University of North Texas

CSCE 2110
Foundations of Data Structures

Abstract Data Type, List, Stack, and Queue

Contents

- Abstract data type
  - Array
- List, Stack, and Queue
  - ADT
  - Implementations
  - Algorithm analysis
  - Applications
  - Postfix conversion

Elementary Data Structures

"Mankind's progress is measured by the number of things we can do without thinking."

Elementary data structures such as stacks, queues, lists, and heaps are the "off-the-shelf" components we build our algorithm from.

A data organization, management, and storage format that enables efficient access and modification.

There are two aspects to any data structure:

- The abstract operations which it supports (abstract data type, ADT).
- The implementation of these operations.

Data Abstraction

- That we can describe the behavior of our data structures in terms of abstract operations is why we can use them without thinking.
  - Push(x, s) – Insert item x at the top of stack s.
  - Pop(s) – Return (and remove) the top item of stack s.

- That there are different implementations of the same abstract operations enables us to optimize performance in difference circumstances.

Fundamental Data Structures

Data structures can be neatly classified as either contiguous or linked, depending upon whether they are based on arrays or pointers:

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.

- Linked data structures are composed of multiple distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

Array — Abstract Data Type

- The array is an abstract data type (ADT) that holds a collection of elements accessible by an index.
  - elements can be anything, primitive types such as integers to more complex types like instances of classes

Array — Abstract Data Type

- The array is an abstract data type (ADT) that holds a collection of elements accessible by an index.
  - elements can be anything, primitive types such as integers to more complex types like instances of classes

- Minimal Required Functionality
  - set(i, v) -> Sets the value of index i to v
  - get(i) -> Returns the value of index i in the array

Array — Data Structure

- An array is a structure of fixed-size data records such that each element can be efficiently located by its index or (equivalently) address.
- Advantages of contiguously-allocated arrays include
  - Constant-time access given the index.
  - Arrays consist purely of data, so no space is wasted with links or other formatting information.
  - Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures.

Dynamic Arrays

- Unfortunately, we cannot adjust the size of simple arrays in the middle of a program's execution.
- Compensating by allocating extremely large arrays can waste a lot of space.
- With dynamic arrays we start with an array of size 1 and double its size from m to $2m$ each time we run out of space.

Dynamic Arrays

- Unfortunately, we cannot adjust the size of simple arrays in the middle of a program's execution.
- Compensating by allocating extremely large arrays can waste a lot of space.
- With dynamic arrays we start with an array of size 1 and double its size from m to $2m$ each time we run out of space.

How many times will we double for n elements?

Dynamic Arrays

- Unfortunately, we cannot adjust the size of simple arrays in the middle of a program's execution.
- Compensating by allocating extremely large arrays can waste a lot of space.
- With dynamic arrays we start with an array of size 1 and double its size from m to $2m$ each time we run out of space.

How many times will we double for n elements?

$$\lceil \log_2 n \rceil$$

How Much Total Work?

- The apparent waste in this procedure involves the recopying of the old contents on each expansion.

Item No.:	1	2	3	4	5	6	7	8	9	10 …

How Much Total Work?

- The apparent waste in this procedure involves the recopying of the old contents on each expansion.

Item No.:	1	2	3	4	5	6	7	8	9	10 …
Array size:	1	2	4	4	8	8	8	8	16	16 …
Cost:	1	2	3	1	5	1	1	1	9	1 …

$$\underbrace{1 + 1 + \cdots 1}_{n \text{ items}} + \underbrace{(1 + 2 + 4 + \cdots)}_{\lg(n-1)+1 \text{ items}}$$

$$\text{Cost} \le n + 2n = 3n = O(n)$$

List ADT

- A list contains elements of same type arranged in sequential order operations performed on the list:
  - get() – Return an element from the list at any given position.
  - insert() – Insert an element at any position of the list.
  - remove() – Remove the first occurrence of any element from a non-empty list.
  - removeAt() – Remove the element at a specified location from a non-empty list.
  - replace() – Replace an element at any position by another element.
  - size() – Return the number of elements in the list.
  - isEmpty() – Return true if the list is empty, otherwise return false.
  - isFull() – Return true if the list is full, otherwise return false.

Pointers and Linked Structures

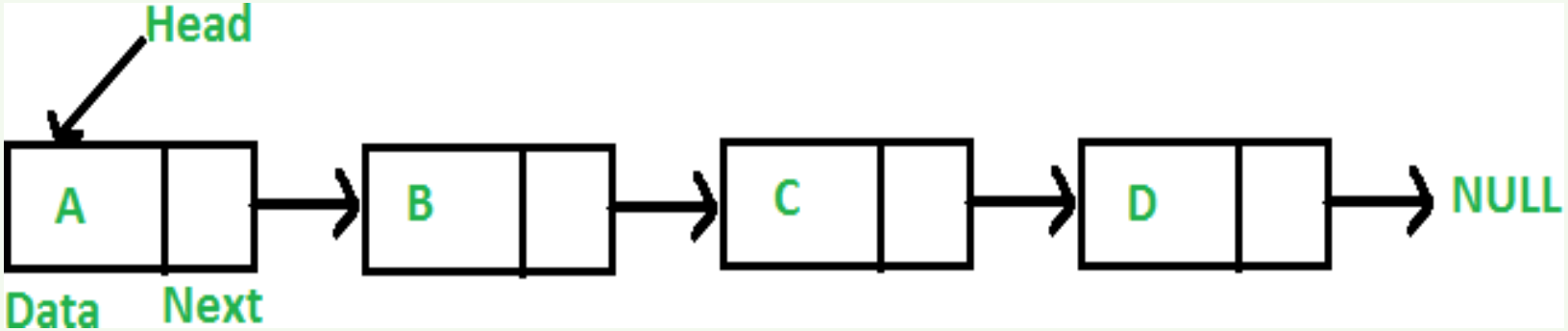
- Pointers represent the address of a location in memory. A cell-phone number can be thought of as a pointer to its owner as they move about the planet.

- In C/C++, *p denotes the item pointed to by p, and &x denotes the address (i.e., pointer) of a particular variable x.

- A special NULL pointer value is used to denote structure terminating or unassigned pointers.

Linked List

```
typedef struct list {
    item_type item;          /* data item */
    struct list *next;       /* point to successor */
} list;
```



Searching a List

- Searching in a linked list can be done iteratively or recursively.

```
list *search_list(list *l, item_type x) {
    if (l == NULL) {
        return(NULL);
    }

    if (l->item == x) {
        return(l);
    } else {
        return(search_list(l->next, x));
    }
}
```

Insertion into a List

- Since we have no need to maintain the list in any particular order, we might as well insert each new item at the head.

```
void insert_list(list **l, item_type x) {
     list *p;      /* temporary pointer */

     p = malloc(sizeof(list));
     p->item = x;
     p->next = *l;
     *l = p;
}
```

Note the `**l`, since the head element of the list changes.

Deleting from a List: Find Predecessor

```
list *item_ahead(list *l, list *x) {
     if ((l == NULL) || (l->next == NULL)) {
          return(NULL);
     }

     if ((l->next) == x) {
          return(l);
     } else {
          return(item_ahead(l->next, x));
     }
}
```

Deleting from a List: Remove Item

```
void delete_list(list **l, list **x) {
    list *p;                /* item pointer */
    list *pred;             /* predecessor pointer */

    p = *l;
    pred = item_ahead(*l, *x);

    if (pred == NULL) { /* splice out of list */
        *l = p->next;
    } else {
        pred->next = (*x)->next;
    }
    free(*x);               /* free memory used by node */
}
```

Advantages of Linked Lists

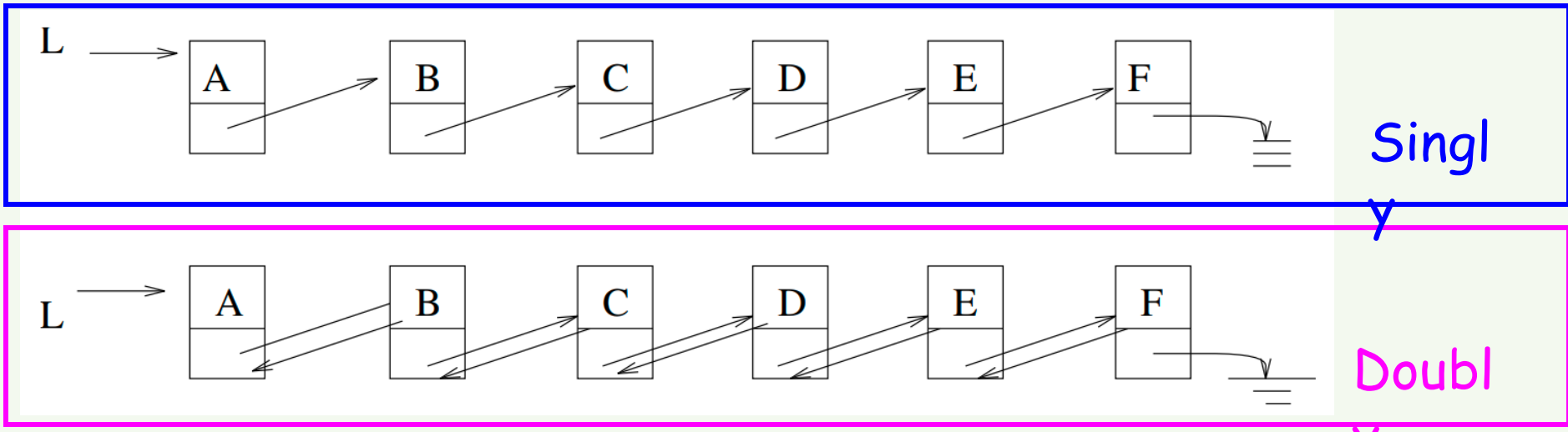
The relative advantages of linked lists over static arrays include:

1. Overflow on linked structures can never occur unless the memory is actually full.
2. Insertions and deletions are *simpler* than for contiguous (array) lists.
3. With large records, moving pointers is easier and faster than moving the items themselves.

Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

Singly or Doubly Linked Lists

We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.



What Is a Stack?

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT
- = LIFO
- It means: the last element inserted is the first one to be removed

What Is a Stack?

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT
- = LIFO
- It means: the last element inserted is the first one to be removed

- Example

Which is the first element to pick up?

What Is a Stack?

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT
- = LIFO
- It means: the last element inserted is the first one to be removed
- Example

Which is the first element to pick up?

- Applications
  - "Undo" operation
  - Call function in program
  - …

Stack ADT

- A finite ordered list with zero or more elements
- Methods
  - push() – Insert an element at one end of the stack called top.
  - pop() – Remove and return the element at the top of the stack, if it is not empty.
  - peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
  - size() – Return the number of elements in the stack.
  - isEmpty() – Return true if the stack is empty, otherwise return false.
  - isFull() – Return true if the stack is full, otherwise return false.

Implementing a Stack

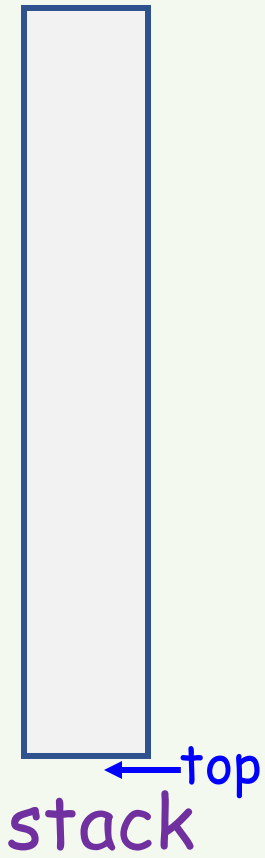
- At least two different ways to implement a stack
  - array
  - linked list

- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

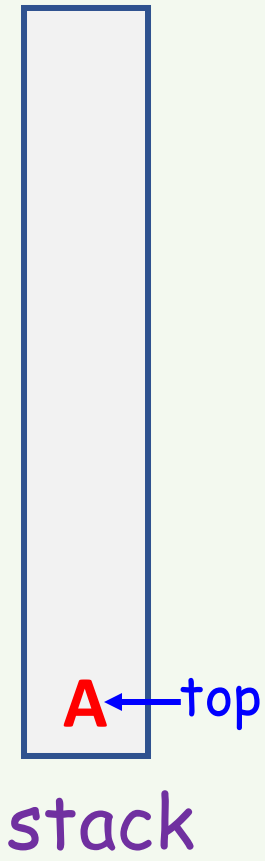
Implementing a Stack: Array

- Advantages
  - best performance
- Disadvantage
  - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false, otherwise adds it into the correct spot
  - if array is empty, pop() returns null, otherwise removes the next item in the stack

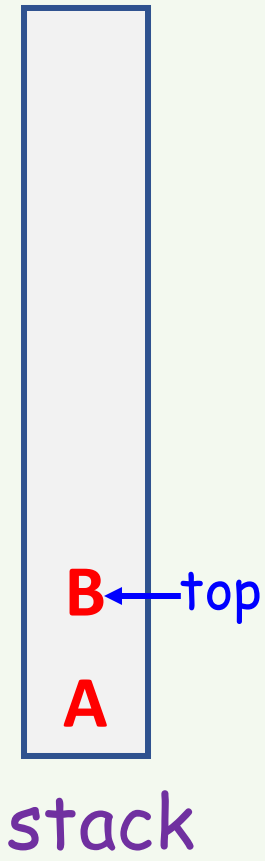
Last In First Out



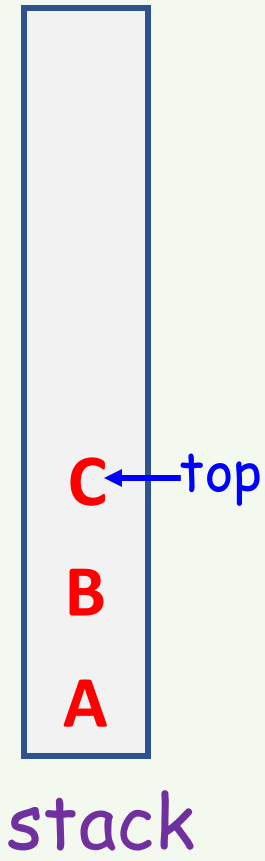
Last In First Out



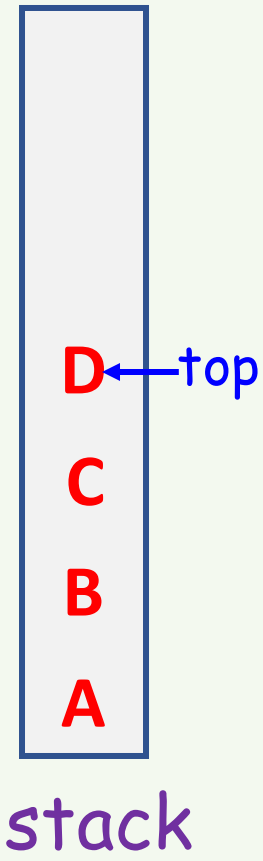
Last In First Out



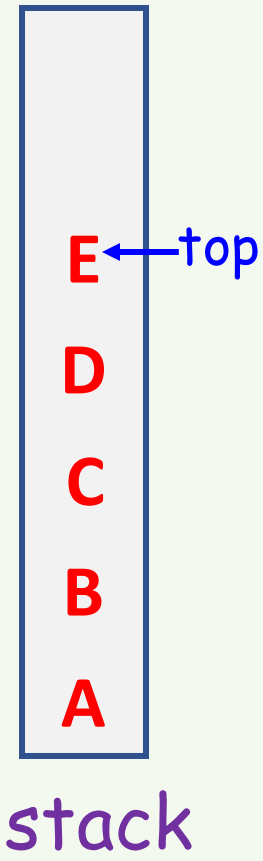
Last In First Out



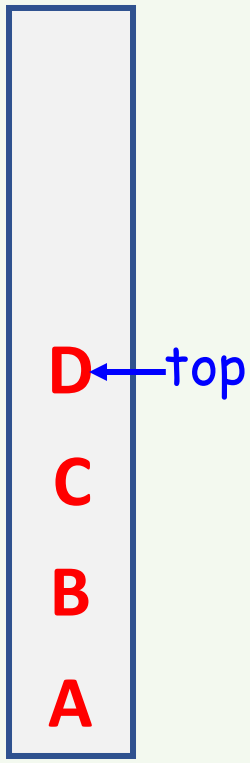
Last In First Out



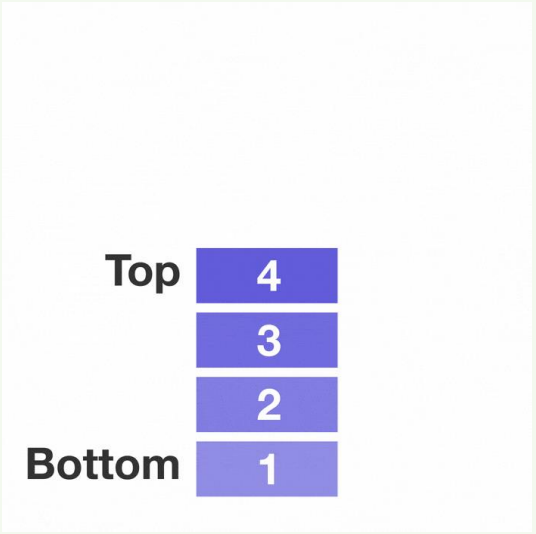
Last In First Out



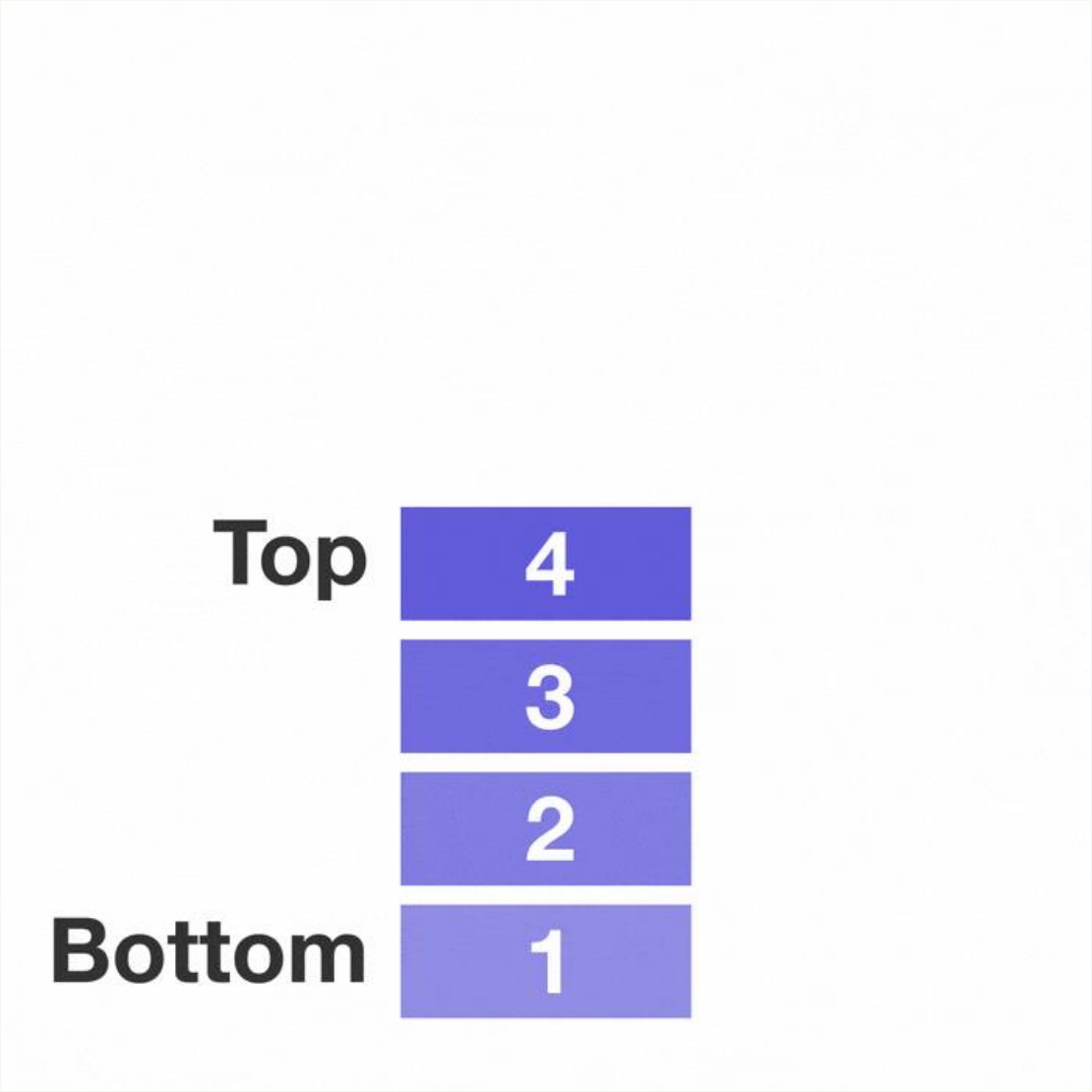
Last In First Out



stack



Last In First Out



Array-based Stack Implementation

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Associated with each stack is `top`
  - Bottom stack element stored at element 0
  - for an empty stack, set top to -1
  - last index in the array is the top
- Increment `top` when one element is pushed, decrement it after pop
  - Push
    - (1) Increment `top` by 1
    - (2) Set Stack[`top`]=x
  - Pop
    - (1) Set return value to Stack[`top`]
    - (2) Decrement `top` by 1.

Stack Class

```
class Stack {
public:
    Stack(int size = 10);                    // constructor
    ~Stack() { delete [] values; }           // destructor
    bool IsEmpty() { return top == -1; }
    bool IsFull() { return top == maxTop; } double
    Top();
    void Push(const double x);
    double Pop();
    void DisplayStack();
private:
    int maxTop;  // max stack size = size - 1 int top;
                                  // current top of stack  double*
    values;          // element array
};
```

Create Stack

- The constructor of `Stack`
  - Allocate a stack array of `size`. By default, `size` = 10
  - When the stack is full, `top` will have its maximum value, i.e., `size` - 1
  - Initially `top` is set to -1. It means the stack is empty

```
Stack::Stack(int size /*= 10*/) {
    maxTop      =     size - 1;
    values      =     new double[size];
    top         =     -1;
}
```

Although the constructor dynamically allocates the stack array, the stack is still static. The size is fixed after the initialization.

Push Stack

- `void Push(const double x)`
  - Push an element onto the stack
  - If the stack is full, print the error information
  - Note `top` always represents the index of the top element. After pushing an element, top increased by 1

```
void Stack::Push(const double x) {
   if (IsFull())
         cout << "Error: the stack is full." << endl;
   else
         values[++top] = x;
}
```

Pop Stack

- `double Pop()`
  - Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Don't forgot to decrement `top`

```
double Stack::Pop() {
    if (IsEmpty()) {
            cout << "Error: the stack is empty." << endl;
            return -1;
    } else {
            return values[top--];
    }
}
```

Remaining Methods (array based)

```
double Stack::isEmpty() {
    return top == -1;
}
```

```
double Stack::isFull() {
    return top == maxTop;
}
```

Algorithm Analysis

- `Push` $O(?)$
- `Pop` $O(?)$
- `isEmpty` $O(?)$
- `isFull` $O(?)$

Implementing a Stack: Linked List

- Advantages
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantage
  - the common case is slower
  - can grow to an infinite size
- Basic implementation
  - `list` is initially empty
  - `push()` method adds a new item to the head of the list
  - `pop()` method removes the head of the `list`

List-based Stack Implementation: Push

```
void push(element item)
{
   /* add an element to the top of the stack */
   pnode temp = (pnode) malloc (sizeof (node));
   if (IS_FULL()) {
      fprintf(stderr, " The memory is full\n");
      exit(1);
   }
   temp->item  = item;
   temp->next  = top;
   top         = temp;
}
```

Pop

```
element pop(pnode top) {
   /* delete an element from the stack */
   pnode temp = top;
   element item;
   if (IS_EMPTY(temp))  {
      fprintf(stderr,   "The stack is empty\n");
      exit(1);
   }
   item = temp->item;
   top  = temp->next;
   free(temp);
   return item;
}
```

Algorithm Analysis

- `Push` $O(?)$
- `Pop` $O(?)$
- `isEmpty` $O(?)$

Applications

- Balancing Symbols
- Evaluation of Postfix Expressions
- Infix to Postfix conversion

Balancing Symbols

- To check that every right brace, bracket, and parentheses must correspond to its left counterpart
  - e.g., [( )] is legal, but [( ]) is illegal

A stack is useful for checking symbol balance. When a closing symbol is found it must match the most recent opening symbol of the same type.

Applicable to checking html and xml tags!

Balancing Symbols

- Algorithm

  (1) Make an empty stack

  (2) Read characters until end of file

  - If the character is an opening symbol, push it onto the stack
  - If it is a closing symbol, then if the stack is empty, report an error
  - Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error

  (3) At the end of file, if the stack is not empty, report an error

Mathematical Calculations

- What does 3 + 2 * 4 equal?

  2 * 4 + 3?

  (3 + 2) * 4?

Mathematical Calculations

- What does 3 + 2 * 4 equal?

  2 * 4 + 3?

  (3 + 2) * 4?

- A mathematical expression cannot simply be evaluated left to right
- The precedence of operators affects the order of operations

Infix and Postfix Expressions

- The way we use to write expressions is known as infix notation

- Postfix notation is a notation that the operands appear before their operators
- Postfix expression does not require any precedence rules

- 3 2 * 1 + is postfix of 3 * 2 + 1

Evaluation of Postfix Expressions

- Easy to do with a stack
- Given a proper postfix expression:
  - get the next token
  - if it is an operand push it onto the stack
  - else if it is an operator
    - pop the stack for the right-hand operand
    - pop the stack for the left-hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted, the result is the top (and only element) of the stack

Clicker Question

- What does the following postfix expression evaluate to?

```
6 3 2 + *
```

A. 18
B. 36
C. 24
D. 11
E. 30

Clicker Question

- What does the following postfix expression evaluate to?

```
6 3 2 + *
```

A. 18

B. 36

C. 24

D. 11

E. 30

Evaluation of Postfix Expressions

- The time to evaluate a postfix expression is $O(n)$
  - processing each element in the input consists of stack operations and thus takes constant time
- Evaluate the following postfix expressions and write out a corresponding infix expression:

```
2 5 ^ 1 -

2 3 2 4 * + *

1 2 3 4 ^ * +

1 2 - 3 2 ^ 3 * 6 / +
```

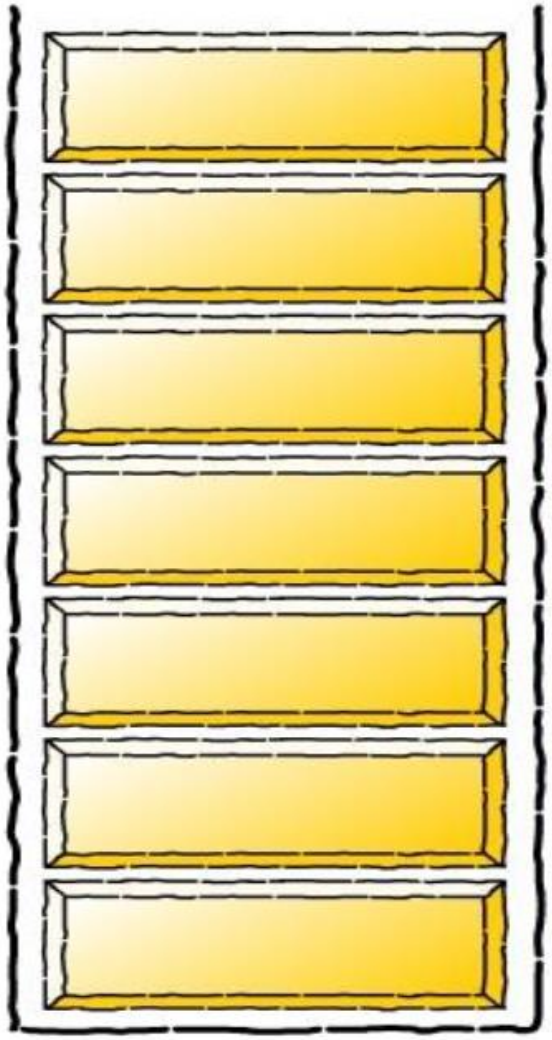
Infix to Postfix Conversion

Requires operator precedence parsing algorithm

- Parse: To determine the syntactic structure of a sentence or other utterance
- Operands: add to the output expression
- Open parenthesis: push onto stack
- Close parenthesis: pop stack symbols until an open parenthesis appears
- Operators:
  - Compare on stack and off stack precedence, except open parenthesis
  - Pop all stack symbols until a symbol of lower precedence appears. Then push the operator
- End of input: Pop all remaining stack symbols and add to the output expression

Infix to Postfix Conversion

stack

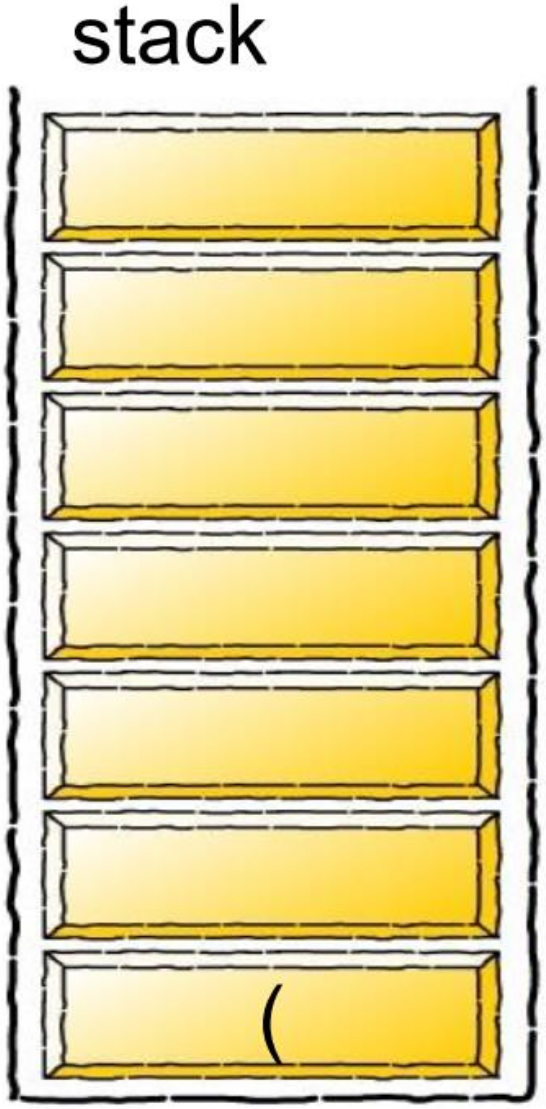


infix expression

(a + b - c) * d – (e + f)

postfix expression

Infix to Postfix Conversion

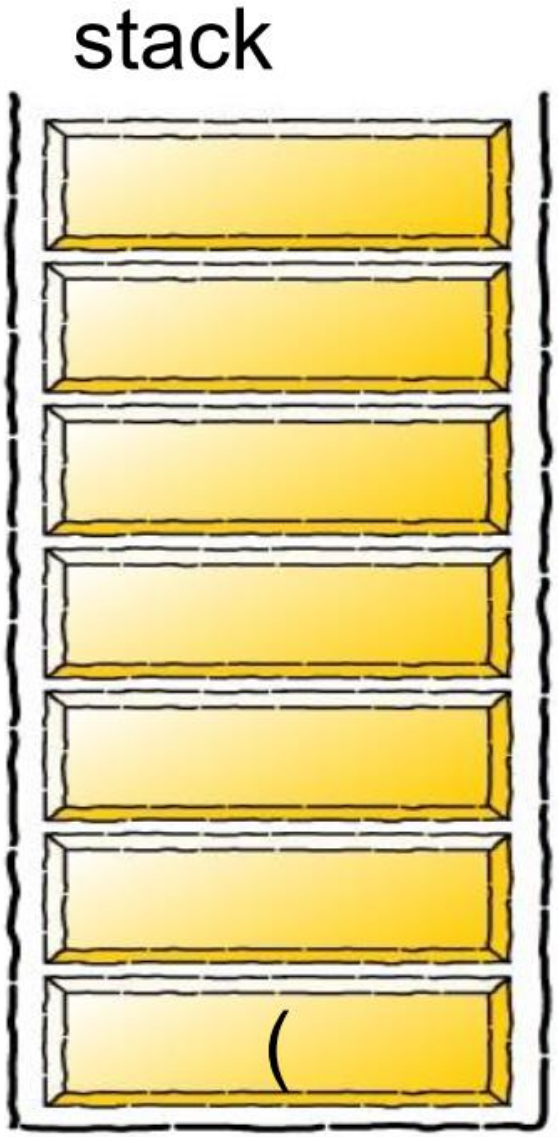


infix expression

a + b - c) * d – (e + f)

postfix expression

Infix to Postfix Conversion



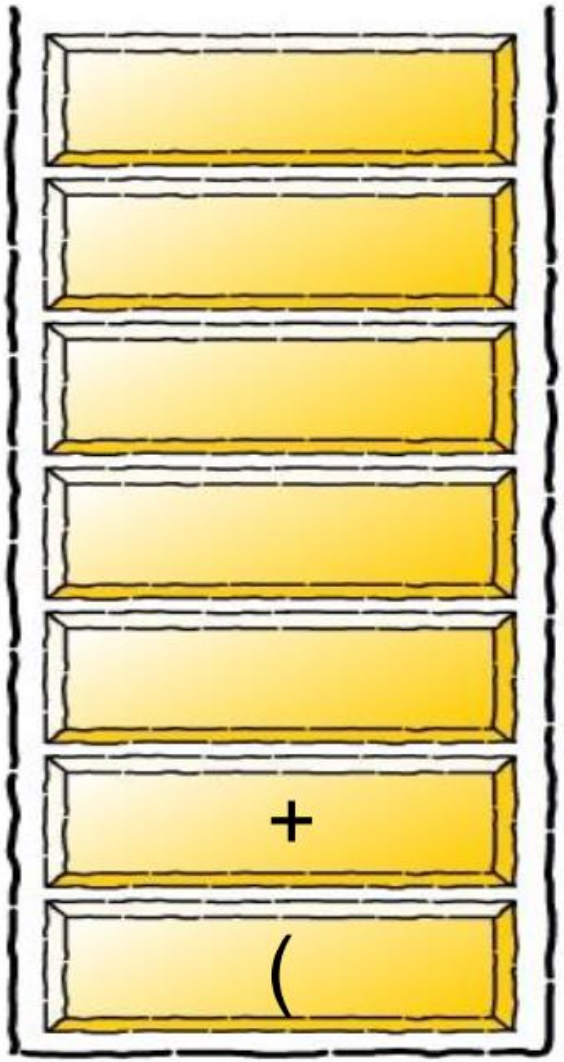
infix expression

+ b - c) * d – (e + f)

postfix expression

a

Infix to Postfix Conversion



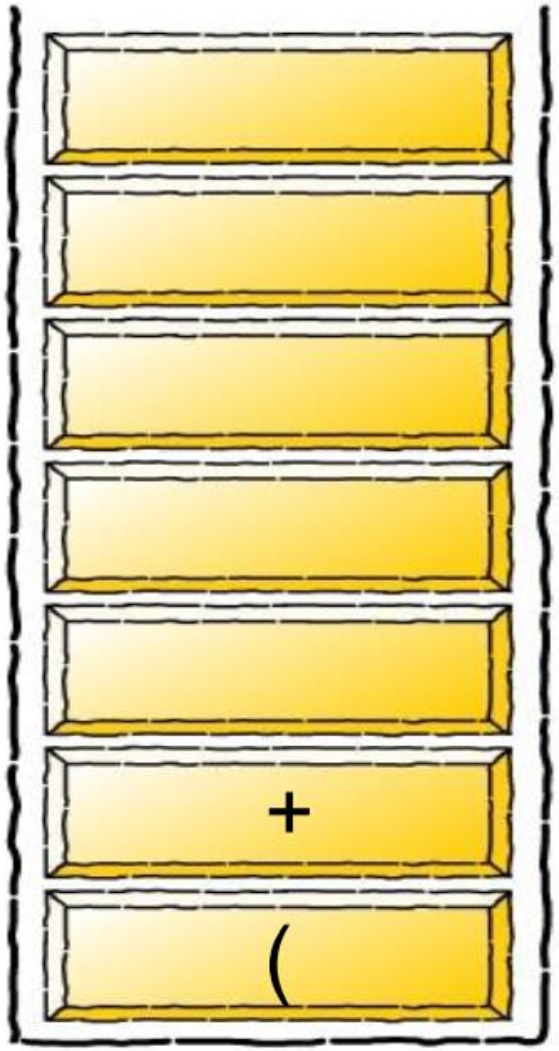
infix expression

b - c) * d – (e + f)

postfix expression

a

Infix to Postfix Conversion



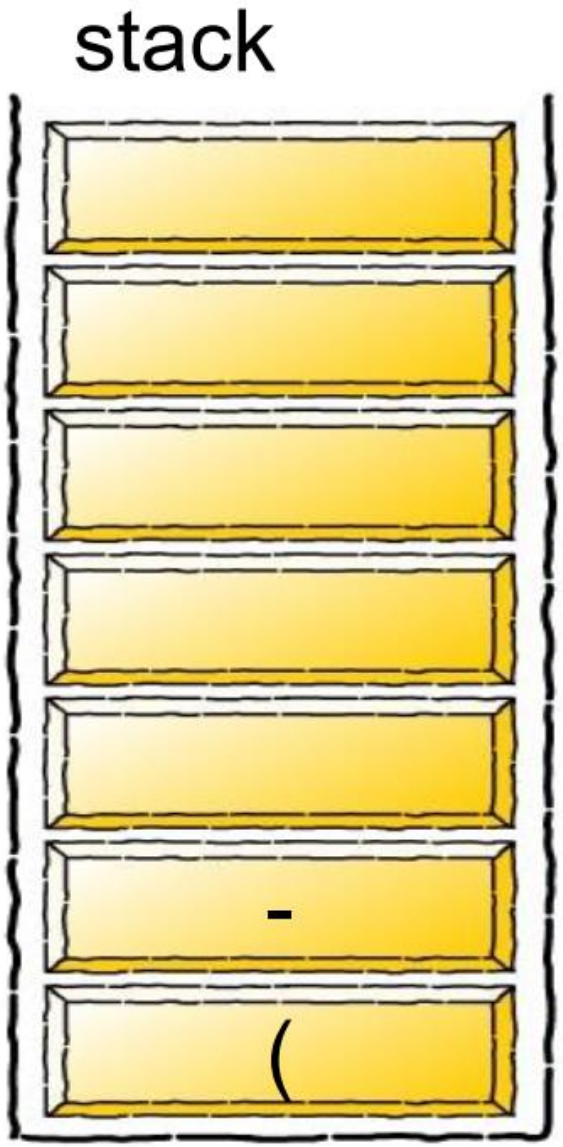
infix expression

- c) * d – (e + f)

postfix expression

a b

Infix to Postfix Conversion



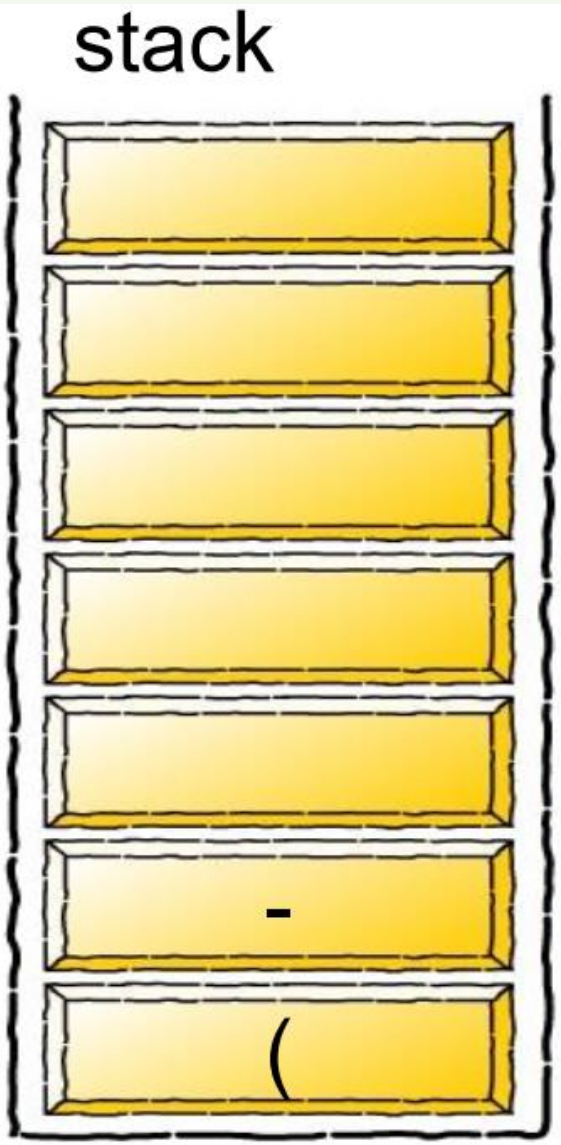
infix expression

c) * d – (e + f)

postfix expression

a b +

Infix to Postfix Conversion



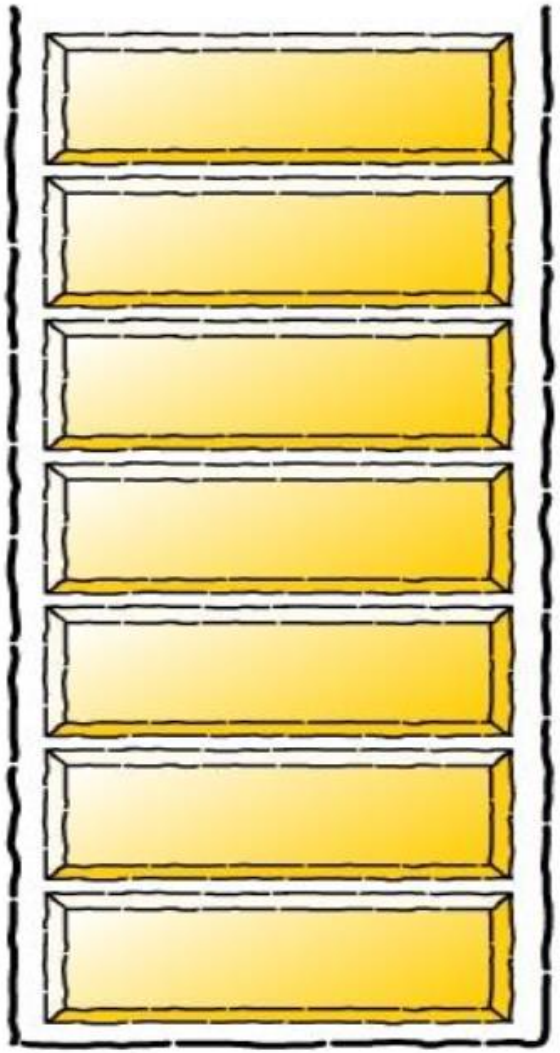
infix expression

) * d – (e + f)

postfix expression

a b + c

Infix to Postfix Conversion



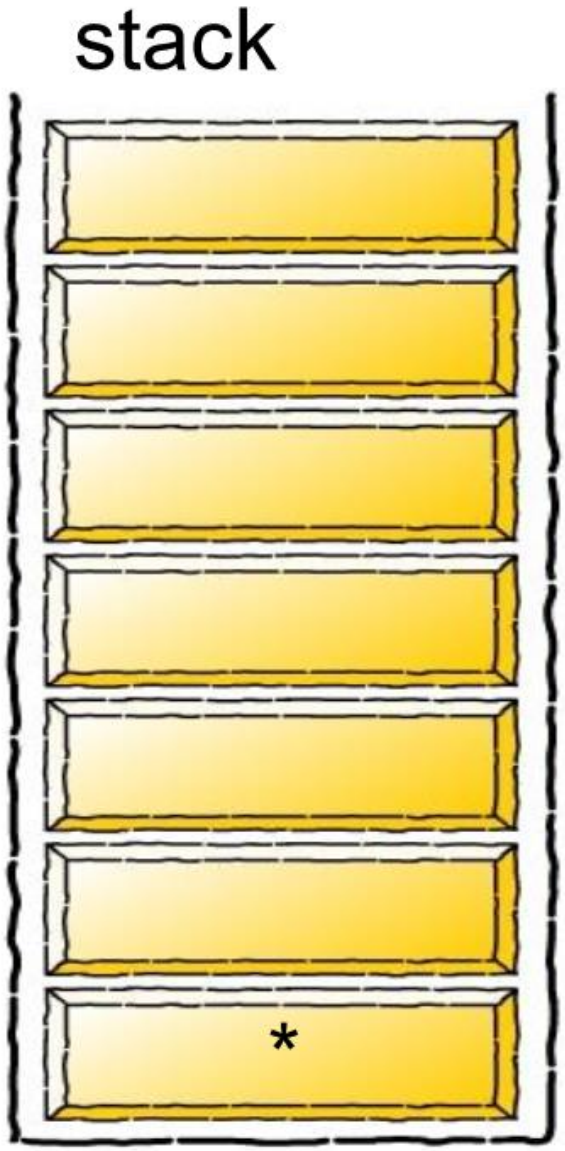
infix expression

* d – (e + f)

postfix expression

a b + c –

Infix to Postfix Conversion



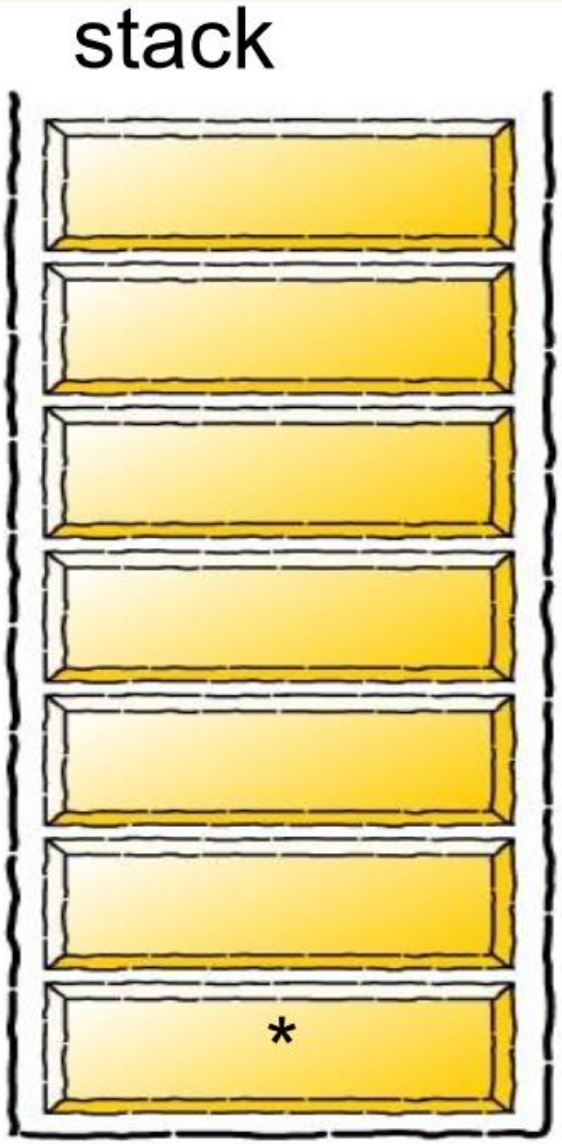
infix expression

d – (e + f)

postfix expression

a b + c –

Infix to Postfix Conversion



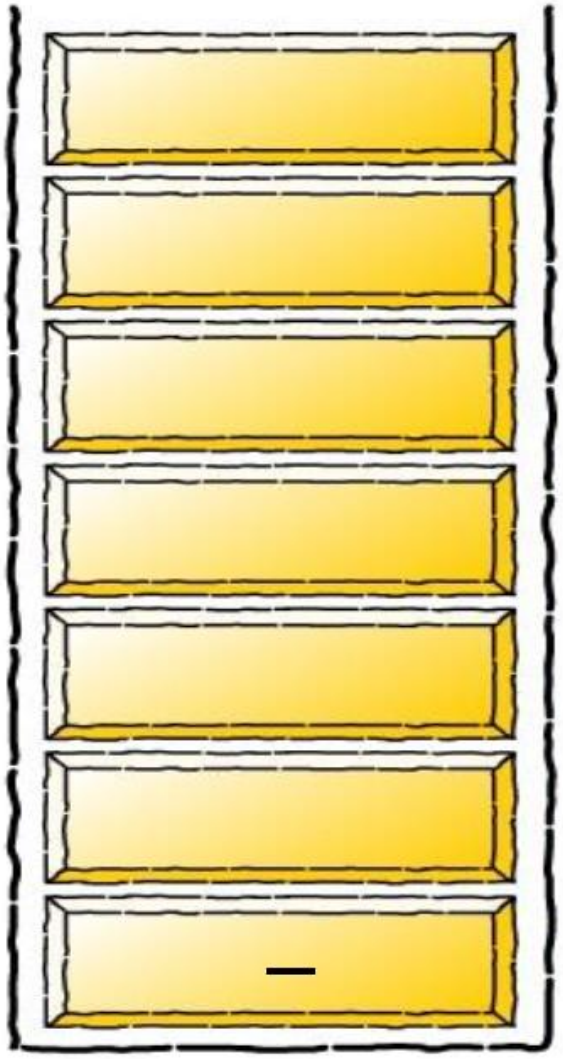
infix expression

– (e + f)

postfix expression

a b + c – d

Infix to Postfix Conversion



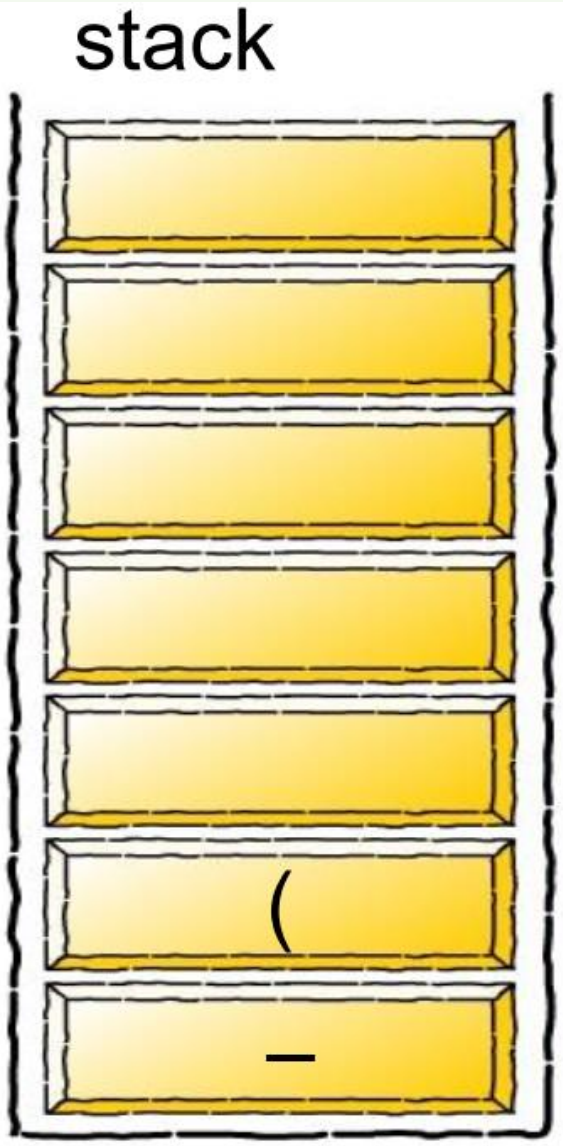
infix expression

(e + f)

postfix expression

a b + c – d *

Infix to Postfix Conversion



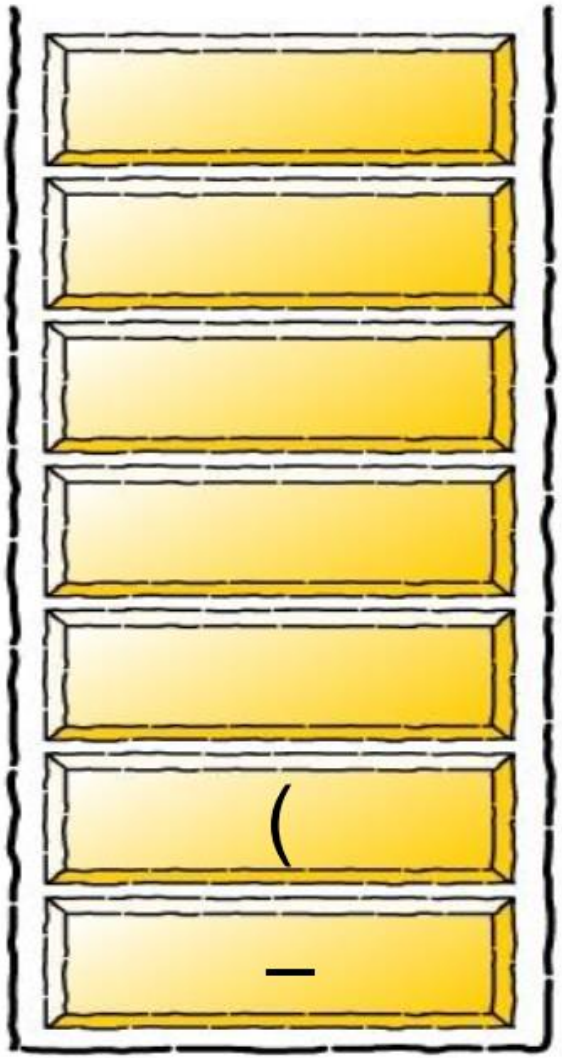
infix expression

e + f)

postfix expression

a b + c – d *

Infix to Postfix Conversion



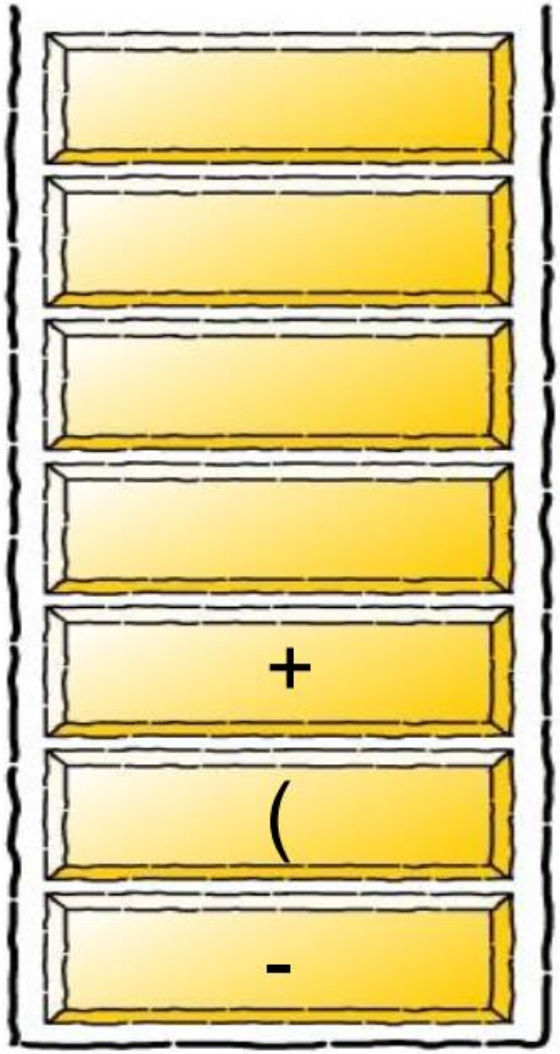
infix expression

+ f)

postfix expression

a b + c – d * e

Infix to Postfix Conversion



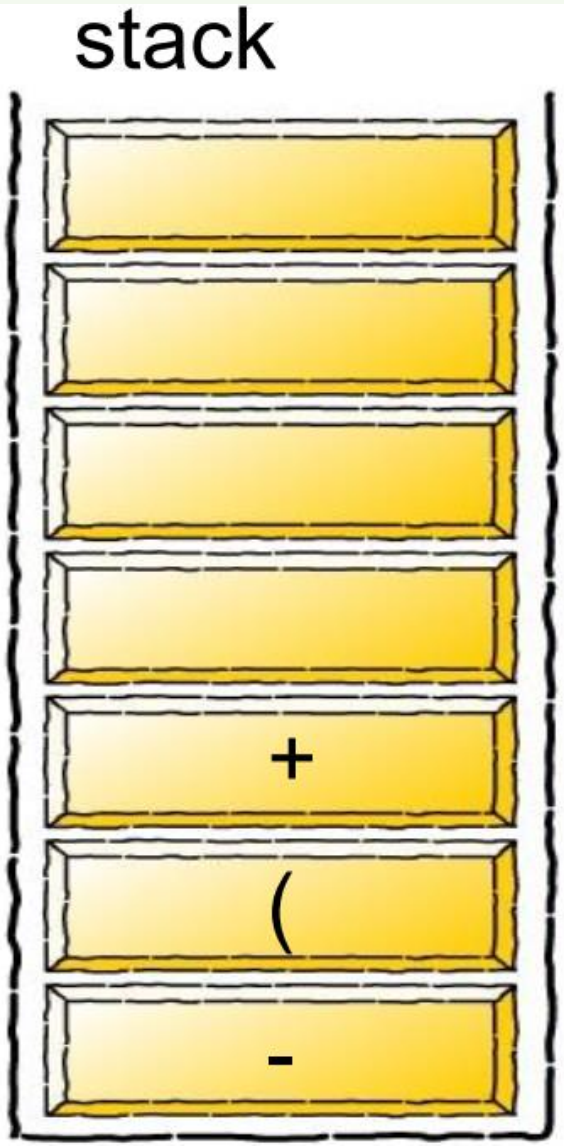
infix expression

f)

postfix expression

a b + c – d * e

Infix to Postfix Conversion



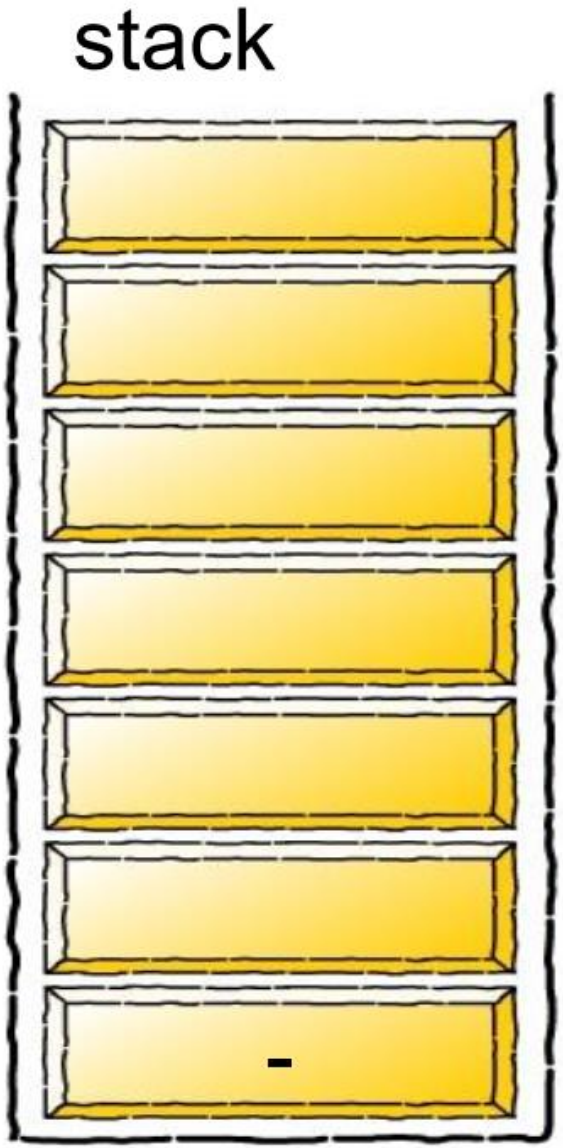
infix expression

)

postfix expression

a b + c – d * e f

Infix to Postfix Conversion

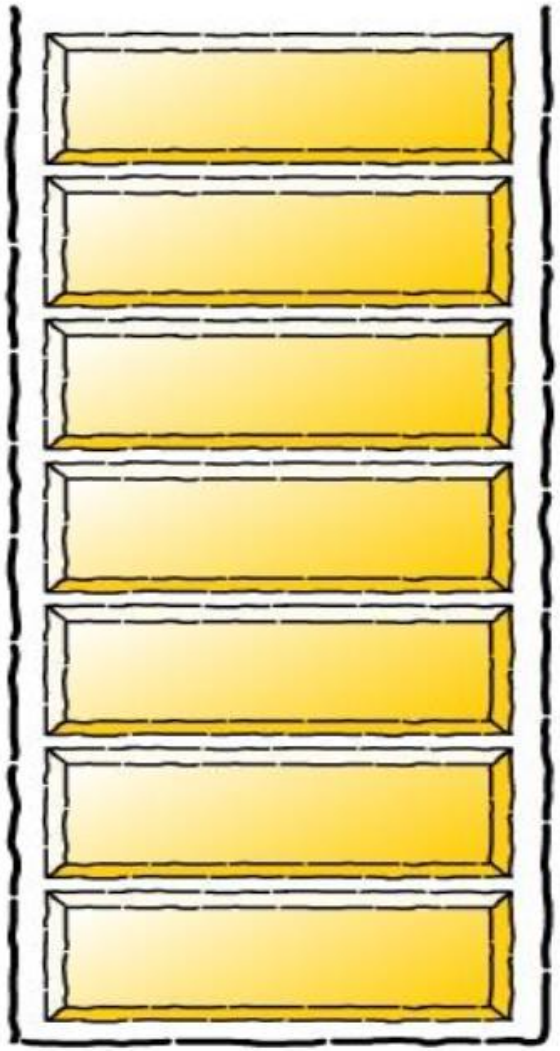


infix expression

postfix expression

a b + c – d * e f +

Infix to Postfix Conversion



infix expression

postfix expression

a b + c – d * e f + –

Infix to Postfix

- Convert the following equations from infix to postfix:

```
(2 ^ 3) ^ 3 + 5 * 1
```

```
1 + 2 - 1 * 3 / 3 + 2 ^ 2 / 3
```

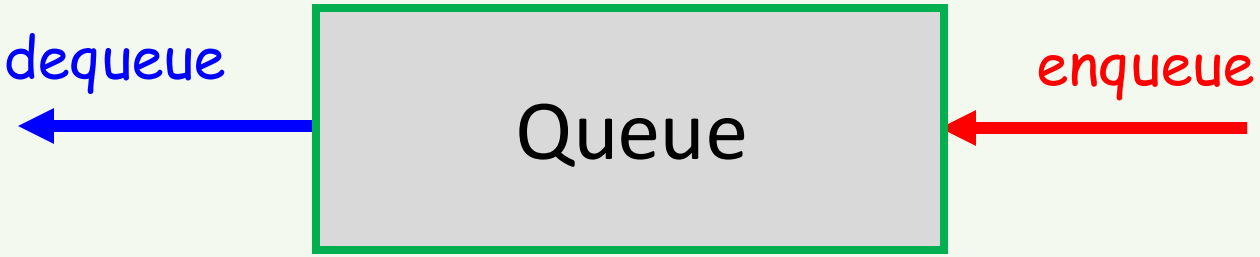
Queue Overview

- Queue ADT
- Basic operations of queue
  - Enqueuing, dequeuing, etc
- Implementation of queue
  - Array
  - Linked list

The Queue ADT

- Another form of restricted list
  - Insertion is done at one end, whereas deletion is performed at the other end

- Basic operations of queue
  - enqueue: insert an element at the rear of the list
  - dequeue: delete the element at the front of the list



- First-in First-out (FIFO) list

Queue Applications

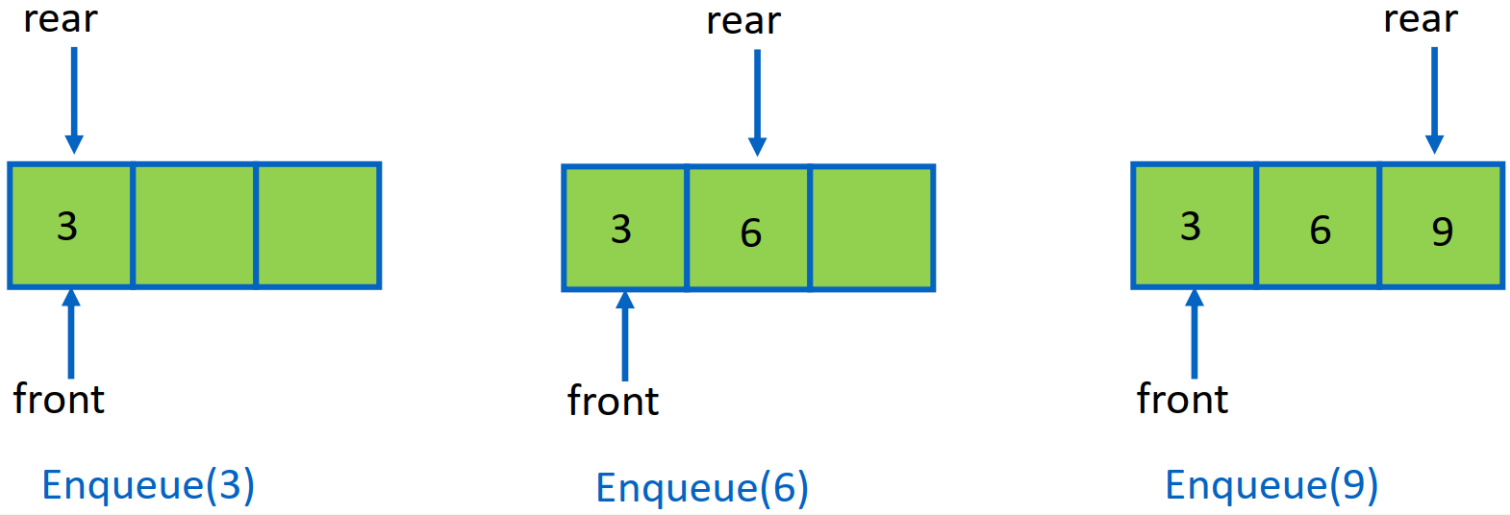
- Real life examples
  - Waiting in line
  - Waiting on hold for tech support

- Applications related to Computer Science
  - Threads
  - Job scheduling (e.g., Round-Robin algorithm for CPU allocation)

Queue Implementation of Array

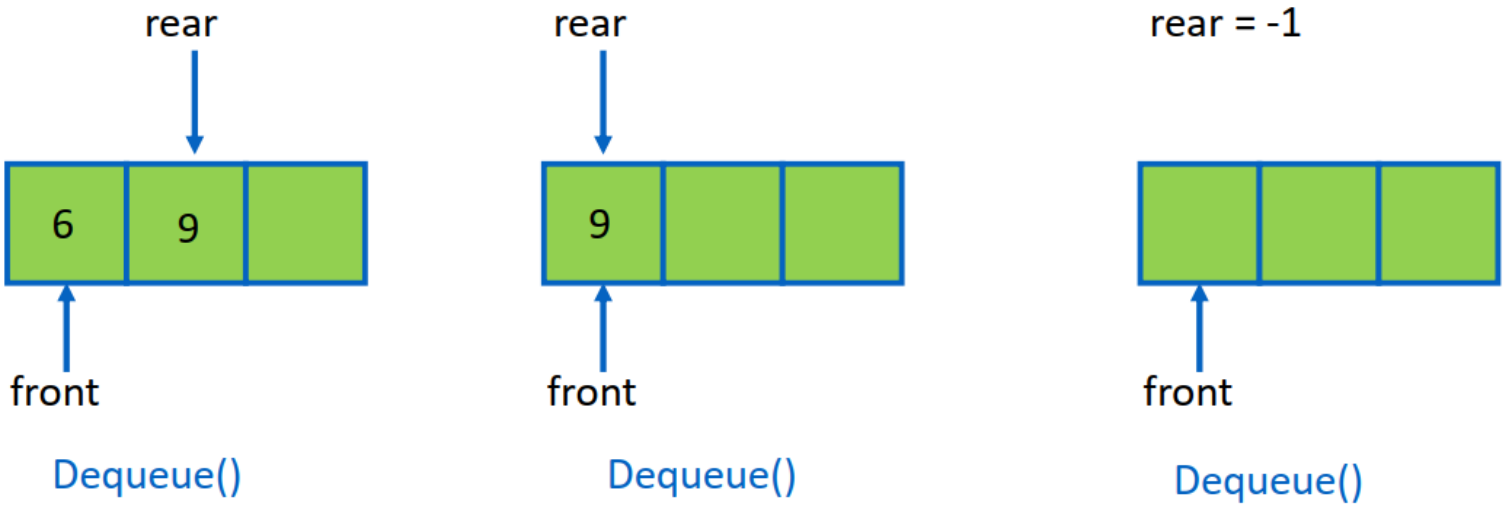
- There are several different algorithms to implement Enqueue and Dequeue

- Naïve way
  - When enqueuing, the front index is always fixed, and the rear index moves forward in the array



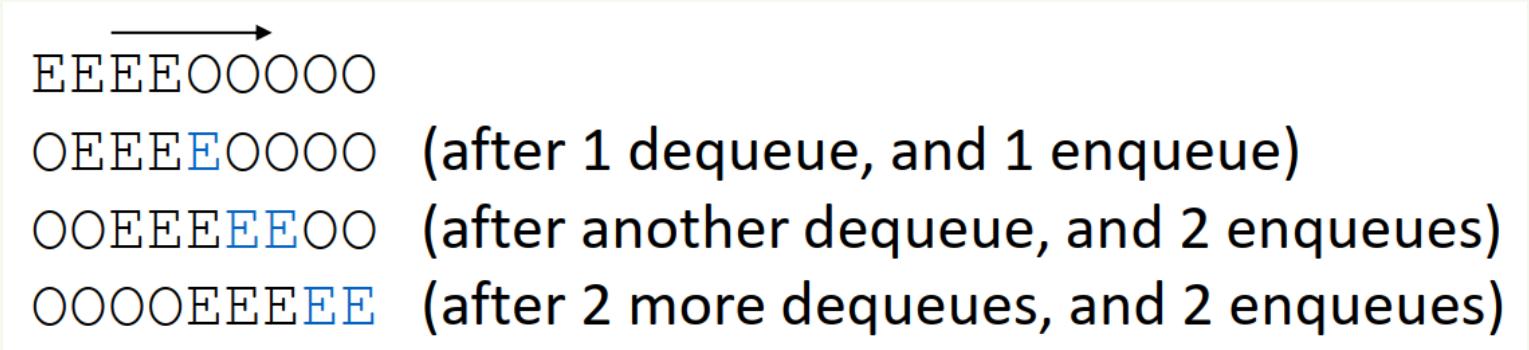
Queue Implementation of Array

- Naïve way
  - When enqueuing, the front index is always fixed, and the rear index moves forward in the array
  - When dequeuing, the element at the front the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Queue Implementation of Array

- Better way
  - When an item is enqueued, make the rear index move forward
  - When an item is dequeued, the front index moves by one element towards the back of the queue (thus removing the front item, so no copying to neighboring elements is needed)

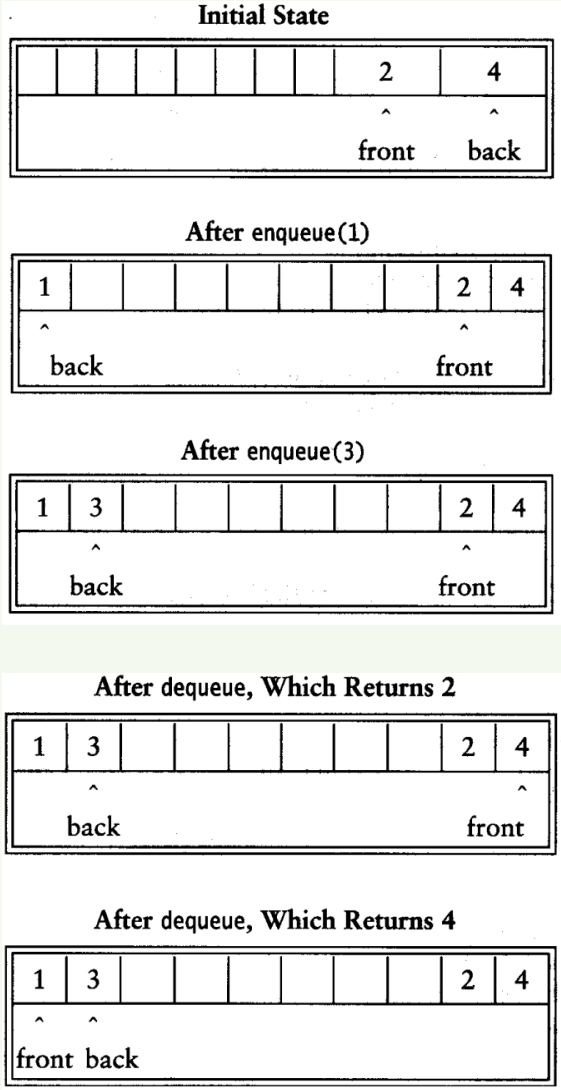


The problem here is that the rear index cannot move beyond the last element in the array.

Implementation using Circular Array

- Using a circular array
- When an element moves past the end of a circular array, it wraps around to the beginning, e.g.,
  - OOOOO7963 -> 4OOOO7963 (after Enqueue(4))
  - After Enqueue(4), the rear index moves to 0

Implementation using Circular Array



Empty or Full?

- Empty queue
  - rear = front - 1
- Full queue?
  - the same!
  - Reason: n values to represent n + 1 states
- Solutions
  - Use a boolean variable to say explicitly whether the queue is empty or not
  - Make the array of size n + 1 and only allow n elements to be stored
  - Use a counter of the number of elements in the queue

Queue Implementation of Circular Array

```
class Queue {
   public:
      Queue(int size = 10);              // constructor
      ~Queue() { delete [] values; }     // destructor
      bool IsEmpty(void);
      bool IsFull(void);
      bool Enqueue(double x);
      bool Dequeue(double & x);
      void DisplayQueue(void);
   private:
      int front;        // front index
      int rear;         // rear index
      int counter;      // number of elements
      int maxSize;      // size of array queue
      double* values;   // element array
};
```

Create Queue

- Allocate a queue array of `size`. By default, `size = 10`
- `front` is set to 0, pointing to the first element of the array
- `rear` is set to -1. The queue is empty initially

```
Queue::Queue(int size /* = 10 */) {
    values          =           new double[size];
    maxSize         =           size;
    front           =           0;
    rear            =           -1;
    counter         =           0;
}
```

IsEmpty & IsFull

- Since we keep track of the number of elements that are actually in the queue: `counter`, it is easy to check if the queue is empty or full

```
bool Queue::IsEmpty() {
    if (counter) return false;
    else         return true;
}
```

```
bool Queue::IsFull() {
    if (counter < maxSize)
            return false;
    else
            return true;
}
```

Enqueue

```
bool Queue::Enqueue(double x) {
    if (IsFull()) {
            cout << "Error: the queue is full." << endl;
            return false;
    }
    else {
            // calculate the new rear position (circular)
            rear = (rear + 1) % maxSize;
            // insert new item
            values[rear] = x;
            // update counter
            counter++;  return true;
    }
}
```

Dequeue

```
bool Queue::Dequeue(double & x) {
    if (IsEmpty()) {
            cout << "Error: the queue is empty." << endl;
            return false;
    }
    else {
            // retrieve the front item
            x = values[front];
            // move front
            front = (front + 1) % maxSize;
            // update counter
            counter--;
            return true;
    }
}
```

Algorithm Analysis

- `Enqueue` $O(?)$
- `Dequeue` $O(?)$
- `size` $O(?)$
- `isFull` $O(?)$
- `isEmpty` $O(?)$

List-based Queue Implementation: Enqueue

```
void enqueue(element item)
{
    /* add an element to the rear of the queue */
    pnode  temp = (pnode) malloc(sizeof (queue));
    if (IS_FULL(temp)) {
           fprintf(stderr, " The memory is full\n");
           exit(1);
    }
    temp->item = item;
    temp->next = NULL;
    if (front)    { (rear) -> next= temp;}
    else front = temp;
    rear = temp;
}
```

List-based Queue Implementation: Dequeue

```
element dequeue(pnode &front)
{
    /* delete an element from the queue */
    pnode temp = front;  element item;
    if (IS_EMPTY(front)){
          fprintf(stderr,     "The queue is empty\n");
          exit(1);
    }
    item = temp->item;
    front = temp->next;
    free(temp);
    return item;
}
```

Algorithm Analysis

- `Enqueue` $O(?)$
- `Dequeue` $O(?)$
- `size` $O(?)$
- `isFull` $O(?)$
- `isEmpty` $O(?)$