
Data Structure: List

chap. 4.1-4.2, 4.8

Yongjun Park
Hanyang University



List ADT

- an ordered sequence of element $\langle A_1, A_2, A_3, \dots, A_N \rangle$
 - the size of the list is N
 - a list of size 0 is an empty list
 - A_{i+1} follows (succeeds) A_i ($i < N$) and A_{i-1} precedes A_i ($i > 1$)
 - the position of an element A_i in a list is i
- operations in the List ADT
 - MakeEmpty (List L): constructor
 - DeleteList (List L): destructor
 - Find (List L, Key K): returns the position of the key
 - Insert (Key K, List L, Position P): insert K after P in L
 - Delete (Key K, List L): delete K from L
 - Concat (List L1, List L2): returns the concatenation of L1 and L2



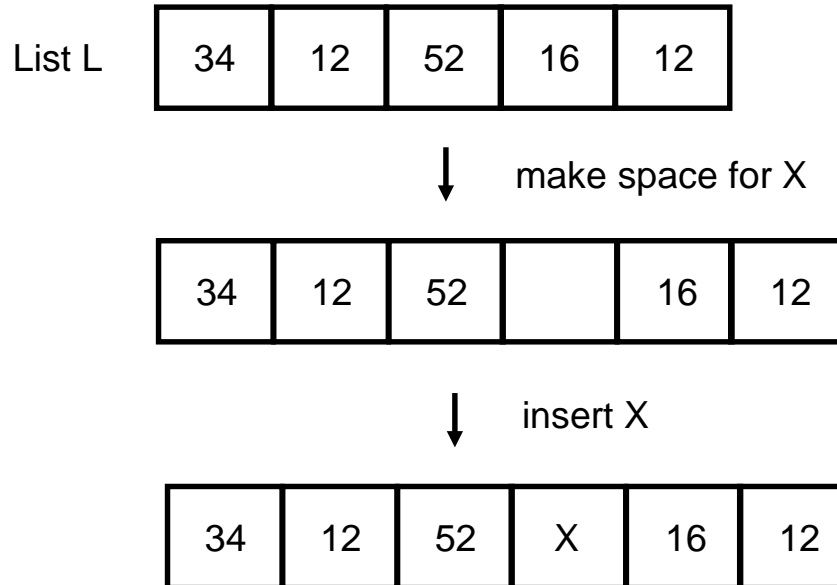
List ADT: an example

- List: $L = \langle 34, 12, 52, 16, 12 \rangle$
 - Find($L, 52$): 3
 - Insert($X, L, 3$): 34, 12, 52, X, 16, 12
 - Delete($52, L$): 34, 12, X, 16, 12

- Find (List L , Key K): returns the position of the key
- Insert (Key K , List L , Position P): insert K after P in L
- Delete (Key K , List L): delete K from L

List ADT: simple implementation with array

Insert X after the position 3 in the list L



List ADT: simple implementation with **array**

- it is inefficient because . . .

-

-

-

-

-

-



List ADT: simple implementation with **array**

- it is inefficient because . . .
 - an estimate of the maximum size of the list is required
 - it requires overestimating the amount of storage needed for the list
 - it is hard to insert or delete at the beginning or in the middle of the list
 - worst case: $O(N)$
 - average case: half of the list $O(N)$
 - building a list by N successive inserts: $O(N^2)$

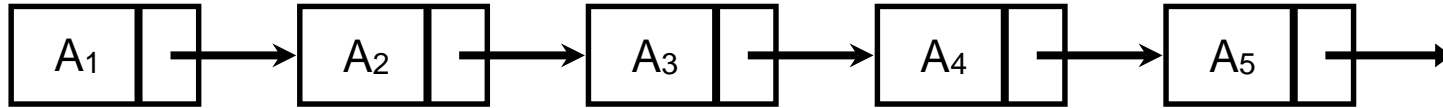


List ADT: **linked list**

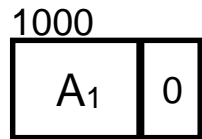
- a linked list consists of a series of structures, which are **not necessarily adjacent in memory**
- each structure contains **an element and a pointer** to a structure of its successor
- the last cell's pointer points to NULL



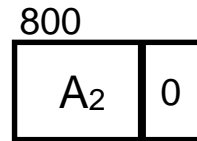
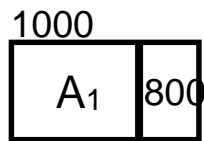
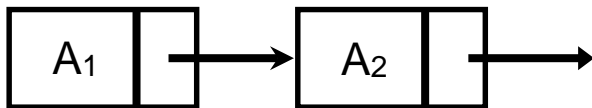
List ADT: **linked list**



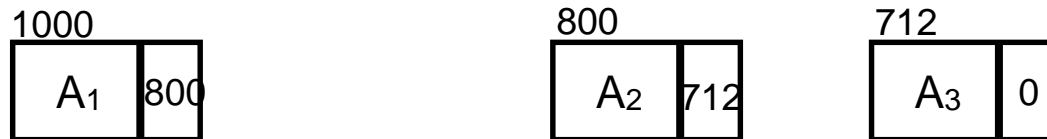
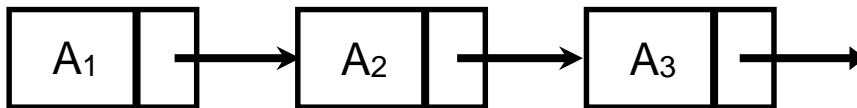
List ADT: **linked list**



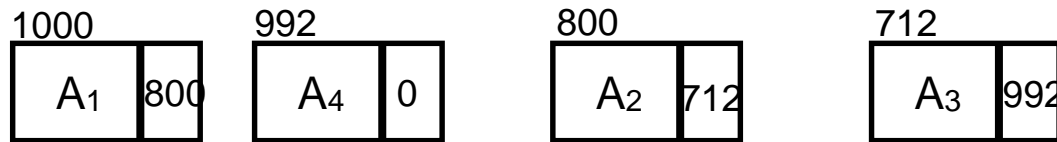
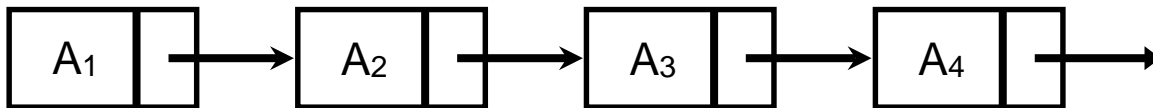
List ADT: **linked list**



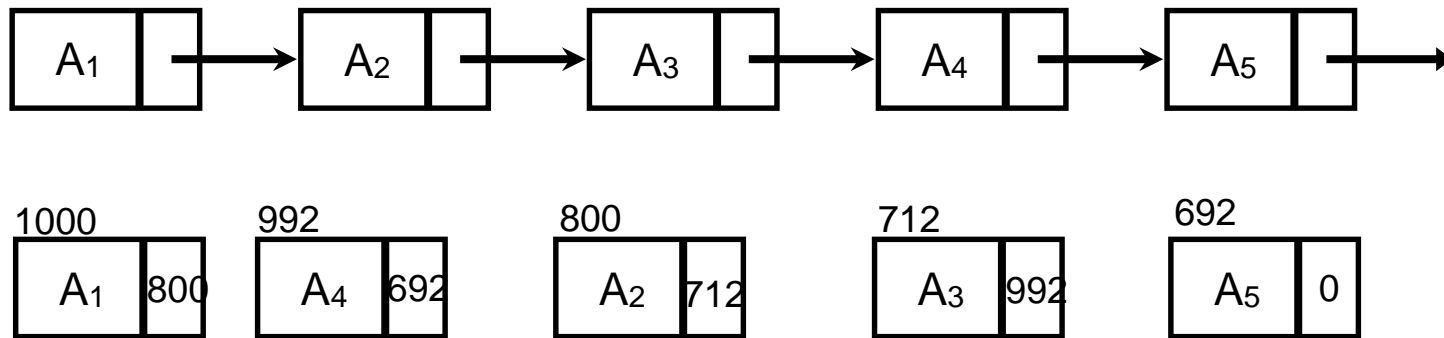
List ADT: linked list



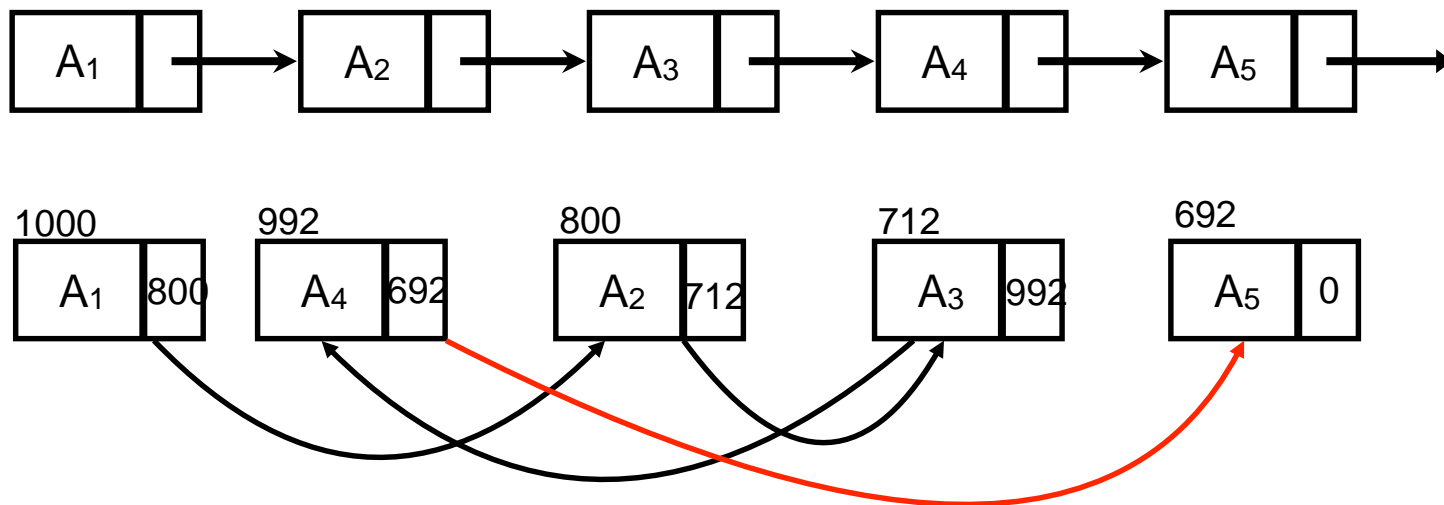
List ADT: **linked list**



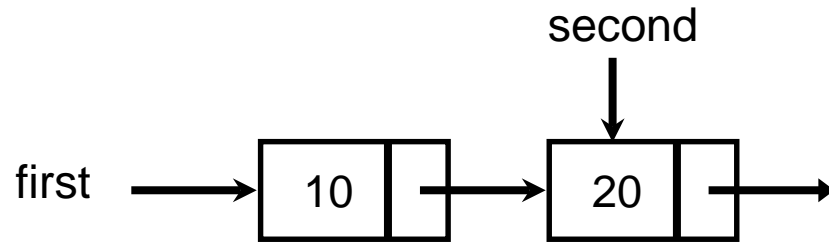
List ADT: **linked list**



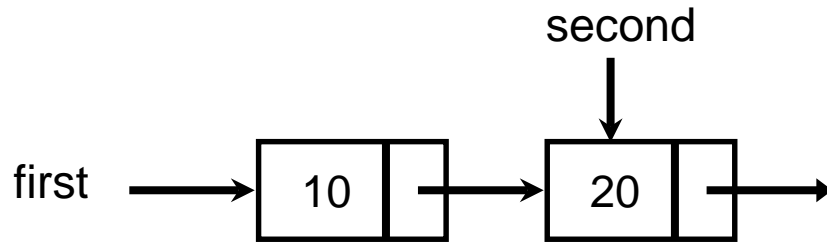
List ADT: **linked list**



List ADT: example



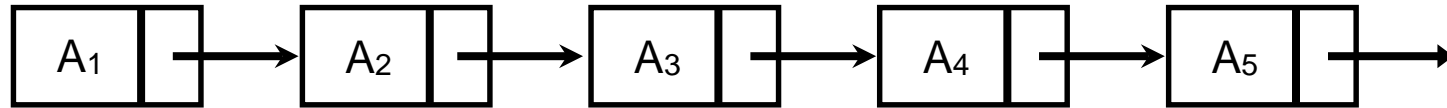
List ADT: example



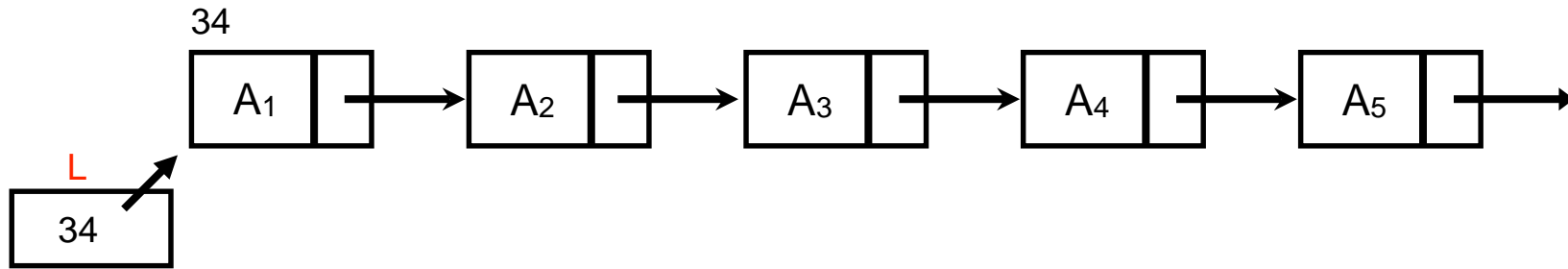
```
typedef struct Node *PtrToNode;
typedef struct Node {
    int data;
    PtrToNode link;
};

PtrToNode create2(){
    PtrToNode first, second;
    MALLOC(first, sizeof(*first));
    MALLOC(second, sizeof(*second));
    second -> link = NULL;
    second -> data = 20;
    first -> data = 10;
    first -> link = second;
    Return first;
}
```

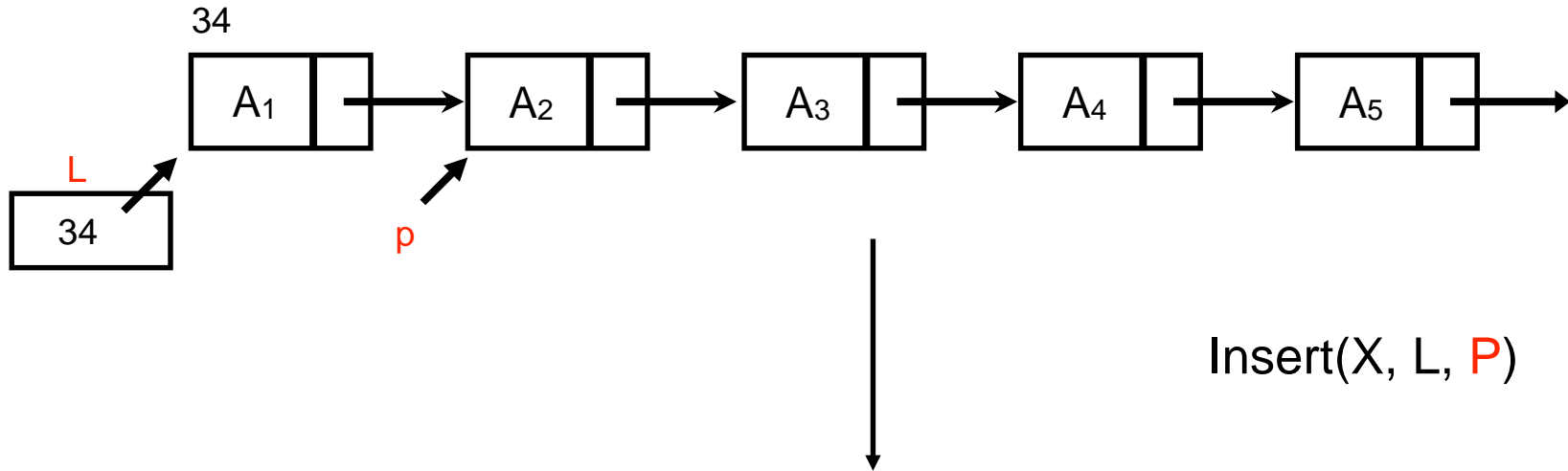

List ADT: insertion



List ADT: insertion



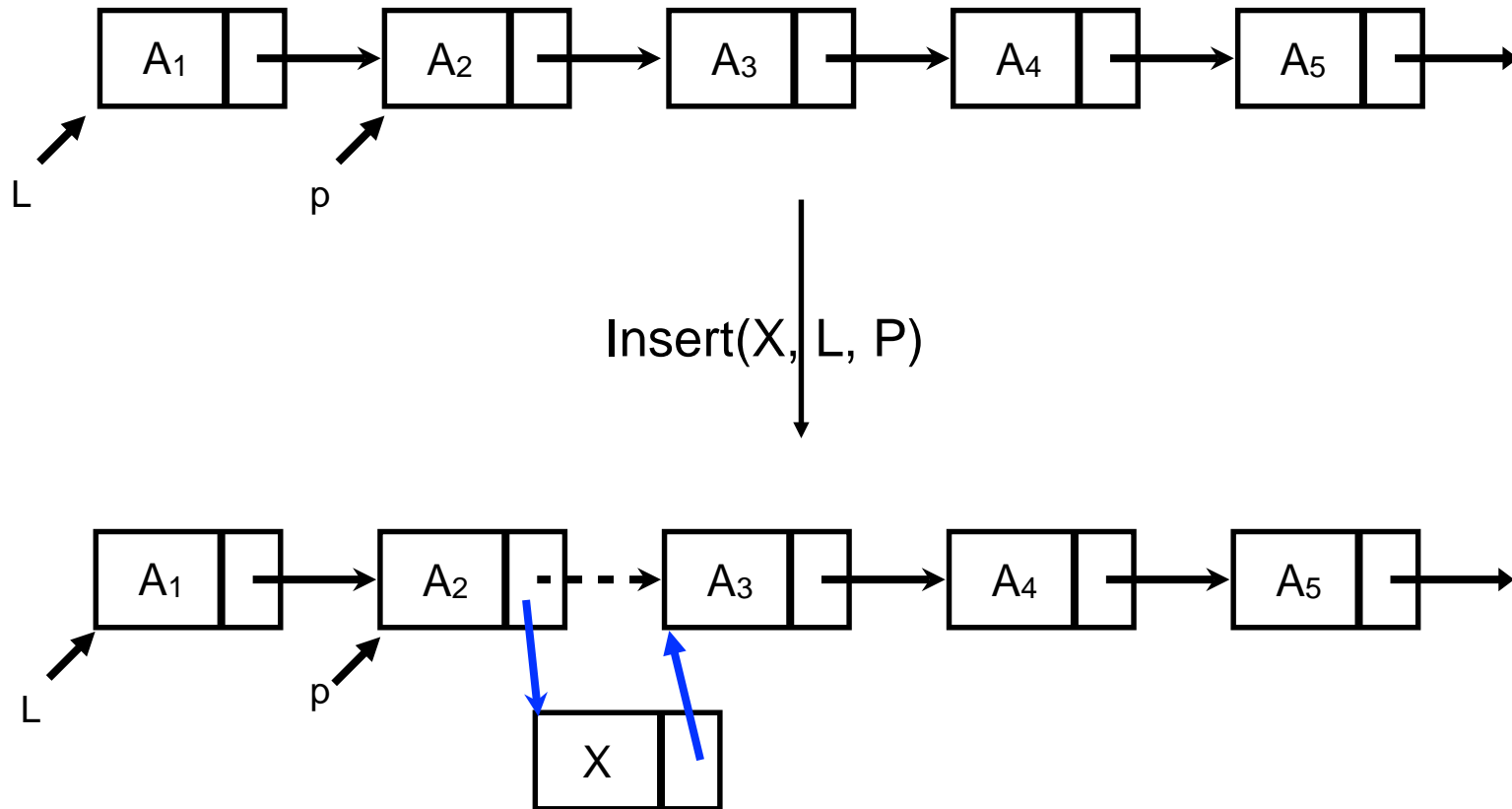
List ADT: insertion



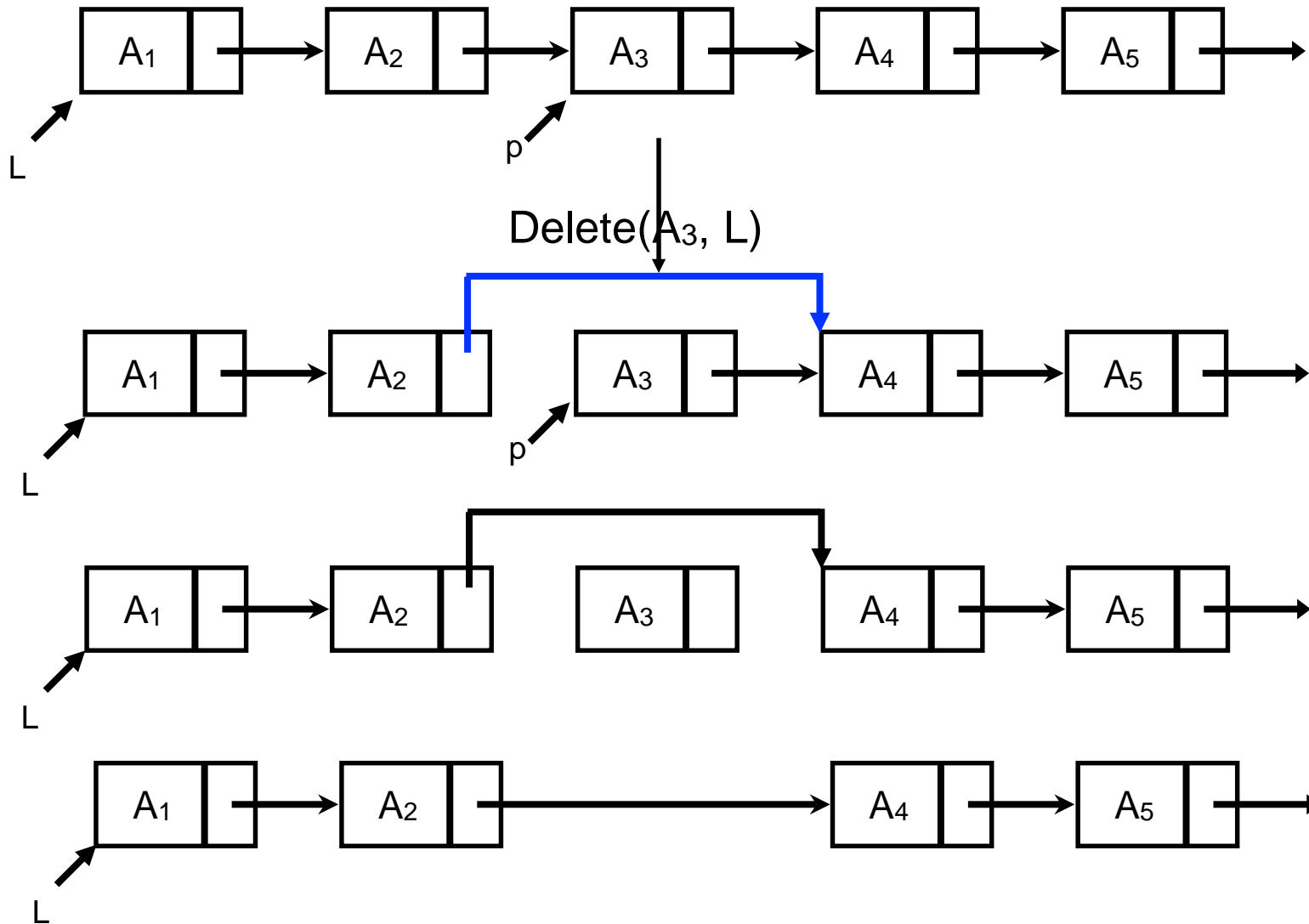
Insert(X, L, P)

- Insert (Key K, List L, Position P): insert K after P in L

List ADT: insertion



List ADT: deletion



list ADT: type declaration for a linked list

```
typedef struct Node* PtrToNode;
typedef int ElementType;
typedef PtrToNode Position;
typedef PtrToNode List;
struct Node
{
    ElementType Element;
    Position Next;
};

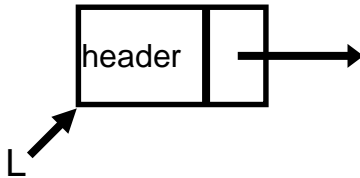
List MakeEmpty( List L );
int IsEmpty( List L );
int IsLast( Position P, List L );
Position Find( ElementType X, List L );
Position FindPrevious ( ElementType X, List L );
void Delete( ElementType X, List L );
void Insert ( ElementType X, List L, Position P );
void DeleteList ( List L );
```

list ADT: MakeEmpty

/* create header node */

```
List MakeEmpty( List L )  
{  
    L = (List)malloc(sizeof(struct Node));  
    L->element = header;  
    L->next = NULL;  
    return L  
}
```

```
struct Node  
{  
    ElementType Element;  
    Position     Next;  
};
```

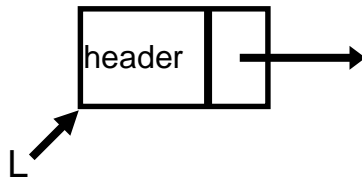


list ADT: IsEmpty

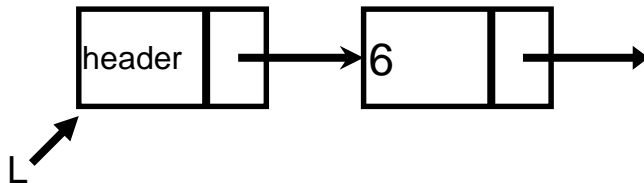
/* return true if L is empty */

```
int IsEmpty( List L )  
{  
  
}
```

```
struct Node  
{  
    ElementType Element;  
    Position     Next;  
};
```



true



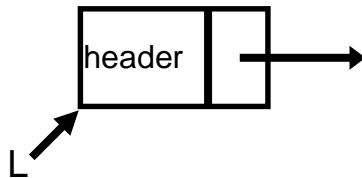
false

list ADT: IsEmpty

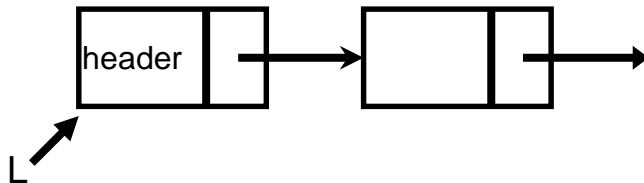
/* return true if L is empty */

```
int IsEmpty( List L )  
{  
    return L->Next == NULL;  
}
```

```
struct Node  
{  
    ElementType Element;  
    Position     Next;  
};
```



true



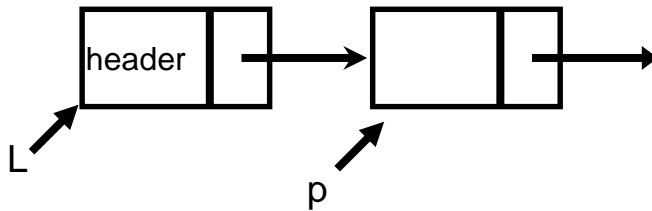
false

list ADT: IsLast

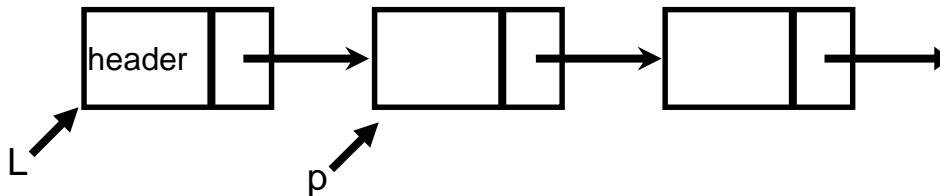
/* return true if P is the last position in list L */

```
int IsLast( Position P, List L )  
{  
  
}
```

```
struct Node  
{  
    ElementType Element;  
    Position Next;  
};
```



true



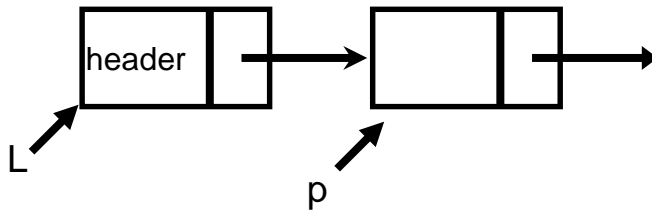
false

list ADT: IsLast

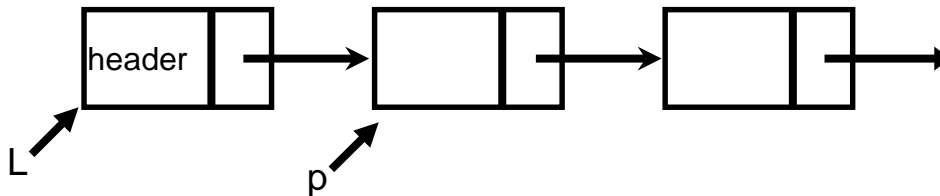
/* return true if P is the last position in list L */

```
int IsLast( Position P, List L )  
{  
    return P->Next == NULL;  
}
```

```
struct Node  
{  
    ElementType Element;  
    Position Next;  
};
```



true



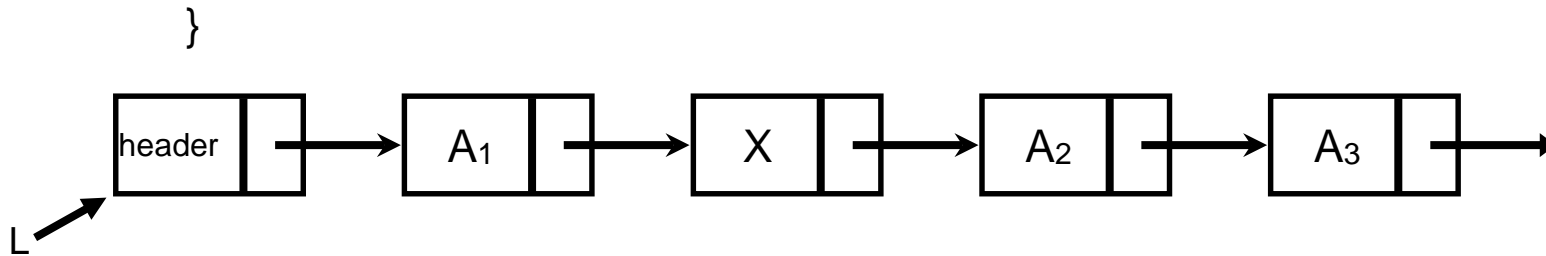
false

list ADT: Find

/* return position of X in L; NULL if not found */

Position Find(ElementType X, List L)
{

```
struct Node  
{  
    ElementType Element;  
    Position     Next;  
};
```



list ADT: Find

/* return position of X in L; NULL if not found */

Position Find(ElementType X, List L)

{

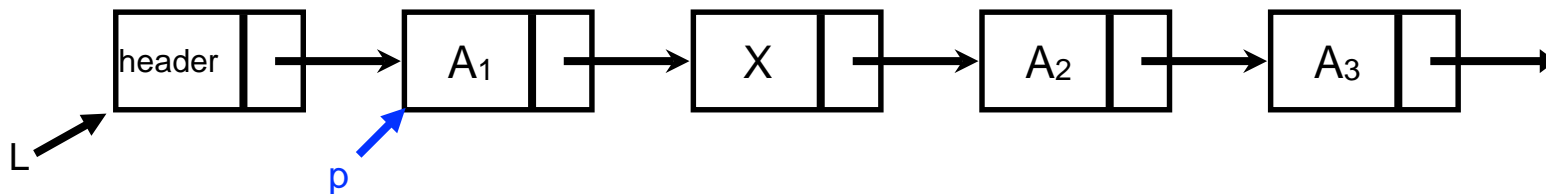
Position P;

P = L->Next;

return P;

}

```
struct Node
{
    ElementType Element;
    Position Next;
};
```



list ADT: Find

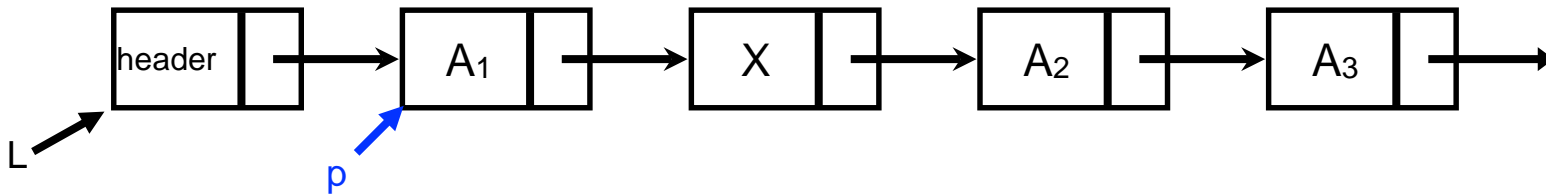
/* return position of X in L; NULL if not found */

```
Position Find( ElementType X, List L )
{
    Position P;

    P = L->Next;
    while( P != NULL && P->Element != X )

    return P;
}
```

```
struct Node
{
    ElementType Element;
    Position     Next;
};
```



list ADT: Find

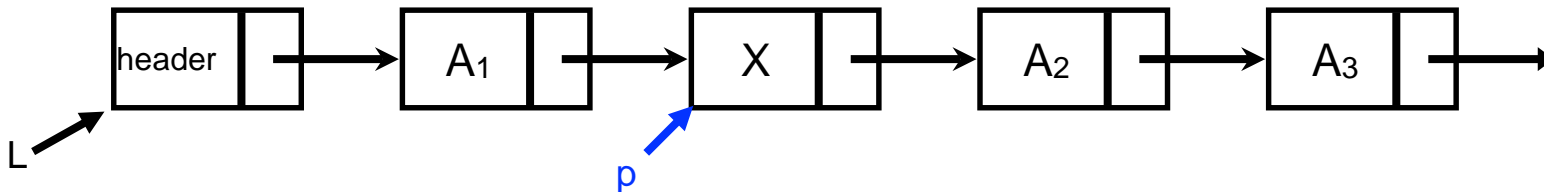
/* return position of X in L; NULL if not found */

```
Position Find( ElementType X, List L )
{
    Position P;

    P = L->Next;
    while( P != NULL && P->Element != X )
        P = P->Next;

    return P;
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
};
```



list ADT: Find

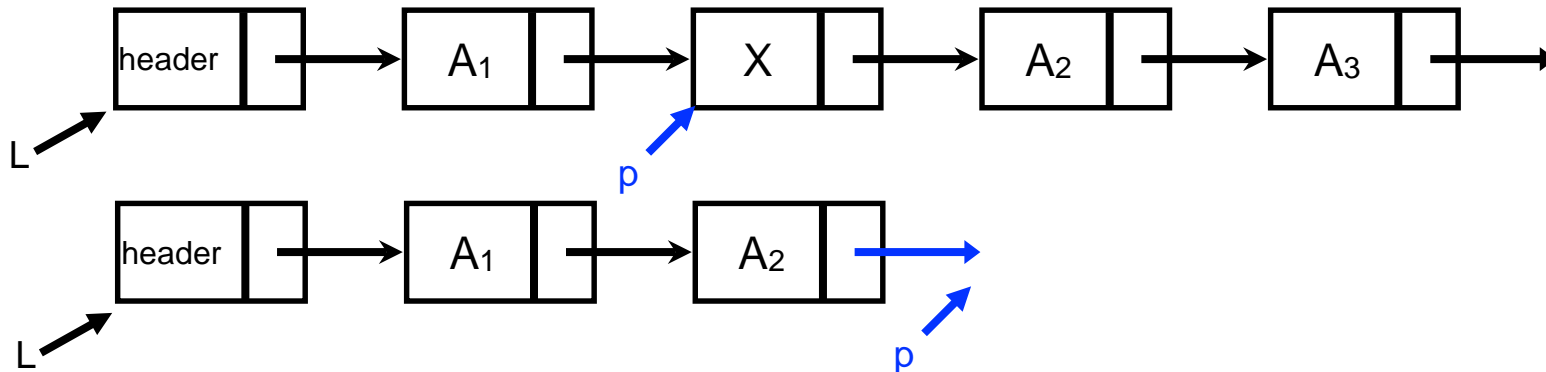
/* return position of X in L; NULL if not found */

```
Position Find( ElementType X, List L )
{
    Position P;

    P = L->Next;
    while( P != NULL && P->Element != X )
        P = P->Next;

    return P;
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
};
```

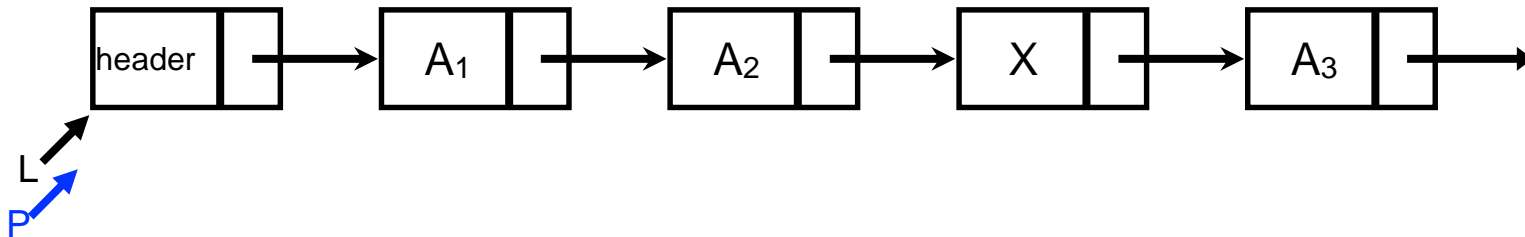
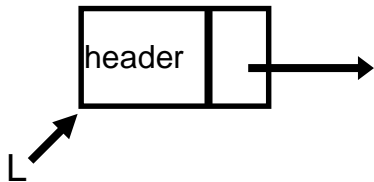


list ADT: FindPrevious

```
Position FindPrevious( ElementType X, List L )
{
    Position P;

    P = L;
    while( P->Next != NULL &&
           P->Next->Element != X )
        P = P->Next;
    return P;
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
};
```

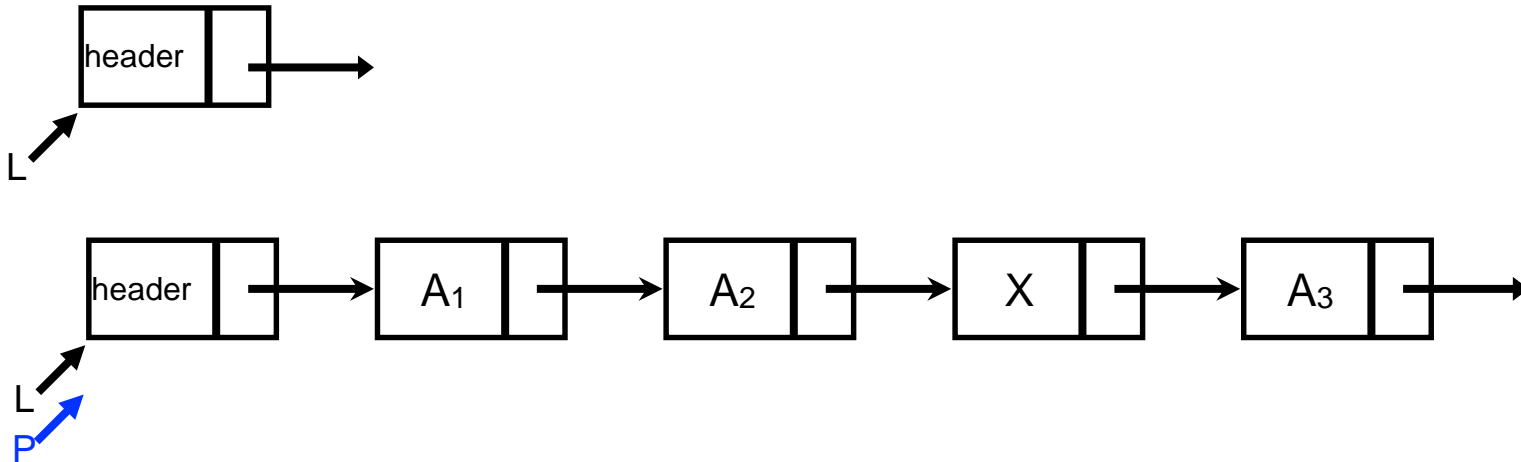


list ADT: FindPrevious

```
Position FindPrevious( ElementType X, List L )
{
    Position P;

    P = L;
    while( P->Next != NULL && P->Next->Element != X )
        P = P->Next;
    return P;
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
};
```

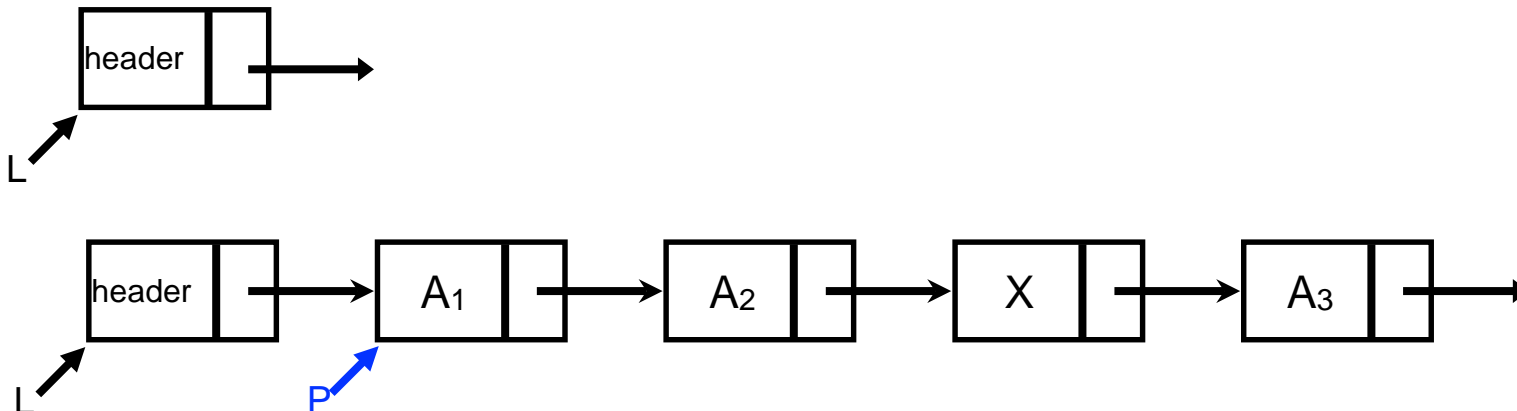


list ADT: FindPrevious

```
Position FindPrevious( ElementType X, List L )
{
    Position P;

    P = L;
    while( P->Next != NULL && P->Next->Element != X )
        P = P->Next;
    return P;
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
};
```

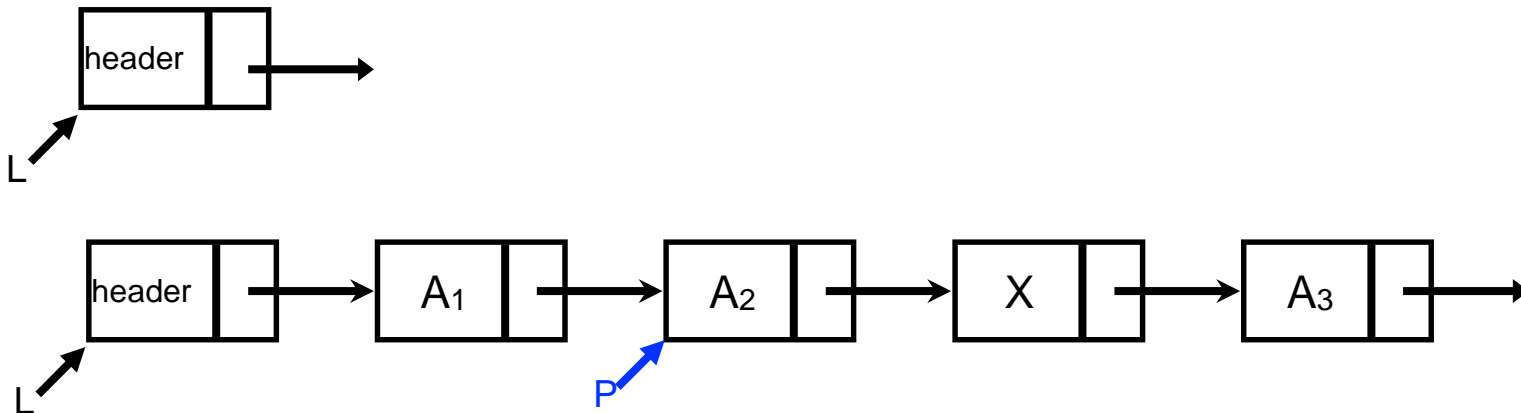


list ADT: FindPrevious

```
Position FindPrevious( ElementType X, List L )
{
    Position P;

    P = L;
    while( P->Next != NULL && P->Next->Element != X )
        P = P->Next;
    return P;
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
};
```

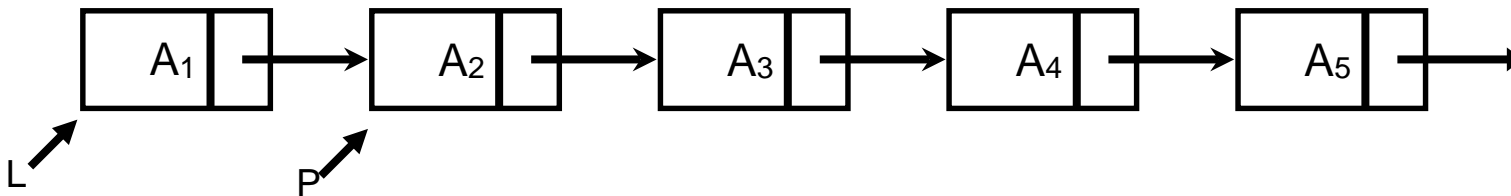


List ADT: Insert

```
void Insert( ElementType X, List L, Position P )  
{
```

```
struct Node  
{  
    ElementType Element;  
    Position Next;  
};
```

```
}
```

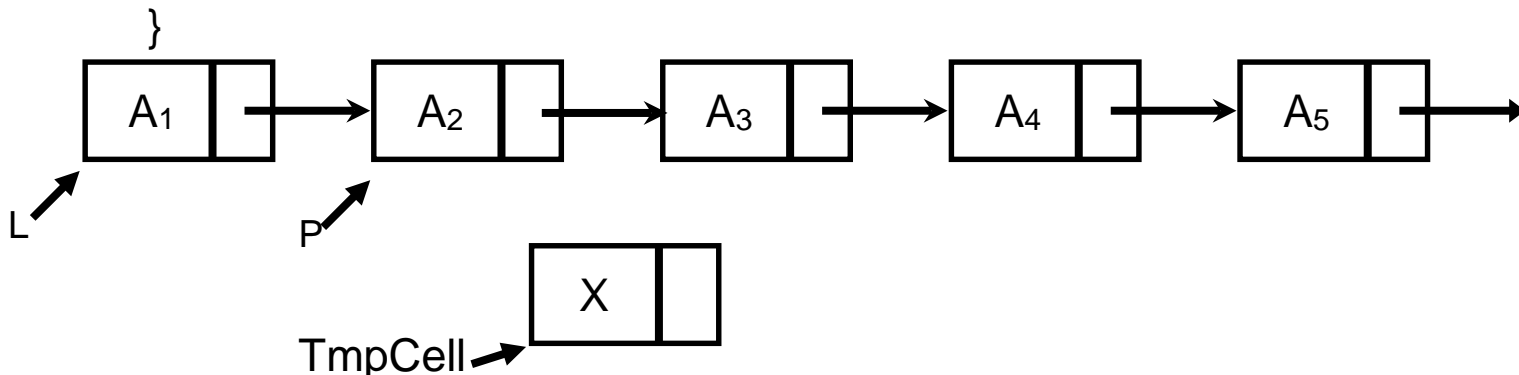


List ADT: Insert

```
void Insert( ElementType X, List L, Position P )  
{  
    Position TmpCell;  
  
    TmpCell = malloc( sizeof( struct Node ) );  
    if ( TmpCell == NULL )  
        FatalError( "Out of space!!!" );
```

```
struct Node  
{  
    ElementType Element;  
    Position Next;  
};
```

TmpCell->Element = X;



List ADT: Insert

```
void Insert( ElementType X, List L, Position P )
```

```
{
```

```
    Position TmpCell;
```

```
    TmpCell = malloc( sizeof( struct Node ) );
```

```
    if ( TmpCell == NULL )
```

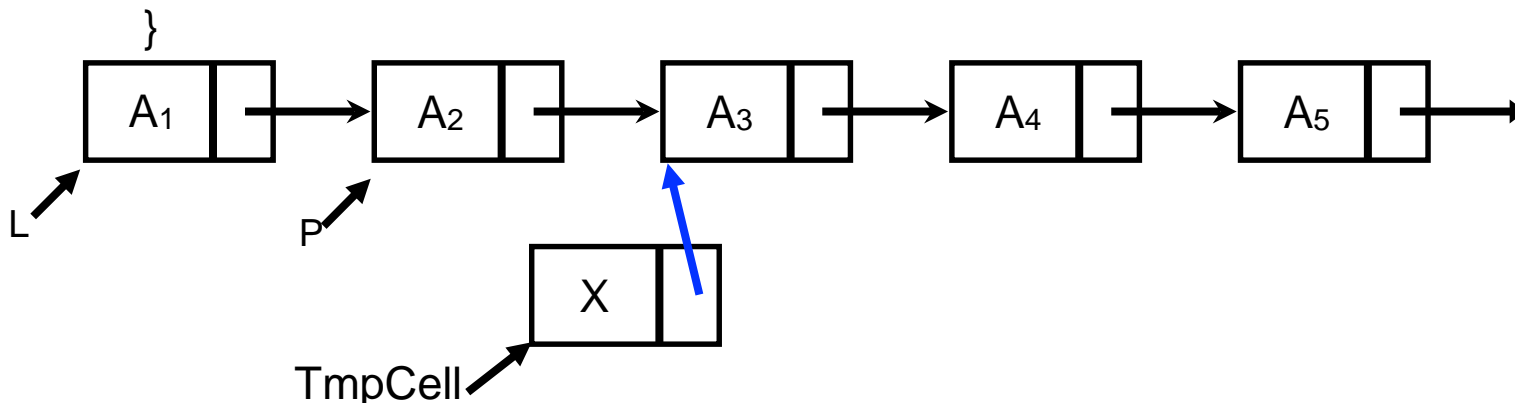
```
        FatalError( "Out of space!!" );
```

```
    TmpCell->Element = X;
```

```
    TmpCell->Next = P->Next;
```

```
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
};
```



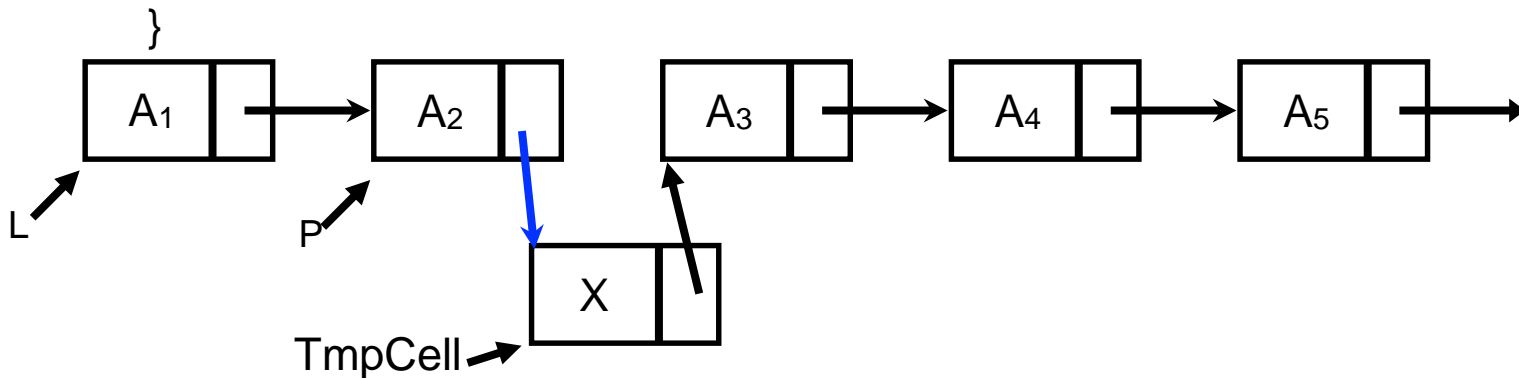
List ADT: Insert

```
void Insert( ElementType X, List L, Position P )
{
    Position TmpCell;

    TmpCell = malloc( sizeof( struct Node ) );
    if ( TmpCell == NULL )
        FatalError( "Out of space!!" );

    TmpCell->Element = X;
    TmpCell->Next = P->Next;
    P->Next = TmpCell;
}
```

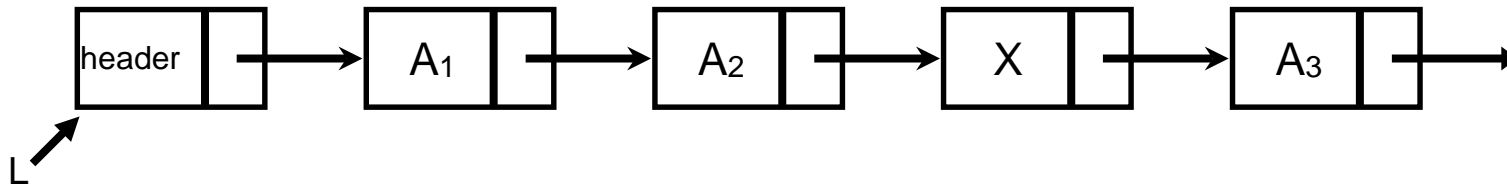
```
struct Node
{
    ElementType Element;
    Position Next;
};
```



list ADT: Delete

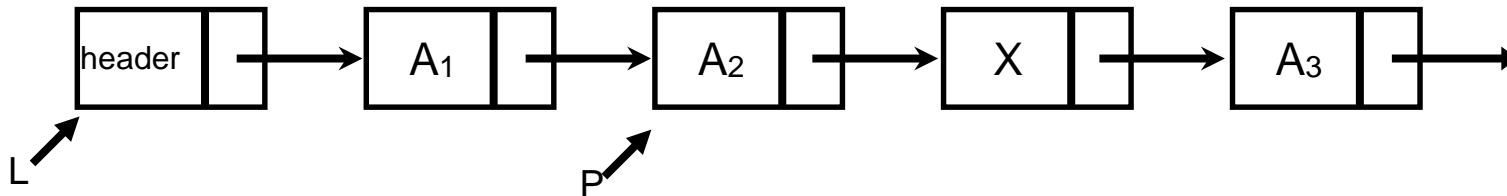
```
void Delete( ElementType X, List L )  
{
```

```
}
```



list ADT: Delete

```
void Delete( ElementType X, List L )  
{  
    Position P, TmpCell;  
  
    P = FindPrevious( X, L );  
  
}
```



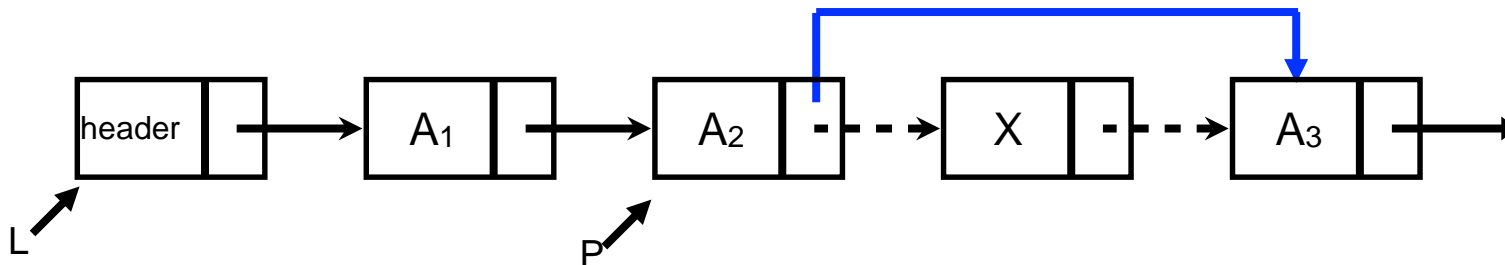
list ADT: Delete

```
void Delete( ElementType X, List L )
{
    Position P, TmpCell;

    P = FindPrevious( X, L );
    if ( !IsLast( P, L ) )           /* Assumption of header use */
    {                                 /* X is found; delete it */

        P->Next = ???; /* Bypass deleted cell */

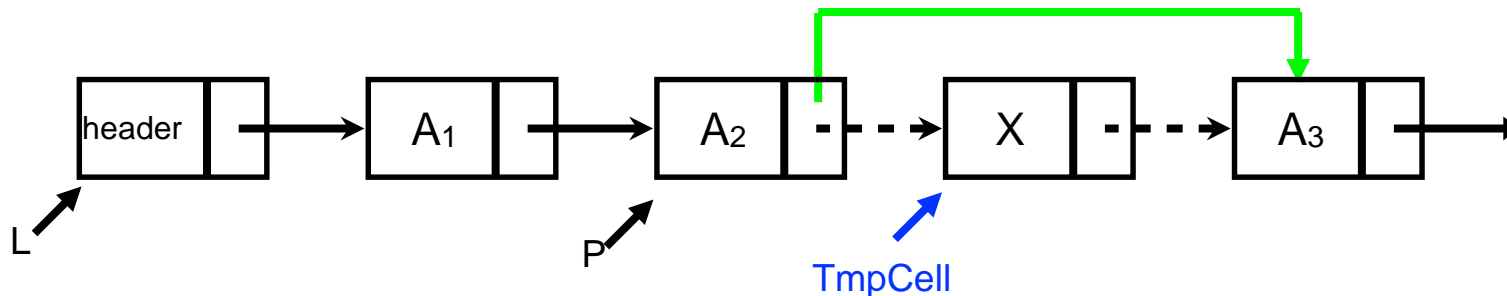
    }
}
```



list ADT: Delete

```
void Delete( ElementType X, List L )
{
    Position P, TmpCell;

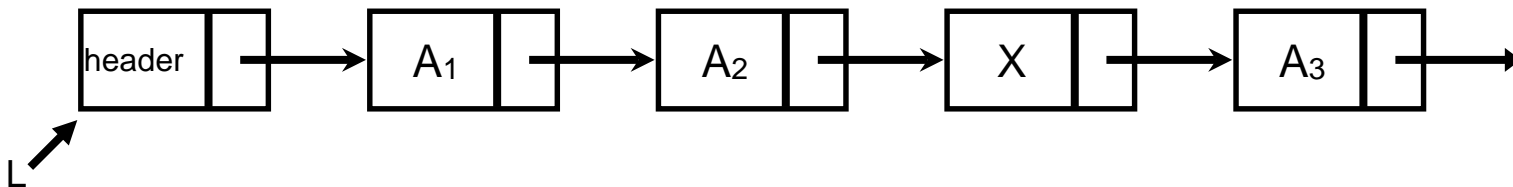
    P = FindPrevious( X, L );
    if ( !IsLast( P, L ) )           /* Assumption of header use */
    {                                 /* X is found; delete it */
        TmpCell = P->Next;
        P->Next = TmpCell->Next;    /* Bypass deleted cell */
        free( TmpCell );
    }
}
```



list ADT: DeleteList

```
void DeleteList( List L )  
{
```

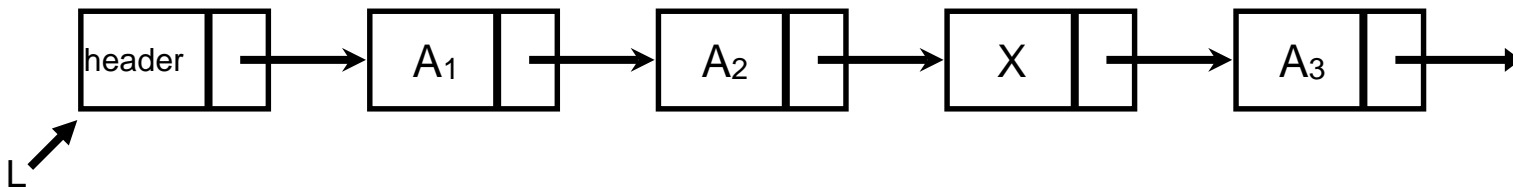
```
}
```



list ADT: DeleteList

```
void DeleteList( List L )  
{  
    Position P;  
  
    P = L->Next; /* Header assumed */  
    L->Next = NULL;  
    while( P != NULL )  
    {  
        free( P );  
        P = P->Next;  
    }  
}
```

Is it okay?



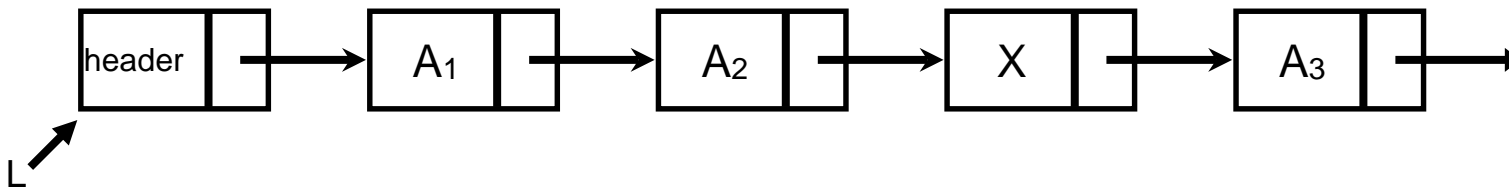
list ADT: DeleteList

```
void DeleteList( List L )
{
    Position P;

    P = L->Next; /* Header assumed */
    L->Next = NULL;
    while( P != NULL )
    {
        free( P );
        P = P->Next;
    }
}
```

```
void DeleteList( List L )
{
    Position P, Tmp;

    P = L->Next; /* Header assumed */
    L->Next = NULL;
    while( P != NULL )
    {
        Tmp = P->Next;
        free( P );
        P = Tmp;
    }
}
```

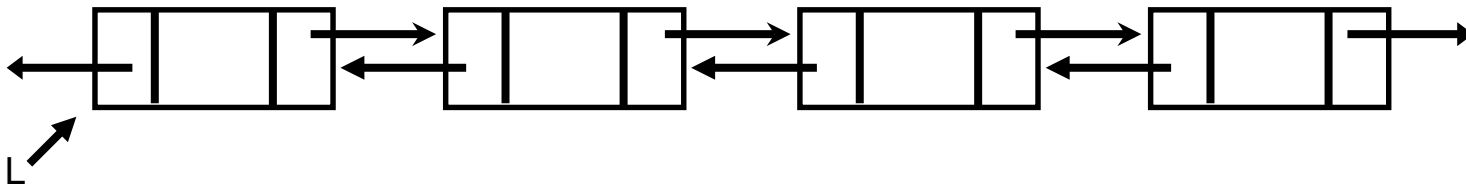


Doubly Linked List

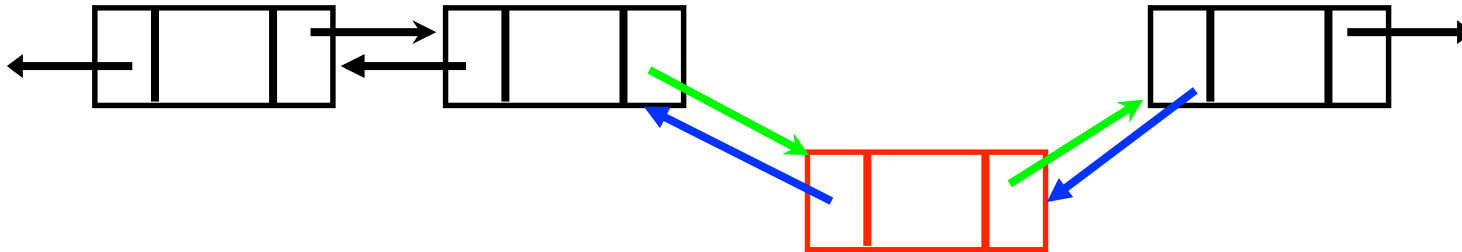
- a list that contains links to next and previous nodes

```
struct Node
{
    ElementType Element;
    Position    Next;
    Position    Prev;
};
```

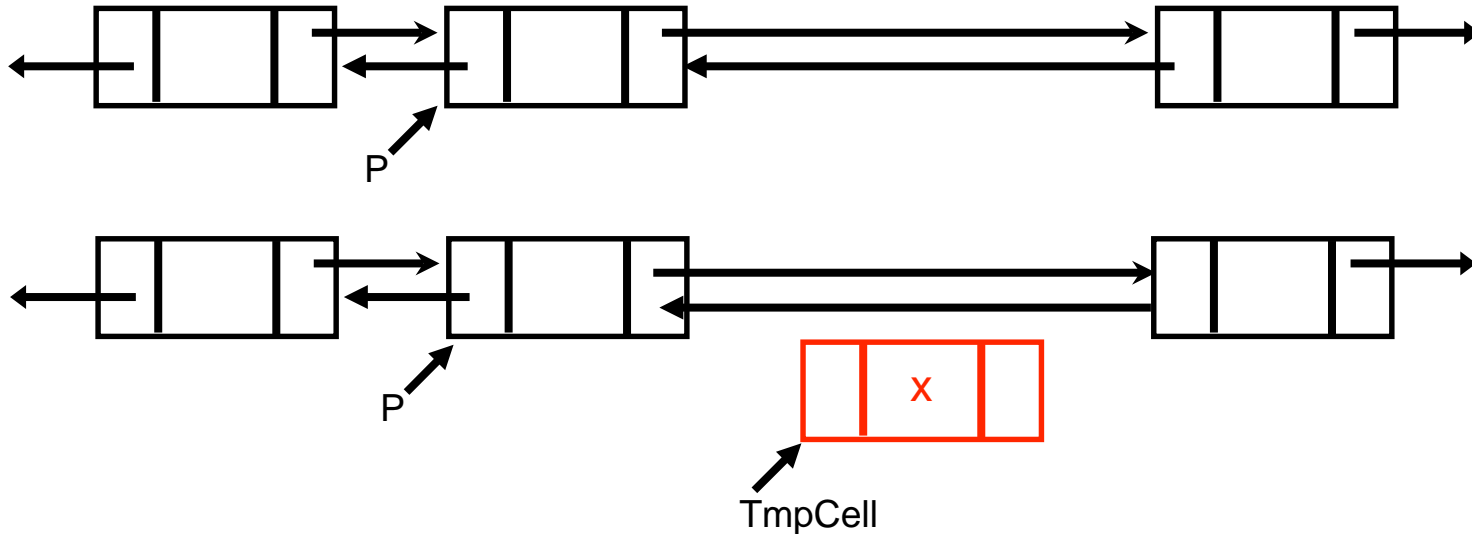
- FindPrevious() in singly linked list is not necessary



Doubly Linked List: insert



Doubly Linked List: insert



```
void Insert( ElementType X, List L, Position P )
{
    Position TmpCell;

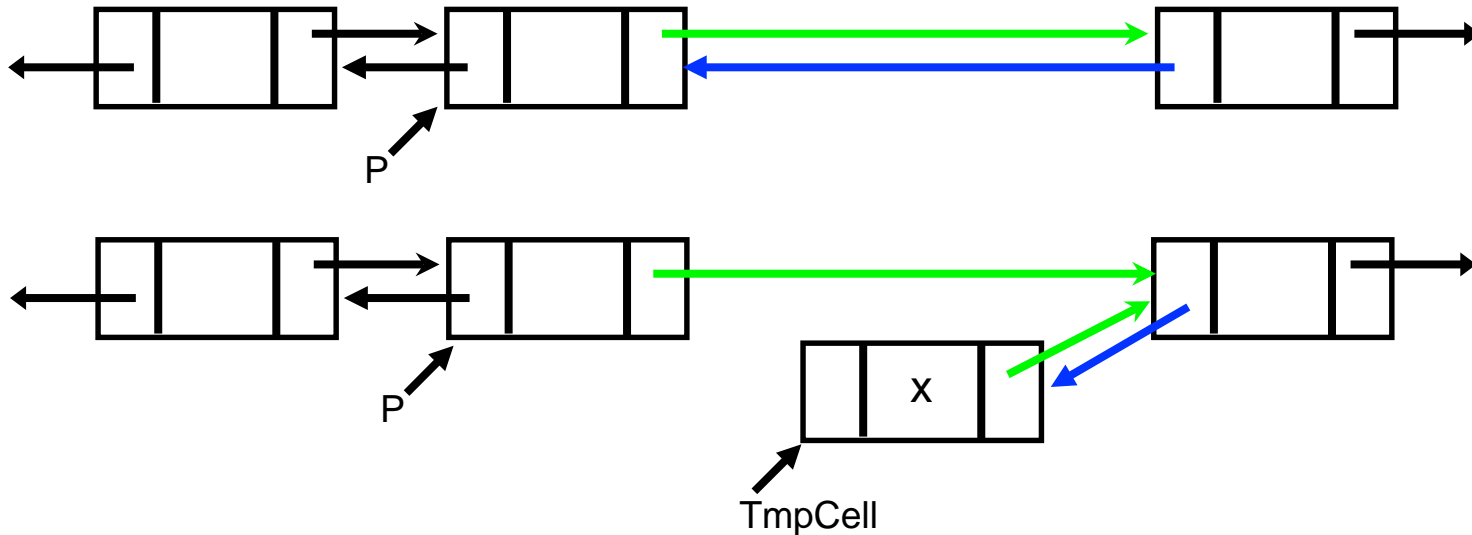
    TmpCell = malloc( sizeof( struct Node ) );
    if ( TmpCell == NULL )
        FatalError( "Out of space!!!" );

    TmpCell->Element = X;

}
```

```
struct Node
{
    ElementType Element;
    Position Next;
    Position Prev;
};
```

Doubly Linked List: insert



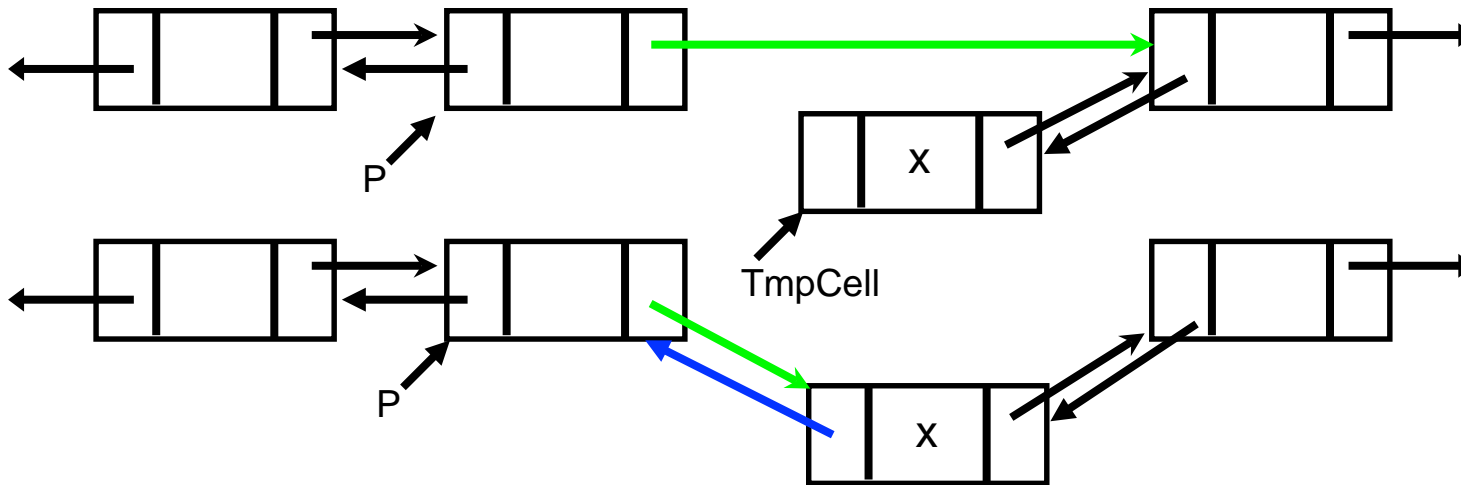
```
void Insert( ElementType X, List L, Position P )
{
    Position TmpCell;

    TmpCell = malloc( sizeof( struct Node ) );
    if ( TmpCell == NULL )
        FatalError( "Out of space!!!" );

    TmpCell->Element = X;
    TmpCell->Next = P->Next;
    TmpCell->Next->Prev = TmpCell;
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
    Position Prev;
};
```

Doubly Linked List: insert



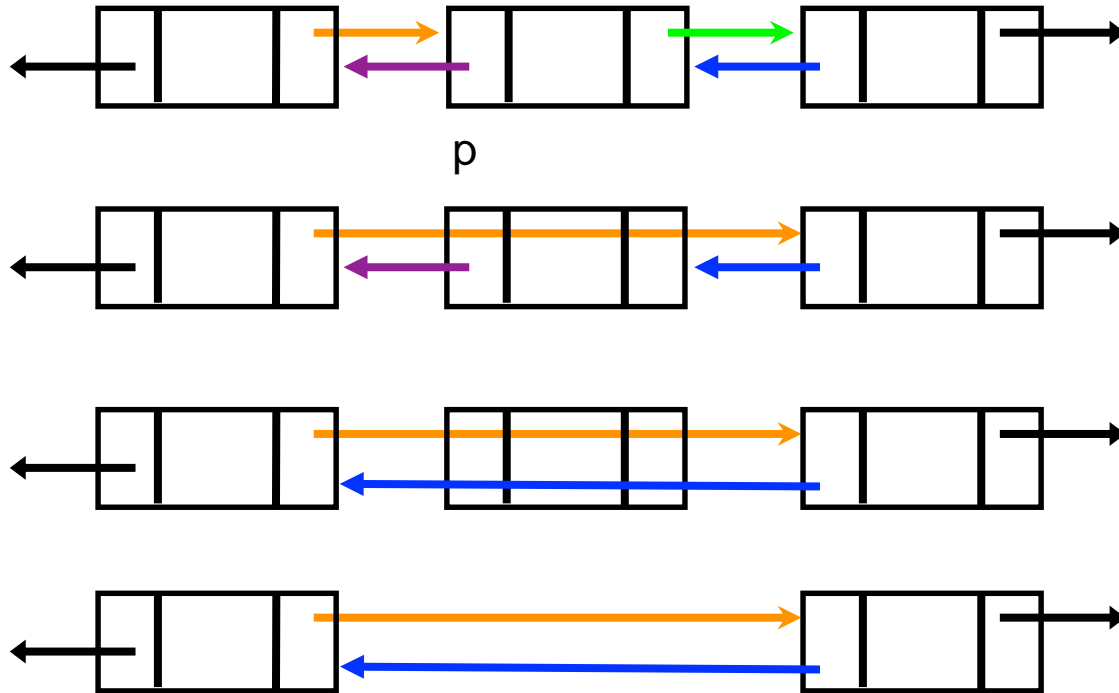
```
void Insert( ElementType X, List L, Position P )
{
    Position TmpCell;

    TmpCell = malloc( sizeof( struct Node ) );
    if ( TmpCell == NULL )
        FatalError( "Out of space!!!" );

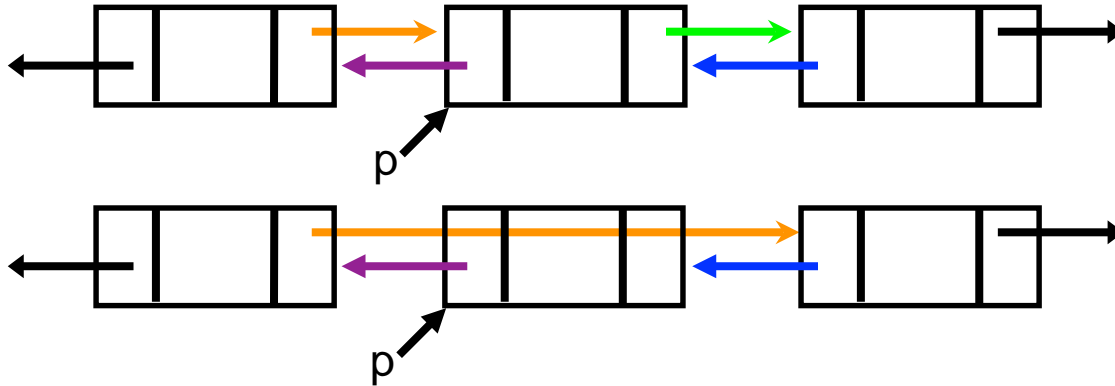
    TmpCell->Element = X;
    TmpCell->Next = P->Next;
    TmpCell->Next->Prev = TmpCell;
    P->Next = TmpCell;
    TmpCell->Prev = P;
}
```

```
struct Node
{
    ElementType Element;
    Position Next;
    Position Prev;
};
```

Doubly Linked List: delete



Doubly Linked List: delete

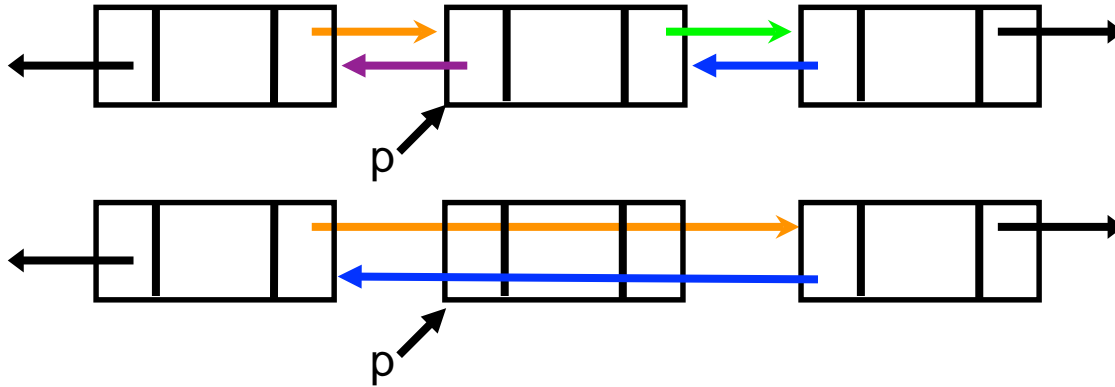


```
void Delete( ElementType X, List L)
{
    Position P;
    P = Find(X, L);

    P->Prev->Next = P->Next;

}
```

Doubly Linked List: delete

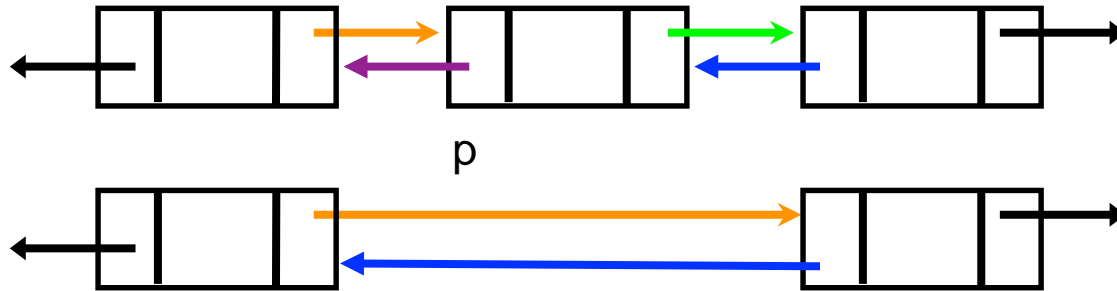


```
void Delete( ElementType X, List L)
{
    Position P;
    P = Find(X, L);

    P->Prev->Next = P->Next;
    P->Next->Prev = P->prev;

}
```

Doubly Linked List: delete

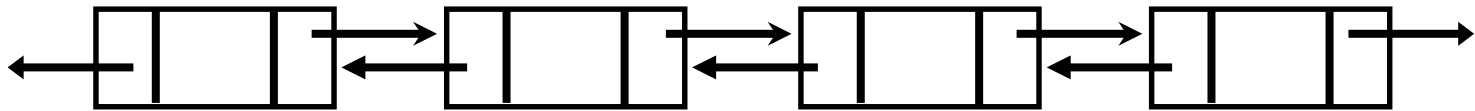


```
void Delete( ElementType X, List L)
{
    Position P;
    P = Find(X, L);

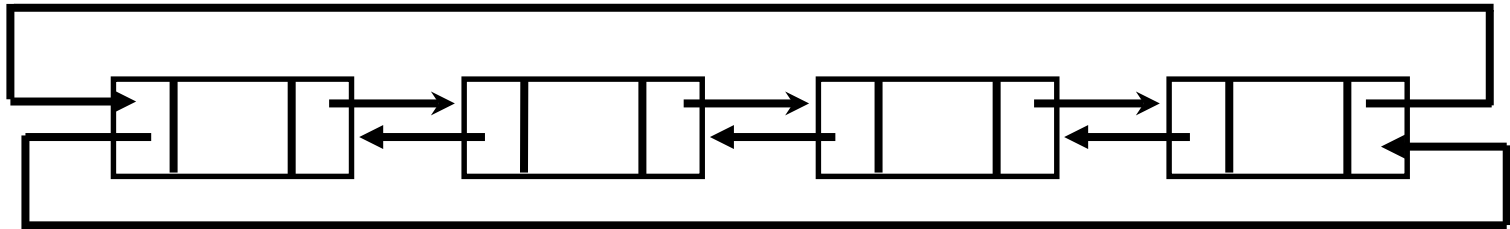
    P->Prev->Next = P->Next;
    P->Next->Prev = P->prev;
    free(P);
}
```


Circularly Linked List

- Doubly Linked List



- Circularly Linked List



Header?