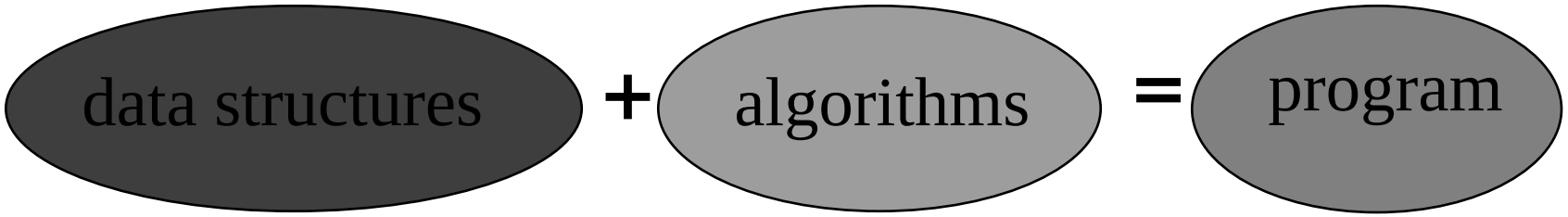
Data Structures & Algorithms
Lists Processing

- In this lecture we will be :
  - looking at data structures and algorithms
  - taking a closer look at the implementation of the List
  - examining some of the algorithms that are used to manipulate such a data structure.

Data Structures &Algorithms



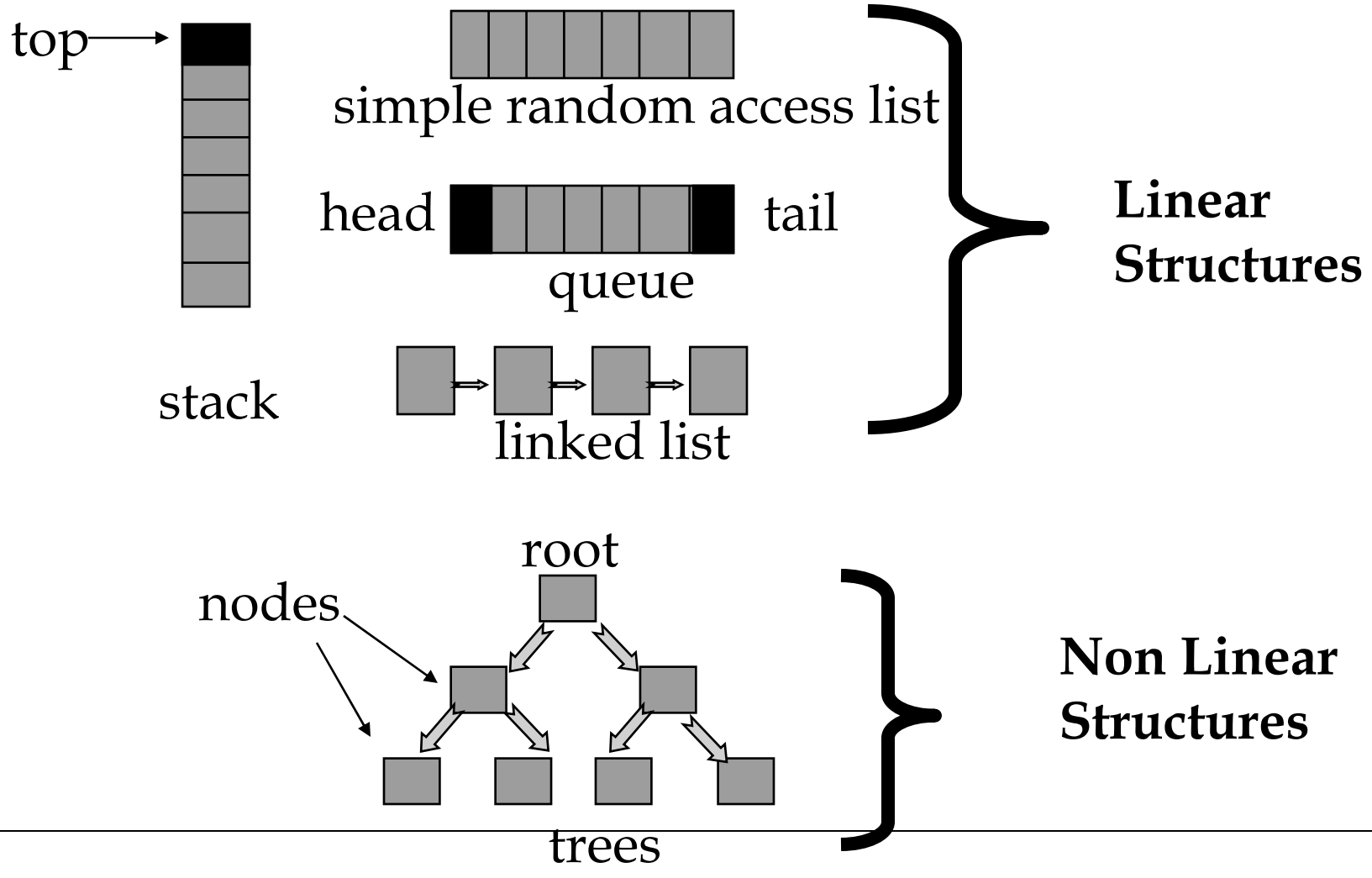
Data Structures

- data is organised for computer processing
- static or dynamic
- numerous types:
  - nature of each item
  - number of items to be processed
  - access of an item
  - computer resources

Data Structures

- **simple list**
- **linked list**
- **stacks**
- **queues**
- **binary trees**

Data Structures



Algorithms

- **procedures for manipulating data**
- **a finite set of instructions:**
  - input (zero or more)
  - output (zero or more)
  - unambiguous
  - finite
  - effective

Algorithms

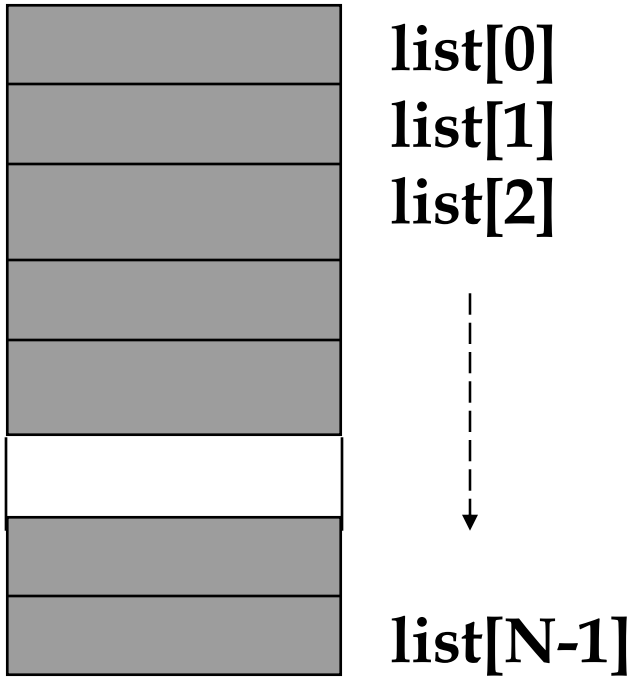
- **most important aspects:**
  - **correctness**
  - **finiteness**
  - **effectiveness**
- **Complexity**
  - Complexity can be taken to be the time or the number of steps of computation taken to process a set of data (ASYMPTOTIC ALGORITHM ANALYSIS)

Algorithms

- When analysing an algorithm, normally 3 situations are taken into account:
    - **best case**
    - **worst case**
    - **average case**

Lists

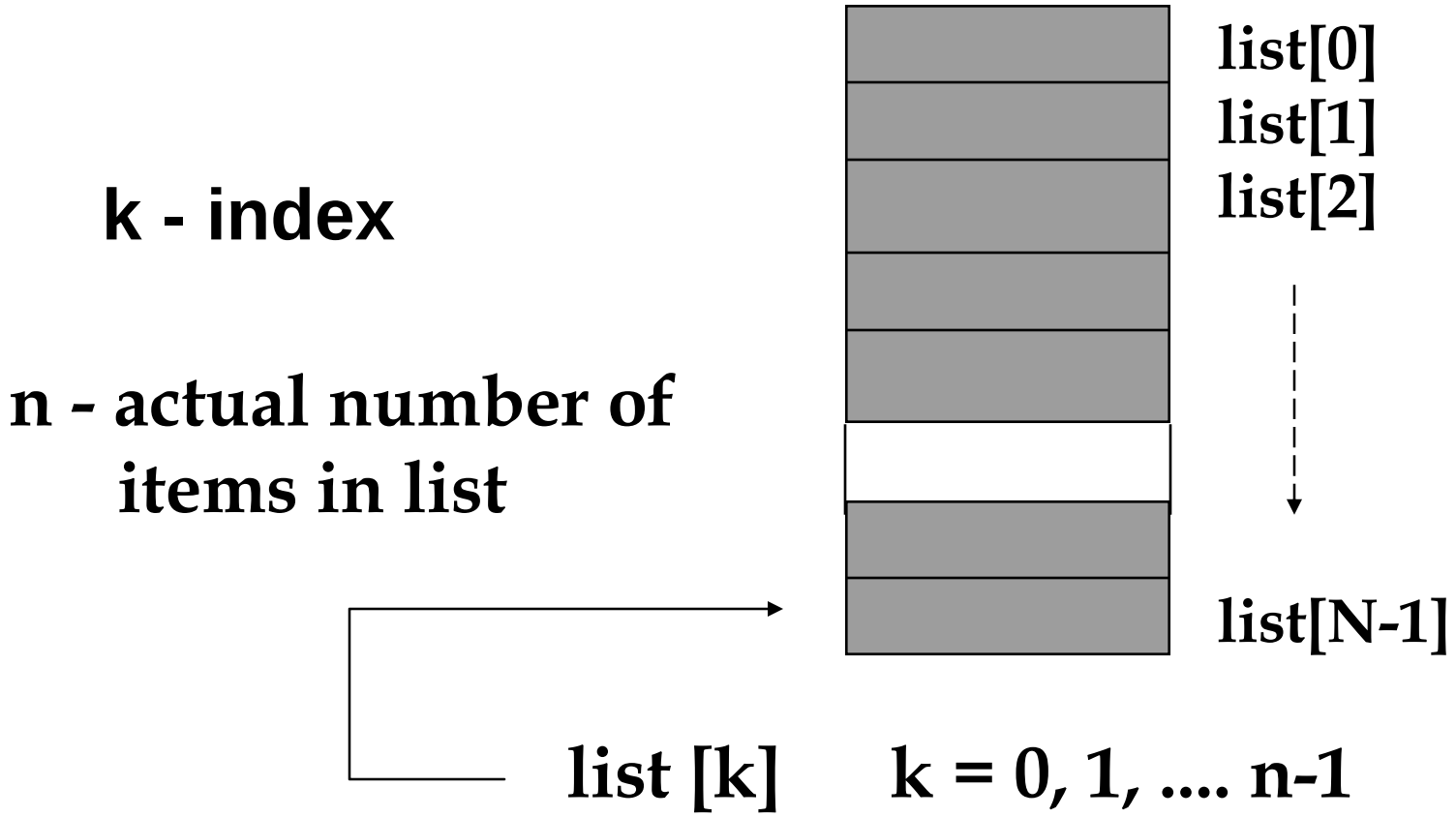
- **random-access list data structure**



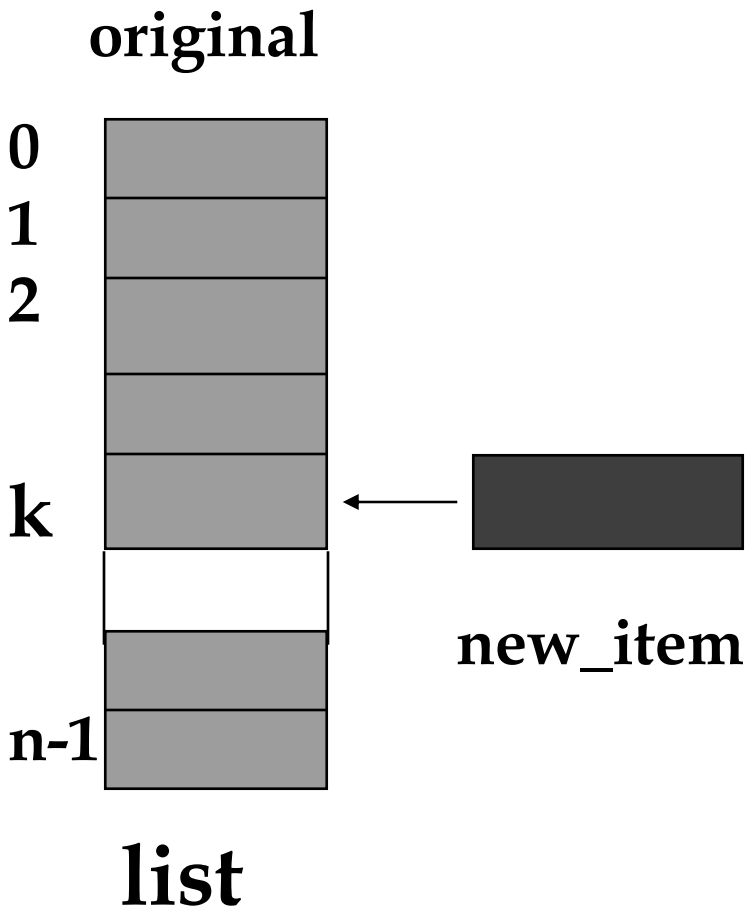
List - Fundamental Operations

- access
- replace
- append
- insert
- delete
- search
- sort

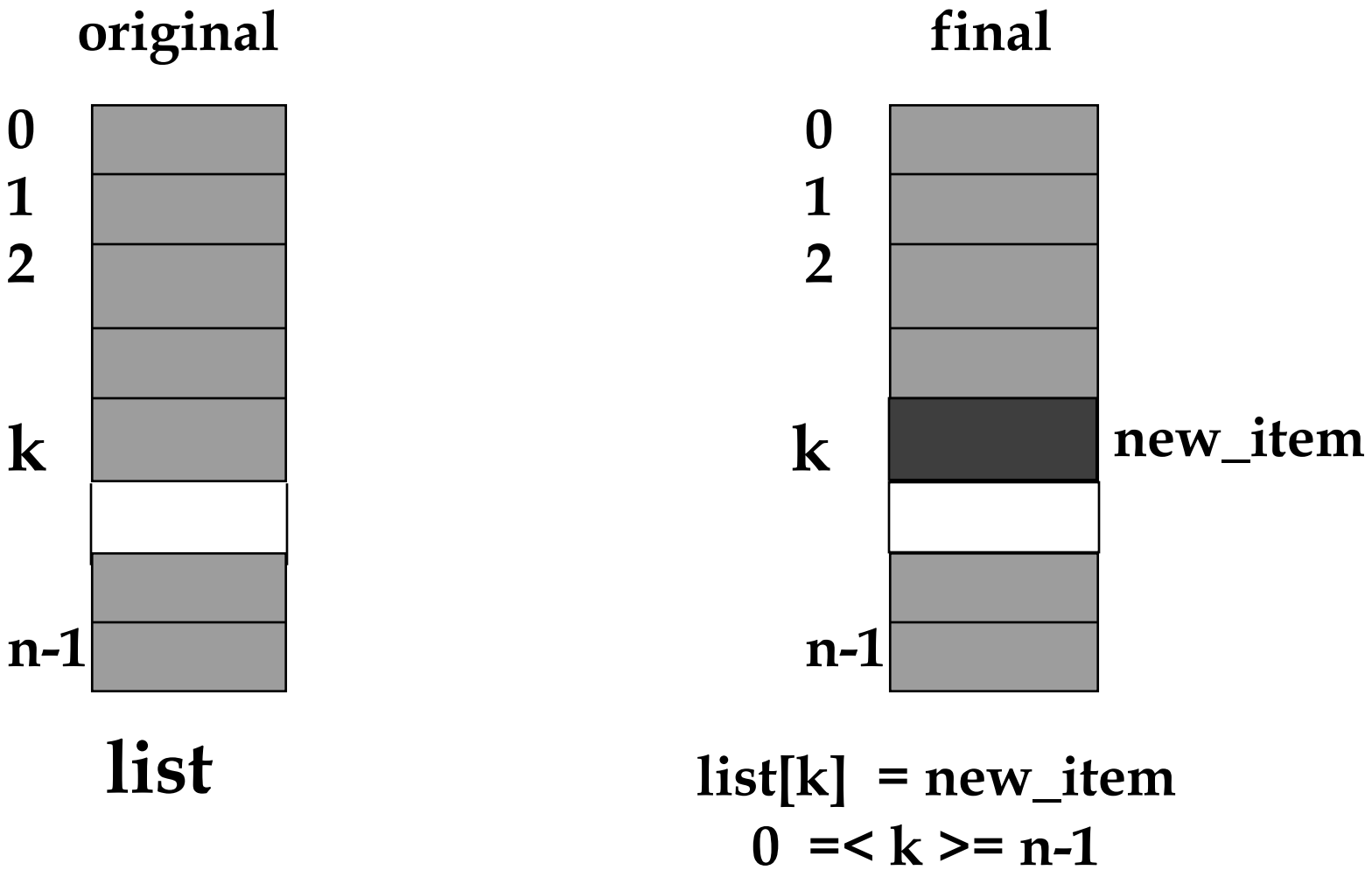
Random Access Lists - Access



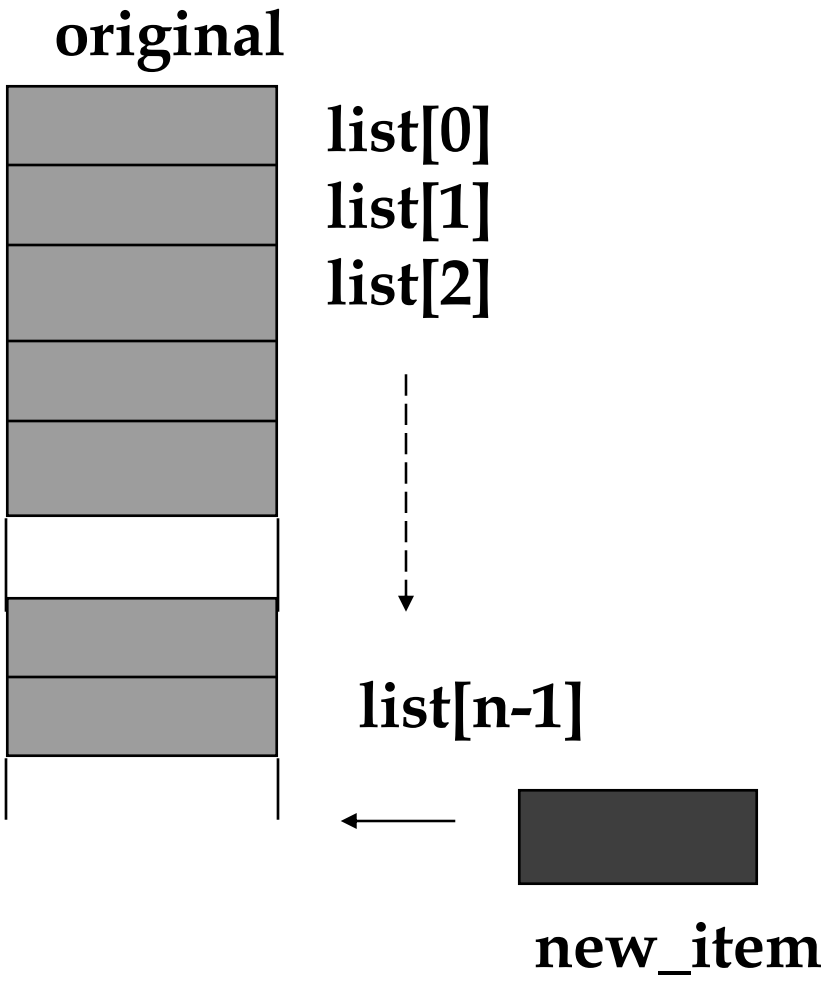
Random Access Lists - Replace



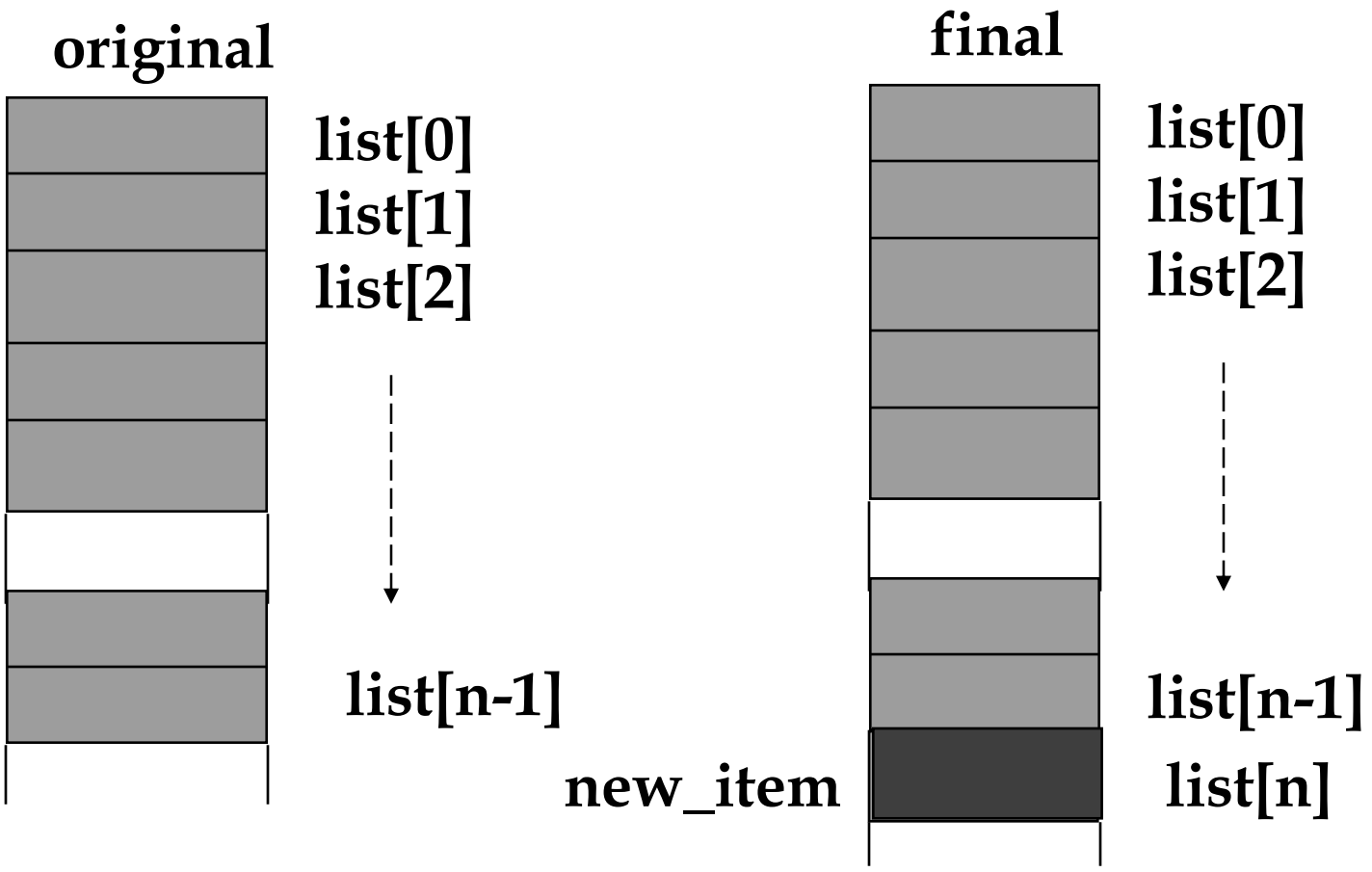
Random Access Lists - Replace ***



Random Access Lists - Append

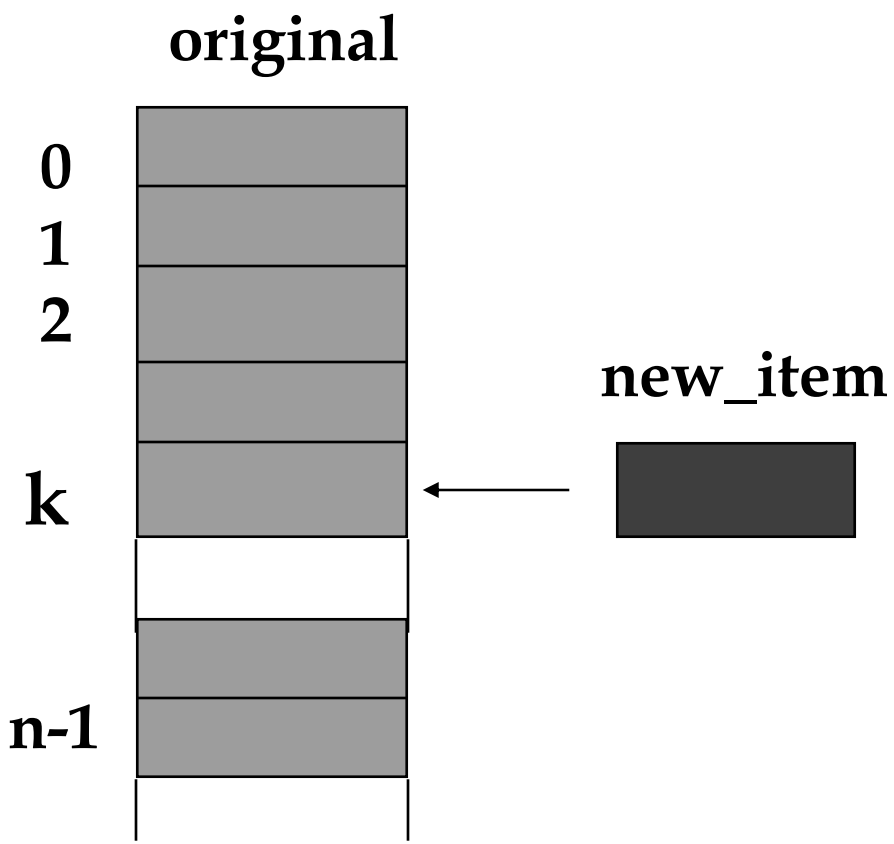


Random Access Lists - Append

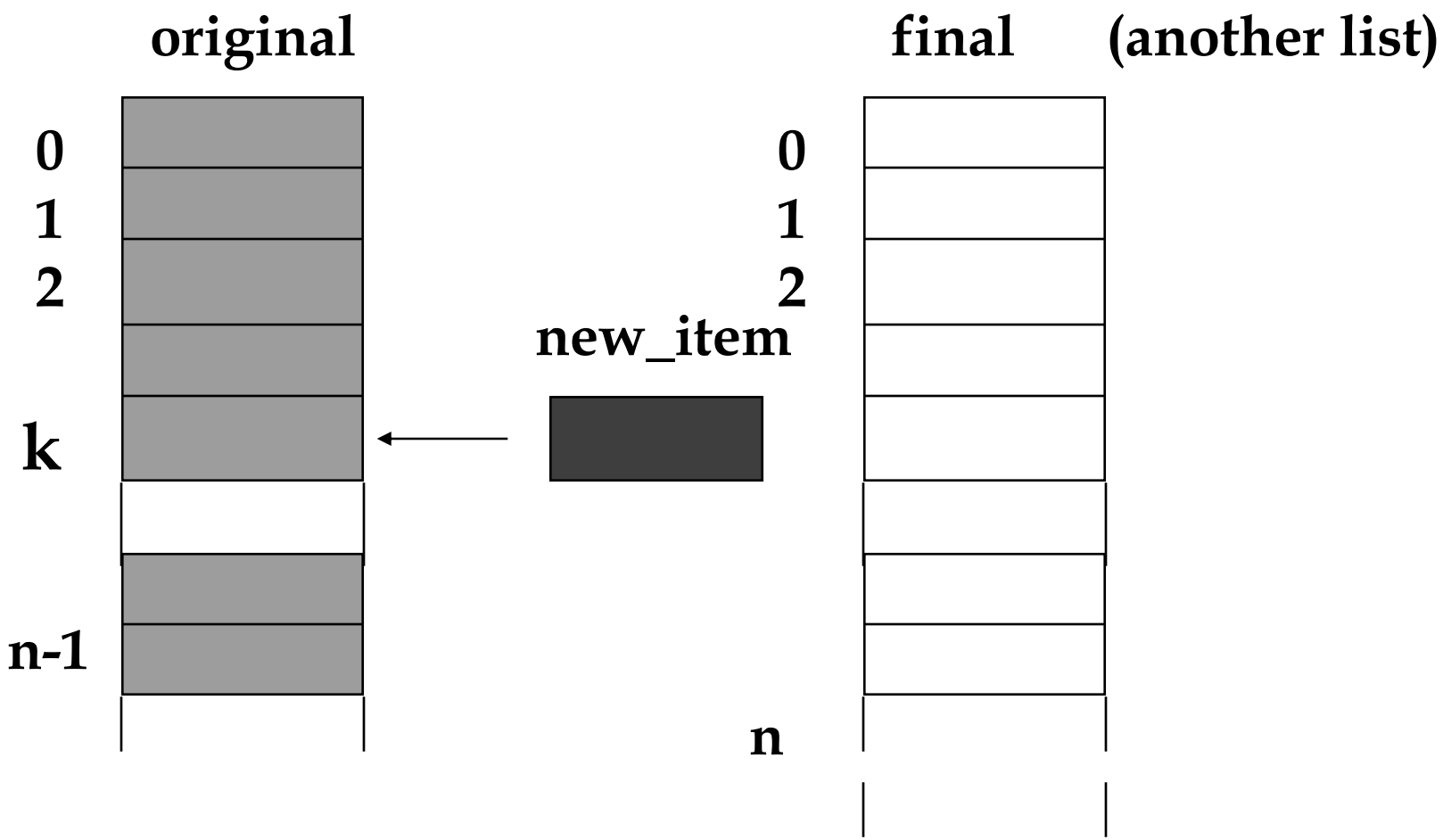


list[n] = new_item; n++;

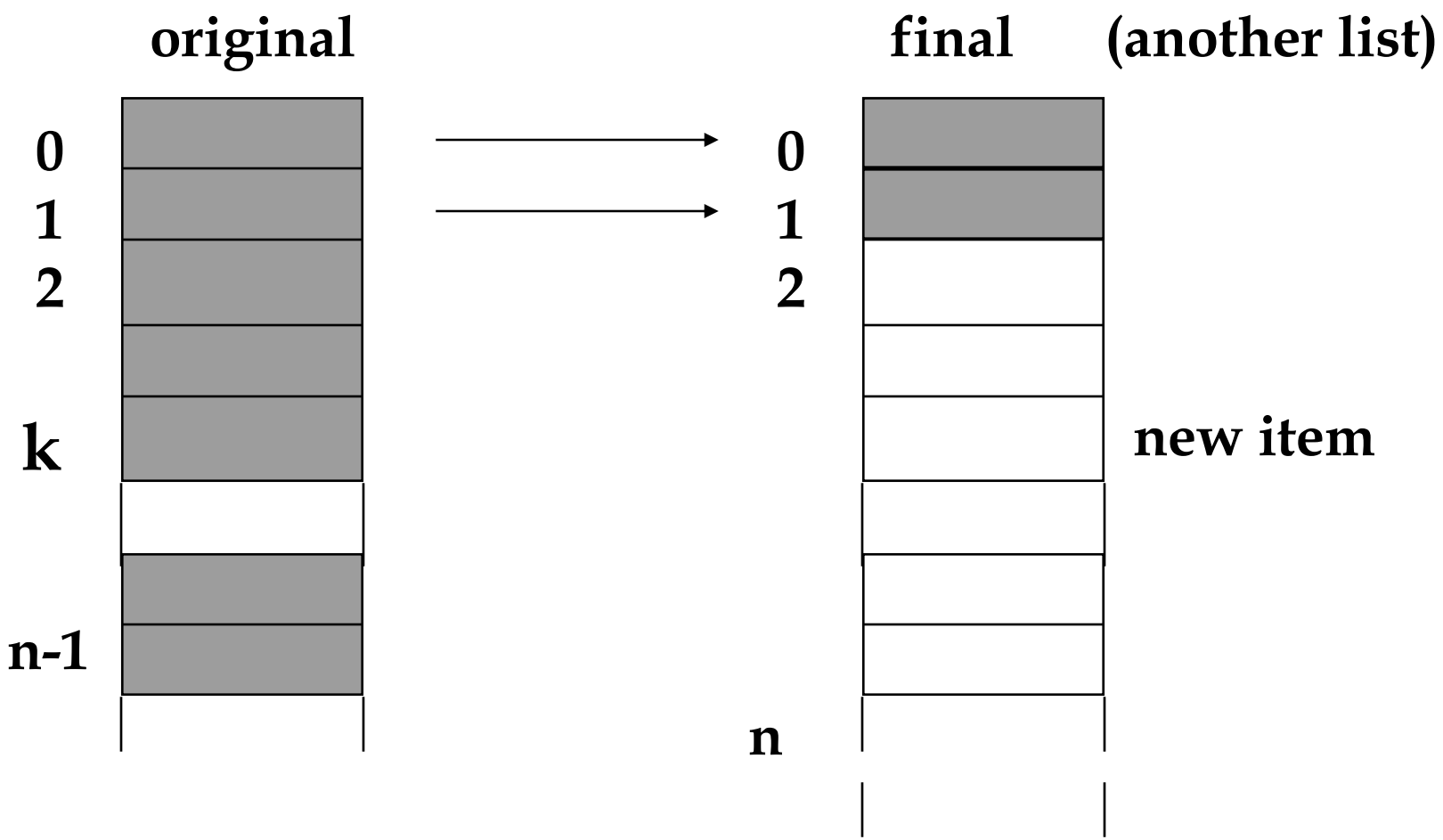
Random Access Lists - insert



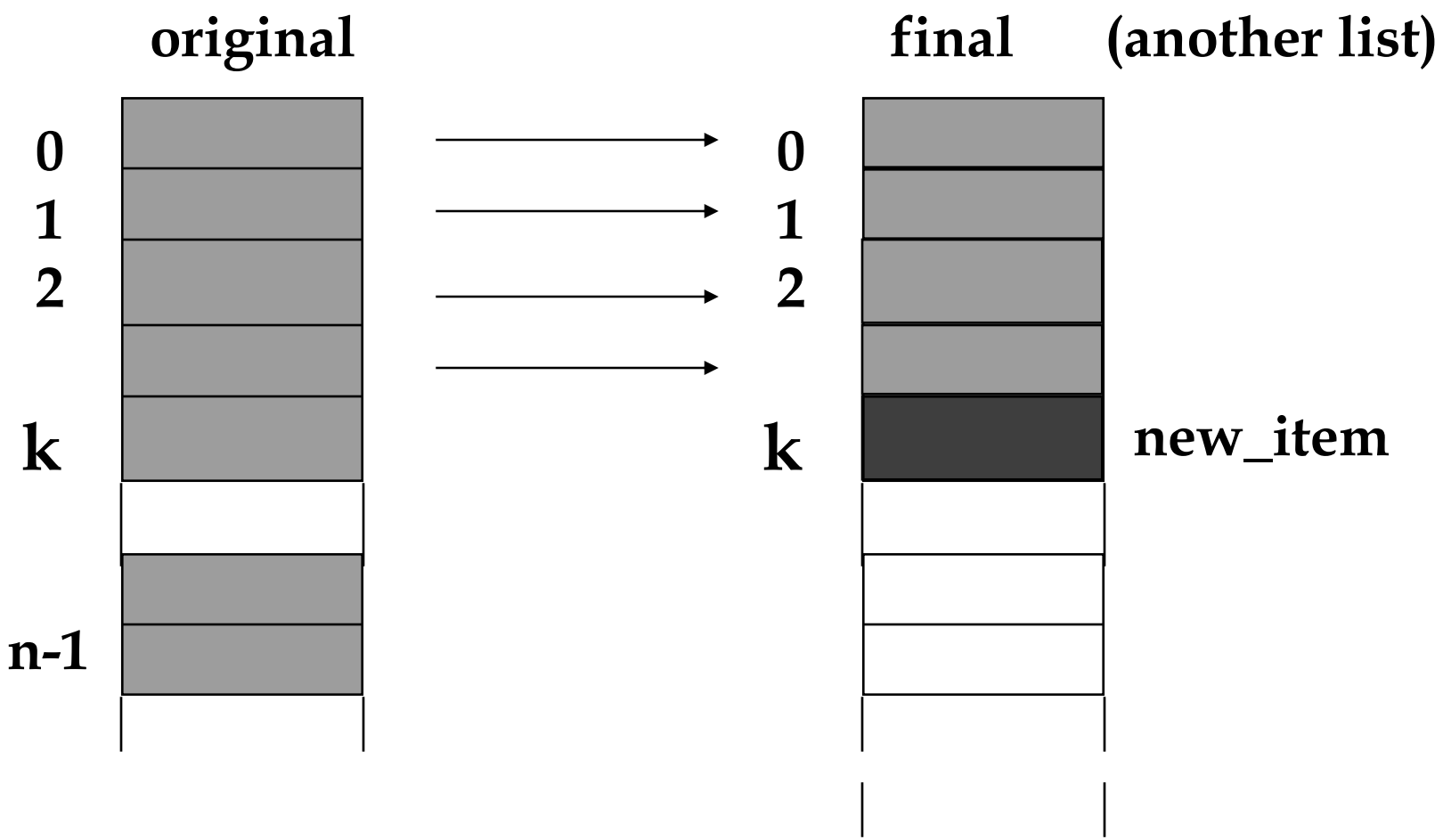
Random Access Lists - insert



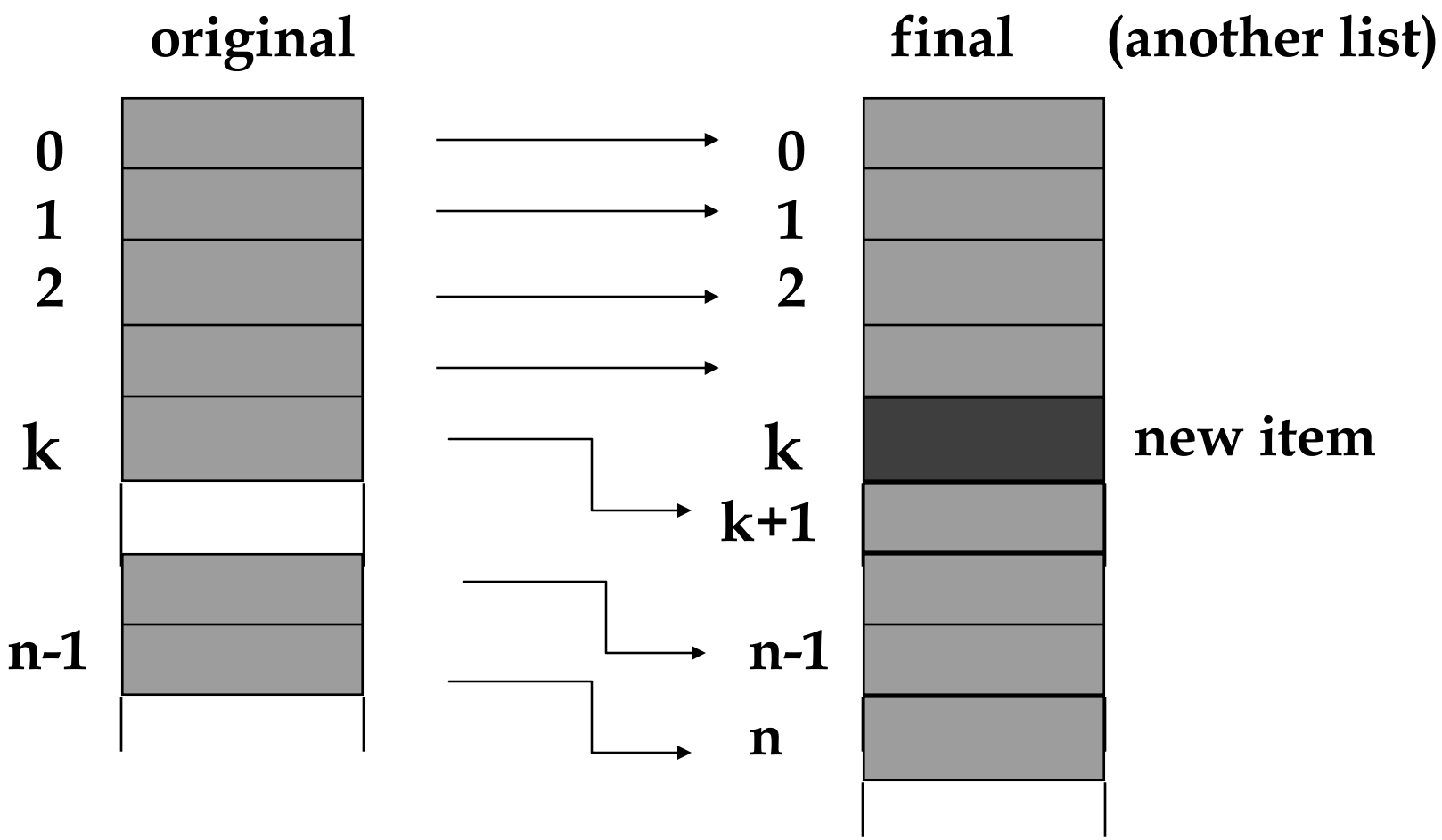
Random Access Lists - insert



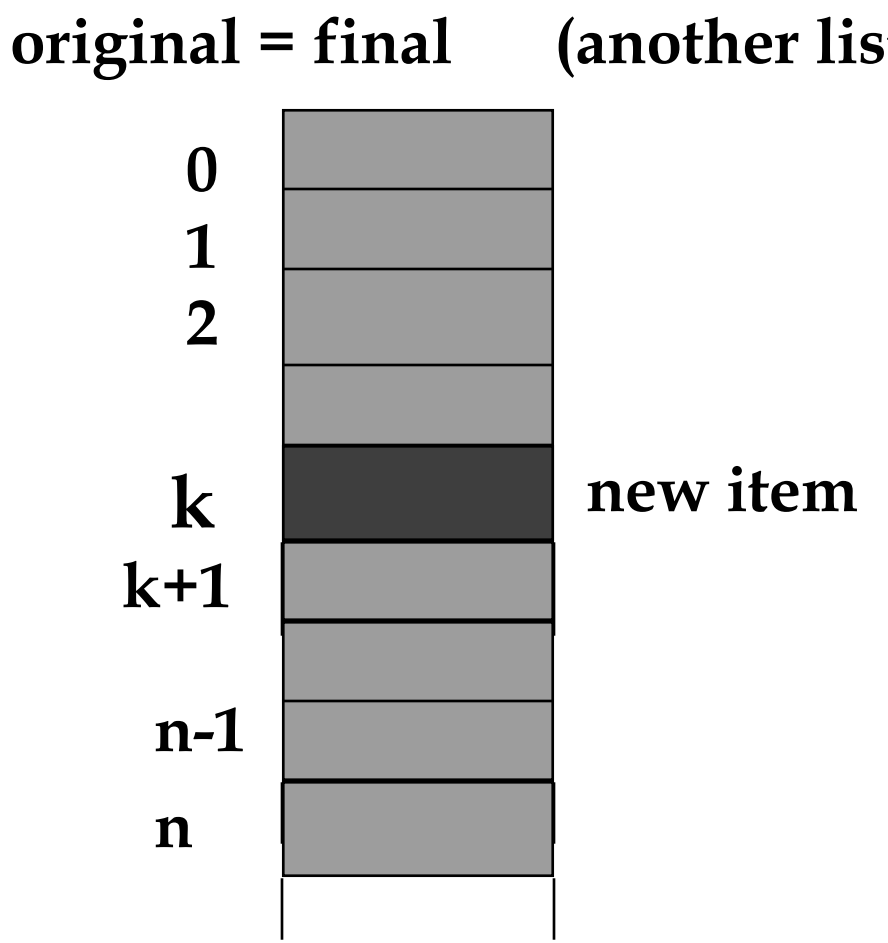
Random Access Lists - insert



Random Access Lists - insert

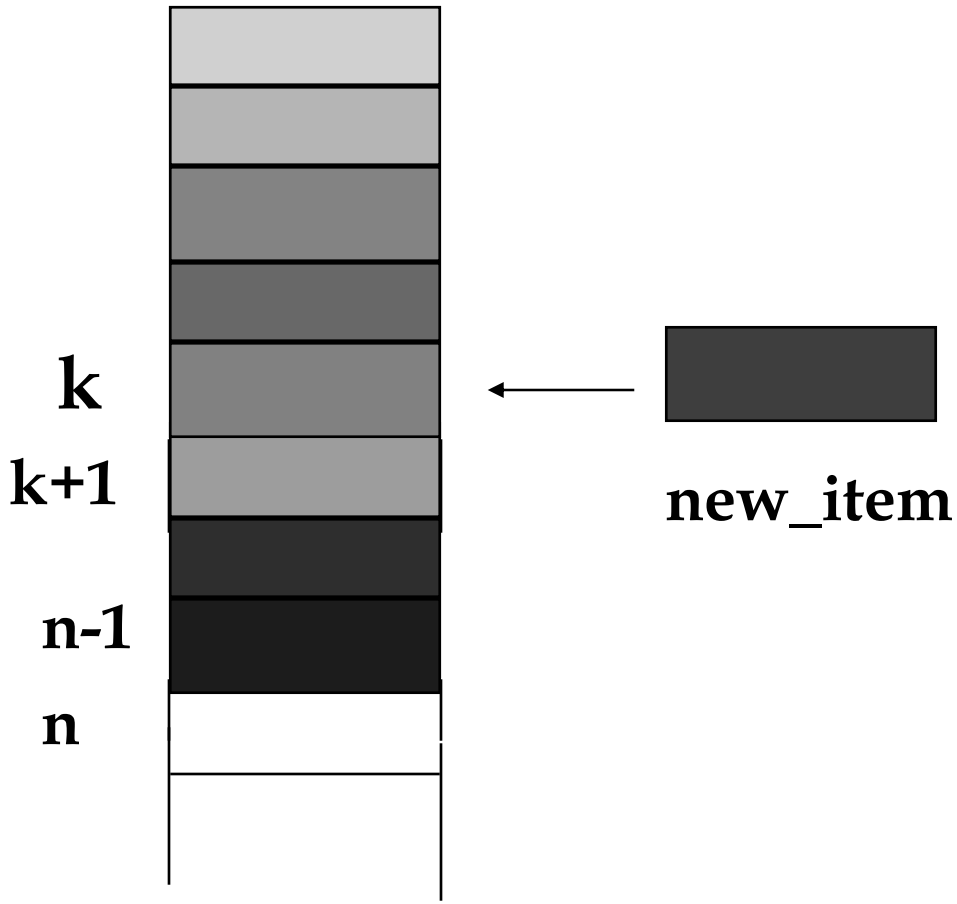


Random Access Lists - insert

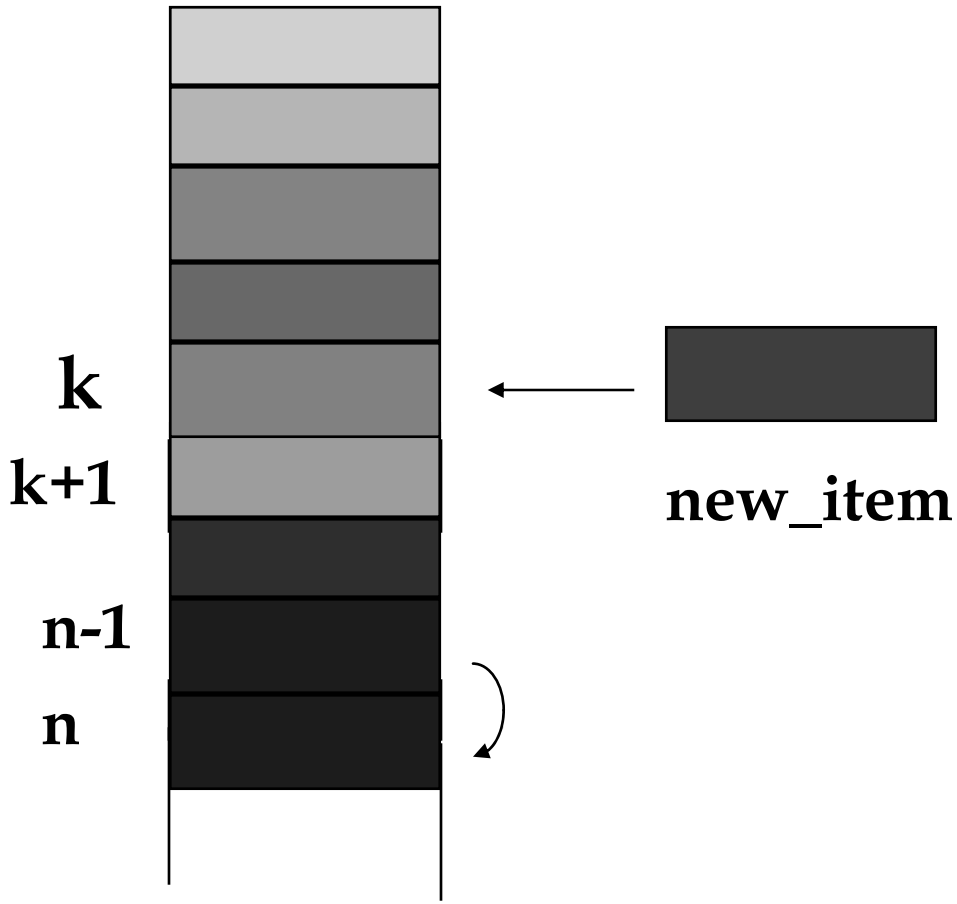


```
Random_Access_Lists_insert(Original_List[1..n-1])
        for (int i=0; i<k; i++)
           final_list[i] = original_list[i];
        final_list[k] = item;
        for (int i=k+1; i<=n; i++)
           final_list[i] = original_list[i-1];
        n++;
        original_list = final_list;
```

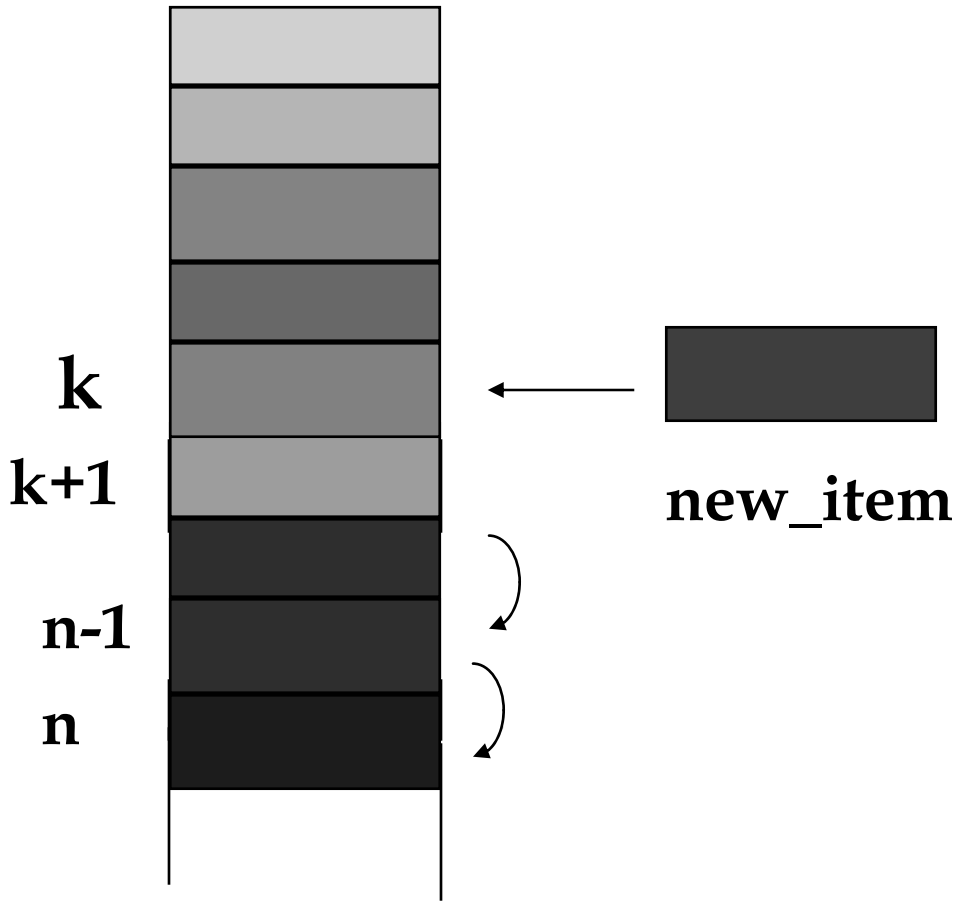
Random Access Lists -insert (in place)



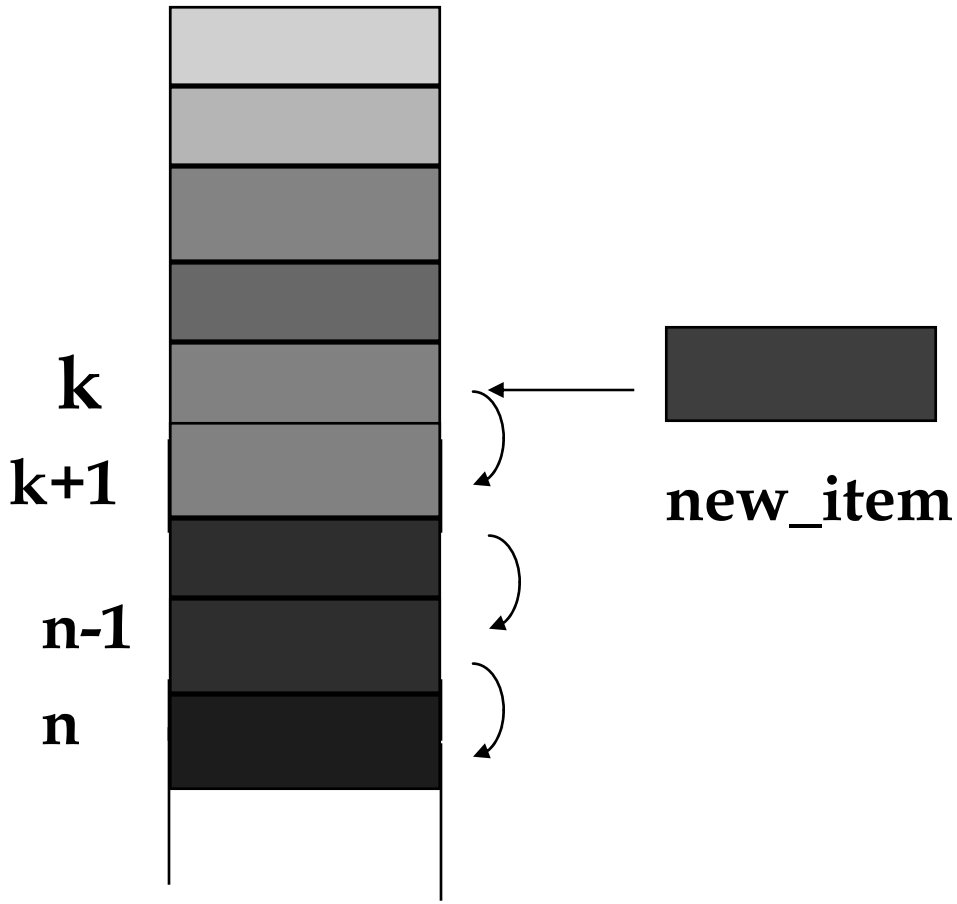
Random Access Lists -insert (in place)



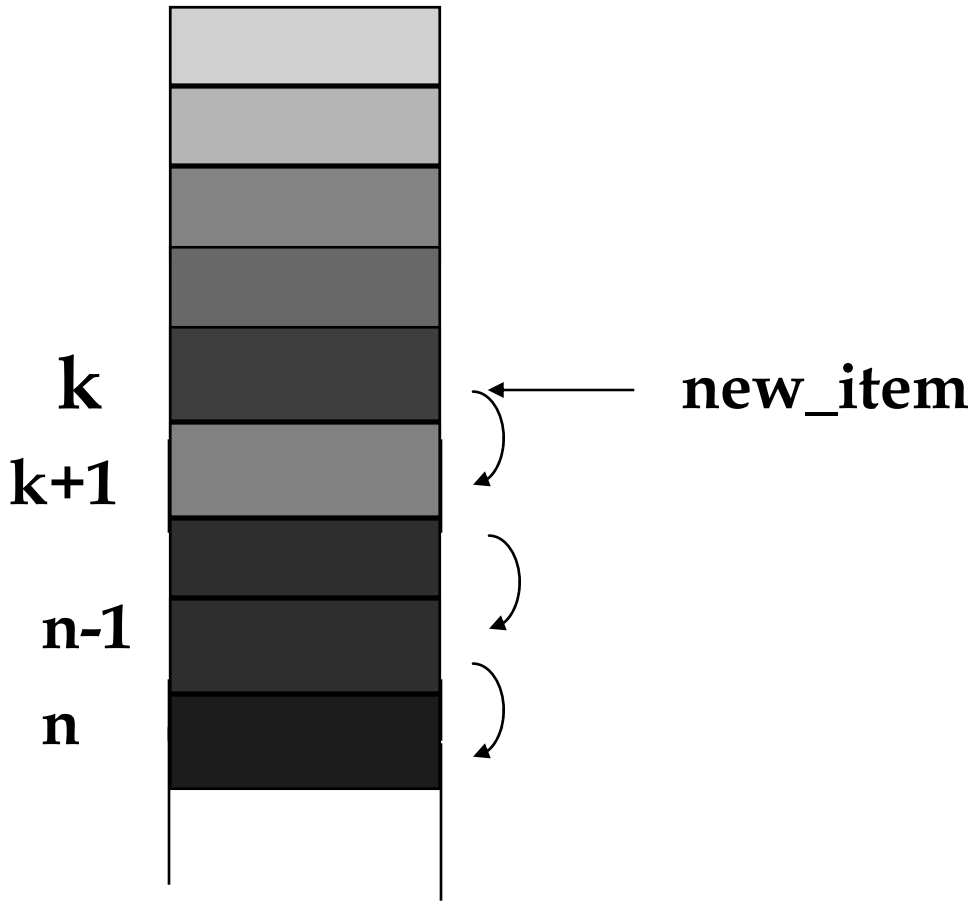
Random Access Lists -insert (in place)



Random Access Lists -insert (in place)



Random Access Lists -insert (in place)



Random Access Lists -insert (in place)

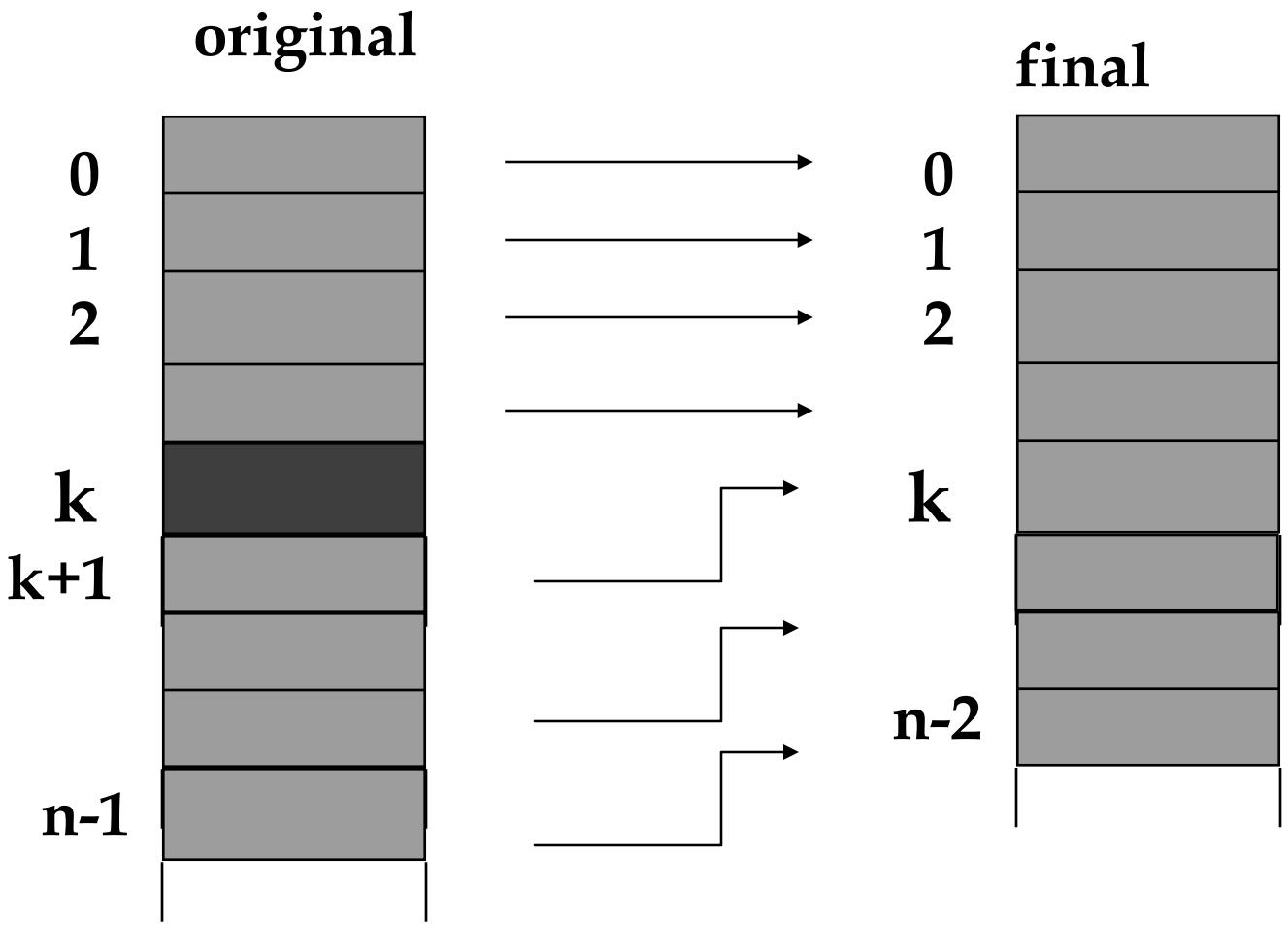
Algorithm (IL[1..n-1])

- **IL1 [move remainder]**
- **IL2 [move item]**
- **IL3 [store item]**
- **IL4 [update number]**

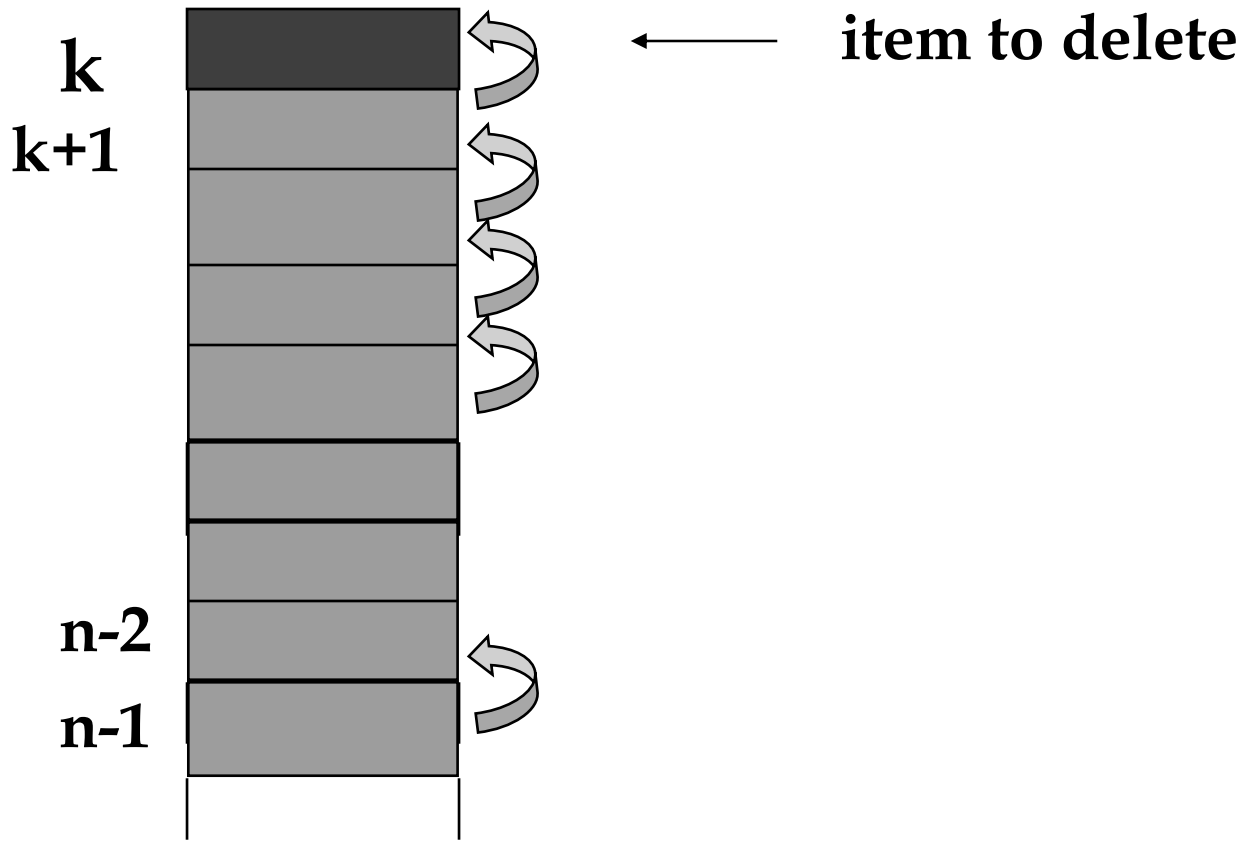
Random_Access_Lists_Insert_ In_Place([1..n-1]))

```
for (int  i=n; i>k; i- -)
    list[i] = list[i-1];
list[k] = item;
n ++;
```

Random Access Lists - delete



Random Access Lists - delete (in place)



Queues

Queue

- The data structure called a queue can also be implemented quite efficiently in a one-dimensional array. Here items are added and removed on a first-in-first-out (FIFO) basis.
- Adding an item is called an enqueue operation, while removing one is called dequeue.

Queue

- For both stacks and queues, one disadvantage of using an implementation in an array is that space usually has to be allocated at the outset for the maximum number of items envisages; if ever this space is not sufficient, it is difficult to add more, while if too much space is allocated, space wastage results.

Queue

- Arrays with two or more indexes can be declared in a similar way. For instance,

  matrix : array[1. . 20, 1 . . 20] of complex

- As in the case of an n x n matrix, then operations such as initializing every item of the array, or finding the largest item, now take a time in $\theta(n^2)$.
- Provided we are willing to use rather more space, the technique called virtual initialization allows us to avoid the time spent setting all the entries in the array.

Queue

- Suppose the array to be virtually initialized is T[1. . n]. Then we also need two auxiliary arrays of integers the same size as T, and an integer counter. Call these auxiliary arrays a[1. . n] and b[1. . n] and the counter ctr.
- Subsequently ctr tells us how many elements of T have been initialized, while the values a[1] to a[ctr] tell us which these elements are: a[1] points to the element initialized first, a[2] to the element initialized second, and so on.
- Figure depicts where three items of the array T have been initialized.

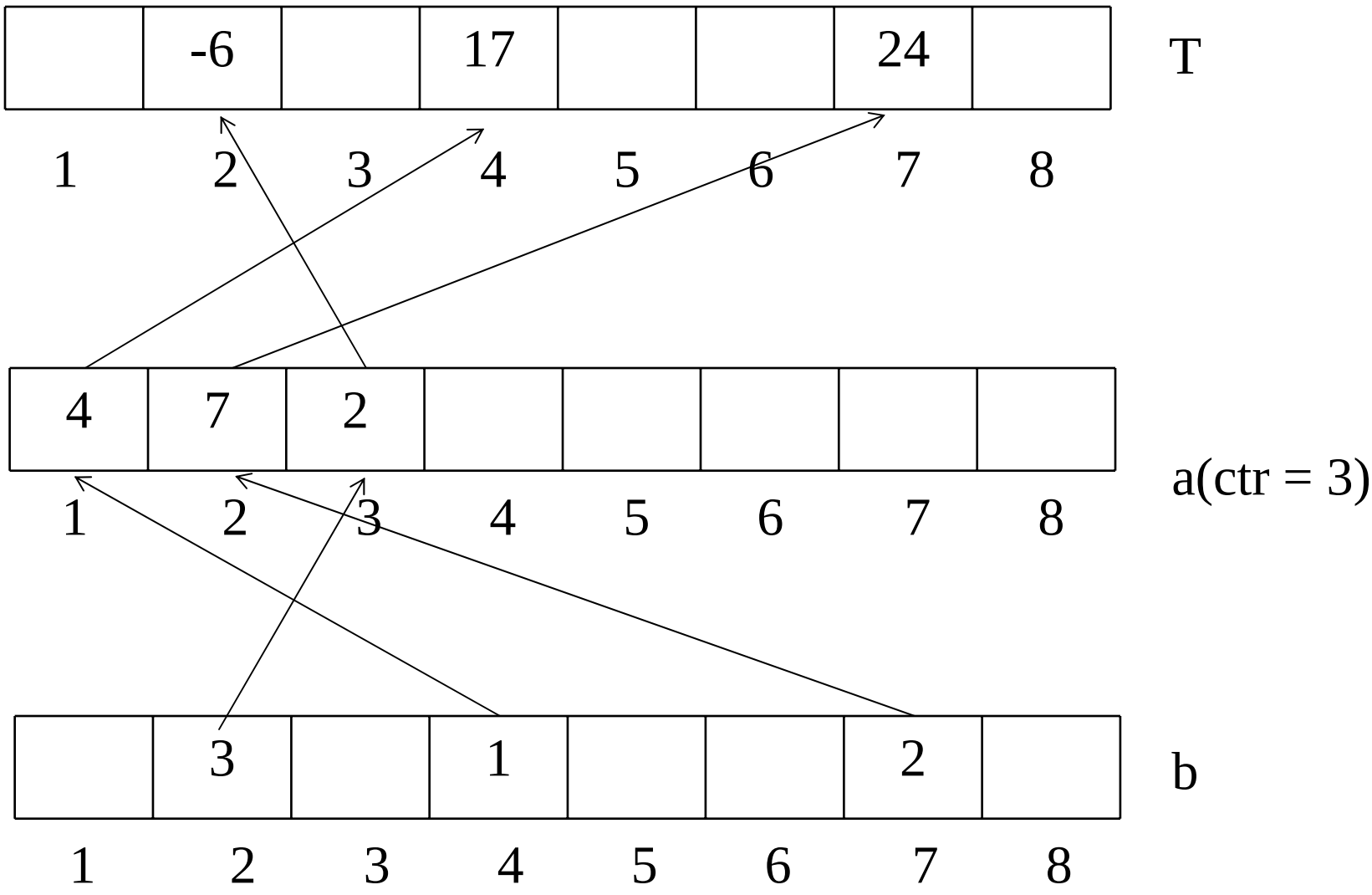


Figure : A virtually initialized array

Queue

- If a[b[i]] = i, so this test is conclusive: if it is satisfied, then T[i] has been initialized, and if not, not.
- To assign a value to T[i] for the first time, increment the counter ctr, set a[ctr] to i, set b[i] to ctr, and load the required value into T[i].

Queue

- A Queue is often called a "*First-in-First-Out*" (*FIFO*) data structure. This is because the data items are always added at one end of a queue, and removed from the other end.

- Analogy : people queuing up for tickets.

- *Operation* : The first data item to be placed in a queue is always the first one to be removed later.

Common operations on Queues

- create a queue
- add an item to a queue
- remove an item from a queue

  typically also return that item
- display a queue
- check if a queue is empty
- check if a queue is full

Projector reservation system

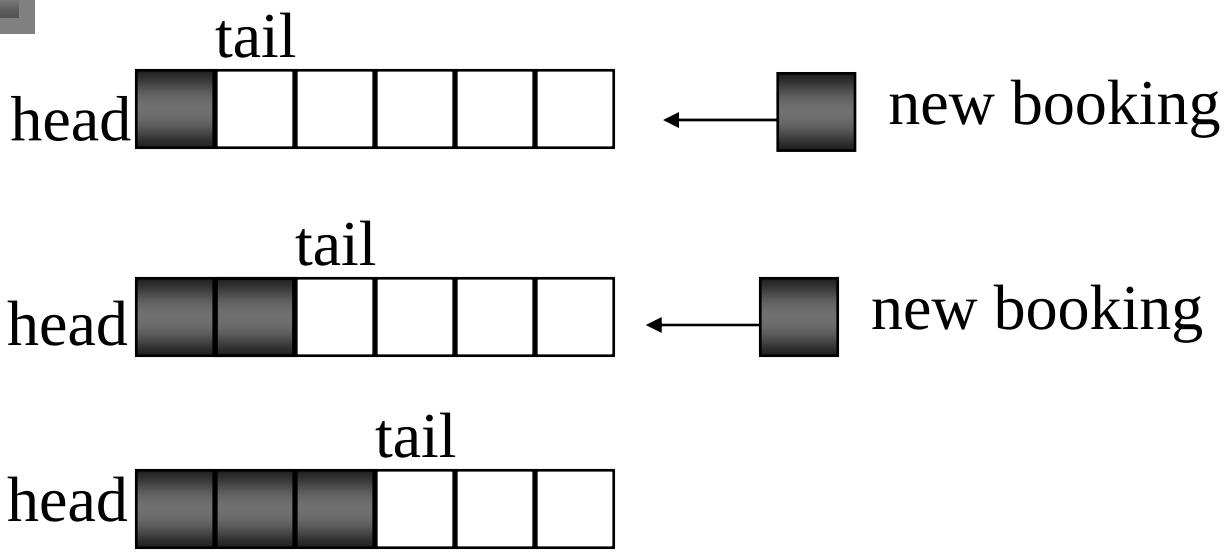
In the computer department, staff wishing to use the one and only projector would have to book it with the Projector Incharge. When the projector becomes available it will be allocated on a first come first served basis. The Projector Incharge also needs to know who was the last person to have used the projector or the one before that in case problems or damages arise with the projector so that the culprit can be identified. Implement a system which will allow the Projector Incharge to take bookings and also to find out who to allocate the projector to next as well as to output a list of most recent users.

Important Structures and Element(s) in this Problem

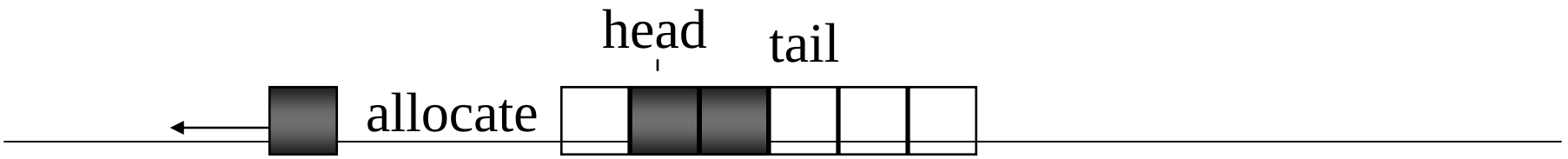
- **A list to hold bookings in the order the booking is received.**
- **The next person to be allocated the projector will be the one at the front of the list**
- **A list which can be used to output the most recent users.**

Restricted Access Lists

Booking:



Projector becomes available, allocate to next in line



Restricted Access Lists

Abstract Data Structure:

QUEUE (FIFO)

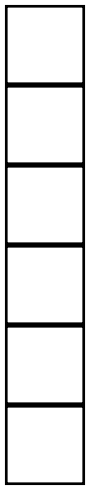
Adding is done at one end of the list (tail) and removing is done at the other end of the list (head)

Restricted Access Lists

When projector becomes available, the booker is removed from the booking queue and added to the list of users:

(bookings)

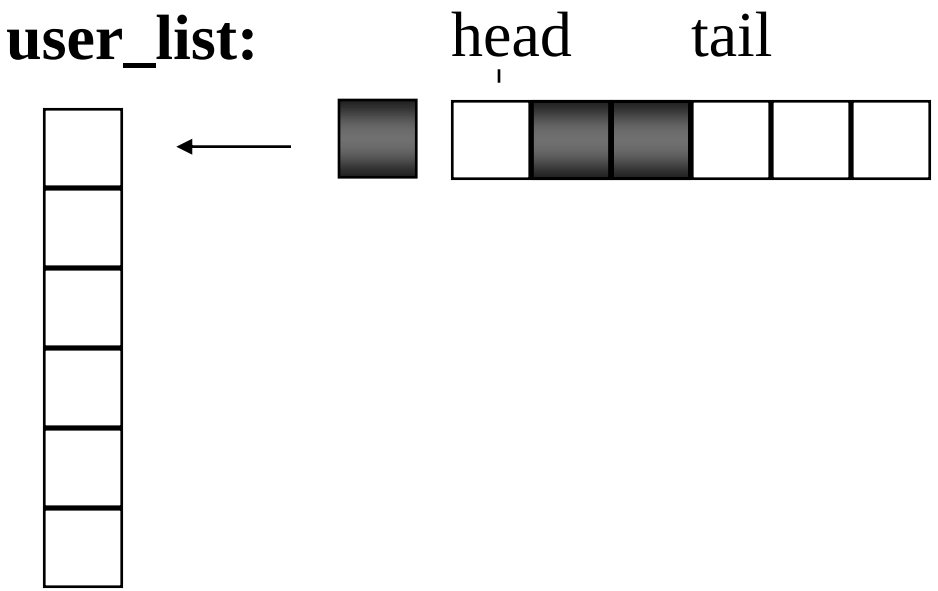
user_list:



head tail

Restricted Access Lists

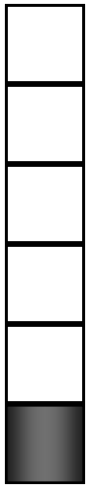
(bookings)

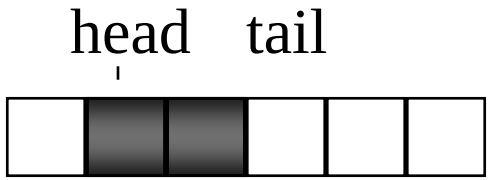


Restricted Access Lists

(bookings)

_list user :





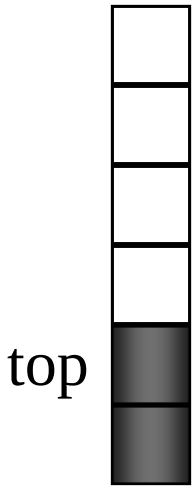
Restricted Access Lists

(bookings)

head

tail

user_list:



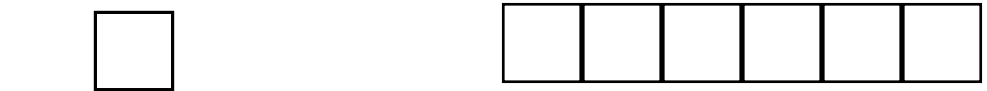
top

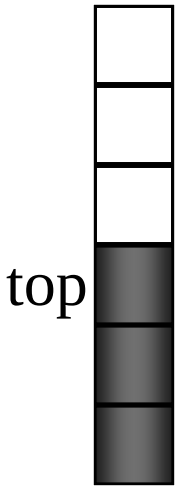
Restricted Access Lists

(bookings)

head
tail

user_list:

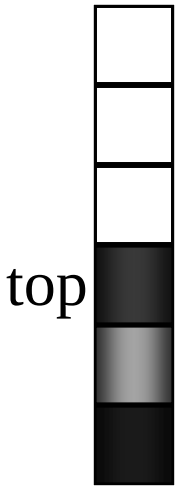




Restricted Access Lists

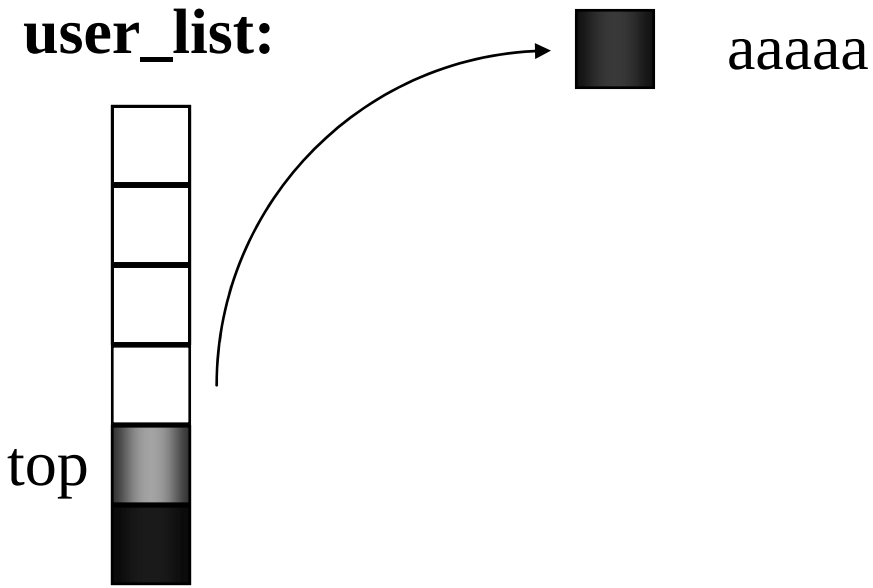
When printing list of users:

user_list:



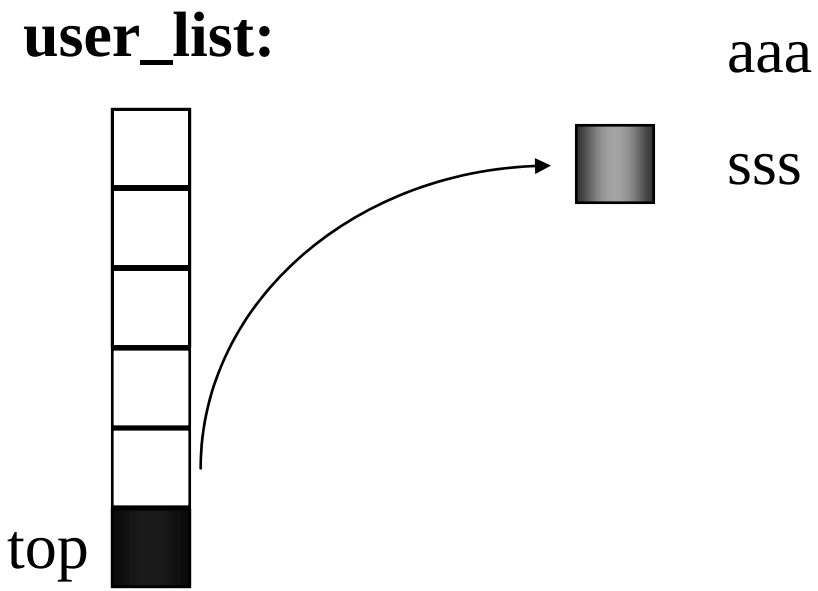
Restricted Access Lists

When printing list of users:



Restricted Access Lists

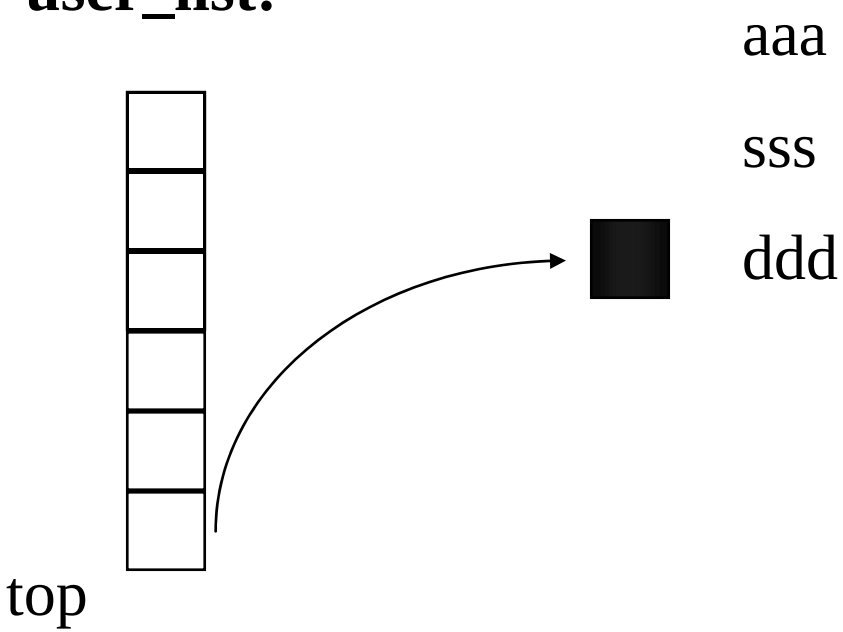
When printing list of users:



Restricted Access Lists

When printing list of users:

user_list:



Stack

- Items are added to the structure, and subsequently removed, on a Last-In-First-Out (LIFO) basis. The situation can be represented using an array called stack, whose index runs from 1 to the maximum required size of the stack.
- To set the stack empty, the counter is given the value zero; to add an item to the stack the counter is incremented, and then the new item is written into stack[counter]; to remove an item, the value of stack[counter] is read out, and then the counter is decremented.

Stack

- Adding an item to a stack is usually called a push operation, while removing one is called pop.

Restricted Access Lists

Abstract Data Structure:

STACK (LIFO)

Adding and removing is done at one end of the list (top) only

Stacks & Queues

Situations to watch for : -

- **empty stack and queue - no removing operation allowed (underflow)**
- **full stack and queue - no adding operation allowed (overflow)**

Projector reservation system

Now for the problem at hand:

booking_list is a Queue

users_list is a Stack

Declare the instances above:

Queue booking_list;

Stack users_list;