COM 201 – Data Structures and Algorithms

Abstract Data Types - Stack

Assist. Prof. Özge ÖZTİMUR KARADAĞ

Department of Computer Engineering – ALKÜ

Alanya

Previously

- Linear Data Structures
  - Arrays
  - Linked Lists

  - Allowe one to insert and delete elements at any place in the list.

Next

- Need data structures which restrict insertions and deletions so that they can take place only at the beginning or in the end of the list, not in the middle.
- Abstract Data Types
- Stacks
- Queues

Abstract Data Type (ADT)

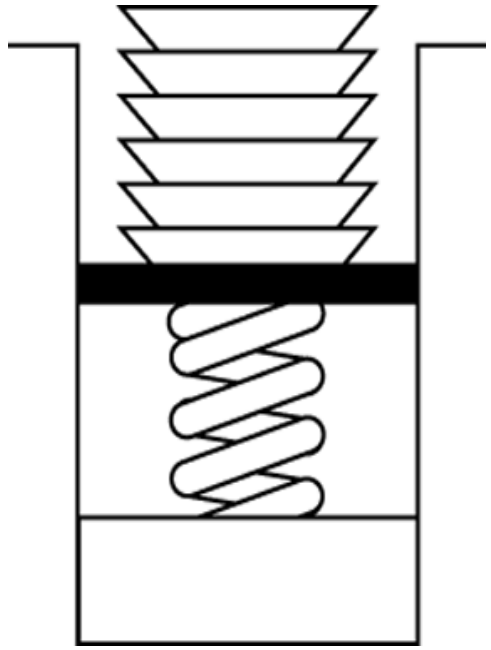
- Mathematical model for data types
- ADT is defined by its behaviour (from the point of view of a user):
  - Possible values,
  - Possible operations and behaviour of these operations

- Implementer → interested in the data structure
- User → interested in the ADT

The Stack ADT

The Stack ADT

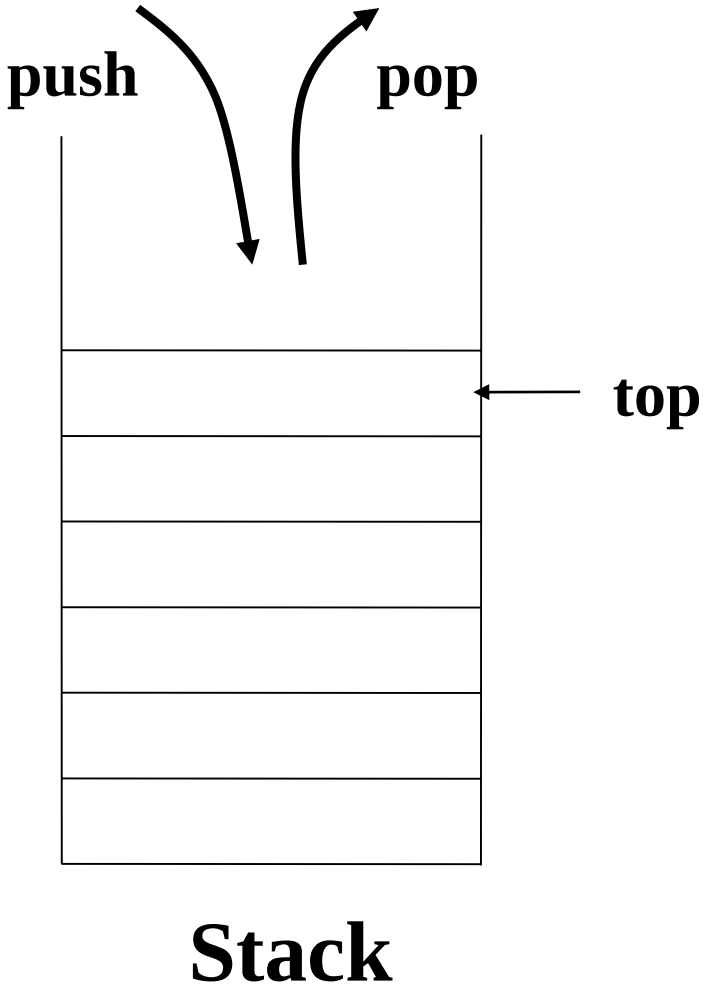
- The Stack ADT stores arbitrary objects.
- Insertions and deletions follow the *last-in first-out* (LIFO) scheme.
    - The last item placed on the stack will be the first item removed.
      (similar to a stack of dishes)



Stack of Dishes

Stack Operations

- Create an empty stack
- Destroy a stack
- Determine whether a stack is empty
- Add a new item -- **push**
- Remove the item that was added most recently -- **pop**
- Retrieve the item that was added most recently



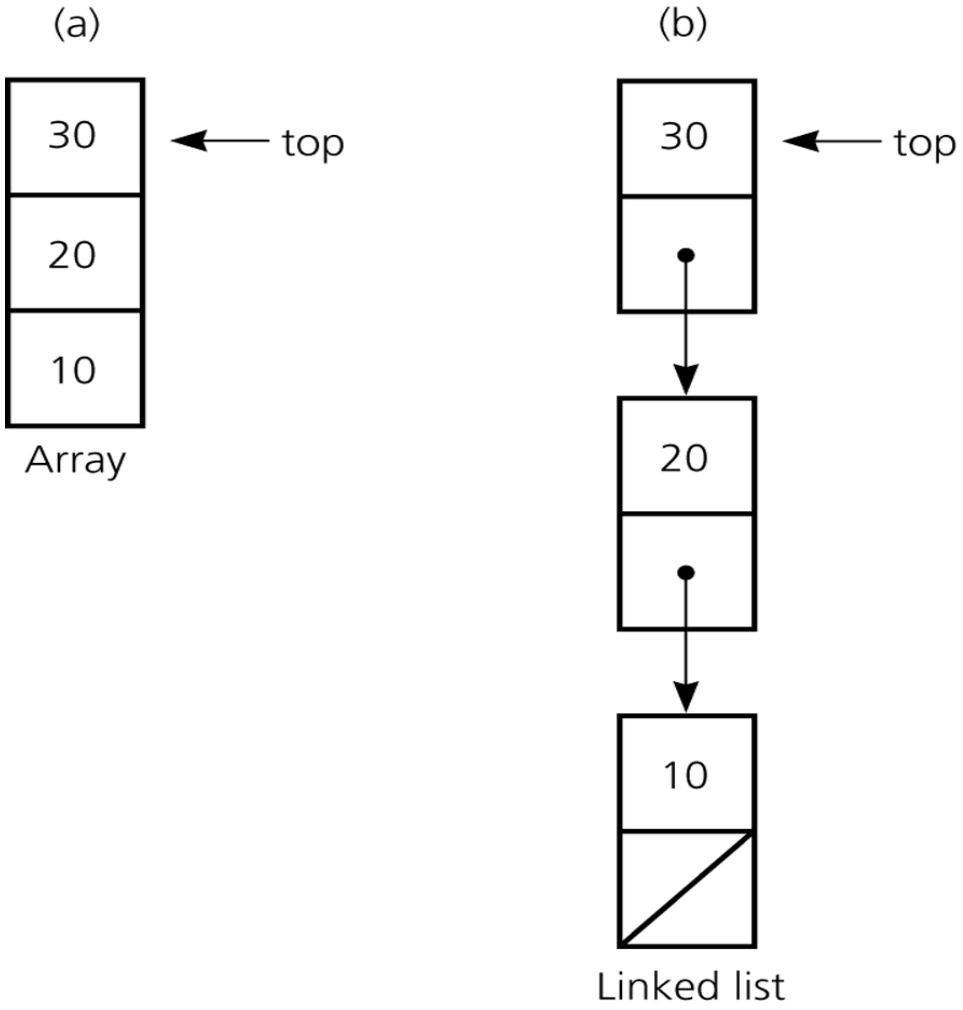
Stack Operations

- **Stack**()
  - creates a an empty stack
- **~Stack**()
  - destroys a stack
- **isEmpty**():boolean
  - determines whether a stack is empty or not
- **push**(in newItem:StackItemType)
  - Adds newItem to the top of a stack
- **pop**() throw StackException
- **topAndPop**(out stackTop:StackItemType)
  - Removes the top of a stack (ie. removes the item that was added most recently
- **getTop**(out stackTop:StackItemType)
  - Retrieves the top of stack into stackTop

Implementation of the Stack ADT

- Stack can be implemented using;
  - An array
  - A linked list

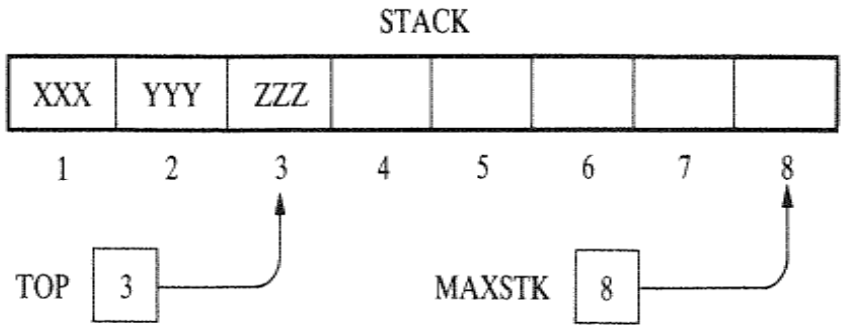
Implementation of Stack



An Array Based Implementation of Stack

- A pointer variable TOP (contains the location of the top element of the stack)
- A variable MAXSTK which gives the maximum number of elements that can be held by the stack.

- Ex: A stack with three elements with size 8



An Array Based Implementation of Stack

- Push

PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack.

1. [Stack already filled?]
   If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP := TOP + 1. [Increases TOP by 1.]
3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]
4. Return.

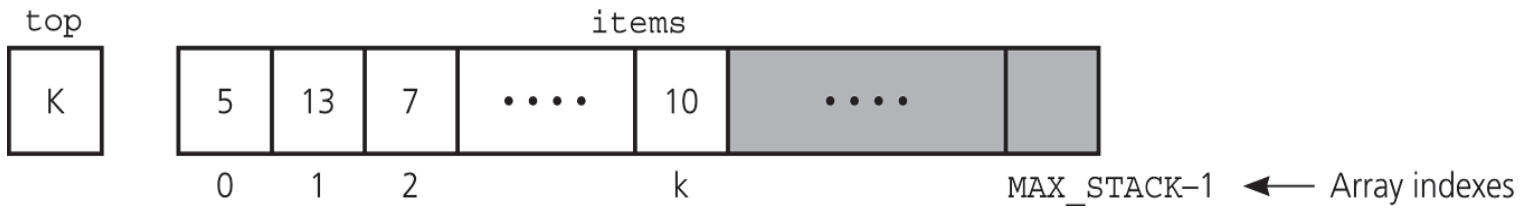
- Pop

POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]
   If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP := TOP − 1. [Decreases TOP by 1.]
4. Return.

An Array Based Implementation of Stack

- Private data fields
  - An array of `items` of type `StackItemType`
  - The index `top`
- Compiler-generated destructor, copy constructor, and assignment operator

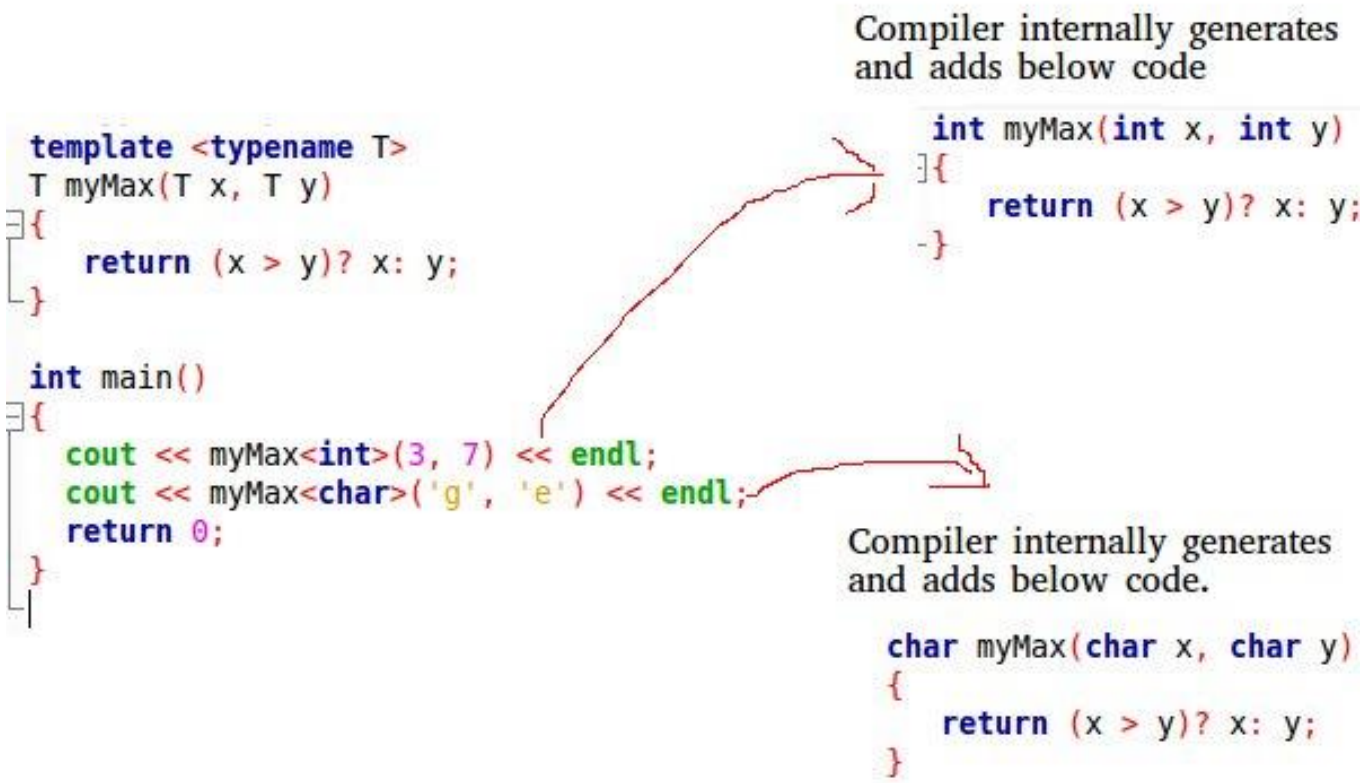


Programming Languages Concepts Used

- Templates
- Special Class Member Functions
  - Constructor
    - Deafult constructor
    - Copy constructor
  - Destructor
  - Assignment operator
- Pass by value vs. pass by reference

Templates in C++

- Pass data type as parameter, no need to write same code for different data types.
- Ex: Template function



Pass by value vs. pass by reference

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

void my_function(int x) {
   x = 50;
   cout << "Value of x from my_function: " << x << endl;
}

main() {
   int x = 10;
   my_function(x);
   cout << "Value of x from main function: " << x;
}
```

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

void my_function(int &x) {
   x = 50;
   cout << "Value of x from my_function: " << x << endl;
}

main() {
   int x = 10;
   my_function(x);
   cout << "Value of x from main function: " << x;
}
```

- Output:

```
Value of x from my_function: 50
Value of x from main function: 10
```

```
Value of x from my_function: 50
Value of x from main function: 50
```

An Array Based Implementation – Header File

```
#include "StackException.h"
const int MAX_STACK = maximum-size-of-stack;
template <class T>
class Stack {
public:
  Stack();  // default constructor; copy constructor and destructor are supplied by the compiler

  // stack operations:
  bool isEmpty() const;                 // Determines whether a stack is empty.
  void push(const T& newItem);          // Adds an item to the top of a stack.
  void pop();                           // Removes the top of a stack.
  void topAndPop(T& stackTop);
  void getTop(T& stackTop) const;   // Retrieves top of stack.

private:
  T items[MAX_STACK];          // array of stack items
  int top;                     // index to top of stack
};
```

Defined as constant function, hence is not allowed to change the values of the data members of its class.

An Array Based Implementation - push

```
template <class T>
void Stack<T>::push(const T& newItem) {


  if (top >= MAX_STACK-1)
    throw StackException("StackException: stack full on push");
  else
    items[++top] = newItem;
}
```

An Array Based Implementation – top, isEmpty

```
template <class T>
Stack<T>::Stack(): top(-1) {}       // default constructor


template <class T>
bool Stack<T>::isEmpty() const {
   return top < 0;
}
```

An Array Based Implementation – pop

```
template <class T>
void Stack<T>::pop() {

   if (isEmpty())
      throw StackException("StackException: stack empty on pop");
   else
      --top;     // stack is not empty; pop top
}
```

An Array Based Implementation – topAndPop

```
template <class T>
void Stack<T>::topAndPop(T& stackTop) {

  if (isEmpty())
    throw StackException("StackException: stack empty on pop");
  else // stack is not empty; retrieve top
    stackTop = items[top--];
}
```

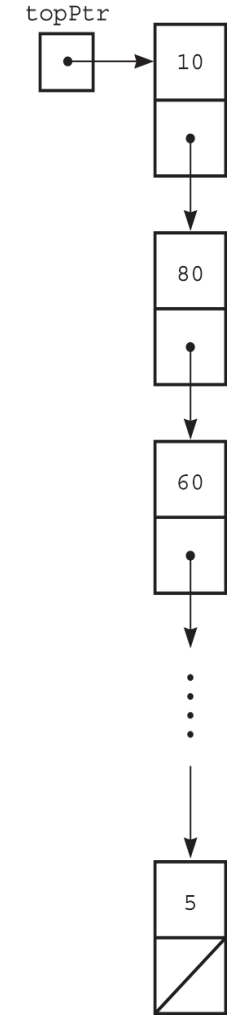
An Array Based Implementation – getTop

```
template <class T>
void Stack<T>::getTop(T& stackTop) const  {
   if (isEmpty())
      throw StackException("StackException: stack empty on getTop");
   else
       stackTop = items[top];
}
```

An Array Based Implementation

- Disadvantages of the array based implementation is similar to the disadvantages of arrays
  - It forces all stack objects to have MAX_STACK elements

A Pointer Based Implementation of Stack



- A pointer-based implementation
  - Required when the stack needs to grow and shrink dynamically
  - Very similar to linked lists
- `top` is a reference to the head of a linked list of items
- A copy constructor, assignment operator, and destructor must be supplied

A Pointer Based Implementation of Stack

```
template <class Object>
class StackNode
{
   public:
      StackNode(const Object& e = Object(), StackNode* n = NULL)
         : element(e), next(n) {}

      Object item;
      StackNode* next;
};
```

A Pointer Based Implementation of Stack

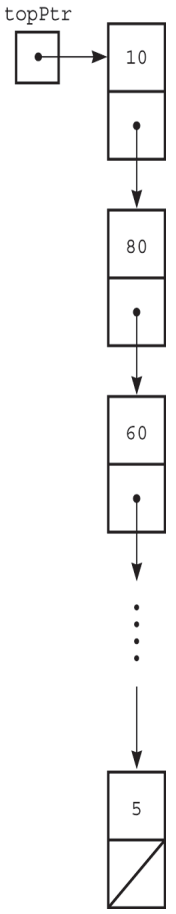
```
#include "StackException.h"
template <class T>
class Stack{
public:
  Stack();                                          // default constructor
  Stack(const Stack& rhs);                          // copy constructor
  ~Stack();                                         // destructor
  Stack& operator=(const Stack& rhs);   // assignment operator
  bool isEmpty() const;
  void push(const T& newItem);
  void pop();
  void topAndPop(T& stackTop);
  void getTop(T& stackTop) const;
private:
•    StackNode<T> *topPtr;              // pointer to the first node in the stack
};
```

A Pointer Based Implementation of Stack – Constructor, isEmpty

```
template <class T>
Stack<T>::Stack() : topPtr(NULL) {}      // default constructor

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

A Pointer Based Implementation of Stack – push

```
template <class T>
void Stack<T>::push(const T& newItem) {
  // create a new node
  StackNode *newPtr = new StackNode;

  newPtr->item = newItem; // insert the data


  newPtr->next = topPtr; // link this node to the stack
  topPtr = newPtr;              // update the stack top
}
```



A Pointer Based Implementation of Stack – pop

```
template <class T>
void Stack<T>::pop() {
  if (isEmpty())
    throw StackException("StackException: stack empty on pop");
  else {
    StackNode<T> *tmp = topPtr;
    topPtr = topPtr->next; // update the stack top
    delete tmp;
  }
}
```

A Pointer Based Implementation of Stack – topAndPop

```
template <class T>
void Stack<T>::topAndPop(T& stackTop) {
  if (isEmpty())
    throw StackException("StackException: stack empty on  topAndPop");
  else {
    stackTop = topPtr->item;
    StackNode<T> *tmp = topPtr;
    topPtr = topPtr->next; // update the stack top
    delete tmp;
  }
}
```

A Pointer Based Implementation of Stack – getTop

```
template <class T>
void Stack<T>::getTop(T& stackTop) const  {
   if (isEmpty())
      throw StackException("StackException: stack empty on getTop");
   else
      stackTop = topPtr->item;
}
```

A Pointer Based Implementation of Stack - destructor

```
template <class T>
Stack<T>::~Stack() {
  // pop until stack is empty
  while (!isEmpty())
      pop();
}
```

A Pointer-Based Implementation – assignment

```
template <class T>
Stack<T>& Stack<T>::operator=(const Stack& rhs) {
   if (this != &rhs) {
      if (!rhs.topPtr)
         topPtr = NULL;
      else {
         topPtr = new StackNode<T>;
         topPtr->item = rhs.topPtr->item;
         StackNode<T>* q = rhs.topPtr->next;
         StackNode<T>* p = topPtr;
         while (q) {
            p->next = new StackNode<T>;
            p->next->item = q->item;
            p = p->next;
            q = q->next;
         }
         p->next = NULL;
      }
   }
   return *this;
}
```

A Pointer-Based Implementation – copy constructor

```
template <class T>
Stack<T>::Stack(const Stack& rhs) {
    *this = rhs; // reuse assignment operator
}
```

Testing the Stack Class

```
int main() {
    Stack<int> s;
    for (int i = 0; i < 10; i++)
        s.push(i);

    Stack<int> s2 = s; // test copy constructor (also tests assignment)

    std::cout << "Printing s:" << std::endl;
    while (!s.isEmpty()) {
        int value;
        s.topAndPop(value);
        std::cout << value << std::endl;
    }
```

Testing the Stack Class

```
    std::cout << "Printing s2:" << std::endl;
    while (!s2.isEmpty()) {
        int value;
        s2.topAndPop(value);
        std::cout << value << std::endl;
    }

    return 0;
}
```

References

- Lecture Notes, Yusuf Sahillioğlu, METU.
- Schaum’s Outline of Theory and Problems of Data Structures, Seymour Lipschutz.