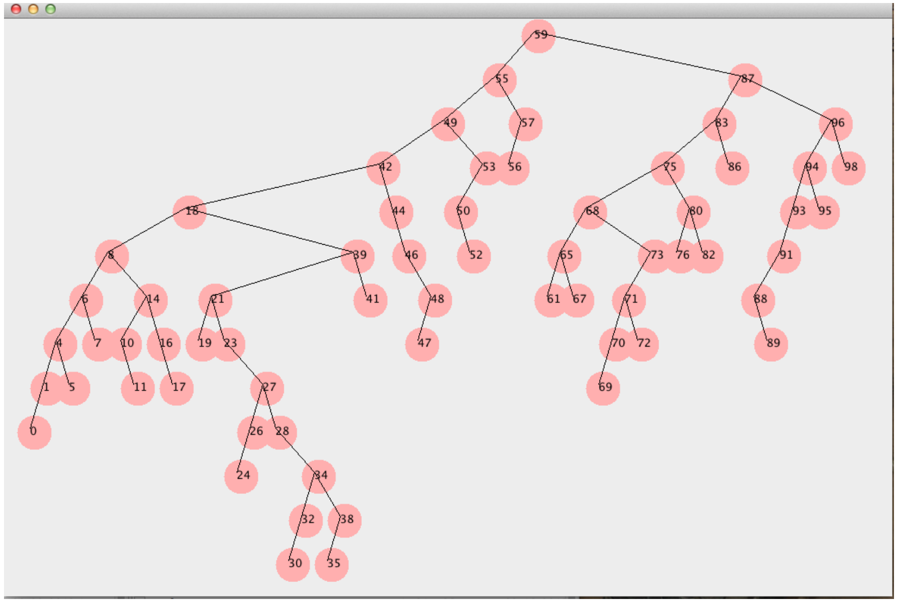
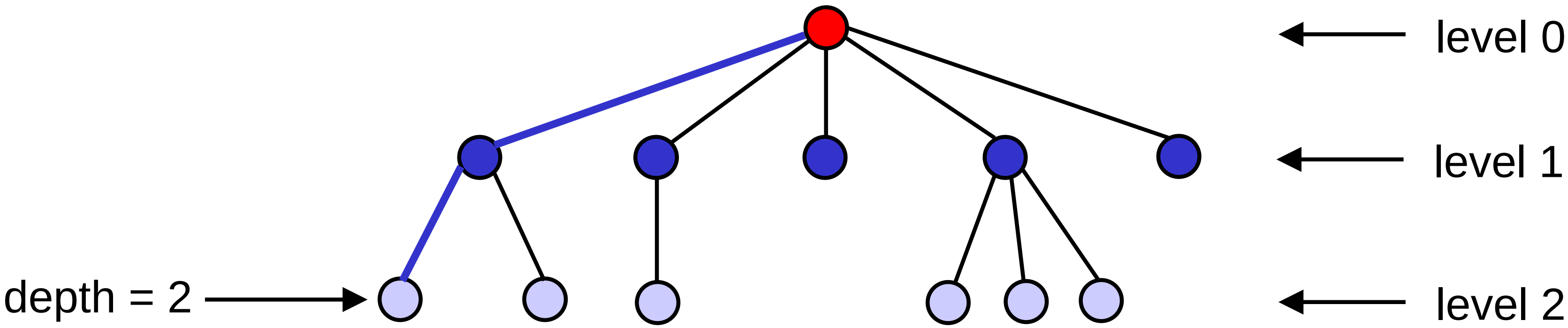


Binary Search Trees

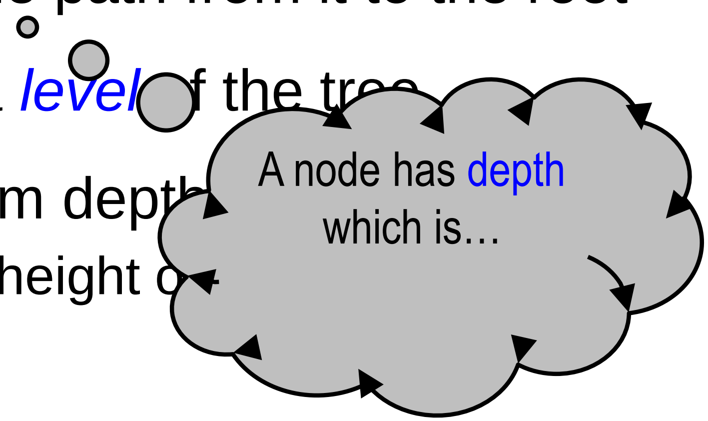
Computer Science 112
Boston University

Christine Papadakis-Kanaris

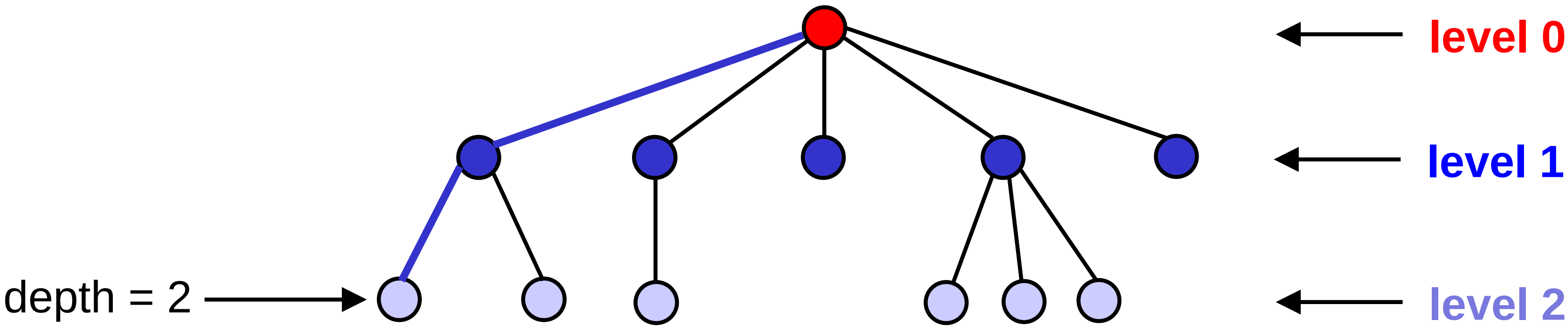
Recall: Path, Depth, Level, and Height

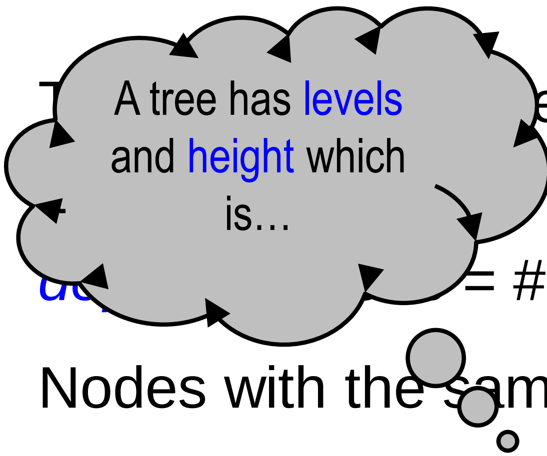


- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree
- The *height* of a tree is the maximum depth
    - example: the tree above has a height o



Recall: Path, Depth, Level, and Height

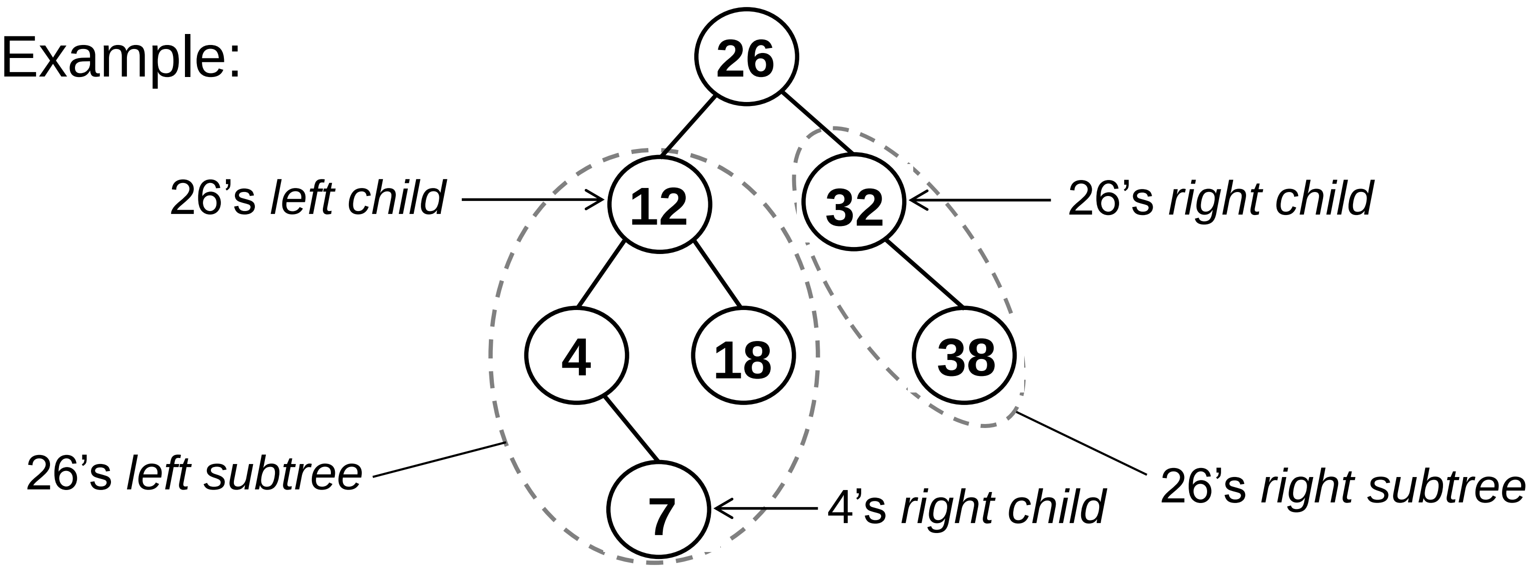




- T e *path* (one sequence of edges) connecting ot.
- *de* = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the maximum depth of its nodes.
    - example: the tree above has a height of 2

Recall: **Binary** Trees

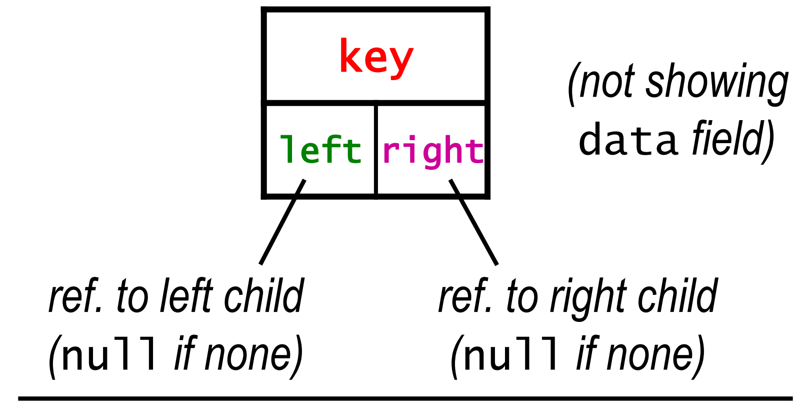
- In a *binary tree*, nodes have *at most two* children.
    - distinguish between them using the direction *left* or *right*
- Example:

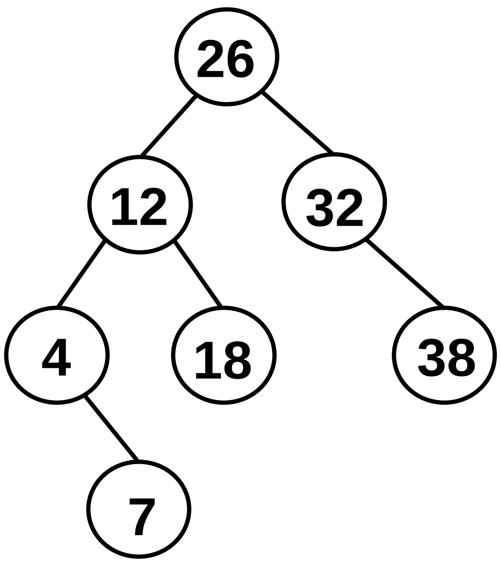


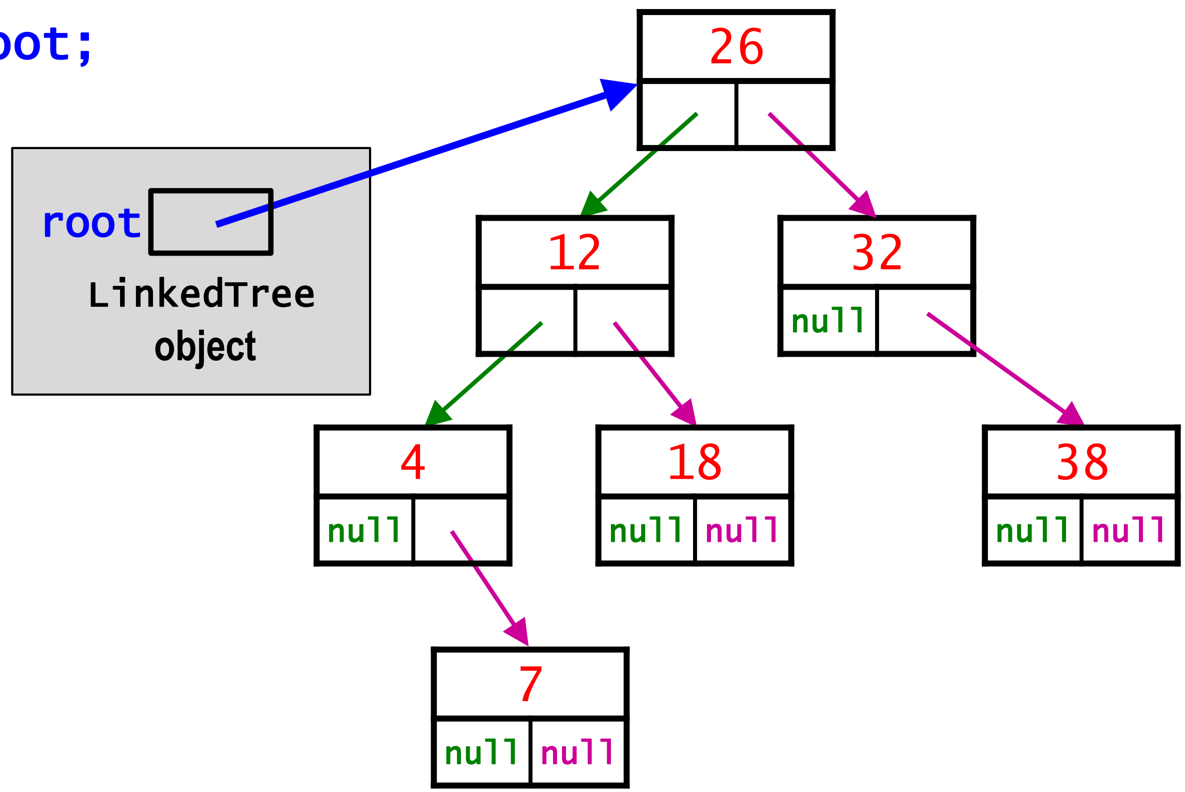
- Recursive definition: a binary tree is either:
    1) empty, or
    2) a node (the root of the tree) that has:
        - one or more pieces of data (the key, and possibly others)
        - a *left subtree*, which is itself a binary tree
        - a *right subtree*, which is itself a binary tree

Recall: A **Binary Tree** Using Linked Nodes

```
public class LinkedTree {
    private class Node {
        private int key;
        private LLList data;
        private Node left;
        private Node right;
        …
    }

    private Node root;
    …
}
```

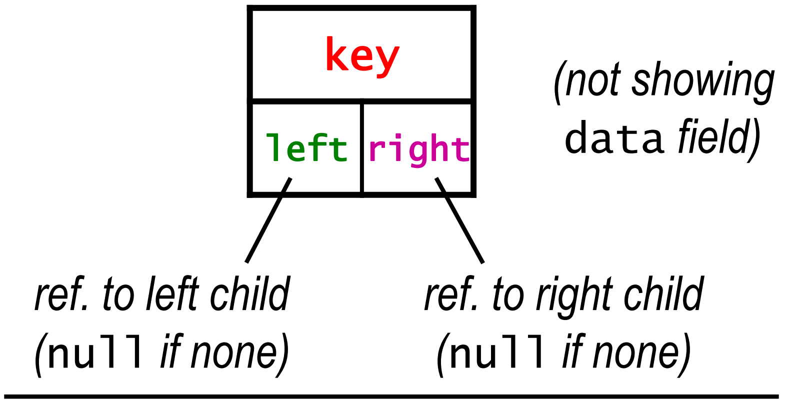


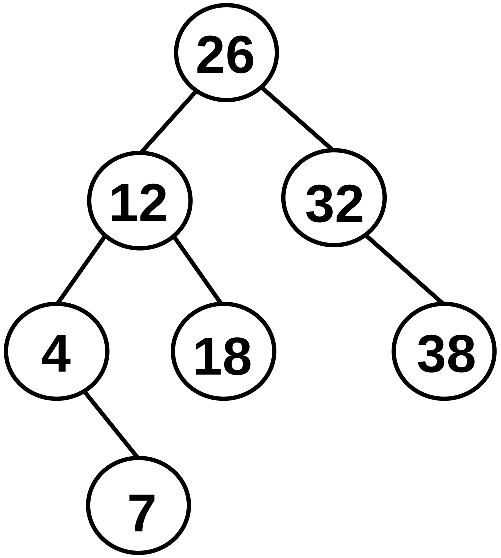


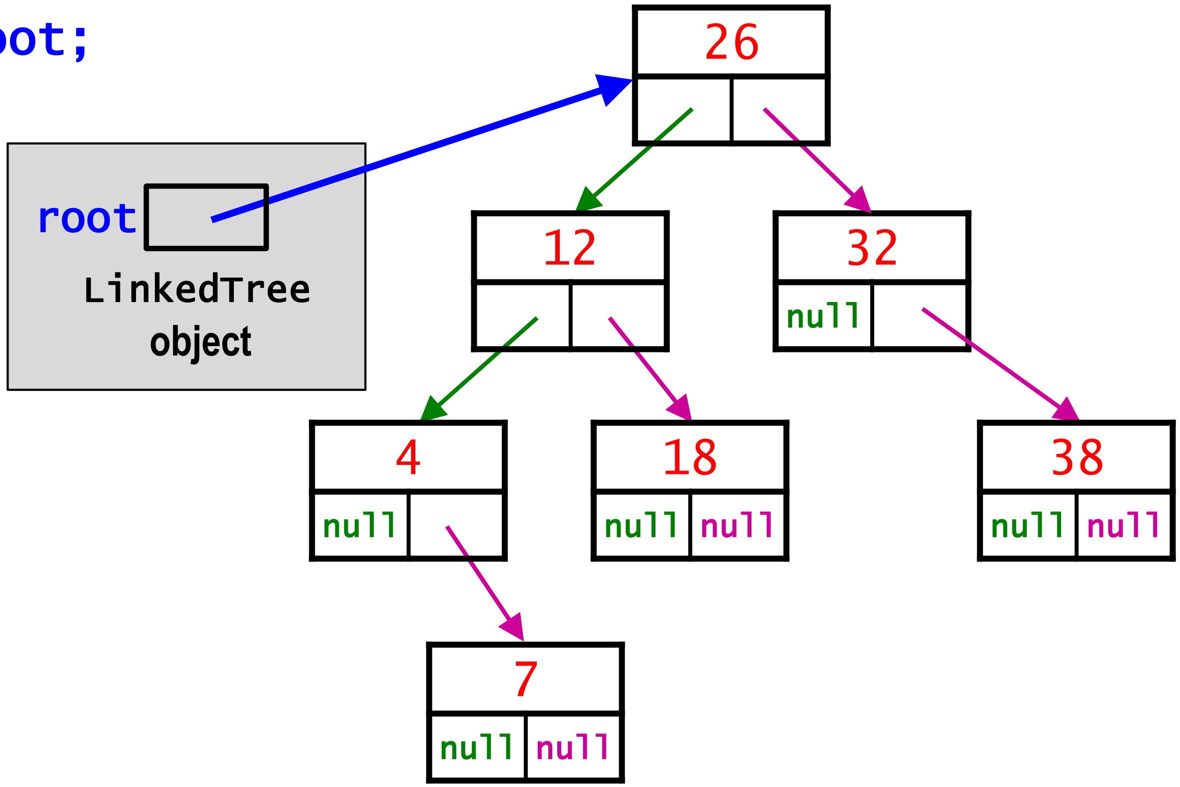


Recall: A **Binary Tree** Using Linked Nodes

```
public class LinkedTree {
    private class Node {
        private int key;
        private LLList data;
        private Node left;
        private Node right;
        …
    }

    private Node root;
    …
}
```



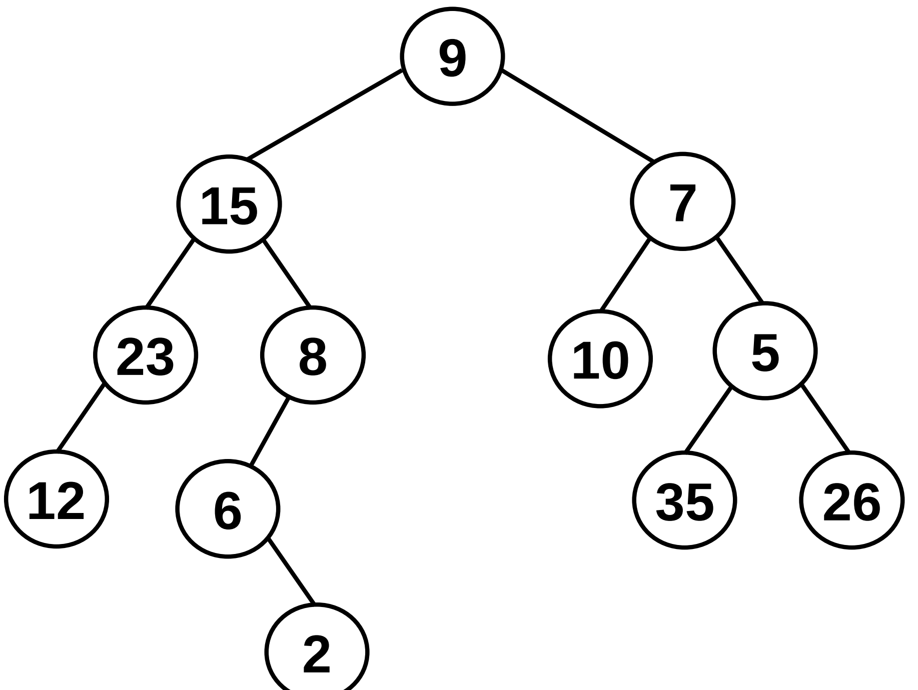




Recall: Tree Traversals

preorder:	root, left subtree, right subtree
postorder:	left subtree, right subtree, root
inorder:	left subtree, root, right subtree
level-order:	top to bottom, left to right

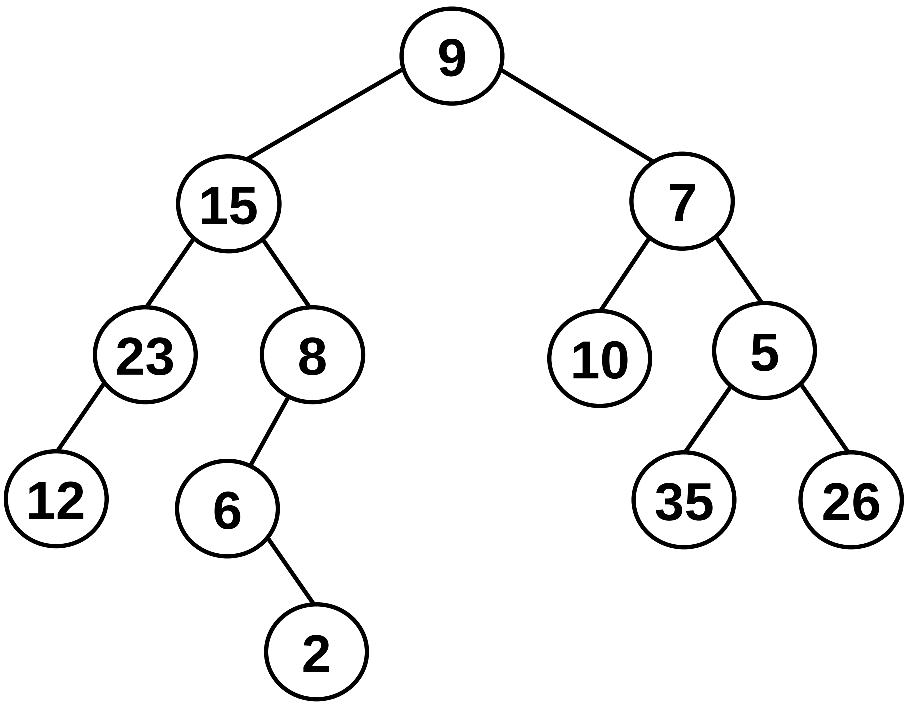
- Perform each type of traversal on the tree below:



Recall: Tree Traversals

preorder:	root, left subtree, right subtree
postorder:	left subtree, right subtree, root
inorder:	left subtree, root, right subtree
level-order:	top to bottom, left to right

- Perform each type of traversal on the tree below:



pre: 9 15 23 12 8 6 2 7 10 5 35 26

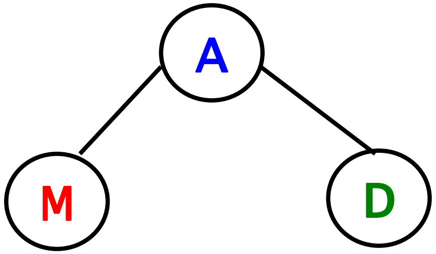
post: 12 23 2 6 8 15 10 35 26 5 7 9

in: 12 23 15 6 2 8 9 10 7 35 5 26

level: 9 15 7 23 8 10 5 12 6 35 26 2

Tree Traversal Puzzle

- preorder traversal: **A** **M** P K L **D** H T
- inorder traversal: **P M L K A H T D**
  left subtree *right subtree*
- Draw the tree!



- What's one fact that we can easily determine from one of the traversals?
  A is the root of the entire tree
  because it is visited first by a preorder traversal.
- How could we determine the nodes in each of the root's subtrees?
- *What are the roots of each subtree?*

A) `P` and `H`

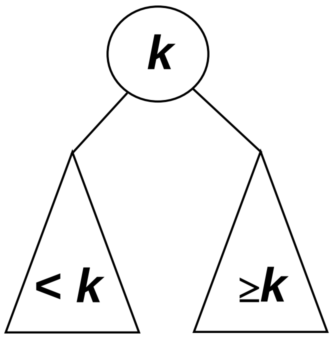
B) `P` and `D`

C) `M` and `H`

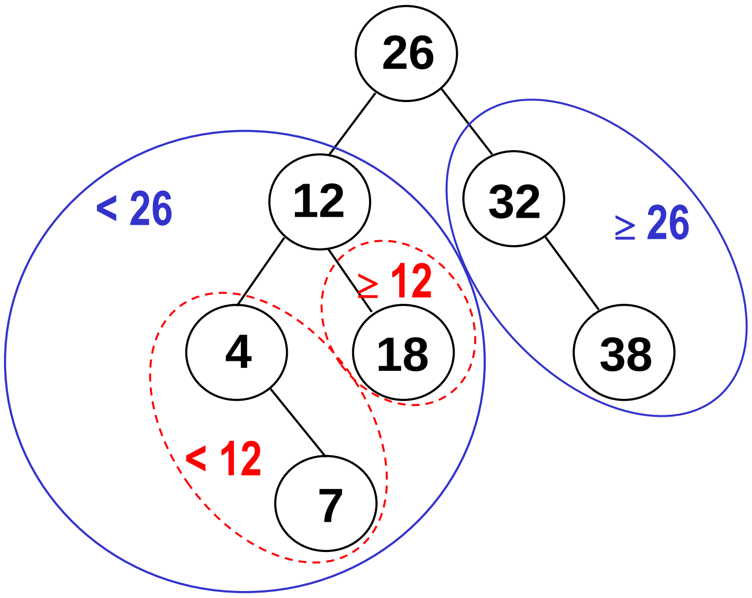
D) **`M` and `D`**

Binary *Search* Trees

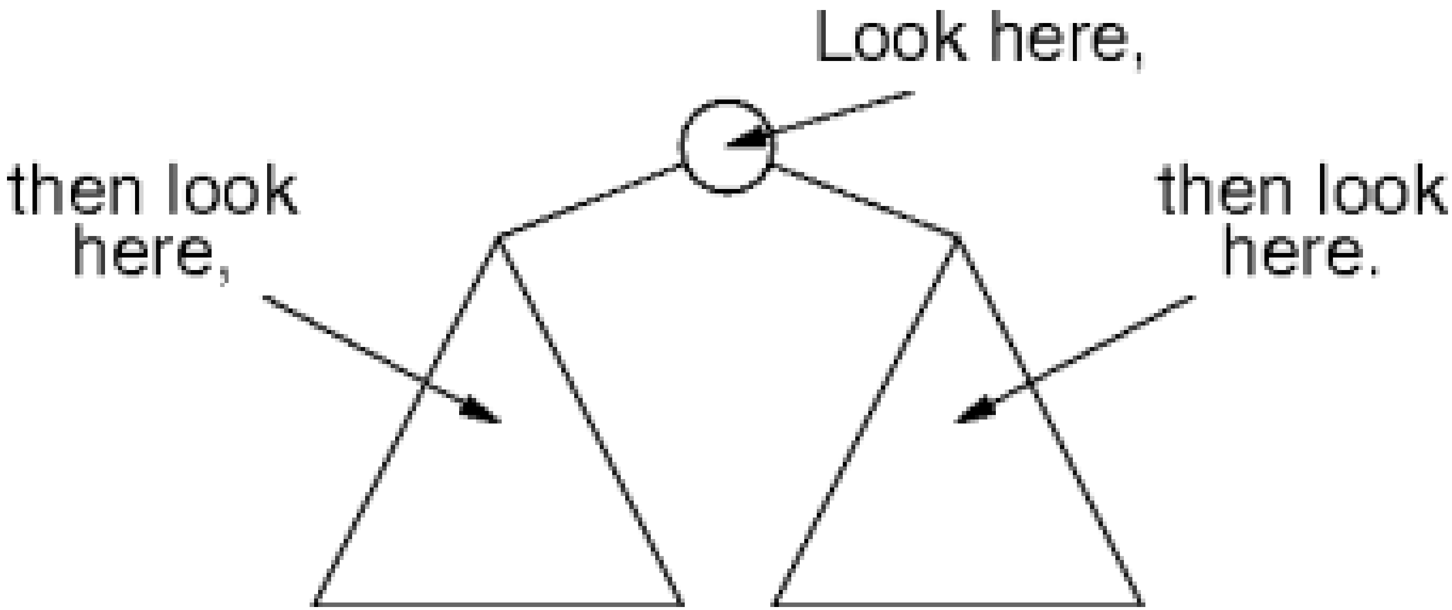
- Search-tree property: for each node *k* (*k* is the key):
  - all nodes in *k*'s left subtree are < *k*
  - all nodes in *k*'s right subtree are >= *k*



- Our earlier binary-tree example is a search tree:



The search



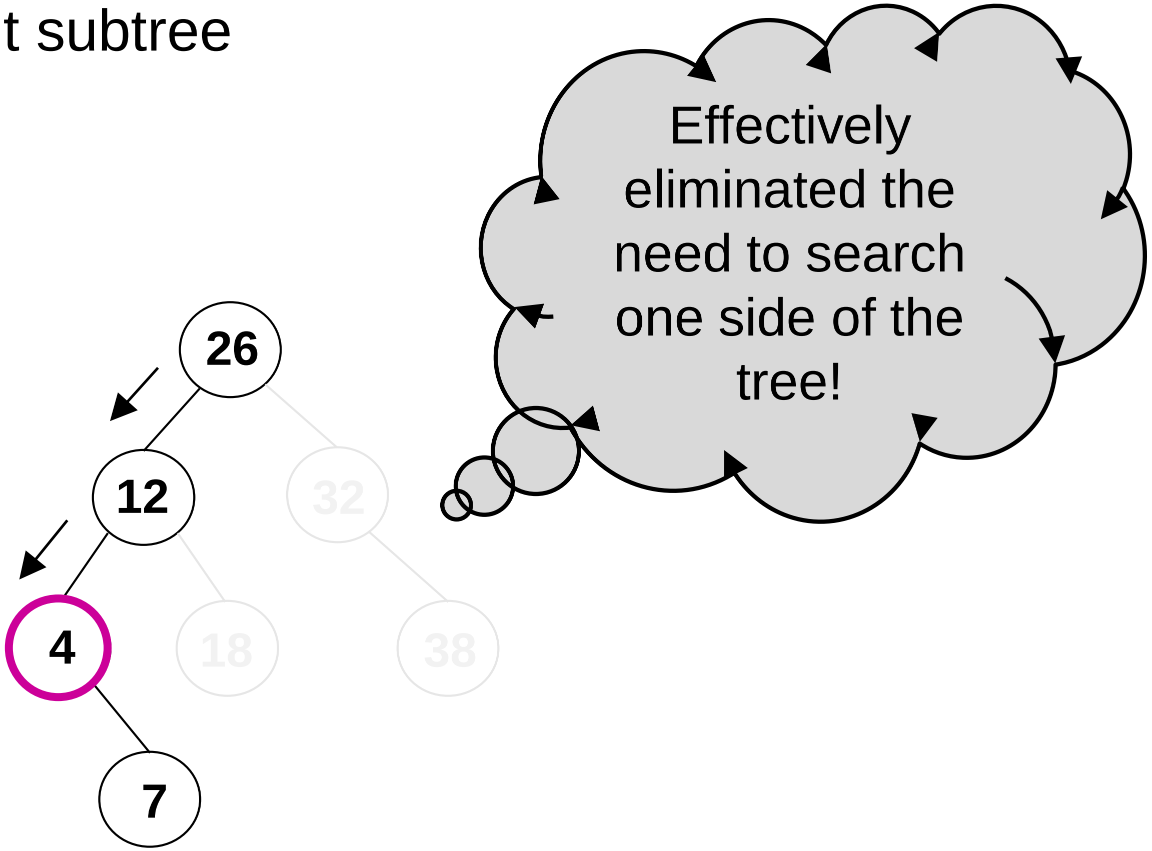
Searching for an Item in a Binary Search Tree

another look

- Algorithm for searching for an item with a key *k*:

  if *k* == the root node's key, you're done
  else if *k* < the root node's key, search the left subtree
  else search the right subtree

- Example: search for 7



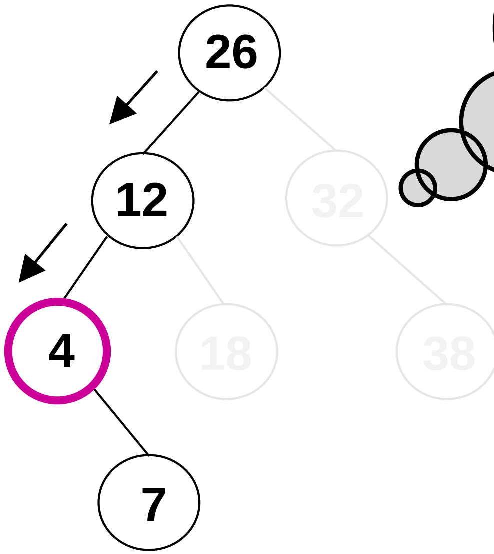
Searching for an Item in a Binary Search Tree

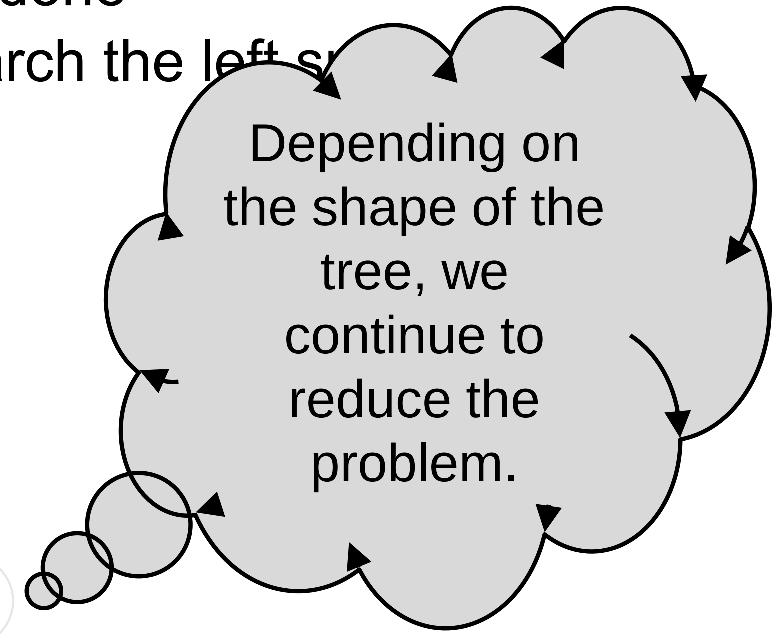
another look

- Algorithm for searching for an item with a key *k*:

  if *k* == the root node's key, you're done
  else if *k* < the root node's key, search the left s
  else search the right subtree

- Example: search for 7





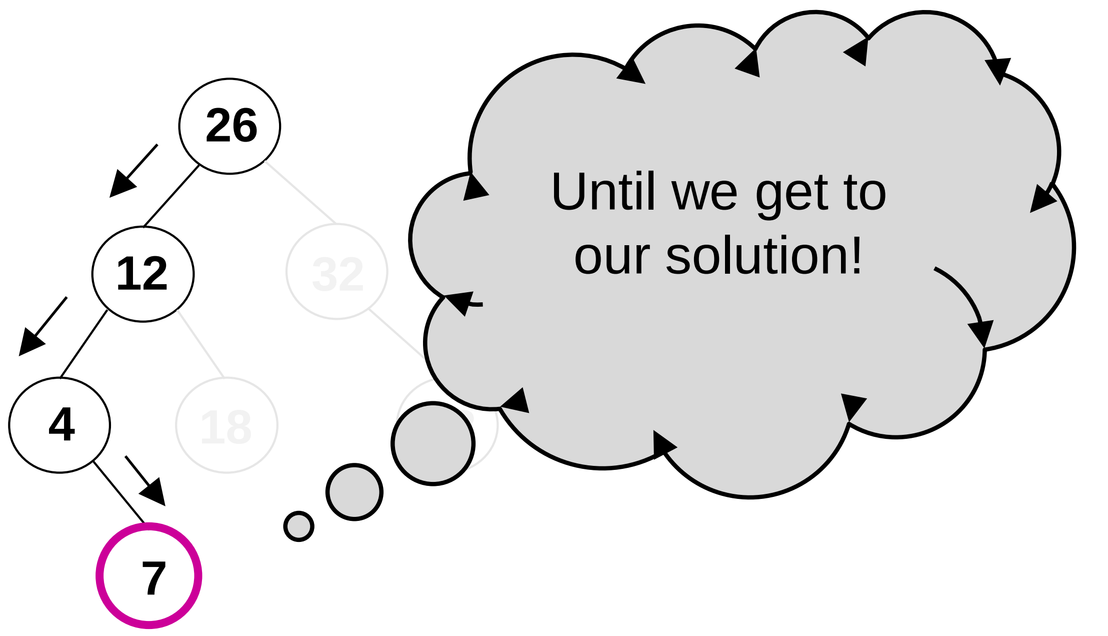
Searching for an Item in a Binary Search Tree

another look

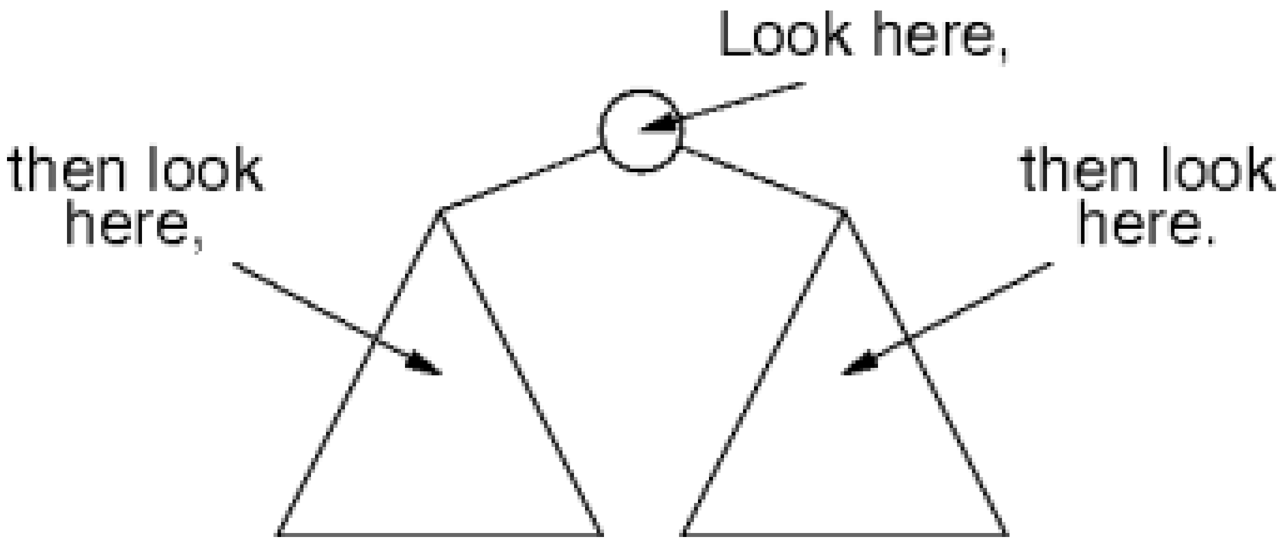
- Algorithm for searching for an item with a key *k*:

  if *k* == the root node's key, you're done
  else if *k* < the root node's key, search the left subtree
  else search the right subtree

- Example: search for 7



Insert algorithm



Then insert!

Inserting an Item in a Binary Search Tree

- We need a method with this header

  `public void insert(int key, O`

  that we can use to add a new (k

Note that inserting two different key .. value pairs would result in two new nodes being added to the tree!

- Example 1: a search tree cont
  - key = the student's ID numb
  - data = a string with the rest of the student record
  - we want to be able to writ client code that looks like this:

```
LinkedTree students = new LinkedTree();
students.insert(23, "Jill Jones,sophomore,comp sci");
students.insert(45, "Al Zhang,junior,english");
```

- Example 2: a search tree containing scrabble words
  - key = a scrabble score (an integer)
  - data = a word with that scrabble score

```
LinkedTree tree = new LinkedTree();
tree.insert(4, "lost");
```

Inserting an Item in a Binary Search Tree

- We need a method with this header

```
public void insert(int key, Object data)
```

that we can use to add a new (key, data) pair to the tree.

- Example 1: a search tree containing student records
  - key = the student's ID number (an integer)
  - data = a string with the rest of the student record
  - we want to be able to

```
LinkedTree studen
students.insert(2
students.insert(4
```

How would the tree change if the following insert was performed?

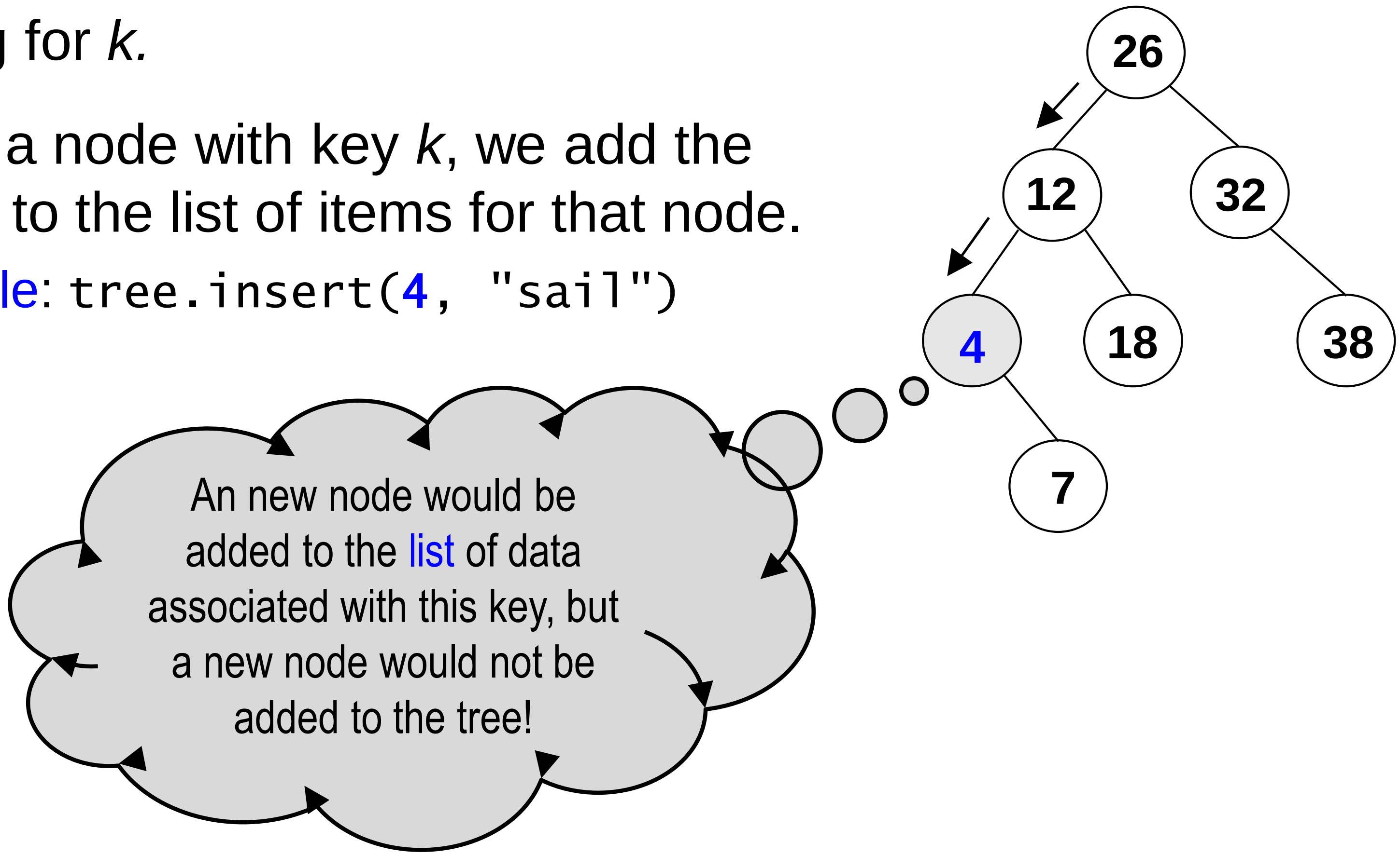
```
tree.insert(4, “sail");
```

- Example 2: a search tre
  - key = a scrabble score (an integer)
  - data = a word with that scrabble score

```
LinkedTree tree = new LinkedTree();
tree.insert(4, "lost");
```

Inserting an Item in a Binary Search Tree

- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k.*
- If we find a node with key *k*, we add the data item to the list of items for that node.
  - example: `tree.insert(4, "sail")`

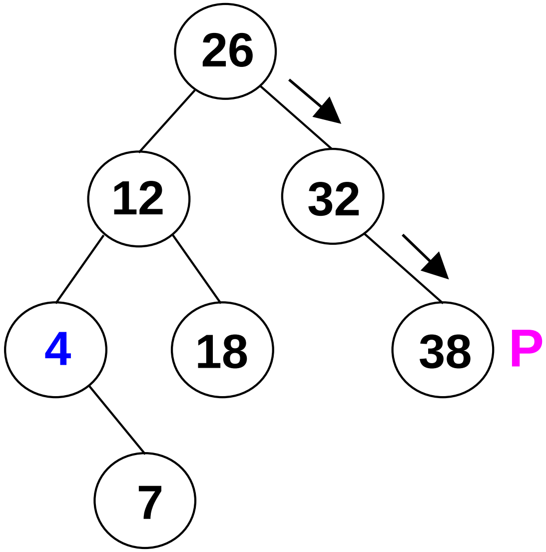


Inserting an Item in a Binary Search Tree

- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k.*
- If we find a node with key *k*, we add the data item to the list of items for that node.
  - example: `tree.insert(4, "sail")`
- If we don't find it, the last node we encounter will be the parent **P** of the new node (see example at right).

example:

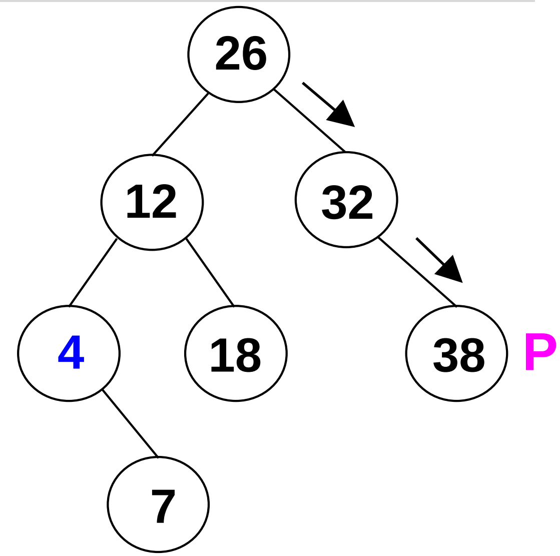
```
tree.insert(35,
  "photooxidizes")
```



Inserting an Item in a Binary Search Tree

- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k.*
- If we find a node with key *k*, we add the data item to the list of items for that node.
  - example: `tree.insert(4, "sail")`
- If we don't find it, the last node we encounter will be the parent **P** of the new node (see example at right).
  - if *k* < P's key, make the new node P's left child
  - else make the node P's right child

example:
```
tree.insert(35,
  "photooxidizes")
```

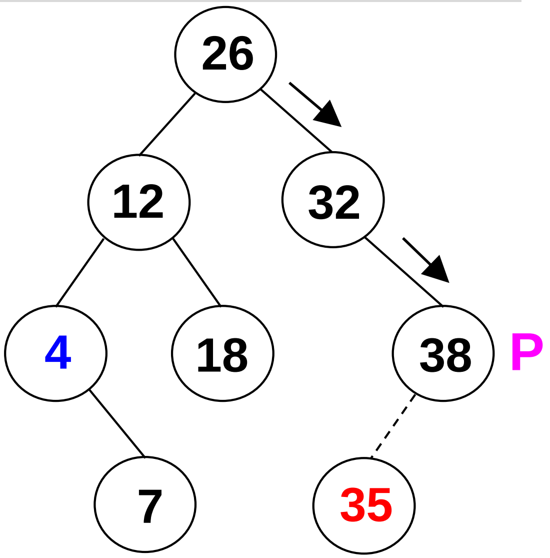


Inserting an Item in a Binary Search Tree

- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k.*
- If we find a node with key *k*, we add the data item to the list of items for that node.
  - example: `tree.insert(4, "sail")`
- If we don't find it, the last node we encounter will be the parent **P** of the new node (see example at right).
  - if *k* < P's key, make the new node P's left child
  - else make the node P's right child

example:

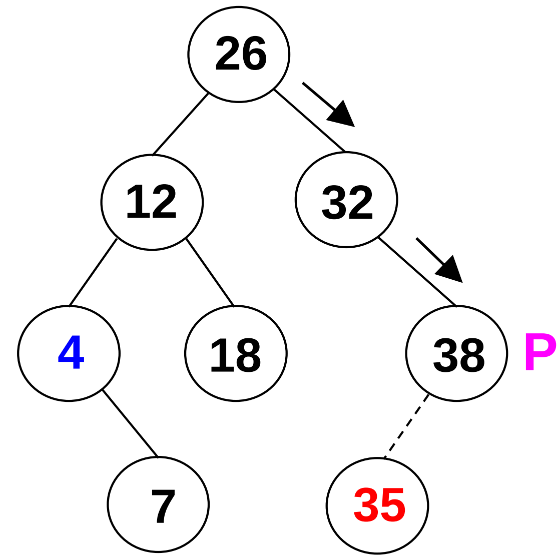
```
tree.insert(35,
  "photooxidizes")
```



Inserting an Item in a Binary Search Tree

- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k.*
- If we find a node with key *k*, we add the data item to the list of items for that node.
  - example: `tree.insert(4, "sail")`
- If we don't find it, the last node we encounter will be the parent **P** of the new node (see example at right).
  - if *k* < P's key, make the new node P's left child
  - else make the node P's right child
- *Special case:* if the tree is empty, make the new node the root of the tree.

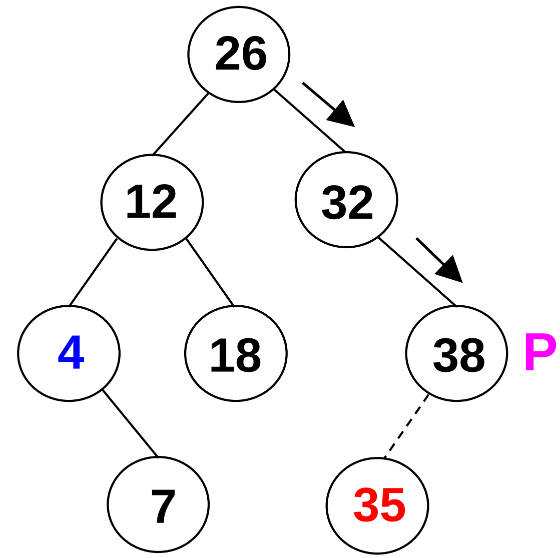
example:
```
tree.insert(35,
  "photooxidizes")
```



Inserting an Item in a Binary Search Tree

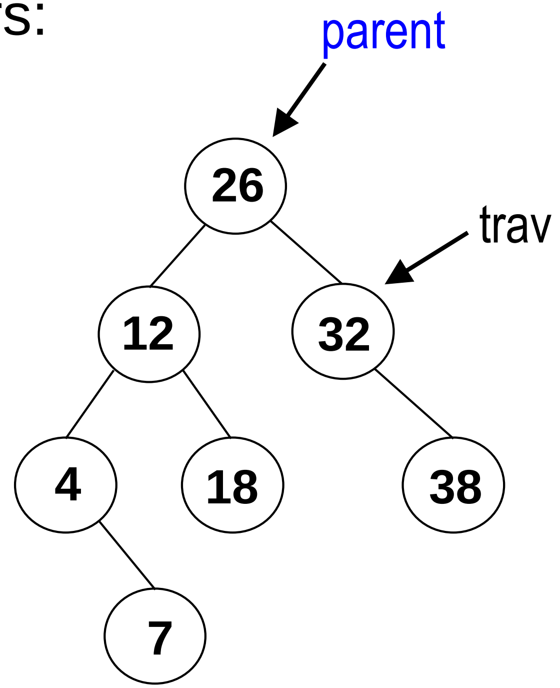
- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k.*
- If we find a node with key *k*, we add the data item to the list of items for that node.
  - example: `tree.insert(4, "sail")`
- If we don't find it, the last node we encounter will be the parent **P** of the new node (see example at right).
  - if *k* `<` P's key, make the new node P's left child
  - else make the node P's right child
- *Special case:* if the tree is empty, make the new node the root of the tree.
- ***Important:*** The resulting tree is still a search tree!

example:
```
tree.insert(35,
  "photooxidizes")
```



Implementing Binary-Tree Insertion

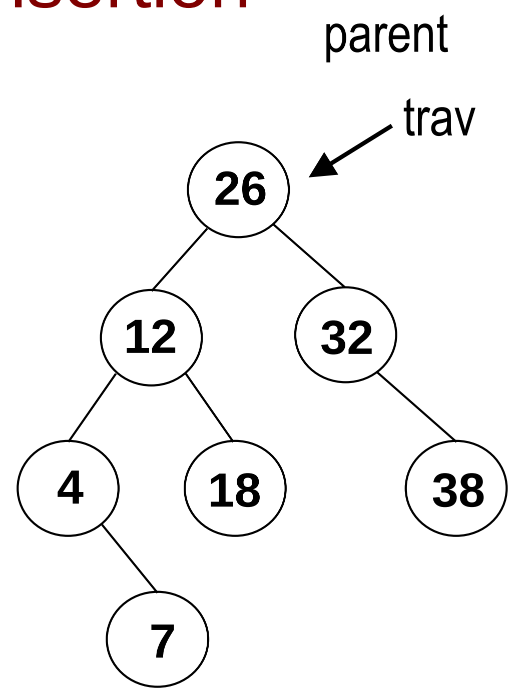
- We'll implement part of the `insert()` method together.
- We'll use iteration rather than recursion.
- Our method will use two references/pointers:
  - `trav`: performs the traversal down to the point of insertion
  - `parent`: stays one behind `trav`
    - like the *`trail`* reference that we sometimes use when traversing a linked list



Implementing Binary-Tree Insertion

insert 35:

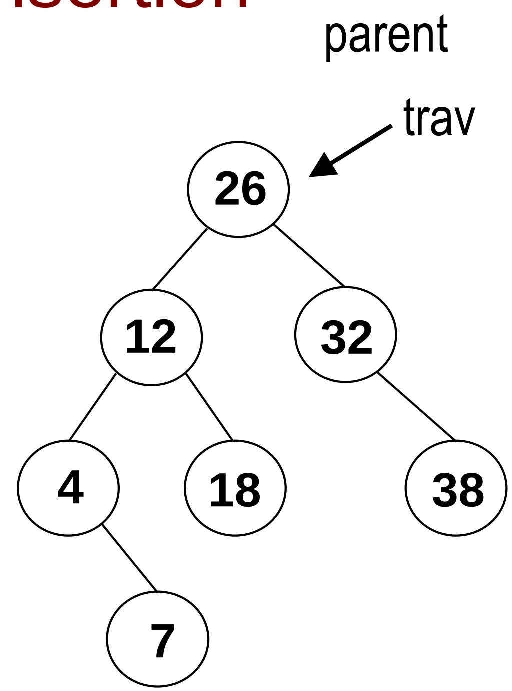
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }



    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return; // nothing more to do!
        }







    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```

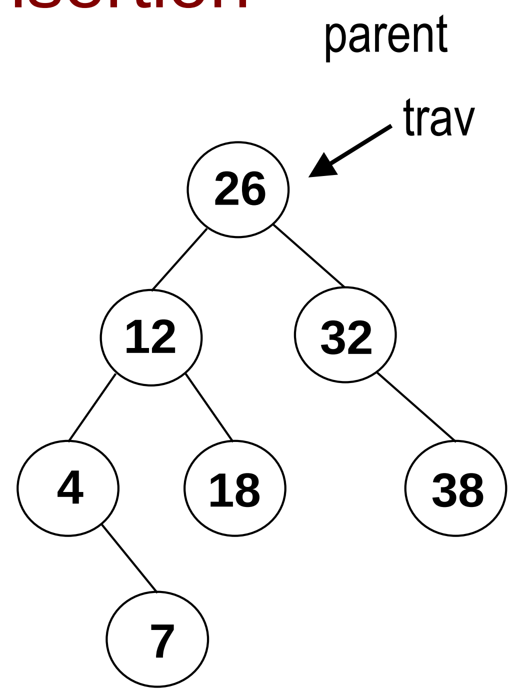
insert 35:



Implementing Binary-Tree Insertion

```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return; // nothing more to do!
        }







    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```

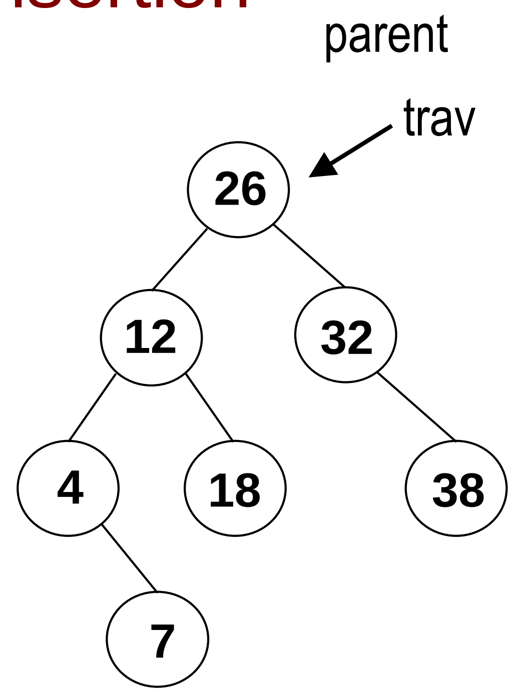
insert 35:



Implementing Binary-Tree Insertion

```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        // update references. Which first?





    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```

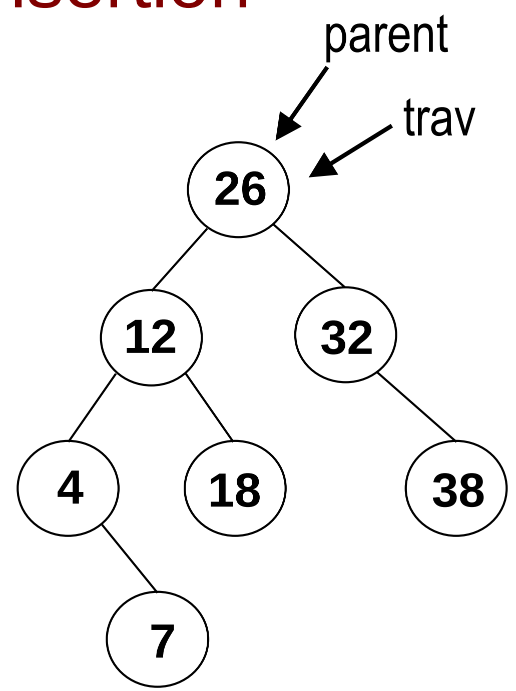
insert 35:



Implementing Binary-Tree Insertion

```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;





    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```

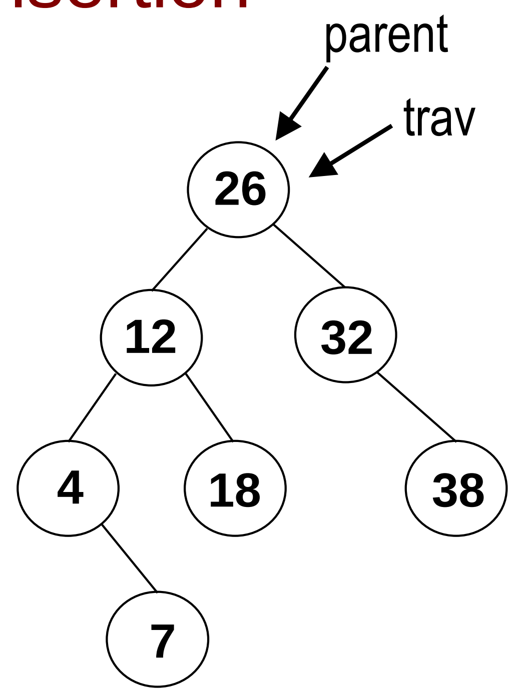
insert 35:



Implementing Binary-Tree Insertion

insert 35:

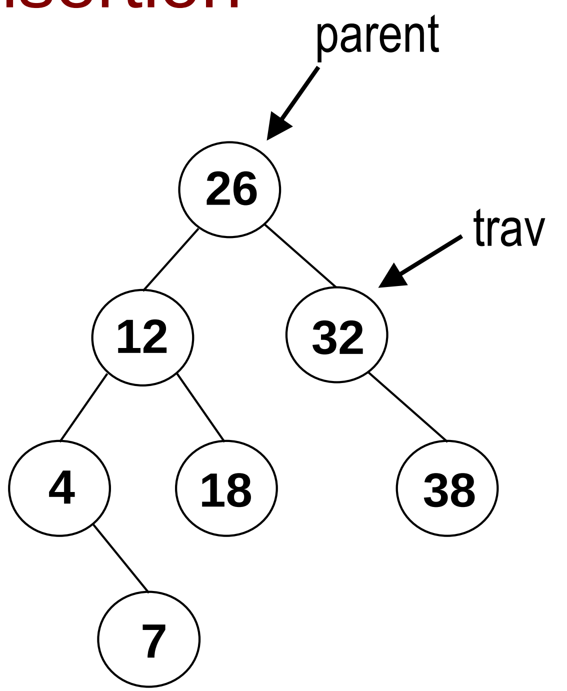
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

insert 35:

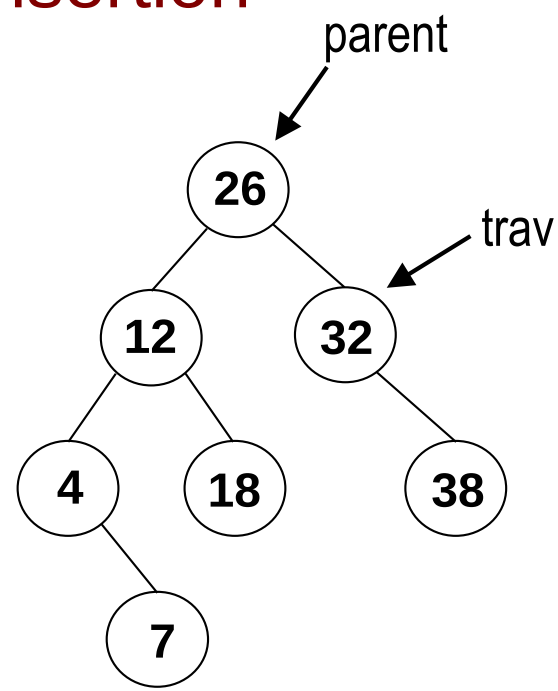
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

insert 35:

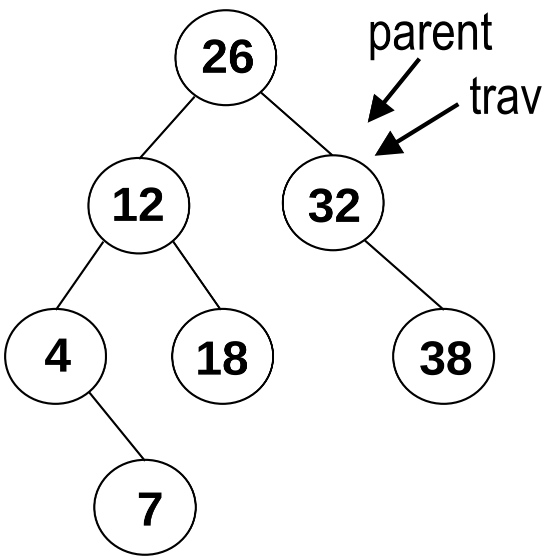
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

insert 35:

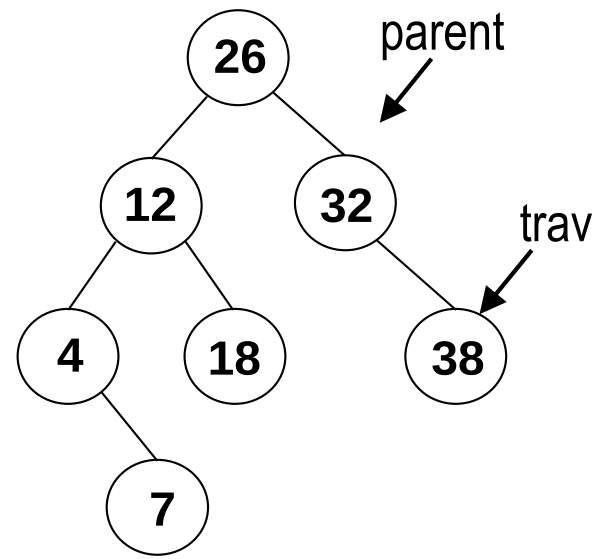
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

insert 35:

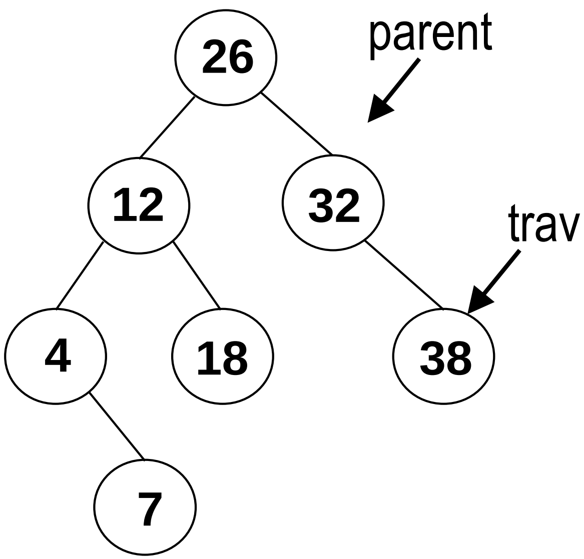
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

insert 35:

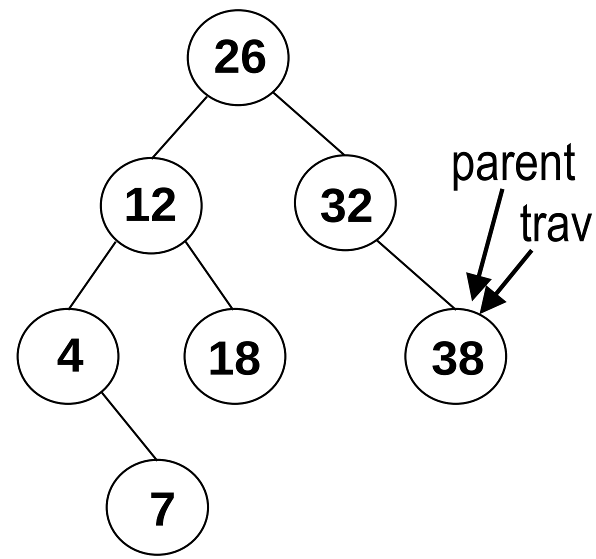
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

insert 35:

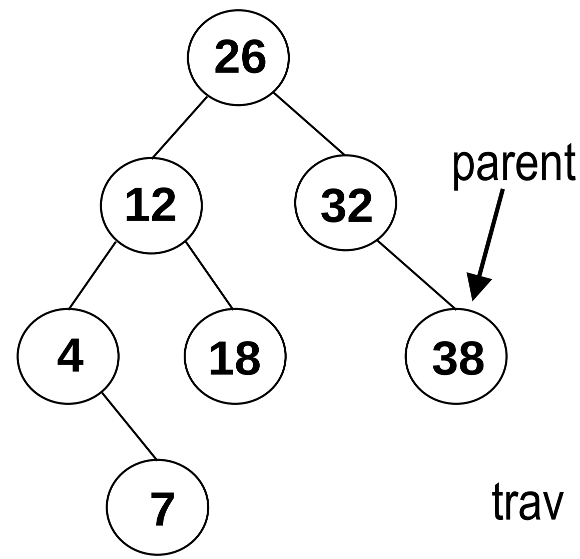
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



insert 35:

Implementing Binary-Tree Insertion

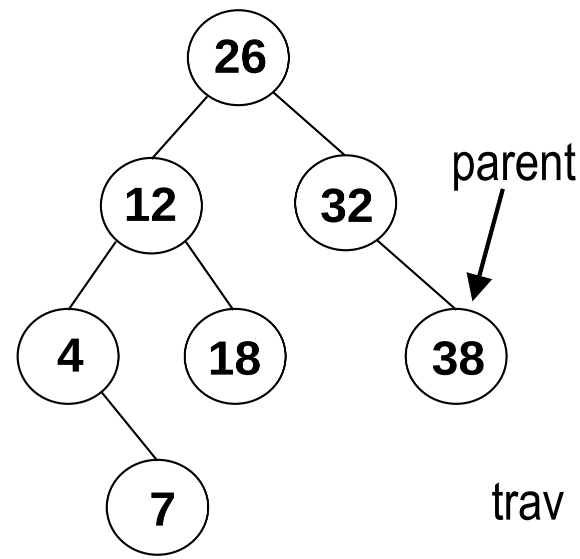
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

insert 35:

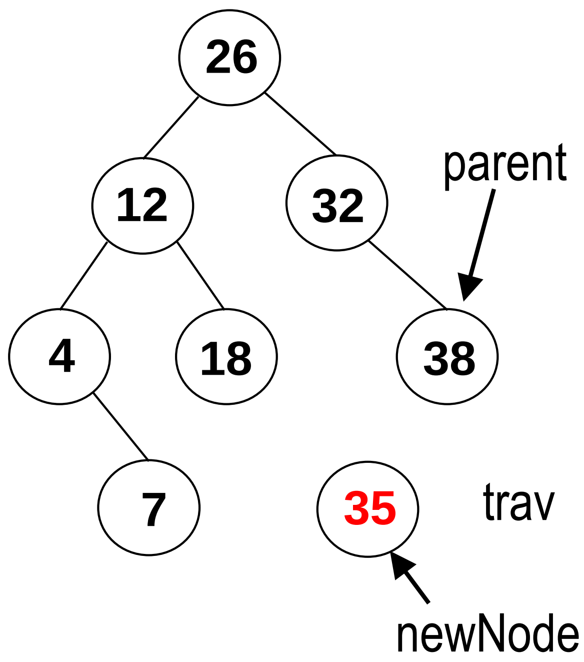
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Implementing Binary-Tree Insertion

insert 35:

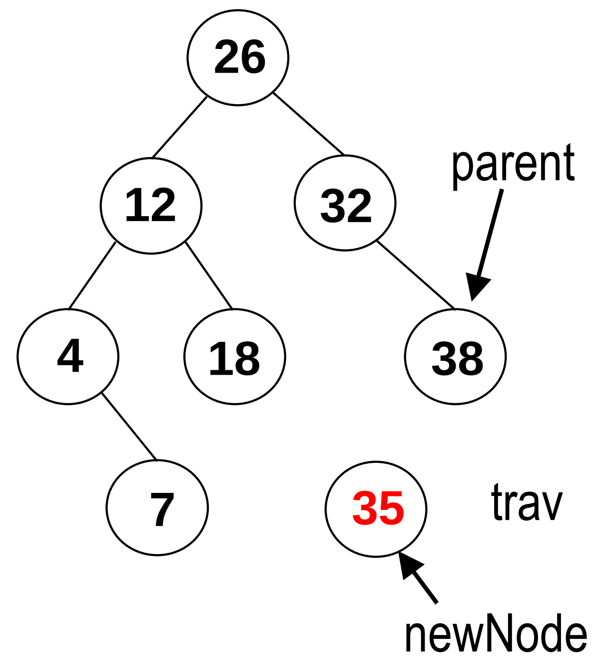
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



insert 35:

Implementing Binary-Tree Insertion

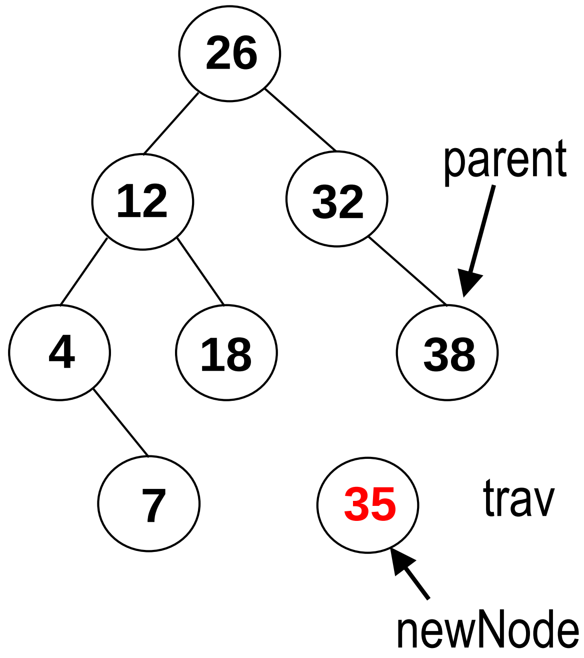
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



insert 35:

Implementing Binary-Tree Insertion

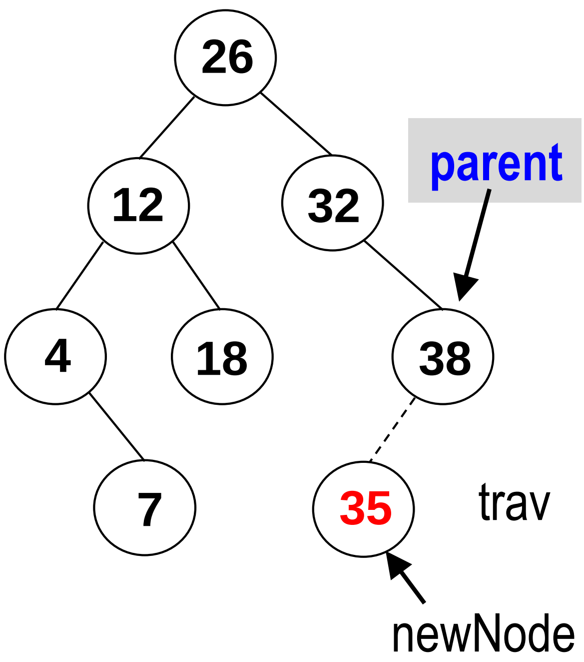
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



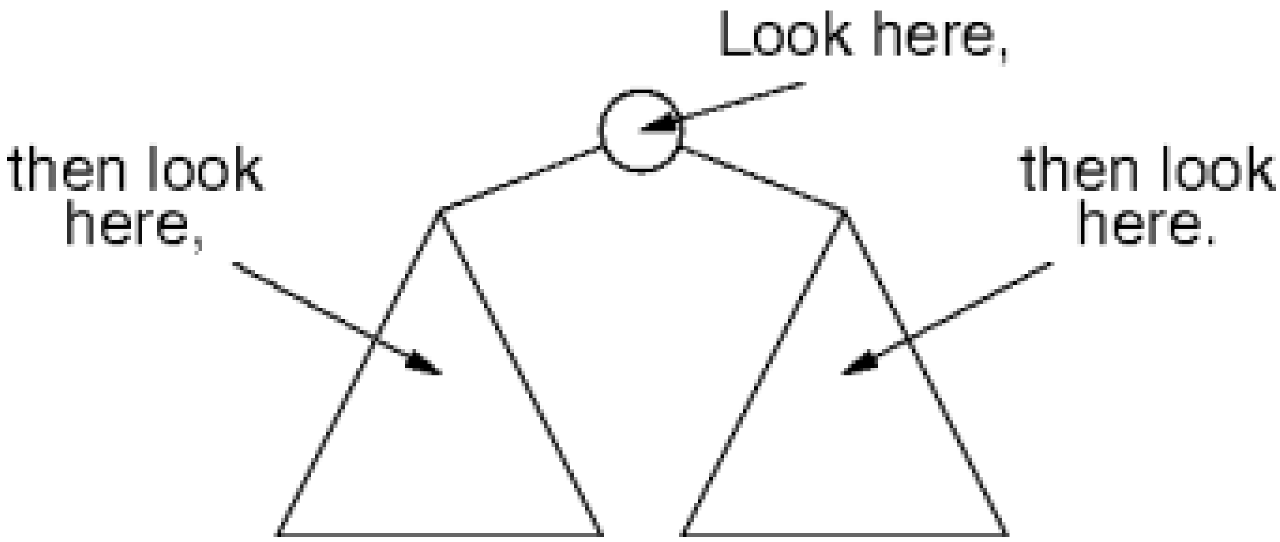
insert 35:

Implementing Binary-Tree Insertion

```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```



Delete algorithm

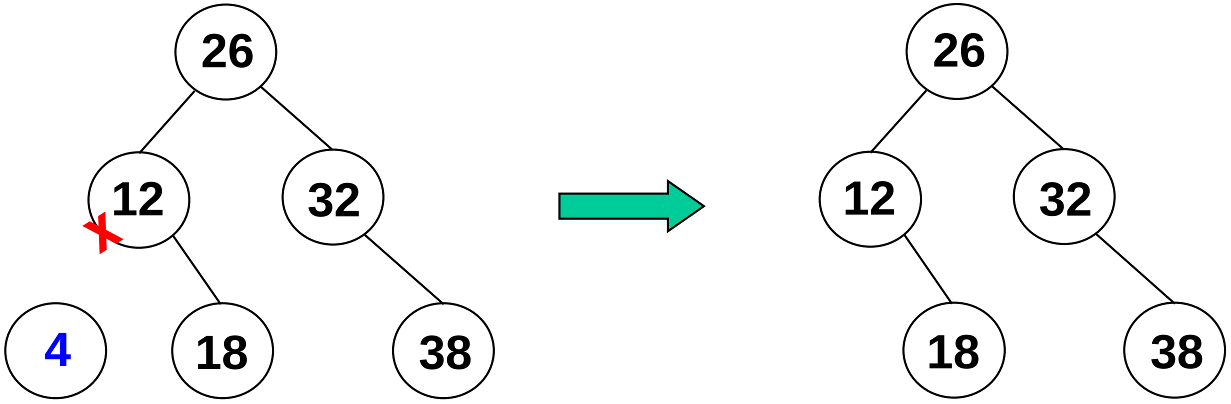


Then delete!

Deleting Items from a Binary Search Tree

- Three cases for deleting a node *x*
- **Case 1:** *x* has no children.
  Remove *x* from the tree by setting its parent's reference to null.

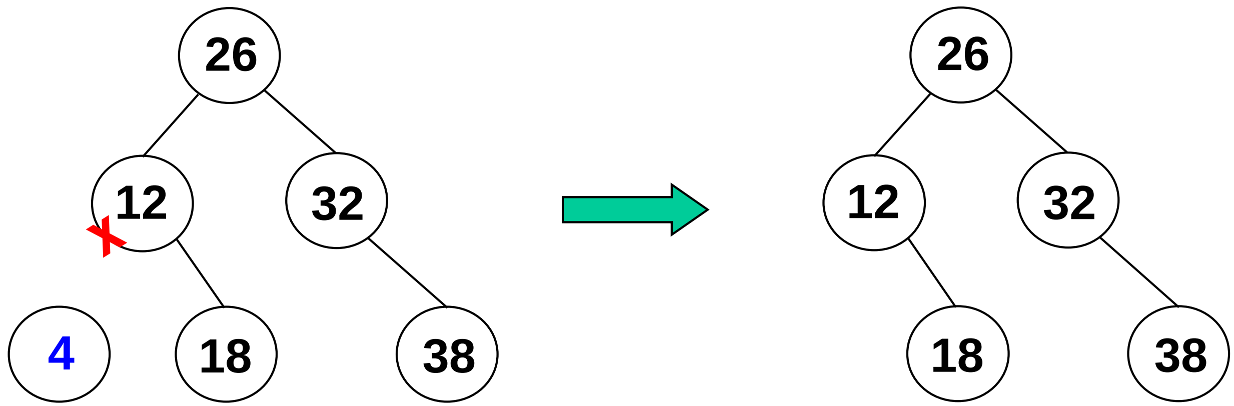
ex: delete 4



Deleting Items from a Binary Search Tree

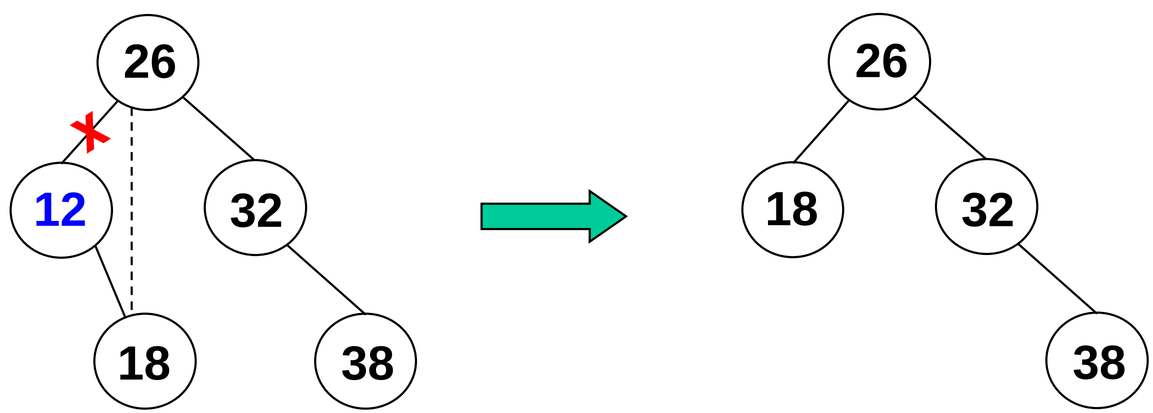
- Three cases for deleting a node *x*
- **Case 1:** *x* has no children.
  Remove *x* from the tree by setting its parent's reference to null.

ex: delete 4



- **Case 2:** *x* has one child.
  Take the parent's reference to *x* and make it refer to *x*'s child.

ex: delete 12



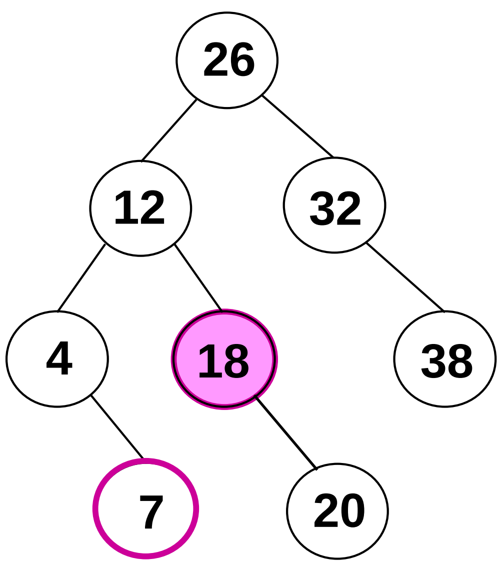
Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** *x* has two children
  - we can't give both children to the parent. why?

    both of *x'*s children are either:

    - less than *x*'s parent, but they can't both be its left child
    - greater than *x*'s parent, but they can't both be its right child
  - instead, we leave *x's* node where it is, and we replace its contents with those from another node
    - the replacement must maintain the search-tree inequalities

ex:
delete 12



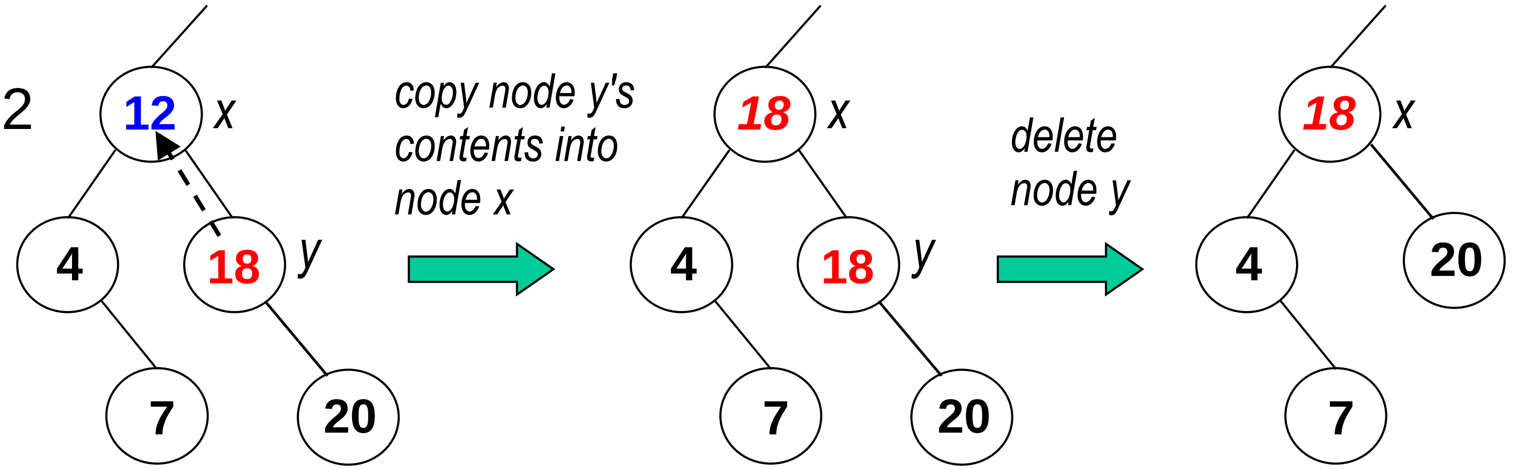
two options:

- largest in left subtree
  - everything else in left subtree is < it
- *smallest in right subtree*
  - everything else in right subtree is > it

Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** *x* has two children (continued):
  - replace *x*'s contents with those from the smallest node in *x*'s right subtree—call it *y*
  - we then delete *y*
    - it will either be a leaf node or will have one right child. why? if it had a left child, it wouldn't be the smallest in the subtree!
    - thus, we can delete it using case 1 or 2

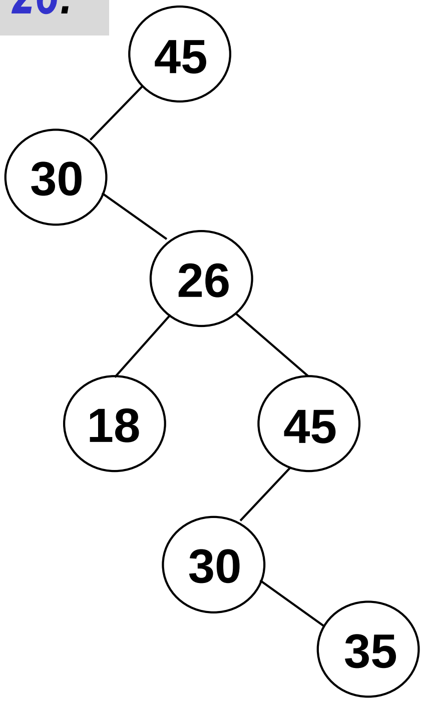
ex:
delete 12



Implementing Deletion

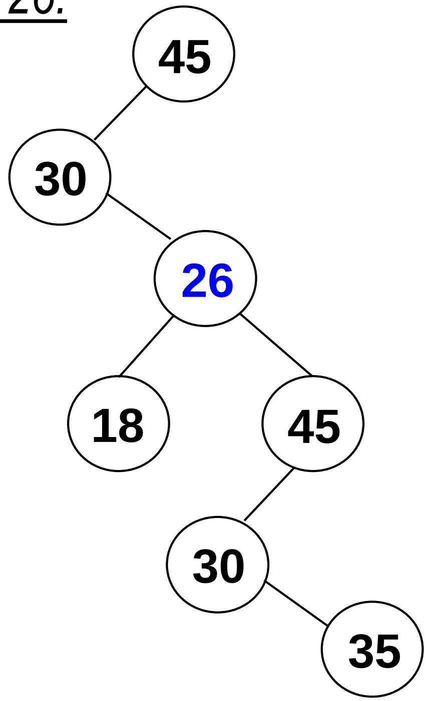
```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```

delete 26:



Implementing Deletion

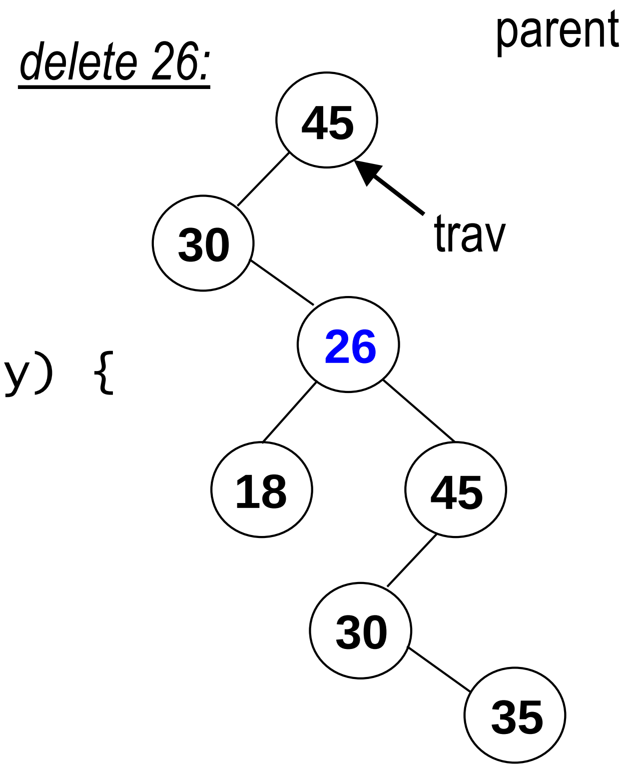
delete 26:




```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```

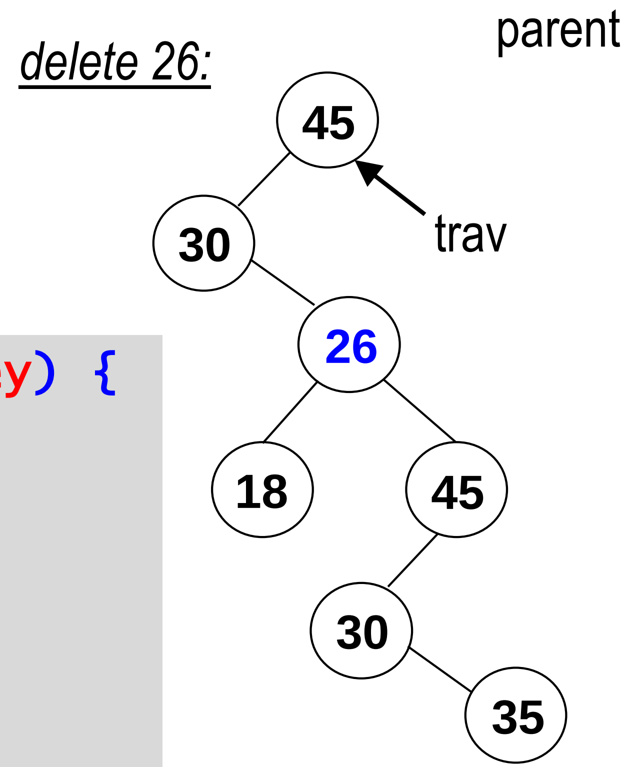
Implementing Deletion

```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```



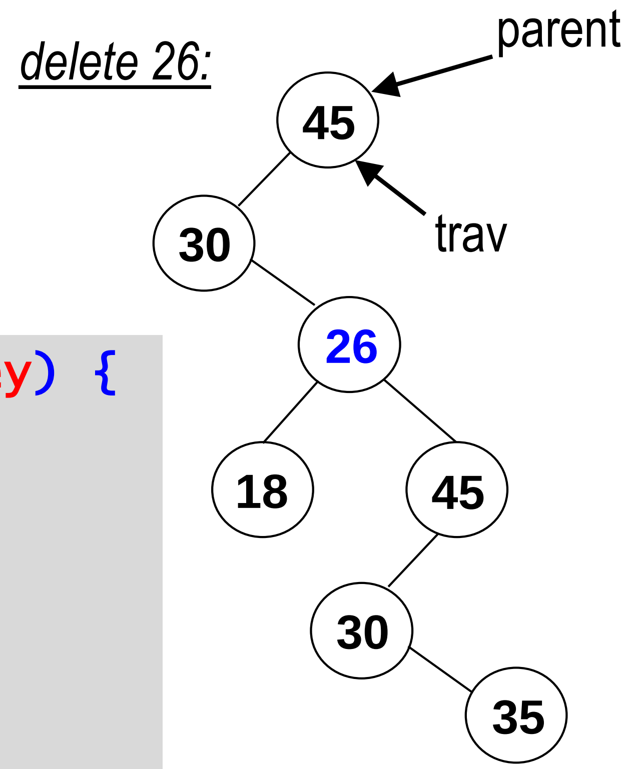
Implementing Deletion

```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    // Delete the node (if any) and return the removed items.
    if (trav == null) {    // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```



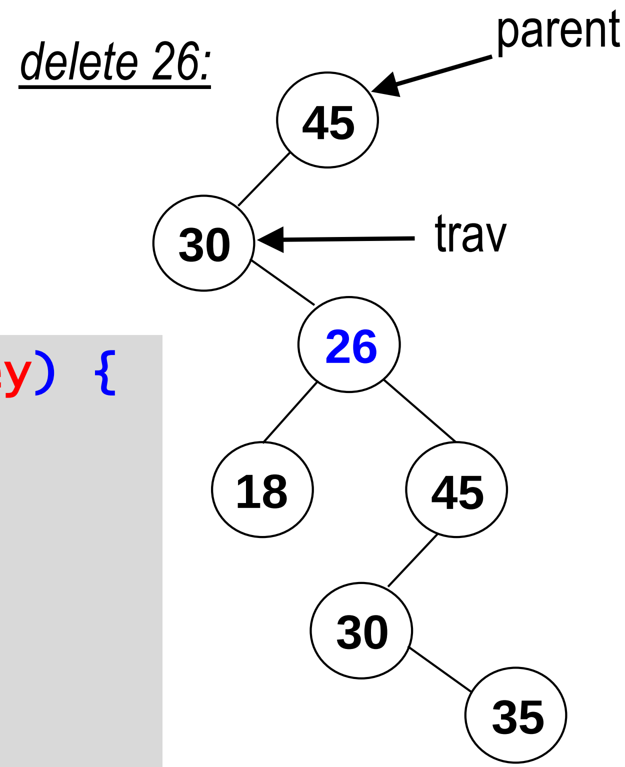
Implementing Deletion

```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    // Delete the node (if any) and return the removed items.
    if (trav == null) {    // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```



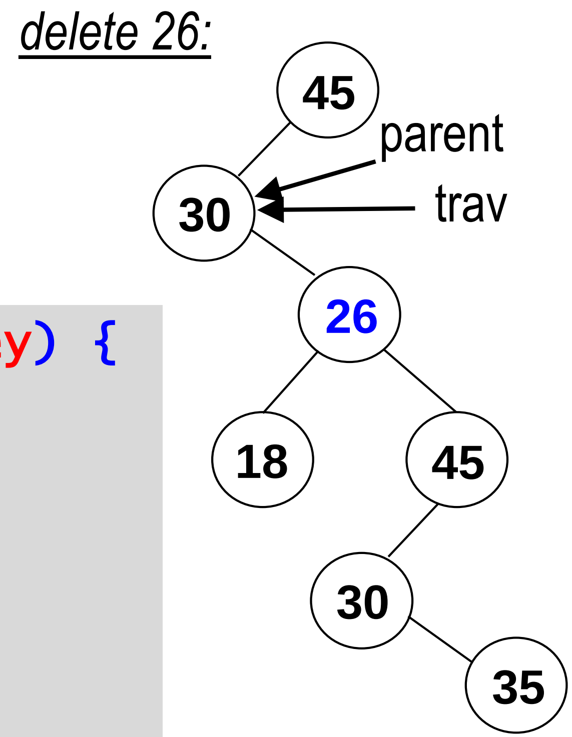
Implementing Deletion

```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```



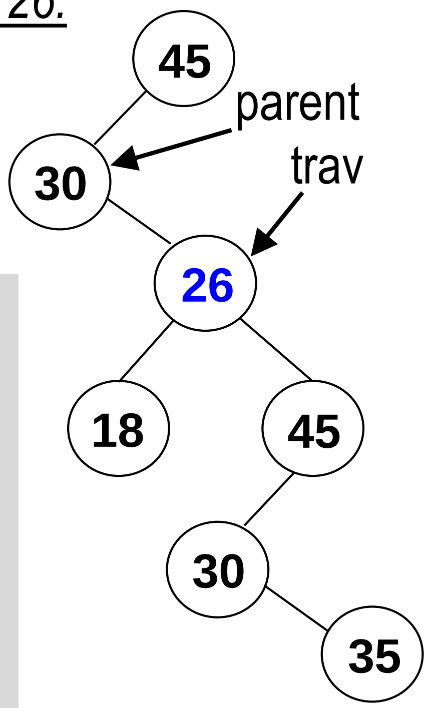
Implementing Deletion

```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```



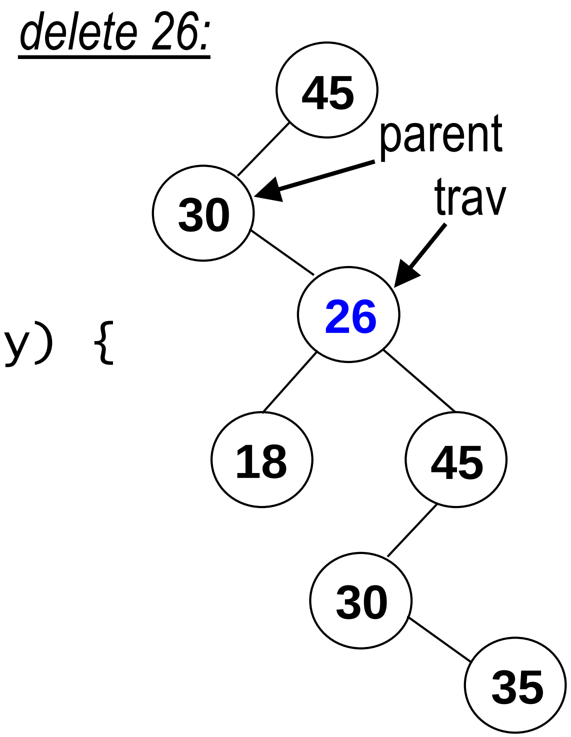
Implementing Deletion

delete 26:




```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```

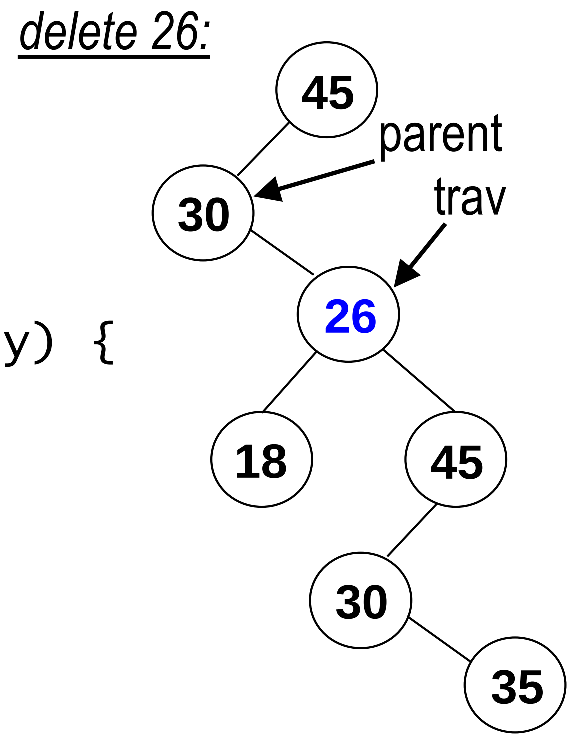
Implementing Deletion




```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```

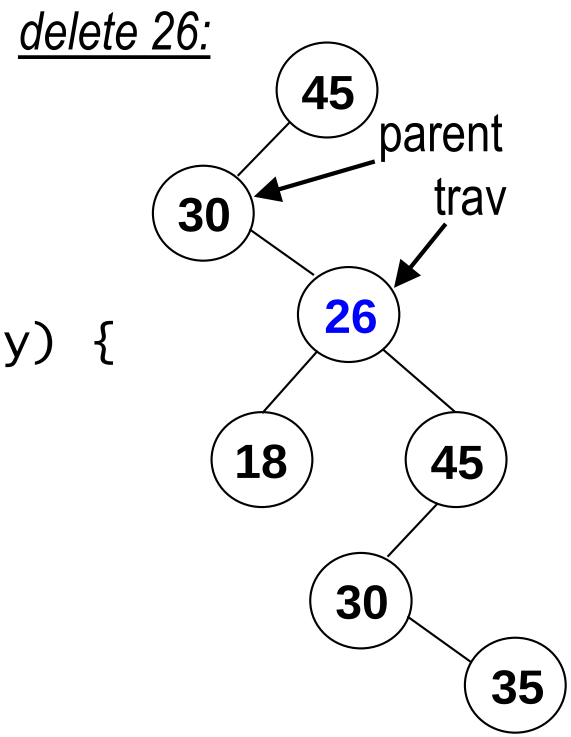
Implementing Deletion

```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```



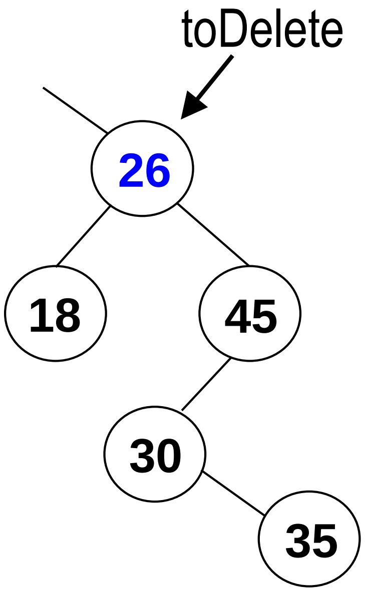
Implementing Deletion

```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null) {   // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);   // call helper method
        return removedData;
    }
}
```



Method that performs the actual delete…

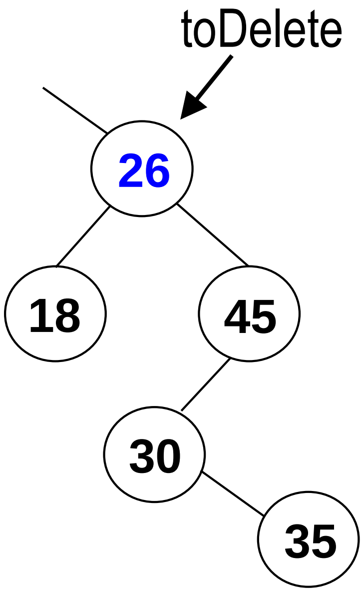
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // What should go here?




        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Case 3:

node to delete has two children

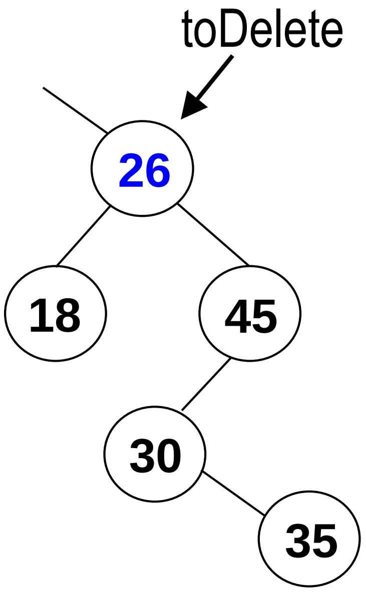
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // What should go here?




        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Case 3:

node to delete has two children

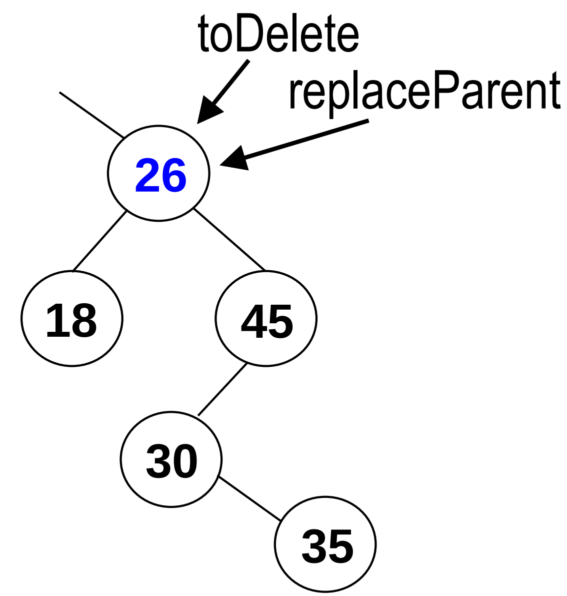
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // What should go here?



        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Case 3:

node to delete has two children

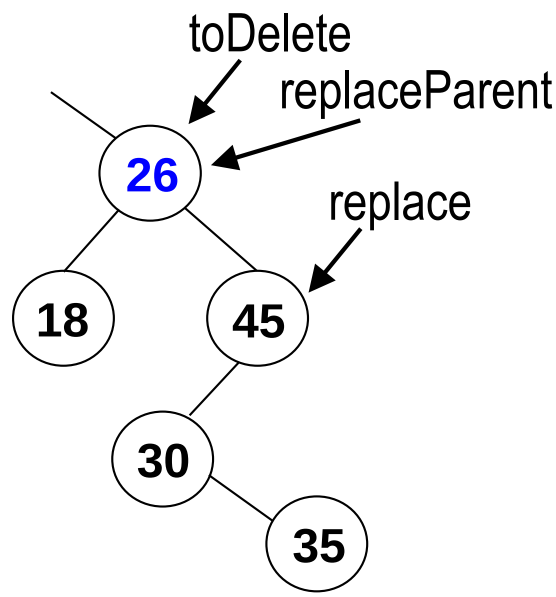
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // What should go here?




        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Case 3:

node to delete has two children

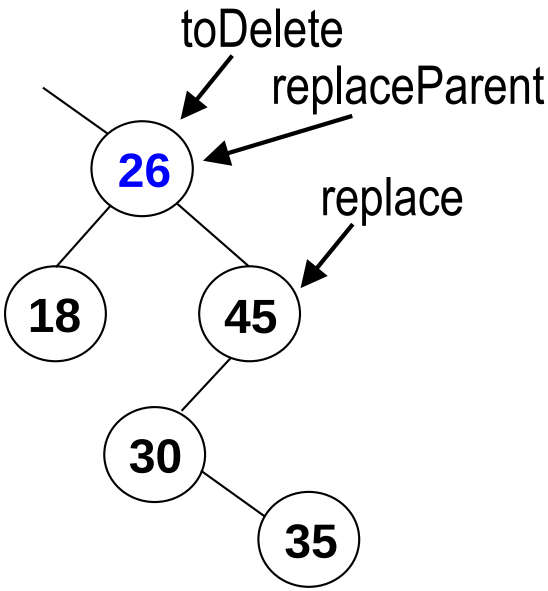
```java
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // What should go here?



        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Case 3:

node to delete has two children

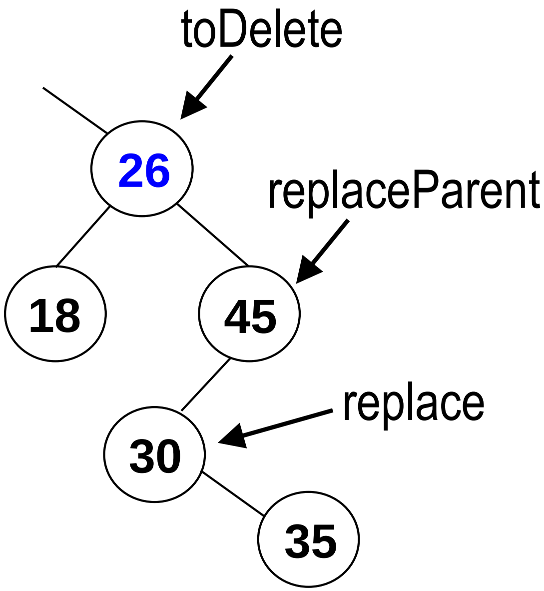
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        while (replace.left != null) {
            replaceParent = replace;
            replace = replace.left;
        }

        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Case 3:

node to delete has two children

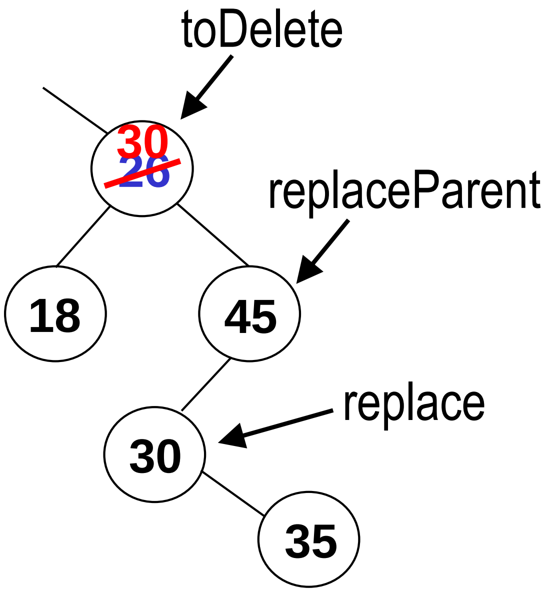
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        while (replace.left != null) {
            replaceParent = replace;
            replace = replace.left;
        }

        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Case 3:

node to delete has two children

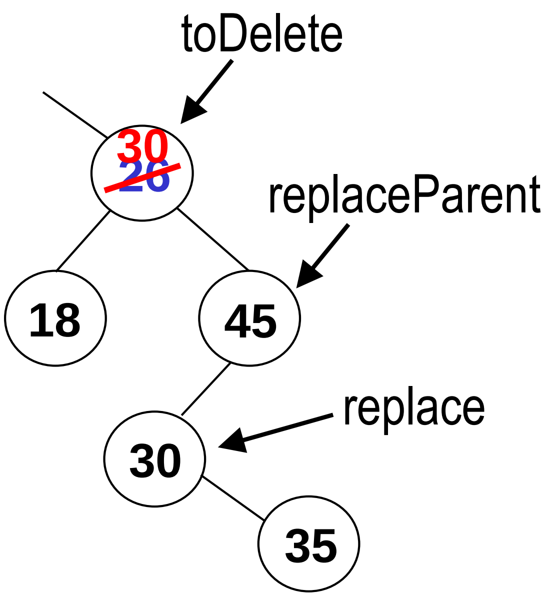
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        while (replace.left != null) {
            replaceParent = replace;
            replace = replace.left;
        }

        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Case 3:

node to delete has two children

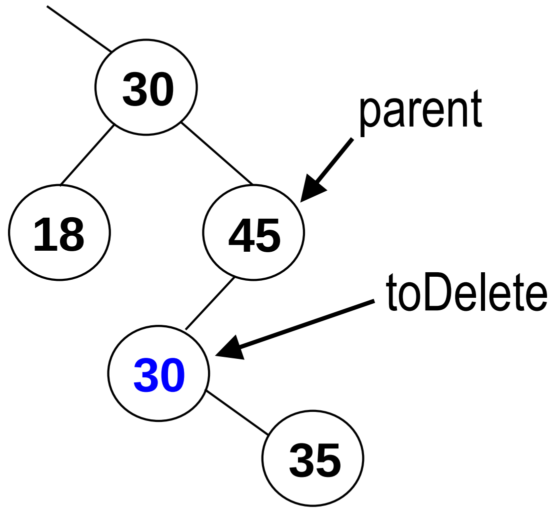
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        while (replace.left != null) {
            replaceParent = replace;
            replace = replace.left;
        }

        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Cases 1 and 2:

node to delete has at most one child

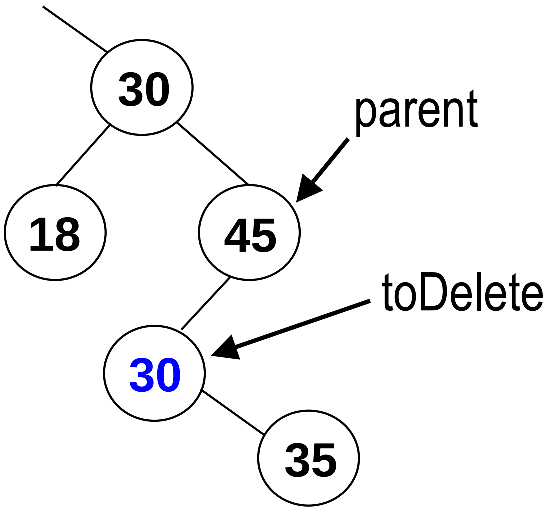
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Implementing Cases 1 and 2:

node to delete has at most one child

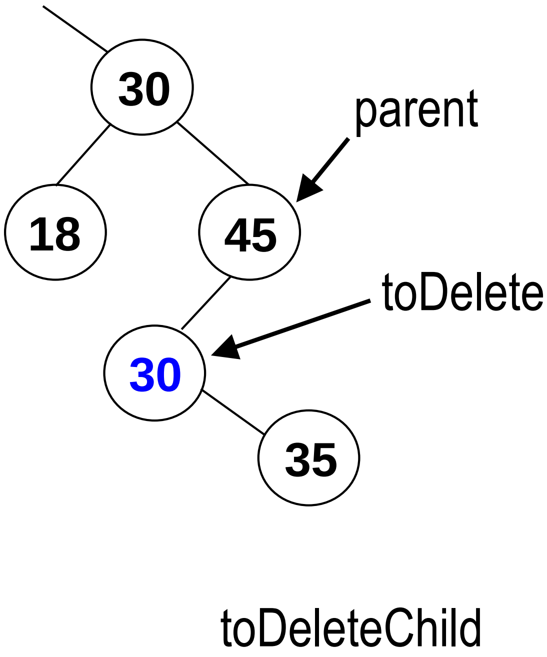
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Implementing Cases 1 and 2:

node to delete has at most one child

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Implementing Cases 1 and 2

node to delete has at most on

```
private void deleteNode(Node toDelete, Node
    if (toDelete.left != null && toDelete.ri
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null)
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```

Note that we can enter this segment of the code either by calling this method recursively from Case 3 or by calling this method directly if deleting a node in Case 1 or 2. Therefore the node we want to delete will either have no children or one child (left or right).

Delete

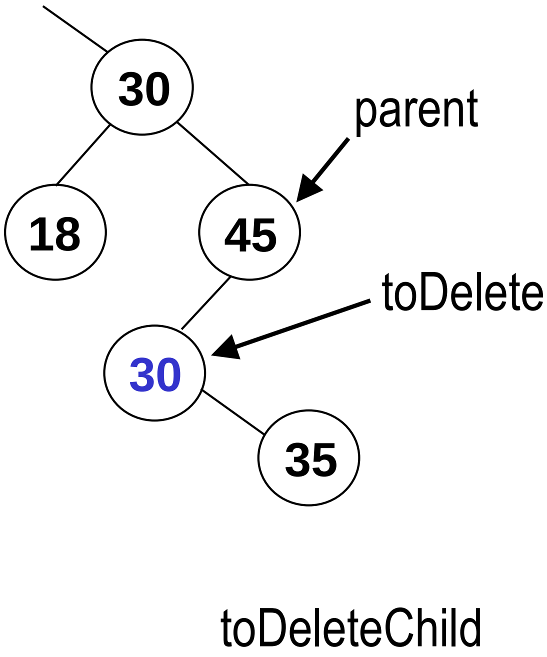
35

toDeleteChild

Implementing Cases 1 and 2:

node to delete has at most one child

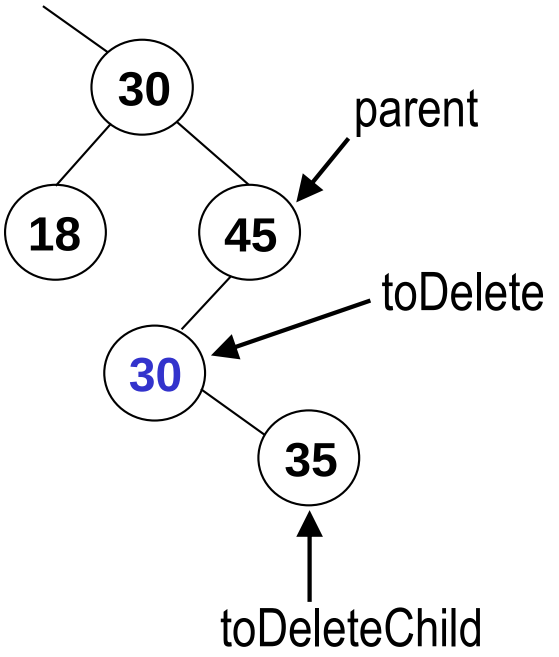
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Implementing Cases 1 and 2:

node to delete has at most one child

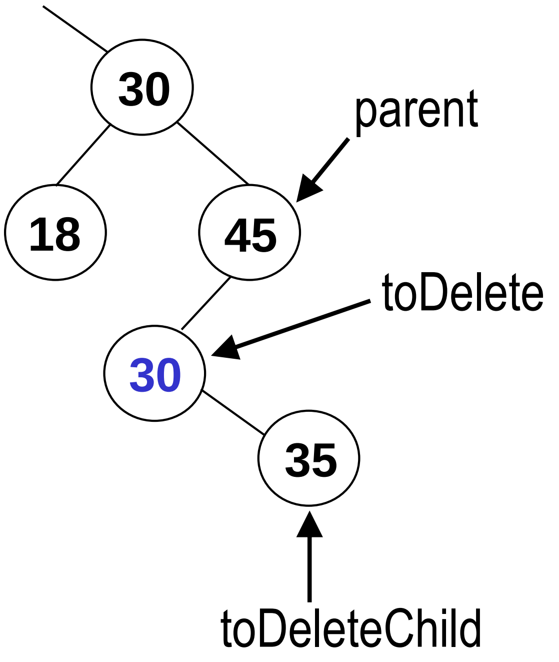
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Implementing Cases 1 and 2:

node to delete has at most one child

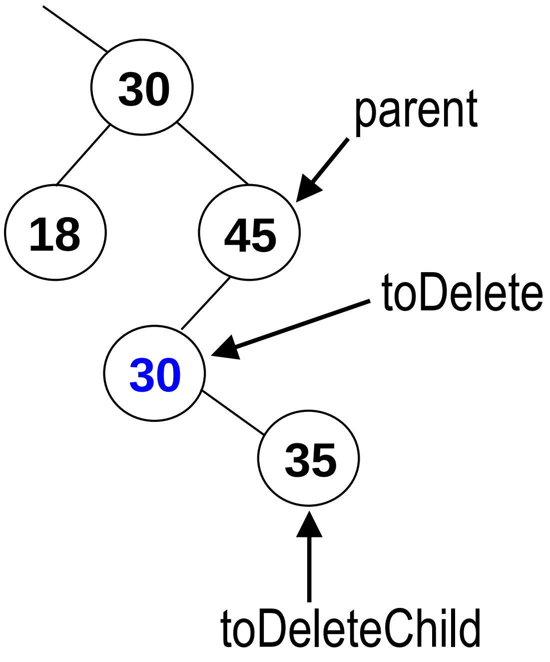
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Implementing Cases 1 and 2:

node to delete has at most one child

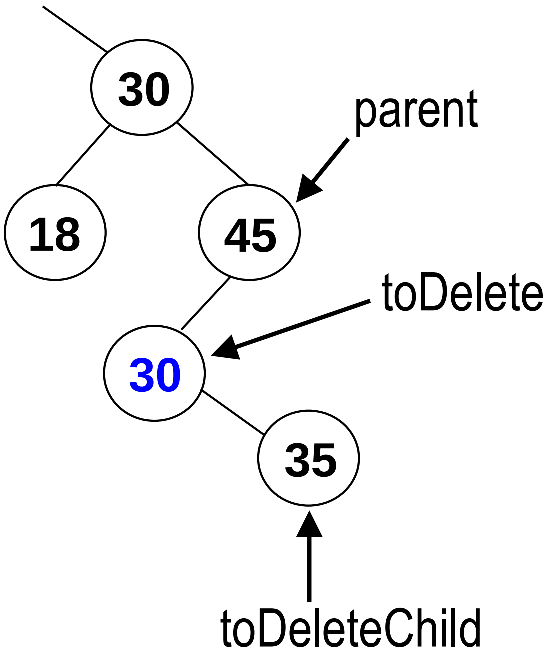
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Implementing Cases 1 and 2:

node to delete has at most one child

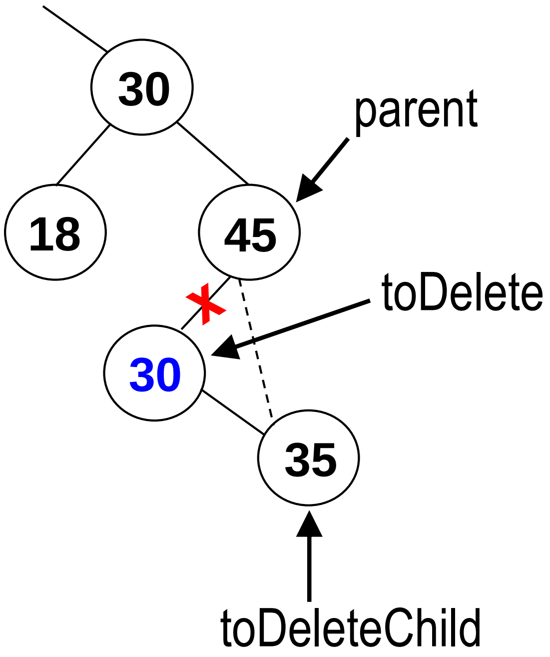
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Implementing Cases 1 and 2:

node to delete has at most one child

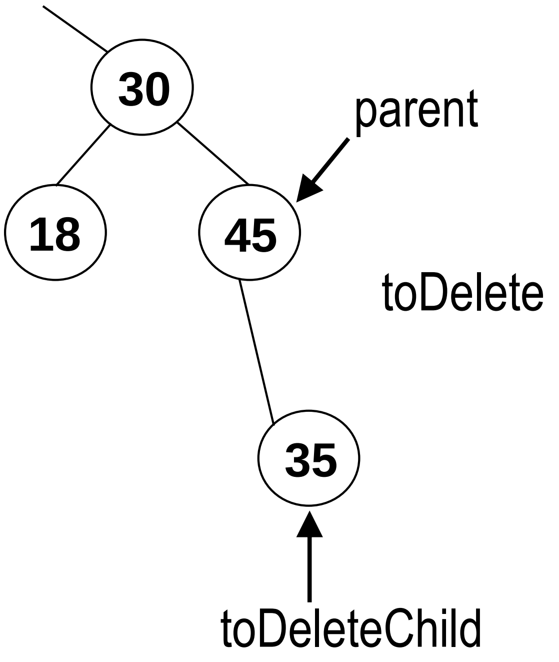
```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



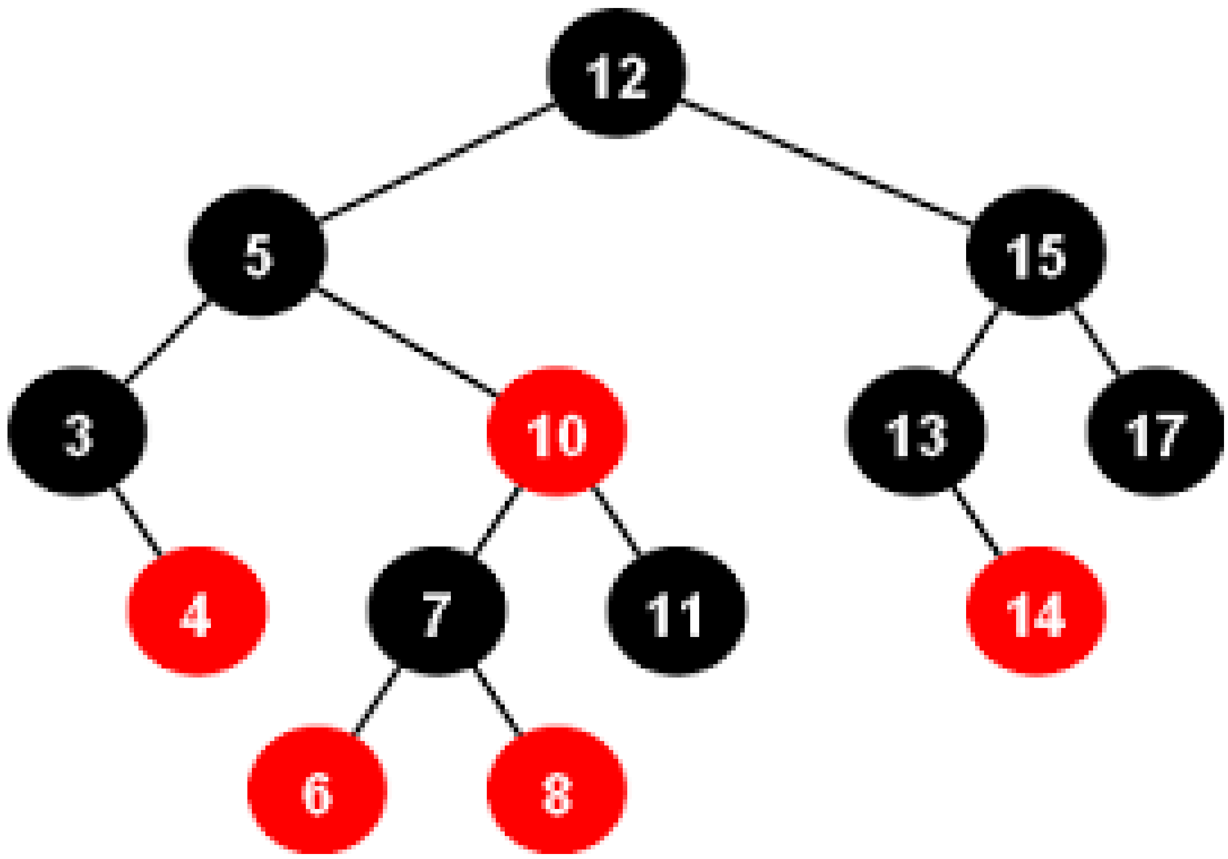
Implementing Cases 1 and 2:

node to delete has at most one child

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null) {
            toDeleteChild = toDelete.left;
        } else {
            toDeleteChild = toDelete.right;
        }
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root) {
            root = toDeleteChild;
        } else if (toDelete.key < parent.key) {
            parent.left = toDeleteChild;
        } else {
            parent.right = toDeleteChild;
        }
    }
}
```



Efficiency of Binary Search Trees



Efficiency of a Binary Search Tree

- For a tree containing *n* items, what is the efficiency of any of all of the traversal algorithms?

Efficiency of a Binary Search Tree

- For a tree containing *n* items, what is the efficiency of any of all of the traversal algorithms? $O(n)$

Efficiency of a Binary Search Tree

- For a tree containing n items, what is the efficiency of any of all of the traversal algorithms? $O(n)$
  - you process all n of the nodes
  - you perform $O(1)$ operations on each of them

Efficiency of a Binary Search Tree

- For a tree containing *n* items, what is the efficiency of any of all of the traversal algorithms? $O(n)$
  - you process all *n* of the nodes
  - you perform $O(1)$ operations on each of them
- Search, insert, and delete all have the same time complexity.

Efficiency of a Binary Search Tree

- For a tree containing n items, what is the efficiency of any of all of the traversal algorithms? $O(n)$
  - you process all n of the nodes
  - you perform $O(1)$ operations on each of them
- Search, insert, and delete all have the same time complexity.
  - insert is a search followed by a $O(1)$ operations

Efficiency of a Binary Search Tree

- For a tree containing n items, what is the efficiency of any of all of the traversal algorithms? $O(n)$
  - you process all n of the nodes
  - you perform $O(1)$ operations on each of them

- Search, insert, and delete all have the same time complexity.
  - insert is a search followed by a $O(1)$ operations
  - delete involves either:
    - a search followed by $O(1)$ operations (cases 1 and 2)

Efficiency of a Binary Search Tree

- For a tree containing *n* items, what is the efficiency of any of all of the traversal algorithms? *O*(*n*)
  - you process all *n* of the nodes
  - you perform *O*(1) operations on each of them

- Search, insert, and delete all have the same time complexity.
  - insert is a search followed by a *O*(1) operations
  - delete involves either:
    - a search followed by *O*(1) operations (cases 1 and 2)
    - a search partway down the tree for the item, followed by a search further down for its replacement, followed by *O*(1) operations (case 3)

Efficiency of a Binary Search Tree

- For a tree containing *n* items, what is th… of all of the traversal algorithms…
  - you process all *n* of the n…
  - you perform $O(1)$ operatio…

> Complexity of each algorithm is dominated by the complexity of the search!

- Search, insert, and delete all ha… …omplexity.
  - insert is a search followe… by a $O(1)$ operations
  - delete involves either:
    - a search followed by $O(1)$ operations (cases 1 and 2)
    - a search partway down the tree for the item, followed by a search further down for its replacement, followed by $O(1)$ operations (case 3)

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case:

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root

Does this imply that the best case is when the tree has one node?

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root
    - **note:** the best case is *not* when the tree has one node!

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root
    - **note:** the best case is *not* when the tree has one node!
  - worst case: $O(h)$, where h is the height of the tree

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root
    - **note:** the best case is *not* when the tree has one node!
  - worst case: $O(h)$, where h is the height of the tree
    - you have to go all the way down to level h before finding the key or realizing it isn't there

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root
    - **note:** the best case is *not* when the tree has one node!
  - worst case: $O(h)$, where h is the height of the tree
    - you have to go all the way down to level h before finding the key or realizing it isn't there
    - along the path to level h, you process $h + 1$ nodes

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root
    - **note:** the best case is *not* when the tree has one node!
  - worst case: $O(h)$, where h is the height of the tree
    - you have to go all the way down to level h before finding the key or realizing it isn't there
    - along the path to level h, you process $h + 1$ nodes
  - average case: $O(h)$
    - sometimes you find the key in the root
    - sometimes you go down 1 level, sometimes 2 levels, etc.

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root
    - **note:** the best case is *not* when the tree has one node!
  - worst case: $O(h)$, where h is the height of the tree
    - you have to go all the way down to level h before finding the key or realizing it isn't there
    - along the path to level h, you process $h + 1$ nodes
  - average case: $O(h)$
    - sometimes you find the key in the root
    - sometimes you go down 1 level, sometimes 2 levels, etc.
    - on average, you go down $h/2$ levels, but that's still $O(h)$!

Efficiency of a Binary Search Tree (cont.)

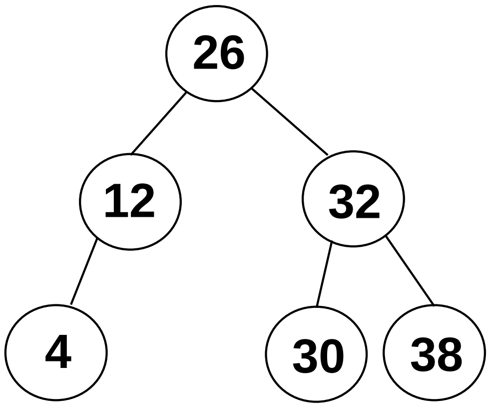
- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root
    - **note:** the best case is *not* when the tree has one node!
  - worst case: $O(h)$, where h is the height of the tree
    - you have to go all the way down to level h before finding the key or realizing it isn't there
    - along the path to level h, you process $h + 1$ nodes
  - average case: $O(h)$
    - sometimes you find the key in the root
    - sometimes you go down 1 level, sometimes 2 levels, etc.
    - on average, you go down $h/2$ levels, but that's still $O(h)$!
- What is the height of a tree containing n items?

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching for a key:
  - best case: $O(1)$, when you find the key in the root
    - **note:** the best case is *not* when the tree has one node!
  - worst case: $O(h)$, where h is the height of the tree
    - you have to go all the way down to level h before finding the key or realizing it isn't there
    - along the path to level h, you process $h + 1$ nodes
  - average case: $O(h)$
    - sometimes you find the key in the root
    - sometimes you go down 1 level, sometimes 2 levels, etc.
    - on average, you go down $h/2$ levels, but that's still $O(h)$!
- What is the height of a tree containing n items?
  - it depends!

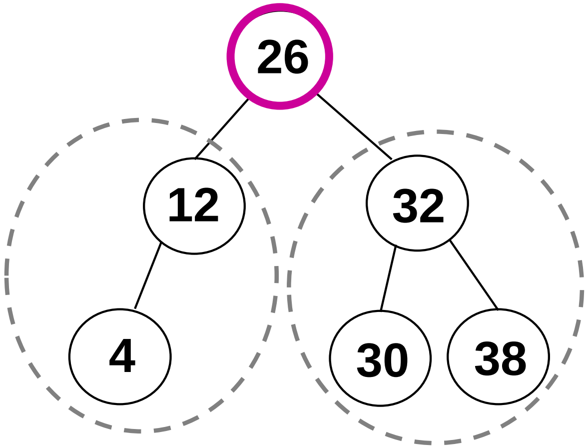
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:



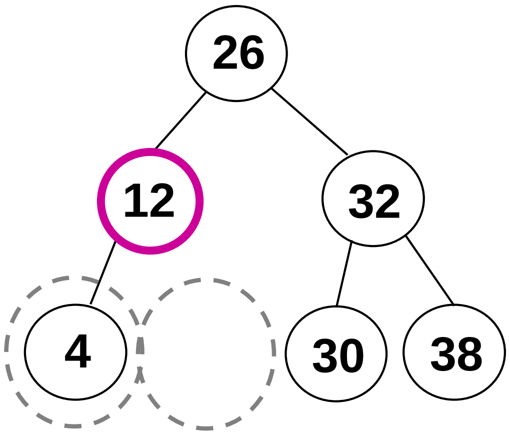
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1



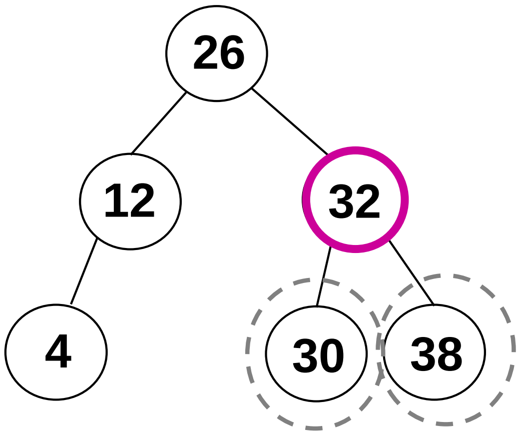
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)



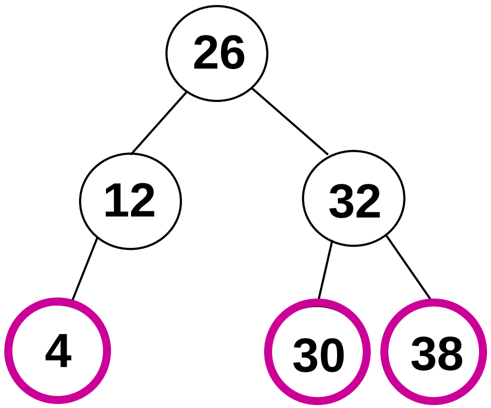
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)
    - 32: both subtrees have a height of 0



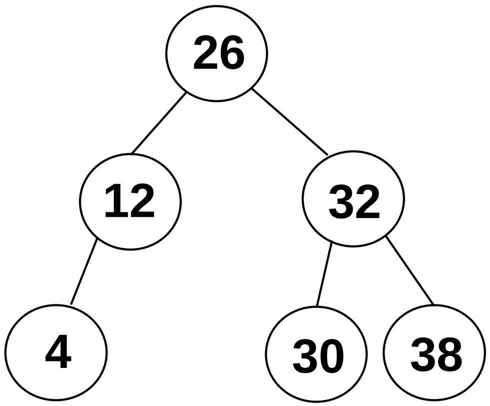
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)
    - 32: both subtrees have a height of 0
    - all leaf nodes: both subtrees are empty



Balanced Trees

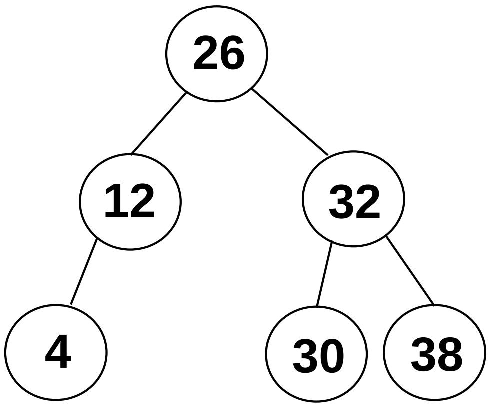
- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)
    - 32: both subtrees have a height of 0
    - all leaf nodes: both subtrees are empty



- For a balanced tree with *n* nodes, height = $O(\log n)$
  - every time you follow an edge down the longest path, you cut the problem size roughly in half!

Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)
    - 32: both subtrees have a height of 0
    - all leaf nodes: both subtrees are empty

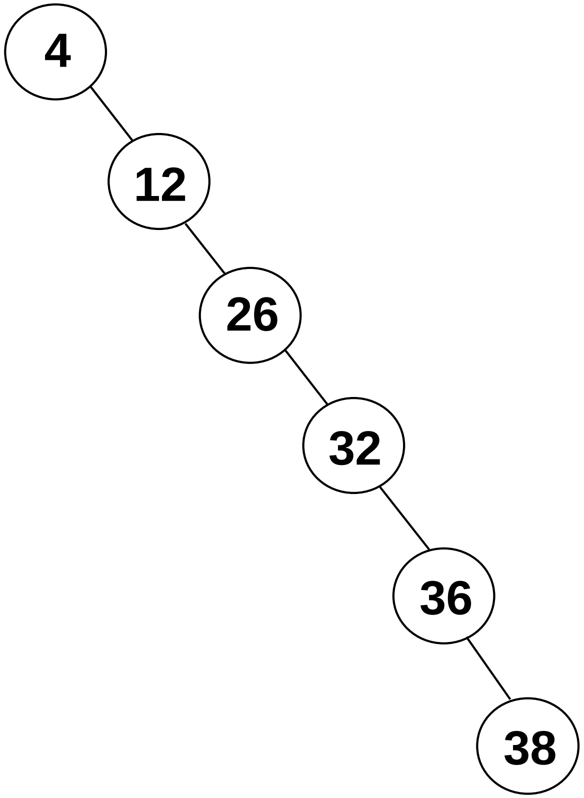


- For a balanced tree with n nodes, height = $O(\log n)$
  - every time you follow an edge down the longest path, you cut the problem size roughly in half!

- Therefore, for a balanced binary search tree, the worst case for search / insert / delete is $O(h) = O(\log n)$
  - the "best" worst-case time complexity

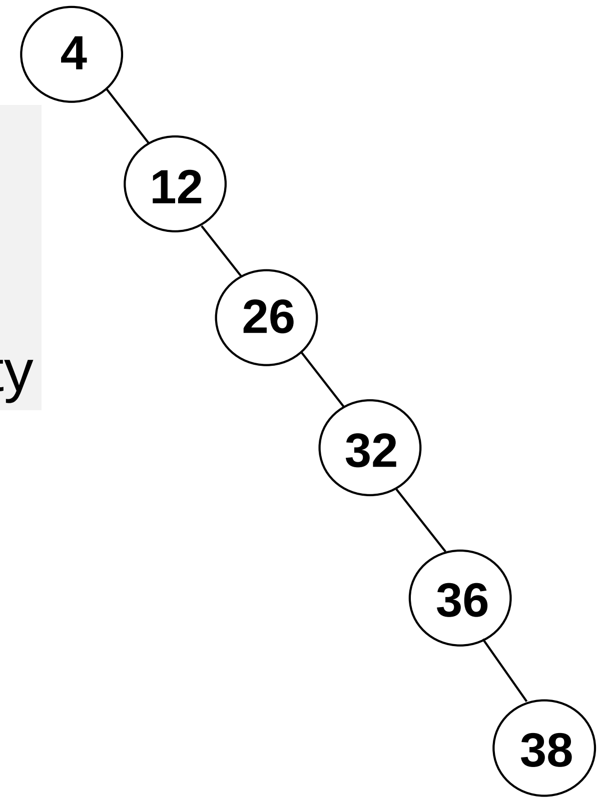
What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
  - height = $n - 1$



What If the Tree Isn't Balanced?

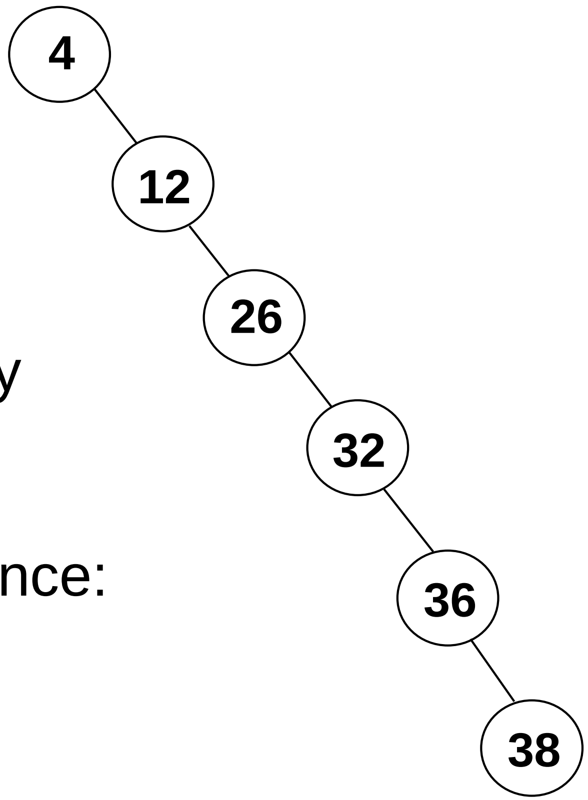
- Extreme case: the tree is equivalent to a linked list
  - height = $n - 1$



- Therefore, for a unbalanced binary search tree, the worst case for search / insert / delete is $O(h) = O(n)$
  - the "worst" worst-case time complexity

What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
  - height = $n - 1$



- Therefore, for a unbalanced binary search tree, the worst case for search / insert / delete is $O(h) = O(n)$
  - the "worst" worst-case time complexity

- We'll look next at search-tree variants that take special measures to ensure balance:
  - 2-3 trees
  - B-trees