Lecture - 8
On
Queues

QUEUES

- A Queue is a linear list of elements in which deletions can take place only at one end, called the *front*, and insertions can take place only at the other end, called the *rear*. The terms “front” and “rear” are used in describing a linear list only when it implemented as a queue.
- Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are Last-in First-out (LIFO) lists.

Queues abound in everyday life.
For example:

- The automobiles waiting to pass through an intersection form a queue, in which the first car in line is the first car through;
- the people waiting in line at a bank form a queue, where the first person in line is the first person to be waited on; and so on.
- An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed.
- Queue for printing purposes

Why Queue?

There are many practical situations which involve Queue

- The queue of people waiting to be served by an ATM machine
- Collection of documents sent to a shared printer

Basic operations that involved in Queue:

- Create queue, *Create* (Q)
- Identify either queue is empty
- Add new item in queue
- Delete item from queue
- Call first item in queue

Example :

Figure 6-15 (a) is a schematic diagram of a queue with 4 elements, where AAA is the front element and DDD is the rear element. Observe that the front and rear elements of the queue are also, respectively, the first and last elements of the list. Suppose an element is deleted from the queue. Then it must be AAA. This yields the queue in figure 6-15(b), where BBB is now the front element. Next, suppose EEE is added to the queue and then FFF is now the rear element. Now suppose another element is deleted from the queue; then it must be BBB, to yield the queue in fig.6-15(d). And so on. Observe that in such a data structure, EEE will be deleted before FFF because it has been placed in the queue before FFF. However, EEE will have to wait until CCC and DDD are deleted.

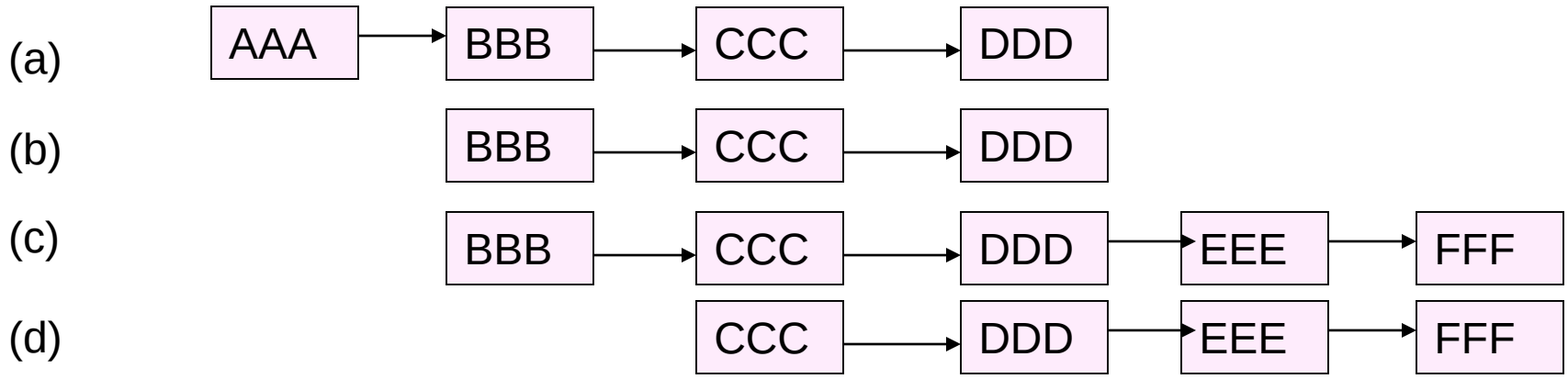


Fig. 6-15

REPRESENTATION OF QUEUE

- Queue may be represented in the computer in various ways, usually by means of one-way lists or linear arrays.
- Each of our queues will be maintained by a linear array QUEUE and two pointer variables:
- FRONT, containing the location of the front element of the queue;
- and REAR, containing the location of the rear element of the queue.
- The condition FRONT = NULL will indicate that the queue is empty.

REPRESENTATION OF QUEUE

- when an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment
- FRONT := FRONT + 1
- Similarly, whenever an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment
- REAR := REAR + 1
- This means that after N insertions, the rear element of the queue will occupy QUEUE[N] or, in other words, eventually the queue will occupy the last part of the array. This occurs even through the queue itself may not contain many elements.

REPRESENTATION OF QUEUE

- Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the past part of the array,
- i.e. when REAR = N. One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above.
- Array QUEUE is circular, that is, that QUEUE[1] comes after QUEUE[N] in the array. With this assumption, we insert ITEM into the queue by assigning ITEM to QUEUE[1]. Specifically, instead of increasing REAR to N+1, we reset REAR=1 and then assign

  QUEUE[REAR] := ITEM
- Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N + 1.
- Suppose that our queue contains only one element, i.e., suppose that
- FRONT = REAR = NULL
- And suppose that the element is deleted. Then we assign
- FRONT := NULL and REAR := NULL

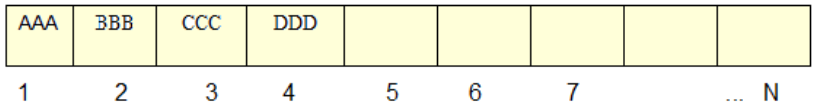
  To indicate that the queue is empty.

REPRESENTATION OF QUEUE

QUEUE

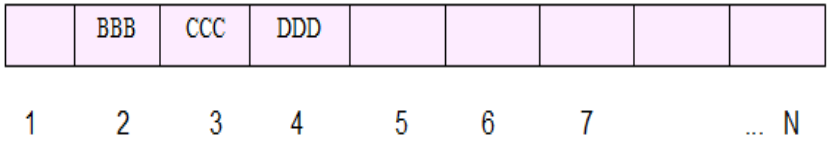
FRONT : 1

REAR : 4



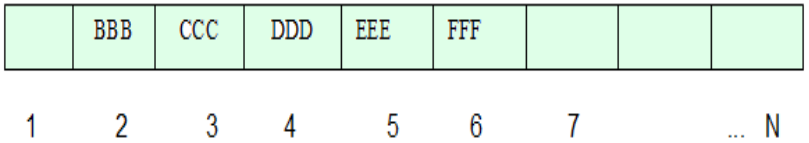
FRONT : 2

REAR : 4



FRONT : 2

REAR : 6



FRONT : 3

REAR : 6

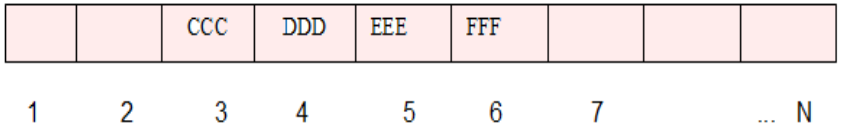
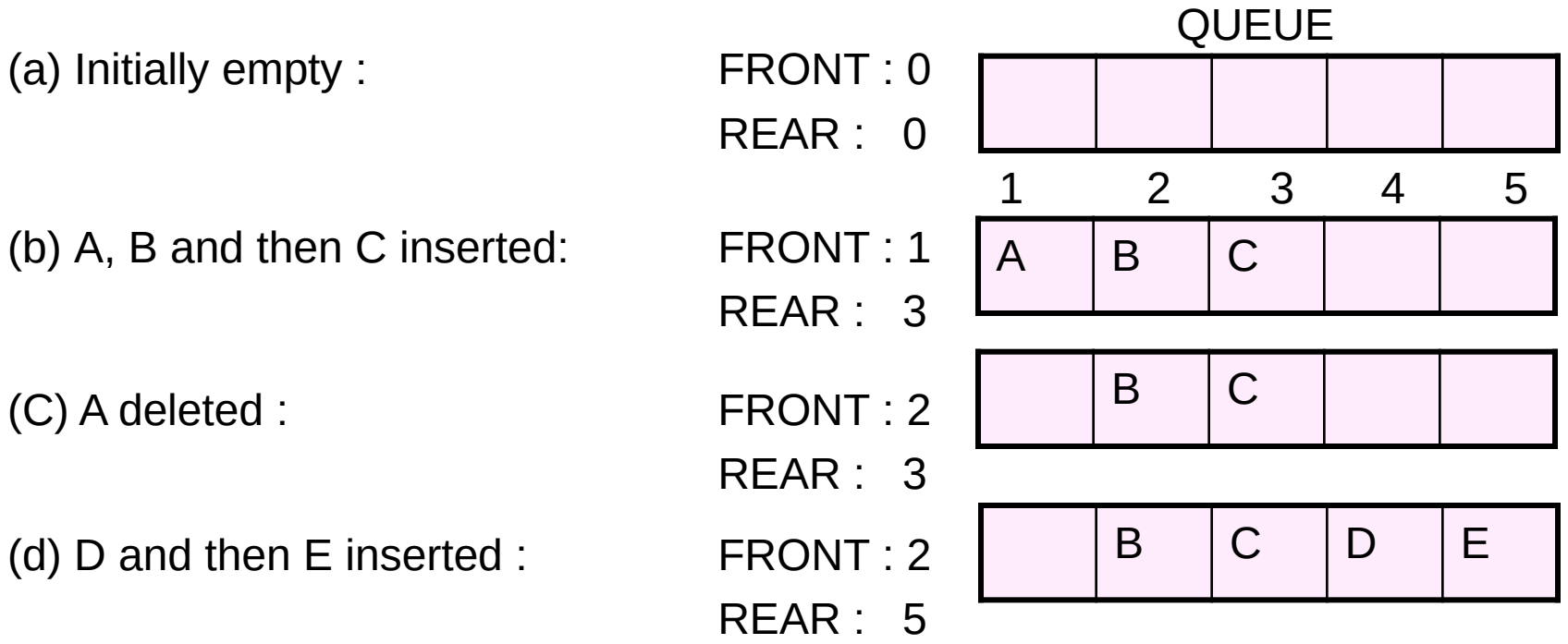


Fig : 6.16 Array representation of a queue

Example

Figure 6-17 shows how a queue may be maintained by a circular array QUEUE with N = 5 memory locations. Observe that the queue always occupies consecutive locations except when it occupies locations at the beginning and at the end of the array. If the queue is viewed as a circular array, this means that is still occupies consecutive locations. Also, as indicated by fig. 6-17(m), the queue will be empty only when FRONT = REAR and an elements is deleted. For this reason, NULL is assigned to FRONT and REAR in fig. 6-17(m).



Example

QUEUE

(e) B and C deleted :

FRONT : 4

REAR : 5

(f) F inserted :

FRONT : 4

REAR : 1

(g) D deleted :

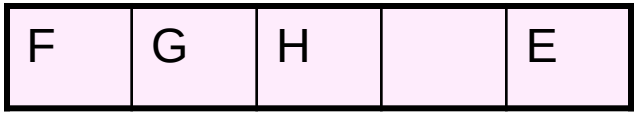
FRONT : 5

REAR : 1

(h) G and then H inserted :

FRONT : 5

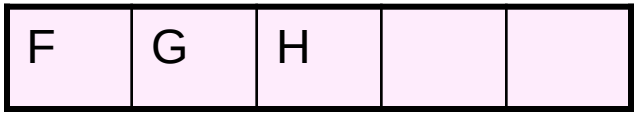
REAR : 3



(i) E deleted :

FRONT : 1

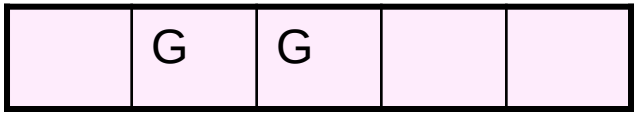
REAR : 3



(j) F deleted :

FRONT : 2

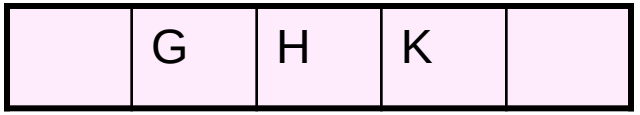
REAR : 3



(k) K inserted :

FRONT : 2

REAR : 4

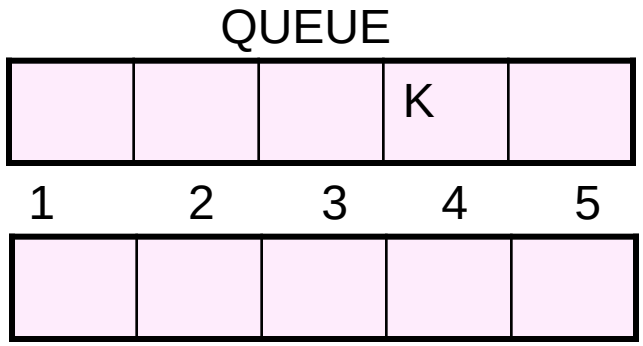


Example

(l) G and H deleted :

FRONT : 4

REAR : 5



(m) K deleted, QUEUE is empty :

FRONT : 4

REAR : 1

Fig :6.17

QINSERT

QINSERT(QUEUE,N,FRONT,REAR,ITEM)

This procedure inserts an element ITEM into a queue.

[Queue already fill]

1. If FRONT=1 and REAR=N, or if FRONT=REAR+1, then :
   Write: Overflow, and Return.
2. [Find new value of REAR]
   If FRONT=NULL, then :[Queue initially empty]
   Set FRONT:=1 and REAR:=1
   Else if REAR =N then
   Set REAR:=1
   Else Set REAR:=REAR+1
   [End of if structure]
3. Set QUEUE[REAR]:=ITEM [This inserts new element]
4. Return.

QDELETE

ITEM.QDELETE(QUEUE,N,FRONT,REAR,ITEM)

This procedure deletes an element from the queue and assigns it to the variable

- [Queue already empty]

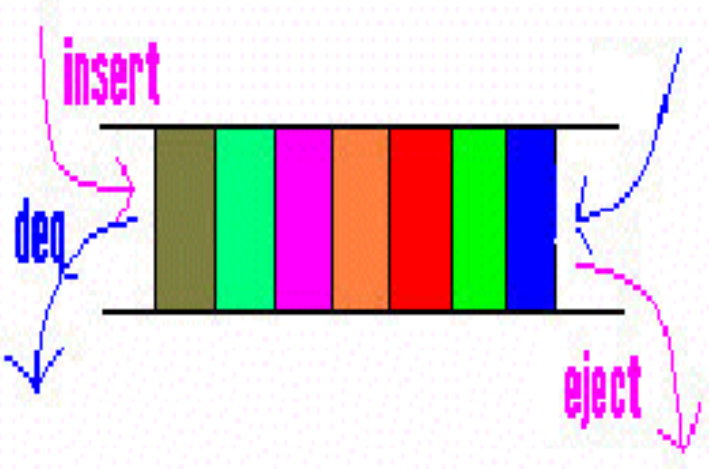
If FRONT=NULL, then Write: Underflow, and Return.

2. Set ITEM=QUEUE[FRONT]
3. [Find new value of FRONT]
   If FRONT=REAR, then [Queue has only one element to start]
   Set FRONT=NULL and REAR=NULL
   Else if FRONT =N then
   Set FRONT =1
   Else Set FRONT = FRONT +1
   [End of if structure]
4. Return

Deque

- The mathematical model of a **Deque** (usually pronounced like *"deck"*) is an irregular acronym (Operation) of **d**ouble-**e**nded **que**ue.
- Double-ended queues are a kind of sequence containers.
- Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

•The model allows data to be entered and withdrawn from the front and rear of the data structure.



Deques

- Insertions *and* deletions can occur at *either* end but not in the middle
- Implementation is similar to that for queues
- Deques are not heavily used
- You should know what a deque is, but we won't explore them much further

Double-Ended-QUE

- There are two variations of deque

–Input-restricted deque

An input restricted deque is a deque which allows insertion at only one end of the list but allows deletion a both end of the list.

–Output-restricted deque

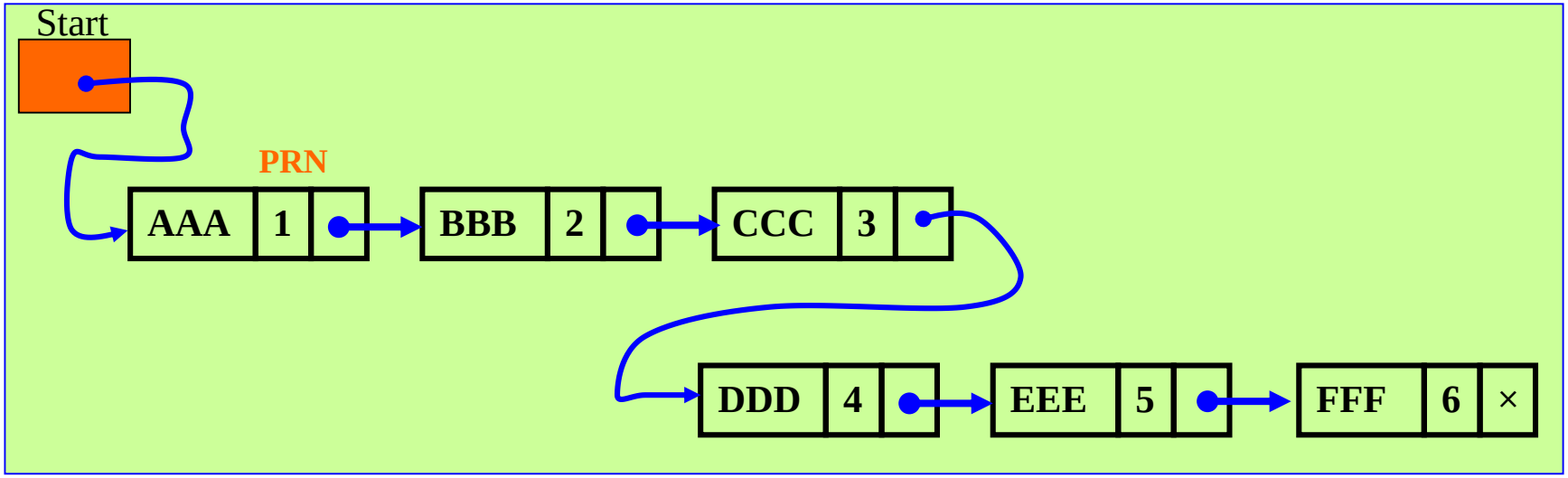
An output restricted deque is a deque which allows deletion at only one end of the list but allows insertin a both end of the list

Priority Queue

- A priority queue is a collection of elements such that each element has been assigned a priority, such that the order in which the elements are deleted and processed comes from the following rules:
  - An element with higher priority will be processed before any element with lower priority.
  - Two elements with the same priority will be processed in order in which they are add to the queue.

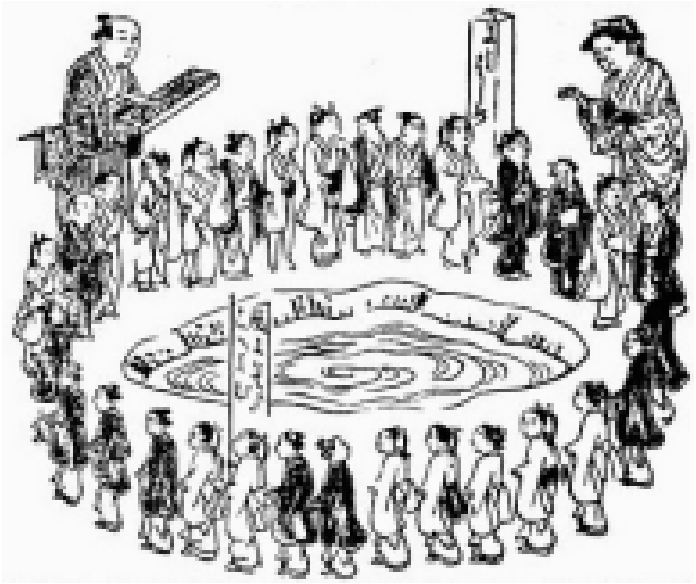
One way list representation

- Each node contains three items of information
- A node X proceeds with a node Y in the list when
  - X has higher priority than Y **or**
  - Both has same priority but X was added in the list before Y.



- Property:
  - First node will processed first

Josephus problem



- *Given*: *n* people numbered 1 to *n* around a circule
- *Rules*:
  - Eliminate every *second* remaining person until only one survives.
- *Problem*: Determine the survivor's number.

Josephus problem

Strategy

- Introduce appropriate notation:
  - $J(n)$ denotes the survivor's assigned number.
- Look at small cases first:

n	$J(n)$
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5

Josephus problem

- We may start with:
  - An even number $2n$ of people
  - An odd number $2n + 1$ of people
- *First round* eliminates all people assigned with *even* numbers.
- $2n$ is the last person eliminated in the first round.

Josephus problem

A General Solution Pattern

Consider the cases of starting number of people separately:

- An even number $2n$ of people
  - After 1st round, n people remain: $1, 3, 5, \ldots, 2n-3, 2n-1$
  - How would the survivor of these n people relate to $J(n)$?
  - $J(2n) = 2J(n) - 1$
- An odd number $2n+1$ of people
  - After 1st round, $n+1$ people remain: $1, 3, 5, \ldots, 2n-3, 2n-1, 2n+1$
  - In order to relate to $J(n)$, we need to eliminate one more person.
  - After the first elimination in 2nd round, n people remain:

    $3, 5, \ldots, 2n-3, 2n-1, 2n+1$
  - How would the survivor of these n people relate to $J(n)$?
  - $J(2n+1) = 2J(n) + 1$

Recurrence Relation for $J(n)$

We end up with the following recurrence relation, split into the cases of the starting number of people being *even* or *odd* :

$$\begin{aligned} J(1) &= 1 \\ J(2n) &= 2J(n) - 1 \quad \text{for } n > 0 \\ J(2n+1) &= 2J(n) + 1 \quad \text{for } n > 0 \end{aligned}$$

That is, the solution to the *original problem* can be constructed from the solution to *a half-size subproblem*.

How many times of unfolding are required for calculating $J(100)$?

6 times: $J(50)$, $J(25)$, $J(12)$, $J(6)$, $J(3)$, $J(1)$

However, the above relation only gives us *indirect* information. Instead, we need a *closed-form solution* to the above recurrence relation.

Guessing Closed Form Solution to $J(n)$

n	$J(n)$
1	1
2	1
3	3
4	1
5	3
6	5
7	7
8	1
9	3
10	5
11	7
12	9
13	11
14	13
15	15
16	1

Pattern of the Closed Form Solution to $J(n)$

n	$J(n)$
~~1~~	~~1~~
~~2~~	~~1~~
3	3
~~4~~	~~1~~
5	3
6	5
7	7
~~8~~	~~1~~
9	3
10	5
11	7
12	9
13	11
14	13
15	15

A Hypothesized Closed Form Solution to $J(n)$

It seems that we can split n into ($log_2 n + 1$) groups, such that:

- A a group starting with the number 2^g contains 2^g numbers.
- Each starts with a *power of 2* (i.e., $2^0, 2^1, 2^2, 2^3, \ldots$).
- $J(n)$ in the *beginning* of the group is always 1.
- $J(n)$ increases by 2 as we move towards the *end* of the group.

How would you formulate this?

Given any number in the interval of $[1, n]$, we can locate it by:

- identifying *which group* it belongs to;
- measuring *how far* it is from the beginning of the group.

For a given number that belongs to the group starting with 2^g and is h away from the beginning of its belonging group, we have:

$$J(2^g + h) = 2h + 1 \qquad \text{for } g \geq 0 \text{ and } 0 \leq h < 2^g$$

Prove by Mathematical Induction on

Basis:
$J(1) = J(2^0 + 0) = 2 \times 0 + 1 = 1$

Inductive Step Part I: $g > 0$, $0 \leq h < 2^g$, h is <u>even</u>

- **Assume**: the closed-form formula holds for the group starting with the number 2^{g-1}, i.e., interval $[2^{g-1}, 2^g)$.

  - h is <u>even</u> implies that $2^{g-1} + \frac{h}{2} < 2^g$ (why?!)

- then

$$
\begin{aligned}
& J(2^g + h) \\
= \; & \{\texttt{h is even, Recurrence relation for } J(2n)\} \\
& 2J(2^{g-1} + \tfrac{h}{2}) - 1 \\
= \; & \{2^{g-1} + \tfrac{h}{2} < 2^g, \texttt{ Inductive assumption }\} \\
& 2(2 \times \tfrac{h}{2} + 1) - \\
= \; & \{\texttt{Arithmetic}\} \\
& 2h + 1
\end{aligned}
$$

Prove by Mathematical Induction on

Inductive Step Part II: $g > 0, 0 \leq h < 2^g$, h is odd

- **Assume**: the closed-form formula holds for the group starting with the number 2^{g-1}, i.e., interval $[2^{g-1}, 2^g)$.
  - h is odd implies that $2^{g-1} + \frac{h-1}{2} < 2^g$ (why?!)
- then

$$
\begin{aligned}
& J(2^g + h) \\
= \; & \{\texttt{h is even, Recurrence relation for } J(2n+1)\} \\
& 2J(2^{g-1} + \tfrac{h-1}{2}) + 1 \\
= \; & \{2^{g-1} + \tfrac{h-1}{2} < 2^g, \texttt{ Inductive assumption}\} \\
& 2(2 \times \tfrac{h-1}{2} + 1) + 1 \\
= \; & \{\texttt{Arithmetic}\} \\
& 2h + 1
\end{aligned}
$$

Palindrome checker using stack and queue

- The string that reads the same backward as well as forward is called as palindrome string.

Given string **str**, the task is to find whether the given string is a palindrome or not using a stack.

Examples:

Input: *str = "geeksforgeeks"*
Output: *No*
Input: *str = "madam"*
Output: *Yes*

Palindrome checker using stack and queue

- Algorithm -

  - Find the length of the string say **len**. Now, find the mid as **mid = len / 2**.
  - Push all the elements till mid into the stack i.e. **str[0...mid-1]**.
  - If the length of the string is odd then neglect the middle character.
  - Till the end of the string, keep popping elements from the stack and compare them with the current character i.e. **string[i]**.
  - If there is a mismatch then the string is not a palindrome. If all the elements match then the string is a palindrome.

Introduction to Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
- Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Ignored/Preorder/Postorder Tree Traversals, DFS of Graph, etc.
- A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.

Recursion

Algorithm: Steps

The algorithmic steps for implementing recursion in a function are as follows:

Step1 - Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

Step2 - Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

Step3 - Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

step4 - Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

Recursion

- **A Mathematical Interpretation**

approach(1) – Simply adding one by one

$f(n) = 1 + 2 + 3 + \ldots\ldots + n$

but there is another mathematical approach of representing this,

approach(2) – Recursive adding

$f(n) = 1 \qquad n=1$

$f(n) = n + f(n-1) \qquad n>1$