Lecture 06

구조체

리뷰

- 포인터
  - 변수의 메모리 번지를 기억하는 것
  - 메모리 주소를 알아내는 연산자 &
  - 포인터가 가리키는 변수의 값을 알아내는 연산자 *
  - 포인터 선언 및 사용

```
int x=10;
int *px = &x;
*px += 10; // x=20
```

  - 번지 연산
    - sizeof()
    - 증가/감소
    - 포인터의 형에 따라 포인터가 가리키는 변수의 값이 해석됨

리뷰

- 문자열
  - `string.h` 표준 라이브러리에서 문자열 관련 함수들이 제공됨
  - 문자열 복사: `strcpy()`
  - 문자열 비교: `strcmp()`
  - 문자열의 길이 계산: `strlen()`
  - 문자열 접합: `strcat()`
  - 문자열 내에서 탐색: `strchr()`, `strstr()`
  - …

리뷰

- 탐색(searching)
  - 선형 탐색(linear search)
    - 복잡도 O(n)
  - 이분 탐색(binary search)
    - 복잡도 O(logn)
    - 탐색 전 입력 배열이 정렬되어 야 함
- 정렬(sorting)
  - 삽입 정렬(insertion sort)
    - 복잡도 $O(n^2)$
  - 퀵 정렬(quick sort)
    - 복잡도 O(nlogn)

제일 효율적임

O(1)

O(logn)

O(n)

O(nlogn)

$O(n^2)$

$O(n^k)$, $k>2$

$O(k^n)$

제일 비효율적임

O(n!)

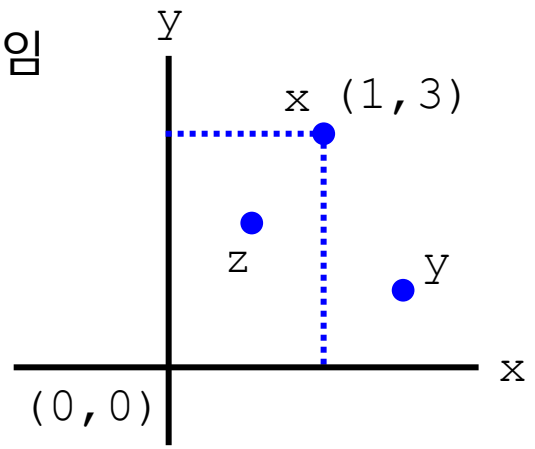
기본 개념

- 여러 개의 데이터형을 포함하는 새로운 형을 정의하는 것

```
struct point {
    int x;
    int y;
};
```

  - **struct** 뒤에 오는 것은 이름 또는 tag임
  - **struct**에 속하는 변수들은 멤버(member)임
  - 끝의 우측 중괄호 뒤에 변수 정의 가능

```
struct point {
    int x;
    int y;
} x, y, z;
```



기본 개념

- 구조체 이름을 사용해서 변수를 선언할 수 있음

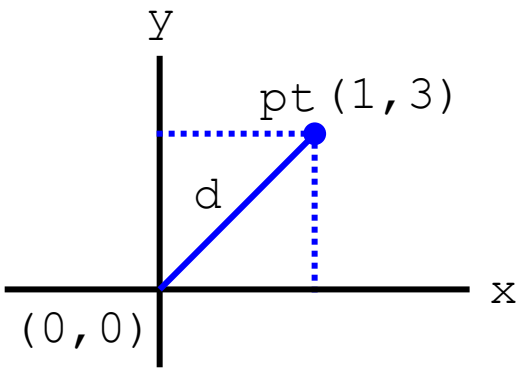
```
struct point pt1;
```

- 선언 시 각 요소에 대한 초깃값을 같이 정의할 수 있음

```
struct point pt2 = {100, 200};
```

- 구조체 멤버 연산자 “.”를 통해 멤버를 사용할 수 있음
  - 예로 점 pt의 원점에서부터의 거리 계산

```
float d;
d = sqrt((float)pt.x*pt.x +
         (float)pt.y*pt.y);
```



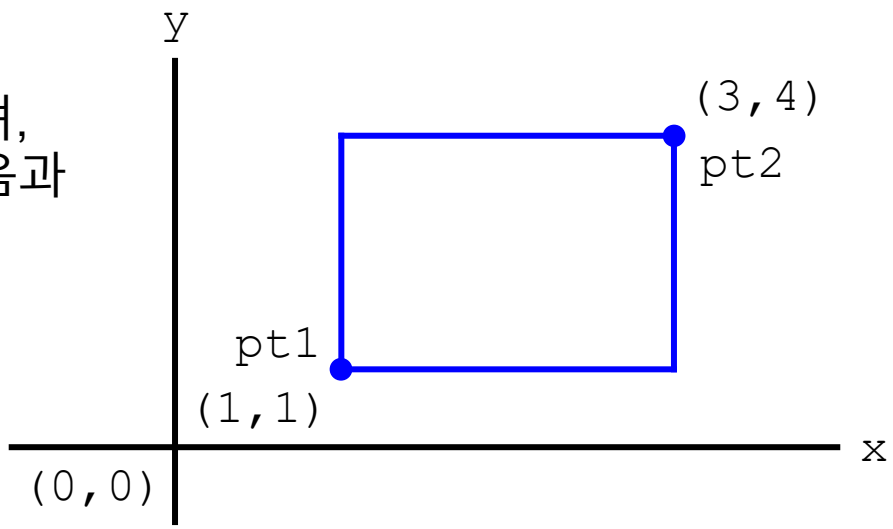
기본 개념

- 구조체의 구조체도 있을 수 있음
  - 예로 대각선으로 마주 본 두 점을 지정해주는 사각형

```
struct rect {
    struct point pt1;
    struct point pt2;
} screen;
```

  - screen 변수는 선언되었으며, screen의 pt1의 좌표는 다음과 같이 사용될 수 있음

```
screen.pt1.x
screen.pt1.y
```



구조체와 함수

- 구조체에 할 수 있는 연산
  - 복사
  - 지정
  - 주소를 알아내기(&)
- 예로 점을 만들기

```
struct point makepoint(int x, int y) {
    struct point tmp;
    tmp.x = x;
    tmp.y = y;
    return tmp;
}
```

```
int main() {
    ...
    struct point pt1 = {10,20};
    struct point pt2 = pt1;
    struct point pt3 = makepoint(10,20);
    ...
    return 0;
}
```

구조체와 함수

- 구조체에 더하기
  - + 연산자를 사용할 수 없으므로 함수로 해 야 함

```
struct point pt1 = {10,20}, pt2 = {5,5};
struct point pt3 = pt1+pt2; // 에로
```

대신

```
struct point addpoint(struct point pt1, struct point pt2) {
    pt1.x += pt2.x;
    pt1.y += pt2.y;
    return pt1;
}
struct point pt3 = addpoint(pt1, pt2);
```

구조체의 포인터

- 구조체가 함수로 전달된다면 구조체의 멤버들이 다 복사됨
  - 큰 구조체의 경우는 효율적이 아님

```
struct employee {
    int age;
    char name[100];
    char position[100];
};
struct company {
    struct employee engineer[100];
    struct employee personnel[50];
}
```

  - 포인터를 함수로 넘겨주는 것이 더 효율적임

```
struct company com;
struct company *pc = &com;
```

구조체의 포인터

- 구조체가 함수로 전달된다면 구조체의 멤버들이 다 복사됨
  - 포인터를 통해 구조체의 멤버를 사용할 수 있음

```
struct employee em1 = {40, "John", "Tech Leader"};
struct employee *pe1 = &em1;
printf("%s - %d - %s\n", (*pe1).name, (*pe1).age, (*pe1).position);
```

  - 구조체 멤버 연산자 .는 *보다 우선순위가 높아서 괄호가 필요함
  - 괄호를 사용하지 않는 표현 방법도 있음

```
pe1->name
pe1->age
pe1->position
```

  - 구조체의 구조체의 포인터가 마찬가지

```
struct company com, *pc = &com;
```

pc->engineer[0].age, (*pc).engineer[0].age하고 com.engineer[0].age는 같은 결과를 가져옴

구조체의 배열

- C에서 사용되는 키워드 중 어느 것이 몇 번 나왔는지 세는 프로그램을 작성하려면 2개의 배열이 필요함
  - 키워드를 나타내는 문자열의 배열 : `char *keyword[32];`
  - 키워드가 나오는 횟수를 세기 위한 정수의 배열 : `int keycount[32];`
- 구조체의 배열을 사용할 수도 있음

```
struct key {
    char *word;
    int count;
} keytab[32];
```

또는

```
struct key {
    char *word;
    int count;
};
struct key keytab[32];
```

구조체의 배열

- 구조체의 배열을 정의할 때 초깃값을 같이 줄 수 있음

```
struct key keytab[] {
    "auto", 0,
    "break", 0,
    "case", 0,
    /* ... */
    "volatile", 0,
    "while", 0
};
```

```
struct key keytab[] {
    {"auto", 0},
    {"break", 0},
    {"case", 0},
    /* ... */
    {"volatile", 0},
    {"while", 0}
};
```

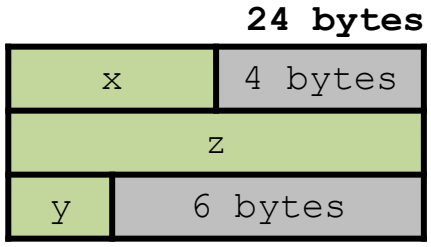
구조체의 배열

- 구조체의 배열의 길이를 계산하기

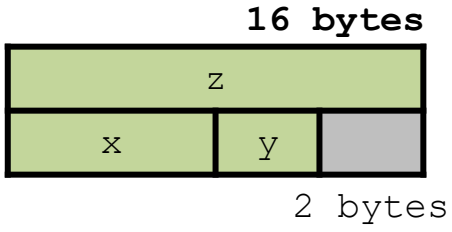
  sizeof(구조체의 배열)/**sizeof**(구조체)

- 구조체의 크기
  - 멤버의 크기의 합보다 크거나 같음
  - 멤버 순서와 시스템 메모리에 따라 패딩(padding)이 실행될 수 있음

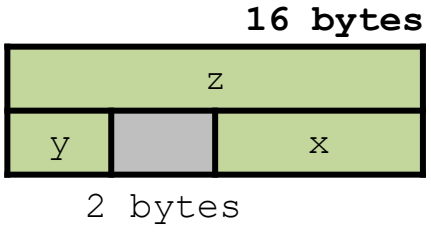
```
struct A {
   int x; // 4 바이트
   double z; // 8 바이트
   short int y; // 2 바이트
};
```




```
struct B {
   double z;
   int x;
   short int y;
};
```




```
struct C {
   double z;
   short int y;
   int x;
};
```



동적 메모리 배당

- **void** *malloc(size_t n)
  - 원하는 바이트만큼 메모리를 할당하주고 **void** 포인터를 리턴함
  - 실패 시 NULL을 리턴함
  - 초기화해주지 않음(리턴한 메모리에서 있는 값은 쓰레기 값임)
  - 사용 시 리턴한 **void** 포인터를 원하는 형으로 변환해 야 함

```
int *px = (int*)malloc(sizeof(int)*100); // 100개의 정수로 이루어진 배열 할당
```

- **void** *calloc(size_t n, size_t size)
  - n개의 size 크기인 요소로 이루어진 배열에 대한 메모리를 할당해줌
  - 실패 시 NULL을 리턴함
  - 배열의 요소들을 0으로 초기화해줌

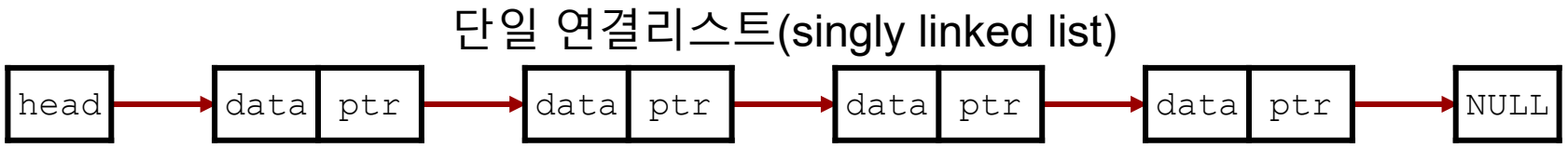
- **void** free(**void***)
  - malloc나 calloc로 할당한 메모리를 해제해줌

동적 메모리 배당

- 실수하기 쉬운 부분
  - 동적으로 할당한 메모리를 사용한 뒤 `free`하지 않음
    - 메모리 누수 문제가 발생함
    - 항상 함수 끝에 동적으로 할당한 메모리를 해제해 야 함
  - 성공적으로 할당했는 지를 체크하지 않음
    - 동적 할당 실패 시 프로그램이 크래시됨
    - 동적 할당 시 항상 리턴한 포인터가 `NULL`인지 체크해 야 함
  - `free`로 해제했던 메모리를 다시 사용함
    - `free`했던 메모리를 접속하면 프로그램이 크래시됨
    - 항상 접속할 메모리가 유효한 지 체크해 야 함

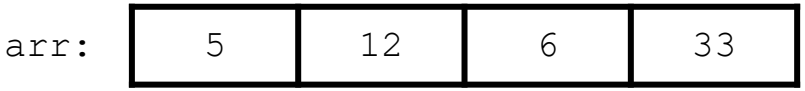
연결리스트(linked list)

- 일반적으로 사용하는 배열과 달리 동적으로 각 칸들이 앞, 뒤로 사슬처럼 연결되어 있는 자료구조

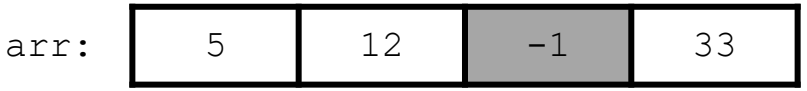


연결리스트(linked list)

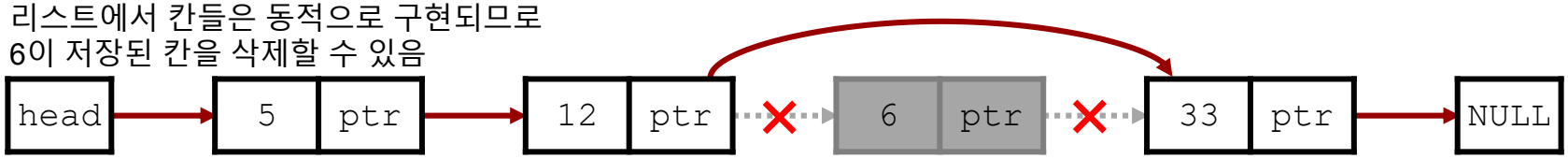
- 연결리스트를 사용하는 이유
  - 칸 자체를 삭제할 수 있음



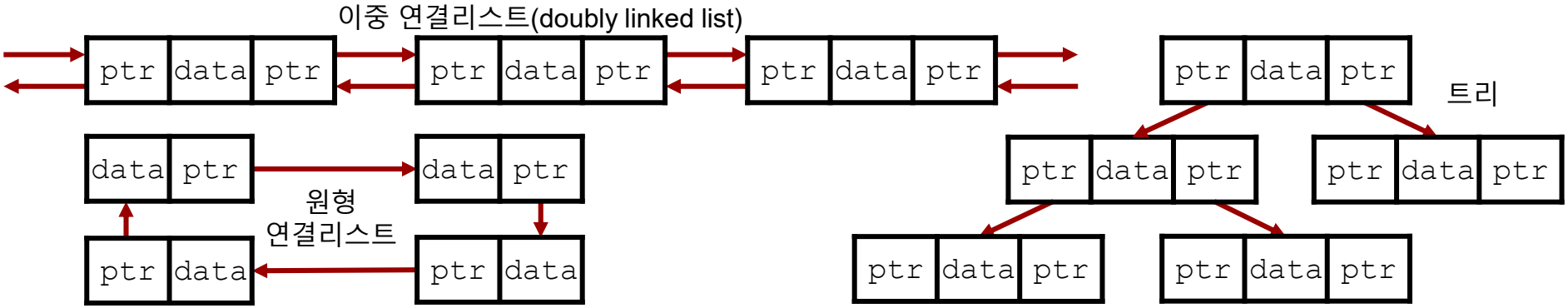
배열에서 6을 삭제하고 싶다면 삭제되었다는 "표시"만 남길 수 있음



리스트에서 칸들은 동적으로 구현되므로 6이 저장된 칸을 삭제할 수 있음



  - 원하는 형태대로 리스트를 구현할 수 있으므로 다양하게 활용할 수 있음



연결리스트(linked list)

- 단일 연결리스

```
struct node {
    int data;
    struct node *next;
};
```

- 연산
  - 삽입(insertion)
    - 리스트 처음에 삽입
    - 리스트 끝에 삽입
    - 특정 위치에 삽입
  - 삭제(deletion)
    - 리스트의 처음 노드 삭제
    - 리스트의 끝 노드 삭제
    - 리스트의 특정 노드 삭제
  - 종주(traversal)

연결리스트 연산		시간 복잡도	공간 복잡도
삽입 (insertion)	처음에	O(1)	O(1)
	끝에	O(N)	O(1)
	특정 위치에	O(N)	O(1)
삭제 (deletion)	처음 노드	O(1)	O(1)
	끝 노느	O(N)	O(1)
	특정 노드	O(N)	O(1)
종주(traversal)		O(N)	O(1)

연결리스트(linked list)

- 단일 연결리스
  - 노드 만들기

```
struct node *createNode(int data) {
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}
```

  - 리스트 종주

```
void traversal(struct node *head) {
   while (head != NULL) {
      printf("%d ", head->data);
      head = head->next;
   }
   printf("\n");
}
```

연결리스트(linked list)

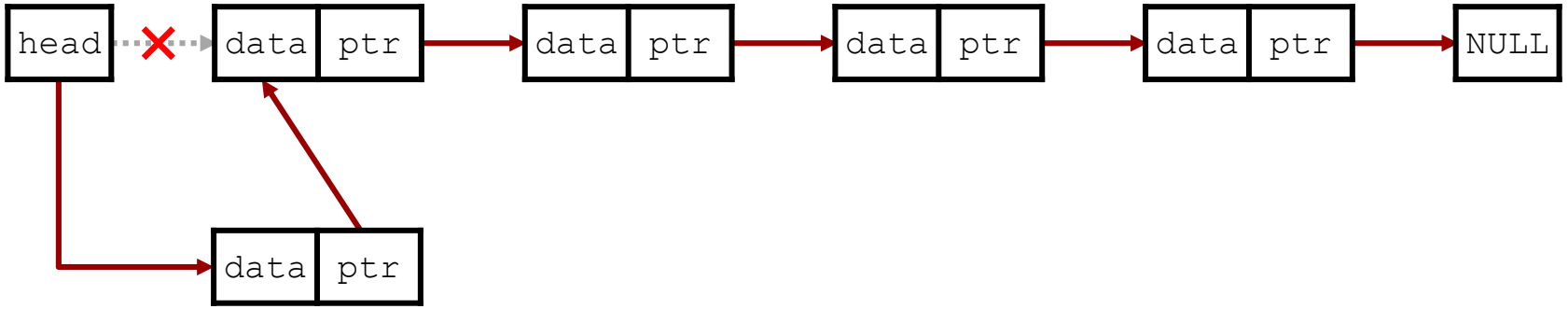
- 단일 연결리스
  - 예로 리스트를 만들고 종주하기

```
int main() {
    // 10->20->30->40 연결리스트를 만들기
    struct node *head = createNode(10);
    head->next = createNode(20);
    head->next->next = createNode(30);
    head->next->next->next = createNode(40);

    traversal(head);

    return 0;
}
```

연결리스트(linked list)

- 단일 연결리스
  - 리스트 처음에 삽입

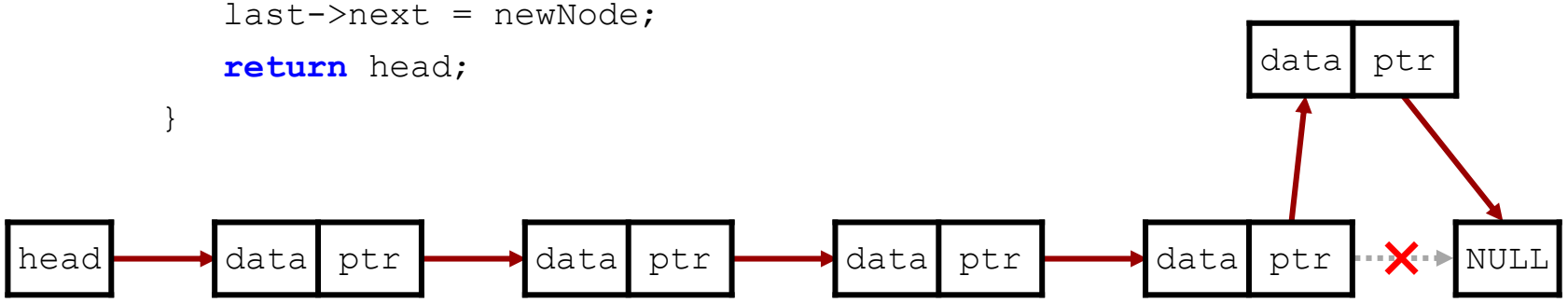
```
struct node *insertFront(struct node* head, int data) {
    struct node *newNode = createNode(data);
    newNode->next = head;
    return newNode;
}
```



연결리스트(linked list)

- 단일 연결리스
  - 리스트 끝에 삽입

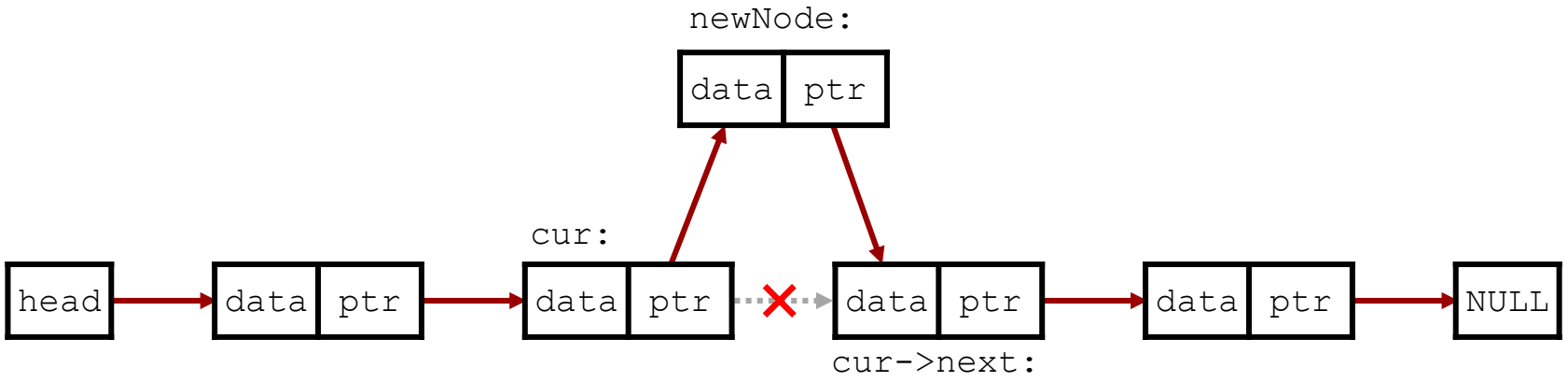
```
struct node *insertTail(struct node* head, int data) {
    struct node *newNode = createNode(data);
    if (head == NULL)
        return newNode;
    struct node *last = head;
    while (last->next != NULL)
        last = last->next;
    last->next = newNode;
    return head;
}
```



연결리스트(linked list)

- 단일 연결리스
  - 리스트 특정 위치에 삽입
    - cur 포인터를 생성하여 삽입 위치로 이동시킴
    - newNode를 삽입함

```
newNode->next = cur->next;
cur->next = newNode;
```



연결리스트(linked list)

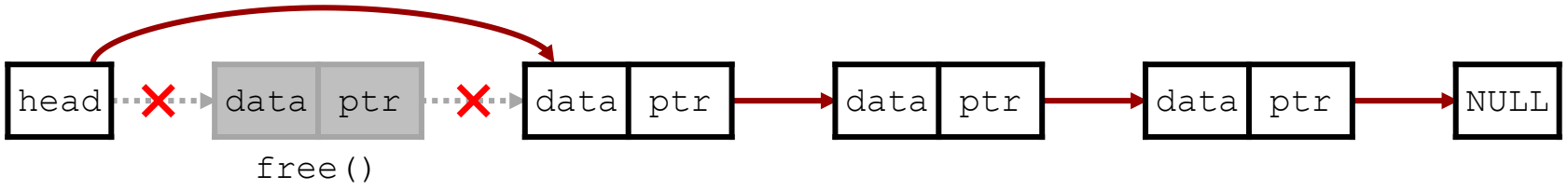
- 단일 연결리스
  - 리스트 특정 위치에 삽입

```
struct node *insertPos(struct node* head, int pos, int data) {
   if (pos < 1) return head; // 유효하지 않는 위치
   if (pos == 1) { // 처음에 삽입
      struct node *newNode = createNode(data);
      newNode->next = head;
      return newNode;
   }
   struct node *cur = head;
   for (int k=1; k < pos-1 && cur != NULL; k++)
      cur = cur->next;
   if (cur == NULL) return head; // 유효하지 않는 위치
   struct node *newNode = createNode(data);
   newNode->next = cur->next;
   cur->next = newNode;
   return head;
}
```

연결리스트(linked list)

- 단일 연결리스
  - 리스트의 처음 노드 삭제

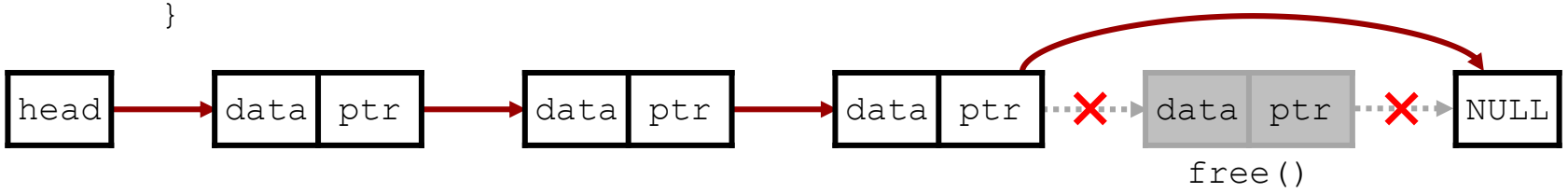
```
struct node *deleteFront(struct node *head) {
    if (head == NULL)
        return NULL;
    struct node *tmp = head;
    head = head->next;
    free(tmp);
    return head;
}
```



연결리스트(linked list)

- 단일 연결리스
  - 리스트의 끝 노드 삭제

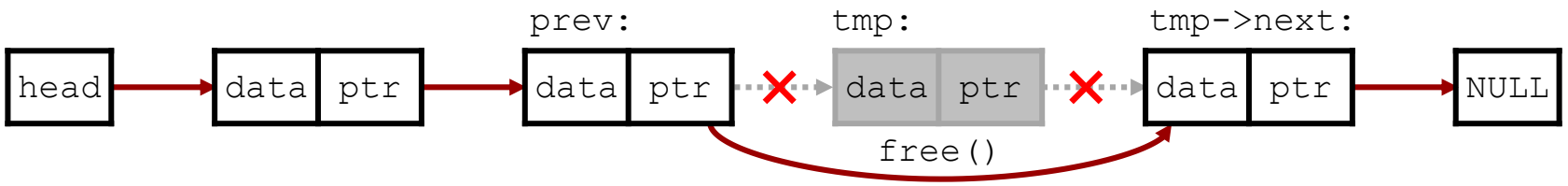
```
struct node *deleteTail(struct node *head) {
   if (head == NULL) return NULL; // 빈 리스트
   if (head->next == NULL) { // 하나의 노드만 있을 경우
      free(head);
      return NULL;
   }
   struct node* second_last = head;
   while (second_last->next->next != NULL)
      second_last = second_last->next;
   free(second_last->next);
   second_last->next = NULL;
   return head;
}
```



연결리스트(linked list)

- 단일 연결리스
  - 리스트의 특정 노드 삭제
    - tmp 포인터를 생성하여 삭제 위치로 이동시킴
    - prev는 tmp의 한 칸 뒤 포인터임
    - tmp가 가리키는 노드를 삭제함

```
prev->next = tmp->next;
free(tmp);
```



연결리스트(linked list)

- 단일 연결리스
  - 리스트의 특정 노드 삭제

```
struct node *deletePos(struct node *head, int pos) {
   struct node *tmp = head, *prev = NULL;
   if (temp == NULL) return head; // 빈 리스트
   if (pos == 1) { // 리스트의 처음 노드 삭제
      head = tmp->next;
      free(tmp);
      return head;
   }
   for (int k=1; tmp != NULL && k < pos; k++) {
      prev = tmp;
      tmp = tmp->next;
   }
   if (tmp != NULL) {
      prev->next = tmp->next;
      free(tmp);
   } else {
      printf("Node not found!\n");
   }
   return head;
}
```

형 정의

- 새로운 데이터형의 이름을 만들고 싶다면 typedef을 사용할 수 있음
  - 아래 2개는 **struct** node 구조체의 이름을 node로 침

```
struct node {
    int data;
    struct node *next;
};
```

```
typedef struct node {
    int data;
    struct node *next;
} node;
```

```
typedef struct node node;
```

  - node 구조체를 생성하고 싶다면 다음과 같이 하면 됨

```
node *head = createNode(10);
```

연결리스트(linked list)

- 이중 연결리스

```
typedef struct node {
    int data;
    struct node *next;
    struct node *prev;
} node;
```

- 연산
  - 삽입(insertion)
    - 리스트 처음에 삽입
    - 리스트 끝에 삽입
    - 특정 위치에 삽입
  - 삭제(deletion)
    - 리스트의 처음 노드 삭제
    - 리스트의 끝 노드 삭제
    - 리스트의 특정 노드 삭제
  - 종주(traversal)

연결리스트 연산		시간 복잡도	공간 복잡도
삽입 (insertion)	처음에	O(1)	O(1)
	끝에	O(N)	O(1)
	특정 위치에	O(N)	O(1)
삭제 (deletion)	처음 노드	O(1)	O(1)
	끝 노느	O(N)	O(1)
	특정 노드	O(N)	O(1)
종주(traversal)		O(N)	O(1)

연결리스트(linked list)

- 이중 연결리스
  - 노드 만들기

```
node *createNode(int data) {
   node *newNode = (node*)malloc(sizeof(node));
   newNode->data = data;
   newNode->next = NULL;
   newNode->prev = NULL;
   return newNode;
}
```

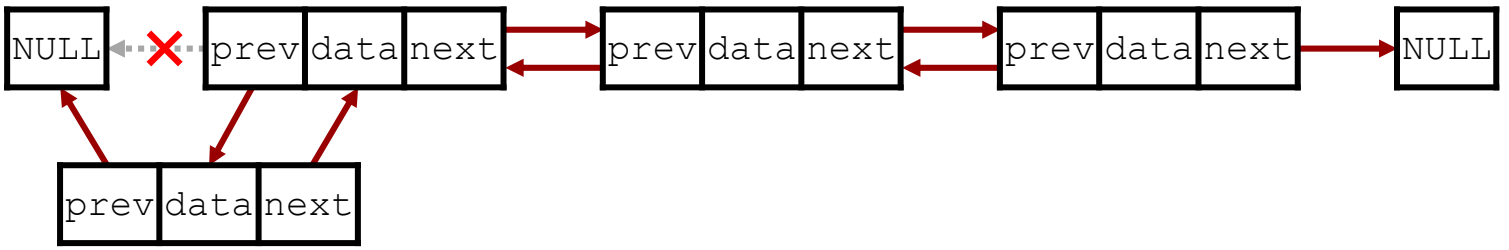
  - 리스트 종주

```
void traversalForward(node *head) {
   while (head != NULL) {
      printf("%d ", head->data);
      head = head->next;
   }
   printf("\n");
}
```

연결리스트(linked list)

- 이중 연결리스
  - 리스트 처음에 삽입

```
void insertFront(node **head, int data) {
    node *newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    newNode->next = *head;
    (*head)->prev = newNode;
    *head = newNode;
}
```



연결리스트(linked list)

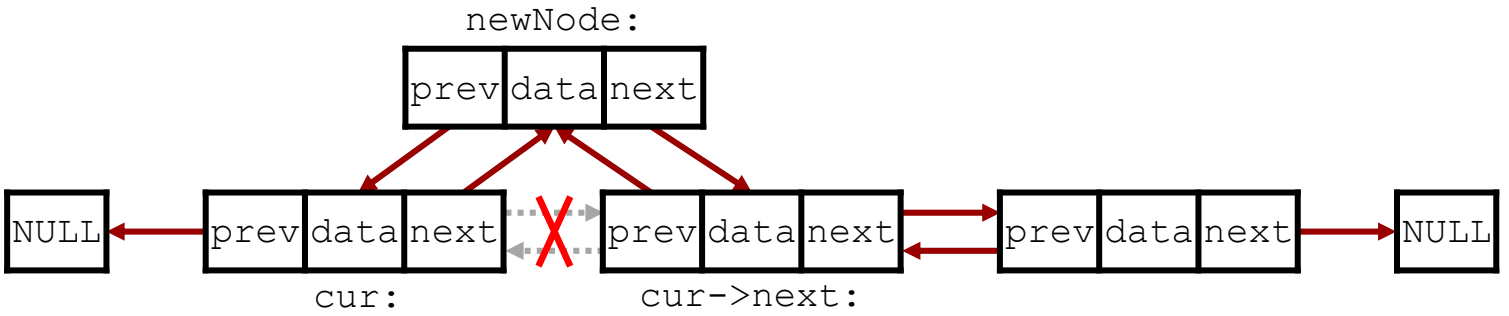
- 이중 연결리스
  - 리스트 끝에 삽입

```
void insertTail(node **head, int data) {
    node *newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    node *cur = *head;
    while (cur->next != NULL)
        cur = cur->next;
    cur->next = newNode;
    newNode->prev = cur;
}
```

연결리스트(linked list)

- 이중 연결리스
  - 리스트 특정 위치에 삽입
    - cur 포인터를 생성하여 삽입 위치로 이동시킴
    - newNode를 삽입함

```
newNode->next = cur->next;
newNode->prev = cur;
cur->next->prev = newNode;
cur->next = newNode;
```



연결리스트(linked list)

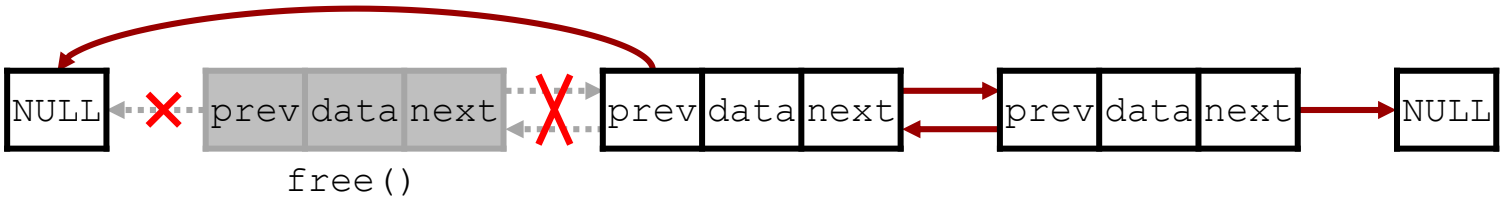
- 이중 연결리스
  - 리스트 특정 위치에 삽입

```
void insertPos(node **head, int data, int pos) {
   if (pos < 1) return; // 유효하지 않는 위치
   if (pos == 1) { // 리스트 처음에 삽입
      insertFront(head, data);
      return;
   }
   node *newNode = createNode(data);
   node *cur = *head;
   for (int k=1; cur != NULL && k < pos-1; k++)
      cur = cur->next;
   if (cur == NULL) return; // 삽입 위치는 리스트의 길이보다 큼
   newNode->next = cur->next;
   newNode->prev = cur;
   if (cur->next != NULL) cur->next->prev = newNode;
   cur->next = newNode;
}
```

연결리스트(linked list)

- 이중 연결리스
  - 리스트의 처음 노드 삭제

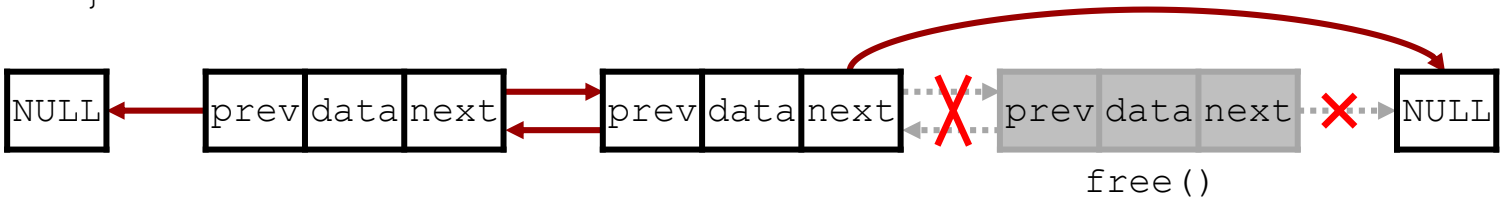
```
void deleteFront(node **head) {
    if (*head == NULL) return; // 빈 리스트
    node *cur = *head;
    *head = (*head)->next;
    if (*head != NULL)
        (*head)->prev = NULL;
    free(cur);
}
```



연결리스트(linked list)

- 이중 연결리스
  - 리스트의 끝 노드 삭제

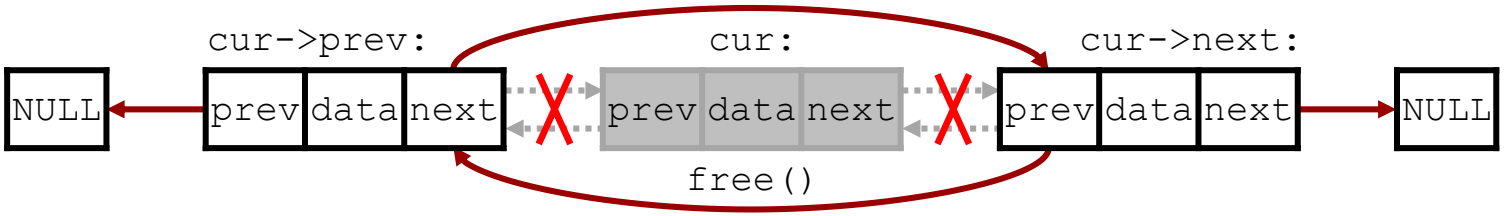
```
void deleteTail(node **head) {
    if (*head == NULL) return; // 빈 리스트
    node *cur = *head;
    if (cur->next == NULL) { // 하나의 노드만 있을 경우
        *head = NULL;
        free(cur);
        return;
    }
    while (cur->next != NULL)
        cur = cur->next;
    cur->prev->next = NULL;
    free(cur);
}
```



연결리스트(linked list)

- 이중 연결리스
  - 리스트의 특정 노드 삭제
    - cur 포인터를 생성하여 삭제 위치로 이동시킴
    - cur가 가리키는 노드를 삭제함

```
cur->next->prev = cur->prev;
cur->prev->next = cur->next;
free(cur);
```



연결리스트(linked list)

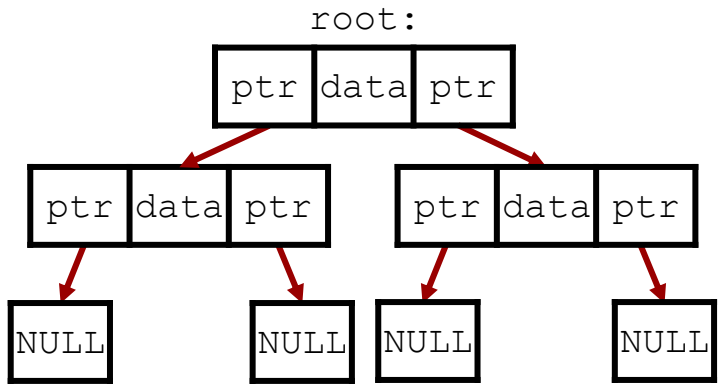
- 이중 연결리스
  - 리스트의 특정 노드 삭제

```
void deletePos(node **head, int pos) {
   if (*head == NULL) return; // 빈 리스트
   node *cur = *head;
   if (pos == 1) { // 리스트의 처음 노드 삭제
      deleteFront(head);
      return;
   }
   for (int k=1; cur != NULL && k < pos; k++)
      cur = cur->next;
   if (cur == NULL) return; // 삭제 위치는 리스트의 길이보다 큼
   if (cur->next != NULL)
      cur->next->prev = cur->prev;
   if (cur->prev != NULL)
      cur->prev->next = cur->next;
   free(cur);
}
```

이진 트리(binary tree)

- 트리는 부모-자식 관계를 갖고 있는 노드들로 이루어진 구조체이며, 각 노드는 다음과 같은 정보를 가짐
  - 데이터
  - 왼쪽 노드에 대한 포인터
  - 오른쪽 노드에 대한 포인터

```
typedef struct node {
    int data;
    struct node *left;
    struct node *right;
} node;
```



이진 트리(binary tree)

- 트리에 자주 하는 연산
  - 삽입(insertion)
  - 삭제(deletion)
  - 탐색(search)
  - 종주(traversal)

트리 연산	시간 복잡도	공간 복잡도
삽입	O(logN)	O(N)
삭제	O(logN)	O(N)
탐색	O(logN)	O(N)
종주	O(N)	O(1)

이진 트리(binary tree)

- 노드를 만들기

```
node *createNode(int data) {
    node *newNode = (node*)malloc(sizeof(node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

- 트리 종주

```
void traversal(node *root) {
    if (root == NULL) return; // 빈 트리
    traversal(root->left);
    printf("%d ", root->data);
    traversal(root->right);
}
```

이진 트리(binary tree)

- 트리에 삽입

```
node *insert(node *root, int data) {
    if (root == NULL)
        return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > node->data)
        root->right = insert(root->right, data);
    return node;
}
```

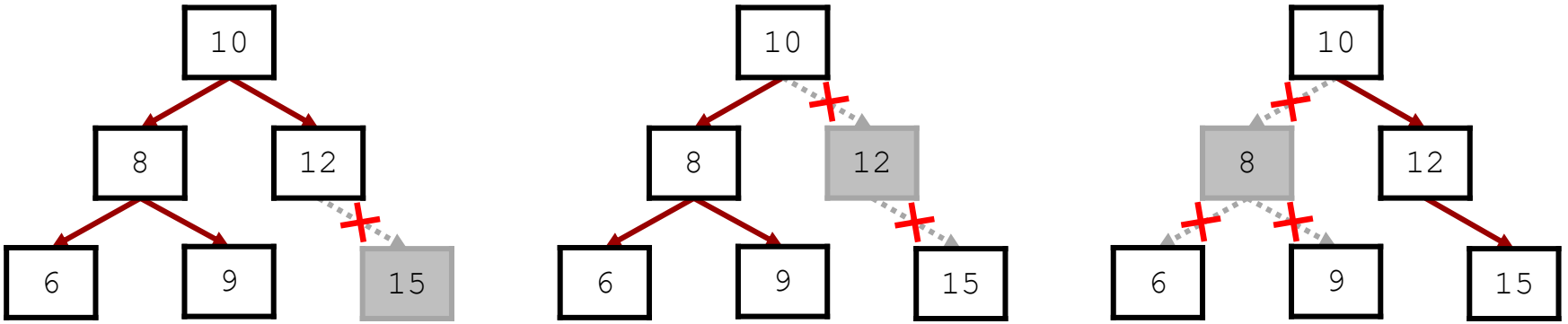
이진 트리(binary tree)

- 트리에 삽입

```
node *insert(node *root, int data) {
    if (root == NULL)
        return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > node->data)
        root->right = insert(root->right, data);
    return node;
}
```

이진 트리(binary tree)

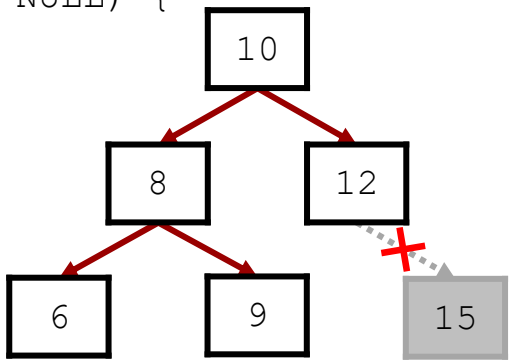
- 트리의 노드 삭제
  - 삭제할 노드는 자식이 없는 노드
  - 삭제할 노드는 하나의 자식이 있는 노드
  - 삭제할 노드는 2개의 자식이 있는 노드



이진 트리(binary tree)

- 트리의 노드 삭제
  - 삭제할 노드는 자식이 없는 노드

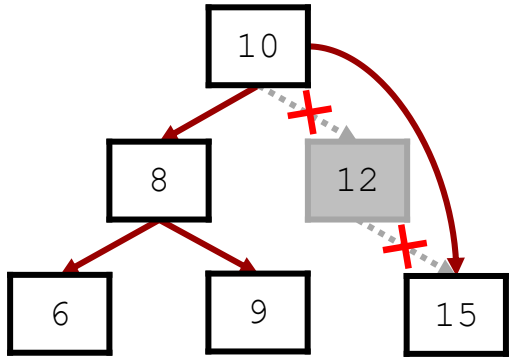
```
node *delete(node *root, int data) {
    if (root == NULL) return root; // 빈 트리
    if (data > root->data)
        root->right = delete(root->right, data);
    else if (data < root->data)
        root->left = delete(root->left, data);
    else {
        // 자식이 없는 노드 삭제
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        }
        else ...
    }
}
```



이진 트리(binary tree)

- 트리의 노드 삭제
  - 삭제할 노드는 하나의 자식이 있는 노드

```
node *delete(node *root, int data) {
    ...
        // 하나의 자식이 있는 노드
        else if (root->left == NULL || root->right == NULL) {
            node *cur;
            if (root->left == NULL) cur = root->right;
            else cur = root->left;
            free(root);
            return cur;
        }
        else ...
    }
}
```



이진 트리(binary tree)

- 트리의 노드 삭제
  - 삭제할 노드는 2개의 자식이 있는 노드

```
node *delete(node *root, int data) {
    ...
        // 2개의 자식이 있는 노드
        else {
            node *cur = findMin(root->right);
            root->data = cur->data;
            root->right = delete(root->right, cur->data);
        }
    }
    return root;
}
```

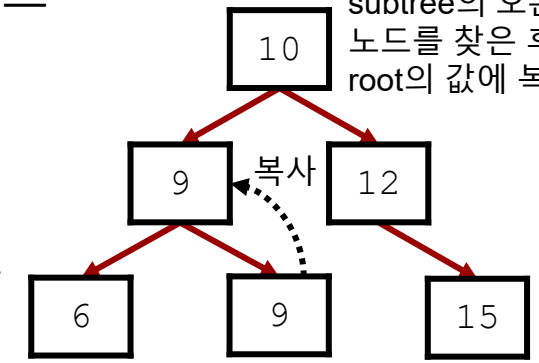
findMin() 함수는 매개변수로 전달되는 트리의 최솟값을 갖는 노드를 리턴함

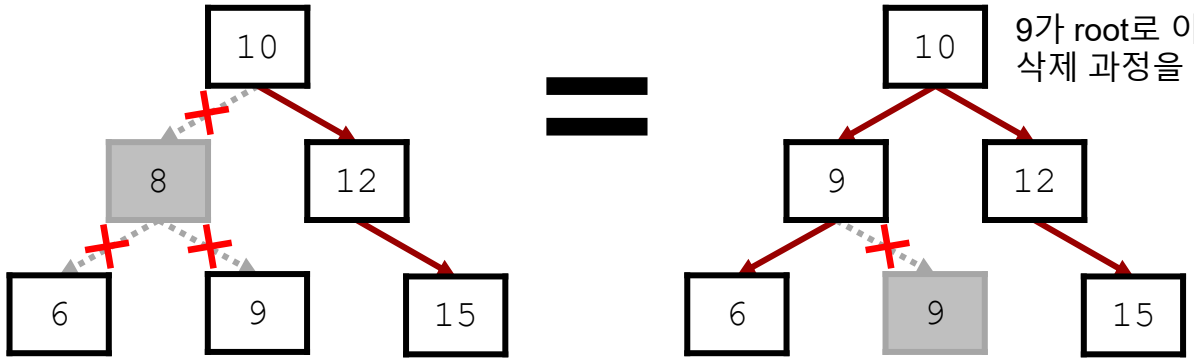
이진 트리(binary tree)

- 트리의 노드 삭제
  - 삭제할 노드는 2개의 자식이 있는 노드

```
node *findMin(node *root) {
    if (root == NULL)
        return NULL;
    else if (root->left != NULL)
        return findMin(root->left);
    return root;
}
```

삭제하려는 8이 root로 이루어진 subtree의 오른 쪽 최숫값을 갖는 노드를 찾은 후 그 노드의 값을 root의 값에 복사함





9가 root로 이루어진 subtree에 삭제 과정을 반복함

이진 트리(binary tree)

- 트리의 노드 삭제
  - 트리 탐색

```
node *search(node *root, int data) {
   if (root == NULL || root->data == data)
      return root;
   if (root->data < data)
      return search(root->right, data);
   return search(root->left, data);
}
```

유니언(union)

- 프로그램이 기계에 종속되지 않으면서도 한 기억장소 내에서 서로 다른 데이터형을 처리할 수 있음
- 유니언의 멤버를 다음과 같이 사용할 수 있음

  유니언 이름.멤버

  유니언의 포인터->멤버
- **struct**와 달리 한 번에 하나의 멤버만 사용할 수 있음

```
// 유니언 정의 및 변수 선언
union utype {
    int d;
    float f;
    char *msg;
} tmp;
```

```
// 유니언 정의 및 변수 선언
union utype {
    int d;
    float f;
    char *msg;
};
union utype tmp;
```

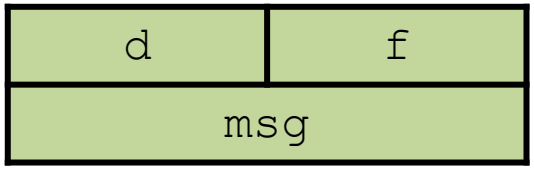
유니언(union)

- **struct**와 달리 한 번에 하나의 멤버만 사용할 수 있음

```
struct stype {
    int d;
    float f;
    char *msg;
} tmp;
```

```
union utype {
    int d;
    float f;
    char *msg;
} tmp;
```

16 bytes



8 bytes

크기가 제일 큰 멤버가 들어갈 수 있는 만큼 메모리는 할당됨

유니언(union)

- 유니언은 구조체나 배열 안에 나타날 수가 있고 또한 그 반대로도 됨

```
union student {
    int id;
    union grade {
        float mark;
    } gpa;
} u;
u.gpa.mark = 3.63;
```

```
struct employee {
    int age;
    char *name;
    char *position;
    union { // 이름 생략 가능
        int monthly;
        int bonus;
        float factor;
    } salary;
} em[100];
em[11].salary.factor = 3.0;
```

비트 필드(bit fields)

- 구조체 멤버의 크기는 비트 단위로 설정할 수 있음
  - 기억장소가 부족한 시스템에 유용함
  - 프로그램의 크기를 줄이고 처리 속도를 향상시킬 수 있음
- 문법

```
struct 이름 {
    데이터형 멤버이름 : 비트크기;
};
```

```c
struct A {
   unsigned int x; // 0~2^32-1
   unsigned int y; // 0~2^32-1
};

// 8 bytes
printf("%ld\n", sizeof(struct A));
```

```c
struct B {
   unsigned int x : 4; // 0~15
   unsigned int y : 3; // 0~7
};

// 4 bytes
printf("%ld\n", sizeof(struct B));
```

비트 필드(bit fields)

- 데이터형보다 더 많은 비트를 할당할 수 없음

```
struct A {
    int x : 10;
    int y : 35; // 35는 sizeof(int)보다 커서 컴파일 시 에러 발생
};
```

- 비트 필드는 0로 설정할 경우 다음 경계 부분에 맞추어 바이트를 정렬해줌

```
struct A {
   unsigned int x : 5;
   unsigned int y : 8;
};

// 4 bytes
printf("%ld\n", sizeof(struct A));
```

```
struct B {
   unsigned int x : 5;
   unsigned int : 0;
   unsigned int y : 8;
};

// 8 bytes
printf("%ld\n", sizeof(struct B));
```

비트 필드(bit fields)

- 여러 데이터형을 사용할 경우 크기는 가장 큰 데이터형의 크기와 같음

```
struct A {
    unsigned int x : 6;
    unsigned long y : 16;
    int z;
} tmp;
printf("%ld\n", sizeof(struct A)); // 8 bytes
```

- 비트 필드에 주소 연산자(&)를 사용할 수 없음

```
unsigned int *p1 = &tmp.x; // 에러
int *p2 = &z; // 정상
```

비트 필드(bit fields)

- 비트 필드의 배열이 선언될 수 없음

```
struct A {
    unsigned int x[10] : 6; // 에러
    unsigned long y : 16;
    int z;
};
```