Data structures in memory

Abstract data types
and
data structures

- An abstract data type is a data container
  - Examples:
    - in Python: `list`, `dict`, `set`, …
    - in C++: `std::vector`, `std::unordered_map`, …
  - Specifies which operations are (natively) supported
  - Does not specify how data is stored
  - Does not specify how the operations are implemented

- A data structure is an implementation of an abstract data type
  - Specifies how data is layed out in memory
  - Specifies which algorithms are used for operations
    - We can compute the computational complexity of those algorithms

Lists

- Lists are one of the simplest abstract data type
- Just a collection of ordered elements
- They support
  - storing multiple elements together
  - and optionally
    - appending an element (at the end of the list)
    - discarding the last element (at the end of the list)
    - inserting an element (in any position) in the list
    - deleting an element (in any position) in the list
    - accessing or modifying all elements in order
    - accessing or modifying an element at an arbitrary index ("random access")
    - …

Arrays

Static arrays

- Static arrays implement lists of a fixed size n
- Elements are stored contiguously, one after another, in memory
- They implement
  - accessing or modifying an element at an arbitrary index
    - `element_address = array_address + index * element_size`
    - complexity $O(1)$
  - accessing or modifying all elements in order (direct consequence of random access)
    - complexity $O(n)$

Dynamic arrays

- Dynamic arrays implement lists of a variable size n
- Elements are stored contiguously, one after another, in memory
- They implement static array operations, plus
  - changing the size n of the list

    complexity $O(n)$ in theory
  - as a consequence, we can
    - append an element (at the end of the list) in $O(n)$
    - discard the last element (at the end of the list) in $O(n)$
    - insert an element (in any position) in the list in $O(n)$
    - delete an element (in any position) in the list in $O(n)$
    - …

Size increase

- An array occupies the bytes in memory:
  - from `array_address`
  - to `array_address + n * element_size - 1`

- Increasing n has $O(n)$ complexity, because the memory at

  `array_address + n * element_size`

  may be occupied by other data

- In that case, the dynamic array must be relocated elsewhere in memory (changing `array_address`)

- All `n * element_size` bytes must be copied to the new location, hence $O(n)$ complexity

Size decrease

- Conversely, if the memory before and/or after an array is free,
  - we may want to move the array
  - in order to create a larger block of free memory
- Not doing this may cause “memory fragmentation”

In theory:

operation	complexity
access/modify element at arbitrary index	$O(1)$
increase n	$O(n)$
decrease n	$O(n)$
append an element	$O(n)$
discard last element	$O(n)$
insert an element	$O(n)$
delete an element	$O(n)$

In practice: Almost all implementations ignore fragmentation due to shrinking

(no move when decreasing $n > 0$)

operation	complexity
access/modify element at arbitrary index	$O(1)$
increase n	$O(n)$
decrease n	$O(1)$
append an element	$O(n)$
discard last element	$O(1)$
insert an element	$O(n)$
delete an element	$O(n)$

Over-allocation

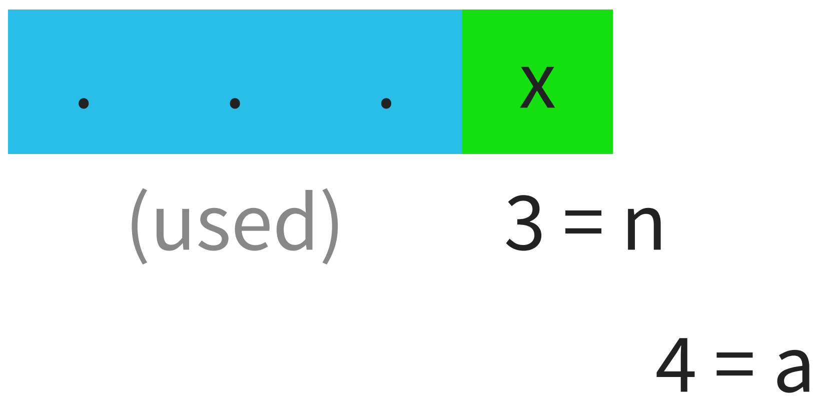
- We have two distinct quantities:
  - the user-visible size n
  - the allocated size a

- If the user requests a size increase $n' > n$
  - as long as $n' \le a,$ nothing needs to happen

- a is never incremented (no $a' = a + 1$)

- instead, we increase a exponentially $(a' = 2a)$

Exponential allocation (n = 3)



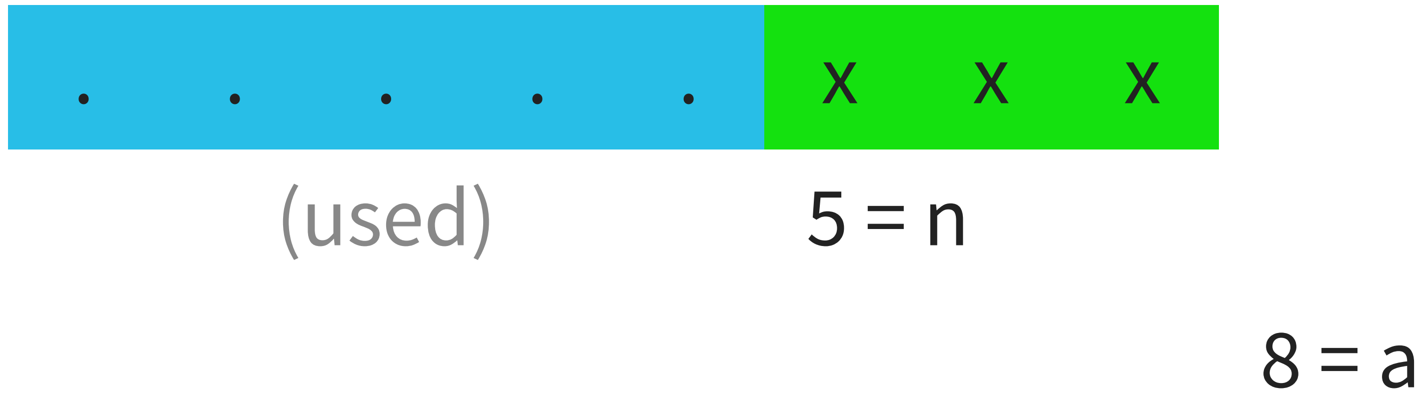
Exponential allocation (n = 4)

(used)

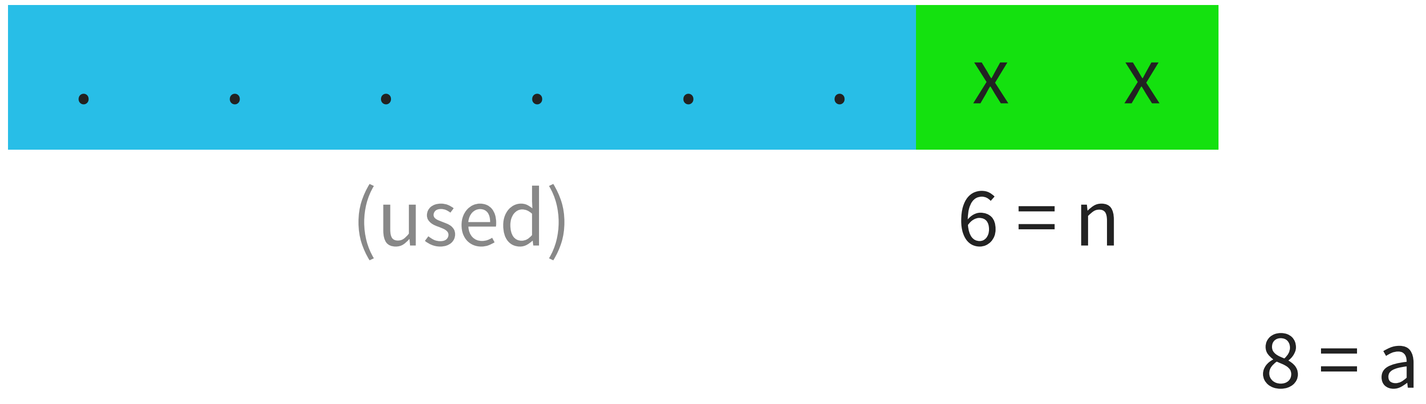
4 = n

4 = a

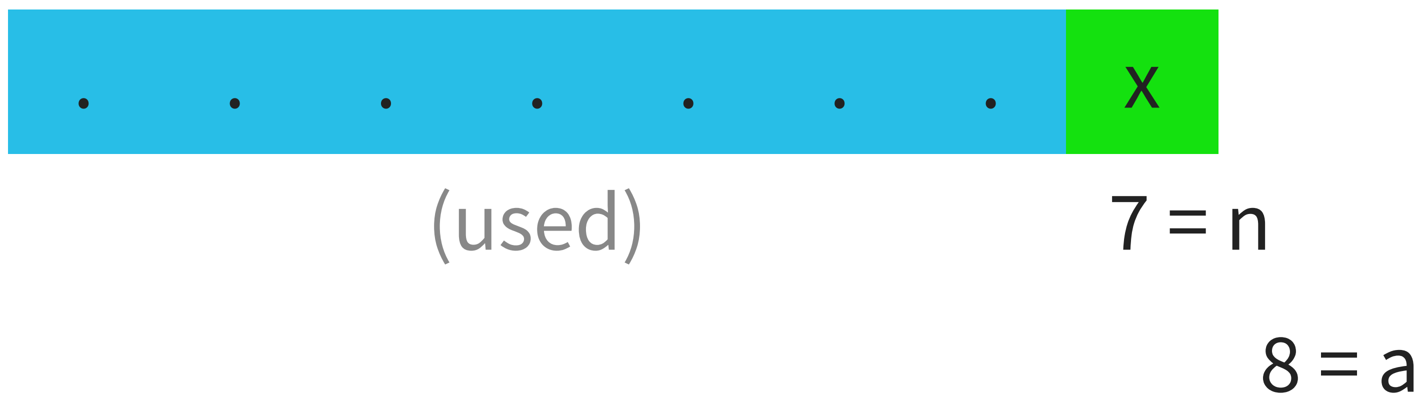
Exponential allocation (n = 5)



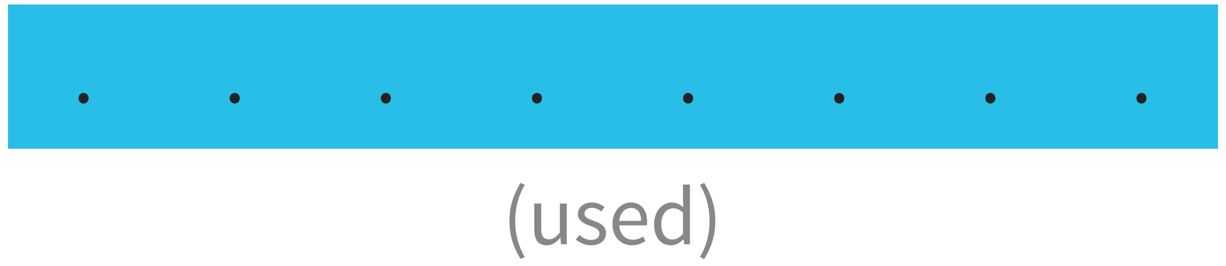
Exponential allocation (n = 6)



Exponential allocation (n = 7)



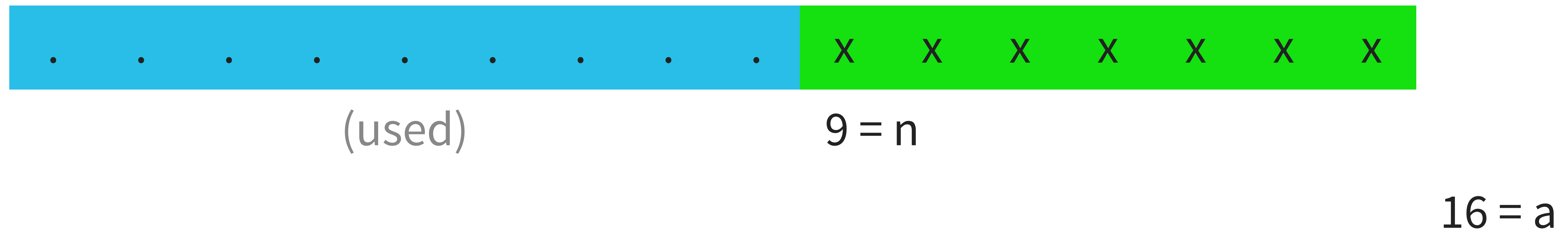
Exponential allocation (n = 8)



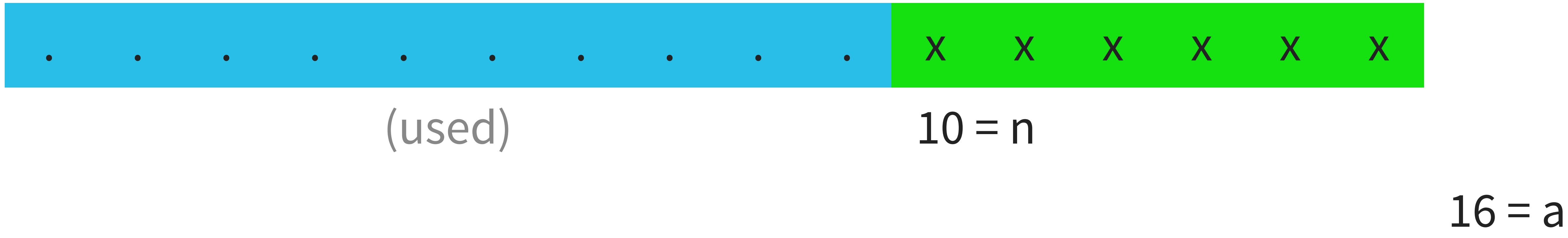
8 = n

8 = a

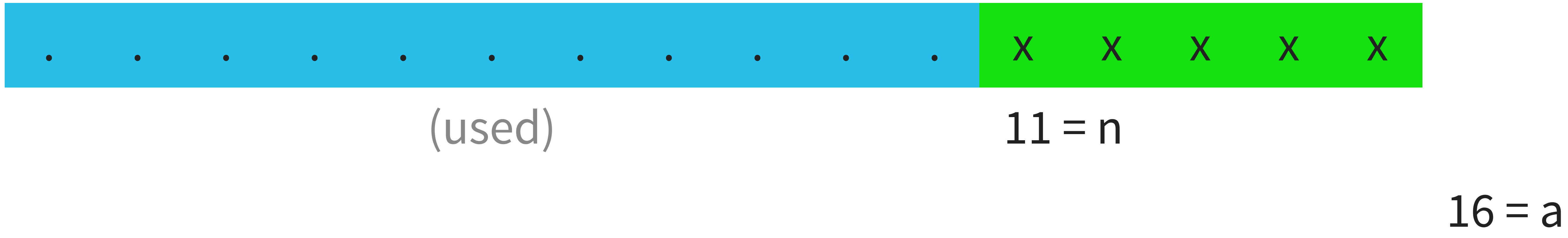
Exponential allocation (n = 9)



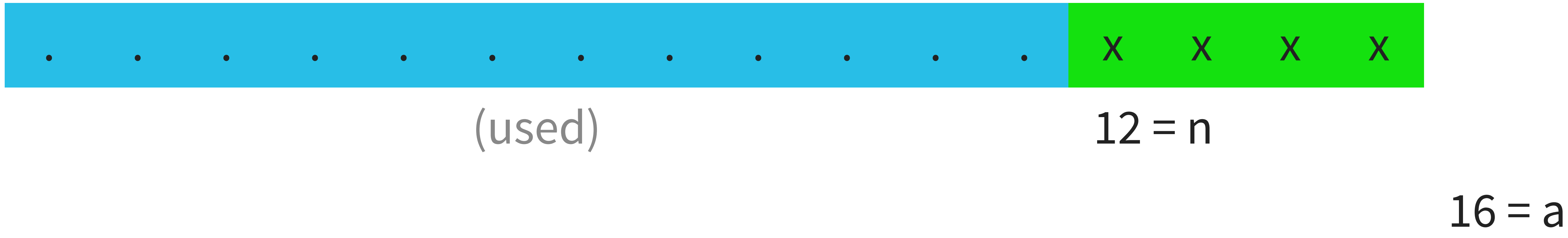
Exponential allocation (n = 10)



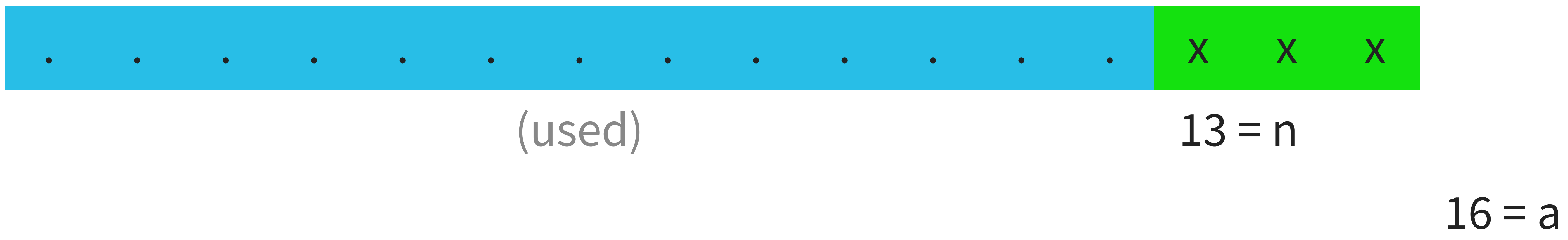
Exponential allocation (n = 11)



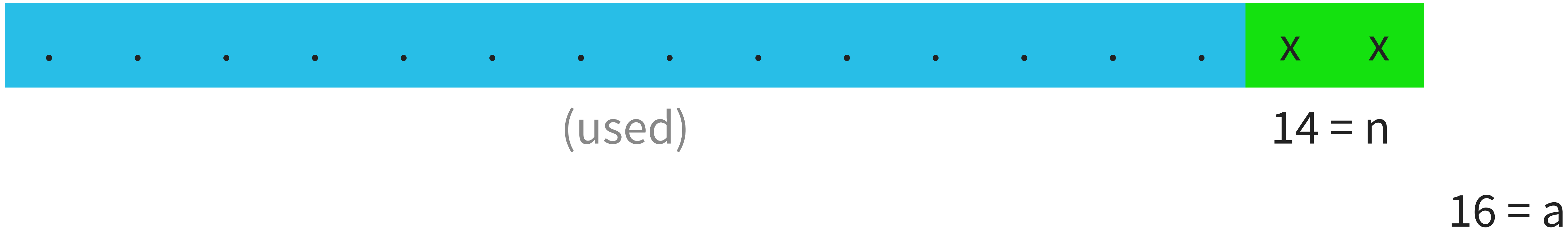
Exponential allocation (n = 12)



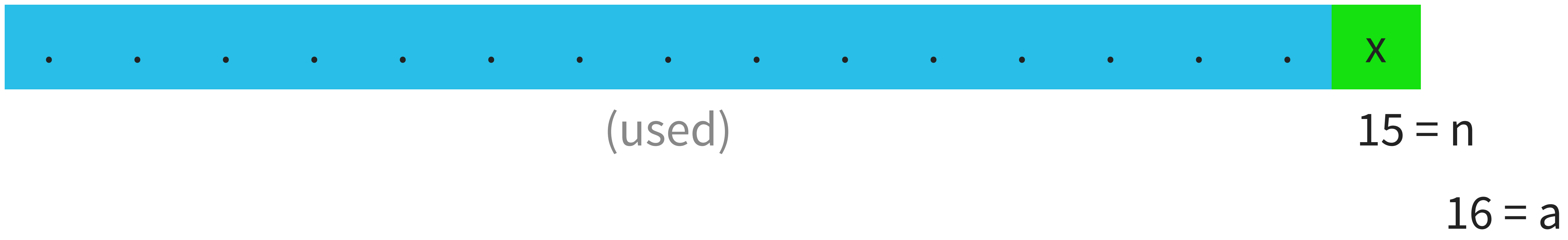
Exponential allocation (n = 13)



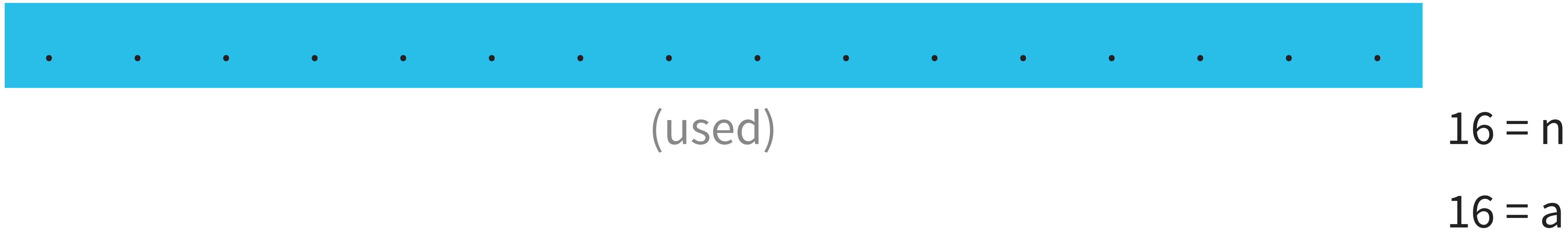
Exponential allocation (n = 14)



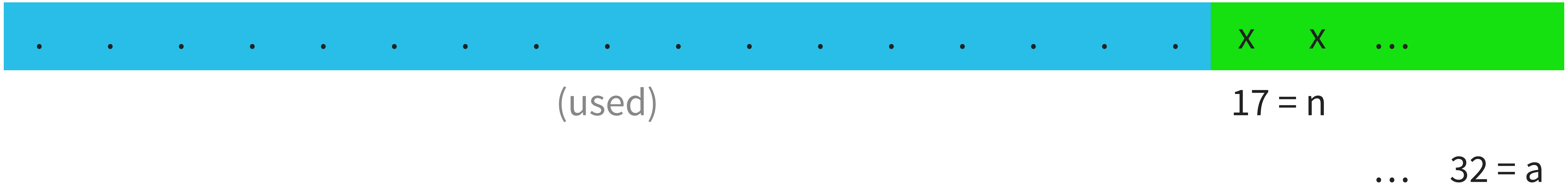
Exponential allocation (n = 15)



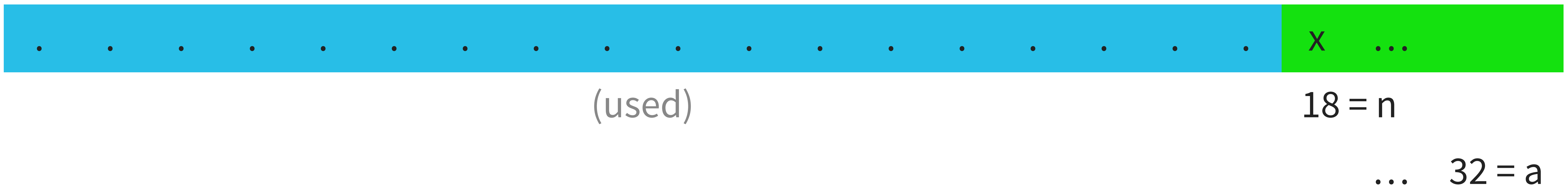
Exponential allocation (n = 16)



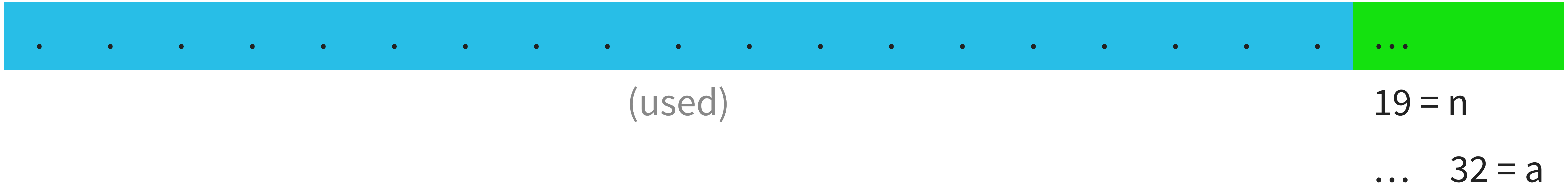
Exponential allocation (n = 17)



Exponential allocation (n = 18)



Exponential allocation (n = 19…)



```c
struct dynamic_array {
    void *address;
    size_t n;
    size_t a;
};

int grow(struct dynamic_array *d, size_t new_n)
{
    if (new_n <= d->a) {
        d->n = new_n;
        return SUCCESS;
    }

    size_t new_a = d->a;

    while (n > new_a)
        new_a = new_a * 2;

    void *new_addr = malloc(new_a);

    if (new_addr == NULL)
        return ERROR;

    memcpy(new_addr, d->address, d->n);     // O(n)
    free(d->address);

    d->address = new_addr;
    d->n = new_n;
    d->a = new_a;

    return SUCCESS;
}
```

Drawback of exponential allocation

We waste some memory.

However, we always have $a \leq 2n$ (specifically, $a = 2^{\lceil \log_2(n) \rceil}$)

Complexity of exponential allocation (loose analysis)

- start with an empty array
- increment its size n times
- $\Rightarrow$ we perform (at most) $k := \lceil \log_2(n) \rceil$ moves, of sizes $1, 2, 4, 8, 16, \ldots, 2^{k-1}$.
- $\Rightarrow$ total cost:

$$\underbrace{\begin{matrix} 1 & + & 2 & + & 4 & + & 8 & + & \ldots & + & 2^{k-1} \\ \leq n & & \leq n & & \leq n & & \leq n & & \ldots & & \leq n \end{matrix}}_{k \text{ terms}}$$

- $\leq kn$ total (for n size increments)
- $\leq k$ amortized (for each size increment)
- $O(\log_2(n))$ amortized

Complexity of exponential allocation (better analysis)

- start with an empty array
- increment its size n times
- $\Rightarrow$ we perform (at most) $k := \lceil \log_2(n) \rceil$ moves, of sizes $1, 2, 4, 8, 16, \ldots, 2^{k-1}$.
- $\Rightarrow$ total cost:

$$1 \quad + \quad 2 \quad + \quad 4 \quad + \quad 8 \quad + \quad \ldots \quad + \quad 2^{k-1}$$

- $= \; 2^k - 1$ (power series)
- $= \; 2^{\lceil \log_2(n) \rceil} - 1$
- $\leq \; 2n$
- $O(n)$ total (for n size increments)
- $O(1)$ amortized (for each size increment)

operation	complexity
access/modify element at arbitrary index	$O(1)$
increase n	$O(1)$ amortized
decrease n	$O(1)$
append an element	$O(1)$ amortized
discard last element	$O(1)$
insert an element	$O(n)$
delete an element	$O(n)$

Virtual memory

In terms of asymptotic complexity, the cost of changing n comes from

```
memcpy(new_addr, d->address, d->n);     // O(n)
```

- But memory is virtualized,
- we do not need to physically move bytes around.

- Instead we can use the page table to
  - remap the physical memory associated to a virtual address (`d->address`)
  - to a different virtual address (`new_addr`).

Remapping virtual memory using the page table

- Pro: Memory move becomes essentially $O(1)$ in practice
- Con: Need to call the OS kernel to change page table
  - context switch (swap page table, pollute caches)
  - large fixed cost

- As a consequence, this is done only when n grows very large (multiple megabytes of data).
- $a' = a + K$ for some large K (avoids waste of exponential increase)

For very large n (multiple megabytes):

operation	complexity
access/modify element at arbitrary index	$O(1)$
increase n	$O(1)$ (roughly)
decrease n	$O(1)$
append an element	$O(1)$ (roughly)
discard last element	$O(1)$
insert an element	$O(n)$
delete an element	$O(n)$

Linked lists

- Linked lists implement lists of a variable size n
- They implement
  - inserting, deleting, modifying an element (in any position): $O(1)$
  - accessing or modifying all elements in order: $O(n)$
- They do not have special support for accessing or modifying an element at an arbitrary index ("random access")
- but it can be implemented using above ("accessing all elements"), with complexity $O(n)$

Doubly-linked lists

```
struct element {
    struct payload data;

    struct element *prev;
    struct element *next;
};

int insert_after(struct element *e, struct payload data)
{
    struct element *x = malloc(sizeof(struct element));

    if (x == NULL)
        return ERROR;

    struct element *f = e->next;

    x->data = data;
    x->prev = e;
    x->next = f;

    e->next = x;
    f->prev = x;

    return SUCCESS;
}
```

operation	dynamic array	doubly-linked list
access/modify element at arbitrary index	$O(1)$	$O(n)$
increase n	$O(1)$	$O(1)$
decrease n	$O(1)$	$O(1)$
append an element	$O(1)$	$O(1)$
discard last element	$O(1)$	$O(1)$
insert an element	$O(n)$	$O(1)$
delete an element	$O(n)$	$O(1)$

Memory management considerations

Memory allocation is slow

```
struct element *x = malloc(sizeof(struct element));
```

compared to dynamic arrays' fast case

```
if (new_n <= d->a) {
    d->n = new_n;
    return SUCCESS;
}
```

operation	dynamic array	doubly-linked list
access/modify element at arbitrary index	$O(1)$	$O(n)$
increase n	$O(1)$	$O(1)$
decrease n	$O(1)$	$O(1)$
append an element	$O(1)$	$O(1)$
discard last element	$O(1)$	$O(1)$
insert an element	$O(n)$	$O(1)$
delete an element	$O(n)$	$O(1)$

Memory caches considerations

List traversal

```
for (int i = 0; i < n; i++) {
    struct payload data = dynamic_array[i];
    ...
}
```

```
struct element *e = first_element;

while (1) {
    struct payload data = e->data;
    ...
    e = e->next;
    if (e == first_element)
        break;
}
```

- assuming deep pipelines and good branch prediction,
- the processor can start fetching `dynamic_array[i + 1]` while waiting for `dynamic_array[i]`
- but it cannot start fetching `e->next->data` while waiting for `e->data` / `e->next` (data dependency)

- Linked list have fewer applications than one could expect
- However, when they are appropriate, they can be extremely useful

More options

- Indirection (dynamic array of pointers)
- In-memory tree data structures

```
struct nary_node {
  struct payload data;

  struct nary_node *children[MAX_CHILDREN];
}
```

```
struct dll_node {
  struct payload data;

  struct dll_node *prev_sibling;
  struct dll_node *next_sibling;
  struct dll_node *first_child;
}
```

- …