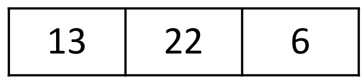
Lists and
Recursive Data Structures

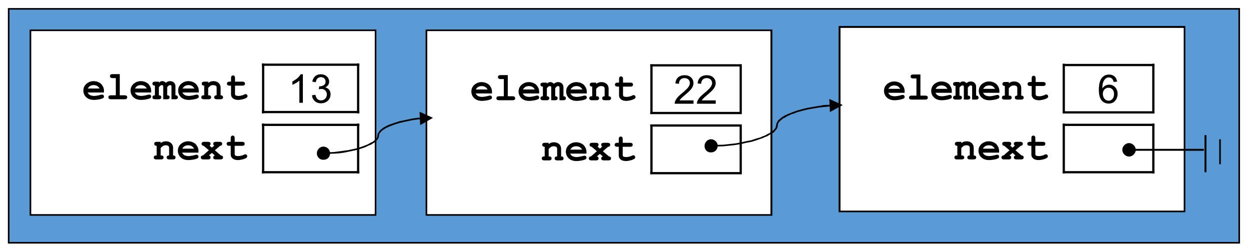
What is a List?

- Informally, a ***list*** is an *ordered* sequence of items.

- Examples
  - [10, 3, 6, 4, -3, 0]
  - [0, -3, 4, 6, 3, 10]
  - [3.6, 4.2, -1.1, 0.0, 999.9, 210.0]
  - [‘a’, ‘t’, ‘z’, ‘1’, ‘(’, ‘)’]
  - [“s10051111”, “s10052222”, “s10053333”, “s10054444”]

A list of [13, 22, 6]





Stacks and Queues are lists.

What is a List?

- Formally, a ***list*** is either
  - [ ] (called the empty list); or
  - an *element* followed by a ***list***.

- We are defining ***list*** in terms of ***list***. This is an example of ***recursive definitions***.

This is for the introduction of the concept of "recursive definitions" and "recursive functions". List is only used as an example to have a better understand of the concepts.

Recursive Types

- According to the definition,

A ***list*** is either
- the empty list []; or
- an *element* followed by a ***list***.

base case

recursive case

list

[]

$[e_1\ list_1]$

If $list_1$ is empty, list = $[e_1\ []]=[e_1]$

A list with one element

It is possible to have a list with one element

If $list_1$ is not empty

If $list_1$ has one element e_2,

list = $[e_1\ [e_2]] = [e_1\ e_2]$

If $list_1$ is in the form of $[e_2\ list_2]$, ,
list = $[e_1\ [e_2\ list_2\ ]] = [e_1\ e_2\ list_2]$

$[e_1\ e_2\ list_2]$

$[e_1\ e_2\ e_3]$

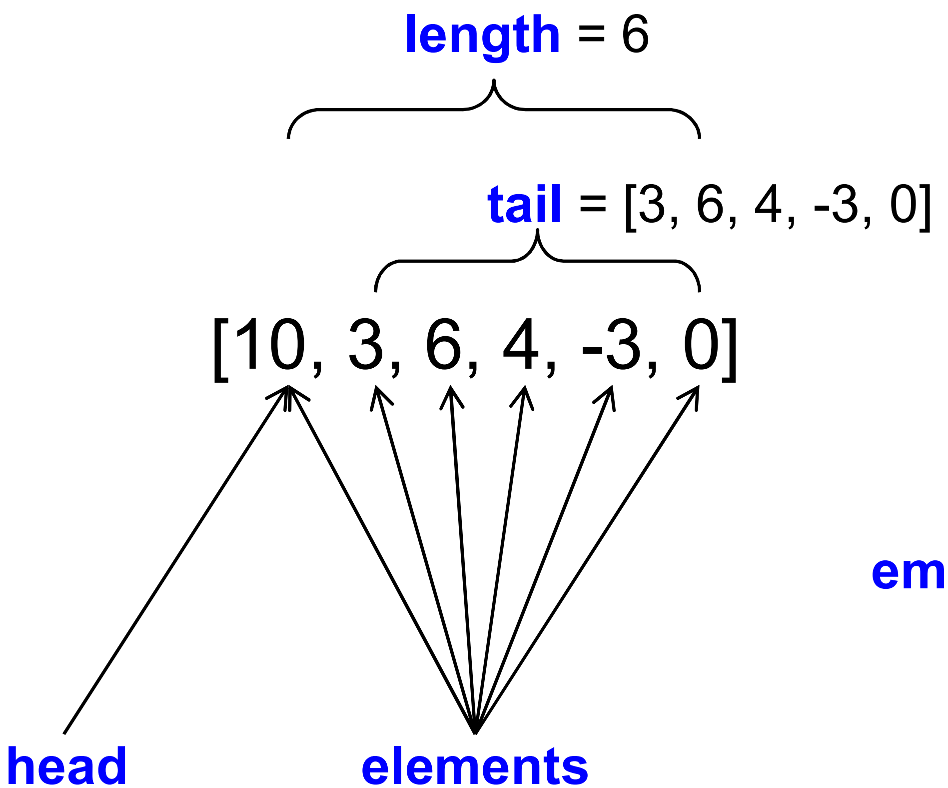
$[e_1\ e_2\ e_3\ list_3]$

$[e_1\ e_2\ e_3\ e_4]$

$[e_1\ e_2\ e_3\ e_4\ list_4\ ]$

- A data type which is partially defined in terms of itself is called a ***recursive type***.

List Terminologies



[ ]

empty list (length = 0)

List: *Head* and *Tail*

A ***list*** is either
- the empty list []; or
- an *element* followed by a ***list***.
  (i.e. [Head Tail])

- ***Very important*** concept

Head is an ***element***.
Tail is a ***list***.

- Example 1: [‘2’, ‘1’, ‘0’, ‘0’]
  - ***Head*** is the *element* ‘2’.
  - ***Tail*** is the *list* [‘1’, ‘0’, ‘0’].

This is ***unlike*** the tail of a queue, which is also an element.

List: *Head* and *Tail*

- Example 2: [1510, 2700]
  - ***Head*** is the *element* 1510.
  - ***Tail*** is the *list* [2700].
- Example 3: [2700]
  - ***Head*** is the *element* 2700.
  - ***Tail*** is the (empty) *list* [].
- Example 4: []
  - ***Head***?
  - ***Tail***?

A ***list*** is either
- the empty list []; or
- [Head Tail]

Head is an ***element***.
Tail is a ***list***.

An element is an integer

By definition, the empty list [] has ***no*** *head* and ***no*** *tail*.

List: *Head* and *Tail*

- Example 5: [[11, 1], [0], [2, 10, 0]]
  - ***Head*** is the *element* [11, 1].
  - ***Tail*** is the ***list*** [[0], [2, 10, 0]].

- Exercise: [[210, 0], []]
  - ***Head***?
  - ***Tail***?

- Exercise: [[]]
  - ***Head***?
  - ***Tail***?

A ***list*** is either
- the empty list []; or
- [Head Tail]

Head is an ***element***.
Tail is a ***list***.

An element is a list

Operations on Lists

- There are many possible operations on lists, but the *basic* ones are
  - ***CreateList***: creating a list from a *head* and a *tail*.
  - ***ListHead***: obtaining the *head* of a list.
  - ***ListTail***: obtaining the *tail* of a list.

List Operations: CreateList

- ***CreateList***
  - *Create* a list from a *head* and a *tail*.
  - CreateList(h, [e_1, e_2, e_3, …]) → [h, e_1, e_2, e_3, …]

- Examples
  - CreateList(2, [6, 8, 4, 5, 3]) → [2, 6, 8, 4, 5, 3]
  - CreateList(3, [4]) → [3, 4]
  - CreateList(5, []) → [5]

List Operations: CreateList

- *Any* list can be constructed using CreateList and the empty list [ ].

CreateList(3, CreateList(4, CreateList(5, [ ])))

→CreateList(3, CreateList(4, **[5]**))

→CreateList(3, **[4, 5]**)

→**[3, 4, 5]**

CreateList: Creating a list from a *head* and a *tail*.

List Operations: CreateList

CreateList([], CreateList([6], CreateList([4, 5], [])))

The element is a list.

→CreateList([], CreateList([6], **[[4, 5]]**))

→CreateList([], **[[6], [4, 5]]**)

***Note*:**
An empty list is an element that cannot be ignored.

→**[[], [6], [4, 5]]**

CreateList: Creating a list from a *head* and a *tail*.

List Operation: ListHead

- ***ListHead***
  - Obtaining the *head* of a list.
  - ListHead([e_1, e_2, e_3, e_4, …]) → e_1

Remember:
head is an ***element***.

- Examples
  - ListHead([2, 6, 8, 4, 5, 3]) → 2
  - ListHead([3]) → 3
  - ListHead([[3, 4], [7, 6, 9]]) → [3, 4]
  - ListHead([]) → *error*

List Operation: ListTail

- ***ListTail***
  - Obtaining the *tail* of a list.
  - ListTail([e_1, e_2, e_3, e_4, …]) → [e_2, e_3, e_4, …]

Remember: ***tail*** is a ***list***.

- Examples
  - ListTail([2, 6, 8, 4, 5, 3]) → [6, 8, 4, 5, 3]
  - ListTail([3]) → []
  - ListTail([[3, 4], [7], [2, 9, 6]]) → [[7], [2, 9, 6]]
  - ListTail([]) → *error*

List Operations

CreateList(ListHead(ListTail(CreateList(3, [4]))), [6])

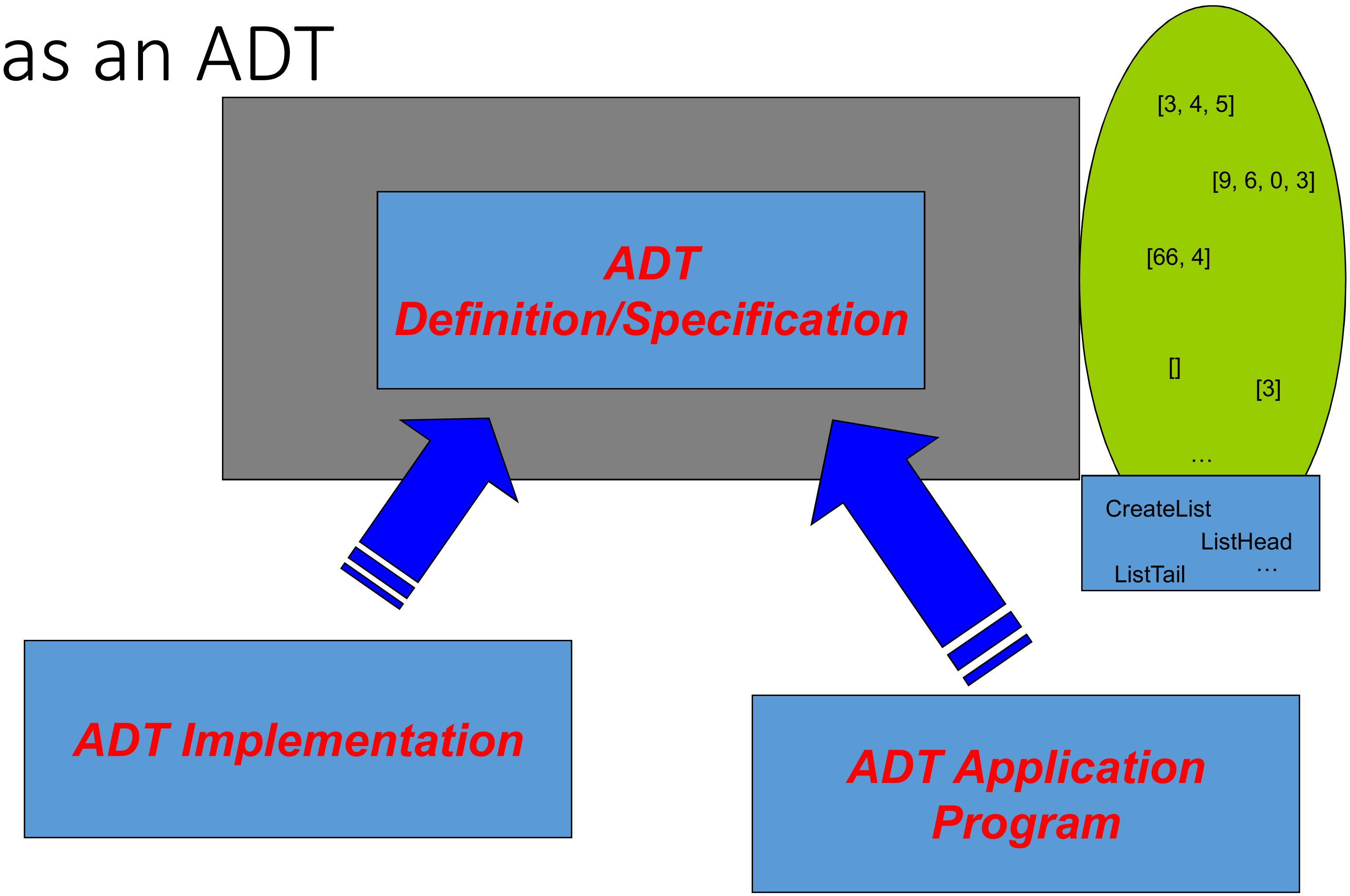
→CreateList(ListHead(ListTail(**[3, 4]**)), [6])

→CreateList(ListHead(**[4]**), [6])

→CreateList(**4**, [6])

→**[4, 6]**

List as an ADT



Defining a List ADT: list.h

list.h

```
typedef struct listCDT *listADT;

typedef int listElementT;

listADT EmptyList();
listADT CreateList(listElementT head, listADT tail);
listElementT ListHead(listADT list);
listADT ListTail(listADT list);
int ListIsEmpty(listADT list);
```

We define list of *integers* here. Changing int to void * defines list of "*anything*."

Defining a List ADT

```
listADT EmptyList();
```

- Returns/creates the empty list [].
- (We need this function since we cannot use `[]` in C to denote the empty list directly. "`[]`" is just a symbol.)

```
listADT CreateList(listElementT head,
                   listADT tail);
```

- Creates and returns a new list with `head` followed by `tail`.

Defining a List ADT

`listElementT ListHead(listADT list);`

- Returns the head of **`list`**.

`listADT ListTail(listADT list);`

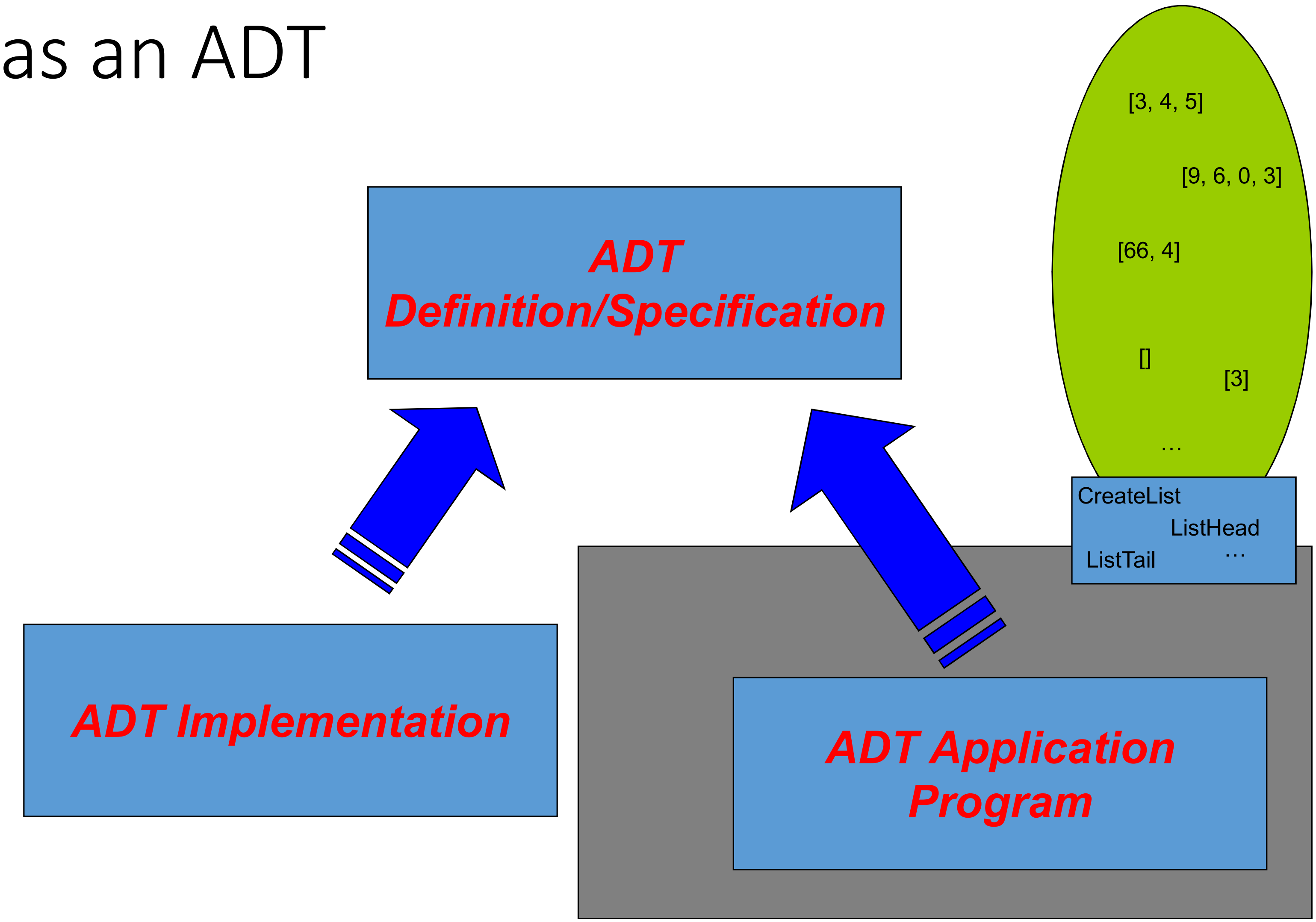
- Returns the tail of **`list`**.

`int ListIsEmpty(listADT list);`

- Returns 1 if **`list`** is the empty list []; 0 otherwise.

Head is an ***element***.
Tail is a ***list***.

List as an ADT



Using the List ADT

- Do not underestimate the power of this highly simple interface. We can use it to develop very useful applications.

- Instead of giving a complete application that uses lists, we discuss several common *client-level* functions that work with the list ADT.

- We start with a function that returns the *length* of a list.

List Length: Iteration

- A straightforward implementation using iteration.

```
int ListLength(listADT list) {
   listADT L;
   int n = 0;
   for (L = list; !ListIsEmpty(L); L = ListTail(L))
      n++;
   return n;
}
```

- ListLength([3, 7, 6, 4]) → 4
- ListLength([5]) → 1
- ListLength([]) → 0

list = [3, 7, 6, 4]	
L = [3, 7, 6, 4]	n = 1
L = [7, 6, 4]	n = 2
L = [6, 4]	n = 3
L = [4]	n = 4
L = []	

List Length: Recursion

- It is also possible to write the function that more closely reflect the *recursive* nature of a list.

```
int ListLength(listADT list) {
   if (ListIsEmpty(list))
      return 0;
   else
      return 1 + ListLength(ListTail(list));
}
```

- A ***recursive function*** is a function that makes a call to itself.

Recursive Functions

- Any recursive function will include the following three basic elements.
  - A *test* to stop or continue the recursion.
  - An *end case* that terminates the recursion.
  - A *recursive call* that continues the recursion.

```
int ListLength(listADT list) {
    if (ListIsEmpty(list))
        return 0;
    else
        return 1 + ListLength(ListTail(list));
}
```

A *test* to stop or continue

An *end case* to terminate the recursion

A *recursive call* to continue recursion

```
int ListLength(listADT list) {
    if (ListIsEmpty(list))
        return 0;
    else
        return 1 + ListLength(ListTail(list));
}
```

ListLength([3, 7, 6, 4])

→1 + ListLength([7, 6, 4])

→1 + (1 + ListLength([6, 4]))

→1 + (1 + (1 + ListLength([4])))

→1 + (1 + (1 + (1 + ListLength([]))))

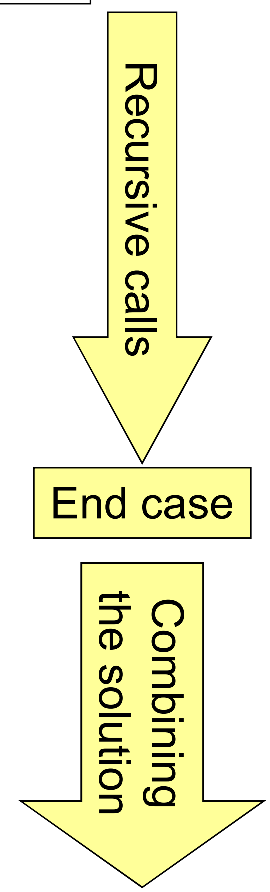
→1 + (1 + (1 + (1 + 0)))

→1 + (1 + (1 + 1))

→1 + (1 + 2)

→1 + 3

→4



Think Recursively

- Writing recursive functions requires recursive thinking style.

- A *recursive* definition of list length
  - The ***length*** of the empty list [] is 0.
  - The ***length*** of a non-empty list is "1 + the ***length*** of its tail."

```
int ListLength(listADT list) {
   if (ListIsEmpty(list))
      return 0;
   else
      return 1 + ListLength(ListTail(list));
}
```

Recursion

- ***Recursion*** usually leads to more *elegant* and *simpler* solutions, although it incurs *larger* memory and time overhead.

- An important recursive problem-solving skill is ***divide-and-conquer***.
  - *Divide* the problem into smaller pieces.
  - *Tackle* each sub-task either directly or by recursion.
  - *Combine* the solutions of the parts to form the solution of the whole

Obtaining List Element

- The next function is to obtain the *n*th element of a list.
- NthElement(list, n) → result
- Examples :
  - NthElement([6, 9, 5, 2, 3], 0) → 6
  - NthElement([6, 9, 5, 2, 3], 3) → 2
  - NthElement([6, 9, 5, 2, 3], 6) → *error*

Obtaining List Element

- The next function is to obtain the n^{th} element of a list.
- NthElement(list, n) → result

- Think recursively
  - The 0^{th} element of a list is its head.
    - NthElement(L, 0) → ListHead(L)
  - For $n > 0$, the n^{th} element of a list is the $(n\text{-}1)^{th}$ element of its tail.
    - NthElement([e_1, e_2, e_3, …] , n) → NthElement([e_2, e_3, …], n-1)

$$[e_1\ e_2 \ldots e_n] \xrightarrow{\text{tail}} [e_2\ e_3 \ldots e_n]$$

For $n > 0$ The n^{th} element of the list [head tail]

= The $n\text{-}1^{th}$ element of the list [tail]

Obtaining List Element

- Obtaining the *n*th element of a list.

```
listElementT NthElement(listADT list, int n) {
   if (ListIsEmpty(list))
      exit(0);
   else if (n == 0)
      return ListHead(list);
   else
      return NthElement(ListTail(list), n - 1);
}
```

A *test* to stop or continue

An *end case* to terminate the recursion

A *recursive call* to continue recursion

- The 0th element of a list is its head.
- The *n*th element of a list is the (*n* - 1)th element of its tail.

```
listElementT NthElement(listADT list, int n) {
   if (ListIsEmpty(list))
      exit(0);
   else if (n == 0)
      return ListHead(list);
   else
      return NthElement(ListTail(list), n - 1);
}
```

NthElement([6, 9, 5, 2, 3], 3)

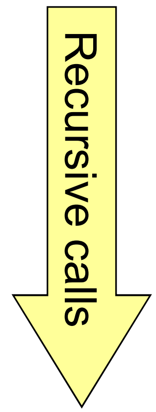
→ NthElement([9, 5, 2, 3], 2)

→ NthElement([5, 2, 3], 1)

→ NthElement([2, 3], 0)

→ ListHead([2, 3])

→ 2



End case

No need to combine solution in this example

```c
listElementT NthElement(listADT list, int n) {
   if (ListIsEmpty(list))
      exit(0);
   else if (n == 0)
      return ListHead(list);
   else
      return NthElement(ListTail(list), n - 1);
}
```

NthElement([6, 9, 5, 2, 3], 6)

→ NthElement([9, 5, 2, 3], 5)

→ NthElement([5, 2, 3], 4)

→ NthElement([2, 3], 3)

→ NthElement([3], 2)

→ NthElement([], 1)

→ *error*

Recursive calls

End case
(error in this example)

Displaying a List

```
void DisplayList(listADT list) {
   printf("[");
   RecDisplayList(list);
   printf("]\n");
}

void RecDisplayList(listADT list) {
   listADT tail;

   if (!ListIsEmpty(list)) {
      printf("%d", ListHead(list));
      tail = ListTail(list);
      if (!ListIsEmpty(tail))
         printf(", ");
      RecDisplayList(tail);
   }
}
```

A *test* to stop or continue

A *recursive call* to continue recursion

An *end case* to terminate the recursion, i.e., *else: do nothing*

```
void DisplayList(listADT list) {
   printf("[");
   RecDisplayList(list);
   printf("]\n");
}

void RecDisplayList(listADT list) {
   listADT tail;

   if (!ListIsEmpty(list)) {
      printf("%d", ListHead(list));
      tail = ListTail(list);
      if (!ListIsEmpty(tail))
         printf(", ");
      RecDisplayList(tail);
   }
}
```

DisplayList([3, 4, 2])
→ RecDisplayList([3, 4, 2])
→ RecDisplayList([4, 2])
→ RecDisplayList([2])
→ RecDisplayList([]);
→ *done*

```
[
[3,_
[3,_4,_
[3,_4,_2
[3,_4,_2
[3,_4,_2]↵
```

Nothing to do in *end case*

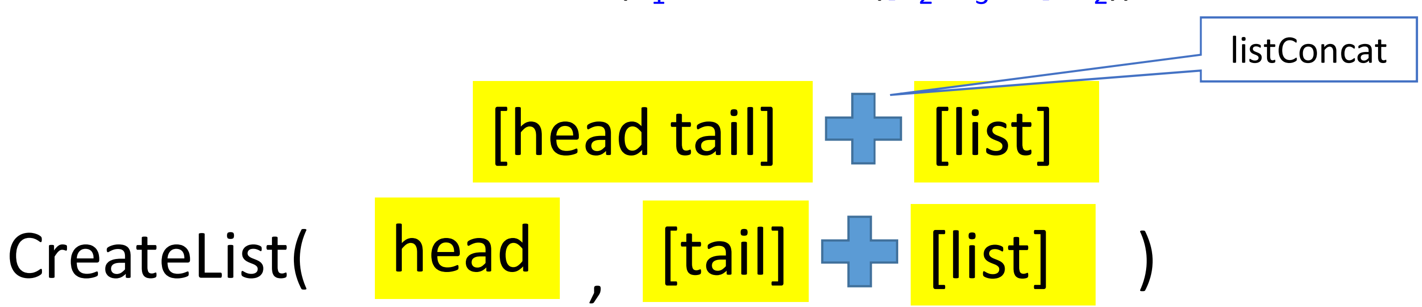
List Concatenation

- Returns a list which is a concatenation of two lists.

- ListConcat([3, 4, 5], [6, 7]) → [3, 4, 5, 6, 7]
- ListConcat([9, 6, 5, 4], []) → [9, 6, 5, 4]
- ListConcat([], [6, 7]) → [6, 7]
- ListConcat([], []) → []

List Concatenation

- Again, think recursively
  - ***Concatenation*** of [] and a list L is L.
    - ListConcat([], L_2) → L_2
  - ***Concatenation*** of a non-empty list L_1 and another list L_2 is the head of L_1 followed by ***concatenation*** of the tail of L_1 and L_2.
    - ListConcat([e_1, e_2, e_3, …], L_2)
      → CreateList(e_1, ListConcat([e_2, e_3, …], L_2))



ListConcat([4, 2, 6], [5, 7])

→ **CreateList(4, ListConcat([2, 6], [5, 7]))**

→ CreateList(4, **CreateList(2, ListConcat([6], [5, 7]))**)

→ CreateList(4, CreateList(2, **CreateList(6, ListConcat([], [5, 7]))**))

→ CreateList(4, CreateList(2, CreateList(6, **[5, 7]**)))

→ CreateList(4, CreateList(2, **[6, 5, 7]**))

→ CreateList(4, **[2, 6, 5, 7]**)

→ **[4, 2, 6, 5, 7]**

Recursive calls

End case

Combining the solution

ListConcat([], L_2) → L_2

ListConcat([e_1, e_2, e_3, …], L_2)

→ CreateList(e_1, ListConcat([e_2, e_3, …], L_2))

List Concatenation

- A direct translation of the recursive definition to programming code.

```
listADT ListConcat(listADT list1, listADT list2) {

   if (ListIsEmpty(list1))
      return list2;
   else
      return CreateList(ListHead(list1), ListConcat(ListTail(list1), list2));
}
```

A *test* to stop or continue

An *end case* to terminate the recursion

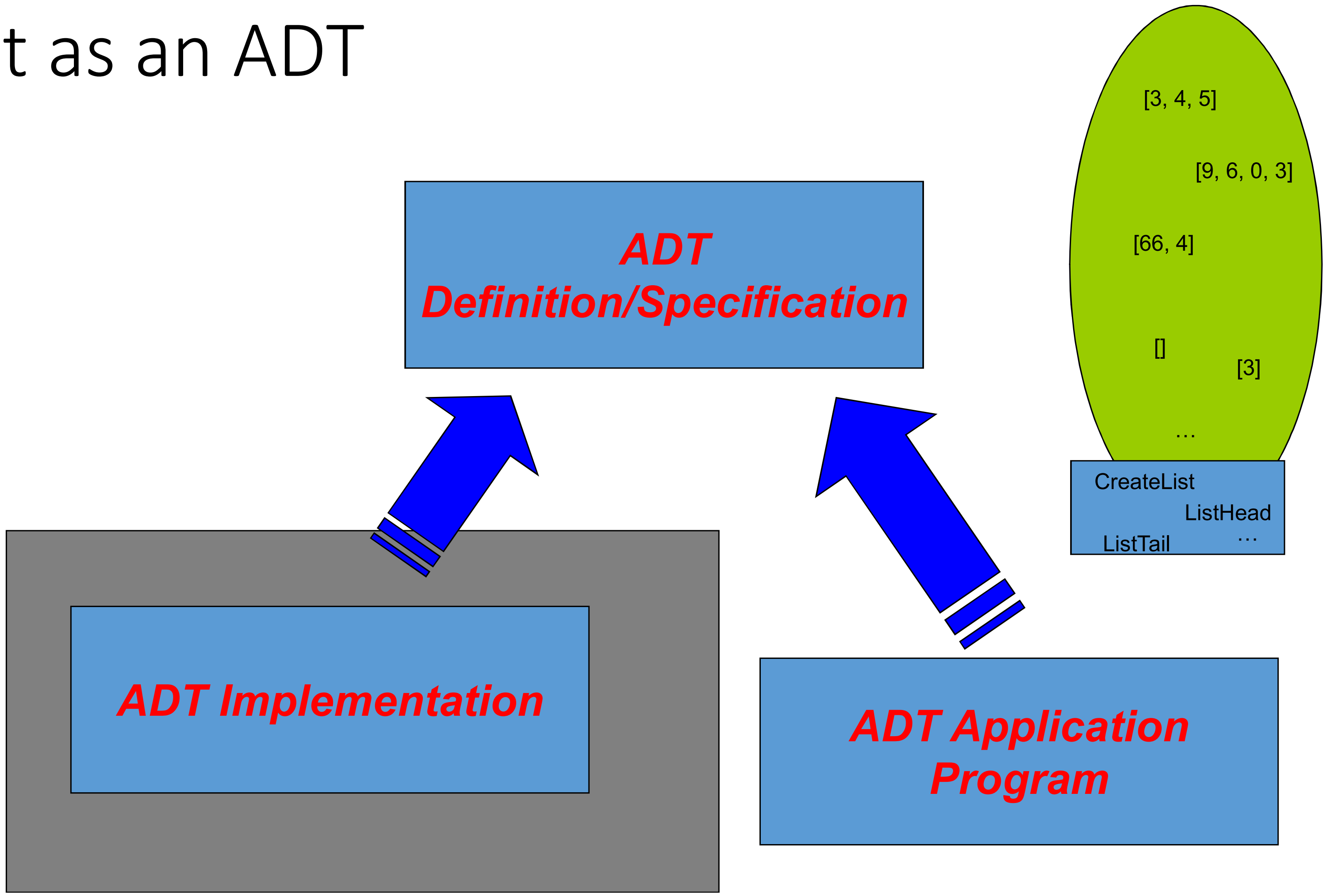
A *recursive call* to continue recursion

ListConcat([], L_2) → L_2

ListConcat([e_1, e_2, e_3, …], L_2)

→ CreateList(e_1, ListConcat([e_2, e_3, …], L_2))

List as an ADT

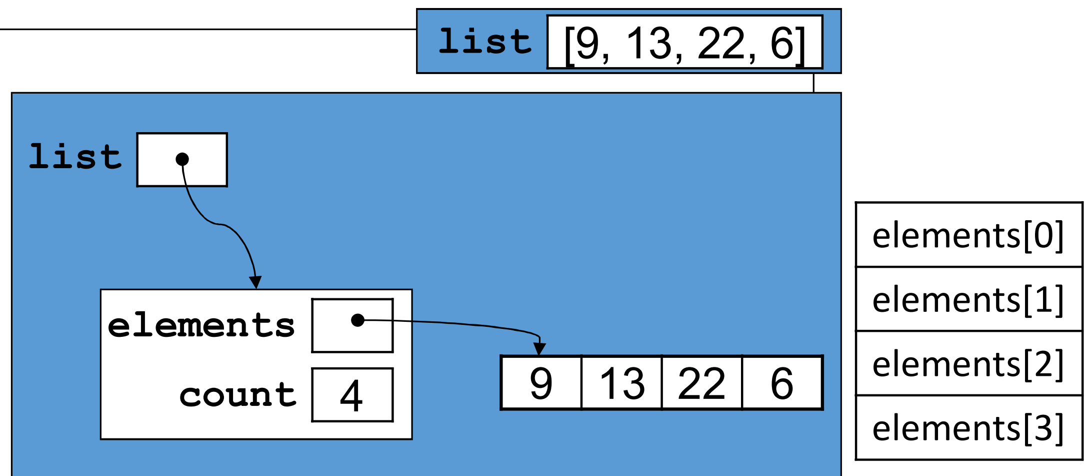


Implementing the List ADT

- It is straightforward to use arrays to implement the list ADT. (Version ***1.0***)

list.c

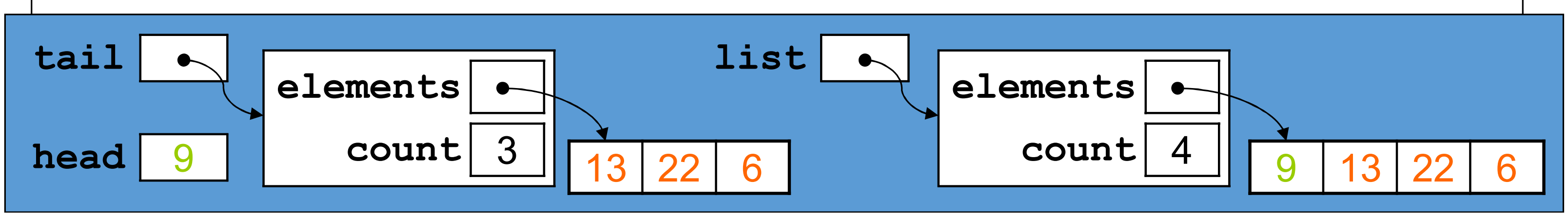
```c
#include "list.h"
#include <stdlib.h>

struct listCDT {
   listElementT *elements;
   int count;
};

listADT EmptyList() {
   listADT list;
   list = (listADT)malloc(sizeof(struct listCDT));
   list->elements = NULL;
   list->count = 0;
}
```



`list.c` (continue)

```
listADT CreateList(listElementT head, listADT tail) {
   int i;
   listADT list;

   list = EmptyList();
   list->count = tail->count + 1;
   list->elements = (listElementT *)malloc(
                       list->count * sizeof(listElementT));
   list->elements[0] = head;    // copy the head
   for (i = 0; i < tail->count; i++)    // copy the tail
      list->elements[i + 1] = tail->elements[i];
   return list;
}

listElementT ListHead(listADT list) {
   if (ListIsEmpty(list))
      exit(0);
   return list->elements[0];
}
```

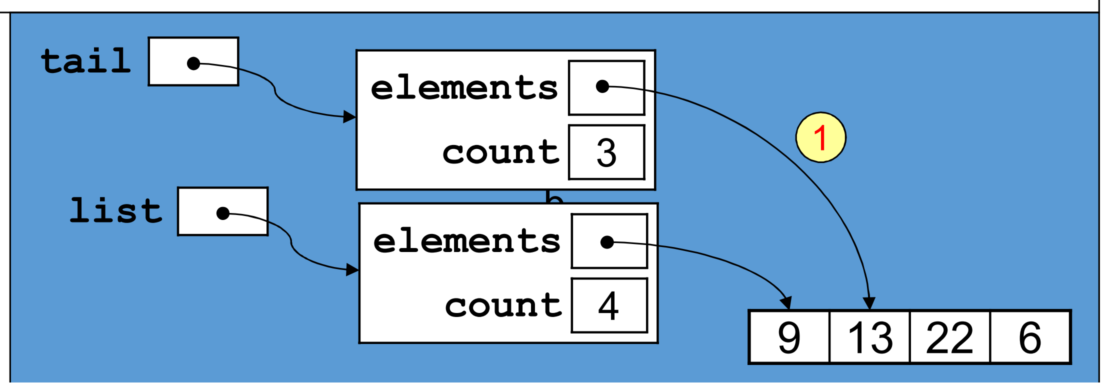
Allocate memory to elements



`list.c` (continue)

```
listADT ListTail(listADT list) {
   int i;
   listADT tail;

   if (ListIsEmpty(list))
      exit(0);
   tail = EmptyList();
   tail->count = list->count - 1;
1  tail->elements = list->elements + 1;    // pointer arithmetic
   return tail;
}

int ListIsEmpty(listADT list) {
   return (list->count == 0);    // or (list->elements == NULL)
}
```

tail	[13, 22, 6]
list	[9, 13, 22, 6]



List *Implementation* (Ver *2.0*)

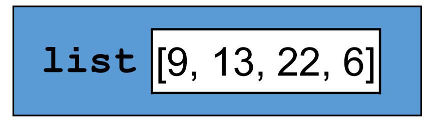
- Ver 1.0 is an ***in***efficient implementation, since we potentially need to copy many elements during CreateList.

- The *recursive nature* of a list suggests that we can represent a list using its *head* and *tail*. (Ver ***2.0***)

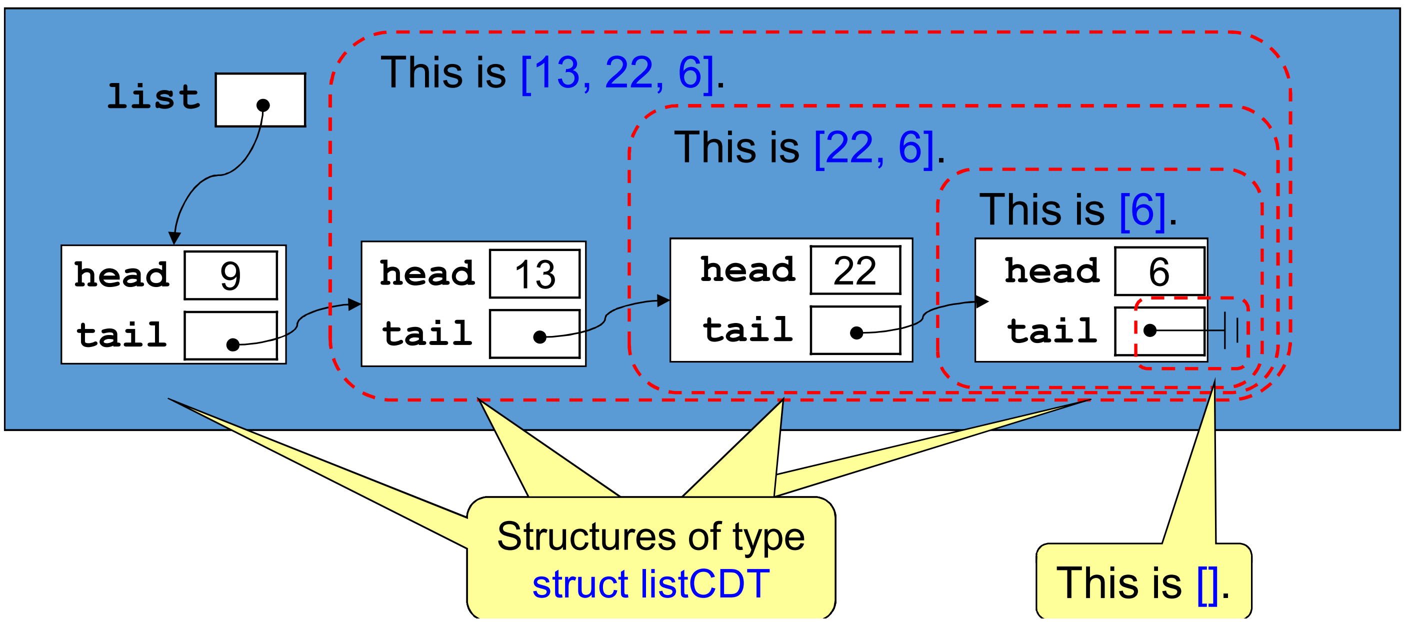
```
struct listCDT {
   listElementT head;
   listADT tail;
};
```

List *Implementation* (Ver *2.0*)

```
struct listCDT {
    listElementT head;
    listADT tail;
};
```



- In fact, this is a linked list representation.

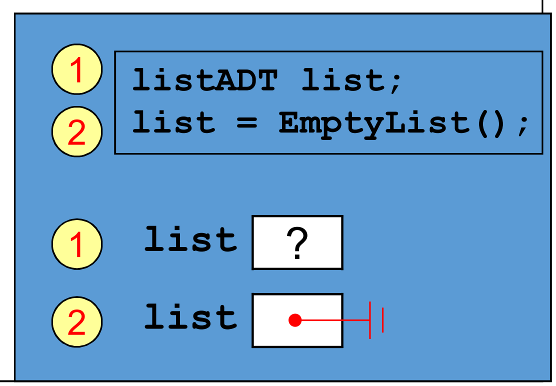


List *Implementation* (Ver *2.0*)

- With this recursive representation, the implementation is very simple.

list.c

```c
#include "list.h"
#include <stdlib.h>

struct listCDT {
   listElementT head;
   listADT tail;
};

listADT EmptyList() {
   return NULL;
}
```

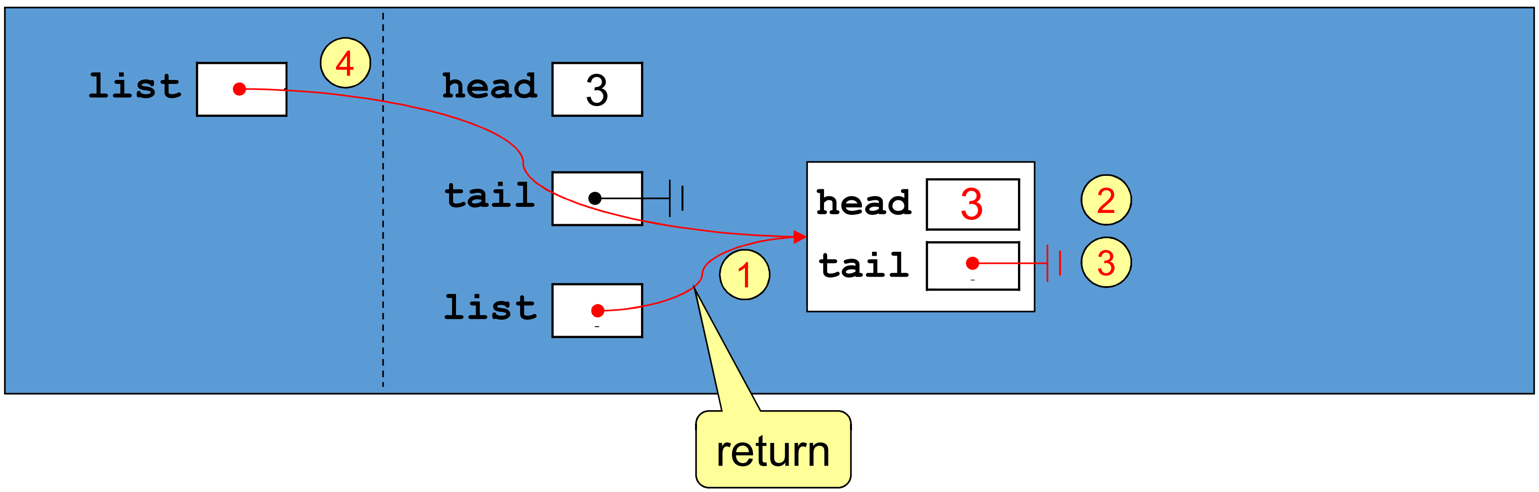


list.c (continue)

```
listADT CreateList(listElementT head, listADT tail) {
   listADT list;

   list = (listADT)malloc(sizeof(struct listCDT)); 1
   list->head = head; 2
   list->tail = tail; 3
   return list; 4
}
```

```
listADT list;
list = EmptyList();
list = CreateList(3, list);
list = CreateList(4, list);
```

CreateList(3, []) → [3]

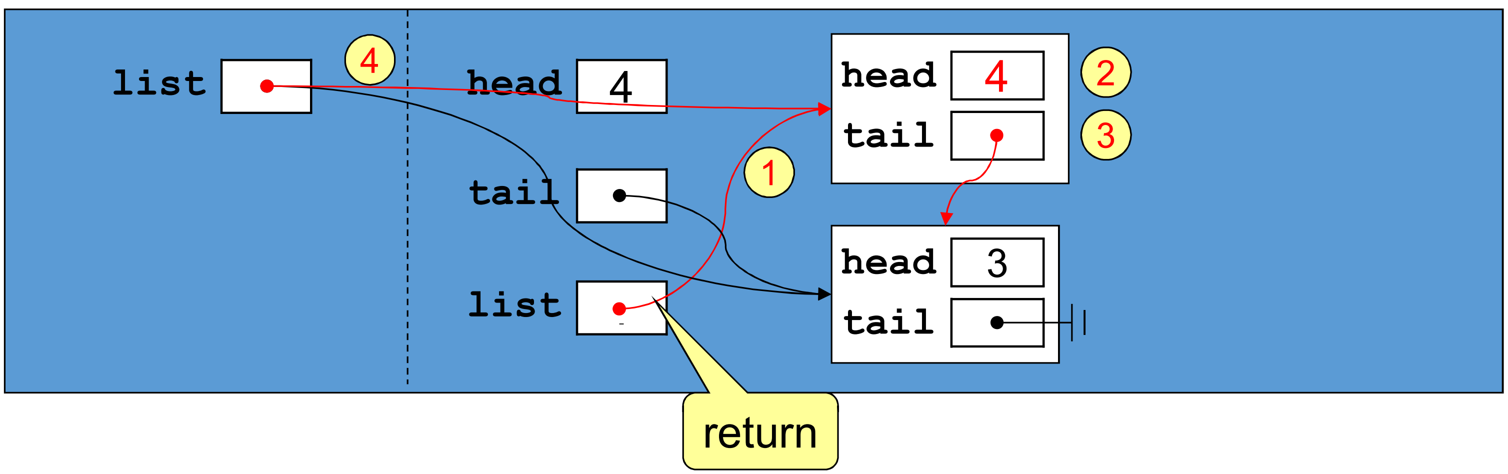


list.c (continue)

```
listADT CreateList(listElementT head, listADT tail) {
   listADT list;

   list = (listADT)malloc(sizeof(struct listCDT));  1
   list->head = head;  2
   list->tail = tail;  3
   return list;  4
}
```

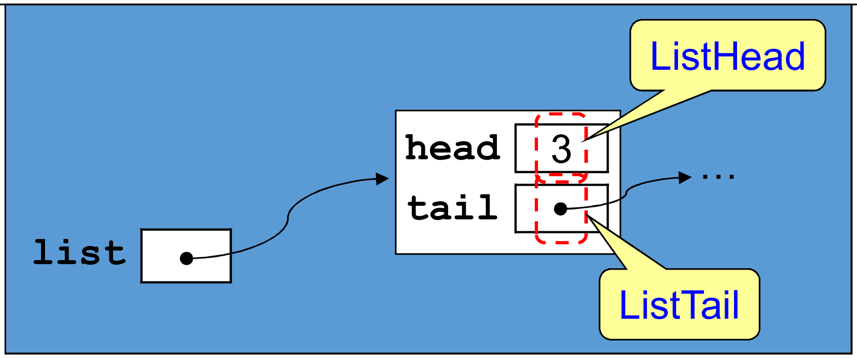
```
listADT list;
list = EmptyList();
list = CreateList(3, list);
list = CreateList(4, list);
```

CreateList(4, [3]) → [4, 3]



`list.c` (continue)

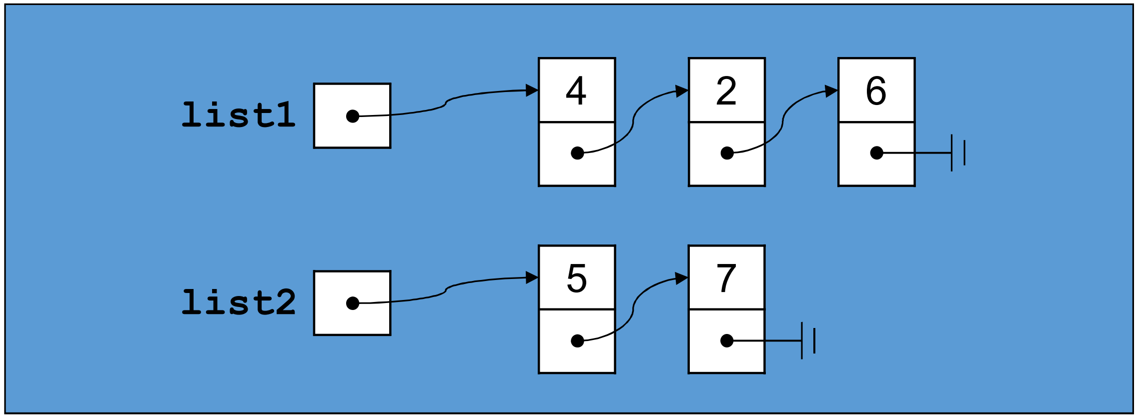
```
listElementT ListHead(listADT list) {
   if (ListIsEmpty(list))
      exit(0);
   return list->head;
}

listADT ListTail(listADT list) {
   if (ListIsEmpty(list))
      exit(0);
   return list->tail;
}

int ListIsEmpty(listADT list) {
   return (list == NULL);
}
```



Internal Sharing

- Recall the example
  - ListConcat([4, 2, 6], [5, 7]).

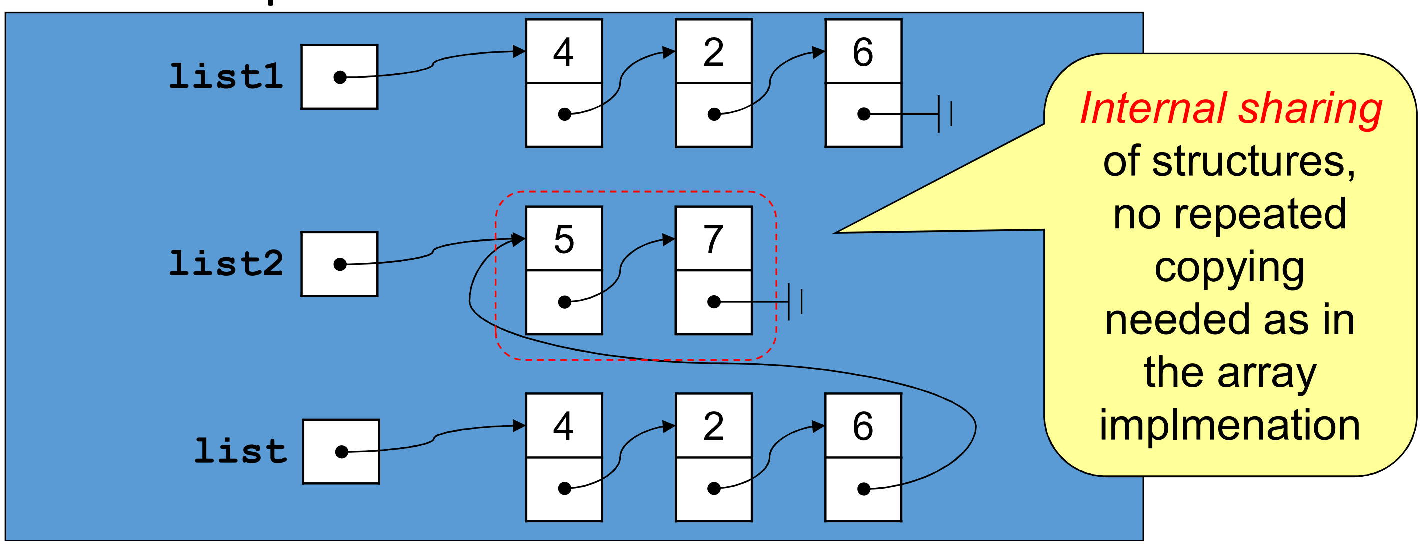
```
listADT list1, list2, list;
list1 = CreateList(4, CreateList(2, CreateList(6, EmptyList())));
list2 = CreateList(5, CreateList(7, EmptyList()));
list = ListConcat(list1, list2);
```



Internal Sharing

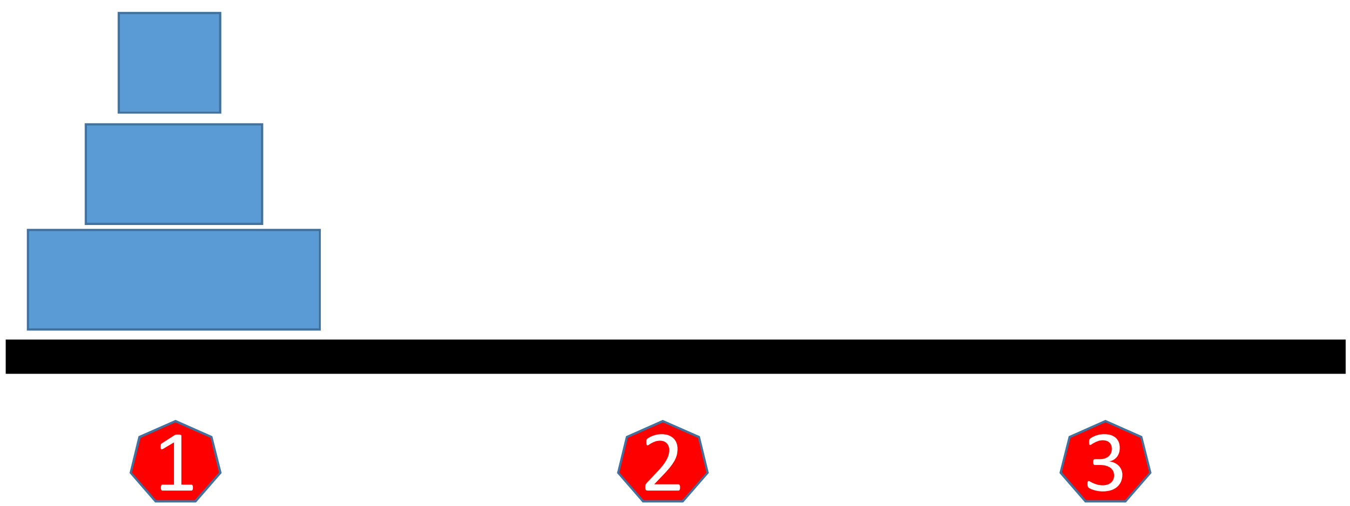
```
listADT list1, list2, list;
list1 = CreateList(4, CreateList(2, CreateList(6, EmptyList())));
list2 = CreateList(5, CreateList(7, EmptyList()));
list = ListConcat(list1, list2);
```

- If you follow the logic of ListConcat carefully, you can see an advantage of this representation.



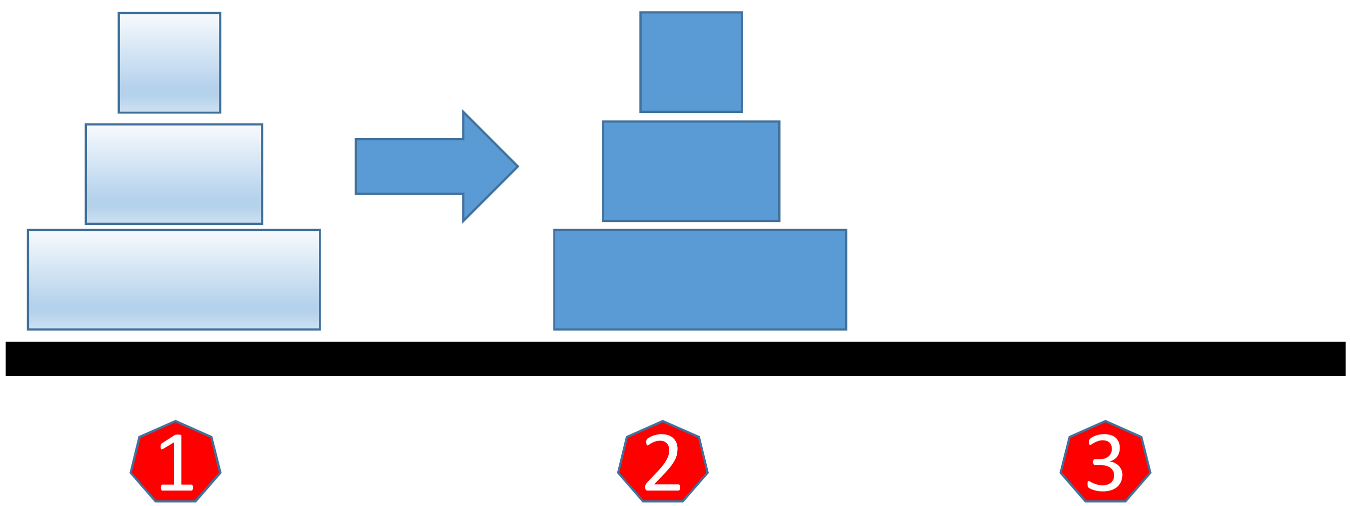
- Recursion is very useful in many applications.
- It helps us to solve many problems.
- An example is the tower of Hanoi.

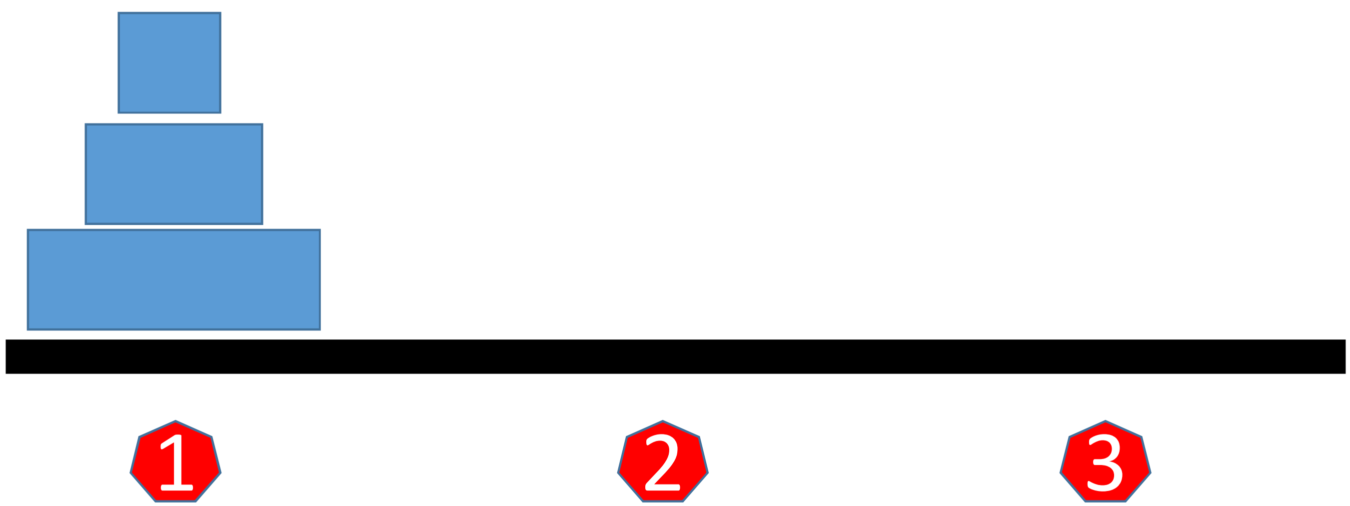
Task : move the blocks from the source position 1 to 2

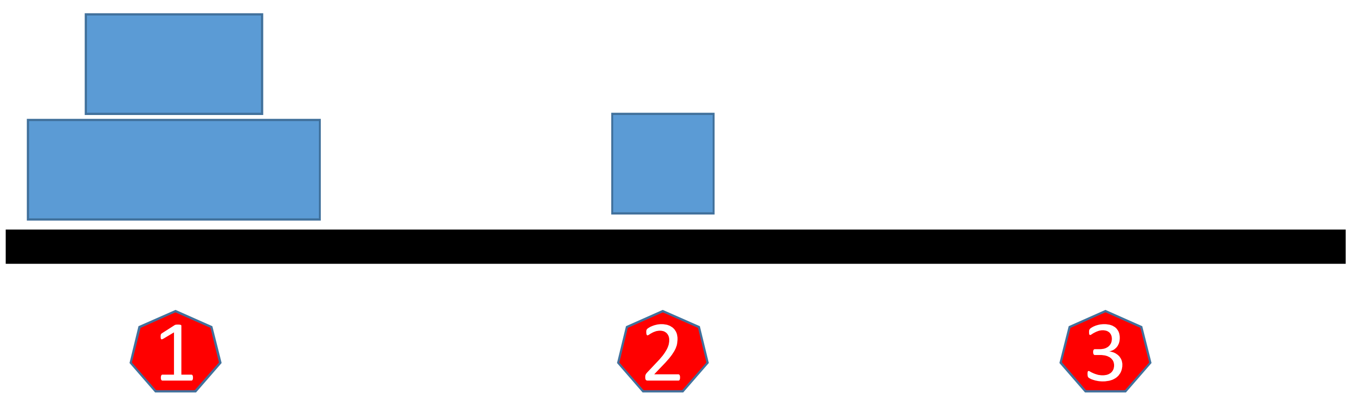


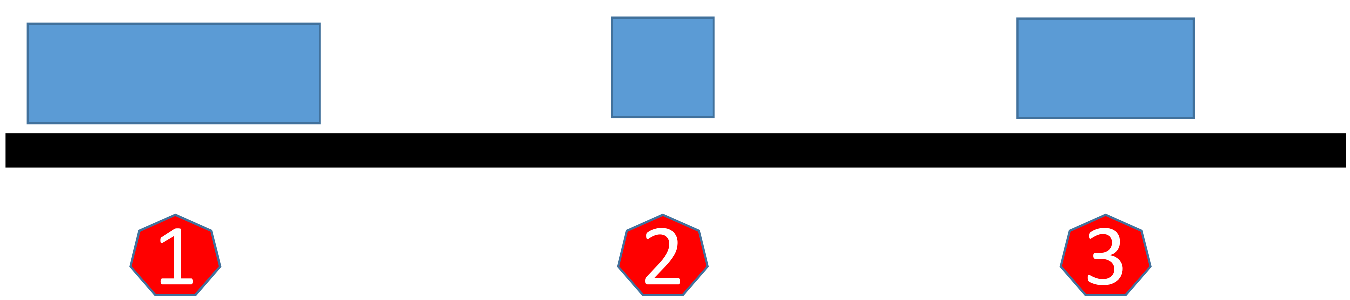
Task : move the blocks from the source position 1 to 2

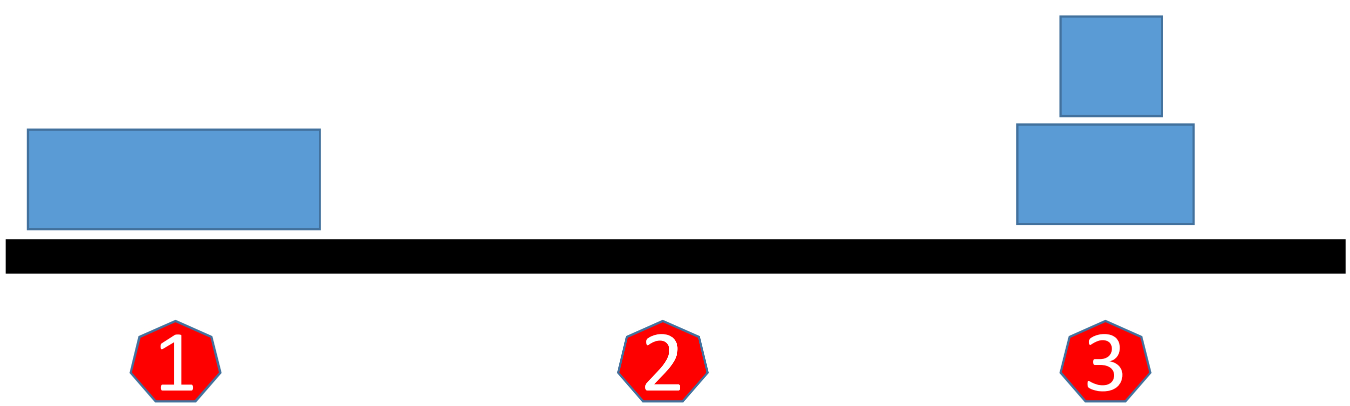
- Only 1 block is allowed to move at each step
- Large block cannot be placed on top of the smaller ones.

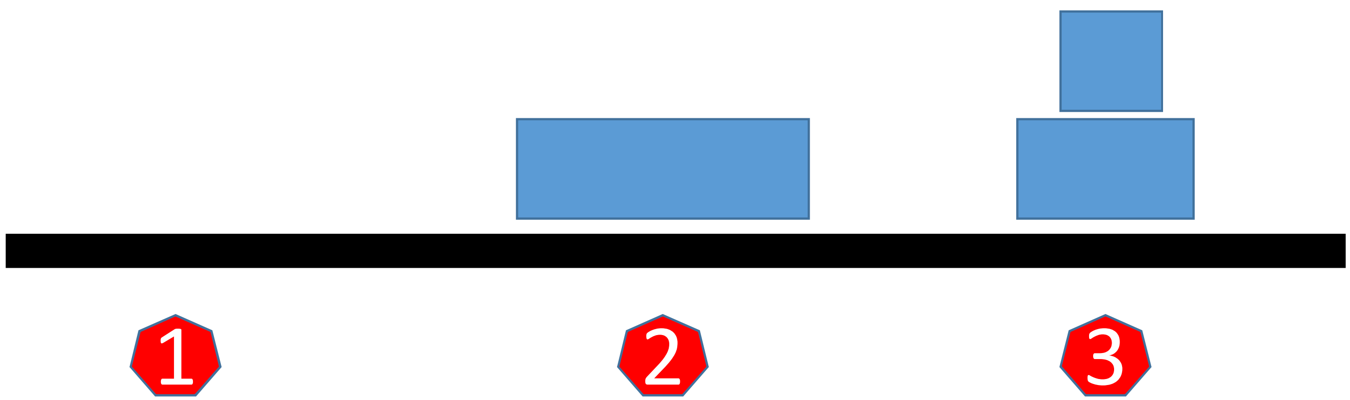


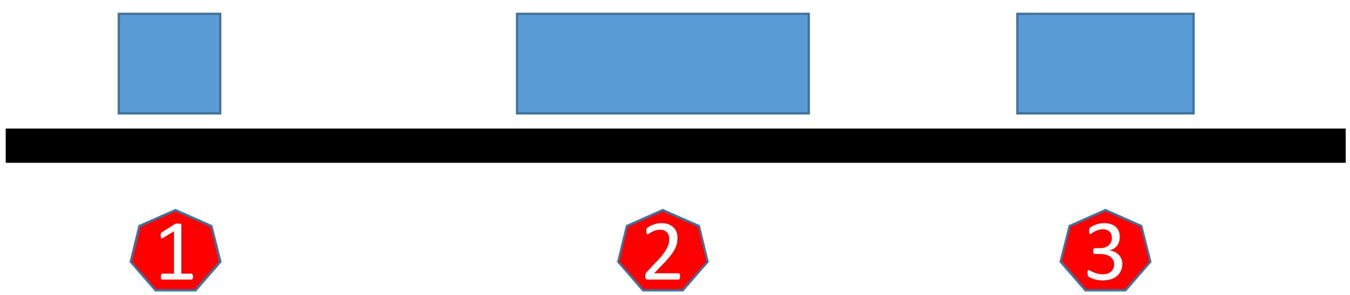

1
2
3

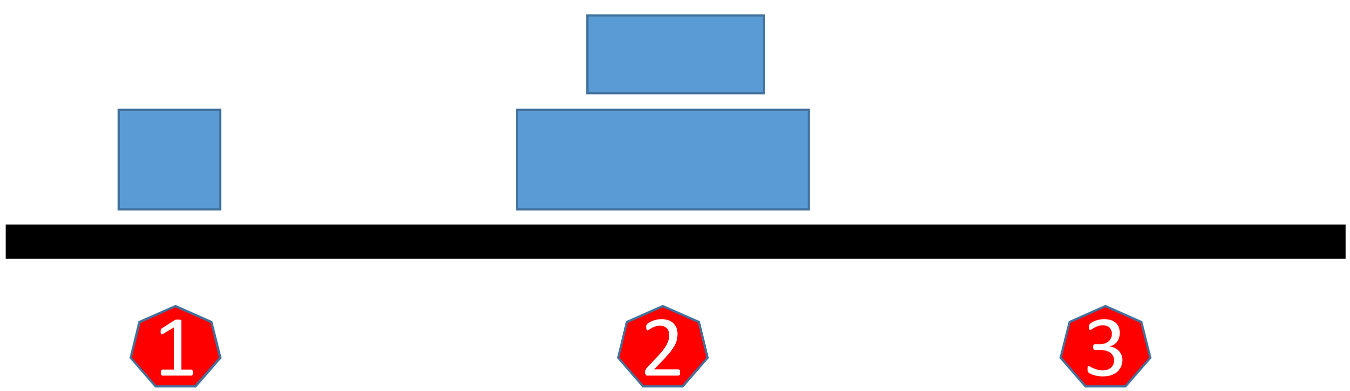

1
2
3

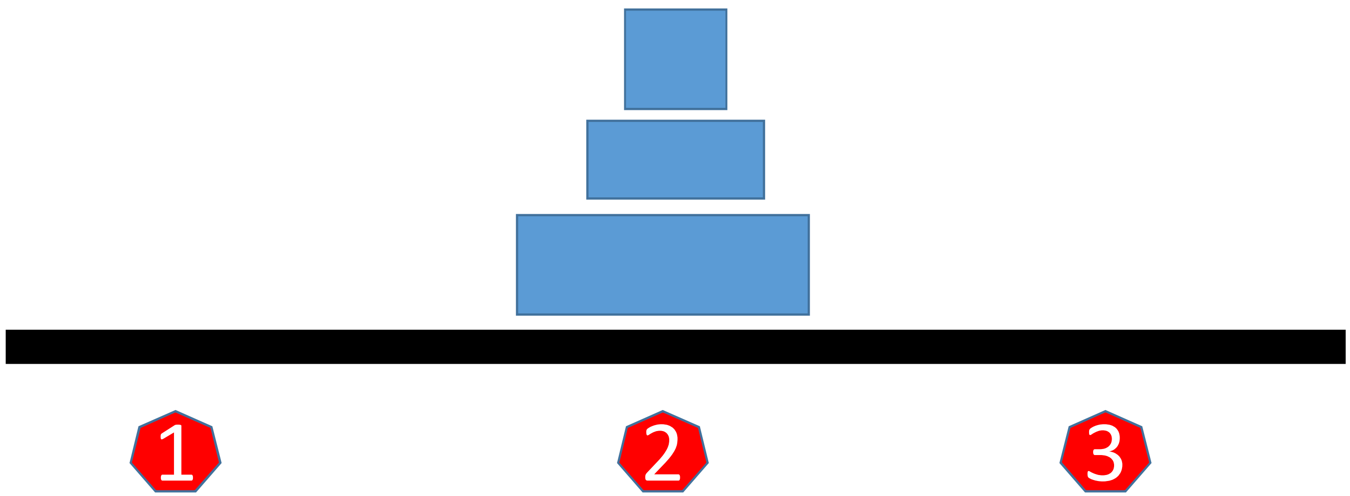

1
2
3


1
2
3

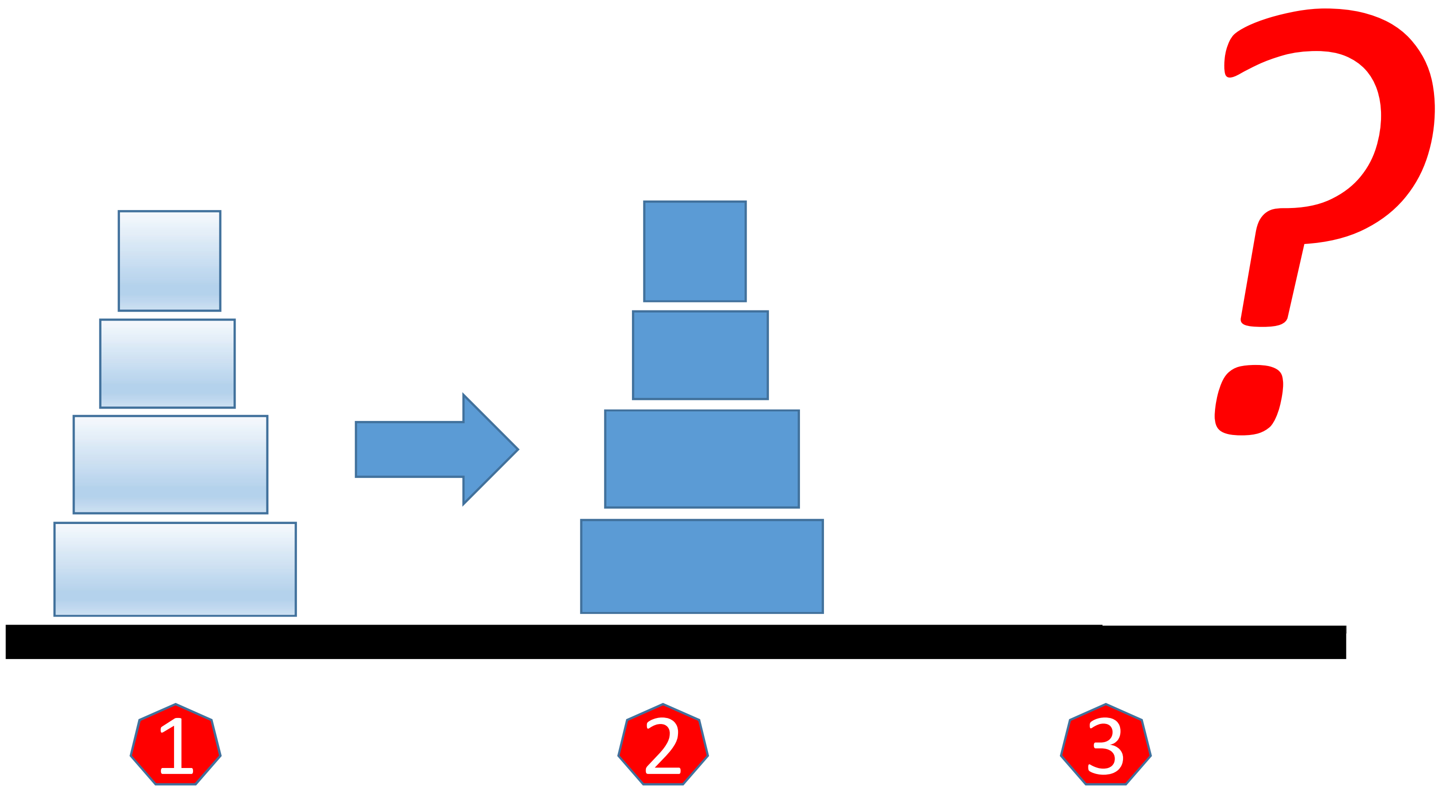

1
2
3

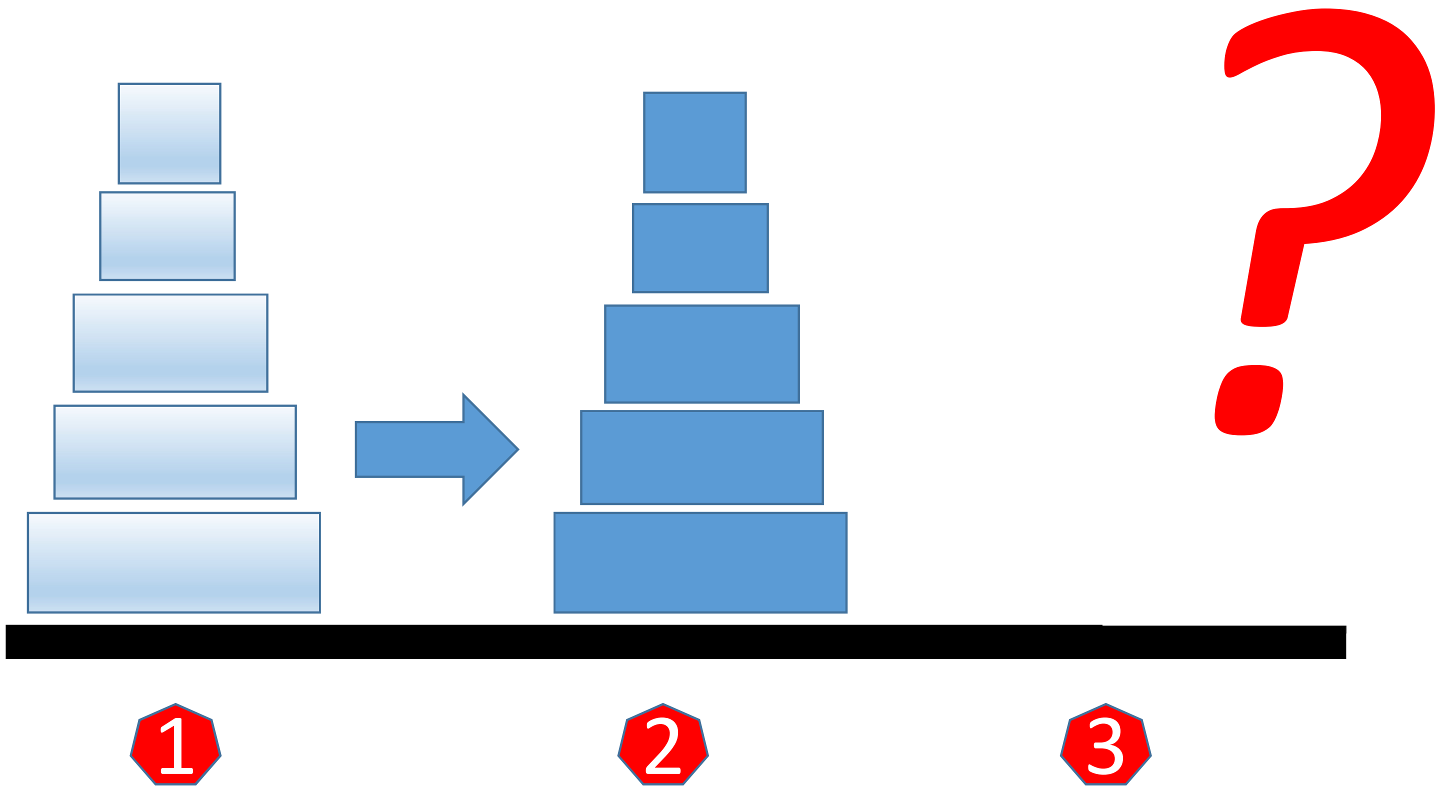

1
2
3


1
2
3

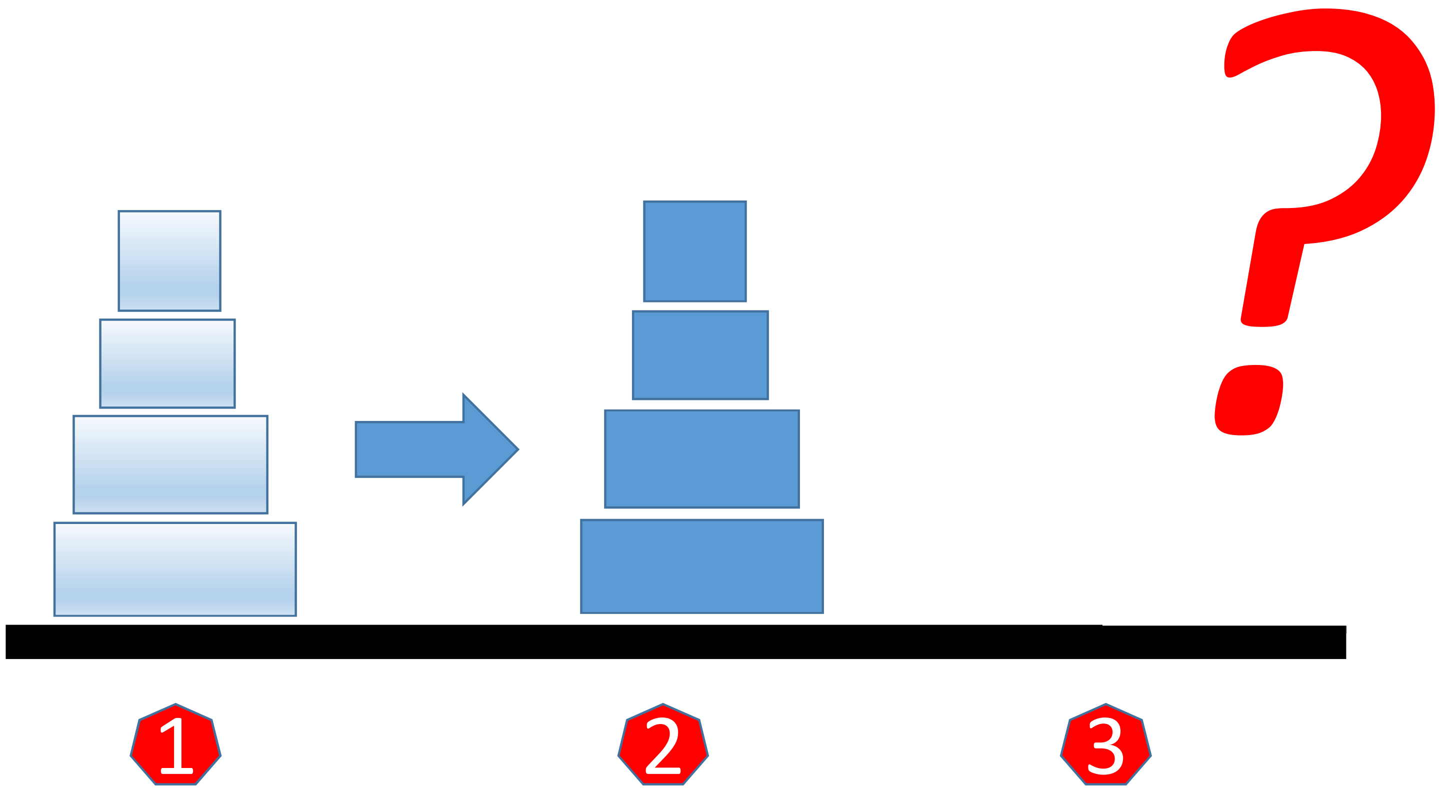

1
2
3

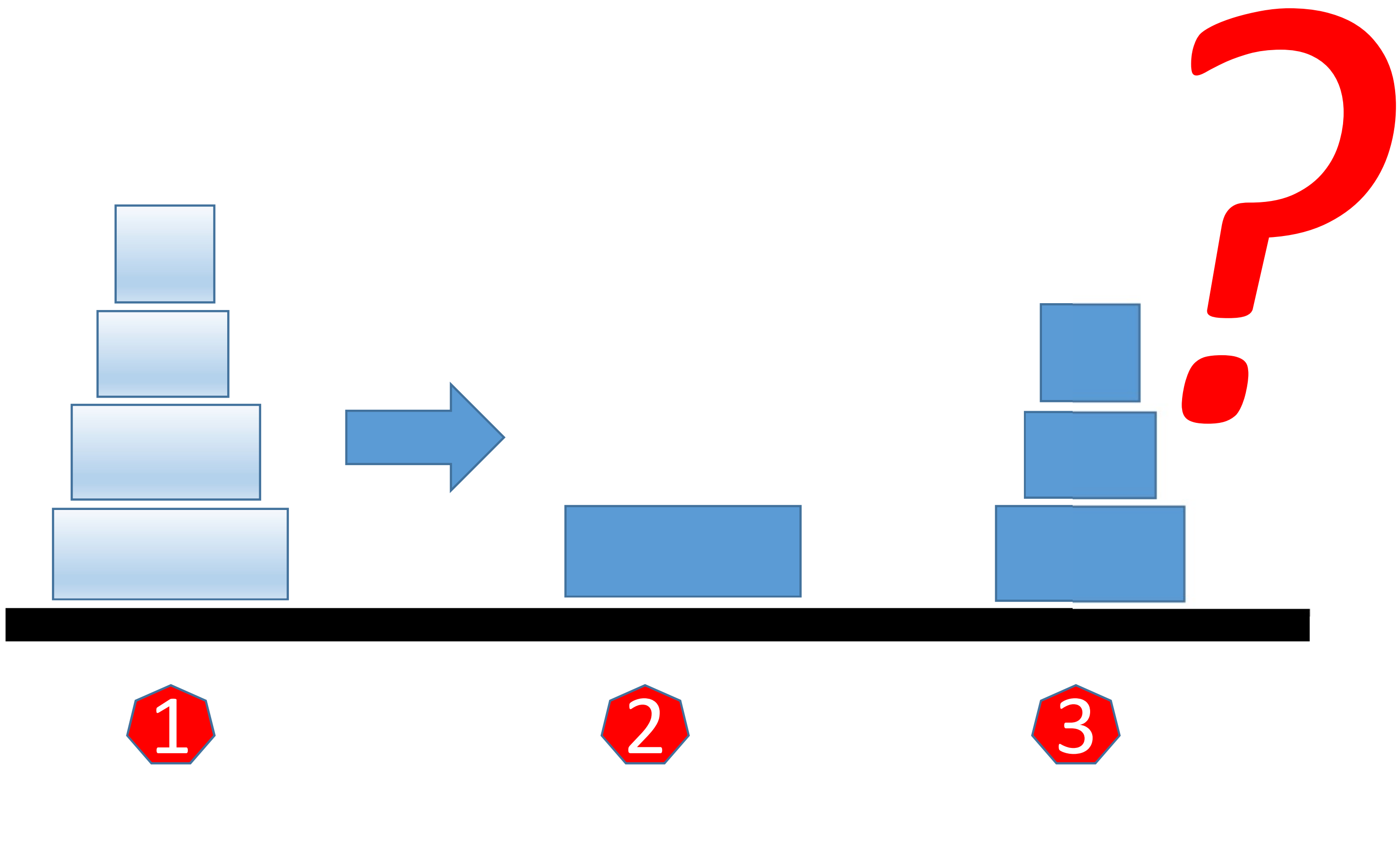
- It is not difficult to move the tower when there are only 3 blocks.
- But how to move the tower with n blocks ?

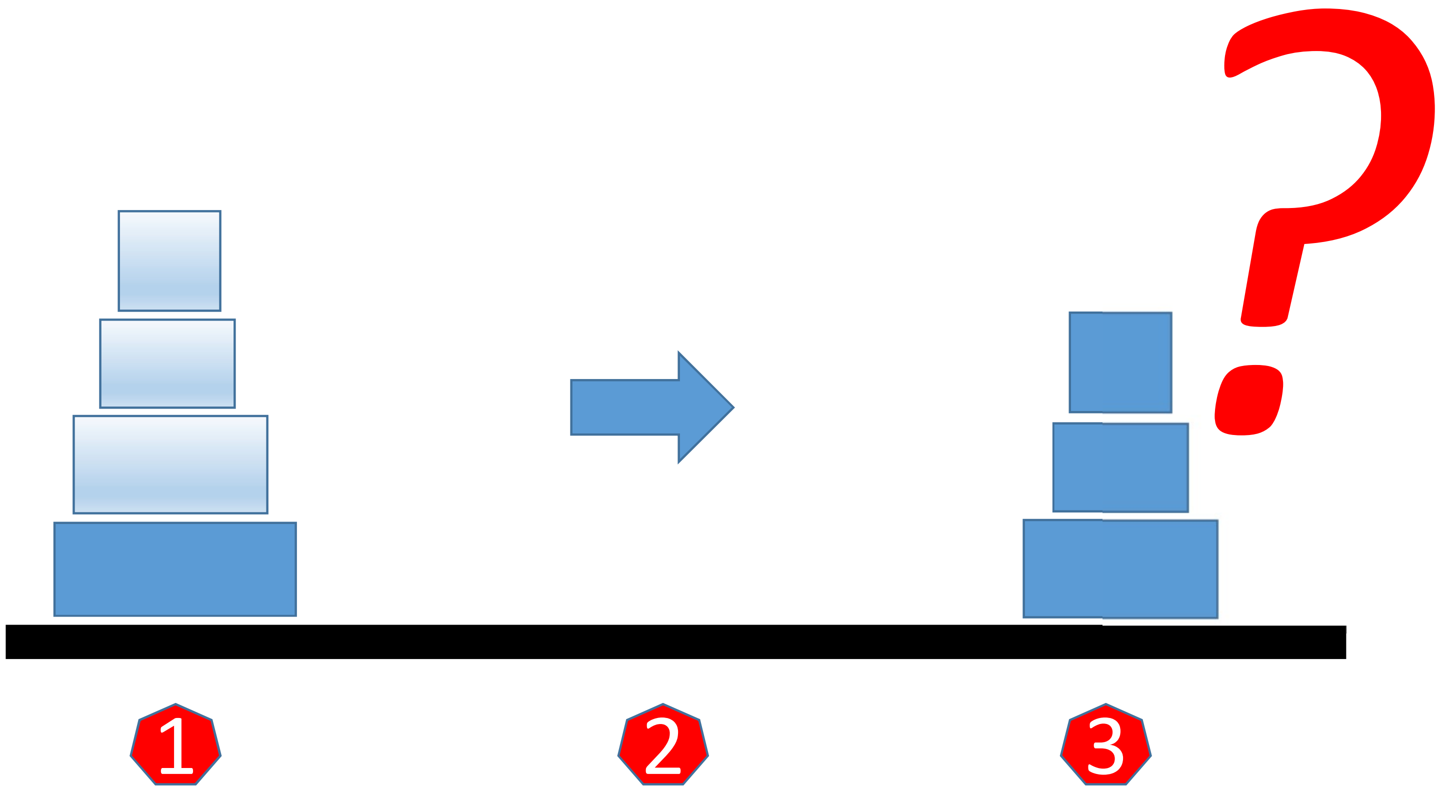

1
2
3

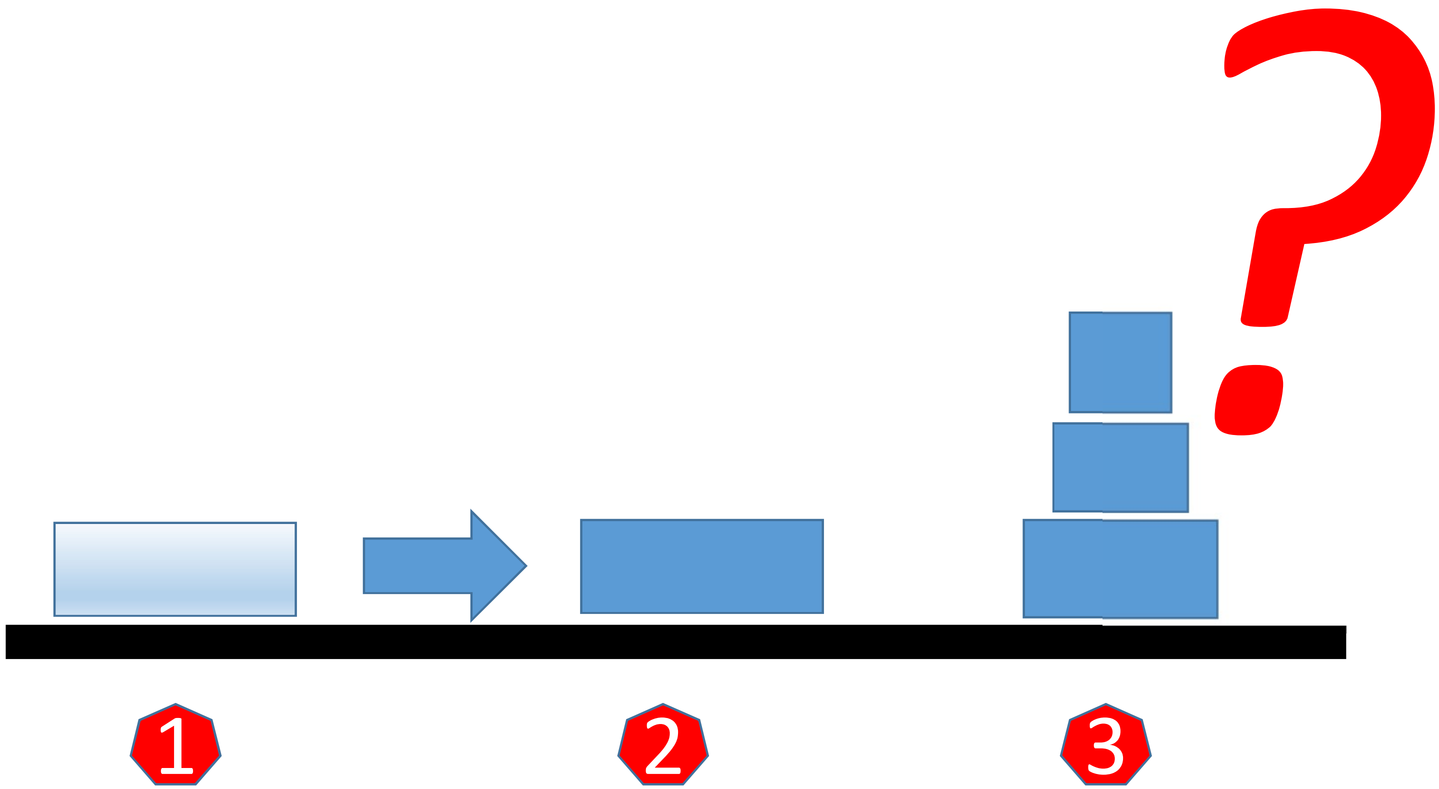


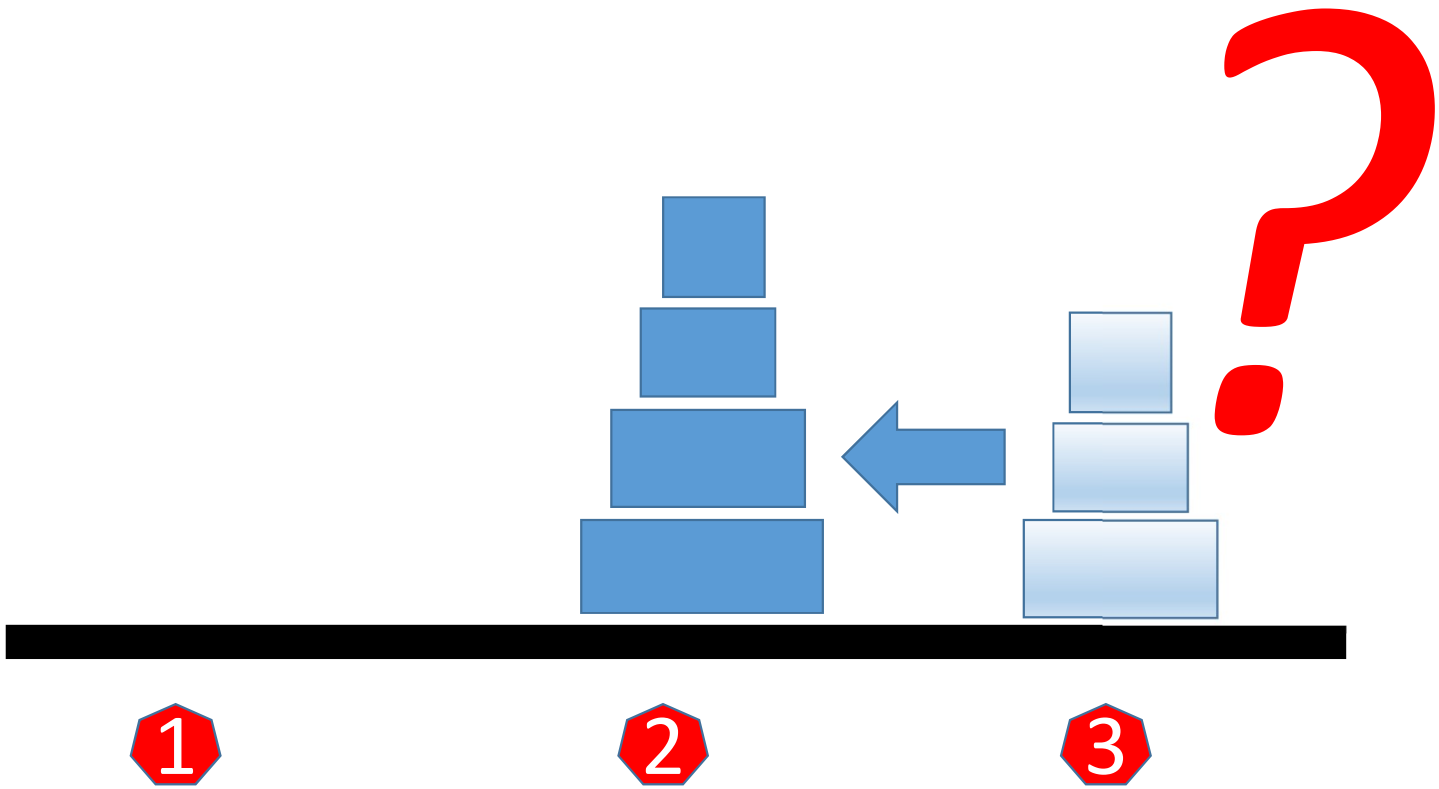
?
1
2
3

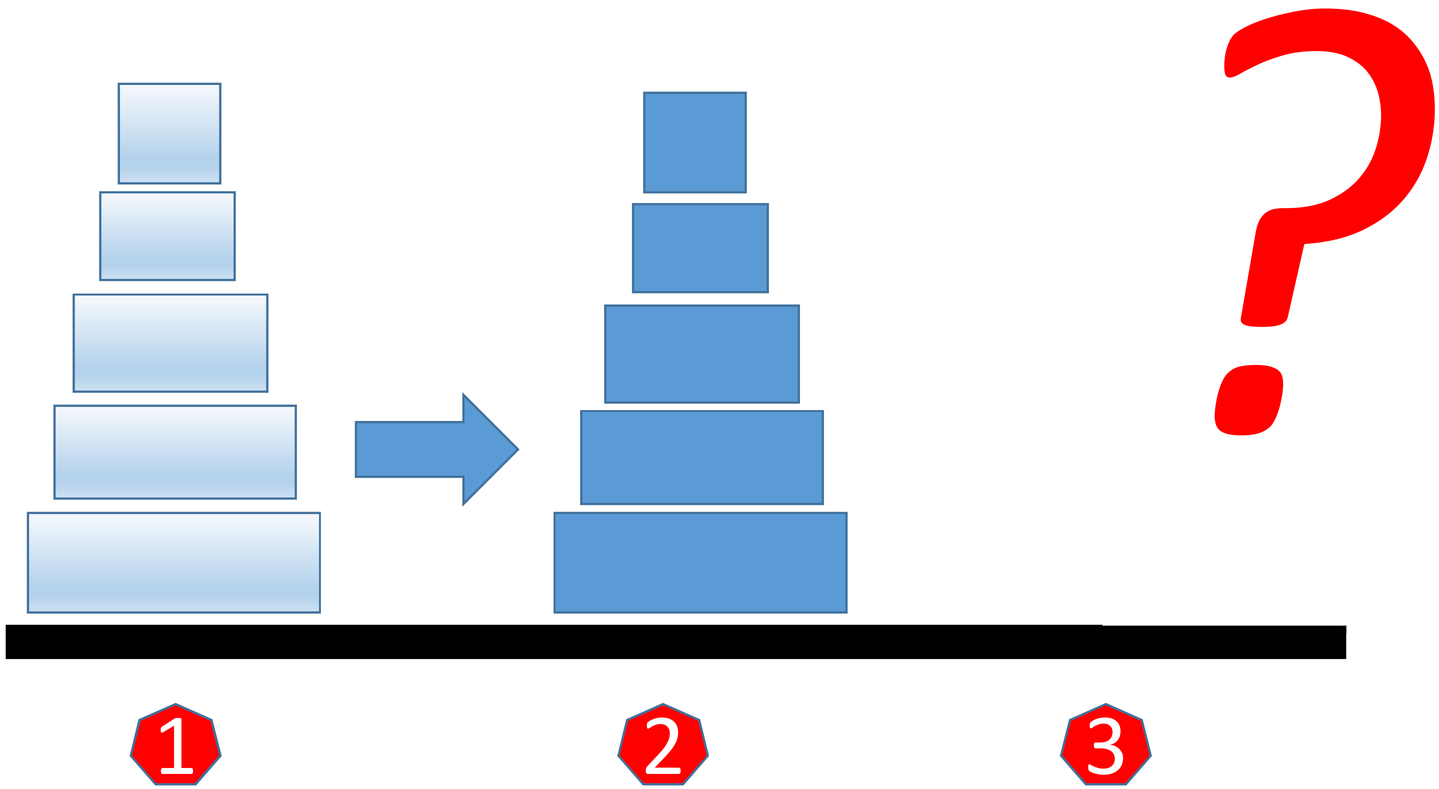

?
1
2
3

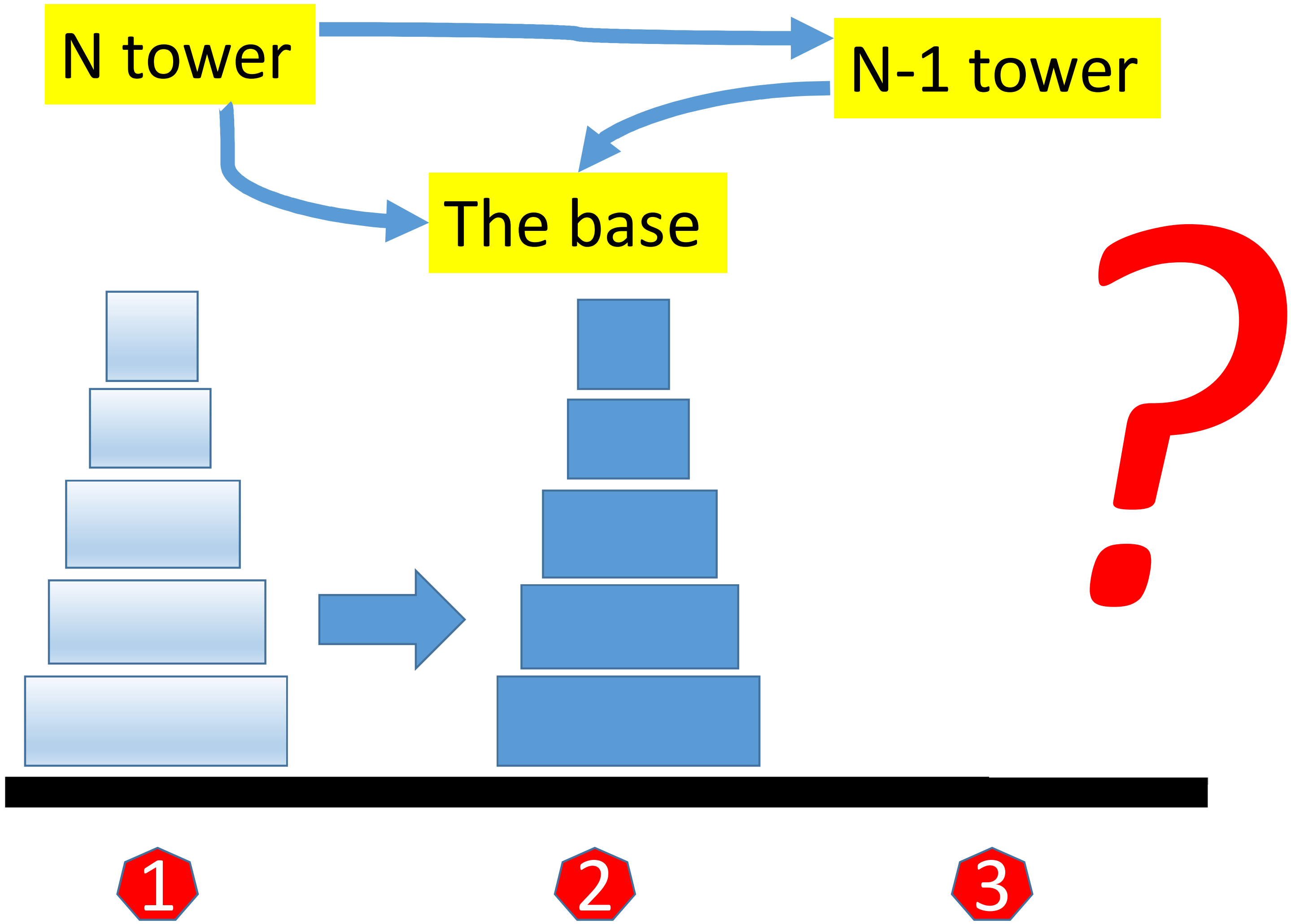

1
2
3
?


1
2
3


1
2
3
?


1
2
3
?


?
1
2
3


N tower
N-1 tower
The base
?
1
2
3