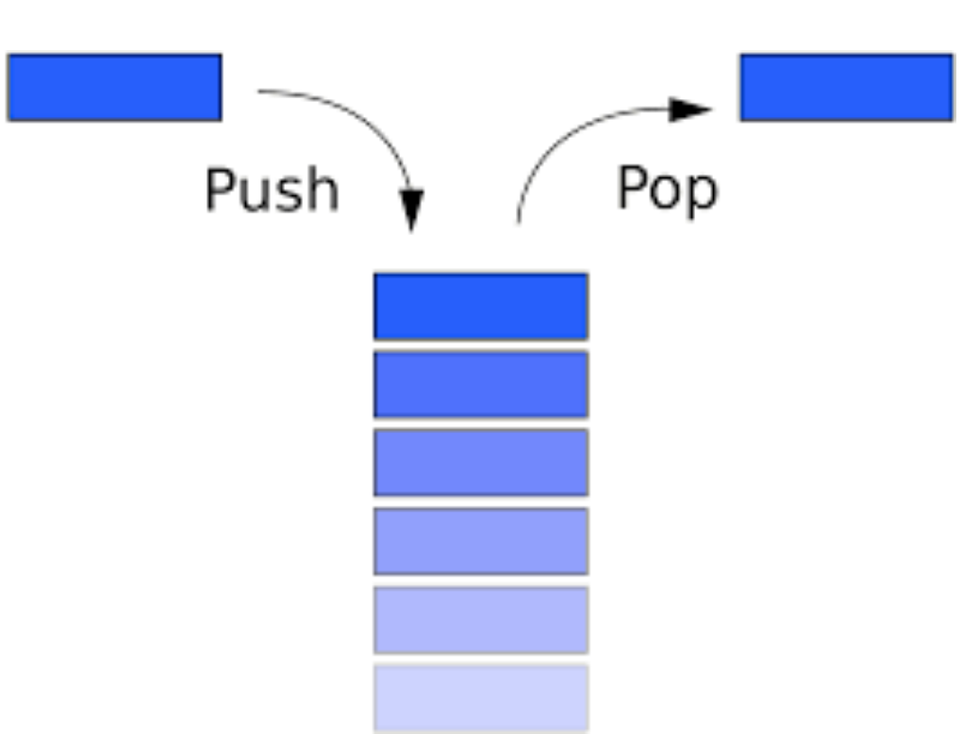
Stacks

The Stack

The stack is a "**LIFO**" data structure.

Basic operations:

- **`push`**
- **`pop`**
- **`peek`** (sometimes equivalent to pop)



An ADT for Stacks

```
template<class T>
class StackInterface
{
 public:
   virtual bool isEmpty() const = 0;
   virtual bool push(const T& newEntry) = 0;
   virtual bool pop() = 0;
   virtual T peek() const = 0;
};
```

We need a specific implementation now..

A flexible choice: the Linked List template class

```
#include "StackInterface.h"
#include "Node.h"
#include <cassert>

template<class T>
class LinkedStack : public StackInterface<T>
{
 private:
  Node<T>* topPtr;

 public:
  LinkedStack(); //constructor
  LinkedStack(const LinkedStack<T>& aStack);//copy constructor
  virtual ~LinkedStack();//destructor

  bool isEmpty() const;
  bool push(const T& newItem);
  bool pop();
  T peek() const;
};
```

Used for stopping the program in case of errors, see next slides.

We need the Node class again (see “Bag” data structure)

Methods: Implementation

```
//Destructor
template<class T>
LinkedStack<T>::~LinkedStack()
{
    while (!isEmpty()) pop();
}
```

```
template<class T>
bool LinkedStack<T>::isEmpty() const
{
  return topPtr == nullptr;
}
```

Methods: Implementation

```
template<class T>
bool LinkedStack<T>::push(const T& newItem)
{
  //Create a new node and make it the new top node
  Node<T>* newNodePtr = new Node<T>(newItem,topPtr);
  topPtr = newNodePtr;
  newNodePtr = nullptr;

  return true;
}
```

```
template<class T>
T LinkedStack<T>::peek() const
{
  assert(!isEmpty());
  return topPtr->getItem();
}
```

```
template<class T>
bool LinkedStack<T>::pop()
{
  bool result = false;
  if (!isEmpty()){

    //Delete the top node
    Node<T>* nodeToDeletePtr = topPtr;
    topPtr = topPtr->getNext();

    nodeToDeletePtr->setNext(nullptr);
    delete nodeToDeletePtr;
    nodeToDeletePtr = nullptr;

    result = true;
  }
  return result;
}
```

Example Applications

The Stack is very useful in many situations and helps realizing in practice several kinds of concepts.

Some simple examples:

- Reverse the order of a string
- Check for balanced braces in an expression
- Evaluating algebraic expressions (e.g.: infix, postfix, …)

An important application of the Stack concept is related to **RECURSION**.

Stacks and Recursion

Recursion has to do with functions calling themselves.

Question: if a compiler+language allows the use of recursion,is it realized in practice?

WITH A STACK!

When recursion takes place, the system has to “remember” the current state before the next recursive call to the function.
The Stack provides this “memory”.

We know that it is always possible to convert recursion to iteration and *viceversa*. The conversion is straightforward in the case of tail recursion.

What about the other cases?
If the compiler uses a stack for implementing recursion, this means that if we have a stack, we can for sure convert a recursion in a program with iteration+stack!!!

A simple example: the Factorial again.

We already know the iterative and recursive implementations of the factorial function. Now consider this iteration+stack:

```
#include "LinkedStack.h"
#include <iostream>

using namespace std;

int factorial(int n)
{
  LinkedStack<int> *stack = new LinkedStack<int>();
  stack->push(1);
  for (int i=1 ; i<=n ; i++) stack->push( i * stack->peek());
  return stack->peek();
}

int main()
{
  //Factorial with a stack:
  cout << factorial(5) << endl;
  return 0;
}
```

Exercise / Assignment

1) Using iteration+stack, implement the Fibonacci-numbers function!
2) You can use the stack implementation you prefer.
3) Discuss from the algorithmic complexity point of view the stack functions compared with the “Bag” functions we have seen before.

Print on paper the code (not the stack implementation, just the Fibonacci function) and your discussion of point 3).

The assignment must be given to the instructor by next Friday during the class.