자료구조 실습

11/05

김민서 (imkmsh@khu.ac.kr)

목차

- Array List (구현 X)
- Linked List
  - 동작 이론
  - Head에 node 추가
  - List 중간에 node 추가
  - List 내에서 node 제거
  - 인덱스를 이용한 데이터 조회
- List 구현 방법에 따른 시간 차이 확인

김민서 (imkmsh@khu.ac.kr)

Array List - main.hpp

```c
#define AL_CAP 1000                  // 배열 용량을 상수로 정의

typedef struct {                     // 배열 리스트를 나타내는 구조체
    int list[AL_CAP];                    // 실제 데이터를 담을 배열
    int n;                           // 현재 저장된 개수
} ArrayList;

void al_init(ArrayList* L);                          // 리스트 초기화
int  al_is_empty(const ArrayList* L);                // 비었는지 확인
int  al_size(const ArrayList* L);                    // 크기 반환
void al_clear(ArrayList* L);                         // 비우기
int  al_append(ArrayList* L, int x);                 // 뒤에 추가(성공1/실패0)
int  al_remove(ArrayList* L, int x);                 // 값 x 첫 1개 삭제
int  al_get(const ArrayList* L, int idx);            // idx번째 값(잘못되면 -1)
void al_print(const ArrayList* L, const char* tag); // 내용 출력
```

김민서 (imkmsh@khu.ac.kr)

Array List

```c
void al_init(ArrayList* L){ L->n = 0; }          // 개수를 0으로 두어 초기화
int  al_is_empty(const ArrayList* L){ return (L->n == 0); } // 비었으면 1, 아니면 0
int  al_size(const ArrayList* L){ return L->n; }             // 현재 개수 반환
void al_clear(ArrayList* L){ L->n = 0; }          // 개수를 0으로 만들어 비우기

int  al_append(ArrayList* L, int x){              // 뒤에 값 x 추가
    if(L->n >= AL_CAP) return 0;                  // 용량을 넘으면 실패
    L->list[L->n] = x;                               // 뒤 위치에 값 저장
    L->n = L->n + 1;                              // 개수 1 증가
    return 1;                                     // 성공 반환
}

int  al_remove(ArrayList* L, int x){              // 값 x 첫 1개 삭제
    int i = 0;                                    // i는 탐색 인덱스
    while(i < L->n && L->list[i] != x) i = i + 1;  // x를 찾을 때까지 전진
    if(i == L->n) return 0;                       // 못 찾았으면 실패
    int j = i + 1;                                // j는 한 칸 뒤부터 시작
    while(j < L->n){ L->list[j-1] = L->list[j]; j = j + 1; } // 뒤를 한 칸씩 당김
    L->n = L->n - 1;                              // 개수 1 감소
    return 1;                                     // 성공 반환
}
```

김민서 (imkmsh@khu.ac.kr)

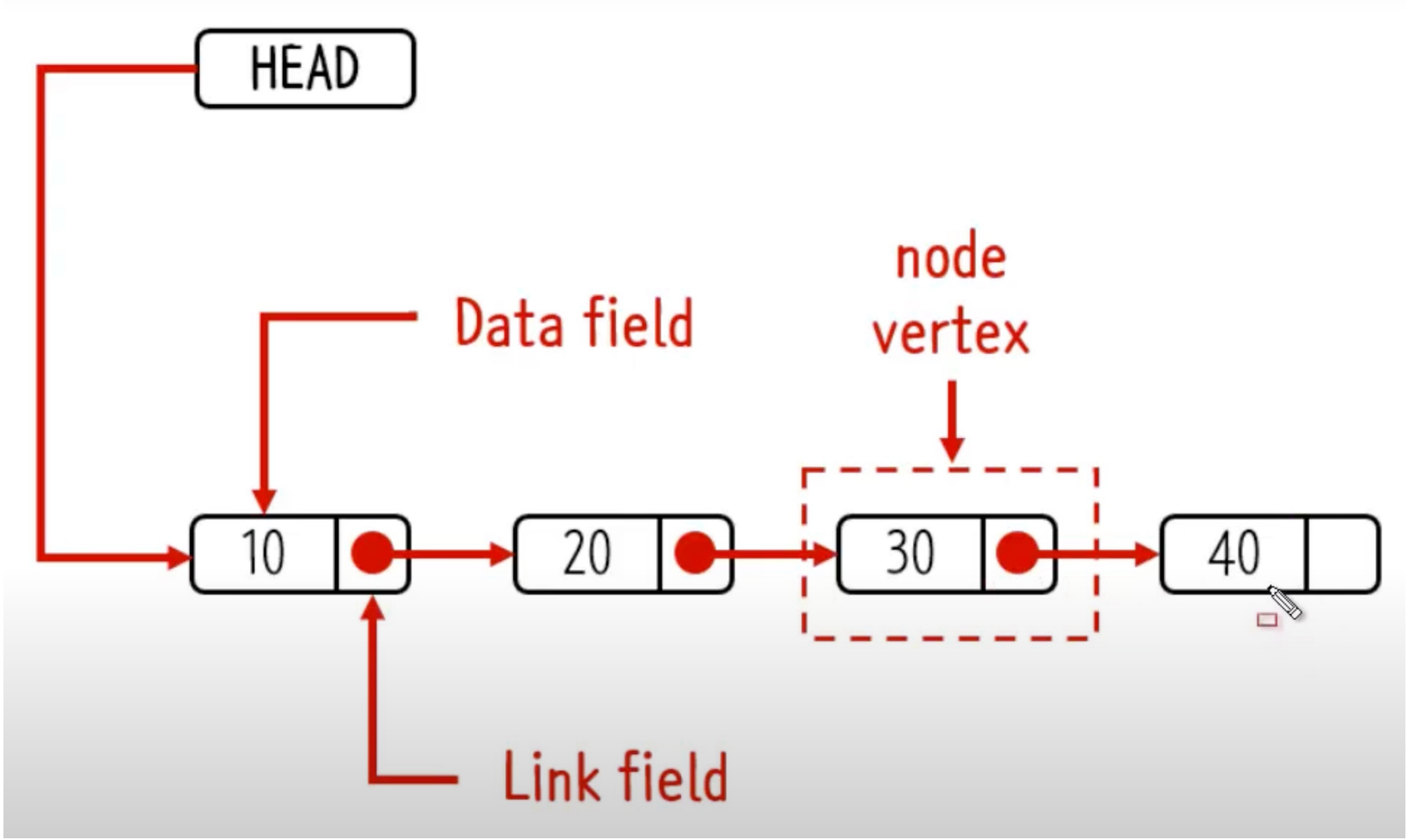
Array List

```c
int  al_get(const ArrayList* L, int idx){          // idx번째 값
    if(idx < 0 || idx >= L->n) return -1;          // 잘못된 인덱스면 -1
    return L->list[idx];                               // 올바르면 해당 값 반환
}

void al_print(const ArrayList* L, const char* tag){ // 내용 출력
    printf("[%s] ", tag);                          // 태그 출력
    int i = 0;                                     // i는 인덱스
    while(i < L->n){                               // 전체 반복
        printf("%d", L->list[i]);                      // 값 출력
        if(i + 1 < L->n) printf(" ");              // 다음 값이 있으면 공백
        i = i + 1;                                 // 다음으로 이동
    }
    printf(" (size=%d)\n", L->n);                  // 크기 출력
}
```

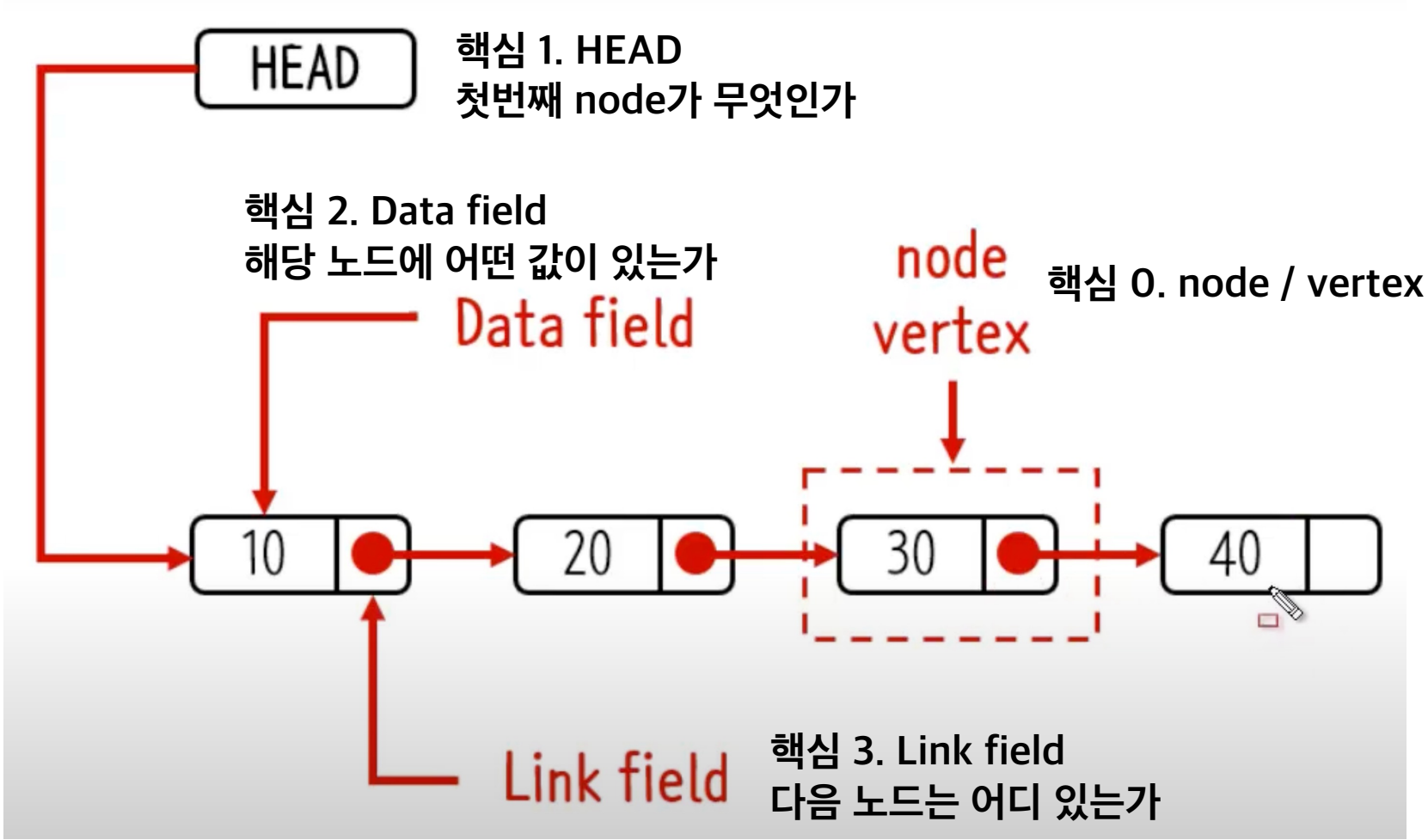
김민서 (imkmsh@khu.ac.kr)

Linked List - 동작 이론



김민서 (imkmsh@khu.ac.kr)

Linked List - 동작 이론



김민서 (imkmsh@khu.ac.kr)

Linked List

```c
typedef struct Node {              // 연결 리스트의 노드
    int x;                         // 데이터 값
    struct Node* next;             // 다음 노드를 가리키는 포인터
} Node;

typedef struct {                   // 연결 리스트를 나타내는 구조체
    Node* head;                    // 머리(첫 노드)를 가리키는 포인터
    int n;                         // 현재 저장된 개수
} LinkedList;

void ll_init(LinkedList* L);                      // 리스트 초기화
int  ll_is_empty(const LinkedList* L);            // 비었는지 확인
int  ll_size(const LinkedList* L);                // 크기 반환
void ll_clear(LinkedList* L);                     // 모두 삭제
void ll_push_front(LinkedList* L, int x);         // 머리에 삽입
void ll_insert(LinkedList* L, int idx, int x);    // idx 위치에 삽입
int  ll_remove(LinkedList* L, int x);             // 값 x 첫 1개 삭제
int  ll_get(const LinkedList* L, int idx);        // idx번째 값(잘못되면 -1)
void ll_print(const LinkedList* L, const char* tag); // 내용 출력
```

김민서 (imkmsh@khu.ac.kr)

Linked List

```c
int NEED_TO_SOLVE = 1;

void ll_init(LinkedList* L){ L->head = NULL; L->n = 0; } // head=NULL, 개수=0
int  ll_is_empty(const LinkedList* L){ return (L->n == NEED_TO_SOLVE); } // 비었으면 1
int  ll_size(const LinkedList* L){ return L->n; } // 현재 개수 반환
```

김민서 (imkmsh@khu.ac.kr)

Linked List

```
int NEED_TO_SOLVE = 1;

void ll_init(LinkedList* L){ L->head = NULL; L->n = 0; } // head=NULL, 개수=0
int  ll_is_empty(const LinkedList* L){ return (L->n == NEED_TO_SOLVE); } // 비었으면 1
int  ll_size(const LinkedList* L){ return L->n; } // 현재 개수 반환
```

```
void ll_init(LinkedList* L){ L->head = NULL; L->n = 0; } // Head=NULL, 개수=0
int  ll_is_empty(const LinkedList* L){ return (L->n == 0); } // 비었으면 1
int  ll_size(const LinkedList* L){ return L->n; } // 현재 개수 반환
```

김민서 (imkmsh@khu.ac.kr)

Linked List


```c
void ll_clear(LinkedList* L){                    // 모든 노드를 삭제
    Node* p = L->NEED_TO_SOLVE;                  // p는 현재 노드
    while(p != NULL){                            // p가 NULL이 아닐 동안
        Node* d = p;                             // d는 삭제할 노드
        p = NEED_TO_SOLVE;                       // p를 다음으로 이동
        free(d);                                 // d를 해제
    }
    L->head = NEED_TO_SOLVE;                     // head를 NULL로
    L->n = NEED_TO_SOLVE;                        // 개수 0으로
}
```

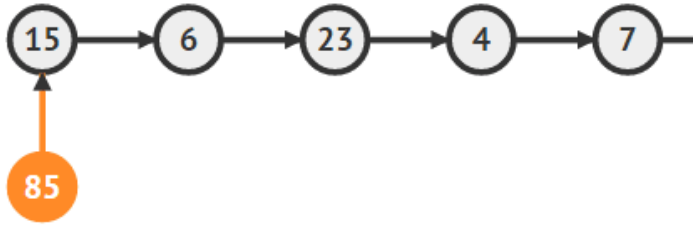
김민서 (imkmsh@khu.ac.kr)

Linked List


```c
void ll_clear(LinkedList* L){                    // 모든 노드를 삭제
    Node* p = L->head;                           // p는 현재 노드
    while(p != NULL){                            // p가 NULL이 아닐 동안
        Node* d = p;                             // d는 삭제할 노드
        p = p->next;                             // p를 다음으로 이동
        free(d);                                 // d를 해제
    }
    L->head = NULL;                              // Head를 NULL로
    L->n = 0;                                    // 개수 0으로
}
```

김민서 (imkmsh@khu.ac.kr)

Linked List

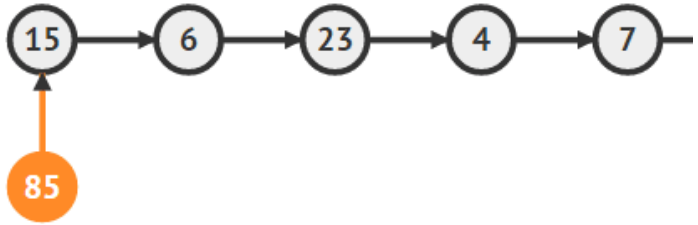



```c
void ll_insert(LinkedList* L, int idx, int x){      // idx 위치에 새 값 x 삽입
    if(idx < 0 || idx > L->n) return;               // 범위 벗어나면 무시
    Node* node = (Node*)malloc(sizeof(Node));       // 새 노드 생성
    if(node == NULL) return;                        // 메모리 부족 시 종료
    node->x = NEED_TO_SOLVE;                        // 새 노드에 값 저장

    // (1) 맨 앞(head) 삽입일 경우
    if(idx == 0){
        node->next = NEED_TO_SOLVE;                 // 새 노드가 기존 head를 가리킴
        L->head = NEED_TO_SOLVE;                    // head를 새 노드로 변경
        L->n = NEED_TO_SOLVE;                       // 개수 증가
        // 생각해 보기: L->n = n + 1;
        return;
    }
```

김민서 (imkmsh@khu.ac.kr)

Linked List




```c
void ll_insert(LinkedList* L, int idx, int x){      // idx 위치에 새 값 x 삽입
    if(idx < 0 || idx > L->n) return;               // 범위 벗어나면 무시
    Node* node = (Node*)malloc(sizeof(Node));       // 새 노드 생성
    if(node == NULL) return;                        // 메모리 부족 시 종료
    node->x = x;                                    // 새 노드에 값 저장

    // (1) 맨 앞(Head) 삽입일 경우
    if(idx == 0){
        node->next = L->head;                       // 새 노드가 기존 Head를 가리킴
        L->head = node;                             // Head를 새 노드로 변경
        L->n = L->n + 1;                            // 개수 증가
        return;
    }
```

김민서 (imkmsh@khu.ac.kr)

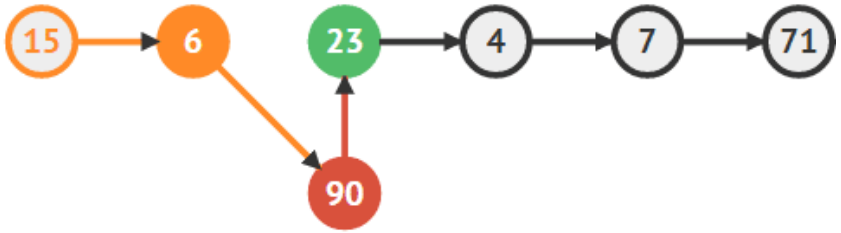
Linked List - 생각해 보기 (1)

```c
void ll_insert(LinkedList* L, int idx, int x){      // idx 위치에 새 값 x 삽입
    if(idx < 0 || idx > L->n) return;               // 범위 벗어나면 무시
    Node* node = (Node*)malloc(sizeof(Node));       // 새 노드 생성
    if(node == NULL) return;                        // 메모리 부족 시 종료
    node->x = x;                                    // 새 노드에 값 저장

    // (1) 맨 앞(Head) 삽입일 경우
    if(idx == 0){
        node->next = L->head;                       // 새 노드가 기존 Head를 가리킴
        L->head = node;                             // Head를 새 노드로 변경
        L->n = L->n + 1;                            // 개수 증가
        return;
    }
```

- L->n = L->n+1; VS L->n = n+1;
- L은 **LinkedList 구조체를 가리키는 포인터** (구조체 자체가 아니라 주소)
- 우리가 바꾸고 싶은 건, 이 포인터가 **가리키고 있는 구조체 안의 멤버 n**

Linked List

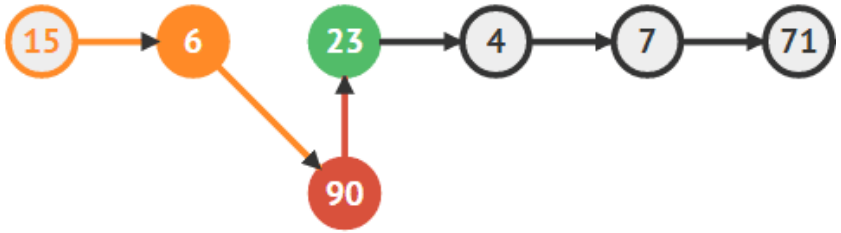



```c
void ll_insert(LinkedList* L, int idx, int x){      // idx 위치에 새 값 x 삽입
    if(idx < 0 || idx > L->n) return;               // 범위 벗어나면 무시
    Node* node = (Node*)malloc(sizeof(Node));       // 새 노드 생성
    if(node == NULL) return;                        // 메모리 부족 시 종료
    node->x = x;                                    // 새 노드에 값 저장

    // (2) 중간 혹은 맨 끝 삽입
    Node* prev = L->head;                           // 이전 노드 시작은 head
    int i = 0;
    while(i < idx - 1 && prev->next != NULL){       // 삽입 위치 직전까지 이동
        prev = NEED_TO_SOLVE;
        i = i + 1;
    }
    node->next = NEED_TO_SOLVE;                     // 새 노드가 이전 노드의 다음을 가리킴
    prev->next = NEED_TO_SOLVE;                     // 이전 노드가 새 노드를 가리키게 변경
    L->n = NEED_TO_SOLVE;                           // 개수 증가
}
```

김민서 (imkmsh@khu.ac.kr)

Linked List

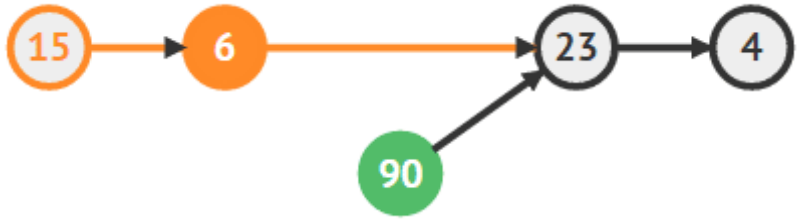



```c
void ll_insert(LinkedList* L, int idx, int x){      // idx 위치에 새 값 x 삽입
    if(idx < 0 || idx > L->n) return;               // 범위 벗어나면 무시
    Node* node = (Node*)malloc(sizeof(Node));       // 새 노드 생성
    if(node == NULL) return;                        // 메모리 부족 시 종료
    node->x = x;                                    // 새 노드에 값 저장

    // (2) 중간 혹은 맨 끝 삽입
    Node* prev = L->head;                           // 이전 노드 시작은 Head
    int i = 0;
    while(i < idx - 1 && prev->next != NULL){       // 삽입 위치 직전까지 이동
        prev = prev->next;
        i = i + 1;
    }
    node->next = prev->next;                        // 새 노드가 이전 노드의 다음을 가리킴
    prev->next = node;                              // 이전 노드가 새 노드를 가리키게 변경
    L->n = L->n + 1;                                // 개수 증가
}
```


김민서 (imkmsh@khu.ac.kr)

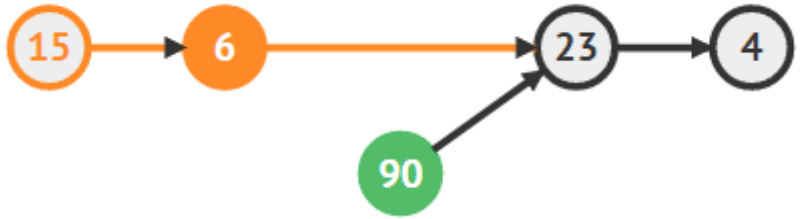
Linked List




```
int  ll_remove(LinkedList* L, int x){          // 값 x 첫 1개 삭제
    if(L->head == NULL) return 0;              // 비었으면 실패
    if(L->head->x == x){                       // head가 삭제 대상이면
        Node* d = L->head;                     // d는 삭제할 노드
        L->head = NEED_TO_SOLVE;               // head를 다음으로 이동
        free(d);                               // d 해제
        L->n = NEED_TO_SOLVE;                  // 개수 1 감소
        return 1;                              // 성공 반환
    }
    Node* prev = L->head;                      // prev는 이전 노드
    while(prev->next != NULL && prev->next->x != x) prev = NEED_TO_SOLVE; // 대상 찾기
    if(prev->next == NULL) return 0;           // 못 찾으면 실패
    Node* d = NEED_TO_SOLVE;                   // d는 삭제할 노드
    prev->next = NEED_TO_SOLVE;                // 이전 노드가 d를 건너뛰게 연결
    free(d);                                   // d 해제
    L->n = NEED_TO_SOLVE;                      // 개수 1 감소
    return 1;                                  // 성공 반환
    // 생각해 보기: 왜 자꾸 대상 노드가 Head일 때를 따로 처리해 주어야 할까?
}
```


김민서 (imkmsh@khu.ac.kr)

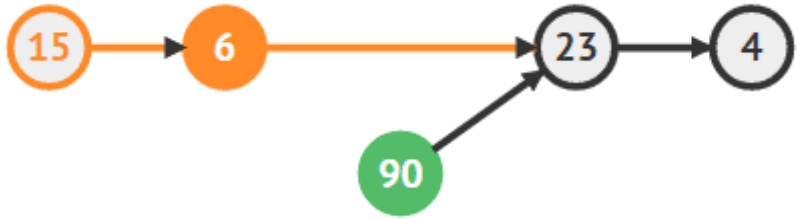
Linked List




```
int  ll_remove(LinkedList* L, int x){          // 값 x 첫 1개 삭제
    if(L->head == NULL) return 0;              // 비었으면 실패
    if(L->head->x == x){                       // Head가 삭제 대상이면
        Node* d = L->head;                     // d는 삭제할 노드
        L->head = L->head->next;               // Head를 다음으로 이동
        free(d);                               // d 해제
        L->n = L->n - 1;                       // 개수 1 감소
        return 1;                              // 성공 반환
    }
    Node* prev = L->head;                      // prev는 이전 노드
    while(prev->next != NULL && prev->next->x != x) prev = NEED_TO_SOLVE; // 대상 찾기
    if(prev->next == NULL) return 0;           // 못 찾으면 실패
    Node* d = NEED_TO_SOLVE;                   // d는 삭제할 노드
    prev->next = NEED_TO_SOLVE;                // 이전 노드가 d를 건너뛰게 연결
    free(d);                                   // d 해제
    L->n = NEED_TO_SOLVE;                      // 개수 1 감소
    return 1;                                  // 성공 반환
    // 생각해 보기: 왜 자꾸 대상 노드가 Head일 때를 따로 처리해 주어야 할까?
}
```


김민서 (imkmsh@khu.ac.kr)

Linked List




```
int  ll_remove(LinkedList* L, int x){          // 값 x 첫 1개 삭제
    if(L->head == NULL) return 0;              // 비었으면 실패
    if(L->head->x == x){                       // Head가 삭제 대상이면
        Node* d = L->head;                     // d는 삭제할 노드
        L->head = L->head->next;               // Head를 다음으로 이동
        free(d);                               // d 해제
        L->n = L->n - 1;                       // 개수 1 감소
        return 1;                              // 성공 반환
    }
    Node* prev = L->head;                      // prev는 이전 노드
    while(prev->next != NULL && prev->next->x != x) prev = prev->next; // 대상 찾기
    if(prev->next == NULL) return 0;           // 못 찾으면 실패
    Node* d = prev->next;                      // d는 삭제할 노드
    prev->next = d->next;                      // 이전 노드가 d를 건너뛰게 연결
    free(d);                                   // d 해제
    L->n = L->n - 1;                           // 개수 1 감소
    return 1;                                  // 성공 반환
    // 생각해 보기: 왜 자꾸 대상 노드가 Head일 때를 따로 처리해 주어야 할까?
}
```

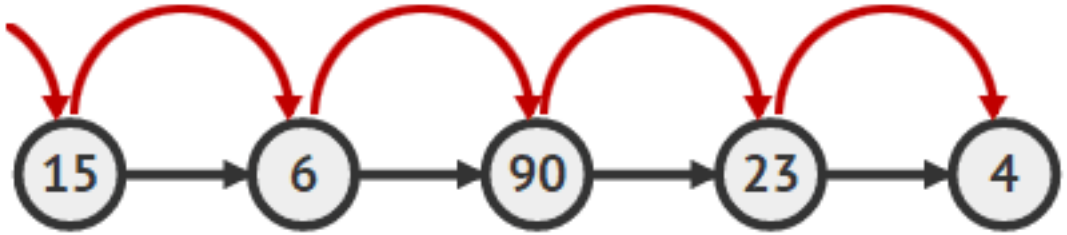

김민서 (imkmsh@khu.ac.kr)

Linked List - 생각해 보기 (2)

- 왜 대상 노드가 Head일 때를 따로 처리해 주어야 할까?
- 연결 리스트의 기본 구조
  - head → [data][next] → [data][next] → [data][next] → NULL
- 보통의 삭제나 삽입 로직은 “이전 노드”가 필요
  - prev->next = target->next; // 이전 노드가 target 다음 노드를 가리키게 변경
    free(target);
  - 첫 번째 노드(head)는 이전 노드가 없음
- 삽입할 때도 마찬가지
  - 중간 이후 노드 → 이전 노드(prev)가 있으니까 prev->next를 수정
  - 첫 번째 노드(head) → 이전 노드가 없으니까 head 자체를 바꿔야 함

김민서 (imkmsh@khu.ac.kr)

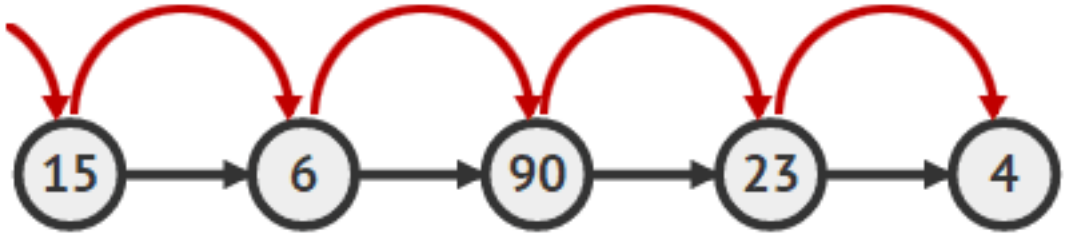
Linked List




```
int  ll_get(const LinkedList* L, int idx){      // idx번째 값
    if(idx < 0 || idx >= L->n) return -1;       // 잘못된 인덱스면 -1
    Node* p = L->head;                          // p는 현재 노드
    int i = 0;                                  // i는 현재 위치
    while(i < idx){ p = p->next; i = i + 1; }   // idx까지 이동
    return NEED_TO_SOLVE;                       // 해당 노드의 값 반환
}
```


김민서 (imkmsh@khu.ac.kr)

Linked List




```
int  ll_get(const LinkedList* L, int idx){      // idx번째 값
    if(idx < 0 || idx >= L->n) return -1;       // 잘못된 인덱스면 -1
    Node* p = L->head;                          // p는 현재 노드
    int i = 0;                                  // i는 현재 위치
    while(i < idx){ p = p->next; i = i + 1; }   // idx까지 이동
    return p->x;                                // 해당 노드의 값 반환
}
```


김민서 (imkmsh@khu.ac.kr)

List 구현 방법에 따른 시간 차이 확인

- main.cpp 실행

```
(base) minseo@minseo-MacBookAir-2 1105-linked-list % g++ -std=c++17 -O2 main.cpp answer_official.cpp -o run
./run

== Linked List vs Array ==
- 비교 항목: 추가/삭제(HEAD) / 인덱스 조회

[추가/삭제(Head)] Array = 47 ms, Linked = 7 ms
  해석: Array는 앞에서 넣고 빼려면 뒤 원소들을 '밀고/당겨야' 해서 느림(O(N)).
       Linked는 Head 포인터만 바꾸면 되므로 빠름(O(1)).

[인덱스 조회]   Array = 0 ms, Linked = 541 ms (sum 39990/502658476)
  해석: Array는 연속 메모리라 list[i]에 즉시 접근(O(1))이 가능.
       Linked는 i번째를 찾으려면 Head부터 i번 이동(O(N))이라 느림.
```

김민서 (imkmsh@khu.ac.kr)

수고하셨습니다

자료구조 실습 11/05

EOF

김민서 (imkmsh@khu.ac.kr)