

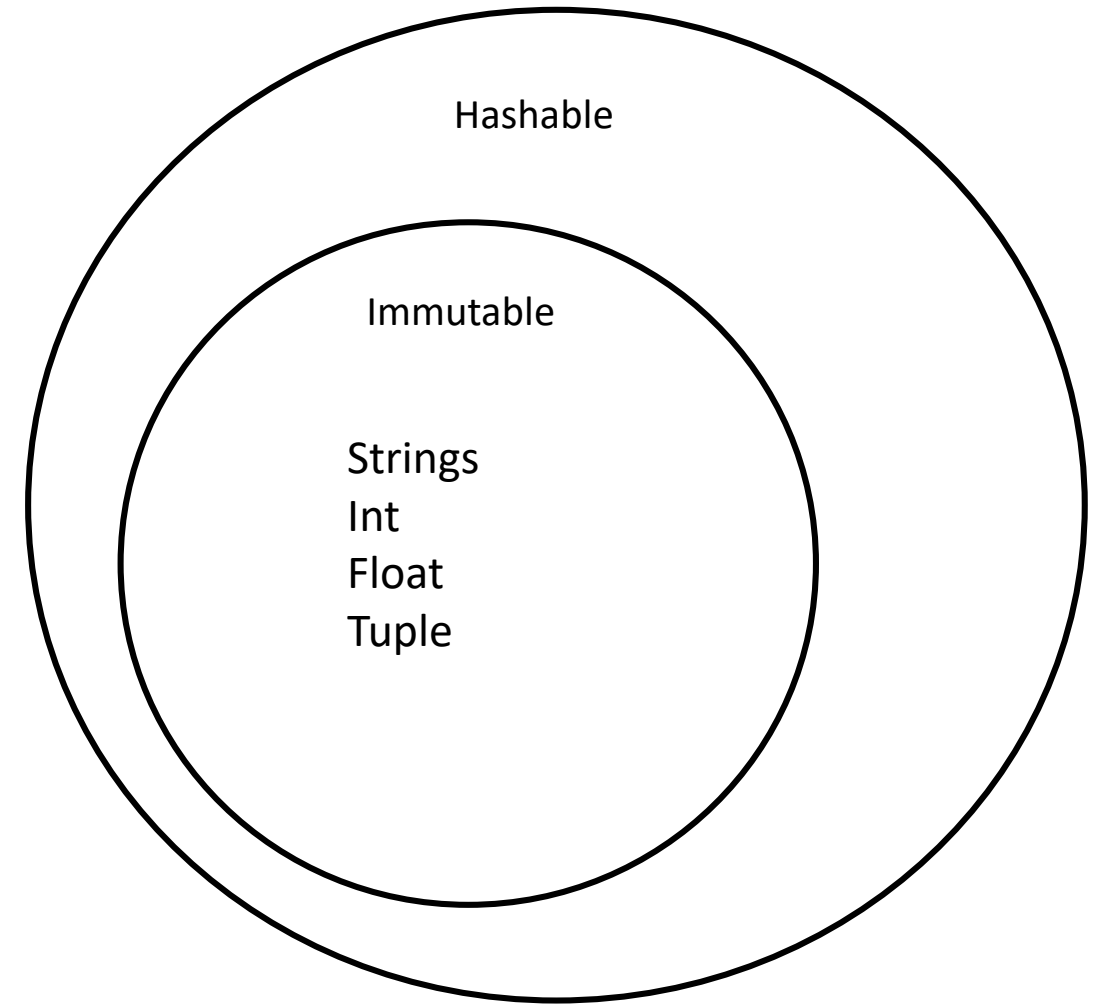
IT5001 Software Development Fundamentals

8. Sequences Contd.

Sirigina Rajendra Prasad

Python: Hashability and Immutability

Data Type	Immutable	Hashable
Integer	Yes	Yes
Float	Yes	Yes
String	Yes	Yes
Tuple	Yes	Yes
List	No	No
Set	No	No
Dictionary	No	No



All immutable objects are Hashable but not vice-versa

Returning Multiple Values

```
def f():  
    return 1, 2
```

→ Treated as a tuple

```
x = f()  
print(x)  
print(type(x))  
x, y = f()  
print(x, y)  
print(type(x), type(y))
```

→ Unpacking of tuple

Output:

```
(1, 2)  
<class 'tuple'>  
1 2  
<class 'int'> <class 'int'>
```

If you don't know number of arguments

Unknown number of positional arguments

```
def f(*args):  
    print(f'Type of input arguments: {type(args)}')  
    for i in args:  
        print(i)  
  
f(1,2,3,4)
```

Output:

```
Type of input arguments: <class 'tuple'>  
1  
2  
3  
4
```

If you don't know number of arguments

Unknown number of keyword arguments

```
def f(**kwargs):  
    print(f'Type of input arguments: {type(kwargs)}')  
    for i,j in kwargs.items():  
        print(f'{i} = {j}')  
  
f(arg_1 = 1, arg_2 = 2, arg_3 = 3)
```

Output:

```
Type of input arguments: <class 'dict'>  
arg_1 = 1  
arg_2 = 2  
arg_3 = 3
```

Accessing Global Variables

- Can a function access (read) a global variable?
 - Variable is Immutable
 - Yes
 - Variable is Mutable
 - Yes

```
def my_func_1(y):  
    return x+y  
  
x = 2  
print(x)  
print(my_func_1(4))  
print(x)  
  
x = [2,3]  
print(x)  
print(my_func_1([1,4]))  
print(x)
```

Output:

```
2  
6  
2  
[2, 3]  
[2, 3, 1, 4]  
[2, 3]
```

Modifying Global Variables

- Can a function modify a global variable with variable declared **global** within local function?
 - Yes, for both mutable and immutable variables

```
def my_func_2(y):  
    global x  
    x = x+y  
    return x
```

```
x = 2  
print(x)  
my_func_2(3)  
print(x)  
  
x = [2,3]  
print(x)  
my_func_2([4,5])  
print(x)
```

```
x = (2,3)  
print(x)  
my_func_2((4,5))  
print(x)
```

Output:

```
2  
5  
[2, 3]  
[2, 3, 4, 5]  
(2, 3)  
(2, 3, 4, 5)
```

Modifying Global Variables

- Can a function modify a global variable with variable **not** declared as global within local function?
 - No, for immutable variables
 - Yes, for mutable variables with **only** the methods that **mutate data** (append, sort, etc.)

```
def my_func_3(y):  
    return x.append(y)
```

```
x = [2, 3]  
print(x)  
print(my_func_3([4, 5]))  
print(x)
```

No assignment to variable *x*

Output:

```
[2, 3]  
None  
[2, 3, [4, 5]]
```


Scope

- Passing a mutable variable as argument

Python 3.6
([known limitations](#))

```
1 def my_func_4(y):  
2     print(y)  
→ 3     y.append(4)  
→ 4     print(y)  
5  
6 x = [2,3]  
7 print(x)  
8 my_func_4(x)  
9 print(x)
```

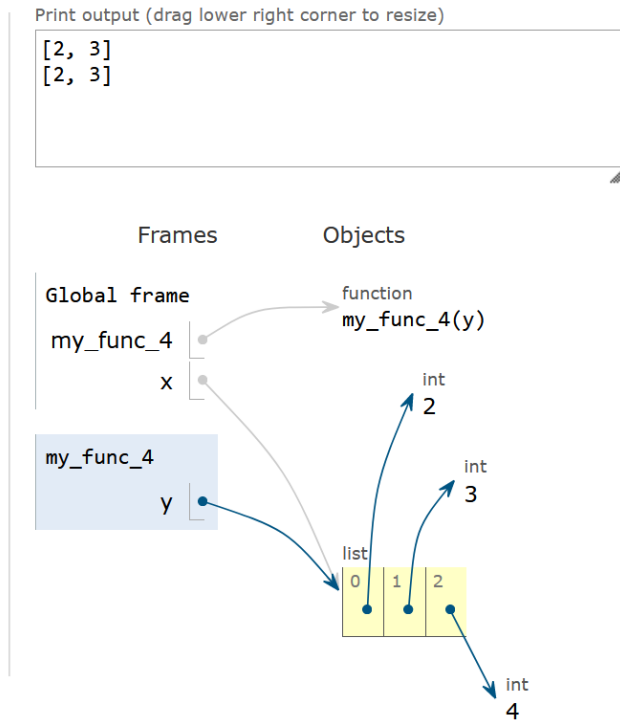
[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > Last >>

Step 8 of 10

[Customize visualization](#)



Output:

```
[2, 3]  
[2, 3]  
[2, 3, 4]  
[2, 3, 4]
```

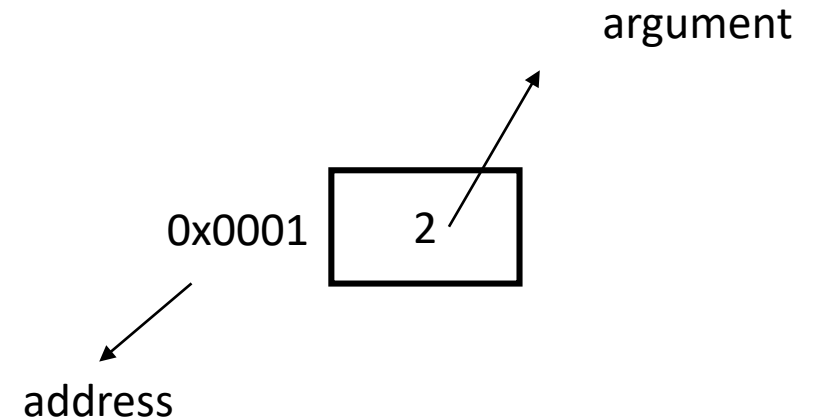
How are arguments passed to functions?

- Pass-by-Value

- An independent (duplicate) copy of argument is passed as input
- Not good if the argument size is very large
 - Requires additional memory and execution time

- Pass-by-Reference

- Address of argument is passed and function access the value from address
- Efficient for arguments of very large size



How about Python?

- Pass-by-Value or Pass-by-Reference?
 - Neither of them
- Python passes objects by assignment
 - Pass-by-Assignment
- Only exception is with *global* keyword

With **global** keyword

```
def square_1():  
    global x  
    x = x**2
```

equivalent to pass-by-reference

```
x=2  
square_1()
```

Pass-by-Assignment

- Best Practice
 - Pass value by assignment, modify, and reassign

```
def square_2(y):  
    return y**2
```

```
x = 2  
x = square_2(x)
```

```
def modifyTup(t):  
    return t + (999,)
```

```
>>> tup = (1, 2, 3, 4, 5)  
>>> tup = modifyTup(tup)
```

Pass-by-Assignment

- For mutable variables

```
def my_func_1(y):  
    return x+y
```

```
x = [2,3]  
print(x)  
print(my_func_1([1,4]))  
print(x)
```

Effect is similar to pass-by-value

```
def my_func_3(y):  
    return x.append(y)
```

```
x = [2,3]  
print(x)  
print(my_func_3([4,5]))  
print(x)
```

Effect is similar to pass-by-reference

Scope: Summary

- Parameter Passing
 - Pass-by-Assignment
- Immutable Variable
 - Can access global variable
 - Cannot modify global variable unless declared global
- Mutable Variable
 - Can access global variable
 - Can modify the variable
 - Need to use **global** keyword for methods that do not mutate objects
 - No need of **global** keywords for methods that mutate objects

How are lists resized with *append*?

Ternary operator in C

```
additional_allocated = current_size >> 3 + current_size < 9 ? 3 : 6
```

floored-division by 8

$$\left\lfloor \frac{current_size}{8} \right\rfloor$$

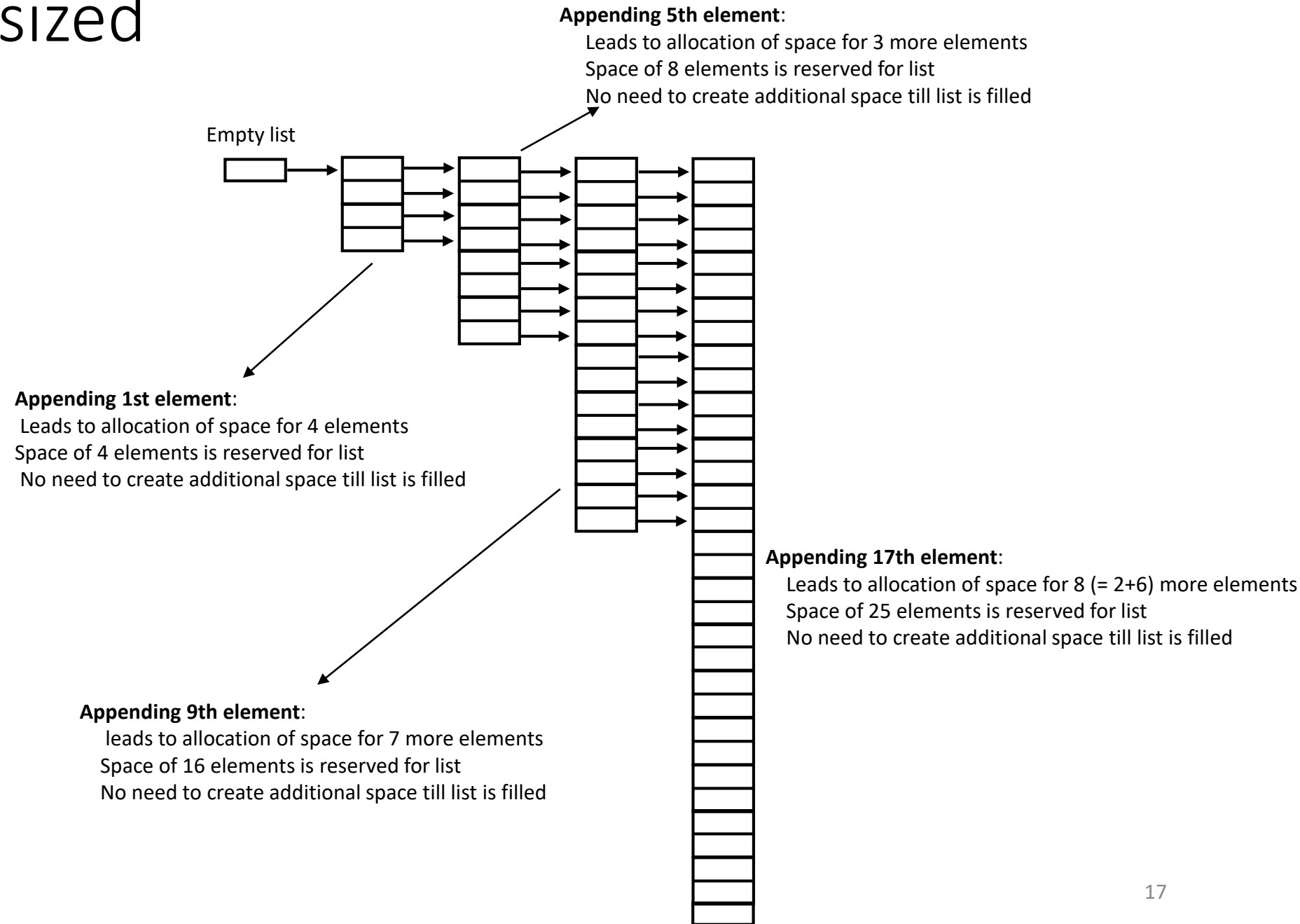
if current_size < 9:

3

else:

6

How are lists resized with *append*?

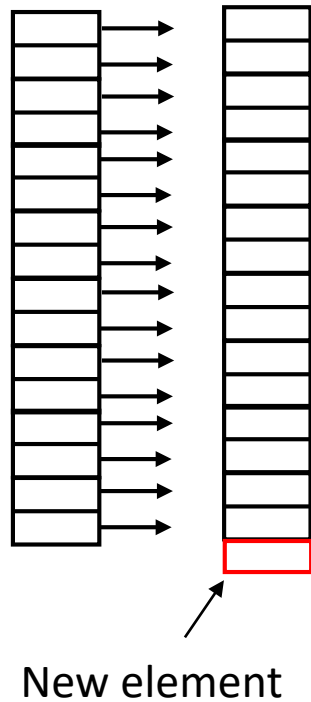


How are lists resized with *append*?

```
import sys
my_list = []
my_list_length = []
my_list_size = []
limit= 100
for k in range(limit):
    my_list_length.append(len(my_list))
    my_list_size.append(sys.getsizeof(my_list))
    my_list.append(k)
```

How are lists resized with *concatenation*?

- Always create a new list
 - Copies the contents of old list and new list combined



List Append Vs Concatenation

```
from time import time
import sys
my_list_append = []
my_list_append_length = []
my_list_append_size = []

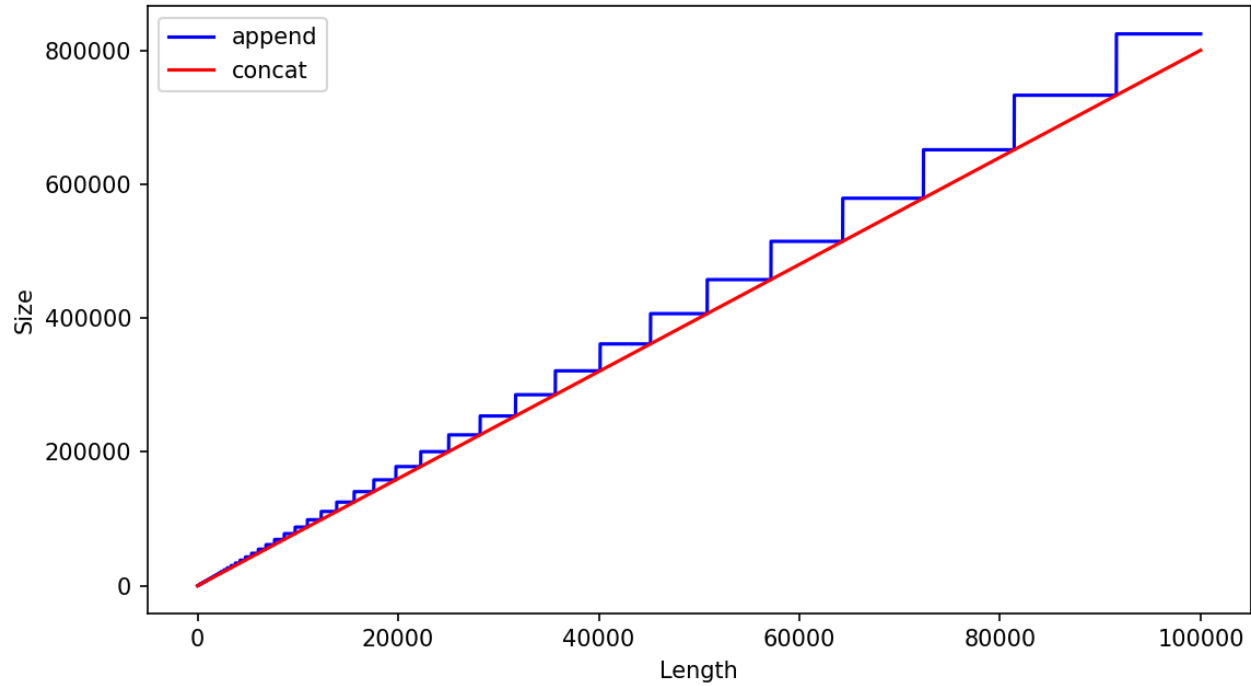
start= 0
end= 10**5

append_start_time = time()
for k in range(start,end):
    my_list_append_length.append(len(my_list_append))
    my_list_append_size.append(sys.getsizeof(my_list_append))
    my_list_append.append(k)
append_end_time = time()

my_list_concat = []
my_list_concat_length = []
my_list_concat_size = []
concat_start_time = time()
for k in range(start,end):
    my_list_concat_length.append(len(my_list_concat))
    my_list_concat_size.append(sys.getsizeof(my_list_concat))
    my_list_concat = my_list_concat + [k]
concat_end_time = time()
```

```
print(f'Append: {round(append_end_time-append_start_time,4)}s')
print(f'Concatenation: {round(concat_end_time - concat_start_time,4)}s')
from matplotlib import pyplot
pyplot.plot(my_list_append_length, my_list_append_size,color = 'b')
pyplot.plot(my_list_concat_length, my_list_concat_size,color='r')
pyplot.xlabel('Length')
pyplot.ylabel('Size')
pyplot.legend(['append', 'concat'])
pyplot.show()
```

List Append Vs Concatenation

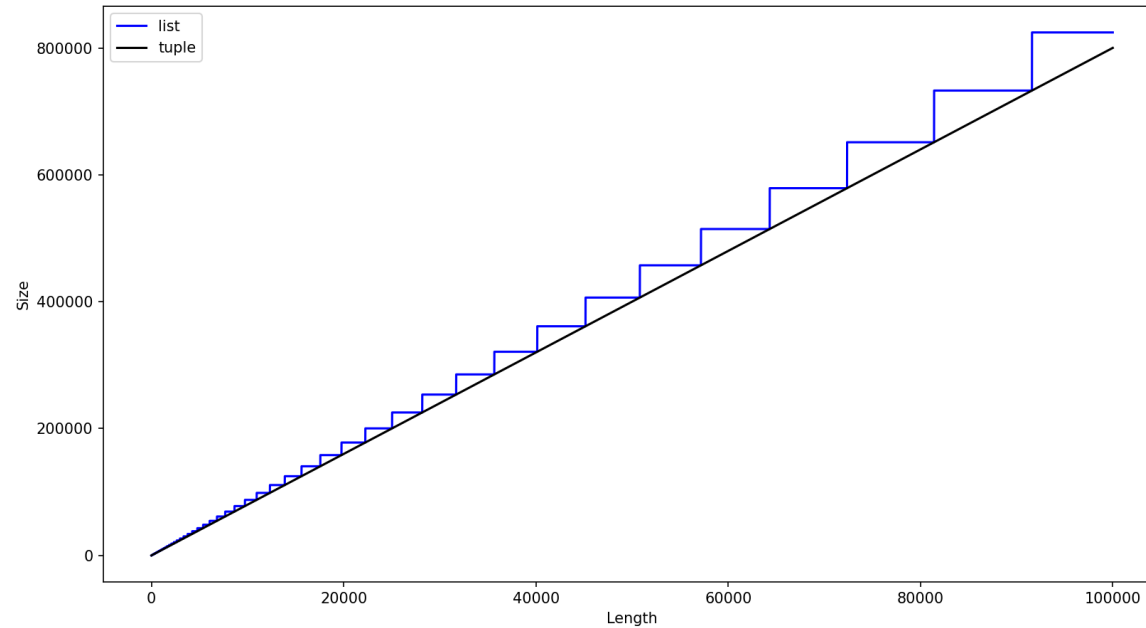


Append: 0.0316s

Concatenation: 20.7656s

Concatenation is slow as it creates new list for every new concatenation

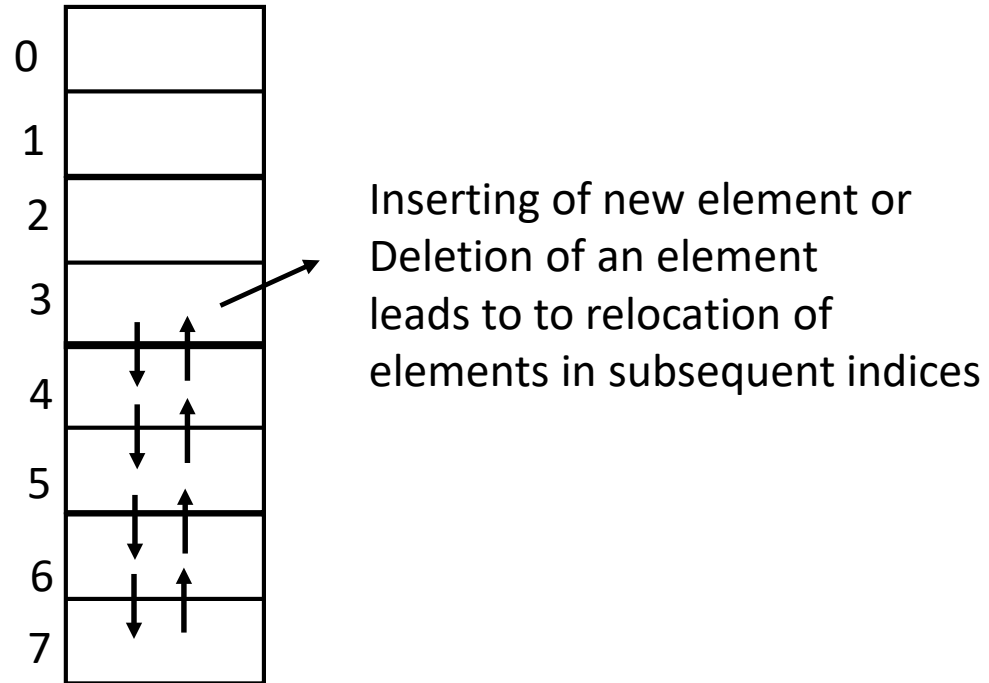
List Append Vs Tuple Concatenation



Append: 0.0211s
Concatenation: 20.3768s

Modifying tuple is slow as it creates new tuple for every new concatenation

Insertion/Deletion in lists?



List Vs Tuple: Conclusion

Tuple Version



Preserve input



Must return result



Slower

List Version



May or may not preserve input

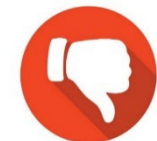
- May modify input



May or may not need to return result



Faster (append)



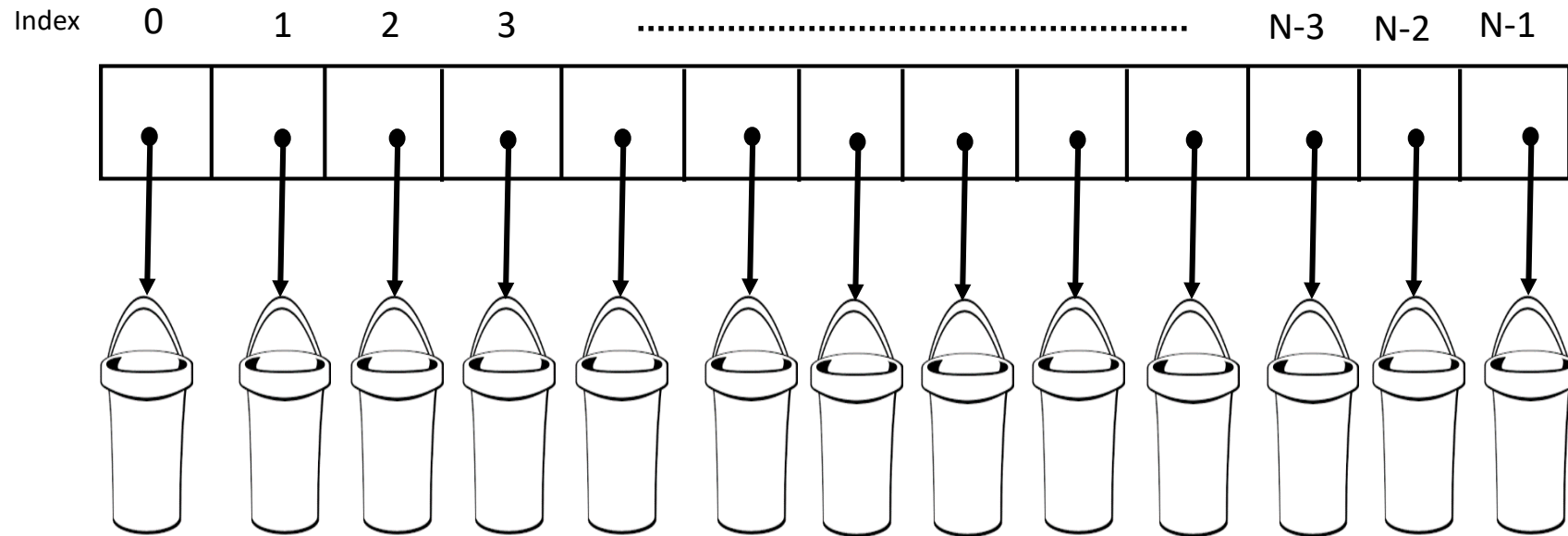
Slower (concatenation)

Miscellaneous

Dictionary – Bucket Array

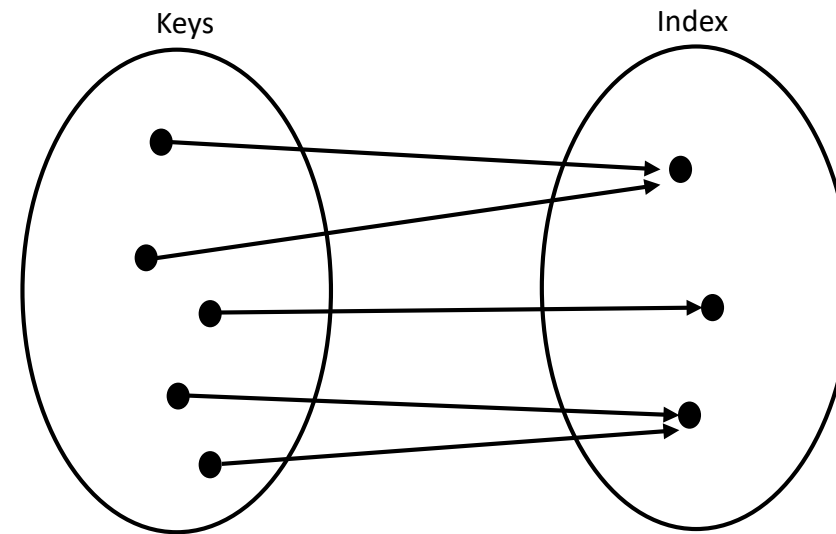
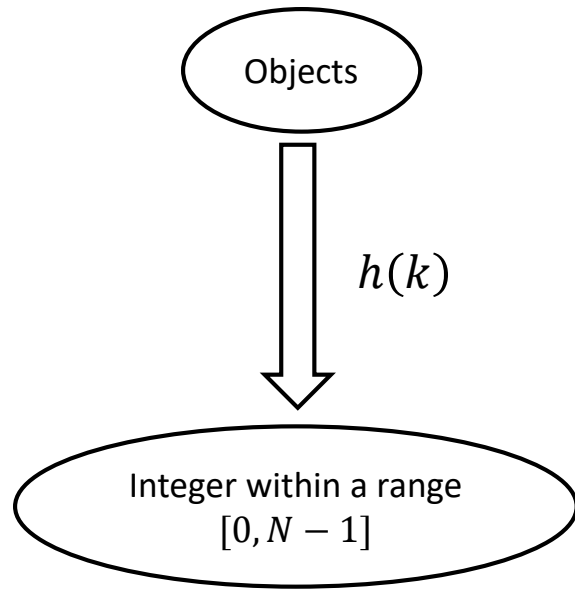
$$A(h(k)) = v$$

Key (k) is mapped to an index (0 to N-1) and Value (v) is stored in the corresponding bucket

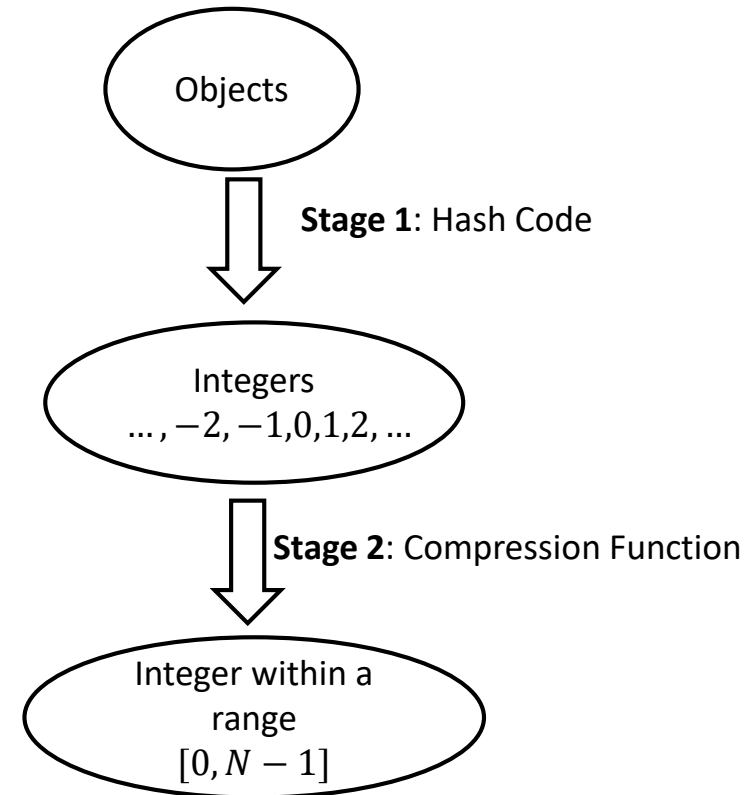
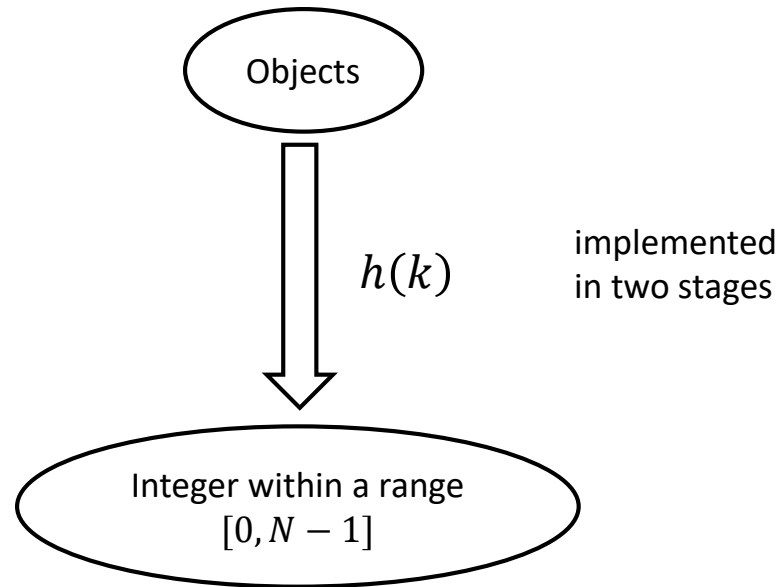


Key to index mapping is done through Hash Function ($h(k)$)

Hash Function - $h(k)$



Hash Function - $h(k)$



Hash Code

- Bit representation as hash codes
- Polynomial hash codes
 - $x_0a^{n-1} + x_1a^{n-2} + \dots + x_{n-2}a + x_{n-1}$
 - x_i : Coefficients
 - a : Constant
- Cyclic-shift hash codes

Compression Functions

- Division Method $i \bmod N$

- Multiply-Add-and-Divide (MAD) Method

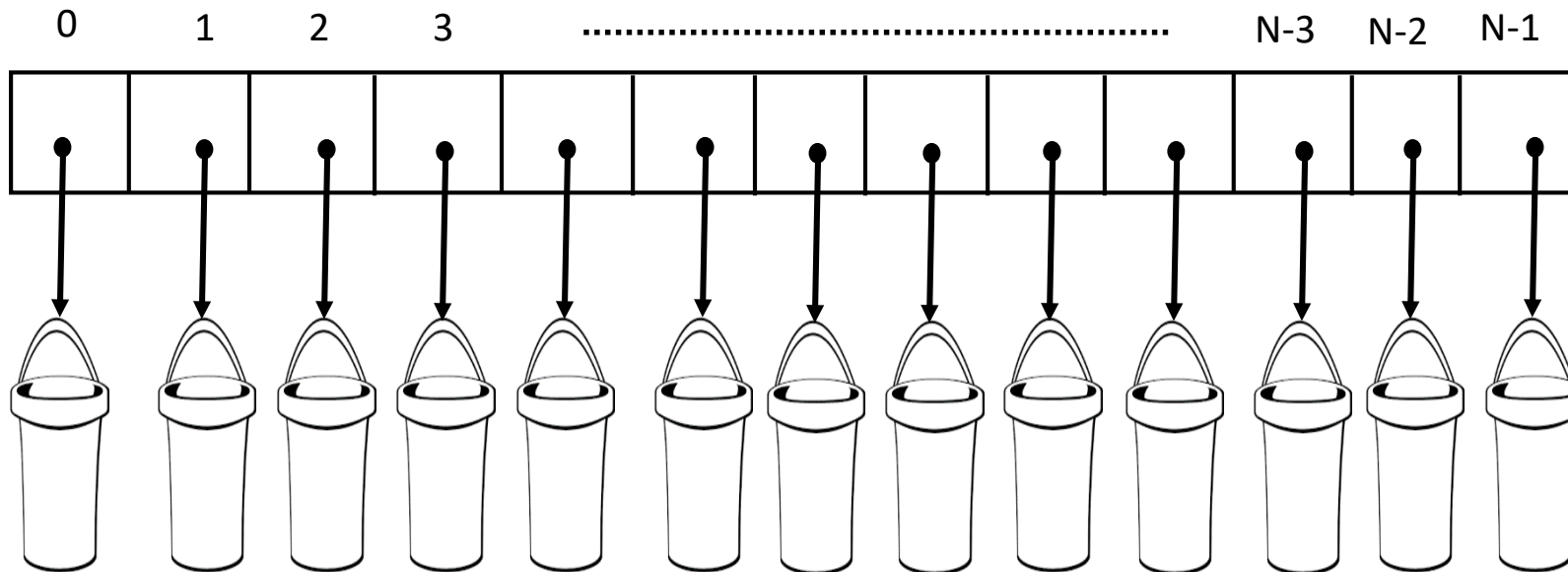
$$[(ai + b) \bmod p] \bmod N$$

Collisions

Collision occurs if multiple keys produce same hash value



$$h(k_1) = h(k_2)$$

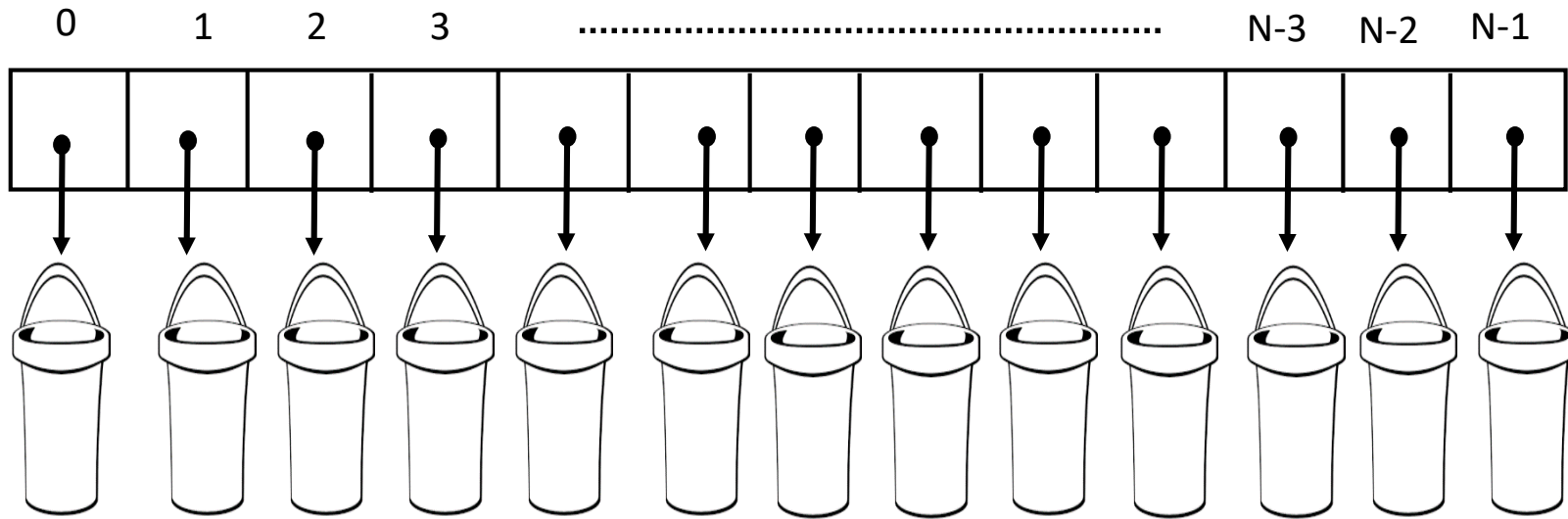


Collision-Handling

- Separate Chaining
- Open Addressing
 - Linear Probing
 - Quadratic Probing
 - Double Hashing

Collision-Handling

- Separate Chaining



- Each bucket contains a list of values v_i whose $h(k_i)$ are same
 - Requires additional *list* data structure
 - Slows down the access as the *list* need to be searched for the key

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing