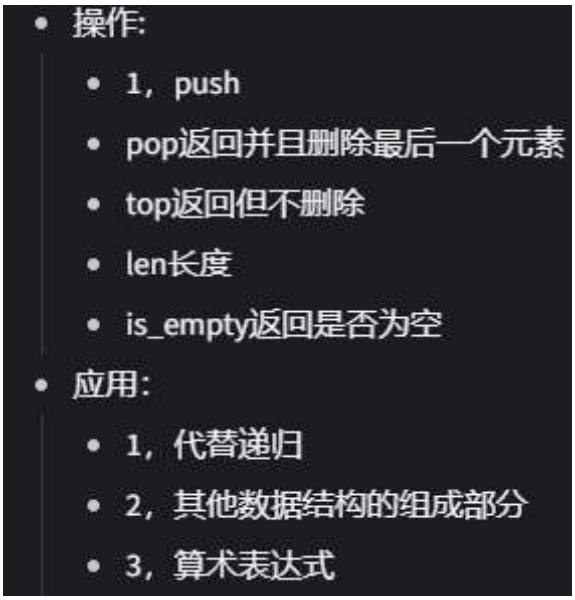
ADT抽象数据：

ADT包含：
1，数据的存储
2，数据的操作
3，与操作相关的错误条件

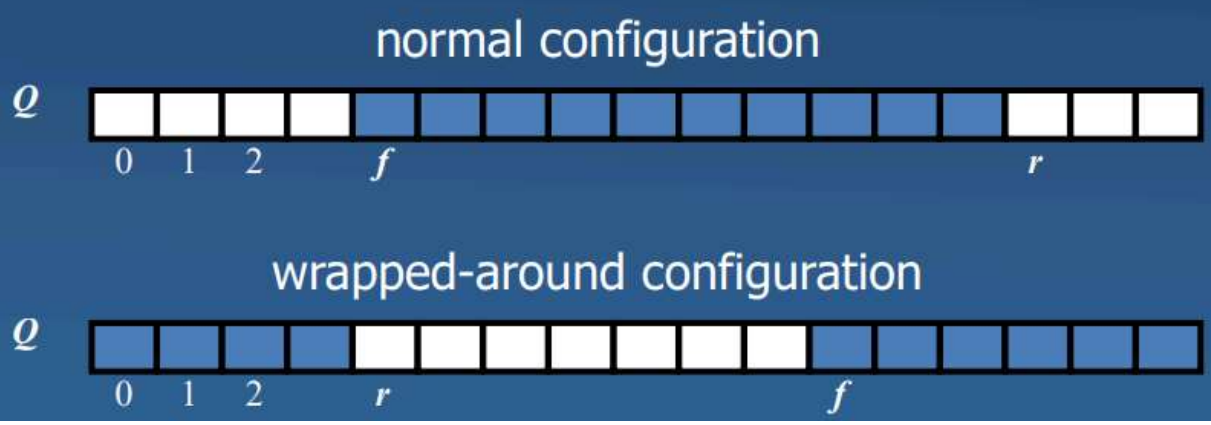
栈



队列

Array-based Queue

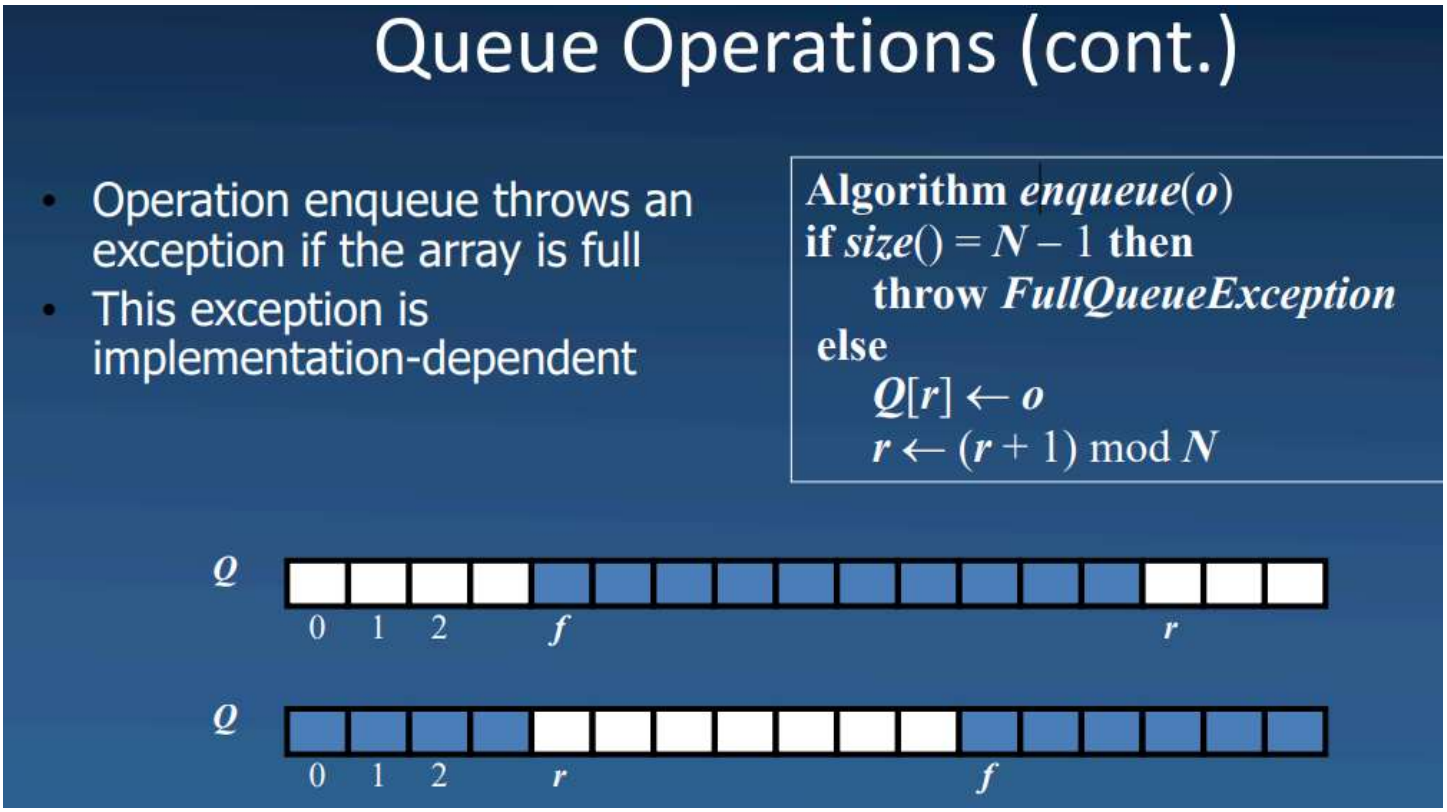
- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty



We use the modulo operator (remainder of division)

Algorithm *size*()
return $(N - f + r) \bmod N$

Algorithm *isEmpty*()
return $(f = r)$



链表 linked list

数组和队列的缺点：

除了在前面插入外，任何其他位置的插入都是昂贵的。
动态数组的长度可能比空间的数量更长。
操作的平摊边界在实时系统中可能不可接受。

链表提供了一个有用的替代方案：

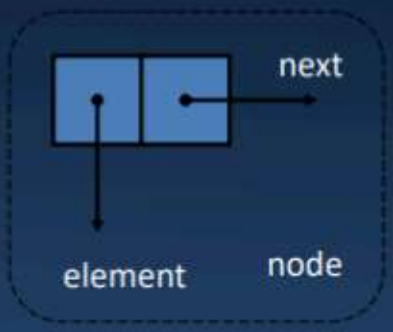
插入更容易，因为我们追踪前端和（通常）末尾。
长度是成比例的。
最坏情况的时间复杂度为 `O(n)`。

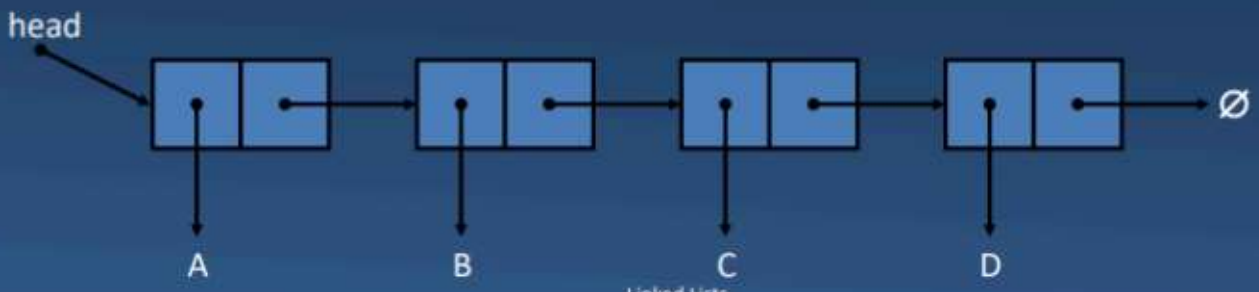
单链表结构：

每一个data由一个元素和指针构成

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





```
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None #初始化为null

class LinkList:
    def __init__(self):
        self.head=None

    def add_first(self,data):
        new_node = Node(data)
        new_node.next=self.head
        self.head=new_node
    def remove_first(self):
        if self.head:
            remove_data=self.head.data
            self.head=self.head.next
            return remove_data
        else:
            return None
    def add_list(self,data):
        new_node=Node(data)
        if not self.head: #如果没有东西的话，那就加上咯
            self.head=new_node
        else:
            current=self.head
            while current.next:
                current=current.next
            current.next=new_node

llist=LinkList()
llist.head=Node(1)
second=Node(2)
third=Node(3)

llist.head.next=second

second.next=third
```

在单向链表中从尾部移除是低效的：

- 没有一种常数时间的方法来更新尾部指向前一个节点。
- 我们需要知道末尾前面的元素，这并不容易做到。

作为链表的堆栈：

我们可以使用单向链表来实现堆栈。
栈顶元素存储在链表的第一个节点。
使用的空间为 O(n)，每个堆栈 ADT 操作的时间复杂度为 O(1)。

Linked-List Stack in Python

```
1  class LinkedStack:
2    """LIFO Stack implementation using a singly linked list for storage."""
3
4    #---------- nested _Node class ----------
5    class _Node:
6      """Lightweight, nonpublic class for storing a singly linked node."""
7      __slots__ = '_element', '_next'         # streamline memory usage
8
9      def __init__(self, element, next):      # initialize node's fields
10       self._element = element               # reference to user's element
11       self._next = next                     # reference to next node
12
13   #---------- stack methods ----------
14   def __init__(self):
15     """Create an empty stack."""
16     self._head = None                       # reference to the head node
17     self._size = 0                          # number of stack elements
18
19   def __len__(self):
20     """Return the number of elements in the stack."""
21     return self._size
22
```

```
23   def is_empty(self):
24     """Return True if the stack is empty."""
25     return self._size == 0
26
27   def push(self, e):
28     """Add element e to the top of the stack."""
29     self._head = self._Node(e, self._head)    # create and link a new node
30     self._size += 1
31
32   def top(self):
33     """Return (but do not remove) the element at the top of the stack.
34
35     Raise Empty exception if the stack is empty.
36     """
37     if self.is_empty():
38       raise Empty('Stack is empty')
39     return self._head._element                # top of stack is at head of list
```

```
40   def pop(self):
41     """Remove and return the element from the top of the stack (i.e., LIFO).
42
43     Raise Empty exception if the stack is empty.
44     """
45     if self.is_empty():
46       raise Empty('Stack is empty')
47     answer = self._head._element
48     self._head = self._head._next             # bypass the former top node
49     self._size -= 1
50     return answer
```

UNIVERSITY OF ABERDEEN

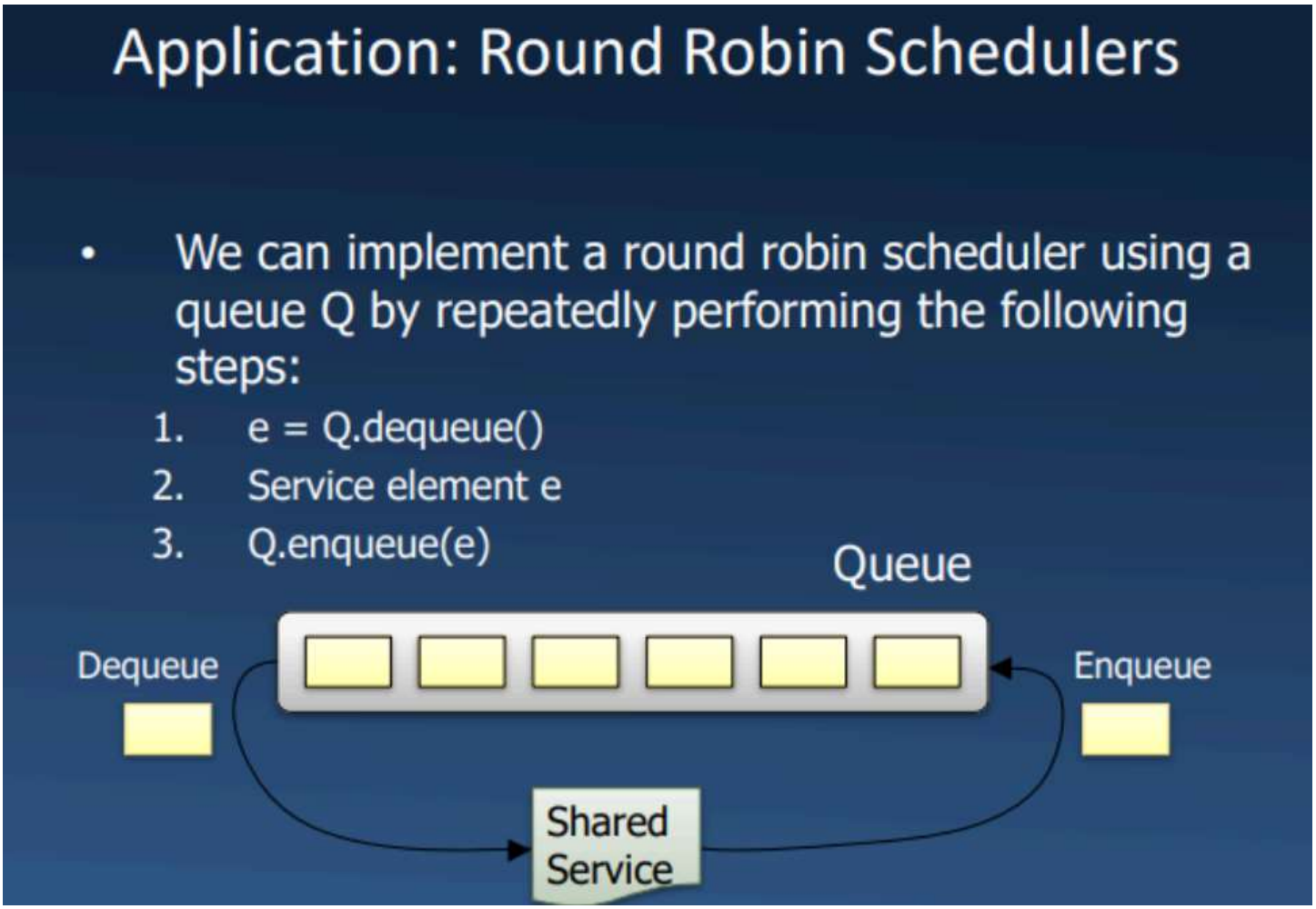
作为链表的队列：

我们可以使用单向链表来实现队列。
前端元素存储在第一个节点。
后端元素存储在最后一个节点。
使用的空间为 O(n)，每个队列 ADT 操作的时间复杂度为 O(1)。

Linked-List Queue in Python

```
1  class LinkedQueue:
2    """FIFO queue implementation using a singly linked list for storage."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a singly linked node."""
6      (omitted here; identical to that of LinkedStack._Node)
7
8    def __init__(self):
9      """Create an empty queue."""
10     self._head = None
11     self._tail = None
12     self._size = 0                           # number of queue elements
13
14   def __len__(self):
15     """Return the number of elements in the queue."""
16     return self._size
17
18   def is_empty(self):
19     """Return True if the queue is empty."""
20     return self._size == 0
21
22   def first(self):
23     """Return (but do not remove) the element at the front of the queue."""
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._head._element               # front aligned with head of list
```

```
27   def dequeue(self):
28     """Remove and return the first element of the queue (i.e., FIFO).
29
30     Raise Empty exception if the queue is empty.
31     """
32     if self.is_empty():
33       raise Empty('Queue is empty')
34     answer = self._head._element
35     self._head = self._head._next
36     self._size -= 1
37     if self.is_empty():                      # special case as queue is empty
38       self._tail = None                      # removed head had been the tail
39     return answer
40
41   def enqueue(self, e):
42     """Add an element to the back of queue."""
43     newest = self._Node(e, None)             # node will be new tail node
44     if self.is_empty():
45       self._head = newest                    # special case: previously empty
46     else:
47       self._tail._next = newest
48     self._tail = newest                      # update reference to tail node
49     self._size += 1
```

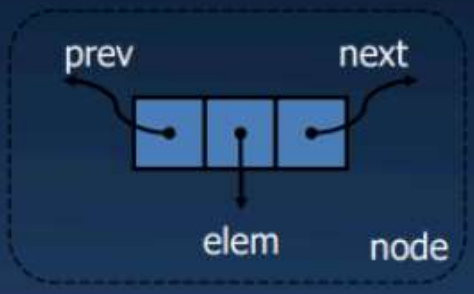


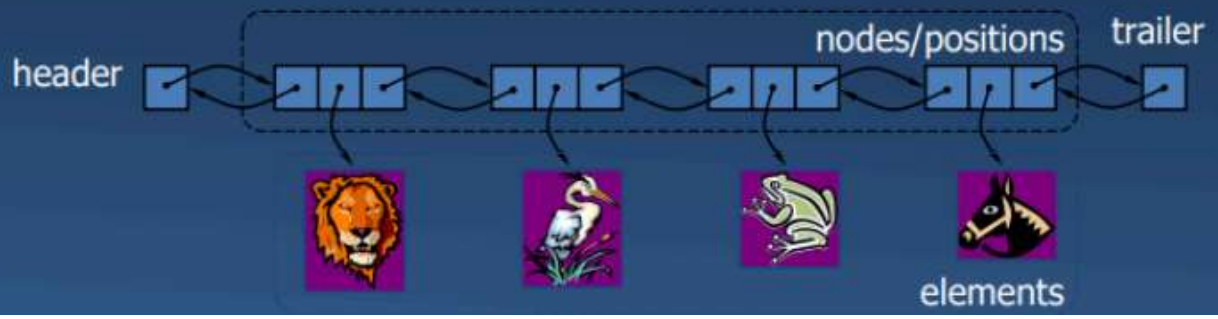
双链表结构 double linked list

双向链表的构成：
前指针，数据，后指针

Doubly Linked List

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





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


class DoublyLinkedList:
    def __init__(self):
        # 通常使用头部指针 head 来表示链表的起始节点，而尾部指针 tail 表示链表的最后一个节点。
        self.head = None
        self.tail = None

    def add_last(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node

    def remove_at(self, position):
        if position < 0:
            return False
        if position == 0:
            return self.remove_first()

        current = self.head
        count = 0
        while current and count < position:
            current = current.next
            count += 1

        if not current:
            return False

        if current == self.tail:
            return self.remove_last()

        current.prev.next = current.next
        current.next.prev = current.prev
```

```
        return True
```

优点：空间占有率：O(n),每一个O(1),运行时间O(1)

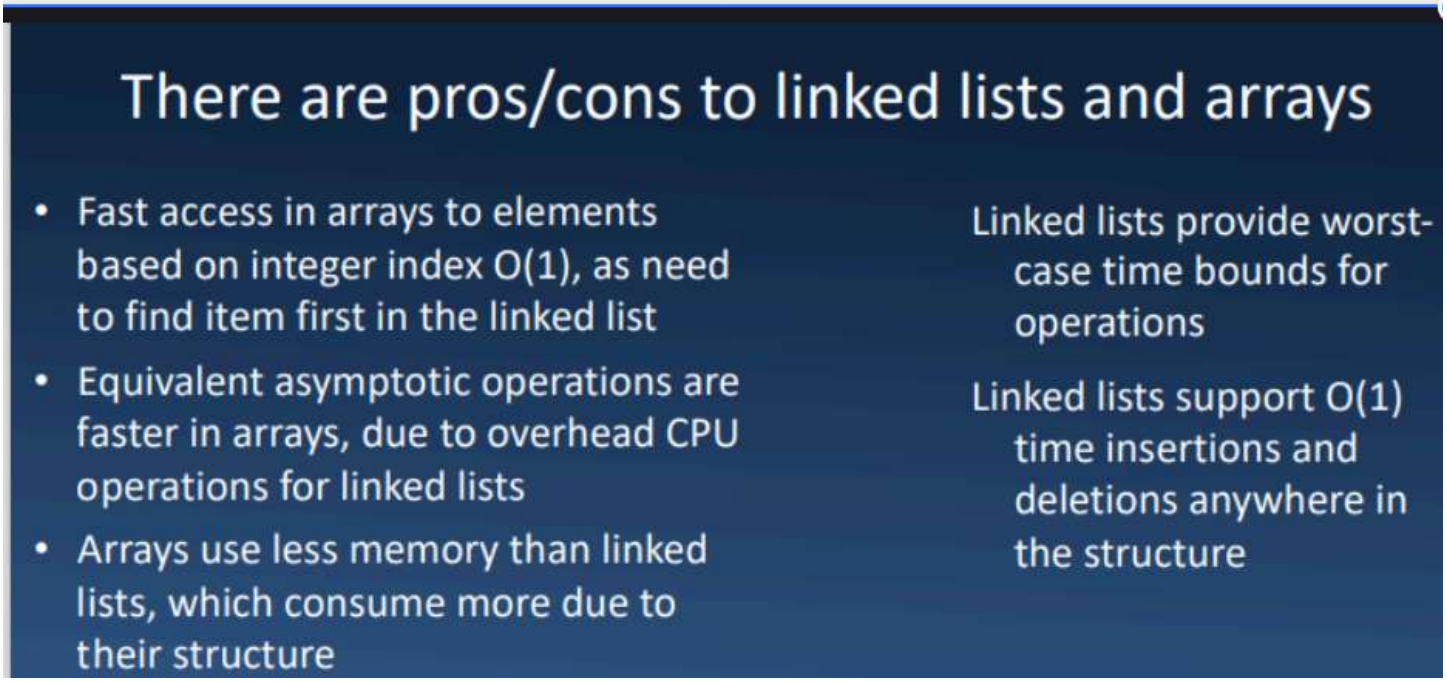
位置列表：能够标识元素的位置

链表和数组都有其优缺点：

- 数组可以通过整数索引 O(1) 快速访问元素，而链表需要首先找到元素。
- 等效的渐近操作在数组中更快，因为链表需要额外的 CPU 操作。
- 数组使用的内存比链表少，因为链表的结构会消耗更多内存。

链表为操作提供了最坏情况的时间界限：

链表支持在结构中任何位置进行 O(1) 时间的插入和删除操作。



合理地翻译成中文并和我解释：Positional List – identifying an element in a sequence
• To provide for a general abstraction of a sequence of elements with the ability
to identify the location of an element, we define a positional list ADT.
• A position acts as a marker or token within the broader positional list.
• A position p is unaffected by changes elsewhere in a list; the only way in which
a position becomes invalid is if an explicit command is issued to delete it.
• A position instance is a simple object, supporting only the following method:
– p.element( ): Return the element stored at position p.
Think of a cursor in your editor:
We just need to identify the
character under it

ChatGPT
位置列表 - 识别序列中的元素
• 为了提供对元素序列的一般抽象，并能够识别元素的位置，我们定义了一个位置列表ADT。
• 位置充当更广泛的位置列表中的标记或令牌。
• 位置p不受列表其他位置变化的影响；位置变得无效的唯一方式是发出显式命令将其删除。
• 位置实例是一个简单的对象，仅支持以下方法：
– p.element()：返回存储在位置p处的元素。
想象一下编辑器中的光标：
我们只需要识别
它下面的字符。

这段话介绍了位置列表（Positional List）ADT，它是一种用于处理元素序列的抽象数据类型。每个位置在序列中充当标记或令牌，具有独立于列表其他部分的特性。位置只有在显式删除命令发出时才会失效。位置实例是简单的对象，仅支持获取存储在位置处的元素。通过比喻编辑器中的光标，可以更容易地理解位置列表的概念，就像光标标识编辑器中的特定位置一样。

树 Tree

一些名词：

数的组成部分：

Tree Terminology

Root: node without parent (A)

Internal node: node with at least one child (A, B, C, F)

Siblings: Child nodes at the same depth (B, C, D)

External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)

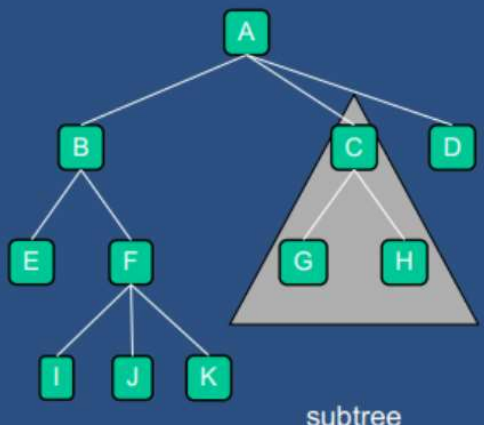
Ancestors of a node: parent, grandparent, grand-grandparent, etc.

Depth of a node: number of ancestors

Height of a 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



There are edges and paths in trees

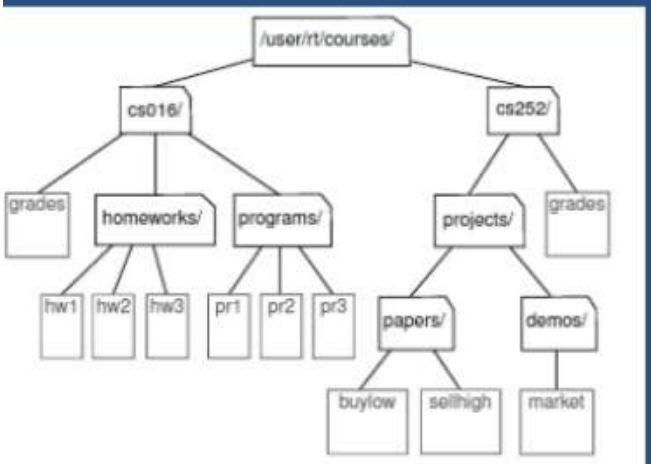


Figure 8.3: Tree representing a portion of a file system.

Edge = a pair of nodes where one is parent of the other.

Path = sequence of nodes such as cs252/projects/demos/market

树的遍历方式：

preorder Traversal前序遍历：

先访问其节点的数据，然后再访问后代的节点

postorder traversal后序遍历：

先访问其后代的节点，再访问它的数据

Breadth-First Traversal广度优先遍历：

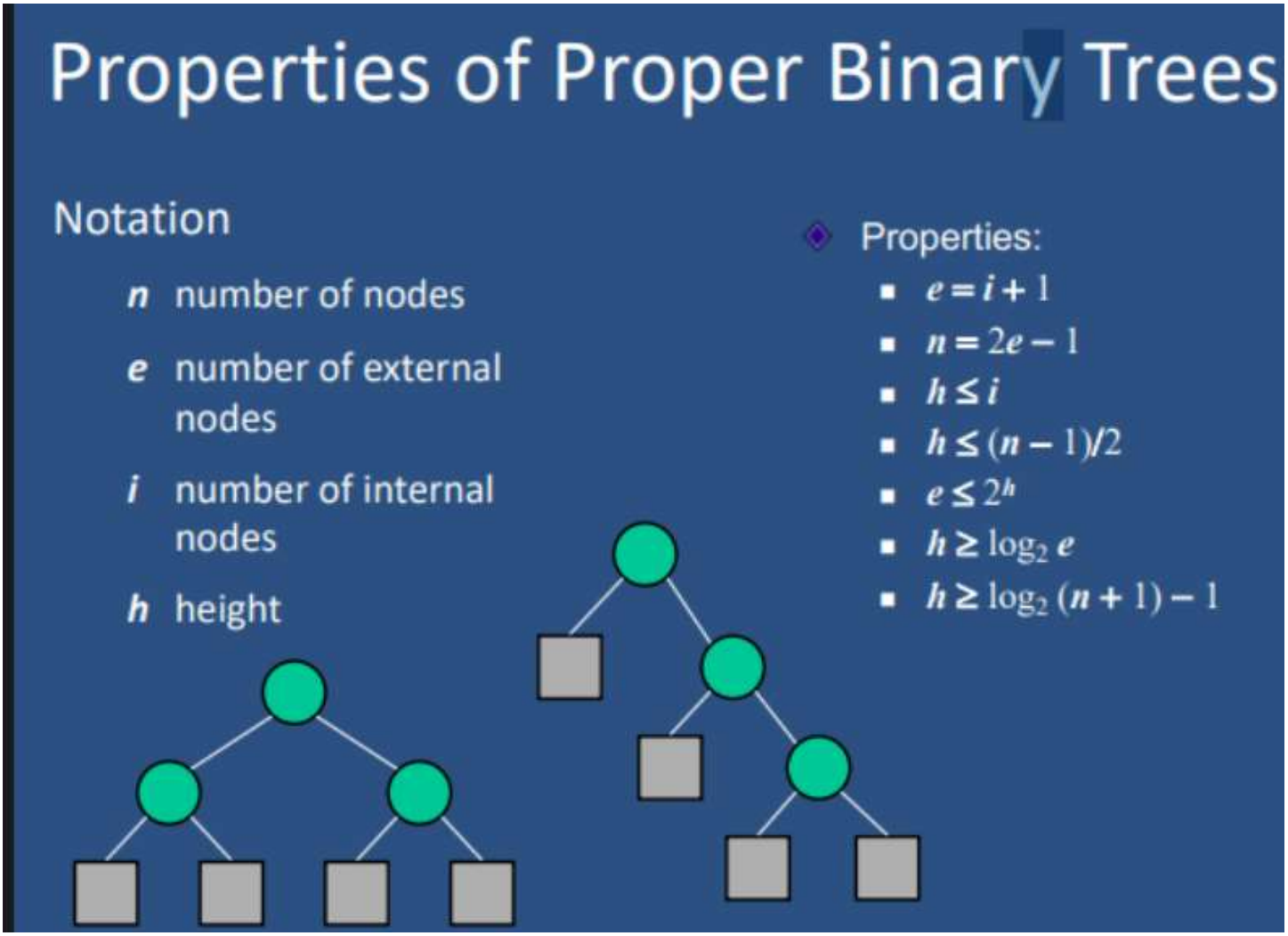
在广度优先遍历中，每个级别的所有节点都先于其后代访问

二分树 binary trees

属性:

1，每一个内部的节点最有2个孩子

2，每个节点的孩子是有顺序的(ordered pair)



递归定义：Alternative recursive definition: a binary tree is either n a tree consisting of a single node, or n a tree whose root has an ordered pair of children, each of which is a binary tree

替代递归定义：二叉树要么是 n 个由单个节点组成的树，要么是 n 个根具有一对有序子节点的树，每个子节点都是二叉树

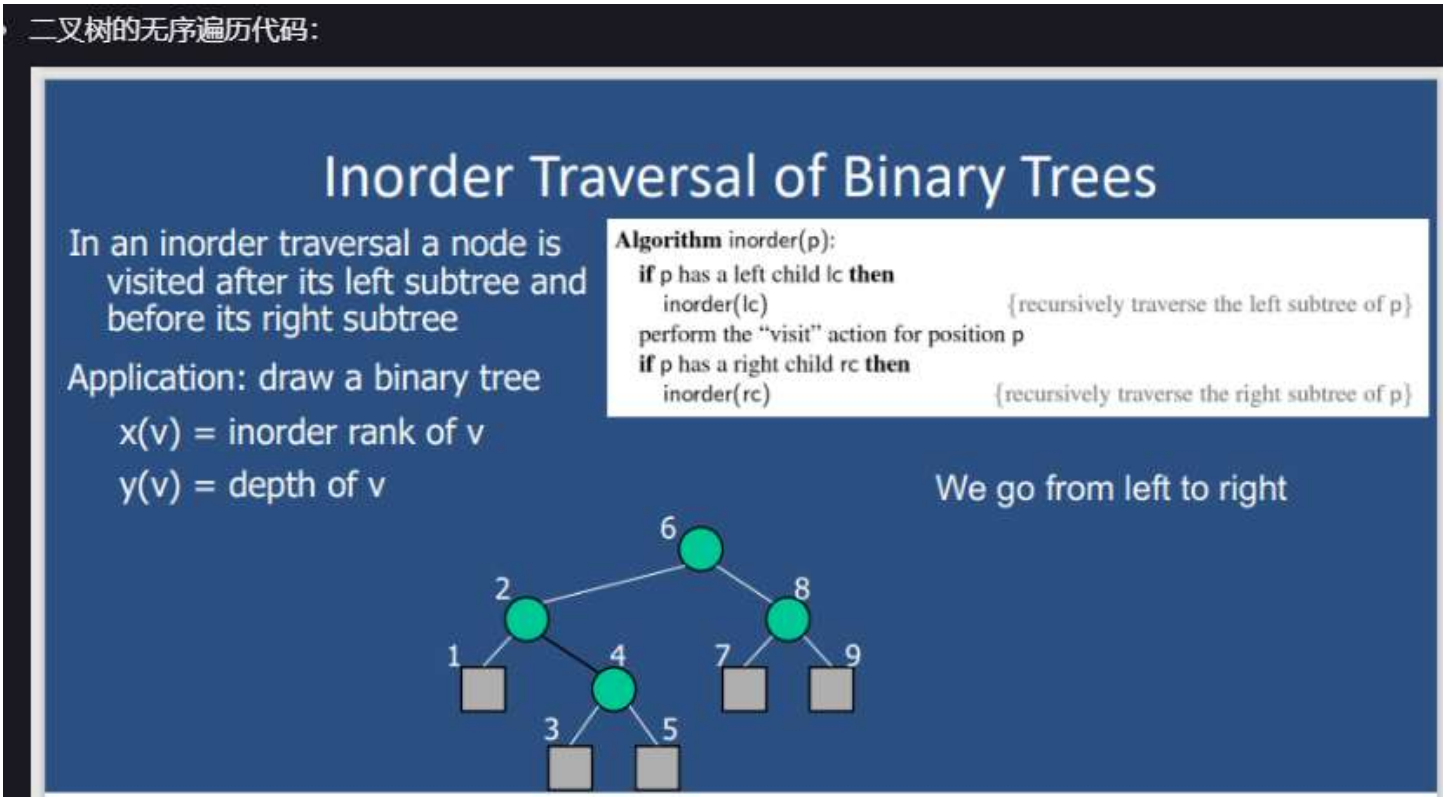
应用:

1,算术表达树：只有叶子代表数字，其他都代表着运算符

2，决策树:

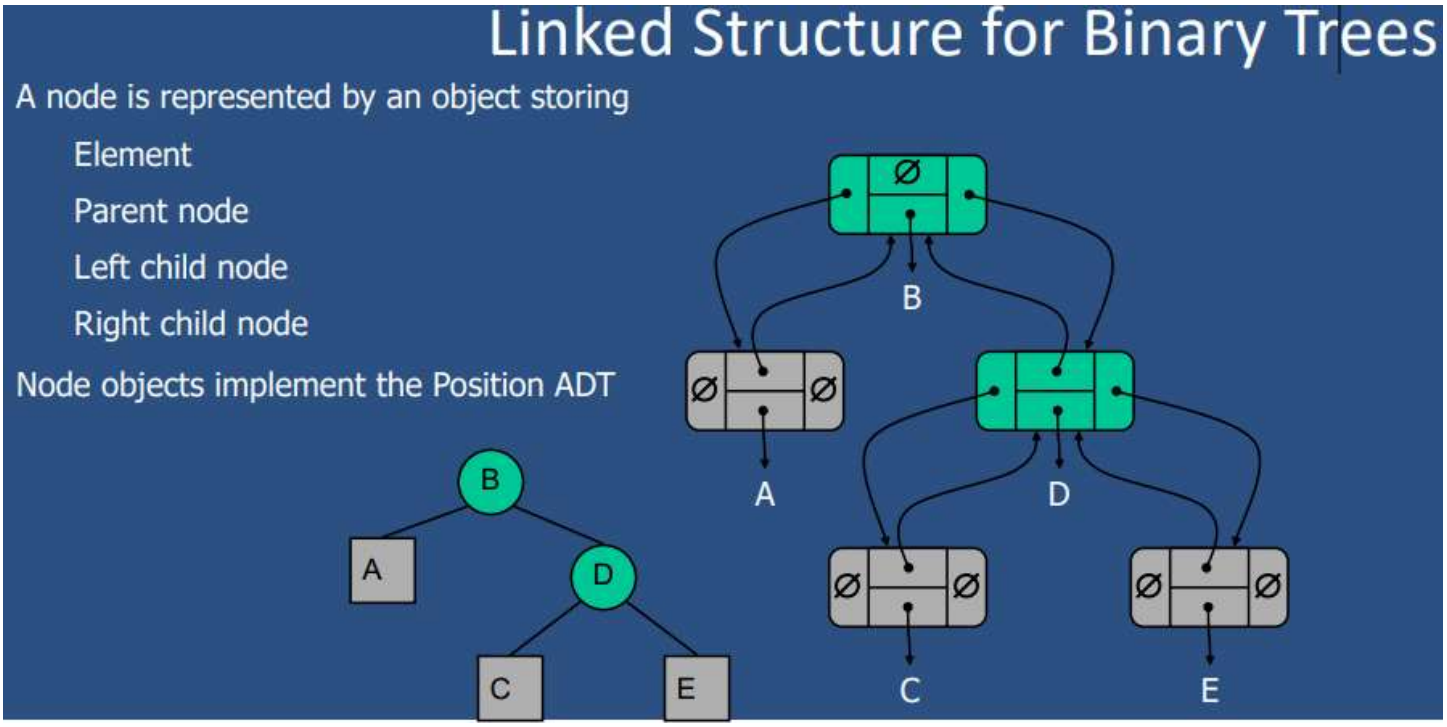
二叉树的Inorder Traversal of Binary Trees中序遍历

节点在其左子树之后，右子树之前被访问



二叉树的实现：

1，ADT的实现：



2，数组的实现：

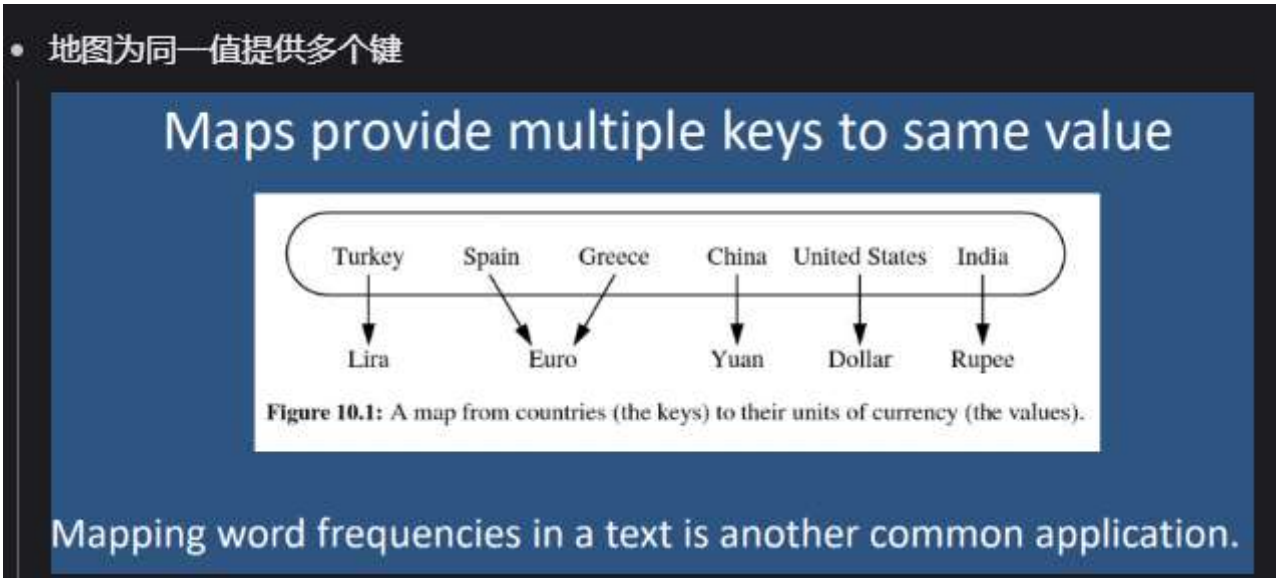
左孩子=2*父节点+1

右孩子=2*父节点+2

Map

可搜索的项的集合，key-value的形式

多个item不可能拥有相同的key,但是可以为同一个value提供同一个key

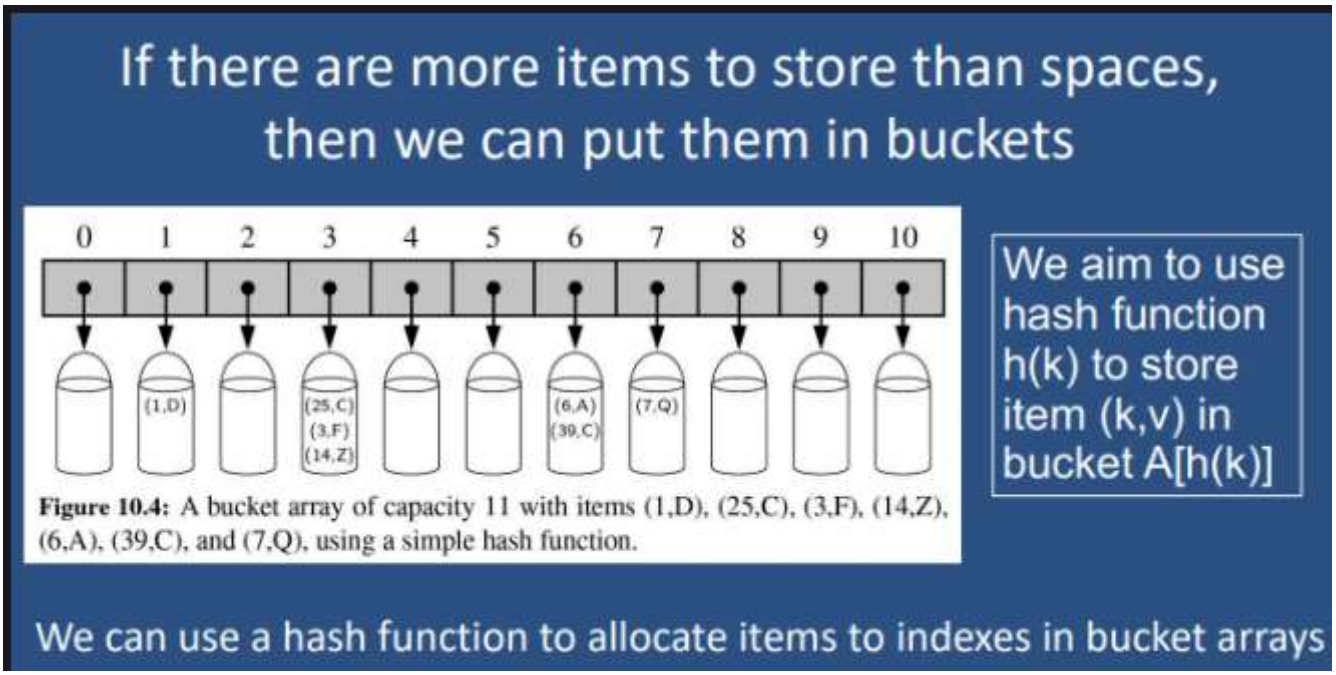


字典dict

在这里，我们在专门讨论 Python 的 dict 类时使用术语“dictionary”，在讨论抽象数据类型的更一般概念时使用术语“map”。

哈希表

物品比存储的空间还要多：使用桶



我们可以使用哈希函数将项目分配给存储桶数组中的索引

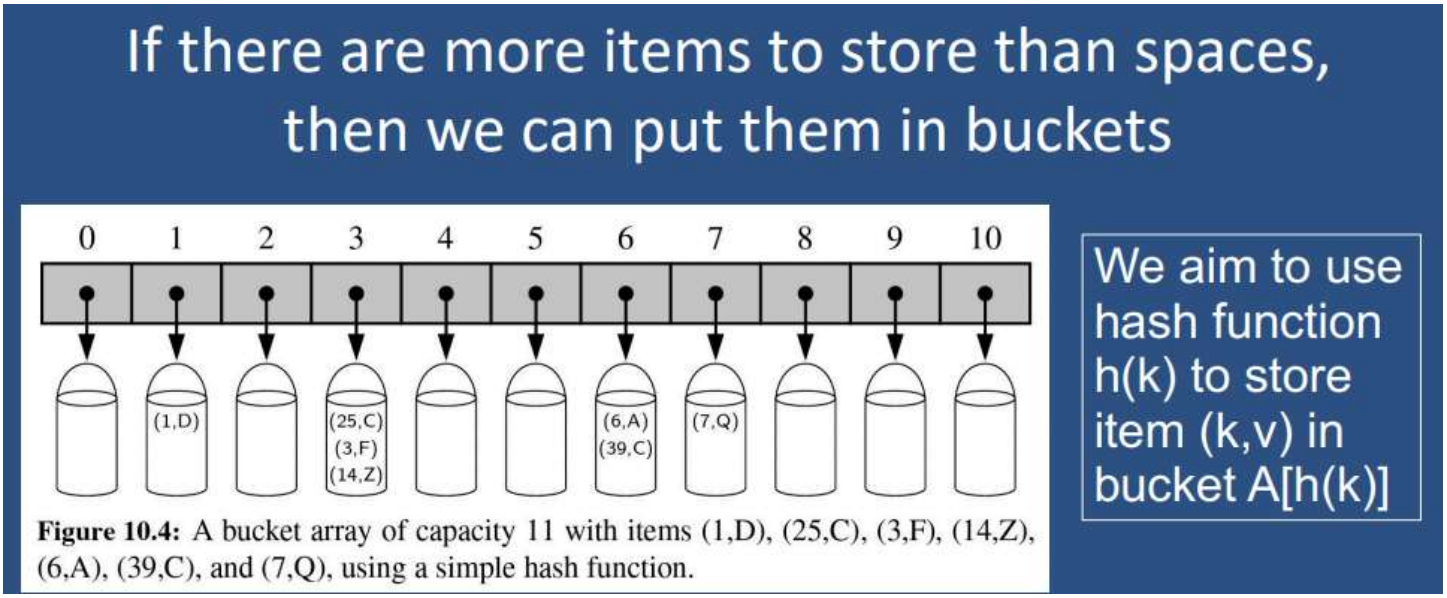
哈希函数：

哈希表是实现映射的最实用的数据结构之一。
Python的字典被实现为哈希表。

1，hash code将一个key映射到一个整数上

2，compression function:将哈希代码映射到存储桶数组的索引 [0， N-1] 范围内的整数

We can use a hash function to allocate items to indexes in bucket arrays我们可以使用哈希函数将项目分配给存储桶数组中的索引



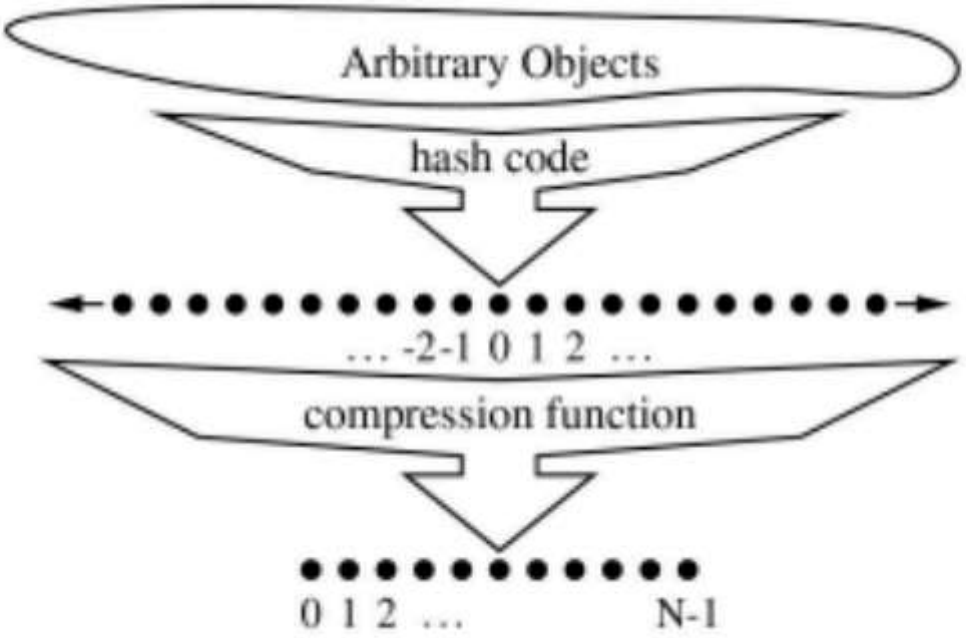


Figure 10.5: Two parts of a hash function: a hash code and a compression function.

- eg:
  - 示例：h（x）= x mod N 是整数键的哈希函数
  - 整数 h（x）称为键 x 的哈希值 给定键类型的哈希表由 哈希函数 h，大小为 N 的数组（称为表）组成
  - 当实现带有哈希表的映射时，目标是存储项目 （k，o）在索引 i = h（k）

Example:

$h(x) = x \bmod N$

is a hash function for integer keys

The integer $h(x)$ is called the hash value of key x

A hash table for a given key type consists of

Hash function h

Array (called table) of size N

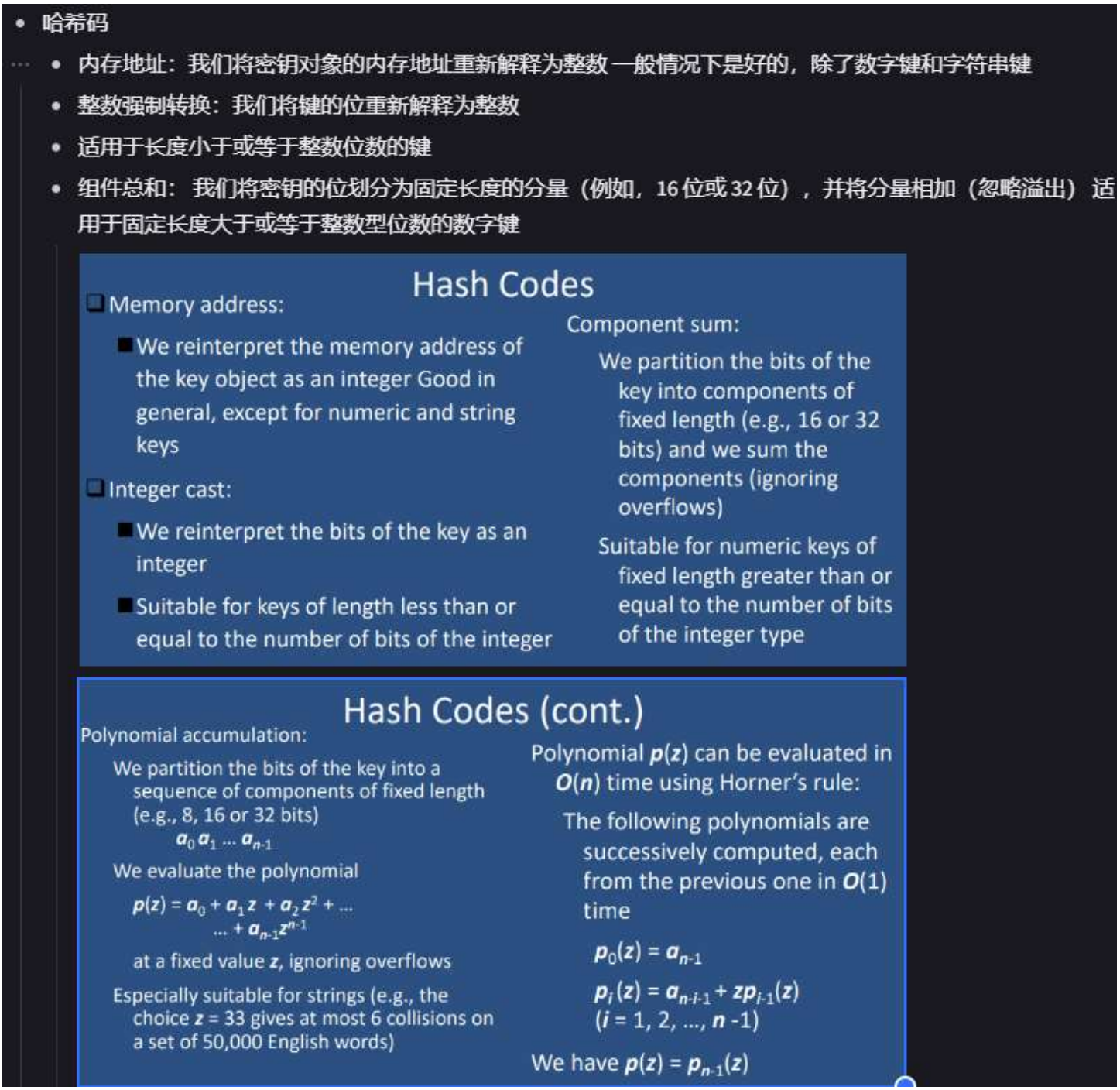
When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

- 哈希函数的意义：
  - 哈希函数旨在随机分散键 哈希函数通常被指定为两个函数的组合：
  - 哈希码： h1：键->整数
  - 压缩函数： h2： 整数 -> [0， N - 1]
  - 首先应用哈希码，然后对结果应用压缩函数，即 h（x）= h2（h1（x））哈希函数的目标是以一种看似随机的方式“分散”键

哈希函数的意义：

哈希函数旨在随机分散键 哈希函数通常被指定为两个函数的组合：
哈希码： h1：键->整数
压缩函数： h2： 整数 -> [0， N - 1]
首先应用哈希码，然后对结果应用压缩函数，即 h（x） = h2（h1（x））希函数的目标是以一种看

- 哈希码
  - 内存地址：我们将密钥对象的内存地址重新解释为整数 一般情况下是好的，除了数字键和字符串键
  - 整数强制转换：我们将键的位重新解释为整数
  - 适用于长度小于或等于整数位数的键
  - 组件总和：我们将密钥的位划分为固定长度的分量（例如，16 位或 32 位），并将分量相加（忽略溢出）适用于固定长度大于或等于整数型位数的数字键



- 压缩函数-最小冲突

Compression Functions minimize collisions

Division:

- $y \bmod N$
- The size N of the bucket array is usually chosen to be a prime
- The reason has to do with number theory and is beyond the scope of this course

Multiply, Add and Divide (MAD):

- $[(ay + b) \bmod p] \bmod N$
- With integer y, p is a prime number larger than N. a and b are nonnegative integers chosen at random such trhat
  - $a \bmod N \neq 0$
- Otherwise, every integer would map to the same value b

冲突处理：冲突处理是必要的 当不同的元素映射到同一个单元格时，会发生冲突 单独链接：让表中的每个单元格指向映射到那里的条目的链接列表 单独链接很简单，但需要表外的额外内存

Collision Handling is necessary

Collisions occur when different elements are mapped to the same cell

Separate Chaining: let each cell in the table point to a linked list of entries that map there

Separate chaining is simple, but requires additional memory outside the table

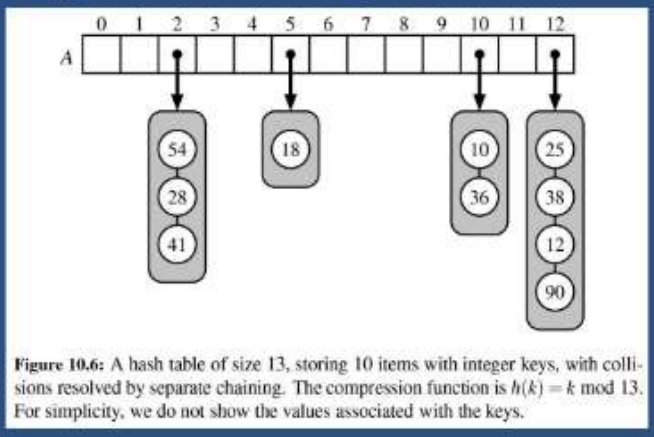
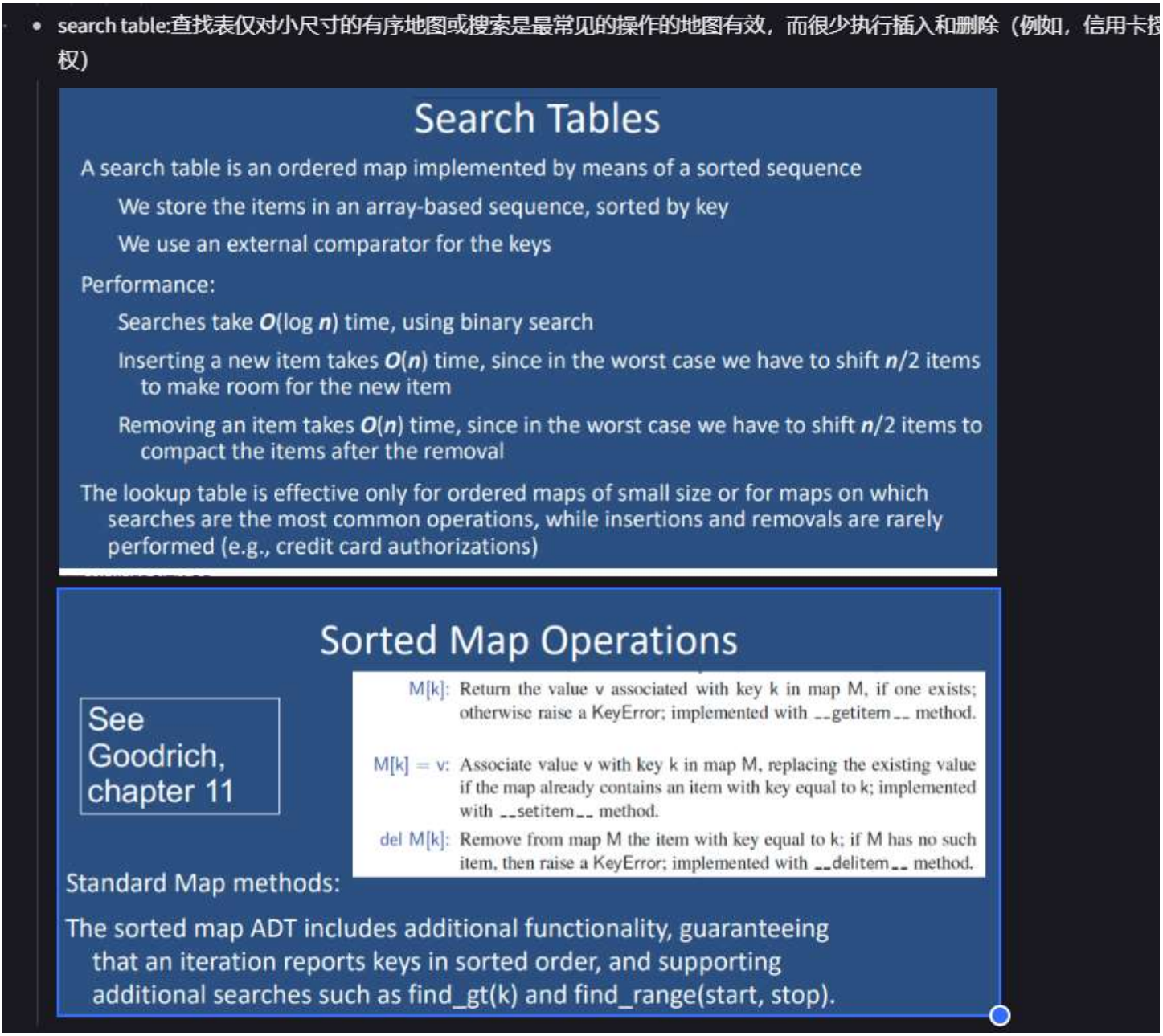


Figure 10.6: A hash table of size 13, storing 10 items with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

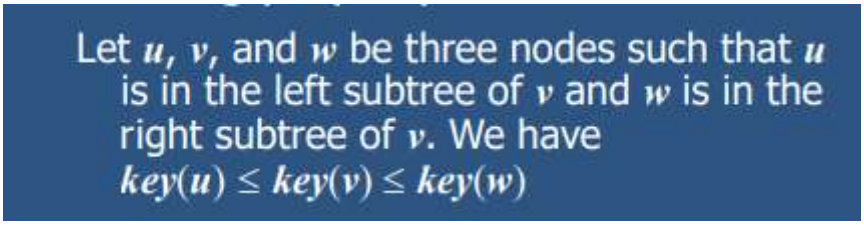
- search table:查找表仅对小尺寸的有序地图或搜索是最常见的操作的地图有效，而很少执行插入和删除（例如，信用卡授权）

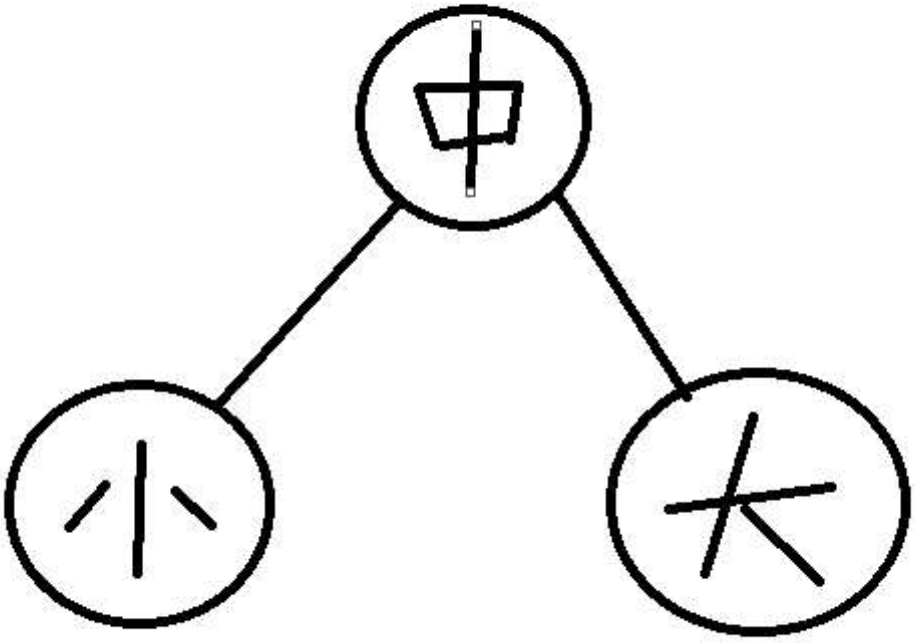


二叉搜索树：

搜索表：A search table is an ordered map implemented by means of a sorted sequence搜索表是通过排序序列实现的有序映射

Binary Search Trees：





二叉搜索树的的中序遍历：

可以以递增的顺序访问键

搜索：

Search

To search for a key ***k***, we trace a downward path starting at the root

The next node visited depends on the comparison of ***k*** with the key of the current node

If we reach a leaf, the key is not found

```
Algorithm TreeSearch(T, p, k):
    if k == p.key() then
        return p                                        {successful search}
    else if k < p.key() and T.left(p) is not None then
        return TreeSearch(T, T.left(p), k)              {recur on left subtree}
    else if k > p.key() and T.right(p) is not None then
        return TreeSearch(T, T.right(p), k)             {recur on right subtree}
    return p                                            {unsuccessful search}
```

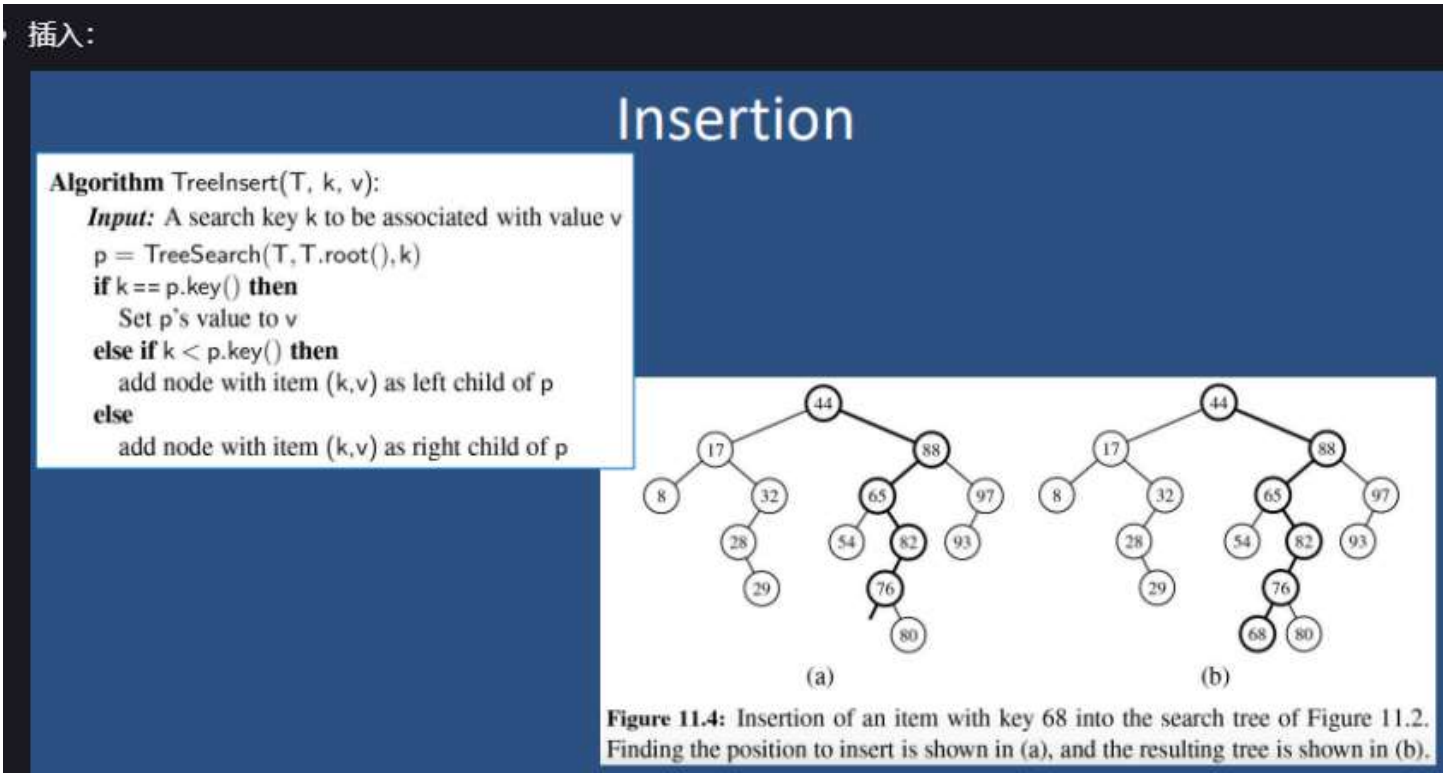
```
def search(root,key):

    # Base Cases: root is null or key is present at root
    if root is None or root.val == key:
        return root

    # Key is greater than root's key
    if root.val < key:
        return search(root.right,key)

    # Key is smaller than root's key
    return search(root.left,key)
```

插入：



```
def insert(root, key):
    if root is None:
        return Node(key)

    prev = None
    temp = root

    while temp:
        if temp.val > key:
            prev = temp
            temp = temp.left
        elif temp.val < key:
            prev = temp
            temp = temp.right

    if prev.val > key:
        prev.left = Node(key)
    else :
        prev.right = Node(key)


    return root
```

删除：