Recursive Algorithms Implemented in Python

Reading

- Zelle 13.2
- Recursive algorithms (pseudocode, Algorithms slides)

Recursion

- A problem solving paradigm
- An approach for designing algorithms
- Given a recursive algorithm, there is always an equivalent non-recursive algorithm
  - Recursive algorithm often simpler

Recursion problem solving paradigm

- You don’t solve the problem directly
- Split the problem until it becomes trivial
- Compute solution to problem by combining solutions of sub-problems

3 elements of recursive algorithm

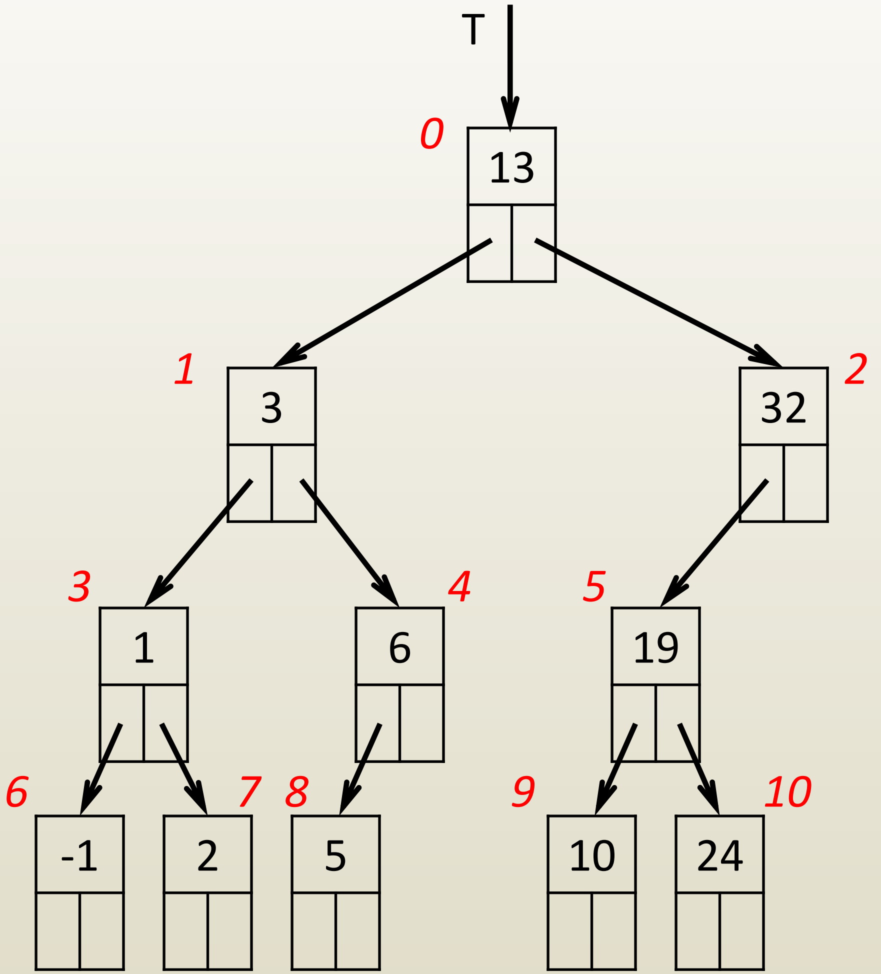
- Termination condition
  - At some point recursion has to stop
  - For example, don't go beyond leafs
    - Leafs don't have children, referring to children leafs causes algorithm to crash
- Recursive call
  - Algorithm calls itself on subsets of the input data
  - One ore more recursive calls
    - For binary tree we had two recursive calls, one for each child
- Work done before, between, and after recursive calls

Examples

- We have seen several recursive algorithms
  - Binary tree traversal
    - Counting number of nodes in binary trees
    - Evaluation of arithmetic expression stored in a binary tree
    - Printing out arithmetic expression stored in a binary tree
  - Computing factorial of n
  - Finding the minimum element of an array of numbers
  - Binary search
- Now let’s implement these and other recursive algorithms in Python

Binary trees in Python

- An array of triples (i.e. an array of arrays with 3 elements, a 2-D array of nx3 in size)
- One triple per node
  - Data, index of left child, index of right child
- First triple corresponds to root
- An index of -1 corresponds to null (i.e. no such child)
- Example for tree on the right
  - [[13, 1, 2], [3, 3, 4], [32, 5, -1], [1, 6, 7], [6, 8, -1], [19, 9, 10], [-1, -1, -1], [2, -1, -1], [5, -1, -1], [10, -1, -1], [24, -1, -1]]
  - Black numbers are data
  - Red numbers are indices



iClicker Question

- Given a binary tree *T* encoded using an array of triples, how does one test whether a node with index *currNode* is a leaf?

A. T[currNode][0] == -1 and T[currNode][1] == -1
B. T[currNode][1] == -1 and T[currNode][2] == -1
C. T[currNode][1] == -1 or T[currNode][2] == -1
D. Either B or C
E. None of the above

Counting nodes in binary tree

```
// PSEUDOCODE

Input:
    T // link to root of binary tree
Output:
    // count of nodes
CountBTR(T)
    if T == NULL
        return 0
    endif
    return CountBTR(T->left) + 1 +
                    CountBTR (T->right)
endCountBTR
```

```
# Python
# currNode is index of current node


def CountBTR(T, currNode):
   if currNode == -1:
      return 0
   left = T[currNode][1]
   right = T[currNode][2]
   return CountBTR(T, left) + 1 +
                    CountBTR(T, right)
```

Arithmetic expression evaluation

Input:

```
    T // link to root of arithm. expr. tree
```

Output:

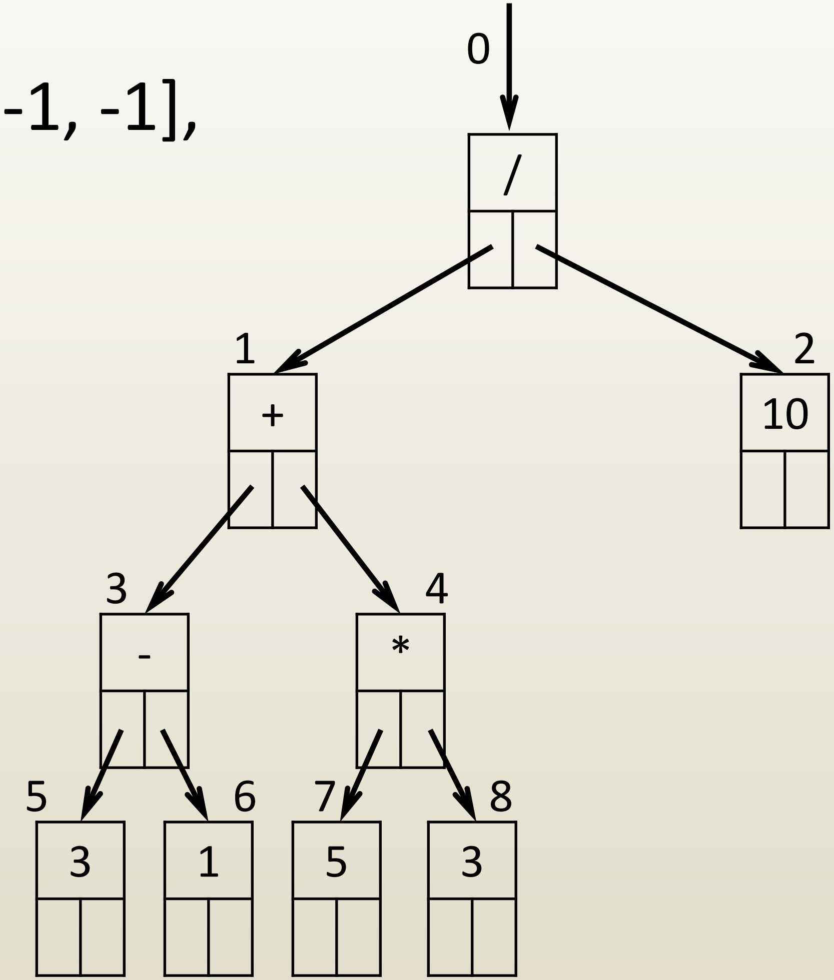
```
    // value of arithmetic expression
```

```
EvalAEBTR(T)
    if T-> left == NULL
        return T->val
    endif
    switch T->symbol
        case ‘+’: return EvalAEBTR(T->left) +
                        EvalAEBTR(T->right)
        case ‘-’: return EvalAEBTR(T->left) -
                        EvalAEBTR(T->right)
        case ‘*’: return EvalAEBTR(T->left) *
                        EvalAEBTR(T->right)
        case ‘/’: return EvalAEBTR(T->left) /
                        EvalAEBTR(T->right)
    endswitch
endEvalAEBTR
```

```
def EvalAEBTR(T, currNode):
  left = T[currNode][1]
  right = T[currNode][2]
  if left == -1:
    return T[currNode][0]
  if T[currNode][0] == '+':
    return EvalAEBTR(T, left) +
           EvalAEBTR(T, right)
  elif T[currNode][0] == '-':
    return EvalAEBTR(T, left) -
           EvalAEBTR(T, right)
  elif T[currNode][0] == '*':
    return EvalAEBTR(T, left) *
           EvalAEBTR(T, right)
  elif T[currNode][0] == '/':
    return EvalAEBTR(T, left) /
           EvalAEBTR(T, right)
```

Example

- T = [[‘/’, 1, 2], [‘+’, 3, 4], [10, -1, -1], [‘-’, 5, 6], [‘*’, 7, 8], [3, -1, -1], [1, -1, -1], [5, -1, -1], [3, -1, -1]]



Printing out arithmetic expression

Input:

T // link to root of arithmetic expression binary tree

Output:

// expression printed out with parentheses

```
PrintAEBTR(T)
    if T->left != NULL
        print “(“
        PrintAEBTR(T->left)
    endif
    print T->string
    if T->left != NULL
        PrintAEBTR(T->right)
        print “)“
    endif
endPrintAEBTR
```

```
# Example: (((3-1)+(5*3))/10) for tree on
#              previous slide


def PrintAEBTR(T, currNode):
  if currNode == -1:
    return
  left = T[currNode][1]
  right = T[currNode][2]
  if left != -1:
    print(‘(‘, end='')
    PrintAEBTR(T, left)
  print(T[currNode][0], end='')
  if right != -1:
    PrintAEBTR(T, right)
    print(‘)’, end='')
```

Drawing Binary Tree

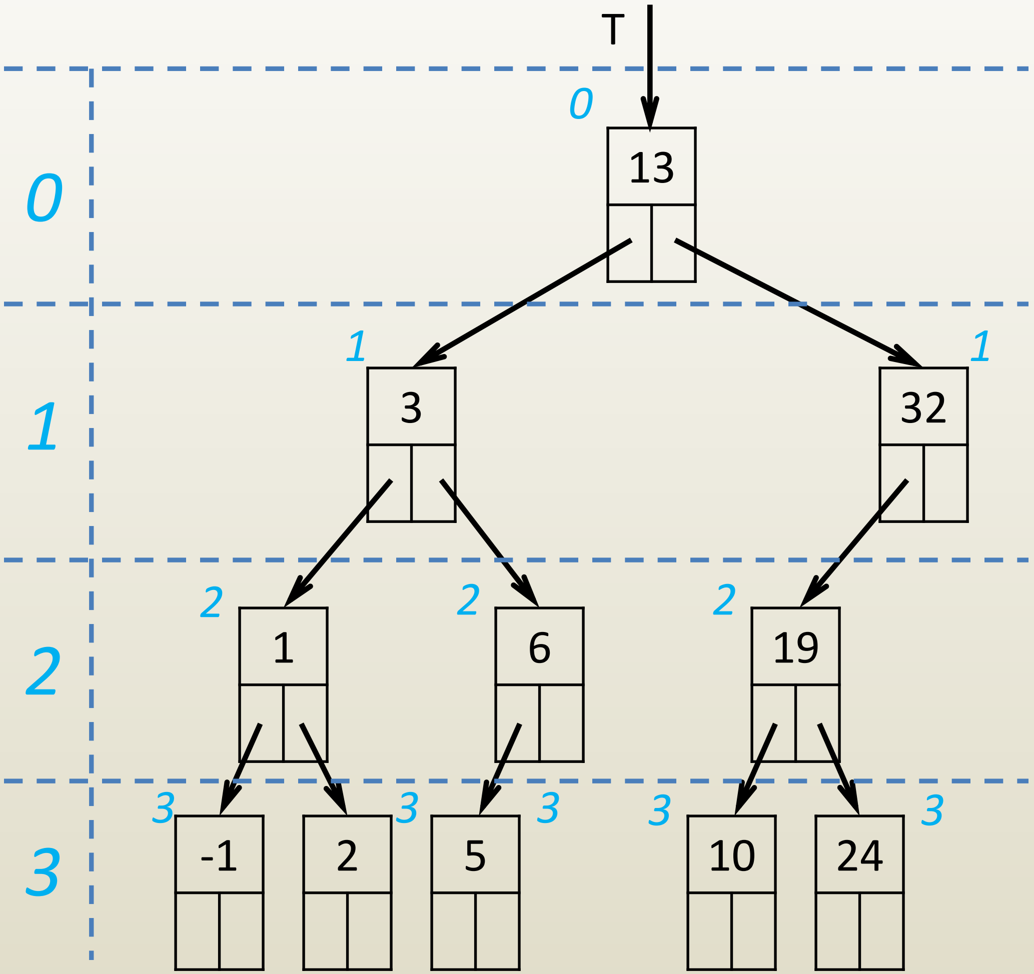
- Needed ingredients
  - How to draw
    - a node—*as a circle*
    - an edge between parent and child—*as a line segment*
    - text data—*using text object*
  - Where to draw
    - Node vertical coordinate given by depth in tree
    - Node horizontal coordinate given by in-order traversal index

2 step approach

- Step 1: set coordinates of each node
  - Recursive
  - Coordinates appended to node triple, which is now defined by 5 numbers: [data, leftIndex, rightIndex, hCoord, vCoord]
- Step 2: actually draw the nodes (and edges)
  - Recursive
  - Uses coordinates stored during step 1

Setting node coordinates

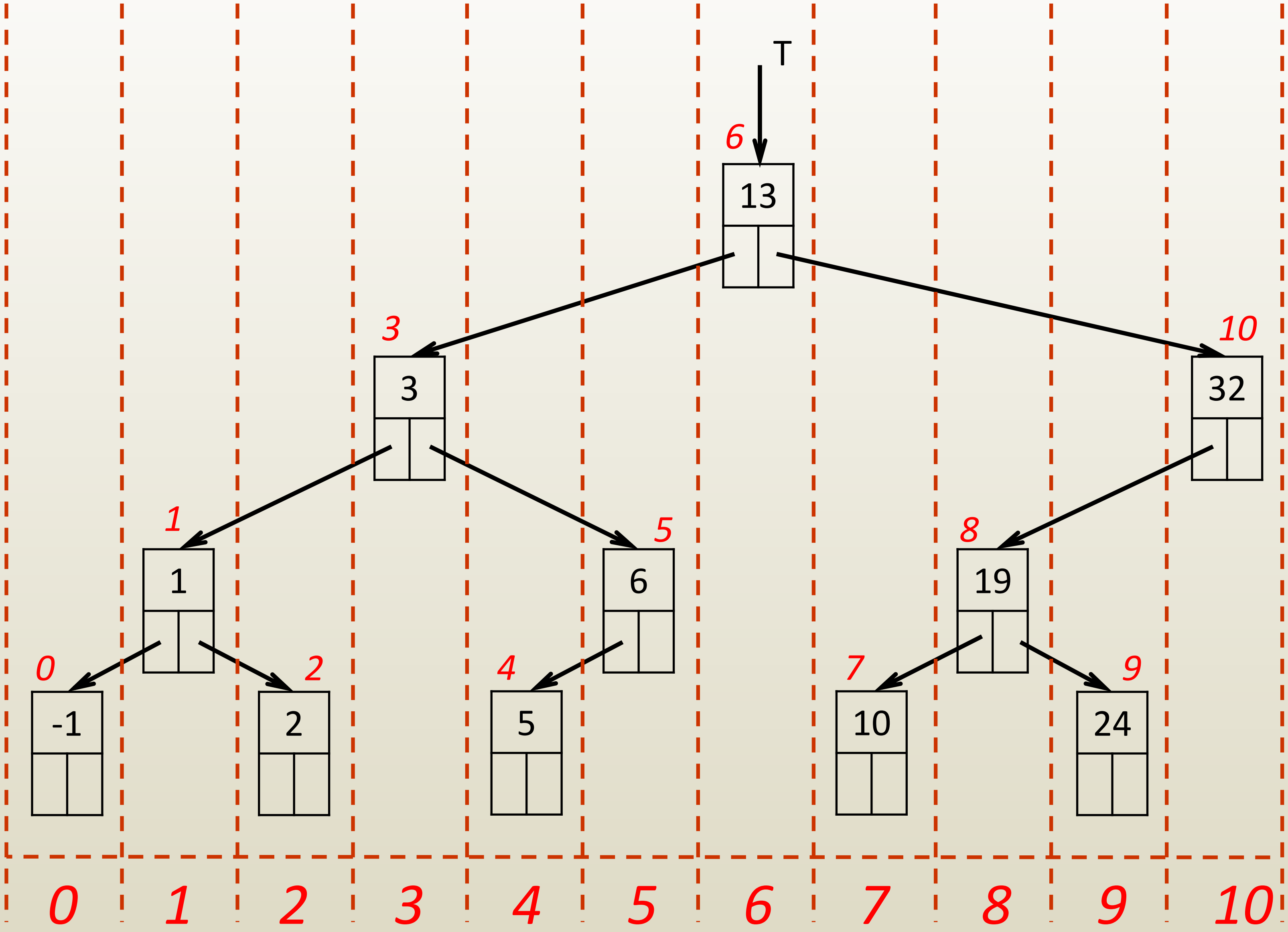
- Vertical coordinate "vCoord"
  - Equals depth in tree
  - Same for all nodes on same level



Setting horizontal node coordinates

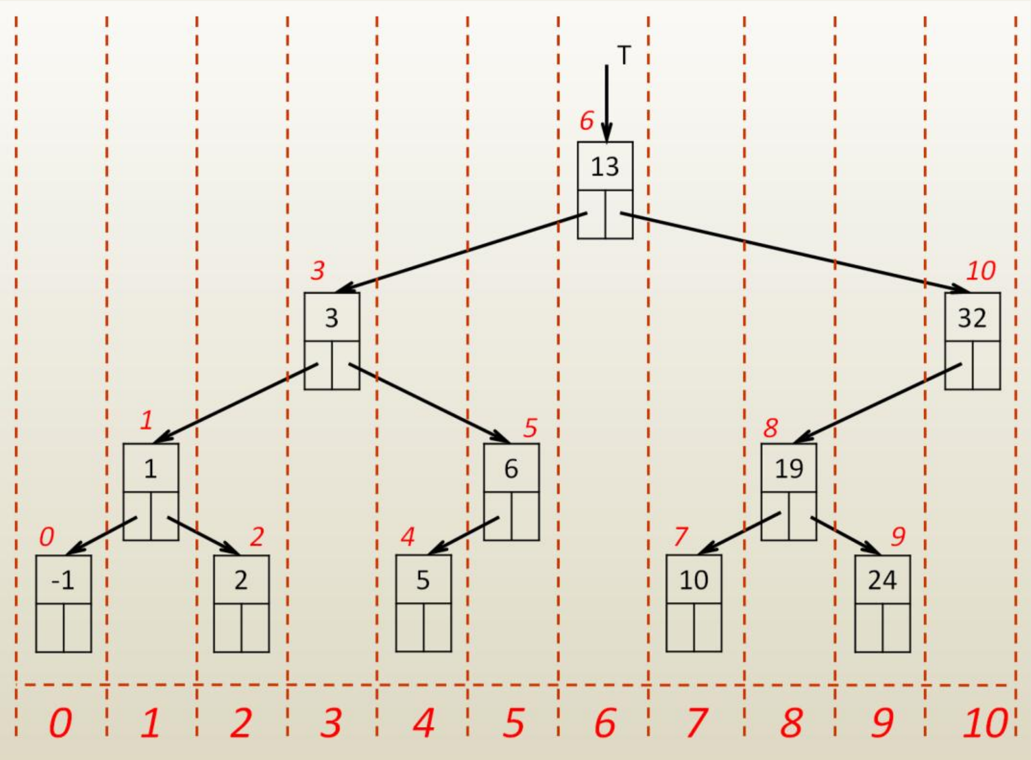
- Each node has a different coordinate
- Each node belongs to a different column
- If there are *n* nodes, we need *n* columns
- A node has a greater coordinate than any node in its left sub-tree
- Left-most node has coordinate 0

Setting horizontal node coordinates

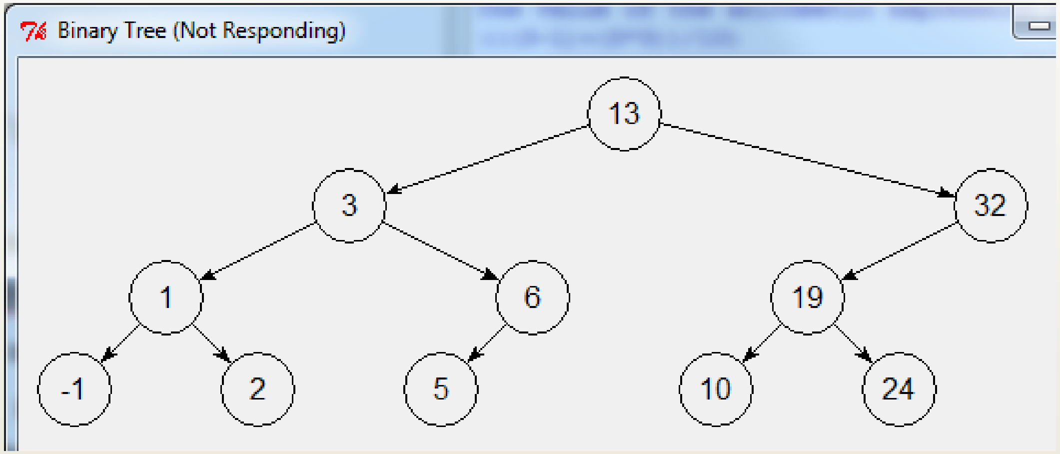


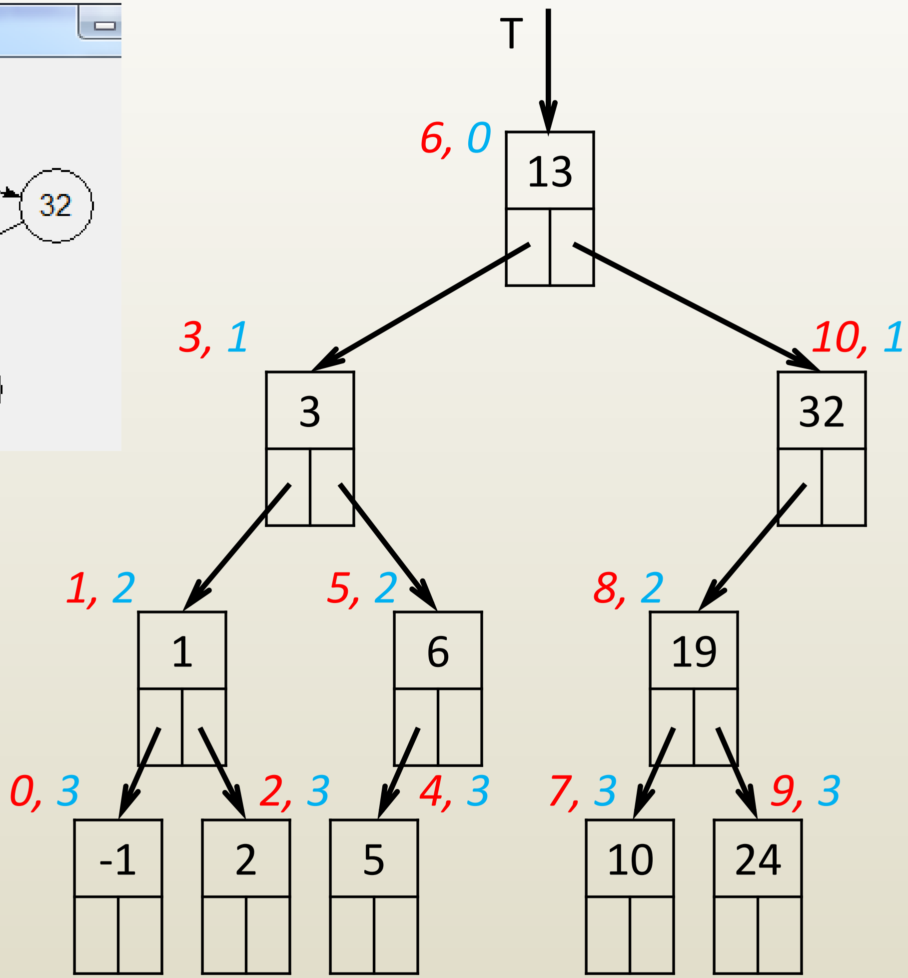
Setting horizontal node coordinates

- Each node has a different coordinate
- Each node belongs to a different column
- If there are *n* nodes, we need *n* columns
- A node has a greater coordinate than any node in its left sub-tree
- Left-most node has coordinate 0

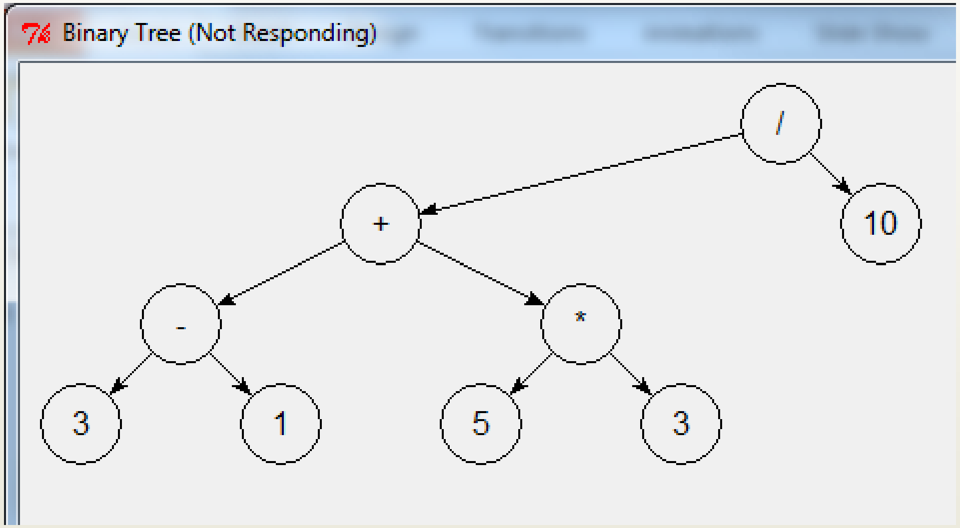


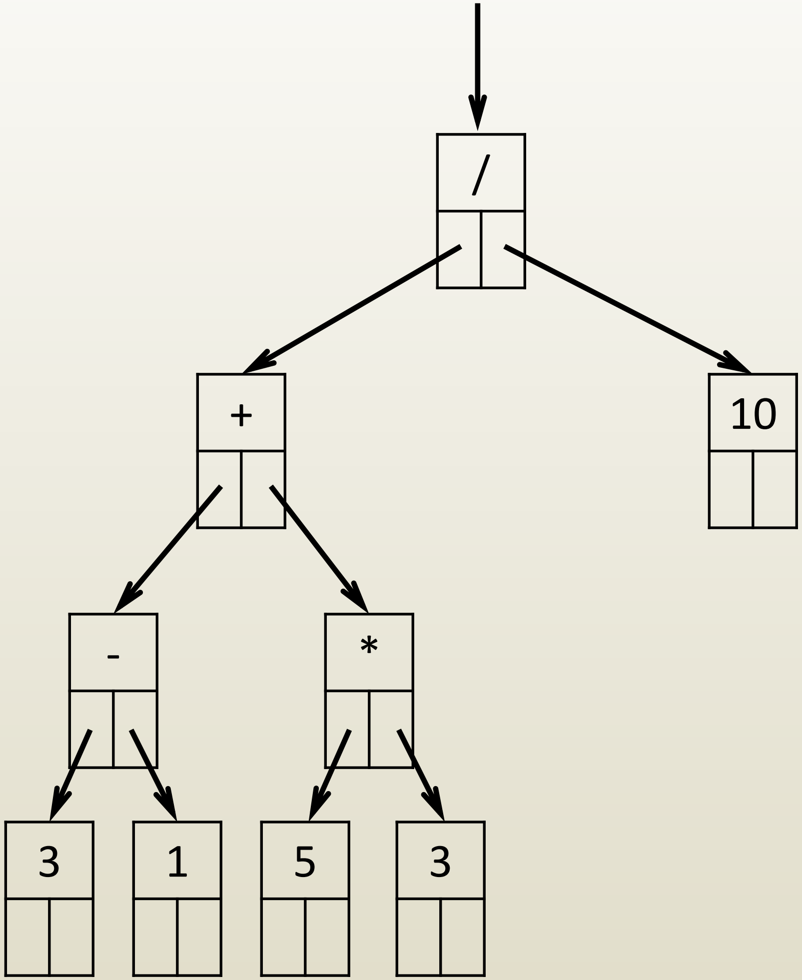
Example 1


Binary Tree (Not Responding)
13
3
32
1
6
19
-1
2
5
10
24

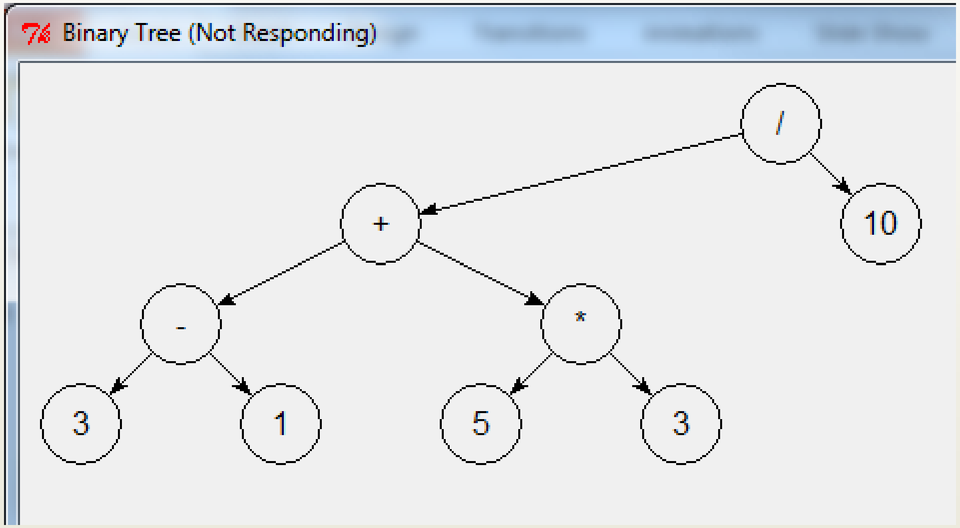

T
6, 0
13
3, 1
10, 1
3
32
1, 2
5, 2
8, 2
1
6
19
0, 3
2, 3
4, 3
7, 3
9, 3
-1
2
5
10
24

Example 2





iClicker question



Our tree drawing algorithm doesn't place the root half way btw. its children

A. The implementation is not correct.

B. The implementation is correct.

C. We could enhance the algorithm to avoid this.

D. B and C

E. A and C

Computing factorial

```
Input:
   n // factorial function argument
Output:
   // n!
Factorial(n)
   if n == 1
      return 1
   endif
   return n*Factorial(n-1)
endMinimum
```

```
def Fact(n):
   if n == 1:
      return 1
   return n * Fact(n-1)
```

Finding minimum in array

```
Input:
  A // array of integers
  n // number of elements in array (array size)
  $i_0$ // consider elements from $i_0$ onwards
Output:
  Min // value of element with smallest value
MinR(A, n, $i_0$) // recursive version
  if i0 == n-1 // last element
    return A[i0]
  endif
  tmp = MinR(A, n, i0+1)
  if A[i0] < tmp
    return A[$i_0$]
  else
    return tmp
endMinR
```

```
def MinR(A, i0):
  if i0 == len(A)-1:
    return A[i0]
  tmp = MinR(A, i0+1)
  if A[i0] < tmp:
    return A[i0]
  else:
    return tmp
```

Binary search

- Finds whether a number appears in a sorted array in logarithmic time
  - logn, where n is the number of elements in the array

```
def BinSearchR(A, l, r, a):
  if l == r:
    return A[l] == a
  m = (int) ((l+r)/2)
  if A[m] >= a:
    return BinSearchR(A, l, m, a)
  else:
    return BinSearchR(A, m+1, r, a)
```