Project Report

Implementations of Finger Search:
An Extended Feature of Some Data Structures

Minh Thang Cao

14 April 2021

1. Introduction

Finger Search is an extended feature, which is suitable for some popular data structures, that improves the running time of search operations as well as other ones that depend on searching. Finger Search uses a pointer to an element, called *finger*, in the data structure to reduce the length of search path. Instead of searching from a normal start point, Finger Search starts at the *finger*. Let d be the difference between ranks[1] of the *finger* and the target value, the running time of Finger Search is proportional to d.

2. Finger Search on Treaps

2.1. Overview

Briefly, Treap is a Randomized Binary Search Tree whose properties are the combination of Binary Search Tree and Heap. Each nodes in Treap holds a randomized priority which is used to maintain the Heap properties. The nodes order in a Treap satisfies ascending values in an inorder traversal while their priority satisfies min-heap (or max-heap) property, in which priority of a node is always smaller (or larger) than its children. This randomized combination creates a special property that the *expected* height of a Treap is $O(\log n)$ in average, with n is the total number of nodes. Therefore, it supports searching and some associated operations in expected $O(\log n)$ time which is standard for most balanced trees. Since Finger Search achieve its goal in an expected running time, Treap is a very suitable randomized data structure for Finger Search.

In this section, Finger Search on Treaps will be introduced along with my implementation. Due to R. Seidel and C.R. Aragon (see [1]), there are several ways to achieve the running time proportional to the distance d such as using multiple pointers, multiple keys, or even without anything by relying on the shortness of the search path, and these approaches all give us the $O(\log d)$ expected running time, see [1] for more details. What makes Finger

[1]Denoted i if x is the i^{th} smallest element in the data structure (consistent with only smallest or largest for all elements)

Search different from normal search on Treaps is that if we start at the finger f and look for a node v, Finger Search would points out the lowest common ancestor, denoted LCA, of f and v, then it performs searching down as normal. Since if we do Finger Search without any assistance of other elements, the excess path in some cases may be longer compared to the path that we are supposed to search on. To minimize the probability that we get these cases, multiple extra pointers are definitely useful, which is the approach used in this implementation.

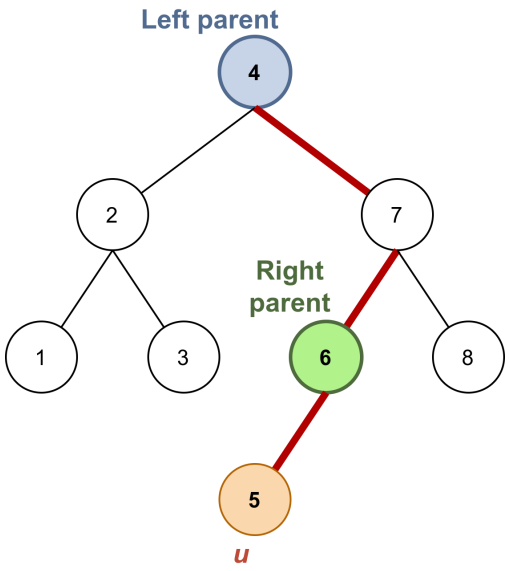

Figure 1: Visualization of *left parent* and *right parent* of a node u in a binary tree.

Based on what R. Seidel and C.R. Aragon found (see Section 5.4 of [1]), denote a *left parent* or a *right parent* of a node u is the first ancestor of u on the path from u to the *root* node so that u is on the ancestor's right or left subtree, respectively (see Figure 1). With this property, we can get the LCA quickly by jumping through *left parents* or *right parents* without traveling through all nodes on the upward path. From the LCA, we start doing Binary Search downwards to get to the node v.

Note: Implementation of data structures in this implementation is using `template` for generic type `T`.

2.2. Implementation

Since Finger Search does not affect much on the normal implementation of Treap, only some small changes are made outside of the Finger Search function. The main addition is a new *finger* node pointer property of the Treap class. Other important changes are the Node class's properties and rotation functions.

2.2.1. The Node class

To handle *left parent* and *right parent* which vary between different nodes, I added two corresponding pointers to each node other than the actual parent. Other properties remain the same. The code is given below.

```
template<typename T>
class Node {
  public:
    T data;
    double priority;
    Node *parent, *left, *right;
    Node *leftParent, *rightParent; // new properties
    Node(double value, Node* parent=nullptr,
         Node* left=nullptr, Node* right=nullptr,
         Node* leftParent=nullptr, Node* rightParent= nullptr) {
      this->data = value;
      this->parent = parent;
      this->left = left;
      this->right = right;
      this->leftParent = leftParent;
      this->rightParent = rightParent;
      priority = Random::getReal(0, 1);
    };
};
```

2.2.3. Finger Search

As described above, Finger Search would find the LCA of the finger and the target node before doing normal Binary Search from the LCA. Let assume f and v be the value in the finger and the target value. The procedure is described below:

1. Check if $f = v$, then return the *finger*.
2. If $f > v$, we start chasing *left parent* from f until reaching a null pointer or a node has data greater than both f and v. The previous node at the end of the loop is the LCA. If $f < v$, we chase *right parent* instead.
3. Perform the usual downward search from the LCA.
4. Before returning the node, if it is found, it will be the new *finger*.

The C++ code is provided below.

```
template<typename T>
Node<T>* Treap<T>::fingerSearch(T value) {
  if (value == finger->data) return finger;
  Node<T>* LCA = root; // The lowest common ancestor
  Node<T>* current = finger;
  if (value > finger->data) {
```

```
    while (current && current->data <= value) {
      LCA = current;
      current = current->rightParent;
    }
  } else {
    while (current && current->data >= value) {
      LCA = current;
      current = current->leftParent;
    }
  }
  Node<T>* foundNode = binarySearch(value, LCA); // downard search from LCA
  if (foundNode) {
    finger = foundNode;
  }
  return foundNode;
}
```

2.2.2. The rotation functions

Rotation is a main part of Treaps which maintains the heap property. Since a node's parent may be changed after a rotation, we need to modify the corresponding functions to keep the nodes' *left* and *right parents* correct. The changes are a very simple for both left and right rotation. For the right rotation at a node u, the only two node has their "special parents" changed are u and the left child of u, see Figure 2 for example. Even though the right child of u's left child changes its parent, which is u after the rotation, its *left parent* and *right parent* do not change. The procedure is completely identical for the left rotation.

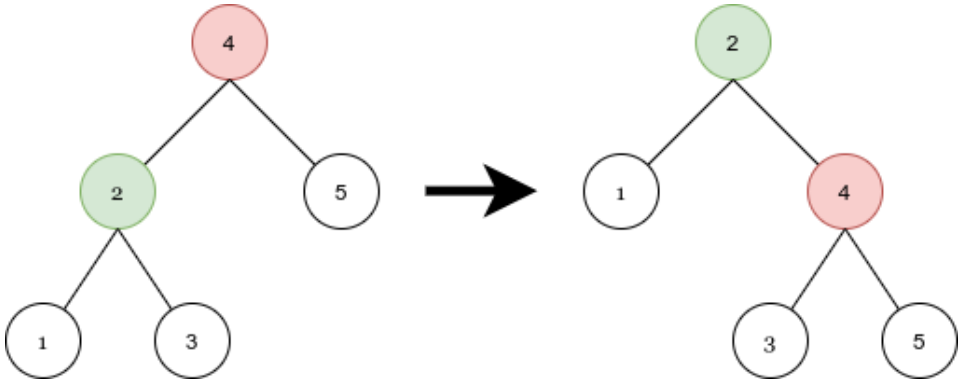

Figure 2: Visualization of right rotation at node 4. The colored nodes are ones has their *left parent* or *right parent* changed. The *right parent* of node 2, the left child of 4, changed to the *right parent* of 4, and *left parent* of node 4 changed to 2. Left rotation is identical.

Going to the real life implementation, those changes are only a couple of lines for each rotation type. The implementations of left and right rotation are given below.

```cpp
template<class T>
void Treap<T>::leftRotate(Node<T>* &node) {
  Node<T> *parent = node->parent;
  Node<T> *r = node->right;
  node->parent = r;
  node->rightParent = r; // set right parent of the given node
  r->leftParent = node->leftParent; // set left parent of right child
  if (r->left)
    r->left->parent = node;
  node->right = r->left;
  r->left = node;
  r->parent = parent;
  if (parent) {
    if (parent->left == node) {
      parent->left = r;
    } else {
      parent->right = r;
    }
  }
  node = r;
}
template<class T>
void Treap<T>::rightRotate(Node<T>* &node) {
  Node<T> *parent = node->parent;
  Node<T> *l = node->left;
  node->parent = l;
  node->leftParent = l; // set left parent of the given node
  l->rightParent = node->rightParent; // set right parent of left child
  if (l->right)
    l->right->parent = node;
  node->left = l->right;
  l->right = node;
  l->parent = parent;
  if (parent) {
    if (parent->left == node) {
      parent->left = l;
    } else {
      parent->right = l;
    }
  }
  node = l;
}
```

2.3. Running time

Let d be a difference between *ranks*[2] of any two node values on a Treap. Due to Seidel and Aragon [1], the expected running time of Finger Search on Treap theoretically is $O(\log d)$. Therefore, the closer to the $finger$ the target value is, the faster the expected running time will be. Thus, to test and prove if the running time of Finger Search in practice is identical to the theory, the search value would be chosen based on a given d.

At first, the Treap is initialized with 5 000 000 nodes that contain all values from 1 to 5 000 000, so it is easy to see that the difference between *ranks* of two nodes are exactly the difference between their values. The chosen values of d for testing starts from a small value (20) and is doubled when d increases, specifically let $D = [20, 20 \cdot 2^1, 20 \cdot 2^2, \ldots, 20 \cdot 2^{15}]$. The testing program will do the following procedure:

1. Picks a random node in the Treap and set it to be the $finger$.
2. For each d in D:
    - Repeat 5 000 000 times: search for a value $finger + d$ or $finger - d$ which is chosen randomly in the two. Each time a node is found, the $finger$ is set to that node.
    - Calculate the average running time of the trials.

See figure 3 below for the result data.

d	Finger Search time	Binary Search form root time
20	7.89250	27.2041
40	9.95326	28.0649
80	11.9180	28.8736
160	13.9715	27.4262
320	15.8764	28.0812
640	18.3024	29.7978
1280	20.2644	29.4003
2560	22.3453	28.0397
5120	24.6583	28.8494
10240	26.4185	28.1270
20480	28.3653	28.4053
40960	30.1493	28.3681
81920	32.8924	28.7313
163840	33.6583	27.9549
327680	35.6414	27.9997
655360	40.2822	27.4228

Figure 3: Average running time of Finger Search and Binary Search from root on Treap with d in $[20, 20 \cdot 2^1, 20 \cdot 2^2, \ldots, 20 \cdot 2^{15}]$. The total number of search times for each d is 5 000 000. The running times are in *number of nodes visited* at the end of the search function.

[2]position of the value in the sorted array of all values on the tree

As you can see, while the running time of Finger Search increases when d increases, the usual Binary Search from root remains about the same for all d, so we can say that Finger Search depends on d. Obviously, since the time increases about 2 when d is doubled, see figure 3, the running time of Finger Search is approximately $2 \log d = O(\log d)$. From that, in the below graph, the running time for each d in the graph is divided by $2 \log d$, denoted r_d ratio. To see how the running time of Finger Search related to $O(\log d)$ in practice, figure 4 is provided below. It is clear that the ratios calculated are almost the same for all d. This reflects that the running time of Finger Search is $O(\log d)$ in practice.

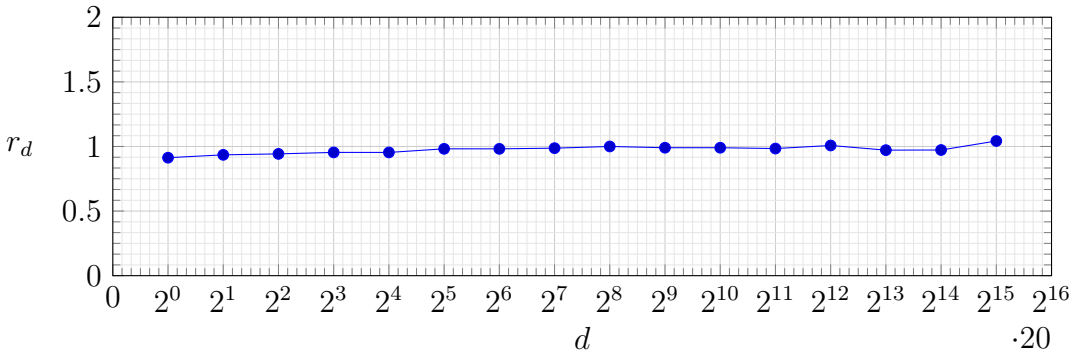

Figure 4: The graph of ratios r_d versus different values of d given in figure 3.

---

3. Finger Search on Skiplists

3.1 Overview

In this section, we go over the structure of Skiplist and how Finger Search is performed on it. Skiplist is an interesting and beautiful structure which uses randomization to optimize the times to search for an element and some other related operations to $O(\log n)$. It has a structure of multiple Linked Lists with common nodes (specifically they are the same pointers). On each level, each node is randomly determined if it exists on the next higher level. With this randomized feature, Skiplist provides a "magical" search path that produces $O(\log n)$ running time.

With the randomized property, Skiplist supports Finger Seach in $O(\log d)$ time. Due to W. Pugh (see [2]), we store the *finger* differently. Other than just one node at a place, we store the whole "cover" from the lowest node to the highest node of the list that are equal or less than the node at level 1 of the finger. In other words, the finger is an array of nodes that "cover" the bottom-left corner of the Skiplist from the chosen node. See figure 5 for example.

With this *finger* array, the main idea if Finger Search on Skiplist due to W. Pugh is:

- If value of the node at the lowest level of the *finger* is less than the search value, find the highest level whose node of the *finger* has *next* pointer less than the search value.
- If value of the node at the lowest level of the *finger* is larger than the search value, find the lowest level whose node of the *finger* has *next* pointer less than the search value.

From the found level, we start searching normally (on the normal Skiplist search path) from the *next* node of the *finger* pointer at that level.

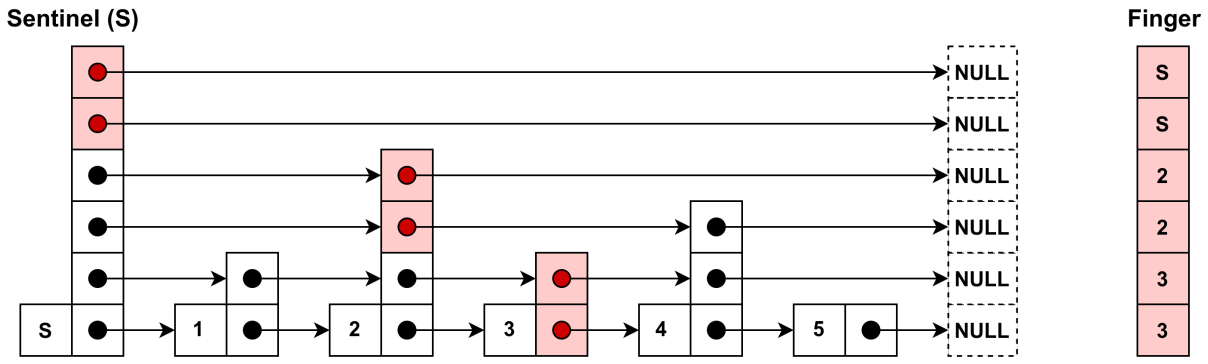

Figure 5: Visualization of a *finger* on a Skiplist. In this example, finger is an array of all nodes cover the bottom-left corner of the Skiplist from node 3.

3.2. Implementation

Due to W. Pugh (see [2]), Finger Search on Skiplist does not require any additional properties of any other functions or the node class to perform searching in $O(\log d)$. Thus, Finger Search is implemented in its own function without any outside requirements. Let x be the search value and assume indices of arrays starts at 1. The steps of Finger Search on Skiplist is described below:

1. If $finger[1] < x$, then start from level 2, move upwards until the next pointer of current node points to a node larger than x, null or the current *level* out of bound. At that time, $level - 1$ is the level we start searching normally.

   Otherwise, starting from level 2, we also search upwards but until we see the first level such that the next node of the $finger$ at that level is less than the search node.

2. From the next node of the $finger$ pointer at the *level* found at the previous step, perform search as usual (if next node $< x$, move to the *next* pointer, otherwise, move down one level until reaching the lowest).

   Each time it searches **down**, set the finger at the current *level* to the current node to update the $finger$.

Implementation of Skiplist in this project is provided by Pat Morin, see [2]. The C++ code is provided below. More error checking is provided while searching upwards.

```
template<class T>
T Skiplist<T>::fingerSearch(T x) {
  int level = 1;
  Node* start;
  if (compare(finger[0]->x, x) == -1) {
    while (level <= h &&
           (compare(finger[level]->x, x) >= 0
             || (finger[level]->next[level] != nullptr
```

```
                  && compare(finger[level]->next[level]->x, x) == -1))) {
      if (compare(finger[level]->x, x) == 0) {
        return x;
      }
      level++;
    }
    start = finger[level]->next[level];
  } else {
    while (level <= h &&
            (compare(finger[level]->x, x) >= 0
              || (finger[level]->next[level] != nullptr
                   && compare(finger[level]->next[level]->x, x) >= 0))) {
      if (compare(finger[level]->x, x) == 0) return x;
      level++;
    }
    if (level > h) {
      level = h;
      start = sentinel;
    } else {
      start = finger[level]->next[level];
    }
  }
  for (int i = level; i >= 0; i--) {
    while (start->next[i] != nullptr && compare(start->next[i]->x, x) < 0)
     {
      start = start->next[i];
    }
    finger[i] = start;
  }
  return start->next[0] == nullptr ? null : start->next[0]->x;
}
```

3.3. Running time

The running time of Finger Search on Skiplist is identical to Treap. The concept of distance between *ranks* of the lowest node of the *finger* and the search value is applied the same as Treap. Theoretically, the running time of Finger Search on Skiplist is $O(\log d)$, so if the search value x is closer to the lowest node of *finger*, search time would be faster.

The same testing method is used as on Treap: initialize the Skiplist with 5 000 000 integers from 1 to 5 000 000, so the difference between *ranks* of values is the same as the difference between those values. The testing d is also $\in \{20, 20 \cdot 2^1, 20 \cdot 2^2, \ldots, 20 \cdot 2^{15}\}$, and using the procedure the same on Treap (see Section 2.3 for more details).

3.3.1. Normal running time due to W. Pugh's approach

The result data with W. Pugh's approach is provided below.

d	Finger Search time	Usual search path from Sentinel
20	39.8817	44.7127
40	40.3754	41.5027
80	41.9393	41.3186
160	43.7000	41.5630
320	46.2551	43.6607
640	48.3684	44.7823
1280	48.1796	44.3474
2560	50.3686	44.5856
5120	51.7555	43.8798
10240	53.2053	43.3164
20480	54.6277	43.2839
40960	56.1563	43.2849
81920	57.9879	43.4521
163840	58.7704	42.8980
327680	60.6042	43.0317
655360	63.1080	42.6243

Figure 6: Average running time of Finger Search and Usual search from sentinel on Skiplist with d in $[20, 20 \cdot 2^1, 20 \cdot 2^2, \ldots, 20 \cdot 2^{15}]$. The total number of search times for each d is 5 000 000. The running times are in *number of nodes visited* at the end of the search function.

As you can see, the running time behaviour of Finger Search on Skiplist is identical to Treap. It is increasing by about 2 nodes visited when d is doubled, so it leads to the running time $2 \log d = O(\log d)$. The only difference is the running time of Finger Search on Skiplist is much slower compared to Treap (it starts from 40 nodes visited while on Treap, it starts from 8 nodes visited), see Figure 6 and Figure 3 for more details.

3.3.2. Running time with additional concurrent linear search

Additional to the approach of W. Pugh (see [2]), I introduce a simple concurrent linear search along with Finger Search when we search upwards or downwards. For every iteration and with a tracking variable, we perform a linear search (go to the next node or previous node after every iteration) from the lowest level of the $finger$ towards the search value x. This approach is created in the hope of reduce the running time of Finger Search if the $finger$ is close to x.

Running time of Finger Search with linear search is provided below.

d	Finger Search time with linear search	Usual search path from Sentinel
20	20	43.5811
40	40	41.5027
80	40.8913	40.0918
160	43.3681	41.7235
320	45.2700	42.1257
640	47.6589	43.9823
1280	48.7145	44.4602
2560	49.1672	43.8023
5120	51.0105	43.5328
10240	53.1527	44.0019
20480	54.3721	43.4647
40960	55.8258	42.5021
81920	57.0091	44.1179
163840	60.1422	43.0084
327680	61.7512	44.1739
655360	63.5892	44.8285

Figure 7: Average running time of Finger Search with linear search and Usual search from sentinel on Skiplist with d in $[20, 20 \cdot 2^1, 20 \cdot 2^2, \ldots, 20 \cdot 2^{15}]$. The total number of search times for each d is 5 000 000. The running times are in *number of nodes visited* at the end of the search function.

It is obvious that the linear search helps improve running time searching for a value very close to the $finger$. As long as the Finger Search path to the value is longer than the $ranks$ between them, the linear search reduce the search time equals to the difference in $ranks$. However, when d gets larger, Finger Search remains search time $O(\log d)$ while the linear search is $O(d)$, so the larger d gets, the faster Finger Search is compared to linear search. As you can see in Figure 7, with $d \in \{20, 40\}$, Finger Search time is equal to d (which is the linear search time) while if $d \in \{80, 160, \ldots, 655360\}$, running time of Finger Search is less than d (which is the normal Finger Search time). Since there is only a small portion of d in which linear search is faster, we can conclude that the addition of it in Finger Search is not useful, and it does not reflect and have any contribute to the consistent $O(\log d)$ running time of Finger Search.

4. Finger Search on Treaps and Skiplists comparison

4.1. Running time

In theory, since both Treap and Skiplist are randomized data structures, they are very suitable for Finger Search. With that property, they produce search time $O(\log d)$ with d is the difference between $ranks$ of the $finger$'s value and the search value. In practice, the

running time of both reflect the $O(\log d)$ time. Specifically, times search on both structures increase by about 2 in average when d is doubled. This is consistent for both randomized data structure. However, when we compare them, the time of Finger Search on Treap is much faster, see Section 3.3.1 and Section 2.3 for detailed data.

4.2. Implementation difficulties

Although all data structures have the same idea about Finger Search, implementation of each data structure is very different from the others. Through implementations of Finger Search of Treap and Skiplist, to me, the easier one is Treap. Even I needed to change some functions other than the new Finger Search function, while the only change in Skiplist is the new search function, the implementation of Finger Search on Treap is very simple compared to Skiplist.

On Treap, changing the rotation functions takes only two lines and easy to keep track. For the Finger Search function, I did not need complicated error checking for special cases since they are automatically excluded in Finger Search. Therefore, the implementation is straightforward and even shorter.

On Skiplist, there are cases that the *next* pointer is *null* or the pointer is going out of the list. To handle these cases, error checking is needed in the function. In addition, since the $finger$ is not just a pointer while it is an array in Skiplist, changing between different level is needed and and it is more complicated compare to Treap while it just compares the value of the pointer with current node's value.

5. Concluding remarks

Finger Search and its implementations has been described detailedly in the report. Its performance is exactly as theory as producing running time $O(\log d)$. There are several conclusions regarding to Finger Search in the real life:

1. The algorithm's searching approach is not the same for every data structure. Each of them has its own implementation and running time.
2. For both Treap and Skiplist, Finger Search runs in $O(\log d)$. Specifically, it is $\approx 2 \log d$ while the running time increases by 2 if d is doubled.
3. Finger Search on Treap is much faster than on Skiplist.

References

[1] Seidel, R., Aragon, C.R. Randomized search trees. *Algorithmica* 16, 464–497 (1996). https://doi.org/10.1007/BF01940876

[2] W. Pugh. *A skip list cookbook*. Technical Report CS-TR-2286.1, Dept. of Computer Science, University of Maryland, College Park, 1989.

[3] Morin, P. (2014). *Open Data Structures*. Edmonton: Athabasca University Press. https://opendatastructures.org/