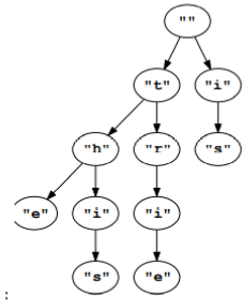
```
def make_trie(words):
    """ Makes a tree where every node is a letter of a wor
        All words end as a leaf of the tree.
        words is given as a list of strings.
    """
    trie = Tree('')
    for word in words:
        add_word(trie, word)
    return trie


def add_word(trie, word):

    if word == '':
        return

    branch = None

    for b in trie.branches:

        if b.label == word[0]:

            branch = b

    if not branch:

        branch = Tree(word[0])

        trie.branches.append(branch)

    add_word(branch, word[1:])
```




```
def get_words(trie)
    if trie.is_leaf():
        return [trie.label]
    return sum([[trie.label + word for word in get_words(branch)] for branch in trie.branches], [])
```

```
def separate(separator, lnk):
    if lnk is Link.empty:
        return Link.empty
    elif lnk.rest is Link.empty:
        return Link(lnk.first)
    elif lnk.first == lnk.rest.first:
        return Link(lnk.first, Link(separator, separate(separator, lnk.rest)))
    return Link(lnk.first, separate(separator, lnk.rest))
```

```
def digit_replacer(predicate, transformer):
    def func(n):
        if n == 0:
            return 0
        digit = n % 10
        if predicate(digit):
            digit = transformer(digit)
        return func(n // 10) * 10 + digit
    return func
```

```
def thanos_messenger(word):
    """A messenger function that discards every other word.

    >>> thanos_messenger("I")("don't")("feel")("so")("good")(".")
    'I feel good.'
    >>> thanos_messenger("Thanos")("always")("kills")("half")(".")
    'Thanos kills.'
    """
    assert word != '.', 'No words provided!'
    def make_new_messenger(message, skip_next):
        def new_messenger(word):
            if word == '.':
                return message + '.'
            if skip_next:
                return make_new_messenger(message, False)
            return make_new_messenger(message + " " + word, True)
        return new_messenger
    return make_new_messenger(word, True)
```

```
class LearnableContent: :
  def __init__(self, title, author):
    self.title = title self.author = author
  def __str__(self):
    return f"{self.title} by {self.author}"
class Video(LearnableContent):
    license = 'CC-BY-NC-SA'
    def __init__(self, title, author, num_seconds):
        super().__init__(title, author)
        self.num_seconds = num_seconds
    def __str__(self):
        return super().__str__() +
      f" ({self.num_seconds} seconds)"
```

```
max([('A', 1), ('L', 8), ('P', 5)],key=lambda tup: tup[1])
Out[27]: ('L', 8)
```

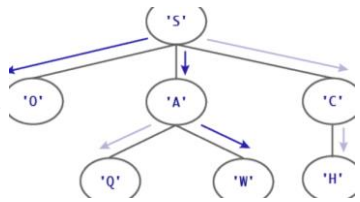
‘Print(l) ‘= str(l),’l’=repar(l)

```
artist = "Lil Nas X“  song = "Industry Baby“  place = 2
print("Debuting at #" + str(place) + ": '" + song + "' by " + artist)
print(f"Debuting at #{place}: '{song}' by {artist}")
Debuting at #2: 'Industry Baby' by Lil Nas X
```

```
def __repr__(self):
return f"PaperReam({repr(self.color_name)}, {repr(self.num_sheets)})" OR
return f"PaperReam('{self.color_name}', {self.num_sheets})"
```

```
def count_paths(t, total):
 if label(t) == total: found = 1
 else: found = 0
 return found + sum([count_paths(b,
 total - label(t)) for b in
branches(t)])
def count_leaves(t):
 if is_leaf(t): return 1
 else:
   leaves_under = 0
   for b in branches(t):
   leaves_under += count_leaves(b)
 return leaves_under
def tree_map(t):
 return
tree(fn(label(t)),[tree_map(fn,t)]
       for t in branches(t))
```

```
def word_finder(letter_tree, words_list):
    def string_builder(t, str):
        str += t.label

        if t.is_leaf() and str in words_list:
            yield str
        for b in t.branches:
            yield from string_builder(b, str)

    yield from string_builder(letter_tree, "")
```



>words = ['SO', 'SAT', 'SAME', 'SAW', 'SOW']
> list(word_finder(t, words))
['SO', 'SAW']

```
def max_tree(t, key):
    """Return the label n of t for whi

    >>> t = Tree(6, [Tree(3, [Tree(5)]
    >>> max_tree(t, key=lambda x: x)
    7
    >>> max_tree(t, key=lambda x: -x)
    2
    >>> max_tree(t, key=lambda x: -abs
    4
    """
    if t.is_leaf():

        return t.label

    x = t.label

    for b in t.branches:

        m = max_tree(b, key)

        if key(m) > key(x):

            x = m

    return x
```

```
def layer(t, d):
    """Return a linked list containing al

    >>> a_tree = Tree(1, [Tree('b', [Tree
    ...                   Tree('a', [Tree
    ...                   Tree('d', [Tree
    >>> print(layer(a_tree, 0))
    <1>
    >>> print(layer(a_tree, 1))
    <b a d>
    >>> print(layer(a_tree, 2))
    <mas co t>
    >>> print(layer(a_tree, 3))
    <!>
    """
    return helper(t, d, Link.empty)


def helper(t, d, s):

    if d == 0:

        return Link(t.label, s)

    else:

        for b in reversed(t.branches):

            s = helper(b, d - 1, s)

        return s
```

```
>>> print_tree(exp_tree(lst))
6561
  3
    3
    1
  8
    2
    3
"""
if len(values) == 1
    return tree(values[0])
else:
    def tree_at_split(i):
        base = exp_tree(_________)
                          #  (c)
        exponent = exp_tree(_________)
                          #  (d)
        return tree(_________, [base, exponent])
                          #  (e)
    trees = [tree_at_split(i) for i in range(1, len(values))]
    return max(trees, key=_________)
```

C:values[:i]
d:values[i:]
e:label(base) **
label(exponent)
f:label

[max_tree(b, key) for b in t.branches], key=key

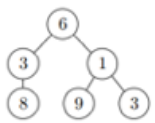
```
class C:
    x = 'e'
    def f(self, y):
        self.x = self.x + y
        return self

    def __str__(self):
        return 'go'

class Big(C):
    x = 'u'
    def f(self, y):
        C.x = C.x + y
        return C.f(self, 'm')

    def __repr__(self):
        return '<bears>'

m = C().f('i')
n = Big().f('o')
```

[m.x, n.x]: ['ei', 'um']
[C.f(n, 'a').x, C().x] ['uma', 'eo']
print(m, n) go go
n <bears>

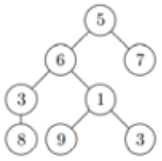
```
def preorder(t):
    """Return a list of the entries in this tre
    would be visited by a preorder traversal (s

    >>> numbers = tree(1, [tree(2), tree(3, [tr
    >>> preorder(numbers)
    [1, 2, 3, 4, 5, 6, 7]
    >>> preorder(tree(2, [tree(4, [tree(6)])]))
    [2, 4, 6]
    """
    if branches(t) == []:
        return [label(t)]
    flattened_branches = []
    for child in branches(t):
        flattened_branches += preorder(child)
    return [label(t)] + flattened_branches
```

```
def max_path(t, k):
    """ Return a list of the labels on any path in tree t of length at most k

    >>> t1 = tree(6, [tree(3, [tree(8)]), tree(1, [tree(9), tree(3)])])
    >>> max_path(t1, 3)
    [6, 3, 8]
    >>> max_path(t1, 2)
    [3, 8]
    >>> t2 = tree(5, [t1, tree(7)])
    >>> max_path(t2, 1)
    [9]
    >>> max_path(t2, 2)
    [5, 7]
    >>> max_path(t2, 3)
    [6, 3, 8]
    >>> max_path(t1, 4)
    [6, 3, 8]
    """
    def helper(t, k, on_path):
        if k == 0:
            return []
        elif is_leaf(t):
            return [label(t)]
        a = [[label(t)] + helper(b, k - 1, True) for b in branches(t)]
        if on_path:
            return max(a, key = sum)
        else:
            b = [helper(b, k, False) for b in branches(t)]
            return max(a + b, key = sum)
    return helper(t, k, False)
```

t1



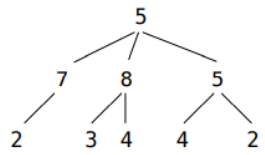
t2




```
def filter_index(f, s):
    """Return a Link containing the eleme
    which f(i) is a true value.

    >>> powers = Link(1, Link(2, Link(4,
    >>> filter_index(lambda x: x < 4, pow
    Link(1, Link(2, Link(4, Link(8))))
    >>> filter_index(lambda x: x % 2 == 1
    Link(2, Link(8, Link(32)))
    """

    def helper(i, s):

        if s is Link.empty:

            return s
                helper(i + 1, s.rest)
        filtered_rest = _________
                                (a)
        if    f(i)   __:
               (b)

            return _________
          Link(s.first, filtered_rest)
        else:

            return filtered_rest
    return   helper(0, s)
                (d)
```

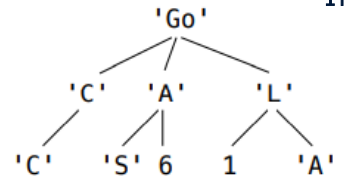
```
def max_path_sum(t):
if is_leaf(t): return label(t)
else: return label(t) +
max([max_path_sum(b) for b in
branches(t)])
```

k:




```
def find_path(t, x):
if label(t) == x: return [label(t)]
for b in branches(t):
   path = find_path(b, x)
   if path:
     return [label(t)] + path
```

v:




```
def height(t):
 if is_leaf(t): return 0
 return 1 + max([height(branch)
for branch in branches(t)])
```

```
def lookups(k, key):
    """Yield one lookup function for each node of k that

    >>> [f(v) for f in lookups(k, 2)]
    ['C', 'A']
    >>> [f(v) for f in lookups(k, 3)]
    ['S']
    >>> [f(v) for f in lookups(k, 6)]
    []
    """

    if k.label == key:

        yield lambda v: v.label

    for i in range(len(k.branches)):

        for lookup in lookups(k.branches[i], key):

            yield new_lookup(i, lookup)

def new_lookup(i, f):

    def g(v):

        return f(v.branches[i])

    return g
```

```
def sum_tree(t):
total = 0
for b in branches(t): total +=sum_tree(b)
return label(t) + total
```

```
def balanced(t):
for b in branches(t):
 if sum_tree(branches(t)[0]) != sum_tree(b)
or not balanced(b): return False
return True
```

```
def hailstone_tree(n, h):
    """Generates a tree of hailstone numbers that will reach N, with height H
    >>> print_tree(hailstone_tree(1, 0))
    1
    >>> print_tree(hailstone_tree(1, 4))
    1
        2
            4
                8
                    16
    >>> print_tree(hailstone_tree(8, 3))
    8
        16
            32
                64
            5
                10
    """
    if h == 0:
        return tree(n)
    branches = [hailstone_tree(n * 2, h - 1)]
    if (n - 1) % 3 == 0 and ((n - 1) // 3) % 2 == 1 and (n - 1) // 3 > 1:
        branches += [hailstone_tree((n - 1) // 3, h - 1)]
    return tree(n, branches)

def print_tree(t):
    def helper(i, t):
        print("    " * i + str(label(t)))
        for b in branches(t):
            helper(i + 1, b)
    helper(0, t)
```

```
def count_palindromes(L):
    """The number of palindromic words in the sequence of strings
    L (ignoring case).

    >>> count_palindromes(("Acme", "Madam", "Pivot", "Pip"))
    2
    """
    return len(my_filter(lambda s: s.lower() == s[::-1].lower(), L))
```

```
def flip_two(s):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
    # Recursive solution:
    if s is Link.empty or s.rest is Link.empty:
        return
    s.first, s.rest.first = s.rest.first, s.first
    flip_two(s.rest.rest)
```

```
def leaves(t):
    """Returns a list of all the labe

    >>> leaves(Tree(1))
    [1]
    >>> leaves(Tree(1, [Tree(2, [Tree
    [3, 4]
    """
    if t.is_leaf():
        return [t.label]
    all_leaves = []
    for b in t.branches:
        all_leaves += leaves(b)
    return all_leaves
```

```
def find_paths(t, entry):
paths = []
if t.label == entry:
 paths.append([t.label])
for b in t.branches:
 for path in find_paths(b,
entry):
   paths.append([t.label] +
path)
return paths
```