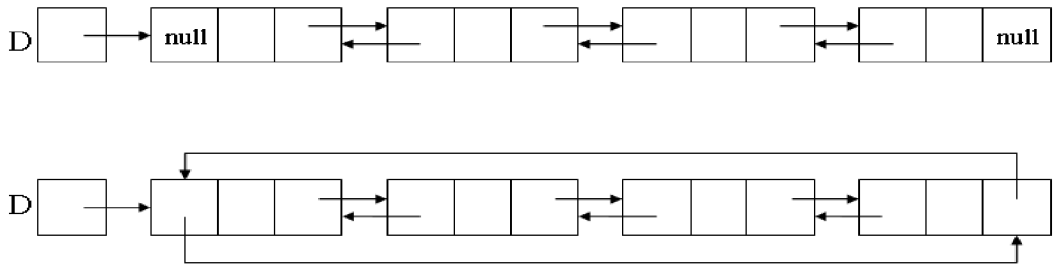
이중 연결 리스트 (doubly linked list)

① 3개의 필드 data, llink, rlink를 가진 노드로 구성
② 어떤 노드 p의 선행자를 쉽게 찾을 수 있음
③ p = p.llink.rlink = p.rlink.llink



노드 삭제

```
deleteD(D, p)
                    // D는 공백이 아닌 이중 연결 리스트, p는 삭제할 노드
          if (p = null) then return;
          p.llink.rlink ← p.rlink;
          p.rlink.llink ← p.llink;
end deleteD()
```

노드 삽입

```
insertD(D, p, q)
                    // D는 이중 연결 리스트이고, 노드 q를 노드 p 다음에 삽입
          q.llink ← p;
          q.rlink ← p.rlink;
          p.rlink.llink ← q;
          p.rlink ← q;
end deleteD()
```

헤더 노드

기존의 연결 리스트 처리 알고리즘

- 첫 번째 노드나 마지막 노드, 그리고 리스트가 공백인 경우를 예외적인 경우로 처리해야 함

헤더 노드 (header node) 추가

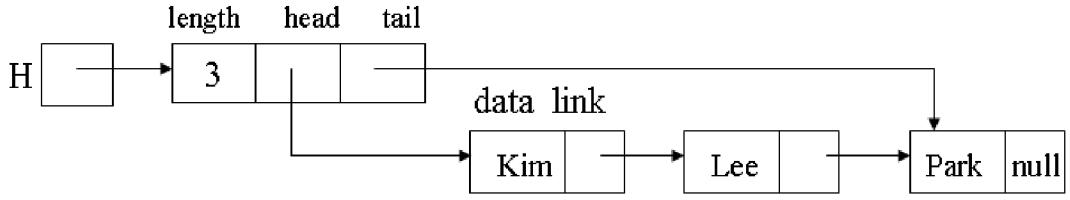
① 예외 경우를 제거하고 코드를 간단하게 하기 위한 방법
② 연결 리스트를 처리하는 데 필요한 정보를 저장
③ 헤더 노드의 구조가 리스트의 노드 구조와 같을 필요는 없음
④ 헤더 노드에는 리스트의 첫 번째 노드를 가리키는 포인터, 리스트의 길이, 참조 계수, 마지막 노드를 가리키는 포인터 등의 정보를 저장

헤더 노드 기능을 갖는 연결 리스트의 정의

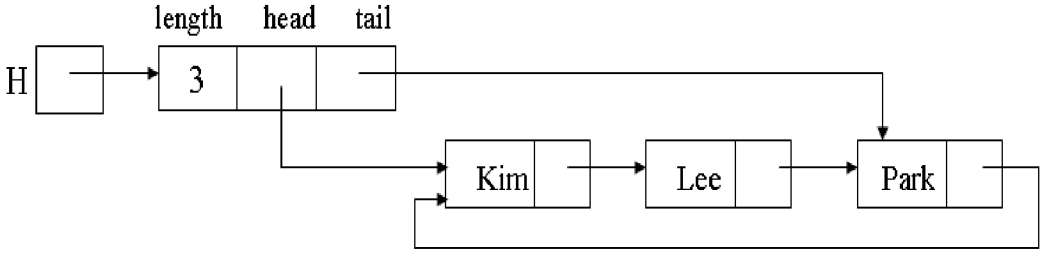
```
typedef struct listNode {
                    /* 리스트 노드 구조 */
                        char data[5];
                        struct listNode* link;
    } listNode;

    typedef struct {    /*리스트 헤더 노드 구조 */
                    int length;       /* 리스트의 길이(노드 수) */
                    listNode* head;    /* 리스트의 첫 번째 노드에 대한 포인터 */
                    listNode* tail;         /* 리스트의 마지막 노드에 대한 포인터 */
    } h_linkedList ;

void addLastNode(h_linkedList* H, char* x) {
                   /*  헤더  노드를  가진  연결  리스트의  끝에  원소  삽입  */

                                ?
}
void reverse(h_linkedList* H) {
                  /* 헤더 노드를 가진 연결 리스트의 원소를 역순으로 변환 */
                                ?
}
void deleteLastNode(h_linkedList* H) {
                   /* 헤더 노드를 가진 연결 리스트의 마지막 원소를 삭제 */
                                ?
}
void printList(h_linkedList* H) {
                    /* 헤더 노드를 가진 연결 리스트의 원소들을 프린트 */
                                ?
}

/* 기타 다른 함수 정의 */
```

헤더 노드를 가진 연결 리스트 표현

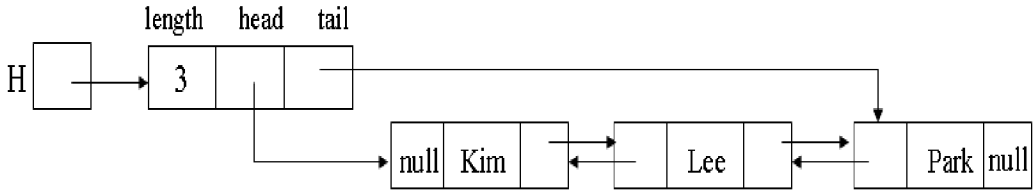
단순 연결 리스트



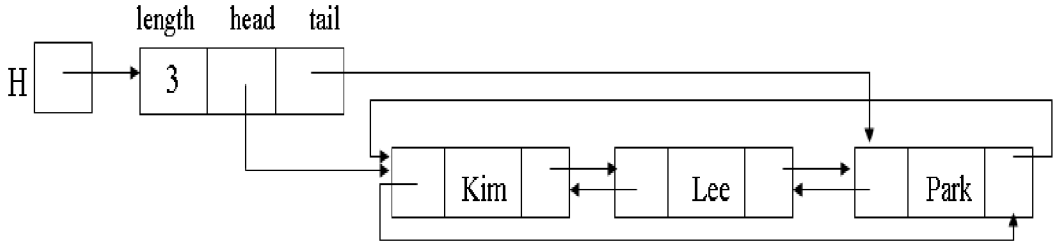
원형 연결 리스트



이중 연결 리스트

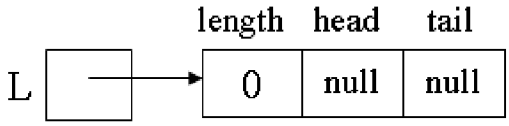


이중 원형 연결 리스트

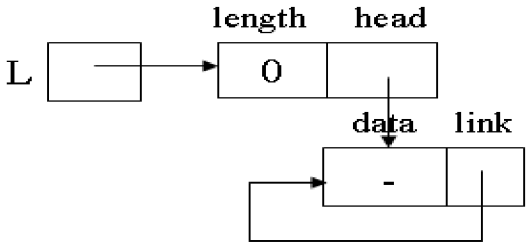


공백 리스트

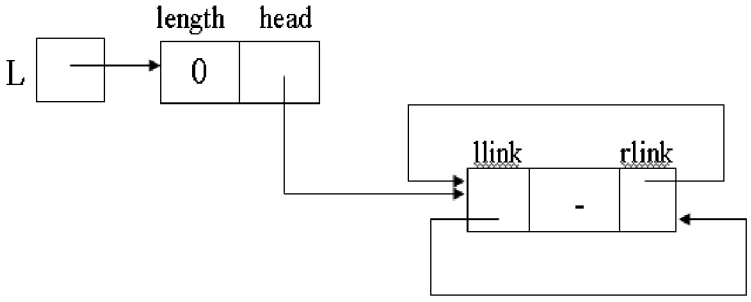
length가 0이고 head가 null, tail이 null



공백 리스트를 표현하는 단순 연결 원형 리스트의 헤더 노드의 구조



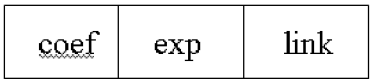
공백 리스트를 표현하는 이중 연결 원형 리스트의 헤더 노드의 구조



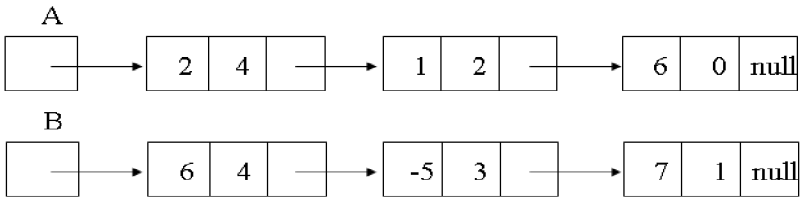
다항식의 표현

① 다항식은 일반적으로 0이 아닌 항들의 합으로 표현
② 다항식을 단순 연결 리스트로 표현 가능한데 각 항을 하나의 노드로 표현.
③ 각 노드는 계수(coef)와 지수(exp) 그리고, 다음 항을 가리키는 링크(link) 필드로 구성

다항식 노드 :



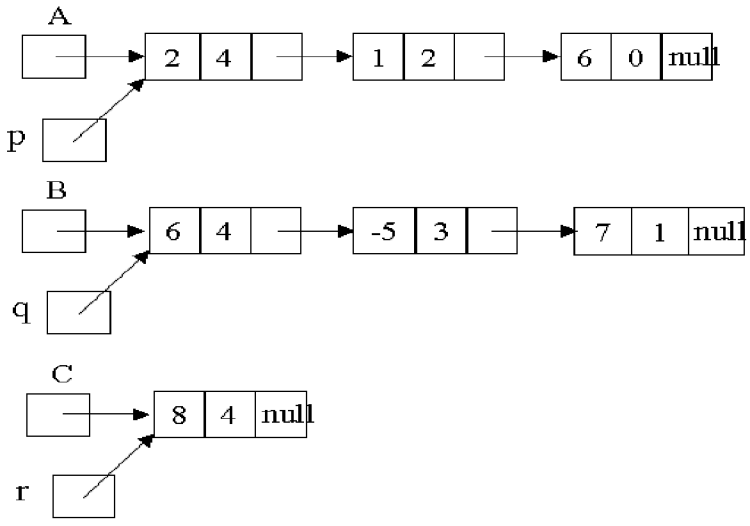
④ 예) 다항식
$A(x) = 2x^4 + x^2 + 6$ 과 $B(x) = 6x^4 - 5x^3 + 7x$을 표현



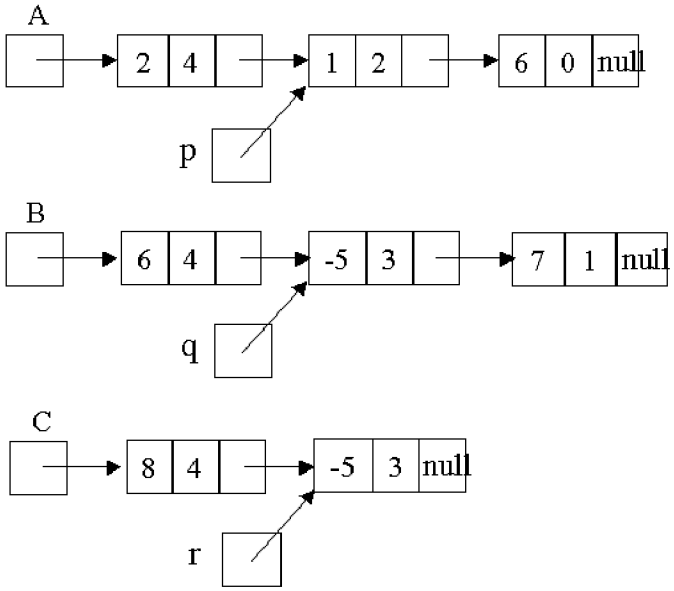
두 다항식을 더하는 연산 C(x) ← A(x) + B(x)

- 포인터 변수 p와 q : 다항식 A와 B의 항들을 따라 순회하는 데 사용
- p와 q가 가리키는 항의 지수에 따라 3가지 경우에 따라 처리
  - ① p.exp = q.exp : 두 계수를 더해서 0이 아니면 새로운 항을 만들어 결과 다항식 C에 추가한다. 그리고 p와 q는 모두 다음 항으로 이동한다.
  - ② p.exp < q.exp : q가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 C에 추가한다. 그리고 q만 다음 항으로 이동한다.
  - ③ p.exp > q.exp : p가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 C에 추가한다. 그리고 p만 다음 항으로 이동한다.

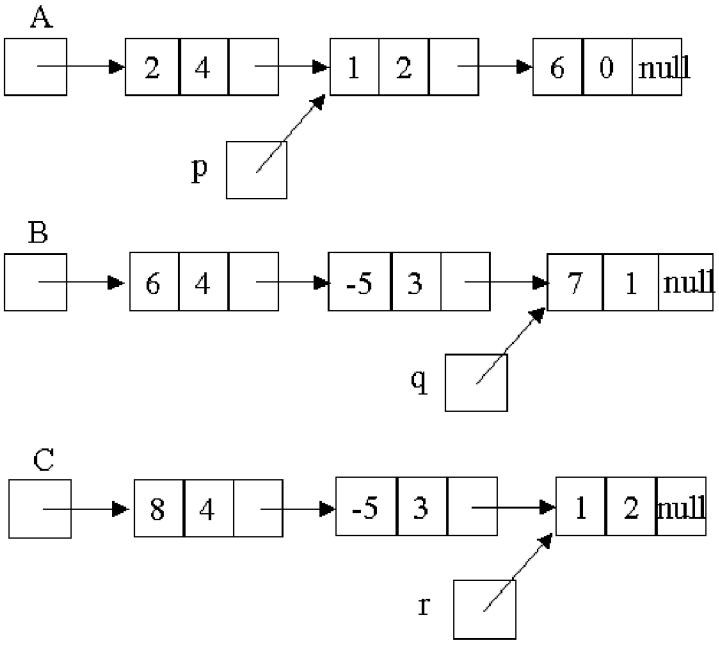
p.exp = q.exp :



p.exp ＜ q.exp ：



p.exp ＞ q.exp ：



다항식에 새로운 항을 첨가

```
appendTerm(poly, c, e, last)
          // c는 계수, e는 지수, last는 poly의 마지막 항을 가리키는 포인터
        newNode ← getNode();
        newNode.exp ← e;
        newNode.coef ← c;
        if (poly = null) then {
                  poly ← newNode;
                  last ← newNode;
        }
        else {
                  last.link ← newNode;
                  last ← newNode;
        }
end appendTerm()
```

연결 리스트로 표현된 다항식의 덧셈

```
polyAdd(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.exp = q.exp :
                                sum ← p.coef + q.coef;
                                if sum ≠ 0 then appendTerm(C, sum, p.exp,
r);
                                p ← p.link;
                                q ← q.link;
                p.exp < q.exp :
                                appendTerm(C, q.coef, q.exp, r);
                                q ← q.link;
                else :       // p.exp > q.exp인 경우
                                appendTerm(C, p.coef, p.exp, r);
                                p ← p.link;
              } // end case
        } // end while

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

일반 리스트 (general list)

① n≥0개의 원소 e1, e2, · en의 유한 순차
② 리스트의 원소가 원자(atom)일 뿐 아니라 리스트도 허용
③ 리스트의 원소인 리스트를 서브리스트(sublist)라 함.
④ 리스트는 L=(e1, e2, · en)과 같이 표현함. L은 리스트 이름이고 n은 리스트의 원소수 즉 리스트의 길이가 됨
⑤ n≥1인 경우 첫번째 원소 e1을 L의 head 즉 head(L)라 하고, 첫번째 원소를 제외한 나머지 리스트 (e2, · en)을 L의 tail 즉 tail(L)이라 함
⑥ 공백 리스트에 대해서는 head와 tail은 정의되지 않음
⑦ 정의 속에 다시 리스트를 사용하고 있기 때문에 순환적 정의

일반 리스트의 예

(1) A=(a, (b, c)) : 길이가 2이고 첫 번째 원소는 a이고 두 번째 원소는 서브리스트 (b, c)이다.
  - 리스트 A에 대해 head(A) = a, tail(A) = ((b, c))
  - tail(A)에 대해 head(tail(A)) = (b,c). tail(tail(A)) = ()

(2) B=(A, A, ( )) : 길이가 3이고 처음 두 원소는 서브리스트 A이고 세 번째 원소는 공백 리스트이다. 여기서 리스트 A를 공용하고 있다.
  - 리스트 B에 대해 head(B) = A, tail(B) = (A, ())
  - tail(B)에 대해 head(tail(B)) = A, tail(tail(B)) = (())

(3) C=(a, C) : 길이가 2인 순환리스트로서 두 번째 원소 C는 무한리스트 (a, (a, (a, ...)에 대응된다.

(4) D=( ) : 길이가 0인 널(null) 또는 공백 리스트이다.

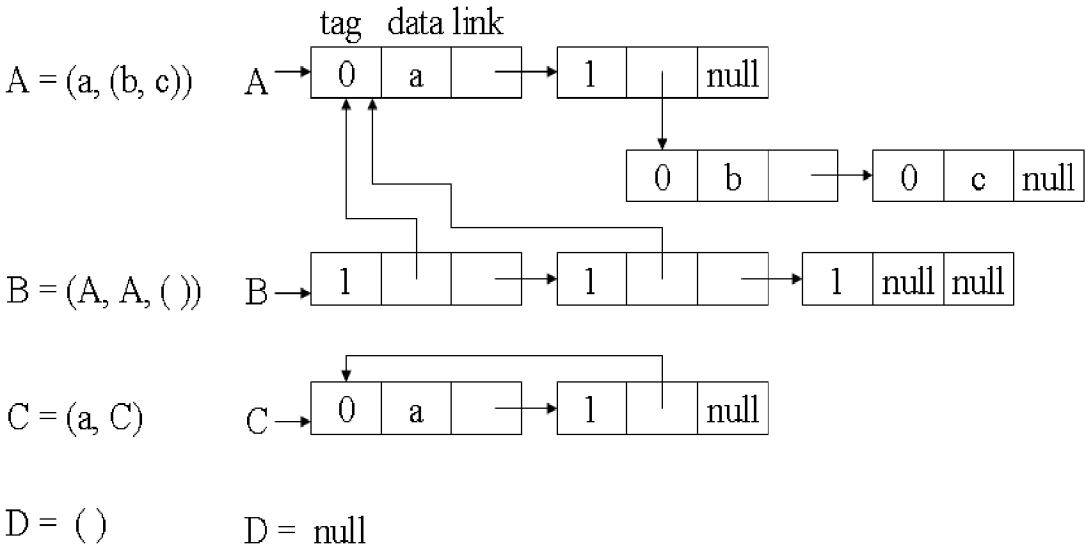
일반 리스트의 노드 구조

data 필드 : 리스트의 head를 저장
  - head(L)이 원자인가 또는 서브리스트인가에 따라 원자값 또는 서브리스트의 포인터가 저장됨

tag 필드 : data 필드 값이 원자인지 포인터 값인지를 표시
  - data 값이 원자값 : tag는 0
  - data 값이 서브리스트에 대한 포인터 값 : tag는 1

link 필드 : 리스트의 tail에 대한 포인터를 저장

앞의 리스트의 예



서브리스트의 공용

① 저장 공간을 절약할 수 있는 장점
② 공용 리스트 앞에 노드를 새로 삽입하거나 삭제할 때 그 리스트를 공용하는 리스트도 변경되어야 함
③ 앞의 일반 리스트 예에서
- 리스트 A의 첫 번째 노드 삭제 : A를 가리키는 포인터 값을 A의 두 번째 노드를 가리키도록 변경
- 새로운 노드를 리스트 A의 첫 번째 노드로 첨가 : B의 포인터들을 이 새로 삽입된 첫 번째 노드를 가리키도록 변경

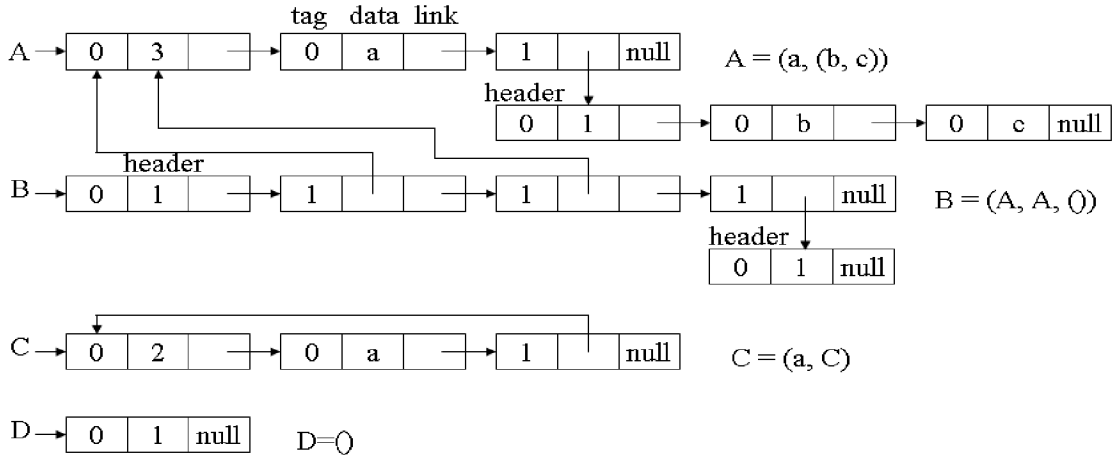
④ 한 리스트가 어떤 리스트에 의해 참조되는지 알 수 없으므로 연산 시간이 많이 걸림

헤더 노드 추가로 문제 해결

① 공용 리스트를 가리킬 때 헤더 노드를 가리키게 함
② 공용 리스트 내부에서 노드 삽입과 삭제가 일어나더라도 포인터는 영향을 받지 않게 됨

헤더 노드가 첨가된 리스트 표현

헤더 노드의 data 필드는 참조 계수(자기를 참조하고 있는 포인터 수)를 저장하는 데 사용



참조 계수 (reference count)

① 각 리스트를 참조하고 있는 포인터 수를 헤더 노드의 data 필드에 저장
② 리스트를 자유 공간 리스트에 반환할 것인가를 결정할 때, 리스트 헤더에 있는 참조 계수가 0인가만 확인하고 0일 때 반환하면 됨
③ 예)
- A.ref = 3 : A와 B의 두 링크에서 참조
- B.ref = 1 : B만 참조
- C.ref = 2 : C와 리스트 자체 내에서 참조
- D.ref = 1 : D만 참조

자유 공간 리스트로 리스트 반환

① A가 가리키는 리스트 삭제하려면 A의 참조 계수 1 감소
② A의 참조 계수 A.ref가 0이 되면 A의 노드들은 반환
③ A의 서브리스트에 대해서도 순환적으로 수행

```
removeList(L)
        // 헤더 노드의 ref 필드는 참조 계수를 저장
       L.ref ← L.ref - 1;       // 참조 계수를 1 감소시킴
       if (L.ref ≠ 0) then return;
       p ← L;       // p는 순환 포인터
       while (p.link ≠ null) do {
        p ← p.link;
        if p.tag = 1 then removeList(p.data); // tag=1이면 서브리스트로서 순환
       }
       p.link ← Free;       // Free는 자유 공간 리스트
       Free ← L;
 end removeList()
```

쓰레기 수집

참조 계수 사용의 한계

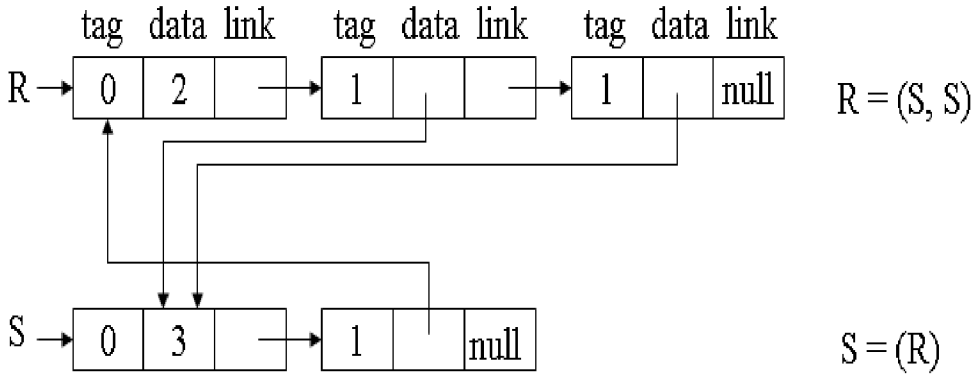
① 순환 리스트

- 반환되어야 될 리스트인데도 그 참조 계수가 결코 0이 되지 않음

예) removeList(C) : C의 참조 계수를 1로 만들지만 (0이 되지 않지만) 이 리스트는 다른 포인터나 리스트를 통해 접근이 불가능

② 간접 순환

예) removeList(R)와 removeList(S)가 실행된 뒤에 R.ref = 1, S.ref = 2가 되지만 더 이상 참조할 수 있는 것이 아님. 그러나 참조 계수가 0이 아니기 때문에 반환될 수 없음



쓰레기 (garbage)

① 실제로 사용하고 있지 않으면서도 자유 공간 리스트에 반환될 수 없는 메모리

② 시스템 내에 쓰레기가 많이 생기면 가용공간 리스트가 고갈되어 프로그램 실행이 중지되는 경우 발생

③ 자유 공간이 고갈되어 프로그램 실행을 더 이상 진행할 수 없는 경우 발생 가능

쓰레기 수집 (garbage collection)

① 사용하지 않는 모든 노드들을 수집하여 자유 공간 리스트에 반환시켜 시스템 운영 지속

② 모든 리스트 노드가 크기가 일정하다고 가정

③ 각 노드에 추가로 할당된 mark bit라는 특별한 비트는 0과 1만을 갖도록 하여 노드의 사용 여부(사용되지 않음 : 0, 사용 중 : 1)를 표현

쓰레기 수집 과정

① 초기화 단계 (initialization)
- 메모리에 있는 모든 노드의 마크 비트를 0으로 설정하여 사용하지 않는 것으로 표시

② 마크 단계 (marking phase)
- 현재 사용되고 있는 리스트 노드들을 식별해서 마크 비트를 1로 변경한 뒤 이 노드의 data 필드를 검사
- data 필드가 다른 리스트를 참조하면 이 필드에서부터 이 단계를 순환적으로 진행
- data 필드를 따라 처리를 끝낸 뒤에는 다시 link 필드를 따라 다음 노드를 처리 (이 때 mark bit가 1이면 그 노드를 통한 경로는 진행할 필요 없음)

③ 수집 단계 (collecting phase)
- 모든 노드의 mark bit를 검사해서 0으로 마크된 모든 노드들을 자유 공간 리스트에 반환

쓰레기 수집 알고리즘

```
garbageCollection()
            // ListNode는 markBit, tag, data, link로 구성
            // ListNode들은 listNodeArray라는 노드 배열로 할당되어 있다고 가정
            // listNodeArray의 크기는 listNodeArraySize로 가정

            // 초기화 단계 : 모든 ListNode들의 markBit을 0으로 설정
       for (i ← 0; i < listNodeArraySize; i ← i + 1) do
            listNodeArray[i].markBit ← 0;

            // 마크단계 : 사용중인 노드의 markBit을 모두 1로 표시
       for (i ← 0; i < numberofPointers; i ← i + 1) do
            markListNode(p[i]);      // p[i]는 사용중인 포인터 변수

           // 수집단계 : markBit이 0인 노드들을 수집, Free 리스트에 연결
       for (i ← 0; i < listNodeArraySize; i ← i + 1) do {
            if (listNodeArray[0].markBit = 0) then {
                 listNodeArray[i].link ← Free;
                 Free ← listNodeArray[i];
            }
       }
markListNode(p)      // p는 포인터 변수
       // 사용중인 노드를 마크, p는 포인터 변수
            if ((p ≠ null) and (p.markBit = 0)) then
                 p.markBit ← 1;
            if (p.tag = 1) then markListNode(p.data);
             // data 경로를 따라 markBit을 검사
            markListNode(p.link);      // link 경로를 따라 markBit을 검사
        end markListNode()
end garbageCollection()
```

일반 리스트를 위한 함수

리스트의 복사본 생성 함수

```
copyList(L)
        // L은 비순환 리스트로서 공용 서브 리스트가 없음
        // L과 똑같은 리스트 p를 만들어 그 포인터를 반환
          p ← null;
          if (L ≠ null) then {
               if L.tag = 0 then q ← L.data;       // 원자 값을 저장
               else q ← copyList(L.data);       // 순환 호출
               r ← copyList(L.link);       // tail(L)을 복사
               p ← getNode();       // 새로운 노드 생성
               p.data ← q;
               p.link ← r;    // head와 tail을 결합
               p.tag ← L.tag;
          }
          return p;
end copyList()
```

두 리스트의 동일성 검사 함수

- 구조가 같고 대응되는 필드의 데이타 값이 같아야 동일

```
equalList(S, T)
      // S와 T는 비순환 리스트, 각 노드는 tag, data, link 필드로 구성
      // S와 T가 똑같으면 true, 그렇지 않으면 false를 반환
      b ← false;
      case {
           S = null and T = null : b ← true;
           S ≠ null and T ≠ null : if S.tag = T.tag then {
                        if (S.tag = 0) then b ← (S.data = T.data);
                        else b ← equalList(S.data, T.data);
                        if (b) then b ← equalList(S.link, T.link);
                   }
      }
      return b;
end equalList()
```

리스트의 깊이 계산

```
depthList(L)
        // L는 비순환 리스트, 각 노드는 tag, data, link로 구성
        // 리스트 L의 깊이를 반환
    max ← 0;
    if (L = null) then return(max);       // 공백 리스트의 깊이는 0
    p ← L;
    while (p ≠ null) do {       // p는 순환 포인터
          if (p.tag = 0) then d ← 0;
          else d ← depthList(p.data);       // 순환
          if (d > max) then max ← d;       // 새로운 max
          p ← p.link;
    }
    return max+1;
end depthList()
```

C에서의 일반 리스트 구현

struct genListNode

- 노드에 어떤 타입의 포인터도 저장할 수 있도록 data 필드를 void*로 정의

genList

- 연결 리스트의 첫 번째 genListNode에 대한 참조를 저장할 head 필드를 포함한 헤더 노드 가짐
- 서브리스트는 하나의 genList 타입의 변수인 data로 표현

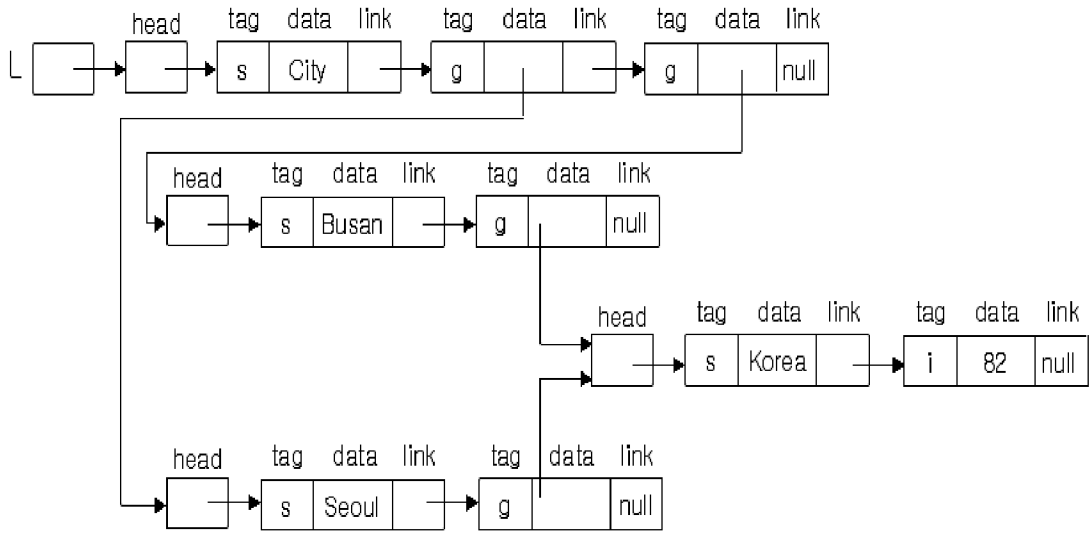
예제 프로그램 10.6

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct genListNode {
                    char tag;          /* data의 타입: s(스트링), g(일반 리스트)
*/
                    void* data;        /* 스트링이나 일반 리스트에 대한 포인터
*/
                    struct genListNode* link;
} genListNode ;
typedef struct {
          genListNode* head;          /* head는 genList의 헤더 노드로서 */
                                              /* 리스트의 첫 번째 노드에 대
한 포인터를 저장 */
} genList;
```

```
genList* createGenList(void);
void freeGenList(genList*);
void printGenList(genList*);
void insertData(genList*, char, void*);
genList* createGenList( void ) { /* 공백 리스트를 만듦 */
       genList * L;   L = (genList *)malloc(sizeof(genList));
       L->head = NULL;
   return L;
}
void freeGenList(genList* L) {
             genListNode* p;
            while(L->head != NULL) {
                        p = L->head;     L->head = L->head->link;
                       free(p);              p = NULL;
             }
}
void insertData(genList* L, char type, void* p) {
                              /* 리스트 L에 새로운 genListNode를 삽입 */
         genListNode* newNode;
         newNode = (genListNode*)malloc(sizeof(genListNode));
         newNode->tag = type;
         newNode->data = p;
         newNode->link = L->head;
         L->head = newNode;
}

void printGenList(genList* L) {                    /* 일반 리스트 L을 출력 */

         genListNode* p;
         p = L->head;
        printf("(");
        while (p != NULL) {     /* 공백 리스트가 아닌 경우 */
                 switch (p->tag) {  /* tag 값으로 data의 type을 검사 */
                          case 'g' :          /* 일반 리스트인 경우 */
                      printGenList((genList*) p->data);
                                        /* 서브리스트를 순환적으로 출력 */
                                  break;
                          case 's' :      /* 스트링인 경우 */
                                  printf("%s", p->data);
                                  break;
                          default :
                                  printf("Invalid type. \n");
                                        /* 불법 데이타 타입인 경우 */
                                  exit(1);
                  }
                 if ((p = p->link) != NULL) printf(", ");
         }
        printf(")");
}
```

공용 서브 리스트를 가진 일반리스트

L=(City,(Seoul,(Korea,82)),(Busan,(Korea,82)))



프로그램 10.7 (일반 리스트 생성 및 출력)

```
/* 프로그램 10.6을 여기에 삽입 */
int main() {
        genList *p, *q, *r, *L;
        p = createGenList();
        insertData(p, 's', "82");       insertData(p, 's', "Korea");
        q = createGenList();
        insertData(q, 'g', p);          insertData(q, 's', "Seoul");
        r = createGenList();
        insertData(r, 'g', p);insertData(r, 's', "Busan");
        L = createGenList();
        insertData(L, 'g', r);insertData(L, 'g', q); insertData(L, 's', "City");
        printGenList(L);            printf("\n");
        freeGenList(p);    /* 메모리 반환 */
        freeGenList(q);         freeGenList(r);freeGenList(L);
        printGenList(L);
      printf("\n");
      return 0;
}
```