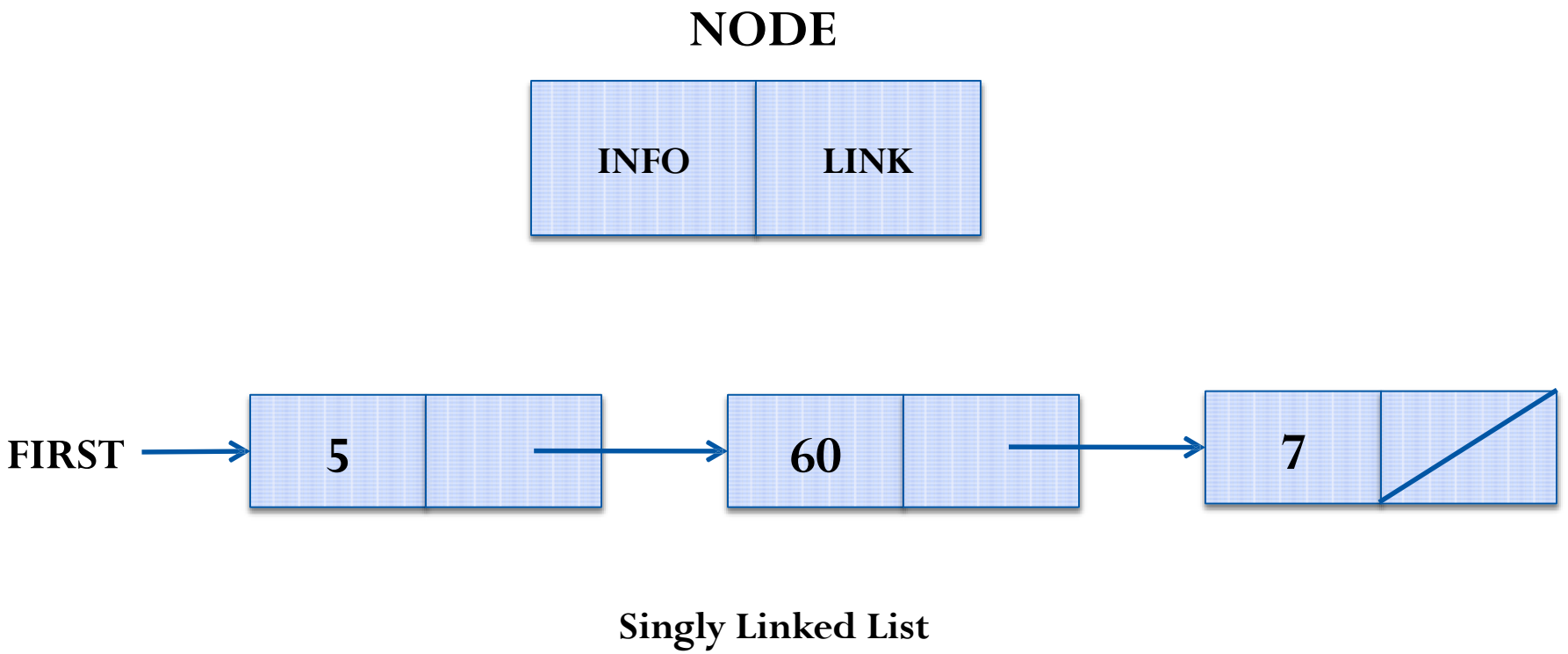
Linked List

Singly Linear Linked List

Node



Singly Linked List

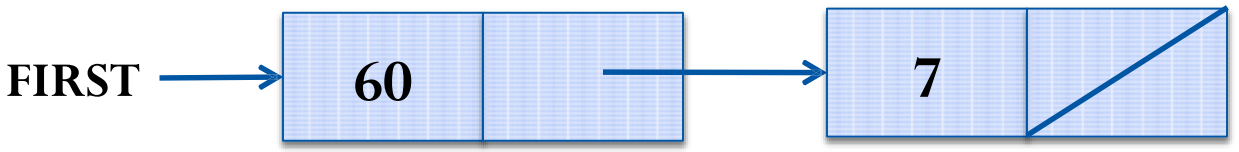
Insertion at beginning

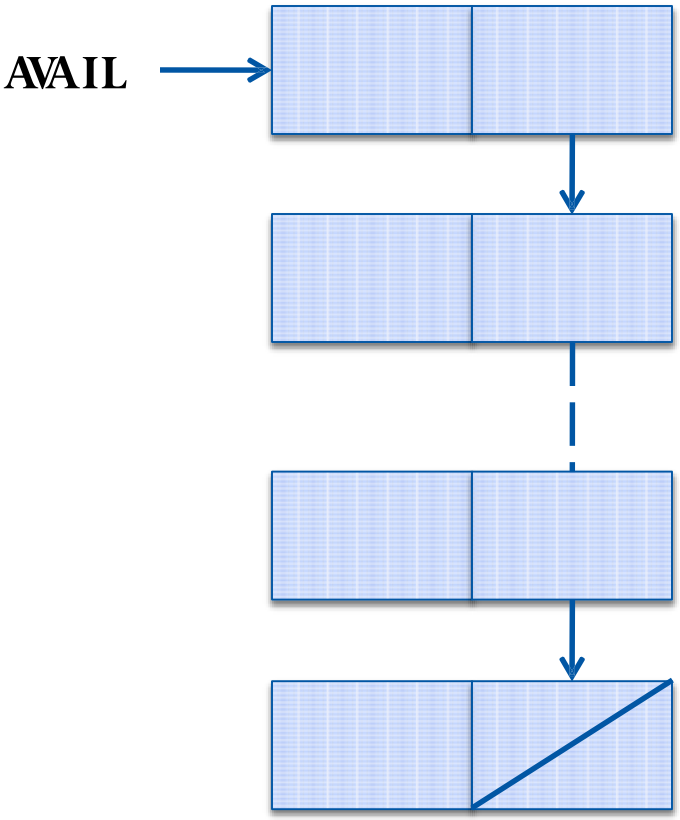
FIRST = INSERT (X, FIRST)

Function : INSERT (X, FIRST)

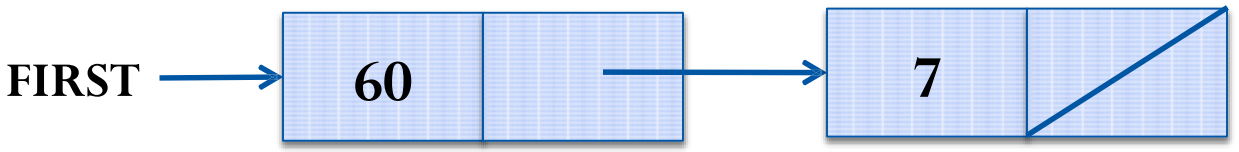
- X – a new element
- FIRST – a pointer to the first element of a linked linear list
- INFO – data field of the node
- LINK – a pointer to the next element in the list
- AVAIL – a pointer to the top element of the availability stack
- NEW – a temporary pointer variable

Insertion at beginning

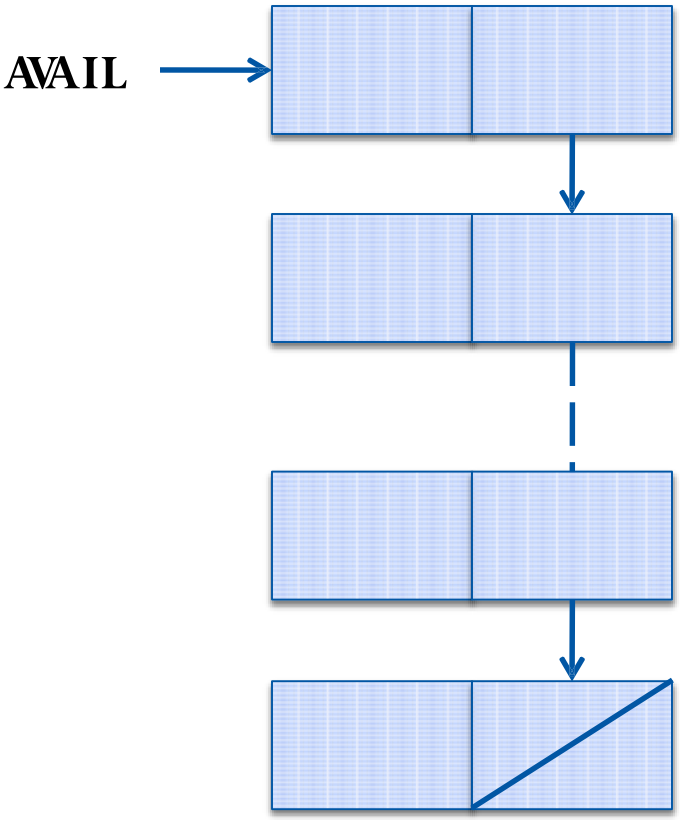




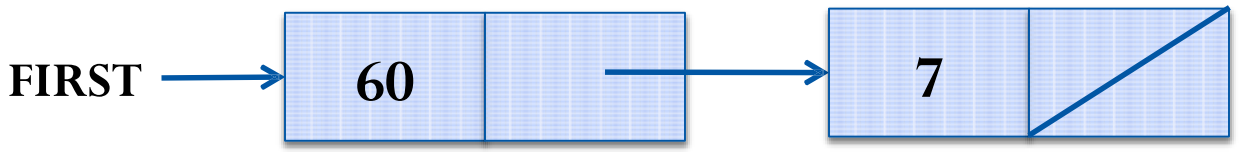
Insertion at beginning

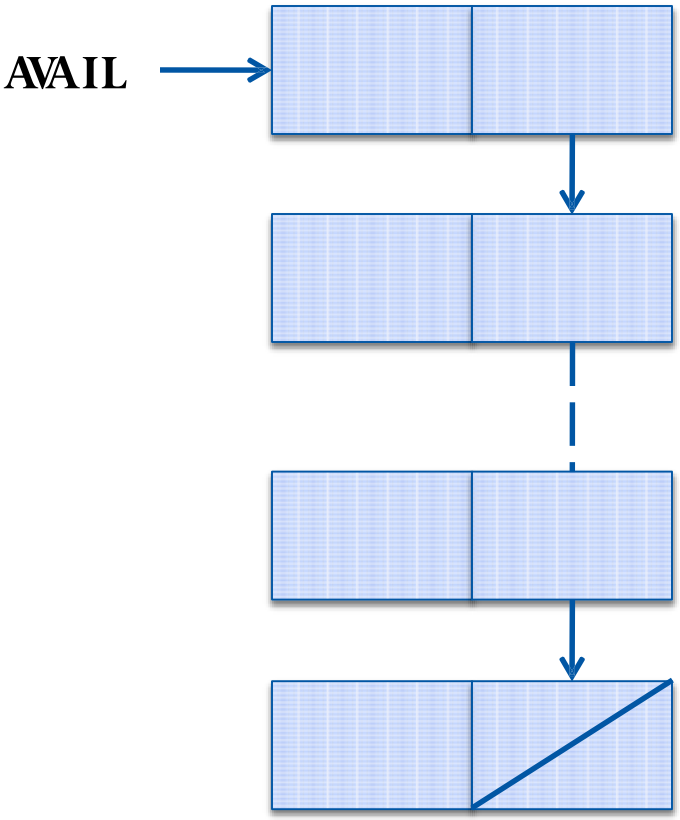


Function : INSERT (X, FIRST)



Insertion at beginning

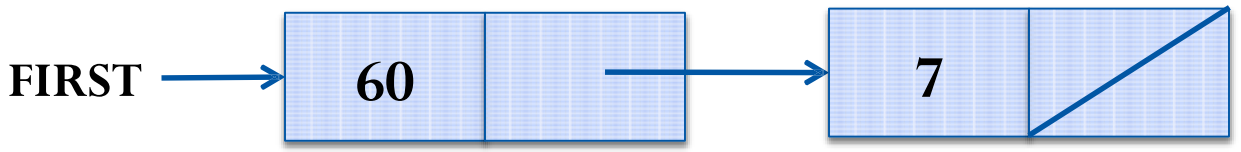


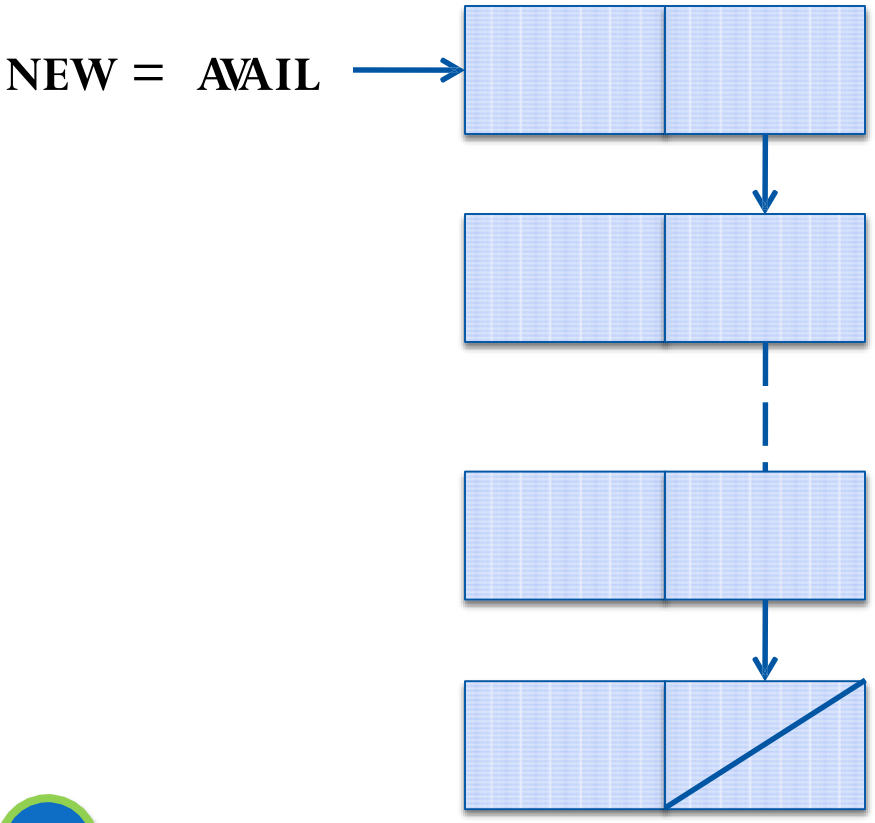


Function : INSERT (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   then Write('AVAIL UNDERFLOW')
   Return(FIRST)

Insertion at beginning

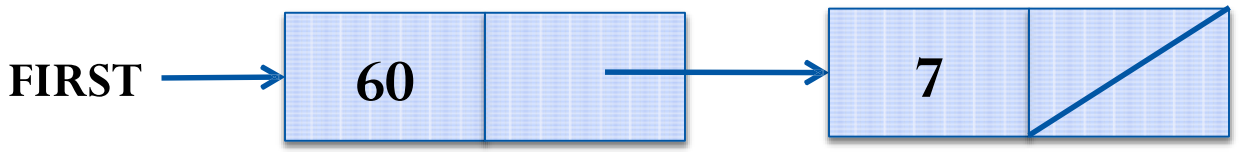


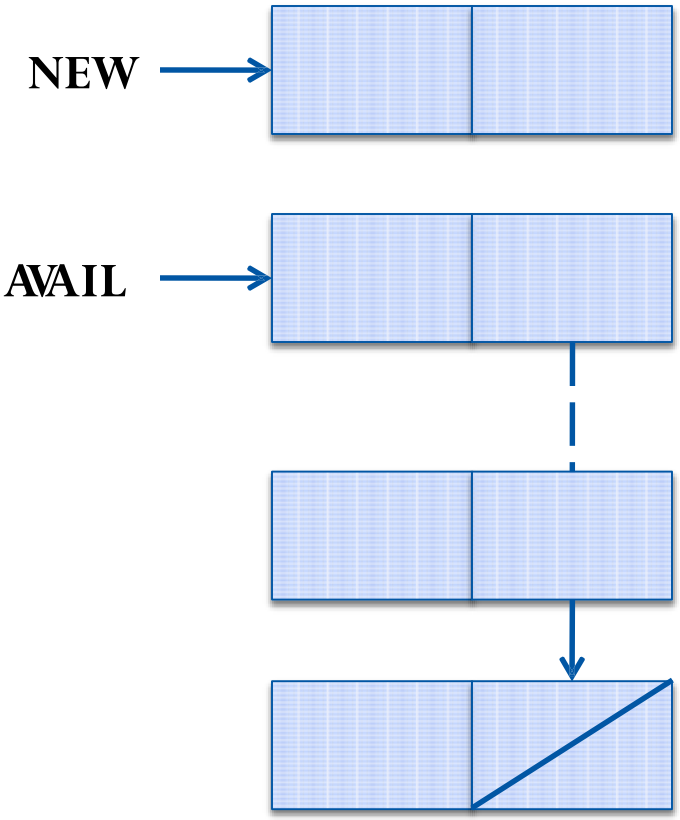


Function : INSERT (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   then Write('AVAIL UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node] NEW ← AVAIL

Insertion at beginning

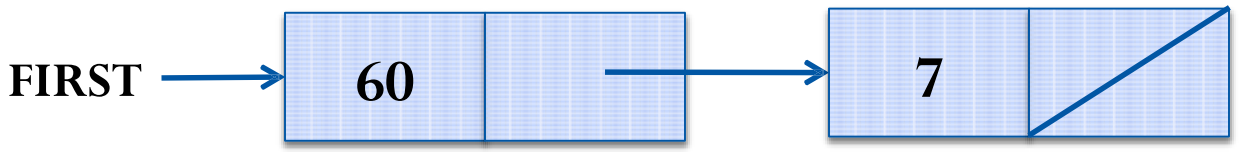


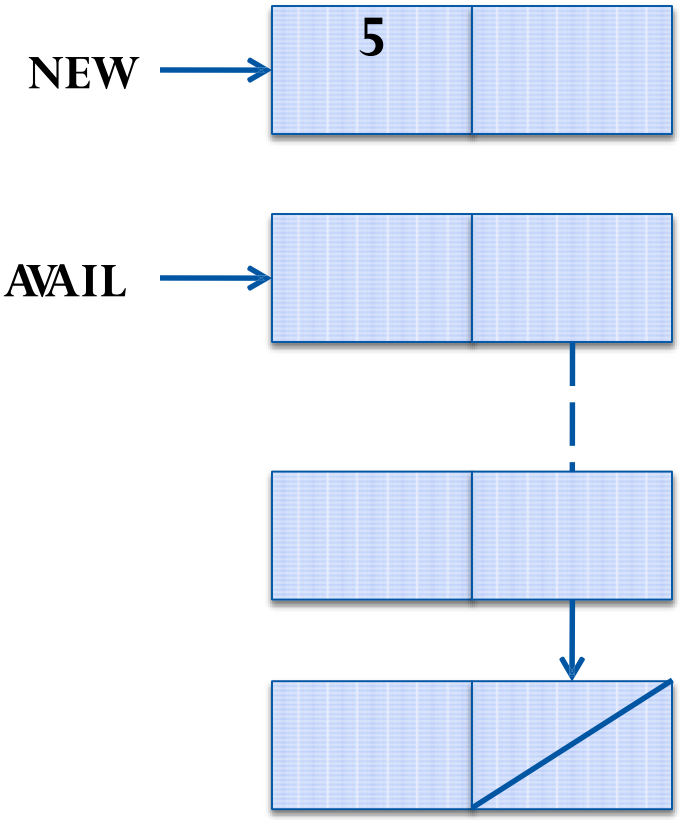


Function : INSERT (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   thenWrite('AVAIL UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node] NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)

Insertion at beginning

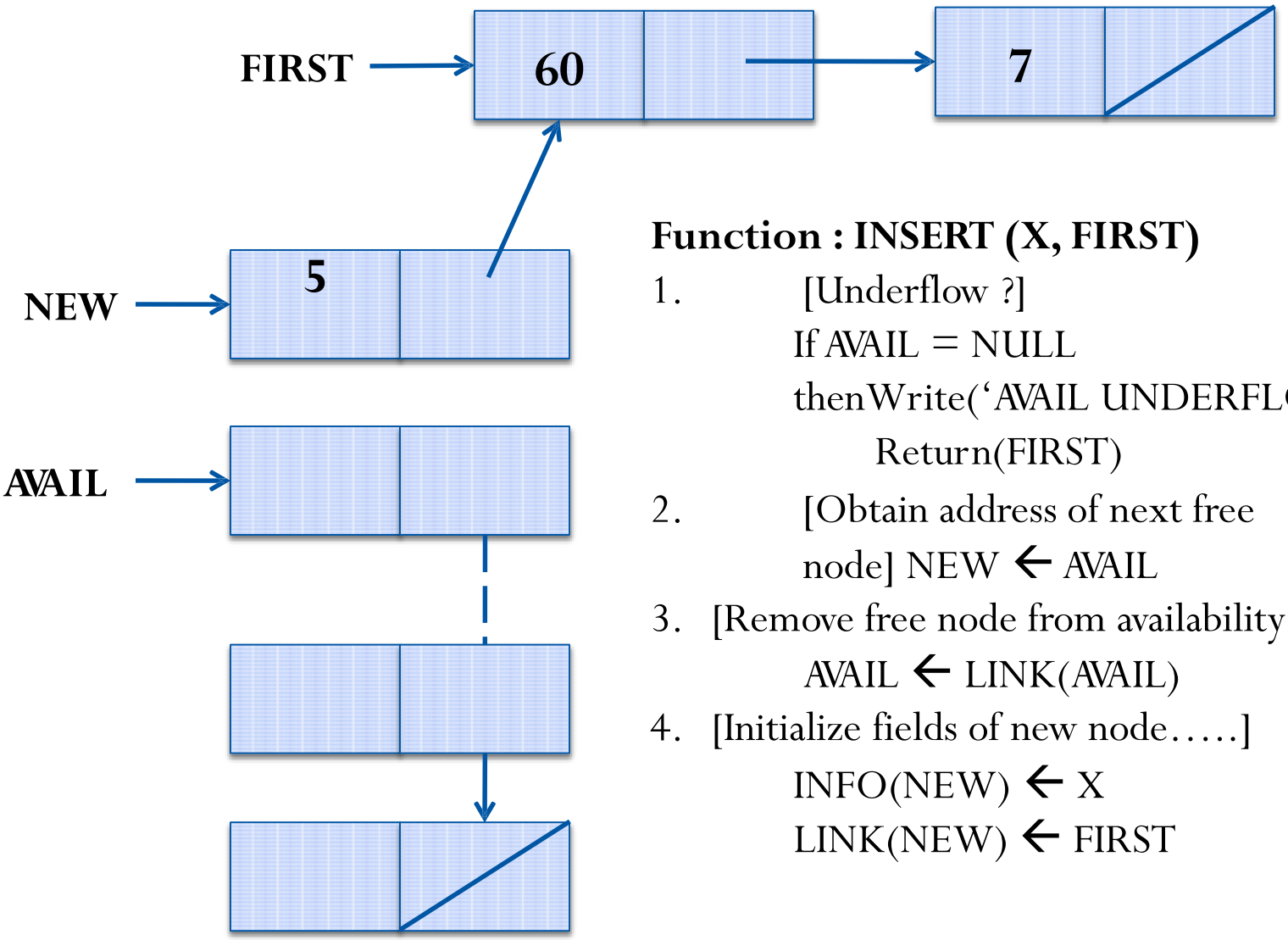




Function : INSERT (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   thenWrite('AVAIL UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node] NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node.....]
   INFO(NEW) ← X

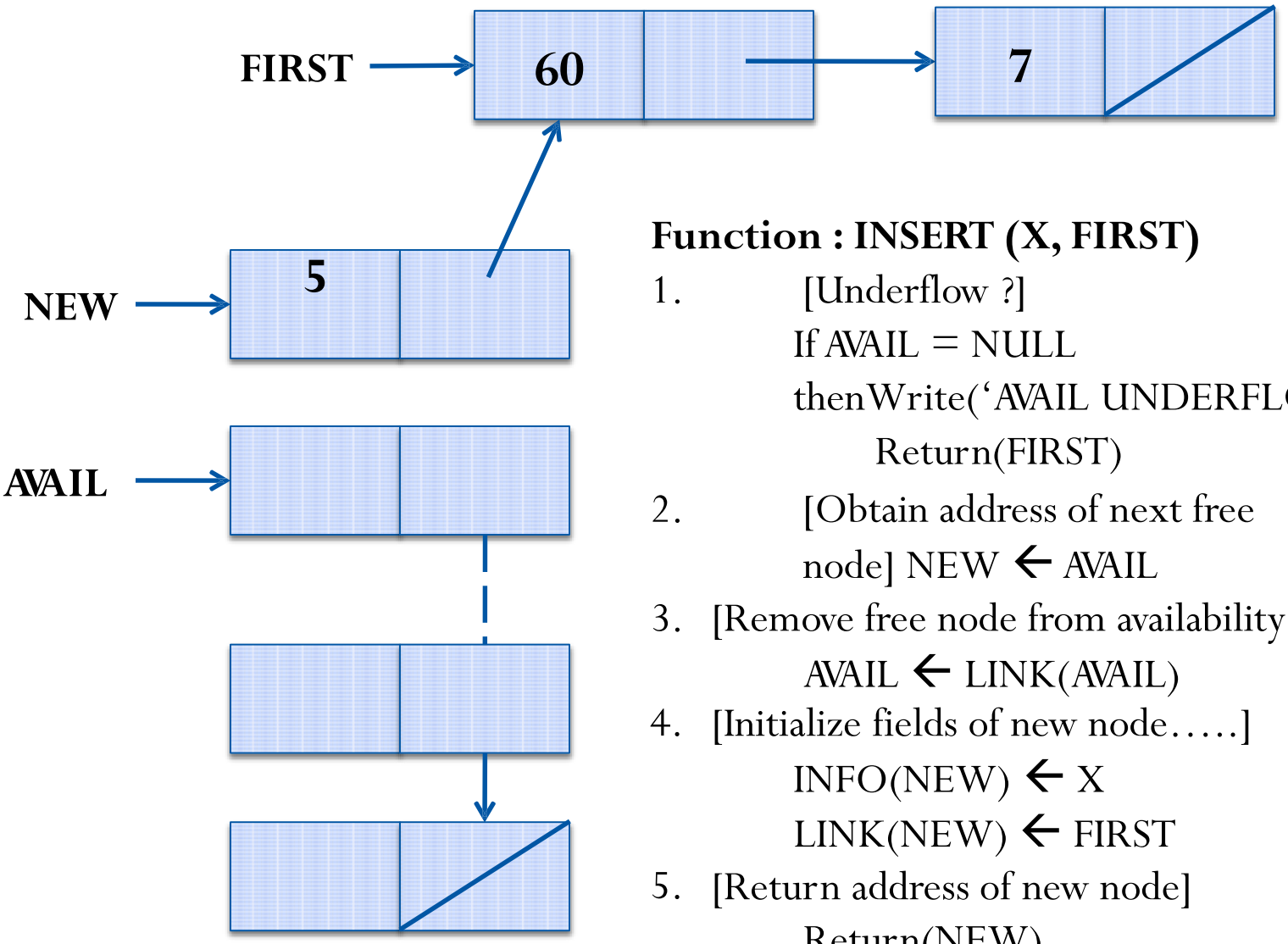
Insertion at beginning



Function : INSERT (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   thenWrite('AVAIL UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node] NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node.....]
   INFO(NEW) ← X
   LINK(NEW) ← FIRST

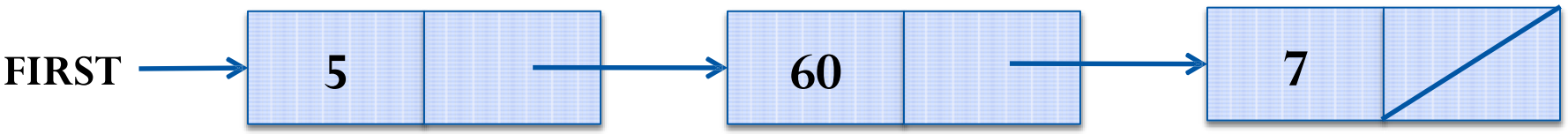
Insertion at beginning

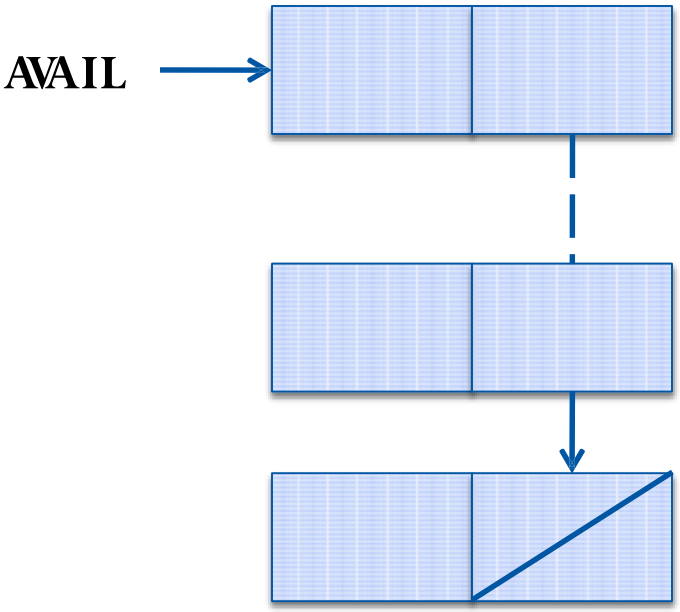


Function : INSERT (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   thenWrite('AVAIL UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node] NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node.....]
   INFO(NEW) ← X
   LINK(NEW) ← FIRST
5. [Return address of new node]
   Return(NEW)

Insertion at beginning





Function : INSERT (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   thenWrite('AVAIL UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node] NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node.....]
   INFO(NEW) ← X
   LINK(NEW) ← FIRST
5. [Return address of new node]
   Return(NEW)

Insertion at beginning

Function : INSERT (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   thenWrite('AVAILABILITY STACK UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node]
   NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node and its link to the list]
   INFO(NEW) ← X
   LINK(NEW) ← FIRST
5. [Return address of new node]
   Return(NEW)

Program snippet

```
struct node {
   int data;
   struct node *link;
};
struct node *insert (int x, struct  node * first)
{
   struct node *new;

   new=(struct node *)malloc(sizeof(node));
   new -> data = x;
   if(first = = NULL)
          new ->link = NULL;
   else
          new -> link = first;
   return(new);
}
```

Insertion at end

FIRST = INSEND(X, FIRST)

Function : INSEND (X, FIRST)

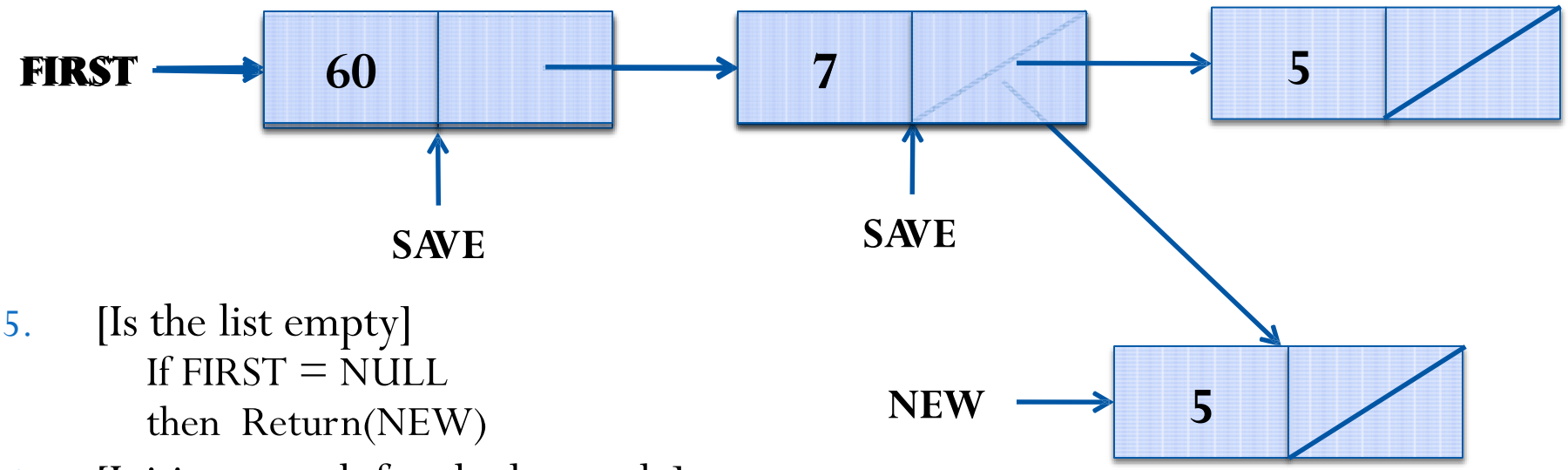
- X – a new element
- FIRST – a pointer to the first element of a linked linear list
- INFO – data field of the node
- LINK – a pointer to the next element in the list
- AVAIL – a pointer to the top element of the availability stack
- NEW – a temporary pointer variable
- SAVE – a temporary pointer variable

Insertion at end

Function : INSEND (X, FIRST)

1. [Underflow ?]
   If AVAIL = NULL
   thenWrite('AVAILABILITY STACK UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node]
   NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node and its link to the list]
   INFO(NEW) ← X
   LINK(NEW) ← NULL

Insertion at end



5. [Is the list empty]
   If FIRST = NULL
   then Return(NEW)
6. [Initiate search for the last node]
   SAVE ← FIRST
7. [Search for end of list]
   Repeat while LINK(SAVE) != NULL
   SAVE ← LINK(SAVE)
8. [Set LINK field of last node to NEW]
   LINK(SAVE) ← NEW
9. [Return first node pointer]
   Return(FIRST)

Insertion at end

Function : INSEND (X, FIRST) (……continue)

1. [Underflow ?]
   If AVAIL = NULL
   thenWrite('AVAILABILITY STACK UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node]
   NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node and its link to the list]
   INFO(NEW) ← X
   LINK(NEW) ← NULL
5. [Is the list empty]
   If FIRST = NULL
   then Return(NEW)

Insertion at end

6. [Initiate search for the last node]

   SAVE ← FIRST

7. [Search for end of list]

   Repeat while LINK(SAVE) != NULL

   SAVE ← LINK(SAVE)

8. [Set LINK field of last node to NEW]

   LINK(SAVE) ← NEW

9. [Return first node pointer]

   Return(FIRST)

Ordered Linked List

- Inserting a node into an ordered linear list

Algorithm

1. Remove a node from the availability stack
2. Set the fields of the new node
3. If the linked list is empty

   then return the address of the new node
4. If the node precedes all others in the list

   then insert the node at front of the list and return its address
5. Obtain the next node in the linked list
6. Insert the new node in the list and return address of its first node

Insertion in ordered linked list

FIRST = INSORD(X, FIRST)

Function : INSORD (X, FIRST)

- X – a new element
- FIRST – a pointer to the first element of a linked linear list
- INFO – data field of the node
- LINK – a pointer to the next element in the list
- AVAIL – a pointer to the top element of the availability stack
- NEW – a temporary pointer variable
- SAVE – a temporary pointer variable

Insertion in ordered linked list

- **Function : INSORD (X, FIRST)**

1. [Underflow ?]
   If AVAIL = NULL
   then Write('AVAILABILITY STACK UNDERFLOW')
   Return(FIRST)
2. [Obtain address of next free node]
   NEW ← AVAIL
3. [Remove free node from availability stack]
   AVAIL ← LINK(AVAIL)
4. [Copy information contents into new node]
   INFO(NEW) ← X
5. [Is the list empty ?]
   If FIRST = NULL
   then LINK(NEW) ← NULL
   Return(NEW)

Insertion in ordered linked list

6. [Does the new node precede all others in the list ?]
   If INFO(NEW) <= INFO(FIRST)
   then LINK(NEW) ← FIRST
   Return(NEW)
7. [Initialize the temporary pointer]
   SAVE ← FIRST
8. [Search for predecessor of new node]
   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. [Return first node pointer]
    Return(FIRST)

Deleting a node from linked linear list

Algorithm

1. If the linked list is empty

   then write underflow and return
2. Repeat step 3 while the end of the list has not been reached and the node has not been found
3. Obtain the next node in the list and record its predecessor node
4. If the end of the list has been reached

   then write node not found and return
5. Delete the node from the list
6. Return the node to the availability area

Deleting a node from linked linear list

DELETE(X, FIRST)

Procedure : DELETE (X, FIRST)

- X – the address of a node to be deleted
- FIRST – a pointer to the first element of a linked linear list
- INFO – data field of the node
- LINK – a pointer to the next element in the list
- TEMP – a pointer used to find the desired node
- PRED – pointer keeps track of the predecessor of TEMP

Deleting a node from linked linear list

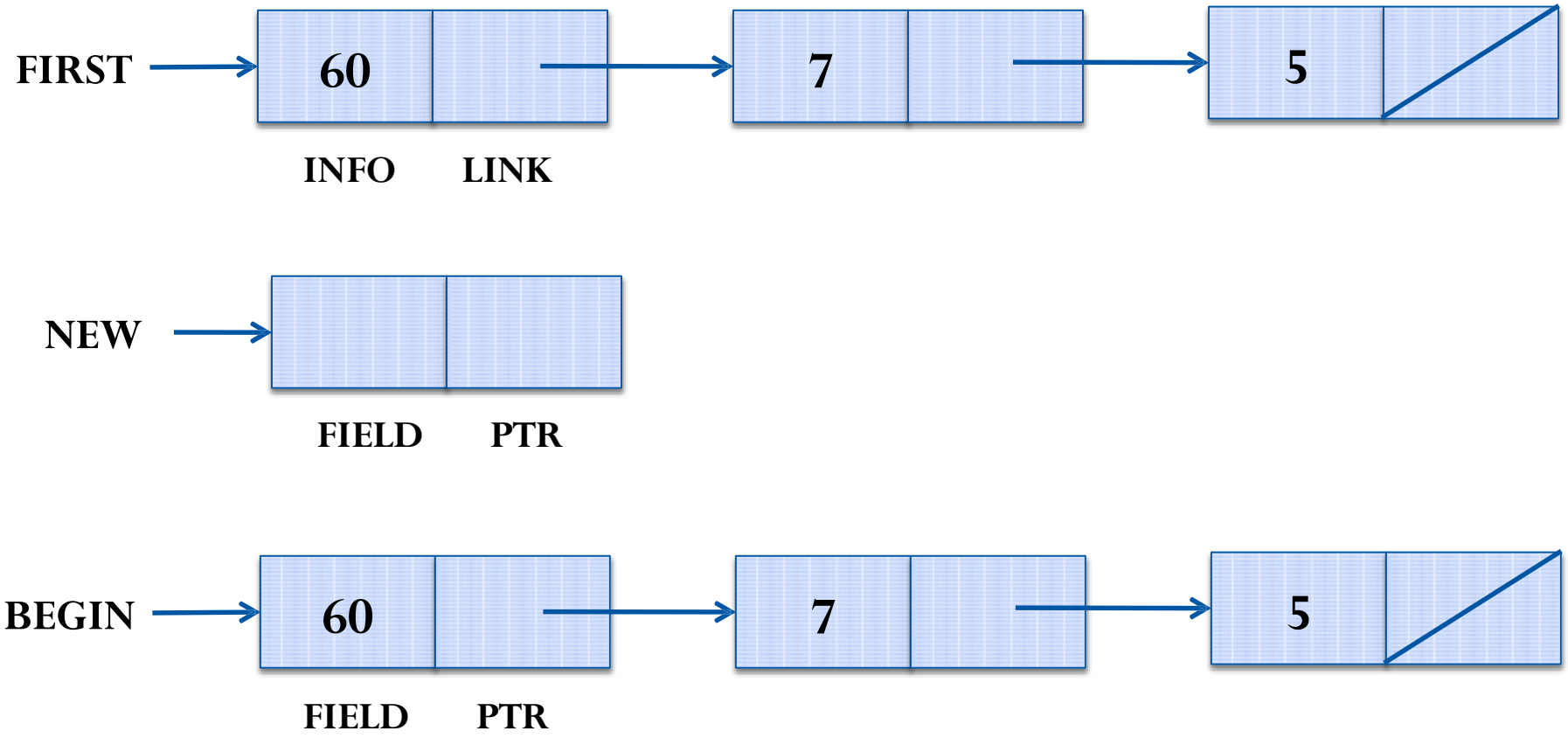
- **Procedure : DELETE (X, FIRST)**

1. [Empty list ?]
   If FIRST = NULL
   thenWrite('UNDERFLOW')
   Return
2. [Initialize search for X
   TEMP ← FIRST
3. [Find X]
   Repeat thru step5 whileTEMP != X and LINK(TEMP) != NULL
4. [Update predecessor marker]
   PRED ← TEMP
5. [Move to next node]
   TEMP ← LINK(TEMP)

Deleting a node from linked linear list

6. [End of the list ?]
   If TEMP != X
   then Write ('NODE NOT FOUND')
   Return
7. [Delete X]
   If X = FIRST (Is X the first node ?)
   then FIRST ← LINK(FIRST)
   else LINK(PRED) ← LINK(X)
8. [Return node to availability area
   LINK(X) ← AVAIL
   AVAIL ← X
   Return

Copy a linked linear list



Copy a linked linear list

BEGIN=COPY(FIRST)

Function : COPY (FIRST)

- FIRST – a pointer to the first element of a linked linear list
- INFO – data field of the node
- LINK – a pointer to the next element in the list
- BEGIN – the address of the first element in newly created list
- FIELD – data field of node in newly created list
- PTR – a pointer to the next element in the newly created list
- NEW, SAVE – pointer variables
- PRED – pointer pointing to the predecessor to the SAVE
- AVAIL – a pointer pointing to the top of availability stack

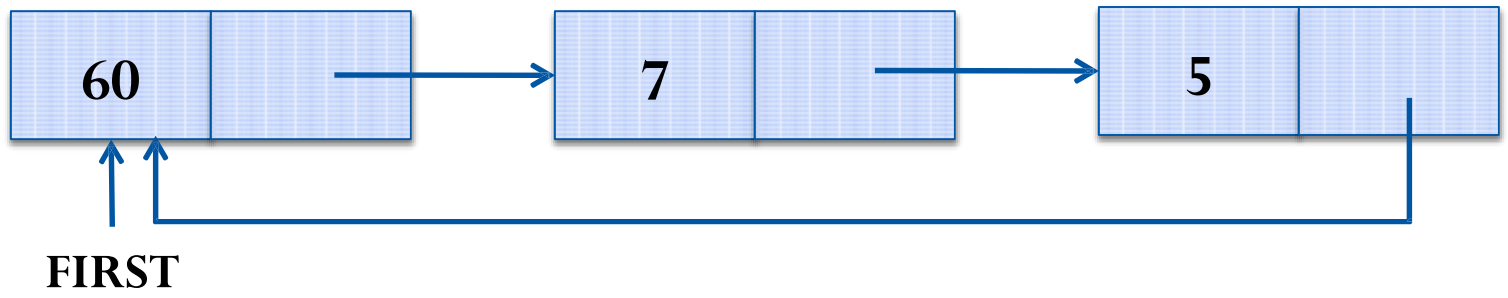
Copy a linked linear list

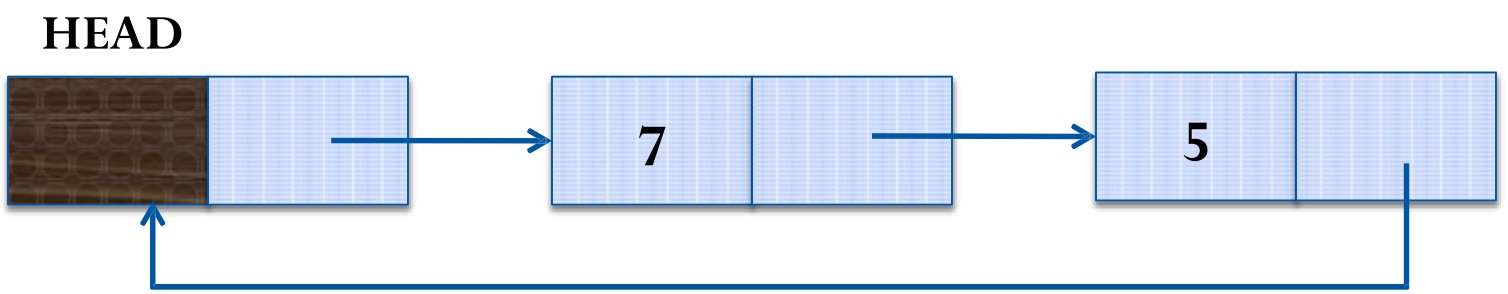
1. [Empty list ?]
   If FIRST = NULL
   then Return(NULL)
2. [Copy first node]
   If AVAIL = NULL
   thenWrite ('AVAILABILITY STACK UNDERFLOW')
   Return(0)
   else NEW ← AVAIL
   AVAIL ← LINK (AVAIL)
   FIELD (NEW) ← INFO(FIRST)
   BEGIN ← NEW
3. [Initialize traversal]
   SAVE ← FIRST
4. [Move to the next node if not at the end of the list]
   Repeat thru step 6 while LINK(SAVE) != NULL

Copy a linked linear list

5. [Update predecessor and save pointers]
   PRED ← NEW
   SAVE ← LINK (SAVE)
6. [Copy node]
   If AVAIL = NULL
   thenWrite ('AVAILABILITY STACK UNDERFLOW')
   Return(0)
   else NEW ← AVAIL
   AVAIL ← LINK(AVAIL)
   FIELD(NEW) ← INFO(SAVE)
   PTR(PRED) ← NEW
7. [Set link of last node and return]
   PTR(NEW) ← NULL
   Return (BEGIN)

Circularly Linked Linear Lists





Circularly Linked Linear Lists

Advantages:

- Every node is accessible from given node
- Deletion operation is easy

Disadvantages:

- Without some care in processing, it is possible to get into an infinite loop

Stack and Queue using Linked List

Doubly Linked Linear List

Insertion in Doubly Linked Linear List

Procedure: DOUBINS(L, R, M, X)

- L – pointer giving left-most node address
- R – pointer giving right-most node address
- NEW – new node to be inserted in list
- LPTR, RPTR – left and right link of node
- INFO – information part of node
- M – a pointer variable (insertion is performed at left of M)
- X – information of new node

Insertion in Doubly Linked Linear List

1. [Obtain new node from availability stack]

   NEW ← NODE

2. [Copy information field]

   INFO(NEW) ← X

3. [Insertion into an empty list ?]

   If R= NULL

   then LPTR(NEW) ← RPTR(NEW) ← NULL

   L ← R ← NEW

   Return

Insertion in Doubly Linked Linear List

4. [Left-most insertion ?]

   If M=L

   then LPTR(NEW) ← NULL

   RPTR(NEW) ← M

   LPTR(M) ← NEW

   L← NEW

   Return

5. [Insert in middle]

   LPTR(NEW) ← LPTR(M)

   RPTR(NEW) ← M

   LPTR(M) ← NEW

   RPTR(LPTR(NEW)) ← NEW

   Return

Deletion in Doubly Linked Linear List

Procedure: DOUBDEL(L, R, OLD)

- L – pointer giving left-most node address
- R – pointer giving right-most node address
- LPTR, RPTR – left and right link of node
- OLD – address of node to be deleted

Deletion in Doubly Linked Linear List

1. [Underflow ?]
   If R=NULL
   thenWrite('UNDERFLOW')
   Return
2. [Delete node]
   If L=R (Single node in list)
   then L← R ← NULL
   else If OLD=L (Left-most node being deleted)
   then L ← RPTR(L)
   LPTR(L) ←NULL
   else If OLD = R (Right-most node being deleted)
   then R← LPTR(R)
   RPTR(R) ← NULL
   else RPTR(LPTR(OLD)) ← RPTR(OLD)
   LRPTR(RPTR(OLD)) ← LPTR(OLD)

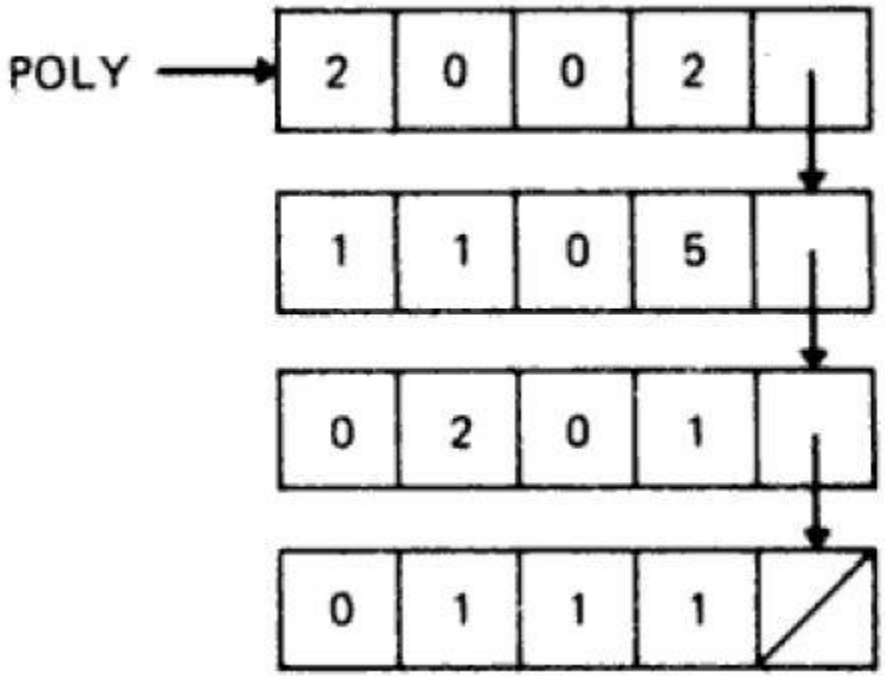
Deletion in Doubly Linked Linear List

3. [Return deleted node]

   Restore (OLD)

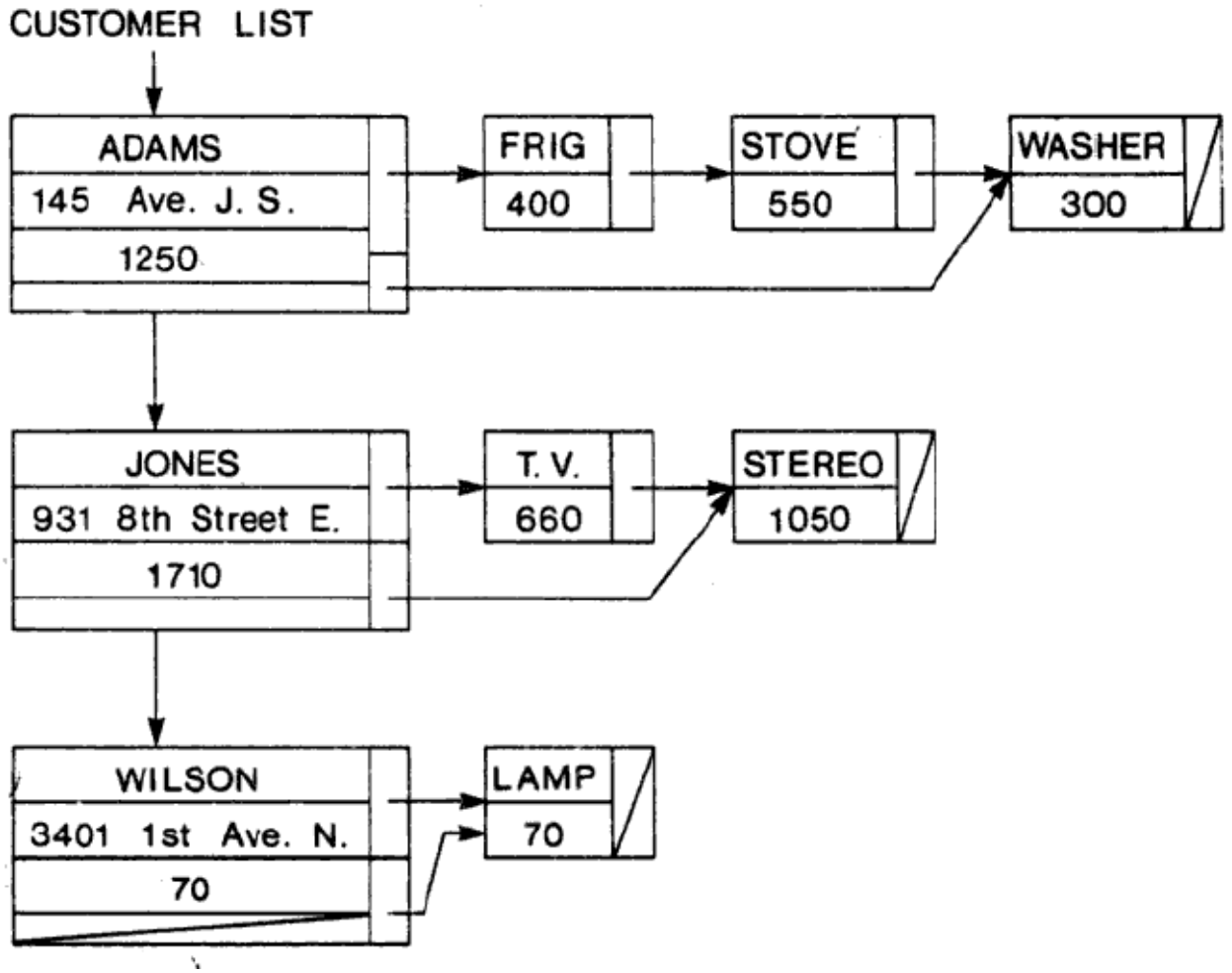
   Return

Polynomial



$2x^2 + 5xy + y^2 + yz$

Customer List



Application of Linked List

1. Linked lists are used as a building block for many other data structures such as **Stack** and **Queue**. Stack and Queue can be represented dynamically using linked list.
2. Linked list is used for representation and manipulation of **Polynomial Equation**.
3. Linked list is widely used by compiler for the construction and maintenance of **Symbol Table**.
4. Linked list is also used to implement **Associative Arrays**.
5. Linked list is also used to implement **Sparse Matrix**.