Unit 4
Linked List

Covered CO : C205_N.3: Apply basic operations on the linked list data structure.
ESE Weightage: 14 Marks

Topics-Part1

- 4.1 Pointers Revision
- 4.2 Revision of Structure
- 4.3 Revision of structure using pointers
- 4.4 Dynamic Memory Allocation
- 4.5 Linked list Presentation
- 4.6 Types of Linked List

Pointers Revision

- When a variable is created in C, a memory address is assigned to the variable.
- The memory address is the location of where the variable is stored on the computer.
- When we assign a value to the variable, it is stored in this memory address.
- To access memory address, use the reference operator (&), and the result will represent where the variable is stored
- ```
  int myAge = 43;
  printf("%p", &myAge); // Outputs 0x7ffe5367e044
  ```
  ```

Pointers Revision

- Pointer : A **pointer** is a variable that stores the memory address of another variable as its value.

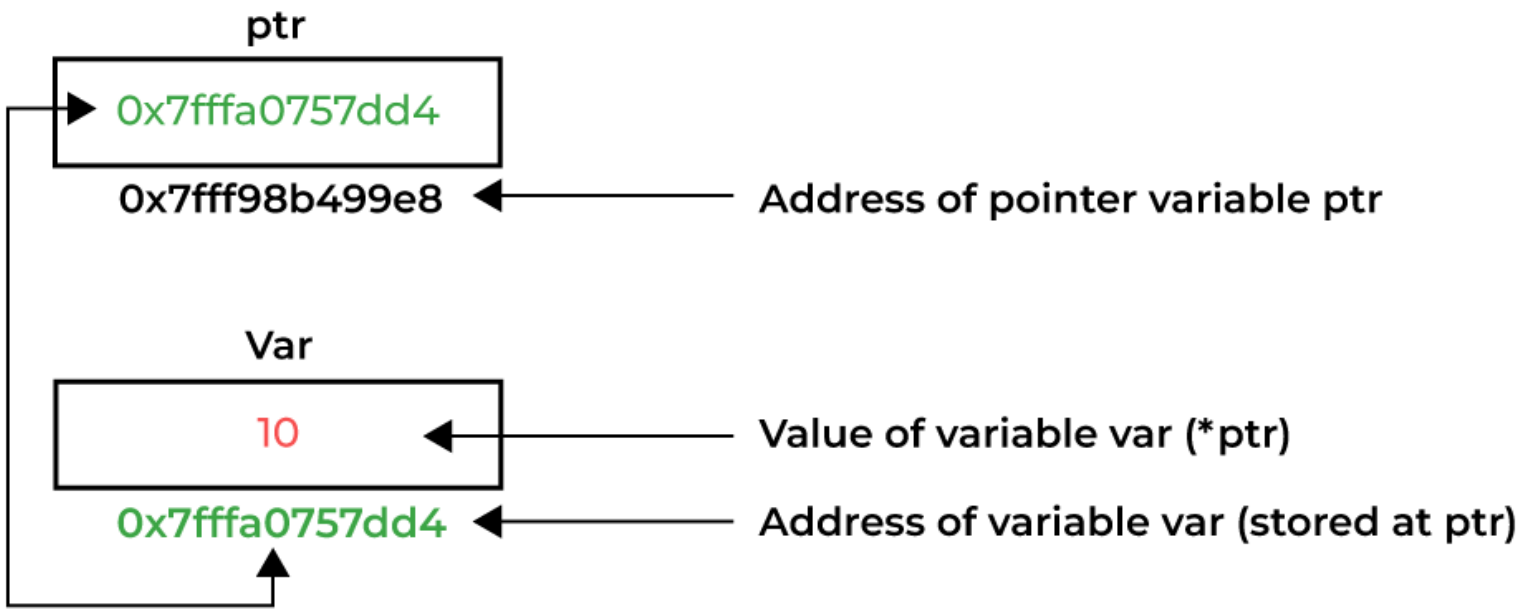
```
int myAge = 43;      // An int variable
int* ptr = &myAge;  // A pointer variable, with the name ptr, that
stores the address of myAge

// Output the value of myAge (43)
printf("%d\n", myAge);

// Output the memory address of myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer
(0x7ffe5367e044)
printf("%p\n", ptr);
```

Pointers Revision



Pointer

- Type of the pointer has to match the type of the variable you're working with.
- However, you can alsoget the value of the variable the pointer points to, by using the * operator (the **dereference** operator):

Pointer

```
int myAge = 43;     // Variable declaration
int* ptr = &myAge;  // Pointer declaration
// Reference: Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);
// Dereference: Output the value of myAge with the pointer (43)
printf("%d\n", *ptr);
```

Advantages of Pointers

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

Disadvantages of Pointers

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are majorly responsible for memory leaks in C.
- Uninitialized pointers might cause a segmentation fault.
- Pointers are comparatively slower than variables in C.
- Pointers are a little bit complex to understand.
-

Revision of Structure

- A structure is a user defined data type that can store related information even of different data type together.
- A structure is therefore a collection of variables under a single name.
- Unlike an array, The variables within a structure are of different data types
- You can create a structure by using the struct keyword and declare each of its members inside curly braces:

Revision of Structure

```
struct MyStructure {   // Structure declaration
  int myNum;           // Member (int variable)
  char myLetter;       // Member (char variable)
}; // End the structure with a semicolon
```

- Now the structure has become a user-defined data type. Each variable name declared within a structure is called a member of the structure.
- The structure declaration, however, does not allocate any memory or consume storage space.
- It just gives a template that conveys to the C compiler how the structure would be laid out in the memory and also gives the details of member names.
- Like any other data type, memory is allocated for the structure when we declare a variable of the structure
- To access the structure, you must create a variable of it.
- Use the struct keyword inside the main() method, followed by the name of the structure and then the name of the structure variable:

```
int main() {
  struct myStructure s1;
// Assign values to members of s1
  s1.myNum = 13;
  s1.myLetter = 'B';
  // Print values
  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);

    return 0;
}
```

Accessing the Members of a Structure

- A structure member variable is generally accessed using a '.' (dot) operator. The syntax of accessing a structure or a member of a structure can be given as: struct_var.member_name

Revision of structure using pointers

- A structure pointer is defined as the pointer which points to the address of the memory block that stores a structure known as the structure pointer.
- The structure pointer tells the address of a structure in memory by pointing the variable to the structure variable.

```
#include <stdio.h>
struct point {
    int value;
};
 int main()
{
    struct point s;
     // Initialization of the structure pointer
    struct point* ptr = &s;
     return 0;
}
```

- There are two ways to access the members of the structure with the help of a structure pointer:

1. With the help of (*) asterisk or indirection operator and (.) dot operator.
2. With the help of (->) Arrow operator.

```c
struct Student s1;
 struct Student* ptr = &s1;

 s1.roll_no = 27;
 strcpy(s1.name, "Kamlesh Joshi");
 strcpy(s1.branch, "Computer Science And Engineering");
 s1.batch = 2019;

 printf("Roll Number: %d\n", (*ptr).roll_no);
 printf("Name: %s\n", (*ptr).name);
 printf("Branch: %s\n", (*ptr).branch);
 printf("Batch: %d", (*ptr).batch);
```

```
#include <stdio.h>
struct student
{
   int rollnum;
   char name[20];
   int marks;
   }s2={2,"riya",60};
int main()
{
   struct student s1 ={1,"ravi",70};
   struct student *ptr;
   ptr=&s1;
   printf("Details of student\n");
   printf("student rollnum=%d\n",s1.rollnum);
   printf("studnet name=%s\n",s1.name);
   printf("Student marks=%d\n",s1.marks);
   printf("student rollnum=%d\n",s2.rollnum);
   printf("studnet name=%s\n",s2.name);
   printf("Student marks=%d\n",s2.marks);
      printf("Details of student\n");
   printf("student rollnum=%d\n",(*ptr).rollnum);
   printf("student rollnum=%d\n",ptr->rollnum);
   return 0;
```

- }

SELF-REFERENTIAL STRUCTURES

- Self-referential structures are those structures that contain a reference to the data of its same type.
- That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure. For example, consider the structure node given below.

```
struct node
{
        int val;
        struct node *next;
};
```

- Here, the structure node will contain two types of data: an integer val and a pointer next. You must be wondering why we need such a structure. Actually, self-referential structure is the foundation of other data structures.
- We will be using them throughout this book and their purpose will be clearer to you when we discuss linked lists, trees, and graphs

DYNAMIC MEMORY ALLOCATION

- The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*.
- Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1.malloc()- allocates single block of requested memory.

2.calloc()-allocates multiple block of requested memory.

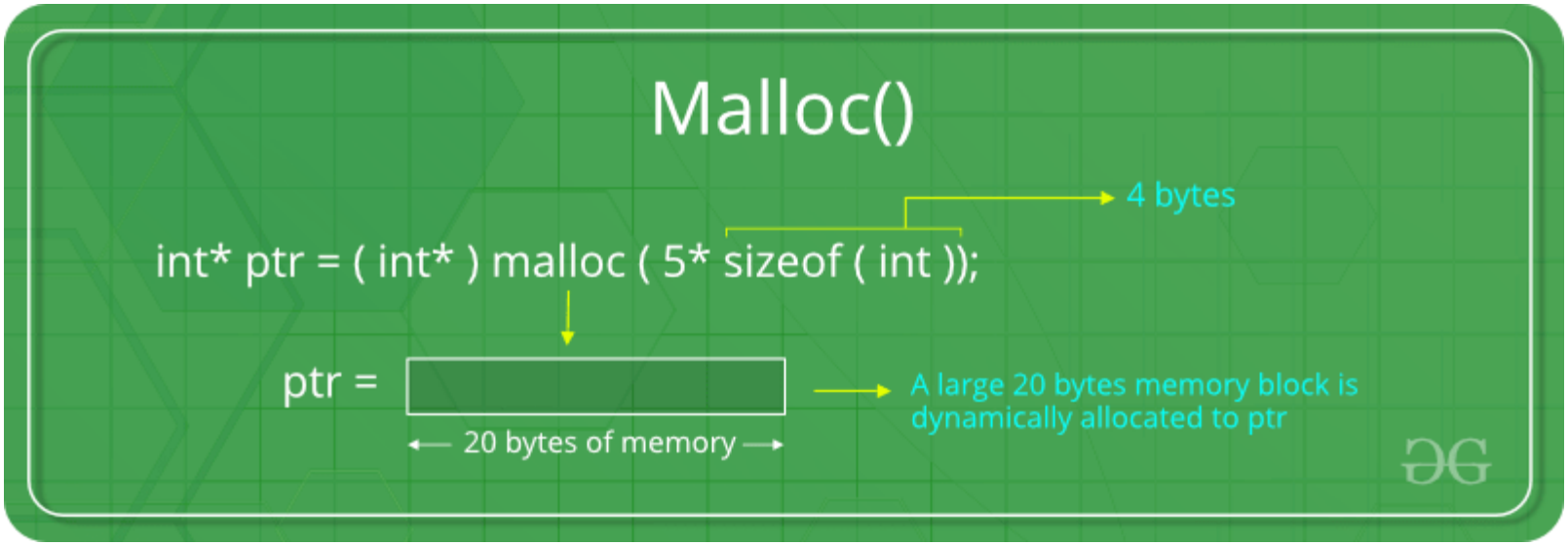
3.realloc()-reallocates the memory occupied by malloc() or calloc() functions.

4.free()-frees the dynamically allocated memory.

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

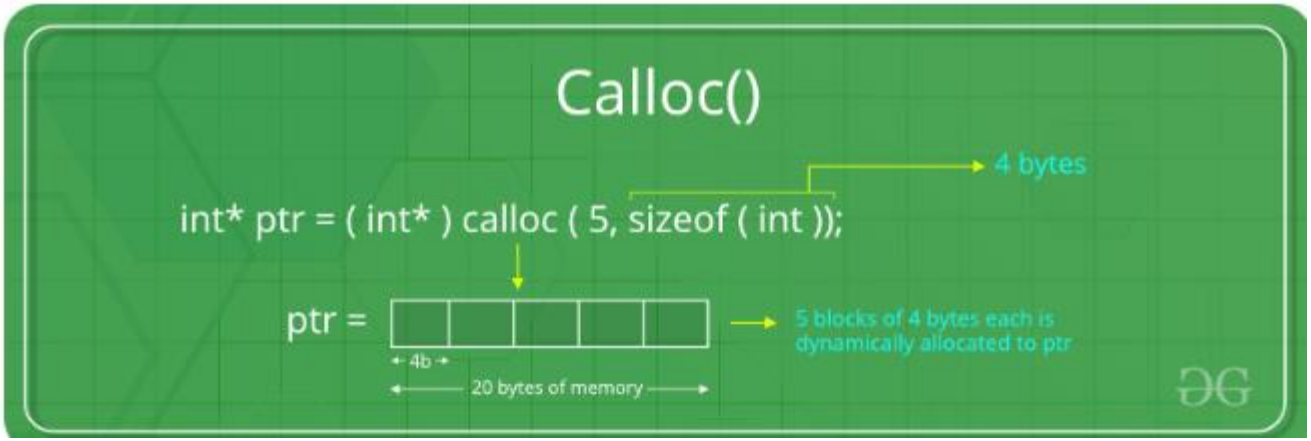
- The malloc() function allocates single block of requested memory.
- Working: It returns pointer to the allocated memory block on successful execution else returns NULL on failure.
- ptr=(cast-type*)malloc(byte-size)



```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int *ptr,i;
  ptr = (int*)malloc(4*sizeof(int));
 if(ptr== NULL)
{       printf("fail");    }
  else    {
 ptr[0]=25;
     ptr[1]=35;
       ptr[2]=45;
   ptr[3]=55;
    for(i=0;i<4;i++)
   {
  printf("%d\n",ptr[i]);
  }    }    return 0;}
```

calloc

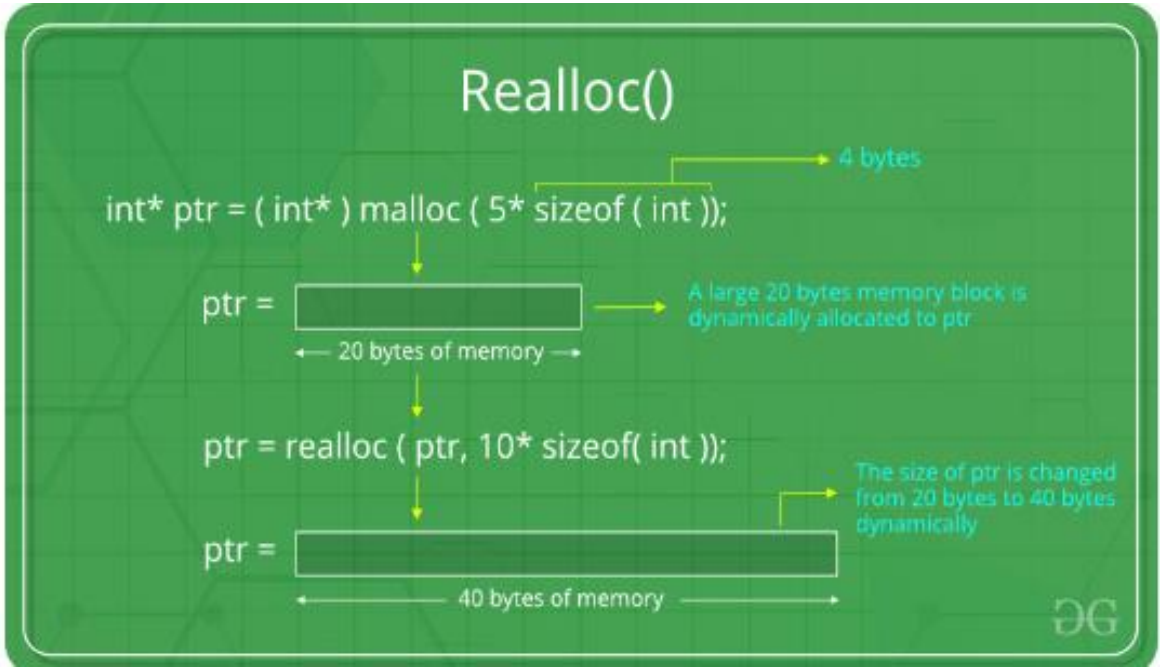
- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.
- ptr=(cast-type*)calloc(number, byte-size)
-



```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int *ptr,i;
  ptr = (int*)calloc(4,sizeof(int));
 if(ptr== NULL)
{       printf("fail");    }
  else    {
 ptr[0]=25;
      ptr[1]=35;
        ptr[2]=45;
    ptr[3]=55;
     for(i=0;i<4;i++)
    {
   printf("%d\n",ptr[i]);
  }    }    return 0;}
```

realloc()

- 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)



```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr,i;
    ptr = (int*)calloc(4,sizeof(int));
    if(ptr== NULL)
    {
        printf("fail");
    }
    else
    {
        ptr[0]=25;
        ptr[1]=35;
        ptr[2]=45;
        ptr[3]=55;

 ptr= realloc(ptr,8*sizeof(int));

        ptr[4]=75;
        ptr[5]=85;
        ptr[6]=95;
        ptr[7]=155;

        for(i=0;i<8;i++)
        {
            printf("%d\n",ptr[i]);
        }
    }
    return 0;
}
```

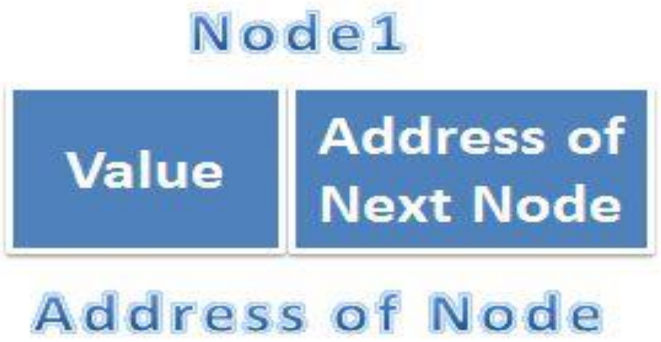
free()

- 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)
-

Introduction to Linked List

- It is a data Structure which consists if group of nodes that forms a sequence.

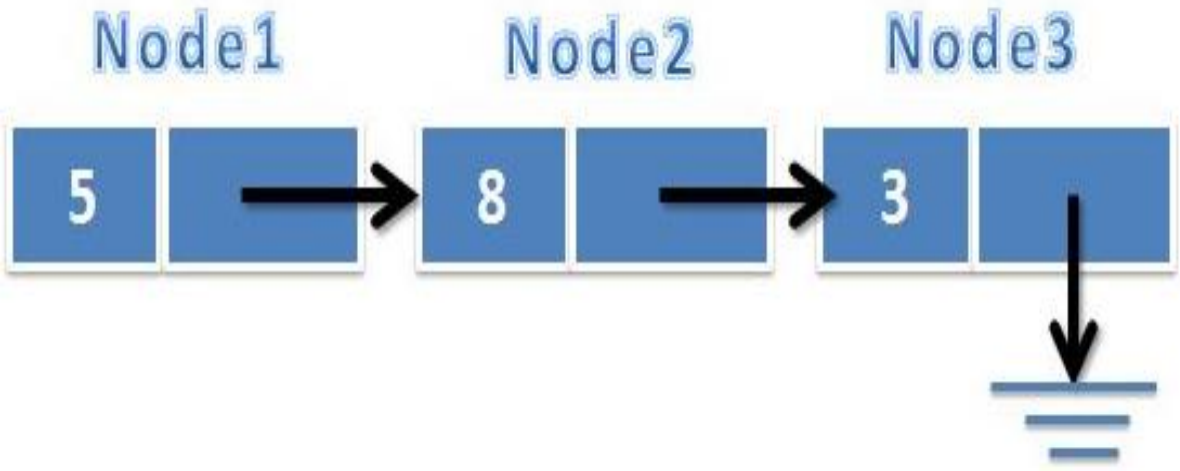
- It is very common data structure that is used to create tree, graph and other abstract data types.



- Linked list comprise of group or list of nodes in which each node have link to next node to form a chain

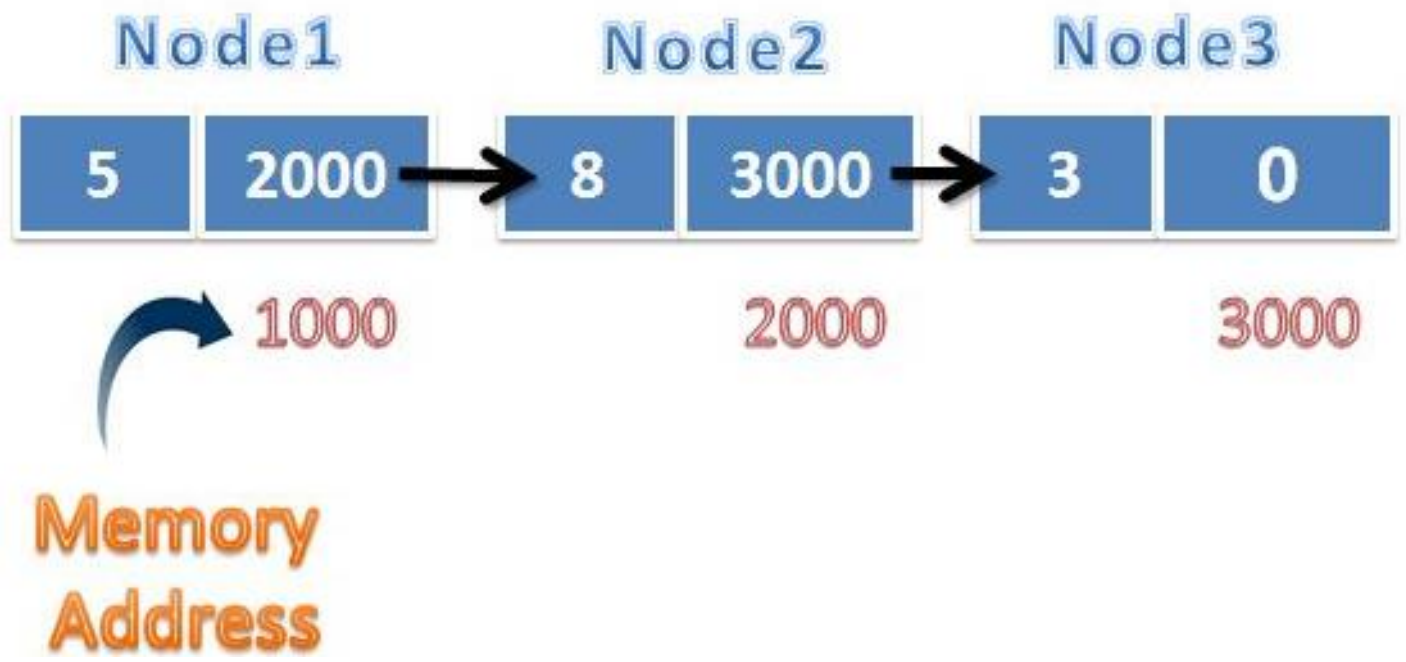
Linked List definition

- Linked List is **series of Nodes**
- Each node Consist of two Parts **Data Part & Pointer Part**
- Pointer Part stores the **address of the next node**



What is linked list Node ?

Node A has two part one data part which consists of the 5 as data and the second part which contain the address of the next node (**i.e it contain the address of the next node**)

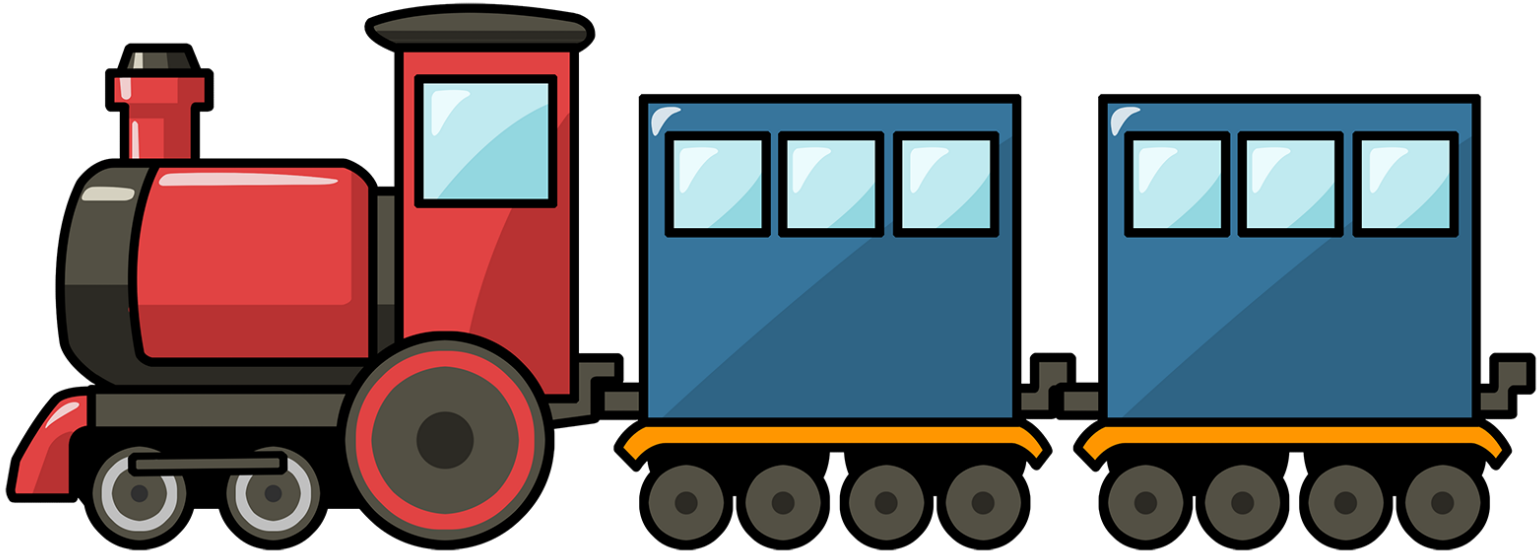


Linked list Blocks

No	Element	Explanation
1	Node	Linked list is collection of number of nodes
2	Address Field in Node	Address field in node is used to keep address of next node
3	Data Field in Node	Data field in node is used to hold data inside linked list.

Linked list Blocks

We can represent linked list in real life using train in which all the buggies are nodes and two coaches are connected using the connectors.

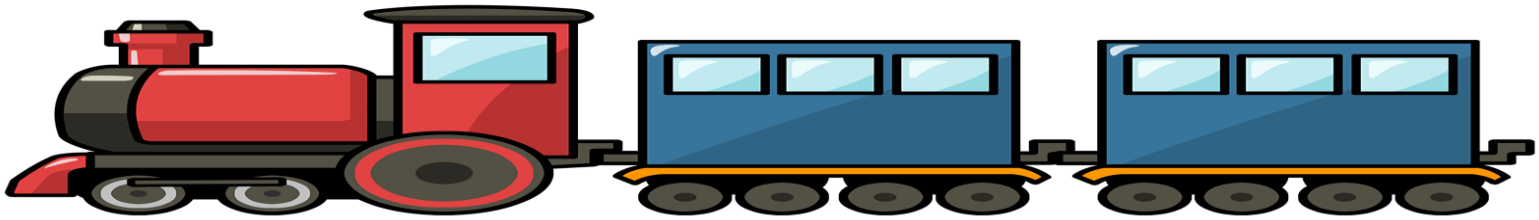


Linked list Blocks

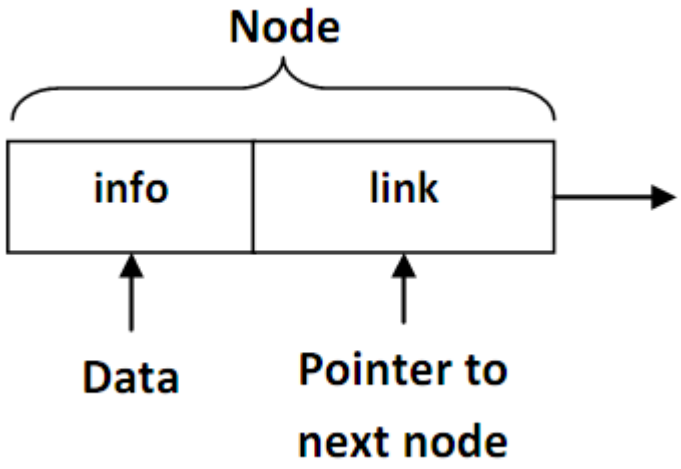
- In case of railway we have peoples seating arrangement inside the coaches is called as data part of lined list while connection between two buggies is address filed of linked list.

- Like linked list, trains also have last coach which is not further connected to any of the buggie.

- Engine can be called as first node of linked list



linked list




```
// C Structure to represent a node
struct node
{
int info;
struct node *link;
};
```

Array	Linked List
An array stores its element in consecutive memory locations	A linked list does not store its nodes in consecutive memory locations.
Array allows random access of data	linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner starting from the first node.
Insertion and deletion of data is difficult.	Insertion and deletion of data is easy.
Memory allocation is static.	Memory allocation is dynamic.
Size specified during declaration	No need to specify size. Size can be grow and shrink during execution.
Binary search and linear search is used.	Linear search is used.
Less memory required.	More memory required.
Memory utilization is ineffective	Memory utilization is efficient.

Types of Linked List

1. singly linked list
2. singly circular linked list
3. Doubly linked list
4. Doubly circular linked list

Topics-PartII

4.7 Basic operations on singly linked list :

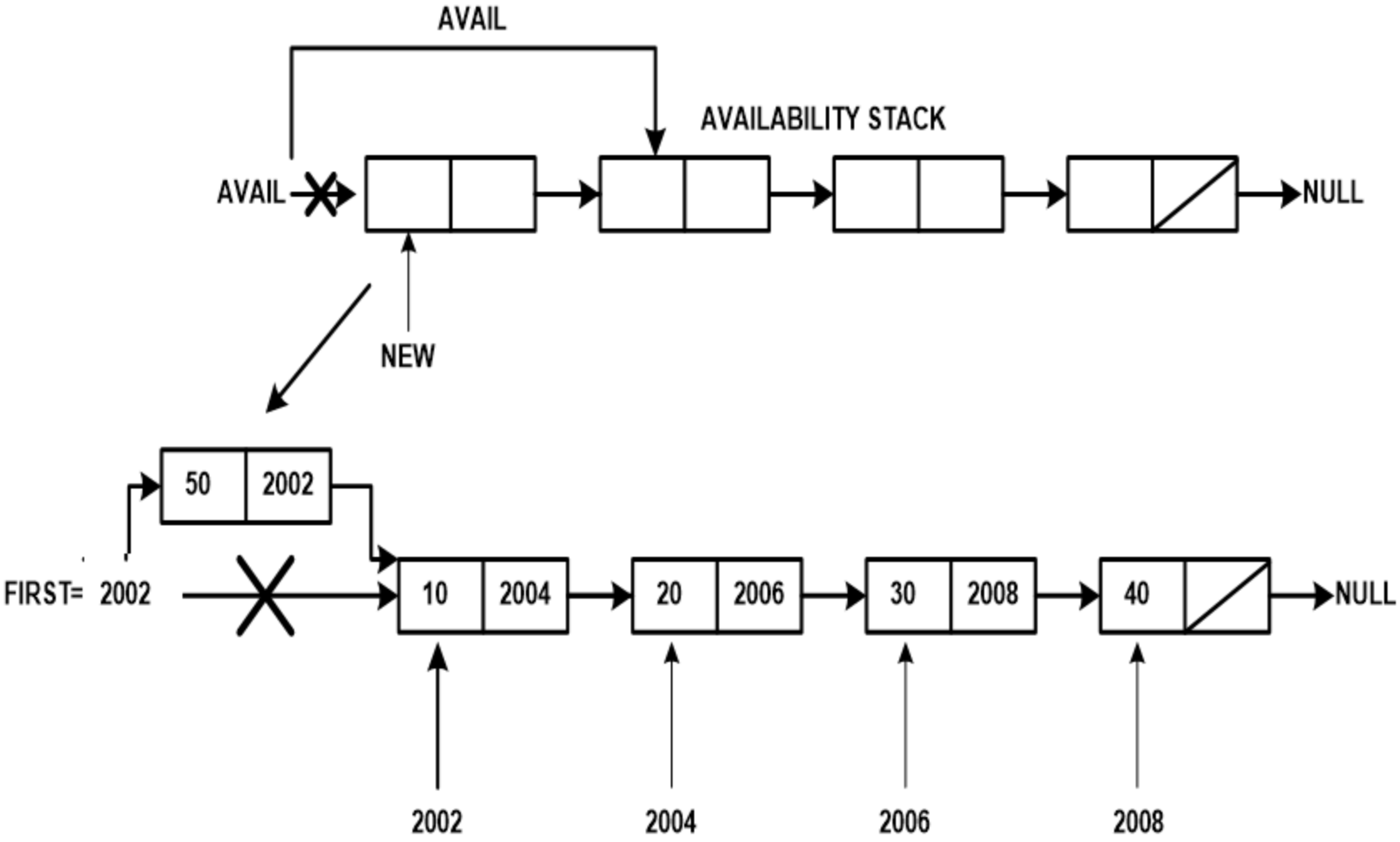
- Insertion of a new node in the beginning of the list
- at the end of the list
- after a given node
- before a given node
- In sorted linked list
- Deleting the first and last node from a linked list,
- Searching a node in Linked List
- Count the number of nodes in linked list

Operations on linked list

- Insert
  - Insert at first position
  - Insert at last position
  - Insert into ordered list
  - Insert before specific node
  - Insert after specific node
- Delete
- Traverse list (Print list)

Algorithms for Singly linked list

➢Algorithm to insert new node at beginning of the linked list



➢Algorithm to insert new node at beginning of the linked list

INSERTBEG (VAL,FIRST)

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

➢FIRST is a pointer which contains address of first node in the list.

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)

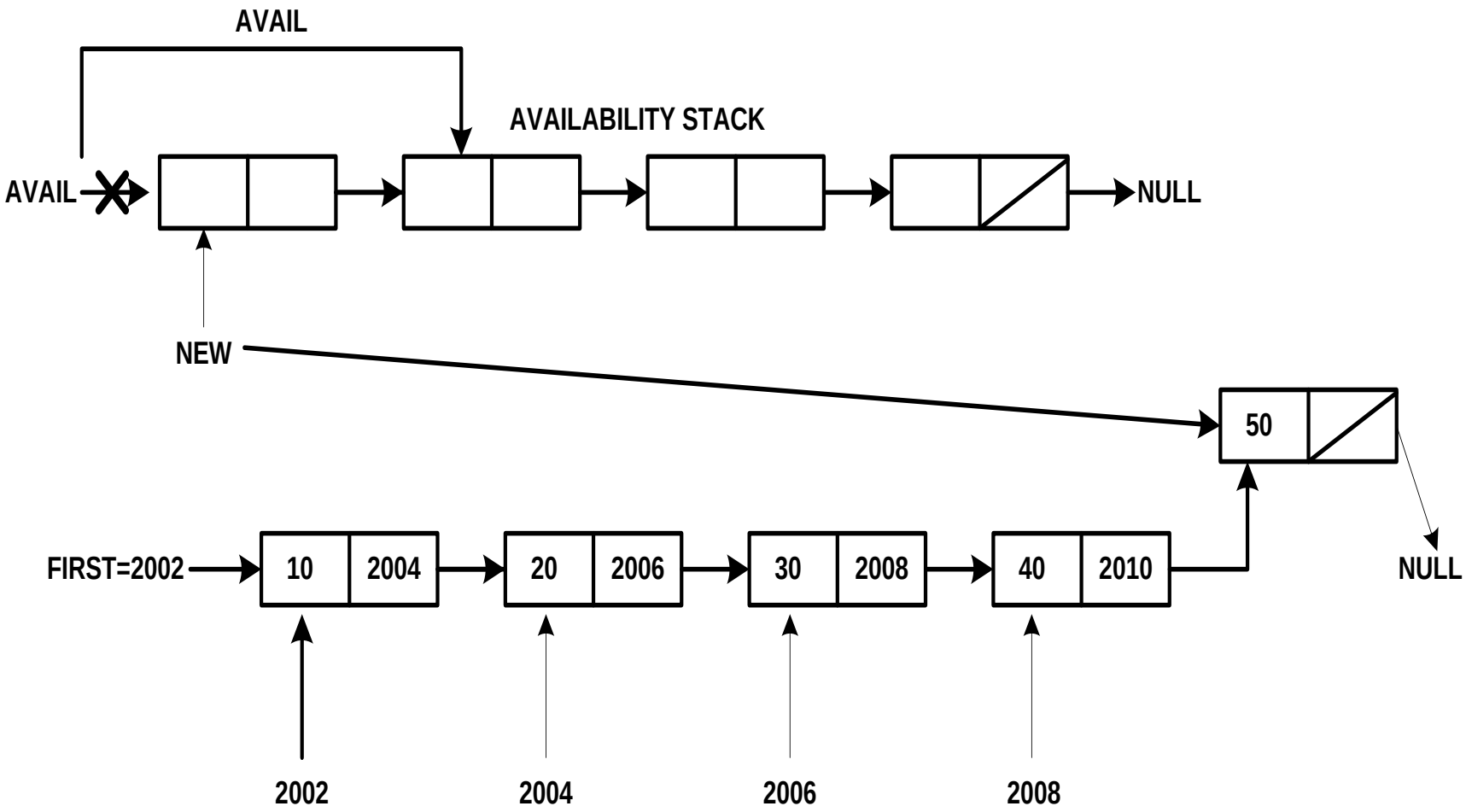
4[Initialize node to the linked list]
INFO (NEW) ←VAL

LINK (NEW) <- First

5 [Assign the address of the Temporary node to the First Node]
FIRST←NEW

6[Finished]
Return (FIRST)

➢Algorithm to insert new node at end of the linked list



➢Algorithm to insert new node at end of the linked list

INSERTEND (VAL,FIRST)

- This function inserts a new element VAL at the end of the linked list.
- FIRST is a pointer which contains address of first node in the list.

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 field of new node]
INFO (NEW) ←VAL

LINK (NEW) ←NULL

5[If list is empty?]
If FIRST = NULL then

FIRST←NEW

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]
Return (FIRST)

➢Algorithm to insert new node into ordered Linked list

INSORD (FIRST, X)

➢This function inserts a new element X into the ordered linked list.

➢FIRST is a pointer which contains address of first node in the list.

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 field of new node]
INFO (NEW) ←X

5[If list is empty?]
If FIRST = NULL then

LINK (NEW) ←NULL

FIRST←NEW, Return FIRST

6[Does the new node precede all others in the list]
If INFO(NEW) <= INFO(FIRST) then

LINK (NEW) ←FIRST

FIRST←NEW, Return FIRST

7[Search for predecessor of new node]
SAVE←FIRST

8. Repeat while LINK (SAVE) ≠ NULL and INFO(LINK(SAVE)) <= INFO(NEW)

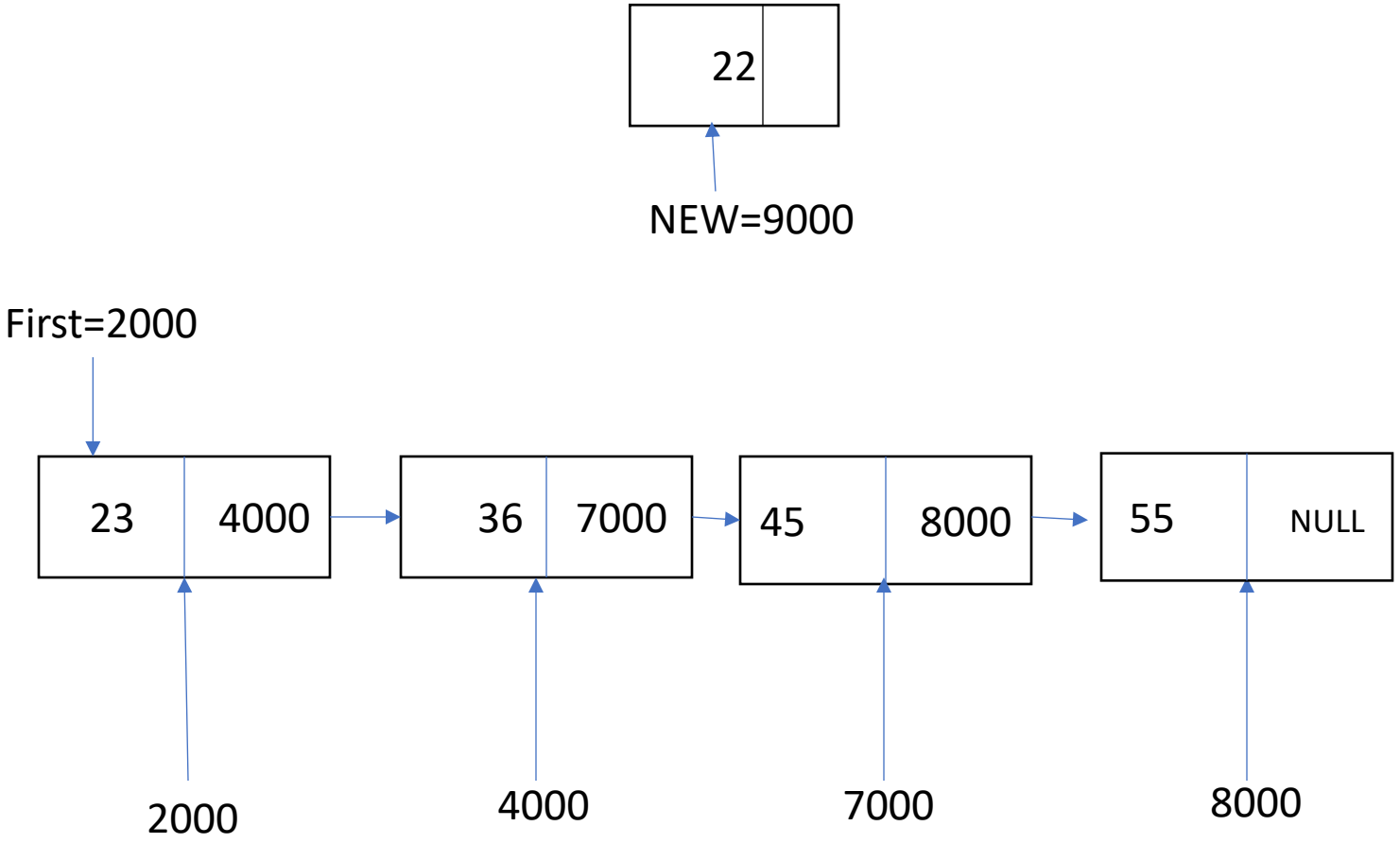
SAVE←LINK (SAVE)

9[Set link fields of new node and its predecessor]
LINK (NEW) ←LINK(SAVE)

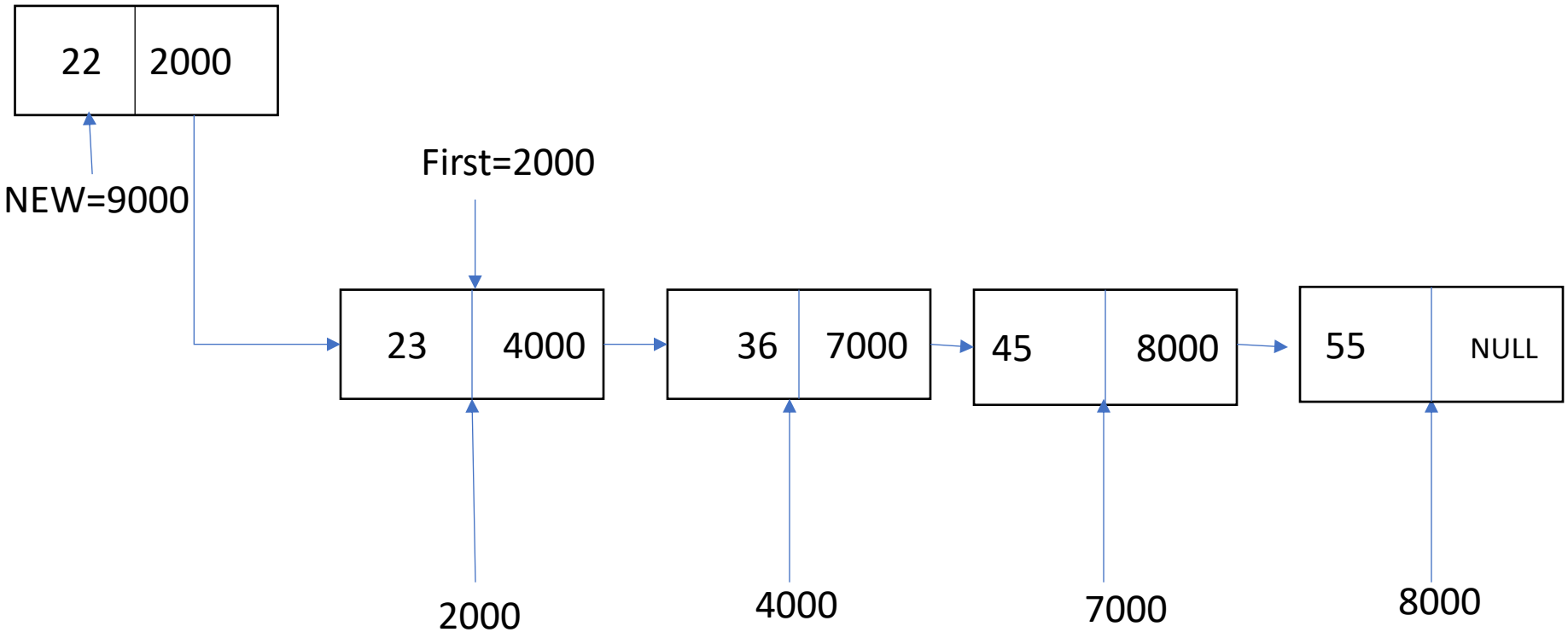
LINK(SAVE)←NEW

10.[Finished]
Return (FIRST)

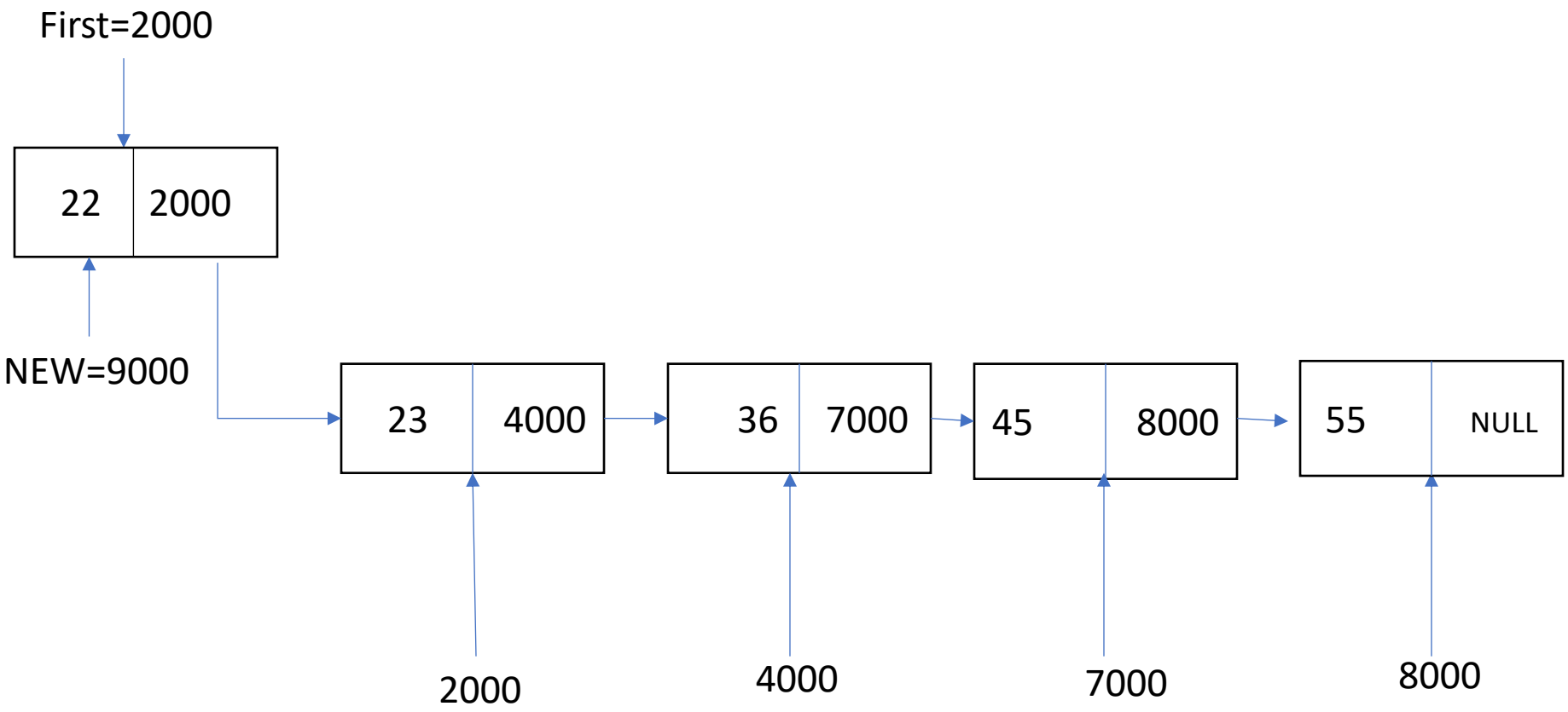
Case 2: If New node is smallest ,then it should be added at the beginning of linked list.



Case 2: If New node is smallest ,then it should be added at the beginning of linked list.



Case 2: If New node is smallest ,then it should be added at the beginning of linked list.



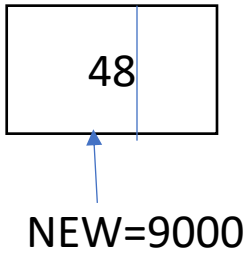
Case 3: If New node to be inserted in-between ,then find the node greater than new node

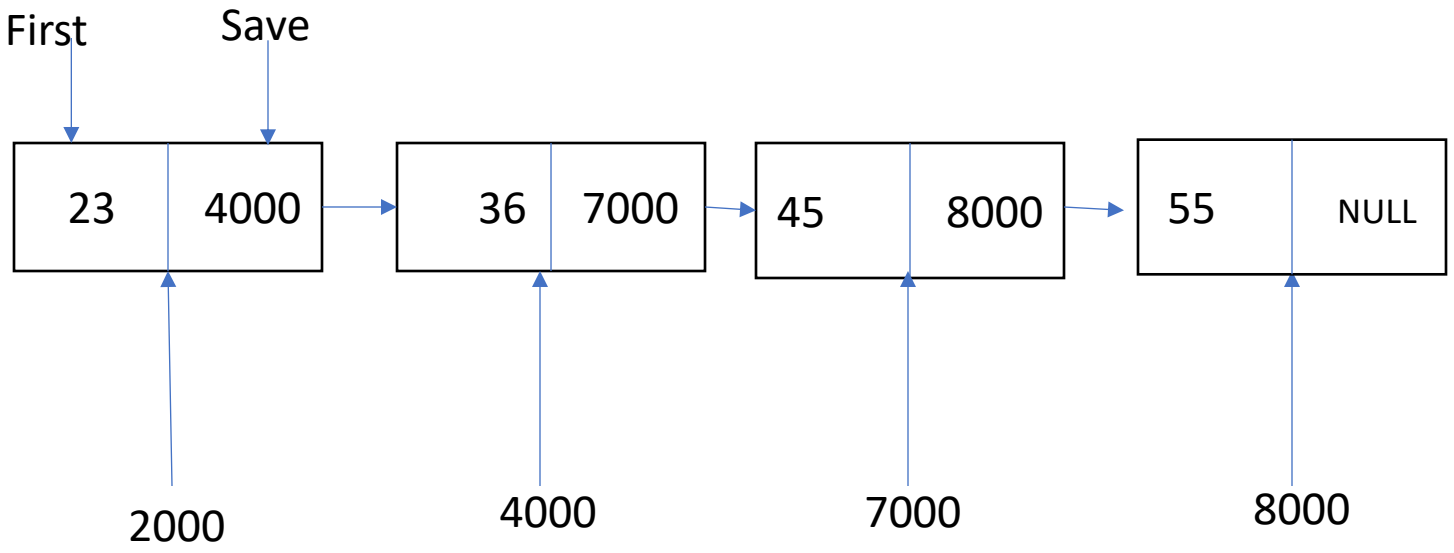
Check if INFO(LINK(SAVE)) <= INFO(NEW)

INFO(4000) <= INFO(NEW)

36<= 48

Then Increment Save pointer





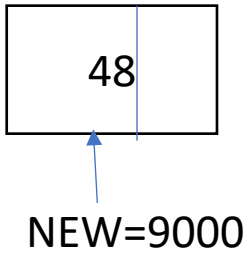
Case 3: If New node to be inserted in-between ,then find the node greater than new node

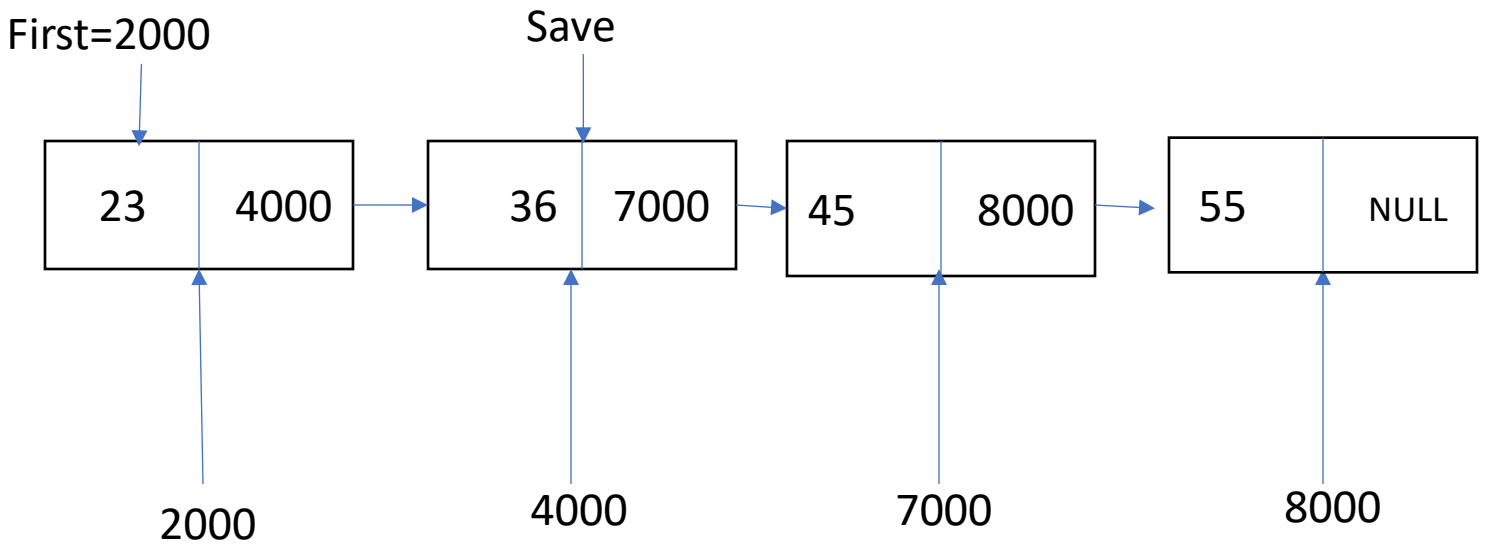
Check if INFO(LINK(SAVE)) <= INFO(NEW)

INFO(7000) <= INFO(NEW)

45<= 48

Then Increment Save pointer





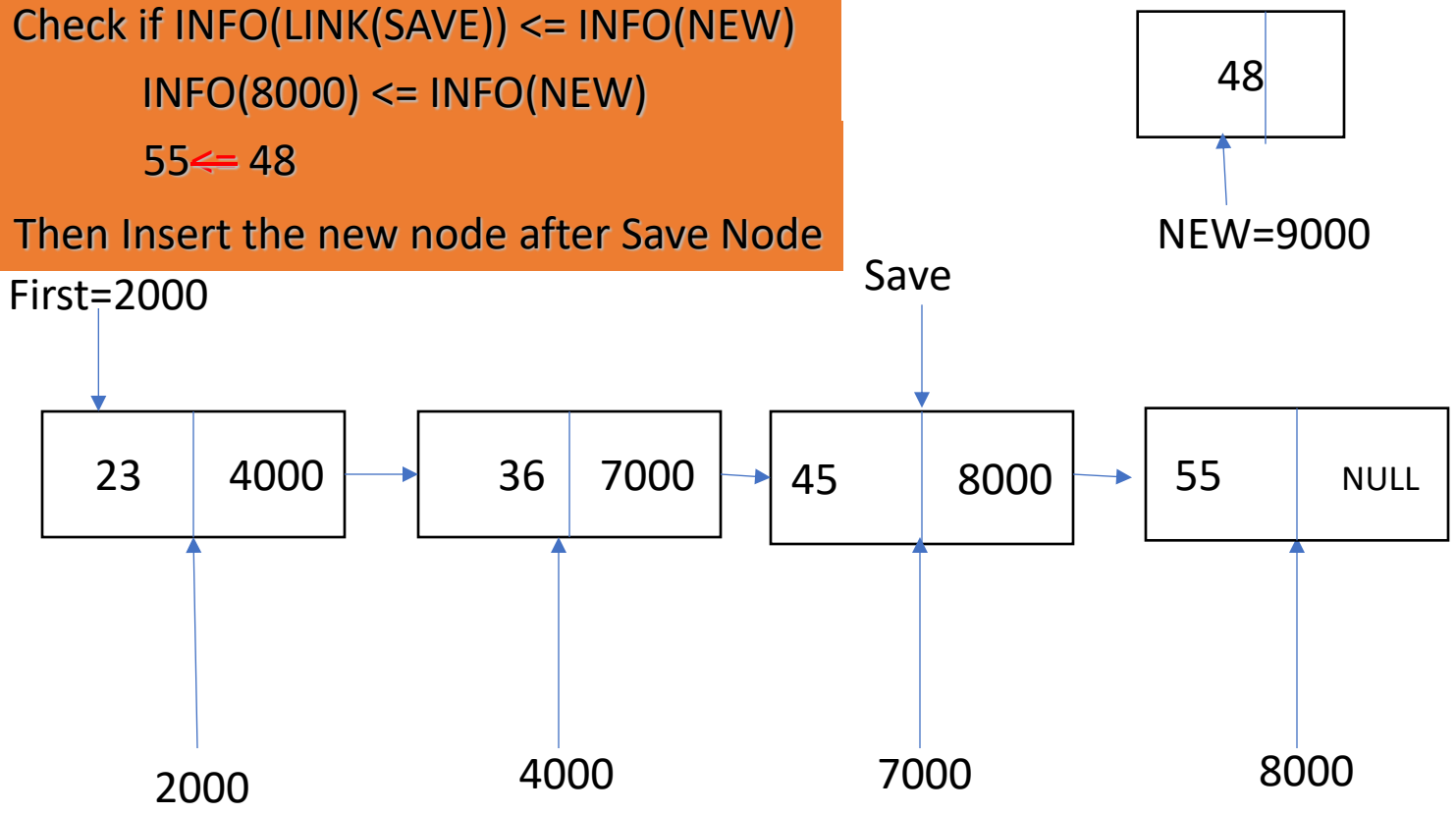
Case 3: If New node to be inserted in-between ,then find the node greater than new node

Check if INFO(LINK(SAVE)) <= INFO(NEW)

INFO(8000) <= INFO(NEW)

55<= 48

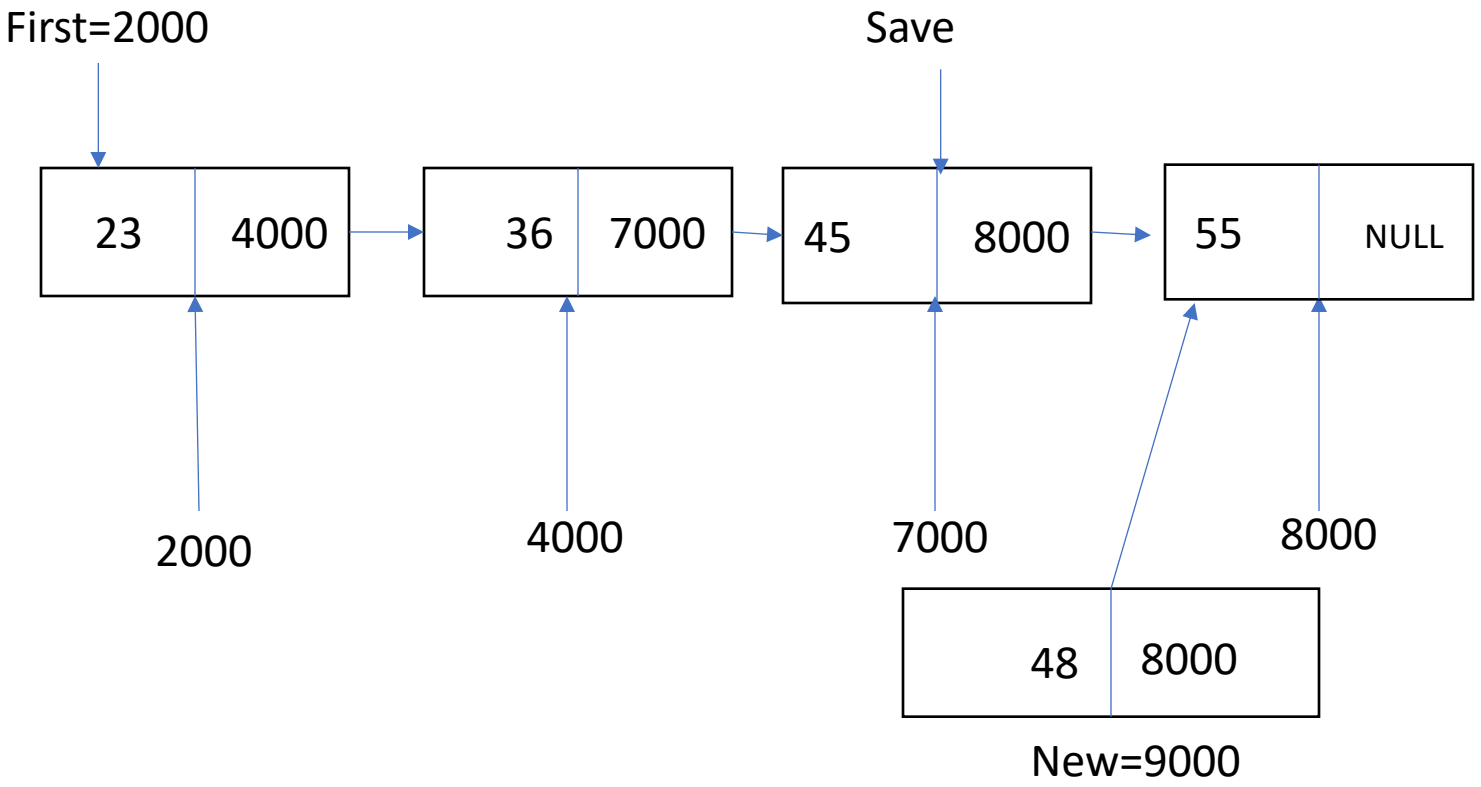
Then Insert the new node after Save Node



Case 3: If New node to be inserted in-between ,then find the node greater than new node

Then Insert the new node after Save Node

LINK (NEW) ←LINK(SAVE)

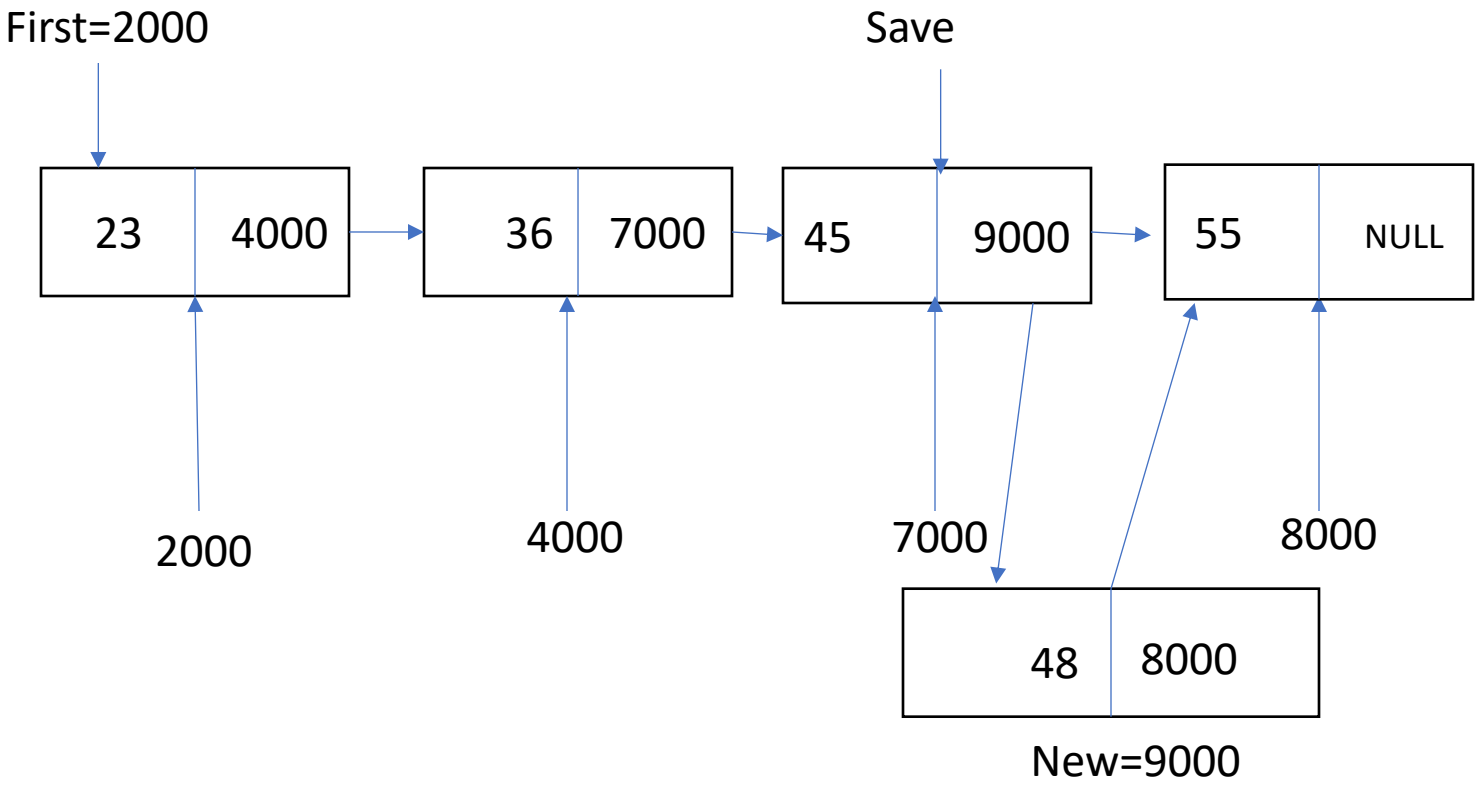


Case 3: If New node to be inserted in-between ,then find the node greater than new node

Then Insert the new node after Save Node

LINK (NEW) ←LINK(SAVE)

LINK(SAVE)←NEW



➢Algorithm to insert new node before specific node

INSPOS (VAL, FIRST, X)

➢This function inserts a new element X into the linked list before the node value VAL.

➢FIRST is a pointer which contains address of first node in the list.

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 field of new node]
INFO (NEW) ←X

5[If list is empty?]
If FIRST = NULL then

LINK (NEW) ←NULL

FIRST←NEW, Return FIRST

6[If list contain only one node?]
If INFO(FIRST) = VAL then

LINK (NEW) ←FIRST

FIRST←NEW, Return FIRST

7[Search the list until desired address found]
SAVE←FIRST

8. Repeat while LINK (SAVE) ≠ NULL and INFO(SAVE) ≠ VAL

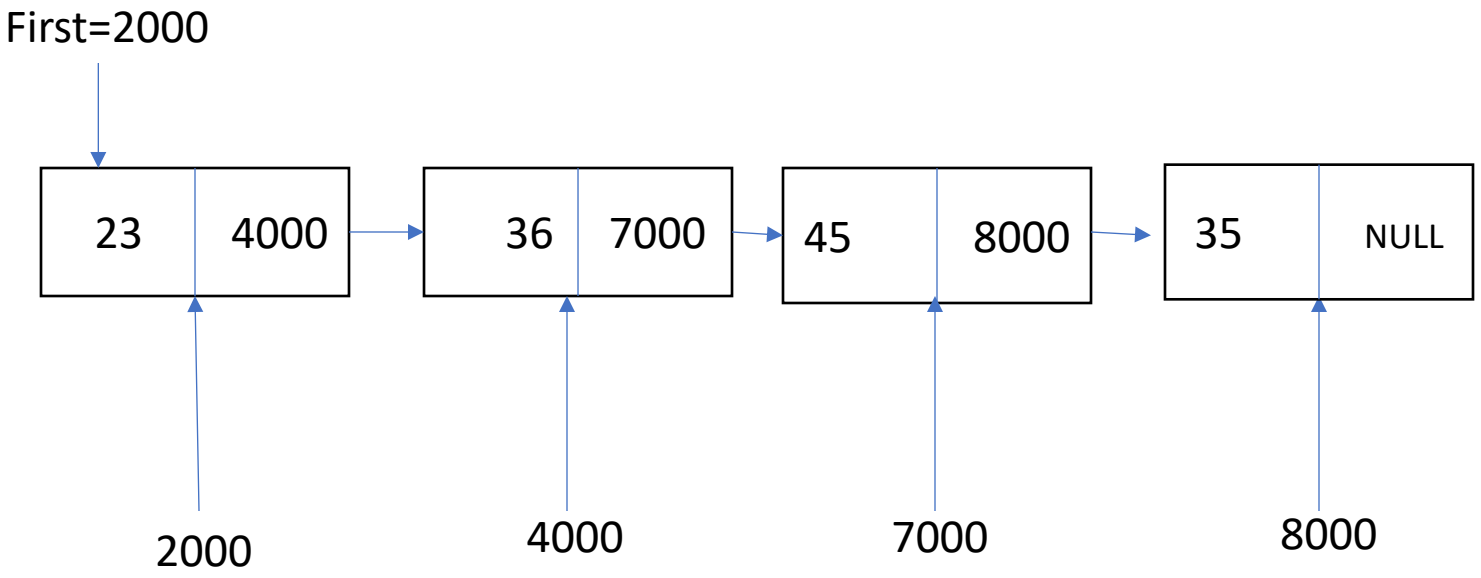
PRED←SAVE
SAVE←LINK (SAVE)

9[Insert node]

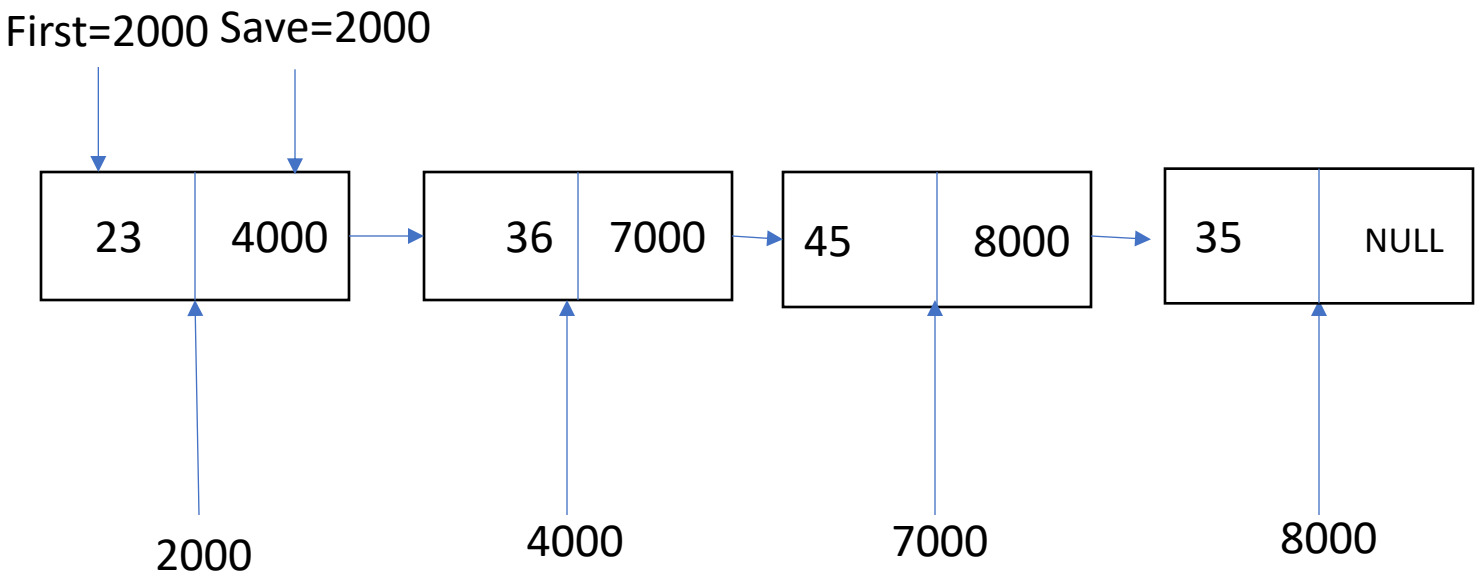
LINK (NEW) ←SAVE

LINK(PRED)←NEW

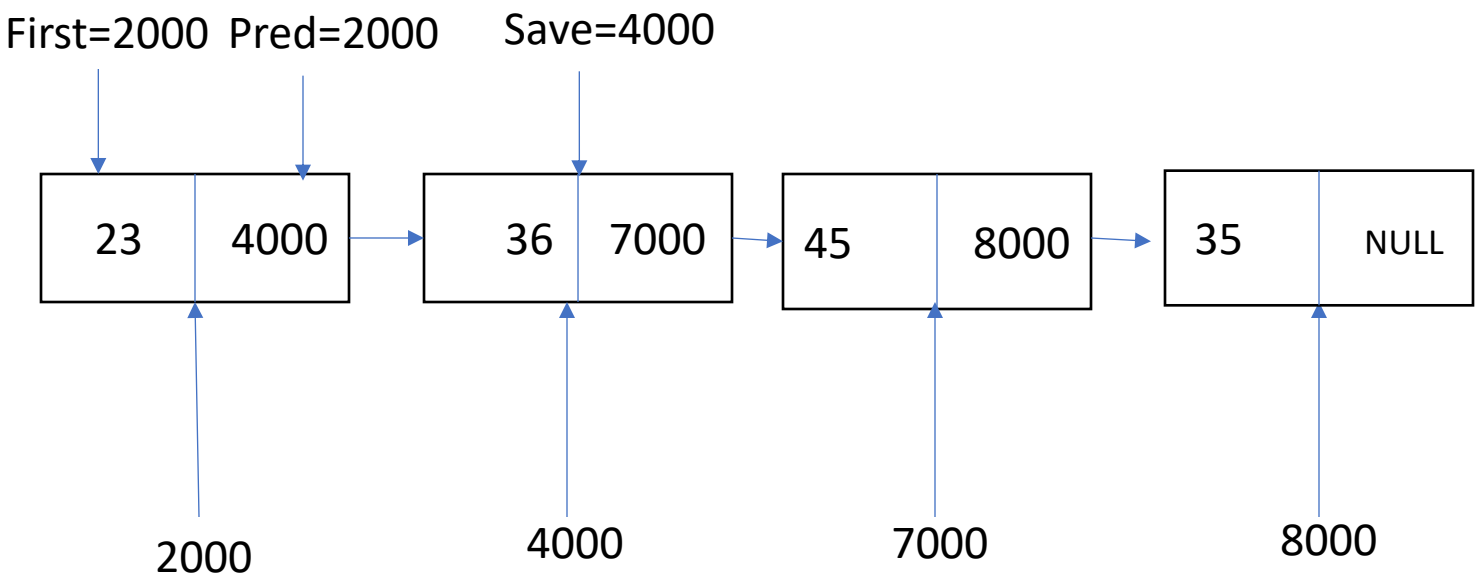
10.[Finished]
Return (FIRST)



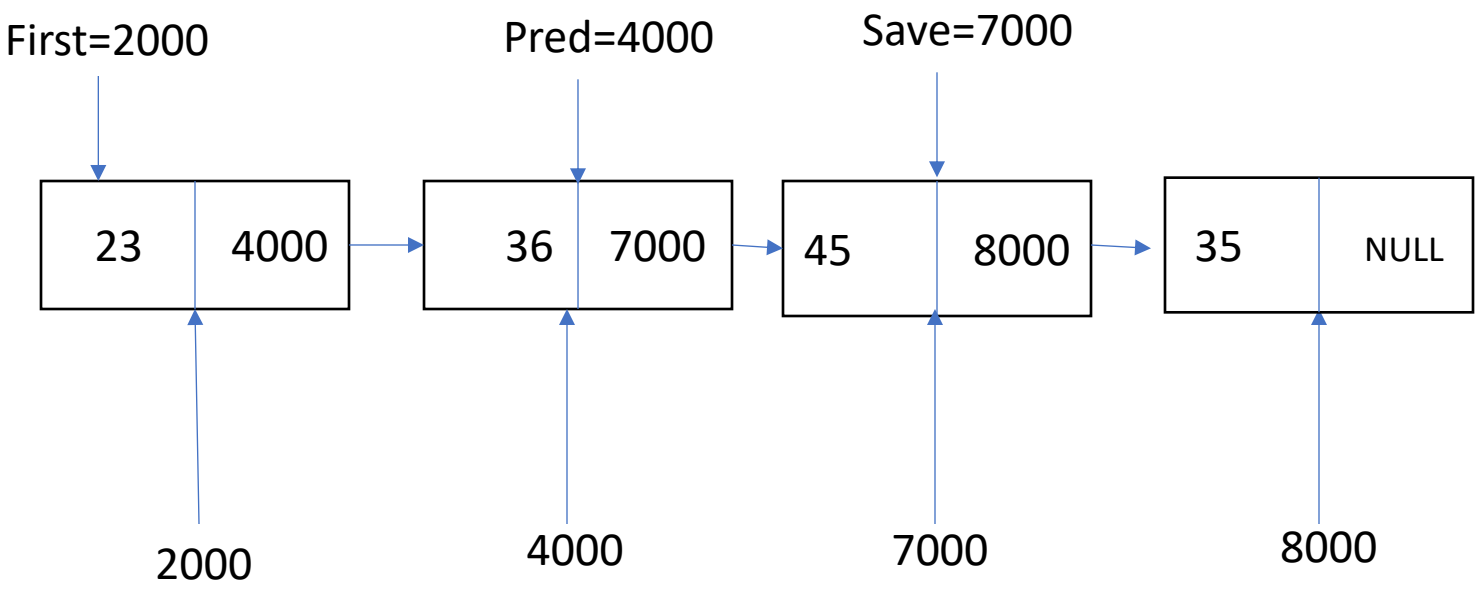
Insert new node with X=26 Before node having val 35.



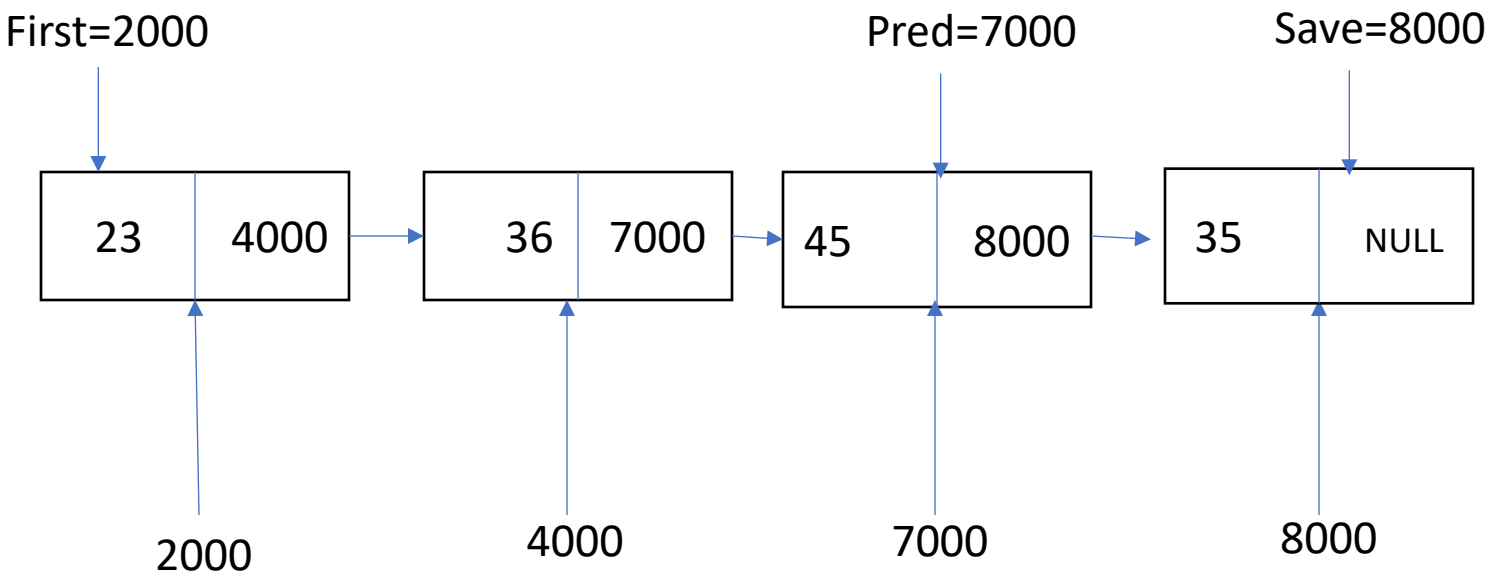
Insert new node with X=26 Before node having val 35.



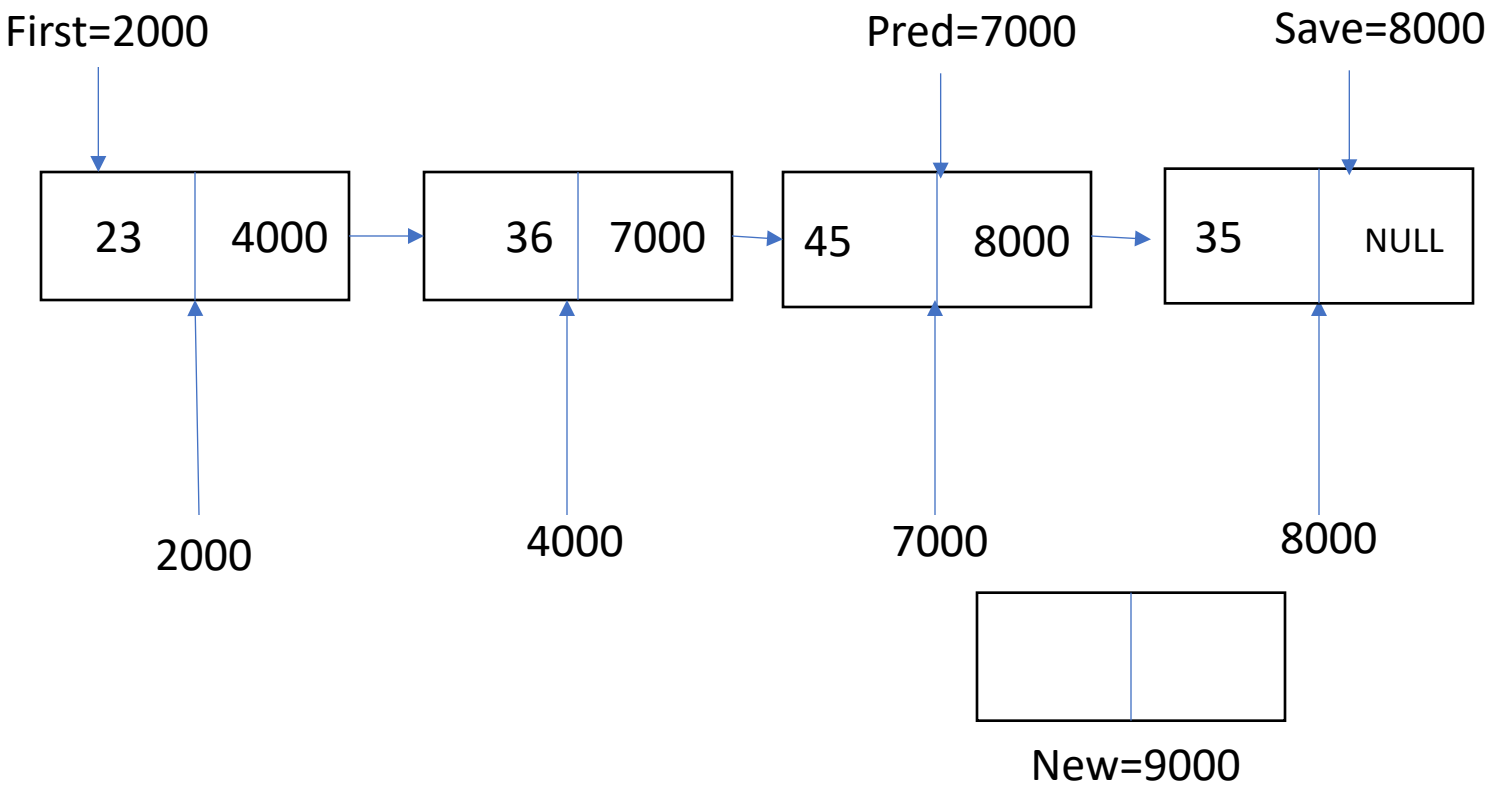
Insert new node with X=26 Before node having val 35.



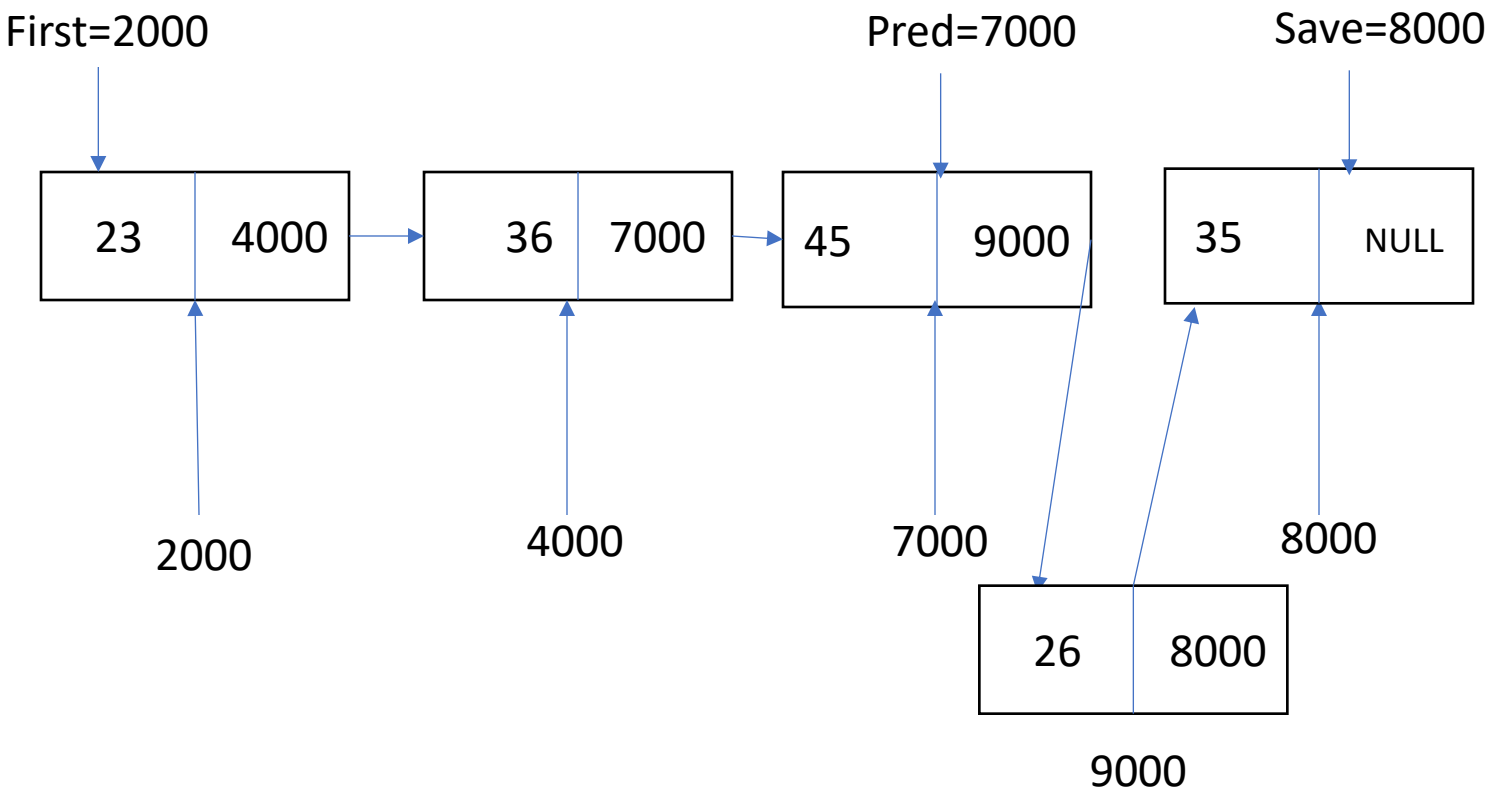
Insert new node with X=26 Before node having val 35.



Insert new node with X=26 Before node having val 35.



Insert new node with X=26 Before node having val 35.



Insert new node with X=26 Before node having val 35.

➢Algorithm to insert new node after specific node

INSPOS (VAL, FIRST, X)

➢This function inserts a new element X into the linked list after the node value VAL.

➢FIRST is a pointer which contains address of first node in the list.

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 field of new node]
INFO (NEW) ←X

5[If list is empty?]
If FIRST = NULL then

LINK (NEW) ←NULL

FIRST←NEW, Return FIRST

6.[Search the list until desired address found]
SAVE←FIRST

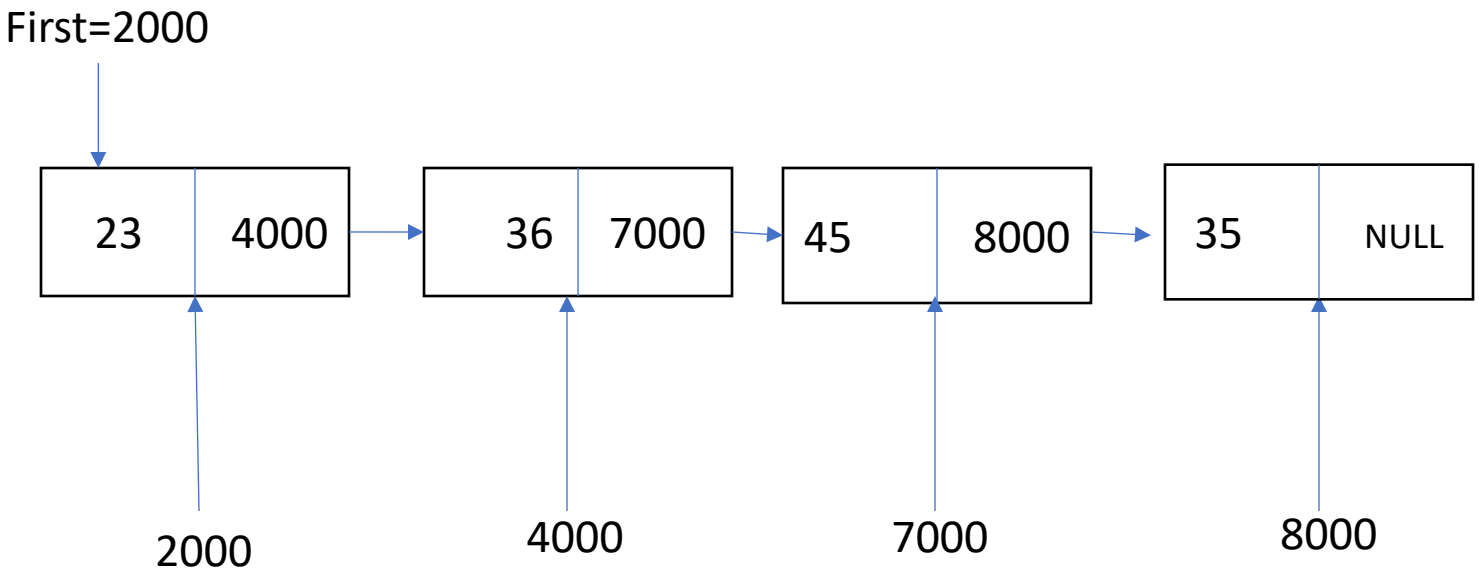
7. Repeat while LINK (SAVE) ≠ NULL and INFO(SAVE) ≠ VAL

SAVE←LINK (SAVE)

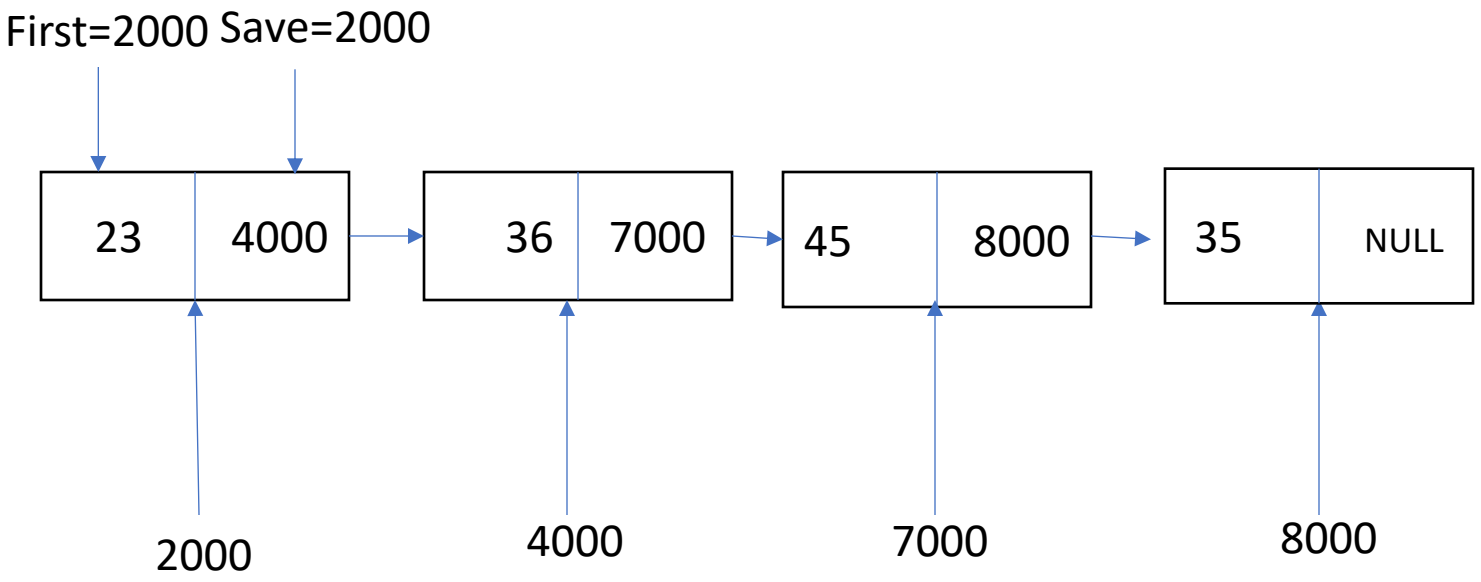
8[Insert node]
LINK (NEW) ←LINK(SAVE)

LINK(SAVE)←NEW

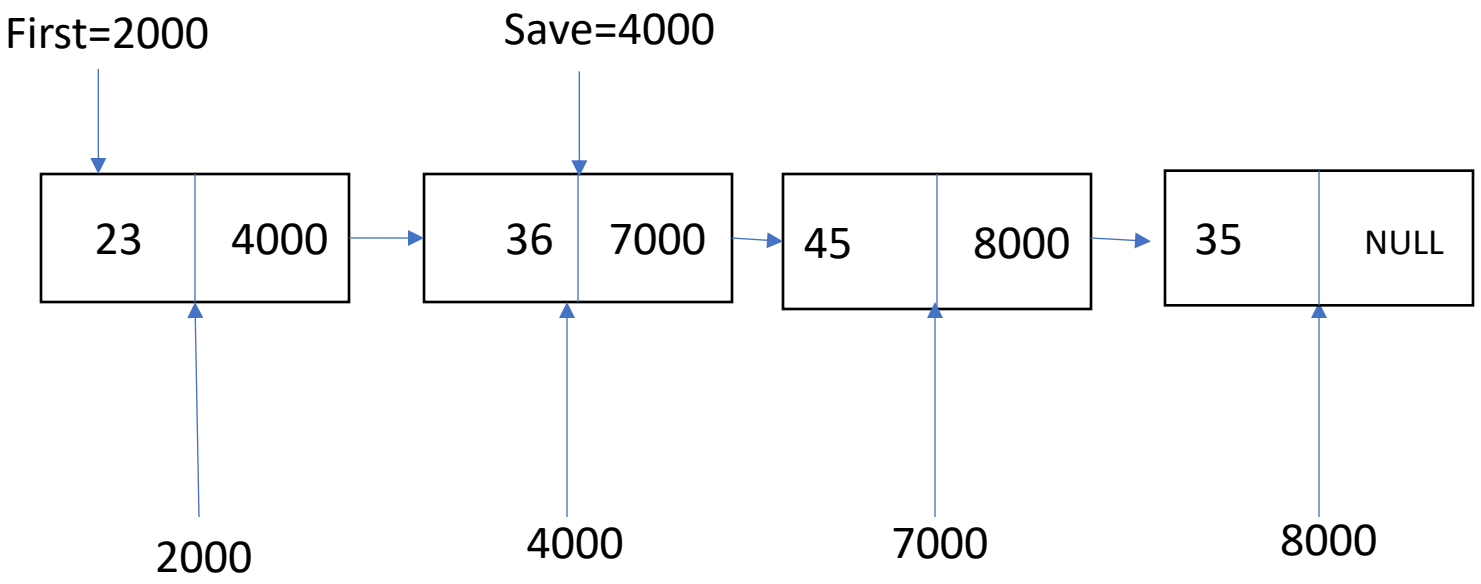
10.[Finished]
Return (FIRST)



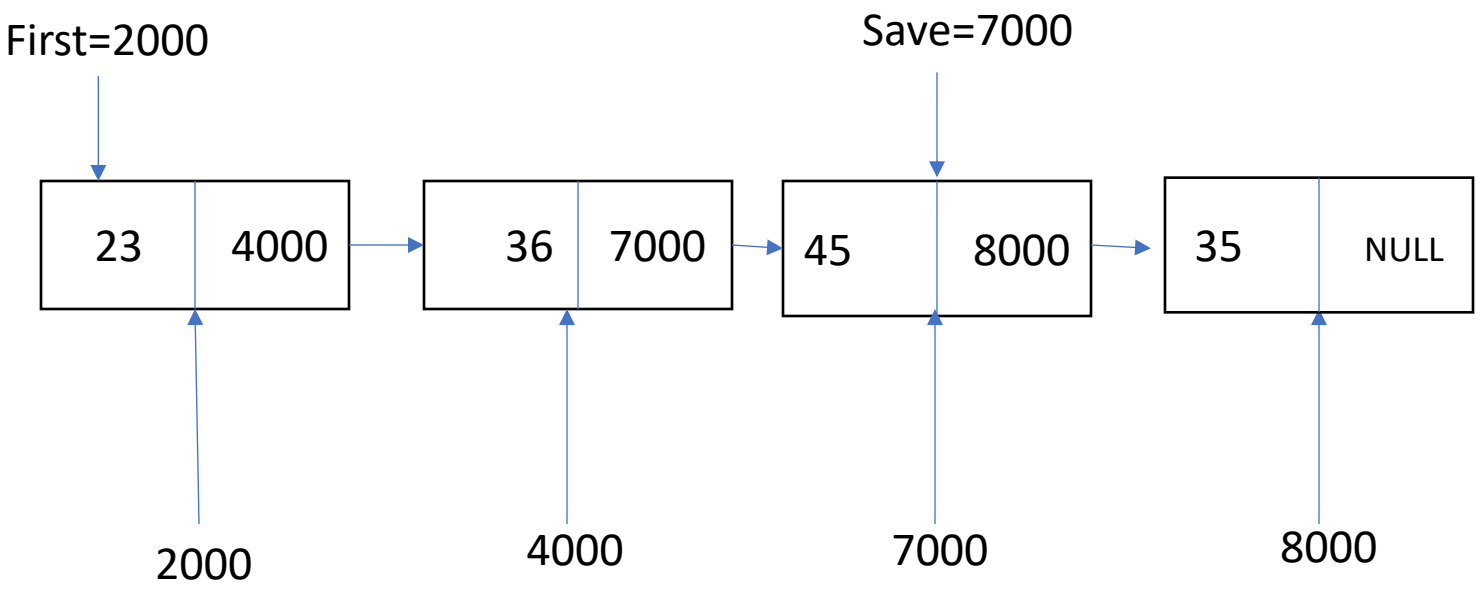
Insert new node with X=26 after node having val 45.



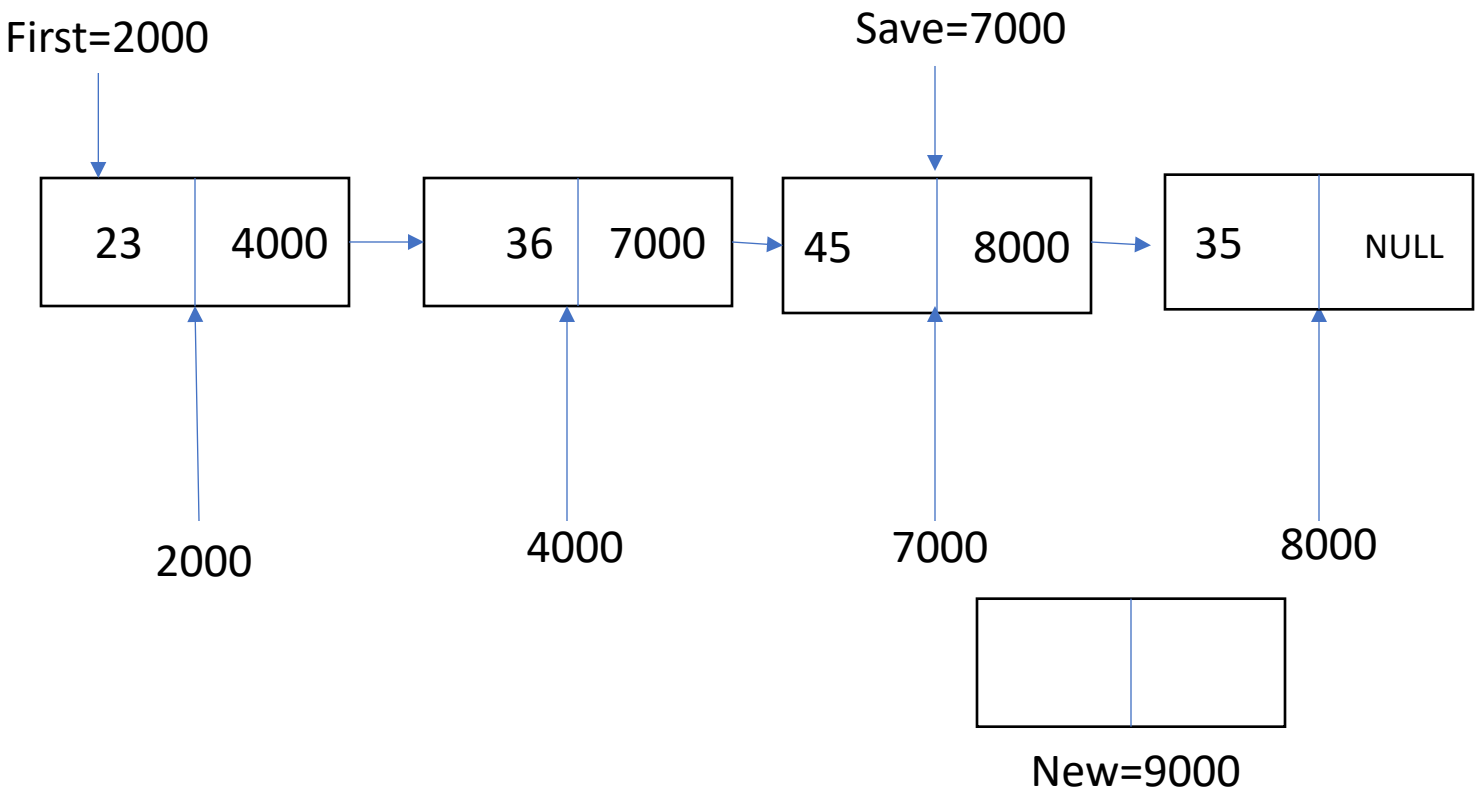
Insert new node with X=26 after node having val 45.



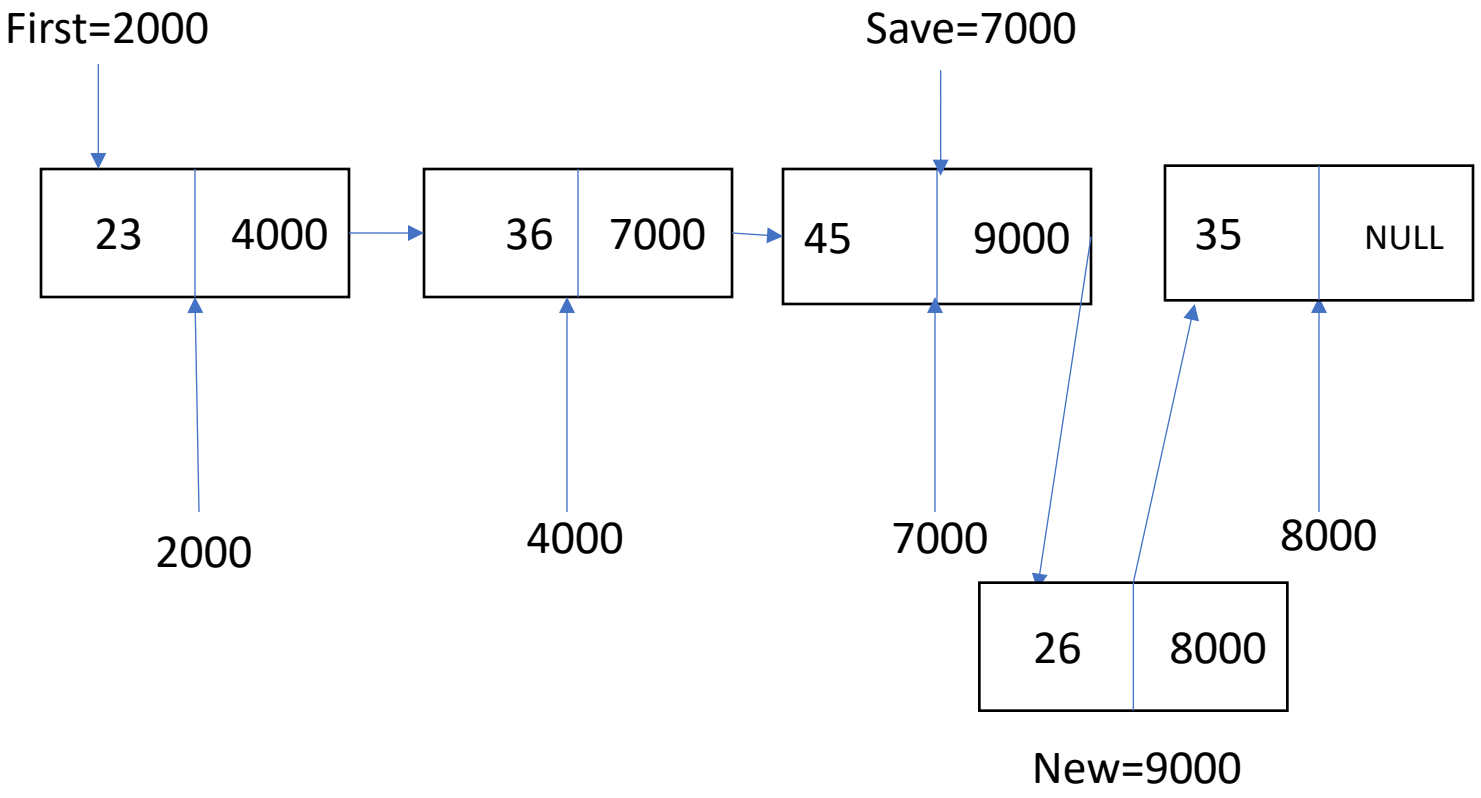
Insert new node with X=26 after node having val 45.



Insert new node with X=26 after node having val 45.



Insert new node with X=26 Before node having val 35.



Insert new node with X=26 Before node having val 35.

Q/A

- **What would be the asymptotic time complexity to add a node at the end of singly linked list, if the pointer is initially pointing to the head of the list?**

  a) O(1)
  b) O(n)
  c) θ (n)
  d) θ (1)

- **. What would be the asymptotic time complexity to find an element in the linked list?**

  a) O(1)
  b) O(n)
  c) $O(n^2)$
  d) None

Summary

- We have learnt how to insert new node before or after specific node.

Session Outcome

At the end of the Session, you will be able to

1. Apply algorithm of deletion of first node.
2. Apply algorithm of deletion of last node.

➢Algorithm to delete first node of linked list

DELFIRST(FIRST)

➢This function deletes a first node from the list.

➢FIRST is a pointer which contains address of first node in the list.

1[Check for empty list]

If FIRST = NULL then

Write “List is empty”

Return FIRST

2. Y←INFO (FIRST)

3. If LINK (FIRST) = NULL then
   Return NULL

4. TEMP ← FIRST

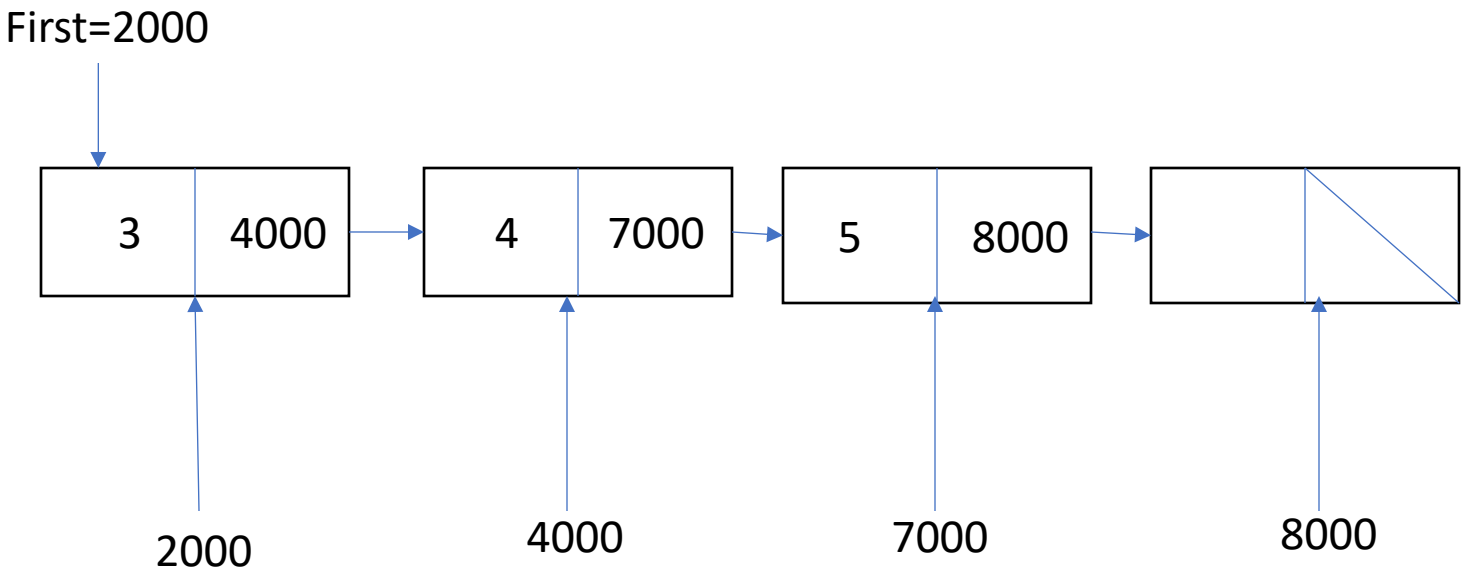
5. FIRST←LINK (FIRST)

6.[Return Node to Availability Area]

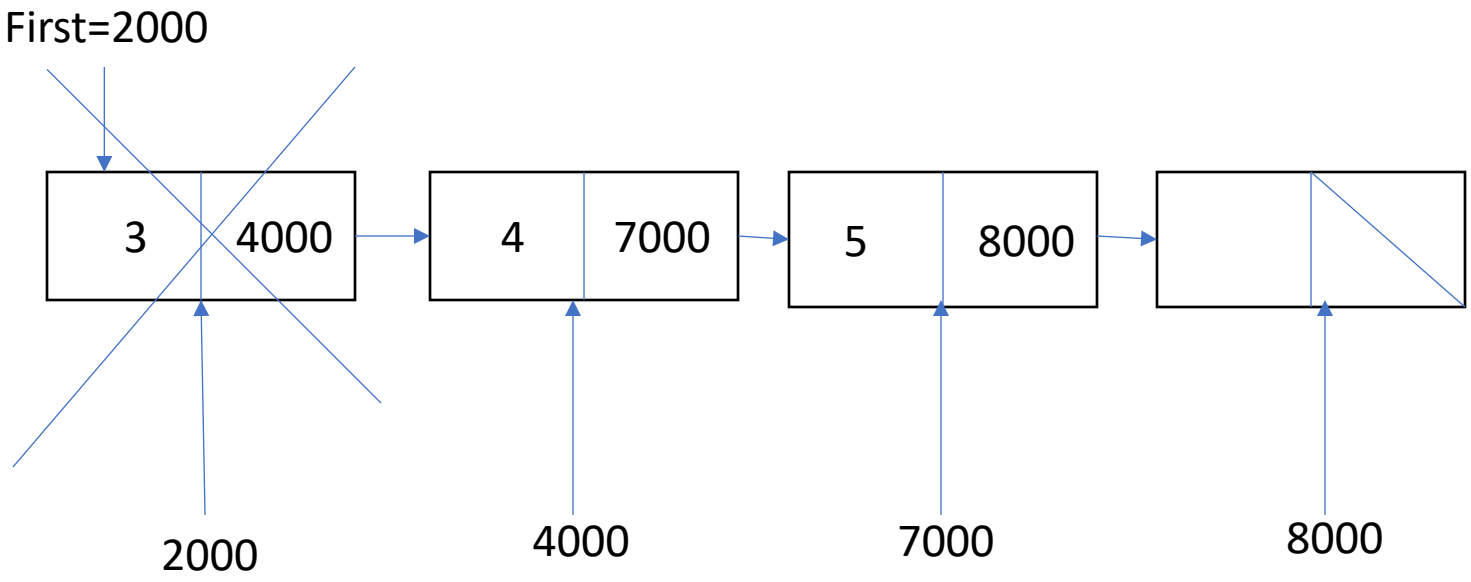
LINK(TEMP) ← AVAIL

AVAIL ← TEMP

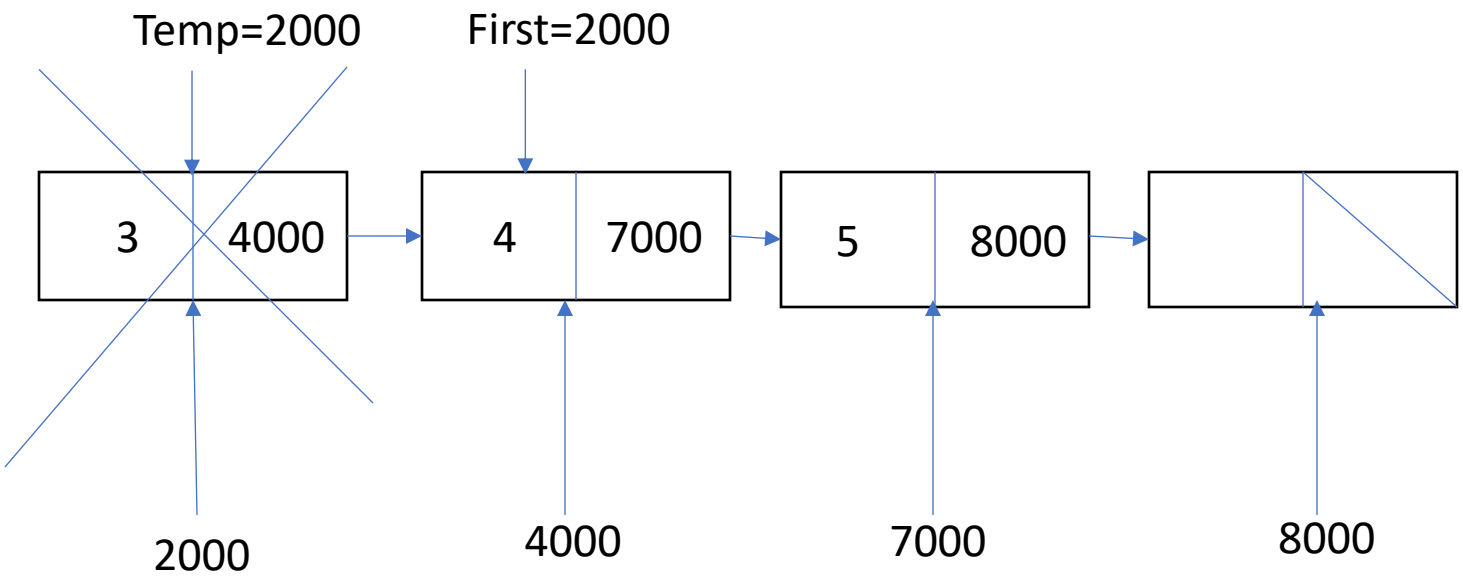
7. [Finished]
   Return (FIRST)

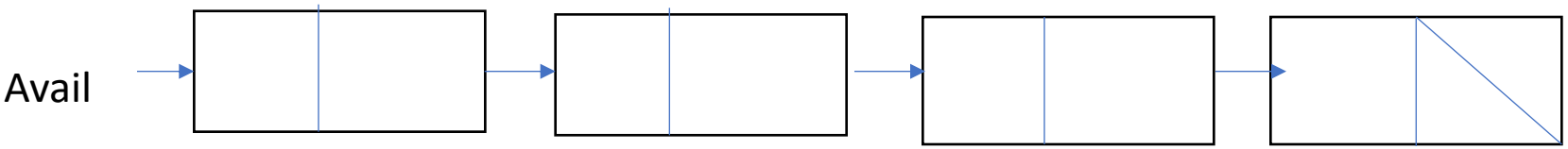

First=2000
3
4000
4
7000
5
8000
2000
4000
7000
8000

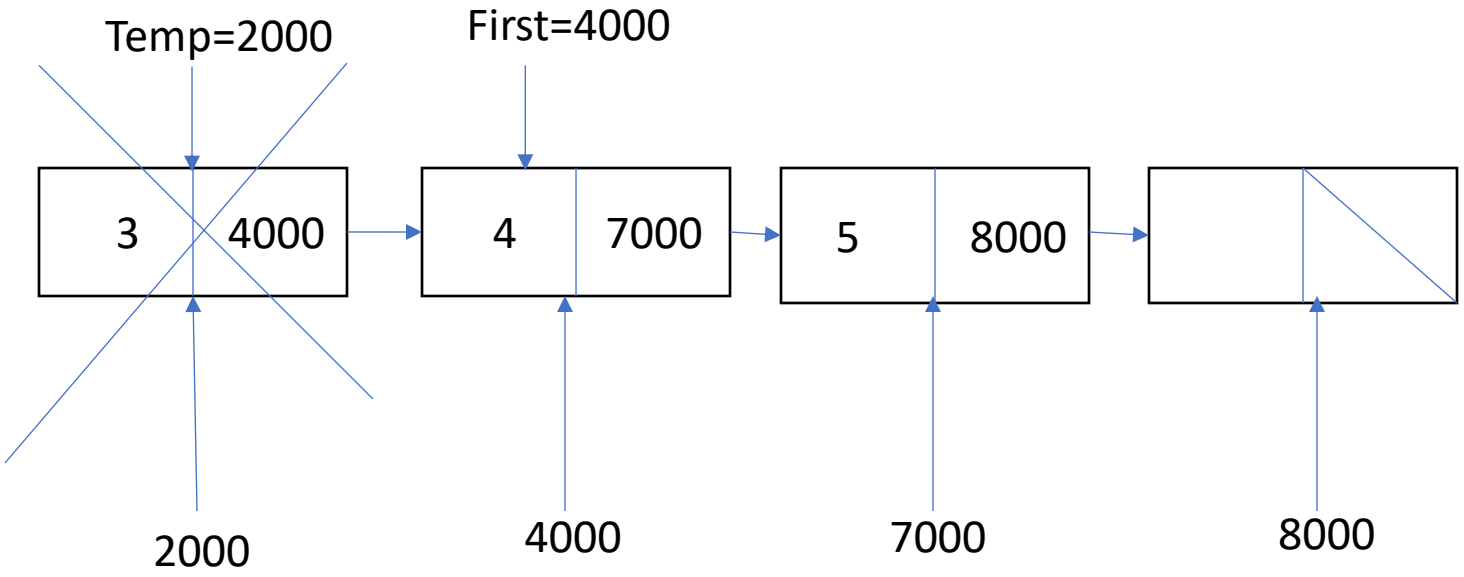
Y=3

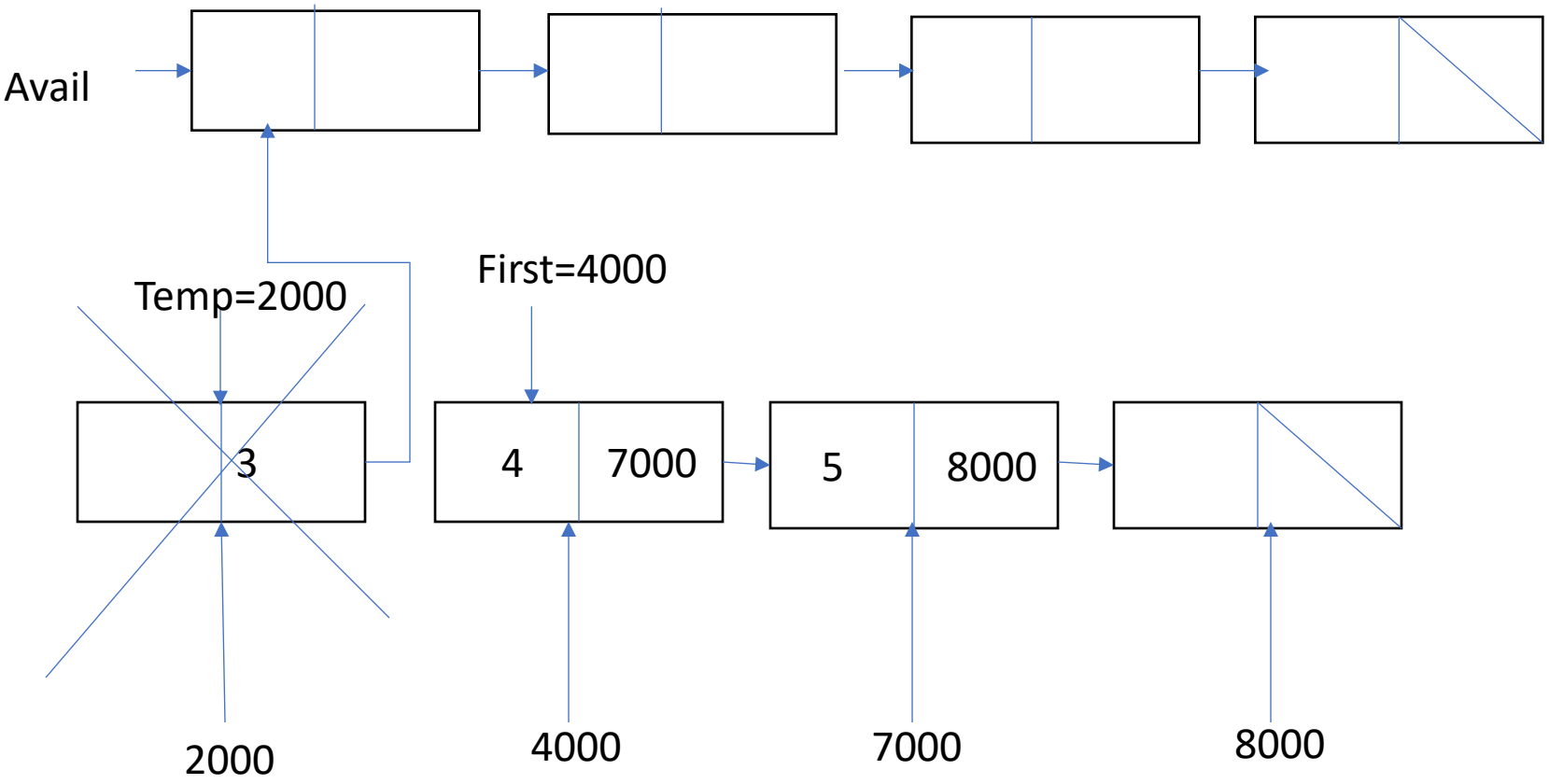


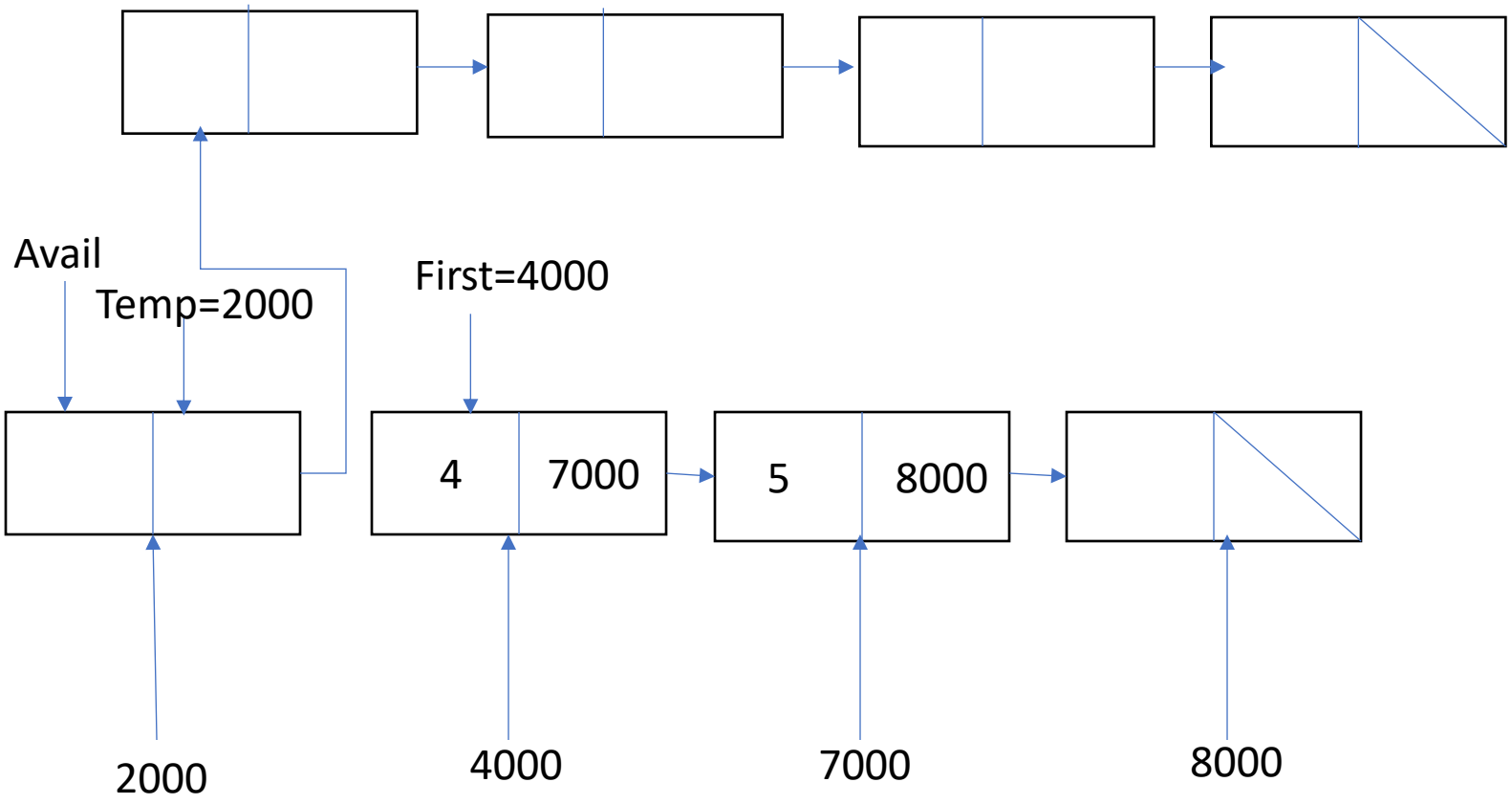
Y=3




Avail


Temp=2000
First=4000
3
4000
4
7000
5
8000
2000
4000
7000
8000


Avail
Temp=2000
First=4000
3
4
7000
5
8000
2000
4000
7000
8000


Avail
Temp=2000
First=4000
4
7000
5
8000
2000
4000
7000
8000

Algorithm to delete last node of linked list

DEL_END(FIRST)

- This function deletes a last node from the list.
- FIRST is a pointer which contains address of first node in the list.

1[Check for empty list]

If FIRST = NULL then

Write “List is empty”

Return FIRST

2[Check for the element in the list and delete it]

If LINK (FIRST) = NULL then

Y←INFO (FIRST)

Temp ←First

FIRST←NULL

Link(Temp) ←Avail

Avail ← Temp

Return NULL

3.(Assign the address pointed by FIRST pointer to SAVE pointer)

SAVE←FIRST

4. Repeat while LINK (SAVE) ≠ NULL

PRED←SAVE

SAVE←LINK (SAVE)

5 [Delete Last Node]

Y←INFO (SAVE)

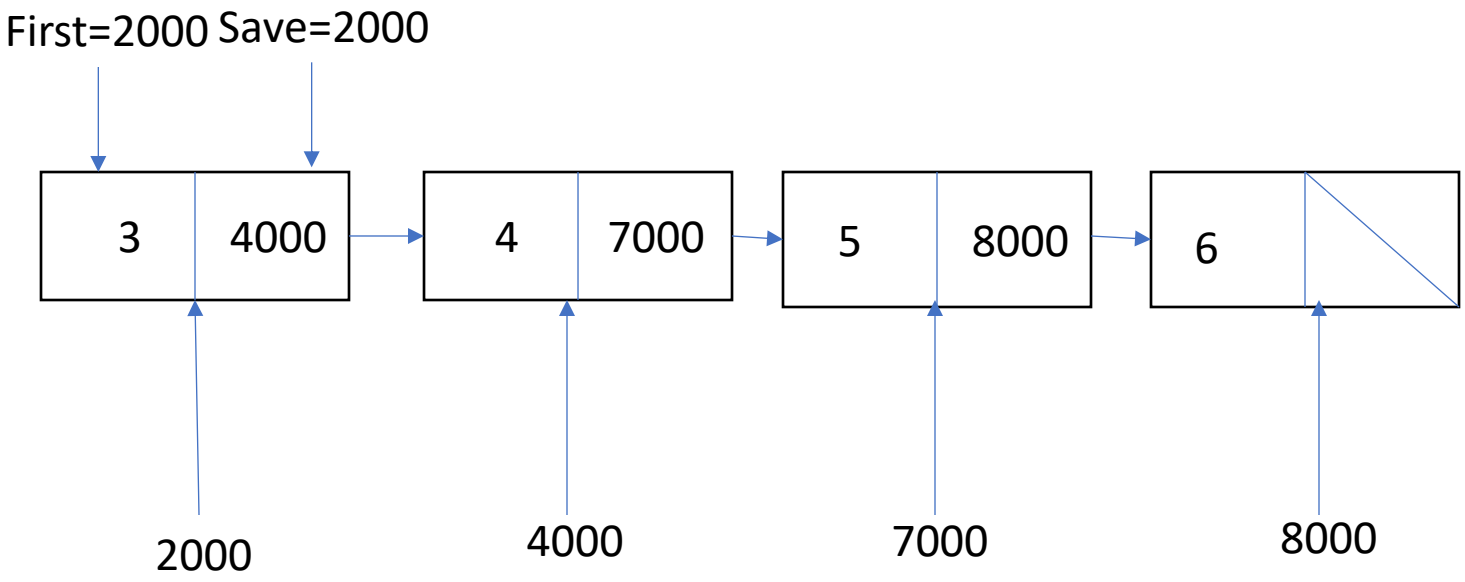
LINK (PRED) ←NULL

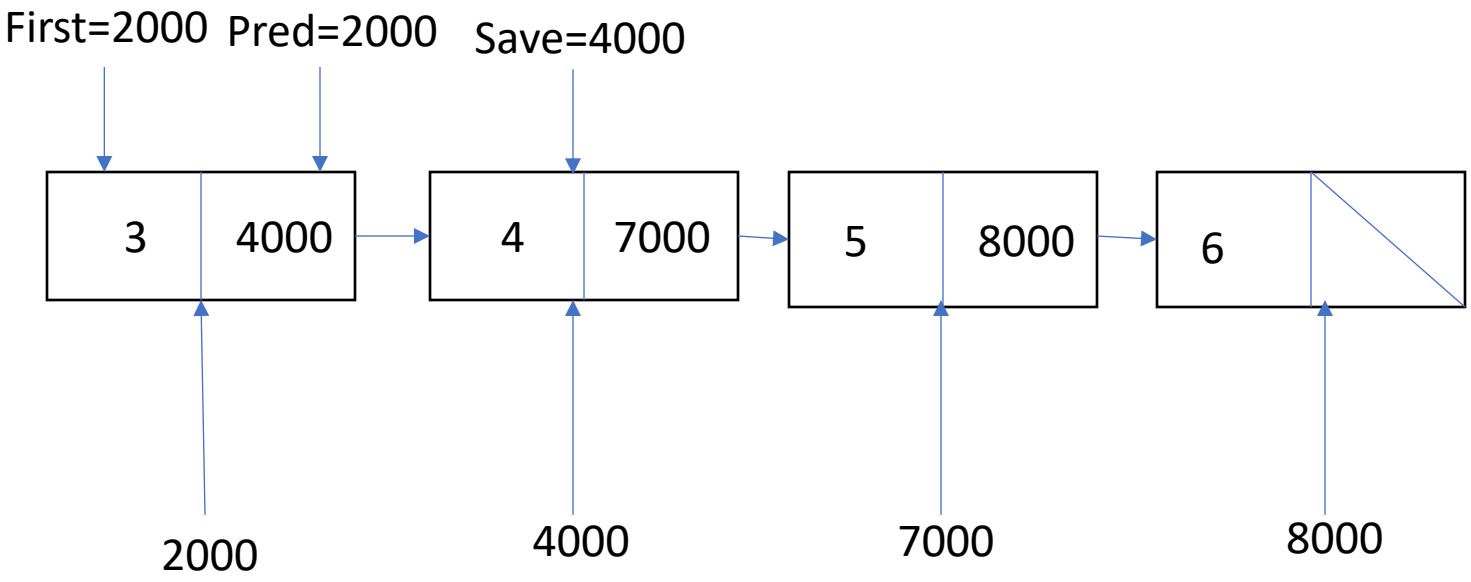
6. Link(Save) ← Avail

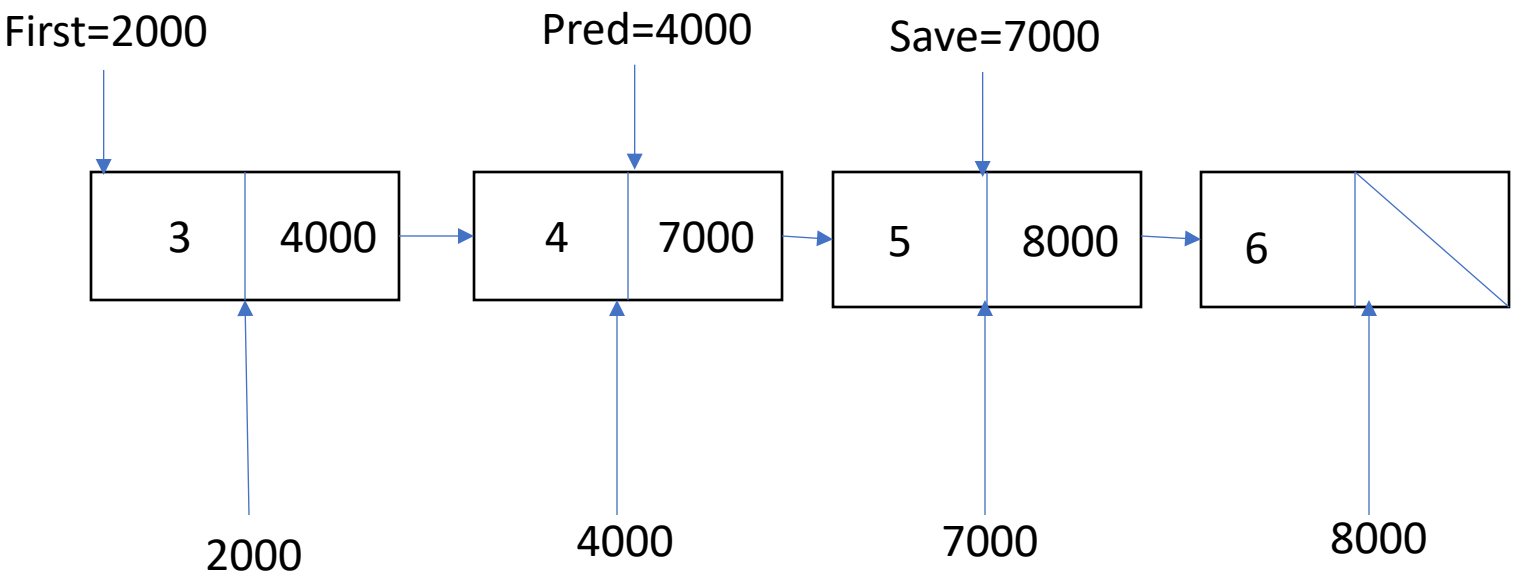
Avail ← Save

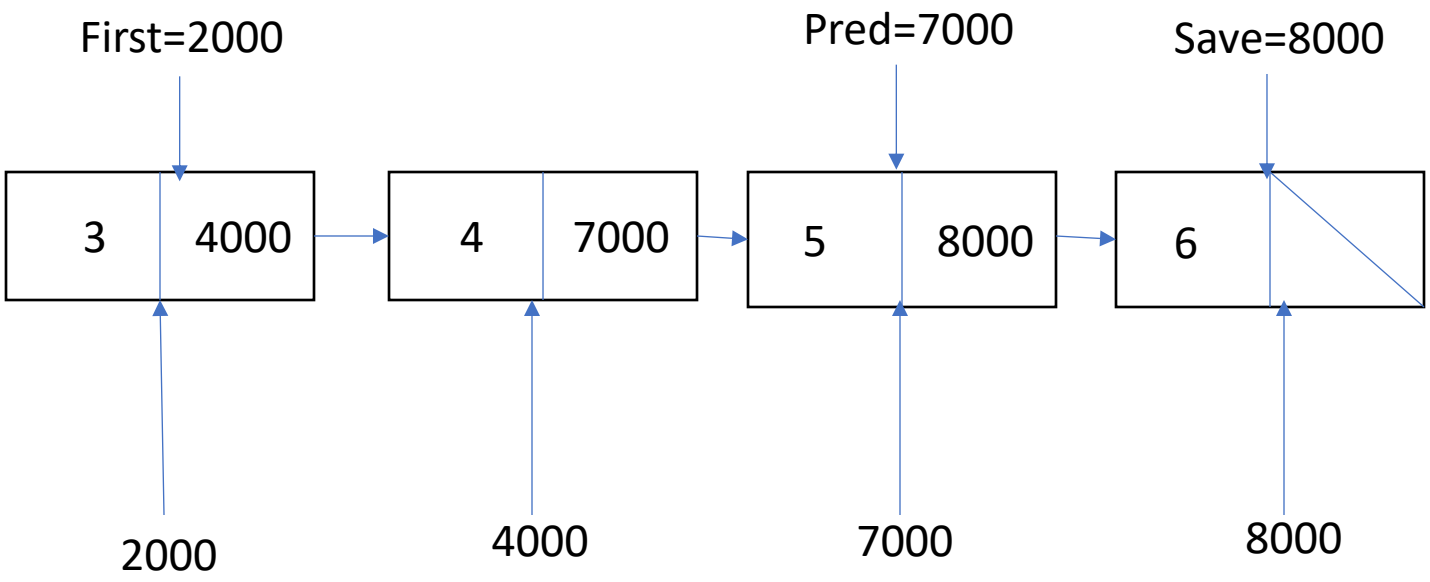
7[Finished]

Return (FIRST)

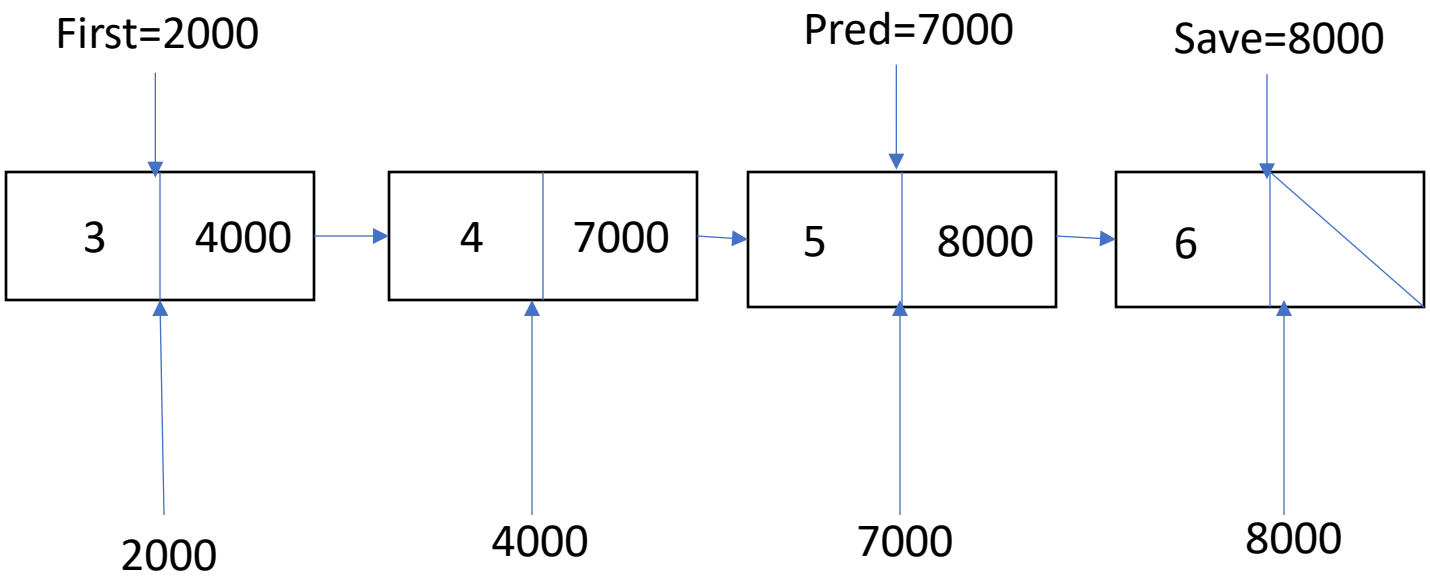

First=2000
Save=2000
3
4000
4
7000
5
8000
6
2000
4000
7000
8000

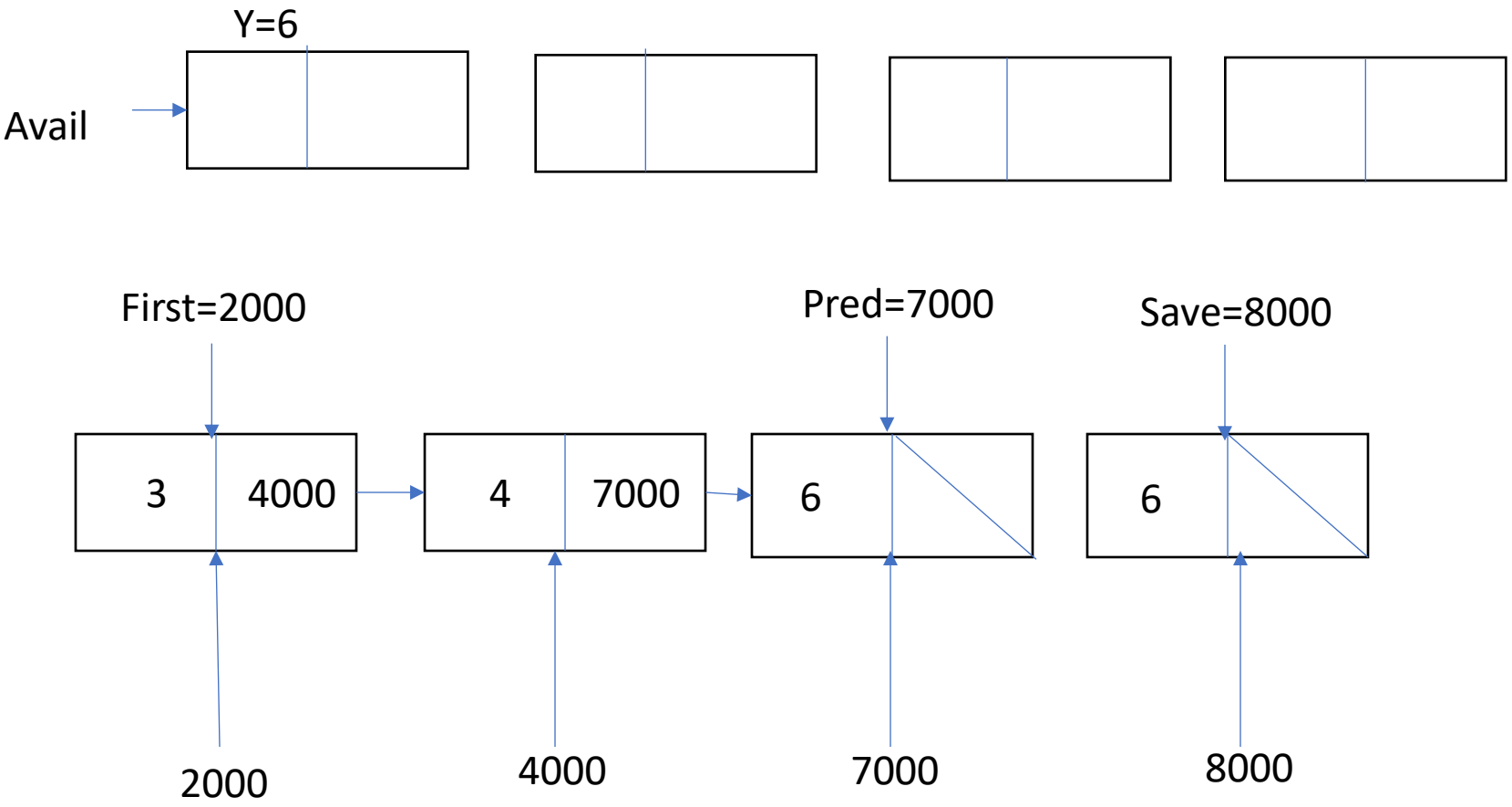

First=2000
Pred=2000
Save=4000
3
4000
4
7000
5
8000
6
2000
4000
7000
8000

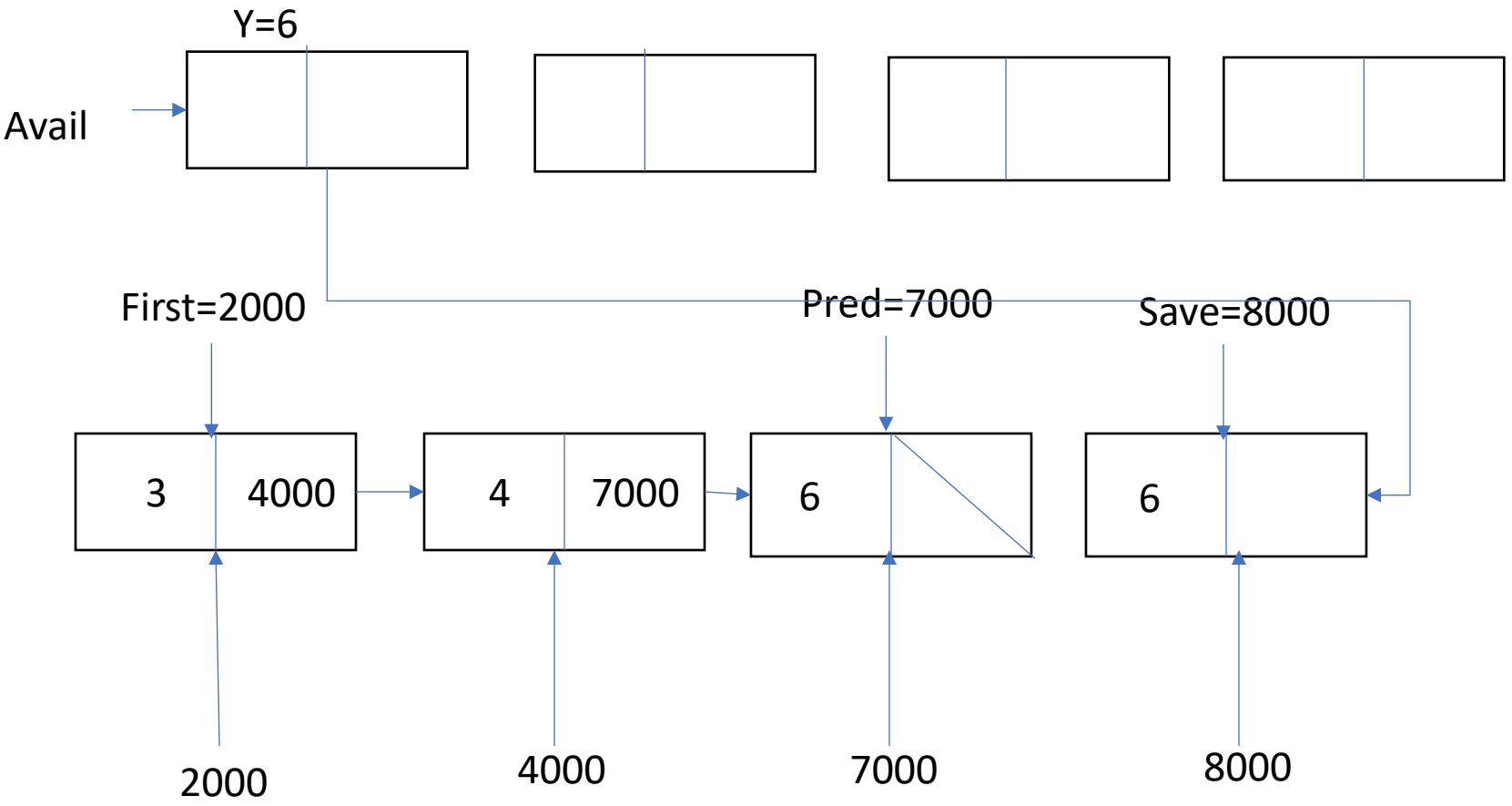

First=2000
Pred=4000
Save=7000
3
4000
4
7000
5
8000
6
2000
4000
7000
8000

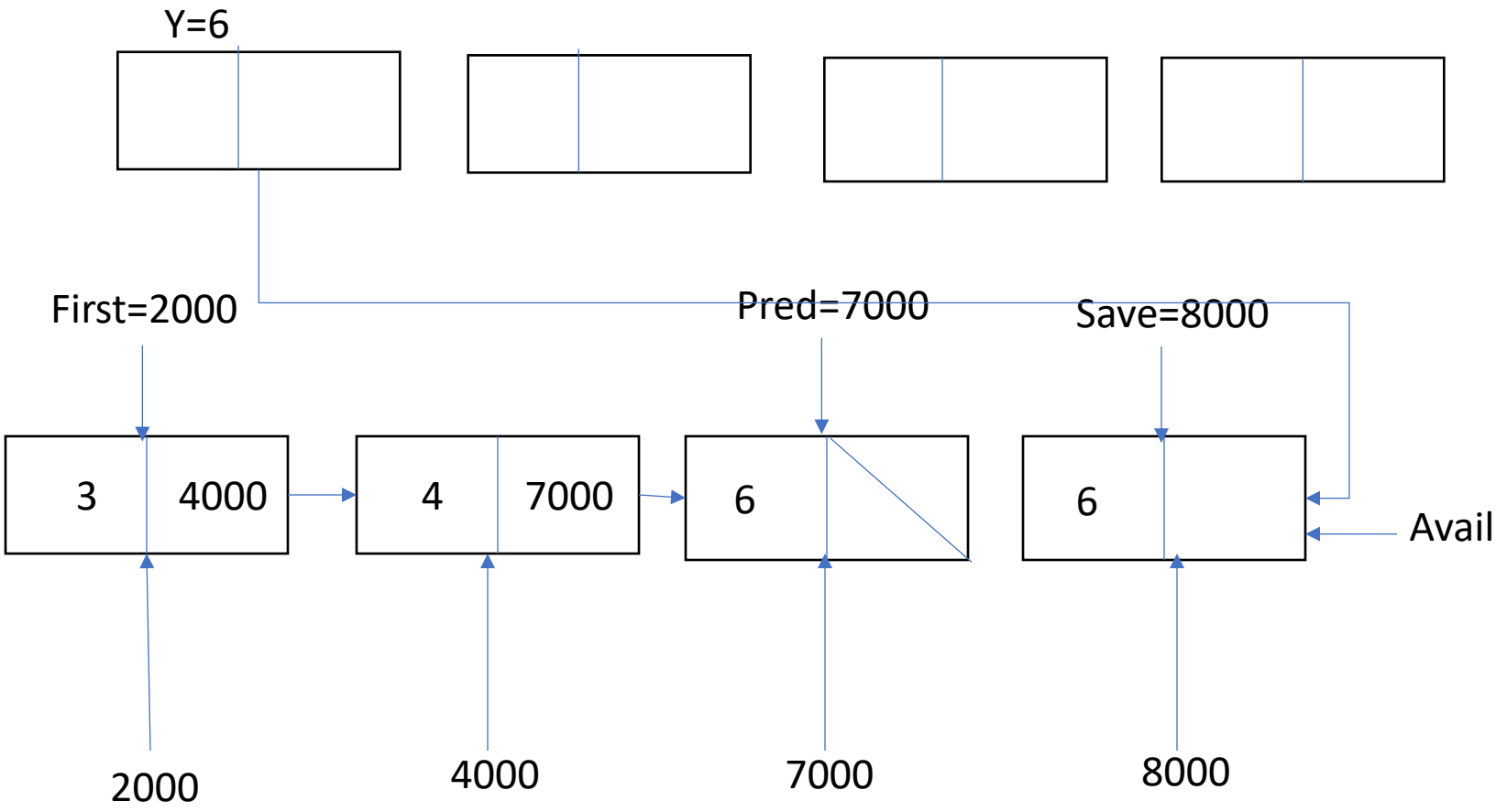

First=2000
Pred=7000
Save=8000
3
4000
4
7000
5
8000
6
2000
4000
7000
8000

Y=6


First=2000
Pred=7000
Save=8000
3
4000
4
7000
5
8000
6
2000
4000
7000
8000


Y=6
Avail
First=2000
Pred=7000
Save=8000
3
4000
4
7000
6
6
2000
4000
7000
8000


Y=6
Avail
First=2000
Pred=7000
Save=8000
3
4000
4
7000
6
6
2000
4000
7000
8000


Y=6
First=2000
Pred=7000
Save=8000
3
4000
4
7000
6
6
Avail
2000
4000
7000
8000

Q/A

1. Linked list is considered as an example of _____ type of memory allocation.

A. Dynamic
B. Static
C. Compile time
D. None of the mentioned

2. Consider an implementation of unsorted singly linked list. Suppose it has its representation with a head pointer only. Given the representation, which of the following operation can be implemented in O(1) time?
i) Insertion at the front of the linked list
ii) Insertion at the end of the linked list
iii) Deletion of the front node of the linked list
iv) Deletion of the last node of the linked list

ALGORITHM to Count the number of nodes in linked list

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:              SET COUNT = COUNT + 1
Step 5:              SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

Algorithm to Search a node in Linked List,

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:      IF VAL = PTR -> DATA
                  SET POS = PTR
                  Go To Step 5
             ELSE
                  SET PTR = PTR -> NEXT
             [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

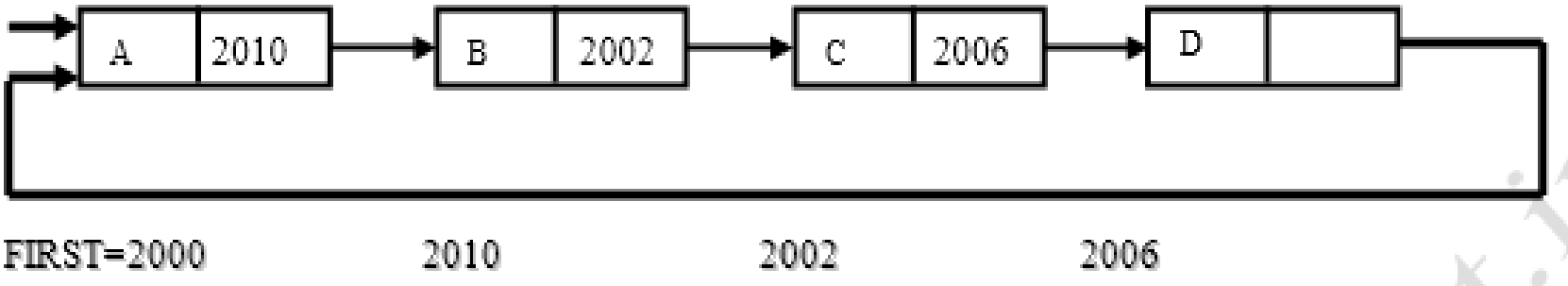
Topics –Part III

- 4.8 Concepts of circular linked list
- 4.9 Difference between circular linked list and singly linked list
- 4.10 Doubly linked list: Representation
- 4.11Difference between Doubly linked list and singly linked list
- 4.12 Applications of the linked list

Circular Linked list

Circular Linked list

- A list in which last node contains a link or pointer to the first node in the list is known as circular linked list.
- Representation of circular linked list is shown below:



- While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending

Circular Linked list Representation

```
struct node
{
int info;
struct node *link;
};
struct node *first;
```

Insert new node at the begging :Circular linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Circular Linked list

- Circular linked lists are often used when you need to access elements in a continuous loop
- For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application.
- It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- Circular Doubly Linked Lists are used for the implementation of advanced data structures like the Fibonacci Heap.

Singly Linked List	Circular Linked list
In singly linked list the last node contains NULL address. So if we are at middle of the list and want to access the first node we can not go backward in the list. Thus every node in the list is not accessible from given node.	While in Circular linked list last node contains an address of the first node so every node is in the list is accessible from given node.
Concatenation and splitting operations is difficult in the singly linked list.	Concatenation and splitting operations are efficient in circular linked list as compared to the singly linked list.
First node is set as starting point.	Any node can be set as the starting point.
Detection of end of list is easy. Null in link part of node indicates end of list.	Without some care in processing it is possible to get into the infinite loop. So we must able to detect end of the list.
It is simple.	Compared to singly linked lists, circular lists are more complex.
Reversing a list is simple in singly linked list.	Reversing a circular list is more complicated than singly or doubly linked list.
Singly linked lists are more commonly used for simple linear data structures where you only need to traverse in one direction	Circular linked lists are often used when you need to access elements in a continuous loop

Doubly link list

Doubly link list

- In Singly linked list we are able to traverse the list in only one direction. However in some cases it is necessary to traverse the list in both directions.

- This property of linked list implies that each node must contain two link fields instead of one link field.

- One link is used to denote the predecessor of the node and another link is used to denote the successor of the node.
- Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown

Doubly link list

Thus each node in the list consist of three fields:

1. Information
2. LPTR
3. RPTR

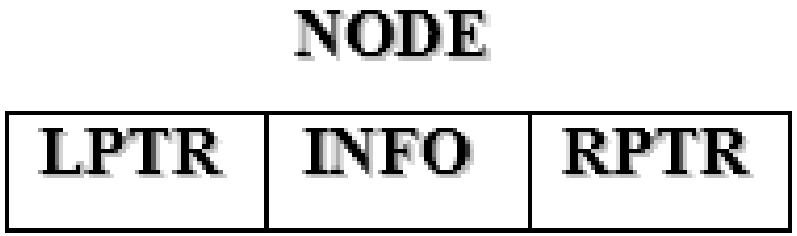


Figure 6.37 Doubly linked list

Doubly link list

```
struct node
 {
        struct node *prev;
        int data;
        struct node *next;
};
```

The PREV field of the first node and the NEXT field of the last node will contain NULL.

The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

Singly linked list.	Doubly Linked List
Node contains only 2 part. Data part and Link part.	Node contains 3 parts. One data part ,2 links parts that is one for previous and one for next.
It can be traversed only in single direction and forward only manner.	It can be traversed in both direction ,forward and backward manner.
Deletion operation is slower.	Deletion operation is faster
Searching operation is not efficient compared to doubly linked list.	Searching operation is efficient.
It requires less memory space per node compared to doubly linked list.	It requires more memory space per node.
Preferred for implementation of Stack and Queue	Preferred for implementation of Tree.

Insert New node at the beginning of Doubly linked list

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Insert New node at the end of Doubly linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Applications of the linked list

1. Implementation of stacks and queues
2. Implementation of graphs: Adjacency list representation of graphs is the most popular which uses a linked list to store adjacent vertices.
3. Implementation of Hash Table.
4. Performing arithmetic operations on long integers
5. Manipulation of polynomials :We can make use of a linked list to portray a polynomial. In the linked list, every node contains two data fields, namely coefficient, and power. Thus, every node denotes a term of a polynomial
6. Dynamic memory allocation: We use a linked list of free blocks.
7. In compiler ,for crating the symbol table.

Applications of the linked list

1. Image viewer – Previous and next images are linked and can be accessed by the next and previous buttons.
2. Previous and next page in a web browser – We can access the previous and next URL searched in a web browser by pressing the back and next buttons since they are linked as a linked list.
3. Music Player – Songs in the music player are linked to the previous and next songs. So you can play songs either from starting or ending of the list.