스택과 큐의 동작

연결리스트 활용

2021.10.04 이종민

연결리스트(linked list)

연결리스트란?

- 각 노드가 데이터와 포인터를 가지고 연결되어있는 자료구조
- 장점 : 자료의 추가 / 삭제 => 결과를 상수 시간 내에 계산
- 단점 : 데이터 검색 => 선형시간(linear time)

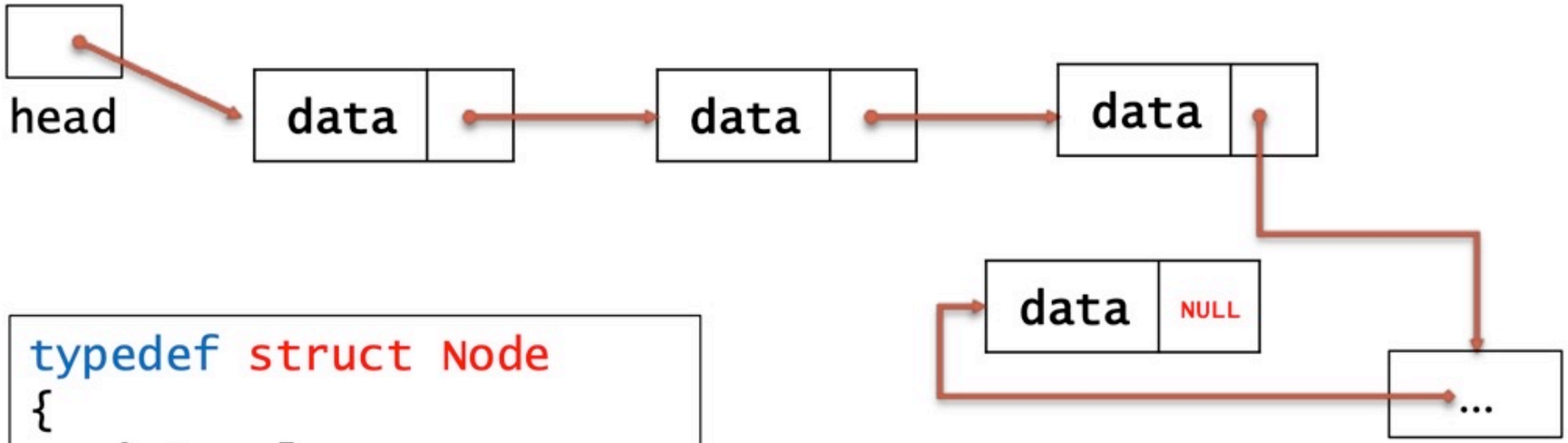
연결리스트(linked list)

연결리스트의 구성요소

- 머리(head) : 연결리스트에서 첫 번째 노드의 주소를 저장한 포인터
- 노드(node) : 연결리스트의 요소
- 값(value) : 노드에 저장된 데이터
- 연결(link) : 다른 노드와의 연결을 위한 포인터

단일 연결리스트(singly linked list)

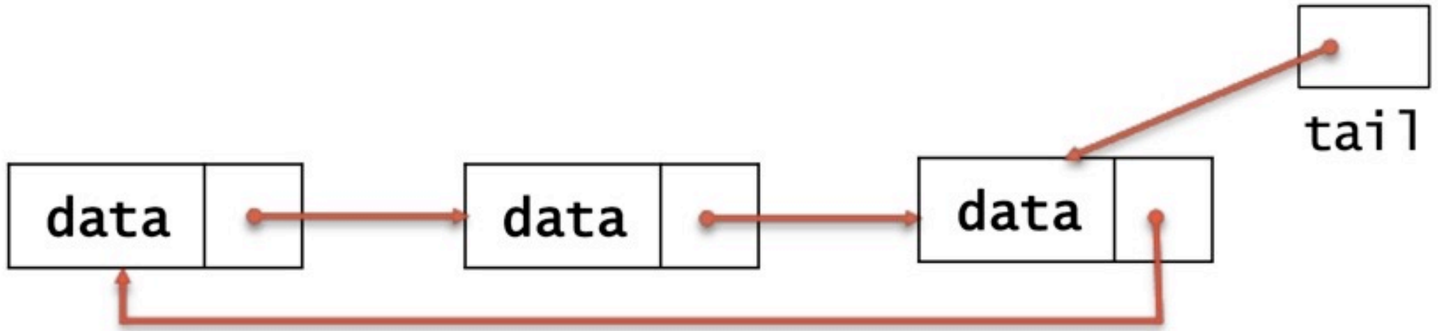
- 각 노드에 다음 노드에 대한 참조가 있다
- 마지막 노드의 연결 부분은 NULL값을 가진다




```
typedef struct Node
{
    int val;
    struct Node* pNext;
}NODE;
```

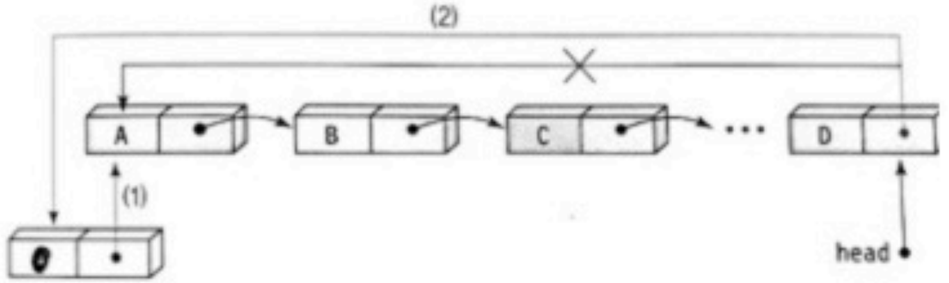
원형 연결리스트(circular linked list)

- 리스트의 마지막 노드의 링크가 첫 번째 노드를 가리키는 연결리스트

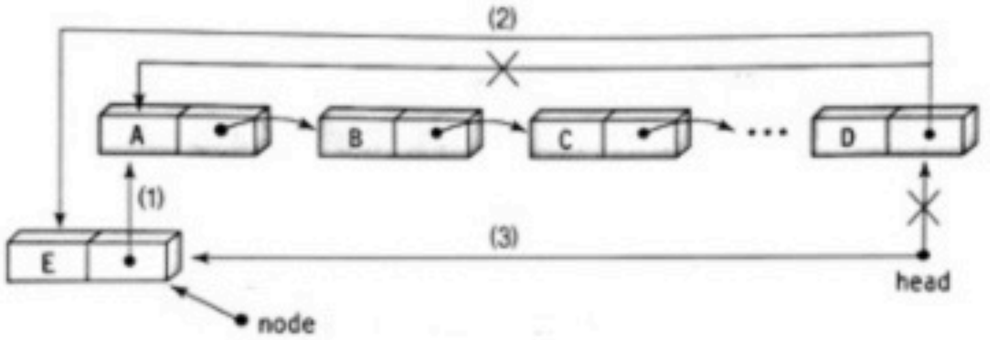



```
typedef struct Node
{
    int val;
    struct Node* pNext;
}NODE;
```

```
void insert_first(ListNode **phead,    ListNode *node)
{
    if( *phead == NULL ){
        *phead = node;
        node->link = node;
    }
    else {
        node->link = (*phead)->link;
        (*phead)->link = node;
    }
}
```



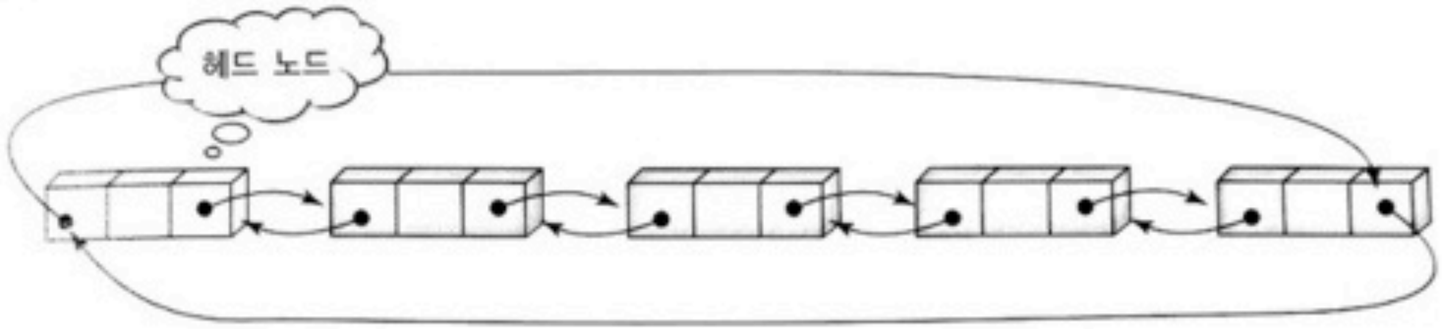
```
// phead: 리스트의 헤드 포인터의 포인터
// node : 삽입될 노드
void insert_last(ListNode **phead, ListNode *node)
{
    if( *phead == NULL ){
        *phead = node;
        node->link = node;
    }
    else {
        node->link = (*phead)->link;
        (*phead)->link = node;
        *phead = node;
    }
}
```

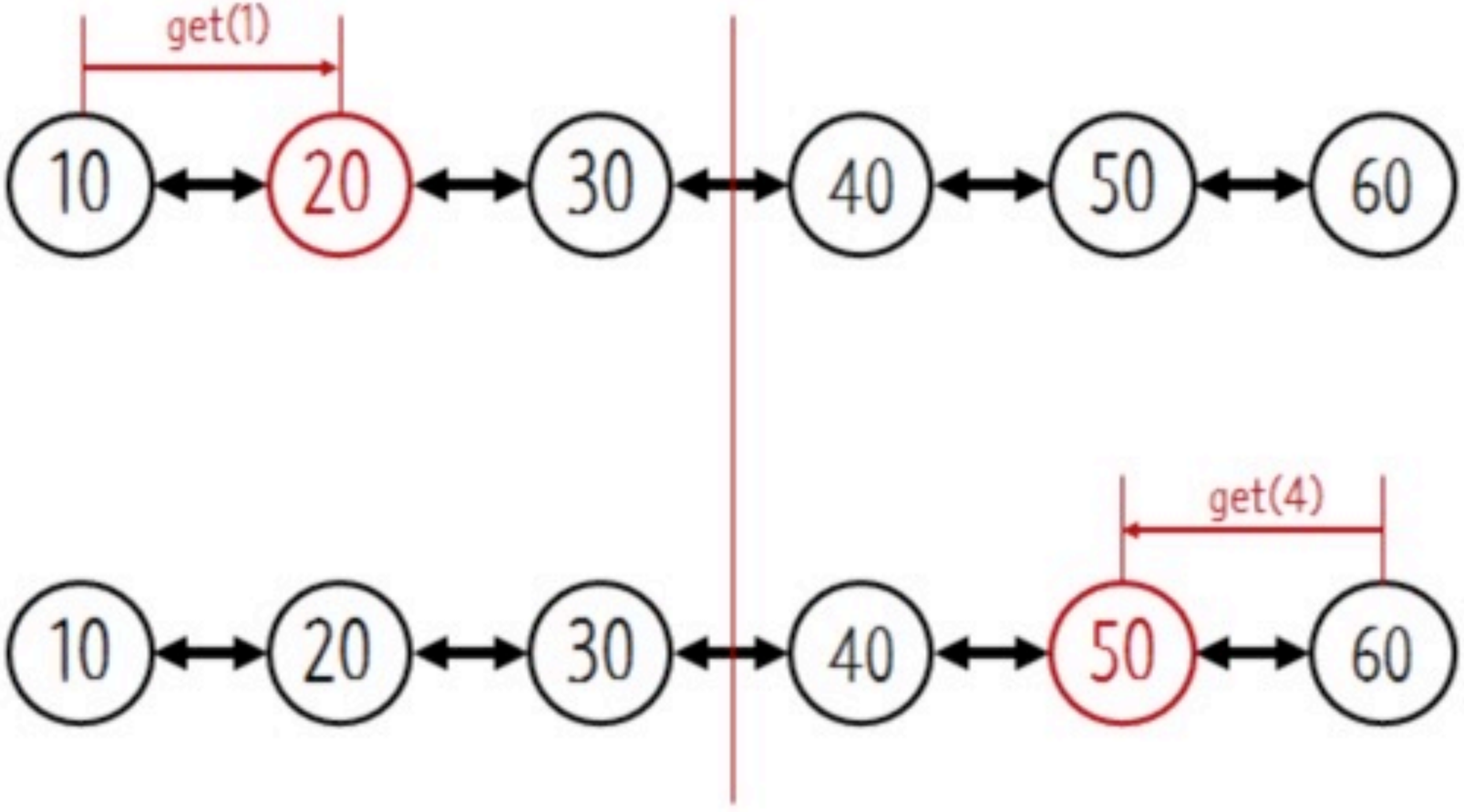


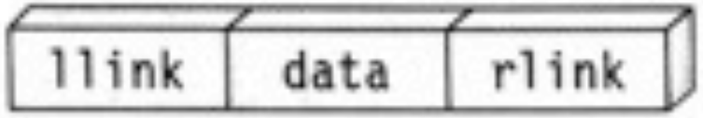
```c
void display(ListNode *head)
{
    ListNode *p;
    if( head == NULL ) return;

    p = head;
    do {
        printf("%d->", p->data);
        p = p->link;
    } while(p != head);
}
```

이중 연결리스트(doubly linked list)

- 하나의 노드가 선행노드와 다음노드에 대한 두 개의 링크를 가지는 연결리스트
- 이중연결리스트는 헤드노드라는 특별한 노드를 가진다
- 헤드노드란 데이터를 가지지 않는 노드로써 첫 번째 노드를 가리킨다




get(1)
10
20
30
40
50
60
get(4)
10
20
30
40
50
60


llink
data
rlink


헤드 노드

(2) 이중 연결 리스트의 관계

-> p == p -> llink -> rlink == p -> rlink -> llink

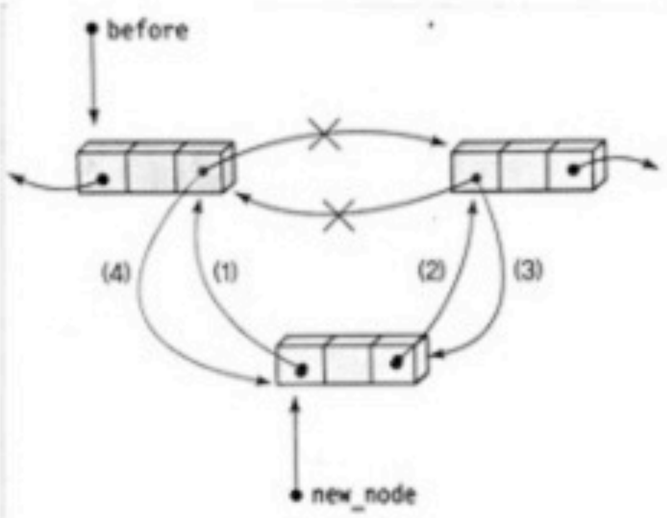
-> 어떤 노드를 가리키는 포인터 p가 자신의 왼쪽링크를 갔다가 오른쪽 링크를 가는 것은 p 자기자신을 가리킨다.

-> 어떤 노드를 가리키는 포인터 p가 자신의 오른쪽 링크를 갔다가 왼쪽링크를 가는 것은 p 자기자신을 가리킨다.

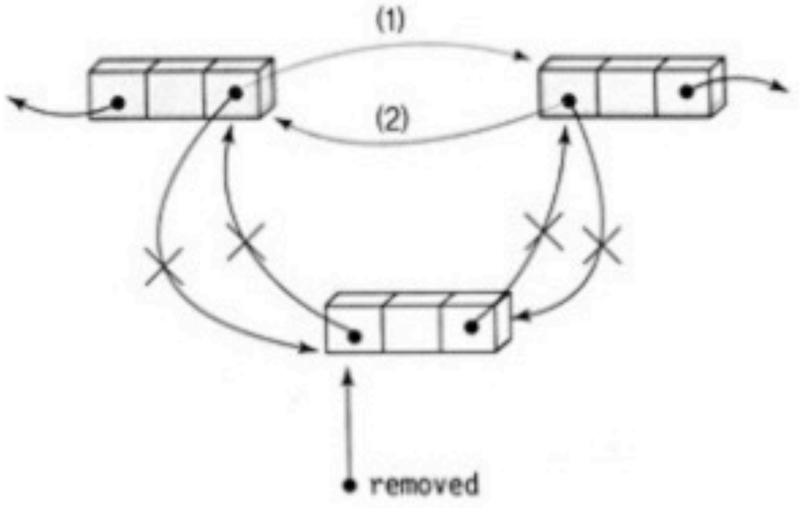
-> 즉 이중 연결 리스트는 양방향 링크를 이용하여 자기자신으로 돌아올 수 있다.

(3) 이중 연결 리스트의 정의

```
typedef int element;
typedef struct DlistNode {
    element data;
    struct DlistNode *llink;
    struct DlistNode *rlink;
} DlistNode;
```



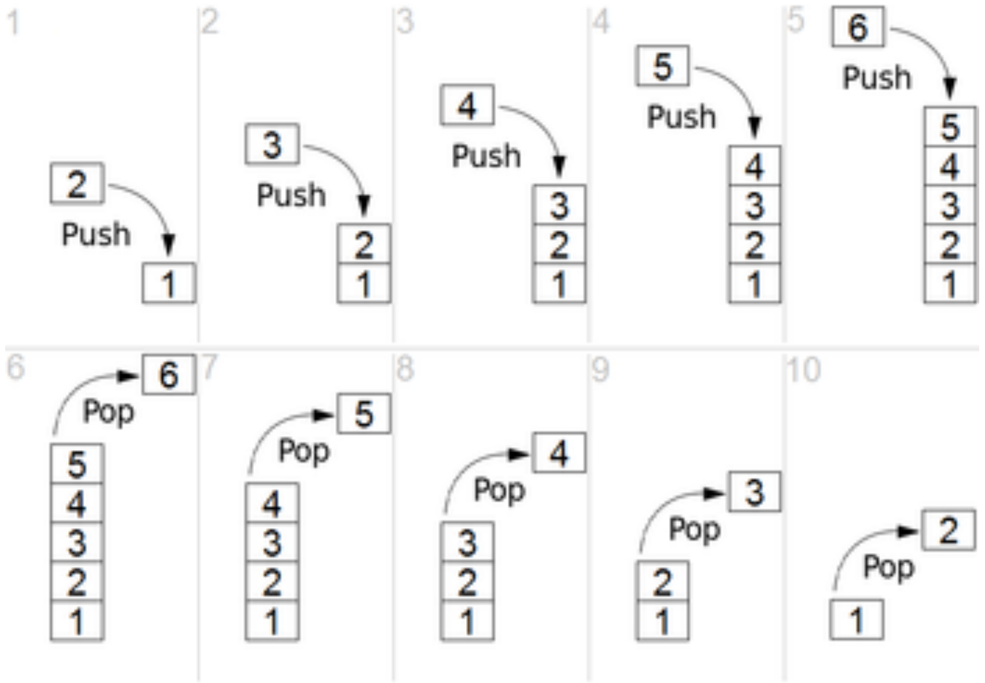

```
// 노드 new_node를 노드 before의 오른쪽에 삽입한다.
void dinsert_node(DlistNode *before,   DlistNode *new_node)
{
    new_node->llink = before;
    new_node->rlink = before->rlink;
    before->rlink->llink = new_node;
    before->rlink = new_node;
}
```




```
// 노드 removed를 삭제한다.
void dremove_node(DlistNode *phead_node,
                  DlistNode *removed)
{
    if( removed == phead_node ) return;
    removed->llink->rlink = removed->rlink;
    removed->rlink->llink = removed->llink;
    free(removed);
}
```

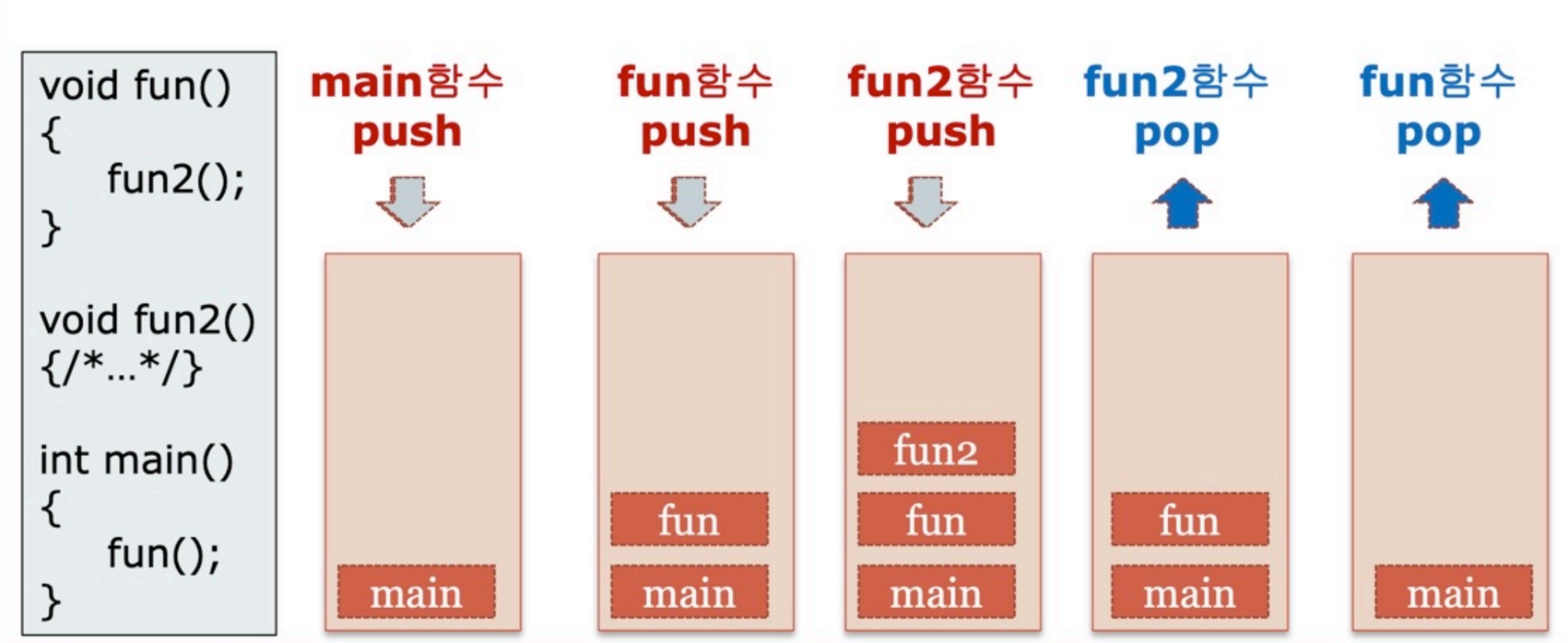
스택(stack) 이란

- 한 쪽 끝에서만 자료를 넣고 뺄 수 있는 LIFO(Last In First Out) 형식의 자료구조
- 함수의 호출, 웹페이지의 URL 스택으로 관리
- push(x) : stack top에 추가
- pop() : stack top의 데이터 반환 및 삭제
- top : push, pop 하는 위치 나타내는 포인터 변수
- isEmpty() : 스택이 비어있을 때 true 반환



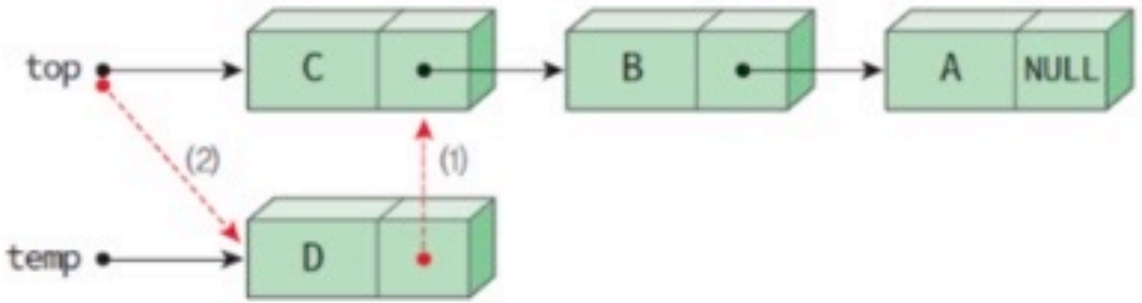
function call stack

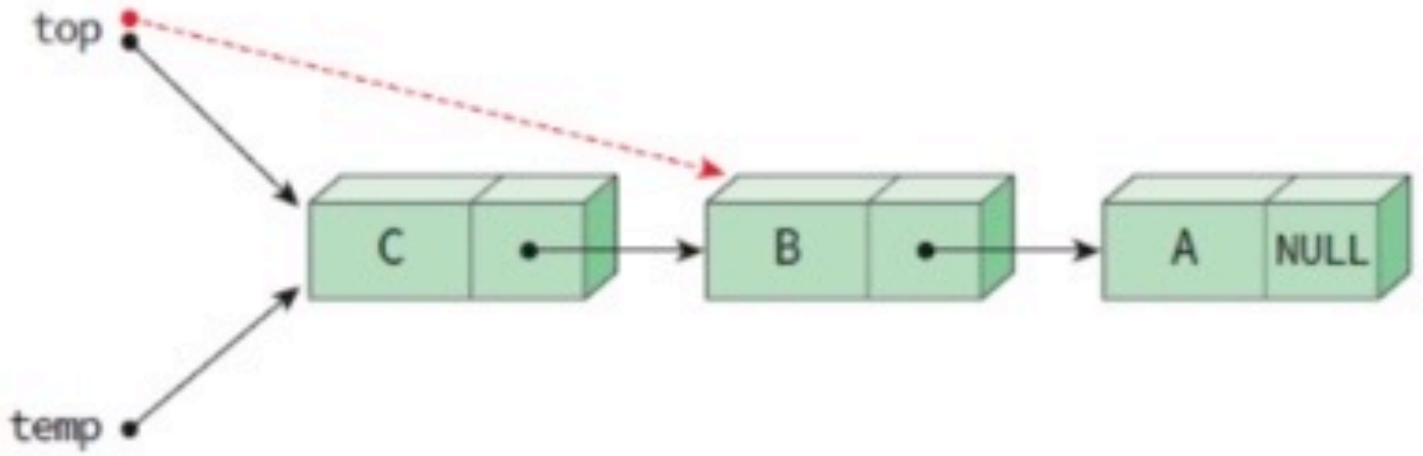
- C언어 프로그램에서 함수를 호출하고 실행할 때 프로그램 내부에서 스택 사용
- 함수가 호출될 때 다음 명령의 주소를 스택에 push한다
- 함수가 종료될 때 다음 명령의 주소를 스택에서 pop한다



스택, 큐 구현

- 배열(array) / 연결리스트(linked list) 로 구현 가능
- 배열은 인덱스로 접근 가능하기 때문에 접근 속도에서 장점, 하지만 변경이 발생했을 때 비효율적
- 연결리스트는 변경 빠름, 하지만 메모리가 연속된 공간에 존재하지 않기 때문에 데이터를 찾으려면 head node부터 순회해야하므로 검색속도가 느림
- => 잦은 검색이 필요하고 메모리가 처음 만들어진 상태로 픽스 : 배열 사용
- => 검색보다는 데이터 등록, 수정이 빈번하게 일어남 : 연결리스트 사용


top
C
B
A
NULL
(2)
(1)
temp
D


top
C
B
A
NULL
temp

```c
// 삽입
void push(LinkedStackType* s, element item) {
    StackNode* temp =
        (StackNode*)malloc(sizeof(StackNode));
    temp->data = item;
    temp->link = s->top;
    s->top = temp;
}
// 삭제
element pop(LinkedStackType* s)
{
    if (is_empty(s)) {
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else {
        StackNode* temp = s->top;
        element data = temp->data;
        s->top = s->top->link;
        free(temp);
        return data;
    }
}
```

큐(queue) 란

- 한 쪽에서 자료를 넣고 다른 한 쪽에서 뺄 수 있어서 먼저 넣은 데이터가 먼저 나오는 FIFO(First In First Out)
- 큐를 사용하는 방식에는 선형 큐, 원형 큐, 덱
- enqueue(x) : 데이터를 리스트의 끝 부분에 추가
- dequeue() : 리스트의 첫 번째 항목 제거
- front : pop하는 위치
- rear : push하는 위치
- is_Empty() : 큐가 비어있을 때 true 반환

```c
void Enqueue(Queue *queue, int data)
{
    Node *now = (Node *)malloc(sizeof(Node)); //노드 생성
    now->data = data;//데이터 설정
    now->next = NULL;

    if (IsEmpty(queue))//큐가 비어있을 때
    {
        queue->front = now;//맨 앞을 now로 설정
    }
    else//비어있지 않을 때
    {
        queue->rear->next = now;//맨 뒤의 다음을 now로 설정
    }
    queue->rear = now;//맨 뒤를 now로 설정
    queue->count++;//보관 개수를 1 증가
}

int Dequeue(Queue *queue)
{
    int re = 0;
    Node *now;
    if (IsEmpty(queue))//큐가 비었을 때
    {
        return re;
    }
    now = queue->front;//맨 앞의 노드를 now에 기억
    re = now->data;//반환할 값은 now의 data로 설정
    queue->front = now->next;//맨 앞은 now의 다음 노드로 설정
    free(now);//now 소멸
    queue->count--;//보관 개수를 1 감소
    return re;
}
```

선형 큐

- 배열을 선형으로 시용하여 구현된 큐

- 인덱스가 감소하지 않고 증가만 하면서 사용하는 방식 => 만약 push 다음 pop이 발생해서 첫 번째 배열이 비어있을 때에도 비어있는 공간 활용 X => 선언된 배열의 크기가 다 찰 때까지 삽입하기 위해서는 데이터 이동이 필요

원형 큐

- 선형 큐의 대안
- front : 첫 번째 요소 바로 앞의 인덱스
- rear : 마지막 요소의 인덱스

```c
void init_queue(QueueType *q){
    q->front = q->rear = -1;
}

int is_Empty(QueueType *q){
    return (q->front >=  q->rear);
}

int is_Full(QueueType *q){
    return (q->rear)-(q->front)>=MAX_QUEUE_SIZE;
}
```

```c
void enqueue (QueueType *q, element item){
    if(is_Full(q)){
        printf("overflow!\n");
        return;
    }
    q->rear++;
    q->data[(q->rear)%MAX_QUEUE_SIZE]=item;
}

void dequeue(QueueType *q, element * item){
    if(is_Empty(q)){
        printf("underflow!\n");
        return;
    }
    q->front++;
    *item=q->data[(q->front)%MAX_QUEUE_SIZE];
}
```