COMP319 Algorithms 1
Lecture 6
ADT, Lists, Stacks, Queues

Instructor: Gil-Jin Jang

Abstract data type (ADT) / Array Lists / Linked Lists

Stacks / Queues

Textbook Chapter 11

Slide credits: 홍석원, 명지대학교, Discrete Mathematics, Spring 2013

김한준, 서울시립대학교, 자료구조 및 실습, Fall 2016

J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Abstract data type

ADT
ABSTRACT DATA TYPE

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Data Abstraction

- 컴퓨터를 이용한 문제해결에서의 추상화
  - 크고 복잡한 문제를 단순화시켜 쉽게 해결하기 위한 방법
- 자료 추상화(Data Abstraction)
  - 처리할 자료, 연산, 자료형에 대한 추상화 표현
  - 자료(data): 프로그램의 처리 대상이 되는 모든 것을 의미
- 연산(Operations)
  - 어떤 일을 처리하는 과정. 연산자에 의해 수행
  - 예) 더하기 연산은 +연산자에 의해 수행

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Data Abstraction

- 추상자료형(ADT, abstract data type)
  - 처리할 자료의 집합과 자료에 대해 수행할 연산자의 집합
  - 예) 정수 자료형
    - 자료 : 정수의 집합. {…, -1, 0, 1, …}
    - 연산자 : 정수에 대한 연산자 집합. {+, -, x, ÷, modulus}
- 추상화와 구체화
  - 추상화 – "무엇(what)인가?"를 논리적으로 정의
  - 구체화 – "어떻게(how) 할 것인가?"를 실제적으로 표현

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Abstract Data Type (ADT)

- Definition: a collection of *data* together with a set of *operations* on that data
  - specifications indicate *what* ADT operations do, but not *how* to implement them
  - data structures are part of an ADT's implementation

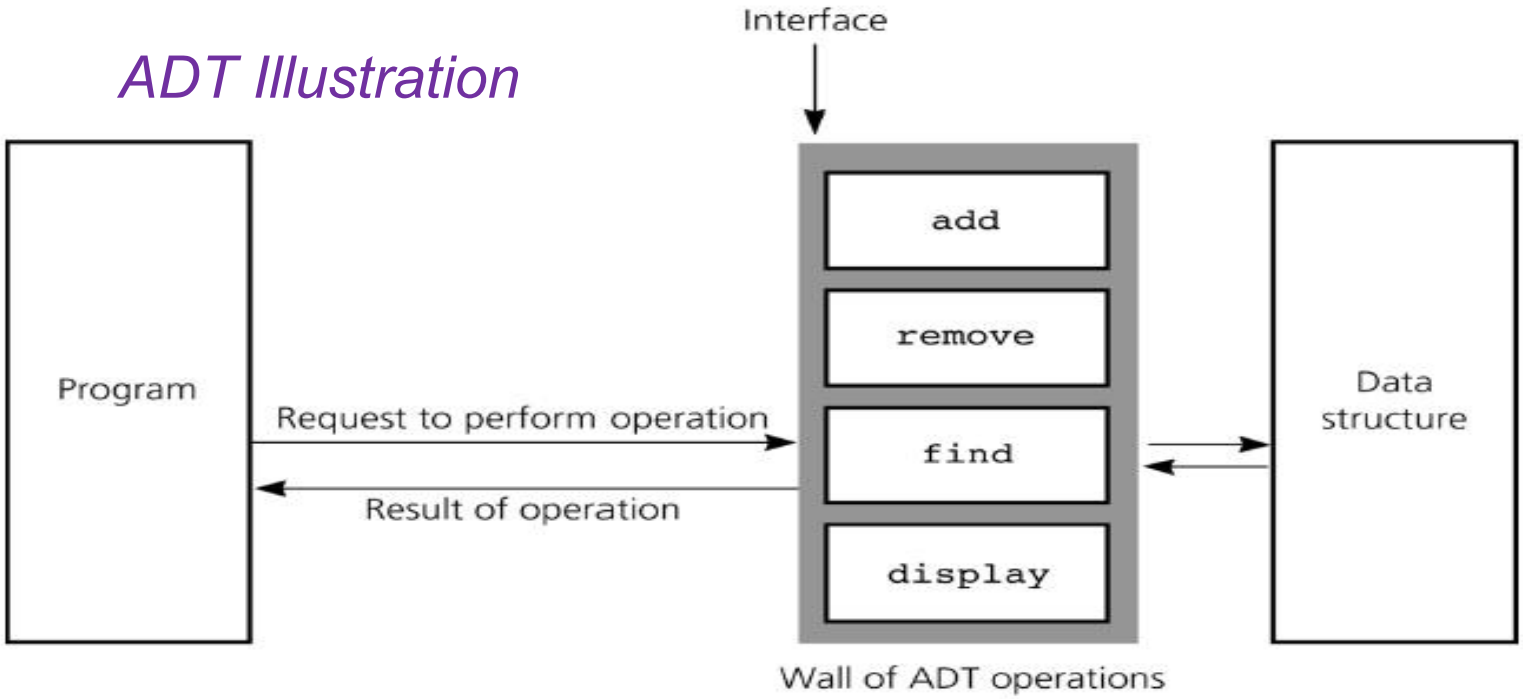
- Programmer can use an ADT without knowing its implementation.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Typical operations on Data

- Add data to a data collection
- Remove data from a data collection
- Ask questions about the data in a data collection
  - (Retrieval) What is the value at a particular location
  - (Search) Is ***x*** in the collection?

ADT Illustration



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Why ADT?

- Hide the unnecessary details
- Help manage software complexity
- Easier software maintenance
- Functionalities are less likely to change
- Localised rather than global changes

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

LISTS IMPLEMENTATION BY ARRAYS

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

An ADT Interface for Array List

- Functions
  - `isEmpty`
  - `getLength`
  - `Insert`
  - `Delete`
  - `Lookup`
  - …
- Data Members
  - `head`
  - `Size`

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Lists

- List: a finite sequence of data items

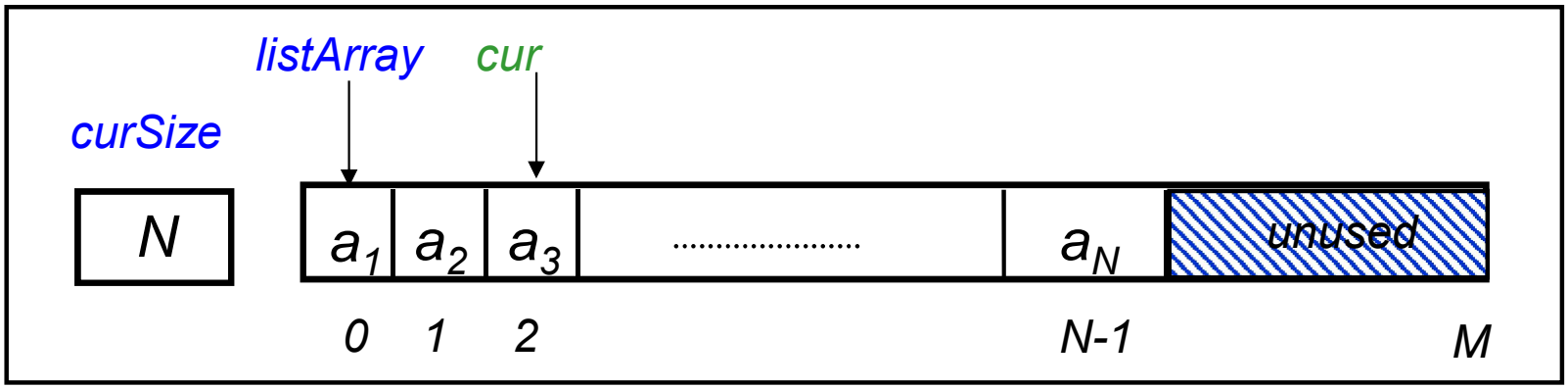
  $[a_1, a_2, a_3, \ldots, a_N]$

  [5, 6, 2, 8, …., 6]
- Lists are pervasive in computing
  - e.g. class list, list of characters (string), list of events
- Typical operations:
  1. Create a list
  2. Insert / Remove an element
  3. Test for emptiness
  4. Find an item / value / *k*-th element
  5. Retrieve current element / next / previous
  6. Print the entire list

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Array-Based List Implementation

- An array is usually a sequence of *N*-elements stored in consecutive memory locations
  - Maximum (allocation) size (*M*) is anticipated *a priori* (should be determined before allocated) and larger than array size (*N*)
- Internal variables:
  - Maximum size *maxSize* (*M*)
  - Current size *curSize* (*N*)
  - Current index *cur*
  - Array of elements listArray



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Coding: ArrayList Definition

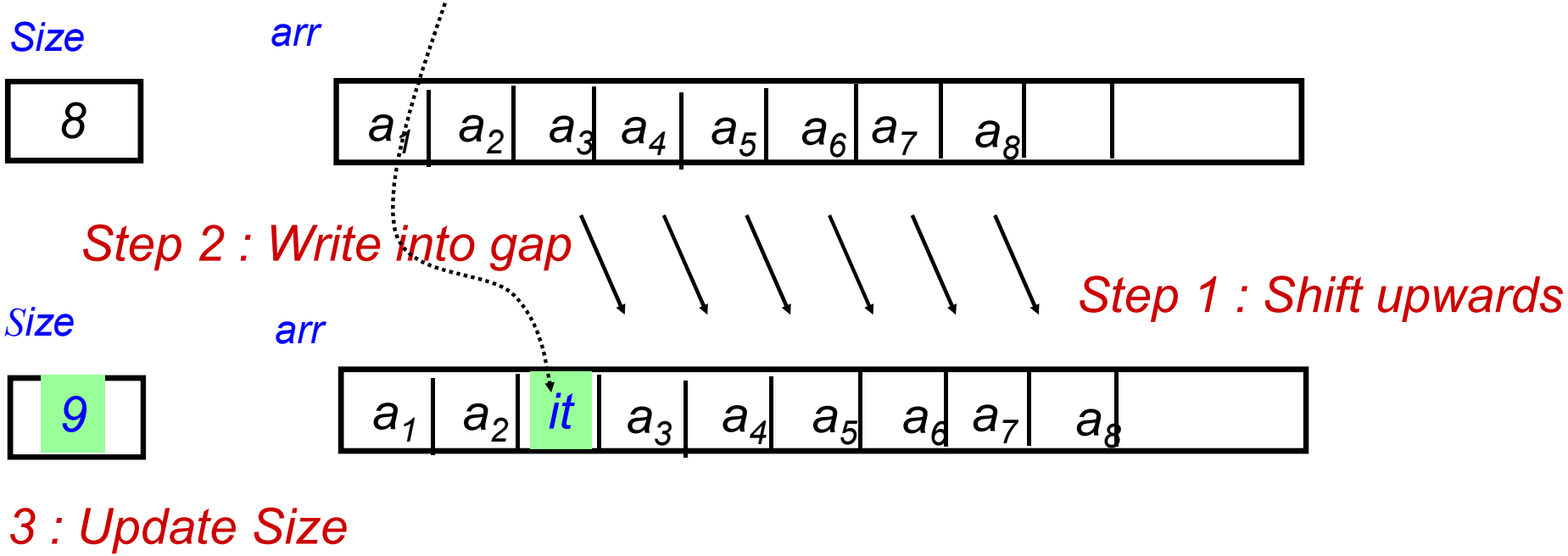
```
#define MAX 100    // in C, array size is unknown
                   // so programmers usually predict
                   // the maximum list size

struct ArrayList {
   int arr[MAX];
   int size;
};

void initialize(struct ArrayList *pl)
// start from empty list
{    pl->size = 0;  }

void print(struct ArrayList *pl)
{
   int i;
   printf("[");
   for (i=0; i<pl->size; i++) printf("%d ",pl->arr[i]);
   printf("]\n");
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Inserting Into an Array

- While retrieval is very fast, insertion and deletion are very slow
  - Function ***insert()*** has to shift upwards to create gap

Example : insert(2, it, arr)



Step 3 : Update Size

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Coding: insert()

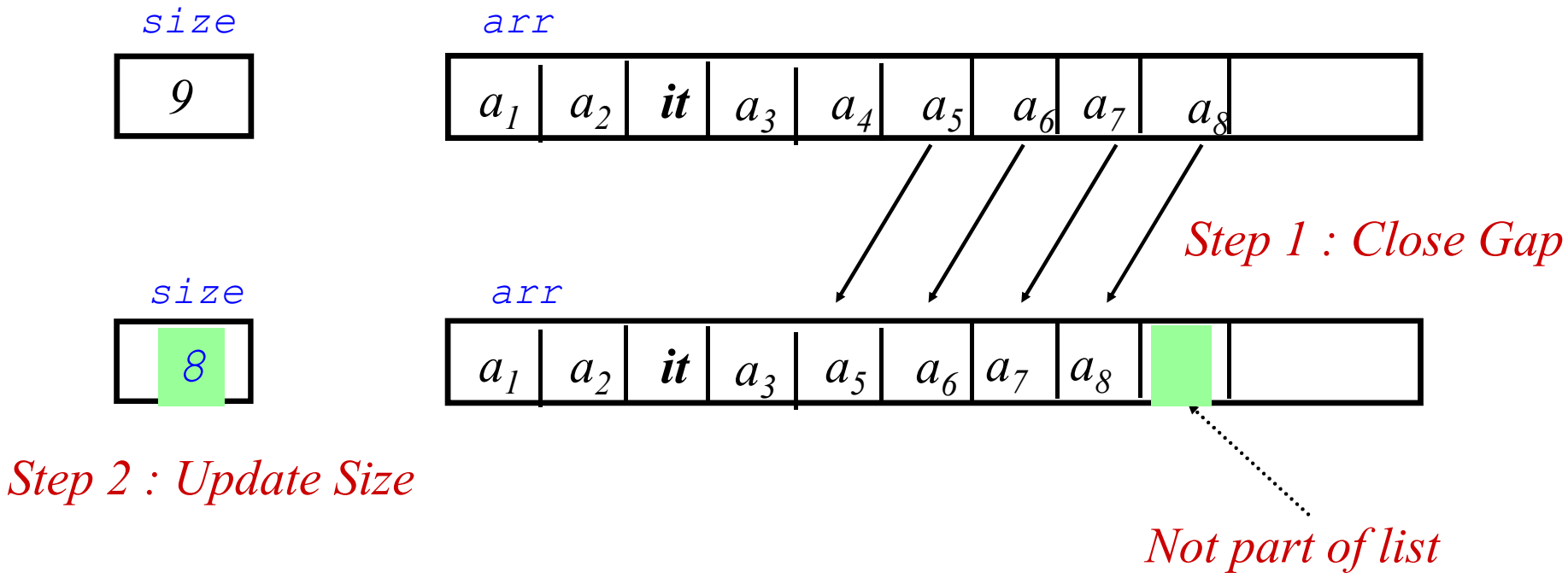
```
void insert(int j, int it, struct ArrayList *pl)
  { // precondition : 0<=j<=size
    // in this code, checking max is missing

      int i;

      for (i=pl->size-1; i>=j; i=i-1)
                                  // Step 1: Create gap
        { pl->arr[i+1]= pl->arr[i]; }

      pl->arr[j]= it;          // Step 2: Write to gap

      pl->size = pl->size + 1; // Step 3: Update size
  }
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Deleting from an Array

- Delete has to shift downwards to close gap of deleted item

Example: **`deleteItem(4, arr)`**



Coding: delete()

```
void delete(int j, struct ArrayList *pl)
{       // precondition : 0<=j<size
   int i;

   for (i=j+1; i<pl->size; i=i+1)
                // Step1: Close gap
     { pl->arr[i-1]=pl->arr[i]; }
                // Step 2: Update size
     pl->size = pl->size - 1;
   }
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Coding: lookup()

```
#define TRUE 1
#define FALSE 0
typedef int BOOLEAN;
BOOLEAN lookup (int x,
    struct ArrayList *pl)
{
  int i;
  for (i=0; i<pl->size; i=i+1) {
    if (x == pl->arr[i]) return TRUE;
  }
  return FALSE;
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

LINKED LISTS

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

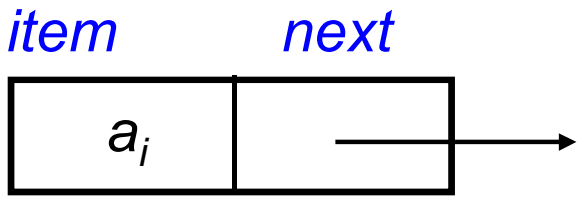
An ADT Interface for Linked List

- Functions
  - `isEmpty`
  - `getLength`
  - `Insert`
  - `Delete`
  - `Lookup`
  - …
- Data Members
  - `head`
  - `Size`
- **Local variables to member functions**
  - **`cur`**
  - **`prev`**

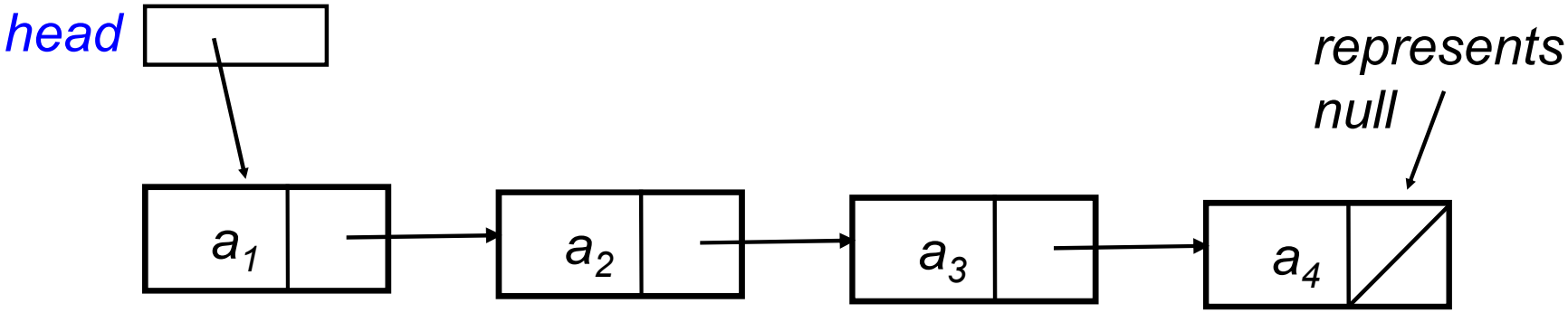
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Linked List Approach

- Main problem of array is the slow deletion/insertion since it has to shift items in its *contiguous* memory
- **Solution**: linked list where items need *not be contiguous* with nodes of the form



- Sequence (list) of four items $< a_1, a_2, a_3, a_4 >$ can be represented by:

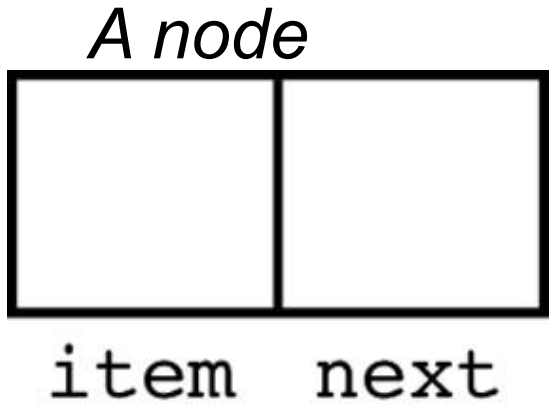


Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Pointer-Based Linked Lists

- A node in a linked list is usually a `struct`

```
struct Node
{ int item
  struct Node *next;
}; //end struct
```



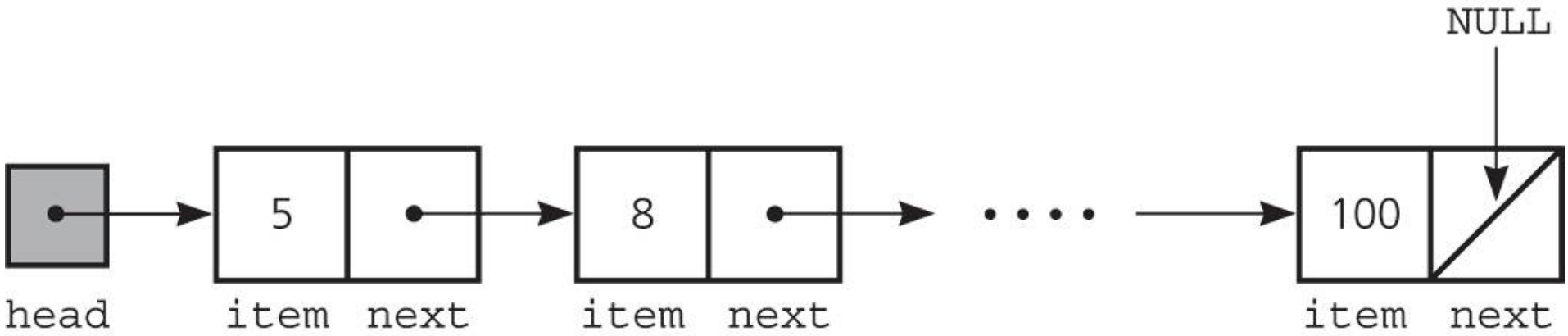
- A node is dynamically allocated

```
struct Node *p;
p = malloc(sizeof(struct Node));
```

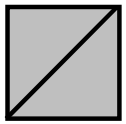
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

A Sample Linked List

- The head pointer points to the first node



- If head is *NULL*, the linked list is empty
  - `head=NULL`



represents empty list

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

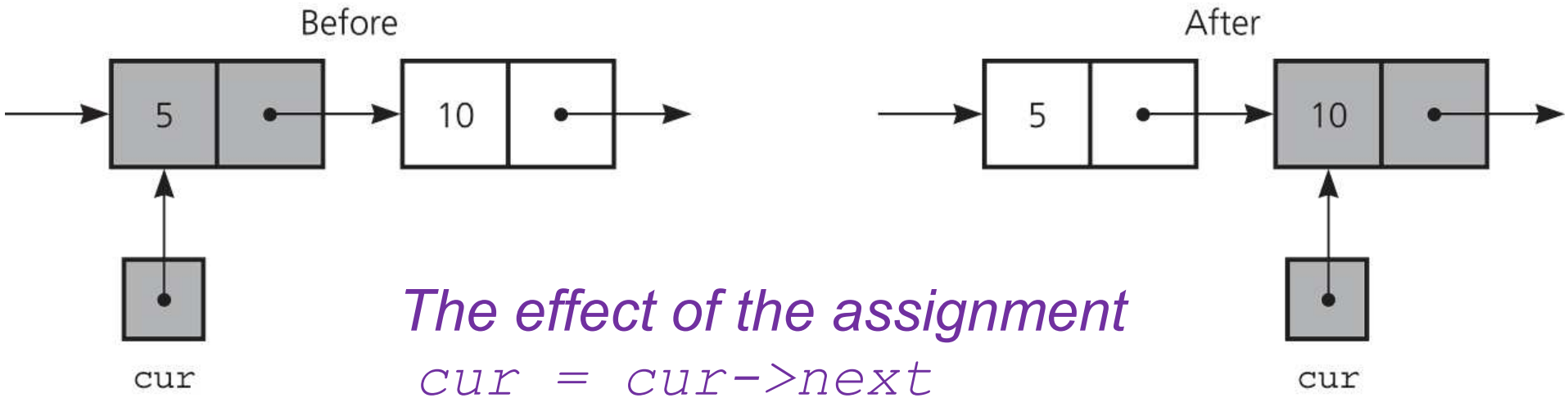
Traverse a Linked List

- Reference a node member with the -> operator

```
p->item;
```

- A traverse operation visits each node in the linked list
  - A pointer variable cur keeps track of the current node

```
struct Node *cur;
for (cur=head; cur!=NULL; cur = cur->next)
        printf("%d ",cur->item);
```



The effect of the assignment

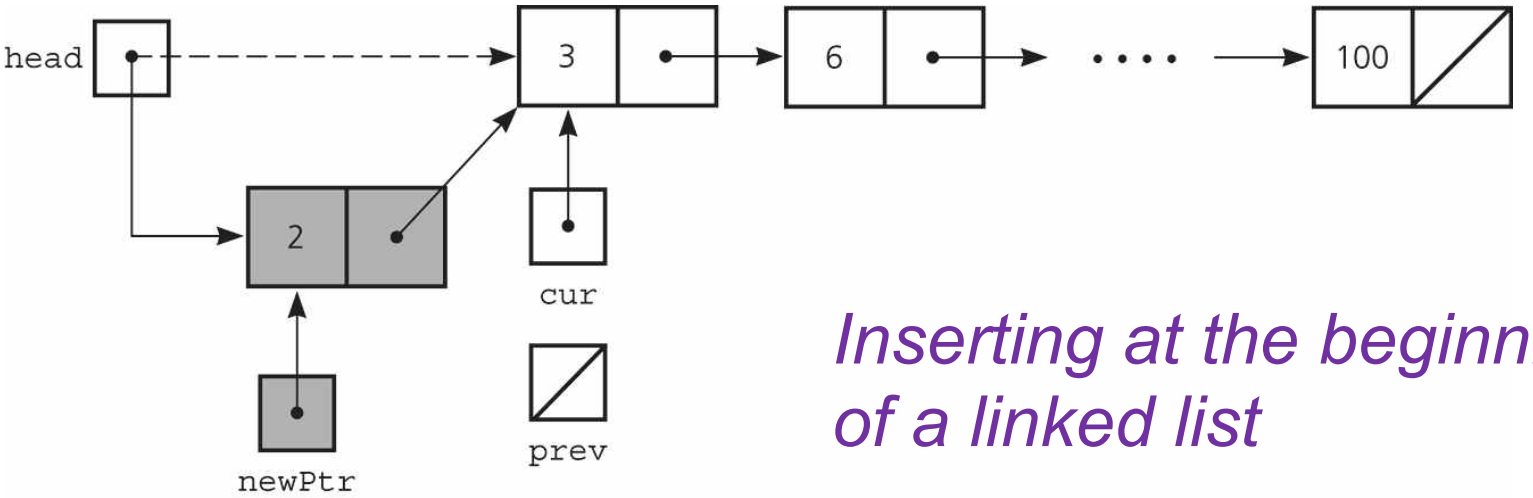
`cur = cur->next`

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Insert a Node into a Linked List

- To insert a node at the **beginning of a linked list**

```
newPtr->next = head;
head = newPtr;
```



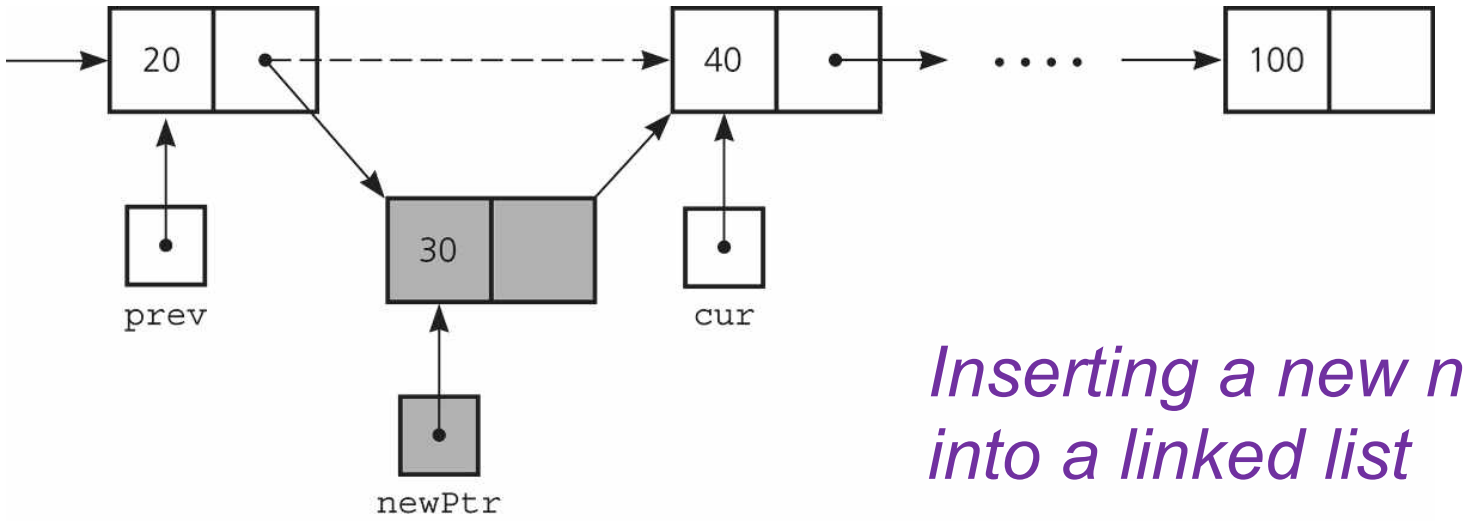
Inserting at the beginning of a linked list

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Insert a Node into a Linked List

- To insert a node **between two nodes**

```
newPtr->next = cur;
prev->next = newPtr;
```

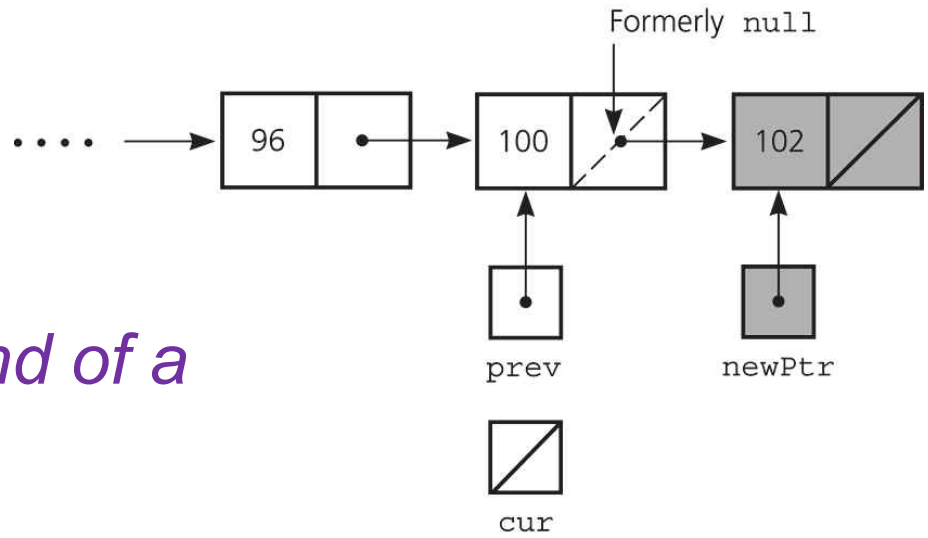


Inserting a new node into a linked list

Insert a Node into a Linked List

- Inserting at the **end of a linked list** (if `cur` is *NULL*) is not a special case

```
newPtr->next = cur;
prev->next = newPtr;
```



Inserting at the end of a linked list

Delete a Node from a Linked List

- Deleting an interior/last node with returning the deleted node to system

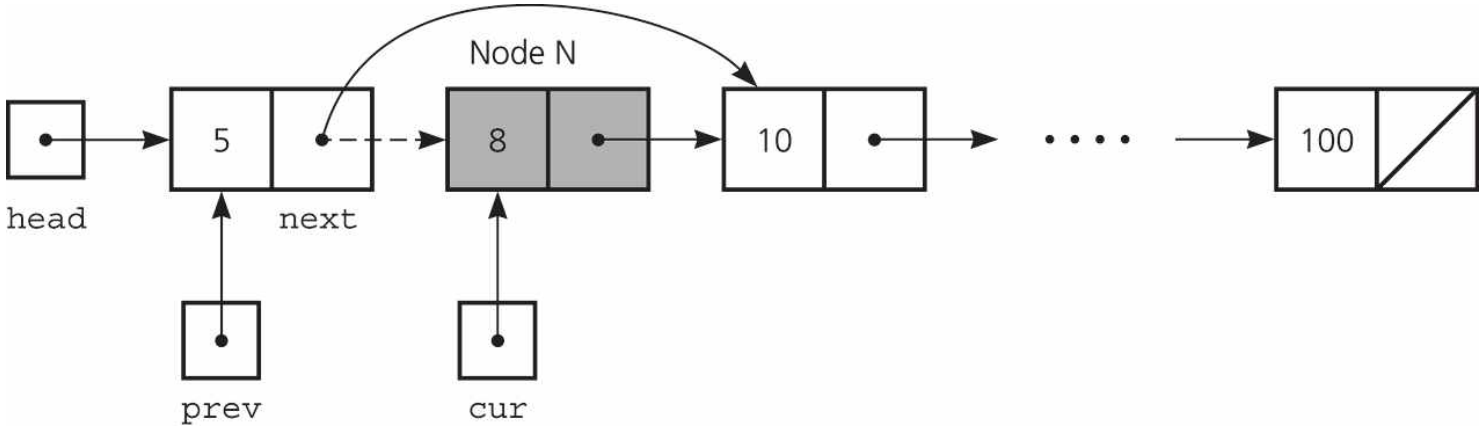
```
prev->next=cur->next;
cur->next = NULL;      // not necessary
free(cur);
cur = prev->next;      // only if keep cur
```

- Deleting the first node with returning the deleted node to system

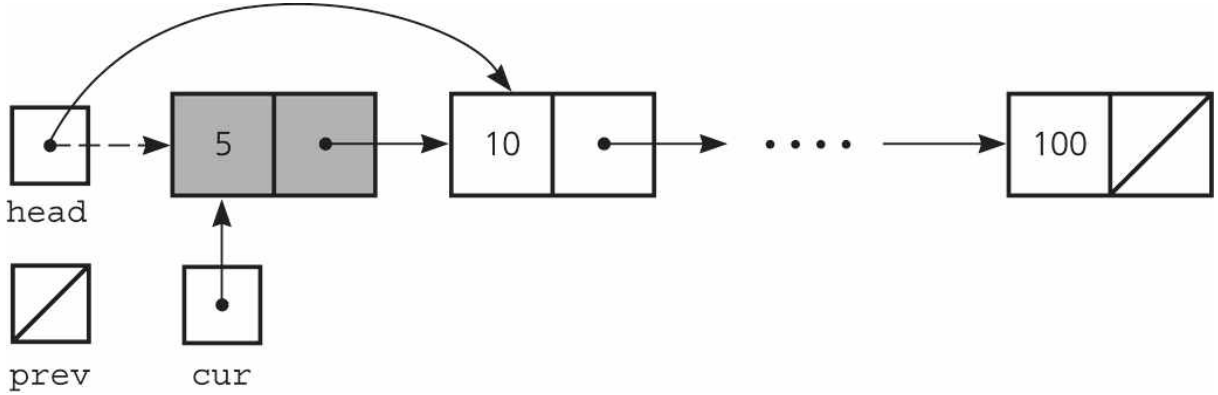
```
cur = head;
head = head->next;
cur->next = NULL;      // not necessary
free(cur);
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Delete a Node from a Linked List



Deleting a node from a linked list



Deleting the first node

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Look up

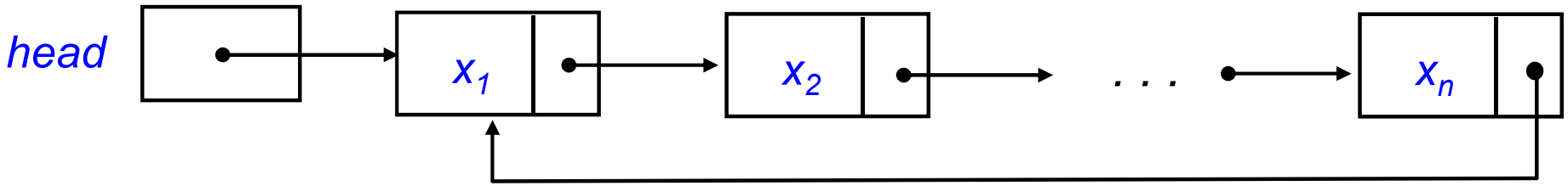
```
struct Node* lookup_recursive (int x, struct Node *L)
{  if (L == NULL) return NULL;
     else if (x == L->item) return L;
     else return lookup_recursive (x, L->next);
}
Struct Node* lookup_iterative (int x, struct Node *L)
{  struct Node *cur;
   for (cur=head; cur!=NULL; cur = cur->next)
          if ( x == cur->item ) return cur;
   return NULL; // not found
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Circular Linked Lists

- May need to cycle through a list repeatedly, e.g. round robin system for a shared resource
- *Solution* : Have the last node point to the first node

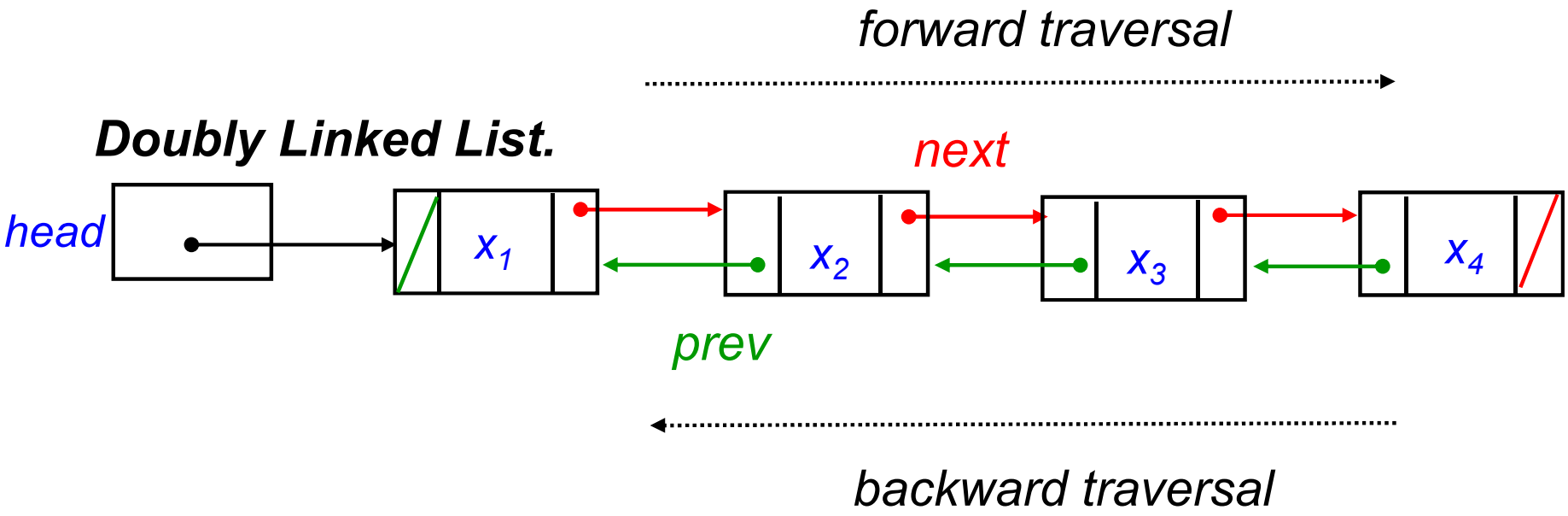
Circular Linked List.



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Doubly Liked Lists

- Frequently, we need to traverse a sequence in BOTH directions efficiently
- *Solution* : Use doubly-linked list where each node has two pointers



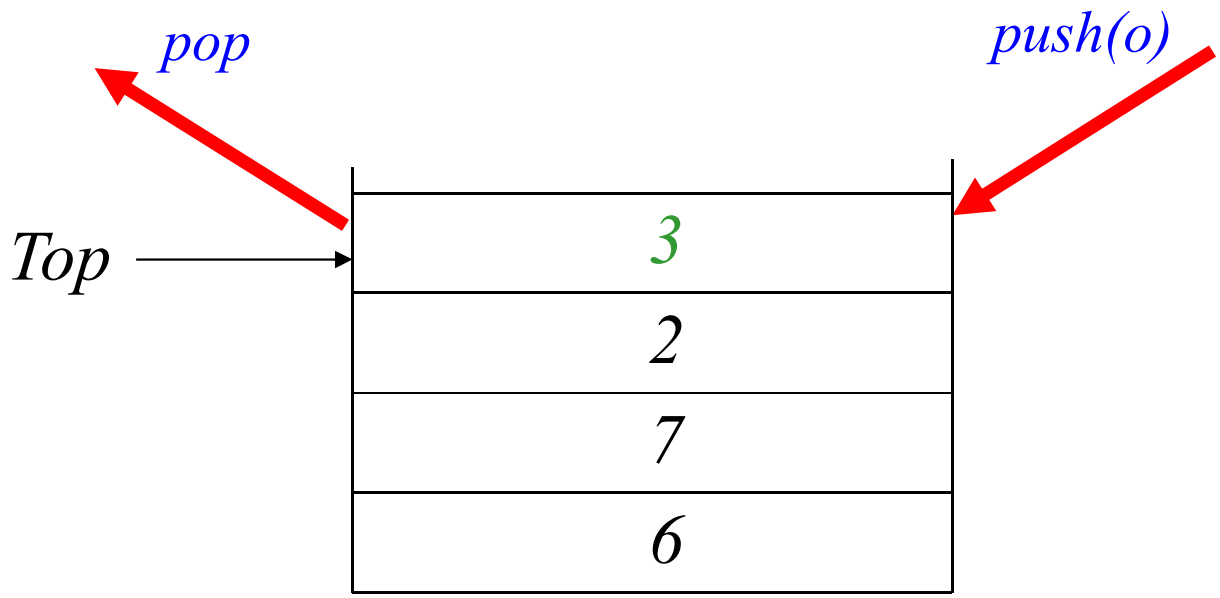
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

STACKS

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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***.
- The operations: **push** (insert) and **pop** (delete)
  - Other operations: top, clear, etc.



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Stack ADT Interface

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

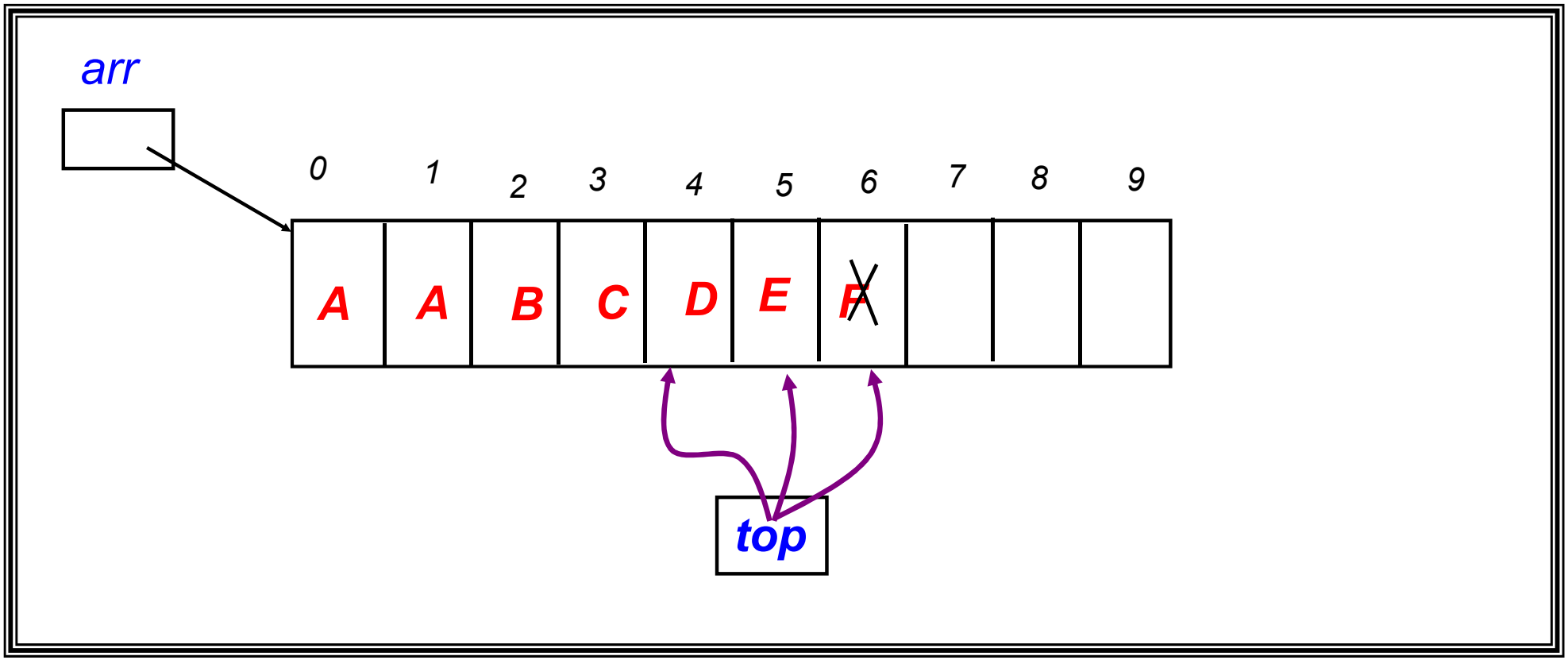
```
boolean isEmpty(S);        // return true if empty
boolean isFull(S);         // return true if full
void push(S, item);        // insert item into stack
Item pop(S);               // return and remove most recent item
void clear(S);             // remove all items from stack
Item top(S);       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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Implementation by Array

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

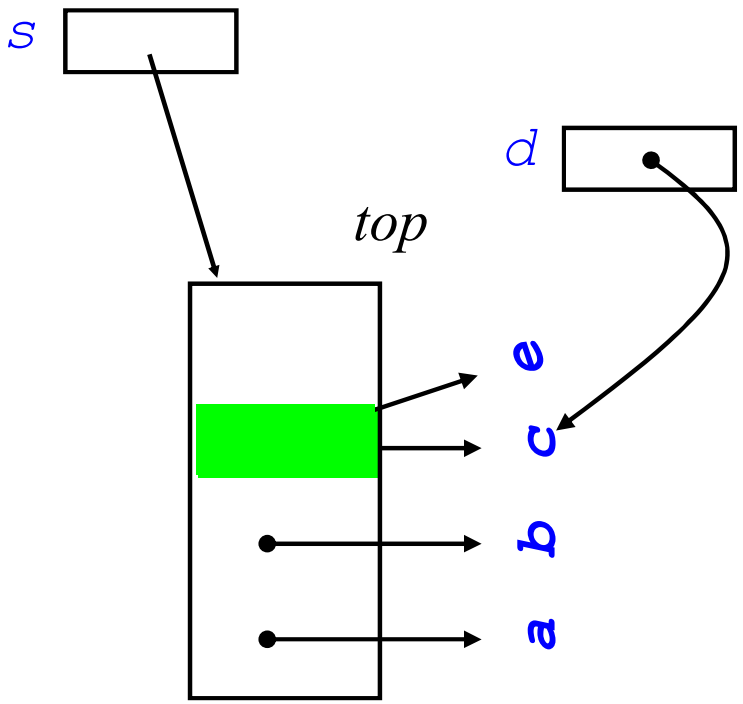
StackAr



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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);
```



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Stack Implementation in C

```
typedef short BOOLEAN;
#define TRUE  ((short)(1))
#define FALSE ((short)(0))

/* maximum stack size */
#define MAX (1024)

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);
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Stack in C: push and pop

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

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Applications

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Stack Example: Line Editing

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

Input : abc_defgh←←2klpq←←←wxyz

Corrected Input : abc_defg2klpwxyz

Reversed Output : zyxwplk2gfed_cba

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

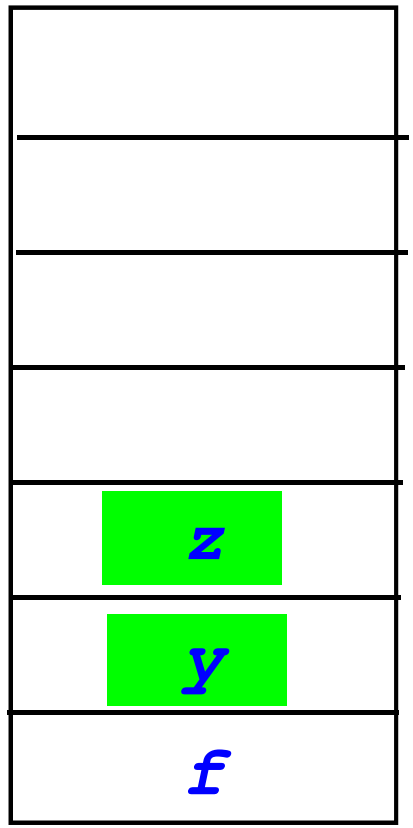
The Procedure

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

Input : `fgh←r←←yz`

Corrected Input : `fyz`

Reversed Output : `zyf`



Stack

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Informal Procedure

Initialize the stack to empty

For every char read

if open bracket then *push onto stack*

if close bracket, then

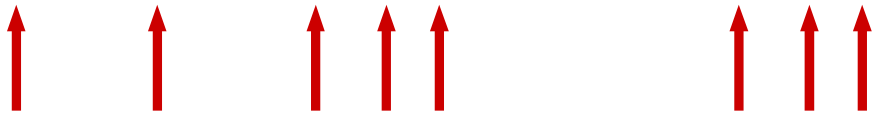
return & remove most recent item from *the stack*

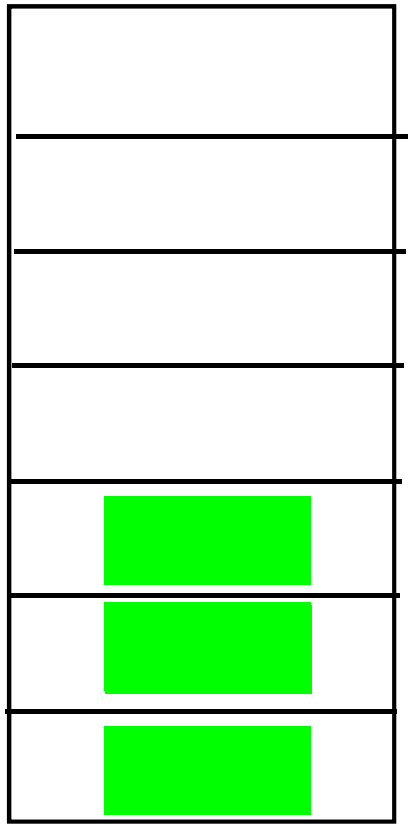
if doesn't match then *flag error*

if non-bracket, *skip the char read*

Example

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



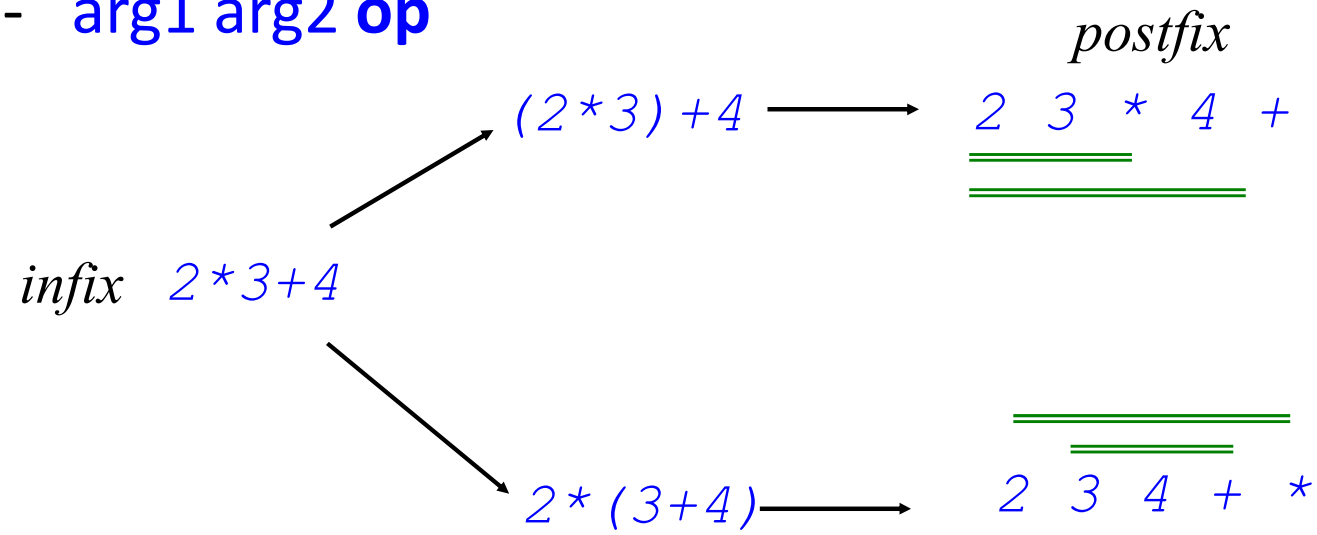


Stack

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**



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

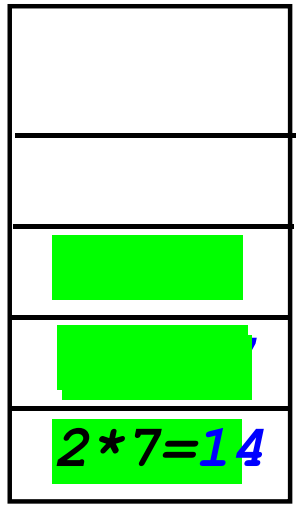
Informal Procedure

Initialise stack S
For each item read.
    If it is an operand,
        push on 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=pop(S)`
	`arg1=pop(S)`
	`push(S, arg1+arg2)`

*	`arg2=pop(S)`
	`arg1=pop(S)`
	`push(S, arg1*arg2)`



Stack

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

STACKS: LINKED LIST IMPLEMENTATION

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Recap: Stack ADT Interface

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

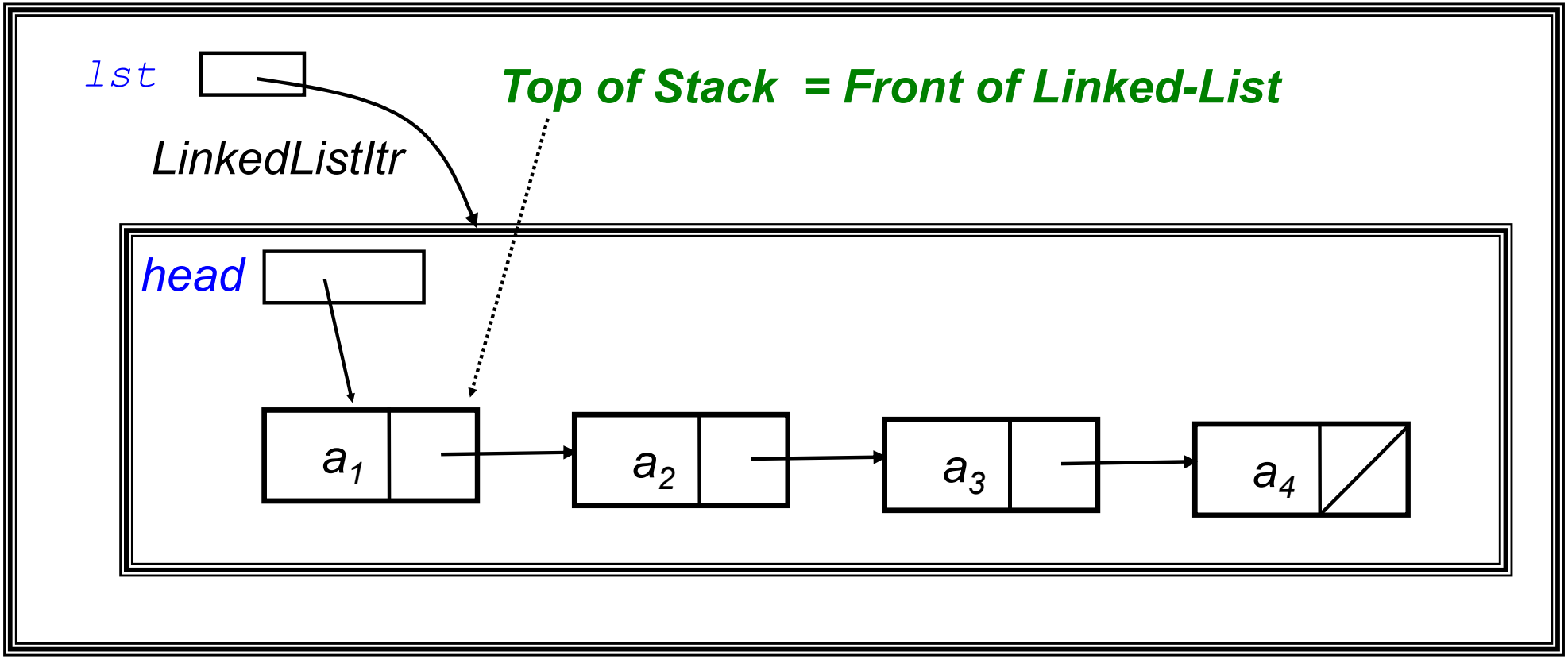
```
boolean isEmpty(S);        // return true if empty
boolean isFull(S);         // return true if full
void push(S, item);        // insert item into stack
Item pop(S);               // return and remove most recent item
void clear(S);             // remove all items from stack
Item top(S);       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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Implementation by Linked Lists

- Can use a **Linked List** as implementation of stack

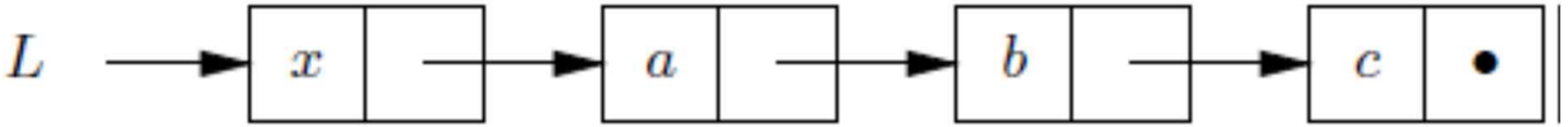
StackLL



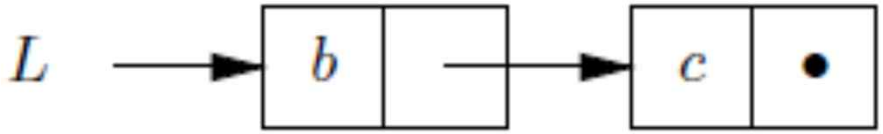
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Illustration of push and pop

(a) List L.



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



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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Stack by Linked Lists in C

```
typedef short BOOLEAN;
#define TRUE  ((short)(1))
#define FALSE ((short)(0))

struct NODE {
   int element;
   struct NODE *next;
};

typedef struct {
   struct NODE *head;
} STACK;
```

```
void initialize(STACK *pS)
{ pS->head = NULL; }

void clear(STACK *pS) {
   struct NODE *n, *pre;
   n = pS->head;
   while ( n ) {
      pre = n;
      n = n->next;
      free(pre); }
   initialize(pS);
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Stack by Linked Lists in C: pop

```
BOOLEAN isEmpty(STACK *pS) { return (pS->head == NULL); }
BOOLEAN isFull(STACK *pS) {
  // list grows as much as memory size, so no full stack
  return FALSE;
}

BOOLEAN pop(STACK *pS, int *px) {
  struct NODE *n;
  if ( isEmpty(pS) ) return FALSE;
  else {
    n = pS->head; pS->head = n->next;
    (*px) = n->element; free(n);
    return TRUE;
  }
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Stack by Linked Lists in C: push

```
BOOLEAN push(int x, STACK *pS) {
  struct NODE *n;
  n = (struct NODE*) malloc(sizeof(struct NODE));
  n->element = x;
  n->next = pS->head;
  pS->head = n;
  return TRUE;
}
```

```
/* EXAMPLE
[37 75 45 80 24 13 94 42 21 74 ]
37 [75 45 80 24 13 94 42 21 74 ]
75 [45 80 24 13 94 42 21 74 ]
[102 45 80 24 13 94 42 21 74 ]
[106 102 45 80 24 13 94 42 21 74 ]
106 [102 45 80 24 13 94 42 21 74 ]
*/
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Summary

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

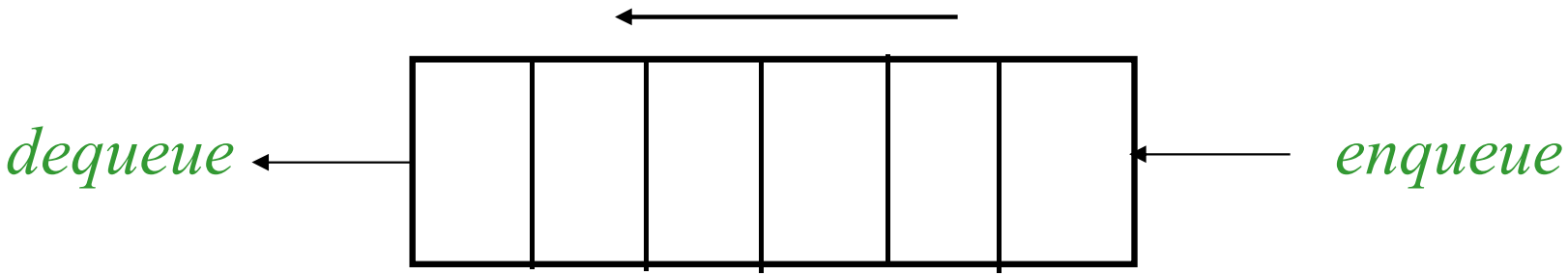
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

QUEUES

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

What is a Queue?

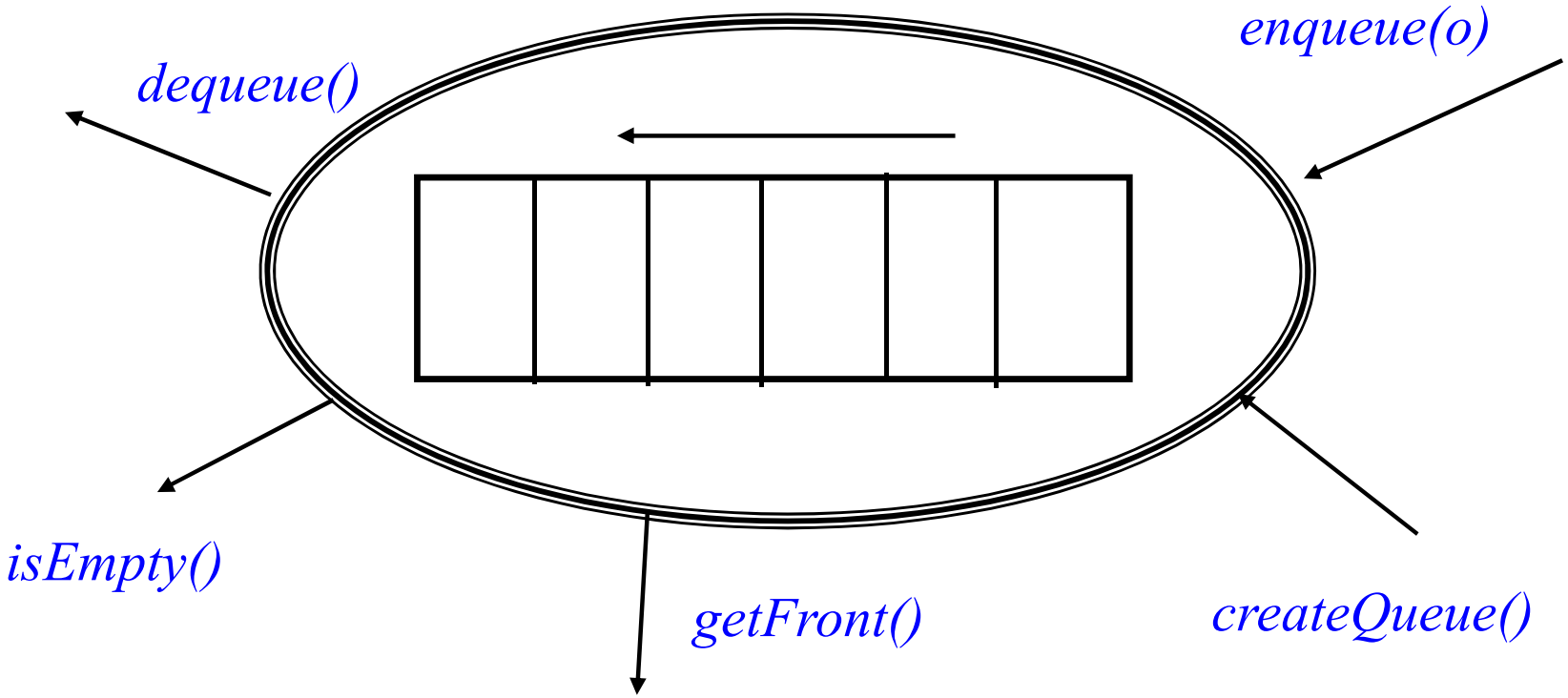
- Like stacks, queues are lists. With a queue, however, insertion is done at one end whereas deletion is done at the other end.
- Queues implement the FIFO (first-in first-out) policy. E.g., a printer/job queue!
- Two basic operations of queues:
  - dequeue: remove an item/element from front
  - enqueue: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



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Queue ADT

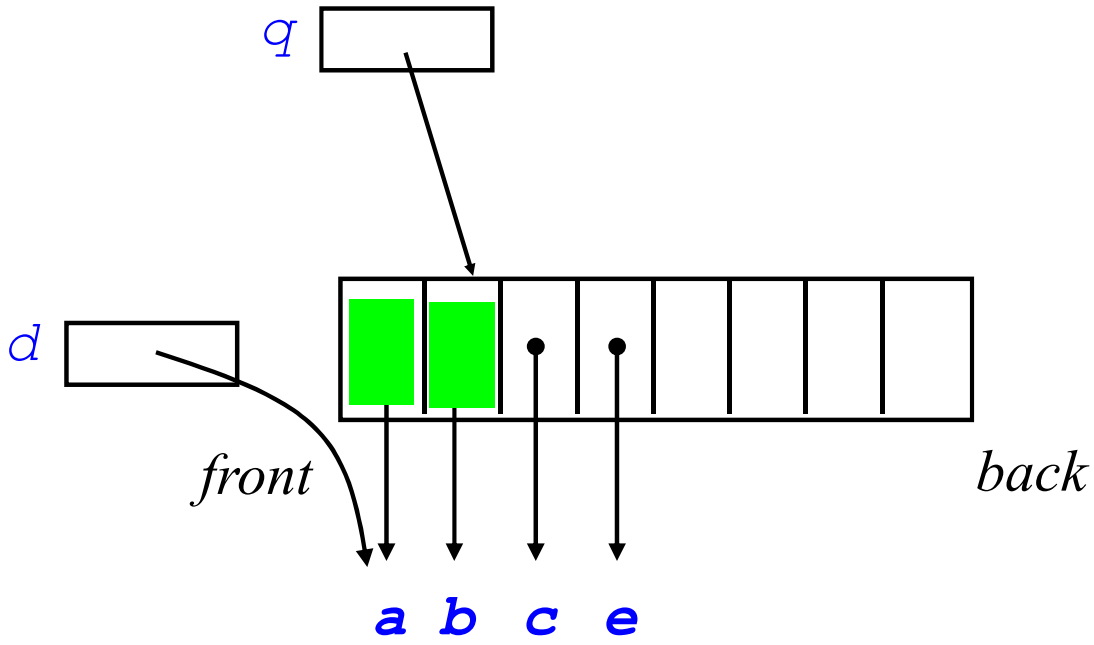
- Queues implement the FIFO (first-in first-out) policy
  - An example is the printer/job queue!



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

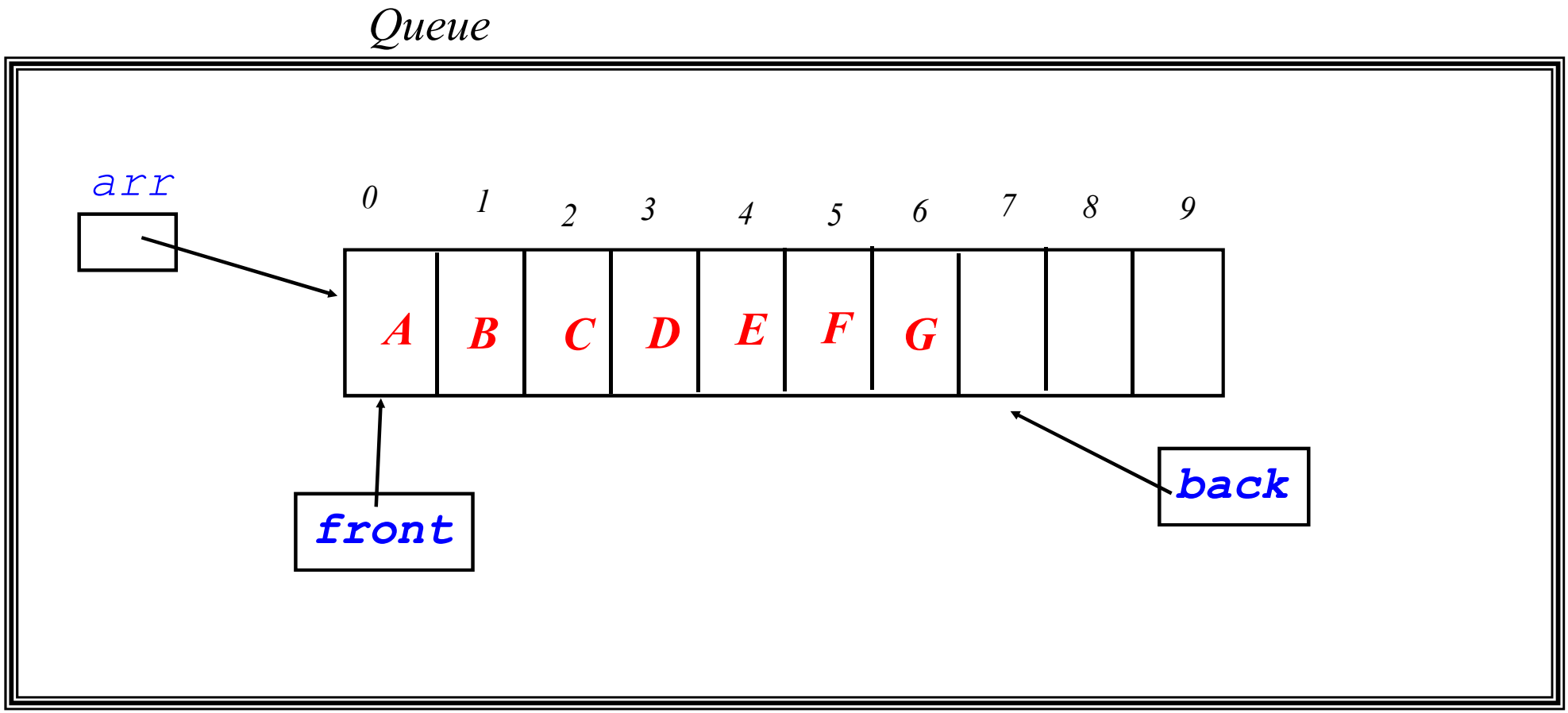
- The main functions in the Queue ADT are (Q is the queue)

```
void enqueue(o, Q)      // insert o 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
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Implementation of Queue (Array)

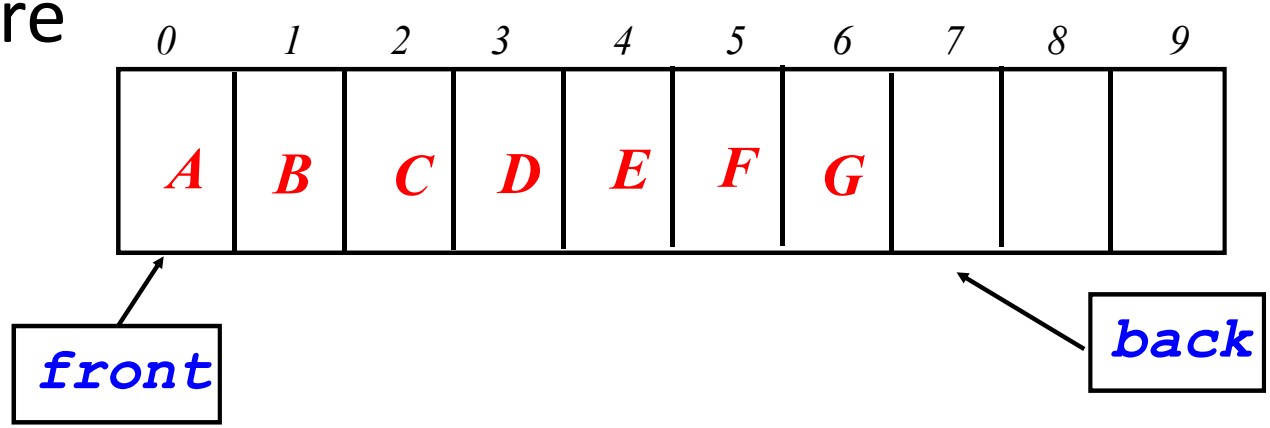
- use Array with **front** and **back** pointers as implementation of queue



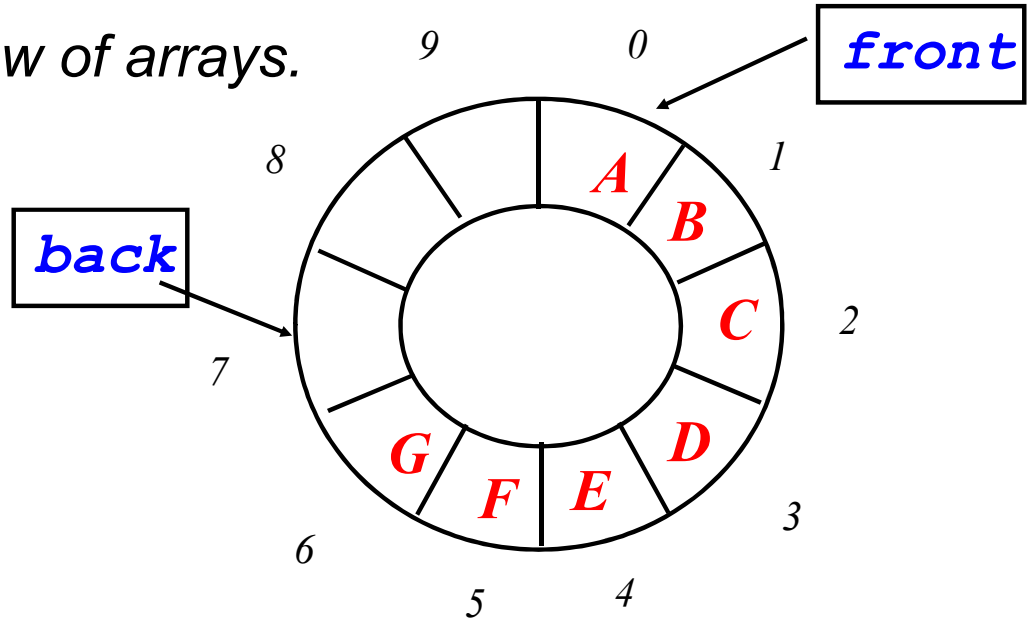
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Circular Array

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



Circular view of arrays.



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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:

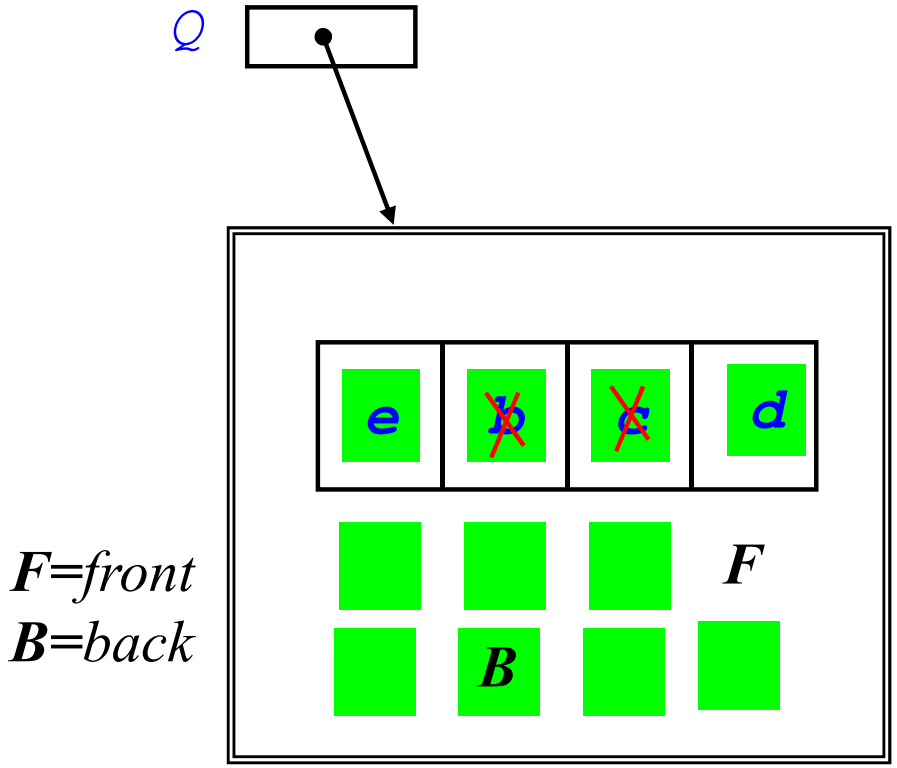
```
int adv(int p)
 { return ((p+1) % maxsize);
 }
```

mod operator

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

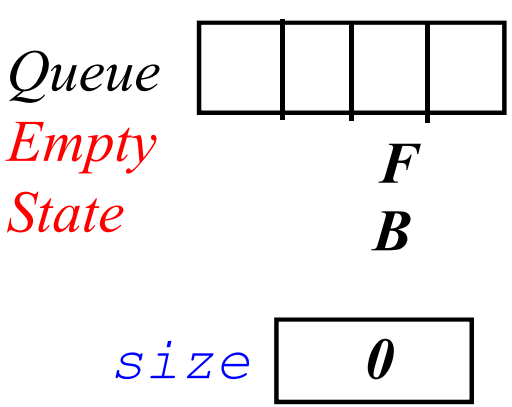
Sample

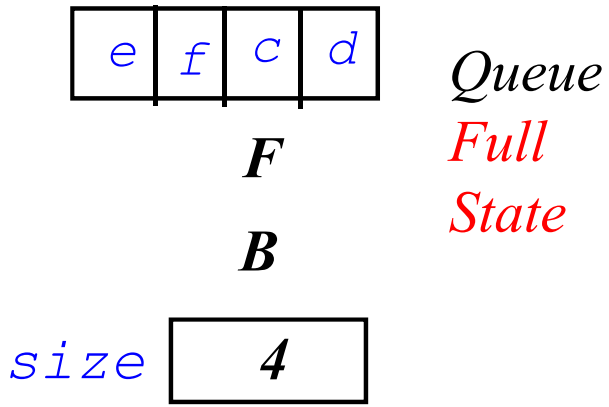
```
Queue *Q;
enqueue(Q, "a");
enqueue(Q, "b");
enqueue(Q, "c");
dequeue(Q);
dequeue(Q);
enqueue(Q, "d");
enqueue(Q, "e");
dequeue(Q);
```



Checking for Full/Empty State

What does (F==B) denote?

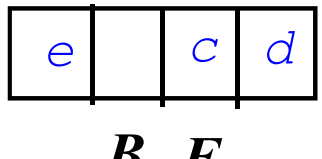




Alternative - Leave a Deliberate Gap!

No need for `size` *field.*

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

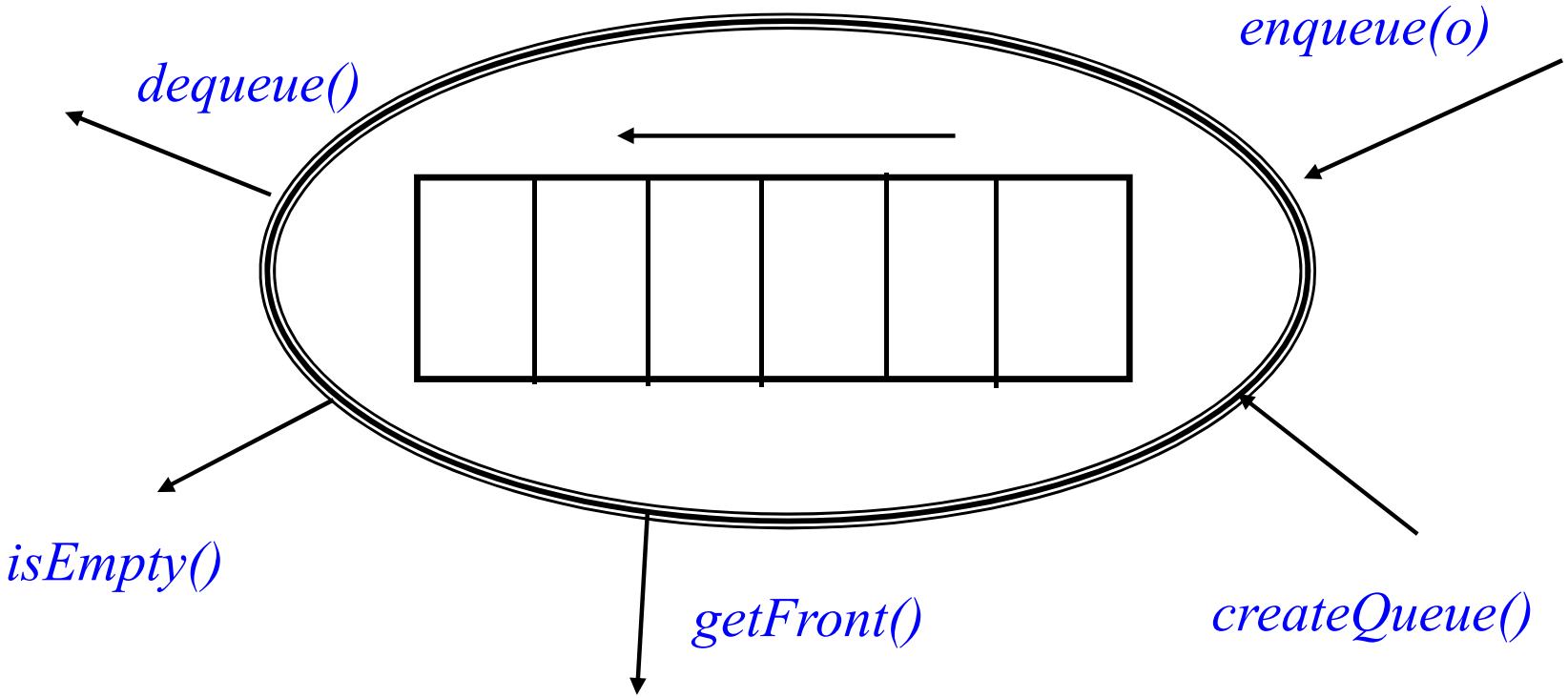


QUEUES: LINKED LIST IMPLEMENTATION

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Recap: Queue ADT

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



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Recap: Queue ADT interface

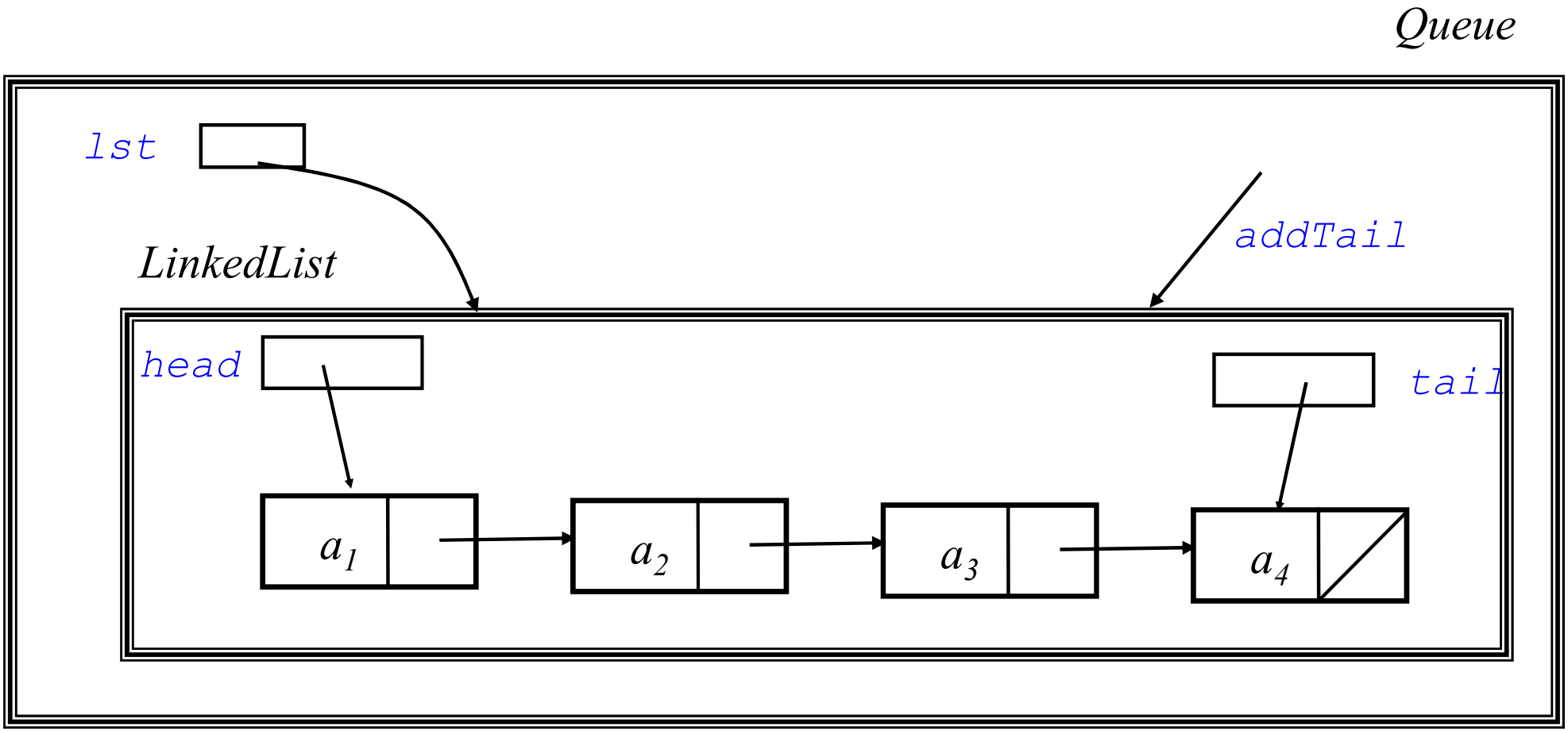
- The main functions in the Queue ADT are (Q is the queue)

```
void enqueue(o, Q)      // insert o 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
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Queue by Linked List

- Can use LinkedList as underlying implementation of Queues



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Queue by Linked Lists in C

```
typedef short BOOLEAN;
#define TRUE  ((short)(1))
#define FALSE ((short)(0))

struct NODE {
  int element;
  struct NODE *next;
};

typedef struct {
   struct NODE
     *front, *rear;
} QUEUE;
```

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

void clear(QUEUE *pQ) {
  struct NODE *n, *pre;
  n = pQ->front;
  while ( n ) {
    pre = n;
    n = n->next;
    free(pre);
  }
  initialize(pQ);
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Queue by Linked Lists: dequeue

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

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Queue by Linked Lists: enqueue

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

```
/* EXAMPLE
[74 21 42 94 13 24 80 45 75 37 ]
74 [21 42 94 13 24 80 45 75 37 ]
21 [42 94 13 24 80 45 75 37 ]
[42 94 13 24 80 45 75 37 102 ]
[42 94 13 24 80 45 75 37 102 106 ]
42 [94 13 24 80 45 75 37 102 106 ]
*/
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

END OF LECTURE 6

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I