Project Report

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

Minh Thang Cao

09 March 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 $O(logn)$ in average, with n is the total number of nodes. Therefore, it supports searching and some associated operations in expected $O(logn)$ 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 $O(logd)$ expected running time, see [1] for more details. What makes Finger Search

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

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.

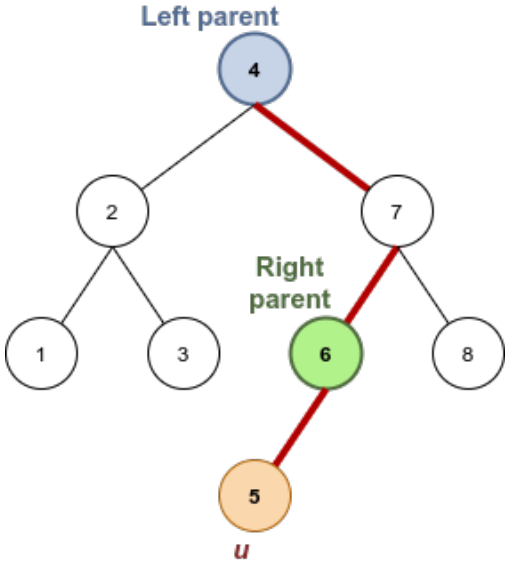

Figure 1: Visualization of *left parent* and *right parent* of a node u in a binary tree.

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.

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.

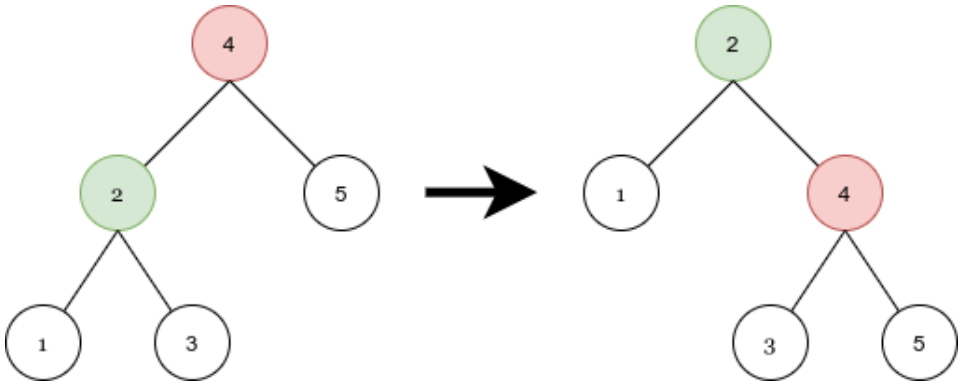

Figure 2: Visualization of right rotation at node 4. The colored nodes are ones has their *left parent* or *right parent* changed. The *right parent* of node 2, the left child of 4, changed to the *right parent* of 4, and *left parent* of node 4 changed to 2. Left rotation is identical.

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.

```
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 $O(logd)$. 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 $2logd = O(logd)$. From that, in the below graph, the running time for each d in the graph is divided by $2logd$, denoted r_d ratio. To see how the running time of Finger Search related to $O(logd)$ 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 $O(logd)$ in practice.

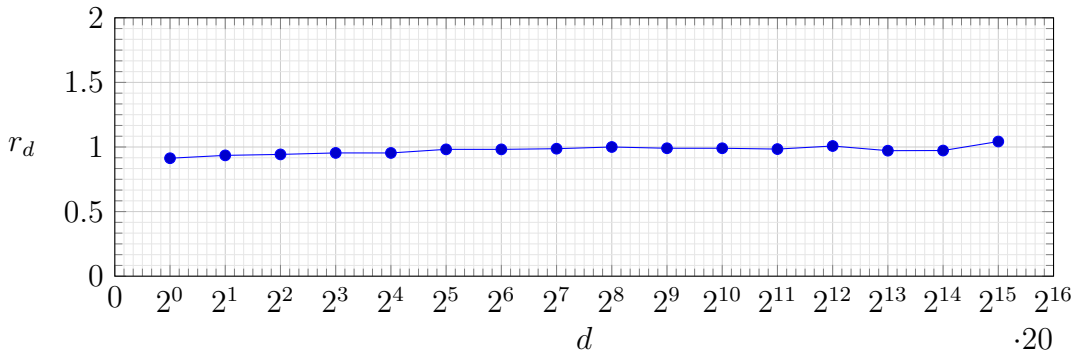

Figure 4: The graph of ratios r_d versus different values of d given in figure 3.

---

3. Finger Search on Skiplists

References

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