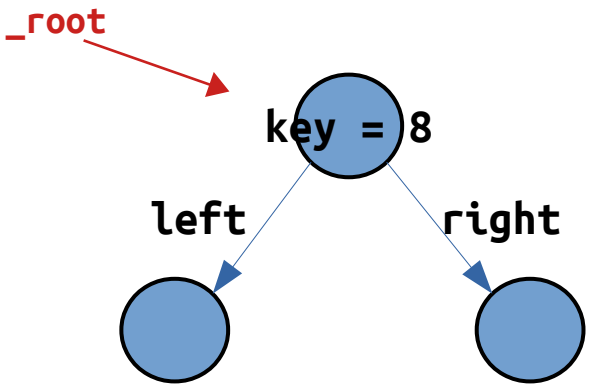
Binary Search Tree

```
 6  class BSTree
 7  {
 8      private:
 9
10          Node* _root;
11
12      public:
13
14          class Node
15          {
16              public:
17                  Node(){}
18                  Node(int key) : key(key), right(nullptr), left(nullptr){}
19                  int   key;
20                  Node* right;
21                  Node* left;
22          };
```



Binary Search Tree – Insertion

```
178 int main()
179 {
180     BSTree myTree;
181
182     std::vector<int> data = {8, 3, 10, 1, 6, 14, 4, 7, 13};
183
184     for(int i = 0; i < data.size(); i++)
185         myTree.insert(data[i]);
```

Binary Search Tree – Insertion

```
void insert(int key)
{
    Node* it = _root;

    Node* newNode = new Node(key); //  new node
    Node* parent = nullptr;          // new node's parent

    while(it != nullptr)
    {
        parent = it; // keep trail of the parent
        it = newNode->key < it->key ? it->left : it->right;
    }

    if(!parent)
    {
        _root = newNode; // tree was empty
    }
    else if(newNode->key < parent->key)
    {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }
}
```

Binary Search Tree – Insertion

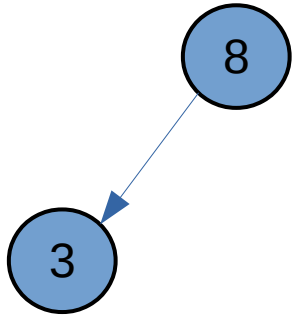
Insert elements: 8 3 10 1 6 14 4 7 13


```
void insert(int key)
{
    Node* it = _root;

    Node* newNode = new Node(key); //  new node
    Node* parent = nullptr;          // new node's parent

    while(it != nullptr)
    {
        parent = it; // keep trail of the parent
        it = newNode->key < it->key ? it->left : it->right;
    }

    if(!parent)
    {
        _root = newNode; // tree was empty
    }
    else if(newNode->key < parent->key)
    {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }
}
```

Binary Search Tree – Insertion

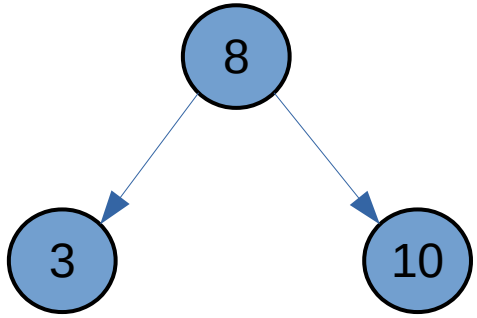
Insert elements: 8 3 10 1 6 14 4 7 13

```
void insert(int key)
{
    Node* it = _root;

    Node* newNode = new Node(key); //  new node
    Node* parent = nullptr;          // new node's parent

    while(it != nullptr)
    {
        parent = it; // keep trail of the parent
        it = newNode->key < it->key ? it->left : it->right;
    }

    if(!parent)
    {
        _root = newNode; // tree was empty
    }
    else if(newNode->key < parent->key)
    {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }
}
```



Binary Search Tree – Insertion

Insert elements: 8 3 10 1 6 14 4 7 13

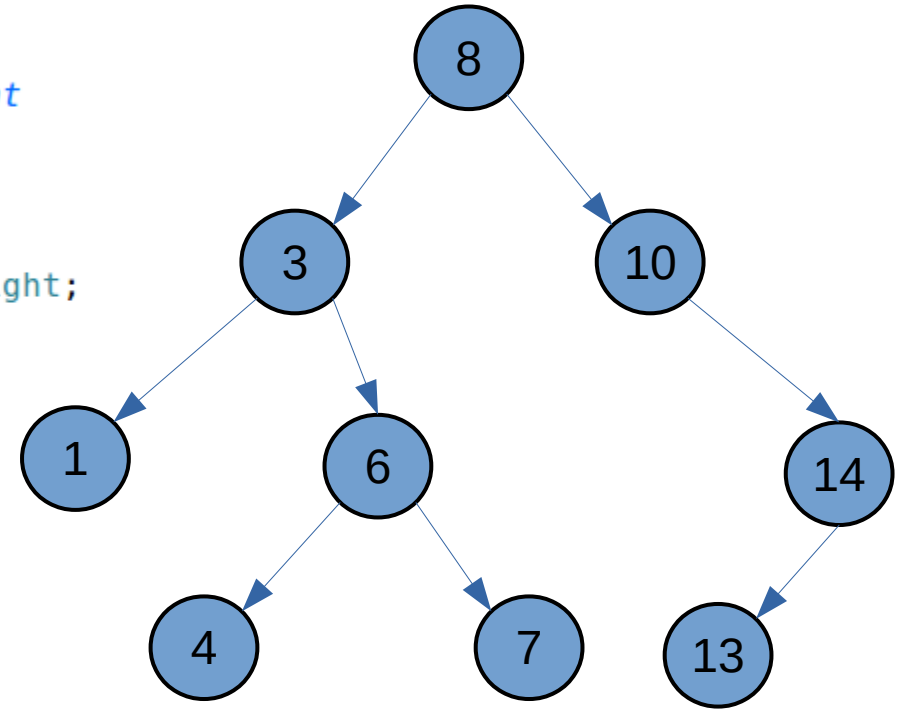



```
void insert(int key)
{
    Node* it = _root;

    Node* newNode = new Node(key); //  new node
    Node* parent = nullptr;          // new node's parent

    while(it != nullptr)
    {
        parent = it; // keep trail of the parent
        it = newNode->key < it->key ? it->left : it->right;
    }

    if(!parent)
    {
        _root = newNode; // tree was empty
    }
    else if(newNode->key < parent->key)
    {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }
}
```

Binary Search Tree – Insertion

Insert elements: 8 3 10 1 6 14 4 7 13

```
void insert(int key)
{
    Node* it = _root;

    Node* newNode = new Node(key); //  new node
    Node* parent = nullptr;          // new node's parent

    while(it != nullptr)
    {
        parent = it; // keep trail of the parent
        it = newNode->key < it->key ? it->left : it->right;
    }

    if(!parent)
    {
        _root = newNode; // tree was empty
    }
    else if(newNode->key < parent->key)
    {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }
}
```



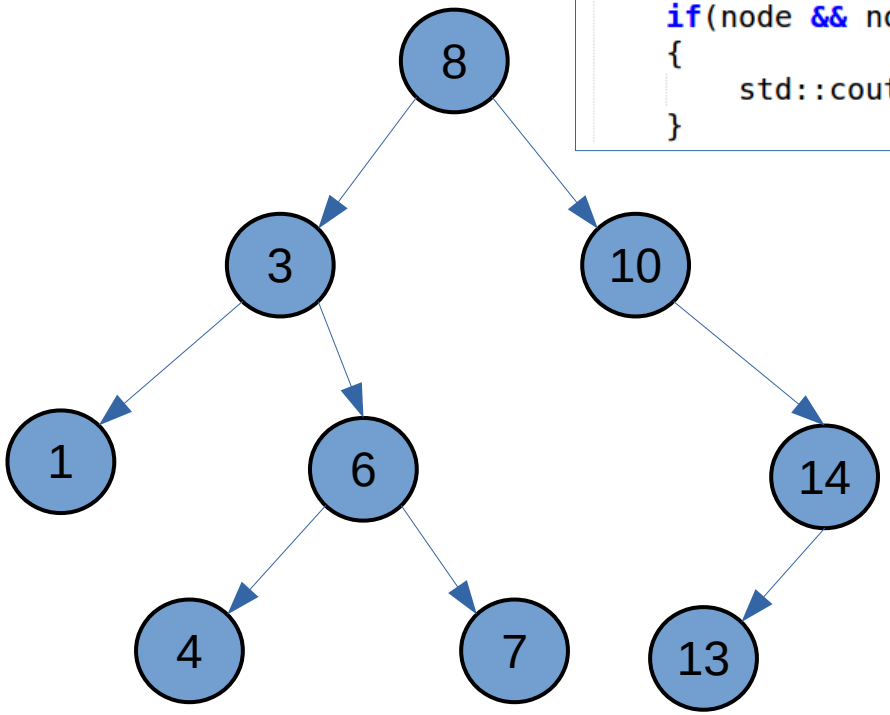
Binary Search Tree – Search

```
// class method to search that returns the pointer to the node with
key k
Node* search(int k)
{
    Node* x = _root;

    while(x is not a null pointer AND
        the key of x is not equal to k)
        {
            if(k < key of x)
                x = left child of x;
            else
                x = right child of x;
        }

    return x;
}
```

Binary Search Tree – Search

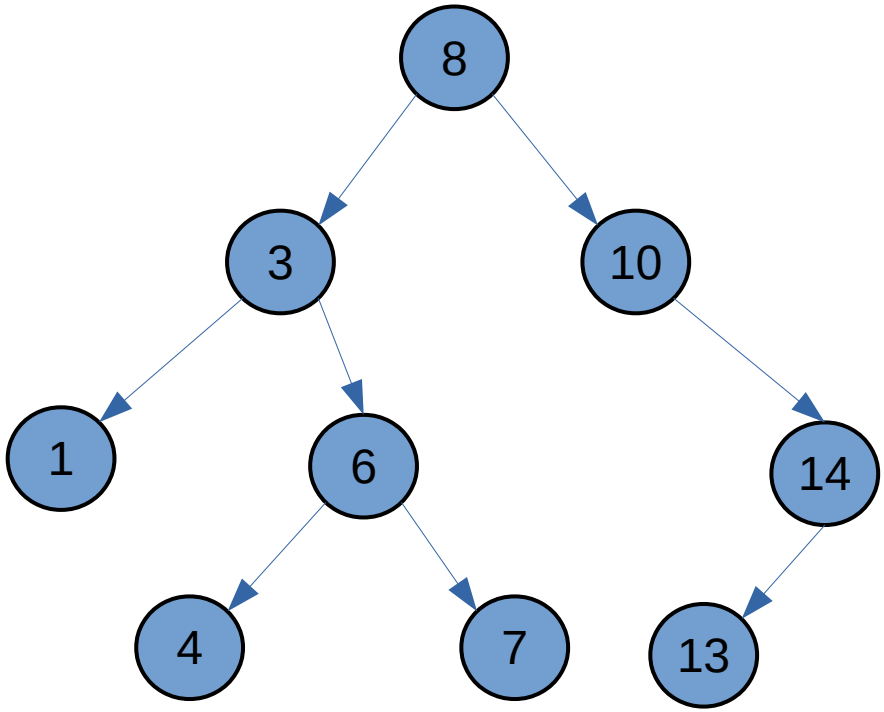
```
int main()
{
    BSTree myTree;

    std::vector<int> data = {8, 3, 10, 1, 6, 14, 4, 7, 13};

    for(int i = 0; i < data.size(); i++)
        myTree.insert(data[i]);

    if(! myTree.search(99) )
    {
        std::cout << "Not Found!" << std::endl;
    }

    BSTree::Node* node = myTree.search(10);

    if(node && node->right)
    {
        std::cout<< "The right son of 10 is: " << node->right->key << std::endl;
    }
```

```
Not Found!
The right son of 10 is: 14
```



Tree Traversals – Inorder

**In a given node: Visit left subtree, then the node,
and finally the right subtree**



Tree Traversals – Inorder

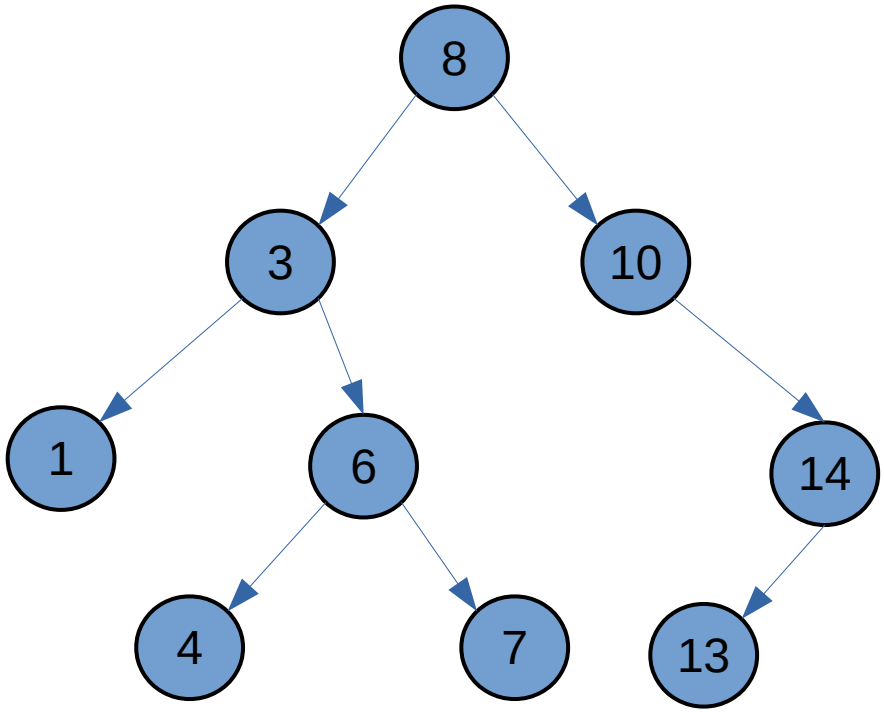
In a given node: Visit left subtree, then the node, and finally the right subtree

```
void inorder(BSTree::Node* currentNode)
{
    if(currentNode)
    {
        inorder(currentNode->left);
        std::cout << currentNode->key << ' ';
        inorder(currentNode->right);
    }
}
```

```
int main()
{
    BSTree myTree;

    std::vector<int> data = {8, 3, 10, 1, 6, 14, 4, 7, 13};

    for(int i = 0; i < data.size(); i++)
        myTree.insert(data[i]);

    BSTree::Node* root = myTree.getRoot();

    inorder(root);
    std::cout << std::endl;
```

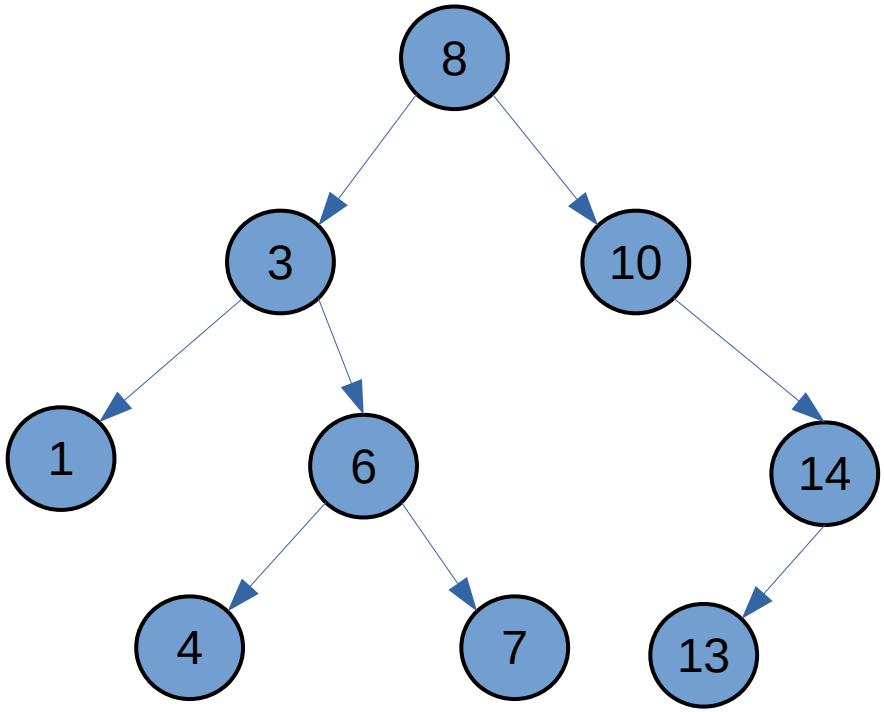
Tree Traversals – Preorder

In a given node: Visit first the node, then the left subtree, and finally the right subtree



Tree Traversals – Postorder

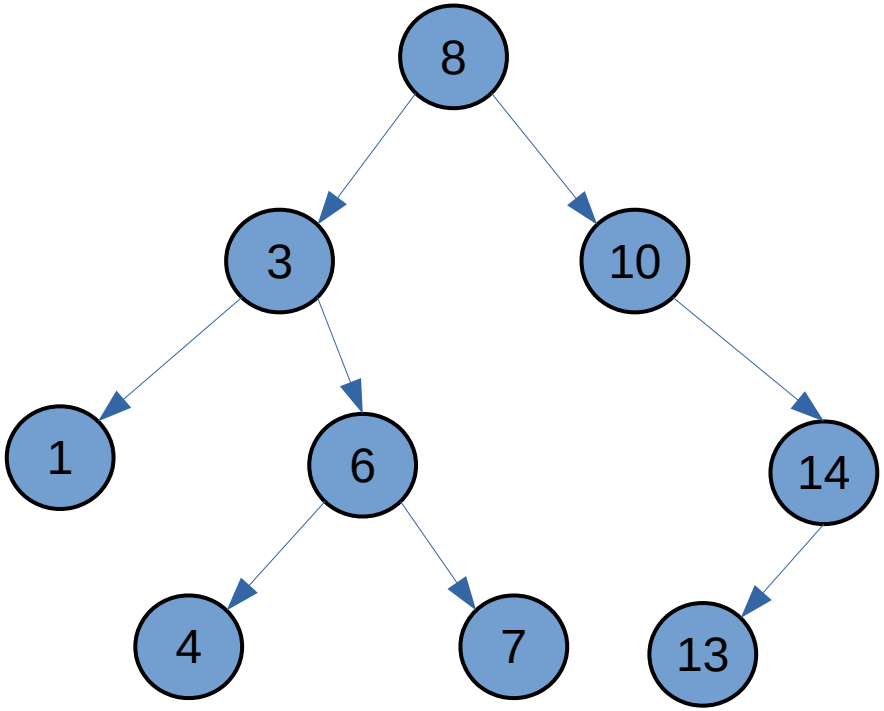
In a given node: Visit first the left subtree, then the right subtree, and finally the node



Tree Traversals – (Iterative) Level-order

Idea: visit level-by-level
This is also called Breadth-First Search (BFS)
Previous order traversals are instead Depth-First Search (DFS)

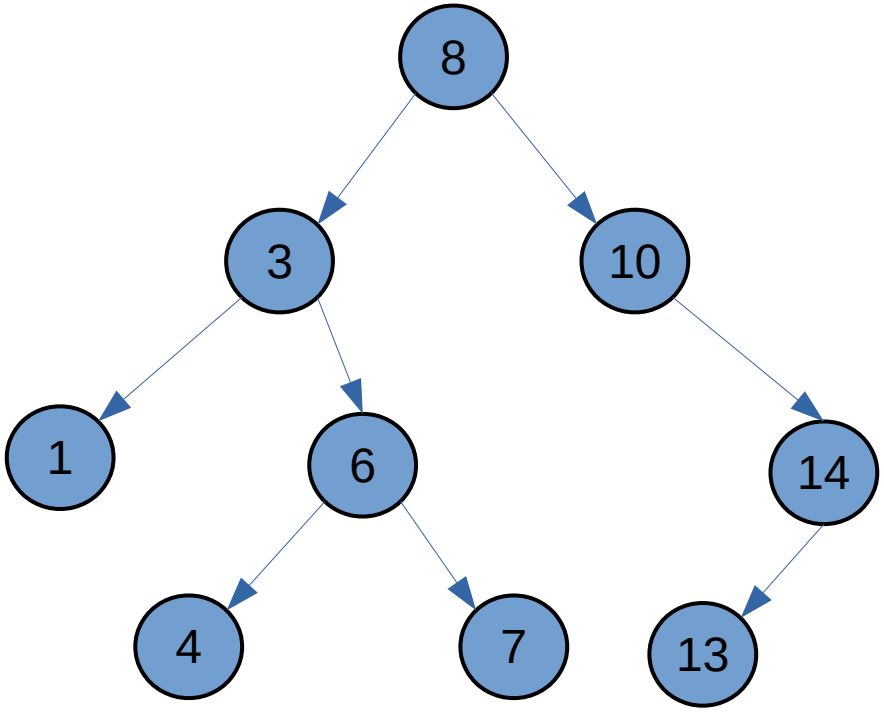
```
void levelorder(BSTree& myTree)
{
    BSTree::Node* root = myTree.getRoot();

    if(!root)
        return;

    std::queue<BSTree::Node*> queue;

    queue.push(root);

    while(!queue.empty())
    {
        BSTree::Node* node = queue.front();
        std::cout << node->key << " ";
        queue.pop();

        if (node->left)
            queue.push(node->left);

        if (node->right)
            queue.push(node->right);
    }
}
```

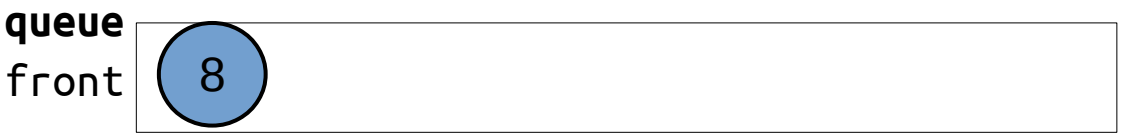


Tree Traversals – (Iterative) Level-order

Idea: visit level-by-level
This is also called Breadth-First Search (BFS)
Previous order traversals are instead Depth-First Search (DFS)

```
void levelorder(BSTree& myTree)
{
    BSTree::Node* root = myTree.getRoot();

    if(!root)
        return;

    std::queue<BSTree::Node*> queue;

    queue.push(root);

    while(!queue.empty())
    {
        BSTree::Node* node = queue.front();
        std::cout << node->key << " ";
        queue.pop();

        if (node->left)
            queue.push(node->left);

        if (node->right)
            queue.push(node->right);
    }
}
```



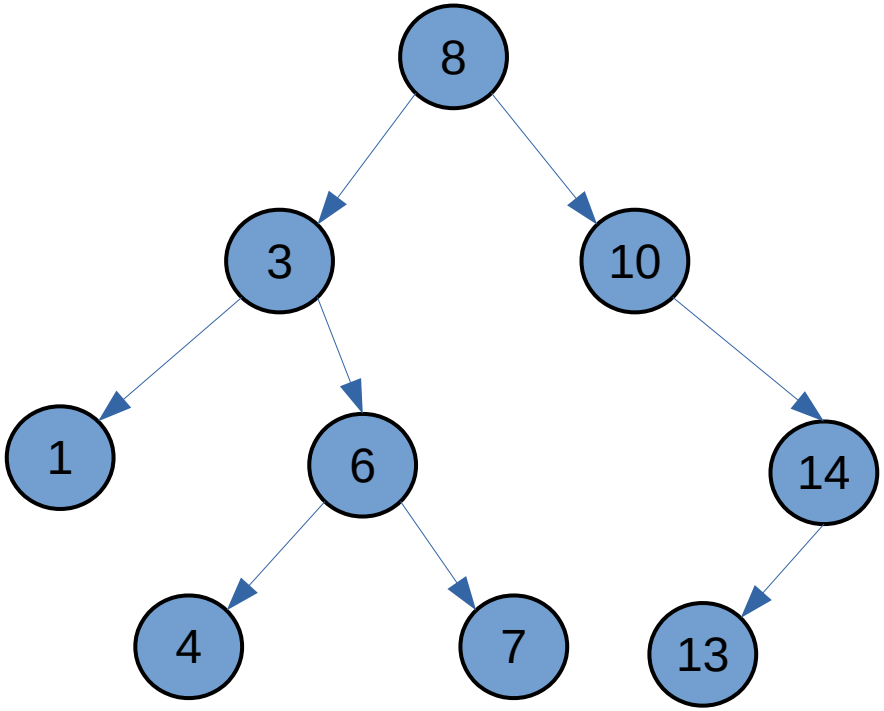


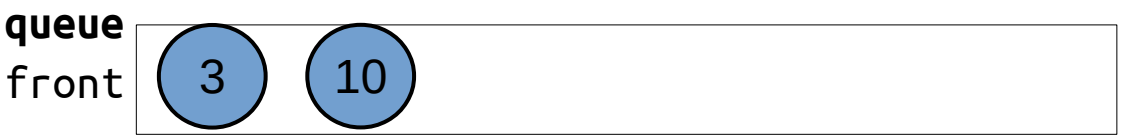

```
Output:

8
```

Tree Traversals – (Iterative) Level-order

Idea: visit level-by-level
This is also called Breadth-First Search (BFS)
Previous order traversals are instead Depth-First Search (DFS)

```
void levelorder(BSTree& myTree)
{
    BSTree::Node* root = myTree.getRoot();

    if(!root)
        return;

    std::queue<BSTree::Node*> queue;

    queue.push(root);

    while(!queue.empty())
    {
        BSTree::Node* node = queue.front();
        std::cout << node->key << " ";
        queue.pop();

        if (node->left)
            queue.push(node->left);

        if (node->right)
            queue.push(node->right);
    }
}
```



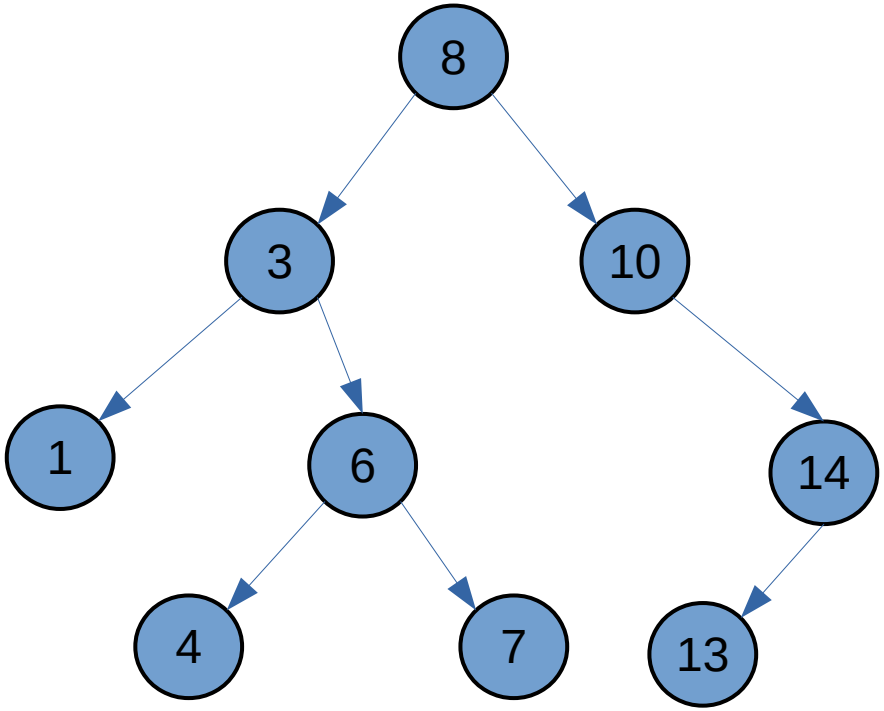


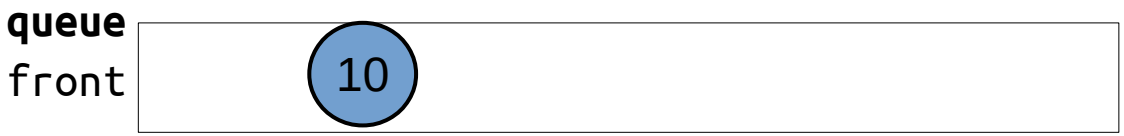

```
Output:

8
```

Tree Traversals – (Iterative) Level-order

Idea: visit level-by-level
This is also called Breadth-First Search (BFS)
Previous order traversals are instead Depth-First Search (DFS)

```
void levelorder(BSTree& myTree)
{
    BSTree::Node* root = myTree.getRoot();

    if(!root)
        return;

    std::queue<BSTree::Node*> queue;

    queue.push(root);

    while(!queue.empty())
    {
        BSTree::Node* node = queue.front();
        std::cout << node->key << " ";
        queue.pop();

        if (node->left)
            queue.push(node->left);

        if (node->right)
            queue.push(node->right);
    }
}
```



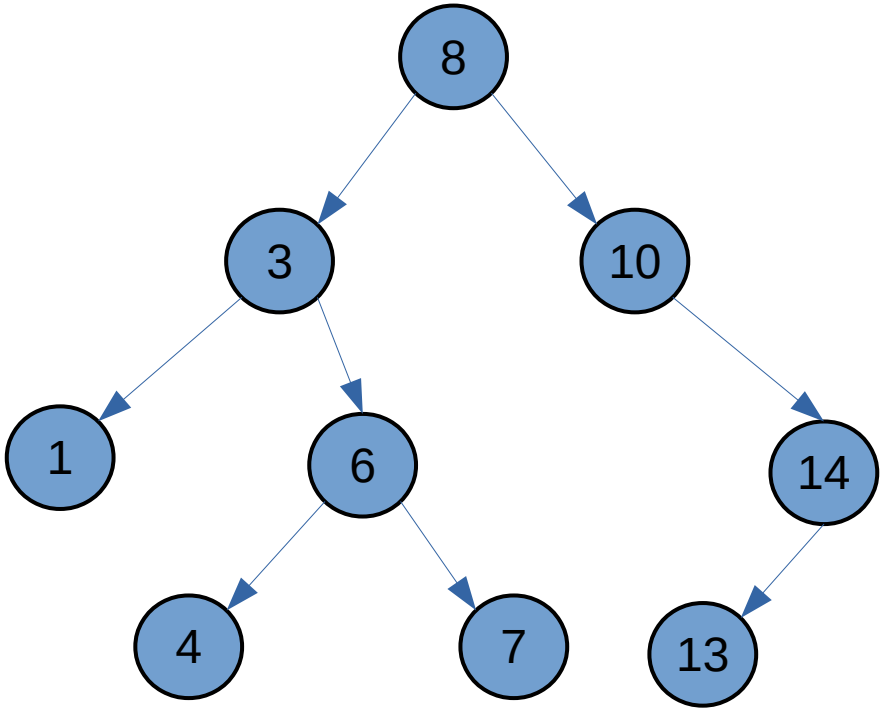


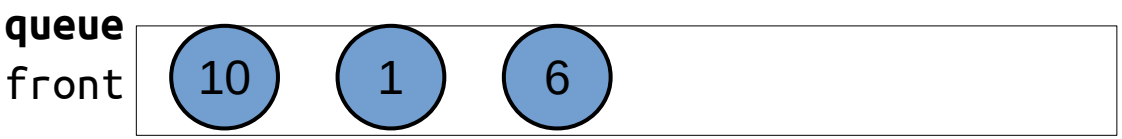

```
Output:

8       3
```

Tree Traversals – (Iterative) Level-order

Idea: visit level-by-level
This is also called Breadth-First Search (BFS)
Previous order traversals are instead Depth-First Search (DFS)

```
void levelorder(BSTree& myTree)
{
    BSTree::Node* root = myTree.getRoot();

    if(!root)
        return;

    std::queue<BSTree::Node*> queue;

    queue.push(root);

    while(!queue.empty())
    {
        BSTree::Node* node = queue.front();
        std::cout << node->key << " ";
        queue.pop();

        if (node->left)
            queue.push(node->left);

        if (node->right)
            queue.push(node->right);
    }
}
```



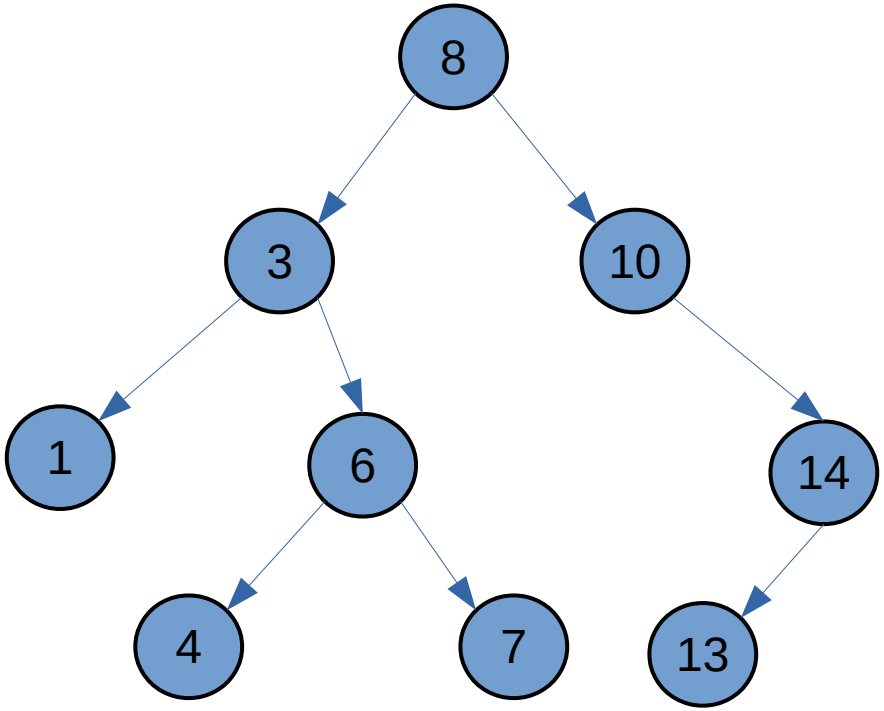


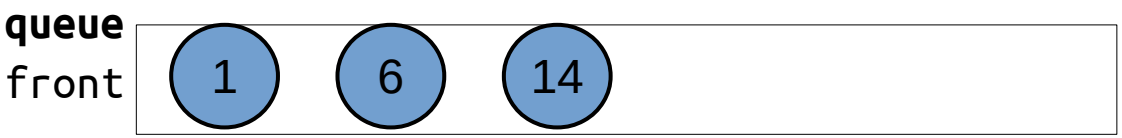

```
Output:

8       3
```

Tree Traversals – (Iterative) Level-order

Idea: visit level-by-level
This is also called Breadth-First Search (BFS)
Previous order traversals are instead Depth-First Search (DFS)

```
void levelorder(BSTree& myTree)
{
    BSTree::Node* root = myTree.getRoot();

    if(!root)
        return;

    std::queue<BSTree::Node*> queue;

    queue.push(root);

    while(!queue.empty())
    {
        BSTree::Node* node = queue.front();
        std::cout << node->key << " ";
        queue.pop();

        if (node->left)
            queue.push(node->left);

        if (node->right)
            queue.push(node->right);
    }
}
```






```
Output:

8       3       10
```