

Data structures

Python Reference

Abstract Data Types (ADTs)

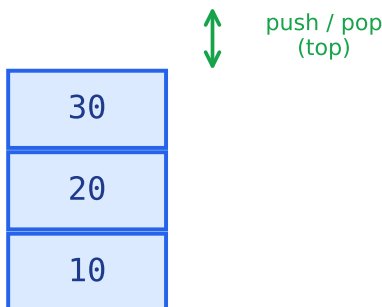
An abstract data type (ADT) describes some data plus the operations on it, separate from how it is built. You use it through its operations, not through its inner storage.

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

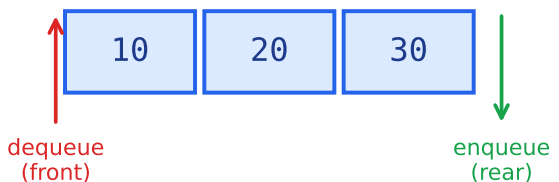
Stacks

A stack is last-in, first-out (LIFO). You push onto the top and pop from the top.

Stack (LIFO)



Queue (FIFO)



stack: LIFO · queue: FIFO

A stack removes from the top (LIFO); a queue removes from the front (FIFO)

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

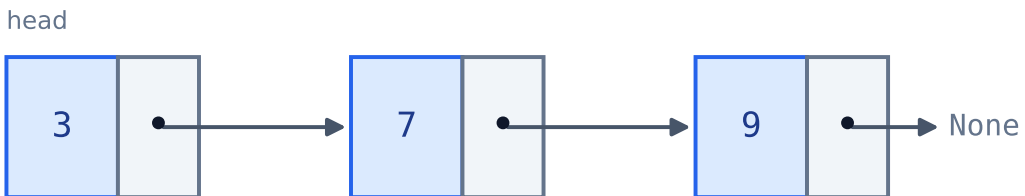
Queues

A queue is first-in, first-out (FIFO). You enqueue at the back and dequeue from the front.

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

Linked lists

A linked list is a chain of nodes . Each node holds data and a pointer to the next node; the last points to **None**.



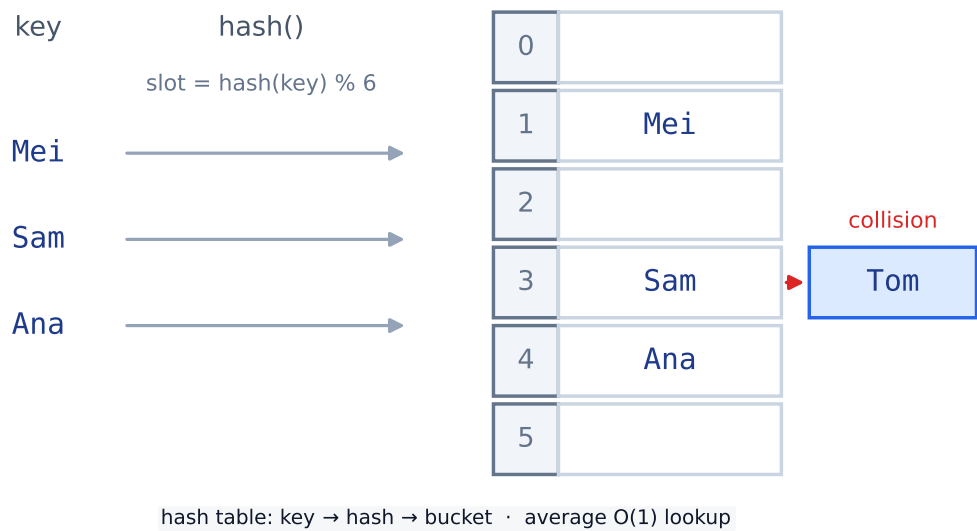
linked list: node → next · O(1) insert at head

A linked list: each node holds data and a pointer to the next node, ending at 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
```

Hash tables

A hash table maps a key to a slot with a hash function . Two keys can land in the same slot — a collision . Python’s `dict` is a hash table, so lookup is fast.

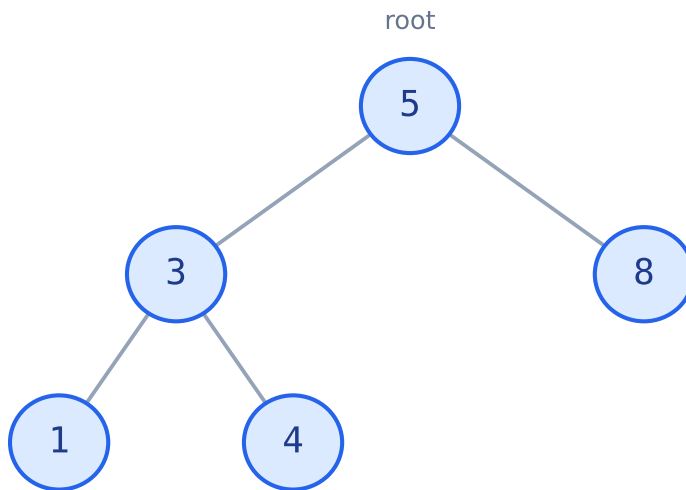


A hash function maps each key to a slot; two keys can collide in one slot

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

Binary search trees

A binary search tree (BST) keeps order: every left child is smaller than its node, every right child is larger. Search stays fast.



in-order traversal: 1 3 4 5 8

BST: left < node < right · search $O(\log n)$ if balanced

A binary search tree: smaller values go left, larger values go right

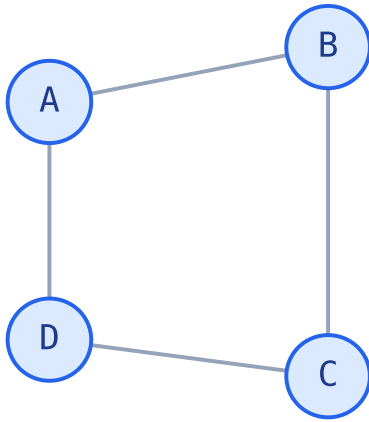
```
def insert(root, val):
    if root is None:
        return {"val": val, "left": None, "right": None}
    if val < root["val"]:
        root["left"] = insert(root["left"], val)
    else:
        root["right"] = insert(root["right"], val)
    return root

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]
```

Graphs

A graph is a set of vertices joined by edges. An adjacency list — a dict of neighbour lists — is a common way to store one.



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

graph: vertices + edges · BFS/DFS traverse

A graph of vertices and edges, with its adjacency-list form

```
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)
```

Common mistakes

- A stack is last-in-first-out; a queue is first-in-first-out. Do not mix them up.
- Check whether a structure is empty before you pop or dequeue from it.
- In a linked list, losing the **head** pointer loses the whole list.