# データ構造とプログラミング技法（第２回）

## ―線形構造―

# 線形構造

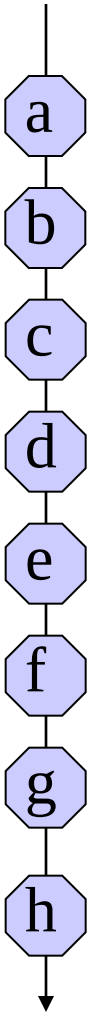
用語：

レコード：ひとまとまりのデータ（構造体）

- 線形リスト：n≧0個のレコードの1次元並び
  - 順配置：　　表
  - リンク配置：　連鎖リスト

# 順配置された線形リスト：表

論理構造



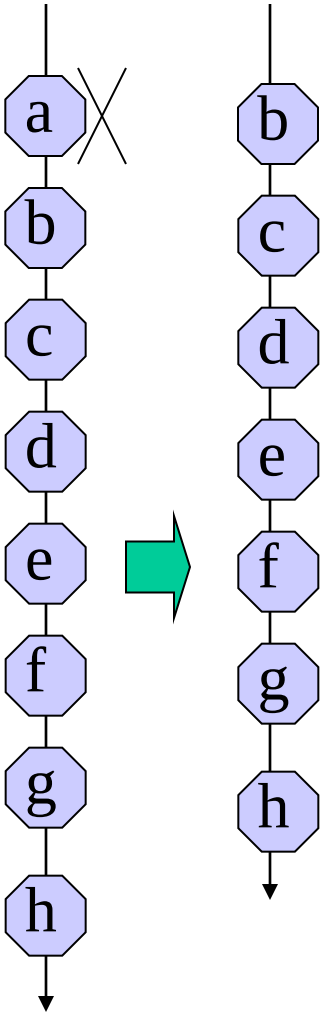

物理構造

要素の「位置」の順序関係を、アドレスの値の順序関係で表現する方法。

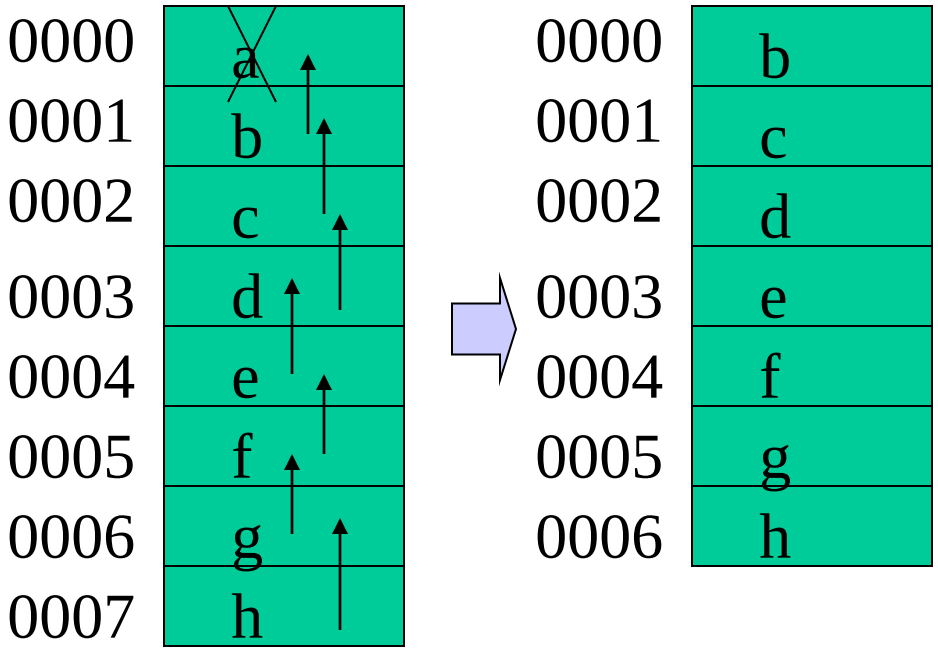
| | |
|---|---|
| 0000 | a |
| 0001 | b |
| 0002 | c |
| 0003 | d |
| 0004 | e |
| 0005 | f |
| 0006 | g |
| 0007 | h |

# 表に対する操作：要素の挿入・削除

論理構造



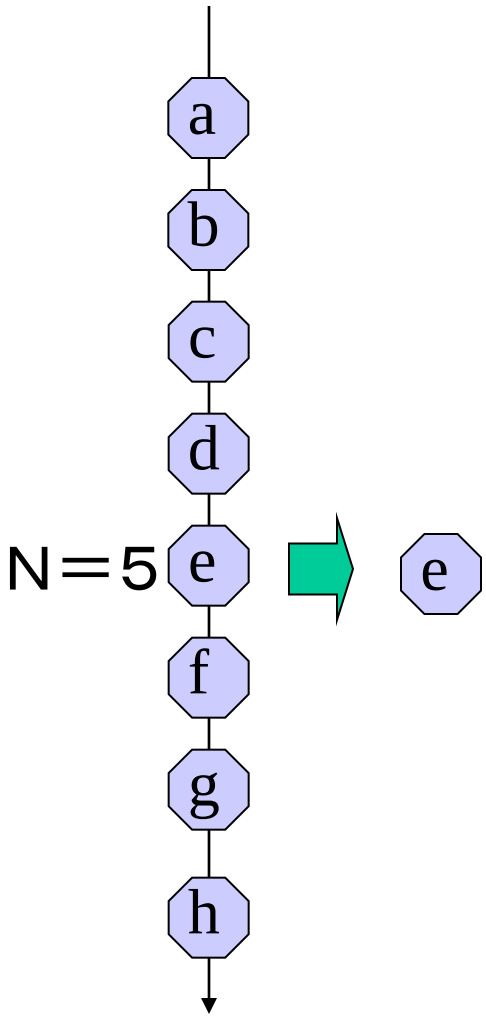

物理構造

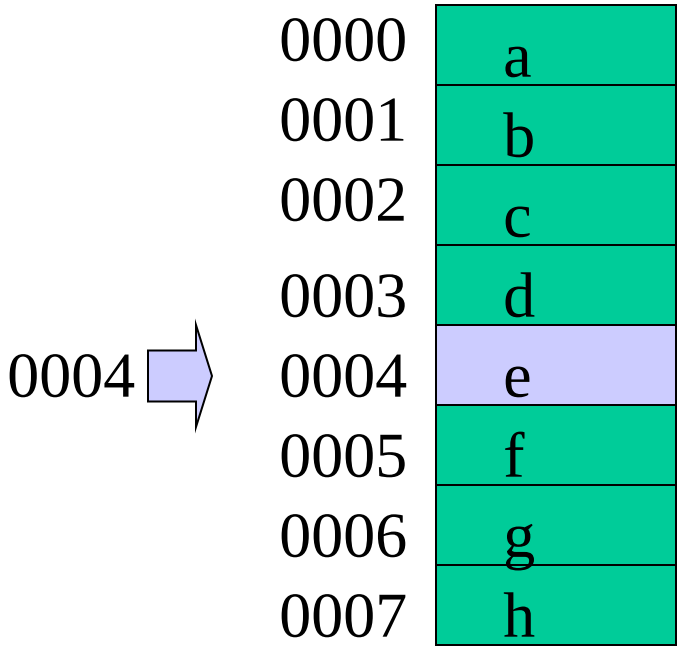


# 表に対する操作：アクセス

N番目の要素　N番目の要素のアドレス

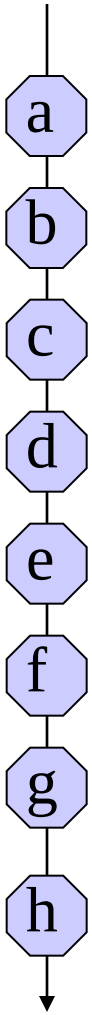
＝先頭番地＋（N－1）×要素サイズ



# リンク配置された線形リスト：連鎖リスト

論理構造

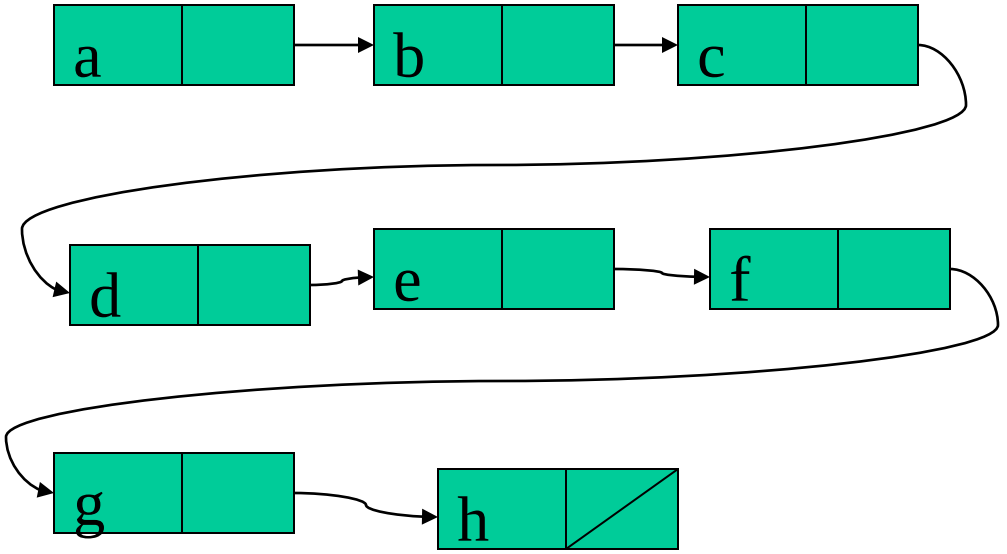



物理構造

実際

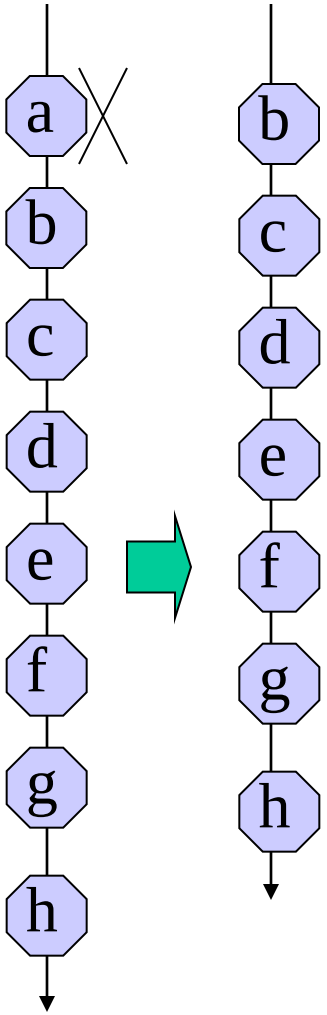
| | |
|---|---|
| 0000 | a 100b |
| 0003 | h null |
| 0008 | c 0032 |
| 000d | e 0106 |
| 001a | g 0003 |
| 0032 | d 000d |
| 0106 | f 001a |
| 100b | b 0008 |

記法

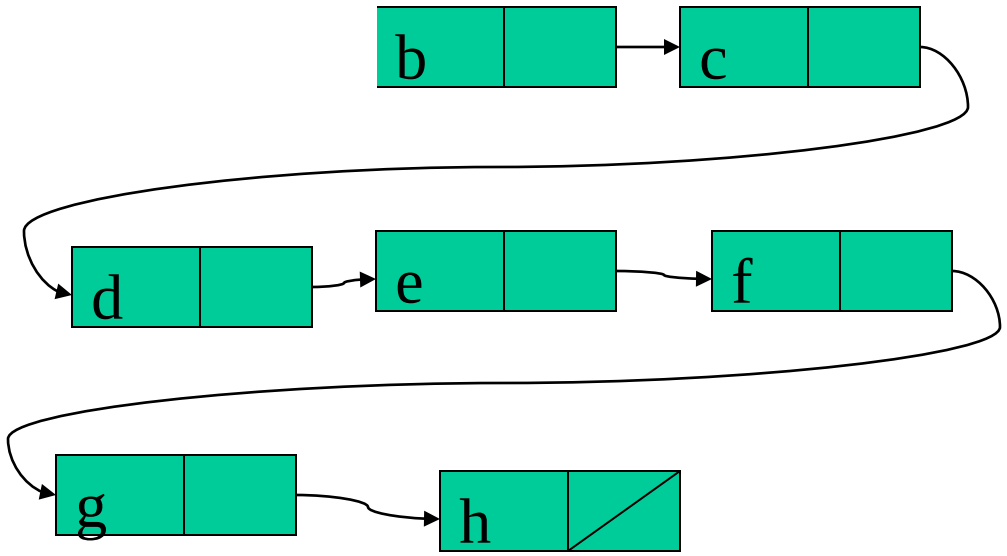


# 連鎖リストに対する操作：要素の挿入・削除

論理構造




物理構造



# 連鎖リストに対する操作：要素の挿入・削除

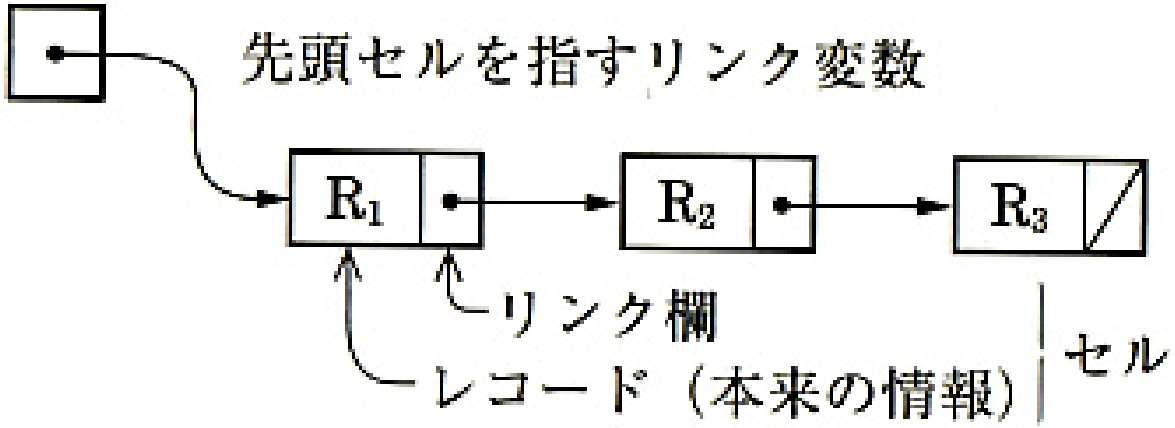
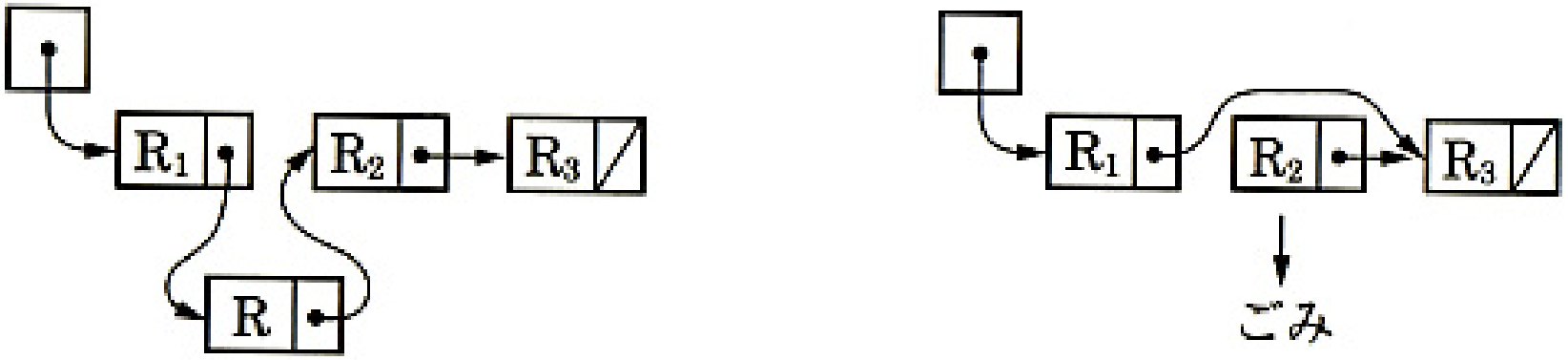



図 2・4　連鎖リストの基本構成

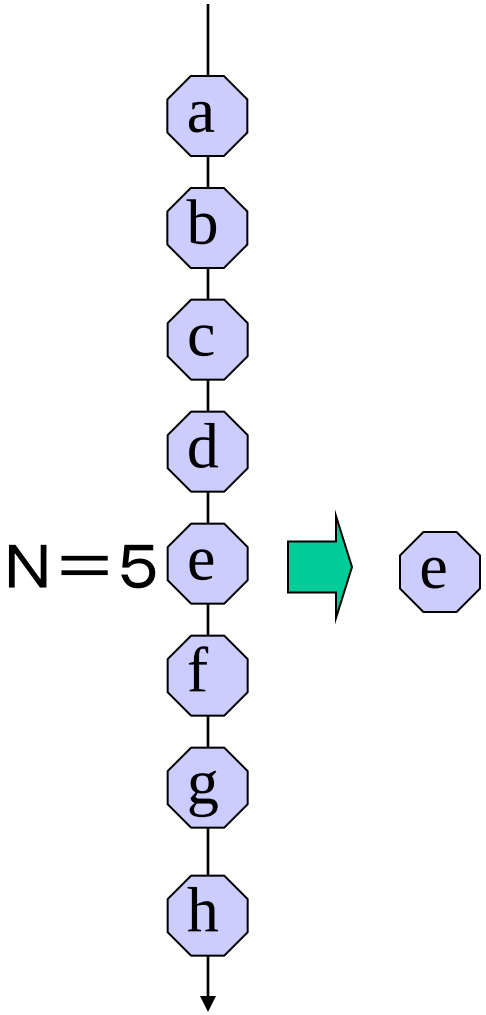



（a）$R_1$ と $R_2$ の間に R を追加　　（b）$R_2$ を削除

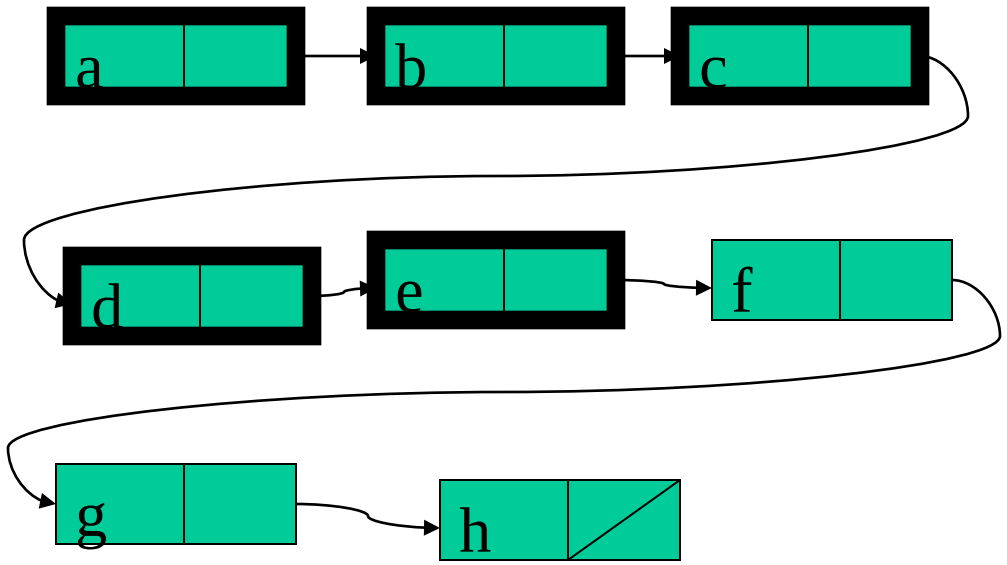
図 2・5　連鎖リストへのセルの追加と削除

# 連鎖リストに対する操作：アクセス

N番目の要素




N-1回リンクを辿る



# 連鎖リストの変種

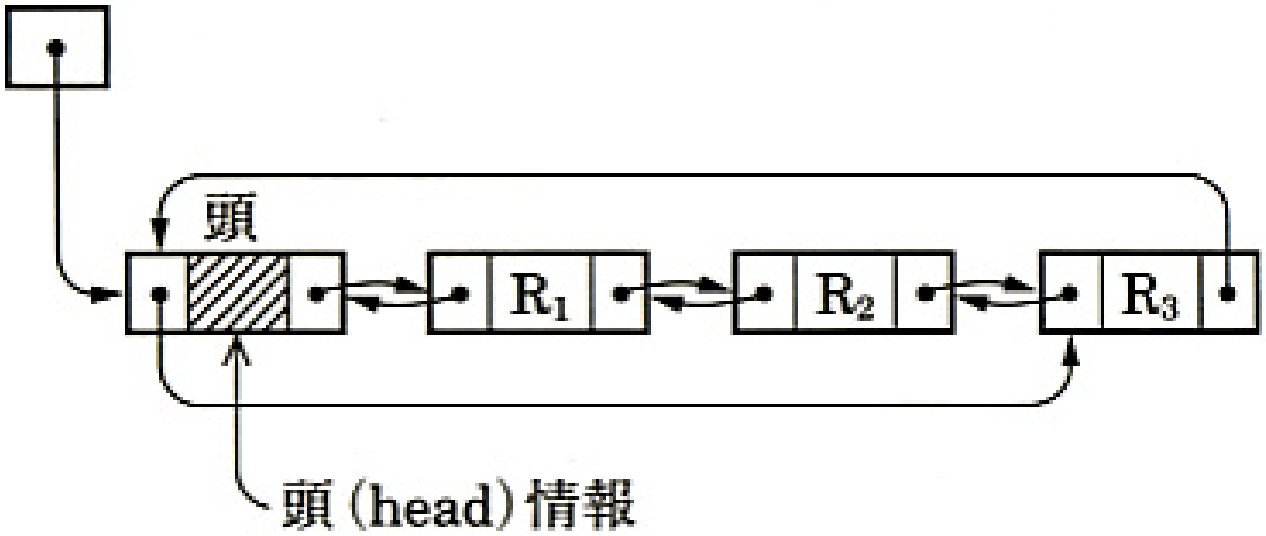



図 2·7　頭付き環状両方向リスト

# 論理構造の表現法（物理構造）

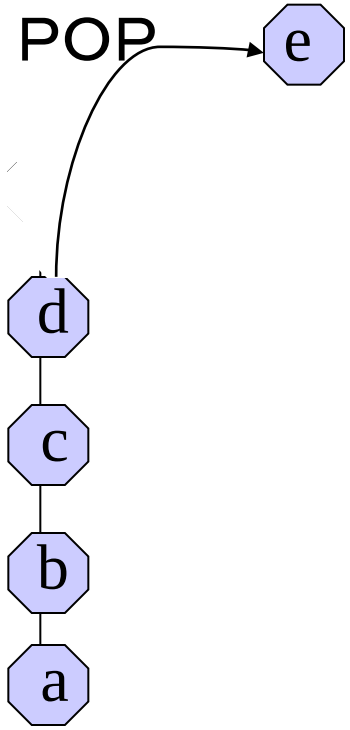
- 論理構造＝線形リスト
- 物理構造＝
  - **順配置**（表）
    - アクセスが早い、追加、削除、などの変更に弱い
  - **リンク配置**（連鎖リスト）
    - アクセスが遅い、追加、削除などの変更に適する

# スタックと待ち行列：データ抽象化

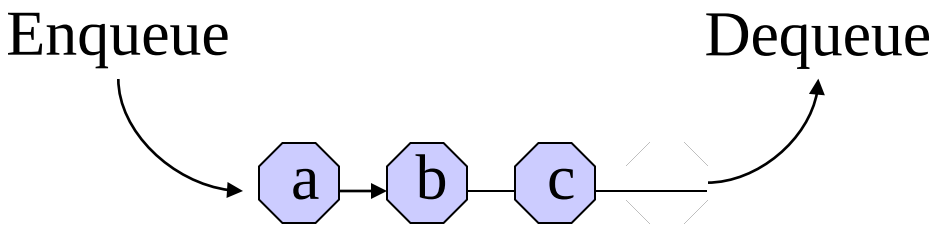
データ構造＋操作手続き

例：

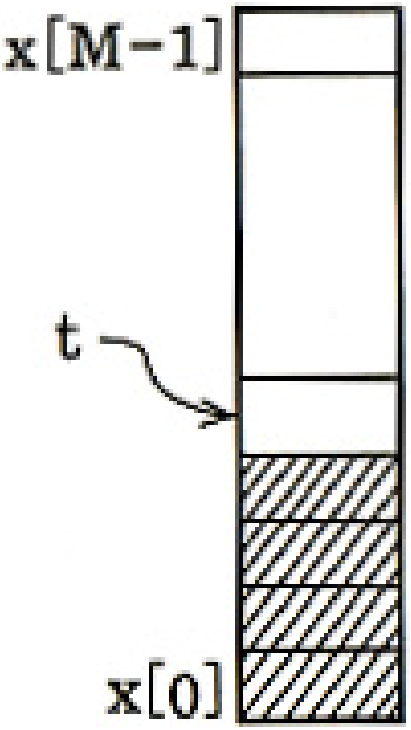
スタック




待ち行列



# スタックの操作（表を用いた場合）

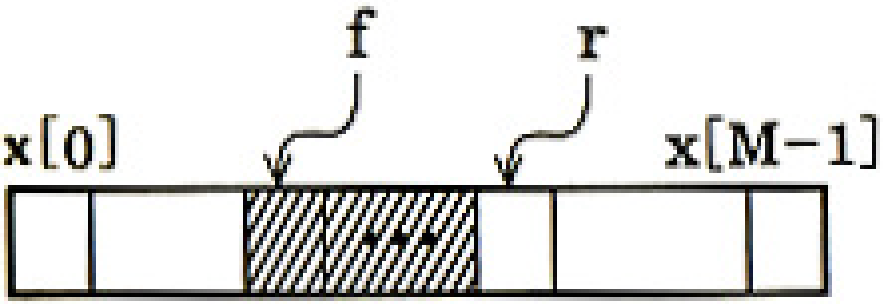



```
#define  M         1000
#define  OVERFLOW  -1
#define  UNDERFLOW -2

if (t < M) x[t++] = y;
else return OVERFLOW;        } (y を) プッシュ

if (t > 0) y = x[--t];
else return UNDERFLOW;       } (y へ) ポップ
```

図 2·12　順配置されたスタックの操作

# キューの操作(表を用いた場合)

```
#define  M          1000
#define  OVERFLOW   -1
#define  UNDERFLOW  -2
```




```
if (f-r == 1||f-r+m == 1)
    return OVERFLOW;
else {
    x[r] = y;
    if (++r == M) r = 0;
}
```

(y を) enqueue

```
if (f == r) return UNDERFLOW;
else {
    y = x[f++]
    if (f == M) f = 0;
}
```

(y へ) dequeue

図 2·13　順配置された待ち行列の操作

# スタック／キューは何のために用いるか

系統的な記憶と想起のメカニズム

- スタック（LIFO）
  - 環境の保存と参照ー＞再帰呼び出し
  - 木・グラフの縦型探索
- キューー（FIFO）
  - バッファ（緩衝用）メモリ、装置間の速度差の吸収
  - 木・グラフの横型探索

# スタックの利用の例：1

迷路の探索：

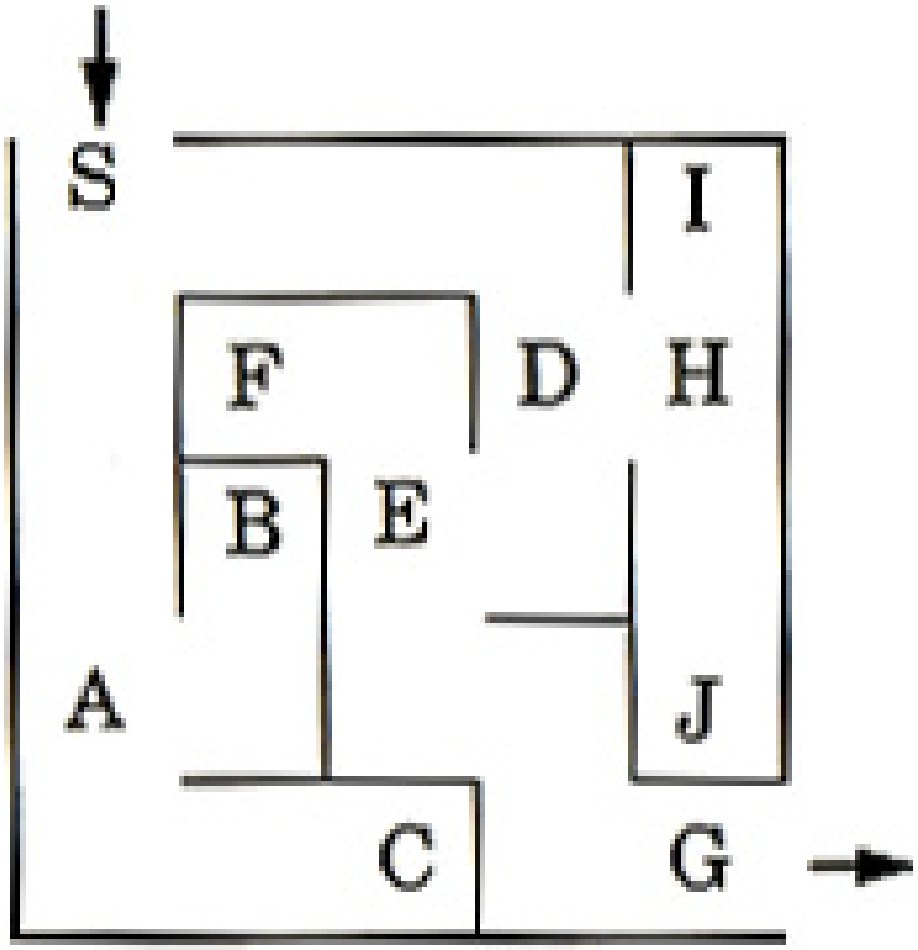



図 2·17　迷路

# 探索木

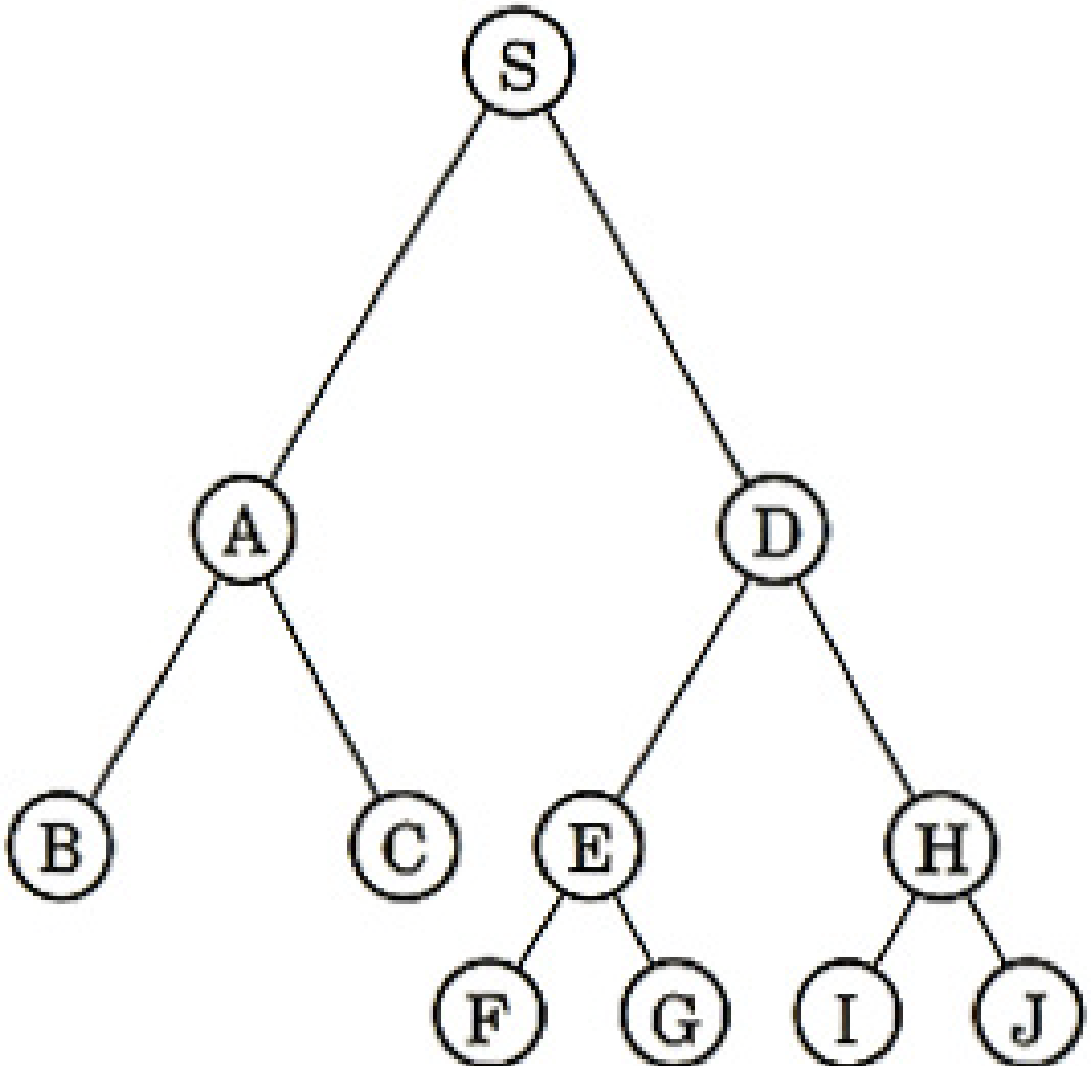



図 2･18　迷路の探索木

# 迷路の探索アルゴリズム

```
void search_solution()
{
    list = create_list();                          } ①
    add_list(S,list);
    while (empty(list) == 0) {                       ②
        n = branch_delete(list);                     ③
        if (goal(n)) return n;                       ④
        else expand_add(n,list);                     ⑤
    }
    return NULL;                                     ⑥
}
```

図 2·19　迷路の探索アルゴリズム

# list にスタックを用いた場合

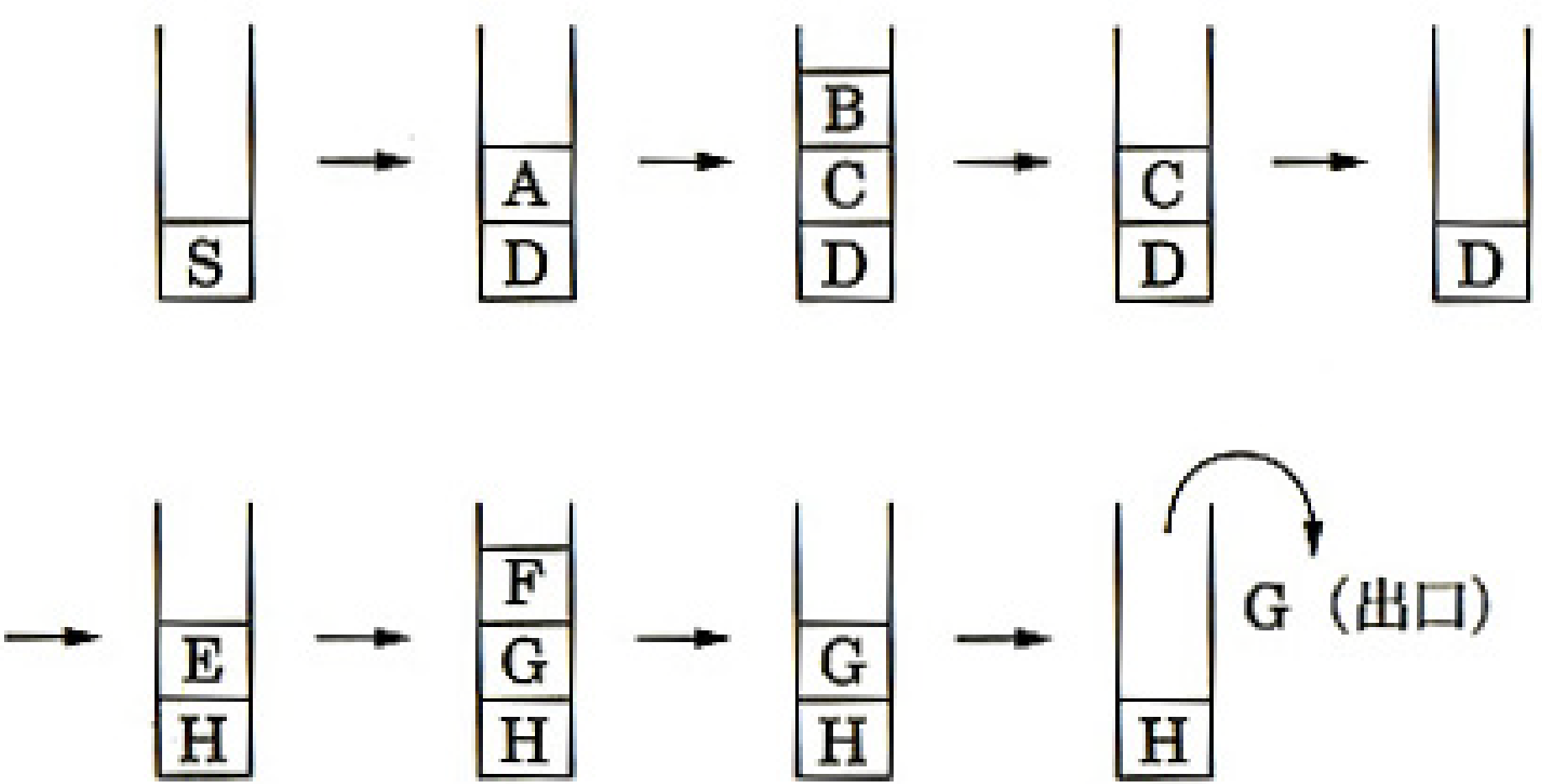



図 2·20　縦型探索におけるスタックの移り変わり

# 木の縦型探索とスタック

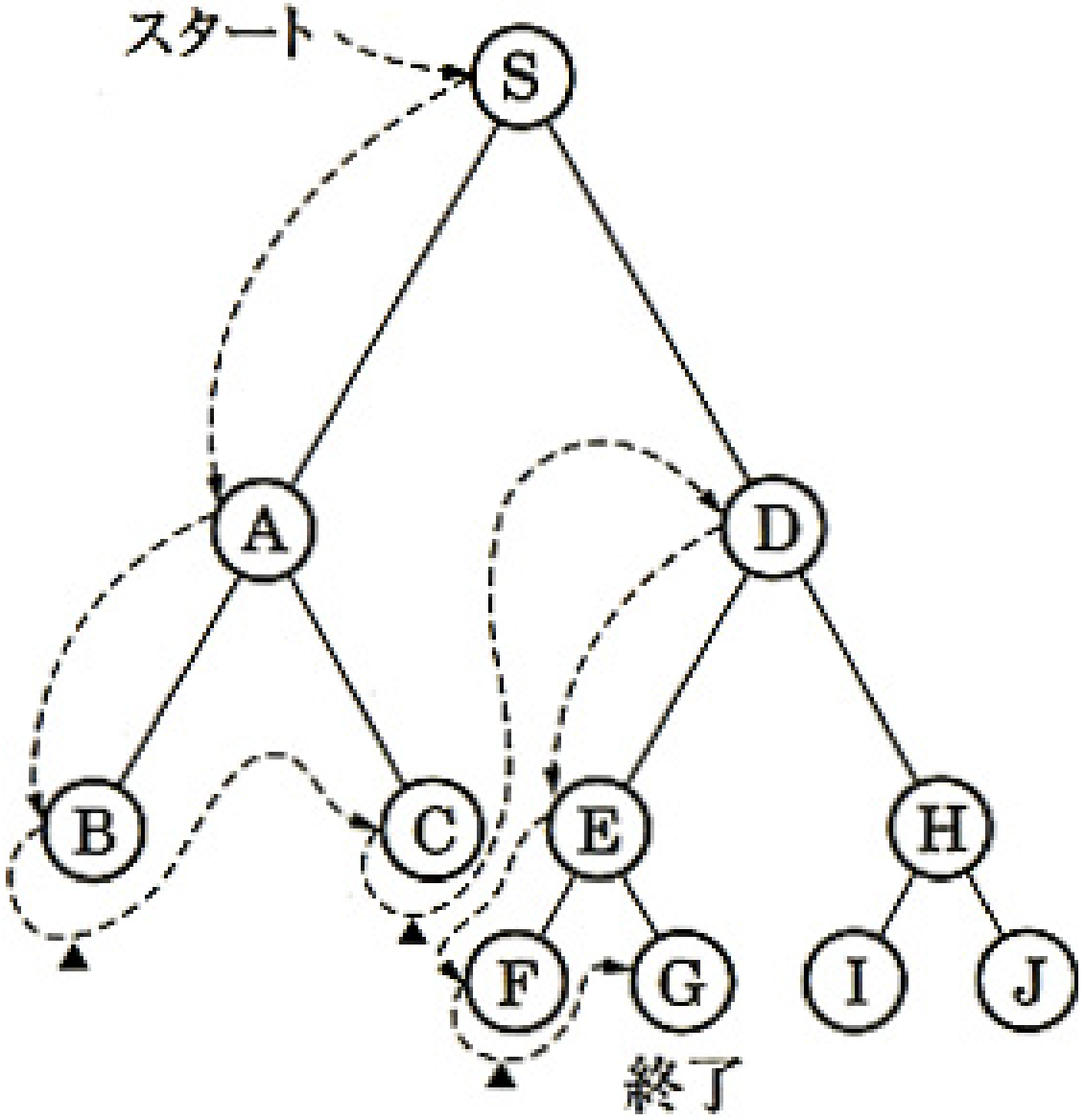



図 2·21　縦型探索における探索の流れ（▲は後戻り）

# 再帰的構造

$$f(i)=\begin{cases} 0, & i=0 \\ 1, & i=1 \\ f(i-1)+f(i-2), & i \geqq 2 \end{cases}$$

Fibonacci 数

```
int Fib(int i)
{ switch(i){
  case 0: return(0);break;
  case 1: return(1);break;
  default: return(Fib(i-1)+Fib(i-2));}
}
```

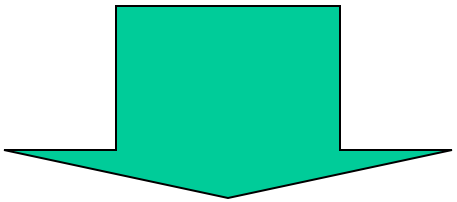
# 再帰的構造

```
int Fib(int i)
{ switch(i){
  case 0: return(0);break;
  case 1: return(1);break;
  default: return(Fib(i-1)+Fib(i-2));}
}
```

$$\begin{aligned} Fib(3) &\rightarrow Fib(2)+Fib(1) \\ &\rightarrow (Fib(1)+Fib(0))+1 \\ &\rightarrow (1+0)+1 \end{aligned}$$

# 再帰的構造

同じ変数名であっても、関数呼び出しの度に、別の記憶領域が確保される（スタックを利用している。）



変数の内容は、他の関数呼び出しによって破壊されることはない。

# Advanced-1 Fibonacci数を再帰呼び出しで求めるプログラム

```
#include <stdio.h>

int fib(int x)
{
    switch(x){
    case 0: return 0; break;
    case 1: return 1; break;
    default: return fib(x-1)+fib(x-2);
    }
}
```

```
int main(int argc, char *argv[])
{
  int x;
    while(--argc){
        x=atoi(*++argv);
        printf("Fib %d = %d¥n",x,fib(x));
    }
    return 0;
}
```

実行例： % ./fib 15
Fib 15 = 610

# Advanced-2 環状連鎖リストを使った電光掲示板風表示ログラム

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
//Self referential structure type definition
typedef struct node{
    char c;
    struct node *next;
} NODE;
```

実行例:

% ./link1 This is a pen

与えた文字列が横スクロールする

```
NODE *AllocNodes(int num)
{//Memory allocation
    NODE *rt;
    int i;
    rt=(NODE *)malloc(sizeof(NODE)
    *num   );
    for (i=0; i<num; i++){
      if (i<num-1) (rt+i)->next = (rt+i+1);
      else (rt+i)->next = rt;//connect tail to head
    }
    return rt;
}
```

# Advanced-2 続き

```
void SetChar(NODE *p, char *ref)
{
    while(*ref){
        p->c=*ref;
        p=p->next;
        ref++;
    }
}
```

```
PrintNode(NODE *p,int n)
{
    while(n--){
        printf("%c",p->c);
        p=p->next;
    }
    fflush(stdout);
}
```

```
int main(int argc, char *argv[])
{
int len;
char line[300];
NODE *s;
    line[0]=(char)NULL;
    while(--argc){//concatinate args
      strcat(line,*++argv);
      strcat(line," ");
      }
    len=strlen(line);//string length
    s=(NODE *)AllocNodes(len);
        SetChar(s,line);
    while(1){    //Endless Loop
        PrintNode(s,len);
        putchar ('¥r'); //Carridge return
        s=s->next;
        usleep(80000);
    }
}
```

# Advanced -2 STACK/QUEUE

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

//#define QUEUE      // キュー
#define STACK       //スタック
#define SIZ 10
```

```
/********** キューの定義**************/
#ifdef QUEUE
#define Add(Q,x)                en_queue(&Q,x)
#define Get_And_Delete(Q)  de_queue(&Q)
#define Not_Empty(Q)          Q.QF!=Q.QR
#define Init(Q)                   Q.QF=Q.QR=0
#define Stack_or_Queue       queue
#define QUEUE_SIZ SI     //キューのサイズ
struct Queue {
    int Buf[QUEUE_SIZ];
    int QF;
    int QR;
};
typedef struct Queue queue; /* QUEUE 用 */
```

```
void en_queue();
void en_queue(queue * Q, int v)
{
    Q->QR = (Q->QR+1)%QUEUE_SIZ;
    if (Q->QR == Q->QF) {
                fprintf(stderr,"Queue Overflow¥n");
                exit(1);
    }else{
                Q->Buf[Q->QR] = v;
    }
}

de_queue(queue * Q)
{
    if (Q->QR == Q->QF) {
                fprintf(stderr,"Queue Underflow¥n");
                exit(1);
    }else{
                Q->QF = (Q->QF+1)%QUEUE_SIZ;
                return Q->Buf[Q->QF];
    }
}
#endif
/********* キューの定義終わり **************/
```

# Advanced -2続き

```
/******** スタックの定義 *****************/
#ifdef STACK
#define Add(S,x)                push(&S,x)
#define Get_And_Delete(S)   pop(&S)
#define Not_Empty(S)          S.SP>0
#define Init(S)                  S.SP=0
#define Stack_or_Queue       stack
#define STACK_SIZ SIZ   //スタックのサイズ

struct Stack {
    int Buf[STACK_SIZ];
    int SP;
};
typedef struct Stack stack;

void push(stack* S,int d)
{
    if (S->SP<STACK_SIZ-1) {
                S->Buf[S->SP++]=d;
    }else{
                fprintf(stderr,"Stack overflow.¥n");
                exit(1);
    }
}
pop(stack* S)
{
    if (S->SP > 0) {
                return S->Buf[--(S->SP)];
    }else{
                fprintf(stderr,"Stack underflow.¥n");
                exit(1);
    }
}
#endif
/******* スタックの定義終わり **********/
```

```
main()
{
  int i,c;
    Stack_or_Queue X;


    Init(X);
    for (i=0; i< SIZ-1; i++) {
      c=(int)('A'+i);
      Add(X,c);putchar(c);
    }
    putchar('¥n');
    while(Not_Empty(X))
      putchar(Get_And_Delete(X));
    putchar('¥n');
}
```

実行例:

```
% stack-queue
ABCDEFGHI
IHGFEDCBA
```