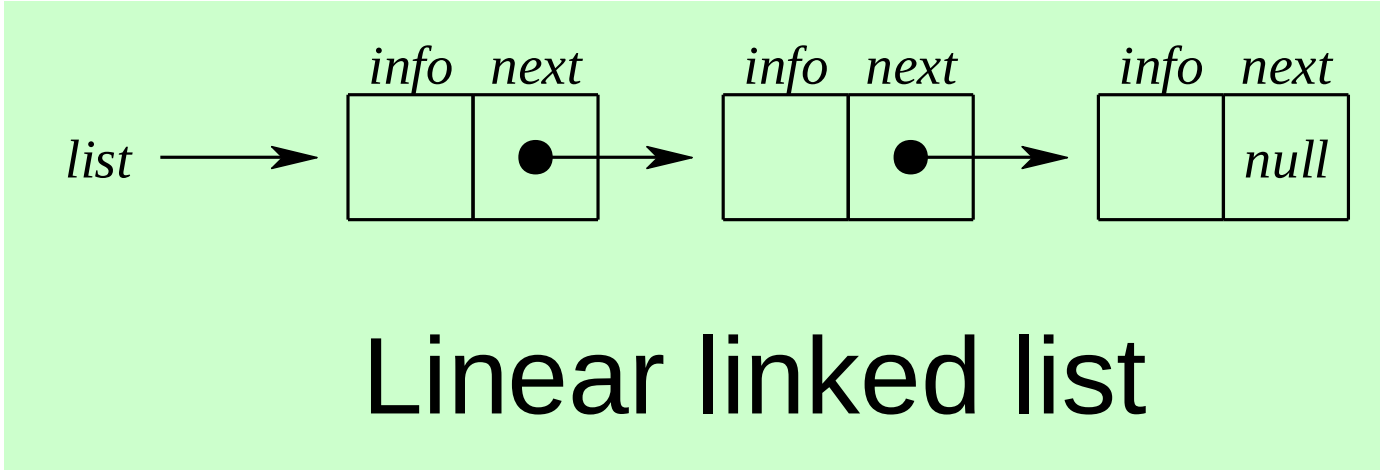
Linked-list

Introduction

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

Linked-list



Linear linked list

Traversing linked-list

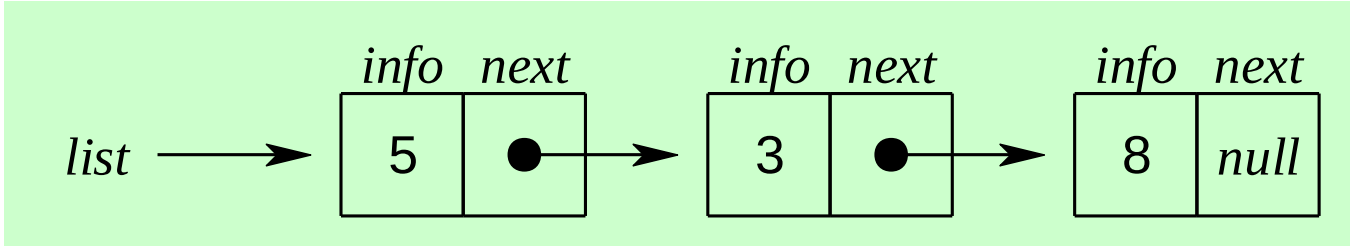
1. Set Ptr = Start **[Initializes pointer Ptr]**
2. Repeat step 3 and 4 while Ptr != NULL
3. Apply Process to Ptr→info
4. Set Ptr = Ptr→link **[Ptr now points to next node]**

   [End of step 2 loop]
5. Exit

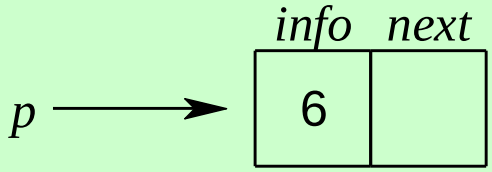
Searching a linked-list

1. Set Ptr = Start **[Initializes pointer Ptr]**
2. Repeat step 3 while Ptr != NULL
3. If ITEM == Ptr→info , then
   Set Loc = Ptr and Exit
   Else
   Set Ptr = Ptr→link **[Ptr now points to next node]**
   [End of If structure]
   [End of step 2 loop]
4. **[Search is unsuccessful]** Set Loc = NULL
5. Exit

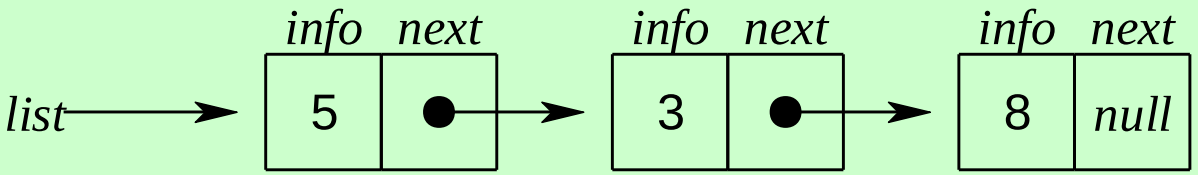
Adding an Element to the front of a Linked List



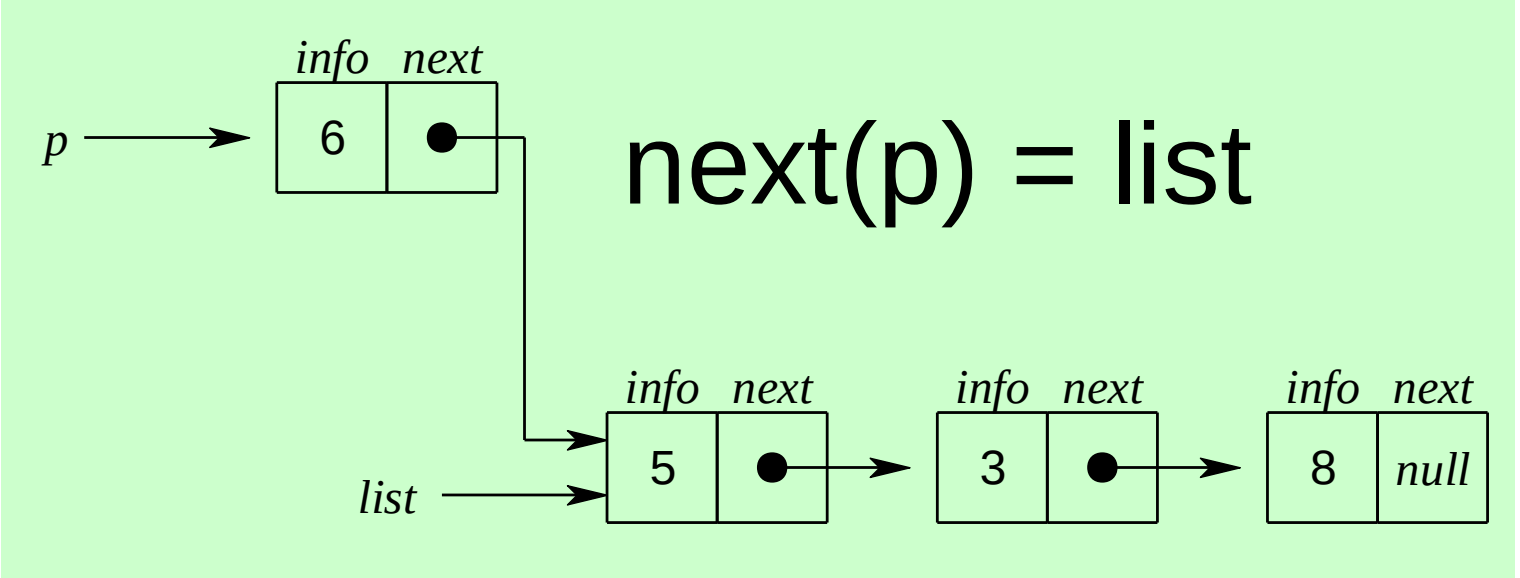
Adding an Element to the front of a Linked List



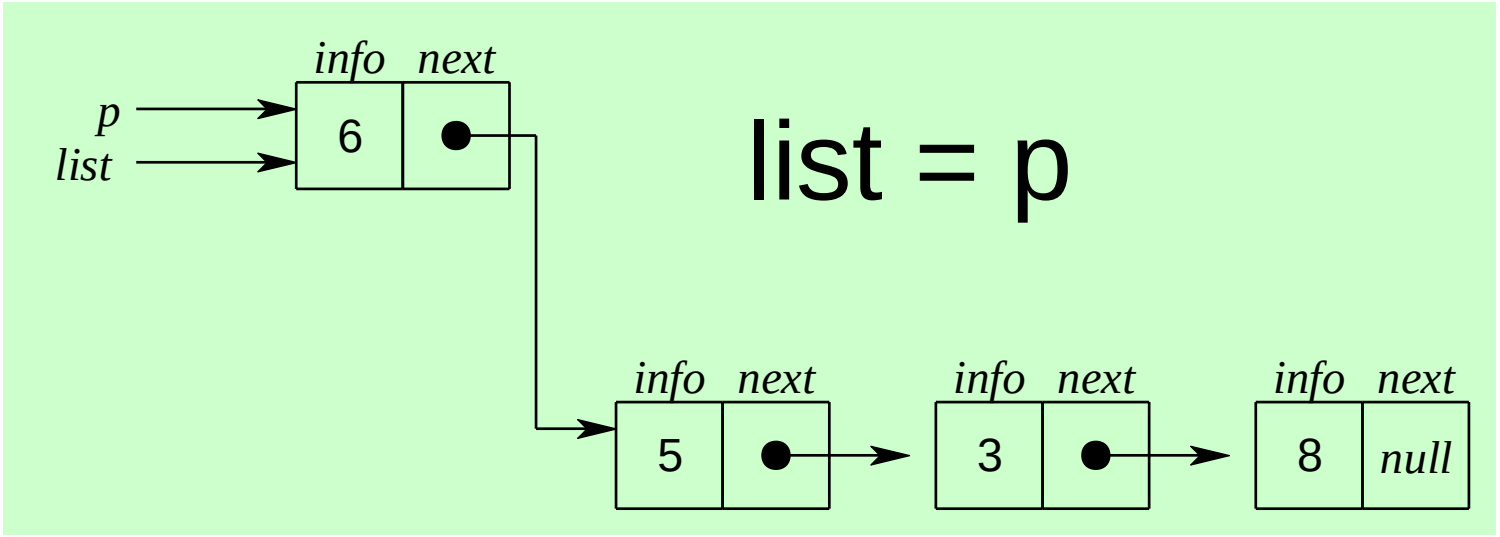
info(p) = 6



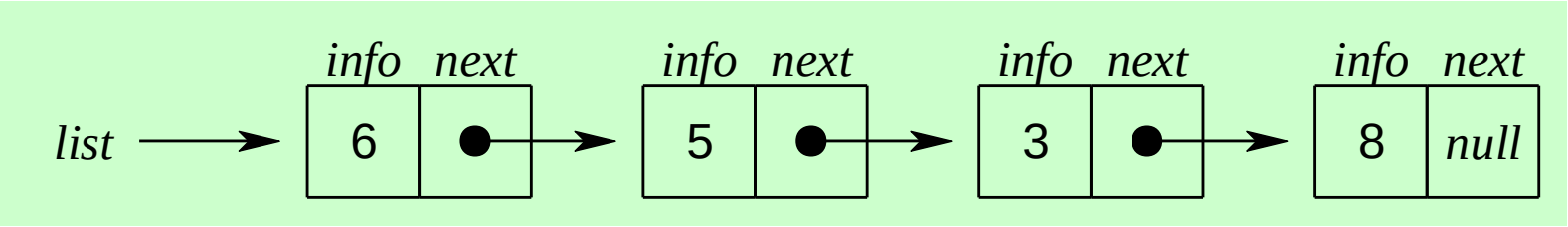
Adding an Element to the front of a Linked List



Adding an Element to the front of a Linked List



Adding an Element to the front of a Linked List



Insert at beginning

inFirst(info, link, Start, avail, item)

This algo. Inserts item as first node in list.

1. **[Overflow?]** If Avail == NULL, then

   Write: Overflow and Exit

2. **[Remove first node from avail list]**

Making new node from avail

Set New = Avail and Avail = Avail→link [shifting privious link of avail to new avail list]

3. Set New→info = item **[copies new data into new node]**
4. Set New→link = Start **[new node points to original first node]**
5. Set Start = New **[changes Start so it points to new node]**
6. Exit

```c
struct Node {
    int data;
    struct Node* next;
};
struct Node* head; // global variable, can be accessed anywhere
void Insert(int x);
void Print();
int main() {
    head = NULL; // empty list;
    printf("How many numbers?\n");
    int n,i,x;
    scanf("%d",&n);
    for(i = 0;i<n;i++){
        printf("Enter the number \n");
        scanf("%d",&x);
        Insert(x);
        Print();
    }
}
struct Node* head;
void Insert(int x)
{
    Node* temp = (Node*)malloc(sizeof(struct Node));
    temp->data = x;
    temp->next = NULL;
    head = temp;
}
```

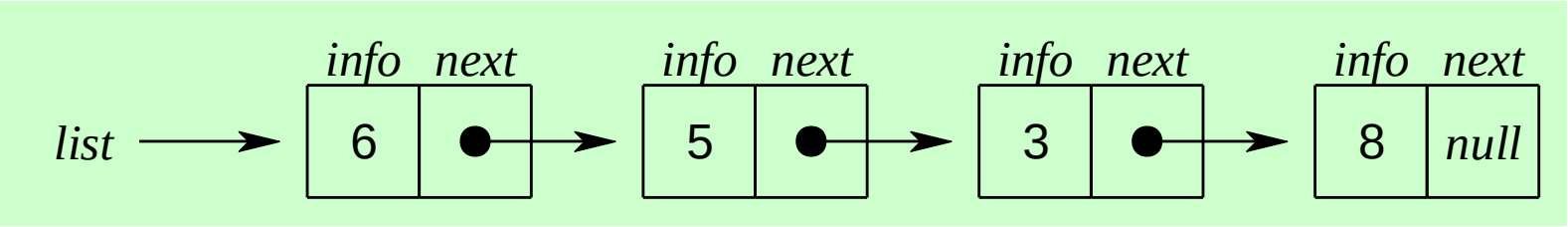
Inserting after a given node location

insLOC(info, link, Start, avail, loc, item)

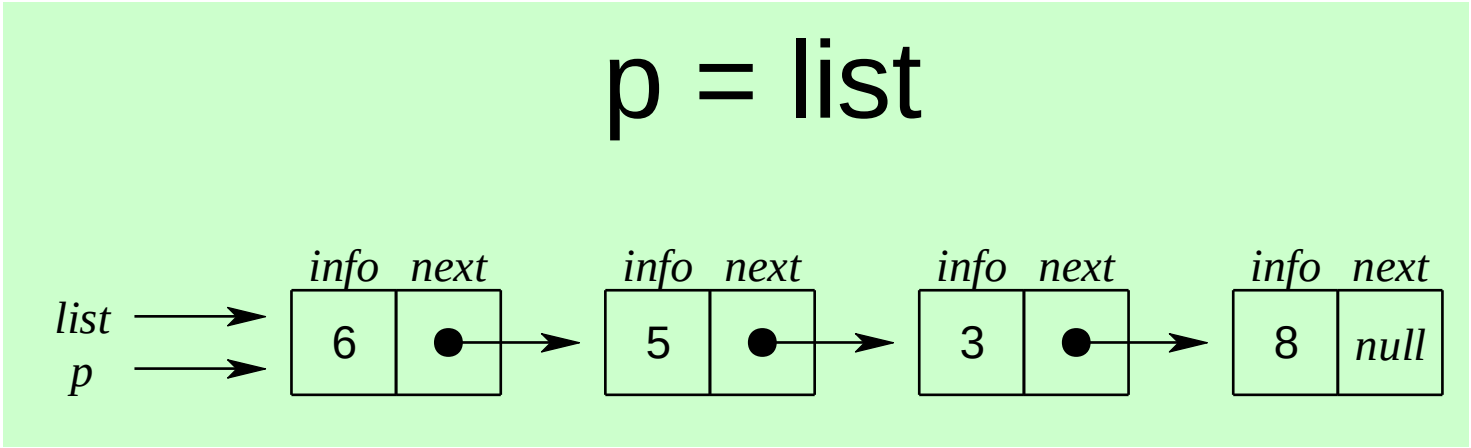
This algo. inserts ITEM so that ITEM follows node with location Loc or inserts ITEM as first node when Loc=NULL

1. **[Overflow?]** If Avail == NULL, then
   Write: Overflow and Exit
2. **[Remove first node from avail list]**
   Set New = Avail and Avail = Avail→link
3. Set New→info = ITEM **[copies new data into new node]**
4. If Loc == NULL, then **[Insert as first node when list is empty]**
   Set head =New →Loc (add. Part of node) and Start = New
   Else **[insert after node with location Loc]**
   Set New→link = Loc→link and PrevLoc→link = New
   [End of If structure]
5. Exit.

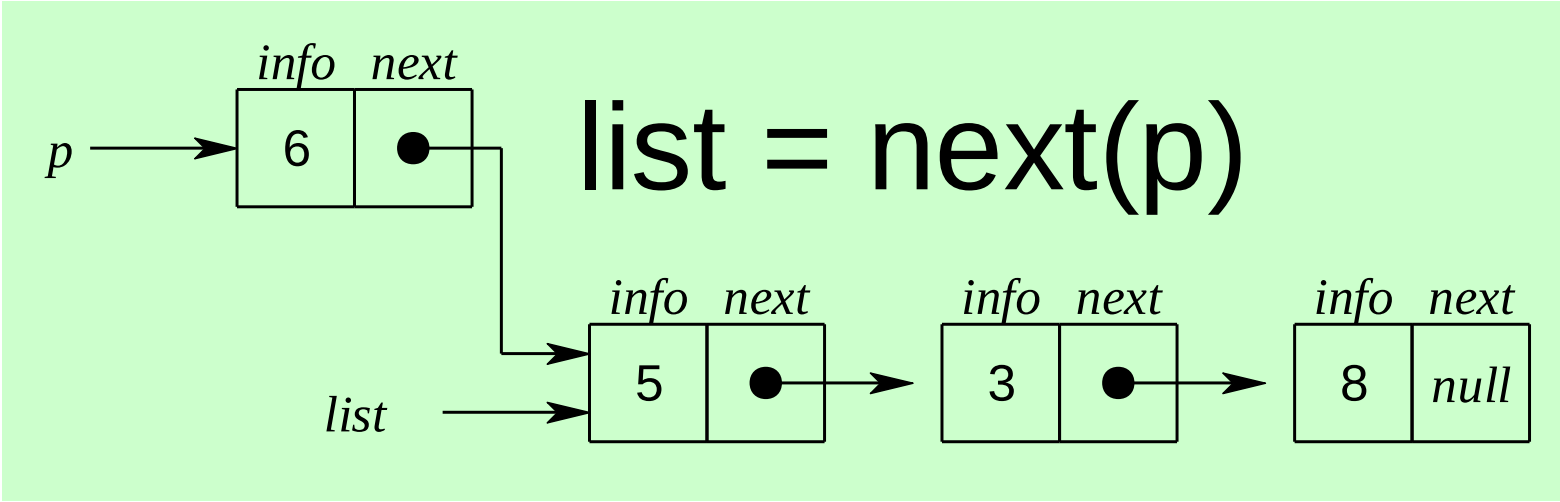
Removing an Element from the front of a Linked List



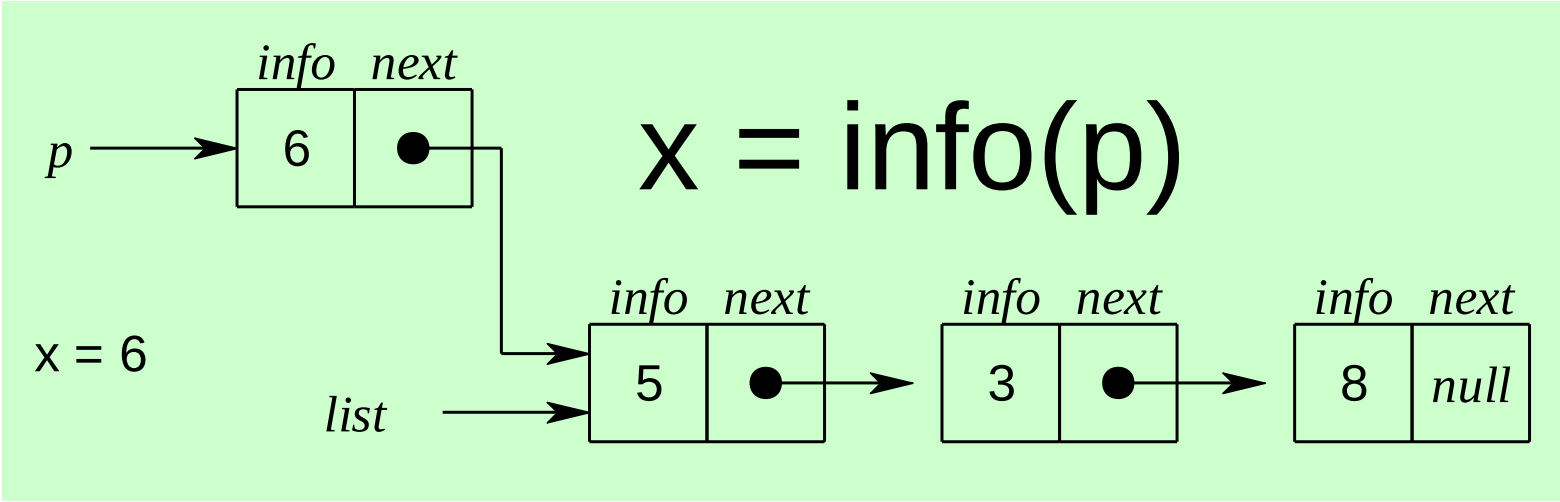
Removing an Element from the front of a Linked List



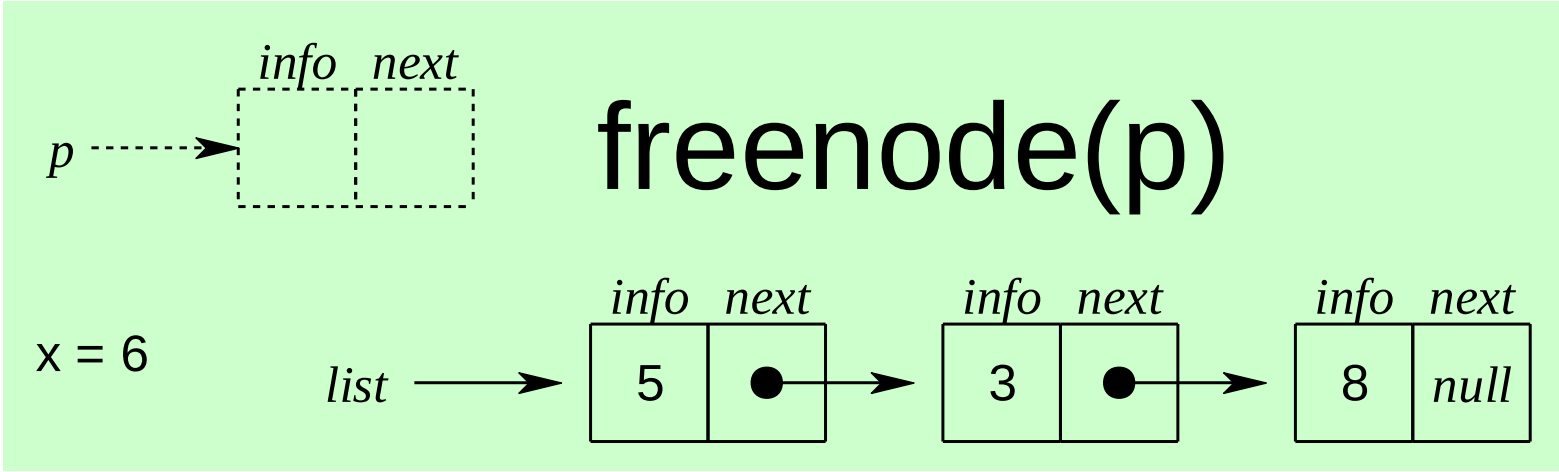
Removing an Element from the front of a Linked List



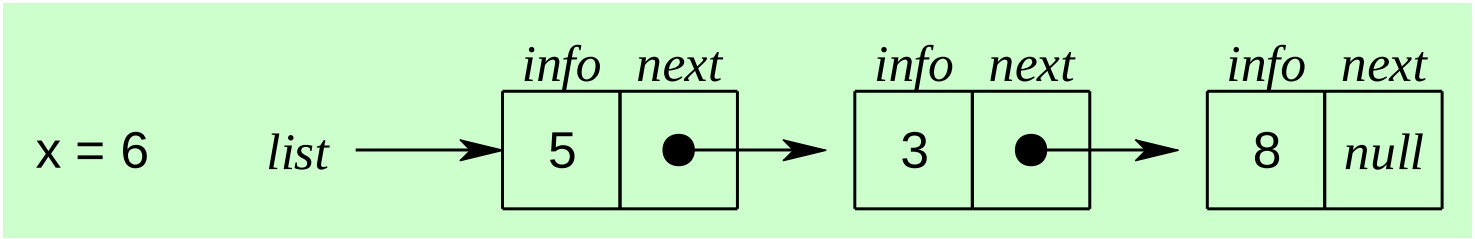
Removing an Element from the front of a Linked List



Removing an Element from the front of a Linked List



Removing an Element from the front of a Linked List

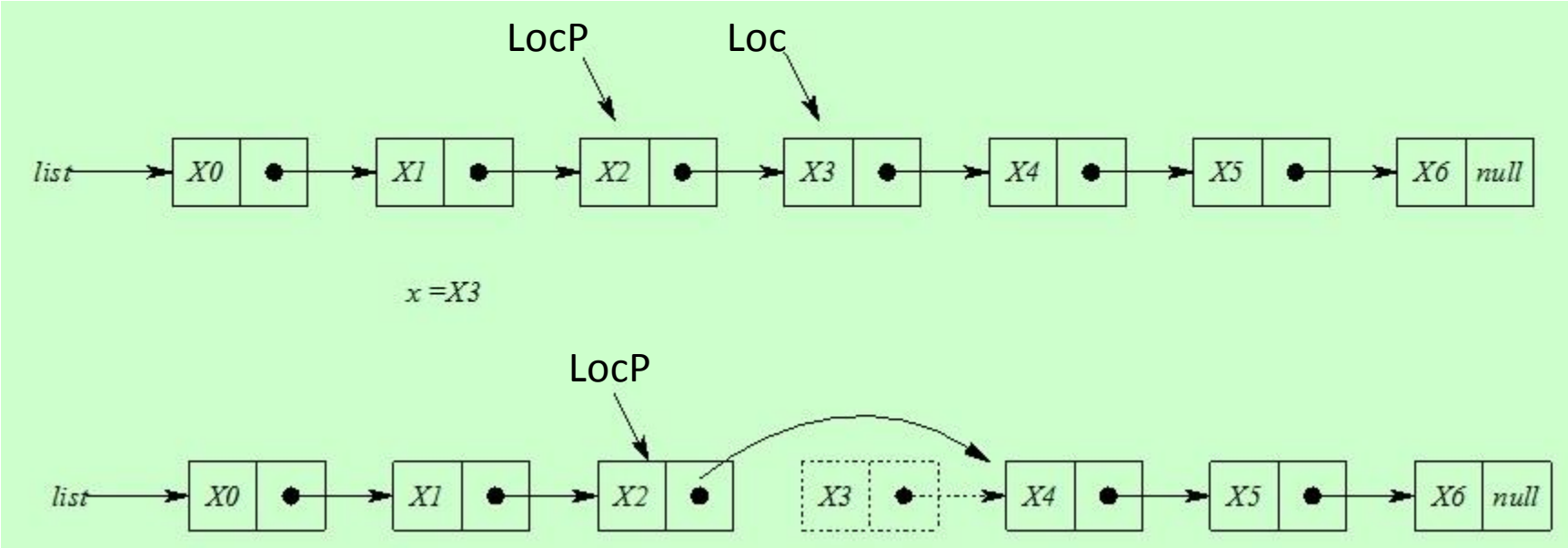


Delete at beginning

Delete()

1. If Head == NULL, then
   Write: underflow and Exit
2. Set item = Start→info
3. Set Loc = Start
3. Set Start = Start →link **[Deletes 1st node]**
4. **[Return deleted node to Avail list]**
   Set Avail →link =Loc and Avail = Loc
5. Exit

Deleting an item *x* from a list pointed to by *Loc*



Delete a node at given location/position

Del(info, link, Start, Avail, Loc, LocP)

This algo deletes node N with location Loc. LocP is location of node which precedes N or when N is first node then LocP = NULL

1. If Head == NULL, then

   Write: underflow and Exit

2. Set item = Start→info
3. If LocP == NULL then,

   Set Start = Start →link **[deletes 1st node]**

Else

   Set LocP →link = Loc →link **[deletes node N]**

[Return deleted node to avail list]

   Set Avail →link =Loc and Avail = Loc and Loc →Link=Null

4. Exit.

Delete a node at END

Del(info, link, Start, Avail, Loc, LocP)

This algo deletes node N with location Loc. LocP is location of node which precedes N or when N is first node then LocP = NULL

1. If Head == NULL, then

   Write: underflow and Exit

2. Set item = Start→info

3. If LocP == NULL then,

   Set Start = Start →link **[deletes 1st node]**

Else

   Set LocP →link = Null **[deletes node N]**

[Return deleted node to avail list]

   Set Avail →link =Loc and Avail = Loc and Loc →Link=Null

4. Exit.

Header Linked Lists

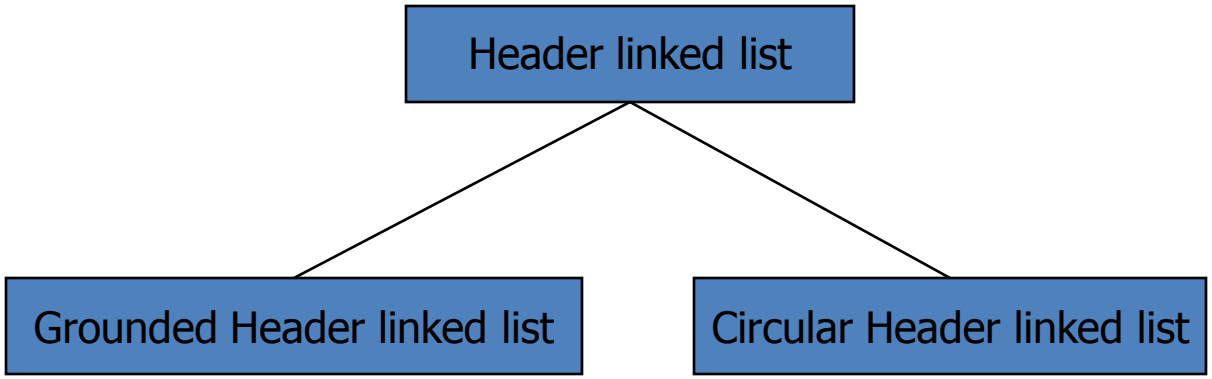
- Header linked list is a linked list which always contains a special node called the Header Node, at the beginning of the list.
- It has two types:

  a) Grounded Header List

  Last Node Contains the NULL Pointer

  b) Circular Header List

  Last Node Points Back to the Header Node



Grounded Header Link List

- A grounded header list is a header list where the last node contains the null pointer.
- The term "**grounded**" comes from the fact that many texts use the electrical ground symbol to indicate the null pointer.

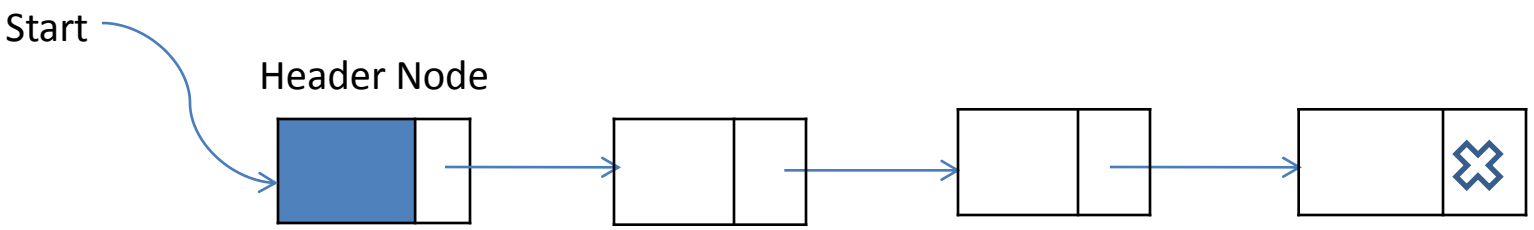


Figure: Grounded Header Link List

Circular Header Linked List

- A circular header Link list is a header list where the last node points back to the header node.
- The chains do not indicate the last node and first node of the link list.
- In this case, external pointers provide a frame reference because last node of a circular link list does not contain null pointer.

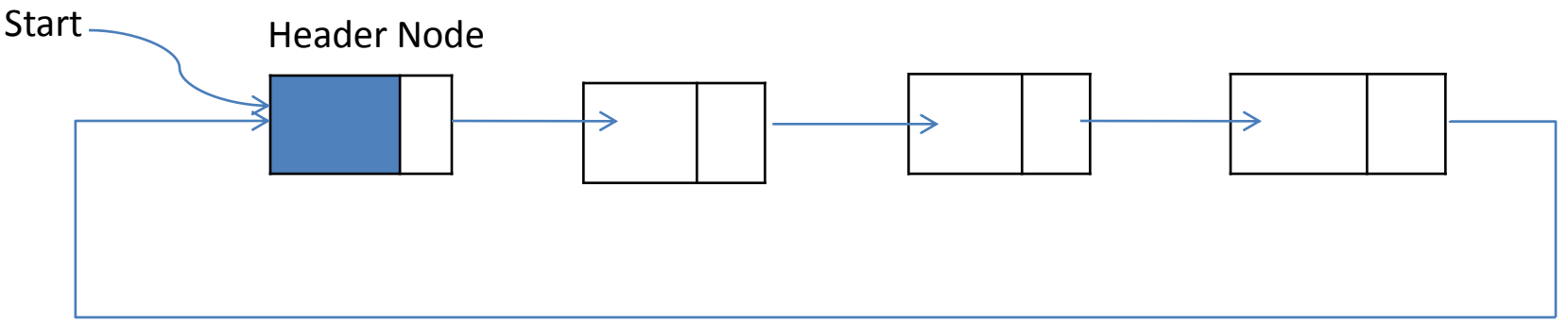
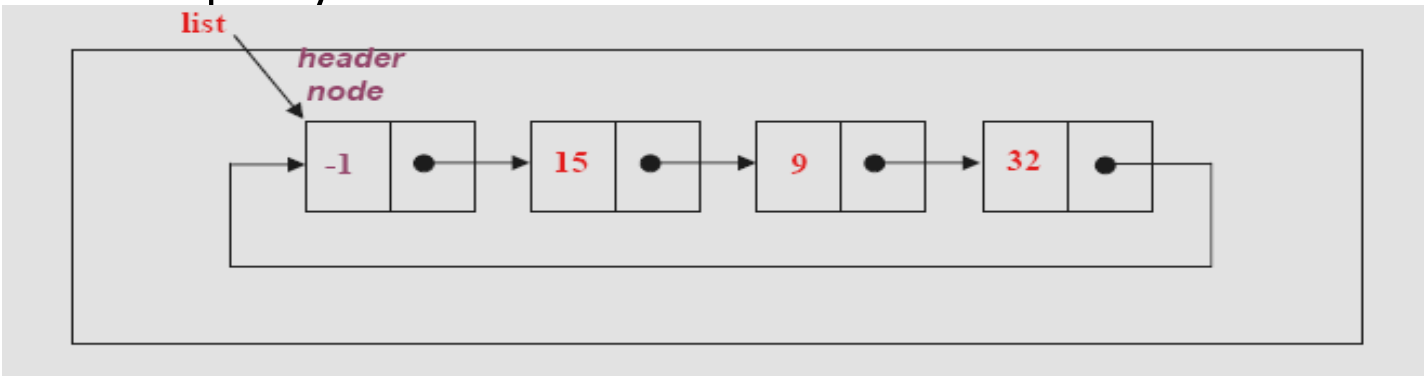
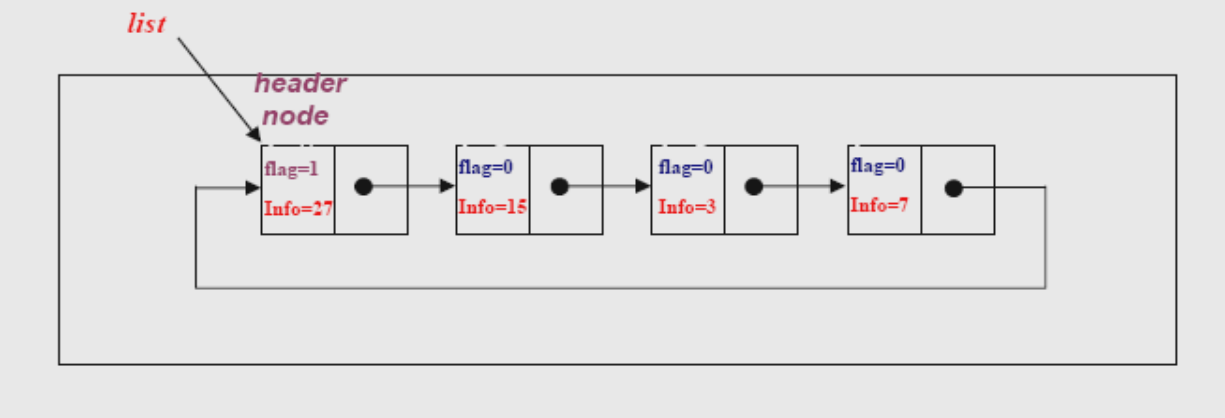


Figure: Circular Linked List with header node

Circular Linked Lists

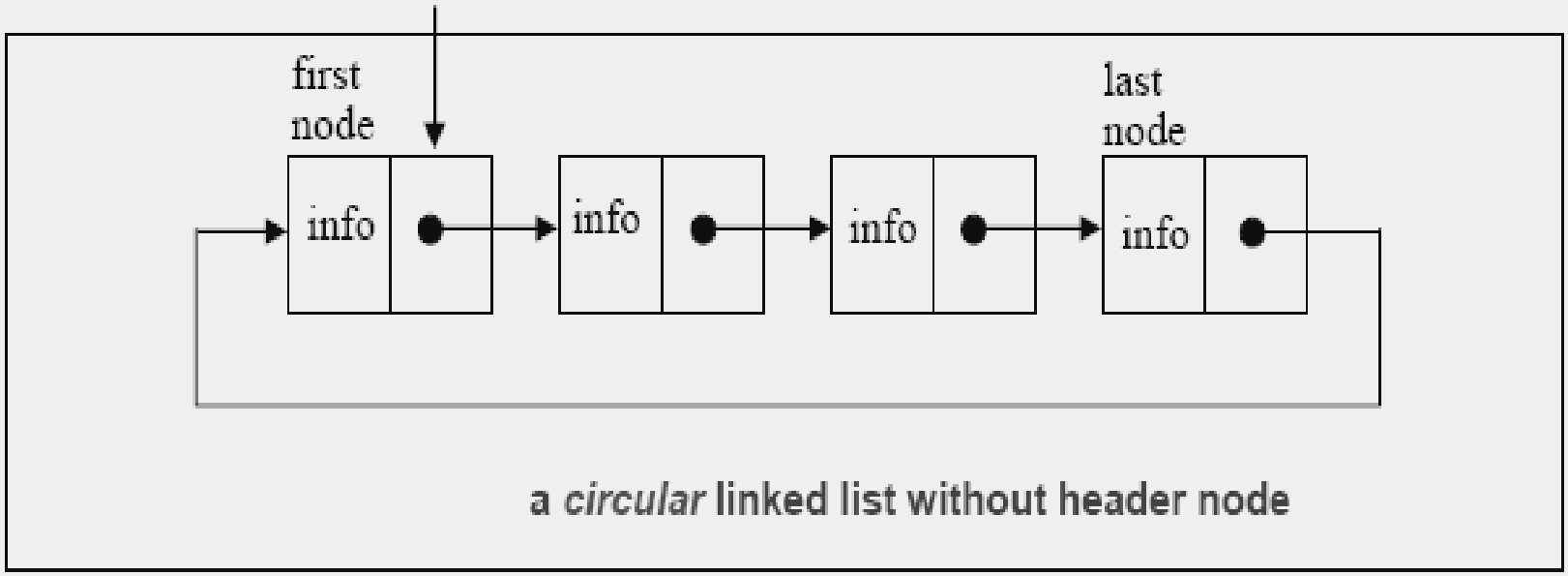
- In a circular linked list there are two methods to know if a node is the first node or not.
  - Either a external pointer, ***list***, points the first node or
  - A ***header node*** is placed as the first node of the circular list.
- The header node can be separated from the others by either heaving a ***sentinel value*** as the info part or having a dedicated ***flag*** variable to specify if the node is a header node or not.





Circular Linked Lists

- In linear linked lists if a list is traversed (all the elements visited) an external pointer to the list must be preserved in order to be able to reference the list again.
- Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.



Traverse Circular list

1. Set Ptr = Start →link **[initializes pointer Ptr]**
2. Repeat steps 3 & 4 while Ptr != Start
3. Apply Process to Ptr →info
4. Set Ptr = Ptr →link **[Ptr now points to next node]**

   [End of Step 2 loop]
5. Exit.

Search in circular list

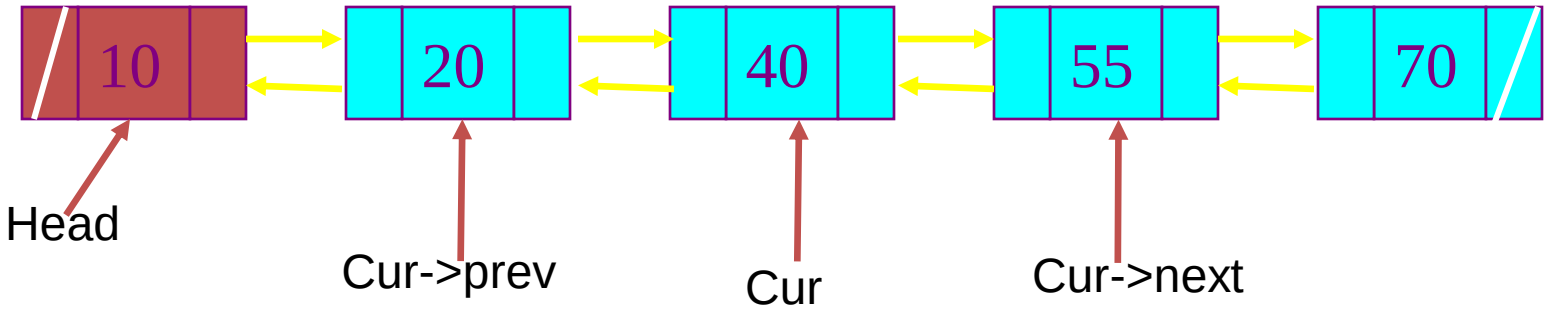
SRCHHL(info, link, Start, item, Loc)

list is circular header list in memory. This algo. Finds location of loc of node where item first appers in list or sets loc=null

1. Set ptr = Start →link
2. Repeat while ptr →info != item and ptr != Start
   Set ptr = ptr →link //[ptr now points to next node]
3. If ptr →info == item, then
   Set Loc = ptr
   else
   Set Loc = Null
4. Exit

Two-way lists

- A two-way list is a linear collection of data elements, called nodes, where each node N is divided into three parts:
  - Information field
  - Forward Link which points to the next node
  - Backward Link which points to the previous node
- The starting address or the address of first node is stored in START / FIRST pointer .
- Another pointer can be used to traverse list from end. This pointer is called END or LAST.



Two-way lists(cont...)

- Every node (except the last node) contains the address of the next node, and every node (except the first node) contains the address of the previous node.
- A two-way list (doubly linked list) can be traversed in either direction.

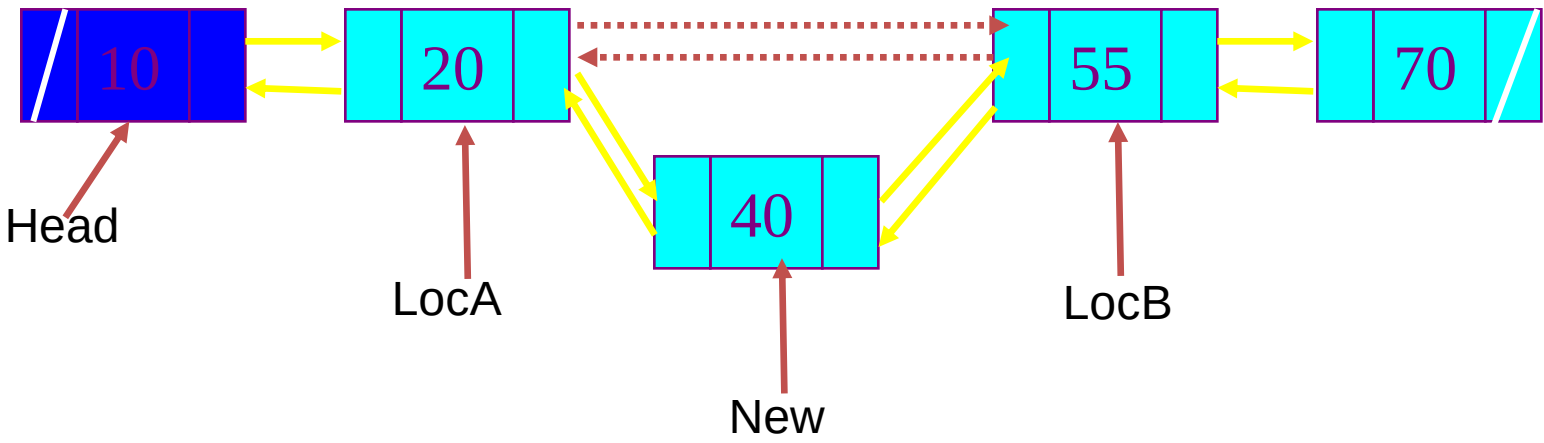
Insertion in 2-way list

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 = Avail →forw,

New →info = item

3. **[Insert node into list]**
   Set LocA →forw = New, New →forw = LocB
   LocB →back = New, New →back = LocA
4. Exit.



Insertion at the beg

insLOC(info, link, Start, avail, item)

This algo. inserts ITEM so that ITEM follows node with location Loc or inserts ITEM as first node when Loc=NULL

1. **[Overflow?]** If Avail == NULL, then

   Write: Overflow and Exit

2. **[Remove first node from avail list]**

   Set New = Avail and Avail = Avail→link

3. Set New→info = ITEM **[copies new data into new node]**

4. New→prev = NULL and Start = New

Start→prev=new and New→forwd=Start

5. Exit.

Inserting after a given node location in 2-way linked list

insLOC(info, link, Start, avail, loc, item)

This algo. inserts ITEM so that ITEM follows node with location Loc or inserts ITEM as first node when Loc=NULL

1. **[Overflow?]** If Avail == NULL, then

   Write: Overflow and Exit

2. **[Remove first node from avail list]**

   Set New = Avail and Avail = Avail→link

3. Set New→info = ITEM **[copies new data into new node]**

4. If Loc == NULL, then **[Insert as first node when list is empty]**

   New→prev = NULL and Start = New

   Start→prev=new and New→forwd=Start

   Else **[insert after node with location Loc]**

   Set New→forw = Loc and (Loc)→back = New

   Set PrevLoc→forw = New and New→Prev = PrevLoc

   [End of If structure]

5. Exit.

Deletion in 2-way list

DelTwl(info, forw, back, Start, Avail, Loc)

1. **[Delete node]**

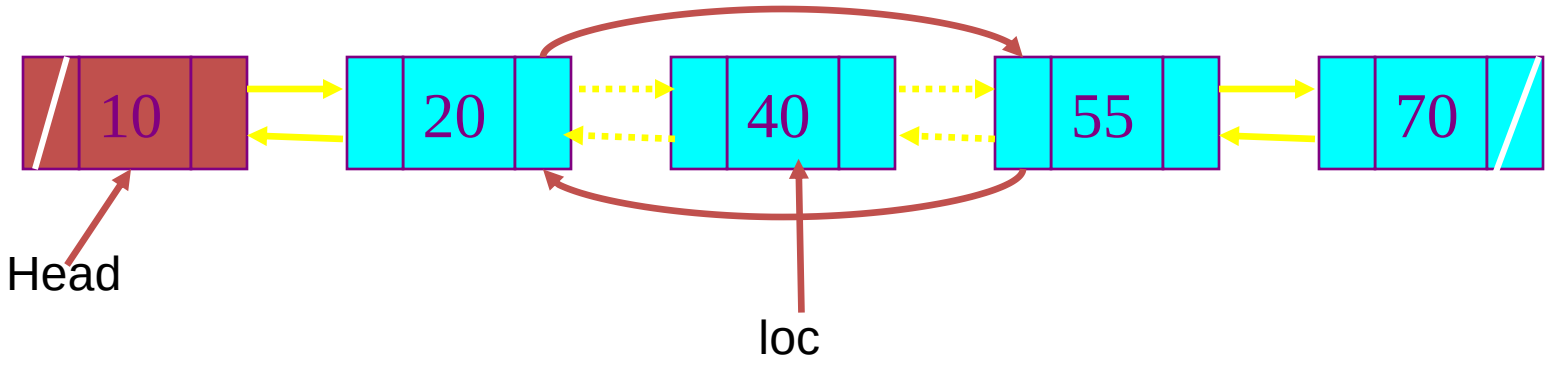
   Set (Loc →back) → forw = Loc →forw and

   (Loc →forw) → back = Loc →back

2. **[Return node to avail list]**

   Set Loc →forw = Avail and Avail = Loc

3. Exit



Deleting an item from 2-way linked list

Delete()

1. Call FindB()
2. If Loc == NULL, then
   Write: ITEM not in list and Exit
3. **[Delete node]**
   If Loc→back == NULL, then
   Set Start = Start →forw **[Deletes 1st node]**
   Else
   Set (Loc →back) → forw = Loc →forw and
   (Loc →forw) → back = Loc →back
   [End of If structure]
4. **[Return node to avail list]**
   Set Loc →forw = Avail and Avail = Loc
5. Exit

FindB()

1. If Head == NULL, then **[List empty?]**
   Set Loc = NULL and **Return**
   [End of If structure]
3. Set Ptr = Start **[Initializes pointer]**
4. Repeat steps 5 and 6 while Ptr != NULL
5. If Ptr→info == ITEM, then
   Set Loc = Ptr and **Return**
   [End of If structure]
6. Set Ptr = Ptr →forw **[Updates pointers]**
   [End of step 4 loop]
7. Set Loc = NULL **[Search unsuccessful]**
8. Return

Doubly linked list

- Two start pointers-first element and last element
- Each node has forward and backward pointer
- Allows traversal both forward and backward

Operations on doubly linked list

- Traversing
- Searching
- Inserting
- Deleting

Insertion at beginning

New→ inserted node

1. Next[New]=Start
2. Prev[New]=Null
3. Prev[start]=New
4. Start=New

Insertion at end

New→ inserted node

1. PTR=Start
2. Repeat while Next[PTR]!=Null
3. PTR=Next[PTR]

   [END of while]
4. Prev[New]=PTR
5. Next[New]=Null
6. Next[PTR]=New

Insertion at middle

- To insert a node ‘New’ after a node ‘S’

1. T=Next[S]
2. Next[New]=T
3. Prev[T]=New
4. Prev[New]=S
5. Next[S]=New

Deletion from beginning

1. PTR=Start
2. Start=Next[PTR]
3. Prev[Start]=Null
4. Next[PTR]=Avail
5. Avail=PTR

Deletion from end

1. PTR=Start
2. Repeat while Next[PTR]!=Null
3. PTR=Next[PTR]

   [END of while]
4. T=Prev[PTR]
5. Next[T]=Null
6. Next[PTR]=Avail
7. Avail=PTR

Deletion from middle

- To delete a node ‘P’ after a given node ‘S’.

1. P=Next[S]
2. T=Next[P]
3. Next[S]=T
4. Prev[T]=S
5. Next[P]=Avail
6. Avail=P

Traversing

- Forward traversing
- Backward traversing

Forward Traversing

Algorithm:

1. Set PTR := START
2. Repeat Steps 3 and 4 while NEXT[PTR] ≠ NULL
3. Apply PROCESS to INFO[PTR]
4. Set PTR := NEXT[PTR].

   [End of Step 2 loop.]
5. Exit.

Backward Traversing

Algorithm:

1. Set PTR := END
2. Repeat Steps 3 and 4 while PREV[PTR] ≠ NULL
3. Apply PROCESS to INFO[PTR]
4. Set PTR := PREV[PTR].

   [End of Step 2 loop.]
5. Exit.

Searching from forward direction

1. Set PTR := START
2. Repeat Step 3 while Forwd[PTR] ≠ NULL.
3. If ITEM = INFO[PTR], then:
   Set LOC := PTR.
   Else:
   Set PTR := NEXT[PTR].
   [End of If structure.]

   [End of Step 2 loop.]
4. [Search is unsuccessful.] Set LOC := NULL.
5. Exit.

Searching from backward direction

1. Set PTR := END
2. Repeat Step 3 while Prev[PTR]≠ NULL.
3. If ITEM = INFO[PTR], then:

   Set LOC := PTR.

   Else:

   Set PTR := PREV[PTR].

   [End of If structure.]

   [End of Step 2 loop.]

4. [Search is unsuccessful.] Set LOC := NULL.
5. Exit.