STSCI 4060

Lecture File 4

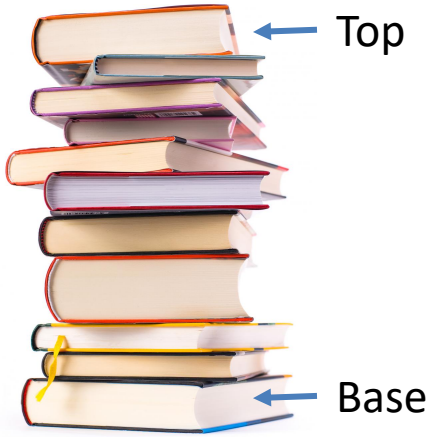
Xiaolong Yang
(xy44@cornell.edu)

Some Important Python Data Structures

- Stack
- Queue
- Linked List
- Tree

Stack

- A stack is an ordered collection of elements where the addition of new elements and the removal of existing elements always takes place at the top end. The end opposite to the top is known as the "base."
- Stack's ordering principle is last-in first-out (LIFO), providing an ordering based on length of time in the collection: newer items are near the top and older items are near the base.



- Stacks can be used to reverse the order of items. The order of insertion is the reverse of the order of removal.
- Examples:
  - Find the employee who was most recently hired.
  - Get a food item on a vertically stored food plates.
  - A list of the URLs of the web pages visited.
  - Most recent open parenthesis in a computer program.

Implement a Python Stack

- Use the Python class mechanism to define a Python stack.
- Use a list to hold stack elements.
- Define some operations to manipulate the stack.
  - push(): To add a new element to the top of the stack.
  - pop(): To remove and return the top element of the stack.
  - peek(): To return the top element of the stack but not remove it.
  - size(): To return the number of elements on the stack.
  - showStack(): To display the whole stack.
  - isEmpty(): To find out if the stack is empty or not.

The Implementation of the Stack Class

stack.py

```
# This stack class is implemented with a list using
# the end of the list as the top of the stack.

class Stack:
    def __init__(self):
        self.stk = []

    # To find out if the stack is empty or not
    def isEmpty(self):
        return self.stk == []

    # To add a new element to the top of the stack
    def push(self, element):
        self.stk.append(element)

    # To remove and return the top element of the stack
    def pop(self):
        return self.stk.pop()

    # To return the number of elements on the stack
    def size(self):
        return len(self.stk)

    # To return the top element of the stack but not remove it
    def peek(self):
        return self.stk[-1]

    # To display the whole stack
    def showStack(self):
        return self.stk
```

Use the Stack Class Just Created

If the class is not used in the same program, you need to run the Python file, stack.py, containing the Stack class definition so that you can use it to instantiate a Stack object. Or, use the import statement to import the class.

```
>>> s=Stack()
>>> s.isEmpty()
True
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.push(50)
>>> s.size()
5
>>> s.peek()
50
>>> s.showStack()
[10, 20, 30, 40, 50]
>>> s.pop()
50
>>> s.showStack()
[10, 20, 30, 40]
```

```
>>> import stack
>>> k=stack.Stack()
>>> k.isEmpty()
True
>>> k.push(5)
>>> k.push(15)
>>> k.push(25)
>>> k.push(35)
>>> k.push(45)
>>> k.size()
5
>>> k.peek()
45
>>> k.showStack()
[5, 15, 25, 35, 45]
>>> k.pop()
45
>>> k.showStack()
[5, 15, 25, 35]
```

Queue

- A queue is an ordered collection of elements where the addition of new elements happens at one end ("tail") and the removal of existing elements occurs at the other end ("head"). As an element enters the queue it starts at the tail and makes its way toward the head, waiting until that time when it is the next element to be removed.
- A queue organized first-in first-out (FIFO) or "first-come first-served." The element that has been in the queue the longest is at the head.

- A queue can represent many real-world activities:
  - A check-out line in a store.
  - A printing queue in a networked computer room.
  - A scheduling method based on a first-come first-served rule.

Implement a Python Queue

- Use the Python class mechanism to define a Python queue.
- Use a list to hold queue elements.
- Define some operations to manipulate the queue.
  - enqueue(): To add a new element to the tail of the queue.
  - dequeue(): To remove and return the head element of the queue.
  - peekHead(): To return the head element of the queue but not remove it.
  - peekTail(): To return the tail element of the queue but not remove it.
  - size(): To return the number of elements in the queue.
  - showQueue(): To display the whole queue.
  - isEmpty(): To find out if the queue is empty or not.

The Implementation of the Queue Class

queue.py

```
# This queue class is implemented with a list using
# the end of the list as the head of the queue and the
# begining of the list as the tail of the queue.

class Queue:
    def __init__(self):
        self.qu = []

    # To find out if the queue is empty or not
    def isEmpty(self):
        return self.qu == []

    # To add a new element to the tail of the queue
    def enqueue(self, element):
        self.qu.insert(0,element)

    # To remove and return the head element of the queue
    def dequeue(self):
        return self.qu.pop()

    # To return the number of elements on the queue
    def size(self):
        return len(self.qu)

    # To return the head element of the queue but not remove it
    def peekHead(self):
        return self.qu[-1]

        # To return the tail element of the queue but not remove it
    def peekTail(self):
        return self.qu[0]

    # To display the whole stack
    def showQueue(self):
        return self.qu
```

Use the Queue Class Just Created

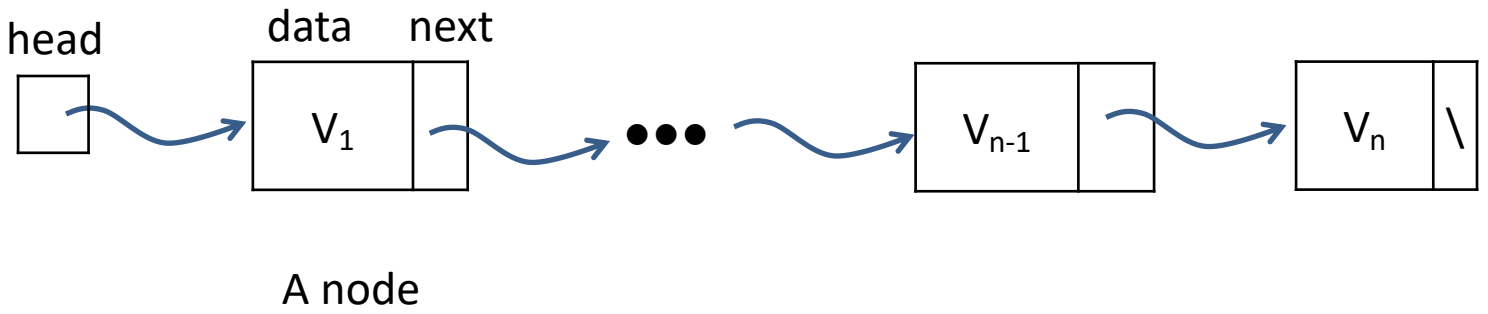
If the class is not used in the same program, you need to run the Python file, queue.py, containing the Queue class definition in order to use it to instantiate a queue object. Or, use the import statement to import the class.

```
>>> q=Queue()
>>> q.isEmpty()
True
>>> q.enqueue(10)
>>> q.enqueue(20)
>>> q.enqueue(30)
>>> q.enqueue(40)
>>> q.enqueue(50)
>>> q.size()
5
>>> q.peekHead()
10
>>> q.peekTail()
50
>>> q.showQueue()
[50, 40, 30, 20, 10]
>>> q.dequeue()
10
>>> q.showQueue()
[50, 40, 30, 20]
```

```
>>> import queue
>>> qq=queue.Queue()
>>> qq.isEmpty()
True
>>> qq.enqueue(100)
>>> qq.enqueue(200)
>>> qq.enqueue(300)
>>> qq.enqueue(400)
>>> qq.enqueue(500)
>>> qq.size()
5
>>> qq.peekHead()
100
>>> qq.peekTail()
500
>>> qq.showQueue()
[500, 400, 300, 200, 100]
>>> qq.dequeue()
100
>>> qq.showQueue()
[500, 400, 300, 200]
```

Linked List

- A linked list is constructed as an unordered collection of elements, where each element holds a relative position with respect to the others.
- It is important to maintain the locations of the list elements, especially that of the next element in the list, then the relative position of each item can be expressed by simply following the link from one item to the next.
- We use nodes as the basic building blocks for the linked list. Each node object must hold two pieces of information: the list item itself (**data field**) and a **reference** to the next node.



Implementing a Linked List

We need to create two classes:

1. Node class.

2. LinkedList class.

Implementing the Node Class

To construct a node, an initial data value for the node must be provided. The Node class includes some methods to access and modify the data that the node contains and the reference to the next node, called ***next***.

- getData() to return the data of the current node.
- getNext() to return the reference to the next node.
- setData() to assign a new data value to the current node.
- setNext() to assign a new reference pointing to the next node.

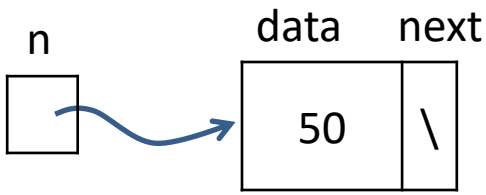
Implementing the Node Class

node.py

```
# Implement the Node class for the linked list
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    # return the data of a node
    def getData(self):
        return self.data

    # return the reference to next node
    def getNext(self):
        return self.next

    # assign a new data value to the current node
    def setData(self,newdata):
        self.data = newdata

    # assign a new reference pointing to next node
    def setNext(self,newnext):
        self.next = newnext
```

Using the Node Class

```
 RESTART: C:/Teaching/STSCI4060Spring2018/1Lec/File4/node.py
>>> n=Node(50)
>>> n.getData()
50
>>> n.getNext()
>>> n.setData(100)
>>> n.getData()
100
>>>
```



Implementing the LinkedList Class

A linked is built from a collection of nodes, each linked to the next node by an explicit reference. The LinkedList class must maintain a reference to the first node. The following operations may be included.

- isEmpty(): tests if the list is empty.
- add(e): adds a new element e to the list as the first element.
- remove(e): removes an element e from the list.
- search(e): to see if element e is in the list.
- size(): returns the number of elements in the list.
- insert(p, e): inserts a new element e at position p.
- append(e): adds a new element e to the end of the list.
- pop(): removes and returns the last element of the list.
- pop(p): removes and returns the element at position p.
- index(e): returns the position of element e in the list.
- printList(): print the values of the list.

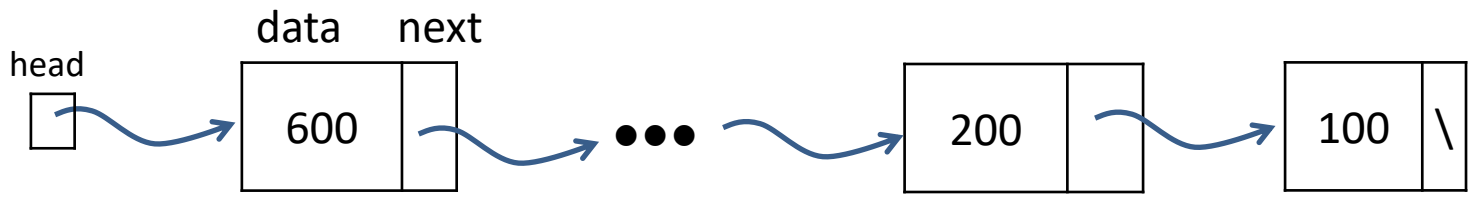
Using the LinkedList Class

```
from node import Node
class LinkedList:

    def __init__(self):
        self.head = None

    def isEmpty(self):
        return self.head == None

    def add(self,element):
        temp = Node(element)
        temp.setNext(self.head)
        self.head = temp

    def size(self):
        current = self.head
        count = 0
        while current != None:
            count = count + 1
            current = current.getNext()

        return count

    def search(self,element):
        current = self.head
        found = False
        while current != None and not found:
            if current.getData() == element:
                found = True
            else:
                current = current.getNext()

        return found

    def remove(self,element):
        current = self.head
        previous = None
        found = False
        while not found:
            if current.getData() == element:
                found = True
            else:
                previous = current
                current = current.getNext()

        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
```

Using the LinkedList Class

```
RESTART: C:/Python27/STSCI4060Spring2018/linkedlist.py
>>> LL=LinkedList()
>>> LL.isEmpty()
True
>>> LL.add(100)
>>> LL.add(200)
>>> LL.add(300)
>>> LL.add(400)
>>> LL.add(500)
>>> LL.add(600)
>>> LL.size()
6
>>> LL.search(300)
True
>>> LL.remove(400)
>>> LL.size()
5
```

The LinkedList Created

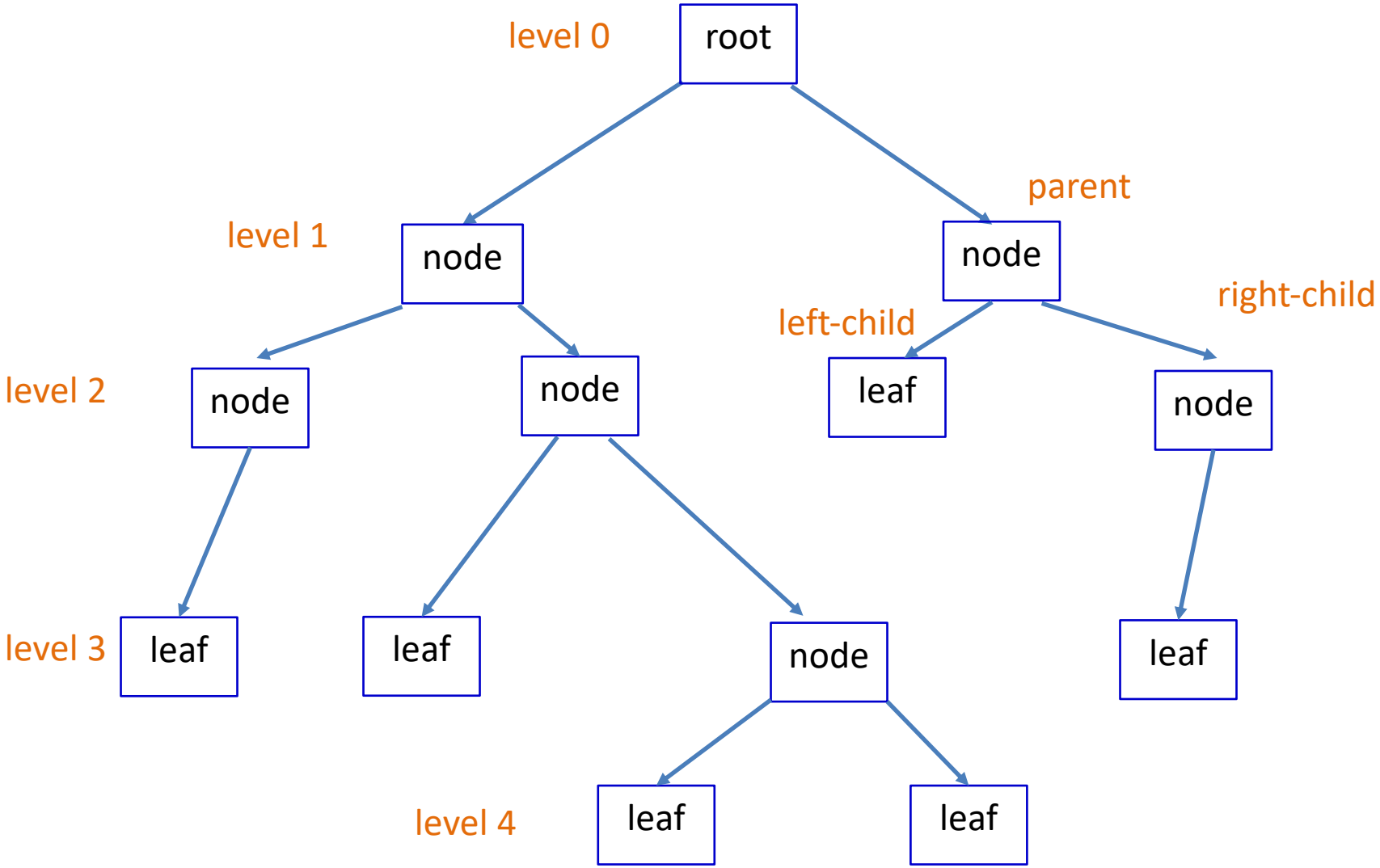


Tree

A tree is a nonlinear data structure, having a root, branches, and leaves, normally presented as an upside-down "tree" (compared to its botanical cousins), consisted of a set of nodes and edges, with its root at the top and its leaves on the bottom. Normally it has different levels.

A tree can be viewed as a recursive structure. It is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree.

A Tree Diagram



Some Terms Related to Trees

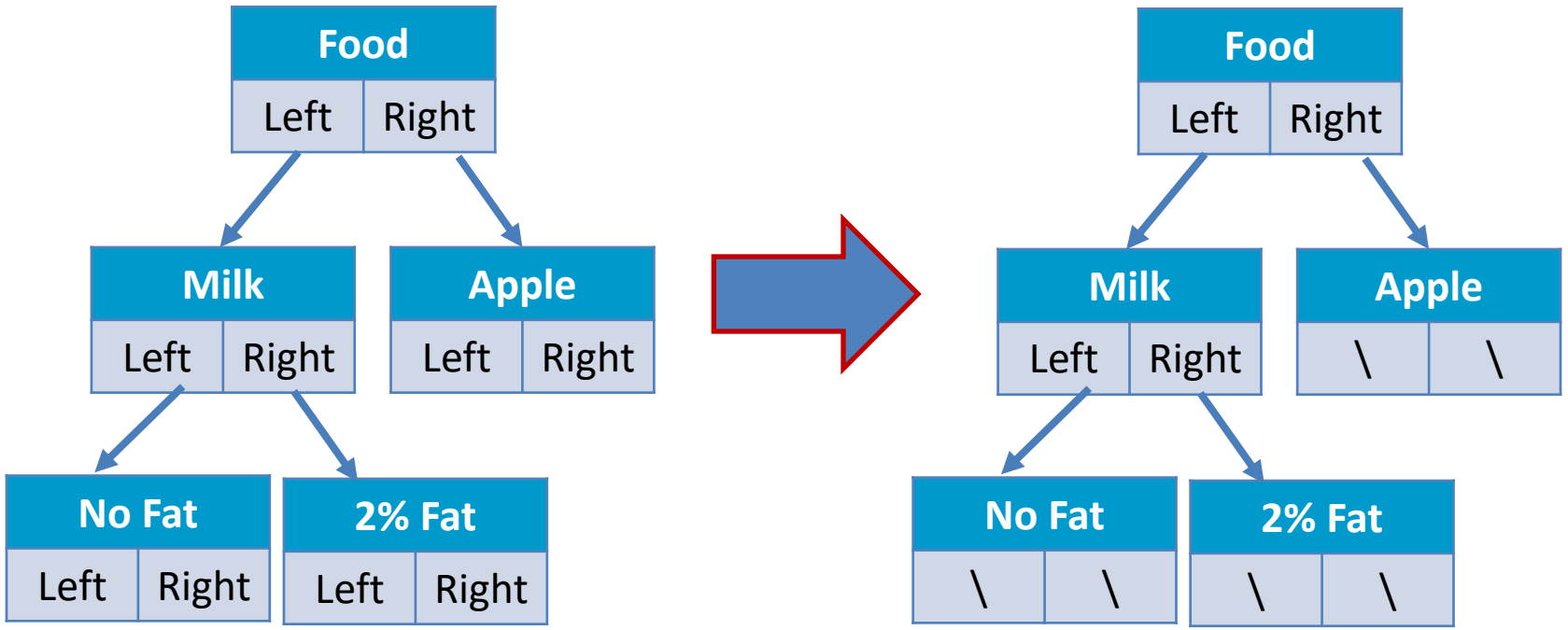
- **Root**: the topmost node of the tree.
- **Node**: building blocks of a tree.
- **Edge**: a connection that connects two nodes. Every node (except the root) is connected by exactly one incoming edge from another node. Each node may have more than one outgoing edge.
- **Path**: an ordered list of nodes that are connected by edge(s).
- **Child**: a node that has an incoming edge from another node.
- **Parent**: a node that has an outgoing edge(s).
- **Sibling**: nodes in the tree that are children of the same parent.
- **Leaf (node)**: a node that has no child(ren).
- **Subtree**: a set of nodes and edges comprised of a parent and all the descendants of that parent.
- **Level:** the number of edges on the path from the root node to the node of interest.
- **Height**: the maximum level of any node in the tree.

Tree Properties

- Every node, except the root node, is connected by an edge from exactly one parent node.
- All of the children of one node are independent of the children of another node.
- There is a unique path that traverses from the root to each node.
- Trees are hierarchical, structured in layers with the more general things near the top and the more specific things near the bottom.

A Simple Binary Tree

(A tree whose nodes have a maximum of two children)



Implementing a Tree as a List of Lists

- The root node is the first element of the list.
- The second element of the list is a list representing the left subtree.
- The third element of the list is another list representing the right subtree.
- …
- A subtree that has a root value and two empty lists is a leaf node.

```
>>> aTree =['Food', ['Milk', ['No Fat', [], []], ['2% Fat',
[], []]], ['Apple', [], []]]
>>> print aTree
['Food', ['Milk', ['No Fat', [], []], ['2% Fat', [], []]], [
'Apple', [], []]]
>>> print 'left subtree = ', aTree[1]
left subtree =  ['Milk', ['No Fat', [], []], ['2% Fat', [],
[]]]
>>> print 'root = ', aTree[0]
root =  Food
>>> print 'right subtree = ', aTree[2]
right subtree =  ['Apple', [], []]
```

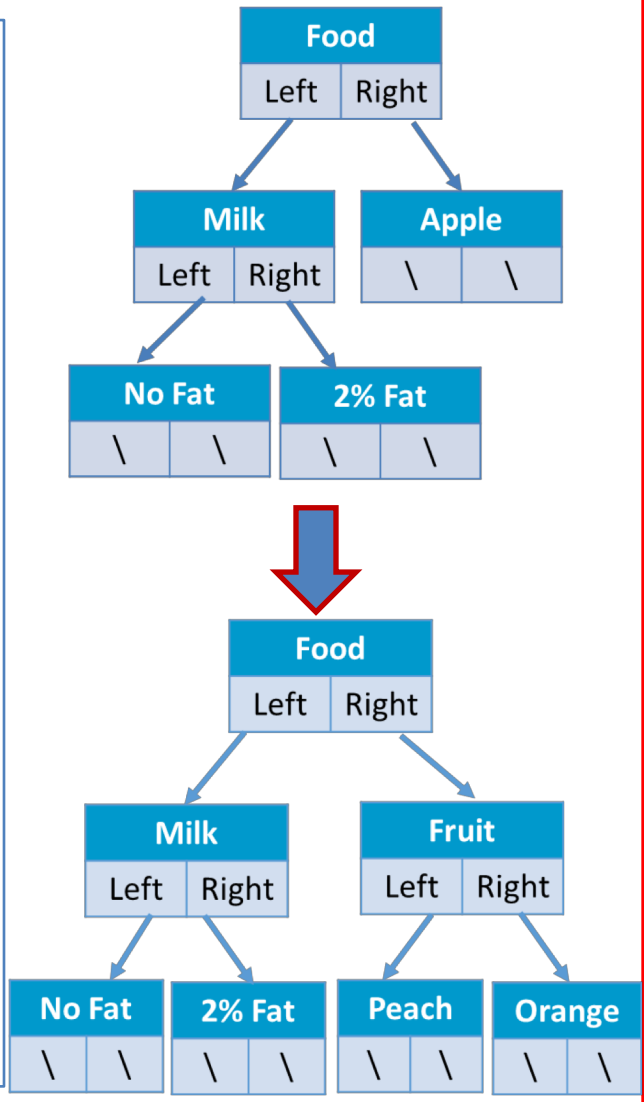
Implementing a Tree as a List of Lists

```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
```

```
def getRootVal(root):
    return root[0]

def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]
```

Create, Change and Traverse a Tree

```
aTree=BinaryTree('Food')
insertLeft(aTree, 'Milk')
insertLeft(getLeftChild(aTree), 'No Fat')
insertRight(getLeftChild(aTree), '2% Fat')
insertRight(aTree, 'Apple')
print 'The tree created is ', aTree
setRootVal(getRightChild(aTree), 'Fruit')
print '\nThe tree is ', aTree
insertLeft(getRightChild(aTree), 'Peach')
insertRight(getRightChild(aTree), 'Orange')
print '\nThe tree after insertion is ', aTree
Print '\n', getRootVal(getRightChild(getLeftChild(aTree)))
```



Left: left subtree
Right: right subtree

Create, Change and Traverse a Tree

The output:

```
The tree created is  ['Food', ['Milk', ['No
Fat', [], []], ['2% Fat', [], []]], ['Apple'
, [], []]]

The tree updated is  ['Food', ['Milk', ['No
Fat', [], []], ['2% Fat', [], []]], ['Fruit'
, [], []]]

The tree after insertion is  ['Food', ['Milk
', ['No Fat', [], []], ['2% Fat', [], []]],
['Fruit', ['Peach', [], []], ['Orange', [],
[]]]]

2% Fat
```

Implementing a Binary Tree Recursively

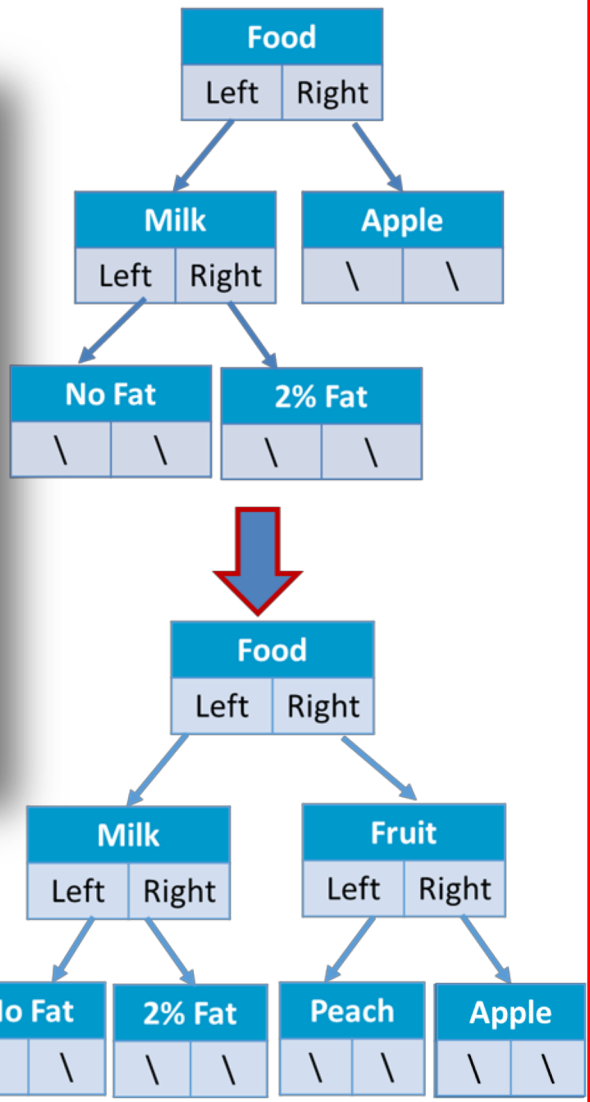
```
# Build a simple binary tree
class BinaryTree:
    def __init__(self,val):
        self.key = val
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t


    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self,newValue):
        self.key = newValue

    def getRootVal(self):
        return self.key
```

Create, Change and Traverse a Binary Tree Using the BinaryTree Class

```
r = BinaryTree('Food')
r.insertLeft('Milk')
r.insertRight('Apple')
r.getLeftChild().insertLeft('No Fat')
r.getLeftChild().insertRight('2% Fat')

print r.getRootVal()
print r.getLeftChild().getRootVal()
print r.getRightChild().getRootVal()
print r.getLeftChild().getRightChild().getRootVal()

r.getRightChild().setRootVal('Fruit')
r.getRightChild().insertLeft('Peach')
r.getRightChild().insertRight('Apple')
print r.getRightChild().getRootVal()
print r.getRightChild().getRightChild().getRootVal()
```

Output

```
Food
Milk
Apple
2% Fat
Fruit
Apple
```



Left: leftChild, Right: rightChild