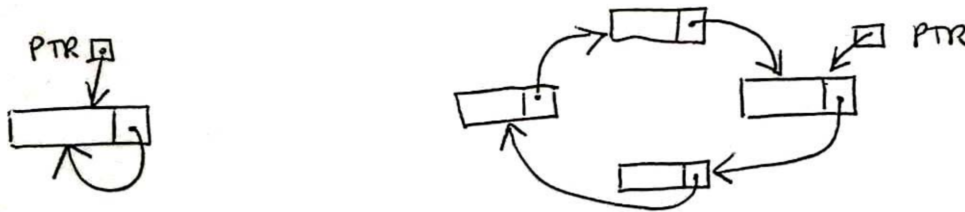
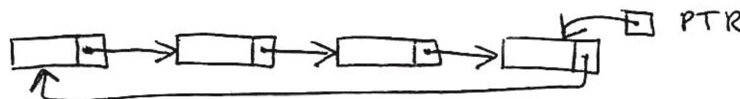
Singly-linked CIRCULAR lists

There is no `nil` pointer in a circular list. There is a pointer to the list, say `PTR`, which is `nil` for an empty list. The entire list may be accessed from any of its nodes.



For convenience we may draw the second of these lists thus:



enabling us to speak of the *left node* `*(PTR↑LINK)` and the *right node* `*PTR`. Operations on a singly-linked circular list include the following:

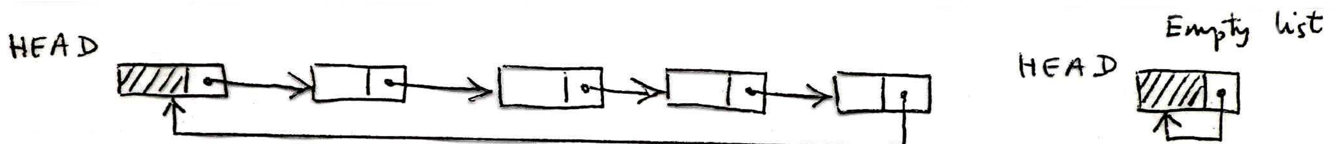
```
LEFTINSERT(PTR,y)

    P ← GETNODE();
    P↑INFO ← y;
    if (PTR = nil)  PTR ← P;
    else            P↑LINK ← PTR↑LINK;
    PTR↑LINK ← P;


RIGHTINSERT(PTR,y)

//  As LEFTINSERT, then
    PTR ← P;


LEFTDELETE(PTR)

    if (PTR = nil)
        deal_with_UNDERFLOW();
    else
    {   P ← PTR↑LINK;
        if (PTR = P)  PTR ← nil;
        else          PTR↑LINK ← P↑LINK;
        Result  ← P↑INFO;
        RELEASE(P);
        return Result;
    }
```

Operations `LEFTINSERT` and `LEFTDELETE` give the effect of a stack; `RIGHTINSERT` and `LEFTDELETE` the effect of a queue; all three the effect of an output-restricted deque. Once again, note that `RIGHTDELETE` (the extra operation needed to give the effect of a general deque) would be inefficient.

```
ERASE(PTR) -- Erase a complete circular list

    if (PTR ≠ nil)              /* Swap AVAIL and PTR↑LINK */
    {   P ← AVAIL;
        AVAIL ← PTR↑LINK;
        PTR↑LINK ← P;
        PTR ← nil;
    }
```

```
CONCAT(PTR1,PTR2) -- If PTR1 and PTR2 point to two disjoint circular lists,
                     this algorithm appends the second at the right of the first.

    if (PTR2 ≠ nil)             /* Swap PTR1↑LINK and PTR2↑LINK */
    {   if (PTR1 ≠ nil)
        {   P ← PTR1↑LINK;
            PTR1↑LINK ← PTR2↑LINK;
            PTR2↑LINK ← P;
        }
        PTR1 ← PTR2;
        PTR2 ← nil;
    }
```

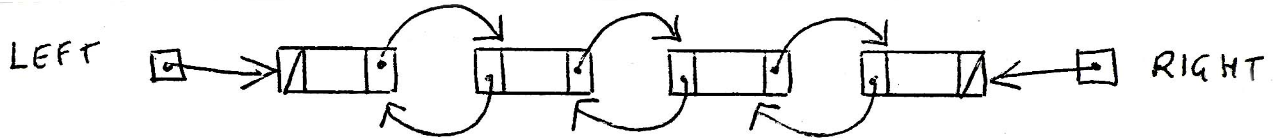
The pointer variable PTR may be replaced by a special node known as a *list head*, which is never deleted. (An empty list consists only of a list head.) This gives the list a clearly defined "end" and makes it easier to handle an empty list. It may also be possible to hold some useful information about the list in the INFO field of the list head, such as the number of nodes currently in the list. However, the insert and delete operations are efficient *only at one end* (Which end?)



Note that HEAD is the name of a node, not of a pointer. It occupies a fixed place in store. The pointer field of the HEAD node is HEAD.LINK, not HEAD↑LINK. To get a pointer pointing at the list head, in C++ and C you may use &HEAD, e.g., P ←&HEAD. This is not possible in some languages, e.g., Java and Pascal.

Doubly-linked lists

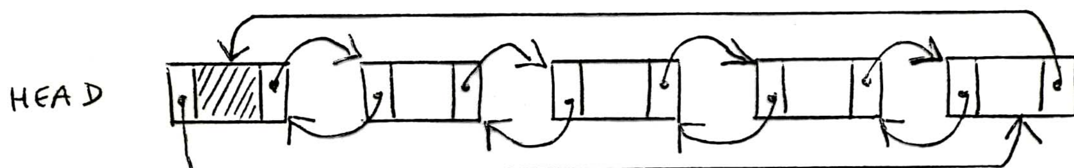
We have already noted the inefficiency involved in implementing a general deque using a singly-linked list, whether circular or not. To meet this problem, we may use a *doubly-linked list* pointed to by *two* pointer variables `LEFT` and `RIGHT`.



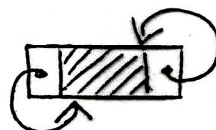
For an empty list, `LEFT = RIGHT = nil`. Each node contains two pointers `LLINK` and `RLINK`.

```
LEFTINSERT(LEFT,RIGHT,y)

    P ← GETNODE();
    P↑INFO ← y;
    P↑LLINK ← nil;
    P↑RLINK ← LEFT;
    if (LEFT = nil) RIGHT ← P;
    else            LEFT↑LLINK ← P;
    LEFT ← P;


LEFTDELETE(LEFT,RIGHT)

    if (LEFT = nil)
        deal_with_UNDERFLOW();
    else
    {   P ← LEFT;
        LEFT ← P↑RLINK;
        if (LEFT = nil) RIGHT ← nil;
        else            LEFT↑LLINK ← nil;
        Result ← P↑INFO;
        RELEASE(P);
        return Result;
    }
```

Because of the symmetry of a doubly-linked list, the insert and delete operations at the right are similar. We thus have all the operations needed for a general deque. But perhaps the best arrangement for a doubly-linked list is to make it *circular* and use a list head, thus:



An empty list consists only of a list head:



Note: `X↑LLINK↑RLINK = X↑RLINK↑LLINK = X` where `X` is a pointer to any node, including the list head.

When compared with a singly-linked list, a doubly-linked list has the following (dis)advantages:

1. Uses more storage (two links per node).
2. Algorithms somewhat slower.
3. Allows easy scanning in either direction from any given node.
4. Allows easy insertion and deletion anywhere in the list.

To illustrate (4), consider a pointer variable `X` pointing to a node to be deleted from a list (assuming use of a list head):

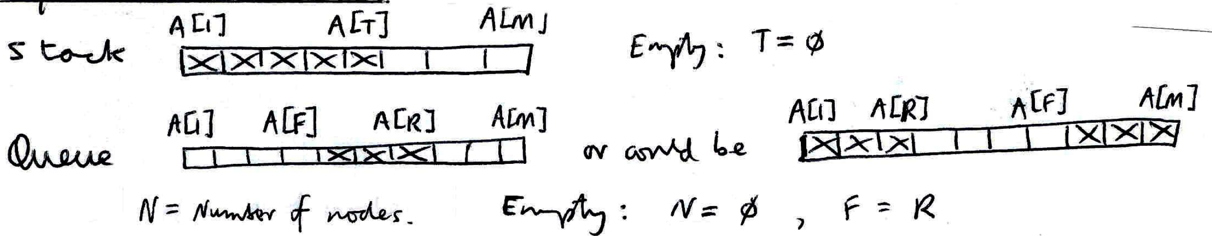
```
X↑LLINK↑RLINK  ← X↑RLINK;
X↑RLINK↑LLINK  ← X↑LLINK;
Result ← X↑INFO;
RELEASE(X);
return Result;
```

To insert a node to the right of the node pointed to by `X`:

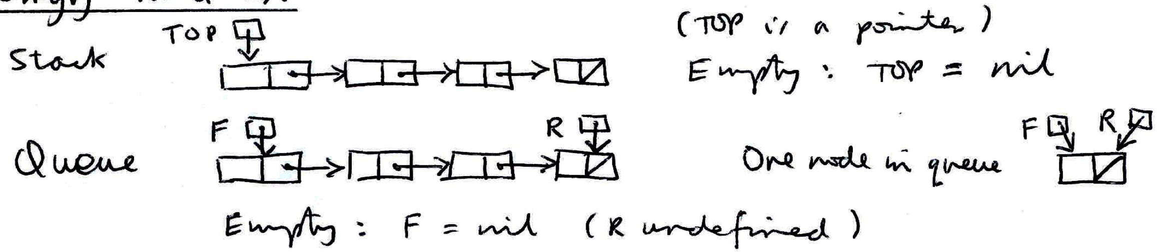
```
P ← GETNODE();
P↑INFO  ← y;
P↑LLINK ← X;
P↑RLINK  ← X↑RLINK;
X↑RLINK↑LLINK ← P;
X↑RLINK ← P;
```

Summary of linear lists

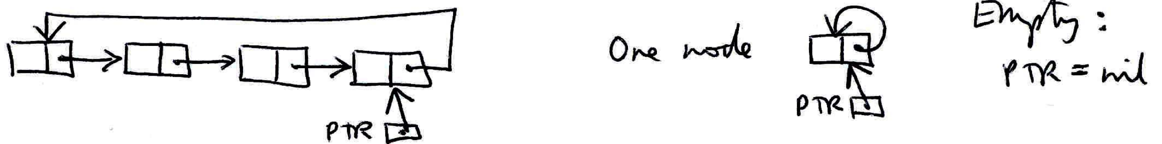
Sequential allocation



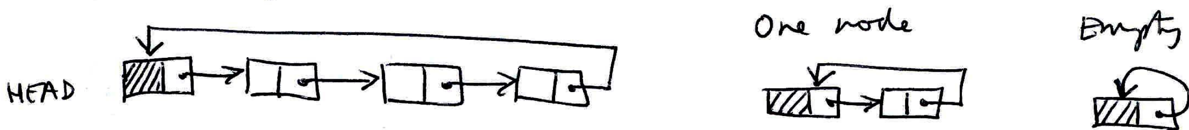
Singly-linked list



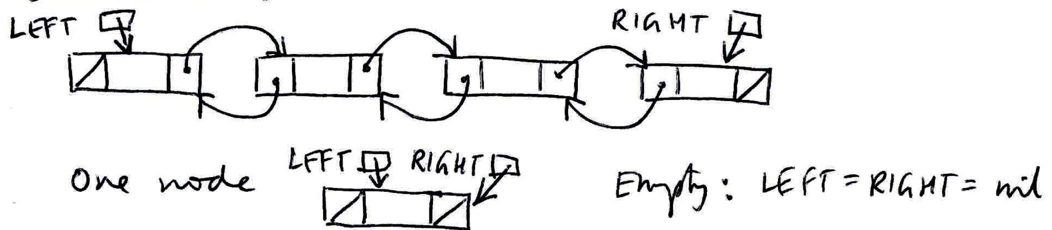
Circular singly-linked list



Circular singly-linked list with head



Doubly-linked list



Circular doubly-linked list with head