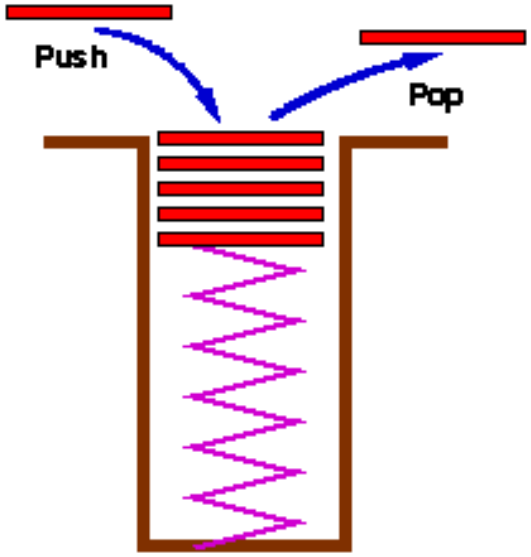
Part 5

COMMON CONTAINER CLASSES

Stack, Queue, Tree, Graph

Stack Abstract Data Type
Description

- Sequence type (container) in which elements are pushed and popped out from the top end
- AKA **LIFO** – Last In First Out



Stack Test 1

"Code without tests is broken by design." - Jacob

```
def test1():
    s = Stack()
    s.push(1)
    s.push(2)
    s.push(2)
    s.push(3)
    assert s.peek() == 3
    assert s.pop() == 3
    assert s.pop() == 2
    assert s.pop() == 2
    assert s.pop() == 1
    assert s.is_empty()
    print "test1 PASSED"
```

Stack Test 2

```
def test2():
    expression = "a+(b*(c+d)+x*(y-a)+z)-n"
    s = Stack()
    # Check if left/right parens are
    # legally balanced
    for char in expression:
        if char == '(':
            s.push(char)
        if char == ')':
            assert not s.is_empty(), "Redundant right parens"
            s.pop()

    assert s.is_empty(), "Unbalanced expression – over left parens"
    print "test2 PASSED"
```

Stack Abstract Data Type
Interface

- `s = Stack()` — **Constructor**
  - Create a new empty stack
- `s.push(item)` — **Mutator**
  - Add an item to the top of the stack
- `s.pop()` — **Mutator**
  - Pop an item from the top of the stack
- `s.peek()` — **Accessor**
  - Return the item to the top of the stack (don't pop it!)
  - Return `None` if stack is empty (this is not a good idea, why? Issue error?)
- `s.size()` — **Accessor**
  - Return the number of items in the stack
- `s.is_empty()` — **Accessor**
  - Return `True` if stack is empty, `False` if stack is non-empty

Stack Test 2: Stack Frames

```
s = Stack()
expression = "a+(b*(c+d)+x*(y-a)+z)-n"
             0  1  2   3   4   5  6

Frame 0: empty stack
Frame 1: (
Frame 2: (, (
Frame 3: (
Frame 4: (, (
Frame 5: (
Frame 6: empty stack
```

Stack Implementation

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

  def push(self, item) :
    self.items.append(item)

  def pop(self) :
    return self.items.pop()

  def peek(self):
    return self.items[-1]

  def is_empty(self) :
    return (self.items == [])
```

http://www.greenteapress.com/thinkpython/thinkCSpy/html/chap18.html

Queue Abstract Data Type

- Sequence type (container) in which elements are pushed on one side (back) and popped out from the other side (front)
- AKA **FIFO** – **F**irst **I**n **F**irst **O**ut

- Operations:
  - enqueue push a new item to the back (mutator)
  - dequeue pop an element from the front (mutator)
  - first Get the first element (accessor)
  - size Get the size of the Queue (accessor)
  - Is_empty Check if the Queue is empty

Queue Test 1 (Client Code)

"Code without tests is broken by design." **-** Jacob Kaplan-Moss

```
def test1():
    q = Queue()
    q.enqueue(10)
    q.enqueue(20)
    q.enqueue(20)
    q.enqueue(30)
    assert q.first() == 10
    assert q.dequeue() == 10
    assert q.dequeue() == 20
    assert q.dequeue() == 20
    assert q.dequeue() == 30
    assert q.is_empty()
    print "test1 PASSED"
```

Queue Abstract Data Type (Interface)

- `q = Queue()` — **Constructor**
  - Create a new empty Queue
- `q.enqueue(item)` — **Mutator**
  - Add an item to the **back** of the Queue
- `s.dequeue()` — **Mutator**
  - Remove an item from the **front** (and return it)
- `s.first()` — **Accessor**
  - Return the item which is first in line (don't pop it!)
  - Raise an exception if Queue is empty
- `s.size()` — **Accessor**
  - Return the number of items in the Queue
- `s.is_empty()` — **Accessor**
  - Return `True` if Queue is empty, `False` if Queue is non-empty

Queue Implementation 1 (as List)

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

    def enqueue(self, elem):
        self.items.append(elem)

    def dequeue(self):   # High cost! Complexity is O(n)
        return self.items.pop(0)

    def first(self):
        return self.items[0]

    def is_empty(self):
        return self.items == []

    def __len__(self):
        return len(self.items)

    def __str__(self):
        return str(self.items)
```

Queue Implementation 2 (as List)

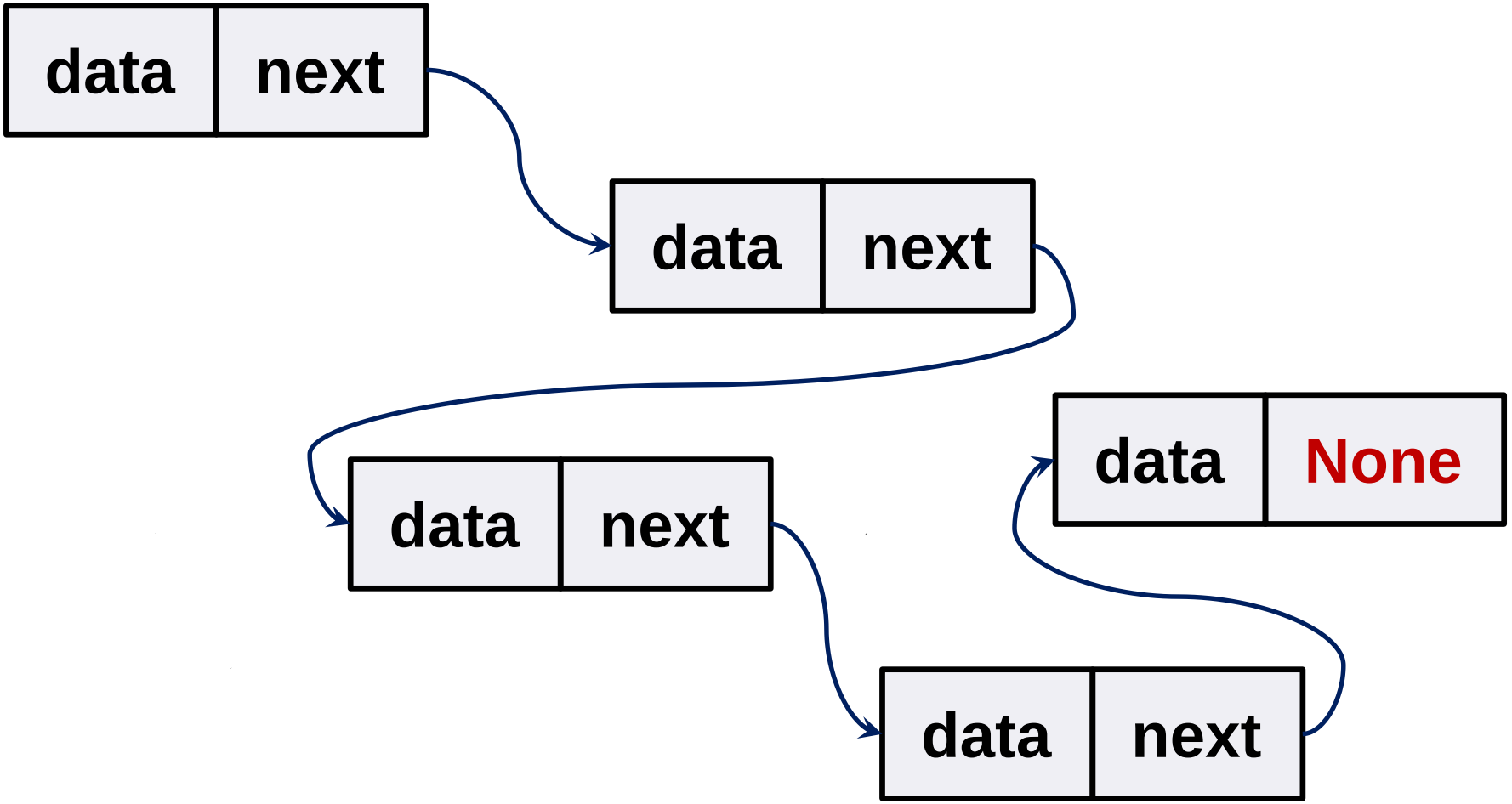
```
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.insert(0, item)  # High cost! Complexity is O(n)

    def dequeue(self):
        return self.items.pop()

    def first(self):
        return self.items[-1]

    def is_empty(self):
        return self.items == []

    def __len__(self):
        return len(self.items)

    def __str__(self):
        return str(self.items)
```

Queue Implementation (as Linked List)

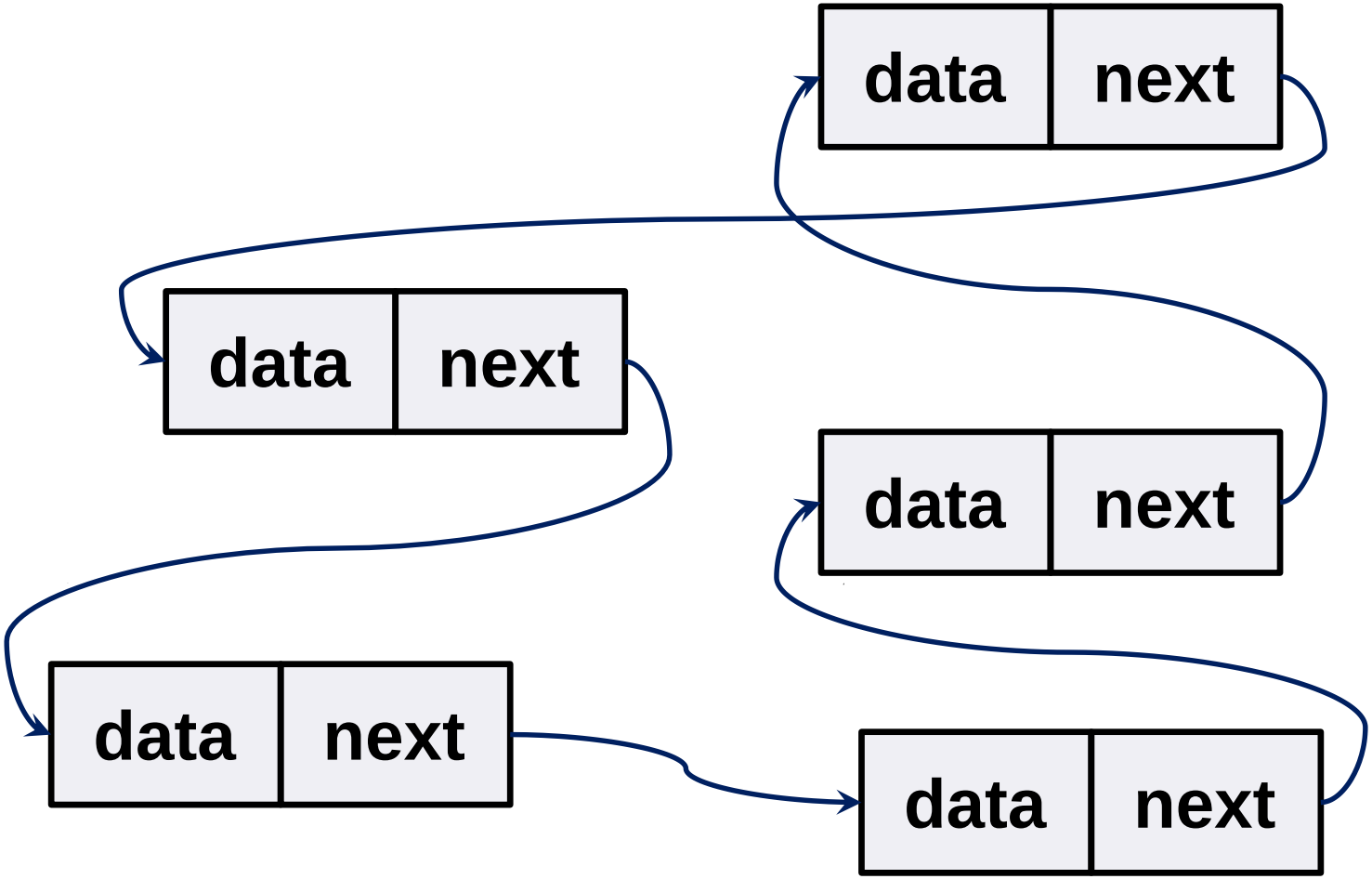
- The linked list implementation is based on a Node class which is the basic element of our linked list
- Each Node object consists of a **data** member and a **next** member
- The next member is a pointer to the next Node in the linked list
- If next == None, then no next Node exist (signals end of list)

```
class Node:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next
```

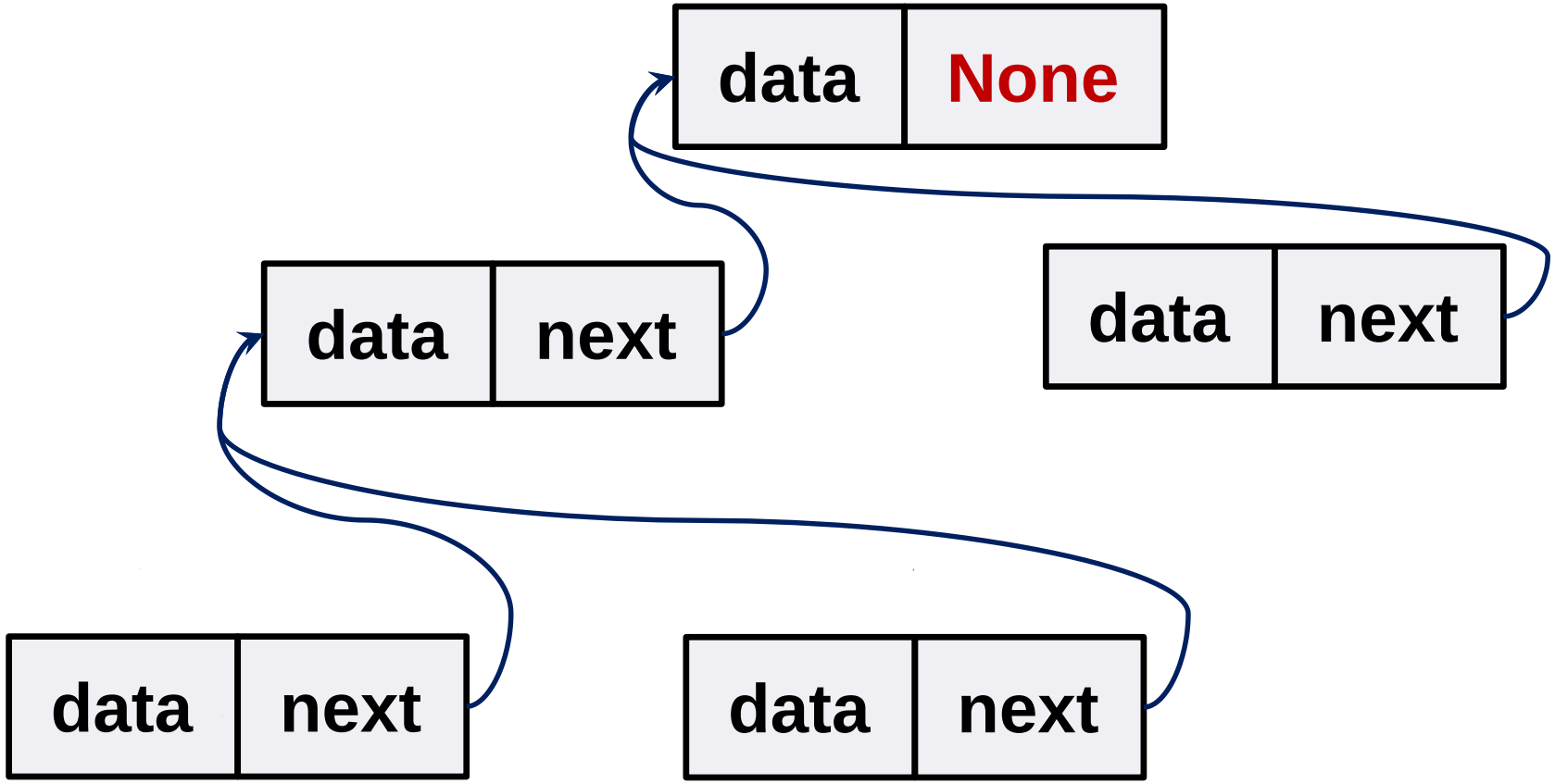
Node objects (Sequence Structure)



Node objects (Circular Structure)



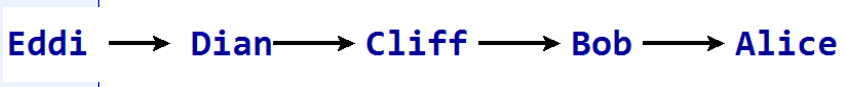
Node objects (Tree Structure)



Node Class: Test 1

```
class Node:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next

    def __str__(self):
        return 'Node(' + str(self.data) + ')'

def test1():
    a = Node('Alice')       # a.next is None
    b = Node('Bob', a)      # b.next = a
    c = Node('Cliff', b)    # c.next = b
    d = Node('Dian', c)     # d.next = c
    e = Node('Eddi', d)     # e.next = d

    assert e.next is d
    assert e.next.next is c
    assert e.next.next.next is b
    assert e.next.next.next.next is a

    assert e.data == 'Eddi'
    assert d.data == 'Dian'
    assert a.next is None
    print "test1 PASSED"
```

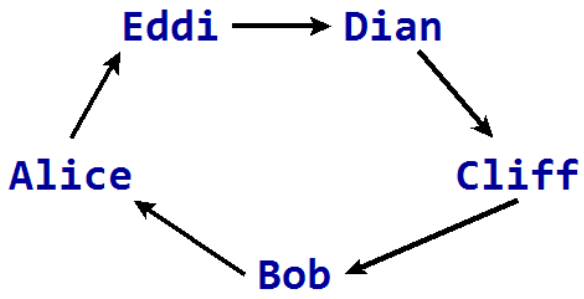
Alice, Bob, Cliff, Dian, and Eddi are secret NSA agents Their "connection graph" is minimal (single connection person) Eddi has a reference to Dian, Dian has a reference to Cliff, Cliff to Bob, and Bob to Alice



Node Class: Test 2 (circular struct)

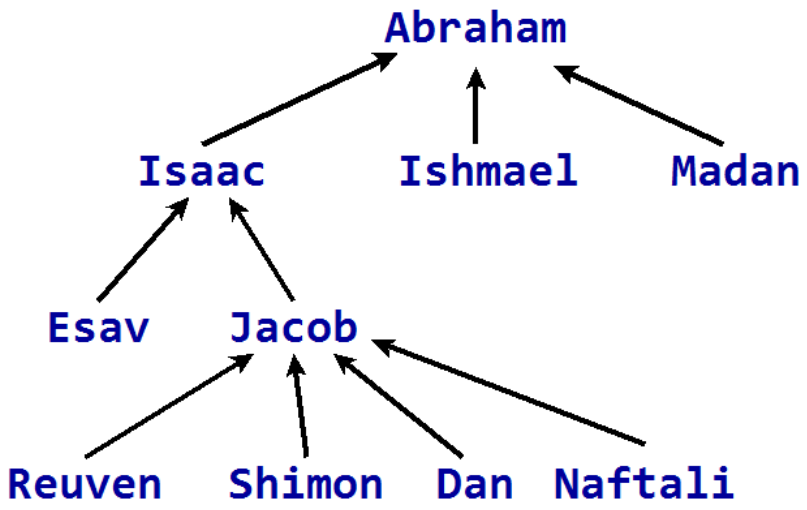
```
def test3():     # Circular list!
    a = Node('Alice')
    b = Node('Bob', a)
    c = Node('Cliff', b)
    d = Node('Dian', c)
    e = Node('Eddi', d)
    a.next = e

    node = e
    while True:
        print node,
        if node is a:
            break
        node = node.next
```

Alice, Bob, Cliff, Dian, and Eddi are
Are now connected through a
Circular structure
Alice finally has a connection
person



Node Class: Test 3 (Tree struct)

```
def test4():  # Tree Structure !!!
    abraham = Node('Abraham')
    isaac   = Node('Isaac', abraham)
    ishmael = Node('Ishmael', abraham)
    madan   = Node('Madan', abraham)
    esav    = Node('Esav', isaac)
    jacob   = Node('Jacob', isaac)
    reuven  = Node('Reuven', jacob)
    shimon  = Node('Shimon', jacob)
    dan     = Node('Dan', jacob)
    naftali = Node('Naftali', jacob)

    nodes = [abraham, isaac, ishmael, \
             madan, esav, jacob, reuven, \
             shimon, dan, naftali]

    for node in nodes:
        print node, level(node)

    for node in nodes:
        if level(node) == 3:
            print_backward(node)
            print
```



Queue Implementation (as Linked List)

```
class Queue:
    def __init__(self):
        "Create an empty queue"
        self.head = None
        self.tail = None
        self.size = 0              # number of queue elements

    def __len__(self):
        "Return the number of elements in the queue"
        return self.size

    def is_empty(self):
        "Return True if the queue is empty"
        return self.size == 0

    def first(self):
        """Return (but do not remove!) the data at the front of the queue.
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Exception('Queue is empty')
        return self.head.data  # front aligned with head of list
```

Queue Implementation (as Linked List)

```
def dequeue(self):
        """Remove and return the first data element of the queue (i.e., FIFO).
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Exception('Queue is empty')
        answer = self.head.data
        self.head = self.head.next  # next in line is now the head
        self.size -= 1
        if self.is_empty():         # special case as queue is empty
            self.tail = None        # removed head had been the tail
        return answer

    def enqueue(self, e):
        """Add an element to the back of queue."""
        newest = Node(e, None)      # node will be new tail node
        if self.is_empty():
            self.head = newest        # special case: previously empty
        else:
            self.tail.next = newest   # this is the old tail ...
        self.tail = newest            # update reference to tail node
        self.size += 1
```

Which Implementation is Best ???

```
from timeit import timeit

def que_bench():
    q = Queue()
    for i in range(1000):
        for j in range(500):
            q.enqueue(j)
        for j in range(500):
            q.dequeue()

if __name__ == '__main__':
    from queue_as_list_1 import Queue
    print "queue_as_list_1:", timeit(que_bench, number=5)

    from queue_as_list_2 import Queue
    print "queue_as_list_2:", timeit(que_bench, number=5)

    from queue_as_list_3 import Queue
    print "queue_as_list_3:", timeit(que_bench, number=5)

    from  queue_as_linked_list import Queue
    print "queue_as_linked_list:", timeit(que_bench, number=5)
```

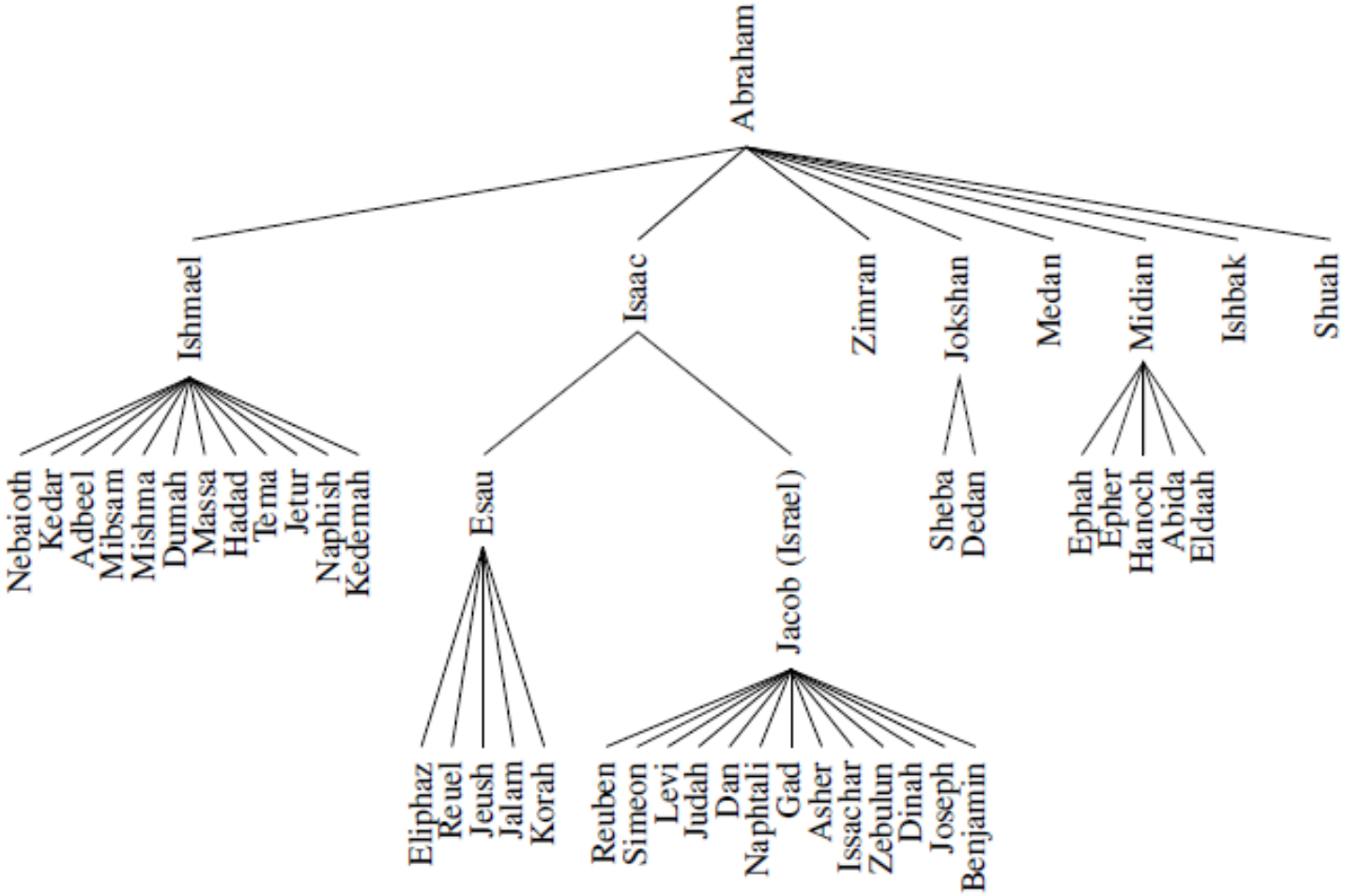
Did Linked List Performed Well ???

```
from timeit import timeit

def que_bench():
    q = Queue()
    for i in range(1000):
        for j in range(500):
            q.enqueue(j)
        for j in range(500):
            q.dequeue()

if __name__ == '__main__':
    from queue_as_list_1 import Queue
    print "queue_as_list_1:", timeit(que_bench, number=5)

    from queue_as_list_2 import Queue
    print "queue_as_list_2:", timeit(que_bench, number=5)

    from queue_as_list_3 import Queue
    print "queue_as_list_3:", timeit(que_bench, number=5)

    from  queue_as_linked_list import Queue
    print "queue_as_linked_list:", timeit(que_bench, number=5)
```

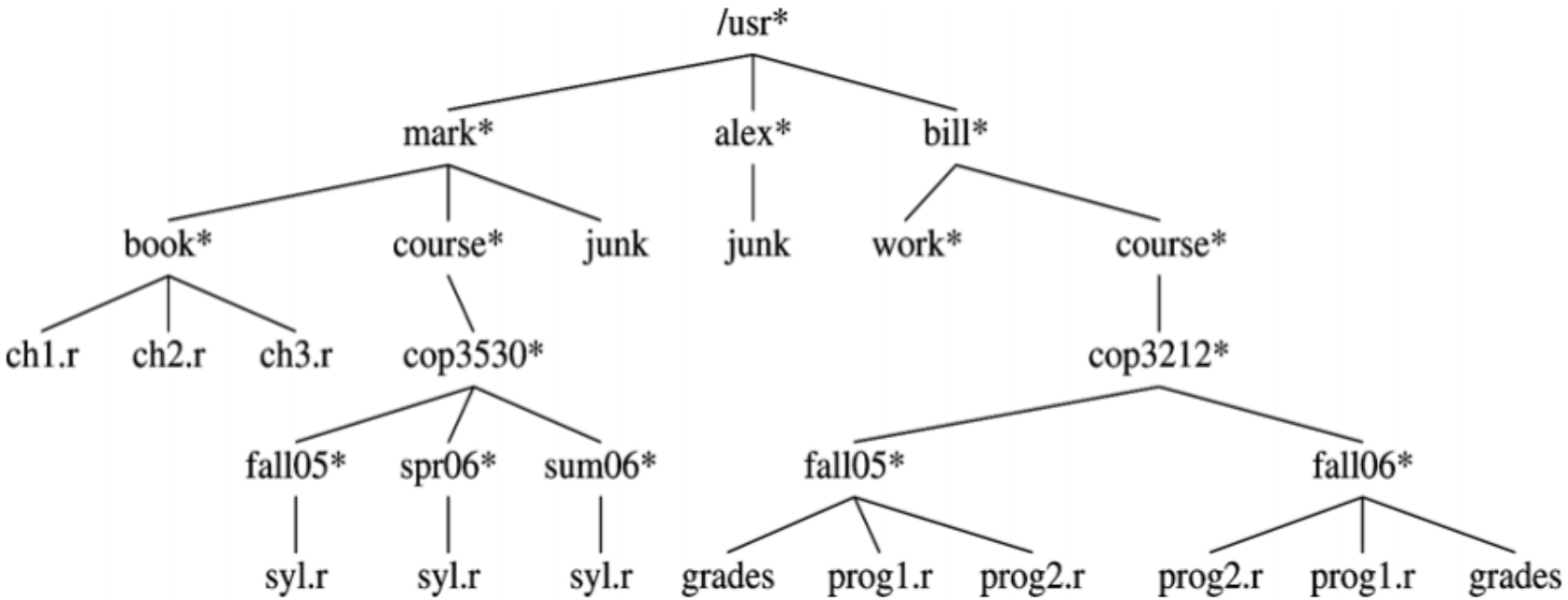
Did Linked List Performed Well ???

- Good article on why we should be careful with linked lists http://kjellkod.wordpress.com/2012/02/25/why-you-should-never-ever-ever-use-linked-list-in-your-code-again
- Bjarne Stroustrup Video: Why you should avoid Linked Lists https://www.youtube.com/watch?v=YQs6IC-vgmo

Trees

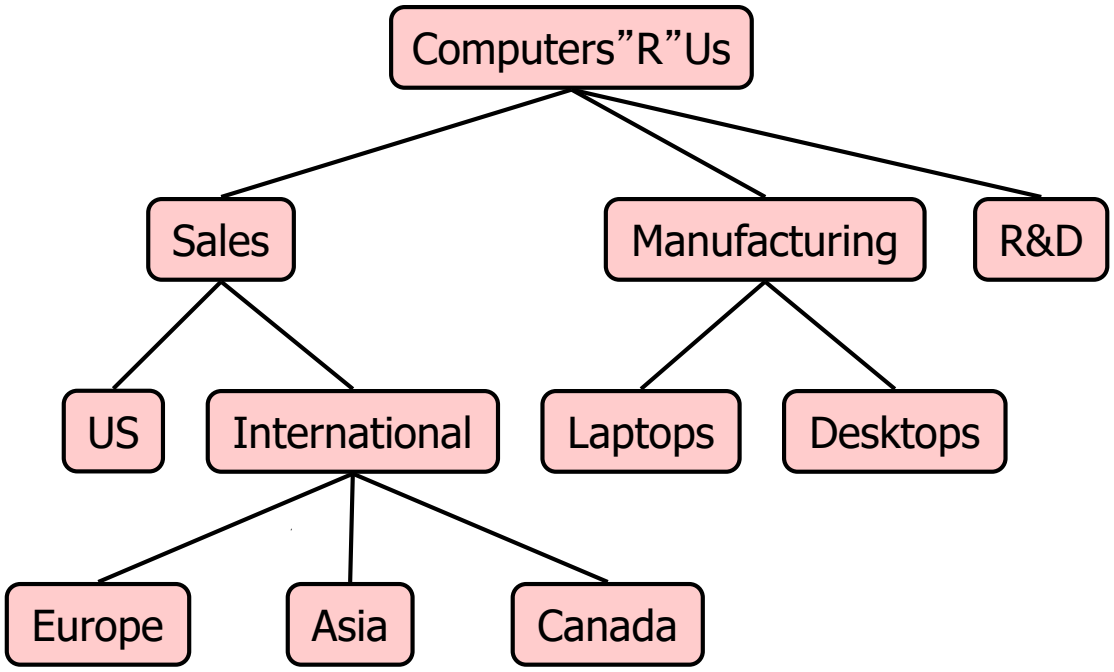

Abraham
Ishmael
Isaac
Zimran
Jokshan
Medan
Midian
Ishbak
Shuah
Nebaioth
Kedar
Adbeel
Mibsam
Mishma
Dumah
Massa
Hadad
Tema
Jetur
Naphish
Kedemah
Esau
Jacob (Israel)
Sheba
Dedan
Ephah
Epher
Hanoch
Abida
Eldaah
Eliphaz
Reuel
Jeush
Jalam
Korah
Reuben
Simeon
Levi
Judah
Dan
Naphtali
Gad
Asher
Issachar
Zebulun
Dinah
Joseph
Benjamin

Example 2: Unix File System

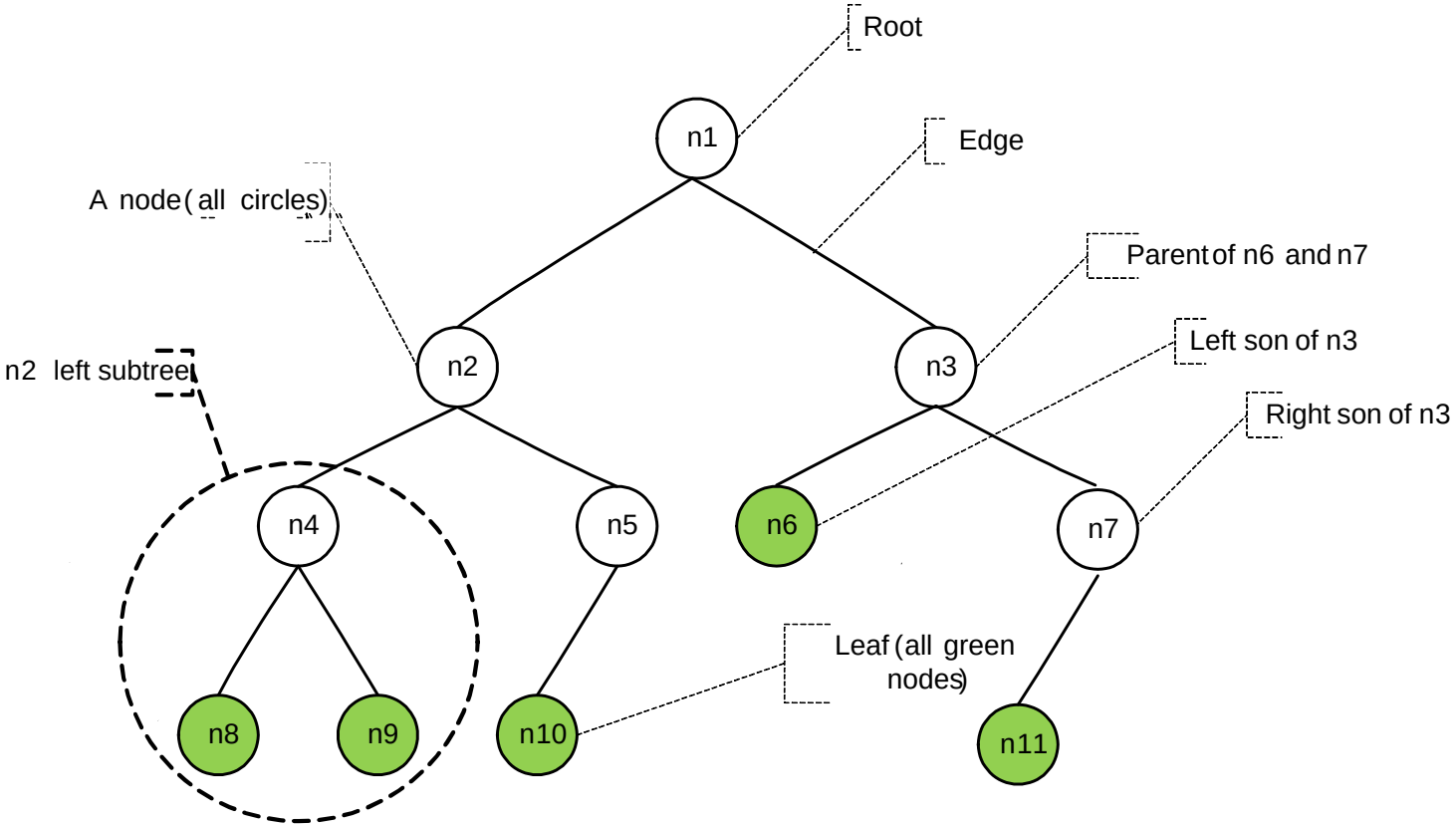


What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Gaming data structures



What is a Tree (Daniel Geva)

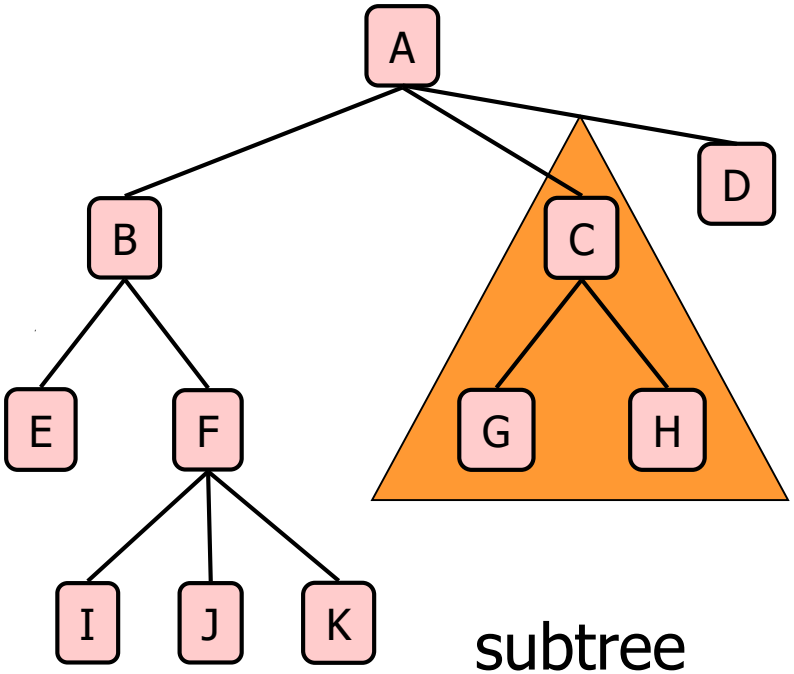


Node height – number of edges on the longest path to a leaf
Tree height – height of the root
Balanced Tree – All non-leaf have two sons

Tree Terminology

- **Root**
  node without parent (A)
- **Internal node**
  node with at least one child (A, B, C, F)
- **Leaf (External node)**
  node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node:
  parent, grandparent, grand-grandparent, etc.
- **Depth of a node:**
  number of ancestors
- **Height of a node:**
  1 + Max height of children
  (leaf height = 0)
- **Height of a tree**
  maximum depth of any node (3)
- **Descendant of a node**
  child, grandchild, grand-grandchild, etc.

- Subtree: tree consisting of a node and its descendants



Tree ADT

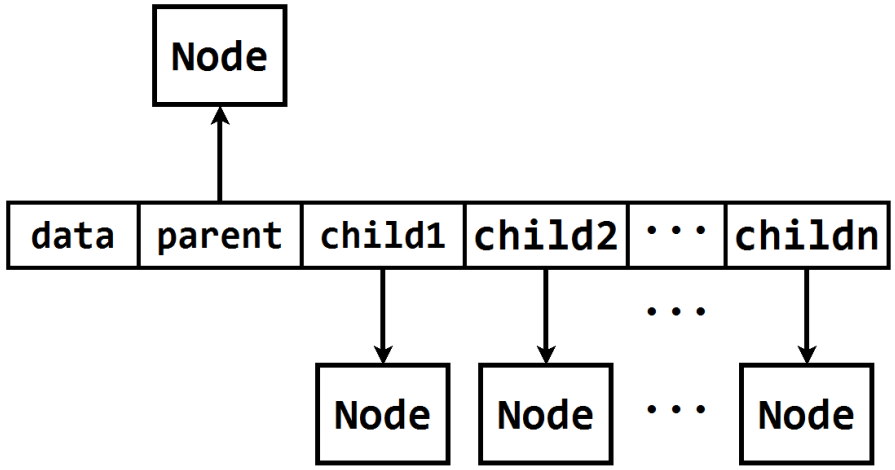
- We use positions to abstract nodes, left key is return type:
- **Generic methods**:
  - Integer len()
  - Boolean is_empty()
  - Iterator nodes()
  - Iterator iter()
- **Accessor methods**:
  - Node root()
  - Node parent(node)
  - Nodes children(node)
  - Integer num_children(node)

- Query methods:
  - Boolean is_leaf(node)
  - Boolean is_root(node)
- Update method:
  - element replace (node, node2)
- Additional update methods may be defined by data structures implementing the Tree ADT

Note: Nodes are the tree elements

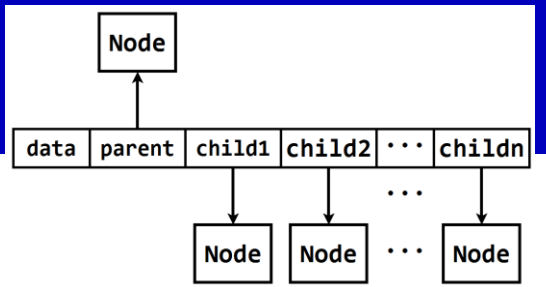
Nodes

- Every Tree element must be of type **Node**
- Nodes are the basic building blocks of a Tree
- Here is the full Node class:




```
class Node:
    "Node class for a general tree"
    def __init__(self, data, parent=None, children=None):
        self.data = data
        self.parent = parent
        if children is None:
            children = []
        self.children = children

    def __str__(self):
        return 'Node(' + str(self.data) + ')'
```

Node Test 1



- In spite of its simple and short code, there are many interesting things we can do with Nodes

```
def print_forward(node, level=0):
    print 4 * level * ' ' + str(node.data)
    for child in node.children:
        print_forward(child, level+1)

def test1():
    ishmael = Node('Ishmael')
    isac = Node('Isac')
    abraham = Node('Abraham', None, [isac, ishmael])
    isac.parent = abraham
    ishmael.parent = abraham
    jacob = Node('Jacob')
    esav = Node('Esav')
    isac.children = [jacob, esav]
    jacob.parent = isac
    esav.parent = isac

    print_forward(abraham)
```

Class BaseTree (part 1)

Abstract base class representing a tree structure

```
# ----- abstract methods that concrete subclass must support -----
def root(self):
    "Return Node representing the tree's root (or None if empty)"
    raise NotImplementedError('Must be implemented by subclass')

def parent(self, node):
    "Return Node representing node's parent (or None if node is root)"
    raise NotImplementedError('Must be implemented by subclass')

def children(self, node):
    "Generate an iteration of Nodes representing node's children"
    raise NotImplementedError('Must be implemented by subclass')

def num_children(self, node):
    "Return the number of children that Node node has"
    raise NotImplementedError('Must be implemented by subclass')

def __len__(self):
    "Return the total number of elements in the tree"
    raise NotImplementedError('Must be implemented by subclass')
```

Class BaseTree (part 2)

Abstract base class representing a tree structure

```
#------ concrete methods implemented in this class ------
def is_root(self, node):
    "Return True if Node node represents the root of the tree"
    r = self.root()
    return r == node

def is_leaf(self, node):
    "Return True if Node node does not have any children"
    return self.num_children(node) == 0

def is_empty(self):
    "Return True if the tree is empty"
    return len(self) == 0

def depth(self, node):
    "Return the number of levels from root to Node node"
    if self.is_root(node):
        return 0
    else:
        return 1 + self.depth(self.parent(node))
```

Class BaseTree (part 3)

Abstract base class representing a tree structure

```
#------ concrete methods implemented in this class ------
def height(self, node=None):
    """Return the height of the subtree rooted at Node node.
    If node is None, return the height of the entire tree.
    """
    if node is None:
        node = self.root()
    return self._height(node)    # start _height recursion

def _height(self, node):  # time is linear in size of subtree
    "Return the height of the subtree rooted at node"
    if self.is_leaf(node):
        return 0
    else:
        return 1 + max(self._height(c) for c in self.children(node))

def __iter__(self):
    "Generate an iteration of the tree's elements"
    for node in self.nodes():      # use same order as nodes()
        yield node.data()          # but yield data element
```

Class BaseTree (part 4)

Abstract base class representing a tree structure

```
#------ concrete methods implemented in this class ------
def nodes(self):
    "Generate an iteration of the tree's nodes"
    return self.preorder()    # return entire preorder iteration

def preorder(self):
    "Generate a preorder iteration of nodes in the tree"
    if not self.is_empty():
        for node in self._subtree_preorder(self.root()):  # start recursion
            yield node

def _subtree_preorder(self, node):
    "Generate a preorder iteration of nodes in subtree rooted at node"
    yield node     # visit node before its subtrees
    for c in self.children(node):             # for each child c
        for n in self._subtree_preorder(c):   # do preorder of c's subtree
            yield n                           # yielding each to our caller
```

Class BaseTree (part 5)

Abstract base class representing a tree structure

```
def postorder(self):
    "Generate a postorder iteration of nodes in the tree"
    if not self.is_empty():
        for node in self._subtree_postorder(self.root()):  # start recursion
            yield node

def _subtree_postorder(self, node):
    "Generate a postorder iteration of nodes in subtree rooted at node"
    for c in self.children(node):             # for each child c
        for n in self._subtree_postorder(c):  # do postorder of c's subtree
            yield n                           # yielding each to our caller
    yield node                                # visit node after its subtrees

def inorder(self):
    "Generate a inorder iteration of nodes in the tree"
    if not self.is_empty():
        for node in self._subtree_inorder(self.root()):  # start recursion
            yield node

def _subtree_inorder(self, node):
    "Generate a inorder iteration of nodes in subtree rooted at node"
    for c in self.children(node):             # for each child c
        for n in self._subtree_inorder(c):    # do inorder of c's subtree
            yield n                           # yielding each to our caller
    yield node                                # visit node after its subtrees
```

Class BaseTree (part 6)

Abstract base class representing a tree structure

```
#------ concrete methods implemented in this class ------
def bfs(self):
    "Generate a Breadth-First-Search (BFS) iteration of the tree nodes"
    if not self.is_empty():
        que = Queue()                       # nodes not yet yielded
        que.enqueue(self.root())            # starting with the root
        while not que.is_empty():
            node = que.dequeue()            # remove from front of the queue
            yield node                      # report this node
            for c in self.children(node):
                que.enqueue(c)              # add children to back of queue
```

BaseTree is too Basic

- The BaseTree class is indeed very basic
- For example, it has no method for:
  - Adding a new node to the tree
  - Removing a node from the tree
  - Replacing a node by a new node
  - We can't even add a root node !!
- In fact, we cannot do much with this class – it has 5 methods that are not implemented !!! (abstract/virtual methods) An object of this class is really not useful at all …
- But still, this is a very useful class, since it is a good basis for super classes that can provide all these methods

Class SimpleTree (part 1)

```
class SimpleTree(BaseTree):
    "Simple implementation of a tree structure"

    def __init__(self):
        "Create an initially empty tree"
        self._root = None       # private member
        self._size = 0          # private member

    def root(self):             # implementing an abstract method
        "Return the root Position of the tree (or None if tree is empty)"
        return self._root

    def parent(self, node):     # implementing an abstract method
        "Return node's parent (or None if p is root)"
        return node.parent

    def children(self, node):   # implementing an abstract method
        "Return node's children"
        return node.children
```

Class SimpleTree (part 2)

```
    def num_children(self, node):  # implementing an abstract method
        "Return the number of children of node"
        return len(node.children)

    def add_root(self, data):      # new method (not in BaseTree)
        """
        Place element data at the root of an empty tree
        and return new node.
        Raise ValueError if tree nonempty.
        """
        if self._root is not None:
            raise ValueError('Root exists')
        self._size = 1
        self._root = Node(data)
        return self._root
```

Class SimpleTree (part 3)

```
    def add_child(self, node, data):  # new method (not in BaseTree)
        """
        add a new child node with data at the end of children of node
        Return the new node.
        """
        child_node = Node(data, node)
        node.children.append(child_node)
        self._size += 1
        return child_node

    def __len__(self):   # implementing an abstract method in base class
        "Return the total number of elements in the tree"
        return self._size

    def __str__(self):   # new method (not in BaseTree)
        return str(self._root)
```

End of Class

Utilities

```
def height(t):
    "Height of SimpleTree t"
    root = t.root()
    if root is None:
        return 0
    return _height(t, root)

def _height(t, node):
    "Height of node in SimpleTree t"
    if t.num_children(node) == 0:
        return 0
    children_heights = [_height(t,c) for c in t.children(node)]
    return 1 + max(children_heights)

def ancestors(t, node):
    parent = t.parent(node)
    if parent is None:
        return []
    else:
        return ancestors(t, parent) + [parent]
```

Building a Family Tree

```
def family_tree():
    t = SimpleTree()
    root = t.add_root('Abraham')
    isac = t.add_child(root, 'Isaac')
    ishm = t.add_child(root, 'Ishmael')
    t.add_child(root, 'Madan')
    esav = t.add_child(isac, 'Esav')
    jaco = t.add_child(isac, 'Jacob')
    t.add_child(jaco, 'Reuven')
    t.add_child(jaco, 'Shimon')
    t.add_child(jaco, 'Dan')
    t.add_child(jaco, 'Naftali')
    return t
```

Test

```
def test1():
    t = family_tree()
    print t
    root = t.root()
    isac = t.children(root)[0]  # isaac is the first child
    print "isac =", isac
    print "Isac's parent =", t.parent(isac)
    print "Isac's children =", t.children(isac)
    print "Isac's children =", [str(node) for node in t.children(isac)]
    print "Height =", height(t)
    for node in t.nodes():
        data = node.data
        A = [a.data for a in ancestors(t, node)]
        print "Ancestors of %s = %s" % (node.data, A)
```