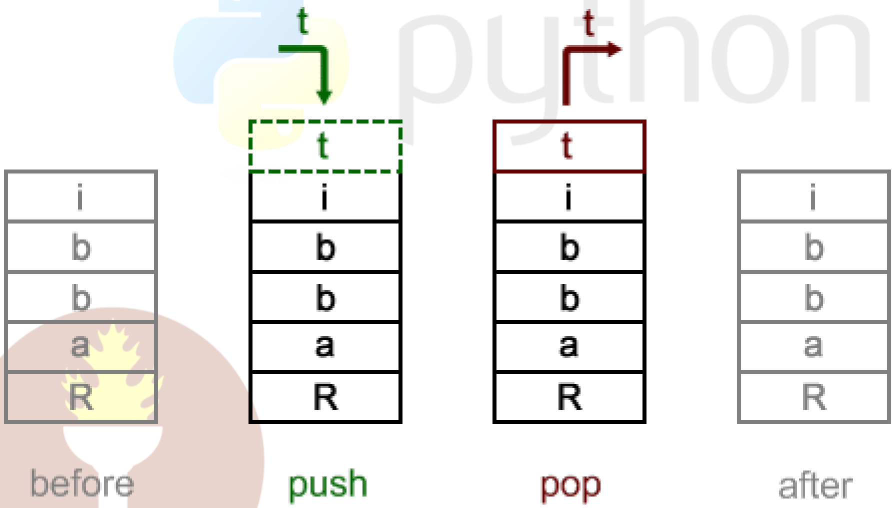
Queues, Stacks, and Buffers in Python

Learning Objectives

- Understand ADT Stack, Queue, and Deque
- Use built-in Stack, Queue, and Deque
- Implement ADT Stack and Queue

ADT Stack

A stack is a linear data structure that stores items in linearly (bottom to top). A new element is added at one end (top) and an element can be removed from that end only. The insert (at top) and remove (at top) operations are often called push and pop. These are the literal names in assembler.



ADT Stack Interface

Besides creation, the functions associated with stack are

`isEmpty()` Returns whether the stack is empty

`size()` Returns the size of the stack

`top()` Returns a reference to the topmost element of the stack. Also called **`peek()`**

`push(a)` Inserts the element **`a`** at the top of the stack

`pop()` Returns a copy of and deletes the topmost element of the stack

ADT Stack Implementation using Python List

The Python Documentation shows how to [implement stacks using Python lists](). In this lecture, we will focus on this simple implementation to completely understand stacks. In the companion Java lecture, we will look at an implementation using arrays.

`isEmpty()` We can use the **`len`** function of list.

`size()` We can use the **`len`** function of list.

`top()` The list accessor **`stack[-1]`** peeks the last entry, which is what top is supposed to accomplish.

`push(a)` The list function **`append`** implements a stack **`push`** as we would want it.

`pop()` The list function **`pop`** implements a stack **`pop`** as we would want it.

```python
class listStack:
    def __init__(self, iList):
        if type(iList) != list:
            raise TypeError("stack needs to initialized with a list")
        self._stack = iList

    # Python empty objects are "false"
    def isEmpty(self):
        return not self._stack

    def size(self):
        return self._stack.__len__()

    def top(self):
        return self._stack[-1]

    def push(self, value):
        self._stack.append(value)

    def pop(self):
        return self._stack.pop()

    def __str__(self):
        return self._stack.__str__()

    def __repr__(self):
        return self._stack.__repr__()
```

ADT Queue

Stacks are Last In First Out collections. A queue is a First In First Out collection, essentially a FIFO stack. The first item in is also the first item out. That means the push and pop operations of a stack need to act on different sides of the stack to create a queue.

Queues are appropriate for many real-world situations, for example a request to print a document (printer queue: requests can be entered faster than jobs can be completed). ***Queues*** are a special case of ***Deques*** (double-ended queues), so in Python they can be implemented as *deque* which was designed to have fast appends and pops from both ends. They do support the standard list interface, and we will look at implementation and use.

ADT Queue Interface

We are going to use the Python Documentation sample implementation for using lists as queues. The interface for queues should include

`createQueue()` Create an empty(!) queue

`isEmpty()` Determine whether queue is empty

`enqueue(item)` Add a new item to the queue

`dequeue()` Remove from the queue the item that was added first

`dequeueAll()` Remove all the items from the queue (clear)

`peek()` Retrieve from the queue the item that was added earliest (without removing)

ADT Queue Implementation Using Deque/List

```python
from collections import deque

class listQueue:
    def __init__(self):
        self._queue = deque([])

    def isEmpty(self):
        return not self._queue

    def enqueue(self, item):
        self._queue.append(item)

    def dequeue(self):
        return self._queue.popleft()

    def __repr__(self):
        return self._queue.__repr__()

    def __str__(self):
        return self._queue.__str__()
```

ADT Deque

As mentioned earlier, a ***deque*** is a double-ended queue. Defining a deque boils down to doubling the interface for queues.

ADT Deque Interface

`createDeque(aList)` Create a deque from a given list

`isEmpty()` Determine whether deque is empty

`enqueueLeft(item)` Add a new item to the deque on front

`dequeueLeft()` Remove from the deque the item that was added to the front

`peekLeft()` Retrieve from the deque the item that was added on the left (without removing)

`enqueueRight(item)` Add a new item to the deque on back

`dequeueRight()` Remove from the deque the item that was to the back

`peekRight()` Retrieve from the deque the item that was added on the right (without removing)

`dequeueAll()` Remove all the items from the queue (clear)

Goodbye

Auf Wiedersehen

再见