Announcements

- Project 5 is due Dec. 6.
- Second part is essay questions for CoS teaming requirements.
  - The first part you do as a team
  - The CoS essay gets individually answered and has separate submission instructions on the home page
- Final on Dec 11 in EE 129, 10:30 – 12:30
  - Also posted on course home

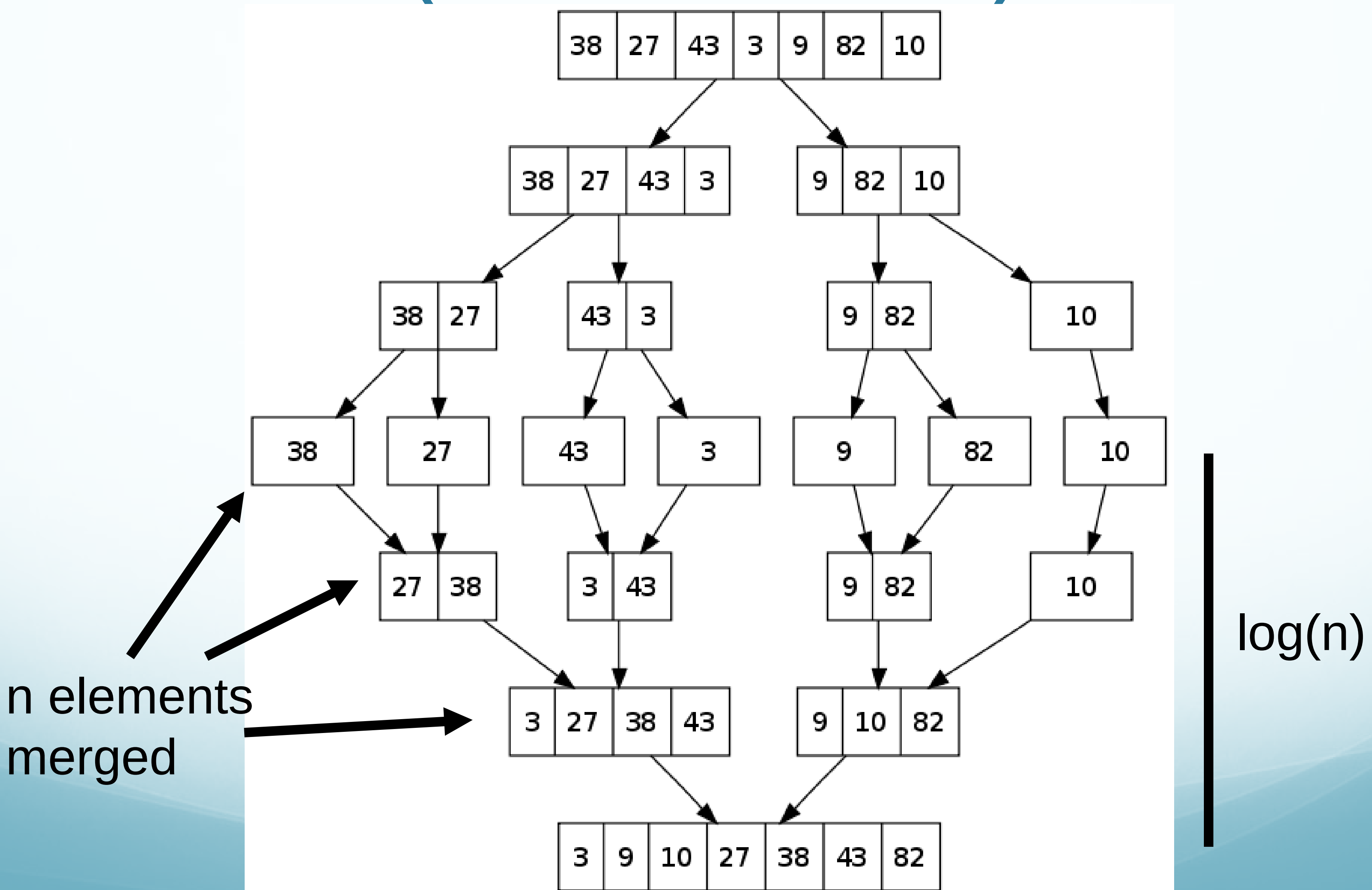
Recursion

- Divide & Conquer
  - Merge sort
  - Binary search
- Permutations generated recursively
  - Using strings (Anagrams)
  - Using lists (making type distinctions
- Tail recursion
  - Unrolling recursive string reversal
  - Unrolling recursive binary search
- Recursive tree traversals
  - Special case priority queue (heap)
  - Special case expression tree
  - General tree traversal

Key Insight

- To understand and be able to program recursively, you must
  - Break down the problem into sub problems and
  - Join the solution of those sub problems back to get the solution of the original problem.
- Merge sort is a good example

Visual Representation (see week 13)



Recall Binary Search

- The basic idea of the binary search algorithm was to iteratively divide the problem in half.
- This technique is known as the *divide and conquer* approach in algorithm design
- Divide and conquer divides the original problem into sub-problems that are smaller versions of the original problem.

Recursive Algorithm for Binary Search

```
def binarySearch(key, low, high, numlist):
    mid = (low + high)//2
    if low > high:
        return -1
    if key == numlist[mid]:
        return mid
    elif key < numlist[mid]:
        return binarySearch(key, low, mid-1, numlist)
    else:
        return binarySearch(key, mid+1, high, numlist)
```

Recursive Definitions Rules

1. All good recursive definitions have these two key characteristics:
   - There are one or more base cases for which no recursion is applied:
     - Empty search interval for binary search
     - Length 1 lists for merge sort
   - All chains of recursion eventually end up at one of the base cases.
     - After probing the mid entry of the search segment, recursion reduces the search interval by half, an ideal case
     - Merge sort splits the list into two halves, each smaller than the input

2. The simplest way for these two conditions to occur is for each recursion to act on a smaller version of the original problem. A very small version of the original problem that can be solved without recursion becomes the base case.
     - See (1)

Example of Call Sequence

Search 7 in L=[1,3,4,5,6,8,9]:

- (**7**,*0,6*,L): call
  - *0 <= 6* bnds chk
  - *(0+6)//2 => 3* mid
  - **7** != 5 key comp
  - (**7**,*4,6*,L): recursion
    - *4 <= 6* bnds chk
    - *(4+6)//2 => 5* mid
    - **7** != 8 key comp
    - (**7**,*4,4*,L): …
      - *4 <= 4*

Continued…

      - *(4+4)//2 => 4*
      - **7** != 6
      - (**7**,*5,4*,L):
        - *5 > 4*
        - return -1
      - return -1
    - return -1
  - return -1
- function has returned

Example: Permutations

- All possible orderings of numbers 1 through n encode the *permutations* of n objects.
- Let's generate all permutations recursively.
  - Caution: there are $n!$ permutations of n objects

1,2,3,4	2,1,3,4	2,3,1,4	2,3,4,1
1,3,2,4	3,1,2,4	3,2,1,4	3,2,4,1
1,3,4,2	3,1,4,2	3,4,1,2	3,4,2,1
1,2,4,3	2,1,4,3	2,4,1,3	2,4,3,1
1,4,2,3	4,1,2,3	4,2,1,3	4,2,3,1
1,4,3,2	4,1,3,2	4,3,1,2	4,3,2,1

Example: Permutations

- All possible orderings of numbers 1 through n encode the *permutations* of n objects.
- Let's generate all permutations recursively.
  - Caution: there are $n!$ permutations of n objects

1,2,3,4	2,1,3,4	2,3,1,4	2,3,4,1
1,3,2,4	3,1,2,4	3,2,1,4	3,2,4,1
1,3,4,2	3,1,4,2	3,4,1,2	3,4,2,1
1,2,4,3	2,1,4,3	2,4,1,3	2,4,3,1
1,4,2,3	4,1,2,3	4,2,1,3	4,2,3,1
1,4,3,2	4,1,3,2	4,3,1,2	4,3,2,1

CQ

There are X permutations of 4 objects, where X is:

A. About 12
B. 24
C. 36
D. 60

Example: Permutations

- Let’s apply this approach
  - Slice the first character off the string.
  - Place the first character in all possible locations within the permutations formed from the “rest” of the original string.

Permuting Characters

- Suppose the original string is “123”. Stripping off the “1” leaves us with “23”.
- Generating all permutations of “23” gives us “23” and “32”.
- To form the permutations of the original string, we place “1” in all possible locations within these two permutations:
  [“123”, “213”, “231”,   “132”, “312”, “321”]

Example: Permutations

- As in the previous example, we can use the empty string as our base case.

- ```
  def permute(s):
      if s == "":
          return [s]
      else:
          ans = []
          for w in permute(s[1:]):
              for pos in range(len(w)+1):
                  ans.append(w[:pos]+s[0]+w[pos:])
          return ans
  ```
  ```

Example: Permutations

- A list is used to accumulate results.
- The outer **`for`** loop iterates through each permutation of the tail of **`s`**.
- The inner loop goes through each position in the permutation and creates a new string with the original first character inserted into that position.
- The inner loop goes up to **`len(w)+1`** so the new character can be added also at the end of the tail permutation.

Example: Permutations

- **`w[:pos]+s[0]+w[pos:]`**
  - **`w[:pos]`** gives the part of **`w`** up to, but not including, **`pos`**.
  - **`w[pos:]`** gives everything from **`pos`** to the end.
  - Inserting **`s[0]`** between them effectively inserts it into **`w`** at **`pos`**.

Now do it with list argument instead of strings

```
def permute(s):
    if s == []:     # was “”
        return s    # was [s]
    else:
        ans = []
        for w in permute(s[1:]):
            for pos in range(len(w)+1):
                ans.append(w[:pos]+s[0]+w[pos:])
        return ans
```

Demo Code

Recursive String Reversal

- Using recursion, we can calculate the reverse of a string without the intermediate list step.
- Think of a string as a recursive object:
  - Divide it up into a first character and “all the rest”
  - Reverse the “rest” and append the first character to the end of it
- Elegant, but don’t forget the base case!

String Reversal?

- ```
  def reverse(s):
      return reverse(s[1:]) + s[0]
  ```
- The slice **`s[1:]`** returns all but the first character of the string.
- We reverse this slice and then concatenate the first character (**`s[0]`**) onto the end.
  ```

String Reversal?

- ```
  >>> reverse("Hello")

  Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
      reverse("Hello")
    File "C:/Program Files/Python 2.3.3/z.py", line 8, in
  reverse
      return reverse(s[1:]) + s[0]
    File "C:/Program Files/Python 2.3.3/z.py", line 8, in
  reverse
      return reverse(s[1:]) + s[0]
  …
   File "C:/Program Files/Python 2.3.3/z.py", line 8, in
  reverse
      return reverse(s[1:]) + s[0]
  RuntimeError: maximum recursion depth exceeded
  ```
- What happened? There were 1000 lines of errors!
  ```

Example: String Reversal

- Remember: To build a correct recursive function, we need a base case that doesn't use recursion.
- We forgot to include a base case, so our program is an *infinite recursion*. Each call to **`reverse`** contains another call to **`reverse`**, so none of them return.

Example: String Reversal

- ```
  def reverse(s):
      if s == "":
          return s
      else:
          return reverse(s[1:]) + s[0]
  ```

- ```
  >>> reverse("Hello")
  'olleH'
  ```

Announcements

- Posted slides updated
- Project 5 is due Dec. 6.
- Second part is essay questions for CoS teaming requirements.
  - The first part you do as a team
  - The CoS essay gets individually answered and has separate submission instructions on the home page
- Final on Dec 11 in EE 129, 10:30 – 12:30
  - Also posted on course home

Tail Recursion

- Characteristic code pattern:

```
def f(X):
    <base case condition & computation>
    <some computation>  #f(X’) at the end
    return result
```

- Can be changed into a loop:

```
def f(X):
    <base case computation>
    while not <base case condition>:
        <some computation>  #X’, inverted
    return result
```

Tail Recursion

- Characteristic code pattern:

```
def revStringRec(L):
    if len(L) == 0:
        return L
    R = revStringRec(L[1:]) + L[0]
    return R
```

- Can be changed into a loop

```
def revStringLoop(L):
    R = ''
    while len(L) != 0:
        R = L[0] + R
        L = L[1:]
    return R
```

Example

rev(‘abc’): ‘cba’

‘a’; rev(‘bc’) ‘cb’+’a’

‘b’; rev(‘c’) ‘c’+’b’

‘c’; rev(‘’) ‘’+’c’

‘’

‘abc’;		‘’
‘a’;	‘bc’;	‘a’+’’
‘b’;	‘c’;	‘b’+’a’
‘c’;	‘’;	‘c’+’ba’
		‘cba’

Inverse also true

- An algorithm with a main loop can also be recast recursively, using tail recursion
- Unroll the loop and look for the pattern

Tail Recursion

Loop can be changed into a recursion:

```
def revStringLoop(L):
    R = ''
    while len(L) != 0:
        R = L[0] + R
        L = L[1:]
    return R
```

Outcome:

```
def revStringRec(L):
    if len(L) == 0:
        return L
    R = revStringRec(L[1:]) + L[0]
    return R
```

Another Conversion Example

```
def binarySearch(key, low, high, numlist):
    mid = (low + high)//2
    if low > high:
        return -1
    if key == numlist[mid]:
        return mid
    elif key < numlist[mid]:
        return binarySearch(key, low, mid-1, numlist)
    else:
        return binarySearch(key, mid+1, high, numlist)
```

Another convertible example

```
def binarySearch(key, low, high, numlist):
    mid = (low + high)//2
    while low <= high:
        if key == numlist[mid]:
            return mid
        elif key < numlist[mid]:
            high = mid-1
        else:
            low = mid+1
  return -1
```

Heap Traversals

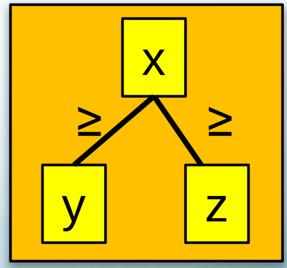
- We discussed heaps (priority queues) in week 13
- Data structure is conceptually a complete binary tree
- Encoded as a flat list, filling the tree layer by layer
- Index mappings for parent → child and child → parent
- Parent not smaller than children (max heap)

Access Mappings:

Parent to left child: $k \text{ -> } 2k+1$

Parent to right child: $k \text{ -> } 2k+2$

Child to parent: $k \text{ -> } (k-1)//2$



CQ: encoding of a valid heap?

[9,7,4,6,5,2,2,1,1,1,1]

A. Yes

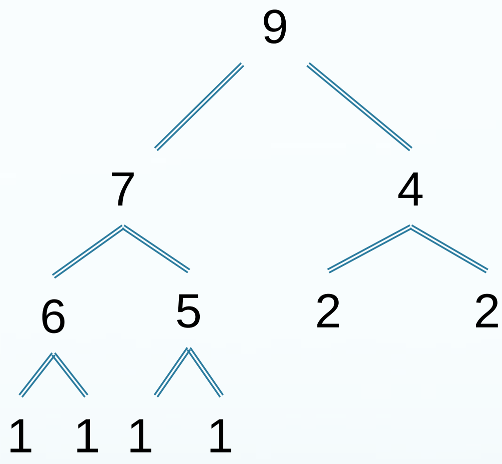
B. No

CQ: encoding of a valid heap?

[9,7,4,6,5,2,2,1,1,1,1]

A. Yes

B. No



CQ: encoding of a valid heap?

[9,7,4,6,5,5,2,2,1,1,1]

A. Yes

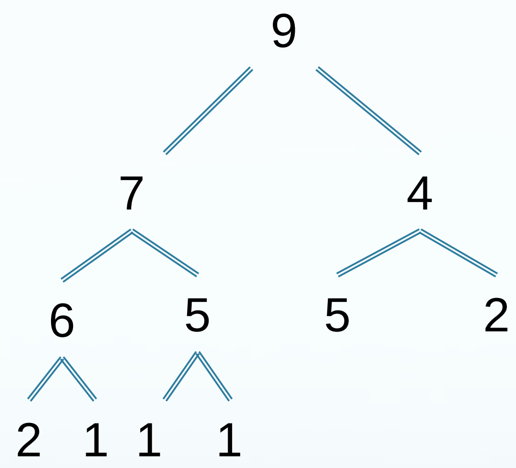
B. No

CQ: encoding of a valid heap?

[9,7,4,6,5,5,2,2,1,1,1]

A. Yes

B. No



CQ: how many children for L[5]?

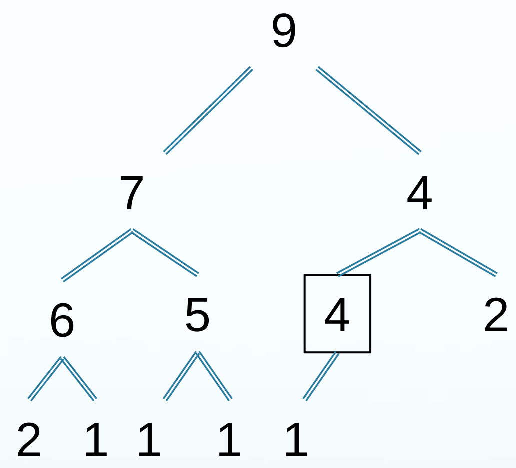
[9,7,4,6,5,4,2,2,1,1,1,1]

A. 0
B. 1
C. 2

CQ: how many children for L[5]?

[9,7,4,6,5,4,2,2,1,1,1,1]

A. 0
B. 1
C. 2



Special Traversal

root to leaf, always leftmost:

```
k = 0
while k < len(L):
   # work on node L[k]
   k = 2*k+1
```

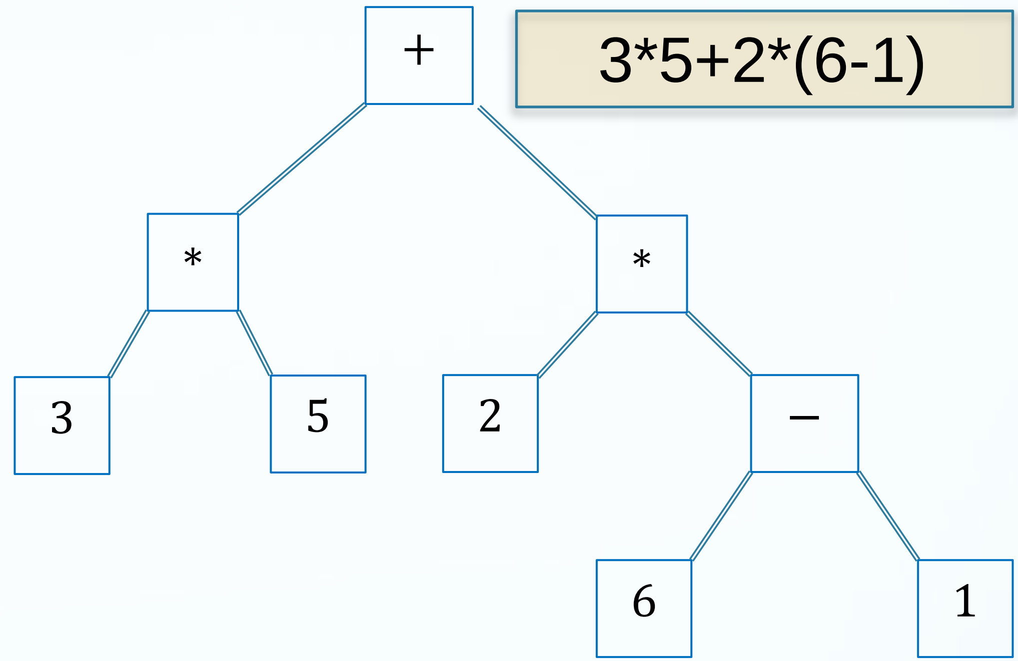
last leaf to root:

```
k = len(L)-1
while k >= 0:
    # process node L[k]
    k = (k-1)//2
```

Expression Tree Traversals

- Preorder:
  - visit node, visit left subtree, visit right subtree
- Inorder:
  - visit left subtree, visit node, visit right subtree
- Postorder:
  - visit left subtree, visit right subtree, visit node

3*5+2*(6-1)



+,*,3,5,*,2,-,6,1

3,*,5,+,2,*,6,-,1

3,5,*,2,6,1,-,*,+

Pre-, In- & Postorder

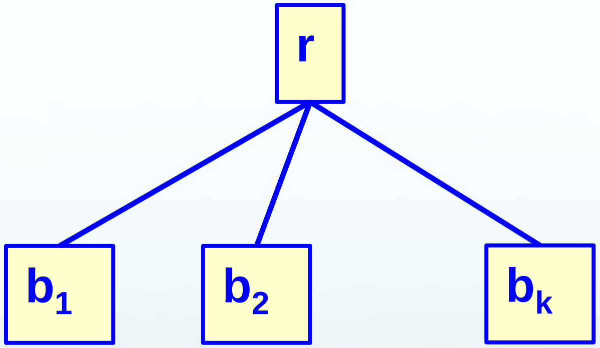
```
def preorder(E):
    print root label
    if E is not a leaf:
        preorder(left(E))
        preorder(right(E))
```

```
def inorder(E):
    if E is not a leaf:
        inorder(left(E))
    print root label
    if E is not a leaf:
        inorder(right(E))
```

```
def postorder(E):
    if E is not a leaf:
        preorder(left(E))
        preorder(right(E))
    print root label
```

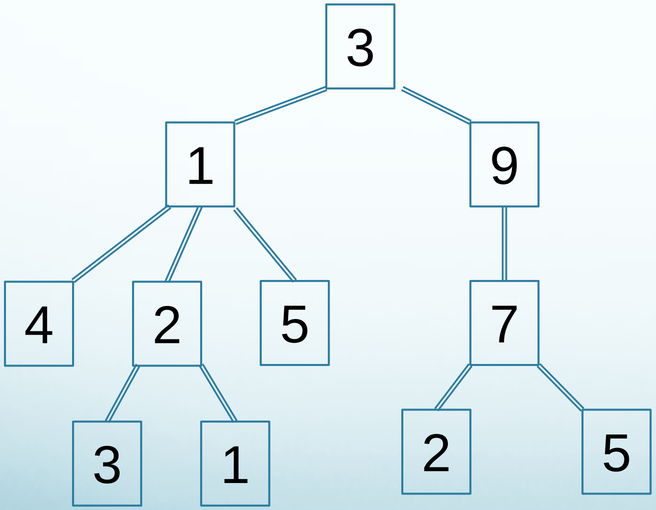
Tree Encoding

- **[r, b_1, …, b_k]** encodes the node r and its descendants
- Nesting builds up the tree
- It is a preorder encoding !!!



Summing all Node Values

- Assume given a list all of whose elements are numbers or sublists of numbers, nested arbitrarily
- This list encodes a tree all of whose nodes, including leaves, are labeled with a number
- We want to sum all numbers in the tree



[3,[1,4,[2,3,1],5],[9,[7,2,5]]]

no distinction between node and subtree...!

```
def sumTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    sum = 0
    for L1 in L:
        sum = sum + sumTree(L1)
    return sum
```

Pre-, In- or Postorder?

```
def sumTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    sum = L[0]
    for L1 in L[1:]:
        sum = sum + sumTree(L1)
    return sum
```