Scientific Programming: Part B

Lecture 5

Luca Bianco - Academic Year 2020-21
luca.bianco@fmach.it
[credits: thanks to Prof. Alberto Montresor]

Dictionary: ADT

DICTIONARY
% Returns the value associated to key k, if present; returns **none** otherwise
OBJECT lookup(OBJECT k)
% Associates value v to key k
insert(OBJECT k, OBJECT v)
% Removes the association of key k
remove(OBJECT k)

Note: insert replaces the object associated to the key if already present

Possible implementations of a dictionary

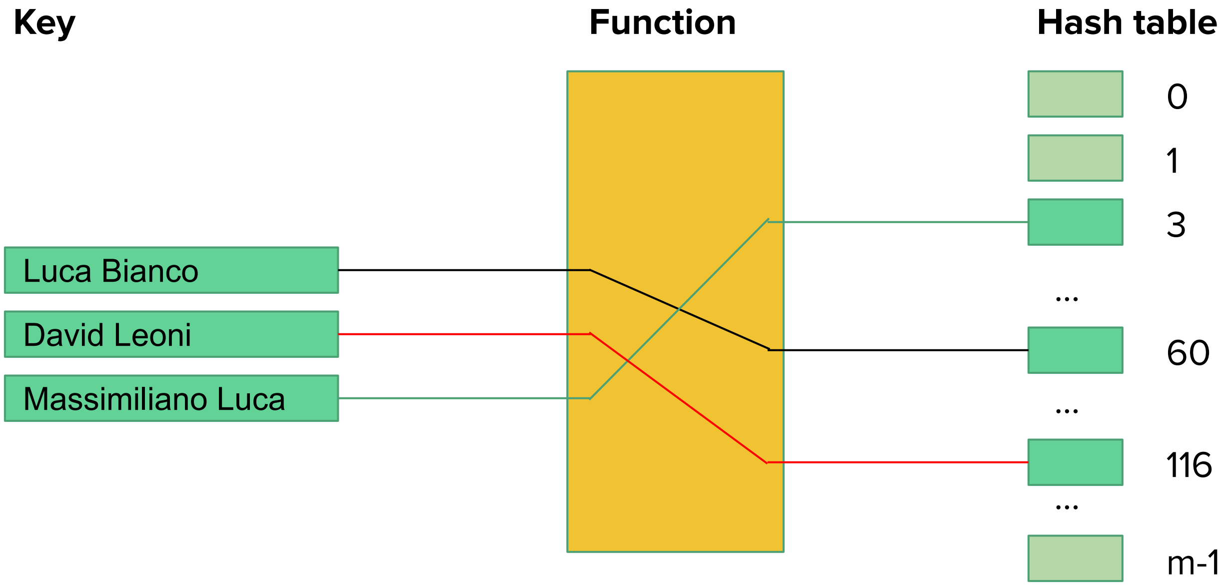
	Unordered array	**Ordered array**	**Linked List**	**RB Tree**	**Ideal impl.**
insert()	$O(1), O(n)$	$O(n)$	$O(1), O(n)$	$O(\log n)$	$O(1)$
lookup()	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
remove()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$

Ideal implementation: hash tables

- Choose a hash function h that maps each key $k \in \mathcal{U}$ to an integer $h(k)$
- The key-value $\langle k, v \rangle$ is stored in a list at position $h(k)$
- This vector is called hash table

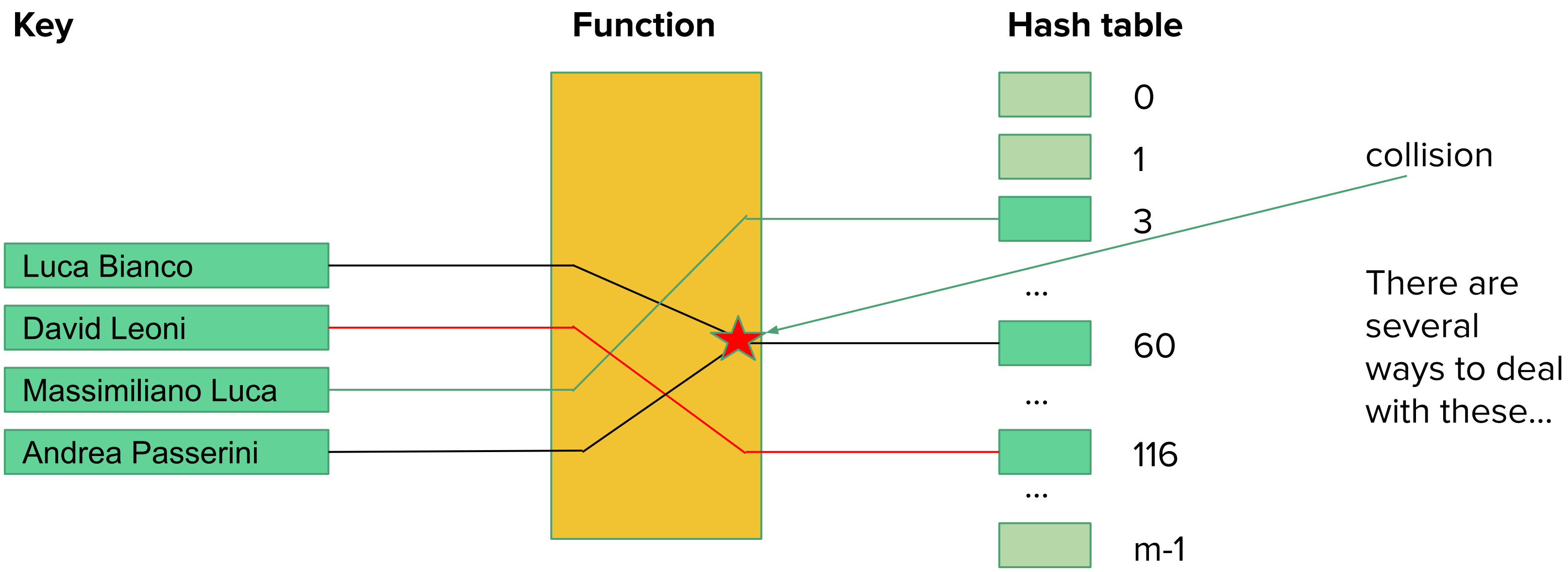
Hash table: definitions

- All the possible keys are contained in the universe set $\mathcal{U}$ of size u
- The table is stored in list $T[0 \ldots m-1]$ with size m
- An hash function is defined as: $h : \mathcal{U} \rightarrow \{0, 1, \ldots, m-1\}$



Hash table: collisions

- When two or more keys in the dictionary have the same hash values, we say that a collision has happened
- Ideally, we want to have hash functions with no collisions



Direct access tables

- All the possible keys are contained in the universe set $\mathcal{U}$ of size u
- The table is stored in list $T[0 \dots m-1]$ with size m
- An hash function is defined as: $h : \mathcal{U} \rightarrow \{0, 1, \dots, m-1\}$

In some cases: the set $\mathcal{U}$ is already a (small) subset of $\mathbb{Z}^+$

Example: days of the year

Direct access tables

- We use the identity function $h(k) = k$ as hash function
- We select $m = u$

Problems

- If u is very large, this approach may be infeasible
- If u is large but the number of keys that are actually recorded is much smaller than $u = m$, memory is wasted

Perfect hash function

- All the possible keys are contained in the universe set $\mathcal{U}$ of size u
- The table is stored in list $T[0 \dots m-1]$ with size m
- An hash function is defined as: $h : \mathcal{U} \rightarrow \{0, 1, \dots, m-1\}$

Definition

A hash function h is called perfect if h is injective, i.e.

$$\forall k_1, k_2 \in \mathcal{U} : k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$$

Examples

- Students ASD 2005-2016
  N. matricola in $[100.090, 183.864]$

  $h(k) = k - 100.090, m = 83.774$

- Studentes enrolled in 2014
  N. matricola in [173.185, 183.864]

  $h(k) = k - 173.185, m = 10.679$

Problems

- Universe space is often large, sparse, unknown
- To obtain a perfect hash function is difficult

Hash functions

If collisions cannot be avoided

- Let's try to minimize their number
- We want hash functions that uniform distribute the keys into hash indexes $[0 \dots m-1]$

→ we will have to deal with collisions anyway. More on this later...

Simple uniformity

- Let $P(k)$ be the probability that key k is inserted in the table
- Let $Q(i)$ be the probability that a key ends up in the i-th entry of the table
$$Q(i) = \sum_{k \in \mathcal{U}: h(k)=i} P(k)$$
- An hash function h has simple uniformity if:
$$\forall i \in [0, \dots, m-1] : Q(i) = 1/m$$

Hash functions

To obtain a hash function with simple uniformity, the probability distribution P should be known

Example

if $\mathcal{U}$ is given by real number in $[0, 1[$ and each key has the same probability of being selected, then $H(k) = \lfloor km \rfloor$ has simple uniformity

In the real world

- The key distribution may unknown or partially known
- Heuristic techniques are used to obtain an approximation of simple uniformity

Simple uniformity

- Let $P(k)$ be the probability that key k is inserted in the table
- Let $Q(i)$ be the probability that a key ends up in the i-th entry of the table
$$Q(i) = \sum_{k \in \mathcal{U} : h(k) = i} P(k)$$
- An hash function h has simple uniformity if:
$$\forall i \in [0, \ldots, m-1] : Q(i) = 1/m$$

Hash functions: possible implementations

Assumption

Each key can be translated in a numerical, non-negative values, by reading their internal representation as a number.

Example: string transformation

- $ord(c)$: ordinal binary value of character c in ASCII
- $bin(k)$: binary representation of key k, by concatenating the binary values of its characters
- $int(b)$: numerical value associated to the binary number b
- $int(k) = int(bin(k))$

Hash functions: possible implementations (the code)

```python
def H(in_string):
    d = "".join([str(bin(ord(x))) for x in in_string]).replace("b","")
    int_d = int(d,2)
    return int_d


L = "Luca"
D = "David"
C = "Massimiliano"
E = "Andrea"
A = "Alberto"
A1 = "Alan Turing"


people = [L, D, C, E, A, A1]

for p in people:
    print("H('{}')\t=\t{:,}".format(p, H(p)))
```

```
L: ord(L) = 76 bin(76) = 0b1001100
u: ord(u) = 117 bin(117) = 0b1110101
c: ord(c) = 99 bin(99) = 0b1100011
a: ord(a) = 97 bin(97) = 0b1100001
01001100011101010110001101100001 -> 1,282,761,569
```

```
H('Luca')          =       1,282,761,569
H('David')         =       293,692,926,308
H('Massimiliano')          =       23,948,156,761,864,131,868,341,923,439
H('Andrea')        =       71,942,387,426,657
H('Alberto')       =       18,415,043,350,787,183
H('Alan Turing')           =       39,545,995,566,905,718,680,940,135
```

ord → ascii representation of a character

Replace the b that stands for binary!

Hash function implementation

So far, we translated strings into big numbers.

Question for you: how do we convert these big numbers into values in [0, ..., m-1] where m is the size of the hash table?

```
H('Luca')         =       1,282,761,569
H('David')        =       293,692,926,308
H('Massimiliano')         =       23,948,156,761,864,131,868,341,923,439
H('Andrea')       =       71,942,387,426,657
H('Alberto')      =       18,415,043,350,787,183
H('Alan Turing')          =       39,545,995,566,905,718,680,940,135
```

Hash function implementation

Division method

- Let m be a odd number (prime)
- $H(k) = int(k) \bmod m$

Be careful that:
m = 2^i means to consider the i least significant bits

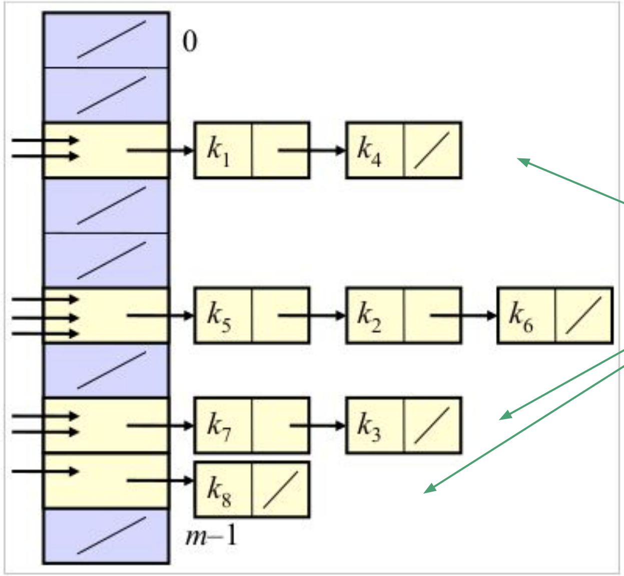
```python
def H(in_string):
    d = "".join([str(bin(ord(x))) for x in in_string]).replace("b","")
    int_d = int(d,2)
    return int_d

def my_hash_fun(key_str, m = 383):
    h = H(key_str)
    hash_key = h % m
    return hash_key

L = "Luca"
D = "David"
C = "Massimiliano"
E = "Andrea"
A = "Alberto"
A1 = "Alan Turing"


people = [L, D, C, E, A, A1]


prime = 383
for p in people:
    print("{} \t {:,} mod {}\t\t Index: {}".format(p, H(p),prime,my_hash_fun(p,prime)))
```

Luca	1,282,761,569 mod 383	Index: 351
David	293,692,926,308 mod 383	Index: 345
Massimiliano	23,948,156,761,864,131,868,341,923,439 mod 383	Index: 208
Andrea	71,942,387,426,657 mod 383	Index: 111
Alberto	18,415,043,350,787,183 mod 383	Index: 221
Alan Turing	39,545,995,566,905,718,680,940,135 mod 383	Index: 314

Conflicts: separate chaining

Idea

- The keys with the same value h are stored in a monodirectional list / dynamic vector
- The $H(k)$-th slot in the hash table contains the list/vector associated to k



add to the tail to be O(1)

Another possible method is to look for another place in the hash table where we can put the value (open addressing).

Separate chaining: complexity

n	Number of keys stored in the hash table
m	Size of the hash table
$\alpha = n/m$	Load factor
$I(\alpha)$	Average number of memory accesses to search a key that is not in the table (insuccess)
$S(\alpha)$	Average number of memory accesses to search a key that is not in the table (success)

Worst case analysis

- All the keys are inserted in a unique list
- insert(): $\Theta(1)$
- lookup(), remove(): $\Theta(n)$

Separate chaining: complexity

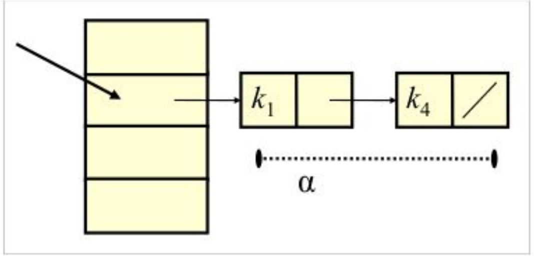
Average case analysis

- Let's assume the hash function has simple uniformity
- Hash function computation: $\Theta(1)$, to be added to all searches

all places have the same probability of contain one element

How long the lists are?

- The expected length of a list is equal to $\alpha = n/m$



alpha is the average length of each list

Separate chaining: complexity

Insuccess

- When searching for a missing key, all the keys in the list must be read
- Expected cost: $\Theta(1) + \alpha$

Success

- When searching for a key included in the table, on average half of the keys in the list must be read.
- Expected cost: $\Theta(1) + \alpha/2$

What is the meaning of the load factor?

- The cost factor of every operation is influenced by the load factor
- If $m = O(n)$, $\alpha = O(1)$
- In such case, all operations are $O(1)$ in expectation
- If α becomes too large, the size of the hash table can be doubled through dynamic vectors

Hash table: rules for hashing objects

Rule: If two objects are equal, then their hashes should be equal

- If you implement `__eq__()`, then you should implement function `__hash__()` as well

Rule: If two objects have the same hash, then they are likely to be equal

- You should avoid to return values that generate collisions in your hash function.

Rule: In order for an object to be hashable, it must be immutable

- The hash value of an object should not change over time

[https://www.asmeurer.com/blog/posts/what-happens-when-you-mess-with-hashing-in-python/]

Hash table: sample code (m = 11)

```python
class HashTable:

    # the table is a list of m empty lists
    def __init__(self, m):
        self.table = [[] for i in range(m)]


    #converts a string into an integer (our keys will be strings only)
    def H(self, key):
        d = "".join([str(bin(ord(x))) for x in key]).replace("b","")
        int_d = int(d,2)
        return int_d

    #gets a string and converts it into a hash-key
    def hash_function(self,str_obj):
        #m is inferred from the length of the table
        m = len(self.table)
        h = self.H(str_obj)
        hash_key = h % m
        return hash_key

    #adds a pair (key,value) to the hash table
    def insert(self, key, value):
        index = self.hash_function(key)
        self.table[index].append((key, value))
    #removes the value associated to key if it exists
    def remove(self, key):
        index = self.hash_function(key)
        for el in self.table[index]:
            if el[0] == key:
                self.table[index].remove(el)
                break
    #returns the value associated to key or None
    def search(self, key):
        index = self.hash_function(key)
        for el in self.table[index]:
            if el[0] == key:
                return el[1]

    #converts the table to a string
    def __str__(self):
        return str(self.table)
```

pair to deal with collisions

```python
if __name__ == "__main__":
    myHash = HashTable(11)
    myHash.insert("Luca",27)
    myHash.insert("David",5)
    myHash.insert("Massimiliano",12)
    myHash.insert("Andrea",15)
    myHash.insert("Alberto",12)
    myHash.insert("Alan",1)
    print(myHash)
    key = "Luca"
    print("{} -> {}".format(key, myHash.search(key)))
    myHash.remove("Luca")
    key = "Thomas"
    print("{} -> {}".format(key, myHash.search(key)))
    print(myHash)
```

[[('Andrea', 15)], [**('Luca', 27), ('David', 5), ('Alberto', 12)**], [], [], [('Alan', 1)], [], [('Massimiliano', 12)], [], [], [], []]

Luca -> 27
Thomas -> None

[[('Andrea', 15)], [**('David', 5), ('Alberto', 12)**], [], [], [('Alan', 1)], [], [('Massimiliano', 12)], [], [], [], []]

SOME CONFLICTS!

Hash table: sample code (m = 17)

```python
class HashTable:

    # the table is a list of m empty lists
    def __init__(self, m):
        self.table = [[] for i in range(m)]


    #converts a string into an integer (our keys will be strings only)
    def H(self, key):
        d = "".join([str(bin(ord(x))) for x in key]).replace("b","")
        int_d = int(d,2)
        return int_d

    #gets a string and converts it into a hash-key
    def hash_function(self,str_obj):
        #m is inferred from the length of the table
        m = len(self.table)
        h = self.H(str_obj)
        hash_key = h % m
        return hash_key

    #adds a pair (key,value) to the hash table
    def insert(self, key, value):
        index = self.hash_function(key)
        self.table[index].append((key, value))
    #removes the value associated to key if it exists
    def remove(self, key):
        index = self.hash_function(key)
        for el in self.table[index]:
            if el[0] == key:
                self.table[index].remove(el)
                break
    #returns the value associated to key or None
    def search(self, key):
        index = self.hash_function(key)
        for el in self.table[index]:
            if el[0] == key:
                return el[1]

    #converts the table to a string
    def __str__(self):
        return str(self.table)
```

```python
if __name__ == "__main__":
    myHash = HashTable(17)
    myHash.insert("Luca",27)
    myHash.insert("David",5)
    myHash.insert("Massimiliano",12)
    myHash.insert("Andrea",15)
    myHash.insert("Alberto",12)
    myHash.insert("Alan",1)
    print(myHash)
    key = "Luca"
    print("{} -> {}".format(key, myHash.search(key)))
    myHash.remove("Luca")
    key = "Thomas"
    print("{} -> {}".format(key, myHash.search(key)))
    print(myHash)
```

```
[[], [], [], [], [], [], [('Alan', 1)], [], [], [('Andrea', 15)], [], [], [('David', 5)], [('Massimiliano', 12)], [], [('Luca', 27)], [('Alberto', 12)]]
Luca -> 27
Thomas -> None
[[], [], [], [], [], [], [('Alan', 1)], [], [], [('Andrea', 15)], [], [], [('David', 5)], [('Massimiliano', 12)], [], [], [('Alberto', 12)]]
```

NO CONFLICTS!

In python...

Python `sets` and `dict`

- Are implemented through hash tables
- Sets are degenerate forms of dictionaries, where there are no values, only keys

Unordered data structures

- Order between keys is not preserved by the hash function; this is why you get unordered results when you print them

Python built-in: set

Operation		Average case	Worst case
`x in S`	Contains	$O(1)$	$O(n)$
`S.add(x)`	Insert	$O(1)$	$O(n)$
`S.remove(x)`	Remove	$O(1)$	$O(n)$
`S|T`	Union	$O(n+m)$	$O(n \cdot m)$
`S&T`	Intersection	$O(\min(n,m))$	$O(n \cdot m)$
`S-T`	Difference	$O(n)$	$O(n \cdot m)$
`for x in S`	Iterator	$O(n)$	$O(n)$
`len(S)`	Get length	$O(1)$	$O(1)$
`min(S), max(S)`	Min, Max	$O(n)$	$O(n)$

$$n = \texttt{len}(\texttt{S}), m = \texttt{len}(\texttt{T})$$

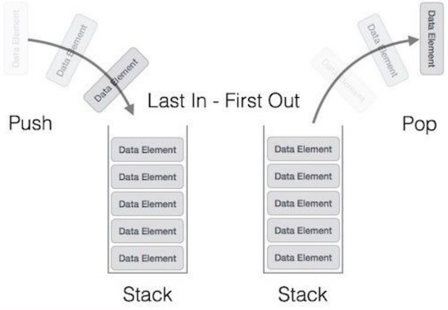
https://docs.python.org/2/library/stdtypes.html#set

Python built-in: dictionary

Operation		Average case	Worst case
`x in D`	Contains	$O(1)$	$O(n)$
`D[] =`	Insert	$O(1)$	$O(n)$
`= D[]`	Lookup	$O(1)$	$O(n)$
`del D[]`	Remove	$O(1)$	$O(n)$
`for x in S`	Iterator	$O(n)$	$O(n)$
`len(S)`	Get length	$O(1)$	$O(1)$

$$n = \texttt{len}(\texttt{S}), m = \texttt{len}(\texttt{T})$$

Stack: Last in, first out queue



Stack

A linear, dynamic data structure, in which the operation "remove" returns (and removes) a predefined element: the one that has remained in the data structure for the least time

STACK

% Returns **True** if the stack is empty
boolean isEmpty()

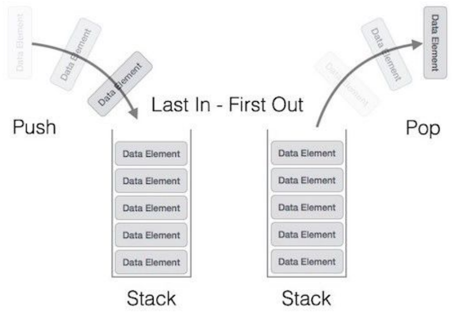
% Returns the size of the stack
int size()

% Inserts v on top of the stack
push(OBJECT v)

% Removes the top element of the stack and returns it to the caller
OBJECT pop()

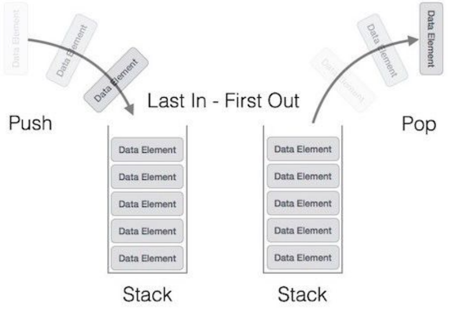
% Read the top element of the stack, without modifying it
OBJECT peek()

Stack: Last in, first out queue



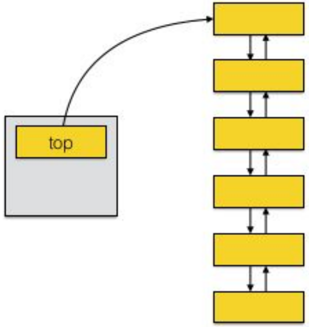
Stack Operation	Stack Contents	Return Value
s.isEmpty()	[]	True
s.push(4)	[4]	
s.push('dog')	[4,'dog']	
s.peek()	[4,'dog']	'dog'
s.push(True)	[4,'dog',True]	
s.size()	[4,'dog',True]	3
s.isEmpty()	[4,'dog',True]	False
s.push(8.4)	[4,'dog',True,8.4]	
s.pop()	[4,'dog',True]	8.4
s.pop()	[4,'dog']	True
s.size()	[4,'dog']	2

Stack: Last in, first out queue



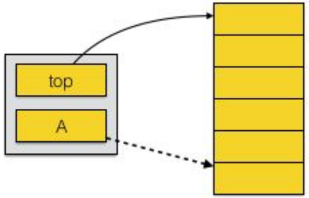
Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

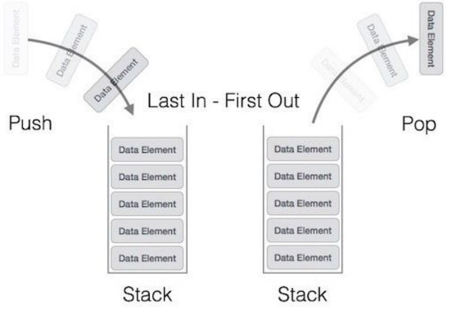


Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead

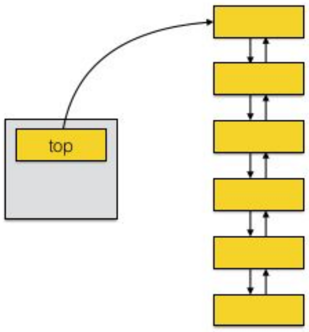


Stack: Last in, first out queue



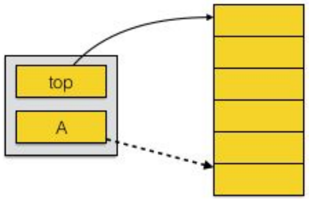
Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph



Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead

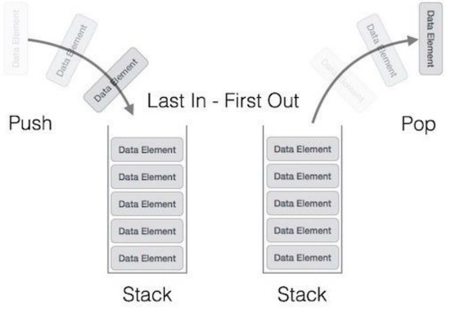



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```

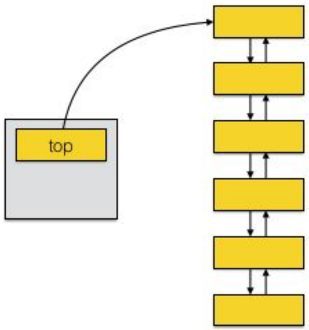
my_func(80)

Stack: Last in, first out queue



Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

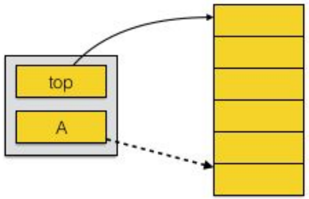



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```

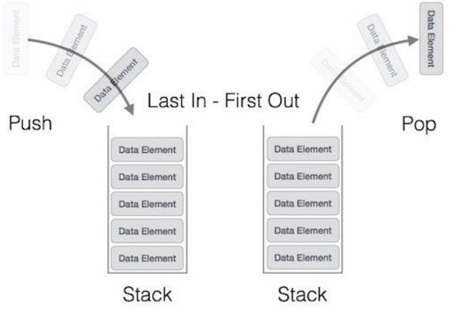
Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead



my_func(20)
my_func(80)

Stack: Last in, first out queue

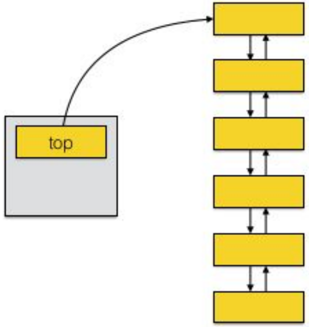


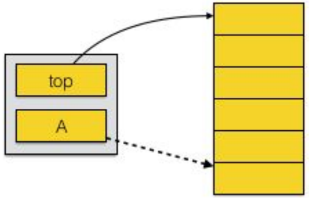
Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead



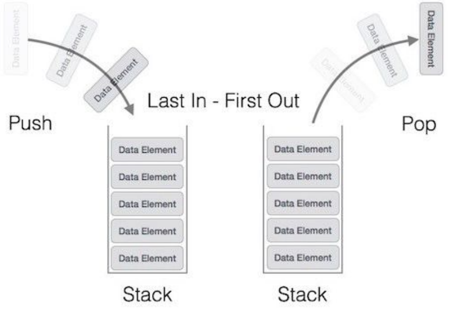



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```

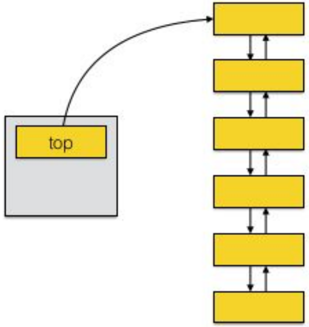
my_func(5)
my_func(20)
my_func(80)

Stack: Last in, first out queue



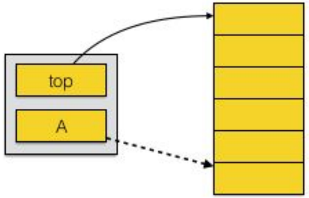
Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph



Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead

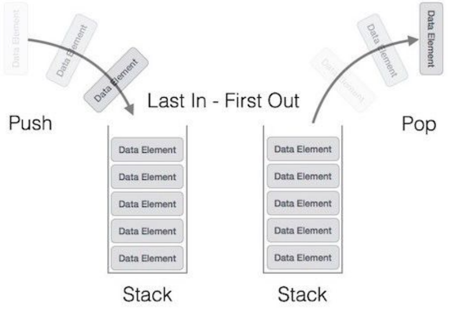



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```

my_func(1)
my_func(5)
my_func(20)
my_func(80)

Stack: Last in, first out queue

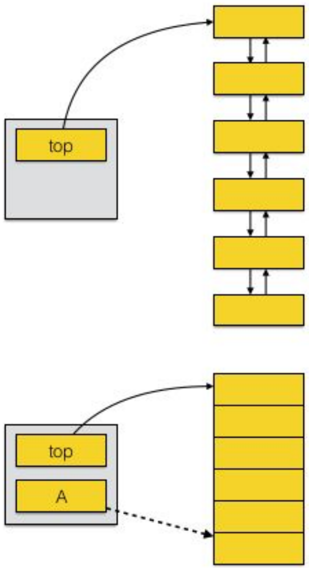


Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

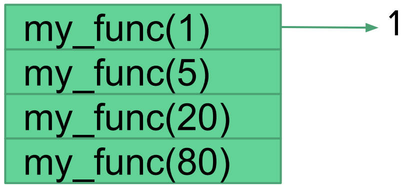
Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead

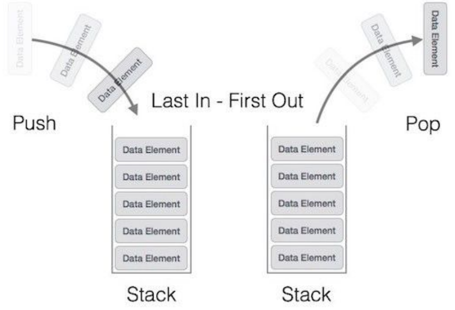



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```



Stack: Last in, first out queue

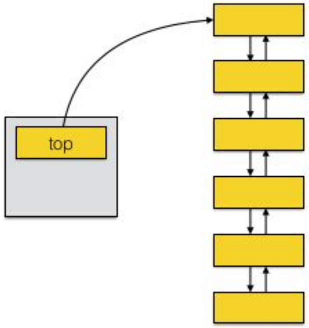


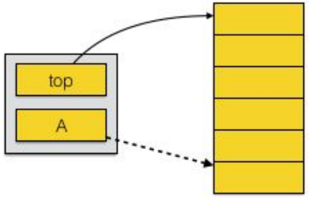
Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

Possible implementations

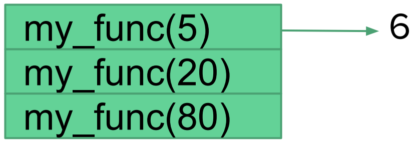
- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead



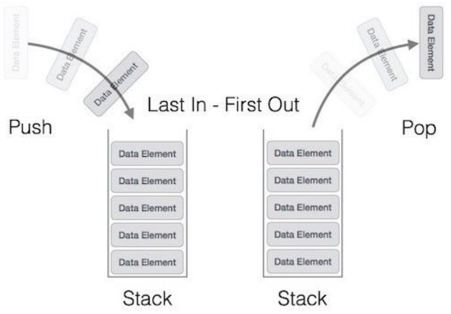



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```



Stack: Last in, first out queue

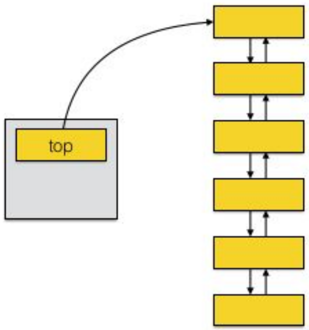


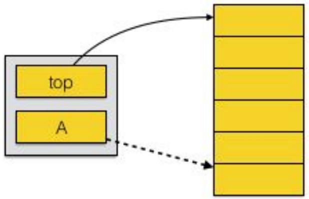
Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

Possible implementations

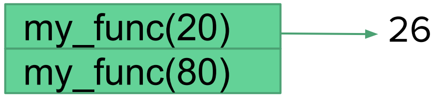
- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead



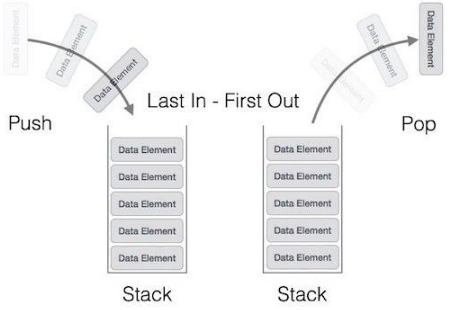



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```



Stack: Last in, first out queue

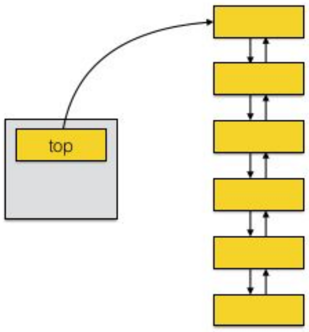


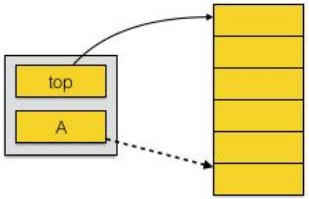
Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead



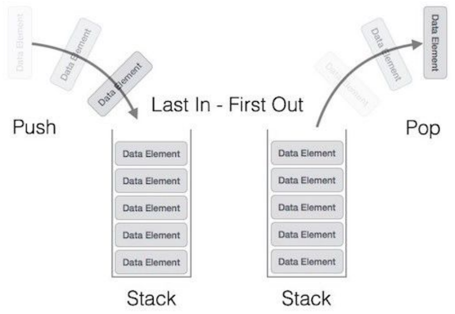



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```

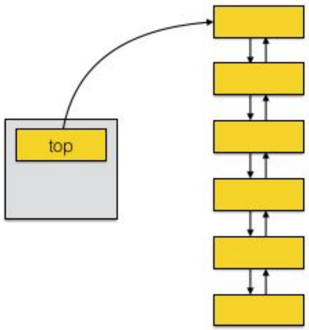
my_func(80) → 106

Stack: Last in, first out queue



Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

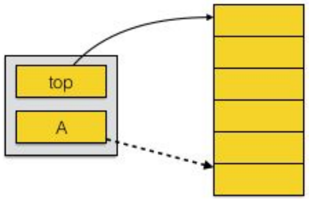



```python
def my_func(x):
    if x <= 2:
        return x
    else:
        print("{} + my_func({})".format(x,x//4))
        return x + my_func(x//4)

print(my_func(80))
```

```
80 + my_func(20)
20 + my_func(5)
5 + my_func(1)
106
```

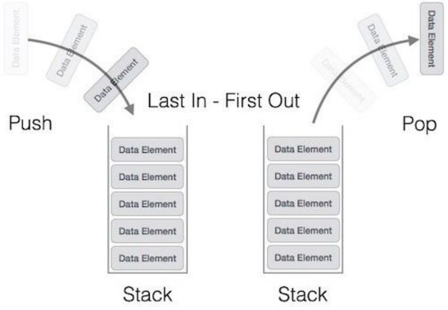
Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead



→ 106

Stack: Last in, first out queue

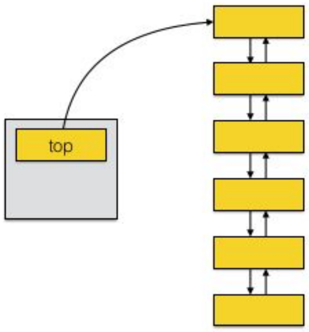


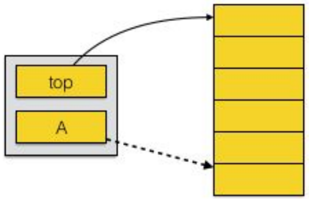
Possible uses

- In languages like Python:
  - Compiler: To balance parentheses
  - In the the interpreter: A new activation record is created for each function call
- In graph analysis:
  - To perform visits of the entire graph

Possible implementations

- Through bidirectional lists
  - reference to the top element
- Through vectors
  - limited size, small overhead





Note: the stack has finite size!

```python
import sys

def my_funct2(x,s):
    if x < 1:
        return s
    else:
        return my_funct2(x-1, s+x)

print(sys.getrecursionlimit())
print(my_funct2(3100,0))
#This would fix it
#print(sys.setrecursionlimit(3200))
#print(my_funct2(3100,0))
```

```
<ipython-input-38-a7a6c79ddbc8> in my_funct2(x, s)
      5         return s
      6     else:
----> 7         return my_funct2(x-1, s+x)
      8
      9 print(my_funct2(3100,0))

RecursionError: maximum recursion depth exceeded in comparison
```

Stack: implementation

```python
class Stack:

    # initializer, the inner structure is a list
    # data is added at the end of the list
    # for speed
    def __init__(self):
        self.__data = []

    # returns the length of the stack (size)
    def __len__(self):
        return len(self.__data)

    # returns True if stack is empty
    def isEmpty(self):
        return len(self.__data) == 0

    # returns the last inserted item of the stack
    # and shrinks the stack
    def pop(self):
        if len(self.__data) > 0:
            return self.__data.pop()


    # returns the last inserted element without
    # removing it (None if empty)
    def peek(self):
        if len(self.__data) > 0:
            return self.__data[-1]
        else:
            return None

    # adds an element to the stack
    def push(self, item):
        self.__data.append(item)

    # transforms the Stack into a string
    def __str__(self):
        if len(self.__data) == 0:
            return "Stack([])"
        else:
            out = "Stack([" + str(self.__data[-1])
            for i in range(len(self.__data) -2,-1, -1):
                out += " | " + str(self.__data[i])
            out += "])"
            return out
```

could have used a deque, linked list,...

STACK

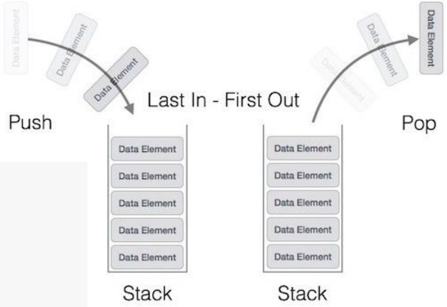
% Returns **True** if the stack is empty
boolean isEmpty()

% Returns the size of the stack
boolean size()

% Inserts v on top of the stack
push(OBJECT v)

% Removes the top element of the stack and returns it to the caller
OBJECT pop()

% Read the top element of the stack, without modifying it
OBJECT peek()




```python
if __name__ == "__main__":
    S = Stack()
    print(S)
    print("Empty? {}".format(S.isEmpty()))
    S.push("Luca")
    S.push(1)
    S.push(27)
    print(S)
    S.push([1,2,3])
    print("The stack has {} elements".format(len(S)))
    print(S)
    print("Last inserted: {}".format(S.peek()))
    print("Removed: {}".format(S.pop()))
    print("Stack now:")
    print(S)
```

```
Stack([])
Empty? True
Stack([27 | 1 | Luca])
The stack has 4 elements
Stack([[1, 2, 3] | 27 | 1 | Luca])
Last inserted: [1, 2, 3]
Removed: [1, 2, 3]
Stack now:
Stack([27 | 1 | Luca])
```

Stack: uses

- Check whether the following sets of parentheses are balanced
  - { { ([ ] [ ]) } ( ) }
  - [[{ { (( )) } }]]
  - [ ] [ ] [ ] ( ) { }
  - ([ )]
  - ((( )]))
  - [{ ( )]

Stack: exercise

Ideas on how to implement **`par_checker`** using a Stack?

Simplifying assumption: only characters allowed in input are "{ [( )] }"

```
p1 = "{{([][])}()}"
p2 = "[{()]"
p3 = "{[(())][{[]}]}"
p4 = "{[(())][{[]}]"

blocks = [p1, p2, p3, p4]
for p in blocks:
    print("{} \t\tbalanced:\t {}".format(p,
                                           par_checker(p)))
```

Possible solution:
Loop through the input string and...

- push opening parenthesis to stack
- when analyzing a closing parenthesis, pop one element from the stack and compare: if matching keep going, else return False

Desired output

{{([][])}()}	balanced:	True
[{()]	balanced:	False
{[(())][{[]}]}	balanced:	True
{[(())][{[]}]	balanced:	False

Stack: exercise

```python
def par_match(open_p, close_p):
    openers = "{[("
    closers = "}])"

    if openers.index(open_p) == closers.index(close_p):
        return True
    else:
        return False



def par_checker(parString):
    s = Stack()

    for symbol in parString:
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                return False
            else:
                top = s.pop()
                if not par_match(top,symbol):
                    return False
    return s.isEmpty()
```

```python
p1 = "{{([][])}()}"
p2 = "[{()]"
p3 = "{[(())][{[]}]}"
p4 = "{[(())][{[]}]"

blocks = [p1, p2, p3, p4]
for p in blocks:
    print("{} \t\tbalanced:\t {}".format(p,
                                      par_checker(p)))
```

Desired output

{{([][])}()}	balanced:	True
[{()]	balanced:	False
{[(())][{[]}]}	balanced:	True
{[(())][{[]}]	balanced:	False

Queue: First in, first out queue (FIFO)

Queue

A linear, dynamic data structure, in which the operation "remove" returns (and removes) a predefined element: the one that has remained in the data structure for the longest time)

QUEUE

% Returns **True** if queue is empty
boolean isEmpty()

% Returns the size of the queue
int size()

% Inserts v at the end of the queue
enqueue(OBJECT v)

% Extracts q from the beginning of the queue
OBJECT dequeue()

% Reads the element at the top of the queue
OBJECT top()

Queue: example

QUEUE

% Returns **True** if queue is empty
boolean isEmpty()

% Returns the size of the queue
int size()

% Inserts v at the end of the queue
enqueue(OBJECT v)

% Extracts q from the beginning of the queue
OBJECT dequeue()

% Reads the element at the top of the queue
OBJECT top()

Queue Operation	Queue Contents	Return Value
q.isEmpty()	[]	True
q.enqueue(4)	[4]	
q.enqueue('dog')	['dog',4]	
q.enqueue(True)	[True,'dog',4]	
q.size()	[True,'dog',4]	3
q.isEmpty()	[True,'dog',4]	False
q.enqueue(8.4)	[8.4,True,'dog',4]	
q.dequeue()	[8.4,True,'dog']	4
q.dequeue()	[8.4,True]	'dog'
q.size()	[8.4,True]	2

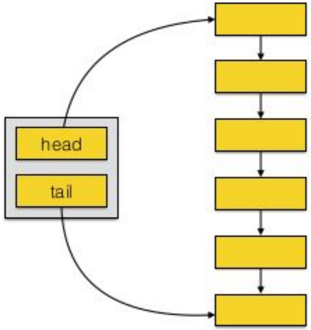
Queue: uses and implementation

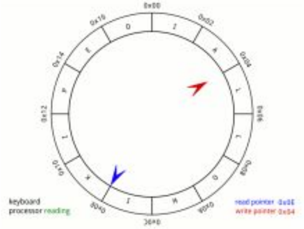
Possible uses

- To queue requests performed on a limited resource (e.g., printer)
- To visit graphs

Possible implementations

- Through lists
  - add to the tail
  - remove from the head
- Through circular array
  - limited size, small overhead





Queue: as a list (with deque)

QUEUE

% Returns **True** if queue is empty
boolean isEmpty()

% Returns the size of the queue
int size()

% Inserts v at the end of the queue
enqueue(OBJECT v)

% Extracts q from the beginning of the queue
OBJECT dequeue()

% Reads the element at the top of the queue
OBJECT top()

```python
from collections import deque

class Queue:

    def __init__(self):
        self.__data = deque()

    def __len__(self):
        return len(self.__data)

    def __str__(self):
        return str(self.__data)

    def isEmpty(self):
        return len(self.__data) == 0

    def top(self):
        if len(self.__data) > 0:
            return self.__data[-1]

    def enqueue(self, item):
        self.__data.appendleft(item)

    def dequeue(self):
        if len(self.__data) > 0:
            return self.__data.pop()
```

```python
if __name__ == "__main__":
    Q = Queue()
    print(Q)
    print("TOP: {}".format(Q.top()))
    print(Q.isEmpty())
    Q.enqueue(4)
    Q.enqueue('dog')
    Q.enqueue(True)
    print(Q)
    print("Size: {}".format(len(Q)))
    print(Q.isEmpty())
    Q.enqueue(8.4)
    print("Removing: '{}'".format(Q.dequeue()))
    print("Removing: '{}'".format(Q.dequeue()))
    print(Q)
    print("Size: {}".format(len(Q)))
```

```
deque([])
TOP now: None
True
deque([True, 'dog', 4])
Size: 3
False
Removing: '4'
Removing: 'dog'
deque([8.4, True])
Size: 2
```

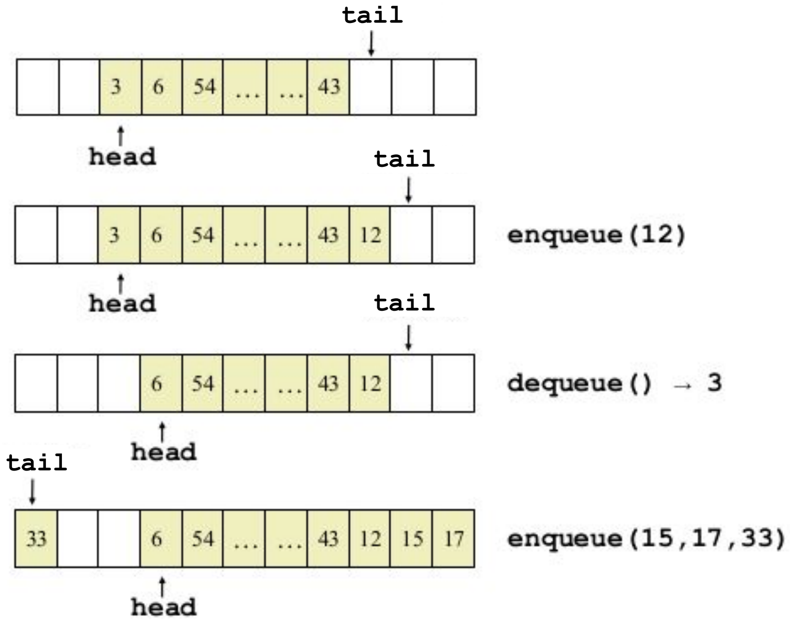
Not very interesting implementation.

Just **pay attention** to the case when the **Queue is empty in top and dequeue**

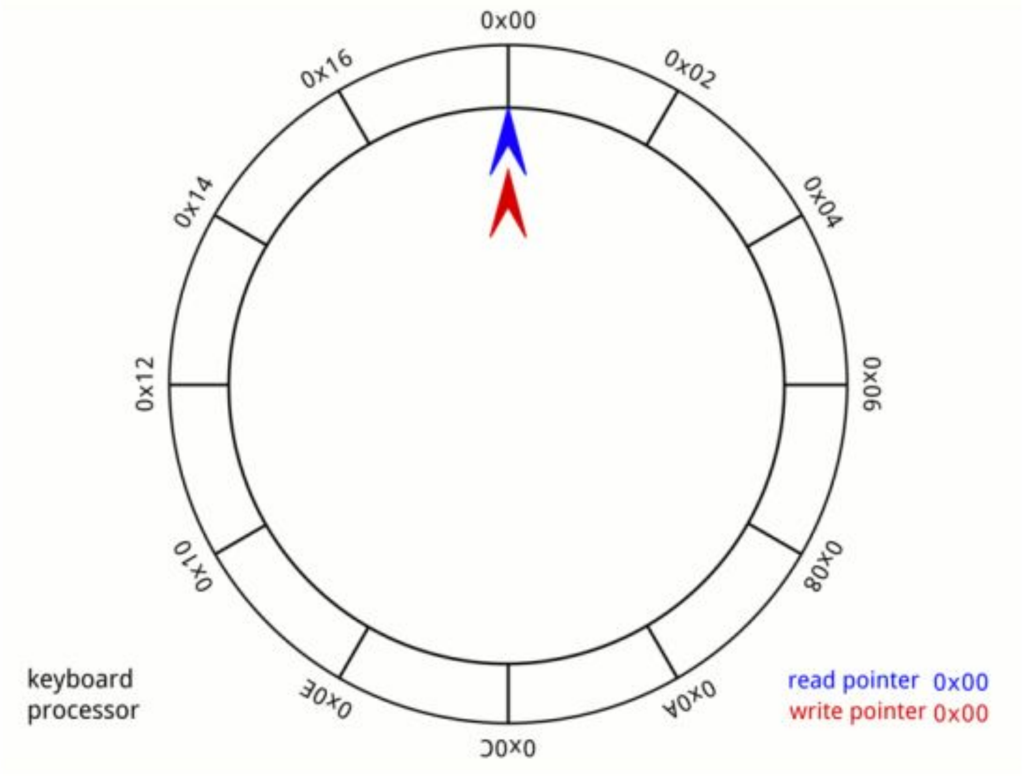
Makes use of efficient deque object that provides ~ O(1) push/pop
https://docs.python.org/3.7/library/collections.html#collections.deque

Queue as a circular list

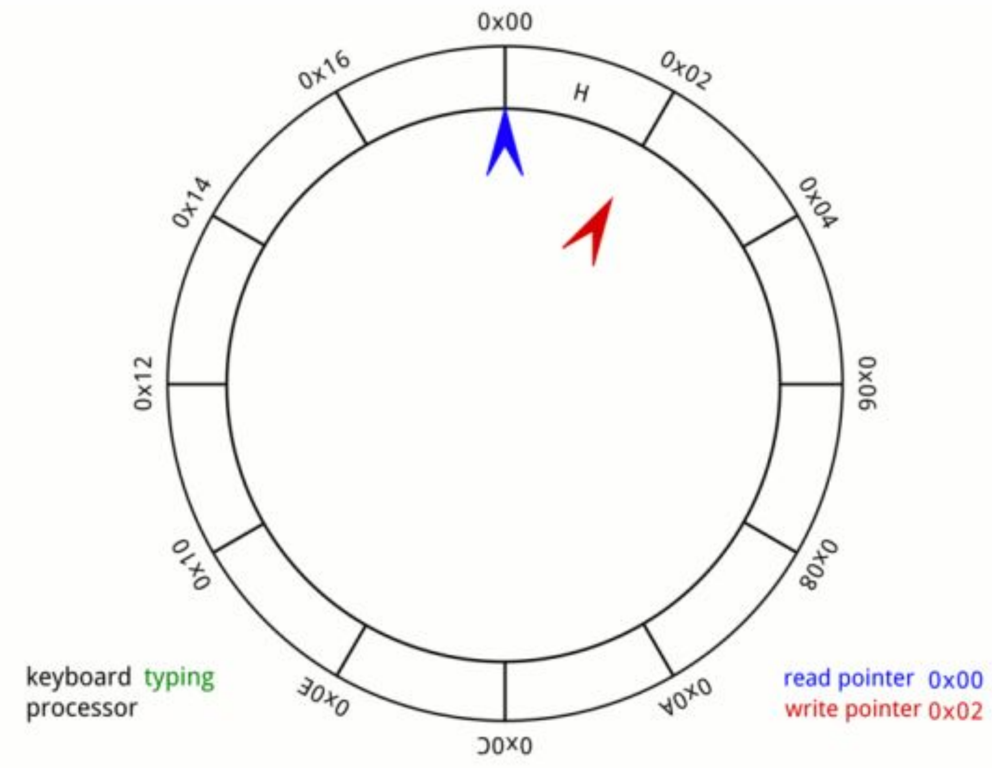
- Implementation based on the modulus operation
- Pay attention to overflow problems (full queue)



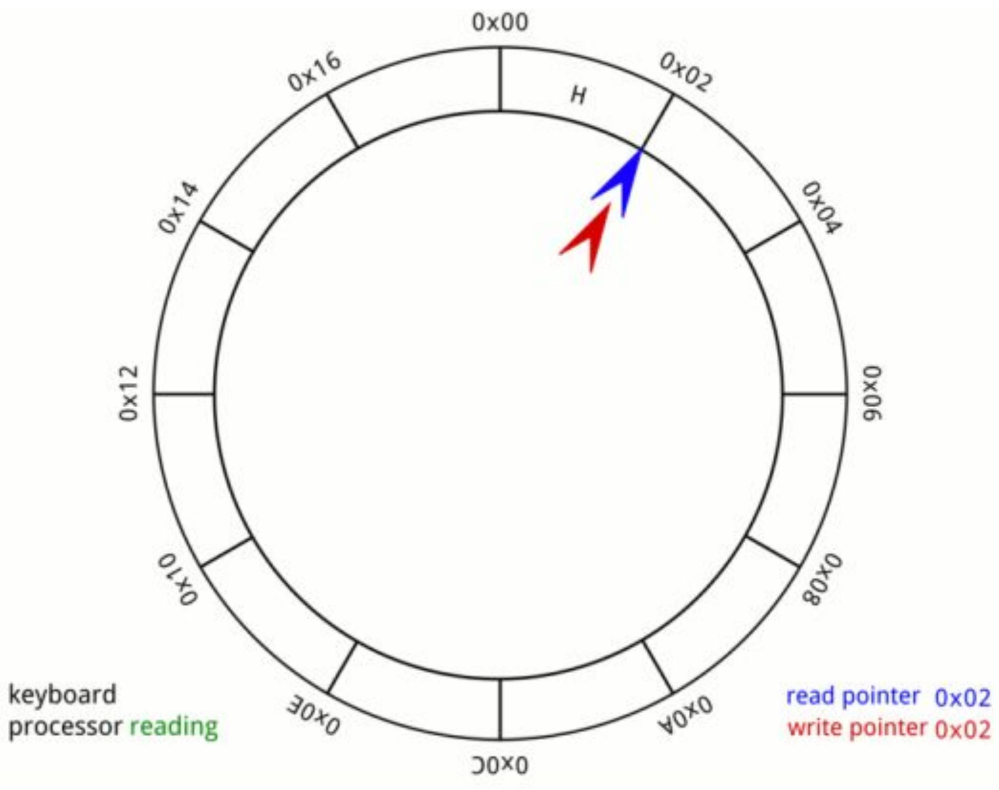
Queue as a circular list: example



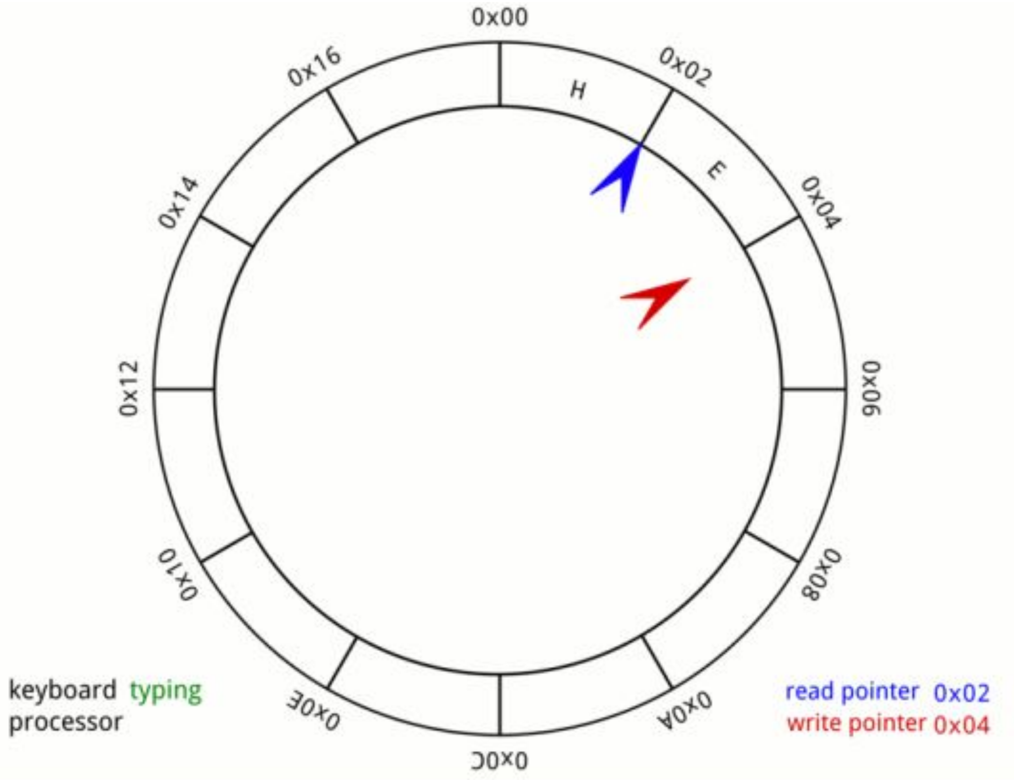
Queue as a circular list: example



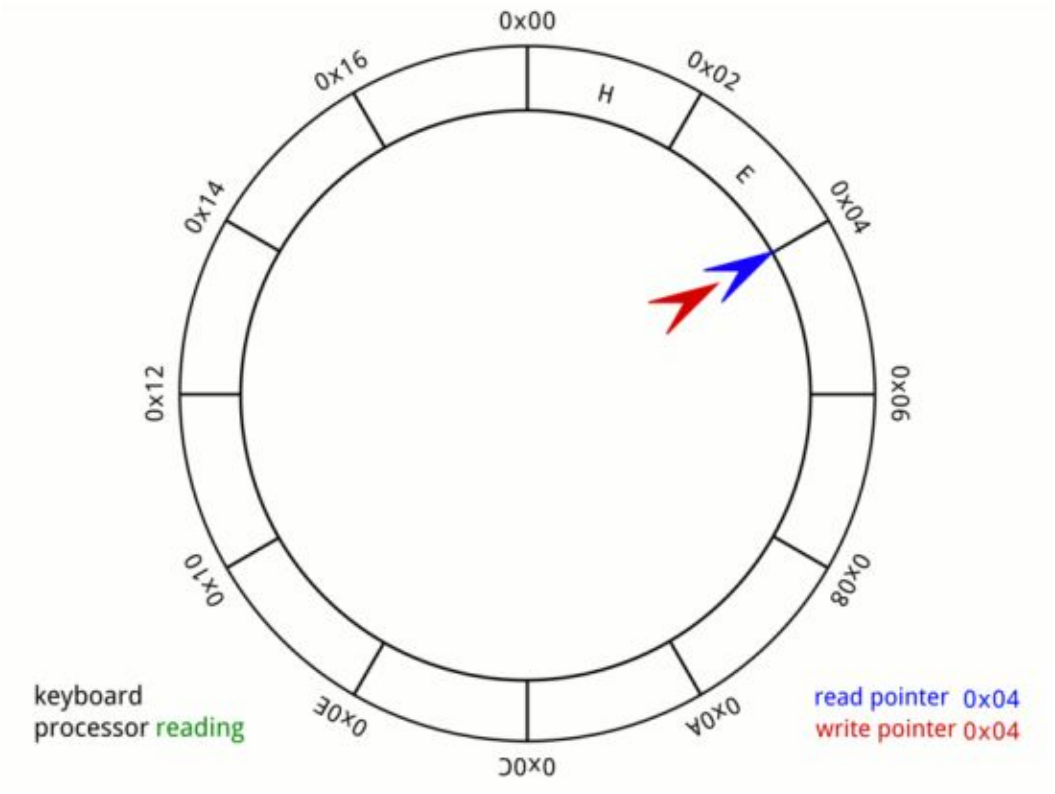
Queue as a circular list: example



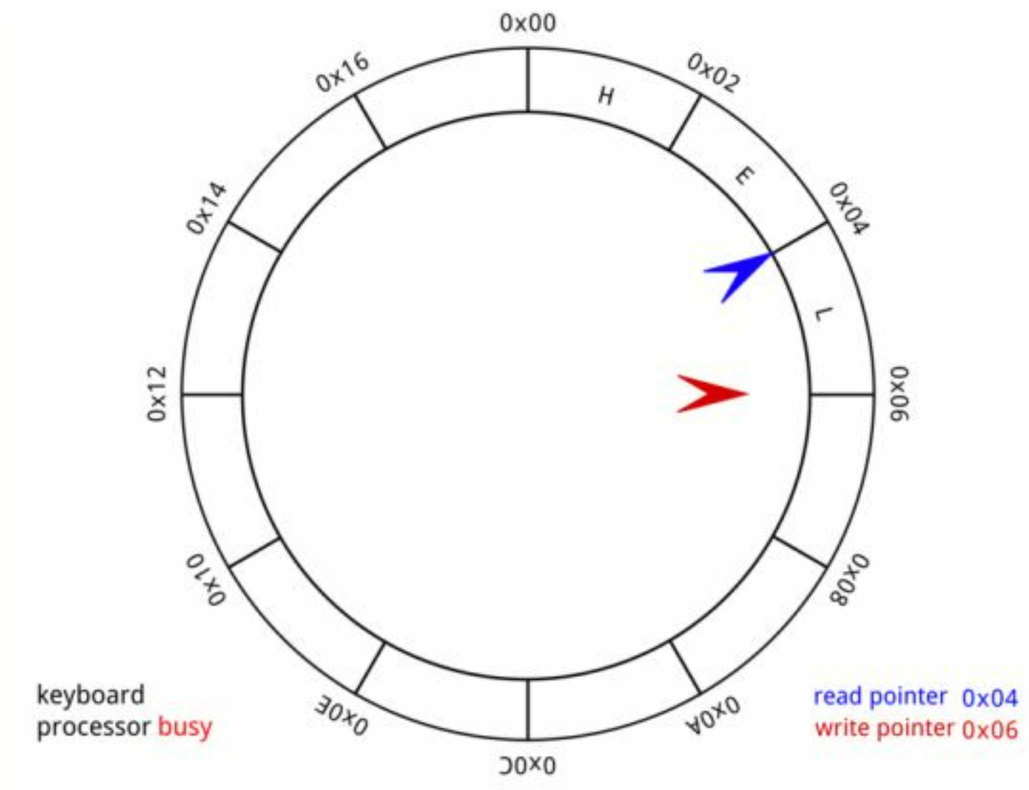
Queue as a circular list: example



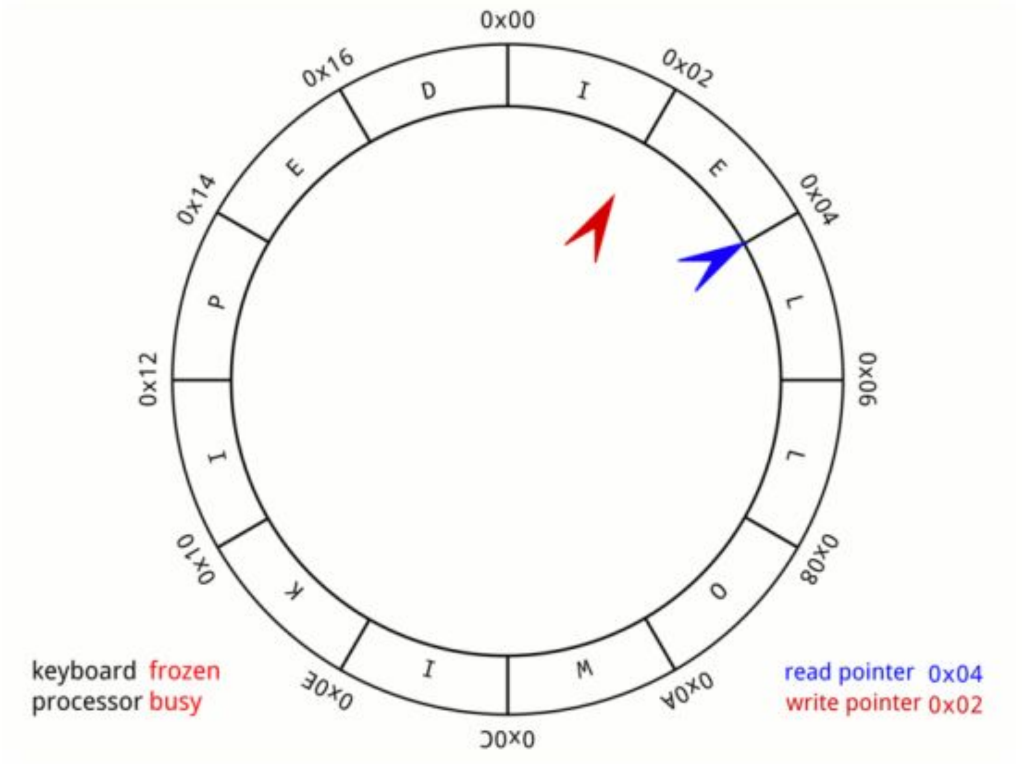
Queue as a circular list: example



Queue as a circular list: example

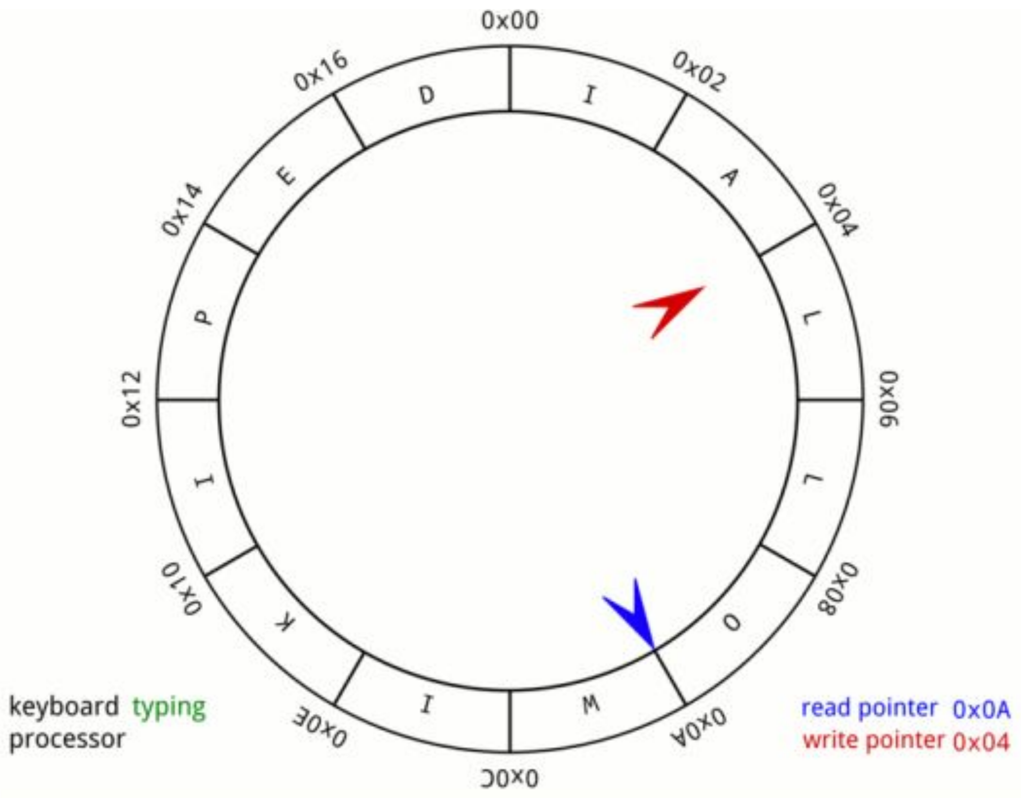


Queue as a circular list: example



skipping a few typing steps...

Queue as a circular list: example



skipping a few typing/reading steps...

Queue as a circular list: exercise

Queue

% Returns **True** if queue is empty
boolean isEmpty()

% Returns the size of the queue
int size()

% Inserts v at the end of the queue
enqueue(OBJECT v)

% Extracts q from the beginning of the queue
OBJECT dequeue()

% Reads the element at the top of the queue
OBJECT top()

Implement the CircularQueue data structure

(without going to the next slide…)

Queue as a circular list: the code

```python
class CircularQueue:

    def __init__(self, N):
        self.__data = [None for i in range(N)]
        self.__head = 0
        self.__tail = 0
        self.__size = 0
        self.__max = N

    def top(self):
        if self.__size > 0:
            return self.__data[self.__head]

    def dequeue(self):
        if self.__size > 0:
            ret = self.__data[self.__head]
            self.__head = (self.__head + 1) % self.__max
            self.__size -= 1
            return ret

    def enqueue(self, item):
        if self.__max > self.__size:
            self.__data[self.__tail] = item
            self.__tail = (self.__tail + 1) % self.__max
            self.__size += 1
        else:
            raise Exception("The queue is full. Cannot add to it")

    def __len__(self):
        return self.__size

    def isEmpty(self):
        return self.__size == 0

    def __str__(self):
        out = ""
        if len(self.__data) == 0:
            return ""
        for i in range(len(self.__data)):
            out += "[{}] ".format(i) + str(self.__data[i])
            if i == self.__head:
                out += " <-- Head"
            if i == self.__tail:
                out += " <-- Tail"
            out +="\n"
        return out
```

```python
if __name__ == "__main__":
    CQ = CircularQueue(10)
    print(CQ.dequeue())
    text = "HELLO W"
    text2 = "IKIPEDIA"
    for t in text:
        CQ.enqueue(t)

    print(CQ)
    out_txt = ""
    for i in range(6):
        out_txt += str(CQ.dequeue())


    print(CQ)
    print(out_txt)
    for t in text2:
        CQ.enqueue(t)
    print(CQ)
    while not CQ.isEmpty():
        out_txt += str(CQ.dequeue())
    print(out_txt)
    print(CQ)
```

QUEUE

% Returns **True** if queue is empty
boolean isEmpty()

% Returns the size of the queue
int size()

% Inserts v at the end of the queue
enqueue(OBJECT v)

% Extracts q from the beginning of the queue
OBJECT dequeue()

% Reads the element at the top of the queue
OBJECT top()

```
None
[0] H <-- Head
[1] E
[2] L
[3] L
[4] O
[5]
[6] W
[7] None <-- Tail
[8] None
[9] None

[0] H
[1] E
[2] L
[3] L
[4] O
[5]
[6] W <-- Head
[7] None <-- Tail
[8] None
[9] None

HELLO
[0] P
[1] E
[2] D
[3] I
[4] A
[5]   <-- Tail
[6] W <-- Head
[7] I
[8] K
[9] I

HELLO WIKIPEDIA
[0] P
[1] E
[2] D
[3] I
[4] A
[5]   <-- Head <-- Tail
[6] W
[7] I
[8] K
[9] I
```

Exercise 1

Consider the following code (where s is a list of n elements). What is its complexity?
Note: res is a string!

```
def reverse(s):
  n = len(s)-1
  res = ""
  while n >= 0:
    res = res + s[n]
    n -= 1
  return res
```

Exercise 1

Consider the following code (where s is a list of n elements). What is its complexity?
Note: res is a string!

```
def reverse(s):
  n = len(s)-1
  res = ""
  while n >= 0:
    res = res + s[n]
    n -= 1
  return res
```

$\boxed{\text{Complexity: } \Theta(n^2)}$

- n string sums
- Each sum copies all the characters in a new string

strings are immutable!

Exercise 2

Consider the following code (where s is a list of n elements). What is its complexity?

```
def reverse(s):
  res = []
  for c in s:
    res.insert(0, c)
  return "".join(res)
```

Exercise 2

Consider the following code (where s is a list of n elements). What is its complexity?


```
def reverse(s):
  res = []
  for c in s:
    res.insert(0, c)
  return "".join(res)
```

$\boxed{\text{Complexity: } \Theta(n^2)}$

- n list inserts
- Each insert moves all characters one position up in the list

Exercise 3

Consider the following code (where s is a list of n elements). What is its complexity?

```
def reverse(s):
  n = len(s)-1
  res = []
  while n >= 0:
    res.append(s[n])
    n -= 1
  return "".join(res)
```

Exercise 3

Consider the following code (where s is a list of n elements). What is its complexity?


```python
def reverse(s):
  n = len(s)-1
  res = []
  while n >= 0:
    res.append(s[n])
    n -= 1
  return "".join(res)
```

Complexity: $\Theta(n)$

- n list append
- Each append has an amortized cost of $O(1)$

Note that: “”.join(res) has complexity O(n)

Better solution

```python
def reverse(s):
  return s[::-1]
```

Exercise 4

Consider the following code (where L is a list of n elements). What is its complexity?

```
def deduplicate(L):
  res=[]
  for item in L:
    if item not in res:
      res.append(item)
  return res
```

Exercise 4

Consider the following code (where L is a list of n elements). What is its complexity?


```
def deduplicate(L):
  res=[]
  for item in L:
    if item not in res:
      res.append(item)
  return res
```

Complexity: $\Theta(n^2)$

- n list append
- n checks whether an element is already present
- Each check costs $O(n)$

Exercise 5

Consider the following code (where L is a list of n elements). What is its complexity?

```
def deduplicate(L):
  res=[]
  present=set()
  for item in L:
    if item not in present:
      res.append(item)
      present.add(item)
  return res
```

Exercise 5

Consider the following code (where L is a list of n elements). What is its complexity?


```
def deduplicate(L):
  res=[]
  present=set()
  for item in L:
    if item not in present:
      res.append(item)
      present.add(item)
  return res
```

Complexity: $\Theta(n)$

- n list append
- n checks whether an element is already present
- Each check costs $O(1)$

Other possibility – destroy original order

```
def deduplicate(L):
  return list(set(L))
```