Trees

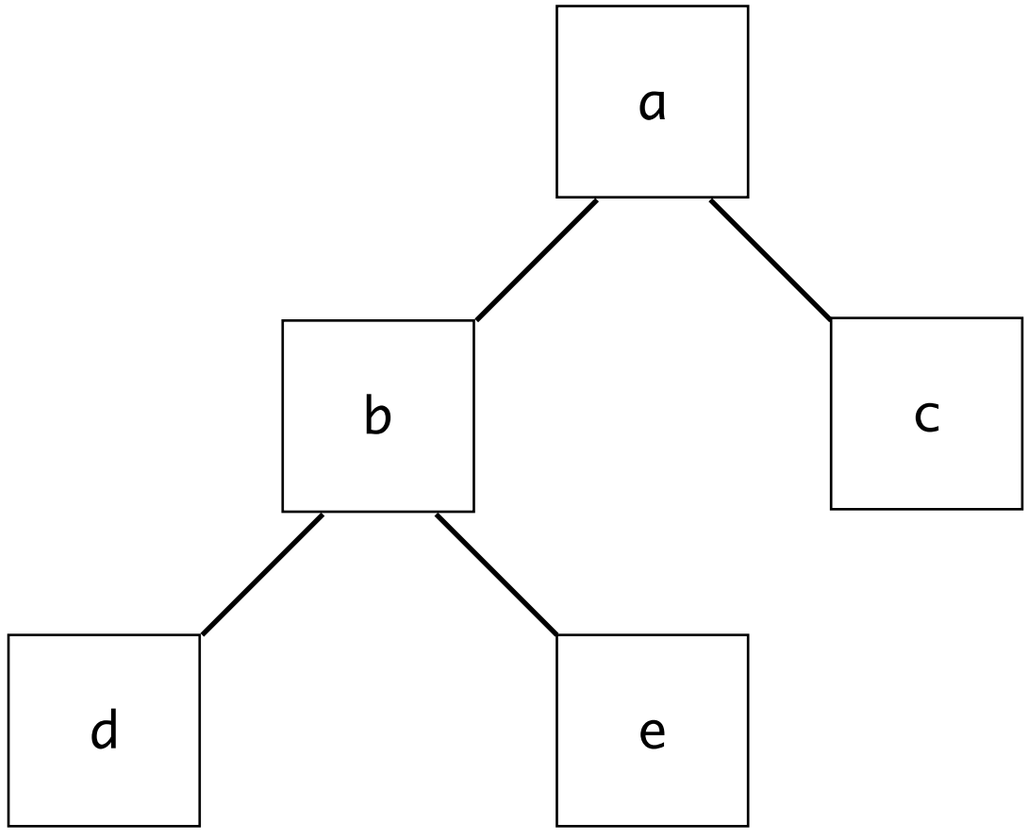
Tree Traversals

Tree Traversals

A tree traversal is a common task

- it's the term used for iterating over the data in a tree

For example, print out the values stored in this tree:



Tree Traversals

A tree traversal is a common task

- it's the term used for iterating over the data in a tree

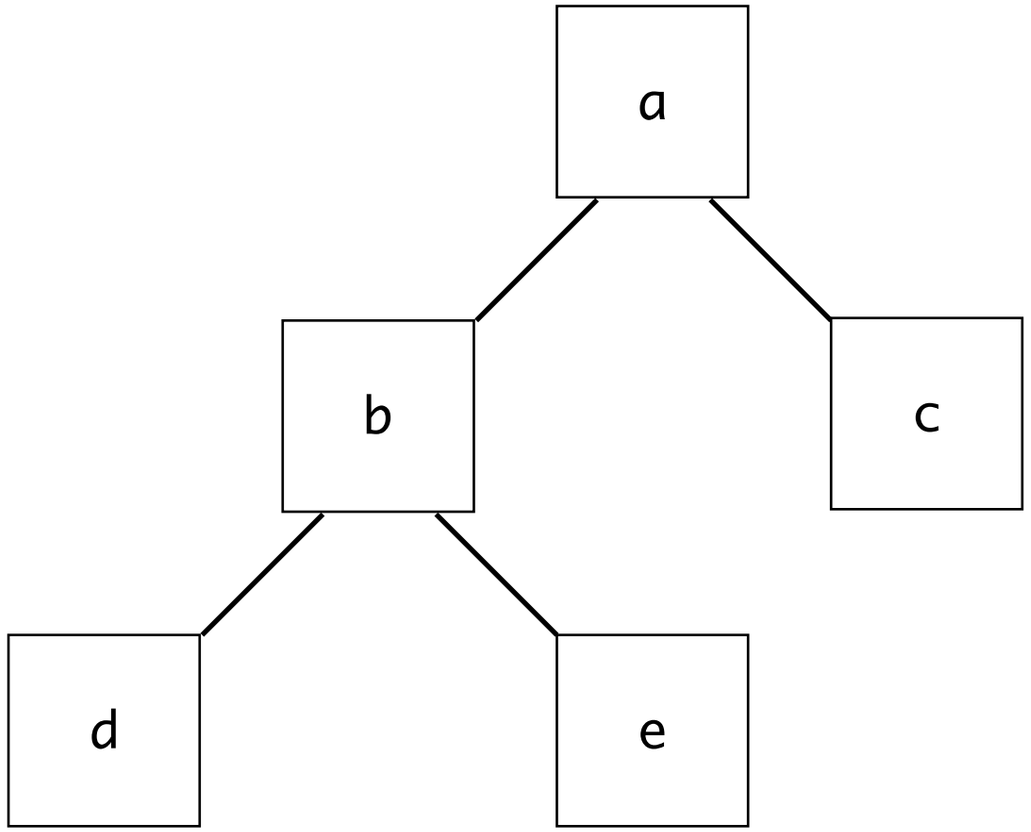
We can visit the nodes in three different orders:

- pre-order traversal
- in-order traversal
- post-order traversal

Pre-Order Traversal

Steps in a pre-order traversal:

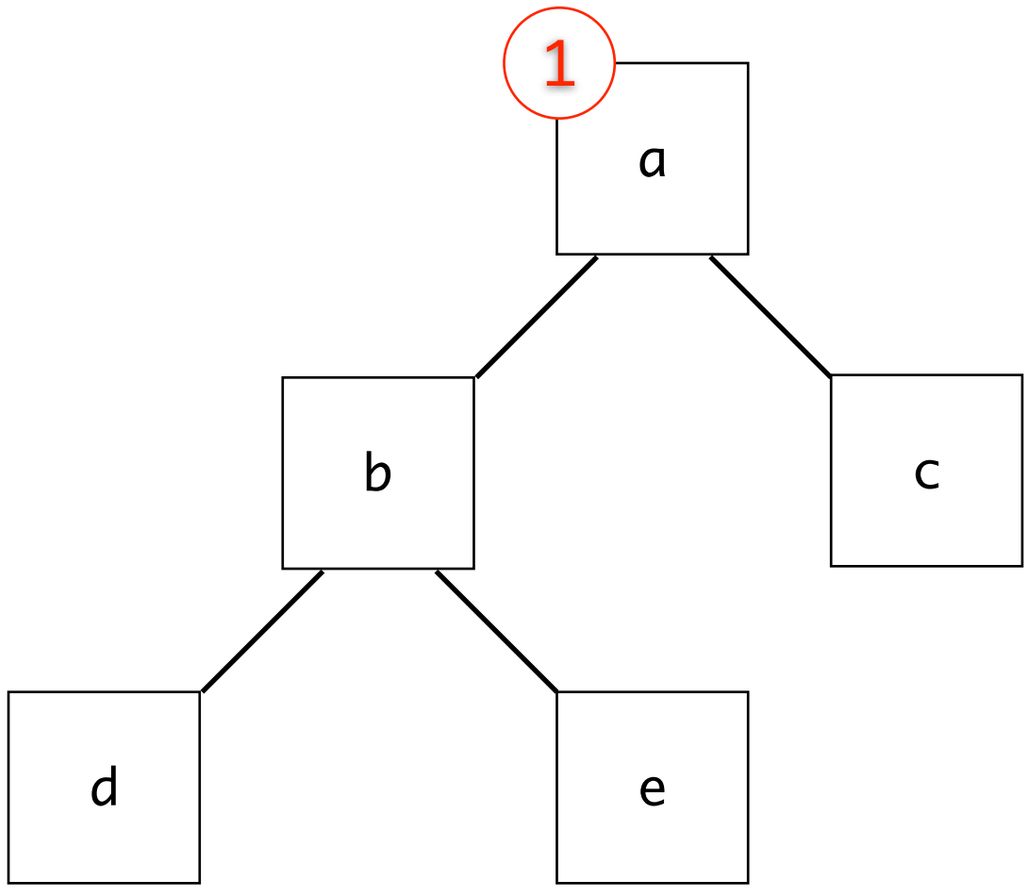
- process the root node itself
- process the left subtree of root node
- process the right subtree of root node



Pre-Order Traversal

Steps in a pre-order traversal:

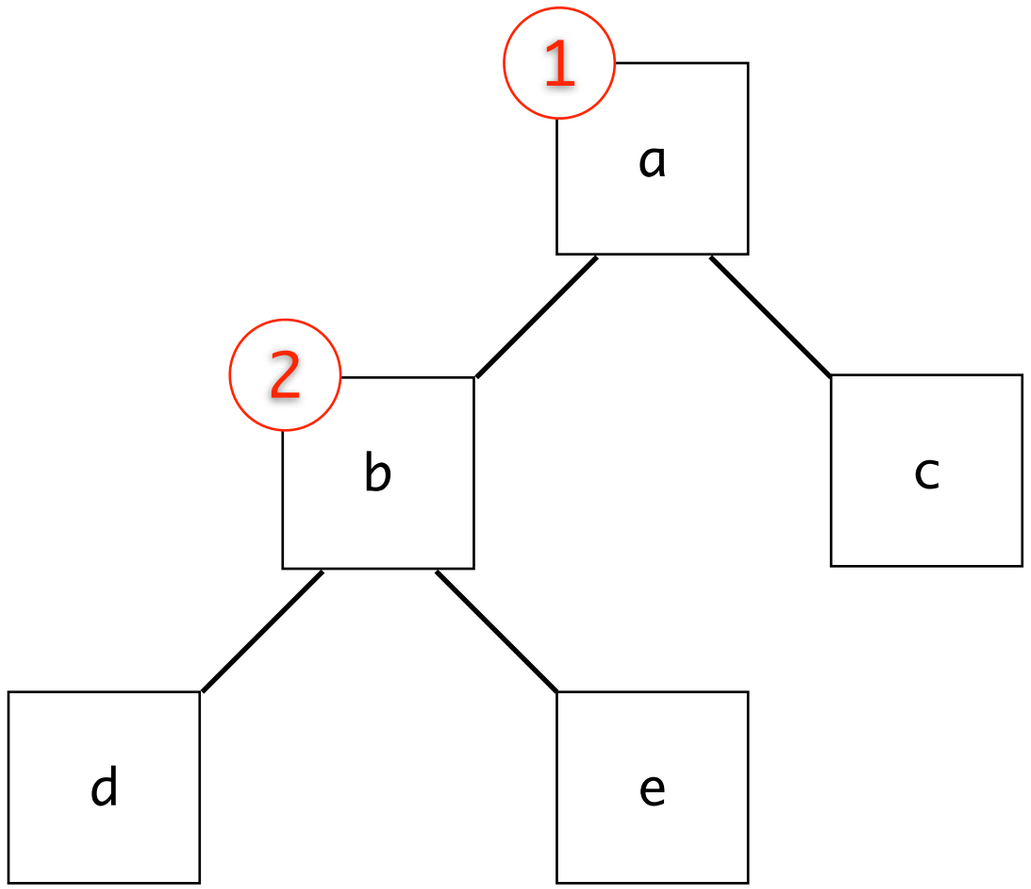
- process the root node itself
- process the left subtree of root node
- process the right subtree of root node



Pre-Order Traversal

Steps in a pre-order traversal:

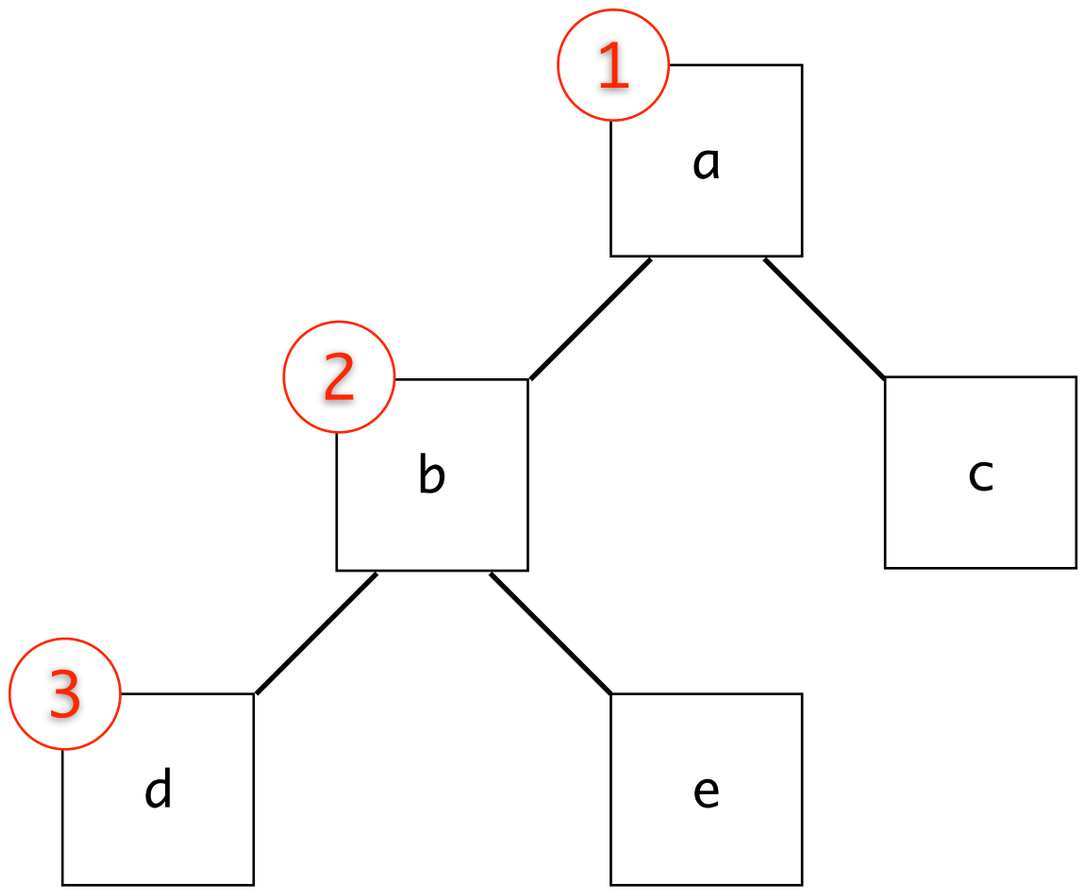
- process the root node itself
- process the left subtree of root node
- process the right subtree of root node



Pre-Order Traversal

Steps in a pre-order traversal:

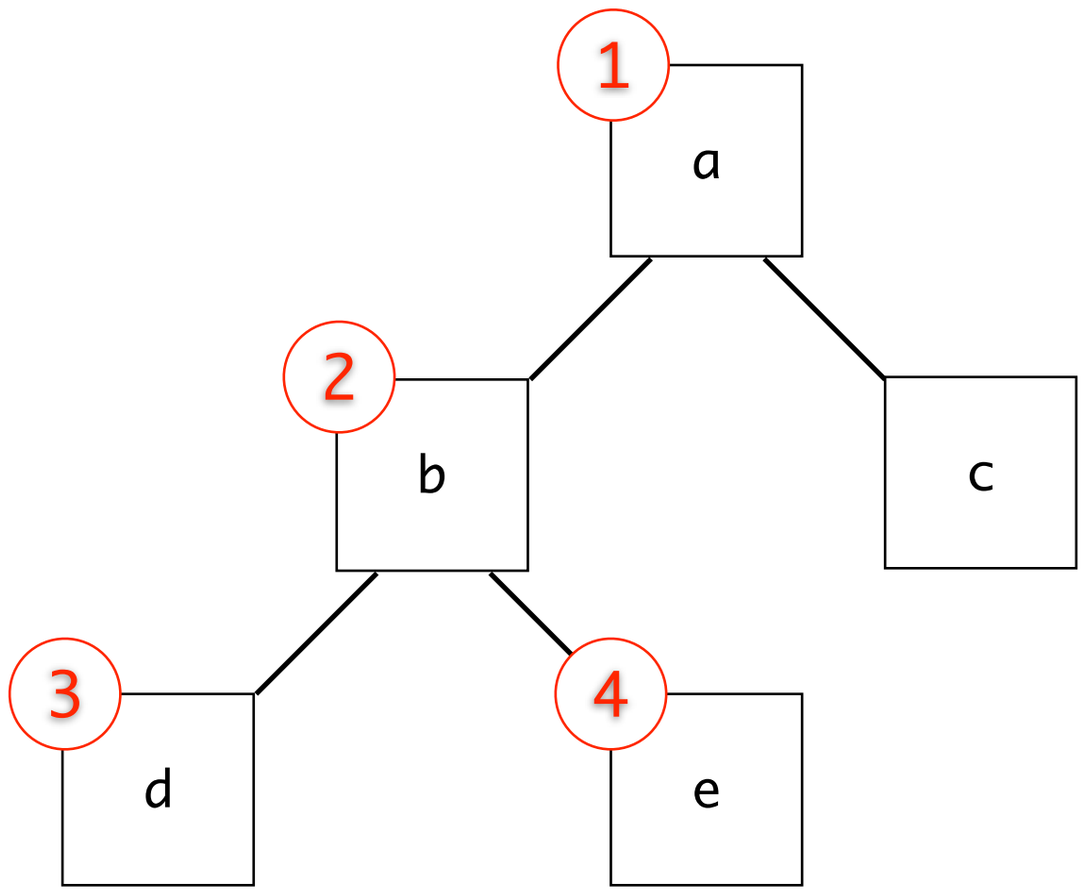
- process the root node itself
- process the left subtree of root node
- process the right subtree of root node



Pre-Order Traversal

Steps in a pre-order traversal:

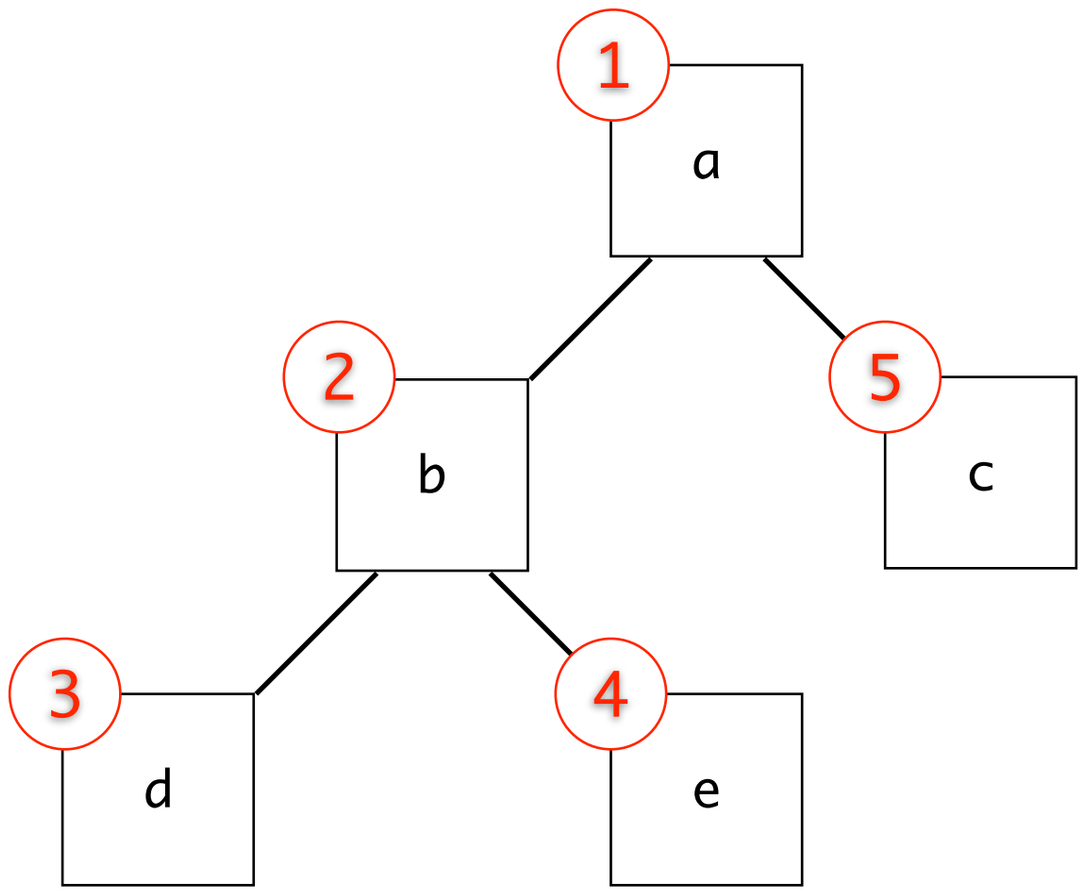
- process the root node itself
- process the left subtree of root node
- process the right subtree of root node



Pre-Order Traversal

Steps in a pre-order traversal:

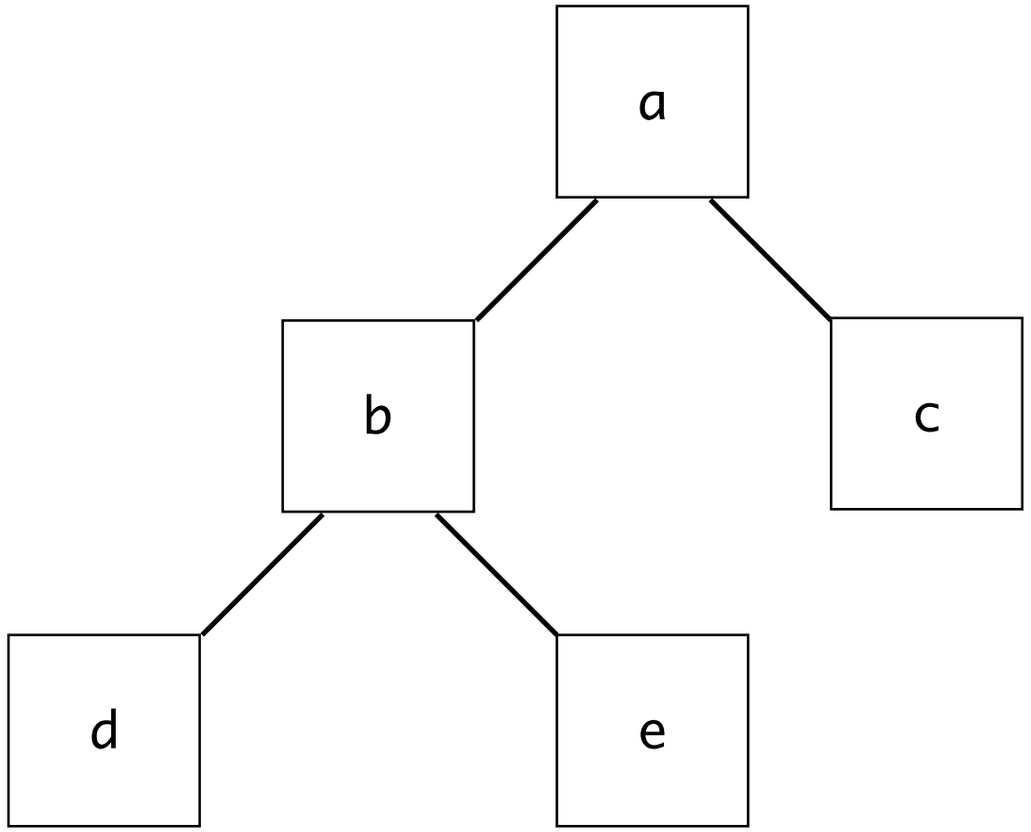
- process the root node itself
- process the left subtree of root node
- process the right subtree of root node



In-Order Traversal

Steps in an in-order traversal:

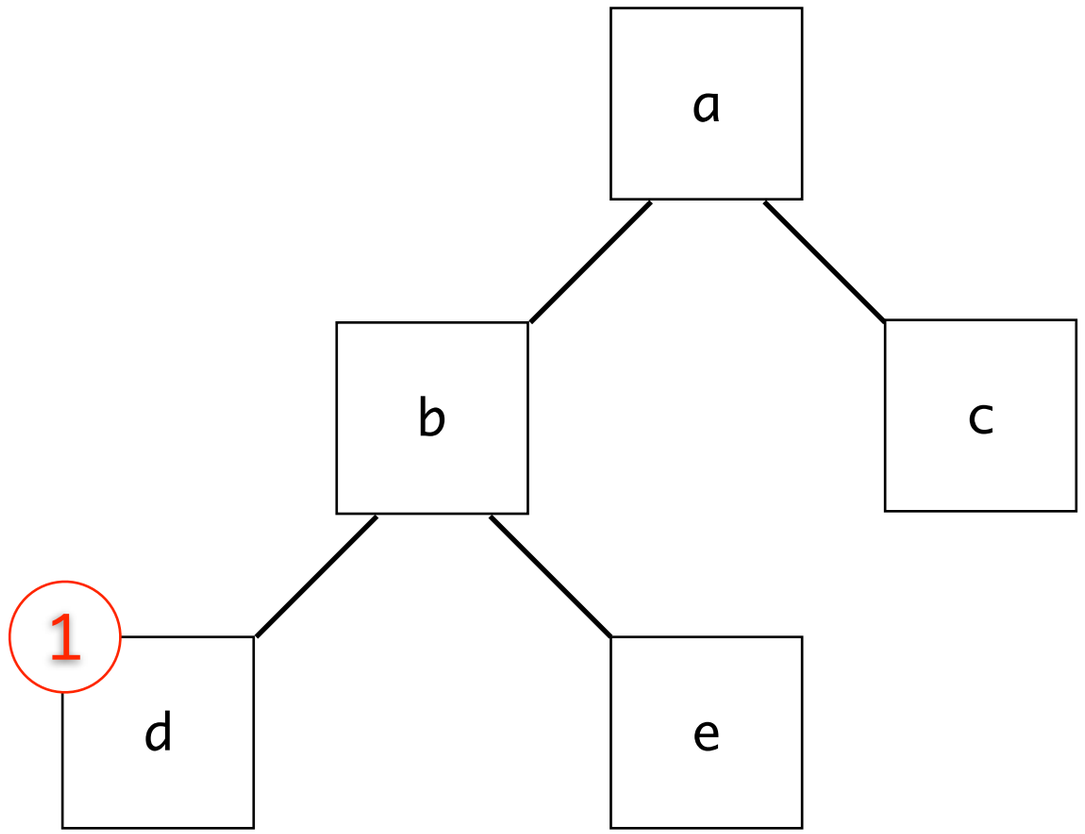
- process the left subtree of root node
- process the root node itself
- process the right subtree of root node



In-Order Traversal

Steps in an in-order traversal:

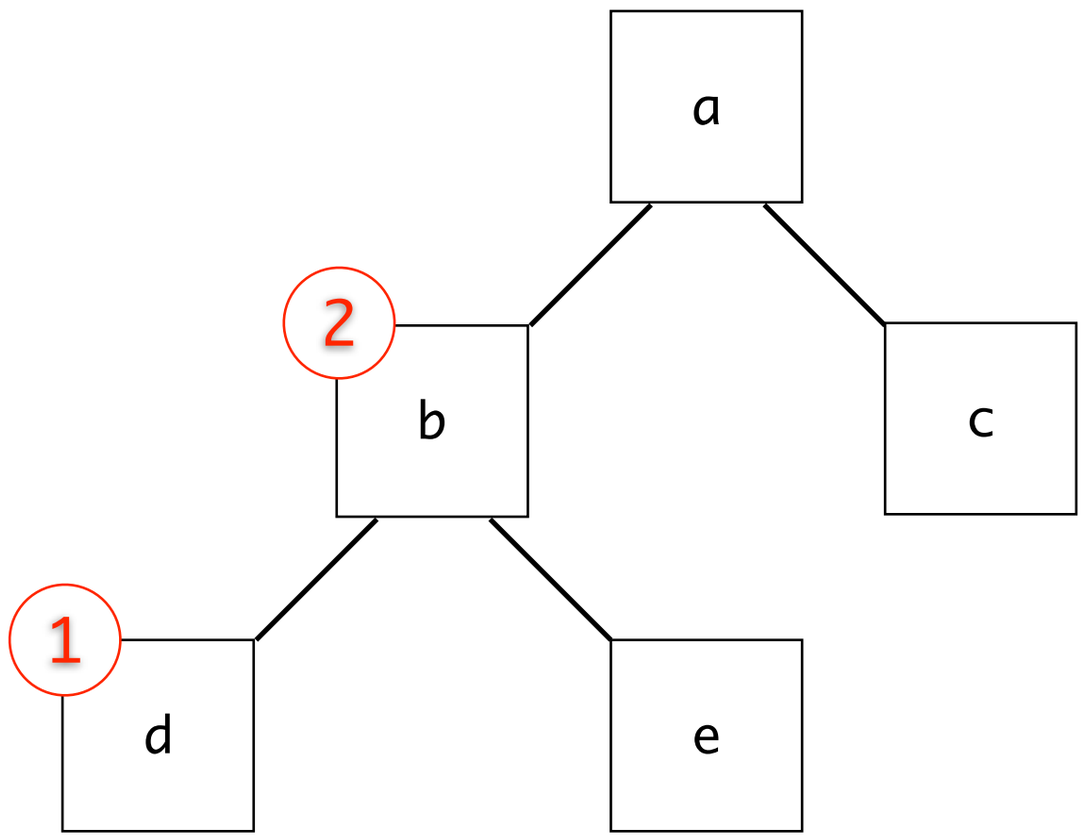
- process the left subtree of root node
- process the root node itself
- process the right subtree of root node



In-Order Traversal

Steps in an in-order traversal:

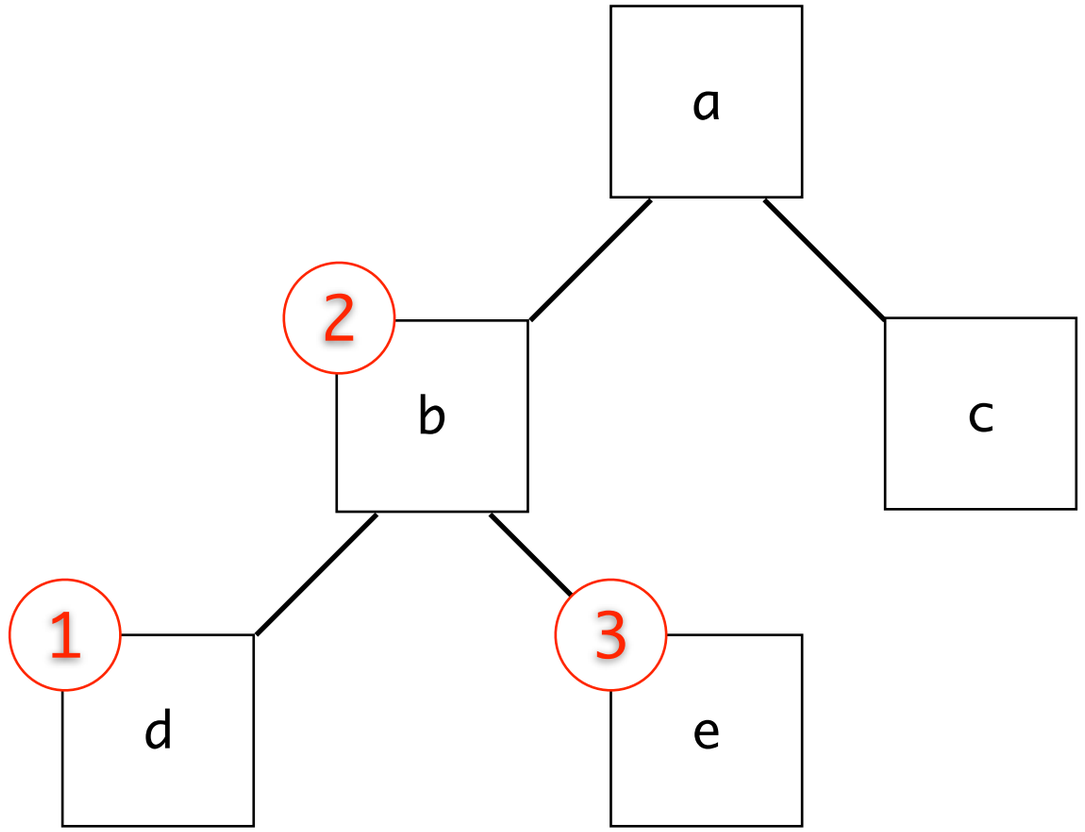
- process the left subtree of root node
- process the root node itself
- process the right subtree of root node



In-Order Traversal

Steps in an in-order traversal:

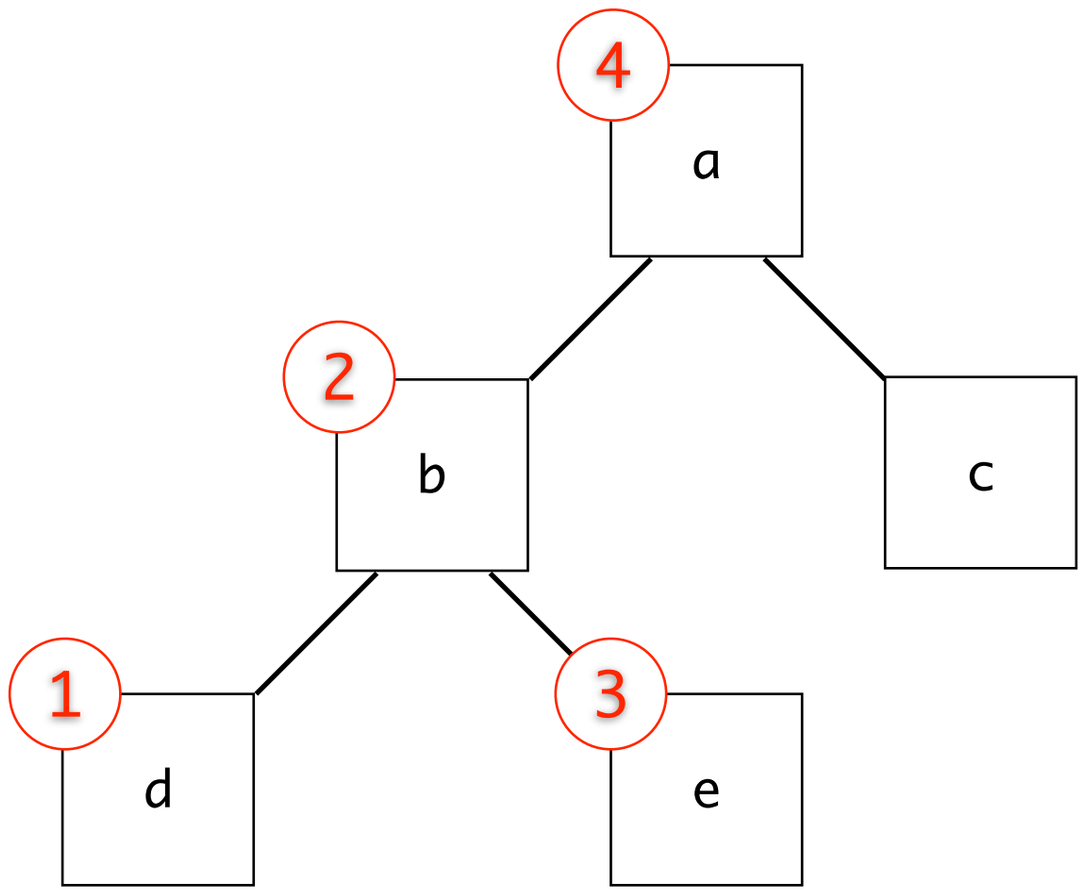
- process the left subtree of root node
- process the root node itself
- process the right subtree of root node



In-Order Traversal

Steps in an in-order traversal:

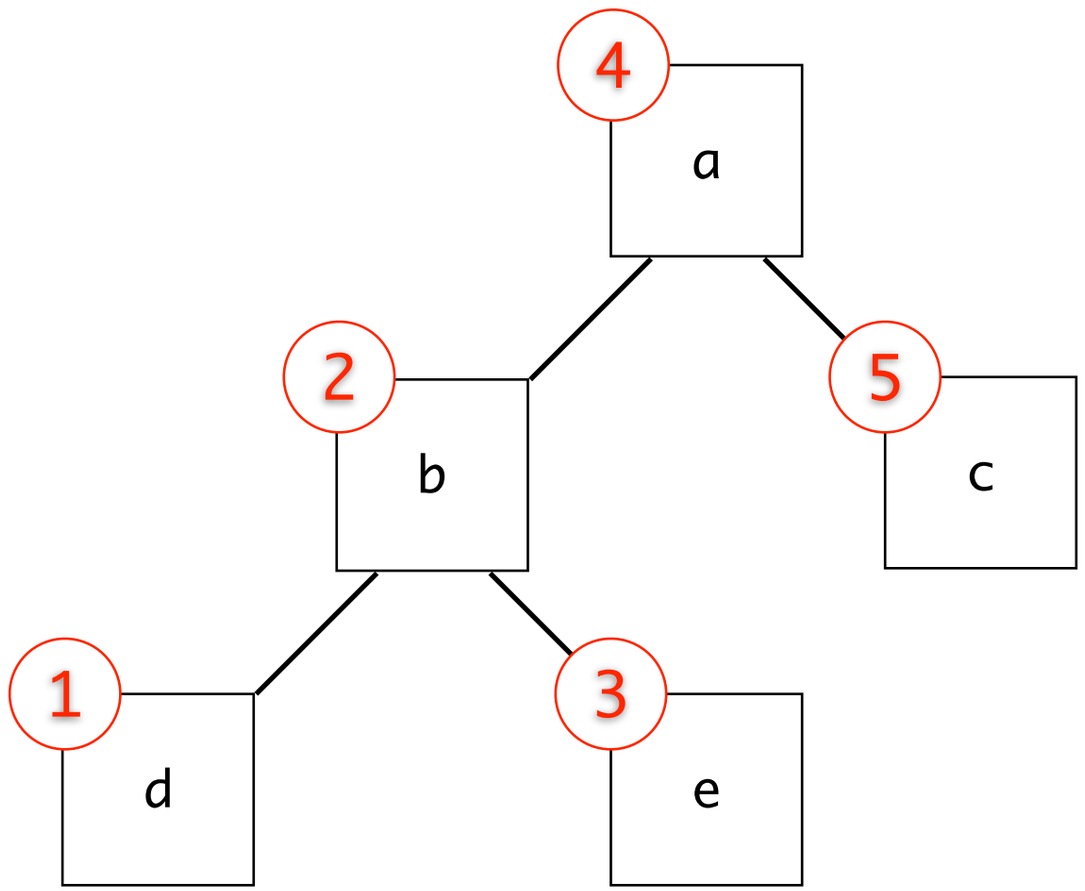
- process the left subtree of root node
- process the root node itself
- process the right subtree of root node



In-Order Traversal

Steps in an in-order traversal:

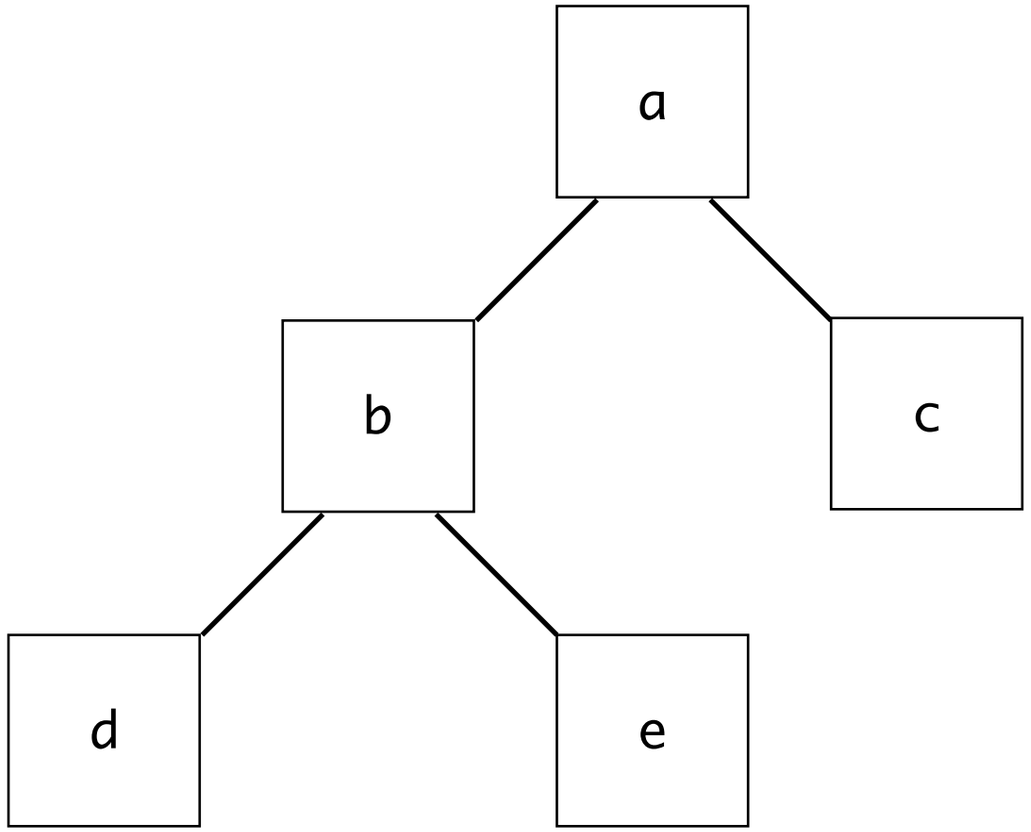
- process the left subtree of root node
- process the root node itself
- process the right subtree of root node



Post-Order Traversal

Steps in a post-order traversal:

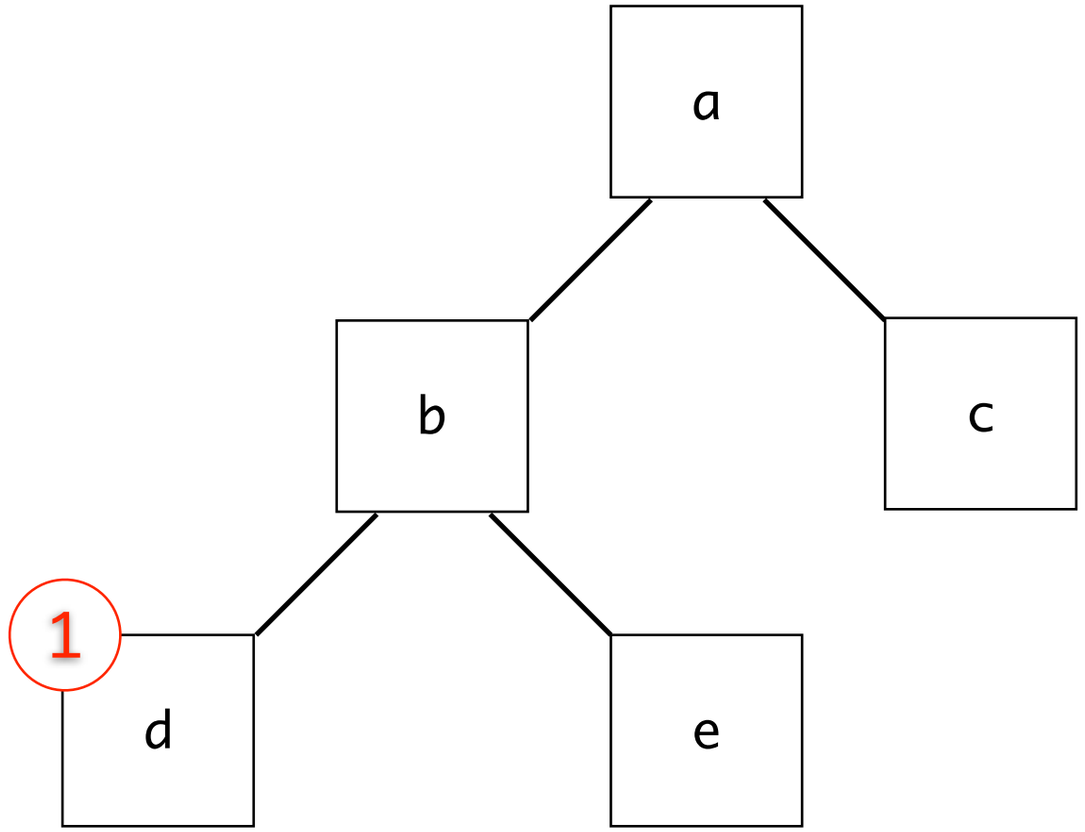
- process the left subtree of root node
- process the right subtree of root node
- process the root node itself



Post-Order Traversal

Steps in a post-order traversal:

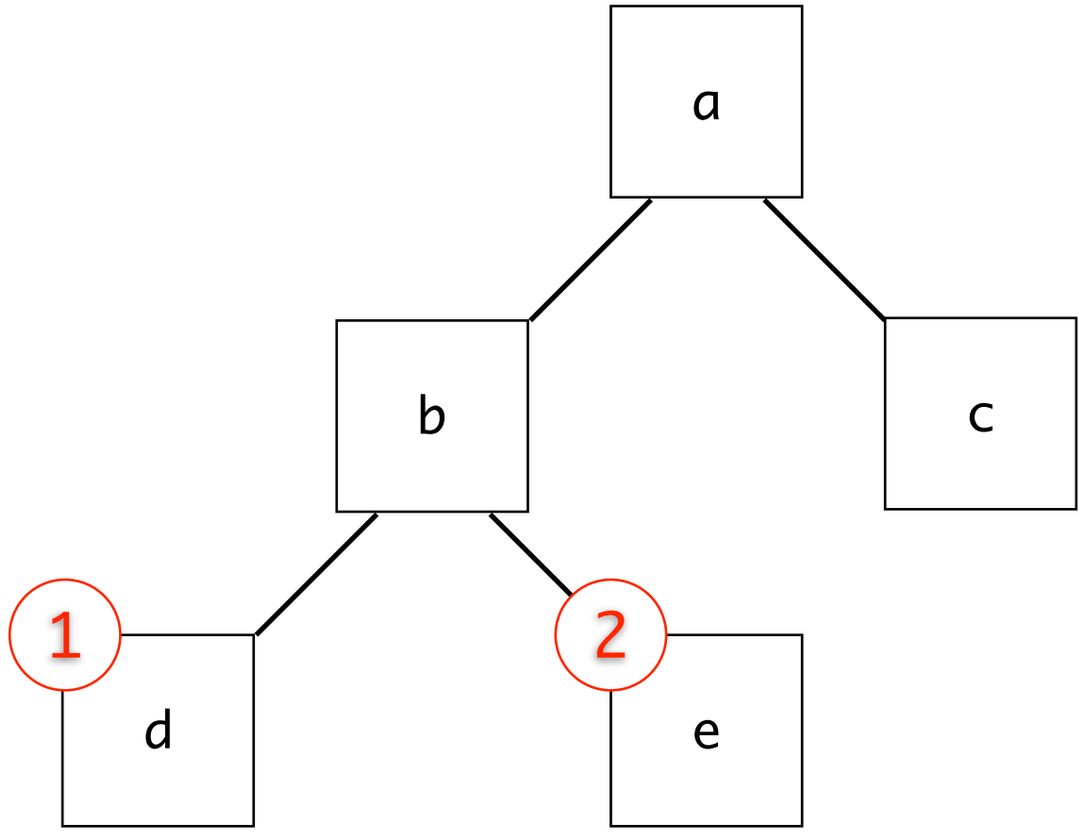
- process the left subtree of root node
- process the right subtree of root node
- process the root node itself



Post-Order Traversal

Steps in a post-order traversal:

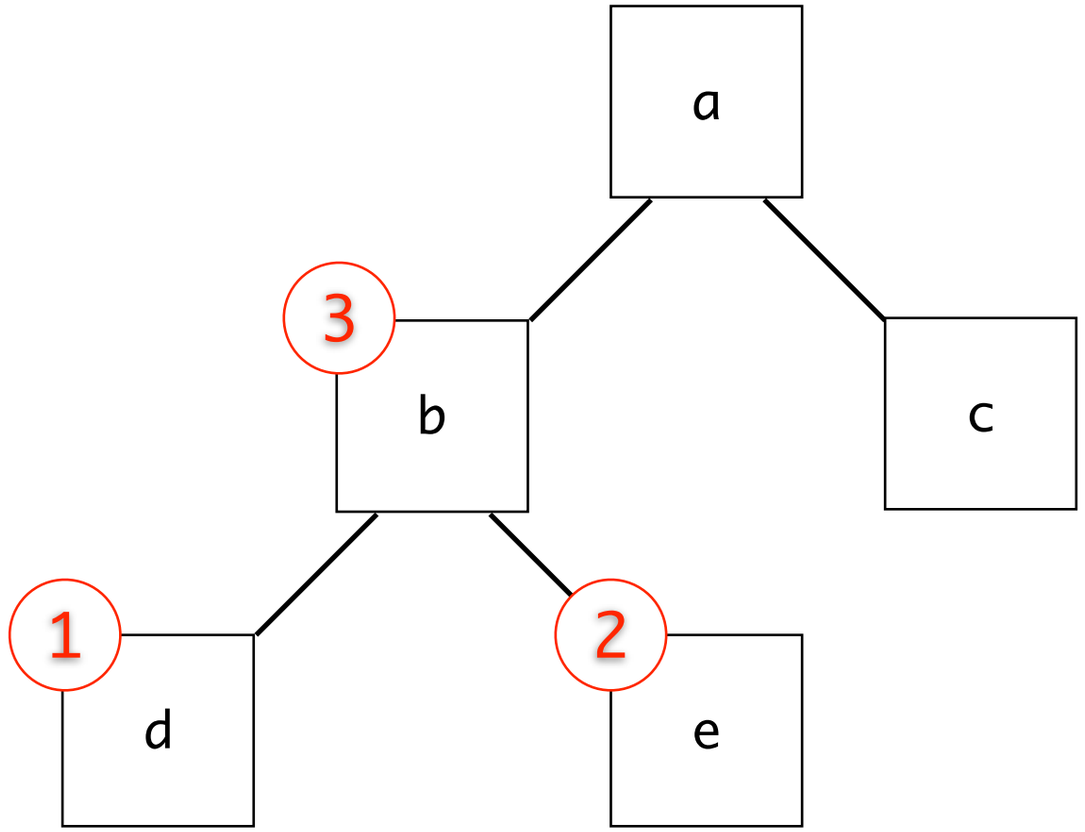
- process the left subtree of root node
- process the right subtree of root node
- process the root node itself



Post-Order Traversal

Steps in a post-order traversal:

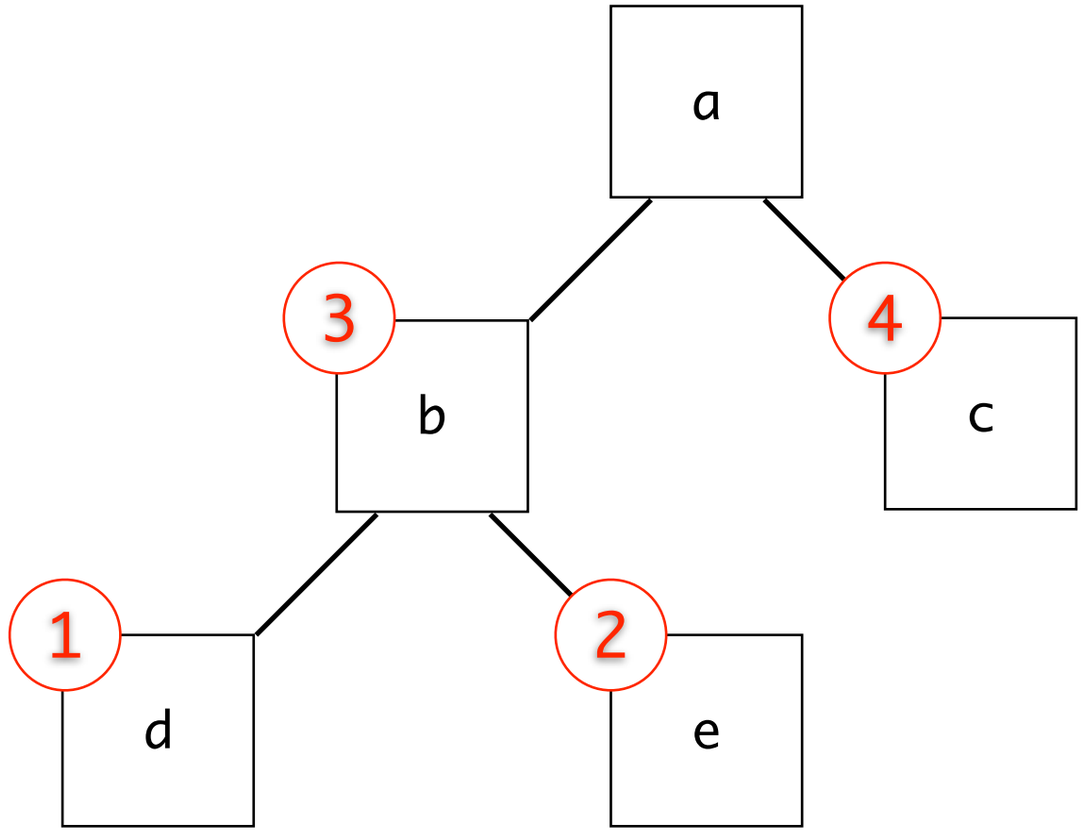
- process the left subtree of root node
- process the right subtree of root node
- process the root node itself



Post-Order Traversal

Steps in a post-order traversal:

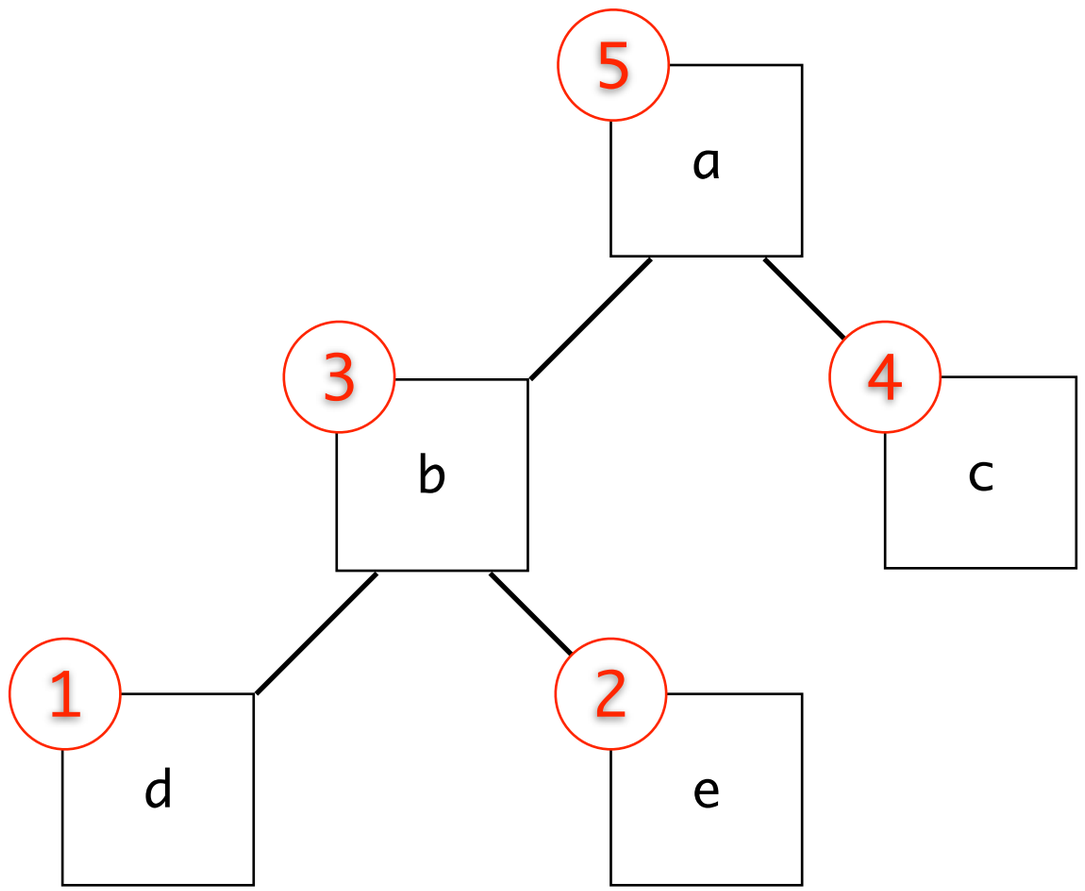
- process the left subtree of root node
- process the right subtree of root node
- process the root node itself



Post-Order Traversal

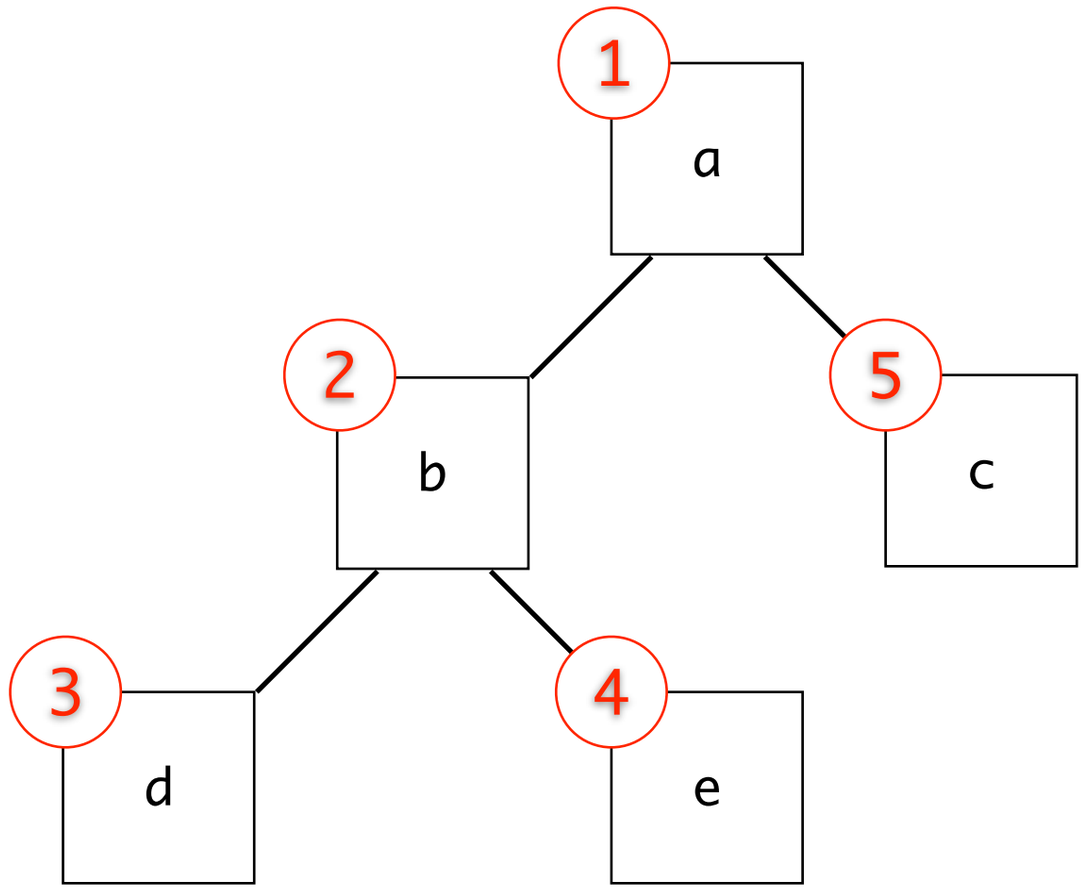
Steps in a post-order traversal:

- process the left subtree of root node
- process the right subtree of root node
- process the root node itself



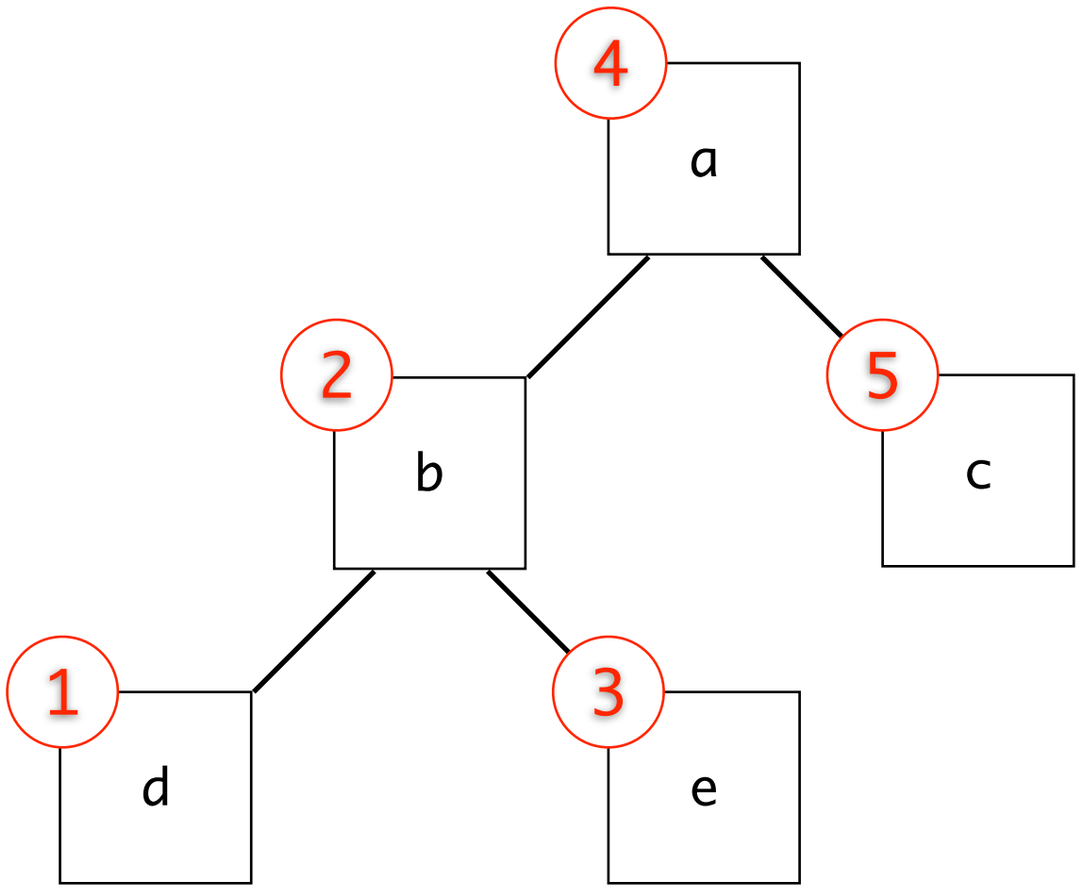
Pre-Order Traversal

```
- process root (a)
- process left tree
   - process root (b)
   - process left tree
      - process root (d)
      - process left tree  - NULL
      - process right tree - NULL
   - process right tree
      - process root (e)
      - process left tree  - NULL
      - process right tree - NULL
- process right tree
   - process root (c)
   - process left tree  - NULL
   - process right tree - NULL
```



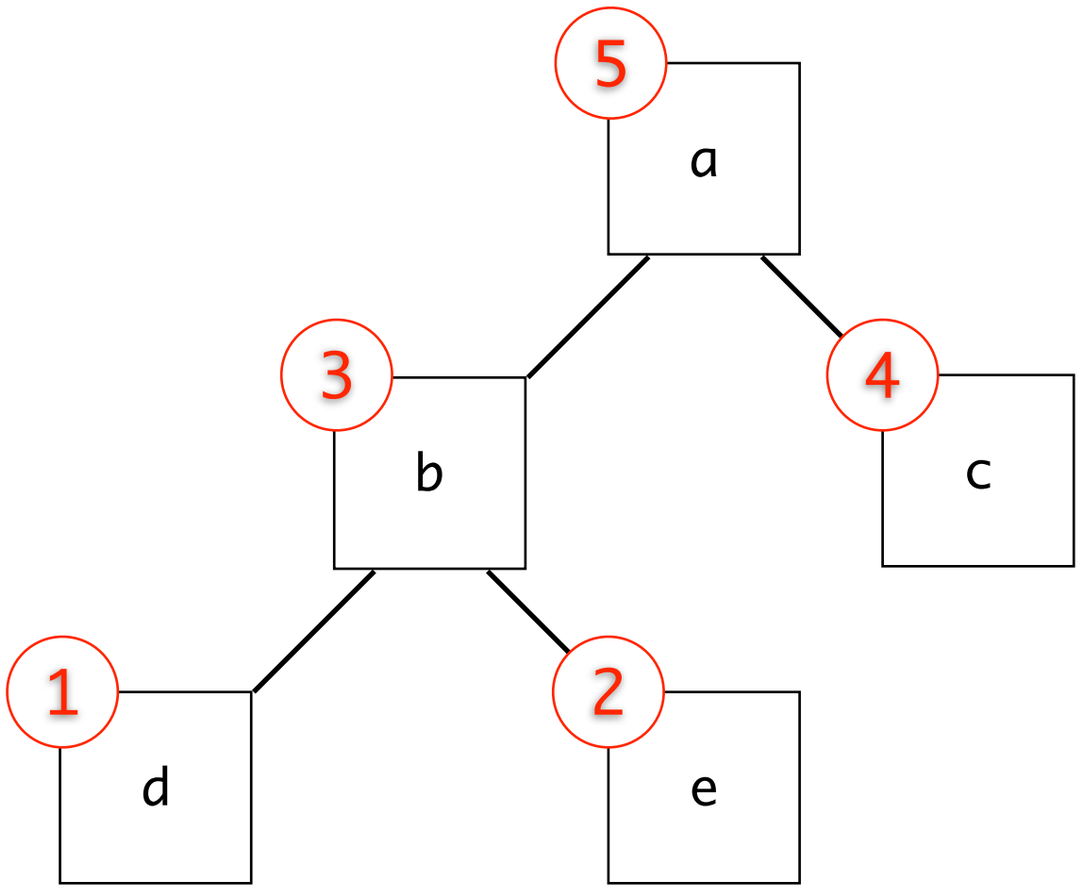
In-Order Traversal

```
- process left tree
   - process left tree
      - process left tree  - NULL
      - process root (d)
      - process right tree - NULL
   - process root (b)
   - process right tree
      - process left tree  - NULL
      - process root (e)
      - process right tree - NULL
- process root (a)
- process right tree
   - process left tree  - NULL
   - process root (c)
   - process right tree - NULL
```



Post-Order Traversal

```
- process left tree
   - process left tree
      - process left tree  - NULL
      - process right tree - NULL
      - process root (d)
   - process right tree
      - process left tree  - NULL
      - process right tree - NULL
      - process root (e)
   - process root (b)
- process right tree
   - process left tree  - NULL
   - process right tree - NULL
   - process root (c)
- process root (a)
```



Traversal Implementations

TreeNode

Assume that our tree is a binary tree as before...

It consists of zero or more of the following nodes:

```
template <typename Item>
struct TreeNode {
    TreeNode* left_child;
    TreeNode* right_child;
    Item data;
};
```

Recursive Traversals

Tree traversals are excellent candidates for recursive solutions

- consider a pre-order traversal of a tree that prints each data value
- what would the base case be?
- what are the recursive steps? (hint: plural)

Remember the steps in a pre-order traversal:

- process the root node itself
- process the left subtree of root node
- process the right subtree of root node

Recursive Traversals

Finish writing the following recursive function:

```
template <typename Item>
void preorder_print(const TreeNode<Item>* node_ptr) {



}
```

Recursive Traversals

Finish writing the following recursive function:

```
template <typename Item>
void preorder_print(const TreeNode<Item>* root_ptr) {
    if (root_ptr == NULL) return;
    cout << root_ptr->data << endl;
    preorder_print(root_ptr->left_child);
    preorder_print(root_ptr->right_child);
}
```

How would you modify this code for an in- or post-order traversal?

Recursive Traversals

For an in-order traversal:

```
template <typename Item>
void inorder_print(const TreeNode<Item>* root_ptr) {
    if (root_ptr == NULL) return;
    inorder_print(root_ptr->left_child);
    cout << root_ptr->data << endl;
    inorder_print(root_ptr->right_child);
}
```

Simply a different order for the recursive calls and output

- other than name changes, that is...

Recursive Traversals

For a post-order traversal:

```
template <typename Item>
void postorder_print(const TreeNode<Item>* root_ptr) {
    if (root_ptr == NULL) return;
    postorder_print(root_ptr->left_child);
    postorder_print(root_ptr->right_child);
    cout << root_ptr->data << endl;
}
```

Simply a different order for the recursive calls and output

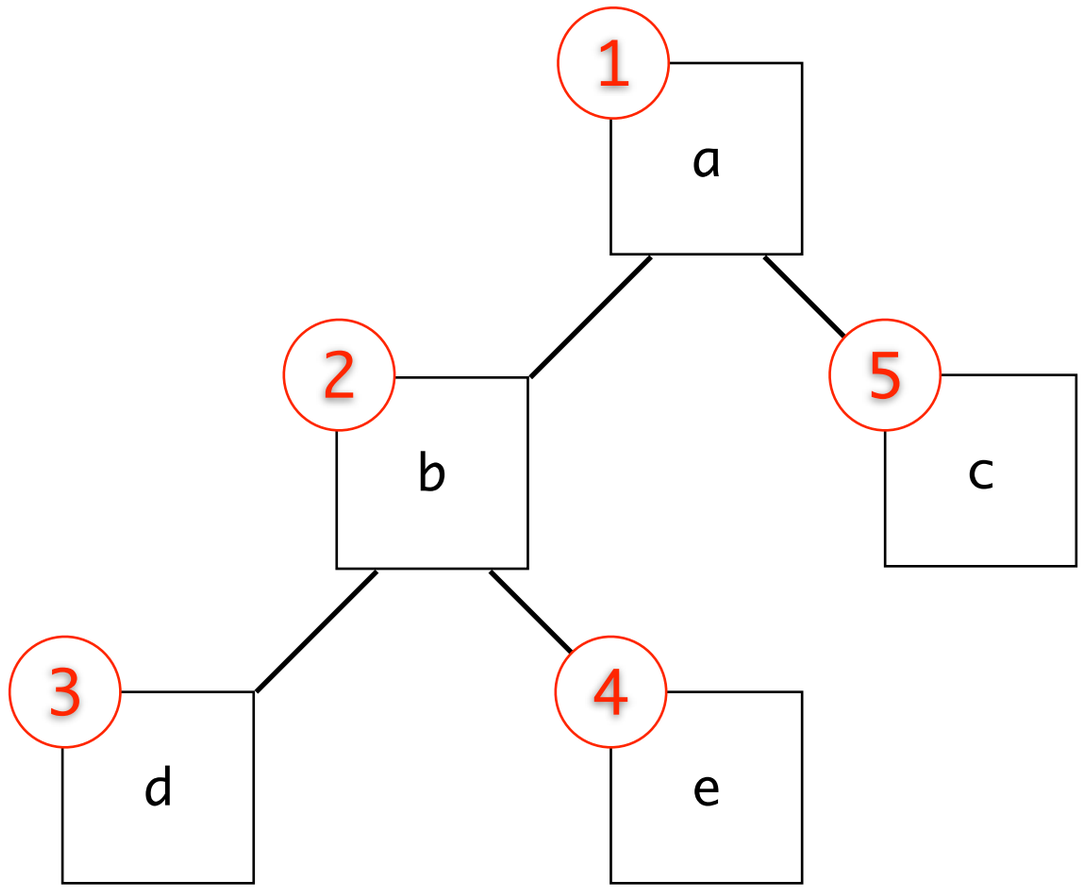
- other than name changes, that is...

Make sure you understand why
recursion works so well for traversals!

Recursive Traversals

Pre-order:

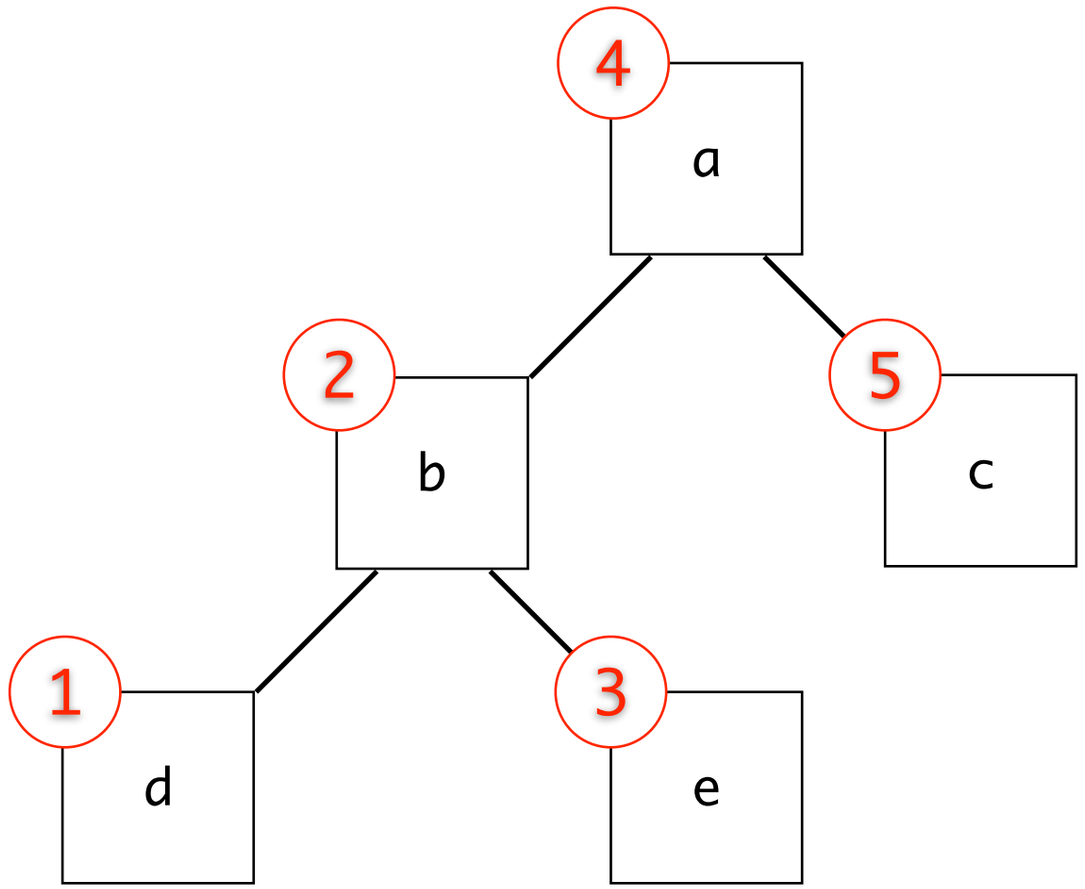
- process the root node itself
- process the left subtree of root node
- process the right subtree of root node



Recursive Traversals

In-order:

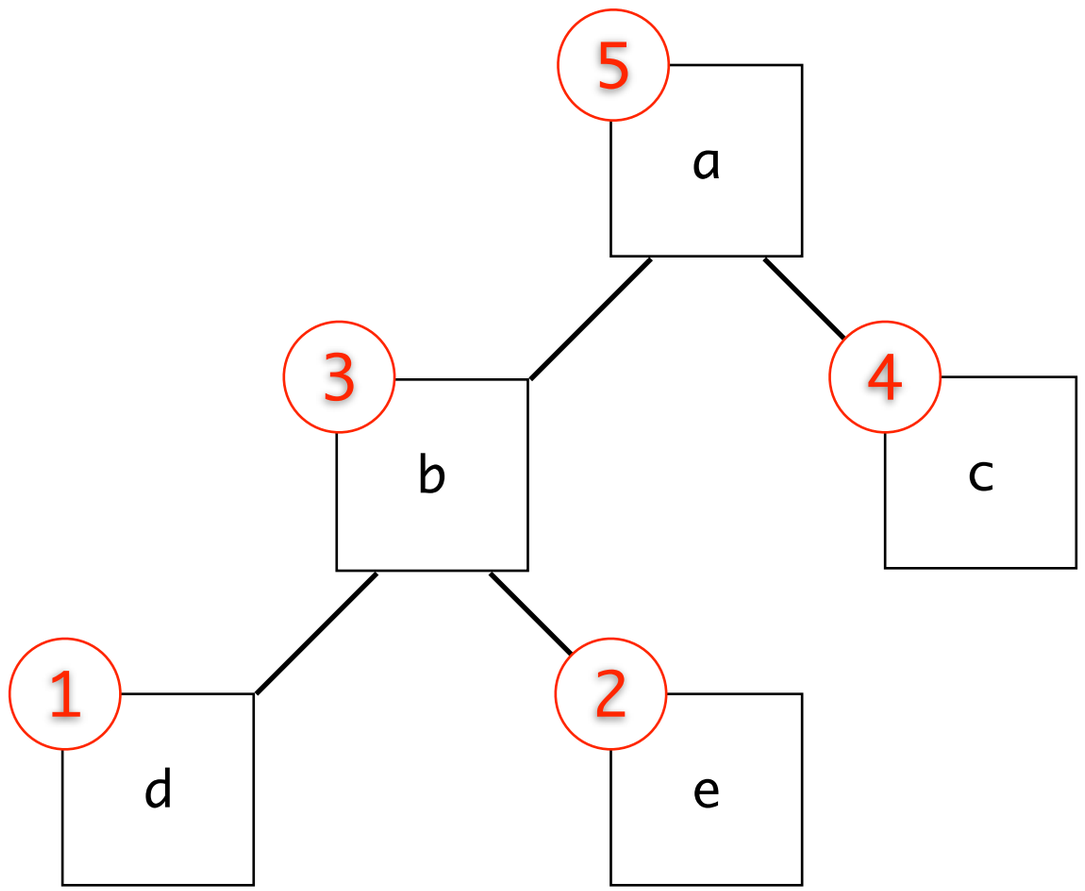
- process the left subtree of root node
- process the root node itself
- process the right subtree of root node



Recursive Traversals

Post-order:

- process the left subtree of root node
- process the right subtree of root node
- process the root node itself



Functions as Arguments

Functions as Arguments

What if we wanted to do an in-order traversal with a generic action?

- you probably wouldn't want to rewrite the traversal logic multiple times for slightly different tasks (printing each data value, finding the max, etc...)

Functions can accept functions as arguments!

- this lets them accept different actions to apply to different situations
- fun stuff =)

Functions as Arguments

Consider the following function prototype:

```
void apply(void f(int&), int array[], size_t size);
```

Its first argument is a function!

```
// argument f, a void function with one int& argument
void f(int&)
```

The syntax:

- function arguments are declared by writing the name of the functions return type, followed by the name of the parameter
- note is that the name is followed by parentheses containing a list of arguments that the function accepts

Functions as Arguments

Example:

```
void apply(void f(int&), int array[], size_t size) {
    for (size_t i = 0; i < size; i++)
        f(array[i]); // call function on each element
}
```

What it does:

- this function accepts three arguments: a void function with one int& argument, an array of integers, and the size of the array
- its body simply loops over the array and calls the function (the first argument) on each element

Functions as Arguments

Calling the function:

```
void apply(void f(int&), int array[], size_t size);

void double_it(int& n) { n *= 2; }

void print_it(int& n) { cout << n << endl; }
```

```
int main() {
    int numbers[] = { 1, 2, 3, 4, 5 };

    // double each value in the array
    apply(double_it, numbers, 5);
}
```

Functions as Arguments

Calling the function:

```
void apply(void f(int&), int array[], size_t size);

void double_it(int& n) { n *= 2; }

void print_it(int& n) { cout << n << endl; }
```

```
int main() {
    int numbers[] = { 1, 2, 3, 4, 5 };

    // print each value in the array
    apply(print_it, numbers, 5);
}
```

Functions as Arguments

A templated version of the function:

```
template <typename Item, typename SizeType>
void apply(void f(Item&), Item array[], SizeType size) {
    for (SizeType i = 0; i < size; i++)
        f(array[i]); // call function on each element
}
```

This function:

- can be called on arrays of any type

Functions as Arguments

A templated version of the function:

```
template <typename Fn, typename Item, typename SizeType>
void apply(Fn f, Item array[], SizeType size) {
    for (SizeType i = 0; i < size; i++)
        f(array[i]); // call function on each element
}
```

This function:

- can be called on arrays of any type
- can be called with any function that accepts a single argument (by value or by reference)

Templated Traversal

A templated version of a function to perform a pre-order traversal:

```
template <typename Method, typename Item>
void preorder(Method f, TreeNode<Item>* root_ptr) {
    if (node_ptr == NULL) return;
    f(root_ptr->data); // apply the method
    preorder(f, root_ptr->left_child);
    preorder(f, root_ptr->right_child);
}
```