CMPT 125

Introduction to Computing Science and Programming II

November 15, 2021

Assignment 4

- Assignment 4 is due to November 19, 23:59

  https://www.cs.sfu.ca/~ishinkar/teaching/fall21/cmpt125/assignments.html

- You need to submit one file to Canvas – *assignment4.c*
- Please make sure it compiles with the provided makefile

>> make

>> ./run_test4

Topics:

- Stacks – using the provided API, without relying on the implementation details
- Binary Trees
- Q2 is probably the longest, but shouldn‘t take more than 50-60 lines

Midterm – Wednesday, Nov 17

- Wednesday, Nov 17, during your lab
- 3 questions
- 50 minutes
- You may use internet (but it might be more distracting than helpful)
- Use your laptop or the CSIL machine
- The grading will be on a CSIL machine
- You will need to submit one file to Canvas. The name is one of the following

  midterm0930.c, midterm1030.c, midterm1530.c, midterm1630.c

- Please make sure it compiles with the provided makefile

```
>> make
>> ./run_test
```

No lecture on Nov 17

Binary Search Trees

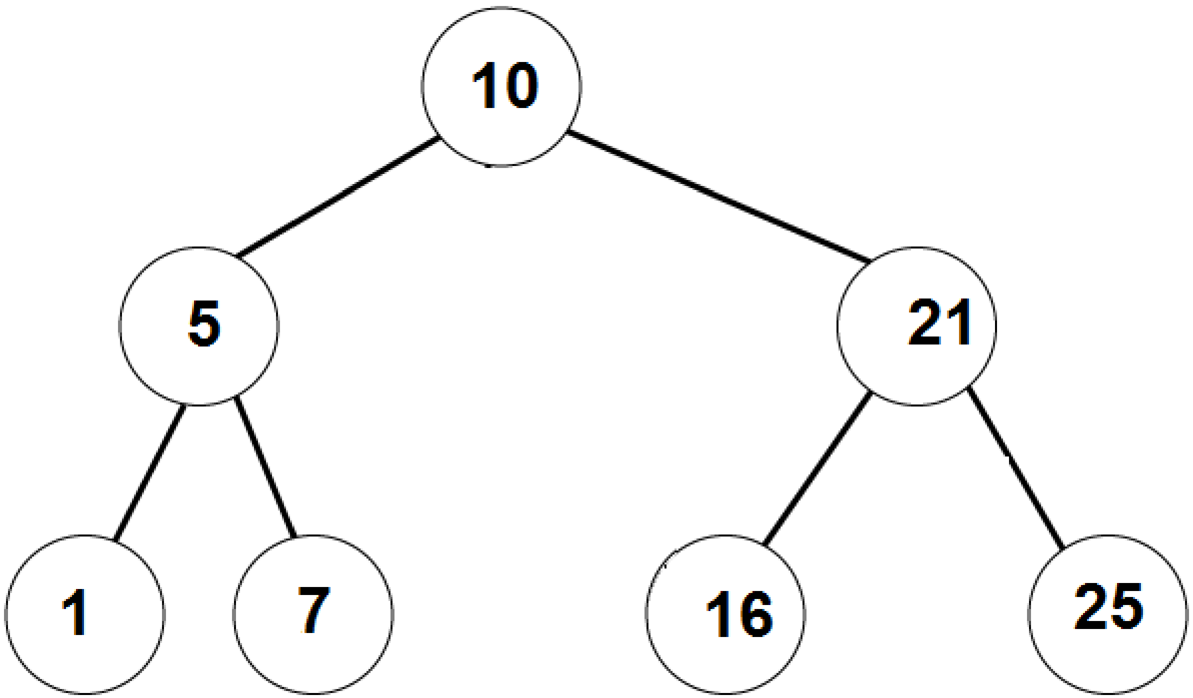
Binary Search Trees

A *binary search tree (BST)* is a binary tree with the following property:

1. For every node, the value in the node is greater or equal than the values in its left child and all its descendants.
2. For every node, the value in the node is smaller or equal than to the value in its right child and all its descendants.

Binary Search Trees

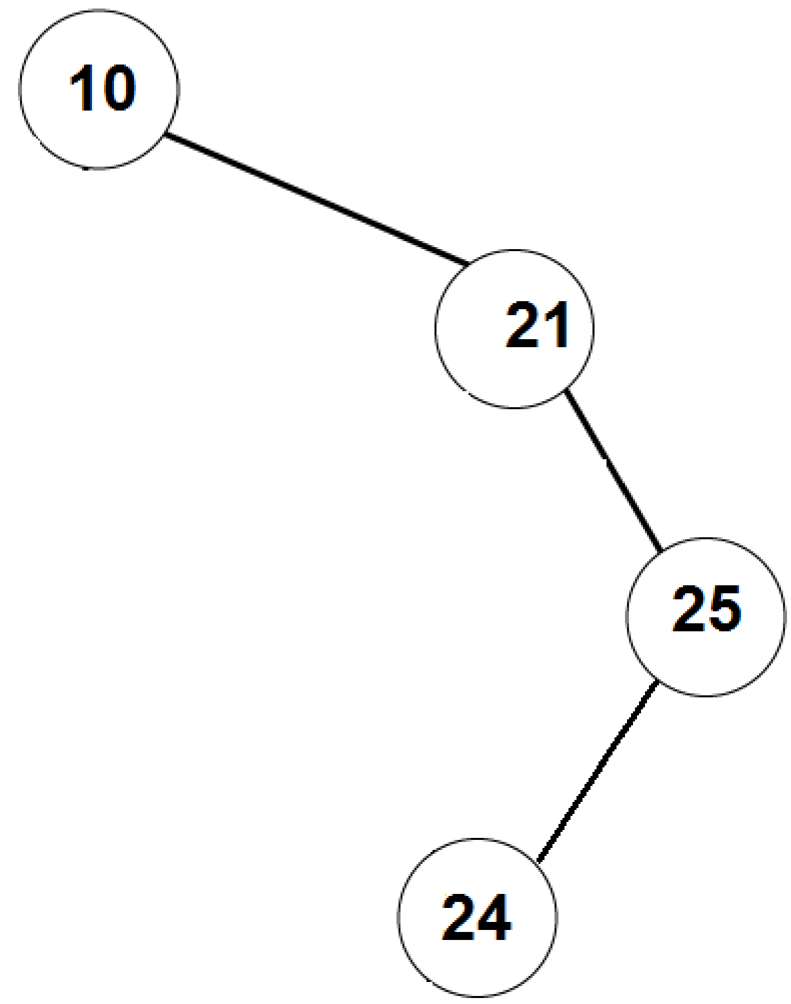
- Example 1:
- A binary search tree can be balanced



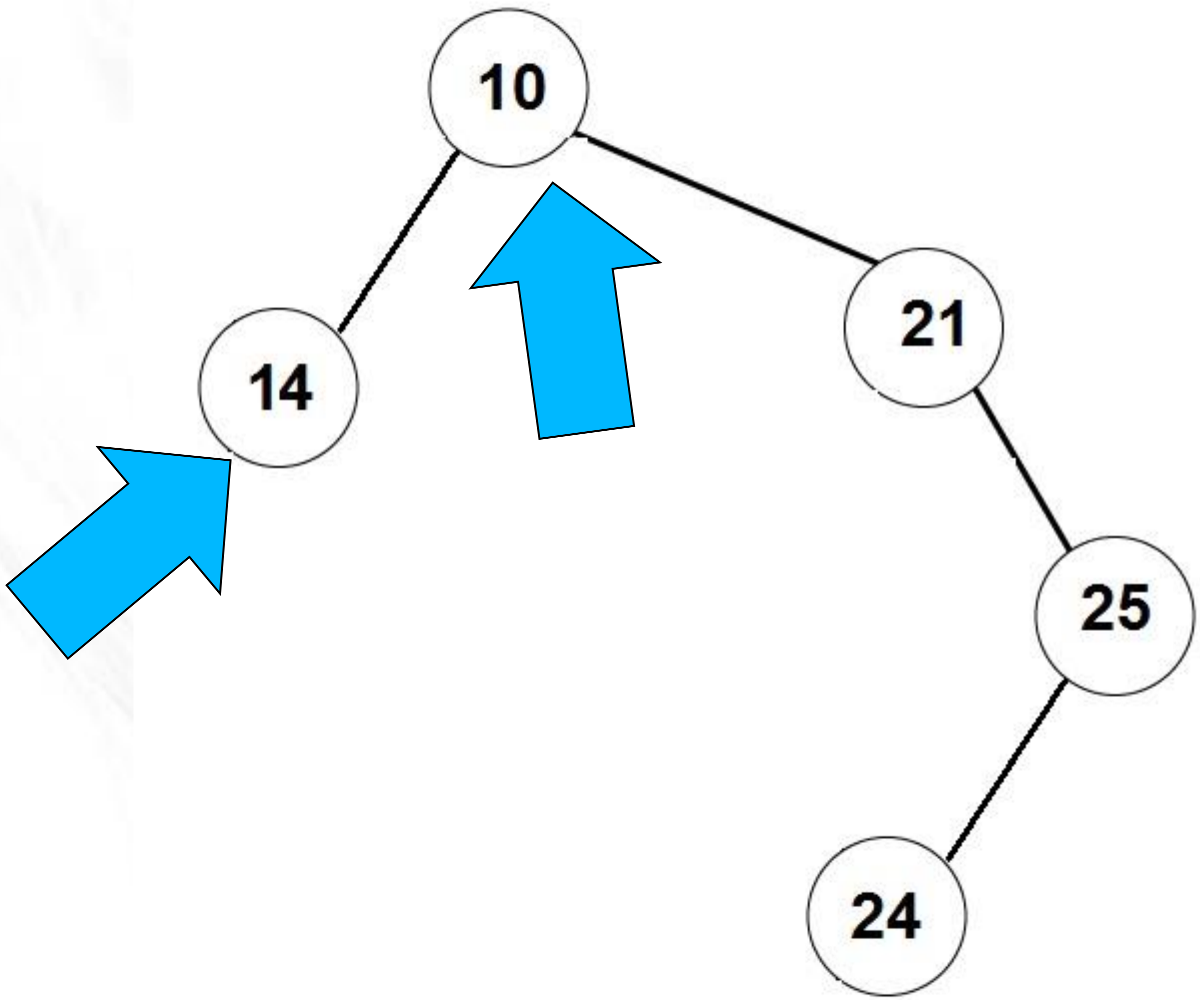
Binary Search Trees

Example 2:

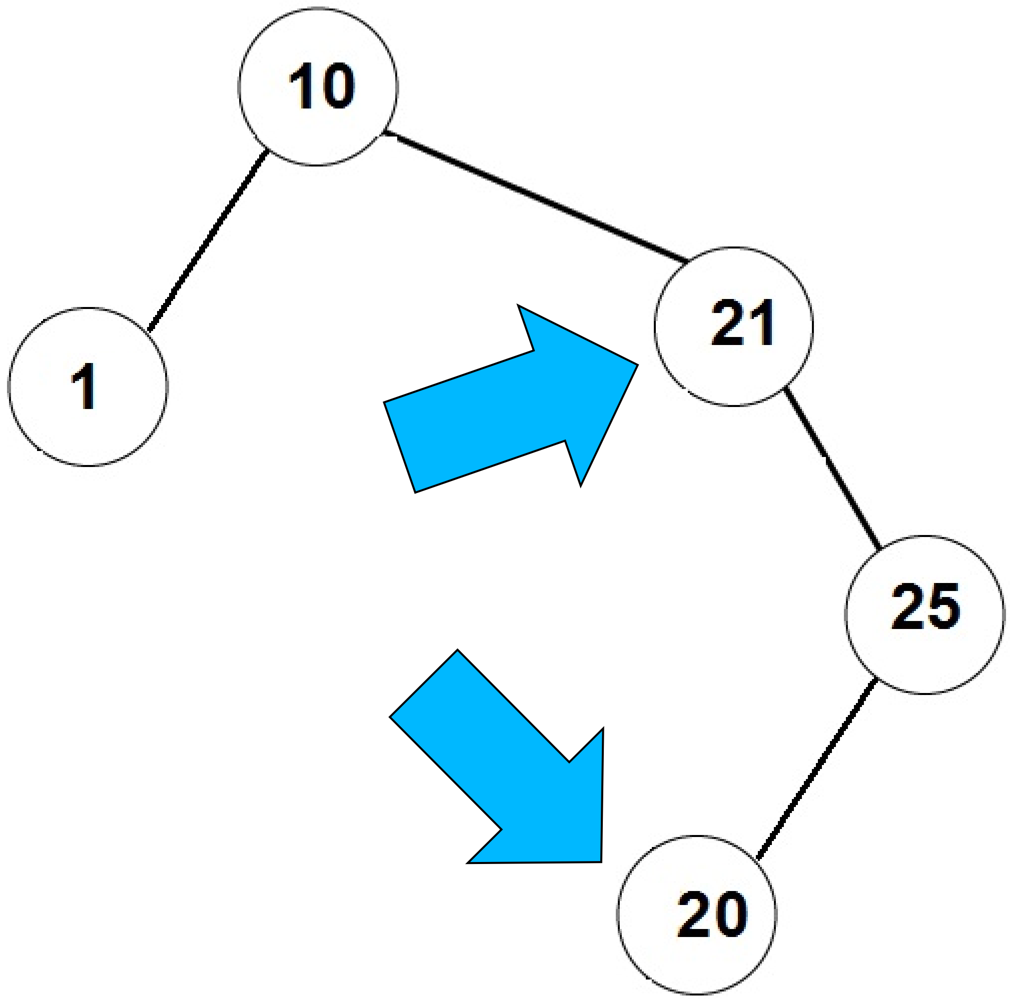
binary search tree can be very skinny / unbalanced



Not a Binary Search Tree



Not a Binary Search Tree



Binary Search Trees

Write a function that gets a binary search tree,
and prints all the values sorted in the increasing order.

```
void print_sorted (BTnode_t* root) {
    if ( root == NULL )
        return;
    print_sorted(root->left);
    print(root->data);
    print_sorted(root->right);
}
```

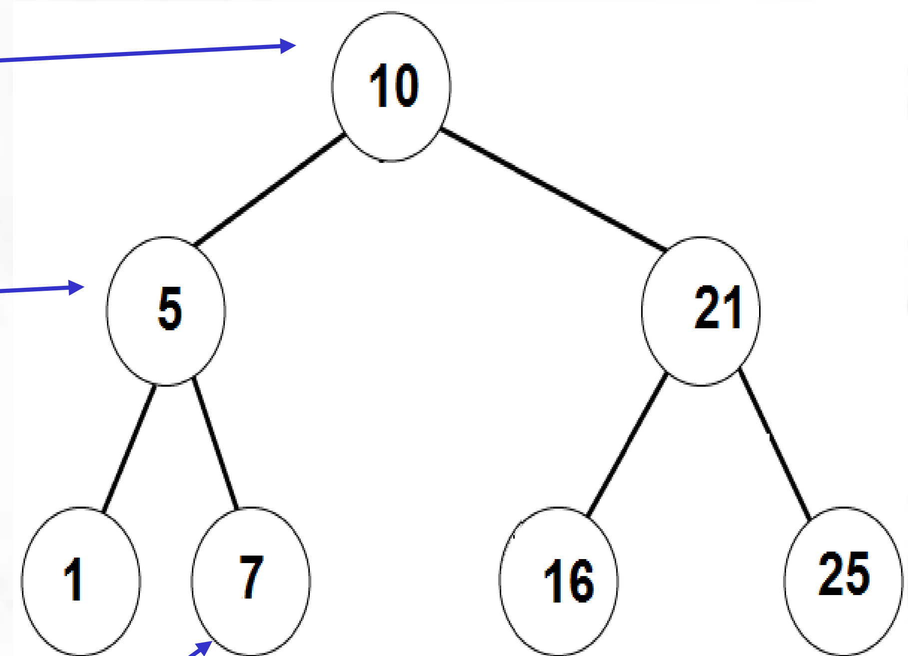
Observation: This is the InOrder traversal on the tree.

Binary Search Trees

Write a function that finds a given item in a BST (or returns NULL).

find(7)

```
BTnode_t* find (BTnode_t* root, int item) {
    if ( root == NULL )
        return NULL;
    if (root->data == item)
        return root;
    if ( root->data > item )
        return find( root->left, item) ;
    else // if ( root->data < item )
        return find( root->right, item) ;
}
```



Found

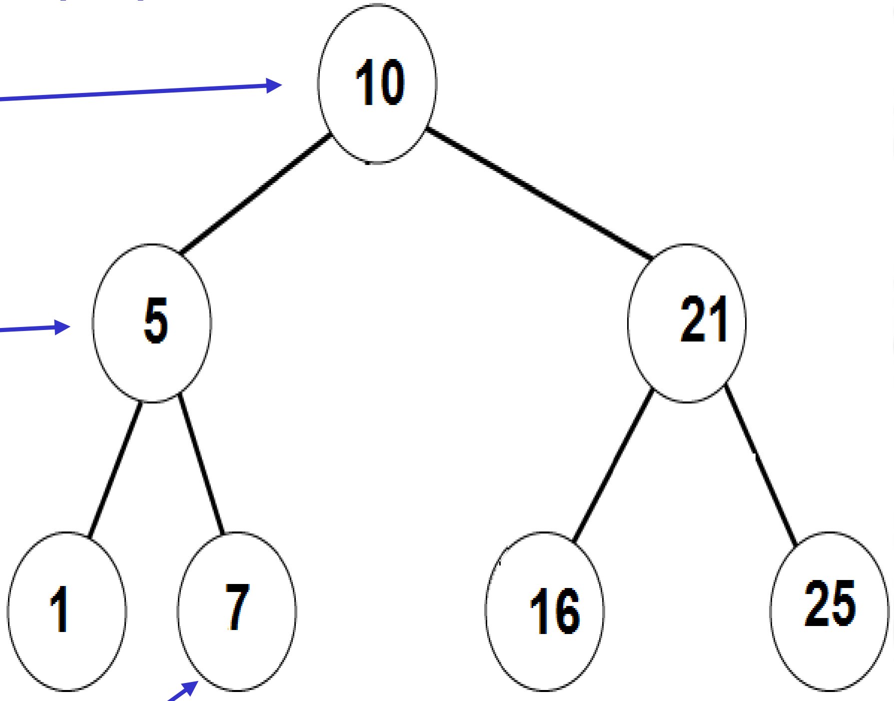
Binary Search Trees

The running time is:
O(depth of the tree)

Write a function that finds a given item in a BST (or returns NULL).

```
BTnode_t* find (BTnode_t* root, int item) {
    if ( root == NULL )
        return NULL;
    if (root->data == item)
        return root;
    if ( root->data > item )
        return find( root->left, item) ;
    else // if ( root->data < item )
        return find( root->right, item) ;
}
```

find(8)



Not found

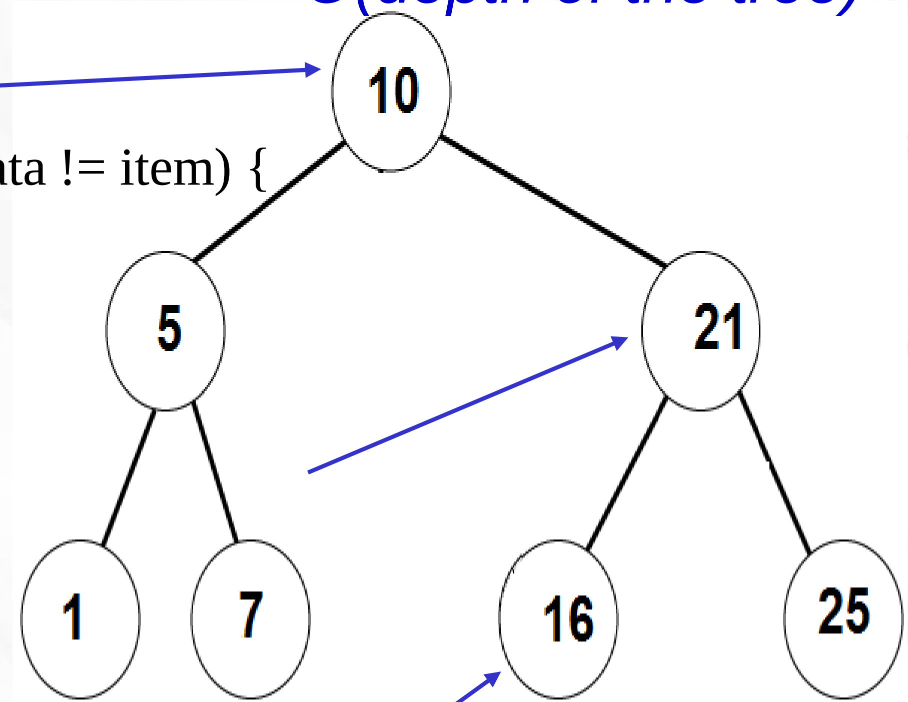
Binary Search Trees

Write an **iterative** function that finds a given item in a BST (or returns NULL)

find(16)

Running time: O(depth of the tree)

```
BTnode_t* find (BTnode_t* root, int item) {
        BTnode_t* current = root;
      while ( current != NULL && current->data != item) {
            if (current->data > item )
                    current = current->left;
            else        // current->data < item
                    current = current->right;
      }
      return current;
}
```



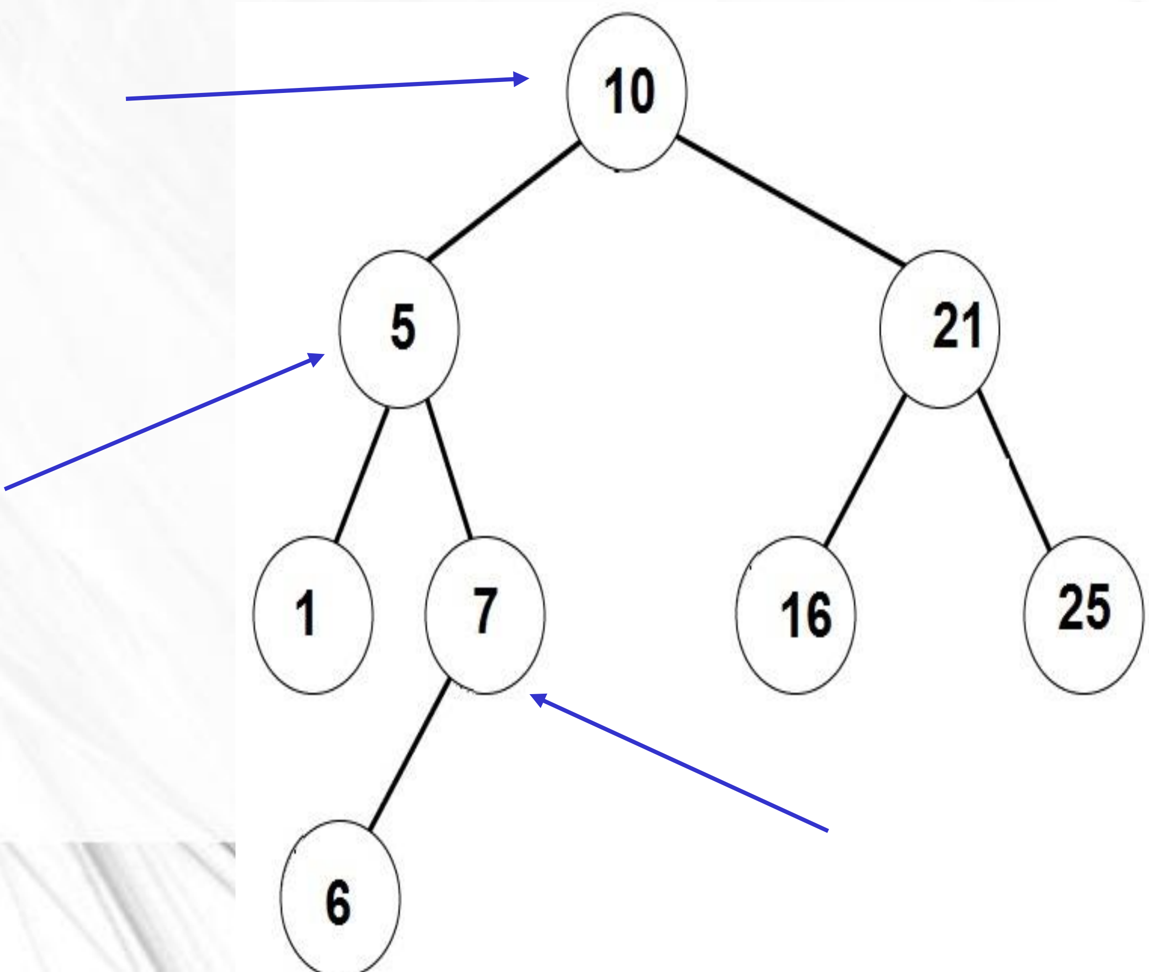
Found

Binary Search Trees

Write a function that adds a given element to a binary search tree.

```
BTnode_t* add_item (binary_tree_t* tree, int item) {
    // implement me
}
```

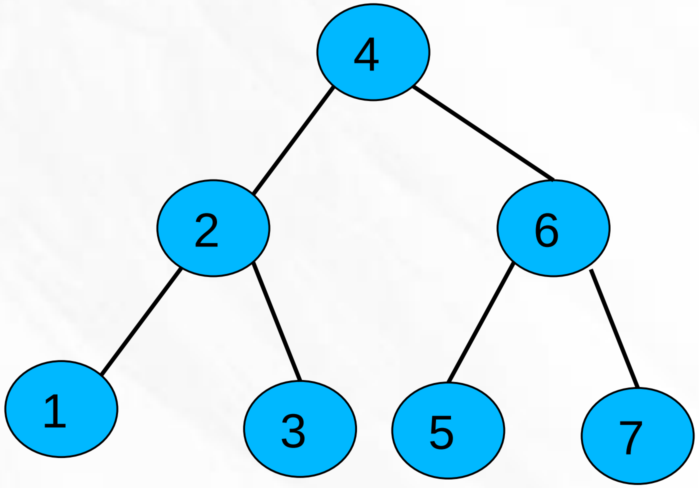
add_item(6)



Binary Search Trees

Create a Binary Search Tree from the following list of insertions:

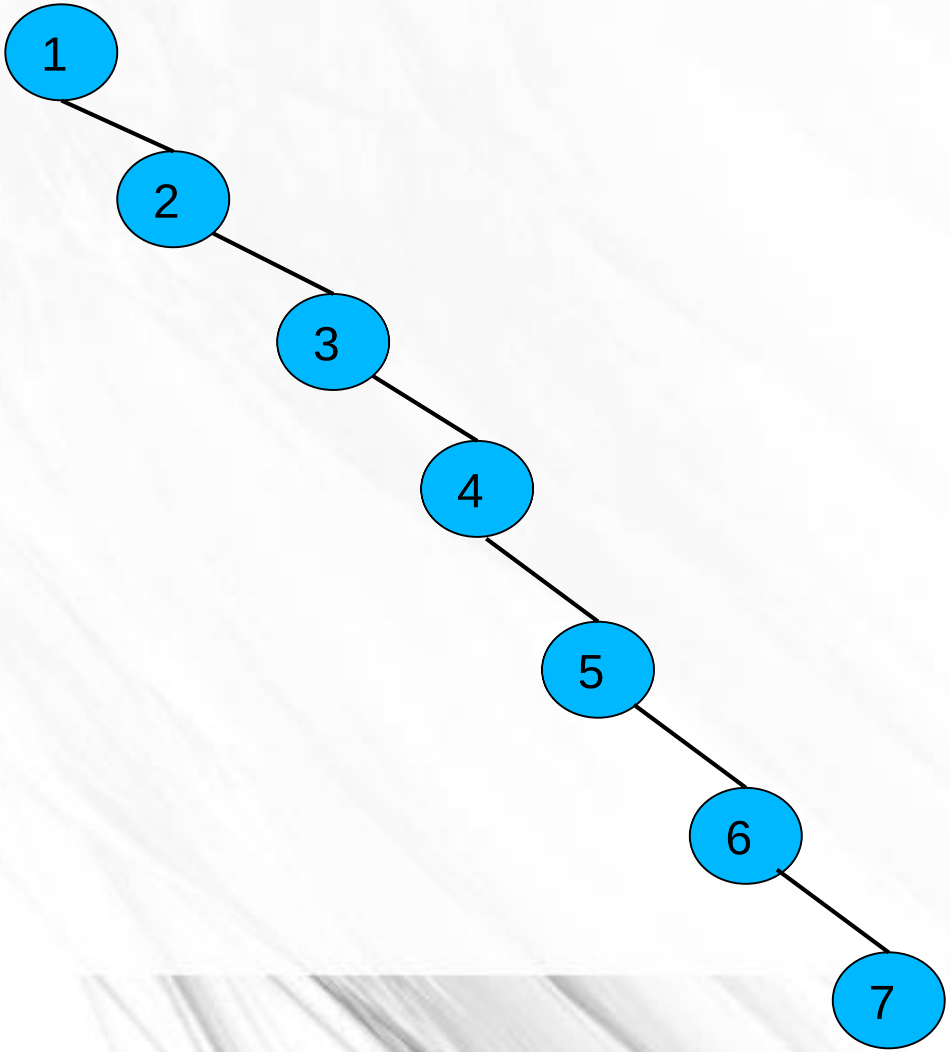
List A: 4, 2, 6, 1, 3 , 5, 7



Binary Search Trees

Create a Binary Search Tree from the following list of insertions:

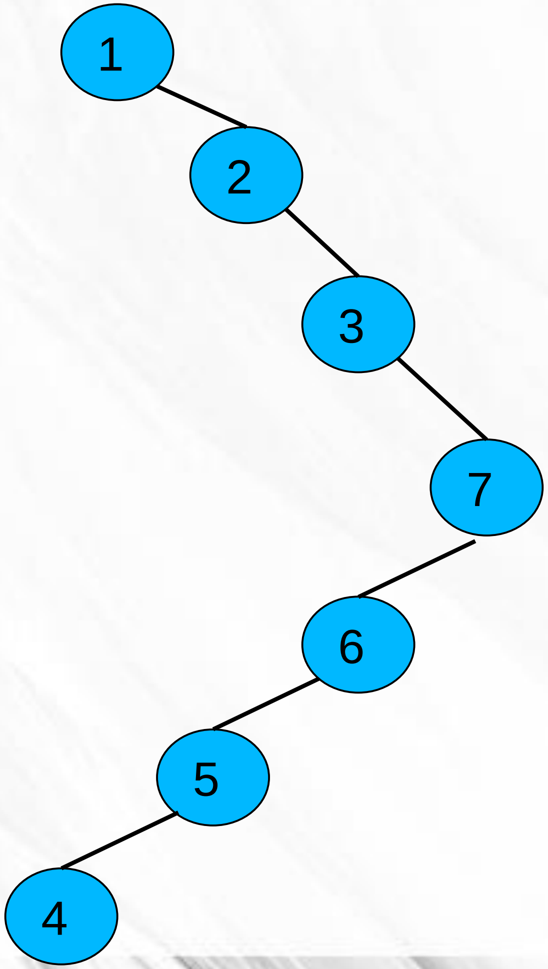
<u>List B</u>: 1, 2, 3, 4, 5, 6, 7



Binary Search Trees

Create a Binary Search Tree from the following list of insertions:

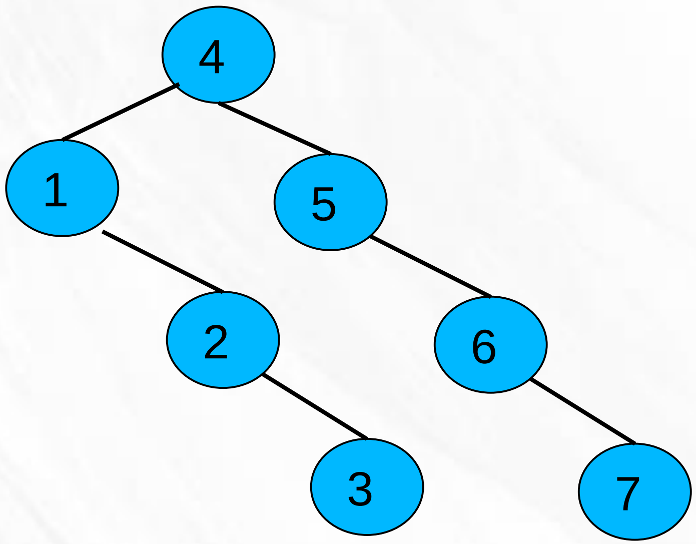
List C: 1, 2, 3, 7, 6, 5, 4



Binary Search Trees

Create a Binary Search Tree from the following list of insertions:

<u>List D</u>: 4, 1, 2, 5, 6, 7, 3



Removing an element from Binary Search Trees

- Write a function that removes a given element from a BST

  - What if we want to remove an element that is not in the tree?
  - What if the remove element has no children?
  - What if the remove element has one child?
  - What if the remove element has two children?

Removing an element from Binary Search Trees

- Step 1: Get a pointer to the node we want to remove

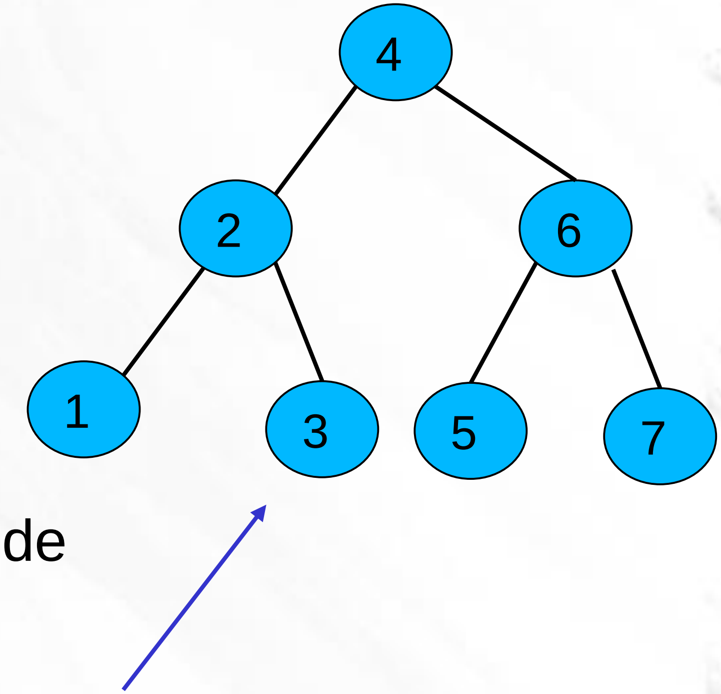
- Removing a node with no children:
  - Just remove the vertex, and update its parent.

- Removing a node with one child:
  - Update the parent to skip over the removed node, and point to its (unique) child.

- Note that if the removed node is the root, then we should update the pointer to the root accordingly.

Remove node with no children

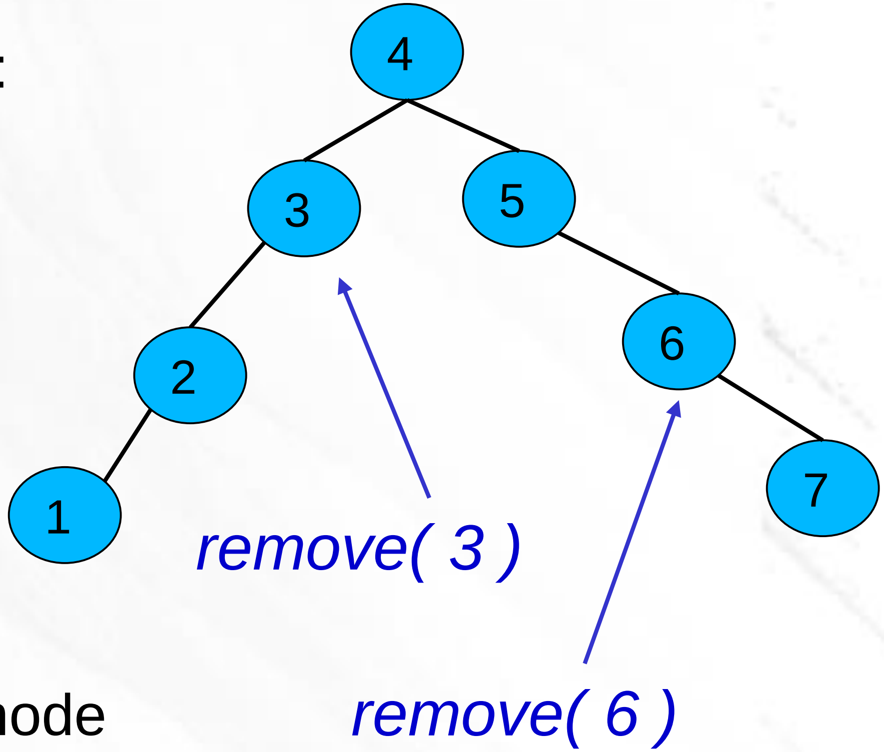
```
Remove (BST_t* tree, BTnode_t* node):
    if (tree-> root == node)
        tree->root = NULL;
    else
        if (node ->parent->left == node)
            node ->parent->left = NULL;
        else   // node ->parent->right == node
            node ->parent->right = NULL;
```



remove(3)

Remove node with one child

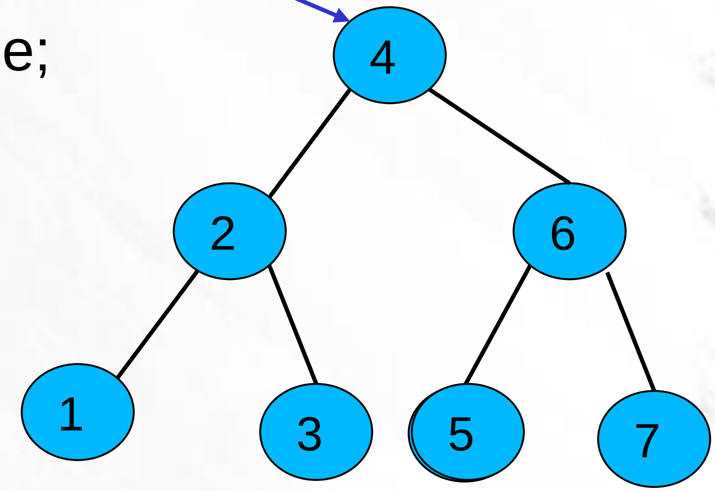
```
Remove (BST_t* tree, BTnode_t* node):
    child = getChild(node);
    if (tree->root == node)
        root = child;
    else
        if (node->parent->left == node)
            node->parent->left = child;
        else   // node->parent->right == node
            node->parent->right = child;
```



Remove node with two children

```
Remove (BST_t* tree, BTnode_t* node)
    next = find successor of node in the tree;
    // next has ≤1 child
    node->value = next->value;
    Remove (tree, next);
```

remove(4)



Successor of 4 is 5

Remove node with two children

```
Remove (BST_t* tree, BTnode_t* node)
    next = find successor of node in the tree;
    node->value = next->value;
    Remove (tree, next);
```

```
// assuming node has right child
find_successor(BST_t* tree, BTnode_t* node) {
    find_min(node->right);
}
```

```
// assuming node != NULL
find_min(BTnode_t* node) {
    BTnode_t* current = node;
    while (current ->left != NULL)
        current  = current->left
    return current
}
```

Balancing a BST

Balancing the tree

How can we make sure that the tree is balanced?

A possible solution: have a function BST_balance(BST_t*)

that creates an <u>*equivalent*</u> balanced tree.

balance()

create a sorted array with all the values in the tree

this is done using *inOrder traversal*

create a new tree from the array

Creating a balanced tree from a sorted array

Idea:

1) Set the median to be the root

2) Construct left and right subtrees recursively.

```
addArrayToTree ( array , first , last )
    if first <= last
        mid = (first+last)/2
    tree.add( array[mid] )
    addArrayToTreeAsLeftSubtree ( array , first , mid-1 )
    addArrayToTreeAsRightSubtree ( array , mid+1 , last )
```

Balancing the tree

Disadvantages of the solution using BST_balance()

1. Add responsibility to the user to invoke BST_balance()
2. The user will need to wait linear time in every invocation of BST_balance.
3. A better solution would be to have a *self-balancing tree* that is makes sure the tree is balanced after each modification (add/remove)
    - AVL trees
    - RedBlack trees
    - 2-3 trees

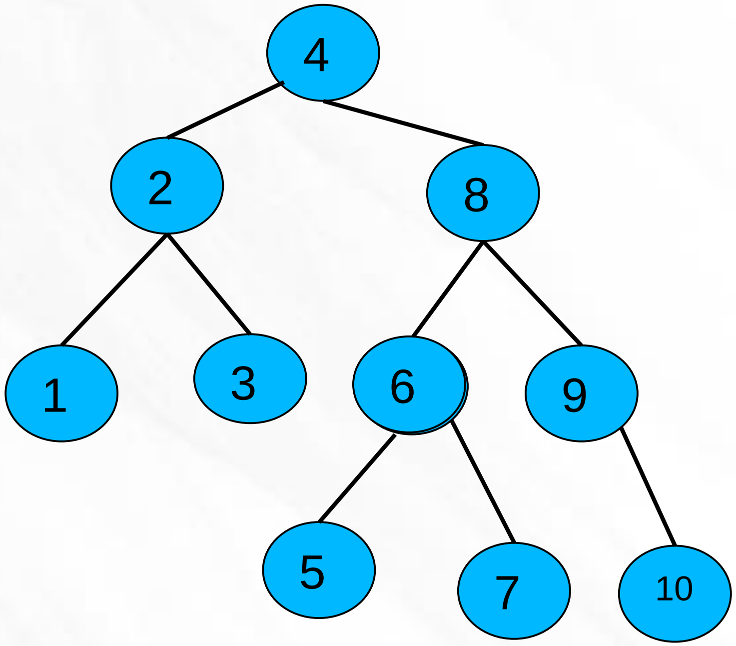
Not in scope for this course. Wait for CMPT 225

Practice Problems

Practice problem 1

Create a Binary Search Tree from the following list of insertions:

List: 4, 2, 8, 1, 3 , 6, 5, 9, 10, 7



Remove 5 from the tree

Remove 9 from the tree

Remove 4 from the tree

Practice problem 2

Write an algorithm that gets a PreOrder traversal of a BST, and returns the tree. Prove that such BST is unique.

<u>Example</u>: Suppose the PreOrder is: 6,4,3,1,5,9,8,7. What is the BST?

Root=6

Left subtree of 6: [4,3,1,5] - elts<6

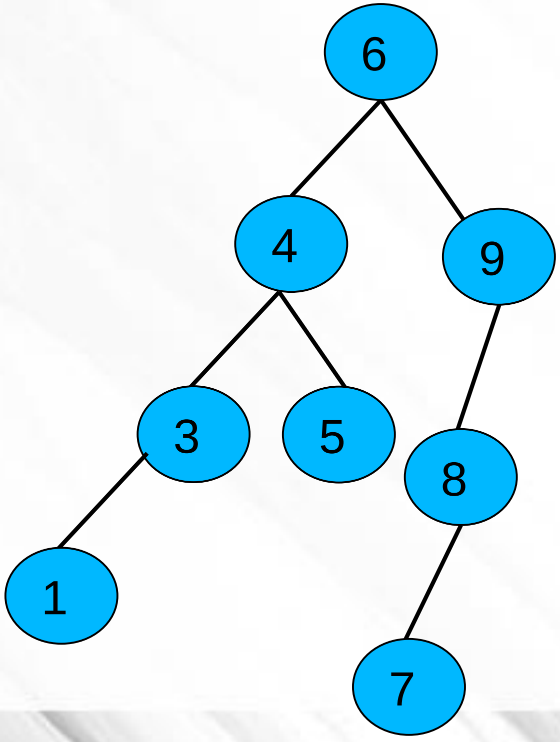
The root is 4.   Left = [3 1] Right = [5]

Look at [3,1] – 3 is the root, 1 is on the left

Right subtree of 6 is [9,8,7] – elts>6

9 is the root [8,7] are both on the left

[8,7]: 8 is the root and 7 is on the left



Practice problem 3

Add to the BST a feature that allows for each node to get the size of the subtree under it in O(1) time.

The running time of the other operations should be

- Find – O(depth of the tree)
- Insert – O(depth of the tree)
- Remove – O(depth of the tree)
- getSize – O(1)

A: (1) add a field size to the struct BTnode

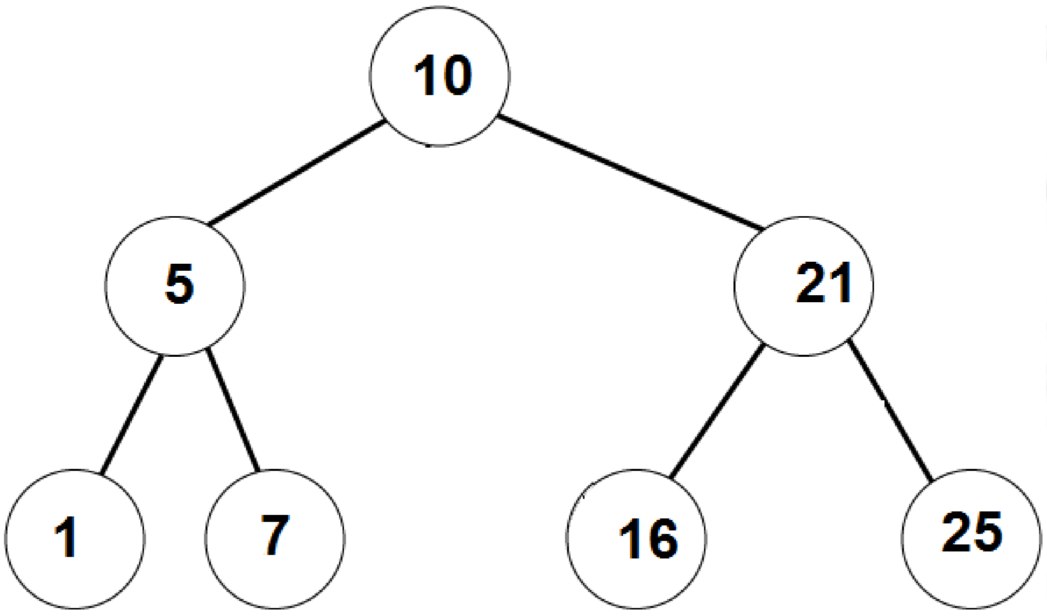
(2) maintain size for all nodes in each operation. Try it!

For example, *insert* updates the size of all ancestors of the new node.

Practice problem 4

Write a non-recursive algorithm that prints PreOrder traversal of a binary tree.

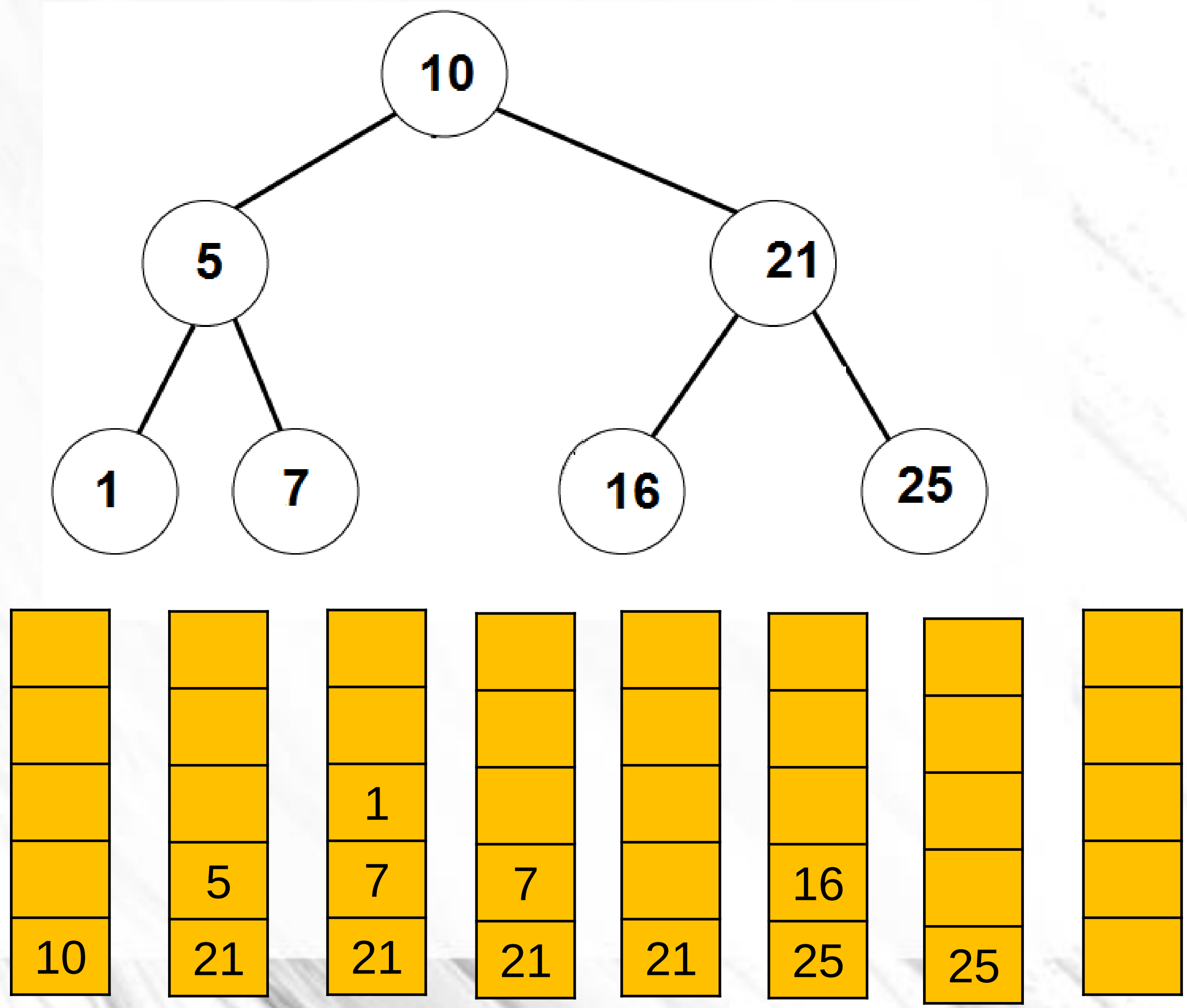
```
PreOrder(root):
    s = create stack of nodes
    s.push(root)
    While s is not empty:
        node = s.pop()
        printf(node->value
        if (node->right != NULL)
            s.push(node->right)
        if (node->left != NULL)
            s.push(node->left)
```



Practice problem 4

Write a non-recursive algorithm that prints PreOrder traversal of a binary tree.

```
PreOrder(root):
    s = create stack of nodes
    s.push(root)
    While s is not empty:
        node = s.pop()
        printf(node->value)
        if (node->right != NULL)
            s.push(node->right)
        if (node->left != NULL)
            s.push(node->left)
```



Practice problem 4

Write a non-recursive algorithm that prints PreOrder traversal of a binary tree.

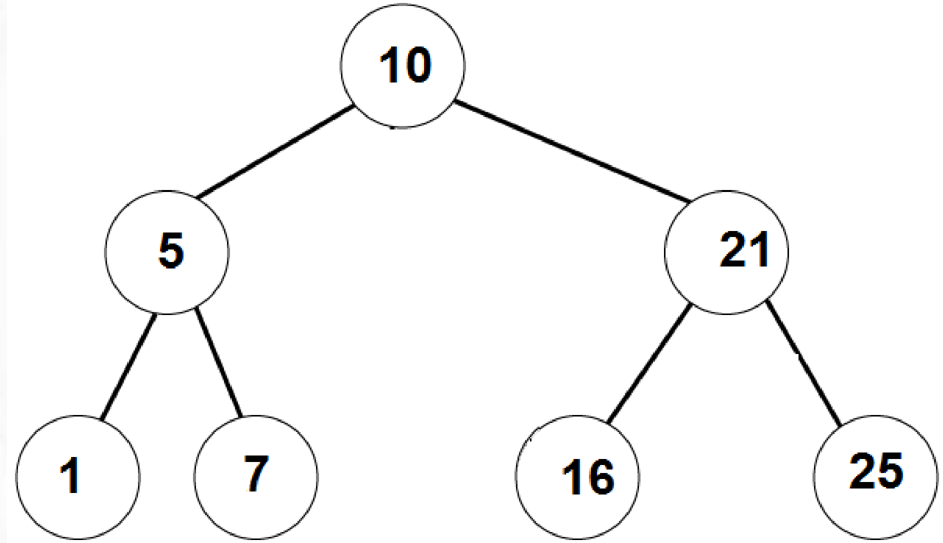
```
PreOrder(root):
    s = create stack of nodes
    s.push(root)
    While s is not empty:
        node = s.pop()
        printf(node->value)
        if (node->right != NULL)
            s.push(node->right)
        if (node->left != NULL)
            s.push(node->left)
```

Implement this algorithm!

What if we replace the stack with a queue?

Practice problem 4

```
BreadthFirstSearch(root):
    q = create queue of nodes
    q.enqueue(root)
    while q is not empty:
        node = q.dequeue()
        printf(node->value)
        if (node->left != NULL)
            q.enqueue(node->left)
        if (node->right != NULL)
            q.enqueue(node->right)
```

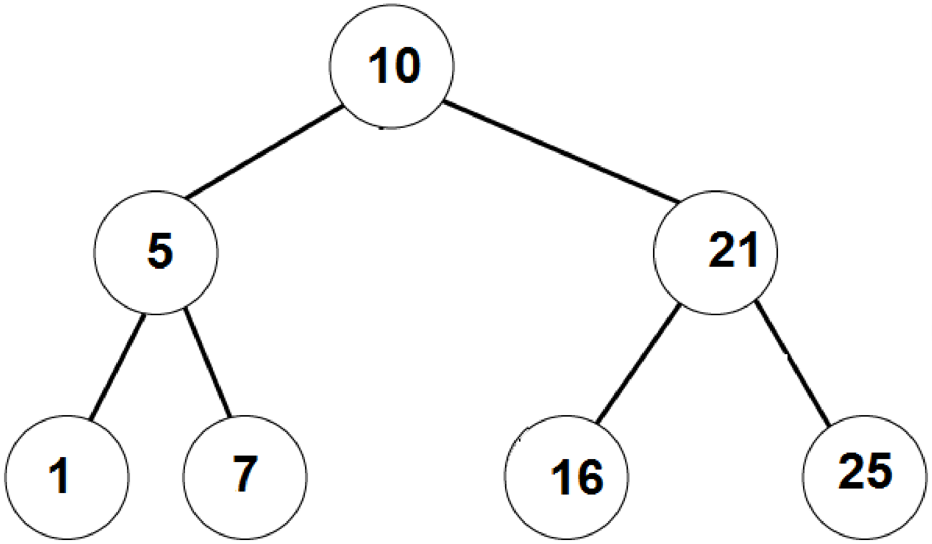


10				*Print 10*
5	21			*Print 5*
21	1	7		*Print 21*
1	7	16	25	*Print 1*
7	16	25		*Print 7*
16	25			*Print 16*
25				*Print 25*

Implement this algorithm!

Practice problem 4

```
BreadthFirstSearch(root):
    q = create queue of nodes
    q.enqueue(root)
    while q is not empty:
        node = q.dequeue()
        printf(node->value)
        if (node->left != NULL)
            q.enqueue(node->left)
        if (node->right != NULL)
            q.enqueue(node->right)
```



These algorithms have applications to

- *Exploring unknown territory*
  Finding shortest paths
- *Some AI tasks*
  Solving puzzles

Questions?

Comments?