Title :

Name :

Date :

Linked List implemented Stack

1. 링크드 리스트 이므로 Node 필요

Node는 당연히 Data 저장 메모리

다음 Node를 가리키는 포인터 메모리 필요

```
⇒ typedef struct nodeStruct {
        char* Data ;          → 문자열 저장하는 Stack
        struct nodeStruct* Next ;
  } Node ;
```

* 자동 메모리 = Stack Memory

→ 주로 지역변수가 Stack 영역에 저장되며, 해당 지역 종료 시 자동으로 메모리가 해제됨

* 자유 메모리 = Heap Memory

→ 주로 동적으로 메모리 할당 시 저장되는 영역. 개발자가 직접 해제해야함.

Title :

Name :

Date :

2. Linked List 이므로 Head, Tail 포인터 필요

⇒ typedef struct LLS_struct {

Node * List // Head

Node * Tail // Tail

} LinkedListStack; → LLS로 줄여씀

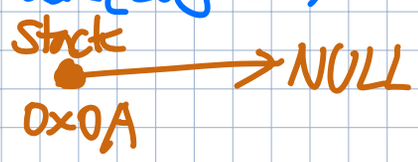
→ 둘다 포인터 이므로 Node의 주소를 가리키게 된다.

3. LinkedListStack 생성

→ LinkedListStack* Stack; // LLS를 가리키는 포인터

→ LLS_CreateStack(&Stack) → 해당 포인터 주소를 넘김

자체의

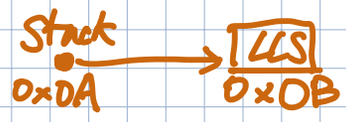


⇒ void LLS_CreateStack(LinkedListStack** Stack)

포인터 주소가 자체의 들어오므로

(*Stack) = (LLS*) malloc(sizeof(LLS));

포인터가 가리키는 위치를 동적으로 할당



(*Stack) → List = NULL;

〃 → Tail = NULL;

} 아직 아무 Node도 없으니까

Title :

Name :

Date :

4. LLS Destroy 방법

```
while ( !LLS_IsEmpty (Stack)) {   스택이 빌때까지
        Node * popped = LLS_POP(Stack);
        free(popped);   // Node 해제
}
free (Stack);   // 마지막으로 Stack 포인터 해제
```

5. LLS_POP 방법

```
if(Stack → List = Stack → Top) {   // 하나밖에 안남았다면
        Stack → List, Tail = NULL;
        return Stack → Top
}
else {
  ① Top 이전까지 Current 이동
   Node* Current = Stack → List;
   while( Current != NULL && Current → Next != Stack → Top)
        Current = Current → Next;

  ② Top 갱신
   Stack → Top = Current;
   Stack → Top → Next = NULL;

   return Stack → Top;
}
```