Linked Lists

Instructors:
Md Nazrul Islam Mondal &
Rizoan Toufiq
Department of Computer Science & Engineering
Rajshahi University of Engineering &
Technology Rajshahi-6204

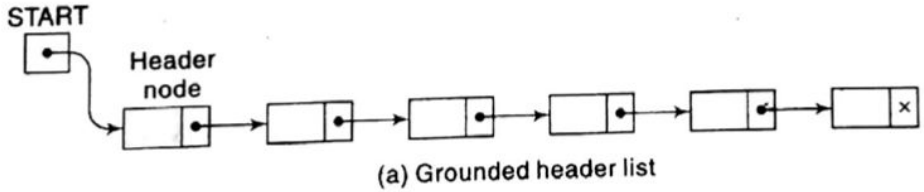
Outline

- Introduction
- Linked List
- Representation of Linked Lists in Memory
- Traversing a Linked List
- Searching a Linked List
- Memory Allocation; Garbage Collection
- Insertion into a Linked List
- Deletion from a Linked List
- **Header Linked List**
- **Two Way Lists**

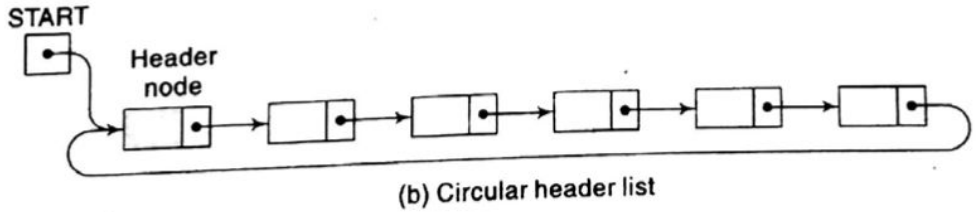
Header Linked List

Header Linked List

- Contains a special node, called the **header node**, at the beginning of the list.
  - A **grounded header** list is a header list where the last node contains the null pointer.
  - A **circular header** list is a header list where the last node points back to the header node.



(a) Grounded header list

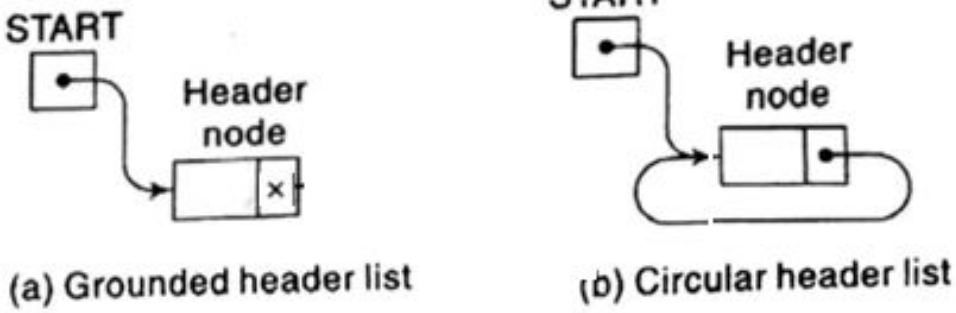


(b) Circular header list

Fig. 5.29

Header Linked List

- **Grounded header list** is empty i.e. LINK[START]=NULL
- **Circular header list** is empty i.e. LINK[START]=START

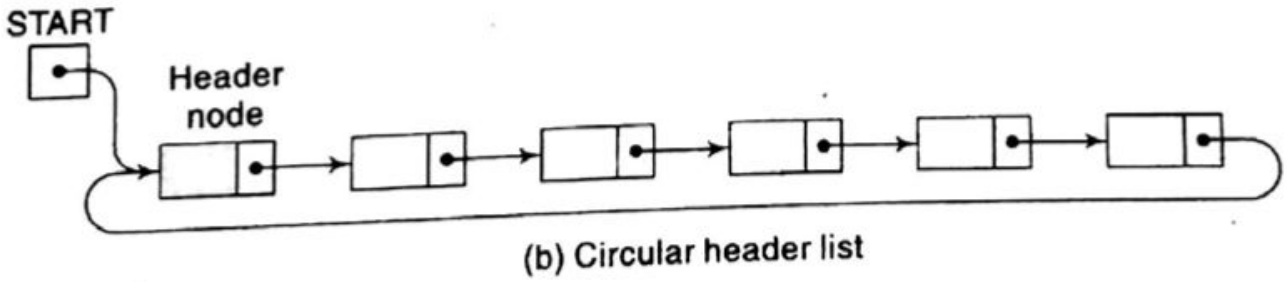


(a) Grounded header list

(b) Circular header list

Header Linked List

- REMARK:
  - Our header lists will always be circular.
  - The header node acts as a sentinel indicating the end of the list. (No data in header node)



(b) Circular header list

Fig. 5.29

Header Linked List

- **Circular header list:** Start traversing from LINK[START]

Algorithm 5.11: (Traversing a Circular Header List) Let LIST be a circular header list in memory. This algorithm traverses LIST, applying an operation PROCESS to each node of LIST.

1. Set PTR := LINK[START]. [Initializes the pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ START:
3. Apply PROCESS to INFO[PTR].
4. Set PTR := LINK[PTR]. [PTR now points to the next node.]

[End of Step 2 loop.]

5. Exit.

Header Linked List

Algorithm 5.12: SRCHHL(INFO, LINK, START, ITEM, LOC)
LIST is a circular header list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC = NULL.

1. Set PTR := LINK[START].
2. Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START:
   Set PTR :=LINK[PTR]. [PTR now points to the next node.]
   [End of loop.]
3. If INFO[PTR] = ITEM, then:
   Set LOC := PTR.
   Else:
   Set LOC := NULL.
   [End of If structure.]
4. Exit.

Header Linked List

Procedure 5.13: FINDBHL(INFO, LINK, START, ITEM, LOC, LOCP)

1. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]
2. Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START.
   Set SAVE := PTR and PTR:= LINK[PTR]. [Updates pointers.]
   [End of loop.]
3. If INFO[PTR] = ITEM, then:
   Set LOC := PTR and LOCP := SAVE.
   Else:
   Set LOC := NULL and LOCP := SAVE.
   [End of If structure.]
4. Exit.

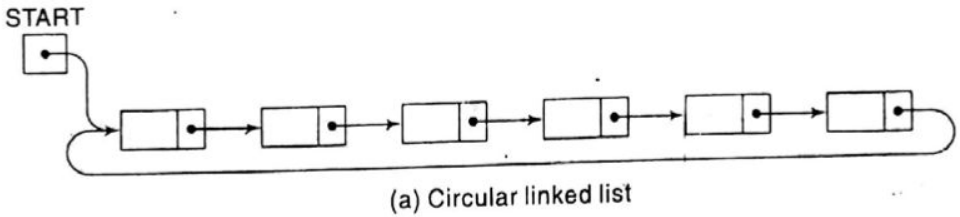
Header Linked List

Algorithm 5.14: DELLOCHL(INFO, LINK, START, AVAIL, ITEM)

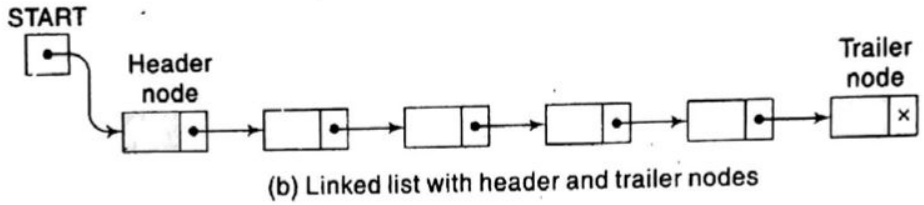
1. [Use Procedure 5.13 to find the location of N and its preceding node.]
   Call LINDBHL(INFO, LINK, START, ITEM, LOC, LOCP).
2. If LOC = NULL, then: Write: ITEM not in list, and Exit.
3. Set LINK[LOCP] := LINK[LOC]. [Deletes node.]
4. [Return deleted node to the AVAIL list.]
   Set LINK[LOC]:= AVAIL and AVAIL:= LOC.
5. Exit.

Header Linked List

- Other two version of linked list
  - A linked list whose last node points back to the first node instead of containing the null pointer, called a **circular list**.
  - A linked list which contains both a special header node at the beginning of the list and a special trailer node at the end of the list.



(a) Circular linked list



(b) Linked list with header and trailer nodes

Fig. 5.31

Two Way Lists

Two Way Lists

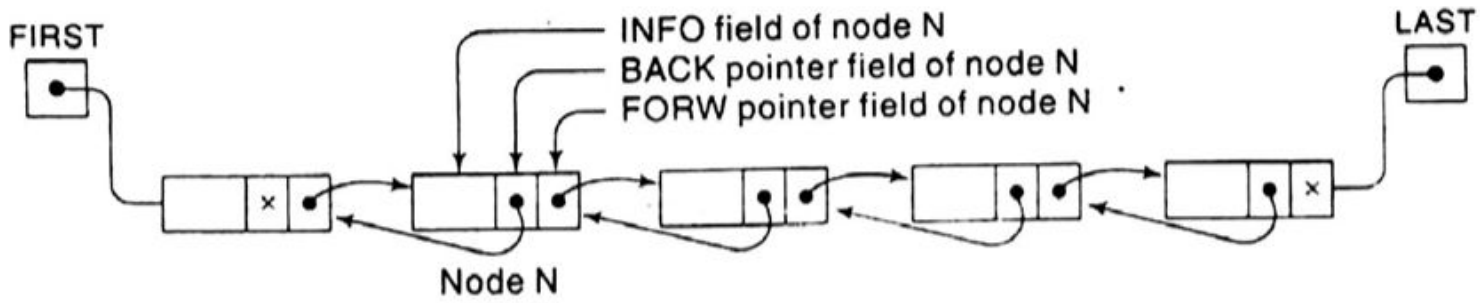
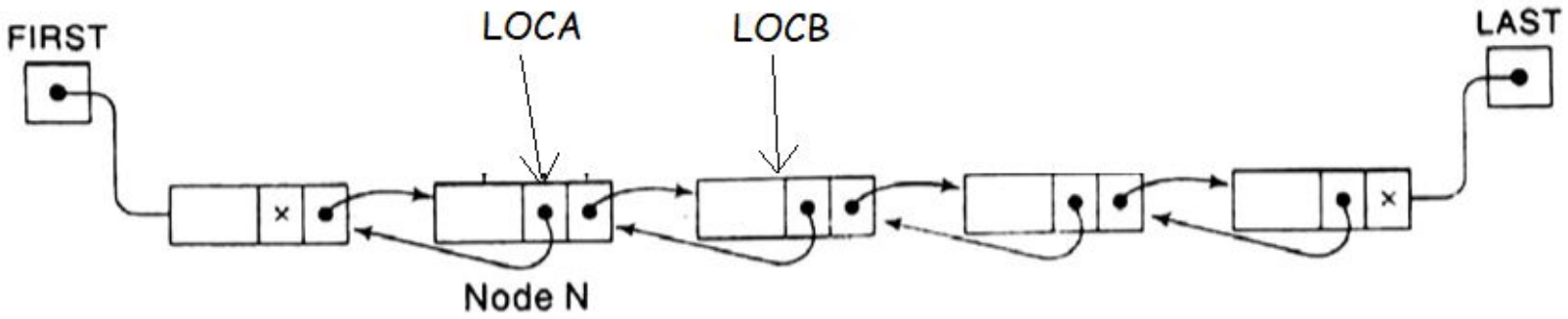


Fig. 5.33 *Two-way List*

Two Way Lists



Pointer Property:

FORW[LOCA] = LOCB **if and only if** BACK[LOCB]=LOCA

Two Way Lists

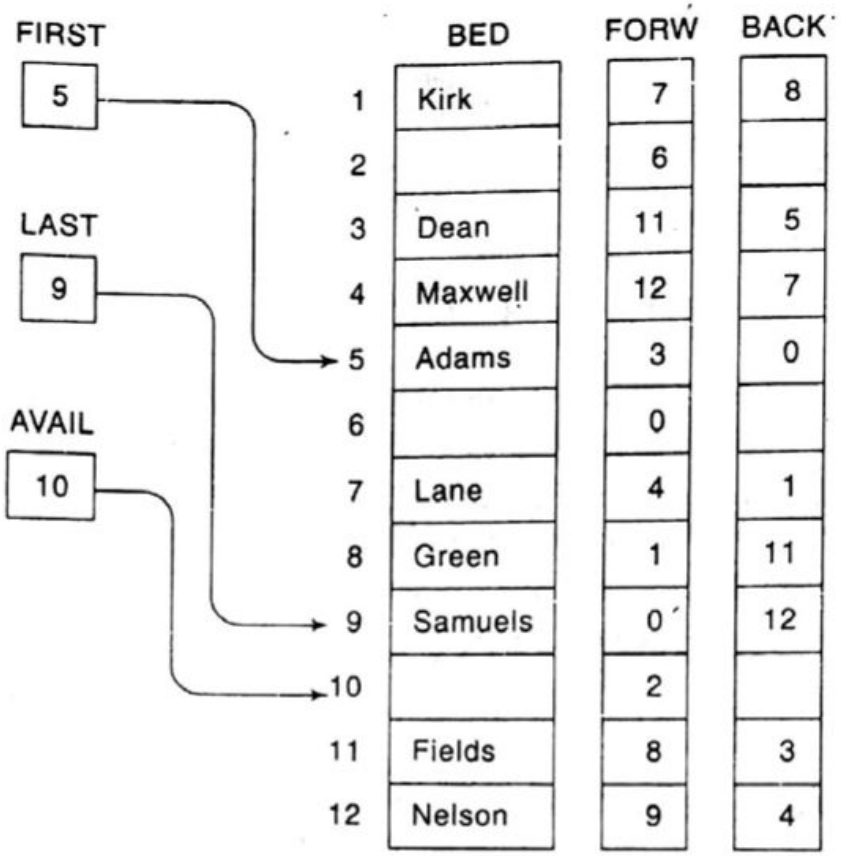


Fig. 5.34

Two Way Lists
(Traversing and Searching)

- Traversing:
  - Similar as one-way list or a header list
- Searching
  - Search for ITEM in the backward direction
  - For example
    - suppose LIST is a list of name sorted alphabetically.
    - If ITEM = Smith, then we would search LIST in the backward direction.
    - IF ITEM = Davis, then we would search LIST in the Forward direction.

Two Way Lists (Deleting)

- Deleting:
  - Given the location LOC of a node N in LIST.
  - Want to delete N from the list.
  - Consider, LIST is a two-way circular header list.

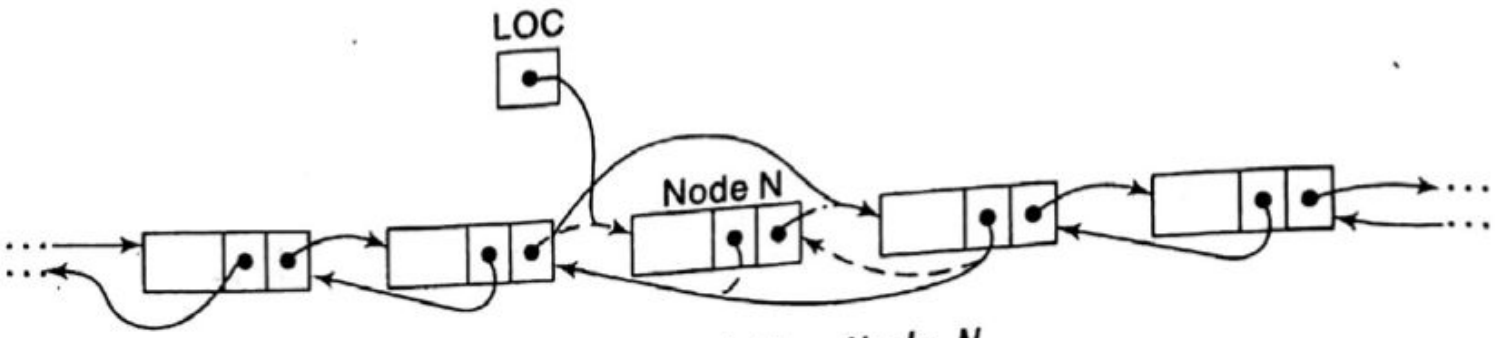


Fig. 5.37 *Deleting Node N*

Two Way Lists (Deleting)

- Deleting:

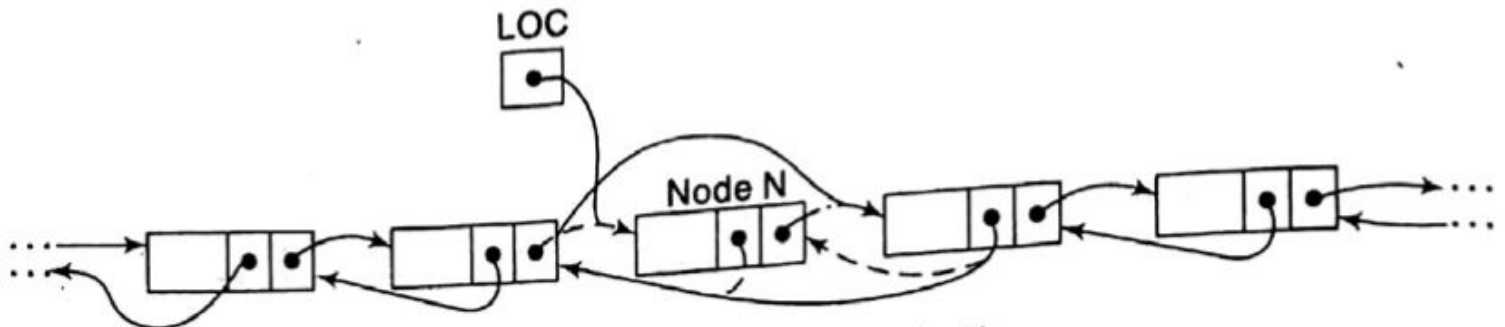


Fig. 5.37 *Deleting Node N*

FORW[BACK[LOC]]:=FORW[LOC]

BACK[FORW[LOC]]:=BACK[LOC]

Two Way Lists (Deleting)

- Deleting:

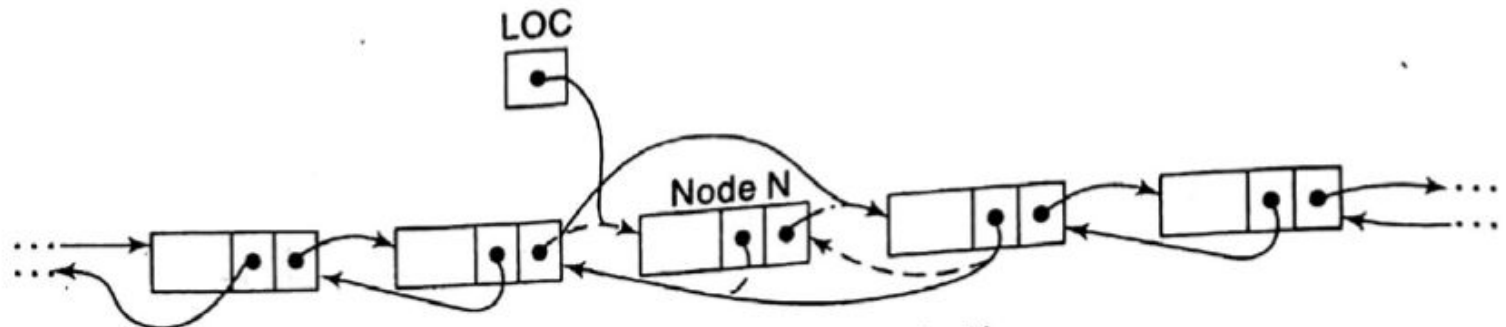


Fig. 5.37 *Deleting Node N*

FORW[LOC]:=AVAIL

AVAIL:=LOC

Two Way Lists (Deleting)

Algorithm 5.15: DELTWL(INFO, FORW, BACK, START, AVAIL, LOC)

1. [Delete node.]
   Set FORW[BACK[LOC]] := FORW[LOC] and
   BACK[FORW[LOC]] := BACK[LOC].
2. [Return node to AVAIL list.]
   Set FORW[LOC] := AVAIL and AVAIL := LOC.
3. Exit.

Two Way Lists (Inserting)

- Inserting:
  - Given the location LOCA and LOCB of adjacent nodes A and B in LIST.
  - Insert a given ITEM of information between nodes A and B.
  - First we remove the first node N from the AVAIL list, using the variable NEW to keep track of its location, and then we copy the data ITEM into the node N.

```
NEW:=AVAIL,
AVAIL:=FORW[AVAIL],
INFO[NEW]:=ITEM
```

Two Way Lists (Inserting)

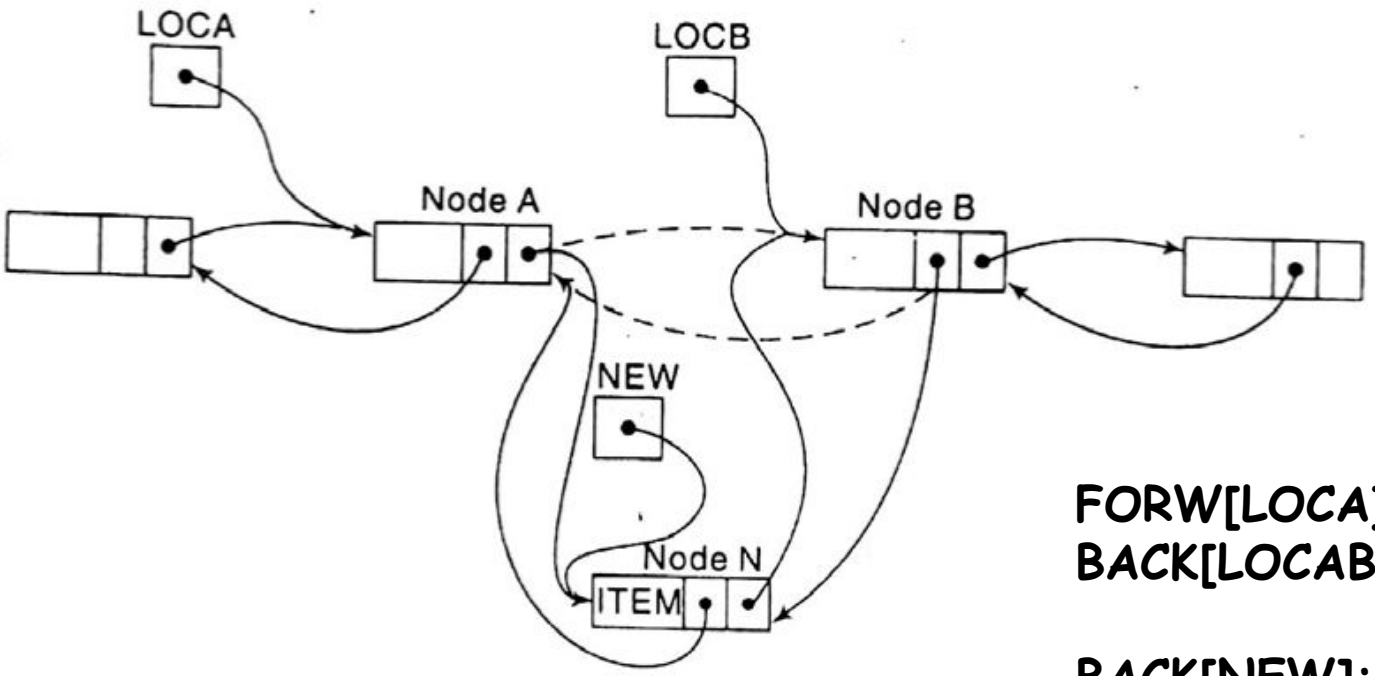


Fig. 5.38 *Inserting Node N*

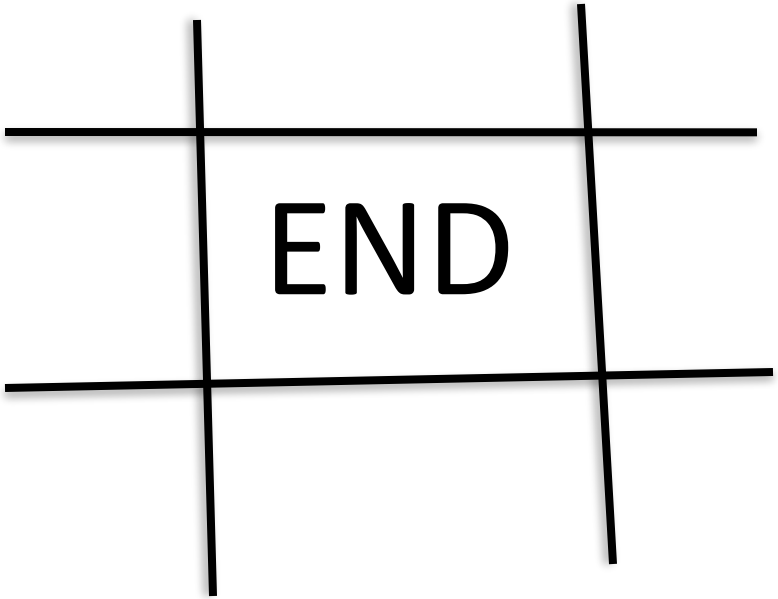
FORW[LOCA]:=NEW
BACK[LOCAB]:=NEW

BACK[NEW]:=LOCA
FORW[NEW]:=LOCB

Two Way Lists (Inserting)

Algorithm 5.16: INSTWL(INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove node from AVAIL list and copy new data into node.]
   Set NEW := AVAIL, AVAIL := FORW[AVAIL], INFO[NEW] := ITEM.
3. [Insert node into list.]
   Set FORW[LOCA] := NEW, FORW[NEW] := LOCB,
   BACK[LOCB] := NEW, BACK[NEW] := LOCA.
4. Exit.


END