Assignment 8

Title: AVL Tree

Problem Statement: A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

Objective: To understand concept of Height balance tree.

Outcome:

- Use of Height balance tree.
- Resolved number of comparisons for searching an element.

Requirements: Dell Optiplex 3020 MT, keyboard, monitor, Fedora 20, Eclipse.

Theory:

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

Class Definition:

```
class lnode
{
```

```
    string data;
    lnode* next;
public:
    lnode(string s)
    {
        data = s;
        next = NULL;
    }
    friend class SLL;
};

class SLL
{
    lnode* head;
public:
    SLL()
    {
        head = NULL;
    }
    void create();
    void show();
    void del_at_pos(int);
    void insert_at_beg(string);
    int count();
    friend class node;
};
```

```
class node
{
	string key;
	SLL meaning;
	node *left,*right;
public:
	node(string x)
	{
		key = x;
		meaning.create();
		left = right = NULL;
	}
	friend class AVL;
};

class AVL
{
	node* root;
public:
	AVL()
	{
		root = NULL;
	}
	void create();
	node* insert(node*,string);
	void inorder(node*);
```

```
        node* del_rec(node*,string);
        void del();
        node* find_min(node*);
        node* rotateleft(node*);
        node* rotateright(node*);
        node* LR(node*);
        node* RL(node*);
        int height(node*);
        int bal_fac(node*);
        node* check_rotate(node*);
        void temp();
        void search(node*,string);
        void update_meaning(node*,string,int);
};
```

Pseudo code:

```
void SLL::create()
{
   string s;
   while(1)
   {
      cout << "Enter meaning: ";
      getline(cin,s);
      if(s == "-1")
      {
         return;
      }
```

```
        lnode* p;
        if(head == NULL)
        {
            head = new lnode(s);
            p = head;
        }
        else
        {
            p->next = new lnode(s);
            p = p->next;
        }
    }
}

void SLL::show()
{
    lnode* p = head;
    int i = 0;
    while(p != NULL)
    {
        cout << i+1 << ". " << p->data << "\n";
        p = p->next;
        i++;
    }
}
```

```
void SLL::insert_at_beg(string s)
{
   lnode* p = new lnode(s);
   p->next = head;
   head = p;
}

int SLL::count()
{
   int i=0;
   lnode* p = head;
   while(p != NULL)
   {
      i++;
      p = p->next;
   }
   return i;
}

void SLL::del_at_pos(int pos)
{
   int cnt = count();
   if(pos < 1 || pos > cnt)
   {
      cout << "Invalid position.";
      return;
```

```
    }
    if(pos == 1)
    {
        lnode* p = head;
        head = p->next;
        delete p;
    }
    else
    {
        lnode* q;
        lnode* p = head;
        for(int i=1;i<pos;i++)
        {
            q = p;
            p = p->next;
        }
        q->next = p->next;
        delete p;
    }
}
void AVL::create()
{
        string x;
        while(1)
        {
                cout << "Key: ";
```

```
                getline(cin,x);
                if(x == "-1")
                {
                        break;
                }
                root = insert(root,x);
        }
}

node* AVL::insert(node* T,string x)
{
        if(T == NULL)
        {
                node* temp = new node(x);
                return temp;
        }
        if(x < T->key)
        {
                T->left = insert(T->left,x);
                if(bal_fac(T) == 2)
                {
                        if(x < T->left->key)
                        {
                                T = rotateright(T);
                        }
                        else
```

```
                {
                        T = LR(T);
                }
        }
        return T;
}
if(x > T->key)
{
        T->right = insert(T->right,x);
        if(bal_fac(T) == -2)
        {
                if(x < T->right->key)
                {
                        T = RL(T);
                }
                else
                {
                        T = rotateleft(T);
                }
        }
        return T;
}
else
{
        cout << "Repeat.\n";
        return NULL;
```

```
        }
}

void AVL::inorder(node *T)
{
        if(T != NULL)
        {
                inorder(T->left);
                cout << T->key << " " << height(T) << " " << bal_fac(T) << "\n" ;
                T->meaning.show();
                cout << "\n\n";
                inorder(T->right);
        }
}

node* AVL::del_rec(node* T,string x)
{
        if(T == NULL)
        {
                cout << "Not found.\n";
                return T;
        }
        if(x < T->key)
        {
                T->left = del_rec(T->left,x);
                T = check_rotate(T);
```

```
        return T;
}
if(x > T->key)
{
        T->right = del_rec(T->right,x);
        T = check_rotate(T);
        return T;
}
if(T->left == NULL && T->right == NULL)
{
        delete T;
        return NULL;
}
if(T->left == NULL)
{
        node* p = T->right;
        delete T;
        return p;
}
if(T->right == NULL)
{
        node* p = T->left;
        delete T;
        return p;
}
node* p = find_min(T->right);
```

```
        T->key = p->key;
        T->meaning = p->meaning;
        T->right = del_rec(T->right,p->key);
        T = check_rotate(T);
        return T;
}

void AVL::del()
{
        string x;
        cout << "Enter key to be deleted: ";
        getline(cin,x);
        root = del_rec(root,x);
}

node* AVL::find_min(node* T)
{
        while(T->left != NULL)
        {
                T = T->left;
        }
        return T;
}

node* AVL::rotateleft(node* T)
{
```

```
        node* p = T->right;
        T->right = p->left;
        p->left = T;
        return p;
}

node* AVL::rotateright(node* T)
{
        node* p = T->left;
        T->left = p->right;
        p->right = T;
        return p;
}

node* AVL::LR(node* T)
{
        T->left = rotateleft(T->left);
        T = rotateright(T);
        return T;
}

node* AVL::RL(node* T)
{
        T->right = rotateright(T->right);
        T = rotateleft(T);
        return T;
```

```
}

int AVL::height(node* T)
{
	if(T == NULL)
	{
		return 0;
	}
	if(T->left == NULL && T->right == NULL)
	{
		return 0;
	}
	int hl = height(T->left);
	int hr = height(T->right);
	if(hl > hr)
	{
		return 1 + hl;
	}
	else
	{
		return 1 + hr;
	}
}

int AVL::bal_fac(node* T)
{
```

```
        int hl = 0,hr = 0;
        if(T->left != NULL)
        {
                hl = 1 + height(T->left);
        }
        if(T->right != NULL)
        {
                hr = 1 + height(T->right);
        }
        return hl-hr;
}

node* AVL::check_rotate(node* T)
{
        if(bal_fac(T) == 2)
        {
                if(bal_fac(T->left) == 1 || bal_fac(T->left) == 0)
                {
                        T = rotateright(T);
                }
                else
                {
                        T = LR(T);
                }
        }
        else if(bal_fac(T) == -2)
```

```
        {
                if(bal_fac(T->left) == -1 || bal_fac(T->left) == 0)
                {
                        T = rotateleft(T);
                }
                else
                {
                        T = RL(T);
                }
        }
        return T;
}

void AVL::search(node* T,string x)
{
   int cnt = 0;
   int flag = 0;
   while(1)
   {
      cnt++;
      if(T == NULL)
      {
         cout << "Not Found.\n";
         return;
      }
      if(T->key == x)
```

```
        {
            break;
        }
        if(x < T->key)
        {
            T = T->left;
        }
        else if(x > T->key)
        {
            T = T->right;
        }
    }
    cout << T->key << " " << height(T) << " " << bal_fac(T) << "\n" ;
    cout << "Number of searches required: " << cnt << endl;
    T->meaning.show();
    cout << "\n\n";
}

void AVL::update_meaning(node* T,string x,int a)
{
    int flag = 0;
    while(1)
    {
        if(T == NULL)
        {
            cout << "Not Found.\n";
```

```
        return;
    }
    if(T->key == x)
    {
        flag = 1;
        break;
    }
    if(x < T->key)
    {
        T = T->left;
    }
    else if(x > T->key)
    {
        T = T->right;
    }
}
if(a == 1)
{
    string s;
    cout << "Enter new meaning: ";
    getline(cin,s);
    T->meaning.insert_at_beg(s);
}
else
{
    int p;
```

```
        cout << "Enter position of meaning: ";
        cin >> p;
        T->meaning.del_at_pos(p);
    }
}

void AVL::temp()
{
          cout << "1. Create Dictionary.\n";
          cout << "2. Display Dictionary.\n";
          cout << "3. Search from Dictionary.\n";
          cout << "4. Insert key into Dictionary.\n";
          cout << "5. Delete key from Dictionary.\n";
          cout << "6. Insert meaning for key.\n";
          cout << "7. Delete meaning of key.\n";
          cout << "8. Exit.\n";
          int ch;
          string t;
          while(1)
    {
        cout << "Enter choice: ";
        cin >> ch;
        switch(ch)
        {
            case 1:
                cin.ignore(1);
```

```
    create();
    break;

case 2:
    inorder(root);
    break;

case 3:
    cin.ignore(1);
    cout << "Enter key to be searched: ";
    getline(cin,t);
    search(root,t);
    break;

case 4:
    cin.ignore(1);
    cout << "Enter key to be inserted: ";
    getline(cin,t);
    root = insert(root,t);
    break;

case 5:
    cin.ignore(1);
    del();
    break;
```

```
            case 6:
                cin.ignore(1);
                cout << "Enter key to be updated: ";
                getline(cin,t);
                update_meaning(root,t,1);
                break;

            case 7:
                cin.ignore(1);
                cout << "Enter key to be updated: ";
                getline(cin,t);
                update_meaning(root,t,2);
                break;

            case 8:
                return;

            default:
                cout << "Invalid choice.\n";
        }
    }
}
```

Test cases:

1. Create Dictionary.
2. Display Dictionary.
3. Search from Dictionary.
4. Insert key into Dictionary.
5. Delete key from Dictionary.
6. Insert meaning for key.
7. Delete meaning of key.
8. Exit.

Enter choice: 1

Key: cde

Enter meaning: x

Enter meaning: y

Enter meaning: z

Enter meaning: stop

Enter meaning: -1

Key: bcd

Enter meaning: 1

Enter meaning: 2

Enter meaning: stop

Enter meaning: -1

Key: abc

Enter meaning: 1

Enter meaning: 2

Enter meaning: -1

Key: -1

Enter choice: 2

abc 0 0

1. 1

2. 2

bcd 1 0

1. 1

2. 2

3. stop

cde 0 0

1. x

2. y

3. z

4. stop

Conclusion: We have understood and implemented set and performed basic operations successfully.