Linked List

Sequential List

- Definition
  - 데이터의 논리적인 순서와 물리적인 순서가 일치하는 data structure
  - 원소의 순서를 주소 (또는 array index) 에 의하여 관리

  (Ex) array

[1]	[2]	[3]	[4]	[5]	[6]	[7]
C	E	G	J	K	P	X

- 문제점
  - 데이터 삽입 시: Insert 'F'
    - O(n)
  - 데이터 삭제 시: Delete 'G'
    - O(n)
  - ➢ 데이터의 삽입 / 삭제가 빈번한 경우 사용 어려움

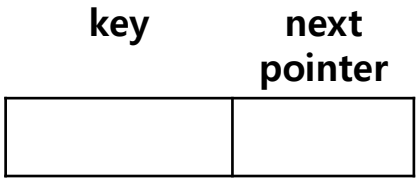
Linked List

- Definition
  - 데이터의 논리적인 순서와 물리적인 순서가 일치하지 않는 data structure
  - 별도의 pointer (또는 link) 에 의하여 원소의 순서가 결정

- Type
  - Singly linked list
  - Doubly linked list
- 특징
  - 데이터 삽입: O(1)
  - 데이터 삭제: O(1)
  - ➢ 데이터 삽입 / 삭제가 빈번한 경우 사용 유리
  - ➢ 메모리의 효과적 사용 가능

Singly Linked List

- 원소 (Node) 구성

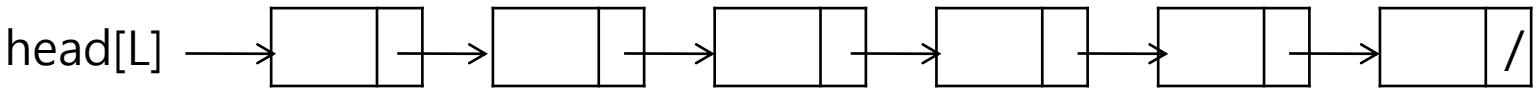


(1) Key field
- 원소의 데이터 저장

(2) Pointer field
- 다음 원소의 주소 저장

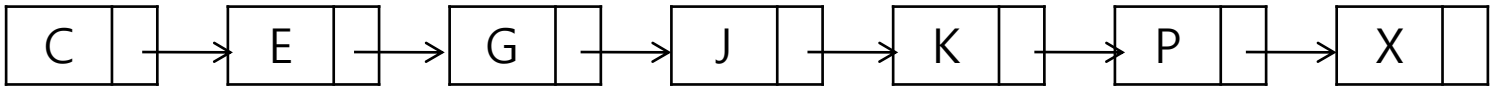
Singly Linked List

- Data Member
  - L (list) : key fields + next pointer fields (next)
  - head[L] : list L 의 첫 원소의 주소
    - List empty: head[L]= NIL
    - Last element: next[x] = NIL;



Singly Linked List

- Array Implementation

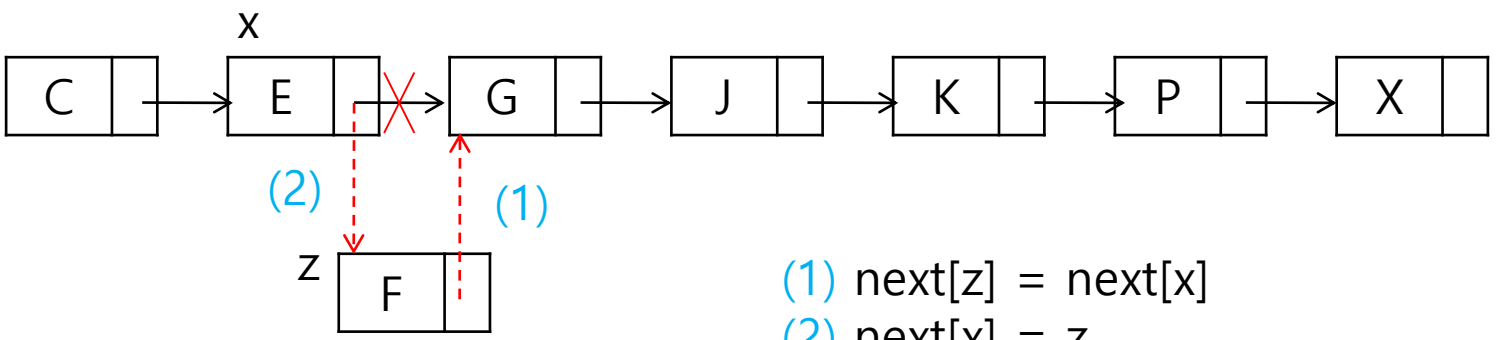


head[L]=3

	key[ ]	next[ ]
1	G	4
2		
3	C	5
4	J	8
5	E	1
6	X	NIL
7		
8	K	9
9	P	6
10		

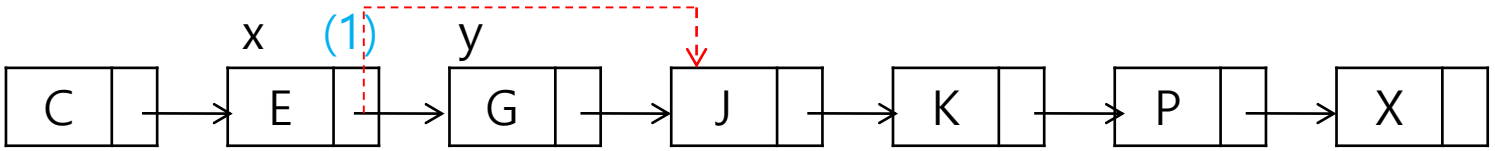
Singly Linked List

- Insert Operation
  - 주소 z 인 element 를 주소 x 의 element 다음에 삽입



(1) next[z] = next[x]
(2) next[x] = z

- Delete Operation
  - 주소 y 의 element 를 list에서 삭제

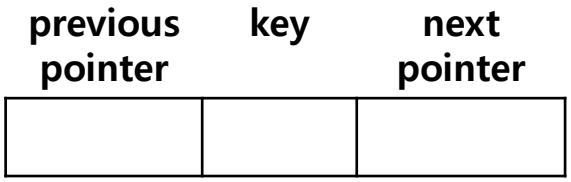


(1) next[x] = next[y]

x 를 어떻게 찾나?
⇒LIST-SEARCH
⇒O(n)

Doubly Linked List

- 원소 (Node) 구성



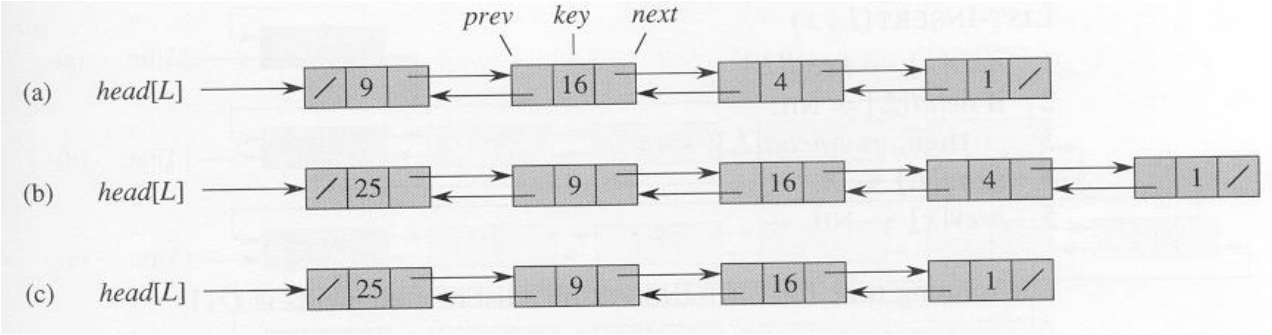
(1) Key field
  - 원소의 데이터 저장

(2) Next pointer field
  - 다음 원소의 주소 저장

(3) Previous pointer field
  - 직전 원소의 주소 저장

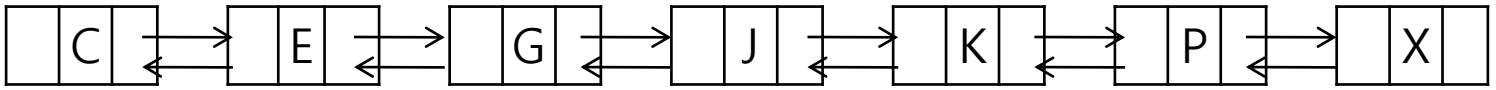
Doubly Linked List

- Data Member
  - L (list) : key fields + next pointer fields (next) + previous pointer fields (prev)
  - head[L] : list L 의 첫 원소의 주소
    - List empty: head[L]= NIL
    - Last element: next[x] = NIL
    - First element: prev[x] = NIL



Doubly Linked List

- Array Implementation



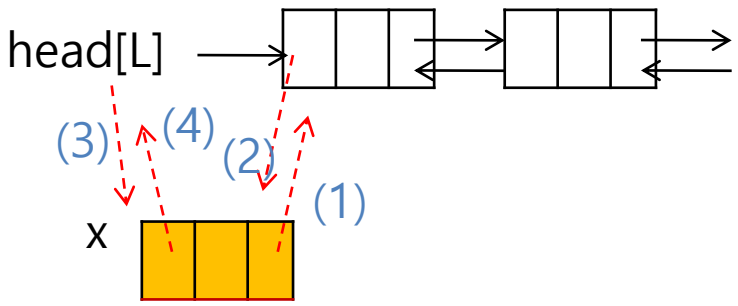
head[L]=3

	key[ ]	next[ ]	prev[ ]
1	G	4	5
2			
3	C	5	NIL
4	J	7	1
5	E	1	3
6	X	NIL	9
7			
8	K	9	4
9	P	6	8
10			

Doubly Linked List

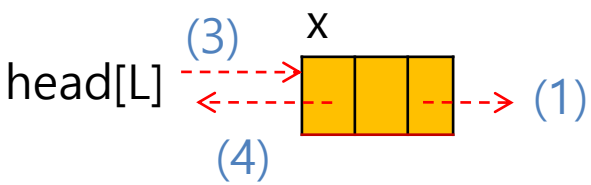
- LIST-INSERT(L, x)
  - list L 의 front 에 x 주소를 갖는 원소 삽입
  - O(1)

(1) List non-empty 인 경우 (head[L] != NIL)



(1) next[x] = head[L]
(2) prev[head[L]] = x
(3) head[L]=x
(4) prev[x]=NIL

(2) List empty 인 경우 (head[L] = NIL)

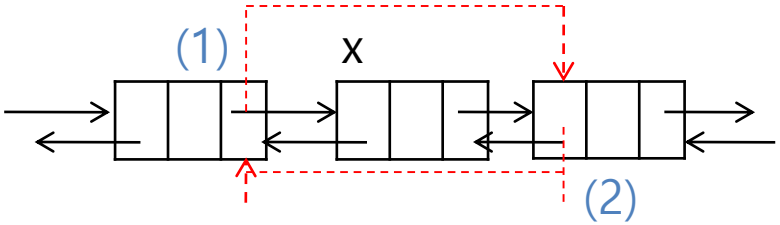



```
LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3      then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

Doubly Linked List

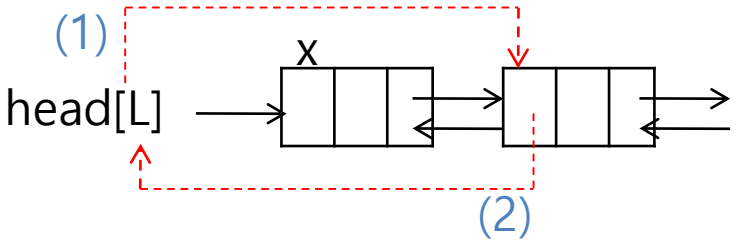
- LIST-DELETE(L, x)
  - list L 의 x 주소에 있는 원소 삭제
  - O(1)

(1) first element 가 아닌 경우



(1) next[prev[x]] = next[x]
(2) prev[next[x]] = prev[x]

(2) first element 인 경우



(1) head[L] = next[x]
(2) prev[next[x]] = prev[x]

```
LIST-DELETE(L, x)
1  if prev[x] ≠ NIL
2     then next[prev[x]] ← next[x]
3     else head[L] ← next[x]
4  if next[x] ≠ NIL
5     then prev[next[x]] ← prev[x]
```

Doubly Linked List

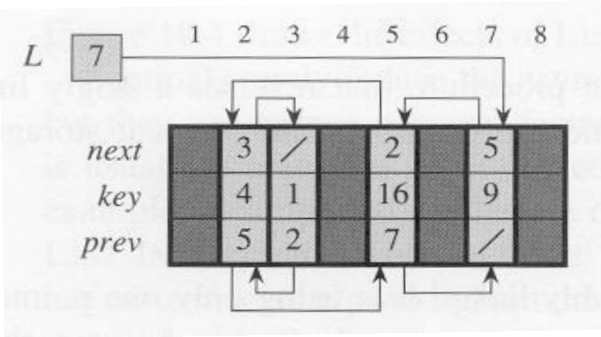
- LIST-SEARCH(L, k)
  - list L 에서 key 값이 k 인 원소를 찾아, 그 주소 return
  - O(n)

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3        do x ← next[x]
4  return x
```

Doubly Linked List

- Implementation
  1) Multiple Array

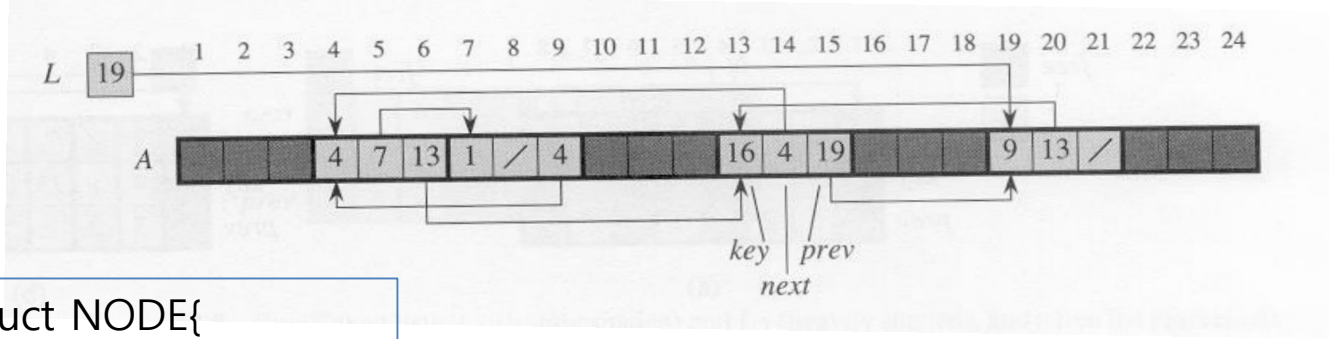
     key[ ], next[ ], prev[ ] 3개의 array 사용



  2) Single Array

     하나의 array 사용

     A[i] → key, A[i+1] → next, A[i+2] → prev



  3) Node

```
struct NODE{
    int key;
    NODE* next;
    NODE* prev;
};
NODE* head;
```

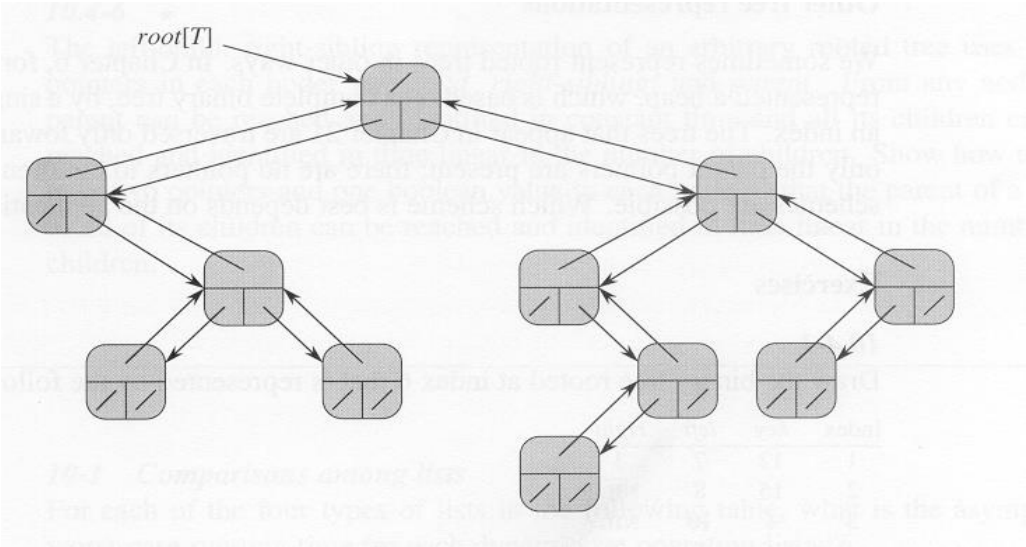
Rooted Tree Representation

- Binary Tree
  - Left-child, right-child representation
  - Data member

    Node: key + parent pointer + left child pointer + right child pointer

    T: node 의 집합

    Root[T]: root node 의 index



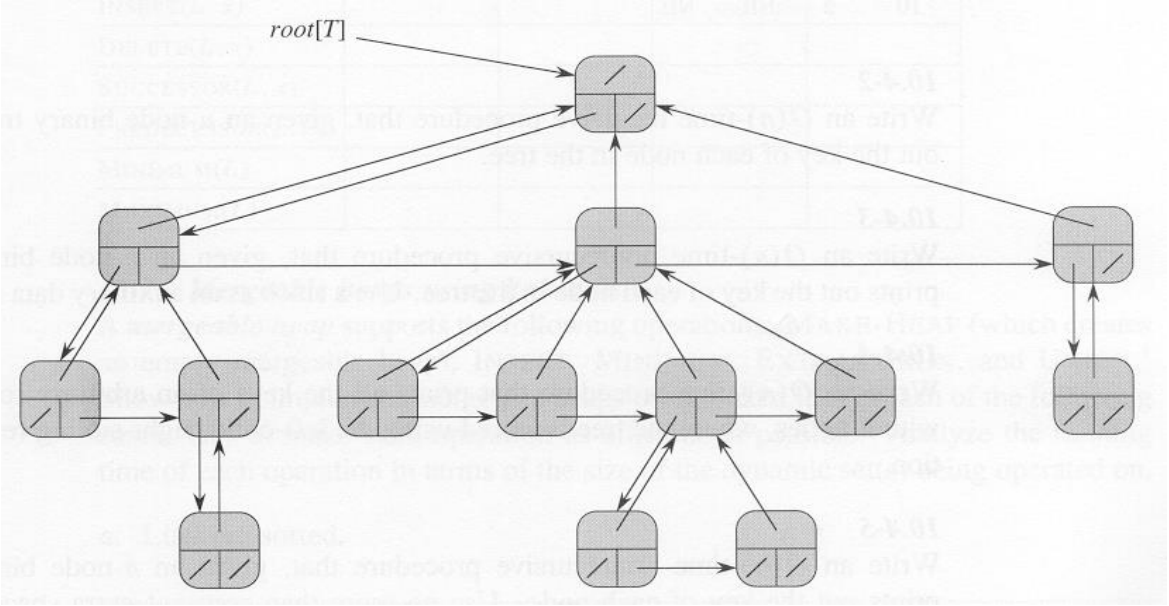
Rooted Tree Representation

- 일반 Tree
  - Left-child, right-sibling representation
  - Data member

    Node: key + parent pointer + left child pointer + right sibling pointer

    T: node 의 집합

    Root[T]: root node 의 index



Polynomials

- Singly linked list 를 이용하여 다항식 표현

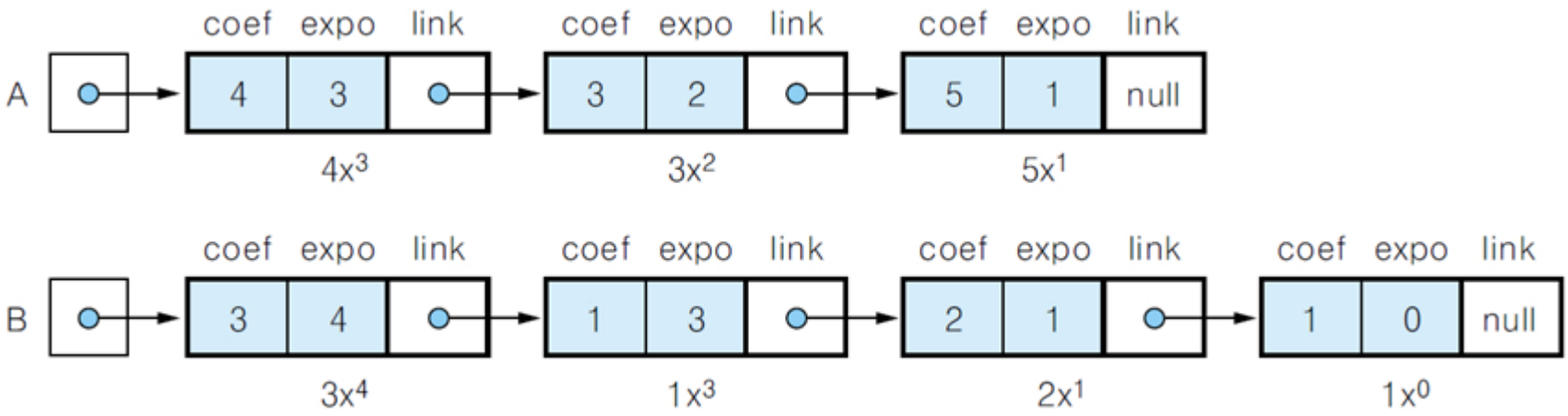
$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

- 원소 (node) 구성
  - Key field = coef (계수) + expo (지수)
  - Next pointer field = link (다음 항 연결)


```
typedef  struct  Node
{
        float coef;
        int expo;
        struct Node *link;
};
```

Polynomials

- 표현 예
  - $A(x)=4x^3+3x^2+5x$
  - $B(x)=3x^4+x^3+2x+1$



⇒ 효과적 메모리 사용

(cf) sequential list

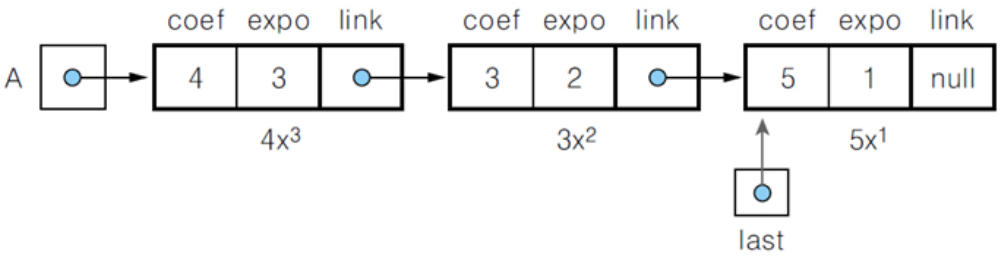
Polynomials

- AppendTerm (PL, coef, expo, last)
  - 다항식의 항 1 개 (coef, expo) 를 추가하는 알고리즘

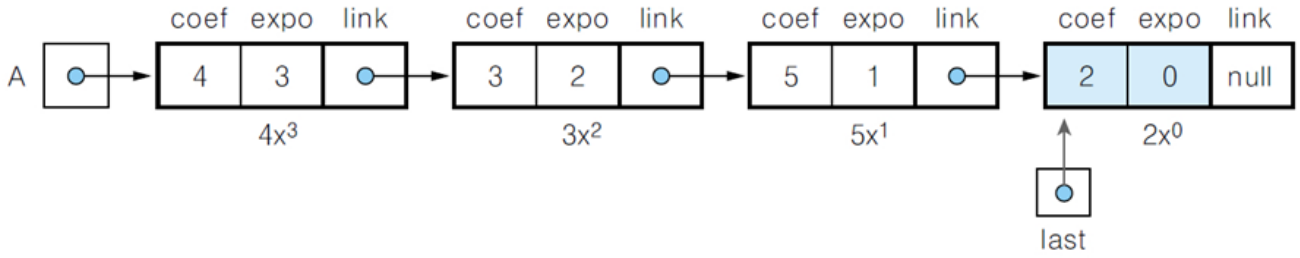
    PL: list head

    last: list 마지막 노드의 주소

(ex) $A(x)=4x^3+3x^2+5x+2$



(a) appendTerm(A,2,0,last) 함수 실행 전의 다항식 리스트 A



(b) appendTerm(A,2,0,last) 함수 실행 후의 다항식 리스트 A

Polynomials

- Append Term

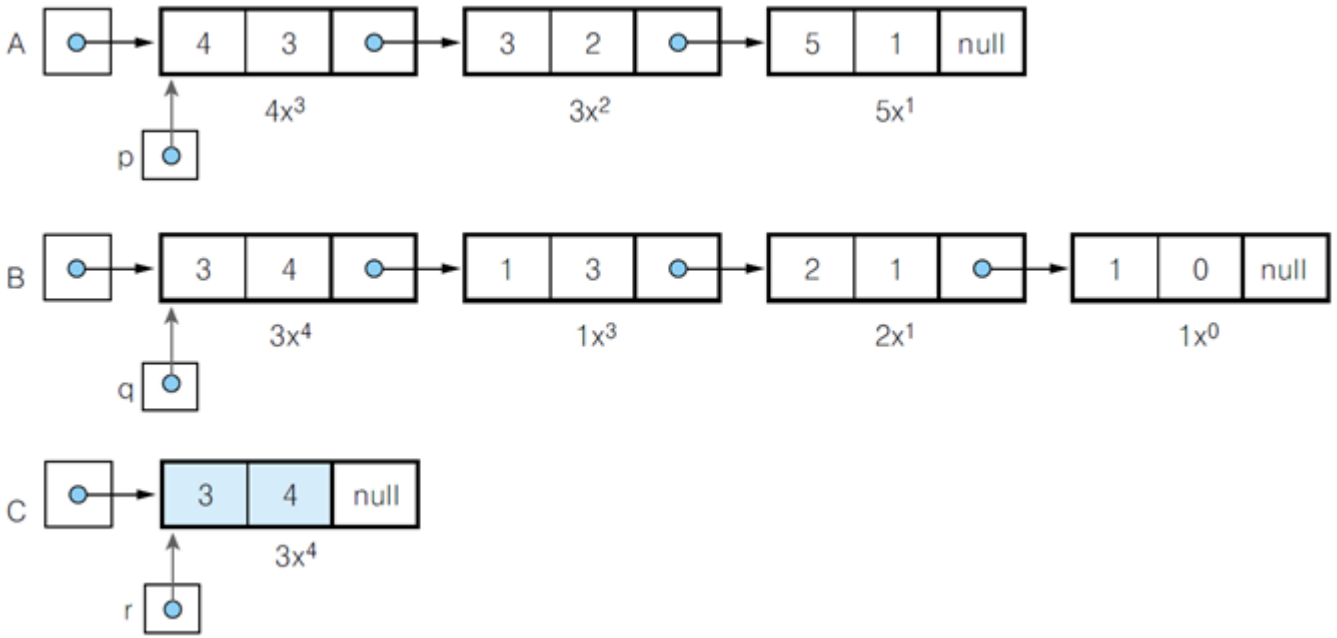
```
appendTerm(PL, coef, expo, last)
    new ← getNode();
    new.expo ← expo;
    new.coef ← coef;
    new.link ← null;
    if (PL = null) then {          // ❶ 공백 리스트인 경우
        PL ← new;                  // ❶-ⓐ
        last ← new;                // ❶-ⓑ
    }
    else {                         // ❷ 공백 리스트가 아닌 경우
        last.link ← new;           // ❷-ⓐ
        last ← new;                // ❷-ⓑ
    }
end appendTerm()
```

Polynomials

- Addition
  - 덧셈 A(x)+B(x)=C(x)를 단순 연결 리스트 자료구조를 사용하여 연산
  - 다항식 A(x)와 B(x), C(x)의 항을 지시하기 위해서 세 개의 포인터를 사용
    - 포인터 p : 다항식 A(x)에서 비교할 항을 지시
    - 포인터 q : 다항식 B(x)에서 비교할 항을 지시
    - 포인터 r : 덧셈연산 결과 만들어지는 다항식 C(x)의 항을 지시

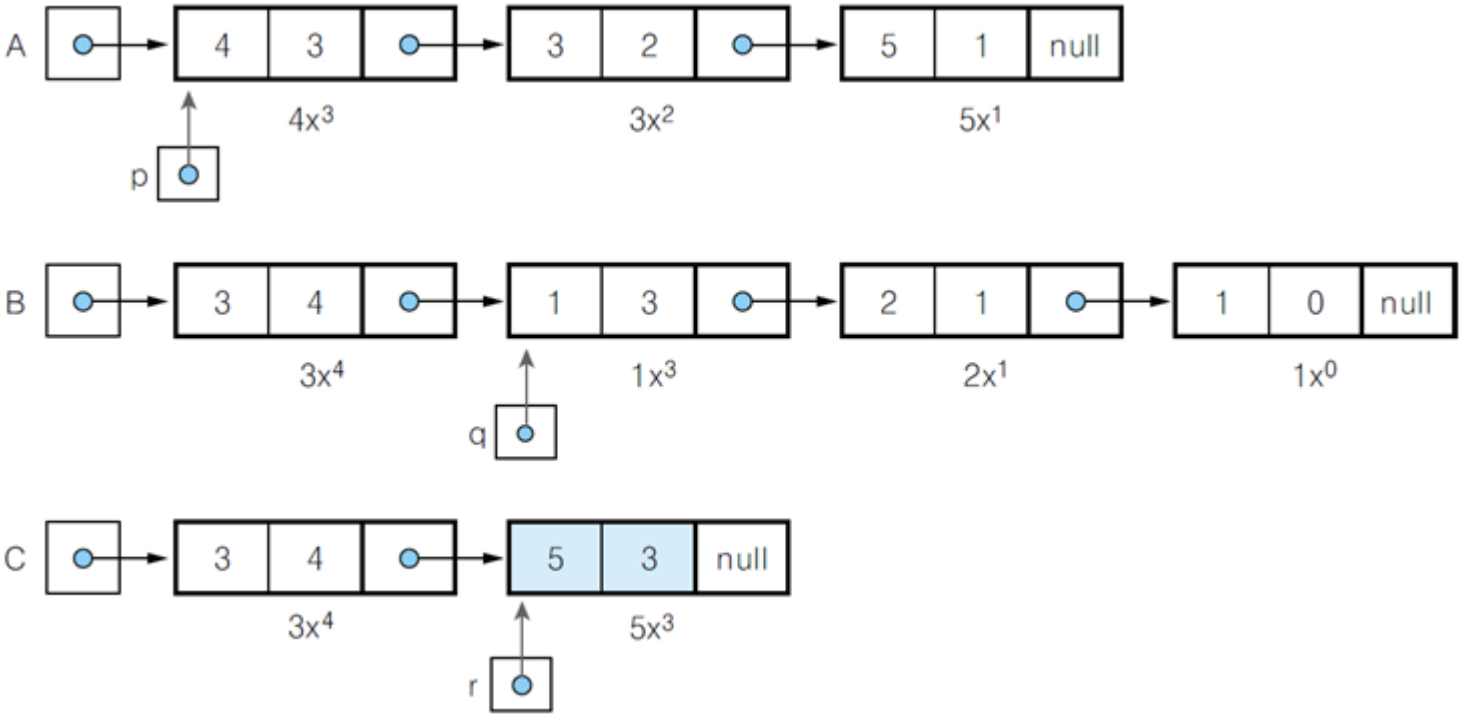
Polynomials

- Addition Algorithm
  - p.expo < q.expo : 다항식 B(x) 항의 지수가 큰 경우
    - 포인터 q가 가리키는 다항식 B(x)의 항을 C(x)의 항으로 복사
    - q가 가리키는 항에 대한 처리가 끝났으므로 q를 다음 노드로 이동



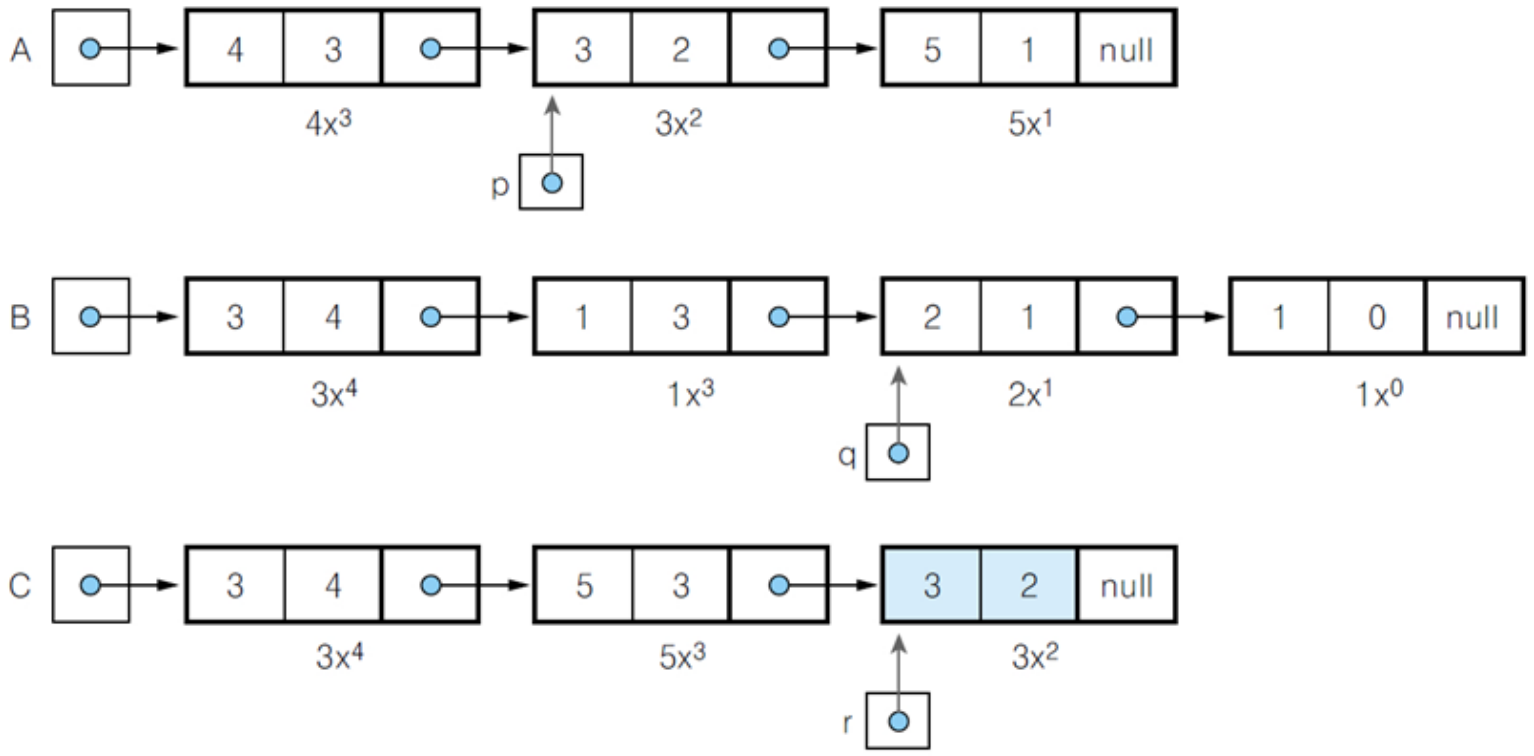
Polynomials

- Addition Algorithm
  - p.expo = q.expo : 두 항의 지수가 같은 경우
    - p.coef와 q.coef를 더하여 C(x)의 항, 즉 r.coef에 저장하고 지수가 같으므로 p.expo(또는 q.expo)를 r.expo에 저장
    - 다음 항을 비교하기 위해 포인터 p와 q를 각각 다음 노드로 이동



Polynomials

- Addition Algorithm
  - p.expo > q.expo : 다항식 A(x) 항의 지수가 큰 경우
    - 포인터 p가 가리키는 다항식 A(x)의 항을 C(x)의 항으로 복사
    - p가 가리키는 항에 대한 처리가 끝났으므로 p를 다음 노드로 이동



Polynomials

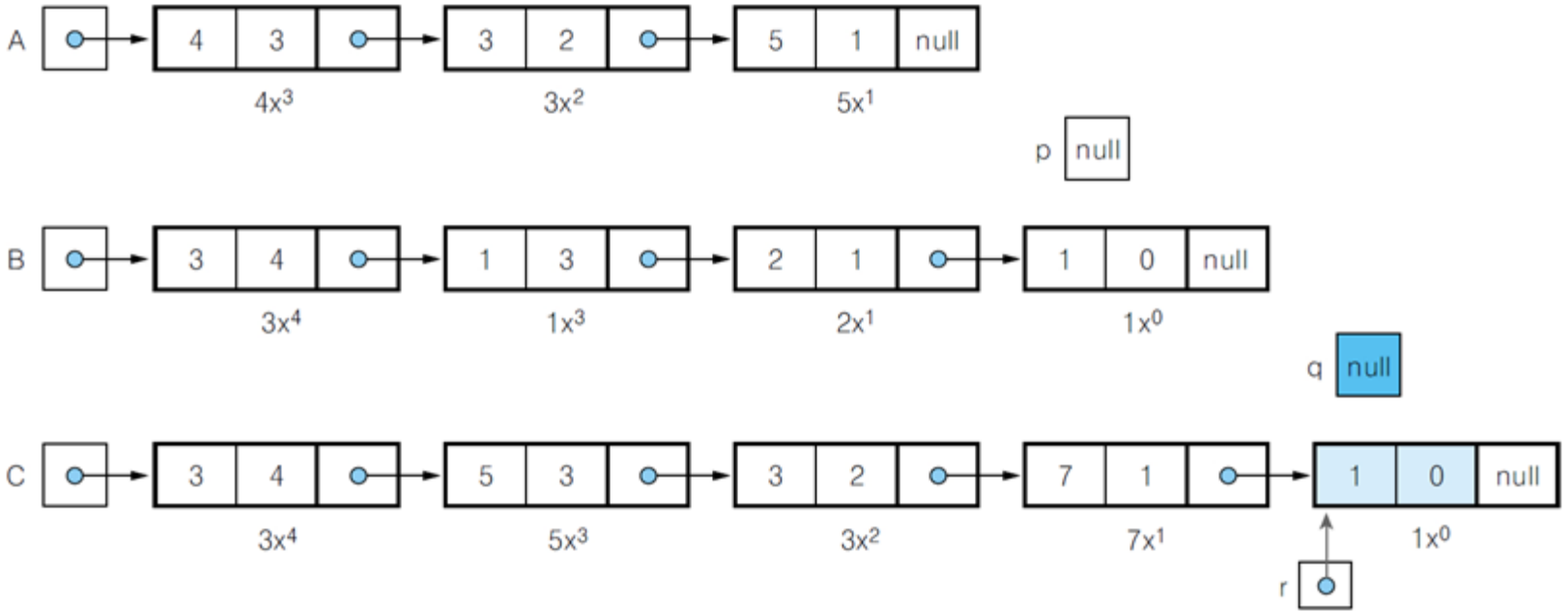
- Addition Algorithm

```
addPoly(A, B)
    // 단순 연결 리스트로 표현된 다항식 A와 B를 더하여
    // 새로운 다항식 C를 반환
    p ← A;
    q ← B;
    C ← null;       // 결과 다항식
    r ← null;       // 결과 다항식의 마지막 노드를 지시
    while (p ≠ null and q ≠ null) do {      // p, q는 순회 포인터
         case {
            p.expo = q.expo :
                      sum ← p.coef + q.coef
                      if (sum ≠ 0) then appendTerm(C, sum, p.expo, r);
                      p ← p.link;
                      q ← q.link;
            p.expo < q.expo :
                     appendTerm(C, q.coef, q.expo, r);
                     q ← q.link;
                else :      // p.expo > q.expo인 경우
                     appendTerm(C, p.coef, p.expo, r);
                     p ← p.link;
          } // end case
    } // end while

    while (p ≠ null) do {      // A의 나머지 항들을 복사
           appendTerm(C, p.coef, p.expo, r);
           p ← p.link;
    }
    while (q ≠ null) do {      // B의 나머지 항들을 복사
          appendTerm(C, q.coef, q.expo, r);
          q ← q.link;
    }
    return C;
end addPoly()
```

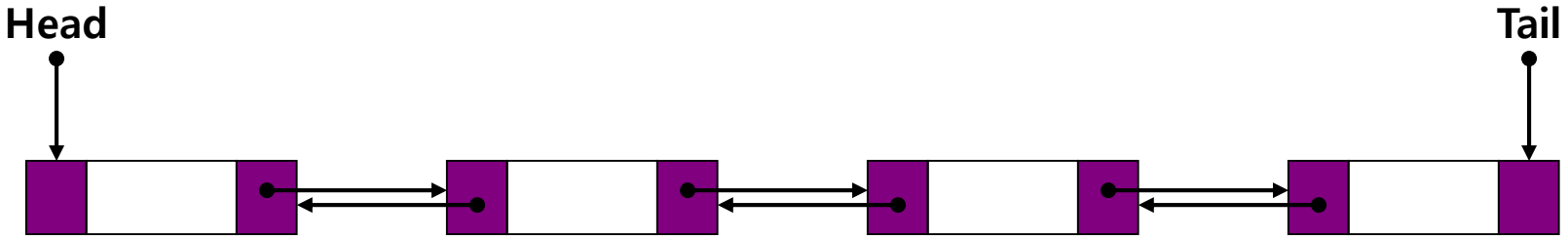
Polynomials

- Addition Results



리스트 클래스

- MFC 리스트 클래스



- 템플릿 클래스
  - afxtempl.h 헤더 파일

클래스 이름	데이터 타입	사용 예
CList	프로그래머가 결정	CList<CPoint, CPoint&> list;

리스트 클래스

- 비 템플릿 클래스
  - afxcoll.h 헤더 파일

클래스 이름	데이터 타입	사용 예
CObList	CObject 포인터	CObList list;
CPtrList	void 포인터	CPtrList list;
CStringList	CString	CStringList list;

리스트 클래스 (3/8)

- 생성과 초기화

```
char *szFruits[ ] = {
    "사과",
    "딸기",
    "포도",
    "오렌지",
    "자두"
};

CStringList list;
for(int i=0; i<5; i++)
    list.AddTail(szFruits[i]);
```

리스트 클래스

- 순환

```
// 리스트 제일 앞에서 출발하여 순환한다.
POSITION pos = list.GetHeadPosition();
while(pos != NULL){
    CString string = list.GetNext(pos);
    cout << (LPCTSTR)string << endl;
}
cout << endl;

// 리스트 제일 뒤에서 출발하여 순환한다.
pos = list.GetTailPosition();
while(pos != NULL){
    CString string = list.GetPrev(pos);
    cout << (LPCTSTR)string << endl;
}
```

리스트 클래스

- 항목 삽입과 삭제

```
// POSITION 타입의 변수 pos는 이전의 예제에서 선언한 것이다.
pos = list.Find("포도");
list.InsertBefore(pos, "살구");
list.InsertAfter(pos, "바나나");
list.RemoveAt (pos);

// 항목 삽입과 삭제 후 결과를 확인한다.
pos = list.GetHeadPosition();
while(pos != NULL){
    CString string = list.GetNext(pos);
    cout << (LPCTSTR)string << endl;
}
```

리스트 클래스

- 템플릿 리스트 클래스

```
#include "stdafx.h"
#include "Console.h"
#include <afxtempl.h>

CWinApp theApp;

using namespace std;

struct Point3D {
    int x, y, z;
    Point3D() {}
    Point3D(int x0, int y0, int z0) { x = x0; y = y0; z = z0; }
};
```

리스트 클래스

```
int _tmain(int argc, TCHAR* argv[ ], TCHAR* envp[ ])
{
    int nRetCode = 0;

    if (!AfxWinInit(...))
    {
        // 생략 ...
    }
    else
    {
        CList<Point3D, Point3D&> list;
        for(int i=0; i<5; i++)
            list.AddTail(Point3D(i, i*10, i*100));
        POSITION pos = list.GetHeadPosition();
```

리스트 클래스

```
        while(pos != NULL) {
            Point3D pt = list.GetNext(pos);
            cout << pt.x << ", " << pt.y << ", " << pt.z << endl;
        }
    }

    return nRetCode;
}
```