5장. 리스트

❑ 리스트

- 목록이나 도표처럼 여러 데이터를 관리할 수 있는 자료형을 추상화
- 데이터 삽입, 삭제, 검색 등 필요 작업을 가함
- 스택과 큐는 리스트의 특수한 경우에 해당

❑ 학습목표

- 추상 자료형 리스트 개념과 필요 작업을 이해한다.
- 배열로 구현하는 방법, 연결 리스트로 구현하는 방법을 숙지한다.
- C로 구현할 때와 C++로 구현할 때의 차이점을 이해한다.
- 자료구조로서 배열과 연결 리스트의 장단점을 이해한다.

목록(도표)

Position	Name	Quantity
1	Beer	10
2	Gum	5
3	Apple	4
4	Potato	8
5	Onion	8
...	...	...

추상 자료형 리스트의 작업

- Insert(Position, Data)
  - 데이터를 해당 위치(Position)에 넣기
- Delete(Position)
  - 해당 위치(Position)의 데이터를 삭제
- Retrieve(Position, Data)
  - 해당 위치(Position)의 데이터를 Data 변수에 복사
- Create( )
  - 빈 리스트 만들기 (종이 준비)
- Destroy( )
  - 리스트 없애기(종이 버리기)
- IsEmpty( )
  - 빈 리스트인지 확인 (아무 것도 안 적혔는지 확인)
- Length( )
  - 몇 개의 항목인지 계산 (몇 개나 적혔는지 세기)

공리

- 공리
  - 추상자료형의 작업을 형식화
- 리스트 공리
  - (aList.Create( )).Length( ) = 0
  - (aList.Insert(i, Data)).Length( ) = aList.Length( ) + 1
  - (aList.Create( )).IsEmpty( ) = TRUE
  - (aList.Insert(i, Data)).Delete(i) = aList
  - (aList.Create( )).Delete(i) = ERROR
- Insert(1, Ramen)
  - 밀릴 것인가, 지울 것인가
  - 공리로도 명시하기 어려움
  - 인터페이스 파일에 정확한 커멘트를 요함

C 배열에 의한 리스트

- 구조체
  - 카운트 변수
  - 데이터 배열

Count	Data[ ]				
2	0	1	2	...	(MAX-1)
	324	256			

C 배열에 의한 리스트

- 코드 5-1: ListA.h (C Interface by Array)

```
#define MAX 100                                  최대 100개 데이터를 저장
typedef struct
{ int Count;                                     리스트 길이(데이터 개수)
를 추적
 int Data[MAX];                          리스트 데이터는 정수형
} listType;                                      리스트 타입은 구조체

void Insert(listType *Lptr, int Position, int Item): 해당위치에
데이터를 삽입
void Delete(listType *Lptr, int Position);              해당위치
데이터를 삭제
void Retrieve(listType *Lptr, int Position, int *ItemPtr);
                                                 찾은 데이터를 *ItemPtr에
넣음
void Init(listType *Lptr);                              초기화
bool IsEmpty(listType *Lptr);                           비어있는지 확인
int Length(listType *Lptr);                             리스트 내 데이터
개수
```

C 배열에 의한 리스트

- void Insert(listType *Lptr, int Position, int Item):
  - 자료구조를 함수호출의 파라미터로 전달
  - 리스트를 가리키는 포인터를 Lptr로 복사해 달라는 것
  - C 언어로 구현할 때 일반특성으로서 전역변수를 회피하기 위함

- void Insert(listType *Lptr, int Position, int Item):
  - 삽입, 삭제는 원본 자체를 바꾸는 작업이므로 참조호출이 필요
  - 리스트 자체 데이터는 복사되지 않음. 복사에 따른 시간도 줄어든다.

- void Retrieve(listType *Lptr, int Position, int *ItemPtr);
  - 호출함수의 원본 데이터를 가리키는 포인터 값을 ItemPtr로
  - *ItemPtr를 변형하면 호출함수의 원본 데이터 값이 변함
  - void main( )
    { listType List1;
    int Result;
    Insert(&List1, 1, 23); 리스트 처음에 23을 넣기
    Retrieve(&List1, 1, &Result); 리스트의 첫 데이터를 Result에 넣기
    }

C 배열에 의한 리스트

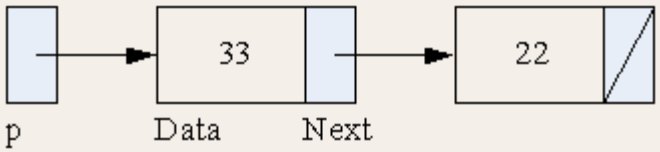
- 코드 5-2: ListA.c (C Implementation by Array)

```
#include <ListA.h>                          헤더파일을 포함
void Init(listType *Lptr)                           초기화 루틴
{ Lptr->Count = 0;                          데이터 수를 0으로 세팅
}
bool IsEmpty(listType *Lptr)                        비어있는지 확인하는 함수
{ return (Lptr->Count = = 0);               빈 리스트라면 TRUE
}
void Insert(listType *Lptr, int Position, int Item)   삽입함수
{ if (Lptr->Count = = MAX)                  현재 꽉 찬 리스트
      printf("List Full");
 else if ((Position > (Lptr->Count+1)) || (Position < 1))
      printf("Position out of Range");      이격된 삽입위치 불허
 else
 {     for (int i = (Lptr->Count-1); i >= (Position-1); i--)끝에서부터
삽입위치까지
            Lptr->Data[i+1] = Lptr->Data[i];        오른쪽으로 한 칸씩
이동
      Lptr->Data[Position-1] = Item;        원하는 위치에 삽입
      Lptr->Count += 1;                     리스트 길이 늘림
 }
}
```

연결 리스트 기초

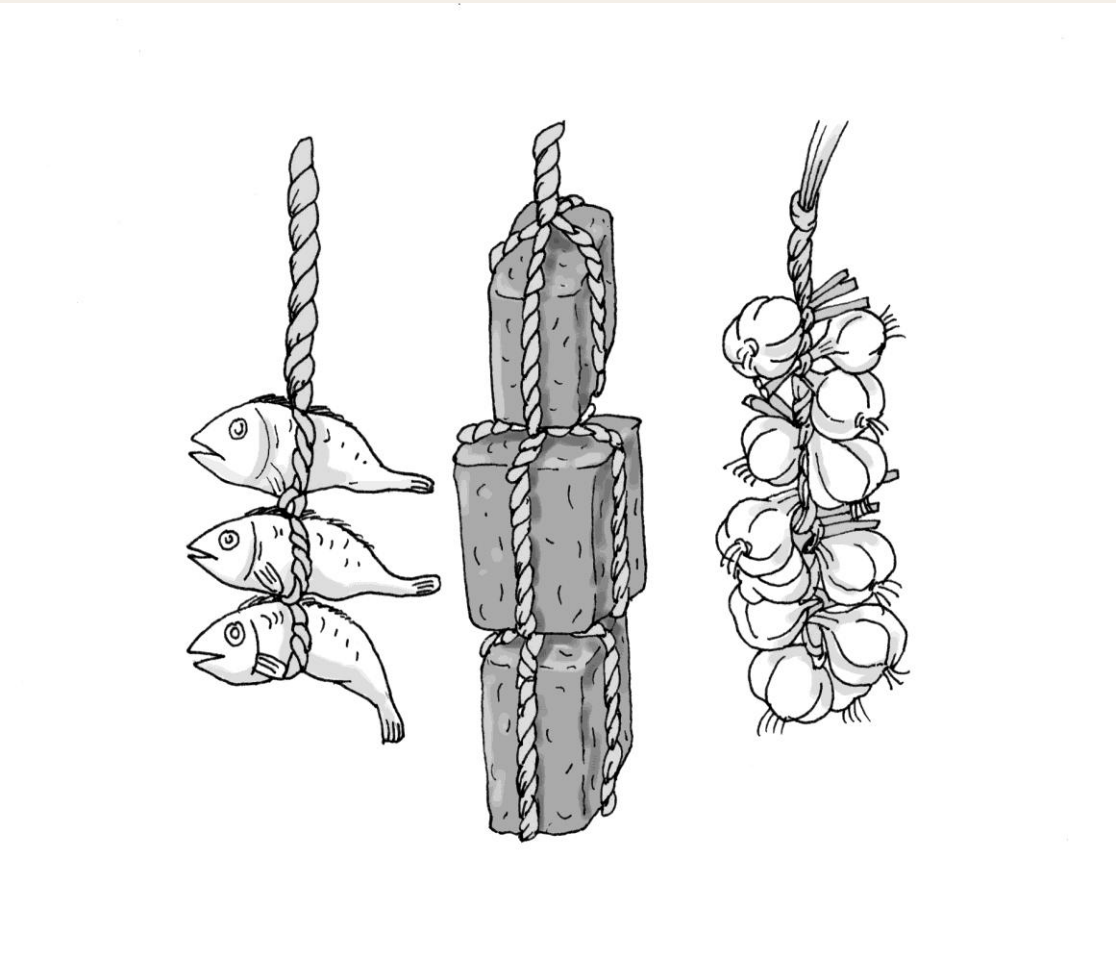
```
typedef struct
{   int Data;          노드 내부의 실제 데이터 또는 레코드
    node* Next;                Next가 가리키는 것은 node 타입
 } node;               구조체에 node라는 새로운 타입명 부여

typedef node* Nptr;    Nptr 타입이 가리키는 것은 node 타입
Nptr p, q;             Nptr 타입 변수 p, q를 선언
```



연결 리스트

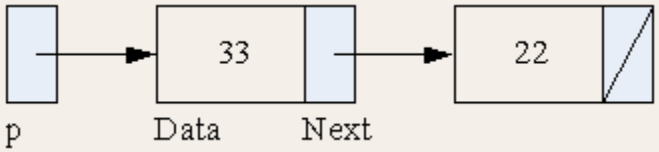
- 연결 리스트 개념



연결 리스트 기초

- 노드 만들기, 이어 붙이기

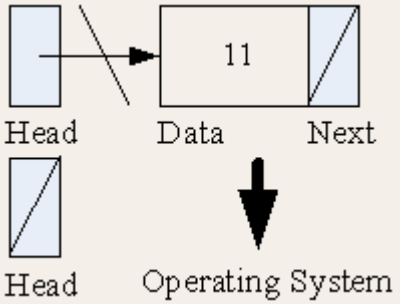
```
p = (node *)malloc(sizeof(node));
p->Data = 33;
p->Next = (node *)malloc(sizeof(node));
p->Next->Data = 22;
p->Next->Next = NULL;
```



연결 리스트 기초

- 공간반납

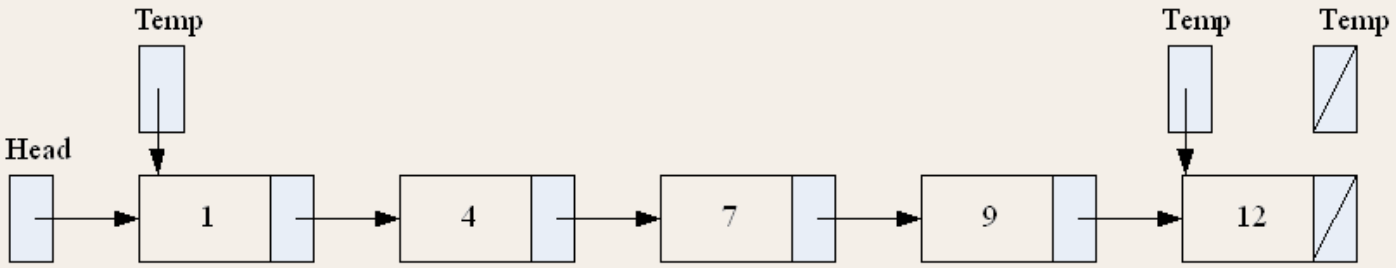
```
Nptr Head;
Head = (node *)malloc(sizeof(node));
Head -> Data = 11;
Head -> Next = NULL
Head = NULL;
```



연결 리스트 기본조작

- 디스플레이

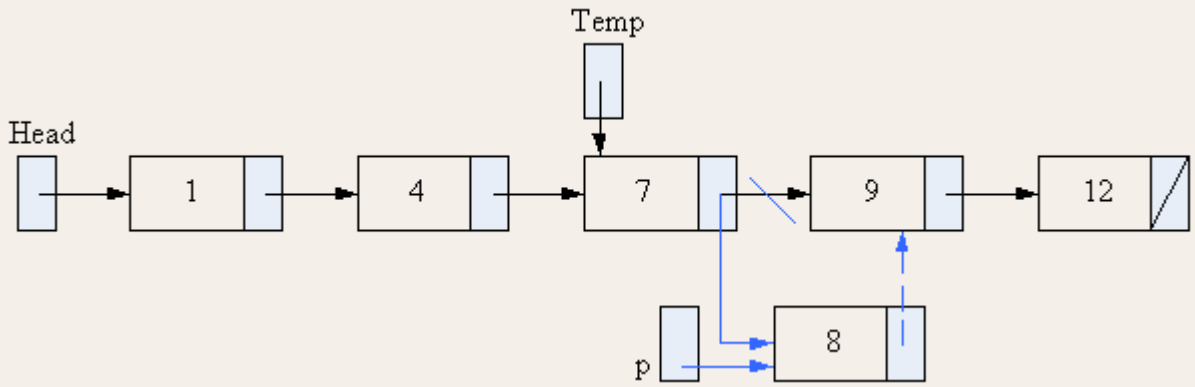
```
Temp = Head;
While (Temp != NULL)
{   printf("%d ", Temp->Data);
        Temp = Temp->Next;
}
```



연결 리스트 기본조작

❑ 간단한 삽입

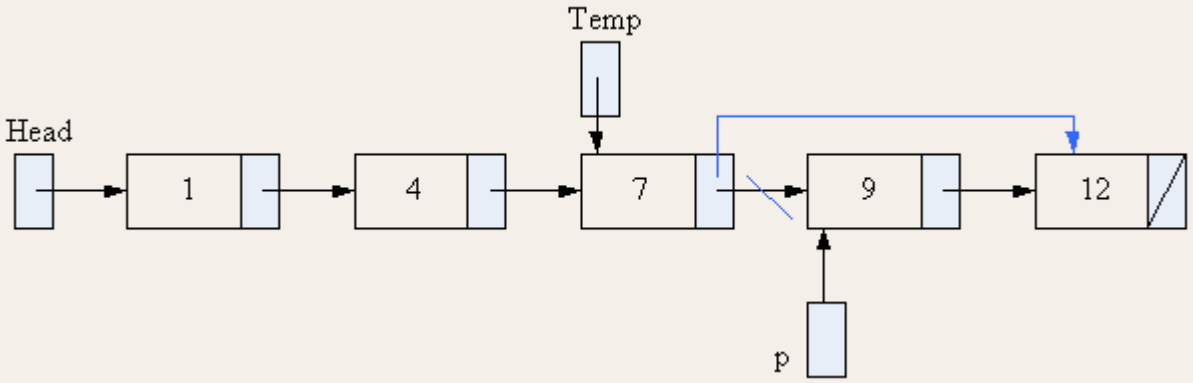
```
p = (node *)malloc(sizeof(node));
p->Data = 8;
p->Next = Temp->Next;
Temp->Next = p;
```



연결 리스트 기본조작

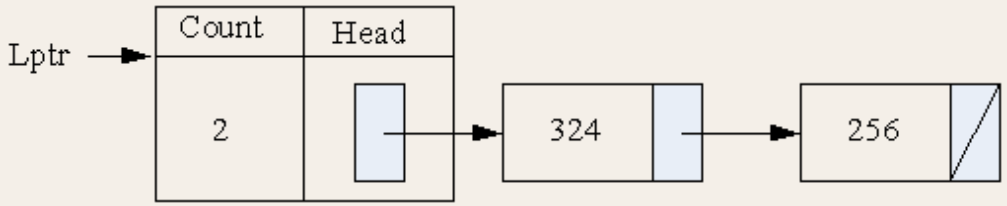
- 간단한 삭제

```
p = Temp->Next;
Temp->Next = Temp->Next->Next;
free p;
```



C 연결 리스트에 의한 리스트

- C 연결 리스트에 의한 리스트



C 연결 리스트에 의한 리스트

- 코드 5-3: ListP.h (C Interface by Linked List)

```
typedef struct
{ int Data;              노드 내부의 실제 데이터 또는 레코드
  node* Next;  Next가 가리키는 것은 node 타입, 즉 자기 자신 타입
} node;          구조체에 node라는 새로운 타입명 부여
typedef node* Nptr;  Nptr 타입이 가리키는 것은 node 타입

typedef struct
{ int Count;   리스트 길이를 추적
  Nptr Head;   헤드 포인터로 리스트 전체를 대변함
} listType;

void Insert(listType *Lptr, int Position, int Item): 해당위치에 데이터
를 삽입
void Delete(listType *Lptr, int Position);          해당위치 데이터를
삭제
void Retrieve(listType *Lptr, int Position, int *ItemPtr);
void Init(listType *Lptr);                              초기화
bool IsEmpty(listType *Lptr);                         비어있는지 확인
int Length(listType *Lptr);                            리스트 내 데이터
수
```

C 연결 리스트에 의한 리스트

- 코드 5-4: ListP.c (C Implementation by Linked List)

```
#include <ListP.h>                  헤더파일을 포함
void Init(listType *Lptr)
{ Lptr->Count = 0;                  리스트 길이를 0으로 초기화
  Head = NULL;                      헤드 포인터를 널로 초기화
}

bool IsEmpty(listType *Lptr)              빈 리스트인지 확인하기
{ return (Lptr->Count = = 0);             리스트 길이가 00이면 빈 리
스트
}
```

C 연결 리스트에 의한 리스트

❑ 코드 5-4: ListP.c (C Implementation by Linked List)

```
void Insert(listType *Lptr, int Position, int Item)     삽입함수
{   if ((Position > (Lptr->Count+1)) || (Position < 1))
       printf("Position out of Range");                  이격된 삽입위치 불허
    else
    {   Nptr p = (node *)malloc(sizeof(node));           삽입될 노드의 공간
    확보
       p->Data = Item;                                   데이터 값 복사
        if (Position = = 1)                              첫 위치에 삽입할 경우
       {    p->Next = Lptr->Head;     삽입노드가 현재 첫 노드를 가리킴
          Lptr->Head = p;             헤드가 삽입노드를 가리키게
       }
        else                                             첫 위치가 아닐 경우
      {    Nptr Temp = Lptr->Head;    헤드 포인터를 Temp로 복사
           for (int i = 1; i < (Position-1); i++)
               Temp = Temp->Next;     Temp가 삽입직전 노드를 가리키게
         p->Next = Temp->Next;        삽입노드의 Next를 세팅
         Temp->Next = p;              직전노드가 삽입된 노드를 가리키게
      }
     Lptr->Count += 1;                                   리스트 길이 늘림
  }
}
```

배열과 연결리스트 비교(삽입)

❑ 연결 리스트

- 공간이 꽉 차 있는지 테스트할 필요가 없음
- 얼마든지 동적으로 새로운 노드를 추가 가능
- 중간위치 삽입에 따른 밀림(Shift)이 불필요
- 삽입직전 노드를 찾아가는 작업이 배열에 비해 오래 걸림
  - 배열: 직접 접근
  - 연결 리스트: 포인터를 따라서 리스트 순회

C 연결 리스트에 의한 리스트

❑ 코드 5-5: 위치기반 연결 리스트의 삭제

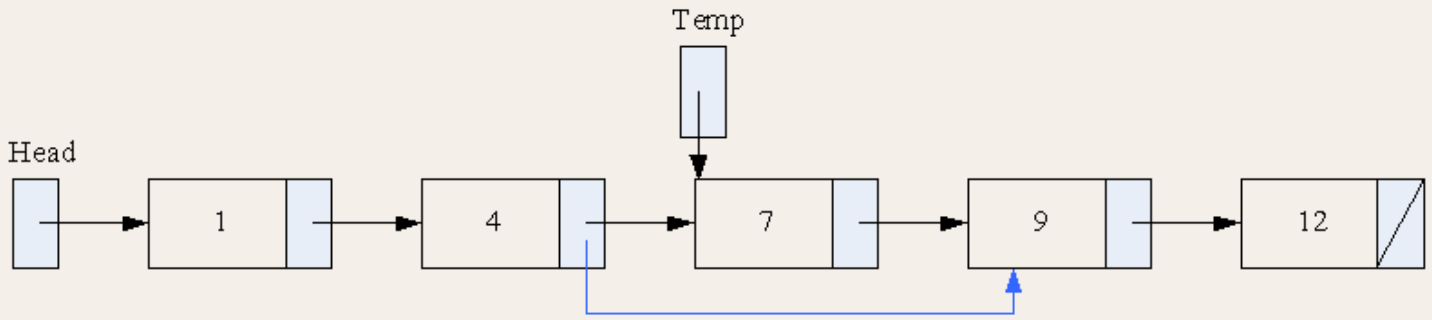
```
void Delete(listType *Lptr, int Position)        삭제함수
{   if (IsEmpty(Lptr))
       printf("Deletion on Empty List");     빈 리스트에서 삭제요구는 오류
    else if (Position > (Lptr->Count) || (Position < 1))
       printf("Position out of Range");       삭제 위치가 현재 데이터 범위
    를 벗어남
    else
    {   if (Position = = 1)                          첫 노드를 삭제하는 경우
      {   Nptr p = Lptr->Head;                      삭제될 노드를 가리키는 포
    인터를 백업
          Lptr->Head = Lptr->Head->Next;  헤드가 둘째 노드를 가리키게
       }
       else
       {  for (int i = 1; i < (Position-1); i++)
         Temp = Temp->Next;                    Temp가 삭제직전 노드를 가리키게
          Nptr p = Temp->Next;          삭제될 노드를 가리키는 포인터를 백업
          Temp->Next = p->Next;         직전노드가 삭제될 노드 다음을 가리키
    게
       }
       Lptr->Count -= 1;                      리스트 길이 줄임
       free (p);                                  메모리 공간 반납
   }
}
```

C 연결 리스트에 의한 리스트

❑ 값 기반 연결리스트의 삭제

```
Temp = Lptr->Head;
while((Temp != NULL) && (Temp->Data != Item)
   Temp = Temp->Next;
```

- 이 코드로는 템프가 삭제 직전에 놓이게 할 수 없음



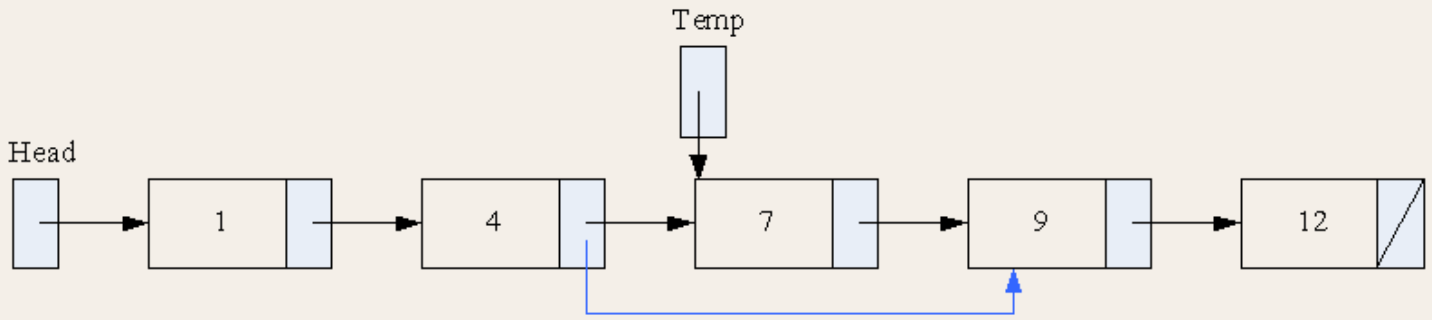
C 연결 리스트에 의한 리스트

예견방식

```
Temp = Lptr->Head;
while((Temp != NULL) && (Temp->Next->Data != Item)
      Temp = Temp->Next;
```

데이터 15인 노드 삭제

- Temp가 데이터 12인 노드를 가리킬 때,
- Temp는 널이 아니지만 Temp->Next는 널임.



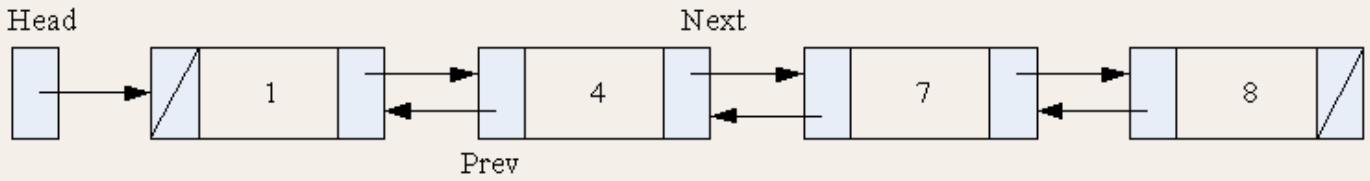
C 연결 리스트에 의한 리스트

❑ 코드 5-6 정렬된 연결 리스트의 삭제

```
Delete(listType *Lptr, int Item)          삭제함수
{  Nptr Prev = NULL;                      Prev는 널로 초기화
   Nptr Temp = Lptr->Head;                Temp는 헤드로 초기화
    while((Temp != NULL) && (Temp->Data != Item)
   {   Prev = Temp;                       Prev가 Temp를 가리키게
     Temp = Temp->Next;                   Temp는 다음 노드로 전진
   }
    if (Prev = = NULL)                    Temp가 한번도 전진하지 않았음
   {   if (Temp = = NULL)                 처음부터 빈 리스트
         printf("No Nodes to Delete");
       else                               삭제 대상이 첫 노드
     {   Lptr->Head = Lptr->Head->Next;
         free (Temp);   Lptr->Count - = 1;
     }
   }
  else                                    Temp가 전진했음
  {   if (Temp = = NULL)                  리스트 끝까지 삭제대상이 없음
       printf("No Such Nodes");
    else                                  삭제대상이 내부에 있음
     {   Prev->Next = Temp -> Next;       직전노드가 삭제될 다음 노드를 가리킴
       free (Temp);  Lptr->Count - = 1;
        }
   }
  }
```

이중연결 리스트

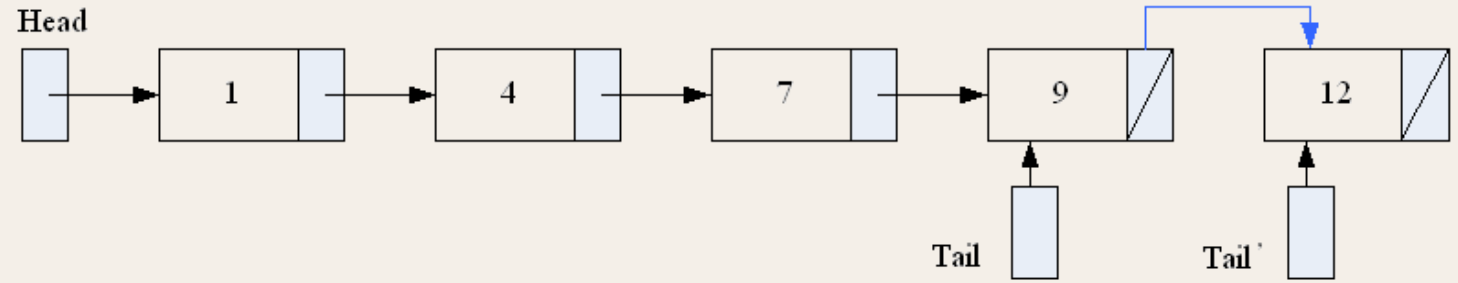
- typedef struct
- { int Data;
- node* Prev, Next; Prev는 이전 노드를, Next는 다음 노드를 가리킴
- } node;

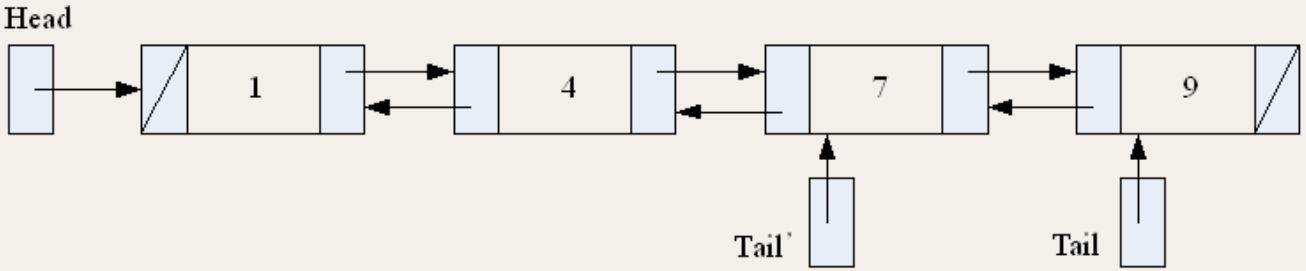


이중연결 리스트

단순연결 리스트의 삽입

- 테일 포인터가 따로 있으면 유리
- 삭제일 경우에는 테일 포인터가 뒤로 이동.
- 이중 연결 이어야 테일 포인터를 뒤로 이동할 수 있음.

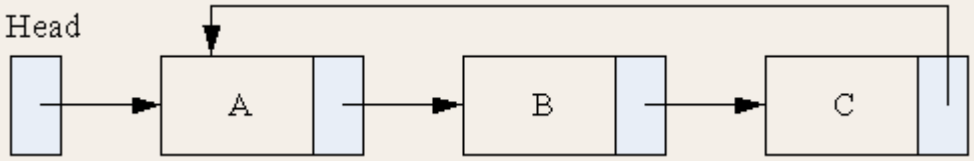




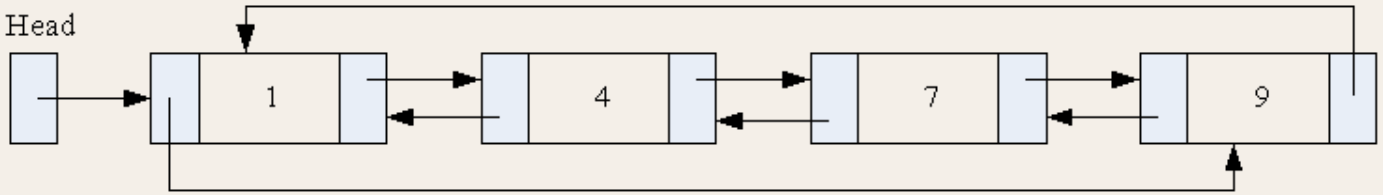
원형 연결, 원형 이중연결

❑ 원형 연결

- 타임 셰어링 시스템의 타임 슬라이스
- 사용자 교대



❑ 원형 이중 연결



C 배열과 C++배열 비교

ListA.h (C++ Interface by Array)	ListA.h (C Interface by Array)
const int MAX = 100;	#define MAX 100
class listClass	
{ public:	
listClass( );	void Init(listType *Lptr);
listClass(const listClass& L);	
~listClass( );	
void Insert(int Position, int Item);	void Insert(listType *Lptr, int Position, int Item)
void Delete(int Position);	void Delete(listType *Lptr, int Position);
void Retrieve(int Position, int & Item);	void Retrieve(listType *Lptr, int Position, int *ItemPtr);
bool IsEmpty( );	bool IsEmpty(listType *Lptr);
int Length( );	int Length(listType *Lptr);
private:	typedef struct
int Count;	{ int Count;
int Data[MAX];	int Data[MAX];
}	} listType;

C++ 배열에 의한 리스트

- 코드 5-7: ListA.cpp (C++ Implementation by Array)

```
#include <ListA.h>
listClass::listClass( )                    생성자
{ Count = 0;                        리스트 길이를 0으로 초기화
}

listClass::~listClass( )                   소멸자
{
}

listClass::listClass(const listClass& L) 복사 생성자
{ Count = L.Count;                  리스트 길이를 복사
   for (int i = 1; i <= L.Count; ++i)        깊은 복사에 의해
배열 요소 모두를 복사
      Data[i-1] = L.Data[i-1];
}
```

C++ 연결 리스트에 의한 리스트

- 코드 5-8: ListP.h (C++ Interface by Linked List)

```
typedef struct
{ int Data;
  node* Next;
} node;
typedef node* Nptr;

class listClass
{ public:
        listClass( );
        listClass(const listClass& L);
        ~listClass( );
        void Insert(int Position, int Item);
        void Delete(int Position);
        void Retrieve(int Position, int& Item);
        bool IsEmpty( );
        int Length( );
 private:
        int Count;
        Nptr Head;
}
```

C++ 연결 리스트에 의한 리스트

- 코드 5-9: ListP.cpp (C++ Implementation by Linked List)

```
#include <ListP.h>
listClass::listClass( )                    생성자 함수
{    Count = 0;                    리스트의 길이를 0으로 초기화
     Head = NULL;                  헤드를 널로 초기화
}

bool listClass::IsEmpty( )                 빈 리스트인지 확인하는 함
수
{  return (Count = = 0);                   배열 길이 0이면 TRUE
}
```

C++ 연결 리스트에 의한 리스트

- 코드 5-9: ListP.cpp (C++ Implementation by Linked List)

```
void listClass::Delete(int Position)   삭제함수
{  Nptr Temp;
    if (IsEmpty( ))
          cout << "Deletion on Empty List";빈 리스트에 삭제요구는 오류
  else if ((Position > Count) || (Position < 1))
         cout << "Position out of Range"; 삭제위치가 현재 데이터 범위를 벗어
남
   else
   {     if (Position = = 1)          삭제될 노드가 첫 노드일 경우
         {   Nptr p = Head;           삭제될 노드를 가리키는 포인터를 백업
             Head = Head->Next;       헤드가 둘째 노드를 가리키게
          }
         else                                  삭제노드가 첫 노드가 아닌 경우
         {   for (int i = 1; i < (Position-1); i++)
                   Temp = Temp->Next;  Temp가 삭제될 노드 직전노드로 이동
              Nptr p = Temp->Next;      삭제될 노드를 가리키는 포인터를 백업
         Temp->Next = p->Next;         직전노드가 삭제될 노드 다음을 가리키게
          }
         Count - = 1;                  리스트 길이 줄임
         delete (p);                   메모리 공간 반납
     }
}
```

C++ 연결 리스트에 의한 리스트

- 코드 5-9: ListP.cpp (C++ Implementation by Linked List)

```
listClass::~listClass( )                          소멸자 함수
{ while (!IsEmpty( ))                    리스트가 완전히 빌 때까지
      Delete(1);                          첫 번째 것을 계속 지우기
}

listClass:ListClass(const listClass& L) 복사 생성자 함수
{  Count = L.Count;                       일단 리스트 개수를 동일하게
    if (L.Head = = NULL)
       Head = NULL                 넘어온 게 빈 리스트라면 자신도 빈 리스트
   else
   {    Head = new node;          빈 리스트 아니라면 일단 새 노드를 만들고
      Head->Data = L.Head->Data;                  데이터 복사
       Nptr Temp1 = Head;               Temp1은 사본을 순회하는 포인터
      for (Nptr Temp2=L.Head->Next; Temp2 != NULL; Temp2=Temp2->Next)
     {   Temp1->Next = new node;   사본의 현재 노드에 새 노드를 붙임
        Temp1 = Temp1 -> Next;     새 노드로 이동
         Temp1->Data = Temp2->Data; 새 노드에 원본 데이터를 복사
       }
      Temp1->Next = NULL;               사본의 마지막 노드의 Next에 널을 기입
  }
   }
```

배열과 연결 리스트의 비교

- 공간
  - 배열은 정적이므로 최대크기를 미리 예상해야 함.
  - 만들어 놓은 배열 공간이 실제로 쓰이지 않으면 낭비
  - 연결 리스트는 실행 시 필요에 따라 새로운 노드 생성
  - 연결 리스트는 포인터 변수공간을 요함
- 검색시간
  - 배열은 단번에 찾아감(묵시적 어드레싱: 인덱스 연산)
  - 연결 리스트는 헤드부터 포인터를 따라감(현시적 어드레싱: 포인터 읽음)
- 삽입, 삭제 시간
  - 배열의 삽입: 오른쪽 밀림(Shift Right)
  - 배열의 삭제: 왼쪽 밀림(Shift Left)
  - 연결 리스트: 인접 노드의 포인터만 변경
- 어떤 자료구조?
  - 검색위주이면 배열이 유리
  - 삽입 삭제 위주라면 연결 리스트가 유리
  - 최대 데이터 수가 예측 불가라면 연결 리스트