A different approach to optimal solutions

- A rooted binary tree is an acyclic directed graph in which:
  - There is exactly one node (the root) with no parents
  - Each non-root node has exactly one parent
  - Each node has at most two children. A childless node is called a **leaf**

A different approach to optimal solutions

Each node is labeled with a quadruple that denotes a partial solution to the knapsack problem:

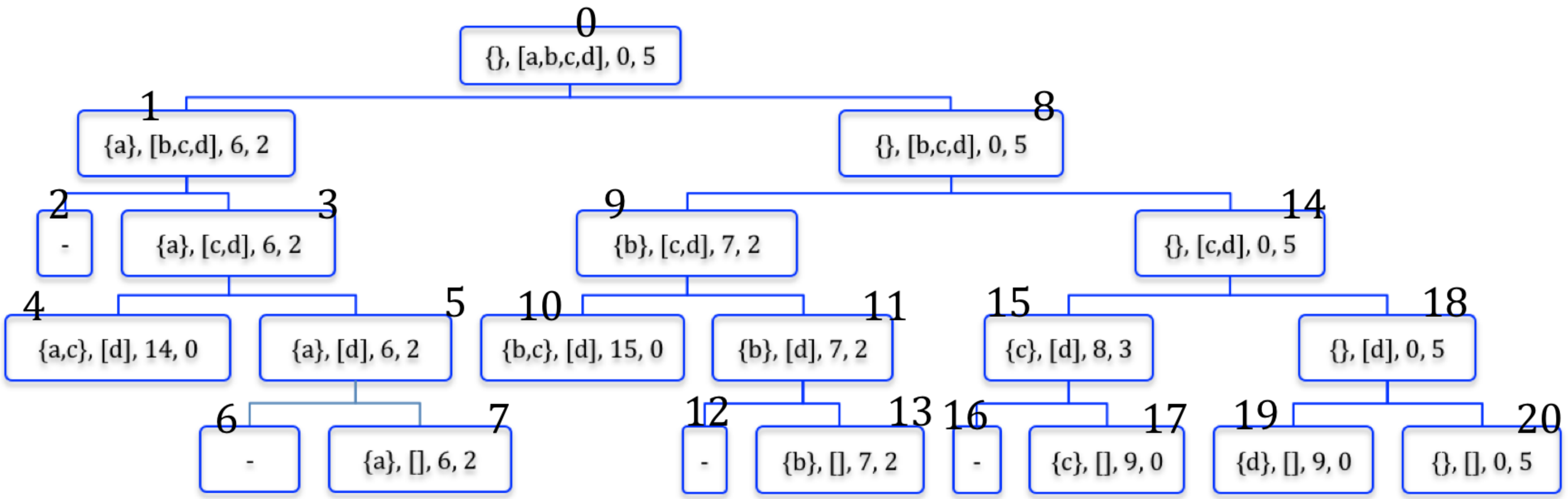
- A (perhaps partial) set of items to be taken.
- The list of items for which a decision about whether or not to take each item in the list has not been made.
- The total value of the items in the set of items to be taken. This is merely an optimization, since the value could be computed from the set.
- The remaining space in the knapsack. Again, this is an optimization since this is merely the difference between the weight allowed and the weight of all the items taken so far.

A different approach to optimal solutions

- Build the tree top down starting with the root.
- Select an element from the still to be considered items.
- If there is room for that item in the knapsack, a node is constructed that reflects the consequence of choosing to take that item. By convention, we draw that as the left child. The right child shows the consequences of choosing not to take that item.
- The process is then applied recursively to non-leaf children. Because each edge represents a decision (to take or not to take an item), such trees are called **decision trees**.

A simple example

Name	Value	Weight
A	6	3
B	7	3
C	8	2
D	9	5



- Numbers above nodes indicate order of search
- Depth first

Recursive implementation

```
def maxVal(toConsider, avail):
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getWeight() > avail:
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getWeight())
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```

Implementation and testing

```
def smallTest():
    Items = buildItems()
    val, taken = maxVal(Items, 20)
    for item in taken:
        print(item)
    print ('Total value of items taken = ' + str(val))
```

This gives us the same solution we saw earlier

But it gives us a different way of thinking about finding the solution

And it doesn’t explore the entire powerset, since it truncates search