which is an exclusive lock
            - (With some ingenuity, it is possible to proceed with the R-lock at least up to level 1, but this is complicated by the fact that the search process must be retried from the beginning in the event of a failure to switch locks)
        - In [3] and [4], when moving forward, store the page ID of the **pred** in **predOfpredId** and the value of the update counter of the **pred** in **predOfpredCounter**
            - The update counter is a counter managed by the page to check whether the node has been updated or not
            - The update counter is incremented when an update operation is performed on a node
- [3] Consideration when node deletion is found to occur
    - This is basically the same as the sequential version
    - However, since simply moving the current location backward would violate the order in which locks are acquired, additional considerations are needed to avoid the violation
        - First, locks of current **pred** are released, and then a lock of a node which is behind (closer to the first node) in the list is acquired as new **pred**

            - (Newly acquired lock have already been acquired and released once in the past)
        - If the node has been updated, the search may not continue correctly, so checking if the update counter of new **pred** candidate node before releasing lock and the update counter after acquiring the lock again are the same, is needed
        - If the values of the two values do not match, it means that an update has taken place, so the search process is given up, all locks held are released, and the search process is retried from the beginning
            - The retry should be designed to be performed by the caller of FindNode. If the return value **isSuccess** is false, it indicates that a retry is necessary
        - If two values match, the search process continues as is

- [4] Processing at the end of one loop in the normal case (the same thing is done in [3])
    - What is done is basically the same as in the sequential version
    - The page ID of the transfer node is stored in **corners**, and the page ID of the node before the transfer node is stored in **predOfCorners**, as in the sequential version, but since it is necessary for splits and node deletions, the SkipListCornerInfo type is used and the update counter value is also stored

- Subsequent operations after finding target node
    - Get
        - Basically, the I/F of the entry acquisition provided by the node indicated by the return value **foundNode** is called and the result is returned
            - Search within the node by binary search (the same applies to other operations)
            - (The same applies to other operations) There may be cases where no entry exists which has key matches witch specified key
            - When acces is fineshed, the R-lock is released and UnpinPage is called (dirty flag=false)
    - Insert
        - Basically, it only calls the I/F for adding an entry provided by the node indicated by the return value **foundNode** and returns the result as the return value
            - However, unlike the sequential version, there are cases where the operation fails, so an additional return value is required to tell the caller that a retry is necessary
        - split
            - Basically the same as in the sequential version
            - However, when updating connection relations, the lock acquisition order convention is needd to be cosindered, so as in the case of node search [3], after releasing all locks held, locks on each node are acquired one by one, checked for updates, and if there are any updates, a retry is performed from node search, and if there are no node which has updates, all checked nodes connection are updated appropriately with all locks retained, and then releases all locks
                - However, if all update checking is OK, the lock of the parent node must not be released until the insert process is completed
                - Similarly, calling UnpinPage is also not allowed, since the total pin count of parent node is +2 at the end of the insert process
            - Notes
                - Remember that the parent node must also be checked for updates

            - A new node should be locked immediately after creation, and should not be released until the Insert process is finished. Do not forget to UnpinPage when releasing the lock also
            - Although it is necessary to release all the locks held once, the pin of the parent node must be left up, otherwise cache out and in will occur, and the address of the reference obtained by fetching will change, which will cause trouble
                - And if you forget to drop the pin you left up with UnpinPage, it will also cause problem
            - Don't forget to increment the update counter and UnpinPage (dirty flag=true) the node that has been updated connection
                - Don't forget about parent nodes and new nodes also
            - **corners_** may contains same node at different levels
    - Remove
        - Basically, what you do is the same as in the sequential version
        - If you need to delete a node and update the connection, you can do so in the same way as described in the split section above
        - Notes
            - As in the sequential version, the page ID that need to be reconnected at level 1 is stored in the index 0 of **predOfCorners_** and do not forget to check for updates of it
            - The node to be deleted also should be checked updating
                - The node ID is stored in index 0 of **corners_**, so we can use it
            - The same node may be set at different levels in **corners_**
    - Iterator
        - Basically the same as in the sequential version
        - Generate SLItr type using the entries obtained
            - When obtaining an entry, do not release the R-lock of the node where the entry was stored
                - If you release the R-lock, other threads may remove the entry, or the addition of the entry may cause a split, which may change the location of the entry
            - Although this is a difficult design issue, we think it is safe to retrieve entries from the entries obtained by FindNode in the Skip List when generating the SLItr type, up to the specified range, and retain the results
                - With this design, it is guaranteed that only entries that exist within the specified range will be returned in ascending order
                    - However, as mentioned above, the retrieval of entries here is not atomic to other operations, so it is not a snapshot of the moment when SkipList::Iterator is called
                - The process is to traverse the consecutive nodes at level 1, repeatedly acquiring and releasing the R-lock.
                - If no special effort is made, the entries collected will be retained in memory, which may result in unacceptable memory consumption, but this concern can be avoided by using the PM or page mechanism or other methods, such as writing to disk and reading back when the data is needed
        - Notes

            - If there is no entry that matches the key specified as the starting point of the range, it is necessary to consider not to return an entry that is a temporary starting point
                - Depends on the implementation of the Node type

[Important] Logic bugs exist on above explanation

- Lacks of consideration in the processing of update checks
    - As a matter of fact, if the implementation follows the code example and explanation above, if another thread deletes a node aside and it is unluckily a node that is subject to [3] or the update check for split and node deletion, null will be returned by FetchPage. Also, if there are threads waiting to acquire a lock of a node to be deleted, the threads that acquires the lock after the node deletion is completed may cause problem
        - (In the author's implementation in SamehadaDB, since PageManager::DeallocatePage is not called, the node is logically deleted, but the used page is not reused, and when FetchPage is done, the page with the original contents is returned. So there is no problem...)
    - When DeallocatePage is called as described, if null is returned in FetchPage at the time of the update check, the check result should be set to NG, and when performing various operations on a node, it should be checked whether the node has been deleted (= whether accessed page data content corresponds to one of atttempted node should have)
- Mistake in timing of page lock releases
    - In the above code examples and explanations, the page lock is released after UnpinPage is called, but originally, after UnpinPage is called, it is not guaranteed that the reference to the page is valid
    - Therefore, lock release must be performed before UnpinPage
        - (The implementation in SamehadaDB is NG code. However, due to the PM implementation and the fact that Go is a GC language, even if the page is cached out, the instance of the Page type exists in memory until GC is run, and the reference is holded by mutex type variable which should be lock released, so the reference will not be GC'ed. Therefore there is no problem...)

[Reference] Author's implementation

- Although there are some differences from the design described in this document, an implementation exists in RDB (SamehadaDB, made in Go language), which is being developed mainly by the author
    - Source files corresponding to the Skip List implementation and the main related source files
        - skip_list.go
        - skip_list_header_page.go
        - skip_list_block_page.go
        - page.go
        - skip_list_iterator.go
        - skip_list_test.go
    - Code tree of the whole RDB
- The explanations in this document may be insufficient, but even in that case, if you read the source code listed above based on the contents of this document, I hope that you will be able to create something similar (regardless of the development language)

Finally

- If you find any bugs or misses in the design or in the code examples, please feel free to point them out!
- I can't deny that I rushed through the important parts, so if you have any questions, please ask in the GitHub issue and I'll try my best to respond to them

Disclaimer

- Author of this document doesn't guarantee that there are no errors (i.e., omissions) in the shared designs, etc., and they should be taken as reference only
    - The implementation based on the design described in this document has been carefully tested and works without any defects, but this does not guarantee that the design is error-free
- This document is provided on the condition that the reader agrees that the author shall not be held liable for any damages incurred by the reader or any other third party's any action in reference to or in use of the contents of this document