Data Structures

Lecture 4: List

Dongbo Min

Department of Computer Science and Engineering

Ewha Womans University, Korea

E-mail: dbmin@ewha.ac.kr

List

- List: a set of ordered items

- Example
  - Day: Sunday, Monday,…,Saturday
  - A list of text messages in mobile phone

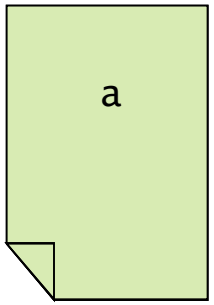
Operations in List

- Add new items to the end, the beginning, and the middle of the list.
- Delete an existing item from an arbitrary position in the list.
- Delete all items.
- Replace existing items.
- Check whether the list has a specific item.
- Return the item at a specific position in the list.
- Count the number of items in the list.
- Check whether the list is empty or full.
- Displays all items in the list.

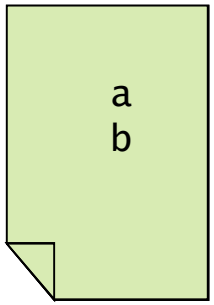
Operations in List

- Object: a group of n ordered elements
- Operation:
  - add_last(list, item): Add ‘item’ to the end.
  - add_first(list, item): Add ‘item’ to the beginning.
  - add(list, pos, item): Add ‘item’ to ‘pos’.
  - delete(list, pos): Removes the element at ‘pos’.
  - clear(list): Removes all elements of the list.
  - replace(list, pos, item): Replace the element at ‘pos’ with ‘item’.
  - is_in_list(list, item): Check to see if ‘item’ is in the list.
  - get_entry(list, pos): Return the element at ‘pos’.
  - Get_length(list): Return the length of the list.
  - Is_empty(list): Check if the list is empty.
  - Is_full(list): Check if the list is full.
  - Display(list): Display all elements in the list.

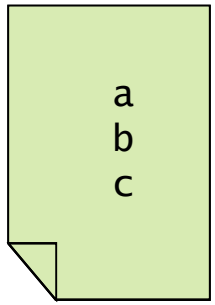
Operations in List



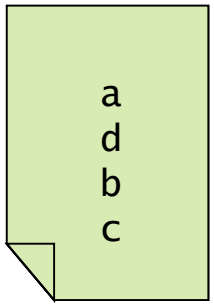
add_last(list1, a)



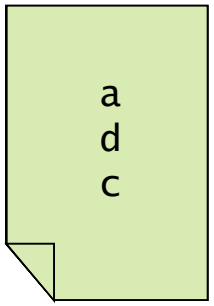
add_last(list1, b)



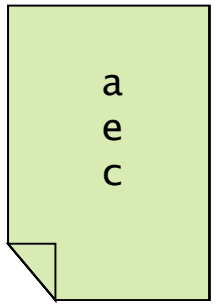
add_last(list1, c)



add(list1,1,d)



delete(list1,2)

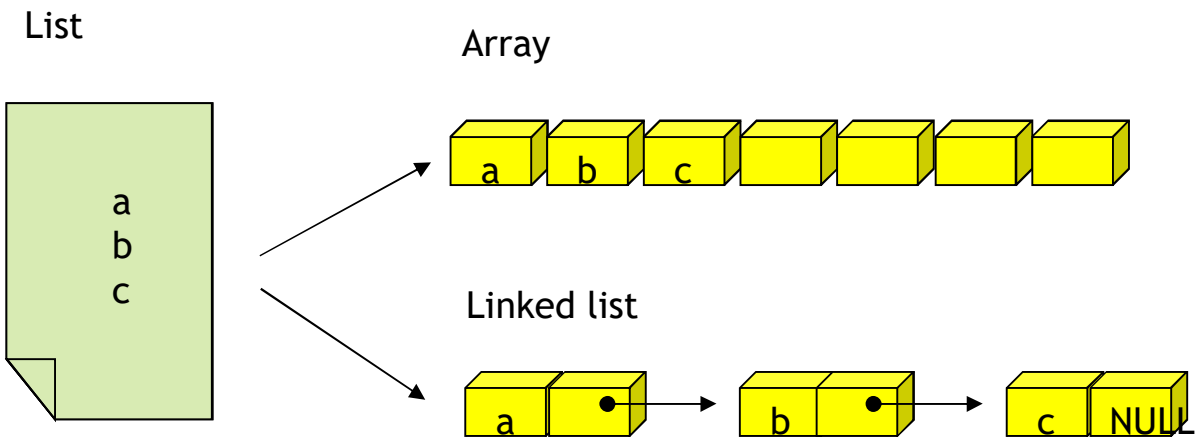


replace(list1,1,e)

List Implementation

- Arrays
  - + Simple to implement
  - - Overhead when inserting or deleting
  - - The number of items is limited

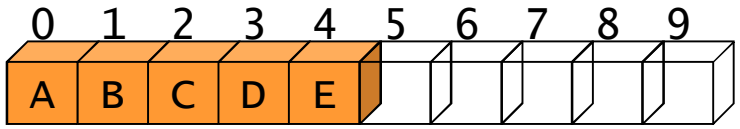
- Linked list
  - + Insertion and deletion are efficient
  - + Size is not limited
  - - Complex implementation



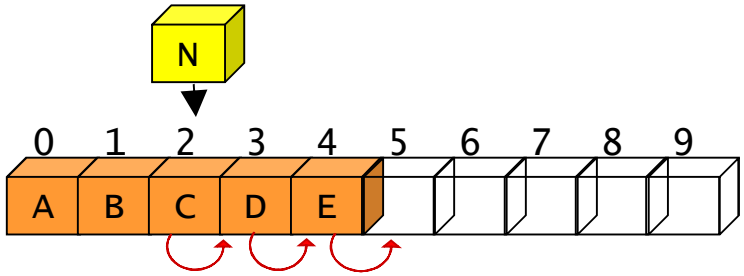
List using Array

- Store items in 1D array in order

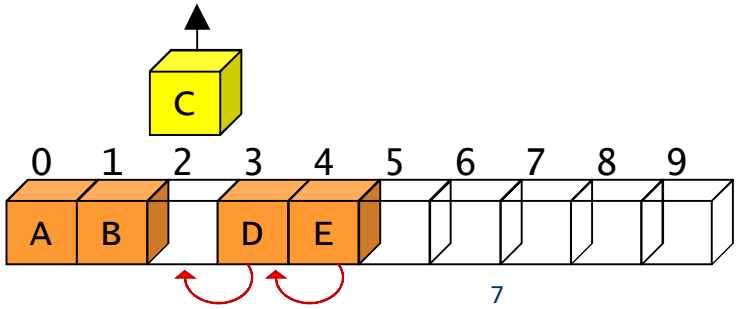
  L = (A, B, C, D, E)



- Insertion operation
  - The items after insertion position should be moved.



- Delete operation
  - The items after delete position should be moved.



List using Array

- The type of items is defined as element
- ‘list’: Save the items in 1D array (list)
- ‘length’: The number of items

```
typedef int element;
typedef struct {
        int list[MAX_LIST_SIZE];   // Define array
        int length;                // The number of items
} ArrayListType;
```

```
// List initialization
void init(ArrayListType *L)
{
        L->length = 0;
}
```

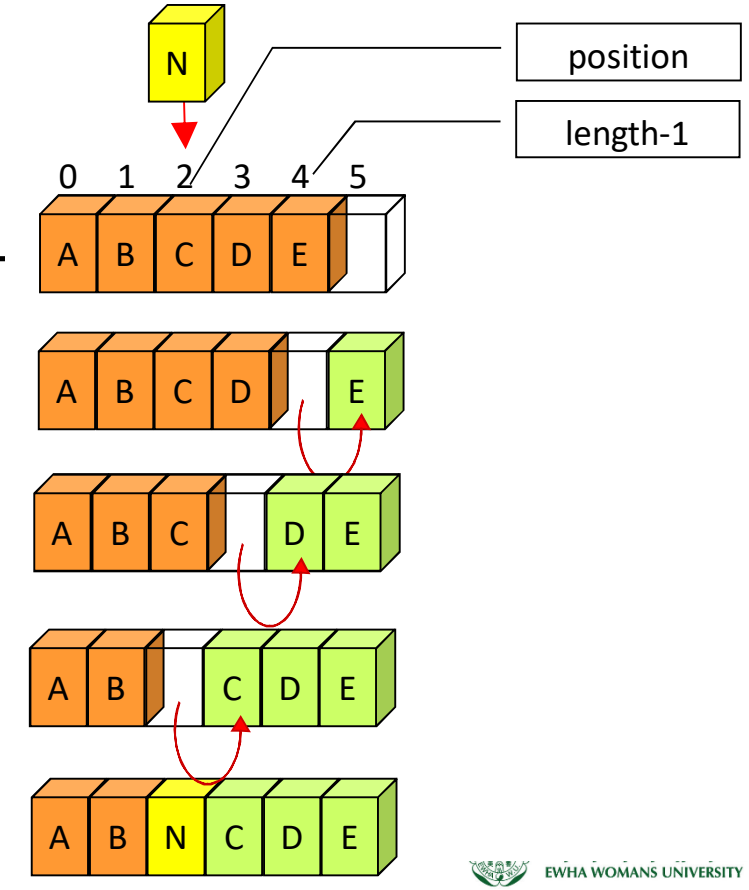
List using Array

- ‘is_empty’ and ‘is_full’

```
int is_empty(ArrayListType *L)
{
        return L->length == 0;
}

int is_full(ArrayListType *L)
{
        return L->length == MAX_LIST_SIZE;
}
```

List using Array: Add

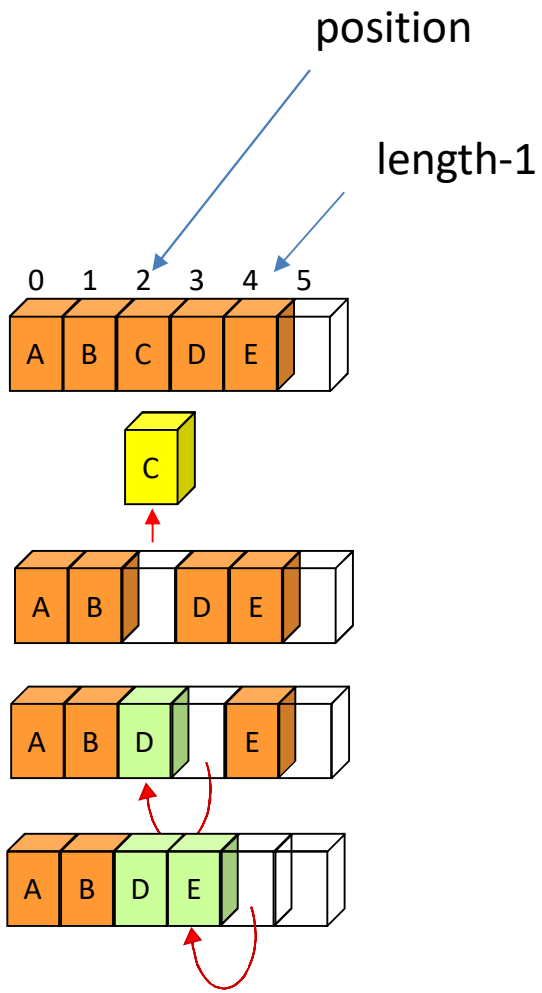
```
// position: location to be inserted
// item: element to be inserted
void add(ArrayListType *L, int position, element item)
{
  if( !is_full(L) && (position >= 0) && (position <= L->length) ){
     int i;
     for(i=(L->length-1); i>=position; i--)
          L->list[i+1] = L->list[i];
     L->list[position] = item;
     L->length++;
  }
}
```



1. The add function first checks to see if the array is saturated and the insertion position is within a range.
2. Move the data after the insertion position one space at a time.

List using Array: Delete

```
// position: position to be deleted
// return the element to be deleted
element delete(ArrayListType *L, int position)
{
        int i;
        element item;

        if( position < 0 || position >= L->length )
                error("Position error");
        item = L->list[position];
        for(i=position; i<(L->length-1);i++)
                L->list[i] = L->list[i+1];
        L->length--;
        return item;
}
```



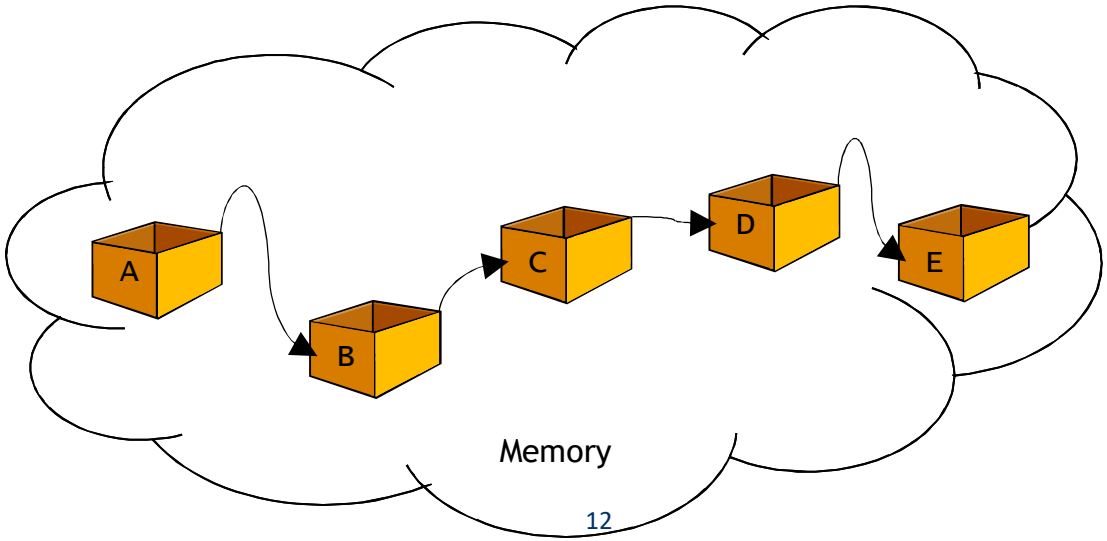
1. Check the delete position.
2. Move data from deletion position to the end by one space.

Linked List

- Linked list
  - A node consists of data and link, and the link connects nodes.

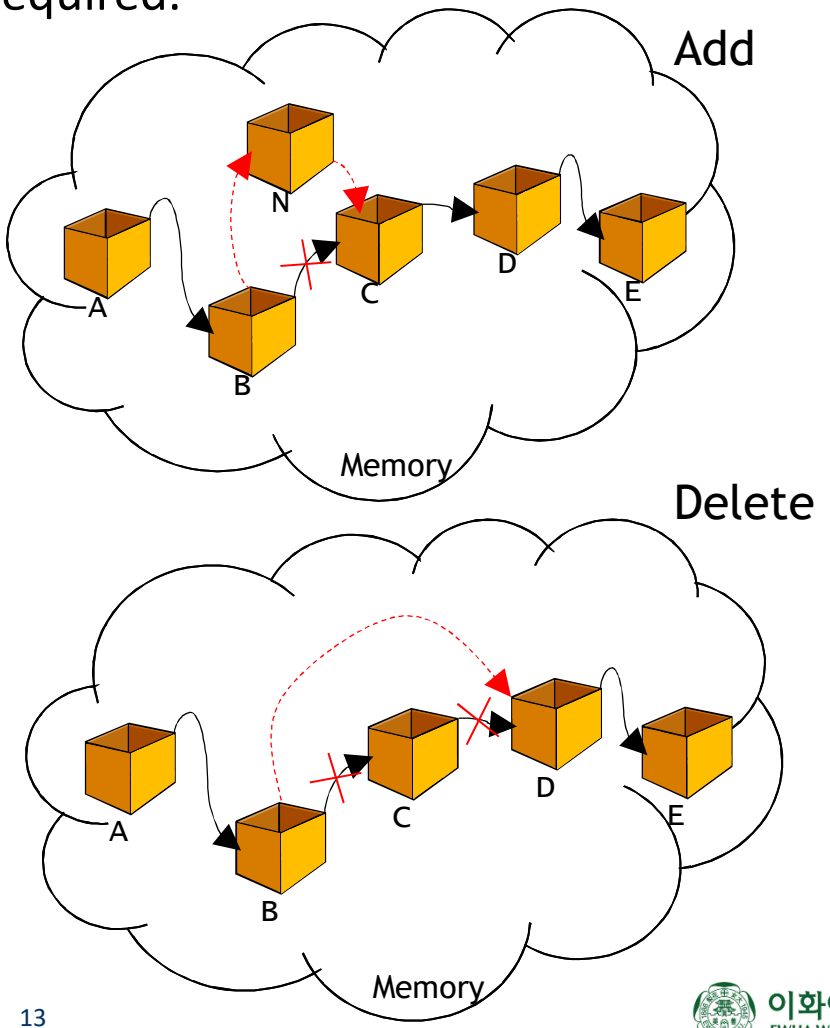
- Node: <entry, address>
  - Data field (entry) - store the data value (element) of the list
  - Link field (address) - stores the address (pointer) of another node

- Note) The physical order of nodes in memory does not need to match the logical order of the list.



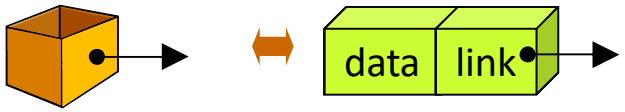
Linked List

- Pros
  - Insertion and deletion are easier.
  - No consecutive memory space is required.
  - There is no size limit.
- Cons
  - Implementation is difficult.
  - Errors are likely to occur.

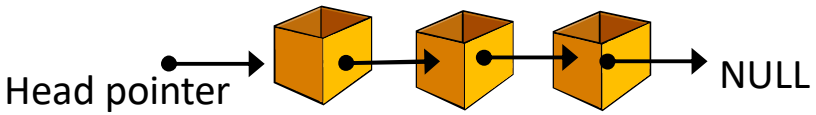


Linked List

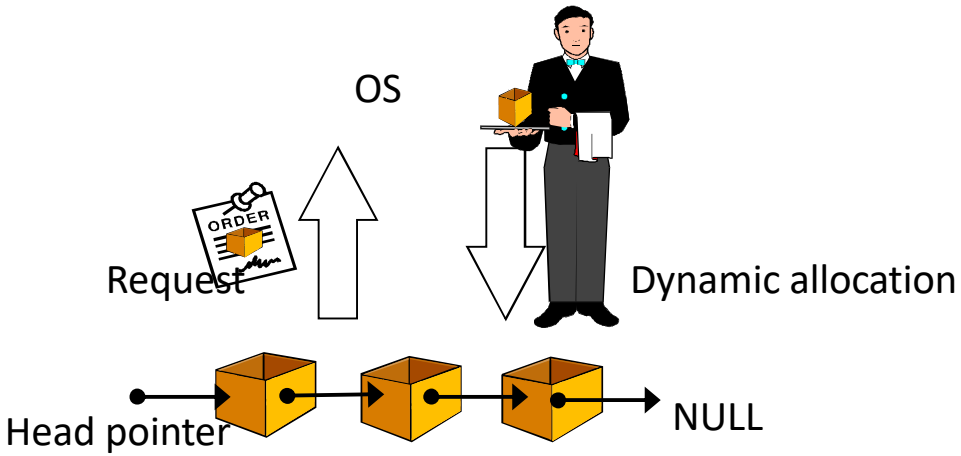
- Node = data field + link field



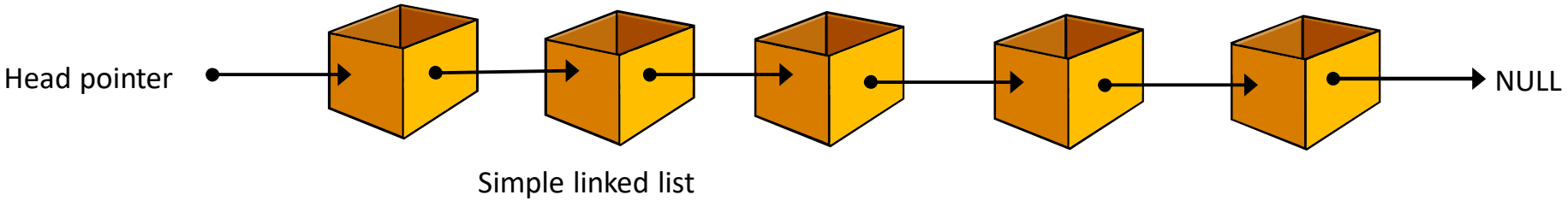
- Head pointer:
  - Variable that points to the first node in the list



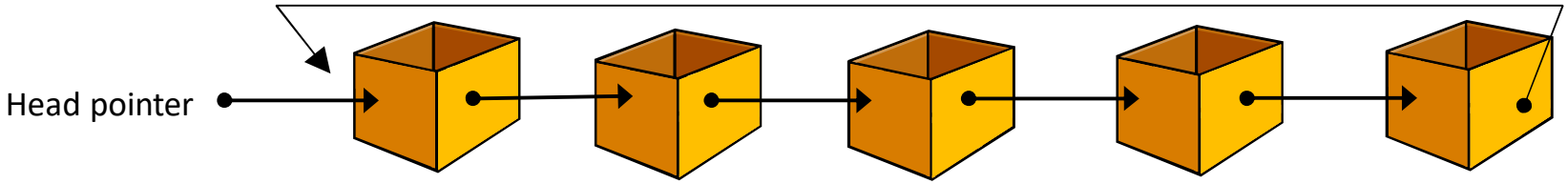
- A node is created by allocating dynamic memory whenever necessary



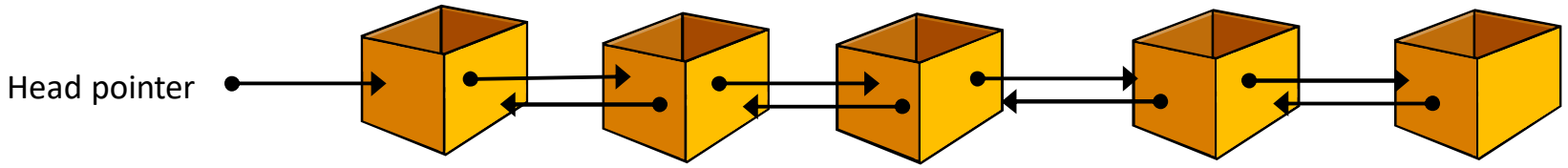
Linked List



Simple linked list



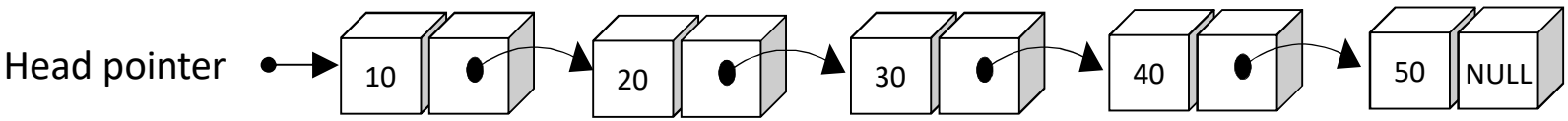
Circular linked list



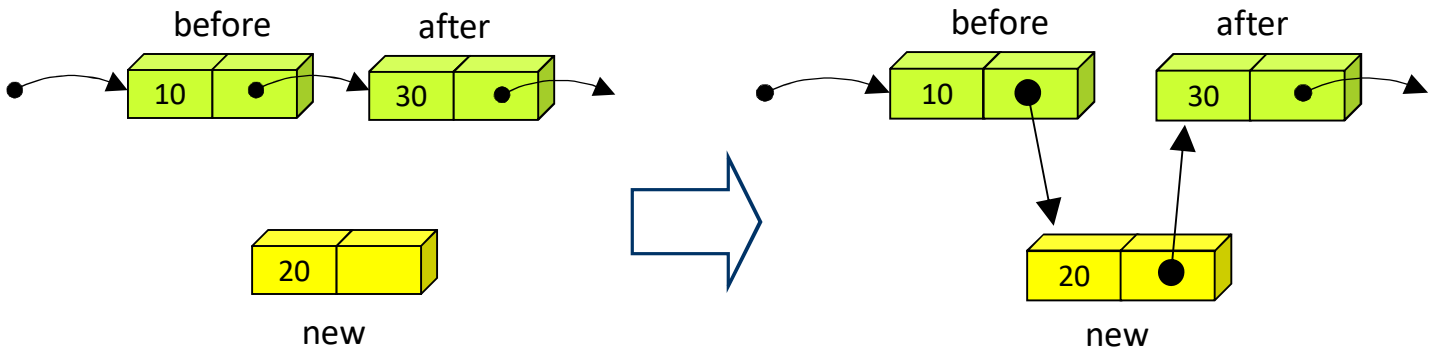
Doubly linked list

Simple Linked List

- Link the data using one link field
- The value of the last node is NULL.

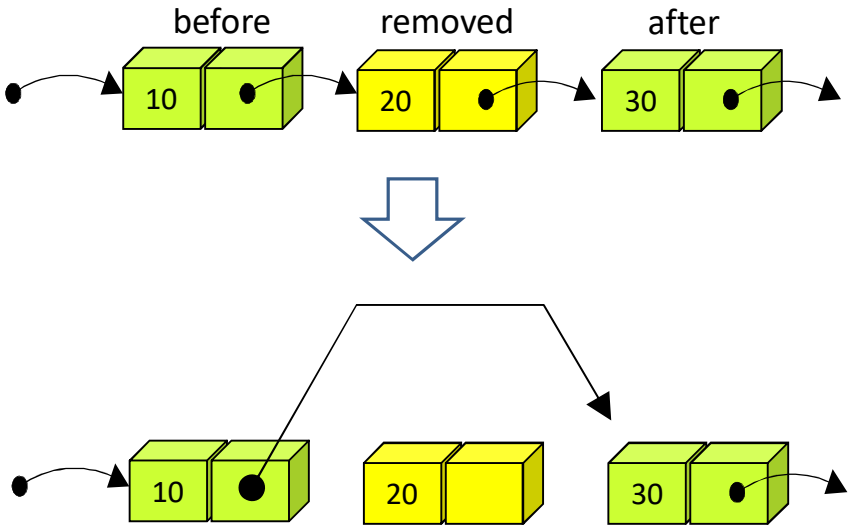


Simple Linked List: Insertion




```
insert_node(L, before, new)

if  L == NULL
        then  L ← new
else   new.link ← before.link
       before.link ← new
```

Simple Linked List: Deletion




```
remove_node(L, before, removed)

if L ≠ NULL
     then        before.link ← removed.link
                 destroy(removed)
```

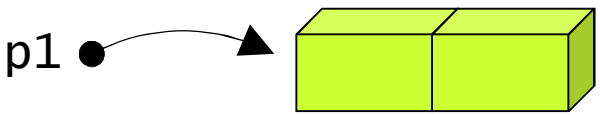
Simple Linked List Implementation

- Data field is defined as a structure
- Link field uses a pointer

```
typedef int element;
typedef struct ListNode {
        element data;
        struct ListNode *link;
} ListNode;
```

- Node generation: using ‘malloc’ function

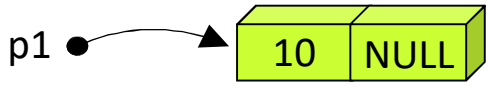
```
ListNode *p1;
p1 = (ListNode *)malloc(sizeof(ListNode));
```



Simple Linked List Implementation

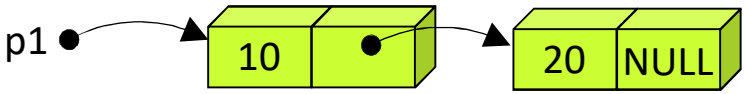
- Setting data fields and link fields

```
p1->data = 10;
p1->link = NULL;
```



- Creating a second node and connecting it to the first node

```
ListNode *p2;
p2 = (ListNode *)malloc(sizeof(ListNode));
p2->data = 20;
p2->link = NULL;
p1->link = p2;
```



- Head pointer: a pointer to the first node in the linked list

Simple Linked List: Insertion

- Insertion function

```
void insert_node(ListNode **phead, ListNode *p, ListNode *new_node)

phead: pointer to the head pointer of the list
p: pointer to the preceding node of the location to be inserted
new_node: pointer to the new node to be inserted
```

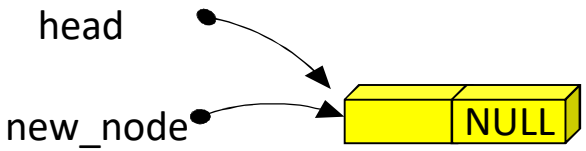
  - Since the head pointer changes inside the function, we need a pointer to the head pointer.

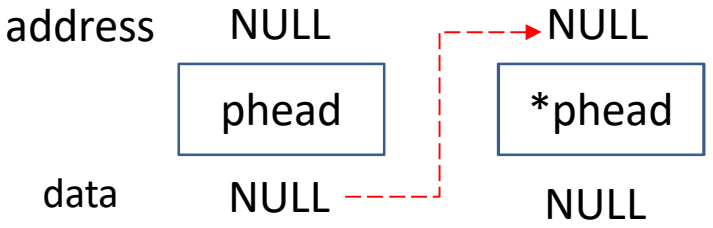
- Four cases of insertion
  1. If head is NULL: insert in a blank list
  2. Insert in the beginning of list
  3. Insert in the middle of list
  4. Insert at the end of list

Note) Explained differently from the textbook

Simple Linked List: Insertion

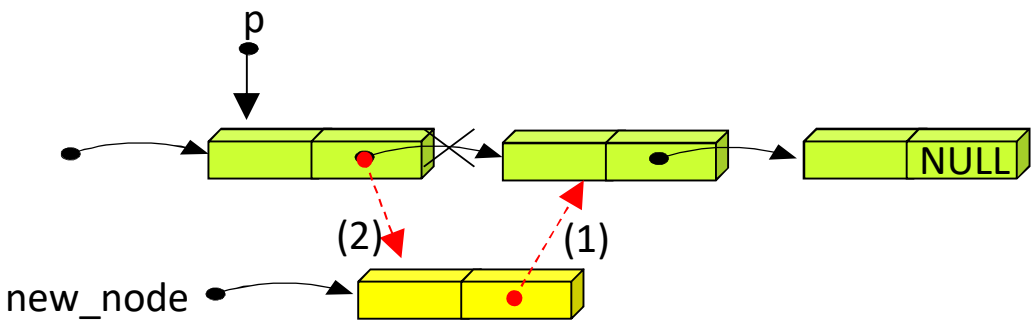
1. If '*phead' is NULL:
   The 'new_node' becomes the first node. So you only need to change the value of head.

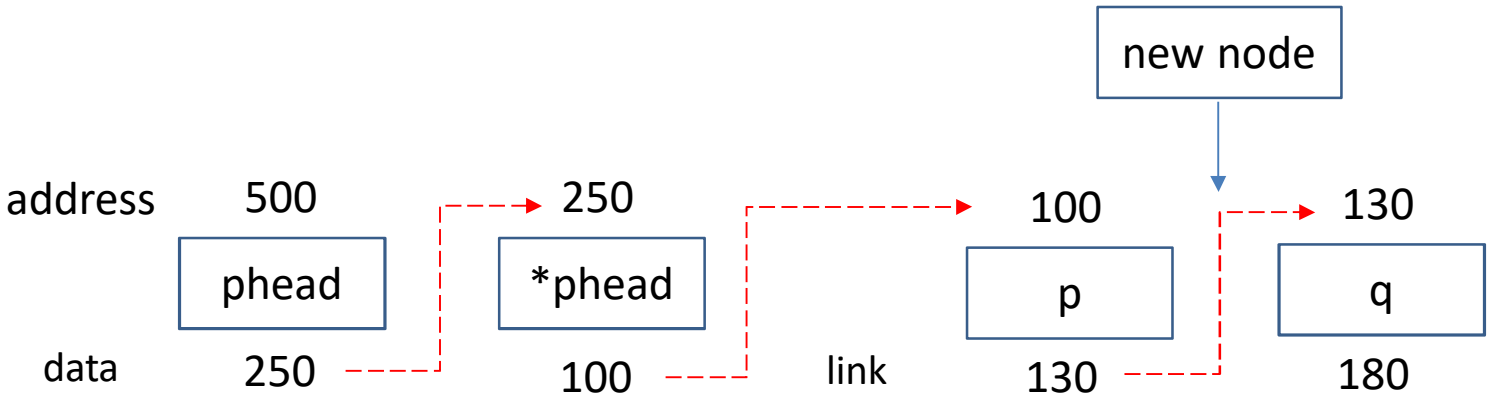




Simple Linked List: Insertion

2. If '*phead' is not NULL, insert in the middle of list
   This is the most common case.
   (1) Copy 'p'->link to 'new_node'->link.
   (2) Let p -> link point to new_node.



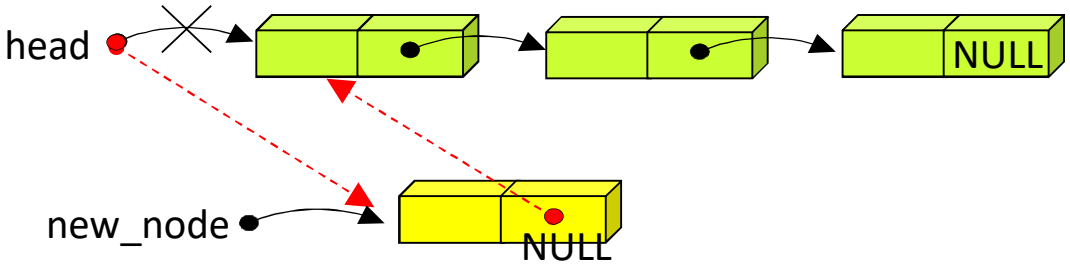


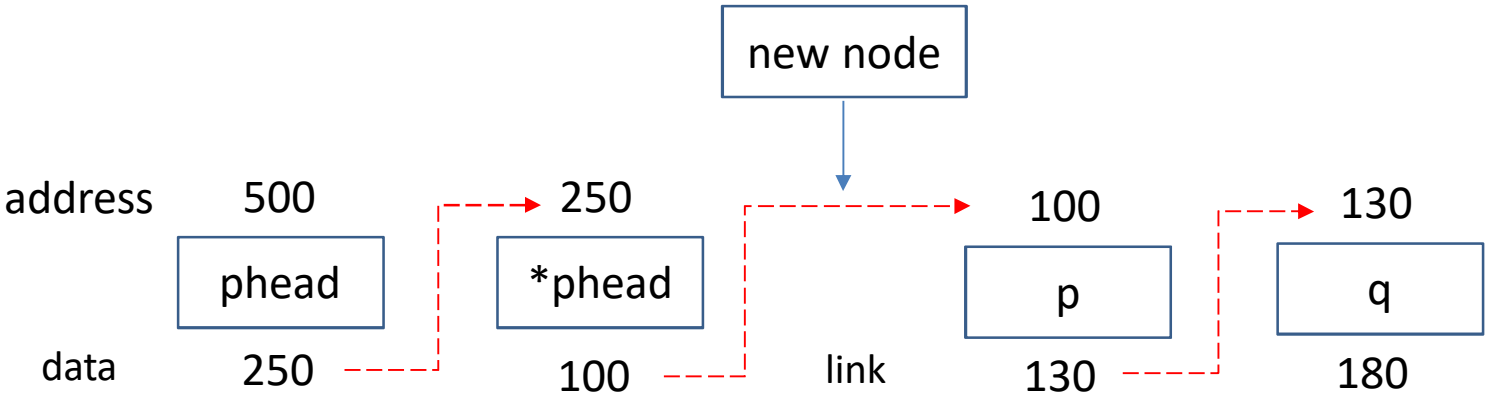
Simple Linked List: Insertion

```
// phead: pointer to the head pointer of the list
// p: preceding node
// new_node: node to be inserted
void insert_node(ListNode **phead, ListNode *p, ListNode *new_node)
{
        if( *phead == NULL ){
                new_node->link = NULL;
                *phead = new_node;
        }
        else {
                if(p == NULL){
                        printf("The preceding node cannot be NULL.\n");
                        exit(1);
                }
                else{
                        new_node->link = p->link;
                        p->link = new_node;
                }
        }
}
```

Simple Linked List: Insertion

- Insert in the beginning of the list: ‘Insert_first’



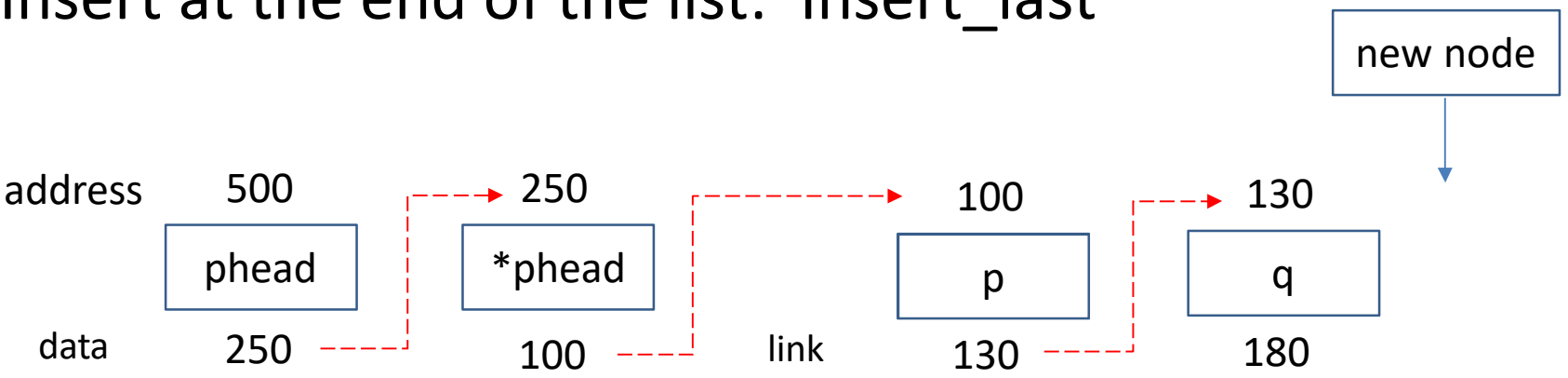


Simple Linked List: Insertion

```
// phead: pointer to the head pointer of the list
// new_node: node to be inserted
void insert_first(ListNode **phead, ListNode *new_node)
{
        if( *phead == NULL ){                  //if the list is blank
                new_node->link = NULL;
                *phead = new_node;
        }
        else{
                new_node->link = *phead;
                *phead = new_node;
        }
}
```

Simple Linked List: Insertion

- Insert at the end of the list: 'Insert_last'




```
// phead: pointer to the head pointer of the list
// new_node: node to be inserted
void insert_last(ListNode **phead, ListNode *new_node)
{
        if( *phead == NULL ){                   //if the list is blank
                new_node->link = NULL;
                *phead = new_node;
        }
        else{
                new_node->link = NULL;
                struct ListNode *last = *phead;
                while(last->link != NULL)
                        last = last->link;
                last->link = new_node;
        }
}
```

Time complexity: O(N)
where N=# of nodes

Simple Linked List: Deletion

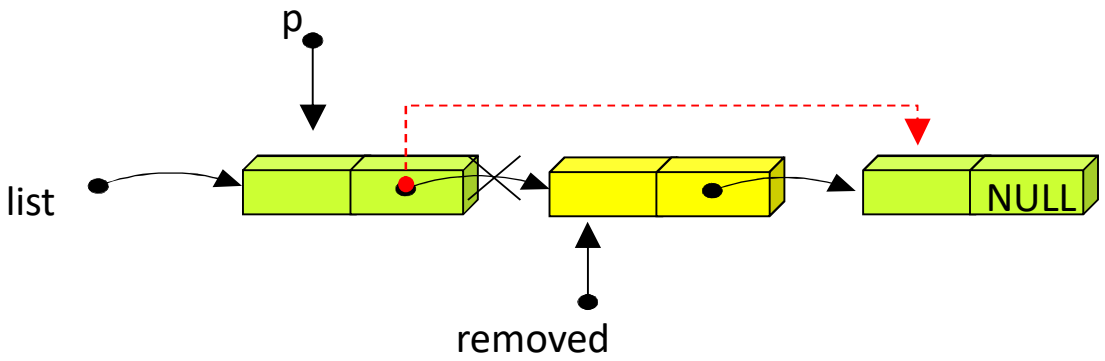
- Deletion function

```
//phead: pointer to the head pointer
//p: pointer to the preceding node of the node to be deleted
//removed: pointer to the node to be deleted
void remove_node(ListNode **phead, ListNode *p, ListNode *removed)
```

Note) Explained differently from the textbook

Simple Linked List: Deletion

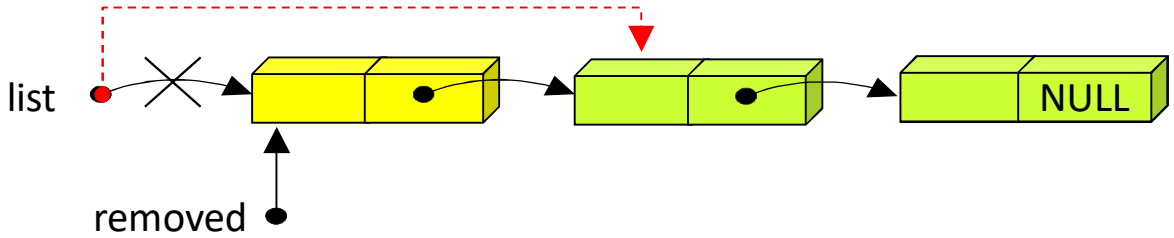
- The link of ‘p’ is changed to point to the next node of ‘removed’.




```
// phead : pointer to the head pointer
// p: preceding node to be deleted
// removed: node to be deleted
void remove_node(ListNode **phead, ListNode *p, ListNode *removed)
{
        if( *phead == NULL ){
                printf(“The list is blank.\n”);
        }
        else {
                if( p == NULL )
                        printf(“The preceding node cannot be NULL.\n”);
                else {
                        p->link = removed->link;
                        free(removed);
                }
        }
}
```

Simple Linked List: Deletion

- Delete the beginning of the list: ‘remove_first’




```
// phead : pointer to the head pointer
// removed: node to be deleted
void remove_first(ListNode **phead, ListNode *removed)
{
        if( *phead == NULL ){
                printf(“The list is blank.\n”);
        }
        else {
                *phead = (*phead)->link;
                free(removed);
        }
}
```

Visit Operation

- Sequentially visit the nodes in the list

- Iteration

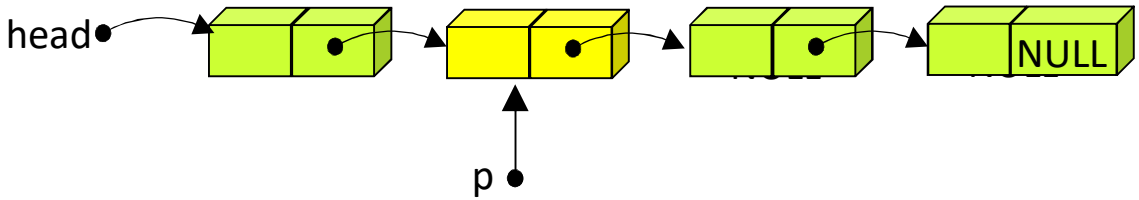
```
void display(ListNode *head)
{
        ListNode *p=head;
        while( p != NULL ){
                printf("%d->", p->data);
                p = p->link;
        }
        printf("\n");
}
```

- Recursion

```
void display_recur(ListNode *head)
{
        ListNode *p=head;
        if( p != NULL ){
                printf("%d->", p->data);
                display_recur(p->link);
        }
}
```

Search Operation

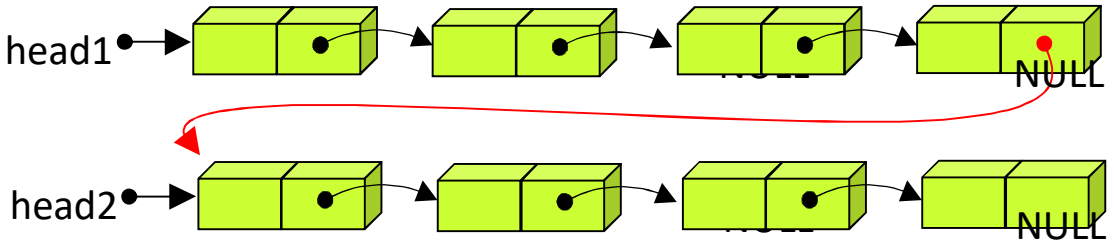
- Search for nodes with specific data values




```
ListNode *search(ListNode *head, int x)
{
        ListNode *p;
        p = head;
        while( p != NULL ){
                if( p->data == x ) return p;
                p = p->link;
        }
        return p;  // return NULL if search fails
}
```

Merge Operation

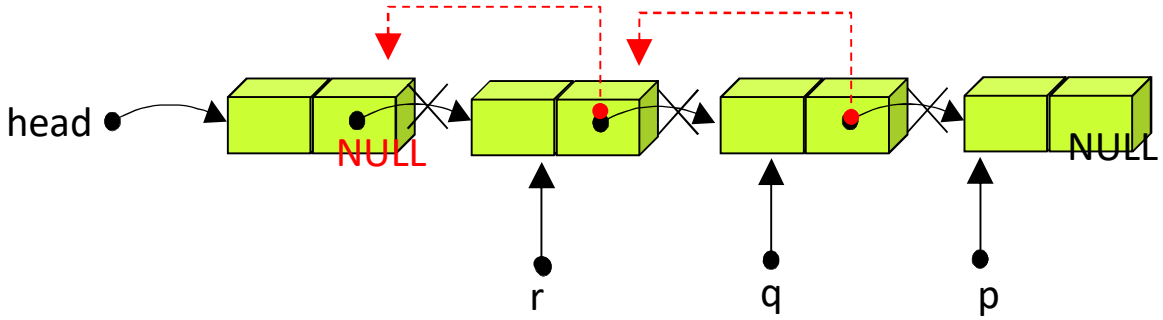
- Merge two lists




```
ListNode *concat(ListNode *head1, ListNode *head2)
{
        ListNode *p;
        if( head1 == NULL ) return head2;
        else if( head2 == NULL ) return head1;
        else {
                p = head1;
                while( p->link != NULL )
                        p = p->link;
                p->link = head2;
                return head1;
        }
}
```

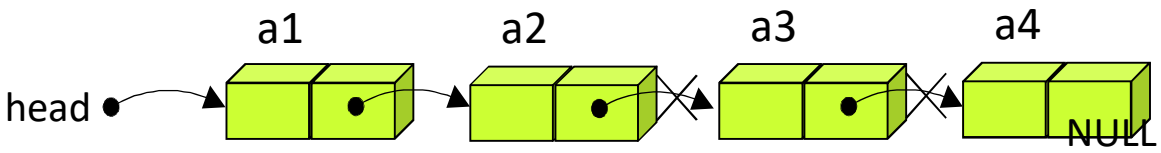
Reverse Operation

- Arrange the nodes of the list in reverse order




```
ListNode *reverse(ListNode *head)
{
   ListNode *p, *q, *r;
   p = head;          // p: node yet not processed
   q = NULL;          // q: node in reverse order
   while (p != NULL){
       r = q;
       q = p;
       p = p->link;
       q->link =r;
    }
    return q;      // q: head pointer of the list in reverse order
}
```

Reverse Operation

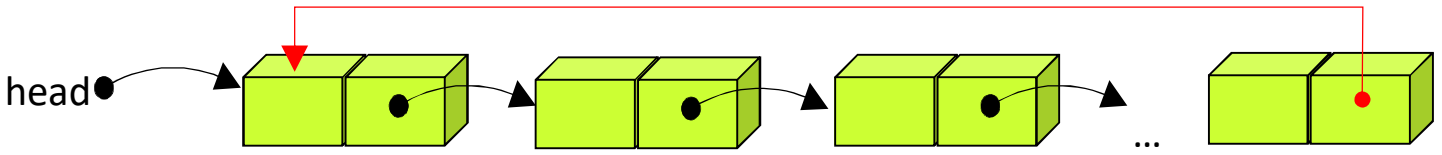



```
ListNode *reverse(ListNode *head)
{
   ListNode *p, *q, *r;
   p = head;
   q = NULL;
   while (p != NULL){
       r = q;
       q = p;
       p = p->link;
       q->link =r;
    }
    return q;
}
```

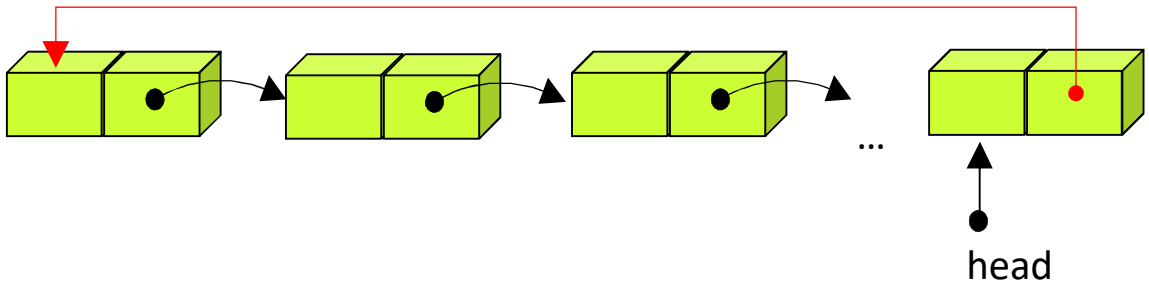
Iteration	r	q	p
0		NULL	head (=a1)
1	NULL ←	head (=a1)	a2
2	a1 ←	a2	a3
3	a2 ←	a3	a4
4	a3 ←	a4	NULL

Circular Linked List

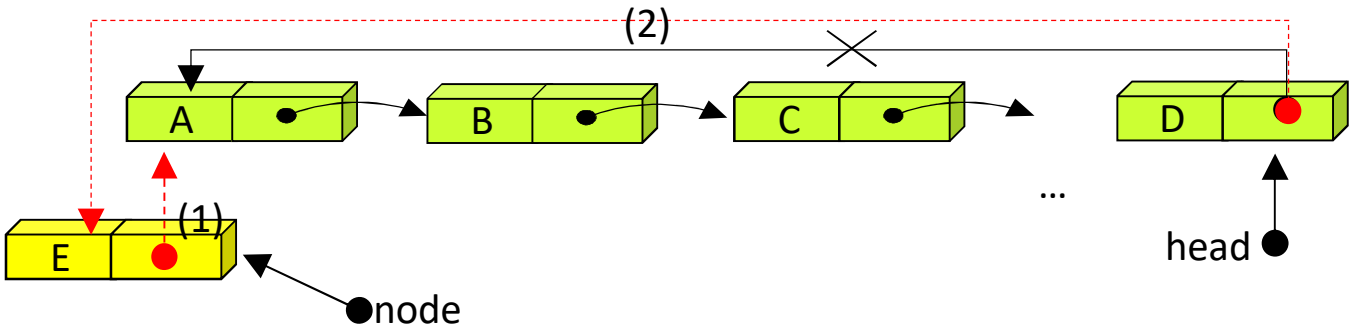
- The link of the last node points to the first node
- It can access from one node to all other nodes.



- When the head pointer is pointed to the last node, the operation of inserting the node at the beginning or end of the list is easier than the simple linked list.

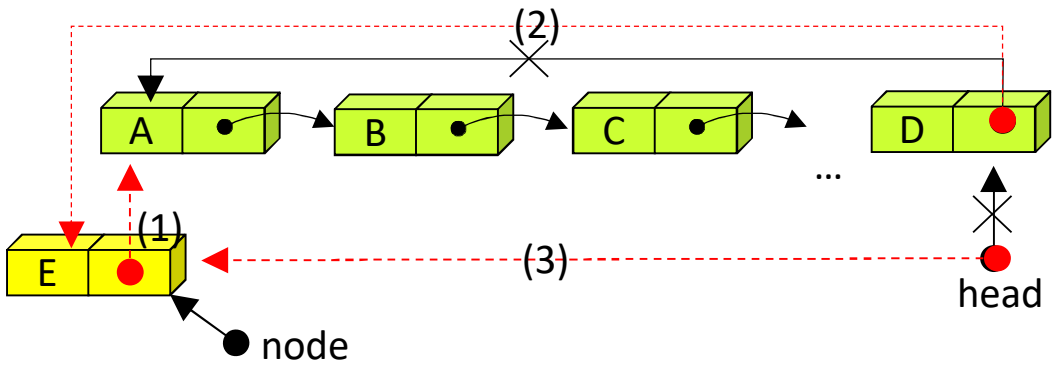


Insertion Operation at the Beginning




```
// phead: pointer of head pointer
// p: preceding node
// node: node to be inserted
void insert_first(ListNode **phead,   ListNode *node)
{
          if( *phead == NULL ){
                    *phead = node;
                    node->link = node;
          }
          else {
                    node->link = (*phead)->link;
                    (*phead)->link = node;
          }
}
```

Insertion Operation at the End




```
// phead: pointer of head pointer
// p: preceding node
// node: node to be inserted
void insert_last(ListNode **phead, ListNode *node)
{
        if( *phead == NULL ){
                *phead = node;
                node->link = node;
        }
        else {
                node->link = (*phead)->link;
                (*phead)->link = node;
                *phead = node;
        }
}
```

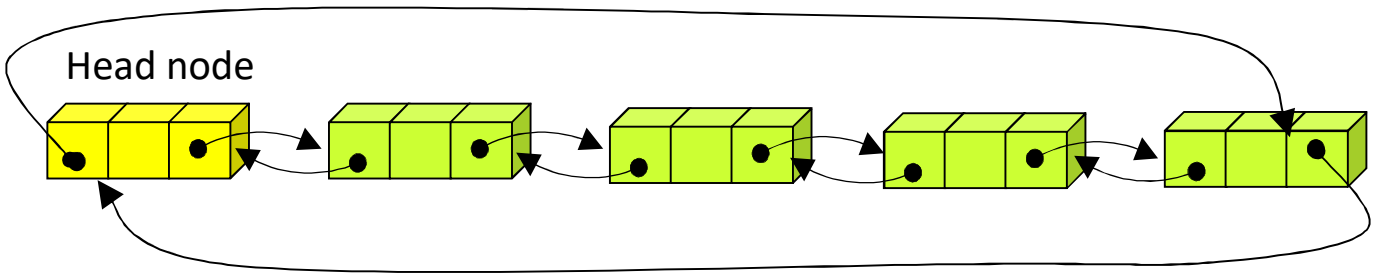
Time complexity: O(1)

Doubly Linked List

- Problem with the simple linked list
  - It is difficult to find the preceding node
  - The preceding node should be defined together when inserting or deleting

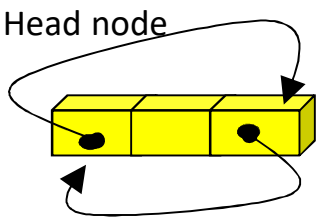
- Doubly linked list
  - A list in which one node has two links to the preceding node and the subsequent node
  - Pros: A link is bi-directional, so you can search in both directions.
  - Cons: It takes up a lot of space, and the code is complex.

```
p == p->llink->rlink == p->rlink->llink
```



Doubly Linked List: Head Node

- Head node
  - has no actual data
  - Is created to simplify insertion and deletion
  - It is distinguished from *head pointer*
  - In the blank, only the head node exists

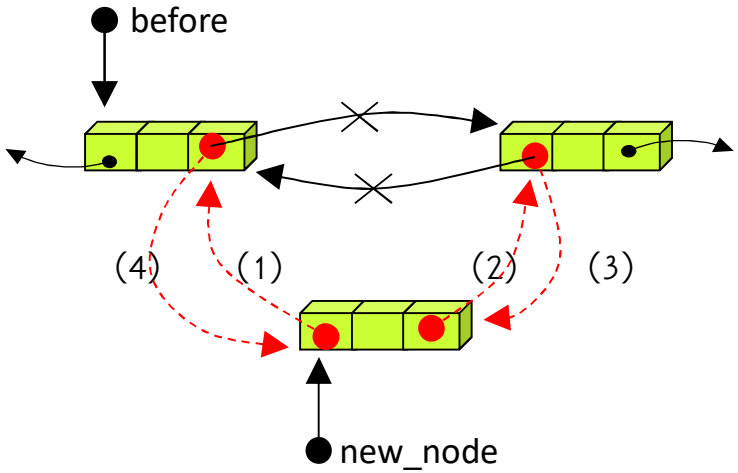



```
void init(DlistNode *phead){
          phead->llink = phead;
          phead->rlink = phead;
}
```

- Structure of a node in a doubly linked list

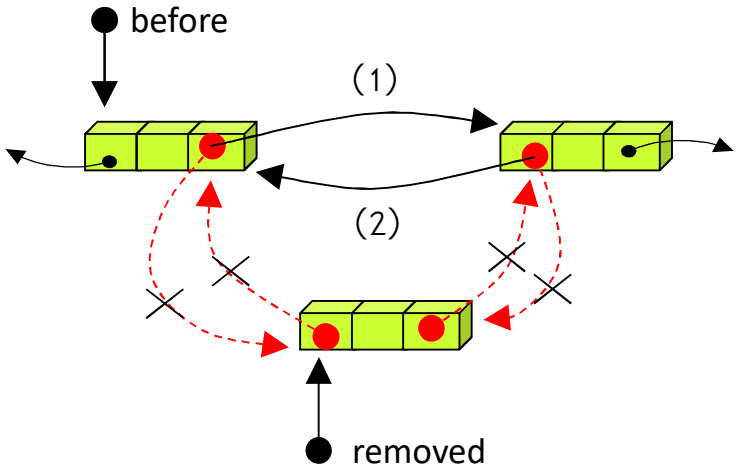
```
typedef int element;
typedef struct DlistNode {
          element data;
          struct DlistNode *llink;
          struct DlistNode *rlink;
} DlistNode;
```

Insertion Operation




```
// Insert 'new_node' into the right of 'before'
void dinsert_node(DlistNode *before, DlistNode *new_node)
{
        new_node->llink = before;
        new_node->rlink = before->rlink;
        before->rlink->llink = new_node;
        before->rlink = new_node;
}
```

Deletion Operation




```
// Delete ‘removed’
void dremove_node(DlistNode *phead_node, DlistNode *removed)
{
        if( removed == phead_node ) return;
        removed->llink->rlink = removed->rlink;
        removed->rlink->llink = removed->llink;
        free(removed);
}
```

Example

```
typedef int element;
typedef struct DlistNode {
            element data;
            struct DlistNode *llink;
            struct DlistNode *rlink;
} DlistNode;

void init(DlistNode *phead) {
            phead->llink = phead;
            phead->rlink = phead;
}

void display(DlistNode *phead) {
            DlistNode *p;
            for (p = phead->rlink; p != phead; p = p->rlink)
            {
                        printf("<--- | %x | %d | %x | --->\n", p->llink, p->data, p->rlink);
            }
            printf("\n");
}
```

Example

```
// Insert ‘new_node’ into the right of ‘before’
void dinsert_node(DlistNode *before, DlistNode *new_node)
{
        new_node->llink = before;
        new_node->rlink = before->rlink;
        before->rlink->llink = new_node;
        before->rlink = new_node;
}
// Delete ‘removed’
void dremove_node(DlistNode *phead_node, DlistNode *removed)
{
        if (removed == phead_node) return;
        removed->llink->rlink = removed->rlink;
        removed->rlink->llink = removed->llink;
        free(removed);
}

void main()
{
        DlistNode head_node;
        DlistNode *p[10];

        init(&head_node);
        for (int i = 0; i < 5; i++)
        {
                p[i] = (DlistNode *)malloc(sizeof(DlistNode));
                p[i]->data = i;
                dinsert_node(&head_node, p[i]);
        }
        dremove_node(&head_node, p[4]);
        display(&head_node);
}
```

```
DlistNode *head_node = (DlistNode *)malloc(sizeof(DlistNode));
DlistNode *p[10];

init(head_node);
for (int i = 0; i < 5; i++)
{
        p[i] = (DlistNode *)malloc(sizeof(DlistNode));
        p[i]->data = i;
        dinsert_node(head_node, p[i]);
}

dremove_node(head_node, p[4]);
display(head_node);
```

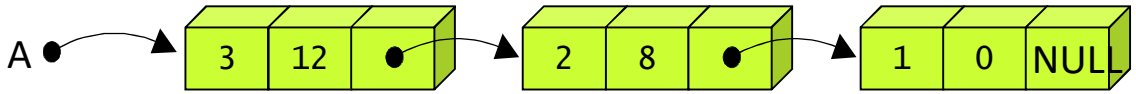
Linked List Application: Polynomial

- Linked list can be used for processing of polynomials

  Ex) Arithmetic operations of polynomials

  – Representation of a polynomial as a linked list

  $A = 3x^{12} + 2x^{8} + 1$

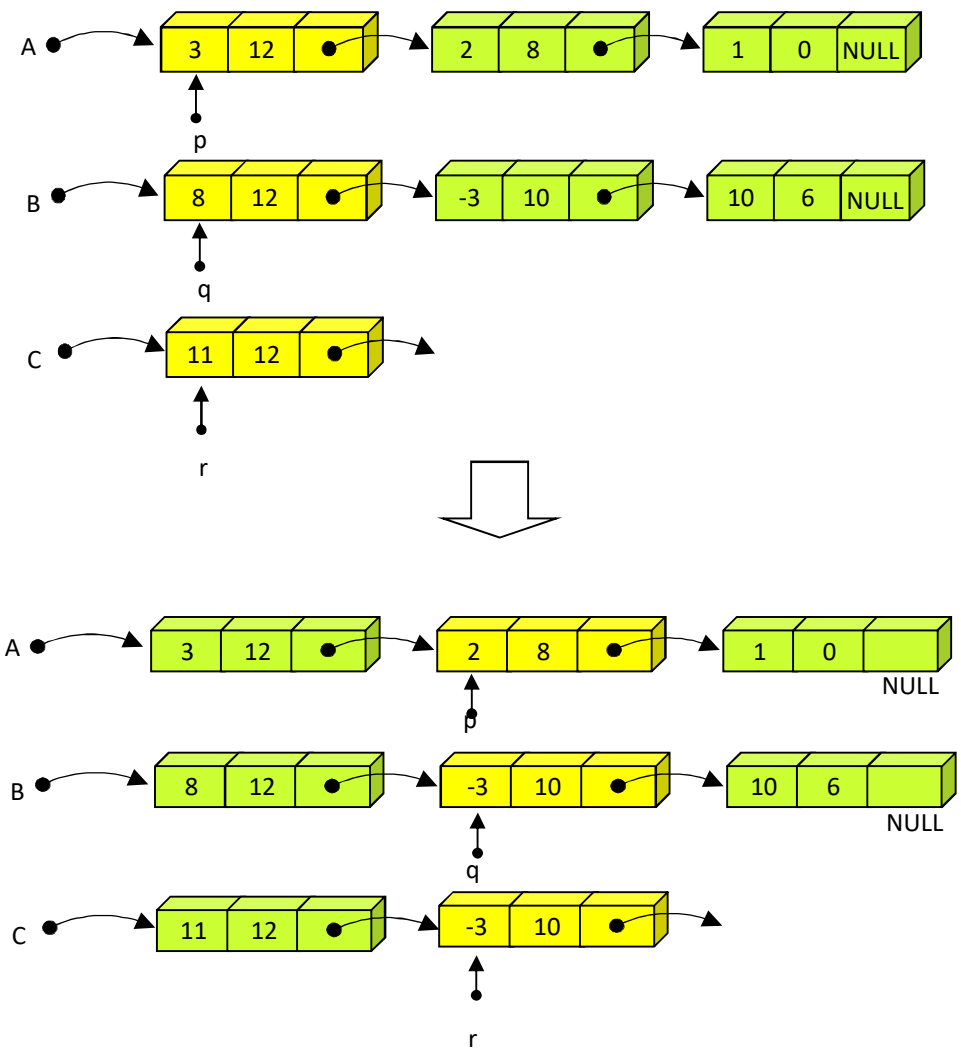



```
typedef struct ListNode {
          int coef;
          int expon;
          struct ListNode *link;
} ListNode;

ListNode *A, *B;
```

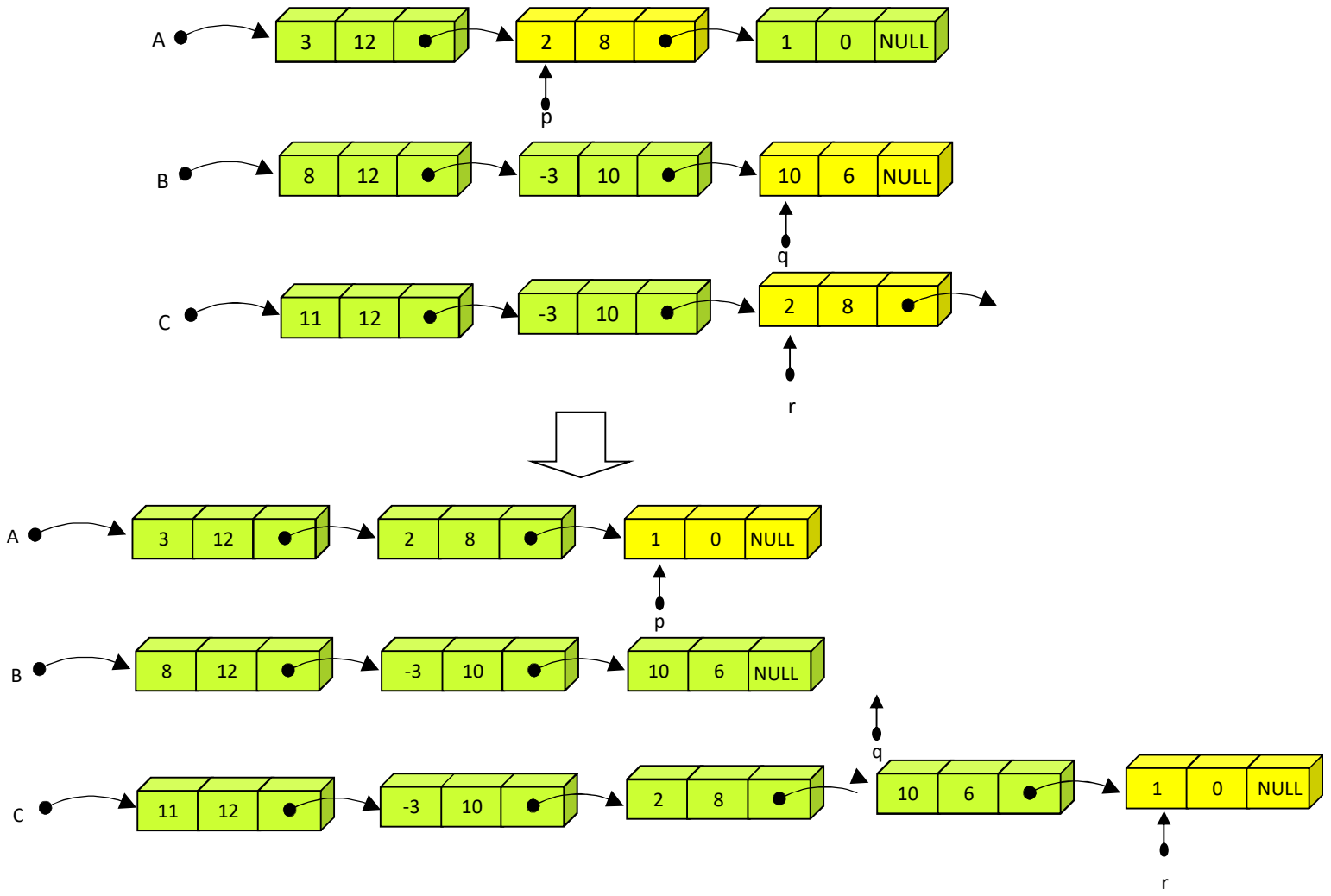
Linked List Application: Polynomial

- Addition of polynomial

$$A = 3x^{12} + 2x^8 + 1$$
$$B = 8x^{12} - 3x^{10} + 10x^6$$
$$A + B = 11x^{12} - 3x^{10} + 2x^8 + 10x^6 + 1$$



Linked List Application: Polynomial



Linked List Application: Polynomial

```
#include <stdio.h>
#include <stdlib.h>
// Structure of node in the linked list
typedef struct ListNode {
          int coef;
          int expon;
          struct ListNode *link;
} ListNode;
// Header in the linked list
typedef struct ListHeader {
          int length;
          ListNode *head;
          ListNode *tail;
} ListHeader;


void error(char *message)
{
          fprintf(stderr, “%s\n”, message);
          exit(1);
}
```

Linked List Application: Polynomial

```
// Initialization
void init(ListHeader *plist)
{
        plist->length = 0;
        plist->head = plist->tail = NULL;
}
// plist: pointer to point the header of the linked list
// coef: coefficient, expon:exponent
void insert_node_last(ListHeader *plist, int coef, int expon)
{
        ListNode *temp = (ListNode *)malloc(sizeof(ListNode));
        if (temp == NULL) error("Memory allocation error");

        temp->coef = coef;
        temp->expon = expon;
        temp->link = NULL;
        if (plist->tail == NULL) {
                plist->head = plist->tail = temp;
        }
        else {
                plist->tail->link = temp;
                plist->tail = temp;
        }
        plist->length++;
}
```

Linked List Application: Polynomial

```
// list3 = list1 + list2
void poly_add(ListHeader *plist1, ListHeader *plist2, ListHeader *plist3)
{
        ListNode *a = plist1->head;
        ListNode *b = plist2->head;
        int sum;
        while (a && b) {
                if (a->expon == b->expon) { // a exponent == b exponent
                        sum = a->coef + b->coef;
                        if (sum != 0) insert_node_last(plist3, sum, a->expon);
                        a = a->link; b = b->link;
                }
                else if (a->expon > b->expon) { // a exponent > b exponent
                        insert_node_last(plist3, a->coef, a->expon);
                        a = a->link;
                }
                else { // a exponent < b exponent
                        insert_node_last(plist3, b->coef, b->expon);
                        b = b->link;
                }
        }
```

Linked List Application: Polynomial

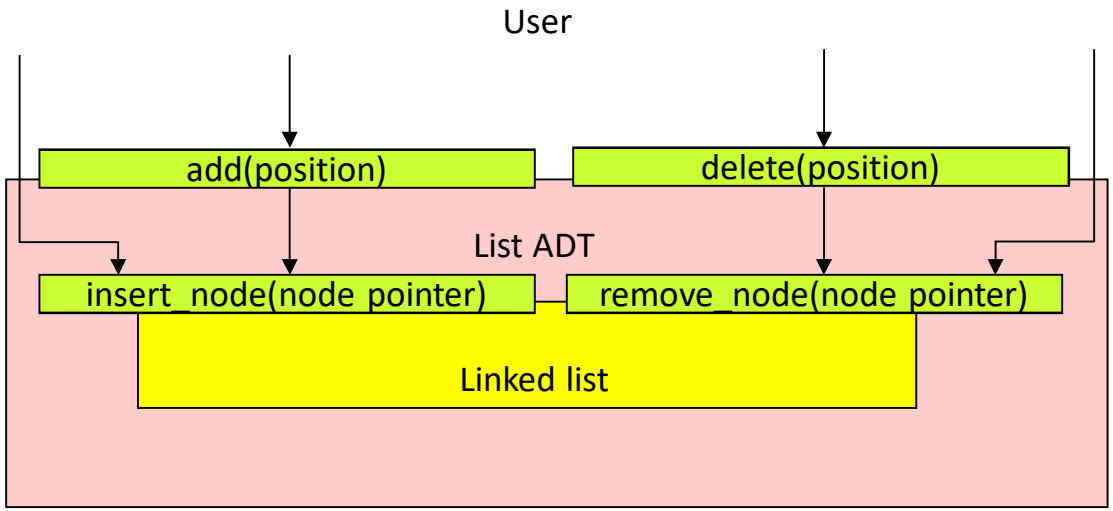
```
        for (; a != NULL; a = a->link)
                insert_node_last(plist3, a->coef, a->expon);
        for (; b != NULL; b = b->link)
                insert_node_last(plist3, b->coef, b->expon);
}
//
void poly_print(ListHeader *plist)
{
        ListNode *p = plist->head;
        for (; p; p = p->link) {
                printf("%d %d\n", p->coef, p->expon);
        }
}
```

Linked List Application: Polynomial

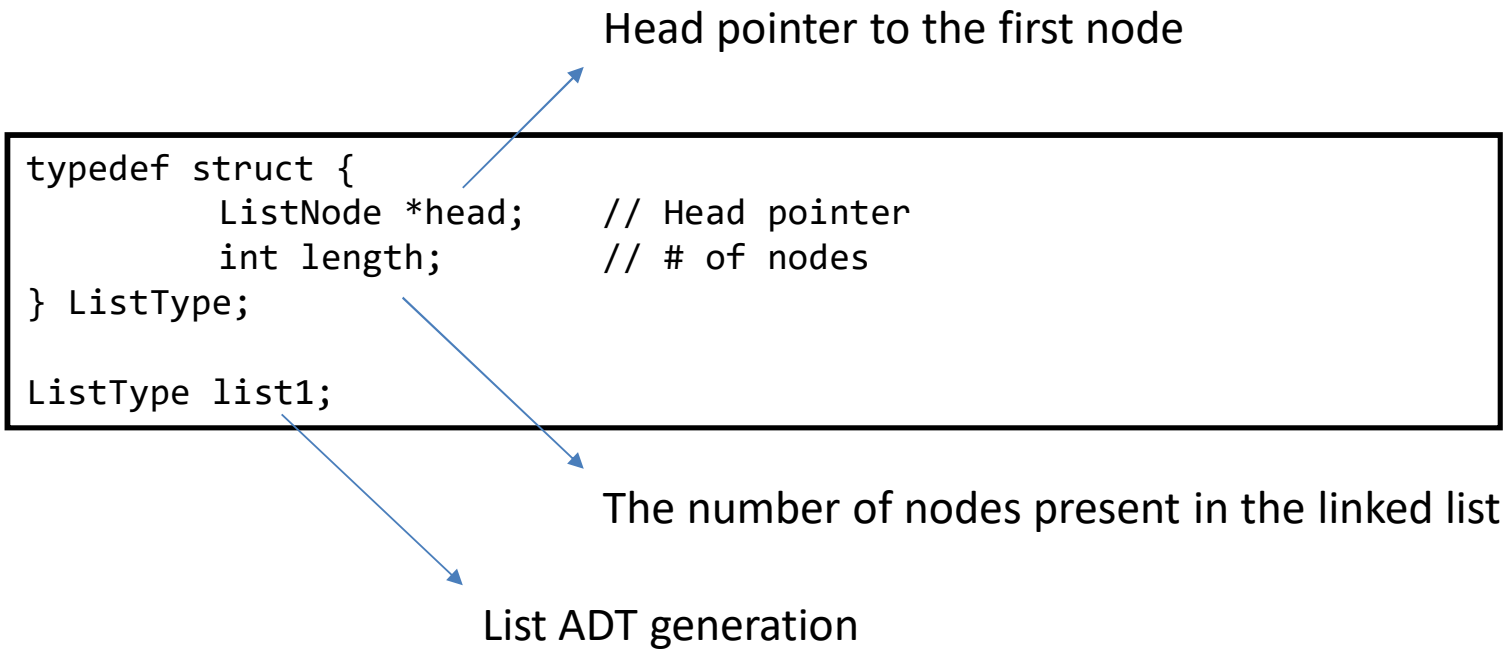
```
//
void main()
{
        ListHeader list1, list2, list3;

        // Initialization of linked list
        init(&list1);
        init(&list2);
        init(&list3);
        // Generate polynomial 1
        insert_node_last(&list1, 3, 12);
        insert_node_last(&list1, 2, 8);
        insert_node_last(&list1, 1, 0);
        // Generate polynomial 2
        insert_node_last(&list2, 8, 12);
        insert_node_last(&list2, -3, 10);
        insert_node_last(&list2, 10, 6);
        // Poly 3 = Ploy 1 + Poly 2
        poly_add(&list1, &list2, &list3);
        poly_print(&list3);
}
```

List ADT using Linked List

- List ADT using the linked list
  - The parameter of addition and deletion operation is position
  - The parameter of ‘insert_node’ and ‘remove_node’ is the node pointer



List ADT



List ADT

```
int is_empty(ListType *list)
{
        if (list->head == NULL) return 1;
        else return 0;
}
```

```
int get_length(ListType *list)
{
        return list->length;
}
```

List ADT

- Insertion operation
  - Insert new data in the list
  - ‘get_node_at’: converts a position to node pointer.

```
// Return node pointer of ‘pos’ in the list.
ListNode *get_node_at(ListType *list, int pos)
{
        int i;
        ListNode *tmp_node = list->head;
        if (pos < 0) return NULL;
        for (i = 0; i<pos; i++)
                tmp_node = tmp_node->link;
        return tmp_node;
}
```

List ADT

- Insertion operation

```
// Insert new data at the ‘position’
void add(ListType *list, int position, element data)
{
        ListNode *p;
        if ((position >= 0) && (position <= list->length)) {
                ListNode *node = (ListNode *)malloc(sizeof(ListNode));
                if (node == NULL) error("Memory allocation error");
                node->data = data;
                p = get_node_at(list, position - 1);
                insert_node(&(list->head), p, node);
                list->length++;
        }
}
```

List ADT

- Deletion operation
  - delete a data in the list
  - ‘get_node_at’: converts a position to node pointer.

```
// delete a data at the ‘pos’ in the list
void delete(ListType *list, int pos)
{
        if (!is_empty(list) && (pos >= 0) && (pos < list->length)) {
                ListNode *p = get_node_at(list, pos - 1);
                ListNode *removed = get_node_at(list, pos);
                remove_node(&(list->head), p, removed);
                list->length--;
        }
}
```

List ADT

- ‘get_entry’ operation

```
// Return the data at the ‘pos’.
element get_entry(ListType *list, int pos)
{
        ListNode *p;
        if (pos >= list->length) error(“Position error");
        p = get_node_at(list, pos);
        return p->data;
}
```

List ADT

- Display operation

```
// Display data in the buffer.
void display(ListType *list)
{
        int i;
        ListNode *node = list->head;
        printf("( ");
        for (i = 0; i<list->length; i++) {
                printf("%d ", node->data);
                node = node->link;
        }
        printf(" )\n");
}
```

List ADT

- ‘is_in_list’ operation

```
// Find a node whose data = item
int is_in_list(ListType *list, element item)
{
        ListNode *p;
        p = list->head;
        while ((p != NULL)) {
                if (p->data == item)
                break;
                p = p->link;
        }
        if (p == NULL) return FALSE;
        else return TRUE;
}
```

Example Code

```
int main()
{
        ListType list1;
        init(&list1);
        add(&list1, 0, 20);
        add_last(&list1, 30);
        add_first(&list1, 10);
        add_last(&list1, 40);

        // list1 = (10, 20, 30, 40)
        display(&list1);

        // list1 = (10, 20, 30)
        delete(&list1, 3);
        display(&list1);

        // list1 = (20, 30)
        delete(&list1, 0);
        display(&list1);
        printf("%s\n", is_in_list(&list1, 20) == TRUE ? "TRUE": "FALSE");
        printf("%d\n", get_entry(&list1, 0));
}
```