CSCI2100C Lab 2

Prepared by:

Lo Chun Hei

Reminders

- Please remember your password for the online judge
- Please start your assignment early
  - And report any issues as early as possible. Some issues regarding the registration of the OJ were reported not until some hours before the deadline. For such cases of failure of code submission, we may not grade your lab.
  - Penalty:
    - -10 marks/day pro rata for first two days after deadline
    - -10 marks/hour pro rata afterwards (so you get 0 marks if you submit 2 day 8 hours after the deadline)
- Grading is based on the **last** submission
- Write your own code
  - We will check your code
  - Suspected cases of plagiarism will be reported
- Questions?

Agenda

- Array
- Linked List
- Stack
- Queue
- Example Code Discussion
- Overview of Lab 2 Problems

Agenda

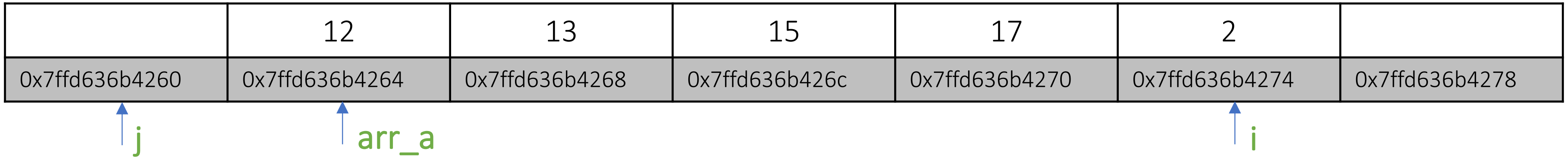
- Array
- Linked List
- Stack
- Queue
- Example Code Discussion
- Overview of Lab 2 Problems

Array

Properties

- An array is a collection of items stored at contiguous memory locations

```
int arr_a[4] = {12,13,15,17}; /* arr_a is an array of 4 integers */
int i = 2, j;
```



	12	13	15	17	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

- Array variable points to the first position of the array in memory
  - Random access (base address + offset (assumed to be constant-time))
- We need to specify the size when we create the array
  - Fixed number of elements decided at the time of declaration
  - Insertion and Deletion is $O(n)$
    - Often accompanied by O(n) searching for position of insertion/deletion as well

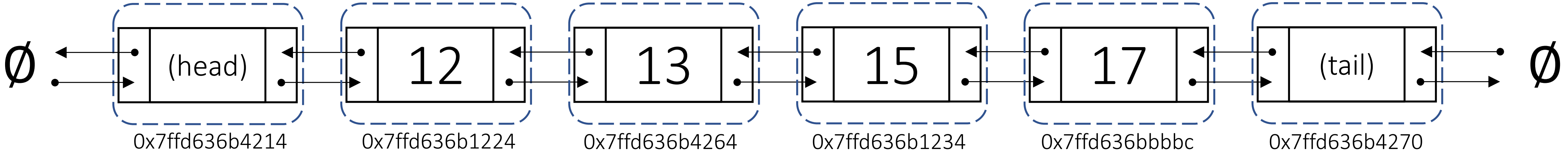
Agenda

- Array
- **Linked List**
- Stack
- Queue
- Example Code Discussion
- Overview of Lab 2 Problems

Linked List

Properties

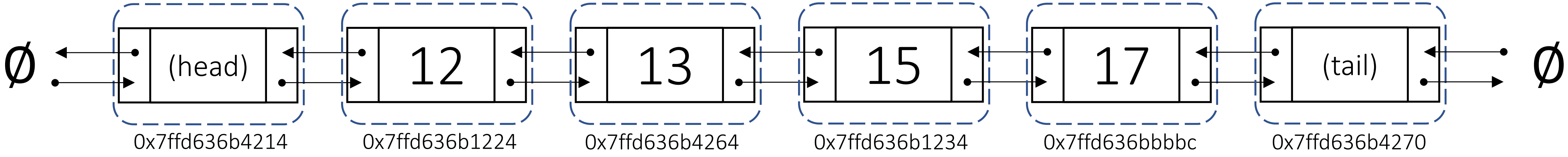
- A linked list a collection of items that are linked using pointers
  - Stored not necessarily at contiguous memory locations:



- A (doubly) linked list consists of one or more node objects, where each node stores:
  - The data to be stored
  - Pointers to the previous node
  - Pointer to the next node
- A head node and a tail node is maintained

Linked List

Properties



- Only neighbouring nodes are accessible
  - Accessing a particular node is $O(n)$
- Nodes can be created dynamically at runtime
  - Insertion and Deletion is $O(1)$ (if position(address) of insertion/deletion is known)
    - Often accompanied by O(n) searching for position of insertion/deletion
  - Unrestricted number of nodes

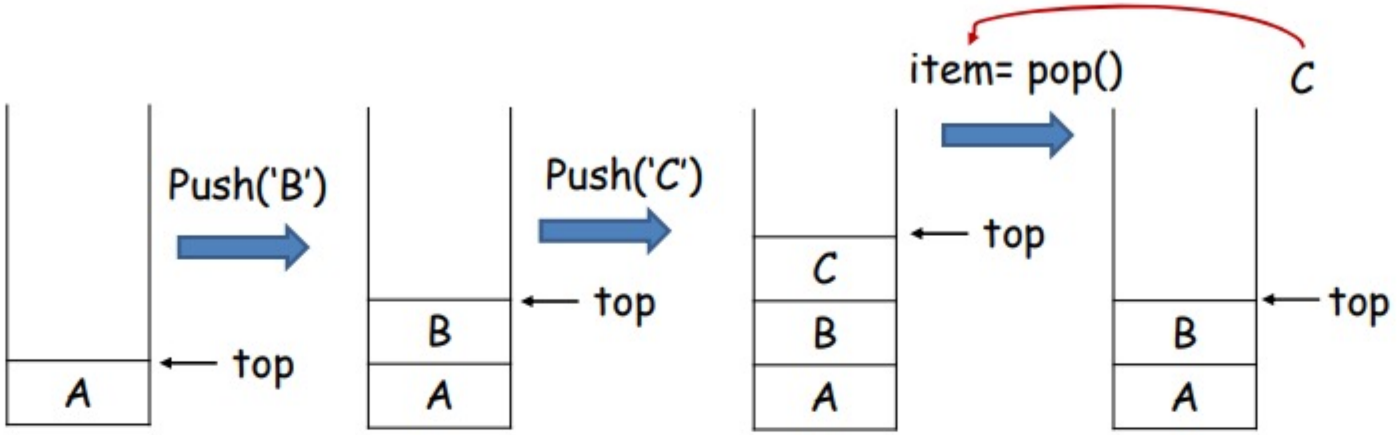
Agenda

- Array
- Linked List
- Stack
- Queue
- Example Code Discussion
- Overview of Lab 2 Problems

Stack

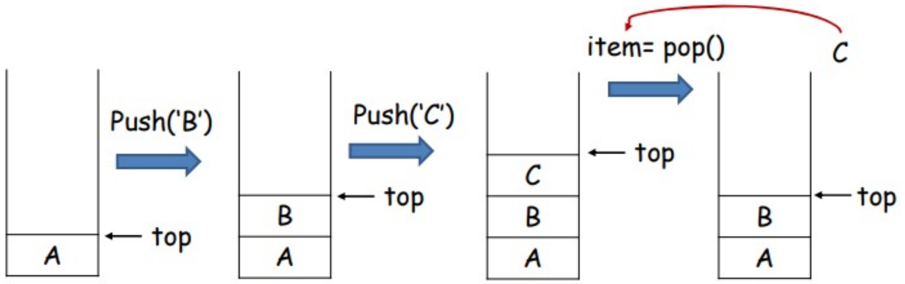
Properties

- A stack stores a set S of elements that only allows two **constrained** updates (a.k.a. Last-In-First-Out (LIFO)):
  - Push(e): add a new element to S
  - Pop(): removes the most recently added element from S
- Only the **top** element is accessible

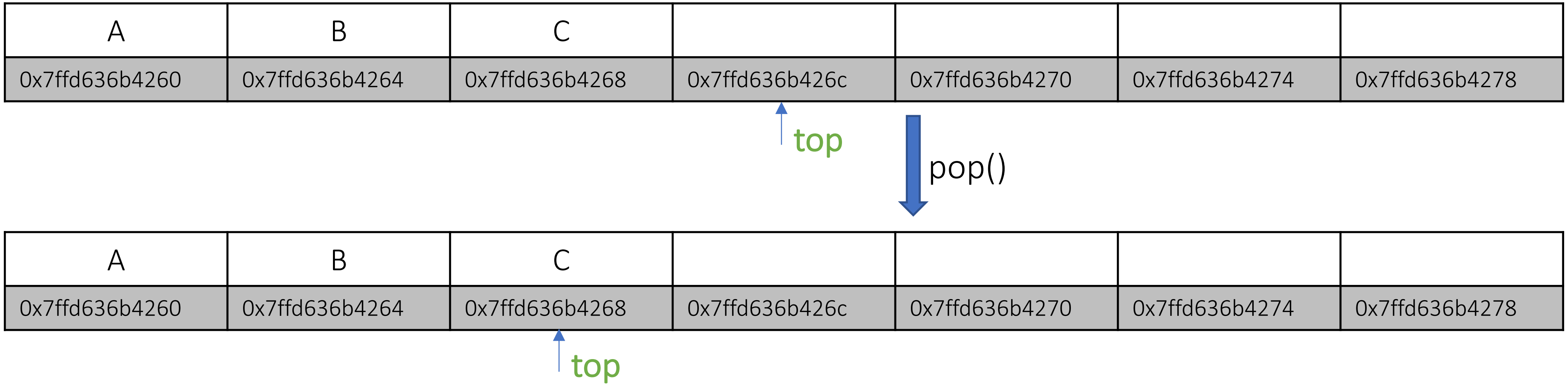


Stack

Implementation



- Implementing with array:



A	B	C				
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

top

pop()

A	B	C				
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

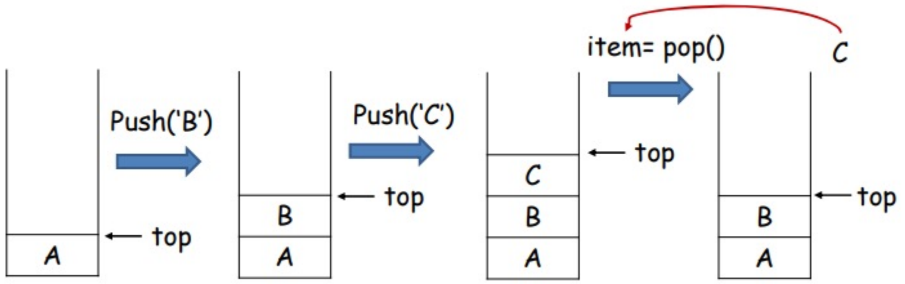
top

For the arrow representation, please note that **top** is an array index.

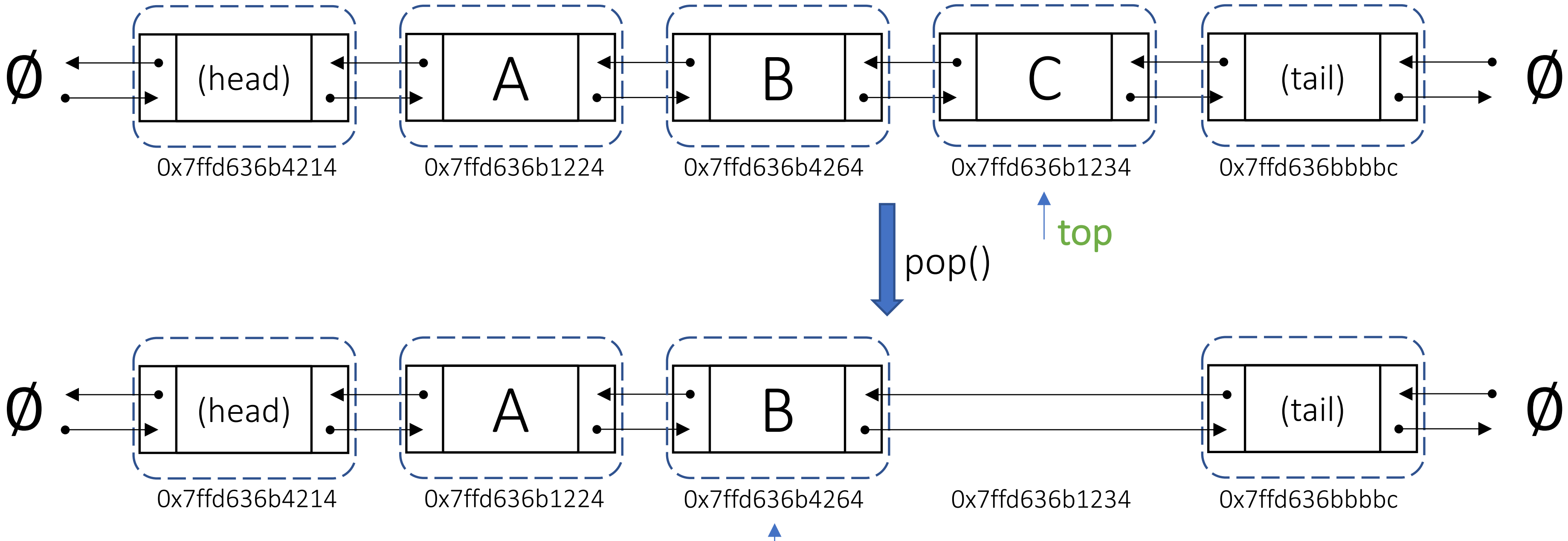
- Removal of element C is optional (why?)
  - Or can you actually remove an element from an array?
  - Change with some value that you never use; or use pointer to refer to another address

Stack

Implementation



- Implementing with linked list:



For the arrow representation, please note that **top** is a pointer variable.

- Removal of element C (from …) is a good practice (why?)
  - From memory, is no longer accessible to the element for better memory usage

Stack

Implementation

- For both array and linked list implementation:
  - Push and pop operation requires only $O(1)$ running cost
- Array implementation
  - More efficient in terms of memory
    - linked lists requires additional storage for next and previous referencing elements.
- Linked list implementation
  - No constraint on the capacity
  - Better utilized memory
    - Array uses fixed size of memory while linked list consumes memory that is proportional to the number of elements in the stack

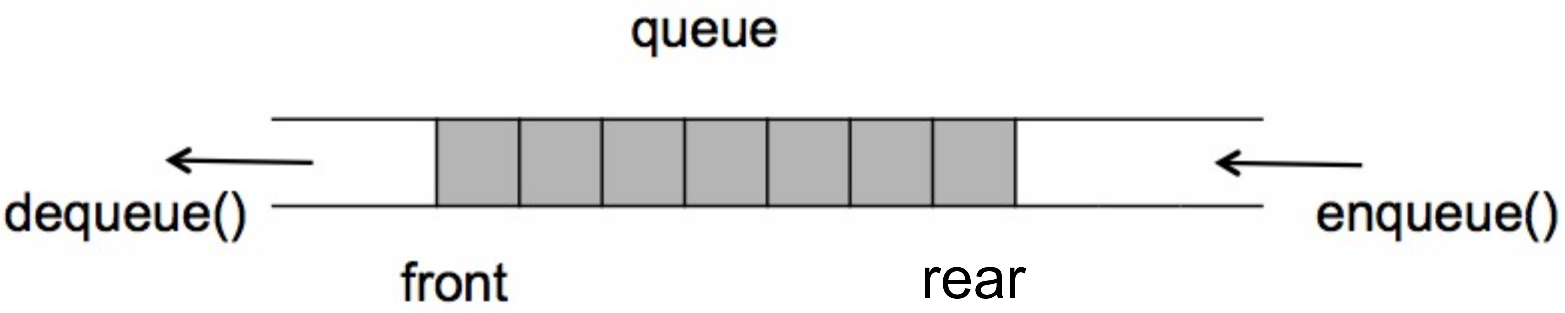
Agenda

- Array
- Linked List
- Stack
- **Queue**
- Example Code Discussion
- Overview of Lab 2 Problems

Queue

Properties

- A queue stores a set S of elements that only allows two **constrained** updates (a.k.a. First-In-First-Out (FIFO)):
  - Enqueue(e): insert a new element e into S
  - Dequeue: remove the least recently inserted element from S, and return it.
- Only the **front** (also **rear**) element is accessible



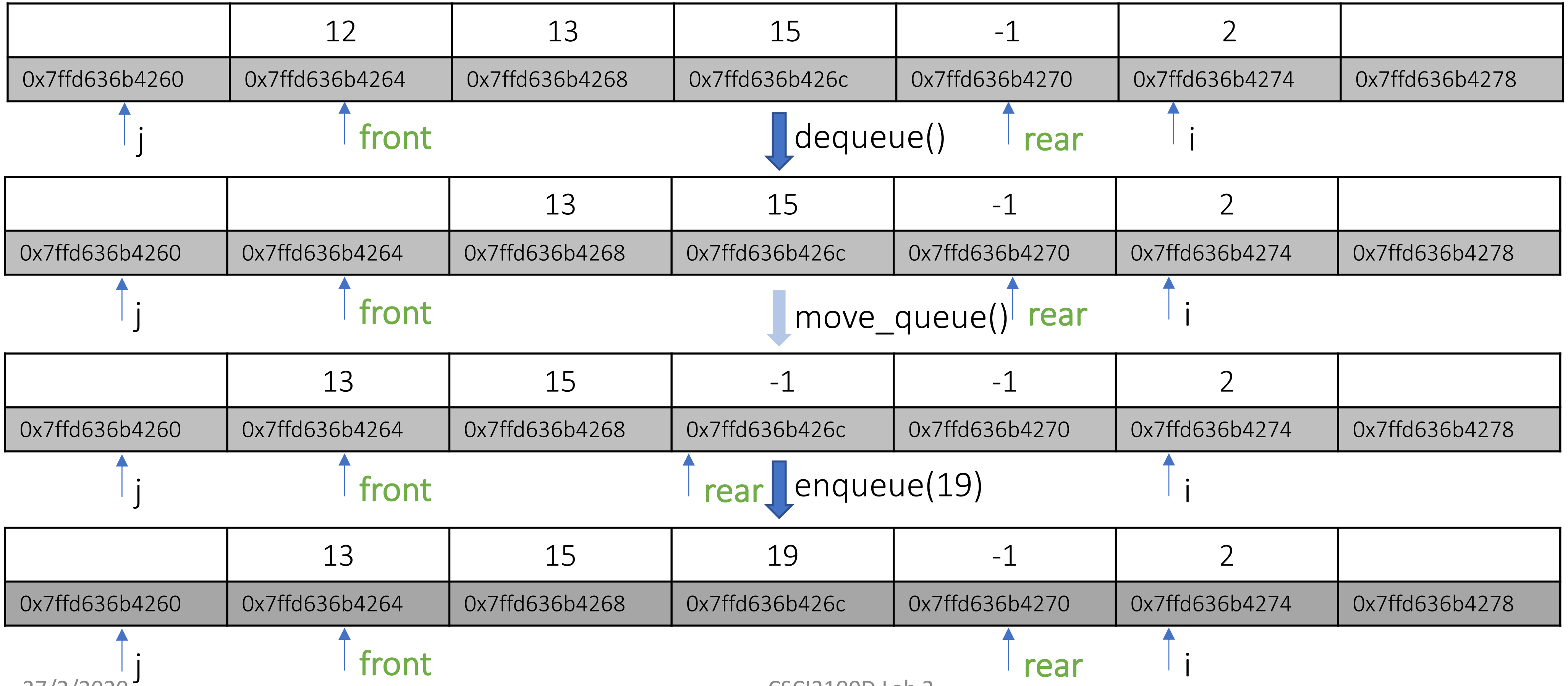
for

Queue

Implementation

```
int queueA[4] = {12,13,15,-1}; /* treat -1 as empty element here */
int i = 2, j, front = 0, rear = 3;
while (1) {
    dequeue();
    move_queue();
    rear -= 1;
    enqueue(19);
    rear += 1;
}
```

- Implementing with array: (keeping the front fixed)



	12	13	15	-1	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278
j	front			rear	i	

dequeue()

		13	15	-1	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278
j	front			rear	i	

move_queue()

	13	15	-1	-1	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278
j	front		rear		i	

enqueue(19)

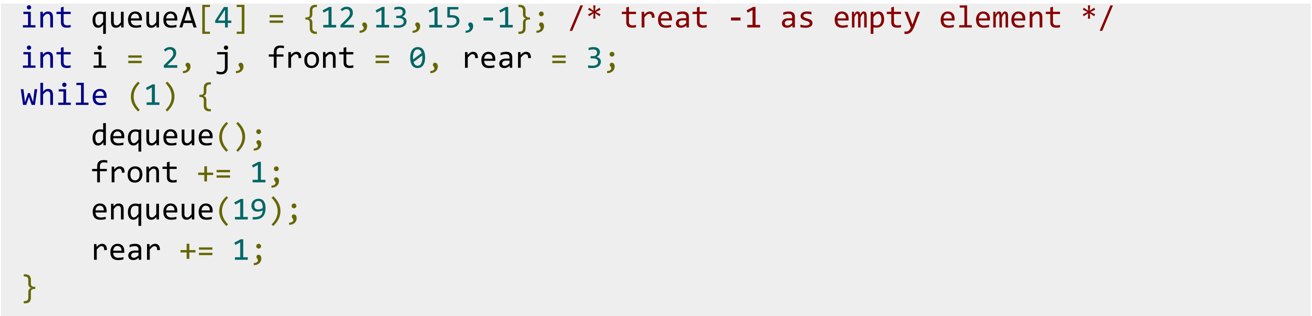
	13	15	19	-1	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278
j	front			rear	i	

For the arrow representation, please note that **front** and **rear** are array indices, while **i** and **j** are variables

for

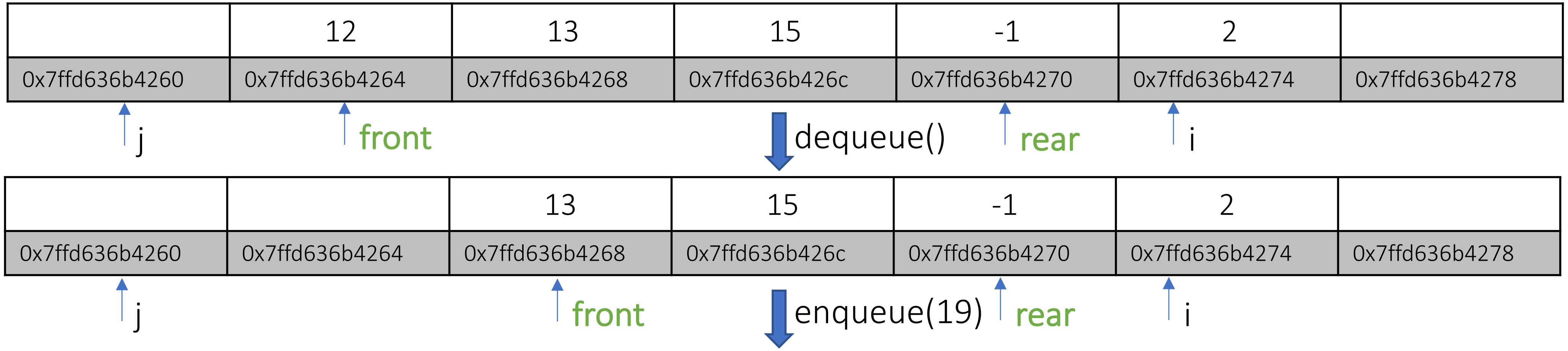
Queue

Implementation



```
int queueA[4] = {12,13,15,-1}; /* treat -1 as empty element */
int i = 2, j, front = 0, rear = 3;
while (1) {
    dequeue();
    front += 1;
    enqueue(19);
    rear += 1;
}
```

- Implementing with array: (dynamic front)



	12	13	15	-1	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

		13	15	-1	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

Enqueuing fails because rear equals size of array

- The queue is “crawling” in the memory
  - Wasting the memory whose address precedes front

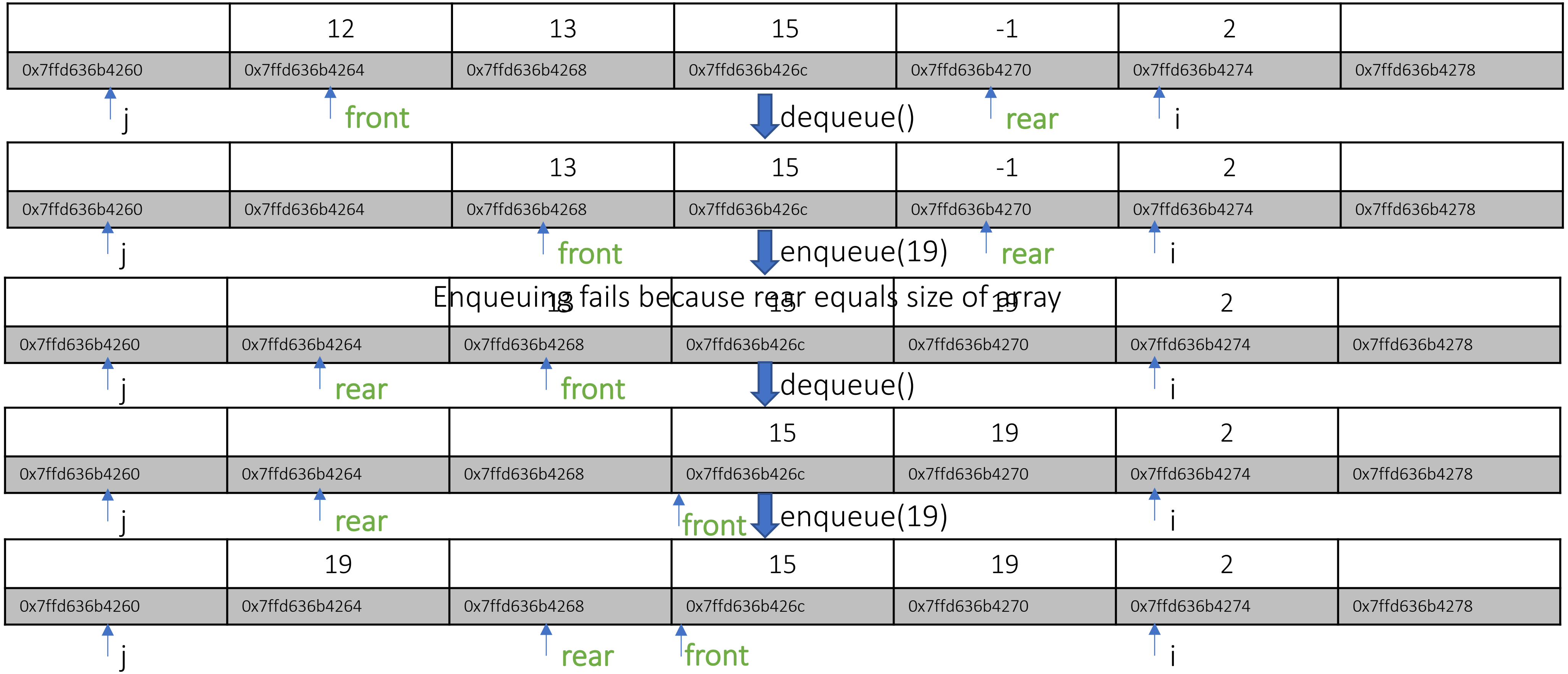
for

Queue

Implementation

```c
int queueA[4] = {12,13,15,-1}; /* treat -1 as empty element */
int i = 2, j, front = 0, rear = 3;
while (1) {
    dequeue();
    front = (front + 1) % 4;
    enqueue(19);
    rear = (rear + 1) % 4;
}
```

- Implementing with array: (circular queue)



	12	13	15	-1	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

		13	15	-1	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

Enqueuing fails because rear equals size of array

		13	15	19	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

Queue is Full (why?)

			15	19	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

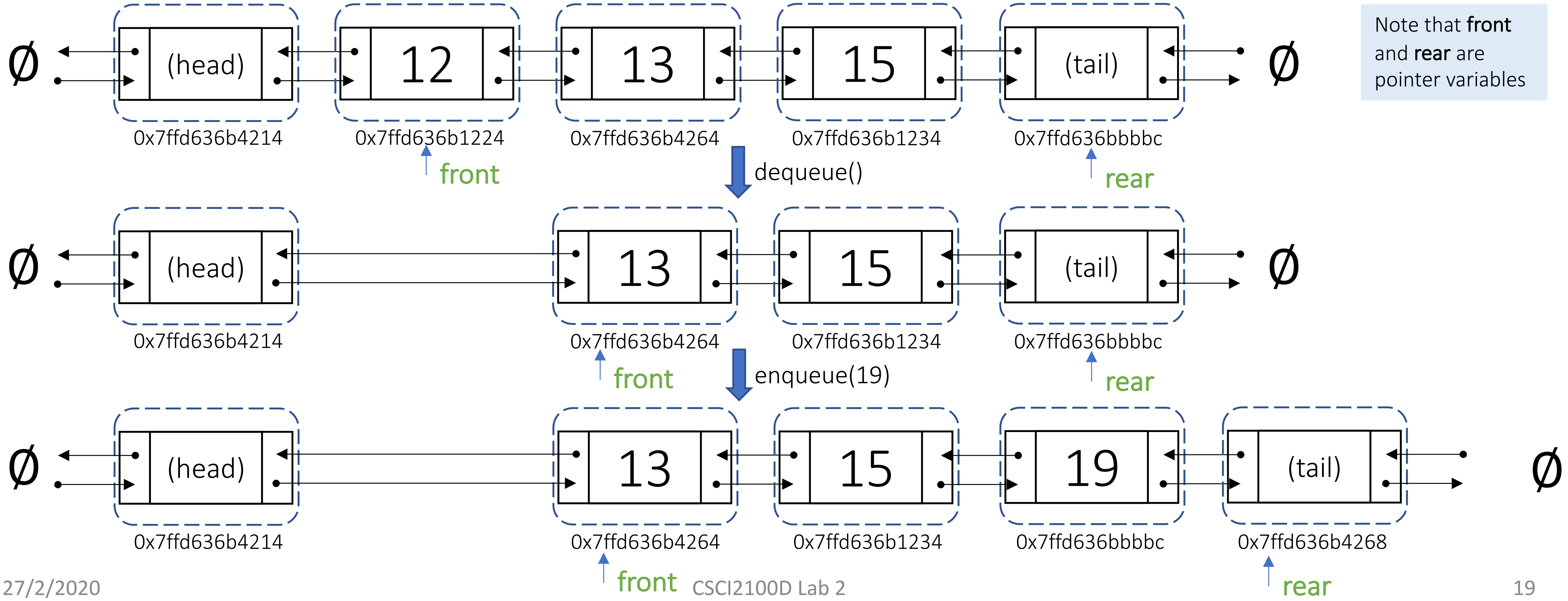
	19		15	19	2	
0x7ffd636b4260	0x7ffd636b4264	0x7ffd636b4268	0x7ffd636b426c	0x7ffd636b4270	0x7ffd636b4274	0x7ffd636b4278

Queue is Full (why?)

Queue

Implementation

- Implementing with linked list:
  - Similar to implementation of stack



Note that **front** and **rear** are pointer variables

Queue

Implementation

- For both array and linked list implementation:
    - Enqueue and Dequeue operation requires only $O(1)$ running cost
- Array implementation
    - More efficient in terms of memory
        - linked lists requires additional storage for next and previous referencing elements.
- Linked list implementation
    - No constraint on the capacity
    - Better utilized memory
        - Array uses fixed size of memory while linked list consumes memory that is proportional to the number of elements in the queue
    - Easy to design

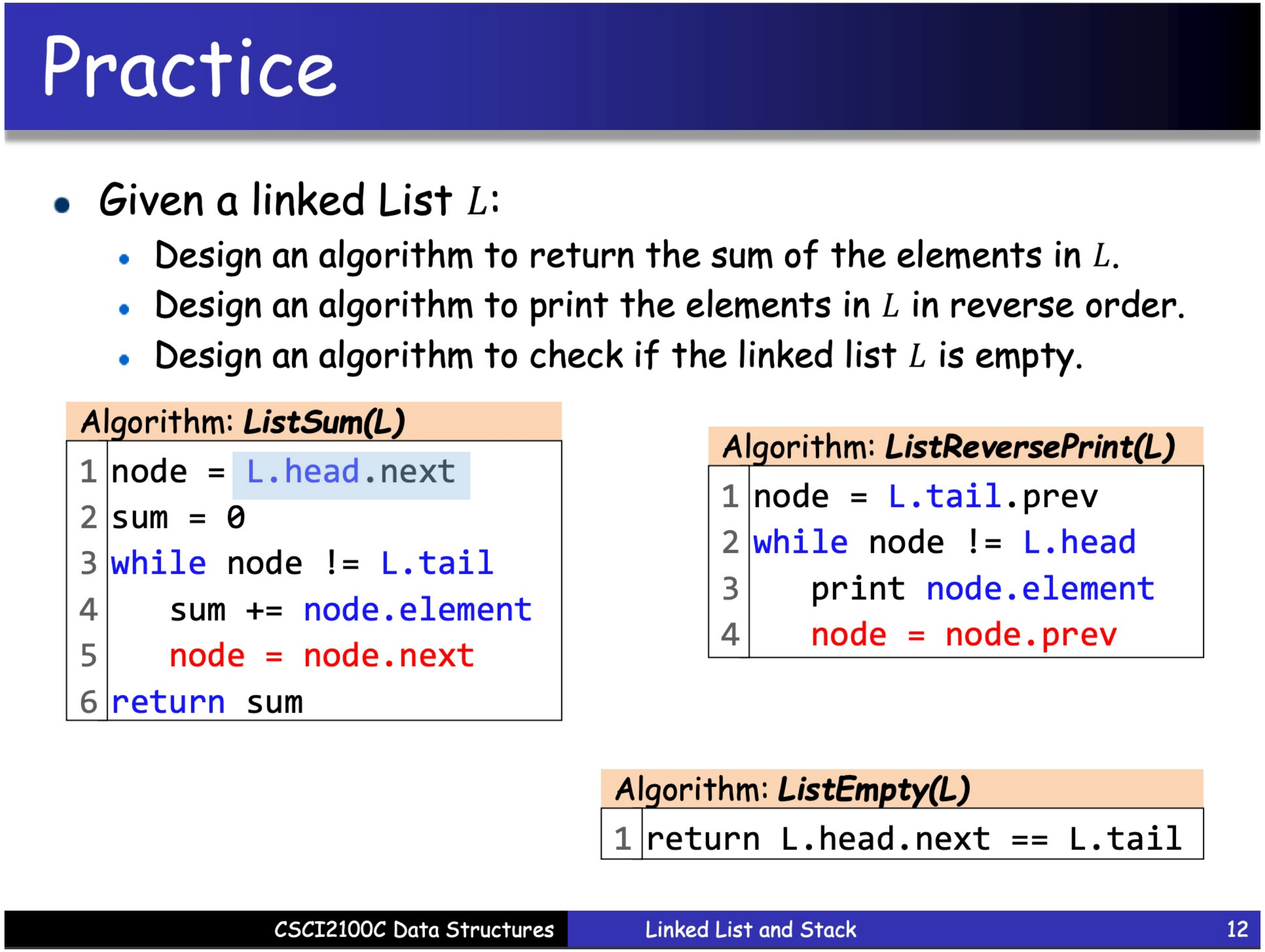
Agenda

- Array
- Linked List
- Stack
- Queue
- Example Code Discussion
- Overview of Lab 2 Problems

Example Code Discussion

Reminder on Lecture Material

- Pseudo code vs. C code
- Normally we access a node in linked list by pointer
  - Nodes of linked list are created dynamically (at runtime) and they can only be accessed through pointer
  - L->head->next



Practice

- Given a linked List L:
  - Design an algorithm to return the sum of the elements in L.
  - Design an algorithm to print the elements in L in reverse order.
  - Design an algorithm to check if the linked list L is empty.

Algorithm: ***ListSum(L)***

```
node = L.head.next
sum = 0
while node != L.tail
    sum += node.element
    node = node.next
return sum
```

Algorithm: ***ListReversePrint(L)***

```
node = L.tail.prev
while node != L.head
    print node.element
    node = node.prev
```

Algorithm: ***ListEmpty(L)***

```
return L.head.next == L.tail
```

CSCI2100C Data Structures Linked List and Stack 12

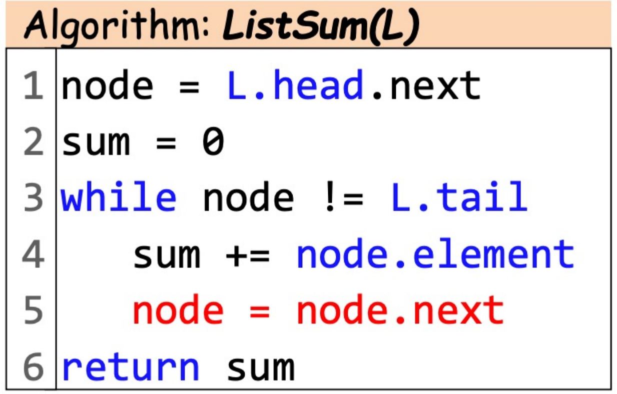
Example Code Discussion

Comparison of ListSum in Pseudo Code and C

- In C, we write ListSum as:

```c
typedef struct Node * PtrToNode;
struct Node {
    int element;
    PtrToNode next;
    PtrToNode prev;
};
typedef struct ListRecord * List;
struct ListRecord {
    PtrToNode head;
    PtrToNode tail;
};
```

```c
int ListSum(List L){
    PtrToNode node = L->head->next;
    int sum = 1;
    while (node != L->tail){
        sum += node->element;
        node = node->next;
    }
    return sum;
}
```



Algorithm: *ListSum(L)*

```
node = L.head.next
sum = 0
while node != L.tail
    sum += node.element
    node = node.next
return sum
```

Agenda

- Array
- Linked List
- Stack
- Queue
- Example Code Discussion
- Overview of Lab 2 Problems

Lab 2

Problem 1

- Straight forward implementation of stack with array
- Key points to note:
  - Conditions for fullness/emptiness of stack

Code segment to complete:

```
struct Stack {
    int * arr;
    int capacity;
    int top;
};
int IsFull(struct Stack * stack){
    // write your code here
}
int IsEmpty(struct Stack * stack){
    // write your code here
}
int Push(struct Stack * stack, int x){
    // write your code here
}
int Pop(struct Stack * stack){
    // write your code here
}
```

Implementing Stack with Array

Description:

Given a series of stack operations, please complete the stack ADT to output for respective operations.

Input:

The first line is a positive integer N, which denotes the capacity of the stack, followed by M lines of stack operations, each chosen from one of the followings: Push x, Pop, Isfull and Isempty, except the last of which, which is End.

Output:

A line per Pop (if stack is empty), Isfull or Isempty operation, each denoting the output of the specified operation, following by a line of remaining elements in the stack from bottom to top upon End

Sample Input 1:

```
4
Pop
Push 1
Push 2
End
```

Sample Output 1:

```
Stack is empty
1 2
```

Sample Input 2:

```
2
Isempty
Push 1
Isfull
Pop
End
```

Sample Output 2:

```
Stack is empty
Stack is not full
```

Lab 2

Problem 2

- Key points to note:
  - How to make sure the queue is sustainable (no “crawling”)?
  - Condition of fullness/emptiness of queue?

Code segment to complete:

```
struct Queue {
    int * arr;
    int capacity;
    int front;
    int rear;
};
int IsFull(struct Queue * queue){
    // write your code here
}
int IsEmpty(struct Queue * queue){
    // write your code here
}
int Enqueue(struct Queue * queue, int x){
    // write your code here
}
int Dequeue(struct Queue * queue){
    // write your code here
}
```

Implementing Queue with Array

Description:

Given a series of queue operations, please complete the queue ADT to output for respective operations.

Input:

The first line is a positive integer N, which denotes the capacity of the queue, followed by M lines of queue operations, each chosen from one of the followings: Enqueue x, Dequeue, Isfull and Isempty, except the last of which, which is End.

Output:

A line per Dequeue (if queue is empty), Isfull or Isempty operation, each denoting the output of the specified operation, following by a line of remaining elements in the queue from head to tail upon End.

Sample Input 1:

```
4
Dequeue
Enqueue 1
Enqueue 2
Dequeue
End
```

Sample Output 1:

```
Queue is empty
2
```

Sample Input 2:

```
2
Isempty
Enqueue 1
Isfull
End
```

Sample Output 2:

```
Queue is empty
Queue is full
1
```

Lab 2

Problem 3

- Key points to note:
  - Condition for emptiness of stack?
  - Best practice to remove the element during pop?

Code segment to complete:

```
typedef struct Node * PtrToNode;
struct Node {
    int element;
    PtrToNode next;
    PtrToNode prev;
};
typedef struct ListRecord * List;
struct ListRecord {
    PtrToNode head;
    PtrToNode tail;
};
void Insertion(int x, List L){
    // write your code here
}
int Deletion(List L){
    // write your code here
}
```

Implementing Stack with Linked List

Description:

Given a series of stack operations, please complete the stack ADT to output for respective operations.

Input:

M lines of stack operations, each chosen from one of the followings: Push x and Pop, except the last of which, which is End.

Output

A line per Pop (if stack is empty) operation, denoting the stack is empty, following by a line of remaining elements in the stack from bottom to top upon End

Sample Input 1:

```
Pop
Push 1
Push 2
End
```

Sample Output 1:

```
Stack is empty
1 2
```

Sample Input 2:

```
Pop
Push 1
End
```

Sample Output 2:

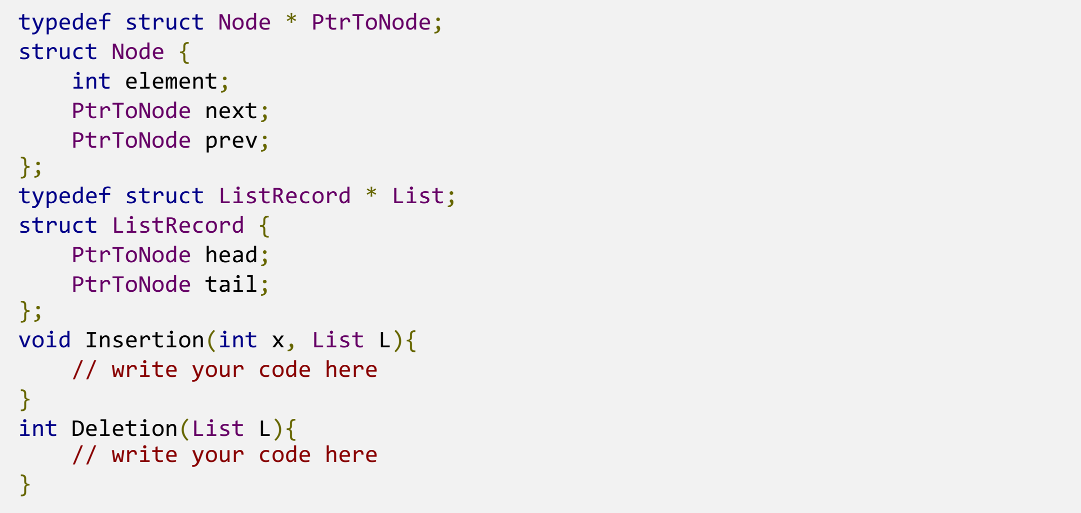
```
Stack is empty
1
```

Lab 2

Problem 4

- Key points to note:
  - Condition for emptiness of queue?
  - Best practice to remove the element during dequeue?

Code segment to complete:



```
typedef struct Node * PtrToNode;
struct Node {
    int element;
    PtrToNode next;
    PtrToNode prev;
};
typedef struct ListRecord * List;
struct ListRecord {
    PtrToNode head;
    PtrToNode tail;
};
void Insertion(int x, List L){
    // write your code here
}
int Deletion(List L){
    // write your code here
}
```

Implementing Queue with Linked List

Description:

Given a series of queue operations, please complete the queue ADT to output for respective operations.

Input:

M lines of queue operations, each chosen from one of the followings: Enqueue x and Dequeue, except the last of which, which is End.

Output:

A line per Dequeue (if queue is empty) operation, denoting the queue is empty, following by a line of remaining elements in the queue from head to tail upon End

Sample Input 1:

```
Dequeue
Enqueue 1
Enqueue 2
Dequeue
End
```

Sample Output 1:

```
Queue is empty
2
```

Sample Input 2:

```
Dequeue
Enqueue 1
Dequeue
End
```

Sample Output 2:

```
Queue is empty
```

Last but not Least

- Please add this declaration on top of (commented as shown) all your codes submitted to the OJ.

```
/*
I, <Your Full Name>, am submitting the assignment for
an individual project.
I declare that the assignment here submitted is original except for
source material explicitly acknowledged, the piece of work, or a
part
of the piece of work has not been submitted for more than one
purpose
(i.e. to satisfy the requirements in two different courses) without
declaration. I also acknowledge that I am aware of University
policy
and regulations on honesty in academic work, and of the
disciplinary
guidelines and procedures applicable to breaches of such policy and
regulations, as contained in the University website
http://www.cuhk.edu.hk/policy/academichonesty/.
It is also understood that assignments without a properly signed
declaration by the student concerned will not be graded by the
teacher(s).
*/
```

Reminders

- Please remember your password for the online judge
- Please start your assignment early
  - And report any issues as early as possible. Some issues regarding the registration of the OJ were reported not until some hours before the deadline. For such cases of failure of code submission, we may not grade your lab.
  - Penalty:
    - -10 marks/day pro rata for first two days after deadline
    - -10 marks/hour pro rata afterwards (so you get 0 marks if you submit 2 day 8 hours after the deadline)
- Grading is based on the **last** submission
- Write your own code
  - We will check your code
  - Suspected cases of plagiarism will be reported
- Questions?

Questions?