자료구조 assignment3

사이버보안학과 201921534 박상현

휴머노이드 캐릭터의 관절 정보를 받아서 트리로 표현하고 순회하며 2차원 공간에서의 좌표를 계산했습니다. offset은 부모 노드와의 거리를 나타내기 때문에 루트 노드(0,0)에서 부터의 거리(절대적 거리) 계산을 위해서는 순회를 하며 부모 노드 좌표에 해당 Joint의 offset을 더해 주어야 했습니다.

Optoinal는 iterativeTree.c에 구현했습니다.

gcc를 사용해 컴파일하여 실행파일을 실행시켰습니다.

Tree 표현

저는 하나의 Joint를 구조체를 사용해 노드로 표현하고 그 안에 필요한 정보들을 담았습니다.

```
typedef struct Joint
{
    char name[20];
    float offsetX, offsetY;
    struct Joint *child;
    struct Joint *sibling;
} Joint;
```

이름, offset을 포함하고 child, sibling 노드의 포인터 값을 가지고 있습니다.

- 모든 노드가 sibling 값을 가지지는 않고 recursive한 트리 순회에 용이하도록 일부 노드들만 sibling 값을 가지고 있습니다.
- 포인터 값을 이용해 연결된 구조는 자료 탐색에는 시간이 걸릴 수 있으나 자료 삽입, 삭제에서 더욱 편리하며 메모리 공간의 낭비가 적어집니다.

Tree - addChild

루트 노드인 Hips에 문제에 제시된 신체 부위(노드)들을 추가하기 위한 함수입니다.

```
void addChild(Joint *parent, Joint *child)
{
    if (parent->child == NULL)
    {
        parent->child = child;
    }
    else
    {
        Joint *temp = parent;
        while (temp->sibling != NULL)
        {
            temp = temp->sibling;
        }
        temp->sibling = child;
    }
}
```

자식 노드가 없는 부모노드에 노드를 추가하려고 하면 바로 자식 노드로 추가해주고 만약 자식이 이미 존재하는 노드라면 자매 노드로 추가해줍니다. 이런 방식으로 하게 되면 특정 노드들만 자매 노드를 갖게 됩니다. 하지만 이런 방식이 왼쪽 탐색과 오른쪽 탐색을 더 용이하게 해주고 트리 순회 과정에서 이미 방문한 노드를 방문하게 되거나 어떤 노드를 방문하지 않는 경우, 잘못된 순서로 작업을 처리하는 경우를 방지하게 해줍니다.

- 해당 문제의 트리에서는 Neck, LeftCollar, LeftUpLeg만이 자매 노드를 갖게 됩니다.
    - 그렇게 되면 왼쪽 탐색(L)은 자신의 자식 노드를 방문하는 것이고 오른쪽 탐색(R)은 자신의 자매 노드를 방문하는 것이 됩니다.

Tree - traverse and print

트리의 모든 노드들을 탐색하고 해당 노드의 좌표를 계산하고 출력하는 함수 입니다.

```
void traverseAndPrint(Joint *joint, float parentX, float pare
{
    if (joint == NULL)
        return;

    float currentX = parentX + joint->offsetX;
    float currentY = parentY + joint->offsetY;
```

```
    //1
    printf("%s: (%.2f, %.2f)\n", joint->name, currentX, curre

    traverseAndPrint(joint->child, currentX, currentY); //L

    //2

    traverseAndPrint(joint->sibling, parentX, parentY);//R

    //3
}
```

부모 노드의 좌표와 현재 노드의 offset을 더해 현재 노드의 좌표를 계산하고 출력합니다.

자식 노드나 자매 노드를 인자로 다시 해당 함수를 호출하는 recursive한 방식으로 구현했습니다. recursive하게 구현하게 되면 메모리 부분에서 낭비가 발생하지만 코드의 위치를 조금만 바꿔도 바로 다른 방식의 트리 탐색 구현이 가능하기 때문에 직관적이고 더욱 유연한 구현이 가능한거 같습니다.

세가지 방식으로 트리를 탐색할 수 있습니다.

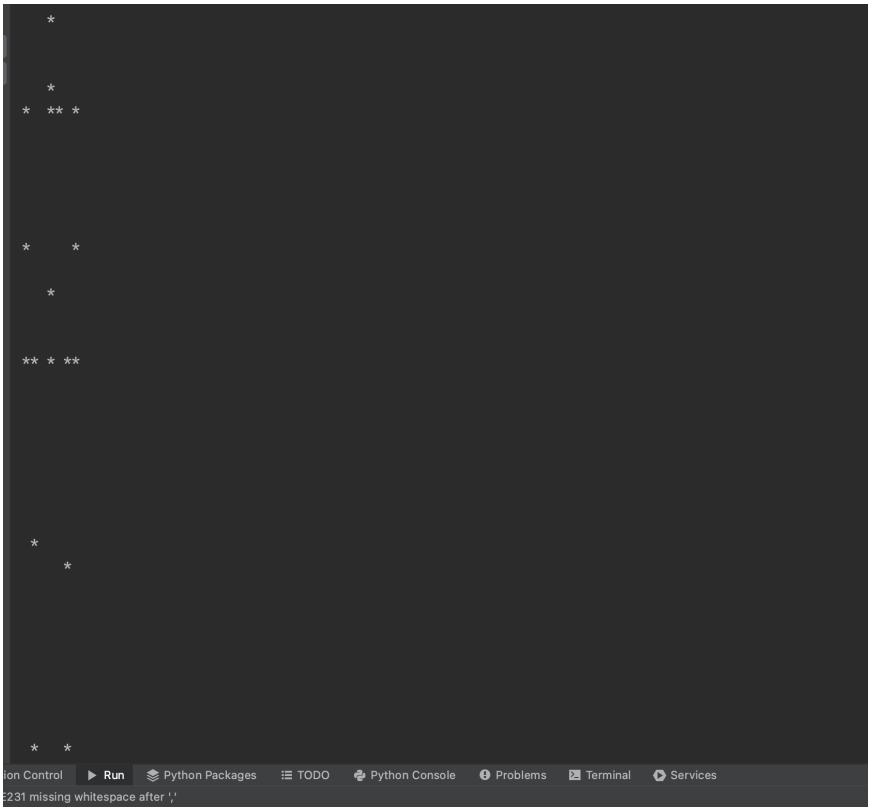
- PreOrder: 현재 위에 적혀있는 코드는 VLR 방식으로 preorder입니다. print 부분이 1번 위치에 들어가게 되는 경우 입니다.
- InOrder: LVR 방식으로 출력 부분이 가운데 오는 경우 입니다. 위 코드에서는 print 부분이 2번 위치에 들어가게 되면 Inorder 방식이 됩니다.
- PostOrder: LRV 방식으로 출력 부분이 마지막에 오면 됩니다. 위 코드에서는 3번 위치에 출력이 오면 됩니다.

저는 순회하는 순서를 직관적으로 보기 위해 preorder 방식을 사용해 결과를 출력했습니다.

```
Humanoid Character Model Coordinates:
Hips: (0.00, 0.00)
Chest: (0.00, 5.21)
Neck: (0.00, 23.86)
Head: (0.00, 29.31)
LeftCollar: (1.12, 21.44)
LeftUpArm: (6.66, 21.44)
LeftLowArm: (6.66, 9.48)
LeftHand: (6.66, -0.45)
RightCollar: (-1.12, 21.44)
RightUpArm: (-7.19, 21.44)
RightLowArm: (-7.19, 9.62)
RightHand: (-7.19, -1.03)
LeftUpLeg: (3.91, 0.00)
LeftLowLeg: (3.91, -18.34)
LeftFoot: (3.91, -35.71)
RightUpLeg: (-3.91, 0.00)
RightLowLeg: (-3.91, -17.63)
```

결과 - 시각화

위에서 출력된 좌표들을 파이썬을 이용해 *을 찍어서 표현했습니다.



특징으로는 LeftLowLeg와 RightLowLeg의 높이가 눈에 띄게 차이가 납니다. 나머지 Joint들은 거의 좌우 대칭인 모습을 보여주고 있습니다.

(Optional) iterative algorithm

트리를 iterative하게 순환하기 위해서는 스택을 활용한 구현이 필요합니다.

따라서 pop, push와 같은 스택과 관련된 함수들을 구현했습니다.

스택에 넣어주고 pop할 때 해당 Joint의 좌표를 구하기 위해서는 부모 노드의 좌표가 필요하기 때문에 Joint 구조체를 StackNode에 넣어서 필요한 추가적인 정보까지 저장했습니다.

```
typedef struct Joint
{
    char name[20];
    float offsetX, offsetY;
    struct Joint *child;
    struct Joint *sibling;
} Joint;

typedef struct StackNode
{
    Joint *joint;
    float parentX, parentY;
    struct StackNode *next;
} StackNode;
```

트리를 순회하는 방식은 자매 노드가 존재한다면 자매 노드가 더 이상 존재하지 않는 노드까지 push를 해준 뒤 이 후 모든 자매들의 child들을 push하는 방식입니다. 이렇게 반복문만을 사용해서 모든 노드들을 순회하고 Joint들의 좌표를 구합니다.

```
while (!isEmpty(stack))
    {
        StackNode *node = pop(&stack); // 스택에서 노드를 꺼냅니다
        Joint *joint = node->joint;
        float currentX = node->parentX + joint->offsetX; // 현
        float currentY = node->parentY + joint->offsetY; // 현

        printf("%s: (%.2f, %.2f)\n", joint->name, currentX, c

        while (joint->sibling)
        {
```

```
            push(&stack, joint->sibling, node->parentX, node->
            joint = joint->sibling;
        }

        if (joint->child)
        {
            push(&stack, joint->child, currentX, currentY);
        }

        free(node); // 사용한 노드 메모리 해제
    }
```