

---

## Stacks and Queues

What is a stack/queue?

What are stacks/queues used for?

Which properties do stacks/queues have?

How can they be implemented?

(Abstract Data Types)

---

# Linear Data Structures

LINEAR: Collection of components arranged in a straight line.

Restrict places where one can add/remove.

- Add/remove only at one end:  
⇒ **Stack** (LIFO list)
- Add at one end, remove from the other:  
⇒ **Queue** (FIFO list)

---

# What are stacks/queues used for?

## Stacks:

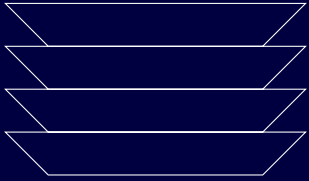
- processing nested structures
  - implementing parsing, evaluation and backtracking algorithms
- ⇒ function calls containing other function calls, e.g. implementation of function calls and returns during execution of C program

## Queues:

- regulating the processing of tasks in a system to ensure “fair” treatment
- ⇒ client-server systems: Clients line up to wait for service
- ⇒ operating systems: Queues of tasks, e.g. print queue
- ⇒ simulation and modelling: Performance analysis of systems under various loads, e.g. air traffic control, urban transport etc.

---

# Background on stacks

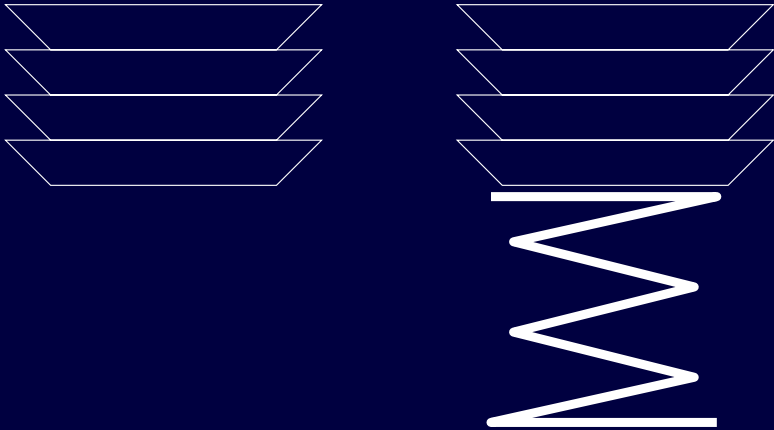


Pile of objects of any kind.

- Add/Remove objects only at the top of the pile.

---

# Background on stacks



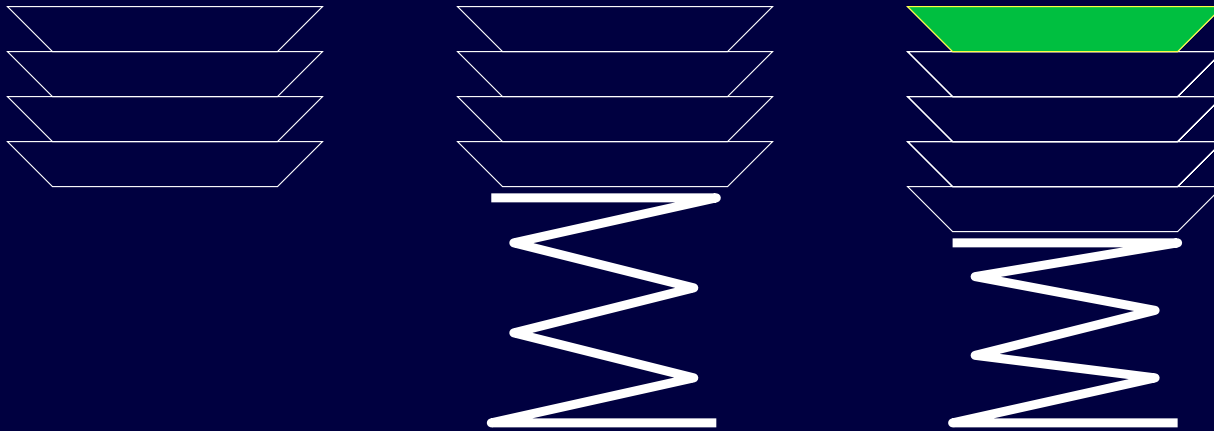
## Spring-loaded pile.

- Add/Remove objects only at the top of the pile.

= Only the top of the pile is accessible.

- The top remains in a fixed position.

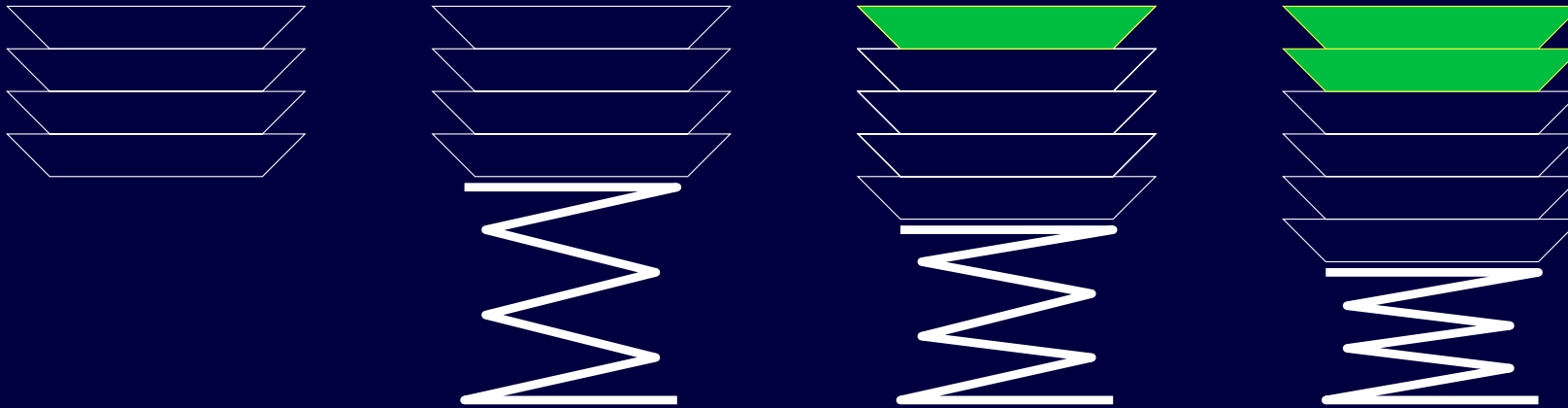
# Background on stacks



## Push-down stack.

- Adding a new object to the top: **pushing** it onto the stack.

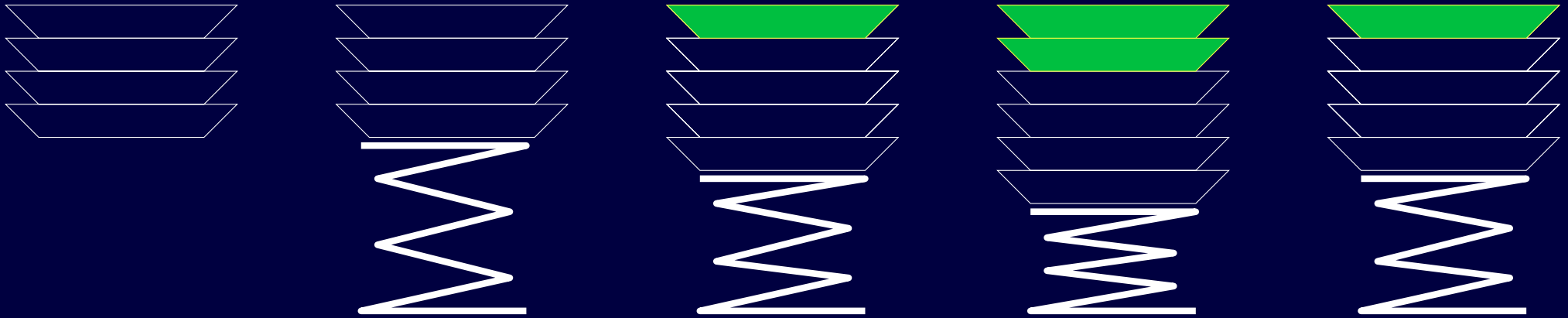
# Background on stacks



## Push-down stack.

- Adding a new object to the top: **pushing** it onto the stack.

# Background on stacks



## Push-down stack.

- Adding a new object to the top: **pushing** it onto the stack.
- Removing an object from the top: **popping** it from the stack.

= Pushing and popping are **inverse operations**:

- Push object and immediately afterwards pop it.
- Original stack remains unchanged.

---

# Stack representation

We imagine that the entire stack:

- moves **down** when a new item is **pushed** onto it, and
- moves **up** when the topmost item is **popped**.

Underlying **data representation** for stacks **does not behave like this.**

(Why not?)

- Use array-based implementation:  
*Growth end of stack travels through memory, and items in the stack remain fixed in place.*

---

# Abstract Design: Data Structures

## Abstract Data Type (ADT):

- collection of **data structures**, together with a
- set of operations defined on those data structures

## Data structures:

- usually composite
- apply a *structuring method* to a collection of (simple) components
- Formation of structs, arrays, strings etc.
- Linked data structures obtained by using pointers to components and embedding pointers inside data structures.

---

# Abstract Design: Operations

## Abstract Data Type (ADT):

- collection of data structures, together with a
- set of **operations** defined on those data structures.

Suppose we have fixed a structuring method:

- Formation of linear sequences of components.
- ⇒ Basis for strings, arrays, lists, stacks and queues.

What makes a sequence a stack or a queue?

- The set of operations; it is *different* for each.
- Stacks and queues can be defined by specifying two separate sets of operations on sequences.

---

# Stack ADT

“State *abstractly* what the operations do, without saying in detail *how* they are accomplished.”

A stack  $S$ , of items of type  $T$  is a sequence of items of type  $T$  on which the following operations are defined:

1. Initialize the stack  $S$  to be the *empty* stack.
2. Determine whether or not the stack,  $S$ , is *empty*.
3. Determine whether or not the stack,  $S$ , is *full*.
4. *Push* a new item onto the top of the stack,  $S$ .
5. If  $S$  is not empty, *pop* an item from the top of the stack,  $S$ .

---

# Queue ADT

“State *abstractly* what the operations do, without saying in detail *how* they are accomplished.”

A queue  $Q$ , of items of type  $T$  is a sequence of items of type  $T$  on which the following operations are defined:

1. Initialize the queue  $Q$  to be the *empty* queue.
2. Determine whether or not the queue,  $Q$ , is *empty*.
3. Determine whether or not the queue,  $Q$ , is *full*.
4. *Insert* a new item onto the rear of the queue,  $Q$ .
5. Provided  $Q$  is not empty, *remove* an item from the front of  $Q$ .

---

# Abstract Design: Summary

## Abstract Data Type (ADT):

- collection of **data structures**, together with a
- set of **operations** defined on those data structures.

ADT **deliberately** does not mention what underlying data representation is required. ( $\Rightarrow$  That's what we **abstract** from.)

Sequences on ADT level can be implemented using:

- linked representations
    - structs hold individual items; link structs in a linear linked list
  - sequential representations
    - use an array
- (• sets of ordered pairs)

---

# Header Files in C

Files with `.h` postfix such as `stdio.h`

- **Primary use of header files:** They (can) contain
  - type declarations and constant definitions, eg:  
`#define PI 3.14`
  - a list of **function prototypes**.
- **Function prototype** tells compiler (and user):
  - types of arguments that get passed to function, and
  - type of value that gets returned by function.

There is no restriction on what an include file can contain.

`#include "filename"`

- Preprocessor replaces this line with copy of contents of *filename*.
- Search for *filename* first in current, then in system directories.

`#include <filename>`

- Search in the system-dependent places only.

---

# Map of C file architecture

Typically written as a collection of `.c` and `.h` files.

- Each `.c` file contains one or more function definitions.
- Each `.h` file contains one or more function prototypes.

Header files are included at the top of the `.c` files.

They act as the **glue** that binds the program together.

Users don't need to know the implementation; it is **hidden**.

---

# Stack ADT interface

```
/* File: Stack.h */
typedef char ItemType ;
typedef struct Stack Stack ;

/* Defined Operations */

extern Stack *StackNew( void );
    /* Initialize the stack S to be the empty stack. */
extern int StackEmpty ( Stack *S );
    /* Returns 1 (True) if (and only if) the stack S is empty*/
extern int StackFull ( Stack *S );
    /* Returns 1 (True) if (and only if) the stack S is full */
extern void StackPush ( ItemType X, Stack *S );
    /* If S is not full, push a new item X onto the top of S */
extern void StackPop ( Stack *S, ItemType *X );
    /* If S is not empty, pop an item off the top of S and put
       it in X. */
```

---

# Queue ADT interface

```
/* File: Queue.h */
typedef char ItemType ;
typedef struct Queue Queue ;
/* Defined Operations */

extern Queue *QueueNew ( void );
    /* Initialize the queue Q to be the empty queue. */

extern int QueueEmpty ( Queue *Q );
    /* Returns 1 (True) if (and only if) the queue Q is empty*/
extern int QueueFull ( Queue *Q );
    /* Returns 1 (True) if (and only if) the queue Q is full */

extern void QueueInsert ( ItemType R, Queue *Q );
    /* If Q is not full, insert new item R onto the end of Q */
extern void QueueRemove ( Queue *Q, ItemType *F );
    /* If Q is not empty, remove its first item, put it in F. */
```

---

## Stack Implementation

Two data structures which can represent a stack.  
How to implement the stack operations.

---

# Using the Stack interface

```
#include "Stack.h"    /* Assume ItemType is char. */
                    /* Access operations and types for stacks.*/
int main ( void )
{
    Stack *ts;      /* Variable ts of type Stack. */
    char    C;

    ts = StackNew( ); /* Make ts empty. */
    StackPush ( 'a', ts );
    StackPush ( 'b', ts );
    StackPush ( 'c', ts );
    StackPop  ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPop  ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPush ( 'd', ts );
    ...
}
```

= Write this code **without knowing how** the stack is implemented.

# Using the Stack interface

```
#include "Stack.h"    /* Assume ItemType is char. */
                    /* Access operations and types for stacks.*/
int main ( void )
{
    Stack *ts;      /* Variable ts of type Stack. */
    char    C;

⇒ ts = StackNew( ); /* Make ts empty. */
    StackPush ( 'a', ts );
    StackPush ( 'b', ts );
    StackPush ( 'c', ts );
    StackPop  ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPop  ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPush ( 'd', ts );
    ...
}
```

Stack: **Empty**

# Using the Stack interface

```
#include "Stack.h"    /* Assume ItemType is char. */
                    /* Access operations and types for stacks.*/
int main ( void )
{
    Stack *ts;      /* Variable ts of type Stack. */
    char    C;

    ts = StackNew( ); /* Make ts empty. */
⇒ StackPush ( 'a', ts );
   StackPush ( 'b', ts );
   StackPush ( 'c', ts );
   StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
   StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
   StackPush ( 'd', ts );
    ...
}
```

Stack: **a**

# Using the Stack interface

```
#include "Stack.h"    /* Assume ItemType is char. */
                    /* Access operations and types for stacks.*/
int main ( void )
{
    Stack *ts;      /* Variable ts of type Stack. */
    char    C;

    ts = StackNew( ); /* Make ts empty. */
    StackPush ( 'a', ts );
    ⇒ StackPush ( 'b', ts );
    StackPush ( 'c', ts );
    StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPush ( 'd', ts );
    ...
}
```

Stack: **b, a**

# Using the Stack interface

```
#include "Stack.h"    /* Assume ItemType is char. */
                    /* Access operations and types for stacks.*/
int main ( void )
{
    Stack *ts;      /* Variable ts of type Stack. */
    char    C;

    ts = StackNew( ); /* Make ts empty. */
    StackPush ( 'a', ts );
    StackPush ( 'b', ts );
    ⇒ StackPush ( 'c', ts );
    StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPush ( 'd', ts );
    ...
}
```

Stack: **c, b, a**

# Using the Stack interface

```
#include "Stack.h"    /* Assume ItemType is char. */
                    /* Access operations and types for stacks.*/
int main ( void )
{
    Stack *ts;      /* Variable ts of type Stack. */
    char    C;

    ts = StackNew( ); /* Make ts empty. */
    StackPush ( 'a', ts );
    StackPush ( 'b', ts );
    StackPush ( 'c', ts );
    ⇒ StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPush ( 'd', ts );
    ...
}
```

Stack: **b, a**      Print: **Just popped: c.**

# Using the Stack interface

```
#include "Stack.h"    /* Assume ItemType is char. */
                    /* Access operations and types for stacks.*/
int main ( void )
{
    Stack *ts;      /* Variable ts of type Stack. */
    char  C;

    ts = StackNew( ); /* Make ts empty. */
    StackPush ( 'a', ts );
    StackPush ( 'b', ts );
    StackPush ( 'c', ts );
    StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
⇒ StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPush ( 'd', ts );
    ...
}
```

Stack: **a**      Print: **Just popped: b.**

# Using the Stack interface

```
#include "Stack.h"    /* Assume ItemType is char. */
                    /* Access operations and types for stacks.*/
int main ( void )
{
    Stack *ts;      /* Variable ts of type Stack. */
    char    C;

    ts = StackNew( ); /* Make ts empty. */
    StackPush ( 'a', ts );
    StackPush ( 'b', ts );
    StackPush ( 'c', ts );
    StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
    StackPop ( ts, &C ); printf( "Popped: %c.\n", C );
⇒ StackPush ( 'd', ts );
    ...
}
```

Stack: **d, a**

---

# Sequential Stack Representation (I)

Use arrays of stack items to implement stacks with bounded capacity.

- Define `MAXSTACKSIZE` as a constant. Stack struct contains
  - number of items currently in stack as e.g. the `Count` member,
  - an array of stack items as e.g. its `Items` member.

```
#include <stdio.h>           /* File: Stack.c */
#include <stdlib.h>
#include "Stack.h"
#define MAXSTACKSIZE 100
struct Stack {
    int      Count;
    ItemType Items[MAXSTACKSIZE];
} ;
```

The file `stack.c` contains the operations `StackNew`, `StackEmpty`, `StackFull`, `StackPush` and `StackPop`, introduced below.

---

# Sequential Stack Representation (II)

## Initialization:

- Initialize the stack  $S$  to be the *empty* stack.
- A stack is defined to be **empty** if its `Count` member is zero.

```
/* Initialize the stack S to be the empty stack. */
Stack *StackNew ( void )
{
    Stack *S = calloc( 1, sizeof( Stack ) ) ;
    S->Count = 0;
    return S ;
}
```

Note: `S->Count`, not `S.Count` Why?

---

# Sequential Stack Representation (III)

**Empty test:** Determine whether or not the stack,  $S$ , is *empty*.

- A stack is defined to be **empty** if its `Count` member is zero.

```
/* Returns 1 (True) if (and only if)
 * the stack S is empty.
 */
```

```
int StackEmpty ( Stack *S )
{
    return ( S->Count == 0 );
}
```

---

# Sequential Stack Representation (IV)

**Full test:** Determine whether or not the stack,  $S$ , is *full*.

- A stack is defined to be **full** if `Count` is `MAXSTACKSIZE`.

```
/* Returns 1 (True) if (and only if)
 * the stack S is full.
 */

int StackFull ( Stack *S )
{
    return ( S->Count == MAXSTACKSIZE );
}
```

---

# Sequential Stack Representation (V)

Push a new item  $X$  onto the top of the stack,  $S$ .

```
/* If S is not full,  
 * push a new item X onto the top of S by:  
 * 1. storing X in the item array at position Count;  
 * 2. incrementing Count by one.  
 */  
  
void StackPush (ItemType X, Stack *S)  
{  
    if (S->Count == MAXSTACKSIZE) {  
        fprintf(stderr, "PUSH:Full stack.\n"); abort() ;  
    } else {  
        S->Items[S->Count] = X;  
        ++ (S->Count);  
    }  
}
```

---

# Sequential Stack Representation (V)

Push a new item  $X$  onto the top of the stack,  $S$ .

```
/* If S is not full,  
 * push a new item X onto the top of S by:  
 * 1. storing X in the item array at position Count;  
 * 2. incrementing Count by one.  
 */  
  
void StackPush (ItemType X, Stack *S)  
{  
    if (StackFull(S)) {  
        fprintf(stderr, "PUSH:Full stack.\n"); abort() ;  
    } else {  
        S->Items[S->Count] = X;  
        ++ (S->Count);  
    }  
}
```

# Prefix and Postfix Increment/Decrement

Increment operator `++`. Decrement operator `--`.

- Can be used in prefix or postfix position, *with different results*.

`++i` and `i++` both: Have a value. **AND**

⇒ Cause stored value of `i` to be incremented by 1. **Side effect!**

**`++i` increments first; value of expression is new stored value of `i`.**

**`i++` has current value of `i`; stored value of `i` is incremented.**

⇒ `int a, b, c = 0;`

`a = ++c;`

`b = c++;`

`printf("%d %d %d\n", a, b, ++c);`

What is printed?

- `++` and `--` change the value of a variable in memory!
- Binary operators `+` and `-` don't do this!

Sometimes can use `++` and `--` in either prefix or postfix position.

⇒ Statements `++i;` and `i++;` are both equivalent to `i = i+1;`

---

# Sequential Stack Representation (V)

Push a new item  $X$  onto the top of the stack,  $S$ .

```
/* If S is not full,  
 * push a new item X onto the top of S by:  
 * 1. storing X in the item array at position Count;  
 * 2. incrementing Count by one.  
 */  
  
void StackPush (ItemType X, Stack *S)  
{  
    if (StackFull(S)) {  
        fprintf(stderr, "PUSH:Full stack.\n"); abort() ;  
    } else {  
        S->Items[S->Count] = X;  
        ++(S->Count) ;  
    }  
}
```

---

# Sequential Stack Representation (V)

Push a new item  $X$  onto the top of the stack,  $S$ .

```
/* If S is not full,  
 * push a new item X onto the top of S by:  
 * 1. storing X in the item array at position Count;  
 * 2. incrementing Count by one.  
 */  
  
void StackPush (ItemType X, Stack *S)  
{  
    if (StackFull(S)) {  
        fprintf(stderr, "PUSH:Full stack.\n"); abort() ;  
    } else {  
        S->Items[S->Count] = X;  
        (S->Count)++;  
    }  
}
```

---

# Sequential Stack Representation (V)

Push a new item  $X$  onto the top of the stack,  $S$ .

```
/* If S is not full,  
 * push a new item X onto the top of S by:  
 * 1. storing X in the item array at position Count;  
 * 2. incrementing Count by one.  
 */  
  
void StackPush (ItemType X, Stack *S)  
{  
    if (StackFull(S)) {  
        fprintf(stderr, "PUSH:Full stack.\n"); abort() ;  
    } else {  
        S->Items[S->Count++] = X;  
    }  
}
```

---

# Sequential Stack Representation (VI)

**Pop** an item from the top of the stack,  $S$ , if  $S$  is not empty.

```
/* If S is not empty,  
 * pop an item off the top of S and put it in X by:  
 * 1. decrementing Count by one;  
 * 2. removing X from the item array at position Count.  
 */  
  
void StackPop ( Stack *S, ItemType *X )  
{  
    if ( S->Count == 0 ) {  
        fprintf(stderr, "POP:Empty stack.\n"); abort() ;  
    } else {  
        --(S->Count) ;  
        *X = S->Items[S->Count] ;  
    }  
}
```

---

# Sequential Stack Representation (VI)

**Pop** an item from the top of the stack,  $S$ , if  $S$  is not empty.

```
/* If S is not empty,  
 * pop an item off the top of S and put it in X by:  
 * 1. decrementing Count by one;  
 * 2. removing X from the item array at position Count.  
 */  
  
void StackPop ( Stack *S, ItemType *X )  
{  
    if ( S->Count == 0 ) {  
        fprintf(stderr, "POP:Empty stack.\n"); abort() ;  
    } else {  
        (S->Count)--;  
        *X = S->Items[S->Count];  
    }  
}
```

---

# Sequential Stack Representation (VI)

**Pop** an item from the top of the stack,  $S$ , if  $S$  is not empty.

```
/* If S is not empty,  
 * pop an item off the top of S and put it in X by:  
 * 1. decrementing Count by one;  
 * 2. removing X from the item array at position Count.  
 */  
  
void StackPop ( Stack *S, ItemType *X )  
{  
    if ( S->Count == 0 ) {  
        fprintf(stderr, "POP:Empty stack.\n"); abort() ;  
    } else {  
  
        *X = S->Items[ -- S->Count ] ;  
    }  
}
```

---

# Programs stored in multiple files

We now have: `stack.h` Supplied Interface.  
and `stackTest.c`, `stack.h`, `stack.c`.

## How to compile a C program which is stored in multiple files?

- Call the compiler several times:
  - Once for each module. Once to link the modules together.

⇒ 

```
gcc -c stack.c
gcc -c stackTest.c
gcc -o mystacktest stack.o stackTest.o
```

- `-c`: instructs compiler to compile only. Does not create final program.  
Creates a file called `stack.o`. (object file)
- `-o`: read object files, link them into executable `mystacktest`.

- Change one of the source files, only need to recompile that one, and link all modules together again.

---

# Intro to Shell scripts

Stored as text

```
emacs comp.sh
```

Written in shell scripting language

```
gcc -c Stack.c
```

```
gcc -c StackTest.c
```

```
gcc -o mystacktest Stack.o StackTest.o
```

Make executable

```
chmod +x comp.sh
```

Execute

```
comp.sh
```

---

# Linked Stack Representation (I)

What if we do not want to fix the maximum size of the stack?

- Represent a stack by a linked list of stack items.
  - The stack is a struct with an `ItemList` member.
  - Each node in the `ItemList` contains a stack item.

```
#include <stdio.h>                /* File: Stack.c */
#include <stdlib.h>
#include "Stack.h"
typedef struct StackNodeTag { /* alternative */
    ItemType      Item; /* definition */
    struct StackNodeTag *Link;
} StackNode;
struct {
    StackNode *ItemList;
} Stack;
```

The operations on this *List* stack are defined below

---

# Linked Stack Representation (II)

## Initialization:

- Initialize the stack  $S$  to be the *empty* stack.
- The empty list is represented by `NULL`.

```
/* Initialize the stack S to be the empty stack. */  
  
Stack *StackNew ( void )  
{  
    Stack *S = calloc( 1, sizeof( Stack ) ) ;  
    S->ItemList = NULL;  
    return S ;  
}
```

---

# Linked Stack Representation (III)

**Empty test:** Determine whether or not the stack,  $S$ , is *empty*.

- The empty list is represented by `NULL`.

```
/* Returns 1 (True) if (and only if)
 * the stack S is empty.
 */
```

```
int StackEmpty ( Stack *S )
{
    return ( S->ItemList == NULL );
}
```

---

# Linked Stack Representation (IV)

**Full test:** Determine whether or not the stack,  $S$ , is *full*.

- We assume that an already constructed stack,  $S$ , is not full. It could potentially grow as a linked structure.

```
/* Returns 1 (True) if (and only if)
 * the stack S is full.
 */

int StackFull ( Stack *S )
{
    return 0;
}
```

---

# Linked Stack Representation (V)

Push a new item  $X$  onto the top of the stack,  $S$ .

- First node in `ItemList` is top of stack.

```
/* Push a new item X onto the top of S. */
void StackPush (ItemType X, Stack *S)
{
    StackNode *Temp;
    /* Try to allocate a new stack node. */
    Temp = (StackNode *) calloc ( 1, sizeof ( StackNode ));

    if ( Temp == NULL ) {                /* Allocation failed.*/
        fprintf( stderr, "PUSH:Storage exhausted.\n");abort() ;
    } else {                             /* Allocation successful.*/
        Temp->Link = S->ItemList; /* Current stack into Link,*/
        Temp->Item = X;           /* new item into Item,*/
        S->ItemList = Temp;      /* makes Temp top stack node.*/
    }
}
```

---

# Linked Stack Representation (VI)

Pop an item from the top of the stack,  $S$ , if  $S$  is not empty.

```
/* If S is not empty,
   pop an item off the top of S and put it in X. */

void StackPop ( Stack *S, ItemType *X )
{
    StackNode *Temp;
    if ( S->ItemList == NULL ) {          /* Stack empty. */
        fprintf( stderr, "POP:Empty stack.\n"); abort();
    } else {                              /* Stack not empty. */
        Temp = S->ItemList;                /* Top stack node into Temp, */
        *X = Temp->Item;                   /* top stack item into X, */
        S->ItemList = Temp->Link;          /* make next node top, */
        free(Temp);                        /* free storage for former top node. */
    }
}
```

---

# Summary

**Stacks:** Sequences of items that can grow/shrink only at the top end.

**Queues:** ... grow at the rear end, and shrink at the front end.

- Using stack and queue **ADTs** is good programming practice.

Stacks/Queues can have both sequential and linked representations.

- The application programmer only needs to know the interface.
- The implementation details can be hidden.
- Stack representations can be substituted (e.g. exchange sequential for linked).

⇒ *The application program works just the same!*

- Increase program modularity, clarity, simplicity and reliability.

---

## Queue Implementation

Only sequential queue representation in this lecture.  
(Linked queue representation left as self-study exercise.)

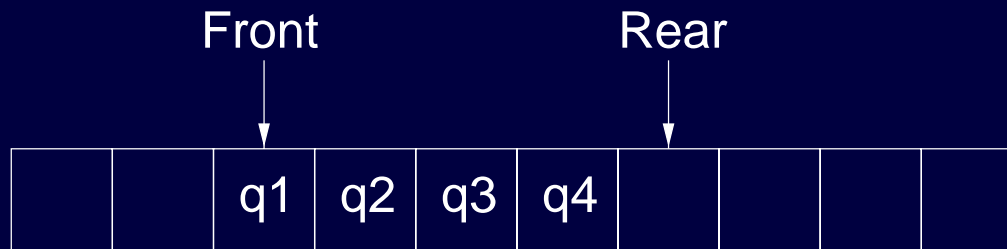
How to implement the specified queue operations.

---

# Queues are more subtle than stacks!

- Need to know where *Front* and *Rear* are, and keep track of them!

For sequential representation use arrays:



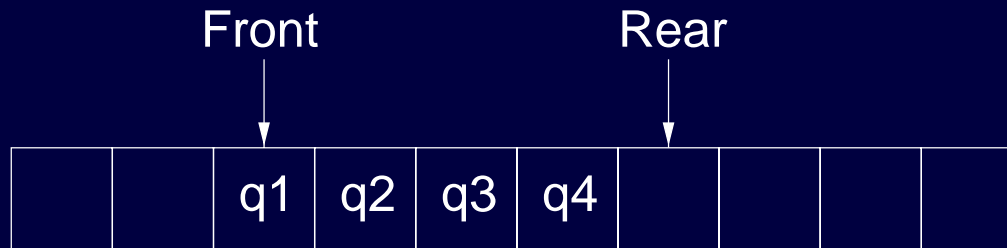
Will this work?

---

# Queues are more subtle than stacks!

- Need to know where *Front* and *Rear* are, and keep track of them!

For sequential representation use arrays:

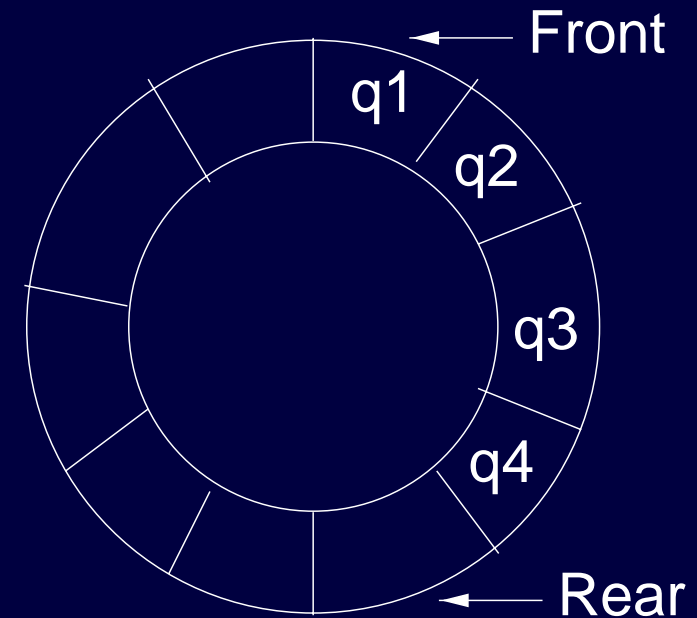


Will this work?

- 💡 Queue representations which march through memory are not very useful!
- Need to confine motion of a queue to bounded region of memory.

# Queues on a Circular Track

- Put sequential queue representation on circular track!



## How can this be done in a program?

- Use linear array of queue items e.g. of size  $N$ .
- Use modular arithmetic: Expression  $X \% N$  keeps values of  $X$  in range  $0..N-1$ .
  - ⇒ For setting *Rear*: Increment old value of *Rear* by one; but when *Rear* reaches  $N$ , set it to zero. (WRAP AROUND!)
  - ⇒  $Rear = (Rear + 1) \% N$ ;
  - ⇒ Same argument for *Front*:  $Front = (Front + 1) \% N$ ;

---

# Sequential Queue Representation (I)

Use arrays of queue items to implement bounded capacity queue.

- Define `MAXQUEUESIZE` as a constant.
- A queue is a `struct` containing
  - number of items currently in queue as e.g. the `Count` member
  - an array of queue items as e.g. its `Items` member
  - an array index (`Front`) for item that can next be removed
  - an array index (`Rear`) that indicates place to insert next item

The file `Queue.c` contains this definition and operations `QueueNew`, `QueueEmpty`, `QueueFull`, `QueueInsert` and `QueueRemove` which are introduced on the following slides.

---

# Sequential Queue Representation (I)

```
/* File: Queue.c */
#include <stdio.h>
#include <stdlib.h>
#include "Queue.h"
#define MAXQUEUE SIZE 100

struct Queue {
    int      Count, Front, Rear;
    ItemType Items[MAXQUEUE SIZE];
} ;
```

---

# Sequential Queue Representation (II)

## Initialization:

- Initialize the queue  $Q$  to be the *empty* queue.
- A queue is defined to be **empty** if its `Count` member is zero.
  - `Front` index is set to zero.
  - `Rear` index is set to zero.

```
/* Initialize the queue Q to be the empty queue. */
Queue * QueueNew ( void )
{
    Queue *Q = calloc( 1, sizeof( Queue ) ) ;
    Q->Count = 0;
    Q->Front = 0;
    Q->Rear = 0;
    return Q ;
}
```

---

# Sequential Queue Representation (III)

**Empty test:** Determine whether or not the queue,  $Q$ , is *empty*.

- A queue is defined to be **empty** if its `Count` member is zero.

```
/* Returns 1 (True) if (and only if)
 * the queue Q is empty.
 */
```

```
int QueueEmpty ( Queue *Q )
{
    return ( Q->Count == 0 );
}
```

---

# Sequential Queue Representation (V)

**Full test:** Determine whether or not the queue,  $Q$ , is *full*.

- A queue is defined to be **full** if its `Count` member is `MAXQUEUESIZE`.

```
/* Returns 1 (True) if (and only if)
 * the queue Q is full.
 */

int QueueFull ( Queue *Q )
{
    return ( Q->Count == MAXQUEUESIZE );
}
```

---

# Sequential Queue Representation (VI)

Insert a new item onto the rear of the queue,  $Q$ .

```
/* If Q is not full, insert a new item R onto rear of Q by:
 * 1. storing R in the item array at position Rear;
 * 2. incrementing Rear by one (WRAP AROUND!);
 * 3. incrementing Count by one.
 */

void QueueInsert (ItemType R, Queue *Q)
{
    if (Q->Count == MAXQUEUE SIZE) {
        printf("INSERT:Not allowed on full queue.\n");exit(1);
    } else {
        Q->Items[Q->Rear] = R;
        Q->Rear = (Q->Rear + 1) % MAXQUEUE SIZE;
        ++ (Q->Count);
    }
}
```

# Sequential Queue Representation (VII)

Provided  $Q$  is not empty, **remove** an item from the front of  $Q$ .

```
/* If Q is not empty, remove first item and put it in F by:  
 * 1. taking F from the item array at position Front;  
 * 2. incrementing Front by one (WRAP AROUND!);  
 * 3. decrementing Count by one.  
 */
```

```
void QueueRemove ( Queue *Q, ItemType *F )  
{  
    if ( Q->Count == 0 ) {  
        printf("REMOVE:Not allowed on empty q.\n");exit(1);  
    } else {  
        *F = Q->Items [Q->Front];  
        Q->Front = (Q->Front + 1) % MAXQUEUE SIZE;  
        -- (Q->Count);  
    }  
}
```

# Using the Queue interface

```
#include "QueueInterface.h"    /* Assume ItemType is char.*/
                               /* Access operations and types for stacks.*/
int main ( void )
{
    Queue *tq;    /* Variable TestQueue of type Queue. */
    char  C;

    tq = QueueNew ( ); /* Make tq empty. */
    QueueInsert ( 'a', tq );
    QueueInsert ( 'b', tq );
    QueueInsert ( 'c', tq );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueInsert ( 'd', tq );
    ...
}
```

= Write this code **without knowing how** the queue is implemented.

# Using the Queue interface

```
#include "QueueInterface.h"    /* Assume ItemType is char.*/
                               /* Access operations and types for stacks.*/
int main ( void )
{
    Queue *tq;    /* Variable TestQueue of type Queue. */
    char C;

⇒  tq = QueueNew ( ); /* Make tq empty. */
    QueueInsert ( 'a', tq );
    QueueInsert ( 'b', tq );
    QueueInsert ( 'c', tq );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueInsert ( 'd', tq );
    ...
}
```

Queue: **Empty**

# Using the Queue interface

```
#include "QueueInterface.h"    /* Assume ItemType is char.*/
                               /* Access operations and types for stacks.*/
int main ( void )
{
    Queue *tq;    /* Variable TestQueue of type Queue. */
    char  C;

    tq = QueueNew ( ); /* Make tq empty. */
⇒ QueueInsert ( 'a', tq );
    QueueInsert ( 'b', tq );
    QueueInsert ( 'c', tq );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueInsert ( 'd', tq );
    ...
}
```

Queue: **a**

# Using the Queue interface

```
#include "QueueInterface.h"    /* Assume ItemType is char.*/
                               /* Access operations and types for stacks.*/
int main ( void )
{
    Queue *tq;    /* Variable TestQueue of type Queue. */
    char  C;

    tq = QueueNew ( ); /* Make tq empty. */
    QueueInsert ( 'a', tq );
    ⇒ QueueInsert ( 'b', tq );
    QueueInsert ( 'c', tq );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueInsert ( 'd', tq );
    ...
}
```

Queue: **a, b**

# Using the Queue interface

```
#include "QueueInterface.h" /* Assume ItemType is char.*/
/* Access operations and types for stacks.*/
int main ( void )
{
    Queue *tq; /* Variable TestQueue of type Queue. */
    char C;

    tq = QueueNew ( ); /* Make tq empty. */
    QueueInsert ( 'a', tq );
    QueueInsert ( 'b', tq );
    ⇒ QueueInsert ( 'c', tq );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueInsert ( 'd', tq );
    ...
}
```

Queue: **a, b, c**

# Using the Queue interface

```
#include "QueueInterface.h"    /* Assume ItemType is char.*/
                               /* Access operations and types for stacks.*/
int main ( void )
{
    Queue *tq;    /* Variable TestQueue of type Queue. */
    char  C;

    tq = QueueNew ( ); /* Make tq empty. */
    QueueInsert ( 'a', tq );
    QueueInsert ( 'b', tq );
    QueueInsert ( 'c', tq );
    ⇒ QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueInsert ( 'd', tq );
    ...
}
```

Queue: **b, c**      Print: **Just got: a.**

# Using the Queue interface

```
#include "QueueInterface.h"    /* Assume ItemType is char.*/
                               /* Access operations and types for stacks.*/
int main ( void )
{
    Queue *tq;    /* Variable TestQueue of type Queue. */
    char  C;

    tq = QueueNew ( ); /* Make tq empty. */
    QueueInsert ( 'a', tq );
    QueueInsert ( 'b', tq );
    QueueInsert ( 'c', tq );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
⇒ QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueInsert ( 'd', tq );
    ...
}
```

Queue: **c**      Print: **Just got: b.**

# Using the Queue interface

```
#include "QueueInterface.h"    /* Assume ItemType is char.*/
                               /* Access operations and types for stacks.*/
int main ( void )
{
    Queue *tq;    /* Variable TestQueue of type Queue. */
    char  C;

    tq = QueueNew ( ); /* Make tq empty. */
    QueueInsert ( 'a', tq );
    QueueInsert ( 'b', tq );
    QueueInsert ( 'c', tq );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    QueueRemove ( tq, &C ); printf ( "Just got: %c.\n", C );
    ⇒ QueueInsert ( 'd', tq );
    ...
}
```

Queue: **c, d**

---

# Where to get more information?

Thomas A. Standish

“Data Structures, Algorithms & Software Principles in C”

Addison-Wesley, 1995

p. 253 ff (Chapter 7)