Linked Lists

- Introduction: memory representation, allocation and garbage collection.
- Operations: Traversal, insertion and deletion.

Introduction

- A linked list (One-way list) is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.

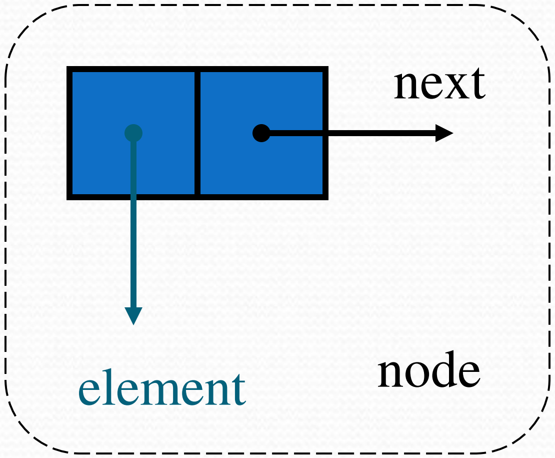
- Each node is divided into two parts.

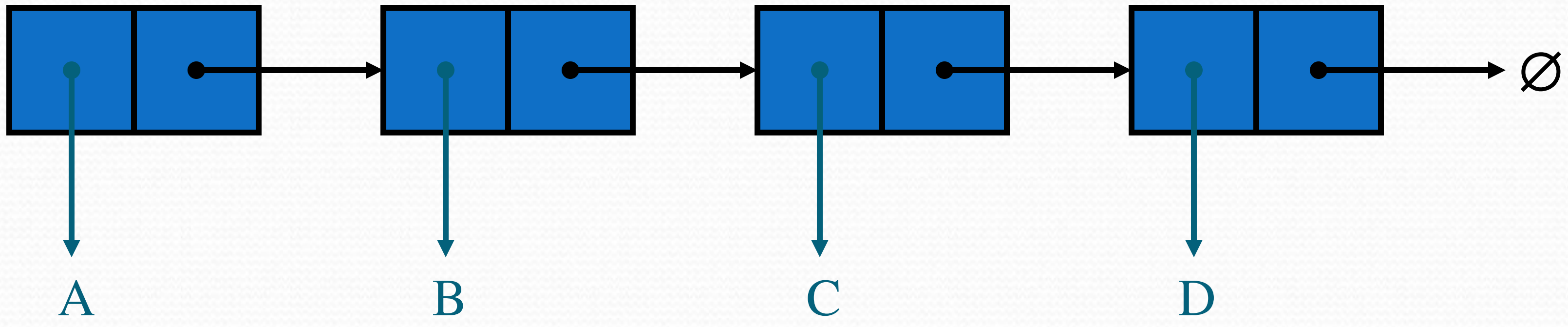
- First part contains the information of the element.

- Second part contains the address of the next node in the list.

Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



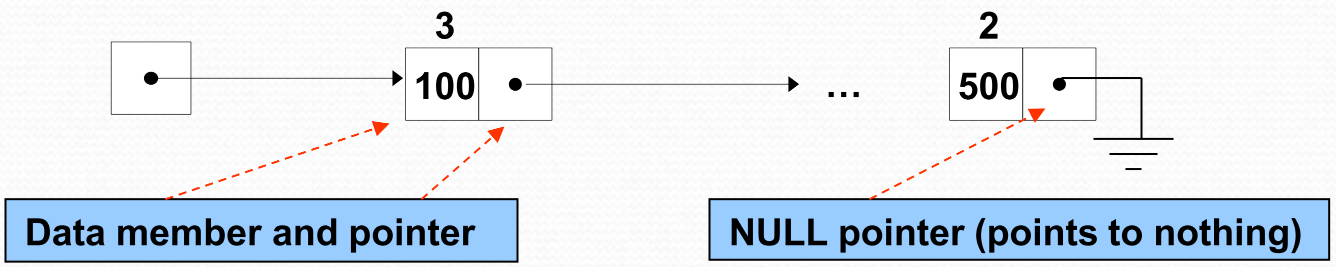


Key Points

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Link pointer in the last node is set to null to mark the list's end
  - Linked list contains a *Pointer Variable* called START, which contains the address of the first node.

Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a **`NULL`** pointer (**`0`**)
- Diagram of two self-referential structure objects linked together

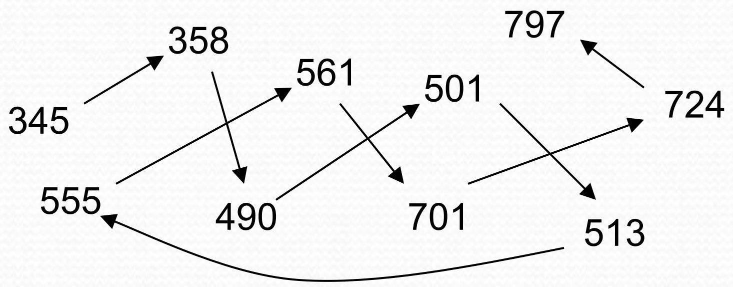


```
struct node {
     int info;
     struct node *link;
  };
```

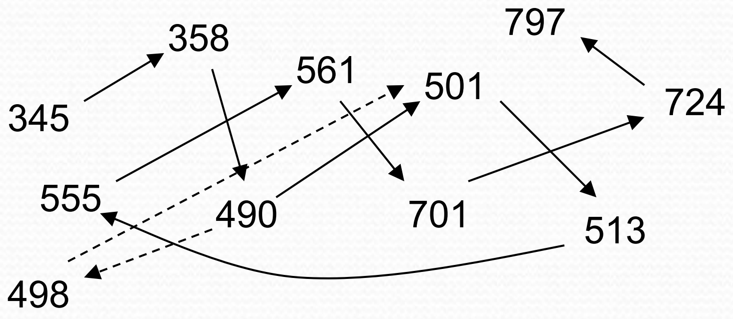
- `link`
  - Points to an object of type node
  - Referred to as a link of one node to next node.

Linked Representation of Data

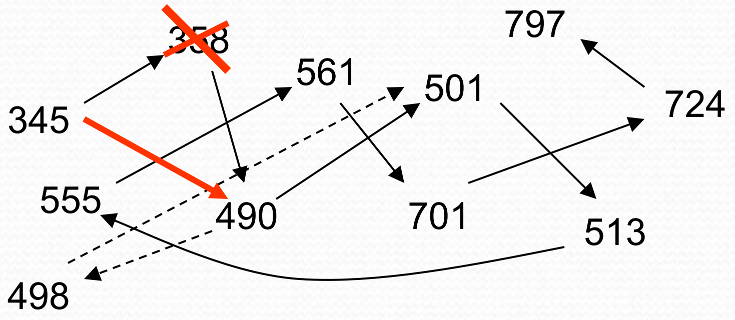
- In a linked representation, data is not stored in a contiguous manner. Instead, data is stored at random locations and the current data location provides the information regarding the location of the next data.



Adding item **498** on to the linked list
Q: What is the cost of adding an item?
Q: how about adding **300** and **800** onto the linked list

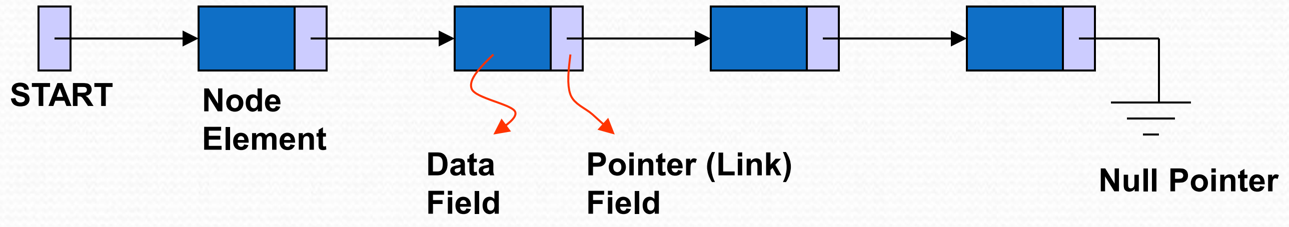


Deleting item **358** from the linked list
Q: What is the cost of deleting an item?
Q: What is the cost of searching for an item?



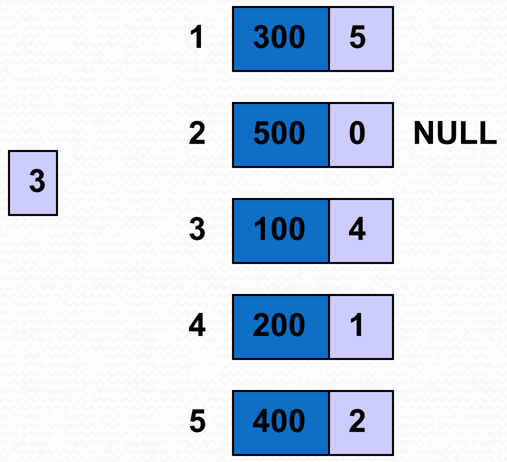
Linked List

- How do we represent a linked list in the memory
  - Each location has two fields: Data Field and Pointer (Link) Field.
- Linked List Implementation



```
struct node {
    int data;
    struct node *link;
 };
 struct node my_node;
```

Example:



1	300	5
2	500	0
3	100	4
4	200	1
5	400	2

Why Linked List?

Arrays: pluses and minuses

+ Fast element access.

-- Expensive insertion deletion

-- Impossible to resize.

Need of Linked List

- Many applications require resizing!
- Required size not always immediately available.

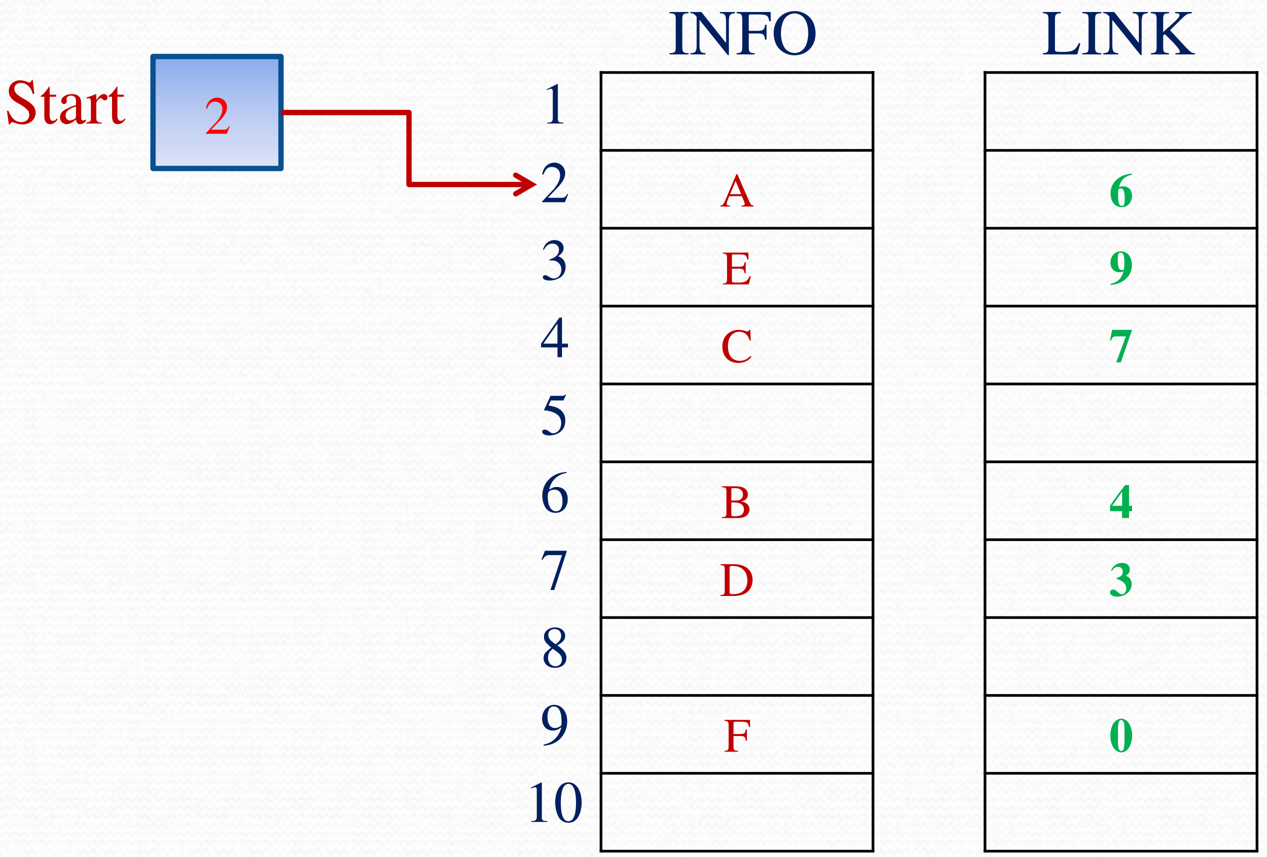
- Use a linked list instead of an array when
  - You have an unpredictable number of data elements
  - You want to insert and delete quickly.

Features of Linked List

- Lists are a particularly flexible structure because they can grow and shrink on demand, and elements can be accessed, inserted, or deleted at any position within a list.
- Lists can also be concatenated together or split into sublists.
- Lists arise routinely in applications such as information retrieval, programming language translation, and simulation.

Memory Representation

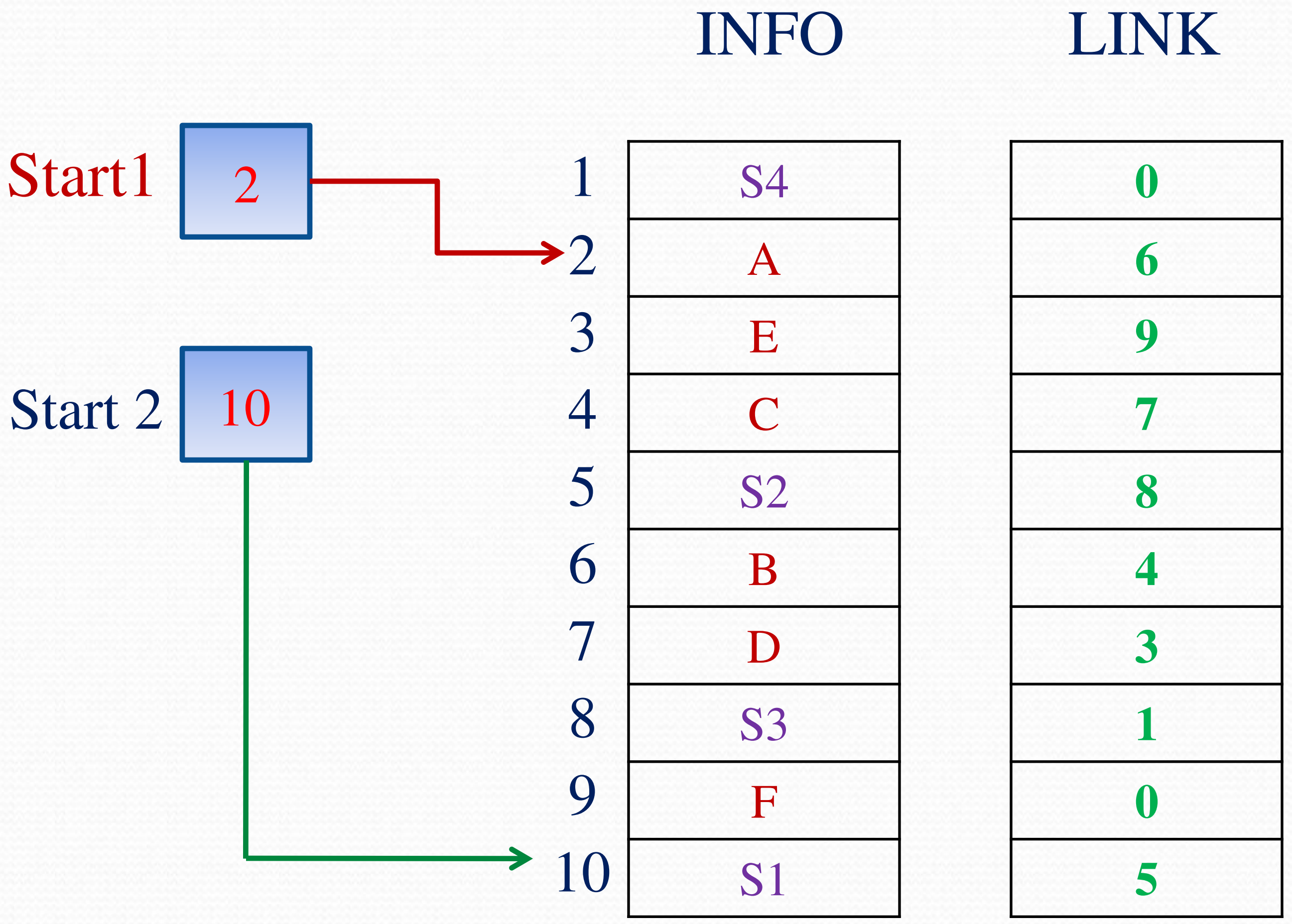
- Linked lists are maintained in memory using linear arrays or Parallel arrays.



	INFO	LINK
1		
2	A	6
3	E	9
4	C	7
5		
6	B	4
7	D	3
8		
9	F	0
10		

Memory Representation (2)

- Multiple lists in memory



	INFO	LINK
1	S4	0
2	A	6
3	E	9
4	C	7
5	S2	8
6	B	4
7	D	3
8	S3	1
9	F	0
10	S1	5

Memory Representation (3)

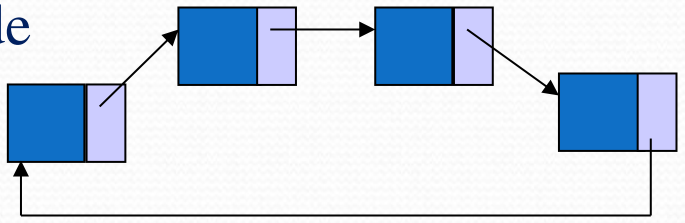
- INFO part of a node may be a record with multiple data items.
- Records in memory using Linked lists

Start: 2

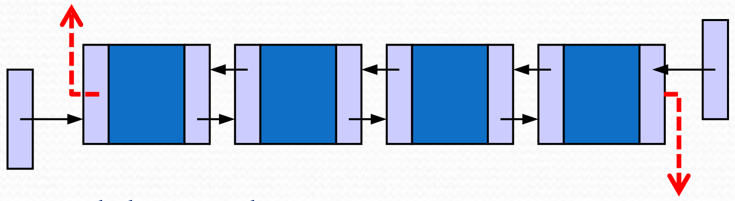
	Name	Age	Sex	LINK
1				
2	A	19	M	6
3	E	21	M	9
4	C	20	F	7
5				
6	B	19	F	4
7	D	22	M	3
8				
9	F	20	F	0
10				

Types of Linked Lists

1. Singly linked list
   - Begins with a pointer to the first node
   - Terminates with a null pointer
   - Only traversed in one direction



2. Circular, singly linked
   - Pointer in the last node points back to the first node

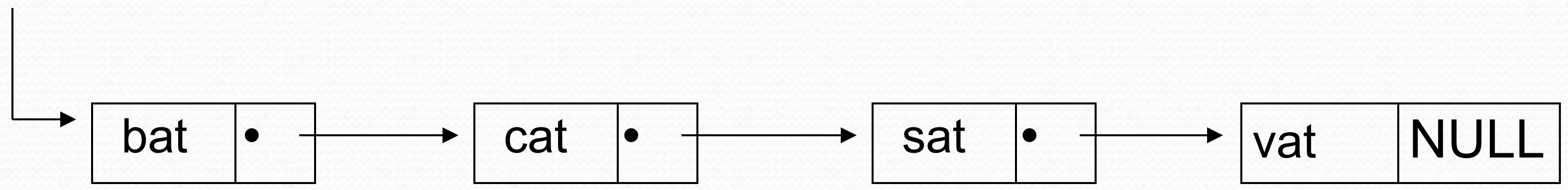


3. Doubly linked list
   - Two “start pointers” – first element and last element
   - Each node has a forward pointer and a backward pointer
   - Allows traversals both forwards and backwards
4. Circular, doubly linked list
   - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Difference between Singly Linked List and Arrays

Singly linked list	**Array**
• Elements are stored in linear order, accessible with links.	• Elements are stored in linear order, accessible with an index.
• Do not have a fixed size.	• Have a fixed size.
• Cannot access the previous element directly.	• Can access the previous element easily.
• No binary search.	• Binary search.

Singly Linked List and algorithms to perform operations on it



Traversal in Linked List

- Is to process information in each nodes' info part
- We initially only knows the address of first node that's to perform operations such as traversal ,searching , insertion at middle, deletion at middle. We have to traverse the linked list sequentially.
- We will use while loop to perform traversal
  - Initialisation ptr=start
  - Condition while ptr!= null
  - Increment ptr=link[ptr]

Algorithm Traversal In linked list

Traversal(INFO,LINK,PTR,START): It is to traverse the linked list .

```
1. Set PTR := START                        [Initializes pointer Ptr]
2. Repeat step 3 and 4 while PTR != NULL   [Condition]
3.     Apply Process to INFO[PTR]
4.     Set PTR := LINK[PTR]    [Ptr now points to next node]
   [End of step 2 loop]
5. Exit
```

Searching in linked list

- As linked list is sequential data structure and initially we only know the address of first node only that's why only linear search can be applied to linked list and **not binary search.**
- to perform searching first traverse the linked list upto the point that
- Info[PTR]=ITEM, if it found then break.

Algorithm of searching in unsorted linked list

Search(INFO,START,LINK,PTR,ITEM,LOC):): It is to search ITEM the linked list where INFO representing the information part in each node, LINK is the link part holding the address of next node. PTR id pointer variable, START is pointer variable holding the address of first node. ITEM holds the value to search. LOC hold address of node found location

```
1.  Set PTR := START                     [Initializes pointer Ptr]
2.  Repeat step 3 while PTR!= NULL
3.      If ITEM = INFO[PTR], then:
              Set LOC := PTR    and  Exit
        Else
              Set PTR := LINK[PTR]          [Ptr now points to next node]
        [End of If structure]
    [End of step 2 loop]
4.  [Search is unsuccessful] Set LOC := NULL
5.  Exit
```

Algorithm of searching in sorted linked list

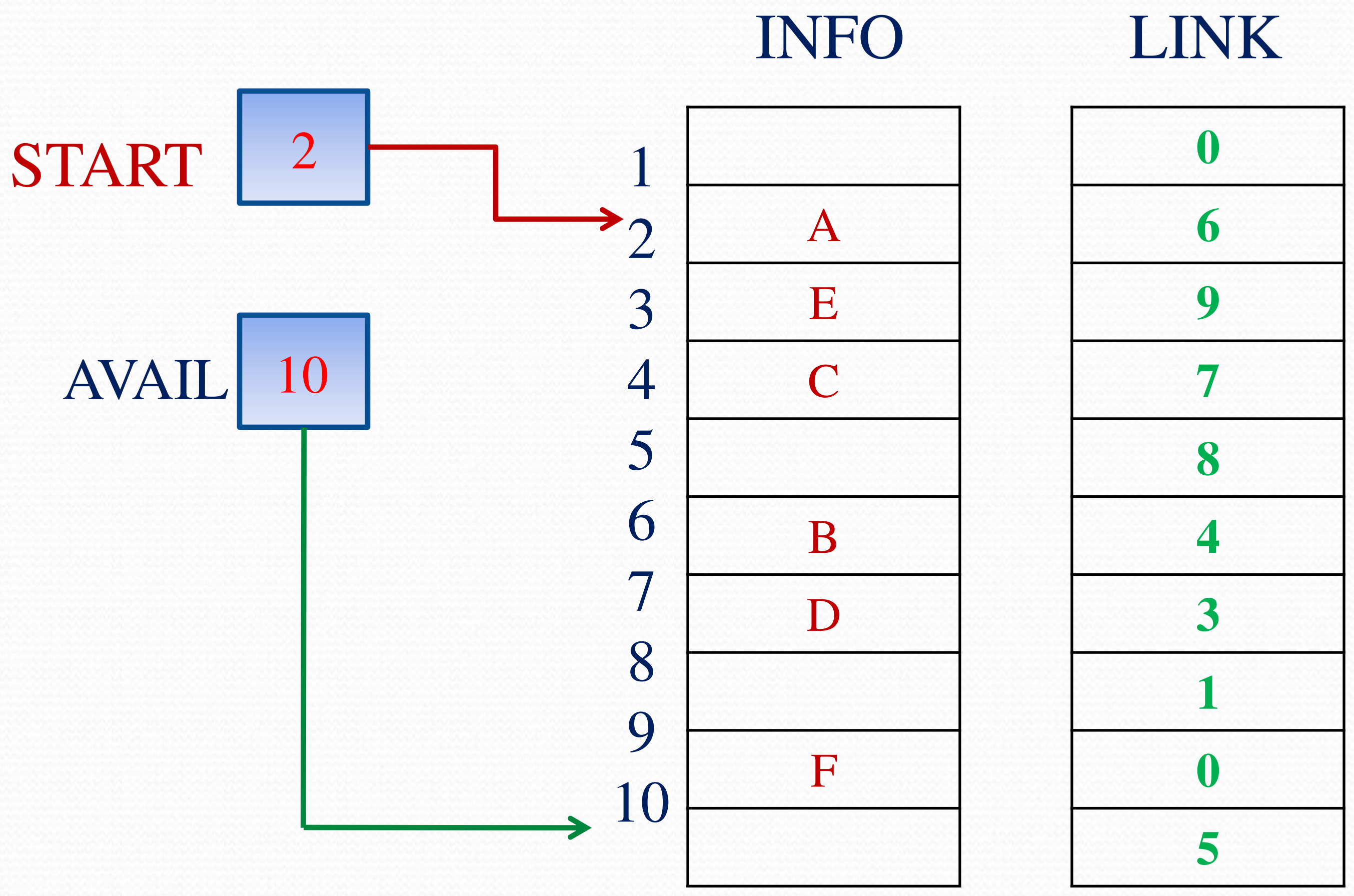
```
SEARCH (INFO, LINK, START, ITEM, LOC)
1.  Set PTR: = START.
2.  Repeat Step 3 While PTR ≠ NULL.
3.      If ITEM > INFO [PTR], then:
            Set PTR: = LINK [PTR].                      [PTR points to next node]
        Else if ITEM = INFO [PTR], then:
            Set LOC: = PTR, and EXIT.                   [Search is successful.]
        Else:
            Set LOC := NULL, and EXIT          . [ITEM exceeds INFO[PTR] ...]
        [End of if Structure.]
    [End of step 2 Loop.]
4.  Return  LOC .
5.  Exit.
```

Memory Allocation

- Together with the linked list, a special list is maintained which consists of unused memory cells.
- This list has its own pointer.
- This list is called *List of available space* or *Free-storage list* or *Free pool*.

Free Pool

- Linked list with free pool or list of Available space. Address of first free node is stored in avail pointer variable



	INFO	LINK
1		0
2	A	6
3	E	9
4	C	7
5		8
6	B	4
7	D	3
8		1
9	F	0
10		5

Garbage Collection

- Garbage collection is a technique of collecting all the deleted spaces or unused spaces in memory.
- The OS of a computer may periodically collect all the deleted space onto the free-storage list.
- Garbage collection may take place when there is only some minimum amount of space or no space is left in free storage list.
- Garbage collection is invisible to the programmer.

Garbage Collection Process

- Garbage collection takes place in two steps.

1. The computer runs through all lists tagging those cells which are currently in use.

2. Then computer runs through the memory, collecting all the untagged spaces onto the free storage list.

Overflow and Underflow

- Overflow: When a new data are to be inserted into a data structure but there is no available space i.e. the free storage list is empty.

- Overflow occurs when *AVAIL = NULL*, and we want insert an element.

- Overflow can be handled by printing the *'OVERFLOW'* message and/or *by adding space* to the underlying data structure.

Overflow and Underflow

- Underflow: When a data item is to be deleted from an empty data structure.

- Underflow occurs when *START = NULL*, and we want to delete an element.

- Underflow can be handled by printing the *'UNDERFLOW'* message.

Insertion into a Linked List

- New node N (which is to be inserted) will come from AVAIL list.
- First node in the AVAIL list will be used for the new node N.

- Types of insertion:
- Insertion at the beginning
- Insertion between two nodes
- Insertion after given node.

Algorithmic syntax to perform 4 steps to insert node(NOT ALGO)

1.Check availability of node in avail list

```
    If AVAIL= null ,then:
    Write: overflow
```

2.If available then remove the node from avail list

```
    Set NEW:=AVAIL
    Set AVAIL:=LINK[AVAIL]
```

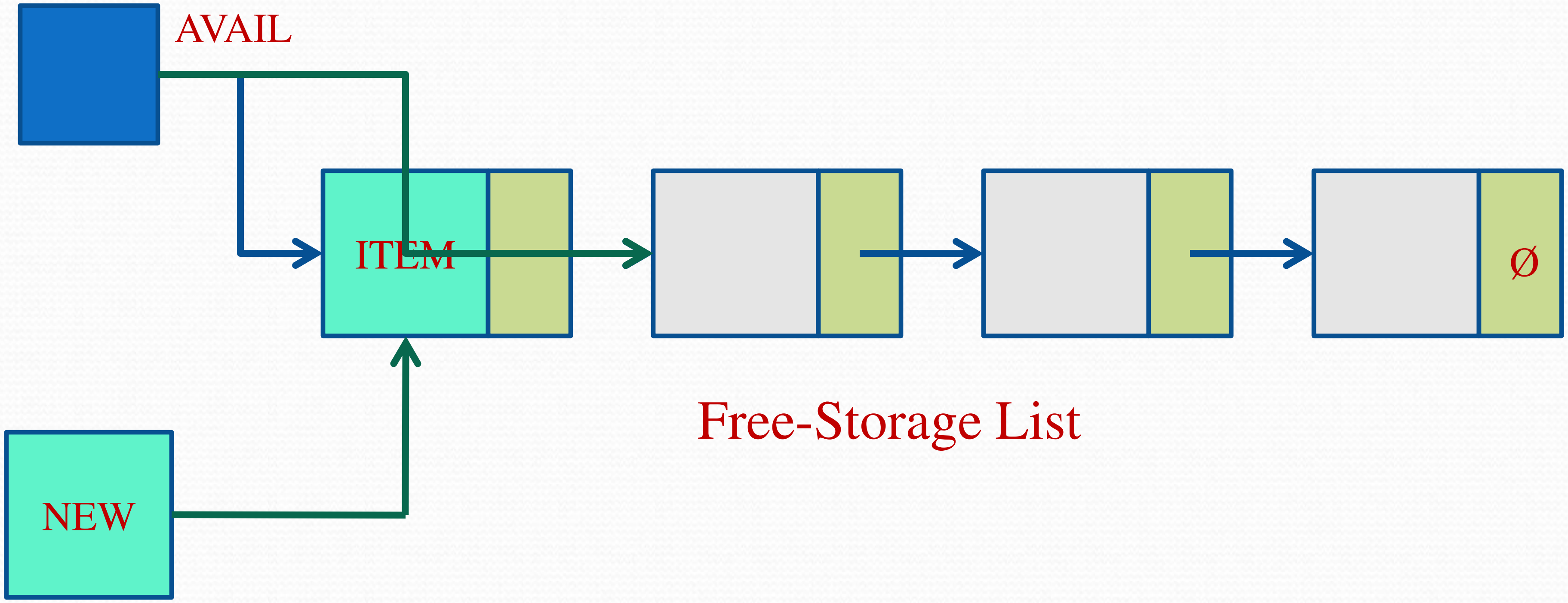
3.Insert element to info part of that node

```
    Set INFO[NEW]=ITEM
```

4.Link the node with linked list where i want to perform insertion

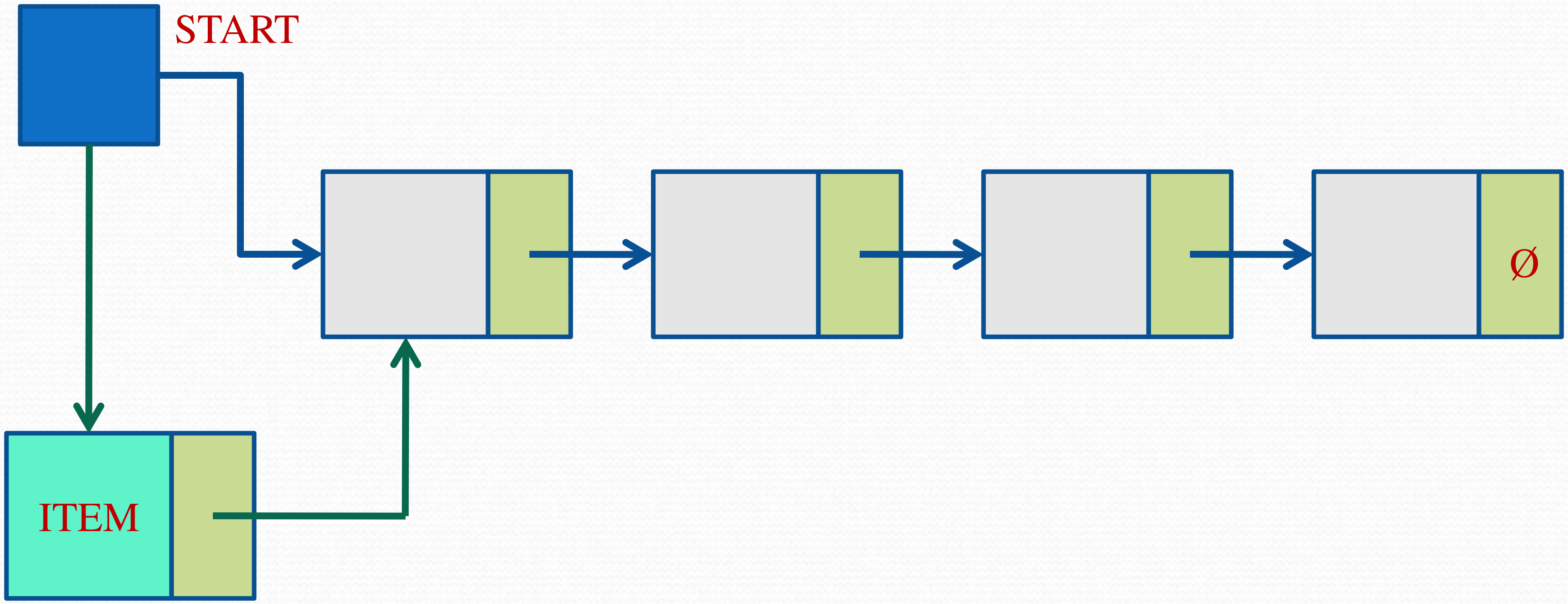
```
     Set LINK[NEW]:=START   //when insert at the beginning
     Set START:=NEW
          OR
     Set LINK[NEW]=LINK[LOC]//when insert after some location LOC
     Set Link[LOC]=NEW
```

Checking the Available List



Free-Storage List

Insertion at the beginning of Linked List

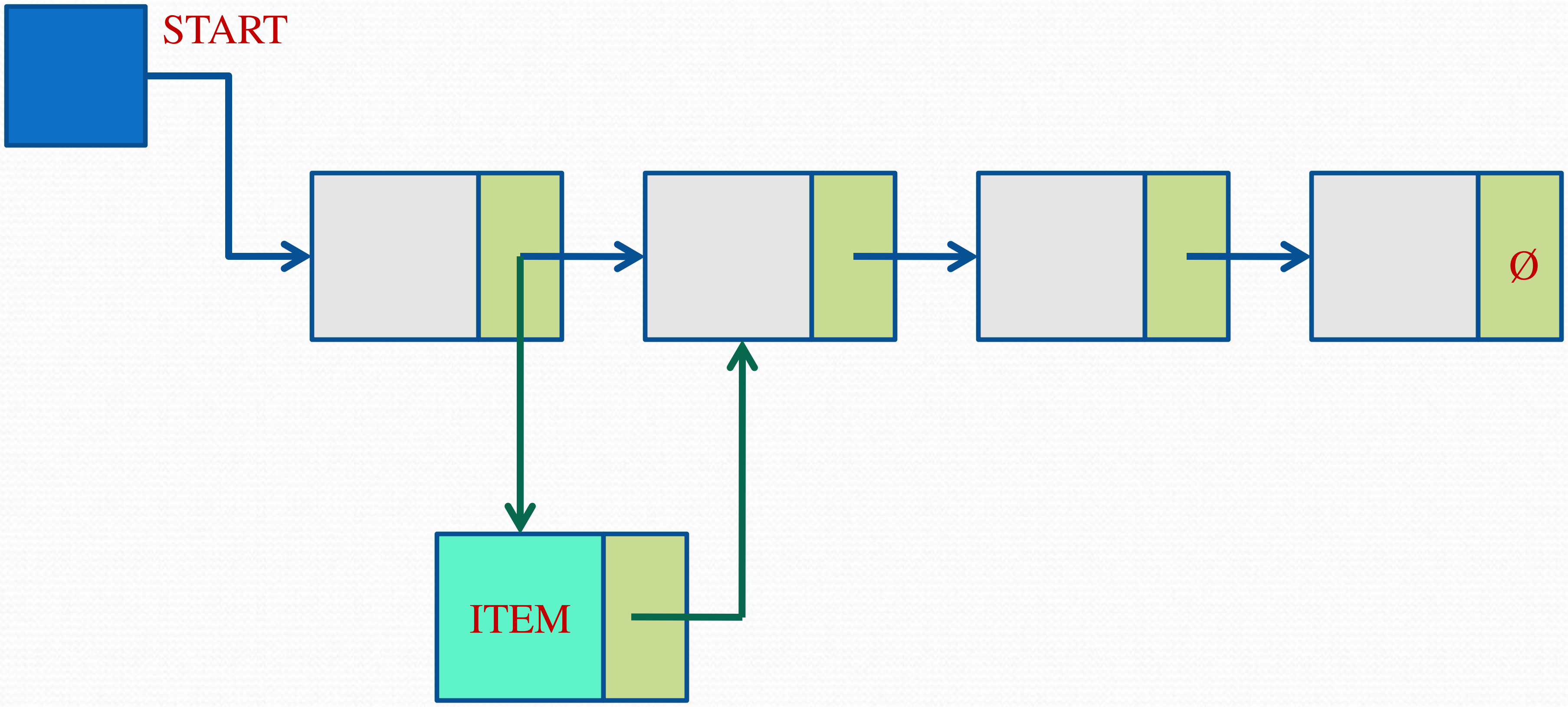


(1) Insertion Algorithm (Beginning of the list)

INSFIRST (INFO, LINK, START, AVAIL, ITEM):to insert element at the beginning

```
1.  [OVERFLOW?] If AVAIL = NULL, then:
        Write: OVERFLOW, and Exit.
2.  [Remove first node from AVAIL list]
    Set NEW:= AVAIL and AVAIL:= LINK [AVAIL].
3.  Set INFO [NEW] := ITEM.     [Insert data to new node].
4.  Set LINK [NEW]: = START. [New node points to the original first
    node].
5.  Set START := NEW.        [START points to the new node.]
6.  EXIT.
```

Insertion after a given node



(2) Insertion Algorithm (After a given node)

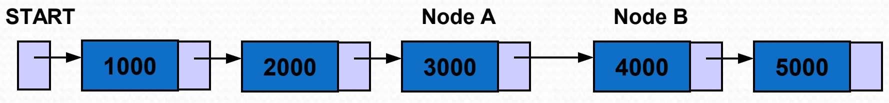
INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM): LOC is location of node after which we want to insert new node

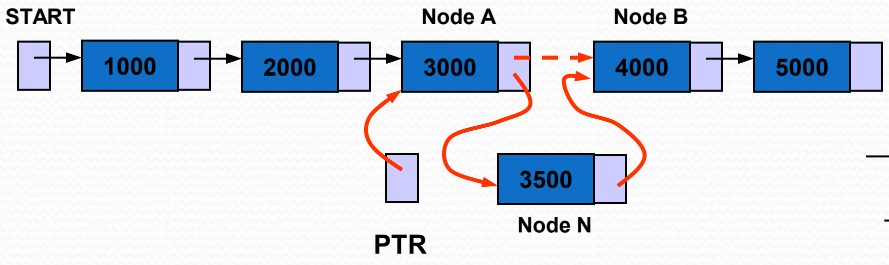
```
1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list]
   Set NEW:= AVAIL and AVAIL:= LINK [AVAIL].
3. Set INFO [NEW] := ITEM. [Copy the new data to the node].
4. If LOC = NULL, then: [Insert as a first node]
        Set LINK [NEW] := START and START := NEW.
   Else: [Insert after node with location LOC.]
        Set LINK [NEW]: = LINK [LOC] and LINK [LOC]: = NEW.
   [End of If structure.]
5. Exit.
```

(3)Algorithm to insert node in between two given nodes node

- INSERT(DATA, LINK, START, A, B, N):Let START be a pointer to a linked list in memory with successive nodes A and B. Write an algorithm to insert N between nodes A and B. Where A and B are address of nodes.

```
1.  Set PTR := START
2.  Repeat step 3 to 8 while PTR ≠ NULL
3.  If PTR = A, then:
4.      Set LINK[N]: = LINK[PTR] (or = B)
5.      Set LINK[PTR]: = N
6.      EXIT
7.  else
8.      Set PTR:=LINK[PTR]
9.  If PTR = NULL,then:
10. Write: insertion unsuccessful.
11. EXIT
```





Insertion into a sorted Linked List

- If ITEM is to be inserted into a sorted linked list. Then ITEM must be inserted between nodes A and B such that:

  INFO [A] < ITEM < INFO [B]

- First of all find the location of Node A
- Then insert the node after Node A.

INSERT (INFO, LINK, START, AVAIL, ITEM)

```
1.  CALL FIND_A (INFO, LINK, START, AVAIL, ITEM)
     [USE Algorithm FIND_A to find the location of node preceding
    ITEM.]
2.  CALL INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)
     [Insert ITEM after a given node with location LOC.]
3.  Exit.
```

FIND_A (INFO, LINK, START, ITEM, LOC)

```
1.  [List Empty?] If START := NULL, then: Set LOC: = NULL, and Return.
2.  [Special Case?] If ITEM < INFO [START], then: Set LOC := NULL, and
    Return.
3.  Set SAVE: = START and PTR := LINK [START].  [Initializes pointers]
4.  Repeat step 5 and 6 while PTR ≠ NULL.
5.          If ITEM < INFO [PTR] then:
                    Set LOC: = SAVE, and Return.
            [End of If Structure.]
6.      Set SAVE: = PTR and PTR: = LINK [PTR]. [Update pointers]
    [End of Step 4 Loop.]
7.  Set LOC: = SAVE.
8.  Return.
```

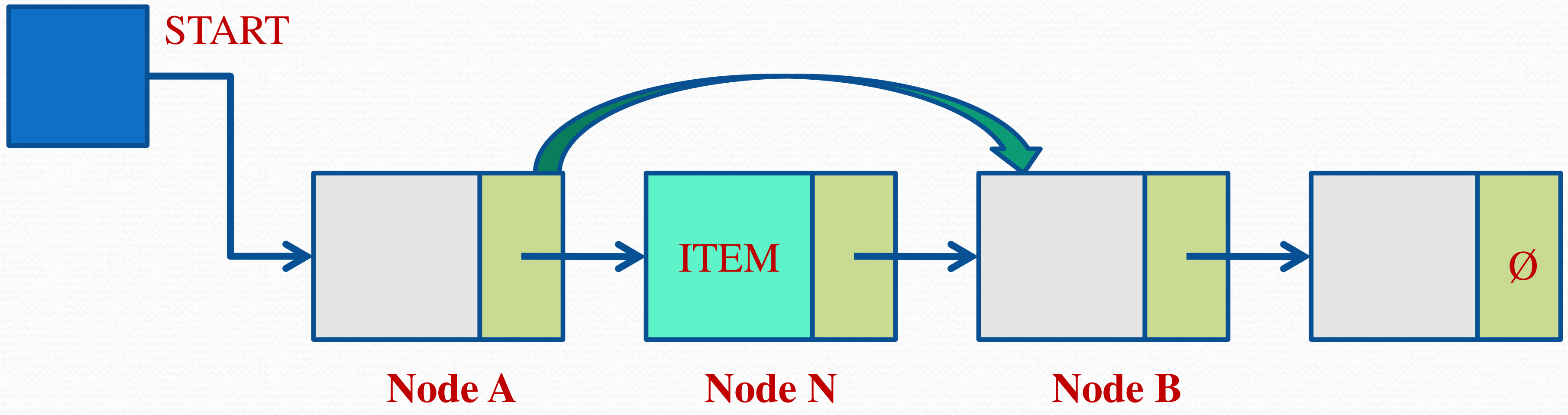
Deletion from a Linked List

- A node N is to be deleted from the Linked List.
- Node N is between node A and node B.

- Deletion occurs as soon as the next pointer field of node A is changed so that it points to node B.

- Types of Deletion:
- Deleting the node following a given node
- Deleting the Node with a given ITEM of Information.

Deletion from Linked List



Maintaining the AVAIL List

- After the deletion of a node from the list, memory space of node N will be added to the beginning of AVAIL List.

- If LOC is the Location of deleted node N:

```
LINK [LOC] := AVAIL
    AVAIL := LOC
```

Deleting the Node Following a given Node

DEL (INFO, LINK, START, AVAIL, LOC, LOCP)

```
1.  If LOCP = NULL, then:
          Set START := LINK [START]. [Delete First node.]
    Else:
          Set LINK [LOCP]: = LINK [LOC]. [Delete node N.]
    [End of If Structure.]

2.  [Return Deleted node to the AVAIL list]
    Set  LINK [LOC] = AVAIL and AVAIL= LOC.

3.  EXIT.
```

Deleting the Node with a given ITEM of Information

DELETE (INFO, LINK, START, AVAIL, ITEM)

```
1.  Call FIND_B (INFO, LINK, START, ITEM, LOC, LOCP)
                    [Find the Location of node N and its preceding node]
2.  If LOC = NULL, then: Write: ITEM not in LIST and EXIT.

3.  [Delete node].
    If LOCP = NULL, then:
        Set START: = LINK [START]. [Delete First node]
    Else:
        Set LINK [LOCP]: = LINK [LOC].
    [End of If Structure.]

4.  [Return Deleted node to the AVAIL list]
    Set  LINK [LOC] := AVAIL and AVAIL:= LOC.
5.  EXIT.
```

FIND_B (INFO, LINK, START, ITEM, LOC, LOCP)

```
1.  [List Empty?] If START = NULL, then:
        Set LOC := NULL, LOCP := NULL and Return.
    [End of If Structure.]

2.  [ITEM in First node?] If INFO [START] = ITEM, then:
        Set LOC := START, and LOCP := NULL, and Return.
    [End of If Structure.]

3.  Set SAVE := START and PTR: = LINK [START].  [Initializes pointers]
4.  Repeat step 5 and 6 while PTR ≠ NULL.
5.          If INFO [PTR] = ITEM, then:
                    Set LOC: = PTR and LOCP: = SAVE, and Return.
            [End of If Structure.]

6.      Set SAVE := PTR and PTR: = LINK [PTR]. [Update pointers]
    [End of Step 4 Loop.]
7.  Set LOC := NULL. [Search Unsuccessful.]
8.  Return.
```

Plus and Minus of linked list of linked list

++No running out of memory, no wastage of memory.

++Insertion and deletion is easy, no physical movement.

--Traversal is difficult, we have to traverse (n-1) elements to go to the nth element.

--Wastage of memory in each element for storing the address of the next element.

Thank You