Algorithms and Their Applications
- Introduction to Data Structures -

Won-Yong Shin

March 30th, 2020

Stacks

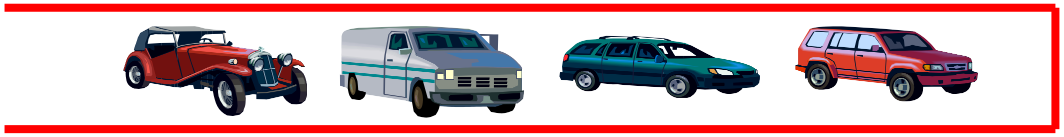
- Basic Stack Operations -

연세대학교
YONSEI UNIVERSITY

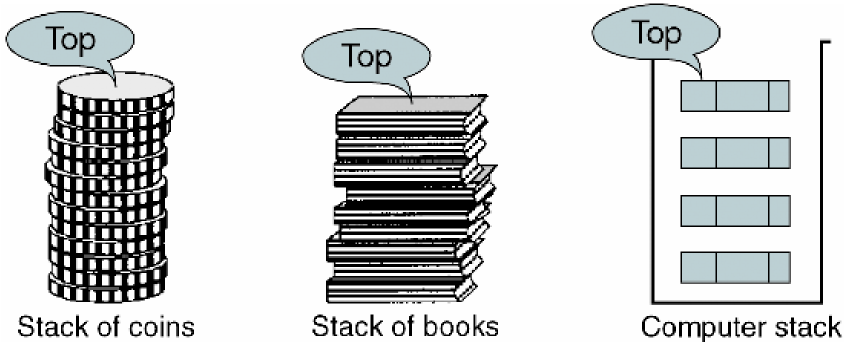
Stacks in Daily Lives

- Coin case

- Terrible parking lot

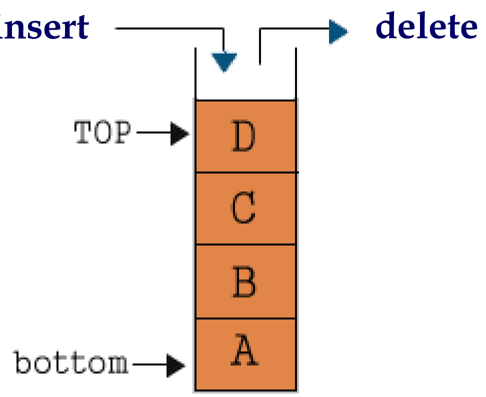


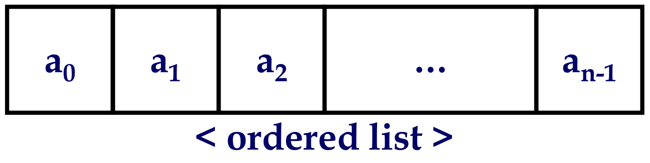
- A data structure, in which we can add or delete from only one side

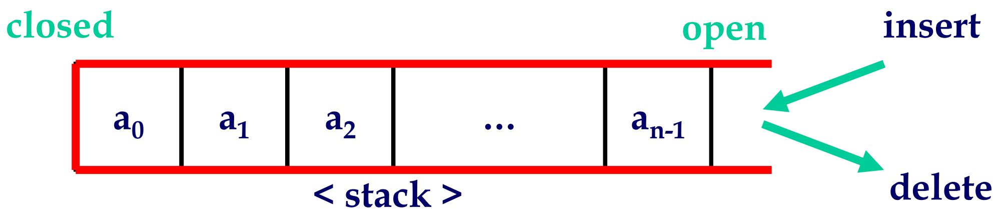


  - LIFO (Last-in first-out)

- Stack
  - A linear list in which all additions and deletions are restricted to one end, called top
    - Given a stack S = (a_0, a_1, … , a_{n-1}), $\mathbf{a_0}$ is the **bottom** element and $\mathbf{a_{n-1}}$ is the **top** element



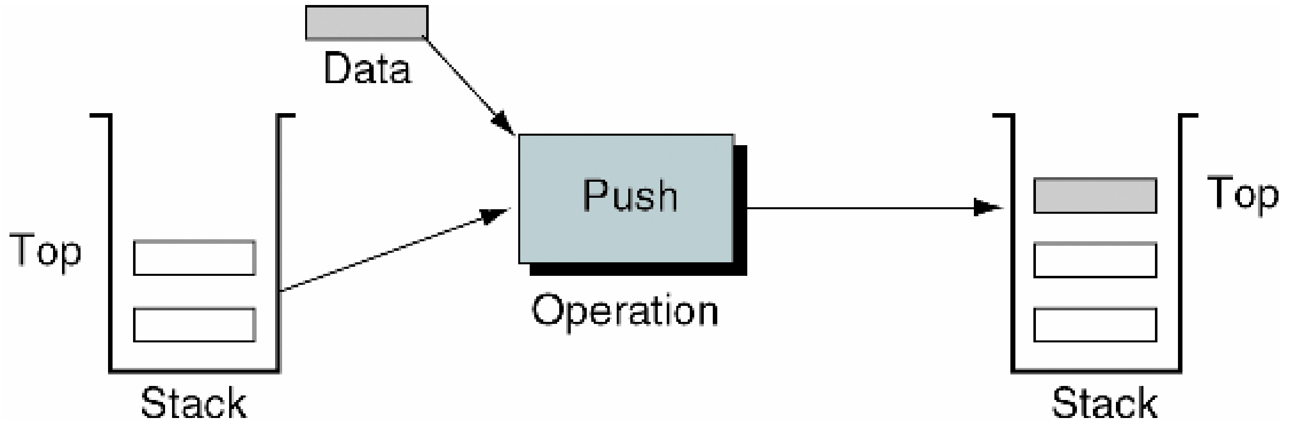




Basic Stack Operations

- Basic stack operations
  - **CreateStack**: allocate memory and initialize
  - **Push**
  - **Pop**
  - **Stack Top**
  - **DestroyStack**: remove all items and deallocate memory

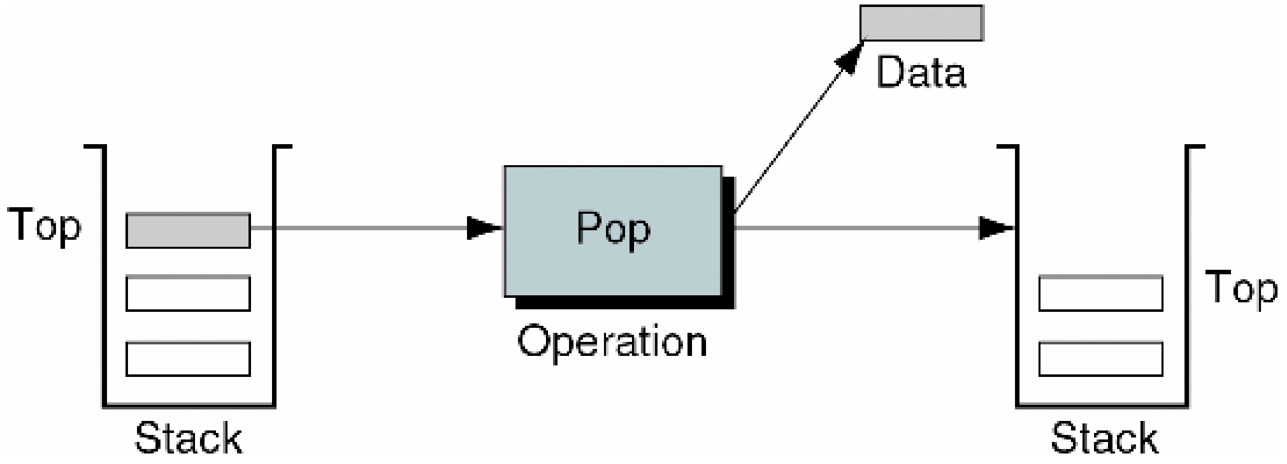
- **Push**: add an item at the top of the stack
  - *New item* becomes top



Note! Before pushing an item, we should check if stack is *full*.
Otherwise stack overflow can occur.

Pop

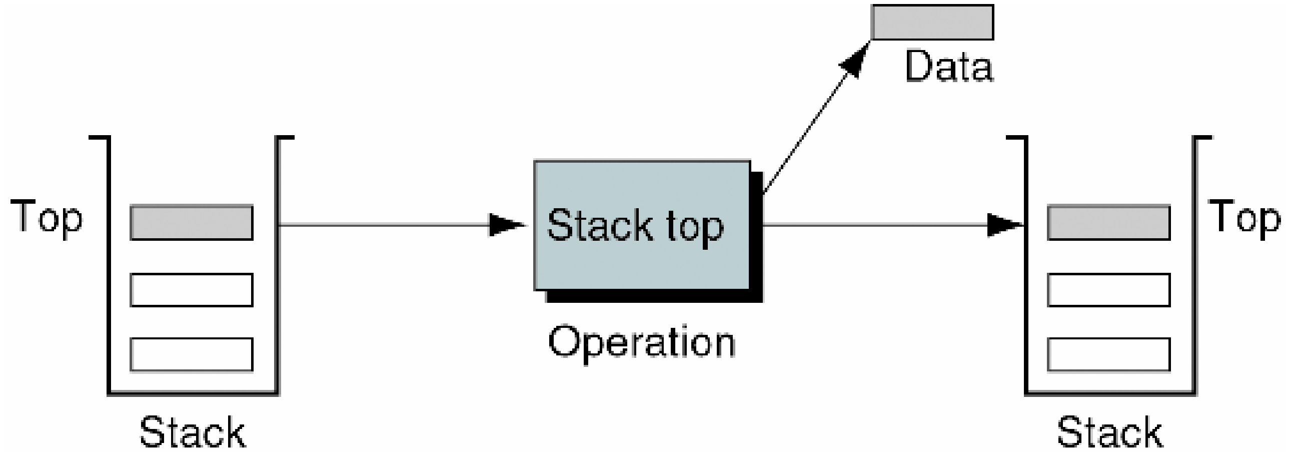
- **Pop**: remove the item at the top of the stack and return it
  - *Next older item* becomes top



Note! Before popping an item, we should check if stack is *empty*.
Otherwise stack underflow can occur.

Stack Top

- **Stack top**: return the item at the top of the stack
  - Top element is **not** deleted



Example of Stack Operations

Push(green):

Push(blue):

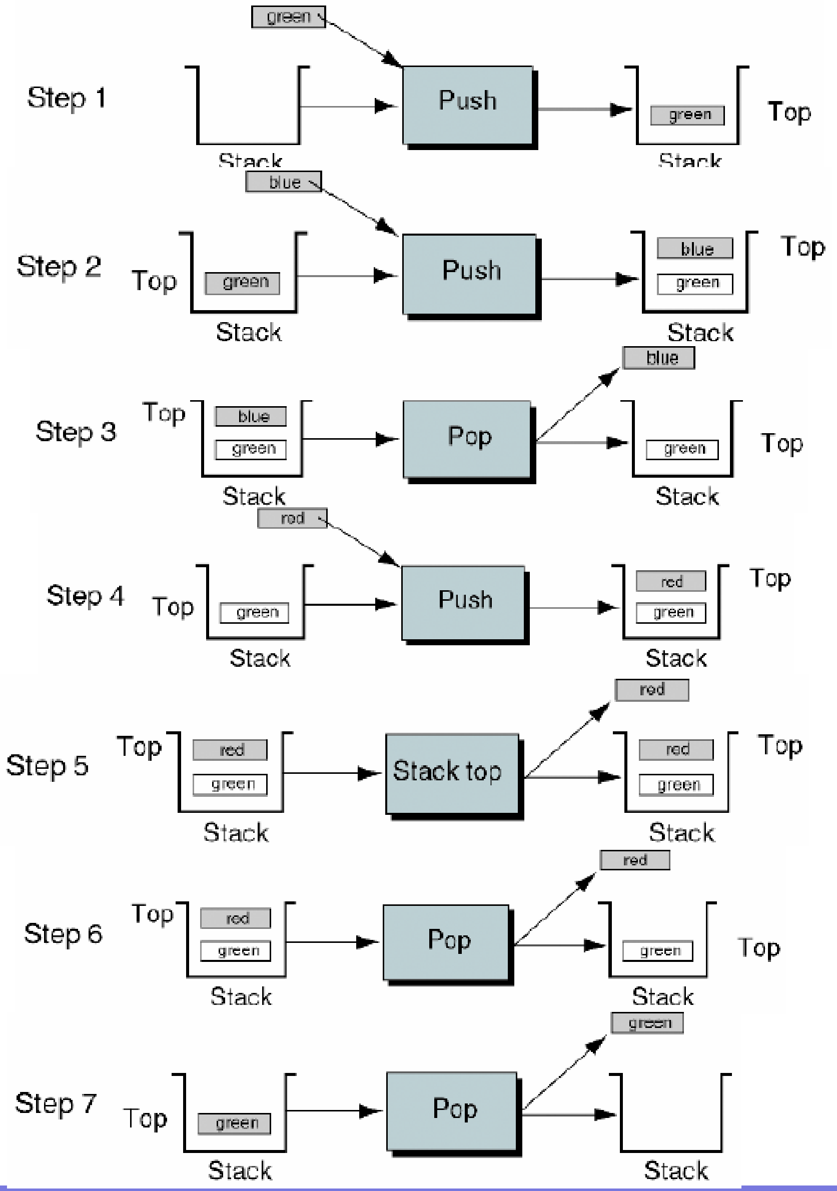
Pop( ):

Push(red):

Top( ):

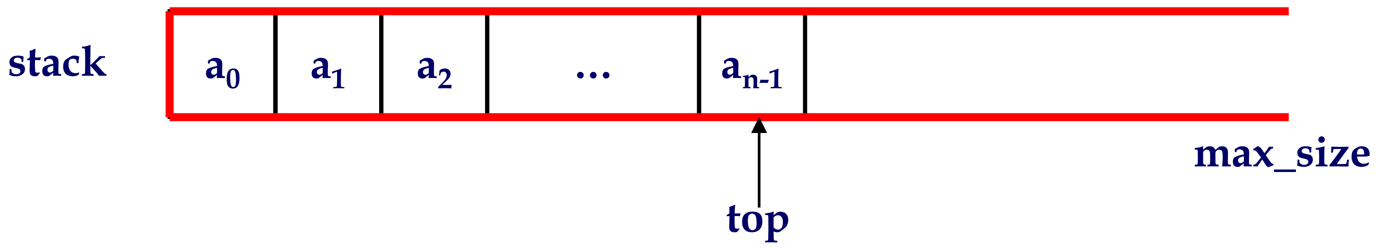
Pop( ):

Pop( ):



Array Implementation of Stack

- Data representation



Stack

```
Element *stack;      // array to store elements
int max_size;        // maximum size
int top;             // stack top
```

Cf. *Element*: an arbitrary type (ex: typedef int *Element*;)

- Linked list implementation will be shown later

Stacks

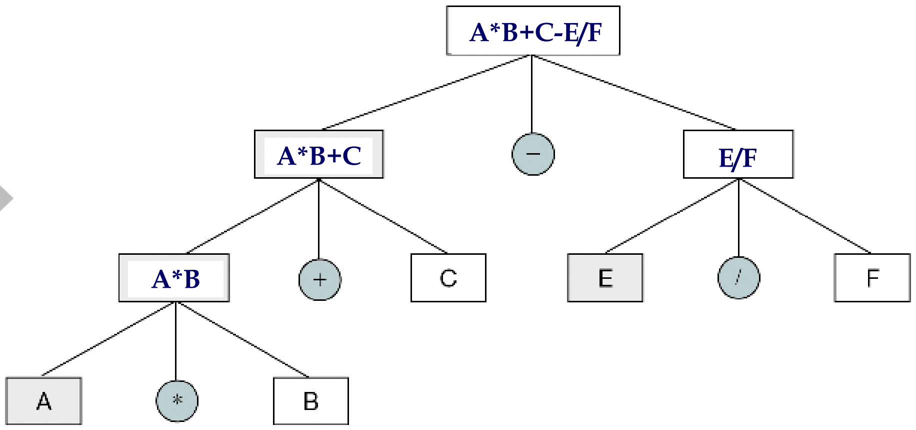
- Stack Applications -

Parsing

- Process of analyzing a sequence of *tokens* to determine its grammatical structure with respect to a given formal grammar

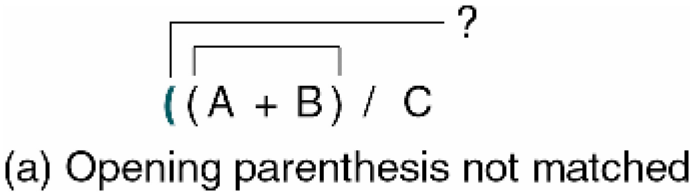
  Ex)

  A*B+C-E/F
  <input>

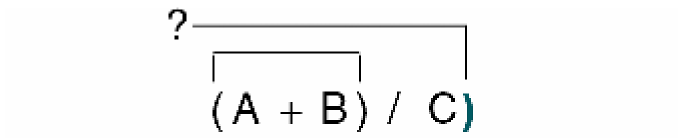


<parse tree>

- A common programming problem: **unmatched parentheses**



(a) Opening parenthesis not matched



(b) Closing parenthesis not matched

Token

- **Token**: usually a word or an atomic element within a string
  - Operator or operand

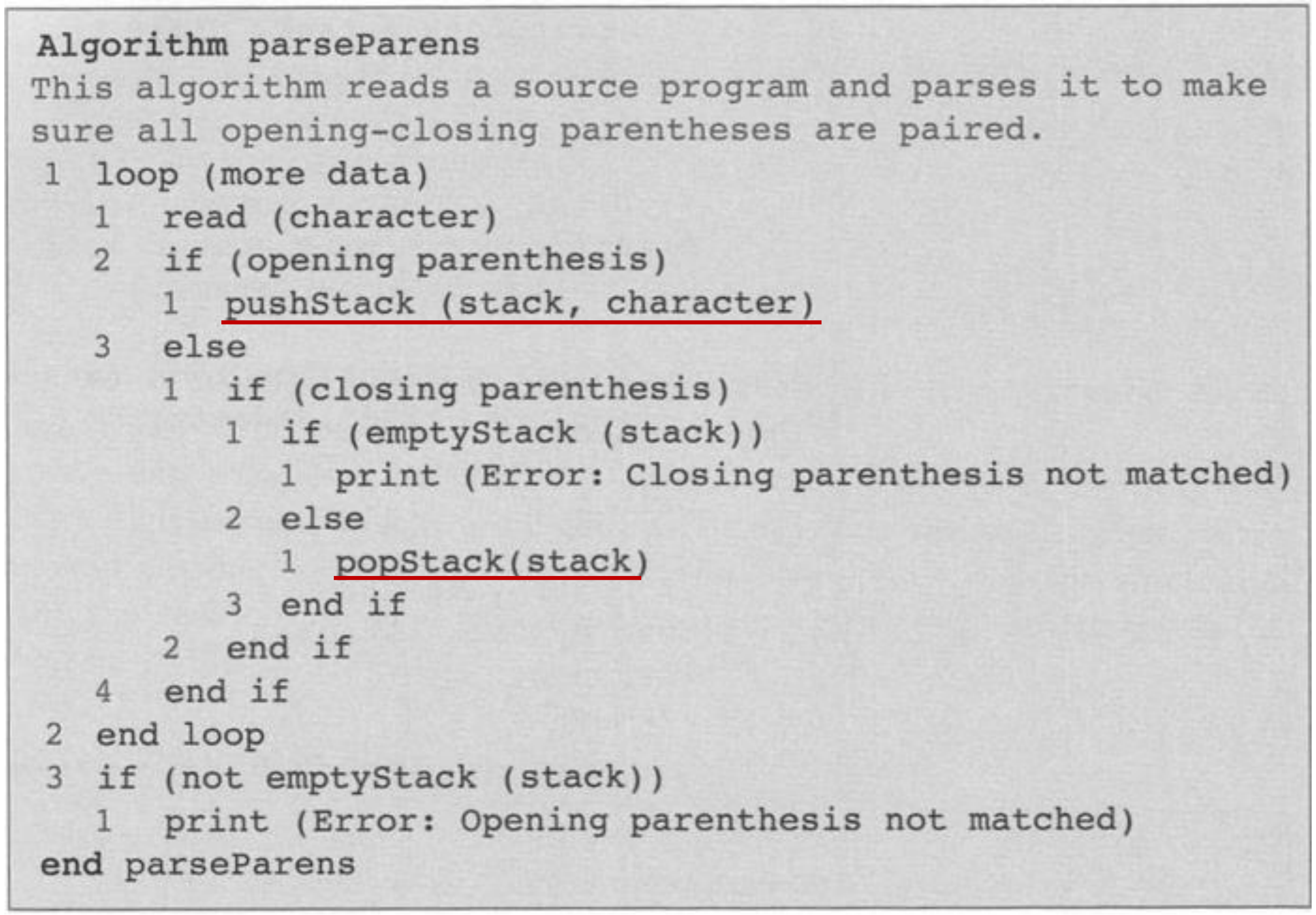
    Ex) <u>15</u> <u>+</u> <u>A</u> <u>/</u> <u>2</u>

- Tokenizing: splitting up a string of characters into a set of tokens

  Ex) 15 + A / 2 ➔ (15, +, A, /, 2)

Parsing: Algorithm

- Algorithm to detect *unmatched* parenthesis



```
Algorithm parseParens
This algorithm reads a source program and parses it to make
sure all opening-closing parentheses are paired.
1 loop (more data)
   1  read (character)
   2  if (opening parenthesis)
         1  pushStack (stack, character)
   3  else
         1  if (closing parenthesis)
               1  if (emptyStack (stack))
                     1  print (Error: Closing parenthesis not matched)
               2  else
                     1  popStack(stack)
               3  end if
         2  end if
   4  end if
2 end loop
3 if (not emptyStack (stack))
   1  print (Error: Opening parenthesis not matched)
end parseParens
```

Infix Notation vs. Postfix Notation

- Infix notation
  - Easy to understand for human
- Postfix notation
  - **No** parenthesis
  - The *operands*' order in postfix is the same as in infix
  - The order of *operators* is the same with the evaluation order

infix	postfix
2+3*4	**2 3 4 * +**
a*b+5	**a b * 5 +**
(1+2)*7	**1 2 + 7 ***
a*b/c	**a b * c /**
(a/(b-c+d))*(e-a)*c	**a b c – d + / e a - * c ***
a/b-c+d*e-a*c	**a b / c - d e * + a c * -**

Infix to Postfix

- Manual process
  1. Fully parenthesize the expression
  2. Move all binary operators to the location of their corresponding closing parentheses
  3. Delete all parentheses

Example)

a/b-c+d*e-a*c

→ `((((a/b)-c)+(d*e))-(a*c))`

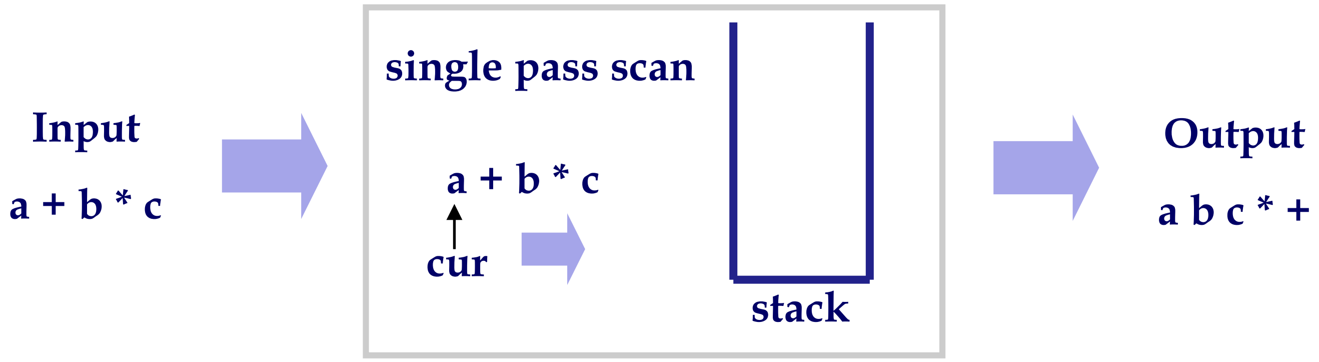
→ `ab/c-de*+ac*-`

Evaluation of Expression

- Procedure
  1. Transform infix notation to postfix notation
  2. Evaluate postfix notation using a **stack**

- Problems of inefficiency
  - Evaluation of postfix notation is already simple
  - Infix to postfix transformation requires (at least) two passes

    → Let's design one pass algorithm to transform infix to postfix



Observations

- Two examples

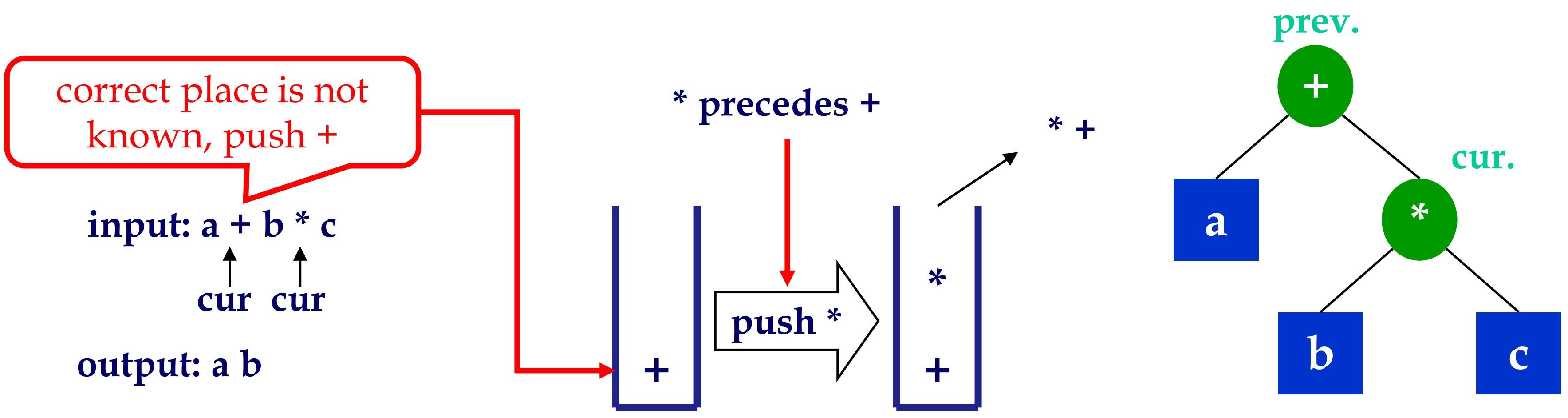
  ex) a + b * c → (a) (b c *) +

  a * b + c → (a b *) (c) +

- Clues
  - The **order of operands** is the same in infix and postfix
    - Just output operands without any rearrangement
  - The **order of operators** in postfix depends on the *precedence* of operators
    - 1. If an operator's priority is higher than the operator at the top of the stack, push it into the stack
    - 2. If the operator at the top of the stack has a higher priority than the current operator, it is popped and placed in the output expression

Infix To Postfix Using Stack (1/2)

- If correct place of current operator is **not known** yet, **push** current operator
  - Current operator (operands are not concluded)
  - If there is previous operator in stack, compare its *precedence* with that of current operator

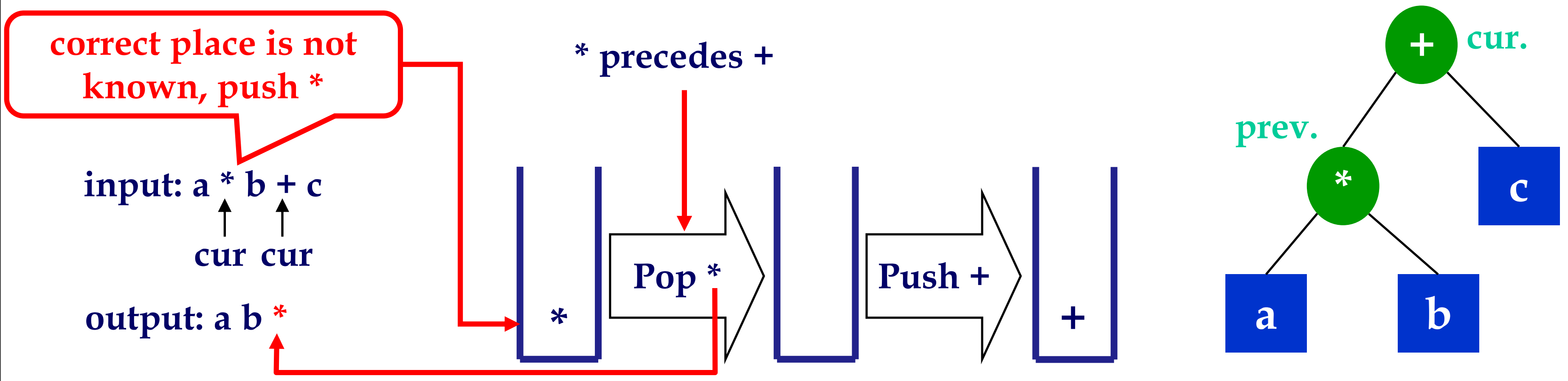


Infix To Postfix Using Stack (2/2)

- If correct place of an operator is **known**, **print** it
  - Scan pointer reaches end of string

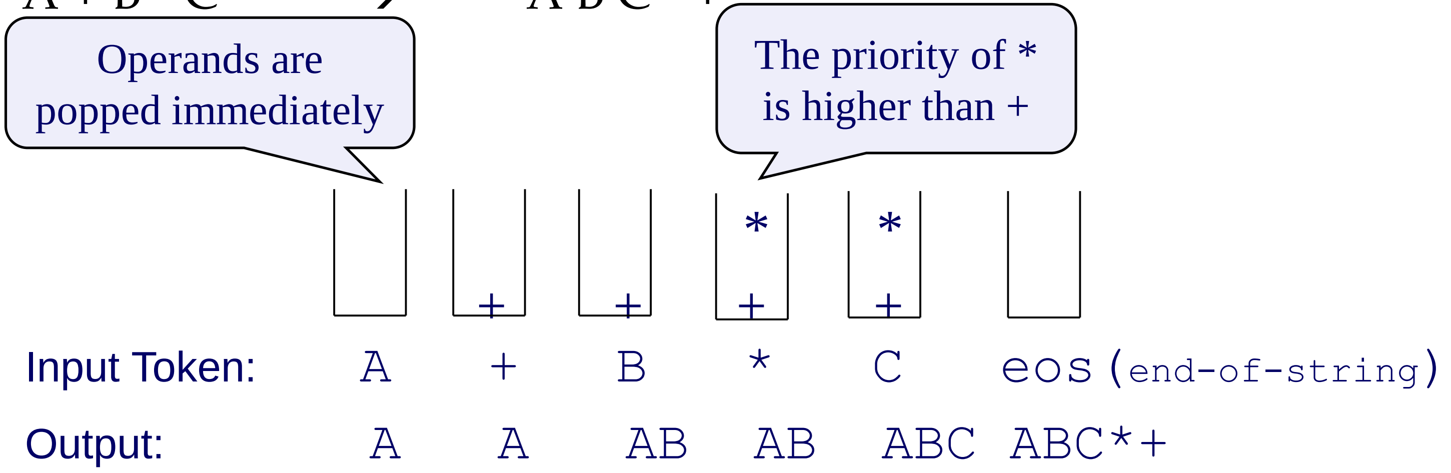
    → Places of all saved operators are determined
  - There is previous operator in stack, whose precedence is higher than current operator

    → Position of **previous operator** is determined



Example

- A + B * C → A B C * +

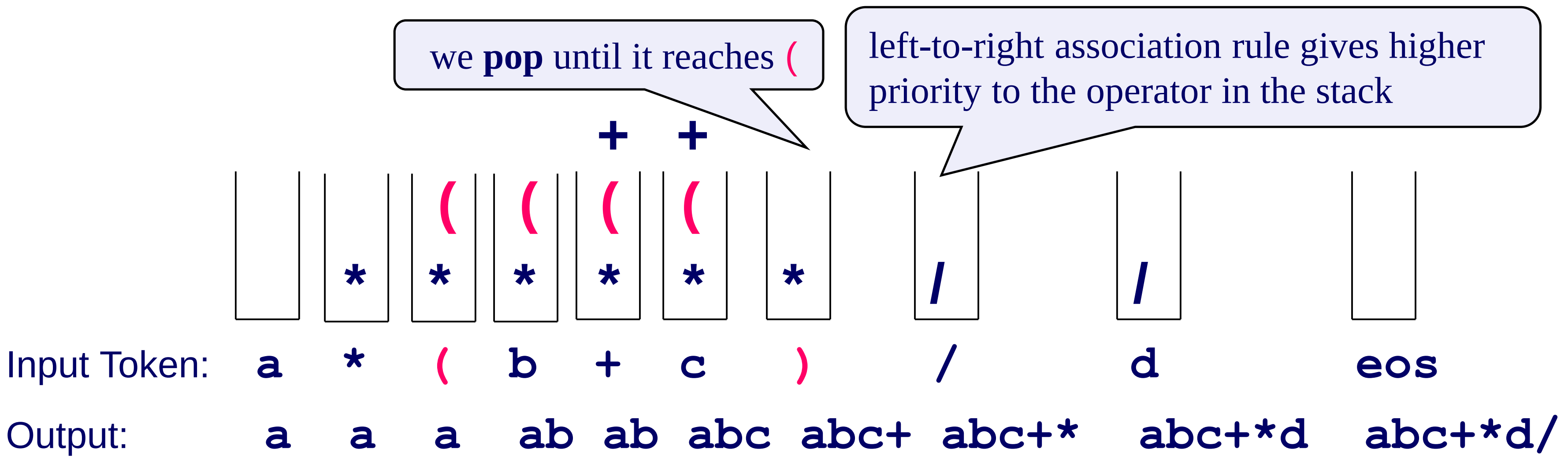

Operands are popped immediately
The priority of * is higher than +
*
*
+
+
+
+
Input Token: A + B * C eos (end-of-string)
Output: A A AB AB ABC ABC*+

- A challenging problem
  - What if current operator's precedence is the **same** with that of previous operator?
    Pop the previous operator!
- Examples: Transfer the following infix notations into postfix notations with a **stack**
  - a+b*c*d+e
  - a/b-c+d*e-a*c

Parenthesized Expression

- Parenthesis overrides precedence of operators
  - Left parenthesis: mark
    - Just **push**
  - Right parenthesis: conclusion of an operand
    - **Pop** and print all operators in stack up to *left parenthesis*

Example) a * (b + c) / d


we **pop** until it reaches (
left-to-right association rule gives higher priority to the operator in the stack
Input Token: a * (b + c) / d eos
Output: a a a ab ab abc abc+ abc+* abc+*d abc+*d/

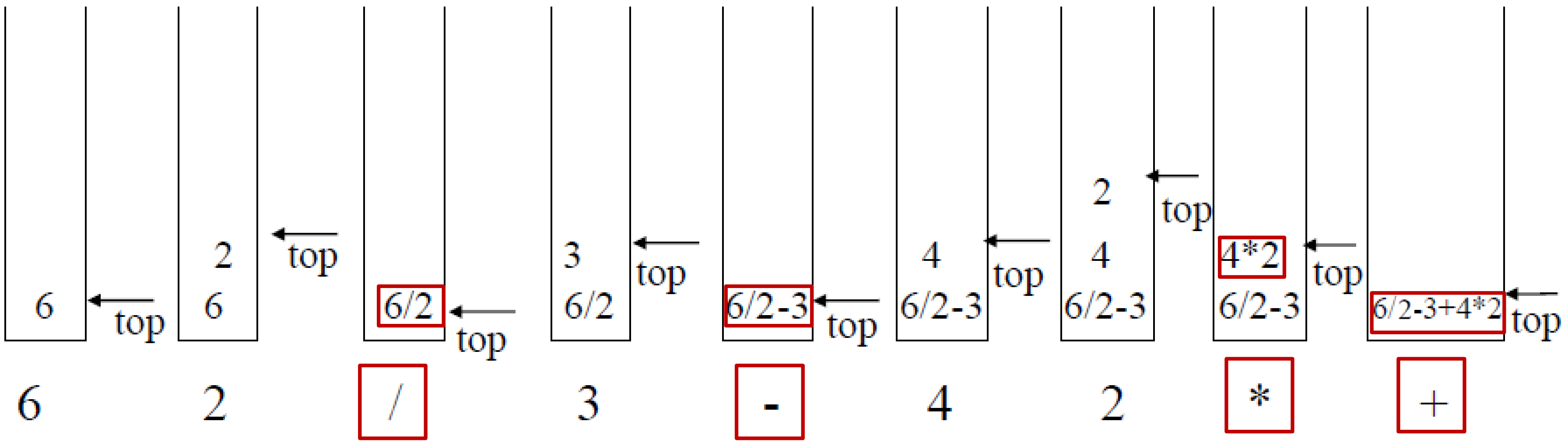
Examples

- Transfer the following infix notations into postfix notations with a **stack**
  - (a/(b-c+d))*(e-a)*c
  - (a/b-c*d)/e+f-g

Evaluation of Postfix Expression

- Using **stack**, evaluation of postfix expression is straightforward
  1. Scan from left to right
  2. If current element is *operand*, **push** it
  3. If current element is *operator*, **pop** correct number of operand and perform the operation, and **push** the result **back** on the stack

Ex) 6/2 – 3 + 4 * 2 ➔ 6 2 / 3 - 4 2 * +



Examples

- Calculate the following expressions using a **stack**
  - abc+*d*
  - ab+d*efad*+/+

Queues

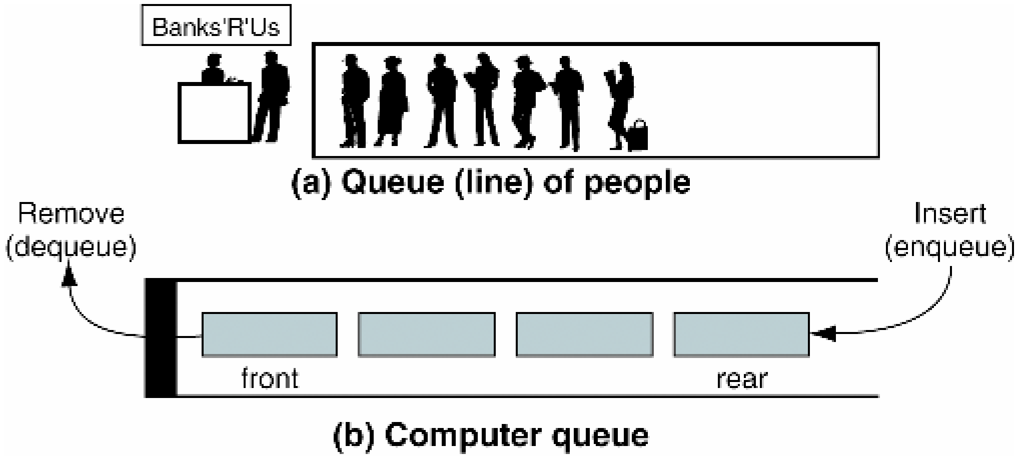
- Basic Queue Operations -

연세대학교
YONSEI UNIVERSITY

Introduction

- Queue
  - A linear list in which data can only be inserted at one end (**rear**) and deleted from the other end (**front**)

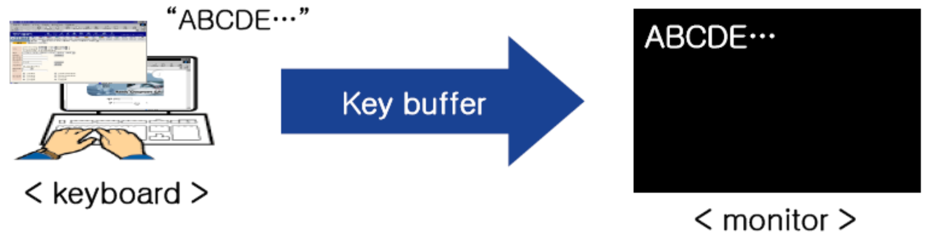
  Ex) Waiting list in real life
  - **FIFO** (First-In-First-Out)



(a) Queue (line) of people

(b) Computer queue

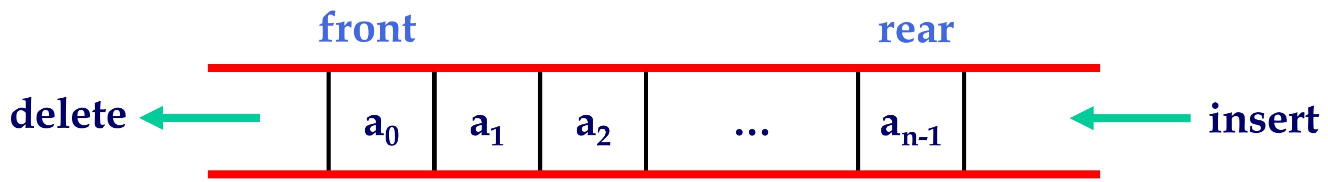
- Waiting line
  - Cafeteria, ticket office,…
- Key buffer



- Computer systems
  - Process scheduling queue
  - Message queue

Queue

- Given a queue Q = (a_0, a_1, …, a_{n-1}), a_0 is the front element and a_{n-1} is the rear element



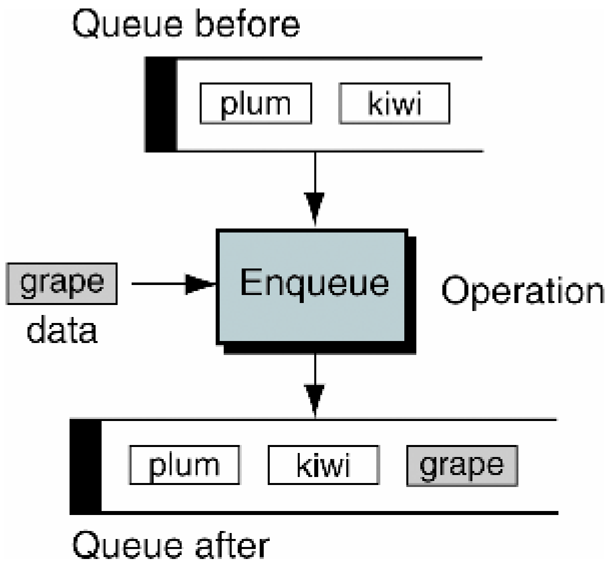
< queue >

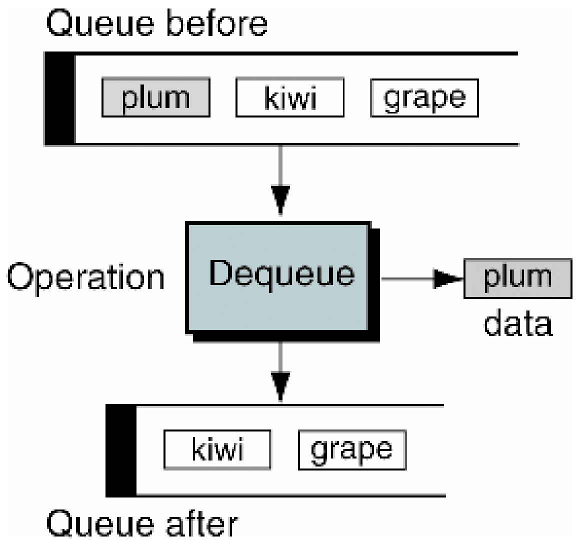
- Major operations
  - **Enqueue**: queue insert
  - **Dequeue**: queue delete
  - **QueueFront**: retrieve the data at the front
  - (**QueueRear**: retrieve the data at the rear)

Queue Operations (1/2)

- **Enqueue**: queue insert operation
  - The new element becomes the *rear*
  - Queue is already full → overflow

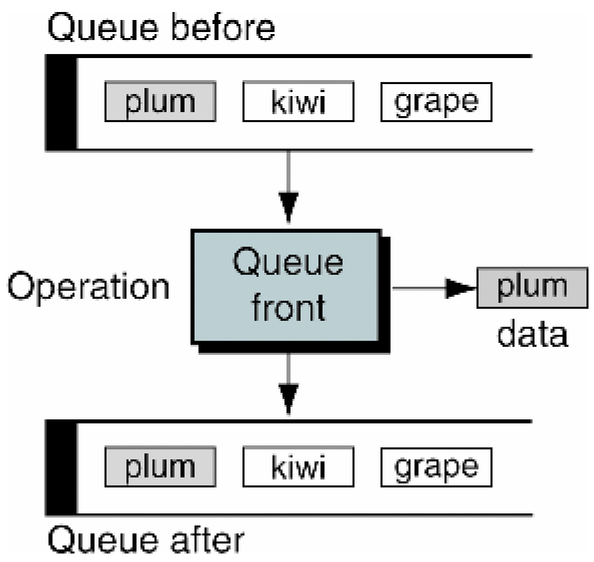
- **Dequeue**: queue delete operation
  - The data at the *front* is returned and removed
  - Queue is empty → underflow



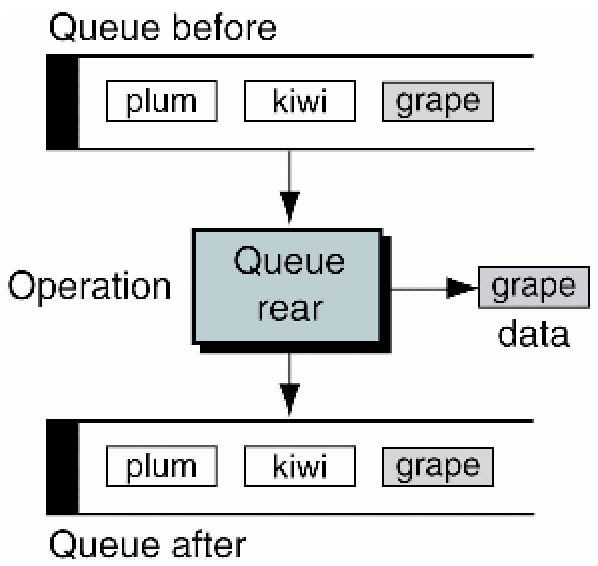


Queue Operations (2/2)

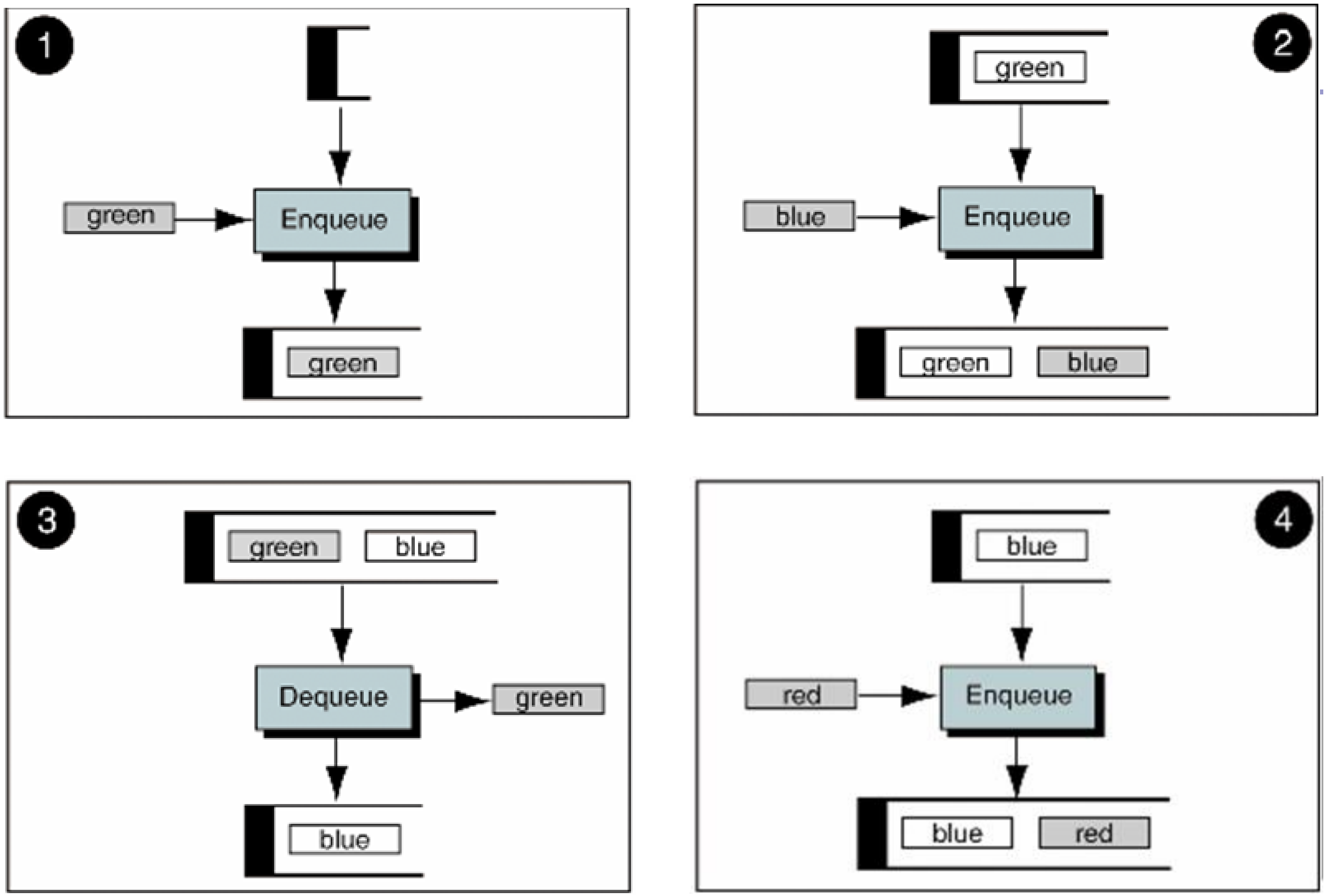
- **QueueFront**: retrieve data at the *front*



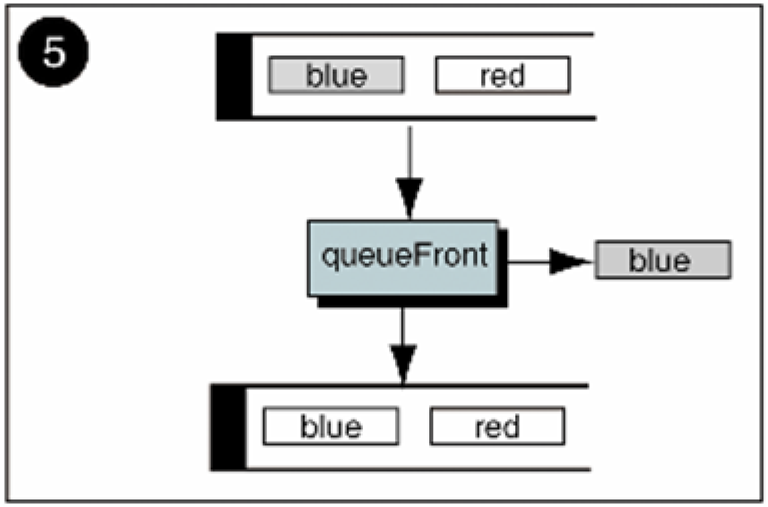
- **QueueRear**: retrieve data at the *rear*
  - Conflict with general concept of Queue

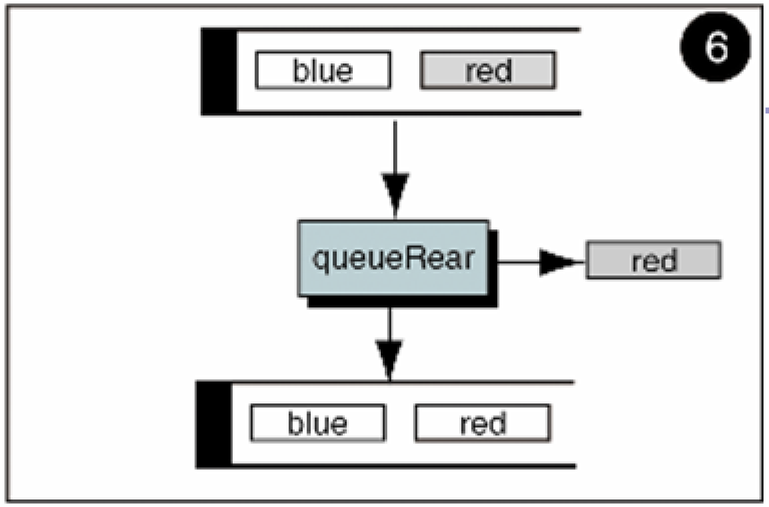


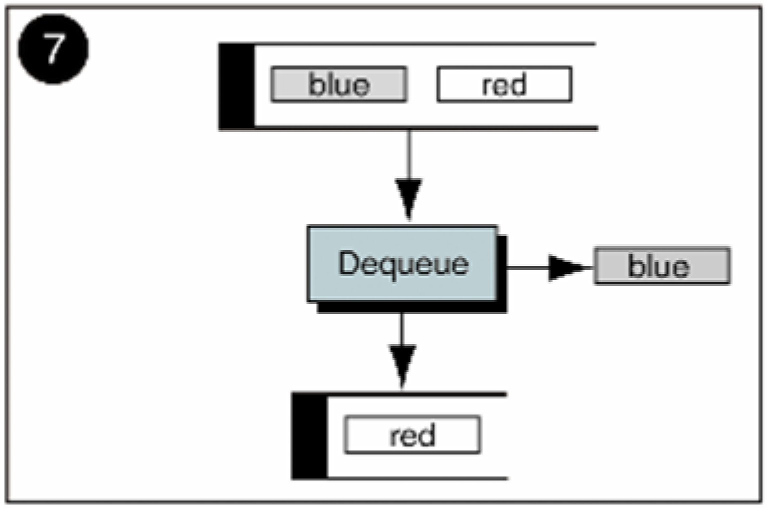
Queue Example (1/2)

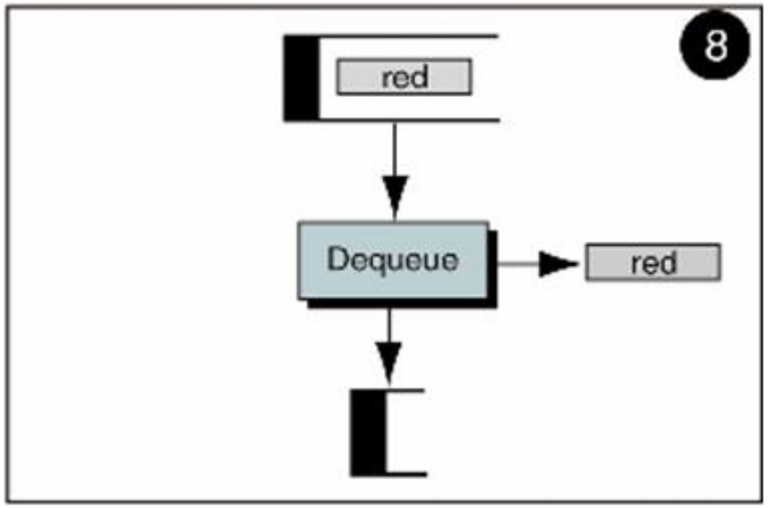


Queue Example (2/2)









Problem of Sequential Queue

- Repetitive **Enqueue** and **Dequeue** require shift of all elements

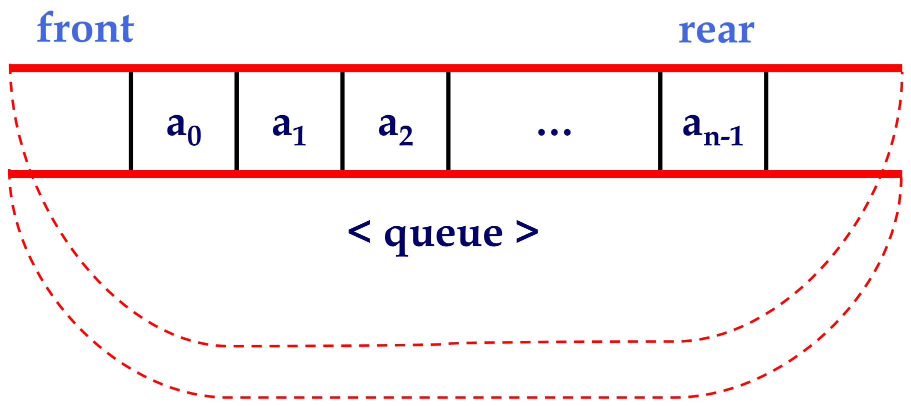
front	rear	Q[0]	Q[1]	Q[2]	Q[3]	comments
-1	-1					queue is empty
-1	0	J1				job 1 is added
-1	1	J1	J2			job 2 is added
-1	2	J1	J2	J3		job 3 is added
0	2		J2	J3		job 1 is deleted
1	2			J3		job 2 is deleted

  - Shift at every deletion is time-consuming

- Alternative method: **circular queue**

Circular Queue (1/2)

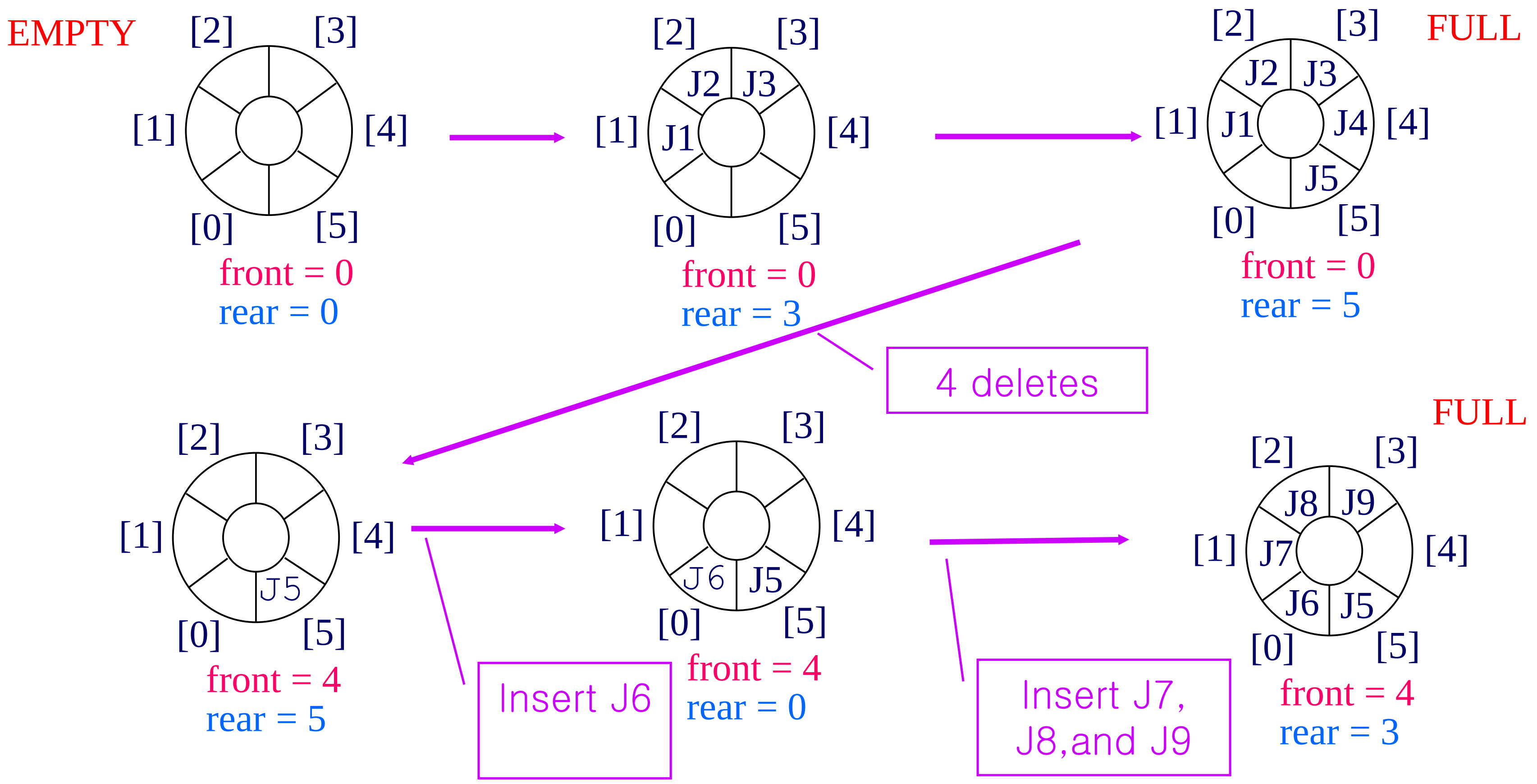
- Circular queue: queue whose logical structure is circular
  - Last element is followed by first element
  - **Enqueue** occurs
    - rear = (rear + 1) **% size**
  - **Dequeue** occurs
    - front = (front + 1) **% size**



Circular Queue (2/2)

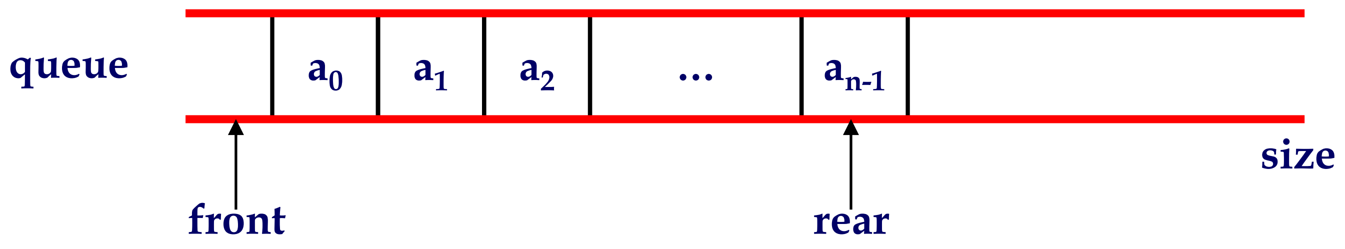
- front == rear → *empty* condition
- (rear+1) % size == front → *full* condition

→ To distinguish empty and full, at least one space is kept empty



Array Implementation of (Circular) Queue

- Data representation



Queue

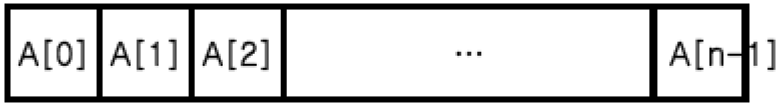
```
Element *queue;     // array to store elements
int max_size;       // maximum size
int front, rear;    // front and rear
```

Cf. *Element*: an arbitrary type (ex: typedef int *Element*;)

Linked Lists

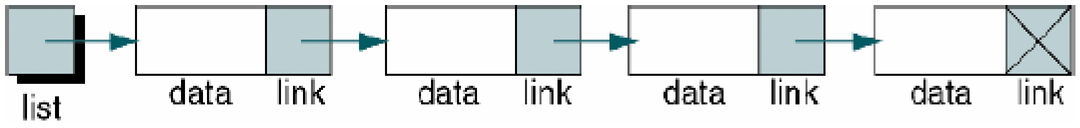
Implementations

- Two basic structures to implement basic operations
  - **Array**: mapping from index to element



< Array A of size n >

  - **Linked list**: (logically) ordered collection of data, in which each element contains location of the next element

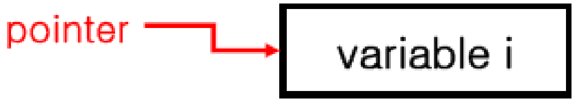


e.g., shift elements

Prerequisite: Pointers in C

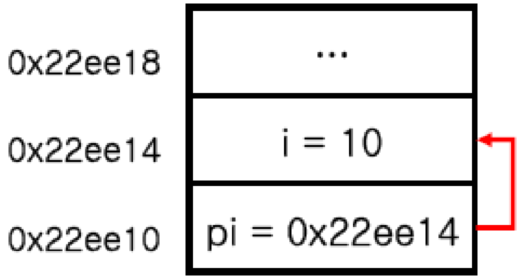
- Pointer
  - A data type whose value is used to refer to ("*points to*") another value stored elsewhere in the computer memory

- Pointer-related operators
  - Address operator **&**
  - Dereferencing operator *



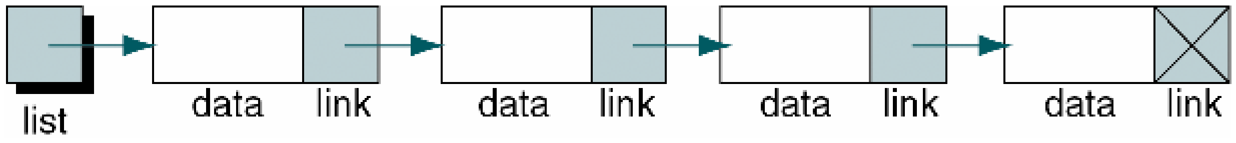
```
int i = 0;          // variable
int *pi = NULL;     // pointer

pi = &i;            // &i: address of i
i = 10;             // store 10 to variable i
*pi = 10;           // store 10 to memory location pointed by pi
```

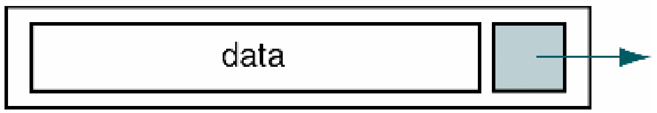


Linear Lists

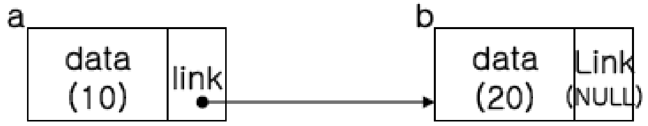
- Linear lists



  - **Node** (element in a list)



- Linking nodes
  - How to make a code to generate the following linked list?

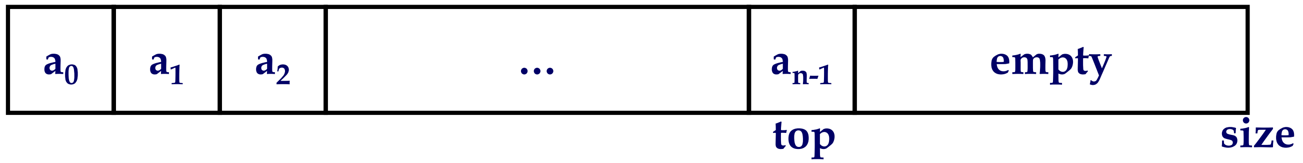


Step 1.Declare nodes

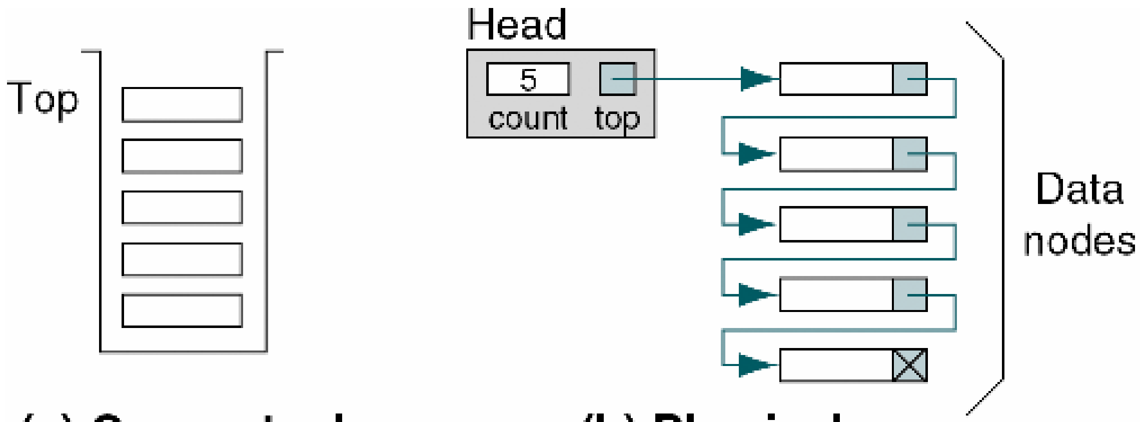
Step 2. Link nodes

Linked List Implementation of **Stack**

- Problems of array implementation
  - Fixed size
  - Inefficiency in insertion and deletion



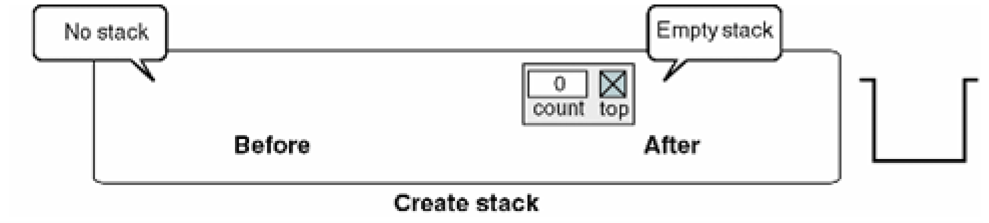
- Alternative: *linked list* representation



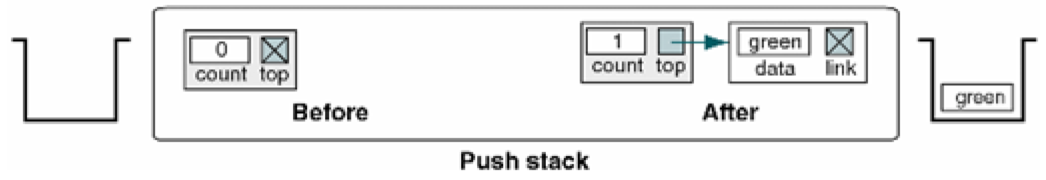
(a) Conceptual (b) Physical

Operations of Linked List Implementation of **Stack**

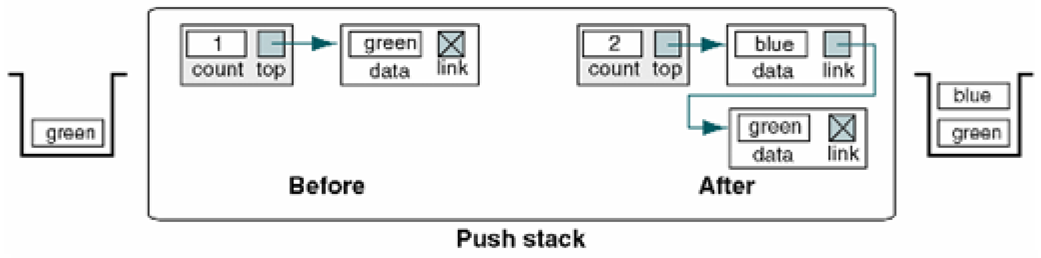
Create:



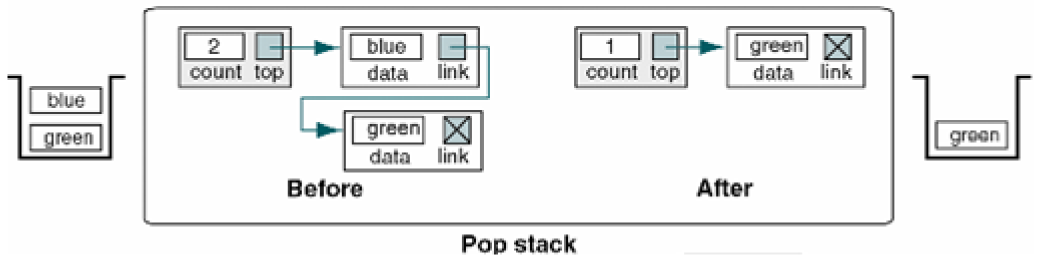
Push:



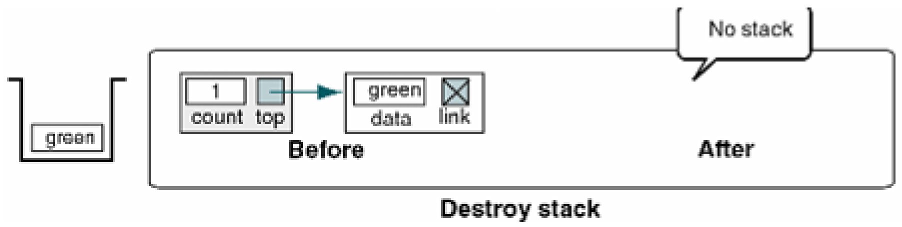
Push:



Pop:



Destroy:



Data Representation of Stacks

- Stack node structure

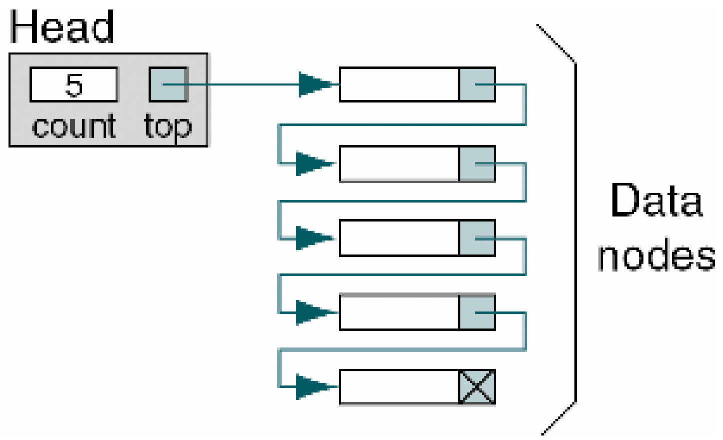
StackNode

```
Element data;
StackNode *link;
```

- Stack head structure

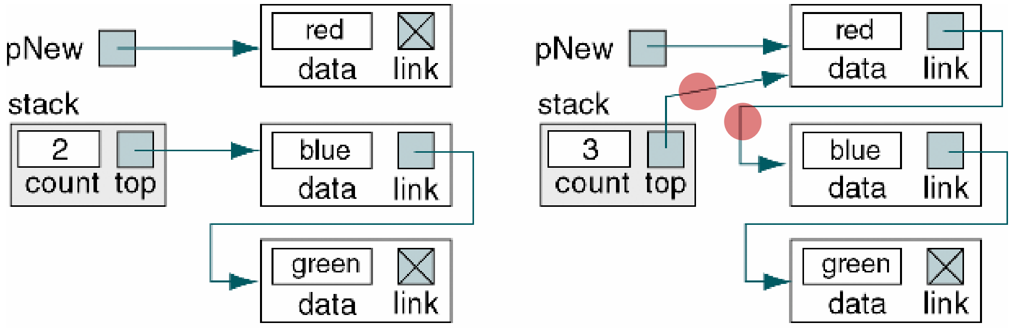
Stack

```
int count;
StackNode *top;
```



Linked List Implementation of Push

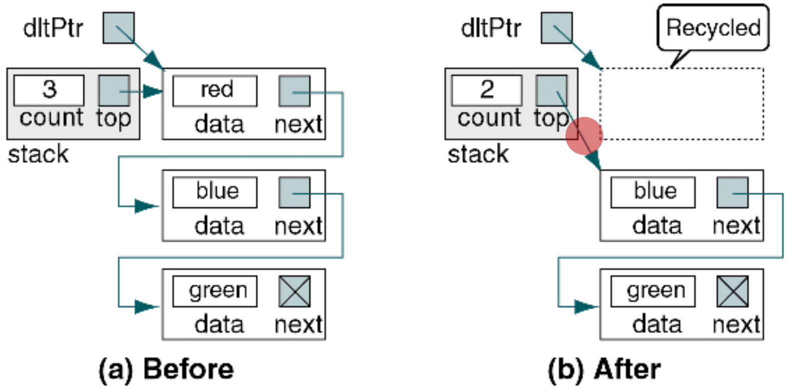
```
Algorithm pushStack (stack, data)
Insert (push) one item into the stack.
   Pre  stack passed by reference
        data contain data to be pushed into stack
   Post data have been pushed in stack
1  allocate new node
2  store data in new node
3  make current top node the second node
4  make new node the top
5  increment stack count
end pushStack
```



(a) Before (b) After

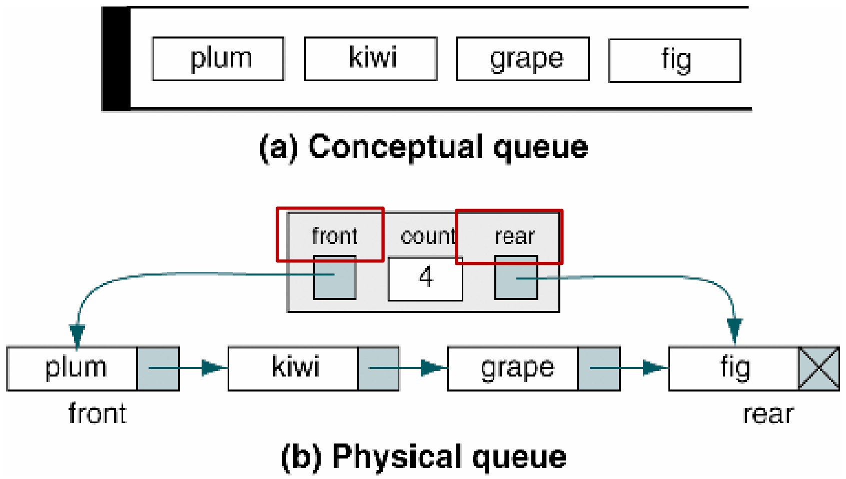
Linked List Implementation of Pop

```
Algorithm popStack (stack, dataOut)
 1  if (stack empty)
     1   set success to false
 2  else
     1   set dataOut to data in top node
     2   make second node the top node
     3   decrement stack count
     4   set success to true
 3  end if
 4  return success
end popStack
```



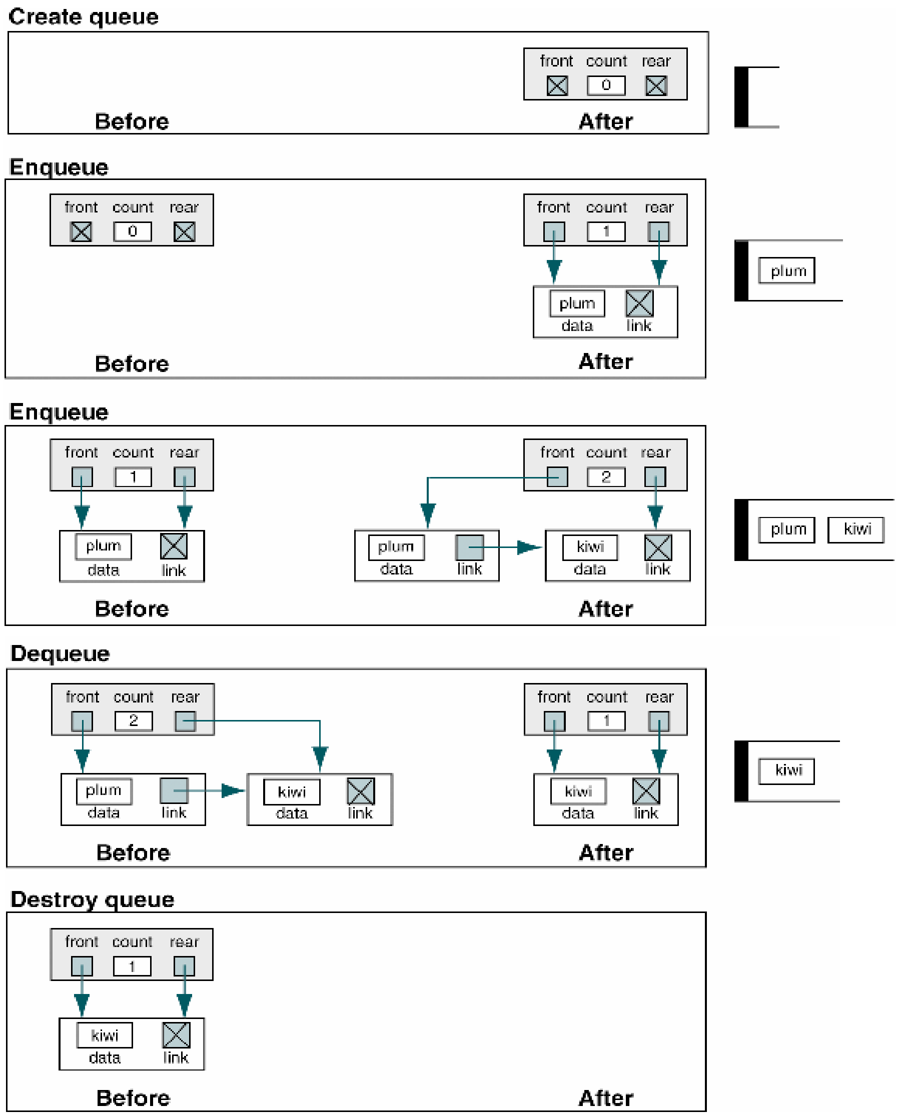
Linked List Implementation of **Queue**

- Linked list implementation

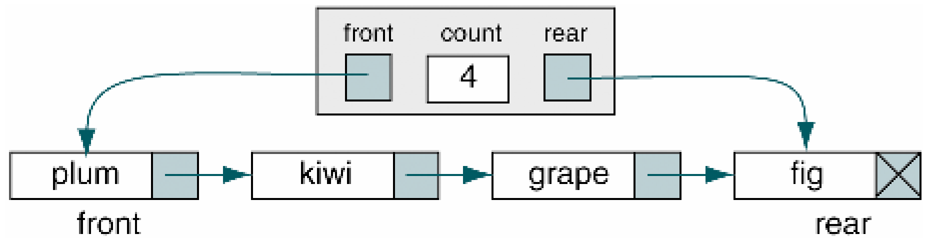


(a) Conceptual queue

(b) Physical queue



Data Representation of Queues



- Queue node

QueueNode

```
Element data;
QueueNode *next;
```

- Queue

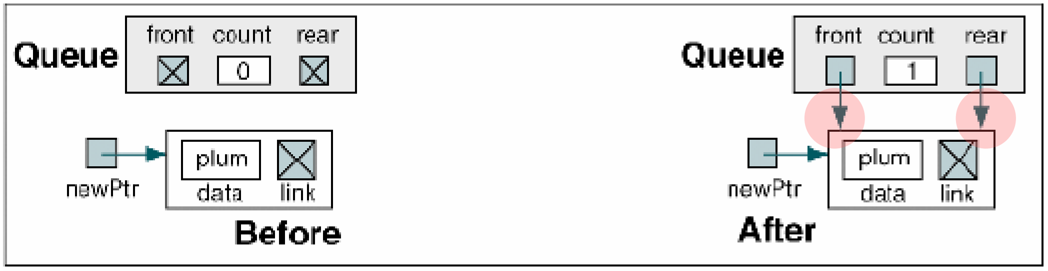
Queue

```
int count;                    // # of elements
QueueNode *front, *rear;   // front and rear
```

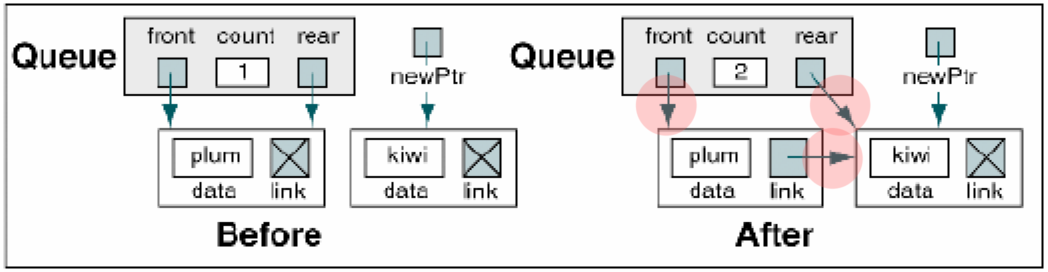
Cf. *Element*: an arbitrary type (ex: typedef int *Element;*)

Linked List Implementation of Enqueue

- **Enqueue**



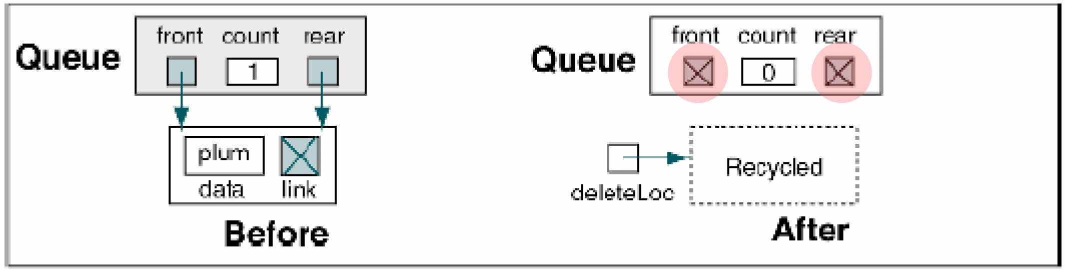
(a) Case 1: insert into empty queue



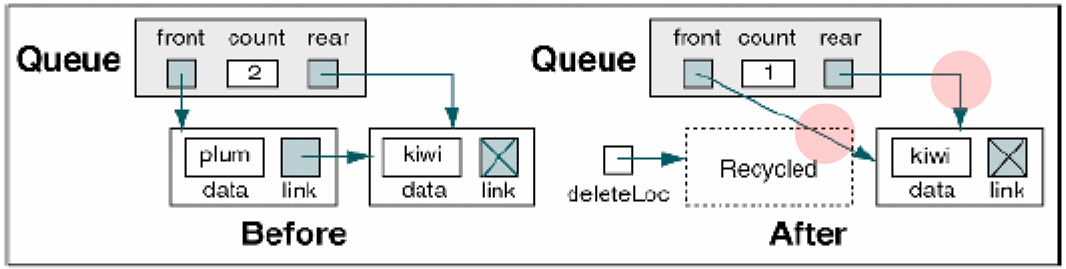
(b) Case 2: insert into queue with data

Linked List Implementation of Dequeue

- **Dequeue**



(a) Case 1: delete only item in queue



(b) Case 2: delete item at front of queue