자료구조(연결 리스트)

유튜브 주소 : https://youtu.be/IdM28Jq6aS0

포인터

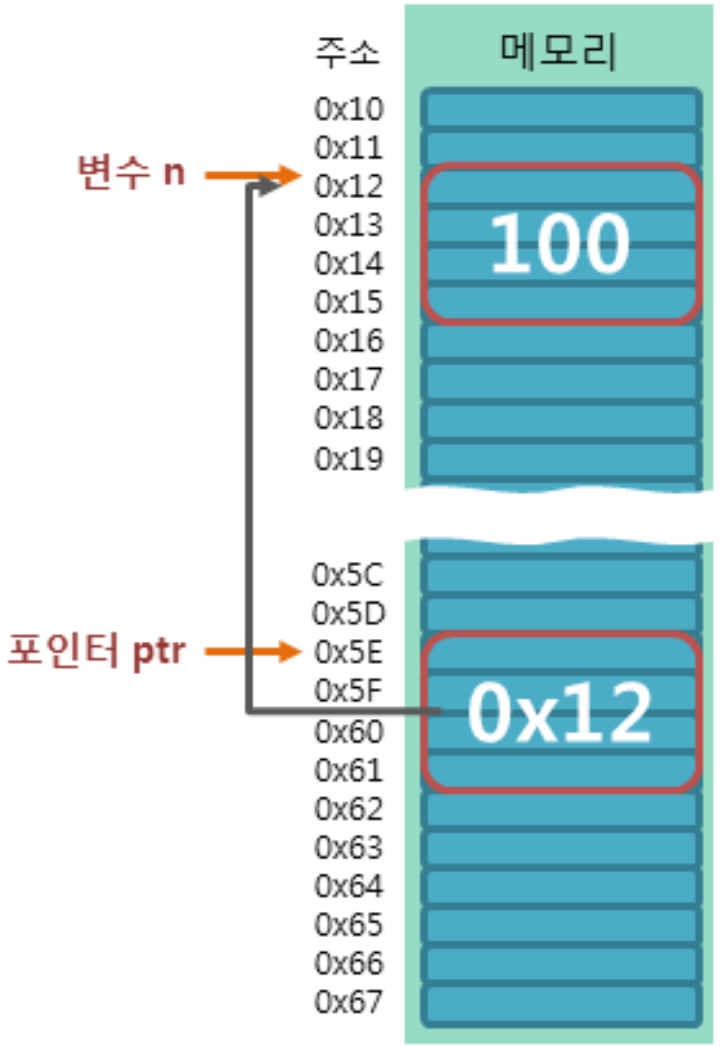
연결리스트

연결리스트 코드 분석

포인터

- 메모리의 주소값을 저장하는 변수

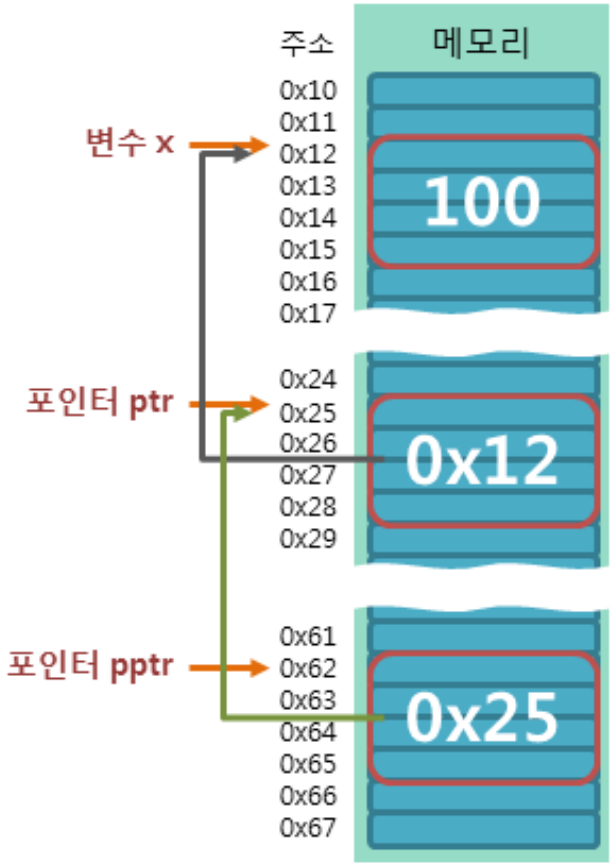
- Int n = 100 // 변수 선언
- Int* ptr = &n; // 포인터 변수 선언



포인터

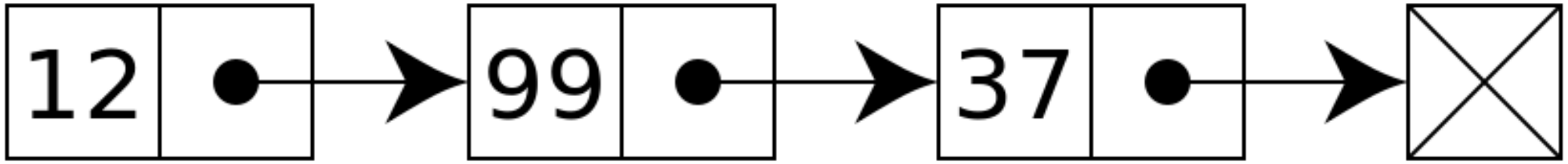
- 포인터 변수의 선언
  - 타입* 포인터이름 = &변수이름 or 주소값;

- 포인터의 참조
  - 포인터는 참조 연산자(*)를 사용하여 참조 가능
  - Int x = 7; // 변수 선언
  - Int* ptr = &x; // 포인터 선언
  - Int* pptr = &ptr; // 포인터 참조



연결 리스트

- 데이터를 화살표로 연결해서 관리하는 데이터 구조
- 포인터를 사용하여 여러 개의 노드를 연결
- 삽입과 삭제 연산에 소요되는 시간을 절약하기 위한 자료구조



- 노드 : 데이터 저장 단위, 데이터 값&포인터로 구성
- 포인터 : 노드 안에서 이전이나 다음 노드와의 연결정보를 담은 공간

연결 리스트

배열과의 차이점

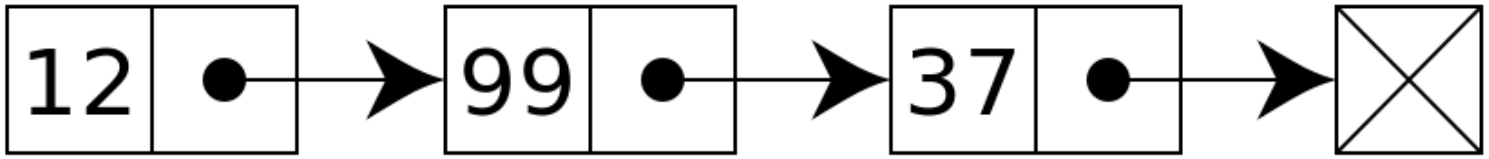
	배열	연결 리스트
크기	고정적	동적
주소	순차적	무작위
접근 속도	빠름	느림
삽입 / 삭제	비효율적	효율적

연결 리스트 시간 복잡도

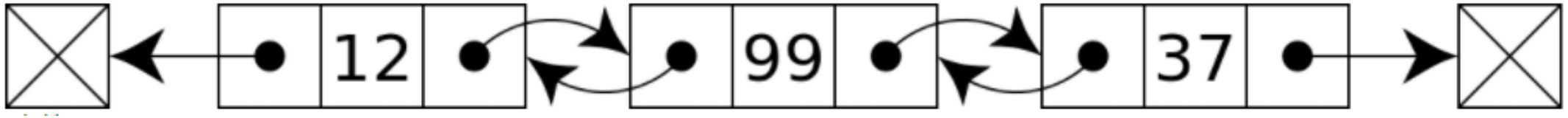
	시간 복잡도
삽입 연산	맨 앞/뒤에 삽입할 경우 : O(1) 중간에 삽입할 경우 : O(N)
삭제 연산	맨 앞/뒤에서 삭제할 경우 : O(1) 중간에서 삭제할 경우 : O(N)
탐색 연산	O(N)

연결 리스트

- 단순 연결 리스트
  - 한 방향으로만 연결된 구조
  - 다음 노드 레퍼런스 정보만 필요
  - 노드 탐색이 한 쪽으로만 가능



- 이중 연결 리스트
  - 양방향으로 연결된 구조
  - 전 후 노드 레퍼런스 정보 모두 필요
  - 노드 탐색이 양쪽으로 모두 가능



연결 리스트

- 연결 리스트 장점
  - 리스트의 길이가 가변적 -> 배열 단점 커버
  - 대용량 데이터 처리 적합
  - 자료 삽입 및 삭제 용이

- 연결 리스트 단점
  - 알고리즘이 복잡함
  - 포인터를 사용하므로 저장 공간이 낭비
  - 자료 탐색 시 비효율적

단순 연결 리스트 구현 코드 분석

```c
#include <stdio.h>
#include <stdlib.h>

typedef int element;
typedef struct ListNode {
    element data;
    struct ListNode *link;
} ListNode;

ListNode* insert_first(ListNode *head, int value)
{
    ListNode *p = (ListNode *)malloc(sizeof(ListNode));
    p->data = value;
    p->link = head;
    head = p;
    return head;
}

ListNode*  insert(ListNode *head, ListNode *pre, element value)
{
    ListNode *p = (ListNode *)malloc(sizeof(ListNode));
    p->data = value;
    p->link = pre->link;
    pre->link = p;
    return head;
}
```

```c
ListNode* delete_first(ListNode *head)
{
    ListNode *removed;
    if (head == NULL) return NULL;
    removed = head;
    head = removed->link;
    free(removed);
    return head;
}

ListNode* delete(ListNode *head, ListNode *pre)
{
    ListNode *removed;
    removed = pre->link;
    pre->link = removed->link;
    free(removed);
    return head;
}

void print_list(ListNode *head)
{
    for (ListNode *p = head; p != NULL; p = p->link)
        printf( _Format: "%d->", p->data);
    printf( _Format: "NULL \n");
}
```

이중 연결 리스트 구현 코드 분석

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef int element;
typedef struct DListNode {
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
} DListNode;

void init(DListNode* phead)
{
    phead->llink = phead;
    phead->rlink = phead;
}


void print_dlist(DListNode* phead)
{
    DListNode* p;
    for (p = phead->rlink; p != phead; p = p->rlink) {
        printf( _Format: "<-| |%d| |-> ", p->data);
    }
    printf( _Format: "\n");
}
```

```c
void dinsert(DListNode *before, element data)
{
    DListNode *newnode = (DListNode *)malloc(sizeof(DListNode));
    strcpy(newnode->data, data);
    newnode->llink = before;
    newnode->rlink = before->rlink;
    before->rlink->llink = newnode;
    before->rlink = newnode;
}

void ddelete(DListNode* head, DListNode* removed)
{
    if (removed == head) return;
    removed->llink->rlink = removed->rlink;
    removed->rlink->llink = removed->llink;
    free(removed);
}
```

Q & A