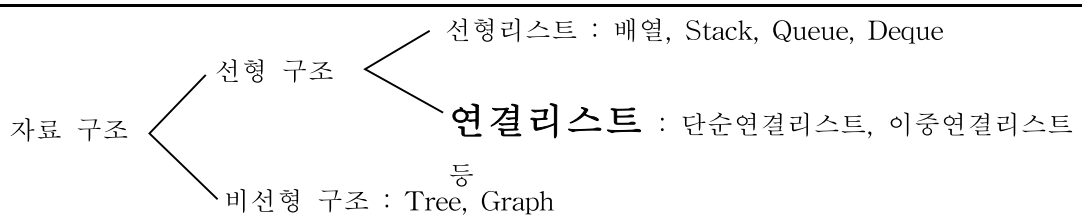
2-2 자료구조-선형구조-연결리스트

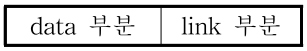
( )학년 ( )반 ( )번
이 름 : 모범답안

2. 자료구조의 구분(선형구조)



나. 선형구조

2) 링크드 리스트(Linked list) : 선형 구조에서 포인터를 사용하는 경우.



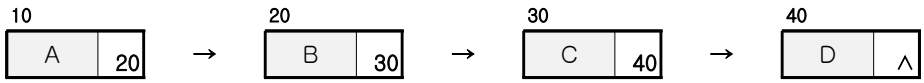
※ link 부분 : 다음에 찾아갈 기억장소의 주소 기억.

자료와 link 부분을 함께 저장하기 위하여 구조체 변수의 선언이 필요하다.

```
typedef struct ListNode {
   char data;
   struct Node *Link;
} ListNode;
```

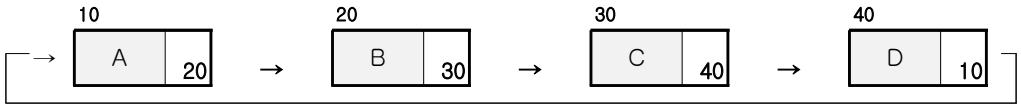
가) 링크드 리스트의 장점 : 중간 노드의 삽입, 삭제시 효율적이고 기억장소로부터 독립적이다.
나) 링크드 리스트의 단점 : Access time이 느리고 링크 포인터만큼의 기억공간을 소모한다.
다) 링크드 리스트의 종류

(1) 단일 선형 링크드 리스트(Singly Linked list)



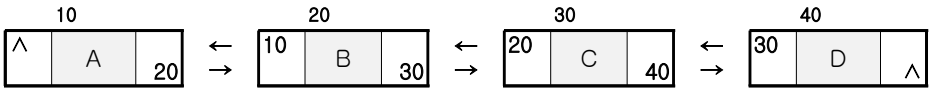
(가) 장점 : 간단하다.
(나) 단점 : 뒤에 있는 노드를 추적할 수 없고, 중간 노드 포인터를 잃어버리면 후속노드를 추적할 수 없다.

(2) 단일 환형 링크드 리스트(Singly Circular Linked list)



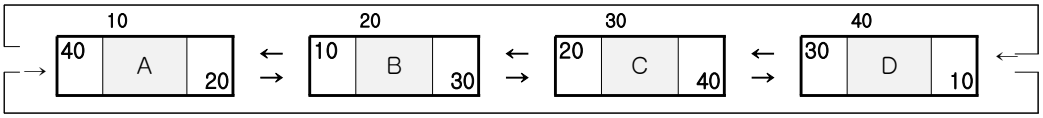
(가) 장점 : 어디에서 시작하든지 모든 노드의 엑세스가 가능하다.
(나) 단점 : 없는 노드를 찾을 경우 무한 루프에 빠질 수 있다. (head 포인터 이용)

(3) 이중 선형 링크드 리스트(Doubly Linked list)



(가) 장점 : 임의의 링크 포인터를 잃어버려도 역으로 읽는 포인터로 복구가 가능.
(순방향, 역방향으로 융통성이 좋다.)
(나) 단점 : 운영이 복잡하다.

(4) 이중 환형 링크드 리스트(Doubly Circular Linked list)



(가) 가장 융통성이 좋으나, 대단히 복잡하다.

[연습문제] 12개의 자료(16, 32, 23, 61, 75, 57, 83, 38, 97, 79, 47, 41)에 대하여

```
typedef struct   ListNode{
   int       data;
   struct   ListNode   *Link;
} ListNode;

ListNode* Head = NULL;
```

```
int main(){
   int   item[NN] = {16, 32, 23, 61, 75, 57, 83, 38, 97, 79, 47, 41};
   int a;

   for(a=0; a<NN; a++) Append(item[a]);   printList();
   printf("\t\t%dth Data: %d Search\n\n", 5, Peek(5));
   printf("\t\t%dth Data: %d Insert\n", 7, Insert(7, 77));   printList();
   printf("\t\t%dth Data: %d Delete\n", 9, Delete(9));   printList();
}
```

1) 단순연결리스트에 차례대로 기억하시오.

새 노드(New)를 만들어

→ New의 data에 item을
→ New의 Link에 Head를 할당하고
→ Head에 New를 할당한다.

```
int Append(int item){
   ListNode* New=(ListNode *)malloc(sizeof(ListNode));

   New->data = item;
   New->Link = Head;  Head=New;
   return  item;
}
```

2) 리스트의 5번째 항목 값을 출력하시오.

Head로부터 n번째 Node인 tmp를 찾아

→ tmp의 data를 반환한다.

```
int Peek(int n){
   ListNode* tmp=Head;
   int a;

   for(a=1; a<n; a++) tmp = tmp->Link;
   return tmp->data;
}
```

3) 리스트의 7번째 항목에 77을 끼워 넣으시오.

새 노드(New)를 만들고

Head로부터 n-1번째 Node인 tmp를 찾아

→ New의 Link에 tmp의 Link를 할당하고
→ tmp의 Link에 New를 할당한 후,
→ New의 data에 item을 할당한다.

```
int Insert(int n, int item){
   ListNode *New=(ListNode *)malloc(sizeof(ListNode));
   ListNode *tmp=Head;
   int  a;

   for(a=1; a<n-1; a++) tmp = tmp->Link;
   New->Link = tmp->Link;   tmp->Link = New;
   New->data = item;
   return  item;
}
```

4) 리스트의 9번째 항목을 삭제하시오.

Head로부터 n-1번째 Node인 tmp를 찾아

→ n번째 Node인 tmp의 Link를 N에 대피하고
→ tmp의 Link에 N의 Link를 할당한 후,
→ Node N의 사용을 해제한다.

```
int Delete(int n){
   ListNode *tmp=Head, *N;
   int a, item;

   for(a=1; a<n-1; a++) tmp = tmp->Link;

   N = tmp->Link; item = N->data;
   tmp->Link = N->Link;   free(N);

   return item;
}
```

[연결리스트를 이용한 삽입 정렬]

```
#include <stdio.h>
#include <stdlib.h>

#define  NN 18

typedef  struct    ListNode {
   int                 data;
   struct ListNode   *Link;
} ListNode;

ListNode* Head = NULL;

void printList();  void Insert(int);
```

자료의 개수

ListNode 타입의 구조체 변수 선언

ListNode 타입의 Head 선언

함수 프로토타입 정의

```
int main() {
   int  item[NN] = {16,32,23,61,75,57,83,38,97,79,47,41,11,34,24,45,53,62};
   int a;

   printf("Array [");
   for(a=0; a<NN; a++)     printf("%3d", item[a]);
   printf(" ]\n\n");

   for(a=0; a<NN; a++) {
      Insert(item[a]);    printList();
   }
}
```

원자료 배열 출력

각 항목을 리스트 Head에 순서에 맞추어 추가함으로써 정렬 완성

```
void printList() {
   ListNode* p = Head;

   printf(" List [");
   while(p) {
      printf(" %d", p->data);    p = p->Link;
   }
   printf(" ]\n");
}
```

[리스트 출력 함수]

리스트에 포함된 모든 Node 자료 출력

```
void Insert(int item) {
   ListNode *New = (ListNode *)malloc(sizeof(ListNode));
   ListNode *q = Head, *p;

   New->data = item;   New->Link = NULL;

   while(q != NULL) {
      if(item <= q->data) break;
      p = q;
      q = q->Link;
   }

   if(q == Head) {
      New->Link = q;    Head = New;
   } else if(q == NULL)
      p->Link = New;
   else {
      New->Link = p->Link;   p->Link = New;
   }
}
```

[노드 삽입 함수]
새 노드 New를 만들어

자료와 연결 pointer를 각각 설정하고,

삽입할 자료 이상의 Node를 찾아

그 위치가 Head인 경우,

리스트의 끝(NULL)인 경우

리스트의 중간인 경우를 구분하여

새 노드를 끼워넣는다.