1 Big-O notation

Consider 2 functions $f, g : \mathbb{N} \to \mathbb{R}^{>0}$
The following are definitions of existing notations:

Def 1.1.1. $f = \Theta(g)$ $\quad \exists N > 0,\ C_1 > 0, C_2 > 0\colon \forall n \geqslant N, C_1 \cdot g(n) \leqslant f(n) \leqslant C_2 \cdot g(n)$

Def 1.1.2. $f = \mathcal{O}(g)$ $\quad \exists N > 0,\ C > 0\colon \forall n \geqslant N, f(n) \leqslant C \cdot g(n)$

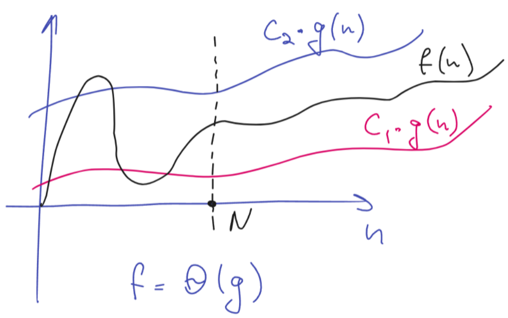
Def 1.1.3. $f = \Omega(g)$ $\quad \exists N > 0,\ C > 0\colon \forall n \geqslant N, f(n) \geqslant C \cdot g(n)$

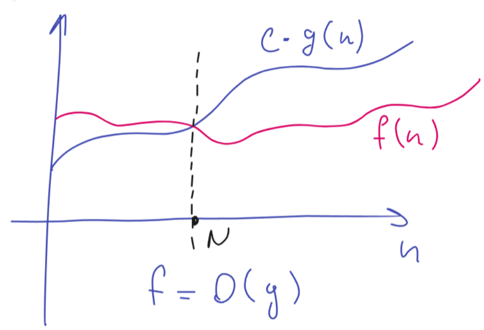
Def 1.1.4. $f = o(g)$ $\quad \forall C > 0\ \exists N > 0\colon \forall n \geqslant N, f(n) \leqslant C \cdot g(n)$

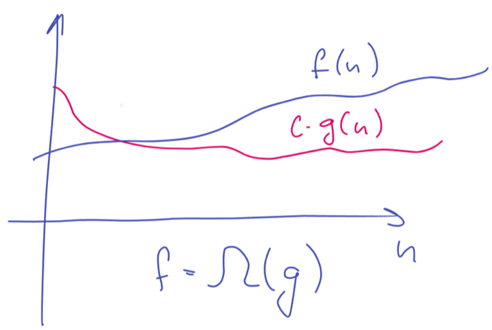
Def 1.1.5. $f = \omega(g)$ $\quad \forall C > 0\ \exists N > 0\colon \forall n \geqslant N, f(n) \geqslant C \cdot g(n)$

Mnemonics:
1. Θ : "equal up to a constant", "asymptotically equal".
2. O : "$\leq$ up to a constant", "asymptotically $\leq$".
3. o : "asymptotically $<$", "for any constant $\leq$", "whatever constant you take there will be a moment where $f(n)$ becomes $\leq C \cdot g(n)$."

Here are graphical representations for some Big-O notations:







You may interpret the Big-O notations in the following manner:

Θ	O	Ω	o	ω
=	≤	≥	<	>

Some remarks:
1. $f = \Theta(g) \Leftrightarrow g = \Theta(f)$.
2. $f = O(g)$ and $g = O(f) \Leftrightarrow f = \Theta(g)$.
3. $f = \Omega(g) \Leftrightarrow g = O(f)$
4. $f = \omega(g) \Leftrightarrow g = o(f)$

5. $\forall C > 0 : C \cdot f = \Theta(f)$

Common asymptotics:
1. Linear: $O(n)$
2. Quadratic: $O(n^2)$
3. Polynomial: $\exists k > 0 :\ O(n^k)$
4. Polylogarithmic: $\exists k > 0 :\ O(\log^k n)$
(logarithm's base does not matter since logarithms with difference bases differ only in a constant).
5. Exponential: $\exists c > 0 :\ O(2^{c \cdot n})$

Important: $\forall a, b > 0,\ c > 1 :\ \exists N :\ \forall n > N :\ \log^a n < n^b < z^c$

1.1 Big-O for Polynomial of degree k

Lemma 1
$n^k = o(n^{k+1})$

Proof. Consider any $C \in \mathbb{R}^{>0}$ and take such N, so that $N \geq C$ holds.

Now $\forall n \geq N : n^{k+1} = n^k \cdot n \geq n^k \cdot N \geq n^k \cdot C \implies n^k \leq \frac{1}{C} \cdot n^{k+1}$ by definition of o we have achieved $n^k = (n^{k+1})$ □

Lemma 2
$g = o(f) \to f \pm g = \Theta(f)$

Proof. $g = o(f) : C := \frac{1}{2}\ \exists N : \forall n \geq N : g(n) \leq \frac{1}{2} \cdot f(n) \to \frac{1}{2} f(n) \leq f(n) \pm g(n) \leq \frac{3}{2} f(n)$

□

Statement: Big-O for $P(x)$
Given a polynomial $P(x) = a_0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \ldots + a_k \cdot x^k$, then $P = \Theta(x^{\mathbf{deg} P})$

Proof. We have shown in the **lemma 1** above that $n^k = o(n^{k+1})$, thus $\forall i = 1..l-1 :\ a_i \cdot x^i = o(x^k)$.

Thus, $P(x) = a_0 + a_1 \cdot x_1 + \ldots + a_{k-1} \cdot x^{k-1} + a_k \cdot x^k = o(x^k) + x^k$.

Now use the **lemma 2** we achieve: $P(x) = o(x^k) + x^k = \Theta(x^k)$ □

2 Basic data structures, beginning

We are going to discuss some data structures in terms of some operations and their **time complexity**.

Def: **time complexity** is measurement that describes the amount of operations it takes to execute an algorithm.

For instance, an algorithm that traverses all elements in an array has the time complexity of $O(n)$ where n is a number of elements (i.e. an input parameter).

The operations and their time complexities that we are interesting in:
1. $get(i)$ - get element by index i.
2. $set(i, x)$ - assign to the element with index i a value x.
3. $find(x)$ - find index of an element x (or return -1 if not found).
4. $add\_begin(x)$, $add\_end(x)$ - add element x to the beginning/end.
5. $remove\_begin()$, $remove\_end()$ - remove element from the beginning/end.

2.1 Plain array

Definition of a fixed-size array in C++:

```cpp
int main() {
    const int N = 10;
    int arr[N] = {0}; // See: list initialization

    for (int i = 0; i < N; ++i) {
        arr[i] = i + 1;
    }

    // See: range-based for-loop
    for (int val : arr) {
        std::cout << val << "\n";
    }

    return 0;
}
```

In the above example the array *arr* is allocated on stack, thus its size N must be a constant.
In order to have an array of an arbitrary size you should allocate it on heap:

```cpp
int n; cin >> n;
int* arr = new int[n];
delete[] arr;
```

Notice the *delete*[] operator: it deallocates the memory occupied for the array.

Arrays in C++ have fixed size, thus operations such as **add_begin/remove_begin** will require to allocate a new array and copy all the elements adding/removing the 1st one accordingly, which is $O(n)$.
The **remove_end** could be done easily: just start to assume that the size of the array now is $N-1$.
find(x) will require to traverse all the element of the array and compare them to x, thus $O(n)$.

Here is the time complexities of the operations:

$get(i)$	$set(i,x)$	$find(x)$	$add\_begin(x)$ $add\_end(x)$	$remove\_begin()$ $remove\_end()$
$O(1)$	$O(1)$	$O(n)$	$O(n)$, $O(n)$	$O(n)$, $O(1)$

2.2 Never use plain arrays

Why? Because there is a much better data structure **std::vector** which is a dynamically re-sizable array. It has greater capabilities over plain arrays and it is greatly optimized.
Most likely you will never encounter a problem where you would benefit using plain arrays rather than **std::vector** in competitive programming.

2.3 Doubly Linked List

Let's start with the implementation:

```cpp
struct Node {
    Node* prev;
    Node* next;
    int x;
};

struct List {
```

```
    Node* head;
    Node* tail;
};

Node* find(int x, List l) {
    for (Node* node = l.head->next; node != l.tail; node = node->next) {
        if (node->x == x) {
            return node;
        }
    }
    return nullptr;
}

Node* remove(Node* node) {
    node->next->prev = node->prev;
    node->prev->next = node->next;
}
```

Here **head** and **tail** serve as illusory nodes that are not considered as nodes with values, but used as starting and terminating nodes:

In order to find an element x we traverse from **head** up to **tail** comparing current node's value.

In order to remove a node we update links of its **prev** and **next** nodes to point to each other, thus removing current node.

Our operations could be implementing either using those defined above or in a similar manner.

$get(i)$	$set(i,x)$	$find(x)$	$add\_begin(x)$ $add\_end(x)$	$remove\_begin()$ $remove\_end()$
$O(i)$	$O(i)$	$O(n)$	$O(1)$, $O(1)$	$O(1)$, $O(1)$

2.4 Singly Linked List

Idea is the same but we keep only references to the next node, i.e. having no information about who is pointing to us.

```
struct Node {
    Node* next;
    int x;
};

Node* head = new Node();

void add_begin(Node* &head, int x) {
    // head is reference to a pointer
    Node* node = new Node();
    node->next = head;
    node->x = x;
    head = node;
```

```
}
```

The difference now is that we cannot add/remove elements from the tail for $O(1)$ because we have to **tail** pointer. In order to add/remove to the end we need to traverse all the way to the end and apply modification.

Why would we need it if we have a doubly linked list? Singly linked list stores almost twice less memory and its operations have time complexities with less constants.

$get(i)$	$set(i,x)$	$find(x)$	$add\_begin(x)$ $add\_end(x)$	$remove\_begin()$ $remove\_end()$
$O(i)$	$O(i)$	$O(n)$	$O(1)$, $O(n)$	$O(1)$, $O(n)$

2.5 std::vector (expanding array)

A regular array is not convenient because its size is fixed in advance and limited.

The idea of improvement: we allocate *size* of memory cells in advance, when the real size of the array **n** becomes larger than **size**, we double the size and reallocate the memory.

```
int size = 1;
int n = 0;
int* arr = new int[size];

void push_back(int x) {
    if (n == size) {
        int* newArr = new int[2 * size];
        copy(arr, arr + n, newArr);

        arr = newArr;
        size *= 2;
    }
    arr[n++] = x;
}
```

$get(i)$	$set(i,x)$	$find(x)$	$add\_begin(x)$ $push\_back(x)$	$remove\_begin()$ $pop\_back()$
$O(1)$	$O(1)$	$O(n)$	$O(n)$, $\Theta(1)$	$O(n)$, $O(1)$

Notice that time complexity of $push\_back(x)$ is $\Theta(1)$, i.e. **average time**.

Statement: Time complexity of push_back

push_back(x) works in $\Theta(1)$.

Proof. Notice that the operation takes $O(n)$ when a new buffer is allocated. **How often does it happen?**

Before the next increase of size $n \to 2n$ there will be n push_backs that took $O(1)$ of time, and **exactly** 1 push_back that took $O(n)$, thus the average time is:

$$T = \frac{O(1) \cdot n + O(n) \cdot 1}{n+1} = \frac{O(n)}{n+1} = \Theta(1)$$

□

Now how to use the **std::vector**:

```
int main() {
    int n; cin >> n;
    int x = 0;
```

```
        // create vector of size n filled with x
        std::vector<int> vec(n, x);

        for (int i = 0; i < n; ++i) {
            vec[i] = vec[i] * 2;
            std::cout << vec[i] << "\n";
        }

        vec.push_back(10);
        std::cout << vec.back() << "\n";
        vec.pop_back(1);
    }
```

3 Practice

3.1 Prefix sums: query for $O(1)$

Task Prefix sums

You are given an array **a** of size n, and q queries $sum(l,r)$, i.e. the sum of elements from index l (inclusive) up to index r (exclusive). Each query must be answered in $O(1)$ of time.

Solution

The trivial approach is to sum all the elements in the provided range $[l,\ r)$ on each request which is $O(r-l) = O(n)$ time.

Let's define the following function:
$sum(i) = a_0 + a_1 + ... + a_{i-1}$ - sum of **first i elements** of the array.
Then the query of $sum(l,r)$ is:

$$sum(l,r) \ = \ sum(r) \ - \ sum(l) =$$

$$= (a_0 + ... + a_l + ... + a_{r-1}) - (a_0 + ... + a_{l-1}) = a_l + a_{l+1} + ... + a_{r-1}$$

Now, we only need to pre-calculate the function $sum(i)$ to give the answer in $O(1)$:

1. **Base case:** $sum(0) = 0$ - i.e. sum of first zero elements is zero.
2. **Transition:** $\forall i = 1..n: \ sum(i) \ = \ sum(i-1) \ + a_{i-1}$ - sum of first i elements is the sum of $(i-1)$ elements and current i-th element placed under the index $(i-1)$.

Task Find Duplicate

You are given an array **arr** of size N with numbers $0 \le a_i \le 10^6$. Return any element that occurs in the array **at least twice**, otherwise return -1. You are allowed to use only data structures that are already learnt so far.

Required space/time: $O(n)$

Notice: look at the constraints of a_i.

Solution

Notice that $0 \le a_i \le 10^6$, then let's create an array $occurrences$ with the following semantics:
$occurrences[a_i]$ - number of times an element a_i occurred in **arr**.

Then the answer is such a_i whose $occurrences[a_i] \ge 2$.

```
int findDuplicate(const vector<int>& arr) {
    int M = 1e6 + 1;
    std::vector<int> occurrences(M, 0);

    for (int v : arr)
        occurrences[v] += 1;

    for (int v = 0; v < M; ++v)
        if (occurrences[v] >= 2)
            return v;

    return -1;
}
```

Task k-th occurrences

Given an array **arr** of size **n**, integer **x**, and an array **ks** of size **m** ($m \le n$) that is **sorted in ascending order**. For each element in ks return an index in arr of $ks[i]$'s occurrence of x. If x does not occur $ks[i]$ times in arr, then return -1.

Time: $O(n+m)$
Space: $O(1)$

The fact that **ks** is already sorted gives us an opportunity to solve this problem using **2 pointers technique**.

Let j denote current index of ks, and $k := ks[j]$. Let cnt be the number of occurrences of x we have encountered in a prefix of arr up to i-th element (inclusive).

Now, if $cnt < k$ then it means that we need iterate over arr starting from i and increment cnt once x encountered until $cnt = k$ or $i = n$.

Once the iteration is stopped, we check whether $cnt = k$, if so, then the $(i-1)$ is the answer for k, otherwise print -1.

Now we need to proof that the time and space complexities are indeed those required.

Proof:

Time: j will iterate over ks, thus it is $O(m)$; the inner **while-loop** can never be executed more the n times because on each iteration i is incremented, thus it is $O(n)$ operations $\implies O(n)+O(m) = O(n+m)$ in total.

Space: we use only constant number of variables, thus it is $O(1)$ of space.

```
void kthOccurrence(vector<int>& arr, vector<int>& ks, int x) {
    /* 'ks' is sorted! */
    const int n = arr.size();
    int i = 0;
    int cnt = 0;

    for (int j = 0; j < ks.size(); ++j) {
        int k = ks[j];

        while (cnt < k && i < n)
            cnt += (arr[i] == x), ++i;

        if (cnt == k)
            cout << (i - 1) << "\n";
        else
```

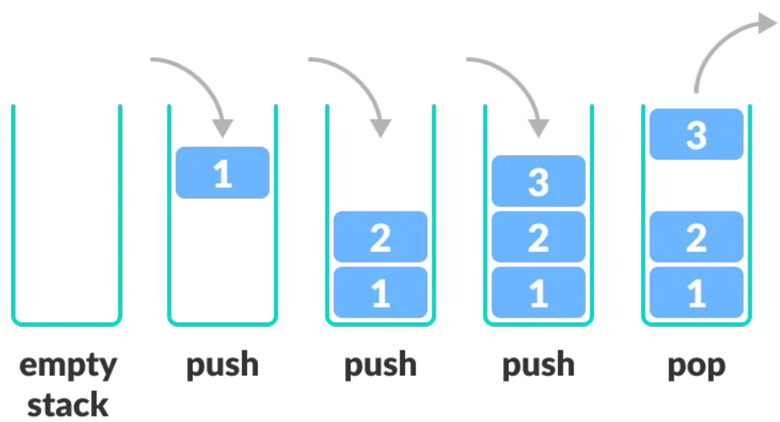
```
            cout << -1 << "\n";
    }
}
```

1 Basic data structures, continuation

Here we will learn stack, queue, and deque. It is important to understand that all 3 are interfaces that could be implemented using data structures such as vector.

1.1 Stack

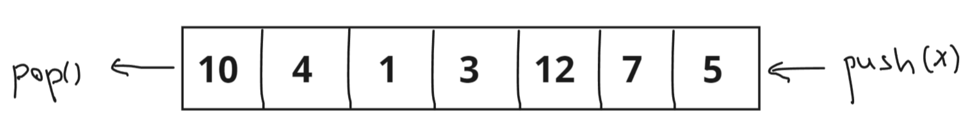
The stack's interface has 2 main operations:
1. $push\_back(x)$ or $push(x)$ - place an element x on top of a stack, $O(1)$ in time.
2. $pop\_back()$ or $pop()$ - remove element from the top, $O(1)$ in time.
Notice that first inserted element will be removed the last. This strategy is called **FILO** (**First In Last Out**).



1.2 Queue

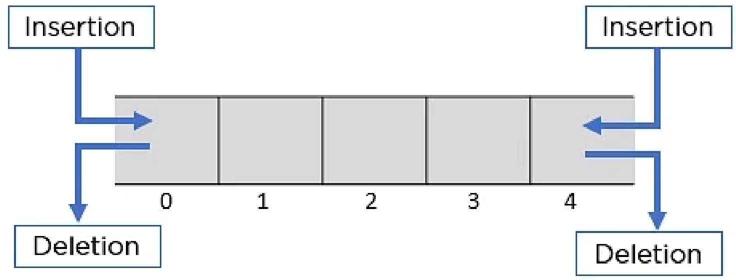
The queue's interface has 2 main operations:
1. $push\_back(x)$ - push an element x into the back of a queue, $O(1)$ in time.
2. $pop\_front()$ - remove and element x from the front of a queue, $O(1)$ in time.
The queue interface conforms to the **FIFO** strategy (**First In First Out**).

If you have ever been in a real queue (e.g. queue in front of a concert entrance) then you are already familiar with this data structure.



1.3 Deque

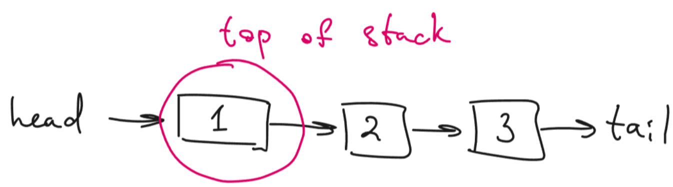
Deque interface supports insertion (push_back, push_front)/removal (pop_back(), pop_front()) from both sides where all operations are in $O(1)$ of time.



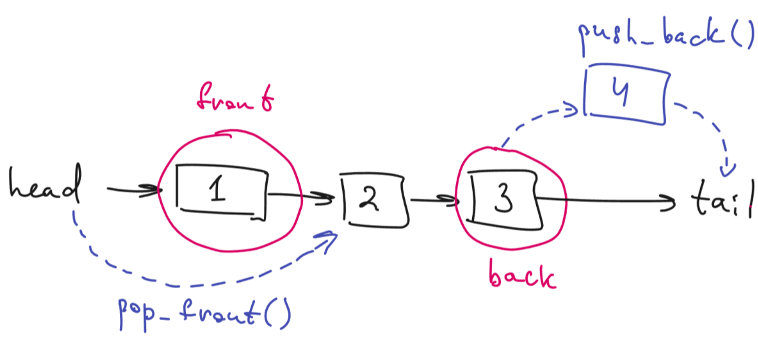
1.4 Implementation

We already know everything required to implement our own stack, queue, deque interfaces using either linked lists or vector.

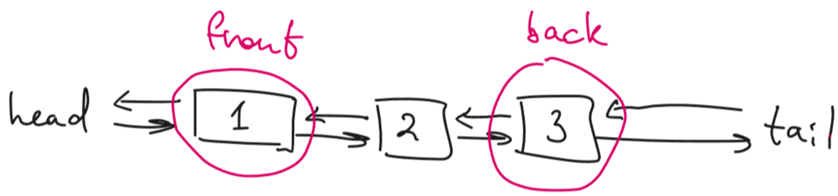
Implementation using linked lists:
1. **Stack:** we need to use only singly linked list by inserting/removing the first node pointed by a head:



2. **Queue:** we also need to use only singly linked list by keeping track of the last element before the tail node to support **push_back** for $O(1)$ in time; to support **pop_front** we remove the element pointed by a head node:



3. **Deque:** here we need to support insert/remove from front and back sides. In order to do that we need for every node to know previous and next elements, thus we can use doubly linked list here:



Implementation using vector:

1. **Stack:** notice that **std::vector** has *push_back*() and *pop_back*() operations which are $O(1)$ in time complexity. Thus, a single vector will do the thing.

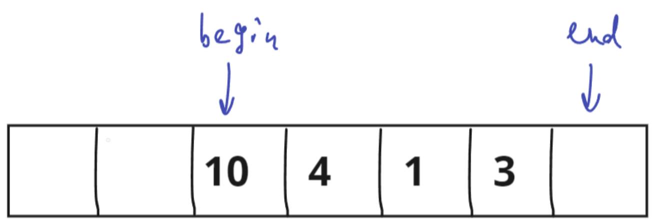
2. **Queue** and **Deque** via cyclic array:

Here we will implement deque and as a consequence we will get queue since its operations are a subset of deque.

We need to keep trace of the **begin** and **end** of the cyclic array where the elements will be placed in range $[begin, end)$.

On *pop_front*/*push_front* we move the **begin** pointer either forward modulo size of the cyclic array (i.e. $begin := (begin + 1)\ mod\ size$) or backward modulo size (i.e. $begin := (begin - 1 + size)\ mod\ size$).

The same strategy is applied to *end* pointer during pop_back/push_back operations:




```
struct deque { vector<int> a; int begin, end; };

int size() { return a.size(); }
int items_count() { return end - begin + (begin <= end ? 0 : size()); }

int get(i) { return a[(i + begin) % size()]; }

void push_front(x) {
    begin = (begin - 1 + size()) % size();
    a[begin] = x;
}

void pop_front() {
    a[begin] = empty;
    begin = (begin + 1) % size();
}

void push_back(x) {
    a[end] = x;
    end = (end + 1) % size();
}
```

Currently the implementation supports only fixed-size deque/queue. In order to support arbitrary size data structures we need to extend the size $n \to 2n$ of the cyclic array once $begin == end$ after some push operation (i.e. if the cyclic array becomes full of elements).

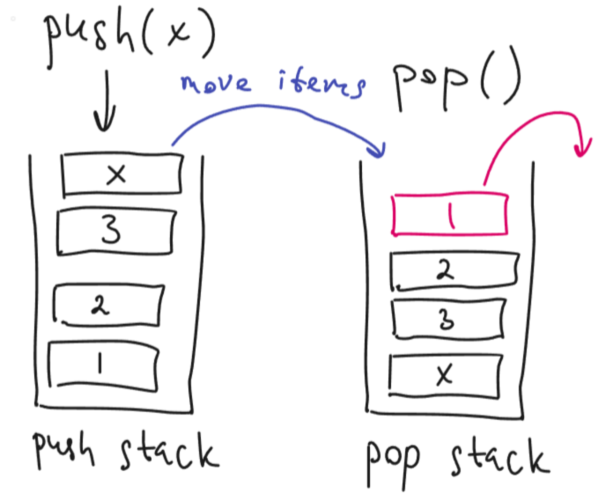
2 Practice: Implement queue/deque using stacks

2.1 Queue

Examine the following scenario: you have pushed elements 1, 2, 3, and x into queue. Want to pop one element: this element must be 1 since it was the one first inserted.

Imagine you have 2 stacks: 1st is used to push elements and the 2nd is used to pop elements. If the 2nd pop-stack is empty and **pop()** operation requested we need to move all the elements from the 1st push-stack into pop-stack by extracting elements from the top of push-stack and pushing them on top of pop-stack until the 1st stack becomes empty.

Such moving strategy will make the bottom element of push-stack to be the top element of pop-stack which will allow us to extract it to satisfy **pop()** operation request.



The question is: Why does it work in $O(1)$ time complexity on average?

Let's prove it:

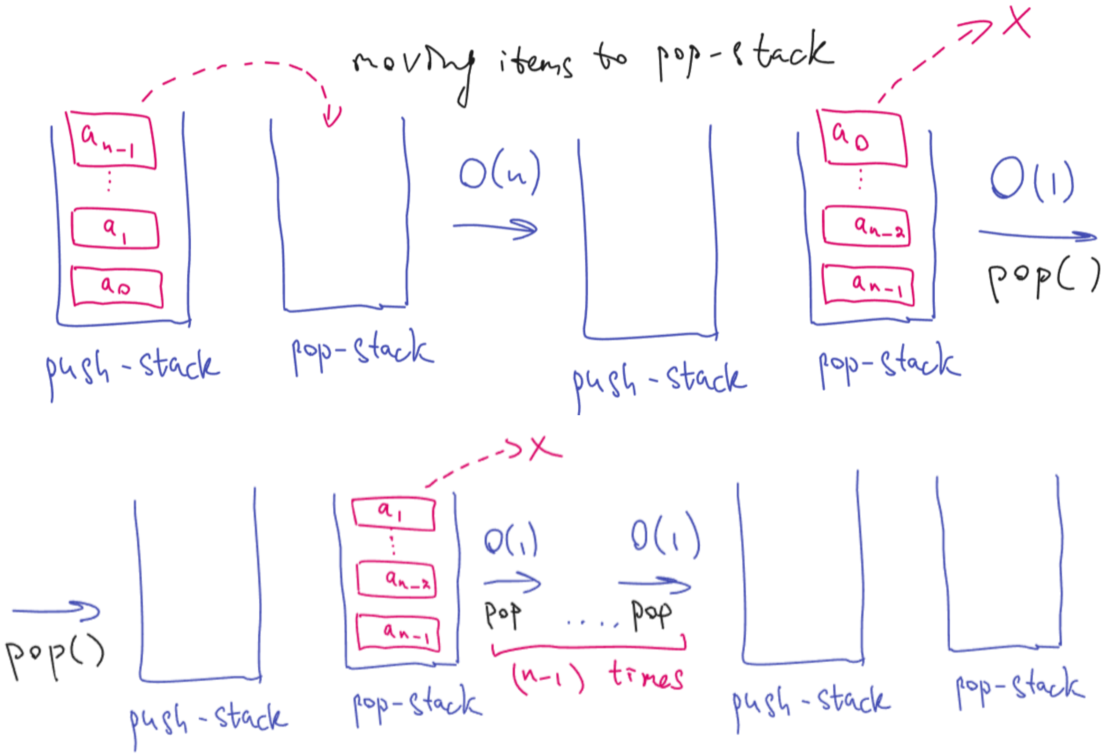
1. Suppose we have executed n **push** operations which pushed elements a_0, a_1, ..., a_{n-1} into push-stack (i.e. a_{n-1} will be on top of the push-stack, and a_0 in the bottom).

2. Now, **pop()** operation is requested: we need to move all the elements from push-stack into pop-stack which takes $O(n)$ time. After that we extract the element a_0 which will be placed on top of the pop-stack.

3. Notice that the following $(n-1)$ **pop()** operations will be executed in $O(1)$ of time because pop-stack will contain some elements.

Thus, the average time of **pop()** operation is as follows (dividing total execution time by the number of **pop** operations):

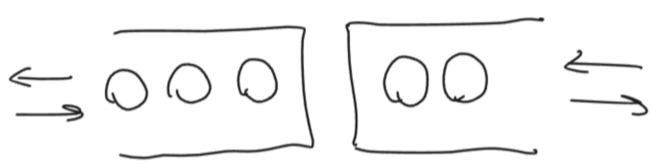
$$T = \frac{O(n) \cdot 1 + O(1) \cdot (n-1)}{n} = \frac{O(n) \cdot 1 + O(n)}{n} = \frac{O(n)}{n} = \Theta(1)$$



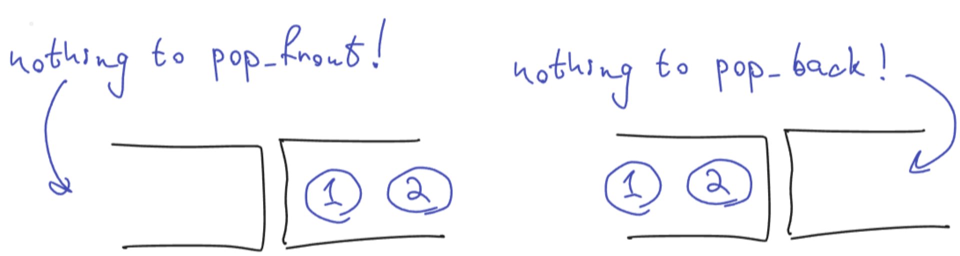
2.2 Deque

Now, let's try to implement deque using 3 stacks.

Notice, that deque could be split into 2 stacks where one is responsible for **front** side (push_front, pop_front) and the other for the **back** side (push_back, pop_back):



But there is an issue with this perception: What should we do once we have one of the following states and need to execute **pop_front**/**pop_back** respectively:



On the left picture we need somehow to remove the element 1 during **pop_front** operation, and on the right remove the element 2.

Remember the technique of moving all elements from the push-stack to the pop-stack in the queue implementation?

The idea will work but the solution will not be sufficient in terms of execution time, since every pop operation may start to work for $O(n)$ (consequent **pop_front** and **pop_back** will trigger moving elements from one stack to another resulting in $O(n)$ for each operation).

How to make both operations work in $O(1)$ of time in average?

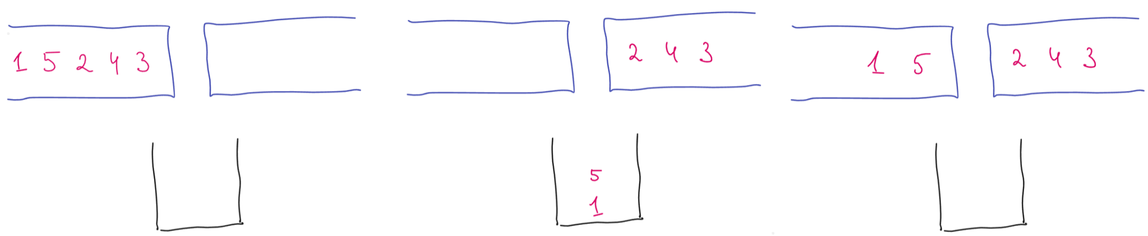
Idea: what if both front and back stack would contain $\frac{n}{2}$ elements: We will balance both stacks to contain $\frac{n}{2}$ elements once a pop operation encounters its stack to be empty.

We will show the correctness of this idea and then proof that the average time is $O(1)$.

1. **Correctness**:

Assume that we are in a situation where back-stack is empty. We want to move $\frac{n}{2}$ elements from front-stack into back-stack preserving the correct order.

Let's move first half of elements into **3rd helper-stack**, then move the rest elements from front-stack into back-stack, and after return elements placed in helper-stack back to front-stack (pictures from left to right):



The order remains the same, thus the implementation is correct.

2. **Average Time Complexity is $O(1)$**:

All operations are executed in $O(1)$ when both front-stack and back-stack contain some elements. Thus, we only need to examine the scenario when one of the stacks becomes empty (assume that back-stack is empty as in the pictures above):

1) Notice that each element from front-stack is touched **not more than twice**: once when moved into either helper-stack of back-stack, and some of the elements are touched 2nd time during returning back from helper-stack into front-stack. Thus, the operation requires $O(n)$ in time.

2) But now both front-stack and back-stack contain $\frac{n}{2}$ elements, thus **at least** $\frac{n}{2}$ of the following **pop_front/pop_back** operations will be executed in $O(1)$, thus the average time is:

$$T \leq \frac{O(n) \cdot 1 + \frac{n}{2} \cdot O(1)}{\frac{n}{2}} = \frac{O(n)}{\frac{n}{2}} = \Theta(1)$$

3 Practice: min stack, min queue

Task Min stack

Design a stack that supports **push()**, **pop()**, and **retrieveMin()** in **constant time**, i.e. $O(1)$.

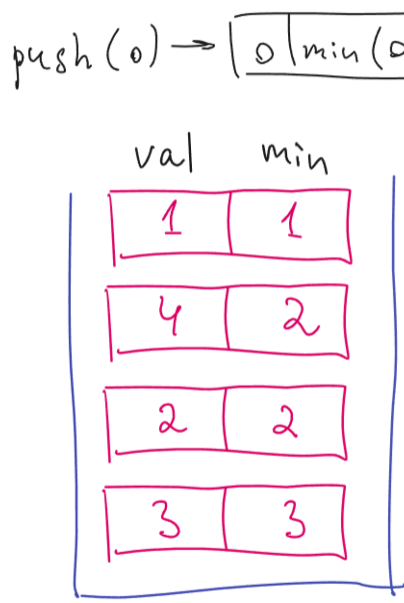
Solution

Suppose that we know the minimum among the already added elements, and we receive a **push(x)** operation: **What is the minimum now?**

Current minimum becomes **min(x, prevMin)**.

Idea: For every added element a_i let's keep minimum among all elements that are under a_i, i.e. i-th entry of a stack is a pair $< a_i,\ min(a_i,\ a_{i-1},\ a_{i-2},\ ...,\ a_0) >$ where $\forall j:\ j < i:\ a_j$ are elements under a_i.

In order to implement this idea upon every **push(x)** operation we need to insert a new pair into the stack which will be: **<x, min(x, prevMin)>** where **prevMin** is a min value stored on top of the stack.




```
struct MinStack {
    vector<pair<int, int>> items;
};

void push(int x) {
    int prevVal, prevMin;
    std::tie(prevVal, prevMin) = min_stack.items.back();
    min_stack.items.push_back(std::make_pair(x, std::min(x, prevMin)));
}

void pop() {
    min_stack.items.pop_back();
}

int get_min() {
    int val, min;
    std::tie(val, min) = min_stack.items.back();
    return min;
}
```

Task Min queue

Design a queue that supports **push_back()**, **pop_front()**, and **retrieveMin()** operations in $O(1)$ of time.

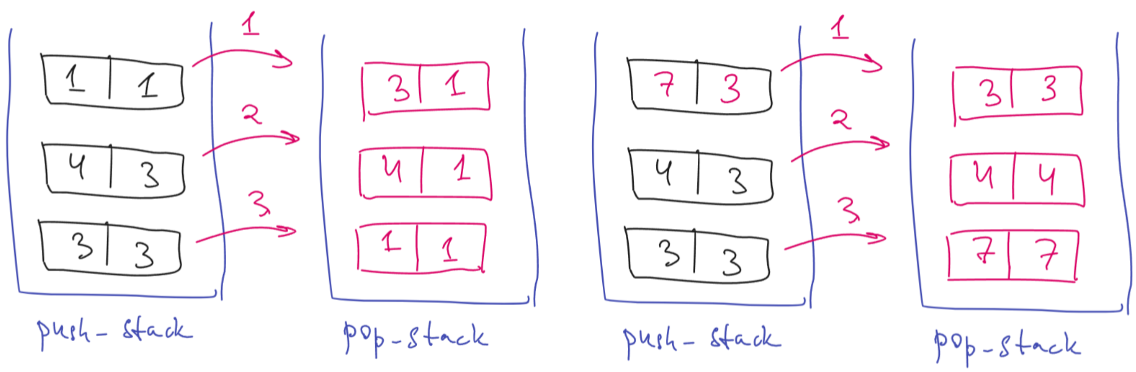
Solution

Remember the implementation of a queue using 2 stacks?

Let's use that implementation and for both push-stack (s1) and pop-stack (s2) we will keep a minimum as in the solution of a **min stack** problem above.

Now, the minimum in a queue is the minimum among both minimums from push-stack and pop-stack: $min(s_1.top().min,\ s_2.top().min)$

The process of moving elements from push-stack into pop-stack should be altered slightly: We need to relax the minimum of each extracted pair **<x, m>** from push-stack by a minimum of **x** and **s2.top().min**, and place a new pair into s2: **<x, min(x, s2.top().min)>**.




```
// Note: checks for emptiness omitted
struct MinQueue {
    vector<pair<int, int>> s1; // push-stack
    vector<pair<int, int>> s2; // pop-stack
};

void push(int x) {
    pair<int, int> p = { x, std::min(x, s1.top().second) };
    s1.push_back(p);
}

void pop() {
    if (s2.empty()) {
        while(!s1.empty()) {
            int val, min;
            std::tie(val, min) = s1.back(); s1.pop_back();
            s2.push_back({ val, std::min(val, s2.back().second) });
        }
    }
    s2.pop_back();
}

void get_min() {
    return std::min(s1.back().second, s2.back().second);
}
```