1. Programming Practice – Data Structures and Algorithms (C++)

**Binary Search Algorithm**
The code is contained in the directory binary_search.

The following algorithm takes an ordered array `a` as an input and it searches for item `v`, by iteratively splitting the list into two segments `a[l..m-1]` and `a[m+1..r].`

```cpp
int search(int a[], int v, int l, int r)
{
    while(r>=l)
    {
        int m=(l+r)/2;
        if(v == a[m])
            return m;
        if(v<a[m])
            r = m - 1;
        else
            l = m + 1;
    }
    return -1;
}
```

**Testing STL merge function**
Example:

*STL merge needs two separate vectors for proper operation.*

*The code below does not work properly.*

```cpp
IntVector a(10);

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

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

for(i=0;i<10;i++)
cout << a[i] << " ";
cout << endl;

IntVectorIt za = a.begin();
merge(za,za+5,za+5,za+10,za);
```
for(i=0;i<10;i++)
    cout << a[i] << " ";

_This is the output we get:_
0 2 4 6 8 1 3 5 7 9
0 1 1 1 1 1 3 5 7 9

_We need to correct the call to merge function as follows:_
IntVector a(10);
IntVector b(10);

int i;
for(i=0;i<5;i++)
a[i]=2*i;
for(i=5;i<10;i++)
a[i]=2*(i-5)+1;
for(i=0;i<10;i++)
cout << a[i] << " ";
cout << endl;

IntVectorIt za= a.begin();
IntVectorIt zb= b.begin();

merge(za,za+5,za+5,za+10,zb);
for(i=0;i<10;i++)
cout << b[i] << " ";

_Now the output is correct:_
0 2 4 6 8 1 3 5 7 9
0 1 2 3 4 5 6 7 8 9

**Merge-Sort algorithm – a recursive version with STL merge**
The code is contained in the directory `merge_sort_recursive_stl_merge`.

```cpp
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) {
    // Function mergesort is a recursive version of Merge-Sort algorithm. We need vector b to store the results of merging. Both vectors a and b are passed by reference. The algorithm needs memory which is twice the size of the array to be sorted.
    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);
}

A simple test:

We print out the array a before sorting and after sorting

Enter the size of the array to be sorted: 10
Before sorting
1 7 4 0 9 4 8 8 2 4
After sorting
0 1 2 4 4 4 7 8 8 9

For more systematic testing we write a Boolean valued function is_sorted:

bool is_sorted(IntVector&a, int s, int r) {
    for (int i = s; i < r; i++)
        if (a[i] > a[i + 1])
            return false;
    return true;
}

Now we may run many tests and just collect the results.

**Basic Data Structures**

1. **Arrays**

   *Dynamic allocation of an array of size n*

   int *A = new int[n];
   for (i = 0; i < n; i++)
       A[i] = i + 1; // We store n consecutive integers inside A
2. Linked Lists

The structure of nodes
struct Node
{
    int data;
    Node *link;
};

Operations on Linked Lists

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;
    }
}

Creating a linked list of size n and storing consecutive n integers inside its data fields
Node *head = NULL;
for(i = 0; i < n; i++)
    insert_head(head, n - i);

Deleting a linked list
void delete_list(Node* &head)
{
    while(head != NULL)
    {
        remove_head(head);
    }
}

3. Trees

The structure of nodes
struct TNode
{
    int data;
    TNode *left_link;
    TNode *right_link;
};

Operations on Trees (part 1)

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;
}

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

Traversals
1. Array Traversal
   for(i=0;i<n;i++)
       cout << A[i] << endl;

2. Linked List Traversal
   Node *ptr;
   for(ptr=head;ptr!=NULL;ptr=ptr->link)
       cout << ptr->data << endl;

3. Pre-order, In-order and Post-order Tree Traversals
   Pre-order
   void preorder(TNode *treep)
   {
       if(treep!=NULL)
       {
           cout << treep->data << endl;
           preorder(treep->left_link);
           preorder(treep->right_link);
       }
   }

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

Comparing Arrays, Linked Lists and Trees
Arrays:
- Easy to use, provide O(1) access to any item
- Work effectively with fixed-size data sets
- Maintaining a changing set of values can be expensive
- Inserting or deleting a single item requires reorganizing the whole array
- Arrays are unbeatable for relatively static data
- Arrays work well with binary search, merge sort and heap sort

Lists:
- Lists have always exactly the size needed (plus some per-item storage to hold the pointers)
- Lists can be easily rearranged by exchanging a few pointers
- When the items are inserted or deleted other items are not moved
- Inserting/deleting the head and the current elements requires O(1) operations
- It takes O(N) time to search for an item
- If the number of items is unpredictable, a list is usually an appropriate way to store them
- Binary search does not apply to lists
- Lists are unbeatable in implementing stacks

Trees:
- If you need to combine frequent update with fast access, a tree usually is a good solution
- Many operations that take time O(N) in lists require only O(log N) in trees
- The multiple pointers at each node reduce the time complexity of operations by reducing the number of nodes one must visit to find an item
perations on Trees (part 2)

Counting the number of nodes of a binary tree

```c
int count_nodes(TNode *treep)
{
    if(treep!=NULL)
    {
        int l,r;
        l=count_nodes(treep->left_link);
        r=count_nodes(treep->right_link);
        return l+r+1;
    }
    else
    return 0;
}
```

Computing the depth of a binary tree

```c
int compute_depth(TNode *treep)
{
    if(treep!=NULL)
    {
        int l,r,max;
        l=compute_depth(treep->left_link);
        r=compute_depth(treep->right_link);
        if(l>r)
            max=l;
        else
            max=r;
        return max+1;
    }
    else
    return -1;
}
```

Creating full binary search trees

An example code in which a binary search tree of depth 3 is created and displayed is available in the directory bst.

**Problem:** Write down a C++ function creating a full binary search tree of depth d. Your tree should contain the numbers 1,2,3,4,5,…,2^{d+1}-1 inside data fields. Performing in-order traversal should result in visiting the nodes in increasing order: 1,2,3,4,5,…,2^{d+1}-1.

Example: A partial solution of the above problem for d=3.
We insert nodes one by one until we get a full binary search tree of depth 3. The order of insertion is: 8,4,12,2,6,10,14,1,3,5,7,9,11,13,15.
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);
...
inorder(root);
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
The following code is used to create random binary search trees. We generate n random integers in range [0..n-1]. We create nodes containing the numbers we have obtained and we insert them into a binary search tree. The insertion process rejects repeated numbers. We modify function insert so that it automatically deletes the nodes containing the numbers, which already are inside the tree.

root=create(rand()%n);

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

Example 1: We create a random tree with n=15, and then we display it with the same commands as before.

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

This is the output we get:
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

Example 2: Again we create a random tree with n=15, but now we use function trace_depth for the display.

int d=0;
trace_depth(root,d);

Function trace_depth
void trace_depth(TNode *treep, int& d)
{
    if(treep!=NULL)
    {
        cout << "data field " << treep->data;
        cout << ", depth " << d << endl;
        d++;
        trace_depth(treep->left_link,d);
        trace_depth(treep->right_link,d);
        d--;
    }
}

This is the output we get:
Number 10 is already included.
Number 5 is already included.
Number 13 is already included.
Number 14 is already included.
Number 14 is already included.
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

2. Experiments with random trees

Stage 1: Small random trees

Methods we use:
TNode* create_node(int m)
{
    TNode *newp=new TNode;
    newp->data=m;
    newp->left_link=NULL;
    newp->right_link=NULL;
    return newp;
}

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

Main method:
    TNode* root=NULL;
    int n=7;

    create_random_tree(root,n);

    cout << "Pre-order traversal:" << endl;
    preorder(root);
    cout << endl;

    cout << "In-order traversal:" << endl;
    inorder(root);
    cout << endl;

    cout << "Post-order traversal:" << endl;
    postorder(root);
    cout << endl;

    cout << "Number of nodes: ";
    cout << count_nodes(root) << endl;
    cout << endl;

    cout << "Depth: ";
    cout << compute_depth(root) << endl;
    cout << endl;

    cout << "Pre-order traversal with depth tracing:" << endl;
    int d=0;
    trace_depth(root,d);

    delete_tree(root);
Examples of output:

(i)
Number 2 is already included.
Number 4 is already included.

Pre-order traversal:
4 2 0 3 5
In-order traversal:
0 2 3 4 5
Post-order traversal:
0 3 2 5 4

Number of nodes: 5
Depth: 2
Pre-order traversal with depth tracing:
data field 4, depth 0
data field 2, depth 1
data field 0, depth 2
data field 3, depth 2
data field 5, depth 1

(ii)
Number 6 is already included.
Number 6 is already included.
Number 1 is already included.

Pre-order traversal:
2 1 6 3
In-order traversal:
1 2 3 6
Post-order traversal:
1 3 6 2

Number of nodes: 4
Depth: 2

Pre-order traversal with depth tracing:
data field 2, depth 0
data field 1, depth 1
data field 6, depth 1
data field 3, depth 2
Stage 2: Large random trees

Methods we use:

```c
TNode* insert_new_delete_repeated(TNode* treep, TNode* newp, int& num_deleted)
{
    if(treep==NULL)
        return newp;
    if(newp->data == treep->data)
    {
        num_deleted++;
        delete newp;
    }
    else if(newp->data < treep->data)
        treep->left_link =
            insert_new_delete_repeated(treep->left_link, newp, num_deleted);
    else
        treep->right_link =
            insert_new_delete_repeated(treep->right_link, newp, num_deleted);
    return treep;
}

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;
}
```

Main method:

```c
TNode* root=NULL;
int n=1000;
int num_deleted;

num_deleted=create_random_tree(root, n);

cout << "Number of nodes inside the tree: ";
cout << count_nodes(root) << endl;
cout << endl;
```
cout << "Depth: ";
cout << compute_depth(root) << endl;
cout << endl;

cout << "Number of deleted nodes: ";
cout << num_deleted << endl;
cout << endl;

delete_tree(root);

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

Stage 3: We add lookup operation and we start timing experiments.
The code is contained in the directory random_trees_tests.

TNode* lookup(TNode* treep, int m)
{
    while(treep!=NULL)
    {
        if(m==treep->data)
            return treep;
        else if (m<treep->data)
            treep=treep->left_link;
        else
            treep=treep->right_link;
    }
    return NULL;
}

Main method:
int n;
cout << "Enter the size of the range of random numbers: ";
cin >> n;
cout << endl;
TNode* root=NULL;
int num_deleted;

num_deleted=create_random_tree(root,n);

cout << "Number of nodes inside the tree: ";
int num_inserted=count_nodes(root);
cout << num_inserted << endl;
cout << endl;

cout << "Depth: ";
cout << compute_depth(root) << endl;
cout << endl;

cout << "Number of deleted nodes: ";
cout << num_deleted << endl;
cout << endl;

TNode* ptr=NULL;
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();
cout << "Total search time: ";
cout << 1000*(stop-start)/CLOCKS_PER_SEC << endl;
cout << endl;

cout << "Number of successfully found nodes: ";
cout << counter << endl;
cout << endl;

if(num_inserted==counter)
    cout << "All nodes of the random tree were found" << endl;
else
    cout << "Searching failed!!" << endl;

delete_tree(root);

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

Comments:
- If we remove the statement `ptr=lookup(root,i)` from the timing loop, then the measurements of the elapsed time give value 0. We conclude that our timing of search operations is done properly.
- The number of inserted nodes is small compared with $n$. There are few distinct numbers, which are generated. Most of the time we deal with repetitions. Increasing $n$ does not help much.