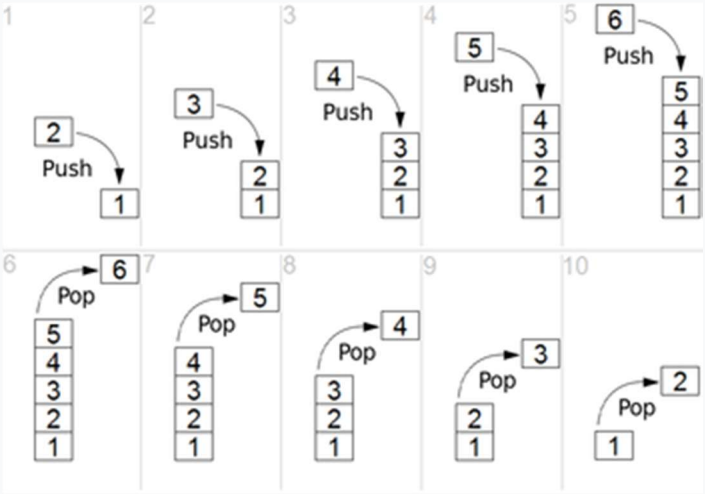
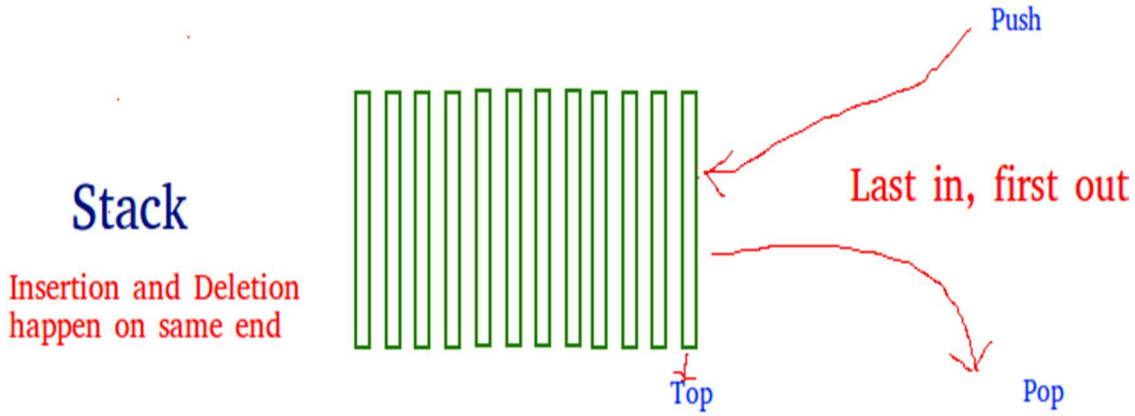
Stack

- Stacks entered the computer science literature in 1946, when Alan M. Turing used the terms "bury" and "unbury" as a means of calling and returning from subroutines.
- Subroutines had already been implemented in Konrad Zuse's Z4 (First Commercial digital Computer) in 1945.
- Klaus Samelson and Friedrich L. Bauer of Technical University Munich proposed the idea of a stack in 1955 and filed a patent in 1957.
- In March 1988, by which time Samelson was deceased, Bauer received the IEEE Computer Pioneer Award for the invention of the stack principle.
- Similar concepts were developed, independently, by Charles Leonard Hamblin in the first half of 1954 and by Wilhelm Kämmerer in 1958.
- Stacks are often described using the analogy of a spring-loaded stack of plates in a cafeteria.
- Clean plates are placed on top of the stack, pushing down any already there.
- When a plate is removed from the stack, the one below it pops up to become the new top plate.
- **Example**
- **There are many real-life examples of a stack (as discussed above).**
    - Consider an example of plates stacked over one another in the canteen.
    - The plate which is at the top is the first one to be removed,
    - i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time.
    - So, it can be simply seen to follow LIFO (Last In First Out)/FILO(First In Last Out) order.

Similar to a stack of plates, adding or removing is only possible at the top.

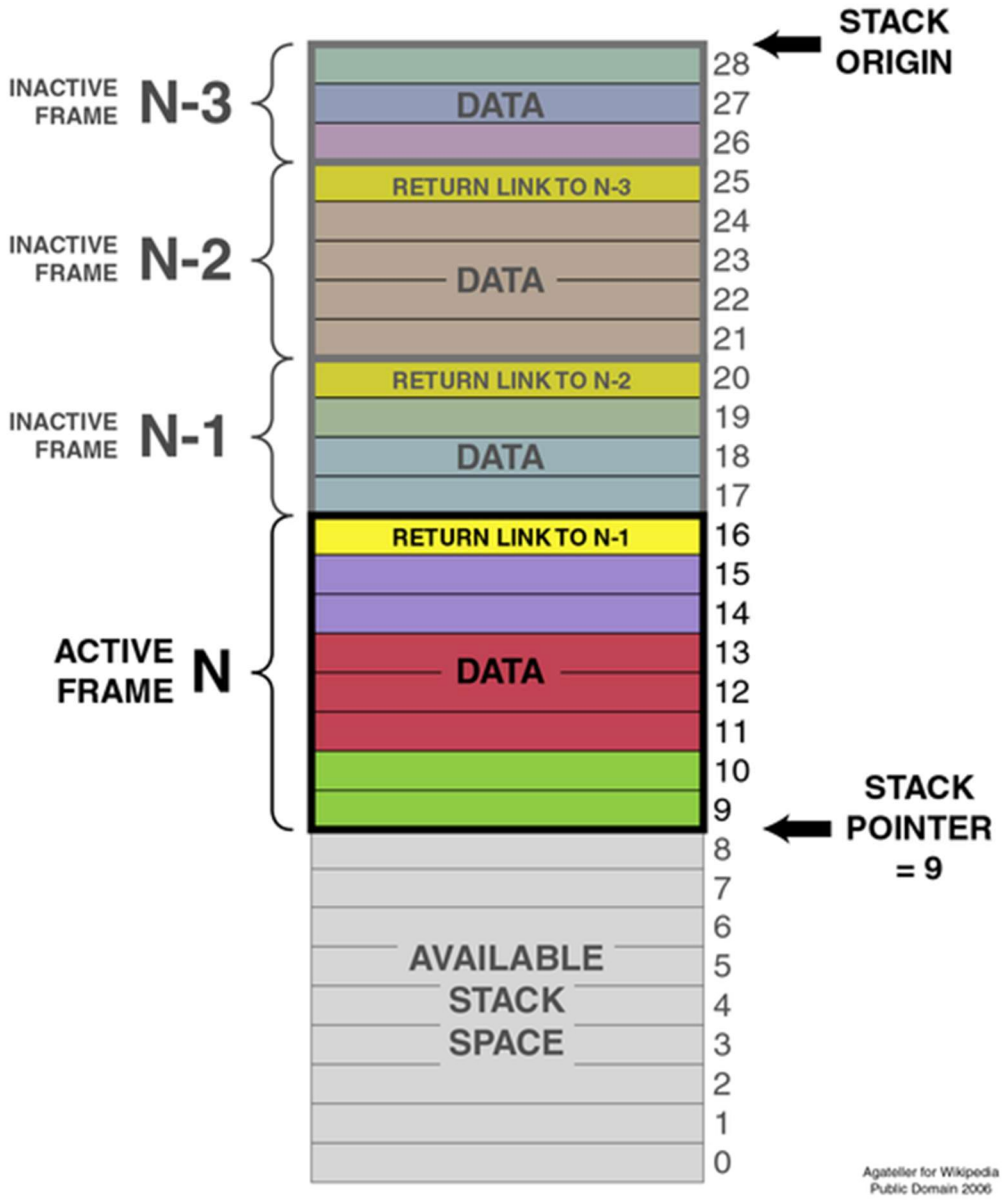


Simple representation of a stack runtime with *push* and *pop* operations.



Stack Typical Structure

- A typical stack, storing local data and call information for nested procedure calls (not necessarily nested procedures).
- This stack grows downward from its origin.
- The stack pointer points to the current topmost datum on the stack.
- A push operation decrements the pointer and copies the data to the stack; a pop operation copies data from the stack and then increments the pointer.
- Each procedure called in the program stores procedure return information (in yellow) and local data (in other colors) by pushing them onto the stack.
- This type of stack implementation is extremely common, but it is vulnerable to buffer overflow attacks.



Definition:

In computer science, a **stack** is an abstract data type that serves as a collection of elements, with two **main principal operations**:

- **Push**, which adds an element to the collection, and
- **Pop**, which removes the most recently added element that was not yet removed.

- The order in which elements come off a stack gives rise to its alternative name, **LIFO** (**last in, first out**).
- Additionally, a peek operation may give access to the top without modifying the stack.
- The name "stack" for this type of structure comes from the analogy to a set of physical items stacked on top of each other.
- This structure makes it easy to take an item off the top of the stack, while getting to an item deeper in the stack may require taking off multiple other items first.
- Considered as a linear data structure, or more abstractly a sequential collection, the push and pop operations occur only at one end of the structure, referred to as the *top* of the stack.
- This data structure makes it possible to implement a stack as a singly linked list and a pointer to the top element.
- A stack may be implemented to have a bounded capacity.
- ***If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state***.
- The pop operation removes an item from the top of the stack.
- ***If the stack is empty and does not contain any space to accept an entity to be popped, the stack is then considered to be in an Underflow state***.
- A stack is needed to implement depth-first search.
- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Some more operations:

- In many implementations, a stack has more operations than the essential "push" and "pop" operations.
- An example of a non-essential operation is "top of stack", or "peek", which observes the top element without removing it from the stack.
- This could be done with a "pop" followed by a "push" to return the same data to the stack, so it is not considered an essential operation.

- If the stack is empty, an underflow condition will occur upon execution of either the "stack top" or "pop" operations.
- Also, implementations often have a function which just returns whether the stack is empty.

Implementation

- A stack can be easily implemented either through an array or a linked list.
- What identifies the data structure as a stack, in either case, is not the implementation but the interface:
- The user is only allowed to pop or push items onto the array or linked list, with few other helper operations. The following will demonstrate both implementations, using pseudocode.

Through Array

- An array can be used to implement a (bounded) stack, as follows.
- The first element, usually at the zero offset, is the bottom, resulting in array [0] being the first element pushed onto the stack and the last element popped off.
- The program must keep track of the size (length) of the stack, using a variable *top* that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted (assuming a zero-based index convention).
- Thus, the stack itself can be effectively implemented as a three-element structure:

```
structure stack:
    maxsize : integer
    top : integer
    items : array of item
procedure initialize(stk : stack, size : integer):
    stk.items ← new array of size items, initially empty
    stk.maxsize ← size
    stk.top ← 0
```

The *push* operation adds an element and increments the *top* index, after checking for overflow:

```
procedure push(stk : stack, x : item):
    if stk.top = stk.maxsize:
        report overflow error
```

```
    else:
        stk.items[stk.top] ← x
        stk.top ← stk.top + 1
```

Similarly, *pop* decrements the *top* index after checking for underflow, and returns the item that was previously the top one:

```
procedure pop(stk : stack):
    if stk.top = 0:
        report underflow error
    else:
        stk.top ← stk.top − 1
        r ← stk.items[stk.top]
        return r
```

- Using a dynamic array, it is possible to implement a stack that can grow or shrink as much as needed.
- The size of the stack is simply the size of the dynamic array, which is a very efficient implementation of a stack since adding items to or removing items from the end of a dynamic array requires amortized O(1) time.

Through Linked list

- Another option for implementing stacks is to use a singly linked list.
- A stack is then a pointer to the "head" of the list, with perhaps a counter to keep track of the size of the list:

```
structure frame:
    data : item
    next : frame or nil
structure stack:
    head : frame or nil
    size : integer
procedure initialize(stk : stack):
    stk.head ← nil
    stk.size ← 0
```

Pushing and popping items happens at the head of the list; overflow is not possible in this implementation (unless memory is exhausted):

```
procedure push(stk : stack, x : item):
    newhead ← new frame
```

```
    newhead.data ← x
    newhead.next ← stk.head
    stk.head ← newhead
    stk.size ← stk.size + 1
procedure pop(stk : stack):
    if stk.head = nil:
        report underflow error
    r ← stk.head.data
    stk.head ← stk.head.next
    stk.size ← stk.size - 1
    return r
```