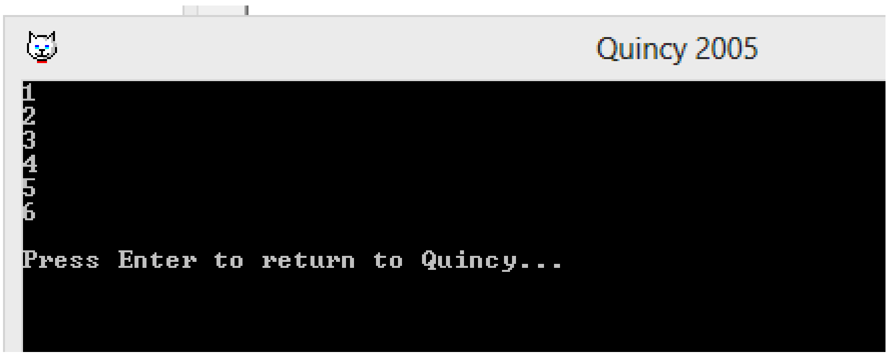
Practical-1: Implementation of randomized quicksort algorithm.

```
#include<iostream.h>
#include<ctime>
#include<stdlib.h>
void swap(int *m,int *n)
{
	int temp=*m;
	*m=*n;
	*n=temp;
}
int partition(int a[],int low,int high)
{
	int x=a[high];
	int i=low-1;
	for(int j=low;j<high;j++)
	{
		if(a[j]<=x)
		{
			i++;
			swap(&a[j],&a[i]);
		}
	}
	swap(&a[i+1],&a[high]);
	return i+1;
}
int randomization(int a[],int low,int high)
{
	int r=low+(rand()%(high-low+1));
	int x=a[r];
	int i=low-1;
	for(int j=low;j<high;j++)
	{
		if(a[j]<=x)
		{
			i++;
			swap(&a[j],&a[i]);
		}
	}
	swap(&a[i+1],&a[high]);
	return i+1;
}
void quickshort(int a[],int low,int high)
{
	if(low<high)
	{
		//int pivotIndex=partition(a,low,high);
```

```
                int pivotIndex=randomization(a,low,high);
                quickshort(a,low,pivotIndex-1);
                quickshort(a,pivotIndex+1,high);
        }
}
int main()
{
        int n=6;
        int arr[6]={1,4,2,3,6,5};
        //int n=10000;
        //int arr[10000];
        //for(int i=0;i<n;i++)
        //        arr[i]=rand()*rand();
        clock_t start_time,end_time;
        start_time = clock();
        quickshort(arr,0,n-1);
        end_time = clock();
        for(int j=0;j<n;j++)
                cout<<arr[j]<<endl;
        //cout<<"Time Taken: "<<(end_time - start_time)<<endl;;
        return 0;
}
```

Output:



```
1
2
3
4
5
6

Press Enter to return to Quincy...
```

Practical-2: Implementation of operations on splay trees.

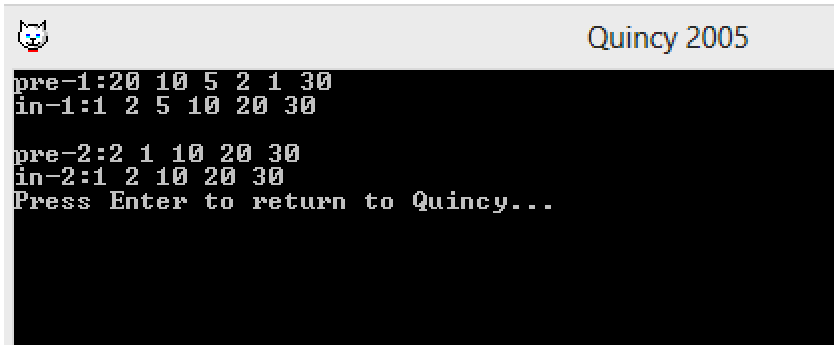
```
#include<iostream.h>
struct node
{
		int key;
		struct node *left,*right;
};
struct node *head=0;
struct node *create(int value)
{
		struct node *a=new(struct node);
		a->key=value;
		a->left=0;
		a->right=0;
		return a;
}
node *zig(struct node *x)
{
   //cout<<endl<<"zig-"<<x->key;
		node *y = x->left;
   x->left = y->right;
   y->right = x;
   return y;
}
node *zag(struct node *x)
{
   //cout<<endl<<"zag-"<<x->key;
		node *y = x->right;
   x->right = y->left;
   y->left = x;
   return y;
}
struct node *splay(struct node *root, int key)
{
   if (root == NULL || root->key == key)
      return root;
   if (root->key > key)
   {
      if (root->left == NULL) return root;
      if (root->left->key > key)
      {

         root->left->left = splay(root->left->left, key);
         root = zig(root);
      }
      else if (root->left->key < key)
```

```
        {
           root->left->right = splay(root->left->right, key);
           if (root->left->right != NULL)
              root->left = zag(root->left);
        }
        return (root->left == NULL)? root: zig(root);
    }
    else
    {
        if (root->right == NULL) return root;

        if (root->right->key > key)
        {
           root->right->left = splay(root->right->left, key);

           if (root->right->left != NULL)
              root->right = zig(root->right);
        }
        else if (root->right->key < key)
        {
           root->right->right = splay(root->right->right, key);
           root = zag(root);
        }
        return (root->right == NULL)? root: zag(root);
    }
}
struct node *search(struct node *root,int key)
{
            return splay(root,key);
}
void preOrder(node *root)
{
            if (root != NULL)
            {
                        cout<<root->key<<" ";
                        preOrder(root->left);
                        preOrder(root->right);
            }
}
void inOrder(node *root)
{
            if (root != NULL)
            {
                        inOrder(root->left);
                        cout<<root->key<<" ";
                        inOrder(root->right);
            }
```

```
}
struct node *insert(node *root, int k)
{
   if (root == NULL) return create(k);
   root = splay(root, k);
          if (root->key == k) return root;
   node *newnode = create(k);
          if (root->key > k)
   {
      newnode->right = root;
      newnode->left = root->left;
      root->left = NULL;
   }
   else
   {
      newnode->left = root;
      newnode->right = root->right;
      root->right = NULL;
   }
   return newnode;
}
struct node* delete_key(struct node *root, int key)
{
   struct node *temp;
   if (!root)
      return NULL;
   root = splay(root, key);
   if (key != root->key)
      return root;
   if (!root->left)
   {
      temp = root;
      root = root->right;
   }
   else
   {
      temp = root;
      root = splay(root->left, key);
      root->right = temp->right;
   }
   free(temp);
   return root;
}
int main()
{
          struct node *root=create(20);
          root->left=create(10);
```

```
        root->right=create(30);
        root->left->left=create(5);
        root->left->left->left = create(2);
        root->left->left->left->left = create(1);
        cout<<"pre-1:";preOrder(root);cout<<endl;
        cout<<"in-1:";inOrder(root);cout<<endl;cout<<endl;
        root=search(root,5);
        root = delete_key(root, 5);
        cout<<"pre-2:";preOrder(root);cout<<endl;
        cout<<"in-2:";inOrder(root);
}
```

Output:



```
pre-1:20 10 5 2 1 30
in-1:1 2 5 10 20 30

pre-2:2 1 10 20 30
in-2:1 2 10 20 30
Press Enter to return to Quincy...
```