Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Engineering
Academic Year 2022-2023

Efficient Phone Book Management System Using Randomized BST

Advance Algorithm Laboratory

By

Jiya Patel **60004200105**
Ayush Jain **60004200132**
Dhruv Gandhi **60004200133**

Guide:

Prof. Chetashri Bhadane

Assistant Professor

Department of Computer Engineering
Academic Year 2022-2023

1. PROBLEM STATEMENT

The problem statement is to maintain the Phone Book Dictionary of a particular user. Phone Book dictionary is the one, which contain details of an individual along with their contact numbers. In Phone Book Dictionary names should be present in alphabetical order so that one can easily find the required person along with their telephone numbers. The phonebook application works specifically for tracking people. The Phonebook application contains a set of basic functions for adding, searching, updating, and deleting new contacts. This mini-C phonebook design allows you to perform simple tasks in your phonebook, such as mobile phones. You can add text to the phonebook, find, edit, search, and delete.

2. INTRODUCTION

a. Need

Phonebook project is a very simple tool that helps you understand the basic concepts of creation, file extensions and data structure. This software teaches you how to add, view, edit or modify, receive and delete data from files.

Adding new items, viewing them by logging in, editing and updating, searching for saved contacts and deleting data in the phonebook is one of the main features of the main phonebook application.

The phonebook application works specifically for tracking people. The Phonebook application contains a set of basic functions for adding, searching, updating, and deleting new contacts.

This application provides information on adding, viewing, modifying, receiving, and deleting data from/to files. Adding new entries, browsing them, editing and updating, searching for saved contacts, and deleting contacts in the phonebook is one of the most important services that become the main menu in the phonebook application.

b. Working

The following are the basic functionalities of the Phonebook Dictionary:

i. Main Menu:

When you start the project from any compiler or by double-clicking the executable.exe file, you'll see the Main Menu screen on window.

ii. Add Contact:

When you Choose to add contact, it opens a form which will be accepting the input from user in order to add new phonebook in the database.

iii. Delete Contact:

When you choose to delete contact its open a form which will be accepting the name of contact which you want to delete and the it will be deleting the phonebook information in the database.

iv. Search by Number:

When you choose to search a contact by number, system will be accepting the number you entered and searching throughout the database and will return the retrieved information and if not found it will be returning a message of Not Found.

v. Display Contacts in Ascending Order:

Its will be displaying the all the contacts in Phonebook Dictionary in Ascending Order.

vi. Display Contacts in Descending Order:

Its will be displaying the all the contacts in Phonebook Dictionary in Descending Order.

c. Applications

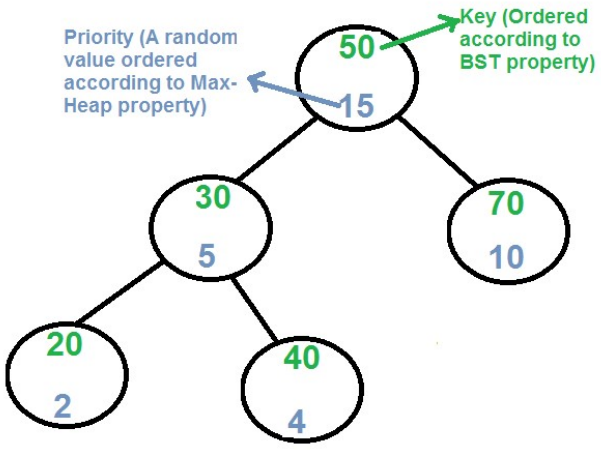
- The phonebook application works specifically for tracking people.
- Adding new records, listing them, modifying them and updating, search for contacts saved, and deleting the phonebook records are the basic functions which make up the main menu of this Phonebook application (as shown in the main menu screenshot below).
- Phonebook application can be used to maintain the users contacts.
- Phonebook records can be modified, listed, searched for and removed.
- And in searching it can be searched in both Ascending and Descending Order.

3. ADVANCED DATA STRUCTURE

a. Theory/Working

Treap (Randomized BST).:

Like Red-Black and AVL Trees, Treap is a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n). The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is O(Log n).



- Every node of Treap maintains two values.
- Key Follows standard BST ordering (left is smaller and right is greater)
- Priority Randomly assigned value that follows Max-Heap property.

Basic Operation on Treap: Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap property during insertion and deletion.

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

```
     y                               x
    / \     Right Rotation          /  \
   x   T3   – – – – – – – >       T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation            T2  T3
```

Keys in both of the above trees follow the following order:

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

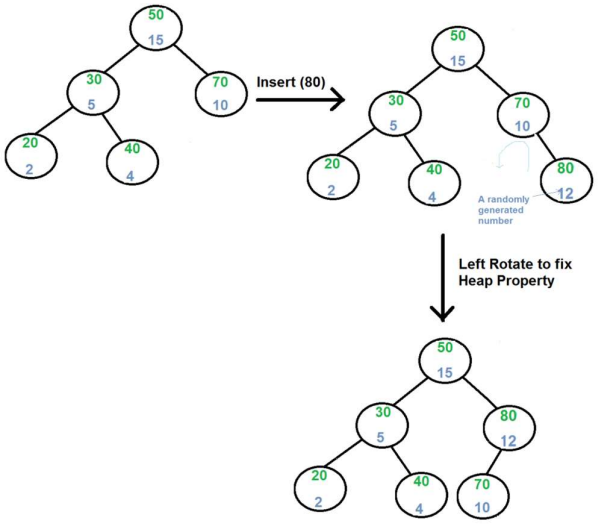
So BST property is not violated anywhere.

search(x):

Perform standard BST Search to find x.

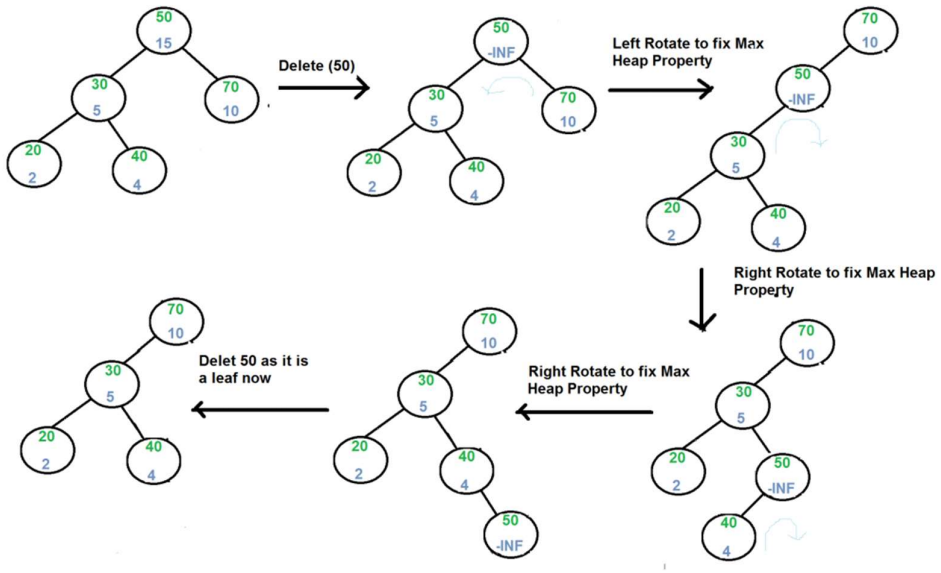
Insert(x):

- Create new node with key equals to x and value equals to a random value.
- Perform standard BST insert.
- Use rotations to make sure that inserted node's priority follows max heap property.



Delete(x):

1. If node to be deleted is a leaf, delete it.
2. Else replace node's priority with minus infinite (-INF), and do appropriate rotations to bring the node down to a leaf.



b. Applications

Randomized BSTs are used for a lot of applications due to its ordered structure.

- Randomized BSTs are used for indexing and multi-level indexing.
- They are also helpful to implement various searching algorithms.
- It is helpful in maintaining a sorted stream of data.
- TreeMap and TreeSet data structures are internally implemented using self-balancing BSTs.

c. Complexity Analysis

A treap provides the following operations:

i. **Insert (X, Y)** in O (log N).

Adds a new node to the tree. One possible variant is to pass only X and generate Y randomly inside the operation.

ii. **Search (X)** in O (log N).

Looks for a node with the specified key value X. The implementation is the same as for an ordinary binary search tree.

iii. **Erase (X)** in O (log N).

Looks for a node with the specified key value X and removes it from the tree.

iv. **Build (**X1**, ...,** XN**)** in O(N).

Builds a tree from a list of values. This can be done in linear time (assuming that X1..., XN are sorted).

v. **Union (**T1**,** T2**)** in O (M log(N/M)).

Merges two trees, assuming that all the elements are different. It is possible to achieve the same complexity if duplicate elements should be removed during merge.

vi. **Intersect (**T1**,** T2**)** in O (M log(N/M)).

Finds the intersection of two trees (i.e., their common elements). We will not consider the implementation of this operation here.

4. IMPLEMENTATION

a. Important screen shots

```
==================Phone Book Management System Using Treaps====================
--------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

--------------MENU--------------

--------------------------------
Enter the choice: 1

--------------------------------


--------------------------------
Enter Name:
Ayush
Enter Phone No.:
9137716225
```

```
---------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

---------------MENU--------------

---------------------------------
Enter the choice: 3

---------------------------------


---------------------------------
Enter the Phone No.:
9137716225
1 Name :            Ayush, Phone No. 9137716225
```

```
---------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

---------------MENU--------------

---------------------------------
Enter the choice: 4

---------------------------------


---------------------------------
Printing Contact in Ascending Order:
1 Name :            Ayush, Phone No. 9137716225
2 Name :            Dhruv, Phone No. 9867517870
3 Name :             Jiya, Phone No. 9137716224
```

```
---------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

---------------MENU--------------

---------------------------------
Enter the choice: 5

---------------------------------


---------------------------------
Printing Contact in Descending Order:
1 Name :             Jiya, Phone No. 9137716224
2 Name :            Dhruv, Phone No. 9867517870
3 Name :            Ayush, Phone No. 9137716225
```

```
---------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

---------------MENU--------------

---------------------------------
Enter the choice: 2

---------------------------------


---------------------------------
Enter Name to Delete:
Jiya
```

b. Code of important functions

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct node{
   char name[40];
   char phone_no[12];
   struct node*left;
   struct node*right;
};
struct node*head = NULL;
char temp_n[40];
char temp_p[12];

struct node* createNode(char*name, char*phone_no){
   struct node*new_node = (struct node*)malloc(sizeof(struct node));
   strcpy(new_node->name, name);
   strcpy(new_node->phone_no, phone_no);
   new_node->left = new_node->right = NULL;
   return new_node;
}

void insert(char*name, char*phone_no)
{
   struct node*root = head;
   struct node*prev = NULL;
   if(root == NULL)
   {
      struct node*new_node = createNode(name, phone_no);
      head = new_node;
      return;
```

```
    }

    while(root!=NULL)
    {
       prev = root;
       if(strcmp(root->name,name)==0)
       {
          printf("Name is already there!\n");
          return;
       }
       else if(strcmp(root->name, name)>0)
       {
          root = root->left;
       }
       else
       {
          root = root->right;
       }
    }
    struct node* new_node = createNode(name, phone_no);
    if(strcmp(prev->name, name)>0)
    {
       prev->left = new_node;

    }
    else
    {
       prev->right = new_node;
    }
}

int count_asc = 1;
void show_ascending(struct node*root)
```

```
{
   if(root!=NULL)
   {
      show_ascending(root->left);
      printf("%d Name : %15s, Phone No. %-10s \n",count_asc, root->name,
root->phone_no);
      count_asc++;
      show_ascending(root->right);
   }
}

int count_desc = 1;
void show_descending(struct node*root)
{
   if(root!=NULL)
   {
      show_descending(root->right);
      printf("%d Name : %15s, Phone No. %-10s \n",count_desc, root->name,
root->phone_no);
      count_desc++;
      show_descending(root->left);
   }
}

int found = 0;
int count_search = 1;
void search(struct node*root, char*phone_no)
{
   if(root!=NULL)
   {
      search(root->left, phone_no);
      if(strcmp(root->phone_no, phone_no) == 0)
      {
```

```
            printf("%d Name : %15s, Phone No. %-10s \n",count_search, root->name, root->phone_no);
            found = 1;
            count_search++;
        }
        search(root->right, phone_no);
    }
}

struct node*predecessor(struct node*root)
{
    while(root->right != NULL)
    {
        root = root->right;
    }
    return root;
}

struct node*delete(struct node*root, char*name)
{
    if(root == NULL)
    {
        return root;
    }

    if(strcmp(root->name, name)>0)
    {
        root->left = delete(root->left, name);
    }
    else if(strcmp(root->name, name)<0)
    {
        root->right = delete(root->right, name);
    }
```

```
    else
    {
       if(root->left == NULL)
       {
          struct node*temp = root->right;
          free(root);
          return temp;
       }
       else if(root->right == NULL)
       {
          struct node*temp = root->left;
          free(root);
          return temp;
       }

       struct node*pre = predecessor(root->left);
       strcpy(root->name, pre->name);
       root->left = delete(root->left, pre->name);

    }
    return root;
}

int main()
{
    printf("\n==================Phone Book Management System Using
Treaps==================");
    //
printf("================================================================
====================\n");
    menu:
    printf("\n---------------MENU--------------\n");
    printf("1. Add Contacts\n");
```

```
printf("2. Remove Contacts\n");
printf("3. Search by Phone No.\n");
printf("4. Shown in Ascending Order\n");
printf("5. Shown in Descending Order\n");
printf("6. Exit Program\n");
printf("\n---------------MENU--------------\n");
int choice;
printf("\n---------------------------------\n");
printf("Enter the choice: ");
scanf("%d", &choice);
printf("\n---------------------------------\n");
printf("\n");
if(choice == 1)
{
   printf("\n---------------------------------\n");
   printf("Enter Name: \n");
   while((getchar())!= '\n');
   scanf("%[^\n]%*c", temp_n);
   printf("Enter Phone No.: \n");
   scanf("%[^\n]%*c", temp_p);
   insert(temp_n,temp_p);
}
else if(choice == 2)
{
   printf("\n---------------------------------\n");
   printf("Enter Name to Delete: \n");
   while((getchar())!= '\n');
   scanf("%[^\n]%*c", temp_n);
   delete(head, temp_n);
}
else if(choice == 3)
{
   printf("\n---------------------------------\n");
```

```
    struct node*root = head;
    if(root == NULL)
    {
       printf("\n No Contacts are there to Search!");
       goto menu;
    }

    printf("Enter the Phone No.: \n");
    while((getchar())!= '\n');
    scanf("%[^\n]",temp_p);
    search(root, temp_p);
    if(found == 0)
    {
       printf("No Contacts Found!\n");
    }
    found = 0;
    count_search = 1;
    printf("\n");
}
else if(choice == 4)
{
    printf("\n---------------------------------\n");

    struct node*root = head;
    if(root == NULL)
    {
       printf("\n No Contacts are there to show!");
       goto menu;
    }
    printf("Printing Contact in Ascending Order: \n");
    show_ascending(root);
    printf("\n");
```

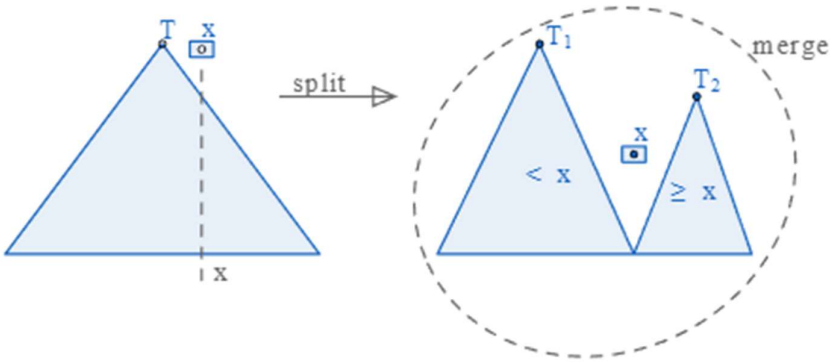
```
            count_asc = 1;
        }
        else if(choice == 5)
        {
            printf("\n---------------------------------\n");
            struct node*root = head;
            if(root == NULL)
            {
                printf("\n No Contacts are there to show!");
                goto menu;
            }
            printf("Printing Contact in Descending Order: \n");
            show_descending(root);
            printf("\n");
            count_desc = 1;
        }
        else if(choice == 6)
        {
            printf("\n-----------------------------------------------------------------\n");
            printf("\nContact Dictinary Signing off! Don't Forget to save
    Contacts!\n");
            printf("\n-----------------------------------------------------------------\n");
            exit(0);
        }
        else
        {
            printf("\n---------------------------------\n");
            printf("Enter the valid option! \n");
        }
        goto menu;

        return 0;
    }
```

5. COMPLEXITY ANALYSIS

Operations	Array and List	Binary Search Tree	Treap(Randomized BST).
Insertion	O(N)	O(N) or O(LOG N)	O(LOG N)
Deletion	O(N)	O(N) or O(LOG N)	O(LOG N)
Sorting	$O(N^2)$ or O(N LOG N)	O(LOG N)	O(LOG N)
Searching	O(N) or O(LOG N)	O(LOG N)	O(LOG N)

In terms of implementation, each node contains X, Y and pointers to the left (L) and right (R) children.

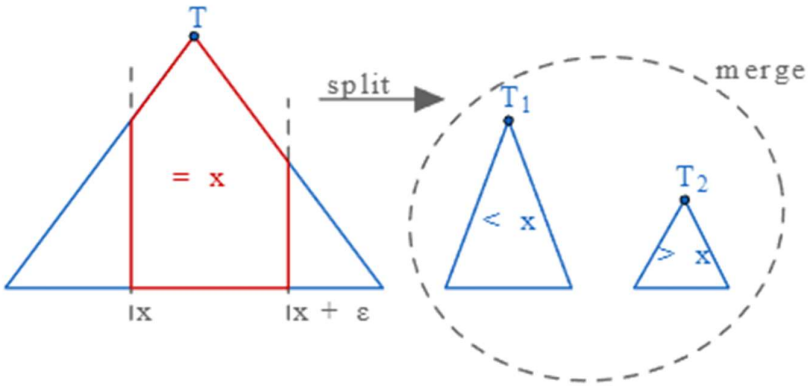
1. Insert



Implementation of **Insert (X, Y)** becomes obvious. First we descend in the tree (as in a regular binary search tree by X), and stop at the first node in which the priority value is less than Y. We have found the place where we will insert the new element. Next, we call **Split (T, X)** on the subtree starting at the found node, and use returned subtrees L and R as left and right children of the new node.

Alternatively, insert can be done by splitting the initial treap on X and doing 2 merges with the new node (see the picture).

2. Delete.



Implementation of **Erase (X)** is also clear. First we descend in the tree (as in a regular binary search tree by X), looking for the element we want to delete. Once the node is found, we call **Merge** on it children and put the return value of the operation in the place of the element we're deleting.

Alternatively, we can factor out the subtree holding X with 2 split operations and merge the remaining treaps (see the picture).

3. Searching:

For searching element 1, we have to traverse all elements (in order 3, 2, 1). Therefore, searching in binary search tree has worst case complexity of O(n). In general, time complexity is O(h) where h is height of Randomized BST.

In which h can be evaluated as the f (log n) in which n is the number of elements present in tree, It can be n in some cases if you get random priority in increasing order. But it's not every time possible in computer environment so Time complexity if Treap Tree which is Randomized Binary Search Tree.

6. CONLCUSION

The application software has been implemented successfully by using test cases. And the language used is C language. This Phonebook application is used to add, search, delete and some functions which is used to remember our contact details more easily. Application is implemented using Treap a Randomized Binary Search Tree. Compared to balanced BSTs that are not probabilistic (AVL trees), a treap is more simplistic and thus, quicker to code, since its structural property relies solely on the priority keys. A Treap can be used to quickly join and split sets of data. Given two sets of data, you can perform a split by merely inserting a dummy node with an infinite priority value. By doing this, the dummy ends up as the root of the tree, with its left and right children being the two sets of data. Treap Data Structure is a self-organizing data structure. They take after themselves and do not require supervision. Unlike other self-balancing trees, they do not require sophisticated algorithms (simple tree rotations will suffice, although simpler algorithms involving arrays can do the job too).