数据结构

Python Reference

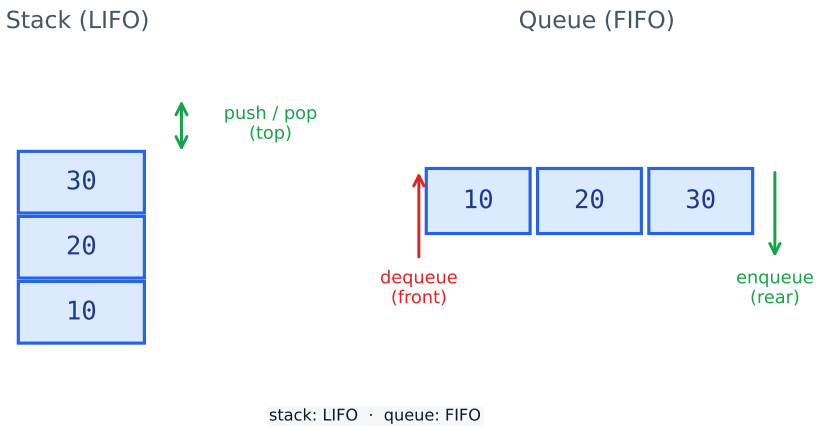
抽象数据类型（ADT）

抽象数据类型 (abstract data type,ADT) 描述一些数据以及对它的操作，与它如何实现分开。你通过它的操作来使用它，而不是通过它内部的存储方式。

```
# A stack ADT, built on a list
s = []
s.append(1)        # add
s.append(2)
print(s.pop())     # 2  (remove the most recent)
```

栈

栈 (stack) 是后进先出 (LIFO)。你把元素压入 (push) 栈顶，也从栈顶弹出 (pop)。



stack: LIFO · queue: FIFO

栈从顶部移除 *(后进先出)*；队列从前端移除 *(先进先出)*

```
stack = []
stack.append("a")
stack.append("b")
print(stack.pop())   # b
print(stack.pop())   # a
```

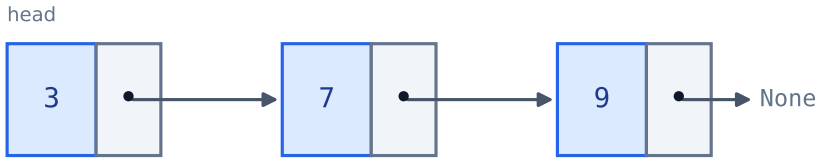

队列

队列 (queue) 是先进先出 (FIFO)。你在队尾入队 (enqueue)，从队首出队 (dequeue)。

```
queue = []
queue.append("a")     # enqueue
queue.append("b")
print(queue.pop(0))   # a  (dequeue the front)
print(queue.pop(0))   # b
```

链表

链表 (linked list) 是一串节点 (node)。每个节点保存数据和一个指向下一个节点的指针 (pointer)；最后一个指向 `None`。



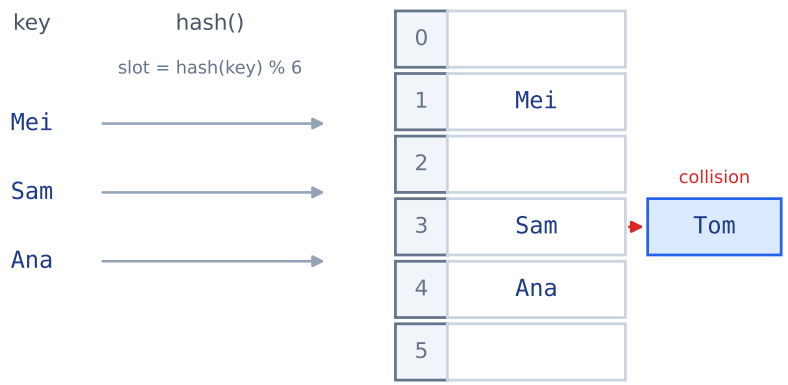
linked list: node → next · O(1) insert at head

链表：每个节点保存数据和指向下一个节点的指针，最后指向 *None*

```
n3 = {"data": 3, "next": None}
n2 = {"data": 2, "next": n3}
n1 = {"data": 1, "next": n2}
node = n1
while node is not None:      # traverse to the end
    print(node["data"])
    node = node["next"]
# 1 2 3
```

哈希表

哈希表 (hash table) 用哈希函数 (hash function) 把键映射到一个槽。两个键可能落入同一个槽 —— 这叫冲突 (collision)。Python 的 `dict` 就是哈希表，所以查找很快。

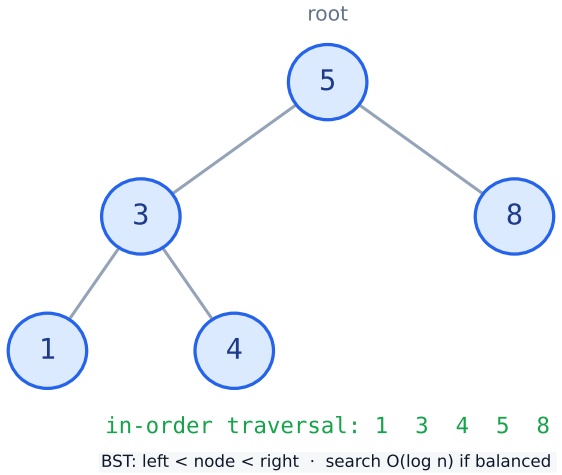


哈希函数把每个键映射到一个槽; 两个键可能落入同一个槽 (冲突)

```
table = {}
table["Mei"] = 88
table["Sam"] = 71
print(table["Mei"])   # 88  (fast lookup by key)
```

二叉搜索树

二叉搜索树 (binary search tree,BST) 保持有序: 每个左孩子都比它的节点小, 每个右孩子都比它大。查找一直很快。



二叉搜索树: 较小的值放在左边, 较大的值放在右边

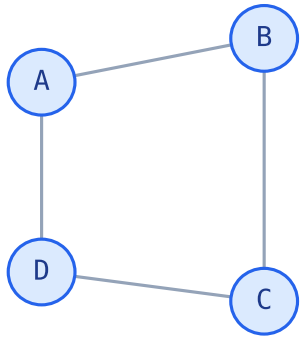
```
def insert(root, val):
    if root is None:
        return {"val": val, "left": None, "right": None}
    if val < root["val"]:
        root["left"] = insert(root["left"], val)
    else:
        root["right"] = insert(root["right"], val)
    return root

def inorder(root):
    if root is None:
        return []
    return inorder(root["left"]) + [root["val"]] + inorder(root["right"])

tree = None
for v in [5, 3, 8, 1, 4]:
    tree = insert(tree, v)
print(inorder(tree))   # [1, 3, 4, 5, 8]
```

图

图 (graph) 是一组由边 (edge) 连接的顶点 (vertex)。邻接表 (adjacency list)—— 一个 “ 邻居列表的字典 ”—— 是常见的存储方式。




```
graph = {
  "A": ["B", "D"],
  "B": ["A", "C"],
  "C": ["B", "D"],
  "D": ["A", "C"],
}
```

graph: vertices + edges · BFS/DFS traverse

由顶点和边组成的图, 以及它的邻接表形式

```
graph = {"A": ["B", "D"], "B": ["A", "C"], "C": ["B", "D"], "D": ["A",
↪  "C"]}
for vertex in graph:
    print(vertex, "->", graph[vertex])
# A -> ['B', 'D']  (and so on for B, C, D)
```

常见错误

- 栈 (stack) 是后进先出, 队列 (queue) 是先进先出, 别搞混。
- 在 pop 或出队之前, 先检查结构是不是空的。
- 在链表 (linked list) 中, 弄丢 `head` 指针就等于弄丢了整条链表。