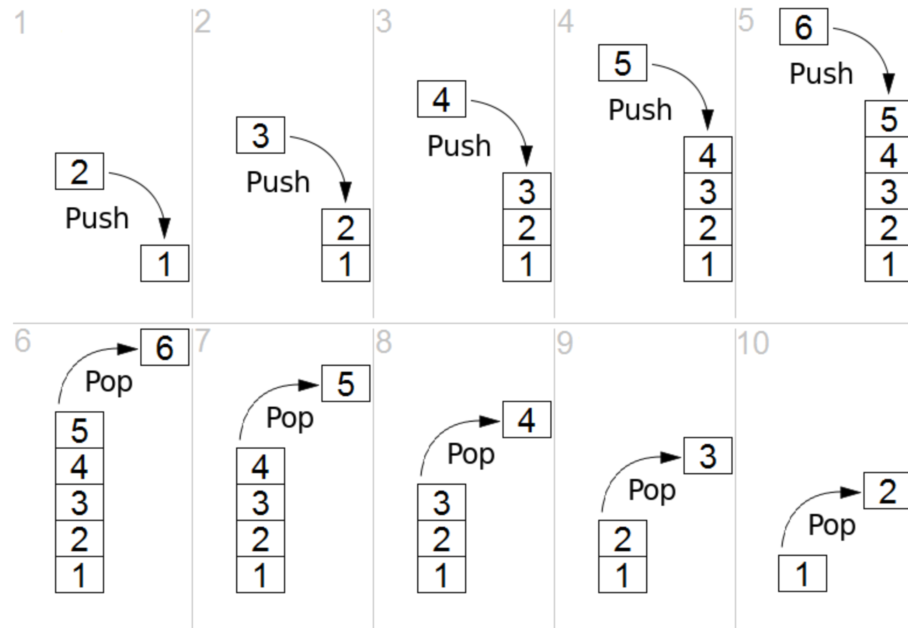
Section 6

Abstract Data Types: Stacks and Queues

Presentation by *Asem Alaa*

Stack



- Abstract data type
- **LIFO** (last in, first out)
- Defines a set of supported operations
- Many implementation options (e.g Array or Linked List)

Stack

Essential Operations

- **push**
- **pop**

Non-Essential Operations

- **front**
- **empty**

Stack Demo

{StackArray}

Stack Applications

- Algorithms
- Call stack (stack frame). See {demo}.

Providing a Stack implementation

```
template< typename T >
class Stack
{
public:
    bool isEmpty() const  { /** logic **/}
    T front() const  { /** logic **/}
    void pop()  { /** logic **/}
    void push( T value ) { /** logic **/}
private:
// Implementation specifics go here.
};
```

Hide Implementation Details From Client

- To comply with the definition.
- To avoid client abusing the ADT.

Image credits: robotmissions.org

Stack Implamentation (using Array)

- Problem: static array size must be know for the compiler.

```
template< typename T >
class Stack
{
public:
    bool isEmpty() const  {}
    T front() const  {}
    void pop() {}
    void push( T value ) {}
private:
    T data[1000];  // Like this
};
```

Issues:

1. Magic numbers need to be avoided.
2. Cannot make stacks with different capacities (**in the same program**)

Stack Implamentation (using Array)

Solution 1 (C style)

```
#define MAX_SIZE 1000
template< typename T >
class Stack
{
public:
    bool isEmpty() const  {}
    T front() const  {}
    void pop() {}
    void push( T value ) {}
private:
    T data[MAX_SIZE];
};
```

1. Solved magic numbers
2. Still inflexible
3. +++Compiler variables in the global scope

Stack Implamentation (using Array)

Solution 2 (C++ style)

```cpp
template< typename T , int MAX_SIZE>
class Stack
{
public:
    bool isEmpty() const  {}
    T front() const  {}
    void pop() {}
    void push( T value ) {}
private:
    T data[MAX_SIZE];
};
```

- `MAX_SIZE` constant is only seen inside the `Stack` template class.
- You can make different stacks with different capacities

```cpp
Stack< int , 2000 > s1; // stack of integers with maximum capacity 2000
Stack< double, 500> s2; // stack of doubles with maximum capacity of 500
```

Stack Implamentation (using Array)

Solution 2 (C++ style) + Default Values

```
template< typename T , int MAX_SIZE = 1024>
class Stack
{
public:
    bool isEmpty() const  {}
    T front() const  {}
    void pop() {}
    void push( T value ) {}
private:
    T data[MAX_SIZE];
};
```

```
// stack of chars with maximum capacity 1024
Stack< char > s1;
// stack of std::string with maximum capacity of 500
Stack< std::string, 500> s2;
```

Final Implementation for Stack Array

```
template< typename T , int MAX_SIZE = 1000 >
class StackArray
{
public:
    bool isEmpty() const { return top == -1; }

    T front() const {
        if( isEmpty()) exit( 1 ); // Crash.
        return data[top];
    }

    void pop() {
        if( isEmpty()) exit( 1 ); // Crash.
        --top;
    }

    void push( T value ){
        if( isFull()) exit( 1 ); // Crash.
        data[ ++top ] = value;
    }
private:
    bool isFull() const { return top + 1 == MAX_SIZE; }
    T data[ MAX_SIZE ];
    int top = -1;
};
```

Singly-linked list vs. doubly-linked list

- Stack pushes and pops **from the same side**
- **Option 1**: push and pop from the back side.
  - $O(1)$ for doubly-ll, while $O(n)$ for singly-ll
- **Option 2**: push and pop from the front side.
  - $O(1)$ for both doubly-ll & singly-ll.
- Singly-ll is more space efficient.
- Can provid $O(1)$ time for the stack operations.
- C++ STL has singly-ll via `std::forward_list`.

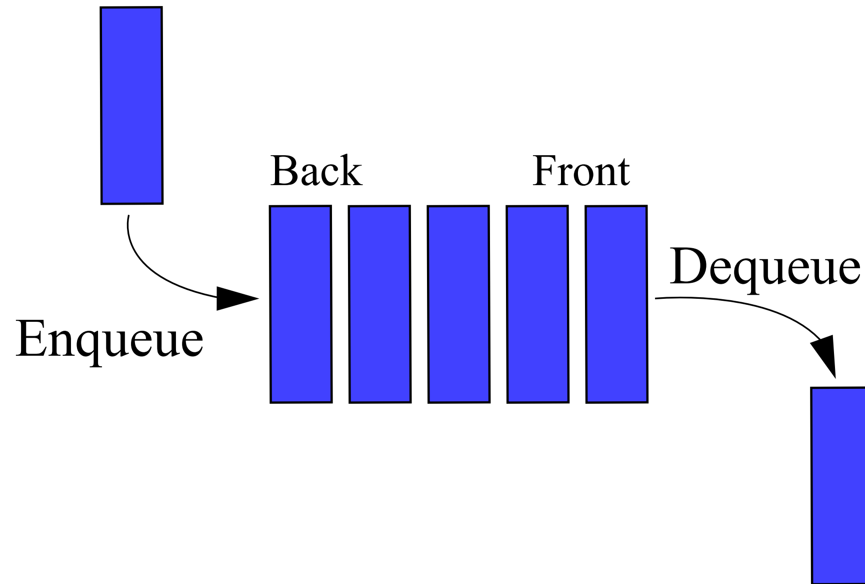
Final Implementation for Stack SLL

```
#include <forward_list>
template< typename T >
class StackSLL
{
public:
    bool isEmpty() const{
        return data.empty();
    }

    T front() const{
        if( isEmpty()) exit( 1 ); // Crash.
        return data.front();
    }

    void pop(){
        if( isEmpty()) exit( 1 ); // Crash.
        data.pop_front();
    }
    void push( T value ){
        data.push_front( value );
    }
private:
    std::forward_list< T > data;
};
```

Queue



- Abstract Data Type (ADT),
- Many implementations (e.g arrays and linked lists)
- **Queue** behaviour == **FIFO** (first in, first out).

Queue

Essential Operations

- **enqueue**: which adds an element to the collection end (back)
- **dequeue**: which removes the first element added (front) that was not yet removed.

Non-Essential Operations

- **front**: which returns the earliest element added to the queue that was not yet removed.
- **empty**: returns whether the queue is empty or not, to avoid dequeueing from empty queue.

Queue

Queue Demo

- {QueueArray}: implementation by **circular arrays**.

Queue Applications

- Algorithms
- For network multiplayer games
- Realtime signal processing
- Multithreaded/Parallel Processing applications.

Queue

Providing a Queue implementation

```cpp
template< typename T >
class Queue
{
public:
    bool isEmpty() const;
    T front() const;
    void dequeue();
    void enqueue( T value );
};
```

Queue Implementation (using circular Array)

```
template< typename T , int MAX_SIZE = 1000 >
class QueueArray
{
public:
    bool isEmpty() const{ return front_ == rear_; }
    T front() const{
        if( isEmpty()) exit( 1 ); // Crash.
        return data_[front_];
    }
    void dequeue(){
        if( isEmpty()) exit( 1 ); // Crash.
        front_ = (front_ + 1) % MAX_SIZE;
    }
    void enqueue( T value ){
        if( isFull()) exit( 1 ); // Crash.
        data_[ rear_ ] = value;
        rear_ = (rear_+1) % MAX_SIZE;
    }
private:
    bool isFull() const{ return (rear_ + 1) % MAX_SIZE == front_; }

    T data_[ MAX_SIZE ];
    int front_ = 0;
    int rear_ = 0;
};
```

Queue

Queue Implementation (using Doubly-LL)

singly-linked list vs. doubly-linked list

- Queues enqueues and dequeues element **from different sides**.
- We either:
  - **push front and pop back** for the enq. & deq.
  - **push back and remove front** for the enq. and deq.
- Only doubly-ll list can afford fast modification from both sides.

Queue Implementation (using Doubly-LL)

```
#include <list>
template< typename T >
class QueueDLL
{
public:
    bool isEmpty() const {
        return data_.empty();
    }
    T front() const
    {
        if( data_.empty()) exit( 1 ); // Crash.
        return data_.front();
    }
    void dequeue()
    {
        if( isEmpty()) exit( 1 ); // Crash.
        data_.pop_front();
    }
    void enqueue( T value ){
        data_.push_back( value );
    }
private:
    std::list< T > data_;
};
```

Read the Notes

Read the notes for more details and to download the source files.

{sbme-tutorials.github.io/2020/data-structures/notes/week06_adt.html}

Thank you