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.*
- ✓ Queues are also called first-in first-out (FIFO) list
- ✓ This contrasts with stacks, which are LIFO

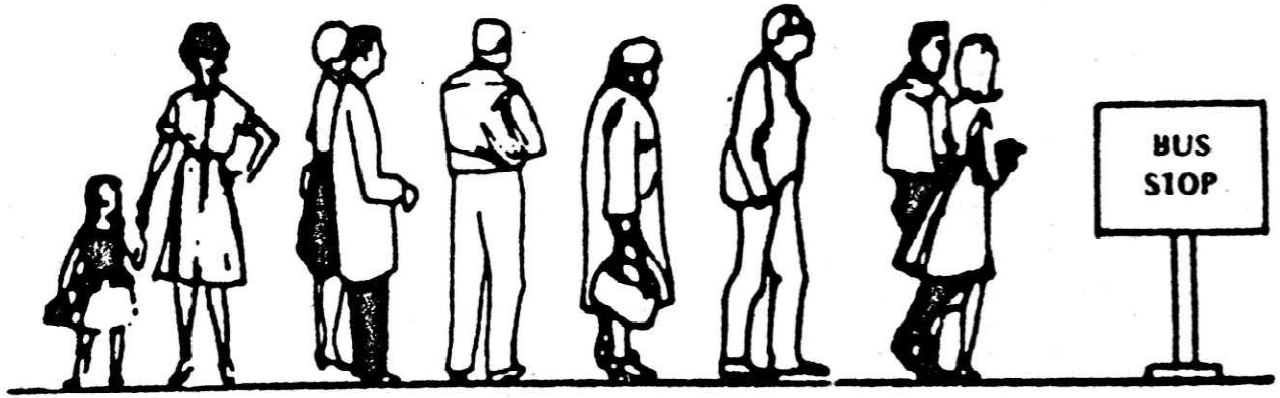
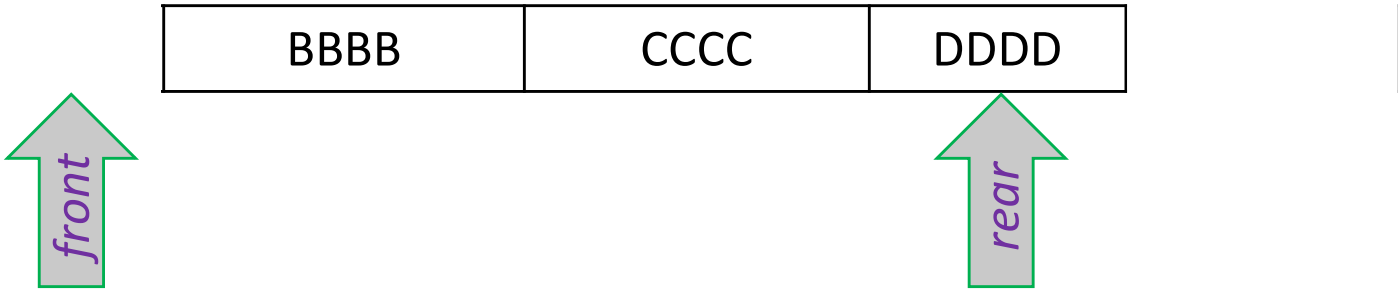


Fig. 6-2 Queue waiting for a bus.

- ✓ 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.

Queues: Example

Consider a Queue with 4 elements



✓**Suppose an element is deleted from the 4 element queue.**

Which is we can delete??

AAAA

Then which one is considered to be the front element?

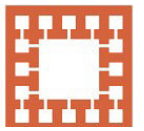
BBBB

Suppose EEEE is added to the queue

Then which one is considered to be the rear element?

EEEE

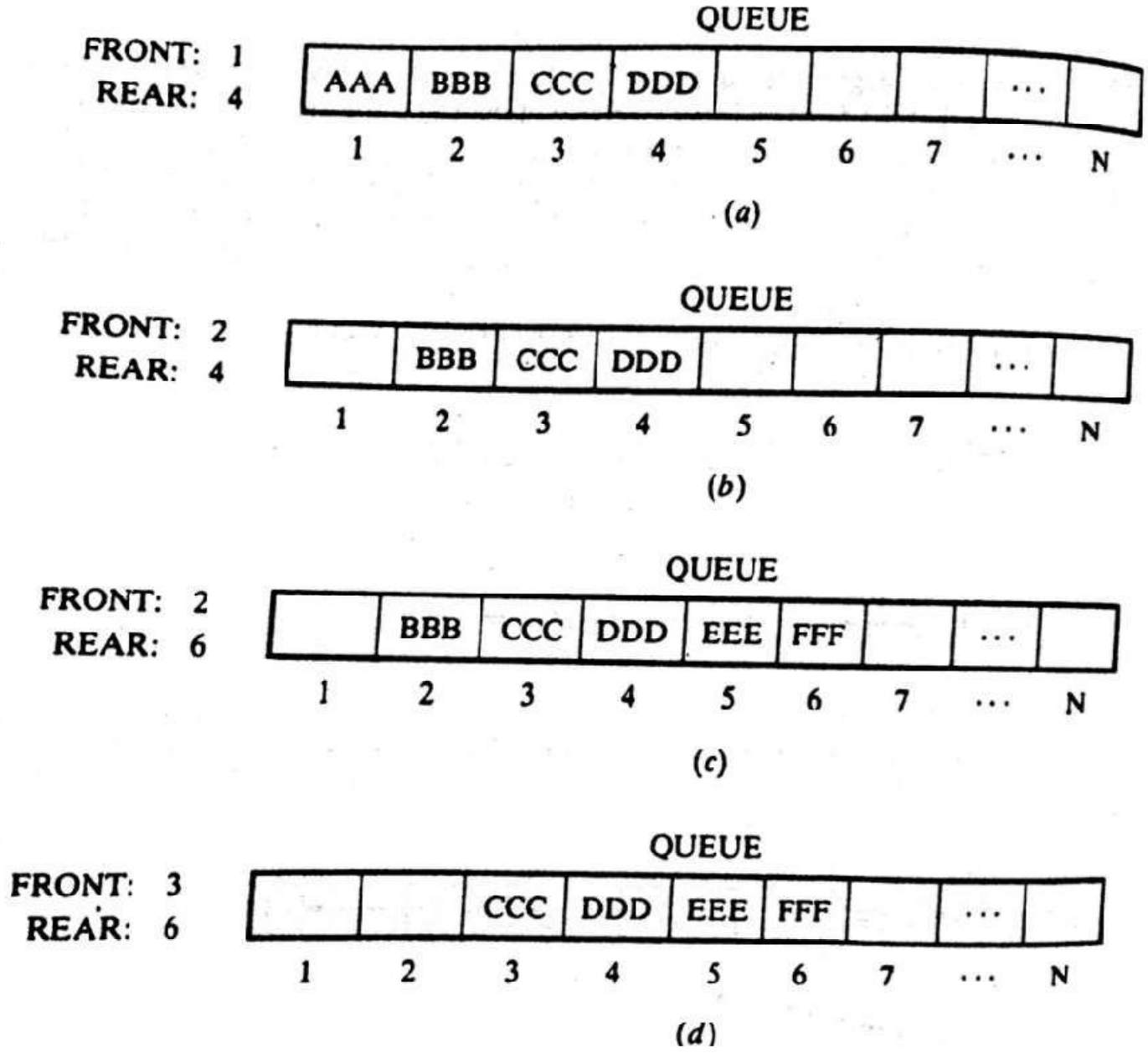
Queues: Representation



- ✓ Queues may be represented in the computer in various ways.
- ✓ Usually by means of
  - ➢ One-way list
  - ➢ Linear arrays
- ✓ Each of the queues will be maintained by …
  - ➢ A Linear Array **QUEUE**
  - ➢ Two pointer variables:
    - ○ **FRONT** (containing the location of the front element of the Queue)
    - ○ **REAR** (containing the location of the rear element of the Queue)
- ✓ The condition **FRONT= NULL** will indicate that the Queue is……

EMPTY.

Queues: Representation



Queues: Representation

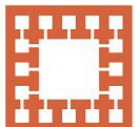
- ✓ After N insertion, the REAR element of the QUEUE will occupy QUEUE [N]
- ✓ This occurs even though the queue itself may not contain many elements

- ✓ Suppose, we want to insert an element **ITEM** into a queue at the time the queue does occupy the last part of the array, that is **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.
- ✓ Then insert item as above.

- ✓ This procedure is very expensive
- ✓ The procedure we adopt is to assume that the array **QUEUE is Circular. ie, QUEUE[1] comes after QUEUE[N]**

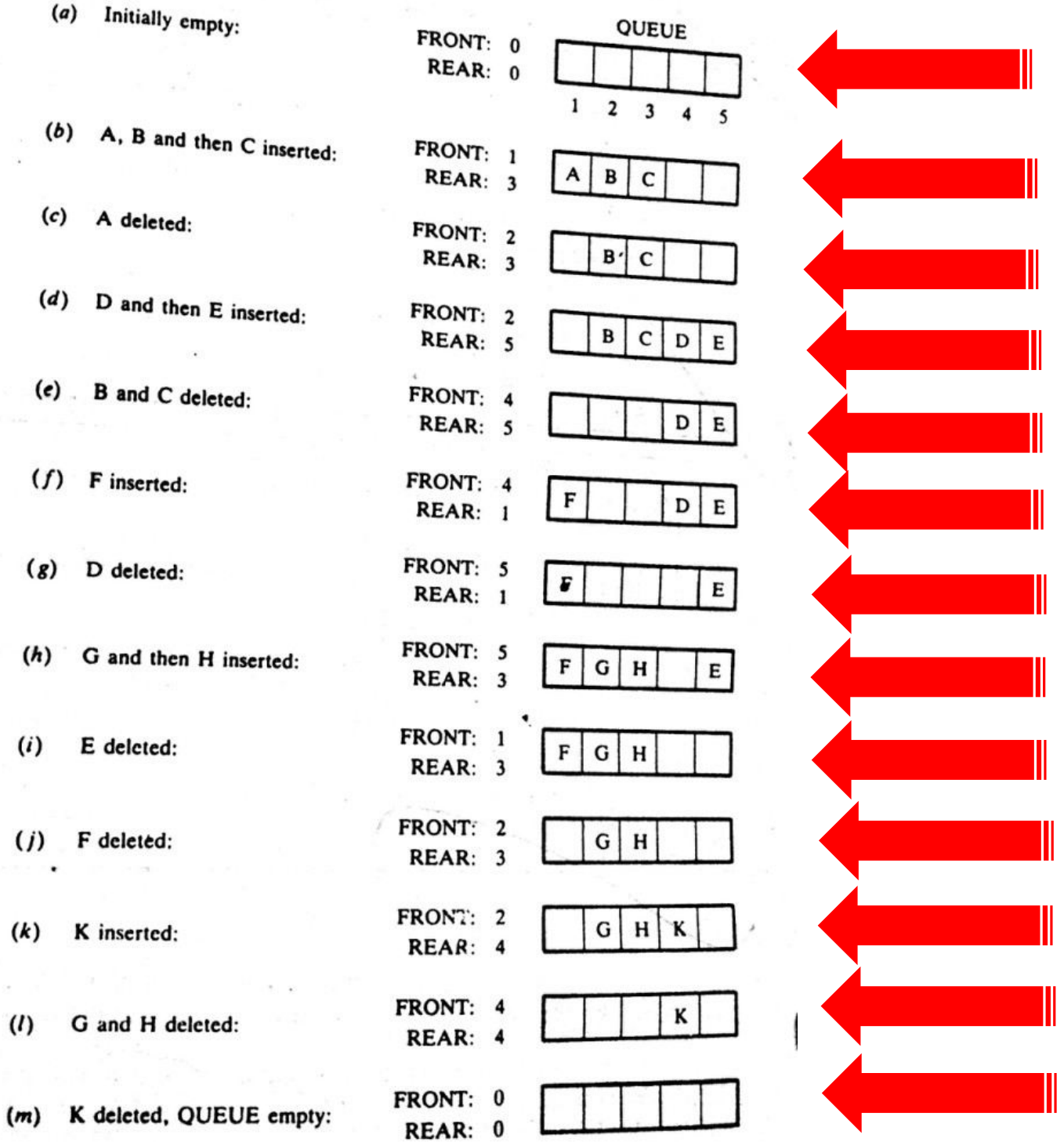
Queues: Representation



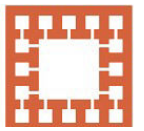
CSE 2103

- ✓ Similarly, if FRONT=N and an element of Queue is deleted.
- ✓ We reset FRONT=1, instead of increasing FRONT to N+1.

Queues: Representation



Queues: Insertion Algorithm



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

Step1. [QUEUE already filled?]

If FRONT=1 and REAR=N, or FRONT=REAR+1, then:

Write: OVERFLOW, and Return

Step2. [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,

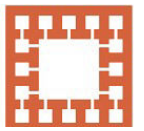
Else:

Set REAR:=REAR+1. [End of If structure]

Step3. **Set QUEUE[REAR]=ITEM** [This inserts new item]

Step4. Return

Queues: Deletion Algorithm



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

Step1. [QUEUE already empty?]

If FRONT=NULL then:

Write: UNDERFLOW, and Return

Step2. **Set** ITEM:= QUEUE[FRONT],

Set3. [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:

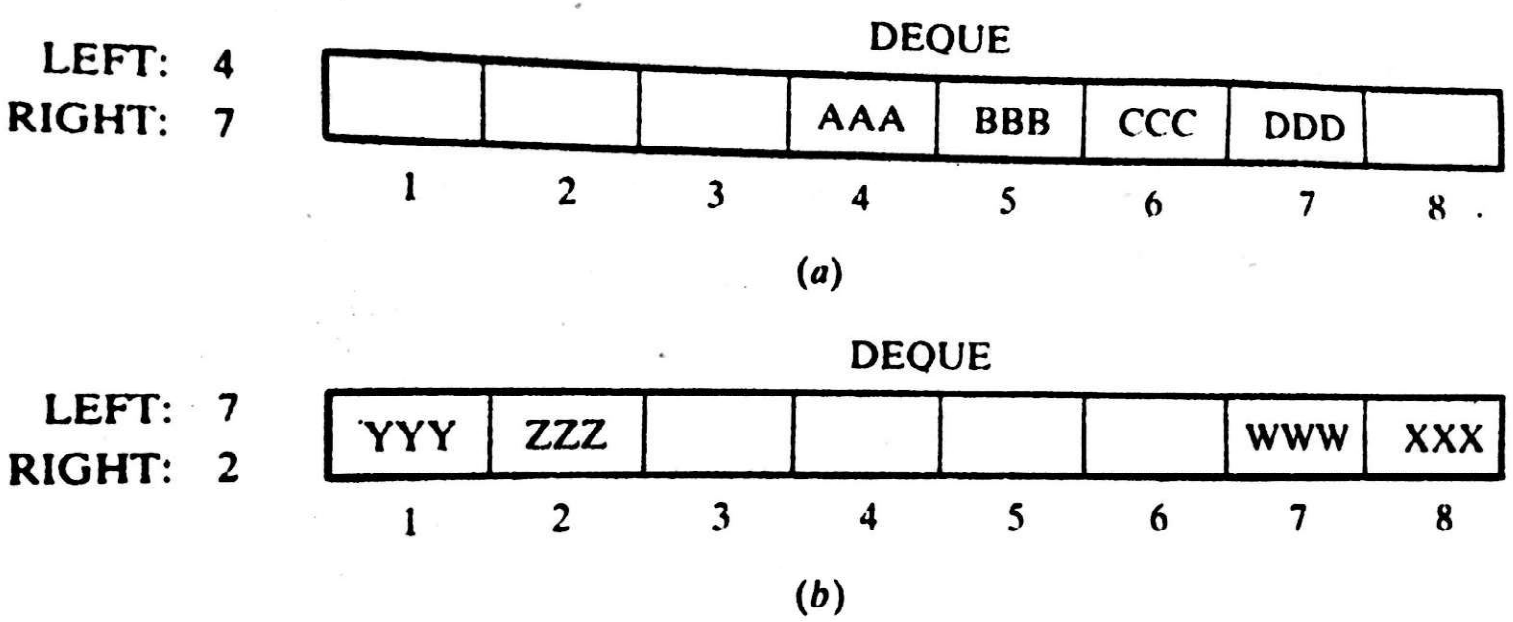
Set FRONT:=FRONT+1.

[End of If structure]

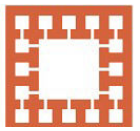
Step4. Return

Deques: Definition

- ✓ A deque is a linear list in which elements can be added or removed at either end but not in the middle.
- ✓ Deque can be maintained by……
  - ➢ a CIRCULAR array
  - ➢ Pointer LEFT-which points left end of the deque
  - ➢ Pointer RIGHT-which points right end of the deque.



Deques: Variation



- ✓ There are TWO variations of a Deque-
  - ➢ An Input-restricted deque, and
  - ➢ An Output-restricted deque
- ✓ This are intermediate between a Deque and Queue

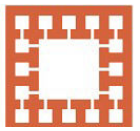
- ✓ An Input-restricted deque is a deque which allow insertion at only one end of the list but allows deletions at both ends of the list
- ✓ An output-restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

Priority Queues: Definition

A priority queue is a collection of elements such that each elements has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

- An element of higher priority is processed before any element of lower priority
- Two elements with same priority are processed according to the order in which they were added to the queue.

Priority Queues: One-way List Representation



One way to maintain a priority queue in memory is by means of a one-way list, as follows

a) Each node in the list will contain three items of information:
   - An Information field INFO,
   - A priority number PRN, and
   - A link number LINK

b) A node X precedes a node Y in the list

(1) when X has higher priority then Y

(2) When both have the same priority but X was added to the list before Y

❖ Priority numbers will operate in the usual way: the Lower the priority number, the higher the priority.

Priority Queues: Schematic Diagram with 7 elemen

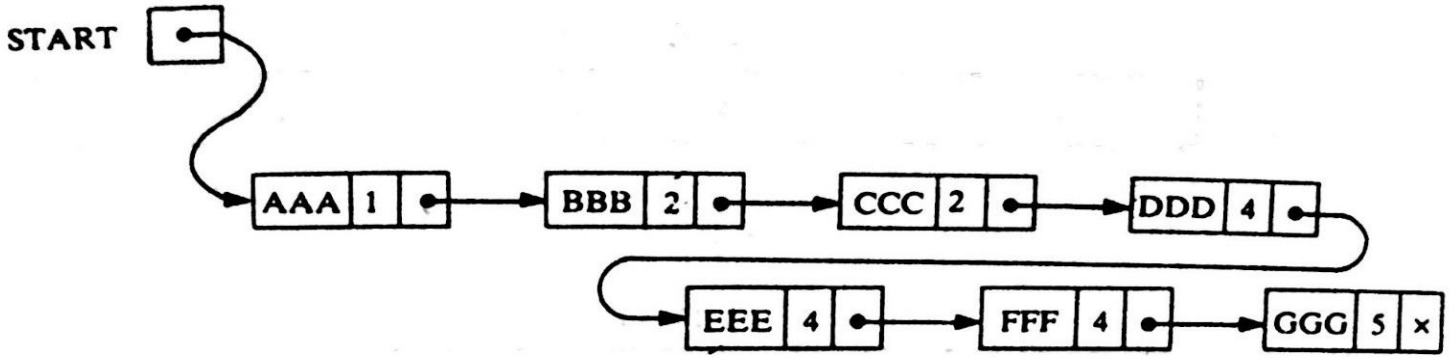


Fig. 6-19

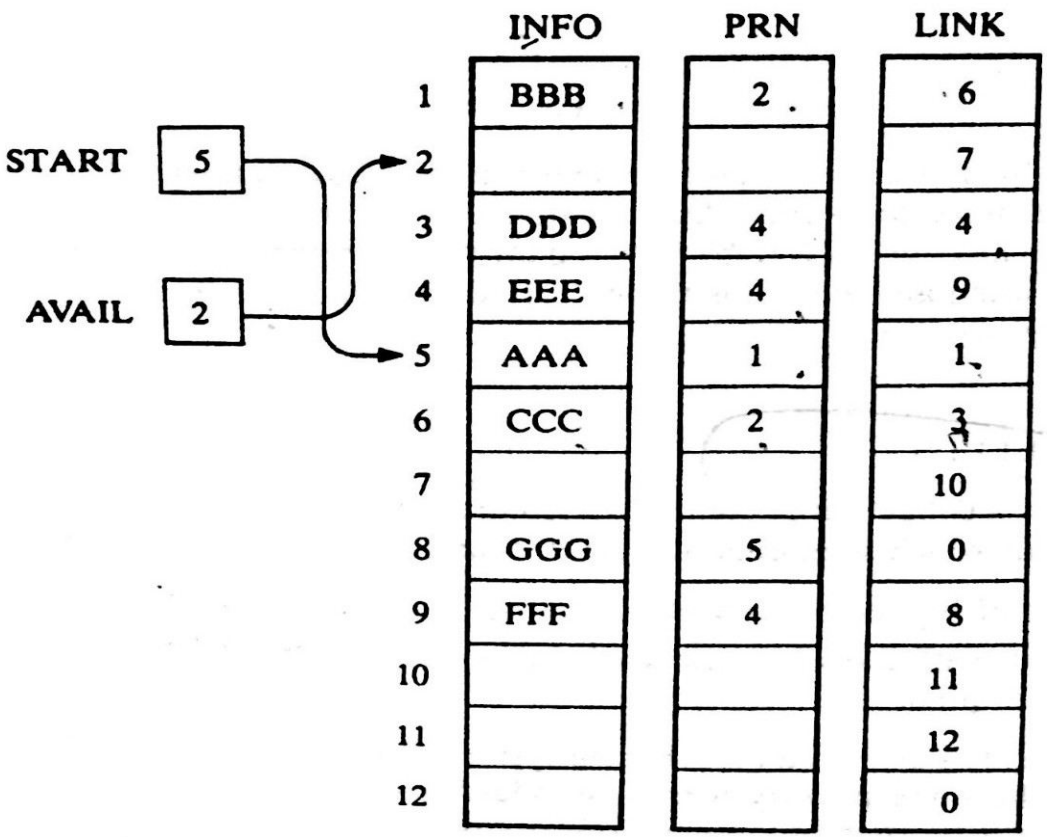
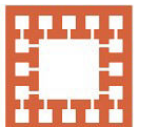


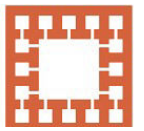
Fig. 6-20

Upcoming Presentation from Students:



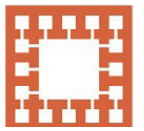
- Implement STACK using one QUEUE
- Implements STACK using two QUEUES
- Implement QUEUE using one STACK
- Implement QUEUE using two STACK

Recursion: Function



- ✓ A function is said to be *Recursively defined* if the function definition refers to itself.
- ✓ A *Recursive Function* must have the following two properties:
  - ➢ There must be certain arguments, called **BASE VALUE**, for which the function does not refer to itself.
  - ➢ Each time the function does refer to itself, the argument of the function must be closer to a **BASE VALUE**.
- ✓ A *Recursive Function* with two properties is also said to be *well-defined*.

Recursion: Factorial Function



CSE 2103

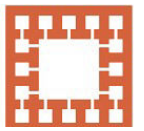
- ✓ In some problems, it may be natural to define the problem in terms of the problem itself.
- ✓ Recursion is useful for problems that can be represented by a **SIMPLER VERSION** of the same problem.
- ✓ Example: the factorial function

```
6! = 6 * 5 * 4 * 3 * 2 * 1
```

We could write:

```
6! = 6 * 5!
```

Recursion: Factorial Function



- ✓ In general, we can express the factorial function as follows:

```
n! = n * (n-1)!
```

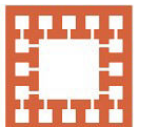
Is this correct? **Well… almost.**

- ✓ The factorial function is **ONLY DEFINED** for *positive* integers. So we should be a bit more precise:

```
i)  n! = 1                (if n is equal to 1)
ii) n! = n * (n-1)!       (if n is larger than 1)
```

- ✓ Observe that, this definition of `n!` is recursive, since it refers to itself when it uses `(n-1)!` , However,
- ✓ **i)** the value of `n!` is explicitly given when `n=0` (**BASE VALUE**)
- ✓ **ii)** the value of `n!` for arbitrary `n` is defined in terms of a smaller value of n which is closer to the **BASE VALUE**

Recursion: Factorial Function



EXAMPLE: Let's calculate `3!` Using the recursive definition.

(1) `3! = 3 . 2!`

(2) `2! = 2 . 1!`

(3) `1! = 1 . 0!`

(4) `0! = 1 (BASE VALUE)`

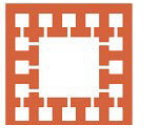
(5) `1! = 1 . 1 = 1`

(6) `2! = 2 . 1 = 2`

(7) `3! = 3 . 2 = 6`

- ✓ **Observe that we back track in the reverse order of the original postponded evaluations.**
- ✓ **Recall that this type of postponed processing tends itself to the use of STACKS.**

Recursion: Function



Assume the number typed is 3, that is, numb=3.

fac(3) :

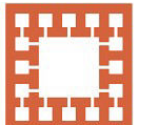
```
3 <= 1 ?                    No.
fac(3) = 3 * fac(2)
   fac(2) :
      2 <= 1 ?              No.
      fac(2) = 2 * fac(1)
         fac(1) :
            1 <= 1 ?        Yes.
            return 1
      fac(2) = 2 * 1 = 2
      return fac(2)
fac(3) = 3 * 2 = 6
return fac(3)
```

```
fac(3)  has the value 6
```

i) n! = 1 (if n is equal to 1)
ii) n! = n * (n-1)! (if n is larger than 1)

```
int fac(int numb){
  if(numb<=1)
      return 1;
   else
      return numb * fac(numb-1);
}
```

Recursion: Function



For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

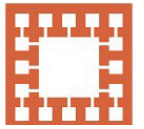
Recursive solution

```
int fac(int numb){
   if(numb<=1)
      return 1;
   else
      return numb*fac(numb-1);
}
```

Iterative solution

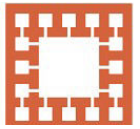
```
int fac(int numb){
int product=1;
    while(numb>1){
        product *= numb;
            numb--;
                }
        return product;
                }
```

Recursion: Overhead



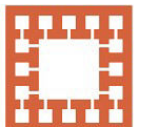
- Space:
  - Every invocation of a function call requires:
    - space for parameters and local variables
    - space for return address
  - Thus, a recursive algorithm needs space proportional to the number of nested calls to the same function.
- Time:
  - Calling a function involves
    - allocating, and later releasing, local memory
    - copying values into the local memory for the parameters
    - branching to/returning from the function
- All contribute to the time overhead.

Recursion: Cost



- ✓ You have to pay a price for recursion:
  - ➢ Calling a function **CONSUMES MORE TIME AND MEMORY** than adjusting a loop counter.
  - ➢ High performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.
- ✓ In **LESS DEMANDING APPLICATIONS** recursion is an attractive alternative for iteration.

Recursion: Precautions



You must always make sure that the recursion *bottoms out*:

- A recursive function must contain at least one non-recursive branch.
- The recursive calls must eventually lead to a non-recursive branch.

```
int fac(int numb){
   if(numb<=1)
       return 1;
    else
       return numb * fac(numb-1);
}
```

Recursion: Function

How many pairs of rabbits can be produced from a single pair in a year's time?

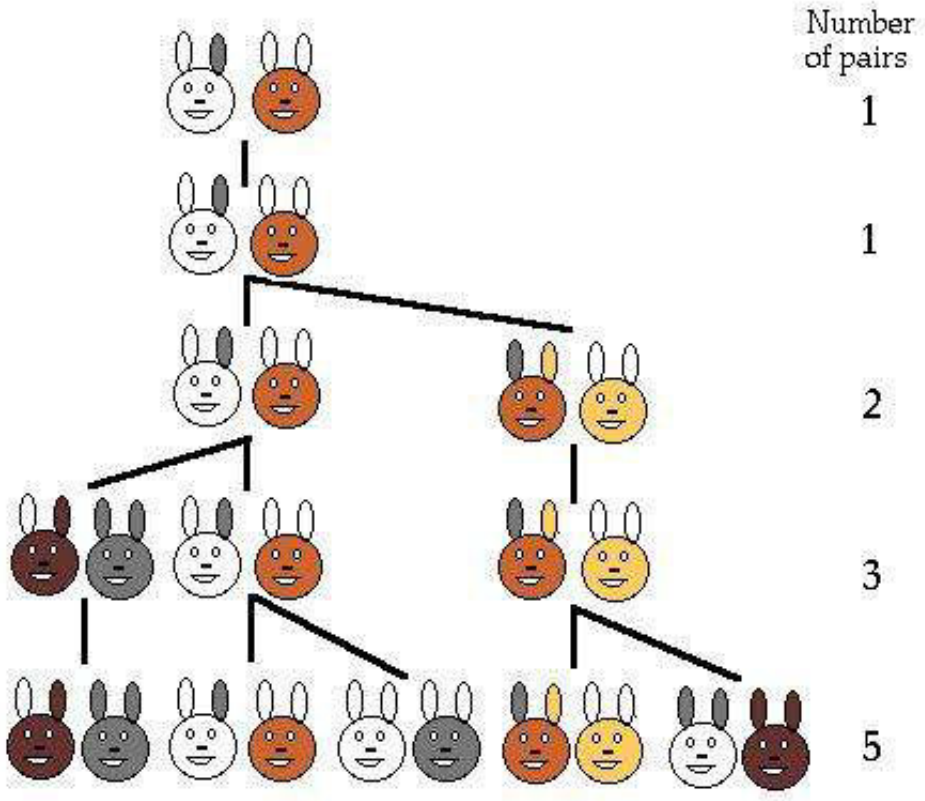
Assumptions:

- Each pair of rabbits produces a new pair of offspring every month;
- each new pair becomes fertile at the age of one month;
- none of the rabbits dies in that year.

Example:

- After 1 month there will be 2 pairs of rabbits;
- after 2 months, there will be 3 pairs;
- after 3 months, there will be 5 pairs (since the following month the original pair and the pair born during the first month will both produce a new pair and there will be 5 in all).

Population Growth in Nature



- ✓ Leonardo Pisano (Leonardo Fibonacci, son of Bonaccio) proposed the (Fibonacci) sequence in 1202 in *The Book of the Abacus*.
- ✓ Fibonacci numbers are believed to model nature to a certain extent, such as Kepler's observation of leaves and flowers in 1611.

Direct Computation Method

Fibonacci numbers:

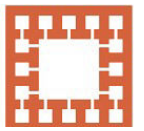
```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

where each number is the sum of the preceding two.

Recursive definition:

```
F(0) = 0;
F(1) = 1;
F(number) = F(number-1)+ F(number-2);
```

Recursion: Function



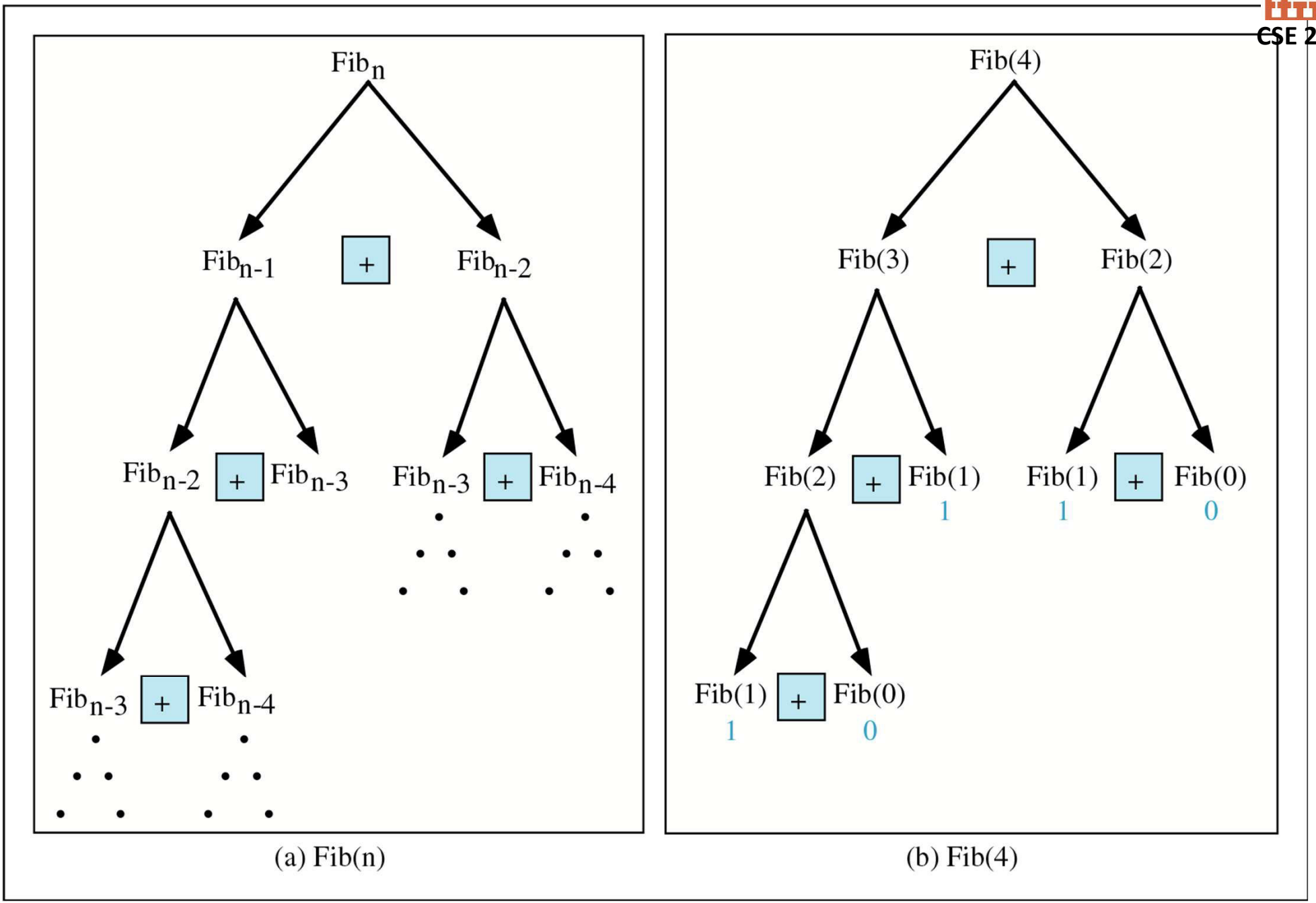

```
// Calculate Fibonacci numbers using recursive function.
// A very inefficient way, but illustrates recursion well
int fib(int number)
{
        if (number == 0) return 0;
        if (number == 1) return 1;
        return (fib(number-1) + fib(number-2));
}

int main(){   // driver function
        int inp_number;
        printf("Please enter an integer: ”);
        scanf(“%d”&inp_number);
        cout << "The Fibonacci number for "<< inp_number
                << " is "<< fib(inp_number)<<endl;
  return 0;
}
```

Recursive definition:

```
F(0) = 0;
F(1) = 1;
F(number)=F(number-1)+ F(number-2);
```

Recursion: Fibbonacci

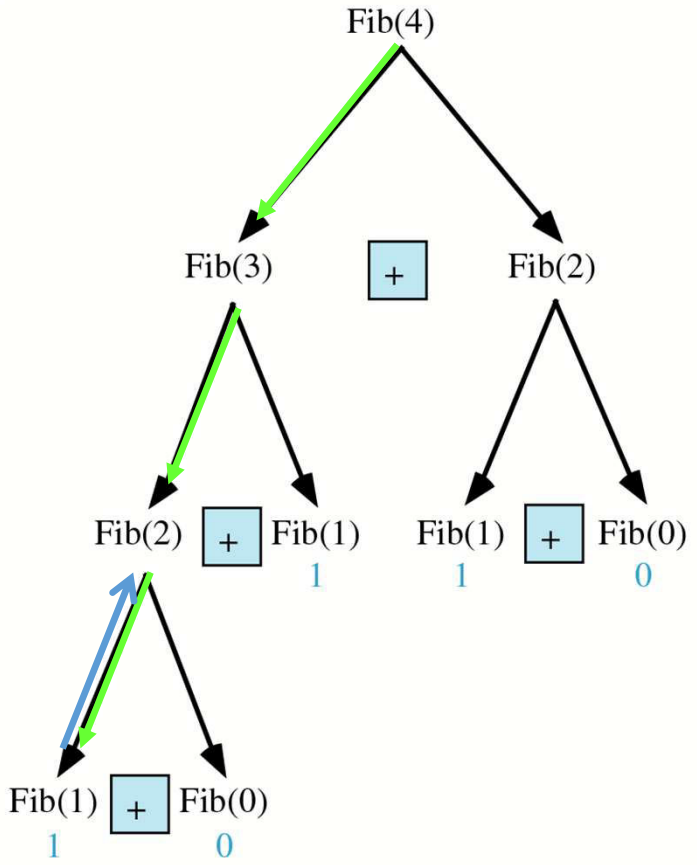


Recursion: Trace a Fibonacci Number


```
int fib(int num)
{
    if (num == 0) return 0;
    if (num == 1) return 1;
    return
      (fib(num-1)+fib(num-2));
}
```

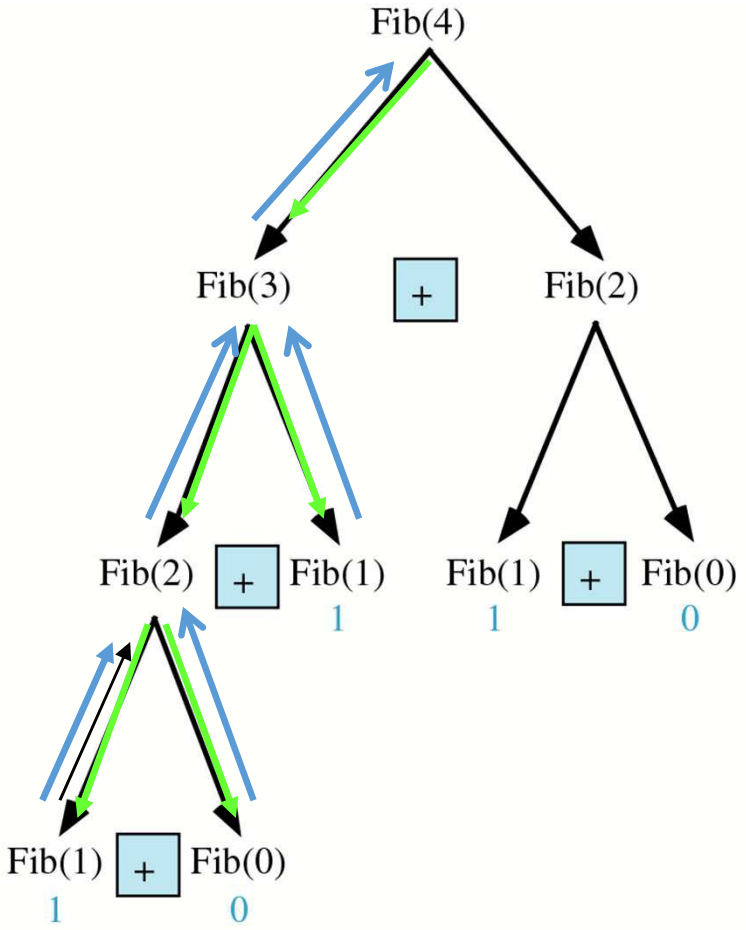
Assume the input number is 4, that is, num=4:

```
fib(4):
    4 == 0 ? No;    4 == 1? No.
    fib(4) = fib(3) + fib(2)
    fib(3):
        3 == 0 ? No; 3 == 1? No.
        fib(3) = fib(2) + fib(1)
        fib(2):
            2 == 0? No; 2==1? No.
            fib(2) = fib(1)+fib(0)
            fib(1):
                1== 0 ? No; 1 == 1? Yes.
                 fib(1) = 1;
                        return fib(1);
```



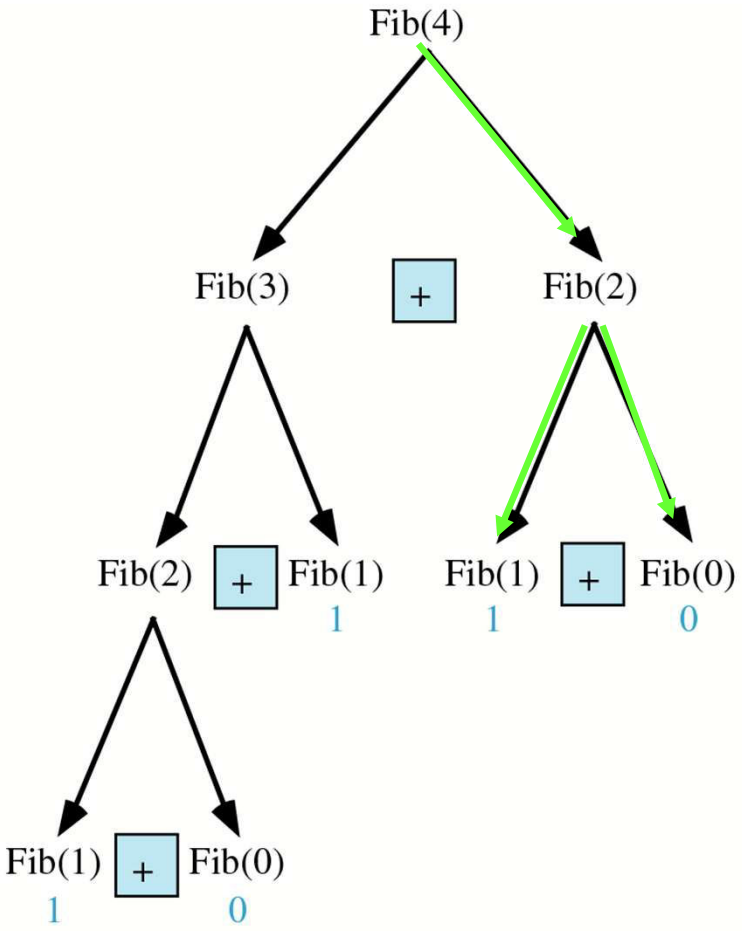
Recursion: Trace a Fibonacci Number


```
        fib(0):
            0 == 0 ?  Yes.
            fib(0) = 0;
            return fib(0);
    fib(2) = 1 + 0 = 1;
    return fib(2);
fib(3) = 1 + fib(1)
    fib(1):
     1 == 0 ? No; 1 == 1? Yes
     fib(1) = 1;
      return fib(1);
fib(3) = 1 + 1 = 2;
return fib(3)
```

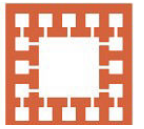


Recursion:


```
        fib(2):
    2 == 0 ? No; 2 == 1?    No.
    fib(2) = fib(1) + fib(0)
    fib(1):
       1== 0 ? No; 1 == 1?  Yes.
        fib(1) = 1;
        return fib(1);
     fib(0):
        0 == 0 ?    Yes.
        fib(0) = 0;
        return fib(0);
     fib(2) = 1 + 0 = 1;
     return fib(2);
fib(4) = fib(3) + fib(2)
         = 2 + 1 = 3;
return fib(4);
```



Recursion: Divide-and-Conquer Algorithms



- ✓ Consider a problem **P** associated with a set S.
- ✓ Suppose A is an algorithm which partitions S into smaller sets such that the solution of the problem **P** for S is reduced to the solution of P for one or more of the smaller sets.
- ✓ Then A is called a divide-and-conquer Algorithm.
- ✓ Examples:

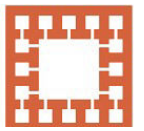
The Quicksort Algorithm

-use reduction step to find the location of a single element and to reduce the problem of sorting the entire set to the problem of sorting smaller sets

The Binary Search Algorithm

-divides the given sorted set into two halves so that the problem of searching for an item in the entire set is reduced to the problem of searching for the item in one of the two halves.

Recursion: Divide-and-Conquer Algorithms



CSE 2103

- ✓ A divide-and-conquer algorithm A may be viewed as a recursive procedure. But Why?
  - ➢ The divide-and-conquer algorithm A may be viewed as calling itself when it is applied to the smaller sets.
  - ➢ The base criteria for these algorithms are usually the one-element sets.
  - ➢ For example, with a sorting algorithm, a one-element set is automatically sorted, and
  - ➢ with a searching algorithm, one-element set requires only a single comparison.

TOWER of HANOI Problem

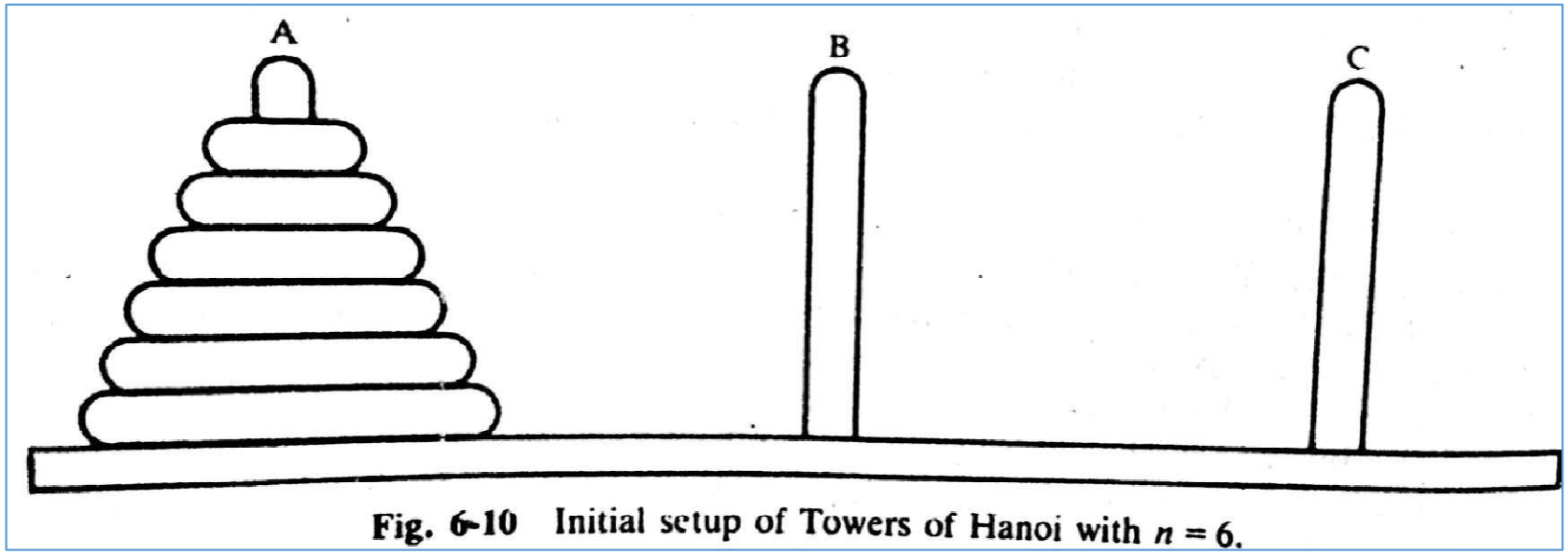


Fig. 6-10 Initial setup of Towers of Hanoi with $n = 6$.

TOWER of HANOI Problem

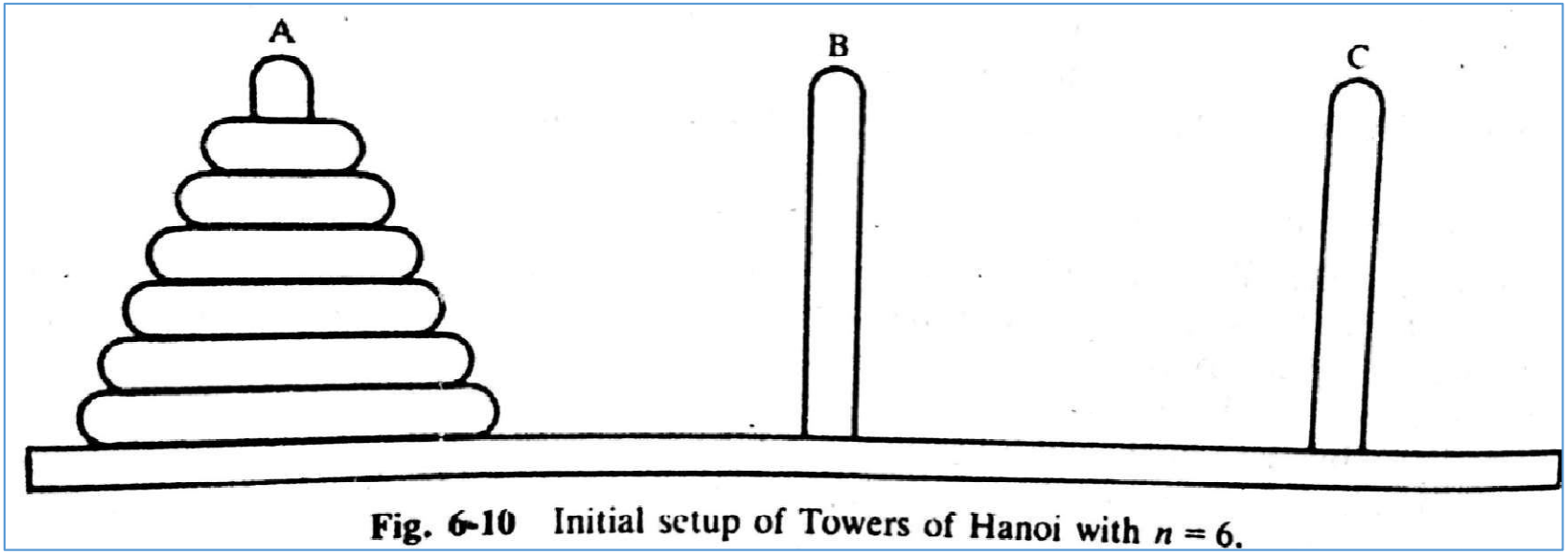
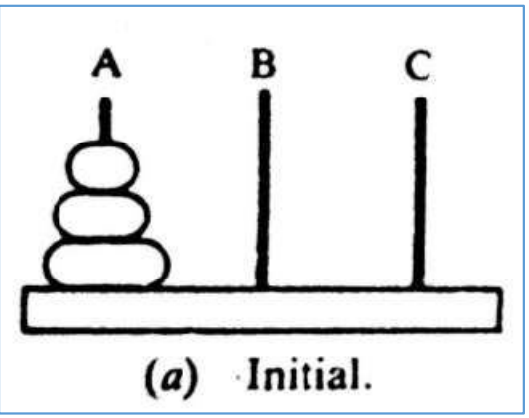


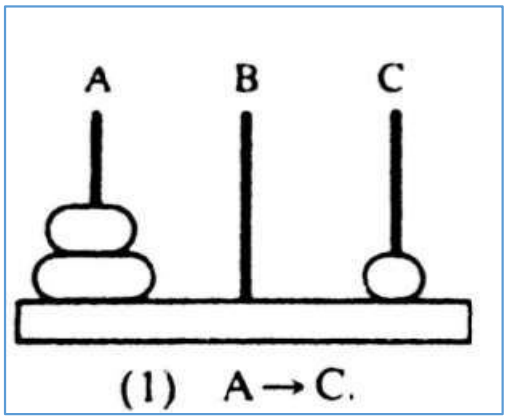
Fig. 6-10 Initial setup of Towers of Hanoi with $n = 6$.

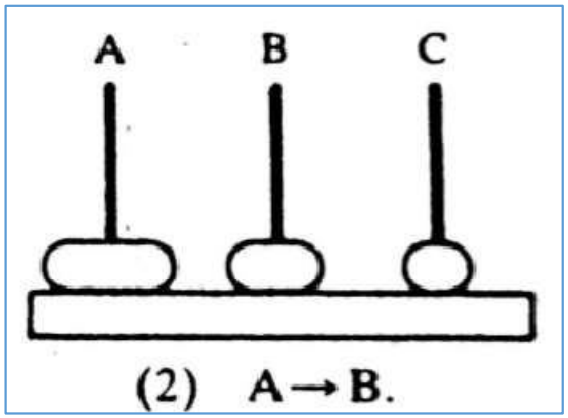
- ✓ Only one disc could be moved at a time
- ✓ A larger disc must never be stacked above a smaller one
- ✓ One and only one extra needle could be used for intermediate storage of discs

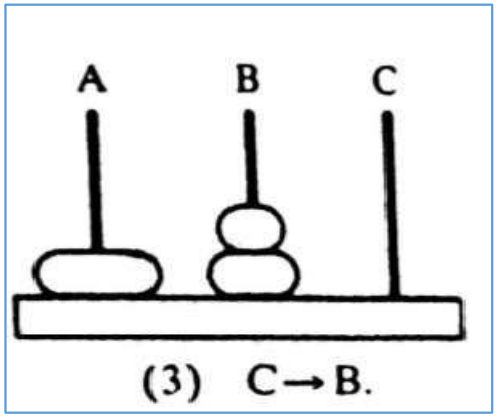
TOWER of HANOI Problem Solution

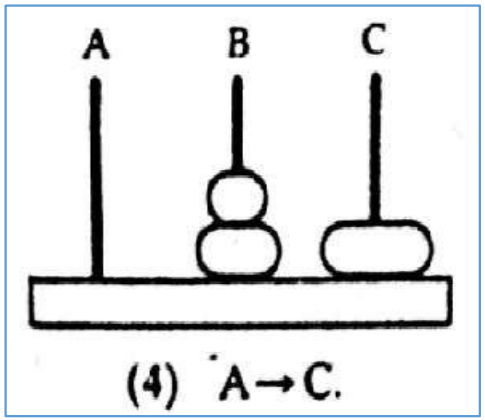
The solution to the Tower of Hanoi problem for n=3 appears

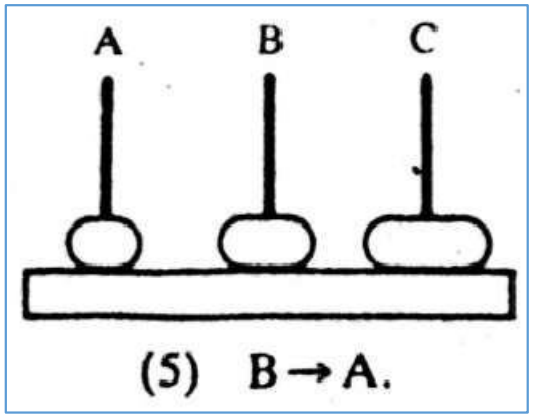


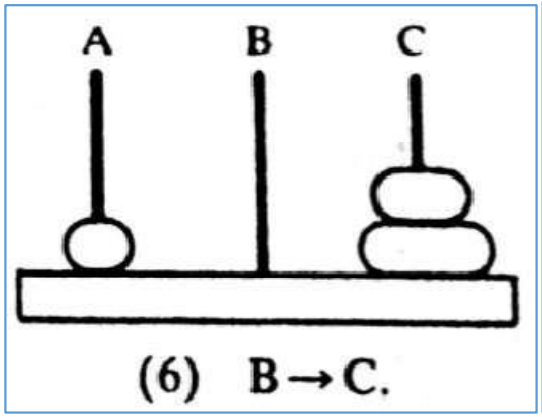


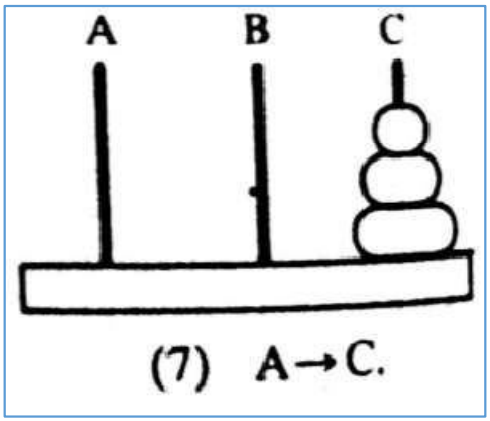






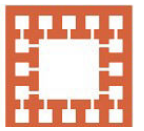






N=3: A→C, A→B, C→B, A→C, B→A, B→C, A→C,

TOWER of HANOI Problem Solution



CSE 2103

The **SEPARATE SOLUTION** to the Tower of Hanoi problem

for **n=1**

Solution: A→C

n=2

Solution: A→B, A→C, B→C

n=3

Solution: A→C, A→B, C→B, A→C, B→A, B→C, A→C

- ✓ **General Solution to the Tower of Hanoi for any # of Disk is PREFERRED.**
- ✓ **Which can be done in RECURSIVE way.**

Solution of TOWER of HANOI in RECURSIVE WAY

✓ The solution to the Tower of Hanoi problem for n>1 disks may be reduced to the following sub-problems:

- Move the top n-1 disks from peg A to peg B.
- Move the top disk from A to peg C: **A→C**
- Move the top n-1 disks from peg B to peg C

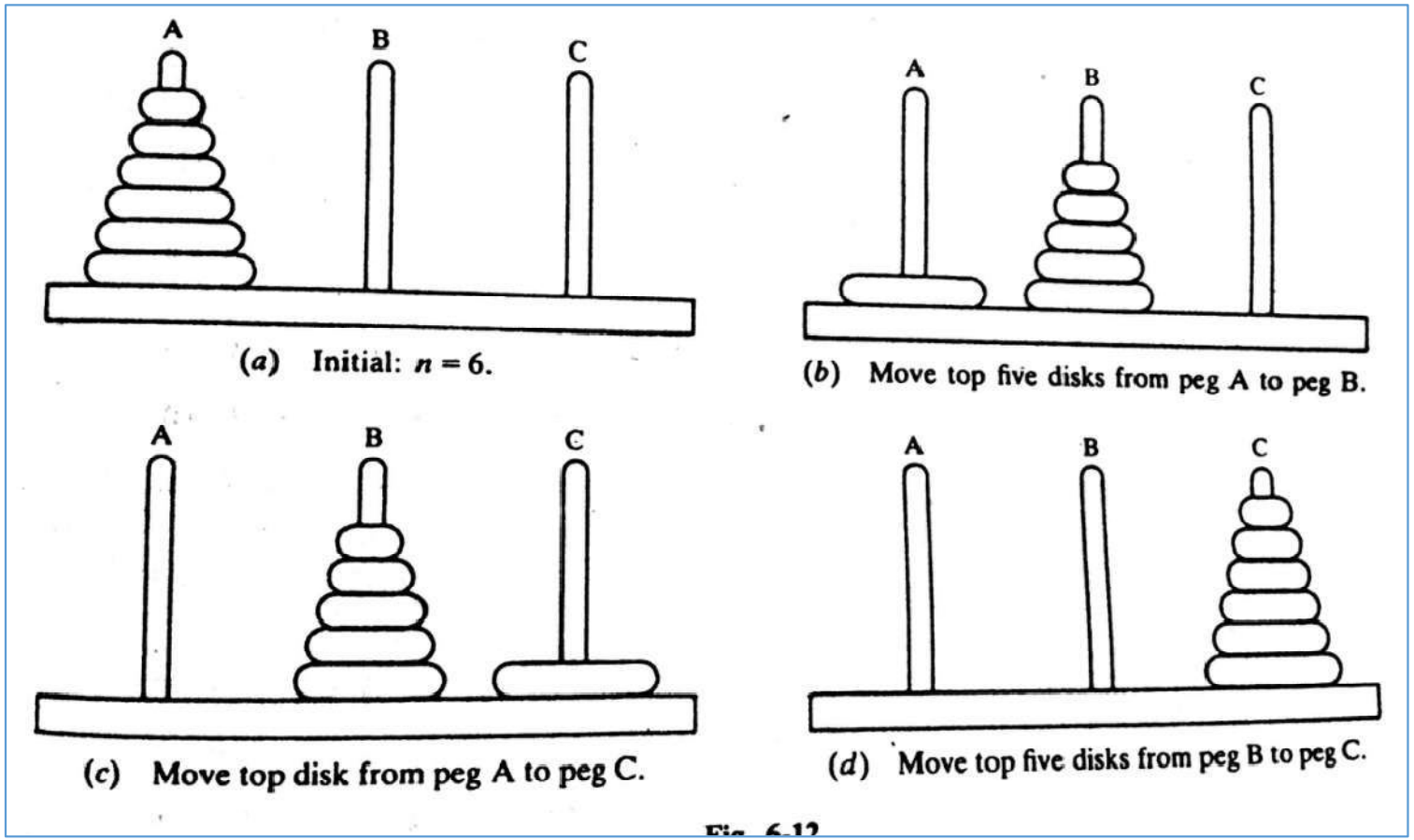
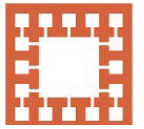


Fig. 6.12

Solution of TOWER of HANOI in RECURSIVE WAY



The general notation of the solution for any # of disk:

TOWER(N, BEG, AUX, END)

When **n=1** then

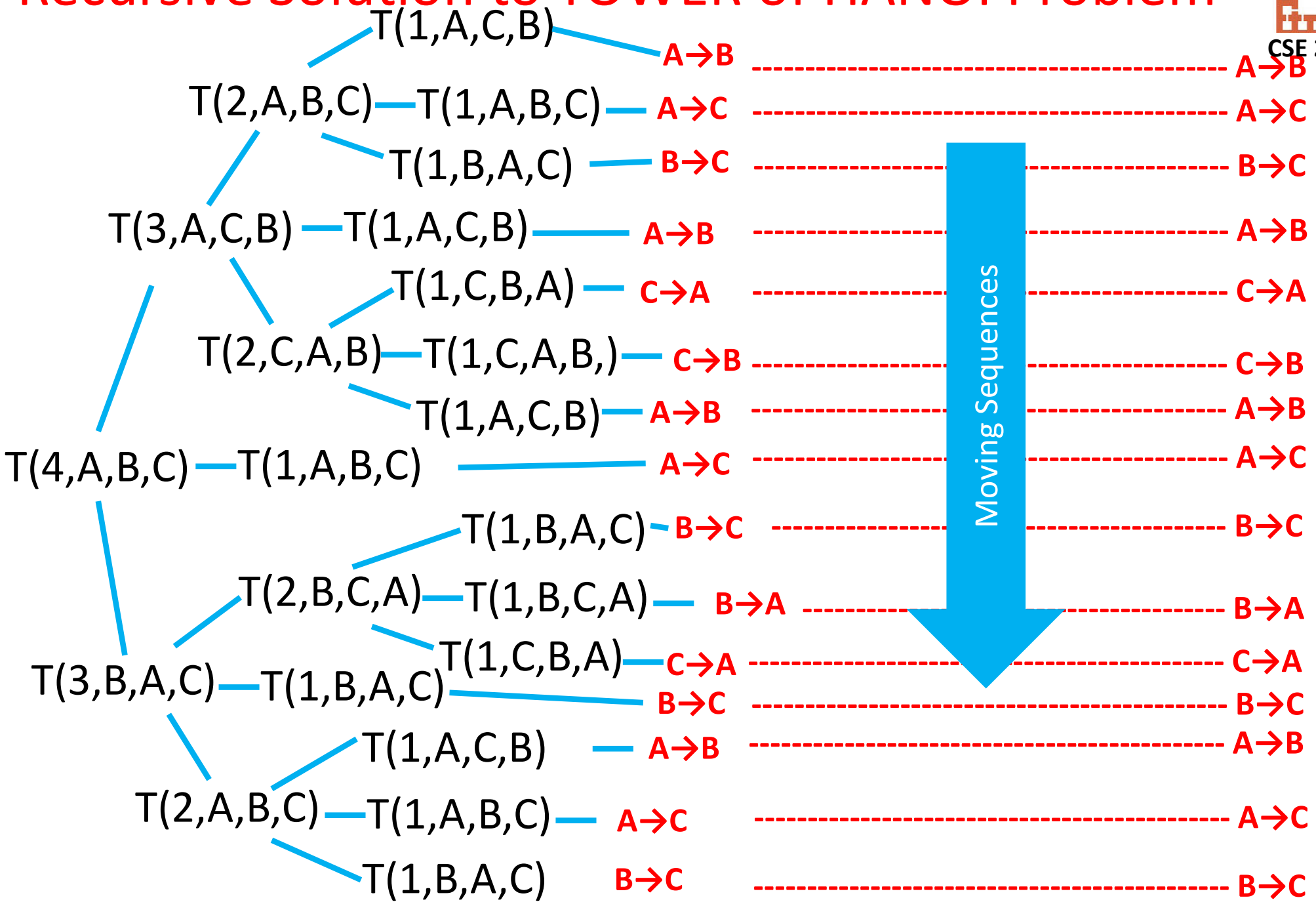
TOWER(1, BEG, AUX, END)

BEG → END, But

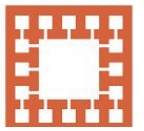
When **n> 1** then then solution may be reduced to the solution of the following three sub-problem:

(1) TOWER (N-1, BEG, END, AUX)
(2) TOWER (1, BEG, AUX, END)
(3) TOWER (N-1, AUX, BEG, END)

Recursive Solution to TOWER of HANOI Problem



Formal TOWER of HANOI Problem Solving Procedure



CSE 2103

TOWER (N, BEG, AUX, END)

Step1. If N=1, then:

(a) Write: BEG→END.

(b) Return.

[End of If Structure]

Step2. [Move N-1 disks from peg BEG to peg AUX.]

Call TOWER (N-1, BEG, END, AUX).

Step3. Write: BEG → END.

Step4. [Move N-1 disks from peg AUX to peg END.]

Call TOWER (N-1), AUG, BEG, END).

Step5. Return

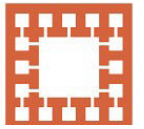
Implementation of Recursive Pro. By STACKS

Before implementing Recursive procedures…..

Imagine a SUBPROGRAM…

- ✓ It may contain both parameters, and local variables.
- ✓ The parameters are the variables which receive values from objects in the **CALLING PROGRAM**.
- ✓ It transmit values back to the **CALLING PROGRAM**.
- ✓ It must also keep track of the return address in the **CALLING PROGRAM**.
- ✓ The return address is essential, since control must be transferred back to its proper place in the **CALLING PROGRAM.**
- ✓ Once the subprogram finished executing and control is transferred back to the **CALLING PROGRAM.**
- ✓ The values of the local variables and the return address are no longer needed.

Implementation of Recursive Pro. By STACKS



CSE 2103

Suppose SUBPROGRAM is a recursive program…then

- ✓ Each level of execution of the subprogram may contain different values for the parameters and local variables, and for the return address.
- ✓ Furthermore, if the recursive program does call itself, then these current values must be saved, since they will be used again when the program is reactivated.