1. Introduction

• Skip Lists is a good alternative to balanced trees such as AVL trees.

• It is simple to implement (comparing with balanced trees)

• It supports many useful operations efficiently: insert, remove, find, min, max, traversal etc

• In general, Skip Lists are sorted linked lists where:

  • nodes in a Skip List may have many 'next' references/pointers
    i.e. a node in a skip list may have number of levels (one next pointer per level)
  • the number of next pointer for a given node is determined probabilistically

• In a sorted linked list, find, insert, and remove need O(n), i.e. scanning all nodes from the beginning until a correct position is found.

  Skip list reduces the cost of scanning nodes by skipping some nodes.
  Expected performance: \( O(\log n) \) per operation.

2. Basic Concepts

• How to accomplish fast searching?
Example 1: Skip every second node in 2nd level

i.e. node 1,3,5,7: each has one next pointer; node 2,4,6,8: each has two next pointers. For node 2, it points to node 4 and 3. Each node has a data (key + some values)

```
header
```

```
1 -- 2 -- 3 -- 4 -- 5 -- 6 -- 7 -- 8
```

Example 2: Skip every second (2nd level) and fourth node (3rd level)

i.e. node 1,3,5,7: each has one next pointer; node 2,6: each has two next pointers and node 4,8: each has three next pointers. For node 4, it points to node 5, 6 and 8.

```
header
```

```
1 -- 2 -- 3 -- 4 -- 5 -- 6 -- 7 -- 8
```
• Pseudo-code for the find(X) operation.
  
  • start at the highest level of the header node
  • follow the next pointers along this level until the value at the next node is
    >= X or until the end of list at this level (i.e. NULL) is reached
  • switch to the next lower level and continue the search using the same rule
  • eventually, stop at a next pointer at the lowest level of a node.
  • if the next node has the value X, then we found X in the list; otherwise we
    found the correct insertion point for value X

For example 1, if X = 7, visit nodes : 2, 4, 6, 8, 7, a max. of n/2+1 visited nodes

For example 2, if X = 7, visit nodes : 4, 8, 6, 8, 7, a max. of n/4+3 visited nodes

• General case : i levels for 2**i Nodes

  • Suppose a skip list contained 1024 nodes and consider a search in it.
  • In the top level, first node is 512 and have cut the search in half.
  • Go down one level in either the left or right half, e.g. (256 512) or (768 1024), again cutting the remaining search in half.
  • We continue in this manner till we find the correct position in the lowest
    level.

**Analysis** : similar to binary search in an array → the maximum number of
nodes visited is O(log n). Note : for each level, we need to look at most 2
nodes
• Problems and solutions

• This data structure is very efficient for find(), which is an important function for insertion, deletion or retrieving operations.

• However, it has major problems during insertion and deletion operations, i.e. operations require to reorganize the list, this cannot be done in $O(\log n)$.

• Solution: Introduce the probabilistic approach for a Skip List.

A Skip List is built with the same distribution of node sizes using random function, i.e. about same number of nodes with one next pointer, two next pointers, etc, but without the requirement for the rigid pattern of node arrangement.

So, it is no longer necessary to maintain the rigid pattern by moving values around after a deletion or insertion operation.

It is showed that this probabilistic approach of skip lists exhibit **expected performance of $O(\log n)$**.

Note: we will not cover probabilistic analysis for skip lists.
Example:

After deleting node #7

After inserting node #9
3. Implementation of Skip Lists

- Every Skip List has an associated (and fixed) probability \( p \) (e.g. 1/2 or 1/4) that determines the distribution of nodes.

Example: for 1024 nodes and \( p=1/2 \). We expect to have:

- 512 nodes with one level, 256 nodes with two levels,
- 128 nodes with three levels, 64 nodes with four levels, etc

- Maximum levels and number of levels in each node

\( \text{maxLevel is suggested to be } \log_{(1/p)} n, \text{ e.g. } p=1/2, \ n=1024 \rightarrow 10 \)

To accomplish expected node distribution, number of levels in a new node is generated by random function as follows:

```cpp
int SkipList::generateRandomLevel()
{
    int levels = 1;
    while (drand48() < probability)
        levels++;
    return (levels > maxListLevels) ? maxListLevels : levels;
}
```

- A possible Node class.

```cpp
class Node
{
public:
    Node();
    Node(const int levels);
    Node(const keyType & key, const int levels);
    Node(const Node &rhs);
    ~Node();
};
```
// standard operations
keyType &getKey();
int getLevels();
Node *getNextPtr(int level);
void setKey(keyType & key);
void setNextPtr(int level, Node* pnode);
void setLevels(int levels);

private:
int numLevels; // number of next ptrs
keyType keyValue; // may add additional data
vector<Node *> nextPtr;

friend class SkipList;
};

• A possible SkipList class

class SkipList
{

public:

SkipList();
SkipList(int maxLevels, double prob);
SkipList(const SkipList &);
~SkipList();

// insert/search/delete item
// return true if OK, false otherwise.

bool insert(keyType & item);
bool find(keyType & item);
bool remove(keyType & item);

void display();
protected:

// important function for insert/remove/find/retrieve:
// locate the point at which item would be inserted or found
// using a "prevPtrs" node of the given levels.

Node * findPrevPtrs(keyType & item, int levels);

// insert item into skiplist, using a node of the
// specified levels. Return true if successful

bool insert(keyType & item, int levels);

// generate number of levels for a new node randomly
int generateRandomLevel();

int setCurrentMaxLevels(const int levels);
int getCurrentMaxLevels();
Node * getHeadPtr();
• **findPrevPtrs() method**

For *insert()* , *delete()* and *find()* in a linked list, we need to know pointer to previous node. In Skip Lists, we need previous pointer on each level. This method returns a "prevPtrs" node containing all previous pointers.

Example:

```cpp
Node * SkipList::findPrevPtrs(keyType & item, int levels)
{
    Node * p = getHeadPtr();
    Node * prevPtrs = new Node(levels);
    int i;

    for (i = 0; i < levels; i++)
        prevPtrs->setNextPtr(i, p);  // all pointers to header

    return prevPtrs;
}
```

<table>
<thead>
<tr>
<th>level</th>
<th>prev. nodes</th>
<th>prev. nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>header</td>
<td>5</td>
</tr>
</tbody>
</table>

Note: Lowest level is level 0

In the above figure, prevPtrs between nodes:

![Diagram](https://via.placeholder.com/150)

In the above figure, prevPtrs between nodes:

<table>
<thead>
<tr>
<th>level</th>
<th>prev. nodes</th>
<th>prev. nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>header</td>
<td>5</td>
</tr>
</tbody>
</table>
for (i = levels - 1; i >= 0; i--)
{
    while (true)
    {
        if (p->getNextPtr(i) == NULL)
            break;

        keyType data = p->getNextPtr(i)->getKey();

        if (data < item)
            p = p->getNextPtr(i);
        else
            break;
    } // while

    prevPtrs->setNextPtr(i, p); // for
}

return prevPtrs;

• insert() method

• with findPrevPtrs(), an insertion is easy.

    bool SkipList::insert(keyType & item)
    {
        int levels = generateRandomLevel();
        return insert(item, levels);
    }
bool SkipList::insert(keyType & item, int levels)
{
    Node * update = findPrevPtrs(item, levels);

    // make sure the item is not in the list
    if ((update->getNextPtr(0)->getNextPtr(0) != NULL) &&
        (update->getNextPtr(0)->getNextPtr(0)->getKey() == item))
        return false;

    Node * newnode = new Node(item, levels);

    // new node may have more levels than are currently in list
    // If so, adjust high_node_size of list.
    if (levels > getCurrentMaxLevels())
        setCurrentMaxLevels(levels);

    for (int i = 0; i < levels; i++)
    {
        Node * prev = update->getNextPtr(i);
        Node * next = prev->getNextPtr(i);
        newnode->setNextPtr(i, next);
        prev->setNextPtr(i, newnode);
    }

    size++;  
    return true;
}

• remove() and find() methods

Similar to insert() method, use findPrevPtrs() to locate the correct positions.
For remove() : if the item is in the list, remove node and restructure the list
For find() : if the item is in the list, return true