C – 자료구조 2

LinkedList, Tree, Hash

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

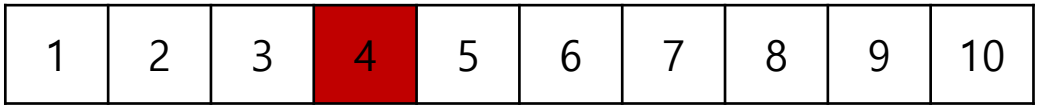
연결 리스트(Linked List)

- 연결 리스트(Linked List)의 필요성

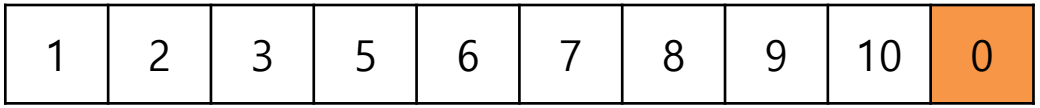
✓ 배열 자료구조의 문제점

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");
```

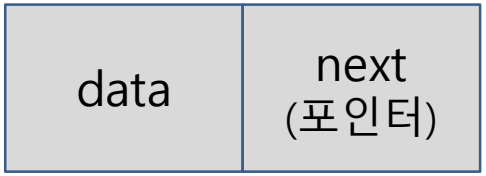
연결 리스트

- 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;
}
```

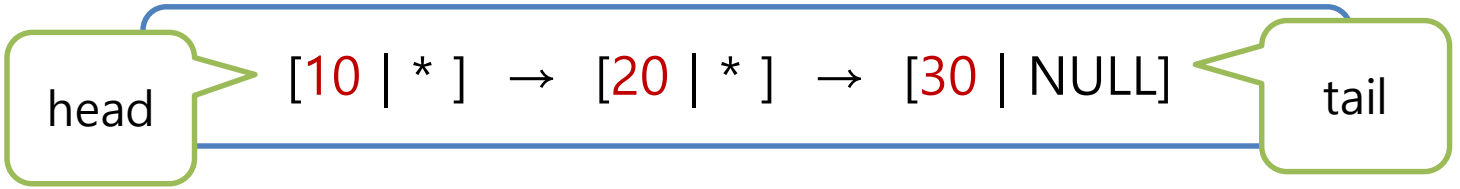
연결 리스트

- 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

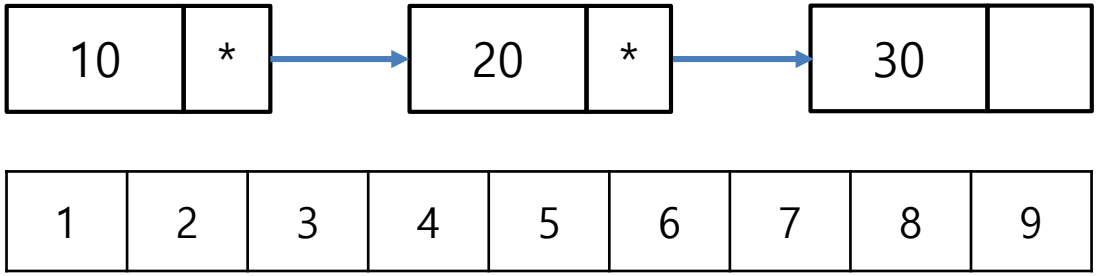


✓ 각 노드가 자신의 데이터와 다음 노드의 주소를 저장.

✓ NULL이 나오면 리스트의 끝.

연결 리스트(Linked List)

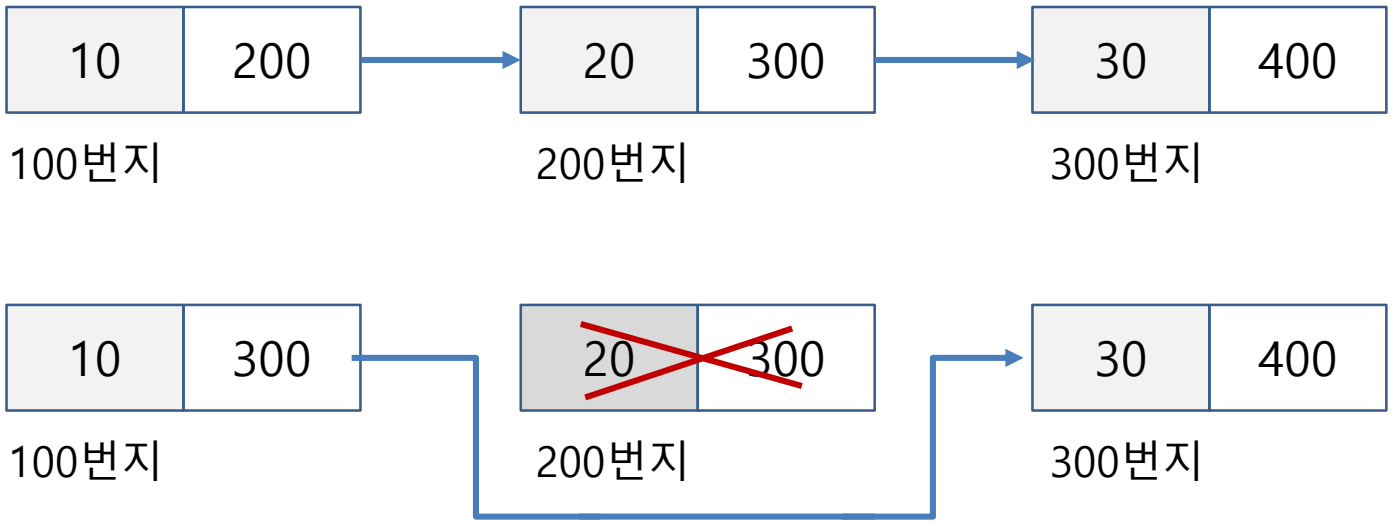
- 배열과 연결 리스트의 비교



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

- 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)

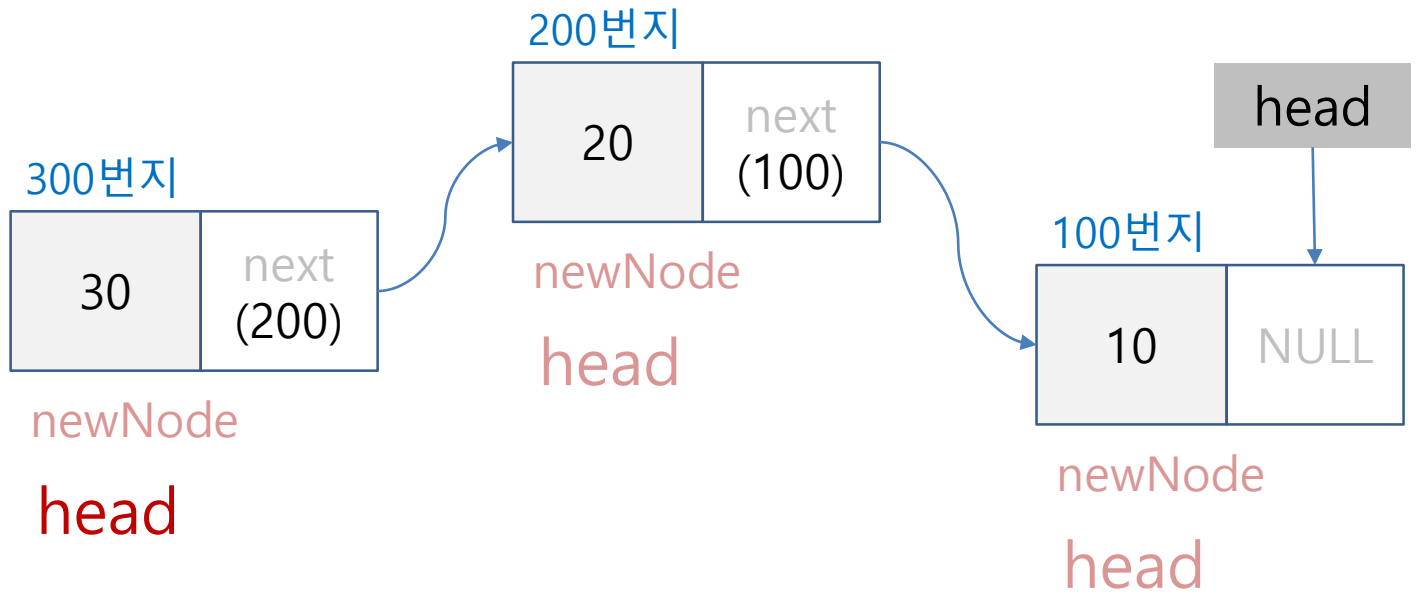
```
    // 출력
    printf("연결 리스트 출력: ");
    printList(head); // 10 -> 20 -> 30 -> NULL

    // 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

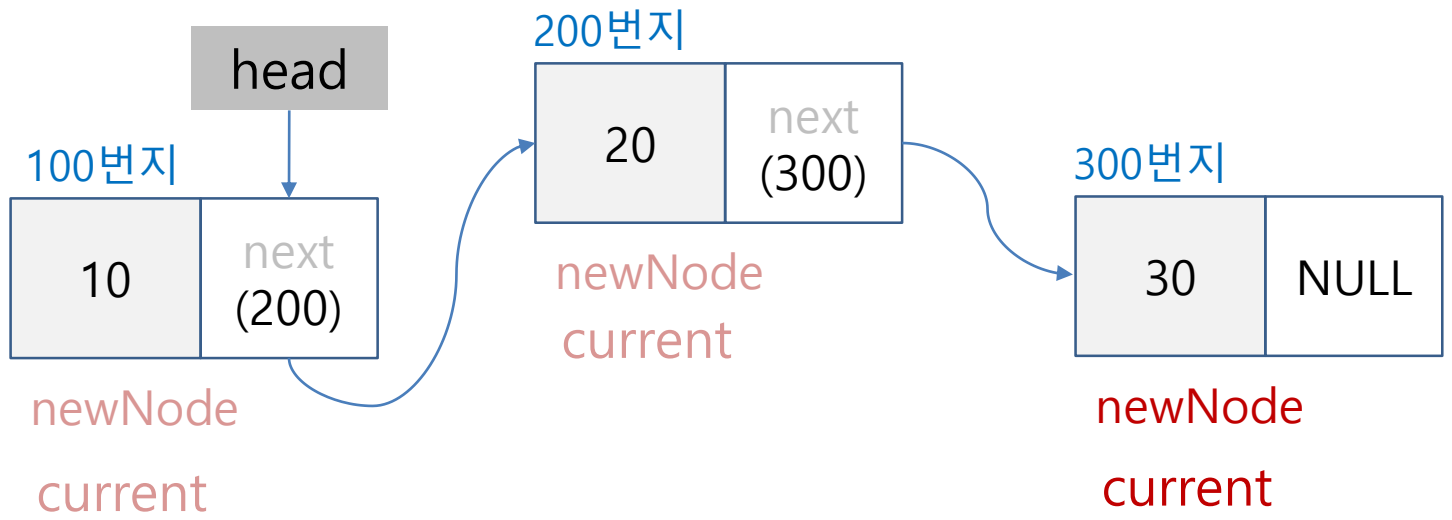
```
노드 개수를 입력하세요: 3
1번째 노드 값 입력: 10
2번째 노드 값 입력: 20
3번째 노드 값 입력: 30

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)

```
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

```
// 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

```c
// 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");
        }
    }
}
```

트리(Tree) 자료 구조

- **트리(Tree)**

  트리는 계층적(hierarchical) 구조를 표현하는 자료구조로,
  노드(node) 와 간선(edge) 으로 이루어진 비선형(non-linear) 자료구조이다.

  - **트리 자료구조의 특징**

    비선형 구조:
    데이터를 순차적으로 나열하는 것이 아니라, 계층적인 방식으로 구성됩니다.

    계층적 관계:
    데이터가 부모-자식 관계를 가지며 상위 노드에서 하위 노드로 연결됩니다.
    루트:
    트리의 최상단에 위치하는 시작 노드입니다.
    노드:
    트리 구조를 이루는 각 요소로, 데이터와 연결 정보(자식 노드)를 가집니다.
    간선(Edge):
    노드와 노드를 연결하는 선으로, 부모-자식 관계를 나타냅니다.

    용도
    파일 시스템, 디렉터리 구조, 조직도, 트리 순회(Pre-order, In-order, Post-order 등) 등 다양한 곳에서 사용됨

트리(Tree) 자료 구조

- **이진 트리(Binary Tree)**

  이진 트리는 각 노드가 최대 두 개의 자식 노드를 가질 수 있는 트리이다.
  왼쪽 자식(Left Child), 오른쪽 자식(Right Child)

  루트(Root): 트리의 최상위 노드 (출발점)

  부모(Parent) / 자식(Child): 노드 간 관계

  형제(Sibling): 같은 부모를 가진 노드

  리프(Leaf): 자식이 없는 노드

  높이(Height): 루트에서 가장 깊은 리프까지의 거리

  완전 이진 트리 (Complete Binary Tree)
  마지막 레벨을 제외하면 모든 레벨이 꽉 차 있고, 마지막 레벨은 왼쪽부터 채워지는 트리
  → 힙(Heap) 자료구조가 이 형태

  이진 탐색 트리 (Binary Search Tree, BST)
  왼쪽 서브트리 값 < 루트 < 오른쪽 서브트리 값
  탐색, 삽입, 삭제가 평균 O(log n)에 가능

트리(Tree) 자료 구조

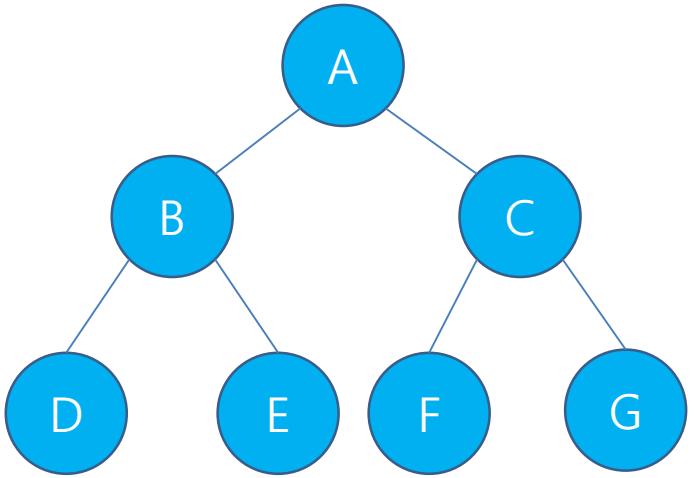
- **이진 트리(Binary Tree) 순회**

  트리의 노드를 방문하는 방식

  전위 순회 (Preorder): **Root** → Left → Right

  중위 순회 (Inorder): Left → **Root** → Right

  후위 순회 (Postorder): Left → Right → **Root**



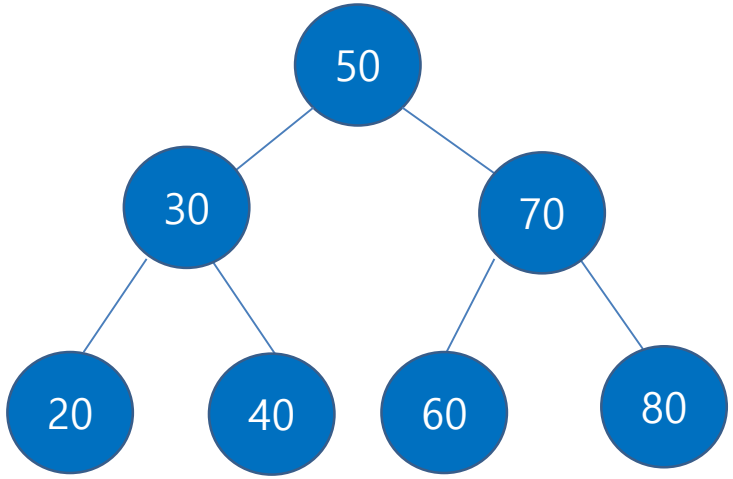
전위 순회 – A B D E C F G

중위 순회 – D B E A F C G

후위 순회 – D E B F G C A

이진 트리(Tree) 자료 구조

- 이진 트리(Tree) 순회 구현




```
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 트리(Tree) 자료 구조

- **이진 트리(Tree) 구현 단계**
  - 노드 구조체 정의
  - 노드 생성 함수 – 새로운 노드를 동적으로 할당 반환
  - 노드 삽입 함수 (insert)
    - ✓ 이진 탐색 규칙에 따라 삽입 – 루트보다 작으면 왼쪽, 크면 오른쪽
    - ✓ 재귀 함수 호출
  - 트리 순회
    - ✓ 전위 순회, 중위 순회, 후위 순회 – 재귀 함수 호출

이진 트리(Tree) 자료 구조

- 이진 트리(Tree)

```c
// 트리 노드 정의
typedef struct Node {
    int data;  //노드에 저장할 데이터
    struct Node* left;  //왼쪽 자식 노드
    struct Node* right; //오른쪽 자식 노드
} 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 = NULL;
    newNode->right = NULL;
    return newNode;
}
```

이진 트리(Tree)

- 이진 트리(Tree)

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

    //재귀 호출
    if (data < root->data)
        root->left = insert(root->left, data); //왼쪽 서브루트에 삽입
    else
        root->right = insert(root->right, data); //오른쪽 서브루트에 삽입

    return root; //변경된 루트 반환
}
```

이진 트리(Tree)

- 이진 트리(Tree)

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

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

이진 트리(Tree)

- 이진 트리(Tree)

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

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

이진 트리(Tree)

- 이진 트리(Tree)

```
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);

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

이진 트리(Tree)

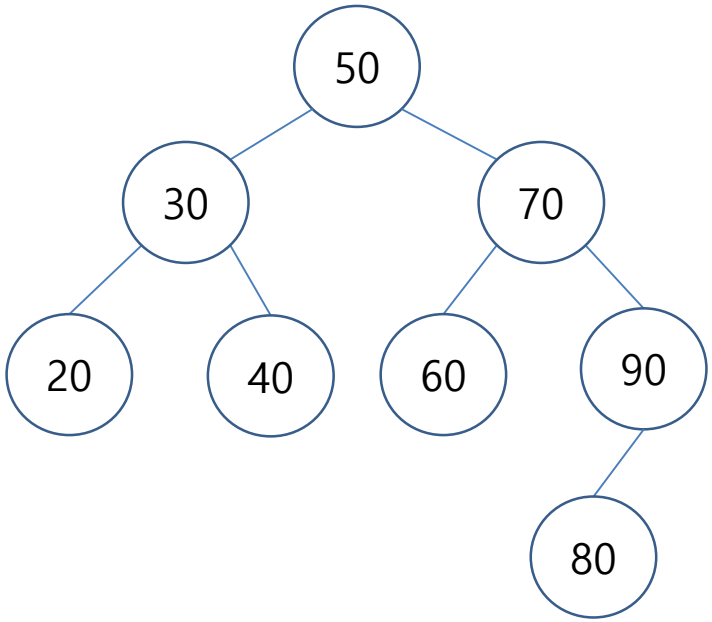
- 이진 트리(Tree)

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

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

freeTree(root); // 메모리 해제
```

```
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

- **이진 탐색 트리(Binary Search Tree)**
  - 모든 원소는 유일한 키 값을 갖는다.
  - 왼쪽 서브트리의 모든 원소들은 루트의 키보다 작은 값을 갖는다.
  - 오른쪽 서브트리의 모든 원소들은 루트의 키보다 큰 값을 갖는다.
  - 왼쪽 서브트리와 오른쪽 서브트리도 **이진 탐색트리**이다.
  - 노드를 하나 통과할 때 마다 검색대상이 1/2로 줄어든다.
    (**Big-O**는 밑이 2인 로그의 자료 개수 N) – 매우 빠르다.



중위 운행(Left–Root-Right)
오름차순 정렬하는 특징을 가진다.
20 30 40 50 60 70 80 90

이진 검색 트리

- 이진 검색 트리(Binary Search Tree)

```c
//노드 구조체
typedef struct {
    int data;  //노드에 저장할 데이터
    struct Node* left;  //왼쪽 자식 노드
    struct Node* right; //오른쪽 자식 노드
}Node;

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

이진 검색 트리(Tree)

- 이진 검색 트리(Binary Search Tree)

```
//노드 삽입
Node* insert(Node* root, int value) {
    //처음 트리가 비었을때(재귀의 종료 조건)
    if (root == NULL)
        return createNode(value);  //새 노드를 root로 반환

    //재귀 호출로 내려갔을때 - 해당 자식노드가 NULL일때 그 자리에 새 노드 삽입
    if (value < root->data)
        root->left = insert(root->left, value); //왼쪽 서브루트에 삽입
    if (value > root->data)
        root->right = insert(root->right, value); //오른쪽 서브루트에 삽입

    return root; //변경된 루트 반환
}
```

이진 검색 트리(Tree)

- 이진 검색 트리(Binary Search Tree)

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

//노드 탐색
Node* search(Node* root, int key) {
    //root가 없거나 그 값이 찾는 값이면 root 반환(종료 조건)
    if (root == NULL || root->data == key)
        return root;

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

이진 검색 트리(Tree)

- 이진 검색 트리(Binary Search Tree) – 노드 삭제

```
//최소값 찾기
Node* findMin(Node* root) {
    while (root->left != NULL)
        root = root->left; //왼쪽 노드값을 찾아서 root에 저장
    return root;
}

//노드 삭제
Node* delete(Node* root, int key) {
    if (root == NULL)
        return root; //트리가 비어있으면 그대로 반환

    if (key < root->data) //찾는 값이 루트보다 작으면 왼쪽으로
        root->left = delete(root->left, key);
    else if (key > root->data)
        root->right = delete(root->right, key);
```

이진 검색 트리(Tree)

- 이진 검색 트리(Binary Search Tree) – 노드 삭제

```
    else { //key == root->data, 값을 찾음
        if (root->left == NULL) { //왼쪽 자식이 없음
            Node* temp = root->right; //오른쪽 자식을 임시변수에 저장
            free(root); //현재 노드 삭제
            return temp;  //오른쪽 자식 반환(부모와 연결)
        }
        else if (root->right == NULL) { //오른쪽 자식이 없음
            Node* temp = root->left;
            free(root);
            return temp;
        }
        else { //자식이 둘 다 있는 경우
            Node* temp = findMin(root->right); //오른쪽에서 최소값 찾음
            root->data = temp->data;  //현재 노드값 교체
            root->right = delete(root->right, temp->data); //최소값 노드 삭제
        }
    }
    return root;
}
```

이진 검색 트리(Tree)

- 이진 검색 트리(Binary Search Tree)

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

int main() {

    Node* root = NULL; //root 노드 생성

    //노드 삽입
    int values[] = {50, 30, 70, 20, 40, 60, 80, 90};
    int size = sizeof(values) / sizeof(values[0]);

    for (int i = 0; i < size; i++) {
        root = insert(root, values[i]);
    }
```

이진 검색 트리(Tree)

- 이진 검색 트리(Binary Search Tree)

```
    printf("중위 운행 결과: ");
    inOrder(root);
    printf("\n");

    //노드 검색
    int key = 40;
    Node* found = search(root, key);
    if (found)
        printf("%d을 찾았습니다.\n", key);
    else
        printf("%d는 트리에 없습니다.\n", key);

    //노드 삭제
    puts("70 삭제 후");
    root = delete(root, 70);
    inOrder(root);

    //메모리 해제
    freeTree(root);
    return 0;
}
```

```
중위 운행 결과: 20 30 40 50 60 70 80 90
40을 찾았습니다.
70 삭제 후
20 30 40 50 60 80 90
```

도서 관리 프로그램

- **도서 관리 프로그램**

```c
// 책 구조체
typedef struct {
    int isbn;           // 책 ISBN (고유번호, 키 값)
    char title[50];     // 책 제목
    char author[30];    // 저자
} Book;

// 트리 노드 구조체
typedef struct Node {
    Book data;
    struct Node* left;
    struct Node* right;
} Node;
```

도서 관리 프로그램

- **도서 관리 프로그램**

```c
// 새 노드 생성
Node* createNode(int isbn, const char* title, const char* author) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data.isbn = isbn;
    strcpy(newNode->data.title, title);
    strcpy(newNode->data.author, author);
    newNode->left = newNode->right = NULL;
    return newNode;
}

// 삽입 (ISBN 기준)
Node* insert(Node* root, int isbn, const char* title, const char* author) {
    if (root == NULL) return createNode(isbn, title, author);

    if (isbn < root->data.isbn)
        root->left = insert(root->left, isbn, title, author);
    else if (isbn > root->data.isbn)
        root->right = insert(root->right, isbn, title, author);

    return root;
}
```

도서 관리 프로그램

- **도서 관리 프로그램**

```
// 탐색 (ISBN 기준)
Node* search(Node* root, int isbn) {
    if (root == NULL || root->data.isbn == isbn)
        return root;

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

// 중위 순회 (ISBN 순으로 정렬 출력)
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("ISBN:%d | 제목:%s | 저자:%s\n",
            root->data.isbn, root->data.title, root->data.author);
        inorder(root->right);
    }
}
```

도서 관리 프로그램

- **도서 관리 프로그램**

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

int main() {
    Node* root = NULL;

    // 도서 추가
    root = insert(root, 9781001, "C 프로그래밍", "김철수");
    root = insert(root, 9781003, "자료구조", "이영희");
    root = insert(root, 9781002, "알고리즘", "박민수");

    printf("도서 전체 목록 (ISBN 순):\n");
    inorder(root);
```

도서 관리 프로그램

- **도서 관리 프로그램**

```
    //도서 검색
    int searchIsbn = 9781002;
    Node* result = search(root, searchIsbn);
    if (result != NULL)
        printf("\n검색 결과 → ISBN:%d | 제목:%s | 저자:%s\n",
            result->data.isbn, result->data.title, result->data.author);
    else
        printf("\nISBN %d 책을 찾을 수 없습니다.\n", searchIsbn);

    // 메모리 해제
    freeTree(root);

    return 0;
}
```

해쉬(Hash) 자료 구조

❖ 해쉬(Hash)

해시(Hash)란 데이터를 특정 규칙(해시 함수) 으로 변환하여 빠른 검색, 삽입, 삭제를 가능하게 하는 자료구조입니다.

즉, 데이터를 Key → Index 로 매핑하여 배열처럼 빠르게 접근합니다.

▪ 주요 개념

해시 함수(Hash Function)

키(key)를 입력받아 배열의 인덱스(index)를 계산하는 함수

예: index = key % TABLE_SIZE;

해시 테이블(Hash Table)

해시 함수로 변환된 인덱스에 데이터를 저장하는 배열

해쉬(Hash) 자료 구조

❖ 해쉬(Hash)

충돌(Collision)

서로 다른 키가 같은 인덱스로 해싱되는 경우 발생

이를 해결하기 위한 기법이 필요함

- **충돌(Collision) 해결 방법**

체이닝(Chaining)

같은 인덱스에 여러 개의 데이터를 연결 리스트로 저장

개방 주소법(Open Addressing)

충돌 발생 시 다른 빈 칸을 찾아 저장

해쉬(Hash) 자료 구조

- 해시 테이블 구현 예제 (체이닝 방식)

```c
#define TABLE_SIZE 10

// 노드 구조체 (체이닝)
typedef struct Node {
    char* key;
    int value;
    struct Node* next;
} Node;

Node* hashTable[TABLE_SIZE];

// 해시 함수 (문자열 → 인덱스)
unsigned int hash(const char* key) {
    unsigned int hash = 0;
    while (*key) {
        hash = (hash * 31) + *key++; // 간단한 다항식 해싱
    }
    return hash % TABLE_SIZE;
}
```

해쉬(Hash) 자료 구조

- 해시 테이블 구현 예제 (체이닝 방식)

```
// 삽입 함수
void insert(const char* key, int value) {
    unsigned int index = hash(key);
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = _strdup(key);
    newNode->value = value;
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}
```

해쉬(Hash) 자료 구조

- 해시 테이블 구현 예제 (체이닝 방식)

```
// 검색 함수
int search(const char* key) {
    unsigned int index = hash(key);
    Node* current = hashTable[index];
    while (current) {
        if (strcmp(current->key, key) == 0) {
            return current->value; // 찾음
        }
        current = current->next;
    }
    return -1; // 못 찾음
}
```

해쉬(Hash) 자료 구조

- 해시 테이블 구현 예제 (체이닝 방식)

```
// 해시 테이블 출력
void printTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("[%d] : ", i);
        Node* current = hashTable[i];
        while (current) {
            printf("(%s, %d) -> ", current->key, current->value);
            current = current->next;
        }
        printf("NULL\n");
    }
}
```

해쉬(Hash) 자료 구조

- 해시 테이블 구현 예제 (체이닝 방식)

```
int main() {
    insert("apple", 100);
    insert("banana", 200);
    insert("grape", 300);
    insert("orange", 400);
    insert("melon", 500);

    printTable();

    printf("검색 결과: apple → %d\n", search("apple"));
    printf("검색 결과: banana → %d\n", search("banana"));
    printf("검색 결과: kiwi → %d\n", search("kiwi"));

    return 0;
}
```

해쉬(Hash) 자료 구조

- **컴퓨터 용어 사전**

```c
// 노드 구조체 (체이닝 방식)
typedef struct Node {
    char* word;        // 용어
    char* meaning;     // 설명
    struct Node* next;
} Node;

Node* hashTable[TABLE_SIZE];

// 해시 함수 (문자열 → 인덱스)
unsigned int hash(const char* key) {
    unsigned int h = 0;
    while (*key) {
        h = (h * 31) + *key++;
    }
    return h % TABLE_SIZE;
}
```

해쉬(Hash) 자료 구조

- 컴퓨터 용어 사전

```
// 단어 삽입 함수
void insert(const char* word, const char* meaning) {
    unsigned int index = hash(word);
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->word = _strdup(word);
    newNode->meaning = _strdup(meaning);  // 의미도 복사
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}
```

해쉬(Hash) 자료 구조

- 컴퓨터 용어 사전

```
// 단어 검색 함수
char* search(const char* word) {
    unsigned int index = hash(word);
    Node* current = hashTable[index];
    while (current) {
        if (strcmp(current->word, word) == 0) {
            return current->meaning;
        }
        current = current->next;
    }
    return NULL; // 못 찾음
}
```

해쉬(Hash) 자료 구조

- 컴퓨터 용어 사전

```
// 사전 출력
void printDictionary() {
    printf("==== 해시 기반 용어 사전 ====\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        Node* current = hashTable[i];
        while (current) {
            printf("%s : %s\n", current->word, current->meaning);
            current = current->next;
        }
    }
    printf("============================\n");
}
```

해쉬(Hash) 자료 구조

- **컴퓨터 용어 사전**

```
// 단어 삽입
insert("Array", "연속된 메모리 공간에 데이터를 저장하는 자료구조");
insert("Stack", "후입선출(LIFO) 방식의 자료구조");
insert("Queue", "선입선출(FIFO) 방식의 자료구조");
insert("Tree", "계층적 구조를 표현하는 자료구조");
insert("Hash", "키를 해시 함수로 변환해 빠르게 검색하는 자료구조");

// 사전 출력
printDictionary();

// 검색 테스트
char* term = "Stack";
char* meaning = search(term);
if (meaning) {
    printf("\n검색 결과 → %s : %s\n", term, meaning);
}
else {
    printf("\n'%s' 단어를 찾을 수 없습니다.\n", term);
}
```