

MIT WORLD PEACE UNIVERSITY

Advanced Data Structures
Second Year B. Tech, Semester 4

IMPLEMENTATION OF DICTIONARY USING BINARY
SEARCH TREE

ASSIGNMENT NO. 3

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

February 15, 2023

Contents

1 Objectives	1
2 Problem Statement	1
3 Theory	1
3.1 Binary Search Tree	1
3.2 Breadth First Traversal	2
3.3 Different operations on binary search tree	2
3.3.1 Copy	2
3.3.2 Insert	2
3.3.3 Mirror Image	2
3.3.4 Delete	3
4 Platform	3
5 Input	3
6 Output	4
7 Test Conditions	4
8 Pseudo Code	4
8.1 Create	4
8.2 Display	5
8.3 Delete	5
8.4 Mirror Image	6
8.5 Copy	6
9 Time Complexity	7
9.1 Create	7
9.2 Display	7
9.3 Delete	7
9.4 Mirror Image	7
9.5 Copy	7
10 Code	7
10.1 Program	7
10.2 Input and Output	16
11 Conclusion	19
12 FAQ	20

1 Objectives

1. To study data structure : Binary Search Tree
2. To study breadth first traversal.
3. To study different operations on Binary search Tree.

2 Problem Statement

Implement dictionary using binary search tree where dictionary stores keywords and its meanings. Perform following operations:

1. Insert a keyword
2. Delete a keyword
3. Create mirror image and display level wise
4. Copy the binary Search Tree

3 Theory

3.1 Binary Search Tree

Binary Search Trees are a special type of binary trees where the value of all the nodes in the left sub-tree is less than the value of the root and the value of all the nodes in the right sub-tree is greater than the value of the root.

The left and right sub-trees are also binary search trees. This property of binary search trees makes them useful for searching, as the desired node can be found by repeatedly comparing the input to the value of the root and choosing the appropriate sub-tree, without having to search through the entire tree.

Binary search trees are also useful for sorting, as it is easy to sort the nodes in ascending order by performing an in-order traversal of the tree.

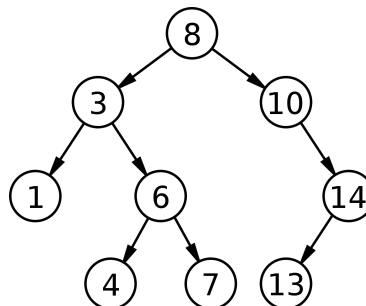


Figure 1: Example of a Binary Search Tree

3.2 Breadth First Traversal

Breadth First Traversal (or Level Order Traversal) is a tree traversal algorithm where we should start traversing the tree from root and traverse the tree level wise i.e. traverse the nodes level by level.

In level order traversal, we visit the nodes level by level from left to right.

3.3 Different operations on binary search tree

3.3.1 Copy

To copy a Binary Search Tree into another Binary Search Tree, we perform a pre-order traversal of the tree. For each node, we create a new node with the same value and then recursively copy the left and right sub-trees of the node.

To copy it Iteratively, we use a queue to store the nodes of the tree. We start by pushing the root node into the queue. We then pop a node from the queue and create a new node with the same value as the popped node. We then push the left and right child of the popped node into the queue and repeat the process until the queue is empty.

3.3.2 Insert

To insert a node in a binary search tree, we start by comparing the value of the node to be inserted with the value of the root node. If the value of the node to be inserted is less than the value of the root node, we move to the left sub-tree and repeat the process. If the value of the node to be inserted is greater than the value of the root node, we move to the right sub-tree and repeat the process.

We repeat this process until we reach a leaf node. The new node is then inserted as the left or right child of the leaf node, depending on the value of the node to be inserted.

3.3.3 Mirror Image

To create a mirror image of a binary search tree, we perform a pre-order traversal of the tree. For each node, we swap the left and right child of the node. We then recursively create the mirror image of the left and right sub-trees of the node.

To create it Iteratively, we use a stack to store the nodes of the tree. We start by pushing the root node into the stack. We then pop a node from the stack and swap the left and right child of the popped node. We then push the left and right child of the popped node into the stack and repeat the process until the stack is empty.

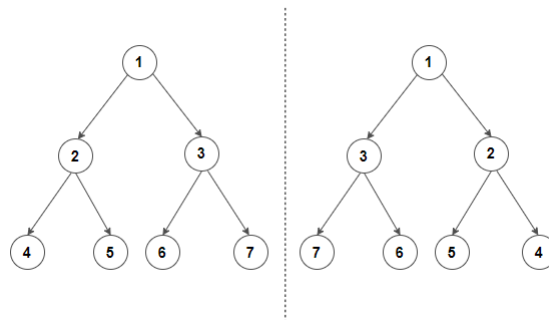


Figure 2: Mirror of a Binary Tree

3.3.4 Delete

There are 3 Cases for deletion of a node in a binary search tree:

1. The node to be deleted is a leaf node. In this case, we simply remove the node from the tree.
2. The node to be deleted has only one child. In this case, we replace the node to be deleted with its child.
3. The node to be deleted has two children. In this case, we find the inorder successor of the node. We replace the value of the node to be deleted with the value of the inorder successor. We then delete the inorder successor. The inorder successor will have at most one child node, so we can use the above two cases to delete it.

It can also be done using the inorder predecessor. The inorder predecessor is the largest node in the left sub-tree of the node. We replace the value of the node to be deleted with the value of the inorder predecessor. We then delete the inorder predecessor. The inorder predecessor will have at most one child node, so we can use the above two cases to delete it.

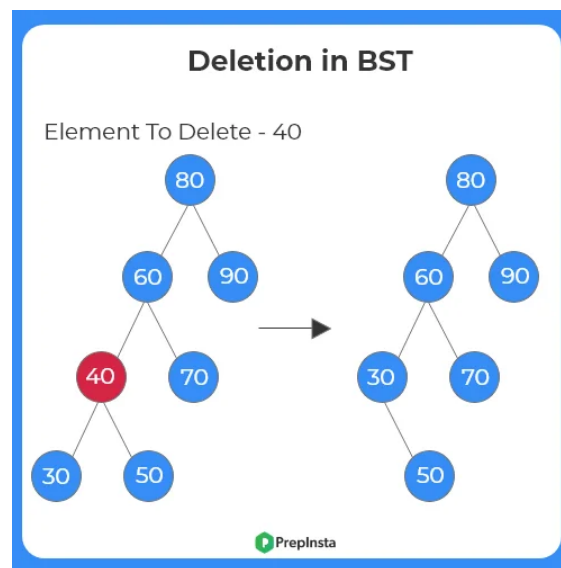


Figure 3: Deleting a node from a Binary Search Tree

4 Platform

Operating System: Arch Linux x86-64

IDEs or Text Editors Used: Visual Studio Code

Compilers : g++ and gcc on linux for C++

5 Input

1. Input at least 10 nodes.

2. Display binary search tree levelwise traversals of binary search tree with 10 nodes
3. Display mirror image and copy operations on BST

6 Output

1. The traversal of the binary tree in different ways.

7 Test Conditions

1. Input at least 10 nodes.
2. Display all traversals of binary tree with 10 nodes.(recursive and nonrecursive)

8 Pseudo Code

8.1 Create

```
1  void create_root()
2      root = new WordNode
3      display - "Enter the data: " << endl
4      Take Input of root->word
5      Take Input of root->definition
6      root->left = NULL
7      root->right = NULL
8      create_recursive(root)
9
10 void create_recursive(WordNode *Node)
11     int choice = 0
12     WordNode *new_node
13     display - "Enter if you want to enter a left node (1/0): "
14         << "for the node -- " << Node->word << ": " << Node->definition << "
15     -- "
16     Take Input of choice
17     if (choice == 1)
18         new_node = new WordNode
19         display - "Enter the data: "
20         Take Input of new_node->word
21         Take Input of new_node->definition
22         Node->left = new_node
23         create_recursive(new_node)
24     display - "Enter if you want to enter a right node (1/0): "
25         << "for the node -- " << Node->word << ": " << Node->definition << "
26     -- "
27     Take Input of choice
28     if (choice == 1)
29         new_node = new WordNode
30         display - "Enter the data: "
31         Take Input of new_node->word
32         Take Input of new_node->definition
33         Node->right = new_node
34         create_recursive(new_node)
```

8.2 Display

```
1 void bfs()
2 {
3     WordNode *temp = root
4     queue<WordNode *> q
5     q.push(temp)
6     while (!q.empty())
7     {
8         temp = q.front()
9         q.pop()
10        display - temp->word << " : " << temp->definition << endl
11        if (temp->left != NULL)
12        {
13            q.push(temp->left)
14        }
15        if (temp->right != NULL)
16        {
17            q.push(temp->right)
18        }
19    }
20 }
```

8.3 Delete

```
1 void delete_node(WordNode *temp, string word)
2
3     WordNode *parent = NULL;
4     while (temp != NULL)
5         if (temp->word == word)
6             break;
7         else
8             parent = temp;
9             if (strcmp(word.c_str(), temp->word.c_str()) < 0)
10                 temp = temp->left;
11             else
12                 temp = temp->right;
13     if (temp == NULL)
14         Print - "Word not found" - endl;
15         return;
16     else
17         // no child node.
18         if (temp->left == NULL && temp->right == NULL)
19             if (parent->left == temp)
20                 parent->left = NULL;
21             else
22                 parent->right = NULL;
23             delete temp;
24
25     // 1 Child case right.
26     else if (temp->left == NULL)
27         if (parent->left == temp)
28             parent->left = temp->right;
29         else
30             parent->right = temp->right;
31         delete temp;
32
```

```
33     // 1 Child case left.
34     else if (temp->right == NULL)
35         if (parent->left == temp)
36             parent->left = temp->left;
37         else
38             parent->right = temp->left;
39         delete temp;
40     else
41         WordNode *temp1 = temp->right;
42         while (temp1->left != NULL)
43             temp1 = temp1->left;
44         temp->word = temp1->word;
45         temp->definition = temp1->definition;
46         delete_node(temp->right, temp1->word);
```

8.4 Mirror Image

```
1  void mirror_recursive(WordNode *temp)
2  {
3      if (temp == NULL)
4      {
5          return
6      }
7      else
8      {
9          WordNode *temp1
10         // Swapping
11         temp1 = temp->left
12         temp->left = temp->right
13         temp->right = temp1
14
15         // Recursively calling the function
16         mirror_recursive(temp->left)
17         mirror_recursive(temp->right)
18     }
19 }
```

8.5 Copy

```
1  WordNode *create_copy_recursive(WordNode *temp)
2  {
3      if (temp == NULL)
4      {
5          return NULL
6      }
7      else
8      {
9          WordNode *new_node = new WordNode
10         new_node->word = temp->word
11         new_node->definition = temp->definition
12         new_node->left = create_copy_recursive(temp->left)
13         new_node->right = create_copy_recursive(temp->right)
14         return new_node
15     }
16 }
```


9 Time Complexity

9.1 Create

- **Time Complexity Worst Case:**
 $\mathcal{O}(n^2)$
- **Time Complexity Best Case:**
 $\mathcal{O}(\log(n))$

9.2 Display

- **Time Complexity:**
 $\mathcal{O}(n)$

9.3 Delete

- **Time Complexity:**
 $\mathcal{O}(h)$

h

is the height of the Binary search tree.

9.4 Mirror Image

- **Time Complexity:**
 $\mathcal{O}(n)$

9.5 Copy

- **Time Complexity:**
 $\mathcal{O}(n)$

10 Code

10.1 Program

```
1 #include <iostream>
2 #include <queue>
3 #include <stack>
4 #include <string.h>
5 using namespace std;
6
7 class WordNode
8 {
9     string word;
10    string definition;
11    WordNode *left;
12    WordNode *right;
13    friend class BinarySearchTree;
```

```
14 };
15
16 class BinarySearchTree
17 {
18 public:
19     WordNode *root;
20     BinarySearchTree()
21     {
22         root = NULL;
23     }
24     void create_root()
25     {
26         root = new WordNode;
27         cout << "Enter the data: " << endl;
28         cin >> root->word;
29         cin >> root->definition;
30         root->left = NULL;
31         root->right = NULL;
32         create_recursive(root);
33     }
34     void create_recursive(WordNode *Node)
35     {
36         int choice = 0;
37         WordNode *new_node;
38         cout << "Enter if you want to enter a left node (1/0): "
39              << "for the node -- " << Node->word << ": " << Node->definition << "
40              -- ";
41         cin >> choice;
42         if (choice == 1)
43         {
44             new_node = new WordNode;
45             cout << "Enter the data: ";
46             cin >> new_node->word;
47             cin >> new_node->definition;
48             Node->left = new_node;
49             create_recursive(new_node);
50         }
51         cout << "Enter if you want to enter a right node (1/0): "
52              << "for the node -- " << Node->word << ": " << Node->definition << "
53              -- ";
54         cin >> choice;
55         if (choice == 1)
56         {
57             new_node = new WordNode;
58             cout << "Enter the data: ";
59             cin >> new_node->word;
60             cin >> new_node->definition;
61             Node->right = new_node;
62             create_recursive(new_node);
63         }
64     }
65     void create_root_and_tree_iteratively()
66     {
67         WordNode *temp, *current_word;
68         int choice = 0;
69         root = new WordNode;
70         cout << "Enter the Word: ";
71         cin >> root->word;
72         cout << "Enter the definition of the word: ";
```

```
71     cin >> root->definition;
72     bool flag = false;
73     cout << "Do you want to enter more Nodes? (0/1) " << endl;
74     cin >> choice;
75     while (choice == 1)
76     {
77         temp = root;
78         flag = false;
79         current_word = new WordNode;
80         cout << "Enter the Word: ";
81         cin >> current_word->word;
82         cout << "Enter the definition of the word: ";
83         cin >> current_word->definition;
84
85         while (!flag)
86         {
87             if (strcmp(current_word->word.c_str(), temp->word.c_str()) <= 0)
88             {
89                 if (temp->left == NULL)
90                 {
91                     temp->left = current_word;
92                     flag = true;
93                 }
94                 else
95                 {
96                     temp = temp->left;
97                 }
98             }
99             else
100             {
101                 if (temp->right == NULL)
102                 {
103                     temp->right = current_word;
104                     flag = true;
105                 }
106                 else
107                 {
108                     temp = temp->right;
109                 }
110             }
111         }
112         cout << "Do you want to enter another word? (1/0): ";
113         cin >> choice;
114     }
115 }
116
117 // breadth First Search using queue.
118 void bfs()
119 {
120     WordNode *temp = root;
121     queue<WordNode *> q;
122     q.push(temp);
123     while (!q.empty())
124     {
125         temp = q.front();
126         q.pop();
127         cout << temp->word << " : " << temp->definition << endl;
128         if (temp->left != NULL)
129         {
```

```
130         q.push(temp->left);
131     }
132     if (temp->right != NULL)
133     {
134         q.push(temp->right);
135     }
136 }
137 }
138
139 WordNode *create_copy_recursive(WordNode *temp)
140 {
141     if (temp == NULL)
142     {
143         return NULL;
144     }
145     else
146     {
147         WordNode *new_node = new WordNode;
148         new_node->word = temp->word;
149         new_node->definition = temp->definition;
150         new_node->left = create_copy_recursive(temp->left);
151         new_node->right = create_copy_recursive(temp->right);
152         return new_node;
153     }
154 }
155
156 WordNode *create_copy_iteratively(WordNode *temp)
157 {
158     // We have to create a queue to pop things
159     queue<WordNode *> q;
160     WordNode *copied_tree;
161     WordNode *new_node = new WordNode;
162     new_node->word = temp->word;
163     new_node->definition = temp->definition;
164     q.push(new_node);
165     while (!q.empty())
166     {
167         copied_tree = q.front();
168         q.pop();
169         if (temp->left != NULL)
170         {
171             WordNode *new_node1 = new WordNode;
172             new_node1->word = temp->left->word;
173             new_node1->definition = temp->left->definition;
174             copied_tree->left = new_node1;
175             q.push(new_node1);
176         }
177         if (temp->right != NULL)
178         {
179             WordNode *new_node1 = new WordNode;
180             new_node1->word = temp->right->word;
181             new_node1->definition = temp->right->definition;
182             copied_tree->right = new_node1;
183             q.push(new_node1);
184         }
185         temp = temp->left;
186     }
187     return copied_tree;
188 }
```

```
189
190 void mirror_recursive(WordNode *temp)
191 {
192     if (temp == NULL)
193     {
194         return;
195     }
196     else
197     {
198         WordNode *temp1;
199         // Swapping
200         temp1 = temp->left;
201         temp->left = temp->right;
202         temp->right = temp1;
203
204         // Recursively calling the function
205         mirror_recursive(temp->left);
206         mirror_recursive(temp->right);
207     }
208 }
209
210 void mirror_iterative(WordNode *node)
211 {
212     WordNode *temp;
213     queue<WordNode *> q;
214     q.push(node);
215     while (!q.empty())
216     {
217         temp = q.front();
218         q.pop();
219         WordNode *temp1;
220
221         // Swapping
222         temp1 = temp->left;
223         temp->left = temp->right;
224         temp->right = temp1;
225
226         if (temp->left != NULL)
227         {
228             q.push(temp->left);
229         }
230         if (temp->right != NULL)
231         {
232             q.push(temp->right);
233         }
234     }
235 }
236
237 WordNode *create_mirror_tree_recursive()
238 {
239     WordNode *mirror_tree = create_copy_recursive(root);
240     mirror_recursive(mirror_tree);
241     return mirror_tree;
242 }
243
244 WordNode *create_mirror_tree_iterative()
245 {
246     WordNode *mirror_tree = create_copy_recursive(root);
247     mirror_iterative(mirror_tree);
```

```
248     return mirror_tree;
249 }
250
251 void delete_node(WordNode *temp, string word)
252 {
253     WordNode *parent = NULL;
254     while (temp != NULL)
255     {
256         if (temp->word == word)
257         {
258             break;
259         }
260         else
261         {
262             parent = temp;
263             if (strcmp(word.c_str(), temp->word.c_str()) < 0)
264             {
265                 temp = temp->left;
266             }
267             else
268             {
269                 temp = temp->right;
270             }
271         }
272     }
273     if (temp == NULL)
274     {
275         cout << "Word not found" << endl;
276         return;
277     }
278     else
279     {
280         // no child node.
281         if (temp->left == NULL && temp->right == NULL)
282         {
283             if (parent->left == temp)
284             {
285                 parent->left = NULL;
286             }
287             else
288             {
289                 parent->right = NULL;
290             }
291             delete temp;
292         }
293         // 1 Child case right.
294         else if (temp->left == NULL)
295         {
296             if (parent->left == temp)
297             {
298                 parent->left = temp->right;
299             }
300             else
301             {
302                 parent->right = temp->right;
303             }
304             delete temp;
305         }
306         // 1 Child case left.
```

```
307         else if (temp->right == NULL)
308         {
309             if (parent->left == temp)
310             {
311                 parent->left = temp->left;
312             }
313             else
314             {
315                 parent->right = temp->left;
316             }
317             delete temp;
318         }
319         else
320         {
321             WordNode *temp1 = temp->right;
322             while (temp1->left != NULL)
323             {
324                 temp1 = temp1->left;
325             }
326             temp->word = temp1->word;
327             temp->definition = temp1->definition;
328             delete_node(temp->right, temp1->word);
329         }
330     }
331 }
332
333 void inorder_iterative(WordNode *temp)
334 {
335     if (!temp)
336     {
337         return;
338     }
339
340     stack<WordNode *> s;
341     WordNode *current = temp;
342
343     while (current != NULL || s.empty() == false)
344     {
345         while (current != NULL)
346         {
347             s.push(current);
348             current = current->left;
349         }
350         current = s.top();
351         s.pop();
352         cout << current->word << " : " << current->definition << endl;
353         current = current->right;
354     }
355 }
356 void preorder_iterative(WordNode *temp)
357 {
358     if (!temp)
359     {
360         return;
361     }
362
363     stack<WordNode *> s;
364     s.push(temp);
365 }
```

```
366     while (s.empty() == false)
367     {
368         WordNode *current = s.top();
369         cout << current->word << " : " << current->definition << endl;
370         s.pop();
371
372         if (current->right)
373         {
374             s.push(current->right);
375         }
376         if (current->left)
377         {
378             s.push(current->left);
379         }
380     }
381 }
382 void postorder_iterative(WordNode *temp)
383 {
384     if (!temp)
385     {
386         return;
387     }
388
389     stack<WordNode *> s1;
390     stack<WordNode *> s2;
391
392     s1.push(temp);
393
394     while (s1.empty() == false)
395     {
396         WordNode *current = s1.top();
397         s1.pop();
398         s2.push(current);
399
400         if (current->left)
401         {
402             s1.push(current->left);
403         }
404         if (current->right)
405         {
406             s1.push(current->right);
407         }
408     }
409     while (s2.empty() == false)
410     {
411         WordNode *current = s2.top();
412         cout << current->word << " : " << current->definition << endl;
413         s2.pop();
414     }
415 }
416 };
417
418 int main()
419 {
420     int choice = 0;
421     string word;
422     BinarySearchTree main_tree, mirror_tree, copy_tree;
423
424     while (choice != 10)
```



```
425 {
426     cout << "\nWhat would like to do? " << endl;
427     cout << "\n\nWelcome to ADS Assignment 2 - Binary Tree Traversals\n\nWhat
would you like to do? " << endl;
428     cout << "1. Create a Binary Search Tree"
429         << endl;
430     cout << "2. Traverse the Tree Inorder Iteratively"
431         << endl;
432     cout << "3. Traverse the Tree PreOrder Iteratively"
433         << endl;
434     cout << "4. Traverse the Tree PostOrder Iteratively"
435         << endl;
436     cout << "5. Traverse it using BFS"
437         << endl;
438     cout << "6. Create a Copy of the tree Recursively and Iteratively"
439         << endl;
440     cout << "7. Create a Mirror of the Tree Recursively"
441         << endl;
442     cout << "8. Create a Mirror of the Tree Iteratively"
443         << endl;
444     cout << "9. Delete a Node from the Tree"
445         << endl;
446     cout << "10. Exit" << endl
447         << endl;
448
449     cin >> choice;
450     switch (choice)
451     {
452     case 1:
453         main_tree.create_root_and_tree_iteratively();
454         break;
455     case 2:
456         cout << "Traversing through the Binary Tree Inorder Iteratively: " <<
endl;
457         main_tree.inorder_iterative(main_tree.root);
458         break;
459     case 3:
460         cout << "Traversing through the Binary Tree PreOrder Iteratively: " <<
endl;
461         main_tree.preorder_iterative(main_tree.root);
462         break;
463     case 4:
464         cout << "Traversing through the Binary Tree PostOrder Iteratively: "
<< endl;
465         main_tree.postorder_iterative(main_tree.root);
466         break;
467     case 5:
468         cout << "Traversing through the Binary Tree using BFS: " << endl;
469         main_tree.bfs();
470         break;
471     case 6:
472         cout << "Creating a copy of the tree" << endl;
473         copy_tree.root = copy_tree.create_copy_recursive(main_tree.root);
474         cout << "Traversing via Breadth First Search: " << endl;
475         copy_tree.bfs();
476         break;
477     case 7:
478         cout << "Creating a mirror of the tree" << endl;
479         mirror_tree.root = main_tree.create_mirror_tree_recursive();
```

```
480         cout << "Traversing via Breadth First Search: " << endl;
481         mirror_tree.bfs();
482         break;
483     case 8:
484         cout << "Creating a mirror of the tree Iteratively" << endl;
485         mirror_tree.root = main_tree.create_mirror_tree_iterative();
486         cout << "Traversing via Breadth First Search: " << endl;
487         mirror_tree.bfs();
488         break;
489     case 9:
490         cout << "Enter the word you want to delete: " << endl;
491         cin >> word;
492         main_tree.delete_node(main_tree.root, word);
493         cout << "Traversing through the Binary Tree Inorder Iteratively: " <<
endl;
494         main_tree.inorder_iterative(main_tree.root);
495         break;
496     case 10:
497         cout << "Exiting the program" << endl;
498         break;
499     default:
500         cout << "Invalid Choice" << endl;
501         break;
502     }
503 }
504 }
```

10.2 Input and Output

```
1 What would like to do?
2
3
4 Welcome to ADS Assignment 2 - Binary Tree Traversals
5
6 What would you like to do?
7 1. Create a Binary Search Tree
8 2. Traverse the Tree Inorder Iteratively
9 3. Traverse the Tree PreOrder Iteratively
10 4. Traverse the Tree PostOrder Iteratively
11 5. Traverse it using BFS
12 6. Create a Copy of the tree Recursively
13 7. Create a Mirror of the Tree Recursively
14 8. Exit
15
16 1
17 Enter the Word: apple
18 Enter the definition of the word: fruit
19 Do you want to enter more Nodes? (0/1)
20
21 1
22 Enter the Word: banana
23 Enter the definition of the word: fruit
24 Do you want to enter another word? (1/0): 1
25 Enter the Word: keyboard
26 Enter the definition of the word: input
27 Do you want to enter another word? (1/0): 1
28 Enter the Word: pears
29 Enter the definition of the word: fruit
30 Do you want to enter another word? (1/0): 1
```

Advanced Data Structures - Assignment 3

```
31 Enter the Word: bottle
32 Enter the definition of the word: water
33 Do you want to enter another word? (1/0): 1
34 Enter the Word: charger
35 Enter the definition of the word: charging
36 Do you want to enter another word? (1/0): 1
37 Enter the Word: monitor
38 Enter the definition of the word: see
39 Do you want to enter another word? (1/0): 1
40 Enter the Word: paper
41 Enter the definition of the word: 1
42 Do you want to enter another word? (1/0): 1
43 Enter the Word: pen
44 Enter the definition of the word: writing
45 Do you want to enter another word? (1/0): 1
46 Enter the Word: phone
47 Enter the definition of the word: scrolling
48 Do you want to enter another word? (1/0): 0
49
50 What would like to do?
51
52
53 Welcome to ADS Assignment 2 - Binary Tree Traversals
54
55 What would you like to do?
56 1. Create a Binary Search Tree
57 2. Traverse the Tree Inorder Iteratively
58 3. Traverse the Tree PreOrder Iteratively
59 4. Traverse the Tree PostOrder Iteratively
60 5. Traverse it using BFS
61 6. Create a Copy of the tree Recursively
62 7. Create a Mirror of the Tree Recursively
63 8. Exit
64
65 2
66 Traversing through the Binary Tree Inorder Iteratively:
67 apple : fruit
68 banana : fruit
69 bottle : water
70 charger : charging
71 keyboard : input
72 monitor : see
73 paper : 1
74 pears : fruit
75 pen : writing
76 phone : scrolling
77
78 What would like to do?
79
80
81 Welcome to ADS Assignment 2 - Binary Tree Traversals
82
83 What would you like to do?
84 1. Create a Binary Search Tree
85 2. Traverse the Tree Inorder Iteratively
86 3. Traverse the Tree PreOrder Iteratively
87 4. Traverse the Tree PostOrder Iteratively
88 5. Traverse it using BFS
89 6. Create a Copy of the tree Recursively
```

```
90 7. Create a Mirror of the Tree Recursively
91 8. Exit
92
93 5
94 Traversing through the Binary Tree using BFS:
95 apple : fruit
96 banana : fruit
97 keyboard : input
98 bottle : water
99 pears : fruit
100 charger : charging
101 monitor : see
102 pen : writing
103 paper : 1
104 phone : scrolling
105
106 What would like to do?
107
108
109 Welcome to ADS Assignment 2 - Binary Tree Traversals
110
111 What would you like to do?
112 1. Create a Binary Search Tree
113 2. Traverse the Tree Inorder Iteratively
114 3. Traverse the Tree PreOrder Iteratively
115 4. Traverse the Tree PostOrder Iteratively
116 5. Traverse it using BFS
117 6. Create a Copy of the tree Recursively
118 7. Create a Mirror of the Tree Recursively
119 8. Exit
120
121 6
122 Creating a copy of the tree
123 Traversing through the Binary Tree Inorder Iteratively:
124 apple : fruit
125 banana : fruit
126 bottle : water
127 charger : charging
128 keyboard : input
129 monitor : see
130 paper : 1
131 pears : fruit
132 pen : writing
133 phone : scrolling
134 Traversing via Breadth First Search:
135 apple : fruit
136 banana : fruit
137 keyboard : input
138 bottle : water
139 pears : fruit
140 charger : charging
141 monitor : see
142 pen : writing
143 paper : 1
144 phone : scrolling
145
146 What would like to do?
147
148
```

```
149 Welcome to ADS Assignment 2 - Binary Tree Traversals
150
151 What would you like to do?
152 1. Create a Binary Search Tree
153 2. Traverse the Tree Inorder Iteratively
154 3. Traverse the Tree PreOrder Iteratively
155 4. Traverse the Tree PostOrder Iteratively
156 5. Traverse it using BFS
157 6. Create a Copy of the tree Recursively
158 7. Create a Mirror of the Tree Recursively
159 8. Exit
160
161 7
162 Creating a mirror of the tree
163 Traversing through the Binary Tree Inorder Iteratively:
164 phone : scrolling
165 pen : writing
166 pears : fruit
167 paper : 1
168 monitor : see
169 keyboard : input
170 charger : charging
171 bottle : water
172 banana : fruit
173 apple : fruit
174 Traversing via Breadth First Search:
175 apple : fruit
176 banana : fruit
177 keyboard : input
178 pears : fruit
179 bottle : water
180 pen : writing
181 monitor : see
182 charger : charging
183 phone : scrolling
184 paper : 1
185
186 What would like to do?
187
188
189 Welcome to ADS Assignment 2 - Binary Tree Traversals
190
191 What would you like to do?
192 1. Create a Binary Search Tree
193 2. Traverse the Tree Inorder Iteratively
194 3. Traverse the Tree PreOrder Iteratively
195 4. Traverse the Tree PostOrder Iteratively
196 5. Traverse it using BFS
197 6. Create a Copy of the tree Recursively
198 7. Create a Mirror of the Tree Recursively
199 8. Exit
200
201 8
202 Exiting the program
```

11 Conclusion

Thus, implemented Dictionary using Binary search tree.

12 FAQ

1. Explain application of BST

The Applications of Binary Search Tree are:

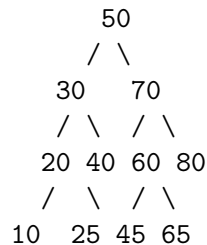
- (a) Binary Search Tree is used to implement dictionaries.
- (b) Binary Search Tree is used to implement priority queues.
- (c) Binary Search Tree is used to implement disjoint sets.
- (d) Binary Search Tree is used to implement sorting algorithms.
- (e) Binary Search Tree is used to implement expression trees.
- (f) Binary Search Tree is used to implement Huffman coding.
- (g) Binary Search Tree is used to implement B-trees.
- (h) Binary Search Tree is used to implement red-black trees.

2. Explain with example deletion of a node having two child.

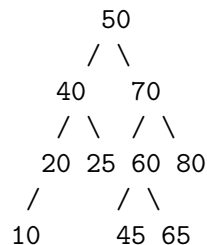
If a node has two children, then we need to find the inorder successor of the node. The inorder successor is the smallest in the right subtree or the largest in the left subtree. After finding the inorder successor, we copy the contents of the inorder successor to the node and delete the inorder successor. Note that the inorder predecessor can also be used.

An Example would be:

Let us consider the following BST as an example.



Deleting 30 will be done in following steps. 1. Find inorder successor of 30. 2. Copy contents of the inorder successor to 30. 3. Delete the inorder successor. 4. Since inorder successor is 40 which has no left child, we simply make right child of 30 as the new right child of 20.



3. Define skewed binary tree.

A binary tree is said to be skewed if all of its nodes have only one child. A skewed binary tree can be either left or right skewed. A left skewed binary tree is a binary tree in which all the nodes have only left child. A right skewed binary tree is a binary tree in which all the nodes have only right child.