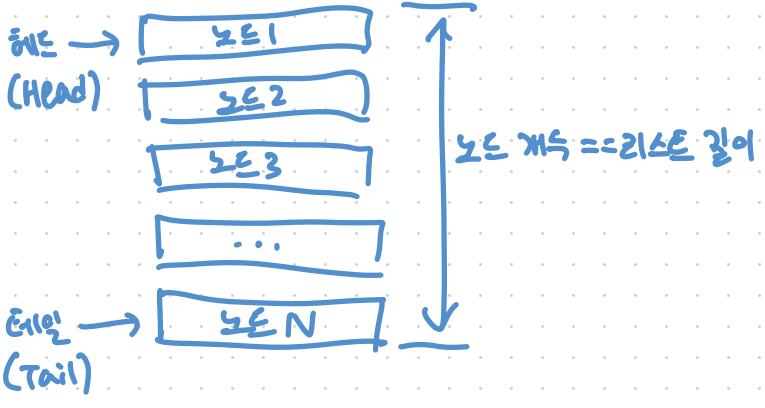
리스트

List

리스트 ADT

리스트는 목록 형태로 이뤄진 데이터 형식이다.



노드 : 리스트의 목록을 이루는 개별요소

리스트가 갖춰야 할 연산

Append : 리스트에 노드를 추가하는 연산

Insert : 노드 사이에 노드를 삽입하는 연산

Remove : 노드를 제거하는 연산

GetAt : 특정 위치에 있는 노드를 반환하는 연산

※ 설계에 따라 리스트 ADT가 갖는 연산 종류는 달라질 수 있다.

리스트와 배열 비교.

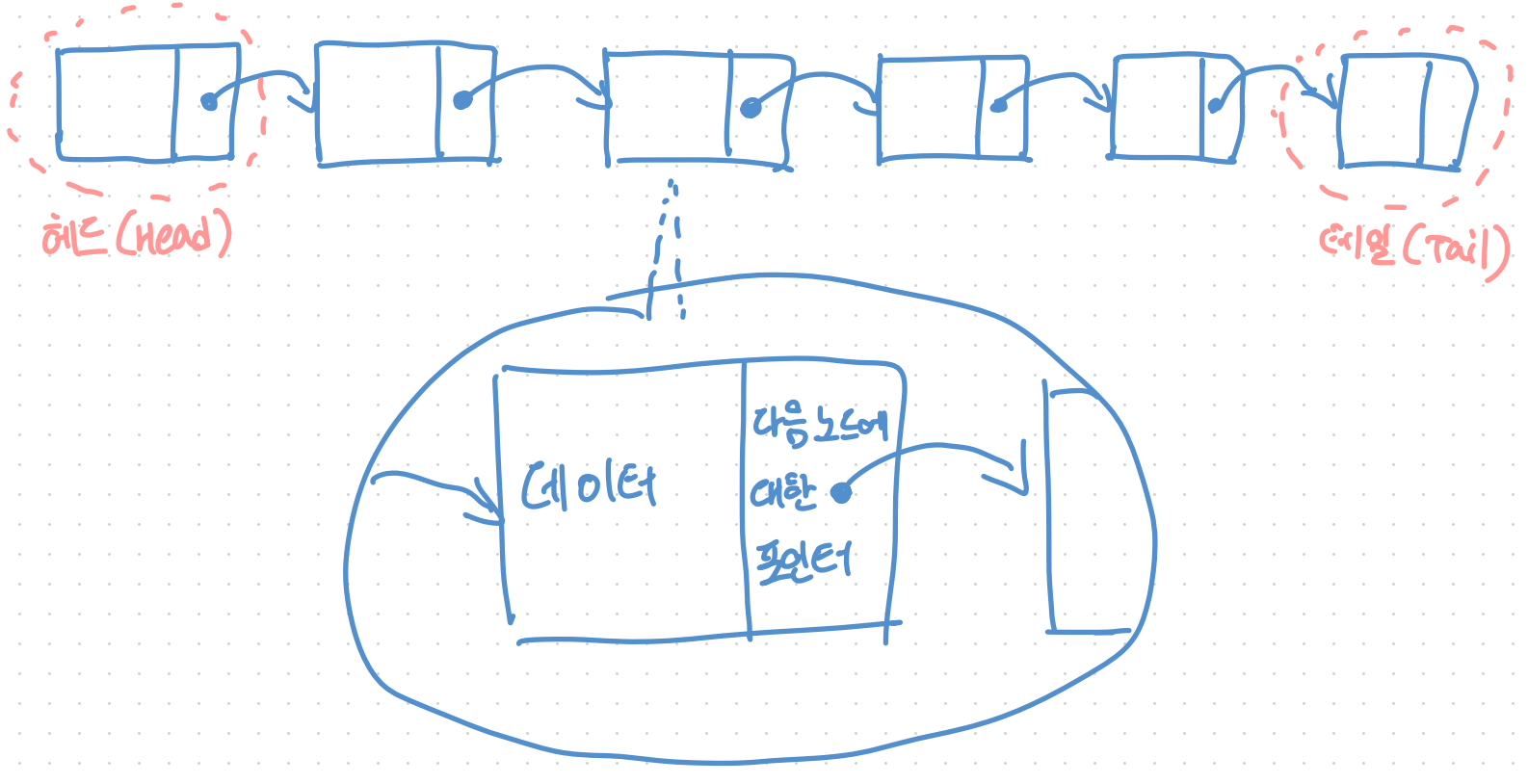
C언어에서 제공하는 배열은 '리스트'가 아니다.

Why? 배열은 생성하는 시점에 크기를 정해주어야 하고,
크기를 변경할 수 없다.

반면에 리스트는 크기를 유연하게 바꿀 수 있다.

Linked List - 링크드 리스트

링크드 리스트 : 노드를 연결해서 만든 리스트



링크드 리스트 주요 연산

1. 자료구조를 구축하기 위한 연산
   - 노드 생성 (CreateNode) / 소멸 (DestroyNode)
   - 노드 추가 (AppendNode) → 노드를 메모리에서 없애는 연산
   - 노드 삭제 (RemoveNode) → 리스트에서 노드를 제외하는 연산
   - 노드 삽입 (InsertAfter, InsertNewHead)

2. 자료구조에 저장된 데이터를 활용하기 위한 연산
   - 노드 탐색 (GetNodeAt)

노드 생성

- 메모리 레이아웃에서 자동메모리와 자유저장소 중에 노드를 어디에 생성하는 것이 좋을까?

```
Node* SLL_CreateNode (ElementType NewData)
{
    Node NewNode;  // 자동 메모리에 새로운 노드 생성
    NewNode.Data = NewData;
    NewNode.NextNode = NULL;

    return &NewNode;  // NewNode가 생성된 메모리의 주소를 반환
}  // 함수가 종료되면서 NewNode로 자동 메모리에서 제거된다.
...

Node* MyNode = SLL_CreateNode (10);  // MyNode는 할당되지 않은 메모리를 가리킨다.
```

자동 메모리에 생성하면, MyNode 포인터는 사라진 NewNode의 주소를 갖게 된다. 따라서 오류가 발생하거나 코드가 원하는대로 작동하지 않게 되므로 자동메모리에 노드를 생성하는 건 적합하지 않다.

노드를 자유 저장소에 저장한다면?

자유 저장소를 활용하려면 malloc 함수가 필요하다.

```
void* malloc (size_t size);
```

- void*: 모든 형식의 메모리를 가리킬 수 있는 포인터
- size_t: sizeof 연산에서 반환형 ⇒ typedef로 선언한 unsigned int의 별칭

malloc 함수를 사용하여 노드를 자유저장소에 생성하는 방법

```
Node* NewNode = (Node*) malloc (sizeof(Node));
```

- (Node*): 할당된 메모리 주소로 변환
- sizeof(Node): Node 구조체의 크기 만큼 메모리 확보

```
Node* SLL_CreateNode (ElementType NewData)
{
    Node* NewNode = (Node*) malloc (sizeof(Node));

    NewNode->Data = NewData; // 데이터를 저장한다.
    NewNode->NextNode = NULL; // 다음 노드에 대한 포인터는 NULL로 초기화한다.

    return NewNode; // Node의 주소를 반환한다.
}
```

노드 소멸

자유 저장소에서 메모리를 소멸시킬 땐 free함수를 사용하면 된다.

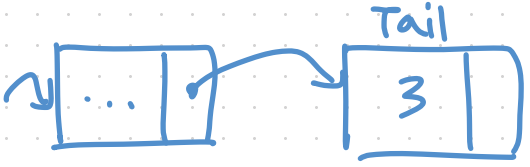
```
void free (void *memblock);
```

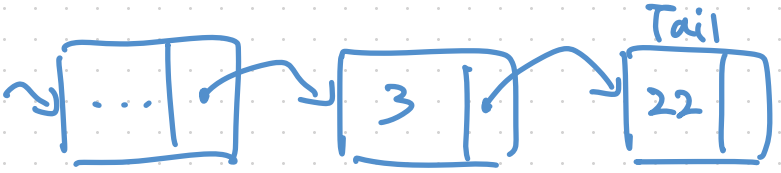
이 때 노드가 존재하는 주소를 가리키는 포인터가 필요하다.

```
void SLL_DestroyNode (Node* Node)
{
    free (Node);
}
```

노드 추가연산

노드 추가 : 링크드 리스트의 테일 뒤에 새로운 노드를 만들어 연결하는 연산






```
void SLL_AppendNode(Node** Head, Node* NewNode){
    // 헤드 노드가 NULL이라면 새로운 노드가 Head가 된다.
    if ((*Head) == NULL) {
        *Head = NewNode;
    }
    else {
        //테일을 찾아 NewNode를 연결한다.
        Node* Tail = (*Head);
        while (Tail->NextNode != NULL)
        {
            Tail = Tail->NextNode;
        }
        Tail->NextNode = NewNode;
    }
}
```

SLL_AppendNode함수 사용

```
Node* List = NULL;
Node* NewNode = NULL;
NewNode = SLL_CreateNode(119); // 자유 저장소에 노드 생성
SLL_Append(&List, NewNode); // 생성한 노드를 List에 추가
NewNode = SLL_CreateNode(119); // 자유 저장소에 또 다른 노드생성
SLL_Append(&List, NewNode); // 생성한 노드를 List에 추가
```

노드 탐색 연산

찾고자 하는 요소가 N번째에 있으면 N-1개의 노드를 지나야함.

```
Node * SLL_GetNodeAt(Node* Head, int Location) {

    Node* Current = Head;

    While (Current != NULL && (--Location) >= 0){
        Current = Current->NextNode
    }
    return Current
}
```

⇓ 함수 사용

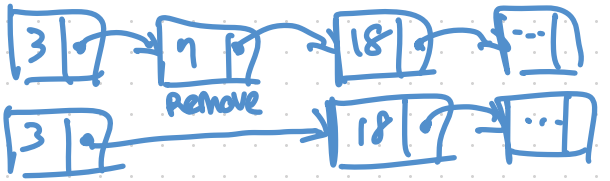
```
Node* List = NULL;
Node* MyNode = NULL;

SLL_AppendNode (&List, SLL_CreateNode(117));  // 노드를 생성하여 List에 추가
SLL_AppendNode (&List, SLL_CreateNode(119));  // 노드를 생성하여 List에 추가

MyNode = SLL_GetNodeAt (List,1);  // 두번째 노드 주소 MyNode에 저장
printf("%d", MyNode->Data);  // 119 출력
```

노드 삭제 연산

삭제하고자 하는 노드를 찾고, 해당 노드의 다음 노드를 이전 노드의 NextNode에 연결하면 된다.




```
Void SLL_RemoveNode(Node** Head, Node* Remove) {
    if (*Head == Remove){
        *Head = Remove->NextNode;
    } else {
        Node* Current = *Head;
        While (Current != NULL && Current->NextNode != Remove){
            Current = Current->NextNode;
        }
        if (Current != NULL)
            Current->NextNode = Remove->NextNode;
    }
}
```

⇓ 함수 사용

```
Node * List = NULL;
Node* MyNod = NULL;

SLL_AppendNode (&List, SLL_CreateNode(117); // 노드를 생성하여 List에 추가
SLL_AppendNode (&List, SLL_CreateNode(119); // 노드를 생성하여 List에 추가
SLL_AppendNode (&List, SLL_CreateNode(212); // 노드를 생성하여 List에 추가

MyNode = SLL_GetNodeAt (List, 1); // 두번째 노드의 주소를 MyNode에 저장
Printf("%d\n", MyNode->Data); // 119를 출력

SLL_RemoveNode ( &List, MyNode); // 두번째 노드 제거
SLL_DestroyNode (MyNode); // 링크드 리스트에서 제거한 노드를 메모리에서
                          완전히 소멸시킴
```

노드 삽입 연산

노드와 노드 사이에 새로운 노드를 끼워 넣는 연산

```
Void SLL_InsertAfter(Node* Current, Node* NewNode) {
    NewNode->NextNode = Current->NextNode;
    Current->NextNode = NewNode;
}
```

노드 개수 세기 연산

리스트의 길이를 구하는 연산

```
int SLL_GetNodeCount(Node* Head) {
    int Count = 0;
    Node* Current = Head;

    while (Current != NULL) {
        Current = Current->NextNode;
        Count++;
    }

    return Count;
}
```

링크드 리스트 장단점

장점 : 새로운 노드의 추가, 삽입, 삭제가 빠르다.

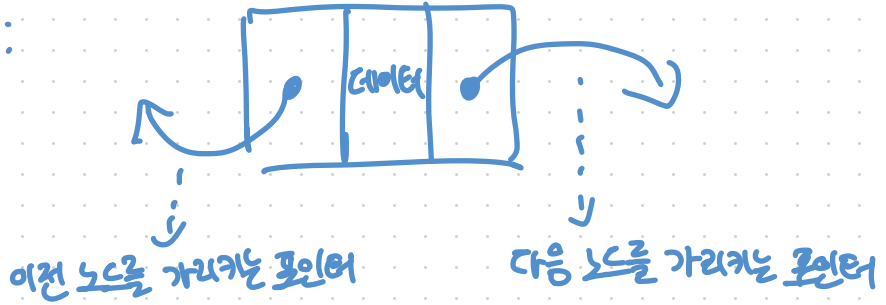
단점 : 특정 위치에 있는 노드에 접근하는 연산은 느리다.

=> 추가, 삽입, 삭제는 잦지만 조회가 드문 곳에서 사용하기 적합하다.

더블 링크드 리스트 Doubly Linked List

양방향 탐색이 가능한 링크드 리스트이다.

노드의 구조 :

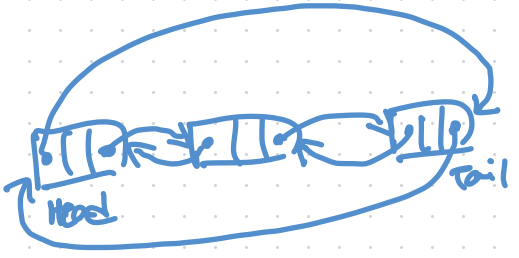


주요 연산 : 노드 생성/소멸 , 노드 추가, 노드 탐색 , 노드 삭제 , 노드 삽입, 노드 개수 세기

환형 링크드 리스트 Circular Linked List

더블 링크드 리스트와 대부분 동일하고,

테일의 다음 노드 포인터가 헤드를,
헤드의 이전 노드 포인터가 테일을 가리키도록 하면 된다.



노드를 추가했을 때, 노드가 '하나'라면
헤드의 다음 노드는 자기 자신이며, 이전 노드도 자기 자신이다.

노드가 '여러 개'라면 테일과 헤드 사이에 새 노드를 삽입한다고
생각하면 편하다.