C – 자료구조 2

Data Structure

자료 구조

- **자료 구조**

  자료구조는 프로그래밍에서 데이터를 효율적으로 저장하고, 접근하고, 관리하기 위한 구조를 의미합니다.

자료 구조	특징	C 언어 구현 방법
스택(Stack)	LIFO(후입선출)	배열 또는 구조체
큐(Queue)	FIFO(선입선출)	배열, 원형 큐, 구조체
연결 리스트	동적 크기, 삽입/삭제 용이	단순, 이중, 원형 연결 리스트
트리	계층 구조(비선형 구조)	구조체, 연결리스트
해시	해시 함수로 변환	배열 및 연결리스트

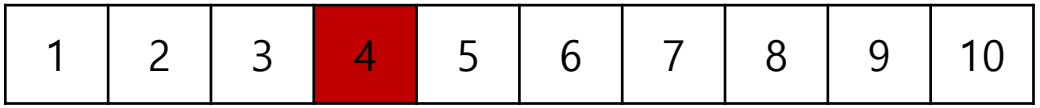
연결 리스트(Linked List)

- 연결 리스트(Linked List)의 필요성

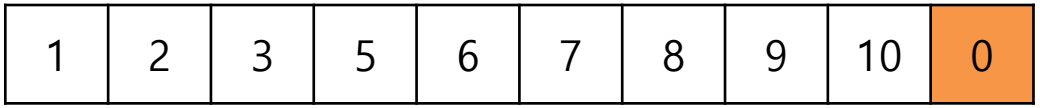
✓ 배열 자료구조의 문제점

배열에서 특정 요소를 삽입하거나 삭제하려면 빈 방을 왼쪽으로 이동하지 않으면, 빈 방을 찾아야하는 번거로움이 있다.

이러한 문제를 보완하는 자료 구조가 연결리스트이다.



3번방 삭제 -> 왼쪽으로 이동



연결 리스트(Linked List)

- 배열에서 요소 삭제

```c
//학생 구조체 정의
typedef struct {
    int num;  //번호
}Student;

int main()
{
    Student st[10]; //구조체 배열 생성
    int i;

    for (i = 0; i < 10; i++) {
        st[i].num = i + 1;  //요소 저장
    }

    for (i = 0; i < 10; i++) {
        printf("%d ", st[i].num);  //요소 출력
    }
    printf("\n");
```

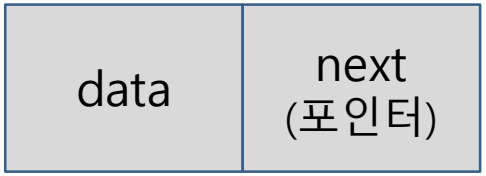
연결 리스트

- 배열에서 요소 삭제

```
    printf("4번 학생 전학\n");
    st[3].num = 0;  //요소 삭제

    printf("방을 왼쪽으로 이동\n");
    for (i = 3; i < 9; i++) {
        st[i].num = st[i + 1].num;  //한 칸 왼쪽으로 밀기
    }
    st[9].num = 0;

    for (i = 0; i < 10; i++) {
        printf("%d ", st[i].num);  //1 2 3 5 6 7 8 9 10 0
    }

    return 0;
}
```

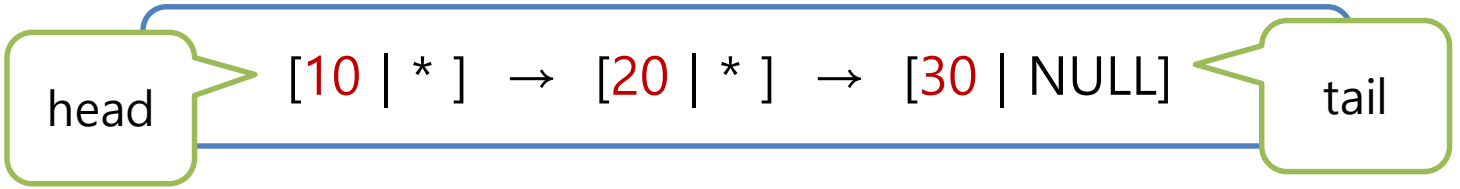
연결 리스트

- 연결 리스트(Linked List)란?
  - 데이터를 연속된 메모리에 저장하는 배열과 달리, **노드(node)**라는 독립적인 메모리 블록들이 포인터로 연결된 자료 구조이다
  - 각 노드는 데이터 + 다음 노드의 주소를 저장합니다.



노드(node)

- data: 노드가 가지는 데이터 값
- next: 포인터, 다음 노드의 주소 저장

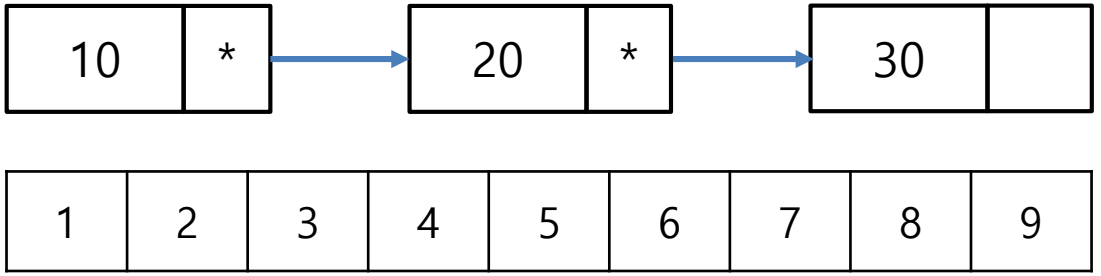


✓ 각 노드가 자신의 데이터와 다음 노드의 주소를 저장.

✓ NULL이 나오면 리스트의 끝.

연결 리스트(Linked List)

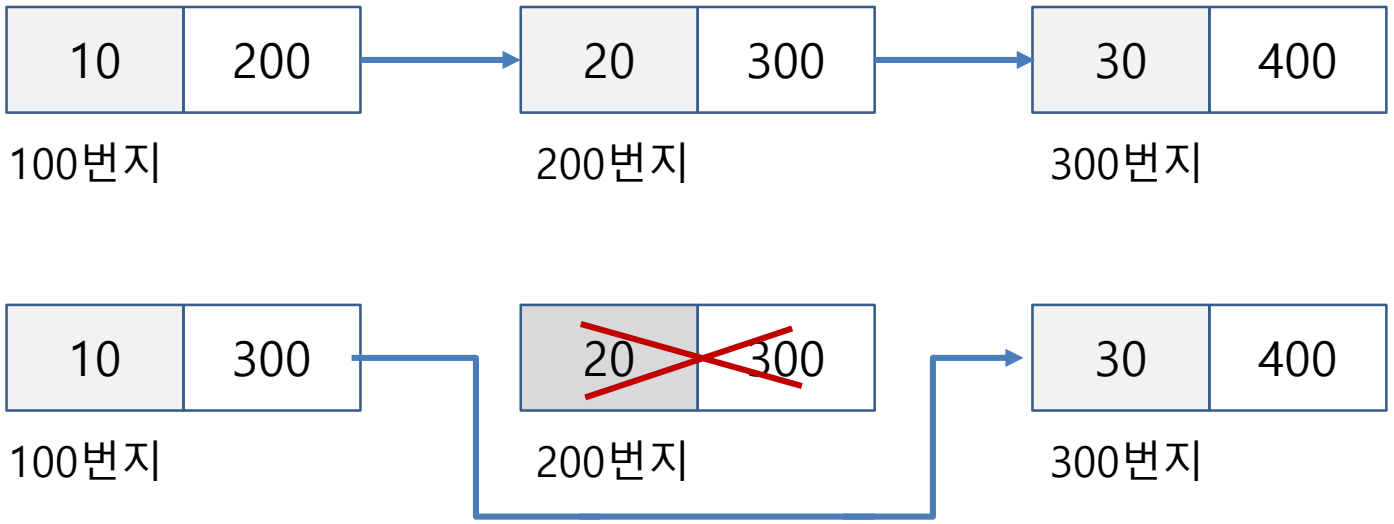
- 배열과 연결 리스트의 비교



비교 항목	배열	연결 리스트
메모리 구조	연속된 공간에 저장	흩어져 있는 노드를 포인터로 연결
크기 변경	어렵다 (재할당 필요)	자유롭게 추가/삭제 가능
접근 방법	random access 가능	random access 불가능
접근 속도	O(1) (인덱스로 접근)	O(n) (처음부터 순차 탐색)
삽입/삭제	느림 (데이터 이동 필요) O(n) 시간 소요	빠름 (포인터만 수정)

연결 리스트

- 노드 연결 및 삭제



연결 리스트

- 연결 리스트 구현

```c
typedef struct{
    int data;
    struct List* next;  //자기 참조 구조체
}List;

int main()
{
    List x, y, z; //노드 생성

    //요소 저장
    x.data = 10;
    y.data = 20;
    z.data = 30;

    //노드 연결
    x.next = &y;    //x -> y
    y.next = &z;    //y -> z
    z.next = NULL; //z는 마지막 노드
```

연결 리스트

- 연결 리스트(Linked List) 구현

```
//노드 출력
List* p;
p = &x;

printf("%d %x\n", x.data, p->next); //첫번째 노드 출력
p = p->next;
printf("%d %x\n", y.data, p->next); //두번째 노드 출력

//전체 노드 출력
for (p = &x; p != NULL; p = p->next) {
    printf("%d ", p->data);
}

printf("\n*** 노드 y 삭제 후 ***\n");
x.next = y.next; //x -> z(연결)
y.next = NULL; //y는 연결에서 제외

for (p = &x; p != NULL; p = p->next) {
    printf("%d ", p->data);
}
```

```
10 3beff868
20 3beff898
10 20 30
*** 노드 y 삭제 후 ***
10 30
```

연결 리스트

- 내 친구를 리스트로 저장하기

```c
typedef struct{
    int age;
    char name[20];
    struct Person* next;  //자기 참조 구조체
}Person;

int main()
{
    //노드 생성 및 초기화
    Person a = { 33, "손흥민" };
    Person b = { 21, "신유빈" };
    Person c = { 22, "임시현" };
    Person d = { 26, "이정후" };

    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = NULL;
```

연결 리스트

- 내 친구를 리스트로 저장하기

```
Person* p; //구조체 포인터
p = &a;

//첫째 노드 출력
//printf("%d %s\n", p->age, p->name); //33 손흥민

//노드 순회 및 출력
for (p = &a; p != NULL; p = p->next) {
    printf("%d %s\n", p->age, p->name);
}

printf("*** 노드 c 삭제 후 ***\n");
b.next = c.next; //b -> d(연결)
c.next = NULL;

//노드 순회 및 출력
for (p = &a; p != NULL; p = p->next) {
    printf("%d %s\n", p->age, p->name);
}
```

```
33 손흥민
21 신유빈
22 임시현
26 이정후
*** 노드 c 삭제 후 ***
33 손흥민
21 신유빈
26 이정후
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트

```c
// 노드 구조 정의
typedef struct{
    int data;            // 노드가 저장하는 값
    struct Node* next;   // 다음 노드의 주소(자기 참조)
} Node;

int main() {
    // 노드 3개 생성 - 동적 할당(힙 메모리 영역)
    Node* head, * second, * third;

    head = (Node*)malloc(sizeof(Node));  //첫 노드
    second = (Node*)malloc(sizeof(Node));
    third = (Node*)malloc(sizeof(Node));

    // 데이터 저장
    head->data = 10;
    head->next = second;
```

연결 리스트

- 동적 메모리 기반 단순 연결 리스트

```
second->data = 20;
second->next = third;

third->data = 30;
third->next = NULL; // 마지막 노드

// 출력
Node* current = head;
while (current != NULL) {
    printf("%d -> ", current->data);
    current = current->next;
}
printf("NULL\n"); //10 -> 20 -> 30->NULL

// 메모리 해제
free(third);
free(second);
free(head);
```

연결 리스트

- 동적 메모리 기반 연결 리스트(함수로 구현)

```c
// 노드 구조 정의
typedef struct Node {
    int data;           // 노드가 저장하는 값
    struct Node* next;  // 다음 노드의 주소(자기 참조)
} Node;

// 노드 생성 함수
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("메모리 할당 실패!\n");
        exit(1);
    }
    newNode->data = value;  //노드의 데이터 값
    newNode->next = NULL;   //다음 노드의 주소 초기화
    return newNode;
}
```

연결 리스트

- 동적 메모리 기반 연결 리스트(함수로 구현)

```c
// 노드 개수 구하는 함수
int getLength(Node* head) {
    int count = 0;
    Node* current = head;  //head 노드를 현재 노드에 저장
    while (current != NULL) {
        count++;
        current = current->next; //현재 노드를 다음 노드로 이동
    }
    return count;
}

// 리스트 출력 함수
void printList(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
```

연결 리스트

- 동적 메모리 기반 연결 리스트(함수로 구현)

```
int main() {

    // 노드 3개 생성 (동적 할당)
    Node* head = createNode(10);
    Node* second = createNode(20);
    Node* third = createNode(30);

    // 노드 연결(link)
    head->next = second;
    second->next = third;

    // 노드의 개수
    printf("노드의 개수: %d\n", getLength(head));
```

연결 리스트

- 동적 메모리 기반 연결 리스트(함수로 구현)

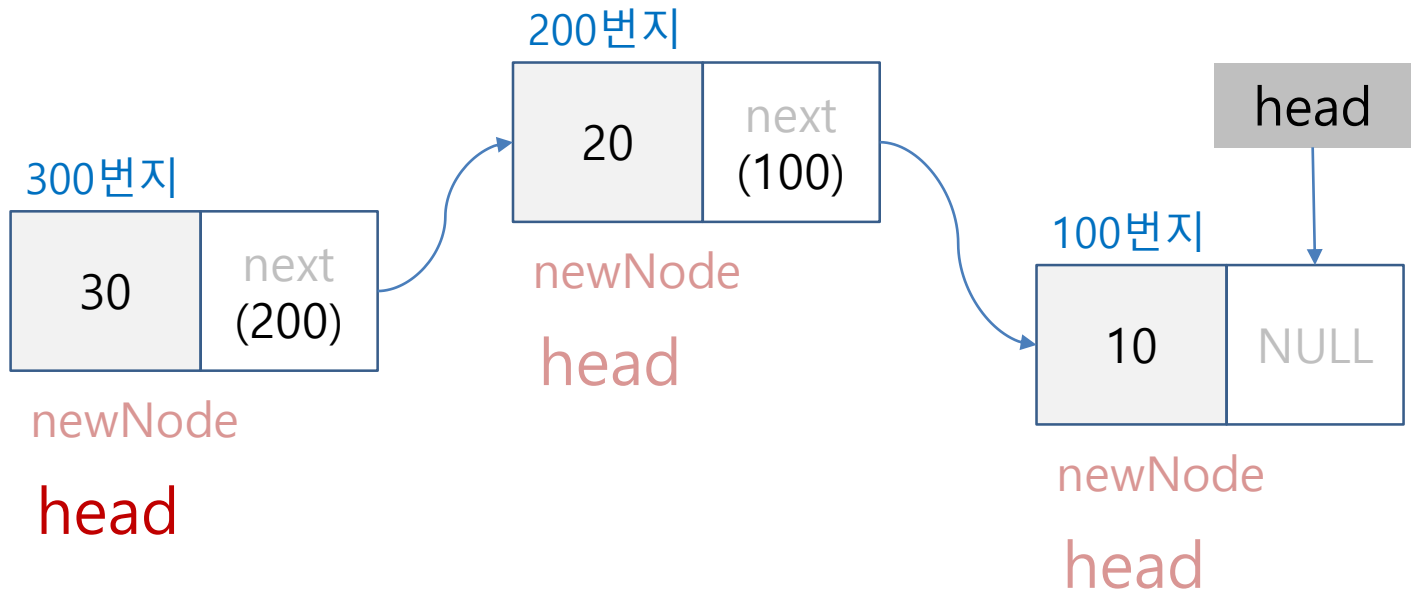
```
    // 출력
    printf("연결 리스트 출력: ");
    printList(head); // 10 -> 20 -> 30 -> NULL

    // 큐 방식(FIFO) 해제
    Node* current = head;
    while (current != NULL) {
        Node* temp = current; //현재 노드를 저장
        printf("free(%d)\n", current->data);
        current = current->next;  //current를 다음 노드로 이동
        free(temp); //현재 노드의 메모리 해제
    }
    return 0;
}
```

```
노드의 개수: 3
연결 리스트 출력: 10 -> 20 -> 30 -> NULL
free(10)
free(20)
free(30)
```

연결 리스트 – 노드 삽입

- 맨 앞에 삽입

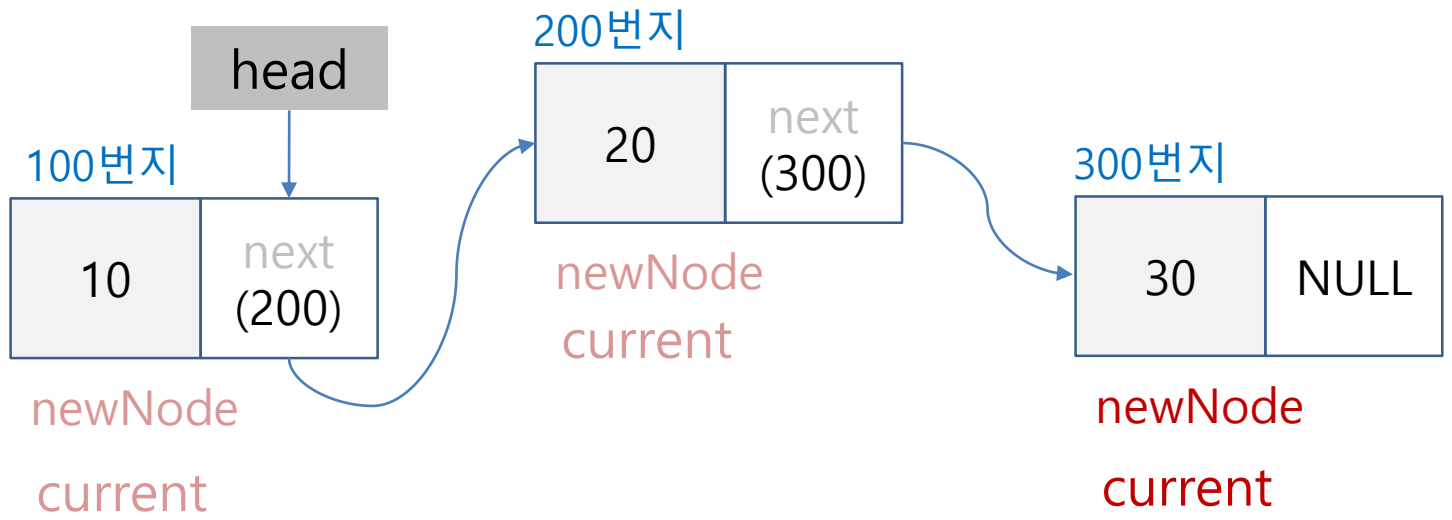
```
노드 개수를 입력하세요: 3
1번째 노드 값 입력: 10
2번째 노드 값 입력: 20
3번째 노드 값 입력: 30

연결 리스트 출력: 30 -> 20 -> 10 -> NULL
```



연결 리스트 – 노드 삽입

- 맨 뒤에 삽입

```
노드 개수를 입력하세요: 3
1번째 노드 값 입력: 10
2번째 노드 값 입력: 20
3번째 노드 값 입력: 30

연결 리스트 출력: 10 -> 20 -> 30 -> NULL
```



연결 리스트 – 사용자 입력

- 동적 메모리 기반 연결 리스트(사용자 입력)

```c
typedef struct {
    int data;
    struct Node* next;
}Node;

int main()
{
    int num;    //노드의 개수
    int value;  //노드의 값
    Node* head = NULL;    //head 노드 초기화
    Node* current = NULL; //현재 노드
    Node* newNode = NULL; //새 노드

    printf("노드 개수를 입력하세요: ");
    scanf("%d", &num);
```

연결 리스트 – 사용자 입력

- 동적 메모리 기반 연결 리스트(사용자 입력)

```
//입력한 개수 만큼 노드 생성 및 연결 반복
for (int i = 0; i < num; i++) {
    newNode = (Node*)malloc(sizeof(Node));

    printf("%d번째 노드 값 입력: ", i + 1);
    scanf("%d", &value);

    //1. 노드 생성 - 맨 앞에서
    newNode->data = value;
    newNode->next = head;  //새 노드가 head와 연결됨
    head = newNode; //새 노드가 head 노드가 됨
```

연결 리스트 – 사용자 입력

- 동적 메모리 기반 연결 리스트(사용자 입력)

```
    //2. 노드 생성 - 맨 뒤에서
    newNode->data = value; //값 저장
    newNode->next = NULL;    //주소 초기화

    if (head == NULL) {
        head = newNode;    //새 노드가 head가 됨
        current = newNode; //새 노드가 현재 노드가 됨
    }
    else {
        current->next = newNode; //새 노드가 현재(이전) 노드와 연결됨
        current = newNode; //새 노드가 현재 노드가 됨
    }
}
```

연결 리스트 – 사용자 입력

- 동적 메모리 기반 단순 연결 리스트(사용자 입력)

```
printf("\n연결 리스트 출력: ");
current = head; //head 노드를 현재 노드에 저장
while (current != NULL) {
    printf("%d -> ", current->data); //현재 노드값 출력
    current = current->next;  //현재 노드를 다음 노드로 이동
}
printf("NULL\n");

//메모리 해제
current = head;
while (current != NULL) {
    Node* temp = current; //현재 노드를 저장
    printf("free(%d)\n", current->data);
    current = current->next;  //current를 다음 노드로 이동
    free(temp); //현재 노드의 메모리 해제
}
```

연결 리스트 – 메뉴 프로그램

- 동적 메모리 기반 연결 리스트(삽입, 삭제)

Node 구조체

```
노드가 저장하는 값
다음 노드의 주소
```

함수의 원형

```
void insertNodeEnd(int value);    //맨 뒤에 노드 삽입
void insertNodeFront(int value); //맨 앞에 노드 삽입
void deleteNode(int value);       //노드 삭제
void printList();                 //노드 출력
void freeList();                  //노드 메모리 해제
```

연결 리스트 – 노드 삭제

- 노드 삭제

초기상태

head → [10 | next] → [20 | next] → [30 | NULL]

1. head 노드(10) 삭제

- 탐색 과정
  current = head (10)
  prev = NULL
  찾음 → prev == NULL 이므로 head 변경

- 링크 재연결
  head = current->next;

- 삭제후 리스트
  head → [20 | next] → [30 | NULL]

연결 리스트 – 노드 삭제

- 노드 삭제

초기상태

head → [10 | next] → [20 | next] → [30 | NULL]

2. 중간 노드 (20) 삭제

- 탐색 시작
  current = [10], prev = NULL
  current = [20], prev = [10] → 찾음

- 링크 재연결
  prev->next = current->next;

- 삭제후 리스트
  head → [10 | next] → [30 | NULL]

연결 리스트 – 노드 삭제

- 노드 삭제

초기상태

head → [10 | next] → [20 | next] → [30 | NULL]

3. 마지막 노드 (30) 삭제

- 탐색 시작
  current = [10], prev = NULL
  current = [20], prev = [10]
  current = [30], prev = [20] → 찾음

- 링크 재연결
  prev->next = current->next; //즉 [20]->next = NULL

- 삭제후 리스트
  head → [10 | next] → [20 | NULL]

연결 리스트 – 노드 삭제

- 노드 삭제

초기상태

head → [10 | next] → [20 | next] → [30 | NULL]

4. 없는 노드 (40) 삭제

- 탐색 시작
  current = [10]
  current = [20]
  current = [30]
  current = NULL → 리스트 끝까지 갔는데 못 찾음

"40 값이 리스트에 없습니다."

연결 리스트

- 동적 메모리 기반 연결 리스트(메뉴 선택)

> 맨 뒤에 노드 삽입

```
=== 연결 리스트 메뉴 ===
1. 맨 뒤에 노드 삽입
2. 맨 앞에 노드 삽입
3. 노드 삭제
4. 리스트 출력
5. 종료
메뉴 선택: 1
삽입할 값 입력: 10
10 맨 뒤 삽입 완료
```

> 리스트 출력

```
=== 연결 리스트 메뉴 ===
1. 맨 뒤에 노드 삽입
2. 맨 앞에 노드 삽입
3. 노드 삭제
4. 리스트 출력
5. 종료
메뉴 선택: 4
리스트: 10 -> 20 -> NULL
```

> 맨 앞에 노드 삽입

```
=== 연결 리스트 메뉴 ===
1. 맨 뒤에 노드 삽입
2. 맨 앞에 노드 삽입
3. 노드 삭제
4. 리스트 출력
5. 종료
메뉴 선택: 2
삽입할 값 입력: 30
30 맨 앞 삽입 완료
```

> 노드 삭제

```
=== 연결 리스트 메뉴 ===
1. 맨 뒤에 노드 삽입
2. 맨 앞에 노드 삽입
3. 노드 삭제
4. 리스트 출력
5. 종료
메뉴 선택: 3
삭제할 값 입력: 20
20 삭제 완료
```

연결 리스트

- 동적 메모리 기반 연결 리스트(메뉴 선택)

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

Node* head = NULL; // 리스트의 시작 노드

// 함수 원형 선언
void insertNodeEnd(int value);   // 맨 뒤 삽입
void insertNodeFront(int value); // 맨 앞 삽입
void deleteNode(int value);
void printList();
void freeList();
```

연결 리스트

- 동적 메모리 기반 연결 리스트(메뉴 선택)

```
int main() {
    bool run = true;
    int choice, value;

    while (run) {
        printf("\n=== 연결 리스트 메뉴 ===\n");
        printf("1. 맨 뒤에 노드 삽입\n");
        printf("2. 맨 앞에 노드 삽입\n");
        printf("3. 노드 삭제\n");
        printf("4. 리스트 출력\n");
        printf("5. 종료\n");
        printf("메뉴 선택: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("삽입할 값 입력: ");
            scanf("%d", &value);
            insertNodeEnd(value);
            break;
```

연결 리스트

- 동적 메모리 기반 연결 리스트(메뉴 선택)

```
            case 2:
                printf("삽입할 값 입력: ");
                scanf("%d", &value);
                insertNodeFront(value);
                break;
            case 3:
                printf("삭제할 값 입력: ");
                scanf("%d", &value);
                deleteNode(value);
                break;
            case 4:
                printList();
                break;
            case 5:
                freeList();
                printf("프로그램 종료\n");
                run = false;
                break;
            default:
                printf("잘못된 선택입니다. 다시 입력하세요.\n");
            }
        }
}
```

연결 리스트

- 노드 삽입 – 맨 뒤에서

```
void insertNodeEnd(int value) { // 맨 뒤 삽입
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("메모리 할당 실패\n");
        return;
    }
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    }
    else {
        Node* current = head;
        while (current->next != NULL)
            current = current->next;
        current->next = newNode;
    }
    printf("%d 맨 뒤 삽입 완료\n", value);
}
```

연결 리스트

- 노드 삽입 - 맨 앞에서

```
void insertNodeFront(int value) { // 맨 앞 삽입
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("메모리 할당 실패\n");
        return;
    }
    newNode->data = value;
    newNode->next = head; // 기존 head 앞에 새 노드 연결
    head = newNode;       // head를 새 노드로 변경
    printf("%d 맨 앞 삽입 완료\n", value);
}
```

연결 리스트

- 노드 삭제

```
void deleteNode(int value) { // 노드 삭제 (값 기준)
    Node* current = head; //현재 탐색중인 노드를 가리킴
    Node* prev = NULL; //current의 이전 노드

    //노드 탐색 - current의 값과 삭제하려는 값이 다를때까지 탐색
    while (current != NULL && current->data != value) {
        //다음 노드로 이동
        prev = current;
        current = current->next;
    }

    if (current == NULL) {
        printf("%d 값이 리스트에 없습니다.\n", value);
        return;
    }
```

연결 리스트

- 노드 삭제

```
    //삭제하려는 값을 찾음 -> 링크 재연결
    if (prev == NULL) { //첫 노드(head)인 경우 삭제
        head = current->next;
    }
    else {
        prev->next = current->next;
    }

    free(current); //현재 노드 메모리 해제
    printf("%d 삭제 완료\n", value);
}
```

연결 리스트

- 리스트 출력

```
// 리스트 출력
void printList() {
    if (head == NULL) {
        printf("리스트가 비어있습니다.\n");
        return;
    }
    Node* current = head;
    printf("리스트: ");
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
```

연결 리스트

- 메모리 해제

```
// 메모리 해제
void freeList() {
    Node* current = head;
    while (current != NULL) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }
    head = NULL;
}
```

고객 대기열 관리

- 고객 대기열 관리

> 고객 추가

```
==== 고객 대기열 관리 ====
1. 고객 추가
2. 고객 처리
3. 대기열 출력
4. 종료
메뉴 선택: 1
고객 이름 입력: 우영우
우영우님이 대기열에 추가되었습니다.
```

> 고객 처리

```
==== 고객 대기열 관리 ====
1. 고객 추가
2. 고객 처리
3. 대기열 출력
4. 종료
메뉴 선택: 2
우영우님 업무 처리 완료.
```

> 대기열 출력

```
==== 고객 대기열 관리 ====
1. 고객 추가
2. 고객 처리
3. 대기열 출력
4. 종료
메뉴 선택: 3
현재 대기열: [우영우] [장그래] [오상식]
```

고객 대기열 관리

- 고객 대기열 관리

```c
// 고객 노드 정의
typedef struct Node {
    char name[20];
    struct Node* next;
} Node;

// 대기열 구조체
typedef struct {
    Node* front;  // 큐의 맨 앞
    Node* rear;   // 큐의 맨 뒤
} Queue;

void initQueue(Queue* q) { // 큐 초기화
    q->front = NULL;
    q->rear = NULL;
}

int isEmpty(Queue* q) { // 큐가 비었는지 확인
    return q->front == NULL;
}
```

고객 대기열 관리

- 고객 대기열 관리

```c
// 고객 추가 (Enqueue)
void enQueue(Queue* q, const char* name) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("메모리 할당 실패!\n");
        exit(1);
    }
    strcpy(newNode->name, name);
    newNode->next = NULL;

    if (isEmpty(q)) {
        q->front = newNode;
        q->rear = newNode;
    }
    else {
        q->rear->next = newNode;
        q->rear = newNode;
    }
    printf("%s님이 대기열에 추가되었습니다.\n", name);
}
```

고객 대기열 관리

- 고객 대기열 관리

```
// 고객 꺼내기 (Dequeue)
int deQueue(Queue* q, char* name) {
    if (isEmpty(q)) {
        printf("대기열이 비어 있습니다!\n");
        return -1;
    }
    Node* temp = q->front;
    strcpy(name, temp->name);

    q->front = q->front->next;
    if (q->front == NULL) {  // 마지막 노드 제거 시 rear도 NULL로
        q->rear = NULL;
    }
    free(temp);
    return 0;
}
```

고객 대기열 관리

- 고객 대기열 관리

```
// 큐 상태 출력
void printQueue(Queue* q) {
    if (isEmpty(q)) {
        printf("대기열이 비어 있습니다.\n");
        return;
    }
    printf("현재 대기열: ");
    Node* cur = q->front;
    while (cur != NULL) {
        printf("[%s] ", cur->name);
        cur = cur->next;
    }
    printf("\n");
}
```

고객 대기열 관리

- 고객 대기열 관리

```
int main() {
    Queue q;
    char name[20];
    bool run = true;
    int choice;

    initQueue(&q); //큐 초기화

    while (run) {
        printf("\n==== 고객 대기열 관리 ====\n");
        printf("1. 고객 추가\n");
        printf("2. 고객 처리\n");
        printf("3. 대기열 출력\n");
        printf("4. 종료\n");
        printf("메뉴 선택: ");
        scanf("%d", &choice);
        getchar(); // 버퍼 정리
```

고객 대기열 관리

- 고객 대기열 관리

```
switch (choice) {
case 1:
    printf("고객 이름 입력: ");
    fgets(name, sizeof(name), stdin);
    name[strcspn(name, "\n")] = '\0'; // 개행 제거
    enQueue(&q, name);
    break;

case 2:
    if (deQueue(&q, name) == 0) {
        printf("%s님 업무 처리 완료.\n", name);
    }
    break;

case 3:
    printQueue(&q);
    break;
```

고객 대기열 관리

- 고객 대기열 관리

```
            case 4:
                printf("프로그램을 종료합니다.\n");
                // 남은 고객 메모리 해제
                while (!isEmpty(&q)) {
                    deQueue(&q, name);
                }
                run = false;
                break;

            default:
                printf("잘못된 선택입니다. 다시 입력하세요.\n");
            }
        }
}
```

이진 트리(Tree) 자료 구조

- 트리(Tree)

```c
// 트리 노드 정의
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// 노드 생성 함수
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("메모리 할당 실패!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

이진 트리(Tree)

- 트리(Tree)

```c
// 이진 트리에 노드 삽입 (정렬된 이진 트리)
Node* insert(Node* root, int data) {
    if (root == NULL) return createNode(data);

    if (data < root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);

    return root;
}

// 중위 순회 (왼쪽 → 현재 → 오른쪽)
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

이진 트리(Tree)

- 트리(Tree)

```
// 후위 순회 (왼쪽 → 오른쪽 → 현재)
void postorder(Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// 전위 순회 (현재 → 왼쪽 → 오른쪽)
void preorder(Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

이진 트리(Tree)

- 트리(Tree)

```
// 트리 메모리 해제
void freeTree(Node* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}

int main() {
    Node* root = NULL;

    // 트리에 데이터 삽입
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);
```

이진 트리(Tree)

- 트리(Tree)

```
    // 출력
    printf("중위 순회 (오름차순 정렬): ");
    inorder(root);
    printf("\n");

    printf("전위 순회: ");
    preorder(root);
    printf("\n");

    printf("후위 순회: ");
    postorder(root);
    printf("\n");

    freeTree(root); // 메모리 해제

    return 0;
}
```

```
중위 순회 (오름차순 정렬): 20 30 40 50 60 70 80
전위 순회: 50 30 20 40 70 60 80
후위 순회: 20 40 30 60 80 70 50
```

이진 검색 트리(Tree)

- 이진 검색 트리(Tree)

```c
// 노드 구조 정의
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// 노드 생성 함수
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("메모리 할당 실패!\n");
        exit(1);
    }
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

이진 검색 트리(Tree)

- 이진 검색 트리(Tree)

```
// 이진 검색 트리에 삽입
Node* insert(Node* root, int value) {
    if (root == NULL) return createNode(value);

    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
    // 중복은 무시 (원한다면 카운트 처리 가능)

    return root;
}

// 중위 순회 (왼쪽 → 현재 → 오른쪽)
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

이진 검색 트리(Tree)

- 이진 검색 트리(Tree)

```
// 노드 탐색
Node* search(Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;

    if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// 트리 메모리 해제
void freeTree(Node* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
```

이진 검색 트리(Tree)

- **이진 검색 트리(Tree) – Main**

```
Node* root = NULL;

// 노드 삽입
int values[] = { 50, 30, 70, 20, 40, 60, 80 };
for (int i = 0; i < 7; i++)
    root = insert(root, values[i]);

// 중위 순회 (정렬된 출력)
printf("중위 순회 결과 (정렬된 데이터): ");
inorder(root);
printf("\n");

// 값 검색
int key = 60;
Node* found = search(root, key);
if (found)
    printf("%d를 찾았습니다!\n", key);
else
    printf("%d는 트리에 없습니다.\n", key);

freeTree(root); // 메모리 해제
```

```
중위 순회 결과 (정렬된 데이터): 20 30 40 50 60 70 80
60를 찾았습니다!
```