Algorithimcs
Project : Skip-list

Kacper PLUTA – kacper.pluta@esiee.fr

Abstract

The goal of this project is to implement a skip list and a basic operations which allow to manipulate this data structure.

Contents

Rules of game

1. Students are allowed to work in pairs only if they will work using *pair programming* technique (`https://en.wikipedia.org/wiki/Pair_programming`).
2. Project have to be submitted via `https://elearning.univ-mlv.fr` before the deadline.
3. Project has to compile on GNU/Linux operating system.
4. Source code have to be written in accordance with "Rules of writing good code" (see English version of TD4).
5. Students have to ask about appointment to present their work. During a presentation an additional questions may appear.
6. Any evidence of cheating can cause 0 points and will be reported.
7. Any additional work have to be consulted before, and can lead to additional points.

Introduction to skip list

In the TD3 we have discovered a linked list (see Figure 1) for which the cost of the basic operations like *delete*, *insert* is related to the cost of *search* operation – we have to find a place in the list to insert a new element or find an element to remove.

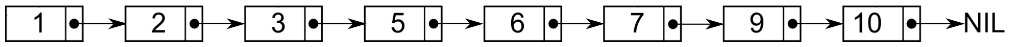

Figure 1: A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to the NIL value used to signify the end of the list.

If the list is stored in sorted order and every node of the list also has a pointers to some number of nodes ahead it in the list we can significantly improve cost of *search* by somehow skipping some elements in the list.
A skip-list [1] is implemented as a hierarchical set of linked lists. Thus, its structure is built up in layers. The bottom layer ($h = 0$) is a traditional ordered linked list, while each list in the upper layer ($h + k$) contains a partial copy of the list in the lower layer ($h + k - 1$). Figure 2 presents a schematic picture of the skip list. Note that, if we define a skip list in a “rigid” way by fixing the number of pointers to nodes ahead of a

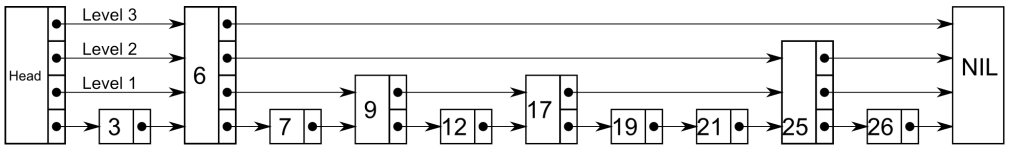

Figure 2: A schematic picture of the skip list.

given node in the list, then the skip list will be very hard to maintain. For instance, if each node will have a pointer to the node two ahead of it in the list, and every fourth node will have a pointer to a node four ahead of it in the list, etc., then operations like delete or insert would not be trivial. Therefore this restriction is relaxed by choosing a random number of levels for each node. This allows us to perform insertion or deletion by performing only a local modification of a skip list.

Skip list algorithms

Initialization

To create a skip list we just create a node where the special key—*head* represents a key bigger than any other legal key, and all levels are terminated with NIL. Note that *current level* of initialized skip list is equal to 0. See Figure 3.

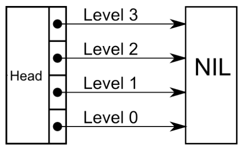

Figure 3: Skip list after initialization.

Search

The search function begins in the uppermost level. If the next node is NIL or its key is higher than the searched key it is necessary to go down a level and horizontally continue. When we cannot make any progress at level 0, that means we are in the front of the node that contains the desired key – if this element exists in the list. Pseudo-code of search is presented in Algorithm 1 and Figure 4 shows steps of Algorithm 1 for a several different keys.

Algorithm 1: Skip list search algorithm.

```
Data: list – a pointer to the first element of the skip list, key – searched key
Result: pointer to the node which contains searched key, NIL otherwise
x = list→header;
for i=currentHighestLevel ( list ) to 0 do
    while x→forward[i]→key < key do
        x = x→forward[i];
    end
end
x = x→forward[0];
if x→key = key then
    return x;
else
    return NIL;
end
```

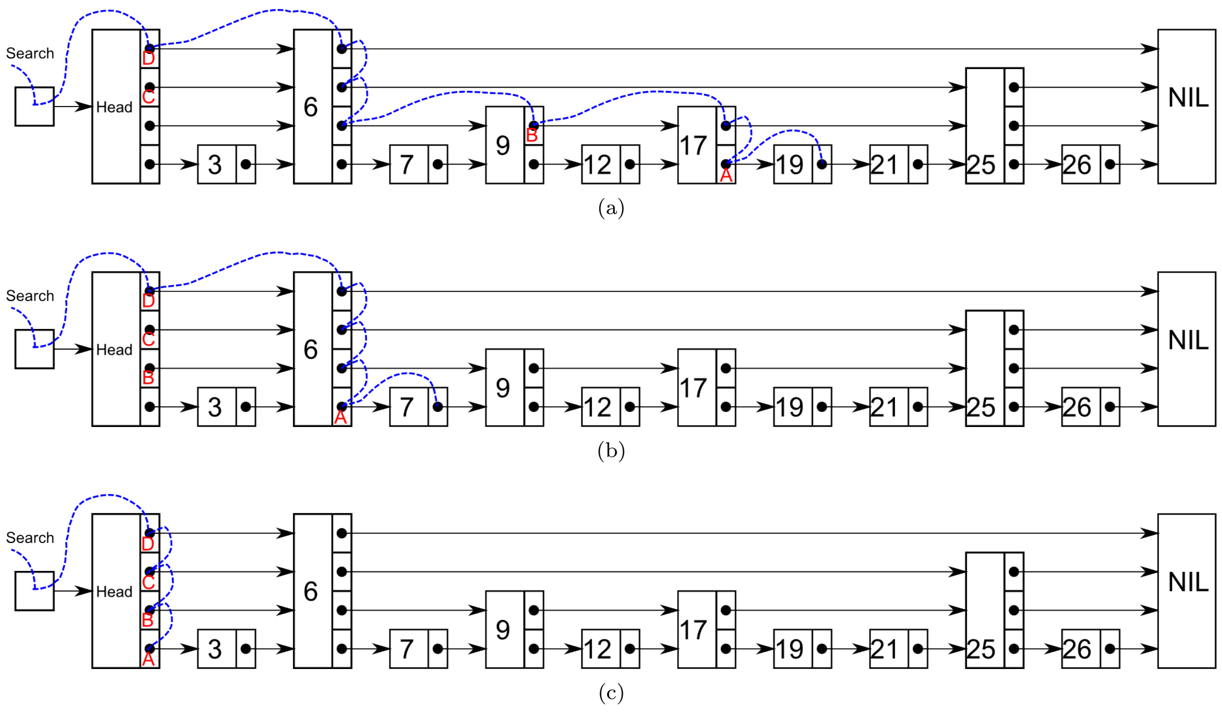

Figure 4: Paths of skip list search algorithm for different keys: (a) 21, (b) 9 and (c) 3.

Insert

The core of insert is search, because to add or remove a node, firstly we have to find its position in the skip list, see Algorithm 2. As was told before, a level of each node is chosen randomly according to some probability distribution p (see Algorithm 3). Usually, we consider $p = \frac{1}{2}$ or $p = \frac{1}{4}$ *i.e.*, half or one-fourth of nodes of level i containing pointers to level $i+1$. Notice that, levels of nodes are generated without reference to the number of elements in the list, where the maximal number of levels of skip list can be calculated in the terms of the number of keys and a given probability distribution—usually $\log_{\frac{1}{p}} n$. For instance, for 2^{16} keys and $p = \frac{1}{2}$ we have the maximal number of levels equal to 16, where for $p = \frac{1}{4}$ we obtain the maximal number of level equal to 8. Figure 5 presents insertion of several new nodes in accordance with Algorithm 2.

Algorithm 2: Skip list insert algorithm.

```
Data: list – a pointer to the first element of the skip list, key – key to insert, value – value of a new
      node
Result: updated input list which contains a new node
x = list→header;
update[0,...,MaxLevel - 1];
for i = currentHighestLevel ( list ) - 1 to 0 do
    while x→forward[i]→key < key do
        x = x→forward[i];
    end
    update[i] = x;
end
x = x→forward[0];
if x→key = key then
    x→value = value;
else
    level = randomLevel();
    if level > currentHighestLevel ( list ) then
        for  i = currentHighestLevel ( list ) to level do
            update[i] = list→header;
        end
        setCurrentHigherLevel ( list, level );
    end
    x = makeNode ( level, key, value );
    for i = 0 to level do
        x→forward[i] = update[i]→forward[i];
        update[i]→forward[i] = x;
    end
end
```

Algorithm 3: Algorithm to calculate a random level.

```
Data: maximal number of levels – maxLevel, p – probability distribution (see text above).
Result: level for new node
level = 0;
while random() < p and level < maxLevel do
    level = level +1;
end
return level;
```

Delete

To delete a node, we have to find its position in the skip list and change pointers of the nodes which pointed on it. Pseudo-code of delete is presented by Algorithm 4 where Figure 6 presents a scheme of deletion of one

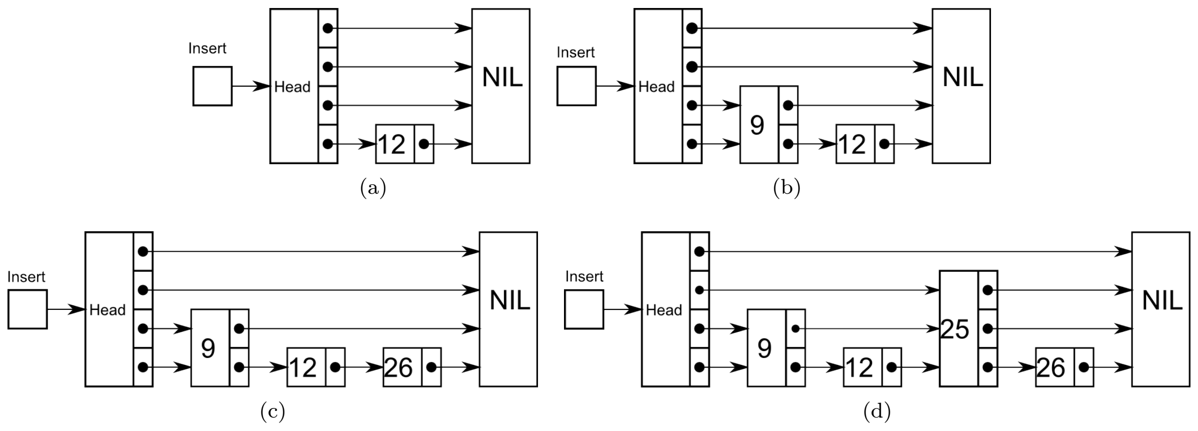

Figure 5: Insertions of a new node with the key: (a) – 12, (b) – 9, (c) 26 and (d) 25.

node.

Algorithm 4: Skip list delete algorithm.

```
Data: list – a pointer to the first element of the skip list, key – key to remove
Result: updated input list which contains one node less
x = list→header;
update[0,...,MaxLevel - 1];
for i = currentHighestLevel ( list ) - 1 to 0 do
    while x→forward[i]→key < key do
        x = x→forward[i];
    end
    update[i] = x;
end
x = x→forward[0];
level = currentHighestLevel ( list );
if x→key = key then
    for i = 0 to level - 1 do
        if update[i]→forward[i] ≠ x then
            break;
        end
        update[i]→forward[i] = x→forward[i];
    end
    free ( x );
    while level > 0 and list→header→forward[level - 1] = NIL do
        setCurrentHigherLevel ( list, level−1 );
        level = level−1;
    end
end
```

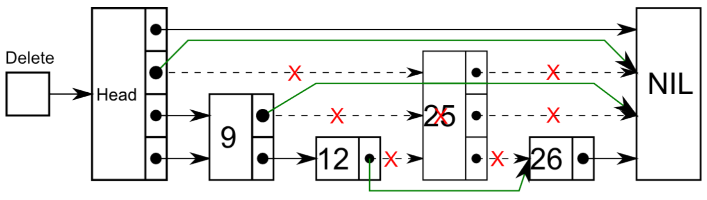

Figure 6: Schematic picture of deletion of node of key 25.

Goals

1. Implementation of the skip list.
2. Implementation of search, insert, delete.
3. Implementation of initializeFromFile.
4. Implementation of printSkipList.
5. Check how different values of p affect efficiency of search, insert, delete.

References

[1] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668676, 1990.