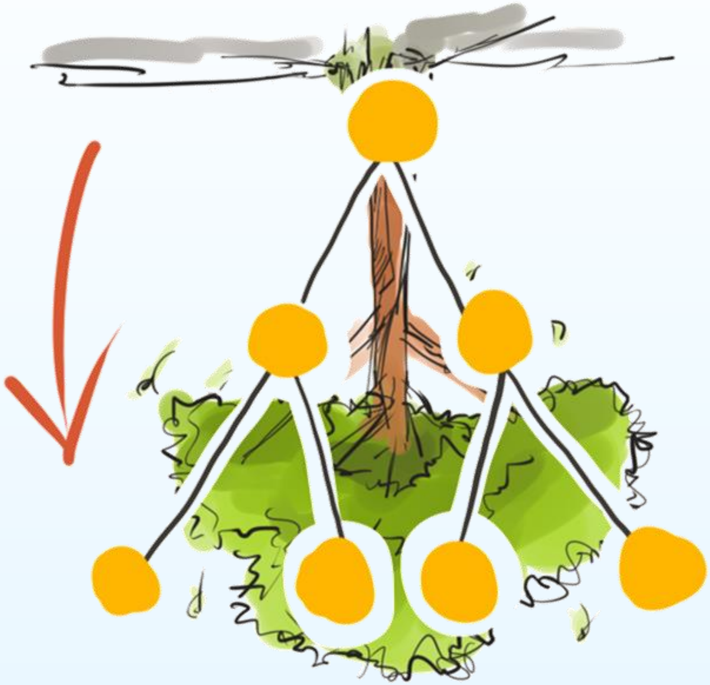


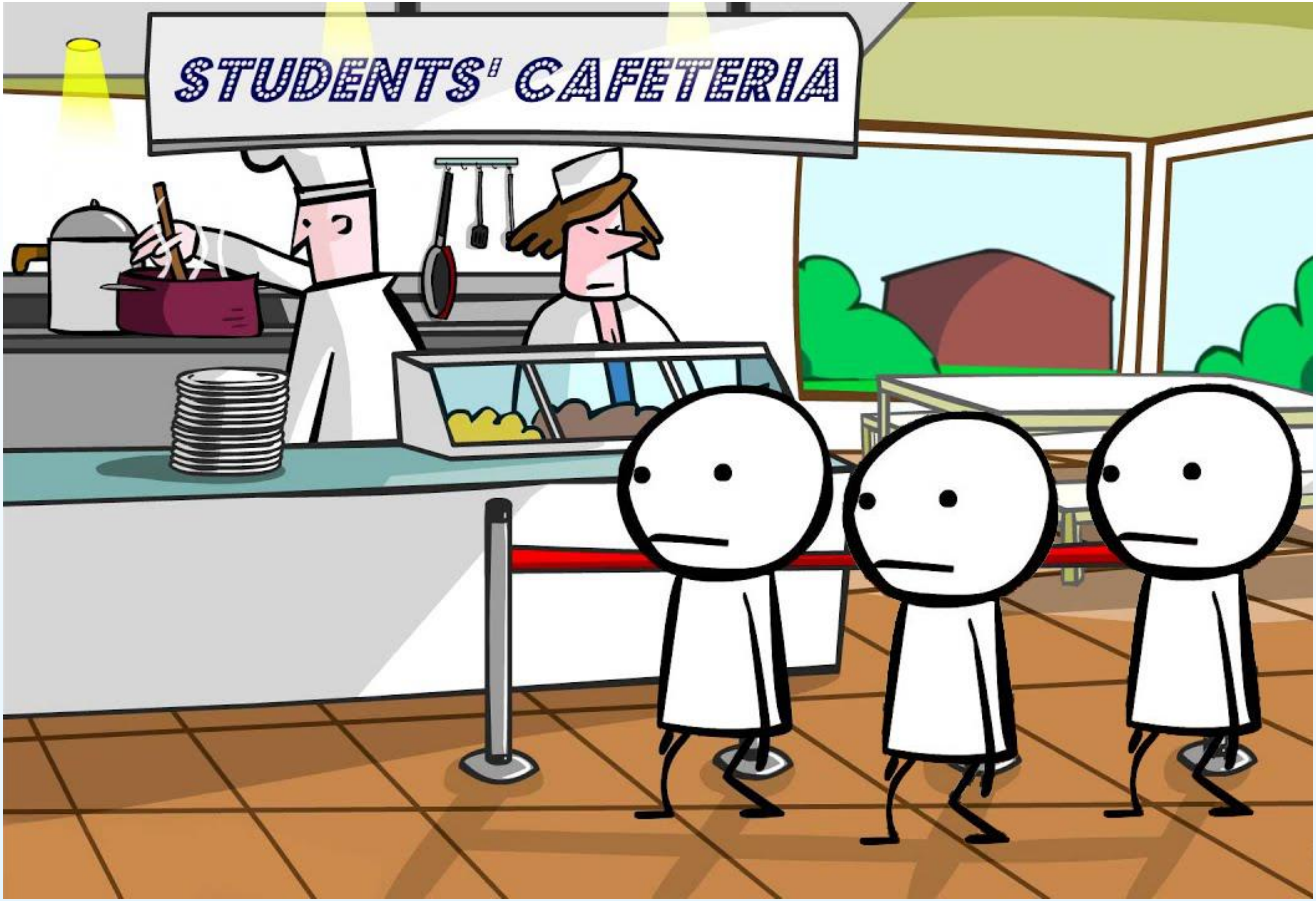
# Lecture 6: Stack ADT & Queue ADT

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.

Department of Computer Engineering Kasetsart University

MIKE LABORATORY

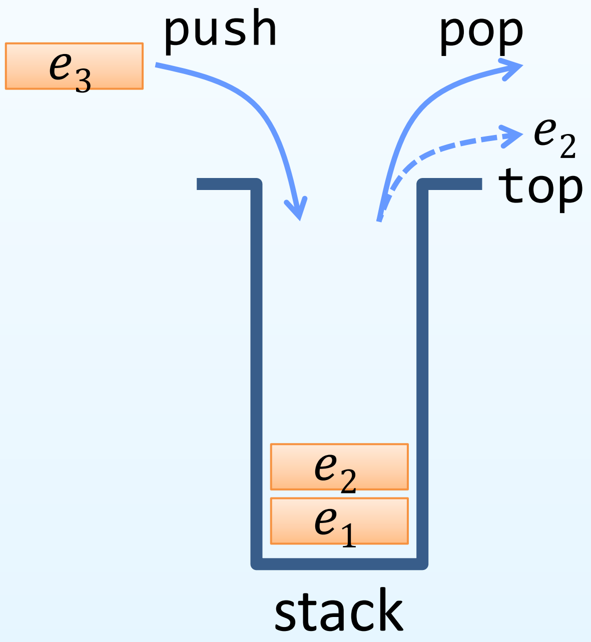

STUDENTS' CAFETERIA

# Stack ADT

# What is a Stack ADT?

- Data:
  - Elements stored in a list linearly, but are allowed insertion and deletion only at one end
  - This mechanism is called LIFO – Last in, First out

- Common operations:
  - push(*stack*, *value*)
  - pop(*stack*)
  - top(*stack*)/peek(*stack*)
  - is_empty(*stack*)
  - is_full(*stack*)
  - …



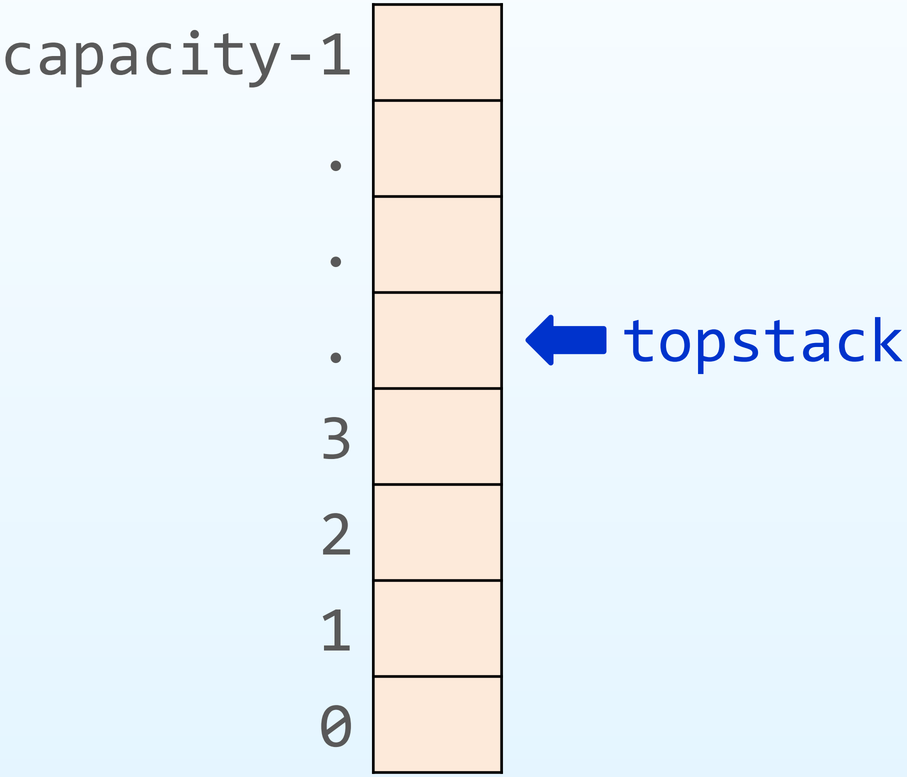
# Stack Implementations

- Two types of implementation
  - Array-based stack
  - Pointer-based stack

# Stack: Array Implementation

Basic idea:

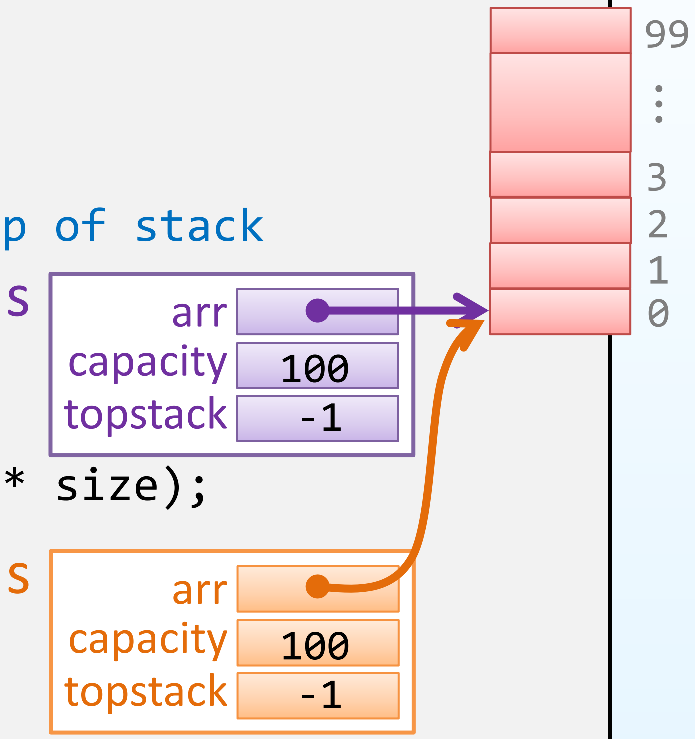
- Allocate a big array (of size `capacity`)
- Keep track of current size (using a variable `topstack`)



# Stack: Array-based Construction

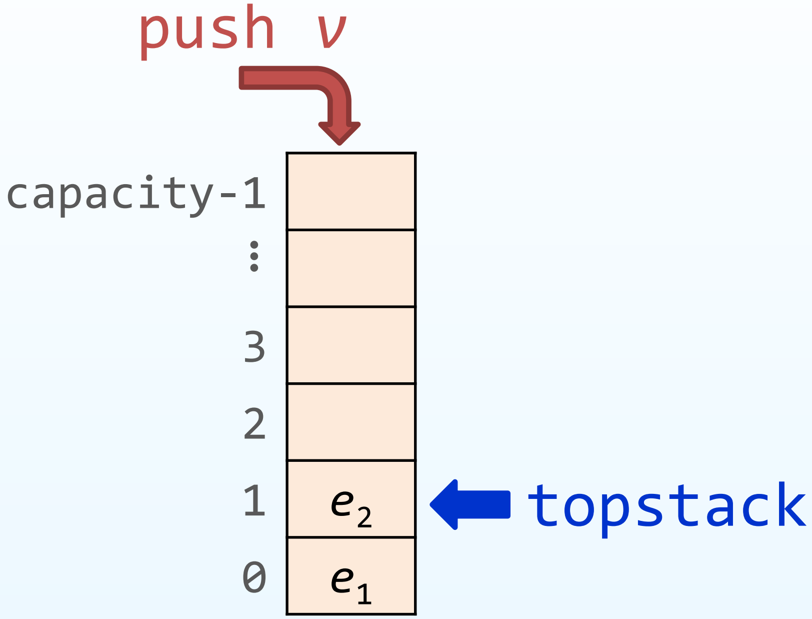
Assume that all data are positive integer

```
#include <stdio.h>
#include <stdlib.h>

typedef struct stack {
  int *arr;      // array-based stack
  int capacity; // size of stack
  int topstack; // position at the top of stack
} stack_t;

stack_t create(int size) {
  stack_t s = {NULL, size, -1};
  s.arr = (int *)malloc(sizeof (int) * size);
  return s;
}
int main(void) {
  stack_t s = create(100);
  return 0;
}
```



# Array-based Stack: `push()` Operation

Push a value *v* at the top of stack *s*



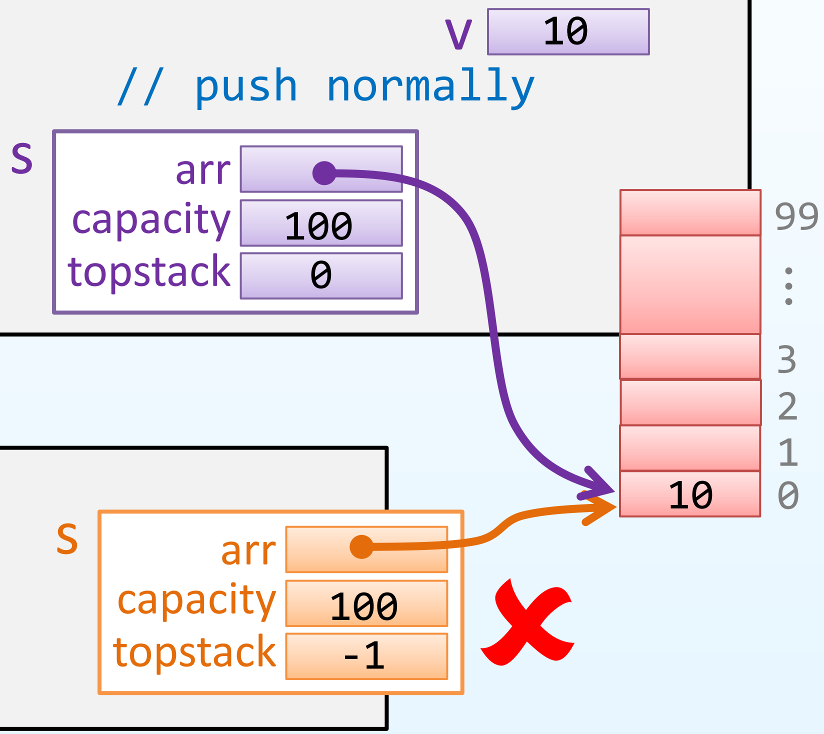
- If `topstack < capacity`, push *v* normally
- Otherwise, the stack is full

# Array-based Stack: push() Operation

Push a value *v* at the top of stack *s*

```
int push(stack_t s, int v) {
  if (s.topstack == s.capacity-1)   // stack is full
    return 0;

  s.topstack++;                     // push normally
  s.arr[s.topstack] = v;

  return 1;
}
```

```
int main(void) {
  stack_t s = create(100);
  push(s, 10);
  return 0;
}
```

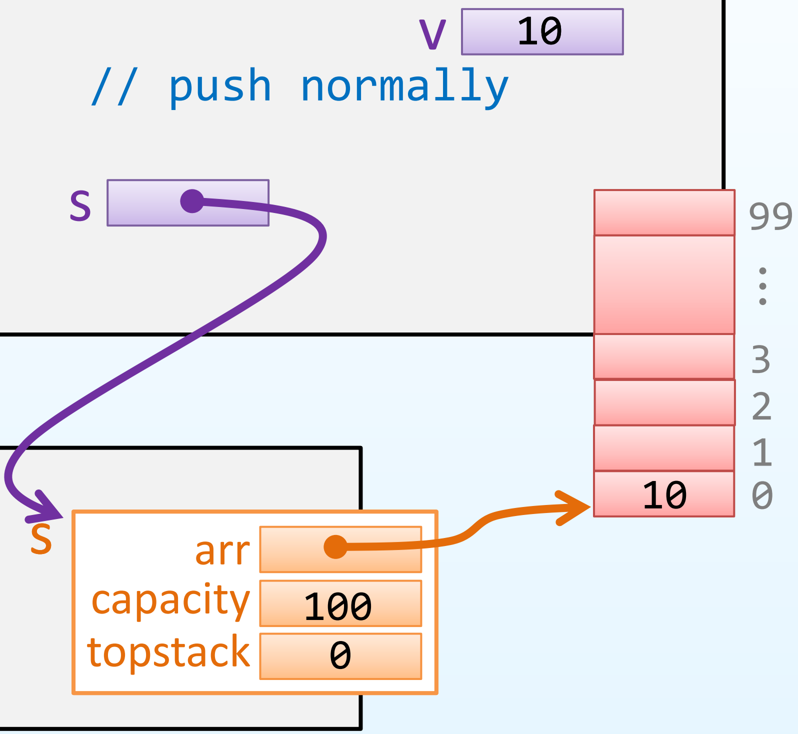


# Array-based Stack: push() Operation

Push a value *v* at the top of stack *s*

```c
int push(stack_t *s, int v) {
  if (s->topstack == s->capacity-1) // stack is full
    return 0;

  s->topstack++;                     // push normally
  s->arr[s->topstack] = v;

  return 1;
}
```

```c
int main(void) {
  stack_t s = create(100);
  push(&s, 10);
  return 0;
}
```



# Exercise 1: Other Operations

Implement the following functions for an array-based stack

- `pop()` – remove the top element of stack *s*
  - Return *v* if the stack is not empty, otherwise -1
- `top()` – peek the top element of stack *s*
  - Return *v* if the stack is not empty, otherwise -1
- `is_empty()` – check whether stack *s* is empty
  - return 1 if the stack is empty, otherwise 0
- `is_full()` – check whether stack *s* is full
  - return 1 if the stack is full, otherwise 0

# Array-based Stack: Running Time

| Operation | Running Time |
|---|---|
| `create()` | $O(1)$ |
| `push()` | $O(1)$ |
| `pop()` | $O(1)$ |
| `top()` | $O(1)$ |
| `is_empty()` | $O(1)$ |
| `make_empty()` | $O(1)$ |
| `is_full()` | $O(1)$ |
| `destroy()` | $O(1)$ |

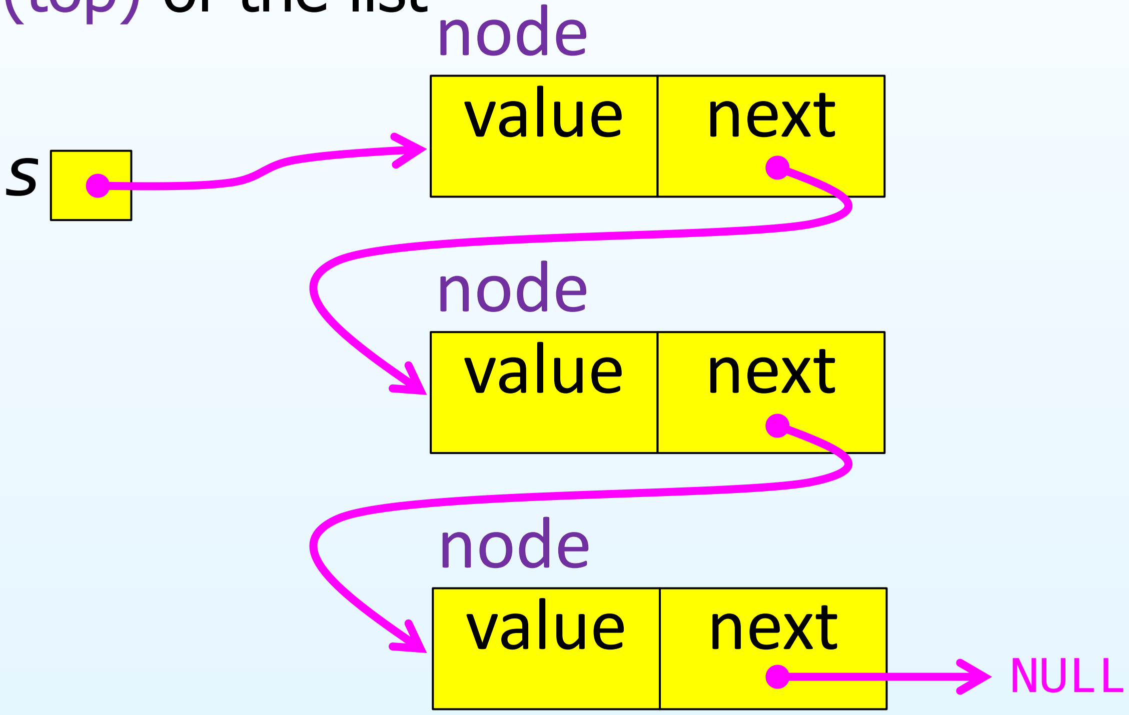
# Limitation of Array-based Stack

- If the stack is full, reallocate a huge new array and move everything over

# Stack: Pointer Implementation

Basic idea:

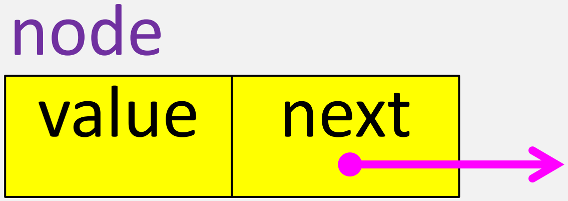
- Allocate nodes for elements, and link them as a list
- Form a stack by manipulating push and pop operations at the head (top) of the list



# List-based Stack: Construction

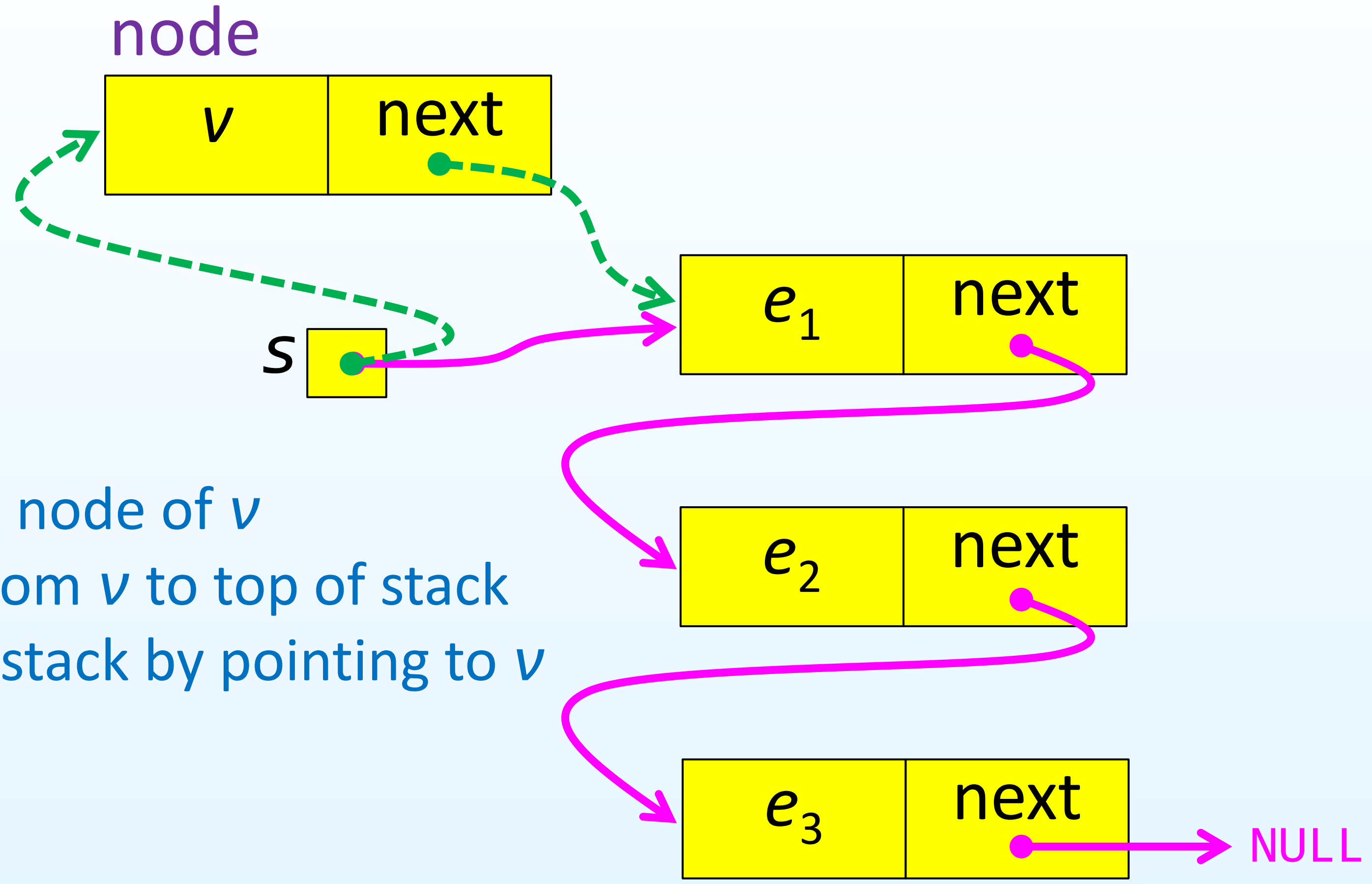
Assume that all data are positive integer

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
  int value;
  struct node *next;
} node_t;

typedef node_t stack_t;

int main(void) {
  stack_t *s = NULL;
  return 0;
}
```



# List-based Stack: `push()` Operation

Push a value *v* at the top of stack *s*

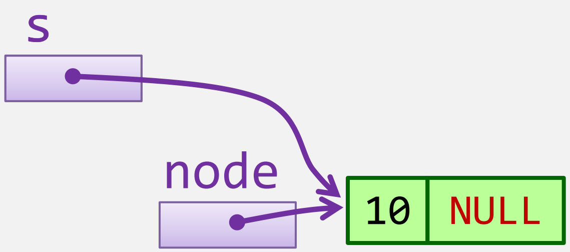


1. Allocate a new node of *v*
2. Create a link from *v* to top of stack
3. Change top of stack by pointing to *v*

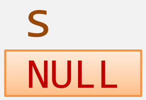
# List-based Stack: push() Operation

Push a value *v* at the top of stack *s*

```
void push(stack_t *s, int v) {
  node_t *node = (node_t *)malloc(sizeof (node_t));
  node->value = v;
  node->next = NULL;

  node->next = s;
  s = node;
}
```



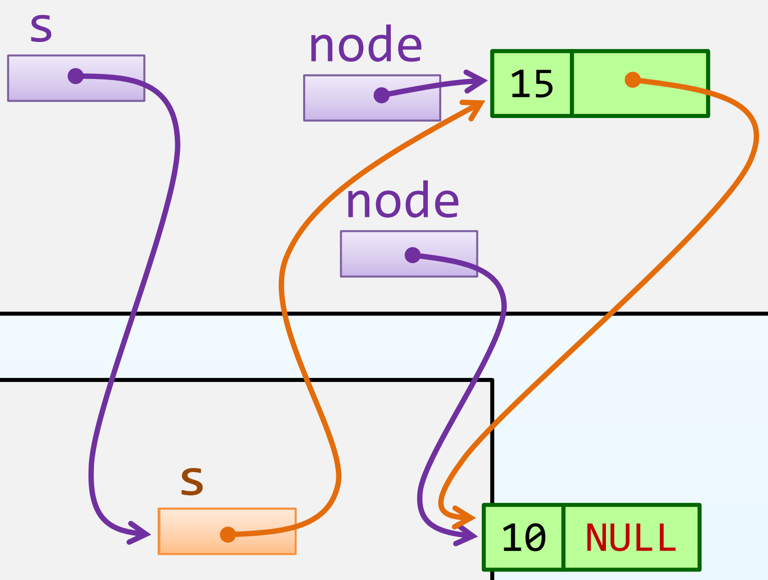
```
int main(void) {
  stack_t *s = NULL;
  push(s, 10);
  return 0;
}
```



# List-based Stack: push() Operation

Push a value *v* at the top of stack *s*

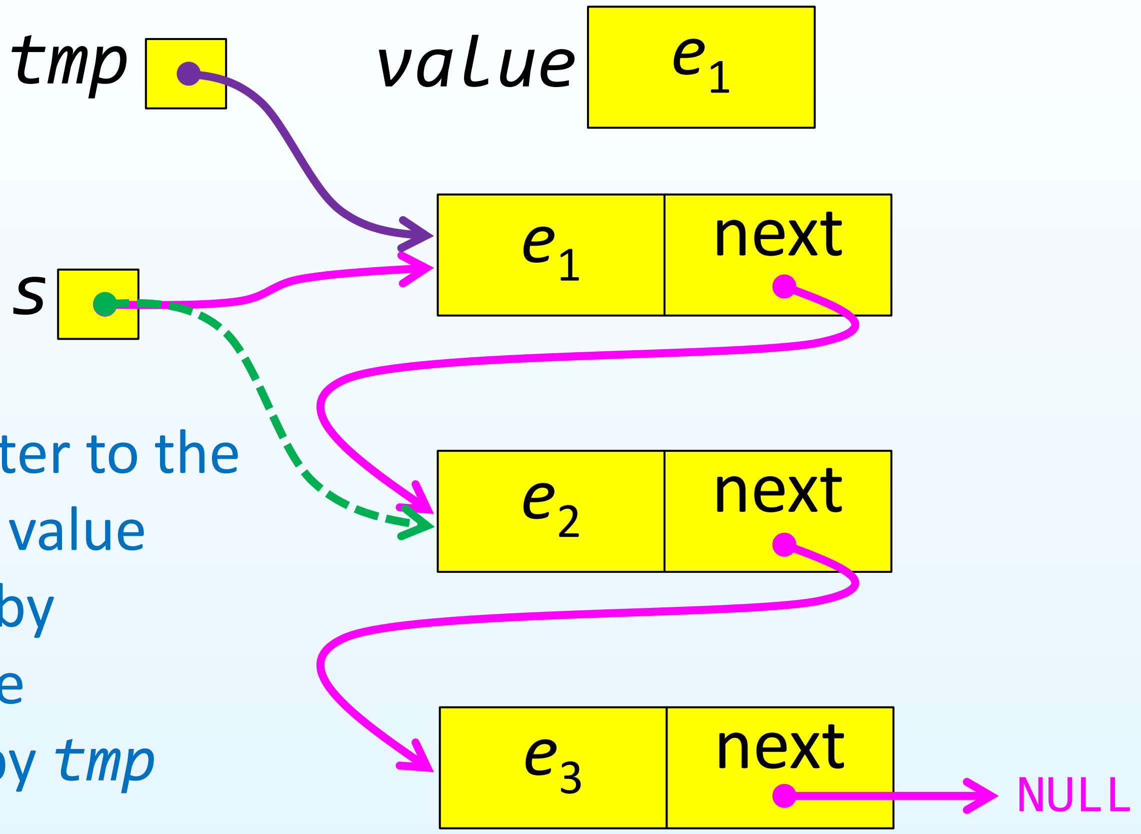
```
void push(stack_t **s, int v) {
  node_t *node = (node_t *)malloc(sizeof (node_t));
  node->value = v;
  node->next = NULL;

  node->next = *s;
  *s = node;
}
```



```
int main(void) {
  stack_t *s = NULL;
  push(&s, 10);
  push(&s, 15);
  return 0;
}
```

# List-based Stack: pop( ) Operation

Remove the top of stack *s* and then return that value



1. Assign a temporary pointer to the top of stack, and get the value
2. Change the top of stack by pointing to the next node
3. Free the node pointing by *tmp*

# Exercise 2: Other Operations

Implement the following functions for a list-based stack

- `pop()` – remove the top element of stack *s* ▶
  - Return *v* if the stack is not empty, otherwise -1
- `top()` – peek the top element of stack *s* ▶
  - Return *v* if the stack is not empty, otherwise -1
- `is_empty()` – check whether stack *s* is empty ▶
  - return 1 if the stack is empty, otherwise 0

# List-based Stack: Running Time

| Operation | Running Time |
|---|---|
| `push()` | $O(1)$ |
| `pop()` | $O(1)$ |
| `top()` | $O(1)$ |
| `is_empty()` | $O(1)$ |
| `make_empty()` | $O(n)$ |

# Limitation of List-based Stack

- Potentially a lot of calls to `malloc()` and `free()` if the stack is actively used
  - In practice, memory allocation and release require expensive trips through the operating system

How can you solve this problem?

# Application: Balancing Symbols

Are the followings correct?

- Arithmetic expression

  (1+5∗(17−2)/(6∗3))

- Python syntax

```
print('''He said, "I 'do not' care."''')
```

- HTML code

```
<html>
<head><title>hello</title></head>
<body>test</body>
</html>
```

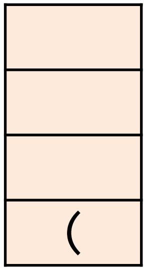
# Application: Balancing Symbols

```
(1+5*(17-2)/(6*3))
```

```
1. Create an empty stack
2. If encounter '(', push onto stack
3. If encounter ')',
    3.1 If stack is empty, report error
    3.2 Else, pop the stack
    3.3 If the popped value is not '(', report error
4. If EOF and stack is not empty, report error
```

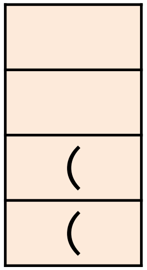
# Application: Balancing Symbols

1



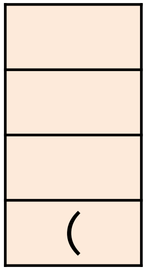
```
(1+5*(17-2)/(6*3))
push('(');
```

2



```
(1+5*(17-2)/(6*3))
push('(');
```

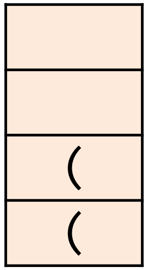
3



```
(1+5*(17-2)/(6*3))
sym = pop();
```

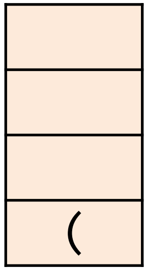
➔ Match

4



```
(1+5*(17-2)/(6*3))
push('(');
```

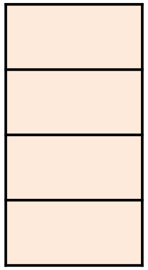
5



```
(1+5*(17-2)/(6*3))
sym = pop();
```

➔ Match

6



```
(1+5*(17-2)/(6*3))
sym = pop();
```

➔ Match

# Application: Arithmetic Expression

What is the result of following expressions?

- Infix notation

```
10 + 6 * 5 / 2 - 4
```

- Postfix notation

```
10 6 5 * 2 / + 4 -
```

- Prefix notation

```
+ 10 - * 6 / 5 2 4
```

# Application: Infix to Postfix Conversion

```
10 + 6 * 5 / 2 - 4
```

Algorithm

```
s ← create_stack()
while (data ← input() and data != EOF)
    if (data is an operand)
        print(data)
    if (data is an operator)
        while (!is_empty(s))
            op ← top(s)
            if (op has higher precedence than or equal to data)
                op ← pop(s)
                print(op)
            else
                break
        push(s, data)

while (!is_empty(s))
    op ← pop(s)
    print(op)
```

# Application: Infix to Postfix Conversion

10 + 6 * 5 / 2 – 4

| Input | Stack | Output |
|---|---|---|
| **10** + 6 * 5 / 2 – 4 | | 10 |
| 10 **+** 6 * 5 / 2 – 4 | + | 10 |
| 10 + **6** * 5 / 2 – 4 | + | 10 6 |
| 10 + 6 ***** 5 / 2 – 4 | + * | 10 6 |
| 10 + 6 * **5** / 2 – 4 | + * | 10 6 5 |
| 10 + 6 * 5 **/** 2 – 4 | + | 10 6 5 * |
| | + / | 10 6 5 * |
| 10 + 6 * 5 / **2** – 4 | + / | 10 6 5 * 2 |
| 10 + 6 * 5 / 2 **–** 4 | + | 10 6 5 * 2 / |
| | | 10 6 5 * 2 / + |
| | – | 10 6 5 * 2 / + |
| 10 + 6 * 5 / 2 - **4** | – | 10 6 5 * 2 / + 4 |
| 10 + 6 * 5 / 2 - 4 | | 10 6 5 * 2 / + 4 – |

Department of Computer Engineering Kasetsart University

MIKE LABORATORY

# Other Applications of Stack

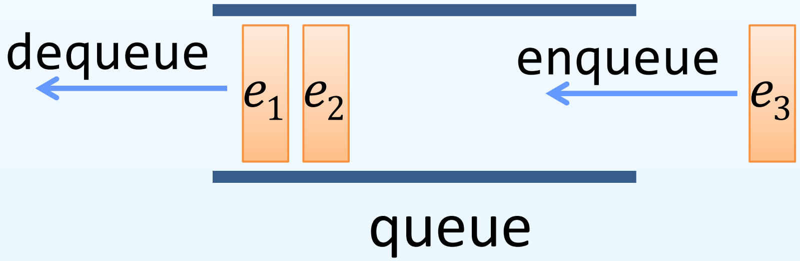
- Backtracking
  - A recursive algorithm which is used for solving the optimization problem
- Function calls
  - Whenever you invoke a function, the address of the calling function gets stored in the stack. This helps in going back when the called function is terminated
- Memory management
  - The stack segment of memory
- Depth-first search

# Queue ADT

# What is a Queue ADT?

- Data:
  - Elements stored in a list linearly, but are allowed insertion at one end and deletion at the other end
  - This mechanism is called FIFO – First in, First out

- Common operations:
  - `enqueue(`*`queue`*`, `*`value`*`)`
  - `dequeue(`*`queue`*`)`
  - `is_empty(`*`queue`*`)`
  - `is_full(`*`queue`*`)`
  - …



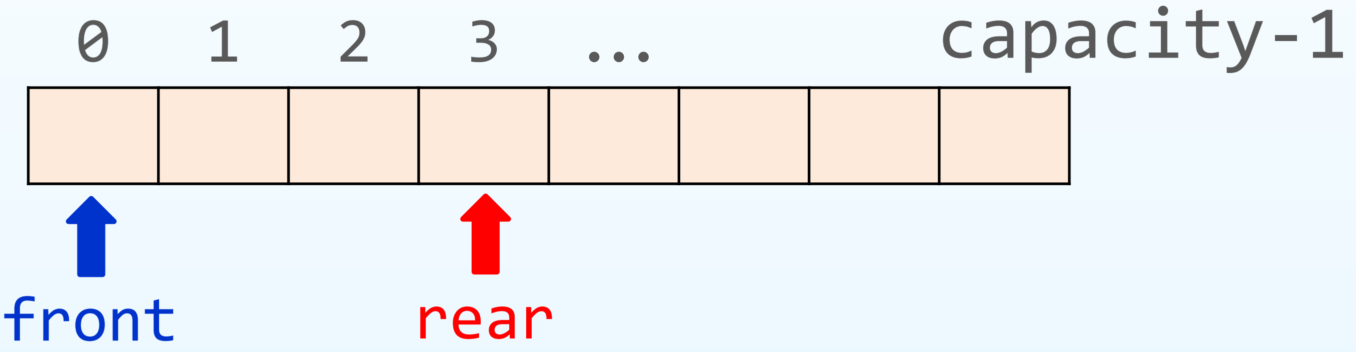
# Queue Implementations

- Two types of implementation
  - Array-based queue
  - Pointer-based queue

# Queue: Array Implementation

Basic idea:

- Allocate a big array (of size `capacity`)
- Keep track two ends (using variables `front` and `rear`)

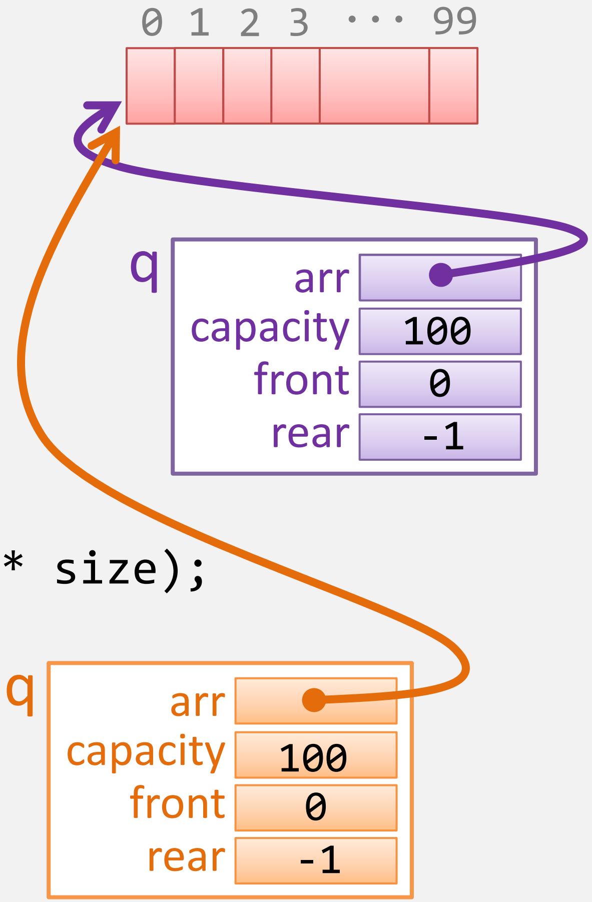


**How should we initialize both variables?**

# Queue: Array-based Construction

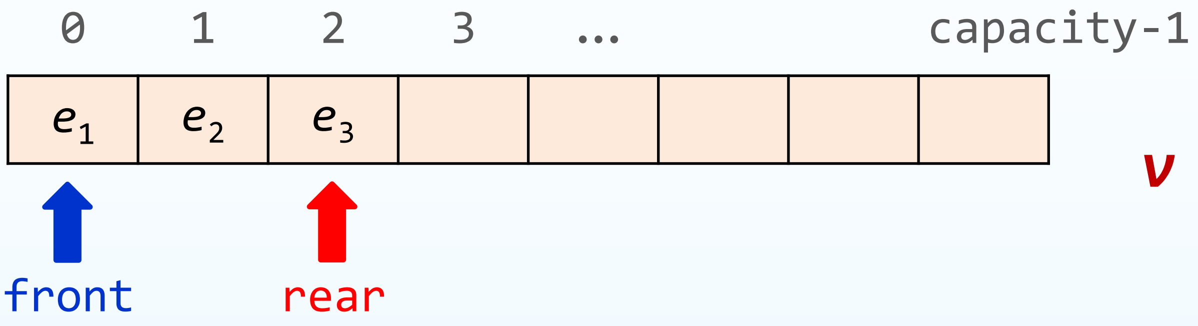
Assume that all data are positive integer

```
#include <stdio.h>
#include <stdlib.h>

typedef struct queue {
  int *arr;     // array-based queue
  int capacity; // size of queue
  int front;    // position of front
  int rear;     // position of rear
} queue_t;

queue_t create(int size) {
  queue_t q = {NULL, size, 0, -1};
  q.arr = (int *)malloc(sizeof (int) * size);
  return q;
}
int main(void) {
  queue_t q = create(100);
  return 0;
}
```



# Array-based Queue: enqueue() Operation

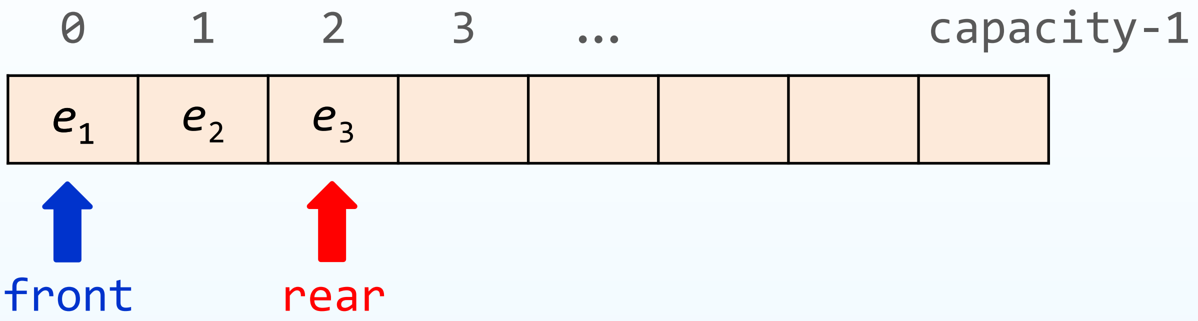
Insert a value *v* into queue *q*



```
int enqueue(queue_t *q, int v) {
  if (q->rear == q->capacity-1)   // queue is full
    return 0;

  q->rear++;                      // enqueue normally
  q->arr[q->rear] = v;
  return 1;
}
```

# Array-based Queue: dequeue() Operation

Remove a value from queue *q* and return it



```
int dequeue(queue_t *q) {
  int v;

  if (q->front > q->rear)           // queue is empty
    return 0;

  v = q->arr[q->front];             // dequeue normally
  q->front++;
  return v;
}
```

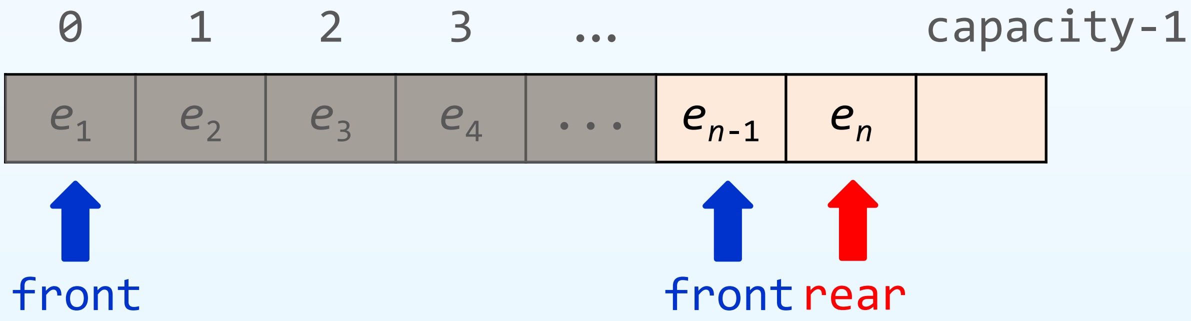
# Array-based Queue: Running Time

| Operation | Running Time |
|---|---|
| `create()` | $O(1)$ |
| `enqueue()` | $O(1)$ |
| `dequeue()` | $O(1)$ |
| `is_empty()` | $O(1)$ |
| `make_empty()` | $O(1)$ |
| `is_full()` | $O(1)$ |
| `destroy()` | $O(1)$ |

# Limitation of Array-based queue

- If the queue is full, reallocate a huge new array and move everything over

- A simple implementation of `dequeue()` can lead to a lot of unused spaces

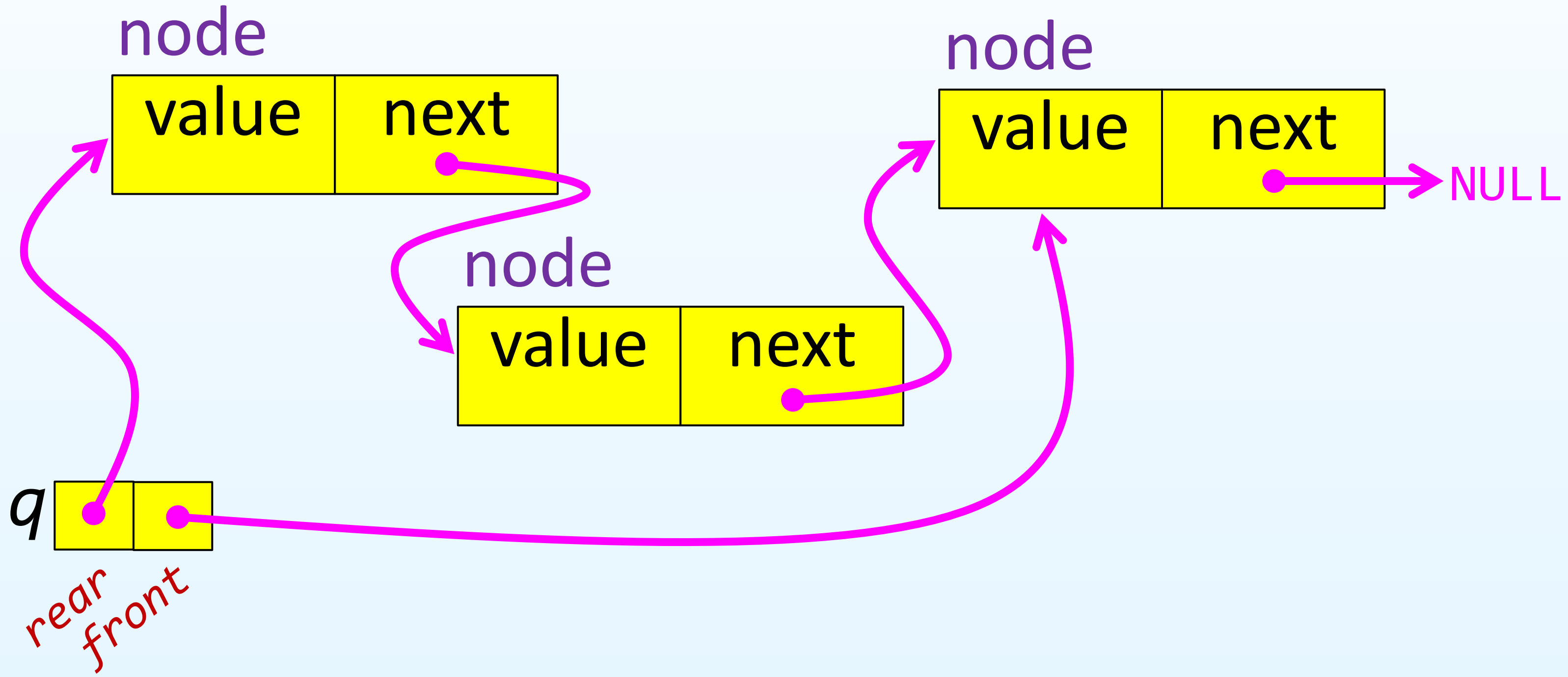


How can you solve this problem?

# Queue: Pointer Implementation

Basic idea:

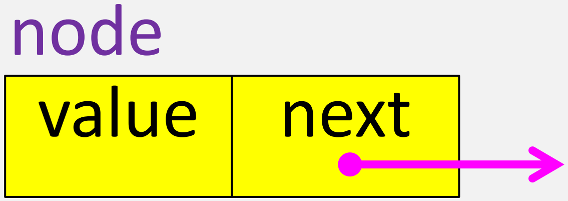
- Allocate nodes for elements, and link them as a list
- Form a queue with front and rear pointers

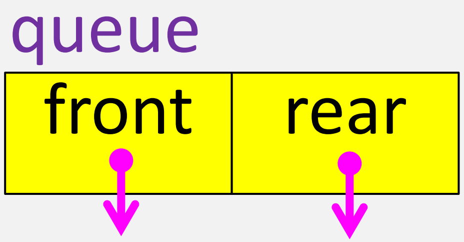


# List-based Stack: Construction

Assume that all data are positive integer

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
  int value;
  struct node *next;
} node_t;

typedef struct queue {
  node_t *front;
  node_t *rear;
} queue_t;

int main(void) {
  queue_t q = {NULL, NULL};
  return 0;
}
```

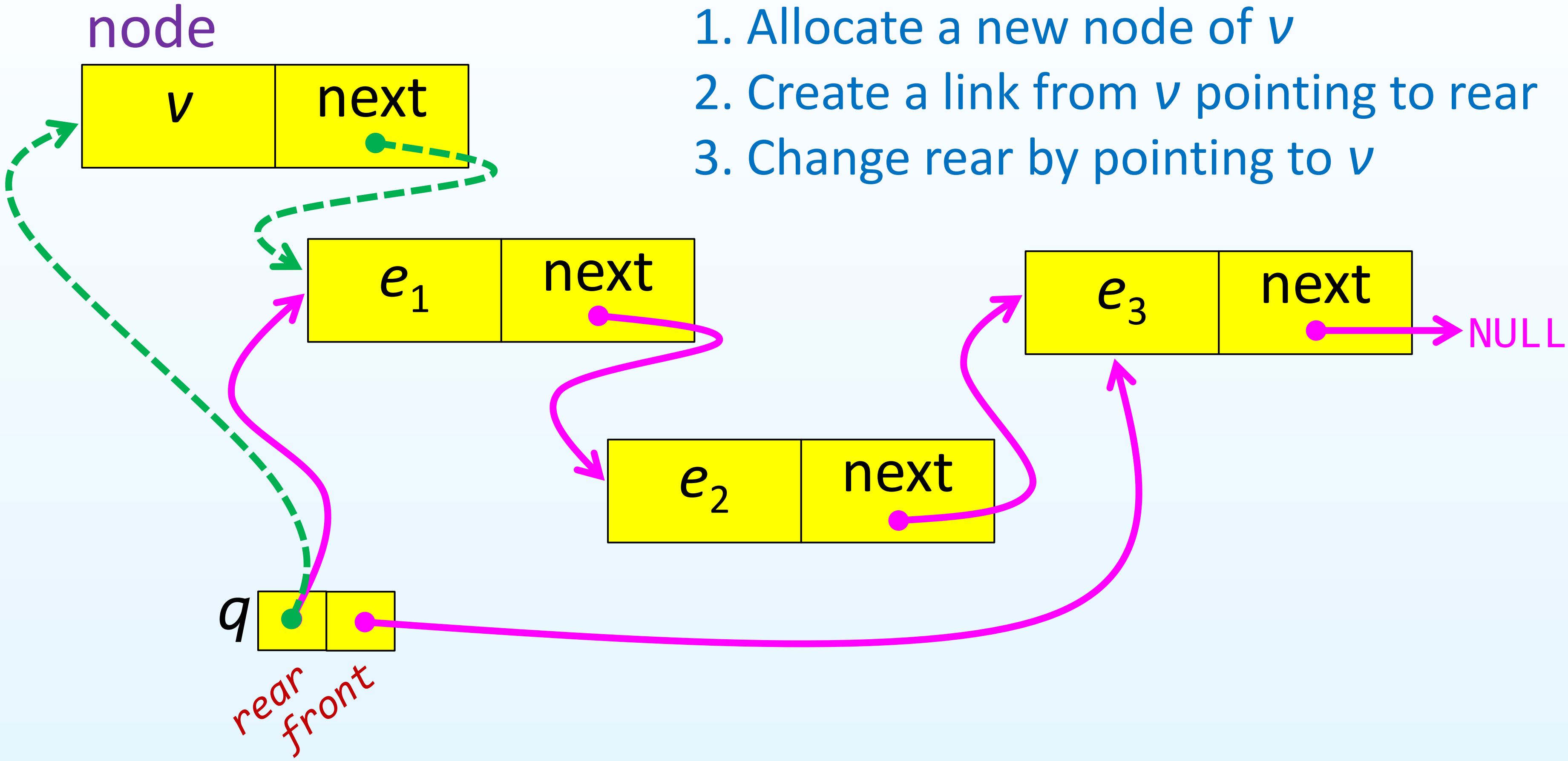




# List-based Queue: enqueue( ) Operation

Insert a value $v$ into queue $q$

1. Allocate a new node of $v$
2. Create a link from $v$ pointing to rear
3. Change rear by pointing to $v$



# List-based Queue: `enqueue()` Operation

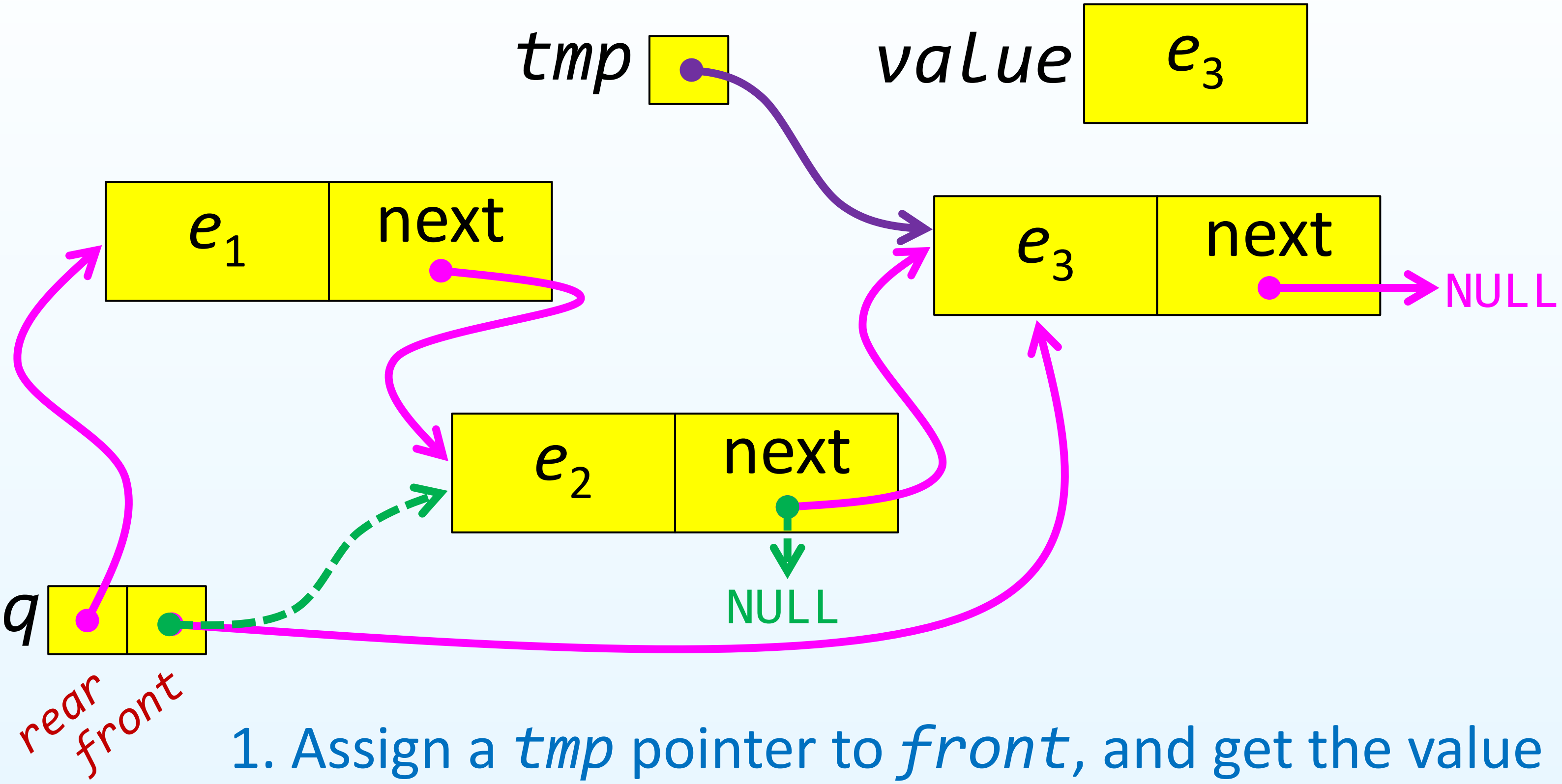
Insert a value *v* into queue *q*

```c
void enqueue(queue_t *q, int v) {
  node_t *node = (node_t *)malloc(sizeof (node_t));
  node->value = v;
  node->next = NULL;

  node->next = q->rear;
  q->rear = node;

  if (q->front == NULL)
    q->front = node;
}
```

1. Allocate a new node of *v*
2. Create a link from *v* pointing to rear
3. Change rear by pointing to *v*

# List-based Queue: dequeue() Operation

Remove a value from queue *q* and return it



1. Assign a *tmp* pointer to *front*, and get the value
2. Change *front* pointing to the 2nd **last node** of list
3. Change next of 2nd **last node** pointing to NULL
4. Free the node pointing by *tmp*

# List-based Queue: dequeue() Operation

Remove a value from queue *q* and return it

```c
int dequeue(queue_t *q) {
  node_t *tmp = NULL;
  int    value = 0;

  if (q->front == NULL)          // queue is empty
    return -1;

  tmp = q->front;
  value = q->front->value;

  if (q->front == q->rear) {     // queue has only one node
    q->front = NULL;
    q->rear = NULL;
  } else {
    q->front = q->rear;          // find 2nd last node
    while (q->front->next != tmp)
      q->front = q->front->next;
    q->front->next = NULL;    // set 2nd last node pointing to NULL
  }
  free(tmp);
  return value;
}
```

# List-based Queue: Running Time

| Operation | Running Time |
|---|---|
| `enqueue()` | $O(1)$ |
| `dequeue()` | $O(n)$ |
| `is_empty()` | $O(1)$ |
| `make_empty()` | $O(n)$ |

# Limitation of List-based Queue

- Potentially a lot of calls to `malloc()` and `free()` if the queue is actively used
  - In practice, memory allocation and release require expensive trips through the operating system

**How can you solve this problem?**

# Applications of Queue

- Printer queue
- Web server queue
- Call Center phone system
- Breadth-first search

# Any Question?