Course outcomes-3

Program 1:

Aim:-

Implementation of BST Operations using C(Linked list data structure)

Source Code:-

```c
#include <stdio.h>
#include <stdlib.h>
struct node
{
  int d;
  struct node *llink, *rlink;
};
struct node *create(int d)
{
  struct node *newnode;
  newnode = ((struct node *)malloc(sizeof(struct node)));
  newnode->d = d;
  newnode->llink = NULL;
  newnode->rlink = NULL;
  return (newnode);
}
void inorder(struct node *ptr)
{
  if (ptr != NULL)
  {
    inorder(ptr->llink);
    printf("%d\n", ptr->d);
    inorder(ptr->rlink);
  }
}
void preorder(struct node *ptr)
{
  if (ptr != NULL)
  {
    printf("%d\n", ptr->d);
    preorder(ptr->llink);
    preorder(ptr->rlink);
  }
}

void postorder(struct node *ptr)
```

```
{
  if (ptr != NULL)
  {
    postorder(ptr->llink);
    postorder(ptr->rlink);
    printf("%d\n", ptr->d);
  }
}
int search_el(int key, struct node *ptr)
{
  int k = 0;
  if (ptr != NULL)
  {
    k = search_el(key, ptr->llink);
    if (ptr->d == key)
    {
      return 1;
    }
    k = search_el(key, ptr->rlink);
  }
  return k;
}
struct node *search_elem(int key, struct node *ptr) //findd data of given
position
{
  struct node *k = NULL;
  if (ptr != NULL)
  {
    k = search_elem(key, ptr->llink);
    if (ptr->d == key)
    {
      return ptr;
    }
    k = search_elem(key, ptr->rlink);
  }
  return k;
}
int larget(struct node *ptr)
{ //largest
  while (ptr->rlink != NULL)
  {
    ptr = ptr->rlink;
  }
  return ptr->d;
}
int sml(struct node *ptr)
{ //smallest
  while (ptr->llink != NULL)
```

```
  {
    ptr = ptr->llink;
  }
  return ptr->d;
}
int inorder_sc(struct node *ptr, int key)
{
  struct node *tmp, *mn;
  mn = search_elem(key, ptr);
  if (mn == NULL)
  {
    return 0;
  }
  else
  {
    if (mn->rlink != NULL)
    {
      mn = mn->rlink;
      while (mn->llink != NULL)
      {
        mn = mn->llink;
      }
      return mn->d;
    }
    else
    {
      while (ptr->d != mn->d)
      {
        if (mn->d < ptr->d)
        {
          tmp = ptr;
          ptr = ptr->llink;
        }
        else
          ptr = ptr->rlink;
      }
      return tmp->d;
    }
  }
}
int main()
{
  struct node *root = NULL;
  struct node *temp, *newnode, *ptr;
  int c, elem, i, p, flag = 0;
  char ch;
  while (1)
  {
```

```
    printf("\nBinary search
tree\n1.Creation\n2.Inorder\n3.Preorder\n4.Postorder\n5.Search an
element\n6.Find max\n7.Find min\n8.inorder successor\n9.Exit\nEnter your
choice:");
    scanf("%d", &c);
    switch (c)
    {
    case 1:
      do
      {
        printf("Enter the element to be inserted:");
        scanf("%d", &elem);
        newnode = create(elem);
        if (root == NULL)
          root = newnode;
        else
        {
          ptr = root;
          while ((ptr != NULL) && (flag == 0))
          {
            if (elem < ptr->d)
            {
              temp = ptr;
              ptr = ptr->llink;
            }
            else if (elem > ptr->d)
            {
              temp = ptr;
              ptr = ptr->rlink;
            }
            else
            {
              flag = 1;
              printf("Item already exist\n");
            }
          }
          if ((ptr == NULL) && (flag == 0))
          {
            if (temp->d > elem)
              temp->llink = newnode;
            else
              temp->rlink = newnode;
          }
        }
        flag = 0;
        printf("Do you want to continue(y/n)?");
        getchar();
        scanf("%c", &ch);
```

```
    } while (ch == 'y' || ch == 'Y');
    printf("BST is created\n");
    break;
  case 2:
    if (root != NULL)
    {
      inorder(root);
    }
    else
      printf("Empty\n");
    break;
  case 3:
    if (root != NULL)
    {
      preorder(root);
    }
    else
      printf("Empty\n");
    break;
  case 4:
    if (root != NULL)
    {
      postorder(root);
    }
    else
      printf("Empty\n");
    break;
  case 5:
    printf("Enter the element to search");
    scanf("%d", &elem);
    if (search_el(elem, root) == 0)
      printf("\n\nelemenst no found\n");
    else
      printf("\n\nElement found\n");
    break;
  case 6:
    printf("\n\nlargest data is %d\n", larget(root));
    break;
  case 7:
    printf("\n\nsmallest data is %d\n", sml(root));
    break;
  case 8:
    printf("Enter an elemenst");
    scanf("%d", &elem);
    elem = inorder_sc(root, elem);
    if (elem == 0)
    {
      printf("element not found");
```

```
      }
      else
      {
        printf("successor is %d", elem);
      }
      break;
    case 9:
      exit(0);
    default:
      printf("Invalid choice\n");
    }
  }
  return 0;
}
```

Program 2:

Aim:-

Implementation of Red-Black Tree using C

Source Code:-

```c
#include <stdio.h>
#include <stdlib.h>
enum nodeColor {
  RED,
  BLACK
};
struct rbNode {
  int data, color;
  struct rbNode *link[2];
};
struct rbNode *root = NULL;
// Create a red-black tree
struct rbNode *createNode(int data) {
  struct rbNode *newnode;
  newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
  newnode->data = data;
  newnode->color = RED;
  newnode->link[0] = newnode->link[1] = NULL;
  return newnode;
}
// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
  ptr = root;
  if (!root) {
    root = createNode(data);
    return;
  }

  stack[ht] = root;
  dir[ht++] = 0;
  while (ptr != NULL) {
    if (ptr->data == data) {
      printf("Duplicates Not Allowed!!\n");
      return;
    }
    index = (data - ptr->data) > 0 ? 1 : 0;
```

```
    stack[ht] = ptr;
    ptr = ptr->link[index];
    dir[ht++] = index;
  }
  stack[ht - 1]->link[index] = newnode = createNode(data);
  while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
    if (dir[ht - 2] == 0) {
      yPtr = stack[ht - 2]->link[1];
      if (yPtr != NULL && yPtr->color == RED) {
        stack[ht - 2]->color = RED;
        stack[ht - 1]->color = yPtr->color = BLACK;
        ht = ht - 2;
      } else {
        if (dir[ht - 1] == 0) {
          yPtr = stack[ht - 1];
        } else {
          xPtr = stack[ht - 1];
          yPtr = xPtr->link[1];
          xPtr->link[1] = yPtr->link[0];
          yPtr->link[0] = xPtr;
          stack[ht - 2]->link[0] = yPtr;
        }
        xPtr = stack[ht - 2];
        xPtr->color = RED;
        yPtr->color = BLACK;
        xPtr->link[0] = yPtr->link[1];
        yPtr->link[1] = xPtr;
        if (xPtr == root) {
          root = yPtr;
        } else {
          stack[ht - 3]->link[dir[ht - 3]] = yPtr;
        }
        break;
      }
    } else {
      yPtr = stack[ht - 2]->link[0];
      if ((yPtr != NULL) && (yPtr->color == RED)) {
        stack[ht - 2]->color = RED;
        stack[ht - 1]->color = yPtr->color = BLACK;
        ht = ht - 2;
      } else {
        if (dir[ht - 1] == 1) {
          yPtr = stack[ht - 1];
        } else {
          xPtr = stack[ht - 1];
          yPtr = xPtr->link[0];
          xPtr->link[0] = yPtr->link[1];
          yPtr->link[1] = xPtr;
```

```
          stack[ht - 2]->link[1] = yPtr;
        }
        xPtr = stack[ht - 2];
        yPtr->color = BLACK;
        xPtr->color = RED;
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        if (xPtr == root) {
          root = yPtr;
        } else {
          stack[ht - 3]->link[dir[ht - 3]] = yPtr;
        }
        break;
      }
    }
  }
  root->color = BLACK;
}
// Delete a node
void deletion(int data) {
  struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
  struct rbNode *pPtr, *qPtr, *rPtr;
  int dir[98], ht = 0, diff, i;
  enum nodeColor color;

  if (!root) {
    printf("Tree not available\n");
    return;
  }

  ptr = root;
  while (ptr != NULL) {
    if ((data - ptr->data) == 0)
      break;
    diff = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    dir[ht++] = diff;
    ptr = ptr->link[diff];
  }

  if (ptr->link[1] == NULL) {
    if ((ptr == root) && (ptr->link[0] == NULL)) {
      free(ptr);
      root = NULL;
    } else if (ptr == root) {
      root = ptr->link[0];
      free(ptr);
    } else {
```

```
      stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
    }
  } else {
    xPtr = ptr->link[1];
    if (xPtr->link[0] == NULL) {
      xPtr->link[0] = ptr->link[0];
      color = xPtr->color;
      xPtr->color = ptr->color;
      ptr->color = color;

      if (ptr == root) {
        root = xPtr;
      } else {
        stack[ht - 1]->link[dir[ht - 1]] = xPtr;
      }

      dir[ht] = 1;
      stack[ht++] = xPtr;
    } else {
      i = ht++;
      while (1) {
        dir[ht] = 0;
        stack[ht++] = xPtr;
        yPtr = xPtr->link[0];
        if (!yPtr->link[0])
          break;
        xPtr = yPtr;
      }

      dir[i] = 1;
      stack[i] = yPtr;
      if (i > 0)
        stack[i - 1]->link[dir[i - 1]] = yPtr;

      yPtr->link[0] = ptr->link[0];

      xPtr->link[0] = yPtr->link[1];
      yPtr->link[1] = ptr->link[1];

      if (ptr == root) {
        root = yPtr;
      }

      color = yPtr->color;
      yPtr->color = ptr->color;
      ptr->color = color;
    }
  }
```

```
if (ht < 1)
  return;

if (ptr->color == BLACK) {
  while (1) {
    pPtr = stack[ht - 1]->link[dir[ht - 1]];
    if (pPtr && pPtr->color == RED) {
      pPtr->color = BLACK;
      break;
    }

    if (ht < 2)
      break;

    if (dir[ht - 2] == 0) {
      rPtr = stack[ht - 1]->link[1];

      if (!rPtr)
        break;

      if (rPtr->color == RED) {
        stack[ht - 1]->color = RED;
        rPtr->color = BLACK;
        stack[ht - 1]->link[1] = rPtr->link[0];
        rPtr->link[0] = stack[ht - 1];

        if (stack[ht - 1] == root) {
          root = rPtr;
        } else {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        dir[ht] = 0;
        stack[ht] = stack[ht - 1];
        stack[ht - 1] = rPtr;
        ht++;

        rPtr = stack[ht - 1]->link[1];
      }

      if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
        (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
        rPtr->color = RED;
      } else {
        if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
          qPtr = rPtr->link[0];
          rPtr->color = RED;
          qPtr->color = BLACK;
```

```
          rPtr->link[0] = qPtr->link[1];
          qPtr->link[1] = rPtr;
          rPtr = stack[ht - 1]->link[1] = qPtr;
        }
        rPtr->color = stack[ht - 1]->color;
        stack[ht - 1]->color = BLACK;
        rPtr->link[1]->color = BLACK;
        stack[ht - 1]->link[1] = rPtr->link[0];
        rPtr->link[0] = stack[ht - 1];
        if (stack[ht - 1] == root) {
          root = rPtr;
        } else {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        break;
      }
    } else {
      rPtr = stack[ht - 1]->link[0];
      if (!rPtr)
        break;

      if (rPtr->color == RED) {
        stack[ht - 1]->color = RED;
        rPtr->color = BLACK;
        stack[ht - 1]->link[0] = rPtr->link[1];
        rPtr->link[1] = stack[ht - 1];

        if (stack[ht - 1] == root) {
          root = rPtr;
        } else {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        dir[ht] = 1;
        stack[ht] = stack[ht - 1];
        stack[ht - 1] = rPtr;
        ht++;

        rPtr = stack[ht - 1]->link[0];
      }
      if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
        (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
        rPtr->color = RED;
      } else {
        if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
          qPtr = rPtr->link[1];
          rPtr->color = RED;
          qPtr->color = BLACK;
          rPtr->link[1] = qPtr->link[0];
```

```
            qPtr->link[0] = rPtr;
            rPtr = stack[ht - 1]->link[0] = qPtr;
          }
          rPtr->color = stack[ht - 1]->color;
          stack[ht - 1]->color = BLACK;
          rPtr->link[0]->color = BLACK;
          stack[ht - 1]->link[0] = rPtr->link[1];
          rPtr->link[1] = stack[ht - 1];
          if (stack[ht - 1] == root) {
            root = rPtr;
          } else {
            stack[ht - 2]->link[dir[ht - 2]] = rPtr;
          }
          break;
        }
      }
      ht--;
    }
  }
}
// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}

int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
    scanf("%d", &ch);
    switch (ch) {
      case 1:
        printf("Enter the element to insert:");
        scanf("%d", &data);
        insertion(data);
        break;
      case 2:
        printf("Enter the element to delete:");
        scanf("%d", &data);
        deletion(data);
        break;
```

```
      case 3:
        inorderTraversal(root);
        printf("\n");
        break;
      case 4:
        exit(0);
      default:
        printf("Not available\n");
        break;
    }
    printf("\n");
  }
  return 0;
}
```

Program 3:

Aim:-

Implementation of Btree using C

Source Code:-

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 3
#define MIN 2
struct BTreeNode
{
int val[MAX + 1], count;
struct BTreeNode *link[MAX + 1];
};
struct BTreeNode *root;
struct BTreeNode *createNode(int val, struct BTreeNode *child) {
struct BTreeNode *newNode;
newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
newNode->val[1] = val;
newNode->count = 1;
newNode->link[0] = root;
newNode->link[1] = child;
return newNode;
}
void insertNode(int val, int pos, struct BTreeNode *node,
struct BTreeNode *child) {
int j = node->count;
while (j > pos) {
node->val[j + 1] = node->val[j];
node->link[j + 1] = node->link[j];
j--;
}
node->val[j + 1] = val;
node->link[j + 1] = child;
node->count++;
}
void splitNode(int val, int *pval, int pos, struct BTreeNode *node,
struct BTreeNode *child, struct BTreeNode **newNode) {
int median, j;
if (pos > MIN)
median = MIN + 1;
else
median = MIN;
*newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
j = median + 1;
```

```
while (j <= MAX) {
(*newNode)->val[j - median] = node->val[j];
(*newNode)->link[j - median] = node->link[j];
j++;
}
node->count = median;
(*newNode)->count = MAX - median;
if (pos <= MIN) {
insertNode(val, pos, node, child);
} else {
insertNode(val, pos - median, *newNode, child);
}
*pval = node->val[node->count];
(*newNode)->link[0] = node->link[node->count];
node->count--;
}
int setValue(int val, int *pval,
struct BTreeNode *node, struct BTreeNode **child) {
int pos;
if (!node) {
*pval = val;
*child = NULL;
return 1;
}
if (val < node->val[1]) {
pos = 0;
} else {
for (pos = node->count;
(val < node->val[pos] && pos > 1); pos--)
;
if (val == node->val[pos]) {
printf("Duplicates are not permitted\n");
return 0;
}
}
if (setValue(val, pval, node->link[pos], child)) {
if (node->count < MAX) {
insertNode(*pval, pos, node, *child);
} else {
splitNode(*pval, pval, pos, node, *child, child);
return 1;
}
}
return 0;
}
void insert(int val) {
int flag, i;
struct BTreeNode *child;
```

```
flag = setValue(val, &i, root, &child);
if (flag)
root = createNode(i, child);
}
void search(int val, int *pos, struct BTreeNode *myNode) {
if (!myNode) {
return;
}
if (val < myNode->val[1]) {
*pos = 0;
} else {
for (*pos = myNode->count;
(val < myNode->val[*pos] && *pos > 1); (*pos)--)
;
if (val == myNode->val[*pos]) {
printf("%d is found", val);
return;
}
}
search(val, pos, myNode->link[*pos]);
return;
}
void traversal(struct BTreeNode *myNode) {
int i;
if (myNode) {
for (i = 0; i < myNode->count; i++) {
traversal(myNode->link[i]);
printf("%d ", myNode->val[i + 1]);
}
traversal(myNode->link[i]);
}
}
int main() {
int val, ch;
insert(8);
insert(9);
insert(10);
insert(11);
insert(15);
insert(16);
insert(17);
insert(18);
insert(20);
insert(23);
traversal(root);
printf("\n");
search(11, &ch, root);
}
```