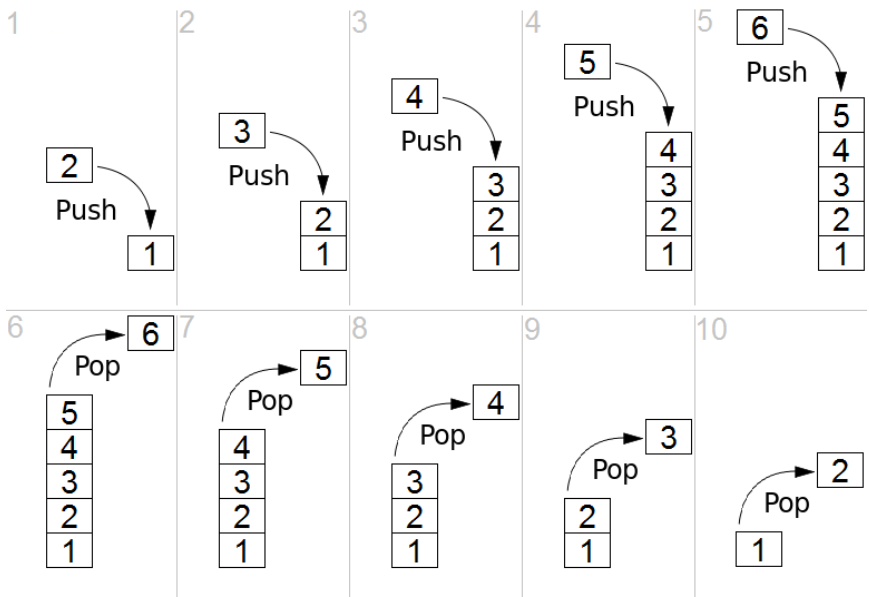
Stacks

Stacks = LIFOs (Last-In-First-Out)

Stack is an **abstract data type** defined by its three operations:

- `push(x)` puts value `x` on the *top* of the stack
- `pop()` takes out a value from the *top* of the stack
  If there are no values in the stack, it raises
  `EmptyStackException`.
- `is_empty()` says whether the stack is empty



(picture source: wikipedia)

Stack is an abstract data type. A stack is a list of values. The two ends of the list are called the *bottom* and *top*. We *push* a value (add it to the top of the stack) and *pop* a value (remove from the top of the stack). Thus a stack behaves in a Last In First Out (LIFO) manner. If we try to pop from an empty stack, we get an `EmptyStackException`. In theory we should be able to push any number of values, but in practice that won't be possible and we will get an exception if we run out of memory.

We can have a stack of any type of value, e.g. a stack of integers, a stack of strings, a stack of Booleans etc.

As a part of the specification of stacks it is usually also said that `push(x)` followed by `is_empty()` gives `false`, and `push(x)` followed by `pop()` gives `x` back.

Since we see a stack as an abstract data type, we do not specify how it is implemented.

Example

Suppose we create an empty stack and we push 3, push 5, push 2 and pop; we get 2. Suppose we then pop; we get 5. Suppose we push 1, push 8, then pop. Pop again three times, what do we get?

Usage

For example, when we are solving tasks with dependencies.

Imagine that we want to complete a task A, `push(A)`, and

1. A depends on B and C $\implies$ `push(B)` and `push(C)`
   (ok, we need to solve C first but ...)
2. C depends on D $\implies$ `push(D)`.

Once we complete D (`pop()`), C (`pop()`), and B (`pop()`), we can also complete A (`pop()`).

(More verbose example of usage:)

One reason that stacks are useful is that sometimes, in order to complete job A we must first do job B and then job C, but in order to complete job B we must first do job D, and in order to do that we must first do job E and job F. Using a stack, we can push the primary job onto the stack first and each time we push any job onto the stack, we follow it by pushing the jobs that it depends on. So long as you then execute the jobs in the order that pop retrieves them, all the jobs will only be executed when the jobs they depend upon are complete.

Another example: the most natural way of evaluating an expression written in *reverse Polish notation* is by using stacks. This is because sub-expressions have to be evaluated before the higher level expressions that must use them, hence evaluating a sub-expression is a job that must be completed before we can evaluate the higher level expressions

Stacks are heavily used in applications that involve exhaustive searches of some problem space and in constructing and searching tree structures.

Stack ADT

Here is a possible list of operations for a Stack ADT (many variations are possible)[1]

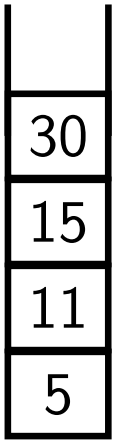
Constructors and Accessors:

- `EmptyStack` : returns an empty Stack
- `push(element, stack)` , pushes an element on top of the given stack.
- `top(stack)` : returns the value at the top of the stack without changing the stack[2]
- `pop(stack)` : returns the stack with the top element removed[2]
- `isEmpty(stack)` : reports whether the stack is empty

[1] Read chapters 1 and 2 of the module handouts

[2] Triggers error if the stack is empty

Stacks as linked lists



To store a stack as a linked list we pick the faster of the two options:

1. the top is at the beginning, i.e. $\langle 30, 15, 11, 5 \rangle$
2. the top is at the end, i.e. $\langle 5, 11, 15, 30 \rangle$

Question: Which one is better and why?

Since inserting and deleting from the beginning of a linked list is constant, the first option is better. In other words, we take

- `push` = `insert_beg`
- `pop` = `delete_beg`
- `is_empty` (for stacks) = `is_empty` (for linked lists)

This way every operation on stacks takes constant time.

The second option would mean that `push` = `insert_end` and `pop` = `delete_end`. Then, even if we stored the position of the end of the linked list (to make sure `insert_end` is fast), `delete_end` would still be slow (linear time) and so the second option is not reasonable.

Stacks as arrays

One can implement Stacks using a simple array in Java:

```
1 // Initialize an empty stack:
2 stack = new int[MAXSTACK];
3 stack_size = 0;
```

Here the bottom of the stack is stored on position `0` of the array and the top of the stack in position `stack_size-1`. We can implement `push` and `pop` for this representation in constant time.

$\Longrightarrow$ No matter if we store stacks as linked lists or arrays, in both cases, `push`, `pop`, and `is_empty` finish in *constant time*.

Storing stacks as arrays has the advantage that we avoid calling `allocate_memory` all the time (this takes time, even if it is done automatically for us, like in Java). On the other hand, we need to know the maximum size of the stack in advance.

In practice, in Java, we normally use the library class `Deque<...>`, which implements the Stack ADT as well as some others we will discuss and adjusts to grow as the stack increases in size. We will see this later in this lecture.