

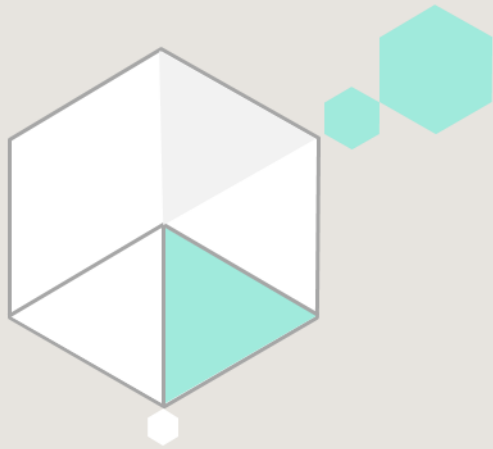
TI DSP, MCU, Xilinx Zynq FPGA 기반의 프로그래밍 전문가 과정

강사 – Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 – 김형주

mihaelkel@naver.com



CONTENTS

Algorithm – Data Structure


- I What's algorithm?
- II Stack
- III Queue
- IV Binary Search Tree
- V AVL Tree
- VI Red-Black Tree



What's algorithm?

Definition :

al·go·rithm

/ˈalgəˌrɪθəm/ 

noun

a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

"a basic algorithm for division"

Description :

Simply, Algorithm is for making what you think of. But there are a lot of methods to do it. If You consider yourself as a programmer ,I mean not a coder, you must think of many things in addition function. Generally there are time complexity and space complexity(Auxiliary space). But these days, the time complexity is more important because of advanced memory technology.

All data structure has a node pointing the next data. So, recursive function is the most simple way to implement that. But when you call a function (using recursive function), 'Pipeline' be broken by using 'callq'. 'for(;;)' or 'while()' also breaks 'Pipeline' by using 'jmp' of course.

```
0x000000000040070a <+52>:    callq  0x40076a <enqueue>    0x00000000004004e1 <+11>:    jmp     0x4004e7 <main+17>
```

but using recursive function be called a lot of times. More data, much more calling. I programed 'Queue data structure' by recursive-function and non-recursive-function, to compare how many time be used.

```
6,300 times inserted!  
res = 203sec (20 times try)
```

```
25,000,000 times inserted!  
res = 110sec (20 times try)
```

During 10.2 seconds, recursive queue received 6,300 data. On the other hand, non-recursive queue received 25,000,000 data during 5.5 seconds, non-recursive algorithm is much faster.

In addition to recursive function, there are many things to efficiently implement a program. For instance, `if(A || B){}` and `if(B || A){}` both perform same result, but execution time can be different. If A is often true, the former is preferred.



Stack

```
#include <stdio.h>
#include <stdlib.h>
#define element int
typedef struct __stack{
    element data;
    struct __stack* p_node;
}stack;

stack* get_node(void);
void push(stack** top,element data);
int pop(stack** top);
void disp_stack(stack **top);
int main(void){
    stack* top = NULL;
    char ch;
    element data;
    while(1){
        printf("Instruction quit(q), insert(i), pop(d), disp(d) :");
        scanf("%c%c",&ch);
        system("clear");
        switch(ch){
            case 'q':
                printf("quit!\n");
                break;
            case 'i':
                printf("data : ");
                scanf("%d%c",&data);
                push(&top,data);
                printf("data %d inserted!\n",data);
                break;
            case 'd':
                printf("data %d deleted!\n",pop(&top));
                break;
            case 'p':
                printf("current stack :\n");
                disp_stack(&top);
                break;
        }
    }
}
```

```
        if(ch=='q')
            break;
    }
    return 0;
}

stack* get_node(void){
    stack* tmp;
    tmp = (stack*)malloc(sizeof(stack)*1);
    tmp->p_node = NULL;
    return tmp;
}

void push(stack** top,element data){
    stack* tmp;
    if(!(*top)){
        *top = get_node();
        (*top)->data = data;
    }
    else{
        tmp = *top;
        (*top) = get_node();
        (*top)->data = data;
        (*top)->p_node = tmp;
    }
}

int pop(stack** top){
    element data;
    if(!(*top)){
        printf("there's no stack!!\n");
        return 0;
    }
    data = (*top)->data;
    *top = (*top)->p_node;
    return data;
}

void disp_stack(stack **top){
    stack* tmp = *top;
    while(tmp){
        printf("%d\n",tmp->data);
        tmp = tmp->p_node;
    }
}
```



Stack

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
data : 10  
data 10 inserted!  
Instruction quit(q), insert(i), pop(d), disp(d) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
data : 20  
data 20 inserted!  
Instruction quit(q), insert(i), pop(d), disp(d) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
data : 30  
data 30 inserted!  
Instruction quit(q), insert(i), pop(d), disp(d) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
data 30 deleted!  
Instruction quit(q), insert(i), pop(d), disp(d) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
current stack :  
20  
10  
Instruction quit(q), insert(i), pop(d), disp(d) :
```



Queue

```
#include <stdio.h>
#include <stdlib.h>
#define element int
typedef struct __queue{
    element data;
    struct __queue* p_node;
}queue;
queue* get_node(void);
void enqueue(queue** top,element data);
int dequeue(queue** top);
void disp_queue(queue** top);
char _getch(void);
int _getData(void);
int main(void){
    queue* top = NULL;
    char ch;
    element data;
    while(1){

        printf("명령어 입력(quit('q'),insert('i'),
display('p'),dequeue('d')) : ");
        ch = _getch();
        system("clear");
        switch(ch){
            case 'q':
                printf("quit!!\n");
                break;
            case 'i':
                printf("input(int type data) : ");
                data = _getData();
                enqueue(&top,data);
                printf("data %d inserted\n",data);
                break;
            case 'p':
                printf("current queue :\n");
                disp_queue(&top);
                break;
```

```
            case 'd':
                printf("dequeue : %d\n",dequeue(&top));
                break;
            default:
                printf("wrong instruction!!\n");
                break;
        }
        if(ch == 'q')
            break;
        printf("\n\n");
    }

    return 0;
}

queue* get_node(void){
    queue* tmp;
    tmp = (queue*)malloc(sizeof(queue)*1);
    tmp->p_node = NULL;
    return tmp;
}

void enqueue(queue** top,element data){
    queue* tmp;
    if(!(*top)){
        tmp = get_node();
        tmp->data = data;
        *top = tmp;
    }
    else
        enqueue(&((*top)->p_node),data);
}

int dequeue(queue** top){
    element data;
    if(!(*top)){
        printf("there's no queue!!\n");
        return 0;
    }
```





Queue

```
        tmp->data = data;
        *top = tmp;
    }
    else
        enqueue(&((*top)->p_node),data);
}
int dequeue(queue** top){
    element data;
    if(!(*top)){
        printf("there's no queue!!\n");
        return 0;
    }
    data = (*top)->data;
    (*top)= (*top)->p_node;
    return data;
}
void disp_queue(queue** top){
    if(*top){
        printf("%d\n",(*top)->data);
        disp_queue(&((*top)->p_node));
    }
    else
        return;
}
char _getch(void){
    char buf,ch;
    while((buf = getchar())!=10)
        ch = buf;
    return ch;
}
int _getData(void){
    int buf,data;
    scanf("%d%c",&data);
    return data;
}
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
input(int type data) : 10
data 10 inserted
```

```
명령어 입력(quit('q'),insert('i'),display('p'),dequeue('d')) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
input(int type data) : 20
data 20 inserted
```

```
명령어 입력(quit('q'),insert('i'),display('p'),dequeue('d')) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
input(int type data) : 30
data 30 inserted
```

```
명령어 입력(quit('q'),insert('i'),display('p'),dequeue('d')) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
dequeue : 10
```

```
명령어 입력(quit('q'),insert('i'),display('p'),dequeue('d')) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test
```

```
current queue :
data : 20
data : 30
```

```
명령어 입력(quit('q'),insert('i'),display('p'),dequeue('d')) :
```



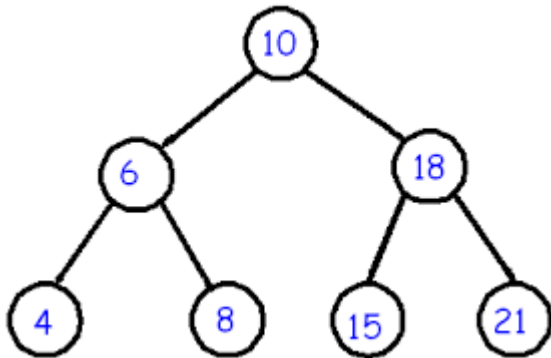


Binary Search Tree(BST)

Characteristics :

Node has 0 – 2 child node. Left child node's data is less than parent node, Right child node has bigger data than parent's.

If it is complete Binary Tree (which means all nodes have 2 child node, except for leaf nodes), searching algorithm's time complexity is $O(\log N)$. And It is much faster than stack or queue's $O(N)$.



In the above tree, total data number = 7. but Searching any data requires less than 4 times searches. So, The biggest feature of BST is fast searching speed.

```
#include <stdio.h>
#include <stdlib.h>
#define element int
typedef struct __tree{
    element data;
    struct __tree *right;
    struct __tree *left;
}tree;
typedef struct __stack{
    element data;
    struct __stack *p_node;
}stack;

tree* get_node(void);
tree* tree_ins(tree** root,element data);
void disp_tree(tree** root);
void del_tree(tree** root,element data);
int main(void){
    tree* root = NULL;
    char ch;
    element data;
    while(1){
        printf("Instruction quit(q), insert(i), delete(d), display(p)");
        scanf("%c%c",&ch);
        system("clear");
        switch(ch){
            case 'q':
                printf("quit!\n");
                break;
            case 'p':
                printf("current tree :\n");
                disp_tree(&root);
                break;
            case 'i':
                printf("data : ");
                scanf("%d%c",&data);
                root = tree_ins(&root,data);
                break;
```




Binary Search Tree(BST)

```
        break;
    case 'd':
        printf("del data : ");
        scanf("%d%c",&data);
        del_tree(&root,data);
        break;
    default:
        printf("wrong instruction!\n");
        break;
}
if(ch == 'q')
    break;
}
}

tree* get_node(void){
    tree* tmp;
    tmp = (tree*)malloc(sizeof(tree)*1);
    tmp->right = NULL;
    tmp->left = NULL;
    return tmp;
}

tree* tree_ins(tree** root,element data){
    tree* b_root = *root;
    tree* prev = *root;
    if(!(*root)){
        *root = get_node();
        (*root)->data = data;
        return *root;
    }
    while(1){
        if(!(*root)){
            (*root) = get_node();
            (*root)->data = data;
            if(prev){
                if(prev->data > (*root)->data)
                    prev->left = *root;
                else
                    prev->right = *root;
            }
        }
    }
}
```

```
        break;
    }
    else if(data < (*root)->data){
        prev = *root;
        (*root) = (*root)->left;
    }
    else if(data > (*root)->data){
        prev = *root;
        (*root) = (*root)->right;
    }
    else{
        printf("duplicated data!!\n");
        break;
    }
}
return b_root;
}

void disp_tree(tree** root){
    if(*root){
        printf("data : %d, ",(*root)->data);
        if((*root)->left)
            printf("left : %d, ",(*root)->left->data);
        else
            printf("left : NULL, ");
        if((*root)->right)
            printf("right : %d\n",(*root)->right->data);
        else
            printf("right : NULL\n");
        disp_tree(&((*root)->left));
        disp_tree(&((*root)->right));
    }
}

void del_tree(tree** root,element data){
    tree* tmp = *root;
    tree* parent = NULL;
    tree* del;
    if(!tmp){
        printf("there is no %d in tree\n",data);
        return;
    }
}
```



Binary Search Tree(BST)

```
while(1){
    if(data > tmp->data){
        parent = tmp;
        if(!tmp->right){
            printf("there is no %d in tree\n",data);
            return;
        }
        tmp = tmp->right;
    }
    else if(data < tmp->data){
        parent = tmp;
        if(!tmp->left){
            printf("there is no %d in tree\n",data);
            return;
        }
        tmp = tmp->left;
    }
    else
        break;
}
del = tmp;

if((!parent)&&(!tmp->left)){
    (*root) = (*root)->right;
    free(tmp);
    return;
}
else if(tmp->left){
    parent = tmp;
    tmp = tmp->left;
    while(1){
        if(tmp->right){
            parent = tmp;
            tmp = tmp->right;
        }
        else
            break;
    }
}
```

```
del->data = tmp->data;

if(tmp->left)
    parent->data < tmp->data ?
    (parent->right = tmp->left) : (parent->left = tmp->left) ;
else{
    parent->data < tmp->data ?
    (parent->right = NULL) : (parent->left = NULL) ;
}

free(tmp);
}
```





Binary Search Tree(BST)

```
howard@ubuntu: ~/HomeworkBackup/ref test  
data : 50  
Instruction quit(q), insert(i), delete(d), display(p) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test  
data : 45  
Instruction quit(q), insert(i), delete(d), display(p) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test  
data : 48  
Instruction quit(q), insert(i), delete(d), display(p) :
```

```
howard@ubuntu: ~/HomeworkBackup/ref test  
current tree :  
data : 50, left : 45, right : NULL  
data : 45, left : NULL, right : 48  
data : 48, left : NULL, right : NULL  
Instruction quit(q), insert(i), delete(d), display(p) :
```





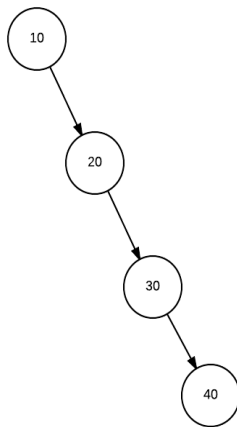
AVL Tree

Characteristics :

assume that received data is Sequence data. The binary search tree has level same as the BST's time complexity is similar to 'Queue' or 'Stack', that is, The biggest strength of BST becomes meaningless.

To make up for this, There is AVL Tree. In AVL Tree, If level-Difference (difference between two children nodes) is bigger than 1, Tree will be rotated to balance the level.

there are 4 cases for rotation. RR, RL, LR, LL.
If you want to implement AVL Tree, You should consider conditions and results of each of cases.



```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define element int

typedef struct __avl{
    element data;
    int lev;
    struct __avl *left;
    struct __avl *right;
}avl;

typedef struct __stack{
    avl* root;
    struct __stack *p_node;
}stack;

avl* get_node(void);
avl* avl_ins(avl** root,element data);
void print_tree(avl** root);
int update_level(avl** root);
void push(stack** top,avl* root);
stack* get_stack_node(void);
avl* pop(stack** top);
int chk_rotate(avl** root);

int main(void){
    avl* root = NULL;

    root = avl_ins(&root,50);
    root = avl_ins(&root,45);
    root = avl_ins(&root,48);

    print_tree(&root);

    return 0;
}
```



AVL Tree

```
avl* get_node(void){
    avl* tmp;
    tmp = (avl*)malloc(sizeof(avl)*1);
    tmp->right = NULL;
    tmp->left = NULL;
    tmp->lev = 1;

    return tmp;
}

avl* avl_ins(avl** root,element data){
    avl* b_root = *root;
    avl* prev;
    avl* tmp = NULL;
    stack* top = NULL;
    if(!(*root)){
        *root = get_node();
        (*root)->data = data;
        return *root;
    }
    else{
        while(1){
            if(!(*root)){
                *root = get_node();
                (*root)->data = data;
                if(prev->data > data)
                    prev->left = *root;
                else
                    prev->right = *root;
                break;
            }
            else if(data < (*root)->data){
                prev = *root;
                push(&top,(*root));
                (*root) = (*root)->left;
            }
        }
    }
}
```

```
        else if(data > (*root)->data){
            prev = *root;
            push(&top,(*root));
            (*root) = (*root)->right;
        }
    }
}

while(1){
    tmp = pop(&top);
    if(!tmp)
        break;

    tmp->lev = update_level(&tmp);
    if(abs(chk_rotate(&tmp)) > 1){
        printf("%d rotate!!\n",tmp->data);

        //rotate and rotated data level update:
    }
}

return b_root;
}

void print_tree(avl** root){
    if(*root){

        printf("data : %d, level : %d, ",(*root)->data,(*root)->lev);
        printf("left : ");
        if((*root)->left)
            printf("%d, ",(*root)->left->data);
        else
            printf("NULL, ");
        if((*root)->right)
            printf("right : %d\n",(*root)->right->data);
        else
            printf("right : NULL\n");
        print_tree(&((*root)->left));
        print_tree(&((*root)->right));
    }
}
```



AVL Tree

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

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

void push(stack** top, avl* root){
    stack* tmp = *top;
    if(!(*top)){
        (*top) = get_stack_node();
        (*top)->root = root;
    }
    else{
        (*top) = get_stack_node();
        (*top)->root = root;
        (*top)->p_node = tmp;
    }
}

stack* get_stack_node(void){
    stack* tmp;
    tmp = (stack*)malloc(sizeof(stack)*1);
    tmp->p_node = NULL;
    return tmp;
}

avl* pop(stack** top){
    if(*top){
        avl* data = (*top)->root;
        (*top) = (*top)->p_node;
        return data;
    }
    return NULL;
}

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

    return right - left;
}
```

재귀함수 없이 AVL Tree 구현 진행 중..

진행상황 :

이진 트리 삽입 구현 후 레벨 업데이트.

남은 작업 :

Rotation 함수 및 delete 구현.

level update시 사용한 Stack or 별도의

Stack 사용하여 avl tree 출력 함수 재귀 없이 구현.

3/13(화)까지는 구현 가능할 것으로 보임.



Red-Black Tree

Characteristics :

Among tree algorithm, AVL Tree has the fastest search speed, Because it is close to perfect binary tree. But, let's think about RL rotation or LR rotation. It has 2 rotations for balancing level. It is too rigorous.

Even though AVL Tree has the fastest speed in searching, inserting time is not. If It has a lot of data, there will be so many rotations. In the system that insertion is frequently occurred, AVL is not the best choice. So, there is RB Tree(Red-Black Tree). RB Tree has 5 rules.

1. The root node always has black color.
2. In the way from the root to any leaf node, there are same number of nodes which has black color.
3. When insertion data's parent node has red color, be rotated or color changed.
4. (3-1) If uncle node has same color as parent node, Don't rotate and change color.
 - >grandfather : black -> red,
 - >parent, uncle : red -> black
 - >child : red
5. (3-2) 4 case is unsatisfied, Don't change color, only do rotate.

아직 프로그래밍 시도 못해봄.

3/13(화) 까지 AVL Tree 구현 후, 3/18(일)까지
재귀함수-RB Tree 및 재귀 없는 RB Tree 구현 예정

