6장. 스택

스택, 큐

- 리스트 작업은 시간과 무관하게 정의
- 스택과 큐의 작업은 시간을 기준으로 정의
- 스택은 가장 나중 삽입된 데이터가 먼저 삭제되는 특성의 자료형을 추상화

학습목표

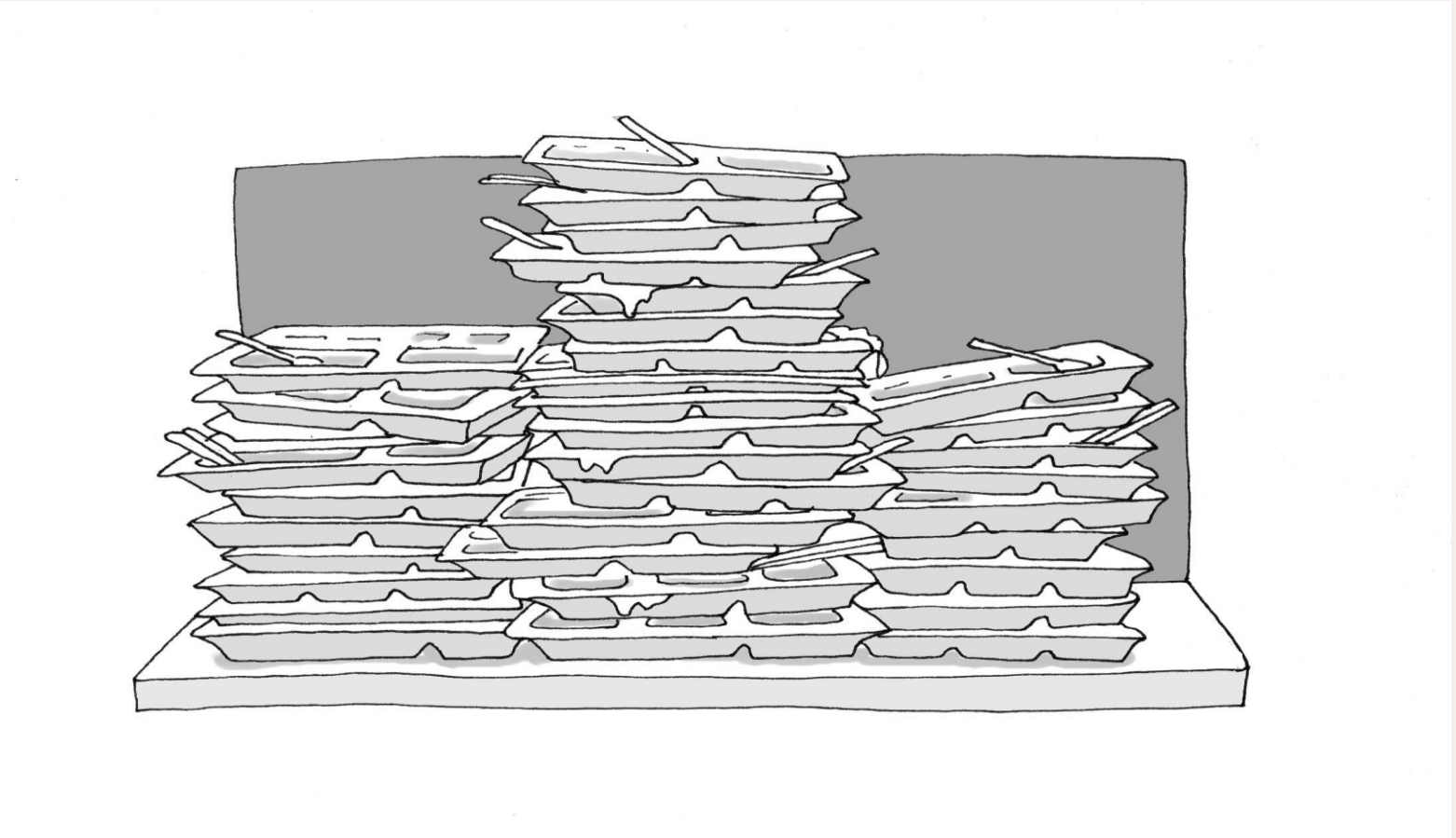
- 추상 자료형 스택의 개념을 이해한다.
- 배열과 연결 리스트로 구현할 때의 차이점을 이해한다.
- 추상 자료형 리스트로 구현하는 방법을 명확히 이해한다.
- 스택의 응용 예, 특히 깊이우선 탐색 방법을 이해한다.
- 스택과 재귀호출의 연관성을 이해한다.

Section 01 스택 개념 – 스택

스택

- 들어온 시간 순으로 데이터를 쌓아갈 때 가장 위에 즉, 가장 최근에 삽입된 위치에 있는 데이터를 삭제하거나 아니면 거기에 이어서 새로운 데이터를 삽입할 수 있도록 하는 추상 자료형
- 후입선출(後入先出)
- **LIFO(Last-In, First-Out: 라이포 또는 리포)**
- **LCFS(Last-Come, First-Served)**
- **cf. 큐: 선입선출(先入先出), FIFO(First-In, First-Out: 파이포 또는 피포)**

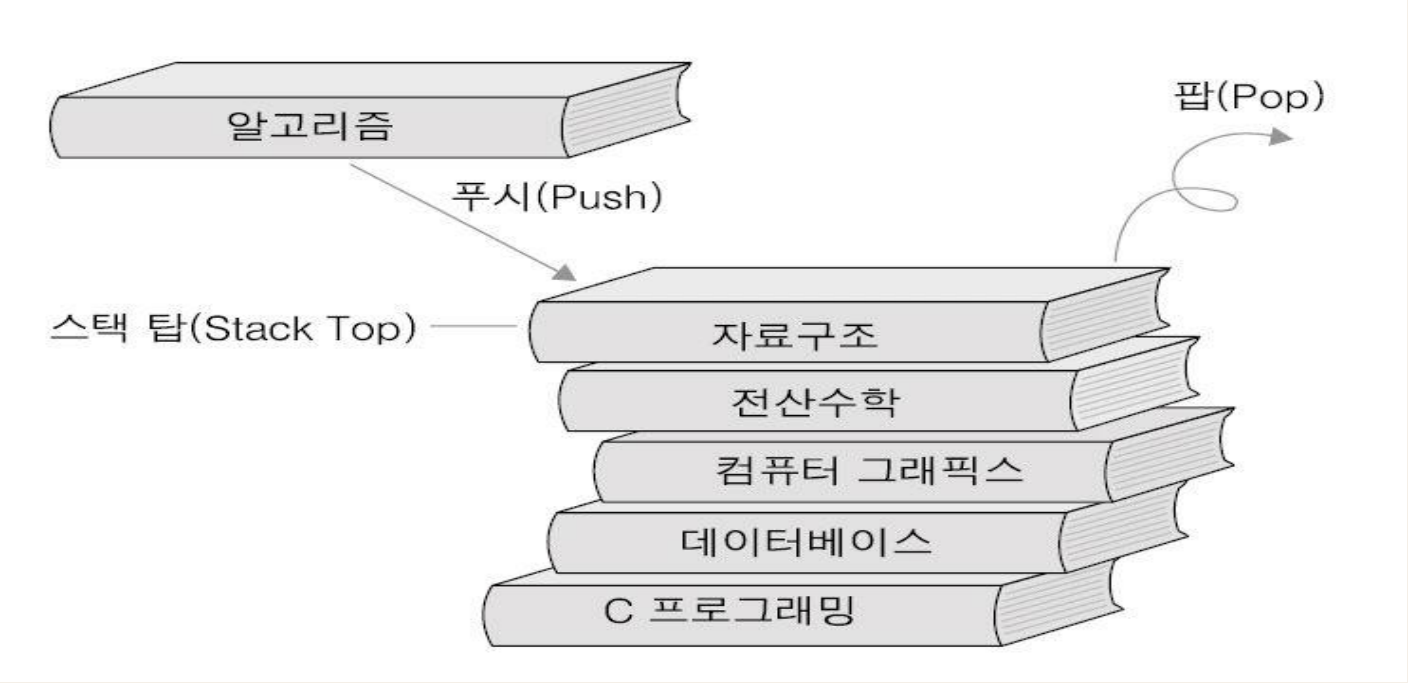
스택 개념



[그림 6-1] 식판 스택

스택 탑, 푸쉬, 팝

- 스택 탑만 접근 가능
- 한쪽 끝에서 삽입, 삭제
  - 삽입 삭제 위치를 스택 탑부근으로 제한 함
  - 구현 자료구조에서 탑이 아닌 다른 곳을 접근할 수 있어도 하지 않기로 약속



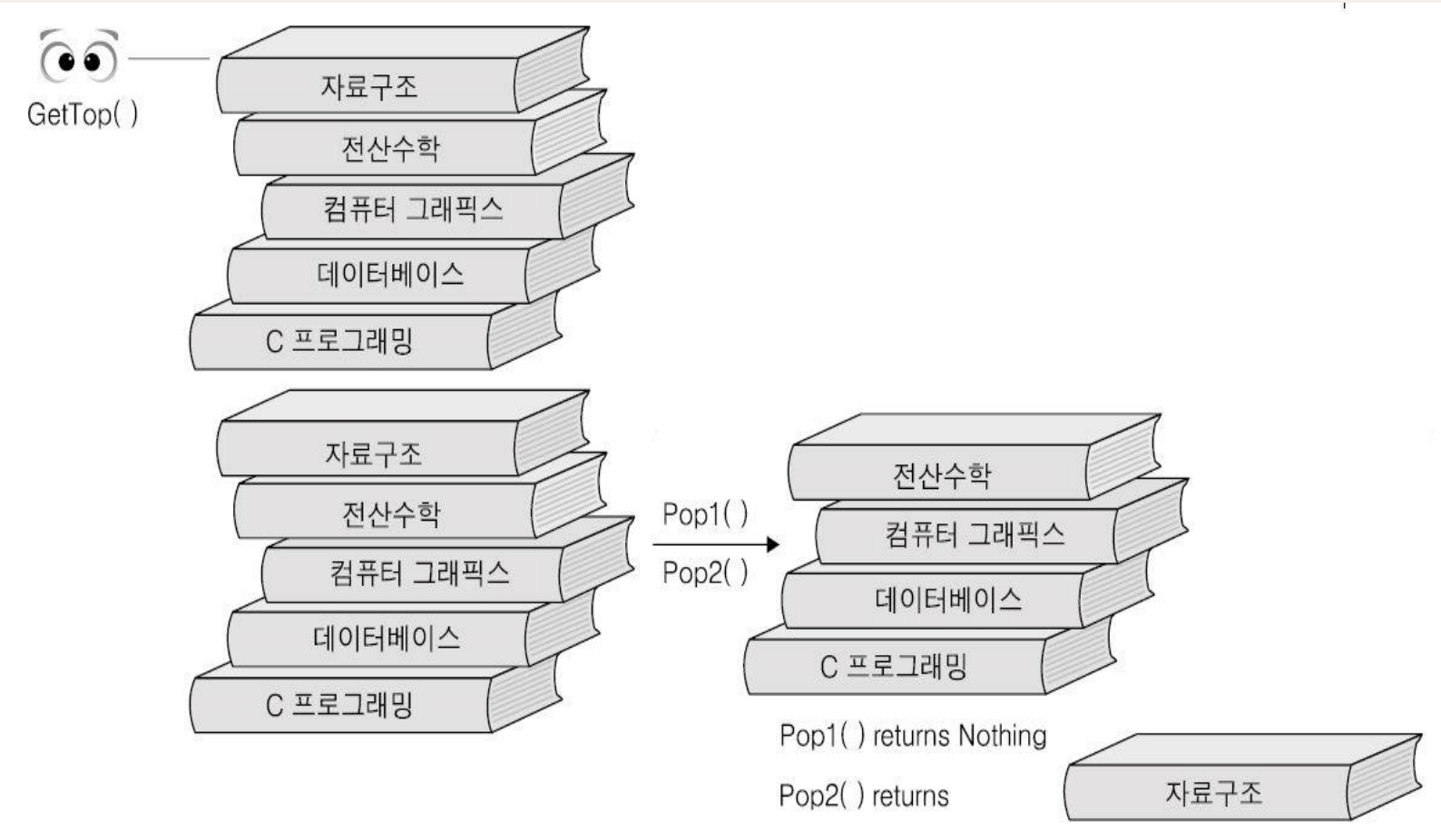
[그림 6-2] 교과서 스택

Section 02 추상 자료형 스택 – 추상 자료형 스택

주요작업

- **Create:** 새로운 스택을 만들기
- **Destroy:** 사용되던 스택을 파기하기
- **Push:** 스택 탑 바로 위에 새로운 데이터를 삽입하기
- **Pop:** 스택 탑 데이터를 삭제하기
- **GetTop:** 스택 탑 데이터를 검색하기
- **IsEmpty:** 현재 스택이 비어있는지를 확인하기
- **IsFull:** 현재 스택이 꽉 차 있는지를 확인하기
- **GetSize:** 현재 스택에 들어가 있는 데이터의 개수를 알려주기

추상 자료형 스택



[그림 6-3] GetTop, Pop1( ), Pop2( )

추상 자료형 스택

- 액시엄
  - **GetTop(Push(S, X)) = X**
  - **Pop(Push(S, X)) = S**
  - **IsEmpty(Create( )) = TRUE**
  - **IsEmpty(Push(S, X)) = FALSE**
  - **GetSize(Push(S, X)) = GetSize(S) + 1**

코드 6-1: StackA.h (C Interface by Array)

```
#define MAX 100                          최대 100개 데이터를 저장
typedef struct
{ int Top;                                    스택 탑의 인덱스를 추적
 int Stack[MAX];                  스택 데이터는 정수형, 최대 100개
} stackType;                      스택 타입은 구조체

void Push(stackType* Sptr, int Item);  스택 데이터를 정수로 가정
int Pop(stackType* Sptr);                 스택 탑의 데이터 값을 리턴 함
void Init(stackType* Sptr);               스택 초기화
bool IsEmpty(stackType* Sptr);            비어 있는지 확인
bool IsFull(stackType* Sptr);             꽉 차 있는지 확인
```

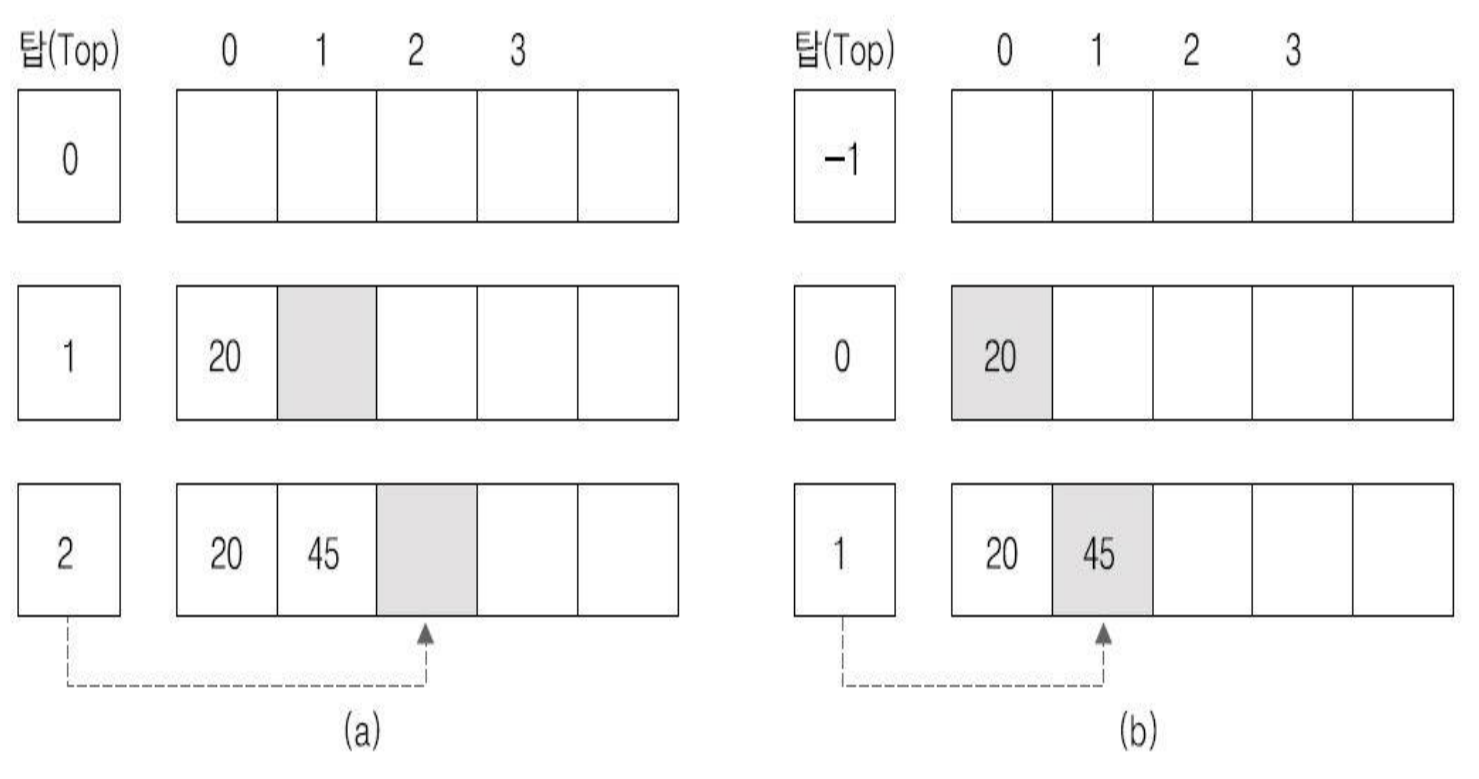
C 배열에 의한 스택

코드 6-2: StackA.c (C Implementation by Array)

```
#include <StackA.h>                          헤더파일을 포함

void Init(stackType* Sptr)                   스택 초기화 함수
{ Sptr->Top = 0;                             탑 인덱스 0으로 빈 스택을 표시
}

boolean IsEmpty(stackType* Sptr)             비어 있는지 확인하는 함수
{ return (Sptr->Top = = 0);                  탑 인덱스 0이면 TRUE
}

void Push(stackType* Sptr, int Item)         스택의 삽입함수
{ Sptr->Stack[Top++] = Item;                 현재 탑에 삽입 후 탑 인덱스를 증가
}                                            IsFull인지를 미리 확인해야 함

int Pop(stackType* Sptr)                     스택의 삭제함수
{ return Sptr->Stack[--Top];                 탑 인덱스 바로 아래 데이터를 리턴
}                                            IsEmpty인지를 미리 확인해야 함
```

C 배열에 의한 스택

- 두가지 탑 인덱스
  - 초기화 방법이 다름
  - 푸쉬, 팝 인덱스 계산이 달라짐



[그림 6-4] 두 가지 탑 인덱스

C 연결 리스트에 의한 스택

- 코드 6-3: StackP.h (C Interface by Linked List)

```
typedef struct
{ int Data;                    스택 데이터를 정수 형으로 가정
  node* Next;                  다음 노드를 가리키는 포인터 변수
} node;                        노드는 구조체 타입
typedef node* Nptr;            Nptr 타입이 가리키는 것은 노드 타입

void Push(Nptr Top, int Item); 스택 데이터를 정수로 가정
int Pop(Nptr Top);                    스택 탑의 데이터 값을 리턴 함
void Init(Nptr Top);                  스택 초기화
bool IsEmpty(Nptr Top);               비어 있는지 확인
void FreeList(Nptr );                 연결 리스트 공간을 반납
```

C 연결 리스트에 의한 스택

- 코드 6-4: StackP.c (C Implementation by Linked List)

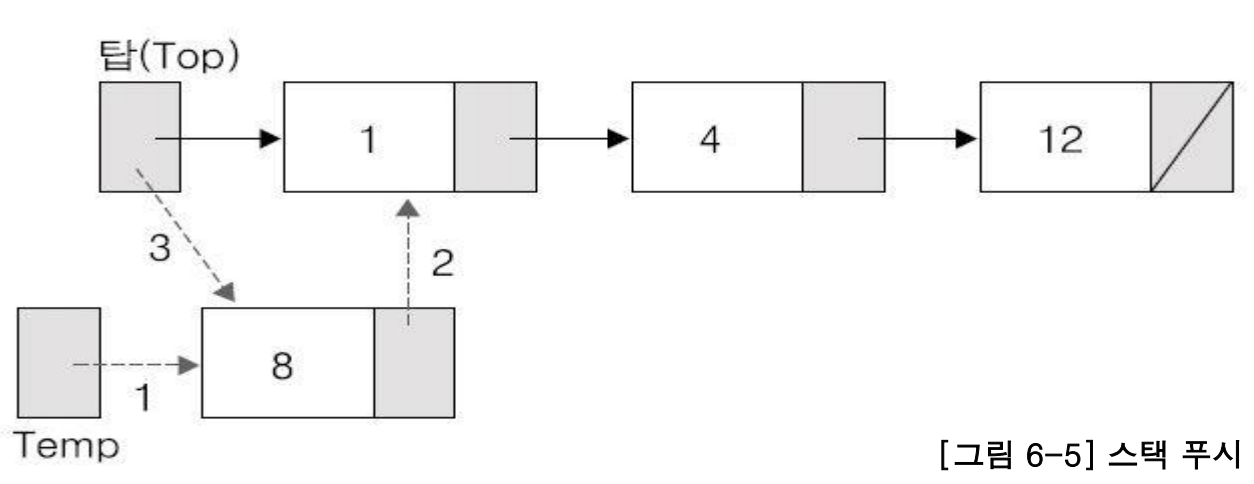
```
#include <StackP.h>                헤더파일을 포함

void Init(Nptr Top);               초기화 함수
{ Top = NULL;                       탑 포인터를 널로
}

bool IsEmpty(Nptr Top);            빈 스택인지 확인하는 함수
{ return (Top = = NULL);           탑 포인터가 널이면 TRUE
}
```

C 연결 리스트에 의한 스택

- **void Push(Nptr Top, int Item)** 스택의 삽입함수

```
{ Nptr Temp = (Nptr)malloc(sizeof(node));   새로운 노드 공간을 확보하고
  Temp->Item = Item;                        넘어온 데이터 복사
  Temp->Next = Top;                         새 노드가 현재상태의 첫 노드를 가리키게
  Top = Temp;                               탑이 새로운 노드를 가리키게
}
```

- 삽입, 삭제
  - 반드시 첫 노드
  - 이유는 무엇인가



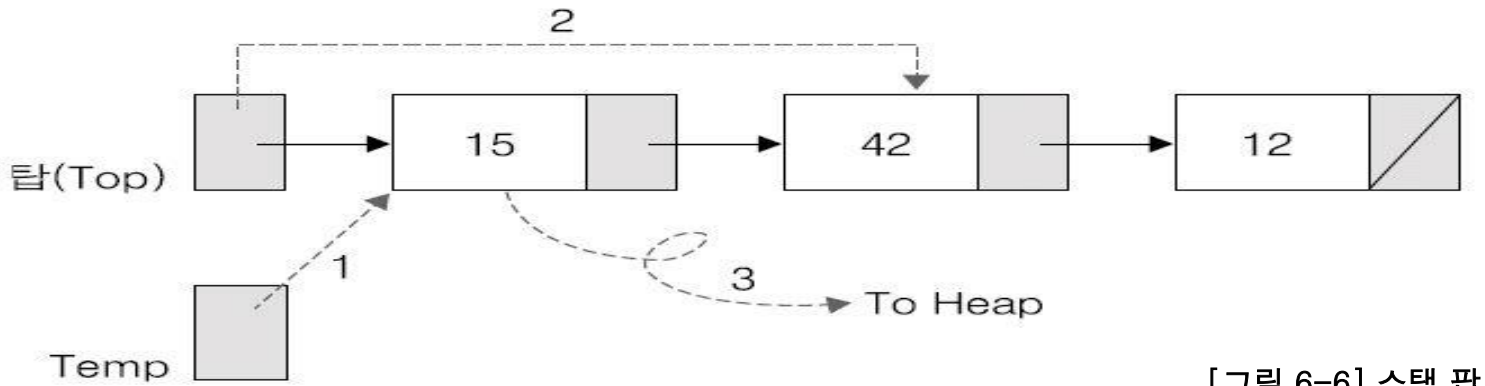
[그림 6-5] 스택 푸시

C 연결 리스트에 의한 스택

```
int Pop(Nptr Top)
{ if (Top = = NULL)
            printf("Empty Stack");
  else
  {   Nptr Temp = Top;
     int Item = Temp->Data;
            Top = Top -> Next;
            free Temp;
            return Item;
  }
}
```

스택의 삭제함수
빈 리스트이면
오류처리

탑 포인터를 백업
이전 탑 데이터 백업
탑이 다음 노드를 가리키게
이전 탑 노드 공간반납
이전 탑 데이터 리턴



[그림 6-6] 스택 팝

C 연결 리스트에 의한 스택

```
void FreeList(Nptr Top )
  { Nptr Temp = Top;
    while (Temp != NULL)
    {    Top = Top->Next;
         free Temp;
        Temp = Top;
     }
  }
```

연결 리스트 공간반납 함수

리스트가 완전히 빌 때까지

탑 포인터 전진

이전 탑 노드 공간반납

탑 포인터 백업

코드 6-5: StackA.h (C++ Interface by Array)

```
const int MAX = 100;
class stackClass
{ public:
                    stackClass( );                  생성자 함수
        stackClass(const stackClass& S);            복사 생성자 함수
        ~stackClass( );                             소멸자 함수
        void Push(int Item);                        Item 값을 스택에 삽입
        int Pop( );                                 스택 탑의 데이터 값을 리턴 함
        boolean IsEmpty( );                         비어 있는지 확인
        boolean IsFull( );                          꽉 차 있는지 확인
private:
        int Top;                                    스택 탑의 인덱스를 추적
        int Stack[MAX];                             정수형 스택 데이터 최대 100개
}
```

C ++ 배열에 의한 스택

코드 6-6: StackA.cpp (C++ Implementation by Array)

```
#include <StackA.h>
stackClass::stackClass( )                          생성자 함수
{  Top = 0;                                        탑 인덱스 0으로 초기화
}

void stackClass::stackClass(const stackClass& S)   복사생성자
{ Top = S.Top;                                     탑 인덱스를 복사
  for (int Index = 0; Index <= S.Top; ++ Index)    인덱스 0부터 S.Top까지
                  Stack[Index] = S.Stack[Index];   배열 요소 복사
}

stackClass::~stackClass( )                         소멸자 함수
{                                                  실행할 일 없음
}

boolean stackClass::IsEmpty( )                     빈 스택인지 확인하는 함수
{ return boolean (Top = = 0);                      탑 인덱스 0 이면 TRUE
}
```

C ++ 연결 리스트에 의한 스택

코드 6-7: StackP.h (C++ Interface by Linked List)

```
typedef struct
{ int Data;                    스택 데이터를 정수 형으로 가정
  node* Next;                  다음 노드를 가리키는 포인터 변수
} node;                        노드는 구조체 타입
typedef node* Nptr;            Nptr 타입이 가리키는 것은 노드 타입

class stackClass
{ public:
      stackClass( );                              생성자 함수
      stackClass(const stackClass& S);            복사 생성자 함수
      ~stackClass( );                             소멸자 함수
      void Push(int Item);                        Item 값을 스택에 삽입
      int Pop( );                                 스택 탑의 데이터 값을 리턴 함
      boolean IsEmpty( );                         비어 있는지 확인
      boolean IsFull( );                          꽉 차 있는지 확인
private:
      Nptr Top;                                   첫 노드를 가리키는 포인터
}
```

C ++ 연결 리스트에 의한 스택

코드 6-8: StackP.cpp (C++ Implementation by Linked List)

```
#include <StackP.h>
stackClass::stackClass( )                生성자 함수
{ Top = NULL;                            탑 포인터 값을 널로 세팅
}

stackClass::~stackClass( )               소멸자 함수
{ int Temp;
  while (!IsEmpty( ))                    스택이 완전히 빌 때까지
      Temp = Pop( );                     계속해서 팝
}

boolean stackClass::IsEmpty( )           비어있는지 확인하는 함수
{ return boolean(Top = = NULL);          탑이 널이면 TRUE
}
```

C ++ 연결 리스트에 의한 스택

```
void stackClass::Push(int Item)                스택의 삽입함수
{ Nptr NewTop = new Nptr;                      새로운 노드 공간을 확보하고
  NewTop->Data = Item;                         넘어온 데이터 복사
  NewTop->Next = Top;                  새 노드가 현재상태의 첫 노드를 가리키게
  Top = NewTop;                        탑이 새로운 노드를 가리키게
}
```

```
int stackClass::Pop( )                         스택의 삭제함수
{ if (IsEmpty( ))                              빈 스택이라면 오류처리
      cout << "Deletion on Empty Stack";
  else                                         빈 스택이 아니라면
  {   Nptr Temp = Top;                         탑 포인터를 백업
      int Item = Temp->Data;                   탑 노드의 데이터를 백업
      Top = Top->Next;                         탑이 다음 노드를 가리키게
      delete Temp;                             이전 탑 노드 공간반납
      return Item;                             이전 탑 데이터 리턴
  }
}
```

코드 6-9: StackL.h (C++ Interface by ADT LIST)

```
#include <ListP.h>
class stackClass
{ public:
      코드 6-7(또는 코드 6-6)과 동일
  private:
      listClass L;
};
```

리스트 객체

- 스택 클래스 객체 S는 리스트 클래스 객체 L을 가짐
- 리스트 클래스 멤버함수 사용을 위해 리스트 클래스 헤더파일 <ListP.h> 또는 <ListA.h>를 포함

추상자료형 리스트에 의한 스택구현

코드 6-10: StackL.cpp (C++ Implementation by ADT LIST)

```
#include <StackL.h>
stackClass::stackClass( )                          생성자 함수
{
}
stackClass::stackClass(const stackClass& S)        복사 생성자 함수
{ L = S.L;
}
```

생성자 함수

- 스택 클래스 객체 S의 L 필드 즉, S.L 필드는 객체 S가 선언되는 순간에 생성
- L은 리스트 클래스 객체이기 때문에 리스트 클래스의 생성자를 통해 초기화

복사 생성자

- stackClass A = B; 에 의해 스택 클래스의 복사 생성자가 불려옴
- 복사 생성자 내부에 A.L = B.L;로 선언되어 있으므로 결과적으로 리스트 클래스의 복사 생성자를 부르게 됨

추상자료형 리스트에 의한 스택구현

```
boolean stackClass::IsEmpty( )
{ return boolean(L.IsEmpty( ));
}
```
빈 스택인지 확인하는 함수

```
void stackClass::Push(int NewItem)
{ L.Insert(1, Item);
}
```
푸쉬 함수

```
void stackClass::Pop( )
{ L.Delete(1);
}
```
팝 함수

```
void stackClass:GetTop(int& Item)
{L.Retrieve(1, Item);
}
```
스택 탑을 읽는 함수

- 추상 자료형 리스트위치기반 리스트
  - 리스트의 처음 위치를 스택 탑으로 간주
  - 스택의 푸쉬 작업은 리스트의 처음 위치에 데이터를 삽입하는 것에 해당
  - 스택의 팝 작업은 리스트의 첫 데이터를 삭제하는 것에 해당

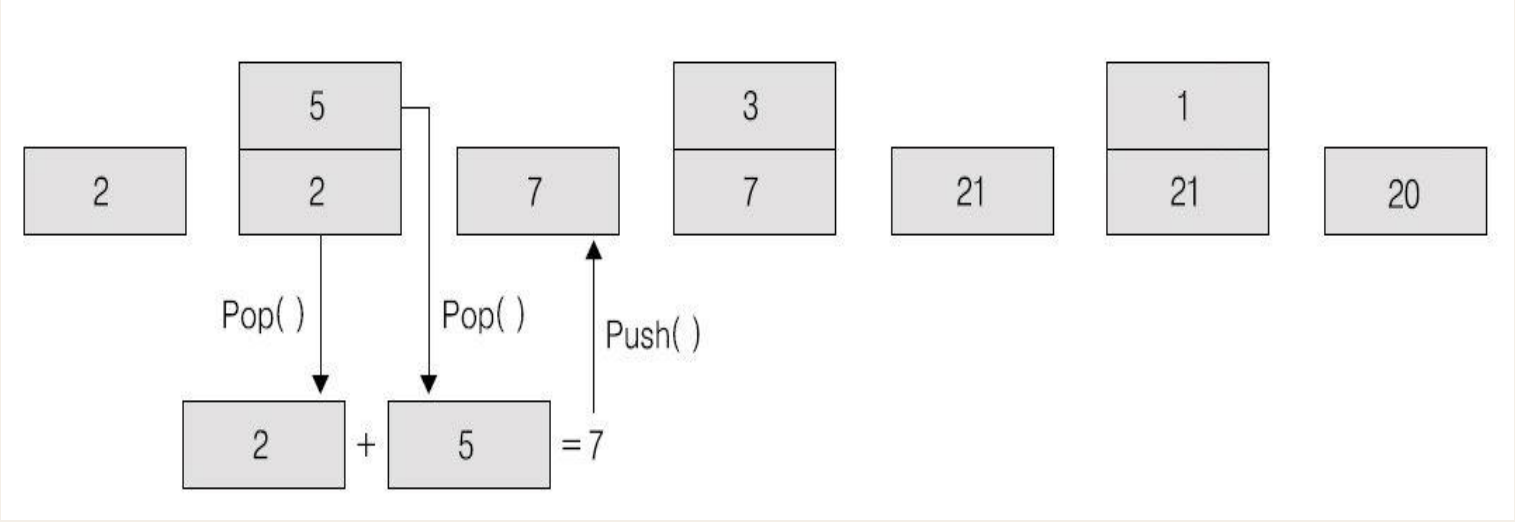
Section 06 스택 응용 예 – 스택 응용 예

- 백 스페이스 키
  - **KY<bs>OTW<bs><bs>REA** 순으로 입력
  - 최종 결과는?
  - 백 스페이스는 어느 위치에 작용하는가

스택 응용 예

- 연산자 표현
  - 중위표현(中位, In-Fix Expression): 5 + 7
  - 연산자(Operator)가 피연산자(Operand)의 가운데 있는 표현
  - 후위표현(後位, Post-Fix Expression) : 5 7 +
  - 연산자가 피 연산자 맨 뒤로 가는 표현. RPN(Reverse Polish Notation)

- 2 5 + 3 * 1 - 의 연산



[그림 6-7] 2 5 + 3 * 1 - 의 연산

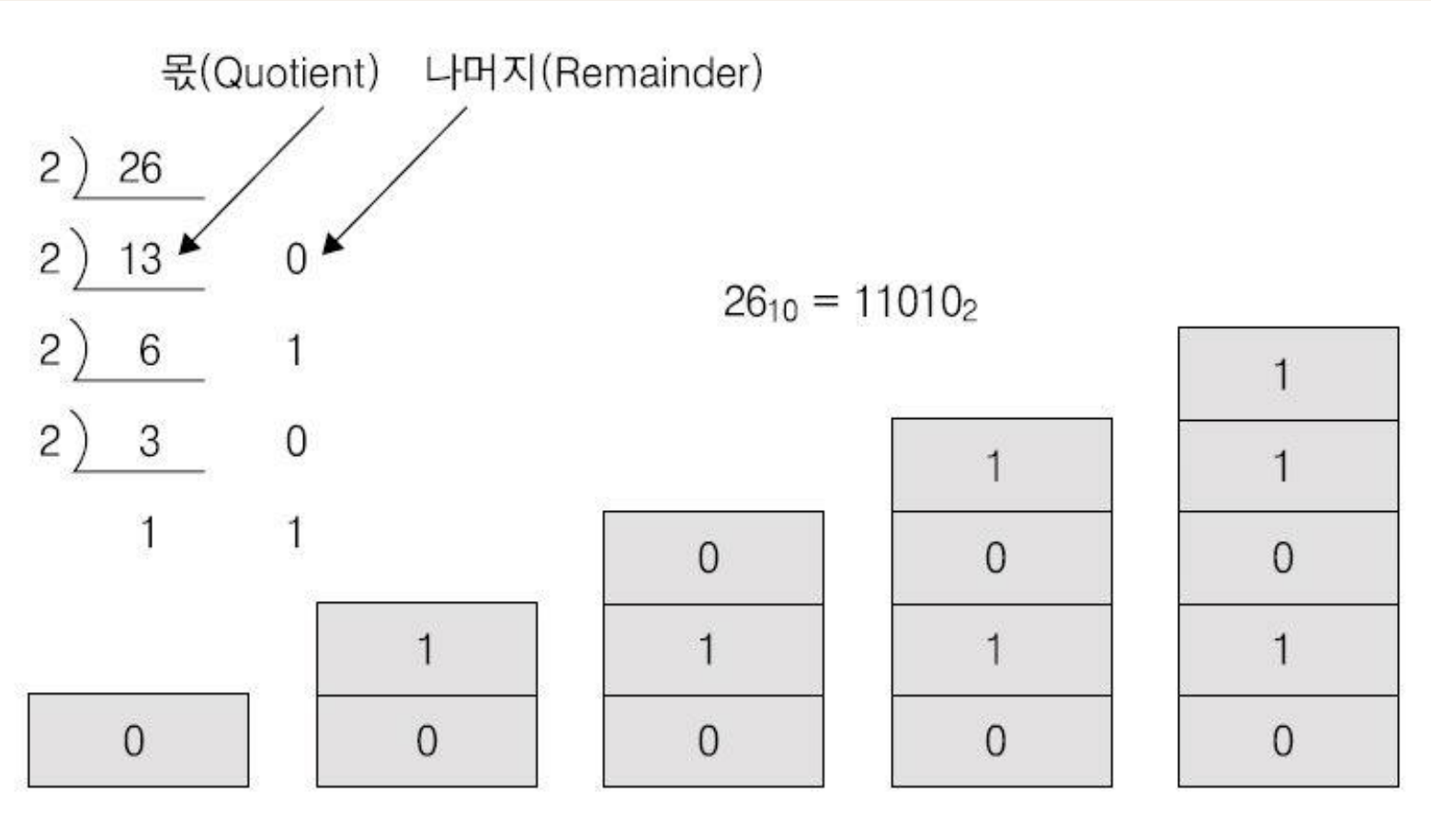
스택 응용 예

- 코드 6-11: 후위표현의 연산

```
Read Symbol Ch                          첫 문자 읽기
While Not End of Expression             수식의 끝을 만날 때까지
{    If Ch Is an Operand                읽은 문자가 피연산자이면
        Push Ch                         스택에 푸쉬
     Else                               읽은 문자가 연산자이면
     {   Pop Operand2                   두 번째 피연산자 팝
         Pop Operand1                   첫 번째 피연산자 팝
         Result = Operand1 Ch Operand2  결과계산
         Push Result                    결과를 스택에 푸쉬
     }
     Read Symbol Ch                     새로운 문자 읽기
}
Pop Stack and Print Result              결과를 출력
```

스택 응용 예

- 진법의 변환(예: 10진수에서 2진수로)



[그림 6-8] 10진수의 2진 변환

스택 응용 예

- 스택과 재귀호출 비교(문자열 뒤집기)
  - 주어진 문자열을 들어오는 대로 스택에 푸쉬 하였다고 가정
  - 프로그래머가 만든 스택: 사용자 스택
  - 재귀호출에 의한 스택: 시스템 스택
  - 사용자 스택이 일반적으로 더욱 유리

스택	재귀호출

while (! StackIsEmpty)
{ NewCharacter = Pop( )
Print NewCharater
}
	void Reverse(char S[ ], int First, int Last)
	{ if (First > Last)
	return;
	else
	{ Reverse(S, Fist+1, Last);
	printf("%c", S[First]);
	}
	}

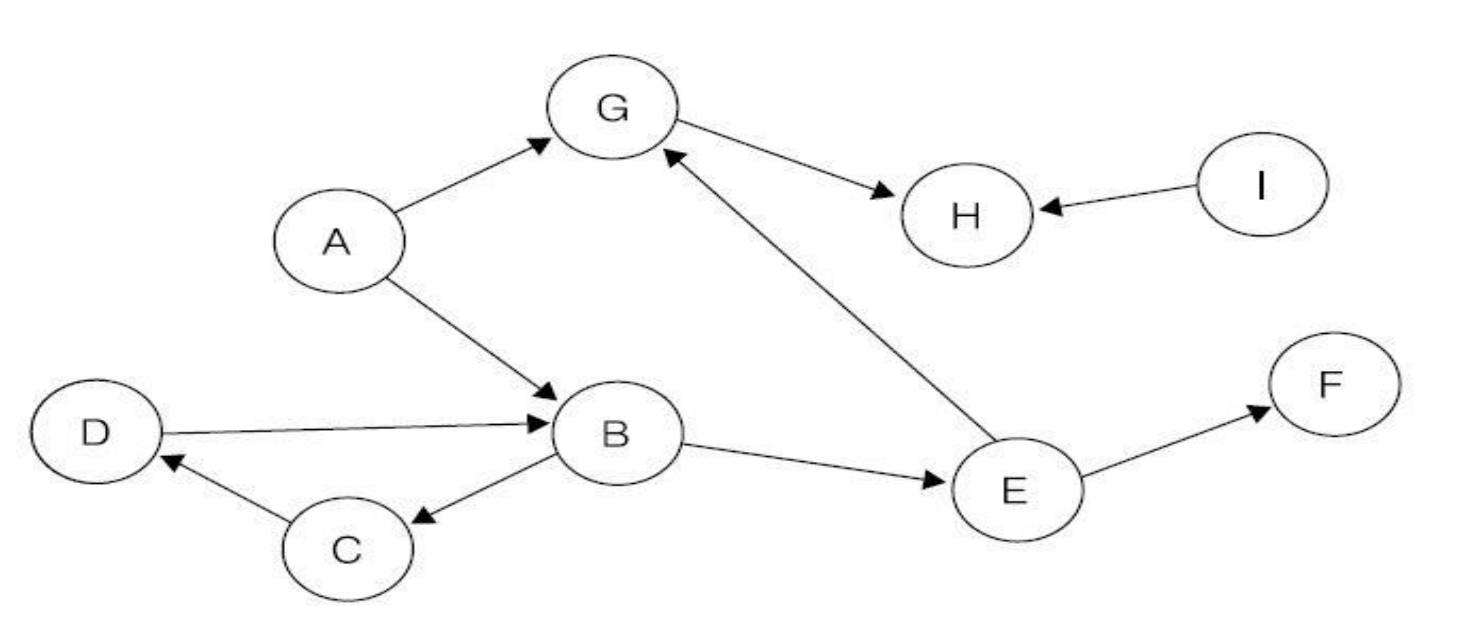
[표 6-1] 스택과 재귀호출의 비교

스택 응용 예

- 코드 6-12 괄호의 매칭

```
InitializeStack( );                                  스택 초기화
Matched = TRUE;                                      매칭 상태를 표시하는 변수 초기화
while (Index < String.Length && Matched)             소스코드 끝까지
{   Character = String[Index++];                     새로운 문자를 읽어서
    if (Character is '{')                            여는 중괄호이면
        Push('{');                                   스택에 푸쉬
    else if (Character is '}')                       닫는 중괄호이면
    {   if (Not StackIsEmpty)                        만약 스택이 비어있지 않다면
            Pop( )                                   팝에 의해 여는 괄호 제거
        else                                         스택이 비어 있다면
        {   Matched = FALSE;                         매칭 되는 여는 괄호가 없었음
            Break;                                   루프를 빠져나감
        }
    }                                                여타문자는 무시함
}
return Matched;                                      매칭결과를 리턴
```

- 노드 A에서 출발해서 노드 F로 가는 경로가 존재하는가
  - 목적지까지 제대로 간다.
    - **A-B-E-F**
  - 어떤 도시로 갔는데 거기서 더 나갈 곳이 없는 막다른 곳이 된다.
    - **A-G-H**
  - 어떤 길을 따라서 계속 원을 그리며 돈다.
    - **A-B-C-D-B-C-D**



[그림 6-9] 탐색을 위한 그래프

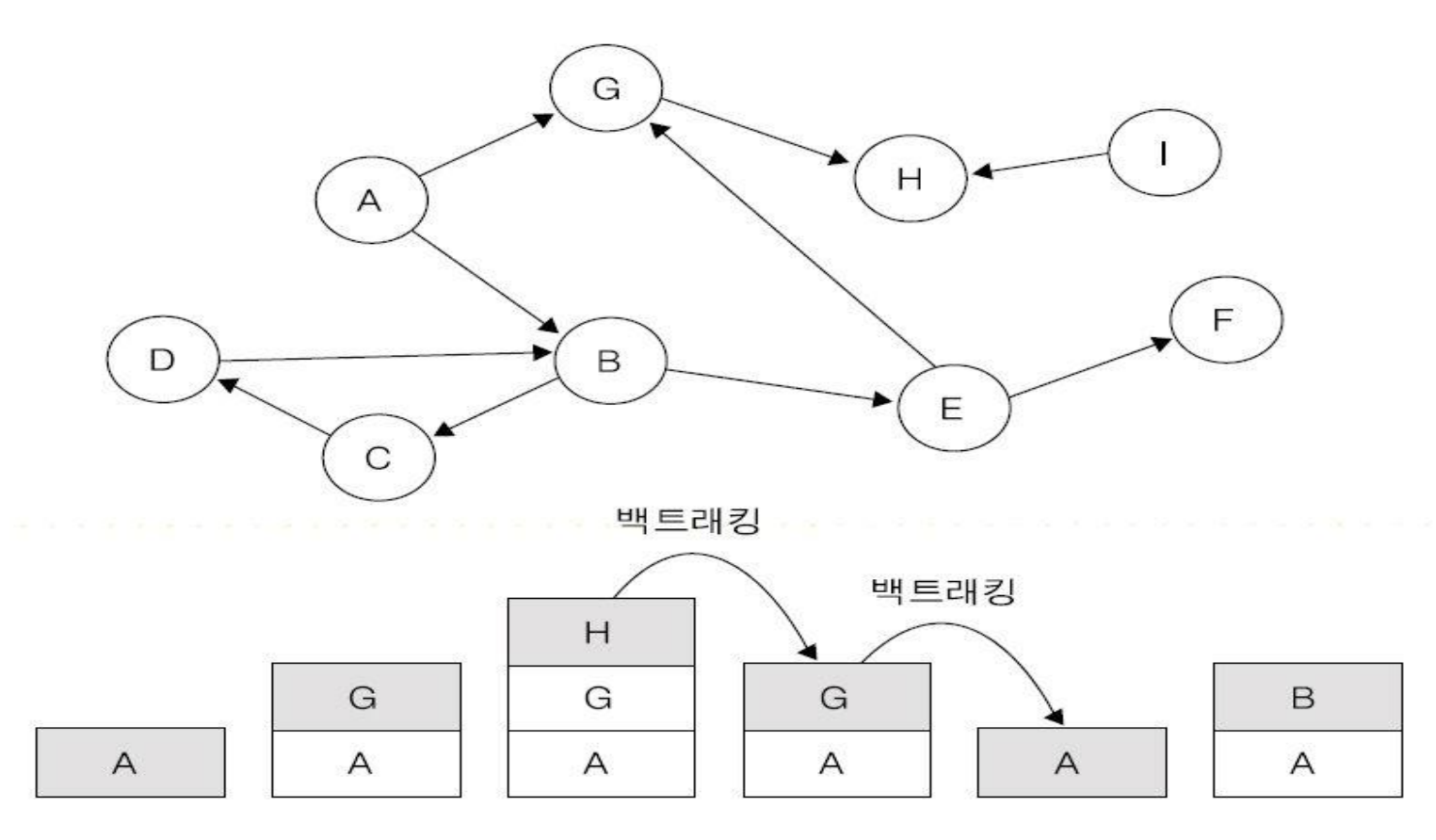
깊이우선 탐색

- 어떤 도시로 갔는데 거기서 더 나갈 곳이 없는 막다른 곳이 된다.
  - 규칙: “막힌 도시라면 다시 바로 그 이전 도시로 되돌아온다.”
  - 되돌아오는 행위를 백 트래킹(Backtracking)이라 함 (한수 물러주기)
  - 되돌아 간 곳에서 막힌 도시 아닌 다른 곳을 시도
  - 선택된 도시를 계속적으로 스택에 푸쉬하면
  - 스택의 내용은 A로부터 출발해서 지금까지 거쳐 온 일련의 도시
  - 백 트래킹은 스택의 팝. 직전에 거쳐온 도시로 되돌아가는 행위

- 어떤 길을 따라서 계속 원을 그리며 돈다.
  - 규칙: “한번 가본 도시는 다시 안 간다.” (재방문 회피)

깊이우선 탐색

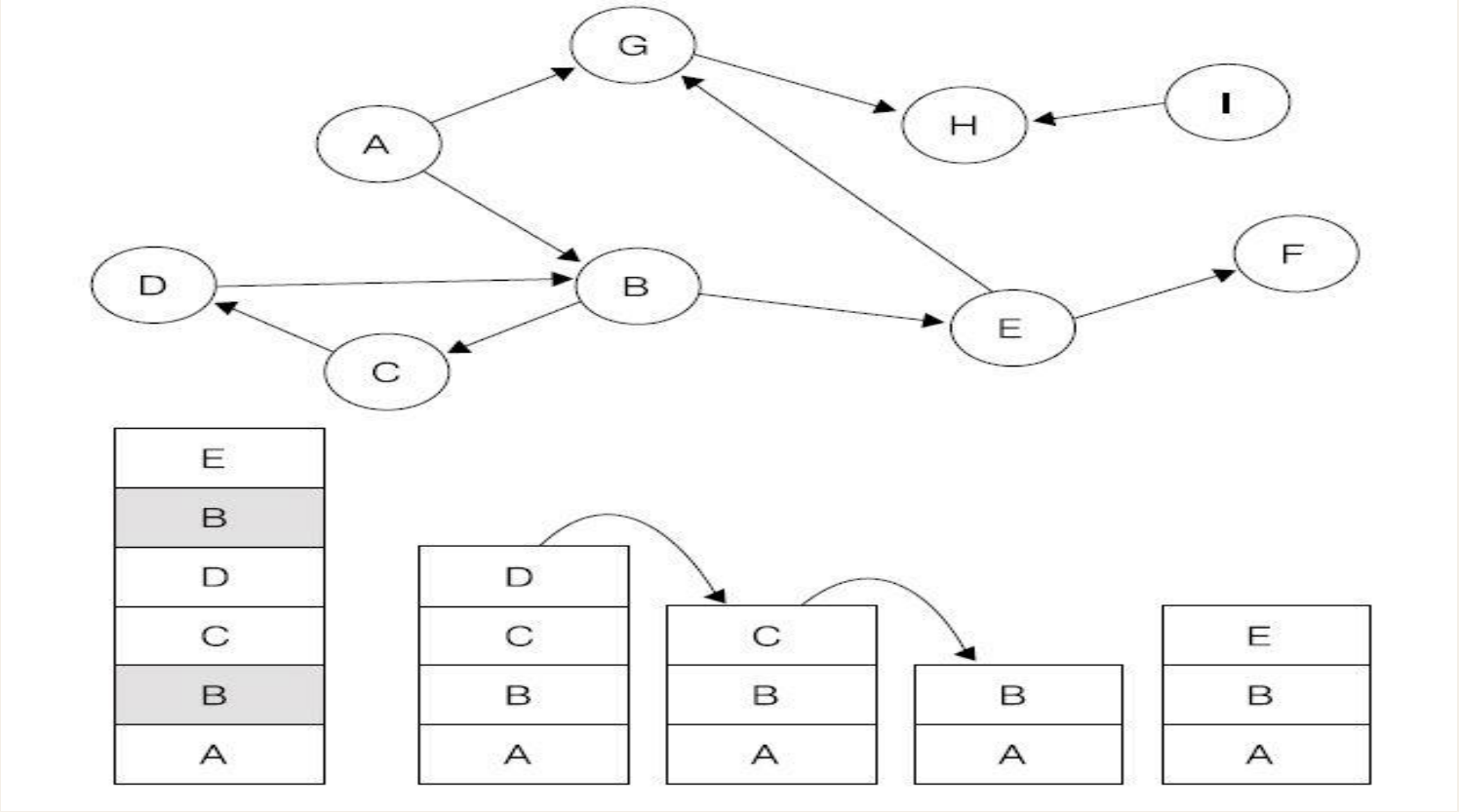
- A-G-H에서 백트래킹
- 다시 B를 시도



[그림 6-10] 백 트래킹

깊이우선 탐색

- 값 재방문 회피 I
  - 스택 내용이 ABCDBE 일 필요가 없음
  - 중간의 BCD를 생략하고 ABE로 갈 수 있음

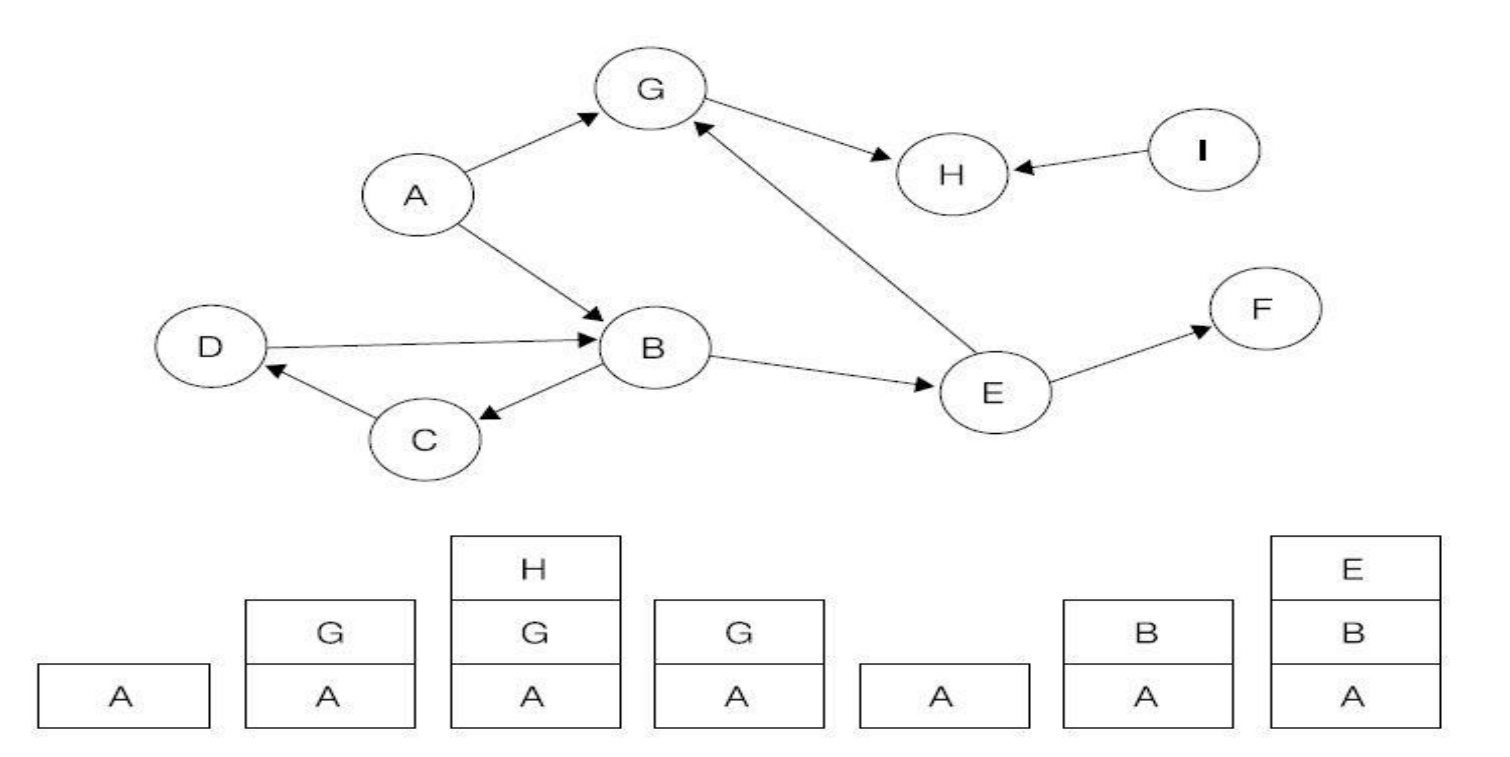


[그림 6-11] 재방문 회피(1)

깊이우선 탐색

재방문 회피 II

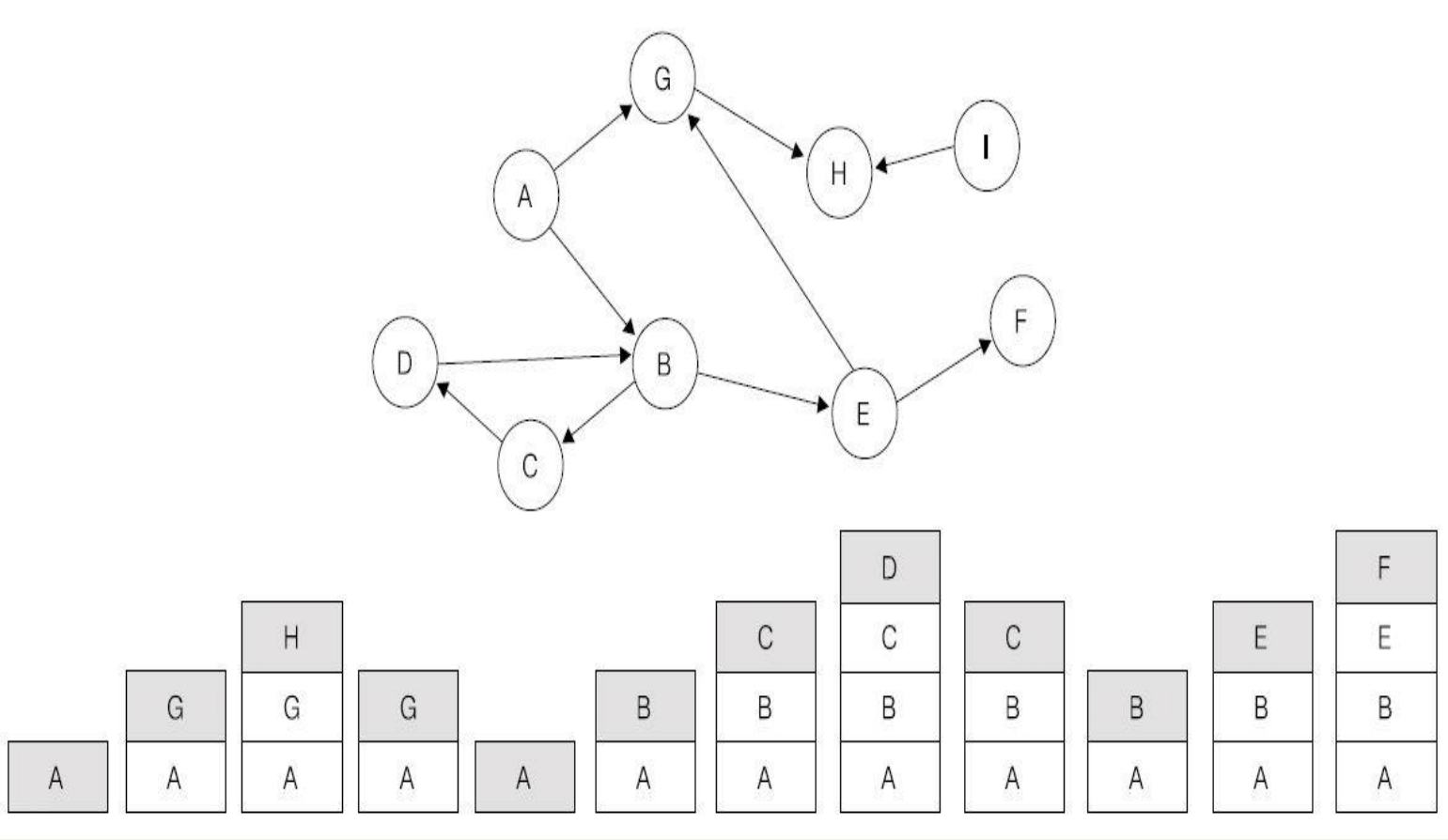
- 현재 상태에서 이미 가본 G를 방문할 필요가 있는가.
- 없다. 만약 G에서 F까지 가는 길이 있었다면 이전에 G로 갔을 때 가 봤을 것이다. 그러한 길이 없어서 백트래킹 한 것이다.



[그림 6-12] 재방문 회피(2)

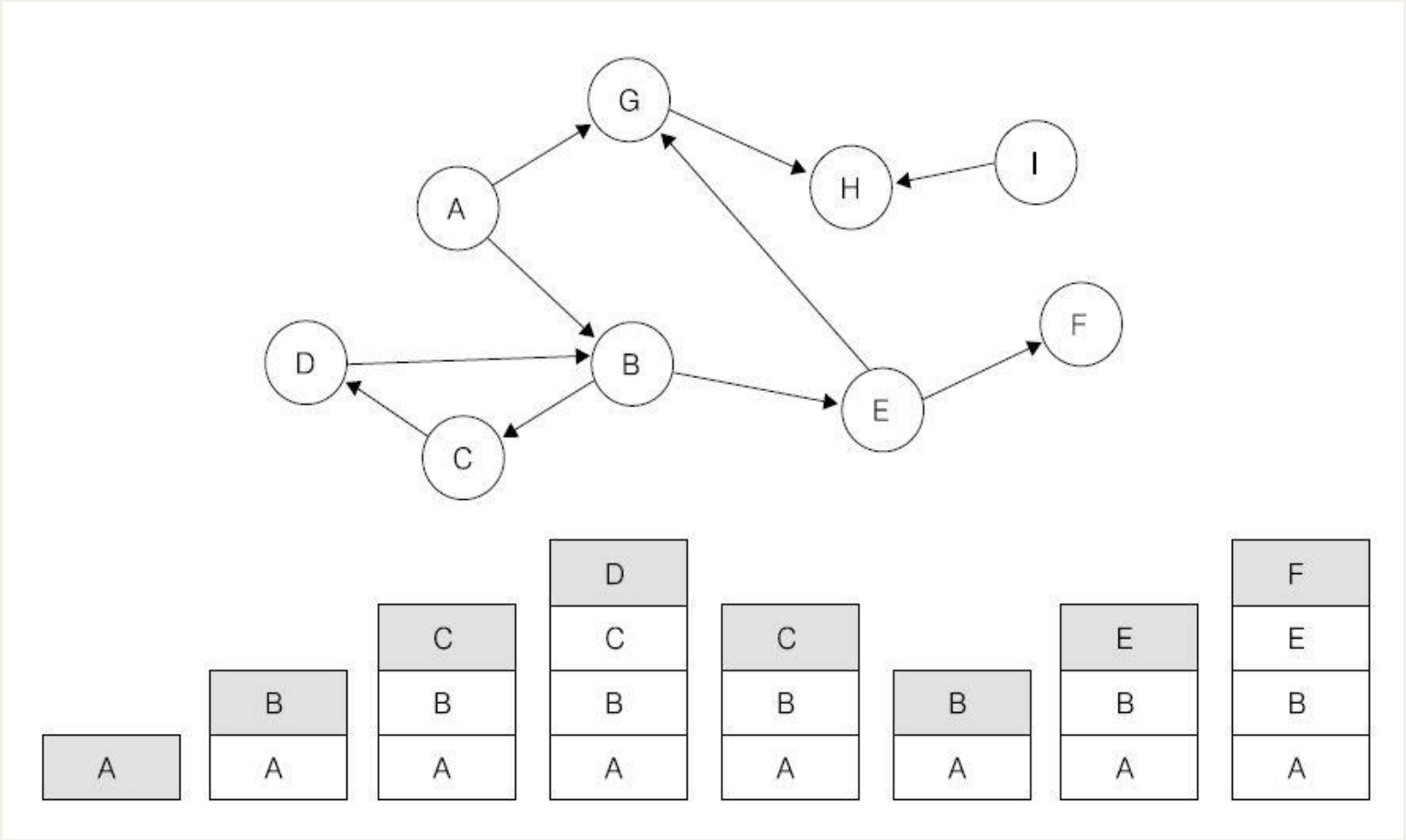
깊이우선 탐색

깊이우선 탐색순서 I



[그림 6-13] 깊이우선 탐색 순서(1)

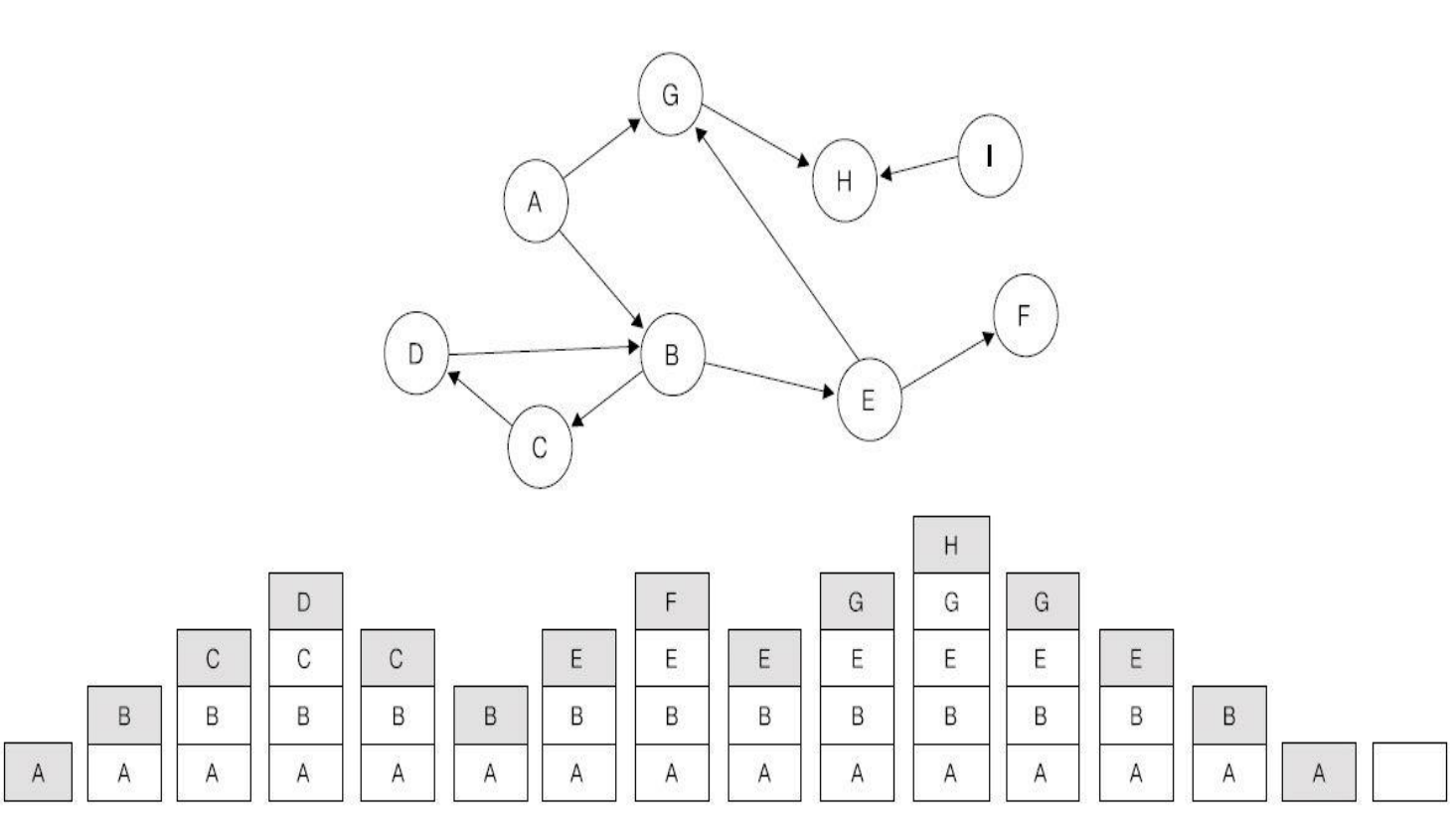
깊이우선 탐색순서 II



[그림 6-14] 깊이우선 탐색순서(2)

깊이우선 탐색

- A에서 K로 가는 길이 있는가?
  - 최종적으로 백트랙에 의해 스택이 비어버림.
  - 그런 경로가 없음을 의미함.



[그림 6-15] 깊이우선 탐색순서(3)

깊이우선 탐색

코드 6-13: 깊이우선 탐색

```
DepthFirstSearch(Origin, Destination)
{  S.Create( );                                  새로운 스택을 만들기
   Mark All Nodes as Unvisited;                  일단 모든 도시를 안 가본 도시로 표시
   S.Push(Origin);                               출발지를 푸쉬
   Mark Origin as Visited;                       푸쉬 된 도시는 가본 것으로 표시
   While (!S.IsEmpty( ) && Destination != S.GetTop( ))
   {  if (All Adjacent Cities Are Visited)
                                                 스택 탑의 인접도시가 모두 가 본 도시이면
         S.Pop( );                               이전 도시로 되돌아 감
      else
      {    Select a New City C;
         S.Push(C);                              새로운 도시를 선택해서 푸쉬
         Mark C as Visited;                      그 도시를 가본 도시로 표시
      }
   }
   if
      (S.IsEmpty( )) return "No";                스택이 비어서 빠져나오면
   else                                          가는 길 없다고 대답
      return "YES";                              현재의 스택 탑이 목적지와 일치하면
}                                                가는 길 있다고 대답
```

```
DepthFirstSearch(Origin, Destination)
{   if (Origin = = Destination)              출발지와 목적지가 같으면
       return “YES";                         가는 경로가 있다고 대답
    else
       for (Each Unvisited Cities C Adjacent to Origin) 안 가본 인접도시에
                                                        대해
          DepthFirstSearch(C, Destination);      거기서부터 목적지까지 재귀
}
```

- 유사성
  - 출발지에서 목적지까지 갈 수 있느냐의 문제는 출발지에서 한 걸음 나아간 도시에서 목적지까지 갈 수 있느냐의 문제와 동일
  - 새로운 탐색을 위해 출발하려는 도시가 이미 목적지와 같다면 목적지까지 가는 경로가 이미 존재(베이스 케이스)
  - 루프가 다 돌아서 더 이상 안 가본 도시가 없으면 이전의 호출함수로 리턴 된다. 이 리턴 하는 행위가 바로 스택을 사용한 탐색에서의 백 트래킹

Thank you