19UCB206 - INTRODUCTION TO DATA STRUCTURES & ALGORITHMS

Unit - II

Mrs.P.SUGANTHI.,
AP(SG)/CSBS.,
Sethu Institute of Technology

UNIT II LINEAR DATA STRUCTURES – LIST

- Array, Linked-list and its types, Various Representations, singly linked lists- circularly linked lists- doubly-linked lists, Stack, Queue, Circular Queue, Operations & Applications of Linear Data Structures

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

1

- 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

1

- 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

1

- 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
- These are “set in stone”.

ADT Definition [Wikipedia]

1

- 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 [Wikipedia]

1

- An ADT may be implemented by specific data types or data structures, in many ways and in many a 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.

ADT in Simple Words

1

- 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?

1

- ***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

1

- 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

1

- 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

2

- MakeEmpty
- PrintList
- Find
- FindKth
- Insert
- Delete
- Next
- Previous

List – An Example

2

- 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

2

- Lists can be implemented using:
  - Arrays
  - Linked List
  - Cursor [Linked List using Arrays]

Array

2

s

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

<u>or</u>

- A ***fixed-size indexed sequence*** of elements, all of the same type.
- Theindividual elements aretypically storedin consecutive memory locations.
- The ***length of the array is determined*** when the array is ***created***, and ***cannot be changed.***

Array

2

S

- Any ***component of the array*** can be ***inspected or updated by using its index.***
  - This is an efficient operation
  - O(1) = constant time
- Thearray 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

2

- **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

2

- 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

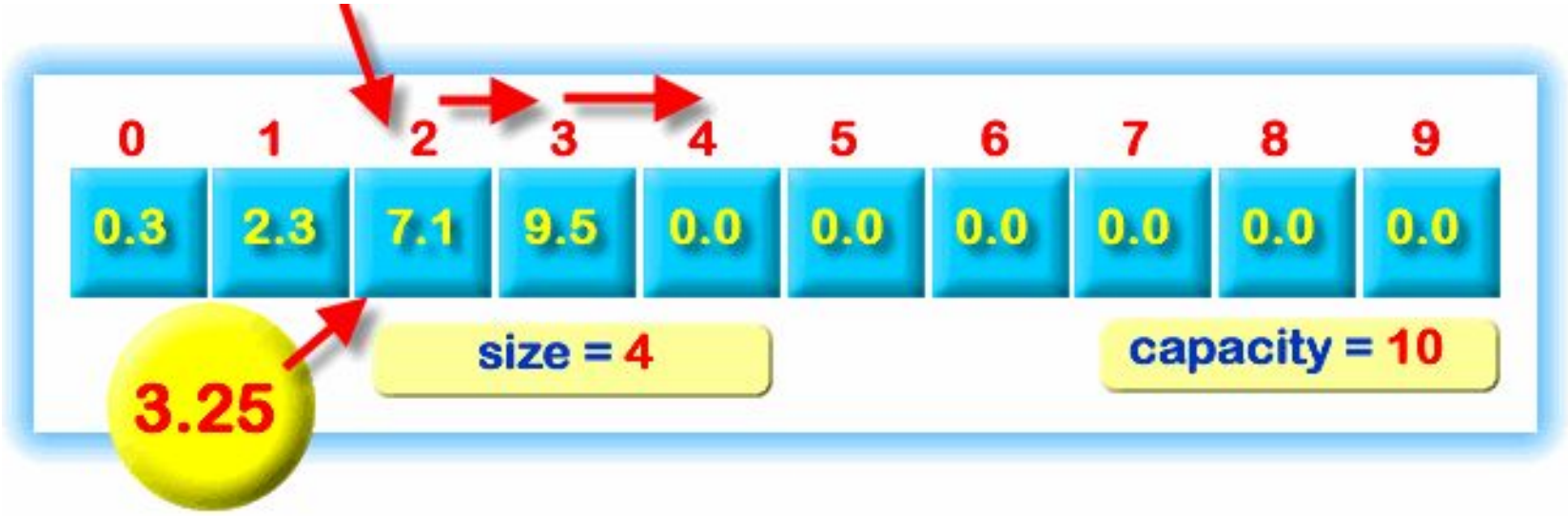

```
void display(int array[],int num_values)
{
  for (int I = 0; i<num_values; i++)
      cout<< array[i] << " ";
}
```

List Implemented Using Array

Operations On Lists

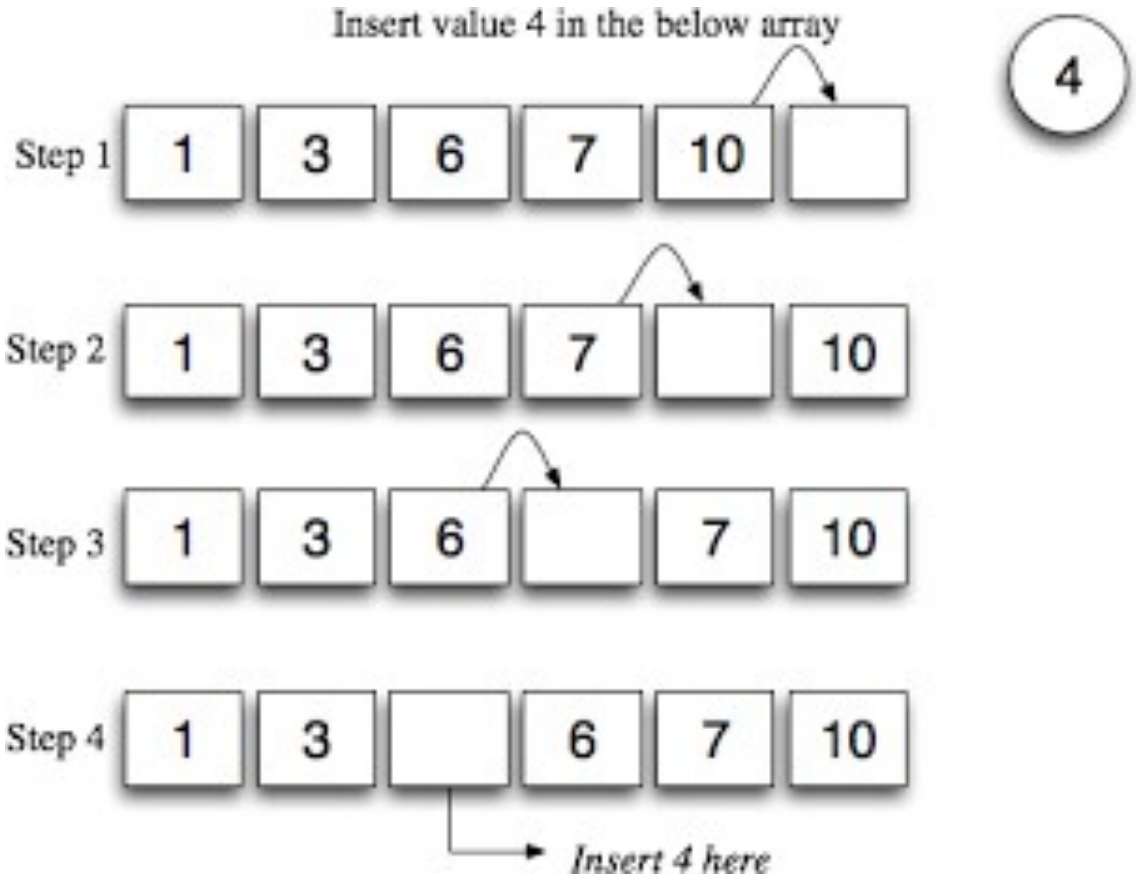
- We‟ll consider only few operations and not all operations on Lists
- Let us consider *Insert*
- There are two possibilities:
  - *Ordered List*
  - *Unordered List*

Insertion into an Ordered List

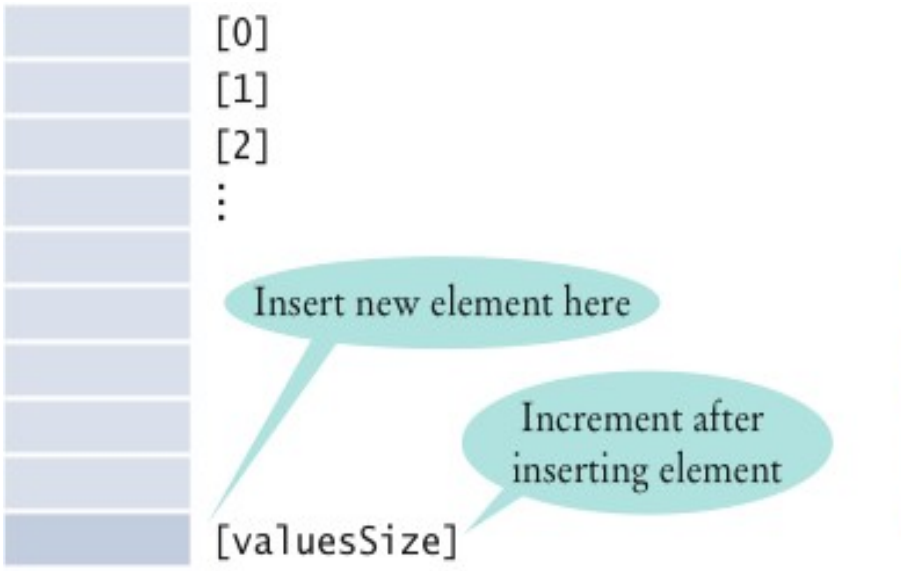


Insertion in Detail

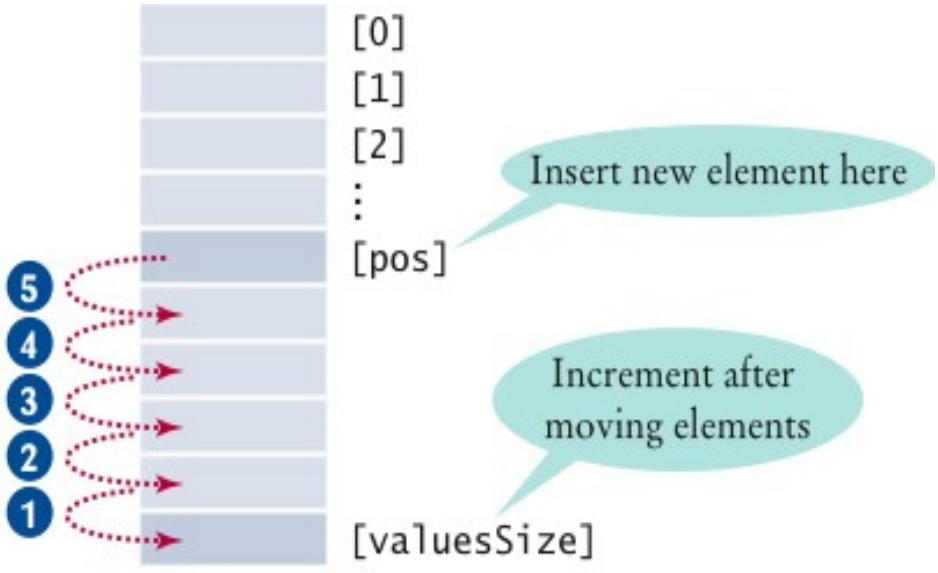
3



Insertion

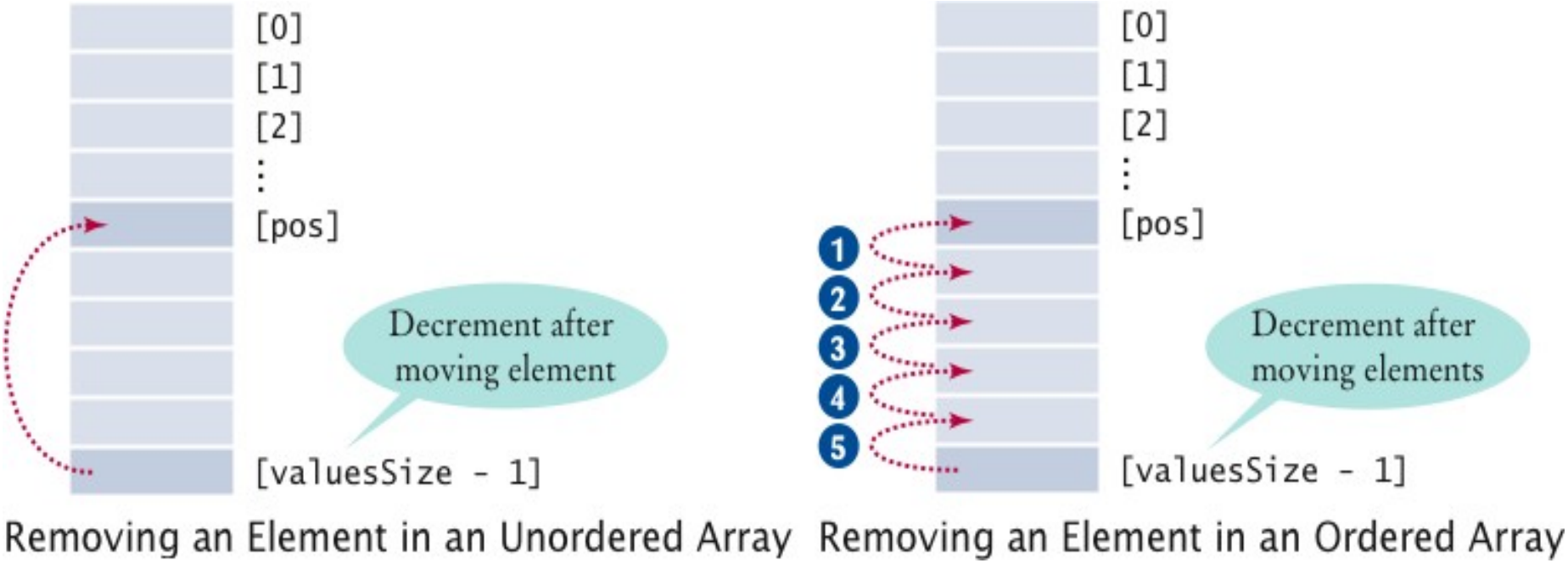


Inserting an Element in an Unordered Array



Inserting an Element in an Ordered Array

Deletion



Removing an Element in an Unordered Array Removing an Element in an Ordered Array

Find /Search

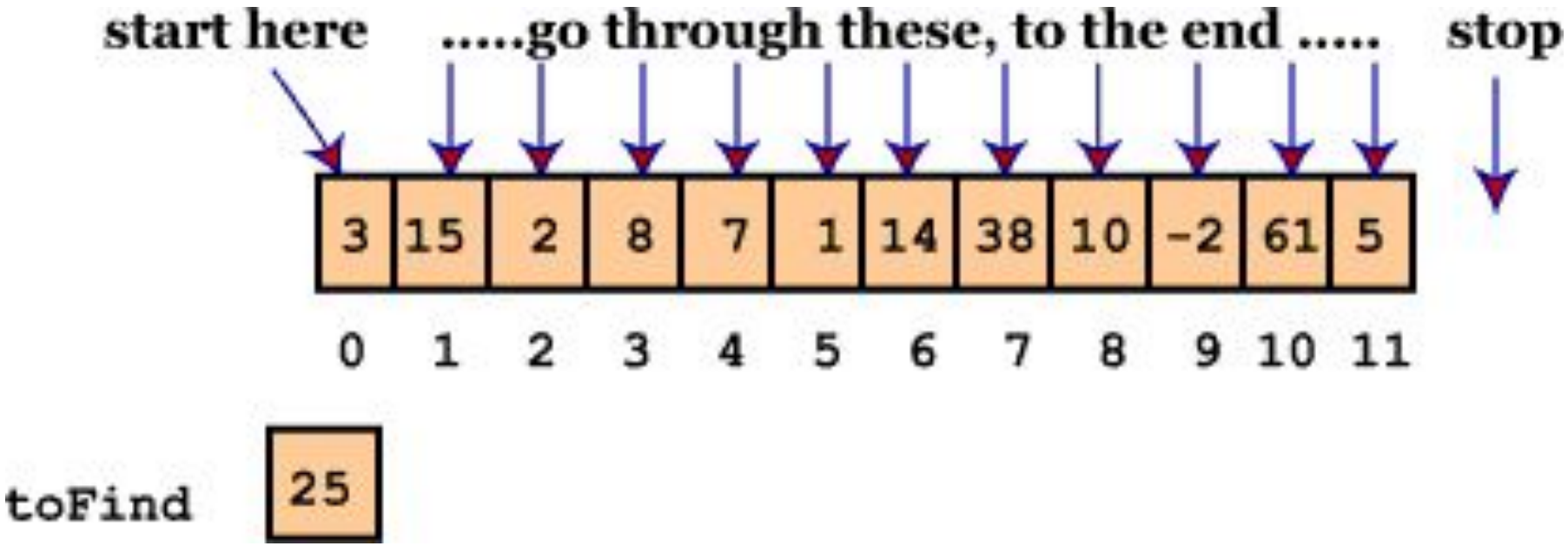
- Searching is the *process of looking for a specific element* in an array
- For example, discovering whether a certain score is included in a list of scores.
- *Searching, like sorting*, is a *common task* in computer programming.
- There are *manyalgorithms and data structures devoted to searching.*
- The most common one is the *linear search*.

Linear Search

3

- 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)`**

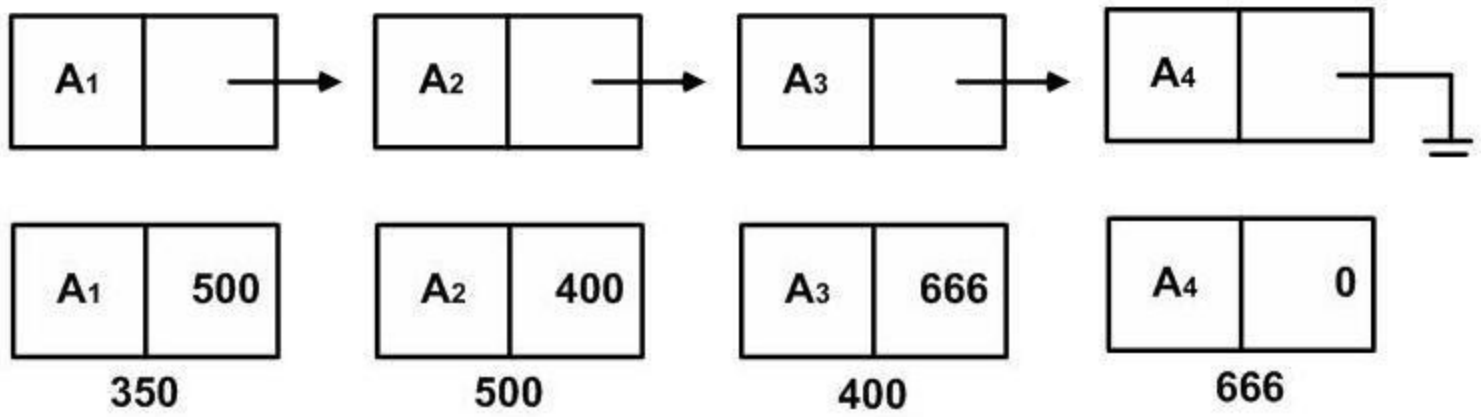
Disadvantages of Using Arrays

4

- 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

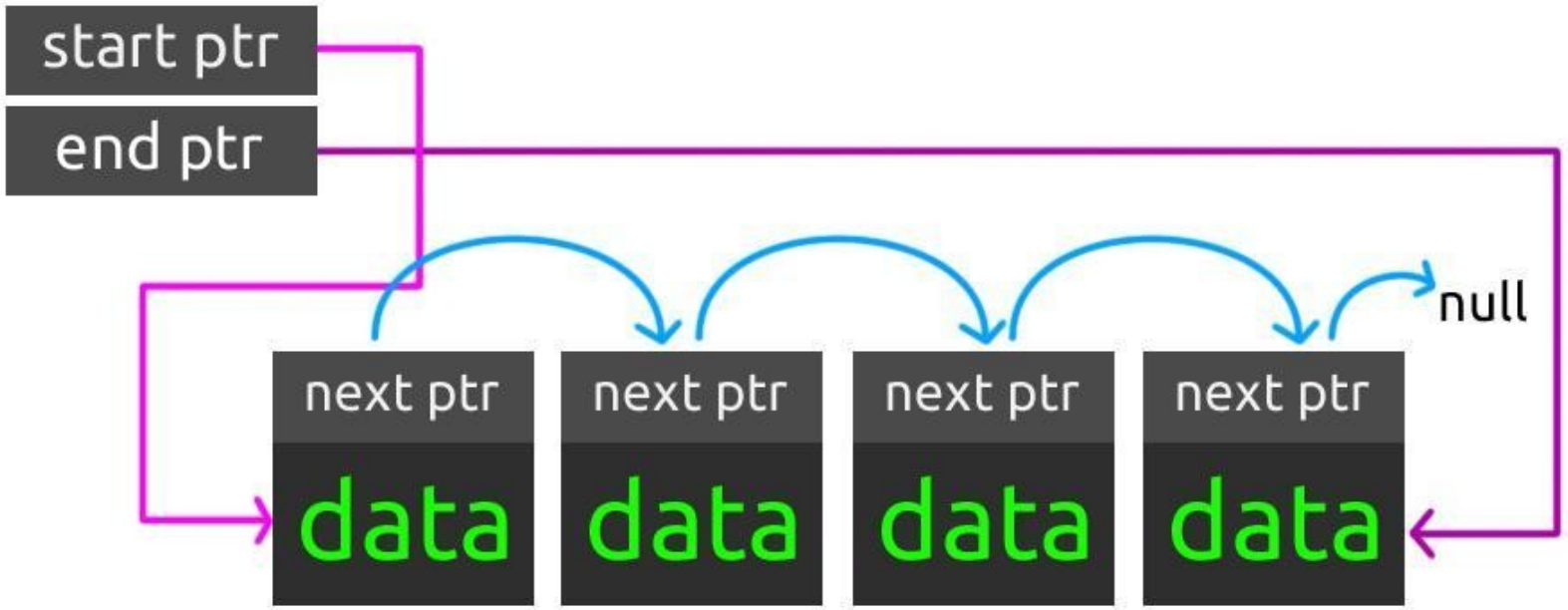
Linked List

- 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!

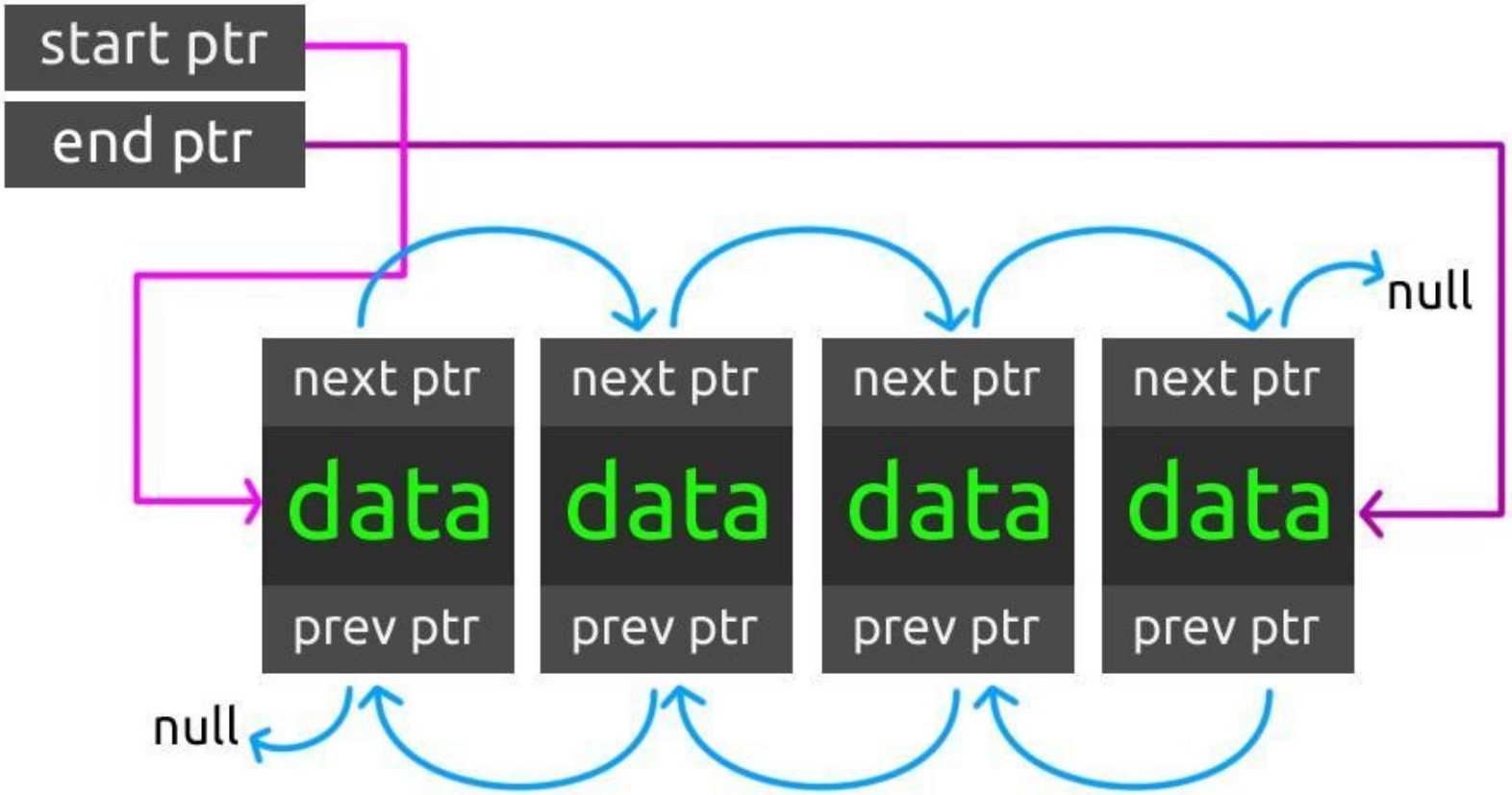


Singly Linked List

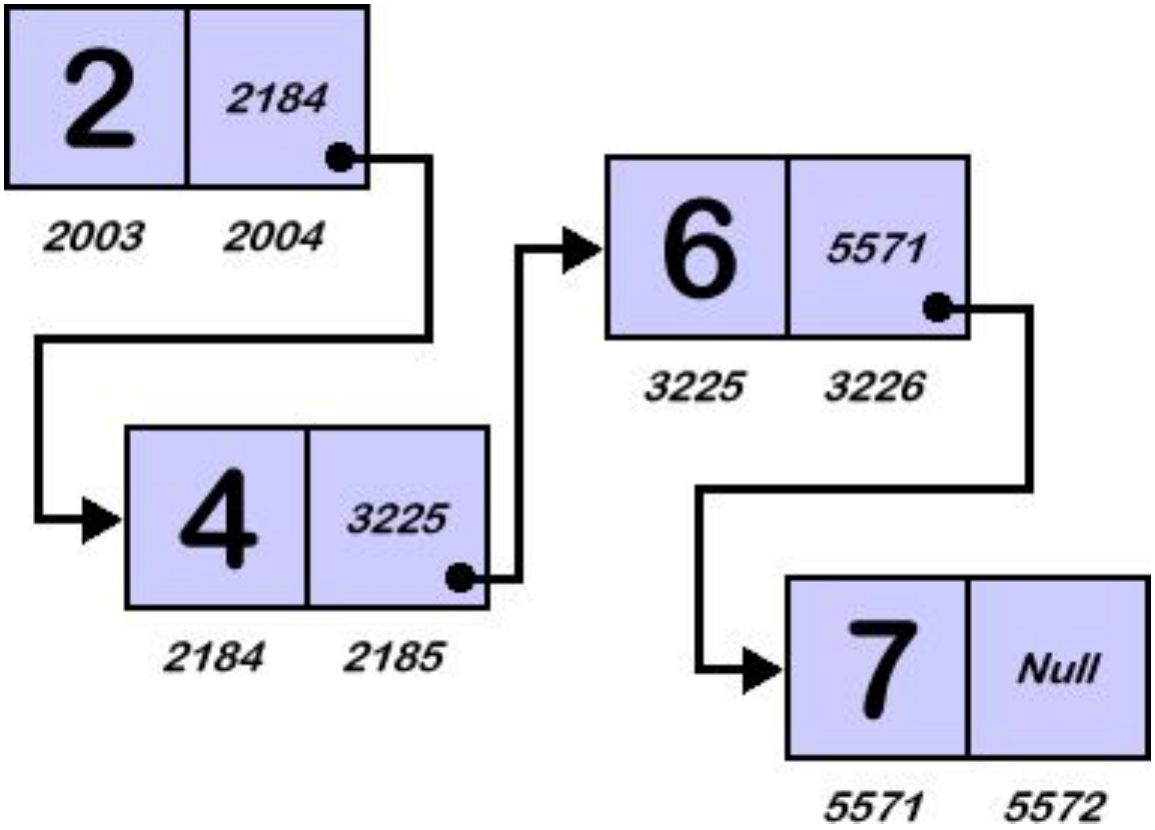
4



Doubly Linked List



Singly Linked List



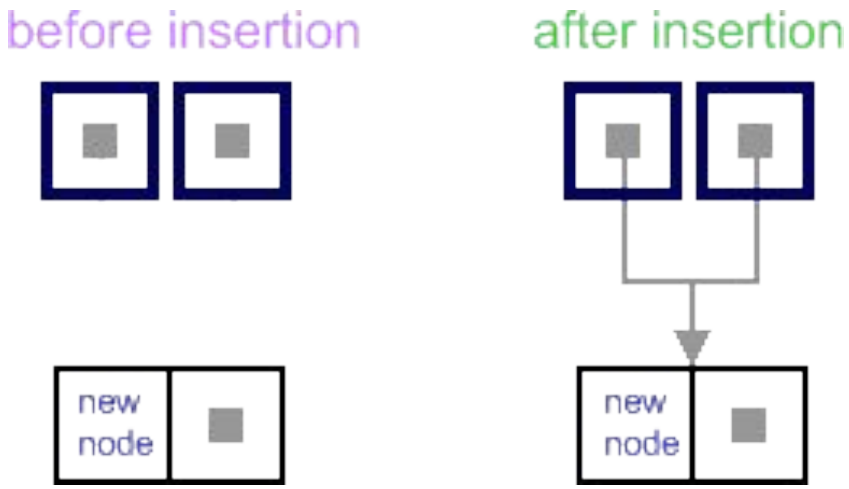
Singly-linked List - Addition

- 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

5

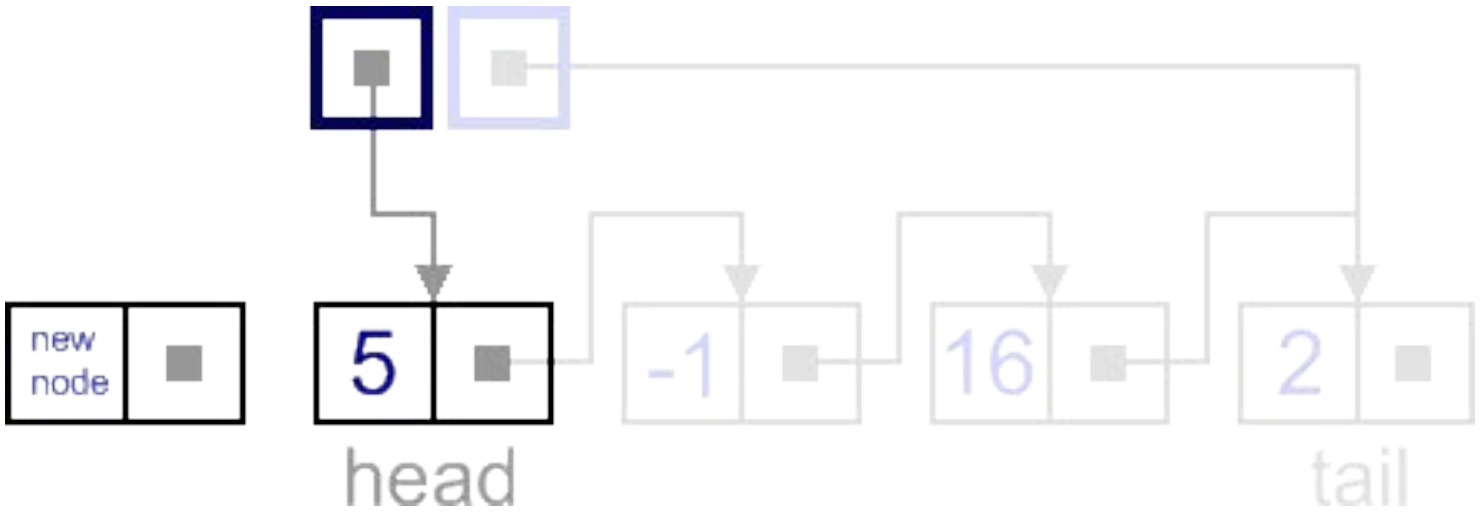
- When list is empty, which is indicated by (head == NULL) condition, the insertion is quite simple.
- Algorithm sets both head and tail to point to the new node.



Add first

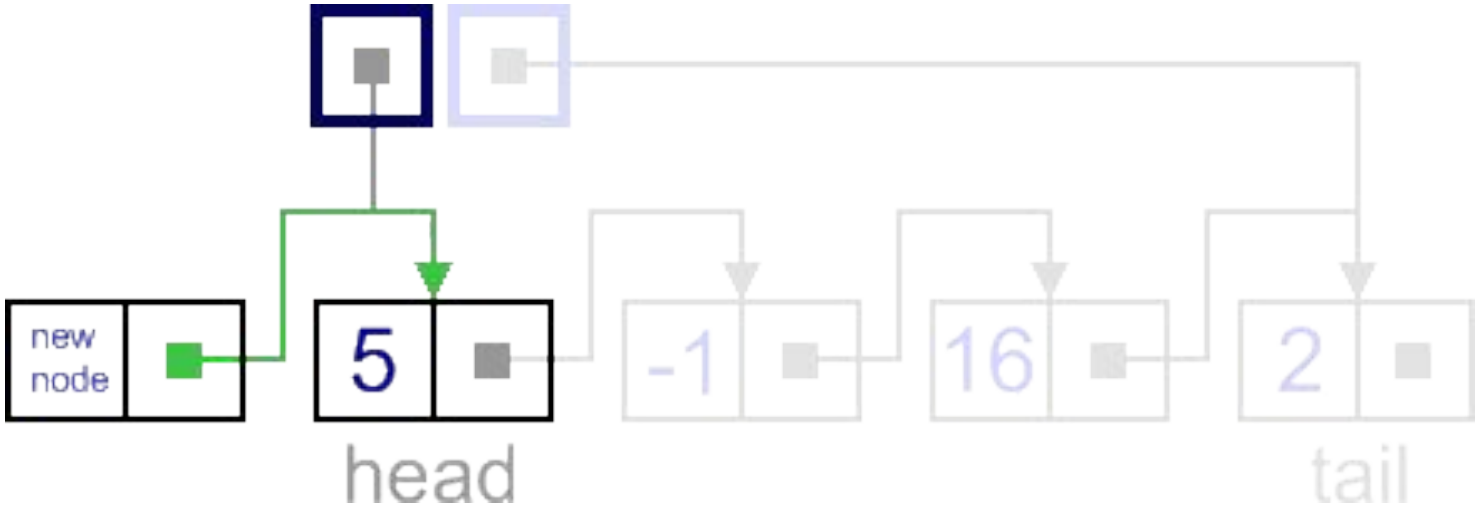
5

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



Add First - Step 1

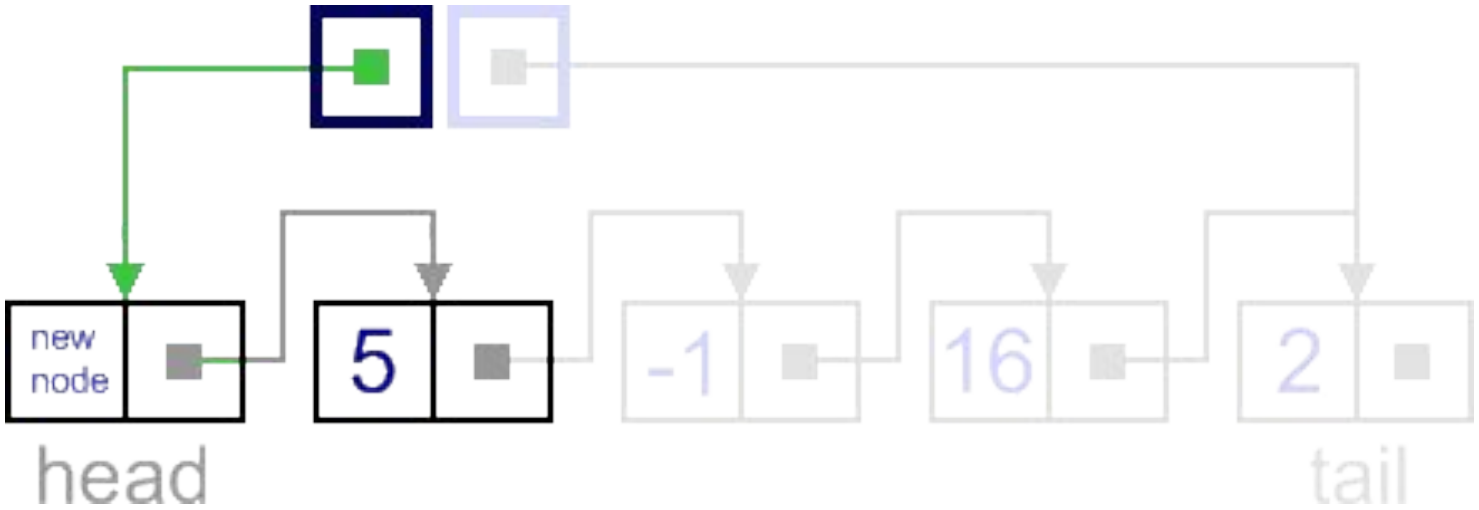
- It can be done in two steps:
  - Update the next link of the new node, to point to the current head node.



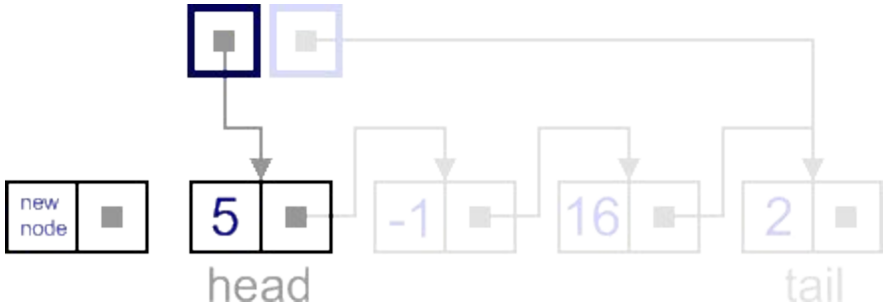
Add First - Step 2

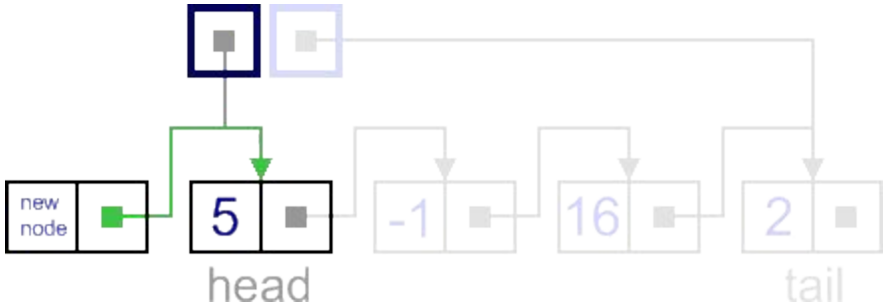
5

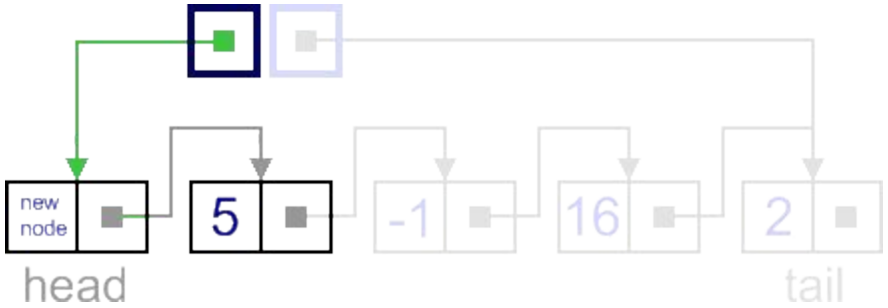
- Update head link to point to the new node.



5


new
node
5
-1
16
2
head
tail

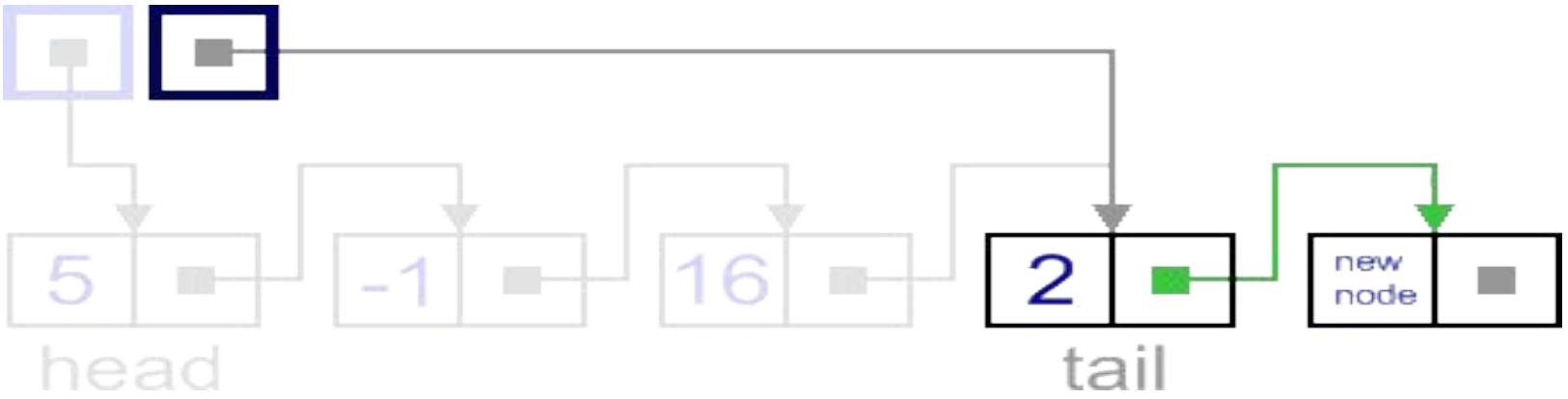

new
node
5
-1
16
2
head
tail


new
node
5
-1
16
2
head
tail

Add last

5

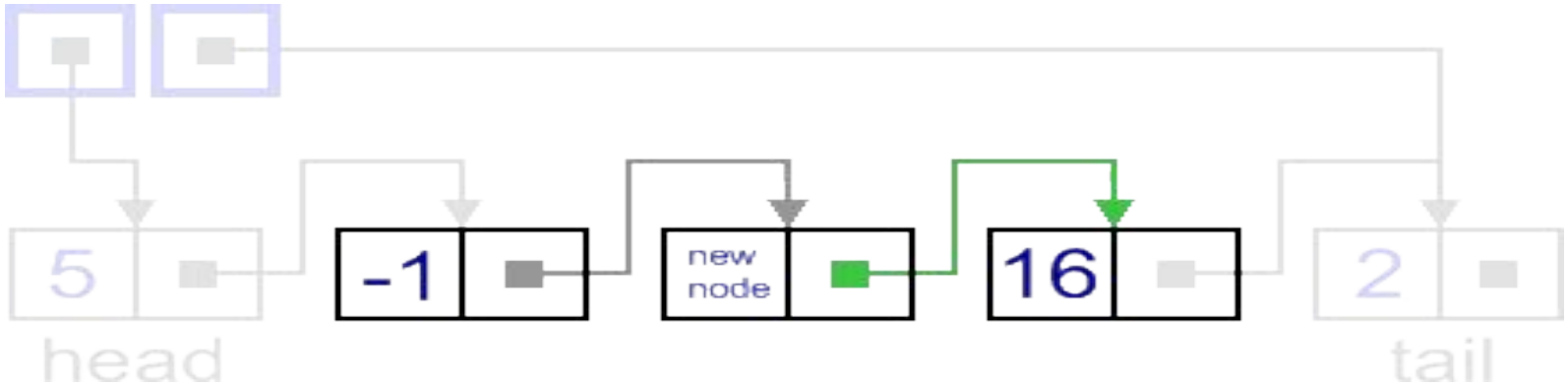
- In this case, *new node is inserted right after the current tail node.*
- It can be done in *two steps*:
  - *Update the next link of the current tail node*, to point to the new node.
  - *Update tail link to point to the new node*.


5
-1
16
2
new
node
head
tail

Insert - General Case

5

- 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
head
-1
new
node
16
2
tail

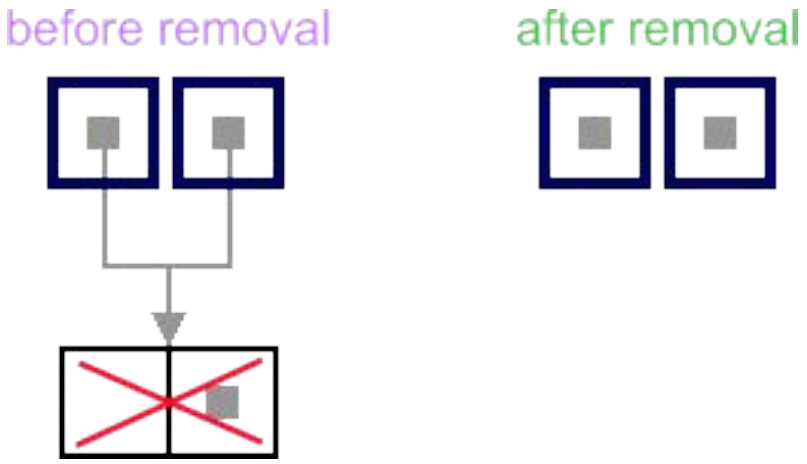
Singly-linked List - Deletion

5

- 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

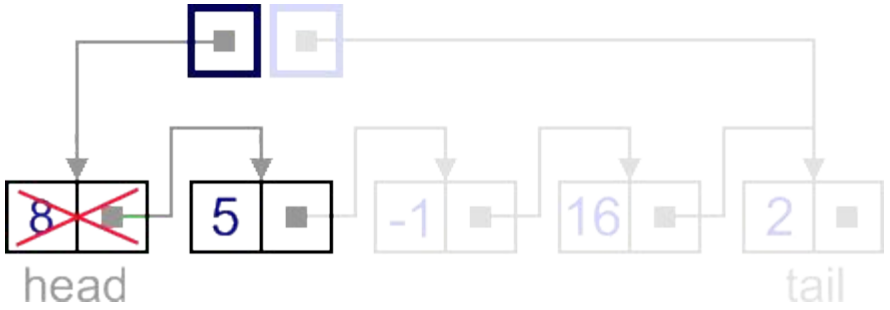
- When list has only one node, that the *head points to the same node as the tail*, the removal is quite simple.
- Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to NULL.

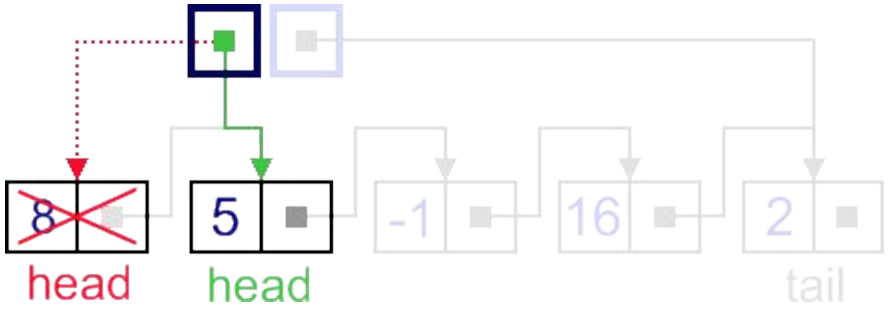


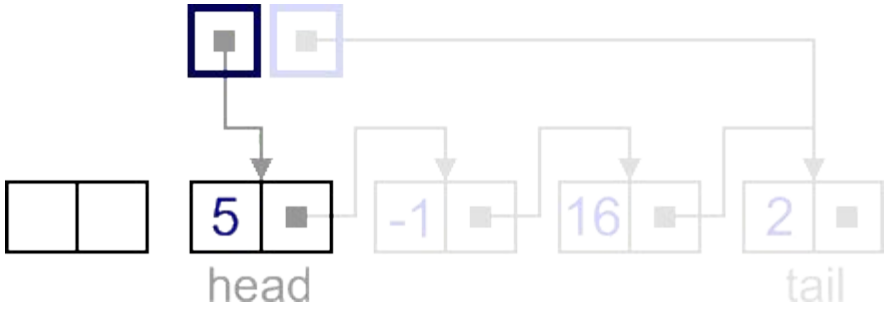
Remove First

6

- 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

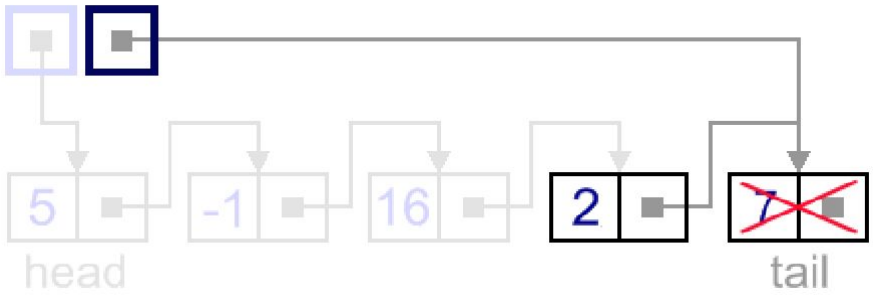

8
5
-1
16
2
head
head
tail

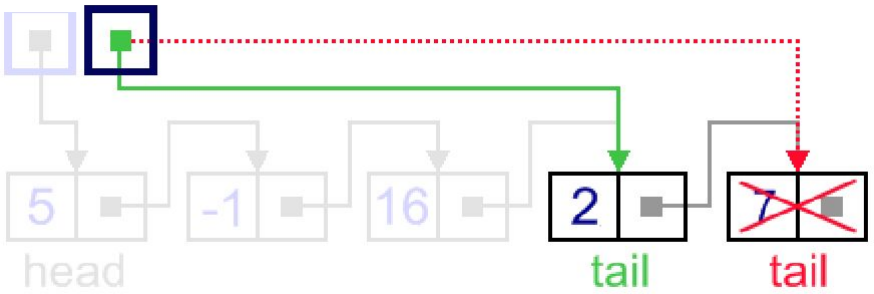

5
-1
16
2
head
tail

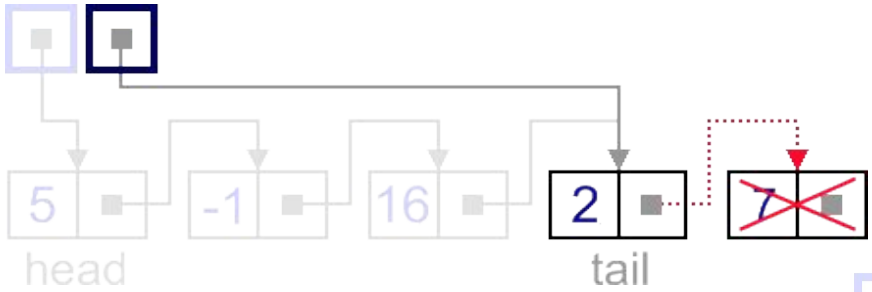
Remove Last

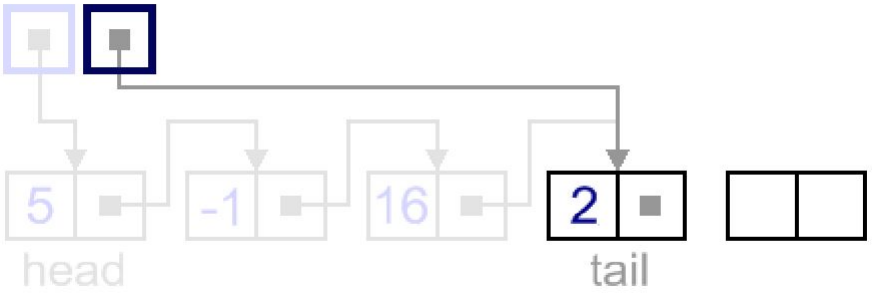
6

- 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

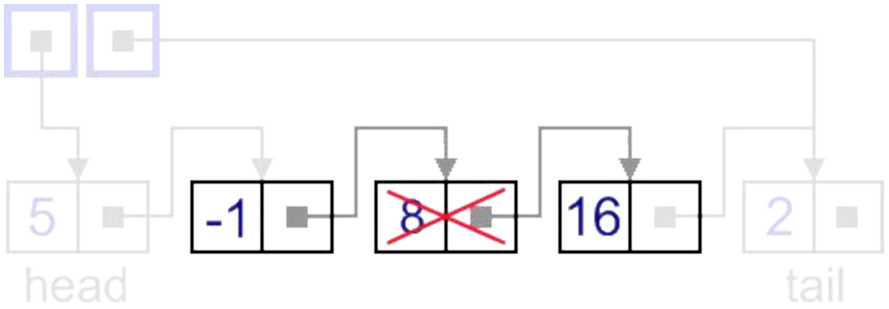

5
-1
16
2
7
head
tail
tail

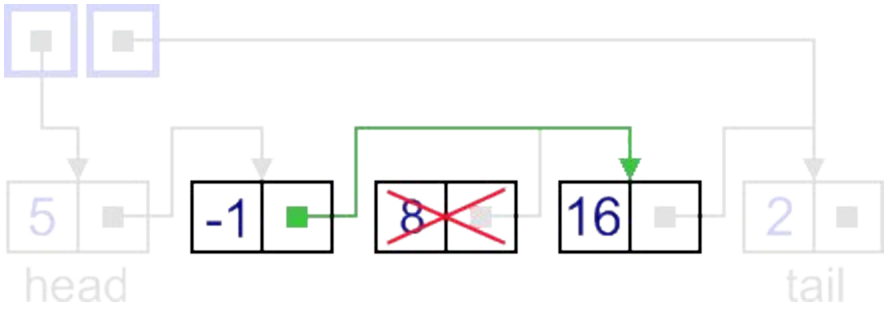

5
-1
16
2
7
head
tail

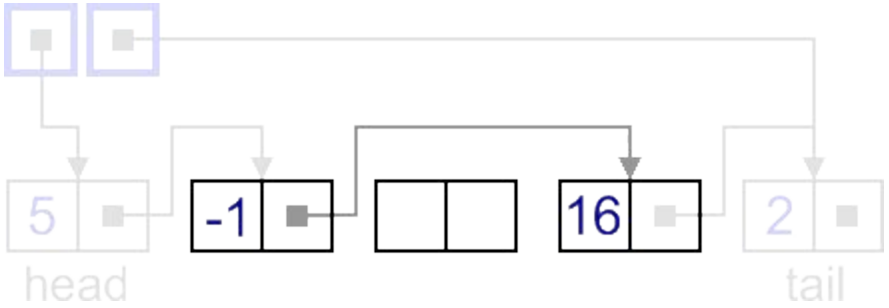

5
-1
16
2
head
tail

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
head
-1
8
16
2
tail


5
head
-1
8
16
2
tail


5
head
-1
16
2
tail

Advantages of Using Linked Lists

6

- Need to *know where the first node* is
  - the rest of the nodes can be accessed
- *No need to movethe elements* in thelist 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

6

- Pros
  - Size determined during runtime
  - Inserting and deleting elements is quick
- Cons
  - No random access
  - User must provide programming support

Application of Lists

7

- 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* thatis 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

7

- 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

7

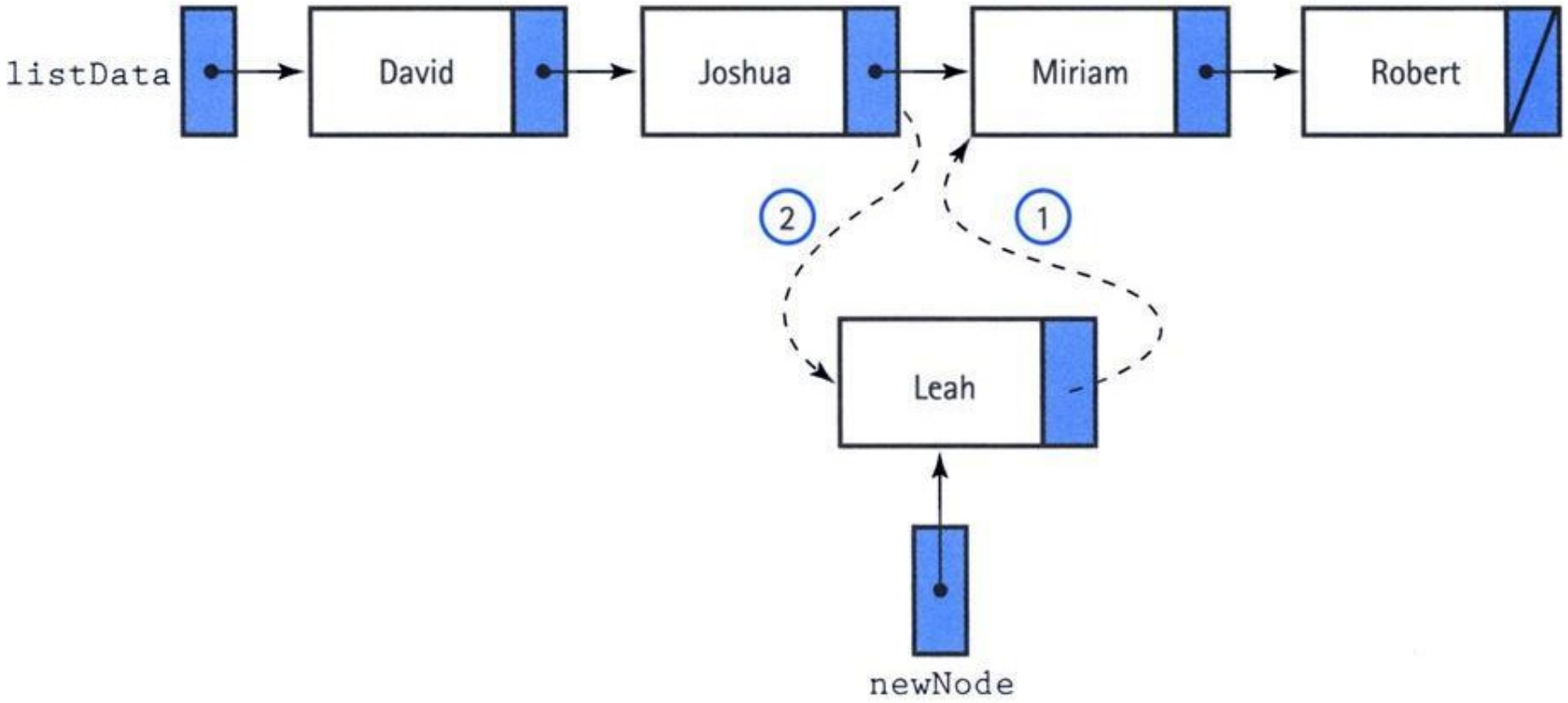
- The algorithms for the insertion and deletion operations on a doubly linked list are somewhat more complicated than the corresponding operations on a singly linked list.
- The reason is clear: There are more pointers to keep track of in a doubly linked list.

Inserting Item

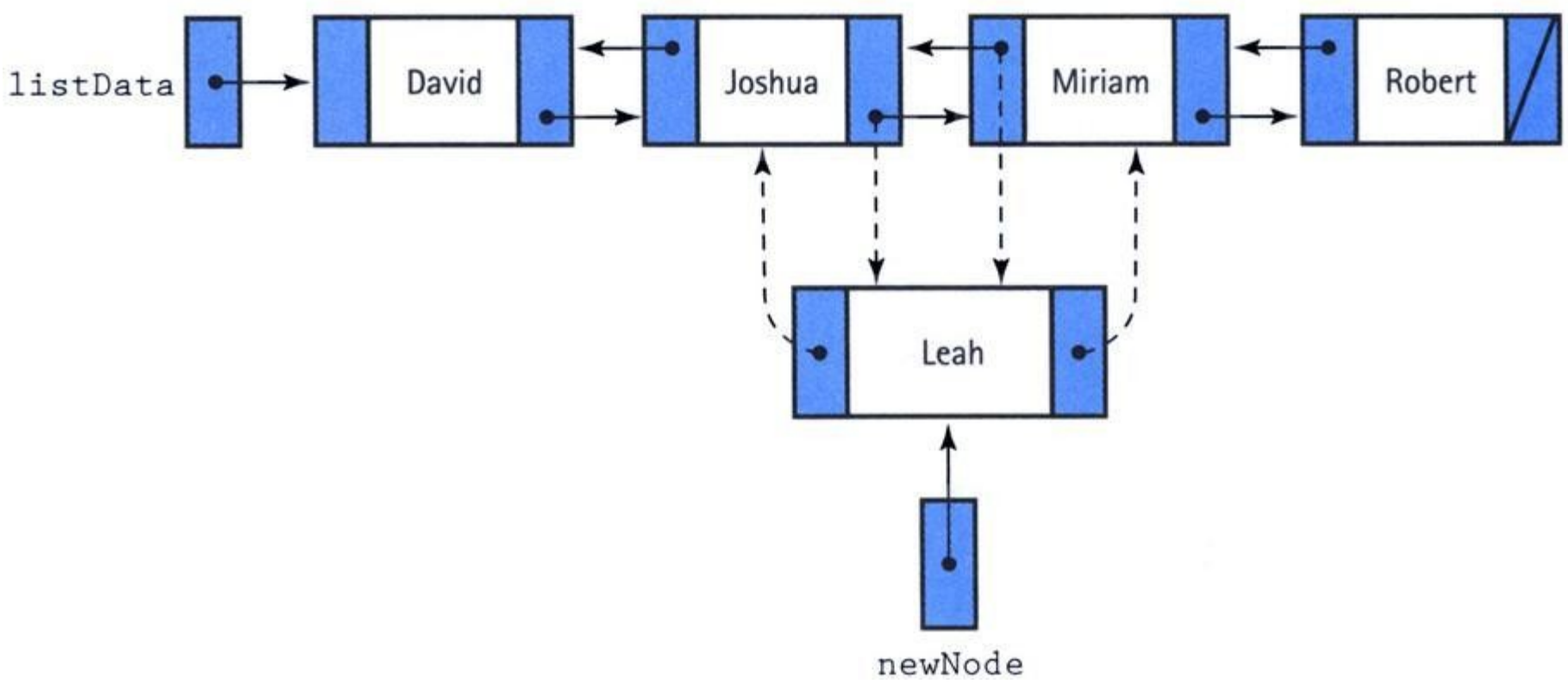
7

- As an example, consider the Inserting an item.
- To link thenew 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.

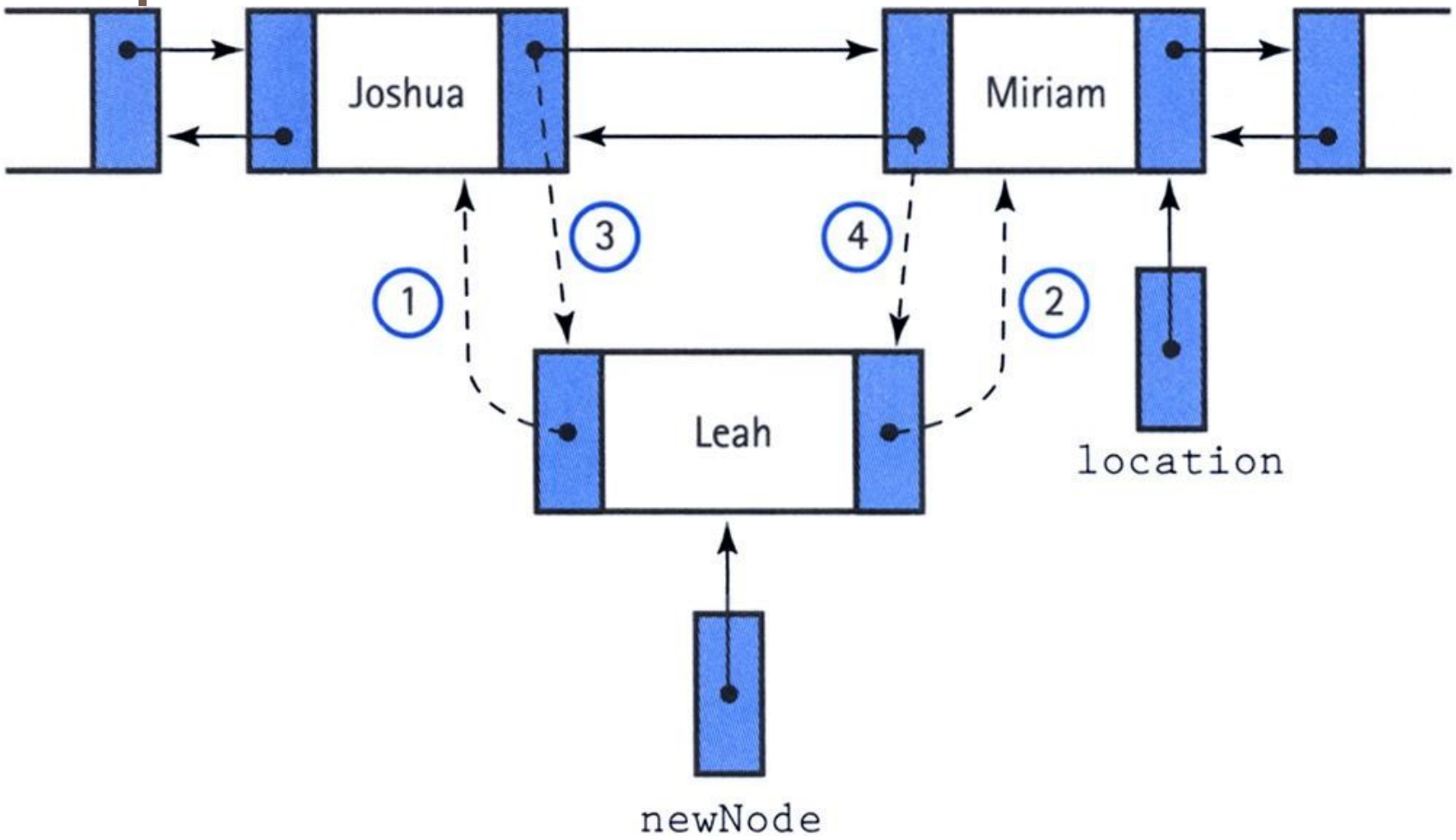
Singly Linked List Insertion



Doubly Linked List Insertion



The Order is Important

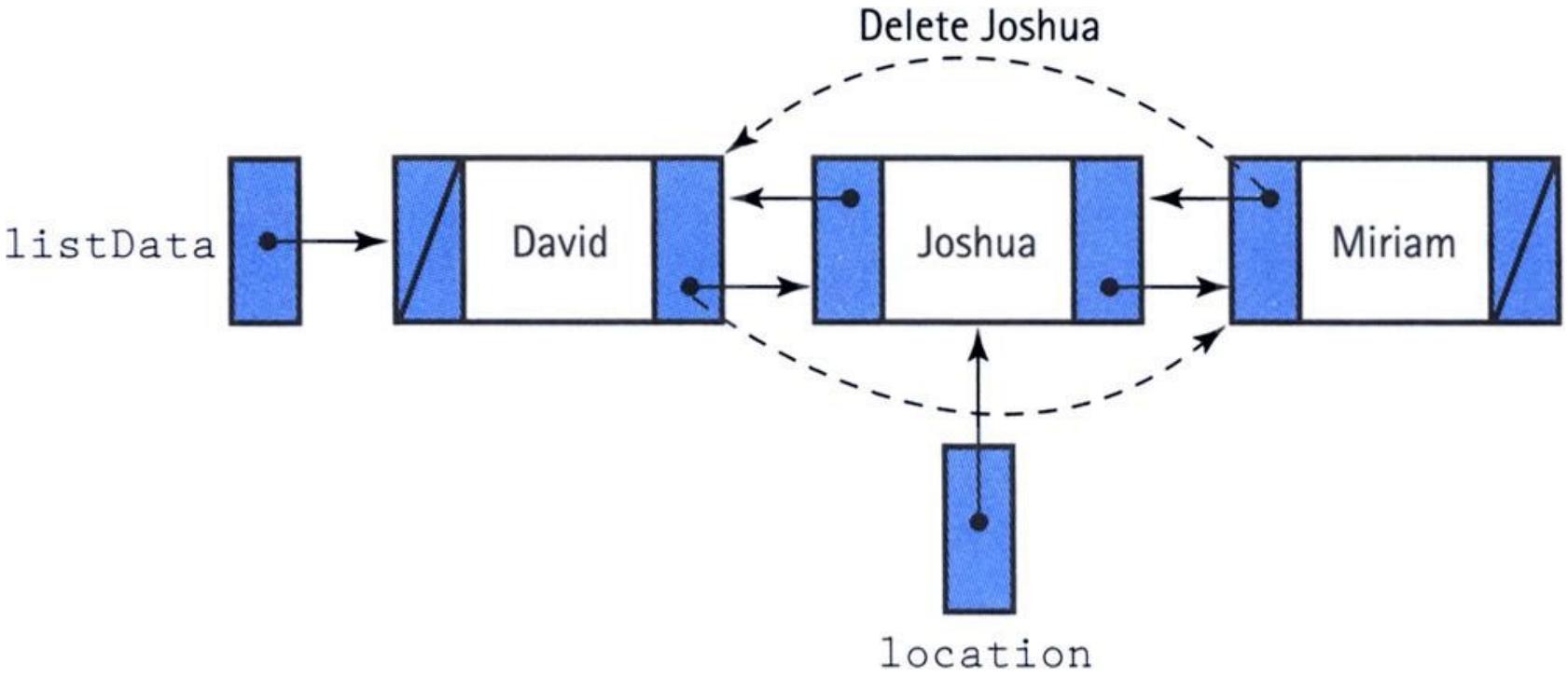


Doubly Linked List - Deletion

7

- 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



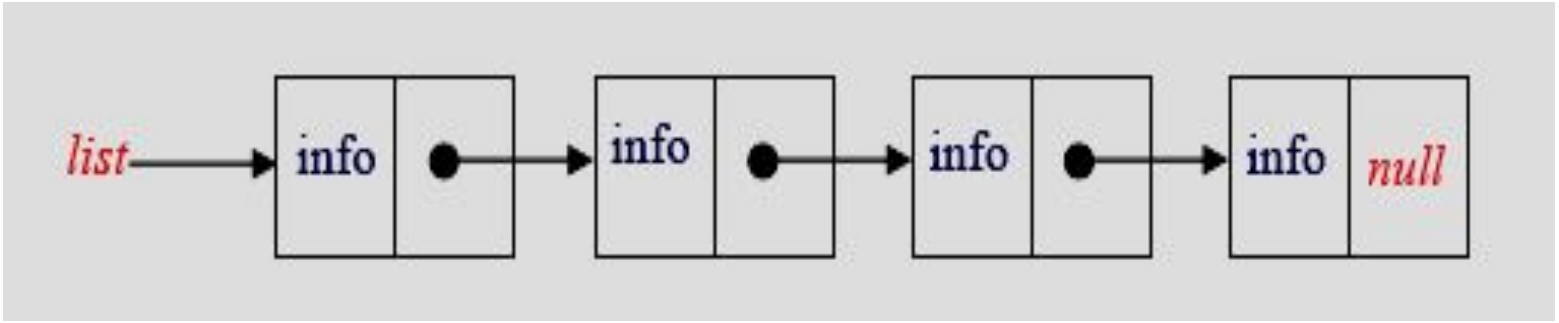
Special Cases of Deletion

- We do, however, have to be careful about the end cases:
  - If location->back is NULL, we are deleting the first node
  - if location->next is NULL, we are deleting the last node.
  - If both location->back and location->next are NULL, we are deleting the only node.

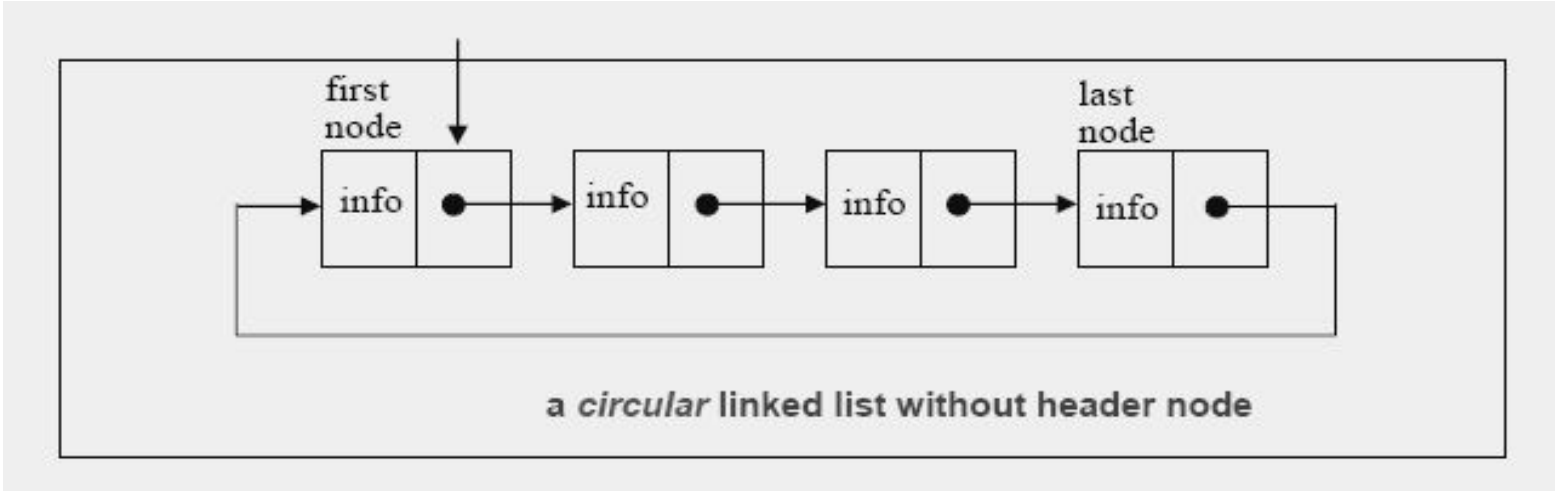
Circular Linked Lists

- In linear linked lists if a list is traversed (all the elements visited) an external pointer to the list must be preserved in order to be able to reference the list again.
- Circular linked lists can be used to *help the traverse the same list again and again* if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.

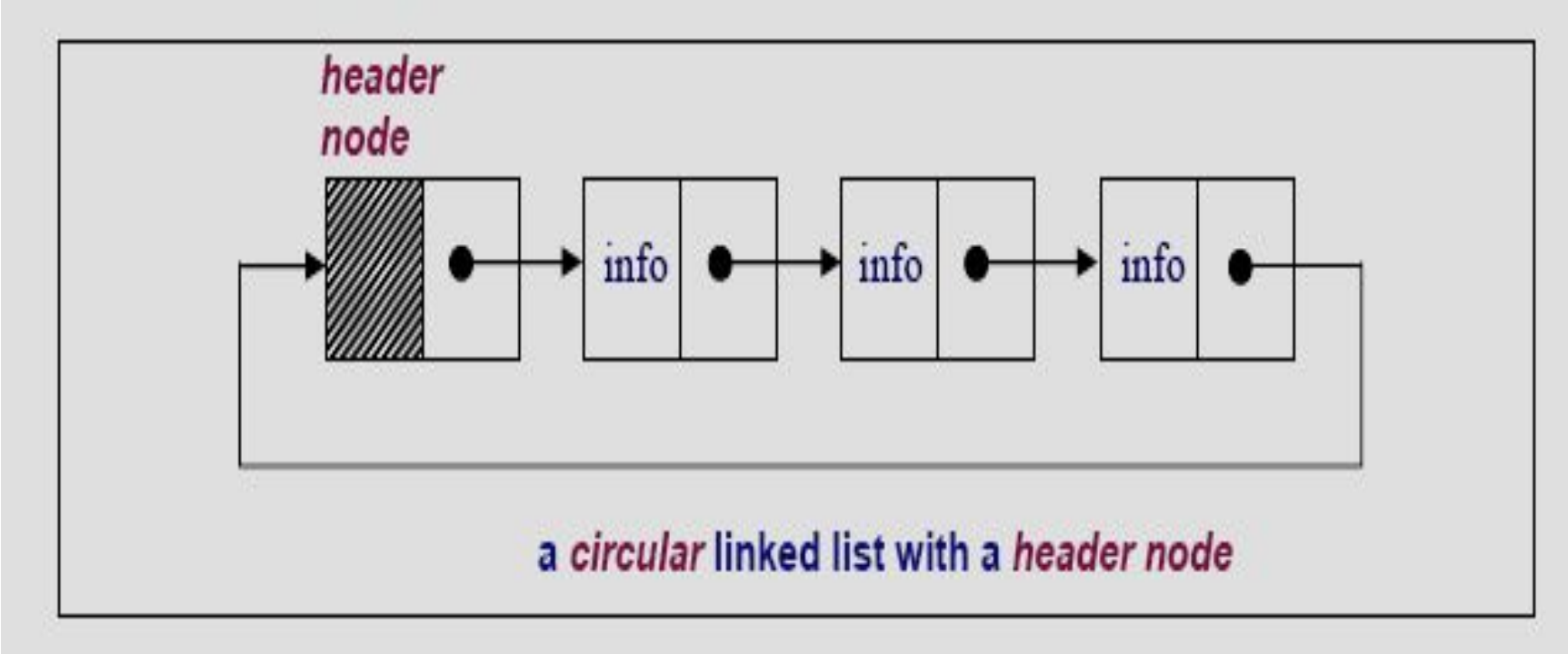
Circular Linked Lists



A Linear Linked



Circular Linked Lists



Circular Linked Lists

- In a circular linked list there are *two methods* to know if a node is the first node or not.
  - Either a external pointer, ***list***, points the first node or
  - A ***header node*** is placed as the first node of the circular list.
- The header node can be separated from the others by either heaving a ***sentinel value*** as the info part or having a dedicated ***flag*** variable to specify if the node is a header node or not.

PRIMITIVE FUNCTIONS IN CIRCULAR LISTS

- The structure definition of the circular linked lists and the linear linked list is the same:

```
struct
    node{
    int info;
    struct node *next;
};
typedef struct node *NODEPTR;
```

PRIMITIVE FUNCTIONS IN CIRCULAR LISTS

- The delete after and insert after functions of the linear lists and the circular lists are almost the same.

The delete after function***: delafter( )***

```
void delafter(NODEPTR p, int *px)
{
NODEPTR q;
if((p == NULL) || (p == p->next)){ /*the
empty list contains a single node and may be
pointing itself*/ printf("void deletion\n");
exit(1);
}
q = p->next;
*px = q->info; /*the data of the deleted node*/
p->next = q->next;
freenode(q);
}
```

PRIMITIVE FUNCTIONS IN CIRCULAR LISTS

- The insertafter function***: insafter( )***

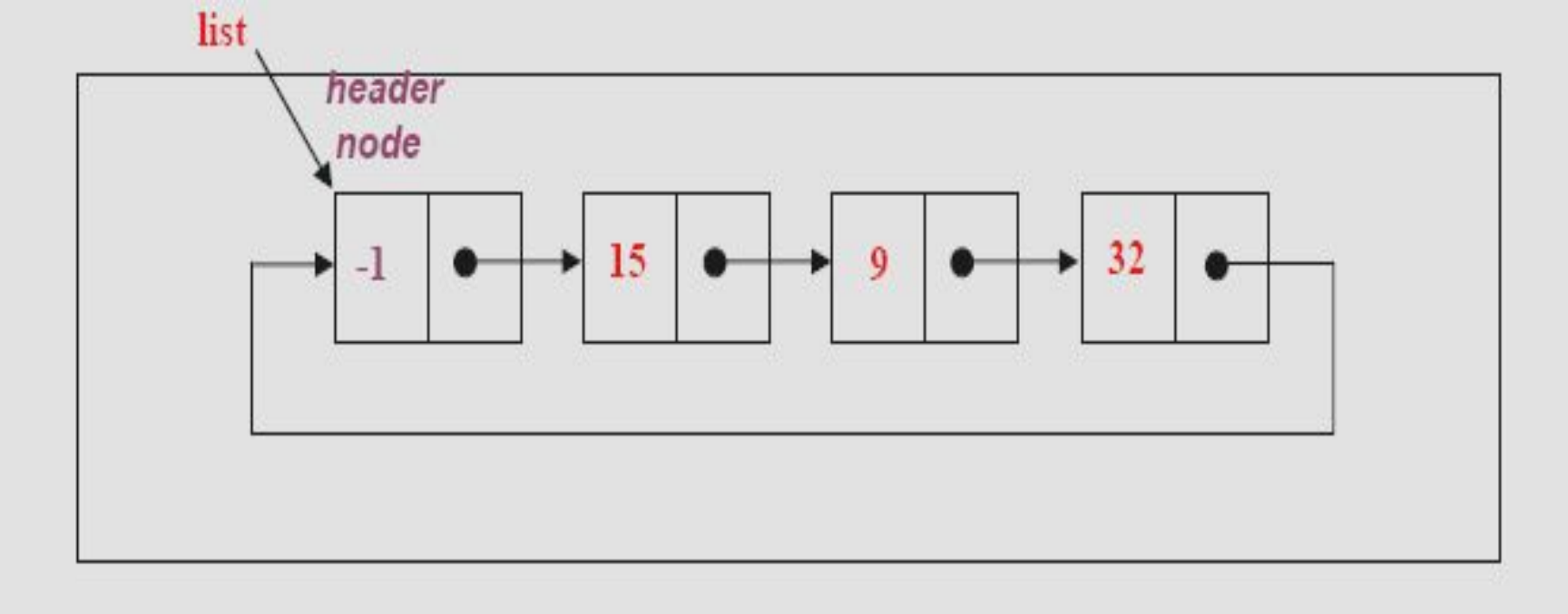
```
void insafter(NODEPTR p, int x)
{
NODEPTR q;
if(p ==
NULL){
printf("void
insertion\n");  exit(1);
}
q = getnode();
q->info = x; /*the data of the inserted
node*/  q->next = p->next;
p->next = q;
}
```

CIRCULAR LIST with header node

- The header node in a circular list can be specified by a ***sentinel value*** or a dedicated ***flag***:
- ***Header Node with Sentinel:*** Assume that info part contains positive integers. Therefore the info part of a header node can be -1. The following circular list is an example for a sentinel used to represent the header node:

```
struct node{
        int info;
        struct node *next;
};
typedef struct node *NODEPTR;
```

CIRCULAR LIST with header node

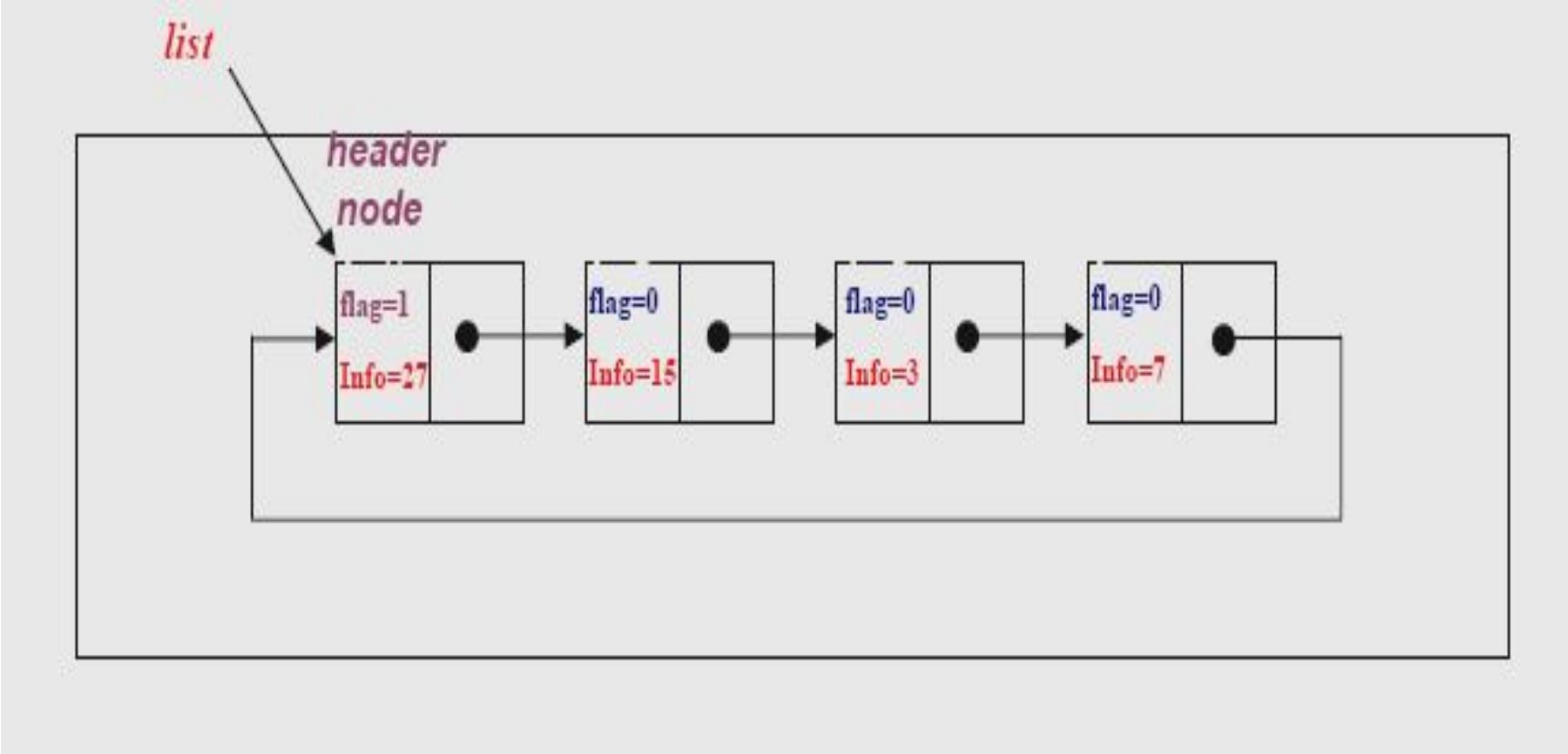


CIRCULAR LIST with header node

- ***Header Node with Flag:*** In this case a extra variable called flag can be used to represent the header node. For example flag in the header node can be 1, where the flag is 0 for the other nodes.

```
struct node{
        int
        flag;
        int
        info;
struct node *next;
};
typedef struct node *NODEPTR;
```

CIRCULAR LIST with header node



Topics

- Stack
- Queue
- Applications of Stacks and Queues

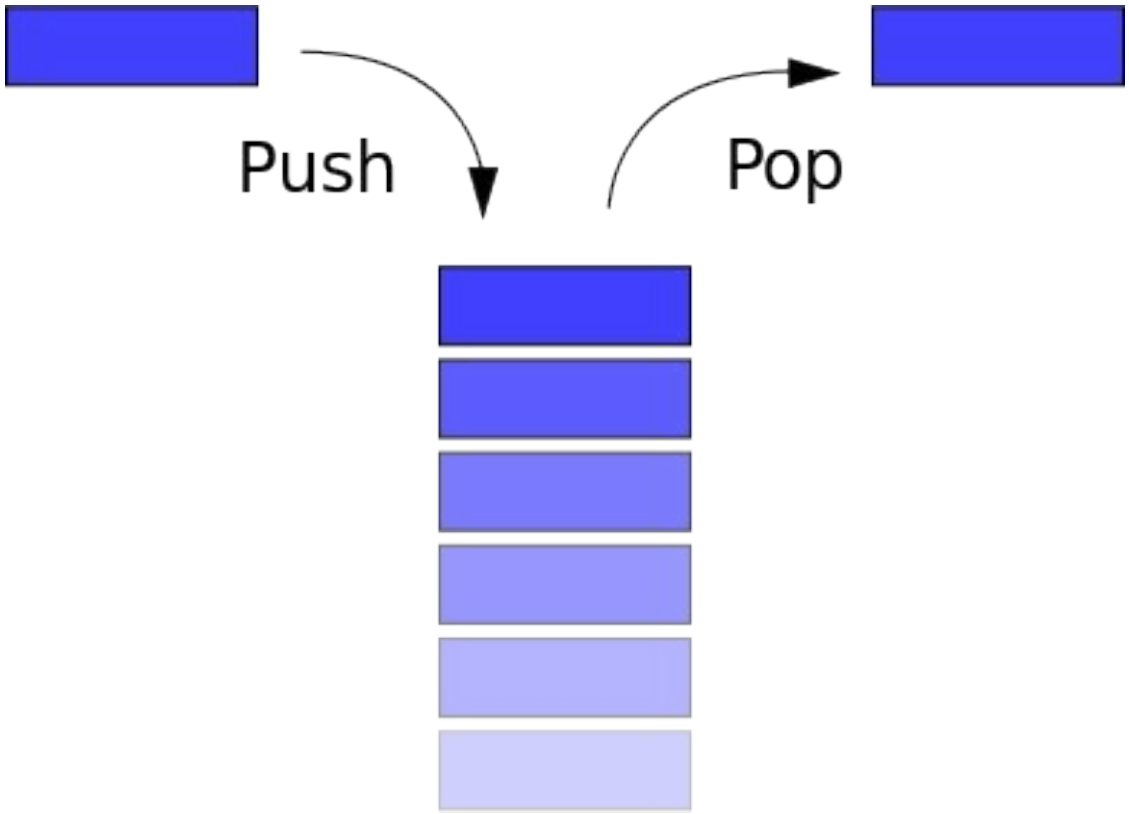
What is a Stack ?

- a stack is a particular *kind of abstract data type* or collection in which the fundamental (or only) *operations* on the collection are the addition of an entity to the collection, known as *push* and removal of an entity, known as *pop*.

Basic Principle

- The relation between the push and pop operations is such that the stack is a *Last-In-First-Out (LIFO)* data structure.
- In a LIFO data structure, the last element added to the structure must be the first one to be removed.

Stack



Operations on Stack

- This is equivalent to the requirement that, considered as a linear data structure, or more abstractly a sequential collection, the *push and pop* operations occur only at one end of the structure, referred to as the *top of the stack.*
- Often a *peek or top* operation is also implemented, *returning the value of the top element without removing it.*

Implementation

- A stack may be implemented to have a bounded capacity. If the *stack is full* and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an *overflow state*. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the *stack is empty*, it goes into *underflow state*, which means no items are present in stack to be removed.

Restricted Data Structure

- A stack is a *restricted data structure*, because only a *small number of operations are performed* on it. The nature of the pop and push operations also means that stack elements have a natural order. *Elements are removed from the stack in the reverse order* to the order of their addition. Therefore, the *lower elements* are those that have been on the *stack the longest*.

Exampl
es

1


$10000
$10000
DJ 36397802 A
100
UNITED DOLLARS
DJ 36397805 A
100
UNITED DOLLARS


Butterflies and Moths
Mammals
of North America
Birds
MARINE

In Simple Words

1

- A stack
  - *Last-in, first-out (LIFO)* property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria
    - A stack of Books
    - A stack of Dosa
    - A stack of Money

ADT Stack

1

- ADT stack operations
  - *Create* an empty stack
  - *Destroy* a stack
  - Determine whether a stack is *empty*
  - *Add* a new item
  - *Remove* the item that was added most recently
  - *Retrieve* the item that was added most recently
- A program can use a stack independently of the stack's implementation

Stack Operations

```
bool isEmpty() const;

    // Determines whether a stack is empty.

    // Precondition: None.

    // Postcondition: Returns true if the
    // stack is empty; otherwise returns
    // false.
```

Stack Operations

1

```
bool push(StackItemType newItem);

   // Adds an item to the top of a stack.

   // Precondition: newItem is the item to
   // be added.

   // Postcondition: If the insertion is
   // successful, newItem is on the top of
   // the stack.
```

Stack Operations

1

```
bool pop(StackItemType& stackTop);

   // Retrieves and removes the top of a stack

   // Precondition: None.

   // Postcondition: If the stack is not empty,
   // stackTop contains the item that was added
   // most recently and the item is removed.
   // However, if the stack is empty, deletion
   // is impossible and stackTop is unchanged.
```

Stack Operations

1

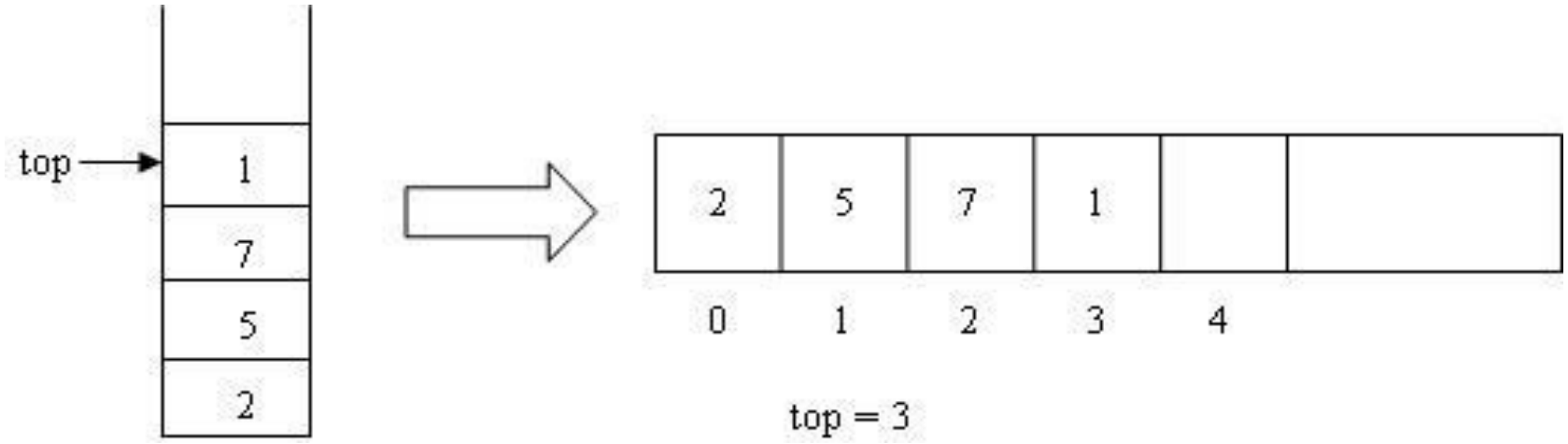
```
bool getTop(StackItemType& stackTop) const;

    // Retrieves the top of a stack.

    // Precondition: None.

    // Postcondition: If the stack is not empty,
    // stackTop contains the item that was added
       // most recently. However, if the stack is
       // empty, the operation fails and stackTop
       // is unchanged. The stack is unchanged.
```

Implementation Using Arrays



The Stack Class - Public Contents

1

```
class Stack
{
    public:

        Stack() { size = 10; current = -1;}
        int pop(){ return A[current--];}
        void push(int x){A[++current] = x;}
        int top(){ return A[current];}
        int isEmpty(){return ( current == -1 );}
        int isFull(){ return ( current == size-1);}
```

The Stack Class - Private Contents

1

```
  private:

        int   object;     // The data element
        int   current;    // Index of the array
        int   size;       // max size of the array
        int   A[10];      // Array of 10 elements
};
```

Stack - a Very Simple Application

2

- The *simplest application* of a stack is to *reverse a word.*
- You push a given word to stack - letter by letter - and then pop letters from the stack.

Applications of Stack

2

- Recursion
- Decimal to Binary Conversion
- Infix to Postfix Conversion and Evaluation of Postfix Expressions
- Rearranging Railroad Cars
- Quick Sort
- Function Calls
- Undo button in various software.

outputInBinary - Algorithm


```
function outputInBinary(Integer n)
    Stack s = new Stack
    while n > 0 do
        Integer bit = n modulo 2
        s.push(bit)
        if s is full then
            return error
        end if
        n = floor(n / 2)
    end while
    while s is not empty do
        output(s.pop())
    end while
end function
```

Algebraic Expressions

- Infix expressions
  - An operator appears between its operands
    - Example: a + b
- Prefix expressions
  - An operator appears before its operands
    - Example: + a b
- Postfix expressions
  - An operator appears after its operands
    - Example: a b +

A complicated example

2

- Infix expressions:

  `a + b * c + ( d * e + f ) * g`

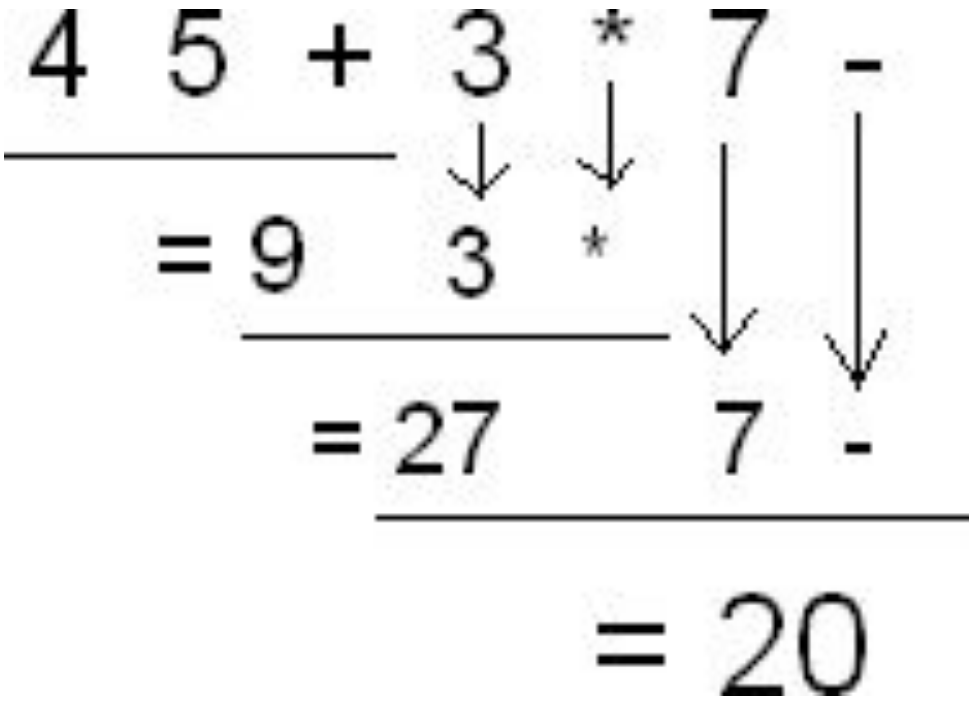
- Postfix expressions:

  `a b c * + d e * f + g * +`

- Prefix expressions:

  `+ + a * b c * + * d e f g`

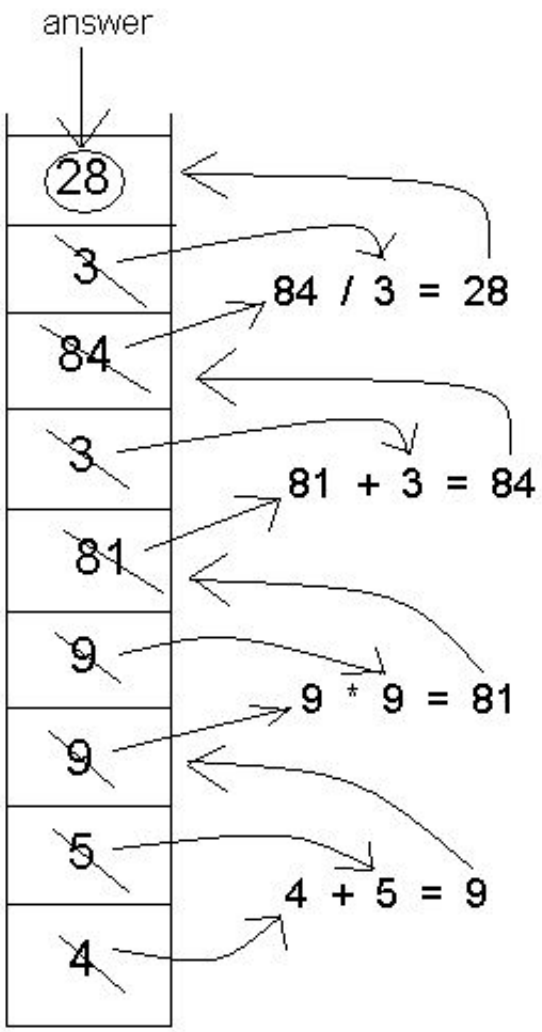
Evaluation of Postfix Expression



Evaluation of Postfix Expression

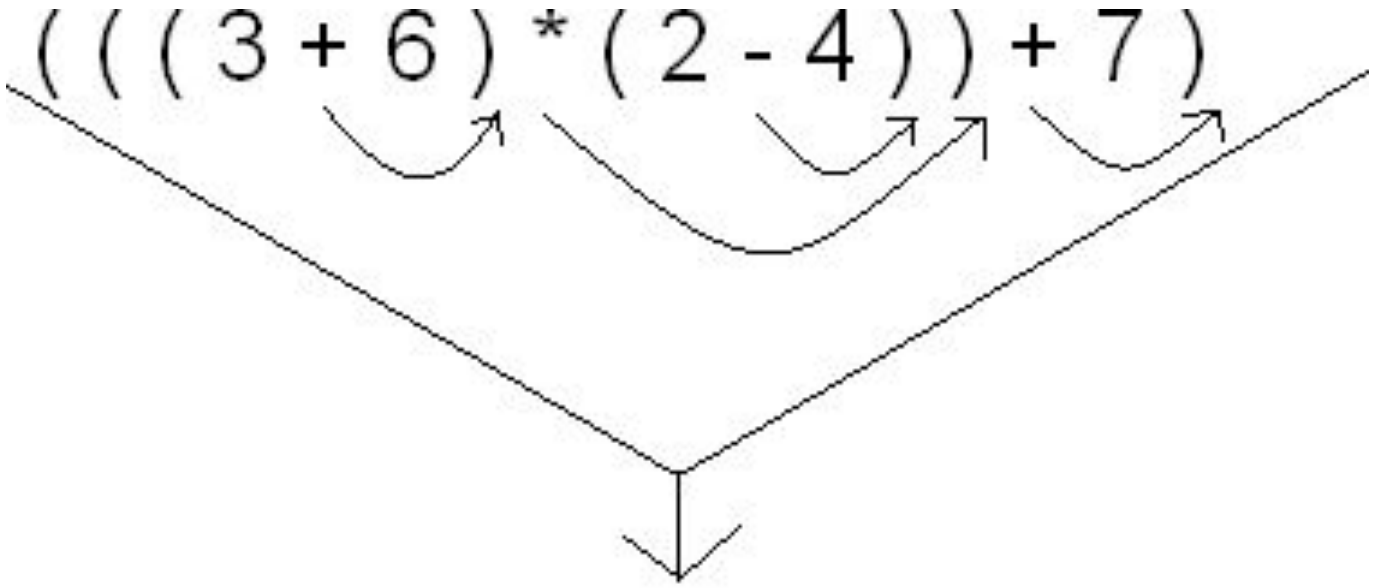
- The calculation: 1 + 2 * 4 + 3 can be written down like this in postfix notation with the advantage of no precedence rules and parentheses needed
- 1 2 4 * + 3 +
- The expression is evaluated from the left to right using a stack:
  - when encountering an operand: push it
  - when encountering an operator: pop two operands, evaluate the result and push it.

4 5 + 9 * 3 + 3 /



1) Push 4 onto the Stack (1 item in Stack)
2) Push 5 onto the Stack (2 items in Stack)
3) See the +, so pop 2 operands off (4 and 5) (0 items in Stack)
4) Push their result (9) onto the Stack (1 item in Stack)
5) Push 9 onto the Stack (2 items in Stack)
6) See the *, so pop 2 operands off (9 and 9) (0 items in Stack)
7) Push their result (81) onto the Stack (1 item in Stack)
8) Push 3 onto the Stack (2 items in Stack)
9) See the +, so pop 2 operands off (81 and 3) (0 items in Stack)
10) Push their result (84) onto the Stack (1 item in Stack)
11) Push 3 onto the Stack (2 items in Stack)
12) See the /, so pop 2 operands off (84 and 3) (0 items in Stack)
13) Push their result (28) onto the Stack (1 item in Stack)
14) See that there is nothing left in the expression, so pop the final element (28) off of the Stack, and you know that must be your answer. (0 items in Stack)

Infix to Postfix Conversion

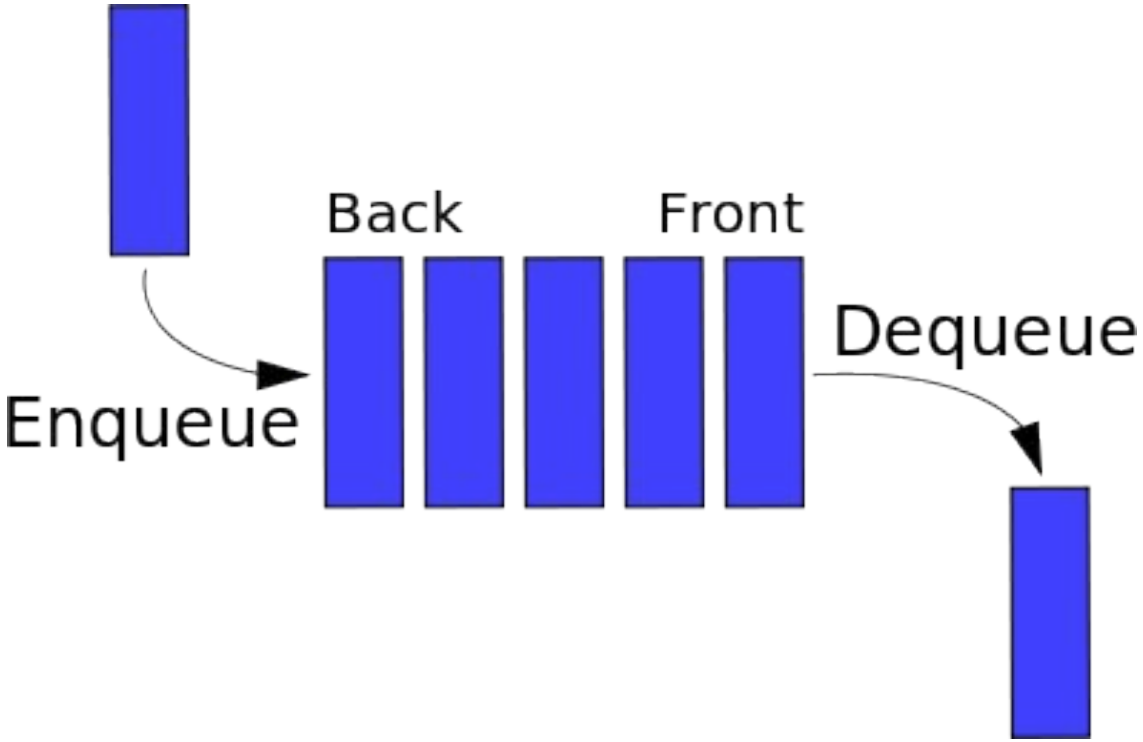


3 6 + 2 4 - * 7 +

What's this ?

2

Queue



Queue

3

- *A stack is LIFO (Last-In First Out)* structure. In contrast, a *queue is a FIFO (First-In First-Out)* structure.
- In a FIFO data structure, the first element added to the queue will be the first one to be removed.
- A queue is a linear structure for which items can be only *inserted at one end* and *removed at another end*.

Operations on Queue

- We will dedicate once again six operations on the Queue.
  - You can *add a person* to the queue (*enque*),
  - Look *who is the first person* to be serviced *(front)*
  - *Rremove the first person* (*dequeue*) and
  - Check whether the `(*isEmpty*).
  - Also there are two more operations to create and to destroy the queue.

Operations

- Queue create()
  - Creates an empty queue
- boolean isEmpty(Queue q)
  - Tells whether the queue q is empty
- enqueue(Queue q, Item e)
  - Place e at the rear of the queue q
- destroy(Queue q)
  - destroys queue q

Operations Continued

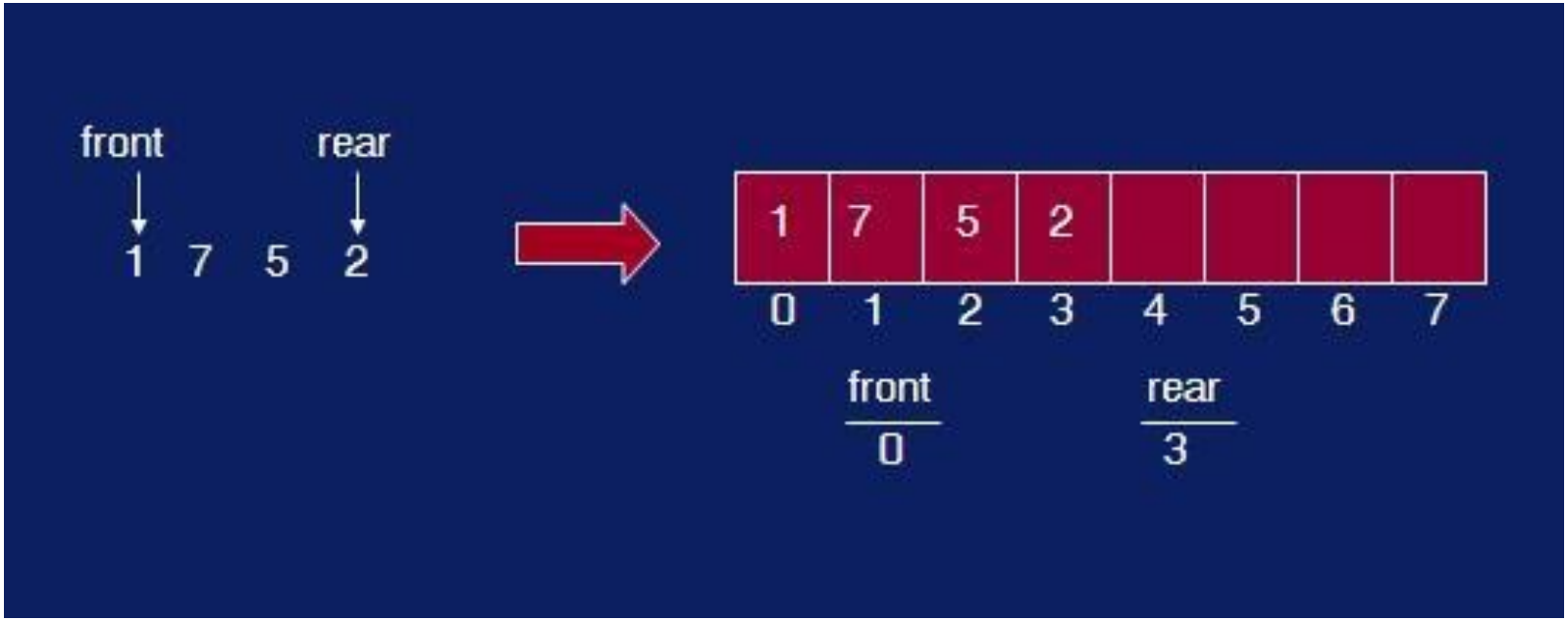
- Item front(Queue q)
  - returns the front element in Queue q without removing it
    Precondition: q is not empty
- dequeue(Queue q)
  - removes front element from the queue q
    Precondition: q is not empty

Implementation Using Arrays

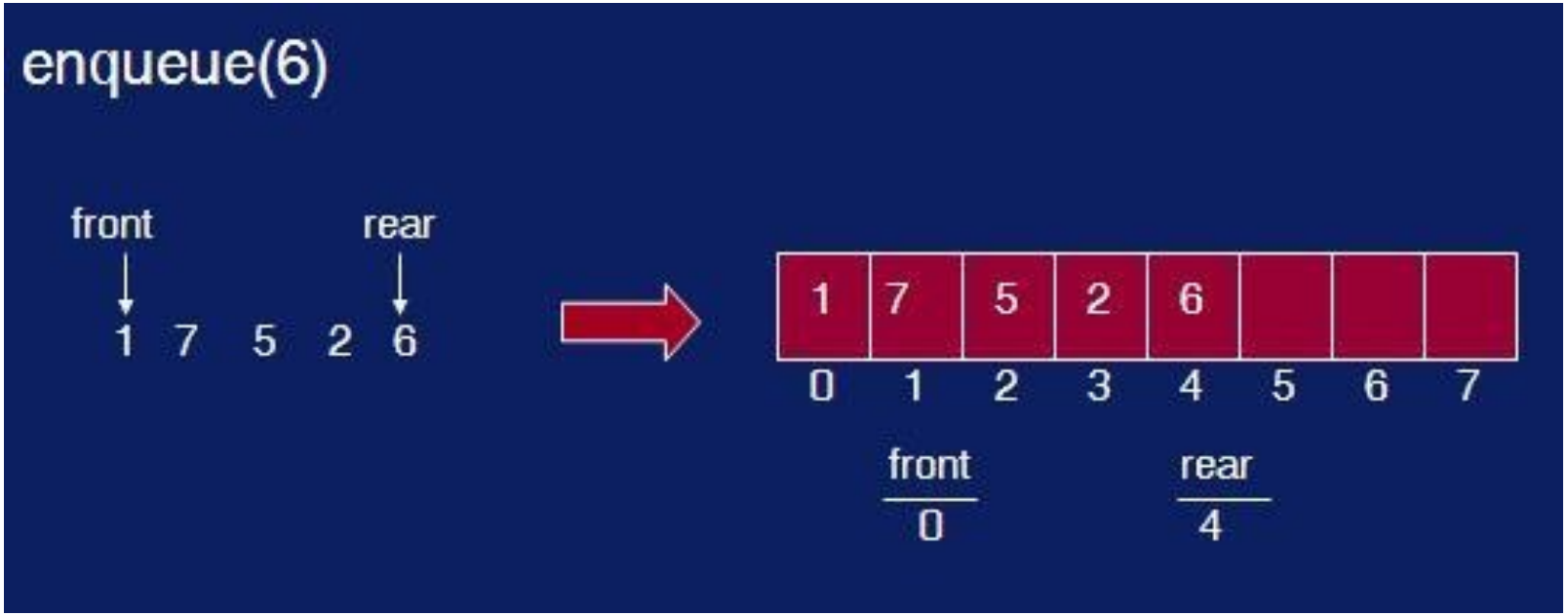
3

- If we use an array to hold queue elements, dequeue operation at the front (start) of the array is expensive.
- This is because we may have to shift up to “n” elements.
- For the *stack, we needed only one end; for queue we need both.*
- To get around this, we will not shift upon removal of an element.

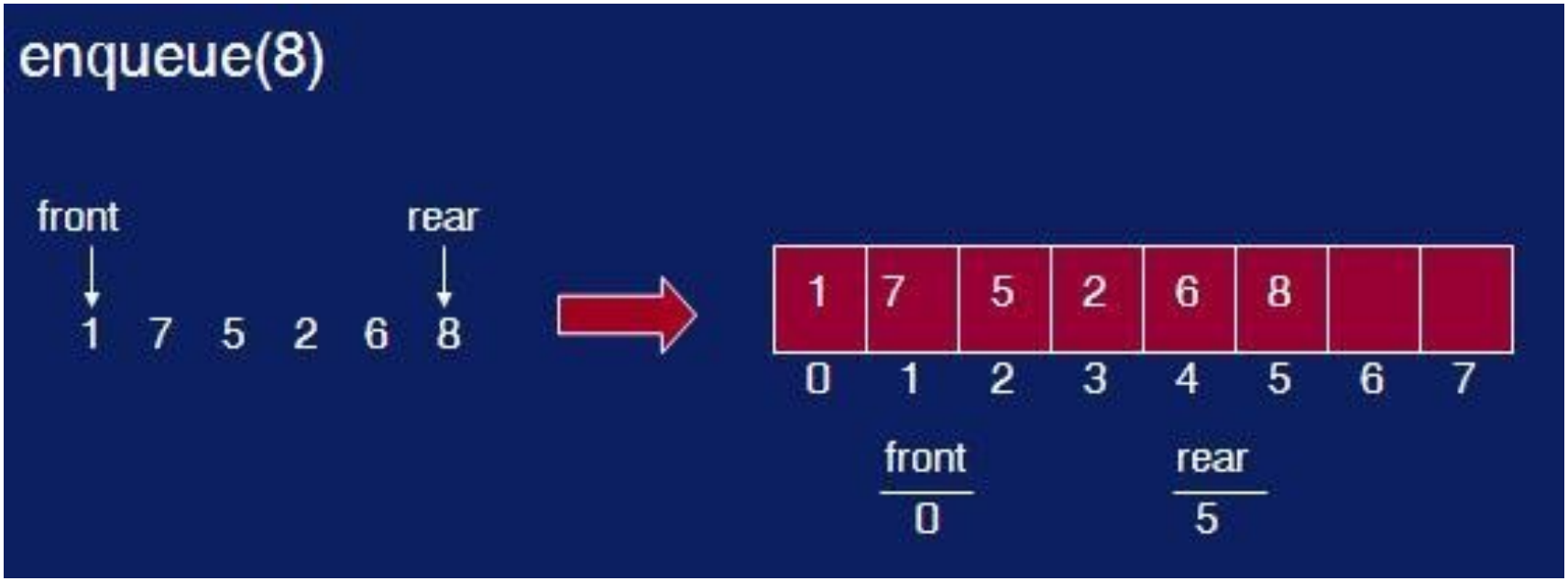
Snapshot of a Queue



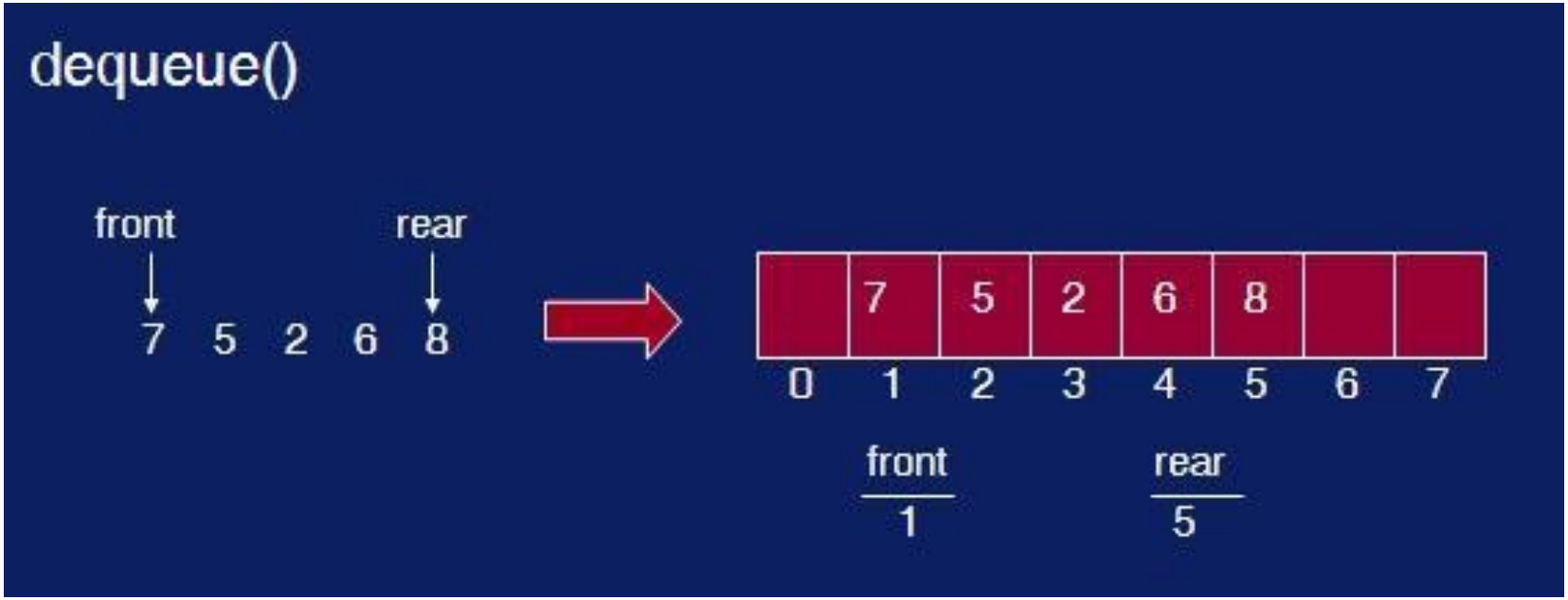
enqueue ()



enqueue() again

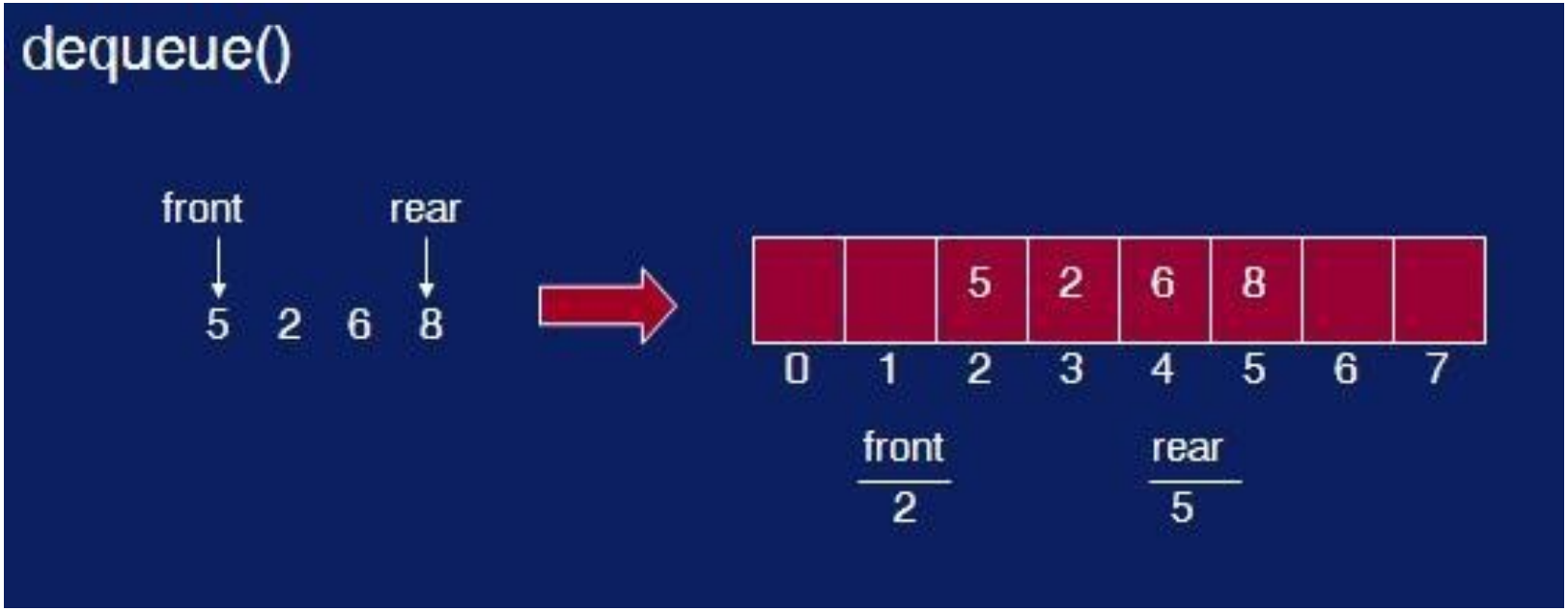


dequeue ()

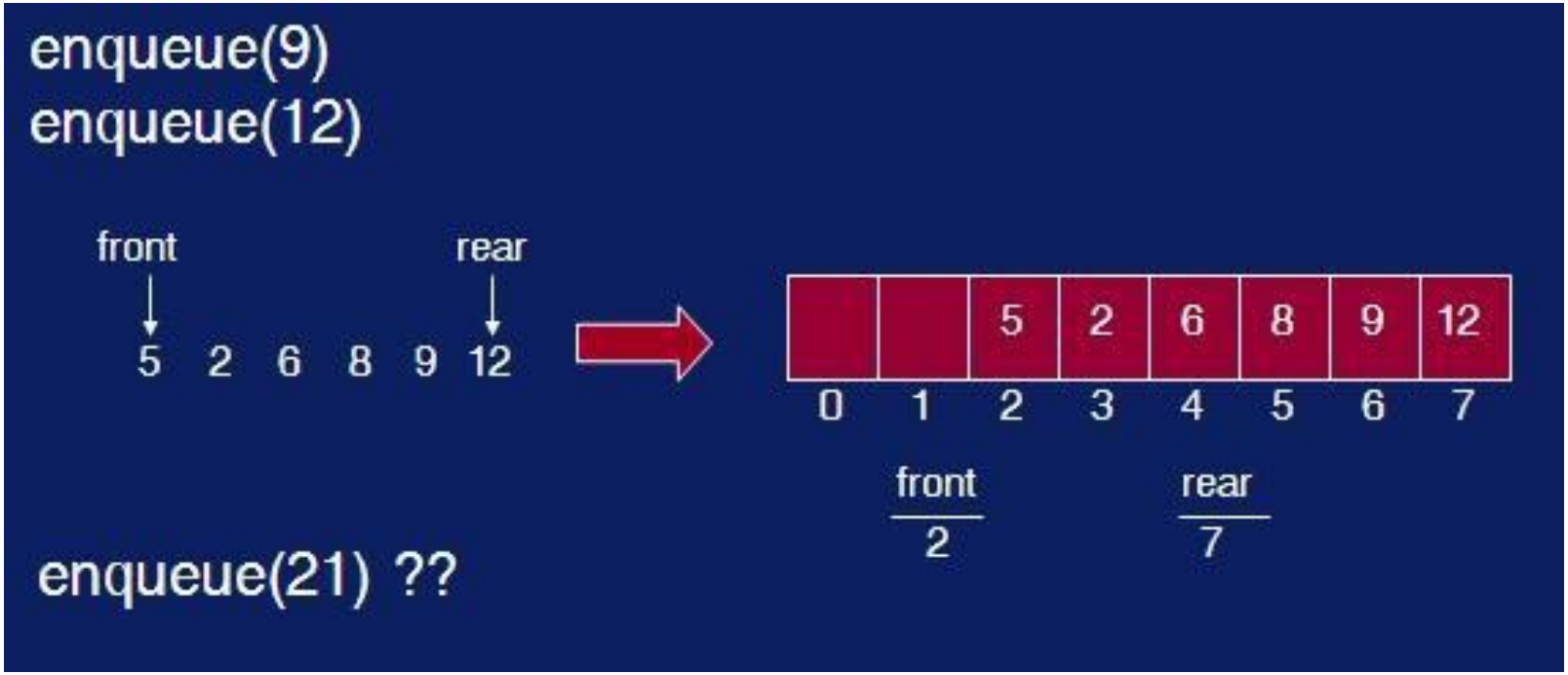


dequeue() again

4



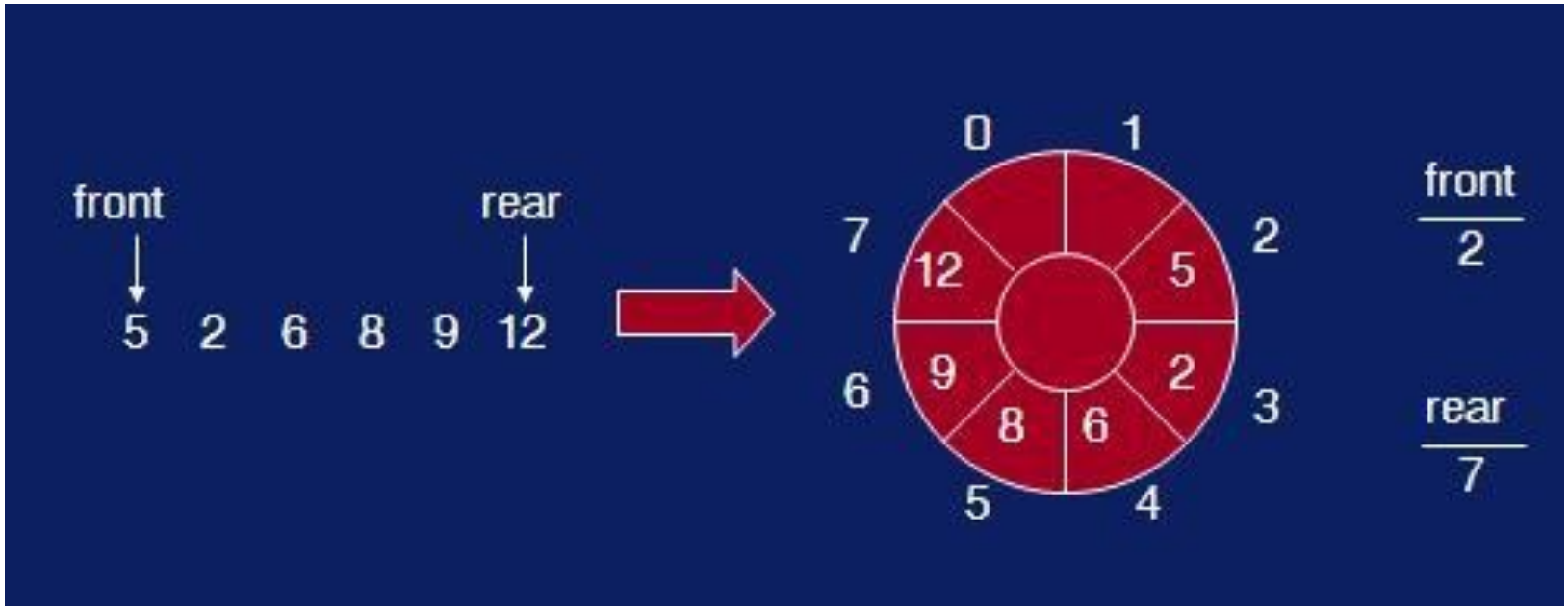
Two more enqueue()



What's Wrong ?

- We have inserts and removal running in constant time but we created a new problem.
- We cannot insert new elements even though there are two places available at the start of the array.
- Solution: allow the queue to "wrap around". Basic idea is to picture the array as a circular array as show below.

Queue array wrap-around



enqueue(element)

4

```
void enqueue(int x)
{
    rear = (rear+1)%size;
    array[rear] = x;
      noElements = noElements+1;
}
```

dequeue ()


```
int dequeue()
{
    int x = array[front];
      front = (front+1)%size;
      noElements = noElements-1;
      return x;
}
```

isEmpty() and isFull()

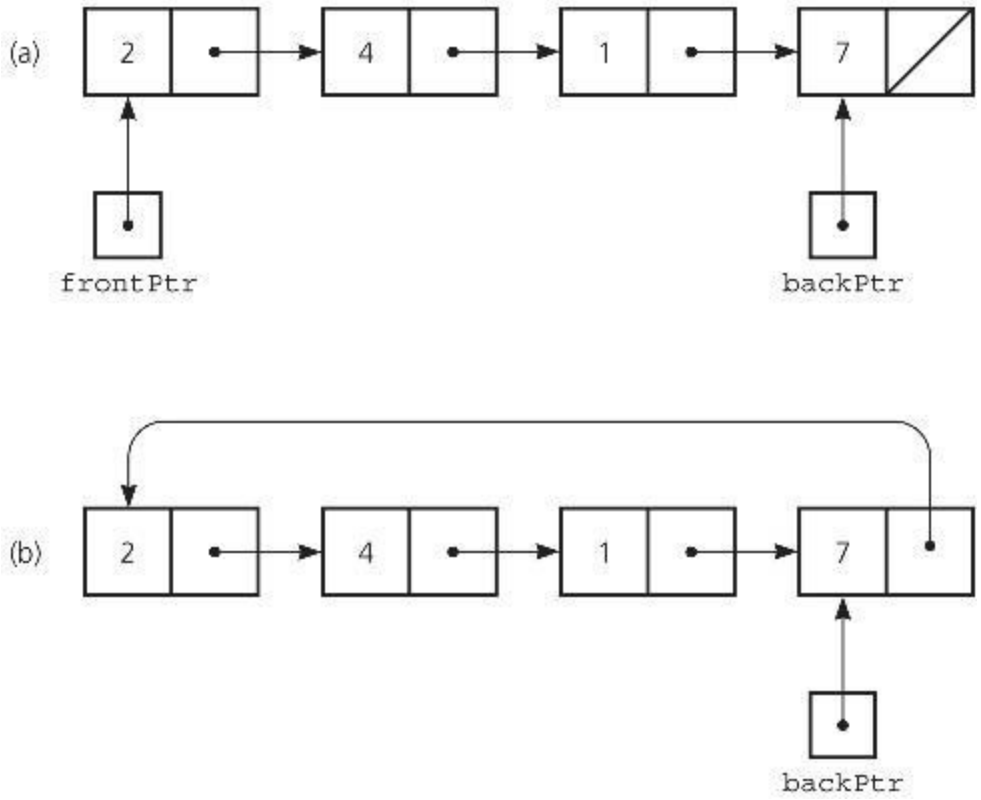

```
int isFull()
{
    return noElements == size;
}

int isEmpty()
{
      return noElements == 0;
}
```

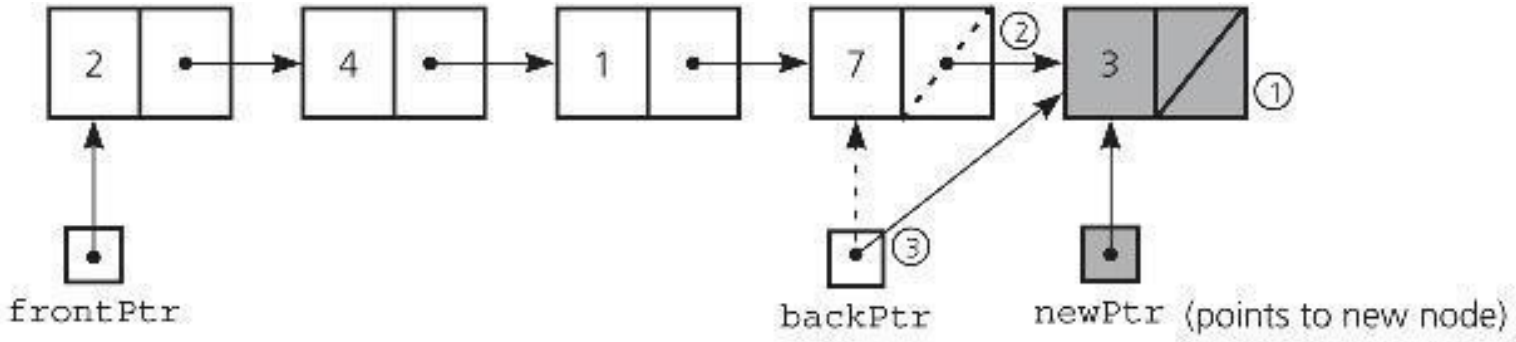
Pointer Based Implementation

- Possible implementations of a pointer-based queue
  - A linear linked list with two external references
    - A reference to the front
    - A reference to the back
  - A circular linked list with one external reference
    - A reference to the back

Queue - Linear and Circular List



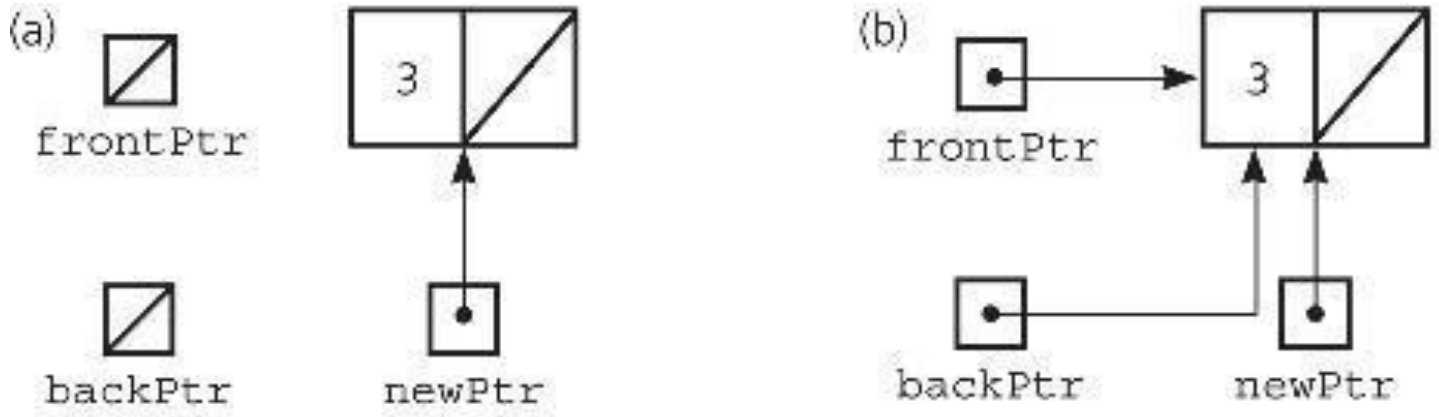
Non-empty Queue - Insertion




```
1. newPtr->next = NULL;
2. backPtr->next = newPtr;
3. backPtr = newPtr;
```

Empty Queue - Insertion

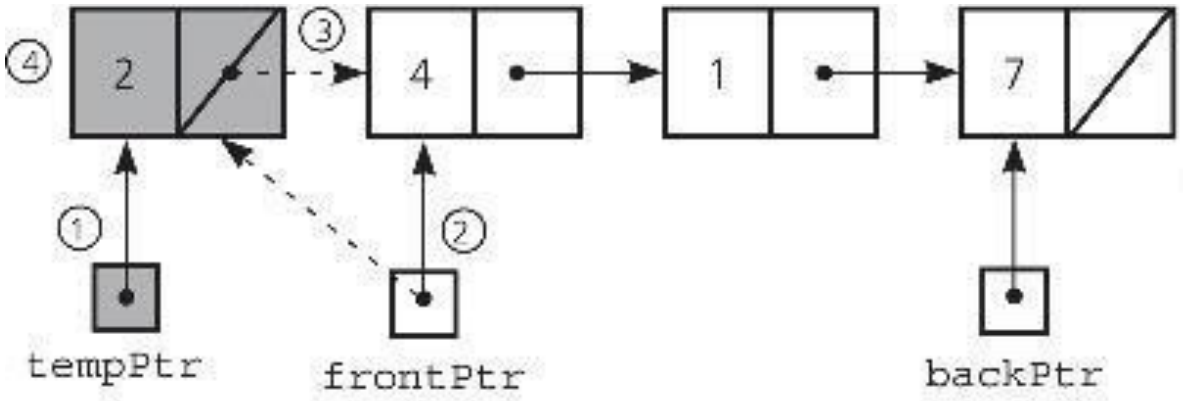
5




```
frontPtr = newPtr;
backPtr = newPtr;
```

Queue with more than 1 element - Deletion

5




```
1. tempPtr = frontPtr;
2. frontPtr = frontPtr->next;
3. tempPtr->next = NULL;
4. delete tempPtr;
```

Applications of Queues

5

- Queue is used when things don't have to be processed immediatly, but have to be processed in First In First Out order like Breadth First Search.
- This property of Queue makes it also useful in following kind of scenarios.

Applications of Queues

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Applications of Queues

5

- Computer systems must often provide a “holding area” for messages between two processes, two programs, or even two systems. This holding area is usually called a “buffer” and is often implemented as a queue.
- Call center phone systems will use a queue to hold people in line until a service representative is free.

Applications of Queues

5

- Buffers on MP3 players and portable CD players, iPod playlist. Playlist for jukebox - add songs to the end, play from the front of the list.
- Round Robin (RR) algorithm is an important scheduling algorithm. It is used especially for the time-sharing system. The circular queue is used to implement such algorithms.