Lab Experiment No. 6

Objective

Implementation and analysis of Skip Lists and comparison with basic data structures.

Theory

A Skip List is a probabilistic data structure that allows fast search, insertion, and deletion operations - similar to balanced trees like AVL or Red-Black trees. It uses multiple levels of linked lists where higher levels act as "express lanes" to skip over many elements, allowing logarithmic average time complexity. The basic idea is to augment a sorted linked list with additional layers. Each layer is a subset of the layer below, where each element appears in a higher layer with some fixed probability. This randomness gives the Skip List its probabilistic balancing.

Key Characteristics:

- Skip lists offer average-case time complexities of O(log n) for search, insert, and delete.
- The structure is easier to implement and maintain than balanced trees.
- Space complexity is O(n log n) in the worst case due to multiple levels.

Algorithm:

1. Search (Locate a Key in Skip List)

1. Initialize a pointer at the top-most level of the header node.
2. While at the current level:
    - Traverse forward as long as the next node's key is less than the target key.
    - If the next key is greater or null, move one level down.
3. Continue the process until you either:
    - Find a node with the target key - return success.
    - Reach level 0 and the key is not found - return failure.

2. Insertion (Insert a New Element)

1. Initialize a path tracker (update[]) to store the last visited node at each level.
2. Starting from the highest level of the header:
    - Traverse forward until the next key is greater than or equal to the target key.
    - Record the last visited node at each level before moving down.
3. At level 0, insert the new key after the appropriate node.
4. Randomly generate a level for the new node using a probabilistic method.
5. For each level up to the generated level:
    - Create forward pointers and adjust the update[] nodes to link to the new node.
6. If the new node exceeds the current maximum level, update the list's level accordingly.

3. Deletion (Remove an Element by Key)

1. Use a similar traversal process as in insertion to track the update array.

2. At level 0, check if the key is present in the list.
    - If not found, return failure.
3. If the node exists:
    - Unlink the node at every level it appears in by updating the forward pointers.
    - Deallocate the node's memory.
4. After deletion, check if the highest level in the skip list has become empty.
    - If yes, reduce the skip list level accordingly.

Code

T-5.1. Implementation using Arrays. Depth-First Search (DFS)

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <climits>
#include <vector>

#define MAX_LEVEL 16

class Node {
public:
   int key, value;
   std::vector<Node*> forward;

   Node(int k, int v, int level) : key(k), value(v),
forward(level + 1, nullptr) {}
};

class SkipList {
private:
   int level;
   Node* header;

   int randomLevel() {
      int lvl = 0;
      while (rand() < RAND_MAX / 2 && lvl <
MAX_LEVEL) lvl++;
      return lvl;
   }

public:
   SkipList() {
      level = 0;
      header = new Node(INT_MIN, 0,
MAX_LEVEL);
   }

   ~SkipList() {
      Node* curr = header->forward[0];
      while (curr) {
         Node* next = curr->forward[0];
    curr = curr->forward[0];

      if (curr && curr->key == key) return curr;
      return nullptr;
   }

 void remove(int key) {
                std::vector<Node*>
update(MAX_LEVEL + 1);
                Node* curr = header;

                for (int i = level; i >= 0; i--) {
                   while (curr->forward[i] &&
curr->forward[i]->key < key) {
                      curr = curr->forward[i];
                   }
                   update[i] = curr;
                }

                curr = curr->forward[0];

                if (curr && curr->key == key) {
                   for (int i = 0; i <= level; i++) {
                      if (update[i]->forward[i] != curr)
break;
                      update[i]->forward[i] =
curr->forward[i];
                   }
                   delete curr;

                   while (level > 0 &&
!header->forward[level]) level--;
                }
             }

             void display() {
                std::cout << "Skip List (Level " << level
<< "):\n";
                for (int i = 0; i <= level; i++) {
                   Node* node = header->forward[i];
```

```
            delete curr;
            curr = next;
        }
        delete header;
    }

    void insert(int key, int value) {
        std::vector<Node*> update(MAX_LEVEL + 1);
        Node* curr = header;

        for (int i = level; i >= 0; i--) {
            while (curr->forward[i] &&
curr->forward[i]->key < key) {
                curr = curr->forward[i];
            }
            update[i] = curr;
        }

        curr = curr->forward[0];

        if (!curr || curr->key != key) {
            int newLevel = randomLevel();

            if (newLevel > level) {
                for (int i = level + 1; i <= newLevel; i++) {
                    update[i] = header;
                }
                level = newLevel;
            }

            Node* newNode = new Node(key, value,
newLevel);
            for (int i = 0; i <= newLevel; i++) {
                newNode->forward[i] =
update[i]->forward[i];
                update[i]->forward[i] = newNode;
            }
        } else {
            curr->value = value;
        }
    }

    Node* search(int key) {
        Node* curr = header;

        for (int i = level; i >= 0; i--) {
            while (curr->forward[i] &&
curr->forward[i]->key < key) {
                curr = curr->forward[i];
            }
        }
```

```
        std::cout << "Level " << i << ": ";
        while (node) {
            std::cout << node->key << " -> ";
            node = node->forward[i];
        }
        std::cout << "NULL\n";
    }
}
};

int main() {
    srand(time(0));

    SkipList list;
    list.insert(3, 30);
    list.insert(6, 60);
    list.insert(7, 70);
    list.insert(9, 90);
    list.insert(12, 120);
    list.insert(19, 190);
    list.insert(17, 170);
    list.insert(26, 260);
    list.insert(21, 210);
    list.insert(25, 250);

    list.display();

    int key = 12;
    Node* result = list.search(key);
    if (result) {
        std::cout << "Found key " << key << "
with value " << result->value << "\n";
    } else {
        std::cout << "Key " << key << " not
found\n";
    }

    std::cout << "Deleting key 21\n";
    list.remove(26);
    list.display();

    return 0;
}
```

Sample Output

```
Skip List (Level 1):
```

```
Level 0: 3 -> 6 -> 7 -> 9 -> 12 -> 17 -> 19 -> 21 -> 25 -> 26 -> NULL
Level 1: 9 -> NULL
Found key 12 with value 120
Deleting key 21
Skip List (Level 1):
Level 0: 3 -> 6 -> 7 -> 9 -> 12 -> 17 -> 19 -> 21 -> 25 -> NULL
Level 1: 9 -> NULL
```

Complexity Analysis

Table 6.1 Analysis of time complexity for Skip List

Operation	Average Case	Worst Case	Space Usage
Search	O(log n)	O(n)	O(n log n)
Insert	O(log n)	O(n)	O(n log n)
Delete	O(log n)	O(n)	O(n log n)

Conclusion

In this lab, we implemented and analyzed Skip Lists and comparison with basic data structures.

- Skip Lists are an efficient alternative to balanced trees for ordered data structures.
- With logarithmic expected time complexity, they provide fast search, insertion, and deletion.
- Their probabilistic nature simplifies implementation compared to trees requiring strict balancing.
- In many practical applications, Skip Lists match or exceed performance of traditional balanced trees.
- They are especially useful in concurrent and distributed systems like databases and memory indexing.