Data Structure

Unit 4: Linked List

Prepared by: Shivangi Malli
A.V.P.T.I- Rajkot

Outline

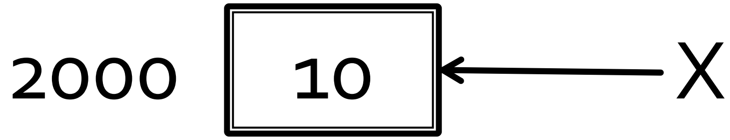
- Pointers
- Structure
- Pointers with structure
- Dynamic memory allocation

Pointers Revisited

- **Pointer**: A pointer is a variable which contains an address of another variable in memory.
- It is declared by * indicator and it is derived data type.
- We can create a pointer variable in C using following syntax:

Data type   *pointer name;

- For example: int *ptr;
- Here, ptr is a pointer to integer data type.
- Suppose one variable called X having value 10 is stored at address(memory location) 2000.



Pointers Revisited

- A pointer variable ptr pointing to the address 2000 :

  ptr = &X;

- & is **"address of"** or **"referencing**" which returns memory location of a variable
- Accessing the value of variable X using pointer variable:

  Y = *ptr;   give 10 to variable Y.

- * is **"indirection"** or **"dereferencing"** operator which returns value stored at that memory location
- Thus the use of a pointer(link) to refer to the element of data structure implies that:
- Elements which are logically adjacent need not be physically adjacent in Memory is known as linked allocation.

Pointers Revisited

```
Void main()
{
    int a=10, *p;
    p=&a;
    printf("%d, %d, %d", a, p, *p);
}
```

Variable	Value	Address
A	10	5000
p	5000	5048

What will be the output?

(consider memory location is 5000)

10, 5000, 10

Structure Revisited

- Structure is a collection of logically related data items of different data types grouped together under a single name
- structure is user defined data type available in C
- Structure helps to organize complex data in a more meaningful way

Defining a Structure

```
struct [structure tag] {
        member definition;
        member definition;
        .....
} [one or more structure variables];
```

```
struct Books {
        char title[50];
        char author[50];
        int book_id;
} book1,book2;
```

- Member are variables of different data types like int, float, char etc

Structure Revisited

- To access any member of a structure **member access operator (.)** is used

StructVariable.StructMember

Initializing structure variables

For book example

book1.book_id=12345

strcpy(book1.title, “DS”)

strcpy(book1.author, “Paul”)

OR

Struct Books book1={“DS”, “Paul”, 12345}

```
struct Books {
        char title[50];
        char author[50];
        int book_id;
} book1,book2;
```

Structure with pointers

- In case of pointer to structure, members are accessed using **arrow (->) operator.**

```
struct Point {
    int x, y;
};
int main() {
    struct Point p1 = {1, 2};
    // p2 is a pointer to structure p1
    struct Point *p2 = &p1;
    // Accessing structure members using structure pointer
    printf("%d %d", p2->x, p2->y);
    return 0;
}
```

Output ?

1 2

Dynamic Memory Allocation

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

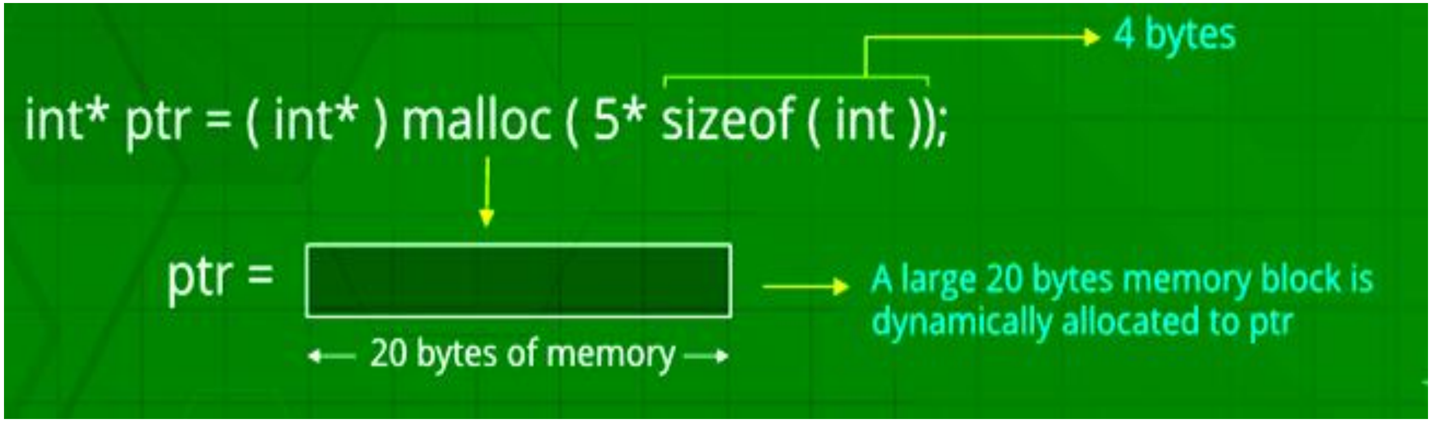
malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

Dynamic Memory Allocation

malloc() function in C

- The malloc() function allocates single block of requested memory.
- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.
- The syntax of malloc() function is given below:

ptr=(cast-type*)malloc(byte-size)

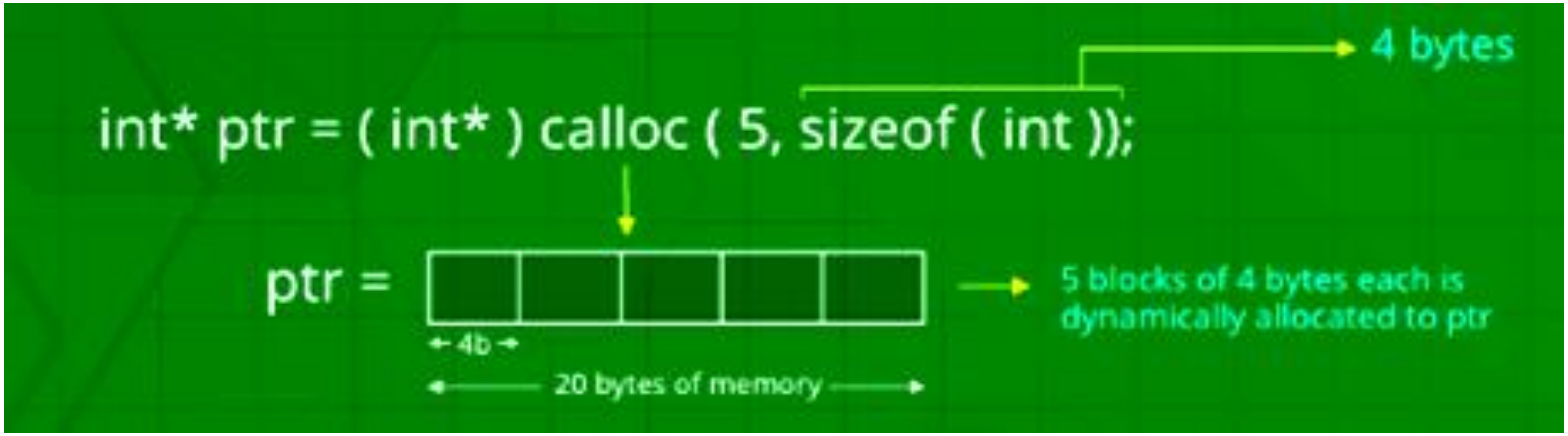


Dynamic Memory Allocation

<u>calloc() function in C</u>

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.
- The syntax of calloc() function is given below:

ptr=(cast-type*)calloc(number, byte-size)

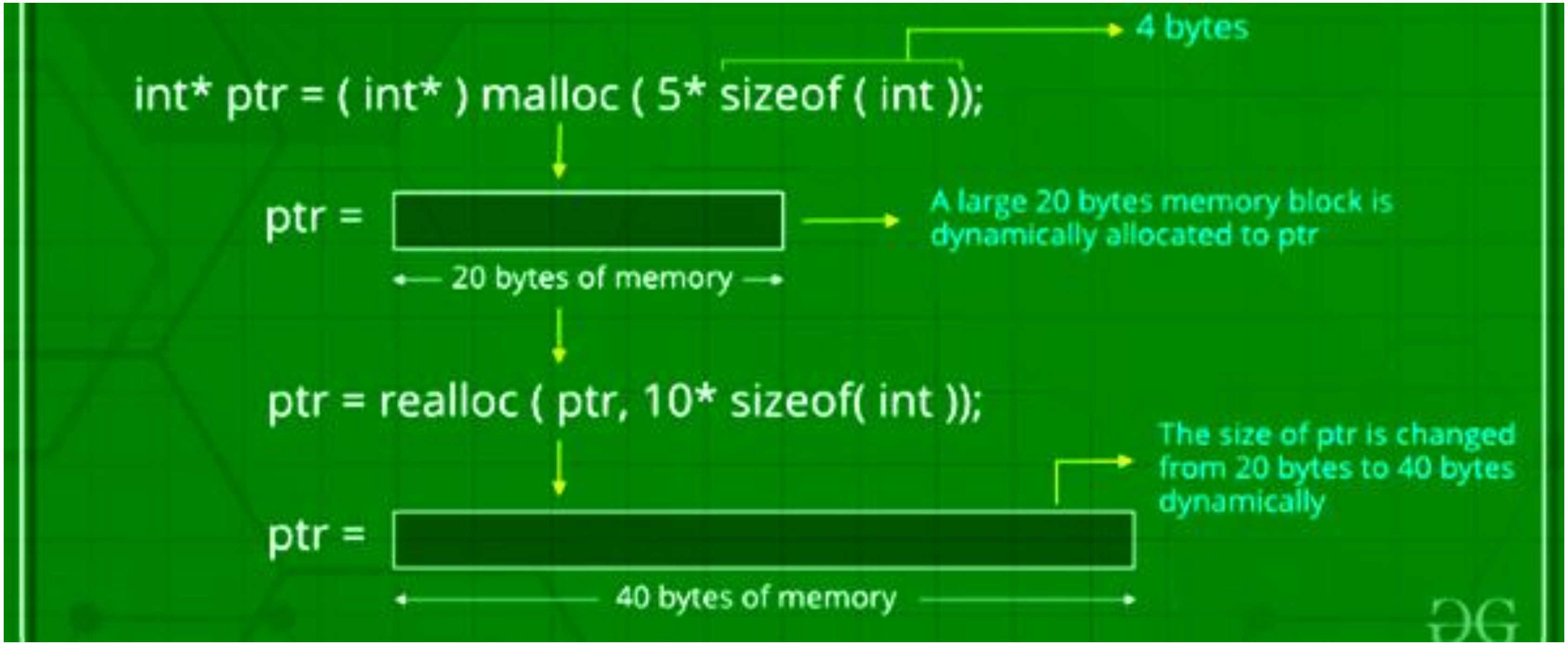


Dynamic Memory Allocation

realloc() function in C

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

ptr=realloc(ptr, new-size)

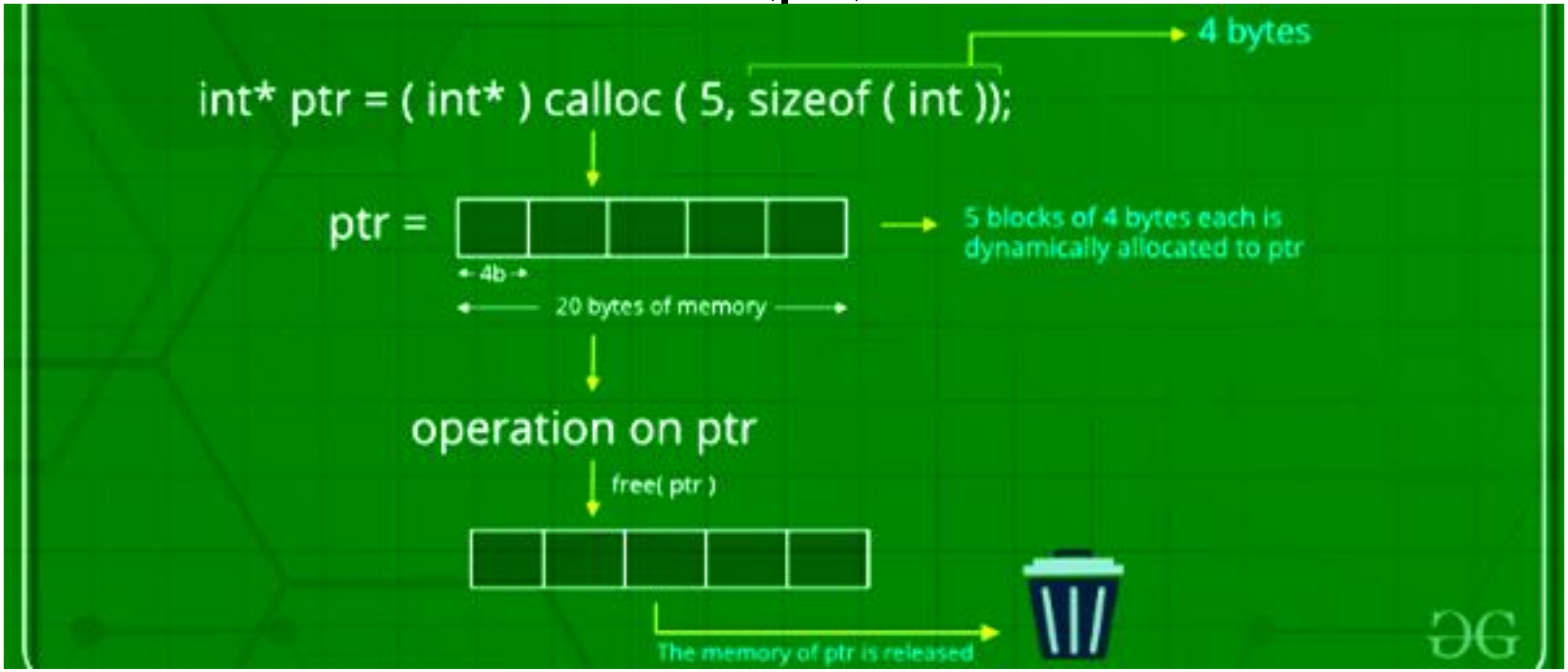


Dynamic Memory Allocation

free() function in C

- The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

free(ptr)



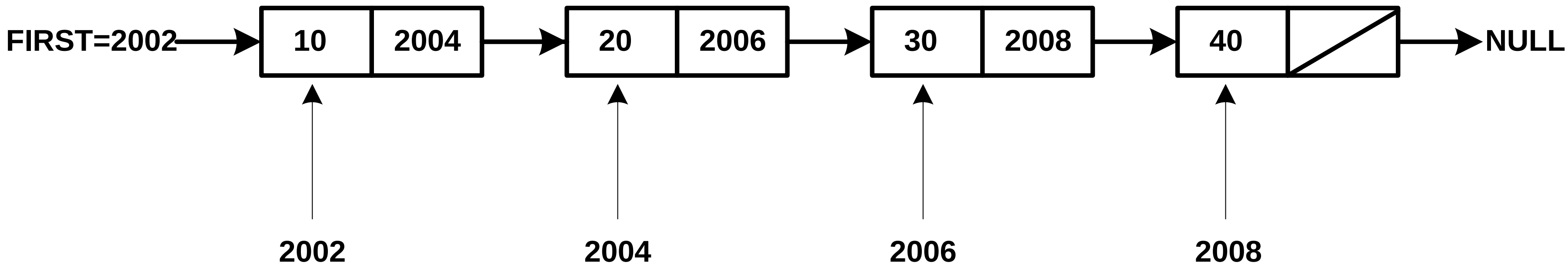
Linked List

- In sequential(linear) allocation, the elements are ordered linearly by allocating consecutive memory locations.
- Another way to maintain linear access is to store the address of the next element in each element of the list by using pointers.
- Here the linear order is maintained logically by using links and the elements need not be physically adjacent in memory. This type of allocation is called Linked Allocation.
- In simple words:
  - **Access to elements → Linear**
  - **Storage→ nonlinear**

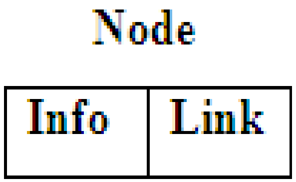
Sequential Allocation	v/s Linked Allocation
all elements are physically adjacent in memory. It is implemented using arrays.	All elements are only logically adjacent in memory. It is implemented using pointers
Once an array is defined, its size remains constant during runtime. Hence, memory might be wasted.	The LL size can change during runtime. Hence, use of memory is more efficient.
The address of any element can be calculated from base address, no need of storing address of all elements. No overhead.	Here, there is an overhead of storing addresses of all the elements.
If a particular element is required, it can be found faster by directly computing its address.	If a particular node in LL is required, we have to follow all the links until the desired node is found.
Operations such as Insert/Delete requires movement of a lot of elements & so it is inefficient.	Here, Insert/Delete operations can be performed by just changing the links and so are more efficient.
Sequential Allocation can be used to implement Linear DS such as Arrays, Stacks, Queues, etc.	Linked Allocation is more useful for complex Non-Linear DS such as trees, graphs, files, etc.

Singly linked list

- A list in which each node contains a link or pointer to next node in the list is known as singly linked list or one way chain.



- Two parts in each node
  - Information (INFO)→ Actual data
  - Address or pointer to next node



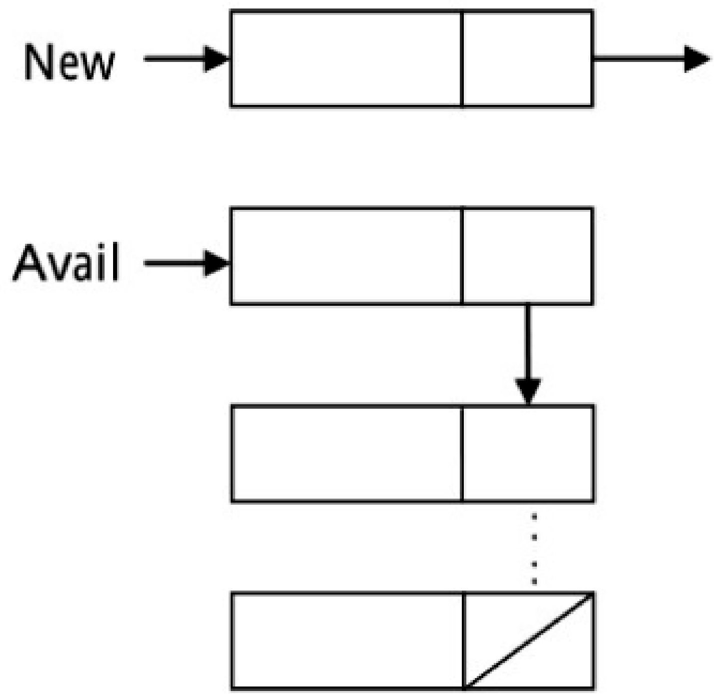
- Here, the variable FIRST contains an address of the first node
- The last node of the list stores NULL as an address. NULL indicates end of the list.

Operations on SLL

- Traversing a linked list.
- Insert new node at beginning of the list
- Insert new node at end of the list
- Insert new node into ordered list
- Insert new node at any position or in between the list.
- Delete first node of the list
- Delete last node of the list.
- Delete node from any specific position in the list.
- Searching element in list.
- Merging of two linked list.
- Sorting operation of list.
- Copy of the list.

Fundamental Things

- System has free pool which is called availability list.
- AVAIL which stores the address of the first free space of the free pool
- During insertion in a list, the memory address pointed by AVAIL pointer will be taken from the availability list and used to store the information.
- After the insertion, AVAIL points to next free node
- during deletion, space occupied by node will be returned to free pool for reuse by other programs



(After)

[To free a Node
Form avail stack]

Insertion at the Beginning of SLL

INSERTBEG (VAL,FIRST)

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write "Availability stack underflow"
   Return
2. [Obtain address of next free node]
   NEW←AVAIL
3. [Remove free node from availability stack]
   AVAIL←LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ←VAL
   LINK (NEW) ←FIRST
5. [Assign the address of the Temporary node to the First Node]
   FIRST←NEW
6. [Finished]
   Return (FIRST)

This function inserts a new element **VAL** at the beginning of the linked list.

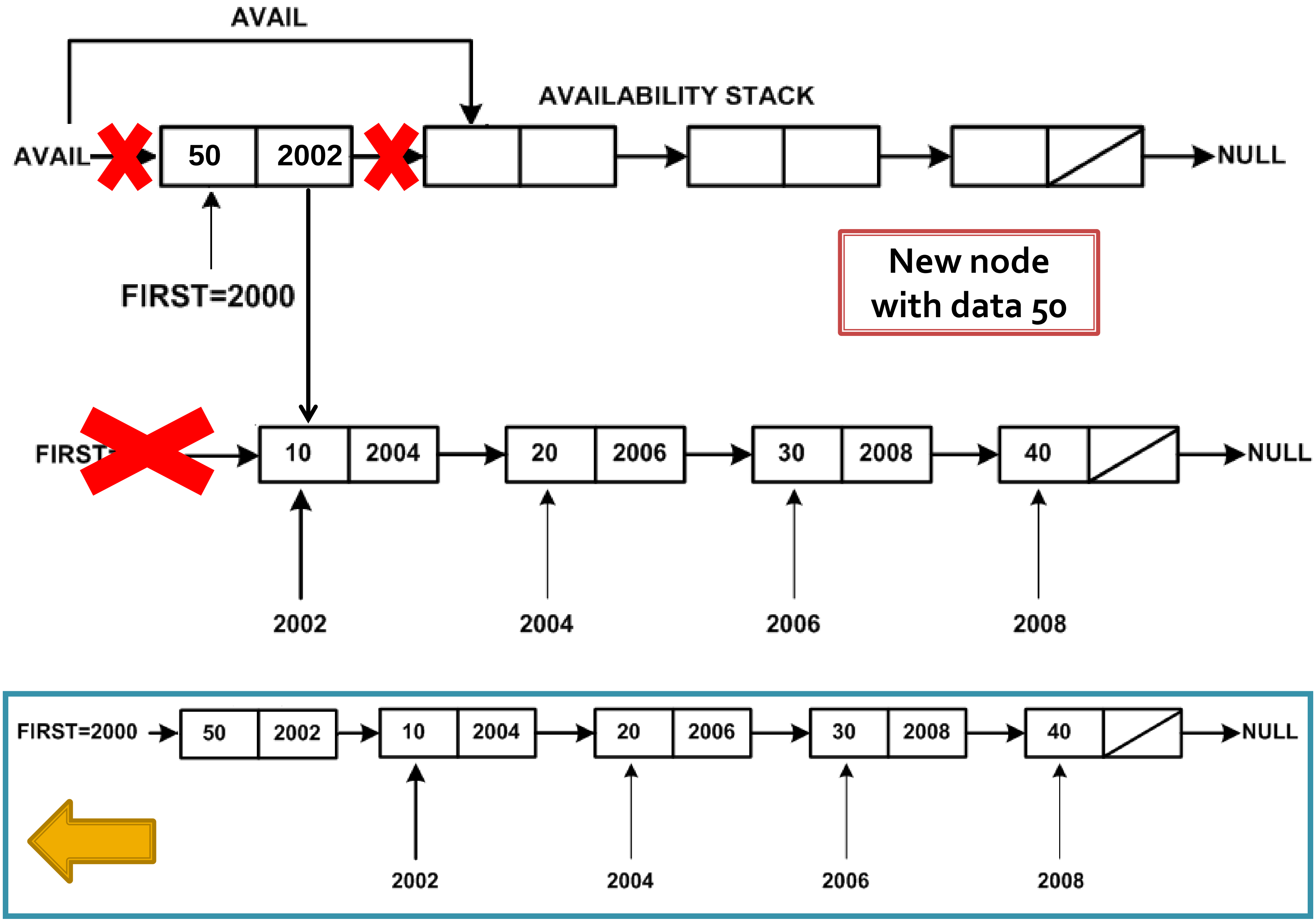
FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

AVAIL : Top node of availability list

INFO: stores Data of node

LINK: stores pointer to next node


AVAIL
AVAILABILITY STACK
AVAIL
50
2002
NULL
FIRST=2000
New node
with data 50
FIRST
10
2004
20
2006
30
2008
40
NULL
2002
2004
2006
2008
FIRST=2000
50
2002
10
2004
20
2006
30
2008
40
NULL
2002
2004
2006
2008

Insertion at the END of SLL

INSERTEND (VAL,FIRST)

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write “Availability stack underflow”
   Return
2. [Obtain address of next free node]
   NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ← VAL
   LINK (NEW) ← NULL
5. [is list empty?]
   If FIRST = NULL then
   FIRST ← NEW

This function inserts a new element **VAL** at the end of the linked list.

FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

AVAIL : Top node of availability list

INFO: stores Data of node

LINK: stores pointer to next node

SAVE : Temporary node pointer for traversal

Insertion at the END of SLL (CONT)

6. [initialize search for last node]
   SAVE←FIRST
7. [Search end of the list]
   Repeat while LINK (SAVE) ≠ NULL
   SAVE←LINK (SAVE)
8. [Set LINK field of last node to NEW]
   LINK (SAVE) ←NEW
9. [Finished]

This function inserts a new element **VAL** at the end of the linked list.

FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

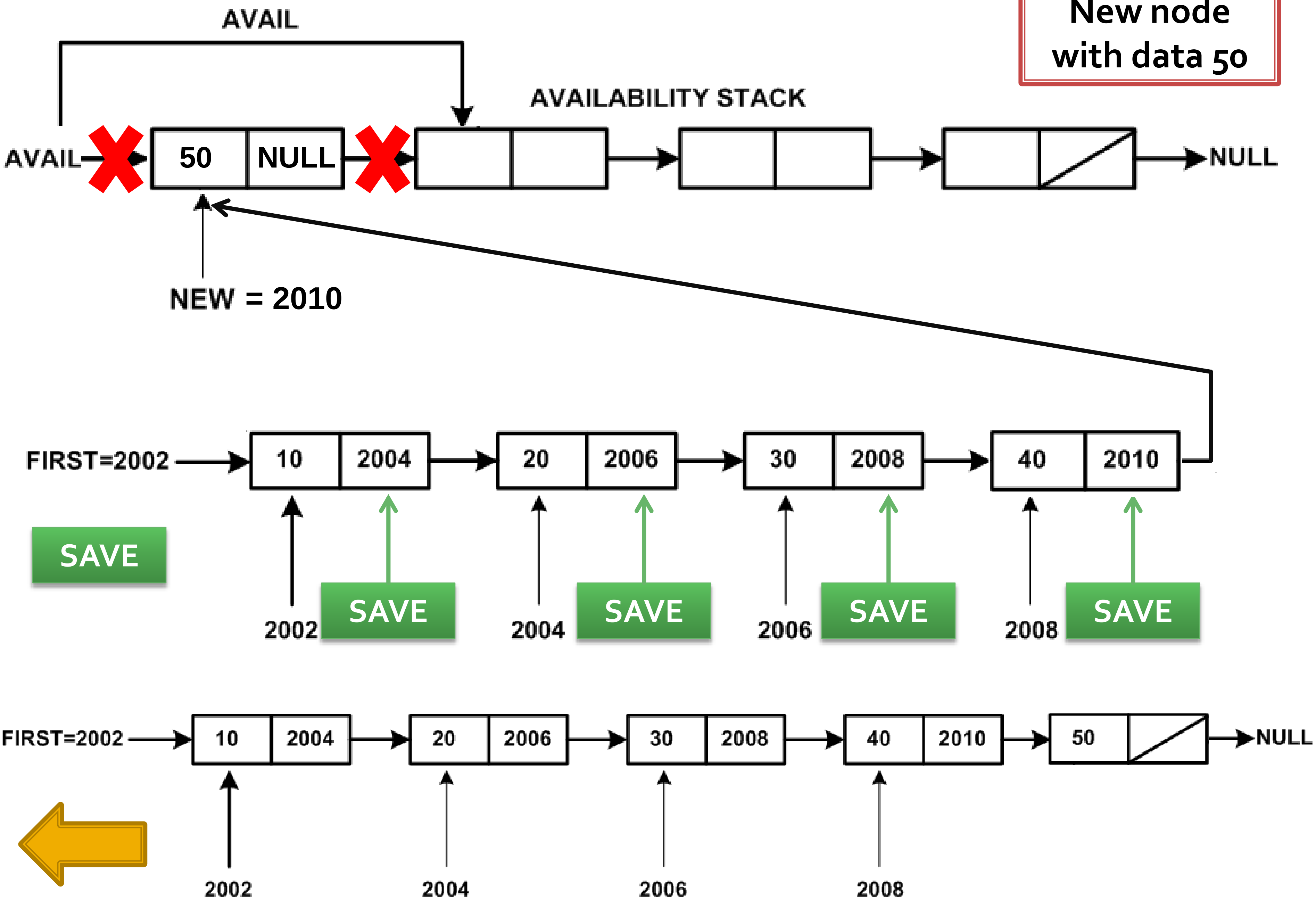
AVAIL : Top node of availability list

INFO: stores Data of node

LINK: stores pointer to next node

SAVE : Temporary node pointer for traversal

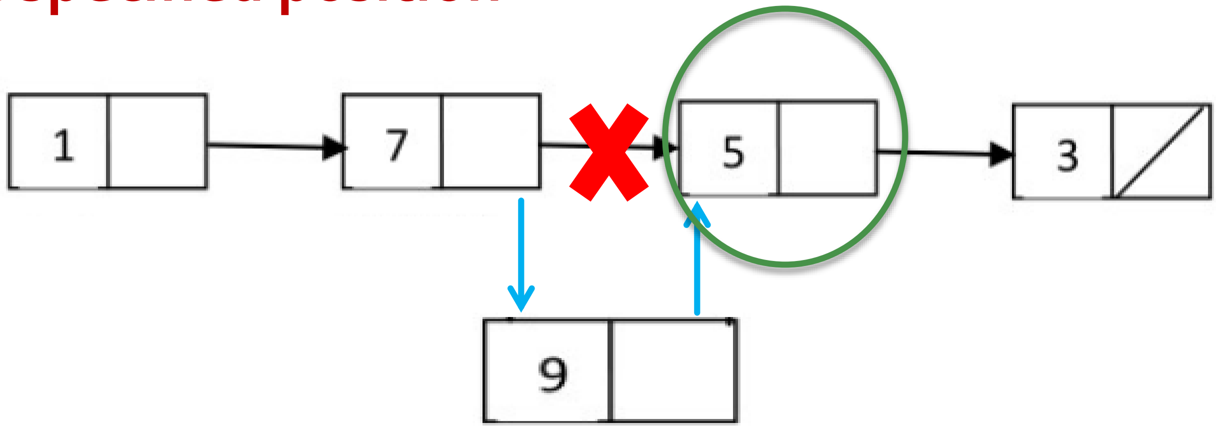
New node with data 50



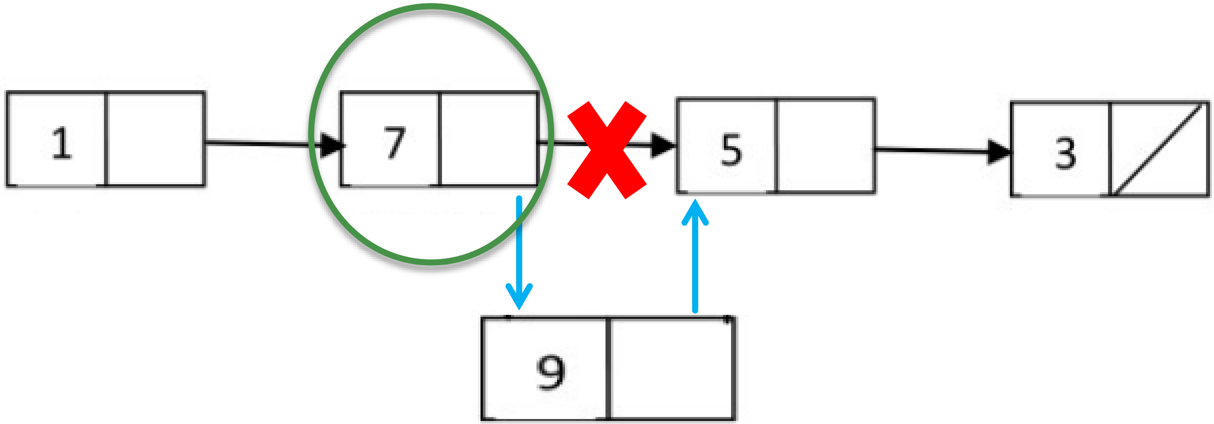
Insertion at the SPECIFIED POSITION in SLL

There are two possibilities:

- **Before specified position**



- **After specified position**



Insertion BEFORE SPECIFIED node in SLL

INSERTBEFORE (VAL,FIRST,N)

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write “Availability stack underflow”
   Return
2. [Obtain address of next free node]
   NEW←AVAIL
3. [Remove free node from availability stack]
   AVAIL←LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ←VAL
5. [set pointers PTR to FIRST]
   PTR← FIRST

This function inserts a new element **VAL** before specified node of the linked list.

FIRST: a pointer which contains address of first node in the list

N: Specified node value

NEW : temporary new node

AVAIL : Top node of availability list

INFO: stores Data of node

LINK: stores pointer to next node

PTR,PREPTR : Temporary node pointers for traversal

Insertion BEFORE SPECIFIED node in SLL

INSERTBEFORE (VAL,FIRST,N)

6. [Reach to specific location]
   Repeat while INFO (PTR) $\neq$ N
   PREPTR $\leftarrow$ PTR
   PTR $\leftarrow$ LINK(PTR)
7. [Insert new node before given location]
   LINK(PREPTR) $\leftarrow$ NEW
   LINK(NEW) $\leftarrow$ PTR
8. [Finished]

This function inserts a new element **VAL** before specified node of the linked list.

FIRST: a pointer which contains address of first node in the list

N: Specified node value

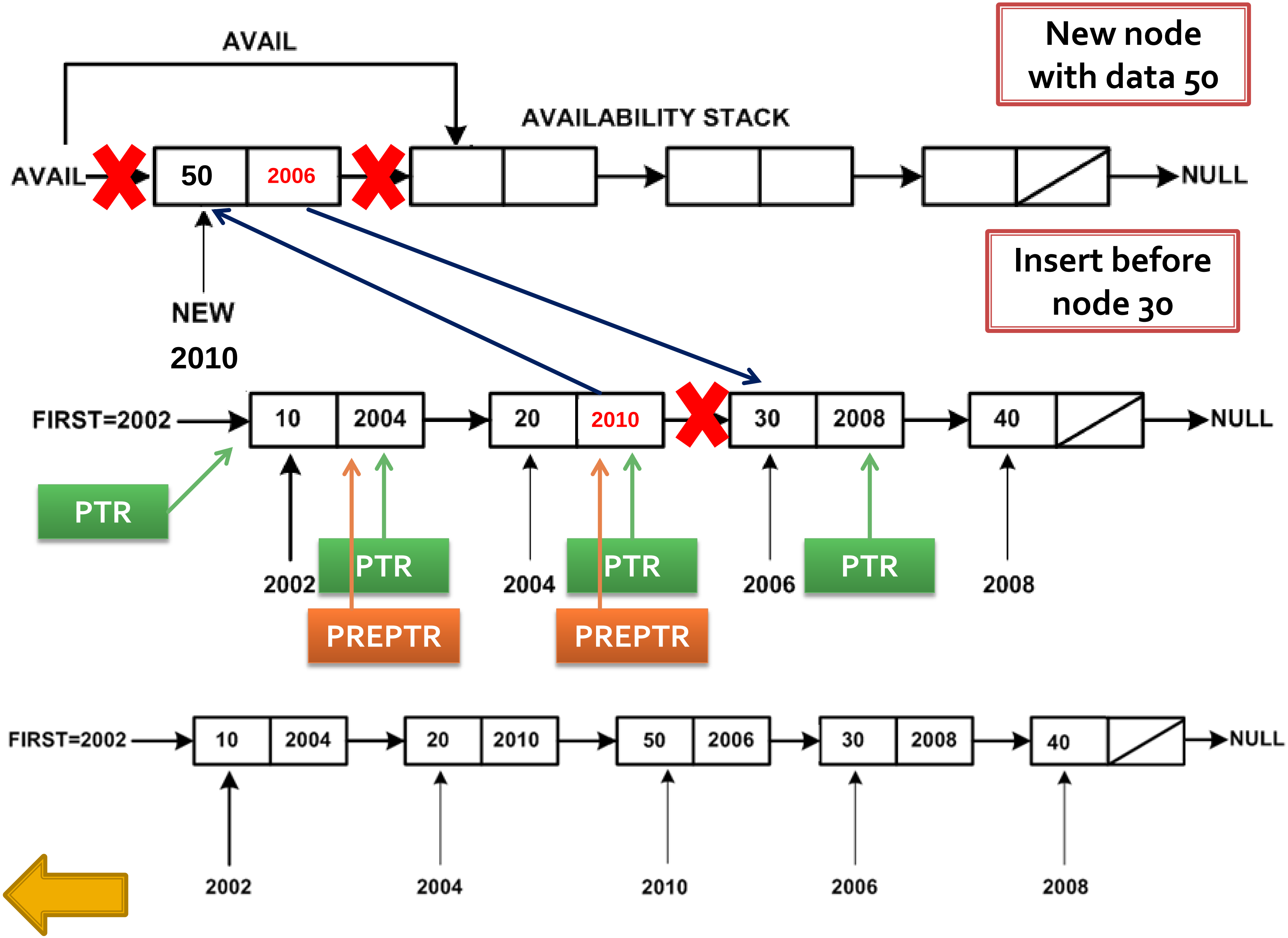
NEW : temporary new node

AVAIL : Top node of availability list

INFO: stores Data of node

LINK: stores pointer to next node

PTR,PREPTR : Temporary node pointers for traversal


New node with data 50
AVAIL
AVAILABILITY STACK
AVAIL
50
2006
NULL
NEW
2010
Insert before node 30
FIRST=2002
10
2004
20
2010
30
2008
40
NULL
PTR
PTR
PTR
PTR
PREPTR
PREPTR
2002
2004
2006
2008
FIRST=2002
10
2004
20
2010
50
2006
30
2008
40
NULL
2002
2004
2010
2006
2008

Insertion AFTER SPECIFIED node in SLL

INSERTAFTER (VAL,FIRST,N)

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write “Availability stack underflow”
   Return
2. [Obtain address of next free node]
   NEW←AVAIL
3. [Remove free node from availability stack]
   AVAIL←LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ←VAL
5. [set pointers PTR to FIRST]
   PTR← FIRST
   PREPTR← FIRST

This function inserts a new element **VAL** after specified node of the linked list.

FIRST: a pointer which contains address of first node in the list

N: Specified node value

NEW : temporary new node

AVAIL : Top node of availability list

INFO: stores Data of node

LINK: stores pointer to next node

PTR,PREPTR : Temporary node pointers for traversal

Insertion AFTER SPECIFIED node in SLL

INSERTAFTER (VAL,FIRST,N)

6. [Reach to specific location]
   Repeat while INFO (PREPTR) ≠ N
   PREPTR←PTR
   PTR←LINK(PTR)
7. [Insert new node after given location]
   LINK(PREPTR)←NEW
   LINK(NEW)←PTR
8. [Finished]

This function inserts a new element **VAL** after specified node of the linked list.

FIRST: a pointer which contains address of first node in the list

N: Specified node value

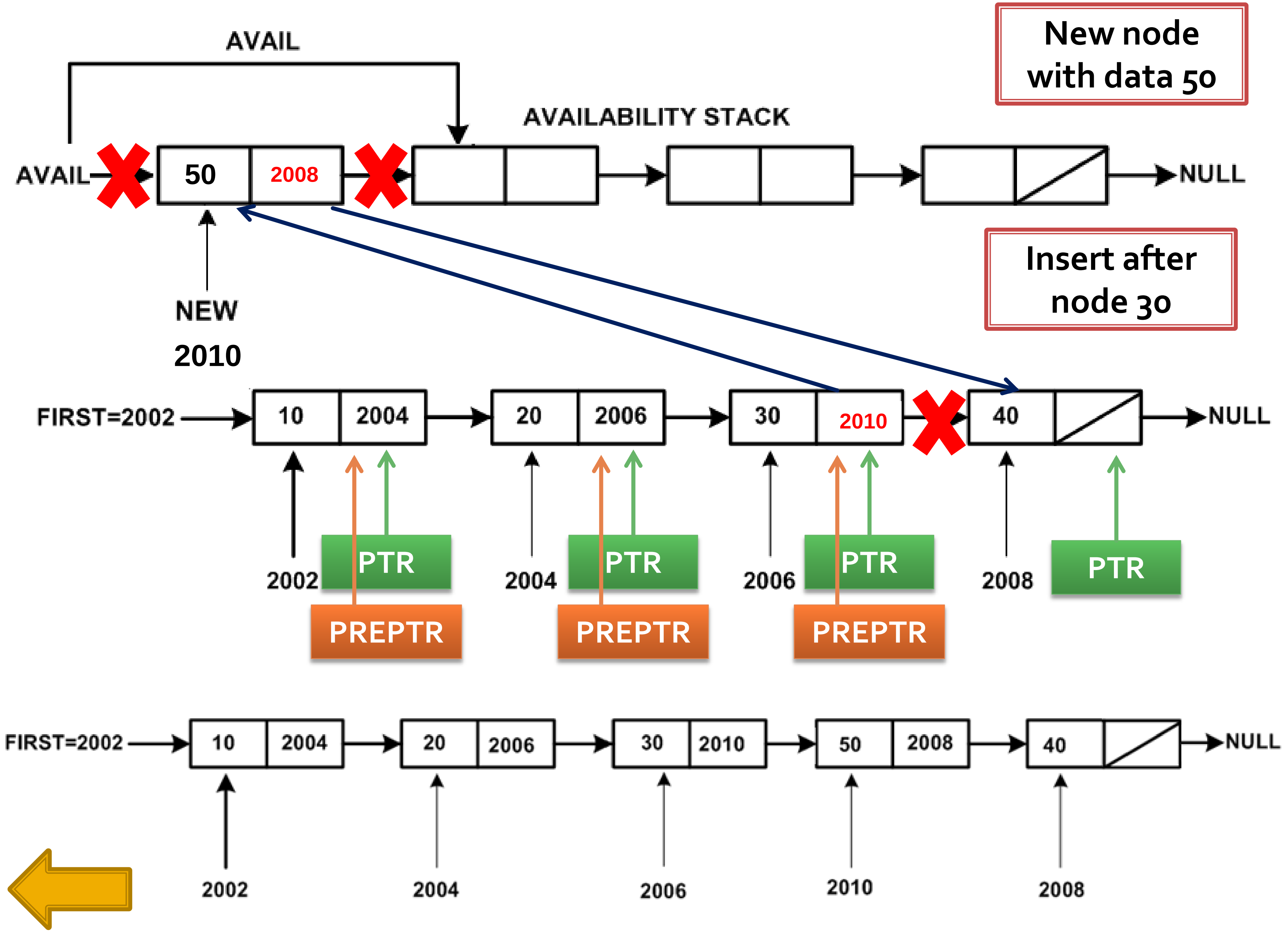
NEW : temporary new node

AVAIL : Top node of availability list

INFO: stores Data of node

LINK: stores pointer to next node

PTR,PREPTR : Temporary node pointers for traversal



New node with data 50
AVAIL
AVAILABILITY STACK
AVAIL
50
2008
NULL
Insert after node 30
NEW
2010
FIRST=2002
10
2004
20
2006
30
2010
40
NULL
PTR
PTR
PTR
PTR
2002
2004
2006
2008
PREPTR
PREPTR
PREPTR
FIRST=2002
10
2004
20
2006
30
2010
50
2008
40
NULL
2002
2004
2006
2010
2008

Insertion into SORTED SLL

INSERTSORTED (VAL,FIRST)

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write “Availability stack underflow”
   Return
2. [Obtain address of next free node]
   NEW←AVAIL
3. [Remove free node from availability stack]
   AVAIL←LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ←VAL
5. [Is list empty?]
   If FIRST = NULL then
   LINK (NEW) ←NULL
   FIRST←NEW
   Return

This function inserts a new element **VAL** into sorted list

FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

AVAIL : Top node of availability list

INFO: stores Data of node

LINK: stores pointer to next node

Insertion into SORTED SLL

6. [Does the new node precede all nodes in the list?]
   If INFO(NEW) ≤ INFO(FIRST) then
   LINK(NEW) ←FIRST
   FIRST←NEW
   Return
7. [[Initialize search pointer]
   SAVE←FIRST
8. [Search for predecessor of new node]
   Repeat while LINK (SAVE) ≠ NULL
   & INFO (LINK(SAVE)) ≤ INFO (NEW)
   SAVE←LINK(SAVE)
9. [insert the node]
   LINK (NEW) ←LINK (SAVE)
   LINK (SAVE) ←NEW
10. [Finished]
    Return (FIRST)

This function inserts a new element **VAL** into sorted list

FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

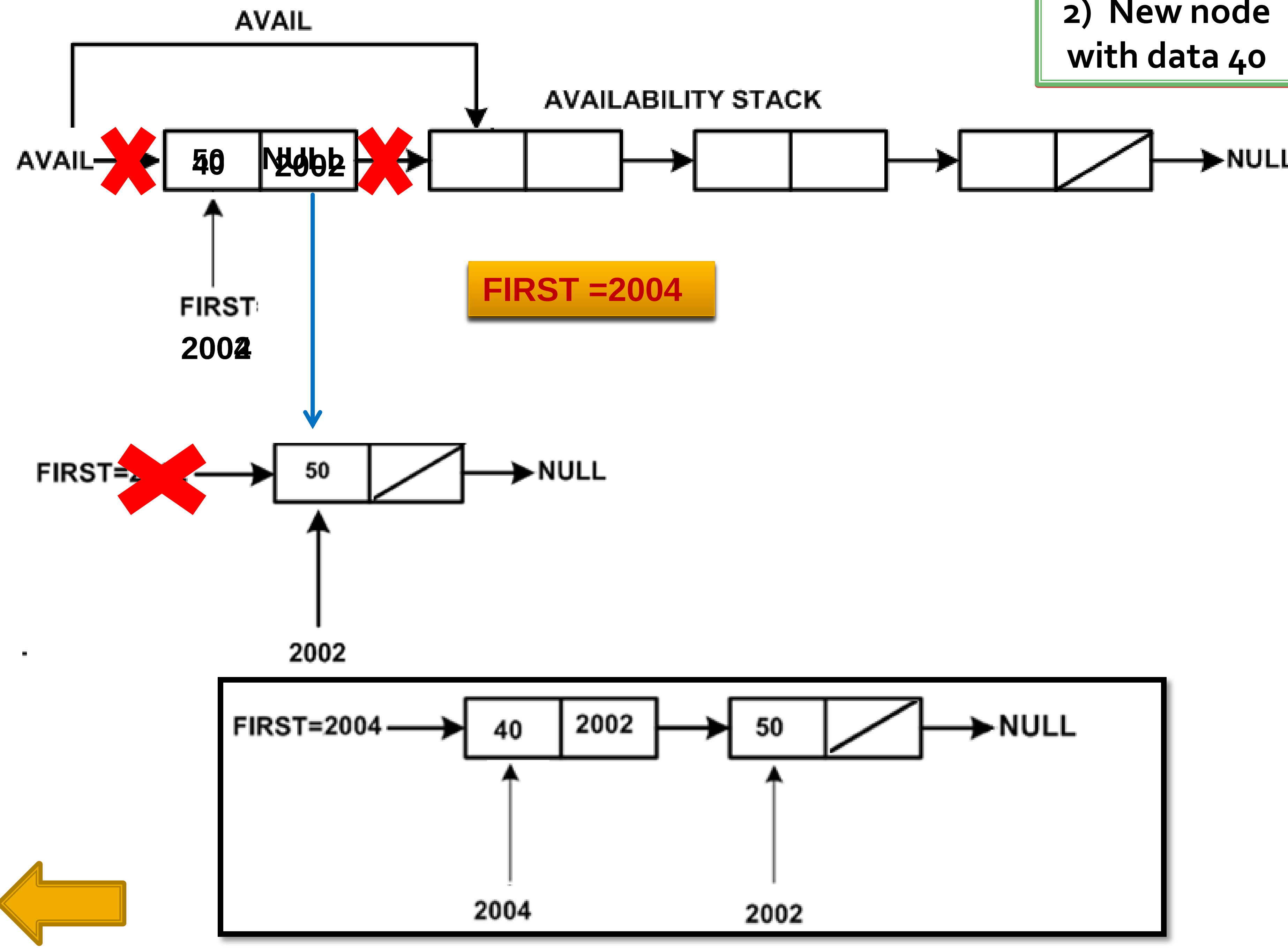
AVAIL : Top node of availability list

INFO: stores Data of node

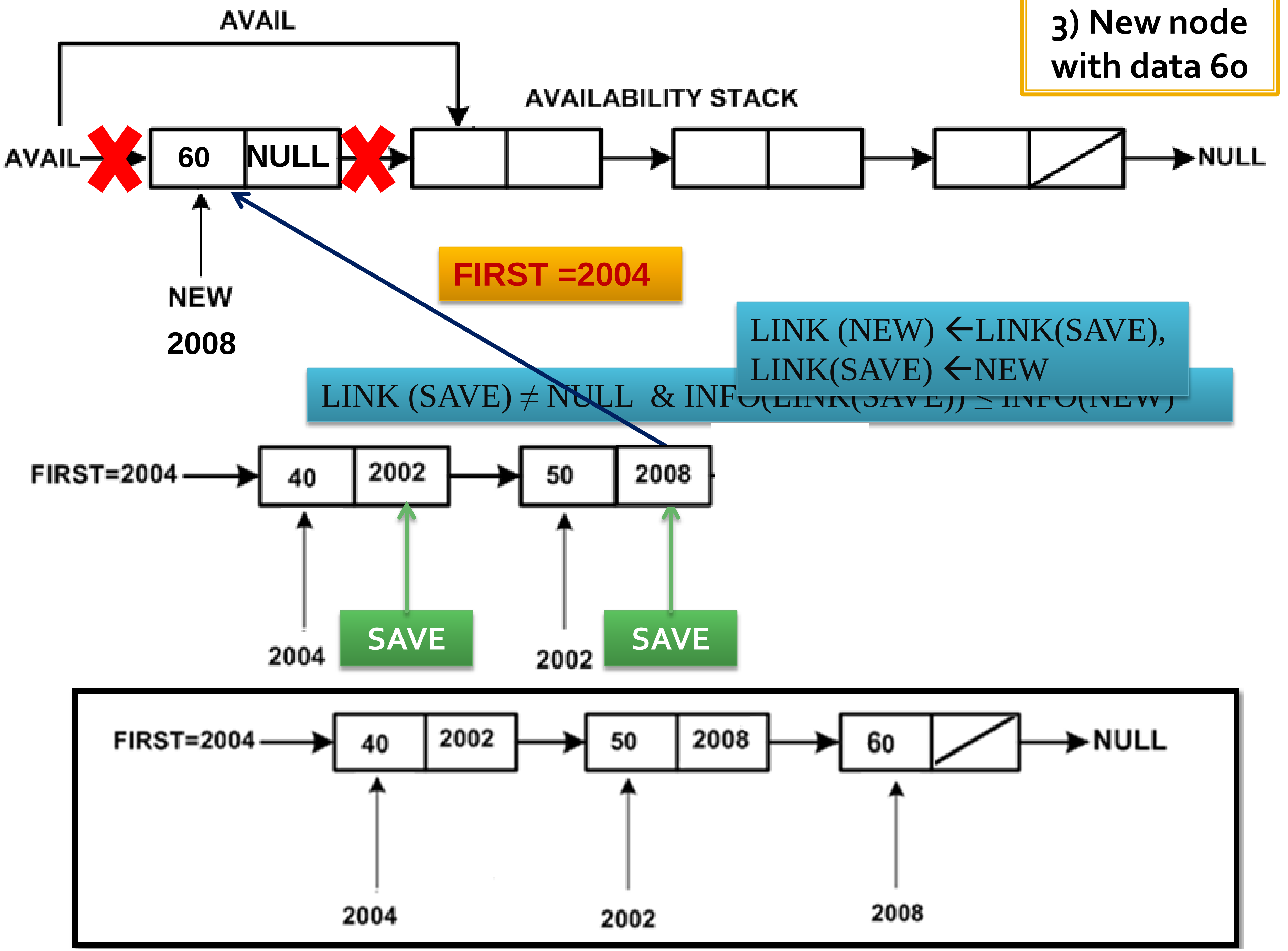
LINK: stores pointer to next node

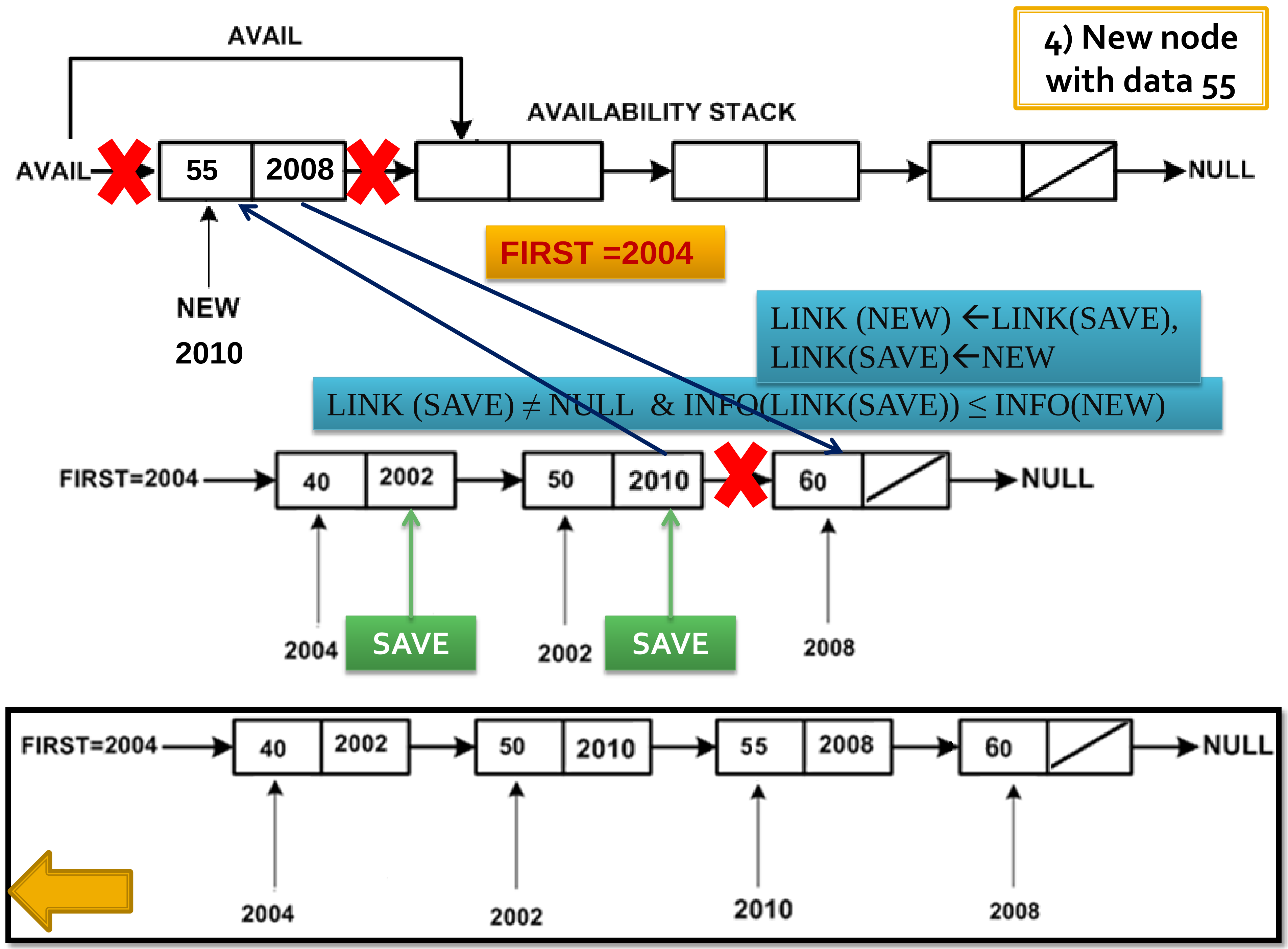
SAVE: Temporary node pointers for traversal

2) New node with data 40



3) New node with data 60





4) New node with data 55
AVAIL
AVAILABILITY STACK
AVAIL
55
2008
NULL
NEW
2010
FIRST =2004
LINK (NEW) ←LINK(SAVE),
LINK(SAVE)←NEW
LINK (SAVE) ≠ NULL & INFO(LINK(SAVE)) ≤ INFO(NEW)
FIRST=2004
40
2002
50
2010
60
NULL
2004
SAVE
2002
SAVE
2008
FIRST=2004
40
2002
50
2010
55
2008
60
NULL
2004
2002
2010
2008

DELETE FIRST node from SLL

DELETEFIRST (FIRST)

1. [Check for empty list]
   If FIRST = NULL then
   Write "List is empty"
   Return
2. [Check for the element and delete it]
   If LINK(FIRST) = NULL then
   Free(FIRST)
   FIRST ← NULL
   else
   TEMP ← FIRST
   FIRST ← LINK(TEMP)
   Free(TEMP)
3. [Finished]
   Return(FIRST)

This function deletes **FIRST** node from the linked list.

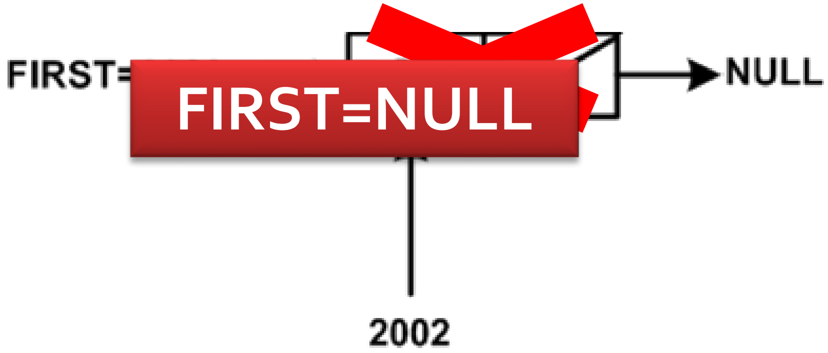
FIRST: a pointer which contains address of first node in the list

INFO: stores Data of node

LINK: stores pointer to next node

TEMP: Temporary node pointers for traversal

[1]



Link(FIRST)=\NULL ??

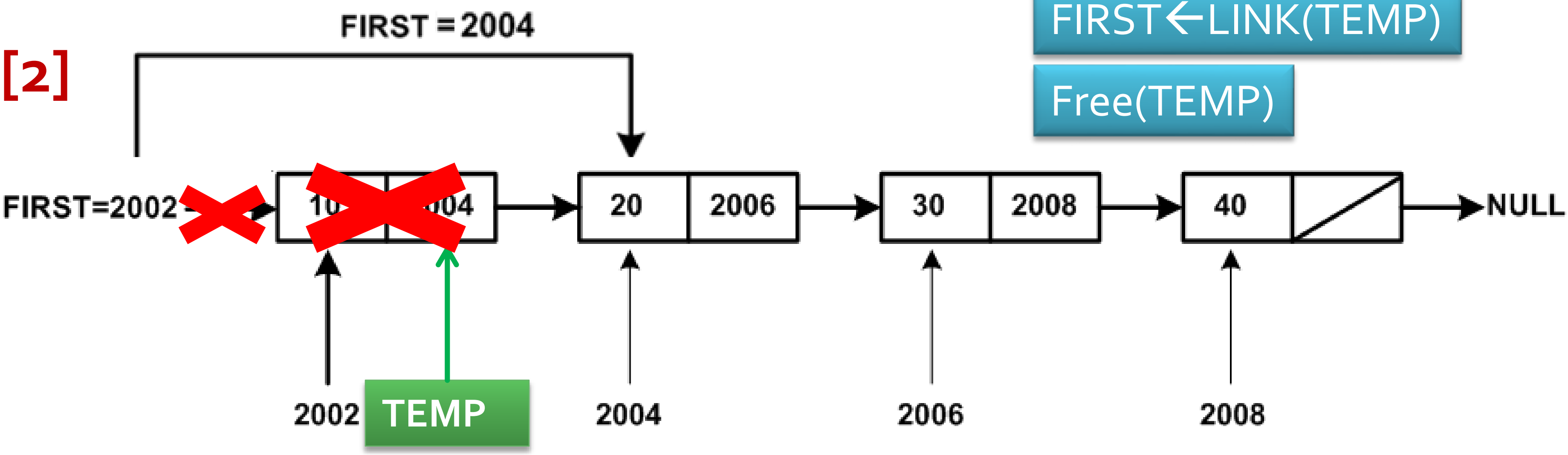
Free(FIRST)

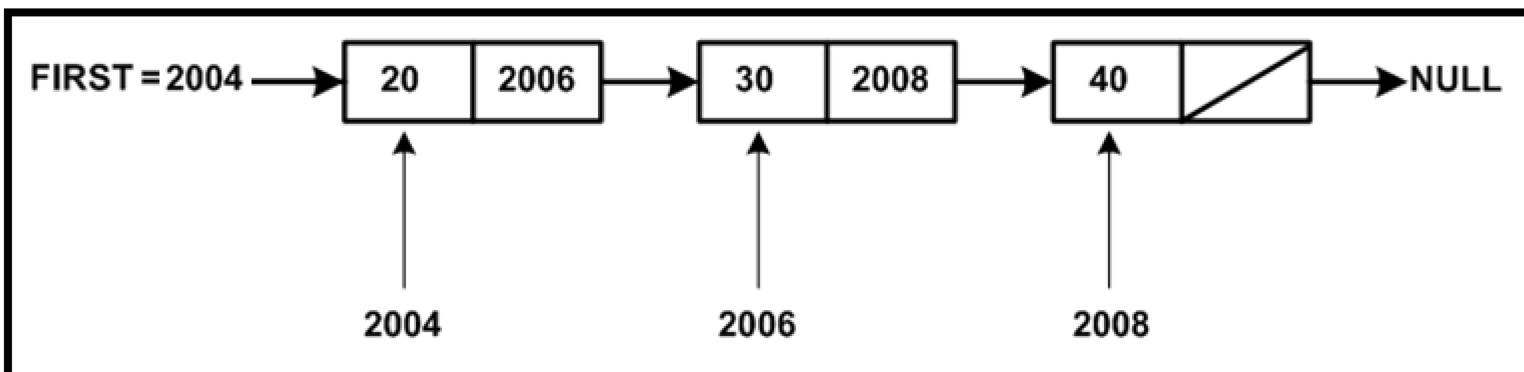
[2]

TEMP]←FIRST

FIRST←LINK(TEMP)

Free(TEMP)





DELETE LAST node from SLL

DELETELAST (FIRST)

1. [Check for empty list]
   If FIRST = NULL then
   Write "List is empty"
   Return
2. [Check for the element and delete it]
   If LINK(FIRST) = NULL then
      Free(FIRST)
      FIRST←NULL
   Else
      PTR←FIRST
      Repeat while LINK (PTR) ≠ NULL
         PREPTR←PTR
         PTR←LINK (PTR)
      [Delete the last node]
      LINK(PREPTR)←NULL
      FREE(PTR)
3. [Finished]
   Return(FIRST)

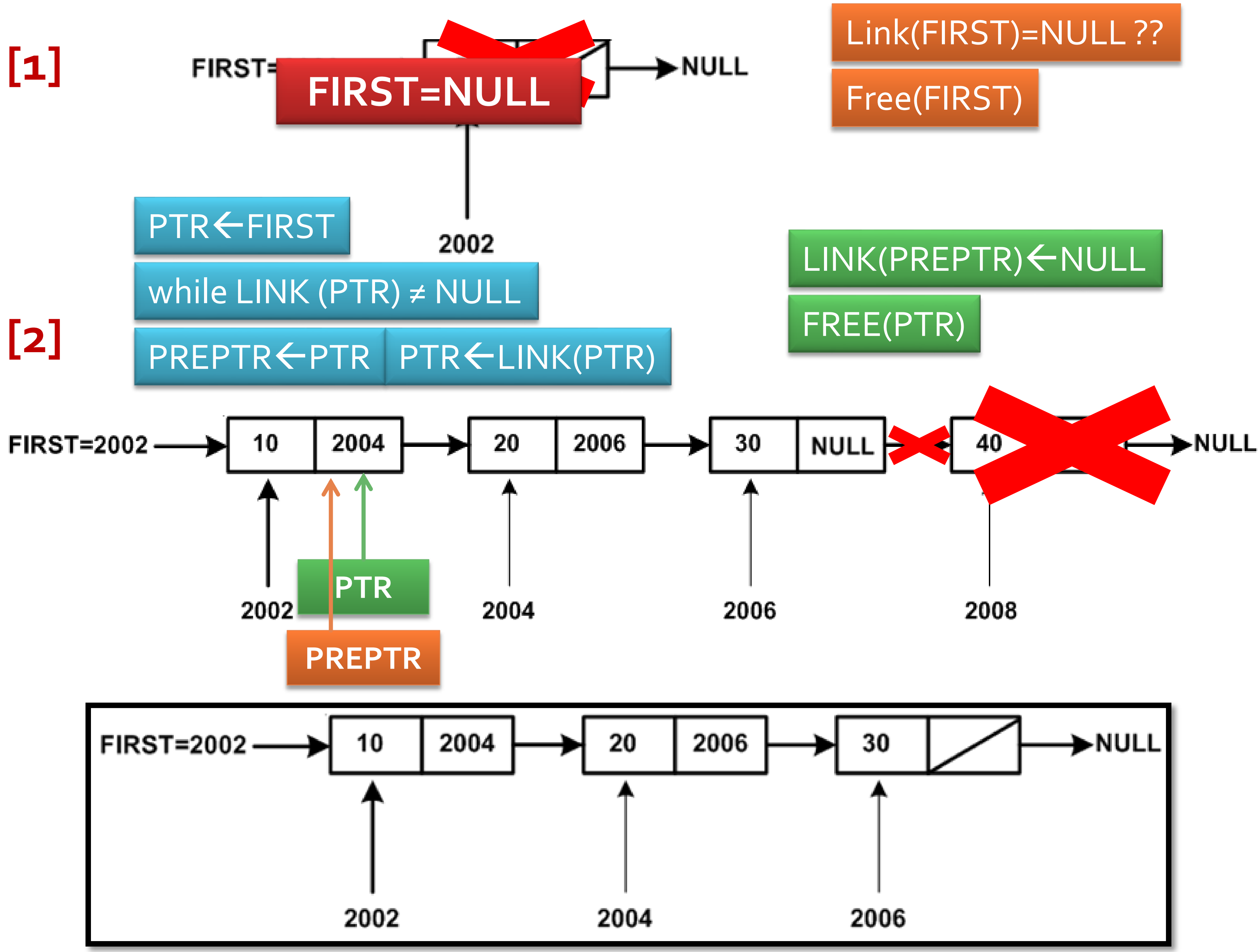
This function deletes the **LAST** node from the linked list.

FIRST: a pointer which contains address of first node in the list

INFO: stores Data of node

LINK: stores pointer to next node

PTR,PREPTR: Temporary node pointers for traversal


[1]
FIRST=
FIRST=NULL
NULL
2002
Link(FIRST)=NULL ??
Free(FIRST)
[2]
PTR←FIRST
while LINK (PTR) ≠ NULL
PREPTR←PTR
PTR←LINK(PTR)
LINK(PREPTR)←NULL
FREE(PTR)
FIRST=2002
10
2004
20
2006
30
NULL
40
NULL
PTR
PREPTR
2002
2004
2006
2008
FIRST=2002
10
2004
20
2006
30
NULL
2002
2004
2006

DELETE SPECIFIED node from SLL

DELETESPEC (FIRST,N)

1. [Check for empty list]
   If FIRST = NULL then
   Write "List is empty"
   Return
2. [initialize temporary pointers]
   PTR=FIRST
   PREPTR=PTR
3. [check for the element and delete it]
   If INFO(FIRST)=N then
      FIRST←LINK(PTR)
      FREE(PTR)
   Else
      Repeat while INFO (PTR) ≠ N
         PREPTR←PTR
         PTR←LINK (PTR)
      [Delete the node]
      LINK(PREPTR)← LINK(PTR)
      FREE(PTR)
4. [Finished]
   Return(FIRST)

This function deletes **Specified** node from the linked list.

FIRST: a pointer which contains address of first node in the list

N: value of specified node

INFO: stores Data of node

LINK: stores pointer to next node

PTR,PREPTR: Temporary node pointers for traversal

PTR←PREPTR←FIRST

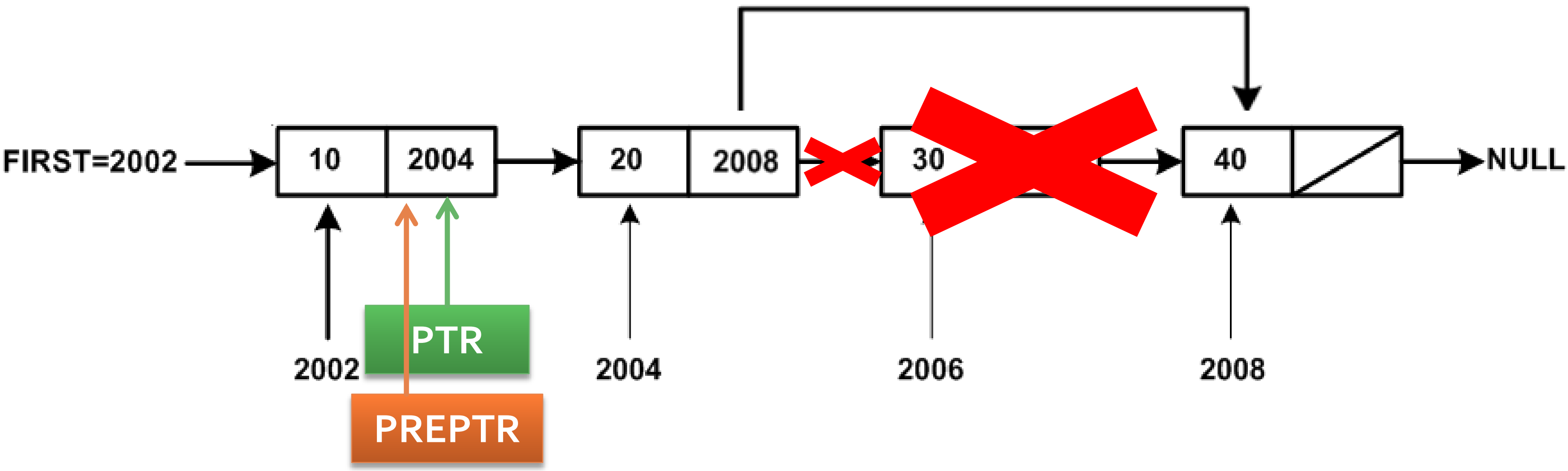
while INFO(PTR) ≠ N

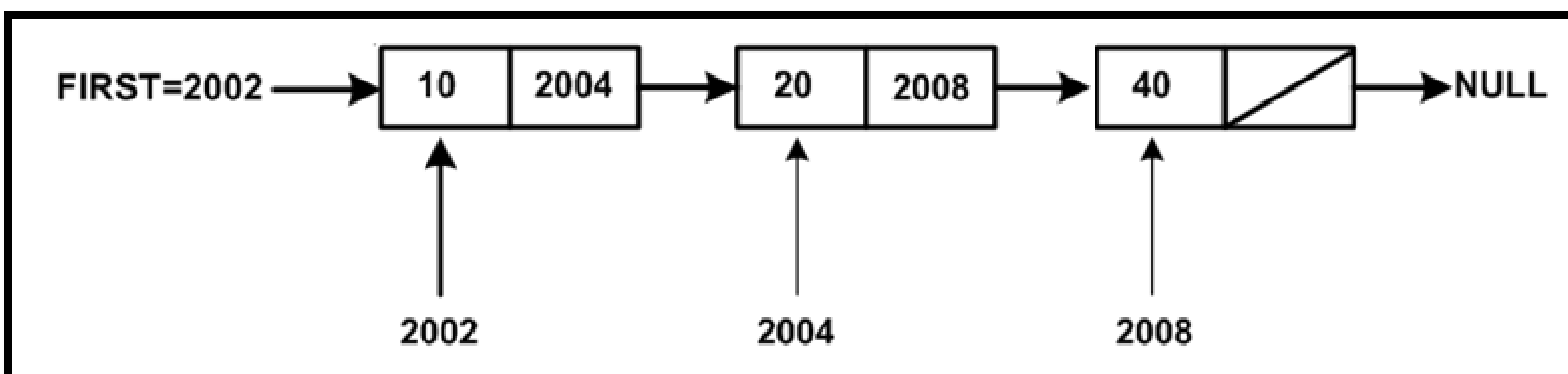
PREPTR←PTR PTR←LINK(PTR)

LINK(PREPTR)←LINK(PTR)

FREE(PTR)

Delete node with value 30





SEARCH node from SLL

SEARCH(FIRST,N)

1. [Check for empty list]
   If FIRST = NULL then
   Write "List is empty"
   Return
2. [Initialize Flag and PTR]
   Flag=0
   PTR=FIRST
3. [Traverse entire list for N]
   Repeat while LINK(PTR) ≠ NULL
      If (INFO(PTR)=N) Then
         Flag←1
         write "node found"
         break;
      Else
         PTR←LINK(PTR)
4. [in case of node not found]
   If Flag=0 then
   write "node not found"
5. [Finished]
   Exit

This function searches **Specified** node from the linked list.

FIRST: a pointer which contains address of first node in the list

N: value to be searched

INFO: stores Data of node

LINK: stores pointer to next node

PTR: Temporary node pointers for traversal

Flag : variable to check node found or not

PTR←FIRST

Flag ← 0

While LINK(PTR) != NULL

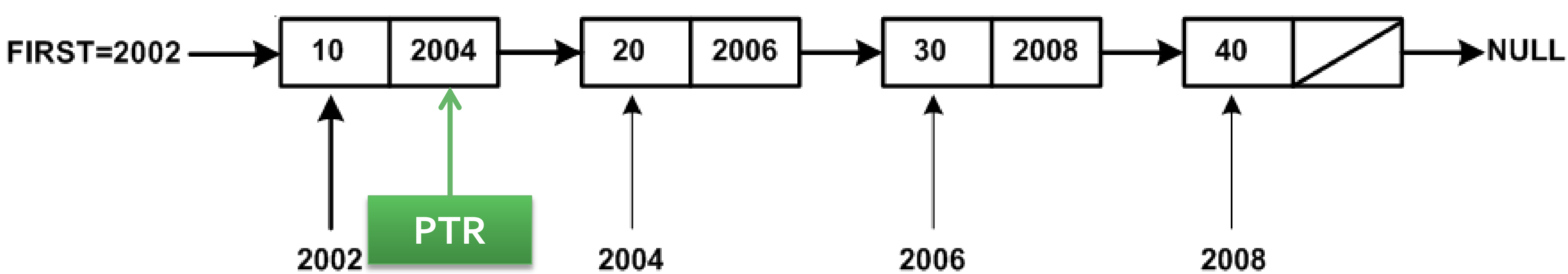
IF INFO(PTR)=N?

Flag ←1 and break out of loop

Else PTR←LINK(PTR)

searching node with value 30

Node Found



COUNT number of nodes in SLL

COUNT(FIRST)

1. [Check for empty list]
   If FIRST = NULL then
   Write "List is empty"
   Return
2. [Initialize Flag and PTR]
   Count=1
   PTR=FIRST
3. [Traverse entire list until end]
   Repeat while LINK(PTR) $\neq$ NULL
      Count=Count+1
      PTR$\leftarrow$LINK(PTR)
4. [Display count]
   write(Count)
5. [Finished]
   Exit

This function counts number of nodes **in** the linked list.

FIRST: a pointer which contains address of first node in the list

INFO: stores Data of node

LINK: stores pointer to next node

PTR: Temporary node pointers for traversal

Count : variable to check node found or not

PTR←FIRST

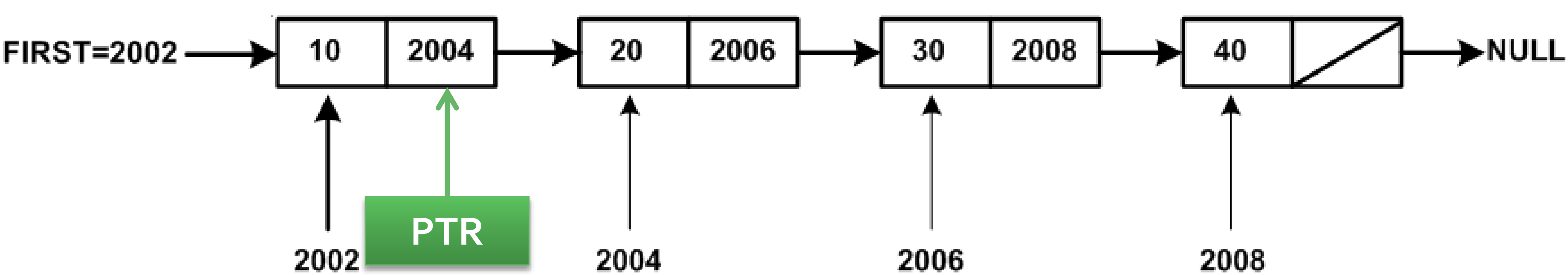
Count ← 1

While LINK(PTR) != NULL

Count←Count+1

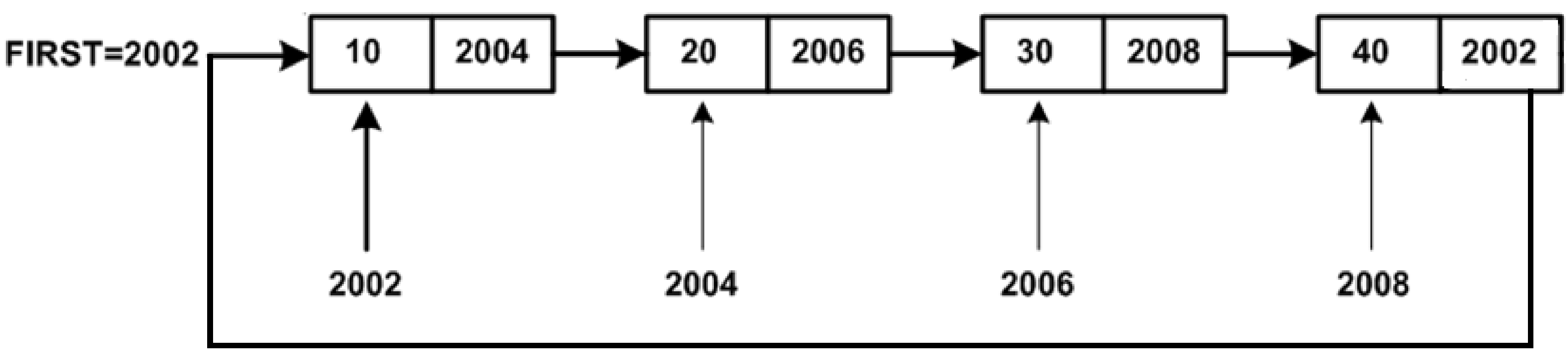
PTR←LINK(PTR)

Count: 4



CIRCULAR Singly Linked List

- If the pointer of the last node contains address of first node then it is known as circular singly linked list.

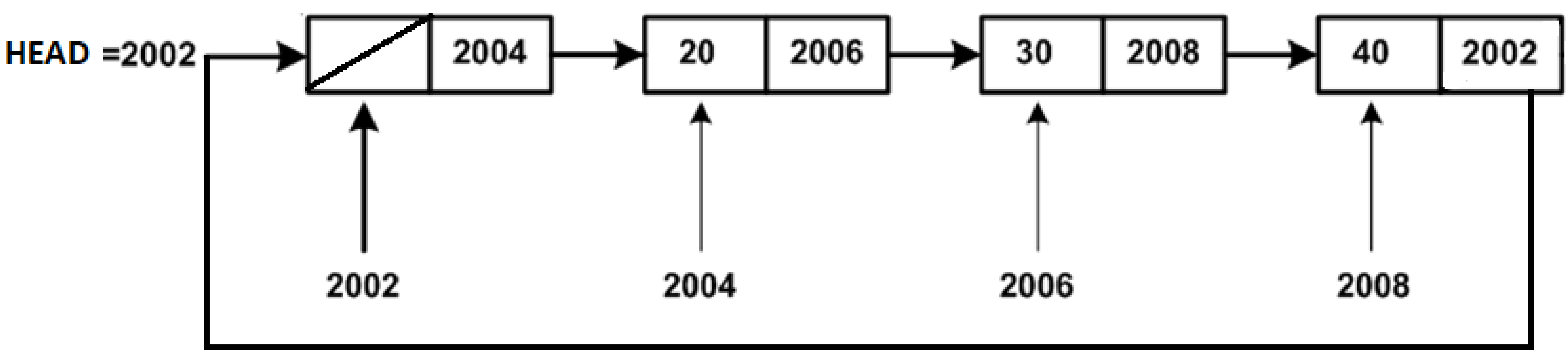


Advantages of circular SLL

- Every node is accessible from a given node i.e. from any given node, all nodes can be reached by chaining through the list.
- To delete a node from a singly linked list. It is require to have the first node’s address. Such a requirement does not exist for a circular linked list.
- Certain operation such as splitting, concatenation becomes more efficient in the circular list.

Disadvantages of Circular SSL

- Without some care in processing it is possible to get into the infinite loop. So we must able to detect end of the list.
- To detect the end of the list in circular linked list we use one special node called the HEAD node.
- INFO field of HEAD node is not used
- For empty list HEAD node points to itself (HEAD=HEAD)



Difference between SLL and Circular SLL

SINGLY LINKED LIST

- SLL has beginning and end.
- Last node link is NULL
- Scanning for node always starts from FIRST node in SLL.
- ALL nodes are not accessible from any given node
- No infinite looping problem

CIRCULAR SINGLY LINKED LIST

- Circular SLL has no end.
- Last node link points to first node of the list.
- Scanning can start from any node in SLL
- All nodes are accessible from any given node
- To prevent infinite loop special HEAD node is required

DOUBLY LINKED LIST

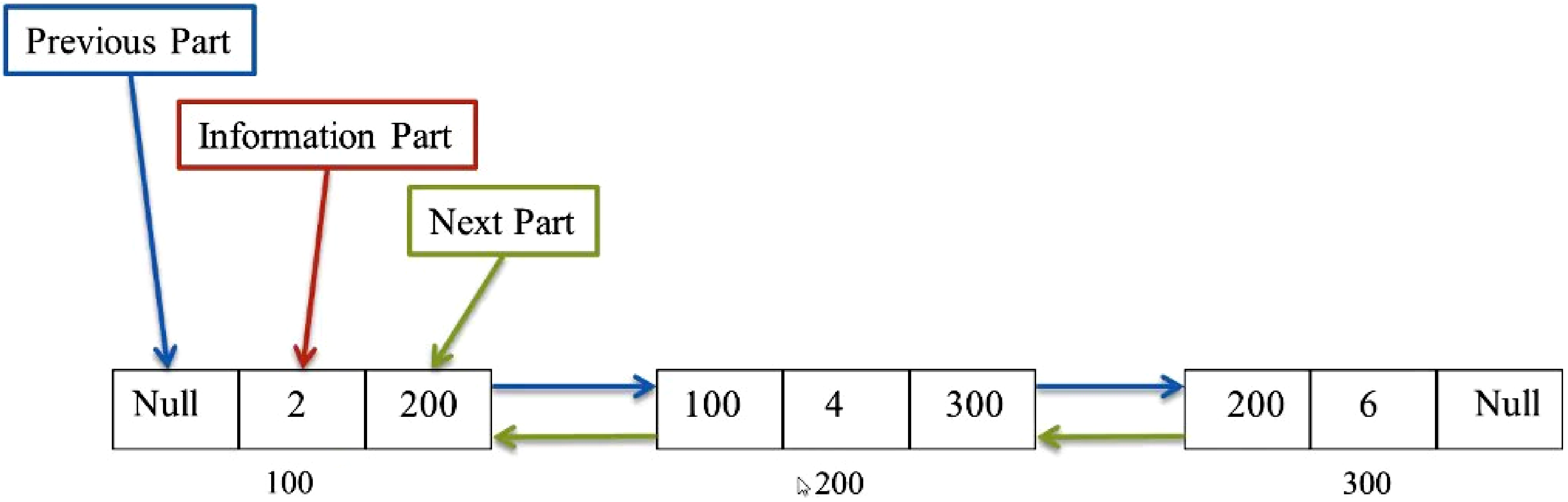
- In SLL traversing is possible in one direction.
- Sometimes it is required to traverse in both the directions : **forward and backward**
- Two link fields are required in a node for traversing in both the directions
- One link denote the **predecessor** of the node and another link is used to denote the **successor** of the node.
- Thus each node in the list consist of three fields:
  - **INFO:** To store data
  - **LPTR :** Pointer to the previous node (predecessor)
  - **RPTR :** Pointer to the next node (successor)

LPTR	INFO	RPTR

DOUBLY LINKED LIST

"A list in which each node contains two links one to the predecessor of the node and one to the successor of the node is known as doubly linked linier list or two way chain"

- The **left link of the left most node and right link of right most node** are set to **NULL** to indicate end of the list in each direction.



DOUBLY LINKED LIST

Advantages over SLL:

- In singly linked list we can traverse only in one direction while in doubly linked list we can traverse the list in both directions.
- Deletion and insertion operation is faster in doubly linked list as compared to the SLL.
- In SLL, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages :

- DLL requires more operations for insertion and deletion (Two pointers needed to be updated).
- DLL requires more space to store node.

SINGLE LINKED LIST	DOUBLE LINKED LIST
A linked list that contains nodes which have a data field and a next field which points to the next node in the line of nodes	A linked list that contains the data field, next field that points to the next node and a previous field that points to the previous node in the sequence
Allows traversing in one direction through the elements	Allows traversing in both directions (backward and forward)
Requires less memory as it stores only one address	Requires more memory as it stores two address
Complexity of insertion and deletion at a known position is O(n)	Complexity of insertion and deletion at a known position is O(1)

https://pediaa.com/wp-content/uploads/2018/12/Difference-Between-Egg-Freezing-and-Embryo-Freezing-Comparison-Summary1.jpg

Operations on Doubly Linked List

- Traversing a linked list.
- Insert new node at beginning of the list
- Insert new node at end of the list
- Insert new node at any position or in between the list.
- Delete first node of the list
- Delete last node of the list.
- Searching element in list.
- Counting number of nodes in the list

Insertion at the Beginning of DLL

INSERTBEG (VAL,FIRST)

This function inserts a new element **VAL** at the the beginning of the DLL

FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

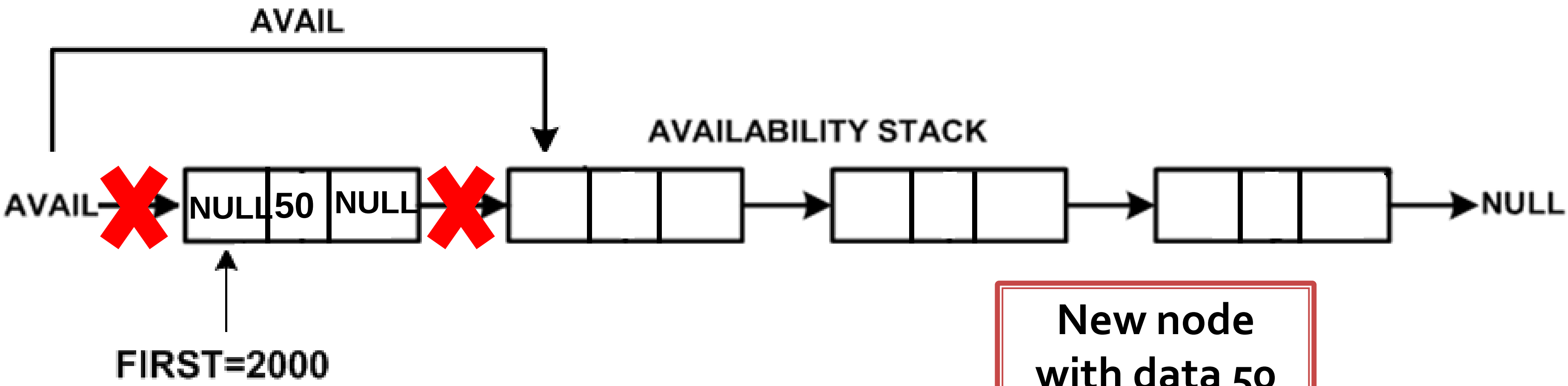
AVAIL : Top node of availability list

INFO: stores Data of node

PREV: stores pointer to the previous node

NEXT: stores pointer to the next node

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write "Availability stack underflow"
   Return
2. [Obtain address of next free node]
   NEW←AVAIL
3. [Remove free node from availability stack]
   AVAIL←LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ←VAL
5. [Insert new node]
   If FIRST= NULL then
   PREV(NEW)← NULL
   NEXT(NEW)←NULL
   Else
   PREV(FIRST)←NEW
   PREV(NEW)←NULL
   NEXT(NEW)←FIRST
6. [Assign the address of the Temporary node to the First Node]
   FIRST←NEW
7. [Finished]
   Return (FIRST)

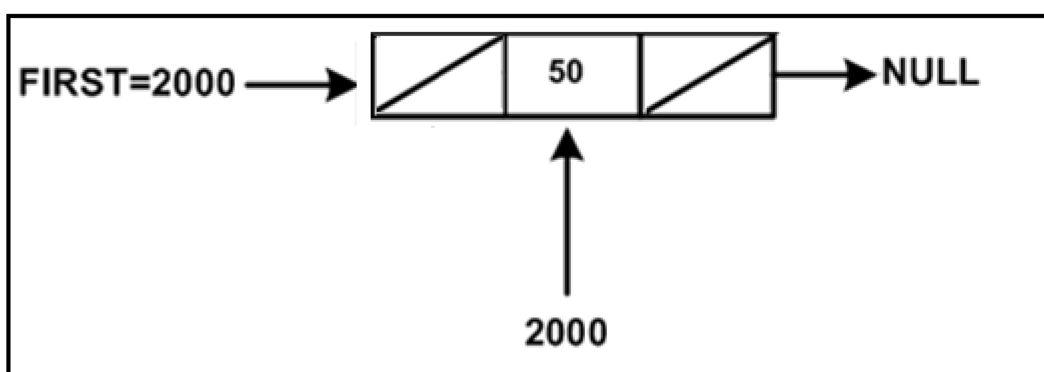


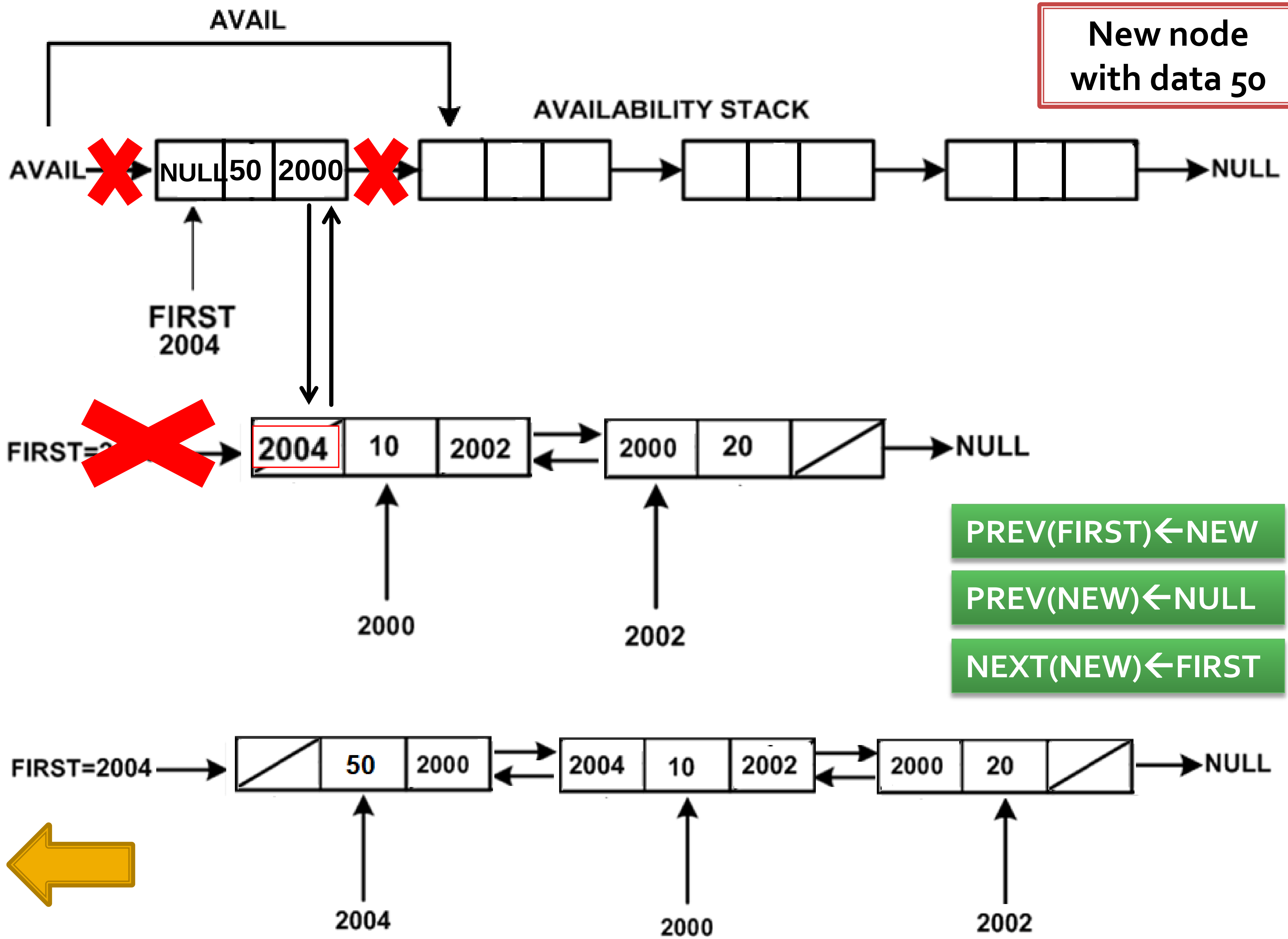
New node with data 50

FIRST←2000

PREV(NEW)←NULL

NEXT(NEW)←NULL





New node with data 50
AVAIL
AVAILABILITY STACK
AVAIL
NULL 50 2000
NULL
FIRST 2004
FIRST=2
2004 10 2002
2000 20
NULL
2000
2002
PREV(FIRST)←NEW
PREV(NEW)←NULL
NEXT(NEW)←FIRST
FIRST=2004
50 2000
2004 10 2002
2000 20
NULL
2004
2000
2002

Insertion at the END of DLL

INSERTEND(VAL,FIRST)

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write “Availability stack underflow”
   Return
2. [Obtain address of next free node]
   NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ← VAL
5. [Insert new node]
   If FIRST= NULL then
   PREV(NEW) ← NULL
   NEXT(NEW) ← NULL
   FIRST ← NEW
   Else
   PTR ← FIRST
   Repeat while NEXT(PTR) ≠ NULL
   PTR ← NEXT(PTR)
   PREV(NEW) ← PTR
   NEXT(NEW) ← NULL
   NEXT(PTR) ← NEW
6. [Finished]
   Return(FIRST)

This function inserts a new element **VAL** at the end of DLL

FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

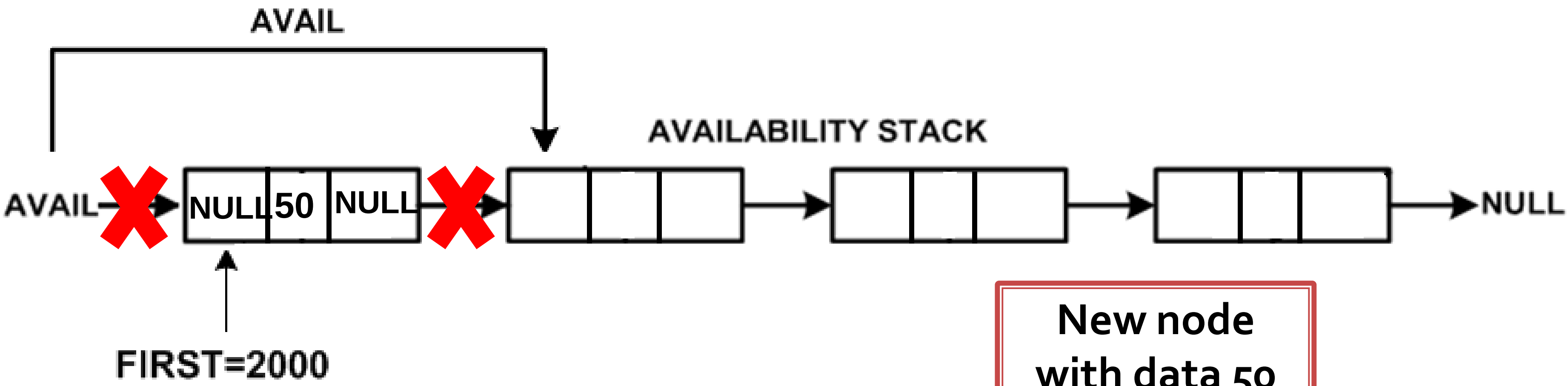
AVAIL : Top node of availability list

INFO: stores Data of node

PREV: stores pointer to the previous node

NEXT: stores pointer to the next node

PTR : temporary pointer for traversal

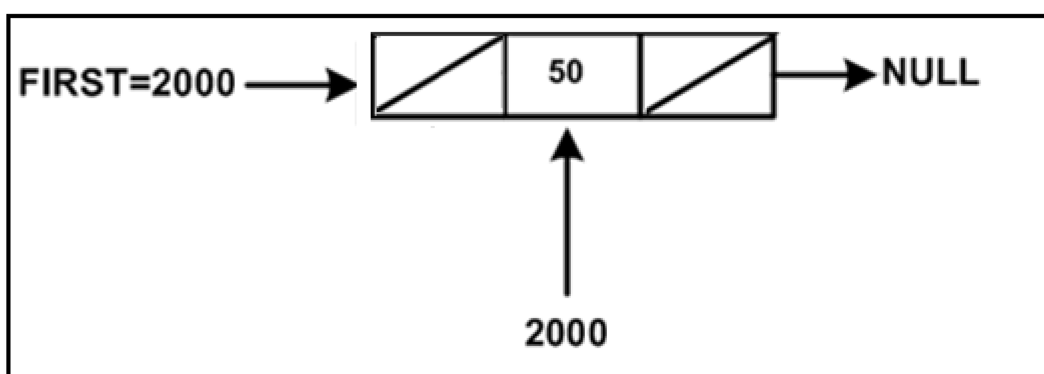


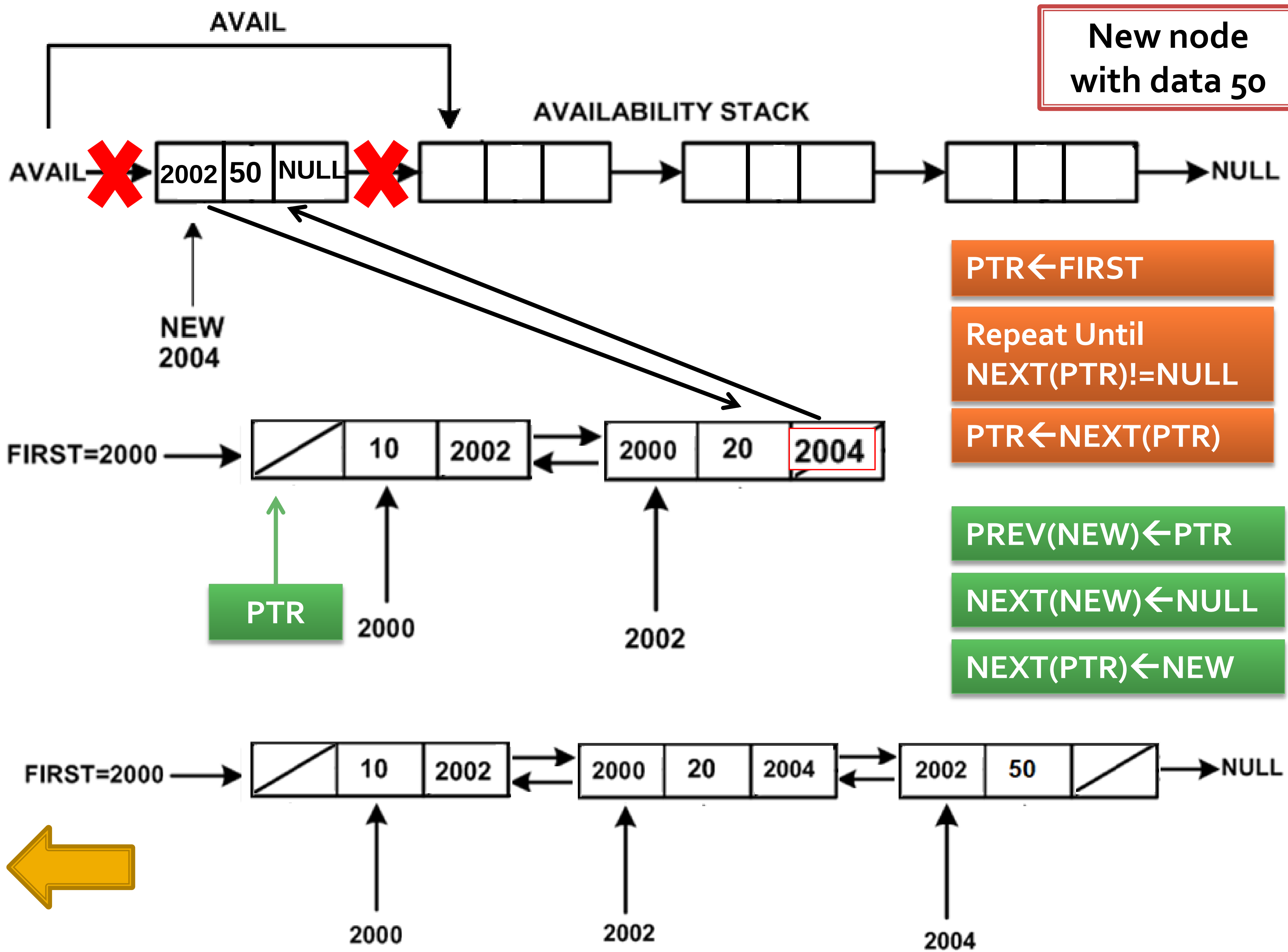
New node with data 50

FIRST←2000

PREV(NEW)←NULL

NEXT(NEW)←NULL



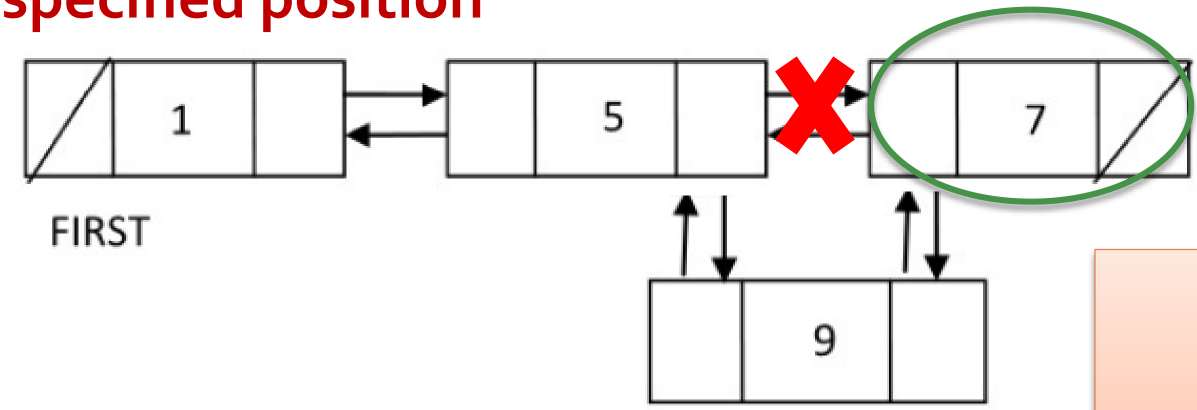


New node with data 50
AVAIL
AVAILABILITY STACK
AVAIL
2002
50
NULL
NULL
NEW
2004
PTR←FIRST
Repeat Until NEXT(PTR)!=NULL
PTR←NEXT(PTR)
FIRST=2000
10
2002
2000
20
2004
PTR
2000
2002
PREV(NEW)←PTR
NEXT(NEW)←NULL
NEXT(PTR)←NEW
FIRST=2000
10
2002
2000
20
2004
2002
50
NULL
2000
2002
2004

Insertion at the SPECIFIED POSITION in DLL

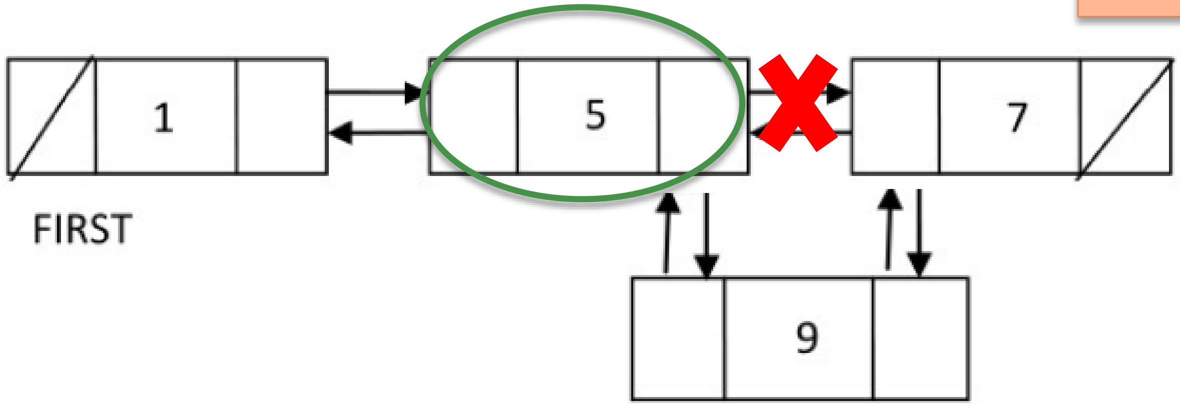
There are two possibilities:

- **Before specified position**



Four
Links
need to be
updated

- **After specified position**



Insertion before specified location in DLL

INSERTBEFORE(VAL,FIRST,N)

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write "Availability stack underflow"
   Return
2. [Obtain address of next free node]
   NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ← VAL
5. [Reach to the specified location]
   PTR ← FIRST
   Repeat while INFO(PTR) ≠ N
       PTR ← NEXT(PTR)
6. [Insert new node]
   If PREV(PTR) = NULL
       FIRST ← NEW
   Else
       NEXT(PREV(PTR)) ← NEW
   PREV(NEW) ← PREV(PTR)
   PREV(PTR) ← NEW
   NEXT(NEW) ← PTR
7. [Finished]
   Return(FIRST)

This function inserts a new element **VAL** before the specified location in DLL

N : DATA value of specified location

FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

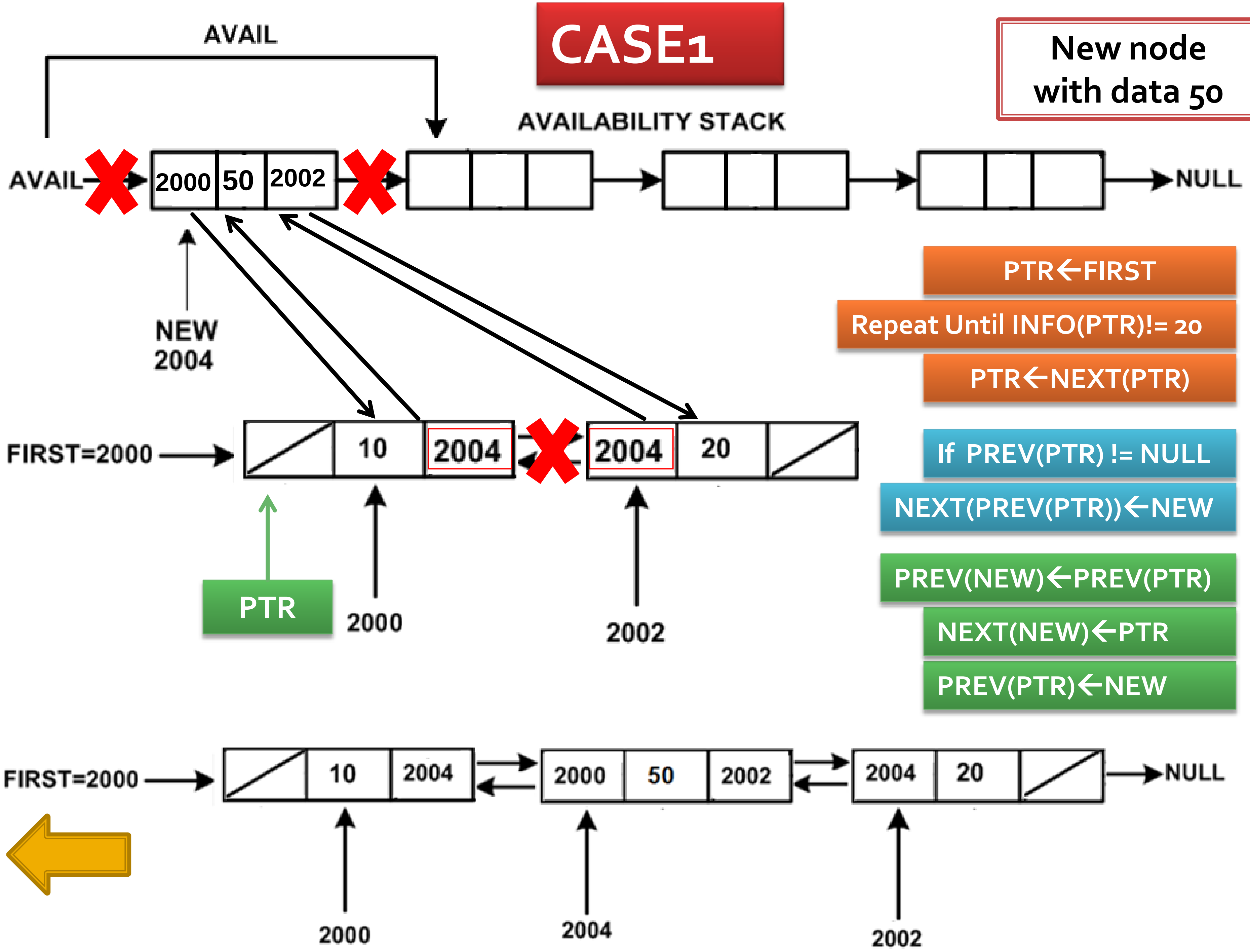
AVAIL : Top node of availability list

INFO: stores Data of node

PREV: stores pointer to the previous node

NEXT: stores pointer to the next node

PTR : Temporary pointer for traversal


CASE1
New node with data 50
AVAIL
AVAILABILITY STACK
AVAIL
2000
50
2002
NULL
NEW
2004
FIRST=2000
10
2004
2004
20
PTR
2000
2002
PTR←FIRST
Repeat Until INFO(PTR)!= 20
PTR←NEXT(PTR)
If PREV(PTR) != NULL
NEXT(PREV(PTR))←NEW
PREV(NEW)←PREV(PTR)
NEXT(NEW)←PTR
PREV(PTR)←NEW
FIRST=2000
10
2004
2000
50
2002
2004
20
NULL
2000
2004
2002

CASE2

New node with data 50

PTR←FIRST

Repeat Until INFO(PTR)!= 10

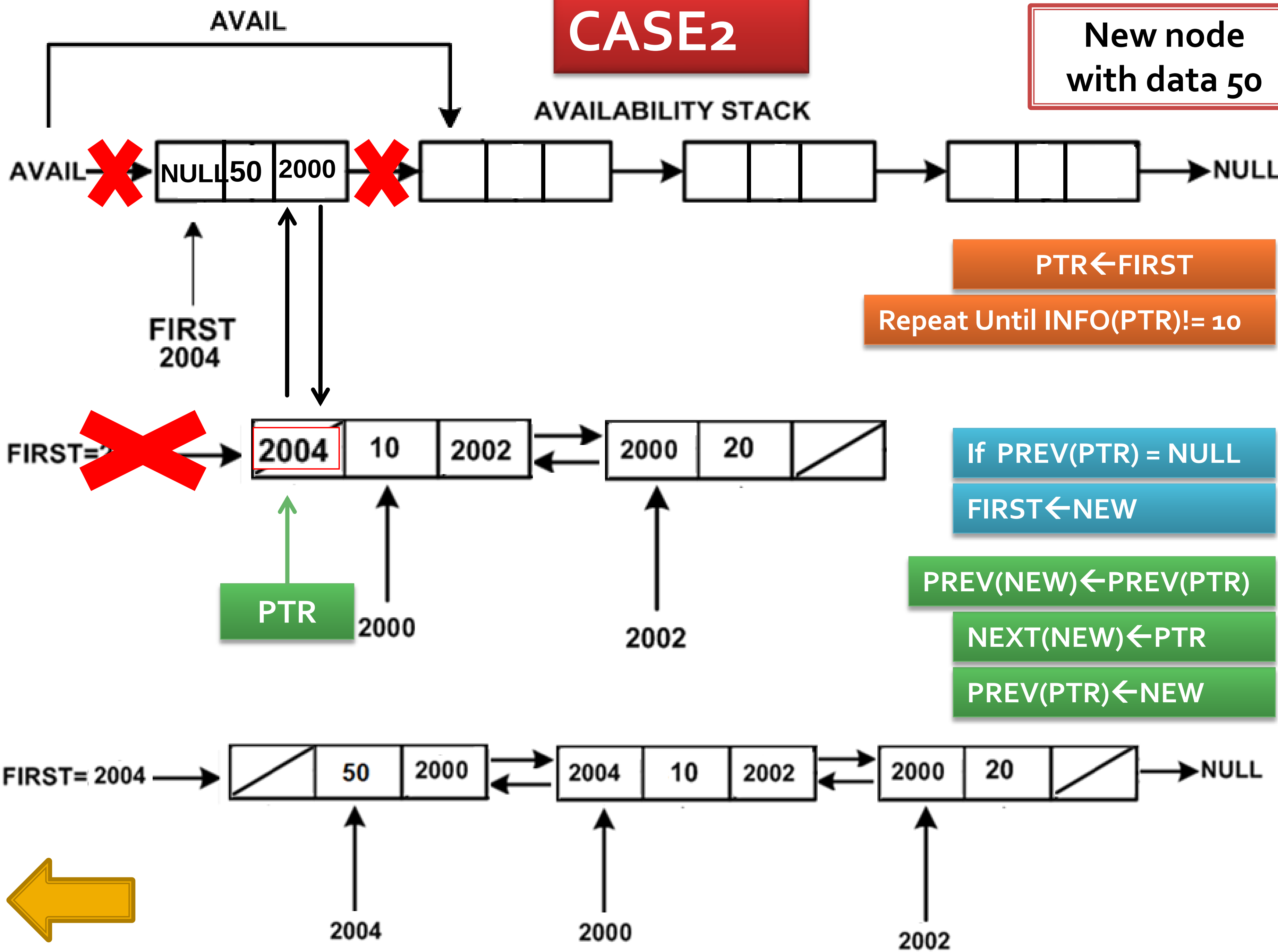
If PREV(PTR) = NULL

FIRST←NEW

PREV(NEW)←PREV(PTR)

NEXT(NEW)←PTR

PREV(PTR)←NEW



Insertion after specified location in DLL

INSERTAFTER(VAL,FIRST,N)

1. [Check for availability stack underflow]
   If AVAIL = NULL then
   Write "Availability stack underflow"
   Return
2. [Obtain address of next free node]
   NEW←AVAIL
3. [Remove free node from availability stack]
   AVAIL←LINK (AVAIL)
4. [Initialize node to the linked list]
   INFO (NEW) ←VAL
5. [Reach to the specified location]
   PTR←FIRST
   Repeat while INFO(PTR) ≠ N
   PTR←NEXT(PTR)
6. [Insert new node]
   If NEXT(PTR) != NULL
   PREV(NEXT(PTR))←NEW
   PREV(NEW)←PTR
   NEXT(NEW)←NEXT(PTR)
   NEXT(PTR)←NEW
7. [Finished]
   Return(FIRST)

This function inserts a new element **VAL** before the specified location in DLL

N : DATA value of specified location

FIRST: a pointer which contains address of first node in the list

NEW : temporary new node

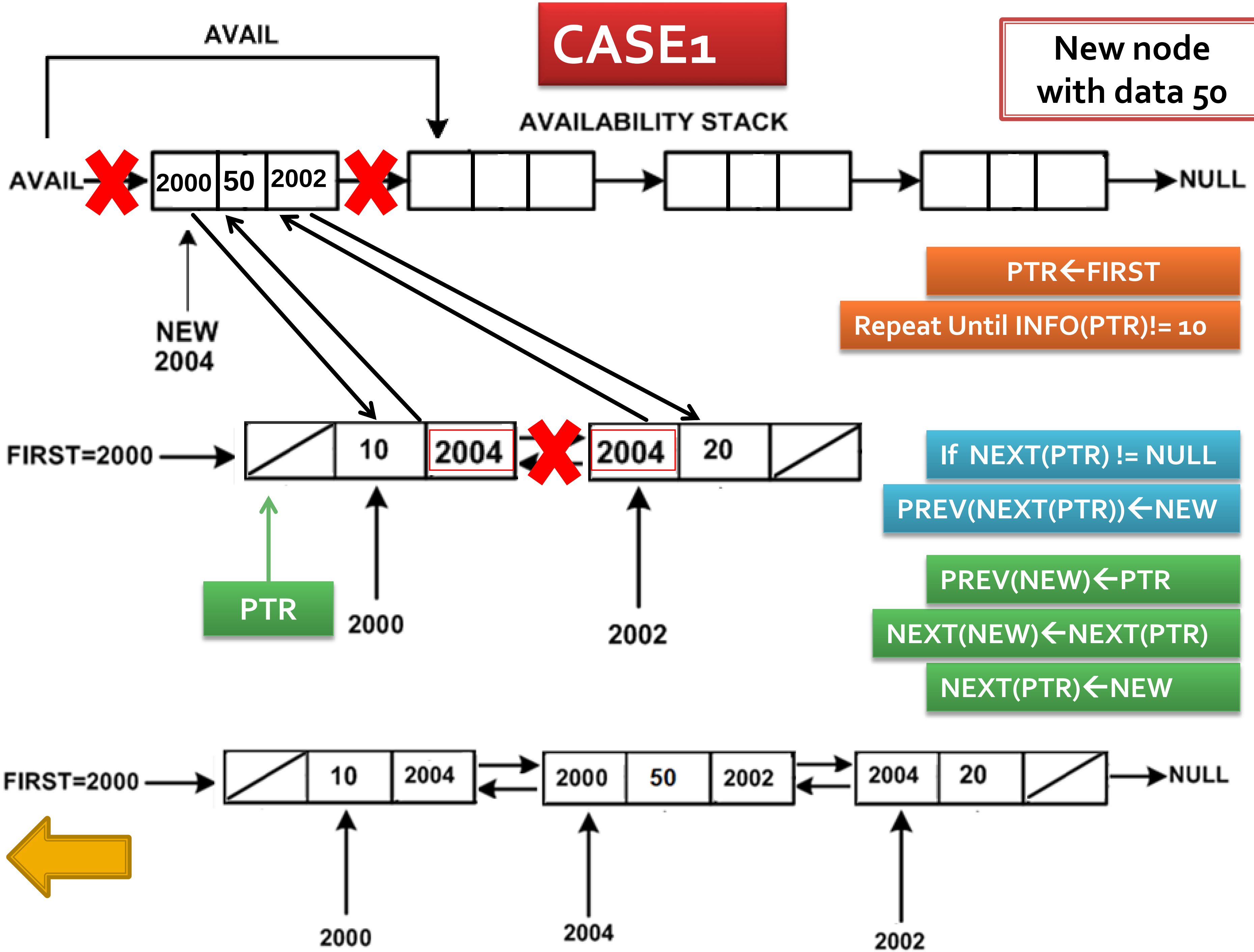
AVAIL : Top node of availability list

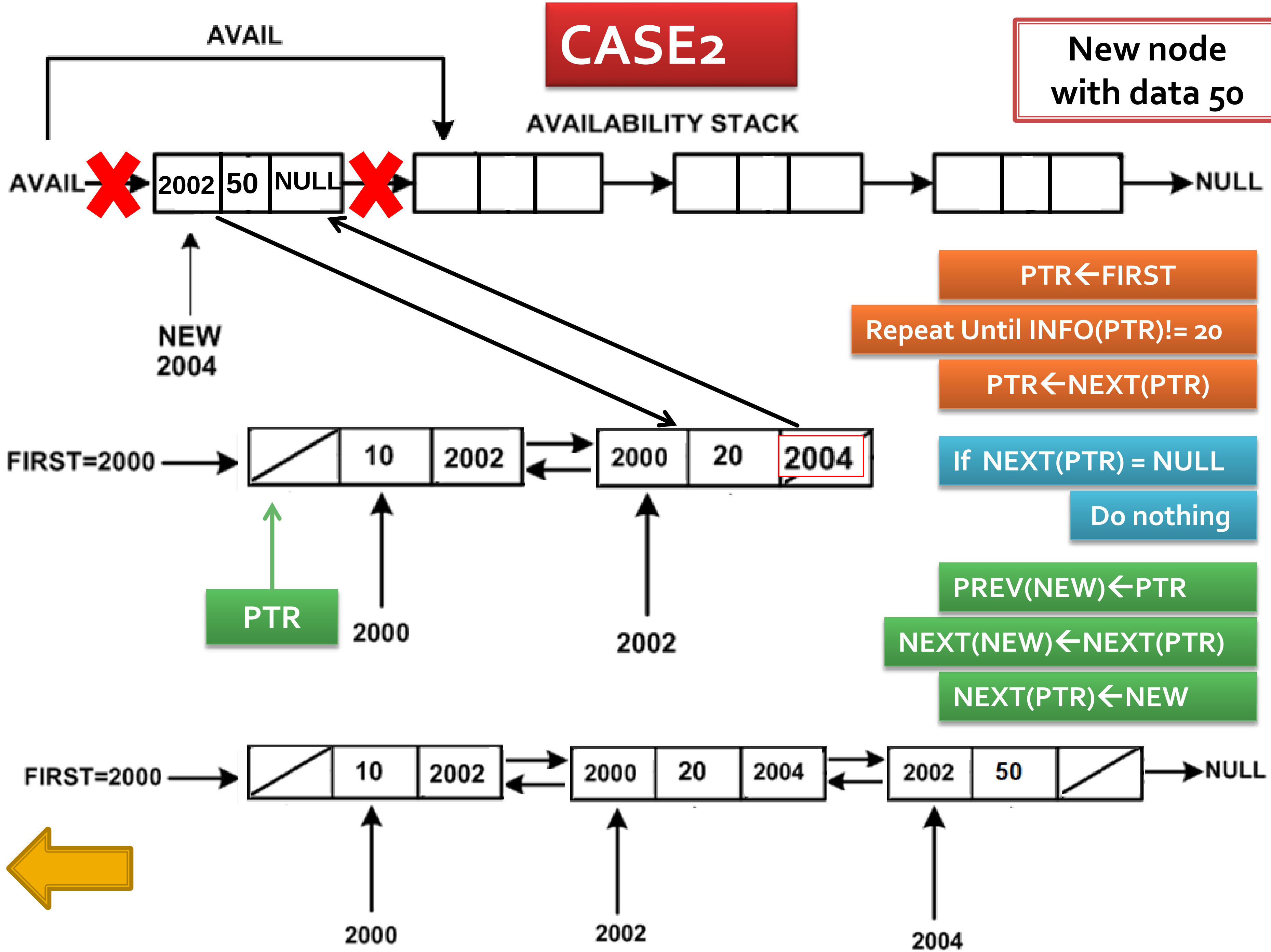
INFO: stores Data of node

PREV: stores pointer to the previous node

NEXT: stores pointer to the next node

PTR : Temporary pointer for traversal


CASE1
New node with data 50
AVAIL
AVAILABILITY STACK
AVAIL
2000
50
2002
NULL
NEW
2004
PTR←FIRST
Repeat Until INFO(PTR)!= 10
FIRST=2000
10
2004
2004
20
If NEXT(PTR) != NULL
PREV(NEXT(PTR))←NEW
PREV(NEW)←PTR
NEXT(NEW)←NEXT(PTR)
NEXT(PTR)←NEW
PTR
2000
2002
FIRST=2000
10
2004
2000
50
2002
2004
20
NULL
2000
2004
2002



CASE2
New node with data 50
AVAIL
AVAILABILITY STACK
AVAIL
2002
50
NULL
NULL
NEW
2004
FIRST=2000
10
2002
2000
20
2004
PTR
2000
2002
PTR←FIRST
Repeat Until INFO(PTR)!= 20
PTR←NEXT(PTR)
If NEXT(PTR) = NULL
Do nothing
PREV(NEW)←PTR
NEXT(NEW)←NEXT(PTR)
NEXT(PTR)←NEW
FIRST=2000
10
2002
2000
20
2004
2002
50
NULL
2000
2002
2004

DELETE FIRST node from DLL

DELETEFIRST (FIRST)

1. [Check for empty list]
   If FIRST = NULL then
   Write “List is empty”
   Return
2. [Take a temporary pointer]
   TEMP ← FIRST
3. [update FIRST to point to next node]
   If NEXT(TEMP)=NULL then
   FIRST=NULL
   Else
   FIRST ← NEXT(TEMP)
   PREV(FIRST) ← NULL
4. [Free the node]
   Free(TEMP)
5. [Finished]
   Return(FIRST)

This function deletes **FIRST** node from the DLL

FIRST: a pointer which contains address of first node in the list

INFO: stores Data of node

NEXT: stores pointer to next node

PREV: stores pointer to previous node

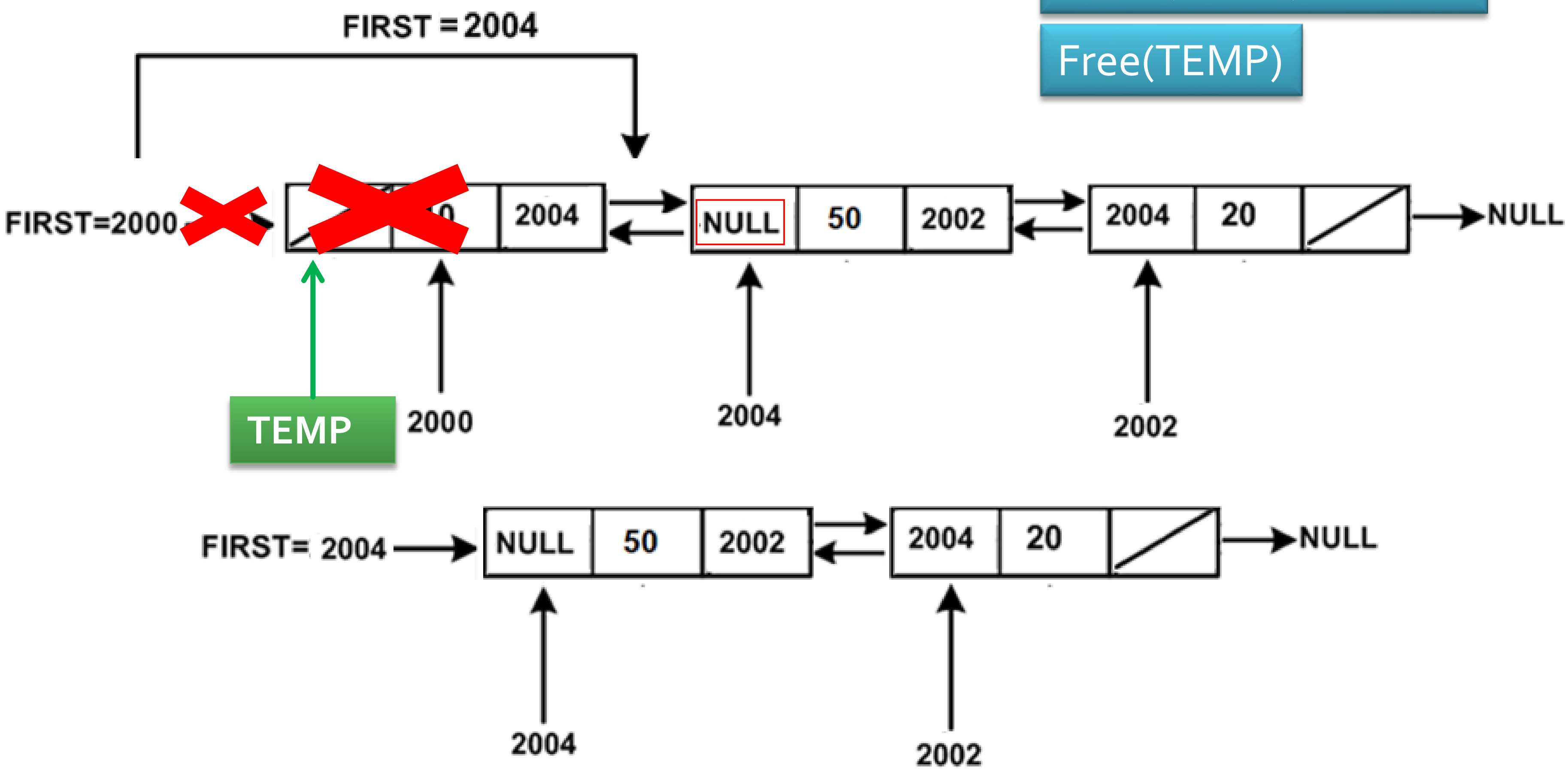
TEMP: Temporary node pointers for traversal

TEMP ← FIRST

FIRST ← NEXT(TEMP)

PREV(FIRST) ← NULL

Free(TEMP)



DELETE LAST node from DLL

DELETELAST (FIRST)

1. [Check for empty list]
   If FIRST = NULL then
   Write "List is empty"
   Return
2. [Take a temporary pointer]
   TEMP ← FIRST
3. [Check for the first node]
   If NEXT(TEMP)=NULL then
      FIRST=NULL
   Else
      Repeat while NEXT(TEMP) ≠ NULL
         TEMP ← NEXT(TEMP)
      [Update NEXT of second last node]
      NEXT(PREV(TEMP)) ← NULL
4. [Free the node]
   Free(TEMP)
6. [Finished]
   Return(FIRST)

This function deletes **LAST** node from the DLL

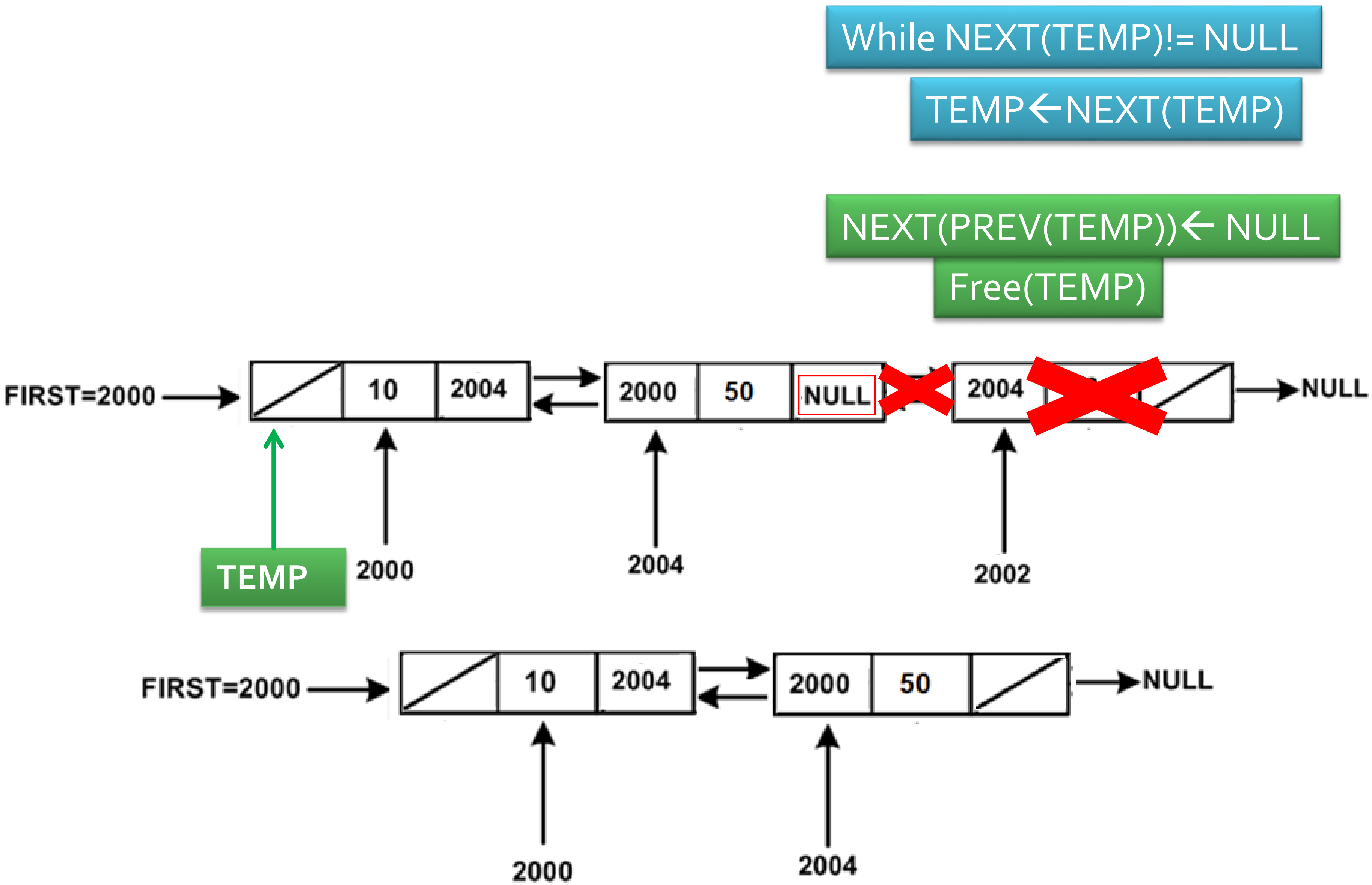
FIRST: a pointer which contains address of first node in the list

INFO: stores Data of node

NEXT: stores pointer to next node

PREV: stores pointer to previous node

TEMP: Temporary node pointers for traversal



While NEXT(TEMP)!= NULL
TEMP←NEXT(TEMP)
NEXT(PREV(TEMP))← NULL
Free(TEMP)
FIRST=2000
10
2004
2000
50
NULL
2004
NULL
TEMP
2000
2004
2002
FIRST=2000
10
2004
2000
50
NULL
2000
2004

Advantages & Disadvantages of LL

Advantages:

- Linked List is **Dynamic data Structure** .
- Linked List **can grow and shrink during run time**.
- **Insertion and Deletion** Operations are Easier
- **Efficient Memory Utilization** ,i.e no need to pre-allocate memory
- Linear Data Structures such as Stack,Queue can be **easily implemeted** using Linked list

Advantages & Disadvantages of LL

Disadvantages:

- **Wastage of memory** due to pointers
- **Searching** for a particular node is difficult and time consuming (no random access).
- Individual nodes are not stored in the contiguous memory Locations so time complexity is more (O(n))
- **Reverse traversing is difficult** in case of singly linked List.
- **Heap space restriction :** Memory is allocated to Linked List at run time if and only if there is space available in heap.

Aplications of Linked List

- Implementation of stacks and queues
- **Implementation of graphs** : Adjacency list representation of graphs.
- **Dynamic memory allocation** : We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list
- representing sparse matrices.
- **Image viewer –** Previous and next images are linked, hence can be accessed by next and previous button.
- **Previous and next page in web browser**
- **Undo and redo** operations in word processor
- Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Application in polynomial

- The polynomial equations are algebraic expression.
- The form of this expression is as below,

$$A_n x^n + A_{n-1} x^{n-1} + A_{n-2} x^{n-2} + \ldots\ldots + A_2 x^2 + A_1 x^1 + A_0 x^0$$

- Where Ai is Co-efficient.

$30x^3 + 20x^2 + 15x + 1$ where a0=1, a1=15, a2=20, a3=30

- For above expression, four nodes are required to store the value of ao,a1,a2& a3.
- Each node contains three part
  - Co- efficient
  - Exponent
  - Address of next node.