BINARY SEARCH TREE

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree

Algorithms

Let getnode() be a function that allocates memory for node, assign data in node's info, makes lcprt and rcptr null and return the address of node.

Insertion

1. Start
2. ptr = root
3. newptr = getnode()
4. while (ptr != NULL)
   i. parent = ptr
   ii. if(newptr->info < ptr->info)
   ptr = ptr->lcptr
   else
   ptr = ptr->rcptr
5. if(newptr->info < parent->info)
   parent->lcptr = newptr
   else
   parent->rcptr = newptr
6. Stop

Deletion

1. Start
2. while(ptr != NULL AND ptr->info != key)
   i. parent = ptr
   ii. if (key < ptr->info)
   ptr = ptr->lcptr
   else
   ptr = ptr->rcptr
3. if(ptr == NULL)
   print "Node with key doesnt exist"
   else if(ptr->lcptr = NULL AND ptr->rcptr = NULL)
   if(key < parent->info)
   parent->lcptr = NULL
   else

parent->rcptr = NULL
else if(ptr->lcptr = NULL OR ptr->rcptr = NULL)
if(key < parent->info)
if(ptr->lcptr = NULL)
parent->lcptr = ptr->rcptr
else
parent->lcptr = ptr->lcptr
else
if(ptr->lcptr = NULL)
parent->rcptr = ptr->rcptr
else
parent->rcptr = ptr->lcptr
else
del_data = Insucc(ptr)
Deletion(root, del_data)
ptr->info = del_data

4. Stop

IMPLEMENTATION

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

struct Node {
    int info;
    struct Node* lcptr;
    struct Node* rcptr;
};

struct Node* getnode(int key) {
    struct Node* newptr = (struct Node*)malloc(sizeof(struct Node));
    newptr->info = key;
    newptr->lcptr = NULL;
    newptr->rcptr = NULL;
    return newptr;
}

struct Node* insertNode(struct Node* root, int key) {
    struct Node *ptr = root, *parent = NULL, *newptr = getnode(key);

    while (ptr != NULL) {
        parent = ptr;
        if (newptr->info < ptr->info)
            ptr = ptr->lcptr;
        else
            ptr = ptr->rcptr;
    }

    if (parent == NULL)
```

```
        root = newptr;
    else if (newptr->info < parent->info)
        parent->lcptr = newptr;
    else
        parent->rcptr = newptr;

    return root;
}

int Insucc(struct Node* root) {
    while (root->lcptr != NULL)
        root = root->lcptr;
    return root->info;
}

struct Node* deleteNode(struct Node* root, int key) {
    struct Node *ptr = root, *parent = NULL;

    while (ptr != NULL && ptr->info != key) {
        parent = ptr;
        if (key < ptr->info)
            ptr = ptr->lcptr;
        else
            ptr = ptr->rcptr;
    }

    if (ptr == NULL)
        printf("Node with key %d doesn't exist.\n", key);
    else if (ptr->lcptr == NULL && ptr->rcptr == NULL) {
        if (key < parent->info)
            parent->lcptr = NULL;
        else
            parent->rcptr = NULL;
        free(ptr);
    } else if (ptr->lcptr == NULL || ptr->rcptr == NULL) {
        struct Node* child = (ptr->lcptr != NULL) ? ptr->lcptr : ptr-
>rcptr;
        if (key < parent->info)
            parent->lcptr = child;
        else
            parent->rcptr = child;
        free(ptr);
    } else {
        int del_data = Insucc(ptr->rcptr);
        root = deleteNode(root, del_data);
        ptr->info = del_data;
    }

    return root;
}

void inorderTraversal(struct Node* root) {
    if (root != NULL) {
```

```
        inorderTraversal(root->lcptr);
        printf("%d ", root->info);
        inorderTraversal(root->rcptr);
    }
}

int main() {
    struct Node* root = NULL;
    int ch, key;

    while (1) {
        printf("1. Insert\n2. Delete\n3. Inorder Traversal\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insertNode(root, key);
                break;
            case 2:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                root = deleteNode(root, key);
                break;
            case 3:
                printf("Inorder Traversal: ");
                inorderTraversal(root);
                printf("\n");
                break;
            case 4:
                exit(0);
                break;
            default:
                printf("Invalid option.\n");
        }
    }

    return 0;
}
```

OUTPUT

```
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 10
```

```
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 5
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 15
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 3
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 23
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice:
1
Enter key to insert: 11
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 5
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 6
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 3
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
```

```
Enter your choice: 1
Enter key to insert: 8
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter key to insert: 34
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 2
Enter key to delete: 8
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 3
Inorder Traversal: 3 3 5 5 6 10 11 15 23 34
1. Insert
2. Delete
3. Inorder Traversal
4. Exit
Enter your choice: 4
```