Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 장성환
redmk1025@gmail.com

* STACK 구조

늦게 들어가는 것이 빨리 나오는 후입 선출의 구조를 가진 자료구조의 형태이다.

구조체를 data 로 하는 Stack 자료 구조 만들기

- 기본 함수 설계

GET_NODE{
메모리 동적 할당을 통하여 힙 영역에 메모리를 생성하고 그 주소를 리턴한다.
}
PUSH{
생성된 힙 영역 주소를 TOP 주소에 담아 항상 스택값을 갱신한다.
해당 함수에 들어온 값을 생성된 힙 영역 주소에 삽입하고 이전 힙 영역 주소를 삽입한다.
}
POP{
해당 TOP 주소의 값을 참조하여 data 를 끄집어 내어 반환하고
TOP 주소를 이전 스택의 주소로 바꾼다.
그리고 값이 추출된 힙 영역은 메모리 해제하여 메모리 누수를 방지한다.
}

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


typedef struct _data{
    char name[30];
    int number;
}data;
typedef data* DATA; //구조체를 스택의 데이터로 만들기 위하여
```

```
typedef struct _stack{
    DATA data;
    struct _stack *link;
}STACK; // 스택 구조체 생성

STACK* GET_NODE(){
    STACK *node;
    node=(STACK*)malloc(sizeof(STACK));
    node->link=NULL;
    return node;
}

void PUSH(STACK **TOP,DATA i_data){
    STACK *tmp;
    tmp = *TOP; //이전 힙 영역 주소값 저장하기 위해서.
    *TOP=GET_NODE(); //현재 생성 힙 주소값 저장
    (*TOP)->data = i_data;
    (*TOP)->link = tmp;
}

DATA POP(STACK **TOP){
    DATA tmp_data;
    STACK *tmp; //
    tmp = *TOP; //

    if(*TOP==NULL){
        printf("저장된 스택이 없습니다.\n");
        return 0; //
    }

    tmp_data = (*TOP)->data;
    *TOP = (*TOP)->link;
```

```
    free(tmp);//
    return tmp_data;
}

int main(void){
    STACK *TOP =NULL;

    DATA name_card; //포인터를 생성했으나 해당 주소값을 넣지 않아서 오류 발생.
    name_card = (DATA)malloc(sizeof(data)); //이렇게 포인터는 주소값을 넣어 초기화 해주어야 한다.

    strcpy(name_card->name,"sunghwan");
    name_card→number=1;*

    PUSH(&TOP,name_card);
    PUSH(&TOP,name_card);

    printf("%s and %d\n",POP(&TOP)->name,POP(&TOP)->number);

    return 0;
}
```

***QUEUE 구조**

먼저 들어간 것이 먼저 나오는 선입선출 형태의 자료구조.

-함수 설계
힙에 저장된 메모리 영역을 차례로 전체 순회한다.
항상 head 는 힙에 저장된 초기값을 가리킨다.
재귀함수로 데이터 저장을 구현(enqueue 재귀)
enqueue 는 입력 데이터 저장 및 주소를 연결을 담당한다.

수업시간의 프린트 함수 구현

```
void print_queue(queue *head){
        queue *tmp = head;
        while(tmp){
                printf("%d\n",tmp->data);
                tmp = tmp->link;
        }
}
```

ENQUEUE 와 DEQUEUE 그리고 전체 데이터 PRINT 코드 구현

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

typedef struct _queue{
   int data;
   struct _queue *link;
}QUEUE;


QUEUE *GET_NODE(){
```

```
    QUEUE * node;
    node = (QUEUE*)malloc(sizeof(QUEUE));
    node->link = NULL;
    return node;
}

void ENQUEUE(QUEUE **head, int data){ //첫 번째 전달되는 head 는 첫 번째 노드를 가리킨다.

    if(*head==NULL){ // GET_NODE 함수에서 메모리 할당 시, link 가 NULL 이라는 것과 처음 head 의 값도 NULL 이라는 조건을 이용
        *head=GET_NODE(); //동적 메모리 할당 링크값은 NULL 로 초기화한다.
        (*head)->data = data; //데이터 삽입

        return;
    }
    else{
        ENQUEUE(&(*head)→link,data);  //재귀 호출을 하며 노드의 각 링크의 주소값을 더블 포인터로 계속 전달한다.
    }
}

void QPRINTF(QUEUE **head){
    if(*head==NULL){
        return;
    }
    printf("%d\n",(*head)->data);
    QPRINTF(&(*head)->link);
}

void DEQUEUE(QUEUE **head,int comp){

    QUEUE *tmp = *head; // 링크값(다른 노드를 가리키는)을 순차적으로 tmp 로 받는다.
    if(tmp->data == comp){  //각 노드의 data 값과 처음 전달 된 data 값을 비교한다.
        *head=tmp→link;  // 현재 전달된 링크값에 링크가 가리키는 노드의 링크값을 삽입한다. (2 번 노드를 삭제시에, 1→2 → 3 일때, 1→ 3 이 되도록)
        free(tmp); // 삭제할 노드를 메모리 반환
```

```
    }
    else{
        DEQUEUE(&(tmp→link),comp); //각 노드의 링크의 주소값 순차적 전달을 재귀구현
    }
}

int main(void){

    QUEUE *head =NULL;

    ENQUEUE(&head,10);
    ENQUEUE(&head,20);
    ENQUEUE(&head,30);

    QPRINTF(&head);

    DEQUEUE(&head,10);
    QPRINTF(&head);


    return 0;
}
```