Chapter 2:
Linear Data Structure - Stack

Data

- **Atomic data**
  - Consists of a single piece of information, i.e., cannot be divided into other meaningful pieces of data
  - Example: integer 1223
- **Composite data**
  - Can be broken down into subfields that have meaning
  - Example: telephone numbers = country code + area code + phone number

Data type

- A **data type** is collection of objects and a set of operations that act on those objects.
- For example, the data type **int** consists of
  - Objects: {0, +1, -1, +2, -2, ..., INT_MAX, INT_MIN}, where INT_MAX and INT_MINare the largest and smallest integers that can be represented on a machine
  - Operations: arithmetic operations +, -, *, /, %, equality/inequality testing etc.

Data Abstraction

Knowing the representation of the objects of a data type can be dangerous. **Why?**

Abstraction

- What a data type can do is known
- How it is done is hidden

Abstract Data Type

- A **specification** of a **set of data** and the **set of operations** that can be performed on the data (without being interested in the specific implementation)
- Independent of various **concrete** implementations
- The definition can be **mathematical** or it can be **programmed** as an **interface** (e.g., as an interface class in C++)
- Allows to have several different implementations and respectively different efficiencies

An ADT is a contract. It does not imply implementation.

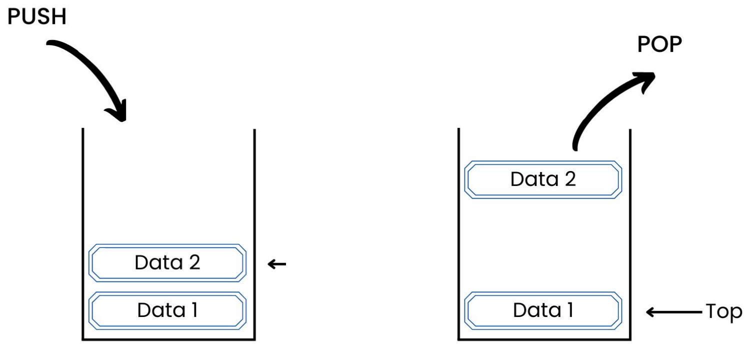
Data Structure

- An aggregation of atomic and composite data into a set with defined relationships
- Data organization and storage format that enables efficient access and modification
- Can be nested
- Examples:
  - Array:
    - Values: homogeneous sequence of data;
    - Operations: create, retrieve, remove etc.
- Applications:
  - memory management in OS, database applications, task scheduling, indexing in databases, social networking applications etc.

Basic Data Structures

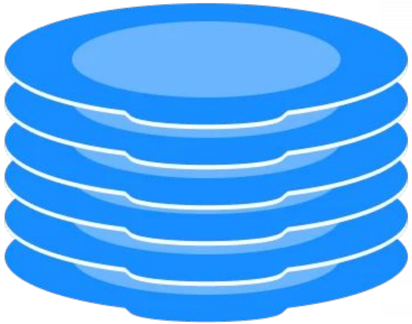
- Linear data structures
  - Organize their data elements in a linear fashion, i.e., sequentially, where each data element attaches one after the other (i.e., each element has a unique successor)
  - Examples: stack, queue, list
- Non-linear data structures
  - Data items are not organized sequentially, i.e., each element can have more than one successor
  - Examples: tree, graph

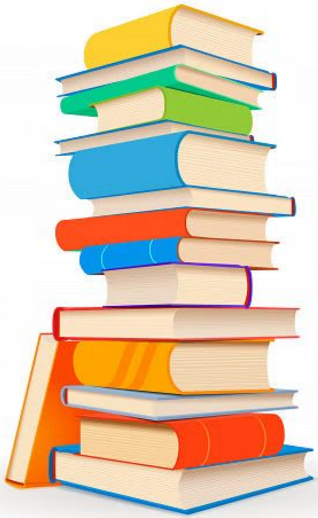
Stack



Stack

- A linear data structure where all **insertions** and **deletions** are **restricted** to one end called the **top**
- Last element inserted into a stack is the first element removed
- **Last-In-First-Out (LIFO)**

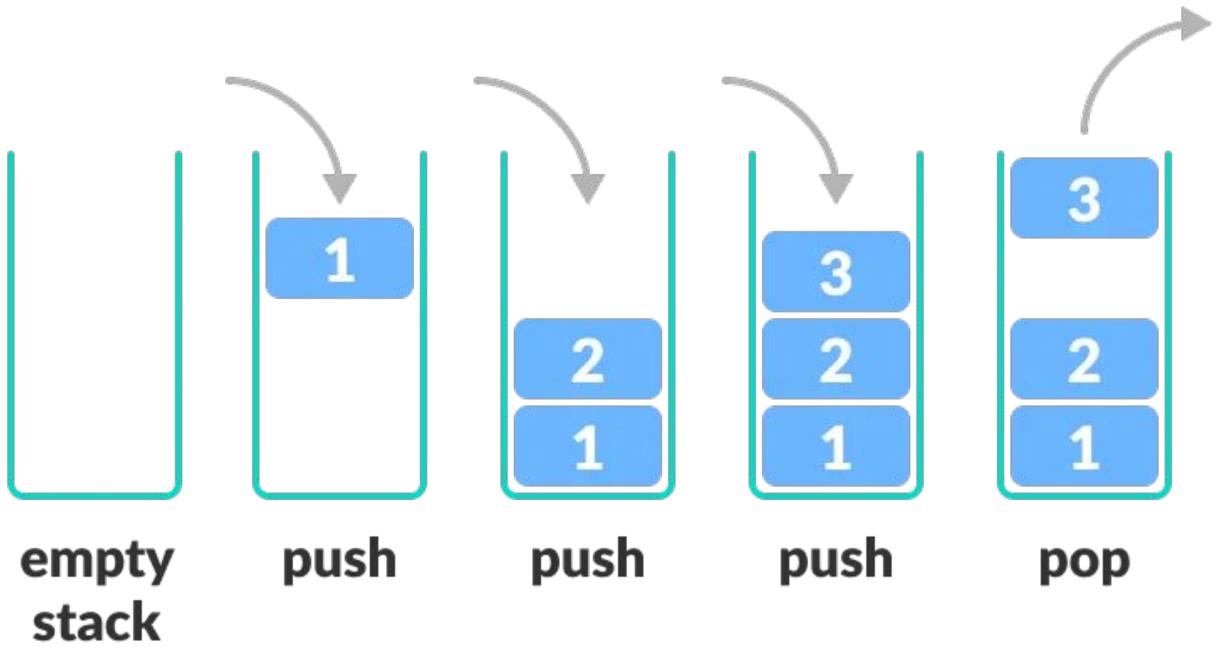




Basic stack operations

- **Push**
  - Adds an item to the top of the stack
  - While pushing, ensure that there is room for the new item
  - Not enough room = (stack) overflow state
- **Pop**
  - Removes the item at the top of the stack
  - While popping, ensure that there is an item at the top of the stack (i.e., the stack is not empty)
- **Top/peek**: copies the item at the top of the stack
- **IsEmpty**: checks if the stack is empty
- **IsFull**: checks if the stack is full

Push and Pop Operations



Stack ADT

```
ADT Stack is
    Objects: a finite ordered list with zero or more elements
    Functions:
        For all stack ∈ Stack, item ∈ element, maxStackSize ∈ positive integer
        Stack CreateS(maxStackSize) :=
            Create an empty stack whose maximum size is maxStackSize
        Boolean isFull(stack, maxStackSize) :=
            if (number of elements in stack == maxStackSize) return TRUE
            else return FALSE
        Stack Push(stack, item) :=
            if (IsFull(stack)) stackFull
            else insert item into top of stack and return
        Boolean isEmpty(stack) :=
            if (stack == CreateS(maxStackSize)) return TRUE
            else return FALSE
        Element Pop(stack) :=
            if (IsEmpty(stack)) return
            else remove and return the element at the top of the stack
```

Applications of Stack

1. Reversing a list
2. Decimal to Binary Conversion
3. Parentheses matching
4. Expression evaluation
    a. Infix to postfix conversion
    b. Postfix expression evaluation

Applications of Stack

See class notes for

1. Reversing a list
2. Decimal to Binary Conversion
3. Parentheses matching

Evaluation of expressions

An arithmetic expression can be represented in three different forms:

1. Infix
   The operator comes between the operands, e.g. a+b
2. Postfix
   The operator comes after the operands, e.g. ab+
3. Prefix
   The operator comes before the operands, e.g. -ab

An arithmetic expression in the infix form may contain parentheses whereas postfix and prefix forms are parenthesis-free.

Evaluation of expressions

Infix expressions cannot be directly evaluated. They must be analyzed to determine the order in which expressions are to be evaluated.

For example, in the expression `a + b / (c - d) * e`, the order of evaluation would be

i. `(c - d)` must be evaluated first; let's say the result is r_{cd}.
ii. $b\ /\ r_{cd}$ must be evaluated next; let's say the result is r_{bcd}.
iii. Then $r_{bcd}\ *\ e$ must be evaluated; let's say the result is r_{bcde}.
iv. Finally, $a\ +\ r_{bcde}$ must be evaluated

Compilers typically use postfix notation, which is parenthesis-free.

Operator Precedence

Within any programming language, there is a precedence hierarchy that determines the order in which operators are evaluated.

Operators with the highest precedence are evaluated first.

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a--	Suffix/postfix increment and decrement	
	type() *type*{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
	. ->	Member access	
3	++a --a	Prefix increment and decrement	Right-to-left
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of[note 1]	
	co_await	await-expression (C++20)	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	<=>	Three-way comparison operator (since C++20)	
9	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
10	== !=	For equality operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13	|	Bitwise OR (inclusive or)	
14	&&	Logical AND	
15	||	Logical OR	
16	a?b:c	Ternary conditional[note 2]	Right-to-left
	throw	throw operator	
	co_yield	yield-expression (C++20)	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= |=	Compound assignment by bitwise AND, XOR, and OR	
17	,	Comma	Left-to-right

Associativity

When an expression contains two or more operators at the same level of precedence, the associativity of the operators determines the order in which operations are performed.

An operator is said to be **left associative** if it groups from left to right.

All binary arithmetic operators are left associative. Thus, `i - j + k - l` is equivalent to `((i - j) + k) - l`

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>1</td><td>::</td><td>Scope resolution</td><td rowspan="2">Left-to-right</td></tr>
<tr><td>2</td><td>a++ a--
type() type{}
a()
a[]
. -></td><td>Suffix/postfix increment and decrement
Functional cast
Function call
Subscript
Member access</td></tr>
<tr><td>3</td><td>++a --a
+a -a
! ~
(type)
*a
&a
sizeof
co_await
new new[]
delete delete[]</td><td>Prefix increment and decrement
Unary plus and minus
Logical NOT and bitwise NOT
C-style cast
Indirection (dereference)
Address-of
Size-of[note 1]
await-expression (C++20)
Dynamic memory allocation
Dynamic memory deallocation</td><td>Right-to-left</td></tr>
<tr><td>4</td><td>.* ->*</td><td>Pointer-to-member</td><td rowspan="12">Left-to-right</td></tr>
<tr><td>5</td><td>a*b a/b a%b</td><td>Multiplication, division, and remainder</td></tr>
<tr><td>6</td><td>a+b a-b</td><td>Addition and subtraction</td></tr>
<tr><td>7</td><td><< >></td><td>Bitwise left shift and right shift</td></tr>
<tr><td>8</td><td><=></td><td>Three-way comparison operator (since C++20)</td></tr>
<tr><td>9</td><td>< <=
> >=</td><td>For relational operators < and ≤ respectively
For relational operators > and ≥ respectively</td></tr>
<tr><td>10</td><td>== !=</td><td>For equality operators = and ≠ respectively</td></tr>
<tr><td>11</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>12</td><td>^</td><td>Bitwise XOR (exclusive or)</td></tr>
<tr><td>13</td><td>|</td><td>Bitwise OR (inclusive or)</td></tr>
<tr><td>14</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>15</td><td>||</td><td>Logical OR</td></tr>
<tr><td>16</td><td>a?b:c
throw
co_yield
=
+= -=
*= /= %=
<<= >>=
&= ^= |=</td><td>Ternary conditional[note 2]
throw operator
yield-expression (C++20)
Direct assignment (provided by default for C++ classes)
Compound assignment by sum and difference
Compound assignment by product, quotient, and remainder
Compound assignment by bitwise left shift and right shift
Compound assignment by bitwise AND, XOR, and OR</td><td>Right-to-left</td></tr>
<tr><td>17</td><td>,</td><td>Comma</td><td>Left-to-right</td></tr>
</table>

Infix to Postfix conversion using stack

Algorithm: Infix to Postfix conversion

Input: An infix expression, exp

Output: exp in postfix form

Steps:

1. Initialize a stack

Infix to Postfix conversion using stack (Contd.)

2. For each character c in exp
    2.1. If c is an operand, concatenate c to the postfix expression
    2.2. Else if c is an open parenthesis, push it into the stack
    2.3. Else if c is a closed parenthesis,
        2.3.1. Repeat
            2.3.1.1. pop the stack and concatenate the popped character to the postfix expression
        2.3.2. until an open parenthesis is encountered
    2.4. Else
        2.4.1. While the stack is not empty and precedence of c is less than or equal to that of the top (and the top at the stack is not an open parenthesis)
            2.4.1.1. Pop the stack and concatenate the popped operator to the postfix expression
        2.4.2. End while
        2.4.3. Push c
    2.5. End if
3. End for

Infix to Postfix conversion using stack (Contd.)

3. While the stack is not empty
   3.1. Pop the stack and concatenate the popped operator to the postfix expression
4. End while
5. Return the postfix expression

Infix to Postfix conversion: Example

Convert A + B * C to postfix

Token	Stack content	Postfix expression	Algorithm step
A	{}	A	2.1

Infix to Postfix conversion: Example

Convert A + B * C to postfix

Token	Stack content	Postfix expression	Algorithm step
A	{}	A	2.1
+	{+}	A	2.4.3

Infix to Postfix conversion: Example

Convert A + B * C to postfix

Token	Stack content	Postfix expression	Algorithm step
A	{}	A	2.1
+	{+}	A	2.4.3
B	{+}	AB	2.1

Infix to Postfix conversion: Example

Convert A + B * C to postfix

Token	Stack content	Postfix expression	Algorithm step
A	{}	A	2.1
+	{+}	A	2.4.3
B	{+}	AB	2.1
*	{+, *}	AB	2.4.3
C	{+, *}	ABC	2.1
	{+}	ABC*	3.1
	{}	ABC*+	3.1

Example: Convert A * (B + C) / (D + E - F) to postfix

Token	Stack content	Postfix expression	Algorithm step
A	{}	A	2.1
*	{*}	A	2.4.3
(	{*, (}	A	2.2
B	{*, (}	AB	2.1
+	{*, (, +}	AB	2.4.3
C	{*, (, +}	ABC	2.1
)	{*}	ABC+	2.3
/	{/}	ABC+*	2.4.1.1, 2.4.3

Example: Convert A * (B + C) / (D + E - F) to postfix

Token	Stack content	Postfix expression	Algorithm step
A	{}	A	2.1
*	{*}	A	2.4.3
(	{*, (}	A	2.2
B	{*, (}	AB	2.1
+	{*, (, +}	AB	2.4.3
C	{*, (, +}	ABC	2.1
)	{*}	ABC+	2.3
/	{/}	ABC+*	2.4.1.1, 2.4.3

Token	Stack content	Postfix expression	Algorithm step
(	{/, (}	ABC+*	2.2
D	{/, (}	ABC+*D	2.1
+	{/, (, +}	ABC+*D	2.4.3
E	{/, (, +}	ABC+*DE	2.1
-	{/, (, -}	ABC+*DE+	2.4.1.1, 2.4.3
F	{/, (, -}	ABC+*DE+F	2.1
)	{/}	ABC+*DE+F-	2.3
	{}	ABC+*DE+F-/	3.1

Infix to Postfix conversion: Exercise

Convert the following infix expression to postfix:

(A + B - C / D) * E + (F - G) / H

Evaluating postfix expressions

The main idea is to

- Read the tokens one-by-one from left to right,
- Push the token into the stack if it is an operand; if it is an operator, pop two operands from the stack and perform the operation, then push the result into the stack.
- Continue until all tokens are considered.

The result of the expression will be the content of the stack.

Evaluating postfix expressions: Algorithm steps

1. Initialize a stack
2. For each token t in the postfix expression
   2.1. If t is an operand, push it into the stack
   2.2. Else
      2.2.1. Operand2 = pop from the stack
      2.2.2. Operand1 = pop from the stack
      2.2.3. Operator = t
      2.2.4. Result = evaluated the expression Operand1 Operator Operand2
      2.2.5. Push the result into the stack
   2.3. End if
3. End for
4. Result = Pop from the stack
5. Return Result

Example: Evaluate the postfix expression 2 3 4 * +

Token	Stack content	Operation	Algorithm step
2	{2}		2.1

Example: Evaluate the postfix expression 2 3 4 * +

Token	Stack content	Operation	Algorithm step
2	{2}		2.1
3	{2, 3}		2.1
4	{2, 3, 4}		2.1
*	{2, 12}	Operand2 = 4 Operand1 = 3 Result = 3 * 4 = 12	2.2

Example: Evaluate the postfix expression 2 3 4 * +

Token	Stack content	Operation	Algorithm step
2	{2}		2.1
3	{2, 3}		2.1
4	{2, 3, 4}		2.1
*	{2, 12}	Operand2 = 4 Operand1 = 3 Result = 3 * 4 = 12	2.2
+	{14}	Operand2 = 12 Operand1 = 2 Result = 2 + 12	2.2

Value of 2 3 4 * + = 14

Evaluating postfix expressions: Exercise

Evaluate the following postfix expression: ABC+*DE+F-/

Where A = 2, B = F = 1, C = 3, D = 5, E = 4.

Stack ADT

```
ADT Stack is
   Objects: a finite ordered list with zero or more elements
   Functions:
         For all stack ∈ Stack, item ∈ element, maxStackSize ∈ positive integer
         Stack CreateS(maxStackSize) :=
                   Create an empty stack whose maximum size is maxStackSize
         Boolean isFull(stack, maxStackSize) :=
                   if (number of elements in stack == maxStackSize) return TRUE
                   else return FALSE
         Stack Push(stack, item) :=
                   if (IsFull(stack)) stackFull
                   else insert item into top of stack and return
         Boolean isEmpty(stack) :=
                   if (stack == CreateS(maxStackSize)) return TRUE
                   else return FALSE
         Element Pop(stack) :=
                   if (IsEmpty(stack)) return
                   else remove and return the element at the top of the stack
```

Stack ADT (as an interface class in C++)

```
class Stack
{
public:
    virtual ~Stack() {}

    virtual bool isEmpty() const = 0;
    virtual bool isFull() const = 0;

    virtual bool push(const int element) =0;
    virtual bool pop(int &element) = 0;
    virtual bool top(int &element) const = 0;
};
```

Stack ADT (as an abstract class in C++)

```
class Stack
{
public:
    virtual ~Stack() {}

    virtual bool isEmpty() const = 0;
    virtual bool isFull() const = 0;
    virtual bool push(const int element) {
        if(!isFull()) {
            return _push(element);
        } else {
            return false;
        }

    }
    virtual bool _push(const int element) = 0;
```

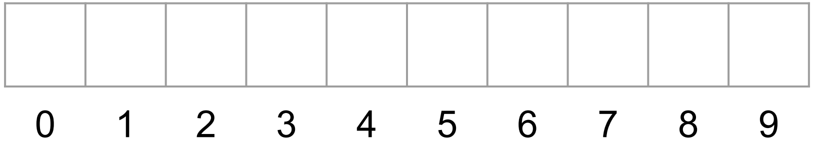
```
    virtual bool pop(int &element) {
        if(!isEmpty()) {
            return _pop(element);
        } else {
            return false;
        }
    }
    virtual bool _pop(int &element) = 0;
    virtual bool top(int &element) const {
        if(!isEmpty()) {
            return _top(element);
        } else {
            return false;
        }
    }
    virtual bool _top(int &element) const =
0;
};
```

Implementation of a stack

1. Array implementation
2. Linked list implementation (will be covered later)

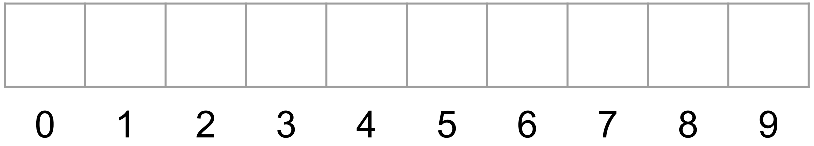
Array-based Stack

- The Stack ADT can be implemented using an array.
- To create a stack, we initialize an array of maxStackSize.

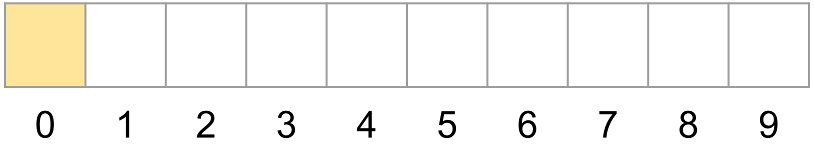


Array-based Stack

- The Stack ADT can be implemented using an array.
- To create a stack, we initialize an array of maxStackSize.
- We add elements from left to right (i.e. starting from index 0).

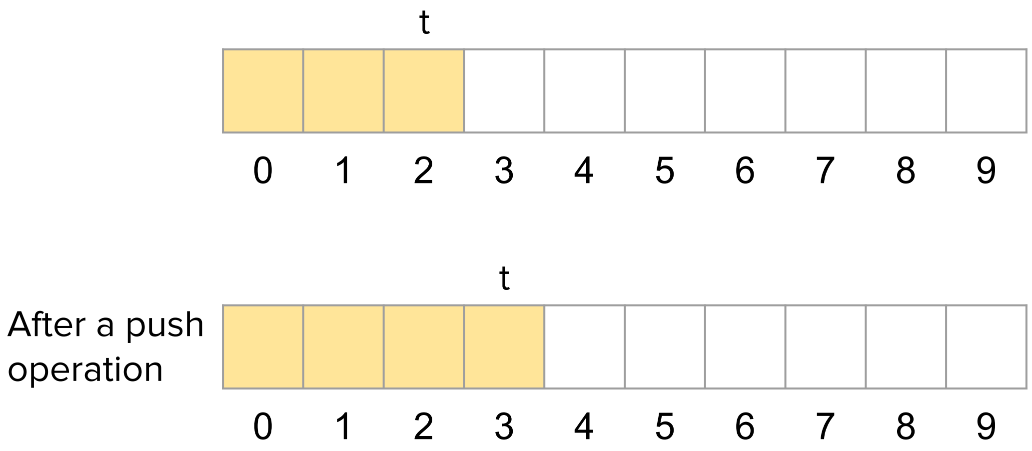


After a push operation



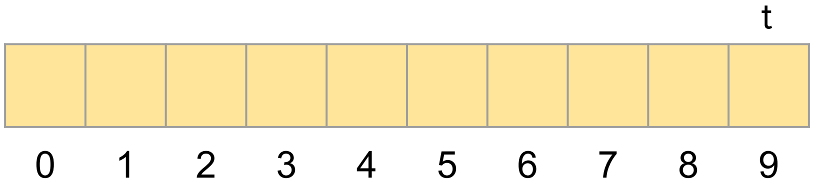
Array-based Stack

- The Stack ADT can be implemented using an array.
- To create a stack, we initialize an array of maxStackSize.
- We add elements from left to right (i.e. starting from index 0).
- We use a variable to keep track of the index of the top element.



Array-based Stack

- The Stack ADT can be implemented using an array.
- To create a stack, we initialize an array of maxStackSize.
- We add elements from left to right (i.e. starting from index 0).
- We use a variable to keep track of the index of the top element.
- The array may become full.



t == maxStackSize - 1

Array-based Stack (in C++)

```
class ArrayStack : public Stack {
private:
   int *data;
   int topIndex;
   int maxStackSize;

public:
   /* Initialize a stack */
   ArrayStack(int maxStackSize);

   ~ArrayStack()  { delete[] data; }

   /* Check if the stack is empty */
   bool isEmpty() const;

   /* Check if the stack is full */
   bool isFull() const;

   /* Push an element to the top of the stack */
   bool push(const int element);

   /* Pop the element at the top of the stack */
   bool pop(int &element);

   /* Copy the element at the top of the stack */
   bool top(int &element) const;
};
```

Array-based Stack (in C++)

```
/* Initialize a stack */
ArrayStack::ArrayStack(int size)
   : maxStackSize(size),
     topIndex(-1),
     data(new int[size])
{}
```

```
/* Initialize a stack */
bool ArrayStack::isEmpty() const
{
   return topIndex < 0;
}
```

```
/* Check if the stack is full */
bool ArrayStack::isFull() const
{
   return topIndex == maxStackSize - 1;
}
```

```
/* Push an element to the top of the stack */
bool ArrayStack::push(const int element) {
   if(!isFull()) {
       data[++topIndex] = element;
       return true;
   } else {
       std::cout << "Stack is full!\n";
       return false;
   }
}
```

Array-based Stack (in C++)

```
/* Pop the element at the top of the stack */
bool ArrayStack::pop(int &element) {
   if(!isEmpty()) {
       element = data[topIndex--];
       return true;
   } else {
       std::cout << "Stack is empty!\n";
       return false;
   }
}
```

```
/* Copy the element at the top of the stack */
bool ArrayStack::top(int &element) const {
   if(!isEmpty()) {
       element = data[topIndex];
       return true;
   } else {
       std::cout << "Stack is empty!\n";
       return false;
   }
}
```

Performance and Limitations

Performance

- Each operation runs in time O(1)

Limitations

- The maximum size of the stack must be defined a priori.

Queue

Contents

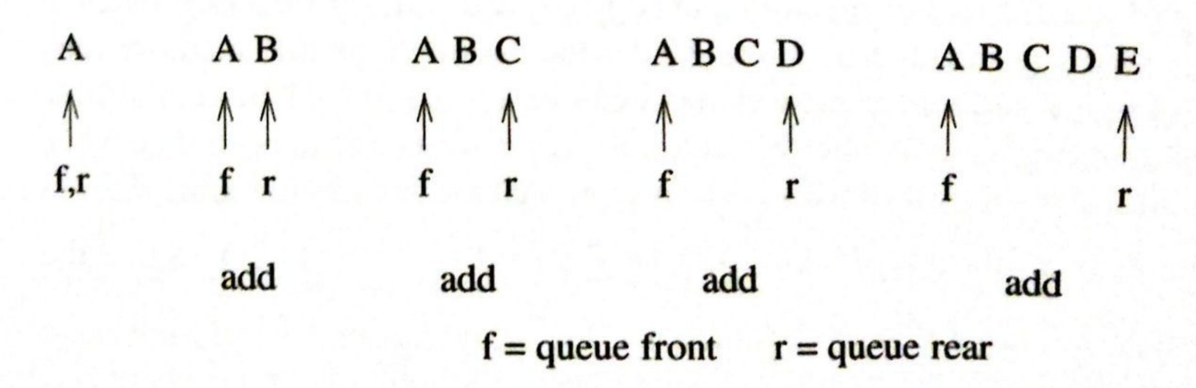
- Definition
- Operations
- Applications
- Queue ADT
- Implementation (Static)
- Circular Queue
- Priority Queue

Queue

- An ordered list in which insertions and deletions take place at different ends
  - Insertions at “rear” end
  - Deletions at “front” end
- First-In-First-Out (FIFO) list
  - The first one inserted is the first one to be removed

Queue

- An ordered list in which insertions and deletions take place at different ends
  - Insertions at “rear” end
  - Deletions at “front” end
- First-In-First-Out (FIFO) list
  - The first one inserted is the first one to be removed



Queue operations

Main queue operations

- **enqueue**: adds a new element at the end of the queue
- **dequeue**: removes and returns the element at the front of the queue

Auxiliary queue operations

- **front**: returns the element at the front without removing it
- **rear**: returns the element at the rear without removing it
- **isEmpty**: indicates whether the queue is empty
- **isFull**: indicates whether there is enough space for a new element

Applications

Direct applications

- Waiting lists
- Job scheduling
- Access to shared resources such as printers etc.

Indirect applications

- Auxiliary data structures for algorithms
- Components of other data structures

Queue ADT

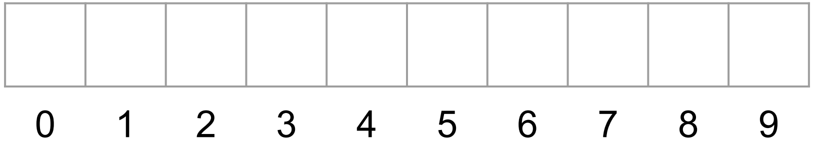
```
ADT Queue is
    Objects: a finite ordered list with zero or more elements
    Functions:
        For all queue ∈ Queue, item ∈ element, maxQueueSize ∈ positive integer
        Queue CreateQ(maxQueueSize) :=
            Create an empty queue whose maximum size is maxQueueSize
        Boolean isFull(queue, maxQueueSize) :=
            if (number of elements in queue == maxQueueSize) return TRUE
            else return FALSE
        Queue Enqueue(queue, item) :=
            if (IsFull(queue)) queueFull
            else insert item at rear of queue and return queue
        Boolean isEmpty(queue) :=
            if (queue == CreateQ(maxQueueSize)) return TRUE
            else return FALSE
        Element Dequeue(queue) :=
            if (IsEmpty(queue)) return
            else remove and return the item at front of queue
```

Static Implementation of a Queue (using arrays)

- The Queue ADT can be implemented using an array.
- To create a queue, we initialize an array of maxQueueSize.

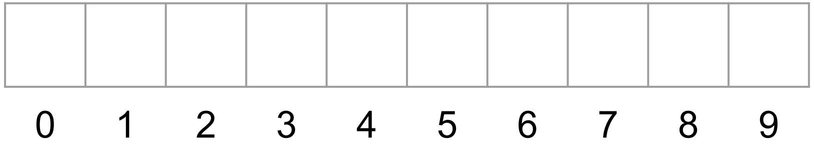
Static Implementation of a Queue (using arrays)

- The Queue ADT can be implemented using an array.
- To create a queue, we initialize an array of maxQueueSize.

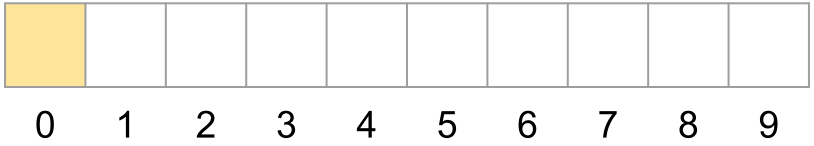


Static Implementation of a Queue (using arrays)

- The Queue ADT can be implemented using an array.
- To create a queue, we initialize an array of maxQueueSize.
- We add elements from left to right (i.e. starting from index 0).



After an enqueue operation



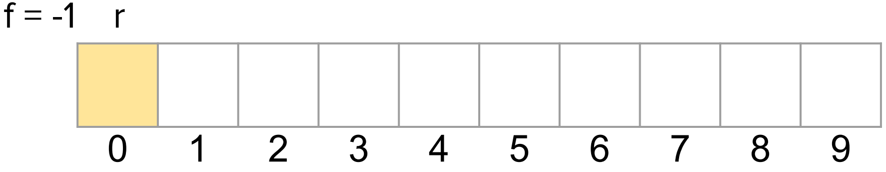
Static Implementation of a Queue (using arrays)

- The Queue ADT can be implemented using an array.
- To create a queue, we initialize an array of maxQueueSize.
- We add elements from left to right (i.e. starting from index 0).
- We use two variables, f and r, to keep track of the index of the front element and that of the rear element.
  Let f and r be - 1 in the beginning. During enqueue, r is increased by 1, and during dequeue, f is increased by 1.

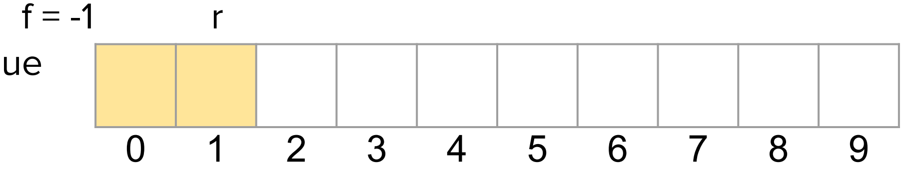
Static Implementation of a Queue (using arrays)

- The Queue ADT can be implemented using an array.
- To create a queue, we initialize an array of maxQueueSize.
- We add elements from left to right (i.e. starting from index 0).
- We use two variables, f and r, to keep track of the index of the front element and that of the rear element.

  Let f and r be - 1 in the beginning. During enqueue, r is increased by 1, and during dequeue, f is increased by 1.



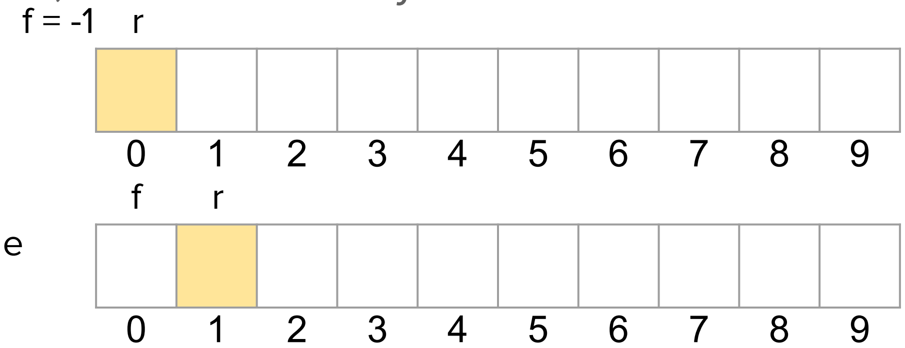
After an enqueue operation



Static Implementation of a Queue (using arrays)

- The Queue ADT can be implemented using an array.
- To create a queue, we initialize an array of maxQueueSize.
- We add elements from left to right (i.e. starting from index 0).
- We use two variables, f and r, to keep track of the index of the front element and that of the rear element.

  Let f and r be - 1 in the beginning. During enqueue, r is increased by 1, and during dequeue, f is increased by 1.



After a dequeue operation

Static Implementation of a Queue (using arrays)

Algorithm: createQueue(maxQueueSize)

Steps:

1. Initialize an array of size maxQueueSize
2. Initialize front and rear to -1

Static Implementation of a Queue (using arrays)

Algorithm: isEmpty()

Steps:

1. If rear == front, return true
2. Else return false

Algorithm: isFull()

Steps:

1. If rear == maxQueueSize - 1, return true
2. Else return false

Static Implementation of a Queue (using arrays)

Algorithm: enqueue(value)
Steps:

1. If queue is not full, increase rear by 1 and store value at index rear of the array
2. Else print Queue overflow message.

Algorithm: dequeue()
Steps:

1. If the queue is not empty, increase front by 1 and return the value at index front of the array.
2. Else print Queue underflow message.

Static Implementation of a Queue (using arrays)

Problem with this implementation

- The queue gradually shifts to the right. Example: suppose maxQueueSize = 4, and operations performed are enqueue(2), enqueue(5), dequeue(), enqueue(4), dequeue(), enqueue(3).

	0	1	2	3	f	r
					-1	-1
enqueue(2)	2				-1	0
enqueue(5)	2	5			-1	1
dequeue()		5			0	1
enqueue(4)		5	4		0	2
dequeue()			4		1	2
enqueue(3)			4	3	1	3
enqueue(7)?						

Static Implementation of a Queue (using arrays)

Problem with this implementation

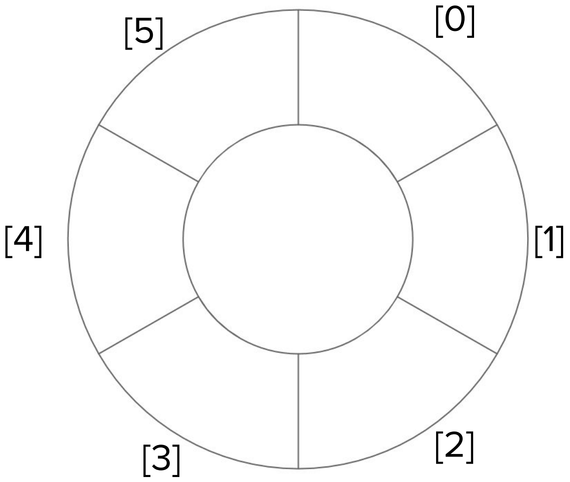
- The queue gradually shifts to the right.

Solution

1. On queue overflow, move all elements to the left so that the queue front is always at 0. However, it is very time consuming.
2. Use circular queue

Circular Queue

- Aka circular buffer, ring buffer
- A FIFO list
- The last position is connected back to the first position to make a circle.
- The position next to maxQueueSize - 1 is 0
- The position that precedes 0 is maxQueueSize - 1
- When the queue rear is at maxQueueSize -1 , the Next element is put into position 0.



Implementation of a circular queue with n-1 space used

Algorithm: createQueue(maxQueueSize)

Steps:

1. Initialize an array of size maxQueueSize
2. Initialize front and rear to 0

Implementation of a circular queue with n-1 space used

Algorithm: isEmpty()

Steps:

1. If rear == front, return true
2. Else return false

Algorithm: isFull()

Steps:

1. If front == (rear + 1) % maxQueueSize, return true
2. Else return false

Implementation of a circular queue with n-1 space used

Algorithm: enqueue(value)

Steps:

1. If queue is not full
    a. rear = (rear + 1) % maxQueueSize
    b. data[rear] = value
2. Else
    a. print Queue overflow message.
3. Endif

Implementation of a circular queue with n-1 space used

Algorithm: dequeue()

Steps:

1. If the queue is not empty
    a. front = (front + 1) % maxQueueSize
    b. return queue[front]
2. Else
    a. print Queue underflow message
3. Endif

Priority Queue

- The element to be deleted is the one with the highest or lowest priority
- At any time, an element with arbitrary priority can be inserted into the queue.

Ascending queue

- Only the smallest item can be removed.

Descending queue

- Only the largest item can be removed.