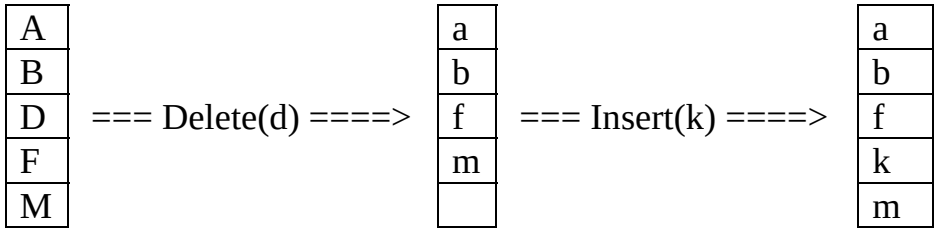
Linked Lists Structures

1. Motivation

- Insertion & Deletion in an ordered array require moving elements up or down (left or right).
- Manipulating ordered lists of varying sizes
- Example:



- Solution:
  - use linked lists
  - creation of variable's storage space in memory during the execution (rather than compilation) of the program
  - Each data (a set of data) is associated with an address: form a node.
- Implementation:
  - Use heap
  - Allocation of variable's storage space: needs a variable that holds the address of that space.
    - Pascal: pointer variable
    - Ada: Access variable
    - C: For any type in C, there is a corresponding type
      pointer-to-T: The value of the pointer is an address of memory.

- Requirements:
  - Mechanism to define the structure of a node (all fields)
  - A way to create a node
  - A way to free no longer needed nodes

2. Declaration of a node

- A C++ Class:

```
class Node{

        Private:
                element_type data;
                Node *left_link;
                Node *right_link;
}
```

- Create an object node:

```
Node head_ptr;
Node tail_ptr;
```

- As a structure:

```
Struct Node {
        element_type data;
        struct Node *left_link;
        struct Node *right_link;
};
```

- Struct Node *node_pointer;

- node_pointer start, end;

- Access Fields:

  start->data = element;

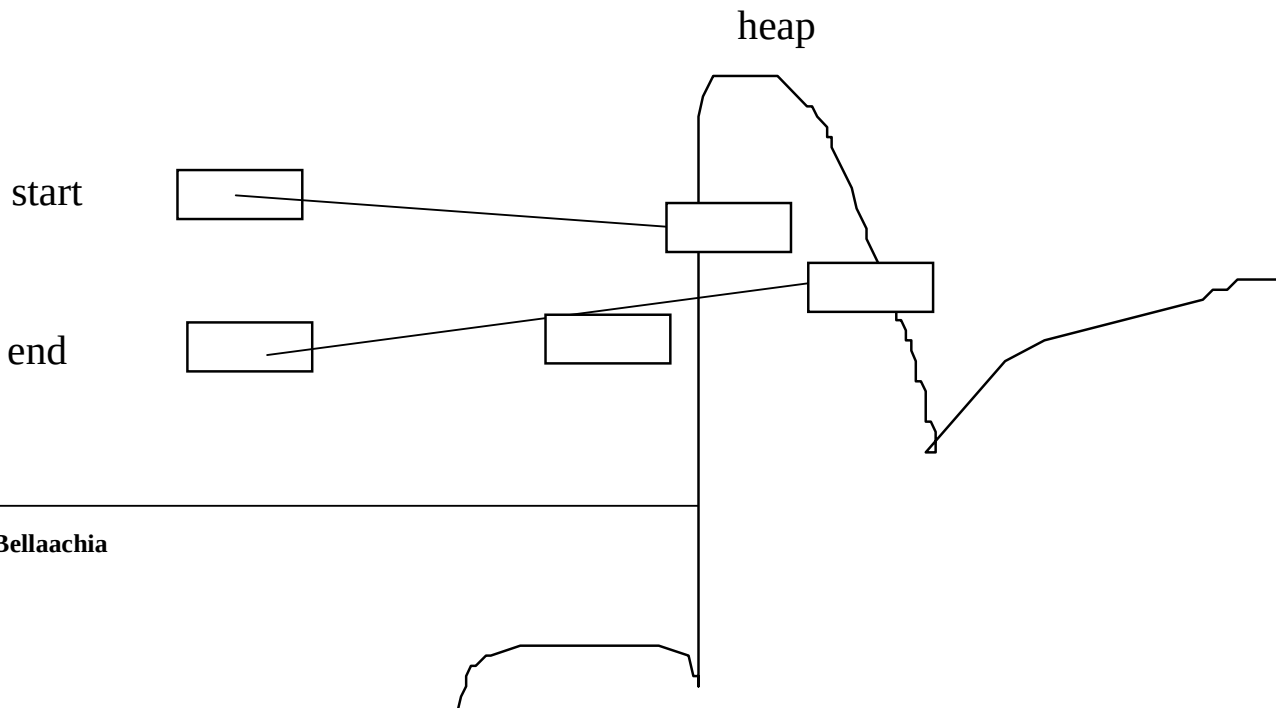
- Constant nil:

  - the pointer does not point to any node
  - start = nil or null;

2.1. Operations on pointers

- Comparison: start == end, etc.
- Assignments: start = end, etc.
- Arithmetic: depending on the language

- Creation of a node:
  - Primitive procedure: malloc, new, etc.



2.2. Disposition of a node

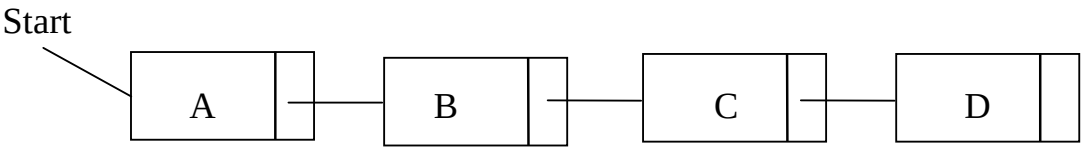
- Primitive procedure: free(p), dispose(p), etc.

2.3. Singly Linked Lists

- One-way linked lists
- One-way linked lists with head and tail
- Circular one-way linked lists

3. One-way linked lists

- **An example:**

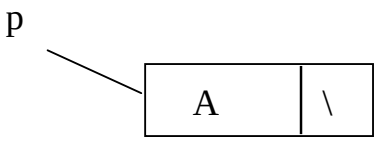


- **Operations**: Sorted or Unsorted

  - Insertion
  - Deletion
  - Find

3.1. Insertion in an unsorted list:

- Creation of a node:



- Case 1: When the list is not empty; start != nil.

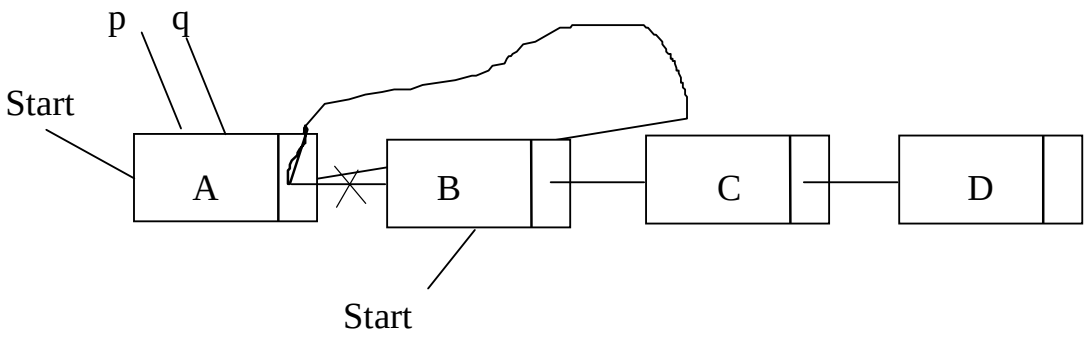
```
p->link = start;
start = p;
```

- Case 2:When the list is empty; start=nil.

```
start = p;
```

3.2. Deletion

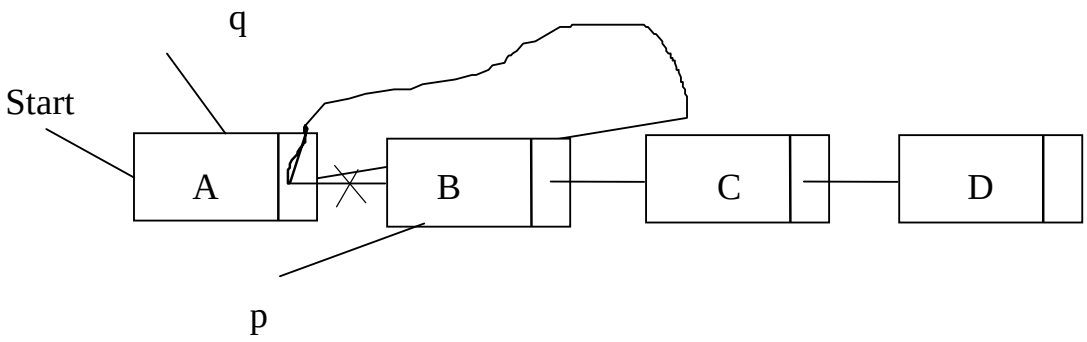
- Case 1: The first element of the list



Delete(A):

```
p = q = Start
Start = Start->link;
free(q);
```

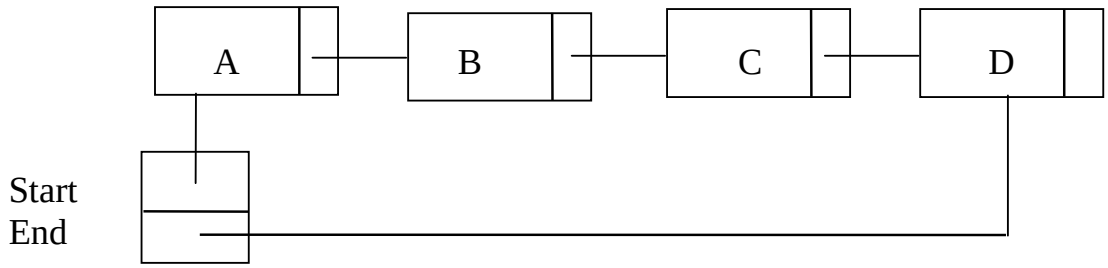
- Case 2: An element different than the first one in the list




```
q ->link = p->link;
free(p);
```

4. One-way linked lists with Head and Tail

- **An example:**

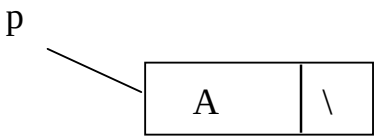


4.1. Operations: **Sorted or Unsorted**

- Insertion
- Deletion
- Find

4.2. Insertion in an unsorted list:

- Creation of a node:



- Case 1: When the list is not empty; start != nil and Tail != nil;

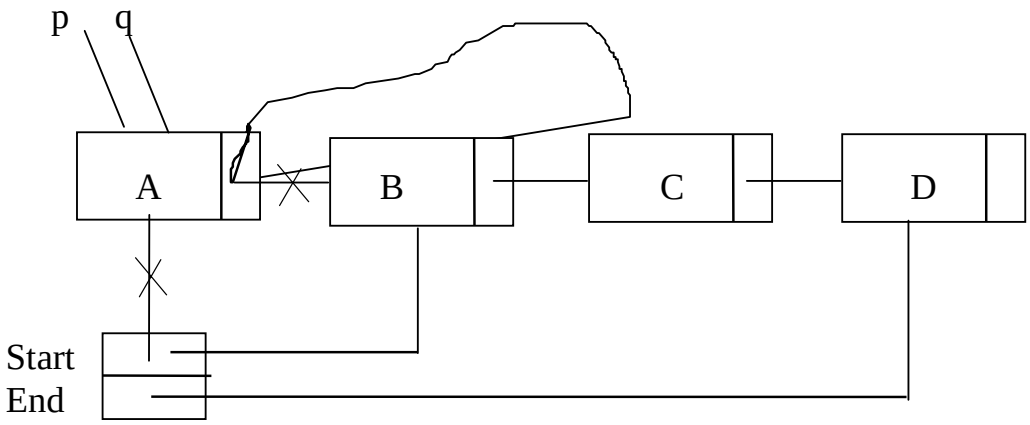
  p->link = start;
  start = p;

- Case 2:When the list is empty; start=nil.

  start = p;
  Tail = p;

4.3. Deletion

- Case 1: The first element of the list



Delete(A):

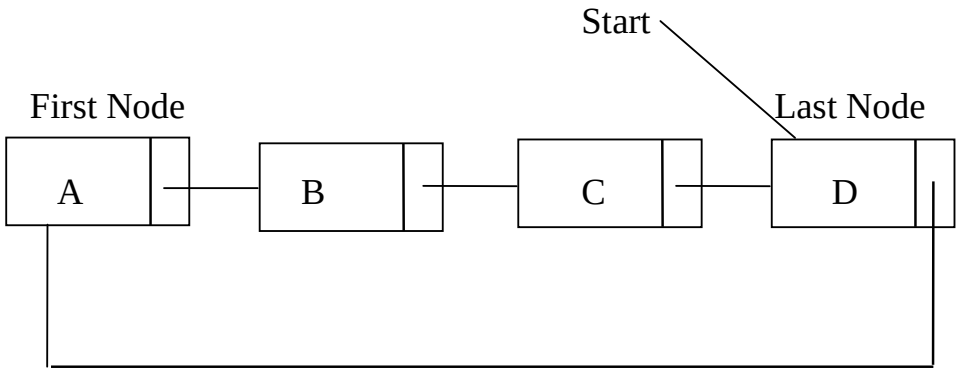
p = q = Start
Start = Start->link;
free(q);

- Case 2: An element different than the first one in the list

Same as in the case of one-way linked list.

5. Singly Circular Linked Lists

- **An example:**

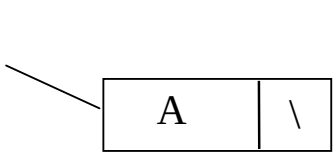


5.1. Operations: **Sorted or Unsorted**

- Insertion
- Deletion
- Find

5.2. Insertion in an unsorted list:

- Creation of a node:



- Case 1: When the list is not empty; start != nil;

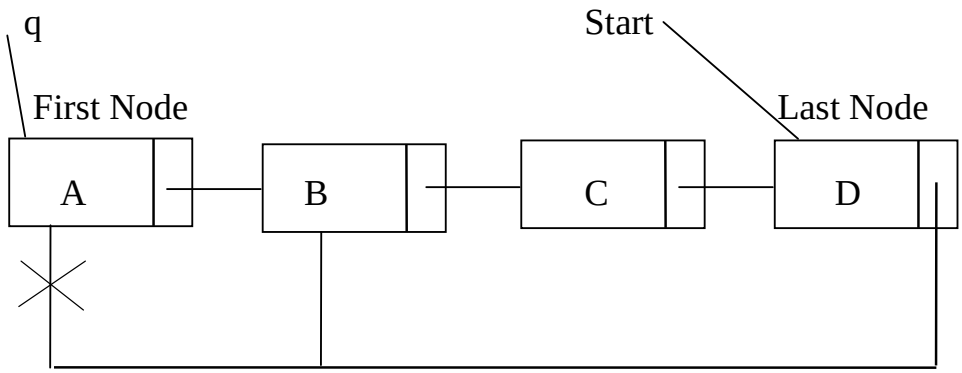
```
p->link = start;
start->link = p;
start = p;
```

- Case 2:When the list is empty; start=nil.

```
start = p;
p-link = p;
```

5.3. Deletion

- Case 1: The first element of the list



Delete(A):

```
q = start->link;
Start->link = q->link;
free(q);
```

- Case 2: The last element of the list

Do as Homework.

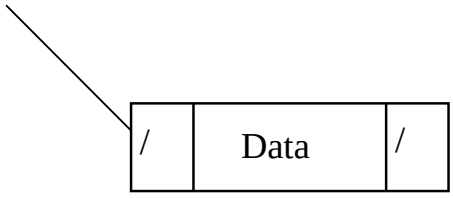
- Case 3: An element different from the first and last element of the list.

Same as in the case of one-way linked list.

6. Doubly Linked Lists

- Type of nodes: Declaration of a node:

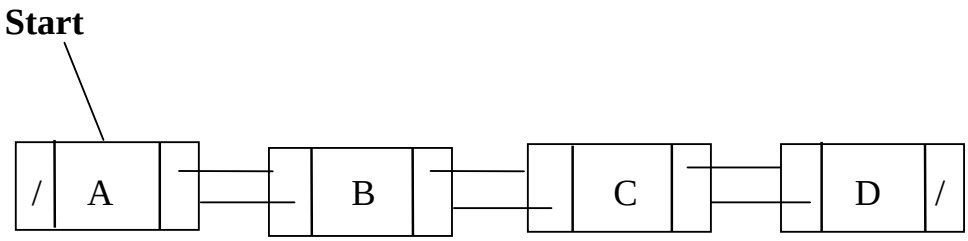
```
struct nodetype{
        struct nodetype *Llink;
        element_type data;
        struct nodetype *Rlink;
        };
```



- Types:
  - Simple doubly linked lists
  - Circular doubly linked lists

7. Simple doubly linked lists

- **An example**

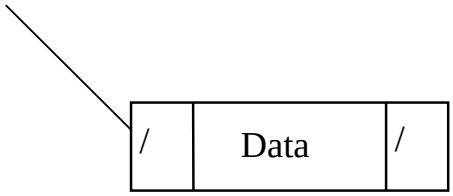


7.1. Operations: **Sorted or Unsorted**

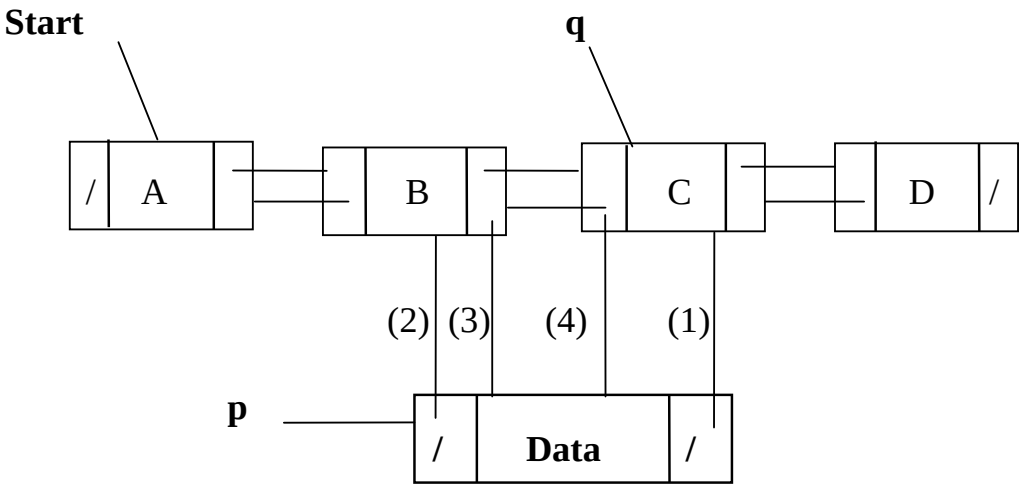
- Insertion
- Deletion
- Find

7.2. Insertion in an unsorted list:

- Creation of a node:



- Case 1: When the list is not empty; start != nil;



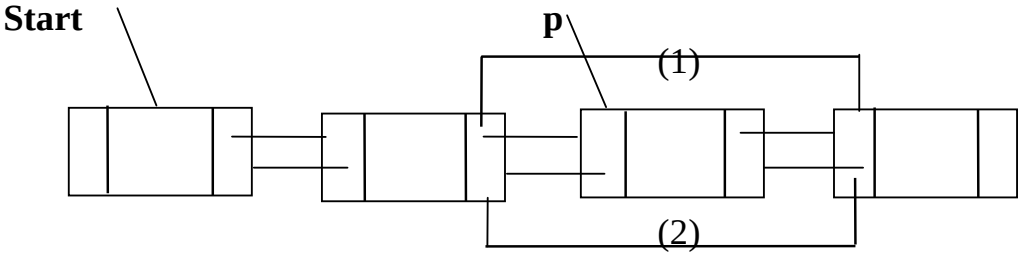
(1) p->Rlink = q;
(2) p->Llink = q->LLink;
(3) q->Llink_>Rlink = p;
(4) q->Llink = p;

- Case 2: When the list is empty; start=nil.

start = p;

7.3. Deletion

- Case 1: An element different from the first and last element of the list.



Delete(C):

```
p->Llink->Rlink = p->Rlink;
p->Rlink->Llink = p->Llink;
free(q);
```

- Case 2: The first element of the list.

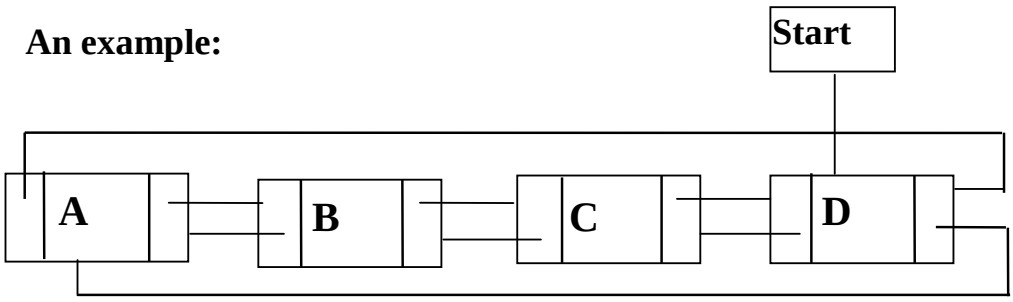
  Do as Homework.

- Case 3: The last element of the list.

  Do as Homework.

8. Circular doubly linked lists

- **An example:**



8.1. Operations: **Sorted or Unsorted**

- Insertion
- Deletion
- Find

8.2. Insertion in an unsorted list

- Case 1: When the list is not empty; start != nil;
- Case 2: When the list is empty; start=nil.

8.3. Deletion

- Case 1: An element different from the first and last element of the list.
- Case 2: The first element of the list.
- Case 3: The last element of the list.