8. 자 료 구 조

데이타의 표현과 이에 적합한 알고리즘의 선정이 중요

효율적이고 정확한 프로그램의 작성을 좌우

ex) 순차 탐색법과 이진 탐색법 : 일반적으로 이진 탐색법이 효율적, 하지만 이진 탐색법은 데이타가 값 크기에 따라 정렬되어 있는 경우에만 적용 가능

추상화(abstraction)

자세한 것은 무시하고 필수적이고 중요한 속성만을 골라서 단순화시키는 과정, 세세한 부분은 가리고 외부에 필요한 부분만 보여주는 것

데이타 추상화(data abstraction)

프로그램 속의 복잡한 데이타에 추상화 개념을 적용, 기존의 잘 정의된 개념들을 이용하여 표현하는 것이 보통

데이타

프로그램의 처리 대상이 되는 모든 것, 특별히 값(value) 자체를 의미하기도 함

데이타 타입

데이타의 집합과 이 데이타에 적용할 수 있는 연산의 집합

-종류

① 시스템 정의(system-defined) 데이타 타입

② 원시(primitive) 데이타 타입 또는 단순(simple) 데이타 타입

③ 복합(composite) 데이타 타입 또는 구조화(structured) 데이타 타입

사용자 정의 (user-defined) 데이타 타입

기존의 데이타 타입을 이용해 정의

일단 정의만 되면 시스템 정의 데이타 타입과 똑같이 사용할 수 있음

ex) integer 데이타 타입

데이타 : 정수 (... 2, -1, 0, 1, 2 ...)

연산자 : +, -, *, /, mod

추상 데이타 타입(abstract data type: ADT)

데이타 타입의 논리적 정의

데이타와 연산의 본질에 대한 명세만 정의한 데이타 타입

데이타가 무엇이고, 각 연산은 무슨 기능을 수행하는가 만을 정의

데이타의 구조, 연산의 구현 방법은 포함시키지 않음

기존의 데이타 타입을 이용하여 정의

데이터 추상화와 추상데이터 타입(ADT)의 차이점

Data Abstraction : " 우리가 데이터에 어떤식으로 접근해야지 나중에 프로그램을 수정할때 덜 힘들것인가", " 이미 만들어진 것을 다시 사용하면 참 편하지 않을까." 를 고민하다가 생겨난 개념

ADT(Abstract Data Type) : 모든 언어에서 int, char, byte 등의 기본적인 자료형이 제공된다. int라는 data type은 표현할 수 있는 최대, 최소값이 한정되어있고 어떤값이 표현할 수 있는 범위를 넘어설 때 어떻게 나타낼 것인가 등에 대한 규칙이 내포되어있다. 또한 같은 int 끼리 값을 더하거나 뺄때 결과는 어떤식으로 도출되어야 하며 서로 다른 자료형에 대해서는 어떻게 처리할 것인지가 정해져있다. 이런 일련의 규칙들을 통해서 하나의 data type 이 정해진다. 결국 data type은 객체와 밀접하게 관련된 operation들과 그 객체를 하나로 묶어놓는 개념이다. ATD는 Data Abstraction의 개념을 적용한 data type이다. 가장 중요한 차이는 데이터의 명세(specification)와 이 데이터들을 건드리는 operation의 명세(secification) 가 실제 데이터의 구현과 operation의 구현과 분리되어 있다.

추상화와 구체화의 관계

	데이터	연산
추상화	추상 데이터 타입	알고리즘
구체화	데이터 타입	프로그램

명세와 구현의 차이

“두 수의 합을 구하는 함수" 의 명세(specification)
int add(int , int); void add(int *, int *, int *);

int 형의 두 수를 더해서 그 결과를 되돌려준다는 뜻이 명세에 나타나있다. 첫번째는 값을 넘겨받아서 결과값을 되돌려주는 방식이고 두번째는 포인터로 접근하는 방식이다.

“두 수의 합을 구하는 함수" 의 실제 구현(implementation)

int add(int a, int b)
{
return (a+b);
}
void add(int *a, int *b, int *result)
{
*result = *a + *b;
}

명세는 "겉포장(사용자 제품 설명서)" 이고 구현은 "내용물(실제 제품)" 이다

- 겉포장과 내용물이 완전히 분리된 data type을 ADT라고 말한다
- 명세와 구현이 완전히 분리된 data type을 ADT라고 말한다

  객체지향 프로그래밍에서는 class 가 바로 ADT에 해당한다.

자연수(Natno) 추상 데이타 타입

```
ADT Natno
        데이타 : { i | i  integer, i ≥ 0}
        연산 :  for all x, y  Natno
                zero() ::= return 0;
                isZero(x) ::= if x then return false
                        else return true;
                succ(x) ::= return x + 1;
                add(x, y) ::= return x + y;
                subtract(x, y) ::= if x < y then return 0
                        else return x - y;
                equal(x, y) ::= if x = y then return true
                        else return false;
        End Natno
```

알고리즘과 문제 해결

알고리즘(algorithm) : 특정 문제를 해결하기 위해 기술한 일련의 명령문

프로그램(program) : 알고리즘을 컴퓨터가 이해하고 실행할 수 있는 특정 프로그래밍 언어로 표현한 것

program = algorithm + data structures

알고리즘의 요건

완전성과 명확성 : 수행 단계와 순서가 완전하고 명확하게 명세되어야 함, 순수하게 알고리즘이 지시하는 대로 실행하기만 하면 의도한 결과가 얻어져야 함

입력과 출력

입력 : 알고리즘이 처리해야 할 대상으로 제공되는 데이타

출력 : 입력 데이타를 처리하여 얻은 결과

유한성 : 유한한 단계 뒤에는 반드시 종료

순환(recursion)

정의하려는 개념 자체를 정의 속에 포함하여 이용

직접 순환 : 함수가 직접 자신을 호출

간접 순환 : 다른 제 3 의 함수를 호출하고 그 함수가 다시 자신을 호출

순환 방식의 적용

분할 정복(divide and conquer) 의 특성을 가진 문제에 적합, 어떤 복잡한 문제를 직접 간단하게 풀 수 있는 작은 문제로 분할하여 해결하려는 방법, 분할한 문제는 원래의 문제와 그 성질이 같기 때문에 푸는 방법도 동일

순환 함수의 골격

```
if (simplest case) then solve directly
else (make a recursive call to a simpler case);
```

순환 함수의 예 (1) -Factorial (n!)
알고리즘 : n >= 1 : n * (n-1) * ・・・ * 2 * 1 = n * (n - 1)!

factorial(n)
// n 은 음이 아닌 정수
if (n <= 1) then return 1
else return (n * factorial(n-1));
end factorial()

비순환 함수로도 표현 가능 : 표현이 좀 길지만 제어의 흐름이나 실행 과정을 쉽게 이해할 수 있음

순환 함수의 예 (2) -이진탐색
알고리즘 : key = a[mid]면 탐색성공 key < a[mid] : a[mid]의 왼편에 대한 이진검색 key > a[mid] : a[mid]의 오른편에 대한 이진검색

```
binsearch(a[], key, left, right)
        if (left ≠ right) then {
                    mid = (lift + right) / 2;
                       case {
                            key = a[mid] : return(mid);
                            key < a[mid] : return(binsearch(a, key, left, mid-1));
                            key > a[mid] : return(binsearch(a, key, mid+1, right));
                       }
              }
                   else return -1;   // key 값이 존재하지 않음
            end binsearch()
```

순환 함수의 예 (3) - 피보나치 수열
알고리즘 : n = 0 : f_0, n = 1 : f_1, n = 2 : $f_n = f_n-1 + f_n-2$

```
fib(n)
            if (n = 0) then return 0;
            else if (n=1) then return 1
            else return (fib(n-1)  + fib(n-2));
end fib()
```

이 fib() 함수는 순환 호출이 증가하여 실행시간으로 볼 때 반복적 함수보다 비효율적임, 순환적 정의가 순환적 알고리즘으로 문제를 해결하는데 최적의 방법이 아닐 수도 있다는 예

성능 분석

프로그램의 평가 기준

① 원하는 결과의 생성 여부

② 시스템 명세에 따른 올바른 실행 여부

③ 프로그램의 성능

④ 사용법과 작동법에 대한 설명 여부

⑤ 유지 보수의 용이성

⑥ 프로그램의 판독 용이

프로그램의 성능 평가

성능 분석(performance analysis) : 프로그램을 실행하는데 필요한 시간과 공간의 추정

성능 측정(performance measurement) : 컴퓨터가 실제로 프로그램을 실행하는데 걸리는 시간 측정

공간 복잡도(space complexity) : 프로그램을 실행시켜 완료하는데 소요되는 총 저장 공간

- Sp = Sc + Se

  Sc : 고정 공간 (명령어 공간, 단순 변수, 복합 데이타 구조와 변수, 상수)

  Se : 가변 공간 (크기가 변하는 데이타 구조와 변수들이 필요로 하는 저장 공간)
  런타임 스택(runtime stack)을 위한 저장 공간

시간 복잡도(time complexity) : 프로그램을 실행시켜 완료하는데 걸리는 시간

- Tp = Tc + Te

  Tc : 컴파일 시간

  Te : 실행 시간 (단위 명령문 하나를 실행하는데 걸리는 시간, 실행 빈도수)

실행 환경

프로그램의 성능 : 프로그램의 실행 시간은 실행 환경에 따라 다르기 때문에 이들을 단순 비교하는 것은 불공정함

실행 환경의 구분

최선의 경우(best case) : 알고리즘의 시간 복잡도가 가장 작게 되는 경우

최악의 경우(worst case) : 알고리즘의 시간 복잡도가 가장 크게 되는 경우

※ 공정한 비교를 위해 두 경우를 모두 계산해 보아야 함

시간 복잡도

알고리즘을 구성하는 명령어들이 몇 번이나 실행이 되는지를 센 결과(frequency count)에 각 명령어의 실행시간(execution time)을 곱한 합계를 의미한다. 그러나 각 명령어의 실행시간은 특정 하드웨어 혹은 프로그래밍 언어에 따라서 그 값이 달라질 수 있기 때문에 알고리즘의 일반적인 시간 복잡도는 명령어의 실제 실행시간을 제외한 명령어의 실행 횟수만을 고려하게 된다."

n번째 항을 계산하는 반복식 프로그램

```
fib_i(n)
        1-2         if (n < 0) then stop;    // error 발생
        3-4         if (n ≤ 1) then return n;
        5-6         fn2 ← 0; fn1← 1;
          7          for (i←2; i<=n; i←i+1) do {
          8                fn ← fn1 + fn2;
          9                fn2 ← fn1;
         10                fn1 ← fn;
         11          }
         12          return fn;
         13      end fib_i()
```

fn 계산을 위한 실행 빈도수

명령문(행)	실행 빈도수	명령문(행)	실행 빈도수
1	1	8	**n-1**
2	0	9	**n-1**
3	1	10	**n-1**
4	0	11	0
5	1	12	1
6	1	13	0
7	**n**		

실행 시간

4n+2 : O(n)

함수 fib_i()의 시간 복잡도는 O(n)이다 라고 말함

Big-Oh (O) : upper bound (상한값), 이보다는 늦지 않는다는 의미

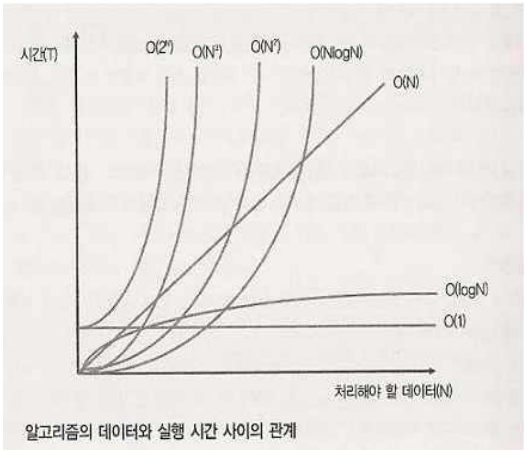
n≥n1(n≥제로) 인 모든 정수 n에 대하여 f(n) ≤ c · g(n1) 를 만족하는 두 정수 c와 n1가 존재할 때, f(n) = O(g(n)) 이라고 정의한다. 그리고 f가 g의 big-oh(빅오)와 같다고 한다

예 :

① f(n) = 3n + 2 : f(n) = O(n)
② f(n) = $1000n^2$ + 100n - 6 : f(n) = $O(n^2)$
③ f(n) = $6 \cdot 2^n + n^2$: f(n) = $O(2^n)$
④ f(n) = 100 : f(n) = O(1)

연산 시간 그룹

① 상수시간 : O(1)
② 로그시간 : O(logn)
③ 선형시간 : O(n)
④ n로그시간 : O(nlogn)
⑤ 평방시간 : $O(n^2)$
⑥ 입방시간 : $O(n^3)$
⑦ 지수시간 : $O(2^n)$
⑧ 계승시간 : O(n!)



알고리즘의 데이터와 실행 시간 사이의 관계

연산 시간의 크기 순서

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

$O(n^k)$: 다항식 시간(polynomial time)

Ω(Omega) : lower bound (하한값) -> 적어도 g(n)이상의 시간을 필요로 한다는 의미

n≥n1인 모든 정수 n에 대하여 f(n) ≥ c · g(n) 를 만족하는 두 정수 c와 n1가 존재할 때, f(n) = Ω(g(n)) 이라고 정의. 그리고 f가 g의 Omega(오메가)와 같다고 한다.

ⓗ(Theta) : upper / lower bound -> 상한인 동시에 하한이므로 보다 더 정확한 한계치

n≥n1인 모든 정수 n에 대하여 $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ 을 만족하는 세 정수 c_1 , c_2 , n1가 존재할 때, f(n) = ⓗ(g(n)) 이라고 정의한다.

※ 시간복잡도가 낮은 알고리즘만이 가장 우수하다고는 할 수 없다. 데이터가 100개 정도라면 n^3까지도 10초정도(계수가 낮을 때) 내에 처리할 수도 있다. 작성하기도 이해하기도 쉽고, 작고, 간단하다면 시간복잡도가 낮은 알고리즘도 효용성이 있다.

9. 연결표현 (Linked List)

순차 표현

장점 : 표현이 간단함

원소의 접근이 빠름(인덱스는 직접 메모리 주소로 변환할 수 있기 때문에 빠른 임의 접근이 가능)

단점 : 원소의 삽입과 삭제가 어렵고 시간이 많이 걸림 (원소의 삽입, 삭제시 해당 원소의 위치 뒤에 있는 모든 원소를 뒤로 물리거나 앞으로 당겨야만 함)

저장공간의 낭비와 비효율성

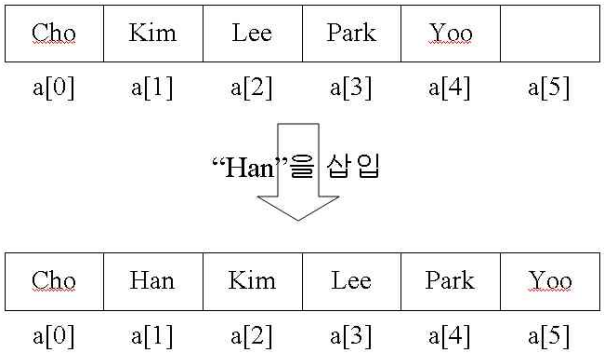
1) 리스트의 길이가 임의로 늘어나고 줄어드는 상황에서 배열의 적정 크기를 미리 결정하기가 어려움

2) 배열의 오버플로우(overflow)나 과도한 메모리 할당 발생

순서 리스트의 순차 표현

리스트 L의 순차 표현과 원소 삽입

L=(Cho, Kim, Lee, Park, Yoo) : 알파벳 순서로 된 성씨의 리스트



연결 표현

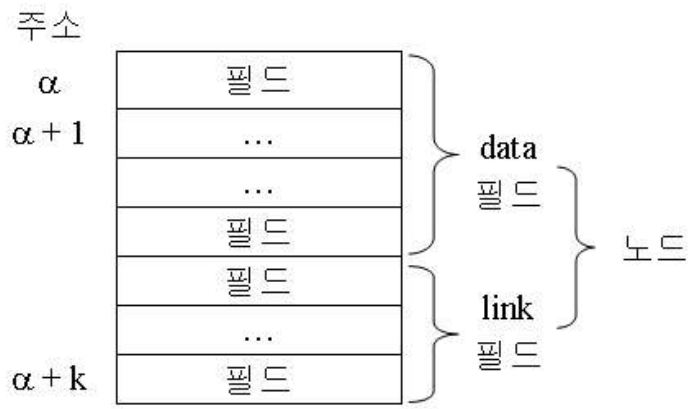
연결 표현(linked representation) 또는 비순차 표현

- 원소의 물리적 순서가 리스트의 논리적 순서와 일치할 필요 없음
- 원소를 저장할 때 이 원소의 다음 원소에 대한 주소도 함께 저장해야 함
- 노드 : < 원소, 주소> 쌍의 저장 구조

노드(node) : 데이타 필드와 링크 필드로 구성

- 데이타 필드 : 리스트의 원소, 즉 데이타 값을 저장하는 곳
- 링크 필드 : 다른 노드의 주소값을 저장하는 장소 (포인터)

물리적 노드 구조



논리적 노드 구조

연결 리스트 (linked list)

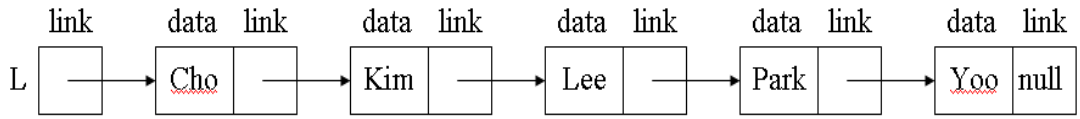
연결 리스트

링크를 이용해 표현한 리스트

예)

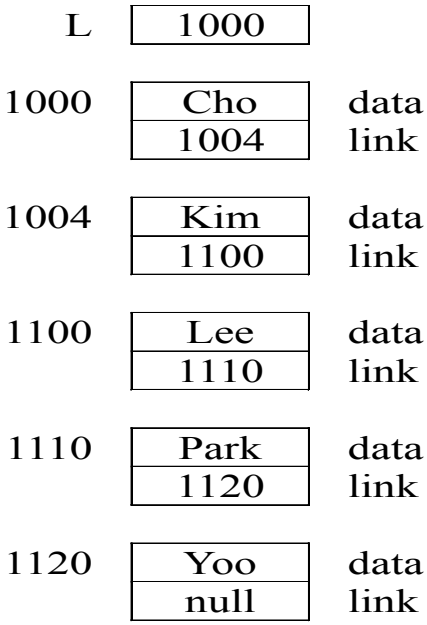
리스트 L : 리스트 전체를 가리킴

노드 L : 리스트의 첫번째 노드(Cho) 를 가리킴



1. 링크에 실제로는 화살표 대신 주소값이 들어가 있음

예) 원소 Cho , Kim , Lee , Park , Yoo 가 저장된 노드들의 주소가 각각 1000, 1004, 1100, 1110, 1120 일 때



2. 연결 리스트의 마지막 노드의 링크 값으로 null을 저장

3. 연결 리스트 전체는 포인터 변수 L 로 나타냄. 여기에는 리스트의 첫 번째 노드의 주소가 저장됨
   - 공백 리스트의 경우 L 의 값은 null 이 됨. 이렇게 null 값을 가진 포인터를 널 포인터라 함

4. 노드의 필드 선택은 점 연산자(. : dot operator)를 이용

예) L.data : L 이 가리키는 노드의 data 필드값, Cho
L.link.data : L이 가리키는 노드의 link 값이 가리키는 노드의 data 값, Kim

5. 포인터 변수에 대한 연산은 제한되어 있음
   ① p = null 또는 p ≠ null : 공백 포인터인가 또는 공백 포인터가 아닌가의 검사
   ② p = q : 포인터 p 와 q 가 모두 같은 노드 주소를 가리키는가?
   ③ p ← null : p 의 포인터 값으로 널을 지정
   ④ p ← q : q 가 가리키는 노드를 p 도 똑같이 가리키도록 지정
   ⑤ p.link ← q : p가 가리키는 노드의 링크 필드에 q의 포인터 값을 지정
   ⑥ p ← q.link : q가 가리키는 노드의 링크 값을 포인터 p의 포인터 값으로 지정

C 언어에서의 포인터

포인터 : 다른 어떤 변수(variable)의 주소(address)를 그의 값으로 저장하는 변수
C 언어에서는 모든 변수가 주소를 가지고 있기 때문에 모든 타입에 대해서 포인터 타입이 존재한다.

포인터 선언

데이타 타입 이름 뒤나 변수 이름(식별자) 앞에 * 를 붙임

예) char 타입의 변수와 포인터 선언

```
char c =  k  ;
char *pc;   /*  또는 char* pc; */
```

포인터의 초기화

① 포인터를 선언만 하고 초기화하지 않으면 값은 미정(undefined)이 됨
② 포인터를 초기화하기 위해서는 변수에 주소를 지정해야 함
③ 변수의 주소는 주소 연산자 &를 변수 앞에 붙이면 됨

```
char c =  k  ;
char *pc;   /*  또는 char* pc; */
pc = &c;
```

이렇게 되면 pc에는 char 타입 변수 c의 주소가 저장됨

값 참조

① 포인터가 지시하는 주소의 값(내용)을 참조하기 위해서는 포인터 역 참조(dereference)를 해야 함
② 즉, 포인터 pc에 저장되어 있는 주소를 참조해 그 주소를 따라 가서 거기에 저장되어 있는 내용(값)을 검색
③ 이 역 참조는 포인터 변수 앞에 *를 붙여 표현함
④ 예)

```
char c = ' k ' ;
char *pc;   /*  또는 char* pc; */
pc = &c;
```

와 같이 초기화되었으면

```
printf ("c is %c\n", c);
printf ("c is %c\n", *pc);
```

이 두 명령문은 같은 값을 프린트함

값의 변경

c 의 값을 y 로 변경할 때는

1) c 에 바로 y 를 지정하거나

2) 포인터 pc를 이용해서 간접적으로 역 참조를 통해 변경할 수 있음

```
pc = &c;        /* pc에 변수 c의 주소를 지정 */
c = 'y';        /* c에  y'를 지정 */
*pc = 'y';      /* pc가 지시하고 있는 주소에  y'를 지정 */
```

배열에 대한 포인터

배열 원소의 타입에 일치하는 포인터 변수를 선언하여 사용

예) 문자 배열과 문자 타입 포인터의 선언

char letters[10], *pc;

문자 배열의 처음 두 원소에 값을 지정하는 방법

① 배열을 이용하는 방법

letters[0] = a ;

letters[1] = b ;

② 포인터를 이용하는 방법

포인터 pc 가 배열 letters[] 의 시작을 지시하게끔 초기화

pc = &letters[0]; 또는 pc = letters;

배열에 값을 지정

*pc = a ; *(pc+1) = b ;

또는 *pc++ = a ; *pc++ = b ;

포인터에 대한 ++ 연산자

① 포인터는 현재의 원소 다음에 있는 원소를 가리키게 됨 : 포인터 값이 증가되면 배열의 다음 원소를 지시하게 됨

② 포인터 pc가 배열 letters[]의 첫 번째 원소, letters[0]을 가리키고 있으면pc++ 는 pc가 배열 letters[]의 두 번째 원소, letters[1]을 가리키도록 이동시킴 : 위 예에서는 포인터가 char타입이기 때문에 포인터 값이 증가되면 포인터는 다음 바이트(byte) 즉, 다음 문자를 가리키게 됨

포인터 배열

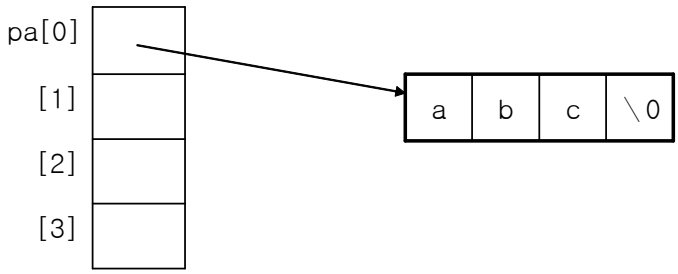
① C 에서는 어떤 타입의 변수 배열도 정의 가능

② 포인터도 다른 변수의 주소를 저장하는 변수이기 때문에 포인터 배열 (array of pointers)을 정의할 수 있음

③ 이 경우에 배열의 각 원소는 포인터가 됨

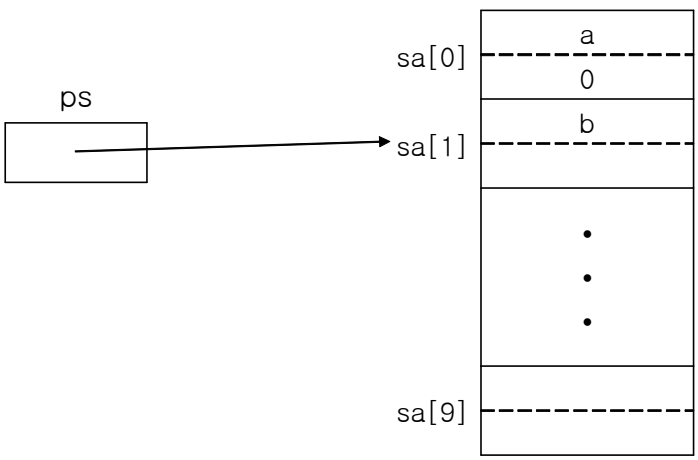
④ 예)

```
char *pa[4];
pa[0] = ‘ abc ’ ;
```



struct 배열

```
struct frequency {
        char letter;
        int count;
};
struct frequency sa[10], *ps;      /* struct 배열과 포인터 */
ps = sa;          /* ps에 배열의 시작 주소, sa[0]를 지정 */
ps->letter = 'a';        ps->count = 0;   /* 첫 번째 원소의 문자와 정수 */
ps++;                    /* ps가 sa[1]을 가리키도록 증가 */
ps->letter = 'b';      /* 두 번째 원소의 문자 */
```



자체 참조 구조(self-referential structure)

struct 의 멤버는 같은 타입의 또 다른 struct 를 지시하는 포인터도 될 수 있음

예) 리스트의 노드를 정의하는데 유용함

```
struct char_list_node {
                        char letter;
                        struct char_list_node    *next;
                    };
                  struct char_list_node    *p;
```

① struct 타입의 char_list_node 를 선언하는 동시에 p 를 이 struct char_list_node 타입에 대한 포인터로 선언

② char_list_node 타입의 노드로 구성된 연결 리스트를 형성할 수 있게 하고 포인터 p 는 이 리스트의 노드를 지시할 수 있게 함

③ p = p->next; 와 같이 포인터를 순회시킴으로써, 이 포인터 p 가 가리키는 노드에 연결된 다음 노드 접근

typedef 키워드

리스트 처리를 위해 노드와 포인터를 정의할 때 typedef를 사용하면 더욱 간결해짐

```
typedef struct char_list_node    *list_pointer;
char_list_node{
                    char letter;
                    list_pointer next;
};

          list_pointer    p = NULL;
```

- 포인터 타입 list_pointer 는 아직 정의되지 않은 char_list_node 라는 struct 타입을 이용하여 정의
- 포인터 p 는 NULL 값으로 초기화

메모리의 동적 할당

프로그램을 작성할 당시에는 얼마나 많은 공간이 필요한지 알 수 없고, 필요 없는 낭비되거나 필요한 공간이 모자라는 상황을 막기 위해 메모리를 동적으로 할당할 필요가 있음

① malloc() : 컴파일 시간에 확정된 크기의 메모리를 할당하지 않고 필요한 때에 필요한 만큼의 공간을 동적으로 운영체제에 요구하게 됨. 사용가능한 기억장소가 있으면 요구한 크기의 메모리 영역에 대한 시작 주소를 포인터에 반환

② free() : malloc()으로 할당한 메모리 영역을 시스템에 반환

예제 프로그램

```
int i, *pi;
float f, *pf;
                    pi = (int *)malloc(sizeof(int));
                    pf = (float *)malloc(sizeof(float));
                    *pi = 10;
                    *pf = 3.14;
                    printf("an integer = %d, a float = %f\n",*pi, *pf);
                    free(pi);
                    free(pf);
```

① sizeof는 필요한 기억 장소의 크기에 대한 정보 제공

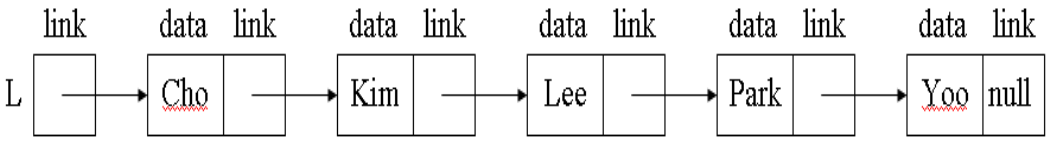
② malloc() 함수의 반환 값은 요청한 크기의 메모리 영역에 대한 첫 번째 주소가 되기 때문에, 요청한 데이타 타입과 일치하도록 타입 캐스트(type cast)를 하고 포인터 변수에 지정

단순 연결 리스트 (singly linked list)

① 하나의 링크 필드를 가진 노드들이 모두 자기 후속노드와 연결되어 있는 노드열

② 마지막 노드의 링크 필드는 리스트의 끝을 표시하는 null 값을 가짐

③ 별칭 : 선형 연결 리스트(linear linked list), 단순 연결 선형 리스트(singly linked linear list), 연결 리스트(linked list), 체인(chain)



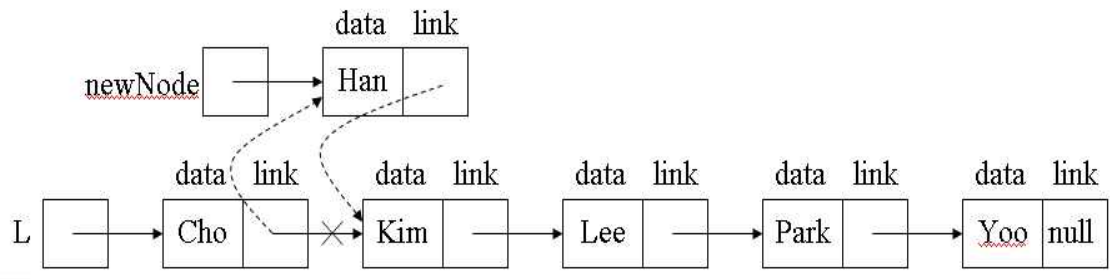
④ 단순 연결 리스트의 예

원소 삽입 알고리즘

① 예) 리스트 L에 원소 Han 을 Cho 와 Kim 사이에 삽입

1. 공백노드를 획득함. newNode 라는 변수가 가리키게 함
2. newNode의 data 필드에 Han을 저장
3. Cho 를 저장하고 있는 노드의 link 값을 newNode의 link 필드에 저장 (즉 Kim을 저장하고 있는 노드의 주소를 newNode의 link에 저장)
4. Cho 를 저장한 노드의 link 에 newNode의 포인터 값을 저장

② 리스트의 기존 원소들을 이동시킬 필요 없음

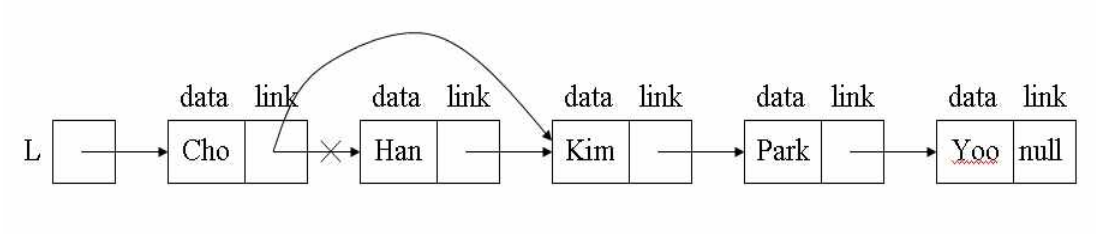


③ 부수적인 link 필드를 위해 저장 공간을 추가로 사용

원소 삭제 알고리즘

예) 리스트 L 에서 원소 Han 을 삭제

1. 원소 Han 이 들어 있는 노드의 선행자 찾음 (Cho가 들어있는 노드)
2. 이 선행자의 link에 Han 이 들어있는 노드의 link 값을 저장



메모리의 획득과 반납

연결 리스트가 필요로 하는 두 가지 연산

① 데이타 필드와 하나의 링크 필드를 가진 하나의 공백 노드를 획득하는 방법

② 사용하지 않는 노드는 다시 반납하여 재사용하는 방법

자유 공간 리스트 (free space list)가 있는 경우

getNode() : data와 link 필드로 되어 있는 새로운 공백 노드를 가용 공간 리스트로부터 할당받아 그 주소를 반환

returnNode() :

① 포인터 변수 p가 지시하는 노드를 자유 공간 리스트에 반환

② 프로그래밍 언어에 따라 필요한 경우도 있고 필요하지 않은 경우도 있음

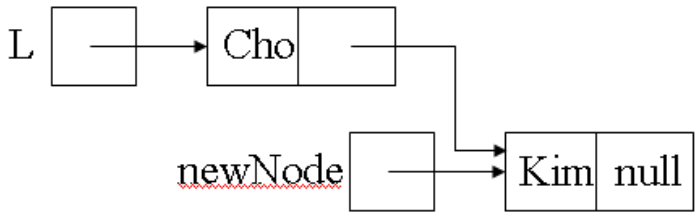
③ C와 같이 쓰레기 수집 시스템(garbage collection system)이 없는 언어에서는 사용자가 직접 일정 메모리 공간을 관리해야 함

앞의 두 함수를 이용한 리스트 생성 알고리즘

```
newNode ← getNode();                // 첫번째 공백 노드를 할당받아
   // 임시 포인터 newNode가 가리키도록 함 newNode.data ←   Cho   ;
   // 원소 값을 저장
L ← newNode;                        // 리스트 L을 생성
L.data ←   Cho   ;
newNode ← getNode();                // 두 번째 공백 노드를 획득
newNode.data ← 'Kim';               // 두번째 노드에 원소 값을 저장
newNode.link ← null;                // 포인터 값 null을 저장
L.link ← newNode;                   // 두 번째 노드를 리스트 L에 연결
```

리스트 생성 알고리즘을 점 표기식으로 작성한 경우

```
L ← getNode();
L.data ←    Cho   ;
L.link ← getNode();
L.link.data ←    Kim   ;
L.link.link ← null;
```



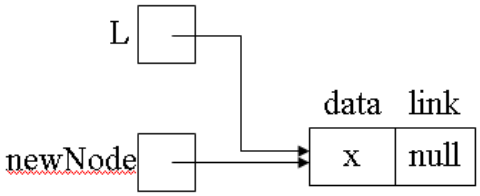
리스트 L의 첫번째 노드로 data 값이 x인 노드를 삽입

```
addFirstNode(L, x)
                newNode ← getNode();
                newNode.data ← x;
                newNode.link ← L;
                L ← newNode;
end addFirstNode()
```

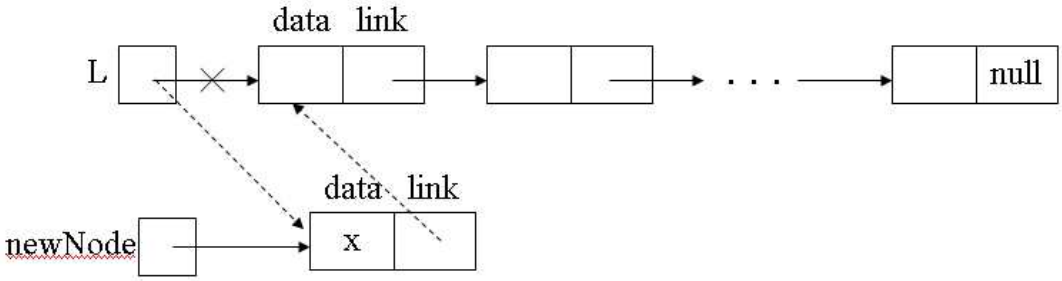
원소값이 x인 노드를 p가 가리키는 노드 다음에 삽입

```
insertNode(L, p, x)
                // 리스트 L에서 p 노드 다음에 원소값 x를 삽입
                newNode ← getNode();    // 공백 노드를 newNode가 지시
                newNode.data ← x;               // 원소 값 x를 저장
                if (L=null) then {          // L이 공백 리스트인 경우
                        L ← newNode;
                        newNode.link ← null;
                }
                else if (p=null) then {         / p가 공백이면 L의 첫 번째 노드로 삽입
                        newNode.link ← L;
                        L ← newNode;
                }
                else {                          // p가 가리키는 노드의 다음 노드로 삽입
                        newNode.link ← p.link;
                        p.link ← newNode;
                }
end insertNode()
```

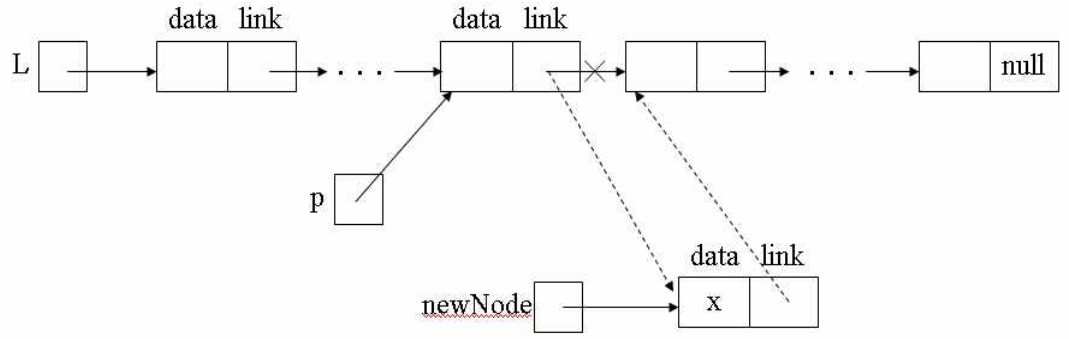
(a) L이 공백 리스트인 경우



(b) p가 null인 경우



(c) L과 p가 null이 아닌 경우



리스트 L의 마지막 노드로 원소값 x를 첨가

```
addLastNode(L, x)
            // 리스트 L의 끝에 원소 x를 삽입
        newNode ← getNode();          // 새로운 노드 생성
        newNode.data ← x;
        newNode.link = null;
        if (L = null) then {
            L ← newNode;
            return;
        }
        p ← L;                        // p는 임시 순회 포인터 변수
        while (p.link ≠ null) do      // 리스트의 마지막 노드를 찾음
            p ← p.link;
        p.link ← newNode;             // 마지막 노드로 첨가
end addLastNode()
```

리스트 L에서 p가 가리키는 노드의 다음 노드를 삭제

```
deleteNext(L, p)
            // p가 가리키는 노드의 다음 노드를 삭제
        if (L = null) then error;
        if (p = null) then {
            q ← L;                    // q는 삭제할 노드
            L ← L.link;               // 첫 번째 노드 삭제
        }
        else {
            q ← p.link;               // q는 삭제할 노드
            if (q = null) then return;  // 삭제할 노드가 없는 경우
            p.link ← q.link;
        }
        returnNode(q);                // 삭제한 노드를 자유 공간 리스트에 반환
end deleteNext()
```

리스트를 역순으로 변환하는 알고리즘(리스트 역순 변환)

```
reverse(L)
        // L = (e1, e2, ·, en)을 L = (en, en-1, ·, e1)으로 변환
            // 순회 포인터로 p, q, r을 사용한다.
    p ← L;          // p는 역순으로 변환될 리스트
    q ← null;       // q는 역순으로 변환될 노드
    while (p ≠ null) do {
        r ← q;          // r은 역순으로 변환된 리스트
                        // r은 q, q는 p를 차례로 따라간다.
        q ← p;
        p ← p.link;
        q.link ← r;     // q의 링크 방향을 바꾼다.
    }
    L ← q;
end reverse()
```

두개의 리스트 L1과 L2를 하나로 만드는 알고리즘(리스트 연결)

```
addList(L1, L2)
    // L1 = (a1, a2, ·, an), L2 = (b1, b2, ·, bm) 일 때,
    // L = (a1, a2, ·, an, b1, b2, ·, bm) 을 생성
    case {
        L1 = null:    return L2;
        L2 = null:    return L1;
        else:    p ← L1;    // p는 순회 포인터
                 while (p.link ≠ null) do
                     p ← p.link;
                 p.link ← L2;
                 return L1;
    }
end addList()
```

원소 값이 x인 노드를 찾는 알고리즘 (원소값 탐색)

```
searchNode(L, x)
        p ← L;
        while (p ≠ null) do {
            if (p.data = x ) then return p;
                                        // 원소 값이 x인 노드를 발견
            p ←  p.link;
        }
        return p;  // 원소 값이 x인 노드가 없는 경우 null 반환
end searchNode()
```

정수 리스트에 대한 연결 표현을 위한 노드 구조

```
typedef struct listNode {
            int    data;
            struct listNode* link;
       } listNode;
       listNode*    L = NULL;
```

원소 값이 x인 노드를 찾는 알고리즘의 C 구현

```
listNode* searchNode(listNode* L, int x) {
        listNode* p;            /* listNode를 가리키는 포인터 변수 */
        p = L;          /* p는 리스트 L의 첫 번째 노드를 지시 */
        while (p != NULL) {
            if (x == p->data))          /* p의 data 필드 값을 x와 비교 */
                return p;
            p= p->link;
        }
        return p; /* x 값을 가진 노드가 없는 경우 NULL을 반환 */
}
```

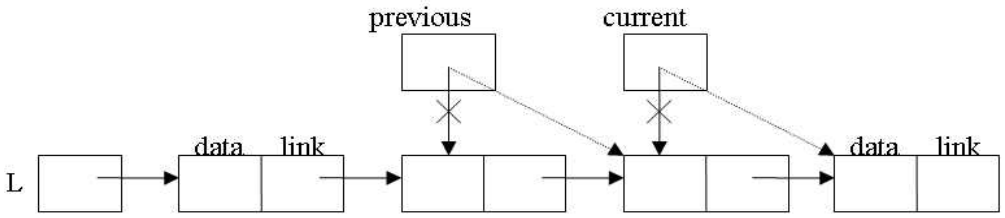
리스트에서 마지막 노드를 삭제하는 프로그램

```
void deleteLastNode(listNode* L) {
        listNode* previous;       /* listNode에 대한 포인터 */
        listNode* current;
                if (L == NULL) return;
                        L이 공백 리스트인 경우에 아무 것도 하지 않음 */
                if (L->link == NULL){
                        free(L);
                        L = NULL;          /* 노드가 하나만 있는 경우 */
                 }
                else {
                    previous=L;  /* 초기에 previous는 첫 번째 노드를 지시 */
                    current=L->link; // 초기에 current는 두 번째 노드를 지시

                while (current->link != NULL) {
                                /* current가 마지막 노드에 도달할 때까지 이동 */
                    previous = current;
                    current = current->link;
                }
                free(current);
                previous->link = NULL;
            // previous가 지시하는 마지막 두 번째 노드를 마지막 노드로 만듦
        }
}
```

current와 previous의 동작 과정

current 포인터가 어떤 노드를 가리키면 previous 포인터는 current가 가리키는 노드의 바로 직전 노드를 가리키도록 함

current가 리스트의 마지막 노드를 가리키게 될 때 previous는 마지막 두 번째 노드를 가리키게 됨



리스트 출력 (연결 리스트의 프린트)

```
void printList(listNode* L) {
        listNode* p;
        printf("(");       /* 제일 먼저   (   를 프린트 */
        p = L;             /* 순회 포인터로 사용 */
        while (p != NULL) {
                printf("%s", p->data);          /* data 필드 값을 프린트 */
                p = p->link;                    /* 다음 노드로 이동하여 */
                if (p != NULL) {          /* 공백인가를 검사 */
                        printf(", ");           /* 원소 사이에   ,    프린트 */
                }
        }
        printf(") \n");          /* 마지막으로 ")"를 프린트 */
}
```

예제 프로그램

struct listNode : char 타입의 data 필드와 listNode 포인터 타입의 link 필드를 가진 노드

struct linkedList :

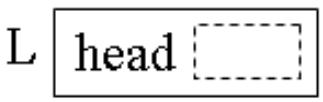
① listNode 타입의 노드로 만들어지는 연결 리스트

② 이 타입의 내부에는 listNode 타입의 포인터 필드 head가 있고, 이 필드는 연결 리스트의 첫 번째 노드를 가리킴

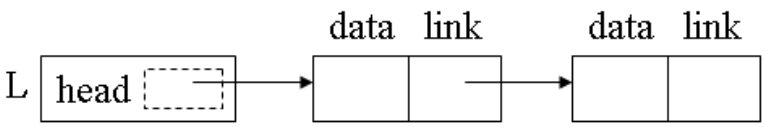
③ 노드가 하나도 없는 공백 리스트는 head = NULL 이 됨

리스트 L을 선언하고 메모리를 할당하는 예

```
linkedList* L;
L = (linkedList *)malloc(sizeof(linkedList));
```



L에 노드가 연결되어 있는 경우



단순 연결 리스트의 처리

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

typedef struct listNode {
                        char data[5];
.
                        struct listNode* link;
} listNode;
typedef struct {
                       listNode* head;
} linkedList_h;

linkedList_h* createLinkedList_h(void);
void freeLinkedList_h (linkedList_h*);
void addLastNode (linkedList_h*, char*);
void reverse (linkedList_h*);
void deleteLastNode (linkedList_h*);
void printList (linkedList_h*);
linkedList_h * createLinkedList_h(void)   {   /*공백 리스트를 만듦 */
              linkedList_h* L;
              L = (linkedList_h *)malloc(sizeof(linkedList_h));
              L->head = NULL;
             return L;
}

void addLastNode (linkedList_h* L, char* x) {
                                    /* addLastNode 알고리즘의 구현 */
              listNode*    newNode;                listNode*    p;
              newNode = (listNode *)malloc(sizeof(listNode));
              strcpy(newNode->data, x);      newNode->link = NULL;
              if (L->head == NULL) {
                        L->head = newNode;
                       return;
              }
              p = L->head;
             while (p->link != NULL)    p = p->link;
              p->link = newNode;
}
```

```
void reverse(linkedList_h* L) {          /* reverse 알고리즘의 구현 */
              listNode*    p;
              listNode*    q;
              listNode*    r;
              p = L->head;
              q = NULL;
              r = NULL;
             while (p != NULL) {
                        r = q;
                        q = p;
                        p = p->link;
                        q->link = r;
              }
              L->head = q;
}    void deleteLastNode(linkedList_h* L) {
              listNode* previous;                listNode* current;
             if (L->head == NULL) return;
             if (L->head->link == NULL) {
                       free(L->head);
                        L->head = NULL;
                       return;
              }
              else {
                        previous = L->head;
                        current = L->head->link;
                       while (current->link != NULL) {
                                 previous = current;
                                 current = current->link;
                        }
                       free(current);
                        previous->link = NULL;
              }
}
void freeLinkedList_h(linkedList_h* L) {
              listNode* p;
             while(L->head != NULL) {
                          p = L->head;     L->head = L->head->link;
                         free(p);              p = NULL;
              }
}
```

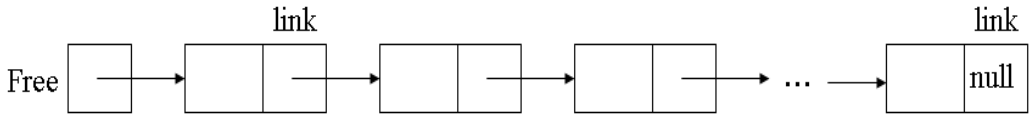
```
void printList(linkedList_h* L) {
              listNode* p;
             printf("(");
              p = L->head;
             while (p != NULL) {
                        printf("%s", p->data);
                         p = p->link;
                        if (p != NULL) printf(", ");
              }
             printf(") \n");
}

int main() {
              linkedList_h*    L;
              L = createLinkedList_h();    /* 공백 리스트 L= ()를 생성 */
              addLastNode(L, "Kim");       /* 리스트 L = (Kim, Lee, Park)을
생성 */
              addLastNode(L, "Lee");
              addLastNode(L, "Park");
              printList(L);  /* (Kim, Lee, Park)을 출력 */
              addLastNode(L, "Yoo");       /* 원소 "Yoo"를 리스트 끝에 첨가
*/
              printList(L);  /* (Kim, Lee, Park, Yoo)를 출력 */
              deleteLastNode(L);       /* 마지막 원소를 삭제 */
              printList(L);  /*(Kim, Lee, Park)을 출력 */
              reverse(L);    /* 리스트 L을 역순으로 변환 */
              printList(L);  /*(Park, Lee, Kim)을 출력 */

              freeLinkedList_h(L);    /* 리스트 L을 해제 */
              printList(L); /* 공백 리스트 ()을 출력 */
             return 0;
}
```

자유 공간 리스트 (free space list)

① 필요에 따라 요구한 노드를 할당할 수 있는 자유 메모리 풀

② 자유 공간 관리를 위해 연결리스트 구조를 이용하기 위해서는 초기에 사용할 수 있는 메모리를 연결 리스트로 만들어 놓아야 함

③ 초기 자유 공간 리스트

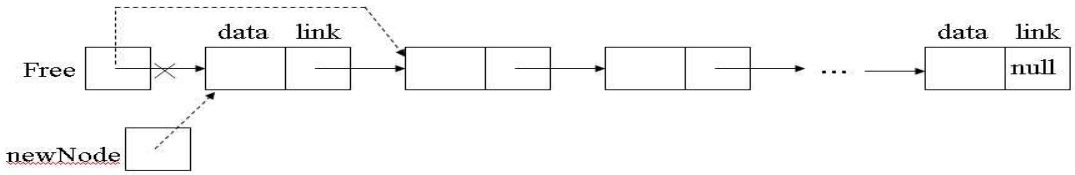


④ 노드 할당 요청이 오면 리스트 앞에서부터 공백 노드를 할당

새로 노드를 할당하는 함수 (getNode)

```
getNode()
        if (Free = null) then
                underflow();        // 언더플로우 처리 루틴
        newNode ← Free;
        Free ← Free.link;
        return newNode;
end getNode()
```

노드를 할당한 뒤의 자유 공간 리스트



가용 공간이 모두 할당된 경우의 처리 방법

1. 프로그램이 더 이상 사용하지 않는 노드들을 자유 공간 리스트에 반환
2. 시스템이 자동적으로 모든 노드들을 검사하여 사용하지 않는 노드들을 전부 수집하여 자유 공간 리스트의 노드로 만든 뒤 다시 노드 할당 작업 재개 (garbage collection)

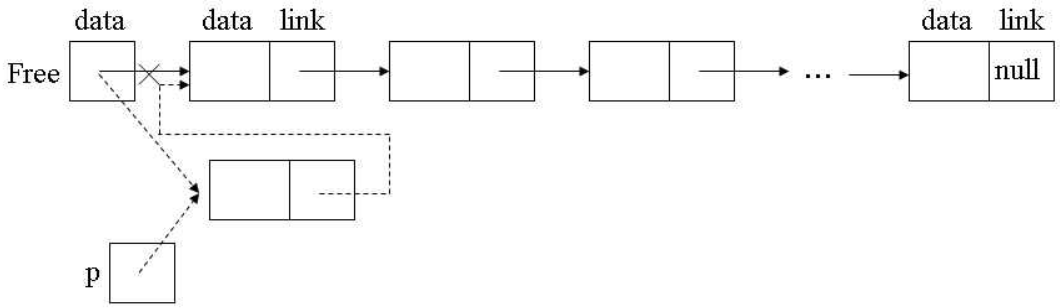
노드 반환

삭제된 노드를 자유 공간 리스트에 반환

- 알고리즘

```
returnNode(p)
        p.link ← Free;
        Free ← p;
end returnNode()
```

반환된 노드가 자유 공간 리스트에 삽입되는 과정



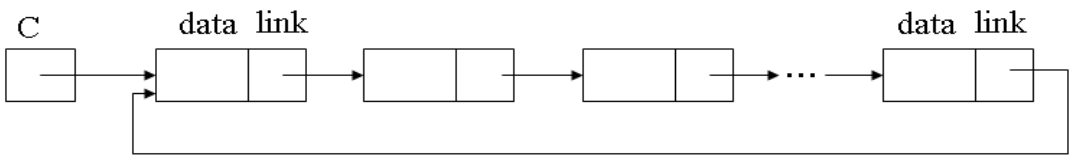
원형 연결 리스트 (circular linked list)

① 마지막 노드가 다시 첫 번째 노드를 가리키는 리스트

② 한 노드에서 다른 어떤 노드로도 접근할 수 있음

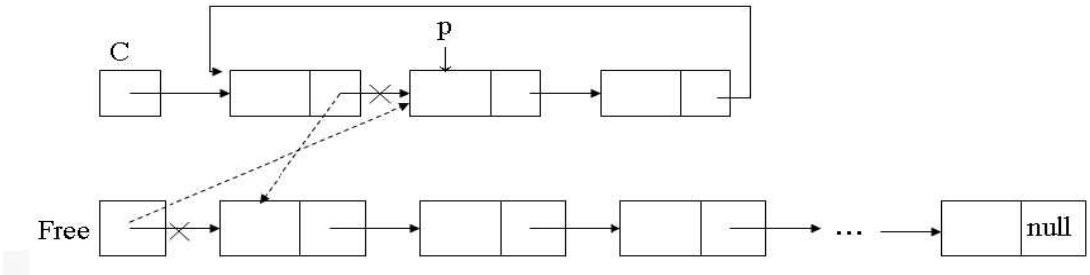
③ 리스트 전체를 가용공간 리스트에 반환할 때 리스트의 길이에 관계없이 일정 시간에 반환할 수 있음

④ 예)



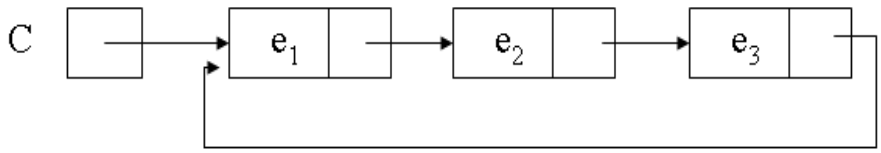
원형 연결 리스트를 자유 공간 리스트에 반환

```
retCList(C)
                // 원형 연결리스트 C를 가용 공간 리스트에 반환
        if (C = null) then return;
        p ← C.link;
        C.link ← Free;
        Free ← p;
end retCList()
```



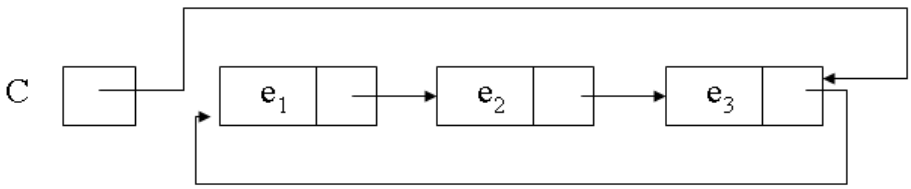
원형 리스트에서의 노드 삽입

l 예) C = (e1, e2, e3)



① 첫 번째 노드로 새로운 노드 p를 삽입할 때 마지막 노드의 link 필드를 변경시키기 위해 마지막 노드를 찾아야 함 (리스트 C의 길이만큼 순회)

② C를 첫 번째 노드가 아니고 마지막 노드를 지시하도록 하면 리스트의 길이에 상관없이 일정 시간에 노드 삽입 가능



노드 삽입 알고리즘

```
insertFront(C, p)
                // CL은 원형리스트의 마지막 노드를 지시
                // p는 삽입할 노드를 지시
            if (C = null) then {
                    C ← p;
                    p.link ← C;
            }
            else {
                    p.link ← C.link;
                    C.link ← p;
            }
end insertFront()
```

리스트 길이 계산

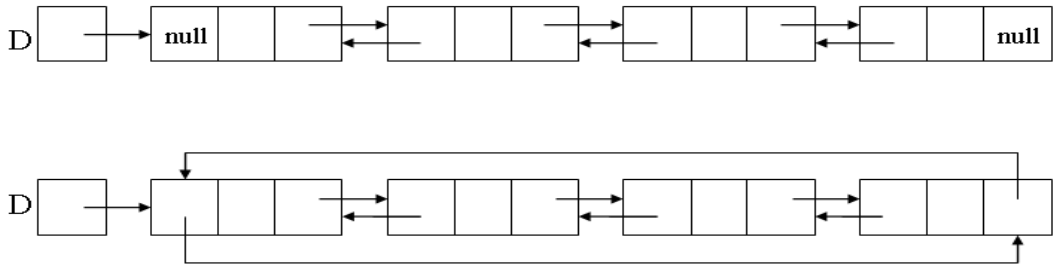
```
lengthC(C)
                    // 원형 리스트 C의 길이 계산
            if (C = null) then return 0;
            length ← 1;
            p ← C.link;       // p는 순회 포인터
            while (p≠C) do {       // p가 처음 출발한 위치에 되돌아 왔는지 검
사
                    length ← length + 1;
                    p ← p.link;
            }
            return length;
end lengthC()
```

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

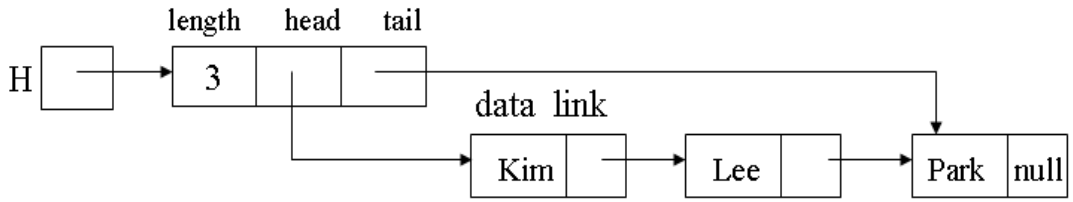
① 예외 경우를 제거하고 코드를 간단하게 하기 위한 방법
② 연결 리스트를 처리하는 데 필요한 정보를 저장
③ 헤더 노드의 구조가 리스트의 노드 구조와 같을 필요는 없음
④ 헤더 노드에는 리스트의 첫 번째 노드를 가리키는 포인터, 리스트의 길이, 참조 계수, 마지막 노드를 가리키는 포인터 등의 정보를 저장

헤더 노드 기능을 갖는 연결 리스트의 정의

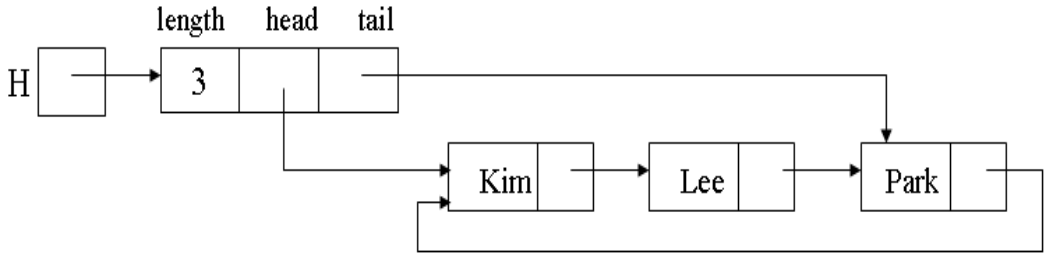
```
typedef struct listNode {
                    /* 리스트 노드 구조 */
                        char data[5];
                        struct listNode* link;
    } listNode;

    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

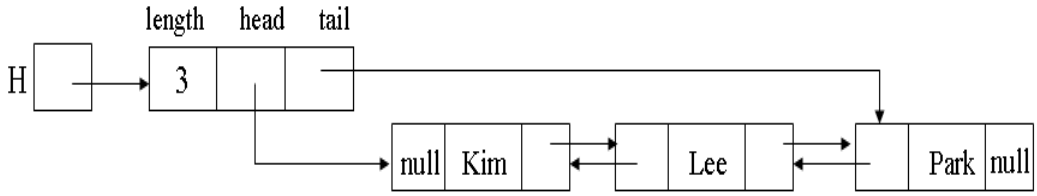
단순 연결 리스트



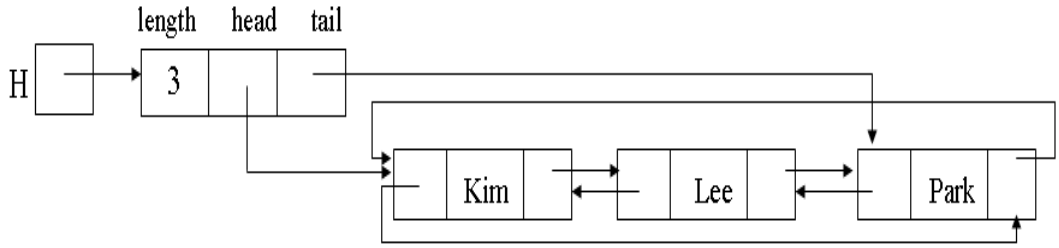
원형 연결 리스트



이중 연결 리스트

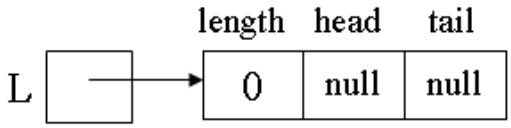


이중 원형 연결 리스트

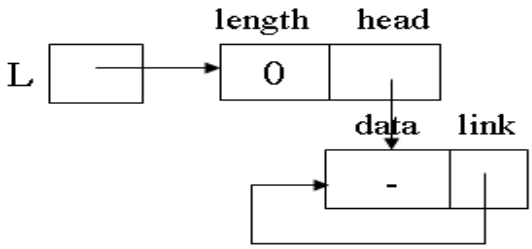


공백 리스트

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