Arrays, Stack & Queues (Data Structures)

Array

- An array is a data structure consisting of a fixed number of items of the same type.
- Tab : array [1 . . 50] of integers
- The essential property of an array is that we can calculate the address of any given item in constant time.
- An operation such as initializing every item, or finding the largest item, usually takes a time proportional to the number of items to be examined. In other words, such operation take a time in $\theta(n)$.

Array

- Deleting or inserting an element may require us to move all or most of the remaining items in the array to keep items in order. Again such operations take a time in $\theta(n)$.

Records

- A record is a data structure comprising a fixed number of items, often called fields in this context, that are possibly of different types.

Example

```
type person = record
          name: string
          age: integer
          weight: real
          male: Boolean
```

Graphs

- A graph is a set of nodes joined by a set of lines or arrows.
- In a directed graph the nodes are joined by arrows called edges.
- In the case of an undirected graph, the nodes are joined by lines with no direction indicated, also called edges.
- In both directed and undirected graphs, sequences of edges may form paths and cycles. A graph is connected if you can get from any node to any other by following a sequence of edges.

Graphs

- There are never more than two arrows joining any two given nodes of a directed graph, and if there are two arrows, they must go in opposite directions.
- An edge from node a to node b of a directed graph is denoted by the ordered pair (a,b), whereas an edge joining nodes a and b in an undirected graph is denoted by the set {a,b}.
- Figure is an informal representation of the graph G = (N,A) where

  N = *{alpha, beta, gamma, delta}*

  A = *{(alpha, beta), (alpha, gamma), (beta, delta), (gamma, alpha), (gamma, beta), (gamma, delta)}*

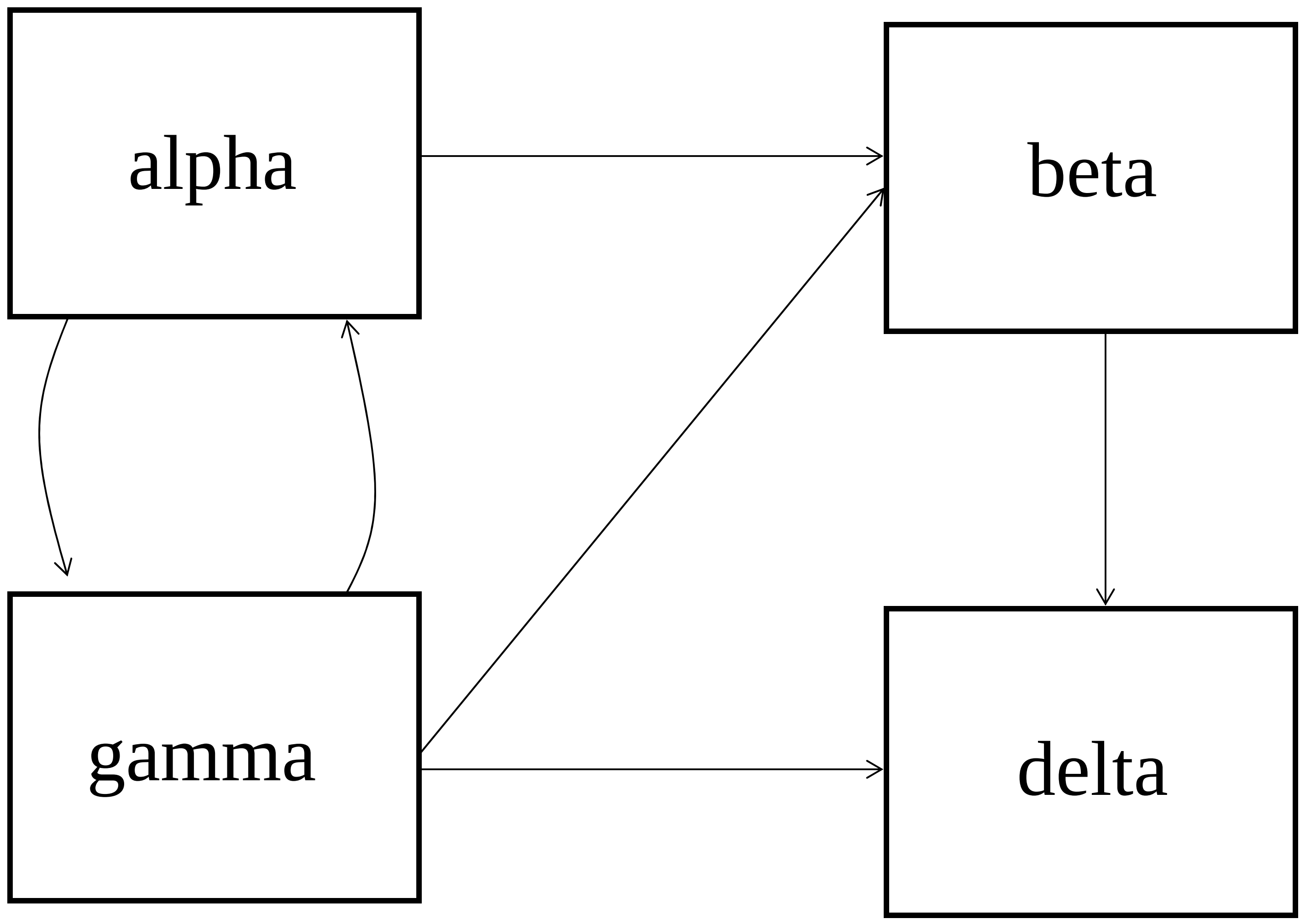


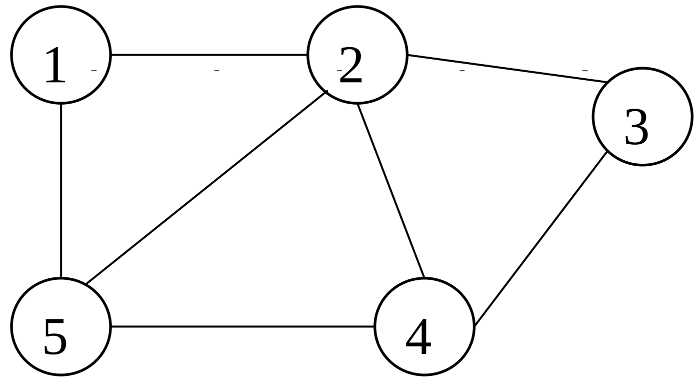
Figure : A directed graph

Graphs

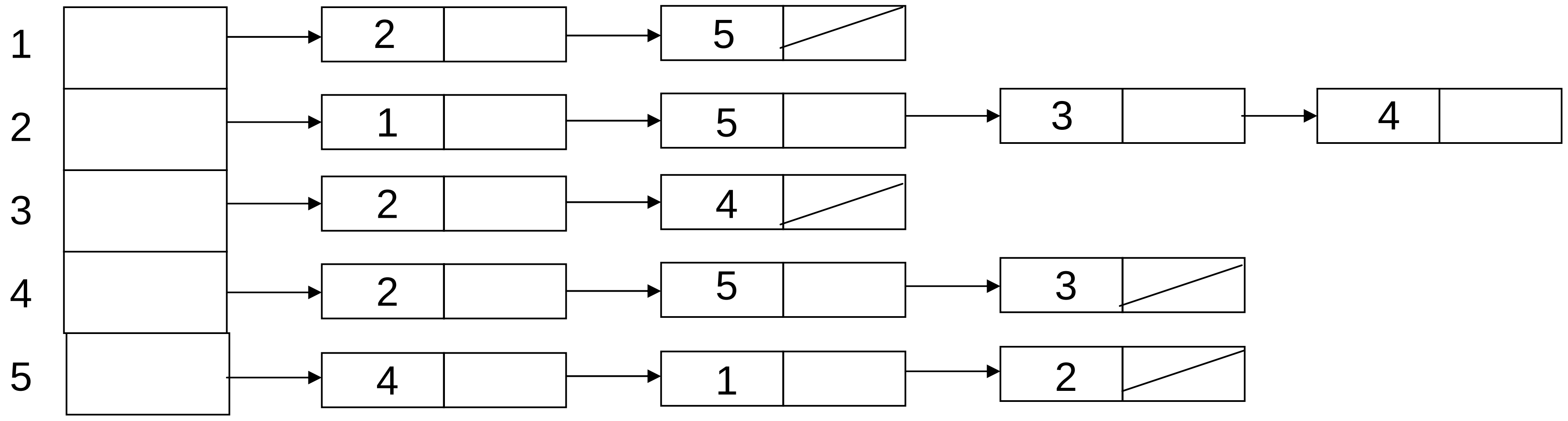
- There are at least two obvious ways to represent a graph on a computer. The first uses an adjacency matrix

```
type adjgraph = record
    value: array [1. . Nbnodes] of information
    adjacent: array [1. . Nbnodes, 1. . Nbnodes]
                        of Boolean
```

Undirected Graph Representation



Adjacency-List Representation



Adjacency-matrix Representation

$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0
\end{array}
$$

Graphs

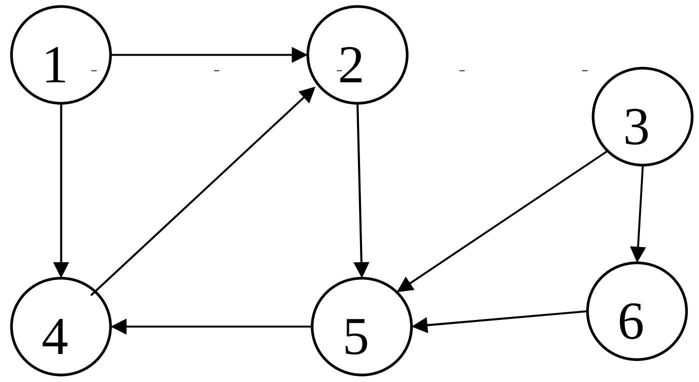
- If the graph includes an edge from node i to node j, then adjacent[i,j] = true; otherwise adjacent[i,j] = false.
- With this representation it is easy to see whether or not there is an edge between two given nodes; to look up a value in any array takes constant time.
- On the other hand, should we wish to examine all the nodes connected to some given node, we have to scan a complete row of the matrix, in the case of an undirected graph, or both a complete row and a complete column, in the case of a directed graph.

Graphs

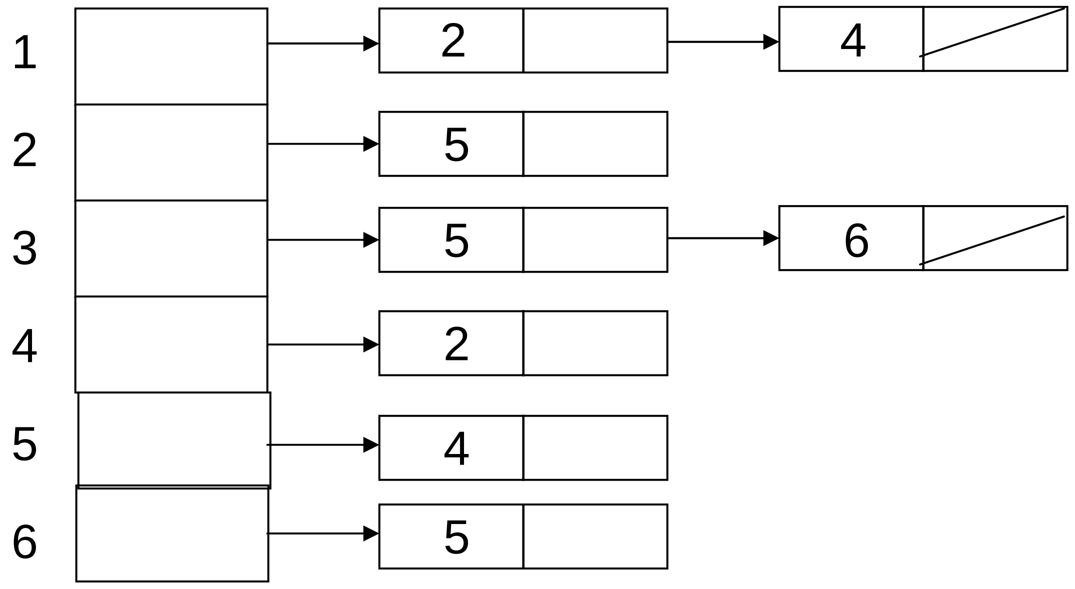
- This takes a time in θ (nbnodes), the number of nodes in the graph, independent of the number of edges that enter or leave this particular node.
- The space required to represent a graph in this fashion is quadratic in the number of nodes.
- A second possible representation is the following:

```
type lisgraph =         array [1 . . Nbnodes] of
                        record
                        value: information
                        neighbours: list
```

Directed Graph Representation



Adjacency-List Representation



Adjacency-matrix Representation

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Graphs

- Here we attach to each node i a list of its neighbours, that is, of those nodes j such that an edge from i to j (in the case of a directed graph), or between i and j (in the case of an undirected graph), exists.
- If the number of edges in the graph is small, this representation uses less storage than the one given previously.
- It may also be possible in this case to examine all the neighbours of a given node in less than nbnodes operations on the average.

Graphs

- On the other hand, determining whether a direct connection exists between two given nodes i and j requires us to scan the list of neighbours of node i (and possibly of node j too, in the case of a directed graph), which is less efficient than looking up a Boolean value in an array.

Trees

- A tree is an acyclic, connected and undirected graph. Equivalently, a tree may be defined as an undirected graph in which there exists exactly one path between any given pair of nodes.
- Since a tree is a kind of graph, the same representations used to implement graphs can be used to implement trees.
- Figure (a) shows two trees, each of which has four nodes.

Trees

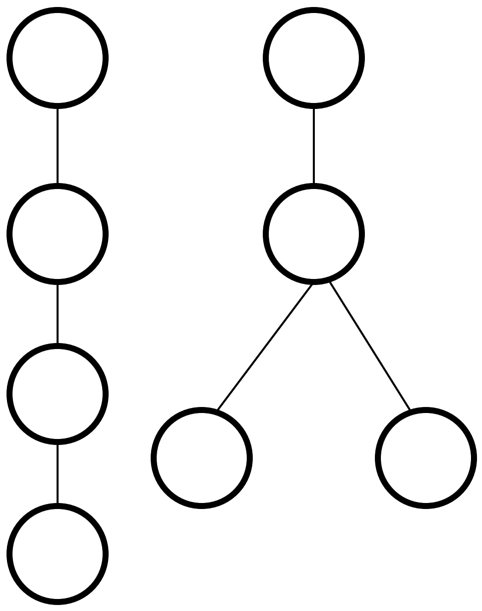


Figure (a) Trees with 4 nodes

Trees

- Figure (b) illustrates four different rooted trees, each with four nodes.
- Trees have a number of simple properties, of which the following are perhaps the most important:
  - A tree with n nodes has exactly n – 1 edges.
  - If a single edge is added to a tree, then the resulting graph contains exactly one cycle.
  - If a single edge is removed from a tree, then the resulting graph is no longer connected.
- These are trees in which one node, called the root, is special. When drawing a rooted tree, it is customary to put the root at the top, like a family tree, with the other edges coming down from it.

Trees

- Figure (b) illustrates four different rooted trees, each with four nodes.

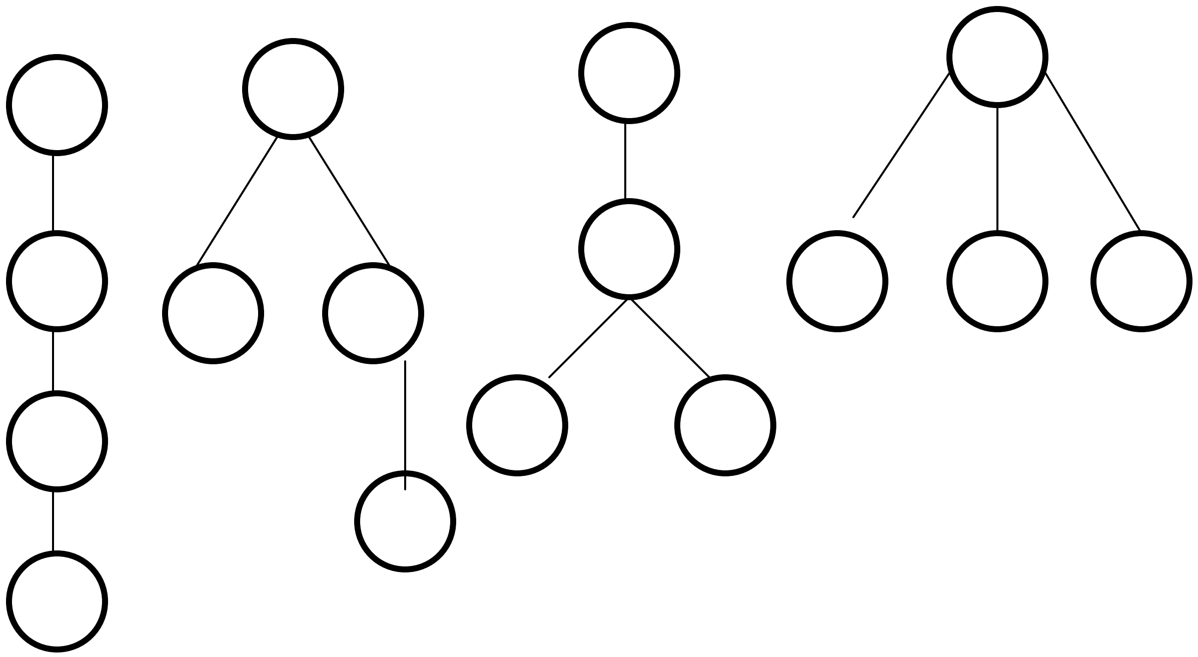


Figure (b) rooted trees with 4 nodes

Trees

- It is customary to use such terms as “parent” and “child” to describe the relationship between adjacent nodes.
- **A**, the root of the tree, is the parent of **B** and **G**; **B** is the parent of **D**, **E** and **Z**, and the child of **A**; while **E** and **Z** are the siblings of **D**. An ancestor of a node is either the node itself, or its parent, its parent’s parent, and so on. Thus both **A** and **B** are ancestors of **Z**.

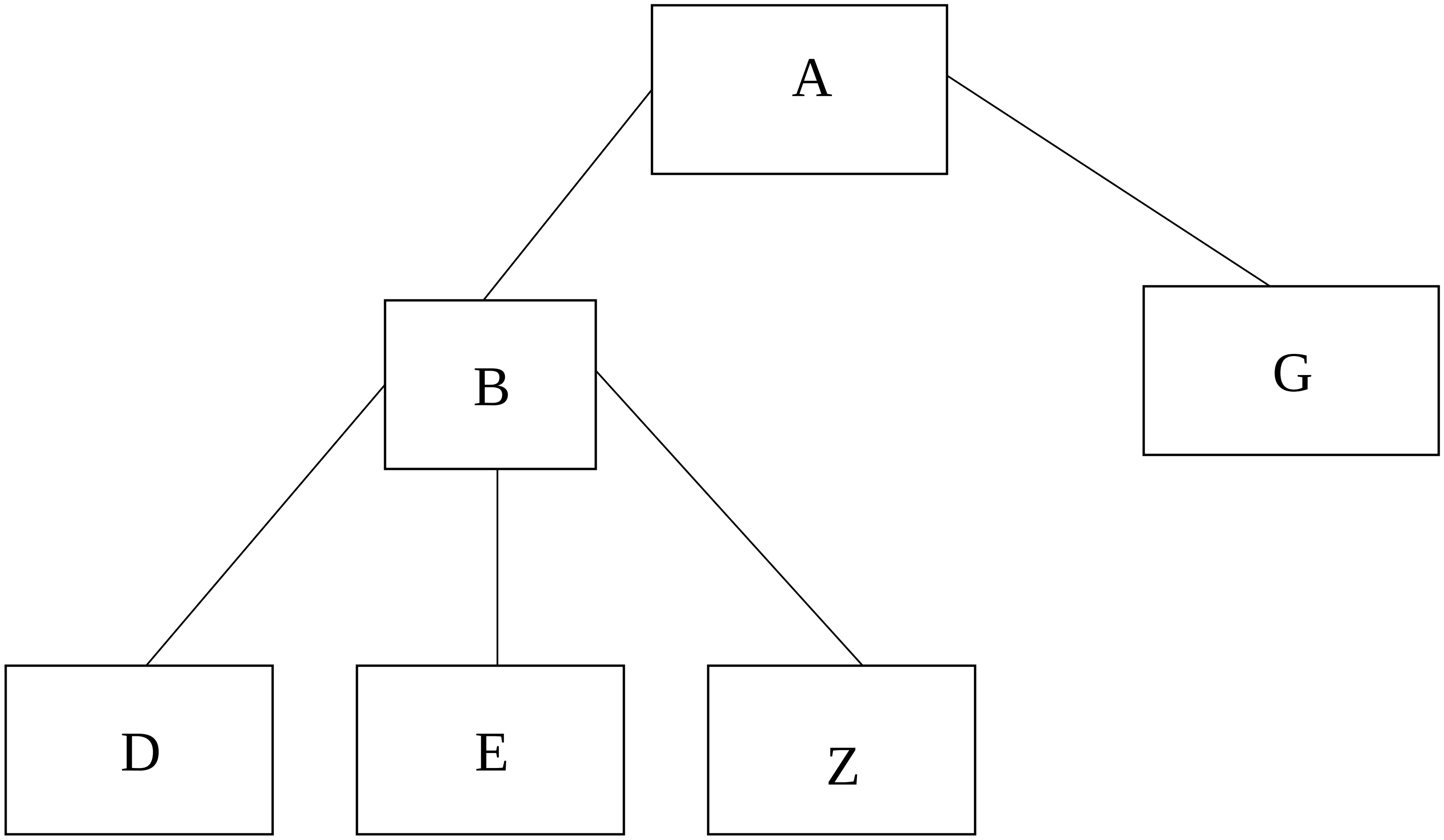


Figure: A rooted tree

Trees

- A leaf of a rooted tree is a node with no children; the other nodes are called internal nodes.
- **B** is situated to the left of **G**, and **D** is called the eldest sibling of **E** and **Z**.
- On a computer any rooted tree may be represented using nodes of the following type.

```
type treenode1 = record
value: information
eldest – child, next – sibling: ↑treenode1
```

Trees

- This representation can be used for any rooted tree; it has the advantage that all the nodes can be represented using the same record structure, no matter how many children or siblings they have. However many operations are inefficient using this minimal representation: it is not obvious how to find the parent of a given node.

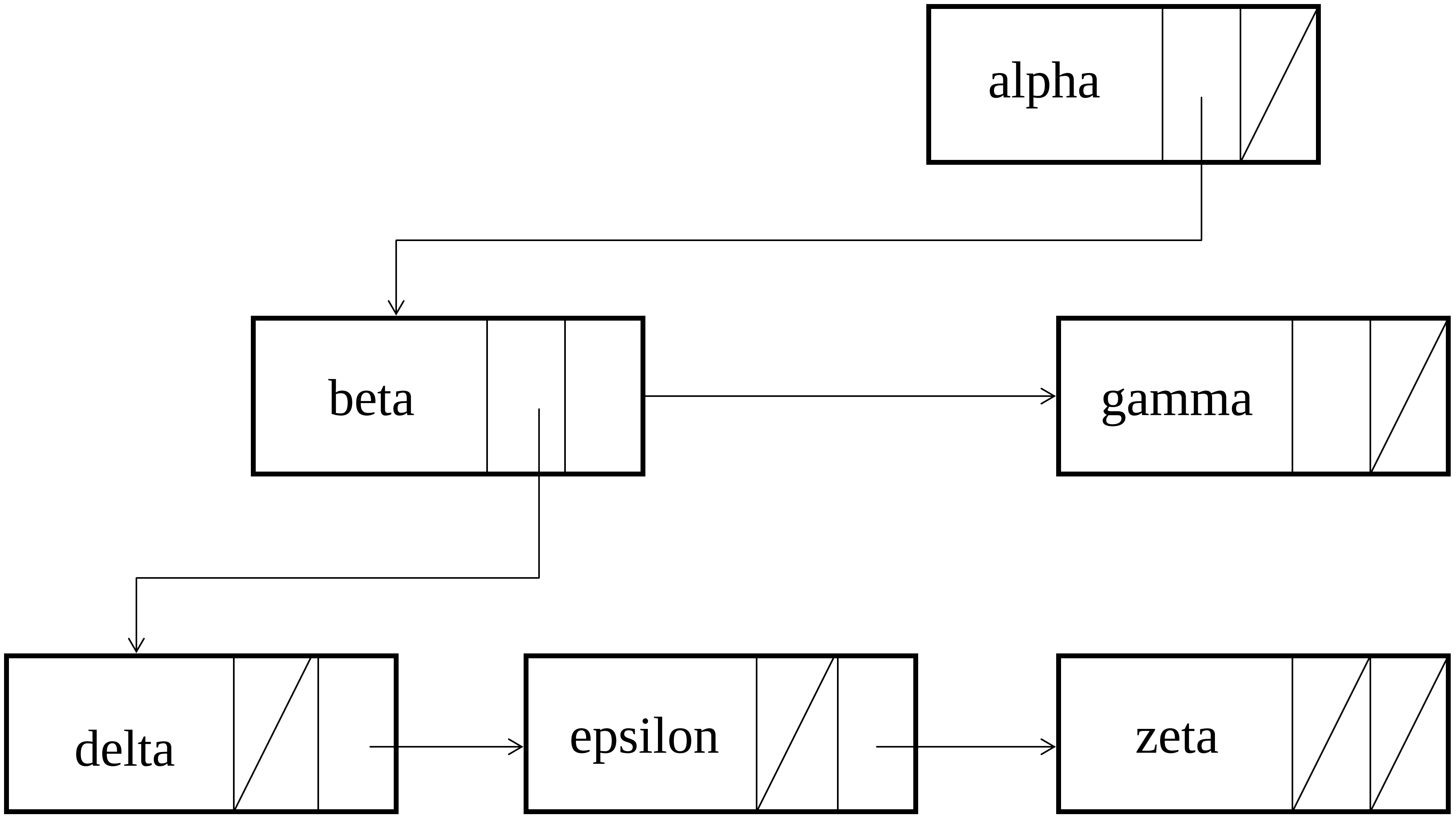


Figure : Possible computer representation of a rooted tree

Trees

- Another representation suitable for any rooted tree uses nodes of the type.

```
type treenode2 = record
        value: information
        parent: ↑treenode2
```

- Now each node contains only a single pointer leading to its parent. This representation is about as economical with storage space as one can hope to be, but it is inefficient unless all the operations on the tree involve starting from a node and going up, never down.
- It does not represent the order of siblings.

Trees

- We shall often have occasion to use binary trees. In such a tree, each node can have 0, 1, or 2 children.
- In fact, we almost always assume that a node has two pointers, one to its left and one to its right, either of which can be nil.
- The two binary trees as shown in figure are not the same: in the first case b is the left child of a and the right child is missing, whereas in the second case b is the right child of a and the left child is missing.

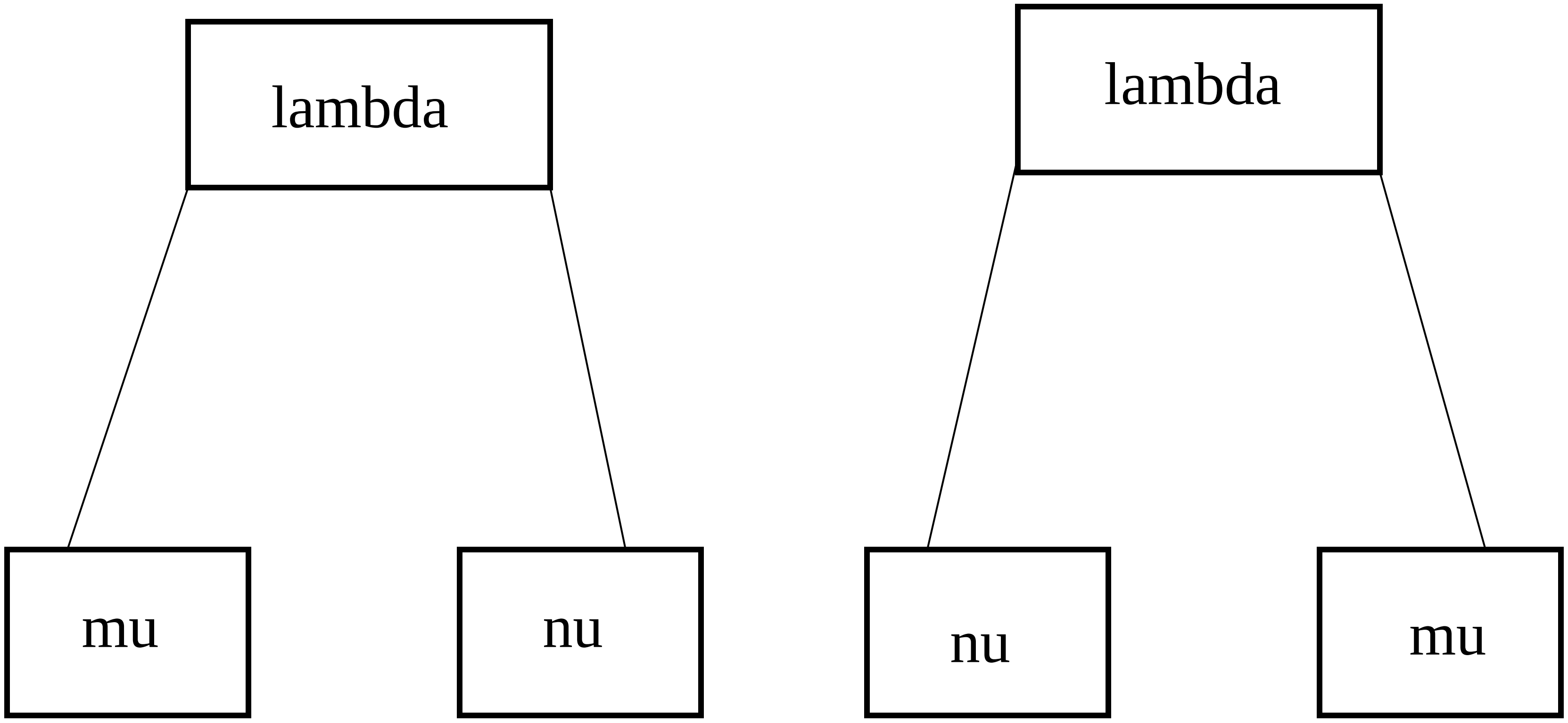


Figure: Two distinct rooted trees

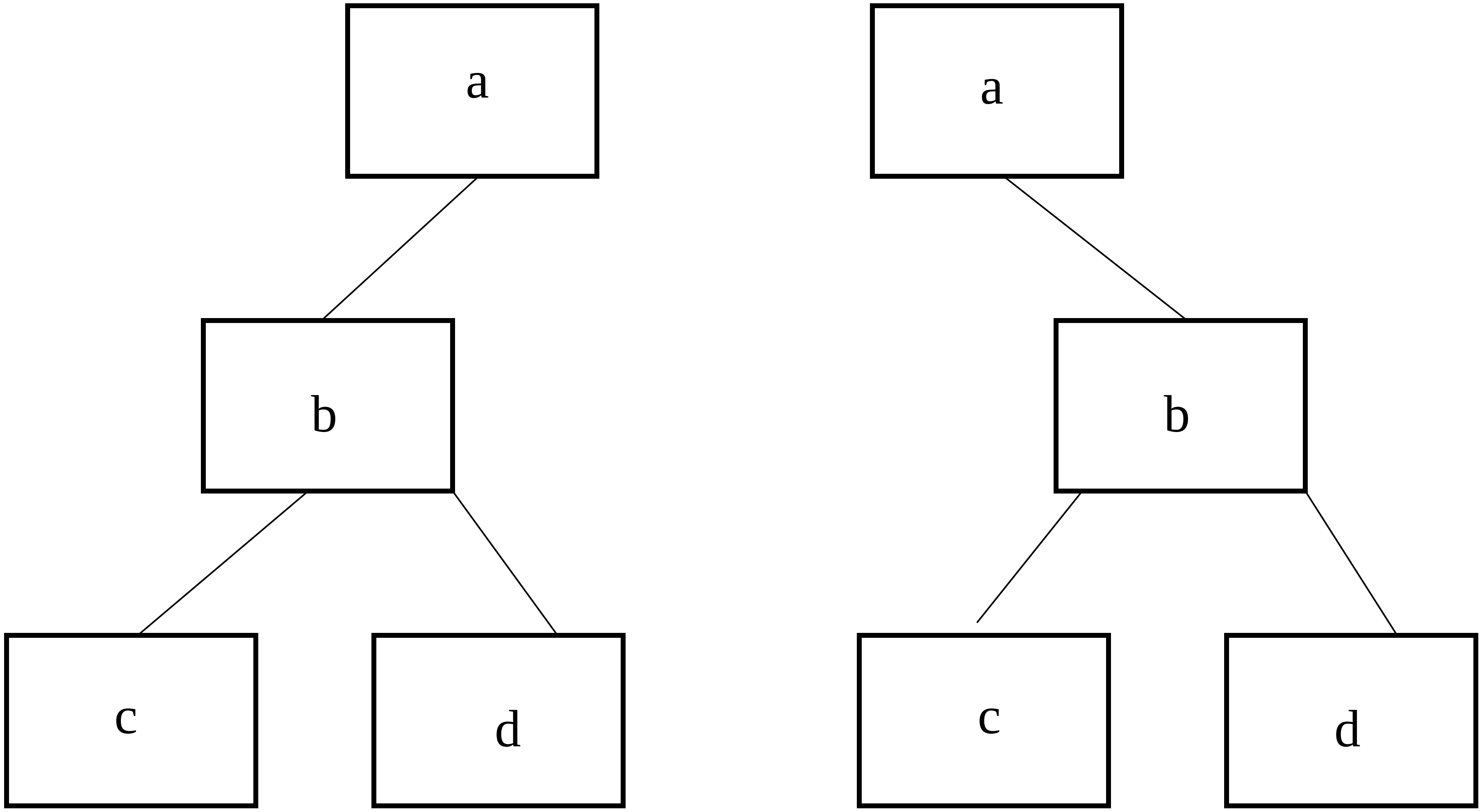


Figure. Two distinct binary trees

Trees

- If each node of a rooted tree can have no more than k children, we say it is a k-ary tree.
- One obvious representation uses nodes of the

```
type k-ary-node = record
          value: information
          child: array [i. . k] of ↑k-ary-node
```

- In the case of a binary tree we can also define

```
type binary-node = record
          value: information
          left-child, right-child : ↑binary-node
```

Trees

- A binary tree is a search tree if the value contained in every internal node is greater than or equal to the values contained in its left child or any of that child’s descendants, and less than to the values contained in its right child or any of that child’s descendants.
- Figure gives an example of a search tree.
- This structure is interesting because, as the name implies, it allows efficient searches for values in the tree.
- We can find 27, say, with only 3 comparisons.

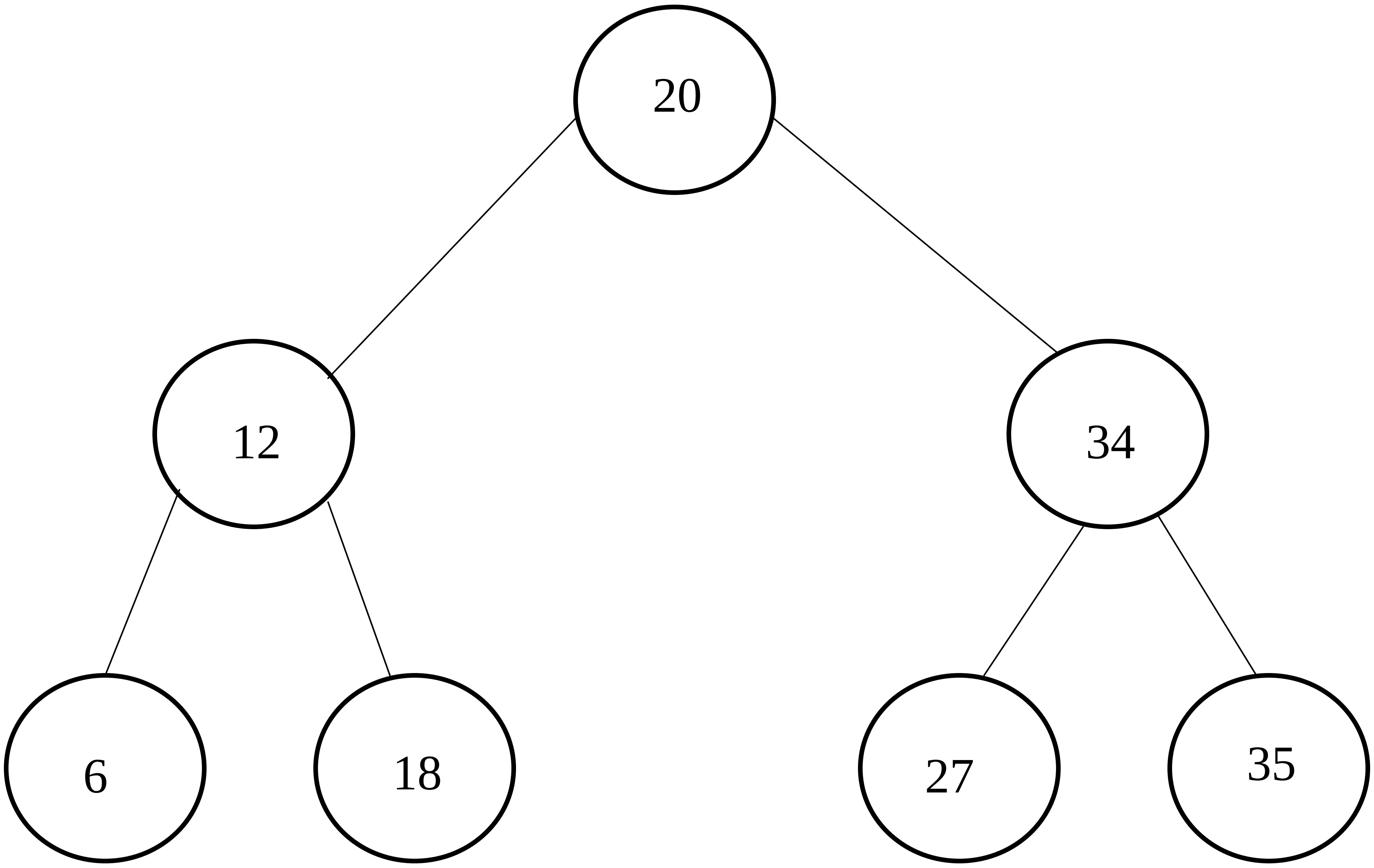


Figure. A search tree

```
function search (x,r)
    {The pointer r points to the root of a search tree. The
    function searches for the value x in this tree and
    returns a pointer to the node containing x. If x is
    missing, the function returns nil.}
    if r = nil then {x is not in the tree.}
                    return nil
    else if x = r↑. Value then return r
    else if x < r ↑. Value then return search (x,r ↑.left-child)
    else return search(x,r ↑.right-child)
```

Trees

- It is simple to update a search tree, that is, to delete a node or to add a new value, without destroying the search tree property.
- In the worst case, every node in the tree may have exactly one child, except for a single leaf that has no children. With such an unbalanced tree, finding an item in a tree with n elements may involve comparing it with the contents of all n nodes.

Trees

- A variety of methods are available to keep the tree balanced, and hence to guarantee that such operations as searches or the addition and deletion of nodes take a O(log n) in the worst case, where n is the number of nodes in the tree.

Height, Depth and Level

- Height and level, for instance, are similar concepts, but not the same.
  - The height of a node is the number of edges in the longest path from the node in question to a leaf.
  - The depth of a node is the number of edges in the path from the root to the node in question.
  - The level of a node is equal to the height of the root of the tree minus the depth of the node concerned.

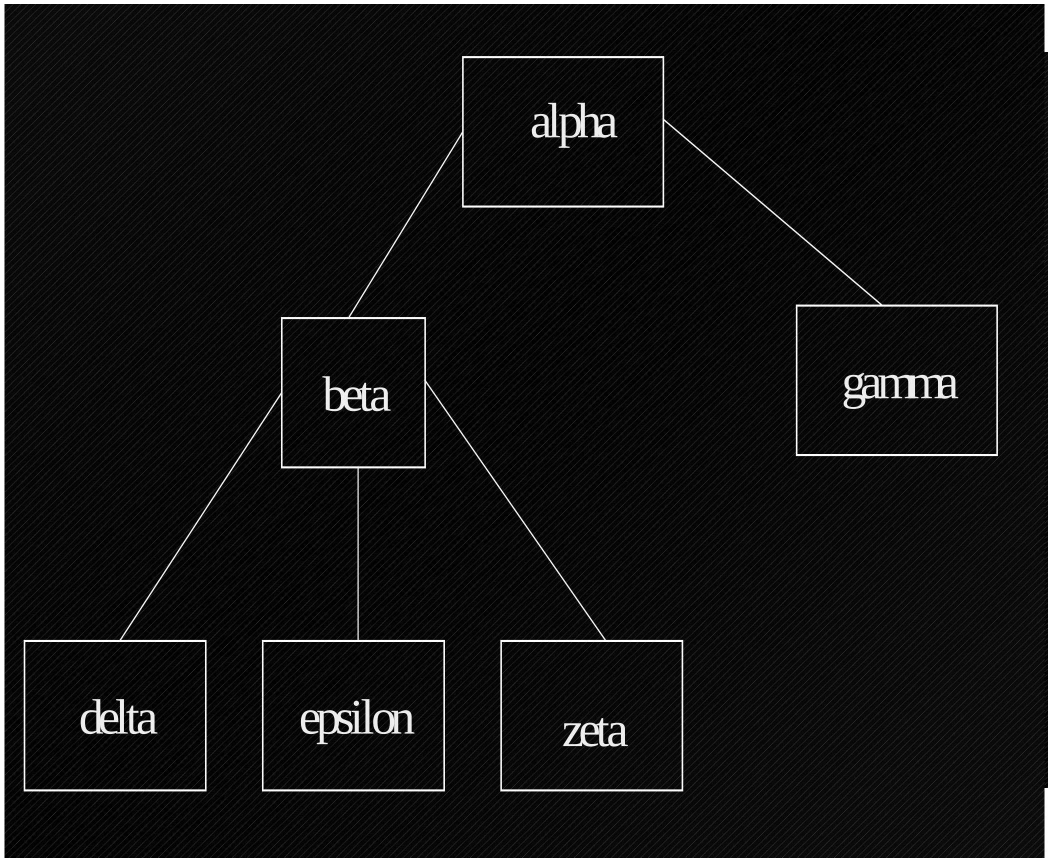
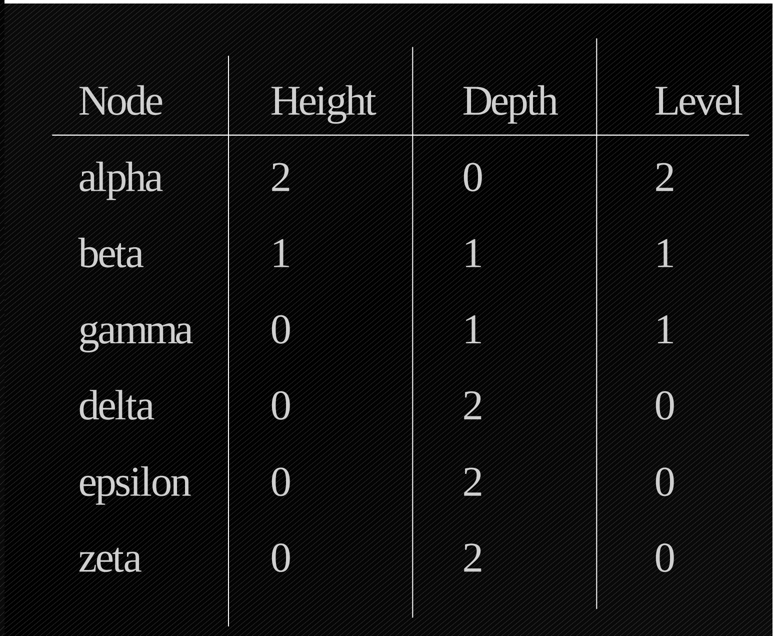


Figure: A rooted tree



Node	Height	Depth	Level
alpha	2	0	2
beta	1	1	1
gamma	0	1	1
delta	0	2	0
epsilon	0	2	0
zeta	0	2	0

Figure: Height, depth and level

Figure: Height, depth and level

Height, Depth and Level

- Informally, if the tree is drawn with successive generations of nodes in neat layers, then the depth of a node is found by numbering the layers downwards from 0 at the root; the level of a node is found by numbering the layers upwards from 0 at the bottom; only the height is a little more complicated.
- Finally we define the height of the tree to be the height of its root; this is also the depth of the deepest leaf and the level of the root.