

数据结构

数据结构

数据是一个抽象的概念,将其进行分类后得到程序设计语言中的基本类型。如: `int`, `float`, `char` 等。数据元素之间不是独立的,存在特定的关系,这些关系便是结构。数据结构指数据对象中数据元素之间的关系。

Python 给我们提供了很多现成的数据结构类型,这些系统自己定义好的,不需要我们自己去定义的数据结构叫做 Python 的内置数据结构,比如列表、元组、字典。而有些数据组织方式,Python 系统里面没有直接定义,需要我们自己去定义实现这些数据的组织方式,这些数据组织方式称之为 Python 的扩展数据结构,比如栈,队列等。

顺序表

在程序中,经常需要将一组(通常是同为某个类型的)数据元素作为整体管理和使用,需要创建这种元素组,用变量记录它们,传进传出函数等。一组数据中包含的元素个数可能发生变化(可以增加或删除元素)。

对于这种需求,最简单的解决方案便是将这样一组元素看成一个序列,用元素在序列里的位置和顺序,表示实际应用中的某种有意义的信息,或者表示数据之间的某种关系。

这样的一组序列元素的组织形式,我们可以将其抽象为线性表。一个线性表是某类元素的一个集合,还记录着元素之间的一种顺序关系。线性表是最基本的数据结构之一,在实际程序中应用非常广泛,它还经常被用作更复杂的数据结构的实现基础。

根据线性表的实际存储方式,分为两种实现模型:

顺序表,将元素顺序地存放在一块连续的存储区里,元素间的顺序关系由它们的存储顺序自然表示。

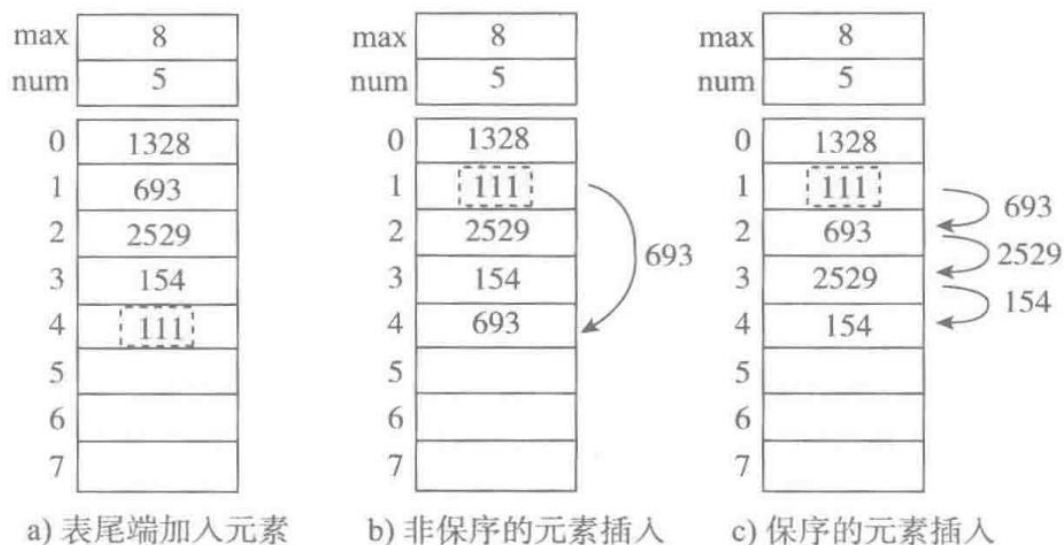
链表,将元素存放在通过链接构造起来的一系列存储块中。

Python 中的 `list` 和 `tuple` 两种类型采用了顺序表的实现技术,`tuple` 是不可变类型,即不变的顺序表,因此不支持改变其内部状态的任何操作,而其他方面,则与 `list` 的性质类似。

顺序表的操作

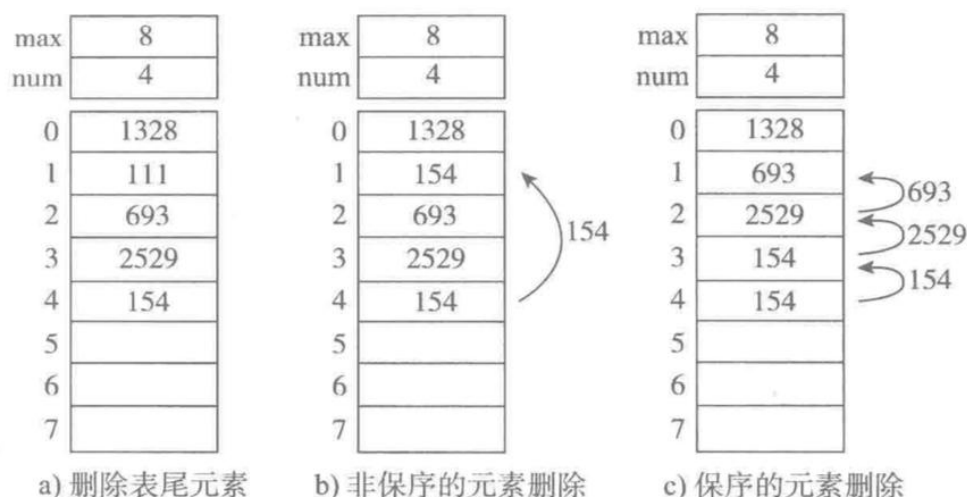
增加元素

如图所示,为顺序表增加新元素 111 的三种方式



- a. 尾端加入元素，时间复杂度为 $O(1)$
- b. 非保序的加入元素（不常见），时间复杂度为 $O(1)$
- c. 保序的元素加入，时间复杂度为 $O(n)$

删除元素



- a. 删除表尾元素，时间复杂度为 $O(1)$
- b. 非保序的元素删除（不常见），时间复杂度为 $O(1)$
- c. 保序的元素删除，时间复杂度为 $O(n)$

Python 标准类型 `list` 就是一种元素个数可变的线性表，可以加入和删除元素，并在各种操作中维持已有元素的顺序（即保序）。

timeit 模块

`timeit` 模块可以用来测试一小段 Python 代码的执行速度。

```
class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>)
```

`Timer` 是测量小段代码执行速度的类。其中 `stmt` 参数是要测试的代码语句（statement）；`setup`

参数是运行代码时需要的设置；timer 参数是一个定时器函数，与平台有关。

```
timeit.Timer.timeit(number=1000000)
```

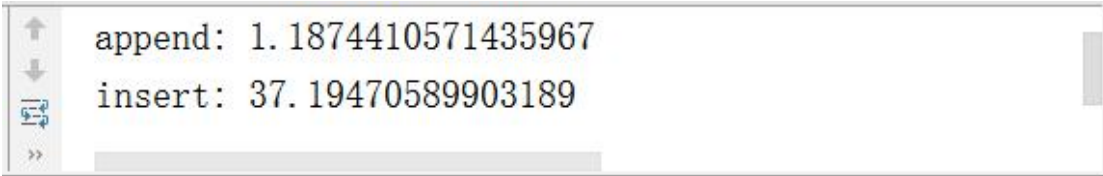
Timer 类中测试语句执行速度的对象方法。number 参数是测试代码时的测试次数，默认为 1000000 次。方法返回执行代码的平均耗时，一个 float 类型的秒数。

【示例】测试 list 列表中 append、insert 方法执行速度

```
from timeit import Timer
def append_test():
    li=[]
    for i in range(10000):
        li.append(i)
def insert_test():
    li=[]
    for i in range(10000):
        li.insert(0,i)
timer1=Timer('append_test()','from __main__ import append_test')
print('append:',timer1.timeit(1000))

timer1=Timer('insert_test()','from __main__ import insert_test')
print('insert:',timer1.timeit(1000))
```

执行结果：



```
↑
↓
append: 1.1874410571435967
insert: 37.19470589903189
>>
```

在 Python 的官方实现中，list 就是一种采用分离式技术实现的动态顺序表。这就是为什么用 list.append(x)（或 list.insert(len(list), x)，即尾部插入）比在指定位置插入元素效率高的原因。

在 Python 的官方实现中，list 实现采用了如下的策略：在建立空表（或者很小的表）时，系统分配一块能容纳 8 个元素的存储区；在执行插入操作（insert 或 append）时，如果元素存储区满就换一块 4 倍大的存储区。但如果此时的表已经很大（目前的阈值为 50000），则改变策略，采用加一倍的方法。引入这种改变策略的方式，是为了避免出现过多空闲的存储位置。

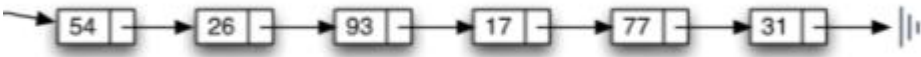
链表

顺序表的构建需要预先知道数据大小来申请连续的存储空间，而在进行扩充时又需要进行数据的搬迁，所以使用起来并不是很灵活。

链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。

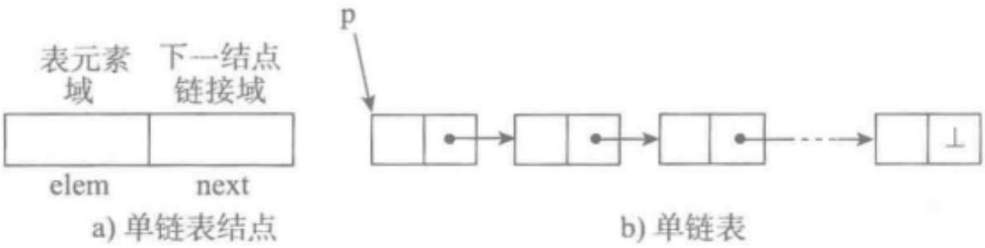
链表的定义

链表（Linked list）是一种常见的基础数据结构，是一种线性表，但是不像顺序表一样连续存储数据，而是在每一个节点（数据存储空间）里存放下一个节点的位置信息（即地址）。



单向链表

单向链表也叫单链表，是链表中最简单的一种形式，它的每个节点包含两个域，一个信息域（元素域）和一个链接域。这个链接指向链表中的下一个节点，而最后一个节点的链接域则指向一个空值。



- (1) 表元素域 elem 用来存放具体的数据。
- (2) 链接域 next 用来存放下一个节点的位置（python 中的标识）
- (3) 变量 p 指向链表的头节点（首节点）的位置，从 p 出发能找到表中的任意节点。

【示例】节点实现

```
class SingleNode(object):
    """单链表的结点"""
    def __init__(self,item):
        # _item 存放数据元素
        self.item = item
        # _next 是下一个节点的标识
        self.next = None
```

方法名	说明
is_empty()	链表是否为空
length()	链表长度
travel()	遍历整个链表
add(item)	链表头部添加元素
append(item)	链表尾部添加元素

insert(pos, item)	指定位置添加元素
remove(item)	删除节点
search(item)	查找节点是否存在

【示例】单链表的实现

```
#构造单向链表类
class SingleLinkedList:
    #初始化方法
    def __init__(self,node=None):
        #判断 node 是否为空
        if node !=None:
            headNode=Node(node)
            self.__head=headNode
        else:
            self.__head=None

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

    def length(self):
        """链表长度"""
        # cur 初始时指向头节点
        cur = self.__head
        count = 0
        # 尾节点指向 None，当未到达尾部时
        while cur != None:
            count += 1
            # 将 cur 后移一个节点
            cur = cur.next
        return count

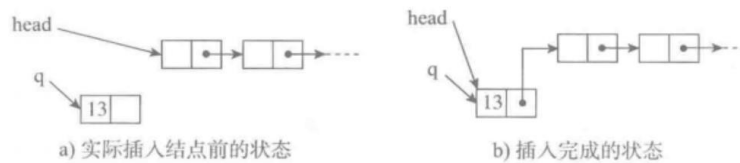
    def travel(self):
        """遍历链表"""
        cur = self.__head
        while cur != None:
```

```

print(cur.item)
cur = cur.next
print("")

```

【示例】头部添加



```

def add(self, item):
    """头部添加元素"""
    # 先创建一个保存 item 值的节点
    node = SingleNode(item)
    # 将新节点的链接域 next 指向头节点，即 _head 指向的位置
    node.next = self._head
    # 将链表的头 _head 指向新节点
    self._head = node

```

【示例】尾部添加

```

def append(self, item):
    # 将传入的值构造节点
    node = Node(item)
    if self.is_empty(): # 单链表为空时候
        self._head = node
    else: # 单链表不为空
        curNode = self._head
        while curNode.next != None:
            curNode = curNode.next
        # 修改节点指向 最后一个节点的 next 指向 node
        curNode.next = node

```

【示例】指定位置添加元素

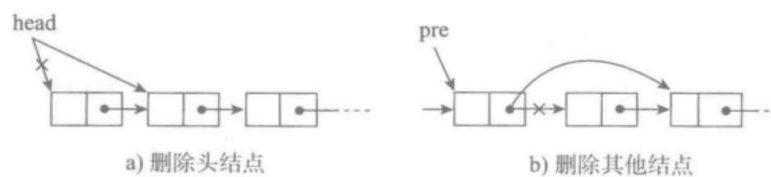


```

def insert(self, pos, item):
    """指定位置添加元素"""
    # 若指定位置 pos 为第一个元素之前，则执行头部插入
    if pos <= 0:
        self.add(item)
    # 若指定位置超过链表尾部，则执行尾部插入
    elif pos > (self.length()-1):
        self.append(item)
    # 找到指定位置
    else:
        node = SingleNode(item)
        count = 0
        # pre 用来指向指定位置 pos 的前一个位置 pos-1，初始从头节点开始移动到指定位置
        pre = self.__head
        while count < (pos-1):
            count += 1
            pre = pre.next
        # 先将新节点 node 的 next 指向插入位置的节点
        node.next = pre.next
        # 将插入位置的前一个节点的 next 指向新节点
        pre.next = node

```

【示例】删除节点



```

def remove(self, item):
    """删除节点"""
    cur = self.__head
    pre = None
    while cur != None:
        # 找到了指定元素
        if cur.item == item:

```

```

        # 如果第一个就是删除的节点
        if not pre:
            # 将头指针指向头节点的后一个节点
            self.__head = cur.next
        else:
            # 将删除位置前一个节点的 next 指向删除位置的后一个节点
            pre.next = cur.next
        break
    else:
        # 继续按链表后移节点
        pre = cur
        cur = cur.next

```

【示例】查找节点是否存在

```

def search(self,item):
    """链表查找节点是否存在，并返回 True 或者 False"""
    cur = self.__head
    while cur != None:
        if cur.item == item:
            return True
        cur = cur.next
    return False

```

【示例】测试插入、删除、查找操作

```

if __name__ == '__main__':
    #初始化元素值为 20 的单向链表
    # singleLinkList=SingleLinkList(20)
    #初始化一个空的单向链表
    singleLinkList=SingleLinkList()
    print('是否是空链表: ',singleLinkList.is_empty())
    print('链表的长度: ',singleLinkList.length())
    print('-----遍历单链表-----')
    singleLinkList.travel()
    print('-----查找-----')
    print(singleLinkList.search(20))
    print(singleLinkList.search(30))
    print('-----头部插入-----')

```



```

singleLinkList.add(1)
singleLinkList.add(2)
singleLinkList.add(3)
singleLinkList.travel()
print('-----尾部追加-----')
singleLinkList.append(10)
singleLinkList.append(20)
singleLinkList.append(30)
singleLinkList.travel()
print('链表的长度: ', singleLinkList.length())
print('-----指定位置插入-----')
singleLinkList.insert(2, 100)
singleLinkList.travel()
singleLinkList.insert(-1, 200)
singleLinkList.travel()
singleLinkList.insert(100, 300)
singleLinkList.travel()
print('-----删除节点-----')
singleLinkList.remove(100)
singleLinkList.travel()
singleLinkList.remove(200)
singleLinkList.travel()
singleLinkList.remove(300)
singleLinkList.travel()

```

链表与顺序表的对比

链表失去了顺序表随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大，但对存储空间的使用要相对灵活。

链表与顺序表的各种操作复杂度如下所示：

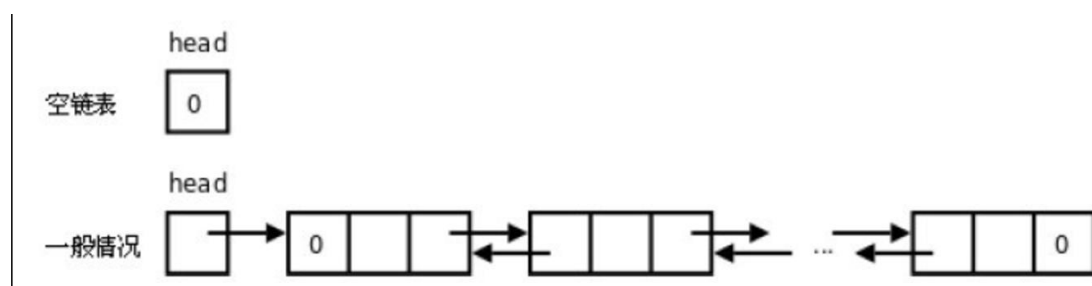
操作	链表	顺序表
访问元素	$O(n)$	$O(1)$
在头部插入/删除	$O(1)$	$O(n)$
在尾部插入/删除	$O(n)$	$O(1)$
在中间插入/删除	$O(n)$	$O(n)$

注意虽然表面看起来复杂度都是 $O(n)$ ，但是链表和顺序表在插入和删除时进行的是完

全不同的操作。链表的主要耗时操作是遍历查找，删除和插入操作本身的复杂度是 $O(1)$ 。顺序表查找很快，主要耗时的操作是拷贝覆盖。因为除了目标元素在尾部的特殊情况，顺序表进行插入和删除时需要对操作点之后的所有元素进行前后移位操作，只能通过拷贝和覆盖的方法进行。

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。



【示例】双向链表实现

```
class Node(object):
    """双向链表节点"""
    def __init__(self, item):
        self.item = item
        self.next = None
        self.prev = None

class DLinkedList(object):
    """双向链表"""
    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):
        """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
    print()

def add(self, item):
    """头部插入元素"""
    node = Node(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 的头节点的 prev 指向 node
        self.__head.prev = node
        # 将_head 指向 node
        self.__head = node

def append(self, item):
    """尾部插入元素"""
    node = Node(item)
    if self.is_empty():
        # 如果是空链表, 将_head 指向 node
        self.__head = node
    else:
        # 移动到链表尾部
        cur = self.__head
```

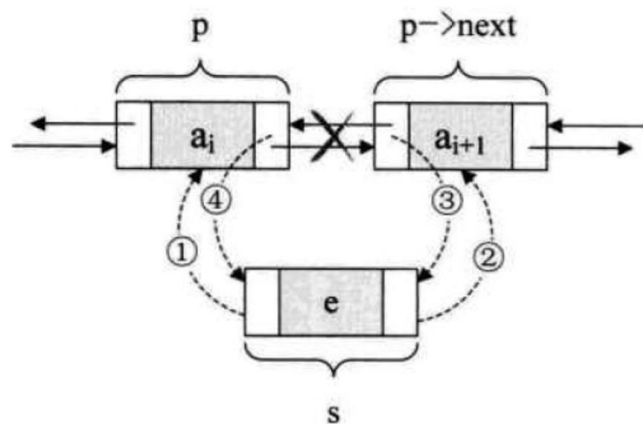
```

while cur.next != None:
    cur = cur.next
# 将尾节点 cur 的 next 指向 node
cur.next = node
# 将 node 的 prev 指向 cur
node.prev = cur

def search(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 False

```

【示例】指定位置插入



#指定位置插入

```

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 = Node(item)

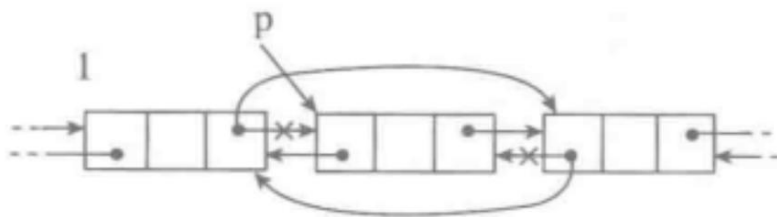
```

```

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 的下一个节点的 prev 指向 node
cur.next.prev = node
# 将 cur 的 next 指向 node
cur.next = node

```

【示例】删除节点



```

#删除节点
def remove(self,item):
    curNode=self.__head
    while curNode !=None:
        if curNode.elem == item:
            #判断是否是头节点
            if curNode == self.__head :#是头节点
                self.__head=curNode.next
            if curNode.next:#判断链表是否只有一个节点
                curNode.next.prev=None
        else:
            #删除
            curNode.prev.next=curNode.next

```

```
        if curNode.next:
            curNode.next.prev=curNode.prev

        break
    else:
        curNode=curNode.next
```

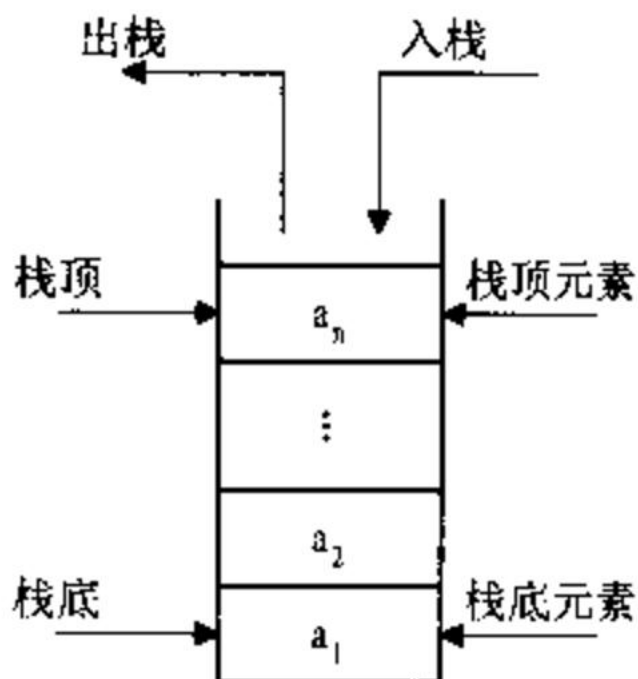
【示例】测试

```
if __name__ == '__main__':
    doubleLinkList=DoubleLinkList()
    doubleLinkList.add(11)
    doubleLinkList.add(22)
    doubleLinkList.add(33)
    doubleLinkList.travel()
    print('-----追加-----')
    doubleLinkList.append(100)
    doubleLinkList.append(200)
    doubleLinkList.append(300)
    doubleLinkList.travel()
    print('指定位置插入')
    doubleLinkList.insert(-1,44)
    doubleLinkList.travel()
    doubleLinkList.insert(100,400)
    doubleLinkList.travel()
    doubleLinkList.insert(2,1000)
    doubleLinkList.travel()
    print('-----删除节点-----')
    doubleLinkList.remove(44)
    doubleLinkList.travel()
    doubleLinkList.remove(1000)
    doubleLinkList.travel()
    doubleLinkList.remove(400)
    doubleLinkList.travel()
    print('链表的长度: ',doubleLinkList.length())
    print('查找节点 11',doubleLinkList.search(11))
    print('查找节点 111',doubleLinkList.search(111))
```

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。



栈结构实现

实现步骤：

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

【示例】栈结构的实现

```
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)

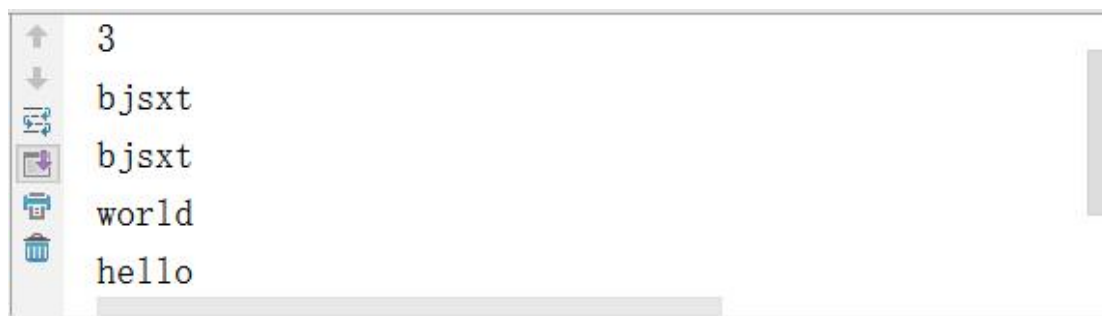
def pop(self):
    """弹出元素"""
    return self.items.pop()

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("bjsxt")
    print(stack.size())
    print(stack.peek())
    print(stack.pop())
    print(stack.pop())
    print(stack.pop())
```

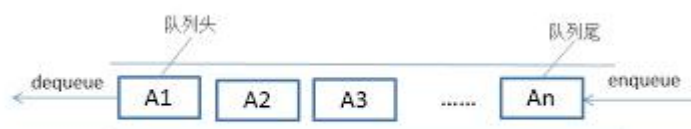
执行结果如图所示：



队列

队列（queue）是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。

队列是一种先进先出的（First In First Out）的线性表，简称 FIFO。允许插入的一端为队尾，允许删除的一端为队头。队列不允许在中间部位进行操作！假设队列是 $q = (a_1, a_2, \dots, a_n)$ ，那么 a_1 就是队头元素，而 a_n 是队尾元素。这样我们就可以删除时，总是从 a_1 开始，而插入时，总是在队列最后。这也比较符合我们通常生活中的习惯，排在第一个的优先出列，最后来的当然排在队伍最后。



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

【示例】队列的实现

```
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("bjsxt")
    print(q.size())
    print(q.dequeue())
    print (q.dequeue())
    print(q.dequeue())

```

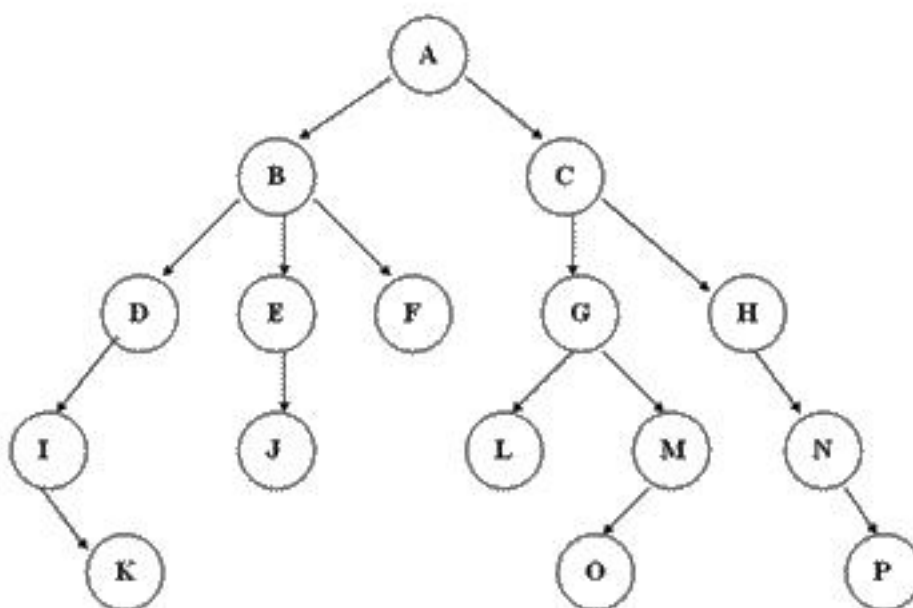
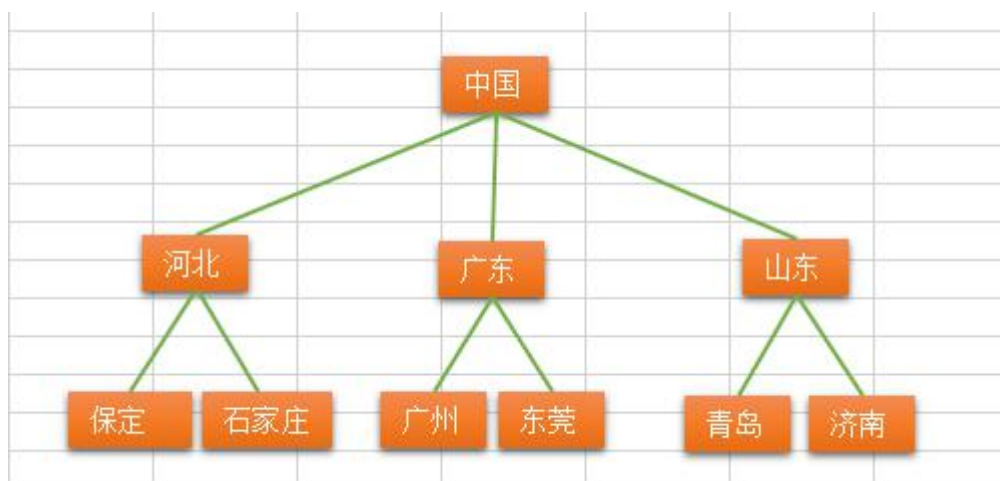
树与树的算法

树的概念

树（英语：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 ($n \geq 1$) 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：

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



树的术语

- (1) 节点的度：一个节点含有的子树的个数称为该节点的度；
- (2) 树的度：一棵树中，最大的节点的度称为树的度；
- (3) 叶节点或终端节点：度为零的节点；
- (4) 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；

- (5) 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；
- (6) 兄弟节点：具有相同父节点的节点互称为兄弟节点；
- (7) 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；
- (8) 树的高度或深度：树中节点的最大层次；
- (9) 堂兄弟节点：父节点在同一层的节点互为堂兄弟；
- (10) 节点的祖先：从根到该节点所经分支上的所有节点；
- (11) 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。
- (12) 森林：由 m ($m \geq 0$) 棵互不相交的树的集合称为森林；

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；

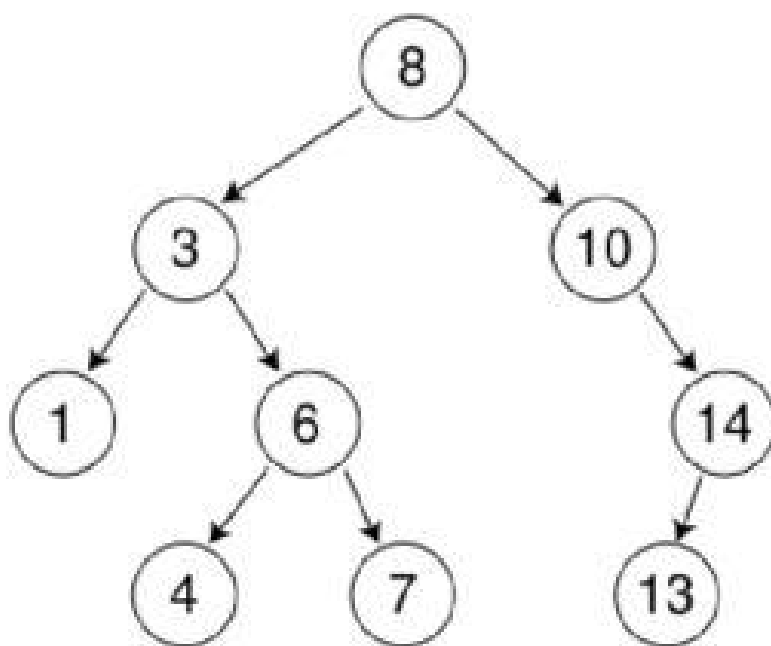
有序树：树中任意节点的子节点之间有顺序关系，这种树称为有序树；

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（英语：Binary Search Tree），也称二叉搜索树、有序二叉树）；

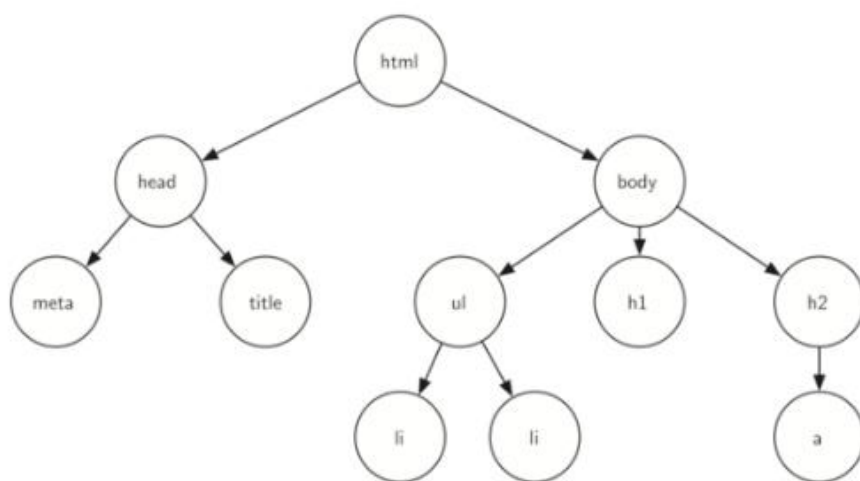


霍夫曼树（用于信息编码）：带权路径最短的二叉树称为哈夫曼树或最优二叉树；

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；

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 也是树结构



二叉树

二叉树的基本概念

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）。

二叉树的性质(特性)

性质 1: 在二叉树的第 i 层上至多有 $2^{(i-1)}$ 个节点 ($i>0$)

性质 2: 深度为 k 的二叉树至多有 $2^k - 1$ 个节点 ($k>0$)

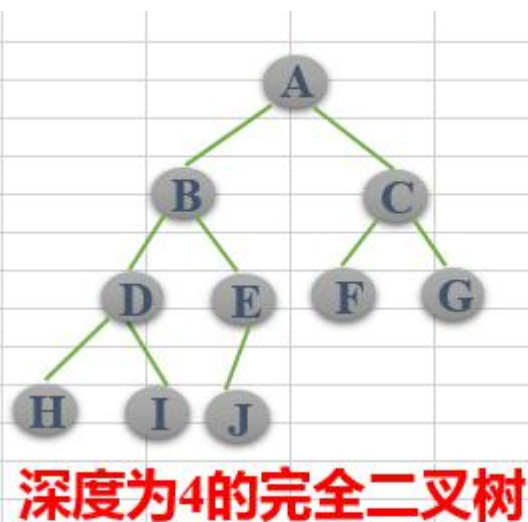
性质 3: 对于任意一棵二叉树，如果其叶节点数为 N_0 ，而度数为 2 的结点总数为 N_2 ，则 $N_0 = N_2 + 1$;

性质 4: 具有 n 个节点的完全二叉树的深度必为 $\log_2(n+1)$

性质 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 $2i$ ，其右孩子编号必为 $2i+1$ ；其双亲的编号必为 $i/2$ ($i=1$ 时为根,除外)

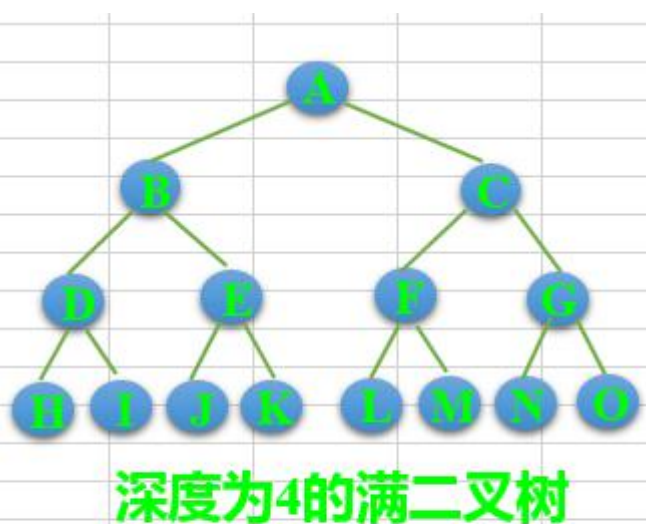
(1) 完全二叉树——若设二叉树的高度为 h ，除第 h 层外，其它各层 ($1 \sim h-1$) 的结点数都达到最大个数，第 h 层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二

叉树。



完全二叉树：
比如有n层
第1 - n-1层与满二叉树一样
第n层最后一个节点前边都挂满了节点

(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。



满二叉树: 每一层都挂满了节点

二叉树的节点及树的创建

【示例】通过使用 Node 类中定义三个属性，分别为 elem 本身的值，还有 lchild 左孩子和

rchild 右孩子

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

【示例】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(object):
    """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)
```

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。

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

【示例】先序遍历

```
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

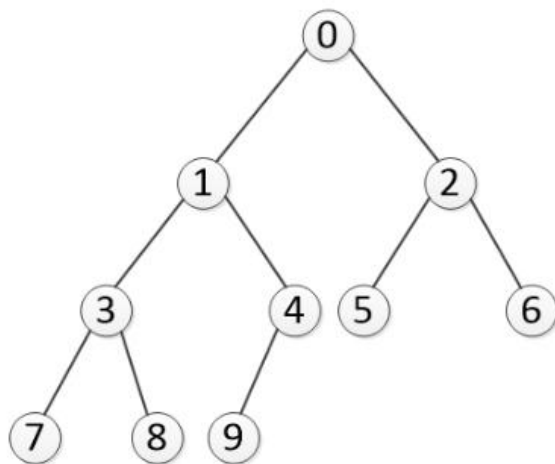
【示例】中序遍历

```
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，我们先递归使用后续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

【示例】后序遍历

```
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)
```

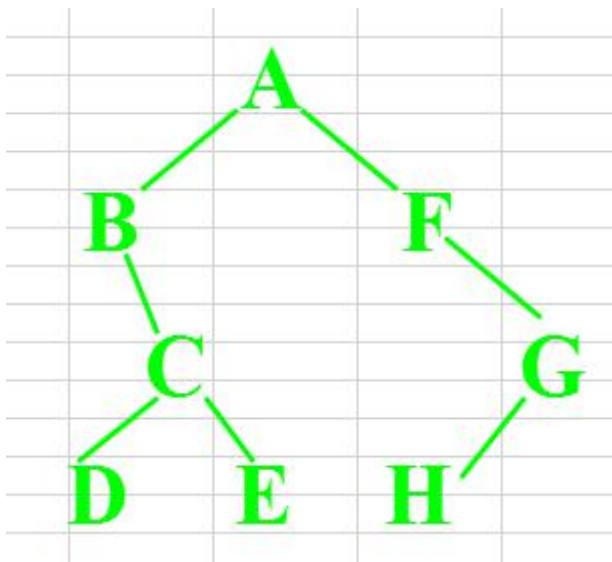


层次遍历: 0 1 2 3 4 5 6 7 8 9

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



遍历结构:

先序:a b c d e f g h

中序:b d c e a f h g

后序:d e c b h g f a

广度优先遍历(层次遍历)

从树的 root 开始，从上到下从左到右遍历整个树的节点。

【示例】广度优先遍历

```
def breadth_travel(self):
    """利用队列实现树的层次遍历"""
    if self.root == None:
        return
    queue = []
    queue.append(self.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)
```