Skip List

Author Name:

Date: 2018-06-09

Chapter 1

Skip list is a data structure that supports both searching and insertion in $O(\log N)$ expected time. The operation is very fast by maintaining a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one.

Chapter 2

Node:

Every node has a key value for sort, a value for store, and a sequence to store the node forward. Then it has a level value to show the level.

```
typedef struct NodeStruct* Node;
struct NodeStruct{
    int key;
    int value;
    Node *forward;  //use array to save level-pointers
    int level;
};
```

Skiplist:

Every list has a head node, and set a level value to show the level.

```
typedef struct ListStruct* List;
struct ListStruct{
    int level;  //the highest level
    Node head;
};
```

Initialize:

An element NIL is given a key greater than any legal key. All levels of all skip lists are terminated with NIL.A new list is initialized so that the level of the list is equal to 1 and all forward pointers of the list's header point to NIL.

```
/* create a new node and make space for it */
Node MakeNode(int level) {
    Node node = (Node)malloc(sizeof(struct NodeStruct));
    node->forward = (Node *)malloc(sizeof(Node) * level);
    node->level = level;
    return node;
}

/* the establishment of skip list */
List MakeList() {
    List list = (List)malloc(sizeof(struct ListStruct));
    list->level = 0;
    list->head = MakeNode(MAXLEVEL);
    list->head->key = -1;
    int i;
    for (i = 0; i < MAXLEVEL; i++) {
        list->head->forward[i] = NULL;  //set pointers to null for each level
    }
    return list;
}
```

Search:

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

```
Search(list, searchKey)
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto1 do
        whilex→forward[i]→key < searchKey do
            x := x→forward[i]
    -- x→key < searchKey ≤x→forward[1]→key
    x := x→forward[1]
    if x→key = searchKey then returnx→value
    else return failure
```

Insert&Delete:

To insert or delete a node, we simply search and splice. A vector update is maintained so that when the search is complete (and we are ready to perform the splice), update[i] contains a pointer to the rightmost node of level i or higher that is to the left of the location of the insertion/deletion. If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After each deletion, we check if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

```
Insert(list, searchKey, newValue)
    localupdate[1..MaxLevel]
    x := list→header
    for i := list→level down to 1 do
        whilex→forward[i]→key < searchKey do
            x := x→forward[i]
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then x→value := newValue
    else
        newLevel := randomLevel()
        if newLevel > list→level then
            for i := list→level + 1 to newLevel do
                update[i] := list→header
            list→level := newLevel
        x := makeNode(newLevel , searchKey, value)
        for i := 1 tonewLevel do
            x→forward[i] := update[i]→forward[i]
            update[i]→forward[i] := x
```

```
Delete(list, searchKey)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level down to 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then
        for i := 1 to list→level do
            if update[i]→forward[i] ≠ x then break
            update[i]→forward[i] := x→forward[i]
        free(x)
        while list→level > 1 and list→header→forward[list→level] = NIL do
            list→level := list→level - 1
```

Chapter3:

大小	插入	删除	查找
1	0.584912s	0.383294s	0.145671s
2	0.621290s	0.400217s	0.172841s
3	0.659595s	0.422581s	0.181163s
4	0.641688s	0.434664s	0.188028s
5	0.665350s	0.457296s	0.201493s
6	0.684668s	0.464894s	0.214273s
7	0.685828s	0.488838s	0.226907s
8	0.698543s	0.490103s	0.233664s
9	0.704472s	0.473960s	0.257718s
10	0.738861s	0.553054s	0.280745s

Chapter4:

As we all know, the redis database use skiplist instead of b-trees, there are few reasons :

- They are not very memory intensive. It's up to you basically. Changing parameters about the probability of a node to have a given number of levels will make then *less* memory intensive than b-trees.
- A sorted set is often target of many ZRANGE or ZREVRANGE operations, that is, traversing the skip list as a linked list. With this operation the cache locality of skip lists is at least as good as with other kind of balanced trees.

  From the test result, we can clearly find that the skiplist search and insert faster than normal linked list. For detailed, the time complexity is O(logn). However, the delete operation is same with normal linked list. And with more and more node being deleted, the randomness will reduce.

Appendix:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAXLEVEL 14
#define PRABABILITY 0.5
/* you can change the maxlevel */

/* generate random level */
int Random(int max){
    int r = 1;
    //every level has probability p to ascend upwards
    while(rand() % 100 < 100 * PRABABILITY){
        r++;
        if(r >= max) return max;
    }
    return r;
}

/* list node */
typedef struct NodeStruct* Node;
struct NodeStruct{
    int key;
    int value;
    Node *forward;  //use array to save level-pointers
    int level;
};

/* skip list */
typedef struct ListStruct* List;
struct ListStruct{
    int level;  //the highest level
    Node head;
};

/* create a new node and make space for it */
```

```
Node MakeNode(int level) {
    Node node = (Node)malloc(sizeof(struct NodeStruct));
    node->forward = (Node *)malloc(sizeof(Node) * level);
    node->level = level;
    return node;
}

/* the establishment of skip list */
List MakeList() {
    List list = (List)malloc(sizeof(struct ListStruct));
    list->level = 0;
    list->head = MakeNode(MAXLEVEL);
    list->head->key = -1;
    int i;
    for (i = 0; i < MAXLEVEL; i++) {
        list->head->forward[i] = NULL;  //set pointers to null for each level
    }
    return list;
}

/* the search operation of skip list */
Node Search(int x, List list){
    Node p = list->head, next = NULL;
    int i;
    //nested loop to search the key
    for(i = list->level - 1; i >= 0; i--){
        //go ahead to reach the closest node to x in the i-th level
        for(next = p->forward[i];
            next && next->key < x;
            p = next, next = p->forward[i]);
        //if the next step is exactly x, return
        if(next && next->key == x) return next;
    }
    return NULL;  //return not found
}

/* the insertion of skip list */
int Insert(int x, List list) {
    Node update[MAXLEVEL];//keep the nodes whose pointers are likely to be changed in the
insertion
    Node p = list->head;
    Node next = NULL;
    int i;
    //nested loop to search the key, same as search()
    for(i = list->level - 1; i >= 0; i--){
        for(next = p->forward[i];
            next && next->key < x;
            p = next, next = p->forward[i]);
        update[i] = p;
    }
    //the keys should be distinct
    if(next && next->key == x) return -1;
    int level = Random(list->level + 1);    //random a level
    if(level > MAXLEVEL) level = MAXLEVEL;  //can't be higher than maxlevel

    //if it is higher than present level of the list, head pointers should also be updated
    if(level > list->level) {
        update[list->level] = list->head;
        list->level = level;
    }

    //new node
    Node node = MakeNode(level);
    node->key = x;

    //update the list of the closest nodes to x in each level, which is below x's level
    for(i = 0; i < node->level; i++){
        node->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = node;
    }
    return 0;
}

/* this insertion will not really insert the node into the list */
int FictionInsert(int x, List list) {
    Node update[MAXLEVEL];  //keep the nodes whose pointers are likely to be changed in the
insertion
    Node p = list->head;
    Node next = NULL;
    int i;
    Node fiction = MakeNode(MAXLEVEL);  //fake node
    //nested loop to search the key, same as search()
    for(i = list->level - 1; i >= 0; i--){
        for(next = p->forward[i];
            next && next->key < x;
            p = next, next = p->forward[i]);
        update[i] = fiction;
    }
    //the keys should be distinct
    if(next && next->key == x) return -1;

    int level = Random(list->level + 1);    //random a level
    if(level > MAXLEVEL) level = MAXLEVEL;  //can't be higher than maxlevel

    //if it is higher than present level of the list, head pointers should also be updated
    if(level > list->level) {
        update[list->level] = fiction;
        list->level = list->level;
    }

    //new node
    Node node = MakeNode(level);
    node->key = x;

    //update the list of the closest nodes to x in each level, which is below x's level
    for(i = 0; i < node->level; i++){
        node->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = node;
    }
    free(node);
    free(fiction);
    return 0;
}

/* delete node x from skip list */
int Delete(int x, List list) {
    Node update[MAXLEVEL];
    Node p = list->head;
    Node next = NULL;
    int i;
    //nested loop to search the key, same as insert()
    for(i = list->level - 1; i >= 0; i--){
        for(next = p->forward[i];
            next && next->key < x;
            p = next, next = p->forward[i]);
        update[i] = p;
    }
    //if x is not in the list, return error
    if(!next || next->key != x) return -1;

    //next is the exactly node to delete, update the pointers in each level, which is below x's level
    for(i = 0; i < next->level; i++)
        update[i]->forward[i] = next->forward[i];

    //update the highest level of the list
    if(next->level > list->level) list->level = next->level;

    free(next);  //delete the node x

    return 0;
}

/* delete node x from skip list */
int FictionDelete(int x, List list) {
    Node update[MAXLEVEL];
    Node p = list->head;
    Node next = NULL;
    int i;
    Node fiction = MakeNode(MAXLEVEL);
    //nested loop to search the key, same as insert()
    for(i = list->level - 1; i >= 0; i--){
        for(next = p->forward[i];
            next && next->key < x;
            p = next, next = p->forward[i]);
        update[i] = fiction;
    }
    //if x is not in the list, return error
```

```c
    if(!next || next->key != x) return -1;

    //next is the exactly node to delete, update the pointers in each level, which is below x's level
    for(i = 0; i < next->level; i++)
        update[i]->forward[i] = next->forward[i];

    //update the highest level of the list
    if(next->level > list->level) list->level = list->level;

    free(fiction);  //delete the node x

    return 0;
}

/* print the whole list, with its level-infomation */
void ShowList(List list){
    Node p = list->head;
    int i;
    printf("%d levels\n", list->level);
    //for each level, print the linked list in order
    for(i = 0; i < list->level; i++){
        p = list->head;
        printf("level %d :", i);
        while(p->forward[i])
        {
            printf("%d ", p->forward[i]->key);
            p = p->forward[i];
        }
        printf("\n");
    }

}

/* return a random number between low and up */
int RandTest(int low, int up){
    return low + rand()%(up - low);
}

void test(int N){
    List list = MakeList(); //empty list
    int i = 0;
    float cpu_time_used;

    printf("Size: %d\n", N);

    clock_t start, end;
    unsigned long sum = 0;
    int val = 0;
    for(i=0; i<N; i++){
        val += 5;
        Insert(val, list);
    }
    start = clock();
    for(i=0; i<1000000; i++){
        val = RandTest(0, N);
        val = val*5 + 1;
        // printf("%lu\n", start);
        FictionInsert(val, list);
    }
    end = clock();
    sum = end - start;
    //print list in level order
    // ShowList(list);
    // printf("%lu\n", sum);
    cpu_time_used = ((double) sum)/CLOCKS_PER_SEC;
    printf("Insert: %lf\n", cpu_time_used);

    start = clock();
    for(i=0; i<1000000; i++){
        val = RandTest(0, N);
        val = val*5;
        // printf("%lu\n", start);
        FictionDelete(val, list);
    }
    end = clock();
    sum = end - start;
    //print list in level order
    // ShowList(list);
    // printf("%lu\n", sum);
    cpu_time_used = ((double) sum)/CLOCKS_PER_SEC;
    printf("Delete: %lf\n", cpu_time_used);
    printf("-----------------\n");
}

int main(){
    srand(time(NULL)); //set rand seed

    int sizes[10] = {1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};
    // int sizes[19] = {5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000, 55000, 60000, 65000, 70000, 75000, 80000};
    for(int i=0; i<10; i++){
        test(sizes[i]);
    }
}
```

References:

[1] Wikipedia, The Free Encyclopedia. Skip list

[2] "战辉",CSDN, 浅析SkipList跳跃表原理及代码实现

Author List:

Declaration

We hereby declare that all the work done in this project titled "XXX" is of our independent effort as a group.

Signatures: