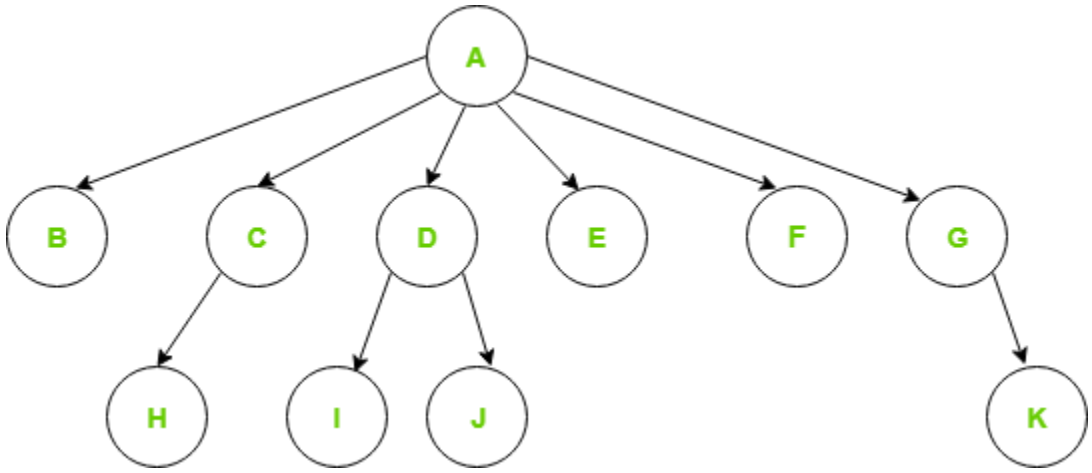
N Array Tree:

Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes. Every node stores address of its children and the very first node's address will be stored in a separate pointer called root.
The Generic trees are the N-ary trees which have the following properties:

1. Many children at every node.
2. The number of nodes for each node is not known in advance.

Example:



To represent the above tree, we have to consider the worst case, that is the node with maximum children (in above example, 6 children) and allocate that many pointers for each node.
The node representation based on this method can be written as:

```
// Java code for above approach
public class Node {
```

```
    int data;
    Node firstchild;
    Node secondchild;
    Node thirdchild;
    Node fourthchild;
    Node fifthchild;
    Node sixthchild;
}
```

Disadvantages of the above representation are:

1. **Memory Wastage** – All the pointers are not required in all the cases. Hence, there is lot of memory wastage.
2. **Unknown number of children** – The number of children for each node is not known in advance.

Simple Approach:

For storing the address of children in a node we can use an array or linked list. But we will face some issues with both of them.

1. In **Linked list**, we can not randomly access any child's address. So it will be expensive.
2. In **array**, we can randomly access the address of any child, but we can store only fixed number of children's addresses in it.

Better Approach:

We can use Dynamic Arrays for storing the address of children. We can randomly access any child's address and the size of the vector is also not fixed.

```
import java.util.ArrayList;
```

```
class Node {
	int data;
	ArrayList<Node> children;

	Node(int data)
	{
		this.data = data;
		this.children = new ArrayList<Node>();
	}
}
```

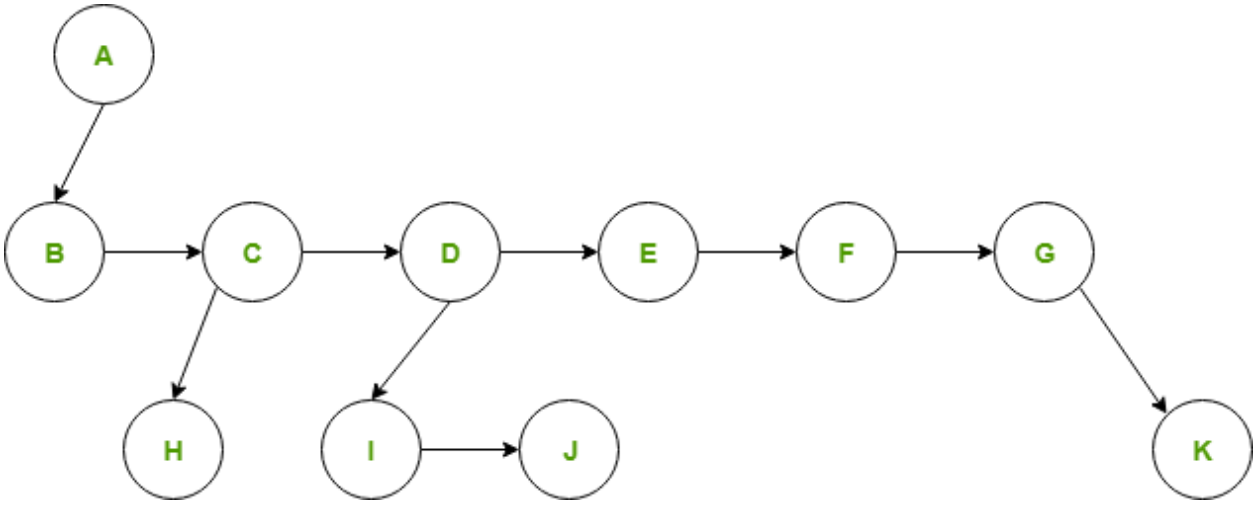
Efficient Approach:
First child / Next sibling representation

In the first child/next sibling representation, the steps taken are:

At each node-link the children of the same parent(siblings) from left to right.

- Remove the links from parent to all children except the first child.

Since we have a link between children, we do not need extra links from parents to all the children. This representation allows us to traverse all the elements by starting at the first child of the parent.



The node declaration for first child / next sibling representation can be written as:

```
class Node {
    int data;
    Node firstChild;
    Node nextSibling;
}
```

Advantages:

- Memory efficient – No extra links are required, hence a lot of memory is saved.
- Treated as binary trees – Since we are able to convert any generic tree to binary representation, we can treat all generic trees with a first child/next sibling representation as binary trees. Instead of left and right pointers, we just use firstChild and nextSibling.
- Many algorithms can be expressed more easily because it is just a binary tree.
- Each node is of fixed size ,so no auxiliary array or vector is required.

Height of generic tree from parent array

Generic tree – level order traversal