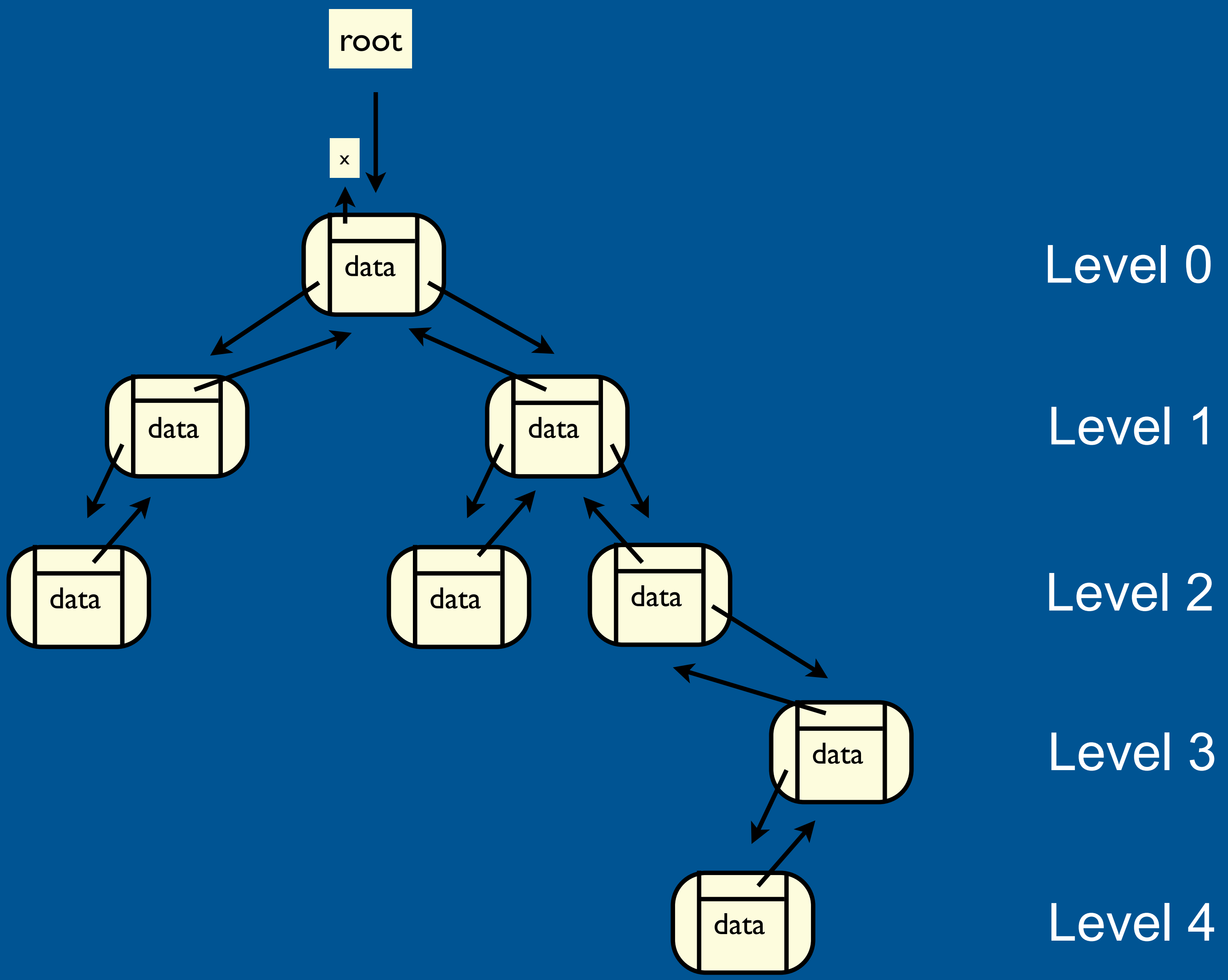
Data Structures Part 2
Binary Trees

Binary Tree

- Like linked lists, binary trees are made of fundamental nodes and links that join them
- Each node can contain as much data as we want plus two links to two *children*
- A binary tree may contain a third link to a *parent* node

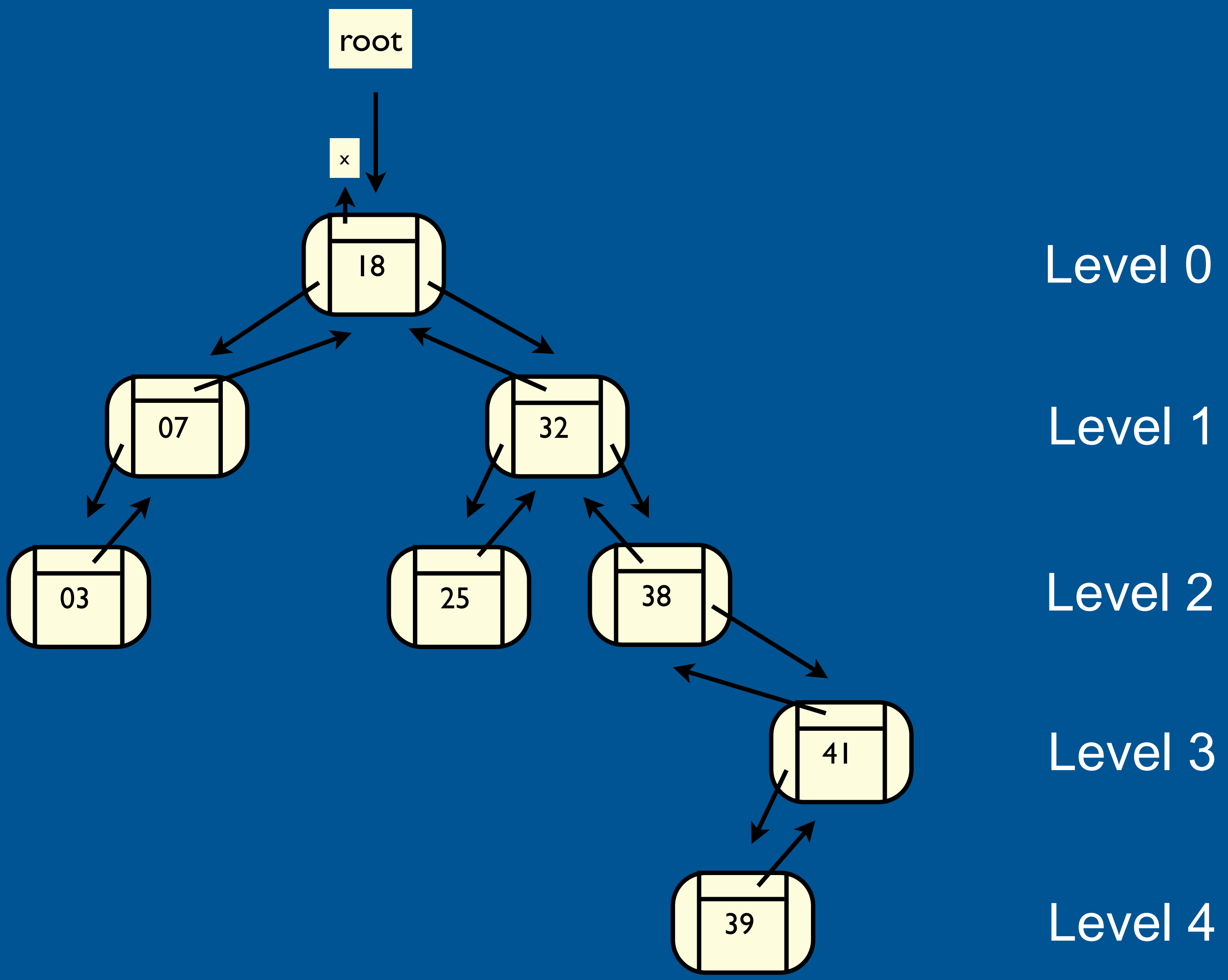
Example of a binary tree



Binary Tree

- Binary trees are usually ordered by one of the data fields
- The example below is ordered by unique identification numbers
- A node that is to the left of another node will always have a smaller identification number
- A node that is to the right of another node will always have a larger identification number
- The order is *depth-first* and moves from *left to right*

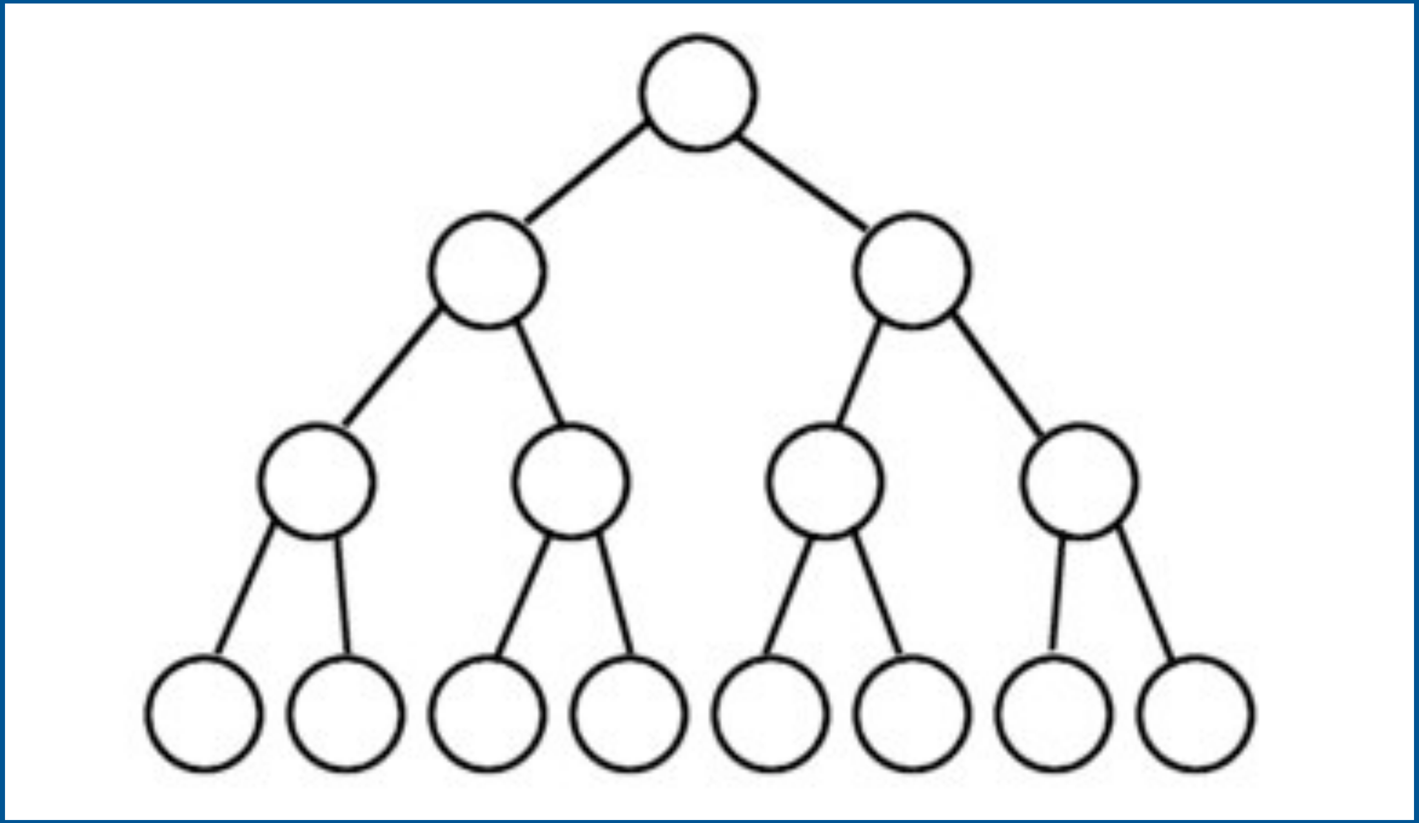
Example of a binary tree



Maximum number of nodes (full binary tree)

$2^n - 1$ in a tree with n levels
2^k in kth level, where $0 <= k < n$

Number of nodes in kth level = Sum of numbers in kth row of Pascal's Triangle



Level 0

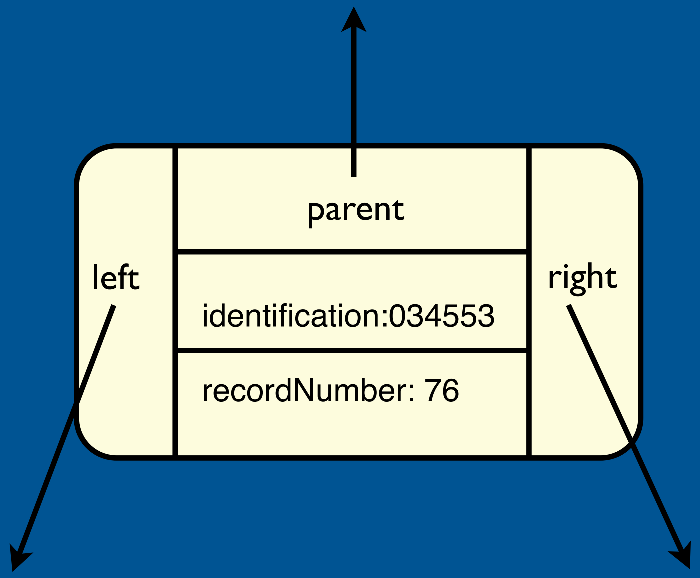
Level 1

Level 2

Level 3

Java: Designing the node with two links to children and one link to parent

TNode



```
public class TNode {
    private String identification = "";
    private int recordNumber = -1;
    private TNode left   = null;
    private TNode right  = null;
    private TNode parent = null;

    public TNode() {
        identification = "";
        recordNumber = -1;
        left   = null;
        right  = null;
        parent = null;
    }

    public TNode(String i, int recNo, TNode l, TNode r, TNode p) {
        identification = i;
        recordNumber = recNo;
        left   = l;
        right  = r;
        parent = p;
    }
}

public class Message {
    public static void main(String[] args) {
        TNode p = new TNode("034553", 76, null, null, null);
    }
}
```

Tree Operations

We will learn how to implement these operations using trees:

- Inserting a node
- Printing a tree in order using *depth-first*
- Finding a node
- Deleting a node

Inserting, printing, and finding will be implemented using recursion
Deleting will be implemented using iteration

Tree Operations - Inserting

pseudo-code

```
insertNode(TNode p) {
    if tree is empty
        let root point to p
    elsif identification of p is lower than identification of root then
        if left child of root is null then
            let left child of root point to p
            let parent of p point to root
        else
            instantiate a new tree t whose root is the left child of root
            t.insertNode(p)
        end if
    elsif identification of p is higher than identification of root then
        if right child of root is null then
            let right child of root point to p
            let parent of p point to root
        else
            instantiate a new tree t whose root is the right child of root
            t.insertNode(p)
        end if
    else
        error: attempting to insert an identification that already exists in tree
    end if
}
```

Tree Operations - Depth-First Printing

pseudo-code

```
printTree() {
    if tree is not empty
        instantiate a new tree t whose root is the left child of the root
        t.printTree()

        print the root

        instantiate a new three t whose root is the right child of the root
        t.printTree()
    end if
}
```

Tree Operations - Finding

pseudo-code

```
TNode findNode(String identification) {
    if tree is empty
        return null
    elsif identification equals the identification of the root
        return root
    elsif identification is lower than the identification of the root
        instantiate a tree t whose root is the left child of root
        return t.findNode(identification)
    elsif identification is higher than the identification of the root
        instantiate a tree t whose root is the right child of root
        return t.findNode(identification)
    else
        fatal error
        return null
    end if
}
```

Exercise 1 Trees

Create a `main()` method that tests the following:

1) Inserting a node on an empty tree, then print the tree
2) Inserting a node that will end up on the left subtree, then print the tree
3) Inserting a node that will end up on the right subtree, then print the tree
4) Inserting five nodes in such a way that a tree with levels 0, 1 and 2 is created
5) Inserting a few nodes in random order, then print the tree (should appear in order)
6) Inserting 500 nodes of random identifications, then print the tree
7) Finding a node in an empty tree
8) Finding a node that we know is to the left of the root
9) Finding a node that we know is to the right of the root
10) Finding a node that is not in the tree

Exercise 2 Trees

Draw the tree that is created when the following pseudo-code is implemented and run:

```
main method() {
    instantiate tree t

    t.insertNode with identification "034"
    t.insertNode with identification "023"
    t.insertNode with identification "164"
    t.insertNode with identification "115"
    t.insertNode with identification "137"
    t.insertNode with identification "004"
    t.insertNode with identification "137"
    t.insertNode with identification "151"
    t.insertNode with identification "128"
    t.insertNode with identification "172"
    t.insertNode with identification "004"
    t.insertNode with identification "170"
    t.insertNode with identification "120"
}
```

Exercise 3 Trees

Implement the pseudo-code of Exercise 2 in Java. Then, write a modified version of `printTree()` that prints the identification of each node along with the level where the node is located in the tree. The signature of this modified `printTree()` is

```
public void printTree(int level)
```

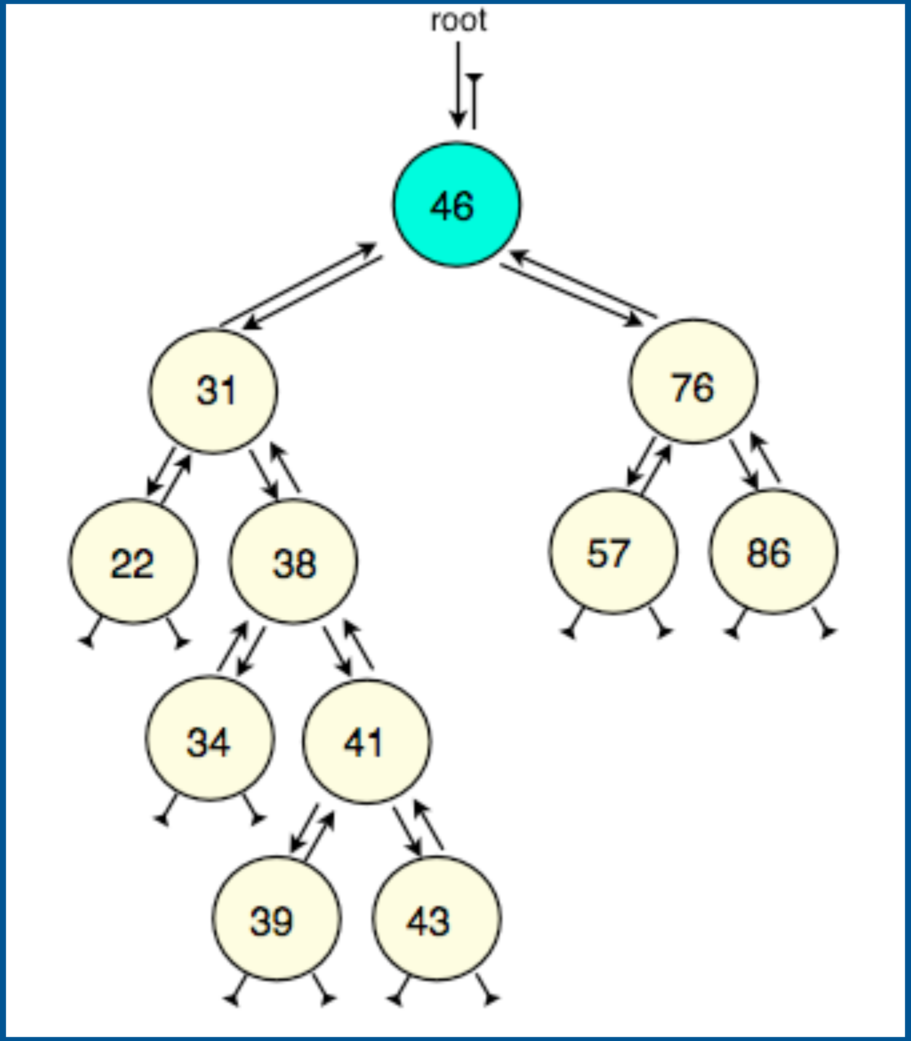
where the value of `level` is increased by 1 everytime a recursive call is made. The initial call is `printTree(0)`

The output from printing the tree of Exercise 2 is:

```
Id: 004 in level 2
Id: 023 in level 1
Id: 034 in level 0
Id: 115 in level 2
Id: 120 in level 5
Id: 128 in level 4
Id: 137 in level 3
Id: 151 in level 4
Id: 164 in level 1
Id: 170 in level 3
Id: 172 in level 2
```

Exercise 4 Trees

Write a `main()` method that generates the tree below. Then use the `printTree(int)` method to check that the `main()` method places the nodes in the correct level



Exercise 5 Trees

Write a `main()` method that generates a full tree of four levels. A full tree is a tree whose every level contains 2^k nodes, where `k` is the level number and `0 <= k < levels`. Confirm that the tree is correct by using the `printTree(int)` method

What is the worst-case scenario when looking for a node in this tree, i.e., how many comparisons would it take to conclude that the node we are looking for is not in the tree? Express your answer in terms of the number of nodes in the tree

Exercise 6 Trees

Write a `main()` method that adds eight nodes to a tree such that every node of the tree will not contain a left child, but only a right child or no child.

What can we say about the worst-case scenario when looking for a node in this tree? How does this compare to a linked list?