CS 265, Part I section 2

1. Data Structures and Algorithms (Examples)
2. Experiments with Random Trees

STL merge needs two separate vectors for proper operation.

Wrong implementation:

```
IntVector a(10);

for(i=0;i<5;i++)
        a[i]=2*i;

for(i=5;i<10;i++)
        a[i]=2*(i-5)+1;

IntVectorIt za= a.begin();

merge(za,za+5,za+5,za+10,za);
```

Right implementation:

```
IntVector a(10);
IntVector b(10);
…
IntVectorIt za= a.begin();
IntVectorIt zb= b.begin();

merge(za,za+5,za+5,za+10,zb);
```

Merge-Sort algorithm – a recursive version with STL merge

```
void stl_merge(IntVector& a,IntVector& b,int s,int m,int r)
{
        //Function stl_merge merges segments
        //a[s..m] and a[m+1..r],
        //stores the result in b[s..r] and
        //then copies the result
        //back to a[s..r]
        IntVectorIt ita=a.begin();
        IntVectorIt itb=b.begin();
        merge(ita+s,ita+m+1,ita+m+1,ita+r+1,itb+s);
        for(int i=s;i<=r;i++)
                a[i]=b[i];
}
```

```
void mergesort(IntVector& a,IntVector& b,int s,int r)
{
        if(r<=s) return;
        int m=(r+s)/2;
        mergesort(a,b,s,m);
        mergesort(a,b,m+1,r);
        stl_merge(a,b,s,m,r);
}
```

Arrays

Dynamic allocation of an array of size n

```
int *A=new int[n];
for(i=0;i<n;i++)
    A[i]=i+1;
```

Linked Lists

The structure of nodes

```
struct Node
{
    int data;
    Node *link;
};
```

Head insertion

```
void insert_head(Node*& head,int m)
{
        Node *ptr;
        ptr=new Node;
        ptr->data=m;
        ptr->link=head;
        head=ptr;

}
```

Head removal

```
void remove_head(Node*& head)
{
        Node *ptr;
        if(head!=NULL)
        {
                ptr=head;
                head=head->link;
                delete ptr;
        }
}
```

Binary Trees

The structure of nodes

```
struct TNode
{
    int data;
    TNode *left_link;
    TNode *right_link;
};
```

Creating a tree node and placing an integer inside its data field

```
TNode *create(int m)
{
        TNode *newp=new TNode;
        newp->data=m;
        newp->left_link=NULL;
        newp->right_link=NULL;
        return newp;
}
```

Pre-order traversal

```
void preorder(TNode *treep)
{
    if(treep!=NULL)
    {
        cout << treep->data << endl;
        preorder(treep->left_link);
        preorder(treep->right_link);
    }
}
```

In-order traversal

```
void inorder(TNode *treep)
{
        if(treep!=NULL)
        {
                inorder(treep->left_link);
                cout << treep->data <<endl;
                inorder(treep->right_link);
        }
}
```

Post-order traversal

```
void postorder(TNode *treep)
{
        if(treep!=NULL)
        {
                postorder(treep->left_link);
                postorder(treep->right_link);
                cout << treep->data <<endl;
        }
}
```

Inserting a tree node into a binary search tree

```
TNode *insert(TNode *treep,TNode *newp)
{
        if(treep==NULL)
                return newp;
        if(newp->data == treep->data)
                cout << "Number " << newp->data <<
                                " is already included."<<endl;
        else if(newp->data < treep->data)
                treep->left_link=insert(treep->left_link,newp);
        else
                treep->right_link=insert(treep->right_link,newp);
        return treep;
}
```

Deleting a binary tree

```
void delete_nodes(TNode *treep)
{
        if(treep!=NULL)
        {
            delete_nodes(treep->left_link);
            if(treep->left_link!=NULL)
            {
                delete treep->left_link;
                treep->left_link=NULL;
            }
```

```
                delete_nodes(treep->right_link);
                if(treep->right_link!=NULL)
                {
                        delete treep->right_link;
                        treep->right_link=NULL;
                }
        }
}
```

```
void delete_tree(TNode*& root)
{
        TNode *treep=root;
        delete_nodes(root);
        delete(treep);
        root=NULL;
}
```

Further operations on trees.

1. Counting the number of nodes
2. Computing the depth of a binary treee
3. Creating full binary search trees

Creating a full binary tree of depth 3.

```
TNode *root=NULL;
TNode *newp;
root=create(8);

newp=create(4);
insert(root,newp);
newp=create(12);
insert(root,newp);
newp=create(2);
insert(root,newp);
…
insert(root,newp);
newp=create(15);
insert(root,newp);
```

After creating the tree we display it. We use commands.

```
preorder(root);
inorder(root);
postorder(root);
cout << count_nodes(root) << endl;
cout << compute_depth(root) << endl;
```

After displaying the tree we delete it.

```
delete_tree(root);
```

This is the output we get.

```
8 4 2 1 3 6 5 7 12 10 9 11 14 13 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 3 2 5 7 6 4 9 11 10 13 15 14 12 8
15
3
```

Creating random binary search trees.

```
root=create(rand()%n);

for(int i=0;i<n-1;i++)
{
        newp=create(rand()%n);
        insert(root,newp);
}
```

This is the output we get for n=15.

Number 12 is already included.
Number 7 is already included.
Number 0 is already included.
Number 9 is already included.
Number 10 is already included.
7 2 0 1 4 6 12 10 9 14
0 1 2 4 6 7 9 10 12 14
1 0 6 4 2 9 10 14 12 7
10
3

Now we try depth tracing during pre-order traversal. Again n=15.

data field 10, depth 0
data field 5, depth 1
data field 0, depth 2
data field 1, depth 3
data field 3, depth 4
data field 2, depth 5
data field 9, depth 2
data field 13, depth 1
data field 11, depth 2
data field 14, depth 2

Experiments with random trees:

1. Small random trees
2. Large random trees
3. Timing experiments

```
void create_random_tree(TNode*& treep, int n)
{
        treep=create_node(rand()%n);
        TNode* newp;

        for(int i=0;i<n-1;i++)
        {
                newp=create_node(rand()%n);
                insert_new_delete_repeated(treep,newp);
        }

}
```

TNode* insert_new_delete_repeated
(TNode* treep,TNode* newp, **int& num_deleted**)

```
int create_random_tree(TNode*& treep, int n)
{
        srand((unsigned)time(NULL));
        treep=create_node(rand()%n);
        TNode* newp;
        int num_deleted=0;

        for(int i=0;i<n-1;i++)
        {
                newp=create_node(rand()%n);
                insert_new_delete_repeated
                                        (treep,newp,num_deleted);
        }
        return num_deleted;
}
```

Examples of output:

(i)
Number of nodes inside the tree: 636
Depth: 19
Number of deleted nodes: 364

(ii)
Number of nodes inside the tree: 622
Depth: 18
Number of deleted nodes: 378

We add lookup operation and we start timing experiments.

```
num_deleted=create_random_tree(root,n);
 …
int counter=0;
clock_t start=clock();
for(int i=0;i<n;i++)
{
        ptr=lookup(root,i);
        if(ptr!=NULL)
                if(ptr->data==i)
                        counter++;
}
clock_t stop=clock();
```

Examples of output:

(i)
Enter the size of the range of random numbers: 200000
Number of nodes inside the tree: 32704
Depth: 34
Number of deleted nodes: 167296
Total search time: 150
Number of successfully found nodes: 32704
All nodes of the random tree were found

(ii)
Enter the size of the range of random numbers: 200000
Number of nodes inside the tree: 32690
Depth: 33
Number of deleted nodes: 167310
Total search time: 110
Number of successfully found nodes: 32690
All nodes of the random tree were found