Project Report

Topic: Digital Search Tree (DST)

Group-5 Members:

Name	ID
Refat Chowdhury	171 1443 642
Md. Raiyanul Islam	171 2148 642
S. M. Al Faruqui	172 1395 042

Course: CSE 225 (Data Structures and Algorithms)

Section: 4

Faculty: Tamanna Motahar (TMM)

Submission Date: 7th September, 2018

Introduction

Binary trees are a well-known data structure with expected access time O(*log*2*n*), where n is the number of nodes (elements) stored in the tree. However, worst case time is O(n) if the elements are added to the tree in a specific order, which makes the tree unbalanced.

To avoid this, many techniques for balancing binary trees have been proposed. Since most operations on binary trees run in the worst case in O(h) time, where h is the height of the tree, keeping the tree balanced is the key to good performance.

Two well-known balanced trees are the red-black trees and the AVL trees. These trees maintain additional information in each node (color or node height) and manipulate the tree to keep it balanced as nodes are added to or removed from the tree. Both of these techniques are fairly complex, they require keeping additional information in the nodes, such as the node color, and performing fairly complex rotation operations when the nodes are inserted or deleted. The same is true for most other tree balancing techniques, which makes them difficult to implement and unpopular in practice.

Digital Search Tree (DST) is a simple data structure to balance a binary tree. Its structure is very similar to regular BST. In an ordinary binary tree a decision on how to proceed down the tree is made based on a comparison between the key in the current node and the key being sought. DSTs use the values of bits in the key being sought to guide this decision.

Basic Principle:

A digital search tree (DST) is a binary tree whose ordering of nodes is based on the values of bits in the binary representation of a node's key. The ordering principle is very simple: at each level of the tree a different bit of the key is checked; if the bit is 0, the search continues down the left subtree, if it is 1, the search continues down the right subtree. The search terminates when the corresponding link in the tree is NIL. Every node in a DST holds a key and links to the left and right child, just like in an ordinary binary search tree. In contrast, a *trie* does *not* store keys in internal nodes, only in leaves.

Advantages:

- The algorithms for node placement and search are conceptually simple and intuitive; implementing insert and search methods for DSTs requires only a few modifications to the corresponding algorithms for ordinary binary trees.

- The delete algorithm is conceptually even simpler than that of ordinary binary tree.

- Does not require keeping additional information in the nodes to maintain the tree balanced, which removes the space overhead of the other two approaches. The tree is balanced because the depth of the tree is limited by the length (number of bits) of the key.

- In addition to being conceptually simpler, DSTs often outperform other popular balanced trees such as AVL or red-black trees.

Disadvantage/Limitations:

- DST is sensitive to bit distribution. The trees will be skewed if the probability of occurrence of 0 and 1 digits is not equal

- DSTs require access to individual bits of the key, which is not easily achievable in some high-level languages.

- According to Flajolet and Sedgewick, digital search trees are easily confused with *radix search tries*, a different application of essentially the same idea. As a consequence, most programmers are not even aware of their existence, and tend to associate advantages and drawbacks of tries with those of DSTs.

- It was thought that these trees can only be used when all the keys in the tree have equal length, or with variable length keys when no key is a prefix of another. However a simple method for storing arbitrary keys of variable lengths into DSTs was later devised.

- In DST, data is unsorted.

Application:

- **Packet classification:** The process of categorizing packets into flows in an internet router is called packet classification. This is one of the application of DST.

- **IP Routing:** IP means 'Internet Protocol'. DST is used in IPv4 and IPv6.

- **Firewalls:** Firewalls is a network security system, either hardware or software based that controls incoming and outgoing network traffic based on a set of rules. DST is also used in this program.

Insertion Algorithm:

To insert an element in DST, we have to follow 4 steps:

1. Convert the keys to their binary bit form. If tree is empty then insert the element in root.
2. If tree is not empty then, and if left most bit of keys is **0** then go to left sub tree. If left most bit is **1** then go to right sub tree.
3. If 1st sub tree of the tree is full then, if 2nd left most bit is **0** then go left sub tree of 1st sub tree. If 2nd left most bit is **1** then go right sub tree of 1st sub tree.
4. This process will continued until the element find any empty sub-tee according to its bites.

Time Complexity: O(b) [b is number of bits in the key]

Search Algorithm:

Searching is based on binary representation of data. For the worst case time complexity of searching is O(b), where b is number of bits in search key. Average search time per operation is O (log N). N is height of tree.

Searching the key k in tree, we have to follow 4 possible steps:

1. If tree is empty then k is not found.
2. If left most bit of k is **0** then go to left sub tree .If left most bit of k is **1** then go to right sub tree, if k matches then k found.
3. If k is not matches in 1st sub tree then, if 2nd left most bit is **0** then go left sub tree of 1st sub tree. If 2nd left most bit is **1** then go right sub tree of 1st sub tree. If k matches then k found.

4. if k not match then previous process will be continued according to bit sequence of k until last bit or final level of tree. If k does not match then return k is not found.

Time Complexity: O(b) [b is number of bits in the key]

Delete Algorithm:

For deleting a node in DST, we may face 3 possible case.

1. With no child.
2. With one child
3. With two child

For deleting a node called k then we have to follow some steps:

1. If tree is empty then return ‘tree is empty’ .if tree is not empty then search for node k in DST.
2. If node k is found and it has no child then simply remove k.
3. If k found and it has one child then link this child with the parent of k and remove k.
4. IF k is found and it has two child then remove k and link any of his child with the parent of k.

Time Complexity: O(b) [b is number of bits in the key]

Division of responsibilities

Refat Chowdhury (ID: 1711443642): Search and Delete functions.

Md. Raiyanul Islam (ID: 1712148642): Power point presentation slides and algorithms.

S. M. Al Faruqui (ID: 1721395042): Insert function, template code and report.

Source Code

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();
    bool searchItem(DataType);
    void makeTreeEmpty();
    void getInOrder(QueType<int>&);
    void getPreOrder(QueType<int>&);
    void getPostOrder(QueType<int>&);
    void counter(int*, tree_node*);
    bool search(DataType);
    int countNode();

};

#endif // DIGITALSEARCHTREE_H_INCLUDED
```

quetype.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();
    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)
            {
                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>;
```

quetype.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>;
```