12차시 - 동적 할당(3)

운영체제의 메모리 관리

운영체제의 메모리 할당/해제

- 프로그램 실행 시 메모리 제공
- 프로그램 실행 도중 메모리 제공/해제
- 프로그램 종료 시 모든 메모리 해제

운영체제의 메모리 관리

- **메모리 관리 테이블** 운영
- 프로세스, 메모리 크기, 시작위치 관리
- 최적의 방법에 의해 메모리를 할당

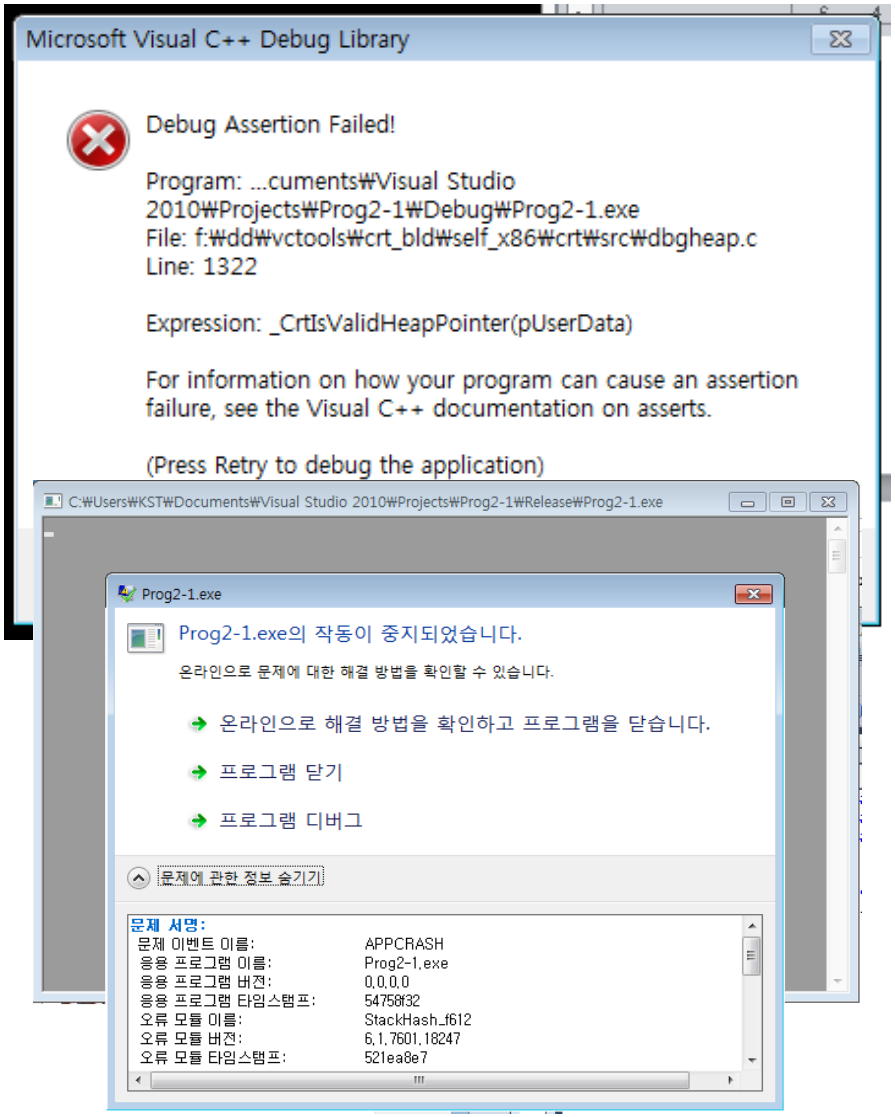
메모리 확보/반납

- **잦은 메모리 확보/반납은 성능을 저하시킨다.**
  - 운영체제의 메모리 관리 테이블을 소진한다.
  - 메모리 관리를 위해 해야 하는 많은 일이 있다.
- **좋은 메모리 사용**

- **직접 메모리 관리를 해야 할 수도 있다.**
  - 고급 프로그래밍
  - 실시간 성능이 중요시되는 게임

메모리 관련 오류들

- 확보되지 않은 공간에 접근
- 반납한 메모리 접근
- 확보에 실패한 것을 모르는 경우
- 그 외

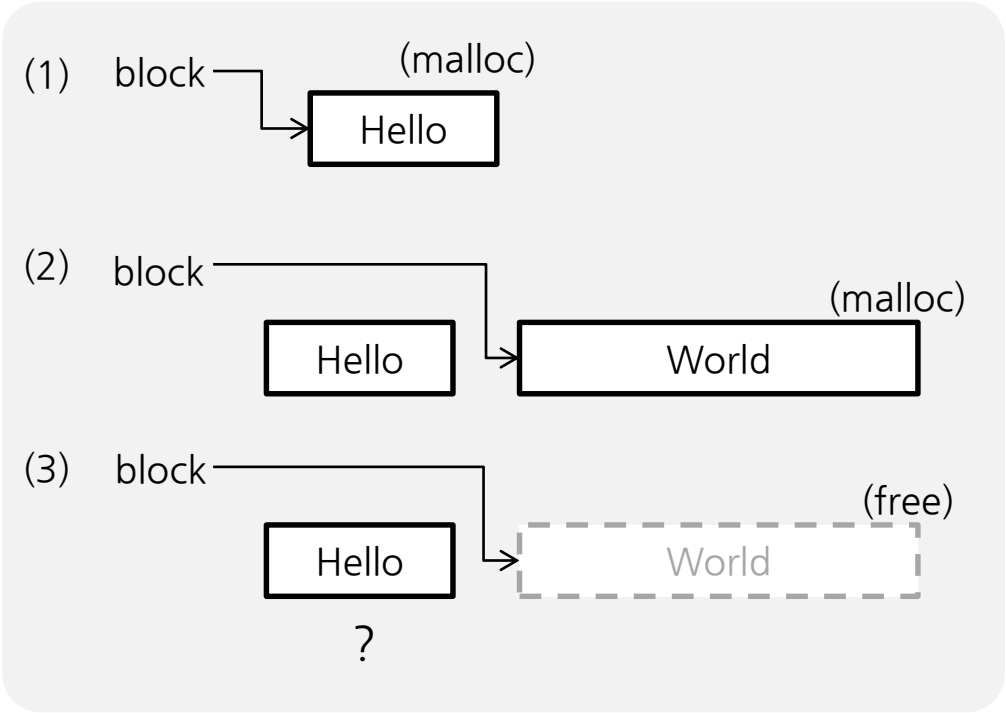


메모리 누수

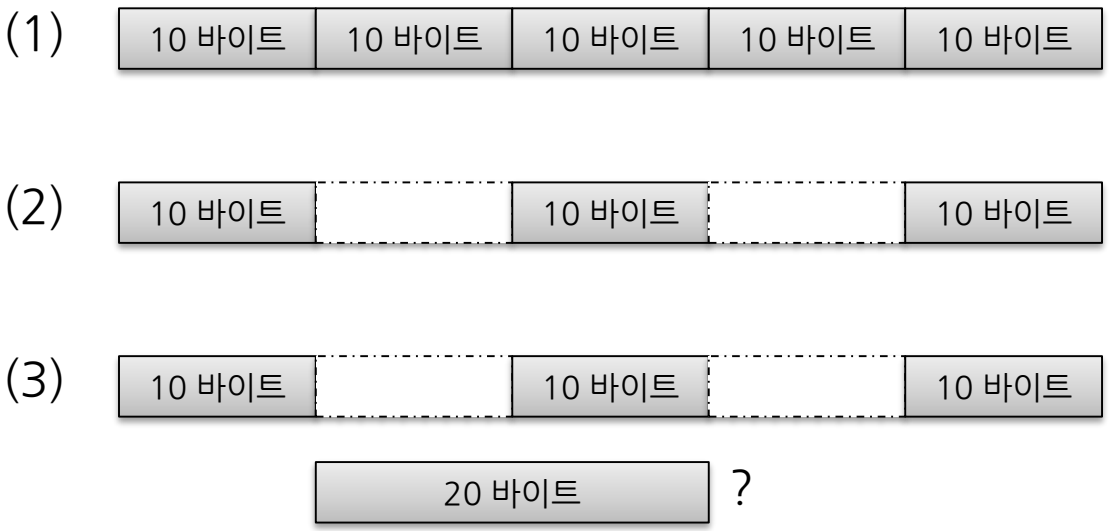
- **메모리가 새는 경우**
  - 사용 가능한 메모리가 줄어든다.
  - 사용중인 메모리가 계속 늘어난다.
- **문제가 되는 경우**
  - 대형 시스템
  - 장기간 동작하는 서버

메모리 누수

```
char   *block;

block = malloc(10);
strcpy(block, “Hello”);

block = malloc(100);
strcpy(block, “World”);

free(block);
```

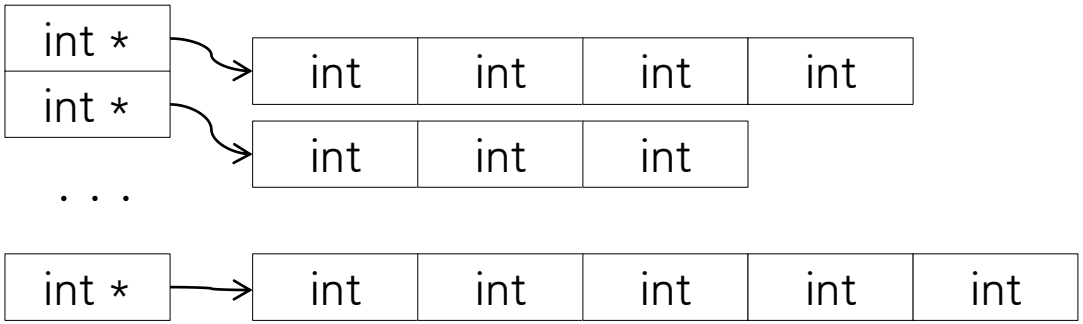


가비지 컬렉션



- **필요성**
  - 공간은 있으나 할당할 수 없는 상황
  - 메모리 관리자(운영체제)의 역할이다.
  - C에서는 없다(Java는 있다).

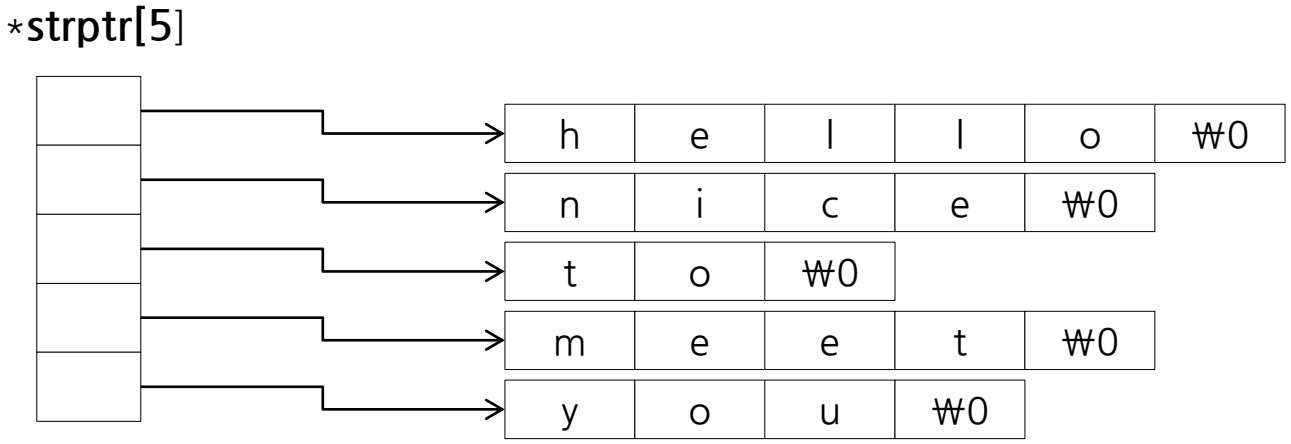
배열과 결합된 형태




```
int  *piarr[5];
int  i;

for( i = 0 ; i < 5 ; i++)
     piarr[i] = malloc (sizeof(int) * 5 );
```

문제점은? 뒤에서 해결.

문자열의 정렬



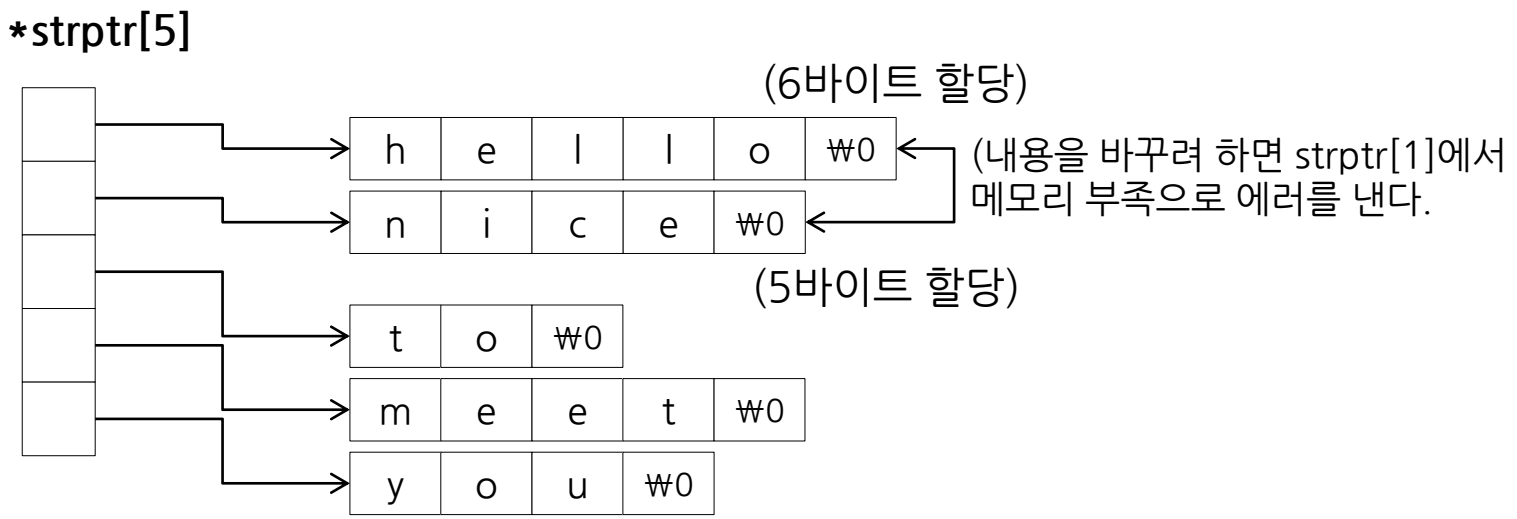
- 저장된 문자열을 알파벳 순으로 정렬하라.

문자열의 정렬

```
char   *strptr[5];
char   input[100], temp[100];
int            i, j;

for (i = 0; i < 5; i++) {
    gets(input);                                  // 문자열을 입력 받은 후,
    strptr[i] = malloc(strlen(input)+1); // 문자열 크기에 맞는 메모리 할당
               strcpy(strptr[i], input);  // 문자열 복사
}
// 버블 소트 시작
for (i = 0; i < 4; i++)
    for (j = i; j < 5; j++)
        if (strcmp(strptr[i], strptr[j]) > 0) { // 앞쪽 문자열이 더 크면
            strcpy(temp, strptr[i]);                  // 앞 뒤 문자열 교환
            strcpy(strptr[i], strptr[j]);
            strcpy(strptr[j], temp);
}
```

메모리 문제가 있다.

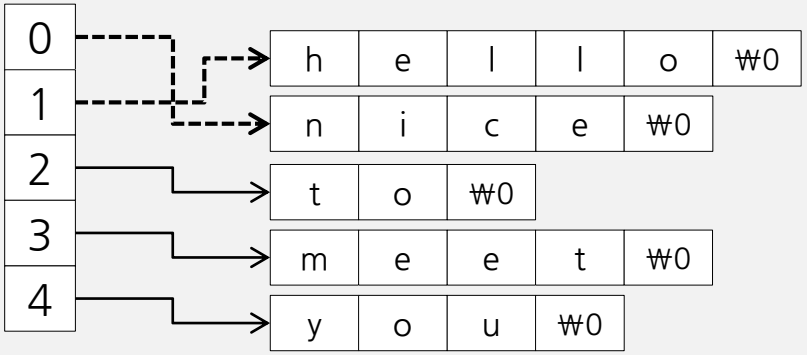
문자열의 정렬



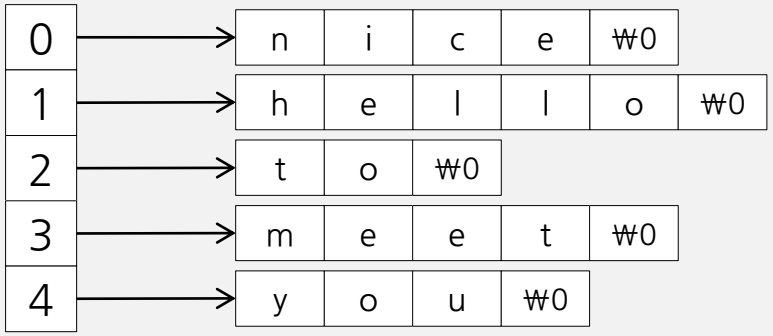

```
if (strcmp(strptr[i], strptr[j]) > 0) {
   strcpy(temp, strptr[i]);

   strptr[i] = realloc(strptr[i], strlen(strptr[j])+1); // 메모리 조정
   strcpy(strptr[i], strptr[j]);

   strptr[j] = realloc(strptr[j], strlen(temp)+1); // 메모리 크기 조정
   strcpy(strptr[j], temp);
}
```

문자열의 정렬

화살표의 방향만 바꾸어주면 된다.

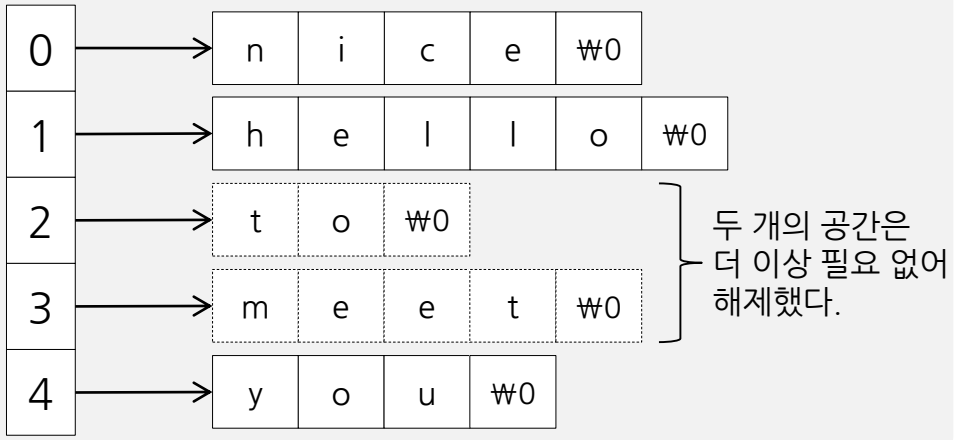


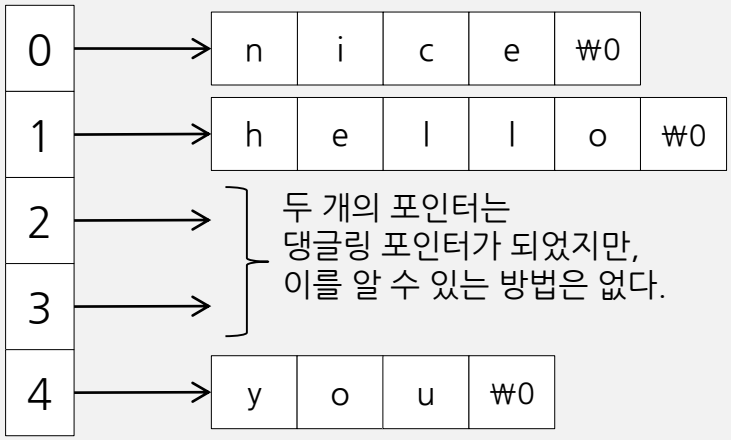
보기 쉽게 재정렬한 형태.

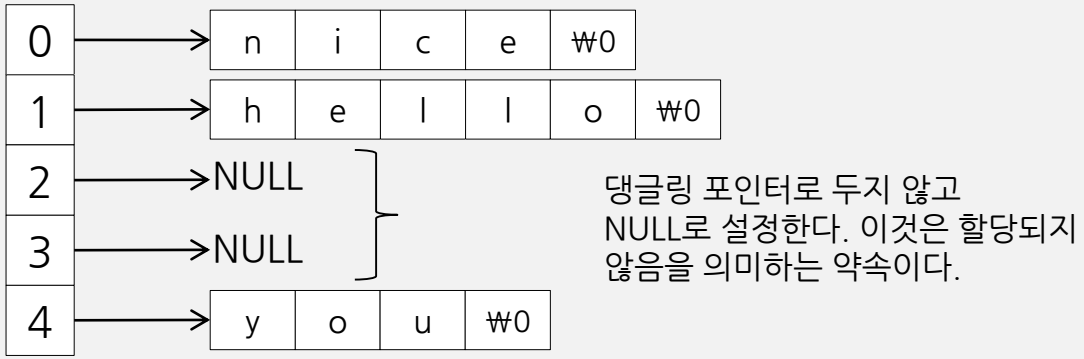



```
if (strcmp(strptr[i], strptr[j]) > 0) {

    char     *tempptr;
    tempptr = strptr[i];
    strptr[i] = strptr[j];
    strptr[j] = tempptr;
}
```

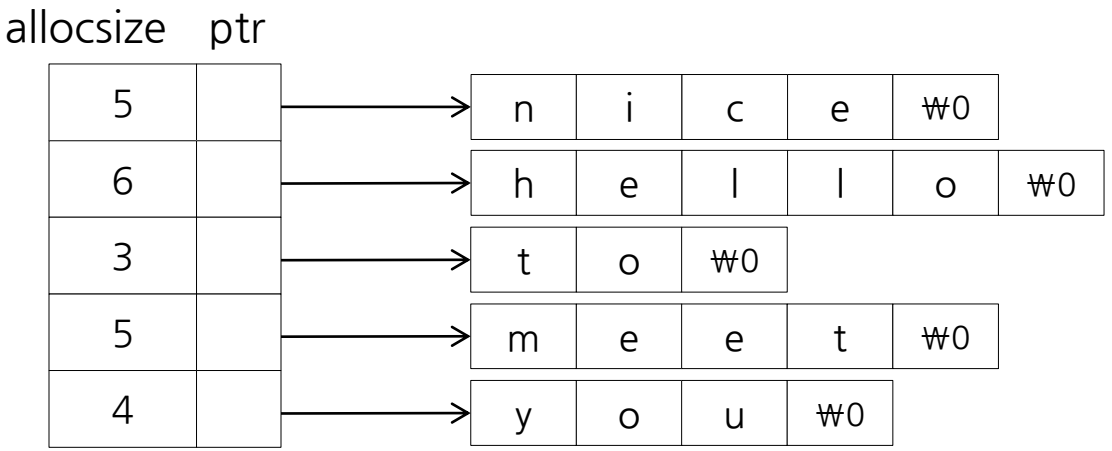
댕글링 포인터의 처리


0
1
2
3
4
n i c e ₩0
h e l l o ₩0
t o ₩0
m e e t ₩0
y o u ₩0
두 개의 공간은
더 이상 필요 없어
해제했다.


0
1
2
3
4
n i c e ₩0
h e l l o ₩0
y o u ₩0
두 개의 포인터는
댕글링 포인터가 되었지만,
이를 알 수 있는 방법은 없다.

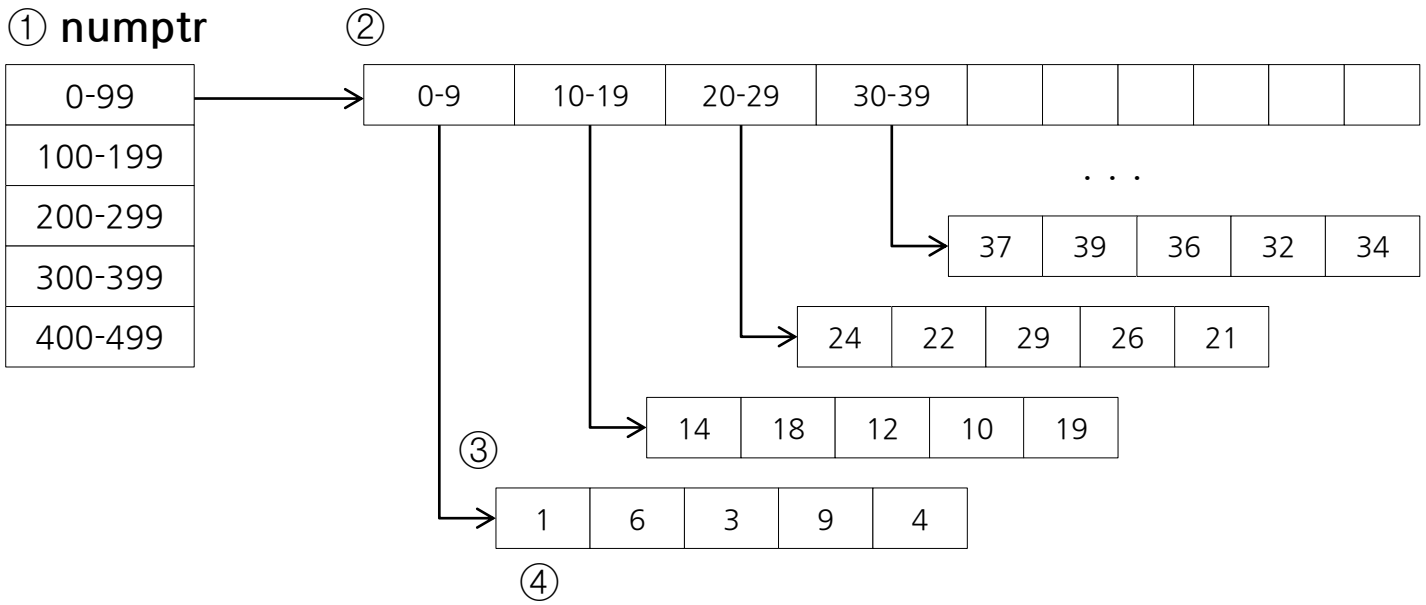

0
1
2
3
4
n i c e ₩0
h e l l o ₩0
NULL
NULL
y o u ₩0
댕글링 포인터로 두지 않고
NULL로 설정한다. 이것은 할당되지
않음을 의미하는 약속이다.

구조체와 결합된 자료형




```
struct {                    // 구조체의 배열로 선언한다.
    int   allocsize;        // 할당한 공간의 크기를 넣는다. 처음에는 0을 넣는다.
    char *ptr;              // 할당할 메모리를 가리킨다.(전과 동일)
} strptr[5];

// 메모리를 할당할 때
strptr[2].ptr = malloc(3);
strptr[2].allocsize = 3;
strcpy(strptr[2].ptr, “to”);

// 해당 메모리에 다른 문자열(tempstr)을 넣을 때는?
```

이중 포인터와 동적 할당

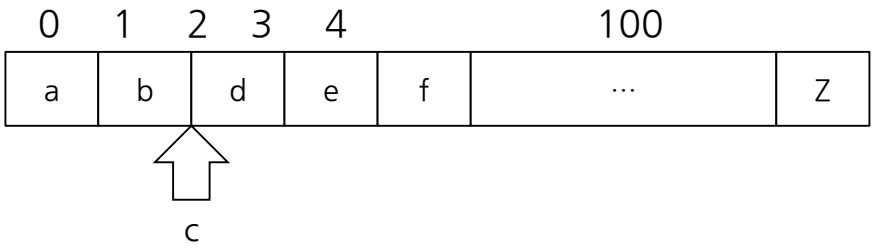


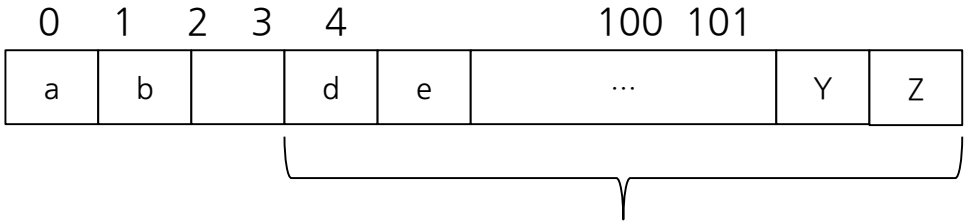

```
int **numptr[5];                                    // ①

numptr[0] = malloc(sizeof(int *) * 10);     // ②
*numptr[0] = malloc(sizeof(int) * 5);       // ③
**numptr[0] = 1;                                    // ④
*(*numptr[0] +1) = 6;

printf("%d\n", **numptr[0]);
```

링크드리스트

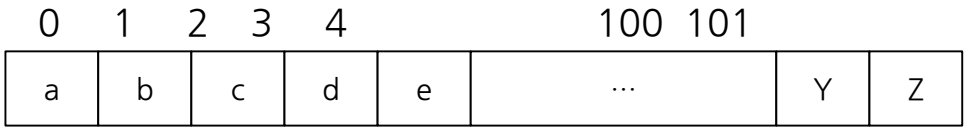
배열의 단점

- 삽입, 삭제가 어렵다.





뒤로 한 칸씩 옮긴다.

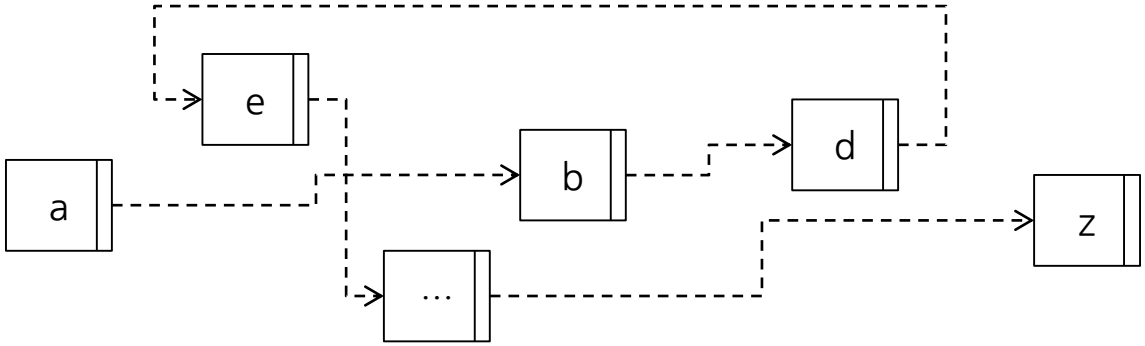


공백이 생겼으므로 c를 넣는다.

뒤로 밀 때 주의해야 한다.

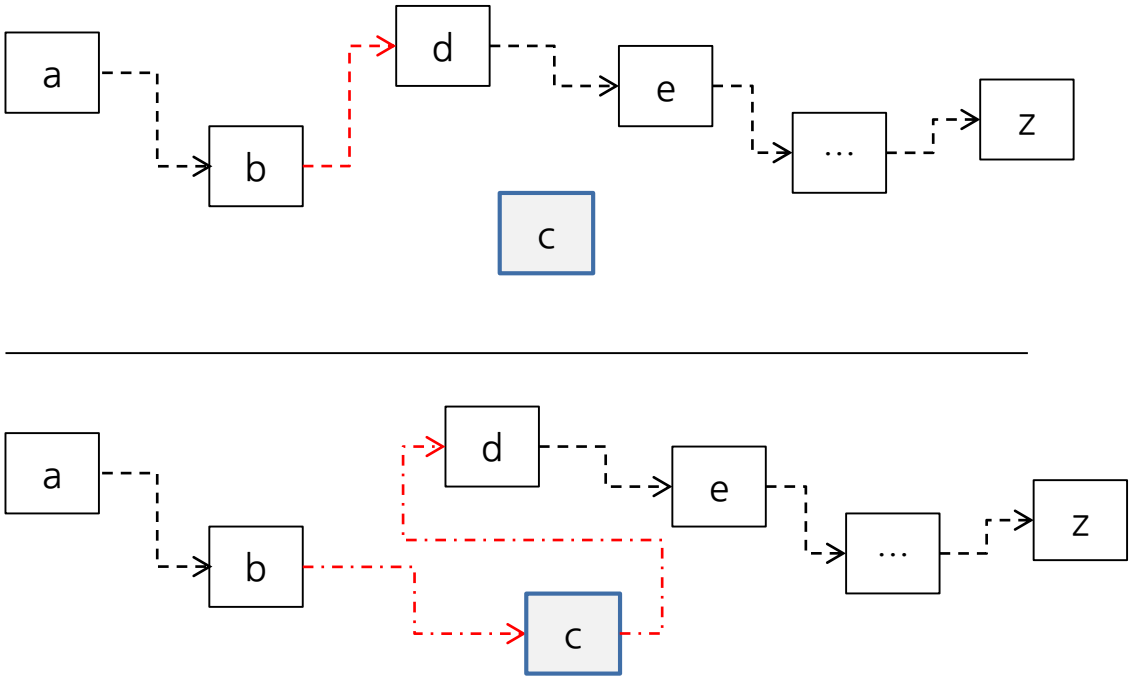
링크드리스트

- 데이터가 어디에 있든 위치만 연결하면 된다.



링크드리스트

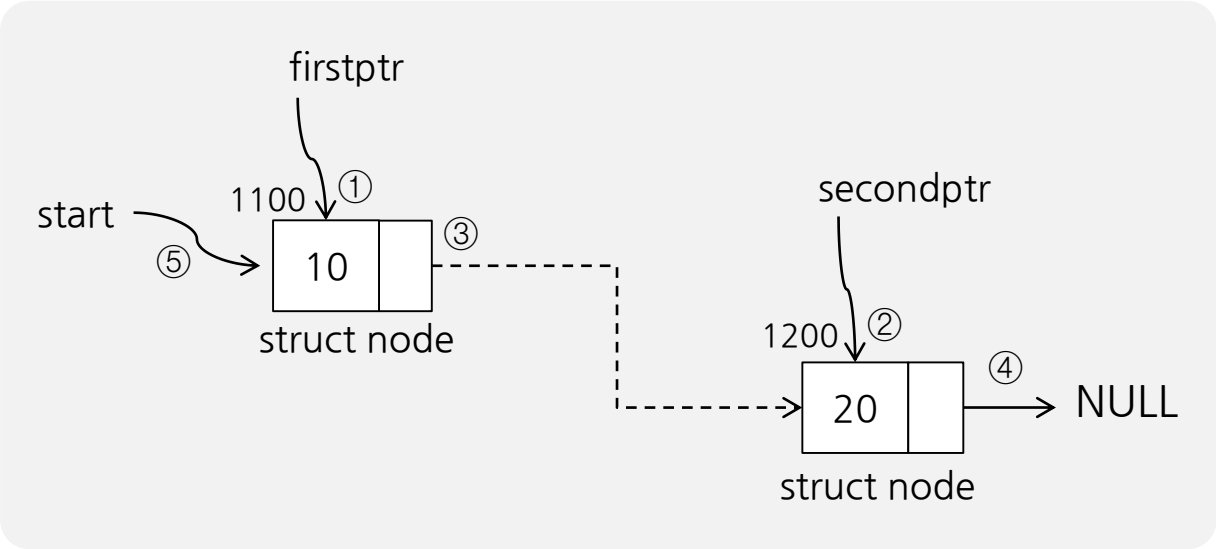
- 삽입

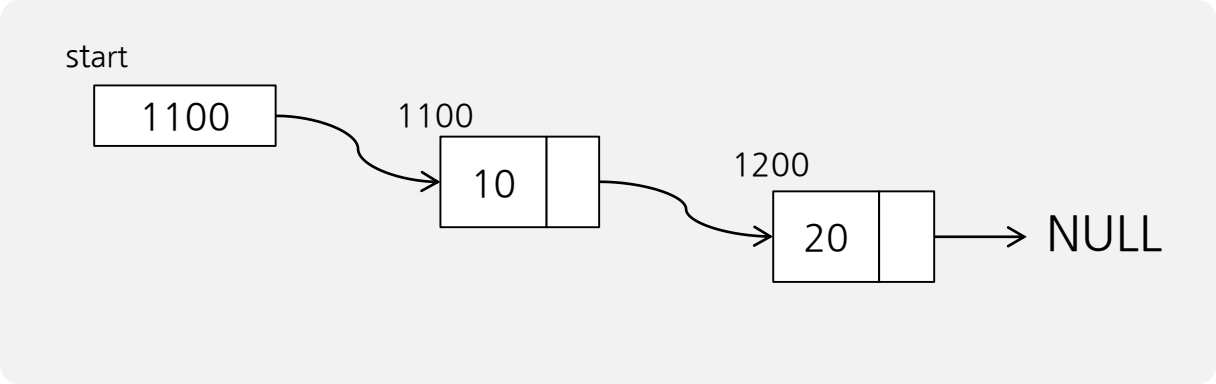


링크드리스트

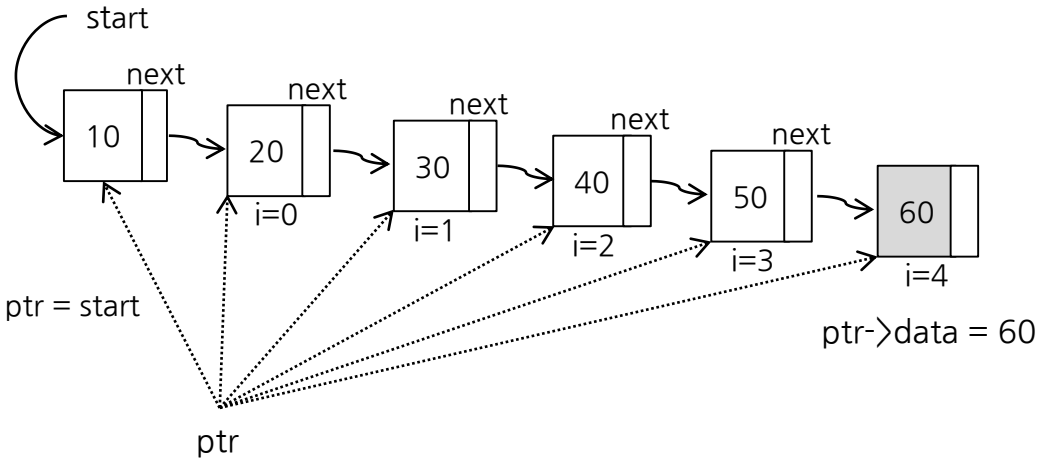
```
struct node {
    int data;
    struct node *next;
};
struct node *firstptr, *secondptr, *start;

firstptr = malloc ( sizeof(struct node) );  // ① 첫 번째 데이터
firstptr->data = 10;

start = firstptr;   // 링크드리스트의 시작을 기억함

secondptr = malloc ( sizeof(struct node) );  // ② 두 번째 데이터
secondptr->data = 20;

firstptr->next = secondptr;  // ③
secondptr->next = NULL;      // ④
```

링크드리스트


firstptr
start
1100
①
③
⑤
10
struct node
secondptr
1200
②
④
20
NULL
struct node

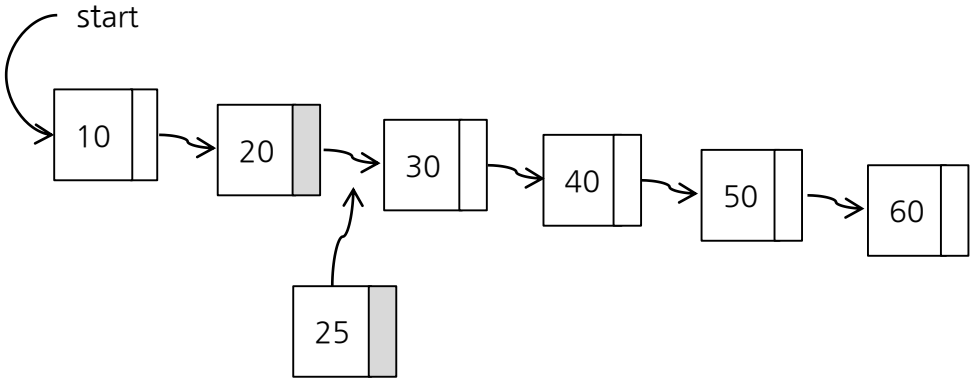

start
1100
1100
10
1200
20
NULL

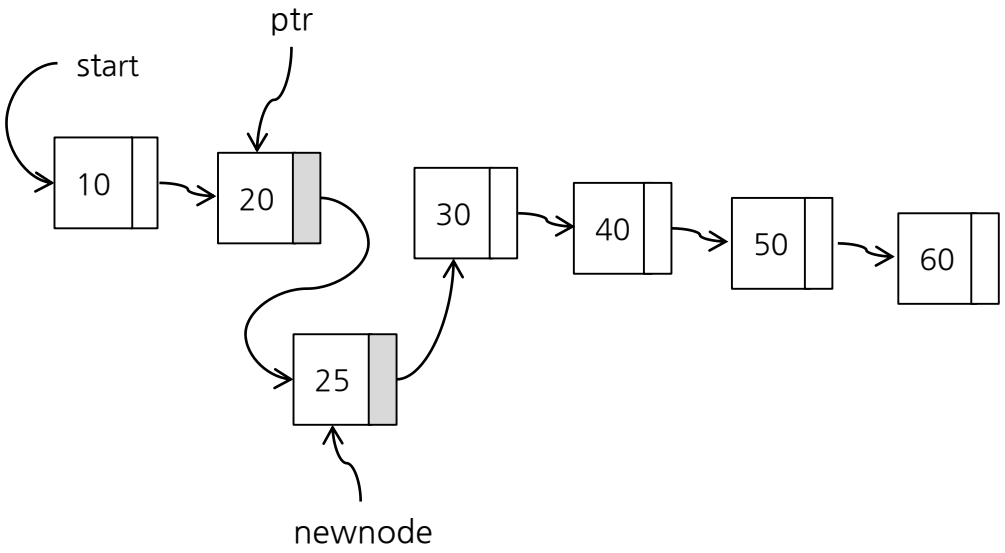
링크드리스트




```
struct node  *ptr, *start;
// 이미 노드가 생성되고 데이터가 들어있어야 한다.

ptr = start;    // 링크드리스트의 시작점부터
for( i = 0 ; i < 5 ; i++)
      ptr = ptr -> next;      // 다섯번 건너뜀

printf("여섯번째 노드의 데이터는 %d\n", ptr->data );
```

링크드리스트





링크드리스트

```
struct node  *ptr, *newnode;
ptr = start;                          // 링크드리스트의 시작점부터
insertvalue = 25;                     // 추가할 데이터

while (ptr) {
    if (ptr->next->data > insertvalue) {    // 삽입할 위치를 찾았다면
        newnode = malloc( sizeof(struct node) );// 새 노드를 만들고
        newnode -> data = insertvalue;                  // 값을 채우고
        newnode -> next = ptr->next;// 추가된 노드의 다음 데이터 화살표 설정
        ptr->next = newnode;        // 앞 노드의 다음 데이터 화살표는 새 노드
        break;                                  // 노드 추가 완료
    }
    ptr = ptr -> next;      // 다음 노드로 이동
}
```

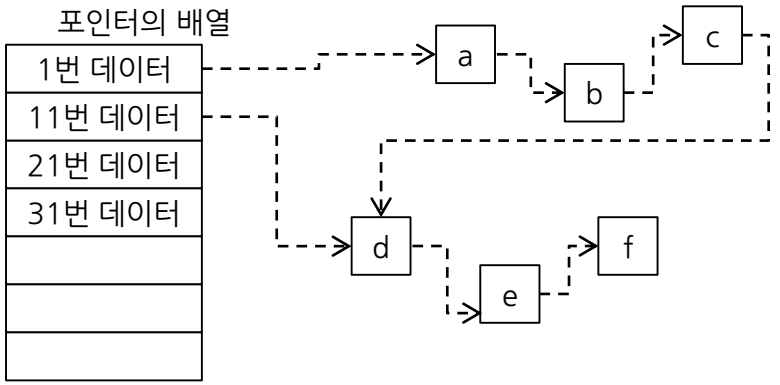
링크드리스트

- **특징**
  - N번째 데이터에 도달하려면 앞에서부터 찾아야 한다.
  - 링크가 끊어지면 데이터가 유실된다.
  - 가장 마지막 노드의 next는 NULL이다. 데이터의 끝을 의미한다.

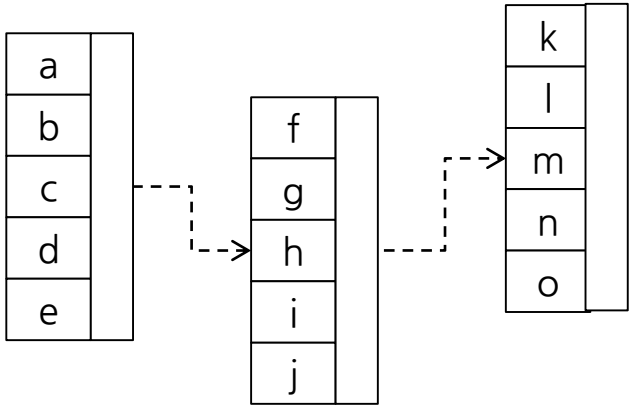
링크드리스트

단점과 그 해결

- N번째 데이터를 찾기 위해 오래 걸린다.
  - 해결 : 배열의 포인터로 색인을 만든다.



- 운영에 데이터보다 링크가 더 많다.
  - 해결 : 데이터 부분을 더 크게 한다.



해결책을 찾는 것은 여러분의 몫