**링크드 리스트**

링크드 리스트를 이해하기 위해서는 어레이 리스트와 ADT에 대한 선행 이해가 필수이다. 또 malloc을 사용한 동적할당과 free 함수에 대해 제대로 이해하고 있어야 한다.

링크드 리스트는 데이터와 다음 노드의 주소값을 갖는 구조체 변수로 이어진다. 선언은 다음과 같다.

typedef struct \_node

{

Int data; // 데이터

Struct \_node \*next; // 다음 노드

} Node;

여기서 다음 주소가 아니라 ‘노드’라는 표현을 굳이 사용하는 이유는 노드가 데이터와 주소값을 동시에 가지는 성질의 개체라는 것을 대표하는 표현이기 때문이다. 노드는 항상 그 자신의 데이터와 방향성을 갖고 다음 주소값을 가리킨다. 또 struct \_node \*next;와 같은 구조체 포인터 선언은 이해의 대상이 아니라 인식의 대상이다. 외워서 활용하자. Struct \_node \*next;

따라서 링크드 리스트의 데이터를 차례로 읽으려면 포인터 변수를 선언해 링크드 리스트의 주소값을 계속 옮겨다니며 읽어야 한다. 삭제할 때도 마찬가지이다. 링크드 리스트의 특정 인자를 삭제하고 다음 주소값을 저장해두지 않으면 다음 리스트에 접근이 불가능해진다. 다음 노드의 주소를 아는 건 이전 노드가 유일한데 이전 노드가 가리키는 다음 노드의 주소를 저장하지 않고 이전 노드를 삭제해버리면 다음 노드를 가리키는 주소값이 어디에도 존재하지 않게 된다. 또한 삭제는 free함수를 통해 이루어진다.

자료구조는 반드시 그림을 먼저 그리고 나서 이를 코드로 구현하는 방식으로 학습하자. 하라는 대로 학습하고 안 되면 다른 방법을 찾는다.

연결리스트는 추가되는 인자를 맨 앞에 넣는 방식과 맨 뒤에 넣는 방식으로 나뉜다. 많은 구현은 맨 앞에 새 노드를 넣는 방식을 택하는데 이렇게 하면 추가적인 포인터 변수 tail을 만들지 않아도 된다. 또 head포인터 변수 뒤에 더미 노드를 둬서 첫 번째 노드와 두 번째 이후 노드의 추가, 삭제를 동일하게 만들어 주기도 한다. 또 연결 리스트의 구조체는 위의 성격을 만족하는 포인터 변수(헤드, 커런트, 비포)와 함수 포인터 등을 멤버변수로 포함하면 좋다. 프로그램을 만들 때 다수의 리스트가 필요함으로 위와 같은 멤버변수들을 메인 함수에 전역으로 선언한다면 좋지 않은 코드가 나오기 때문이다.

연결 리스트의 정렬 삽입을 구현하려면 정렬의 기준이 되는 함수 포인터를 넣어주면 된다. 이를 위해서는 세 가지가 필요하다.

1. 연결 리스트의 정렬 기준이 되는 함수
2. 해당 함수를 통해 전달된 정렬기준을 저장하기 위한 멤버변수
3. 해당 정렬기준을 근거로 데이터를 저장하는 함수

Int WhatIsPrecede(int d1, int d2)

{

If(d1 < d2)

Return 0;

Else

Return 1;

}

위와 같은 함수는 인자의 정렬 우선순위가 높으면 0을 아니면 1을 반환하는 함수다.

값이 작을수록 정렬 우선순위가 높으니 오름차순 정렬이라고 할 수 있다.

자료구조의 구현을 아무런 참조없이 할 수 있을 수준이면 이상적인 수준이다. 한 번은 직접 생각해서 구현을 해보고 다음 번에는 실질적으로 가장 많이 쓰이는 자료구조의 소스코드를 외워가며 자신의 코드를 수정하는 편이 좋다. 왜냐하면 당장에 직접적으로 구현한 코드는 버그가 많을 수 있기 때문이다. 정석적인 코드를 외워가며 쓴다. 그리고 요구되는 능력은 ADT의 변경 및 추가로 인해서

일반적인 연결 리스트에서 마지막 노드가 첫 번째 노드를 가리키게 되면 원형 연결 리스트가 된다.

**변형된 원형 연결 리스트**

테일 -> next가 헤드를 가리키고 테일은 꼬리를 가리키는 원형 연결 리스트이다.

void LInsert(List \*plist, Data data)

{

Node \*newNode = (Node\*)malloc(sizeof(Node));

newNode -> data = data;

if(plist -> tail == NULL)

{

Plist -> tail = newNode;

newNode -> next = newNode;

}

Else

(plist -> numOfData)++;

}

Int LFirst(List \*plist, Data \*pdata)

{

If(plist -> tail == NULL)

Return false;

Plist -> before = plist -> tail;

Plist -> cur = plist -> tail -> next;

\*pdata = plist -> cur -> data;

Return TRUE;

}

Int LNext(List \*plist, Data \*pdata)

{

If(plist -> tail == NULL)

Return FALSE;

Plist -> before = plist -> cur;

Plist -> cur = plist -> cur -> next;

\*pdata =plist -> cur -> data;

Return TRUE;

}

변형된 원형 연결 리스트의 구현3: 노드의 삭제

삭제는 대부분의 경우 상대적으로 복잡하다. 하지만 머리와 꼬리가 연결되어 있다는 점만 제외하면 원형 연결 리스트와 단순 연결 리스트는 구조가 동일하기 때문에 삭제 방법도 유사하다.