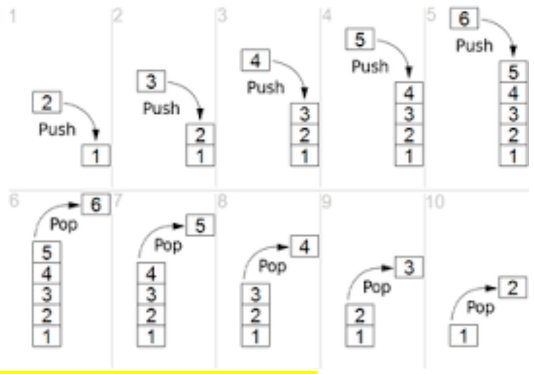
Objective:
- Intro to data structure, stack.
- Applications of Stack.
- An optimized way of resizing the stack.

What is Stack?
A data structure which exhibits LIFO (last in first out) behaviour. It means the element added last will be removed first. In stack, the basic insertion and removal operations are named as push and pop respectively. Push adds an element on top of the stack and pop removes an element from top of the stack.

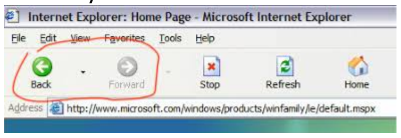


Where do you see such behavior in daily life?

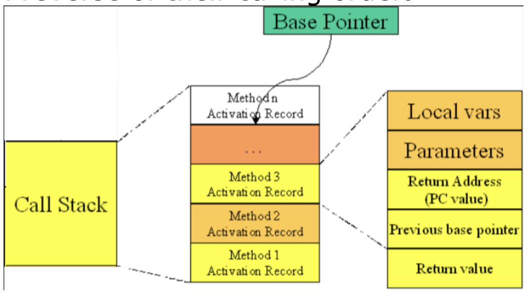
- Stack of Books/Plates

- Order of adding/removing plates to weight bar.

Where do you see such behaviour/application in CS?
- Think about Undo/Redo behaviour in different type of editors.
- Browser History i.e. Page visit history is maintained in LIFO order
  - Observe the behavior of back/forward button in internet explorer.


Internet Explorer: Home Page - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Forward
Stop
Refresh
Home
Address
http://www.microsoft.com/windows/products/winfamily/ie/default.mspx

- Activation Record
  - Local variables/objects in a function: in which order they are created and destroyed.
- Function Call Stack
  - Functions return in reverse of their calling order.


Base Pointer
Call Stack
Method n Activation Record
. . .
Method 3 Activation Record
Method 2 Activation Record
Method 1 Activation Record
Local vars
Parameters
Return Address (PC value)
Previous base pointer
Return value

  - A useful/interesting link to study in this regard.
    - https://en.wikipedia.org/wiki/Call_stack#Structure
- Expression Evaluation

- a+b/(c-d*c)+e
  - Will share a document about it before next online session
- Inherent behavior of Recursion: our upcoming topic.
- Backtracking: An algorithmic design technique. Stack may help to tackle/implement basic backtracking problems/solutions otherwise we got to move towards recursion.
  - Should read this link for the interest: https://www.geeksforgeeks.org/backtracking-introduction/
  - One of the basic application is discussed in your textbook-A: section-3.3 'A Mazing Problem'.

How to resize Stack? How to grow or shrink the size of Stack?

How to grow the size of Stack?

Method-1

- In push operation: increase size by 1.
- In pop operation: decrease size by 1.

Performance

- We need to copy all the items stored in old array to new array.
- Inserting first N items takes time proportional to **$N^2/2$**:
  - On 1st push = 1
  - On 2nd push = 2
  - On 3rd push = 3
  - …
  - …
  - On Nth push = N
- Memory overhead is zero

→ We need to ensure that array resizing should not happen too frequently.

Method-2

If array is full, create a new array of twice the size, and copy items.

- In push operation: double the size of array when it is full.
- In pop operation: discussed at end.

Performance

- We need to copy all the items stored in old array to new array.
- Inserting first N items takes time proportional to **2N**:
  Resizing needed at
  - 1st push = 1
  - 2nd push = 2
  - 3rd push = 4
  - 5th push = 8
  - 9th push = 16
  - 17th push = 32
  - …
  - …
  - N + (1 + 2 + 4 + 8 + … + N) ~ **3N** (not N^2).

→ Array resizing will happen infrequently. ☺

How to shrink the size of Stack?

Method-1

- In push operation: double the size of array when it is full.
- In pop operation: halve the size of array when it is one-half full.

Performance

- Consider push-pop-push-pop---- sequence when array is full.
  - Each operation takes time proportional to N as shown in figure below.

N = 4	Go	Slow	Or	Go				
N = 5 *push*	Go	Slow	Or	Go	Fast			
N = 4 *pop*	Go	Slow	Or	Go				
N = 5 *push*	Go	Slow	Or	Go	Fast			
N = 4 *pop*	Go	Slow	Or	Go				

Method-2

- In push operation: double the size of array when it is full.
- In pop operation: halve the size of array when it is one-quarter full.

Performance

- Amortized analysis. Starting from an empty data structure, average running time per operation over a worst-case sequence of operations.
- Proposition. Starting from an empty stack, any sequence of M push and pop operations takes time proportional to M.

Array shrink/grow Trace

Push	**Pop**	**Items count**	**N = Array Size**	**Array**							
				0	1	2	3	4	5	6	7
		0	0								
Go		1	1	Go							
Slow		2	2	Go	Slow						
Or		3	4	Go	Slow	Or					
Go		4	4	Go	Slow	Or	Go				
Fast		5	8	Go	Slow	Or	Go	Fast			
	Fast	4	8	Go	Slow	Or	Go				
Slow		5	8	Go	Slow	Or	Go	Slow			
	Slow	4	8	Go	Slow	Or	Go				
	Go	3	8	Go	Slow	Or					
Ok		4	8	Go	Slow	Or	Ok				
	Ok	3	8	Go	Slow	Or					
	Or	2	4	Go	Slow						
	Slow	1	2	Go							
What		2	2	Go	Slow						

Order of Growth

	Best	**Worst**	**Amortized**
Initialization	1	1	1
Push	1	N	1
Pop	1	N	1

Stack ADT

Based on the above discussion, the Stack ADT should be implemented as follows:

In tasks given to you in which you need to use stack: you will assume that copy ctor and assignment optr are not available in the ADT unless specified otherwise.

```
#include <iostream>
using namespace std;

template<typename T>
class Stack
{
private:
    T * data;
    int capacity;
    int top;
    void reSize(int);
public:
    Stack();
    Stack(const Stack<T> &);
    Stack & operator = (const Stack<T> & );
    ~Stack();

    void push(T);
    T pop();
```

```
template<typename T>
T Stack<T>::stackTop()
{
    if (isEmpty())
        throw exception();
    return data[top-1];
}
template<typename T>
int Stack<T>::getCapacity()
{
    return capacity;
}
template<typename T>
int Stack<T>::getNumberOfElements()
{
    return top;
}
template<typename T>
Stack<T>::Stack(const Stack<T> & ref)
```

```
    T stackTop();
    bool isFull();
    bool isEmpty();
    int getCapacity();
    int getNumberOfElements();
};
template<typename T>
Stack<T>::Stack()
{
    capacity = 0;
    data = nullptr;
    top = 0;
}
template<typename T>
void Stack<T>::push(T val)
{
    if (isFull())
        reSize(capacity==0?1:capacity*2);
    data[top++] = val;
}
template<typename T>
T Stack<T>::pop()
{
    if (isEmpty())
        throw exception();

    T val = data[--top];
    if (top > 0 && top == capacity/4)
        reSize(capacity/2);
    return val;
}
template<typename T>
void Stack<T>::reSize(int newSize)
{
    T * temp = new T[newSize];
    for(int i=0; i<top; i++)
        temp[i] = data[i];
    this->~Stack();
    data = temp;
    capacity = newSize;
}
{
    //do it yourself
}
template<typename T>
Stack<T> & Stack<T>::operator=(const Stack<T> &
ref)
{
    //do it yourself
}
template<typename T>
Stack<T>::~Stack<T>()
{
    if(!data)
        return;
    delete [] data;
    data = nullptr;
}
template<typename T>
bool Stack<T>::isEmpty()
{
    return top==0;
}
template<typename T>
bool Stack<T>::isFull()
{
    return top==capacity;
}
int main()
{
    Stack<int> s;
    for (int i=1; i<=12; i++)
    {
        s.push(i);
    }
    while(!(s.isEmpty()))
    {
        cout<<endl<<s.pop();
    }
    return 0;
}
```

And Yes! go through STL
http://www.cplusplus.com/reference/stack/stack/
http://www.cplusplus.com/reference/array/array/
http://www.cplusplus.com/reference/bitset/bitset/

In Next Session: we shall explore two applications of stack i.e. Maze Problem and expression evaluation.
For any queries related to current session/concepts you may ping me on piazza.