Data Structure and Algorithms

Chapter 4 - List

Lecturer: Vuong Ba Thinh
Contact: vbthinh@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

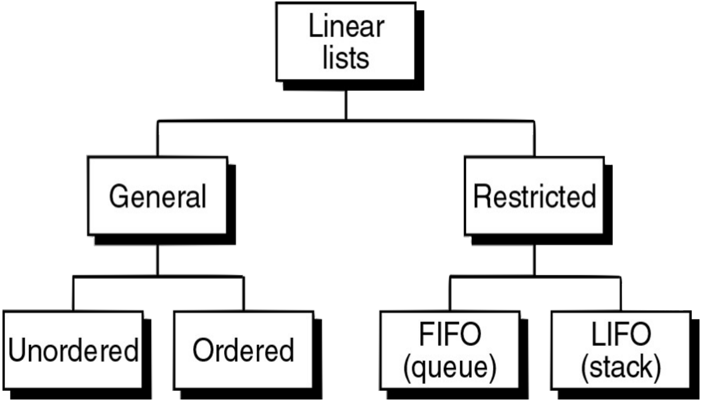
Linear list concepts

Definition
A linear list is a data structure in which each element has a unique successor.

Example

- Array
- Linked list

Linear list concepts



Linear list concepts

General list:

- No restrictions on which operation can be used on the list.
- No restrictions on where data can be inserted/deleted.

- Unordered list (random list): Data are not in particular order.
- Ordered list: data are arranged according to a key.

Restricted list:

- Only some operations can be used on the list.
- Data can be inserted/deleted only at the ends of the list.

- Queue: FIFO (First-In-First-Out).
- Stack: LIFO (Last-In-First-Out).

List ADT

Definition
A list of elements of type T is a finite sequence of elements of T.

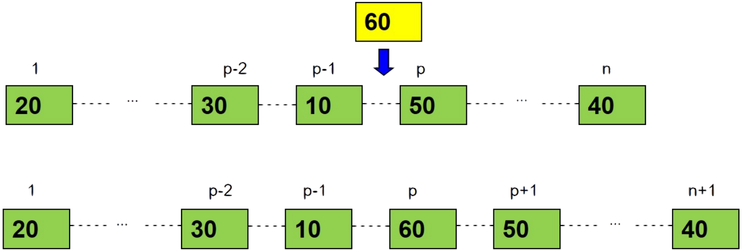
Basic operations:

- Construct a list, leaving it empty.
- Insert an element.
- Remove an element.
- Search an element.
- Retrieve an element.
- Traverse the list, performing a given operation on each element.

Extended operations:

- Determine whether the list is empty or not.
- Determine whether the list is full or not.
- Find the size of the list.
- Clear the list to make it empty.
- Replace an element with another element.
- Merge two ordered list.
- Append an unordered list to another.

- **Insert an element at a specified position p in the list**
  - Only with *General Unordered List*.

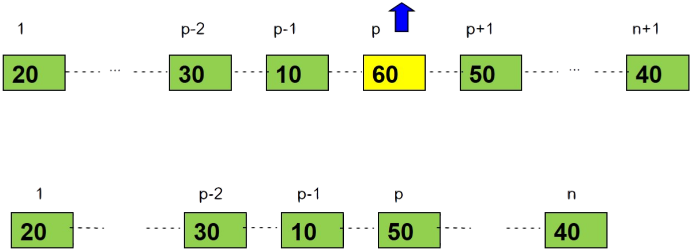


Any element formerly at position p and all later have their position numbers increased by 1.

Insertion

- **Insert an element with a given data**
  - With *General Unordered List*: can be made at any position in the list (at the beginning, in the middle, at the end).
  - With *General Ordered List*: data must be inserted so that the ordering of the list is maintained (searching appropriate position is needed).
  - With *Restricted List*: depend on it own definition (FIFO or LIFO).

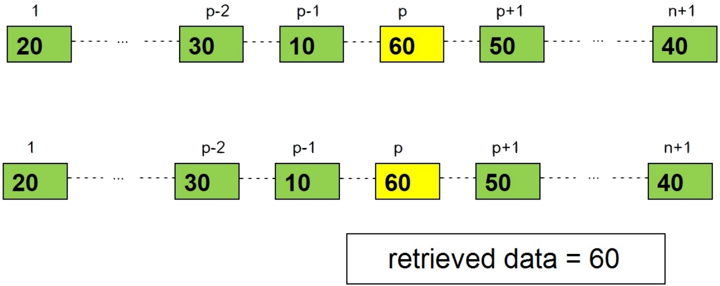
- **Remove an element at a specified position p in the list**
  - With *General Unordered List* and *General Ordered List*.



The element at position p is removed from the list, and all subsequent elements have their position numbers decreased by 1.

Retrieval

- **Retrieve an element at a specified position p in the list**
  - With *General Unordered List* and *General Ordered List*.



All elements remain unchanged.

Removal, Retrieval

- **Remove/ Retrieve an element with a given data**
  - With *General Unordered List* and *General Ordered List*: Searching is needed in order to locate the data being deleted/ retrieved.

Success of Basic Operations

- **Insertion** is successful when the list is not full.
- **Removal, Retrieval** are successful when the list is not empty.

Singly linked list

Definition
A linked list is an ordered collection of data in which each element contains the location of the next element.

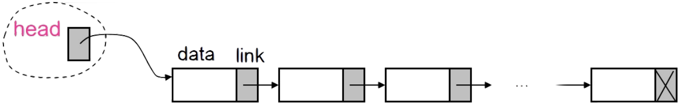


Figure 1: Singly Linked List

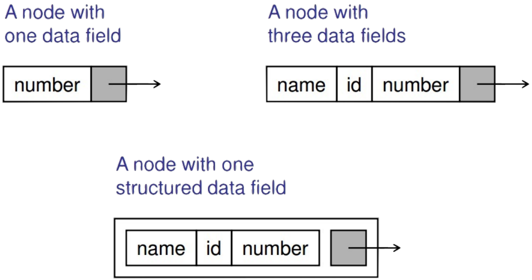
```
list // Linked Implementation of List
  head <pointer>
  count <integer> // number of elements (optional)
end list
```

Nodes

The elements in a linked list are called nodes.

A node in a linked list is a structure that has at least two fields:

- the data,
- the address of the next node.



Nodes

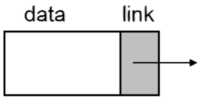
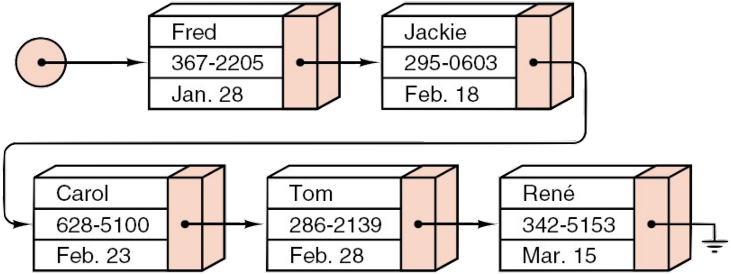


Figure 2: Linked list node structure

```
node
  data <dataType>
  link <pointer>
end node
```

```
// General dataType:
dataType
   key <keyType>
   field1 <...>
   field2 <...>
   ...
   fieldn <...>
end dataType
```

Example



Fred
367-2205
Jan. 28
Jackie
295-0603
Feb. 18
Carol
628-5100
Feb. 23
Tom
286-2139
Feb. 28
René
342-5153
Mar. 15

Example

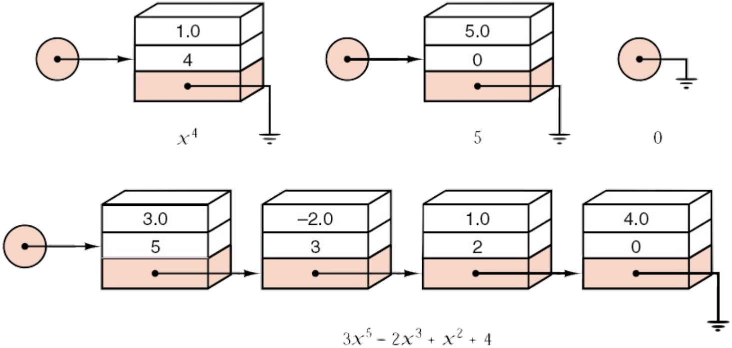


Figure 3: List representing polynomial

Example

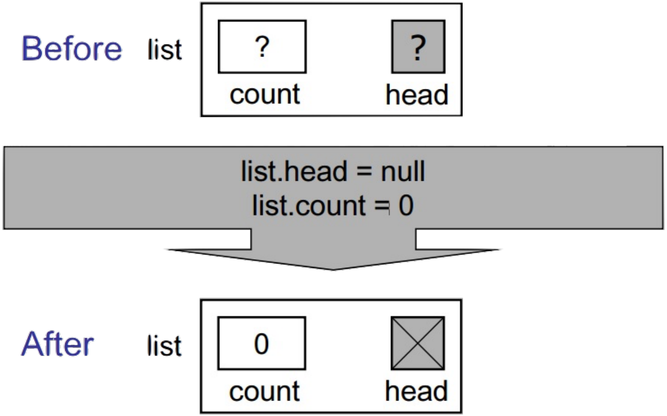
```
node
   data <dataType>
   link <pointer>
end node
```

```python
class ListNode:
    def __init__(self, data):
        "constructor to initiate this object"

        # store data
        self.data = data

        # store reference (next item)
        self.next = None
        return
```

Linked list operations

- Create an empty linked list
- Insert a node into a linked list
- Delete a node from a linked list
- Traverse a linked list
- Destroy a linked list

Create an empty linked list



Create an empty linked list

Algorithm createList(ref list <metadata>)

Initializes metadata for a linked list

Pre: `list` is a metadata structure passed by reference

Post: metadata initialized

list.head = null

list.count= 0

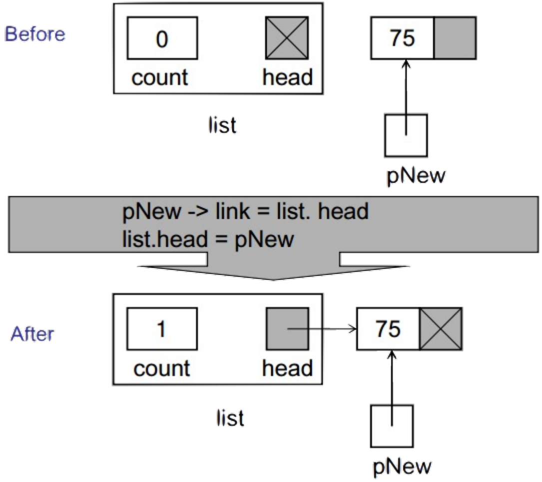
return

End createList

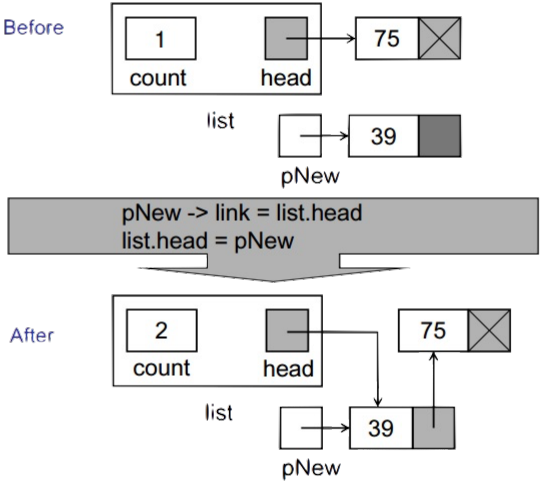
Insert a node into a linked list

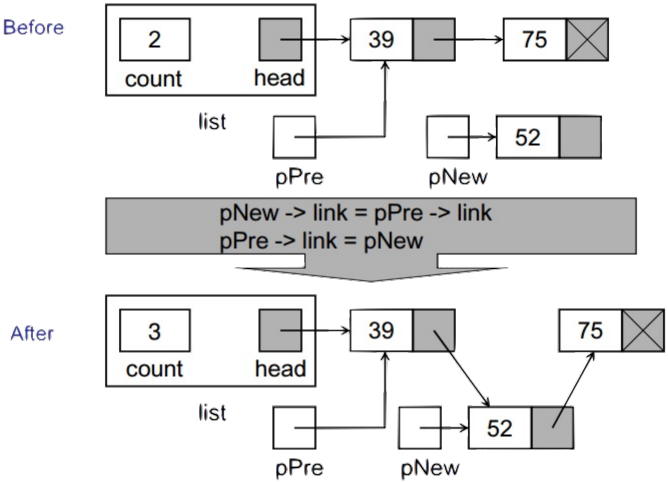
1. Allocate memory for the new node and set up data.
2. Locate the pointer p in the list, which will point to the new node:
   - If the new node becomes the first element in the List: `p is list.head`.
   - Otherwise: `p is pPre->link`, where `pPre` points to the predecessor of the new node.
3. Point the new node to its successor.
4. Point the pointer p to the new node.

Insert into an empty linked list

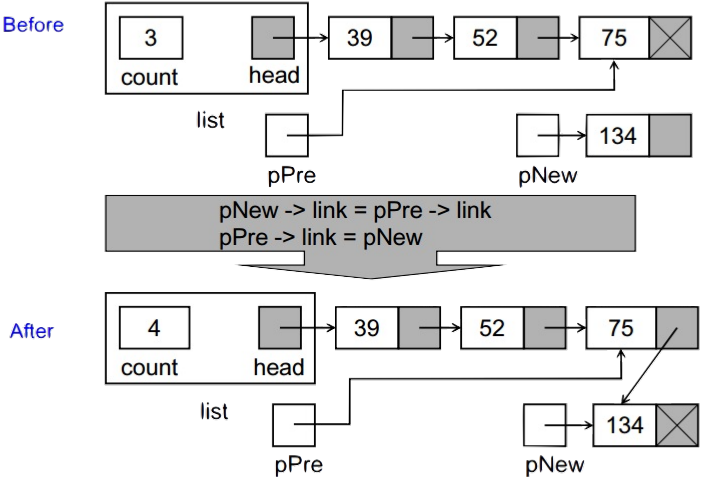


Insert at the beginning




Before
2
count
head
39
75
list
pPre
52
pNew
pNew -> link = pPre -> link
pPre -> link = pNew
After
3
count
head
39
75
list
52
pPre
pNew

Insert at the end



Insert a node into a linked list

- Insertion is successful when allocation memory for the new node is successful.
- There is no difference between insertion at the beginning of the list and insertion into an empty list.

```
pNew->link = list.head
list.head = pNew
```

- There is no difference between insertion in the middle and insertion at the end of the list.

```
pNew->link = pPre->link
pPre->link = pNew
```

Insert a node into a linked list

Algorithm insertNode(ref list <metadata>,
val pPre <node pointer>,
val dataIn <dataType>)

Inserts data into a new node in the linked list.

Pre: `list` is metadata structure to a valid list
`pPre` is pointer to data's logical predecessor
`dataIn` contains data to be inserted

Post: data have been inserted in sequence

Return true if successful, false if memory overflow

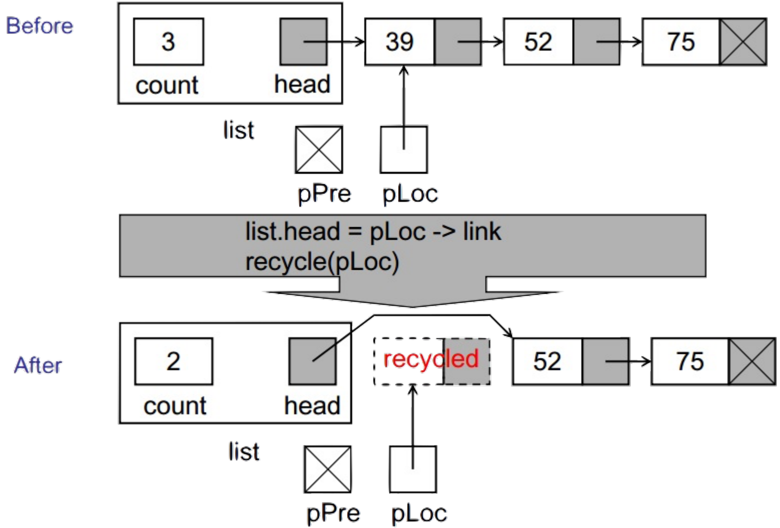
Insert a node into a linked list

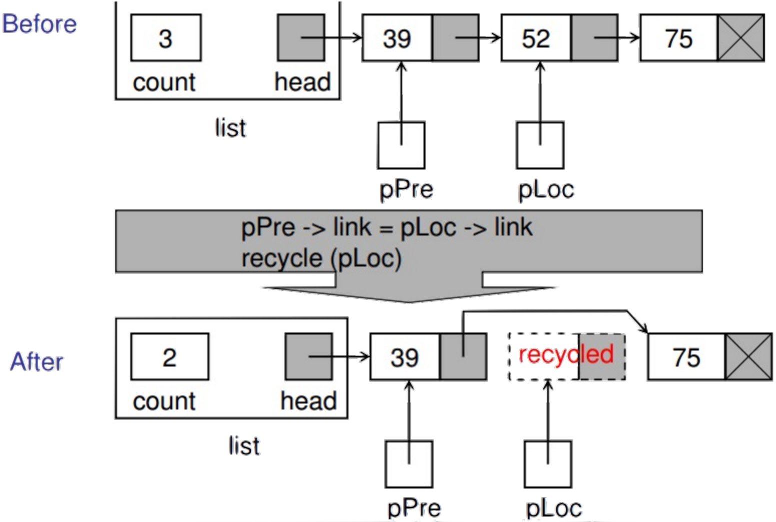
```
allocate(pNew)
if memory overflow then
    return false
end
pNew -> data = dataIn
if pPre = null then
    // Adding at the beginning or into empty list
    pNew -> link = list.head
    list.head = pNew
else
    // Adding in the middle or at the end
    pNew -> link = pPre -> link
    pPre -> link = pNew
end
list.count = list.count + 1
return true
End insertNode
```

Delete a node from a linked list

1. Locate the pointer p in the list which points to the node to be deleted (pLoc will hold the node to be deleted).
   - If that node is the first element in the List: `p is list.head`.
   - Otherwise: `p is pPre->link`, where `pPre` points to the predecessor of the node to be deleted.
2. p points to the successor of the node to be deleted.
3. Recycle the memory of the deleted node.

Delete first node




Before
3
count
head
list
39
52
75
pPre
pLoc
pPre -> link = pLoc -> link
recycle (pLoc)
After
2
count
head
list
39
recycled
75
pPre
pLoc

Delete a node from a linked list

- Removal is successful when the node to be deleted is found.
- There is no difference between deleting the node from the beginning of the list and deleting the only node in the list.

```
list.head = pLoc->link
recycle(pLoc)
```

- There is no difference between deleting a node from the middle and deleting a node from the end of the list.

```
pPre->link = pLoc->link
recycle(pLoc)
```

Delete a node from a linked list

Algorithm deleteNode(ref list <metadata>,
val pPre <node pointer>,
val pLoc <node pointer>,
ref dataOut <dataType>)

Deletes data from a linked list and returns it to calling module.

Pre: `list` is metadata structure to a valid list
`pPre` is a pointer to predecessor node
`pLoc` is a pointer to node to be deleted
`dataOut` is variable to receive deleted data

Post: data have been deleted and returned to caller

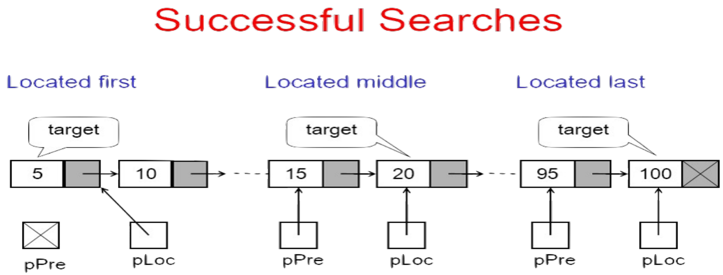
Delete a node from a linked list

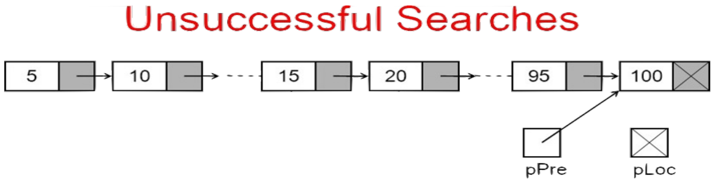
```
dataOut = pLoc -> data
if pPre = null then
    // Delete first node
    list.head = pLoc -> link
else
    // Delete other nodes
    pPre -> link = pLoc -> link
end
list.count = list.count - 1
recycle (pLoc)
return
End deleteNode
```

- Sequence Search has to be used for the linked list.
- Function Search of List ADT:

```
<ErrorCode> Search (val target <dataType>,
            ref pPre <pointer>, ref pLoc <pointer>)
```

  Searches a node and returns a pointer to it if found.


Successful Searches
Located first
Located middle
Located last
target
target
target
5
10
15
20
95
100
pPre
pLoc
pPre
pLoc
pPre
pLoc


Unsuccessful Searches
5
10
15
20
95
100
pPre
pLoc

Algorithm Search(val target <dataType>,
ref pPre <node pointer>,
ref pLoc <node pointer>)

Searches a node in a singly linked list and return a pointer to it if found.

Pre: `target` is the value need to be found

Post: `pLoc` points to the first node which is equal target, or is NULL if not found.
`pPre` points to the predecessor of the first node which is equal target, or points to the last node if not found.

Return `found` or `notFound`

Searching in a linked list

```
pPre = NULL
pLoc = list.head
while (pLoc is not NULL) AND (target != pLoc ->data) do
    pPre = pLoc
    pLoc = pLoc ->link
end
if pLoc is NULL then
    return notFound
else
    return found
end
End Search
```

Traverse a linked list

Traverse module controls the loop: calling a user-supplied algorithm to process data

Algorithm Traverse(ref <void> process (ref Data <DataType>))
Traverses the list, performing the given operation on each element.
Pre: `process` is user-supplied
Post: The action specified by `process` has been performed on every element in the list, beginning at the first element and doing each in turn.

pWalker = list.head
while *pWalker not null* **do**
  process(pWalker -> data)
  pWalker = pWalker -> link
end
End Traverse

Destroy a linked list

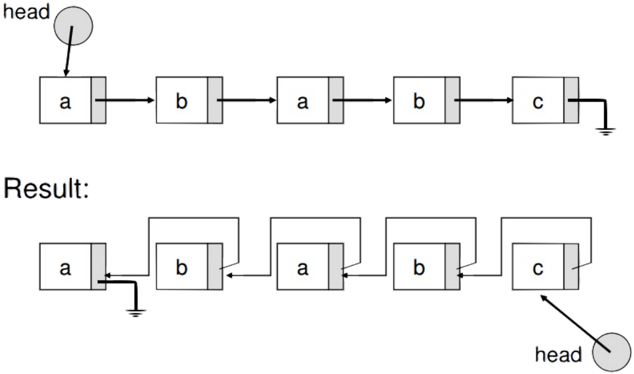
Algorithm destroyList (val list <metadata>)
Deletes all data in list.

Pre: `list` is metadata structure to a valid list
Post: all data deleted
while *list.head not null* **do**
    dltPtr = list.head
    list.head = this.head -> link
    recycle (dltPtr)
end
No data left in list. Reset metadata
list.count = 0
return
End destroyList

Linked list implementation in C++

```
template<class List_ItemType>
class LinkedList{
protected:
  // ...
public:
  LinkedList();
  ~LinkedList();
  void InsertFirst(List_ItemType value);
  void InsertLast(List_ItemType value);
  int InsertItem(List_ItemType value, int position);
  void DeleteFirst();
  void DeleteLast();
  int DeleteItem(int postion);
  int GetItem(int position, List_ItemType &dataOut);
  void Traverse();
  LinkedList<List_ItemType>* Clone();
  void Print2Console();
  void Clear();
};
```

Reverse a linked list



Other linked lists

Doubly Linked List

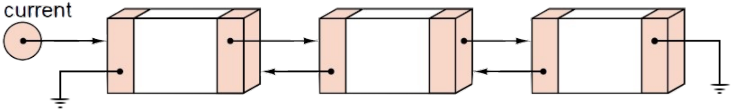


Figure 4: Doubly Linked List allows going forward and backward.

```
node
  data <dataType>
  next <pointer>
  previous <pointer>
end node
```

```
list
  current <pointer>
end list
```

Doubly Linked List

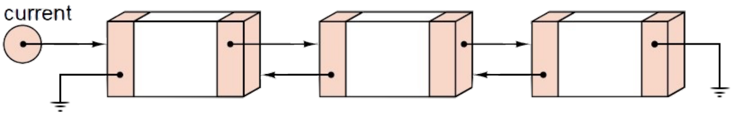


Figure 5: Doubly Linked List allows going forward and backward.

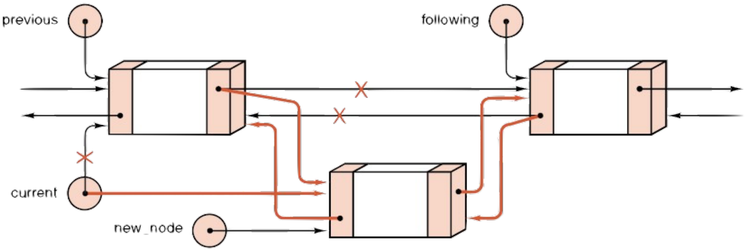
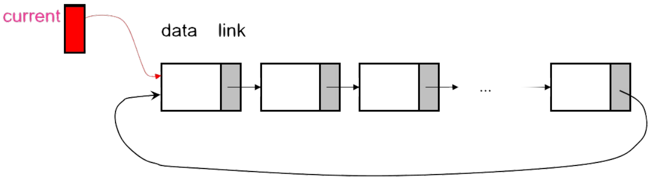


Figure 6: Insert an element in Doubly Linked List.

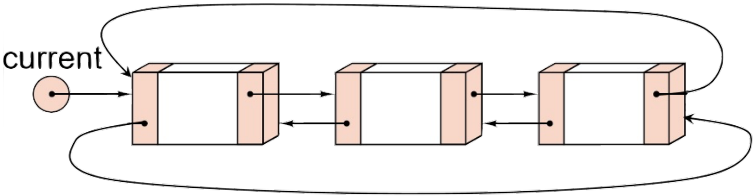
Circularly Linked List




```
node
   data <dataType>
   link <pointer>
end node
```

```
list
   current <pointer>
end list
```

Double circularly Linked List




```
node
   data <dataType>
   next <pointer>
   previous <pointer>
end node
```

```
list
   current <pointer>
end list
```