『IT Cookbook C/C++로 배우는 자료구조론』

(연습문제 답안. 2004년 6월 작성: Version 1.0)

* 답안에 오류가 있을 시에는 한빛미디어 홈페이지의 해당 교재 게시판에 올려 주시기 바랍니다.

1장. 객체지향 방법론

1. 외부사용자로서는 작업명만 알면 될 뿐 자료구조와 구현방법은 알 필요가 없다. 내부 구현자로서는 작업과 그 작업이 가해지는 자료구조를 하나로 묶어서 정의할 수 있다.

2. 객체 각각에 대해 가할 수 있는 작업을 정의함으로써 프로그램이 완성된다. 프로그램의 실행은 객체 사이의 메시지 전달에 의해 객체 간의 상호작용으로 이루어진다.

3. 다 4. 라 5. 나 6. 라 7. 라 8. 마 9. 다
10. 가 11. 라 12. 나

13.

```
void main( )
{ Hello HelloObject;
  char Name[30];
  cout << "Enter Name";
  cin >> Name;
  HelloObject.Greet(Name);
}
```

14.
가. 0 + 0i 나. 1 + 2i 다. 1 - 2i

15. 자료검색 문제

16. “Good morning world"가 출력된다. p가 가리키는 것은 Baseclass로 선언되었으므로 실행시 p = &s 에 의해 p가 가리키는 것을 subclass로 변경하더라고 p의 타입은 선언시 타입 그대로 유지된다.(정적 바인딩). 동적 바인딩(Dynamic Binding, Delayed Binding)을 가하면 실행시 실제 가리키는 객체의 타입을 기준으로 함수가 불리워 온다. 이를 위해서는 Baseclass의 함수 정의를 virtual char* getMessage( ); 로 해야 한다.

17.

```
void Pen:: special( )
{ penDown( );   P
  int saveX = getX( ); P
  move (3,5);   P
  bool saveStat = isUp; P
  setColor(32): X                    Undefined Function
  int c = color; X
```

```
void ColorPen:: CPspecial( )
{ penDown( );   P                    Derived
  int saveX = getX( ); P             Derived Class Can Use Protected Section
  move (3,5);   CP                   Self Class First
  bool saveStat = isUp; P            Derived
  setColor(32): CP                   Self Class
  int c = color; CP
```

18. Polymorphism으로 인해 스위치 문의 해당 경우가 자동으로 처리된다.

2장. 추상 자료형

1. true
2. false
3. false
4. 가,나,다,라,마
5. 친구관계, 주가변동

6. 외부 사용자가 정보를 조작할 수 없게 된다.(고의 조작을 원하는 자)
   외부 사용자가 정보를 조작할 필요가 없게 된다.(고의 조작을 원하지 않는 자)

7. 함수호출을 통해서 사용
8. 사용자 관점: 세부구현 내용을 알 필요없이 추상적 수준에서 정의하고 사용함
   구현자 관점: 추상적 작업과 그 작업을 구현하기 위한 자료구조로 구성

9. (상대적 답안이 가능)
   예: 유지보수의 용이성(유지보수 비용이 과다할 경우)

10. 추상자료형 = 작업 + 작업 구현을 위한 자료구조
                = member fucntion + member data
11. (예) 전화기

```
class telephoneClass{
public:
        telephone( );   새로운 전화기 객체 생성
        void ring( ); 전화 걸려오기
        void dial(char To[ ]); To의 전화번호로 전화걸기
private:
        char Number[ ]; 자체 전화번호
        bool IsOn;        수화기가 올려져 있는지 확인
};
```

12. 사용자 관점: 필요 작업명
    구현자 관점: 필요 작업명 + 작업 구현방법 + 자료구조

13. (예)

```
class heater{
public:
                heater( );   새로운 히터 생성
```

```
                TurnOn(int Temp1);   Temp1에서 켜짐
                TurnOff(int Temp2);   Temp2에서 꺼짐
                IsOn( ); 현재 켜진 상태인지 확인
private:
                int CurrentTemp;   현재 온도
                bool OnOff; 켜진상태, 꺼진상태 저장변수
                int CostPerDay: 일 난방비용
}
class cooler{
public:
                cooler( ); 새로운 에어컨 생성
                CurrentPower( ); 현재까지 사용 전력량
                ...
private:
                int Watts;   현재 사용전력량
}
```

14. 1) ADT Sensor: 톨 게이트 진입 이벤트 처리
    Find Incoming Car
    Find ByPassing Car
    Find Waiting Car
    2) ADT Charger: 요금 정산 처리
    현금으로 정산
    신용카드로 정산
    쿠폰으로 정산
    3) ADT Interaction: 상호작용 처리
    메뉴화면 입출력 처리
    음성 입출력 처리
    차단기 오르내림 처리

15. 1) 구체적 자료구조를 알 필요가 없다. 2) 모듈화된 단위작업을 끼워 맞추어서 새로운 함수를 조립하는 것이므로 코딩 시간이 단축된다. 3) 개념적인 수준에서 작업이 가능하므로 규모가 큰 프로젝트 수행이 용이해 진다.

3장. 포인터, 배열, 구조체

1. 가: false 나: false 다: false
2. 라
3. 가
4. 나
5. 나
6. 다
7. 나
8. 라
9. 가
10. 나
11. 다
12. 다

13. 가, 나, 다
14. 가
15. 나
16. 나
17. 가
18. 다
19. 가
20. 다
21. 라

22. 다: 문자열 관련 라이브러리 함수를 이용하려면 문자열 끝에 backslash zero를 넣어야 함.

23. 라

24. 1->1024 * 4 2-> 10 * 4 3-> 10 * 4

25. no: 서로 다른 주소에 저장됨
    yes: *s= s[0]= 's'
         *t = t[0] = 's'.

26
```
#define MAXNAME = 20
#define MAXSCORE = 5
typedef struct [
     char InstructorName[MAXNAME];
     int Score[MAXSCORE];
     int Sum;
     char Grade;
     int Satisfaction;
} subjectType;
```

27. 실행 시스템마다 결과가 다르면 그때마다 다른 변수 주소값이 출력됨

28. 리턴 값을 NULL로 되돌려 준다. 이를 확인하고 프로그램에서 빠져나오도록 처리해야 할 책임은 프로그래머에게 있다.

29. P1-> i: 42
    P2 -> k:80

30. 가. P1 ->i:80
    나. P1, P2 ->k:80

31. 함수 리턴 값 없다. 파라미터 배열의 요소는 double 타입이다. 배열 데이터를 읽기만 하고 쓰지는 않겠다.

32. *p 즉 정수값은 변경 불가

33.
ptr1 = = ptr2: 포인터 값, 즉 가리키는 변수의 주소가 동일한지(같은 대상을 가리키는지) 검증

*ptr1 = = *ptr2: 가리키는 변수의 값이 같은지, 가리키는 변수가 달라도 변수값은 같을 수 있음
&ptr1 = = &ptr2: 포인터 변수가 있는 주소가 같은지 검증. 서로 다른 포인터 변수가 동일 주소에 있을 수는 없다.

34. pair p = {0, 1}은 스택을 사용한 정적 메모리 공간에 할당
    *p = new pair는 힙을 사용한 동적 메모리 공간에 할당

35. p

36. p -> *p -> **p(정수)
    *p는 정수변수를 가리키는 포인터이고 p는 *p 즉, 포인터를 가리키는 포인터

37. A + 5 * (sizeof (int))

38. Array of Structure: 배열 요소가 구조체로 이루어짐
    Structure of Array: 구조체 요소가 배열들로 이루어짐

39.
```
typedef struct {
        char Name[20];
        char Sex;
        int Age;
        char Telephone[20];
} recordType;
    typedef recordType TableType[20];
```

40.
가. 변수가 스택의 활성화 레코드에 존재하므로 이 함수 실행이 종료되고 호출함수로 되돌아간 상태에서는 Buffer 배열이 이미 없어짐
나. 정적(Static) 변수에서는 함수 실행이 종료되어도 그 공간이 남아있음
다. 배열은 힙에 존재하지만 그 배열의 시작주소를 가리키는 변수 Buffer.는 지역변수로서 스택에 존재한다. 따라서 이 값이 호출함수로 리턴되어도 호출함수에서 힙에 있는 배열을 참조할 수 있다.

41.
```
typedef struct{
    char Name[20];
    char Sex;
    char Origin[20];
    int Age;
} dogInfoType;
```

42.
```
k = 10          k:10
f1(&k);          k:30
f2(k);
f3(k);           k:80
cout <<k;       displays 80
```

43. 3 9 - 4 8 - 3 7 - 6 6 - 5 6 - 4 6 - 3 6

44. 가, 나, 다

```
typedef struct {
      float real;
      float imaginary;
} complexType;
complexType c;   c.real = 15.0;
```

45.
가. 3
나 & x
다.
```
int* CallByReference (int* q)
{ *q = (*q) * (*q);
  return q;
}
```

46.
```
typedef struct {
        char StudentNumber[MAXNO];
        char Name[MAXNAME];
        int Age;
} studentRecordType;
     studentRecordType A[5000];
```

47.
```
char * strcpy(string1, string2)
{ int i = 0;
  while (string2[i] != '\0')
  {     string1[i] = string2[i];
        i ++;
  }
  string1[i] = '\0';
  return string1;
}
```

48.
```
A: int sum = 0;
    for (int i = 0; i <= (n-1); i ++)
        sum += b[i];
    return sum/n;
B: int sum = 0;
    for (int i = 0; i <= (n-1); i++)
        sum += *(b+i);
    return sum/n;
```

A, B가 바뀌어도 무방하다.

4장. 재귀호출

1. 가. 12 나. 21 다. 45
2. 가. 3 나. -4 다. -4
3. No
4. 라
5. 라
6. 다
7. 라
8. 가
9. 마
10. 가

11.

```
void Write(int N)
{ if (N > 1)
        Write(N-1);
  cout << N;
}
```

12.

7	10	2	8	16	1	3	4	22	14	12	13
				12				**13**		16	22
3	1		**4**		10	7	8				
				7	**8**	12	10				

13.
3 4
4 3
4 3
3 3

14.

```
void WriteBackward(char S[ ], int Size)
{ if (Size > 1)
       WriteBackward(S, Size-1);
  cout << S[Size-1];
}
```

15.

```
(n = = 0)
n/10
```

16.

```
return 0
Count++
Count = NumberEqual(A[ ], N-1, X)
return Count
```

17.

```
int Sum(int A[ ], int N)
{ if (N = = 0)
        return 0;
  else
        return(A[N-1] + Sum(A, N-1);
}
```

18. 탐색 알고리즘 참조

19.

```
int gcd(int m, int n)
{ if (n = = 0)
        return m;
  else
        reutrn gcd(n, m%n);
}
```

20.

가. 메인함수에서 N = 10으로 다음 함수를 호출

```
int Sum(int N)
{ if (N = = 0)
        return 0;
  else
  {   int Number;
      cin >> Number;
      return(Number + Sum(N-1));
  }
}
```

나.

```
int Sum( )
{ int Number
  int Temp = 0;
  for (int i = 0; i <=9 ; i ++)
  {   cin >> Number;
      Temp += Number;
  }
  return Temp;
}
```

21.

가.

```
0   3   7
0   1   2
2   2   2.   No Such Record!
```

나.

```
0   3   7
0   1   2
0   0   0. Got It.
```

22.

가. 1 1
나. 3 2 1 1 2 6
다. -1 -2 -3 ... Stack Overflow

23.
```
if(degenerate case)
{   degenerate case handling
}
else
{   reduce the data size and
    make a recursive call
}
```

24.
```
int Sum2(int N)
{ if (N = = 0)
        return 0;
  else
        return ((N * 10) + 2) + Sum2(N-1));
}
```

25.
A(3,m) = 2^{m+3} -3
A(4,m) = 2^{65536}- 3 . Exponential Algorithm

26.
가. 1번
나. 약 1000번
다. 약 $\log_2 1000$ 번

27.
```
void Asterisk(int N)
{ if (N > 0)
  {   cout << '*';
      Asterisk(N-1);
  }
}
```

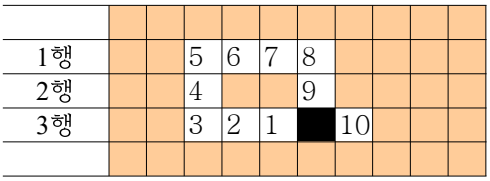
28.
```
void Write3600(int N)
{ if (N <= 3600)
  { cout << N; cout << '\n';
    Write3600(2N);
  }
  cout << N; cout << '\n';
}
```

29.
```
void DisplayNumber(int N)
{ if (N > 0)
```

```
  {   DisplayNumber(N-1);
      cout << N;
      DisplayNumber(N-1);
  }
}
```

30.
첫째 재귀호출 할 때 마다 문제 크기 N이 줄어들어야 하며
둘째 줄어들은 N이 최종적으로는 소스 코드에 명시된 베이스 케이스와 만나야 한다.

31. 경계선 색이 아니면 일단 칠하고 다시 좌, 우, 상, 하의 순서로 보게 된다. 10번 화소는 9번에서 1번으로 되돌아 온 다음에 칠해진다.



32. 1 2 3 4 5 6 7 8 9 10 (줄바꿈 무시)

33. 배열 첫 요소를 피벗으로 선택하는 코드.

5	0	6	1	7	2	8	3	9	4
2		4		3	**5**		7		6

8	0	6	1	7	2	5	3	9	4
4								**8**	9

4	9	8	7	6	5	3	2	1	0
5	0	1	2	3	**4**	6	7	8	9

세 번째 파티션은 주의. 만약 left, right 인덱스가 같으면 while (left < right)에 들어가지 않고 그대로 빠져나와서 a[0]와 a[right]가 스왑됨.

34.
first = 0, last = 4 K
first = 1, last = 4 o
first = 2, last = 4 r
first = 3, last = 4 e
first = 4, last = 4 a
first = 5, last = 4 return ↑

35. Recurse (3*n + 1); 이 무한 루프가 돈다. 이를 Recurse(3/n + 1)로 바꾸되 함수 시작시에 if (n = = 0)로 베이스 케이스를 별도 처리해야 한다.

36. N/2 N%2

37.
n= 1 일때 2 조각

n = 2일 때 4조각
n = 3일때 7조각
n = 4일 때 11조각
n = 5일 때 16조각
따라서 n 번째 자를 때 조각 수 $S_{n+1} = S_n + n$

38. 완전히 동일한 재귀호출임. 첫 번째 else를 사용하는 쪽이 if else로 경우를 나누었다는 점에서 좀더 명확함.

39. 교재참조

5장. 리스트

1. False. 첫 노드에 삽입 삭제가 일어나면 헤드 포인터 값이 변화되어야 하므로 Call by Reference를 사용해야 한다.

2. 다
3. 나
4. 나
5. 라
6. 다
7. 가

8.
```
void Delete(listType *Lptr, int Position)                    삭제함수
{ if (Lptr->Count = = 0)                                     현재 비어있는 리스트
        printf("List Empty");
  else if ((Position > (Lptr->Count)) || (Position < 1))
        printf("Position out of Range");                     이격된 위치 삭제 불허
  else
  { for (int i = Position+1;  i < = Lptr->Count; i++)         삭제위치부터 끝까지
        Lptr->Data[i-1] = Lptr->Data[i];                     왼쪽으로 한 칸씩 이동
    Lptr->Count -= 1;                                                리스트 길이 줄임
  }
}
```

9.
```
void Retrieve(listType *Lptr, int Position, int &Item)
{ if ((Position > (Lptr->Count)) || (Position < 1))
        printf("Position out of Range");                     이격된 위치 검색 불허
  else
  {     int Count = 0;
        Nptr Temp = Lptr->Head;
        for (int Count = 1; Count < Position; Count ++)
             Temp = Temp->Next;
        Item = Temp->Data;
  }
}
```

10.

```
if (Position = = 1)                    첫 위치에 삽입할 경우
{     p->Next = Lptr->Head;            삽입노드의 Next 값이 널로 됨
      Lptr->Head = p;                  헤드가 삽입노드를 가리키게
}
```

11.

```
if ((Position > (Lptr->Count+1)) || (Position < 1)) 에 걸리지 않는다. 이후
for (int i = 1; i < (Position-1); i++)
                  Temp = Temp->Next;       에 의해 세 번째 노드를 가리키고
p->Next = Temp->Next;                  삽입노드의 Next를 널로 세팅
Temp->Next = p;                        세 번째 노드가 삽입된 노드를 가리키게.
```

12.

```
if (Position = = 1)                    첫 노드를 삭제하는 경우
{    Nptr p = Lptr->Head;              삭제될 노드를 가리키는 포인터를 백업
     Lptr->Head = Lptr->Head->Next;    헤드가 널로 바뀜
}
```

13.

```
{ for (int i = 1; i < (Position-1); i++)
          Temp = Temp->Next;           Temp가 2번째 노드를 가리키게
          Nptr p = Temp->Next;         삭제될 노드를 가리키는 포인터를 백업
          Temp->Next = p->Next;        2번째 노드의 Next가 널로 바뀜
 }
```

14.

```
listClass::Delete(int Position)        삭제 함수
{ if (! IsEmpty( ))                    빈 리스트가 아니라면
  { Count --;                          리스트 아이템 수를 줄임
    for (int i = (Position+1); i <= L.Count; ++i)      왼쪽 쉬프트
      Data[i-1] = Data[i];
}
```

15. 교재참조

16. 예를 들어, 어떤 사람의 인적사항이 성명, 주소, 전화번호로 구성되어 있다면 이 것이 각각 구조체의 필드를 구성한다. 또, 성명, 주소, 전화번호 각각이 배열로 잡혀 있다면 전체적으로는 Structure of Arrays에 해당한다.

17.

```
typedef struct{
     char S[20];
     nodeType* Next;
} nodeType;
```

18. 가. 16 나. r 다. Kim

19. for(i = 8; i = 4; i --)

```
        MyArray[i+1] = MyArray[i];
    MyArray[4] = 27;
```

20.
가.
```
    Node* Temp;
    Temp = Head;
    Head = Head-> Next;
    delete Temp;
```

나.
```
Cur = Head;
for (int Count = 1; Count < 4; Count ++)
     Cur = Cur -> Next;
```

다.
```
Cur = Head;
for (int Count = 1; Count < 5; Count ++)
     Cur = Cur -> Next;
Node* Temp = new Node;
Temp->Data = 33;
Temp->Next = NULL;
Cur->Next = Temp;
```

21.
가.
```
Head = new node;
Head->Items = 'B';
Head->Next = new node;
Head->Next->Items = 'E';
Head->Next->Next = new node;
Head->Next->Next->Items = 'J';
Head->Next->Next->Next = NULL;
```
나.
```
ptrType Temp = Head->Next;
Head->Next = Head->Next->Next;
delete Temp;
```

22.
가.
```
ptrType Temp = new node;          새로운 노드를 만들고
Temp->Ch = 'K';                   데이터 채움
Temp->Next = NULL;                일단 Next 값을 널로 초기화
ptrType Prev = NULL;              초기화
ptrType Curr = Head;              초기화
while ((Curr != NULL) && (Curr->Ch < 'K'))
{    Prev = Curr;                 포인터 이동
     Curr = Curr->Next;           포인터 이동
}
if (Prev = = NULL)                while 루프에 들어가지 않았음
```

```
{   if (Curr = = NULL)              빈 리스트
        Head = Temp;                첫 위치에 삽입
    else                            리스트 첫 노드가 'K' 보다 큼
    {   Temp->Next = Head;          새로운 노드가 현재 첫 노드를 가리키게
        Head = Temp;                헤드가 새로운 노드를 가리키게
    }
}
else                                while 루프 실행 중 빠져나옴
{ if (Curr = = NULL)                현재 포인터가 널. 즉 리스트 마지막까지 옴.
      Prev->Next = Temp;            마지막 노드 다음에 새로운 노드 추가
  else                              리스트 중간에 삽입위치 발견
  {   Temp->Next = Curr;            새로운 노드가 자기보다 큰 노드를 가리키게
      Prev->Next = Temp;            삽입직전 노드가 새로운 노드를 가리키게
  }
}
```

나. 교재본문 참조

23.

```
listClass::~listClass
{ while !IsEmpty( )
      Delete(1);
}
```

24.

```
ptrType ReverseList (ptrType& Head)
{ ptrType NewHead, Curr, Prev;
  if (Head != NULL)
  {     Curr = Head;
        NewHead=new node;
        (NewHead->Data = Curr->Data);
        (NewHead->Next = NULL);
        Curr = Curr->Next;
        while (Curr != NULL)
        {   ptrType Temp = new node;
            Temp->Data = Curr->Data;
            Temp->Next = NewHead;
            NewHead = Temp;
            Curr = Curr->Next;
        }
  }
  else
       ( NewHead = NULL );
  return ( NewHead );
}
```

25.

```
Nptr MergeList(Nptr Head1, Nptr Head2)
{   Nptr NewHead = NULL;
    while ((Head1 != NULL) && (Head2 != NULL))
```

```
    {    Nptr Temp = new node;            새로운 노드를 만들고
         Temp->Next = NULL;               Next 값을 초기화
         if (Head1->Data < Head2->Data)   Head1 데이터가 Head2 데이터보자 작으면
         {    Temp->Data = Head1->Data;   새 노드에 작은 데이터를 복사
              InsertLast(NewHead, Temp);   NewHead가 가리키는 리스트의 끝에 삽입
              Head1 = Head1->Next;        다음 노드로 이동
         }
        else
        {    Temp->Data = Head2->Data;
             InsertLast(NewHead, Temp);     NewHead가 가리키는 리스트의 끝에 삽입
             Head2 = Head2->Next;
        }
     }
     if (Head1 = = NULL)                    Head1이 가리키는 리스트가 완료
          for (; Head2 ! = NULL; Head2 = Head2->Next)  Head2에 남아있는 모든 노드를
               InsertLast(NewHead, Head2);   NewHead가 가리키는 리스트의 끝에 삽입
     else if (Head2 = = NULL)
          for (; Head1 ! = NULL; Head1 = Head1->Next)
               InsertLast(NewHead, Head1);
     return NewHead;
}
```

26.

```
typedef struct {
     int Age;
     WaitNode* Next;
} WaitNode;
typedef WaitNode* HeadPtr;

HeadPtr TotalList[200];
```

27.

```
node* ListRetrieve(node* Head)
{ for(node* Temp = Head; Temp!= NULL; Temp=Temp->Next)
        if (Temp->Data = = 25)
             return Temp;
  if (Temp = = NULL)
     return NULL;
}
```

28. 연결 리스트의 노드 순서를 뒤집은 리스트를 만들기 (원래 리스트 변형)

29. 연결 리스트의 노드 순서를 뒤집은 리스트를 만들기 (새로운 리스트 생성)

30. 연결 리스트의 첫 노드를 가리키는 포인터 Head를 Private Variable로 가정

```
void ListClass::DisplayOdd( )
{ for(Nptr Temp = Head; Temp != NULL; Temp = Temp->Next)
        if ((Temp->Data) % 2 = = 1)
             cout << Temp->Data;
}
```

31. 연결 리스트의 첫 노드를 가리키는 포인터 Head를 Private Variable로 가정

```
void ListClass::InsertBeforeMax( )
{ int Max = 32767;
  Nptr MaxPrev;                    가장 큰 노드 바로 앞 노드를 가리키는 포인터
  Nptr Prev = NULL;                초기화
  for (Curr = Head; Curr != NULL; Curr = Curr->Next)  비어있지 않은 리스트
  {   if (Curr->Data > Max)        현재 노드가 이전 최대값보다 크면
      {   Max = Curr->Data;        최대값 변경
          MaxPrev = Prev;          이전 노드 포인터 저장
      }
      Prev = Curr;                 포인터 전진
      Curr = Curr->Next;           포인터 전진
  }
   Nptr Temp = new node;           새로 노드 만들고
   Temp->Data = 25;                데이터 채움
   Temp->Next = MaxPrev->Next;     새 노드가 최대값 노드를 가리키게
   MaxPrev->Next = Temp;
}
```

32. 일단 *p 다음 노드를 Temp로 받아놓는다. *p가 다음 다음 노드를 가리키게 한다. delete 명령에 의해 Temp가 가리키는 노드 공간을 반납한다.

33. 일단 새로운 노드를 만들어서 데이터 필드를 채운후, 그 노드의 Next 필드가 현재 *p 다음 노드를 가리키게 한다. 이후 *p가 새로만든 노드를 가리키게 한다.

34.

```
void DisplaySame(Nptr Head)
{ for (Temp1 = Head; Temp1 != NULL; Temp1 = Temp1->Next)
      for (Temp 2 = Head; Temp2 != NULL; Temp2 = Temp2->Next)
          if((Temp1 != Temp2) && (Temp1->Data = = Temp2->Data))
          {    Cout << Temp1->Data;
               Cout << Temp2->Data;
          }
}
```

35.

```
int Count42(const node* head_ptr)
{ int Count = 0;
  for(node* Temp = head_ptr; Temp != NULL; Temp = Temp->Next)
      if (Temp->Data = = 42)
          Count++;
  return Count;
}
```

36.

```
int sum_data(const node* head_ptr)
{ if (head_ptr = = NULL)
      return 0;
  else
```

```
        return(head_ptr->Data + sum_data(head_prt->Next));
}
```

37.

```
void list_tail_insert(node* head_ptr, int Data)
{ node* Temp = new node;
  Temp->Data = Data;
  Temp->Next = NULL;

  if (head_ptr = = NULL)
        head_ptr = Temp;
  }
  else
  {     node* Prev = NULL;
        node* Curr = head_ptr;
        while (Curr != NULL)
        {     Prev = Curr;
              Curr = Curr->Next;
         }
        Prev->Next = Temp;
  }
}
```

38.

연결 리스트의 끝에 새로운 노드를 삽입하기 위한 명령.

39.

```
void CopyList(const Nptr Head, Nptr& NewHead, Nptr& NewTail)
{ if (Head ! = NULL)
  {   NewTail = new node;
      NewTail->Data = Head->Data;
      NewTail->Next = NULL;
      NewHead = NewTail;
      for (Nptr Temp1 = Head->Next; Temp1 != NULL; Temp1 = Temp1->Next)
      {   Nptr Temp2 = new node;
          Temp2->Data = Temp1->Data;
          Temp2->Next = NULL;
          NewTail->Next = Temp2;
          NewTail = Temp2;
      }
   }
   else
   {     NewHead = NULL;
         NewTail = NULL;
   }
}
```

40. 배열 내부에 존재하는 아이템 개수를 세어야 한다. 이를 위해 필요하다면 배열 아이템의 끝을 의미하는 플래그를 저장할 수 있다. 아이템 수를 확인하고 싶을 때마다 전체 배열을 스캔하는 시간이 걸린다. 연결 리스트일 경우에도 동일하게 헤드로부터 연결 리스트 끝까지

순회하면서 개수를 세어야 한다.

41. 객체가 함수호출의 파라미터로 넘겨질 때, 객체가 함수의 리턴 값으로 되돌려 질 때, 선언시 객체끼리 복사가 일어날 때

42.

```
bool List::DeleteNum(int Num)
{ Nptr Prev = NULL
  Nptr Curr = Head;
  while ((Curr!= NULL) && (Curr->Data != Num))
  {  Prev = Curr;
     Curr = Curr->Next;
  }
  if (Prev = = NULL)
  {  if (Curr = = NULL)
          return FALSE;          빈 리스트
     else                        첫 노드가 찾는 노드
     {   Nptr Temp = Head
         Head = Head->Next;   첫 노드 삭제
         Delete Temp;
         return TRUE;          성공
     }
  }
  if (Curr = = NULL)           리스트 끝까지 찾는 노드 없음
     return FALSE
  else                         리스트 중간에서 찾음
  {  Nptr Temp = Curr;
     Prev->Next = Curr->Next;
     Delete Curr;
     return TRUE;              성공
  }
}
```

43.

```
bool listClass::IsSorted( )
{ for (Nptr Temp = Head; Temp != NULL; Temp = Temp->Next)
  {   if (Temp->Next != NULL)   마지막 노드는 비교대상 없음
      {     if (Temp->Data > Temp->Next->Data)
                return FALSE;
      }
  }
  return TRUE;                  빈 리스트, 하나짜리 리스트는 정렬된 것으로 간주
}
```

44.

```
NodePtr FindOrAdd(NodePtr &Head, Datatype Value )
{ NodePtr Cur = Head;
  NodePtr Temp;
  while ((Cur != NULL) && (Cur->Data != Value))
       Cur = Cur->Next;
```

```
  if (   Cur = = NULL   )            찾는 노드가 없으면
  {     Temp = new NodePtr;
        Temp->Data = Value;
        Temp->Next = Head;
        ( Head = Temp   );            만든 노드를 첫 노드로
        return (Head);
  }
  else
        ( return (Cur) );
 }
```

45. 첫 노드로 삽입할 때에도 별도 처리할 필요 없음(교재참조)

```
void Insert(listType *Lptr, int Position, int Item)
{ 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
        Nptr Temp = Lptr->Head;                  헤드 포인터를 Temp로 복사
        for (int i = 1; i < Position; 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
}
```

46. swapkey가 마지막 노드를 가리키는 일은 없다고 가정한다. 즉, Temp->Next는 항상 널이 아니다라는 것을 전제로 한다.

```
void swap(listnode* first, int swapkey)
{ listnode* Temp = first;
  while ((Temp != NULL) && (Temp->key ! = swapkey))
        Temp = Temp->next
  if (Temp = = NULL)
        cout << "No Such Key";
  else
  {       Nptr After = Temp->Next->Next;
          Temp->prev->next = Temp->next;
          Temp->next->prev = Temp->prev;
          Temp->next->next = Temp;
          Temp->prev = Temp->next;
          Temp->next = After;
  }
}
```

47.

```
Nprt Merge(Nptr a, Nptr b)
{ NPtr first, temp, last;
```

```
  last = NULL;
  while (a != NULL && b != NULL)
  { if (a != NULL)
    {     temp = malloc(Node);
          (temp->Data = a->Data);          a의 데이터 필드 복사
          temp->next = NULL;
          if (first = = NULL)
                ( first = temp );          만든 것이 첫 노드
          if (last != NULL)
                ( last->next = temp  );    만든 것이 마지막 노드 다음으로
          (last = temp );                  마지막 노드는 만든 것
          a = a->next;
    }
    if (b != NULL)
    {     temp = malloc(Node);
          (temp->Data = b->Data );
          temp->next = NULL;
          if (first = = NULL)
            ( first = temp  );
          if (last != NULL)
              ( last->next = temp    );
          last = temp;
          ( b = b->next );
     }
  }
  return first;
}
```

48. 배열 data[ ]에 있는 요소들을 원형 연결 리스트로 구성

49. 왼쪽으로부터 오른쪽으로 가면서 이동해야 덧쓰임(Overwriting)이 방지된다.

50. 교재 참조

6장. 스택

1. 다
2. 라
3. 가
4. 1, ((2-5)+8) / 4

5.
1
8 9 3 2
8 9 5
8 4
8 4 4
8 4 4 3

8 4 1
8 5
1

6. -18
7. True
8. Head->Next = NewHead;
9. 검색문제

10.
```
ListClass::~ListClass( )
{ While (! IsEmpty( ))
        ListDelete(1);
}
```

11.
```
void Push(stackType* Sptr, int Item)
{ if (Sptr->Top = =  MAX)
  {    printf("Stack Full");                    또는 assert macro를 사용
       exit(-1);
  }
  else
       Stpr->Stack[Top++] = Item;
}

int Pop(stacktype* Sptr)
{ if (Top = = 0)
  {    printf("Stack Empty");                   또는 assert macro를 사용
       exit(-1);
  }
  else
       return (Sptr->Stack[--Top]);
}
```

12.
```
boolean IsFull(stackType* Sptr)
{ return (Sptr->Top = =  MAX);
}
```

13.
```
void Push(Nptr Top, int Item)
{ Nptr Temp = (Nptr)malloc(sizeof(node));
  assert (Temp != NULL);
  Temp->Item = Item;
  Temp->Next = Top;
  Top = Temp;
}
```

14.
```
void stackClass::Push(int Item)
```

```
{ Stack[Top++] = Item;
}
```

15.

```
void stackClass::Push(int Item)
{   Nptr NewTop = new node;
    assert (NewTop != NULL);
    NewTop->Data = Item;
    NewTop->Next = Top;
    Top = NewTop;
 }
```

16. 여는 괄호 3, 닫는 괄호 2

17.

```
int stackClass::GetTop( )
{ if (IsEmpty( ))
        cout << "Retrieval on Empty Stack";
  else
        return Top->Data;
}
```

18. C++ 에서는 멤버 함수와 멤버 데이터가 하나의 클래스 내부에 존재한다. 또 멤버 데이터는 객체마다 별도로 존재하며, 멤버 데이터 자체는 멤버 함수에 대해서 일종의 전역변수처럼 사용된다. 따라서 함수 사이에 멤버 데이터를 파라미터로 넘길 필요가 없다.

19.

```
void RemoveAll(struct node Head)
{ While (Head != NULL)
  {    struct node Temp;
       Temp = Head;
       Head = Head->Next;
       delete Temp;
   }
}
```

20. 가. A B C G J H K D F E I
나. B C G J H K
다. F C E I

21.

A
AB
ABC
ABCE
ABCED
ABCE
ABCEG
ABCEGH
ABCEG

ABCEGJ
ABCEGJK
ABCEGJ
ABCEG
ABCE
ABC
ABCF
ABC
AB
ABI
AB
A
Empty. There is no such Path.

22.
```
int stack[200];
int top;
void init( ) { top = -1;}
void push(int val)
{ assert (top != 199);
  stack[++top] = val;
}
void pop( )
{ assert (top != -1);
  top --;
}
```

23.
가. S1: 218866497<- top
나. S1: 3451<-top

24.
```
Stack::Stack( )
{ top = 0;
}
int Stack::Pop( )
{ return A[--top];
}
```

25. 나, 다, 라, 마
26. 20 10
27. 삽입, 삭제가 일어날 때마다 두 번째 요소부터 마지막 요소까지 쉬프트해야 한다.
28.
```
void DecToBin(int N)
{ if (N > 1)
      DecToBin(N / 2);
  cout << N % 2;
}
void DecToBin(int N)
{ stackClass S;
```

```
  while (N > 0)
  {    S.Push(N % 2);
       N = N / 2;
  }
  while (! S.IsEmpty( ))
      cout << S.Pop( );
}
```

29. 배열의 왼쪽 끝을 바닥으로 하는 스택 S1과, 배열의 오른쪽 끝을 바닥으로 하는 스택 S2를 만들면 된다. S1, S2 모두 배열의 가운데를 향해서 자라게 된다. 이렇게 하면 하나의 스택이 커져도 반대쪽 스택이 작다면 별도의 공간을 요하지 않는다. 배열이 꽉찬 경우는 서로 간의 탑 인덱스가 붙어 있을 때이다. 예를 들어 크기 100 짜리 배열에서 S1의 탑 인덱스가 75까지 증가했고, S2의 탑 인덱스가 99부터 76까지 감소했다면 두 개의 배열이 꽉 찬 상태가 된다.

30.

```
int stackClass::CountX(stackClass& S, int X)
{ int Count = 0;
  stackClass B;
  while (!S.IsEmpty( ))
  { if (S.Pop( ) = = X)
       Count ++;
    B.Push(X);
  }
  while (!B.IsEmpty( ))
       S.Push(B.Pop( ));
  return Count;
}
```

31.

```
void stackClass::RecurseStack(stackClass& S)
{ char Temp;
  if (!S.IsEmpty( ))
  {    Temp = S.Pop( );
       RecurseStack(S);
       cout << Temp;
  }
}
```

32.

```
bool StackType::Push(const ItemType Value)
{ bool Success = ( FALSE   );
  if(!IsFull())
  {    top++;
      ( Stack[top] = Value );
      ( Success = TRUE   );
  }
  return Success;
}
```

33.
```
main()
{ int x = 1;
  while ( n <= 10    )
 {       (      printf("%i\n", n       );
          (     n++                      );
  }
}
```

34. 원형 배열로 구현하면 된다. 예를 들어 꽉찬 상태에서 푸쉬가 일어나면 그것을 인덱스 0에 삽입함으로써 스택의 바닥이 자동으로 삭제된다. 이 경우 탑 인덱스를 늘이기 위해서는 는 top++ 대신 (top++ % Array Size)를 사용해야 한다. 예를 들어, 크기 10인 배열 인덱스 0 ... 9까지 데이터가 차 있고 현재의 탑 인덱스를 9라고 하자. 이때 새로운 푸쉬가 들어오면 삽입 인덱스 위치는 (9+1)인 10이 아니라 (9+1) % 10 = 0 이어야 한다.

35. 본문참조
36. 본문참조

7장. 큐

1. 다
2. 가
3. 다
4. 나
5. 가
6. False
7. 나, 다, 마
8. 가: S 나: Q 다. S 라. Q

9.
가. A B D F C G K I E J H
나. B C G K J H

10.
DFS: A B C D E F G H I
BFS: A B G C D E H F I

11.
```
ptrType Temp = malloc(sizeof (node));
assert (Temp != NULL);
Temp->Item = 3;
Temp->Next = BackPtr->Next;
BackPtr->Next = Temp;
BackPtr = Temp;
```

12. 삽입이 일어날 때마다 뒤쪽 인덱스가 증가하고, 삭제가 일어날 때마다 앞쪽 인덱스가 증가하기 때문에 전체적을 데이터 집합이 오른쪽으로 이동한다.

13.
가.

```
QueueClass::QueueAdd(int Newitem, boolean& Success)
{ if (L.ListIsFull( ))
      Success = FALSE;
  else
  {    int Position = L.ListLength( );
       L.ListInsert(Position+1, Newitem, Success);
  }
}
```

나.

```
StackClass::~StackClass( )
{ boolean Success;
  while(!L.ListIsEmpty( ))
        L.ListDelete(1, Success);
}
```

14. 교재참조
15. 먼저 Front를 삭제해 버리면 두 번째 노드를 가리키는 포인터가 없어지기 때문에 안된다.
16.

```
int queueClass::Remove( )
{ int Temp;
  Nptr Front = Rear->Next;
  if (GetSize( ) >= 2)
  {    Rear->Next = Front->Next;
       Temp = Front->Data;
       delete Front;
  }
  else if (GetSize( ) = = 1)
  {    Rear = NULL;
       Temp = Front->Data;
       delete Front;
  }
  else if (GetSize( ) = = 0)
  {    cout << "Deletion on Empty Queue";
       exit(-1);
  }
  return Temp;
}
```

17. 예를 들어 5, 7 이라는 데이터를 각각 인덱스 0, 1에 삽입한 후에 프런트인 5를 삭제하면 이제 프런트와 리어의 인덱스는 모두 1이 된다.

18.
가.

```
queueClass::queueClass( )
{
}
```

리스트 클래스 생성자가 자동으로 실행됨

나.
queueClass::~queueClass( )
{
}　　리스트 클래스 소멸자가 자동으로 실행됨
다.
bool queueClass::IsEmpty( )
{ return L.IsEmpty( );
}
라.
bool queueClass::IsFull( )
{ return L.IsFull( );
}

19.
가. (18 + 21 + 16) / 3
나. (3 + 4 + 1) / 3
다. (2 + 6 + 19) /3

20. 줄 뒤에 서야할 차가 줄의 맨 앞으로 간다는 점에서 덱 구조와 유사함.
21.
Insert 'a'; H=T=0
Insert 'b'; H=0, T=1
Insert 'c'; H=0, T=2
Remove; H=1, T=2
Remove; H=T=2
Insert 'd'; H=2, T=3
Insert 'e'; H=2, T=0
Remove; H=3, T=0
Remove; H=T=0

22.

	A	B	C			
R		B	C			
R			C			
A D			C	D		
A E			C	D	E	
R				D	E	
A A				D	E	A
A B	B			D	E	A
R	B				E	A
R	B					A

23.
가. 느리다
나. 빠르다
다. 빠르다
라. 빠르다.

24. Q1: 9 3 6 7 4 4　Q2:　　S:

25.
```
void StackToQueue(stackClass S, queueClass Q)
{ stackClass Temp;
  while (!S.IsEmpty( ))
      Temp.Push(S.Pop( ));
  while (!Temp.IsEmpty( ))
      Q.Add(Temp.Pop( ));
}
```

26.
```
void QueueToStack(queueClass Q, stackClass S)
{ stackClass Temp;
  while (!Q.IsEmpty( ))
      Temp.Push(Q.Remove( ));
  while (!Temp.IsEmpty( ))
      S.Push(Temp.Pop( ));
}
```

27. 스택 S1의 바닥과 스택 S2의 바닥이 일치되게 마주 놓는다. S1에서는 팝 작업만 가능하고 S2에서는 푸쉬 작업만 가능하게 한다. 이렇게 되면 S1의 팝 작업을 큐 프런트의 삭제로 S2의 푸쉬 작업을 큐 리어의 삽입으로 간주할 수 있다.

28. 큐의 리어를 첫 노드로 잡으면 삽입을 빠르지만 삭제를 위해서는 연결 리스트의 끝까지 포인터를 따라가는 시간을 요한다. 큐의 리어를 마지막 노드로 잡으면 삭제를 빠르지만 삽이를 위해서는 연결 리스트의 끝까지 포인터를 따라가는 시간을 요한다.

29. 인덱스 2번

30. (CurrRead+1) % MAX

31. 교재참조

32.
```
Constructor Function,
Destructor Function,
AddFront( );
RemoveFront( );
AddRear( );
RemoveRear( );
IsFull( );
IsEmpty( );
```

코드 7-4의 자료구조를 사용하되, 위 함수를 추가하면 된다.

33.
```
void copyQueue(Queue &Source, Queue& Dest)
{ bool Success; int Data; Queue Temp;
  while(!Source.IsEmpty( ))
  {   Source.Retrieve(Data);
      Temp.Add(Data);
```

```
        Dest.Add(Data);
        Source.Remove( );
    }
    while(!Temp.IsEmpty( ))
    {     Temp.Retrieve(Data);
          Source.Add(Data);
          Temp.Remove( );
    }
}
```

34.

```
#include <Queue.h>
const int N = 41;
const int M = 3;
main ( )
{ int i;
   queueClass Q;
      for (i = 1; i <= N; i ++)
       (   Q.Add(i) );
   while ( !Q.IsEmpty( ) )
   {       for (i = 0; i < (M-1); i ++)
           { Q.Add(Q.GetFront( ));
             Q.Remove( );
           }
           cout << Q.Remove( );
   }
}
```

8장. 알고리즘과 효율

1. False
2. False
3. False
4. False
5. 가
6. 50 * 60/.3
7. 가
8. 다
9. False
10. 나
11. 가
12. 나
13. 나
14. 라 (log 2000 = log 2 + log 1000)
15. 다 16. 가 17. 마
18. 나(logN with base 10)
19. 나

20.
가. O(N제곱) 나. O(N제곱) 다. O(N) 라. O(N) 마. O(1)

21.
100,000 lg(N) N0.5 ln(N) 9N N2lg(N) 47N+15N2+3N3 N5 N!

22.
가. O(N의 6승) 나. O(N) 다, O(N 제곱) 라. O(N제곱)

23. 프린트 문이 (N-3+1)번 수행된다.

24. 테스트 조건을 비교하는 명령을 제외하고 할당문과 프린트 문은
모두 ((N-1) * 2 + 2)N + 2)N 번 수행된다.

25. O(N의 3승)

26. O(N의 제곱 * lgN)

27. O(NlgN)

28.

	Insert	Delete	IsMember	getMin	getSuccessor
정렬된 배열	O(N) 첫 위치 삽입이면 나머지 모두 뒤로 이동작업	O(N) 첫 위치 삭제면 나머지 모두 앞으로 이동작업	O(lgN) 이진탐색 사용	오름차순이면 O(1)	O(lgN) 이진탐색사용
정렬 안 된 배열	O(1) 마지막에 삽입	O(N) 삭제될 키이를 지닌 레코드를 찾는 작업	O(N)	O(N)	O(N)
정렬된 연결 리스트	O(N) 오름차순일 때 가장 큰 키이를 지닌 레코드 삽입	O(N) 가장 큰 키이를 지닌 레코드 삭제	O(N) 이진탐색 사용하기 어려움.	오름차순이면 O(1)	O(N)
정렬 안 된 연결 리스트	O(1) 첫 노드로 삽입	O(N) 마지막에 삭제 키이를 지닌 레코드	O(N)	O(N)	O(N)

[표 8-7] 문제 28

29. N이 작을 때에는 2N승이 1000N2 보다 작기 때문에 유리하다. 예를 들어 N = 15라면 2의 15승은 32768 정도이지만 1000N2은 225000으로 더 크다. 그러나 N = 20이라면 2의 15승은 1048576이지만 1000N2은 400000으로 더 작다. N이 증가함에 따라 크기 격차는 증가한다.

30.
전제조건:
프로그램/함수가 시작하기 전에 만족되어야 하는 사항으로서 프로그램/함수 시작부분에 주석 형식으로 나열할 수 있음.

후속조건
프로그램/함수가 끝날 때에 만족되어야 하는 사항으로서 프로그램/함수 시작부분에 주석 형식으로 나열할 수 있음.

루프 인베리언트
매번 루프를 돌때 마다, 들어가기 전에도 만족되고 빠져나온 후에도 만족되는 불변의 사실로서 루프를 포함하는 프로그램의 정확성 검증에 사용함.

31.
```
void FindOdd(int A[ ], int B[ ])
//전제조건: 배열 A, B는 크기가 N개인 배열이다. 이 배열을 정수로 구성되어 있으며 내부
//요소 N개가 모두 채워져 있는 상태이다.
//후속조건: A 배열 요소중 홀수만 모두 B 배열로 복사되어 들어간다. B 배열은 중간에
// 빈 칸이 없도록 한다.
//루프 인베리언트: A 배열의 인덱스 i 왼쪽에 존재하는 모든 홀수는 B 배열로 복사되었다.
{ int i = j = 0;
    while (i < N)
  {    if ((A[i] % 2) = = 1)
       {    B[j] = A[i];
            j++;
       }
       i++;
  }
}
```

32. 문법오류, 의미상의 오류, 논리적 오류

33.
가. O(N의 3제곱) 나. O(N제곱) 다. O(lgN) 라.O(2^N)
“라”는 크기 N의 문제가 2개의 크기 (N-1) 문제로, 다시 각각이 2 개의 크기 (N-2)의 문제로 바뀐다. 따라서 재귀호출의 회수는 1 + 2 + 4 + 8 + ... + 2^N 번 일어나고 호출이 일어날 때마다 매번 if 문에 의해 베이스케이스인지 비교가 가해져야 한다.

34.
배열로 구현했을 때는 배열 인덱스에 의해 단번에 찾을 수 있으니 O(1)의 효율이다. 연결리스트일 경우에는 최악의 경우 가장 마지막 노드를 찾아야 되고 이 경우 헤드 포인터로부터 시작해서 리스트 내의 모든 노드를 읽어와서 그 Next 필드를 읽고 따라가야 한다. 따라서 O(N)의 효율이다.

35. N, N/2, N/4 로 줄어들게 되므로 O(lgN)의 복잡도이다.

36. 가장 내부의 Sum++; 명령어는 1 + 2 + 4 + 8 + ... + N 번 수행된다. 이는 2^0, 2^1, 2^2, ..., 2^{lgN} 과도 같으므로 등비 수열의 합 공식에 의해 O(2^{lgN})에 해당한다.

37. O(N)

38. a=2(2개의 문제로 분할), c=1, k=1(합병에 걸리는 시간이 N)

39. 가. O(1) 나. O(NlgN)

9장. 정렬 알고리즘과 효율

1. 다
2. 나
3. 다
4. 가, 나, 라, 마
5. 가, 다, 라, 마
6. 나, 다, 라, 마
7. 다
8. 가
9. 라

10. 단계별로 가장 큰 것을 선택하여 오른쪽 끝으로 이동시킨다. $O(N^2)$, 단계의 수가 대략 N 개 이고, 매 단계별로 가장 큰 것을 찾는 작업이 (N-1) + (N-2) + ... + 1이기 때문이다

11. $O(N^2)$, 이미 정렬된 데이터

12.

```
(Quick Sort)
Sort
{ Partition;
  Sort(Left);
  Sort(Right);
}

(Merge Sort)
Sort
{   Sort(Left);
  Sort(Right);
  Merge;
}
```

13.
가.

성명	학년	학점	주소지
최지우	1	C	대구
박지은	1	B	서울
서철훈	1	A	서울
공지영	3	B	전주
조만식	3	C	인천
김갑이	3	A	수원
차은지	4	A	용인

[표 9-8]

성명	학년	학점	주소지
공지영	3	B	전주
김갑이	3	A	수원
박지은	1	B	서울
서철훈	1	A	서울
조만식	3	C	인천
차은지	4	A	용인
최지우	1	C	대구

[표 9-9] 정렬결과

나.

인덱스	성명	학년	학점	주소지
0	최지우	1	C	대구
1	박지은	1	B	서울
2	서철훈	1	A	서울
3	공지영	3	B	전주
4	조만식	3	C	인천
5	김갑이	3	A	수원
6	차은지	4	A	용인

[표 9-10]

인덱스	0	1	2	3	4	5	6
원래 배열의 인덱스	3	5	1	2	4	6	0

표 11 간접정렬의 결과

14. 버블 정렬이나 삽입 정렬에서 요구하는 스왑의 횟수가 많기 때문이다. 큰 레코드를 스왑하는 것은 그 만큼 많은 데이터를 읽고 써야 하므로 효율에 영향을 준다.

15.
가.

5	3	8	9	1	7	0	2	6	4
5	3	8	4	1	7	0	2	6	9

표 12

나.

5	3	8	9	1	7	0	2	6	4
0	3	8	4	1	7	5	2	6	9

표 13

다.

5	3	8	9	1	7	0	2	6	4
3	5	8	9	1	7	0	2	6	4

표 14

라.

5	3	8	9	1	7	0	2	6	4
1	3	0	2	5	4	8	9	6	7

표 15

16. 매 단계별로 파티션 결과 나누어진 좌 우 배열의 크기가 동일할 때 단계수는 lgN으로 최소이며 이 경우 정렬의 효율은 O(NlgN)이 된다.

17. 마지막 요소를 피벗으로 잡았을 때

피벗과 같은 키 스와핑:

17	2	5	5	5	8	1	2	5
2		1	5	**5**		5	17	5

표 16

피벗과 같은 키는 스와핑 하지 않음:

17	2	5	5	5	8	1	2	5
2					1	**5**	17	8

표 17

18.

1 first	2	5	6	7 middle	8	10	12	25 last
1	2	5	6	25	8	10	12	7
				7				25

표 18

19. 쾌속정렬은 불안정 정렬이다. 다음 배열은 첫 필드인 알파벳 기준으로 정렬이 완료된 것이다. 둘째 필드인 숫자를 기준으로 정렬한 결과는 다음과 같다. A12, B12의 순서가 바뀐 것을 알 수 있다.

A 20	**A 12**	B 3	**B 12**	B 1	C 15	C 1	C 11	D 4
C 1	B 1	B 3	D 4*	A 12	C 15	A 20	C 11	B 12
C 1	B 1	B 3*		C 11	**B 12***	A 20	A 12	C 15
C 1	B 1*					**A 12**	C 15*	A 20

표 19

20. 4, 5, 9가 피벗일 가능성이 있다. 이들을 기준으로 파티션 되어 있기 때문이다.

21. 쾌속정렬이 베이스 케이스 근처에 오면 예를 들어 데이터 한 개, 두 개를 놓고 파티션을 가하기도 한다. 또 이를 위해 인덱스가 교차하는지 비교해야 하고 필요한 스왑을 가하는데 많은 시간을 요한다. 따라서 쾌속정렬 도중에 데이터 수가 일정 수 이하이면 이후에는 삽입정렬을 가하기도 한다. 불필요한 파티션 시간을 줄이기 위해서이다.

22.
가.

5	3	8	9	1	7	0	2	6	4
0	3	4	2	1	**5**	7	9	6	8

표 20

나.

5	3	8	9	1	7	0	2	6	4
5	3	8	9	1	7	0	2	6	4

표 21

다.

5	3	8	9	1	7	0	2	6	4
1	3	5	8	9	0	2	4	6	7

표 22

23.

```
void Merge(int Data[ ], int Group1, int Group2)
{ int Temp[ ];
  int k = 0;
  int i = 0;
  int j = Group1;
  while ((i < Group1) && (j < (Group1 + Group2)))
  {    if (Data[i] < Data[j])
       {    Temp[k++] = Data[i]
            i++;
       }
       else
       {     Temp[k++] = Data[j]
             j++;
       }
    }
    while(i < Group1)
    {    Temp[k++] = Data[i]
         i++;
    }
    while(j < (Group1 + Group2))
    {    Temp[k++] = Data[j]
         j++;
    }
    for (k = 0; k <(Group1 + Group2); k++)
        Data[k] = Temp[k];
}
```

24. O(N) 25. O(NlgN) 26. $O(N^2)$

27.

가. 버블정렬, 삽입정렬 나. 선택정렬, 합병정렬

28.

가.

25	19	3	38	62	95	22	17	4	7	12	18	5	36	55
					55									95
				36									62	
					5							55		
			18								38			
				12						36				
7									25					
						4		22						
	17						19							
			4			18								
	5				17									
				12										
4			7											
	3	5												
3	4													
3														

표 23

나.

25	19	3	38	62	95	22	17	4	7	12	18	5	36	55
3		25			55									95
	4			36				19					62	
		5			25							55		
			7						18		38			
				12						36				
					17		18		25					
						18	19	22						
							19							

표 24

다.

25	19	3	38	62	95	22	17	4	7	12	18	5	36	55
19	25													
3	19	25												
3	19	25	38											
3	19	25	38	62										
3	19	25	38	62	95									
3	19	22	25	38	62	95								
3	17	19	22	25	38	62	95							
3	4	17	19	22	25	38	62	95						
3	4	7	17	19	22	25	38	62	95					
3	4	7	12	17	19	22	25	38	62	95				
3	4	7	12	17	18	19	22	25	38	62	95			
3	4	5	7	12	17	18	19	22	25	38	62	95		
3	4	5	7	12	17	18	19	22	25	36	38	62	95	
3	4	5	7	12	17	18	19	22	25	36	38	55	62	95

표 25

라.

25	19	3	38	62	95	22	17	4	7	12	18	5	36	55
17	4	3	12	18	5	22	25	19	7	38	62	95	36	55
7	4	3	12	18	5	17	25	19	22	36	55	95	38	62
3	4	5	7	12	17	18	19	22	25	36	38	55	62	95

표 26

29. 스왑을 자주 할수록 파티션 결과 균형이 좋다.

30. 파티션 결과 균형을 좋게 하기 위해서

31. 3-Way Merge Sort
1단계
파일 1: FNU * ALS * AGL *
파일 2: ADM * ORT * IOR *
파일 3: ENT * GIN * HMT *

2단계
파일 1: ADEFMNNTU *
파일 2: AGILNORST *
파일 3: AGHILMORT *

3단계
AAADEFGGHIILLMMNNOORRSTTTU

32. 카운트 배열을 만들어야 함.

0	1	2	...	15	...	50
	3	3	...	2	...	1

표 27

0	1	2	...	15	...	50
	3	6	...	8	...	9

표 28

33.
가. LSD

0	f	o	e
1	c	a	t
2	f	e	e
3	b	o	y
4	t	i	p
5	t	e	e
6	t	o	y
7	f	i	n

[표 9-28] 문34

0	f	o	e
1	f	e	e
2	t	e	e
3	f	i	n
4	t	i	p
5	c	a	t
6	b	o	y
7	t	o	y

[표 9-28] 문34

0	c	a	t
1	f	e	e
2	t	e	e
3	f	i	n
4	t	i	p
5	f	o	e
6	b	o	y
7	t	o	y

[표 9-28] 문34

0	b	o	y
1	c	a	t
2	f	e	e
3	f	i	n
4	f	o	e
5	t	e	e
6	t	i	p
7	t	o	y

[표 9-28] 문34

나. MSD

0	f	o	e
1	c	a	t
2	f	e	e
3	b	o	y
4	t	i	p
5	t	e	e
6	t	o	y
7	f	i	n

[표 9-28] 문34

0	b	o	y
1	c	a	t
2	f	o	e
3	f	e	e
4	f	i	n
5	t	i	p
6	t	e	e
7	t	o	y

[표 9-28] 문34

0	b	o	y
1	c	a	t
2	f	e	e
3	f	i	n
4	f	o	e
5	t	e	e
6	t	i	p
7	t	o	y

[표 9-28] 문34

35. 3-Way Partitioning의 문제(Dutch Flag Problem)

피벗을 중심으로 배열을 일단 4 부분으로 나눈다.

equal to V, less than V, greater than V, equal to V

이렇게 하기 위해서는 업 포인터는 피벗보다 크거나 같은 것, 다운 포인터는 피벗보다 작거나 같은 것을 찾아나가면 된다. 업포인터가 피벗보다 큰 것, 다운 포인터가 피벗보다 작은 것을 만나면 서로 스왑한다. 만약 업 포인터가 피벗과 같은 것을 만났다면 그것과 배열 첫 요소를 스왑한다. 물론 다음에 또 피벗과 같은 것을 만나면 배열의 둘 째 요소와 스왑하면 된다. 다운 포인터에 대해서도 동일한 방법을 사용한다.

최종적으로는 위 파티션된 모습에서 피벗과 같은 것을 가운데로 스왑하면 원하는 모습이 된다.

36. 합병정렬

37.

```
int partition(int A[ ], int first, int last)
{   int low, high, pivotindex, p;
    p = A[first];                          첫 요소를 피벗으로
    low = first + 1;
    high = last;
    while(low < high)
    {   while(p >= A[low]) low++;
        (while (p <= A[high]) high-- )
        if(low < high)
              Swap(A, low, high);          A[low]와 A[high]를 스왑
    }
    (Swap(A, first, high )                 A[first]와 A[high]를 스왑
    return (high);                         피벗 인덱스를 리턴
}

void main()
{ int Array[MAX], size;
  size = InputArray(Array);                배열 요소의 개수를 리턴 하는 함수
  Qsort(Array, 0, size-1);
```

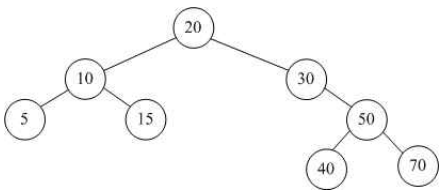
}

39.
가. 선택정렬
나. swap 함수의 마지막 코드는 a[j] = temp; 이어야 한다.
다. 정렬될 데이터의 개수
라. 안정정렬이 아니다. 가장 큰 데이터가 한번에 끝으로 스왑되어 이동한다.

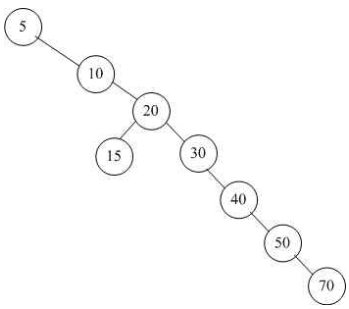
10장. 트리

1. False
2. True
3. False
4. True
5. 나
6. 나, 다
7. 라
8. 후위순회
9. 교재 용어정의 참조

10.
가.

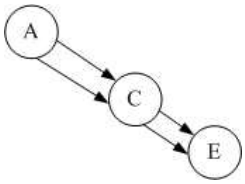


나.



11.
가. Yes 나. Yes 다. No
라. 노드 R을 L에 복사한다. 노드 R의 RChild인 T를 노드 R의 Parent인 노드 U의 LChild로 붙인다.

12.



13. 가. 82　　나. 3

14. 그림 10-16에 표시됨

15.

```
Nptr Insert(Nptr T, int Key)
{ if (T = = NULL)
  {     T = (node*)malloc(sizeof(node));
          T->Data.Key = Key;
          T->LChild = NULL; T->RChild = NULL;
    }
  else if (T->Data.Key > Key)
       T->LChild = Insert(T->LChild, Key);
  else if (T->Data.Key < Key)
          T->RChild = Insert(T->RChild, Key);
  else
          printf("Insertion with Duplicate Key.\n");
  return T;
  }
```

16.

```
void Insert(Nptr& T, int Key)
{ if (T = = NULL)
  {     T = new node;
        T->Data.Key = Key;
        T->LChild = NULL; T->RChild = NULL;
   }
   else if (T->Data.Key > Key)
        Insert(T->LChild, Key);
   else
        Insert(T->RChild, Key);
   }
}
```

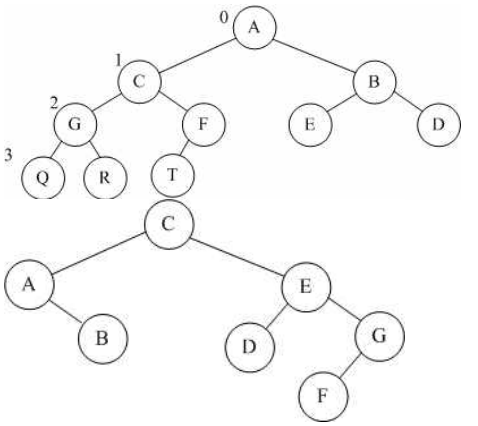
17. G, C, A, E, L의 전위순회 순서로 노드가 만들어진다.

18. 자식노드 없을 때, 하나 일때, 둘 일 때 각각에 대한 삭제 방식을 적용(교재 참조)

19. 어떤 노드에 왼쪽 자식노드를 삽입하고자 할 때, 그 노드의 LChild 변수를 참조호출로 넘겨야 그것이 직접 삽입된 자식노드를 가리킨다. 만약에 값호출로 넘어가면 이러한 연결은 이루어지지 않는다.

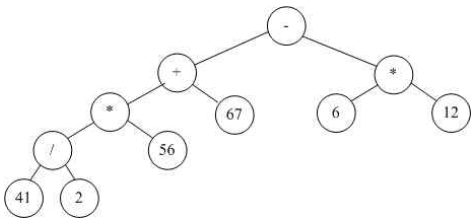
20. 30가지

21. (예)



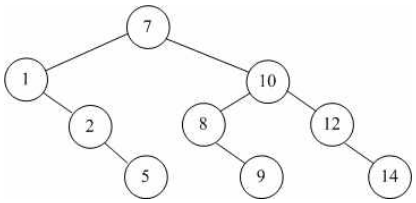
22. 만약에 LChild가 있다면 중위 후속자는 그 노드이거나 거기에 이어진 노드이기 때문이다.

23.

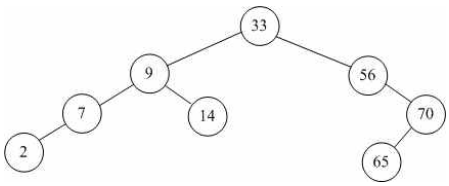


24. 2, 5, 6 이 RChild만으로 연결되어 있는 경우

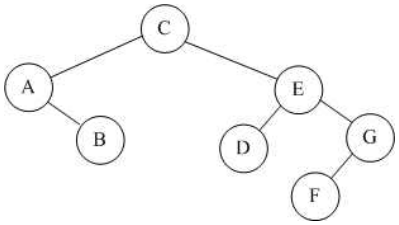
25.



26.



27.



28.

```
void binTreeClass::DestroyTree(ptrType& TreePtr)
{ if (TreePtr!=NULL)
      {    (DestroyTree(TreePtr->LChild));
           (DestroyTree(TreePtr->RChild));
           (Delete TreePtr;)
           TreePtr = NULL;
       }
}
```

29.

```
Temp = Root;
Done = FALSE;
while (!Done)
{    if (Temp = = NULL)
     {    printf("Data not found\n");
          ( Done = TRUE);
     }
     else if (Temp->Data = = key)
     {    printf("Data found\n");
          Done = TRUE;
     }
     else if (Temp->Data > key)
          (Temp = Temp->LChild);
     else
          Temp = Temp->RChild;
}
```

30.

```
node *Minimum(node *Root)
{ node *Tree = Root;
  int Done = FALSE:
  while (!Done)
  {    if ( Tree -> LChild = = NULL )
            Done = TRUE;
       else
            Tree = Tree->LChild;
  }
  return(Tree);
}
```

31.

```
Temp = Root;
Done = FALSE;
while (!Done)
{    if (Temp = = NULL)                          빈 트리
    {  (Root = NewNode);                         루트에 직접 붙임
       Done = TRUE;
    }
    else if (Temp->Data = = NewNode->Data)
       Done = TRUE;                              이미 같은 노드가 존재
    else if (NewNode->Data < Temp->Data)
    {  if (Temp->LChild = = NULL)
       {    (Temp->LChild = NewNode);            왼쪽 자식노드 위치에 삽입
            Done = TRUE;
       }
       else
            (Temp = Temp->LChild);               왼쪽 자식으로 이동
    }
    else
    {    if (Temp->RChild = = NULL)
         {    Temp->RChild = NewNode;
              Done = TRUE;
         }
         else
              (Temp = Temp->RChild);                    오른쪽 자식으로 이동
    }
}
```

32.

```
int TreeSize(Nptr T)
{ int Count = 1;
  if (T->LChild != NULL)
  {    Count += ( TreeSize(T->LChild) );
  }
  if (T->RChild != NULL)
  {    Count += ( TreeSize(T->RChild) );
  }
  return Count;
}
```

33.

```
Nptr FindMin(Nptr T)
{ if (T != NULL)
       while ( T->LChild != NULL )
            T = T->LChild;
  return T;
}
```

34.

```
Nptr FindMax(Nptr T)
{ if (T != NULL)
```

```
        while ( T->RChild != NULL )
            ( T = T->RChild );
    return T;
}
```

35.

```
Nptr FindNodeWithKey(Nptr T, int SearchKey)
{ while (T != NULL)
    {     if ( T->Data > SearchKey)
                T = T->LChild;
          else if ( T->Data < SearchKey)
                T = T->RChild;
          else
                return T;
    }
    return NULL;
}
```

36.

```
struct treenode
{ char Key[10];
    int Height;                                  노드마다 높이정보를 지님
    struct treenode *LChild, *RChild;
};

void ComputeHeights(struct treenode *Root);
{ int hLChild, hRChild;
    if (Root->LChild = = NULL)                   왼쪽 자식이 없다면
          hLChild = -1;                          왼쪽 서브트리의 높이는 -1
    else                                         왼쪽 자식이 있다면
    {     ComputeHeights(Root->LChild);          왼쪽 자식의 높이계산
          hLChild = Root->LChild->Height;        왼쪽 서브트리 높이는 왼쪽자식의 높이
    }
    if (Root->RChild == NULL);                   오른쪽 자식이 없다면
          (hRChild = -1 );                       오른쪽 서브트리의 높이는 -1
    else                                         오른쪽 자식이 있다면
    {     (ComputeHeights(Root->RChild));        오른쪽 자식의 높이 계산
          ( hRChild = Root->RChild->Height); 서브트리 높이는 오른쪽 자식의 높이
    }
    Root->Height = max(hLChild, hRChild) + 1; 노드의 높이는 서브트리 중 큰 것 + 1
}
```

37.

```
struct treenode  {
     char Key[10];
     struct treenode *Left, *Right;
};

treenode *search(treenode *Root, char S[ ])
{ nodetype *p;
```

```
  int cmpval;                                  문자열 비교의 결과를 저장
  while (Root != NULL)                 빈 트리가 아니면
 {    cmpval = strcmp(Root->Key, S);          문자열 비교
      if (cmpval = = 0)                       결과가 0이면 동일 문자
           return Root;
      else if ( cmpval < 0   )
      {    if (Root->Right != NULL)
                Root = Root->Right;
           else
                return Root;                  삽입위치의 노드
      }
      else
      {   if ( Root->Left != NULL)
               Root = Root->Left;
          else
                return Root;
      }
  }
  return NULL;                                 빈 트리라면 널을 리턴
}
```

38.

```
typedef struct
{ DataType Value;   ListPtr Next;
} ListNode;       ListNode* ListPtr;

typedef struct
{ DataType Value;   TreePtr Left, Right;
} TreeNode;       TreeNode* TreePtr;

void TreeToList(TreePtr Tree, ListPtr &List )
{ if (Tree = = NULL)
       return;
  else
  {    ( TreeToList(Tree->Left));
       List.AddToEnd(Tree->Value, List);
       ( TreeToList(Tree->Right));
  }
}
```

39.

```
int Max(int x, int y)
{ return (x > y ? x : y);
}

int MaxElement(Nptr * T)
{ if( T = = NULL )
       return -1;
  else
```

```
    return Max(T->Data, Max(MaxElement(T->Left), MaxElement(T->Right)));
}
```

40.

```
int CountSemi(Nptr &T)
{ if (T = = NULL)
        return 0;
  else if (T->LChild = = NULL && T->RChild = = NULL)
        return 0;
  else if (T->LChild = = NULL || T->RChild = = NULL)
        return (1 + CountSemi(T->LChild) + CountSemi(T->RChild));
  else
        return (CountSemi(T->LChild) + CountSemi(T->RChild));
}
```

41.

```
void DeleteMin(Nptr &T)
{ if (T = = NULL)
        return;
  else if (T->LChild = = NULL)
  {     Nptr Temp = T;
        (T = NULL );
        delete Temp;
        return;
  }
  else DeleteMin( T->LChild );
}
```

위 코드에서 T가 Call by Value로 전달되면 가장 작은 키이를 지닌 노드가 삭제되지 않고 부모노드에 그대로 붙어 있게 된다.

42.

가.

```
int CountNode(Nptr T)
{ if (T = = NULL)
        return 0;
  else if ((T->LChild == NULL) && (T->RChild == NULL))
        return 1;
  else if (T->LChild == NULL)
         return (1 + CountNode(T->RChild));
  else if (T->RChild == NULL)
         return (1 + CountNode(T->LChild));
  else
         return (CountNode(T->LChild) + CountNode(T->RChild));
}
```

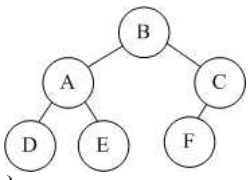
나.

```
int CountLeafNode(Nptr T)
{ if (T = = NULL)                              빈 트리
        return 0;
```

```
   else if ((T->LChild == NULL) && (T->RChild == NULL))
         return 1;
   else if (T->LChild == NULL)
          return (CountLeafNode(T->RChild));
   else if (T->RChild == NULL)
          return (CountLeafNode(T->LChild));
   else
          return (CountLeafNode(T->LChild) + CountLeafNode(T->RChild));
}
```

다.

```
int CountFullNode(Nptr T)
{ if (T = = NULL)
         return 0;
   else if ((T->LChild == NULL) && (T->RChild == NULL))
         return 0;
   else if (T->LChild == NULL)
          return (CountFullNode(T->RChild));
   else if (T->RChild == NULL)
          return (CountFullNode(T->LChild));
   else
          return (1 +CountFullNode(T->LChild) + CountFullNode(T->RChild) );
```




```
}
```

43. 포화이진이나 완전이진 트리 형태 일때 높이가 최소화 된다.
예를 들어, 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15의 순서

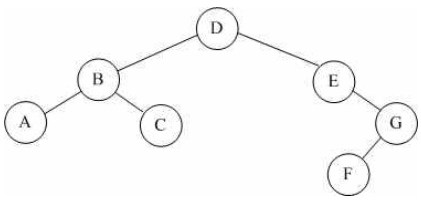
44. 가,나,라,마

45. level 0: 1개, level 1: 2개, level 2: 4개, level 3: 8개 총 15개

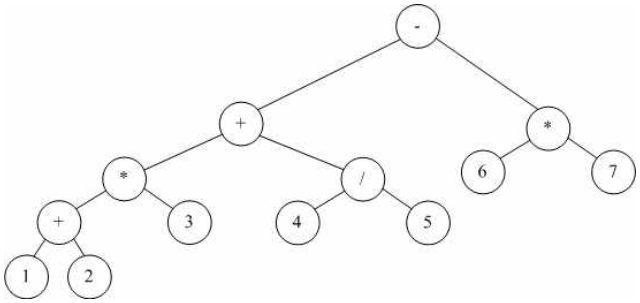
46. 각 레벨별로 1개, 총 4개

47. 최대 개수: $2^{d+1} - 1$ 최소개수: d+1

48.



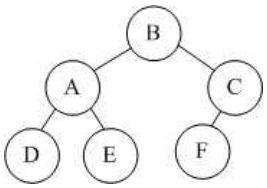
49.



후위표현: 1 2 + 3 * 4 5 / + 6 7 * -

50.
가. 50 20 10 40 80 60 55 57 58 75 90
나. 10 20 40 50 55 57 58 60 75 80 90
다. 10 40 20 58 57 55 75 60 90 80 50
라. 50 20 80 10 40 60 90 55 75 57 58
마. A. 55가 루트로 가고 57이 60의 LChild로 간 모습
B. 40이 루트로 간 모습

51. level 0: 1개, level 1: 2개, level 2: 4개 이면 7개 노드의 포화 이진트리
52.



배열은 BACDEF 순서로 채워져 있음

53.
```
typedef struct {
int Key;
node* Parent;
node* Children[4];
} node;
```

54.
가.
```
void IncrementData(Nptr T)
{ if (T != NULL)
  {    T->Data += 1;
       IncrementData(T->LChild);
       IncrementData(T->RChild);
   }
}
```

나.
```
int CountData70(Nptr T)
{ if (T = = NULL)
```

```
        return 0;
    else
    {     if (T->Data = = 70)
              return (1 + CountData70(T->LChild) + CountData70(T->RChild));
          else
              return (CountData70(T->LChild) + CountData70(T->RChild));
    }
}
```

다.

```
Nptr FindMaxNode(Nptr T)
{ if (T->RChild = = NULL)
        return T;
  else return (FindMaxNode(T->RChild));
}
```

55

```
int CalculateTreeHeight(Nptr T)
{ if (T = = NULL)
        return -1;
  else
        return (1 + Max(CalculateTreeHeight(T->LChild), CalculateTreeHeight(T->RChild));
}
```

56.

```
void SwapSubTree(Nptr& T)
{ if (T != NULL)
  {   Nptr Temp = T->LChild;
      T->LChild = T->RChild;
      T->RChild = Temp;
      SwapSubtree(T->LChild);
      SwapSubtree(T->RChild);
  }
}
```

57. 자신의 중위 후속자를 자신 위치로 복사. 이후 중위후속자를 삭제함.

58. 후위순회 방식. 전위순회는 지워버리면 왼쪽 자식으로 내려갈 수 없음. 중위순회도 지워버리면 오른쪽 자식으로 내려 갈 수 없음.

59. 2i + 2

60.

```
void DeepCopy(Nptr T, Nptr& C)
{ if (T = = NULL)
      C = NULL;
  else
  {   C = New node;
      C.Data = T->Data;
      DeepCopy(T->LChild, C->LChild);
```

```
        DeepCopy(T->RChild,  C->RChild);
    }
}
```

61.

```
void  DescendingOrder  (Nptr  T)
{  if  (T  !=  NULL)
    {    DescendingOrder(T->RChild);
         cout  <<  T->Data;
         DescendingOrder(T->LChild);
    }
}
```

62. 전위순회
63. 리프노드 데이터의 합계
64. 교재참조