ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Basic Data Structures

Prof. Dr. Goran Glavaš

30.10.2023

Content

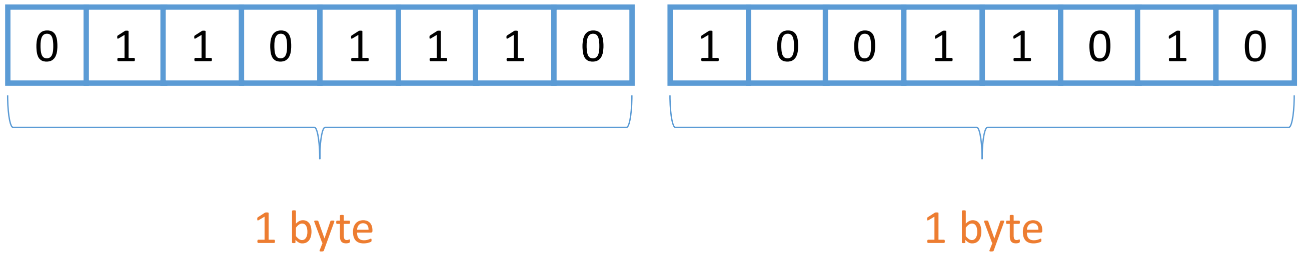
- Primitive Data Types
- Abstract Data Types
  - List
  - Stack
  - Queue

Primitive Data Types: Integer

- Several **universal primitive data types**
  - Exist in all programming languages

1. Integer

- In some programming languages, several subtypes of the integer type
- short: typically allocated 2 bytes of memory
- int: typically allocated 4 bytes of memory
- long: typically allocated 8 bytes of memory



Primitive Data Types: Integer

- Several **universal primitive data types**
  - Exist in all programming languages

1. Integer

- Range of integers covered by the concrete type, depends on the number of bytes allocated to the type
- That can vary across programming languages
- 1 byte = 8 bits. N bytes = 8N bits = 2^{8N} **different numbers**
- But we have both positive and negative integers
- N bytes gives the range of [-2^{8N-1}, $-2^{8N-1} - 1$]
  - For N = 2, we can represent numbers from -32768 (-2^{15}) to 32767 ($2^{15} - 1$)

Primitive Data Types: Float

- Several **universal primitive data types**
  - Exist in all programming languages

2. Float (short for „floating point")
  - Represents real numbers (e.g., 23.4 or -1.532343213)
  - Computer memory is limited (how many bytes for one number?)
    - We cannot store numbers with infinite precision (e.g., 1.333333333333...)
  - So for decimal numbers, we need to decide how much of our „bit budget" do we want to to **integer digits** and how many to **fraction digits**?

Primitive Data Types: Float

2. Float (short for „floating point")

- So for decimal numbers, we need to decide how much of the „bit budget" do we want to for **integer digits** and how many to **fraction digits**?

To an engineer building a highway, it does not matter if the road is 10 meters or 10.00001 meters wide → more capacity for **integer digits**

For a designer of a microchip 0.00001 makes a huge difference. At the same time, they never need numbers larger than 0.1: more capacity for **fraction digits**

A physicist needs numbers that capture distances in space (e.g., millions of km) as well as very small quantities (e.g., gravitational constant, $6.674...*10^{-11}$): large capacity for **both integer and fraction digits**

Primitive Data Types: Float

- If we want to satisfy all the different use cases, **a fixed split of bits** – some to integer digits, rest to fraction digits – won't work

- Solution: format called ***floating point (float)*** consisting of:
  - **Significand:** the number's digits (both from integer part and those from fraction)
  - **Exponent:** specifies where the decimal point is relative to the beginning of the significand
    - We assume the decimal point is after the first digit
    - The exponent then indicates how many places it needs to be moved!

Real number	Significand	Exponent	Exp. format
15.365	1.5365	1	$1.5365 * 10^{1}$
-300.5	-3.005	2	$-3.005 * 10^{2}$
0.00000000456	4.56	-9	$4.56 * 10^{-9}$
17340000000	1.734	10	$1.734 * 10^{10}$

Primitive Data Types: Float

- What is the **range of numbers** we can represent with a floating point?
  - Depends on number of bits/bytes assigned to **significand**
  - Depends on number of bits/bytes assigned to **exponent**

- Most programming languages support two floating point (sub)types
  - **Single precision** (just „float”)
  - **Double prediction** („double”)

Format	Total bits	Significand bits	Exponent bits	Smallest number	Largest number
Single precision	32	23 + 1 sign	8	ca. $1.2 \cdot 10^{-38}$	ca. $3.4 \cdot 10^{38}$
Double precision	64	52 + 1 sign	11	ca. $2.2 \cdot 10^{-308}$	ca. $1.8 \cdot 10^{308}$

Primitive Data Types: Boolean

- Several **universal primitive data types**
  - Exist in all programming languages

3. Boolean

  - This data type has only two possible values: true and false
  - **Q:** How much memory is needed to store one Boolean variable?

Primitive Data Types: Characters

- Several **universal primitive data types**
  - Exist in all programming languages

3. Character (or char) and string (not primitive)

- **Character encoding**: assigning numbers to graphical characters
- How much memory do we need for a character?
  - Depends on how many characters we want to encode/support

- **Strings**: sequences of characters
  - Technically not primitive data type
  - But in most programming languages it is predefined (effectively treated as a primitive) with a lot of built-in functionality for string manipulation

(Data) „Typing” in Programming Languages

- Each programming language has its own (data) type system
- **Strongly** vs. **Weakly** (loosely) typed
  - Colloquial classification, no strict definition
  - Strongly typed: **stricter typing** rules at compile time
    - Regarding variable assignment, function returns values, function arguments, etc.

```
int name = „anonymous” // will give compile error in C++
public int sum(int a, int b) {...}
sum(1, „student”) // will give compile error in Java
```

  - Weakly typed: looser type checking rules (at compile type or in interpreters)
    - looser typing rules/checks in advance
    - Type incompatibilities typically yield errors at runtime
    - Implicit (silent) type conversion may happen at runtime – can cause „nasty bugs”

(Data) „Typing" in Programming Languages

- Each programming language has its own (data) type system
- **Static** vs. **Dynamic** typing
  - Static typing: type checking performed at program compilation time
    - In strong, static typing no type errors should occur at runtime
  - Dynamic typing: type checking happens at runtime (during execution)
    - Values used at runtime classified into types
    - There are restrictions on how values of certain types can be (are allowed to be) used
    - If restrictions are violated, a (dynamic) type error occurs

Primitive Types in Python

- Integer, Float, Bool*, String*
  - *Strictly speaking, not primitive types in Python, implemented as classes
  - **`type()`** built-in function returns the type of any variable or constant

```
x = 32767
type(x)

w = 1.357e-12
type(w)

print(z == w)
print(type(z == w))
```

```
s = "Berlin, Germany"
print(type(s))

# slicing
a[:6], a[8:], a[3:11]
('Berlin', 'Germany', 'lin, Ger')

"lin" in s # True
s.startswith("Berlin") # True
s.endswith("Germany") # True
```

No-value-type („empty” variable)

- In most programming languages, there’s a reserved type indicating that a variable has no value

  - Java, C++, C#: **null** pointer
  - Python: NoneType (keyword None)
    - If evaluated directly as a condition, None value in Python gives **False** (same goes for an integer value of 0 and empty string ””)

```
x = None
type(x)
```

```
if x:
  print("True")
else:
  print("False")
```

```
y = 0
type(y)
```

```
if y:
  print("True")
else:
  print("False")
```

Content

- Primitive Data Types
- Abstract Data Types
  - List
  - Stack
  - Queue

Complex and Abstract Data Types

- **Complex data types**
  - Consist of primitive data types
  - Concrete complex data types are defined by a concrete programming language
- **Algorithms** are independent of programming languages
  - Instead of concrete, we use **abstract data types**
  - **Abstract data types (ADT)** are structures needed for efficient algorithms
  - For each ADT, every programming language has a corresponding concrete data type
  - ADT → classifying data structures according to how they are used / behaviors they provide
  - **ADT** does not specify how the data structure is implemented or represented in memory

Abstract Data Types

Abstract Data Type	Other Common Names	Commonly implemented as
List	**Sequence**	**Array, Linked List**
Queue		**Array, Linked List**
Double-ended Queue	Dequeue, Deque	Array, Doubly-linked List
Stack		**Array, Linked List**
Associative Array	**Dictionary, Hash Map, Map**	**Hash Table**
Set		Red-black Tree or Hash Table
Priority Queue	**Heap**	**Heap**

Dynamic Sets

- **Dynamic sets** of **values** („data”) – collections of items on which the following operations are expected to be commonly executed
  - Addition of elements („INSERT” operation)
  - Removal of elements („DELETE” operation)
  - Replacement of elements
    - Can be seen as removal + addition
- Simple ADTs for dynamic sets
  - Lists
  - Queues
  - Stacks

Lists: Arrays vs. Linked Lists

- ADT: **List** – a linear sequence of elements, ordered collection of values
  - No constraints on INSERT or REMOVE (can insert or fetch from anywhere in the list)

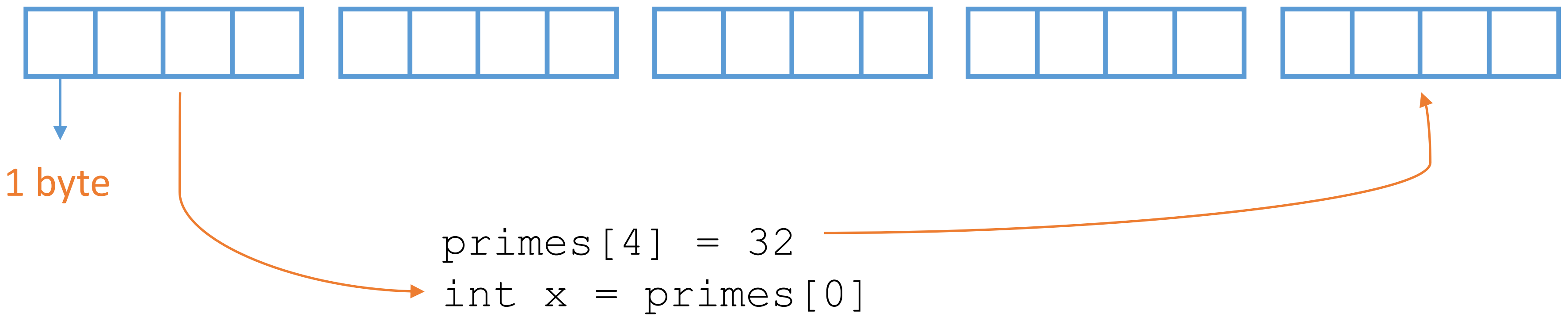
- Commonly implemented as **data structures**:
  - **Array** or
  - **Linked List**

- **Array** (as a concrete **data structure**, not ADT)
  - Among the oldest, most widely used data structures in programming
  - Values are of **homogeneous size** and stored in **contiguous memory**
  - To create an array, we need to **allocate contiguous memory space**
    - Q: How much of it?
  - **Fixed size** (*Dynamic array* allows resizing after creation)

Lists: Arrays vs. Linked Lists

- ADT: **List** – a linear sequence of elements, ordered collection of values
  - When we design algorithms, we typically think in terms of ADTs

- **Array** (as **data structure**, not ADT)
  - Writing values into and reading values from an array is **fast**
  - Example in *C* (as Arrays in Python or Java are implemented differently)
    - `int primes[5];` // allocate 5 x size of int (typically 4 bytes) of contiguous memory

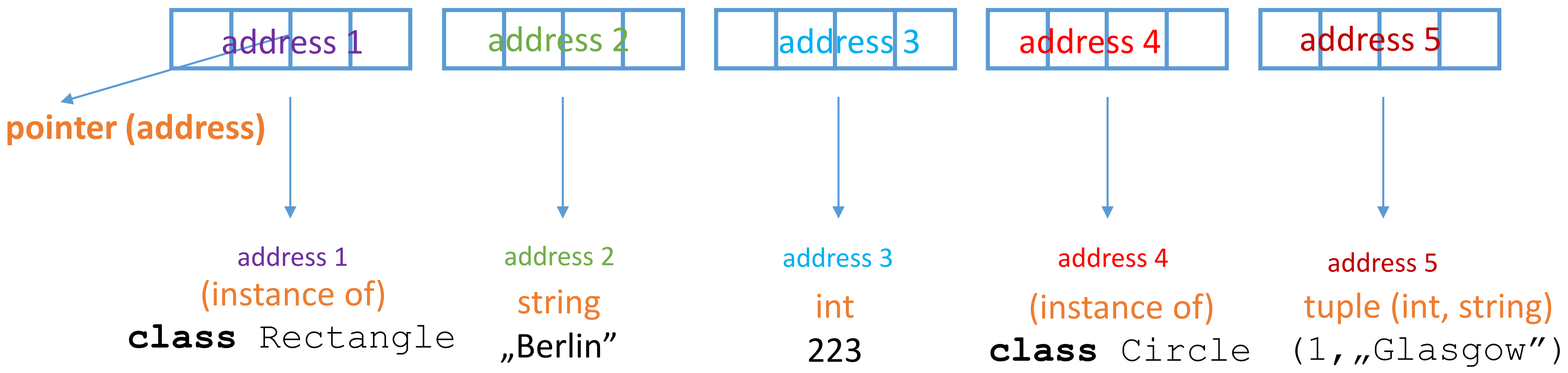


Lists: Arrays vs. Linked Lists

- ADT: **List** – a linear sequence of elements, ordered collection of values
- **Array** (as **data structure**, not ADT): problems
  - Standard array: elements cannot be of different size/type
    - Some languages remedy for this: for example, Python and Java
  - Arrays are of fixed size
    - Lists (as ADT) are, in most algorithms, expected to be of changeable size (elements added, removed, etc.)
    - Changing the size of array (as data structure) requires allocating additional contiguous memory (or releasing some) – what if none is available?

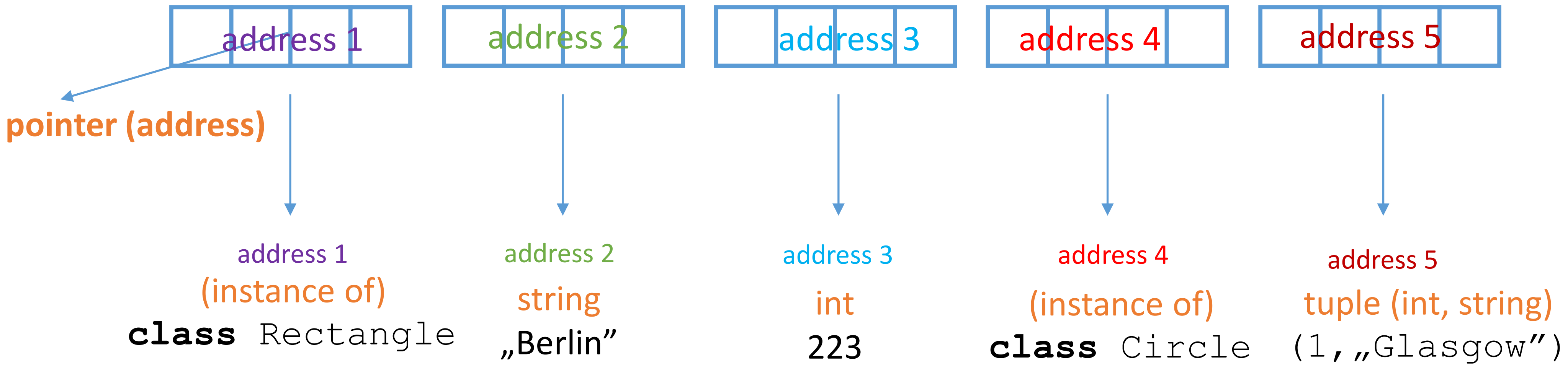
Python and Java arrays

- Java and Python arrays can have **elements of different types/sizes**
- Instead of storing values themselves into the array, they store **pointers to actual values** into arrays
  - Pointers are all of the same size (numbers representing memory addresses)



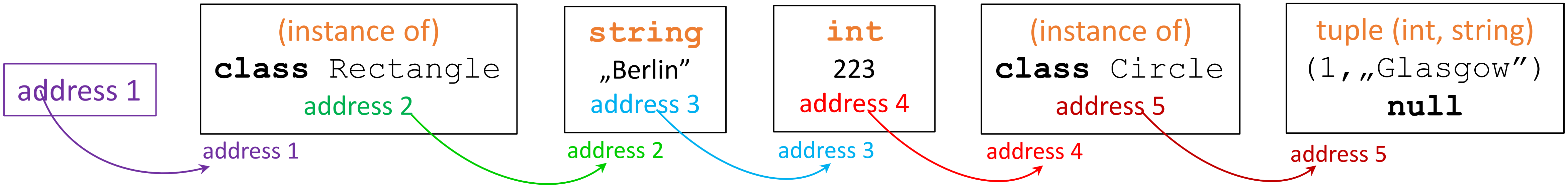
Python and Java arrays

- Java and Python arrays can have **elements of different types/sizes**
- **Pointers** are fast to read and write, but not the values themselves
  - Values are not stored in contiguous memory
  - Value access in non-contiguous memory slower
  - More flexible: can have arrays of arbitrary objects/values
    - Necessary for OO programming languages (remember **inheritance** and **polymorphism**)



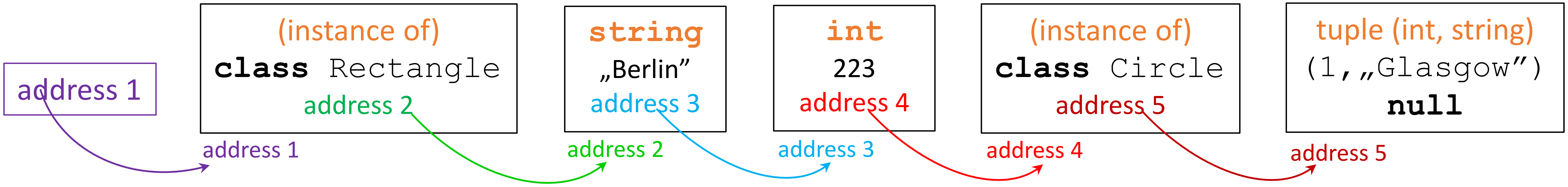
Lists: Arrays vs. Linked Lists

- ADT: **List** – a **linear** sequence of elements
  - When we design algorithms, we typically think in terms of ADTs
- **Linked List**
  - Consists of **nodes**: nodes contain both the data (values) and a **pointer** to the next node in the list
  - Nodes can contain values of different types
  - Dynamic data structure: „resizable” at runtime
    - Non-contiguous memory allocation possible, space for **new nodes** can be allocated dynamically (on „per-need” basis)



Lists: Arrays vs. Linked Lists

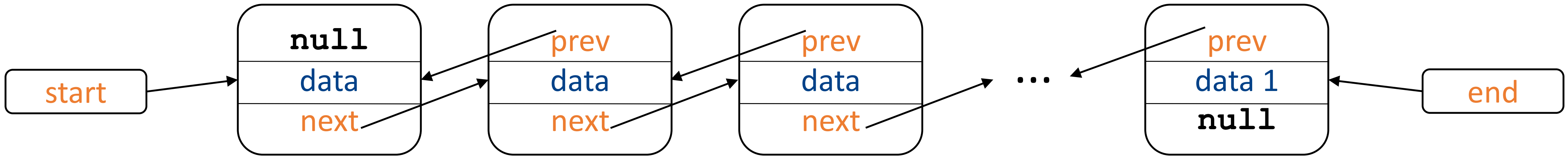
- ADT: **List** – a **linear** sequence of elements
  - When we design algorithms, we typically think in terms of ADTs
- **Linked List**
  - Disadvantage: access to nodes (read, write) is slower than with arrays
  - To access the value of the *n-th* element of the list, we have to pass through the preceding n-1 nodes
  - We have only the pointer to the beginning of the list as the access point



Lists: Doubly Linked List

- ADT: **(Bidirectional) List** – a **linear** sequence of elements
  - When we design algorithms, we typically think in terms of ADTs

- **Doubly Linked List**
  - Iterating through a regular linked list is possible only in one direction
  - Additional pointer in each node → possible to iterate backwards too
  - Each node has two pointers now
    - **Forward pointer** (first node called **head** of the list)
    - **Backward pointer** (last node called **tail** of the list)



- **Q**: what would be a **circular** list?

Stacks and Queues

- Data structures for handling dynamic sets for which the operation for removing an element (**DELETE** operation) is **prespecified**
    - I.e., cannot remove any element, there is a prespecified order of removal
- **Stack** („last-in, first-out”, **LIFO** policy)
    - The element removed is always the element last inserted
- **Queue** („first-in, first-out”, **FIFO** policy)
    - The element removed is always the one inserted the earliest (the one that’s been in the queue the longest)
- How to **efficiently** implement stacks and queues?

Stack

- Think of a physical stack, e.g., stack of plates in a restaurant
  - **INSERT** operation is often called PUSH („push to the stack”)
  - **DELETE** operation is often called POP („pop from the stack”)

- Implementing stack (as ADT)
  - With **Array** or with **Linked List** (as actual data structures)

- Stack as **array**
  - Fixed number of elements
  - Not well-suited for stacks that have an unknown maximal size
  - Index of the first element of array (so, commonly 0) is the **bottom** of the stack, index of the last element is the **top**
  - **If** the **top** of the stack is 0, the stack is empty

Stack

- Stack as **array**
  - We allocate an array of fixed size: *n* elements
  - Not suited for stacks that have an unknown maximal size

  - **If** the **top** of the stack is 0, the stack is empty
  - What happens if we try to POP from an empty stack?
    - This is stack underflow → you'll typically get a runtime error

  - **If** the **top** of the stack is n-1 (stack has n elements)
  - What happens if we try to PUSH one more element to the stack?
    - This is **stack overflow** → depending on the actual implementation of an array: maybe an error, maybe dynamic reallocation in the memory for a larger array

Stack operations (with **array**, pseudocode)

- We assume that stacks *S* consists of two data pieces
  - Array containing the elements: S.elements
  - An integer variable which indicates where the **top** is: S.top

```
create_stack(n)
  S.elements = array[n]
  S.top = 0
  return S
```

```
is_empty(S)
  if S.top == 0
    return True
  else
    return False
```

```
push(S, x)
  if S.top == len(S.elements) - 1
    error „overflow”
  else
    S.elements[S.top] = x
    S.top = S.top + 1
```

```
pop(S)
  if is_empty(S)
    error „underflow”
  else
    x = S.elements[S.top]
    S.top = S.top - 1
    return x
```

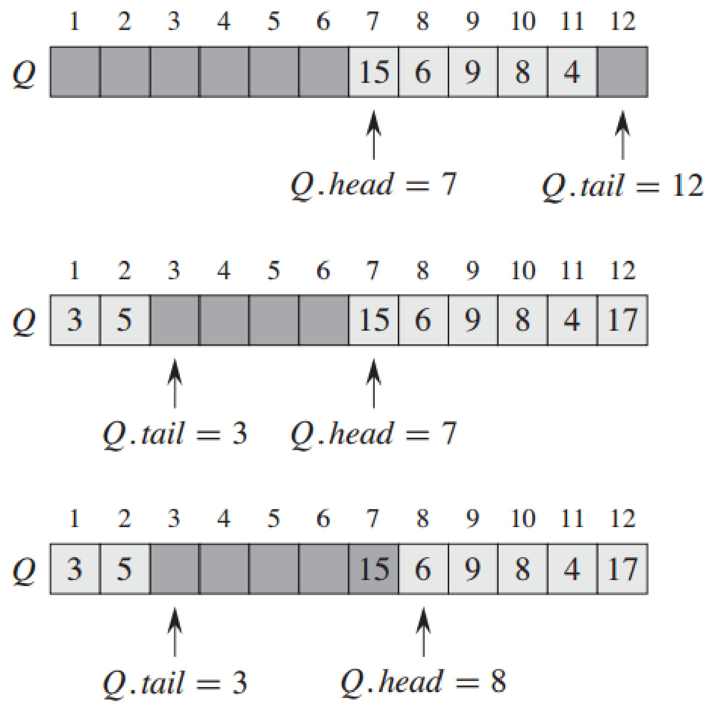
Queue

- Think of a physical queue, e.g., students waiting in the queue in mensa ☺
  - INSERT operation is often called ENQUEUE
  - DELETE operation is often called DEQUEUE
- Implementing queue (as ADT)
  - With **array** or with **linked list** (as actual data structures)
- Queue as **array**
  - Fixed number of elements
  - Not suited for queues that have an unknown maximal size
  - Index of the first element in the queue is the **head** of the queue, index of the first empty element in the array is the **tail**
  - Both head and tail can move; keep track of the number of elements in the queue

Queue

- Start with the queue of size 12 that contains 5 elements (e.g., in positions 7-11)
  - Example and image from Cormen et al., page 234

- Then execute
  1. `Enqueue(Q, 17),`
  2. `Enqueue(Q, 3),`
  3. `Enqueue(Q, 5),`
  4. `Dequeue(Q)`



Queue operations (with **array**, pseudocode)

- We assume that queue *Q* consists of three data pieces
  - Array containing the elements: *Q*.elements
  - Variables indicating where the head and tail are: *Q*.head, *Q*.tail
  - A Queue of size *n* requires an array with *n+1* elements. **Q**: Why?

```
create_queue(n)
  Q.elements = array[n+1]
  Q.head = 0
  Q.tail = 0
  return Q
```

```
is_empty(Q)
  if Q.head == Q.tail
    return True
  else
    return False
```

```
is_full(Q)
  if Q.tail+1 == Q.head
    return True
  else
    return False
```

Stack operations (with **array**, pseudocode)

- We assume that queue *Q* consists of three data pieces
  - **Array** containing the elements: *Q*.elements
  - Variables indicating where the head and tail are: *Q*.head, *Q*.tail
  - A Queue of size *n* requires an array with *n+1* elements. **Q**: Why?

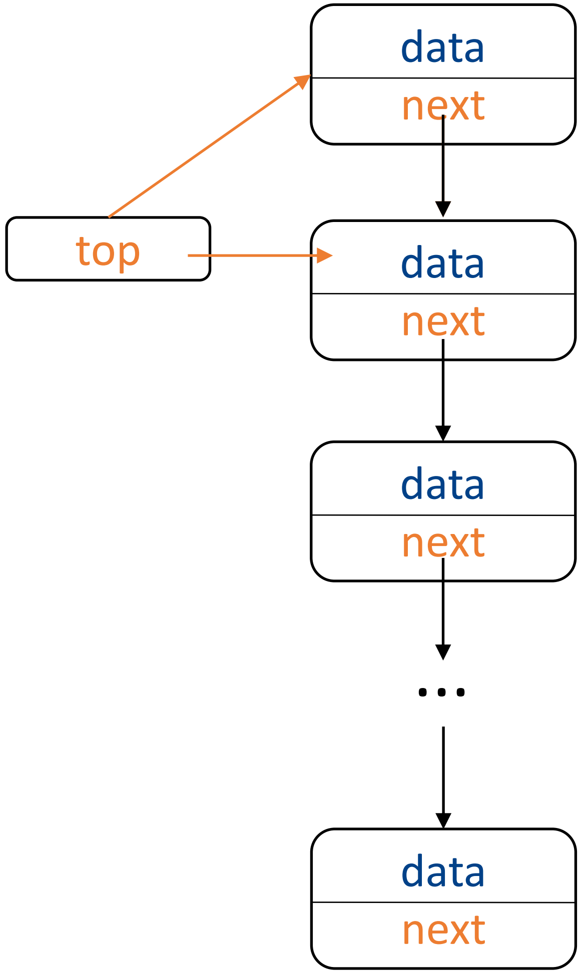
```
enqueue(Q, x)
  if is_full(Q)
    error „overflow”
  else
    Q.elements[Q.tail] = x
    Q.tail = (Q.tail + 1) % len(Q.elements)

dequeue(Q)
  if is_empty(Q)
    error „underflow”
  else
    x = Q.elements[Q.head]
    Q.head = (Q.head + 1) % len(Q.elements)
    return x
```

Stack & Queue with Linked List

- Same as for **List**, the alternative to an **Array** for implementing Stack and Queue is the **Linked List**
- Pros & cons between **Array** and **Linked List** the same as for List (ADT)
  - Working with a fixed **Array** is faster, values stored in contiguous memory
  - Dynamic (arbitrary size) **queues** & **stacks** require dynamic memory allocation
    - Either **Dynamic Array** or **Linked List**
  - With Linked List
    - Dynamic size trivially supported
    - We add **pointers**
    - Values can be of different types
    - Because of **pointer following**, **slower** (more read operations)

Stack with Linked List

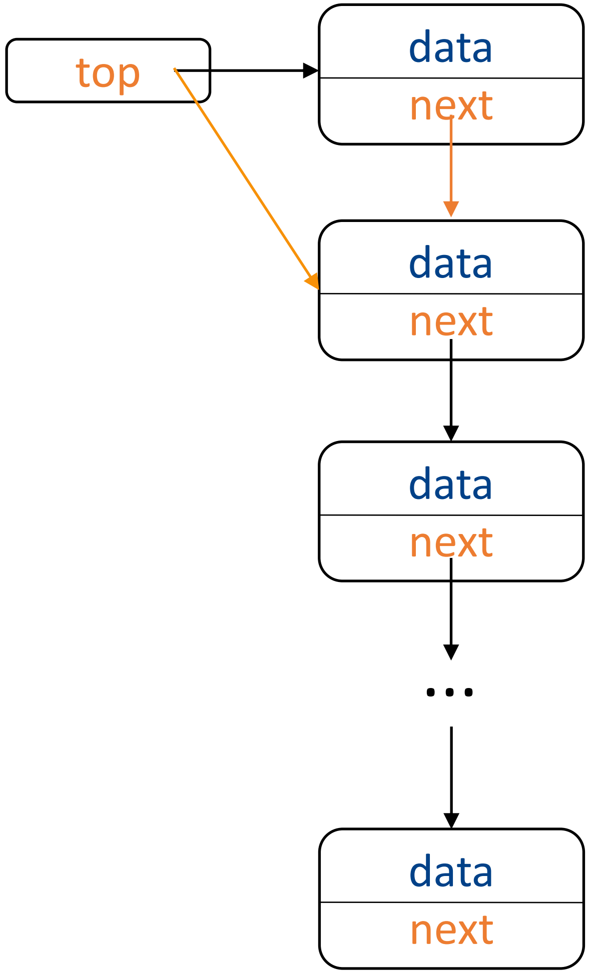


- We assume that each node has the property address specifying the memory address of its data
- We also assume that each node has a **pointer** next in which we store the address of some other data (node)

```
create_stack()
   S.top = null
   return S
```

```
push(S, x)
   x.next = S.top
   S.top = x.address
```

Stack with Linked List



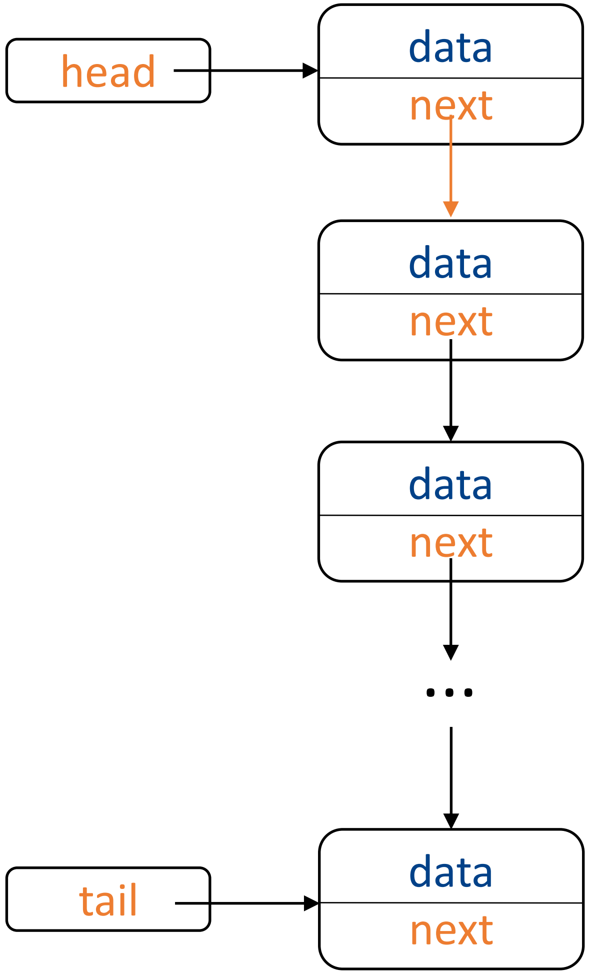
- We assume that each node has the property data getting the actual data (value) stored in the node
- We also assume that each node has a **pointer** next in which we store the address of some other data (node)

```
is_empty(S)
  return (S.top == null)


pop(S)
  if is_empty(S)
    error „underflow”
  else
    x = S.top.data
    S.top = S.top.next
    return x
```

Queue with Linked List

- Similar to Stack with Linked List, only we need two **pointers**
  - **Head** pointer (for dequeing)
  - **Tail** pointer (for enqueing)




```
create_queue()
   Q.head = null
   Q.tail = null
   return Q
```

```
is_empty()
   return Q.head == null
```

```
dequeue(Q)
   if is_empty(Q)
      error „underflow”
   else
      x = Q.head.data
      Q.head = Q.head.next
      if Q.head == null
         Q.tail = null
      return x
```

```
enqueue(Q, x)
   if Q.tail != null
      Q.tail.next = x.address
   Q.tail = x.address
   if Q.head == null
      Q.head = x.address
```

Questions?


Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Frågor?
Questions?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟