Main

```
#include <iostream>
#include "DigitalSearchTree.h"

using namespace std;

int main()
{
    DigitalSearchTree<int> ds;
    DigitalSearchTree<char> s;

    ds.insert(25);
    ds.insert(2);
    ds.insert(5);
    ds.insert(3);

    cout << "\nInorder: ";
    ds.print_inorder();
    cout <<"\nPreorder: ";
    ds.print_preorder();
    cout <<"\nPostorder: ";
    ds.print_postorder();

    int x = ds.countNode();
    cout<< "\nNo. of nodes: "<< x;



    s.insert('a');
    s.insert('z');
    s.insert('S');
    s.insert('d');
    s.insert('A');
    s.insert('G');
    s.insert('E');
    s.insert('g');
    s.insert('e');

    cout << "\nInorder: ";
    s.print_inorder();
    cout <<"\nPreorder: ";
    s.print_preorder();
```

```
cout <<"\nPostorder: ";
s.print_postorder();

x = s.countNode();
cout<< "\nNo. of nodes: "<< x;

cout<<endl;
cout<<endl;
cout<<endl;
cout<<endl;



if(s.search('A'))
{
   cout<<"found";
}
else
   cout<<"not found"<<endl;

if(s.search('a'))
{
   cout<<"found";
}
else
   cout<<"not found"<<endl;


if(s.search('E'))
{
   cout<<"found";
}
else
   cout<<"not found"<<endl;

if(s.search('G'))
{
   cout<<"found";
}
else
   cout<<"not found"<<endl;

if(s.search('e'))
{
        cout<<"found";
    }
    else
        cout<<"not found"<<endl;


    if(s.search('F'))
    {
        cout<<"found"<<endl;
    }
    else
        cout<<"not found"<<endl;


    cout<<endl;
    cout<<"removing g "<<endl;
    s.remove('g');
    s.print_inorder();
    cout<<endl;
    cout<<endl;

    cout<<"removing e, A, z "<<endl;
    s.remove('e');
    s.remove('A');
    s.remove('z');
    cout << "\nInorder: ";
    s.print_inorder();

}
```

Header Files

DigitalSearchTree.h

```
#ifndef DIGITALSEARCHTREE_H_INCLUDED
#define DIGITALSEARCHTREE_H_INCLUDED
#include <bitset>

#include "quetype.h"

template <class DataType>
class DigitalSearchTree
{
private:
    struct tree_node
    {
        tree_node* left;
        tree_node* right;
        DataType data;
    };
    tree_node* root;
    void fillInOrder(QueType<int>&,tree_node*);
    void fillInPreOrder(QueType<int>&,tree_node*);
    void fillInPostOrder(QueType<int>&,tree_node*);
    void makeEmpty(tree_node*&);
    void inorder(tree_node*);
    void preorder(tree_node*);
    void postorder(tree_node*);

public:
    DigitalSearchTree();
    virtual ~DigitalSearchTree();
    bool isEmpty();
    void insert(DataType);
    void remove(DataType);
    void print_inorder();
    void print_preorder();
    void print_postorder();
    void getInOrder(QueType<int>&);
    void counter(int*, tree_node*);
    bool search(DataType);
    int countNode();
};

#endif // DIGITALSEARCHTREE_H_INCLUDED
```

queuetype.h

```
#ifndef QUETYPE_H_INCLUDED
#define QUETYPE_H_INCLUDED
#include <iostream>
using namespace std;
class FullQueue {};
class EmptyQueue {};
template <class DataType>
class QueType
{
    struct NodeType
    {
        DataType info;
```

```
      NodeType* next;
   };
public:
   QueType();
   ~QueType();
   void MakeEmpty();
   void Enqueue(DataType);
   DataType Dequeue();
   bool IsEmpty();
   bool IsFull();
private:
   NodeType *front, *rear;
};
#endif // QUETYPE_H_INCLUDED
```

CPP Files

DigitalSearchTree.cpp

```
#include "DigitalSearchTree.h"

template <class DataType>
DigitalSearchTree<DataType>::DigitalSearchTree()
{
   root = NULL;
}

template <class DataType>
DigitalSearchTree<DataType>::~DigitalSearchTree()
{
   //dtor
}

template <class DataType>
void DigitalSearchTree<DataType>::insert(DataType d)
{
   std::string s = std::bitset< 4 >( (int)d ).to_string();
   int i = 0;
   cout << "\n"<<d<<" is: "<<s<<"\n";
   char c = s[0];

   tree_node* t = new tree_node;
   t->data = d;
   t->left = NULL;
   t->right = NULL;
   tree_node* parent;
   parent = NULL;

   if(isEmpty())
   {
      root = t;
   }
   else // inserting into a non-empty tree
   {
      tree_node* curr;
      curr = root;

      while(curr)
      {
         parent = curr;
         if(c == '0')
         {
            curr = curr->left;
         }
         else
         {
            curr = curr->right;
         }
         i++;
         c = s[i];
      }
      i--;
      c = s[i];
      if(c == '0')
      {
         parent->left = t;
      }
      else
      {
         parent->right = t;
      }
   }
}

template <class DataType>
bool DigitalSearchTree<DataType>::isEmpty()
{
   if(root == NULL)
   {
      return true;
   }
   else
      return false;
}

template <class DataType>
void DigitalSearchTree<DataType>::inorder(tree_node* p)
{
   if(p != NULL)
   {
      if(p->left)
         inorder(p->left);
      cout<<" "<<p->data<<" ";
      if(p->right)
         inorder(p->right);
   }
   else
      return;
}
template <class DataType>
void DigitalSearchTree<DataType>::getInOrder(QueType<int>& q)
{
   if(!q.IsEmpty())
      q.MakeEmpty();
   fillInOrder(q,root);
}
template <class DataType>
void DigitalSearchTree<DataType>::fillInOrder(QueType<int>& q,tree_node* p)
{
   if(p!= NULL)
   {
      if(p->left)
         fillInOrder(q,p->left);
      q.Enqueue(p->data);
      if(p->right)
         fillInOrder(q,p->right);
   }
   else
      return;
}
template <class DataType>
```

```
void DigitalSearchTree<DataType>::print_inorder()
{
   inorder(root);
}


template<class DataType>
void DigitalSearchTree<DataType>::print_preorder()
{
   preorder(root);
}
template<class DataType>
void DigitalSearchTree<DataType>::preorder(tree_node* p)
{
   if(p != NULL)
   {
      cout<<" "<<(DataType)p->data<<" ";
      if(p->left)
         preorder(p->left);
      if(p->right)
         preorder(p->right);
   }
   else
      return;
}
template<class DataType>
void DigitalSearchTree<DataType>::print_postorder()
{
   postorder(root);
}
template<class DataType>
void DigitalSearchTree<DataType>::postorder(tree_node* p)
{
   if(p != NULL)
   {
      if(p->left)
         postorder(p->left);
      if(p->right)
         postorder(p->right);

      cout<<" "<<(DataType)p->data<<" ";
   }
   else
      return;
```

```
}
template <class DataType>
void DigitalSearchTree<DataType>::counter(int* c, tree_node* p)
{
   if(p != NULL)
   {
      if(p->left)
         counter(c, p->left);
      if(p->right)
         counter(c, p->right);
      *c = *c+1;
   }
   else
      return;
}
template <class DataType>
int DigitalSearchTree<DataType>::countNode()
{
   int c=0;
   counter(&c, root);
   return c;
}

template <class DataType>
bool DigitalSearchTree<DataType>::search(DataType d)
{
   std::string s = std::bitset< 4 >( (int)d ).to_string();
   cout << "\n"<<d<<" is: "<<s<<"\n";

   int i = 0;
   char c = s[0];

   bool found = false;

   tree_node* parent;
   parent = NULL;

   if(isEmpty())
   {
      cout<<"tree is empty"<<endl;
   }
   else
   {
      tree_node* curr;
      curr = root;

      while(curr)
      {
         if(curr->data == d)
         {
            found = true;
            break;
         }
         if(curr->left == NULL && curr->right == NULL)
         {
            return found;
         }

         if(c == '0' && curr->left != NULL )
         {
            curr = curr->left;
            if(curr->data == d)
            {
               found = true;
               break;
            }
         }
         else if(curr->right != NULL )
         {
            curr = curr->right;
            if(curr->data == d)
            {
               found = true;
               break;
            }

         }
         i++;
         c = s[i];
      }

   }
   return found;
}

template <class DataType>
void DigitalSearchTree<DataType>::remove(DataType d)
```

```
{
   std::string s = std::bitset< 4 >( (int)d ).to_string();


   int i = 0;
   char c = s[0];

   bool found = false;

   tree_node* parent;
   parent = NULL;

   if(isEmpty())
   {
      cout<<"tree is empty"<<endl;
   }
   else
   {

      tree_node* curr;
      curr = root;

      while(curr)
      {

         if(curr->data == d)
         {
            found = true;
            break;
         }
         if(curr->left == NULL && curr->right == NULL)
         {
            cout<<"not found"<<endl;
            break;
         }

         if(c == '0' && curr->left != NULL )
         {
            parent = curr;
            curr = curr->left;
            if(curr->data == d)
            {
               found = true;
         break;
      }
   }
   else if(curr->right != NULL )
   {
      parent = curr;
      curr = curr->right;
      if(curr->data == d)
      {
         found = true;

         break;
      }

   }
   i++;
   c = s[i];
}

if(found)
{

   if((curr->left == NULL && curr->right != NULL) || (curr->left != NULL
         && curr->right == NULL))
   {
      if(curr->left == NULL && curr->right != NULL)
      {
         if(parent->left == curr)
         {
            parent->left = curr->right;
            delete curr;
         }
         else
         {
            parent->right = curr->right;
            delete curr;
         }
      }
      else
      {
         if(parent->left == curr)
         {
            parent->left = curr->left;
            delete curr;
      }
      else
      {
         parent->right = curr->left;
         delete curr;
      }
   }
   return;
}

if( curr->left == NULL && curr->right == NULL)
{
   if(parent->left == curr)
      parent->left = NULL;
   else
      parent->right = NULL;
   delete curr;
   return;
}

if (curr->left != NULL && curr->right != NULL)
{
   tree_node* chkr;
   chkr = curr->right;
   if((chkr->left == NULL) && (chkr->right == NULL))
   {
      curr->data = chkr->data;
      delete chkr;
      curr->right = NULL;
   }
   else
   {

      if((curr->right)->left != NULL)
      {
         tree_node* lcurr;
         tree_node* lcurrp;
         lcurrp = curr->right;
         lcurr = (curr->right)->left;
         while(lcurr->left != NULL)
         {
            lcurrp = lcurr;
            lcurr = lcurr->left;
         }
```

```
                  curr->data = lcurr->data;
                  delete lcurr;
                  lcurrp->left = NULL;
               }
               else
               {
                  tree_node* tmp;
                  tmp = curr->right;
                  curr->data = tmp->data;
                  curr->right = tmp->right;
                  delete tmp;
               }
            }
            return;
         }
      }

   }
}

template class DigitalSearchTree<int>;
template class DigitalSearchTree<char>;
```

queuetype.cpp

```
#include "quetype.h"
template <class DataType>
QueType<DataType>::QueType()
{
   front = NULL;
   rear = NULL;
}
template <class DataType>
bool QueType<DataType>::IsEmpty()
{
   return (front == NULL);
}
template<class DataType>
bool QueType<DataType>::IsFull()
{
   NodeType* location;
   try
   {
      location = new NodeType;
      delete location;
      return false;
   }
   catch(bad_alloc& exception)
   {
      return true;
   }
}
template <class DataType>
void QueType<DataType>::Enqueue(DataType newItem)
{
   if (IsFull())
      throw FullQueue();
   else
   {
      NodeType* newNode;
      newNode = new NodeType;
      newNode->info = newItem;
      newNode->next = NULL;
      if (rear == NULL)
         front = newNode;
      else
         rear->next = newNode;
      rear = newNode;
   }
}
template <class DataType>
DataType QueType<DataType>::Dequeue()
{
   DataType item;
   if (IsEmpty())
      throw EmptyQueue();
   else
   {
      NodeType* tempPtr;
      tempPtr = front;
      item = front->info;
      front = front->next;
      if (front == NULL)
         rear = NULL;
      delete tempPtr;
      return item;
```

```
    }
}
template <class DataType>
void QueType<DataType>::MakeEmpty()
{
    NodeType* tempPtr;
    while (front != NULL)
    {
        tempPtr = front;
        front = front->next;
        delete tempPtr;
    }
    rear = NULL;
}
template <class DataType>
QueType<DataType>::~QueType()
{
    MakeEmpty();
}
template class QueType<int>;
```