

**04-630**

# **Data Structures and Algorithms for Engineers**

## **Lecture 10: Trees**

# Agenda

## Trees I

- Types of trees
- Binary Tree ADT
- Binary Search Tree
- Height Balanced Trees
  - AVL Trees
  - Red-Black Trees
- Optimal Code Trees
- Huffman's Algorithm

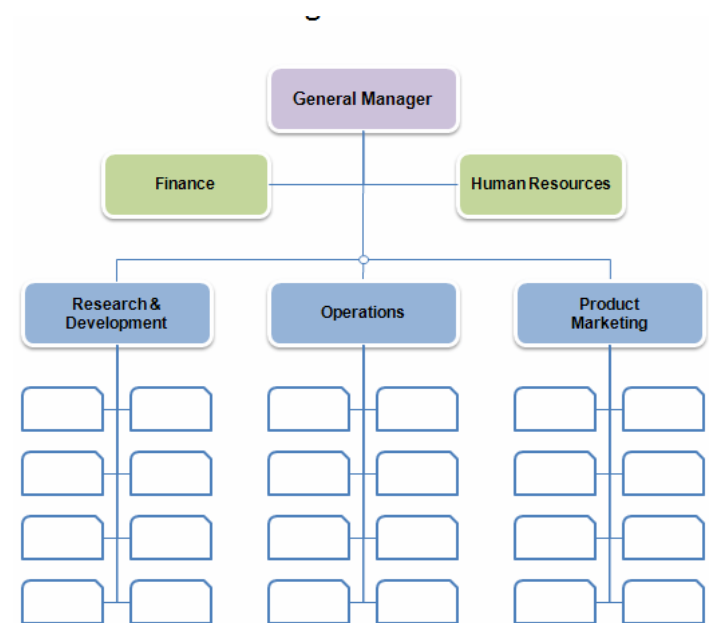
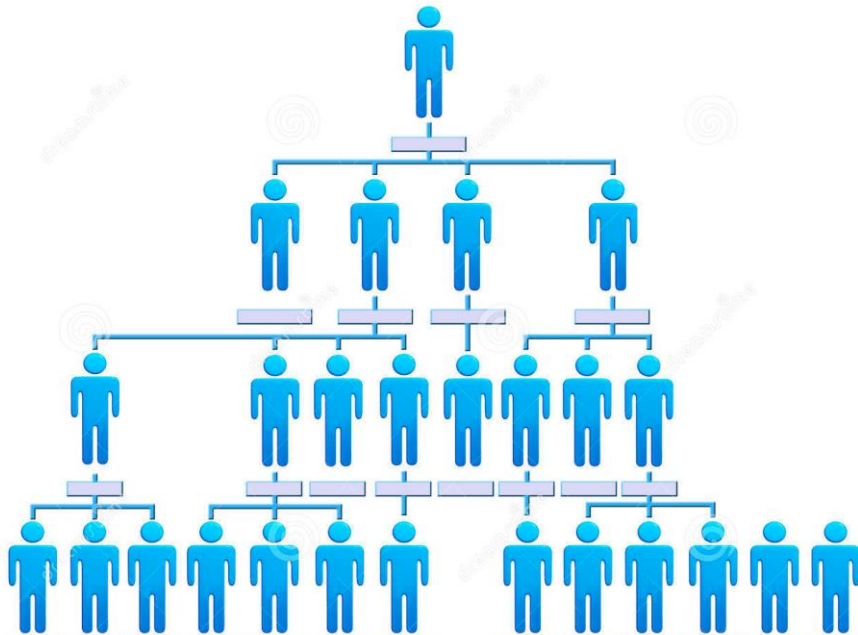
# TREE PRELIMINARIES

# Trees

- Trees are everywhere
- Hierarchical method of structuring data
- Uses of trees:
  - genealogical tree
  - organizational tree
  - expression tree
  - binary search tree
  - decision tree

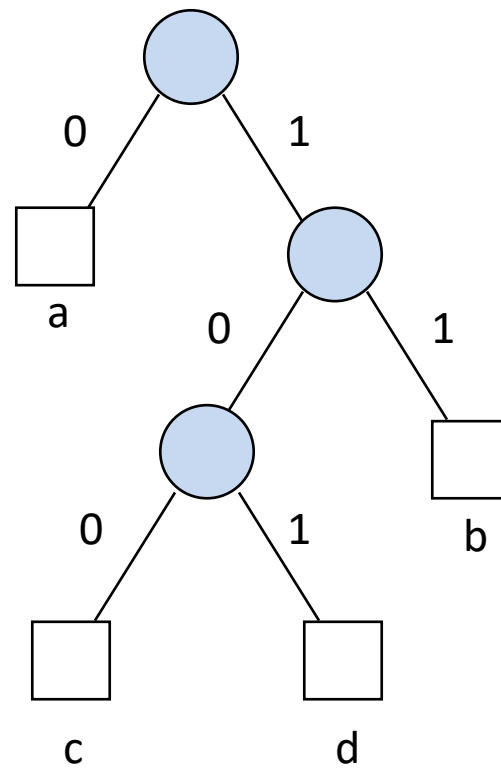
# Uses of Trees

## Organization Tree



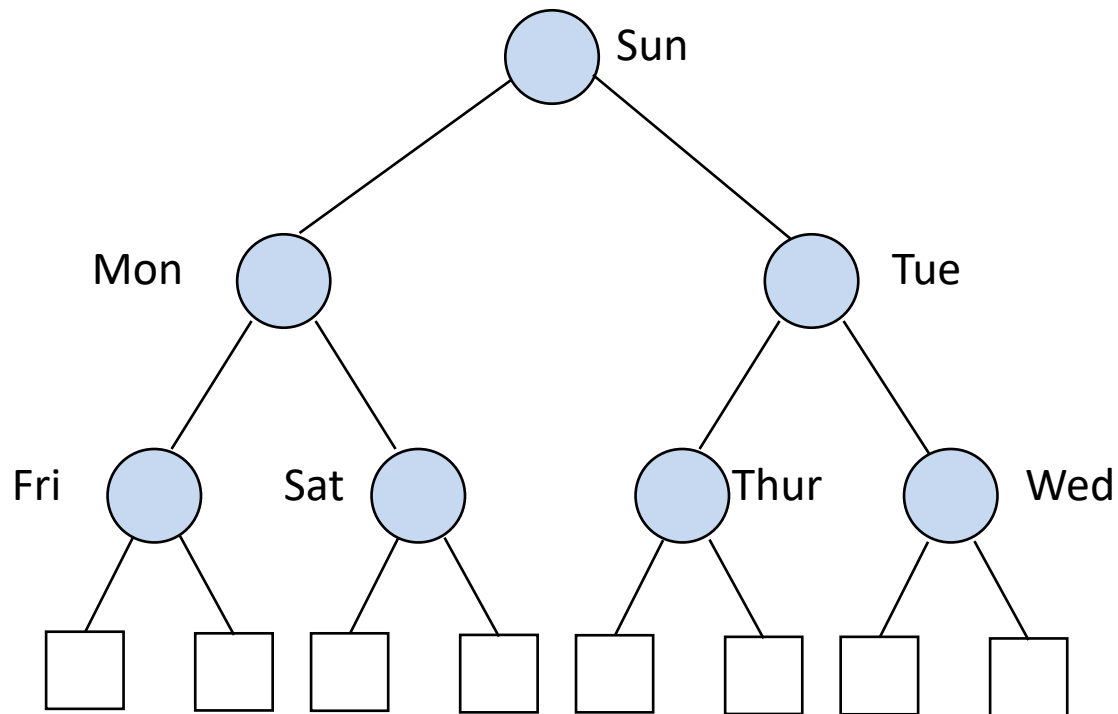
# Uses of Trees

## Code Tree



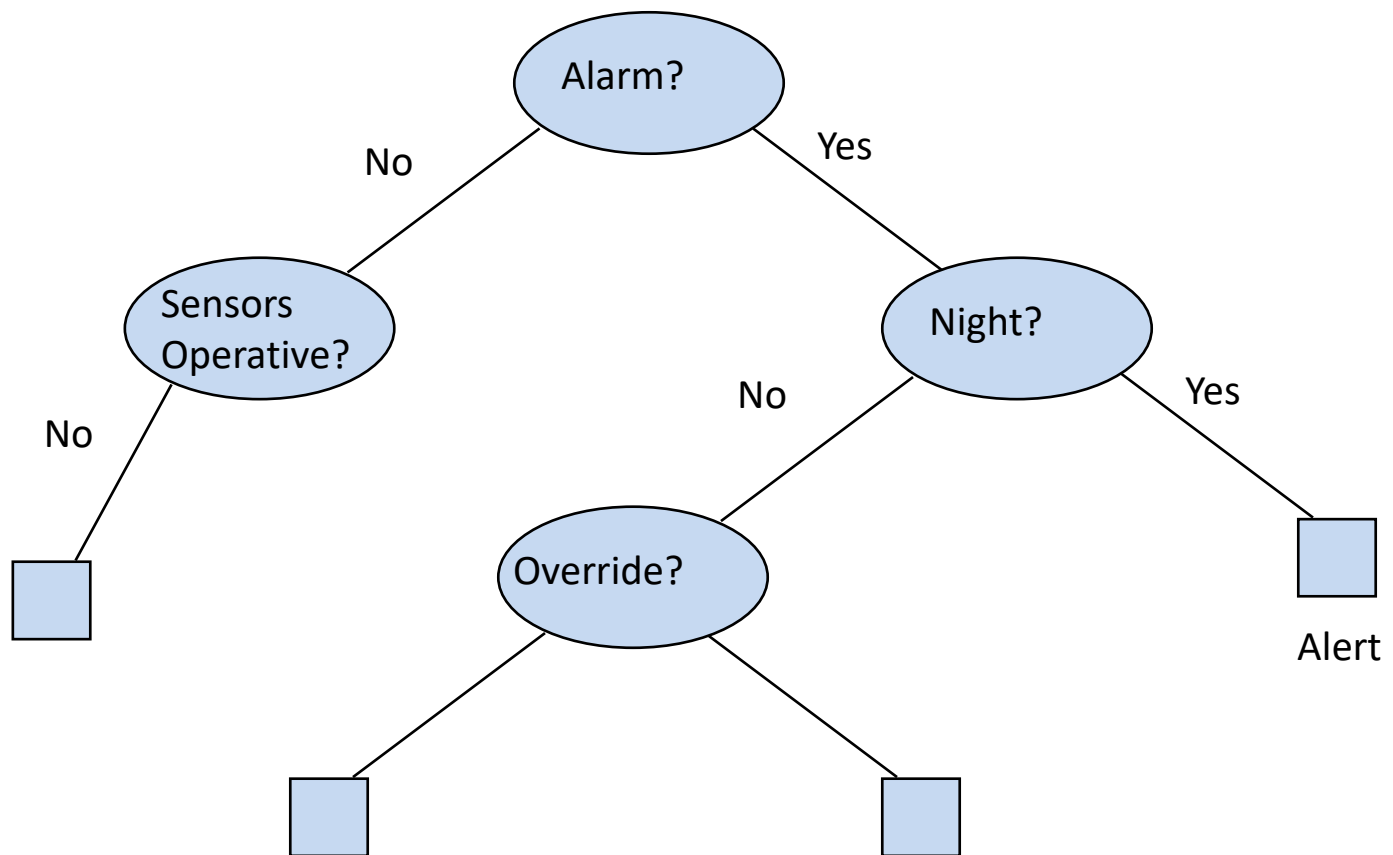
# Uses of Trees

## Binary Search Tree



# Uses of Trees

## Decision Tree





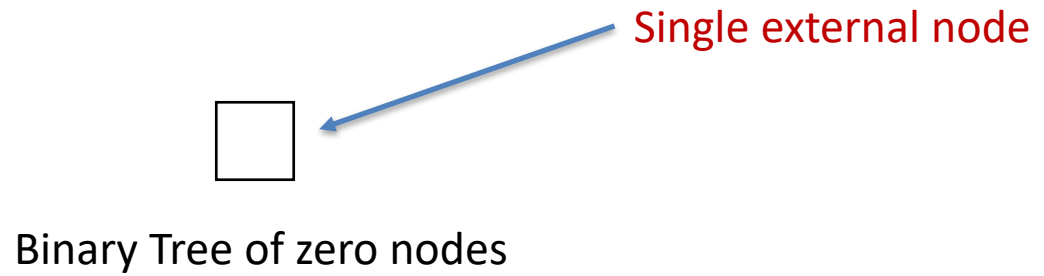
# Trees

- Fundamentals
- Traversals
- Display
- Representation
- Abstract Data Type (ADT) approach
- Emphasis on binary tree
- Also multi-way trees, forests, orchards

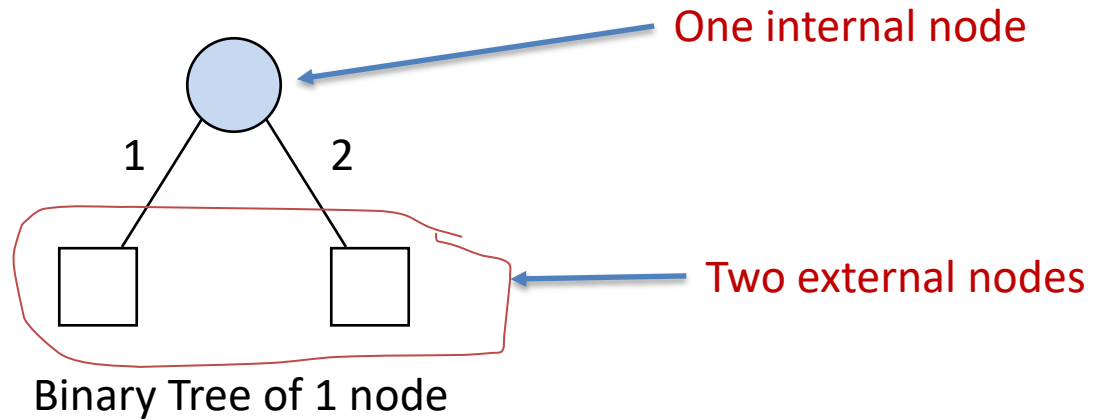
# Tree Definitions

- A **binary tree**  $T$  of  $n$  nodes,  $n \geq 0$ ,
  - either is empty, if  $n = 0$
  - or consists of a **root node**  $u$  and two binary trees  $u(1)$  and  $u(2)$  of  $n_1$  and  $n_2$  nodes, respectively, such that  $n = 1 + n_1 + n_2$
- We say that  $u(1)$  is the **first or left subtree** of  $T$ , and  $u(2)$  is the **second or right subtree** of  $T$

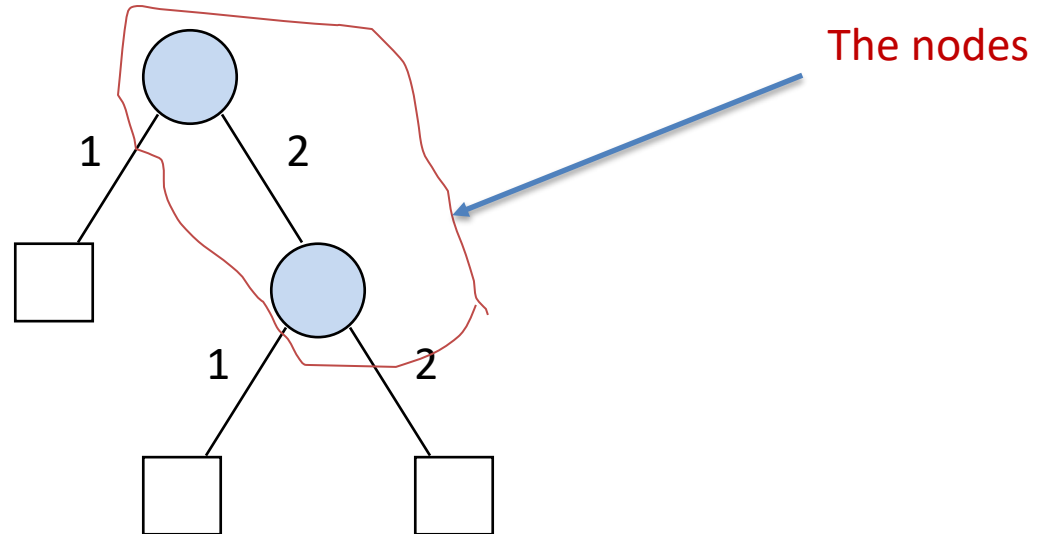
# Binary Tree



# Binary Tree

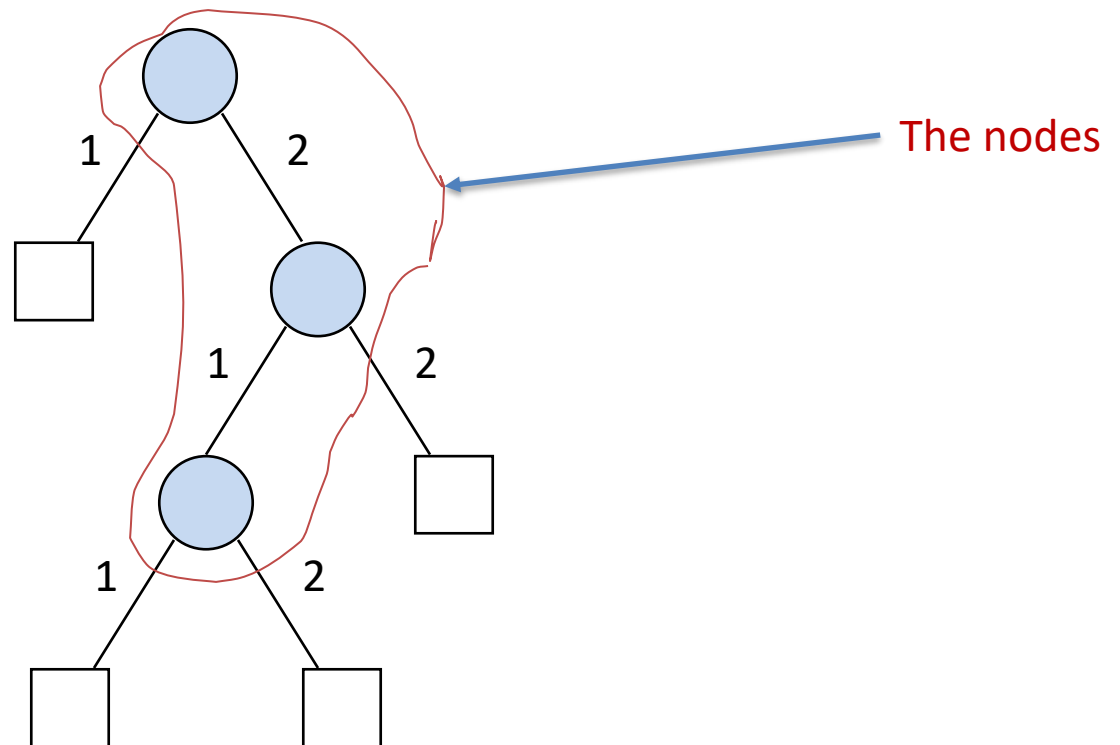


# Binary Tree



Binary Tree of 2 nodes

# Binary Tree

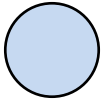


Binary Tree of 3 nodes

# Binary Tree



External nodes - have no subtrees

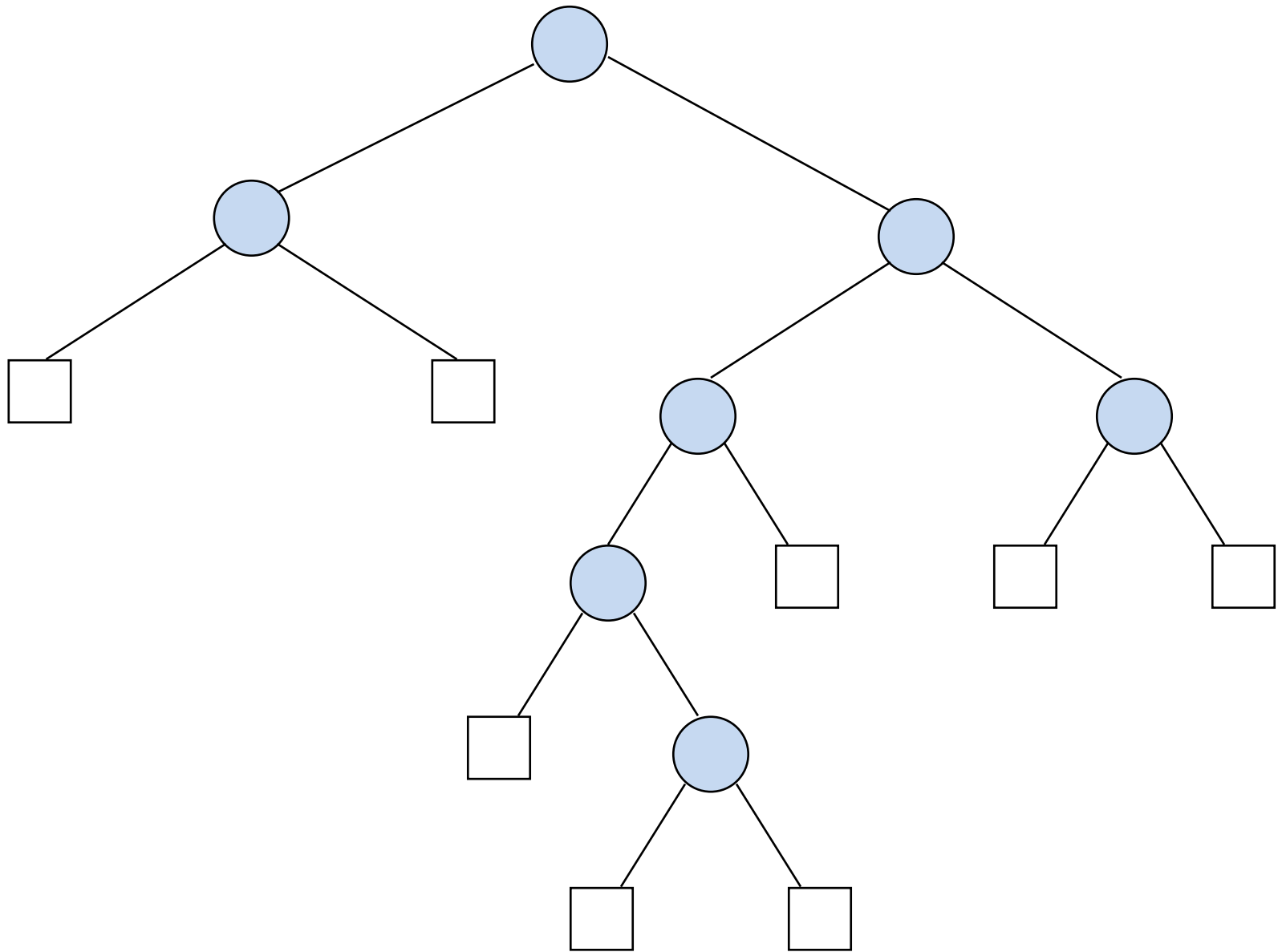


Internal nodes - always have two subtrees

# Binary Tree Terminology

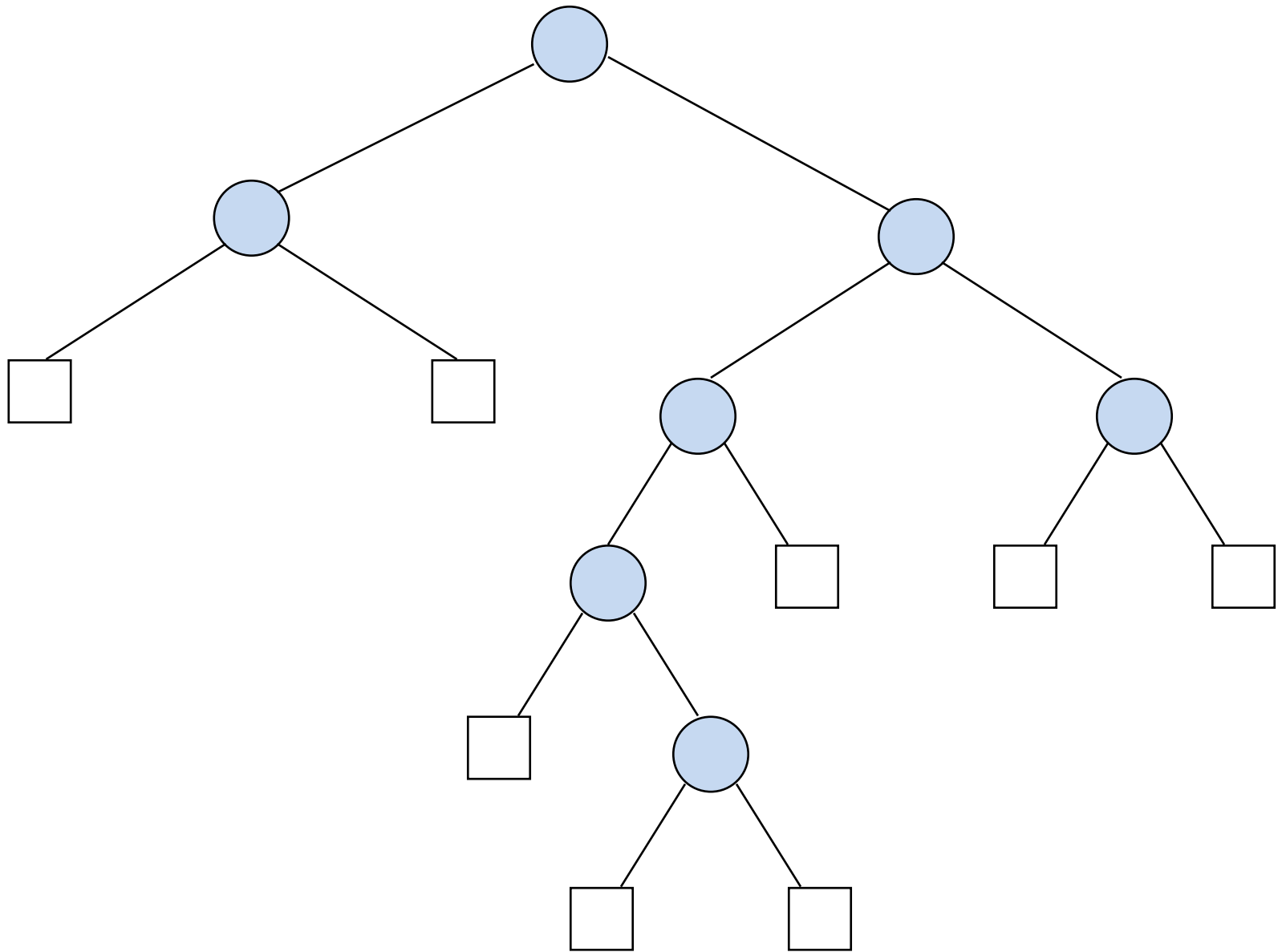
- Let  $T$  be a binary tree with root  $u$
- Let  $v$  be any node in  $T$
- If  $v$  is the root of either  $u(1)$  or  $u(2)$ , then we say  $u$  is the **parent** of  $v$  and that  $v$  is the **child** of  $u$
- If  $w$  is also a child of  $u$ , and  $w$  is distinct from  $v$ , we say that  $v$  and  $w$  are **siblings**





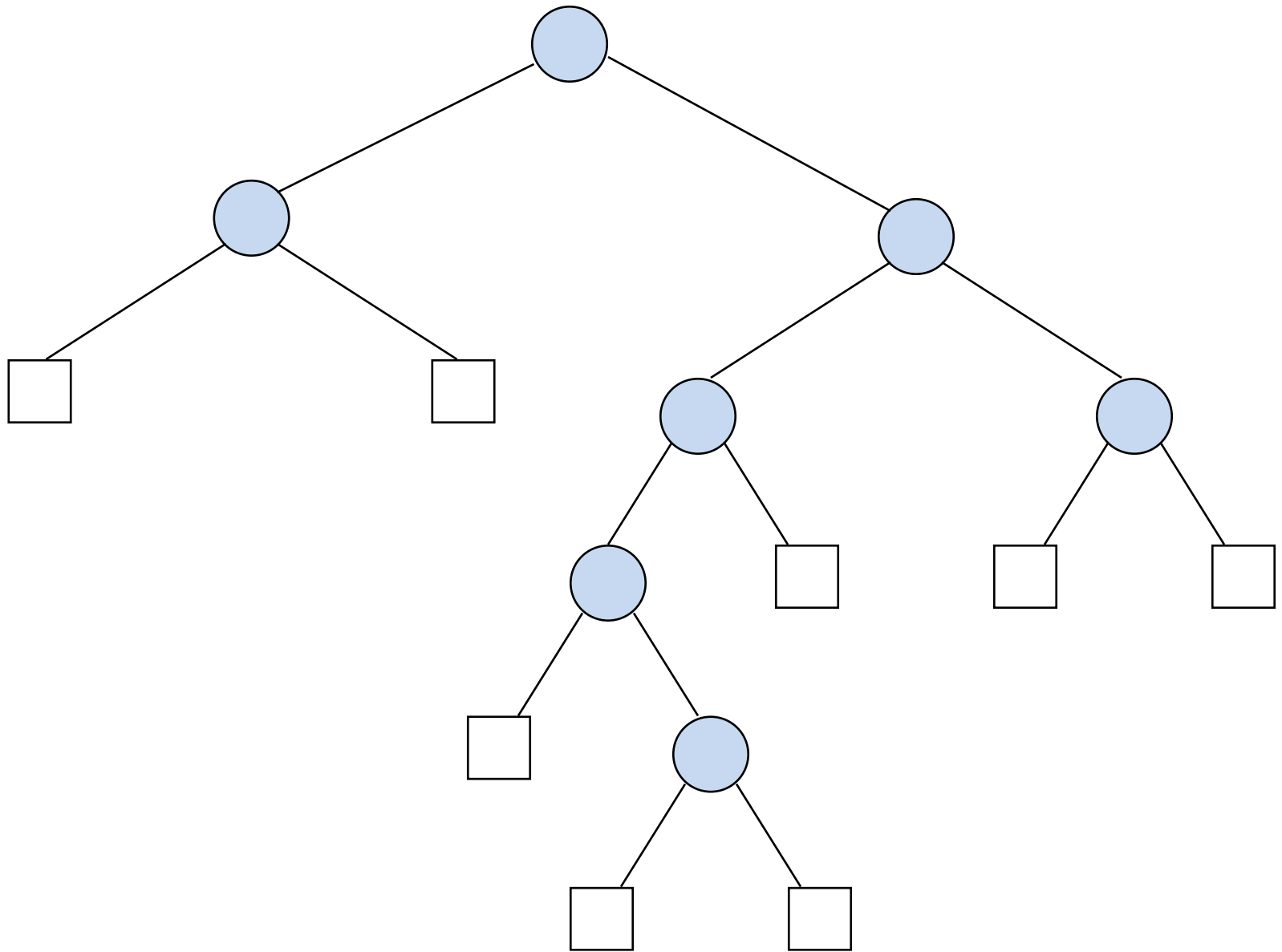
# Binary Tree Terminology

- If  $v$  is the root of  $u(i)$
- then  $v$  is the  $i^{\text{th}}$  child of  $u$ ;  
 $u(1)$  is the **left child** and  $u(2)$  is the **right child**
- Also have **grandparents** and **grandchildren**



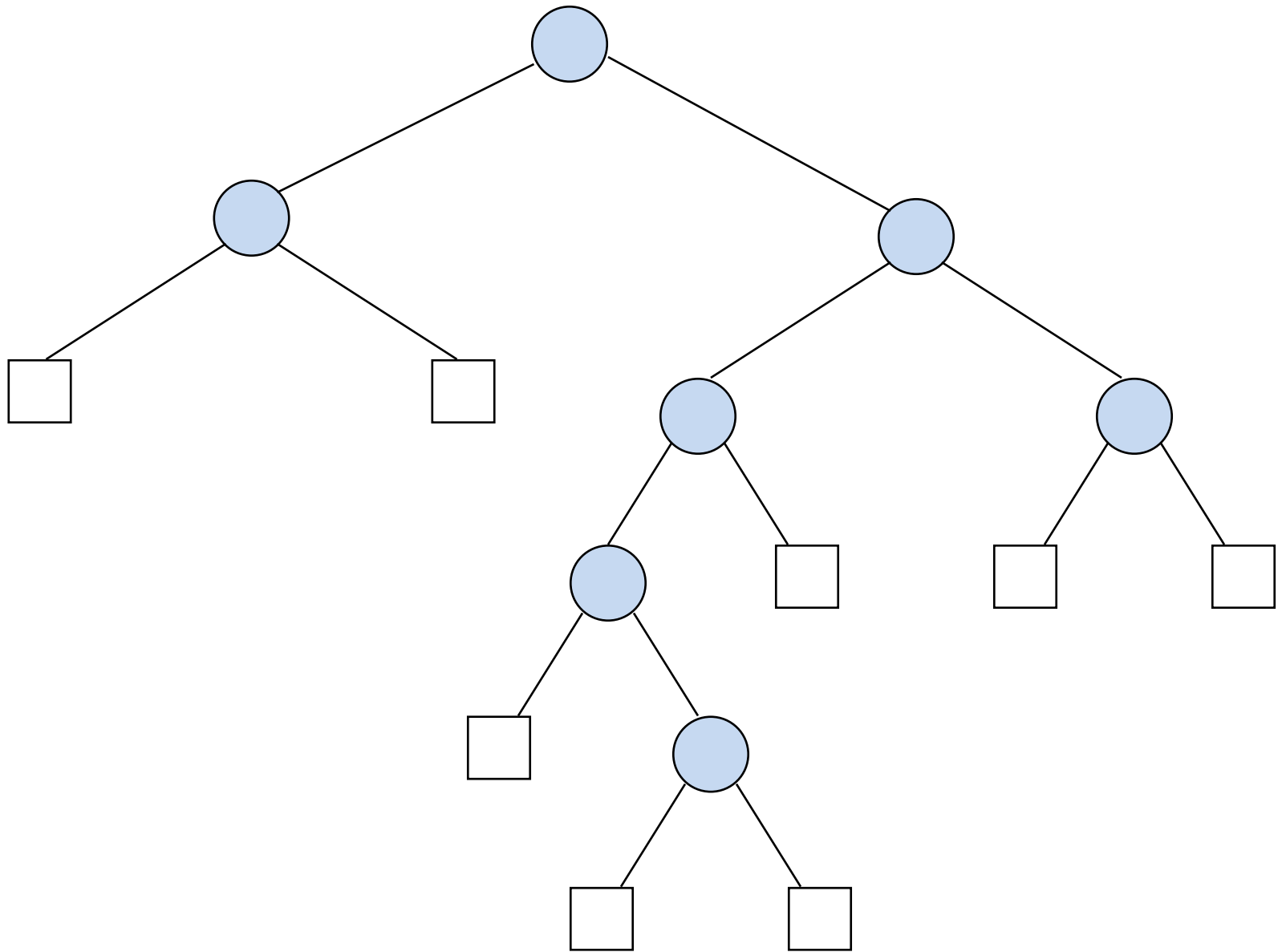
# Binary Tree Terminology

- Given a binary tree  $T$  of  $n$  nodes,  $n \geq 0$
- then  $v$  is a **descendent** of  $u$  if either
  - $v$  is equal to  $u$   
or
  - $v$  is a child of some node  $w$  and  $w$  is a descendant of  $u$
- We write  $v \text{ desc}_T u$
- $v$  is a **proper descendent** of  $u$  if  $v$  is a descendant of  $u$  and  $v \neq u$



# Binary Tree Terminology

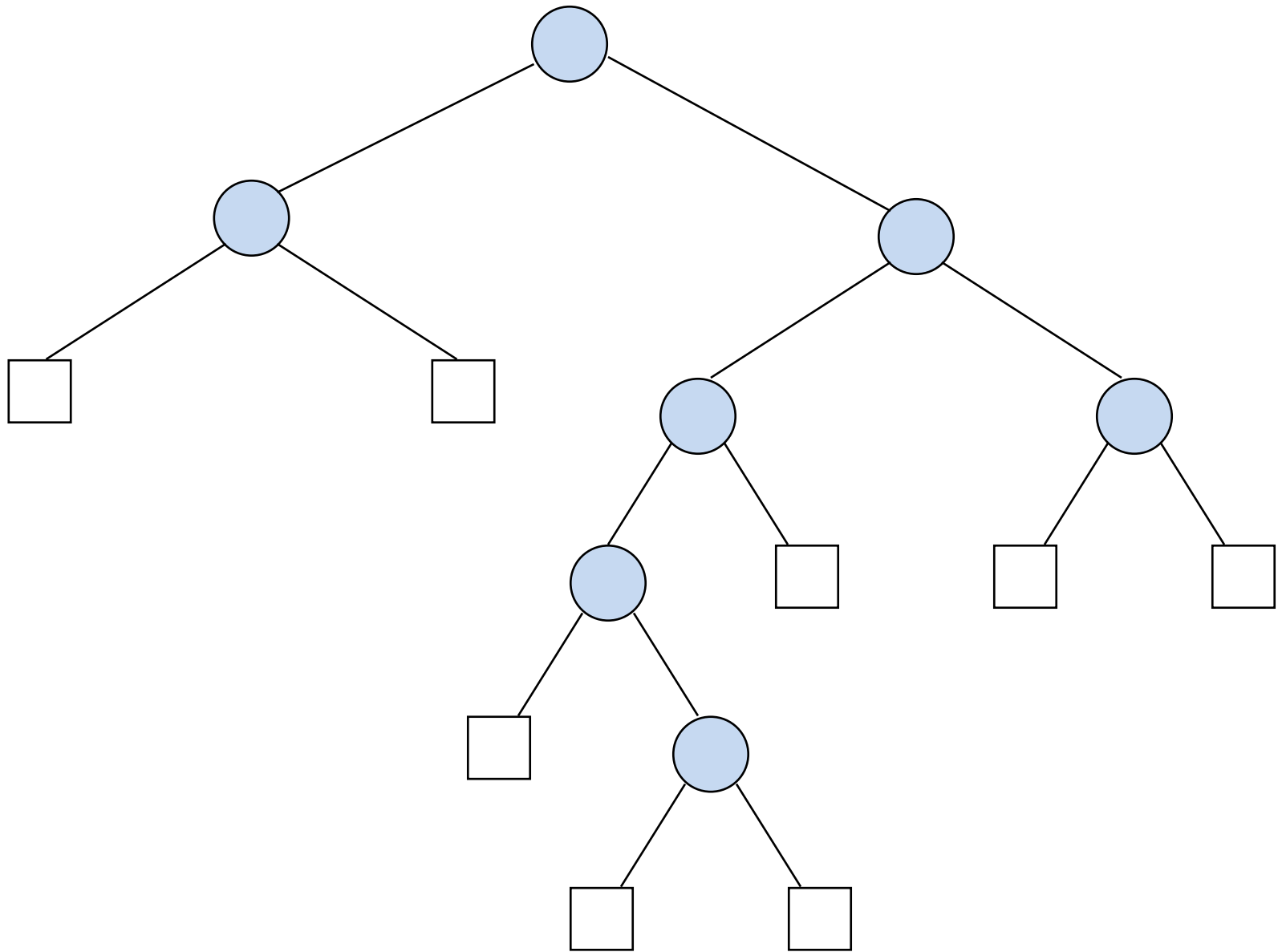
- Given a binary tree  $T$  of  $n$  nodes,  $n \geq 0$
- then  $v$  is a **left descendent** of  $u$  if either
  - $v$  is equal to  $u$   
or
  - $v$  is a left child of some node  $w$  and  $w$  is a left descendant of  $u$
- We write  $v \text{ ldesc}_T u$
- Similarly we have  $v \text{ rdesc}_T u$



# Binary Tree Terminology

- $left_T$  relates nodes **across** a binary tree rather than up and down a binary tree
- Given two nodes  $u$  and  $v$  in a binary tree  $T$ , we say that  $v$  is **to the left** of  $u$  if there is a new node  $w$  in  $T$  such that  $v$  is a left descendant of  $w$  and  $u$  is a right descendant of  $w$
- We denote this relation by  $left_T$  and write  $v left_T u$



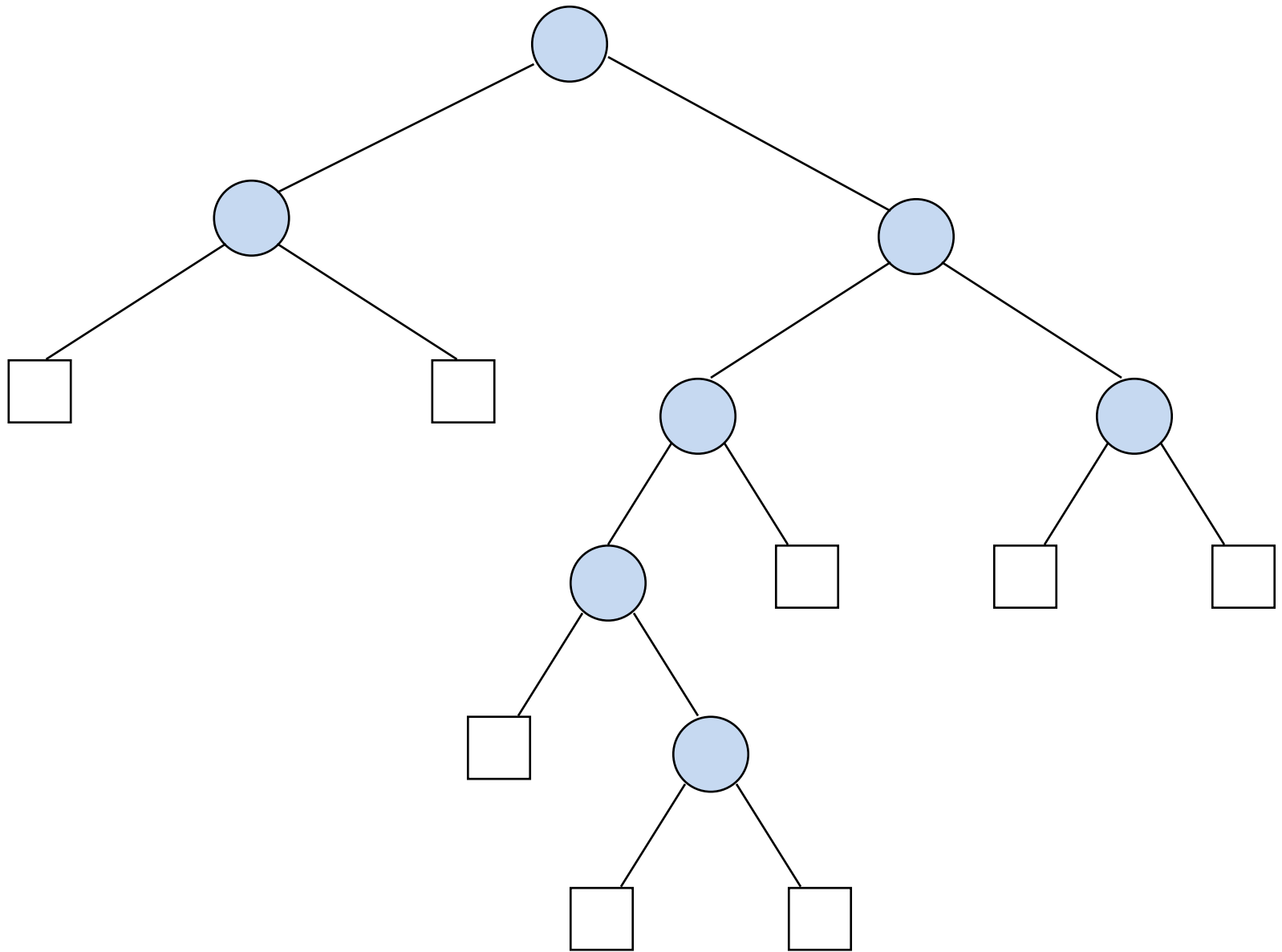


# Binary Tree Terminology

- The external nodes of a tree define its **frontier**
- We can count the number of nodes in a binary tree in three ways:
  - Number of internal nodes
  - Number of external nodes
  - Number of internal and external nodes
- The number of internal nodes is the **size** of the tree

# Binary Tree Terminology

- Let  $T$  be a binary tree of size  $n$ ,  $n \geq 0$ ,
- Then, the number of external nodes of  $T$  is  $n + 1$

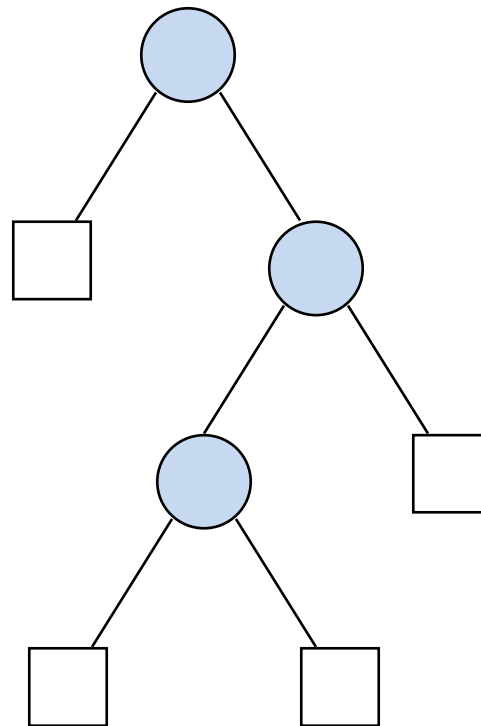


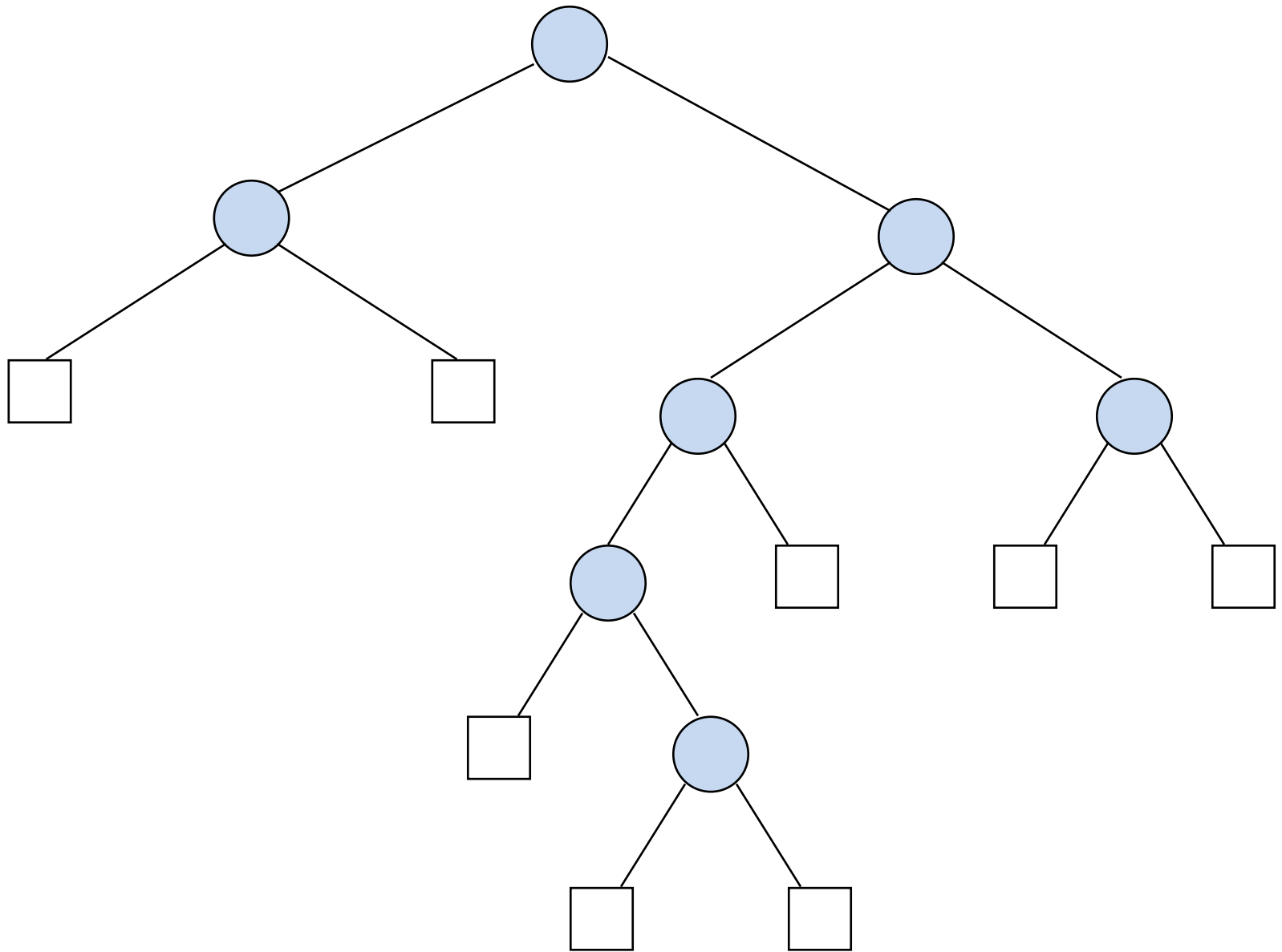
# Binary Tree Terminology

- The **height** of  $T$  is defined recursively as
  - 0 if  $T$  is empty and
  - $1 + \max(\text{height}(T_1), \text{height}(T_2))$  otherwise,  
where  $T_1$  and  $T_2$  are the subtrees of the root
- The height of a tree is the length of a longest chain of descendants

# Binary Tree Terminology

- Height Numbering
  - Number all external nodes 0
  - Number each internal node to be one more than the maximum of the numbers of its children
  - Then the number of the root is the height of  $T$
- The height of a node  $u$  in  $T$  is the height of the subtree rooted at  $u$



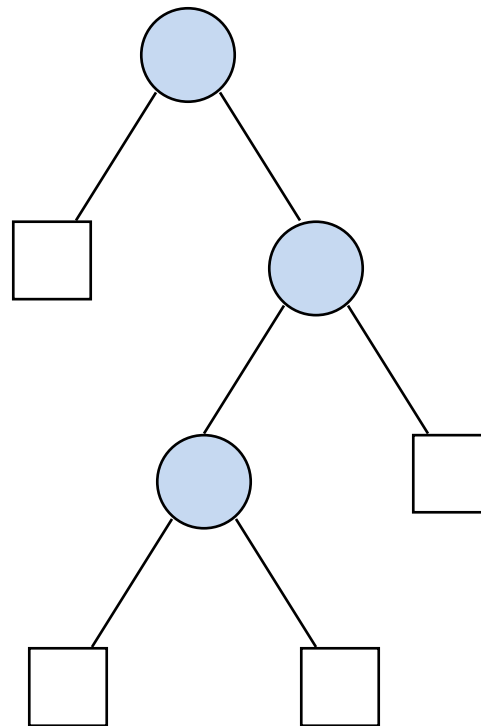


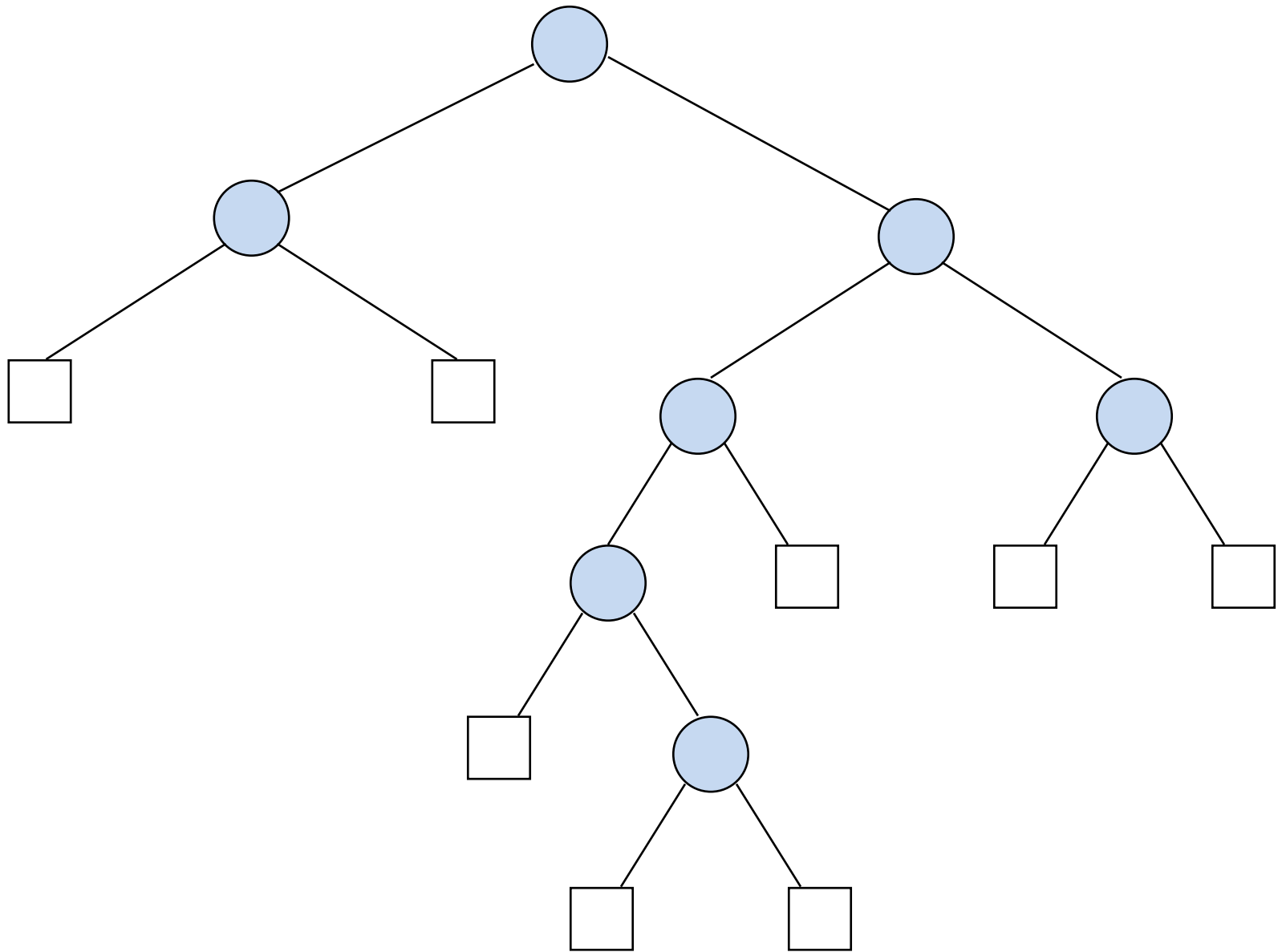


# Binary Tree Terminology

## Levels of nodes

- The level of a node in a binary tree is computed as follows
- Number the root node 0
- Number every other node to be 1 more than its parent
- Then the number of a node  $v$  is that node's level
- The level of  $v$  is the number of branches on the path from the root to  $v$

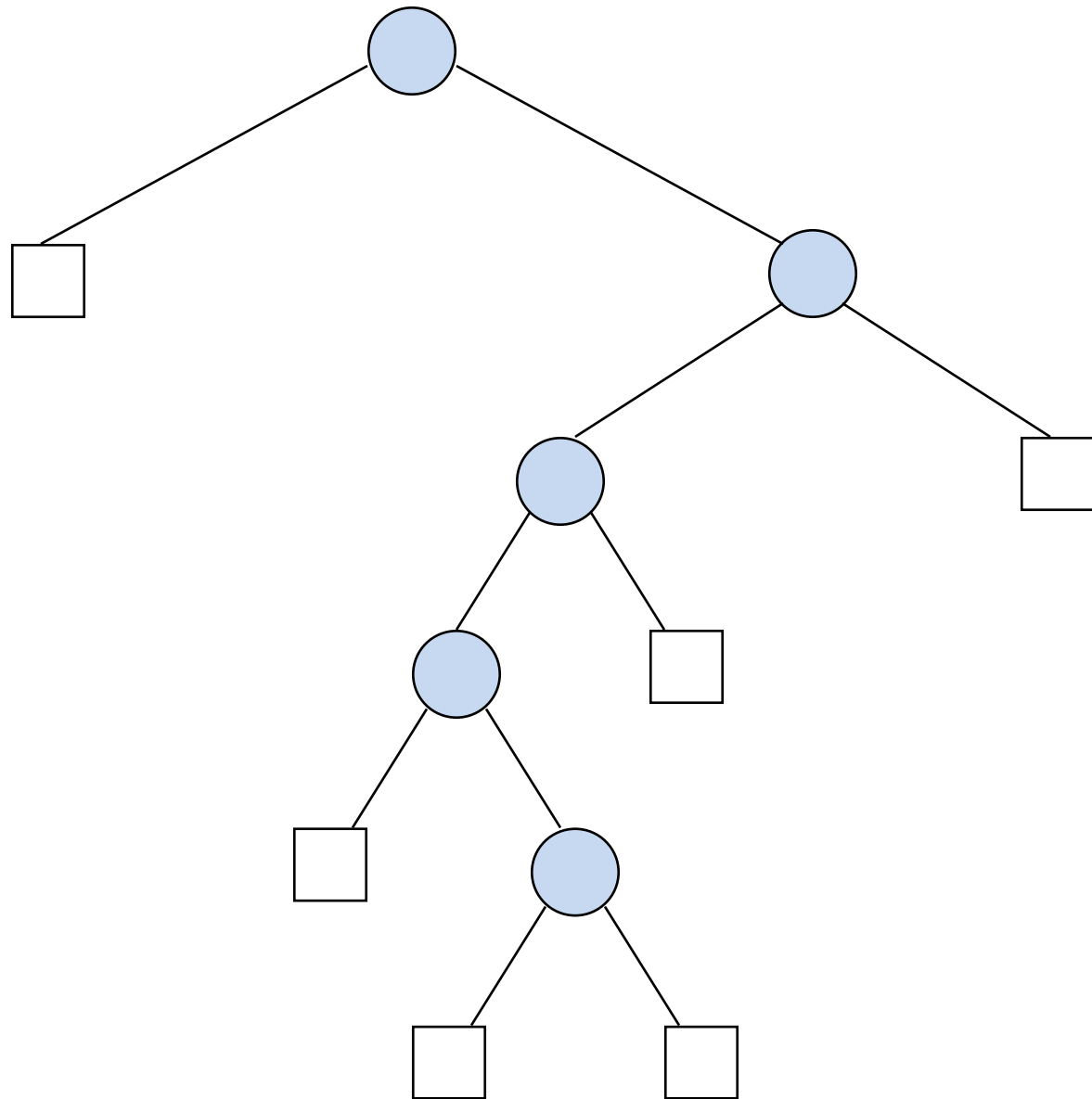




# Binary Tree Terminology

## Skinny Trees

- every internal node has at most one internal child

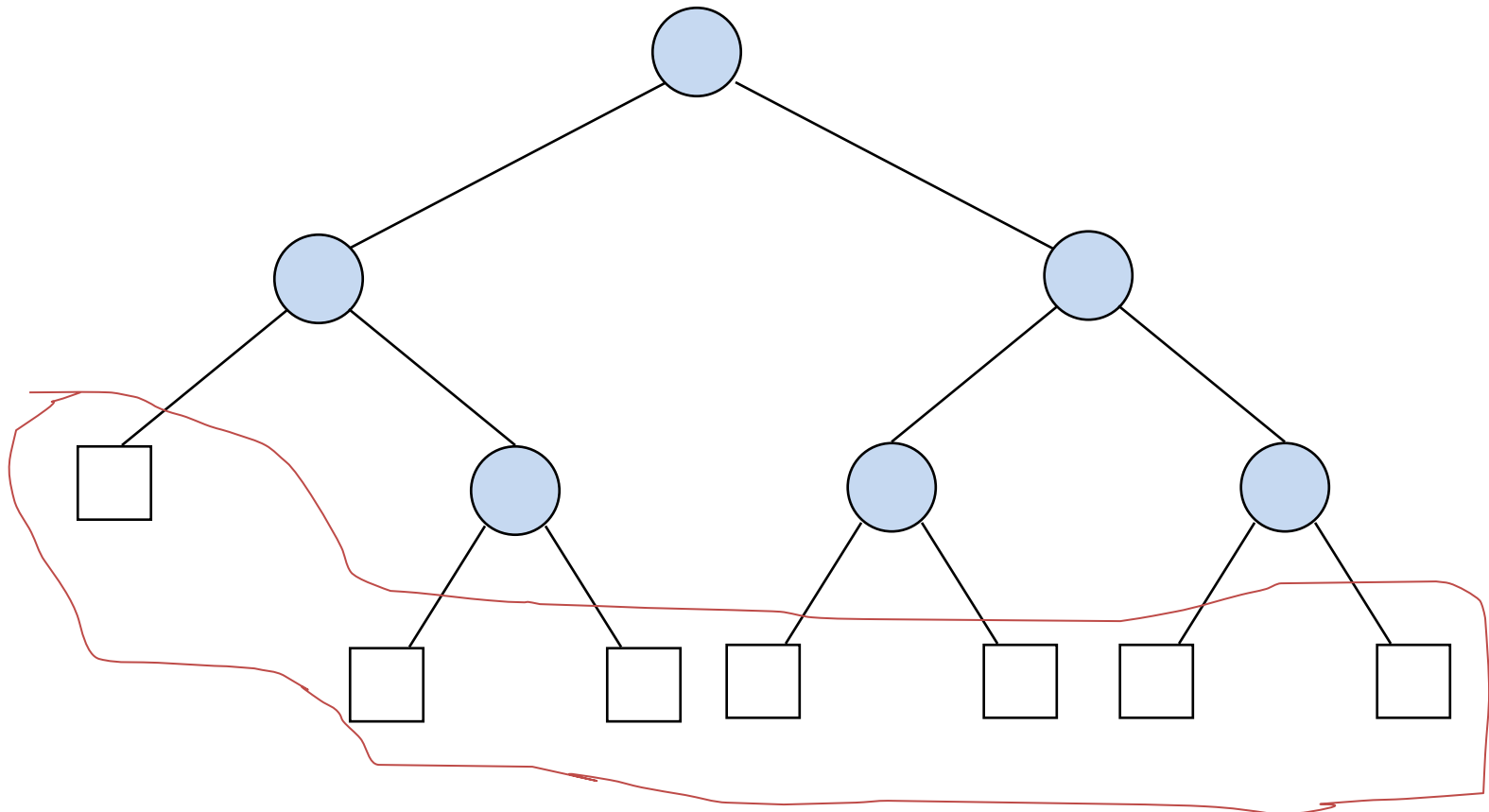


# Binary Tree Terminology

## Complete Binary Trees (Fat Trees)

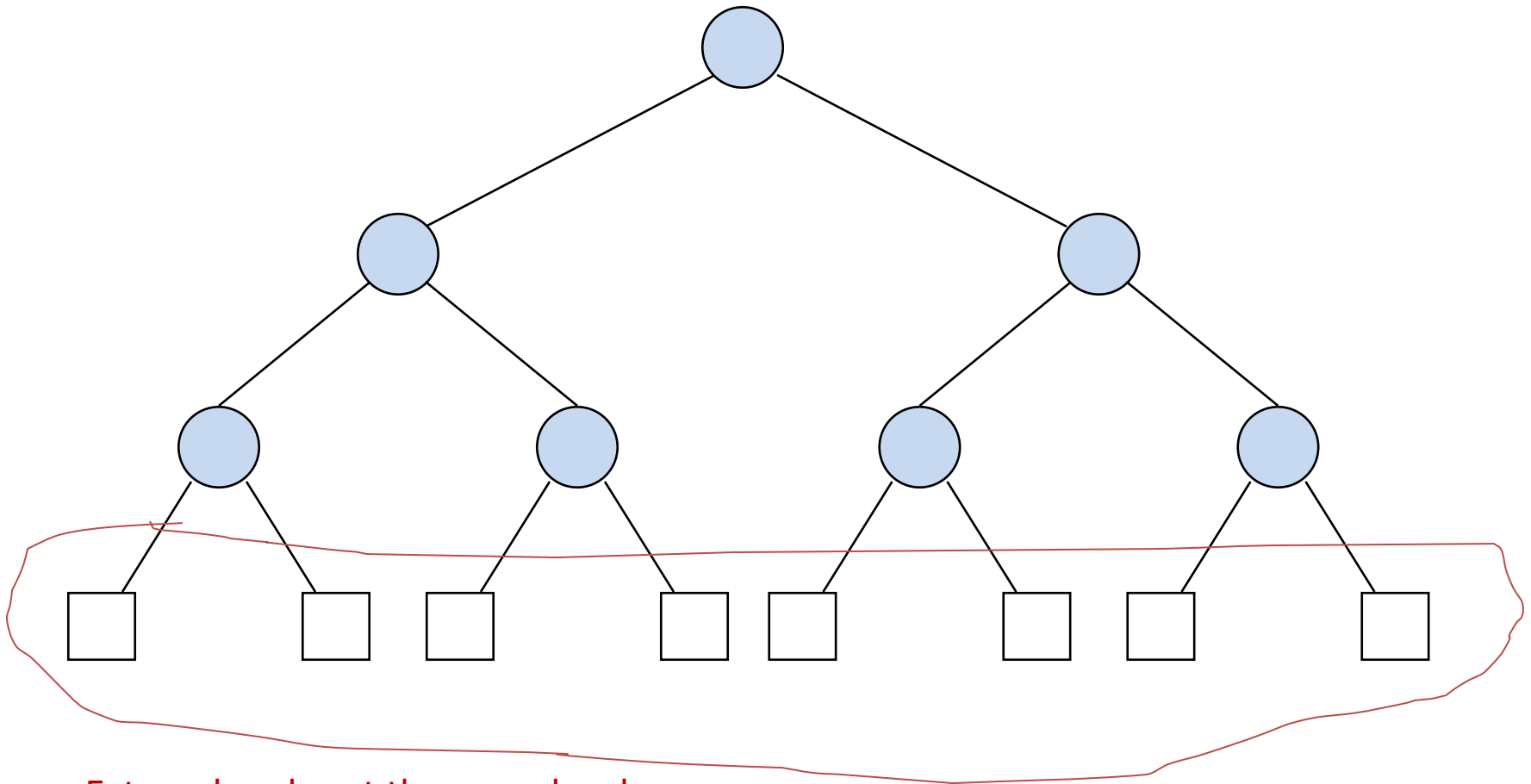
- the external nodes appear on at most two adjacent levels
- **Perfect Trees**: complete trees having all their external nodes on one level
- **Left-complete** Trees: the internal nodes on the lowest level is in the leftmost possible position
- Skinny trees are the highest possible trees
- Complete trees are the lowest possible trees

# Complete Tree



External nodes on at most two adjacent levels

# Perfect Tree

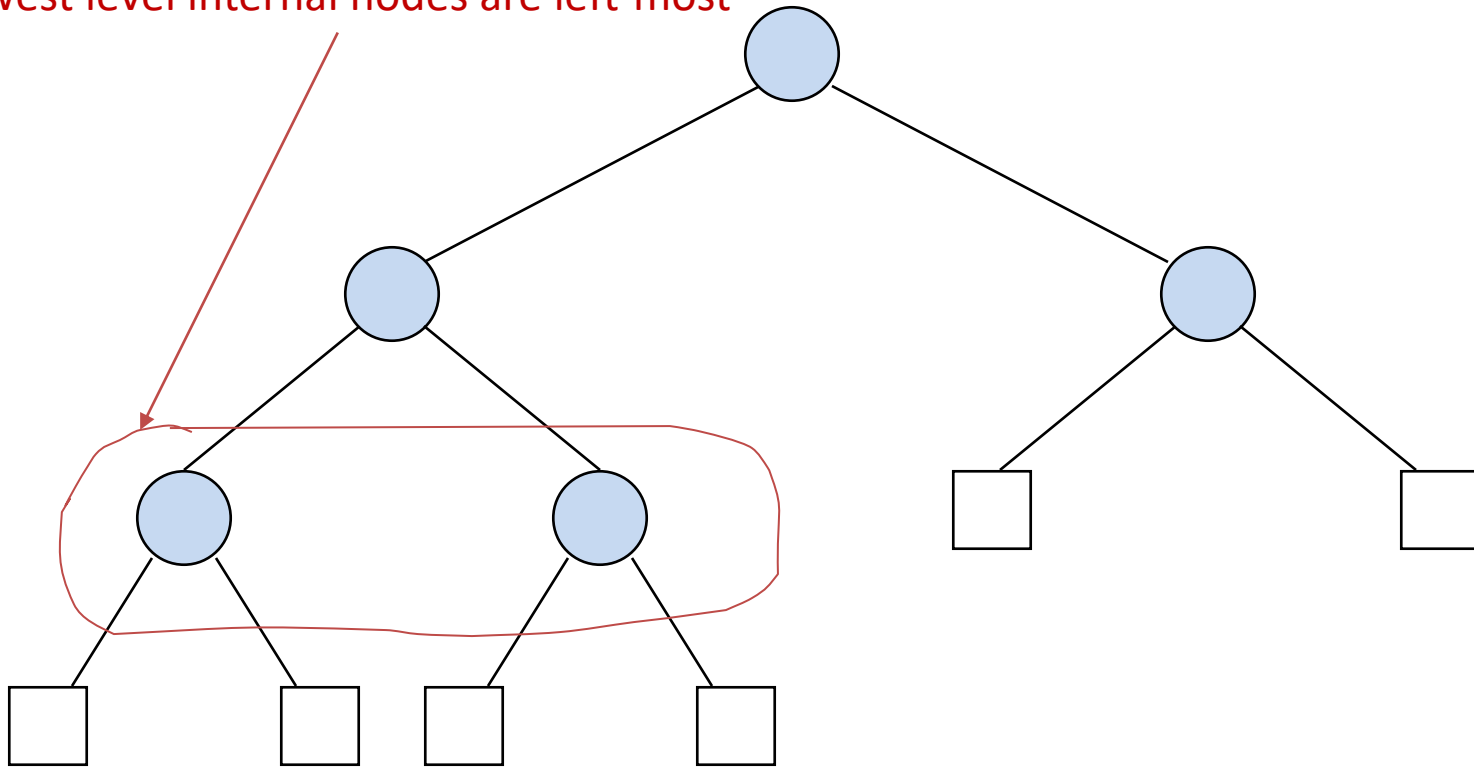


External nodes at the same level



# Left-Complete Tree

Lowest level internal nodes are left-most



# Binary Tree Terminology

- A binary tree of height  $h \geq 0$   
has size at least  $h$
- A binary tree of height at most  $h \geq 0$   
has size at most  $2^h - 1$
- A binary tree of size  $n \geq 0$   
has height at most  $n$
- A binary tree of size  $n \geq 0$   
has height at least  $\lceil \log (n + 1) \rceil$

# Multiway Trees

Multiway trees are defined in a similar way to binary trees, except that the **degree** (**the maximum number of children**) is no longer restricted to the value 2

# Multiway Trees

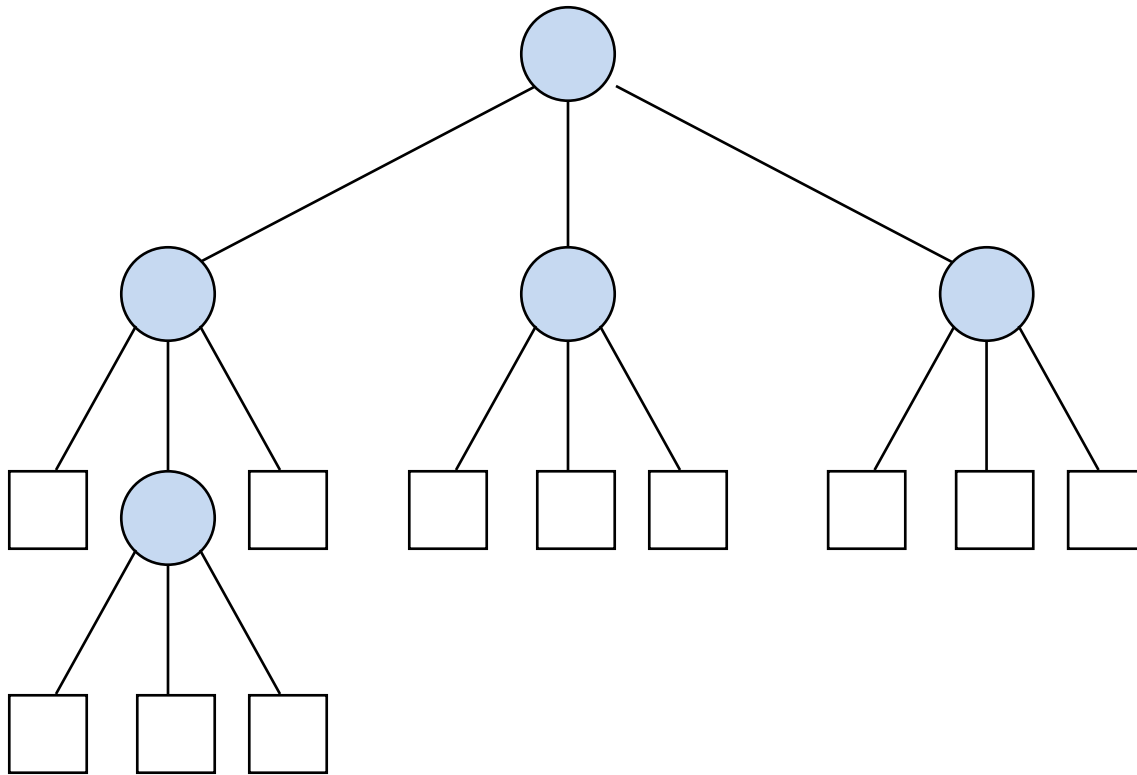
A multiway tree  $T$  of  $n$  internal nodes,  $n \geq 0$ ,

- either is empty, if  $n = 0$ ,
- or consists of
  - a root node  $u$ ,
  - an integer  $d_u \geq 1$ , the degree of  $u$ ,
  - and multiway trees  $u(1)$  of  $n_1$  nodes, ...,  $u(d_u)$  of  $n_{d_u}$  nodes such that  $n = 1 + n_1 + \dots + n_{d_u}$

# Multiway Trees

A multiway tree  $T$  is a **d-ary tree**,  
for some  $d > 0$ ,  
if  $d_u = d$ , for all internal nodes  $u$  in  $T$

# d-ary Tree



# Multiway Trees

- A multiway tree  $T$  is a **(a, b)-tree**,  
if  $1 \leq a \leq d_u \leq b$ , for all  $u$  in  $T$
- Every binary tree is a (2, 2)-tree, and vice versa

# **BINARY TREE SPECIFICATION**



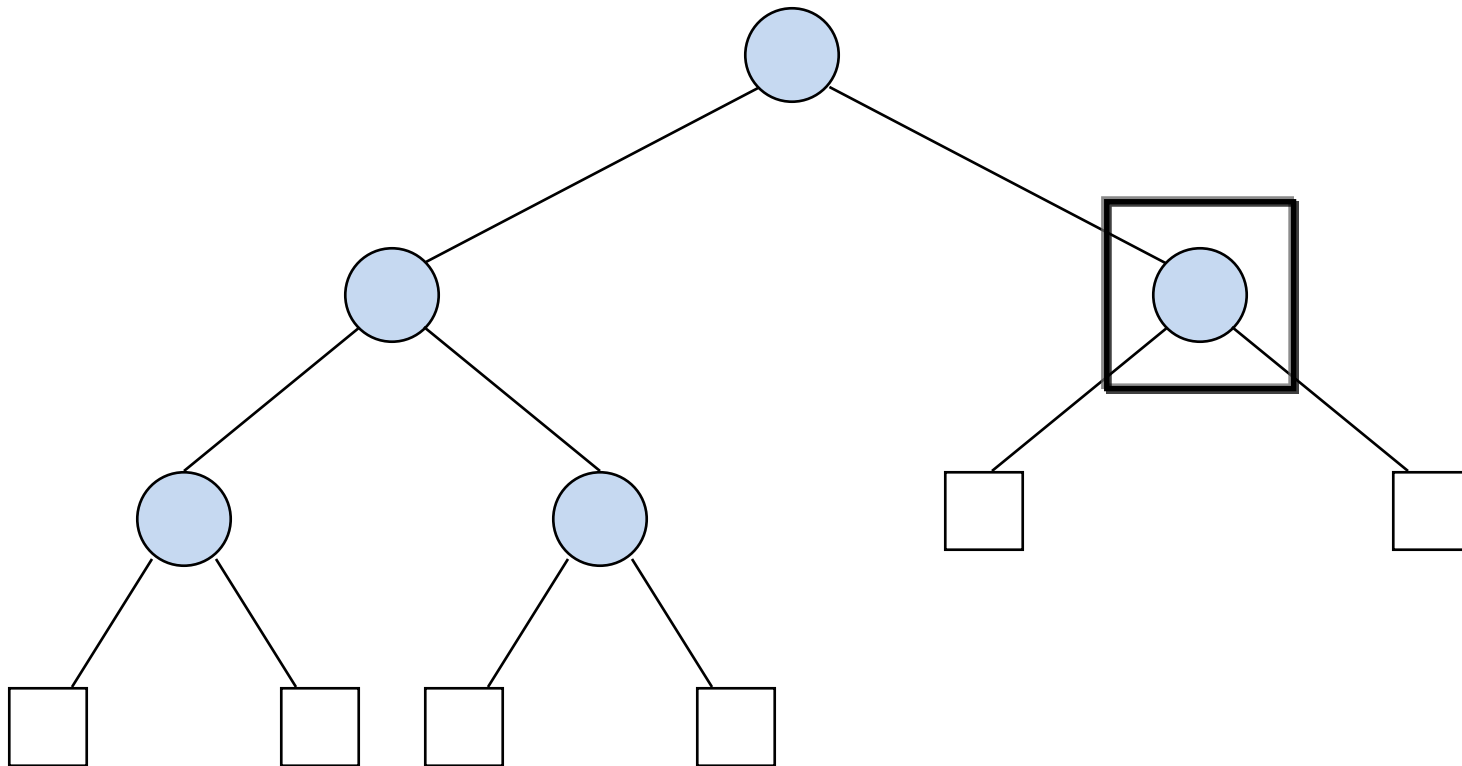
# BINARY\_TREE & TREE Specification

- So far, no values associated with the nodes of a tree
- Now we want to introduce an ADT called BINARY\_TREE
  - Has value of type *elementtype*
  - Sometimes
    - has value of type *intelementtype* associated with the internal nodes
    - has value of type *extelementtype* associated with the external nodes
- These value don't have any effect on BINARY\_TREE operations

# BINARY\_TREE & TREE Specification

- BINARY\_TREE has explicit windows and window-manipulation operations
- A window allows us to 'see' the value in a node (and to gain access to it)
- Windows can be positioned over any **internal** or **external** node
- Windows can be moved from **parent** to **child**
- Windows can be moved from **child** to **parent**

# Window



# BINARY\_TREE & TREE Specification

- Let **BT** denote the set of values of BINARY\_TREE of *elementtype*
- Let **E** denote the set of values of type *elementtype*
- Let **W** denote the set of values of type *windowtype*
- Let **B** denote the set of Boolean values *true* and *false*

# BINARY\_TREE Operations

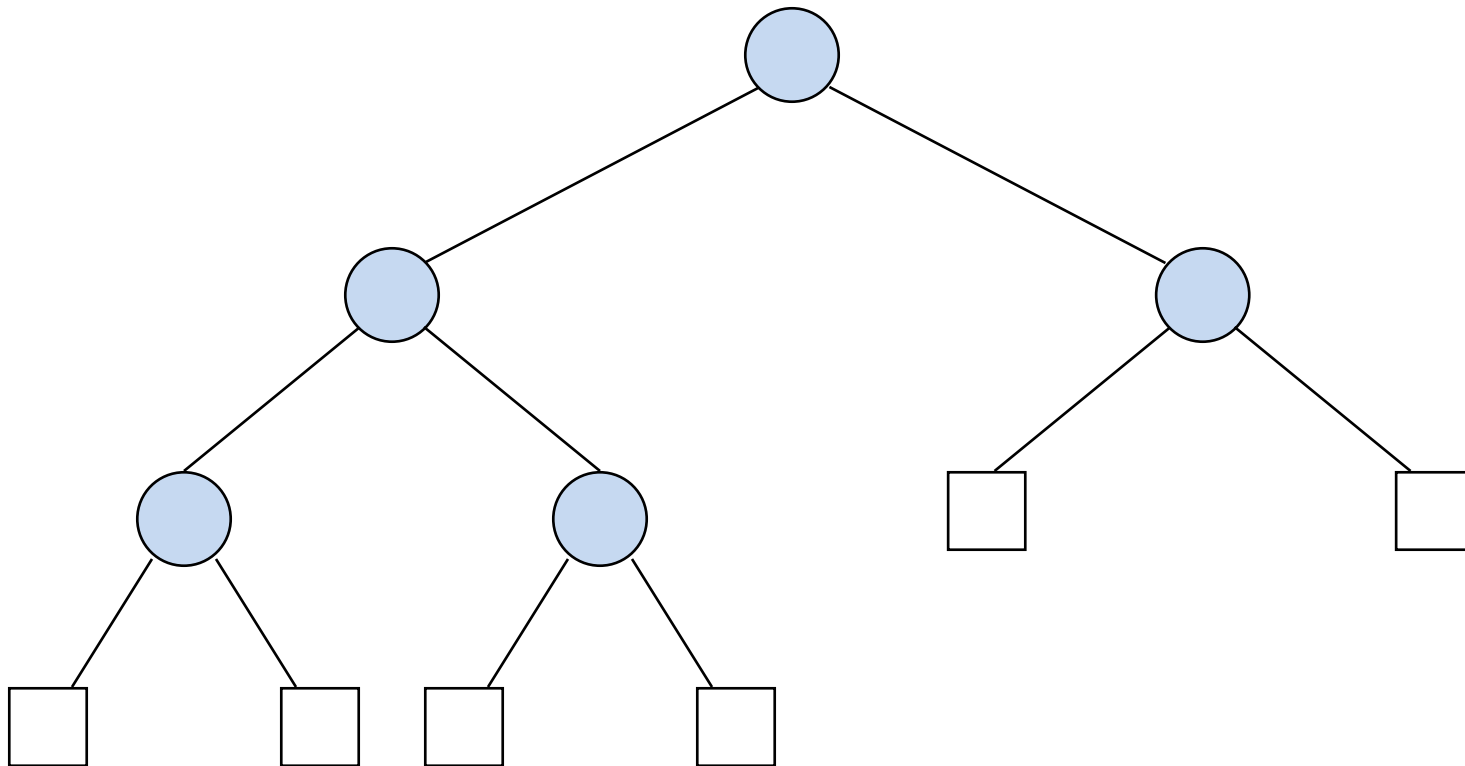
*Empty*:  $BT \rightarrow BT$  :

The function *Empty*( $T$ ) is an empty binary tree; if necessary, the tree is deleted

*IsEmpty*:  $BT \rightarrow B$  :

The function value *IsEmpty*( $T$ ) is *true* if  $T$  is empty; otherwise it is false

## Example



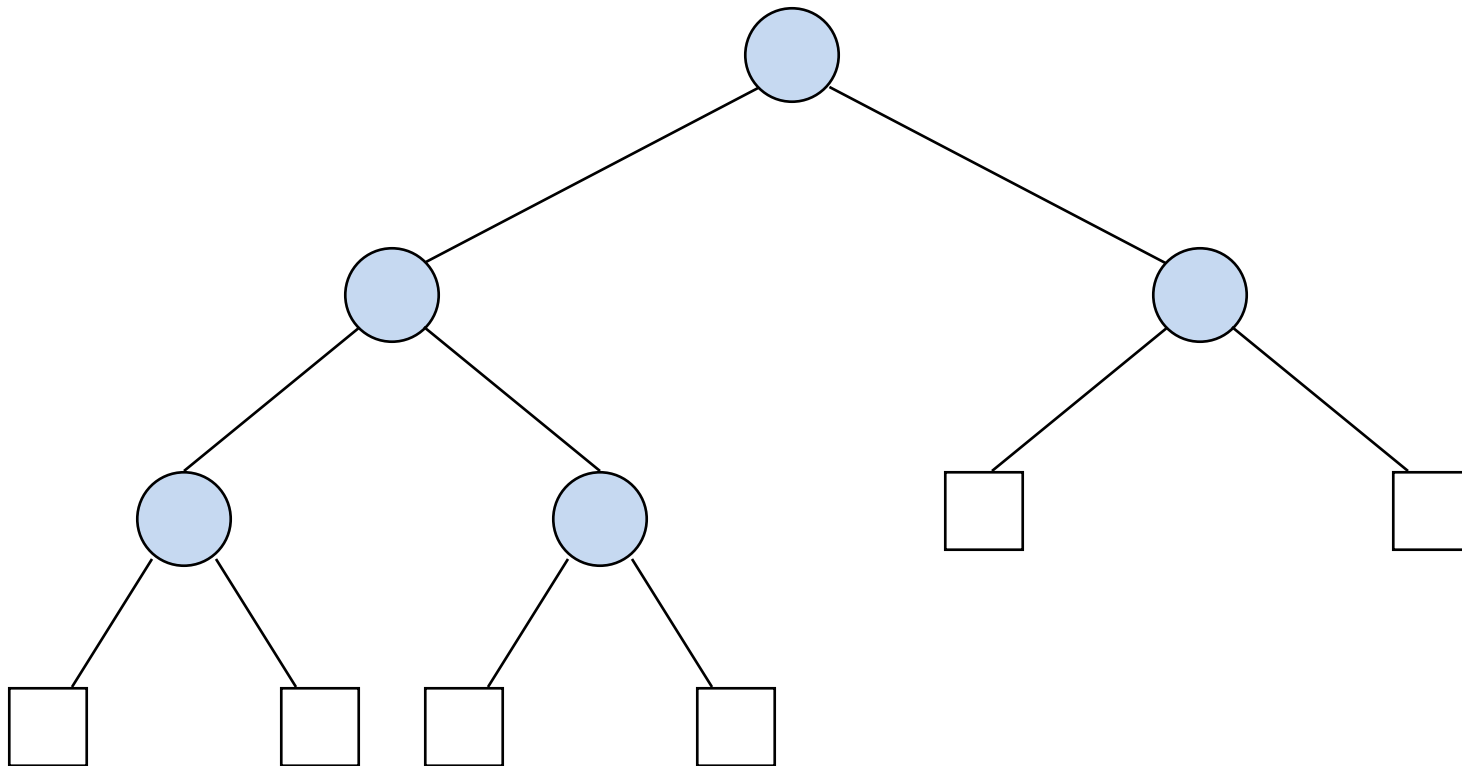
# BINARY\_TREE Operations

*Root*: BT  $\rightarrow$  W :

The function value  $Root(T)$  is the window position of *the single external node* if  $T$  is empty;

otherwise it is the window position of the *root of  $T$*

## Example





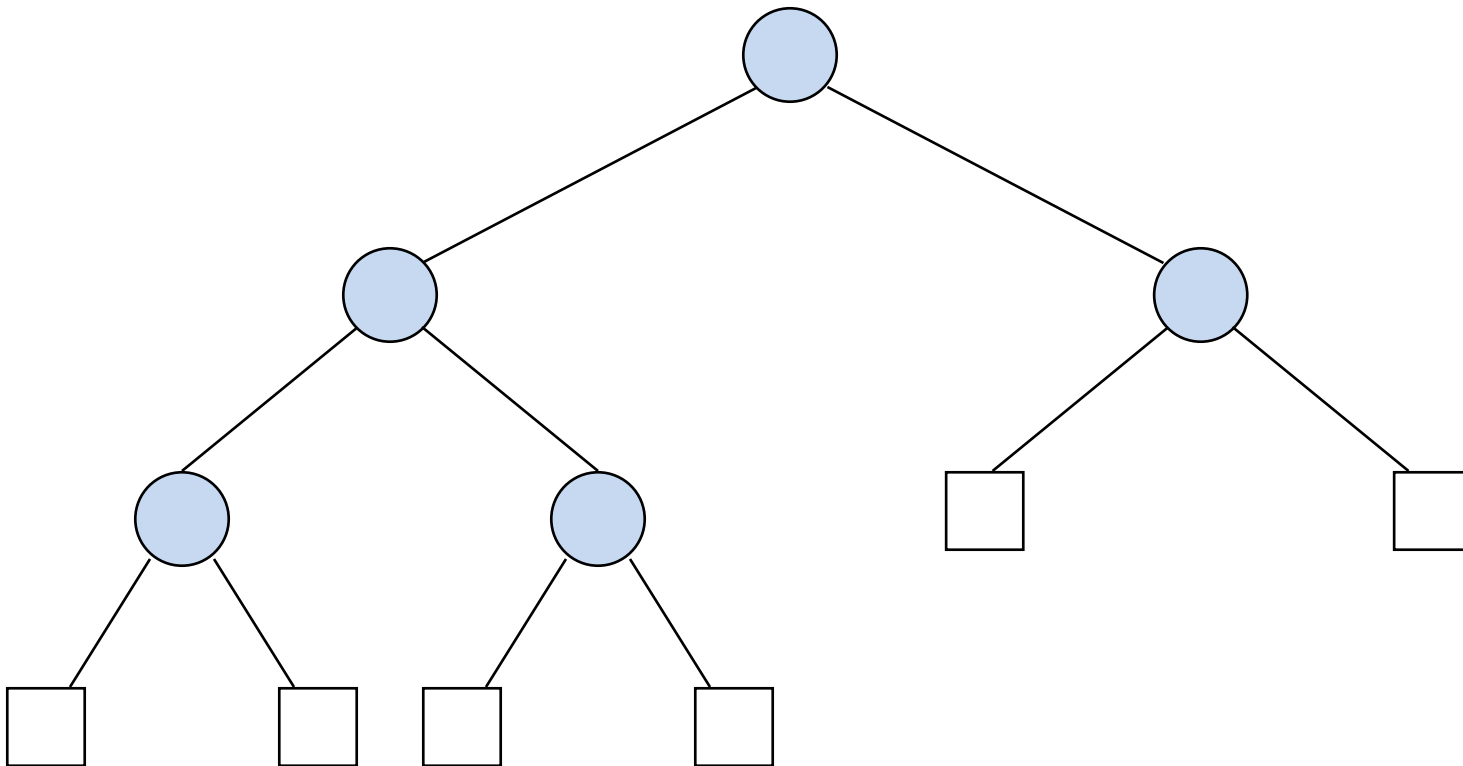
# BINARY\_TREE Operations

*IsRoot*:  $W \times BT \rightarrow B$  :

The function value  $IsRoot(w, T)$  is *true* if the window  $w$  is over the root;

otherwise it is *false*

# Example



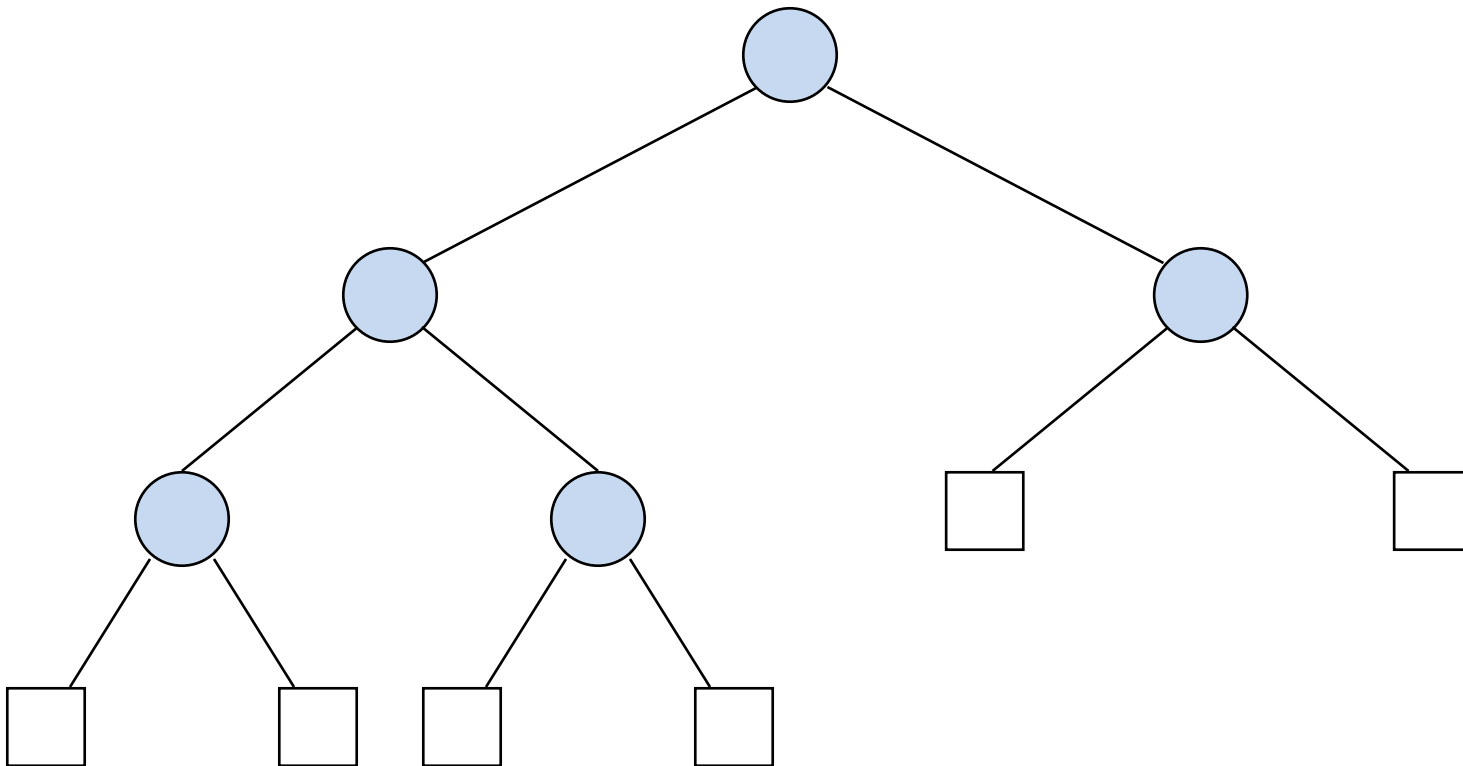
# BINARY\_TREE Operations

*IsExternal*:  $W \times BT \rightarrow B$  :

The function value  $IsExternal(w, T)$  is *true* if the window  $w$  is over an external node of  $T$ ;

otherwise it is *false*

# Example



# BINARY\_TREE Operations

*Child*:  $N \times W \times BT \rightarrow W$  :

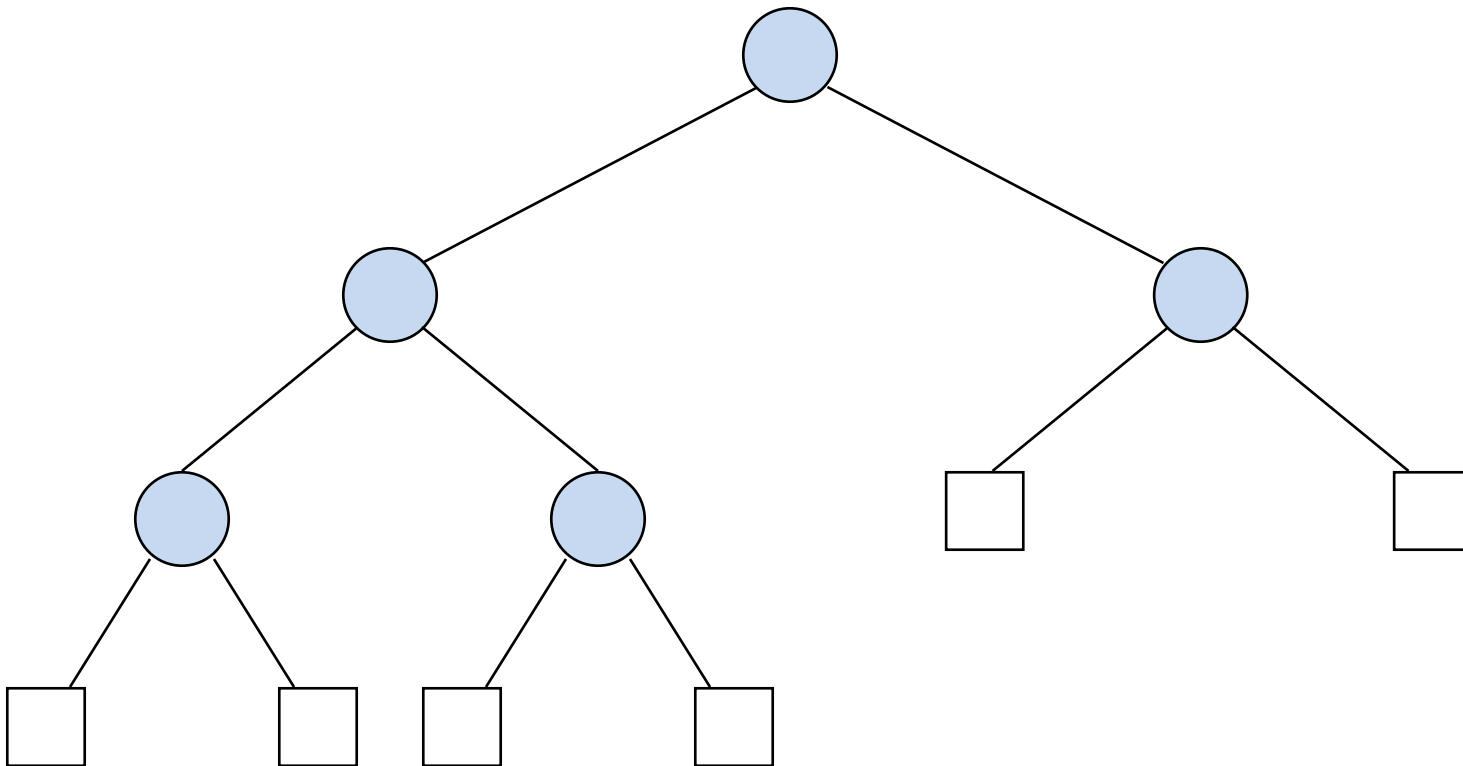
The function value  $Child(i, w, T)$  is undefined if the node in the window  $W$  is external

or

the node in  $w$  is internal and  $i$  is neither 1 nor 2;

otherwise it is the  $i^{\text{th}}$  child of the node in  $w$

# Example



# BINARY\_TREE Operations

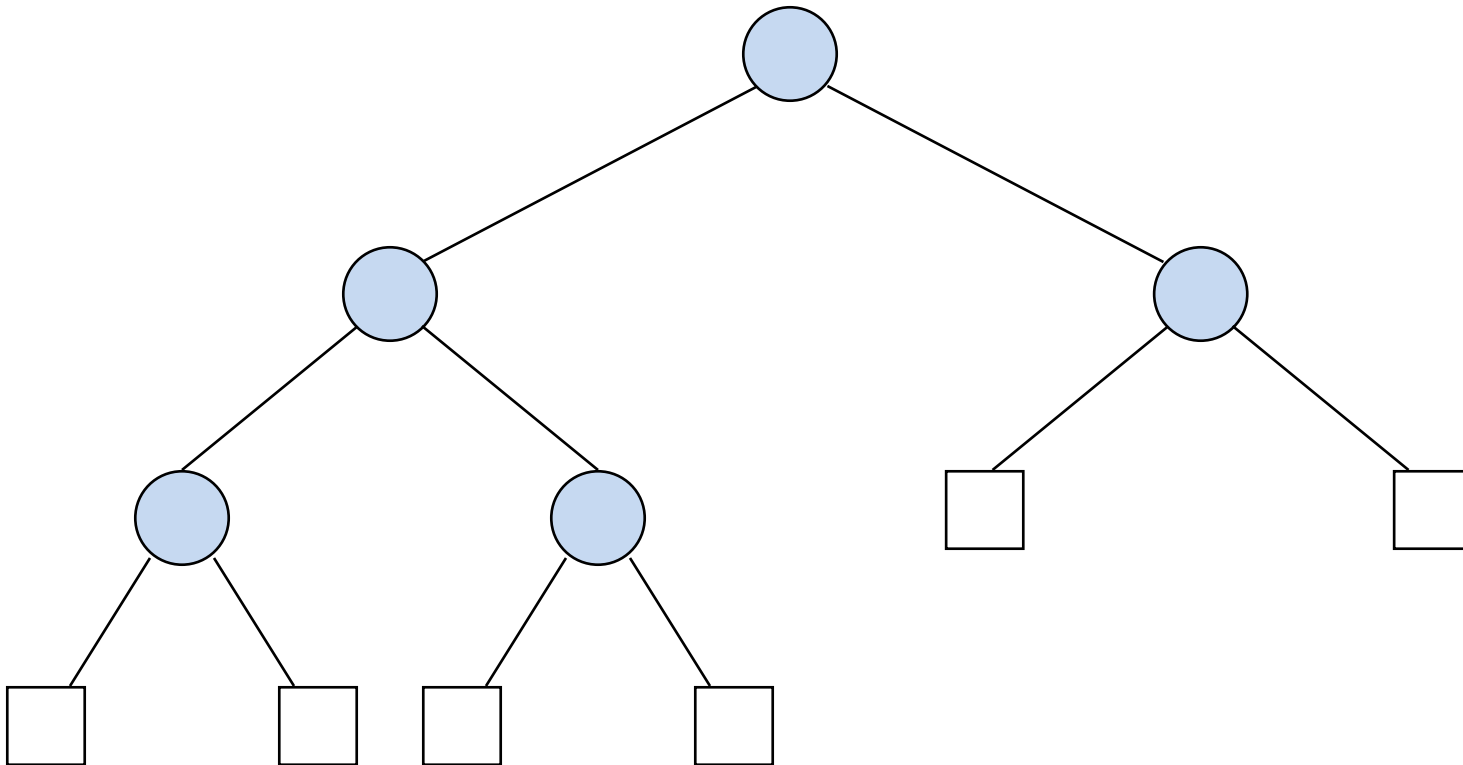
*Parent*:  $W \times BT \rightarrow W$  :

The function value  $Parent(w, T)$  is undefined if  $T$  is empty  
or

$w$  is over the root of  $T$ ;

otherwise it is the window position of the parent of the node in  
the window  $w$

## Example





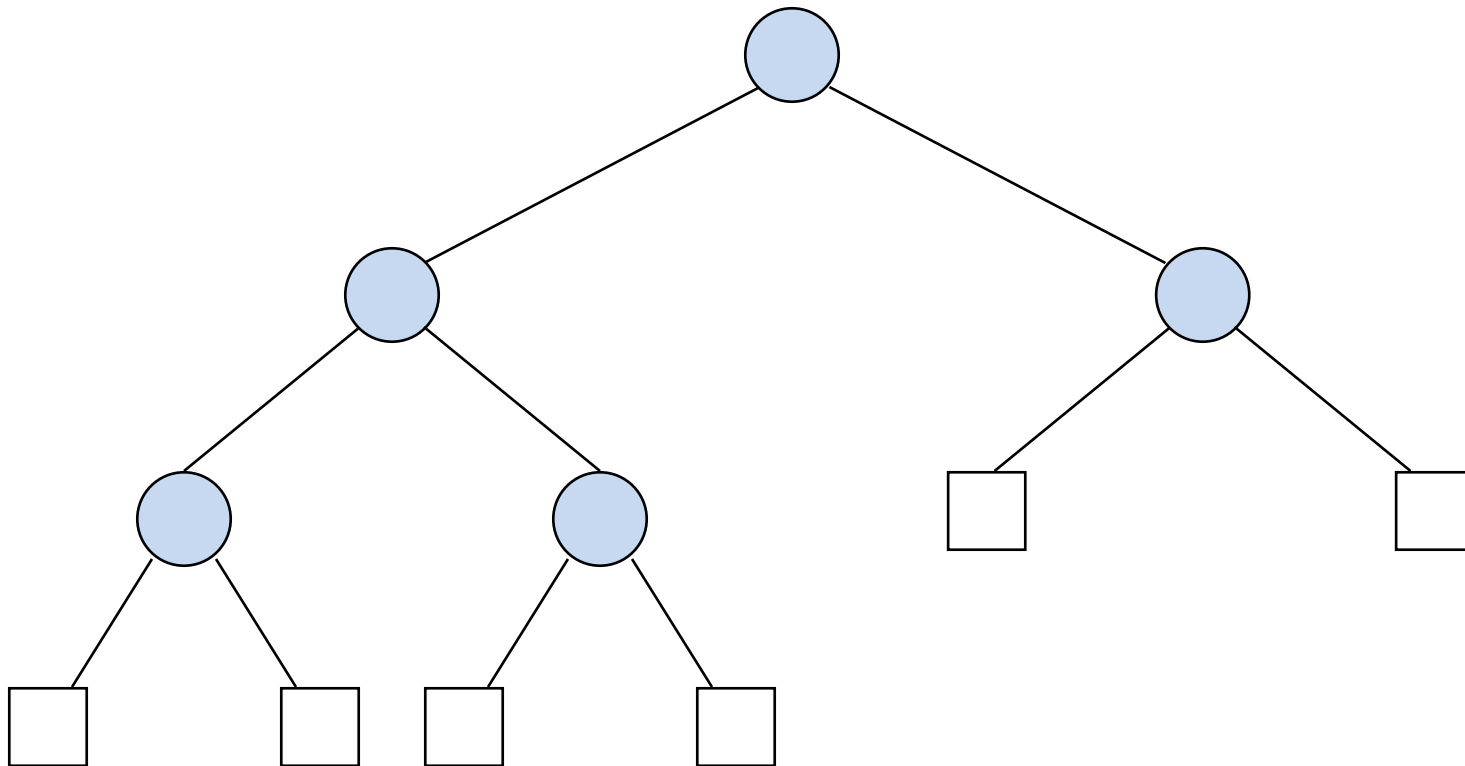
# BINARY\_TREE Operations

*Examine*:  $W \times BT \rightarrow I$ :

The function value  $Examine(w, T)$  is undefined if  $w$  is over an external node;

otherwise it is element at the internal node in the window  $w$

## Example



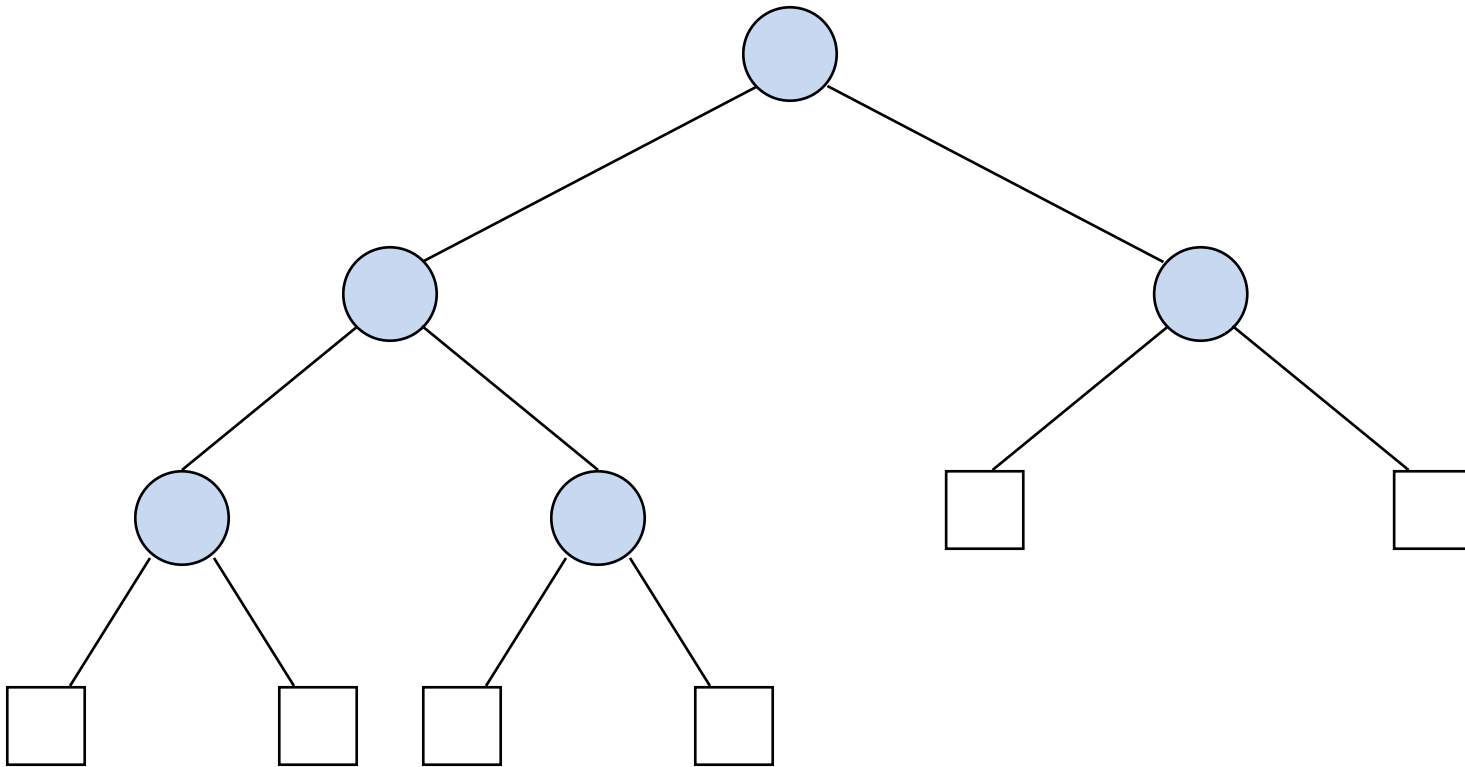
# BINARY\_TREE Operations

*Replace*:  $E \times W \times BT \rightarrow BT$  :

The function value  $Replace(e, w, T)$  is undefined if  $w$  is over an external node;

otherwise it is  $T$ , with the element at the internal node in  $w$  replaced by  $e$

## Example



# BINARY\_TREE Operations

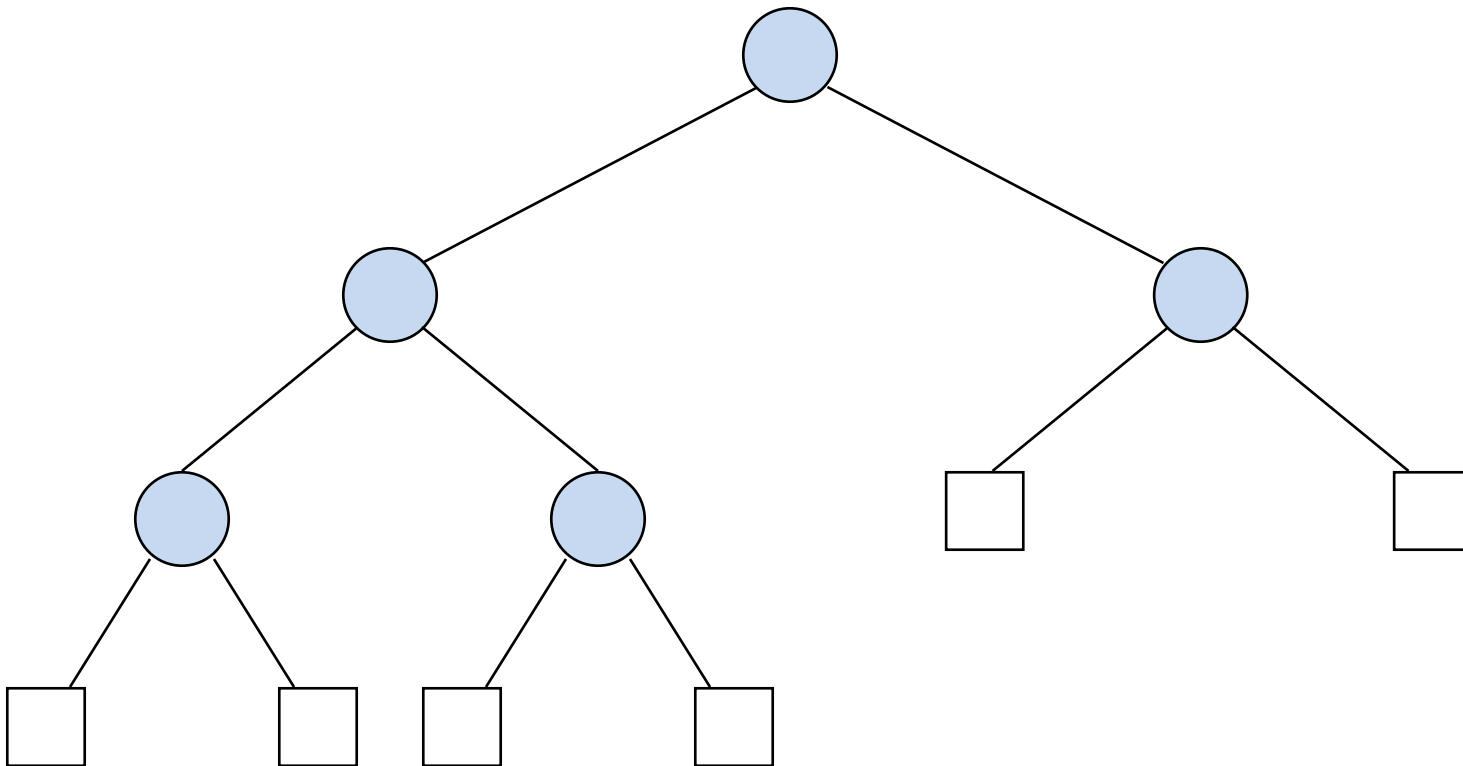
*Insert*:  $E \times W \times BT \rightarrow W \times BT$  :

The function value  $Insert(e, w, T)$  is undefined if  $w$  is over an internal node;

otherwise it is  $T$ , with the external node in  $w$  replaced by a new internal node with two external children.

Furthermore, the new internal node is given the value  $e$  and the window is moved over the new internal node.

# Example



# BINARY\_TREE Operations

*Delete*:  $W \times BT \rightarrow W \times BT$  :

- The function value  $Delete(w, T)$  is undefined if  $w$  is over an external node;
- If  $w$  is over a leaf node (both its children are external nodes), then the function value is  $T$  with the internal node to be deleted **replaced by its left external node**

# BINARY\_TREE Operations

*Delete:*  $W \times BT \rightarrow W \times BT$  :

- If  $w$  is over an internal node with just one internal node child, then the function value is  $T$  with the internal node to be deleted replaced **by its child (internal node)**

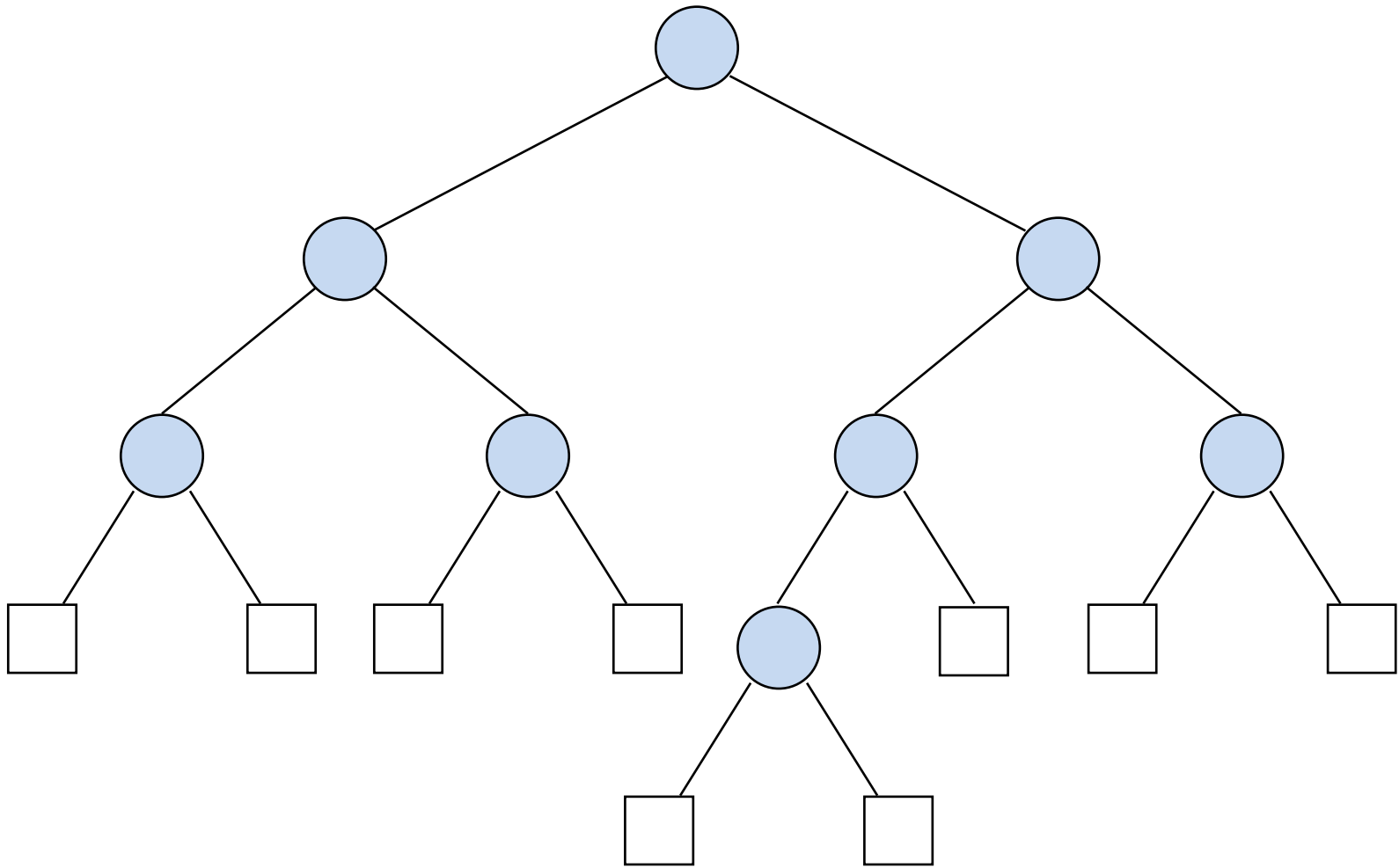


# BINARY\_TREE Operations

*Delete:*  $W \times BT \rightarrow W \times BT$  :

- if  $w$  is over an internal node with **two internal node children**, then the function value is  $T$  with the internal node to be deleted **replaced by the leftmost internal node descendent in its right sub-tree**
- In all cases, the window is moved over the replacement node

# Example



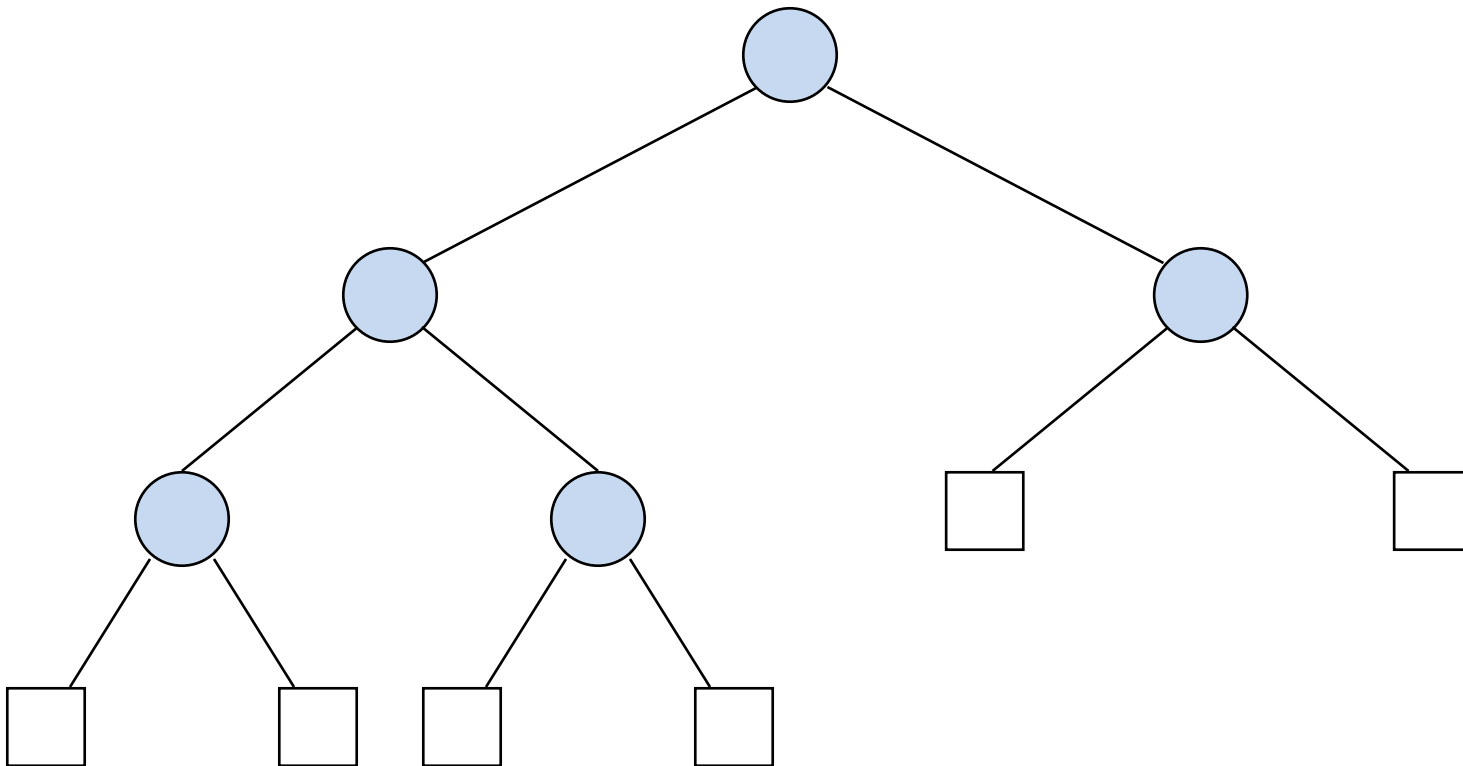
# BINARY\_TREE Operations

*Left*:  $W \times BT \rightarrow W$  :

The function value  $Left(w, T)$  is undefined if  $w$  is over an external node;

otherwise it is the window position of the left (or first) child of the node  $w$

# Example



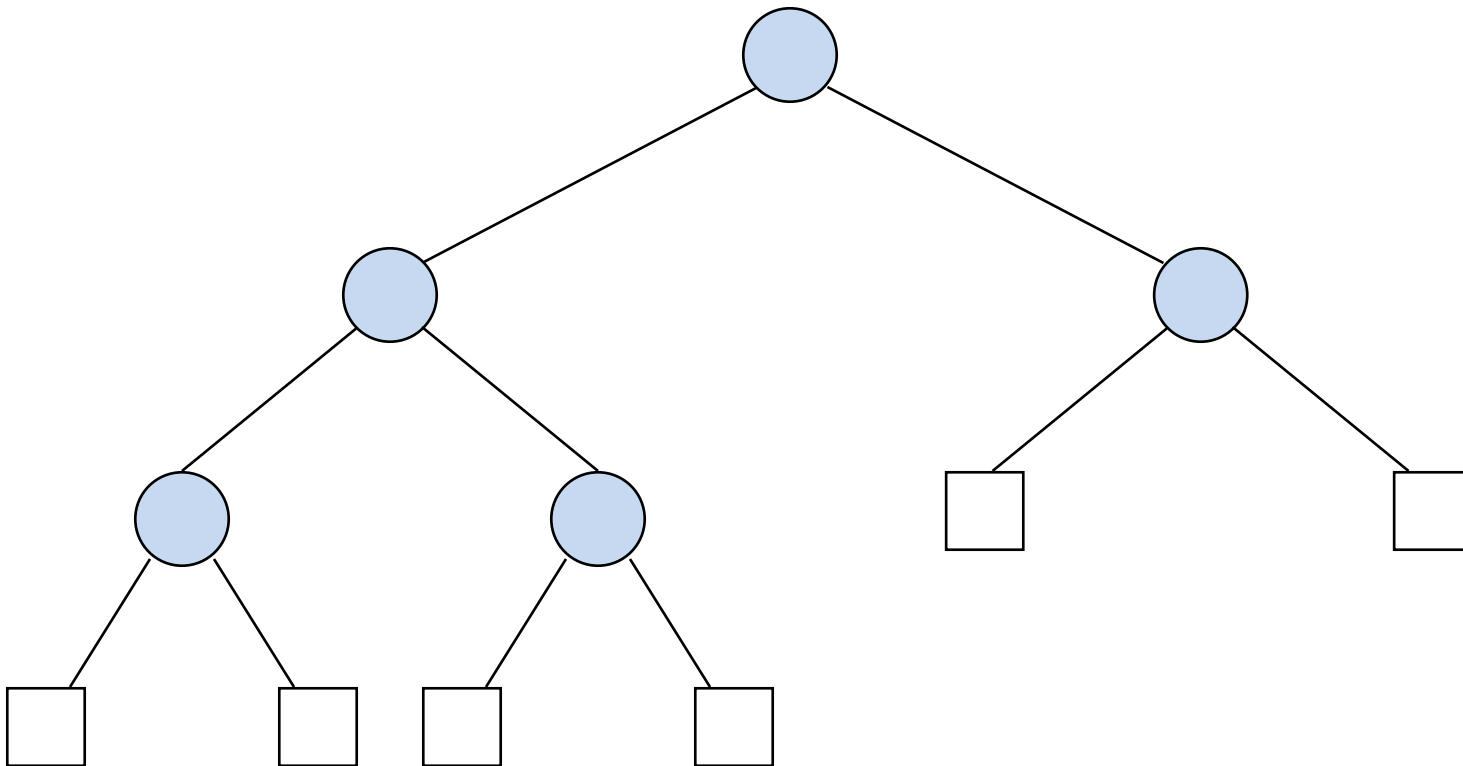
# BINARY\_TREE Operations

*Right*:  $W \times BT \rightarrow W$  :

The function value  $Right(w, T)$  is undefined if  $w$  is over an external node;

otherwise it is the window position of the right (or second) child of the node  $w$

# Example

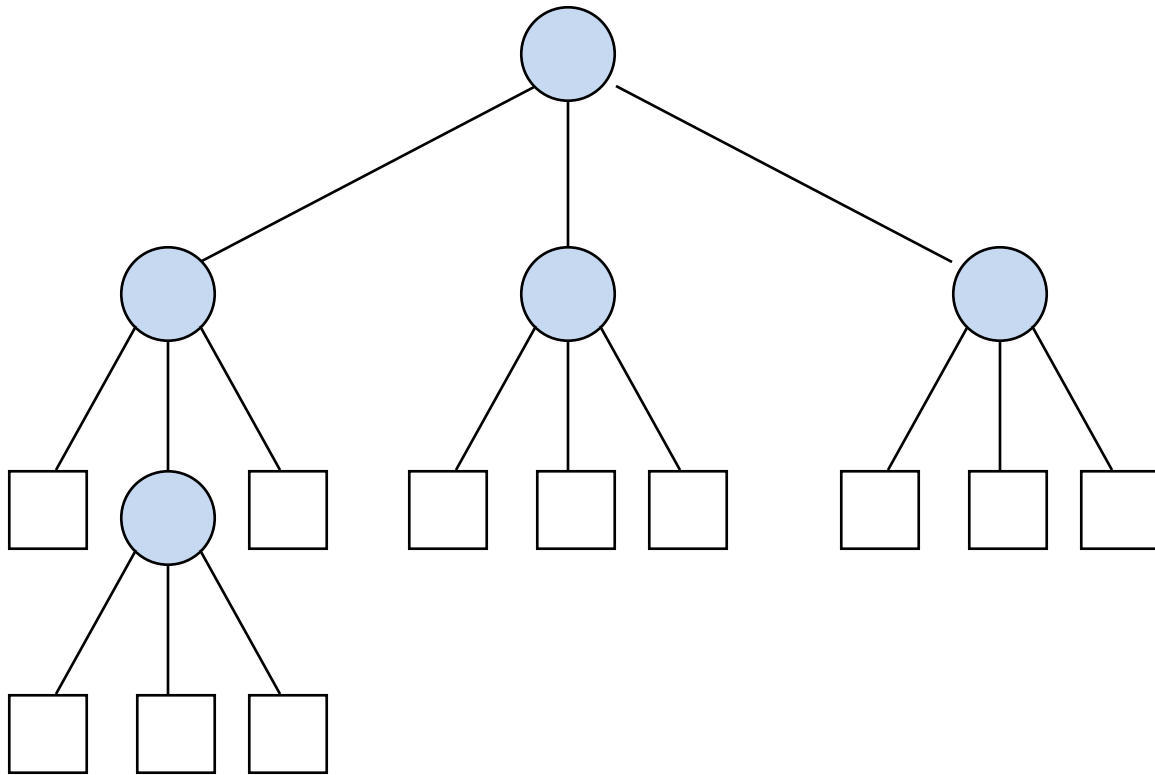


# TREE Operations

*Degree*:  $W \times T \rightarrow I$ :

The function value  $Degree(w, T)$  is the degree of the node in the window  $w$

# d-ary Tree





# TREE Operations

*Child*:  $N \times W \times T \rightarrow W$  :

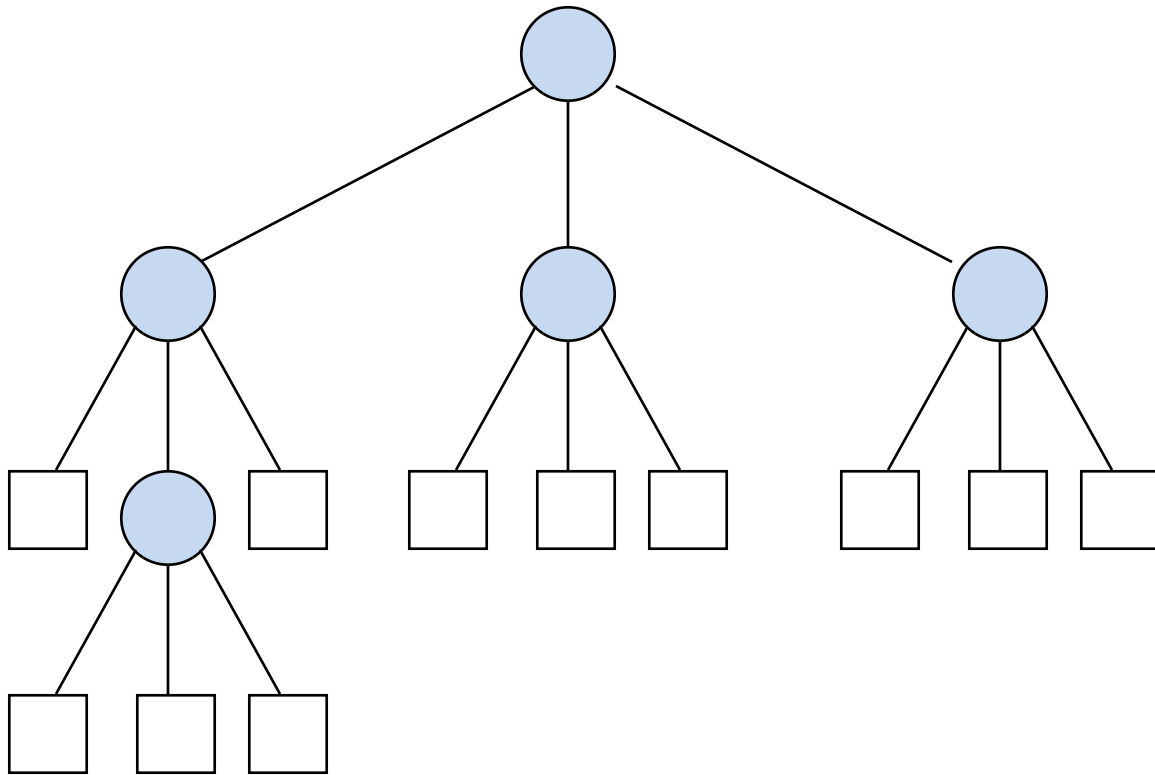
The function value  $Child(i, w, T)$  is undefined if the node in the window  $w$  is external

or

if the node in  $w$  is internal and  $i$  is outside the range  $1..d$ , where  $d$  is the degree of the node;

otherwise it is the  $i^{\text{th}}$  child of the node in  $w$

# d-ary Tree



# BINARY\_TREE Representation

```
/* pointer implementation of BINARY_TREE ADT */

#include <stdio.h>
#include <math.h>
#include <string.h>

#define FALSE 0
#define TRUE 1

typedef struct {
    int number;
    char *string;
} ELEMENT_TYPE;
```

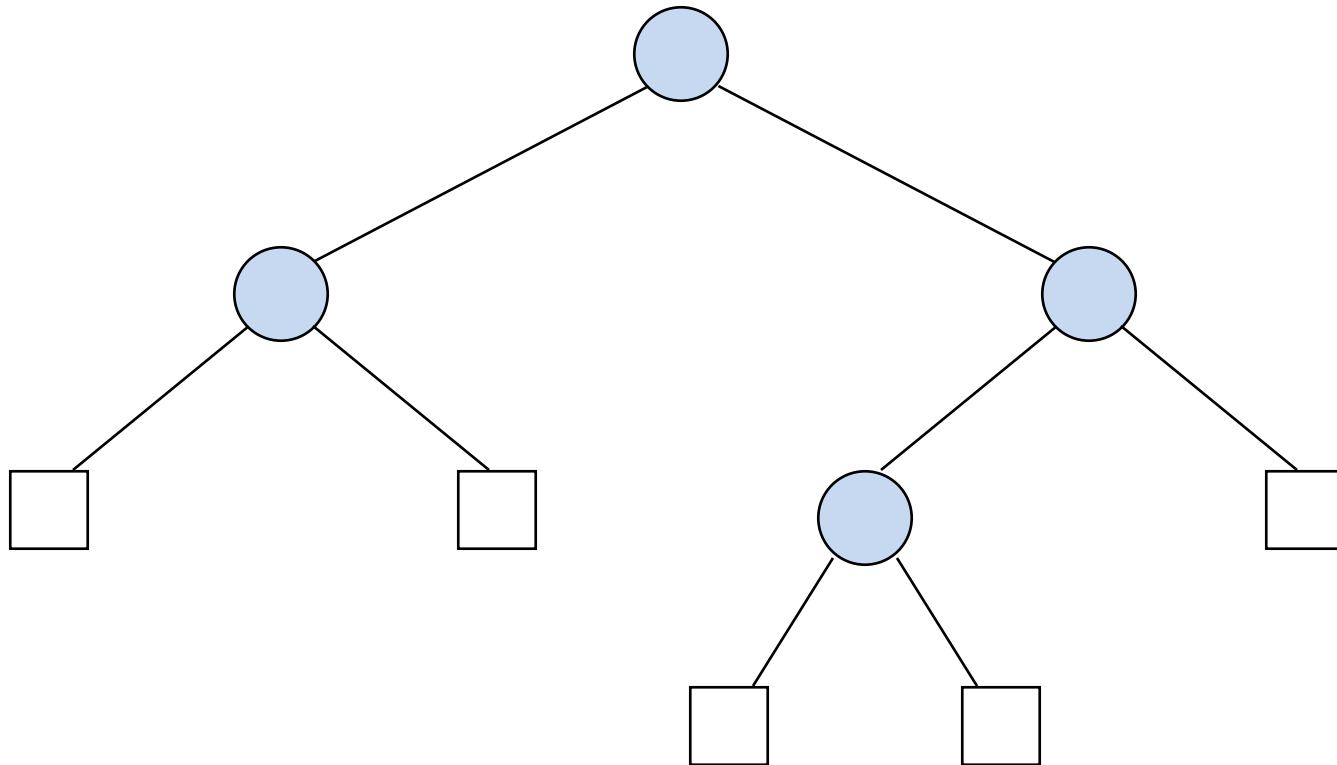
# BINARY\_TREE Representation

```
typedef struct node *NODE_TYPE;

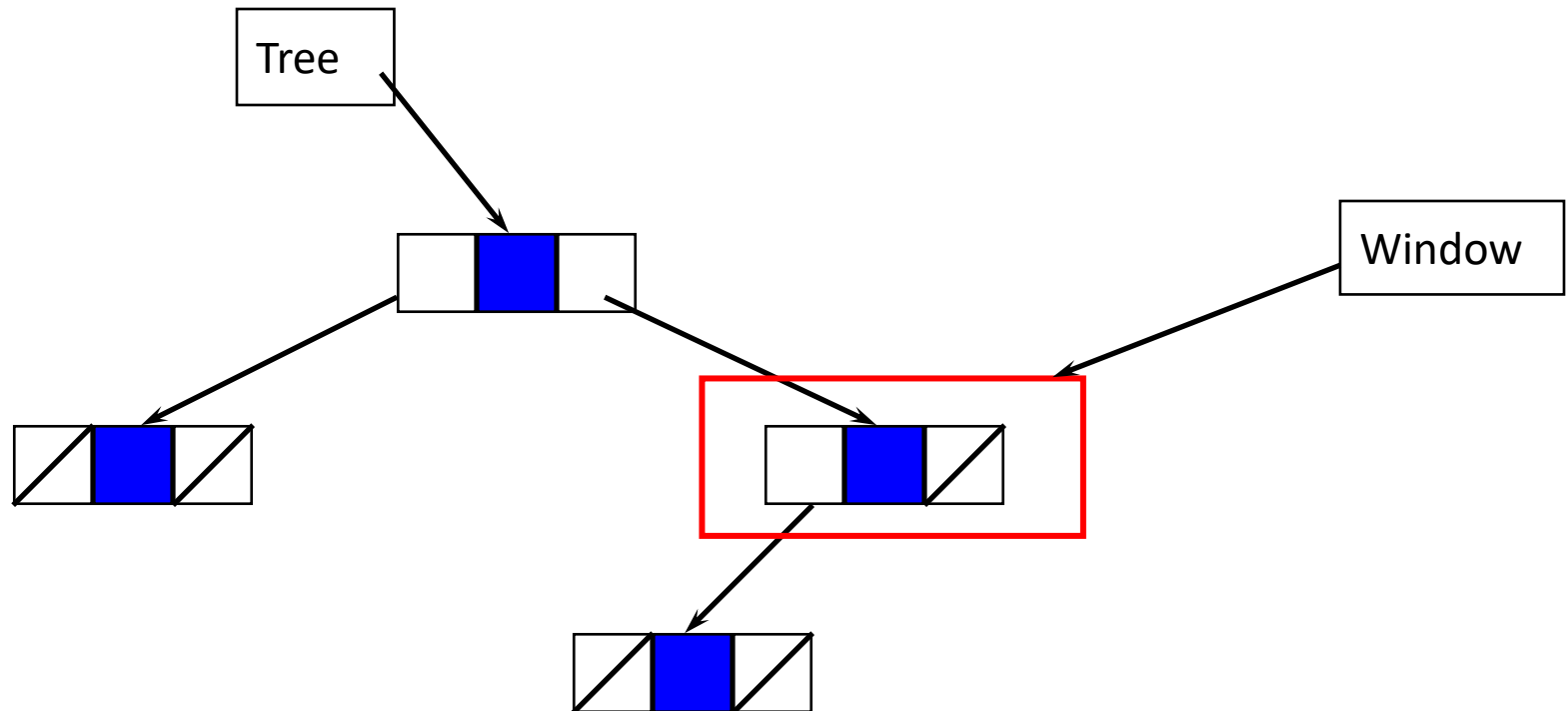
typedef struct node{
    ELEMENT_TYPE element;
    NODE_TYPE left, right;
} NODE;

typedef NODE_TYPE BINARY_TREE_TYPE;
typedef NODE_TYPE WINDOW_TYPE;
```

# BINARY\_TREE Representation



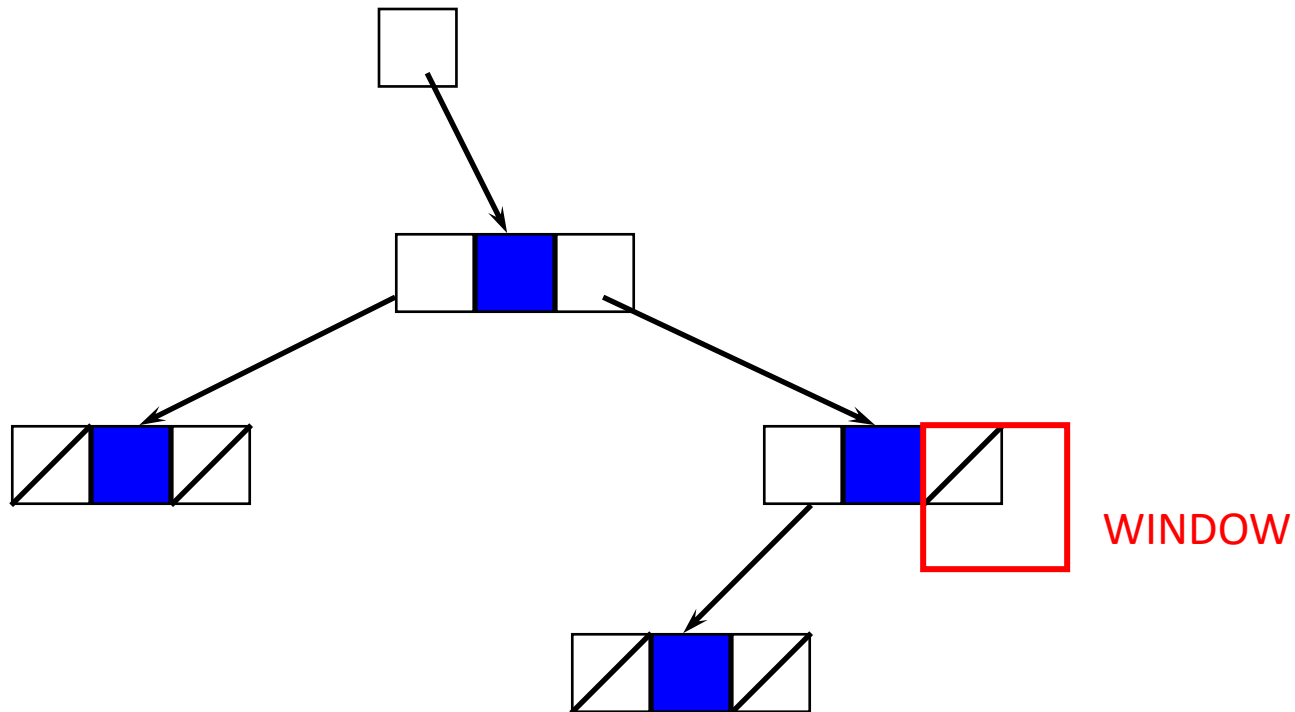
# BINARY\_TREE Representation



# BINARY\_TREE Representation

- This implementation assumes that we are going to represent external nodes as NULL links
- For many ADT operations, we need to know if the window is over an internal or an external node
  - we are over an external node if the window is NULL

# BINARY\_TREE Representation





# BINARY\_TREE Representations

Whenever we insert an internal node

(remember we can only do this if the window is over an external node)

we simply make its two children NULL

# Acknowledgement

- Adopted and Adapted from Material by:
- David Vernon: [vernon@cmu.edu](mailto:vernon@cmu.edu) ; [www.vernon.eu](http://www.vernon.eu)