Data Structures

Lecture 6: Queue

Dongbo Min

Department of Computer Science and Engineering

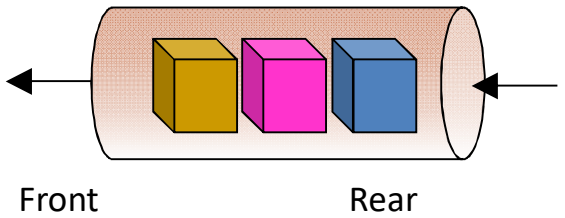
Ewha Womans University, Korea

E-mail: dbmin@ewha.ac.kr

Queue

- Queue: Data structure where an incoming data comes first
- First-In First-Out (FIFO)
  Ex) Queue at ticket office

- Insertion and deletion
  - Insertion takes place at the rear of the queue
  - Deletion takes place at the front of the queue.



ADT of Queue

- Object: an ordered group consisting of n elements

- Operation:

  Create() :: = Creates a queue.

  Init (q) :: = Initialize the queue.

  is_empty (q) :: = Checks if the queue is empty.

  is_full (q) :: = Checks whether the queue is full.

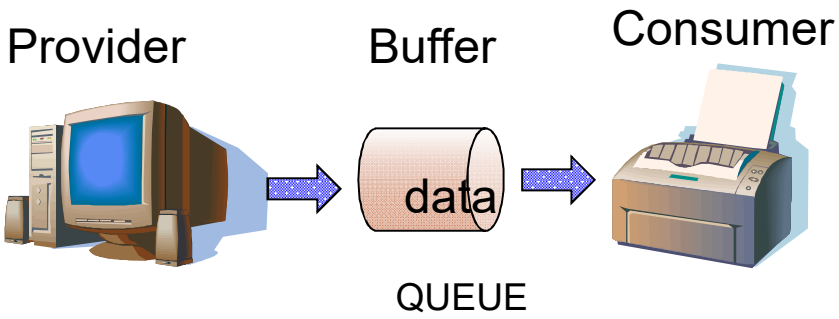
  enqueue (q, e) :: = Add an element at the rear of the queue.

  dequeue (q) :: = return the element at the front of the queue and delete it.

  peek (q) :: = Returns the previous element without deleting it from the queue.

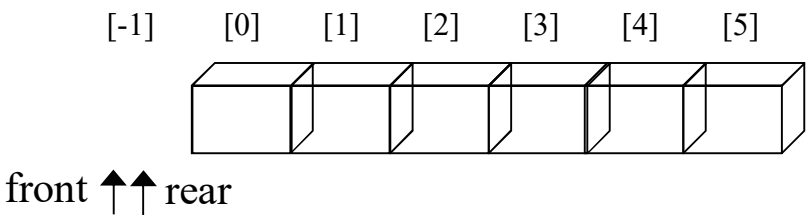
Queue Applications

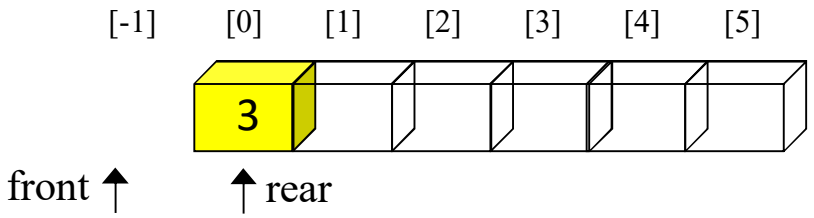
- Application
  - Simulation queues (airplanes at the airport, queues at the bank)
  - Modeling data packets in communication
  - Buffering between printer and computer
  - Like the stack, the programmer's tool
  - Used in many algorithms

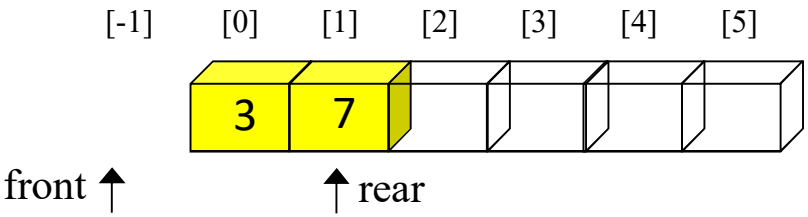


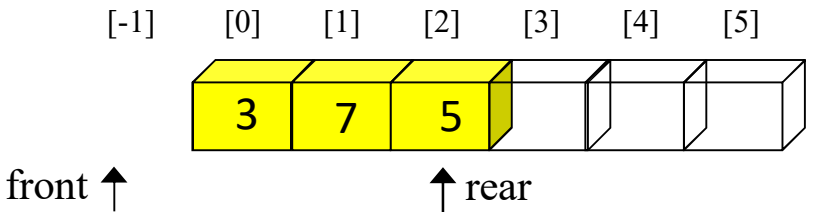
Queue using Array

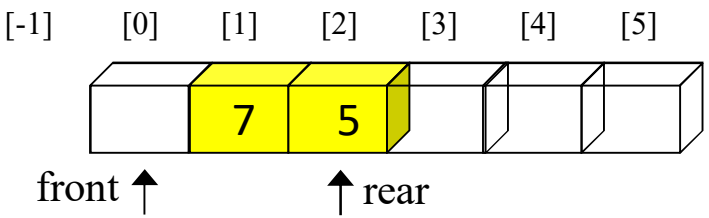
- Linear queue
  - Implements a queue using arrays linearly

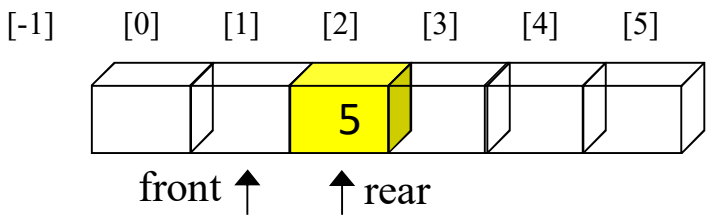






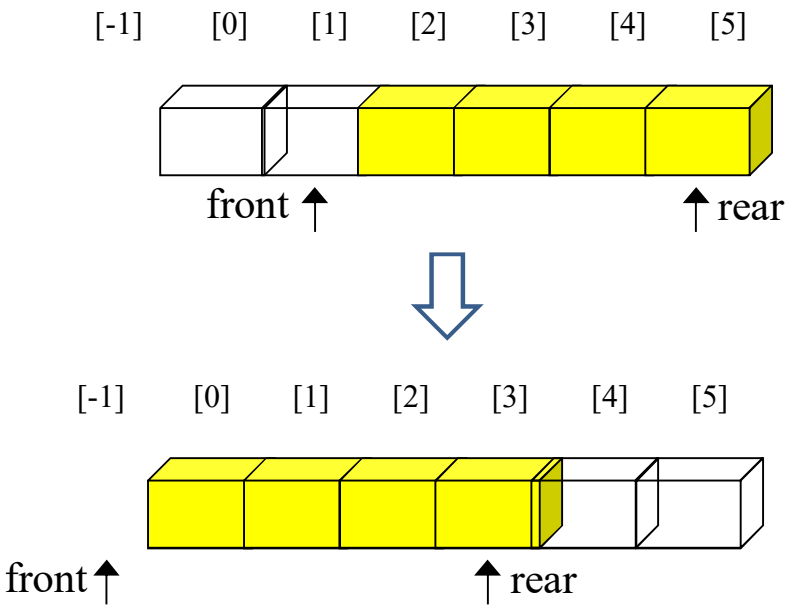






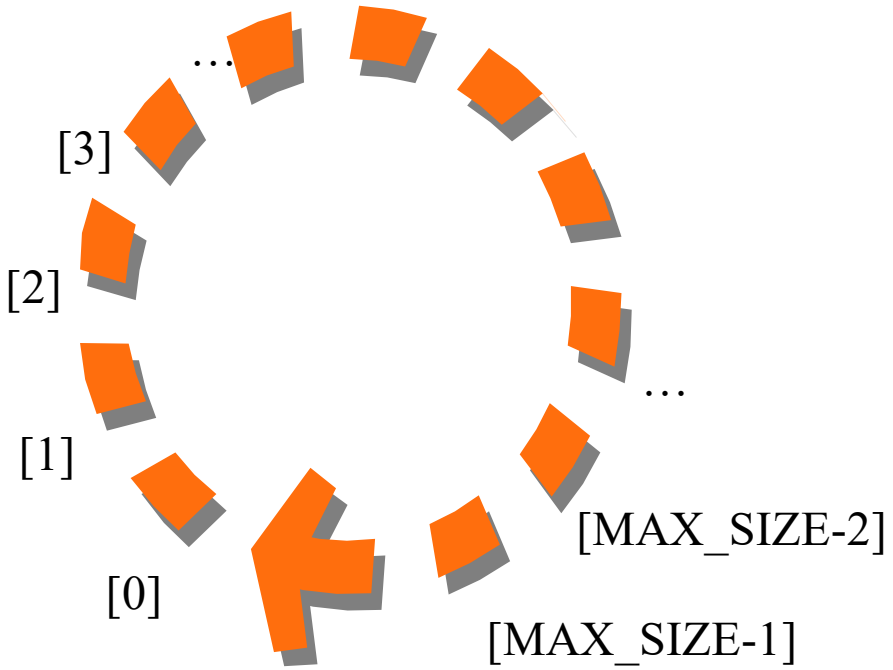
Queue using Array

- Linear queue
  - Moving elements for insertion when reaching at the end of array
  - Not used due to computational overhead



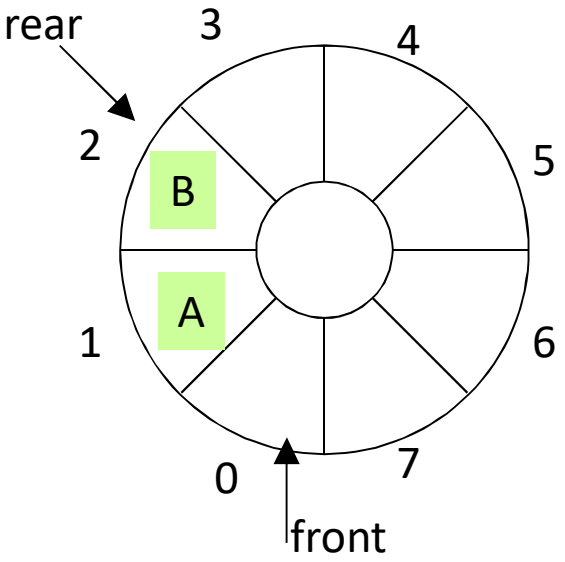
Queue using Array

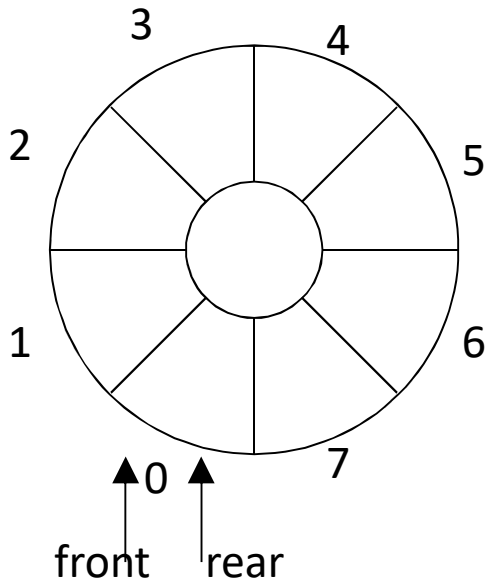
- Circular queue
  - Implements a queue using arrays in a circular form



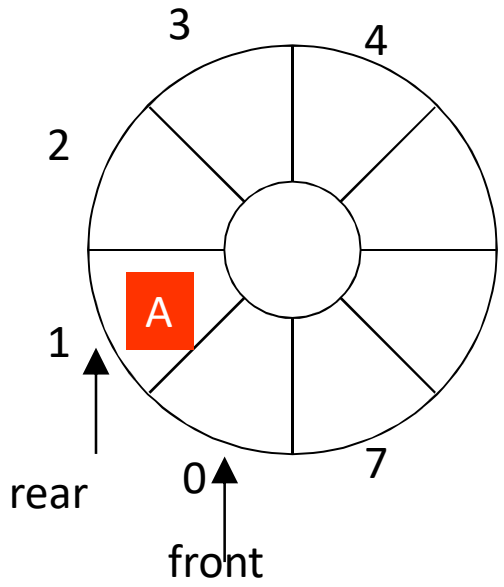
Circular Queue

- Two variables to manage the front and back of the queue
  - `front`: index before the first element
  - `rear`: index of the last element

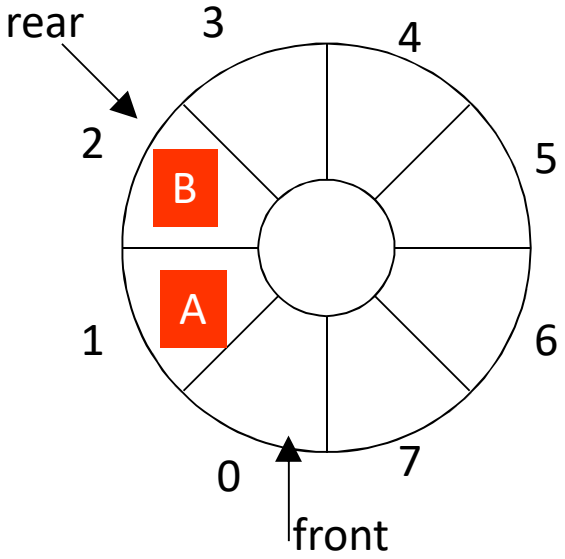




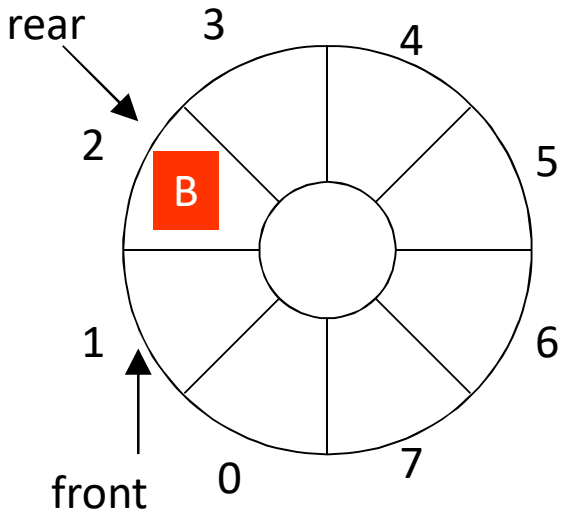
(a) Initialization



(b) Insert A



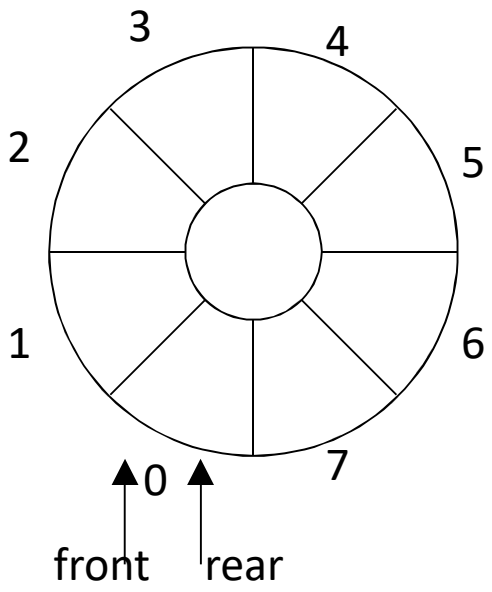
(c) Insert B



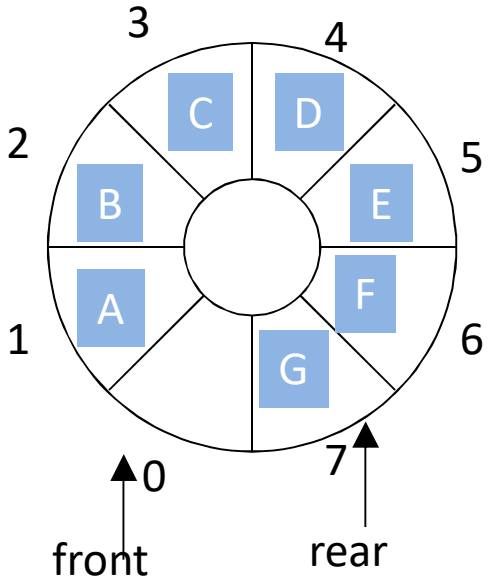
(d) Delete A

Circular Queue

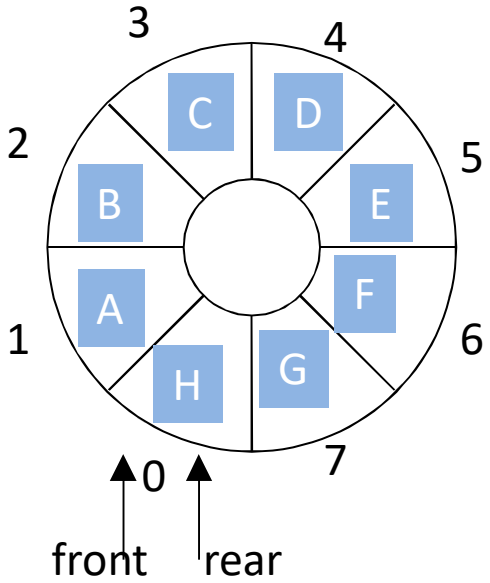
- 'Empty': `front == rear`
- 'Full': `front% M == (rear + 1)% M`
- Note) One space is always left empty to distinguish between 'blank' and 'full' state.



(a) Empty



(b) Full



(c) Error

Queue Operation

- Rotate the index into a circle using the modulo operation.

```
// Empty state detection function
int is_empty(QueueType * q)
{
        return (q->front == q->rear);
}
// Full state detection function
int is_full(QueueType * q)
{
        return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front);
}
```

Queue Operation

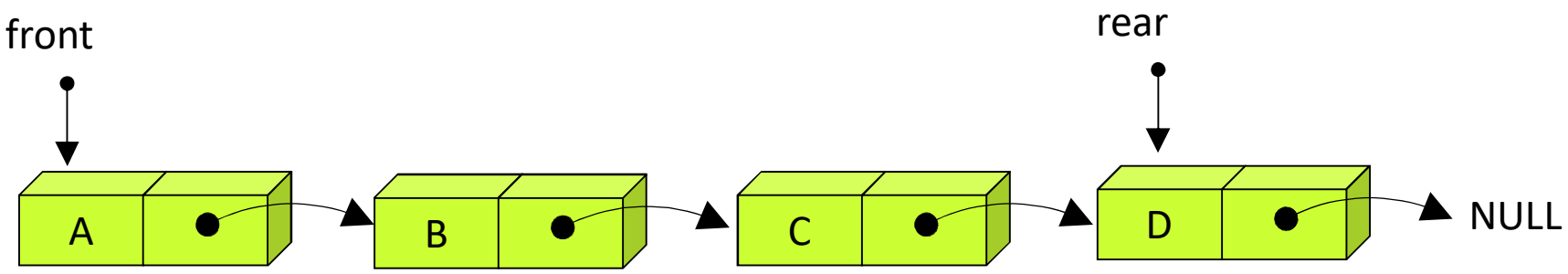
- Rotate the index into a circle using the modulo operation.

```
// Insert function
void enqueue(QueueType * q, element item)
{
        if (is_full(q))
                printf("Queue is full\n");
        q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
        q->queue[q->rear] = item;
}
// delete function
element dequeue(QueueType * q)
{
        if (is_empty(q))
                printf("Queue is empty\n");
        q->front = (q->front + 1) % MAX_QUEUE_SIZE;
        return q->queue[q->front];
}
```

```c
#define MAX_QUEUE_SIZE          100
typedef int element;
typedef struct QueueType {
        element  queue[MAX_QUEUE_SIZE];
        int front, rear;
} QueueType;

void error(char *message) { fprintf(stderr, "%s\n", message); exit(1); }
void init(QueueType *q) { q->front = q->rear = 0; }
int is_empty(QueueType *q) { return (q->front == q->rear); }
int is_full(QueueType *q) { return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front); }
void enqueue(QueueType *q, element item) {}
element dequeue(QueueType *q) {}

element peek(QueueType *q) {
        if (is_empty(q))      printf("Queue is empty\n");
        return q->queue[(q->front + 1) % MAX_QUEUE_SIZE];
}
void main()
{
        QueueType q;
        init(&q);
        printf("front=%d read=%d\n", q.front, q.rear);
        enqueue(&q, 1);
        enqueue(&q, 2);
        enqueue(&q, 3);
        printf("dequeue()=%d\n", dequeue(&q));
        printf("dequeue()=%d\n", dequeue(&q));
        printf("dequeue()=%d\n", dequeue(&q));
        printf("front=%d read=%d\n", q.front, q.rear);
}
```

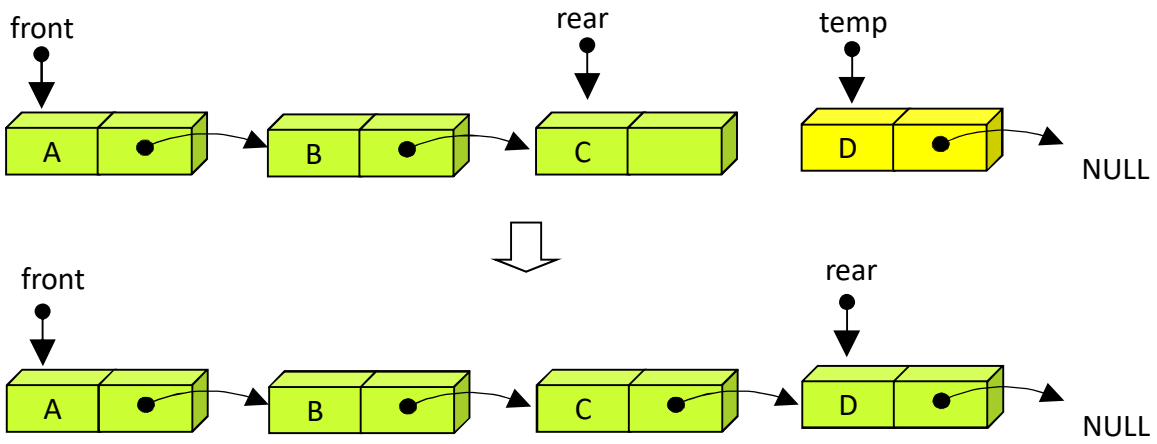
Linked Queue

- Linked queue: A queue that is implemented as a linked list.
  - 'front' is related to deletion and points to the element at the front of the linked list.
  - 'rear' is related to insertion and points to the element at the last of the linked list.
  - If there are no elements in the queue, front and rear are NULL



Insertion at Linked Queue

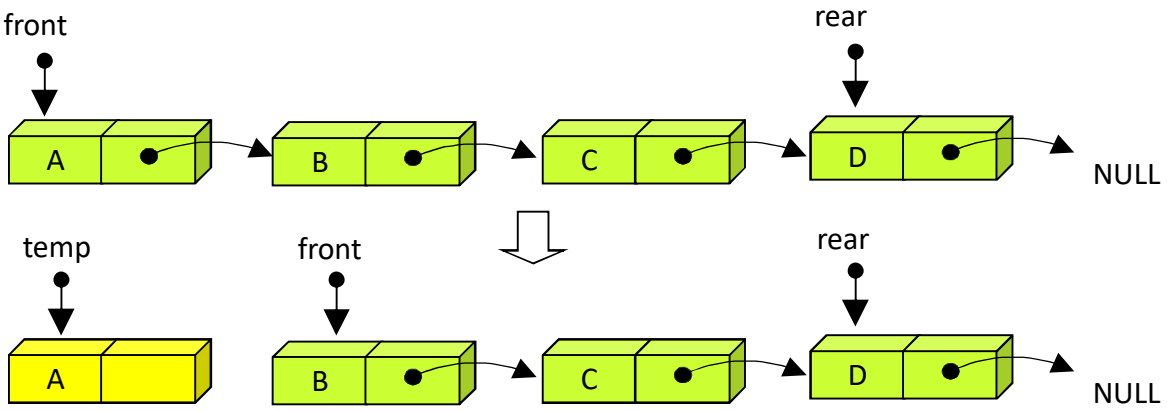
Insertion




```
void enqueue(QueueType *q, element item)
{
        QueueNode *temp = (QueueNode *)malloc(sizeof(QueueNode));
        if (temp == NULL)
                printf("Memory cannot be allocated.\n");
        else {
                temp->item = item;
                temp->link = NULL;
                if(is_empty(q)) {
                        q->front = temp;
                        q->rear = temp;
                }
                else {
                        q->rear->link = temp;
                        q->rear = temp;
                }
        }
}
```

Deletion at Linked Queue

Deletion

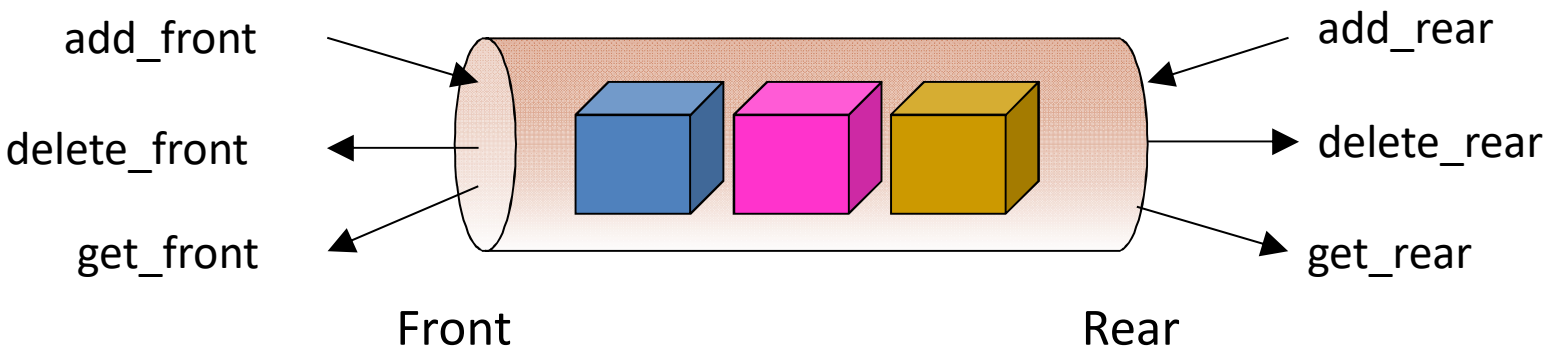



```
element dequeue(QueueType *q)
{
        QueueNode *temp = q->front;
        element item;
        if(is_empty(q))
                error("Queue is empty.");
        else {
                item = temp->item;
                q->front = q->front->link;
                if (q->front == NULL)
                        q->rear = NULL;
                free(temp);

                return item;
        }
}
```

```c
typedef int element;
typedef struct QueueNode {
        element item;
        struct QueueNode *link;
} QueueNode;

typedef struct QueueType {
        QueueNode *front;
        QueueNode *rear;
} QueueType;

void error(char *message) {     fprintf(stderr, "%s\n", message);          exit(1); }
int is_empty(QueueType *q) {    return (q->front == NULL); }
void init(QueueType *q) {       q->front = NULL;     q->rear = NULL;       }

void enqueue(QueueType *q, element item) {}
element dequeue(QueueType *q) {}

element peek(QueueType *q) {
        if (is_empty(q))     error("Queue is empty.");
        else {     return (q->front->item);        }
}

void main()
{
        QueueType q;

        init(&q);
        enqueue(&q, 1);
        enqueue(&q, 2);
        enqueue(&q, 3);

        printf("dequeue()=%d\n", dequeue(&q));
        printf("dequeue()=%d\n", dequeue(&q));
        printf("dequeue()=%d\n", dequeue(&q));
}
```

Deque

- Deque (Double-ended queue)
  - It can be inserted and deleted at the front and rear of the queue.

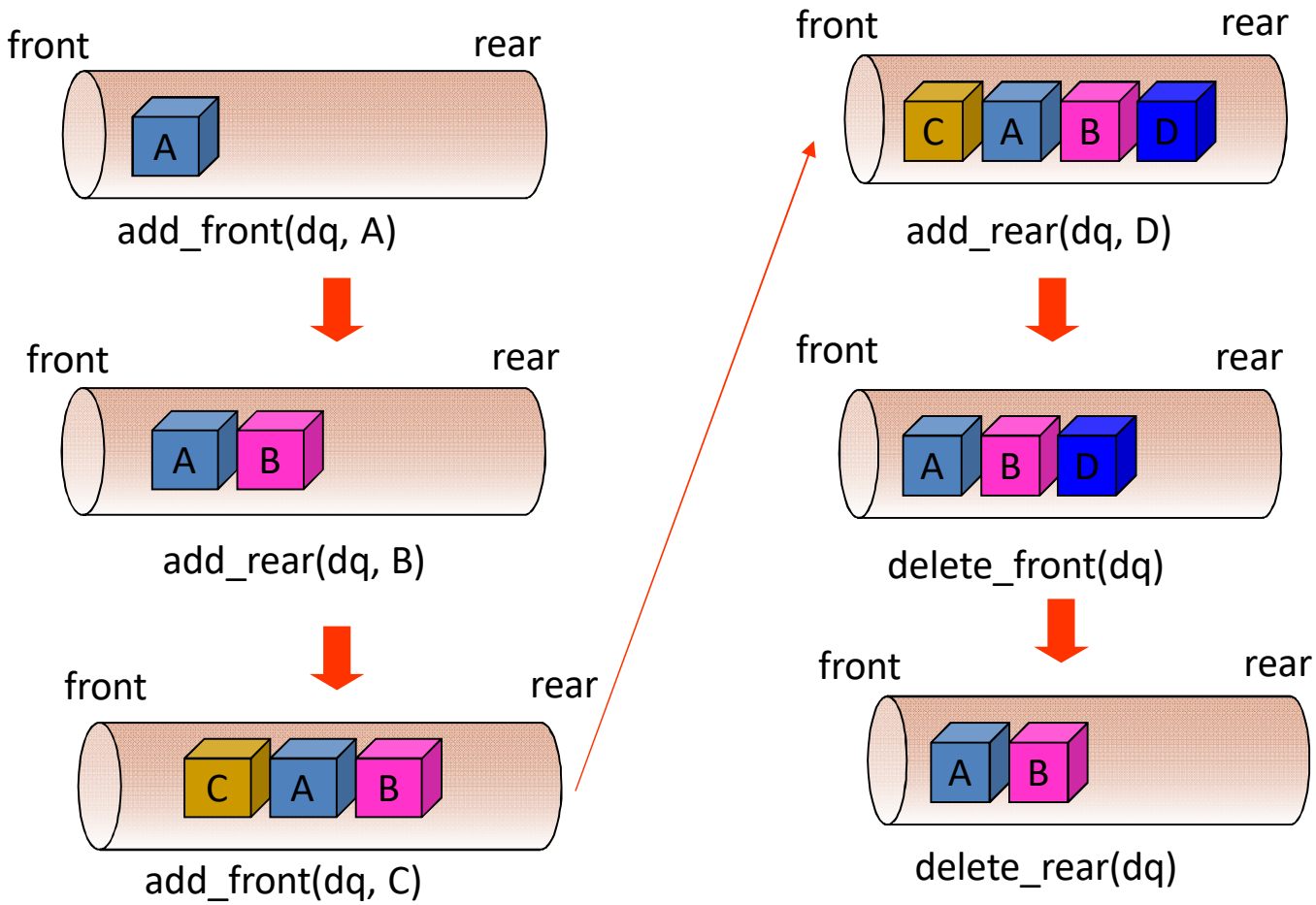


ADT of Deque

- Object: an ordered group of n elements

- Operation:
  create() :: = Creates a deque.
  init(dq) :: = Initialize the deque.
  is_empty(dq) :: = Checks if the deque is empty.
  is_full (dq) :: = Checks if the deque is saturated.
  add_front (dq, e) :: = Add an element before the deque.
  add_rear (dq, e) :: = Add an element after the deque.
  delete_front (dq) :: = Returns the element before the deque and then deletes it.
  delete_rear (dq) :: = Returns the element after the deque and then deletes it.
  get_front (q) :: = Returns the element before the deque without deleting it.
  get_rear (q) :: = Returns the element after the deque without deleting it.

Deque Operation



Deque Implementation

- Since the deque should be able to insert and delete from both front and rear, *doubly linked list* is used

```
typedef int element;

typedef struct DlistNode {
          element data;
          struct DlistNode *llink;
          struct DlistNode *rlink;
} DlistNode;

typedef struct DequeType {
          DlistNode *head;
          DlistNode *tail;
} DequeType;
```

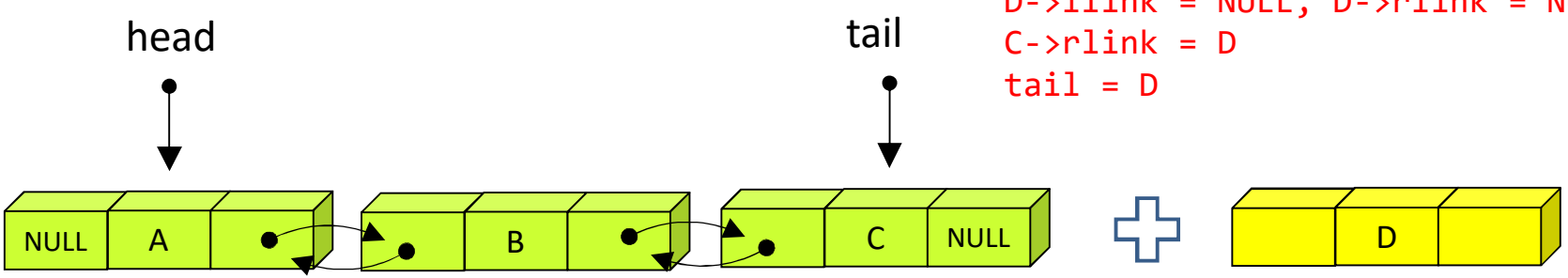
Insertion at Deque

```
DlistNode *create_node(DlistNode *llink, element item, DlistNode *rlink)
{
        DlistNode *node = (DlistNode *)malloc(sizeof(DlistNode));
        if (node == NULL) error("Memory allocation error");
        node->llink = llink;
        node->data = item;
        node->rlink = rlink;
        return node;
}
```

Insertion at Deque

- ‘add_rear’: add node ‘D’ at the rear

Case 1)



The followings should be updated!

```
D->llink = NULL, D->rlink = NULL
C->rlink = D
tail = D
```

Case 2)

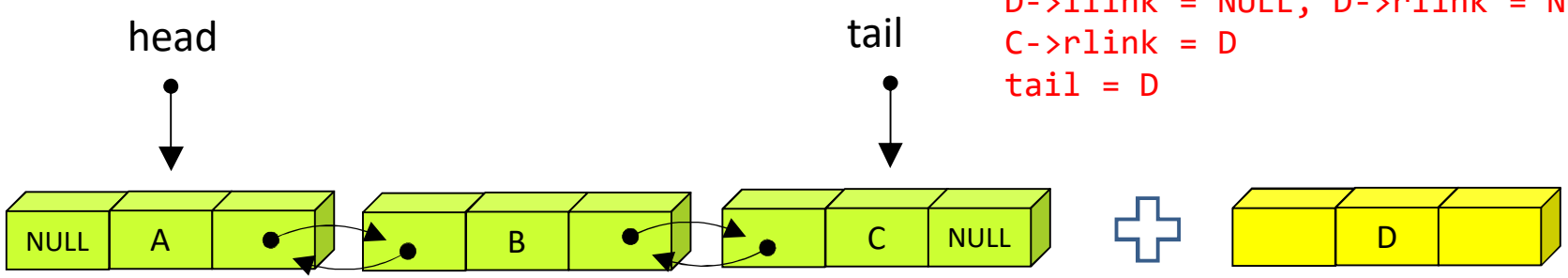
The followings should be updated!

```
D->llink = NULL, D->rlink = NULL
head = D
tail = D
```

Insertion at Deque

- ‘add_rear’: add node ‘D’ at the rear

Case 1)



The followings should be updated!

```
D->llink = NULL, D->rlink = NULL
C->rlink = D
tail = D
```

```
D->llink = C, D->rlink = NULL
```

```
void add_rear(DequeType *dq, element item)
{
        DlistNode *new_node = create_node(dq->tail, item, NULL);
        if (is_empty(dq))
                dq->head = new_node;
        else
                dq->tail->rlink = new_node;
        dq->tail = new_node;
}
```

```
C->rlink = D
```

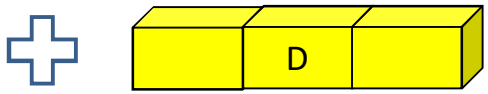
```
tail = D
```

Insertion at Deque

- ‘add_rear’: add node ‘D’ at the rear

The followings should be updated!
D->llink = NULL, D->rlink = NULL
head = D
tail = D

Case 2)

head == tail == NULL 

D->llink = NULL, D->rlink = NULL

```
void add_rear(DequeType *dq, element item)
{
        DlistNode *new_node = create_node(dq->tail, item, NULL);
        if (is_empty(dq))
                dq->head = new_node;
        else
                dq->tail->rlink = new_node;
        dq->tail = new_node;
}
```

head = D

tail = D

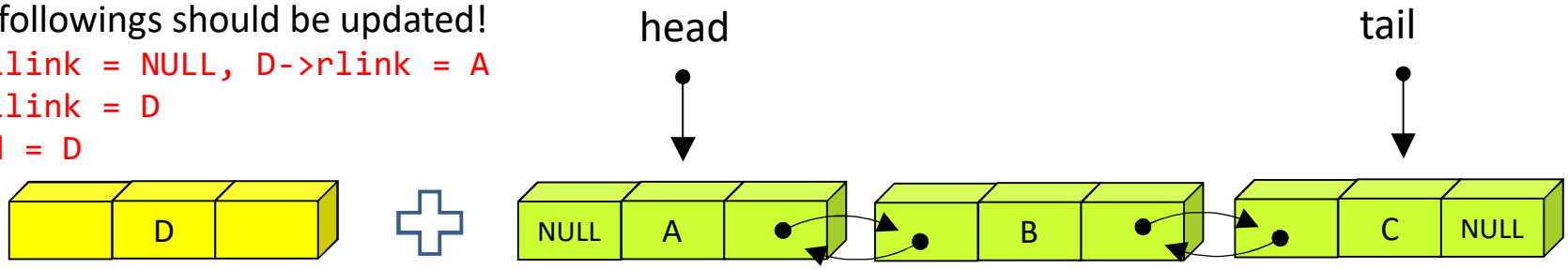
Insertion at Deque

- ‘add_front’: add node ‘D’ at the front

Case 1)

The followings should be updated!

```
D->llink = NULL, D->rlink = A
A->llink = D
head = D
```



Case 2)

The followings should be updated!

```
D->llink = NULL, D->rlink = NULL
head = D
tail = D
```

head = tail = NULL

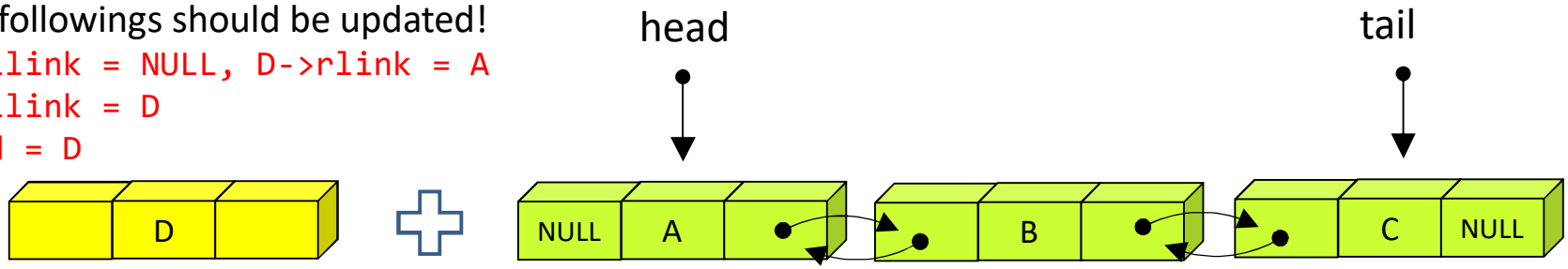
Insertion at Deque

- ‘add_front’: add node ‘D’ at the front

Case 1)

The followings should be updated!

```
D->llink = NULL, D->rlink = A
A->llink = D
head = D
```



D->llink = NULL, D->rlink = A

```
void add_front(DequeType *dq, element item)
{
        DlistNode *new_node = create_node(NULL, item, dq->head);
        if (is_empty(dq))
                dq->tail = new_node;
        else
                dq->head->llink = new_node;
        dq->head = new_node;
}
```

A->llink = D

head = D

Insertion at Deque

- ‘add_front’: add node ‘D’ at the front

Case 2)

The followings should be updated!
D->llink = NULL, D->rlink = NULL
head = D
tail = D

head = tail = NULL

D->llink = NULL, D->rlink = NULL

```
void add_front(DequeType *dq, element item)
{
        DlistNode *new_node = create_node(NULL, item, dq->head);
        if (is_empty(dq))
                dq->tail = new_node;
        else
                dq->head->llink = new_node;
        dq->head = new_node;
}
```

tail = D

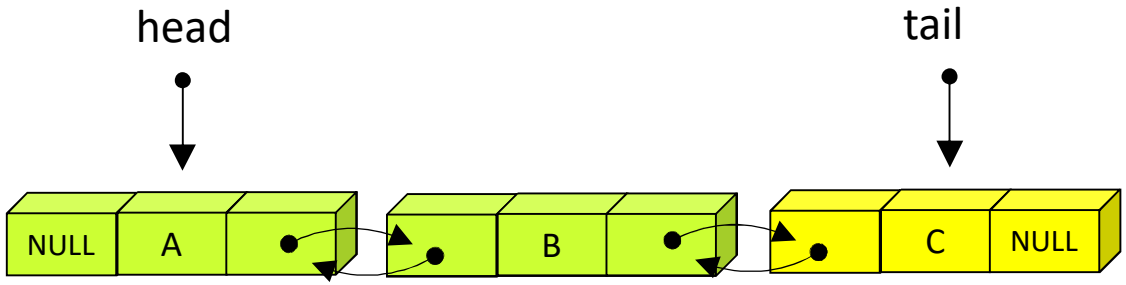
head = D

Deletion at Deque

- ‘delete_rear’: delete a node at the rear

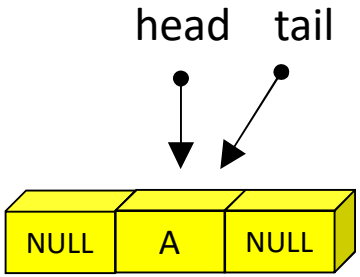
Case 1)

The followings should be updated!
`tail=B, B->rlink=NULL`



Case 2)

The followings should be updated!
`head=NULL, tail=NULL`

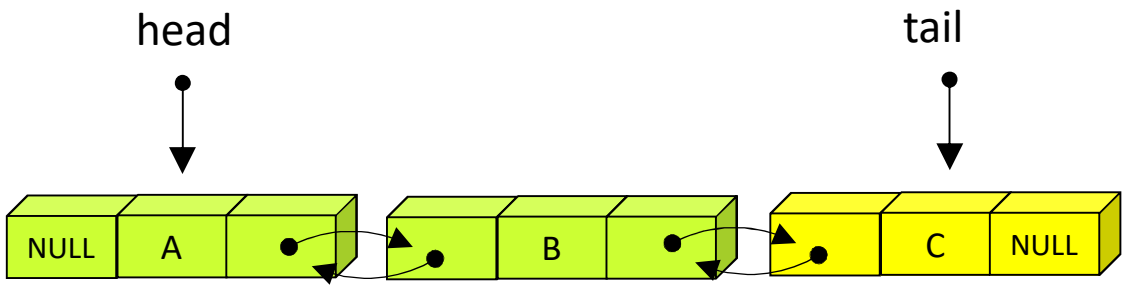


Deletion at Deque

- ‘delete_rear’: delete a node at the rear

Case 1)

The followings should be updated!
tail=B, B->rlink=NULL




```
element delete_rear(DequeType *dq)
{
        element item;
        DlistNode *removed_node;

        if (is_empty(dq))  printf("Deque is empty\n");
        else {
                removed_node = dq->tail;   // Node to be deleted
                item = removed_node->data;
                dq->tail = dq->tail->llink; // Change tail pointer
                free(removed_node);
                if (dq->tail == NULL) // If empty, after removing the node
                        dq->head = NULL;
                else
                        dq->tail->rlink = NULL;
        }
        return item;
}
```

tail=B

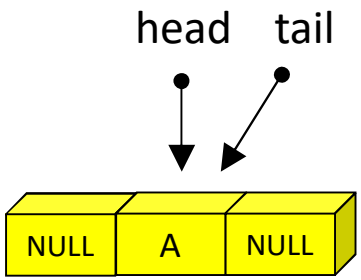
B->rlink=NULL

Deletion at Deque

- ‘delete_rear’: delete a node at the rear

Case 2)

The followings should be updated!
head=NULL, tail=NULL




```
element delete_rear(DequeType *dq)
{
        element item;
        DlistNode *removed_node;

        if (is_empty(dq))  printf("Deque is empty\n");
        else {
                removed_node = dq->tail;   // Node to be deleted
                item = removed_node->data;
                dq->tail = dq->tail->llink; // Change tail pointer
                free(removed_node);
                if (dq->tail == NULL) // If empty, after removing the node
                        dq->head = NULL;
                else
                        dq->tail->rlink = NULL;
        }
        return item;
}
```

tail=NULL

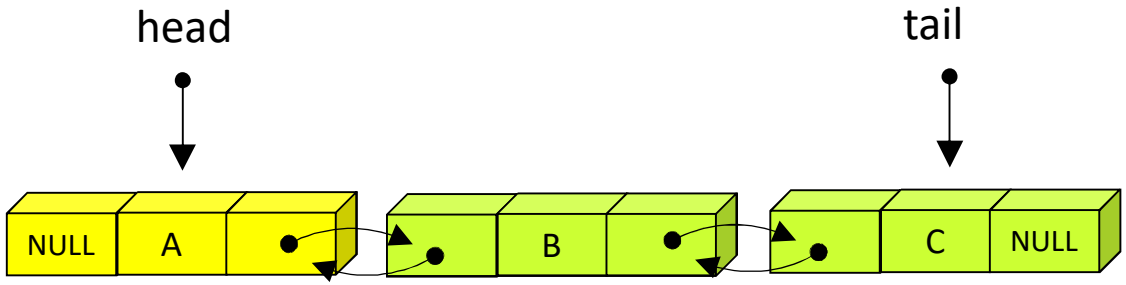
head=NULL

Deletion at Deque

- ‘delete_front’: delete a node at the front

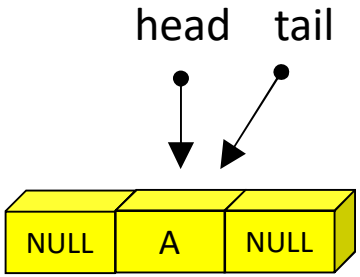
Case 1)

The followings should be updated!
`head=B, B->llink=NULL`



Case 2)

The followings should be updated!
`head=NULL, tail=NULL`

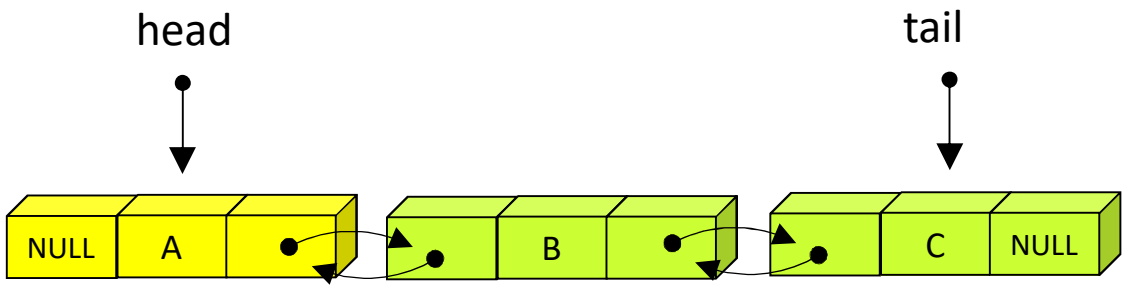


Deletion at Deque

- ‘delete_front’: delete a node at the front

Case 1)

The followings should be updated!
head=B, B->llink=NULL




```
element delete_front(DequeType *dq)
{
        element item;
        DlistNode *removed_node;

        if (is_empty(dq))  printf("Deque is empty\n");
        else {
                removed_node = dq->head;   // Node to be deleted
                item = removed_node->data;
                dq->head = dq->head->rlink; // Change head pointer
                free(removed_node);
                if (dq->head == NULL) // If empty, after removing the node
                        dq->tail = NULL;
                else
                        dq->head->llink = NULL;
        }
        return item;
}
```

head=B

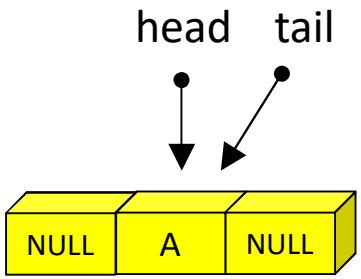
B->llink=NULL

Deletion at Deque

- ‘delete_front’: delete a node at the front

Case 2)

The followings should be updated!
head=NULL, tail=NULL



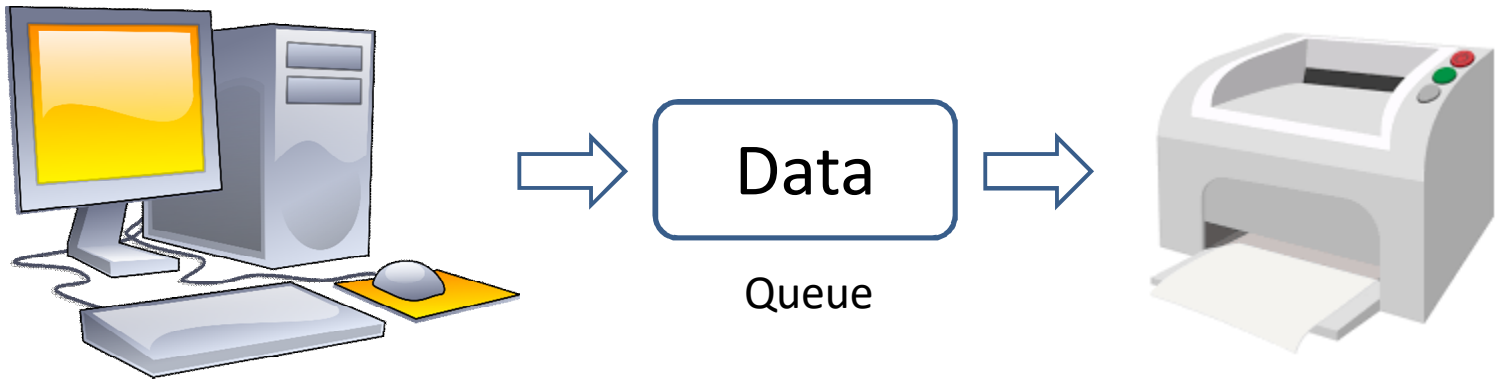

```
element delete_front(DequeType *dq)
{
        element item;
        DlistNode *removed_node;

        if (is_empty(dq))  printf("Deque is empty\n");
        else {
                removed_node = dq->head;   // Node to be deleted
                item = removed_node->data;
                dq->head = dq->head->rlink; // Change head pointer
                free(removed_node);
                if (dq->head == NULL) // If empty, after removing the node
                        dq->tail = NULL;
                else
                        dq->head->llink = NULL;
        }
        return item;
}
```

head=NULL

tail=NULL

Queue Application: Buffer

- Buffer
  - Queues can work as buffers that coordinate an interaction between two processes running at different speeds.
    - Ex) A printing buffer between the CPU and the printer, or a keyboard buffer between the CPU and the keyboard
  - Buffer links between a producer process that produces data and a consumer process that consumes data.



Queue Application: Buffer

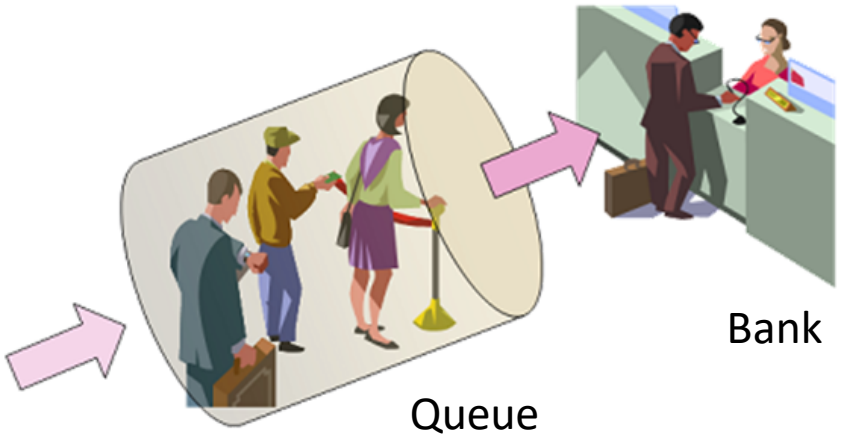
```
QueueType buffer;

/* Producer process */
producer()
{
        while (1) {
                Produce data;
                while (lock(buffer) != SUCCESS);
                if (!is_full(buffer)) {
                        enqueue(buffer, data);
                }
                unlock(buffer);
        }
}
/* Consumer process */
consumer()
{
        while (1) {
                while (lock(buffer) != SUCCESS);
                if (!is_empty(buffer)) {
                        data = dequeue(buffer);
                        Consume data;
                }
                unlock(buffer);
        }
}
```

lock(): function to avoid producer and consumer access the buffer simultaneously

More details will be provided in the lecture 'Operating system (OS)'

Queue Application: Simulation

- Simulation
  - Queue is used to simulate and analyze system characteristics according to queuing theory.
  - The queuing model consists of a server that performs services for customers and a customer who receives services.
  - Example
    In the process of getting customers in and out of the bank, we wish to calculate the average waiting time of customers



Queue Application: Simulation

```
typedef struct element{
          int id;
          int arrival_time;
          int service_time;
} element;                    // Customer structure

typedef struct QueueType {
          element  queue[MAX_QUEUE_SIZE];
          int  front, rear;
} QueueType;
QueueType queue;

// Real random number generation function between 0 and 1
double random() {
          return rand() / (double)RAND_MAX;
}

// Various state variables needed for simulation
int duration = 10; // Simulation time
double arrival_prob = 0.7; // Average number of customers arriving in one time unit
int max_serv_time = 5; // maximum service time for one customer
int clock;

// Results of the simulation
int customers; // Total number of customers
int served_customers; // Number of customers served
int waited_time; // Time the customers waited
```

Queue Application: Simulation

```
// Generate a random number.
// If it is smaller than 'arrival_prov', assume that new customer comes in the bank.
int is_customer_arrived()
{
        if (random() <arrival_prob)
                return TRUE;
        else return FALSE;
}
// Insert newly arrived customer into queue
void insert_customer(int arrival_time)
{
        element customer;

        customer.id = customers++;
        customer.arrival_time = arrival_time;
        customer.service_time = (int)(max_serv_time * random()) + 1;
        enqueue(&queue, customer);
        printf("Customer %d comes in %d minutes. Service time is %d minutes.",
                customer.id, customer.arrival_time, customer.service_time);
}
```

The service time required by the customer is generated using a random number.

Queue Application: Simulation

```
// Retrieve the customer waiting in the queue and return the customer's service time.
int remove_customer()
{
        element customer;
        int service_time = 0;

        if (is_empty(&queue)) return 0;
        customer = dequeue(&queue);
        service_time = customer.service_time - 1;
        served_customers++;
        waited_time += clock - customer.arrival_time;
        printf("Customer %d starts service in %d minutes. Wait time was %d minutes.",
                customer.id, clock,clock - customer.arrival_time);
        return service_time;
}
```

```
// Print the statistics.
void print_stat()
{
        printf("Number of customers served = %d", served_customers);
        printf("Total wait time = %d minutes", waited_time);
        printf("Average wait time per person =% f minutes",
                (double)waited_time / served_customers);
        printf("Number of customers still waiting =% d",
                customers - served_customers);
}
```

Queue Application: Simulation

```
// Simulation program
void main()
{
        int service_time = 0;

        clock = 0;
        while (clock < duration) {
                clock++;
                printf("Current time=%d\n", clock);
                if (is_customer_arrived()) {
                        insert_customer(clock);
                }

                // Check if the customer who is receiving the service is finished.
                if (service_time > 0)       // the customer is receiving service
                        service_time--;

                // no customer is receiving service.
                // So, take out a customer from the queue and start the service.
                else {
                        service_time = remove_customer();
                }
        }
        print_stat();
}
```