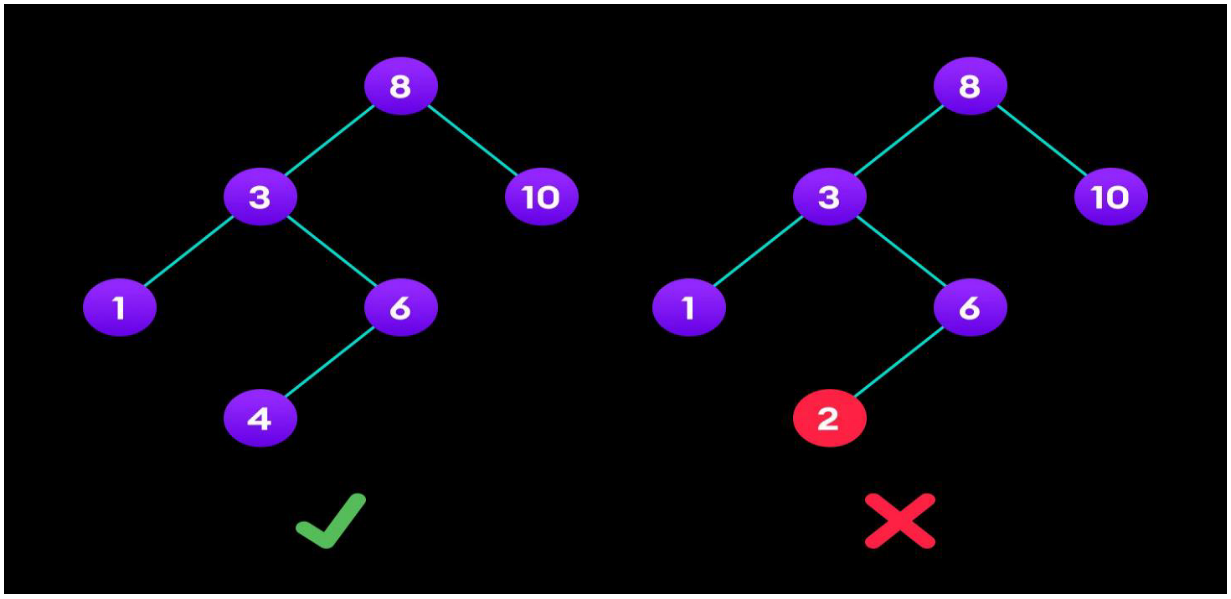
BINARY SEARCH TREE

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

-
- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.
- All nodes of left subtree are less than the root node
- All nodes of right subtree are more than the root node
- Both subtrees of each node are also BSTs i.e. they have the above two properties
- A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree
- The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.



OPERATIONS ON BST:

Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.
If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

Algorithm:

```
If root == NULL
  return NULL;
If number == root->data
  return root->data;
If number < root->data
  return search(root->left)
If number > root->data
  return search(root->right)
```

Insert Operation

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root. We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

Algorithm:

```
If node == NULL
  return createNode(data)
if (data < node->data)
  node->left  = insert(node->left, data);
else if (data > node->data)
  node->right = insert(node->right, data);
return node;
```

Deletion Operation

There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.

Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

3. Get the inorder successor of that node.
4. Replace the node with the inorder successor.
5. Remove the inorder successor from its original position.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
int data;
struct node *right;
struct node *left;
};
struct node * root=NULL,*temp;

struct node* get_new_node(int x){
temp = malloc(sizeof(struct node));
printf("Enter data:");
scanf("%d",&x);
temp->data= x;
temp->left= NULL;
temp->right = NULL;
return temp;
}

struct node *insert_key(struct node *root, int key) {
if (root == NULL) {
return get_new_node(key);
}
if (key < root->data) {
root->left = insert_key(root->left, key);
printf("inserted left of %d\n", root->data);
}
else if (key > root->data) {
root->right = insert_key(root->right, key);
printf("inserted right of %d\n", root->data);
}
return root;
}
struct node *min_Value_of_Node(struct node *node) {
struct node *current = node;
```

```
while (current && current->left != NULL) {
current = current->left;
}

return current;
}
struct node *delete_Node(struct node *root, int key) {
if (root == NULL) {
return root;
}

if (key < root->data) {
root->left = delete_Node(root->left, key);
}
else if (key > root->data) {
root->right = delete_Node(root->right, key);
}
else {
if (root->left == NULL) {
struct node *temp = root->right;
printf("%d is deleted\n", root->data);
free(root);
return temp;
}
else if (root->right == NULL) {
struct node *temp = root->left;
printf("%d is deleted\n", root->data);
free(root);
return temp;
}

struct node *temp = min_Value_of_Node(root->right);
root->data = temp->data;
root->right = delete_Node(root->right, temp->data);
}
return root;
}
void create() {
char ch;
int value;
do {
printf("Enter data: ");
scanf("%d", &value);
```

```c
if (root == NULL) {
root = insert_key(root, value);
printf("Root created\n");
}
else {
insert_key(root, value);
}
printf("Choose (y/Y) to enter more data: ");
scanf(" %c", &ch);
system("clear");

} while (ch == 'y' || ch == 'Y');
}
void inorder_Traversal(struct node *root) {
if (root != NULL) {
inorder_Traversal(root->left);
printf("%d ", root->data);
inorder_Traversal(root->right);
}
}

void preorder_Traversal(struct node *root) {
if (root != NULL) {
printf("%d ", root->data);
preorder_Traversal(root->left);
preorder_Traversal(root->right);
}
}

void postorder_Traversal(struct node *root) {
if (root != NULL) {
postorder_Traversal(root->left);
postorder_Traversal(root->right);
printf("%d ", root->data);
}
}

struct node *search_key(struct node *root, int value) {
if (root == NULL || root->data == value) {
return root;
}
if (value < root->data) {
```

```
return search_key(root->left, value);
}
else {
return search_key(root->right, value);
}
}
void show_BST() {
int ch;
while (1) {
printf("\n1. Preorder\n2. Postorder\n3. Inorder\n");
scanf("%d", &ch);
switch (ch) {
case 1:
printf("Pre-oreder Traversal:\n");
preorder_Traversal(root);
break;
case 2:
printf("In-oreder Traversal:\n");
inorder_Traversal(root);
break;
case 3:
printf("Post-oreder Traversal:\n");
postorder_Traversal(root);
break;
default:
printf("Press spaces to return\n");
return;
}
}

}
void insertNew() {
system("clear");
int value;
printf("Enter value to be inserted: ");
scanf("%d", &value);
insert_key(root, value);
}
void node_Deletion() {
int value;
printf("Enter value to be deleted: ");
scanf("%d", &value);
root = delete_Node(root, value);
```

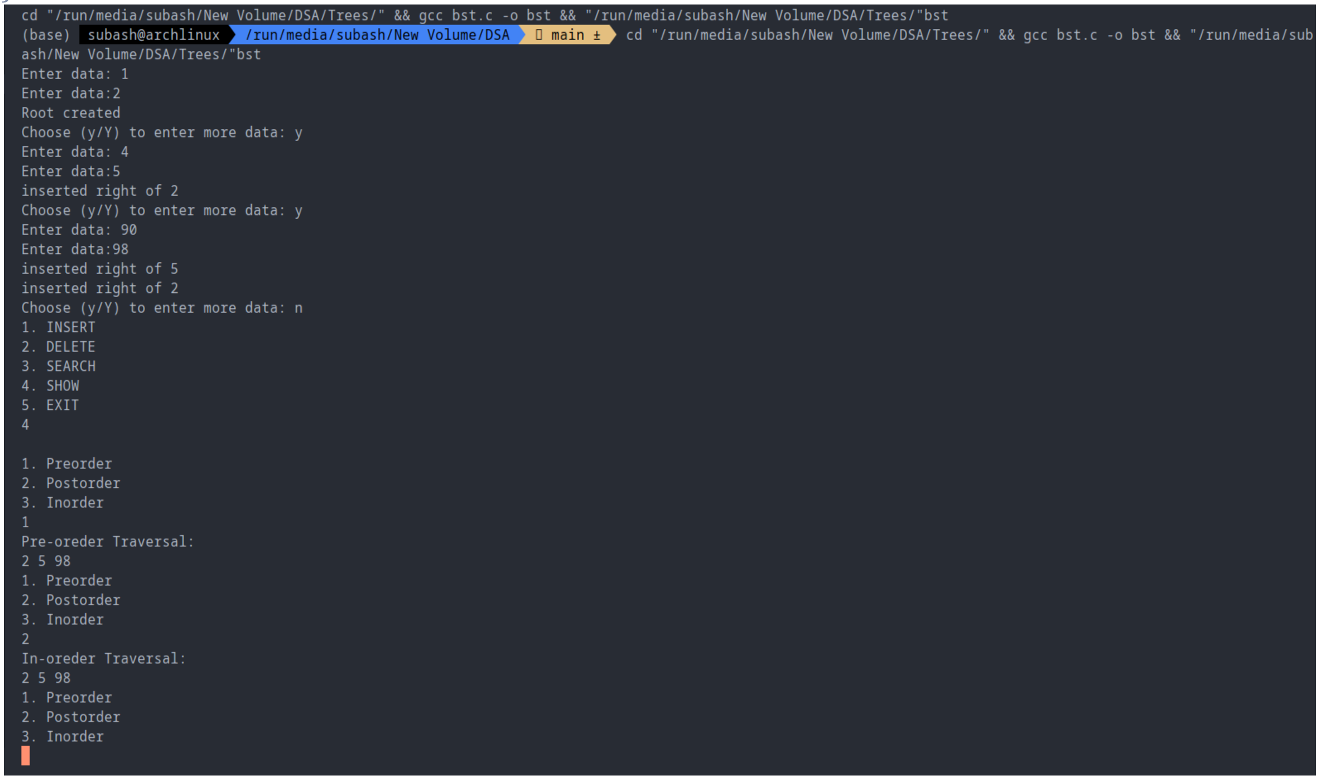
```
}

void search_specific_key() {
int value;
system("clear");
printf("Enter value to be searched: ");
scanf("%d", &value);
struct node *result = search_key(root, value);
if (result != NULL) {
printf("%d found in the tree\n", value);
}
else {
printf("%d not found in the tree\n", value);
}
}

int main() {
int choice;
create();

while (1) {
printf("1. INSERT\n2. DELETE\n3. SEARCH\n4. SHOW\n5. EXIT\n");
scanf("%d", &choice);
switch (choice) {
case 1:
insertNew();
break;
case 2:
node_Deletion();
break;
case 3:
search_specific_key();
break;
case 4:
show_BST();
break;
case 5:
exit(0);
break;
default:
printf("Choose a valid option:\n");
}
}
```

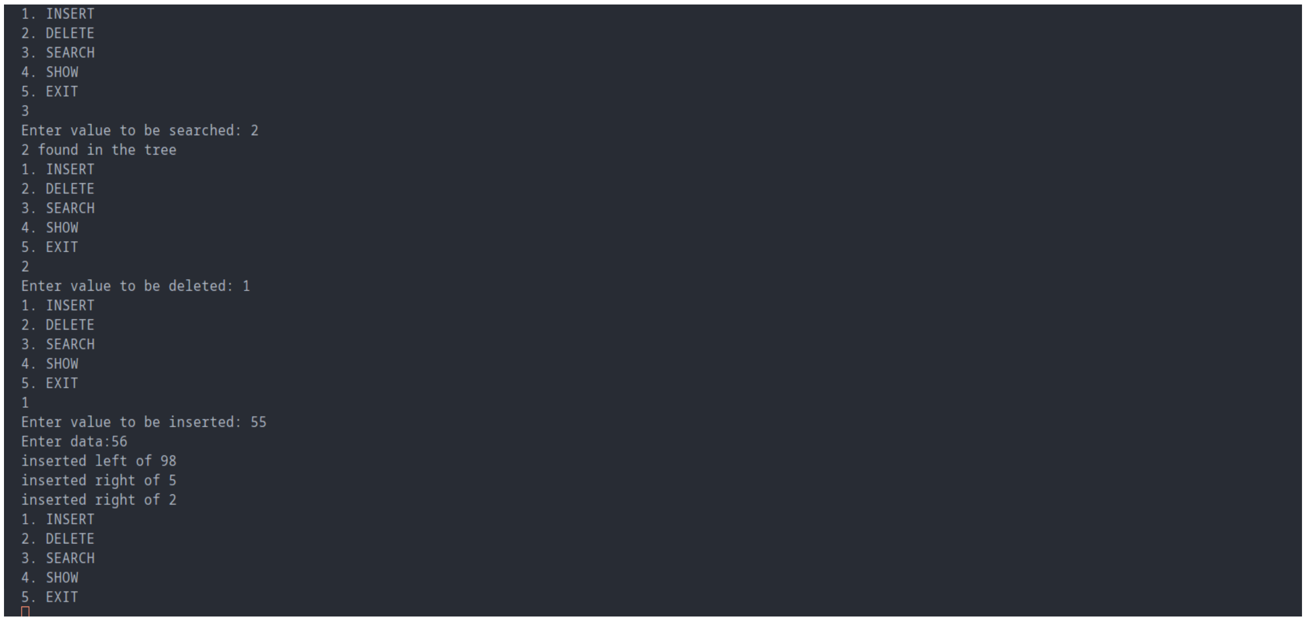
```c
return 0;
}
```



```
cd "/run/media/subash/New Volume/DSA/Trees/" && gcc bst.c -o bst && "/run/media/subash/New Volume/DSA/Trees/"bst
(base)  subash@archlinux  /run/media/subash/New Volume/DSA   main ±  cd "/run/media/subash/New Volume/DSA/Trees/" && gcc bst.c -o bst && "/run/media/sub
ash/New Volume/DSA/Trees/"bst
Enter data: 1
Enter data:2
Root created
Choose (y/Y) to enter more data: y
Enter data: 4
Enter data:5
inserted right of 2
Choose (y/Y) to enter more data: y
Enter data: 90
Enter data:98
inserted right of 5
inserted right of 2
Choose (y/Y) to enter more data: n
1. INSERT
2. DELETE
3. SEARCH
4. SHOW
5. EXIT
4

1. Preorder
2. Postorder
3. Inorder
1
Pre-oreder Traversal:
2 5 98
1. Preorder
2. Postorder
3. Inorder
2
In-oreder Traversal:
2 5 98
1. Preorder
2. Postorder
3. Inorder
```

OUTPUT:



```
1. INSERT
2. DELETE
3. SEARCH
4. SHOW
5. EXIT
3
Enter value to be searched: 2
2 found in the tree
1. INSERT
2. DELETE
3. SEARCH
4. SHOW
5. EXIT
2
Enter value to be deleted: 1
1. INSERT
2. DELETE
3. SEARCH
4. SHOW
5. EXIT
1
Enter value to be inserted: 55
Enter data:56
inserted left of 98
inserted right of 5
inserted right of 2
1. INSERT
2. DELETE
3. SEARCH
4. SHOW
5. EXIT
```