A Mini Project Report

*on*

**Dictionary using Binary Search Tree**

In Subject: Data structure and Discrete Mathematics

*by*

*Sangini Sangle-271049*

*Nidhi Sanklecha-271051*

*Ayush Shende-271056*

*Suraj Wadikar-271064*

*Abhishek Walunj-271065*



Department of Artificial Intelligence and Data Science

VIIT

**2020-2021**

**Contents**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Topic** | | **Page No.** |
|  |  | |  |
| **Chapter-1** | **Introduction** | |  |
|  | 1.1 | Introduction | 3 |
|  | 1.2 | Design & Problem Statement | 5 |
|  | 1.3 | Proposed work | 6 |
| **Chapter-2** | **Methodology** | |  |
|  | 2.1 | Approach | 18 |
|  | 2.2 | Platform and Technology | 23 |
|  | 2.3 | Outcomes | 24 |
|  | 2.4 | Challenges | 26 |
| **Chapter-3** | **Conclusion** | | 27 |
|  | **References** | | 28 |

**Introduction**

* 1. **Introduction**

A dictionary is a data structure that maps keys to values and allows querying the value for a given key. A binary search tree also has keys (which are used for the search) and can also have further payload data (the values). So a BST with payload data is actually already a dictionary.

In this project, we will use a binary search tree (BST) representation to implement a simple dictionary ADT.

The implementation must use a binary search tree (BST) to maintain the contents of the dictionary. Dictionary items are represented using BST Node objects which must be allocated dynamically.

To simplify, a BST Node elides the value part of a key-value pair and only stores the symbolic key in the form of a C-style string. The key symbol must be dynamically allocated , too.

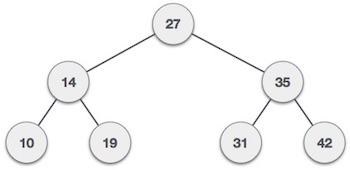
Search is a fundamental process in data processing . The search process is to find a certain value (data) in a set of data of the same type

Items, such as names, numbers, etc. can be stored in memory in a sorted order called binary search trees or BSTs. And some of these data structures can automatically balance their height when arbitrary items are inserted. Therefore, they are known as self-balancing BSTs.

A Binary Search Tree is a sorted binary tree in which all the nodes have following two properties−

1. The right sub-tree of a node has a key greater than to its parent node's key.
2. The left sub-tree of a node has a key less than or equal to its parent node's key.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

The motivation is to obtain a data structure with more power than a contiguous array in memory.

To find the meaning of words, there is no need to open a dictionary from the beginning page to the end of the page one by one, but look for it by separating or dividing it into two parts

The main reason to use a binary search tree is the fact that it extends the capability of a normal array.

Implementing a binary search tree is useful in any situation where the elements can be compared in a less than / greater than manner. For our example, we'll use alphabetical order as our criteria for whether an element is greater than or less than another element

* 1. **Design & Problem Statement**

Implementing a Dictionary using Binary Search Tree Algorithm

And perform following operations on a Dictionary. −

Search − Searches an element in a Dictionary.

Insert − Inserts an element in a Dictionary.

Delete – Delete an element in a Dictionary.

View – Display all elements in a Dictionary.

When we try to implement abstract data type Dictionary, then the node associates with values. A dictionary is basically a set of keys, which must be elements drawn from a total ordering. There may be additional information, which are associated with each key, but it does not lead to any conceptual comprehension.

If the dictionary is implemented using trees, then each node will hold unique keys. Here for each node u in the tree, every key is u.l is strictly smaller than u.k. And every key in u.r, is strictly larger than u.k. A tree is organized according to this invariant as referred to as a binary search tree.

One of the major advantage of this invariant is that, sorted list of keys can be found in linear time using in-order traversal. This can be defined recursively as follows − One empty tree, do nothing, otherwise recurs on the left subtree first, take the root, and report it. Then recur to the right subtree.

We can do multiple operations for binary search tree. The searching cane be done based on the height of the tree. Searching is more important operation of all other operations.

**1.3 Proposed work**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct DICTnode

{

char word[128], meaning[256];

struct DICTnode \*left, \*right;

};

struct DICTnode \*root = NULL;

// creating a node

struct DICTnode \*createNode(char \*word, char \*meaning)

{

struct DICTnode \*newnode; // declaring node of DICTnode structure

newnode = (struct DICTnode \*)malloc(sizeof(struct DICTnode)); // allocating memory in heap for declared node

strcpy(newnode->word, word); // setting value of word

strcpy(newnode->meaning, meaning); // setting value of meaning

newnode->left = newnode->right = NULL; // pointing its right and left to null

printf("Word added to dict. Successfully\n");

return newnode; // returning created node

}

// function to insert word in bst

void insert(char \*word, char \*meaning)

{

struct DICTnode \*parent = NULL, \*current = NULL, \*newnode = NULL; // declaring nodes

int res = 0; // created flag to add word in right node or left node

if (!root) // if root is equal to null then we directly create new node

{

root = createNode(word, meaning);

return;

}

for (current = root; current != NULL;

current = (res > 0) ? current->right : current->left) // updating current w.r.t res means if res is greater than 0 it will update it to right node else left node

{

res = strcasecmp(word, current->word); // it will update res

if (res == 0)

{

printf("Duplicate entry!!\n");

return;

}

parent = current; // updating parent node with current node

}

newnode = createNode(word, meaning); // i had parent node and im creting its childnode

res > 0 ? (parent->right = newnode) : (parent->left = newnode); // if res is greater than 0 then it will be added as right node either left node

return;

}

/\*

there are three cases in deletion of particular node

1. The node is at leaf node

2. The node is non-leaf node

3. The node is root node

\*/

// deleting a word from dict

void deleteNode(char \*str)

{

struct DICTnode \*parent = NULL, \*current = NULL, \*temp = NULL; // declaring nodes

int flag = 0, res = 0; // created flags

if (!root) // if root is null we do this

{

printf("Dict. is Empty!!\n");

return;

}

current = root; // if root is not iqual to null then we will update it will current

while (1) // while true

{

res = strcasecmp(current->word, str); // it will update res

if (res == 0) // we got the word and now it's time to delete it

break;

flag = res; // update the flag value with res

parent = current; // updating parent will current node

current = (res > 0) ? current->left : current->right; // we are updating current w.r.t res value

if (current == NULL) // if current is null then we will simply return

printf("Word is Not Present in Dict.\n");

return;

}

/\* deleting leaf node \*/

if (current->right == NULL)

{

if (current == root && current->left == NULL) // if the leaf node is a root node

{

free(current);

root = NULL;

printf("Word Deleted Successfully.\n");

return;

}

else if (current == root) // current node is a root node but it has left node

{

root = current->left; // updating root with left node

free(current); // deleting current node

printf("Word Deleted Successfully.\n");

return;

}

flag > 0 ? (parent->left = current->left) : (parent->right = current->left);

}

else

{

/\* delete node with single child \*/

temp = current->right;

if (!temp->left) // if left is null of temp. node

{

temp->left = current->left; // set that left node to current node's left

if (current == root)

{

root = temp;

free(current);

printf("Word Deleted Successfully.\n");

return;

}

flag > 0 ? (parent->left = temp) : (parent->right = temp);

}

else

{

/\* delete node with two children \*/

struct DICTnode \*successor = NULL;

while (1)

{

successor = temp->left; // setting the successor

if (!successor->left) // if it is null

break;

temp = successor; // updating temp with successor

}

temp->left = successor->right;

successor->left = current->left; // setting successor

successor->right = current->right;

if (current == root) //deleting condition

{

root = successor;

free(current); // freeing memory

printf("Word Deleted Successfully.\n");

return;

}

(flag > 0) ? (parent->left = successor) : (parent->right = successor);

}

}

free(current);

printf("Word Deleted Successfully.\n");

return;

}

// searching particular word

void findElement(char \*str)

{

struct DICTnode \*temp = NULL; // declaring temporary node

int flag = 0, res = 0; // initializing flags

if (root == NULL) // if tree is Empty

{

printf("Dict. Is Empty.\n");

return;

}

temp = root; // if tree is not empty then set it's root to temp

while (temp) // traversing the temp

{

if ((res = strcasecmp(temp->word, str)) == 0) // if we get a word

{

printf("Word : %s", str);

printf("Meaning: %s", temp->meaning);

flag = 1;

break;

}

temp = (res > 0) ? temp->left : temp->right; // if we does not get word

}

if (!flag) // if flag is 0 then we can say that we dont have word

printf("Search Element not found in Dict.\n");

return;

}

/\*

in inOrder Traversal program we do

1. apply inorder on left node

2. print root node

3. apply inorder on right node

in preOrder Traversal program we do

1. print root node

2. apply preOrder on left node

3. apply preOrder on right node

in postOrder Traversal program we do

1. apply postOrder on left node

2. apply postOrder on right node

3. print root node

simply we can say that its a game of recursion

\*/

// traversing whole dict.

void inorderTraversal(struct DICTnode \*myNode)

{

if(!root){

printf("Dict. Is Empty.\n");

}

if (myNode) // if node is present

{

inorderTraversal(myNode->left); // 1st step of inorder

printf("Word : %s", myNode->word); // 2nd step of inorder

printf("Meaning : %s", myNode->meaning); // 2nd step of inorder

printf("\n");

inorderTraversal(myNode->right); // 3rd step of inorder

}

return;

}

int main()

{

int ch;

char str[128], meaning[256];

while (1)

{

printf("\n1. Insertion\t2. Deletion\n");

printf("3. Searching\t4. Traversal\n");

printf("5. Exit\nEnter ur choice:");

scanf("%d", &ch); // getting input from user

getchar();

switch (ch) // switch statement on char

{

case 1:

printf("Word to insert:");

fgets(str, 100, stdin); // getting input from user

printf("Meaning:");

fgets(meaning, 256, stdin); // getting input from user

insert(str, meaning); // calling insert function

break;

case 2:

printf("Enter the word to delete:");

fgets(str, 100, stdin); // getting input from user

deleteNode(str); // calling delete function

break;

case 3:

printf("Enter the search word:");

fgets(str, 100, stdin); // getting input from user

findElement(str); // calling search function

break;

case 4:

inorderTraversal(root); // calling traversal function

break;

case 5:

exit(0); // exit opt.

default:

printf("You have entered wrong option\n");

break;

}

}

return 0;

}

**Methodology**

**2.1 Approach**

Dictionary can be implemented using binary search tree. A binary search tree is a binary tree such that each node stores a key of a dictionary.

Key 'k' of a node is always greater than the keys present in its left sub tree.

Similarly, key 'k' of a node is always lesser than the keys present in its right sub tree.

Example:

well

/ \

bus xmas

/ \ \

air aero zebra

In the above example, keys present at the left sub-tree of the root node are lesser than the key of the root node. And also, the keys present at the right sub-tree are greater than the key of the root node.

To insert an element in a binary search tree, check whether the root is present or not. If root is absent, then the new node is the root.

If root node is present, check whether the key in new node is greater than or lesser than the key in root node.

If key in new node is less than the key in root, then traverse the left sub-tree recursively until we reach the leaf node. Then, insert the new node to the left(newnode < leaf)/right(newnode > leaf) of the leaf.

If the key in new node is greater than the key in root, then traverse the right sub-tree recursively until we reach the leaf node. Then, insert the new node to the left(newnode < leaf)/right of the leaf

InsertionInBST(T, newnode)

y<-NULL

x <- root[T]

while x != NULL

y<-x

if key[z] < key

then x <- left[x]

else x <- right[x]

parent[newnode] <- y

if y == NULL

then root[T] <- newnode

else if key[newnode] < key[y]

then left[y] <- newnode

else right[y] <- newnode

Insert "yell" to the below binary search tree.

workload

/ \

bus xmas

/ \ \

air aero zebra

workload

/ \

bus xmas

/ \ / \

air aero yell zebra

To delete a node from binary search tree, we have three different cases.

Node X has no children

Node X has one child

Node X has two children

Case 1:

If X has no children

workload

/ \

bus xmas

/ \ / \

air aero yell zebra

Delete "zebra" from above binary search tree.

workload

/ \

bus xmas

/ \ /

air aero yell

Case 2:

If X has only one child, then delete x and point the parent of x to the child of x.

workload

/ \

bus xmas

/ \ /

air aero yell

Delete "xmas" from the above binary search tree.

workload

/ \

bus yell

/ \

air aero

Case 3:

If X has two children, then find its successor 'S'. Remove 'S' from the binary search tree. And replace X with 'S'

workload

/ \

bus xmas

/ \ / \

air aero yell zebra

Remove "workload" from the above binary search tree. The successor for "workload"(smallest element in the right subtree of "workload") is "yell". Remove it and replace "workload with "yell".

yell

/ \

bus xmas

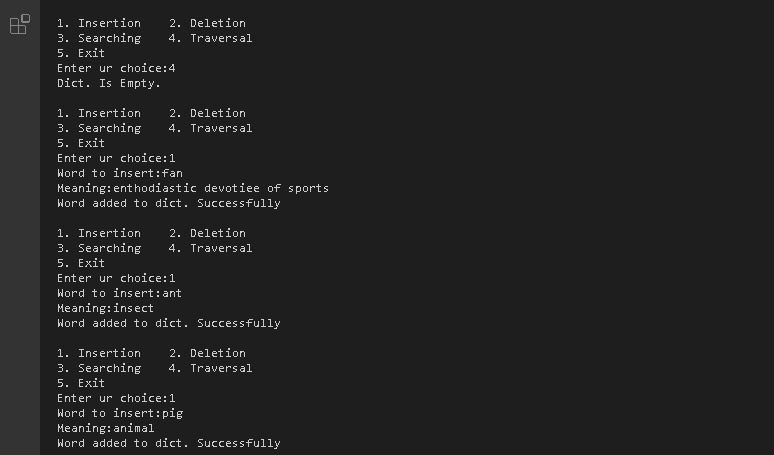
/ \ \

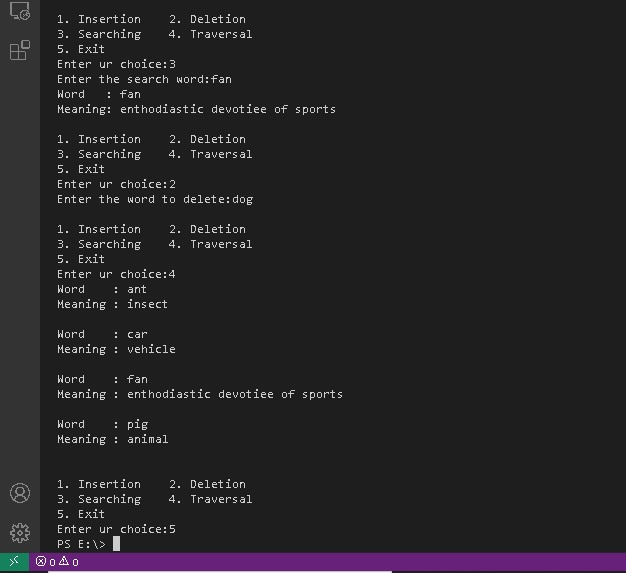
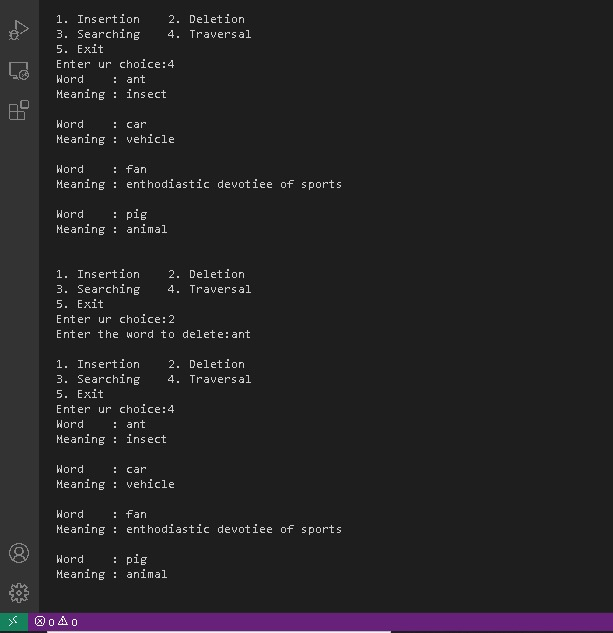
air aero zebra

**2.2 Platform and Technology**

1. GCC compiler is used to compile source code and create exe file for source code
2. An source code editor(VS code, Code Blocks, etc.) can be used for sake of user's simplicity

**2.3 Outcomes**

****

****

**2.4 Challenges**

Programming binary search algorithm is error prone and difficult.

The main Challenge is that we should always implement a balanced binary search Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.

**3. Conclusion**

From the results of the discussions that have been carried out, it can be concluded as follows:

1.The search process on the Dictionary using binary search tree algorithms works well.

2.The Binary Search Algorithm successfully finds every word that is searched if the word is available in the dictionary. The basic principle of the binary search algorithm is to repeat the search space repeatedly until data is found or until the search space cannot be shared (data may not exist).

3. The purpose of searching using a binary search algorithm is to reduce the number of operations that must be compared between the data sought and data in the Dictionary, especially for large amounts of data.

**4. References**

1. Abdulloh, Rohi. AgusSyahputra (Ed). 2015. Super Dahsyat Responsive Web Design Dengan Foundation 5. Cirebon: CV. ASFA SOLUTION.
2. A.S, Rossa & Shalahuddin, M. 2013. Rekayasa Perangkat Lunak. Bandung: INFORMATIKA
3. Binarso, YusiArdi. Eko Adi Sarwoko dan Nurdin Bahtiar. 2012. Pembangunan SistemInformasi Alumni berbasis Web pada Program Studi Teknik Informatika Universitas Diponegoro. UniversitasDiponegoro
4. Munir, Rinaldi. Leony Lidya. 2016. Algoritma dan Pemograman Dalam Bahasa Pascal, C, dan C++. Bandung: Informatika.
5. Roshadi, Aditya Yanuar. 2014. Perancangan Istilah Teknologi Informasi Berbasis Web MenggunakanMetode Web-QEM. Universitas NegeriYogjakarta.
6. <http://sandsoftwaresound.net/wp-content/uploads/2013/07/S05_project7.html>
7. <https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm>
8. <https://www.youtube.com/watch?v=JS297Hq4Mz8>
9. <https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BSTDict.html>
10. <https://www.researchgate.net/publication/335657221_Digital_Dictionary_Using_Binary_Search_Algorithm>
11. <https://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson4_3.htm>