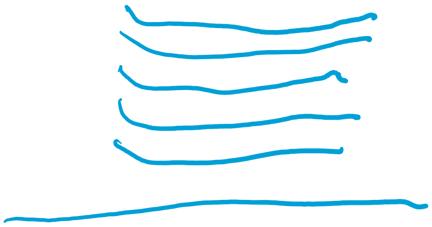
# (1) The Stack Data Structure

- Last In First Out data structure

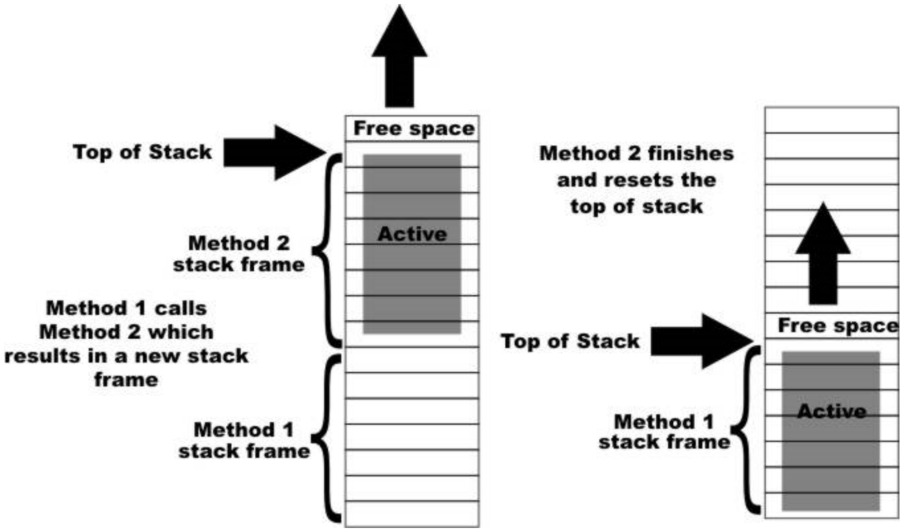
LIFO

- A "limited access" DS
    - can only add to the top (push)
    - can only remove from the top (pop)

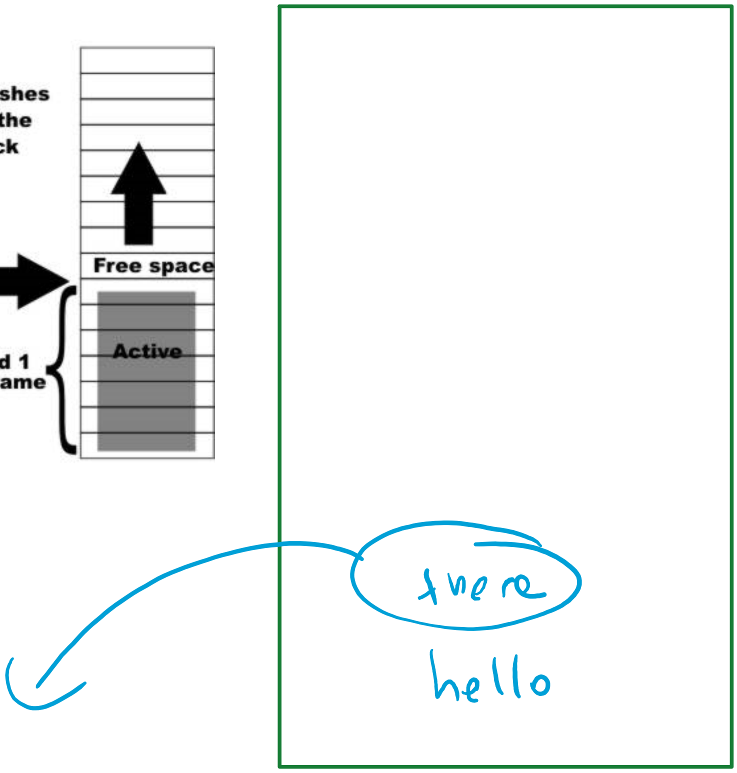
- Usage examples:



example: Undo stack in editor



**Note: the call stack is just one example usage of a stack data structure.**



## (2) Stack ADT

private:

- top - keeps track of the top element
- maxSize - limit on total size of stack (optional - depends on implementation)
- count - current number of elements in stack

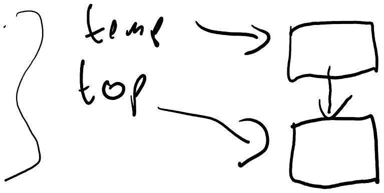
public:

- initialize() - constructor
- bool = isFull() - check whether stack is full
- bool = isEmpty() - check if empty
- value = peek() - show top item
- push(item) - add new item to the top
- pop() - remove from top
- disp() - print contents

Note that the ADT does not specify anything about the implementation.

Array or Linked List

# (3) Stack SLL implementation 0



```
struct Node{
   std::string item;
   Node *next;
};
class Stack{
private:
   // pointer to top of stack
   Node *top;
   // number of nodes currently in stack
   int count;
public:
   Stack(); // constructor
   ~Stack(); // destructor
   bool isEmpty();
   void push( string newItem );
   // Precondition: newItem parameter is a string type
   // Postcondition: dynamically allocate a new nodea and push onto stack
   void pop();
   // Precondition: none
   // Postcondition: remove the node from top of stack and deallocate the
   // node's memory

   Node* peek();
   // Precondition: none
   // Postcondition: return a pointer to the node that corresponds to the
   // top of stack

   void disp();
   // Precondition: none
   // Postcondition: display the contents of entire stack
};
```

(4) Stack SLL impementation

1

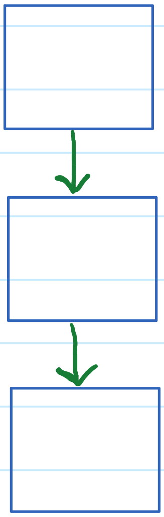
```
Stack::Stack(){
// todo
}

bool Stack::isEmpty(){
// todo
}

// isFull - not needed

void Stack::push( string newItem ){
// todo
}


void Stack::pop(){
}
```

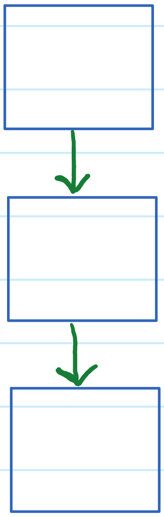


(5) Stack SLL impementation
2

```
Node* Stack::peek(){
   return top;
}
void Stack::disp(){
}


Stack::~Stack(){

}
```



# (6) Stack Array Implementation

```
#define MAXSIZE 9 // set max size for stack
class StackArr{
private:
   int top, count; // Index for next available element and total count
   std::string a[MAXSIZE]; // Stack array

public:
   StackArr(); // Constructor
   bool isEmpty();
   bool isFull();
   void push( string newItem );
   // Precondition: newItem parameter is a string type
   // Postcondition: dynamically allocate a new nodea and push onto stack
   string pop();
   // Precondition: none
   // Postcondition: remove the element from top of stack and update top index
   void disp();
   // Precondition: none
   // Postcondition: display the contents of entire stack
};
```

top = 3

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| hello | what | is | | going | on | | | |

push("hello")
push("what")
push("is")
push("even")
x = pop()
push("going")
push("on")

x = "even"

# (7) Stack implementations pros and cons

In summary: we can implement a stack with an underlying array or Linked List.
How do we decide which one to choose?

Array based:

Pros:

- *fast*

Cons:

- *fixed size*
    - *if using dynamic memory, not linear speed*

LL based:

Pros:

- *no need to set size*

Cons:

- *more constant overhead (the things we drop in big-O)*