# ALGORITHMS -

## DATA STRUCTURES AND ALGORITHMS

الخوارزميات للمرحلة الثانية قسم علوم الحاسبات

الدكتور
اثير العاني

# TOPICS

Introduction

Definitions

Classification of Data Structures

Arrays and Linked Lists

Abstract Data Types [ADT]

- The List ADT
  - Array-based Implementation
  - Linked List Implementation
  - Cursor-based Implementation

Doubly Linked Lists

# DATA STRUCTURE

Data Structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Different kinds of data structures are suited to different kinds of applications.

Storing and retrieving can be carried out on data stored in both main memory and in secondary memory.

Way in which data are stored for efficient search and retrieval.

The simplest data structure is the one-dimensional (linear) array.

Data items stored non-consecutively in memory may be linked by pointers.

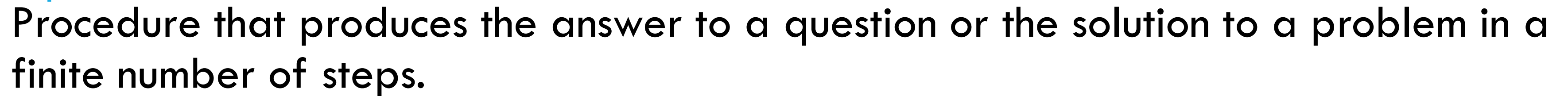
Many algorithms have been developed for storing data efficiently

# ALGORITHMS

An algorithm is a step-by-step procedure for calculations.

An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function.

The transition from one state to the next is not necessarily deterministic; some algorithms incorporate random input.



Procedure that produces the answer to a question or the solution to a problem in a finite number of steps.

An algorithm that produces a yes or no answer is called a decision procedure; one that leads to a solution is a computation procedure.

Example: A mathematical formula and the instructions in a computer program

# DATA STRUCTURE CLASSIFICATION

Primitive / Non-primitive

- Basic Data Structures available / Derived from Primitive Data Structures

Homogeneous / Heterogeneous

- Elements are of the same type / Different types

Static / Dynamic

- memory is allocated at the time of compilation / run-time

Linear / Non-linear

- Maintain a Linear relationship between element

# ADT - GENERAL CONCEPT

Problem solving with a computer means processing data

To process data, we need to define the data type and the operation to be performed on the data

The definition of the data type and the definition of the operation to be applied to the data is part of the idea behind an Abstract Data Type (ADT)

# ADT - GENERAL CONCEPT

The user of an ADT needs only to know that a set of operations are available for the data type, but does not need to know how they are applied

Several simple ADTs, such as integer, real, character, pointer and so on, have been implemented and are available for use in most languages

# DATA TYPES

A data type is characterized by:

- A set of *values*
- A *data representation,* which is common to all these values, and
- A set of *operations,* which can be applied uniformly to all these values

# PRIMITIVE DATA TYPES

Languages like ‘C’ provides the following primitive data types:

- boolean
- char, byte, int
- float, double

Each primitive type has:

- A set of values
- A data representation
- A set of operations

# ADT DEFINITION

In computer science, an abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior.

An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.

# ADT DEFINITION

An ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language.

example, an abstract stack could be defined by three operations:

- push, that inserts some data item onto the structure,
- pop, that extracts an item from it, and
- peek, that allows data on top of the structure to be examined without removal.

Abstract data types or ADTs are a mathematical specification of a set of data and the set of operations that can be performed on the data.

They are abstract in the sense that the focus is on the definitions and the various operations with their arguments.

The actual implementation is not defined, and does not affect the use of the ADT.

# ADT IN SIMPLE WORDS

Definition:

- Is a set of operation
- Mathematical abstraction
- No implementation detail

Example:

- Lists, sets, graphs, stacks are examples of ADT along with their operations

# WHY ADT?

Modularity

- divide program into small functions
- easy to debug and maintain
- easy to modify
- group work

Reuse

- do some operations only once

Easy to change the implementation

- transparent to the program

# IMPLEMENTING AN ADT

To implement an ADT, you need to choose:

- A data representation
  - must be able to represent all necessary values of the ADT
  - should be private
- An algorithm for each of the necessary operation:
  - must be consistent with the chosen representation
  - all auxiliary (helper) operations that are not in the contract should be private

Remember: Once other people are using it

- It's easy to add functionality

# THE LIST ADT

The List is an

- Ordered sequence of data items called elements
- $A_1, A_2, A_3, \ldots, A_N$ is a list of size N
- size of an empty list is 0
- $A_{i+1}$ succeeds $A_i$
- $A_{i-1}$ preceeds $A_i$
- Position of $A_i$ is i
- First element is $A_1$ called "head"
- Last element is $A_N$ called "tail"

# OPERATIONS ON LISTS

MakeEmpty

PrintList

Find

FindKth

Insert

Delete

Next

Previous

# LIST – AN EXAMPLE

The elements of a list are 34, 12, 52, 16, 12

- Find (52) -> 3
- Insert (20, 4) -> 34, 12, 52, 20, 16, 12
- Delete (52) -> 34, 12, 20, 16, 12
- FindKth (3) -> 20

# LIST - IMPLEMENTATION

Lists can be implemented using:

- Arrays
- Linked List

# ARRAYS

Array is a static data structure that represents a collection of fixed number of homogeneous data items or

A fixed-size indexed sequence of elements, all of the same type.

The individual elements are typically stored in consecutive memory locations.

The length of the array is determined when the array is created, and cannot be changed.

# ARRAYS

Any component of the array can be inspected or updated by using its index.

- This is an efficient operation
- O(1) = constant time

The array indices may be integers (C, Java) or other discrete data types (Pascal, Ada).

The lower bound may be zero (C, Java), one (Fortran), or chosen by the programmer (Pascal, Ada)

# DIFFERENT TYPES OF ARRAYS

One-dimensional array: only one index is used

Multi-dimensional array: array involving more than one index

Static array: the compiler determines how memory will be allocated for the array

Dynamic array: memory allocation takes place during execution

# ONE DIMENSIONAL STATIC ARRAY

Syntax:

- `ElementType arrayName [CAPACITY];`
- `ElementType arrayName [CAPACITY] = { initializer_list };`

Example in C++:

- `int  b [5];`
- `int  b [5] = {19, 68, 12, 45, 72};`

# ARRAY OUTPUT FUNCTION

```
void display(int array[],int num_values)
{
 for (int I = 0; i<num_values; i++)
        cout<< array[i] << "  ";
}
```

# LIST IMPLEMENTED USING ARRAY

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 2.3 | 7.1 | 0.3 | 9.5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

size = 4

capacity = 10

# OPERATIONS ON LISTS

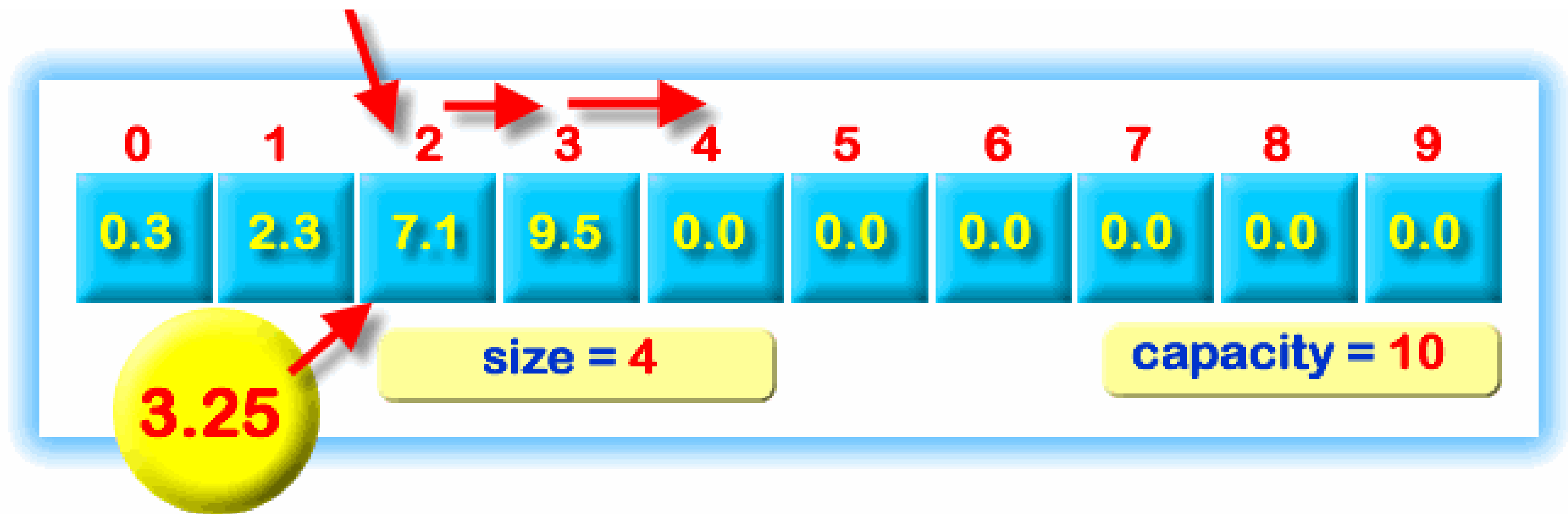
We'll consider only few operations and not all operations on Lists

Let us consider Insert

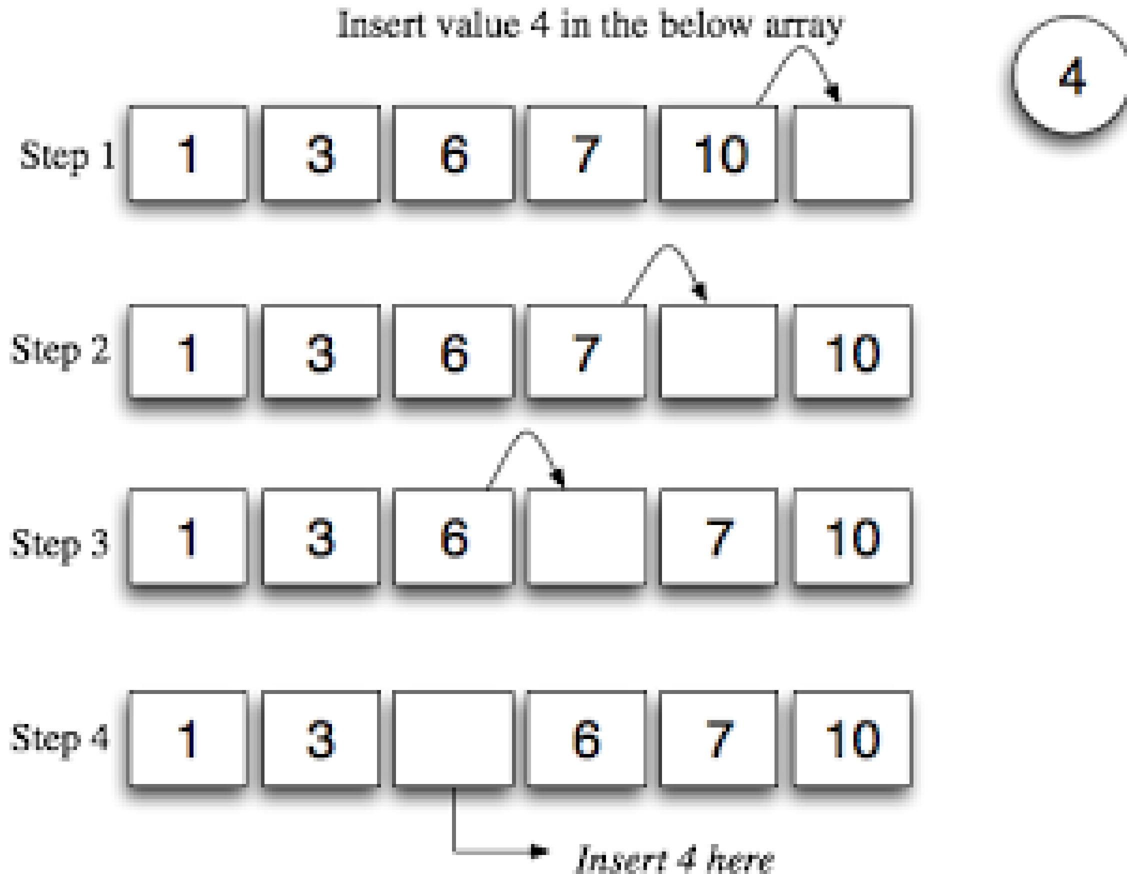
There are two possibilities:

- Ordered List
- Unordered List

# INSERTION INTO AN ORDERED LIST



# INSERTION IN DETAIL



# FIND / SEARCH

Searching is the process of looking for a specific element in an array

For example, discovering whether a certain score is included in a list of scores.

Searching, like sorting, is a common task in computer programming.

There are many algorithms and data structures devoted to searching.

The most common one is the linear search.

# LINEAR SEARCH

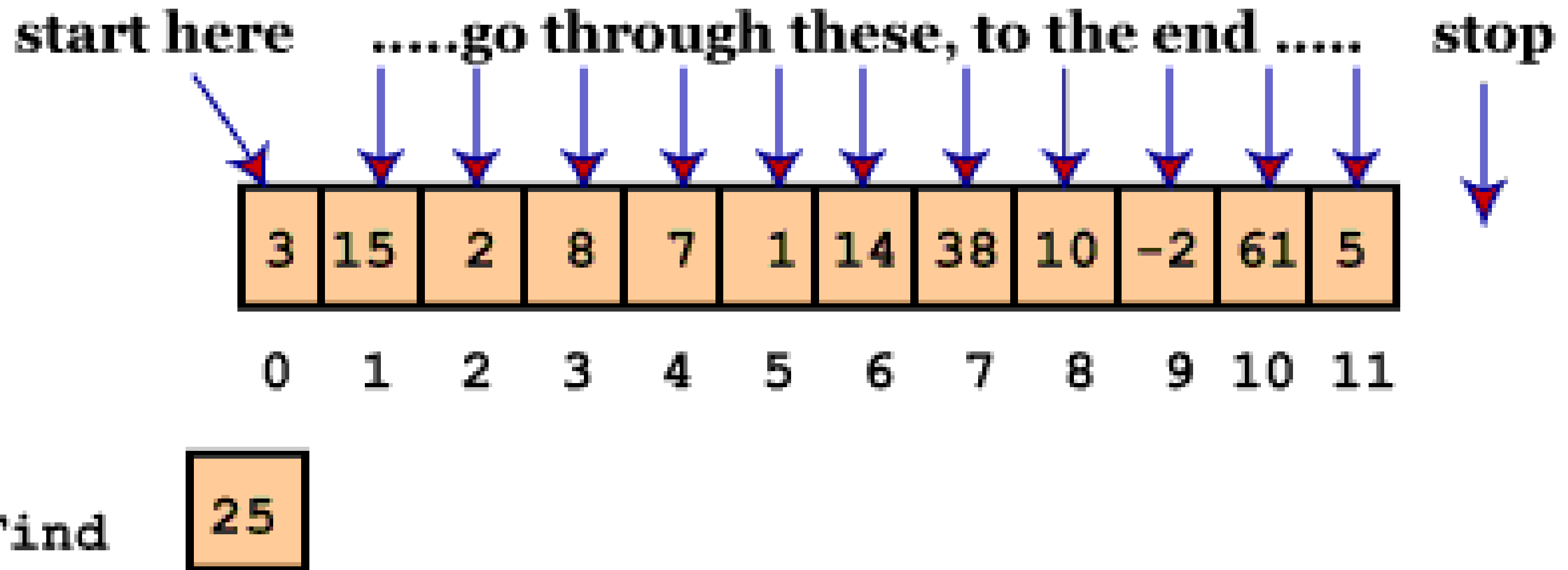
The linear search approach compares the given value with each element in the array.

The method continues to do so until the given value matches an element in the list or the list is exhausted without a match being found.

If a match is made, the linear search returns the index of the element in the array that matches the key.

If no match is found, the search returns -1.

# LINEAR SEARCH



# LINEAR SEARCH FUNCTION

```
int LinearSearch (int a[], int n, int key)
{
   int i;
   for(i=0; i<n; i++)
   {
     if (a[i] == key)
        return i;
   }
   return -1;
}
```

# USING THE FUNCTION

```
LinearSearch (a,n,item,loc)
```

Here "a" is an array of the size n.

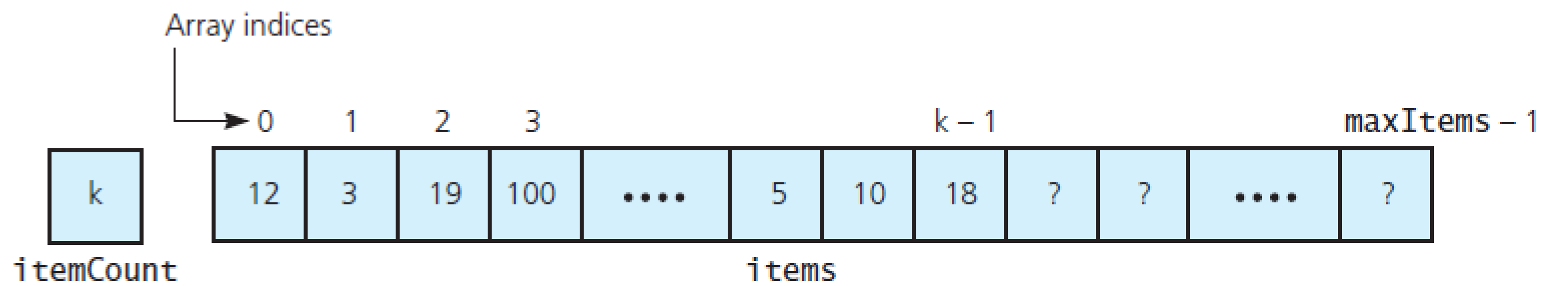
This algorithm finds the location of the element "item" in the array "a".

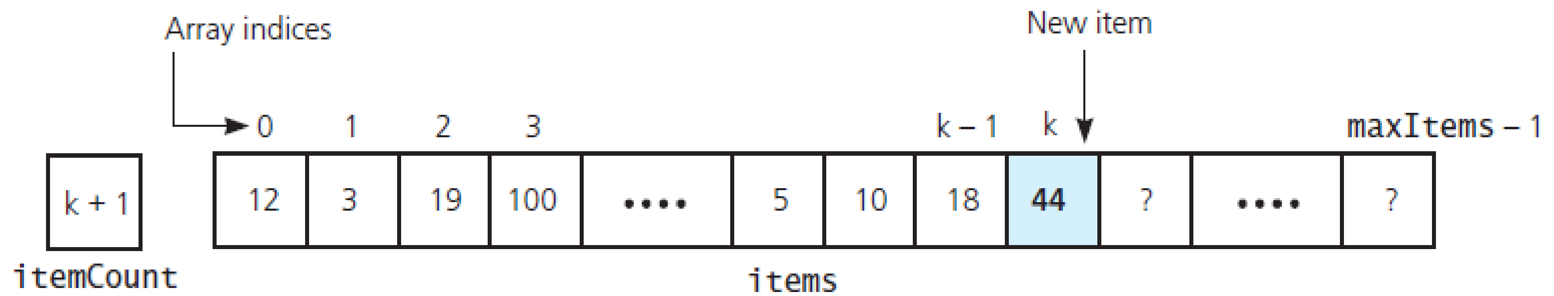
If search item is found, it sets loc to the index of the element; otherwise, it sets loc to -1

```
index=linearsearch(array, num, key)
```

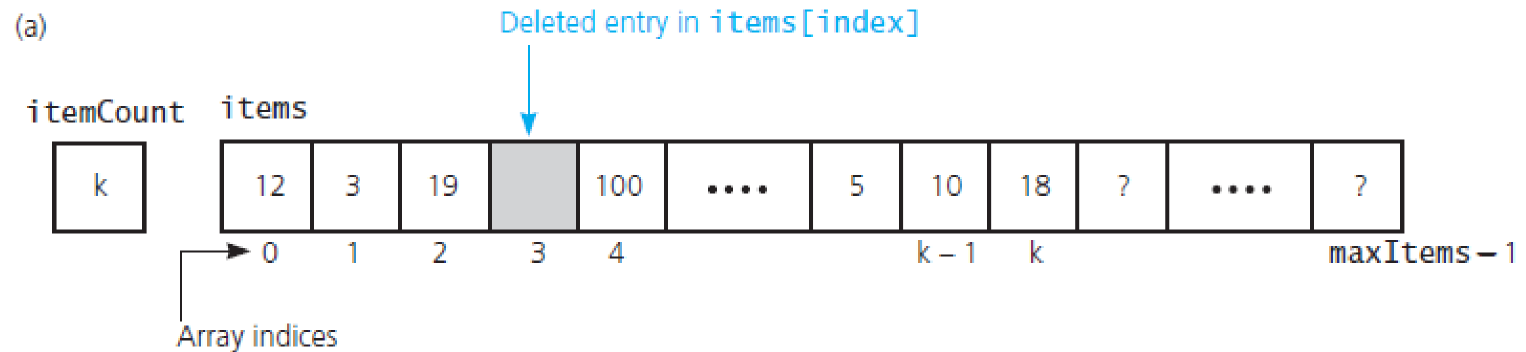
# PRINTLIST OPERATION

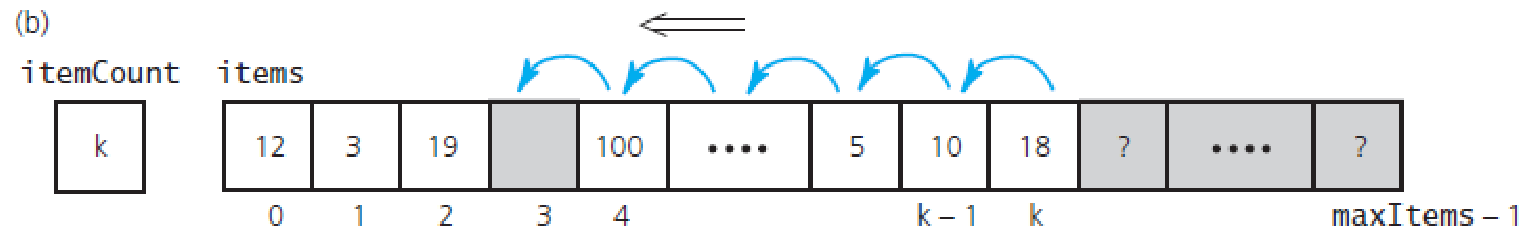
```
int myArray [5] = {19,68,12,45,72};
/* To print all the elements of the array
for (int i=0;i<5;i++)
{
printf("%d", myArray[i]);
}
```


Array indices
0
1
2
3
k – 1
maxItems – 1
k
12
3
19
100
••••
5
10
18
?
?
••••
?
itemCount
items


Array indices
New item
0
1
2
3
k – 1
k
maxItems – 1
k + 1
12
3
19
100
••••
5
10
18
44
?
••••
?
itemCount
items

# IMPLEMENTING DELETION



# OPERATİONS RUNNİNG TİMES

PrintList } O(N)
Find

Insert } O(N) (on avarage half
Delete needs to be moved)

FindKth
Next } O(1)
Previous

# DISADVANTAGES OF USING ARRAYS

Need to define a size for array

- High overestimate (waste of space)

insertion and deletion is very slow

- need to move elements of the list

redundant memory space

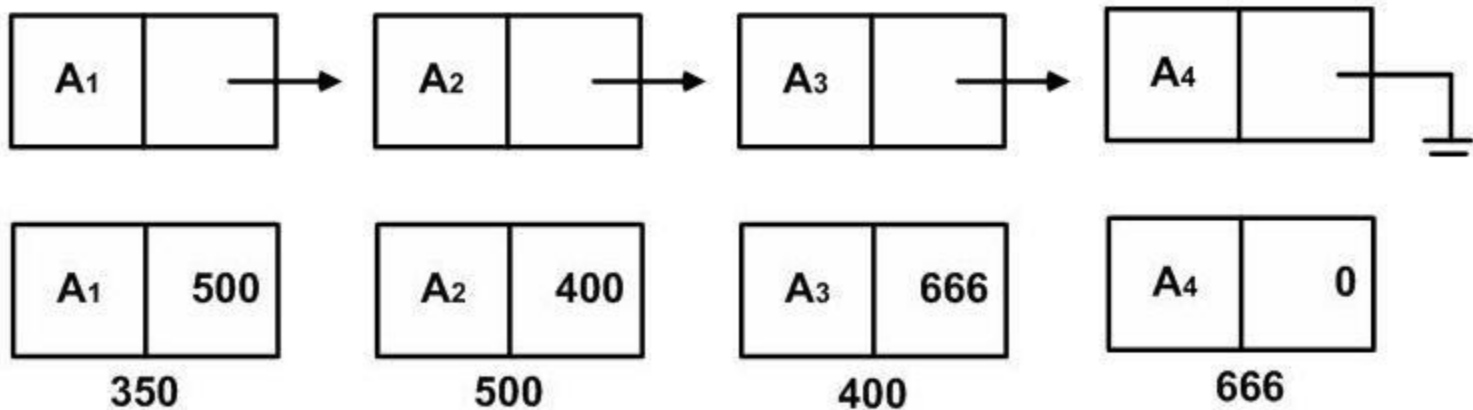
- it is difficult to estimate the size of array

# LINKED LIST

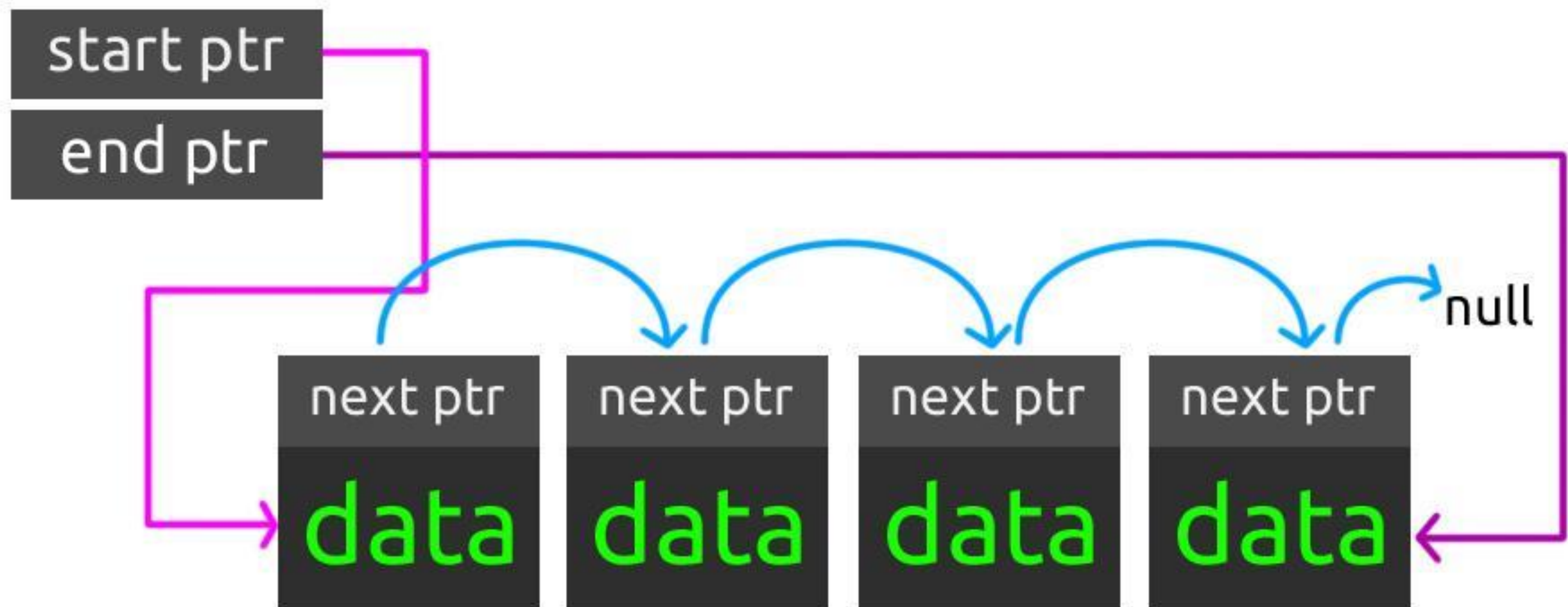
Series of nodes

- not adjacent in memory
- contain the element and a pointer to a node containing its succesor

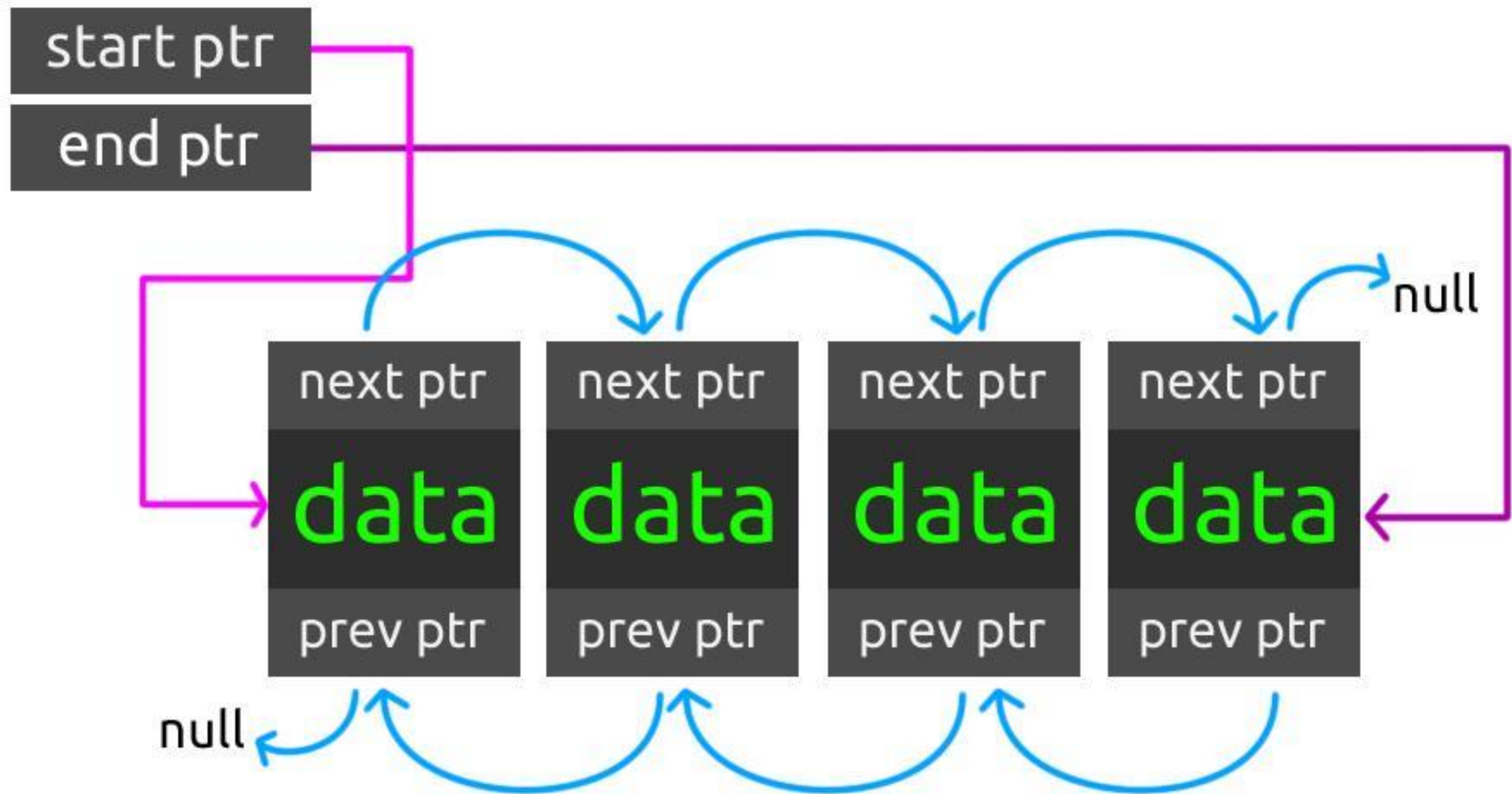
Avoids the linear cost of insertion and deletion!



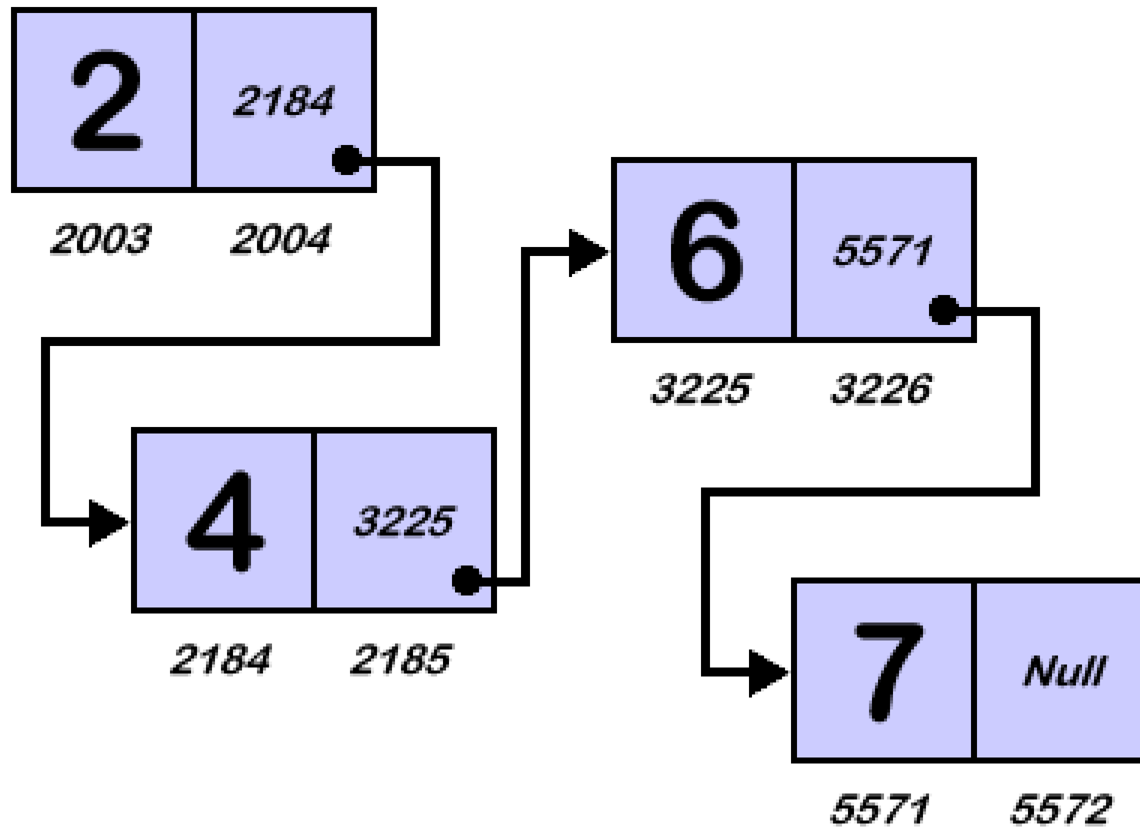
# SINGLY LINKED LIST



# DOUBLY LINKED LIST



# SINGLY LINKED LIST



# SINGLY-LINKED LIST - ADDITION

Insertion into a singly-linked list has two special cases.

It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list).

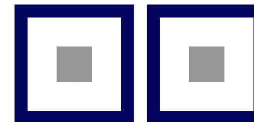
In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list.

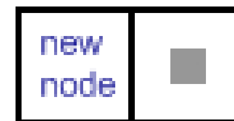
# EMPTY LIST CASE

When list is empty, which is indicated by (head == NULL) condition, the insertion is quite simple.

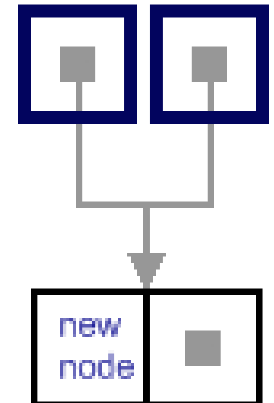
Algorithm sets both head and tail to point to the new node.

before insertion



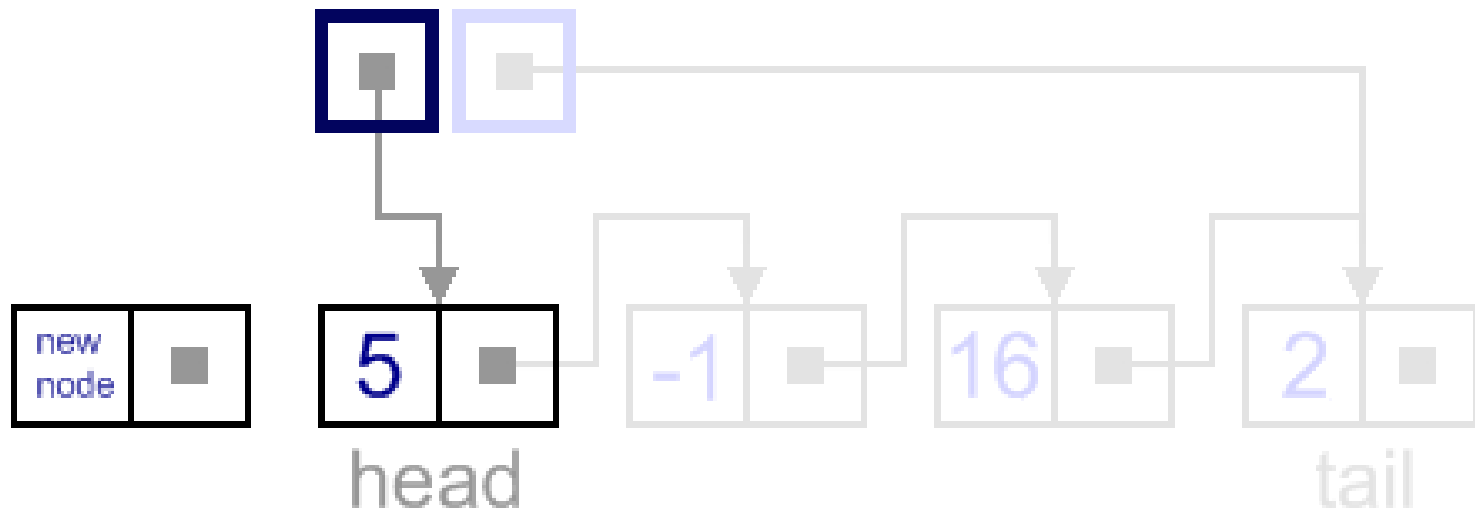



after insertion



# ADD FIRST

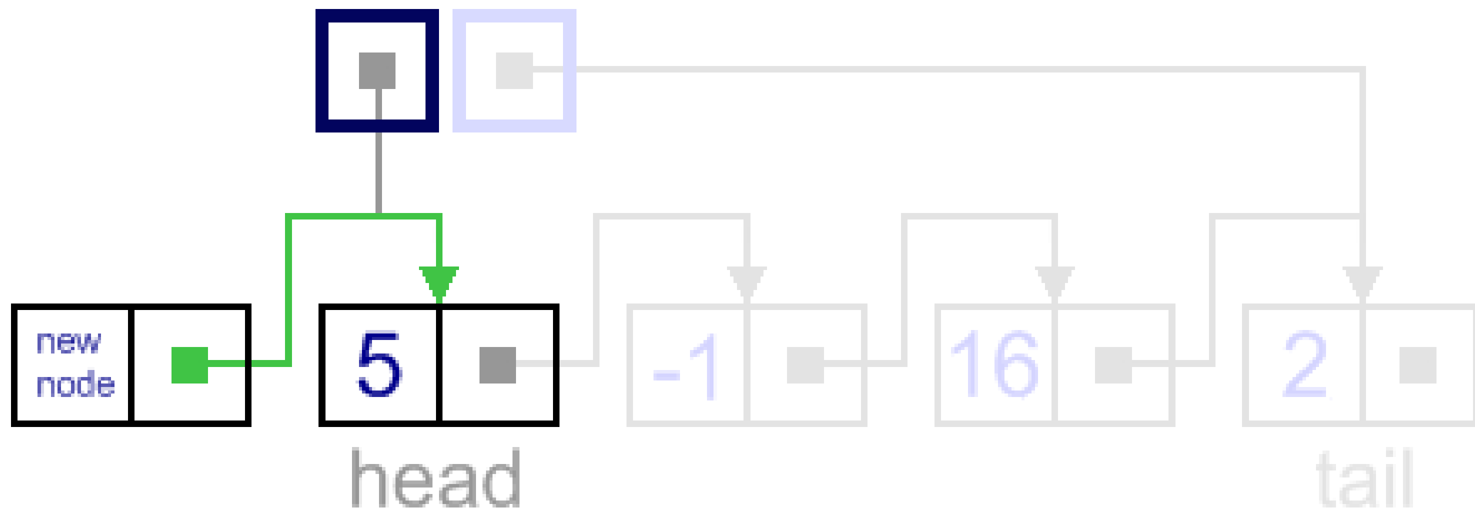
In this case, new node is inserted right before the current head node.



# ADD FIRST - STEP 1

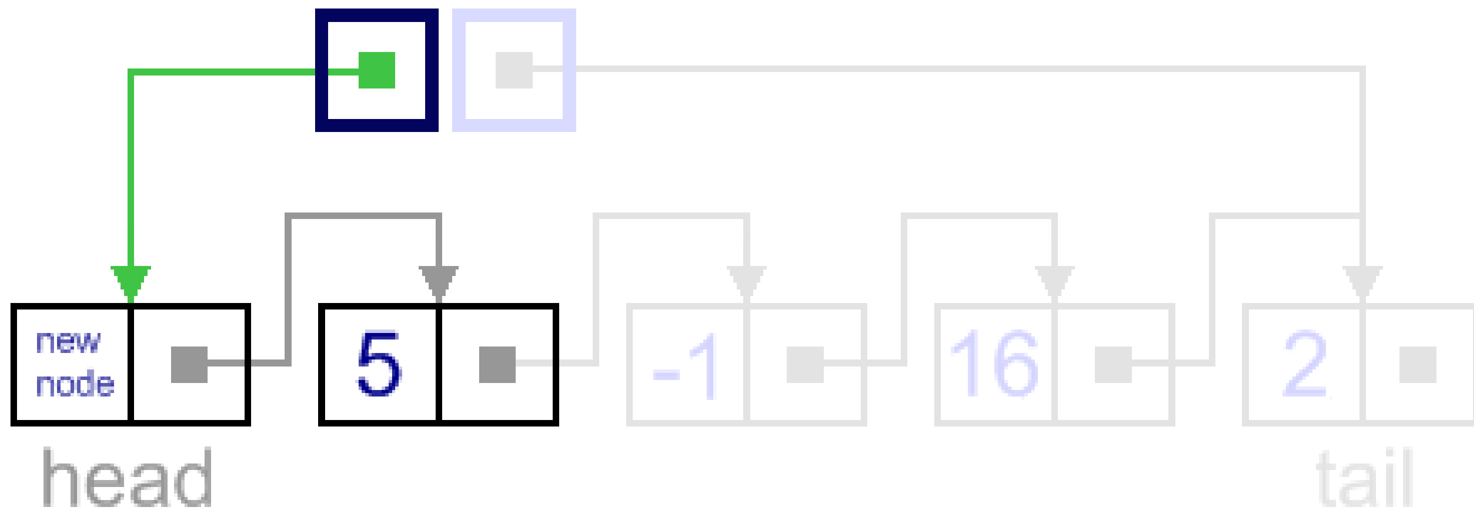
It can be done in two steps:

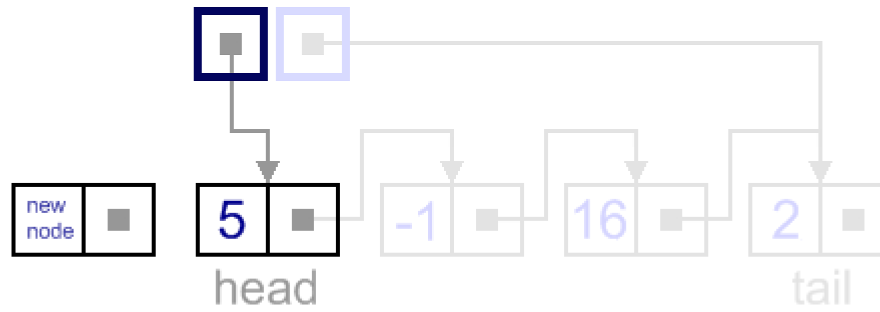
- Update the next link of the new node, to point to the current head node.

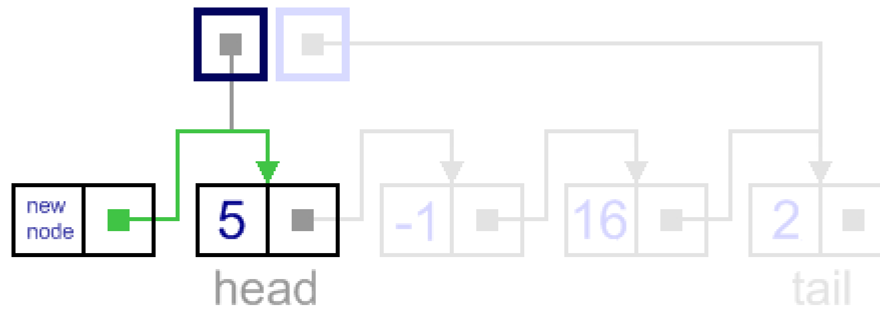


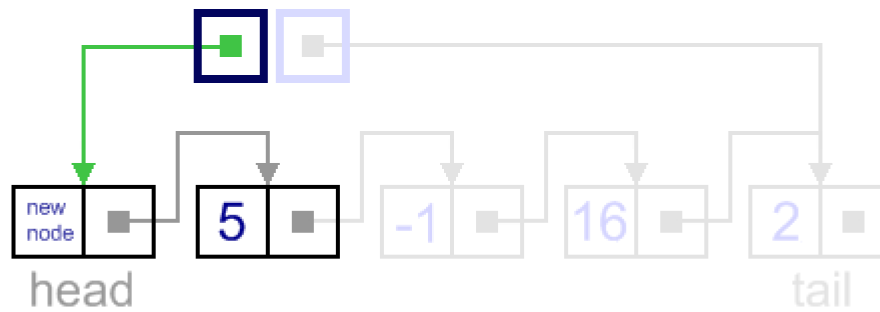
# ADD FIRST - STEP 2

- Update head link to point to the new node.




new
node
5
-1
16
2
head
tail


new
node
5
-1
16
2
head
tail

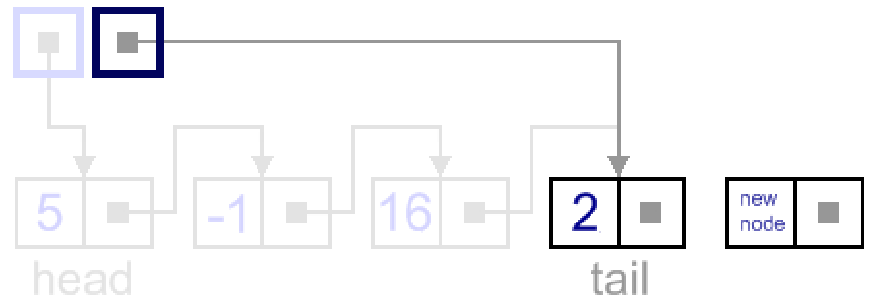

new
node
5
-1
16
2
head
tail

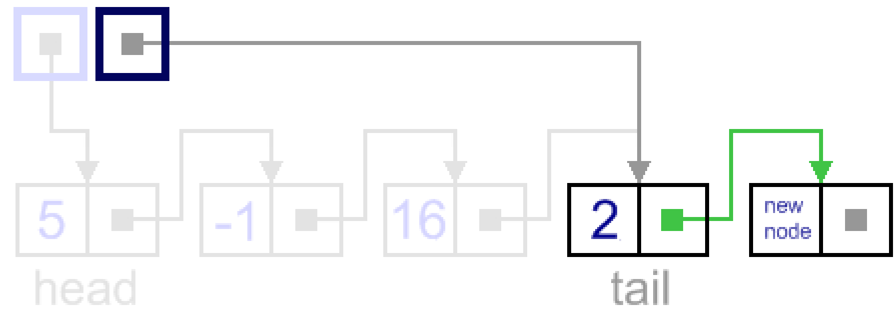
# ADD LAST

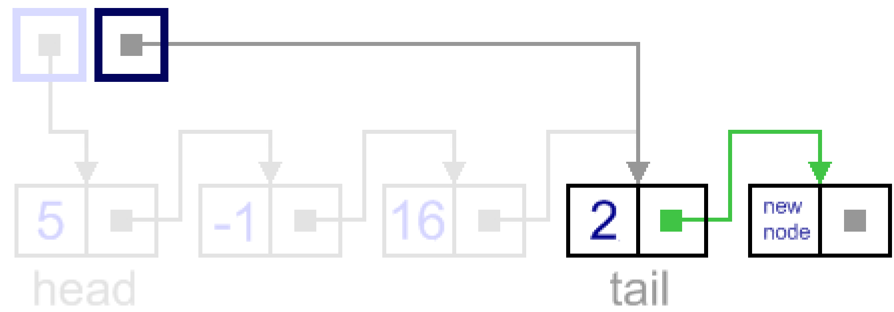
In this case, new node is inserted right after the current tail node.

It can be done in two steps:

- Update the next link of the current tail node, to point to the new node.
- Update tail link to point to the new node.


5
-1
16
2
new
node
head
tail


5
-1
16
2
new
node
head
tail


5
-1
16
2
new
node
head
tail

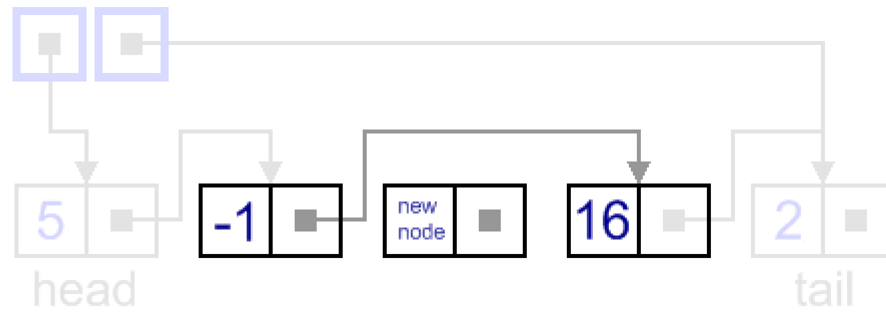
# INSERT - GENERAL CASE

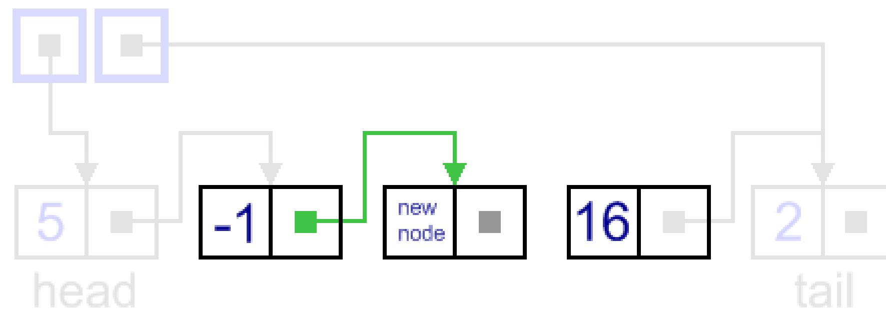
In general case, new node is always inserted between two nodes, which are already in the list. Head and tail links are not updated in this case.

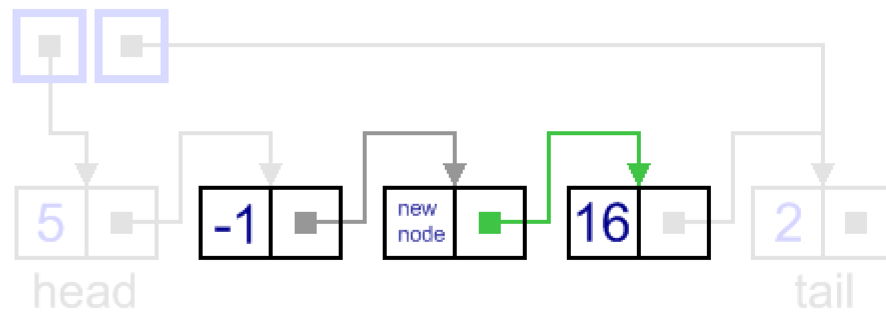
We need to know two nodes "Previous" and "Next", between which we want to insert the new node.

This also can be done in two steps:

- Update link of the "previous" node, to point to the new node.
- Update link of the new node, to point to the "next" node.


5
-1
new
node
16
2
head
tail


5
-1
new
node
16
2
head
tail


5
-1
new
node
16
2
head
tail

# SINGLY-LINKED LIST - DELETION

There are four cases, which can occur while removing the node.

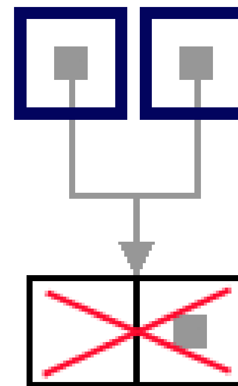
We have the same four situations, but the order of algorithm actions is opposite.

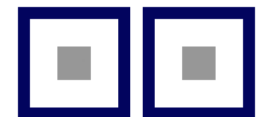
Notice, that removal algorithm includes the disposal of the deleted node - unnecessary in languages with automatic garbage collection (Java).

# LIST HAS ONLY ONE NODE

When list has only one node, that the head points to the same node as the tail, the removal is quite simple.

Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to NULL.



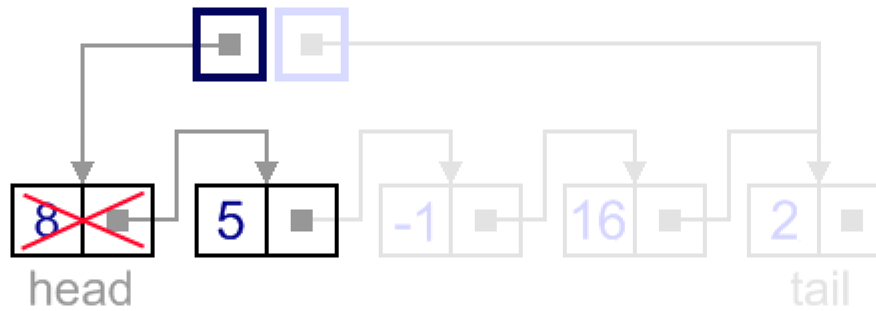



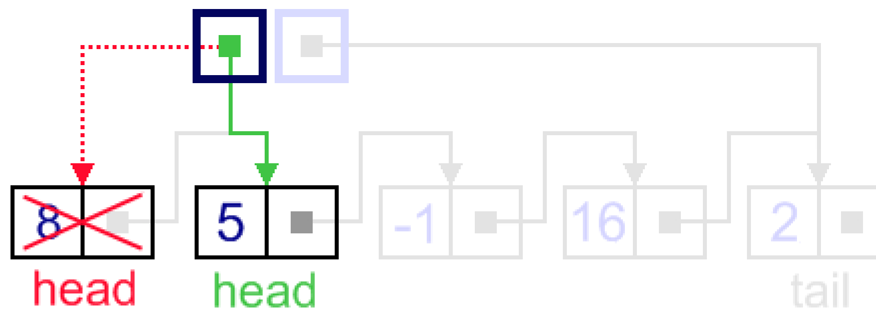
# REMOVE FIRST

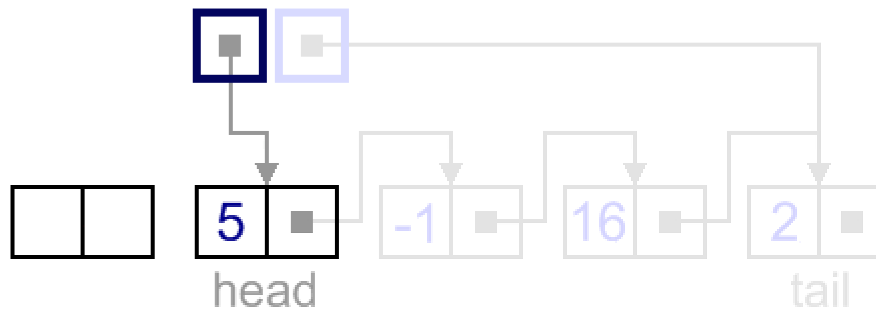
In this case, first node (current head node) is removed from the list.

It can be done in two steps:

- Update head link to point to the node, next to the head.
- Dispose removed node.


8
5
-1
16
2
head
tail


8
5
-1
16
2
head
head
tail

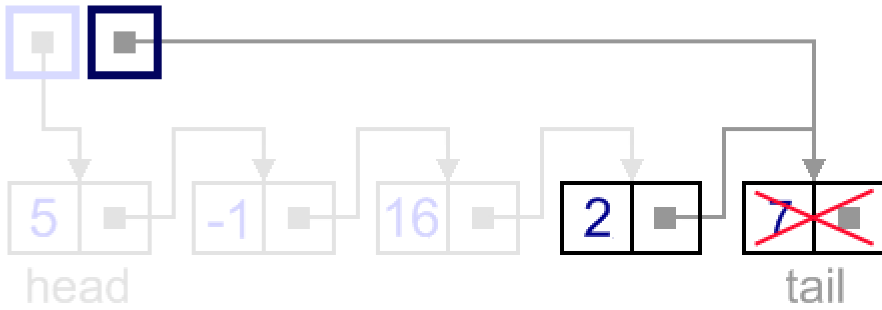

5
-1
16
2
head
tail

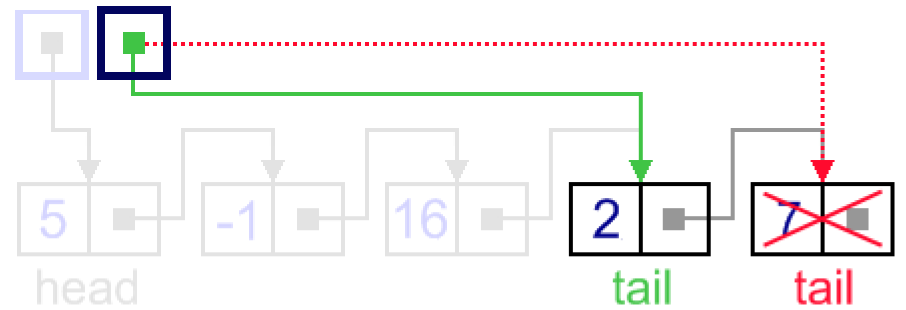
# REMOVE LAST

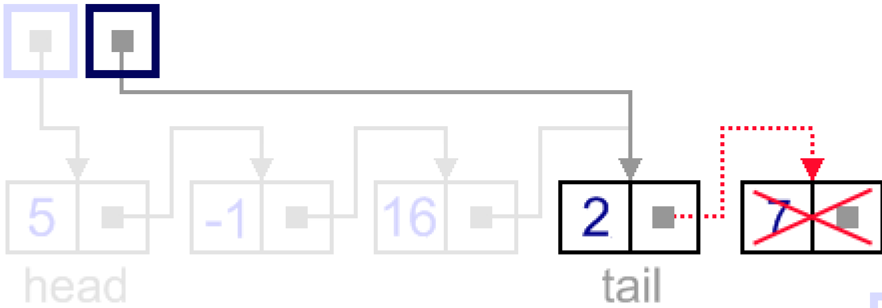
In this case, last node (current tail node) is removed from the list. This operation is a bit more tricky, than removing the first node, because algorithm should find a node, which is previous to the tail first.

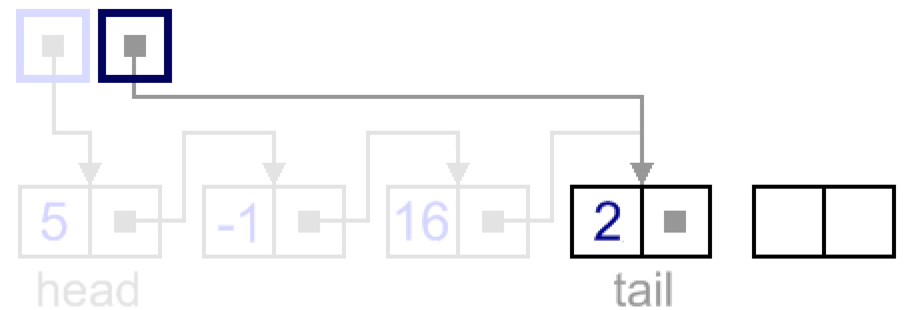
It can be done in three steps:

- Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.
- Set next link of the new tail to NULL.
- Dispose removed node.


5
-1
16
2
7
head
tail


5
-1
16
2
7
head
tail
tail


5
-1
16
2
7
head
tail


5
-1
16
2
head
tail

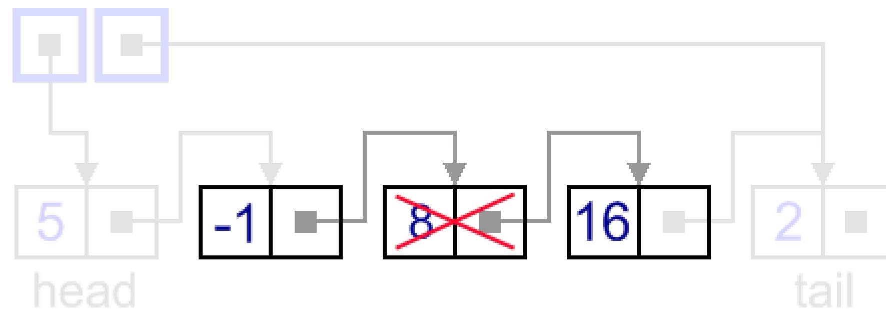
# REMOVE - GENERAL CASE

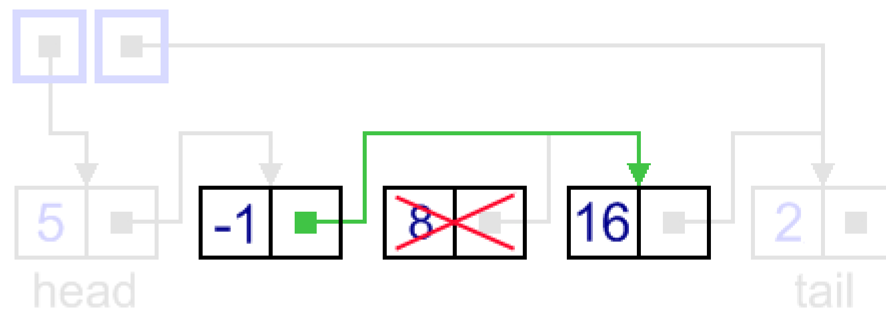
In general case, node to be removed is always located between two list nodes. Head and tail links are not updated in this case.

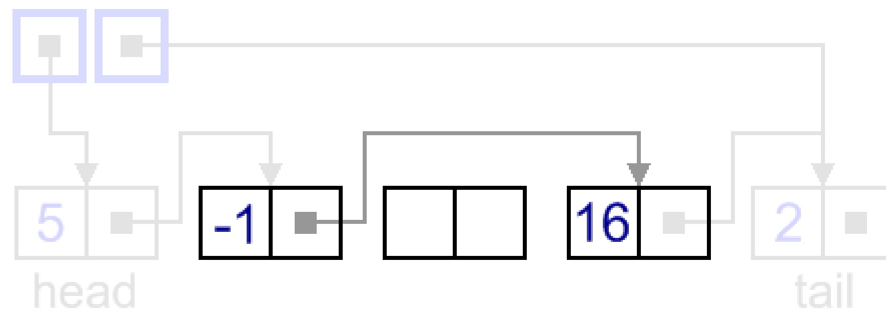
We need to know two nodes "Previous" and "Next", of the node which we want to delete.

Such a removal can be done in two steps:

- Update next link of the previous node, to point to the next node, relative to the removed node.
- Dispose removed node.


5
-1
8
16
2
head
tail


5
-1
8
16
2
head
tail


5
-1
16
2
head
tail

# ADVANTAGES OF USING LINKED LISTS

Need to know where the first node is

- the rest of the nodes can be accessed

No need to move the elements in the list for insertion and deletion operations

No memory waste

# ARRAYS - PROS AND CONS

Pros

- Directly supported by C
- Provides random access

Cons

- Size determined at compile time
- Inserting and deleting elements is time consuming

# LINKED LISTS - PROS AND CONS

Pros

- Size determined during runtime
- Inserting and deleting elements is quick

Cons

- No random access
- User must provide programming support

# APPLICATION OF LISTS

Lists can be used

To store the records sequentially

For creation of stacks and queues

For polynomial handling

To maintain the sequence of operations for do / undo in software

To keep track of the history of web sites visited

# WHY DOUBLY LINKED LIST ?

Given only the pointer location, we cannot access its predecessor in the list.

Another task that is difficult to perform on a linear linked list is traversing the list in reverse.

Doubly linked list, a linked list in which each node is linked to both its successor and its predecessor.

In such a case, where we need to access the node that precedes a given node, a doubly linked list is useful.

# DOUBLY LINKED LIST

In a doubly linked list, the nodes are linked in both directions. Each node of a doubly linked list contains three parts:

- Info: the data stored in the node
- Next: the pointer to the following node
- Back: the pointer to the preceding node

# OPERATIONS ON DOUBLY LINKED LISTS

The algorithms for the insertion and deletion operations on a doubly linked list are somewhat more complicated than the corresponding operations on a singly linked list.

The reason is clear: There are more pointers to keep track of in a doubly linked list.

# INSERTING ITEM

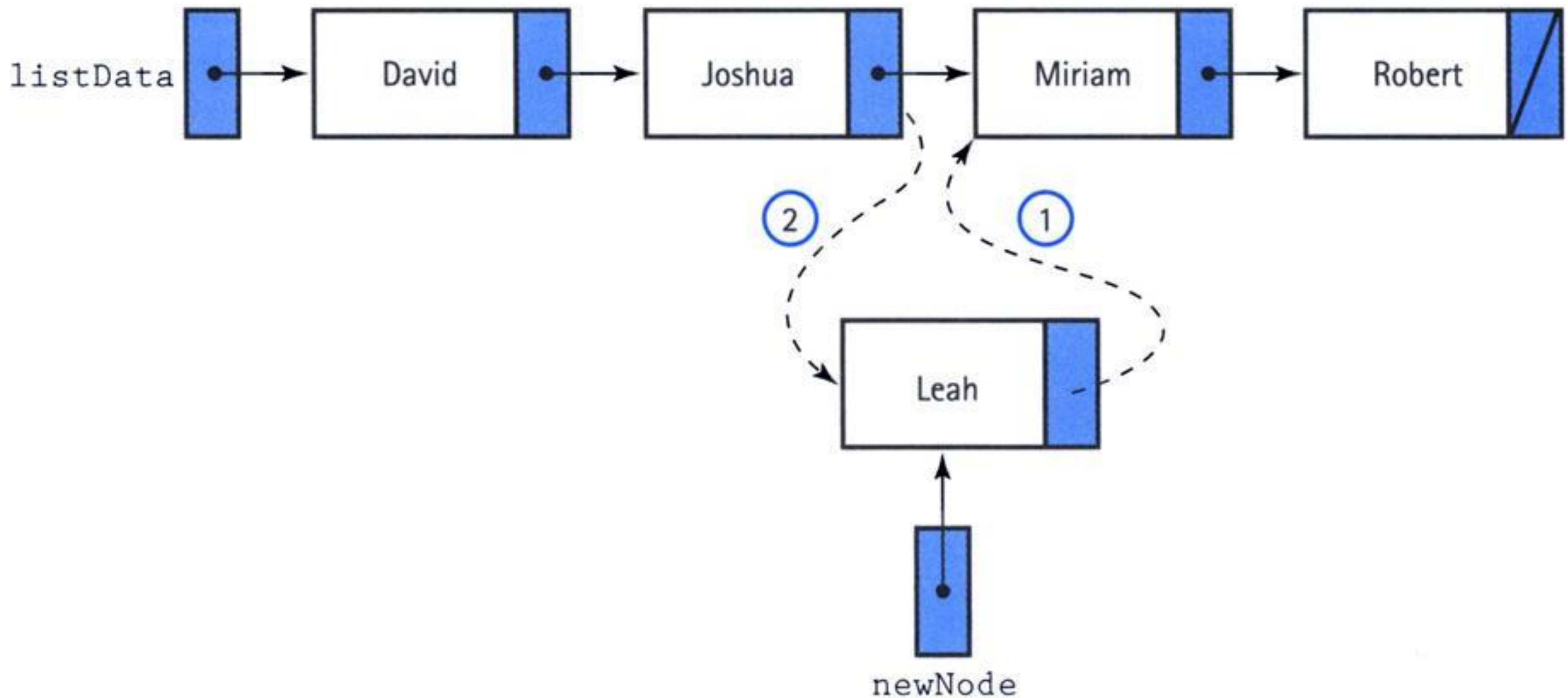
As an example, consider the Inserting an item.

To link the new node, after a given node, in a singly linked list, we need to change two pointers:

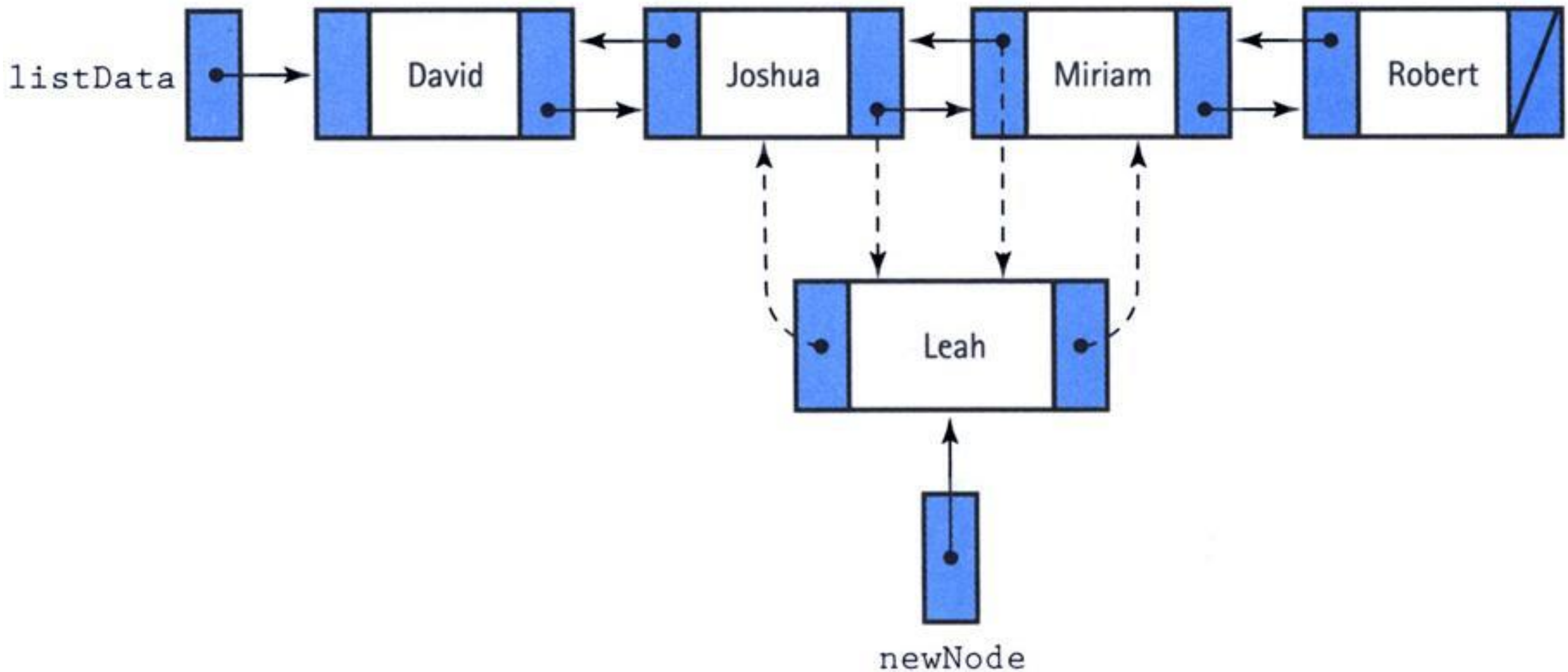
- newNode->next and
- location->next.

The same operation on a doubly linked list requires four pointer changes.

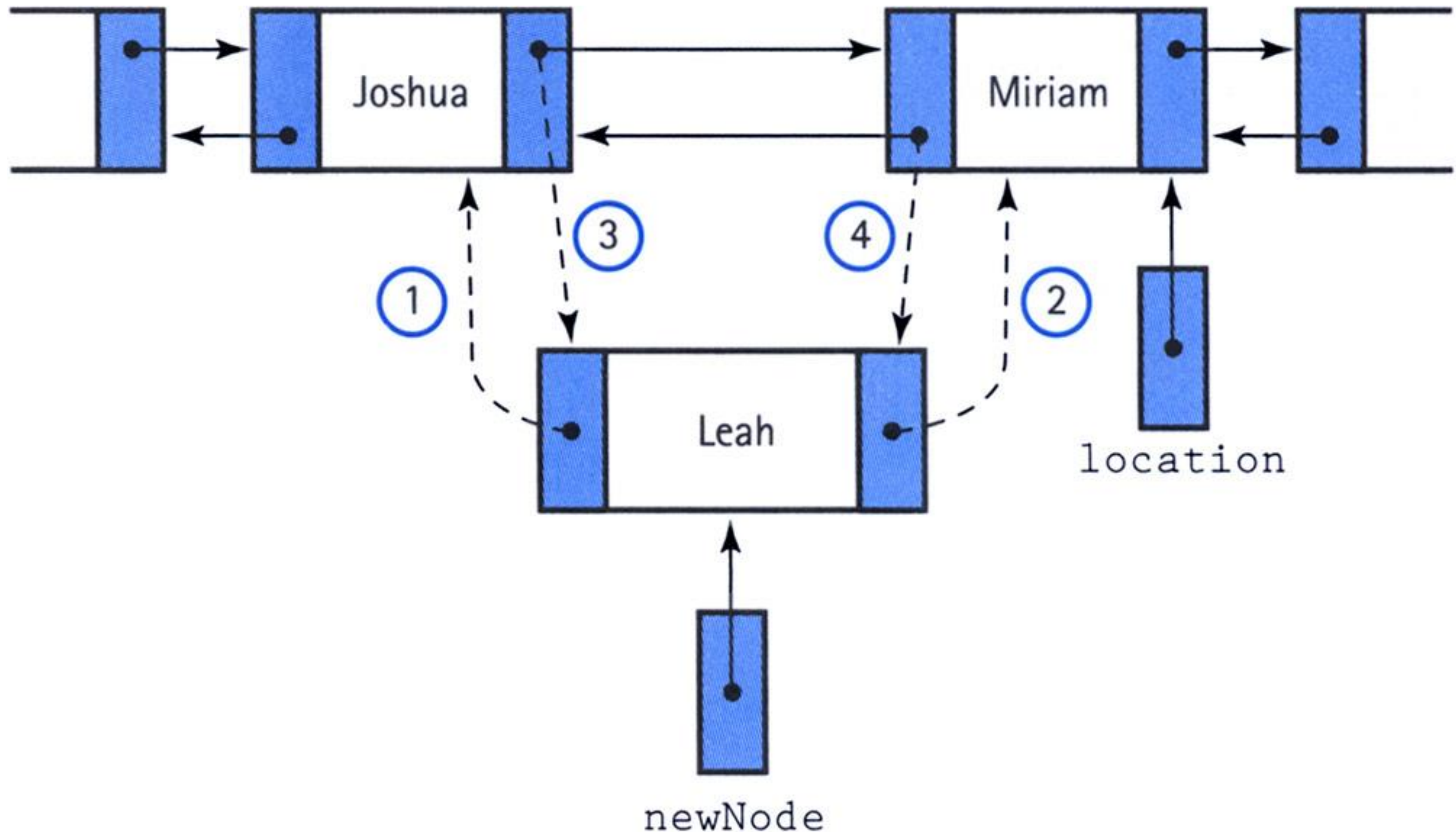
# SINGLY LINKED LIST INSERTION



# DOUBLY LINKED LIST INSERTION



# THE ORDER IS IMPORTANT



# DOUBLY LINKED LIST - DELETION

One useful feature of a doubly linked list is its elimination of the need for a pointer to a node's predecessor to delete the node.

Through the back member, we can alter the next member of the preceding node to make it jump over the unwanted node.

Then we make the back pointer of the succeeding node point to the preceding node.

# DOUBLY LINKED LIST - DELETION