Tree Algorithms

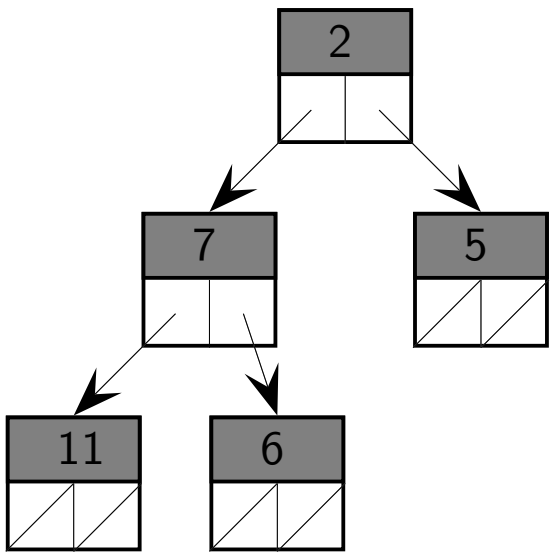
CMPT 145

Sequences and Tree Traversals

- Sequential data stored in a list has a unique ordering.
- Data stored in a tree does not have a unique sequence.
- There are 4 distinct sequences that an algorithm could use to explore a tree.
- Each sequence can be expressed as an algorithm, called a traversal.
- Almost every algorithm you need to work with data stored in a tree will be based on one of these traversals.

Breadth-first Sequence

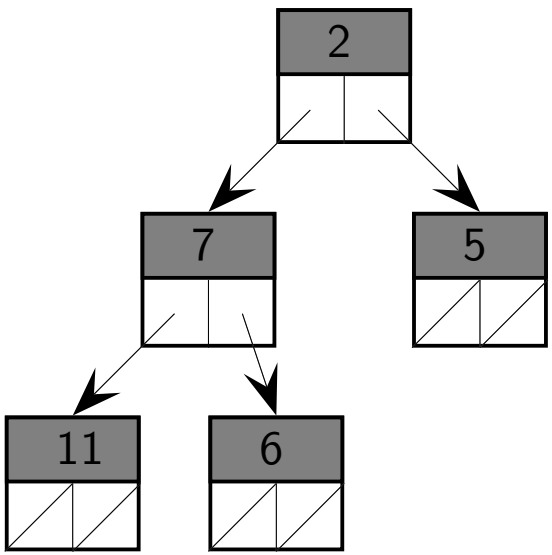
1. Each level from top to bottom, from left to right
2. Siblings before children



Breadth-first sequence: 2 7 5 11 6

Breadth-first Sequence Formally

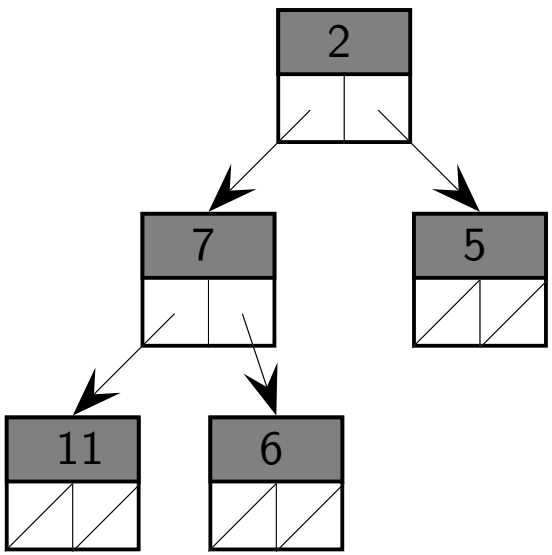
1. Nodes at level i before nodes at level $i + 1$
2. From left to right



Breadth-first sequence: 2 7 5 11 6

Pre-Order Sequence

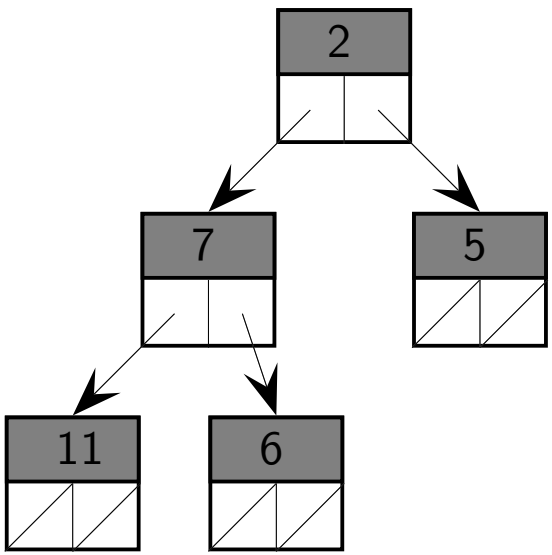
1. Root before children
2. Children from left to right



Pre-Order sequence: 2 7 11 6 5

Post-Order Sequence

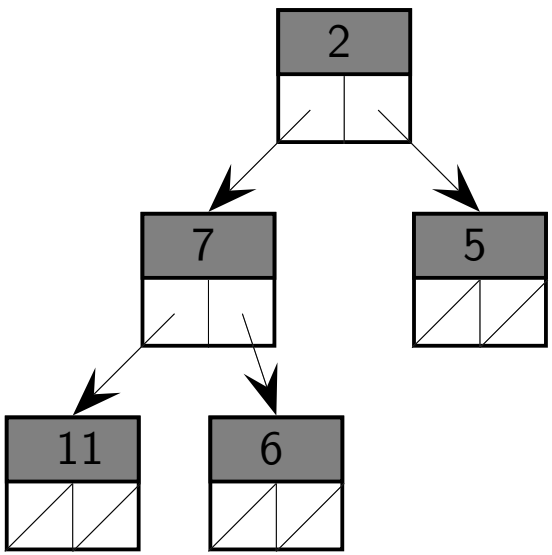
1. Children before root
2. Children from left to right



Post-Order sequence: 11 6 7 5 2

In-Order Sequence

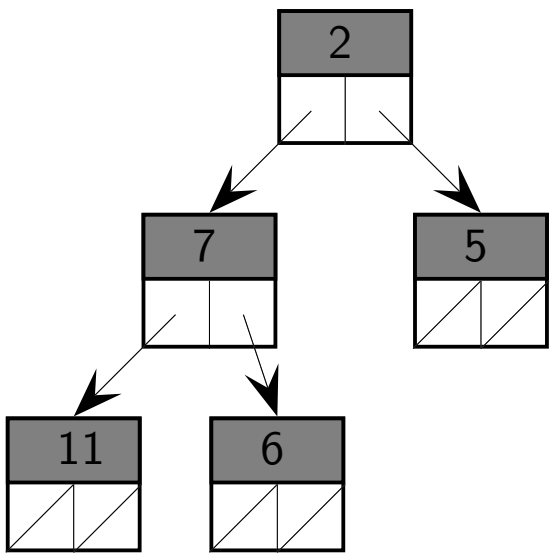
1. Left child before root
2. Root before right child



In-Order sequence: 11 7 6 2 5

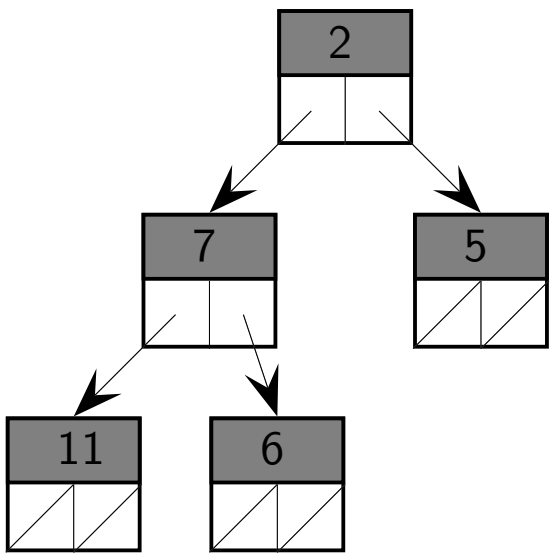
Pre-Order Sequence by hand

1. Write down the root.
2. Draw a box for left and right subtrees after the root.
3. Fill in each box using Pre-order Sequence.



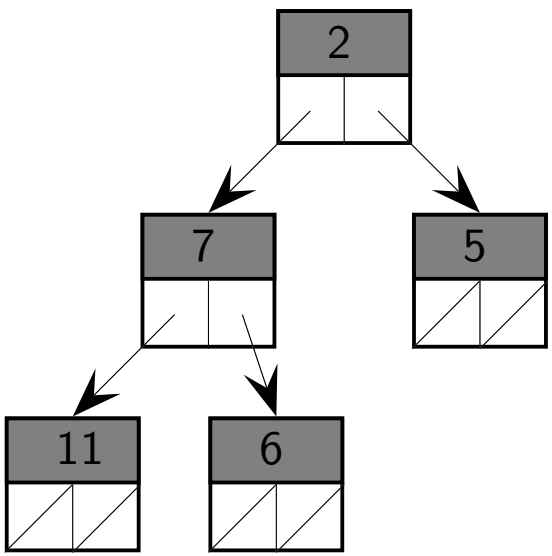
Post-Order Sequence by hand

1. Draw a box for left and right subtrees.
2. Write down the root after the boxes
3. Fill in each box using Post-order Sequence.



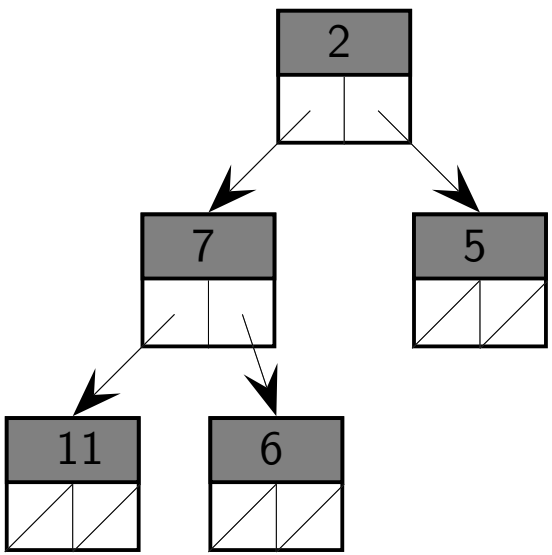
In-Order Sequence by hand

1. Draw a box for left and right subtrees.
2. Write down the root between the two boxes.
3. Fill in each box using In-Order Sequence.



Pre-Order Traversal: Pseudo-code

- If the tree is empty, do nothing.
- Otherwise:
  1. Process the root of the subtree.
  2. Recursively process the left-subtree in pre-order sequence
  3. Recursively process the right-subtree in pre-order sequence



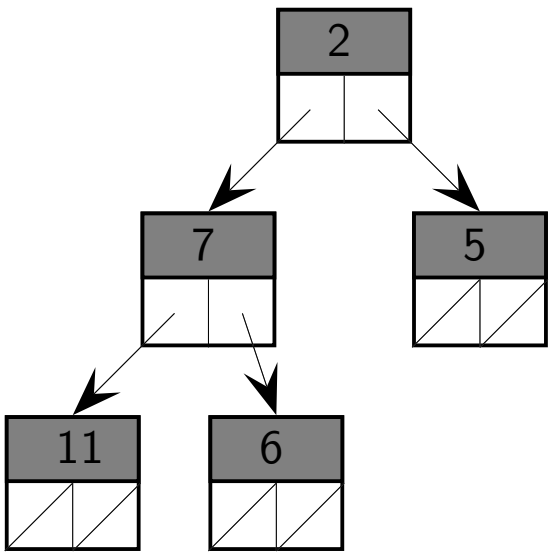
Pre-Order Traversal algorithm

```
1 def pre_order(tnode):
2     if tnode is None:
3         return
4     else:
5         print(treenode.get_data(tnode), end=" ")
6         pre_order(treenode.get_left(tnode))
7         pre_order(treenode.get_right(tnode))
```

Root first; left recursively, then right.

In-Order Traversal: Pseudo-code

- If the tree is empty, do nothing.
- Otherwise:
  1. Recursively process the left-subtree in in-order sequence
  2. Process the root of the subtree.
  3. Recursively process the right-subtree in in-order sequence



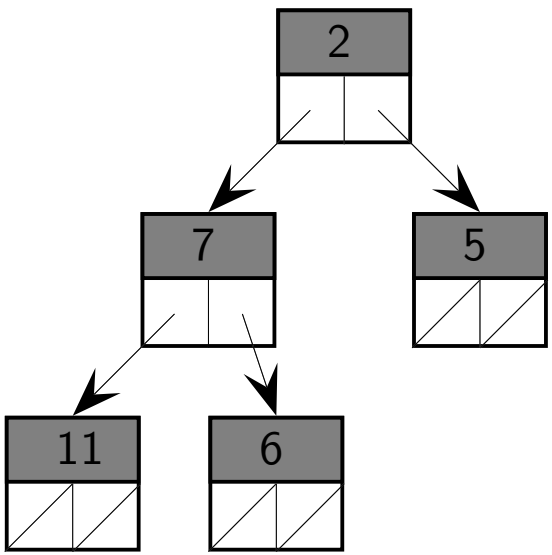
In-Order Traversal algorithm

```
1 def in_order(tnode):
2     if tnode is None:
3         return
4     else:
5         in_order(treenode.get_left(tnode))
6         print(treenode.get_data(tnode), end=" ")
7         in_order(treenode.get_right(tnode)))
```

Left recursively first; root, then right.

Post-Order Traversal: Pseudo-code

- If the tree is empty, do nothing.
- Otherwise:
  1. Recursively process the left-subtree in post-order sequence
  2. Recursively process the right-subtree in post-order sequence
  3. Process the root of the subtree.



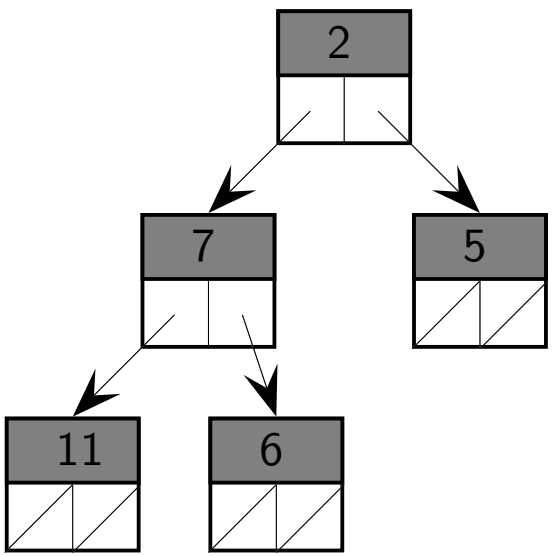
Post-Order Traversal algorithm

```
1 def post_order(tnode):
2     if tnode is None:
3         return
4     else:
5         post_order(treenode.get_left(tnode))
6         post_order(treenode.get_right(tnode))
7         print(treenode.get_data(tnode), end=" ")
```

Left recursively first; right, then root.

Breadth-first Traversal: Pseudo-code

1. Each level from top to bottom, from left to right
2. Siblings before children



Breadth-First-Order Traversal algorithm

```
1  def bft(nodes, order):
2      if Queue.size(nodes) > 0:
3          current = Queue.dequeue(nodes)
4          if current is not None:
5              Queue.enqueue(order, treenode.get_data(current))
6              Queue.enqueue(nodes, treenode.get_left(current))
7              Queue.enqueue(nodes, treenode.get_right(current))
8              bft(nodes, order)
9
10 def breadth_first_order(tnode):
11     explore = Queue.create()
12     Queue.enqueue(explore, tnode)
13     sequenced = Queue.create()
14     bft(explore, sequenced)
15
16     while not Queue.is_empty(sequenced):
17         n = Queue.dequeue(sequenced)
18         print(n, end=" ")
```

Siblings before children

Calculating tree height

```
1  def height(tnode):
2      if tnode is None:
3          return 0
4      else:
5          lh = height(treenode.get_left(tnode))
6          rh = height(treenode.get_right(tnode))
7          return 1 + max(lh, rh)
```

Counting nodes in a tree

```
1 def count(tnode):
2     if tnode == None:
3         return 0
4     else:
5         return 1 + count(treenode.get_left(tnode)) \
6                  + count(treenode.get_right(tnode))
```

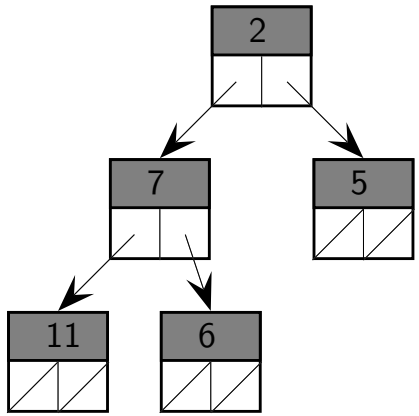
Searching for a data value in a tree

```
1  def member(tnode, val):
2      if tnode == None:
3          return False
4      elif treenode.get_data(tnode) == val:
5          return True
6      else:
7          return member(treenode.get_left(tnode), val) \
8              or member(treenode.get_right(tnode), val)
```

Algorithms on trees

- Almost always mirror the formal definition of trees.
- Frequently perform an implied traversal.
- Very easy after a while.

Exercises



Define the following functions:

- To sum the data values in a tree.
- To find the smallest data value in a tree.
- To substitute a target value with a replacement value
- To swap the left and right branches in the tree.
- To find the level of the leaf closest to the root.