Queues and Singly Linked Lists

Subhabrata Samajder

IIIT, Delhi

Summer Semester,

23rd May, 2022

Queue

Queue

Queue

Queue

Queue

Queue

Queue ADT

- Queues store arbitrary objects.
- **Insertions:** At the end of the queue.
- **Removals:** From the front of the queue.
- The queue has a **head** and a **tail**.

Queue ADT (Cont.)

Main Operations:

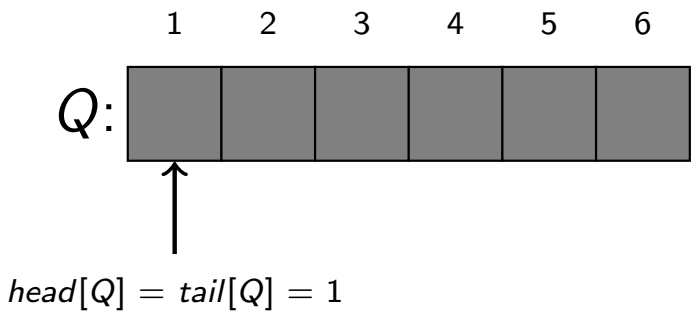
- ENQUEUE(Q, x): Inserts an element at the end.
- DEQUEUE(Q): Removes and returns the element at the front.

Queue ADT (Cont.)

Auxiliary Operations:

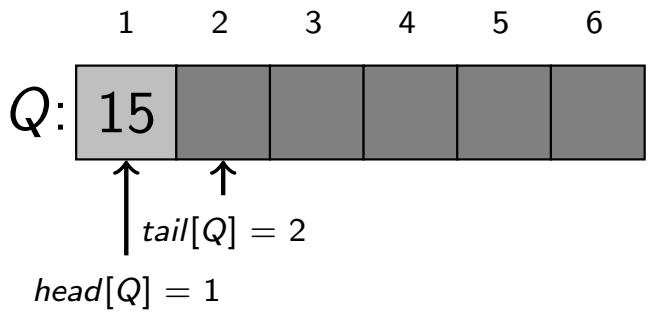
- Front(): Returns the element at the front without removing it.
- Size(): Returns the number of elements stored.
- IsEmpty(): Returns a boolean value indicating if the queue is empty or not.

An Integer Array Implementation



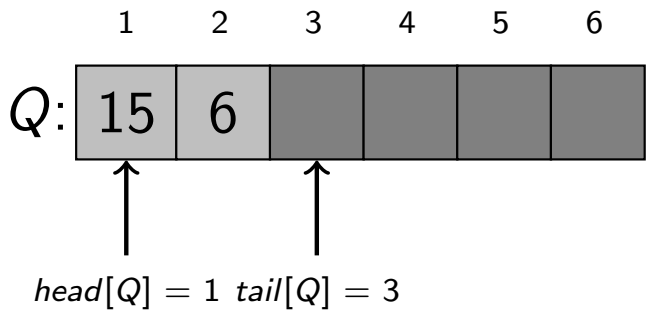
An Integer Array Implementation

ENQUEUE$(Q, 15)$:



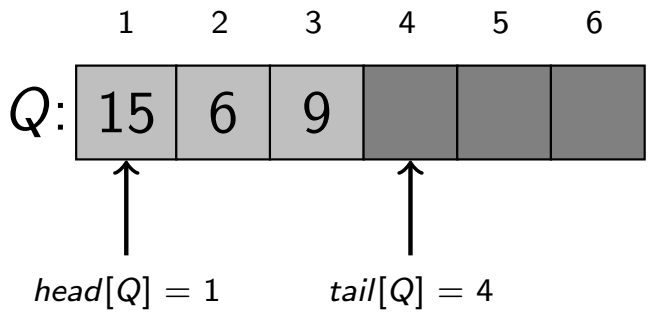
An Integer Array Implementation

ENQUEUE$(Q, 6)$:



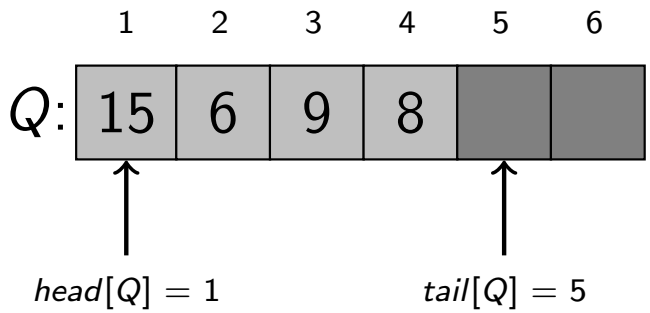
An Integer Array Implementation

ENQUEUE$(Q, 9)$:



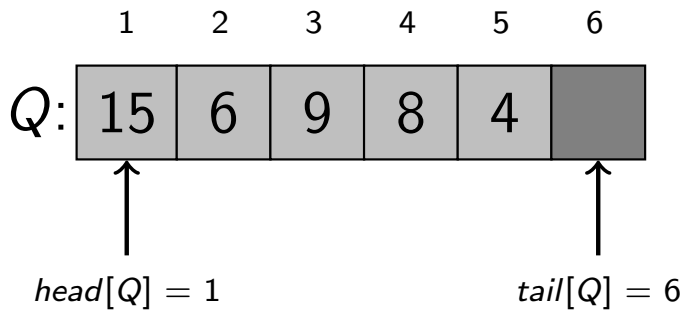
An Integer Array Implementation

ENQUEUE$(Q, 8)$:



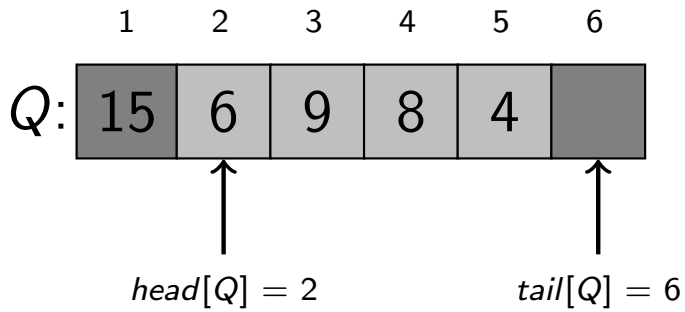
An Integer Array Implementation

ENQUEUE$(Q, 4)$:



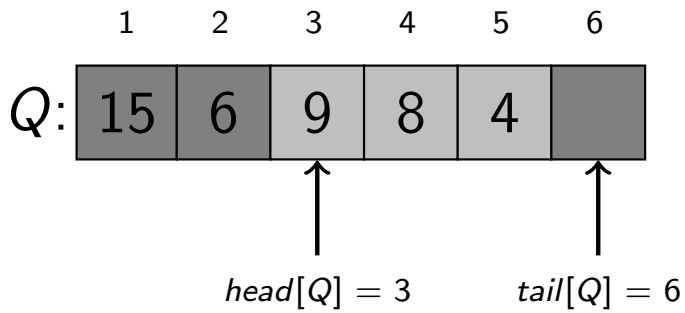
An Integer Array Implementation

DEQUEUE(Q):



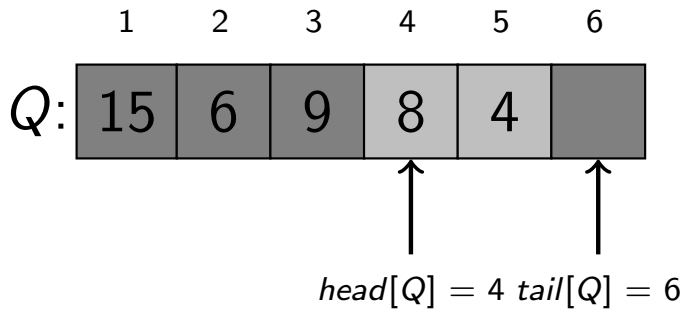
An Integer Array Implementation

DEQUEUE(Q):



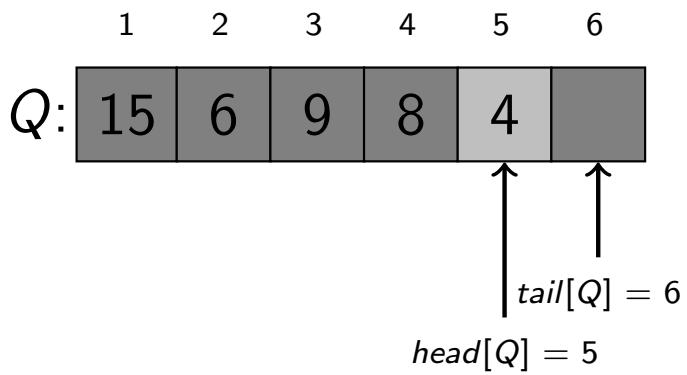
An Integer Array Implementation

DEQUEUE(Q):



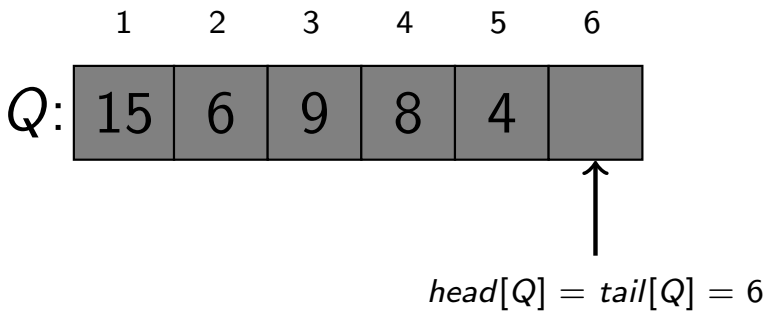
An Integer Array Implementation

DEQUEUE(Q):



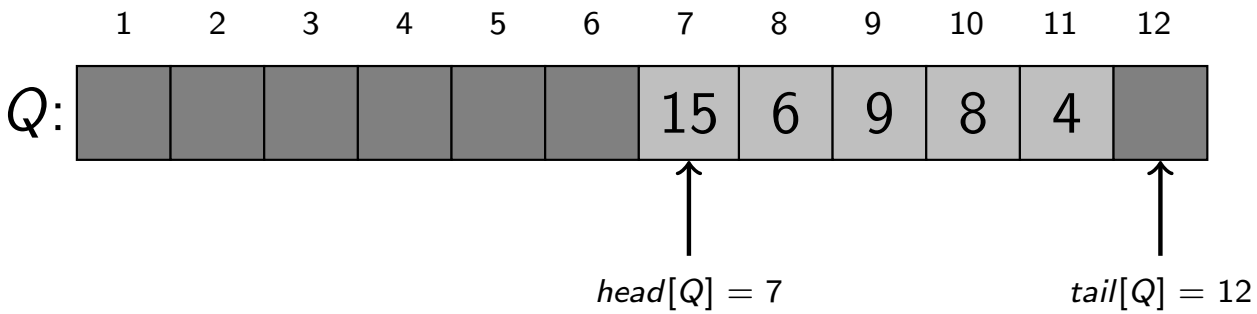
An Integer Array Implementation

DEQUEUE(Q):



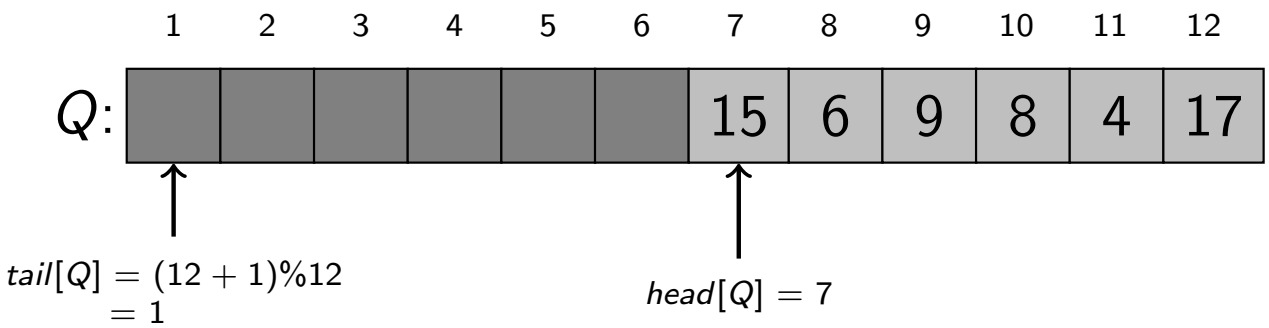
Queue Empty!!

An Improvement



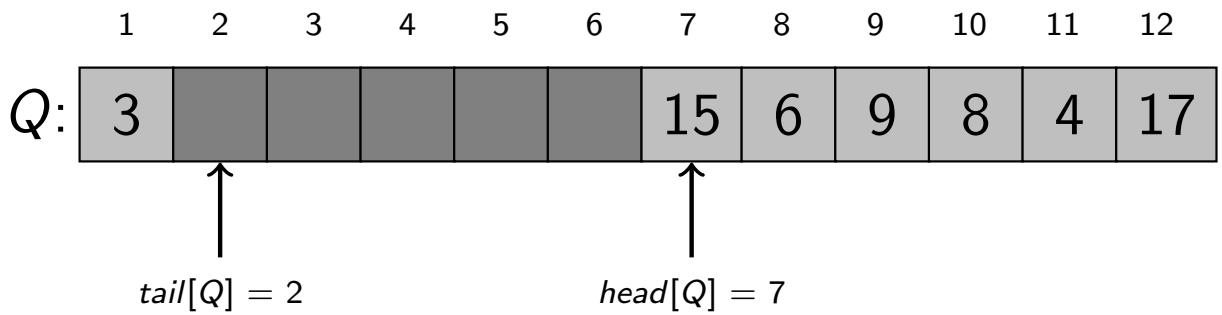
An Improvement

ENQUEUE$(Q, 17)$:



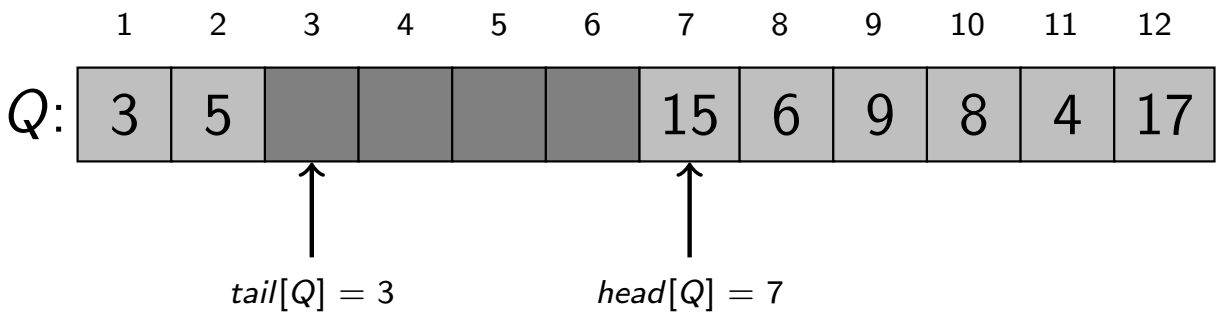
An Improvement

ENQUEUE$(Q, 3)$:



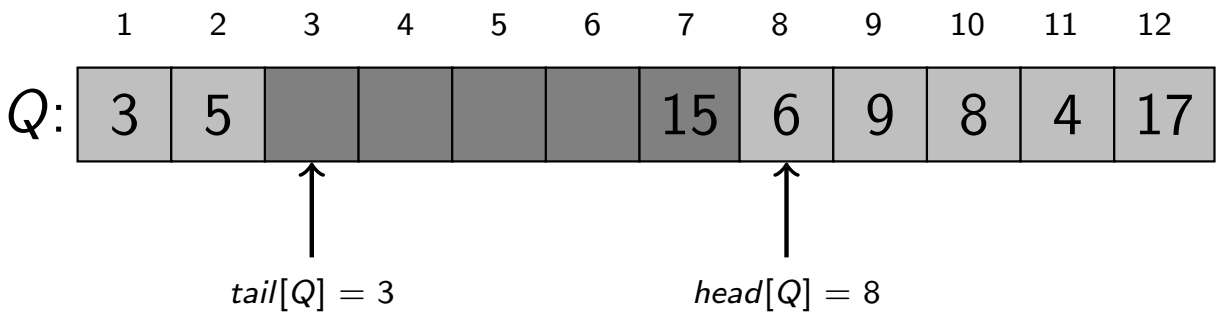
An Improvement

ENQUEUE$(Q, 5)$:



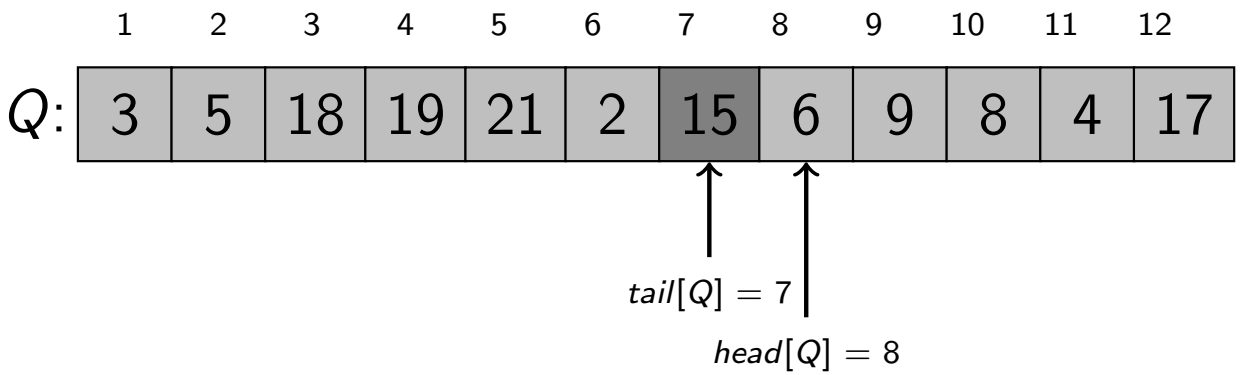
An Improvement

DEQUEUE(Q):



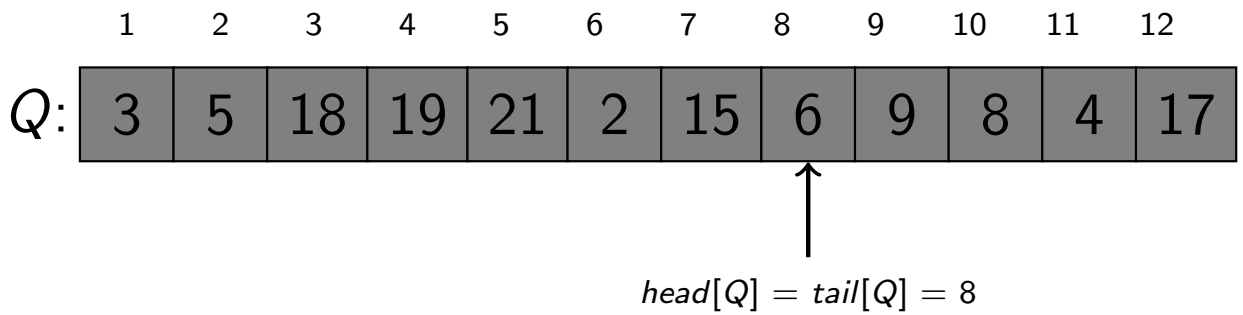
An Improvement

Queue Full: $head[Q] = (tail[Q] + 1) \mod n$.
Error: Overflow.



An Improvement

Queue Empty: $head[Q] = tail[Q]$.
Error: Underflow.



ENQUEUE(Q, x)

Begin
  If $(head[Q] = (tail[Q] + 1) \mod n)$
    return "**overflow error**"

  $Q[tail[Q]] \leftarrow x$;

  If $(tail[Q] = length[Q])$
    $tail[Q] \leftarrow 1$;
  Else
    $tail[Q] \leftarrow tail[Q] + 1$;

End

DEQUEUE(Q)

```
Begin
  If (head[Q] = tail[Q])
    return "underflow error"

  x ← Q[head[Q]];

  If (tail[Q] = length[Q])
    head[Q] ← 1;
  Else
    head[Q] ← head[Q] + 1;

  return x;
End
```

Applications of Queues

- Access to shared resources (e.g., printer).
- Simulations of read world situations of waiting lines (bank teller, flight bookings).
- To efficiently maintain a First-in-first out (FIFO) order on some entities
- In a multitasking operating system, the CPU cannot run all jobs at once, so jobs must be batched up and then scheduled according to order in a queue.
- User input in a game

A C Implemention of a Queue Using An Array

Initialization

```
/* Queue */
int main() {
   int head, tail;
   int Q[len];

   /* Initialisation */
   head = tail = 0;
   .
   .
   .
}
```

Enqueue

Insert an element at the tail of the queue Q and redefine tail:

```
/* Enqueue */
int Enqueue(int data, int *Q) {
   /* check if queue is full or not */
   if (head == (tail + 1)% length) {
      printf("\n ERROR: Queue is full\n");
      return FLAG;
   }
   /* insert element at the tail */
   else {
      Q[tail] = data;
      tail = (tail + 1)% length;
   }
   return 0;
}
```

Dequeue

Delete and return the element pointed by head of the queue:

```
/* Dequeue */
int Dequeue(int *Q) {
   int x;

   if (head == tail) {    // if queue is empty
      printf("\n ERROR: Queue is empty\n");
      return FLAG;
      }
   /* delete element from the head */
   else {
      x = Q[head];
      head = (head + 1)% length;
   }
   return x;
}
```

Front

Return the front element from the queue (if queue is not empty) but do not remove it.

```
/* prints the head of the queue */
void Front() {
   if (head == tail) {
      printf("\n Q is Empty\n");
      return FLAG;
   }

   printf("\n Front Element is: %d", Q[head]);
   return 0;
}
```

Exercise

Describe the output and final structure of the queue after the following operations:

- ENQUEUE(8)
- ENQUEUE(3)
- DEQUEUE()
- ENQUEUE(2)
- ENQUEUE(5)
- DEQUEUE()
- DEQUEUE()
- ENQUEUE(9)
- ENQUEUE(1)

Linked List

Problems With Arrays

- Many applications require that the number of elements changes dynamically as the algorithm progresses.

- **One possibility:** Define all the elements as arrays (or records) large enough to ensure sufficient storage space.
  - Often a good solution.
  - But requires storage according to the worst case (which may be unknown) which may be *inefficient*.
  - **Reason:** Array size has to be fixed in the beginning.

Problems With Arrays

- May need to perform insertions and deletions in the middle.
  - Need to shift all other elements.
  - Therefore very costly for large arrays - $\mathcal{O}(n)$.
- This problem is inherent to consecutive representation of arrays (or records).
- For these cases we need **dynamic data structures**.

Handling Lists

- Consider a list of integers

$$\{16, 8, 10, 2, 34, 20, 12, 32, 18, 9, 3\}$$

- It can be thought of as an element 16 followed by another list

$$16 - -\{8, 10, 2, 34, 20, 12, 32, 18, 9, 3\}$$

- Next list can also be thought of as 8 followed by a list

$$16 - -8 - -\{10, 2, 34, 20, 12, 32, 18, 9, 3\}$$

- $\therefore$ any list can be thought of as an element followed by a list.
- This gives a recursive definition of a linked list.

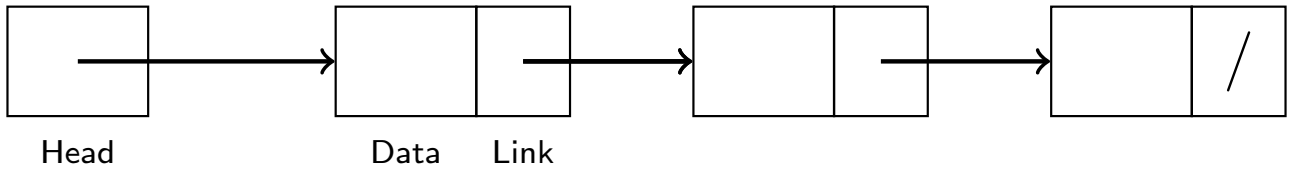
Linked Lists

- Linked lists are the simplest form of dynamic data structures.
- The objects are arranged in a linear order.
- Provides a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations of a dynamic set.

List stored in an array $A[\,]$

Memory Address	Array Index	List Contents
3200	A[0]	36
3204	A[1]	42
3208	A[2]	20
3212	A[3]	16
3216	A[4]	38
3220	A[5]	40
3224	A[6]	12
3228	A[7]	54
3232	A[8]	82

Linked List



- A linked list is a series of connected nodes.
- Each node contains at least
  - A piece of data (any type)
  - Link to the next node in the list
- Head: points to the first node
- Links are generated by system.
- The last node points to **nil**.

List Items Stored In A Linked List

Memory Address	Data Contents	Link Contents
2020	36	450
450	42	3600
3600	20	4200
4200	16	4231
4231	38	760
760	40	5555
5555	12	**nil**

Defining a Node in C

```
typedef struct Node {
   int nData;
   struct Node *pNext;
} Node;

int main() {
   Node Node1, *pNode2;
   .
   .
   .
```

Books Consulted

1. Chapter 10.1 & 10.2 of *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Thank You for your kind attention!

Questions!!