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

* 자료구조 코드 비교분석 STACK(MY)

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 typedef struct _stack{
 5     int data;
 6     struct _stack *link;
 7 }stack;
 8
 9 stack *get_node(){
10     stack *tmp=(stack*)malloc(sizeof(stack));
11     tmp->link=NULL;
12     return tmp;
13 }
14
15 void push(stack **top,int data){
16     stack *tmp=*top;
17     *top=get_node();
18     (*top)->data=data;
19     (*top)->link=tmp;
20 }
21
22 void pop(stack **top){
23     stack *tmp=*top;
24     tmp->data = (*top)->data;
25     *top = (*top)->link;
26     printf("delete value is %d\n",tmp->data);
27     return;
28 }
29
30 void print_stack(stack *top){
31    if(top==NULL)
32        return;
33     printf("%d\n",top->data);
34     print_stack(top->link);
35 }
```

* STACK (LEC)

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

#define EMPTY 0

struct node{
    int data;
    struct node *link;
};

typedef struct node Stack;

Stack *get_node(){
    Stack *tmp;
    tmp=(Stack *)malloc(sizeof(Stack));
    tmp->link=EMPTY;
    return tmp;
}

void push(Stack **top,int data){
    Stack *tmp;
    tmp = *top;
    *top = get_node();
    (*top)->data=data;
    (*top)->link=tmp;
}

int pop(Stack **top){
    Stack *tmp;
    int num;
    tmp = *top;
    if(*top==EMPTY){
        printf("Stack is empty\n");
        return 0;
    }
```

```
36
37 int main(void){
38     int data;
39     stack *top = NULL;
40     push(&top,10);
41     push(&top,20);
42     push(&top,30);
43     print_stack(top);
44
45     pop(&top);
46     print_stack(top);
47     pop(&top);
48     print_stack(top);
49     pop(&top);
50     print_stack(top);
51     return 0;
52 }
```

STACK 은 기존에 설명해준 코드와 차이가 없었다.
다만 POP 에서 동적 메모리 할당을 해제해 주는것을 까먹지 말도록 하자.

```
    num = tmp->data;
    *top =(*top)->link;
    free(tmp);
    return num;
}

int main(void){
    Stack *top = EMPTY;
    push(&top,10);
    push(&top,20);
    push(&top,30);
    push(&top,40);

    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));

    return 0;
}
```

*QUEUE (MY) -

처음부터 다시 작성할 때, 문제점이 있었다.

재귀함수에 대한 이해가 부족해서 리턴을 이용하여 재귀 탈출을 잘 해줘야 하는데 그렇게 하지 못해서 재귀가 무한으로 호출이 되었다.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 typedef struct _queue{
 5     int data;
 6     struct _queue *link;
 7 }queue;
 8
 9 queue* get_node(void){
10     queue *tmp = (queue*)malloc(sizeof(queue));
11     tmp->link=NULL;
12     return tmp;
13 }
14
15 void enqueue(queue **head, int data){
16     if(*head==NULL){
17         *head=get_node();
18         (*head)->data = data;
19         return; // 매우 중요한 부분. 재귀는 리턴을 잘 활용 해야 한다.
20     }
21     enqueue(&(*head)->link,data);
22 }
23
24 void printf_queue(queue *head){
25
26     if(head!=NULL){
27         printf("%d\n",head->data);
28         printf_queue(head->link);
29     }
```

*QUEUE(LEC)

기존 코드와 다른점은 dequeue 함수에서

나는 더블 포인터를 인자로 주고, 강의에서는 포인터를 인자로 보냈다.

```
queue *dequeue(queue *head, int data){
 40     queue *tmp = head;
 41     if(tmp == NULL)
 42         printf("There are no data that you delete\n");
 43     if(head->data != data)
 44         head->link = dequeue(head->link,data);
 45     else{
 47         printf("Now you delete %d \n",data);
 48         free(tmp);
 49         return head->link;
 50           }
 51         return head;
 52 }


void dequeue(queue **head, int data){
 34     if((*head)->data == data){
            queue *dmalloc = *head;
 35         printf("you delete %d\n",(*head)->data);
 36         (*head)=(*head)->link;
            free(dmalloc)
 37         return;
 38     }
 39     dequeue(&(*head)->link,data);
 40 }
```

```
30     return;
31 }
32
33 void dequeue(queue **head, int data){
34     if((*head)->data == data){
           queue *dmalloc = *head;
35         printf("you delete %d\n",(*head)->data);
36         (*head)=(*head)→link;
           free(dmalloc) 다음과 같은 방식으로 메모리 해제를 하도록 하자.
37         return;
38     }
39     dequeue(&(*head)->link,data);
40 }

41 int main(void){
42     queue *head = NULL;
43
44     enqueue(&head,10);
45     enqueue(&head,20);
46     enqueue(&head,30);
47     enqueue(&head,40);
48     printf_queue(head);
49
50     dequeue(&head,30);
51     printf_queue(head);
52
53     return 0;
54 }
```

인자를 더블 포인터로 받는 것이랑 그냥 포인터로 받는 것에서
코드상의 차이점은 재귀함수를 호출할때의 형태를 보면 알 수 있다.

싱글포인터
head→link = dequeue(head→link, data)

더블 포인터
dequeue(&(*head)→link,data)

즉 싱글 포인터는 변경되는 값을 리턴 값으로 하여 재귀 호출
더블 포인터는 재귀함수는 인자를 전달하는 용도로만 사용하였다.
그리고 노란색 마커로 한 부분을 빼도 컴파일러가 잘 작동하여 혼란이 많이 왔는데
컴파일러 상에서 알아서 똑똑하게 처리가 잘 된것으로 보인다.

사실은 명시적으로 해 놓아야 된다는 것!

*TREE (MY)

강의의 코드와 내가 작성한 코드가 다른 부분은 다음과 같다.

```
21 int judge_null(tree *root){
22     if(root->left == NULL && root->right != NULL){
23         return 2;
24     }
25     else if(root->left != NULL && root->right == NULL){
26         return 1;
27     }
28     else if(root->left !=NULL && root->right !=NULL){// both no null
29         return 3;
30     }
31     else{// both null
32         return 4;
33     }
34 }

36 tree *findmax(tree *root){//input left address
37     if(root->right !=NULL)
38         findmax(root->right);
39     else
40         return root;
41 }

       void delete_tree(tree **root, int data){
76
78     if((*root)->data < data){ //bigger source data
79           delete_tree(&(*root)->right,data);//move right
80     }
81     else if((*root)->data > data){//smaller source data
82         delete_tree(&(*root)->left,data); //move left
83     }
```

*TREE (LEC)

```
49 tree *chg_node(tree *root){
50     tree *tmp =root;
51
52     if(!root->right)
53         root = root->left;
54     else if(!root->left)
55         root = root->right;
56     free(tmp);
57
58     return root;
59 }  //chg_node 함수

61 tree *find_max(tree *root, int *data){
62     if(root->right)
63         root->right = find_max(root->right, data);
64     else{
65         *data = root->data;
66         root = chg_node(root);
67     }
68     return root;
69 } //find max 함수

71 tree *delete_tree(tree *root, int data){
72     int num;
73     tree *tmp;
74     if(root == NULL){
75         printf("Not Found\n");
76         return NULL;
77     }
78     else if(root->data > data){
```

```
84    else{// same value
85
86        if(judge_null(*root)==1){//left sub
87            printf("left sub\n");
88            *root=(*root)->left;
89        }
90        else if(judge_null(*root)==2){//right sub
91            printf("right sub\n");
92            *root=(*root)->right;
93        }
94        else if(judge_null(*root)==3){//both sub
95            printf("both sub\n");
96            tree *tmp=findmax((*root)->left);
97            (*root)->data=tmp->data;
98            (*root)->left->right=tmp->left;
99        }
100       else{//no sub
101           printf("no sub\n");
102           *root=NULL;
103           free(*root);
104           return;
105       }
106       return;
107   }
108 }
```

```
79        root->left = delete_tree(root->left,data);
80    }
81    else if(root->data <data){
82        root->right = delete_tree(root->left,data);
83    }
84    else if(root->left && root->right){
85        root->left = find_max(root->left,&num);
86        root->data = num;
87    }
88    else
89        root = chg_node(root);
90    return root;
91 } // 트리 삭제 함수
92
```

*AVL 코드

```
avl* get_node(){
    avl * node = (avl*)malloc(sizeof(avl));
    node->left=NULL;
    node->right=NULL;
    node->lev=0;
    return node;
}

int update_level(avl *root){
    int left = root->left ? root->left->lev : 0;
    int right = root->right ? root->right->lev : 0;

    if(left>right)
        return left+1;
    return right+1;
}

int rotation_check(avl *root){
    int left = root->left ? root->left->lev : 0;
    int right = root->right ? root->right->lev : 0;

    return right - left;
}

int kinds_of_rot(avl *root, int data){
    printf("data = %d",data);

    if(rotation_check(root)>1){
        if(root->right->data > data)
            return RL;
        return RR;
    }
```

* AVL 동작 과정

기본적인 데이터 삽입 과정은 트리와 비슷하나, 밸런스를 판단하고 해당 모델 RR,RL,LR,LL 의 모델을 판단하여 그 모델을 다시 리밸런싱을 통해 완전 이진트리가 될 수 있도록 만든다.

avl_ins 함수가 동작시 avl **root, int data 함수가 인자로 전달 된다.
1. *root 의 값이 NULL 인 경우에는 메모리 동적 할당이 되어 힙 영역의 주소값을 리턴한다.

2. data 값을 비교하여 재귀 호출을 통하여 left 나 right 주소값에 메모리 동적 할당을 할때, 새로 생성된 노드는 리턴이 되고 이전의 노드에서 update_level 함수를 통하여 level 을 설정한다.

3. level 의 설정은 다음과 같다. 노드의 left, right 에 노드가 연결되어 있는지 판단하여 있다면 연결된 노드의 level 값을 저장하고 없다면 0 을 저장한다.
최종적으로 저장된 level 값에 +1 을 하여 리턴을 한다. (이 값이 노드의 level 값으로 갱신이 된다.)

4. 레벨링 이후에 rotation_check 함수를 통하여 해당 노드가 회전이 필요한지 안한지 판단하게 된다. 해당 노드의 left 와 right 의 레벨 값을 받아서
left – right 의 값을 리턴한다. 첫 노드가 left, right 값을 가지지 않는 이상 연속적으로 3 개가 나열이 되면 rotation_check 의 리턴값의 절대값은 2 이다.

5. rotation_check 에서 해당 노드는 회전할 필요가 있다고 판단이 되면, 회전을 시작하게 된다. 이전에 회전의 모델인 LL,LR,RL,RR 을 판단하는 kinds_of_rot 함수를 이용하게 된다.

6. 편의상 첫노드, 둘노드, 삼노드 순으로 트리가 구성되어 있다고 하면,
kinds_of_rot 함수에 첫노드의 주소값과 삼노드의 data 값이 인자로 전달이 된다.
rotation_check 가 양수인지 음수인지에 따라서 처음에 L 인지 R 인지 판단이 된다.

```
    else if(rotation_check(root) < -1){
        if(root->left->data < data)
            return LR;
        return LL;
    }
}

void avl_ins(avl **root, int data){
    if(!(*root)){
        (*root) = get_node();
        (*root)->data = data;
        return;
    }

    if((*root)->data > data)
        avl_ins(&(*root)->left,data);
    else if((*root)->data < data)
        avl_ins(&(*root)->right,data);

    (*root)->lev = update_level(*root);

    if(abs(rotation_check(*root))>1){
            printf("Rotation !\n");
            *root = rotation(*root, kinds_of_rot(*root,data));
    }
}
```

그리고 첫노드의 주소값을 통하여 두번째 노드의 데이터 값과 삼노드의 데이터 값을 비교하여 이후에 L 인지 R 인지 판단하여 LL, RR, LR, RL 을 반환하게 된다.

7. 해당 모델 타입에 따라서 회전을 통해 리밸런싱을 완성한다.