5SENG003W - Algorithms, Week 4

Dr. Klaus Draeger

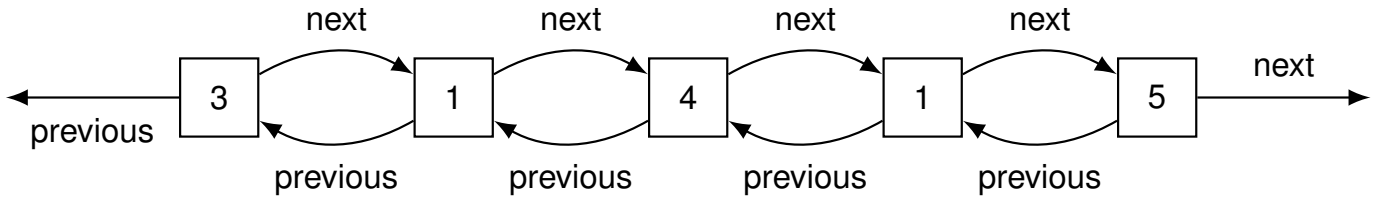
RECAP

Last week. . .

- We talked about the sorting problem and some solutions
  - Selection Sort, Bubble Sort
    - Both grow a sorted region element by element
    - Complexity $O(N^2)$
  - Merge Sort
    - Uses the Divide and Conquer strategy
    - Splits the array in half, sorts each half (recursion), then merges them
    - Complexity $O(N \log N)$
- All operate on sequential data structures, specifically arrays
- This week, we will start looking at other data structures

Linked data structures

- We have briefly seen linked lists before



- The underlying structure is that of a node:

```
public class ListNode{
    private int data;           // contents of this node
    private ListNode next;
    private ListNode previous;
    public void setData(int newData){ data = newData; }
    public int getData(){ return data; }
    // ...
}
```

- This structure - nodes containing data and linked together using pointers - is very general

What are these pointers of which you speak?

- I may occasionally mention "**`next`** pointers" etc
- A pointer is a value which represents the location of data
- This enables several variables to "point at" the same data
  - The situation in C++:

```
int first = 17;       // first contains the value directly
int second = first;   // second contains a COPY of the value
first++;         // changes first, but not second - these are separate data

int *third = new int(17); // third points at the value
int *fourth = third;      // fourth points at the SAME data
(*third)++;               // changes the (shared) data
cout << *fourth;          // outputs 18
```

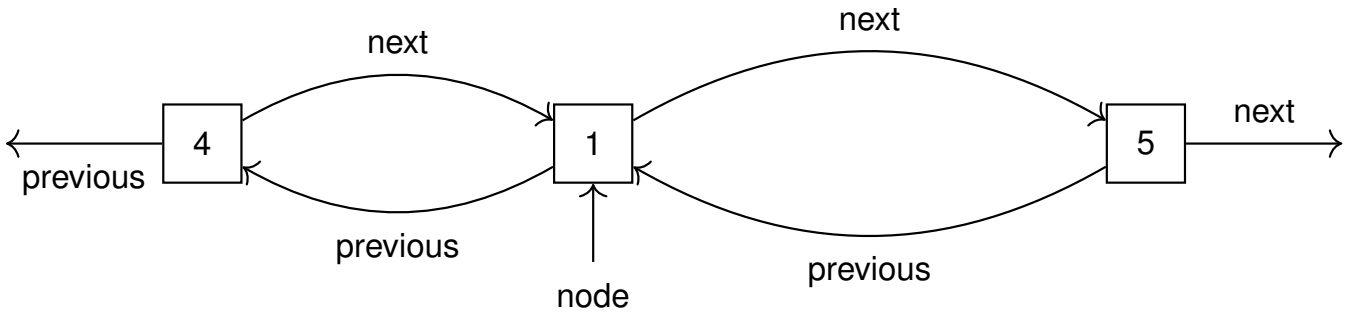
  - In Java:
    - Basic types (like ints) work like first/second above
    - Objects are (implicitly) handled through pointers:

```
public class MyClass{ /* ... */ }
MyClass first = new MyClass();
MyClass second = first;       // second points at the SAME object
```

Operations on lists: insertion

- Suppose we have:
  - Some data, a position in a list (given by a node)
- How can we insert the data in the list after that position?
  - Create a new node containing the data
  - Link the new node with the existing ones

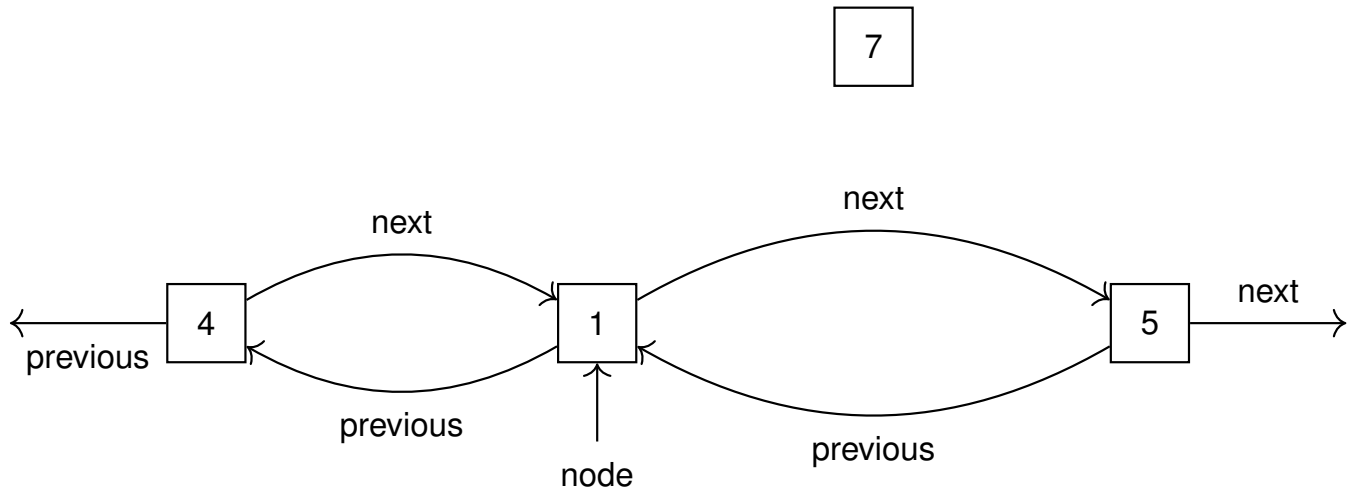
```
public void insertAfter(int newData, ListNode node){
    Node newNode = new ListNode(), next = node.getNext();
    newNode.setData(data);
    newNode.setNext(node.getNext());
    newNode.setPrevious(node);
    node.setNext(newNode);
    if(next != null)
        next.setPrevious(newNode);
}
```



Operations on lists: insertion

- Suppose we have:
  - Some data, a position in a list (given by a node)
- How can we insert the data in the list after that position?
  - Create a new node containing the data
  - Link the new node with the existing ones

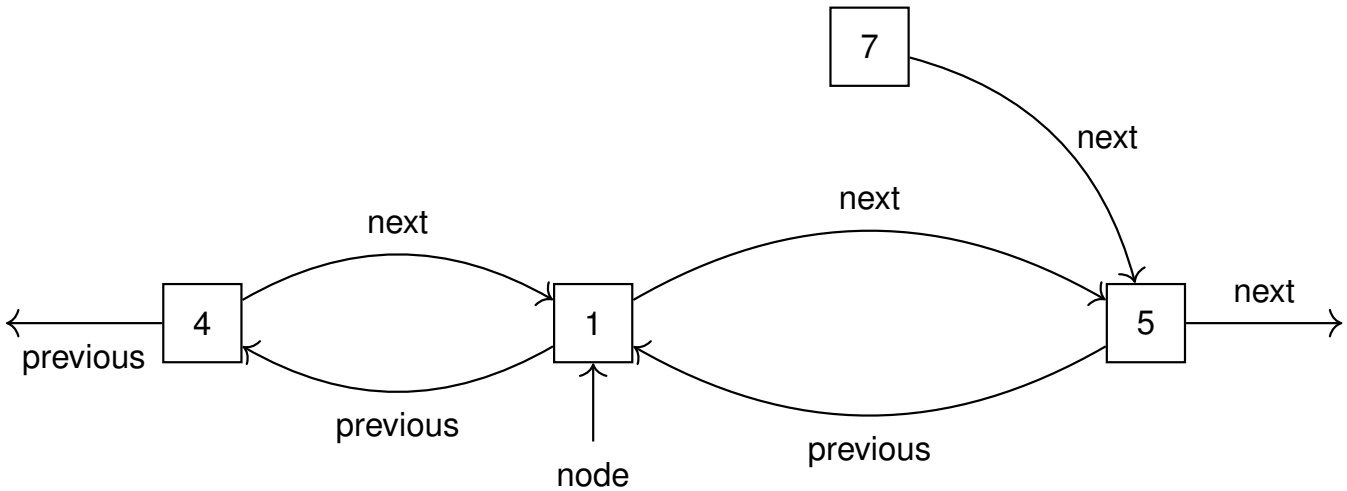
```
public void insertAfter(int newData, ListNode node){
    Node newNode = new ListNode(), next = node.getNext();
    newNode.setData(data);
    newNode.setNext(node.getNext());
    newNode.setPrevious(node);
    node.setNext(newNode);
    if(next != null)
        next.setPrevious(newNode);
}
```



Operations on lists: insertion

- Suppose we have:
  - Some data, a position in a list (given by a node)
- How can we insert the data in the list after that position?
  - Create a new node containing the data
  - Link the new node with the existing ones

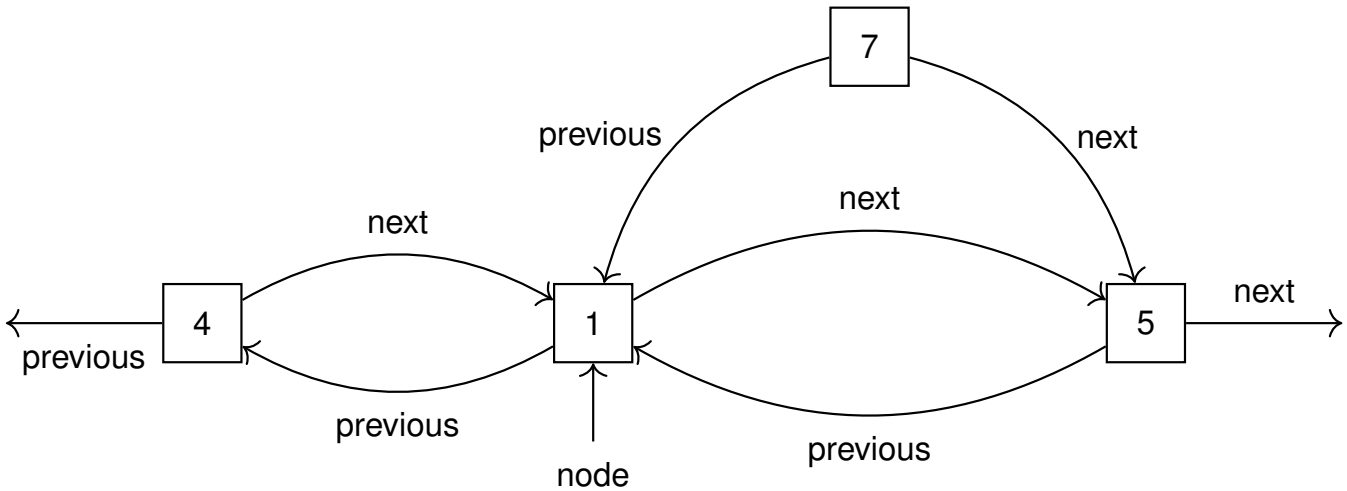
```
public void insertAfter(int newData, ListNode node){
    Node newNode = new ListNode(), next = node.getNext();
    newNode.setData(data);
    newNode.setNext(node.getNext());
    newNode.setPrevious(node);
    node.setNext(newNode);
    if(next != null)
        next.setPrevious(newNode);
}
```



Operations on lists: insertion

- Suppose we have:
  - Some data, a position in a list (given by a node)
- How can we insert the data in the list after that position?
  - Create a new node containing the data
  - Link the new node with the existing ones

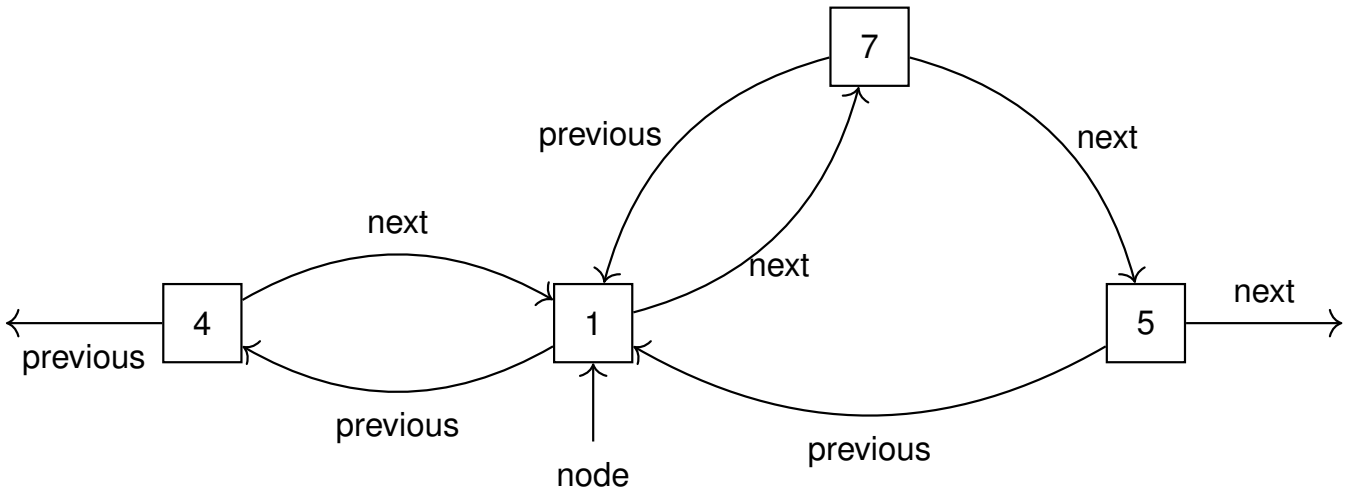
```
public void insertAfter(int newData, ListNode node){
    Node newNode = new ListNode(), next = node.getNext();
    newNode.setData(data);
    newNode.setNext(node.getNext());
    newNode.setPrevious(node);
    node.setNext(newNode);
    if(next != null)
        next.setPrevious(newNode);
}
```



Operations on lists: insertion

- Suppose we have:
  - Some data, a position in a list (given by a node)
- How can we insert the data in the list after that position?
  - Create a new node containing the data
  - Link the new node with the existing ones

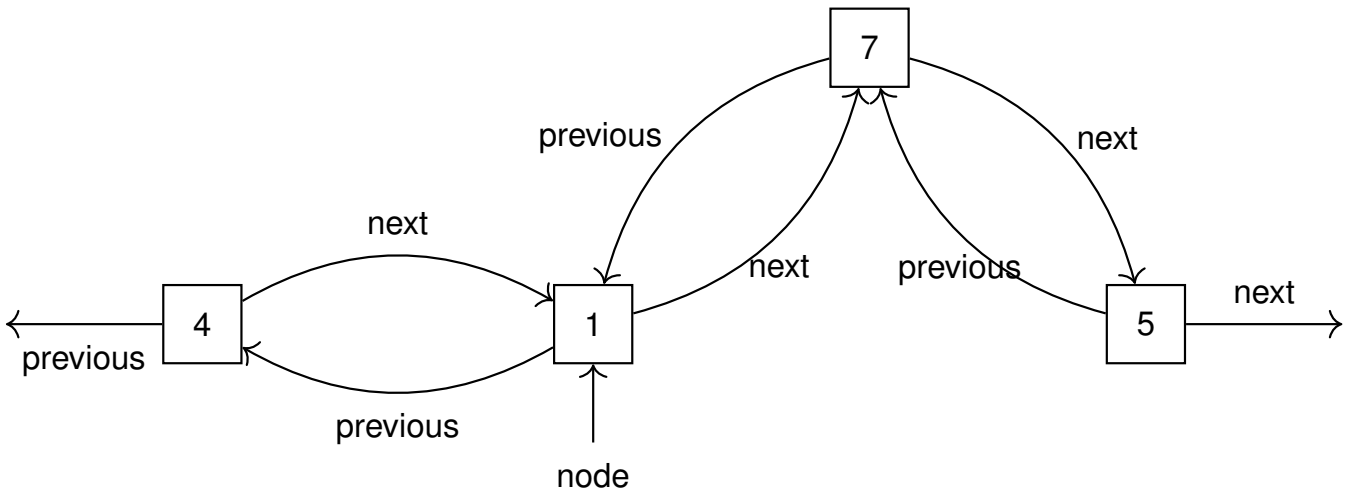
```
public void insertAfter(int newData, ListNode node){
    Node newNode = new ListNode(), next = node.getNext();
    newNode.setData(data);
    newNode.setNext(node.getNext());
    newNode.setPrevious(node);
    node.setNext(newNode);
    if(next != null)
        next.setPrevious(newNode);
}
```



Operations on lists: insertion

- Suppose we have:
  - Some data, a position in a list (given by a node)
- How can we insert the data in the list after that position?
  - Create a new node containing the data
  - Link the new node with the existing ones

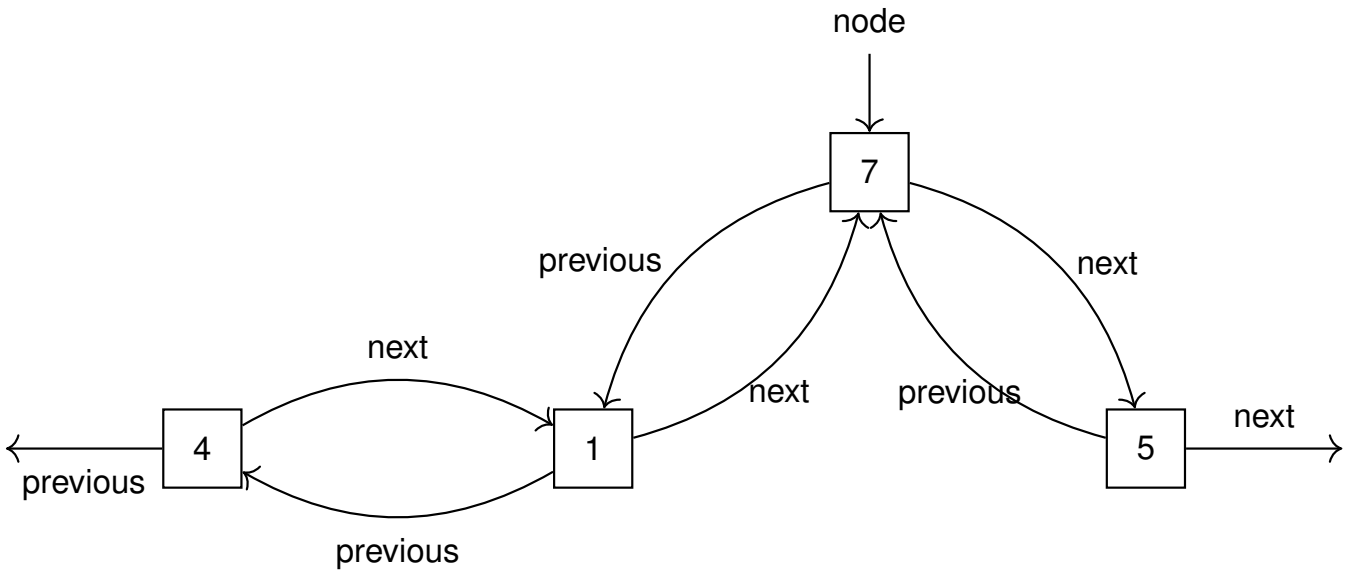
```
public void insertAfter(int newData, ListNode node){
    Node newNode = new ListNode(), next = node.getNext();
    newNode.setData(data);
    newNode.setNext(node.getNext());
    newNode.setPrevious(node);
    node.setNext(newNode);
    if(next != null)
        next.setPrevious(newNode);
}
```



Operations on lists: deletion

- Suppose we have:
  - A position in a list (given by a node)
- How can we remove the data at that position (i.e. the given node)?
  - Redirect previous and next pointers on its neighbours

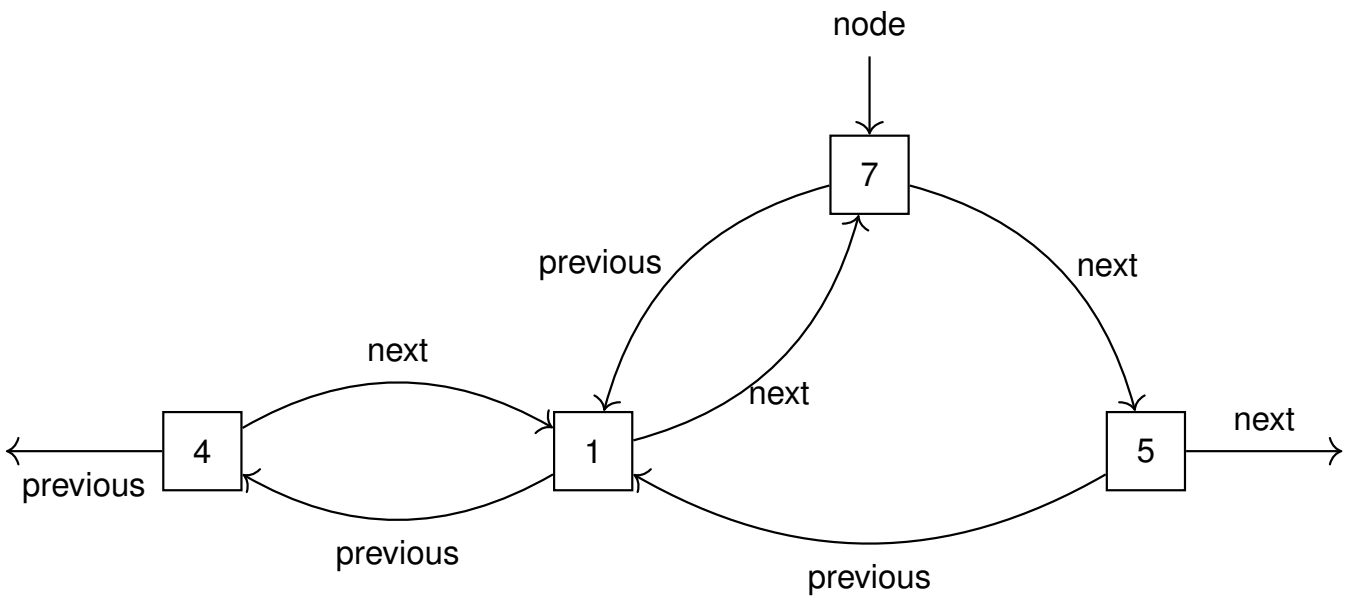
```
public void remove(ListNode node){
    ListNode next = node.getNext(), previous = node.getPrevious();
    if(next != null)
        next.setPrevious(previous);
    if(previous != null)
        previous.setNext(next);
}
```



Operations on lists: deletion

- Suppose we have:
  - A position in a list (given by a node)
- How can we remove the data at that position (i.e. the given node)?
  - Redirect previous and next pointers on its neighbours

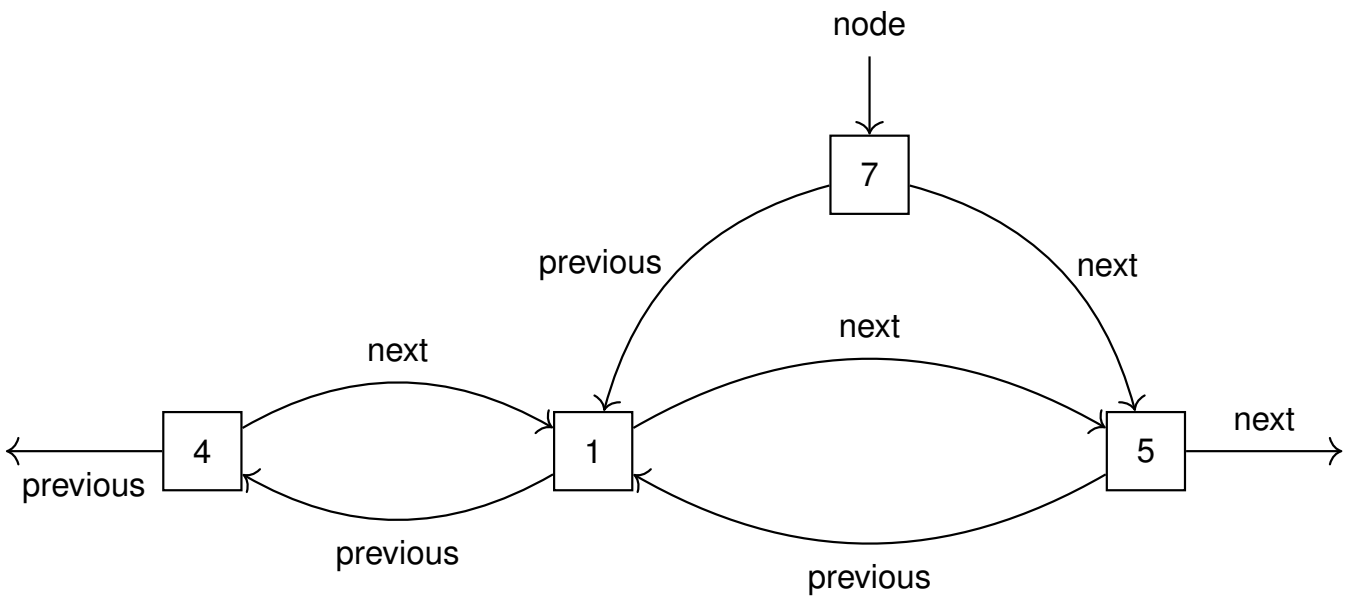
```
public void remove(ListNode node){
    ListNode next = node.getNext(), previous = node.getPrevious();
    if(next != null)
        next.setPrevious(previous);
    if(previous != null)
        previous.setNext(next);
}
```



Operations on lists: deletion

- Suppose we have:
  - A position in a list (given by a node)
- How can we remove the data at that position (i.e. the given node)?
  - Redirect previous and next pointers on its neighbours

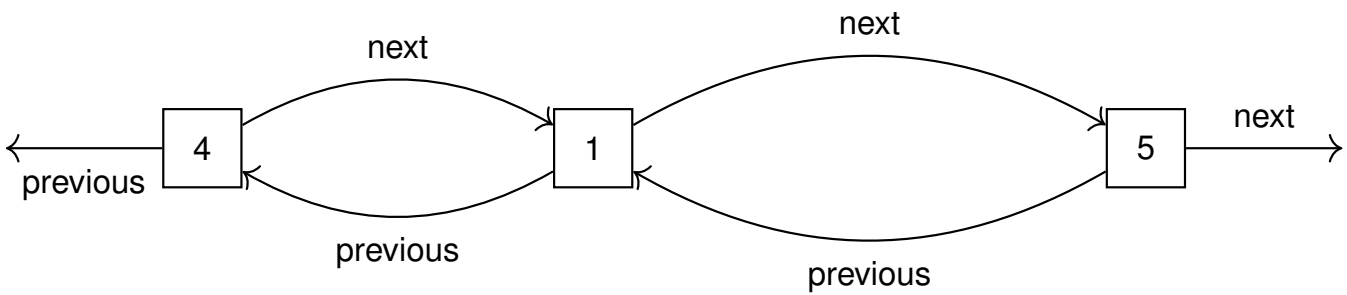
```
public void remove(ListNode node){
    ListNode next = node.getNext(), previous = node.getPrevious();
    if(next != null)
        next.setPrevious(previous);
    if(previous != null)
        previous.setNext(next);
}
```



Operations on lists: deletion

- Suppose we have:
  - A position in a list (given by a node)
- How can we remove the data at that position (i.e. the given node)?
  - Redirect previous and next pointers on its neighbours

```
public void remove(ListNode node){
    ListNode next = node.getNext(), previous = node.getPrevious();
    if(next != null)
        next.setPrevious(previous);
    if(previous != null)
        previous.setNext(next);
}
```



Linked lists

- Linked lists are a common pre-defined data structure
  - Java: **`java.util.LinkedList`**
  - C++: **`std::list`**
- The nodes are generally not directly accessible
  - Inner/nested classes inside the actual list class
  - Manipulated by the list object itself
  - Indirect access through **iterators**

Trees

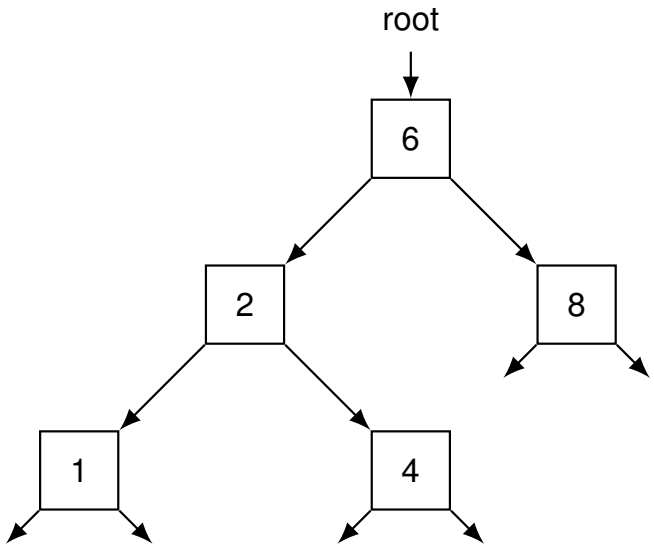
- Another data structure commonly implemented using linked nodes
- Each node has:
  - Some data
  - Some other nodes below it (its **children**)
  - A node above it (its **parent**)
    – not always explicitly included
- Any node `n` together with its children, their children, etc forms a **subtree**, with `n` as its **root**
  - The root of the whole tree will be the only node with no parent
- A node whose children are all `null` is a **leaf**.

Binary trees

- Nodes in a **binary** tree have two children:

```
public class TreeNode{
    public int data;
    public TreeNode leftChild, rightChild, parent;
}
```

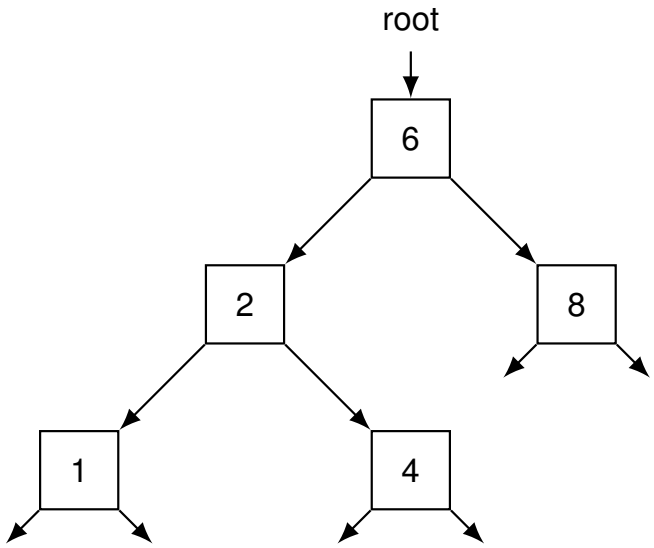
- Again using integer data for simplicity; could be any type



- The left/right child and all nodes below it form the left/right **subtree**, with the child as its root

Binary search trees

- The values in the example are **sorted**: for each node `n`,
  - anything less than `n.data` is in the left subtree
  - anything greater than `n.data` is in the right subtree



- This enables us to do binary search:
  To find the value 5, start at the root.
  - The value there is 6, which is > 5, so go down to the left
  - The value there is 2, which is < 5, so go down to the right
  - The value there is 4, which is < 5, so go down to the right
  - We have hit a `null` pointer, so the value is not in the tree

Searching in a BST

- Let us implement this form of binary search
  - We assume that the **`TreeNode`** class is contained inside a **`BinarySearchTree`** class along with the various operations:

```
public class BinarySearchTree{
    public class TreeNode{  /* as seen before */
        public int data;
        public TreeNode leftChild, rightChild, parent;
    }

    private TreeNode root;

    public TreeNode find(int findMe){
        TreeNode n = root;
        while(n != null){
            if(n.data == findMe)    // found it
                return n;
            if(n.data < findMe)     // too small, try right subtree
                n = n.rightChild;
            else                    // too large, try left subtree
                n = n.leftChild;
        }
        return null;                // value is not in the tree
    }
}
```

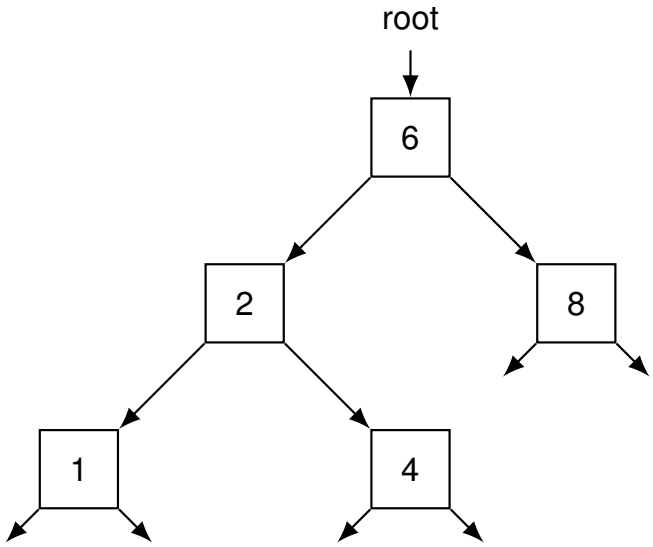
Binary search trees: operations

- There are a number of other operations we often want to do with data structures
  - Output (in increasing order)
  - Insertion
  - Deletion
- Let us have a look at how these should work
- Implementations are part of the next tutorial!

Binary search trees: output

- Suppose we have a binary search tree (given by its root node).
- We want to output its values in increasing order (As a bonus, think about how to modify this in order to output them in decreasing order)
- We will need to output the data in the root node
- All the data in the **left** subtree is **smaller**, so it needs to be output **before** the root
- All the data in the **right** subtree is **larger**, so it needs to be output **after** the root
- So in order to output the tree: if it is not `null`,
  - First output the left subtree
  - Then output the root
  - Then output the right subtree

Binary search trees: output example



To output the example tree (omitting `null` subtrees):

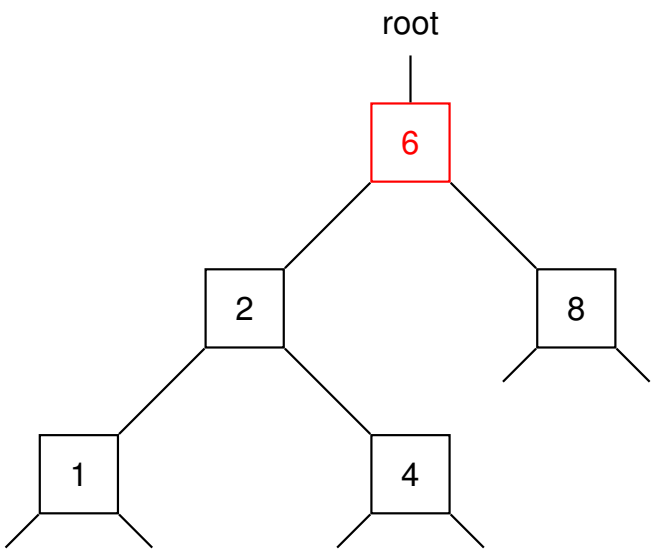
- output the left subtree
  - output the left subtree
    - output the subtree root: prints **1**
  - output the subtree root: prints **2**
  - output the right subtree
    - output the subtree root: prints **4**
- output the root: prints **6**
- output the right subtree
  - output the subtree root: prints **8**

Binary search trees: insertion

- Suppose we want to insert new data in a BST
- One thing to first decide is if we want to allow **duplicates**
  - Trees are often used to implement containers such as sets (Java: java.util.TreeSet, C++: std::set)
  - In this case we want to discard duplicate values
  - Allowing duplicates is a simple modification
- If the root is `null` (the tree is still empty), replace it with a new node containing the data; done
- Otherwise, compare the new data with what is in the root
  - If it is equal, either discard or continue in either subtree
  - If it is less, continue in the left subtree
  - If it is greater, continue in the right subtree

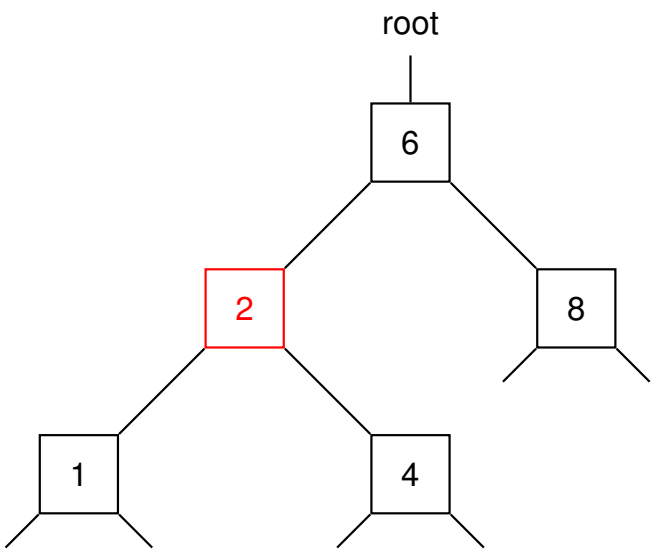
Binary search trees: insertion example

- Suppose we want to insert 3 in the example tree
  - 3 is less than 6, so go down to the left
  - 3 is greater than 2, so go down to the right
  - 3 is less than 4, so go down to the left
  - `null` found; put the value here



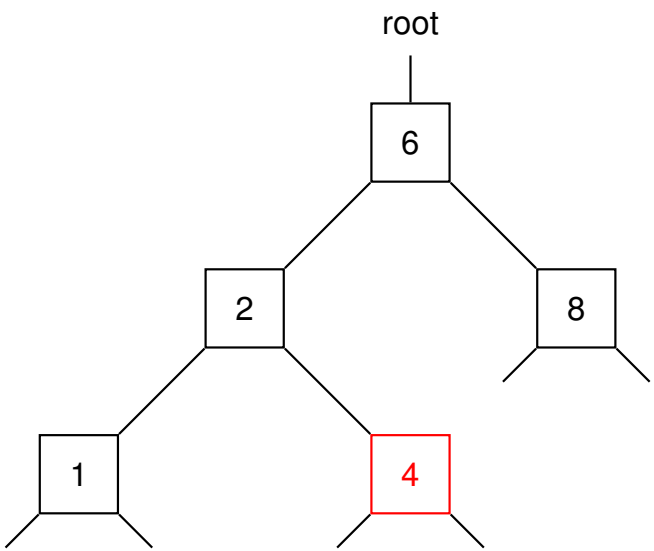
Binary search trees: insertion example

- Suppose we want to insert 3 in the example tree
  - 3 is less than 6, so go down to the left
  - 3 is greater than 2, so go down to the right
  - 3 is less than 4, so go down to the left
  - `null` found; put the value here



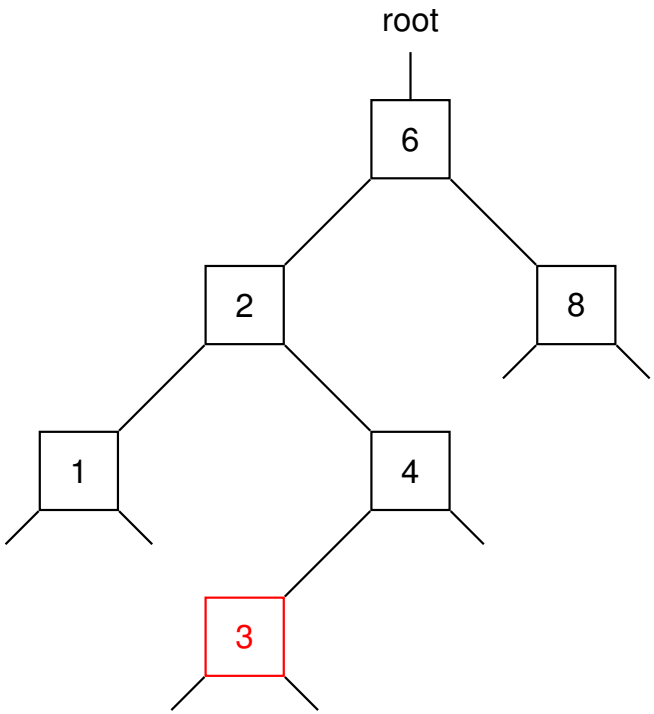
Binary search trees: insertion example

- Suppose we want to insert 3 in the example tree
  - 3 is less than 6, so go down to the left
  - 3 is greater than 2, so go down to the right
  - 3 is less than 4, so go down to the left
  - `null` found; put the value here



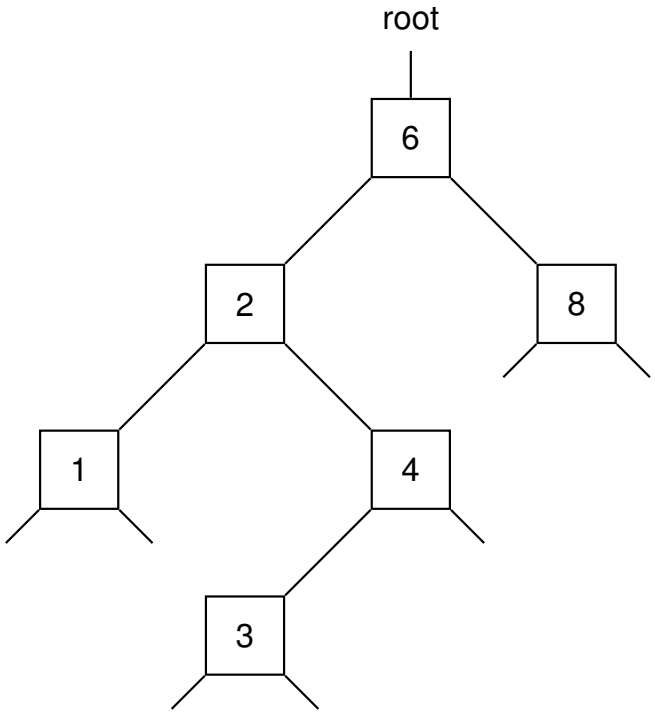
Binary search trees: insertion example

- Suppose we want to insert 3 in the example tree
  - 3 is less than 6, so go down to the left
  - 3 is greater than 2, so go down to the right
  - 3 is less than 4, so go down to the left
  - `null` found; put the value here



Binary search trees: deletion

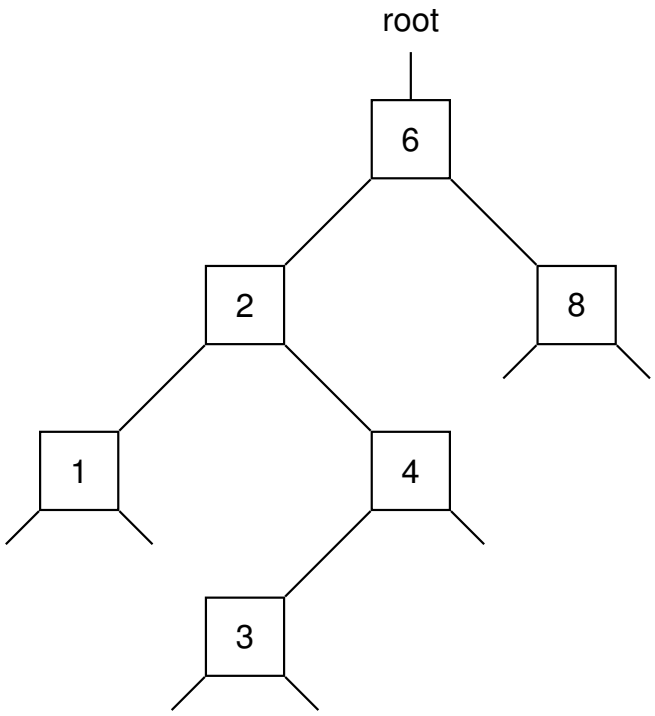
- Suppose we now want to remove a value from our tree:



- How does this work?

Deletion: the easy case

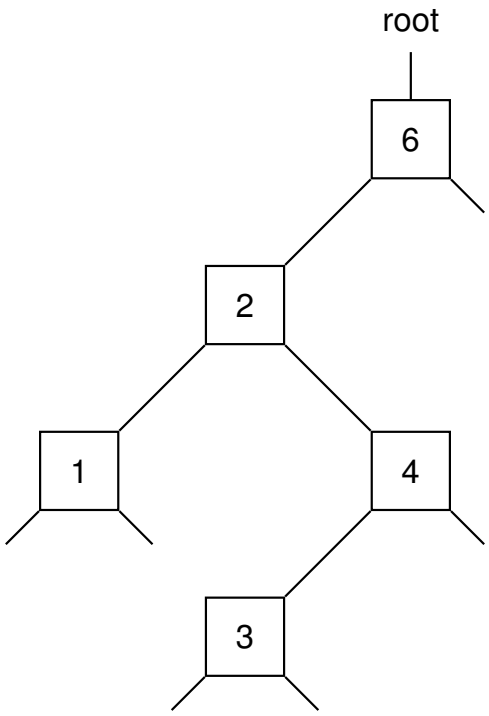
- If the value is in a leaf, we can just remove that leaf
- Example: removing 8



- But what if it isn't a leaf?

Deletion: the easy case

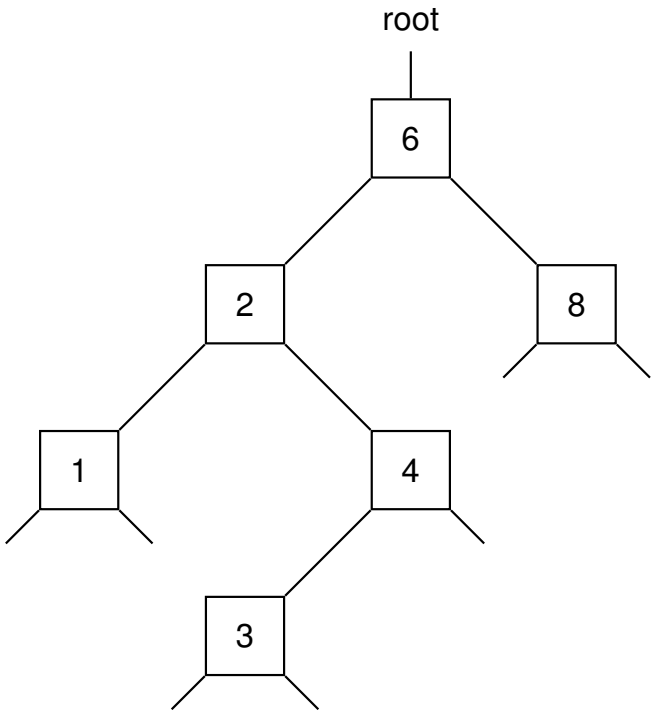
- If the value is in a leaf, we can just remove that leaf
- Example: removing 8



- But what if it isn't a leaf?

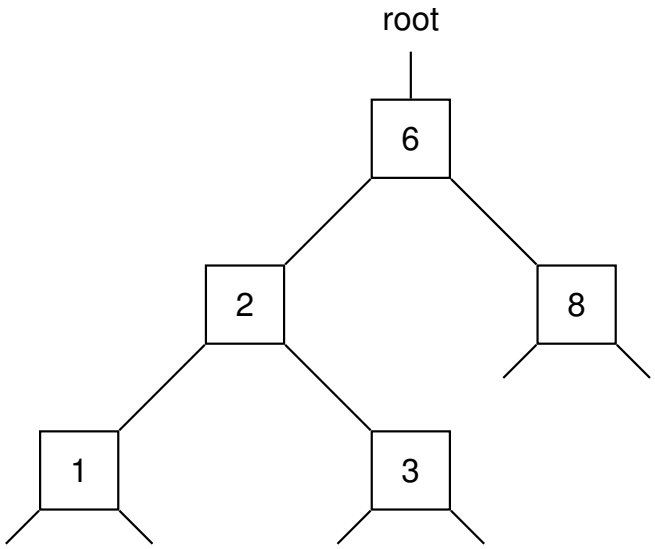
Deletion: the not-too-difficult case

- If the value is in a node with just one child, we can
  - attach that child to the node's parent instead
  - remove the node
- Example: removing 4

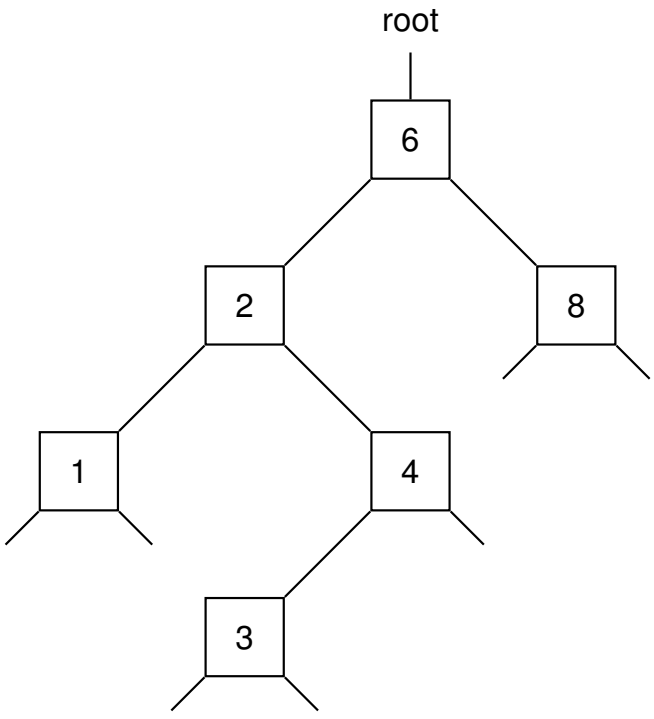


Deletion: the not-too-difficult case

- If the value is in a node with just one child, we can
  - attach that child to the node's parent instead
  - remove the node
- Example: removing 4

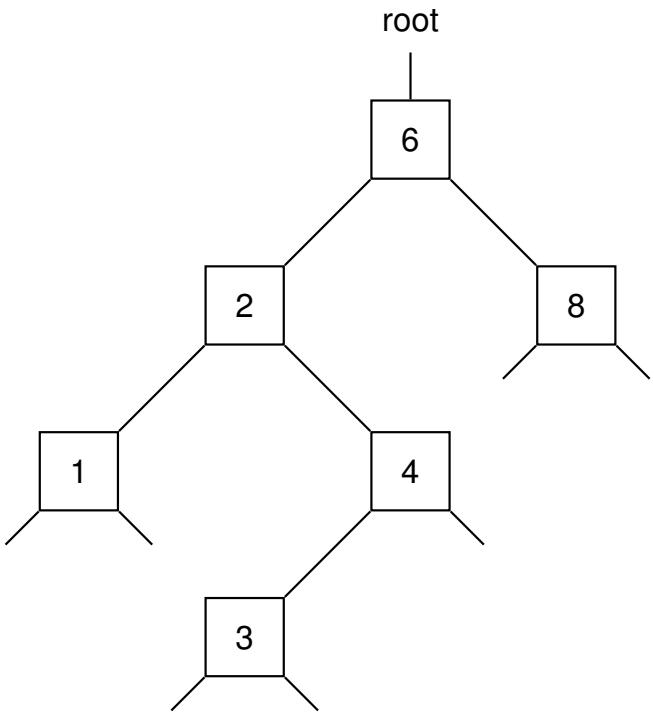


Deletion: the interesting case



- We cannot remove a node in the middle of the tree
- Instead, **move** data from below into it to **replace** the removed data
- To respect the ordering, this replacement should be:
  - greater than the remaining data in the left subtree
  - smaller than the remaining data in the right subtree

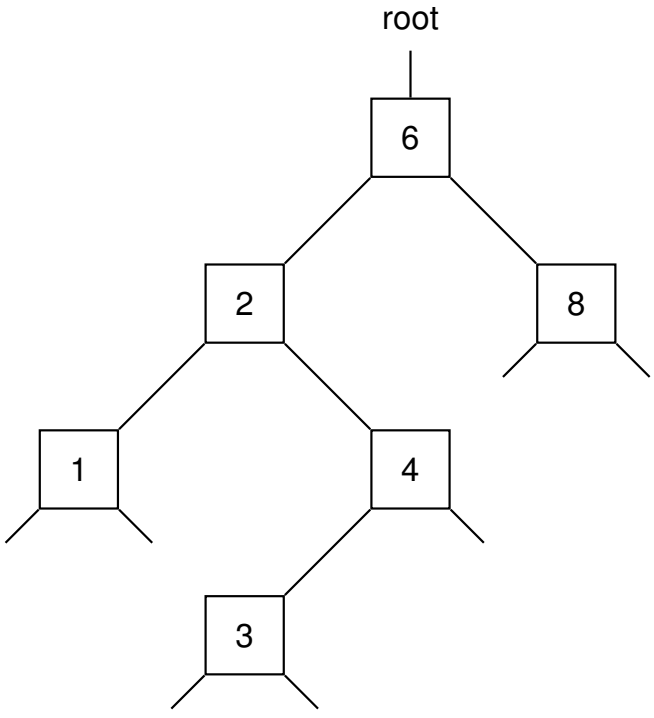
Deletion: the interesting case



- We cannot remove a node in the middle of the tree
- Instead, **move** data from below into it to **replace** the removed data
- To respect the ordering, this replacement should be:
  - either the greatest value in the left subtree
  - or the smallest value in the right subtree

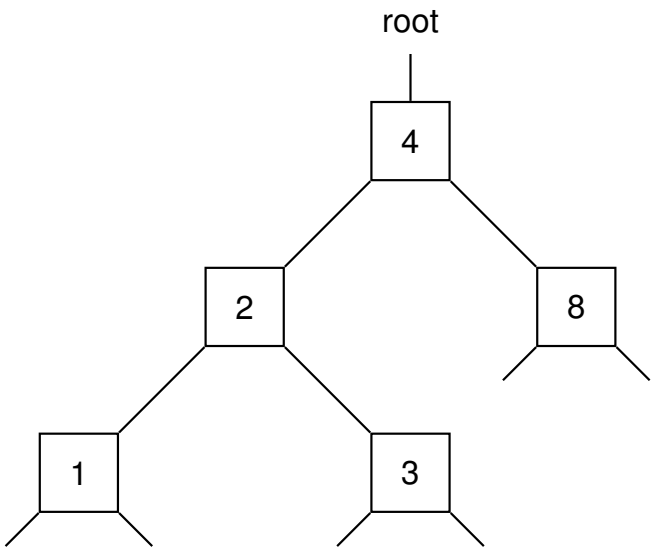
Deletion: example

- Let us remove the value 6 in our tree
  - The greatest value in the left subtree is 4
  - We can move this value into the root to replace the 6
  - The node $\mathtt{m}$ containing the 4 has no right child (since it is maximal in its subtree)
  - $\mathtt{m}$ does have a left child, which we attach to $\mathtt{m}$'s parent (the node containing the 2) instead



Deletion: example

- Let us remove the value 6 in our tree
  - The greatest value in the left subtree is 4
  - We can move this value into the root to replace the 6
  - The node `m` containing the 4 has no right child (since it is maximal in its subtree)
  - `m` does have a left child, which we attach to `m`'s parent (the node containing the 2) instead



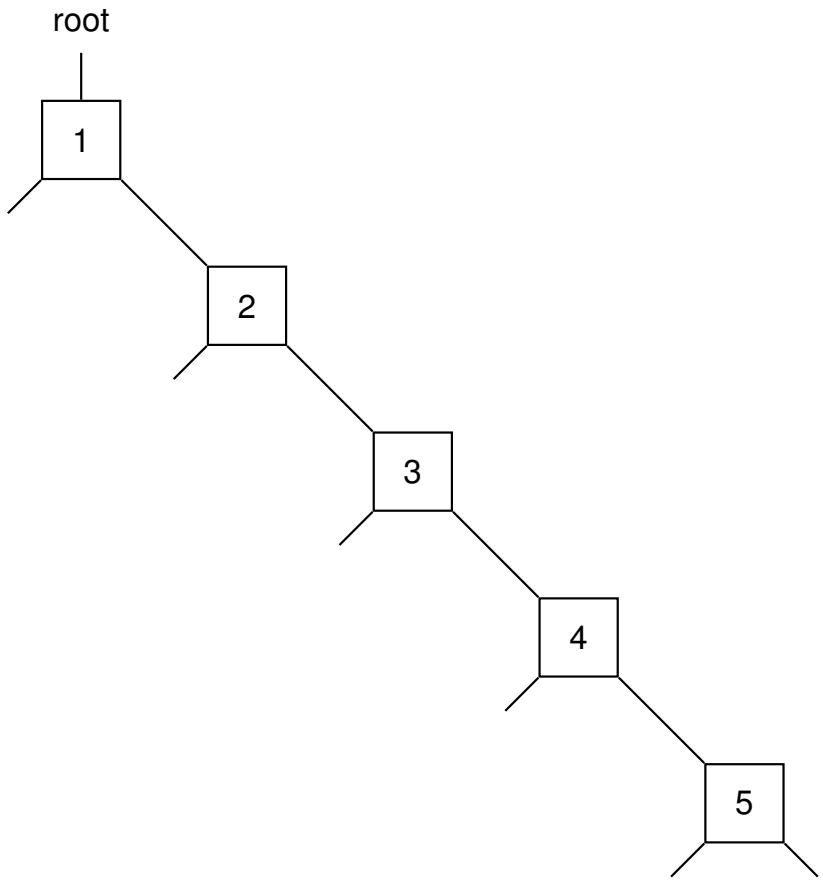
Binary search trees: deletion

So the steps for removing a value are:

- Find the node `n` containing the value
  - If there is none, we are done
- If `n` is a leaf, just remove it
- If `n` has just one (non-`null`) child,
  - Attach that child to `n`'s parent instead
  - Remove `n`
- Otherwise:
  - Find the node `m` containing the left subtree's maximal element
    - Start at `n`'s left child
    - Go down to the right as far as possible
  - Use that element to replace the value in `n`
  - Attach `m`'s left child (if non-`null`) to `m`'s parent
  - Remove `m`

Binary search trees: performance

- The operations so far do not guarantee good performance
- Suppose we insert $1, 2, 3, 4, 5$ into an empty tree
- The tree now is essentially a list



Binary search trees: performance

- We just saw that a tree can be very unbalanced
- This makes search, insertion, deletion $O(N)$ in the worst case
- Next time we will see how to get them all down to $O(\log N)$ by **re-balancing** the tree
- For now though, the tutorial exercise will be to implement and test the basic version