DATA STRUCTURE TEST 03

All forms of DFS

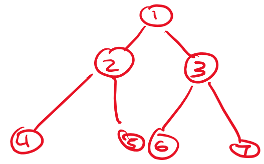
Binary Tree Traversal
- Pre-order
- In Order
- Post Order

Lab Section 13
4-5:50pm
Sage 3101
TA: Matthew
mentors: Aditya & Kajsa & Tyler H.

```
map<string, int>::iterator it = inData.begin();
```

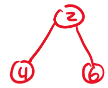
```
void reverse(std::string& str){
    if(str.length()>1){
        reverse(str,0,str.length()-1);
    }
}

void reverse(std::string& str, int i, int j){
    if(i>j){
        return;
    }
    char c = str[i];
    str[i] = str[j];
    str[j] = c;
    reverse(str,i+1,j-1);
}
```

Algorithm: Vist root node, recurse thru left subtree, recurse thru right subtree.

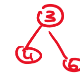


Pre-order (NLR)
- First node is root

(1,2,4,5,3,6,7)



-> left subtree



-> right-subtree

```
void preorder(struct node *root) {
    if (root != NULL) {
        cout<<root->data<<" ";
        preorder(root->left);
        preorder(root->right);
    }
}
```

```
void inorder(struct node *root) {
    if (root != NULL) {
        inorder(root->left);
        cout<<root->data<<" ";
        inorder(root->right);
    }
}
```

In-Order (L,N,R)
(4,2,5,1,6,3,7)

Algorithm: recurse down left tree, then node, then right

Post-order (LRN) (4,5,2, 6,7,3,1)

Algorithm: At each node first go left, right, then chose that node

```
void postorder(struct node *root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        cout<<root->data<<" ";
    }
}
```

map find() Returns an iterator to the element with key-value 'g' in the map if found, else returns the iterator to end

begin() – Returns an iterator to the first element in the map.
end() – Returns an iterator to the theoretical element that follows the last element in the map.
size() – Returns the number of elements in the map.
max_size() – Returns the maximum number of elements that the map can hold.
empty() – Returns whether the map is empty.
pair insert(keyvalue, mapvalue) – Adds a new element to the map.
erase(iterator position) – Removes the element at the position pointed by the iterator.
erase(const g) – Removes the key-value 'g' from the map.
clear() – Removes all the elements from the map.

```
single element (1)  pair<iterator,bool> insert (const value_type& val);
     with hint (2)  iterator insert (iterator position, const value_type& val);
```

Solution:

```
std::set<int> s3;
for (std::set<int>::iterator it = s1.begin(); it != s1.end(); ++it) {
  std::set<int>::iterator it2 = s2.find(*it);
  if (it2 != s2.end()) {
    s3.insert(*it);
    s2.erase(it2);
  }
}
```

If set 1 has k elements and set 2 has n elements, what is the order of your me

Solution: $O(k\ (\log n + \log k))$

```
void findBoxes(const DonutBox& box, DonutBox& current_box, std::vector<DonutBox>& boxes){
    if(box.empty()){
        boxes.push_back(current_box);
        return;
    }
    for(unsigned int i=0; i<box.size(); i++){
        DonutBox tmp_box = box;
        current_box.push_back(box[i]);
        tmp_box.erase(tmp_box.begin()+i);
        findBoxes(tmp_box, current_box, boxes);
        current_box.pop_back();
    }
}

void findBoxes(const DonutBox& box, std::vector<DonutBox>& boxes){
    DonutBox tmp;
    findBoxes(box, tmp, boxes);
}
```

```
bool insert(int val, TreeNode*& p, TreeNode* first, TreeNode* prev = NULL){
    //we've reached a leaf node
    if(!p){
        //set p to a new node (passed by reference, so parent auto updates)
        p = new TreeNode(val);
        //this means this element is not the first, so we can just use prev's next
        if(prev){
            p->next = prev->next;
            prev->next = p;
        }
        else //otherwise, p's next is the first element
            p->next = first;
        return true;
    }
    else if (val < p->value) //if we go left, prev should not be changed
      return insert(val, p->left, first, prev);
    else if (val > p->value) //if we go right, prev should be p
      return insert(val, p->right, first, p);
    else //element already exists in the set
      return false;
}
```

Solution:

```
void RecursiveFor(int i,int max){
  if(i<max){
    x();
    RecursiveFor(i+1,max);
  }
}
```

recursive For loop

DATA STRUCTURE TEST 03

NO	BFS	DFS
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
4.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.
5.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
6.	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.
7.	Here, siblings are visited before the children	Here, children are visited before the siblings

```
//Should advance to the next Node using in-order traversal
//It can point at any Node, not just leaves
rope_iterator& rope_iterator::operator++()
{
    if (ptr_->right != NULL) { // find the leftmost child of the right node
        ptr_ = ptr_->right;
        while (ptr_->left != NULL) { ptr_ = ptr_->left; }
    }
    else { // go upwards along right branches...  stop after the first left
        while (ptr_->parent != NULL && ptr_->parent->right == ptr_) { ptr_ = ptr_->parent; }
        ptr_ = ptr_->parent;
    }
    return *this;
}
```

Operation	Rope	String
Index[1]	O(log n)	O(1)
Split[1]	O(log n)	O(1)
Concatenate (destructive)	O(log n) without rebalancing / O(n) worst case	O(n)
Concatenate (nondestructive)	O(n)	O(n)
Iterate over each character[1]	O(n)	O(n)
Insert[2]	O(log n) without rebalancing / O(n) worst case	O(n)
Append[2]	O(log n) without rebalancing / O(n) worst case	O(1) amortized, O(n) worst case
Delete	O(log n)	O(n)
Report	O(j + log n)	O(j)
Build	O(n)	O(n)

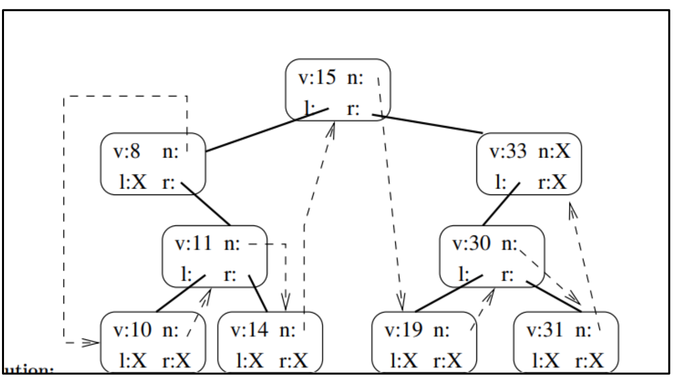
```
template <class T>
TreeNode<T>* FindSmallestInRange(const T& a, const T& b, TreeNode<T>* root, T& best_value){
```

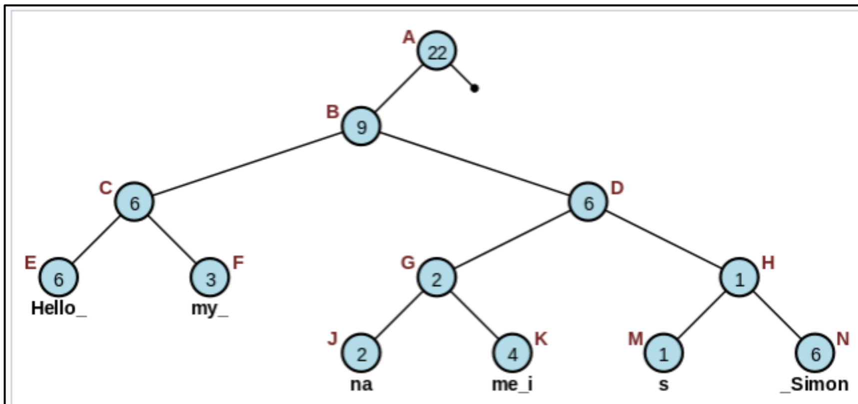
Solution:

```
  if(!root){
    return NULL;
  }

  TreeNode<T>* left_subtree = FindSmallestInRange(a,b,root->left,best_value);
  TreeNode<T>* right_subtree = FindSmallestInRange(a,b,root->right,best_value);
  if(root->value > a && root->value < best_value){
    best_value = root->value;
    return root;
  }
  else if(left_subtree && left_subtree->value == best_value){
    return left_subtree;
  }
  else if (right_subtree){
    return right_subtree;
  }
  return NULL;

}
```

If n is the number of nodes in the tree, what is the worst-case running time of FindSmallestInRange?

Solution: O(n)





A simple rope built on the string of "Hello_my_name_is_Simon".

For set iterator, no first, second
⇒ just dereference the itr ex. (*itr) instead of itr.first