Family Tree

중앙대학교 소프트웨어학과 20186889 권용한

●소스코드 설명

```
typedef struct node *treePointer;
typedef struct node {
    char name;
    treePointer father;
    treePointer mother;
}node;

treePointer get_tree() {
    treePointer *tree;
    tree = (treePointer*)malloc(sizeof(node));
    return tree;
}
```

프로그램에서 사용되는 tree를 구현하고 tree를 동적으로 할당받아 할당받은 주소값을 return하는 함수입니다. tree는 char형의 name과 다른 tree를 가리키는 treePointer형의 father과 mother로 이루어져있습니다.

```
treePointer queue[MAX_QUEUE_SIZE];
int front = 0;
int rear = 0;

void addq(treePointer tree) {
    if ((rear + 1) % MAX_QUEUE_SIZE == front) {
        queue_full();
    }
    rear = (rear + 1) % MAX_QUEUE_SIZE;
    queue[rear] = tree;
}

int queue_full() {
    printf("queue is full, cannot add element");
    exit(EXIT_FAILURE);
}

treePointer deleteq() {
    treePointer x;

    if (front == rear) {
        return NULL;
    }
    front = (front + 1) % MAX_QUEUE_SIZE;
    x = queue[front];
    return x;
}
```

level order 방식으로 값을 출력하고 탐색하기 위해 queue를 사용하는데, 때문에 MAX_QUEUE_SIZE(100)의 크기의 queue를 만들고 queue와 관련된 연산입니다. level order 방식으로 탐색하고 출력하는 과정에서 delete 함수가 호출된 이후 그 값이 null값이 아닌 경우에만 다음 과정이 일어나도록 설계되었기 때문에 delete 연산은 queue가 empty한 경우 종료하는 것이 아닌 null을 리턴합니다.

```
void levelorderprint(treePointer ptr) {
    if (!ptr) return;
    addq(ptr);
    for (;;) {
        ptr = deleteq();
        if (ptr != NULL) {
            printf("%c", ptr->name);
            if (ptr->father != NULL) {
                addq(ptr->father);
            }
            if (ptr->mother != NULL) {
                addq(ptr->mother);
            }
        }
        else break;
    }
}
```

level order 방식으로 입력 받은 tree들을 print하는 함수입니다. ptr변수에는 queue에 삽입되어있는 값을 받아오고,

그 값이 null이 아니라면 ptr 변수중 name값을 출력합니다. 이때 출력되는 값은 subtree의 root입니다.
그리고 그 다음 level의 값들을 queue에 add하고 이때 father을 먼저 add합니다. 이 과정을 queue가 empty하고 ptr에 null이 리턴되는 상황까지 반복하게 됩니다.

```
treePointer levelordersearch(treePointer ptr, char c) {
    treePointer free;
    if (!ptr)
        return NULL;
    addq(ptr);
    for (;;) {
        ptr = deleteq();
        free = ptr;
        if (ptr != NULL) {
            if (ptr->name == c) {
                for (; free != NULL;) {
                    free = deleteq();
                }
                return ptr;
            }
            if (ptr->father != NULL)
                addq(ptr->father);
            if (ptr->mother != NULL)
                addq(ptr->mother);
        }
        else break;
    }
    return NULL;
}
```

level order로 탐색하는 함수입니다. 방식은 위의 level order로 출력하는 경우와 같은 방식으로 코딩이 되었으며, 이 함수의 경우 만일 aFs가 입력된 이후 aMd나 dFa같은 것이 입력되었을 경우, a나 d가 이전에 입력된 값이 어디에 저장되었는지를 파악하기 위해 사용됩니다. 이 함수가 없다면 새로 입력받은 값이 이전의 값들과 연결을 할 수 없기 때문에 필요한 함수입니다.
이때 free라는 변수를 확인할 수 있는데, 이 변수는 만일 queue에 저장된 값들을 검사하던 중 원하는 값을 찾았을 경우, queue와 front, rear가 전역변수이기 때문에 queue에 저장된 값을 모두 제거하지 않는다면 이후 그 다음번에 queue를 사용할 때에도 영향을 주기 때문에 목표값을 찾은 이후 queue의 값들을 delete하기 위해 사용되는 변수입니다.

```
void free_all(treePointer ptr) {
    if (!ptr) return;
    addq(ptr);
    for (;;) {
        ptr = deleteq();
        if (ptr) {
            if (ptr->father != NULL) {
                addq(ptr->father);
            }
            if (ptr->mother != NULL) {
                addq(ptr->mother);
            }
            free(ptr);
        }
        else break;
    }
}
```

이 코드는 동적으로 할당된 tree들이 프로그램 종료 후 모두 free를 하기 위해 작성된 코드로, tree들의 주소를 따로 저장해 두지 않기 때문에 tree들의 주소값을 찾아낼 필요가 있어서 이전에 사용한 level order과 같은 방식으로 tree들을 탐색하고 찾은 tree들을 free하는 역할을 합니다.

```
if (n == 0)
{
    A = get_tree();
    A->name = tree[0];
    A->father = NULL;
    A->mother = NULL;
    if (tree[1] == 'F') {
        A->father = get_tree();
        A->father->father = NULL;
        A->father->mother = NULL;
        A->father->name = tree[2];
    }
    else if (tree[1] == 'M') {
        A->mother = get_tree();
        A->mother->father = NULL;
        A->mother->mother = NULL;
        A->mother->name = tree[2];
    }
    levelorderprint(A);
    printf("₩n");
    n++;
}
```

이 코드부터는 main함수 내부의 코드입니다. 첫 번째 입력받은 경우엔 두 번째 입력받은 경우와는 달리 처음에 입력받은 값에서 2개의 tree를 생성해야 하기 때문에(aFs의 경우 a라는 name을 가진 tree와 s라는 name을 가진 tree를 생성하고, a tree의 father에 s tree의 주소값을 저장합니다.) 첫 번째 입력받은 경우를 따로 분리하여 작성하였습니다. 이때 첫 번째 입력인 경우엔 n이 0이고, 그 이후부터는 n의 값이 0이 아니기 때문에 이것으로 첫 번째 입력과 그 이후의 입력을 구분하였습니다. 그리고 tree가 동적으로 할당된 이후에 tree의 father과 mother이 null로 저장되어있지 않기 때문에 이후 levelordersearch함수나 levelorderprint함수에서 오류가 발생하지 않도록 father과 mother에 null값을 저장합니다. 그리고 F인 경우와 M인 경우를 구분하여 상황에 맞게 값을 저장합니다.

```
else if (n > 0) {
    familytree = levelordersearch(A, tree[0]);
    if (familytree != NULL) {
        if (tree[1] == 'F') {
            familytree->father = get_tree();
            familytree->father->father = NULL;
            familytree->father->mother = NULL;
            familytree->father->name = tree[2];
        }
        else if (tree[1] == 'M') {
            familytree->mother = get_tree();
            familytree->mother->father = NULL;
            familytree->mother->mother = NULL;
            familytree->mother->name = tree[2];
        }
        levelorderprint(A);
        printf("₩n");
        n++;
    }
```

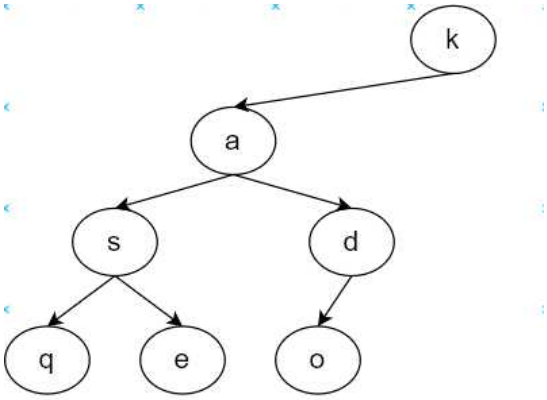
이 부분은 첫 번째 입력 이후 두 번째 입력부터의 과정인데, 예를 들어 aFs가 처음으로 입력되었다고 한다면, 그 다음값이 aMd가 입력될 수도 있고, dFa가 입력될 수도 있을 텐데, 두 경우는 작동하는 방식이 다릅니다.
aMd가 입력된다면 기존에 있던 a의 mother에 새로 동적 할당한 tree의 주소를 저장하고, 새로 동적 할당된 tree의 name에 d값을 저장하지만, dFa의 경우엔 새로 동적 할당을 하고 tree의 name에 d를 저장하고, 동적 할당된 tree의 father에 기존의 a tree의 주소값을 저장해야 합니다. 때문에 이것도 구분하여 코드를 작성하였는데, 두 경우를 구분하기 위해 familytree라는 treePointer형 변수를 선언하여 levelordersearch함수를 통해 입력받은 문자열의 첫 번째 값이 이전에 입력받은 트리중 존재하는 값인지 아닌지를, 만일 존재한다면 그 트리의 주소값을 받아옵니다.
그리고 만일 familytree가 null이 아닌 경우, 즉 입력된 문자열의 첫 번째 값이 이전의 tree에 저장되어 있는 값인 경우, 그 주소값의 tree에 father(두 번째 값이 F인 경우)나 mother(두 번째 값이 M인 경우)에 새로 동적할당을 받고, 그 동적할당을 받은 tree의 name에 세 번째로 입력받은 값을 저장합니다.

```
else {
    familytree = A;
    A = get_tree();
    A->name = tree[0];
    A->father = NULL;
    A->mother = NULL;

    if (tree[1] == 'F') {
        A->father = levelordersearch(familytree, tree[2]);

    }
    else if (tree[1] == 'M') {
        A->mother = levelordersearch(familytree, tree[2]);
    }
    levelorderprint(A);
    printf("₩n");
    n++;
}
```

이 함수 또한 첫 번째 값이 입력된 이후 두 번째 입력부터의 과정인데, 이번엔 앞의 경우와는 달리 입력된 값의 세 번째 값이 기존의 tree에 저장되어 있는 경우, 즉 첫 번째 값이 기존의 tree에 저장되어있지 않은 경우입니다.
이때는 familytree에 기존의 A 주소를 저장하고, A에는 새로 동적 할당하여 새로운 tree를 생성합니다. 그리고 새로 생성된 tree의 name에 첫 번째 값을 저장하고, father 또는 mother에 입력받은 세 번째 값이 저장되어있는 tree의 주소값을 저장합니다. 이때 그 tree를 찾기 위해 levelordersearch함수를 호출하여 그 값을 찾은 후 A의 father 또는 mother에 저장하게 됩니다.

●출력결과

```
>>aFs
as
>>aMd
asd
>>sMe
asde
>>dFo
asdeo
>>sFq
asdqeo
>>kFa
kasdqeo
>>$$$
계속하려면 아무 키나 누르십시오 . . .
```



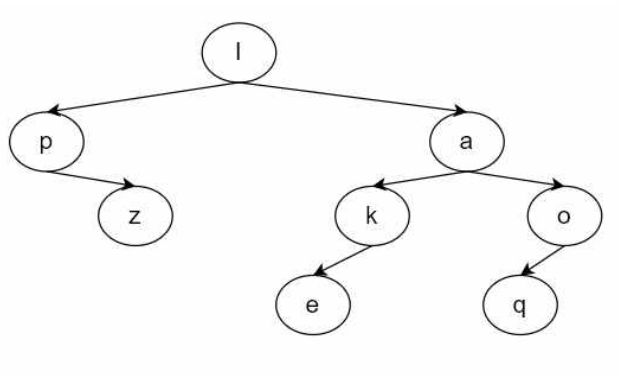
실행결과 1.

처음 aFs를 입력받기 때문에 a tree의 father에 s tree 주소값을 저장하고 as가 출력됩니다.

그 이후 aMd가 입력되는데, a tree의 mother에 d tree를 저장하고 asd가 출력되고, sMe가 입력되어 s의 mother에 e tree를 저장하고 asde출력, dFo가 입력되어 d의 father에 o tree를 저장하고 asdeo가 출력됩니다.

그 히우에 sFq, kFa가 입력받아서 s의 father에는 q tree를 저장하고 asdqeo를 출력, k의 father에 a tree를 저장하고 kasdqeo를 출력합니다. 이후 $$$가 입력되어 프로그램이 종료됩니다.

```
>>kFe
ke
>>aFk
ake
>>aMo
akoe
>>oFq
akoeq
>>lMa
lakoeq
>>lFp
lpakoeq
>>pMz
lpazkoeq
>>$$$
계속하려면 아무 키나 누르십시오 . . .
```



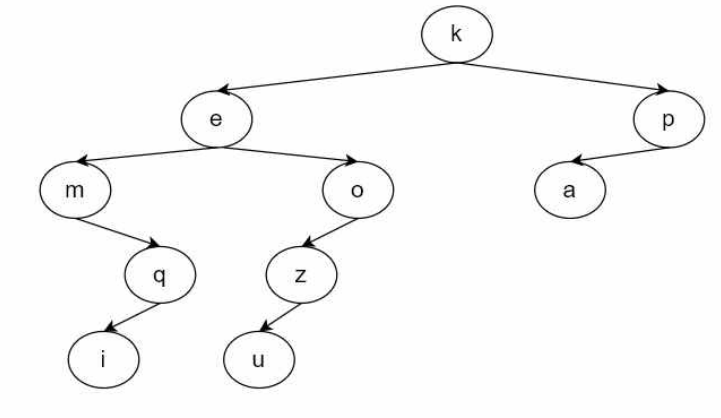
실행결과 2.

처음에 kFe를 입력받아서 k의 father에 e tree를 저장하고 ke가 출력됩니다.

aFk가 입력되어 a의 father에 k tree가 저장되고, ake가 출력됩니다. 그리고 aMo가 입력되서 a의 mother에 o tree를 저장하고 akoe를 출력, oFq가 입력되어 o의 father에 q tree 저장 후 akoeq가 출력됩니다.

lMa, lFp, pMz가 입력되고 각각 l의 mother에 a tree를 저장하고 lakoeq를 출력, l의 father에 p tree를 저장하고 lpakoeq를 출력, p의 mother에 z tree를 저장하고 lpazkoeq를 출력합니다. 그리고 이후 $$$가 입력되어 프로그램이 종료됩니다.

```
>>eFm
em
>>eMo
emo
>>mMq
emoq
>>qFi
emoqi
>>kFe
kemoqi
>>kMp
kepmoqi
>>pFa
kepmoaqi
>>oFz
kepmoaqzi
>>zFu
kepmoaqziu
>>$$$
계속하려면 아무 키나 누르십시오 . . .
```



실행결과 3.

eFm으로 e의 father에 m tree가 저장되고 em을 출력합니다.

eMo 입력으로 e의 mother에 o tree가 저장되고 emo가 출력되고, mMq 입력으로 m의 mother에 q tree가 저장되고 emoq가 출력됩니다. qFi로 q의 father에 i tree를 저장하고 emoqi가 출력됩니다. kFe로 k의 father에 e tree가 저장되고 kemoqi가 저장됩니다. 그리고 kMp로 k의 mother에 p tree를 저장, kepmoqi를 출력, pFa 입력으로 p의 father에 a tree 저장 후 kepmoaqi를 출력합니다. oFz, zFu 입력으로 o의 father에 z tree를 저장한 후 kepmoaqzi를 출력하고, z의 father에 u tree를 저장하고 kepmoaqziu를 출력합니다.

이후 $$$가 입력되어 프로그램을 종료합니다.

```
>>$$$
계속하려면 아무 키나 누르십시오 . . .
```

실행결과 4.

처음에 바로 $$$가 입력되어서 tree를 생성하지 않고 바로 프로그램이 종료됩니다.