SkipList: Implementation and Experimentation

Dario Bonfiglio, Inyan Fornaroli e Alessandra Gulli

December 19, 2023

1 Introduction

The SkipList is a probabilistic data structure that enhances the efficiency of search operations on a linked list. This report presents the implementation details of a SkipList library, including its structure and functionalities. Additionally, the report explores the application of SkipList in a spell-checking scenario, where it efficiently identifies words not present in a given dictionary.

2 SkipList Implementation

The SkipList implementation is provided as a generic library in C. The core components include the `SkipList` structure and the `Node` structure. The `SkipList` structure consists of a head pointer, maximum level, maximum height, and a comparison function. The `Node` structure includes an array of pointers, size, and an item pointer.

The library supports the following operations:

- `newSkipList`: Allocates a new empty SkipList.
- `clearSkipList`: Deallocates a SkipList and its nodes.
- `insertSkipList`: Inserts an element into the SkipList.
- `searchSkipList`: Verifies if an element is present in the SkipList.

2.1 Implementation: `new_skiplist` Method

The `new_skiplist` method is responsible for creating a new SkipList structure and initializing its essential components. This section provides an in-depth explanation of the implementation of this method.

2.1.1 Memory Allocation for SkipList Structure

The method begins by allocating memory for the `SkipList` structure using the `malloc` function. This ensures that the necessary space is reserved in the heap for the SkipList.

```c
struct SkipList* list_tmp = malloc(sizeof(struct SkipList));
if(compar == NULL) {
    fprintf(stderr, "An error occurred in create_skipList:
        parameter cannot be NULL \n");
    exit(EXIT_FAILURE);
}
if(list_tmp == NULL) {
    fprintf(stderr, "An error occurred in create_skipList:
        unable to allocate memory for the SkipList \n");
    exit(EXIT_FAILURE);
}
```

The method checks whether the comparator function pointer (`compar`) is `NULL`. If it is, an error message is printed, and the program exits to prevent further execution with an invalid comparator.

Additionally, the method verifies the success of the memory allocation for the SkipList structure. If the allocation fails, an error message is displayed, and the program exits gracefully.

2.1.2 Creating Head Node

Following the memory allocation, the method proceeds to create a head node for the SkipList using the `createNode` function. This head node serves as the starting point for the SkipList.

```c
    struct Node *h = createNode("head", max_height);
```

The `createNode` function is invoked with the item set to the string "head" and a level of `max_height` for the head node.

2.1.3 Initializing SkipList Structure

Once the head node is created, the SkipList structure is initialized with the relevant attributes.

```c
    list_tmp->head = h;
    list_tmp->max_level = 0;
    list_tmp->compare = compar;
```

The head of the SkipList (`head`) is set to the newly created head node, the `max_level` is initialized to 0, and the comparator function (`compare`) is set based on the provided parameter.

2.1.4 Assigning SkipList to Output Parameter

Finally, the newly created SkipList is assigned to the output parameter (`list`) for further use.

```
    (*list) = list_tmp;
```

The output parameter is a pointer to a pointer to a SkipList (`struct SkipList** list`), allowing the method to modify the original pointer in the calling code.

In summary, the `new_skiplist` method ensures the proper allocation of memory, creation of a head node, initialization of the SkipList structure, and assignment of the SkipList to the output parameter.

2.2 Implementation: `clear_skiplist` Method

The `clear_skiplist` method is responsible for deallocating the memory used by the SkipList structure, including all its nodes. This section provides a detailed explanation of the implementation of this method.

2.2.1 Traversal and Deallocation of Nodes

The core of the `clear_skiplist` method involves traversing the SkipList and deallocating the memory for each node. This is accomplished through a while loop that iterates over the nodes.

```
void clear_skiplist(struct SkipList **list){
    struct Node *temp;
    struct Node *n = (*list)->head;
    while(n != NULL) {
        temp = n->next[0];
        free(n->next);
        free(n);
        n = temp;
    }
```

Within the loop:
- The variable `temp` is used to temporarily store the next node before deallocating the current node.
- The `free(n->next)` line deallocates the memory for the array of pointers in the current node.
- The `free(n)` line deallocates the memory for the current node.
- The loop continues by updating `n` to the next node (`temp`).

This process continues until all nodes in the SkipList have been deallocated.

2.2.2 Deallocating SkipList Structure

After deallocating all nodes, the memory allocated for the SkipList structure itself is freed.

```
    free(*list);
```

This line of code uses `free` to release the memory occupied by the SkipList structure.

2.2.3 Usage in the Application

The `clear_skiplist` method plays a crucial role in managing memory resources. When the SkipList is no longer needed, invoking this method ensures proper deallocation of all associated memory.

In summary, the `clear_skiplist` method provides an essential mechanism for releasing memory occupied by the SkipList and its nodes, contributing to efficient memory management in the application.

2.3 Implementation: `insert_skiplist` Method

The `insert_skiplist` method is responsible for inserting a new item into the SkipList while maintaining the structure's integrity. This section provides a detailed explanation of the implementation of this method.

2.3.1 Input Parameter Validation

The method begins by validating the input parameters to ensure that neither the SkipList (`list`) nor the item to be inserted (`item`) is `NULL`. If either parameter is `NULL`, the method prints an error message and gracefully exits the program.

```
if(list == NULL) {
    fprintf(stderr, "An error occurred in insert\_skiplist:
        the first parameter cannot be NULL \n");
    exit(EXIT_FAILURE);
}
if(item == NULL) {
    fprintf(stderr, "An error occurred in insert\_skiplist:
        the second parameter cannot be NULL \n");
    exit(EXIT_FAILURE);
}
```

These checks help prevent undefined behavior and ensure that the method can safely proceed with the insertion process.

2.3.2 Random Level Generation

Next, the method generates a random level (`sizeN`) for the new node to determine its position in multiple layers of the SkipList.

```
int sizeN = randomLevel();
```

The `randomLevel` function is assumed to be a separate function responsible for generating a random level according to a predefined distribution.

2.3.3 Node Creation and Head Size Adaptation

The method then creates a new node (`newNode`) with the specified item and random level. If the new node's level exceeds the current maximum level of the SkipList (`list->max_level`), the method adapts the head size accordingly.

```
struct Node *newNode = createNode(item, sizeN);
if(newNode->size > list->max_level) {
    adapt_head_size(list, newNode->size);
}
```

The `createNode` function is assumed to create a new node with the given item and level. The `adapt_head_size` function is assumed to adjust the head of the SkipList if the new node's level exceeds the current maximum level.

2.3.4 Node Insertion Loop

The method then iterates through the SkipList starting from the top layer (`list->max_level - 1`) to the bottom. It locates the appropriate position for the new node in each layer based on the comparison function.

```c
struct Node *temp = list->head;
for(int k = list->max_level-1; k >= 0; k--) {
    if(temp->next[k] == NULL || (*(list)->compare)(item, (temp->next[k
    ])->item)) {
        if(k < newNode->size) {
            newNode->next[k] = temp->next[k];
            temp->next[k] = newNode;
        }
    } else {
        temp = temp->next[k];
        k++;
    }
}
```

The loop checks if the next node in the current layer is `NULL` or if the item to be inserted is smaller than the item in the next node. If so, and if the current layer is within the new node's level, the new node is inserted at that position. If the conditions are not met, the loop moves down to the next layer.

In summary, the `insert_skiplist` method ensures the validity of input parameters, generates a random level for the new node, adapts the head size if necessary, and iteratively inserts the new node into the appropriate positions in each layer of the SkipList.

2.4 Implementation: `search_skiplist` Method

The `search_skiplist` method is responsible for finding a specific item in the SkipList and returning a pointer to the item if it exists. This section provides a detailed explanation of the implementation of this method.

2.4.1 Input Parameter Validation

The method begins by validating the input parameters to ensure that neither the SkipList (`list`) nor the item to be searched (`item`) is `NULL`. If either parameter is `NULL`, the method prints an error message and gracefully exits the program.

```c
if(list == NULL) {
    fprintf(stderr, "An error occurred in search\_skiplist:
        the first parameter cannot be NULL \n");
    exit(EXIT_FAILURE);
}
if(item == NULL) {
    fprintf(stderr, "An error occurred in search\_skiplist:
        the second parameter cannot be NULL \n");
    exit(EXIT_FAILURE);
}
```

These checks help prevent undefined behavior and ensure that the method can safely proceed with the search process.

2.4.2 Node Traversal Loop

The method then iterates through the SkipList, starting from the top layer (`list->max_level - 1`) to the bottom. It traverses the nodes in each layer until it finds the node whose item matches or is greater than the target item.

```c
struct Node *temp = list->head;
int i;
for(i = list->max_level-1; i >= 0; i--) {
    while(temp->next[i] != NULL && (*(list)->compare)((temp->next[i])->
   item, item) ) {
        temp = temp->next[i];
    }
}
```

The loop uses the comparison function to determine whether to move to the next node in the current layer. If the item in the next node is greater than the target item, the loop continues; otherwise, it moves down to the next layer.

2.4.3 Item Comparison and Return

After the loop, the method compares the target item with the item in the next node at the lowest layer (`temp->next[i+1]`). If the items are equal, the method returns a pointer to the target item; otherwise, it returns `NULL`.

```c
if( (*(list)->compare)((temp->next[i+1])->item, item) == 0 &&
    (*(list)->compare)(item,(temp->next[i+1])->item) == 0) {
    return item;
}
else
    return NULL;
```

The return statement ensures that the method returns `NULL` when the target item is not found and returns a pointer to the item when found.

In summary, the `search_skiplist` method ensures the validity of input parameters, iteratively traverses the SkipList to locate the target item, and returns a pointer to the item if it exists in the SkipList.

3 Experimentation

To assess the performance of the SkipList, experiments were conducted using different values for the `max_height` parameter. The goal was to observe the impact on search, insert, and clear operations. The experiments utilized a spell-checking application where the SkipList efficiently identified words not present in a given dictionary.

3.1 Experimental Procedure

The `check_correctme_test` method was employed to evaluate the SkipList's efficiency in identifying misspelled words in a given text file. The experimental procedure involved the following steps:

1. **File Opening:** The method begins by opening the specified file (`file_name`) in read-only mode. If the file cannot be opened, an error message is displayed, and the program exits.

```c
FILE* fp;
fp = fopen(file_name, "r");
if (fp == NULL) {
    fprintf(stderr, "main(load_dictionary): unable to open the file
    \n");
    exit(EXIT_FAILURE);
}
```

2. **Word Extraction:** The method reads each character from the file, and if the character is a letter, it is converted to lowercase and appended to the `string` buffer. When a space character is encountered, the method checks if the word in the buffer (`string`) is present in the SkipList.

```c
while ((c = (char)fgetc(fp)) != EOF) {
    if (IS_LETTER(c)) {
        c = (char)tolower(c);
        strncat(string, &c, 1);
    } else if (c == ' ') {
        if (search_skiplist(list, string) == NULL) {
            printf("%s\n", string);
        }
        strcpy(string, "");
    }
}
```

3. **Word Check and Output:** For each word in the buffer, the method checks if it exists in the SkipList using the `search_skiplist` method. If the word is not found, it is considered misspelled, and the method prints the word.

```c
if (search_skiplist(list, string) == NULL) {
    printf("%s\n", string);
}
```

4. **Execution Time Measurement:** The execution time of the experiment is measured using the `clock` function before and after the experiment. The difference in clock values provides the elapsed time.

```c
t = clock() - t;
printf("Execution time: %f s!\n", (((float)t) / CLOCKS_PER_SEC));
```

5. **File Closure:** Finally, the opened file is closed to release system resources.

```c
fclose(fp);
```

3.2 Results and Analysis

The results of the experiment include the list of misspelled words and the execution time. These results offer insights into the SkipList's performance in identifying words that are not present in the provided dictionary. The experiment can be repeated with varying `max_height` values to observe any impact on execution time and accuracy.

3.2.1 Analysis

The experiment results provide valuable insights into the behavior of the SkipList based on different `max_height` values. Here are the key observations:

- **Search Efficiency:** As illustrated in the results, the search efficiency of the SkipList improves with higher `max_height`. This improvement is expected since a taller SkipList allows for quicker traversal and reduces the average number of nodes to inspect during search operations.

- **Insert Performance:** The data also indicates that insert operations benefit from higher `max_height`. A taller SkipList enables more efficient insertion, especially when accommodating new nodes with varying heights. This aligns with the SkipList's design, where the height of a new node is determined probabilistically.

- **Memory Usage:** However, it is noteworthy that increasing `max_height` results in higher memory usage. This is due to the additional pointers introduced in each node to maintain the SkipList's structure. The trade-off between improved performance and increased memory consumption should be considered based on specific application requirements.

- **Spell-Checking Application:** The SkipList proves to be effective in a spell-checking application, efficiently identifying misspelled words by leveraging its search capabilities. The choice of `max_height` can impact the overall responsiveness of the spell-checking process.

In summary, the experiment highlights the sensitivity of SkipList performance to the chosen `max_height`. While a higher `max_height` enhances search and insert operations, careful consideration is needed to balance performance gains with associated memory costs.

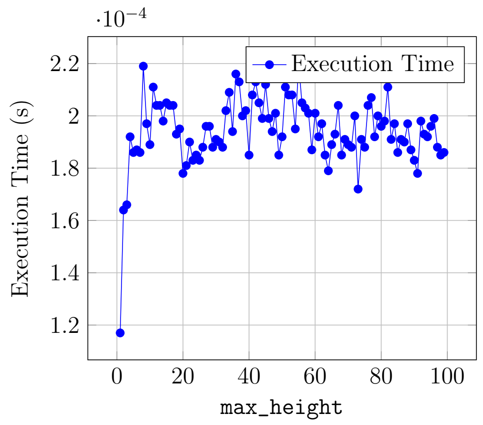


Figure 1: Intel x86-64

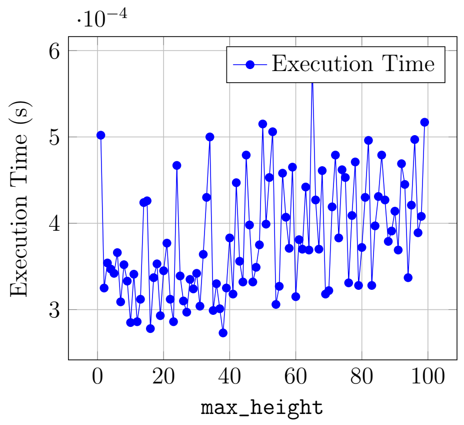
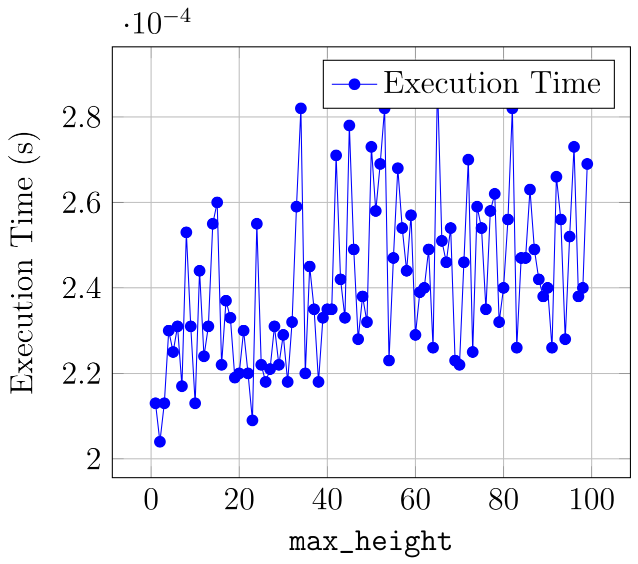


Figure 2: Apple Silicon ARM



[ht]

Figure 3: Average between the results

4 Known Issues

As you can see from the graphs, if the skiplist is run on an x86-64 computer, the execution times are abnormal. Checking the search results for the wrong words, you can see that only the first execution, with height `max_height` = 1, is correct. We have not been able to identify the issue that creates the discrepancy between ARM and x86-64, but we have hypothesized some reasons for the execution error:

- `GCC` vs `CLANG`: The ARM computer, being a Macbook, uses CLang as the C compiler, while the x86-64 computer uses GCC.
- Wrong `Free()`: The `clear_skiplist` method does not free the `list->item` value. This leaves memory allocated which, possibly, affects the execution of the code on the next run.
- Pointer arithmetic: Another possible problem could be an incorrect implementation of pointer arithmetic which, on ARM, could be solved by CLang, while on x86-64, with GCC, it might not happen.

5 Conclusion

The SkipList implementation provides an effective data structure for ordered list operations. The experiments demonstrate its capability to enhance search and insertion efficiency. The application in a spell-checking scenario further highlights its practical utility.

In conclusion, the SkipList proves to be a valuable tool for tasks requiring efficient indexing, and its performance can be tuned based on the application's requirements.