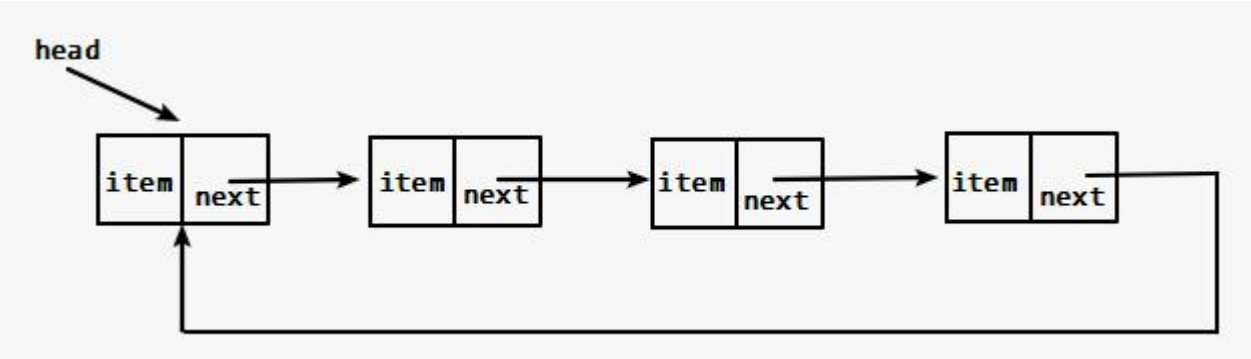
一、链表

1.单向循环链表

单链表的一个变形是单向循环链表，链表中最后一个节点的next域不再为None，而是指向链表的头节点。



操作

- is_empty() 判断链表是否为空
- length() 返回链表的长度
- travel() 遍历
- add(item) 在头部添加一个节点
- append(item) 在尾部添加一个节点
- insert(pos, item) 在指定位置pos添加节点
- remove(item) 删除一个节点
- search(item) 查找节点是否存在

节点实现：

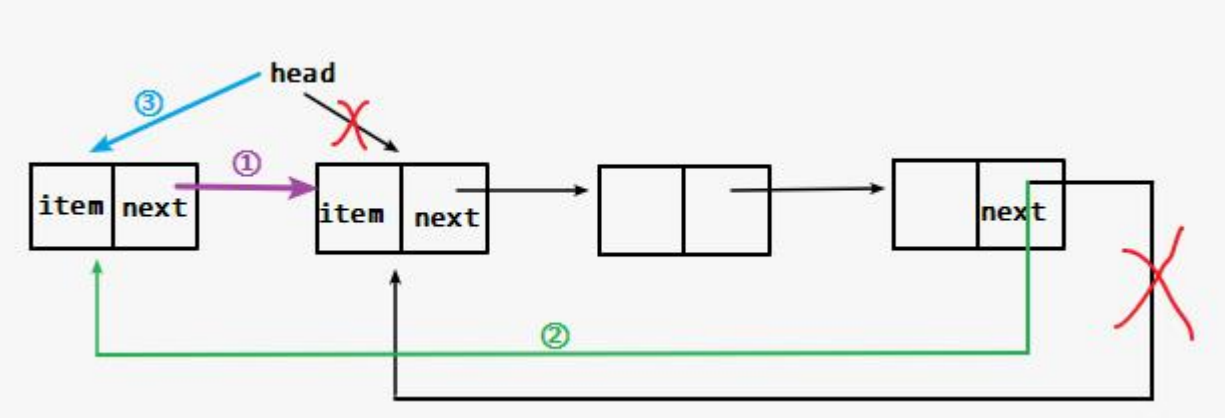
```
class SingleCycleNode():
    '''单循环链表节点'''
    def init (self,item):
        self.item = item
        self.next = None
```

操作实现：

```
class SinCycLinkedlist(object):
    """单向循环链表"""
    def   init  (self):
        self.  head = None

def is_empty(self):
    """判断链表是否为空"""
    return self.  head == None
```

```
def length(self):
    """链表的长度"""
    #如果链表为空，返回0
    if self.is_empty():
        return 0
    count = 1
    cur = self.  head
    while cur.next != self. head:
        count += 1
        cur = cur.next
    return count

def travel(self):
    '''遍历链表'''
    if self.is_empty():
        return
    cur = self. head
    print(cur.item)
    while cur.next != self. head:
        cur = cur.next
        print(cur.item)

def insert(self, pos, item):
    """在指定位置添加节点"""
    #与单向链表相同
    if pos <= 0:
        self.add(item)
    elif pos > (self.length() - 1):
        self.append(item)
    else:
        node = SingleCycleNode(item)
        cur = self.  head
        count = 0
        # 移动到指定位置的前一个位置
        while count < (pos - 1):
            count += 1
            cur = cur.next
        node.next = cur.next
        cur.next = node

def search(self, item):
    """查找节点是否存在"""
    if self.is_empty():
        return  False
    cur = self.  head
    if cur.item == item:
        return True
    while cur.next != self. head:
        cur = cur.next
        if cur.item == item:
            return True
    return False
```

头部添加节点




```
def add(self,item):
    """头部添加节点"""
    node = SingleCycleNode(item)
    #如果为空链表
    if self.is_empty():
        self.  head = node
        node.next = self. head
    else:
        #添加的节点指向  head
        node.next = self.  head
        #移动到链表尾部
        cur = self.  head
        while cur.next != self. head:
            cur = cur.next
        #尾节点的next指向node
        cur.next = node
        #head指向node
        self.  head = node
```

尾部添加节点

```
def append(self,item):
    '''尾部添加节点'''
    node = SingleCycleNode(item)
    if self.is_empty():
        self. head = node
        node.next = self.  head
    else:
        #移到链表尾部
        cur = self.  head
        while cur.next != self. head:
            cur = cur.next
        #将尾节点指向node
        cur.next = node
        #将node的next指向头节点
        node.next = self.  head
```

删除节点

```
def remove(self,item):
    '''删除一个节点'''
    #若链表为空，则直接返回
    if self.is_empty():
        return
    #将cur指向头节点
    cur = self. head
    pre = None
    while cur.next != self. head:
        if cur.item == item:
            #判断是否为头节点
            if cur == self.  head:
                #如果是头节点，则找到尾节点
                tail = self.  head
                while tail.next != self. head:
                    tail = tail.next
                #将head指向头节点的下一个节点
                self.  head = cur.next
                #将尾节点的next指向头节点
                tail.next = self.  head
            else:
                #不是头节点
                pre.next = cur.next
            return
        else:
            pre = cur
            cur = cur.next
    #cur指向了尾部
    if cur.item == item:
        if cur == self.  head:
            #只有一个节点
            self. head = None
        else:
            pre.next = self.  head
```

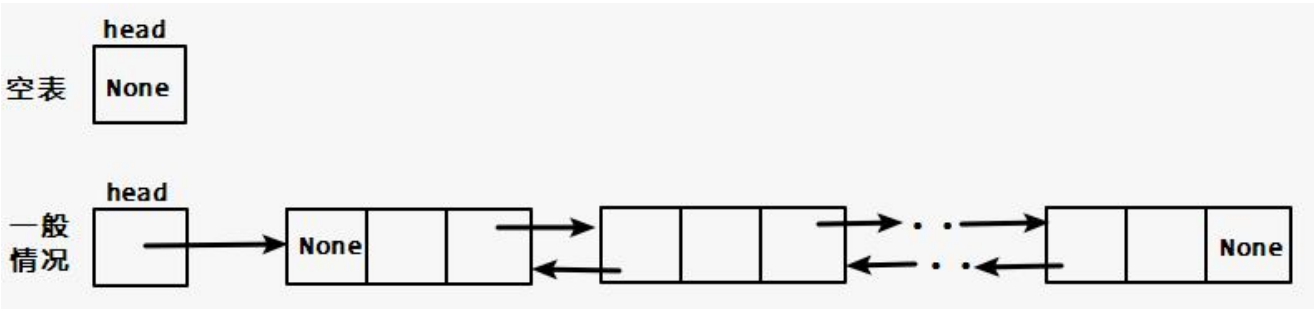
测试

```
if   name   == '  main  ':
    L3 = SinCycLinkedlist()
    L3.add(66)
    L3.append("haha")
    print(L3.is_empty())
    print(L3.length())
    L3.travel()
    L3.insert(1,233)
    print('---------------')
    print(L3.length())
    L3.travel()
    print(L3.search(233))
    print('----------------')
```

```
    L3.remove(66)
    print(L3.length())
    L3.travel()
```

2.双向链表

一种更复杂的链表是“双向链表”或“双面链表”。每个节点有两个链接：一个指向前一个节点，当此节点为第一个节点时，指向空值；而另一个指向下一个节点，当此节点为最后一个节点时，指向空值。



节点实现：

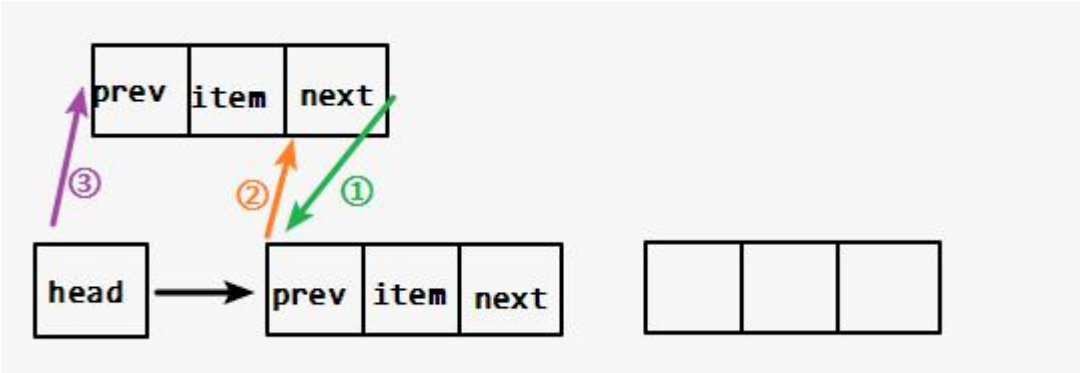
```
class DoubleNode():
    '''双向链表节点'''
    def init (self,item):
        self.item = item
        self.next = None
        self.prev = None
```

操作实现

```
class DoubleLinkList():
    def   init  (self):
        self.  head = None

    def is_empty(self):
        '''判断链表是否为空'''
        return self.  head == None

    def length(self):
        '''返回链表的长度'''
        cur = self. head
        count = 0
        while cur != None:
            count += 1
            cur = cur.next
        return count

    def travel(self):
        '''遍历链表'''
```

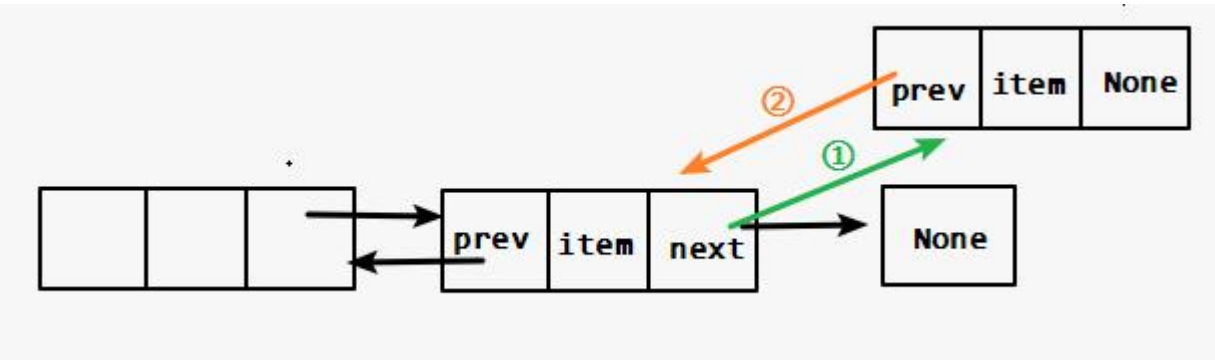
```
        cur = self. head
        while cur != None:
            print(cur.item)
            cur = cur.next

    def serch(self, item):
        '''查找元素是否存在'''
        cur = self. head
        while cur != None:
            if cur.item == item:
                return True
            cur = cur.next
        return Falsec
```

头部插入元素



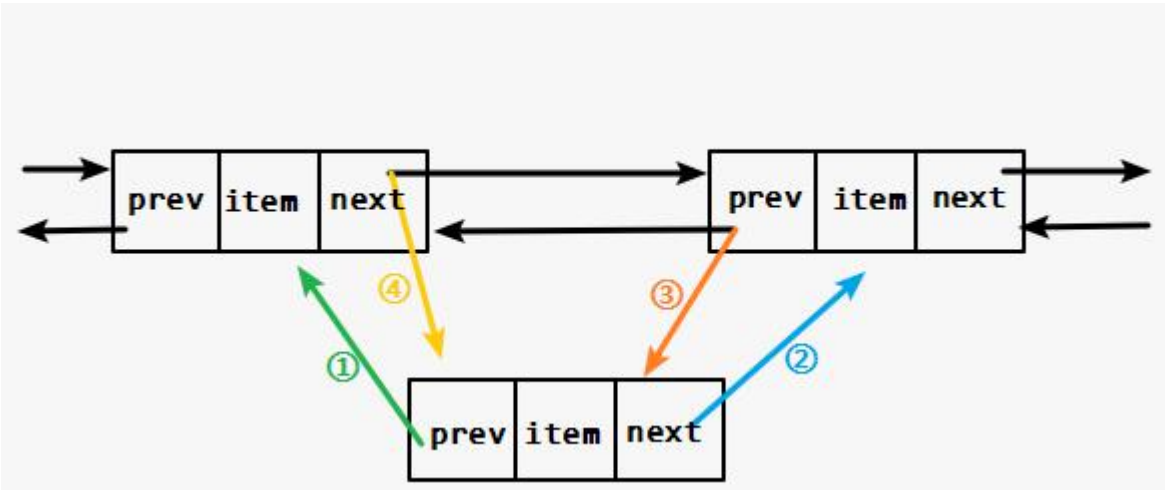

```
def add(self,item):
    '''头部插入元素'''
    node = DoubleNode(item)
    if self.is_empty():
        #如果是空链表，将 head指向node
        self. head = node
    else:
        #将node的next指向 head的头节点
        node.next = self.  head
        #将 head的头节点指向node
        self.  head.prev = node
        #将  head指向node
        self.  head = node
```

尾部插入元素




```
def append(self,item):
    '''尾部插入元素'''
    node = DoubleNode(item)
    if self.is_empty():
        self. head = node
    else:
        cur = self.  head
        #移动到链表尾部
        while cur.next != None:
            cur = cur.next
        #尾节点cur的next指向node
        cur.next = node
        #node的prev指向cur
        node.prev = cur
```

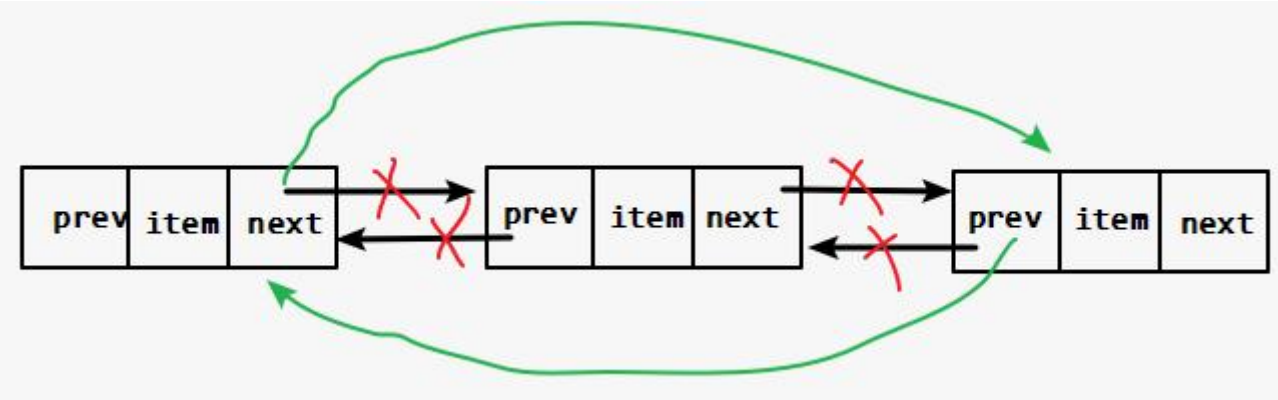
指定位置插入节点




```
def insert(self,pos,item):
    '''在指定位置添加节点'''
    if pos <= 0:
        self.add(item)
    elif pos > self.length()-1:
        self.append(item)
    else:
        node = DoubleNode(item)
        cur = self.  head
        count = 0
```

```
        #移动到指定位置的前一个位置
        while count < pos-1:
            count += 1
            cur = cur.next
        #将node的prev指向cur
        node.prev = cur
        #将node的next 指向cur的下一个节点
        node.next = cur.next
        #将cur下一个节点的prev指向node
        cur.next.prev = node
        #将cur的next指向node
        cur.next = node
```

删除元素




```
def remove(self,item):
    '''删除元素'''
    cur = self.  head
    while cur != None:
        if cur.item == item:
            #判断是否是头节点
            if cur == self. head:
                self.  head = cur.next
                #头节点，不止一个节点时，删除后，元素的prev为None
                if cur.next:
                    cur.next.prev = None
            else:
                cur.prev.next = cur.next
                #后面有节点的话
                if cur.next:
                    cur.next.prev = cur.prev
            break
        else:
            cur = cur.next
```

测试

```
if name  == '  main  ':
    L2 = DoubleLinkList()
```

```
    L2.add("haha")
    L2.add(5)
    L2.add(4)
    L2.travel()
    print(L2.length())
    print(L2.serch(5))
    L2.insert(1,"heihei")
    print('---------------------')
    L2.travel()
    print('---------------------')
    L2.remove(3)
    L2.travel()
```

二、栈和队列

1.栈

栈（stack），有些地方称为堆栈，是一种容器，可存入数据元素、访问元素、删除元素，它的特点在于只能允许在容器的一端（称为栈顶端指标，英语：top）进行加入数据（英语：push）和输出数据（英语：pop）的运算。没有了位置概念，保证任何时候可以访问、删除的元素都是此前最后存入的那个元素，确定了一种默认的访问顺序。

由于栈数据结构只允许在一端进行操作，因而按照后进先出（LIFO, Last In First Out）的原理运

作。**栈的实现**

栈的操作

- Stack() 创建一个新的空栈
- push(item) 添加一个新的元素item到栈顶
- pop() 弹出栈顶元素
- peek() 返回栈顶元素
- is_empty() 判断栈是否为空
- size() 返回栈的元素个数

```
class Stack(object):
    """栈"""
    def init (self):
         self.items = []

    def is_empty(self):
        """判断是否为空"""
        return self.items == []

    def push(self, item):
        """加入元素"""
        self.items.append(item)

    def pop(self):
        """弹出元素"""
        return self.items.pop()
```

```
    def peek(self):
        """返回栈顶元素"""
        return  self.items[len(self.items)-1]

    def size(self):
        """返回栈的大小"""
        return len(self.items)

if name == " main ":
    stack = Stack()
    stack.push("hello")
    stack.push("world")
    stack.push("hahaha")
    print(stack.size())
    print(stack.peek())
    print(stack.pop())
    print(stack.pop())
    print(stack.pop())
```

2.队列

队列(queue)是一种具有先进先出特征的线性数据结构，元素的增加只能在一端进行，元素的删除只能在另一端进行。 类似于在超市收银处排队，队列里的的第一个人首先接受服务，新的元素通过入队的方式添加到队列的末尾，而出队就是将队列的头元素删除。

1） 队列的实现

队列的操作

- Queue() 创建一个空的队列
- enqueue(item) 往队列中添加一个item元素
- dequeue() 从队列头部删除一个元素
- is_empty() 判断一个队列是否为空
- size() 返回队列的大小

```
class Queue(object):
    """队列"""
    def init (self):
        self.items = []

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

    def enqueue(self, item):
        """进队列"""
        self.items.insert(0,item)

    def dequeue(self):
        """出队列"""
        return self.items.pop()

    def size(self):
```

```
        """返回大小"""
        return len(self.items)

if __name__ == "__main__":
    q = Queue()
    q.enqueue("hello")
    q.enqueue("world")
    q.enqueue("itcast")
    print(q.size())
    print(q.dequeue())
    print(q.dequeue())
    print(q.dequeue())
```

2）双端队列

双端队列（deque，全名double-ended queue），是一种具有队列和栈的性质的数据结构。

双端队列中的元素可以从两端弹出，其限定插入和删除操作在表的两端进行。双端队列可以在队列任意一端入队和出队。

双端队列操作

- Deque() 创建一个空的双端队列
- add_front(item) 从队头加入一个item元素
- add_rear(item) 从队尾加入一个item元素
- remove_front() 从队头删除一个item元素
- remove_rear() 从队尾删除一个item元素
- is_empty() 判断双端队列是否为空
- size() 返回队列的大小

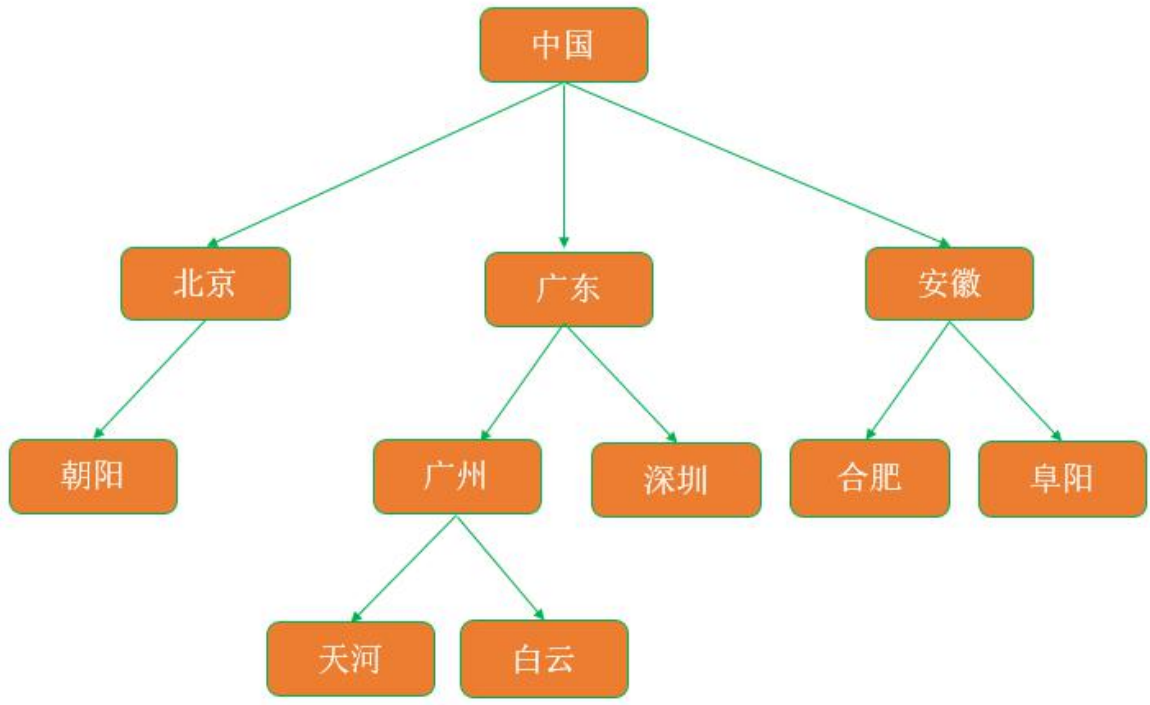
```
class Deque(object):
    """双端队列"""
    def init (self):
        self.items = []

    def is_empty(self):
        """判断队列是否为空"""
        return self.items == []

    def add_front(self, item):
        """在队头添加元素"""
        self.items.insert(0,item)

    def add_rear(self, item):
        """在队尾添加元素"""
        self.items.append(item)

    def remove_front(self):
        """从队头删除元素"""
        return self.items.pop(0)

    def remove_rear(self):
        """从队尾删除元素"""
```

```
        return self.items.pop()

    def size(self):
        """返回队列大小"""
        return len(self.items)


if __name__ == "__main__":
    deque = Deque()
    deque.add_front(1)
    deque.add_front(2)
    deque.add_rear(3)
    deque.add_rear(4)
    print(deque.size())
    print(deque.remove_front())
    print(deque.remove_front())
    print(deque.remove_rear())
    print(deque.remove_rear())
```

三、树

树的概念 树(tree)是一种抽象数据类型（ADT）或是实作这种抽象数据类型的数据结构，用来模拟具有树状结构性质的数据集合。它是由n（n>=1）个有限节点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

- 每个节点有零个或多个子节点；
- 没有父节点的节点称为根节点；
- 每一个非根节点有且只有一个父节点；
- 除了根节点外，每个子节点可以分为多个不相交的子树；



1.树的术语

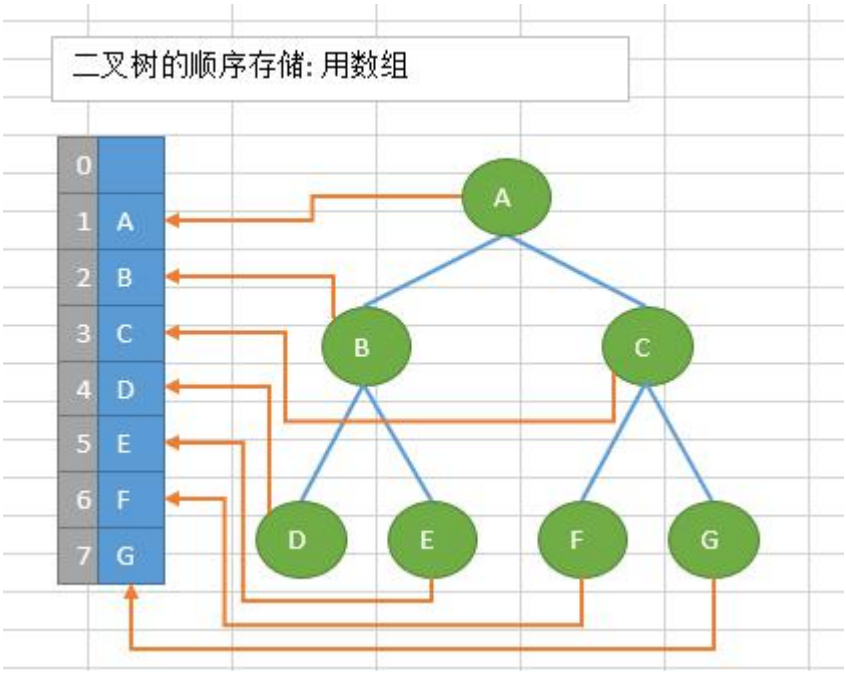
- 节点的度：一个节点含有的子树的个数称为该节点的度；
- 树的度：一棵树中，最大的节点的度称为树的度；
- 叶节点或终端节点：度为零的节点；
- 父亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点；
- 孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点；
- 兄弟节点：具有相同父节点的节点互称为兄弟节点；
- 节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推；
- 树的高度或深度：树中节点的最大层次；
- 堂兄弟节点：父节点在同一层的节点互为堂兄弟；节
- 点的祖先：从根到该节点所经分支上的所有节点；
- 子孙：以某节点为根的子树中任一节点都称为该节点的子孙。
- 森林：由m（m>=0）棵互不相交的树的集合称为森林。

2.树的种类

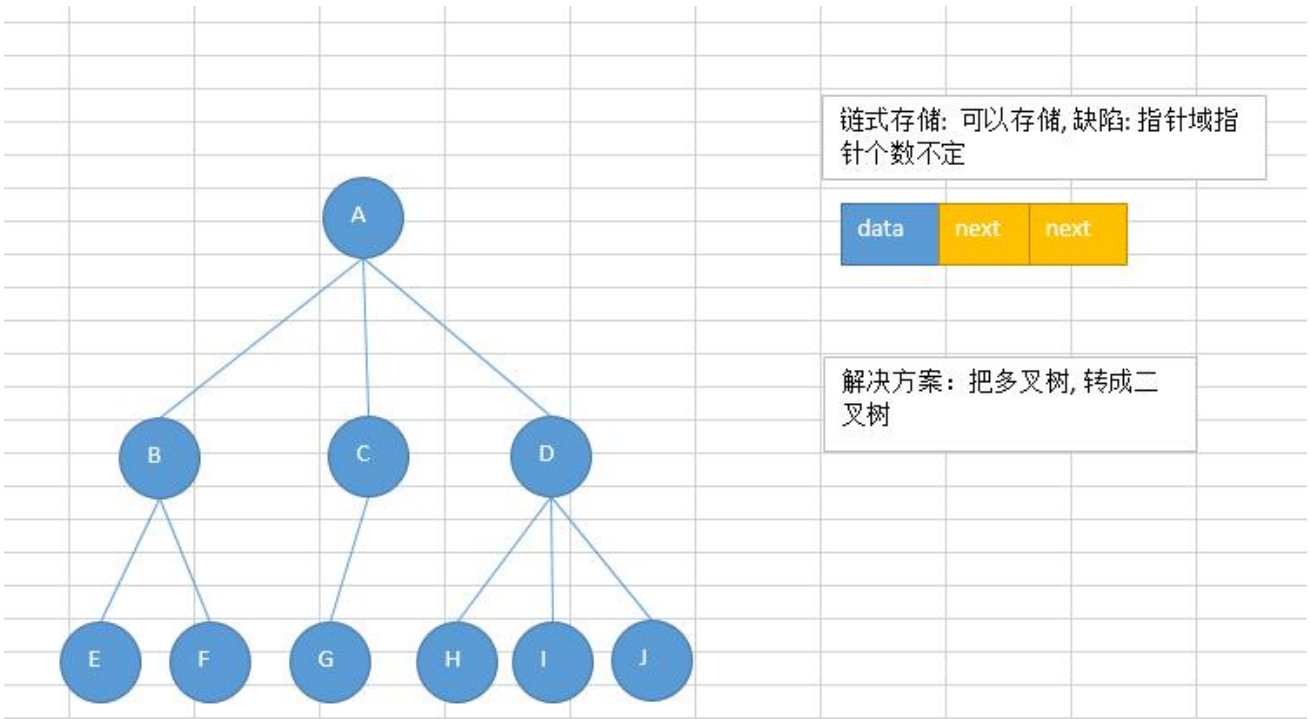
- **无序树**：树中任意节点的子节点之间没有顺序关系，这种树称为无序树，也称为自由树；
- **有序树**：树中任意节点的子节点之间有顺序关系，这种树称为有序树；
  - **二叉树**：每个节点最多含有两个子树的树称为二叉树；
    - **完全二叉树**：对于一颗二叉树，假设其深度为d(d>1)。除了第d层外，其它各层的节点数目均已达最大值，且第d层所有节点从左向右连续地紧密排列，这样的二叉树被称为完全二叉树，其中满二叉树的定义是所有叶节点都在最底层的完全二叉树;
    - **平衡二叉树（AVL树）**：当且仅当任何节点的两棵子树的高度差不大于1的二叉树；
    - **排序二叉树（二叉查找树（Binary Search Tree））**，也称二叉搜索树、有序二叉树);
  - **霍夫曼树**（用于信息编码）：带权路径最短的二叉树称为哈夫曼树或最优二叉树；
  - **B树**：一种对读写操作进行优化的自平衡的二叉查找树，能够保持数据有序，拥有多余两个子树。

3.树的存储与表示

顺序存储：将数据结构存储在固定的数组中，然在遍历速度上有一定的优势，但因所占空间比较大，是非主流二叉树。二叉树通常以链式存储。



链式存储：



4.常见的一些树的应用场景

1.xml，html等，那么编写这些东西的解析器的时候，不可避免用到树

2.路由协议就是使用了树的算法

3.mysql数据库索引

4.文件系统的目录结构

5.所以很多经典的AI算法其实都是树搜索，此外机器学习中的decision　　tree、random forest也是树结构

5.二叉树

二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）

1）二叉树的性质(特性)

1: 在二叉树的第i层上至多有2^(i-1)个结点（i>0）

2: 深度为k的二叉树至多有2^k - 1个结点（k>0）

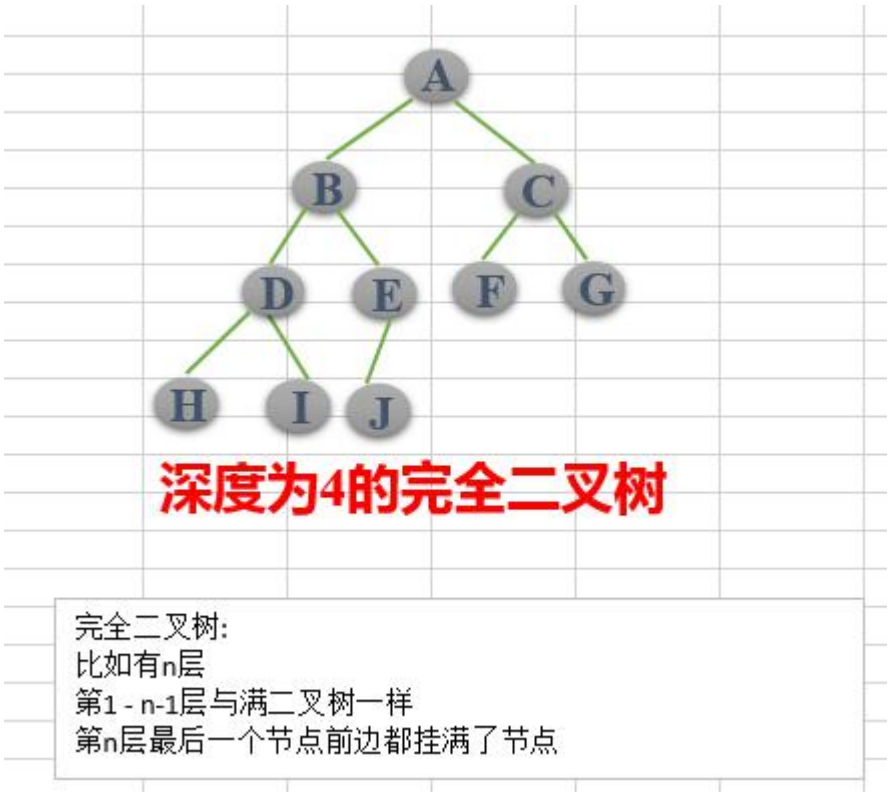
3: 对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1;

证明过程如下：
假设二叉树的0度,1度,2度结点为n0,n1,n2,总节点数为T
则有按照结点求和的
T = n0 + n1 + n2 （1）
按照边求和得：
T = n1 + 2 * n2 + 1 （2）
所以 （2） - （1）可得
n2 + 1 - n0 = 0
所以n0 = n2 + 1

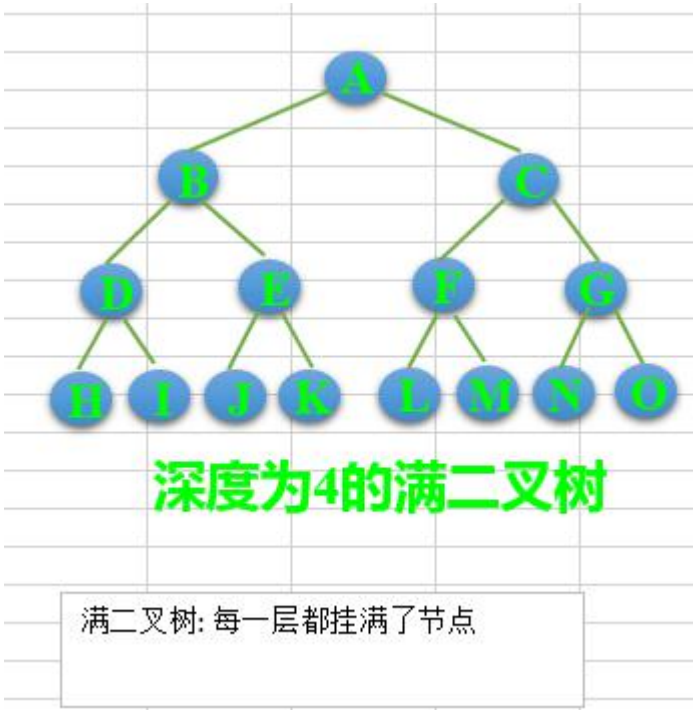
4:具有n个结点的完全二叉树的深度必为 $int(log_2^{n})+1$

5:对完全二叉树，若从上至下、从左至右编号，则编号为i 的结点，其左孩子编号必为2i+1，其右孩子编号必为2i＋2；其双亲的编号必为0.5(i-1)取整（i＝0 时为根）

- (1)完全二叉树——若设二叉树的高度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第h层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二叉树。



- (2)满二叉树——除了叶结点外每一个结点都有左右子叶且叶子结点都处在最底层的二叉树。
-



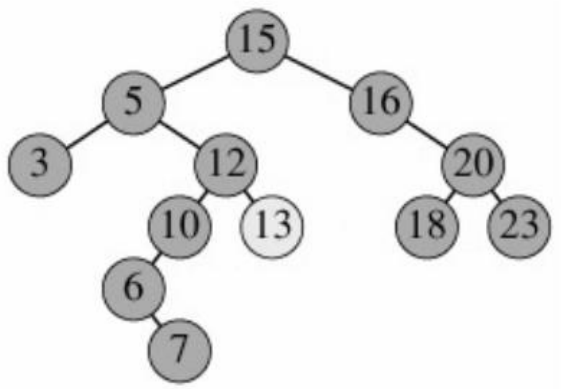
(3)二叉查找树是满足以下条件的二叉树：

1、左子树上的所有结点值均小于根结点值

2、右子树上的所有结点值均不小于根结点值

3、左右子树也满足上述两个条件

二叉树的建立，依次插入：15,5,3,12,16,20,23,13,18,10,6,7



2）二叉树的节点表示以及树的创建

通过Node类定义三个属性，分别为elem本身的值，还有lchild左孩子和rchild右孩子

```
class Node():
    """节点类"""
    def init (self, elem=-1, lchild=None, rchild=None):
        self.elem = elem
        self.lchild = lchild
        self.rchild = rchild
```

树的创建,创建一个树的类，并给一个root根节点，一开始为空，随后添加节点

```
class Tree():
    """树类"""
    def init (self, root=None):
        self.root = root

    def add(self, elem):
        """为树添加节点"""
        node = Node(elem)
        #如果树是空的，则对根节点赋值
        if self.root == None:
            self.root = node
        else:
            queue = []
            queue.append(self.root)
            #对已有的节点进行层次遍历
            while queue:
                #弹出队列的第一个元素
```

```
            cur = queue.pop(0)
            if cur.lchild == None:
                cur.lchild = node
                return
            elif cur.rchild == None:
                cur.rchild = node
                return
            else:
                #如果左右子树都不为空，加入队列继续判断
                queue.append(cur.lchild)
                queue.append(cur.rchild)
```

3）二叉树的遍历

树的遍历是树的一种重要的运算。所谓遍历是指对树中所有结点的信息的访问，即依次对树中每个结点访问一次且仅访问一次，我们把这种对所有节点的访问称为遍历（traversal）。那么树的两种重要的遍历模式是深度优先遍历和广度优先遍历,**深度优先一般用递归，广度优先一般用队列。**一般情况下能用递归实现的算法大部分也能用堆栈来实现。

①深度优先遍历

对于一颗二叉树，深度优先搜索(Depth First Search)是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。那么深度遍历有重要的三种方法。这三种方式常被用于访问树的节点，它们之间的不同在于访问每个节点的次序不同。这三种遍历分别叫做先序遍历（preorder），中序遍历（inorder）和后序遍历（postorder）。我们来给出它们的详细定义，然后举例看看它们的应用。

- 先序遍历 在先序遍历中，我们先访问根节点，然后递归使用先序遍历访问左子树，再递归使用先序遍历访问右子树 根节点->左子树->右子树

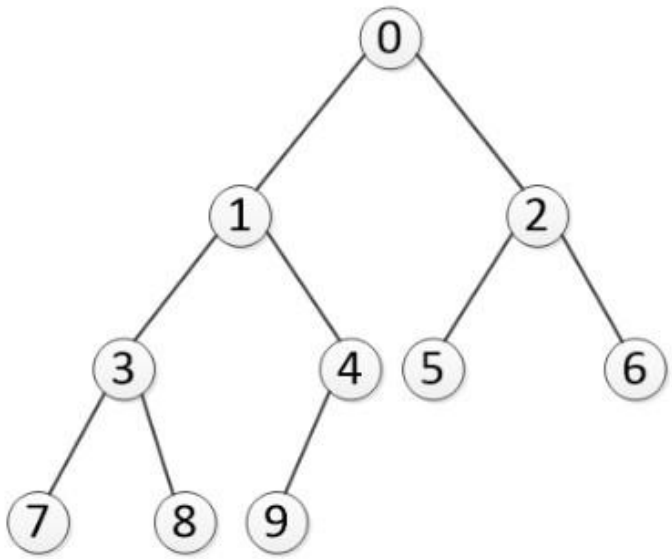
```
def preorder(self, root):
  """递归实现先序遍历"""
  if root == None:
      return
  print(root.elem)
  self.preorder(root.lchild)
  self.preorder(root.rchild)
```

- 中序遍历 在中序遍历中，我们递归使用中序遍历访问左子树，然后访问根节点，最后再递归使用中序遍历访问右子树 左子树->根节点->右子树

```
def inorder(self, root):
  """递归实现中序遍历"""
  if root == None:
      return
  self.inorder(root.lchild)
  print(root.elem)
  self.inorder(root.rchild)
```

后序遍历 在后序遍历中，我们先递归使用后序遍历访问左子树和右子树，最后访问根节点 左子树->右子树->根节点

```
def postorder(self, root):
  """递归实现后续遍历"""
  if root == None:
      return
  self.postorder(root.lchild)
  self.postorder(root.rchild)
  print(root.elem)
```



层次遍历：0 1 2 3 4 5 6 7 8 9

先序遍历：0 1 3 7 8 4 9 2 5 6

中序遍历：7 3 8 1 9 4 0 5 2 6

后序遍历：7 8 3 9 4 1 5 6 2 0

②广度优先遍历(层次遍历)

从树的root开始，从上到下从从左到右遍历整个树的节点

```
def breadth_travel(self,root):
  """利用队列实现树的层次遍历"""
  if root == None:
      return
  queue = []
  queue.append(root)
  while queue:
      node = queue.pop(0)
      print(node.elem)
      if node.lchild != None:
          queue.append(node.lchild)
      if node.rchild != None:
          queue.append(node.rchild)
```