CS121 Data Structures A, C
Tree Traversal Algorithms

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Tree Traversal Algorithms

A **traversal** of a tree T is a systematic way of accessing, or 'visiting,' all the positions of T

The specific action associated with the 'visit' of a position p depends on the application of this traversal

This could involve anything from incrementing a counter to performing some complex computation for p

We describe several common traversal schemes for trees, implement them in the context of our various tree classes, and discuss a common application of tree traversals

Preorder Traversal

In a **preorder traversal** of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively

If the tree is ordered, then the subtrees are traversed according to the order of the children

Preorder traversal of the subtree rooted at a position p:

Algorithm `preorder`(p)
  perform the 'visit' action for position p ▷this happens before any recursion
  for each child c in `children`(p) **do**
    `preorder`(c) ▷recursively traverse the subtree rooted at c

Running time:

Preorder Traversal

In a **preorder traversal** of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively

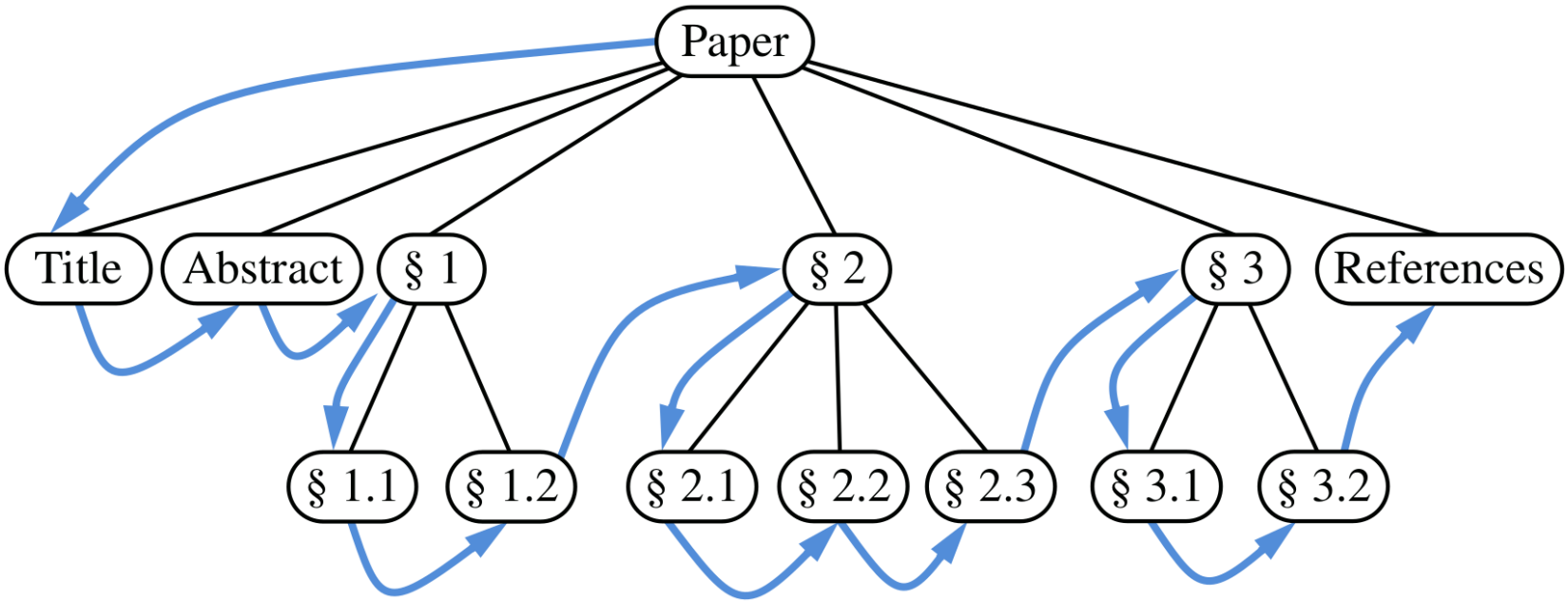
If the tree is ordered, then the subtrees are traversed according to the order of the children

Preorder traversal of the subtree rooted at a position p:

Algorithm `preorder`(p)
  perform the 'visit' action for position p ▷this happens before any recursion
  for each child c in `children`(p) **do**
    `preorder`(c) ▷recursively traverse the subtree rooted at c

Running time: $O(n)$, where n is the number of positions in the tree

Preorder Traversal Example



Postorder Traversal

A **postorder traversal** of a tree T recursively traverses the subtrees rooted at the children of the root first, and then visits the root

If the tree is ordered, then the subtrees are traversed according to the order of the children

Postorder traversal of the subtree rooted at a position p:

Algorithm `postorder`(p)
  for each child c in `children`(p) **do**
    `postorder`(c) ▷recursively traverse the subtree rooted at c
  perform the 'visit' action for position p ▷this happens after any recursion

Running time:

Postorder Traversal

A **postorder traversal** of a tree T recursively traverses the subtrees rooted at the children of the root first, and then visits the root

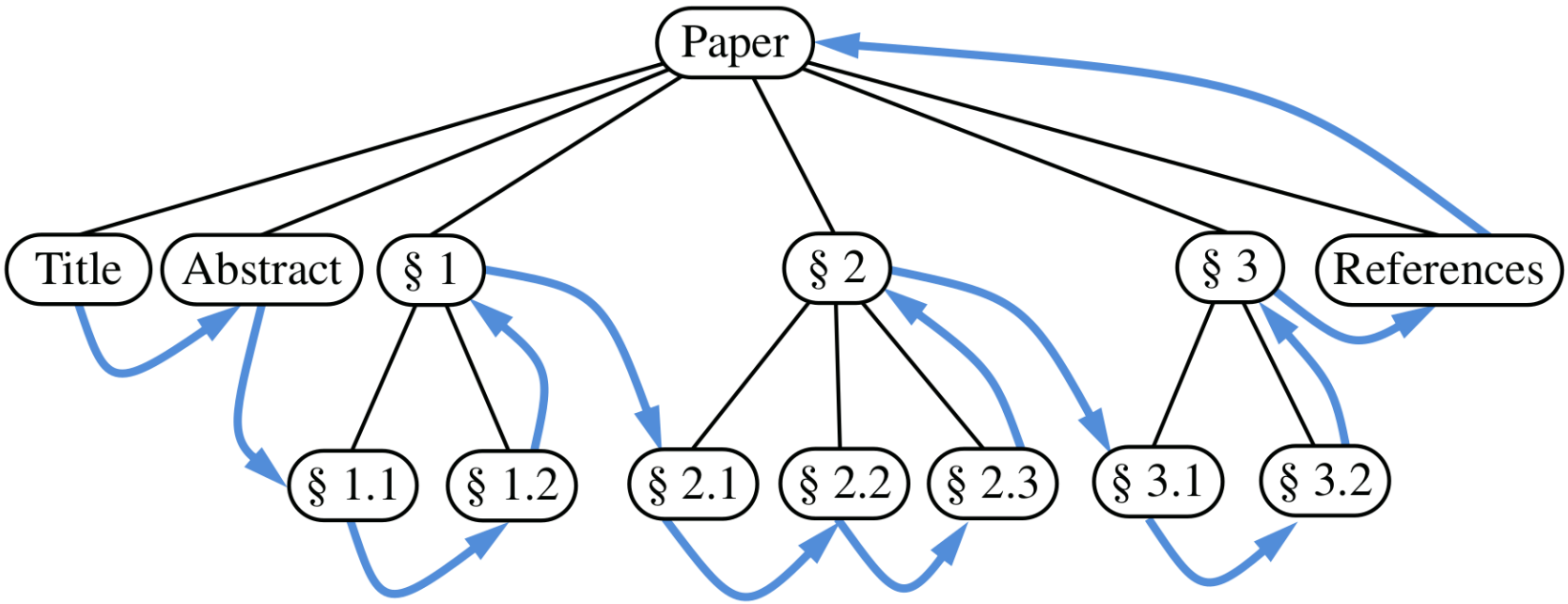
If the tree is ordered, then the subtrees are traversed according to the order of the children

Postorder traversal of the subtree rooted at a position p:

Algorithm `postorder`(p)
  for each child c in `children`(p) **do**
    `postorder`(c) ▷recursively traverse the subtree rooted at c
  perform the 'visit' action for position p ▷this happens after any recursion

Running time: $O(n)$, where n is the number of positions in the tree

Postorder Traversal Example



Breadth-First Tree Traversal

A **breadth-first traversal** visits all the positions at depth d before visiting the positions at depth $d + 1$

Breadth-first traversal of a tree, using a queue to produce a FIFO semantics for the visit order:

Algorithm `breadthfirst( )`
  Initialize queue Q to contain `root( )`
  while Q not empty **do**
    $p = Q$.`dequeue( )` ▷p is the oldest entry in the queue
    perform the 'visit' action for position p
    for each child c in `children`(p) **do**
      Q.`enqueue`(c) ▷add p's children to the end of the queue for later visits

Running time:

Breadth-First Tree Traversal

A **breadth-first traversal** visits all the positions at depth d before visiting the positions at depth $d + 1$

Breadth-first traversal of a tree, using a queue to produce a FIFO semantics for the visit order:

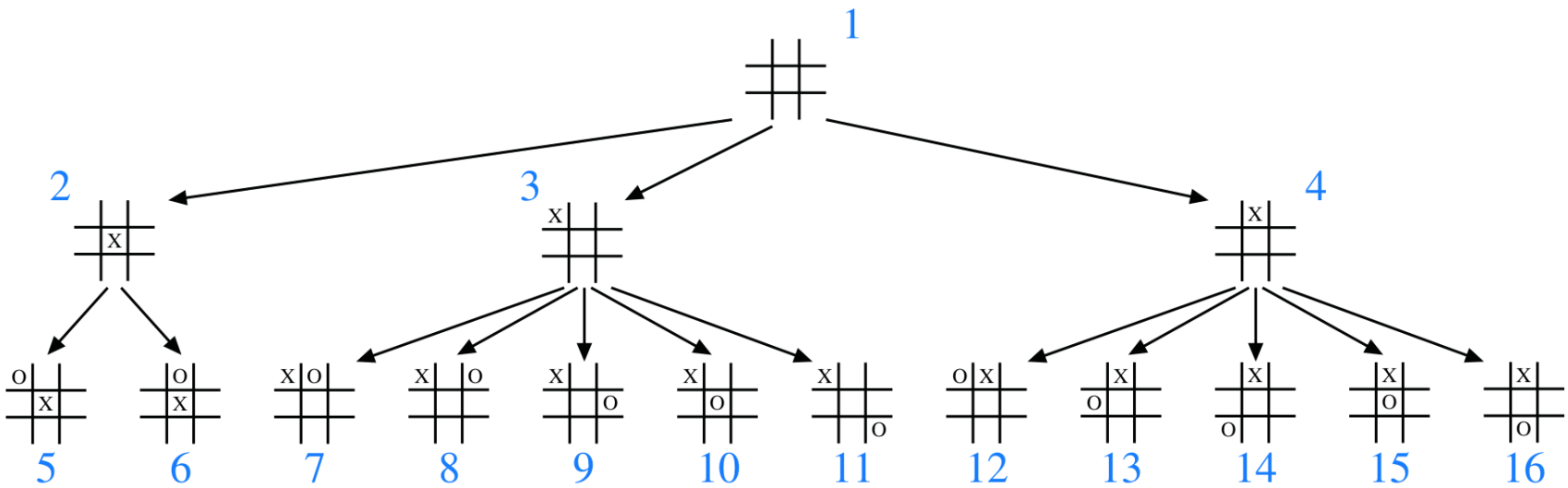
Algorithm `breadthfirst( )`
  Initialize queue Q to contain `root( )`
  while Q not empty **do**
    $p = Q$.`dequeue( )` ▷p is the oldest entry in the queue
    perform the 'visit' action for position p
    for each child c in `children`(p) **do**
      Q.`enqueue`(c) ▷add p's children to the end of the queue for later visits

Running time: $O(n)$, where n is the number of positions in the tree

Breadth-First Example: Game Trees

A **game tree** represents the possible choices of moves that might be made by a player (or computer) during a game, with the root of the tree being the initial configuration for the game

A partial game tree for Tic-Tac-Toe:



A breadth-first traversal of such a game tree is often performed because a computer may be unable to explore a complete game tree in a limited amount of time

Inorder Traversal of a Binary Tree

Preorder, postorder, and breadth-first traversals, introduced for general trees, can be directly applied to binary trees

An **inorder traversal** (specific for binary trees) visits a position between the recursive traversals of its left and right subtrees

Can be informally viewed as visiting the nodes of a tree T "from left to right"

Inorder traversal of the subtree rooted at a position p:

Algorithm `inorder`(p)
  if p has a left child lc **then**
    `inorder`(lc) ▷recursively traverse the left subtree of p
  perform the 'visit' action for position p
  if p has a right child rc **then**
    `inorder`(rc) ▷recursively traverse the right subtree of p

Running time:

Inorder Traversal of a Binary Tree

Preorder, postorder, and breadth-first traversals, introduced for general trees, can be directly applied to binary trees

An **inorder traversal** (specific for binary trees) visits a position between the recursive traversals of its left and right subtrees

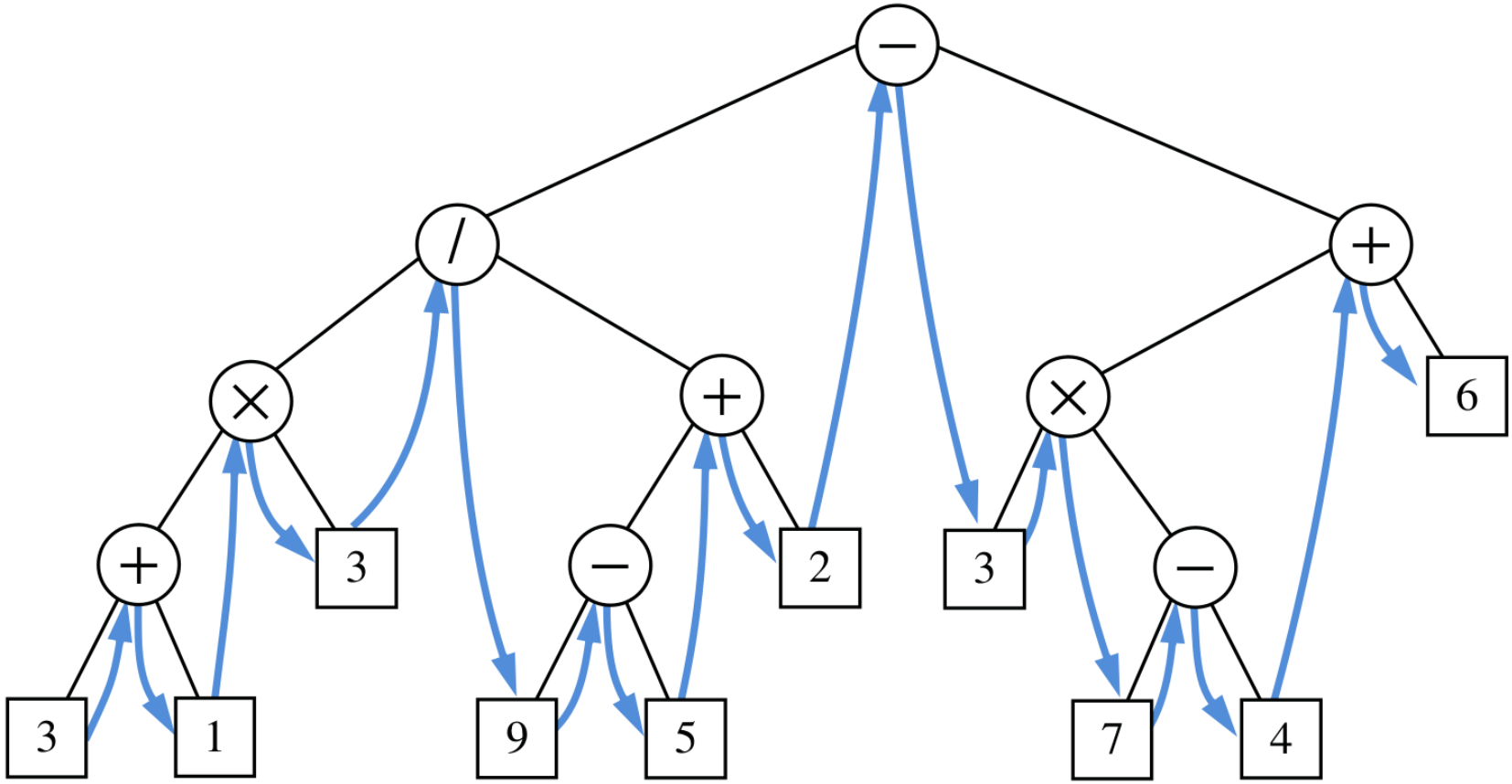
Can be informally viewed as visiting the nodes of a tree T "from left to right"

Inorder traversal of the subtree rooted at a position p:

Algorithm `inorder`(p)
  if p has a left child lc **then**
    `inorder`(lc) ▷recursively traverse the left subtree of p
  perform the 'visit' action for position p
  if p has a right child rc **then**
    `inorder`(rc) ▷recursively traverse the right subtree of p

Running time: $O(n)$, where n is the number of positions in the tree

Inorder Traversal Example



The inorder traversal visits positions in a consistent order with the standard representation of the expression (albeit without parentheses)

Implementing Tree Traversals in Java

Note that the Tree ADT includes the supporting methods:

- iterator(): Returns an iterator for all elements in the tree
- positions(): Returns an iterable collection of all positions of the tree

In what order should these iterations report their results?

We demonstrate how any of the tree traversal algorithms we have introduced can be used to produce these iterations as concrete implementations within the `AbstractTree` or `AbstractBinaryTree` base classes

Iteration of All Elements of a Tree

An iteration of all *elements* of a tree can easily be produced if we have an iteration of all *positions* of that tree

```
1  //---------------- nested ElementIterator class ----------------
2  /* This class adapts the iteration produced by positions() to return elements. */
3  private class ElementIterator implements Iterator<E> {
4    Iterator<Position<E>> posIterator = positions( ).iterator( );
5    public boolean hasNext( ) { return posIterator.hasNext( ); }
6    public E next( ) { return posIterator.next( ).getElement( ); } // return element!
7    public void remove( ) { posIterator.remove( ); }
8  }
9
10 /** Returns an iterator of the elements stored in the tree. */
11 public Iterator<E> iterator( ) { return new ElementIterator( ); }
```

Which class should this code go in?

Iteration of All Positions of a Tree

To implement the `positions()` method, we have a choice of tree traversal algorithms

Given that there are advantages to each of those traversal orders, we provide public implementations of each strategy that can be called directly by a user of our class

We can then trivially adapt one of those as a default order for the `positions` method of the `AbstractTree` class

```
public Iterable<Position<E>> positions( ) { return preorder( ); }
```

Preorder Traversal Implementation

Our goal is to provide a public method `preorder()`, as part of the `AbstractTree` class, which returns an iterable container of the positions of the tree in **preorder**

For ease of implementation, we choose to produce a **snapshot iterator** returning a list of all positions

```
1  /** Adds positions of the subtree rooted at Position p to the given snapshot. */
2  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3    snapshot.add(p);  // for preorder, we add position p before exploring subtrees
4    for (Position<E> c : children(p))
5      preorderSubtree(c, snapshot);
6  }
7  /** Returns an iterable collection of positions of the tree, reported in preorder. */
8  public Iterable<Position<E>> preorder( ) {
9    List<Position<E>> snapshot = new ArrayList<>( );
10   if (!isEmpty( ))
11     preorderSubtree(root( ), snapshot);      // fill the snapshot recursively
12   return snapshot;
13 }
```

Postorder Traversal Implementation

We implement a **postorder traversal** using a similar design as we used for a preorder traversal

The only difference is that a 'visited' position in not added to a postorder snapshot until *after* all of its subtrees have been traversed

```
1  /** Adds positions of the subtree rooted at Position p to the given snapshot. */
2  private void postorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3    for (Position<E> c : children(p))
4      postorderSubtree(c, snapshot);
5    snapshot.add(p);  // for postorder, we add position p after exploring subtrees
6  }
7  /** Returns an iterable collection of positions of the tree, reported in postorder. */
8  public Iterable<Position<E>> postorder( ) {
9    List<Position<E>> snapshot = new ArrayList<>( );
10   if (!isEmpty( ))
11     postorderSubtree(root( ), snapshot);    // fill the snapshot recursively
12   return snapshot;
13 }
```

Breadth-First Traversal Implementation

Recall that the **breadth-first traversal** algorithm is not recursive; it relies on a queue of positions to manage the traversal process

We use the `LinkedQueue` class although any implementation of the queue ADT would suffice

```
1  /** Returns an iterable collection of positions of the tree in breadth-first order. */
2  public Iterable<Position<E>> breadthfirst( ) {
3    List<Position<E>> snapshot = new ArrayList<>( );
4    if (!isEmpty( )) {
5      Queue<Position<E>> fringe = new LinkedQueue<>( );
6      fringe.enqueue(root( ));                    // start with the root
7      while (!fringe.isEmpty( )) {
8        Position<E> p = fringe.dequeue( );        // remove from front of the queue
9        snapshot.add(p);                          // report this position
10       for (Position<E> c : children(p))
11         fringe.enqueue(c);                      // add children to back of queue
12     }
13   }
14   return snapshot;
15 }
```

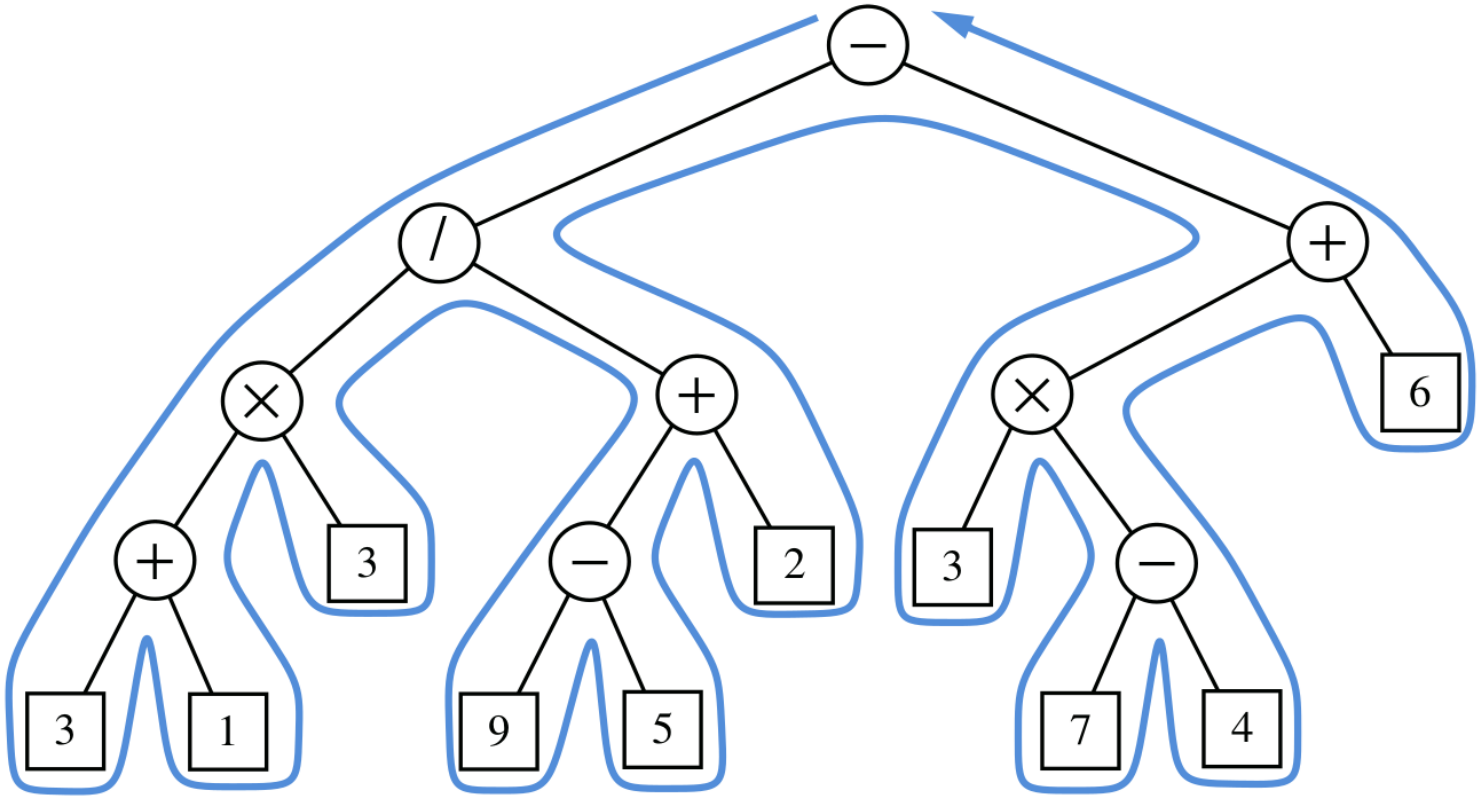
Inorder Traversal Implementation

The **inorder traversal**, unlike preorder, postorder, and breadth-first traversal algorithms, only applies to binary trees
We therefore include its definition within the body of the `AbstractBinaryTree` class

```
1  /** Adds positions of the subtree rooted at Position p to the given snapshot. */
2  private void inorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3    if (left(p) != null)
4      inorderSubtree(left(p), snapshot);
5    snapshot.add(p);
6    if (right(p) != null)
7      inorderSubtree(right(p), snapshot);
8  }
9  /** Returns an iterable collection of positions of the tree, reported in inorder. */
10 public Iterable<Position<E>> inorder( ) {
11   List<Position<E>> snapshot = new ArrayList<>( );
12   if (!isEmpty( ))
13     inorderSubtree(root( ), snapshot);          // fill the snapshot recursively
14   return snapshot;
15 }
16 /** Overrides positions to make inorder the default order for binary trees. */
17 public Iterable<Position<E>> positions( ) {
18   return inorder( );
19 }
```

Euler Tours

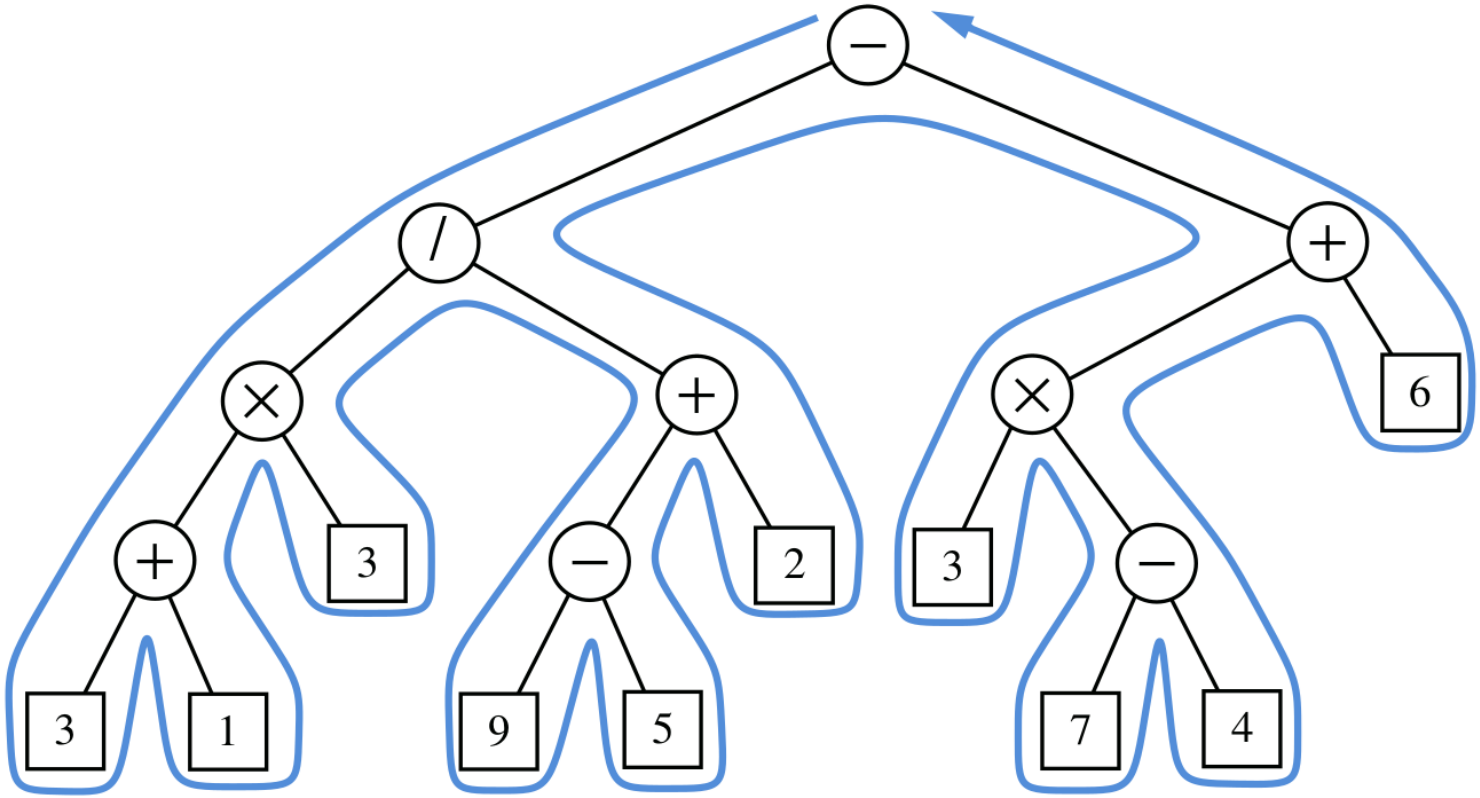
The **Euler tour traversal** of a tree T can be informally defined as a 'walk' around T, where we start by going from the root toward its leftmost child, viewing the edges of T as being 'walls' that we always keep to our left



Running time:

Euler Tours

The **Euler tour traversal** of a tree T can be informally defined as a 'walk' around T, where we start by going from the root toward its leftmost child, viewing the edges of T as being 'walls' that we always keep to our left



Running time: $O(n)$, where n is the number of positions in the tree

Summary

Reading

Section 8.4 Tree Traversal Algorithms

Questions?