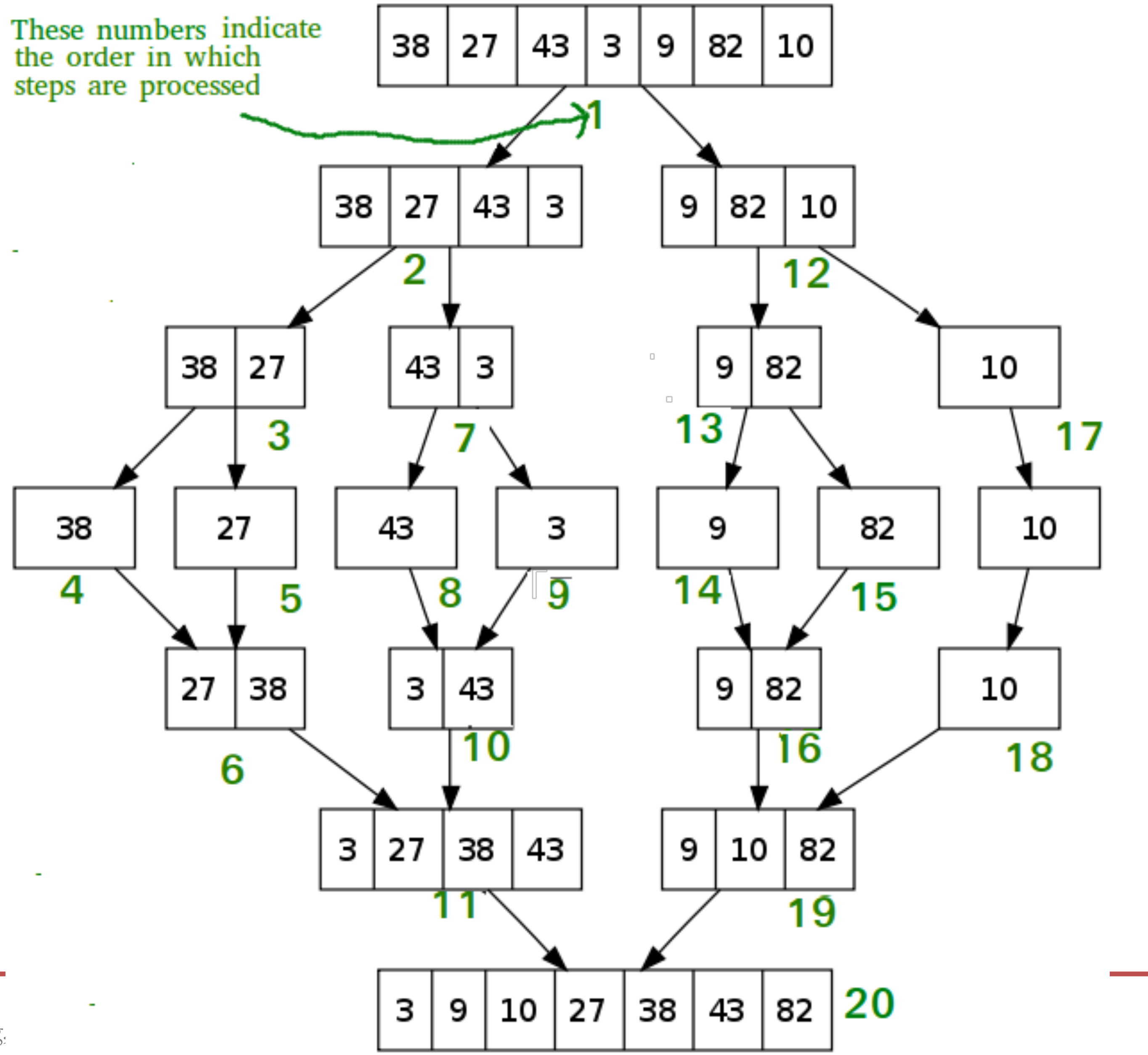
# Data Structures and Algorithms

Lecture 5: Lists, Stacks, and Queues (II)

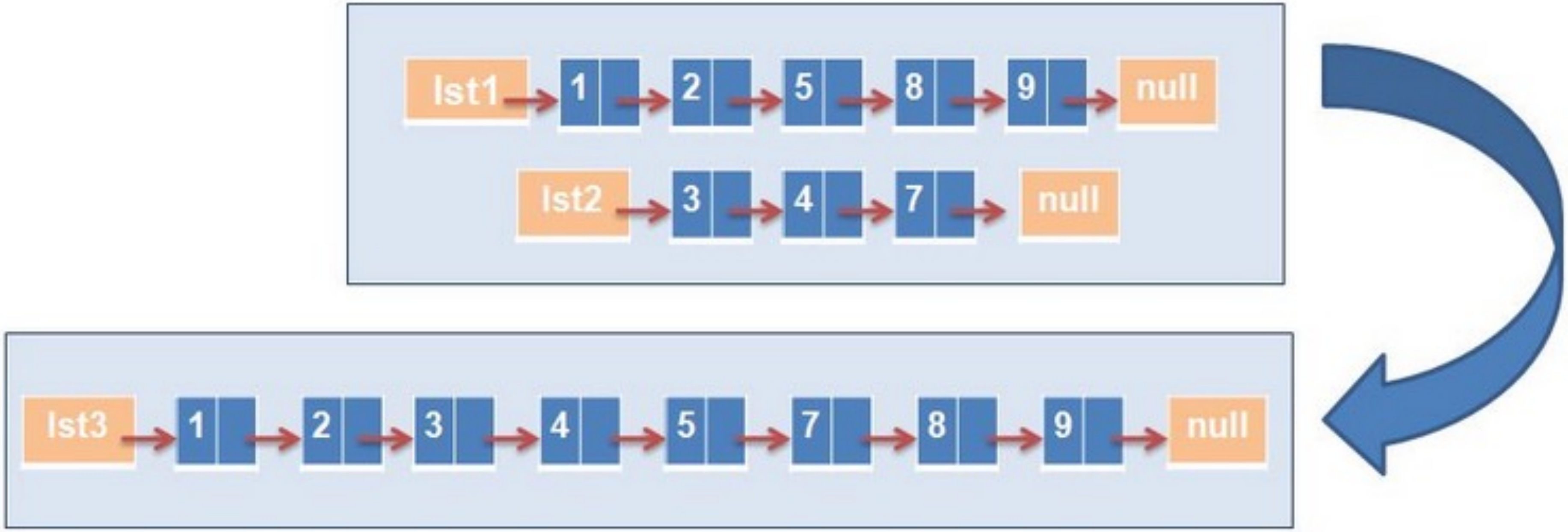
# An application of lists -- merge sort



# Merge Sort

1. If there is only one number in the list, return;
2. **Split** a list into two sub-lists with almost equal length
3. **Recursively sort** the two sub-lists, where the numbers in each sub-lists are in increasing order
4. **Merge** the two sub-lists into one list such that the number the merged list are in increasing order

# How to merge two sorted linked-lists?



# Merge two sorted linked-lists

```
/**
* Definition for singly-linked list.
* struct ListNode {
*   int val;
*   ListNode *next;
*   ListNode() : val(0), next(nullptr) {}
*   ListNode(int x) : val(x), next(nullptr) {}
*   ListNode(int x, ListNode *next) : val(x), next(next){}
* };
*/
```

# Merge two sorted linked-lists

```
/** Recursion Method */
class Solution{
public:
  ListNote* mergeTwoLists(ListNode* l1, ListNote* l2) {
    if(!l1){ // l1 is NULL
        return l2;
    } else if(!l2) { // l2 is NULL
        return l1;
    } else if(l1->val < l2->val) {
        l1->next = mergeTwoLists(l1->next, l2);
        return l1;
    } else { // l1->val >= l2->val
        l2->next = mergeTwoLists(l1, l2->next);
        return l2;
    }
  }
};
```

# Recursion Method

**Function**

$$merge(l1, l2) = \begin{cases} l2, & l1 \text{ is NULL} \\ l1, & l2 \text{ is NULL} \\ merge(l1 \to next, l2), & if\ \ l1 \to val < l2 \to val \\ merge(l1, l2 \to next), & if\ \ l1 \to val\ \geq l2 \to val \end{cases}$$

**Complexity**:

- *Time:* ***O(n+m)***
- *Space:* ***O(n+m)***

# Merge two sorted sub-lists

```
/** Iteration Method */
class Solution{
public:
  ListNote* mergeTwoLists(ListNode* l1, ListNote* l2) {
    ListNote* tem = new ListNode(0);
    ListNode* ans = tem;
    while (l1!=NULL && l2!=NULL)
    { if (l1->val < l2->val)
        { tem->next = l1; l1 = l1->next; }
      else
        { tem->next = l2; l2 = l2->next; }
    }
    if (l1!=NULL) tem->next = l1;
    if (l2!=NULL) tem->next = l2;
    return ans->next;
  }
};
```

# Iteration Method

Algorithm steps

1. **Initialize** two lists *tem*, *ans*;
2. **Iteratively merge** two nodes;
   - Merge the small one, and move pointer forward
3. **Merge tail** the last non-NULL list;
   - Return the result.

**Complexity**:
- *Time:* ***O(n+m)***
- *Space:* ***O(1)***

# How to merge **k** sorted sub-lists?

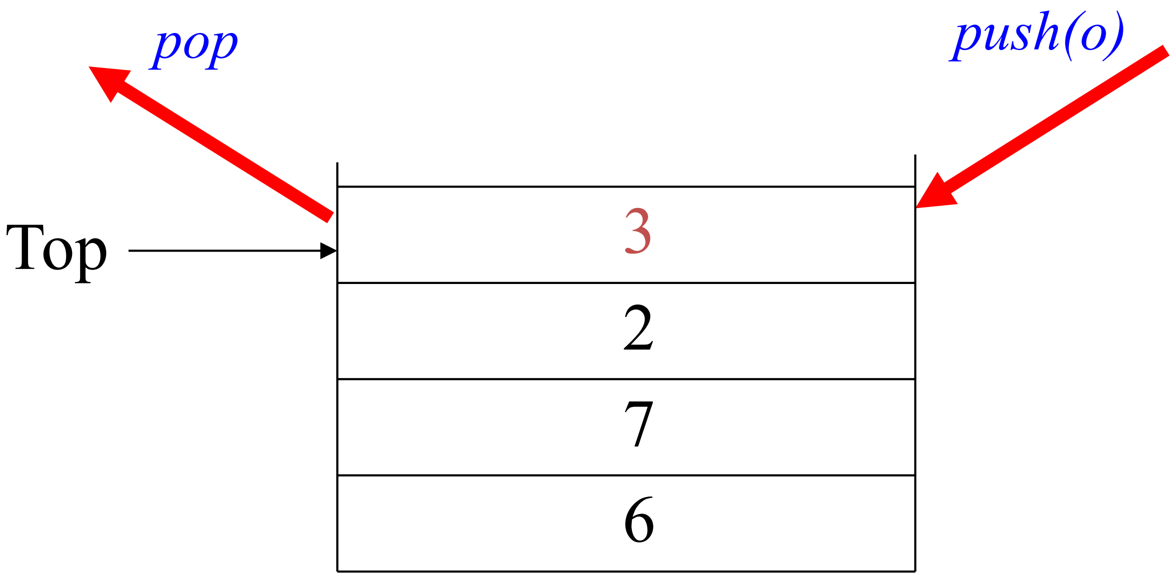
- Merge ***k*** sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

```
Example:
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
  1->4->5,
  1->3->4,
  2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6
```

# Stacks

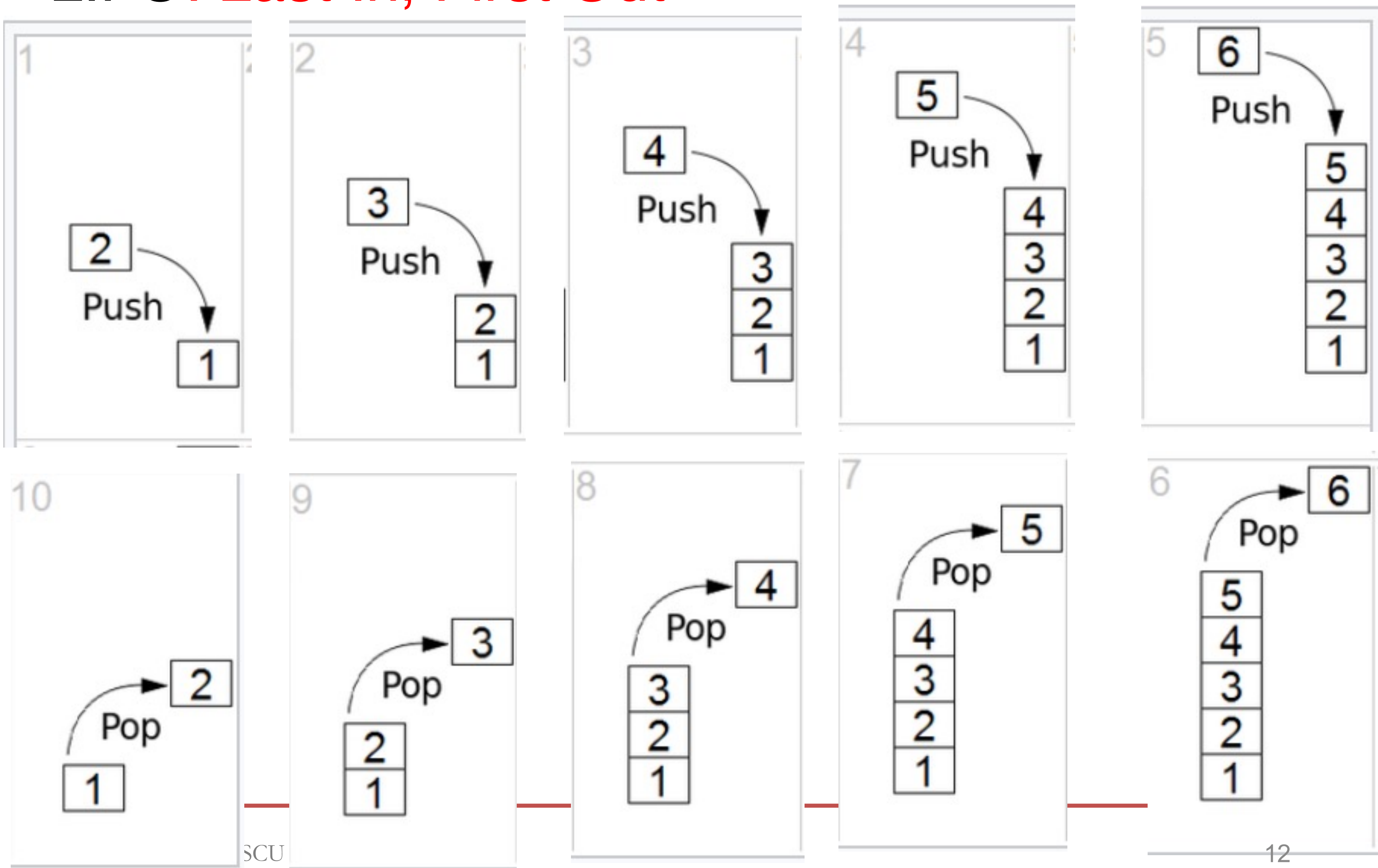
# What is a Stack?

- A ***stack*** is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the ***top***.
- Operations: PUSH (insert) and POP (delete)



# Stacks

- LIFO: Last In, First Out



# Stacks

Notation:

- Insert: PUSH
- Remove: POP
- The accessible element is called TOP.

- Restricted form of list: Insert and remove only at front of list.

# Stack ADT

```
// Stack abstract class
template <typename E> class Stack {
public:
  void clear();

  /** Push an element onto the top of the stack.
  @param it Element being pushed onto the stack.*/
  void push(E& it);

  /** Remove and return top element.
      @return The element at the top of the stack.*/
  E pop();

  /** @return A copy of the top element. */
  E topValue();

  /** @return Number of elements in the stack. */
  public int length();
};
```

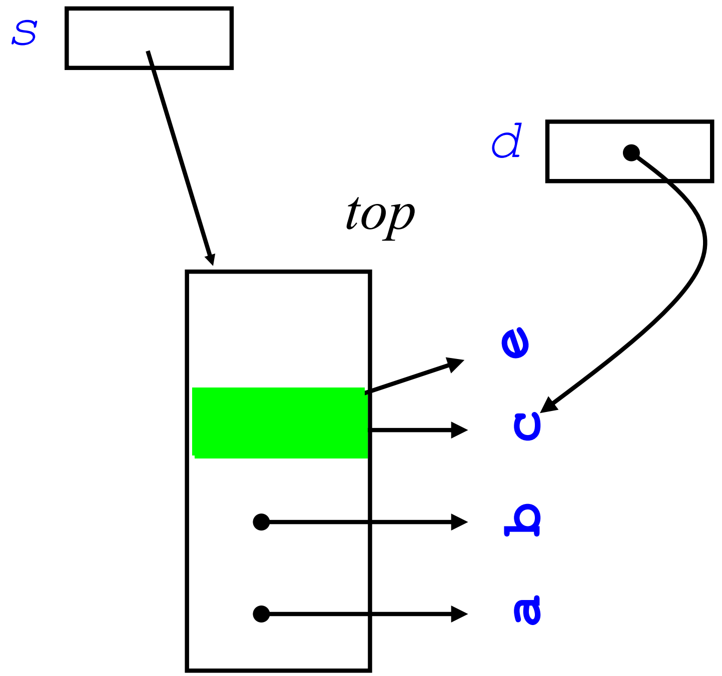
# Stack ADT Interface

- The main functions in the Stack ADT are (S is the stack)

```
boolean isEmpty();              // return true if empty
boolean isFull(S);              // return true if full
void push(S, item);             // insert item into stack
void pop(S);                    // remove most recent item
void clear(S);                  // remove all items from stack
Item top(S);                    // retrieve most recent item
Item topAndPop(S);              // return & remove most recent item
```

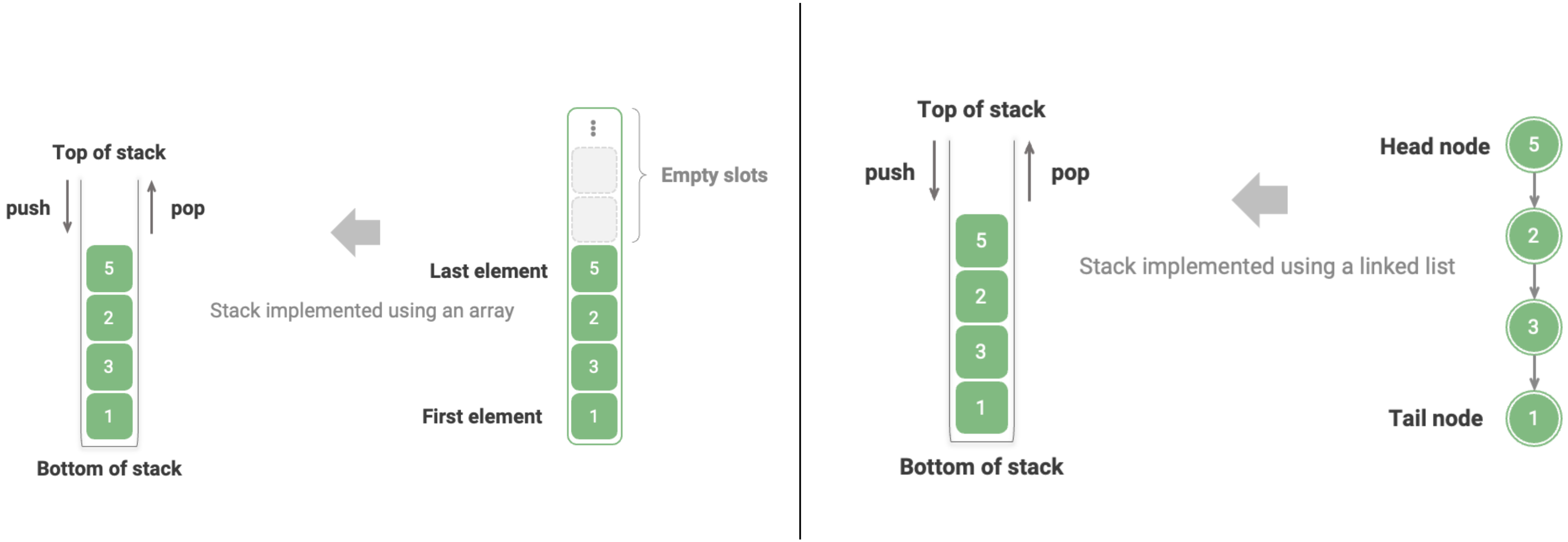
# Sample Operation

```
Stack S = malloc(sizeof(stack));
push(S, "a");
push(S, "b");
push(S, "c");
d=top(S);
pop(S);
push(S, "e");
pop(S);
```



# Implementation of Stacks

- Array-based stacks
- Linked stacks



# Array-Based Stacks

```
// Array-based stack implementation

private:
   int maxSize;   // Maximum size of stack
   int top;       // Index for top element
   E *listArray; // Array holding elements
```

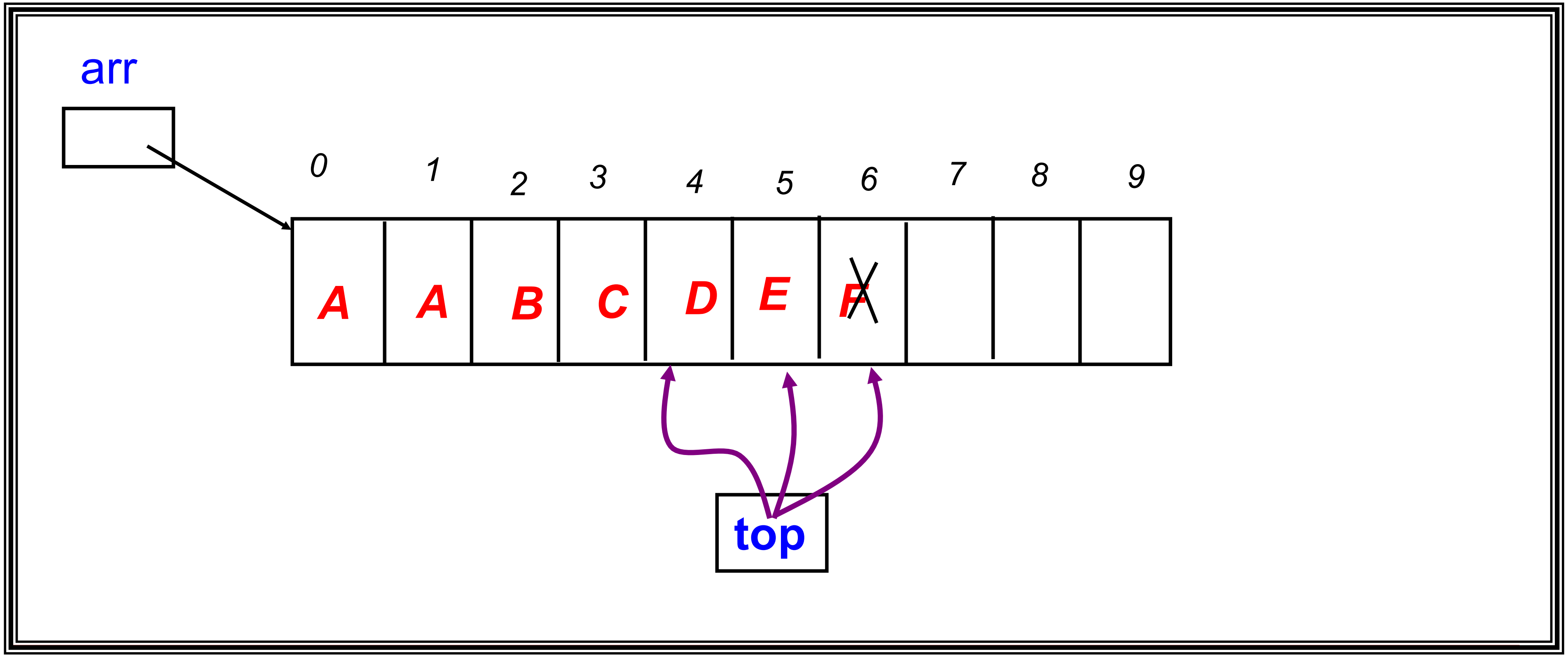
Questions:

- Which end is the top of the stack?
  - Array[0] is the bottom and array[top-1] is the top
- Where does "top" point to?
  - Array index for the top element currently in the stack.
- What is the cost of the operations?
  - $\Theta(1)$ for each push or pop operation.

# Implementation by Array

- Use Array with a top index pointer as an implementation of stack

*StackAr*



# Code

```
typedef struct {
    int A[MAX];
    int top;
} STACK;
```

```
void clear(STACK *pS)
{
    pS->top = -1;
}

BOOLEAN isEmpty(STACK *pS)
{
    return (pS->top < 0);
}

BOOLEAN isFull(STACK *pS)
{
    return (pS->top >= MAX-1);
}
```

# More code

```
BOOLEAN pop(STACK *pS, int *px)
{
    if (isEmpty(pS))
        return FALSE;
    else {
        (*px) = pS->A[(pS->top)--];
        return TRUE;
    }
}
```

# More code

```
BOOLEAN push(int x, STACK *pS)
{
    if (isFull(pS))
        return FALSE;
    else {
        pS->A[++(pS->top)] = x;
        return TRUE;
    }
}
```

# Linked Stacks

```
// Linked stack implementation
private:
   int size;         // Number of elements
   Link<E>* top;     // Pointer to first element
```

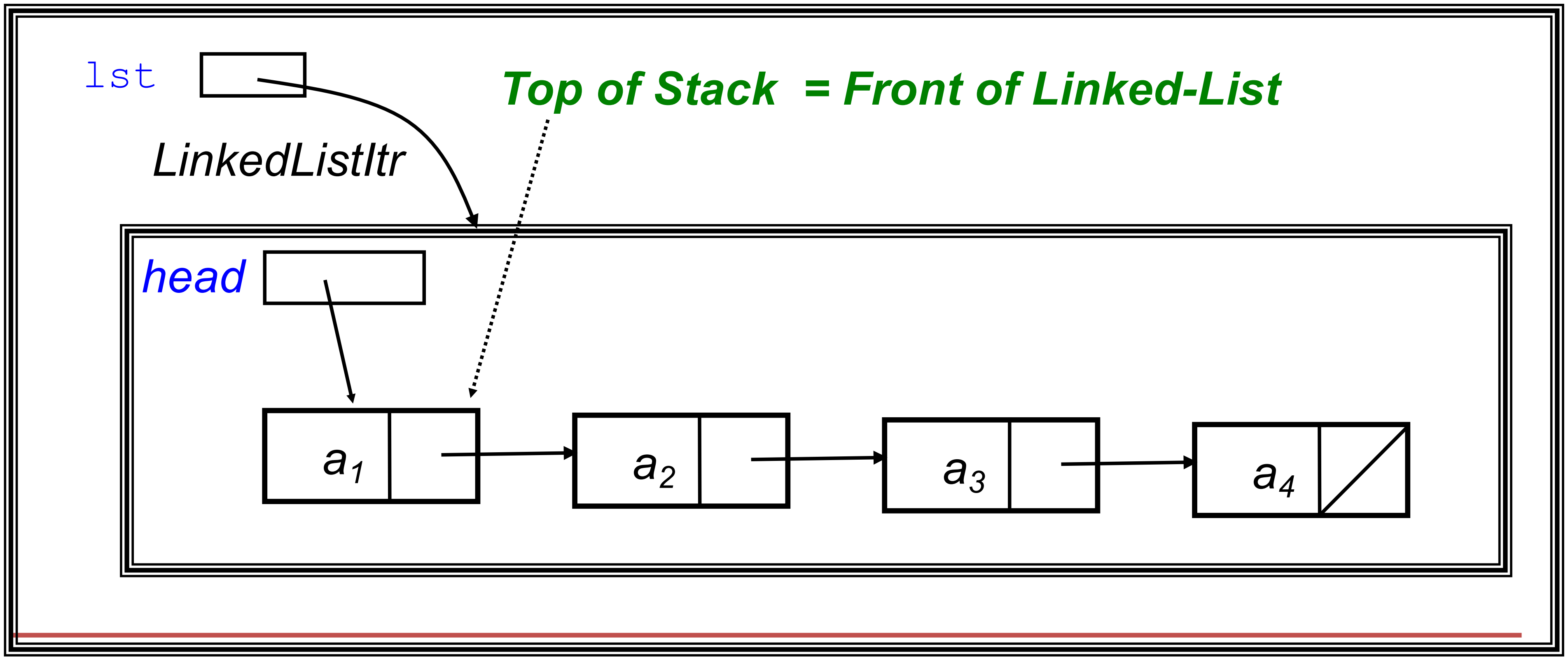
- **Push**/**PoP** operations
  - Elements are inserted and removed only from the head of the list.

- Which end is the top of the stack?
  - Linked list head
- Where does "top" point to?
  - The new/next link node for stores the top nodes
- What is the cost of the operations?
  - $\Theta(1)$

# Implementation by Linked Lists

- Can use a Linked List as implementation of stack

*StackLL*



# Code

```
struct Node {
    int element;
    Node * next;
};
typedef struct Node * STACK;
```

```
void clear(STACK *pS)
{
    (*pS) = NULL;
}

BOOLEAN isEmpty(STACK *pS)
{
    return ((*pS) == NULL);
}

BOOLEAN isFull(STACK *pS)
{
    return FALSE;
}
```

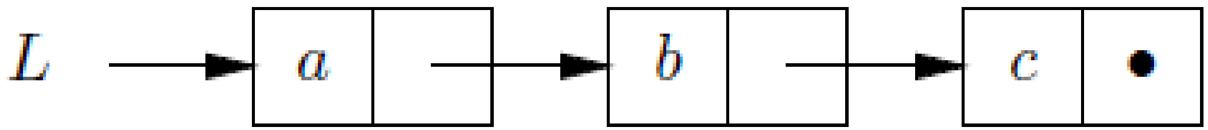
# More code

```
BOOLEAN pop(STACK *pS, int *px)
{
    if ((*pS) == NULL)
        return FALSE;
    else {
        (*px) = (*pS)->element;
        (*pS) = (*pS)->next;
        return TRUE;
    }
}
```

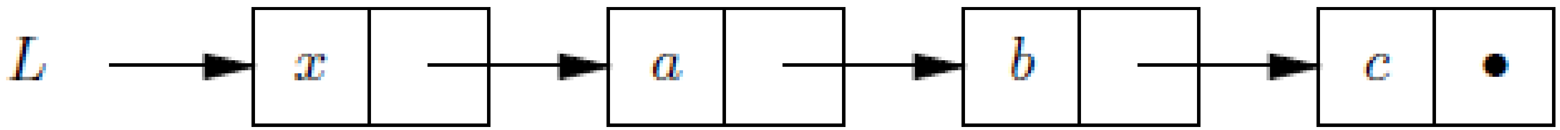
# More Code

```
BOOLEAN push(int x, STACK *pS)
{
    STACK newCell;

    newCell = (STACK) malloc(sizeof(struct CELL));
    newCell->element = x;
    newCell->next = (*pS);
    (*pS) = newCell;
    return TRUE;
}
```

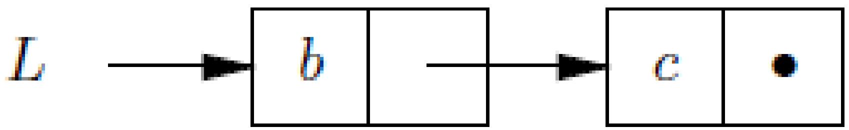
# Effects of Linked Stacks




(a) List $L$.




(b) After executing $push(x, L)$.




(c) After executing $pop(L, x)$ on list $L$ of (a).

# Array-based *vs* Linked Stacks

- Time comparison
  - Operations for both two stacks take *constant* time.

- Space comparasion
  - Array-based stack has an initially **fixed-size** array.
  - Linked stack can **shrink and grow** but requires the *overhead* of a link field for every element.

# Applications of Stacks

- Many application areas use stacks:
  - line editing
  - bracket matching
  - postfix calculation
  - function call stack

# Line Editing

- A line editor would place characters read into a buffer but may use a backspace symbol (denoted by ←) to do error correction
- *Refined Task*
  - read in a line
  - correct the errors via backspace
  - print the corrected line in reverse

Input : abc_defg~~h~~←2klp~~qr~~←←wxyz

Corrected Input : abc_defg2klpwxyz

Reversed Output : zyxwplk2gfed_cba

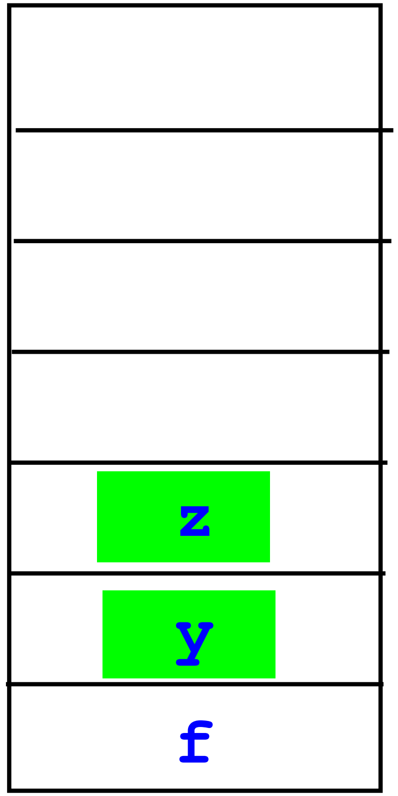
# The Procedure

- Initialize a new stack
- For each character read:
  - if it is a backspace, *pop out last char entered*
  - if not a backspace, *push the char into stack*
- To print in reverse, pop out each char for output

Input : `fgh←r←←yz`

Corrected Input : $\boxed{\texttt{fyz}}$

Reversed Output : $\boxed{\texttt{zyf}}$




***Stack***

# Bracket Matching Problem

- Ensures that pairs of brackets are properly matched
  - An Example: `{a,(b+f[4])*3,d+f[5]}`
  - Bad Examples:

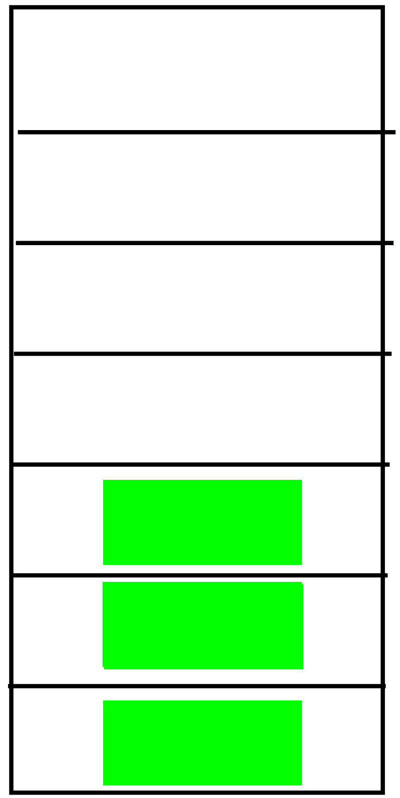
```
(..)..)          // too many closing brackets

(..(..)          // too many open brackets

[..(..]..)       // mismatched brackets
```

# Informal Procedure

Initialize the stack to empty
For every char read
    if open bracket then *push onto stack*
    if close bracket, then
        return & remove most recent item
        from *the stack*
        if doesn't match then *flag error*
    if non-bracket, *skip the char read*

Example

```
{a,(b+f[4])*3,d+f[5]}
```



***Stack***

# Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of a stack.

Infix - arg1 **op** arg2
Prefix - **op** arg1 arg2
Postfix - arg1 arg2 **op**

*infix* *postfix*

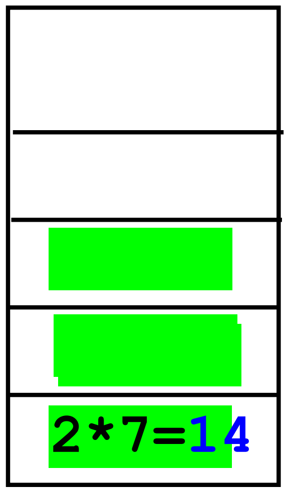
2*3+4 → (2*3)+4 → 2 3 * 4 +

2*3+4 → 2*(3+4) → 2 3 4 + *

# Informal Procedure

Initialize stack S
For each item read.
  If it is an operand,
    *push* onto the stack
  If it is an operator,
    *pop* arguments from stack;
    *perform operation*;
    *push* result onto the stack

| Expr | |
|---|---|
| 2 | `push(S, 2)` |
| 3 | `push(S, 3)` |
| 4 | `push(S, 4)` |
| + | `arg2=topAndPop(S)` |
| | `arg1=topAndPop(S)` |
| | `push(S, arg1+arg2)` |
| * | `arg2=topAndPop(S)` |
| | `arg1=topAndPop(S)` |
| | `push(S, arg1*arg2)` |




*Stack*

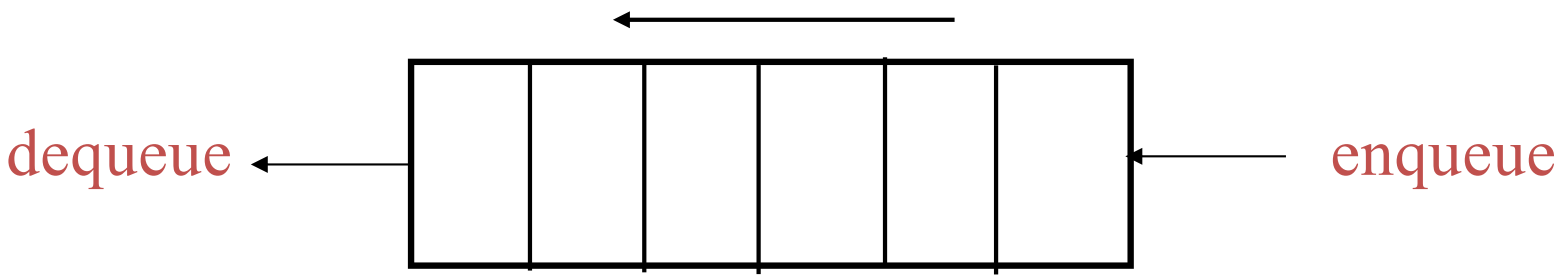
# Summary

- The ADT stack operations have a last-in, first-out (LIFO) behavior.
- Stack can be implemented using array-based or linked lists.
- Stack has many applications
  - algorithms that operate on algebraic expressions
  - a strong relationship between recursion and stacks exists.

# Queues

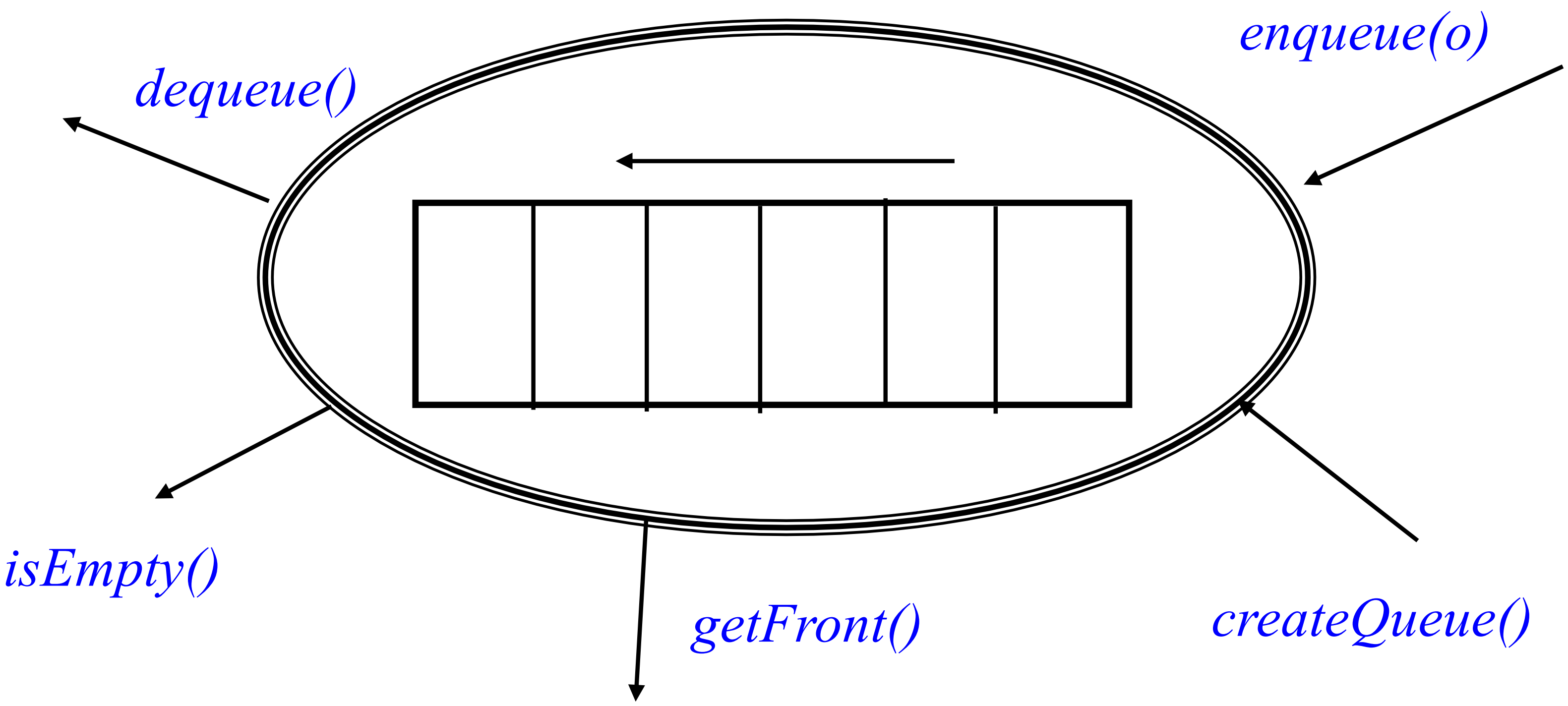
# What is a Queue?

- Like stacks, ***queues*** are lists. With a queue, however, insertion is done at one end whereas deletion is done at the other end.
- Queues implement the FIFO (**first-in first-out**) policy. E.g., a printer/job queue!
- Two basic operations of queues:
  - dequeue: remove an item/element from front
  - enqueue: add an item/element at the back



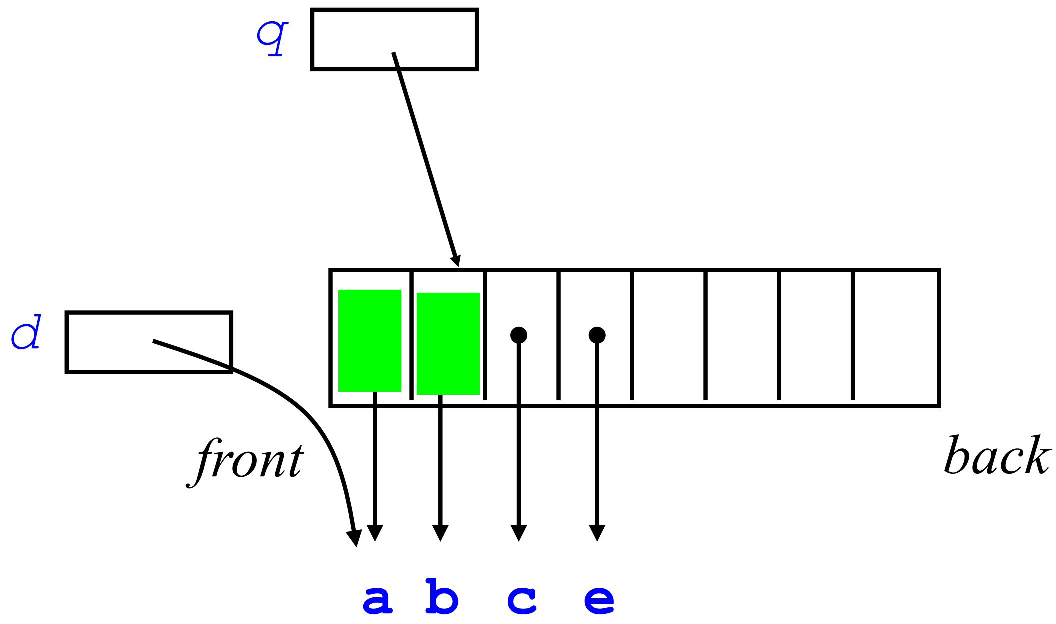
# Queue ADT

- Queues implement the FIFO (first-in first-out) policy
  - An example is the printer/job queue!



# Sample Operation

```
Queue *Q;
enqueue(Q, "a");
enqueue(Q, "b");
enqueue(Q, "c");
d=getFront(Q);
dequeue(Q);
enqueue(Q, "e");
dequeue(Q);
```



# Queue ADT interface

- The main functions in the Queue ADT are (Q is the queue)

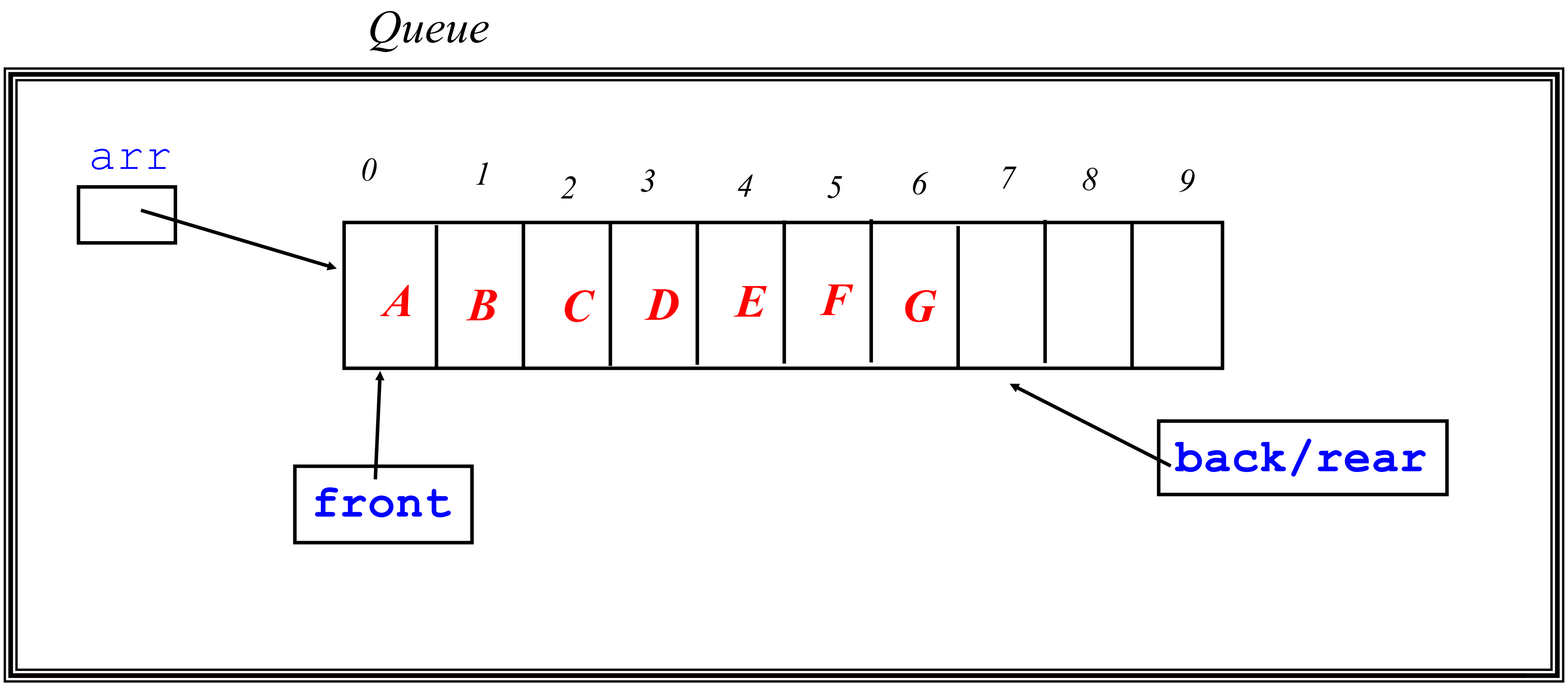
```
void enqueue(it, Q) // insert it to back of Q

void dequeue(Q); // remove oldest item

Item getFront(Q); // retrieve oldest item

boolean isEmpty(Q);// checks if Q is empty

boolean isFull(Q);// checks if Q is full

void clear(Q);// make Q empty

}
```

# Implementation of Queues

- Array-based queue
- Circular queue
- Linked queue

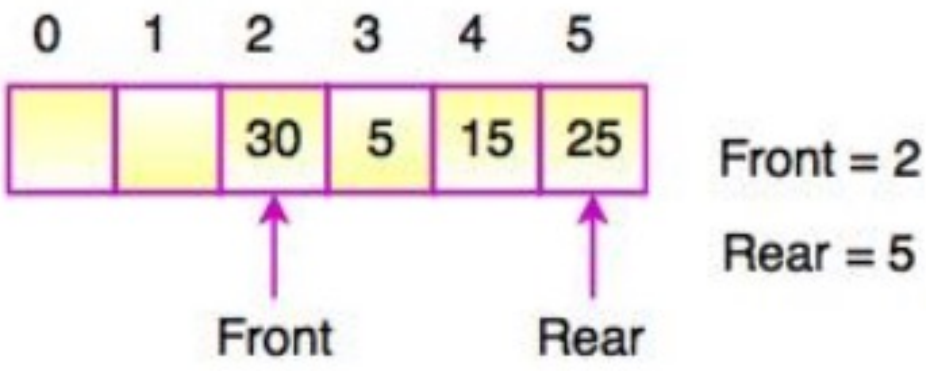
# Array-based Queue

- Use Array with front and back/rear pointers as implementation of queue



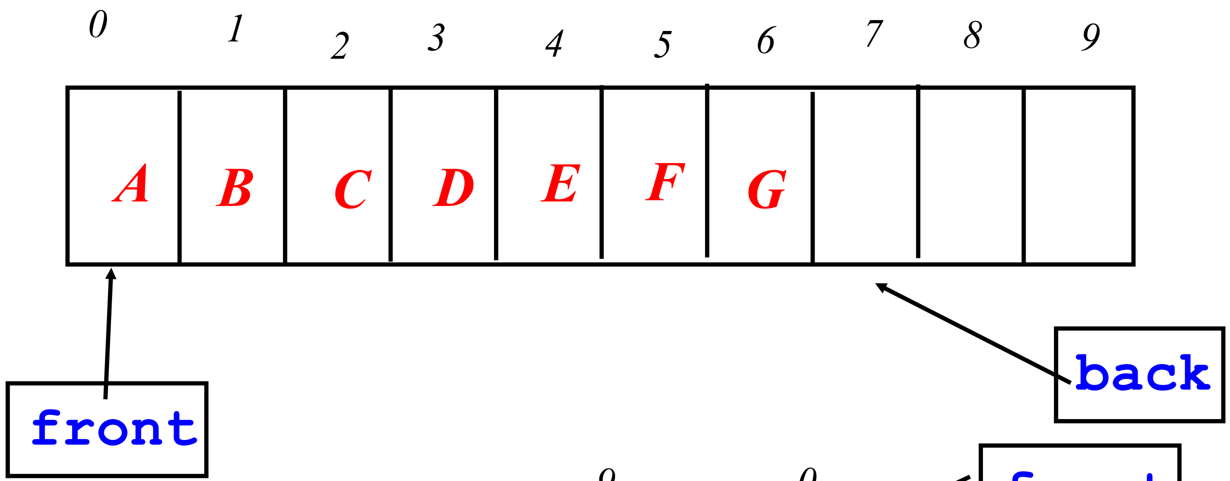
# Array-based Queue

- The queue `drift' towards to the end of the array
- Cannot enqueue when rear = (maxSize-1), even if there are some space left

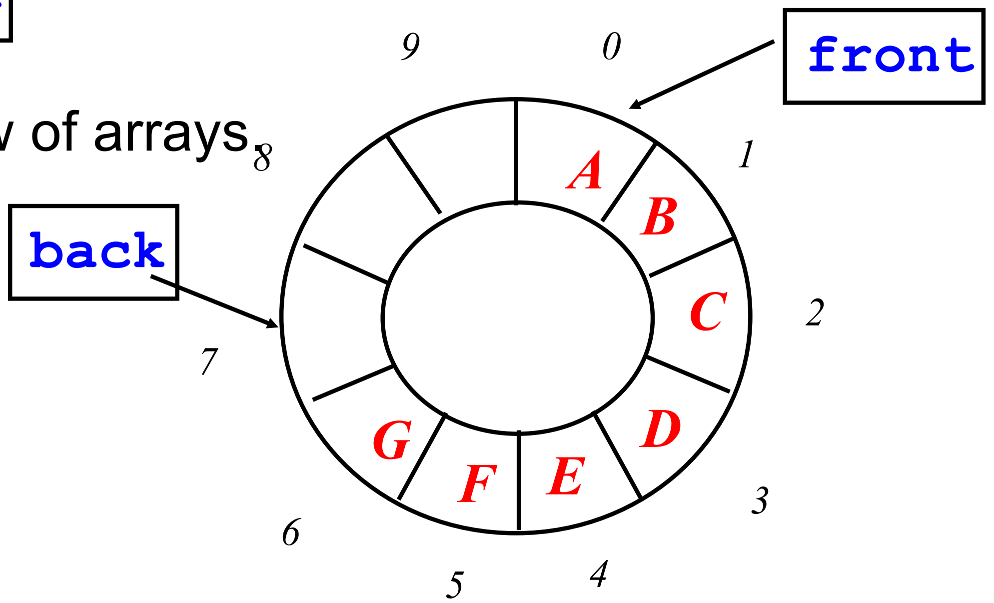


# Circular Queue

- To implement queue, it is best to view arrays as circular structure




Circular view of arrays.



# How to Advance

- Both front & back pointers should make advancement until they reach end of the array. Then, they should re-point to beginning of the array

```
front = adv(front);
back  = adv(back);
```

```
int adv(int p)
 { int r = p+1;
   if (r<maxsize) return r;
   else return 0;
 }
```

*upper bound of the array*

*Alternatively, use modular arithmetic:*

```
int adv(int p)
 { return ((p+1) % maxsize);
 }
```

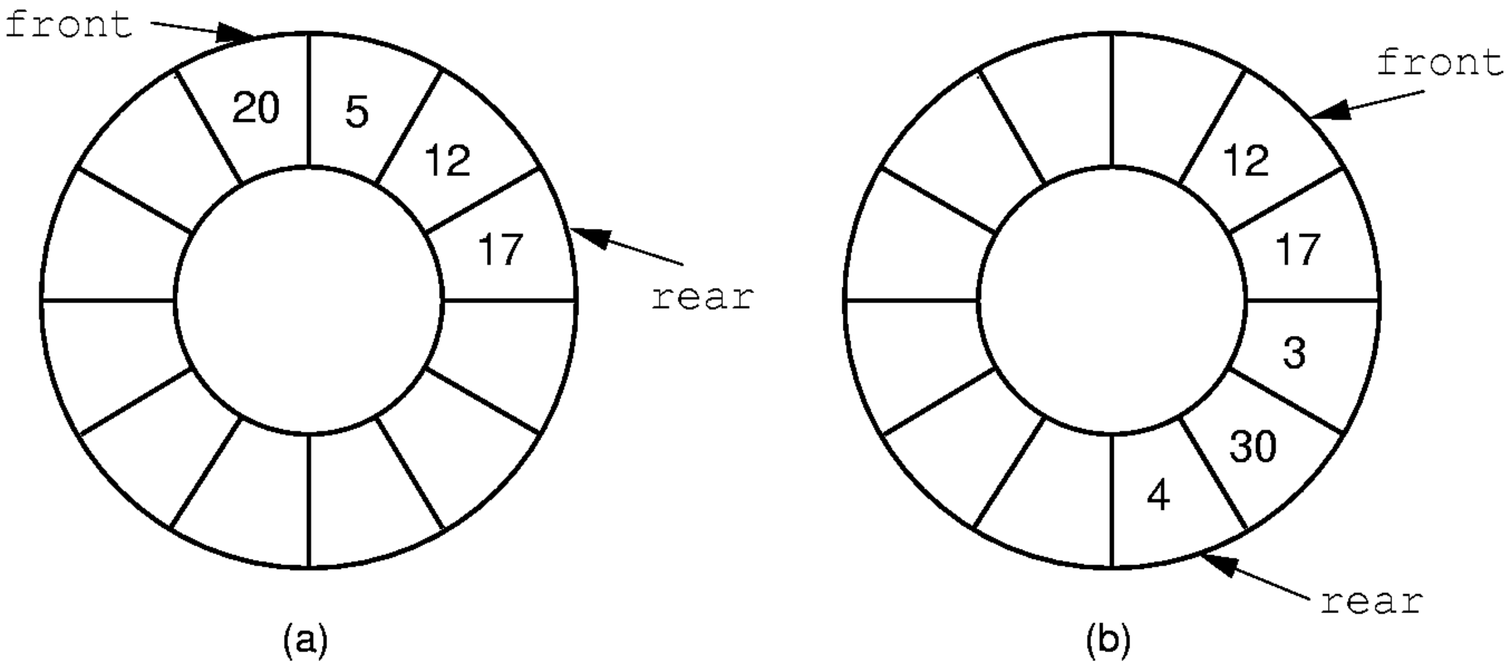
*mod operator*

# Circular Queue-cont.

- Enqueue
  - rear = (real+1)%maxSize;
  - Place the new element at the array with index rear
- Dequeue
  - Serve the first element in the queue, i.e., array[front]
  - front=(front+1)%maxSize;
- Initially
  - front = 0, rear = maxSize-1;

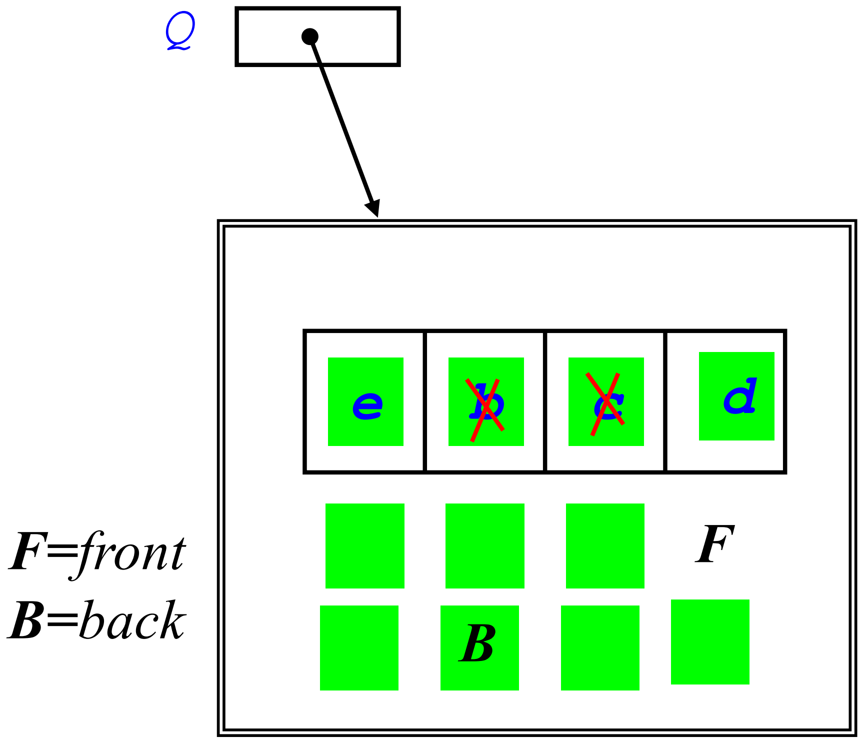
# Effects of Circular Queue

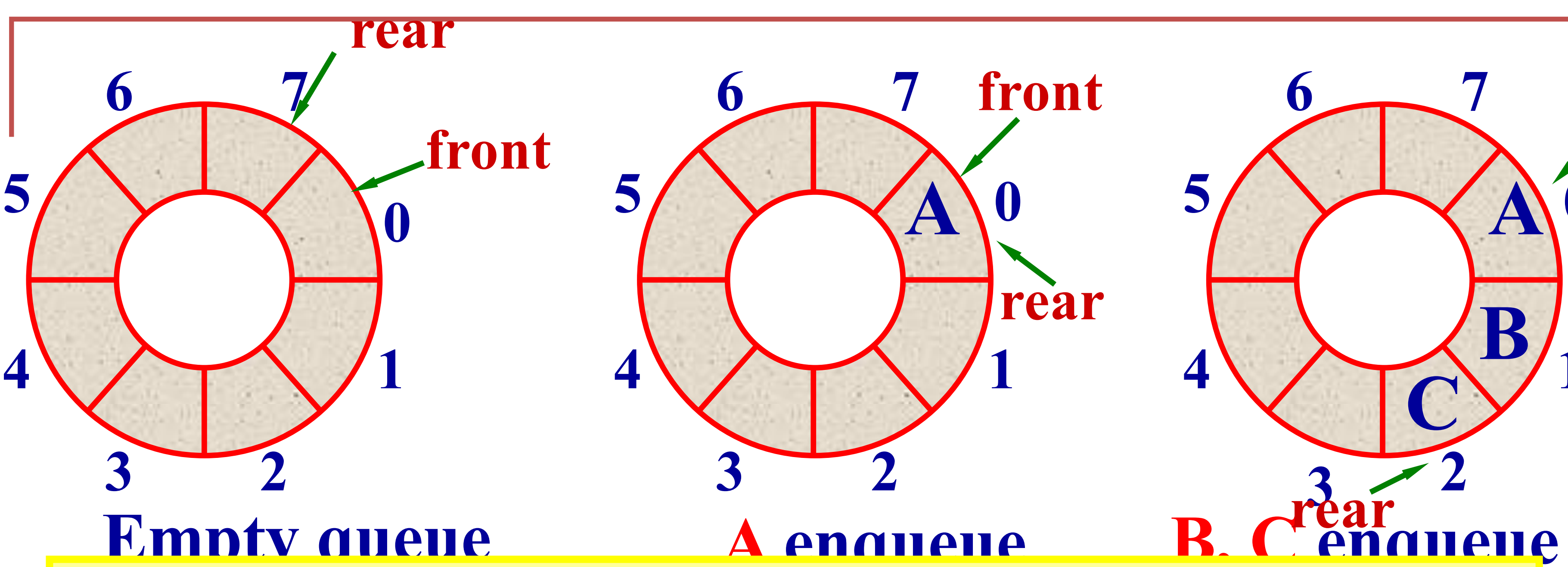
- The position of the element next to the i-th element is (i+1)%maxSize.
- `Length=(rear+size-front+1)%maxSize`
  - where % is the modulus operator.



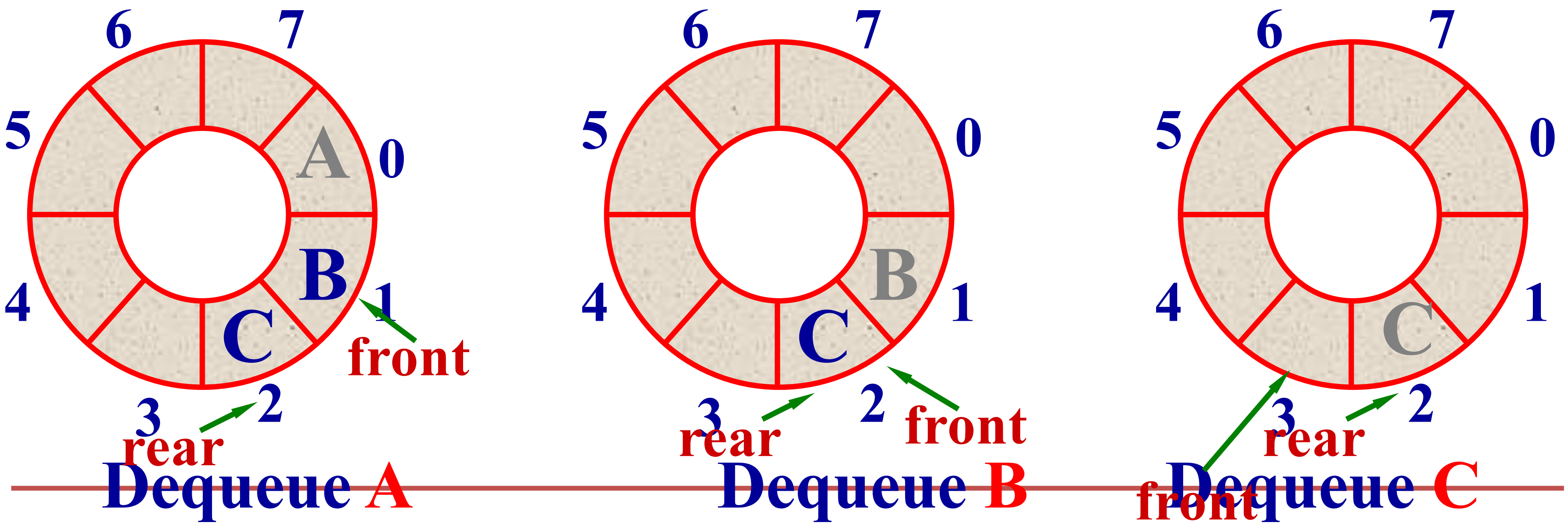
# Sample

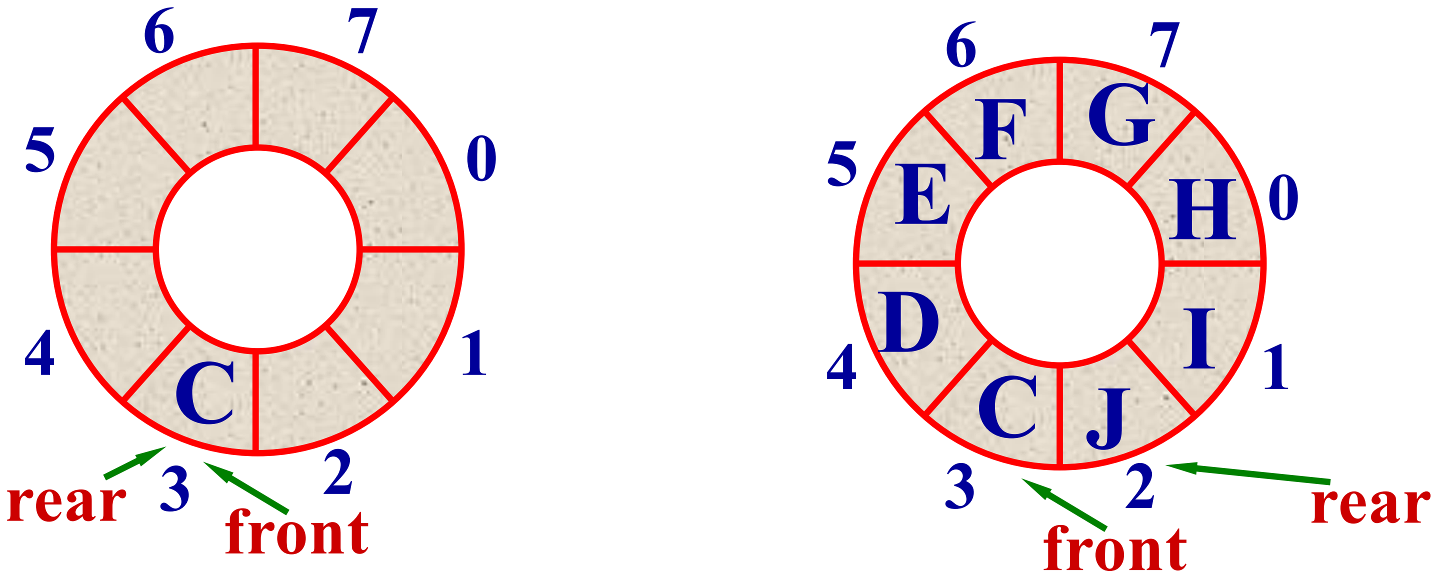
```
Queue *Q;
enqueue(Q, "a");
enqueue(Q, "b");
enqueue(Q, "c");
dequeue(Q);
dequeue(Q);
enqueue(Q, "d");
enqueue(Q, "e");
dequeue(Q);
```






**empty queue：(rear +1)%maxSize = front**






Enqueue D,E,F,G,H,I,J

**Full queue：(rear+1)%maxSize = front**

**Cannot distinguish an empty queue and a full queue !**

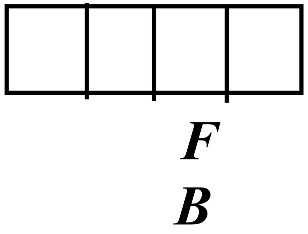
# An empty or a full queue?

- Solution 1: count how many elements in the queue
  - **Empty queue** if and only if the value of the counter is 0
  - **Full queue** iff the value of the counter is equal to the size of the array
- Solution 2: allocate an array with one more space for storing no more than *n* elements, i.e., the size of the array is *n+1*
  - The textbook adopts this solution.

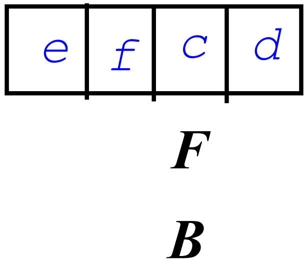
# Checking for Full/Empty State

What does (F==B) denote?

*Queue*
*Empty*
*State*




`size` **0**



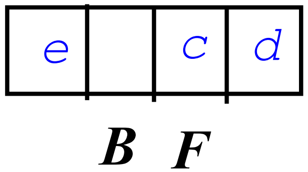

*Queue*
*Full*
*State*

`size` **4**

***Alternative - Leave a Deliberate Gap!***

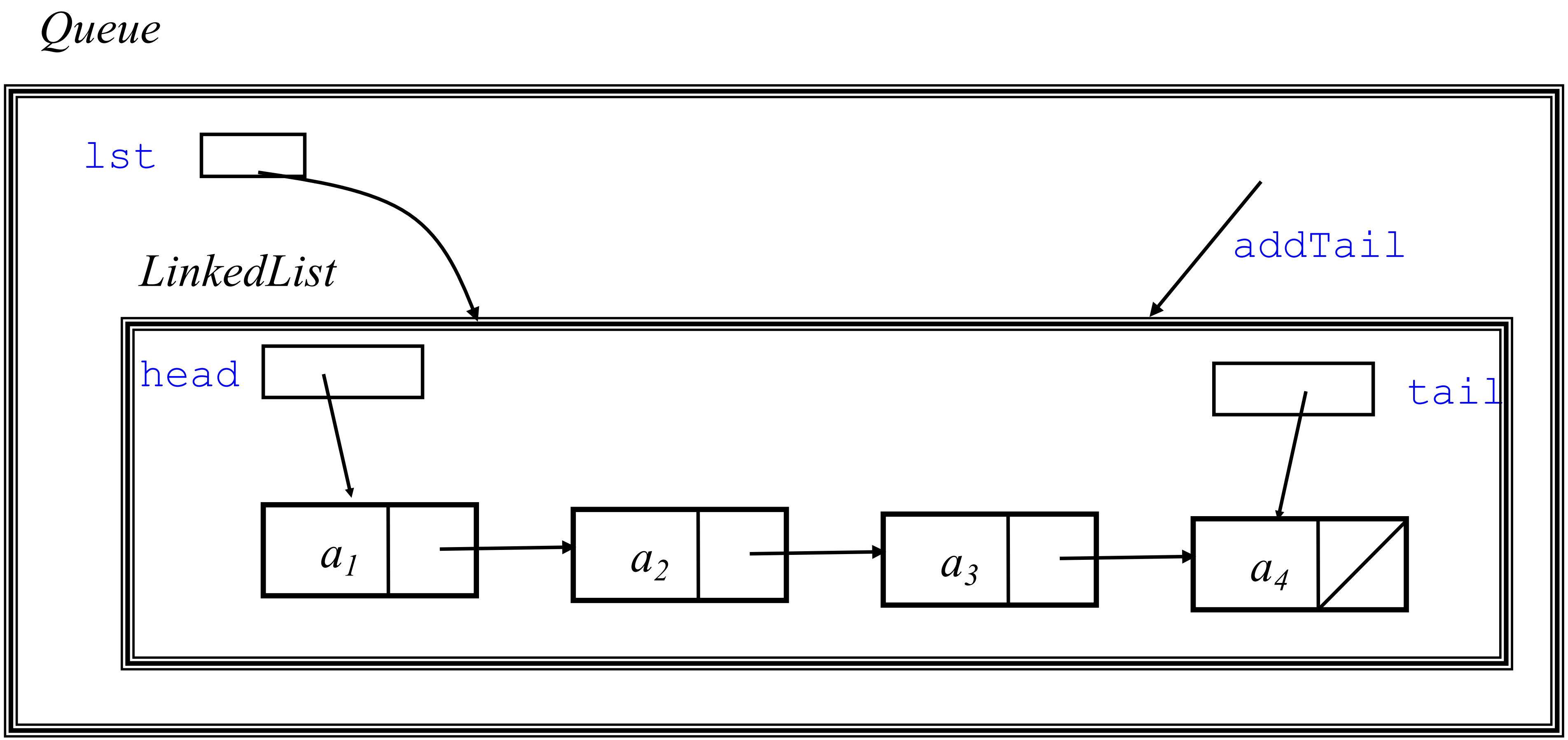
*No need for* `size` *field.*

*Full Case :* `(adv(B)==F)`



# Linked Queue

- Can use Linked Lists as underlying implementation of Queues



# Code

```
struct Node {
    int element;
    Node * next;
};

struct QUEUE {
    Node * front;
    Node * rear;
};
```

```
void clear(QUEUE *pQ)
{
    pQ->front = NULL;
}

BOOLEAN isEmpty(QUEUE *pQ)
{
    return (pQ->front == NULL);
}

BOOLEAN isFull(QUEUE *pQ)
{
    return FALSE;
}
```

# More code

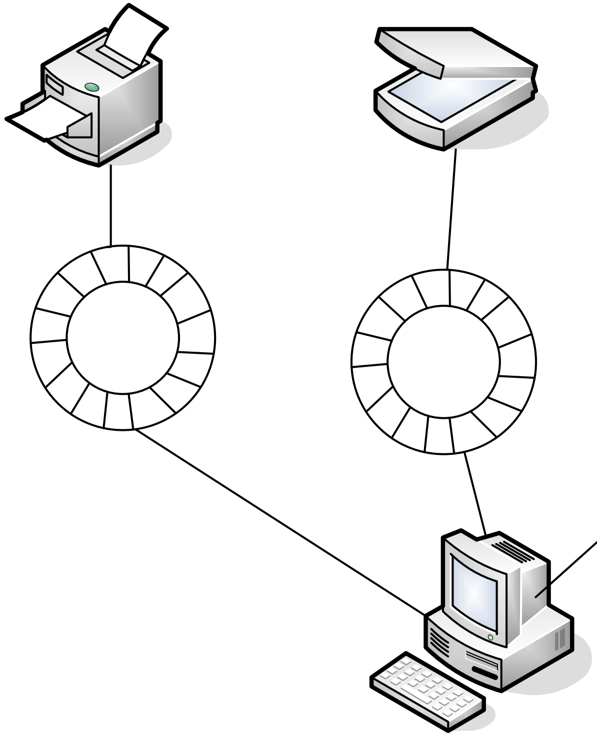
```
BOOLEAN dequeue(QUEUE *pQ, int *px)
{
    if (isEmpty(pQ))
        return FALSE;
    else {
        (*px) = pQ->front->element;
        pQ->front = pQ->front->next;
        return TRUE;
    }
}
```

# More code

```
BOOLEAN enqueue(int x, QUEUE *pQ)
{
    if (isEmpty(pQ)) {
        pQ->front = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->front;
    }
    else {
        pQ->rear->next = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->rear->next;
    }
    pQ->rear->element = x;
    pQ->rear->next = NULL;
    return TRUE;
}
```

CELL is a list node

# Application of Queue(1)- Buffer



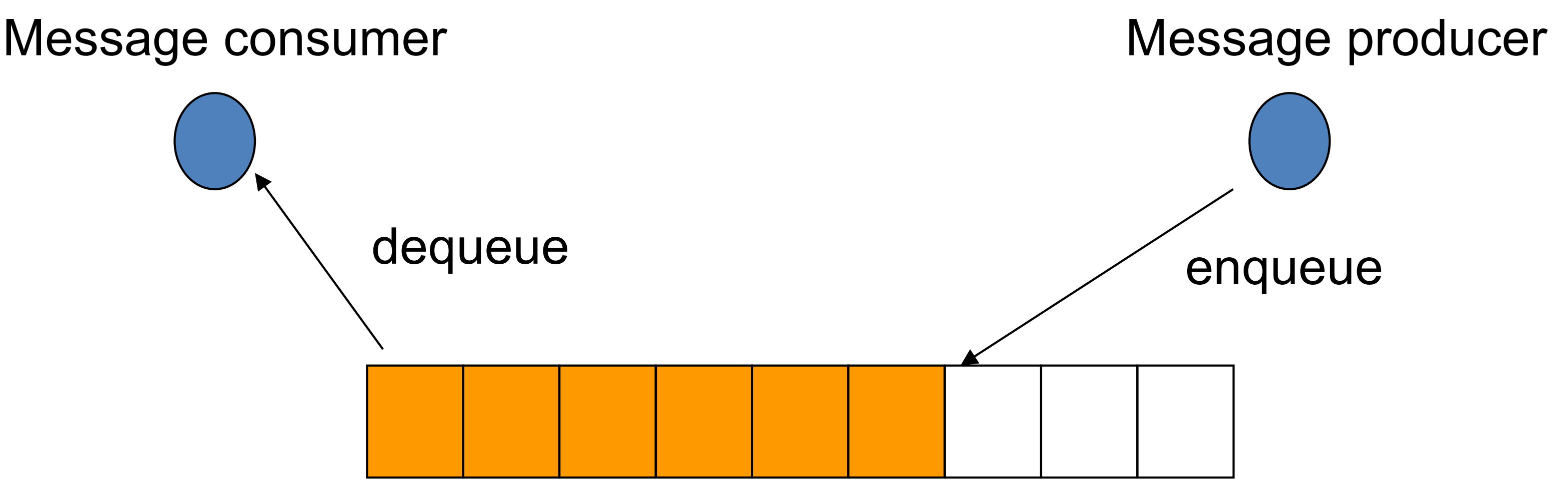
Low speed devices

Queues for data buffer

High speed processor

# Application of Queue(2)- Message Queue

- Asynchronous collaboration between different components.
  - E.g., message queue in Windows OS.





60

# Dictionaries

- A key-value pair
- ADT implementation
  - Array-based list
  - Linked list
- Operations
  - sorting
  - finding
  - inserting
  - removing

```
// The Dictionary abstract class.
template <typename Key, typename E>
class  Dictionary {
private:
  void operator =(const Dictionary&) {}
  Dictionary(const Dictionary&) {}

public:
  Dictionary() {}           // Default constructor
  virtual ˜Dictionary() {} // Base destructor

  // Reinitialize dictionary
  virtual void clear() = 0;

  // Insert a record
  // k: The key for the record being inserted.
  // e: The record being inserted.
  virtual void insert(const Key& k, const E& e) = 0;

  // Remove and return a record.
  // k: The key of the record to be removed.
  // Return: A maching record. If multiple records match
  // "k", remove an arbitrary one. Return NULL if no record
  // with key "k" exists.
  virtual E remove(const Key& k) = 0;

  // Remove and return an arbitrary record from dictionary.
  // Return: The record removed, or NULL if none exists.
  virtual E removeAny() = 0;

  // Return: A record matching "k" (NULL if none exists).
  // If multiple records match, return an arbitrary one.
  // k: The key of the record to find
  virtual E find(const Key& k) const = 0;

  // Return the number of records in the dictionary.
  virtual int size() = 0;
};
```

# Summary

- The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior
- The queue can be implemented by linked lists or by arrays
- There are many applications
  - Printer queues,
  - Telecommunication queues,
  - Simulations,
  - Etc.

# Conclusions

- Array-based lists
  - Fast random access
  - Insertion and removal take long time
- Linked lists
  - Slow for random access
  - Fast insertion and removal
- Singled and doubly linked list
  - The notion of curr
  - Add head and/or tail nodes for convenient coding
  - Pay attention to special cases

# Conclusions (cont'd)

- Stacks (LIFO, last-in first-out)
  - Two implementations
    - array-based and linked stacks
  - Fast operation with time complexity: $\Theta(1)$
- Queues (FIFO, first-in first-out)
  - Three implementations
    - Array-based, circular, and linked queue
  - Fast operation with time complexity: $\Theta(1)$
- Wide applications of stacks and queues

# Homework 2

- See course webpage
- Deadline: midnight before next lecture
- Submit to: cs_scu@foxmail.com
- File name format:
  - CS311_Hw2_yourID_yourLastName.doc (or .pdf)