

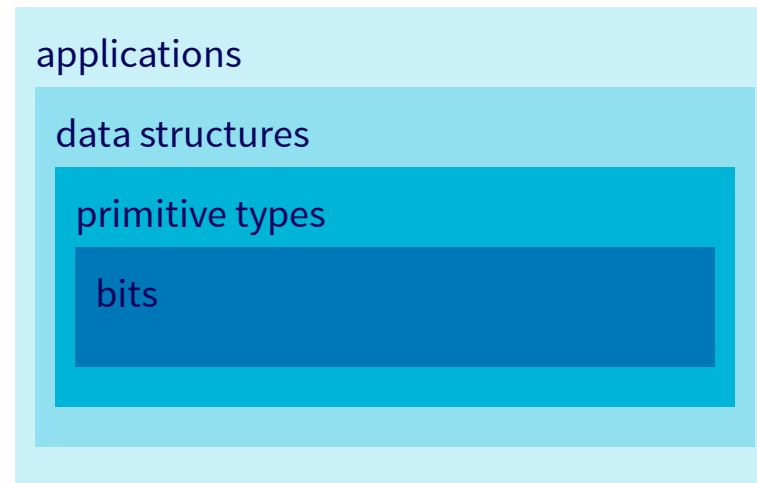
Data structures



A **data structure** is a data organization and storage format.
Usually chosen for **efficient** access to data.



from bits to applications

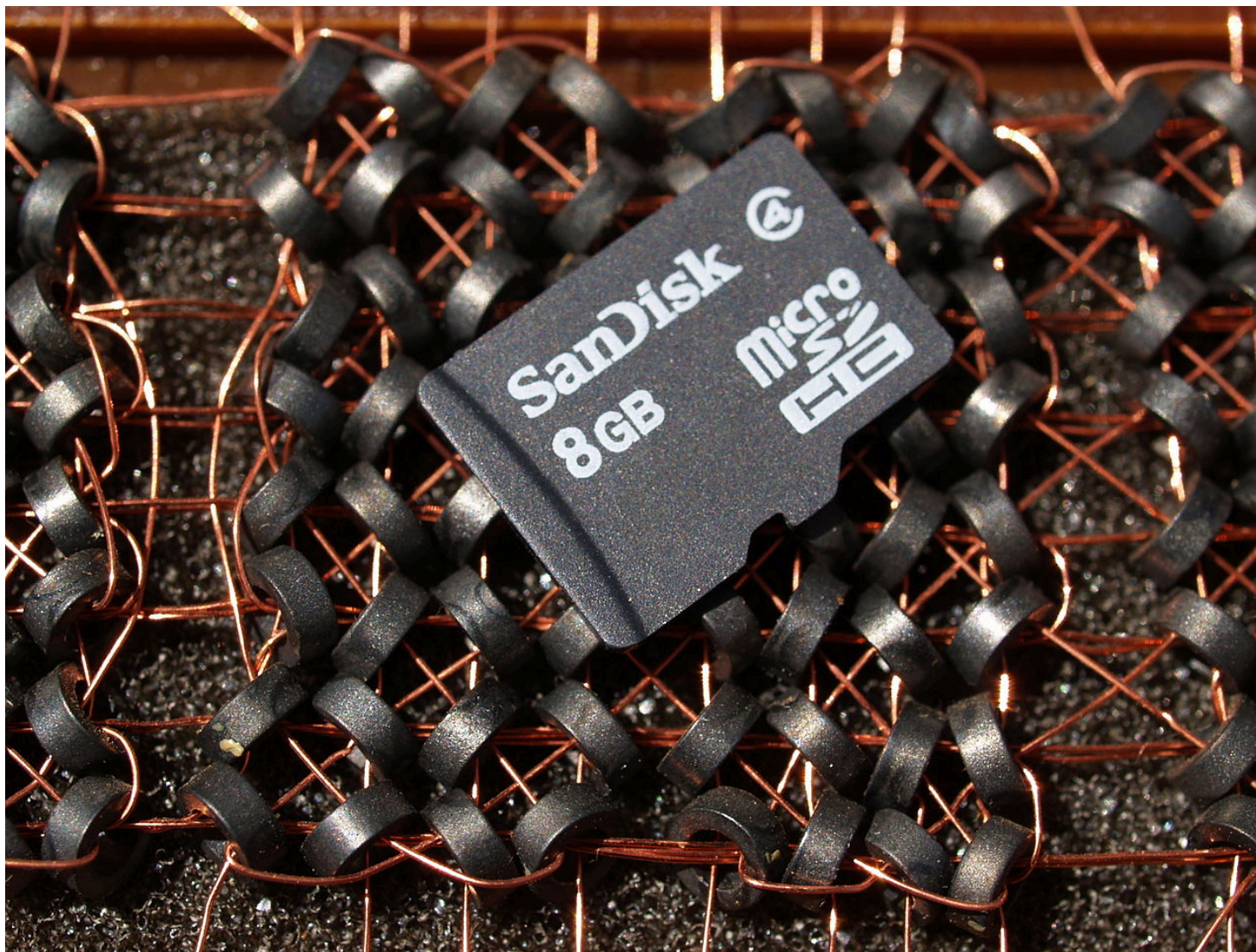


... a story of packing things 

bits & primitive types



bits form computer memory



Core & SD memory by [Daniel Sancho](#) from Málaga, Spain, CC BY 2.0

Generic Software Skills | CC-BY 4.0

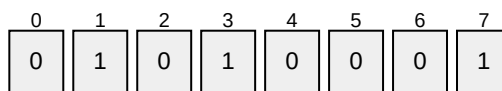


bits & primitive types

- a computer works on bits (which are either 0 or 1)
- bits themselves have a position, but no particular meaning
- bits are usually grouped into bytes, which consist of 8 bits
- (primitive) types provide meaning to groups of bits or bytes

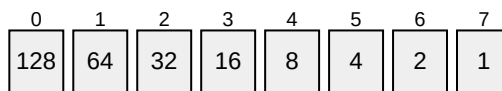


primitive types: unsigned integer



8 bits of the value 81

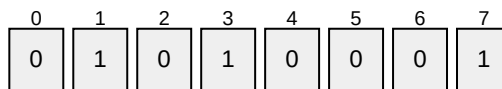
interpretation:



bit values of an 8 bit unsigned integer

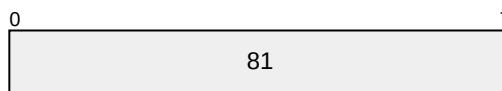
$$0 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 81$$

primitive types: unsigned integer



8 bits of the value 81

used in upcoming slides:

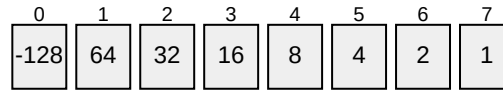


showing value



using hexadecimal

primitive types: signed integer



bit values of an 8 bit signed integer

note, it's -128!

this representation is called two's complement

- the most commonly used
- simplifies hardware implementations

another (very uncommon, but obvious) option:



sign and value representation

primitive types: float (IEEE 754)



32 bit float (single precision)

$$\text{value} = (-1)^{\text{sign}} \cdot 2^{\text{exponent} - 127} \cdot (1 + \text{fraction})$$

- **exponent** is an unsigned integer
- **fraction** is a binary fraction:



bit values of the fraction

primitive types: characters

a character set (e.g. [ASCII](#)) assigns a different meaning to each stored number:

decimal	hex	meaning
10	0x0a	new line
48	0x30	0
63	0x3f	?
65	0x41	A
97	0x61	a

(these days we use [UTF-8](#) instead of ASCII, which uses more than one byte, such that characters of all languages can be represented)



representation vs value

representation: the bit-pattern (`0b01010001` or `0x51`)

value: the interpretation (81 or `Q`)

The same bit-pattern may be interpreted differently depending on the type!



Hands On

Take the bit-representation of a 32 bit float value of `1.0` and re-interpret it as an integer. Which value do you get? What's the bit-representation?

- In Python, you may want to use `numpy` as this provides more fine-grained control about the exact primitive types.
- A 32 bit float has the dtype `<f4`, a 32 bit unsigned integer the dtype `<u4`.
- `.view(dtype)` can be used to re-interpret memory as different type.



solution

```
1 import numpy as np
2 i = int(np.array(1.0, dtype="<f4").view("<u4"))
3 print(f"{i} {i:032b}")
```

```
1065353216 0011111111000000000000000000000000000000
```

```
1 i = int(np.array(1.5, dtype="<f4").view("<u4"))
2 print(f"{i} {i:032b}")
```

```
1069547520 0011111111000000000000000000000000000000
```



compounds



memory locations

Memory is usually addressed (numbered) byte-wise.

From now, we'll use byte-addressing instead of bit-addressing in the figures.

compounds

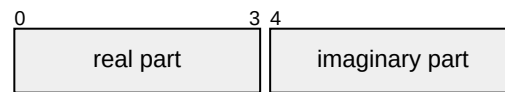
Multiple bits form primitive types.

Multiple primitive types form compound types

`struct` or `@dataclass`, `tuple` etc... form a **fixed** group of elements, each with a **distinct** meaning



compound: complex number



2x4 byte (32 bit) complex number

$$\text{value} = \text{real part} + i \cdot \text{imaginary part}$$

```
1 from dataclasses import dataclass
2
3 @dataclass
4 class ComplexNumber:
5     real: float
6     imag: float
```

it's about co-location

The groupings of bits and bytes we've seen so far use **fixed**, pre-defined and agreed-on sequences of **co-located memory locations** to **provide meaning** to those memory locations. To make use of the bits, we have to **know what to expect**.



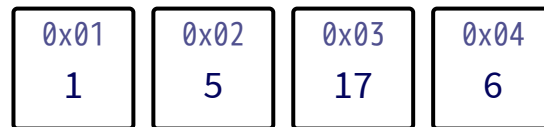
other relations

Are there other (less fixed) ways to **group** and **provide meaning** to memory locations?

- computed addresses (e.g. array)
- tabulated addresses (e.g. pointers)

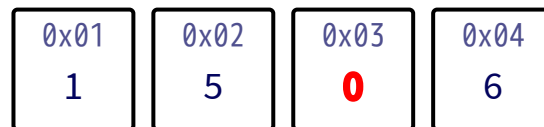


Array



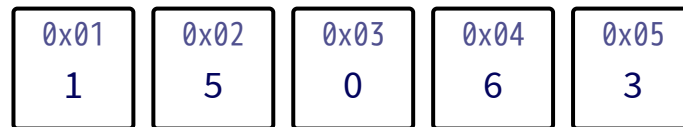
a series of consecutive memory cells

Array



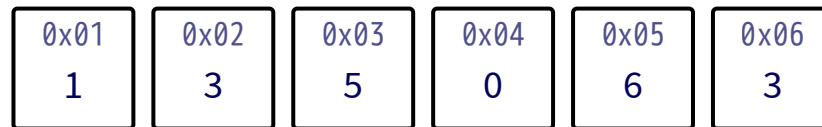
element access is easy

Array



appending: easy if free space available

Array



insert or removal is expensive

Array

Arrays are containers, which store data elements in consecutive memory cells.

- There's no overhead, just data.
- Accessing elements is cheap.
- Modifying the structure / order etc.. can be expensive.
- Some extra metadata may be added for convenience.

other names for similar things: vector, string



Array metadata

What metadata might be relevant for arrays?

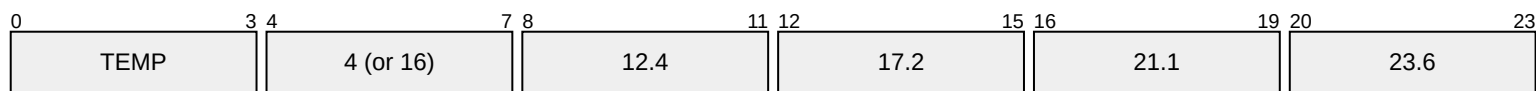
e.g.:

- start location
- size (length / end / sentinel)
- element size
- data type



Type-Length-Value (TLV)

TLV is a typical data structure for data acquisition ^{1 2}



TLV temperature data

- (fixed size) type tag (`TEMP` for temperature as `float`)
- (fixed size) size of data (4 values, alternative: 16 bytes)
- (variable size) actual data values

TLV is a combination of a `struct` and an array.

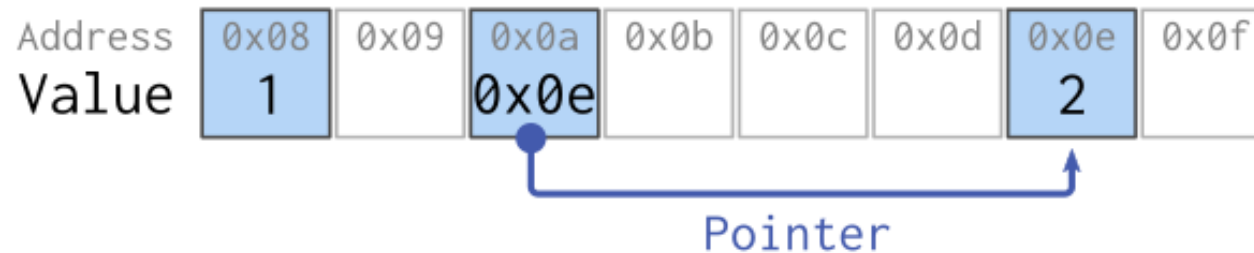
1. [METEK cloud radars](#) for raw data.
2. [CBOR](#) and [msgpack](#) use similar concepts for arbitrary data



Pointers and References

An integer (stored in memory) can be interpreted as a byte-address in memory (where another variable is located).

This is called a **pointer**.



The term “**reference**” is used for variable names which act like pointers, but hide their pointer-like nature.

references in Python

Python usually doesn't expose pointers directly, but references are everywhere.

```
1 a = [5]    # a is a reference to the list
2 print(a)
```

[5]

```
1 b = a      # b is another reference to the SAME list
2 b[0] = 10
3 print(a)   # NOTE, it's a again!
```

[10]



pointers & references form indirections

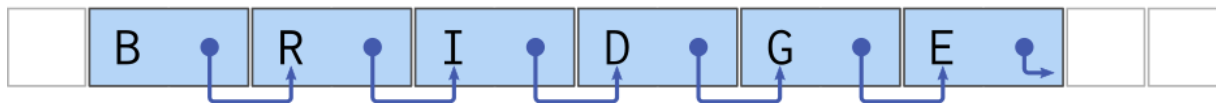
Pointers (and thus references) can be used to form more complex relations and data structures.

More Containers



(Linked) lists

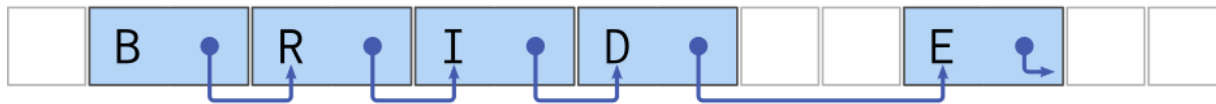
- Variable size, can but needn't be contiguous in memory



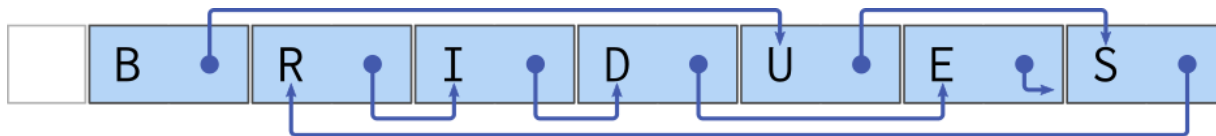
- Next element is at the address saved in the additional pointer
- The previous element can only be found by starting at the beginning

(Linked) lists

- Element removal only requires to change a pointer address



- Element insertion only requires allocating memory for the new value and setting two pointers



(Linked) lists

- in theory, insert + remove is cheaper than on an array
- in practice, memory allocation can be more expensive than moving data
- Python's `list` is more like an array of pointers



Dictionaries

also called associative array, key-value store, map(ping), object etc...

Access elements by key (e.g. name) instead of position

```
1 temperatures = {'hamburg': 21, 'madrid': 37, 'reykjavik': 10}  
2 print(temperatures['hamburg'])
```

21



use dictionaries to group things

```
1 items = ["knife", "fork", "hammer", "nail"]
2 by_length = {}
3 for element in items:
4     l = len(element)
5     if l not in by_length: by_length[l] = []
6     by_length[l].append(element)
7
8 for l, elements in sorted(by_length.items()):
9     print(l, elements)
```

```
4 ['fork', 'nail']
5 ['knife']
6 ['hammer']
```



use dictionaries to group things

```
1 from collections import defaultdict
2 items = ["knife", "fork", "hammer", "nail"]
3 by_length = defaultdict(list)
4 for element in items:
5     by_length[len(element)].append(element)
6
7 for l, elements in sorted(by_length.items()):
8     print(l, elements)
```

```
4 ['fork', 'nail']
5 ['knife']
6 ['hammer']
```



building a dictionary

Key space is often (infinitely) large (e.g. all possible strings)

There can't be one "slot" for every possible key

Let's make a list of keys and values:

```
1 temperatures = [('hamburg', 21), ('madrid', 37), ('reykjavik', 10)]
2
3 def get(kvlist, key):
4     for k, v in kvlist:
5         if k == key:
6             return v
7     raise KeyError(f"key '{key}' not found!")
8
9 print(get(temperatures, 'hamburg'))
```

21



building a dictionary

A dictionary as a list works and is useful for data exchange, but getting an element requires accessing all of them (slow).

Better options for *efficient* access:

- (binary) tree
- hash map



Set

Sets are dictionaries without values.

```
1 {"a", "b", "c", "a"}
```

```
{'a', 'b', 'c'}
```

Combined data structures

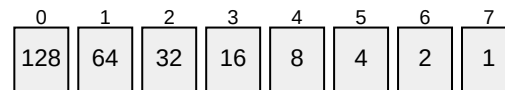
```
1 people = [  
2     {  
3         "givenname": "Jane",  
4         "name": "Doe",  
5         "skills": ["Python", "C++", "Clouds"],  
6     },  
7     {  
8         "givenname": "John",  
9         ...  
10    }  
11    ...  
12 ]
```

Shotgun Buffet

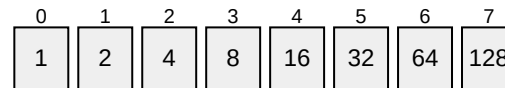


Bit Orderings

Depending on the computer architecture or application, the ordering of the bits may be reversed:



MSB first 8 bit integer

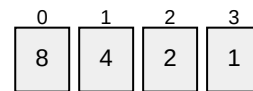


LSB first 8 bit integer

binary-coded decimal (BCD)

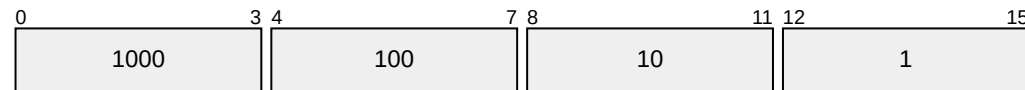
BCD numbers use four bits to encode a digit (0 ... 9).

The possible values 10 ... 15 are unused.



4 bit integer

These digits can be combined to form a decimal number.



4 digit BCD number

(it's mostly an ancient and wasteful thing, but reduces complexity for transformations between bit representation and decimal value interpretation)

Stack

Only add or remove elements at the end (*LIFO*: last in, first out)

```
1 stack = ['this', 'is']  
2 print(stack)
```

```
['this', 'is']
```

```
1 stack.append('new')  
2 print(stack)
```

```
['this', 'is', 'new']
```

```
1 stack.pop()  
2 stack.append('changed')  
3 print(stack)
```

```
['this', 'is', 'changed']
```

(i.e. like an array with enough space at the end)

Queue

Add at the end, remove at the front (*FIFO*: first in, first out)

```
1 from queue import Queue
2
3 queue = Queue()
4 queue.put('First')
5 queue.put('Second')
6 print(list(queue.queue))
```

```
['First', 'Second']
```

```
1 print(queue.get())
```

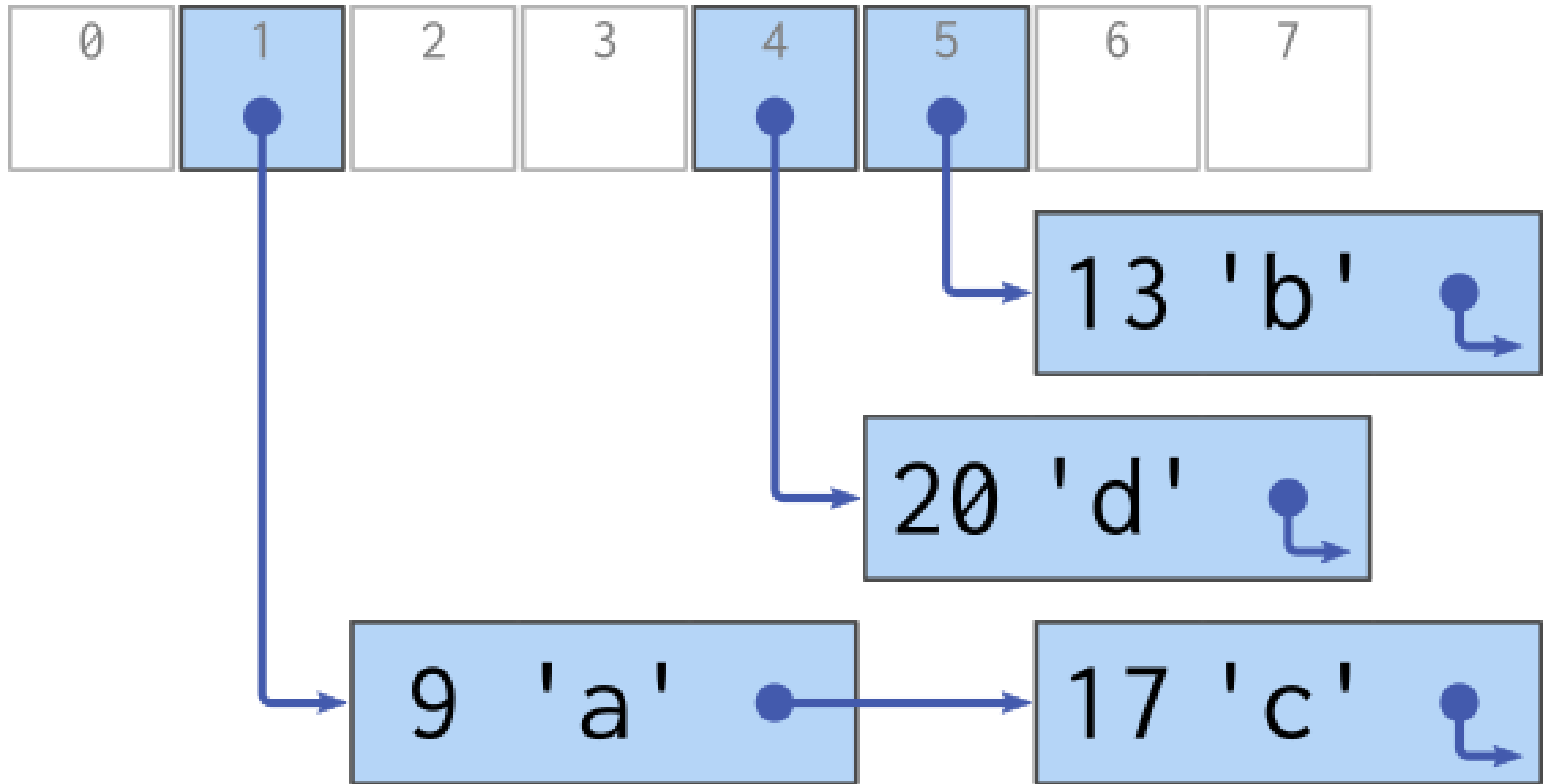
```
First
```

```
1 queue.put('Third')
2 print(list(queue.queue))
```

```
['Second', 'Third']
```

can be implemented using an array as a [circular buffer](#)

Hash Map



hash map