Maintenance Of In Threaded Binary Tree

Code Id	23
Date Updated	3/7/2010
Title	Maintenance of inthreaded binary tree.

Description

This program illustrates Insertion, Deletion and Traversal in fully in-threaded B

Codes Snippet

```
# include
# include
#define infinity 9999
typedef enum { thread,link} boolean;
struct node *in_succ(struct node*p);
struct node *in ""pred( struct node *p);
struct node
 {
struct node *left_ptr;
boolean left;
int info;
boolean right;
struct node *right_ptr;
} *head=NULL;
main( )
{
int choice,num;
insert_head( );
while(1)
{       printf("n");
printf(" 1.Insertn");
printf("2.Deleten");
printf("3.Inorder Traversaln");
printf("4.Preorder Traversaln");
printf("5.Quitn");
printf("Enter your choice: ");
scanf("%d" ,&choice);
switch( choice)
{
case 1:
printf("Enter the number to be inserted: "); .
scanf("%d" ,&num);
insert(num);
break;
case 2:
printf("Enter the number to be deleted: ");
scanf("%d" ,&num);
del(num);
break;
        case 3:
        inorder( );
        break;
        case 4:
        preorder( );
        break;

case 5:
        exit( );
        default:
        printf("Wrong choicen");
        }/*End of switch */
        }/*End of while */
}/*End ofmain( )*/
insert_head( )
{

struct node *tmp;
             head=(struct node *)malIoc(sizeof(struct node));
head->info= infinity;
head->left=thread;
```

```
                if(item==head->left_ptr->info) /* item is at head->left_ptr */
{
*loc=head->left_ptr;
*par=head;
return;
        }
ptr=head->left_ptr;
while(ptr!=head)
{       ptrsave=ptr;
if( item < ptr->info )
{               if(ptr->left==link)
ptr=ptr->left_ptr;
                }

else
break;
else
if(item> ptr->info )
{
if(ptr->right==link)
                                ptr=ptr->right_ptr;
else
break;
                        }
if (item== ptr->info)
{
*loc=ptr;
*par=ptrsave;
return;
}
        }/*End of while*/
*loc=NULL; /*item not found*/
*par=ptrsave;
}/*End of find( )*/
insert(int item)
{
struct node *tmp, *parent, *location;
find( item,&parent,&location);
if(location!=NULL)
{       printf("Item already present");
return;
}
tmp=(struct node *)malloc(sizeof(struct node));
tmp->info=item;
tmp->left=thread;
tmp->right=thread;
if(parent== head) /*tree is empty* /
{       head->left=link;
head->left_ptr=tmp;
tmp->left_ptr=head;
tmp->right_ptr=head;
}

else
if( item < parent->info )
{
tmp->left_ptr=parent->left_ptr;
tmp->ri ght_ptr=parent;
parent->left=link;
parent->left-ptr=tmp;
}
else
{
tmp->left_ptr=parent;
tmp->right_ptr=parent ->right_ptr;
parent->right=link;
parent ->right_ptr=tmp;
}
}/*End of insert( )*/
del(int item)
{
struct node *parent, *location;
if(head==NULL)
{
printf("Tree empty");
return;
}
find( item,&parent,&location);
if(location==NULL)
```

```
}/*End of del( )*/
case_a(struct node *par,struct node *loc )
{
if(par== head) /*item to be deleted is first node*/
{
head->left=thread;
head->left_ptr=head;
}
else
if(loc==par -> left_ptr)
{
par->left=thread;
par->left_ptr=loc->left_ptr;
}
else
{
par ->right=thread;
par ->ri ght_ptr=loc ->ri ght_ptr;
        }
        free(loc );
}/*End of case_a( )*/
case_b(struct node *par,struct node *loc)
{
struct node *child, *s, *p;
/*Initialize child* /
if(loc->left==link) /*item to be deleted has left-ptr */
child=loc->left_ptr;
        /*item to be deleted has right-ptr */
child= Ioc->right_ptr;
else
                if(par==head) /*Item to be deleted is first node*/
                        head->left_ptr=child;
else
if( loc== par->left_ptr)/*item is left_ptr of its parent*/
                        par-> left_ptr=child;
        else    /*item is right_ptr of its parent*'
                        par ->ri gh t_ptr=chi!d;
s=in_succ(loc);
p=in_pred(loc);
if(loc->righl==link) /*ifloc has right subtree*/
                s->left_ptr=p;
            else        /*if loc has left subtree */
{
if( loc->left==link)
                        p->right_ptr=s;
}
 free(loc); .
}/*Endof case_b( )*/
case_c(struct node *par,struct node *loc)
{
struct node *ptr, *ptrsave, *suc, *parsuc, *s, *p;
/*Find in order successor and its parent*/
 ptrsave=loc;
ptr= Ioc->right_ptr;
while(ptr -> left==link)
{
ptrsave=ptr;
ptr=ptr->left_ptr;
}
suc=ptr;
parsuc=ptrsave;
loc->info=suc->info;
        if(suc->left==thread && suc->right==thread)
                case_a(parsuc,suc );
else
case _b(parsuc,suc);
}/*End of case_c( )*/
struct node *in_succ(struct node *ptr)
{
struct node *succ;
if(ptr ->right==thread)
succ=ptr ->right_ptr;
else
 {
ptr=ptr->right_ptr;
while(ptr-> left==l ink)
ptr=ptr->left_ptr;
succ=ptr;
}
```

```
pred=ptr;
}
return pred;
}/*End of in-pred( )*/
inorder( ) .
{
struct node *ptr;
if(head->left_ptr== head)
{
printf("Tree is empty");
return;
}
ptr=head->left_ptr;
/*Find the leftmost node and traverse it */
while(ptr -> left==l ink)
ptr=ptr->left_ptr;
printf("%d ",ptr->info);
while( 1 )
{
ptr=in_succ(ptr);
if(ptr==head) /*If last node reached */
        break;
        printf("%d  ,ptr->info);
        } /*End of while*/
}/*End of inorder( )*/
preoder()
{
struct node *ptr;
if(head->left_ptr==head)
{
                printf(Tree is empty);
                return;
}
ptr=head->left_ptr;
while( ptr!= head)
{
printf("%d ",ptr->info);
if( ptr->left==link)
ptr=ptr -> left_ptr;
else
if(ptr->right_ptr==link)
                                ptr=ptr->right_ptr;
else
{
while( ptr!=head && ptr->right==thread)
                                        ptr=ptr->right_ptr;
if(ptr!=head )
                                        ptr=ptr->righ_ptr;
        }
        }/*End of while*/
} /*End of preorder( )* /
```