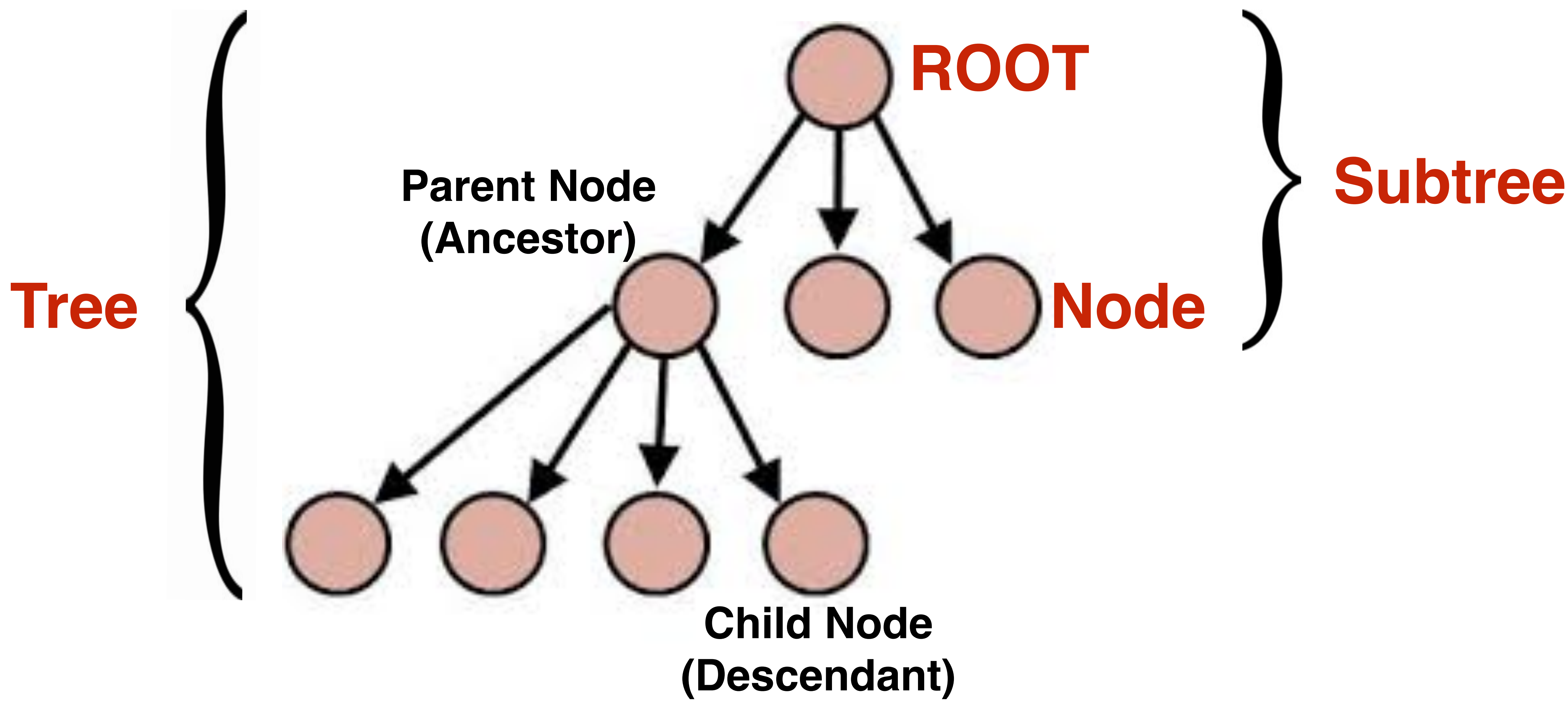
Trees

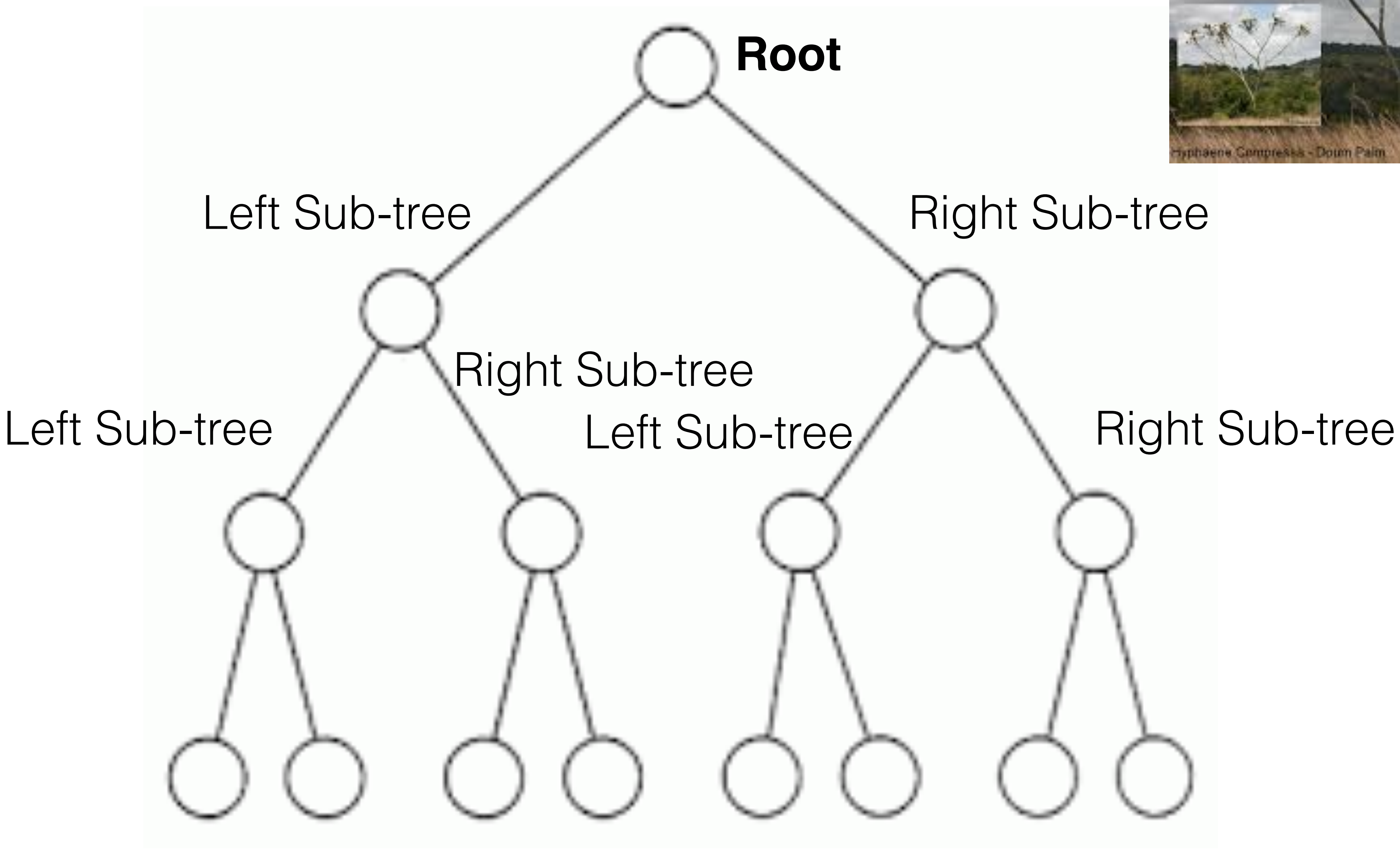
General Trees (1 Root, no restrictions on child nodes)

A “Tree” is a special case of another more general data structure called “Graph”.
A tree is made by connected nodes.
The first node is called “root” of the tree.



A subtree of a node N is a tree with N as its root.
NOTE: A tree is not a linear ADT, therefore it is complicated to address its content with a position number. In a tree, data is organized in a **hierarchical form**.

Binary Tree (1 Root, 2 children per node)



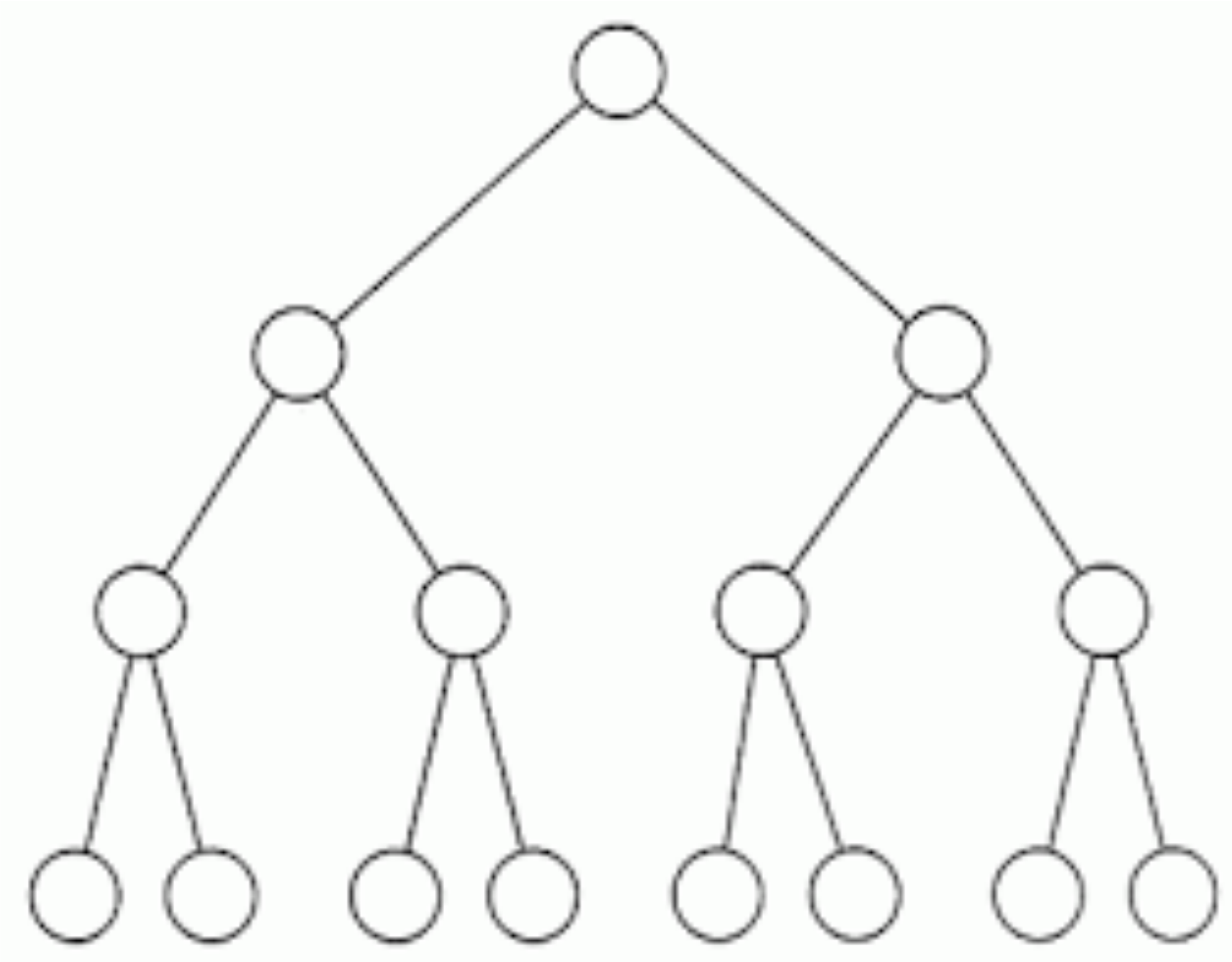
Recursive definition:

1) An empty tree is a binary tree

2) A node with two child subtrees is a binary tree

3) Only what you get from **1)** by a finite application of **2)** is a binary tree

Node Counting in Binary Trees



Level	Nlevel	Ntot
1	$1=2^0$	$1=2^1-1$
2	$2=2^1$	$3=2^2-1$
3	$4=2^2$	$7=2^3-1$
4	$8=2^3$	$15=2^4-1$
⋮	⋮	⋮
L	$\mathbf{1=2^{L-1}}$	$\mathbf{N=2^L-1}$

A **FULL binary tree** is a binary tree where every node has two children, except the last "leaf" nodes.

Node Counting in Binary Trees

Question: Given a binary tree with N nodes, what is its **minimum** height?

- A tree with L-1 levels has at most $2^{L-1}-1$ nodes (see prev. slide). Therefore, it is true that $2^{L-1}-1 < N$.
- On the other side, N cannot exceed the maximum total number of nodes a binary tree can have, so: $N <= 2^{L}-1$.
- If L is the smallest integer such that $N<=2^{L}-1$ If a binary tree has height <= L-1 , then:

$$2^{L-1}-1 < N <= 2^{L}-1$$

which means:

$$2^{L-1} < N+1 <= 2^{L}$$

and therefore:

$$\mathbf{L-1 < \log_2(N+1) <= L}$$

- **Ceiling($\log_2$(N+1))** is the minimum height for a binary tree with N nodes

Node Counting in Binary Trees

Another way to count the total number of nodes is to sum all the nodes in each level:

$$N = \sum_{i=1}^{L} 2^{i-1}$$

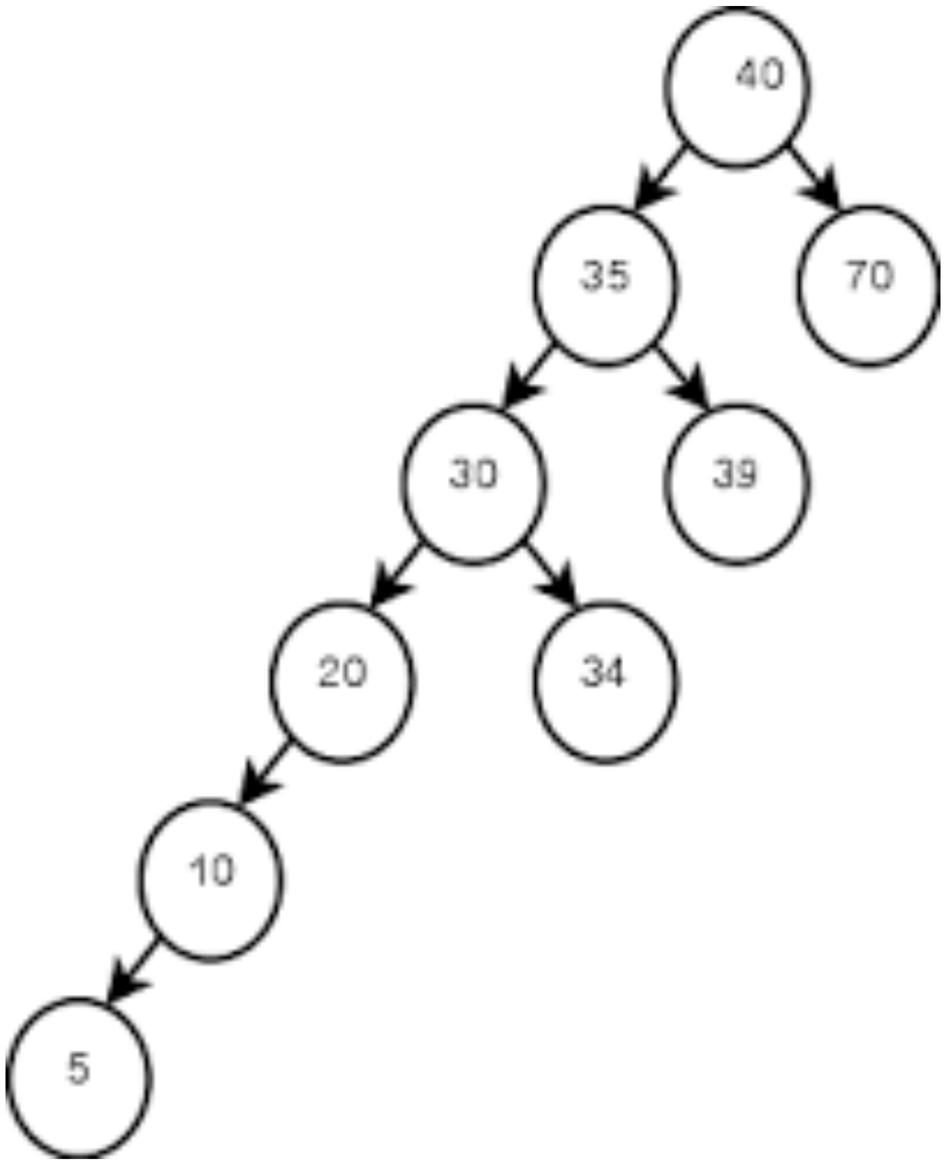
Remembering the geometric sum formula:

$$\sum_{k=0}^{N} r^k = \frac{1 - r^{N+1}}{1 - r}$$

you can prove that indeed

$$N = \sum_{i=1}^{N} 2^{i-1} = 2^{L-1}$$

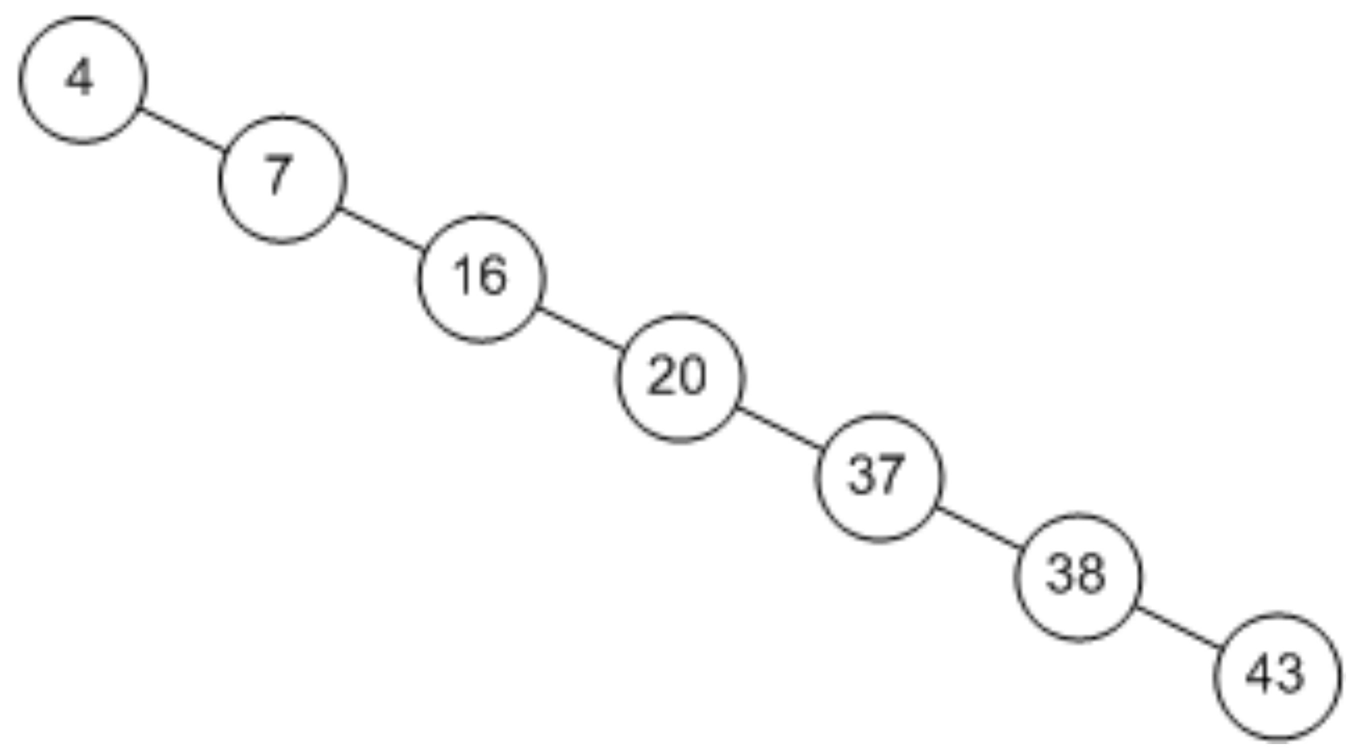
Balanced Trees



When L children are more than R ones (or vice-versa), the trees are called "unbalanced".

If every node has exactly 2 children (in the case of binary trees), the tree is completely balanced.

There are different degrees of balance.
The extreme case of a totally unbalanced tree is the one where there are only L (or R) children and the tree looks basically like a linear data structure:



Tree Traversal

Tree traversal refers to an algorithm for visiting (all) the tree nodes.
Recursion is particularly well suited for this task, as the very definition of binary tree suggests.

```
//Pseudocode
Traverse (Tree){
   Print(Root.Data());
   Traverse (Tree.Left());
   Traverse (Tree.Right());
}
```

Such recursive algorithm first accesses the root and then moves to the subtrees.
The is called **pre-order traversal**.

An alternative is first visit the L subtree, then the root and finally the R tree.
This is called **in-order traversal**.

The remaining case (root at last) is called **post-order traversal**.

Why Trees? Some motivations.

Now we know some properties of the tree ADS, but why should we consider trees?

Trees combine the advantages of **ordered arrays** and **linked** lists together!

Ordered Arrays:

- Quick search (binary)
- Slow insertion

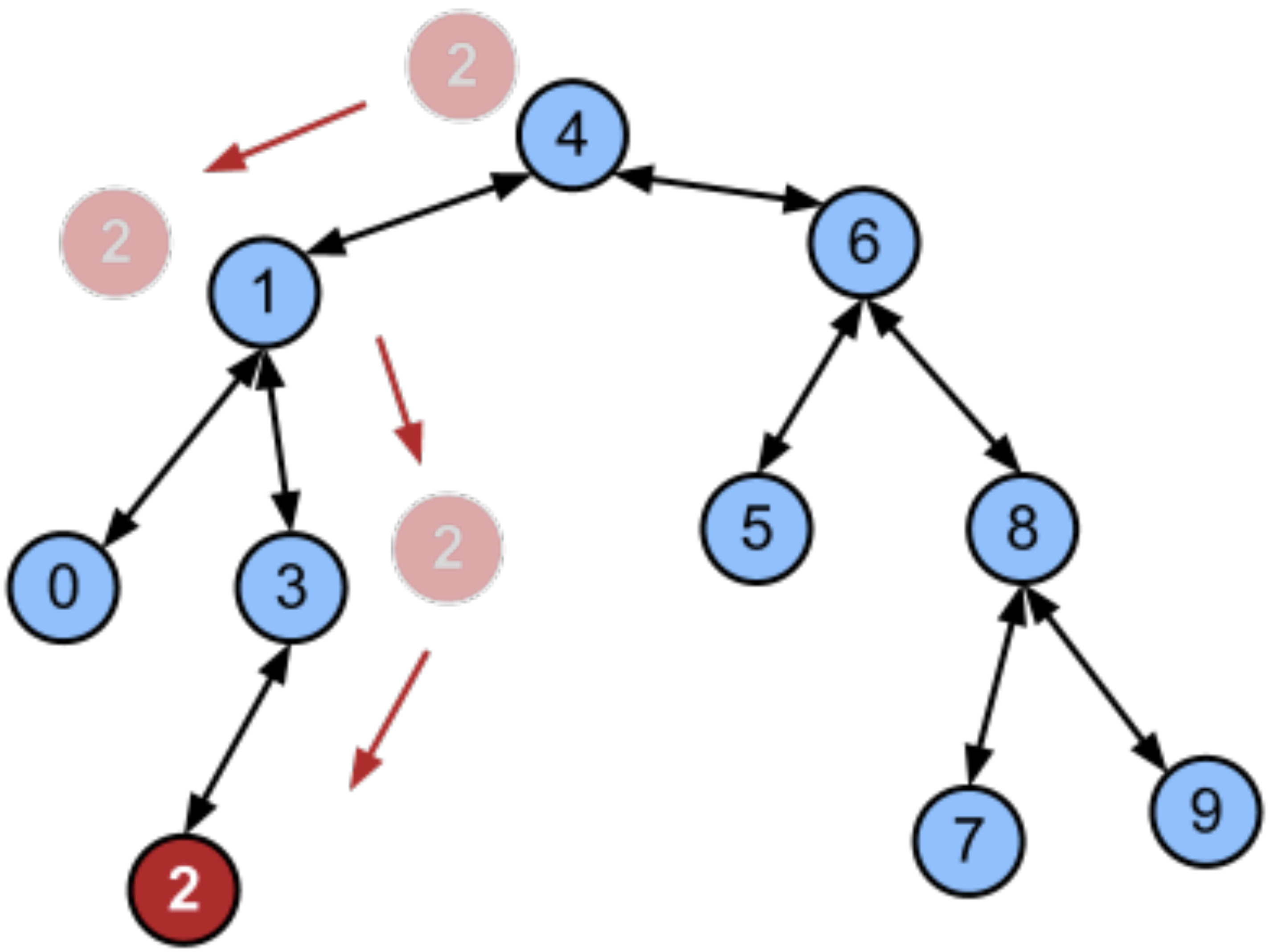
Linked lists:

- Slow search
- Fast insertion

We will see that with a **Binary Tree** we can realize fast insertion and fast search!

Binary Search Tree

A **BST** is a binary tree with a special insertion property:
Every time you insert a new element a new **leaf** is created.
The leaf is created either in the L or R position according to how the new element relates with the parent node.



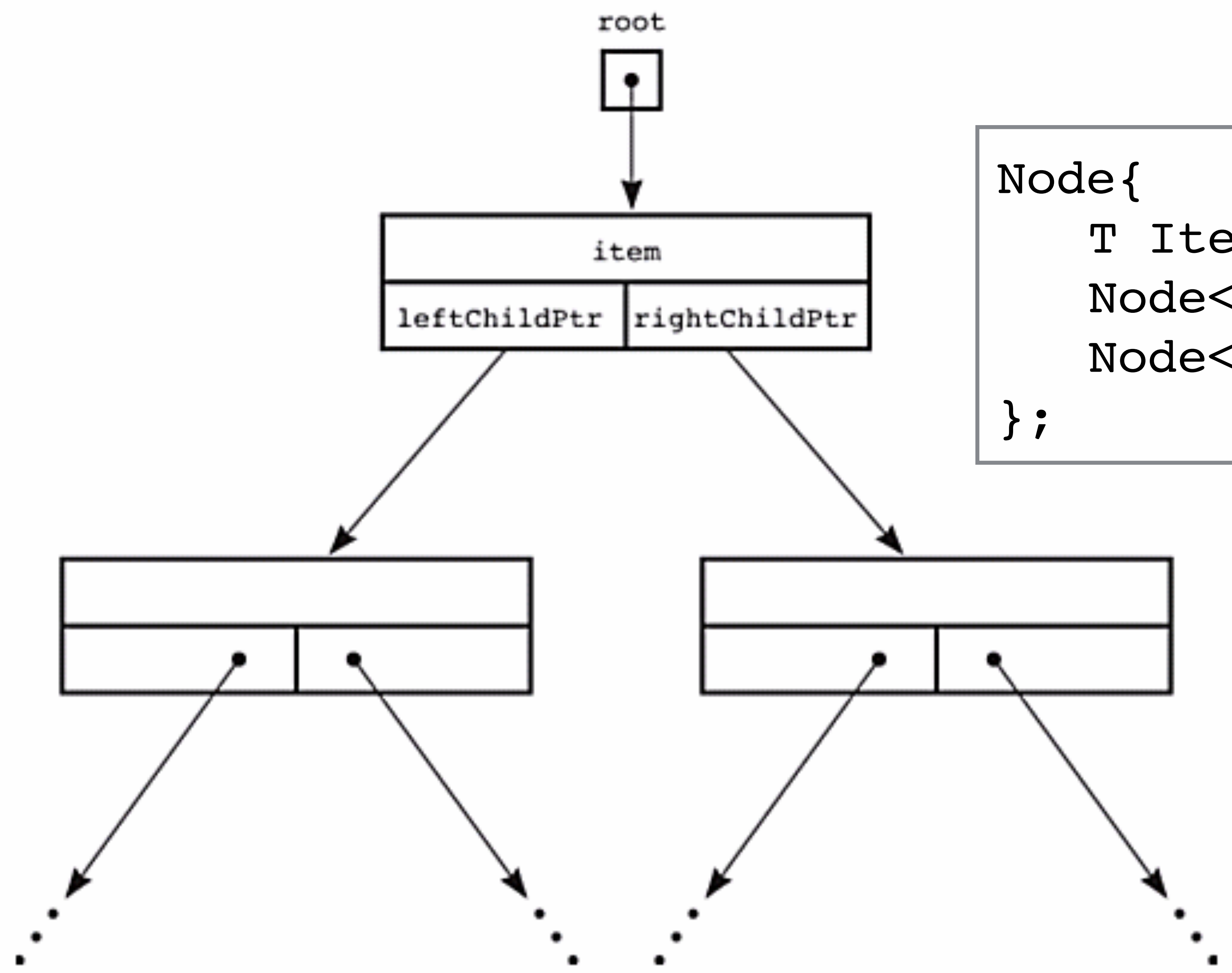
Computational Complexity:

Operation:	Avg	Worst
Retrieval	O(logN)	O(N)
Insertion	O(logN)	O(N)
Removal	O(logN)	O(N)
Traversal	O(N)	O(N)

The Tree ADT

```
template<class T>
class BinaryTreeInterface{

   virtual bool isEmpty() = 0;
   virtual int getHeight() const = 0;
   virtual int getNumberOfNodes() = 0;

   virtual T getRootData() = 0;
   virtual void setRootData(const T& newData) = 0;

   virtual bool add(const T& newData) const = 0;
   virtual bool remove(const T& data) = 0;

   virtual void clear() = 0 ;

   virtual T getEntry(const T& anEntry) const
                  throw(NotFoundException) = 0;

   virtual bool contains(const T& anEntry) const = 0;

   virtual void preorder(void visit(T&)) const = 0;
   virtual void inorder(void visit(T&)) const = 0;
   virtual void postorder(void visit(T&)) const = 0;
};
```

Binary Tree Implementations: Link-Based



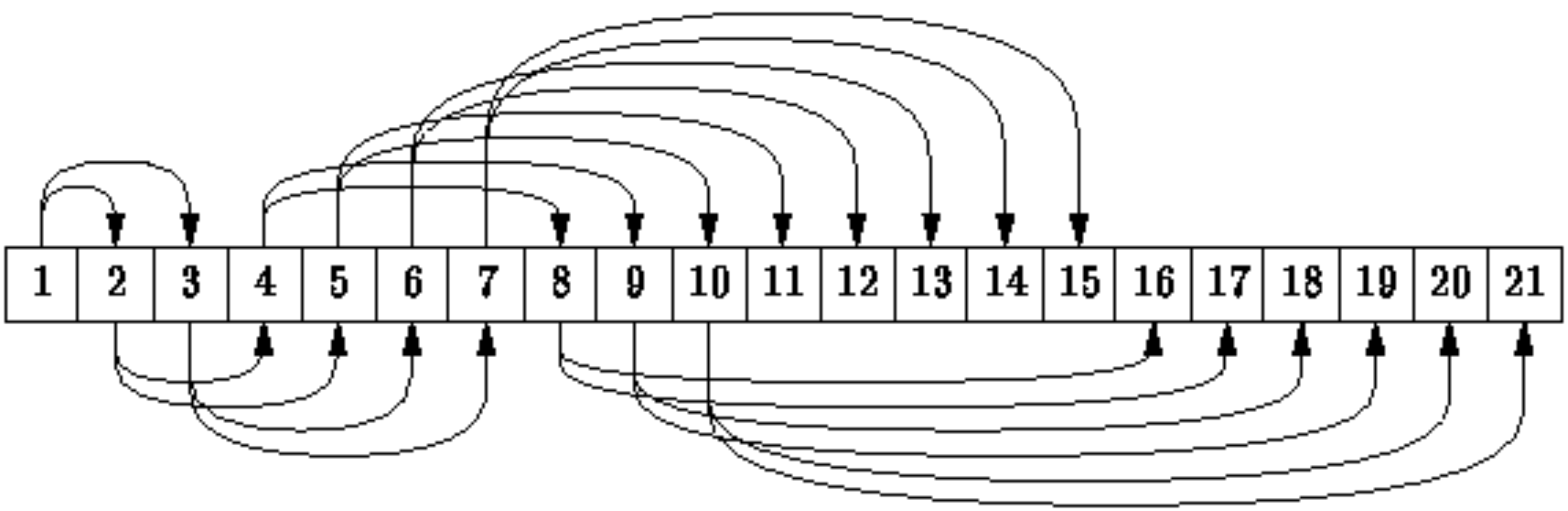

```
Node{
    T Item;
    Node<T> * left;
    Node<T> * right;
};
```

Binary Tree Implementations: Array-Based

We can construct an array of nodes.
The nodes contain the data and the array indices if the child nodes.

```
Node{
    T Item;
    int left;
    int right;
};
```

A complication is the following: as you insert and remove nodes, you have to store the information about which positions in the array are still free (you need a "**free list**").
Insertion and removal operations either consult and modify the free list.



Binary Tree Implementation: keeping the balance.

```
template<class T>
BinaryNode<T>* BinaryNodeTree<T>::balancedAdd
(BinaryNode<T>* subTreePtr,BinaryNode<T>* newNodePtr)
{
   if (subTreePtr == nullptr)
      return newNodePtr;
   else
   {
      BinaryNode<T>* leftPtr = subTreePtr->getLeftChildPtr();
      BinaryNode<T>* rightPtr = subTreePtr->getRightChildPtr();

      if (getHeightHelper(leftPtr) > getHeightHelper(rightPtr)){
         rightPtr = balancedAdd(rightPtr , newNodePtr);
         subTreePtr->setRightChildPtr(rightPtr );
      } else {
         leftPtr = balancedAdd(leftPtr, newNodePtr);
         subTreePtr->setLeftChildPtr(leftPtr);
      }

      return subTreePtr;
   }
}
```

How to calculate the tree hight (max level)

```
template<class ItemType>
int BinaryNodeTree<ItemType>::
getHeightHelper(BinaryNode<ItemType>* subTreePtr) const
{
   if (subTreePtr == nullptr)
      return 0;
   else
      return 1 + max(getHeightHelper(subTreePtr->getLeftChildPtr()),
                     getHeightHelper(subTreePtr->getRightChildPtr()));
}
```

Summary

- A node of a tree references data and 2 (or more) “child” nodes.

- Array or link based implementations.

- The nature of the tree is recursive (a tree can be seen as a tree of trees..), therefore if is natural to use recursive algorithms for operating on it.

- Binary search trees are particularly efficient for search and insertion operations.

- **IDEA**: use a search binary tree to sort a sequence: **Tree Sort**.

  —> How could it work?
  —> What about its computational complexity?