線形探索は数が格納されている配列に,探したい物（キー）が含まれているかどうか,先頭から順に調べていく.入力は配列$(K_1, K_2, \dots, K_N)$,探したい物（key）,配列の長さ(N)とする.線形探索は成功か失敗かで終了する.今回は３種類の線形探索,(1)標準型,(2)番兵法,(3)並んでいるもの（昇順型）,について記述する.

(1) 標準型

標準型はもっとも基本的な線形探索である.前提として配列には必ず数が格納されているとする$(\mathrm{N} \geq 1)$.標準型は探索している配列に数が格納されているか確認$(i \leq N)$する必要がある.key が見つかれば,探索は成功として終了する.また key が見つからず,配列に数が格納されていなければ,探索は失敗として終了する.

(2) 番兵法

番兵法は配列の最後に key と同じ値（ダミー）を格納する.標準型とは異なり,配列に数が格納されているかどうか確認する必要がなくなる.標準型より速くなる.key と同じ値を比較しているとき,それがダミーでなければ,線形探索は成功として終了する,ダミーであれば,線形探索は失敗として終了する.

(3) 並んでいるもの（昇順型）

昇順型は昇順に値が並んでいる配列を探索する.比較している物$(K_i (1 \leq i \leq N))$が key を超えた時点で線形探索が成功か失敗か判定する.$K_i \geq key$のとき$K_i = key$ であれば線形探索は成功として終了する,$K_i != key$であれば失敗として終了する.

C 言語

単純法

```
#include<stdio.h>

int linear(int k[], int key, int n, int f)
{
    int i;

    for (i = 0; (i < n) && (f == -1); i++){
        if (k[i] == key){
            f = 1;
        }
    }

    return f;
```

```
}
```

番兵法__配列

```
#include<stdio.h>

int banpei(int k[], int key, int n)
{
    int i = 0;
    int f;

    k[n] = key;

    while (k[i] != key) {
        i++;
    }

    if (i < n){
        f = 1;
    }
    else{
        f = -1;
    }

    return f;
}
```

昇順の物__配列

```
#include<stdio.h>

int aolinear(int k[], int key, int n, int f)
{
    int i = 0;

    while (key > k[i]){
        i++;
    }
```

```
    if (k[i] == key && i != n){
        f = 1;
    }
    else{
        f = -1;
    }

    return f;
}

#define SMAX 1024

struct stack{
    int box[SMAX];
    int top;
};

void create(struct stack *s)
{
    s->top = -1;
}


void push(int *x, struct stack *s)
{
    s->top += 1;
    s->box[s->top] = *x;
    *x = 0;
}

void pop(int *x, struct stack *s)
{
    *x = s->box[s->top];
    s->top -= 1;
}
```

```
int empty(struct stack *s)
{
    return (s->top == -1);
}

int top(struct stack *s)
{
    return s->box[s->top];
}
```

単純法__スタック

```
int linear(struct stack *s, int key, struct stack *t)
{
    int f = -1;
    int x;

    while(!empty(s) && f == -1){
        if(top(s) == key){
            f=1;
        }
        pop(&x, s);
        push(&x, t);
    }
    return f;
}
```

番兵法__スタック

```
int banpei(struct stack *s, int key, struct stack *t)
{
    int temp = key;
    int x;
    int f = 0;
    push(&temp, t);

    while(!empty(s)){
```

```
        pop(&x, s);
        push(&x, t);
    }

    while(top(t) != key){
        pop(&x, t);
        push(&x, s);
    }
    pop(&x, t);
    push(&x, s);

    if(!empty(t)){
        f = 1;
    }
    else{
        f = -1;
    }

    return f;
}
```

昇順の物__スタック

```
int aolinear(struct stack *s, int key, struct stack *t)
{
    int x;
    int f = -1;

    while(top(s) < key){
        pop(&x, s);
        push(&x, t);
    }
    if(top(s) == key){
        pop(&x, s);
        push(&x, t);
        if(!empty(s)){
            f = 1;
```

```
        }
    }

    return f;
}
```

Janus

単純法__配列

```
procedure linear1(int k[], int key, int n, int f, stack g)
    local int i = 0

      push(1, g)
      from top(g) = 1 loop
        if k[i] = key then
            push(f, g)
            f ^= 1
            push(1, g)
        else
            push(0, g)
        fi top(g) = 1
            i += 1
            push(0, g)
      until i >= n || f != -1

      push(i, g)

    delocal int i = 0

procedure linear2(int k[], int key, int n, int f)
    local stack g = nil
     local int x = -1
      call linear1(k, key, n, x, g)
      f ^= x
      uncall linear1(k, key, n, x, g)
     delocal int x = -1
    delocal stack g = nil
```

番兵法__配列

```
procedure banpei1(int k[], int key, int n, int f, stack g)
  local int i = 0

    push(k[n], g)
    k[n] ^= key

    push(1, g)
    from top(g) = 1 loop
      i += 1
      push(0, g)
    until k[i] = key

    if i < n then
       push(f, g)
       f ^= 1
       push(1, g)
    else
       push(f, g)
       f ^= -1
       push(0, g)
    fi top(g) = 1

    push(i, g)
    delocal int i = 0

procedure banpei2(int k[], int key, int n, int f)
  local stack g = nil
    local int x = 0
      call banpei1(k, key, n, x, g)
      f ^= x
      uncall banpei1(k, key, n, x, g)
    delocal int x = 0
  delocal stack g = nil
```

昇順法__配列

```
procedure aolinear1(int k[], int key, int n, int f, stack g)
  local int i = 0

    push(1, g)
    from top(g) = 1 loop
      i += 1
      push(0, g)
    until key <= k[i]


    if k[i] = key && i != n then
       push(f, g)
       f ^= 1
       push(1, g)
    else
       push(f, g)
       f ^= -1
       push(0, g)
    fi top(g) = 1

    push(i, g)
  delocal int i = 0

procedure aolinear2(int k[], int key, int n, int f)
  local stack g = nil
    local int x = 0
      call aolinear1(k, key, n, x, g)
      f ^= x
      uncall aolinear1(k, key, n, x, g)
    delocal int x = 0
  delocal stack g = nil
```

単純法__スタック

```
procedure search(stack s, int key, stack t, int f, stack g)
```

```
  local int x = 0

   push(1, g)
   from top(g) = 1 loop
     if top(s) = key then
        push(f, g)
        f ^= 1
        push(1, g)
     else
        push(0, g)
     fi top(g) = 1
     pop(x, s)
     push(x, t)
     push(0, g)
   until f != -1   || empty(s)

   push(x, g)
 delocal int x = 0

procedure search1(stack s,int key,int f)
  local stack t = nil
   local stack g = nil
    local int y = -1
     call search(s, key, t, y, g)
     f ^= y
     uncall search(s, key, t, y, g)
    delocal int y = -1
   delocal stack g = nil
  delocal stack t = nil

番兵法__スタック
procedure banpei(stack s, int key, stack t, int f, stack gb)
  local int x = 0

    local int temp = key
      push(temp, t)
```

```
    delocal int temp = 0

    push(1, gb)
    from top(gb) = 1 loop
      pop(x, s)
      push(x, t)
      push(0, gb)
    until empty(s)

    push(1, gb)
    from top(gb) = 1 loop
      pop(x, t)
      push(x, s)
      push(0, gb)
    until top(t) = key

    pop(x, t)
    push(x, s)

    if !empty(t) then
       push(f, gb)
       f ^= 1
       push(1, gb)
    else
       push(f, gb)
       f ^= -1
       push(0, gb)
    fi top(gb) = 1

    push(x, gb)
  delocal int x = 0

procedure banpei1(stack s, int key, int f)
  local stack g = nil
   local stack t = nil
    local int y = 0
```

```
    call banpei(s, key, t, y, g)
    f ^= y
    uncall banpei(s, key, t, y, g)
   delocal int y = 0
  delocal stack t = nil
 delocal stack g = nil
```

昇順法__スタック

```
procedure aolinear(stack s, int key, stack t, int f, stack g)
  local int x = 0

    push(1, g)
    from top(g) = 1 loop
      pop(x, s)
      push(x, t)
      push(0, g)
    until top(s) >= key

    if top(s) = key then
       pop(x, s)
       push(x, t)
       if !empty(s) then
          push(f, g)
          f ^= 1
          push(1, g)
       else
          push(0, g)
       fi top(g) = 1
       push(1, g)
    else
       push(0, g)
    fi top(g) = 1

    push(x, g)
  delocal int x = 0
```

```
procedure aolinear1(stack s, int key, int f)
  local stack g = nil
   local stack t = nil
    local int y = -1
      call aolinear(s, key, t, y, g)
      f ^= y
      uncall aolinear(s, key, t, y, g)
    delocal int y = -1
   delocal stack t = nil
  delocal stack g = nil
```

Janus 言語について

Janus は命令型可逆プログラミング言語である.Janus は可逆性を保証する構文規則を持つ.Janus を利用することで可逆なプログラミングを記述することが出来る.

Janus の変数宣言は int 型の変数,int 型の一次元配列と int 型のスタックを定義する.代入する際、+=（加算）,-=（減算）,^=（排他的論理和）を使用する.ただし代入式の左側に現れた変数は右側に現れてはいけない,すなわち x+=x という計算は行うことが出来ない.代入は変数の値を変える唯一の方法である.

条件式 if e1 then s1 else s2 fi e2 について述べる.式 e1 が真であるとき文 s1 を実行する.文 s1 を実行後,式 e2 は真でなければならない.式 e1 が偽であるとき文 s2 を実行する.文 s2 実行後,式 e2 は偽でなければならない.

繰り返し式 from e1 do s1 loop s2 until e2 について述べる.式 e1 は最初,真でなければならない.後の反復では式 e1 は偽でなければならない.最初に文 s1 が実行される.式 e2 が偽であれば文 s2 を実行し,式 e2 が真であれば実行を終了する.

データ型スタックの操作について述べる.push(x,s)はスタック s に要素 x を追加し,x を 0 クリアする.pop(x,s)はスタック s から一番上の物を取り出し,x に格納する.

local はローカル変数に記憶領域を割り当て,変数を式と同じ値で初期化する.local と delocal に囲まれた部分でしかその変数を扱うことができない.

関数を呼び出すには call と uncall を用いる.call は関数を呼び出すことができる.uncall は関数を逆呼び出しすることが出来る.

C 言語から Janus への書き換え

代入

C 言語

```
x = 1;
```

Janus

```
push(x, g)
x^=1
```

Janus での代入,.x^=1 は x が 0 であるときに実行できるので,push(x, g)することにより x を 0 クリアしてから x ^= 1 を行う.

変数宣言

C 言語

```
int x = 0;
```

Janus

```
local int x = 0
   push(x, g)
delocal int x = 0
```

Janus では変数宣言のとき local int を用いる.delocal int の前に push(x, g)し,x を 0 クリアすることで x=0 を成り立たせている.

if 文

C 言語

```
if (e1) {
    s1
}
else{
    s2
}
```

Janus

```
if e1 then
    s1
    push(1, g)
else
    s2
    push(0, g)
fi top(g)=1
```

assertion は上で記述したように if 節を実行したときは真であり,else 節を実行したときは偽でなければならない.ここでは if 節を実行したとき最後に push(1, g)をすること

で,assertion : top(g) = 1 が必ず真になるようにしている.また,else 節を実行したとき最後に push(0, g)をすることで,assertion が必ず偽になるようにしている.

繰り返し文１
C 言語

```
while (e1){
      s1
}
```

Janus

```
push(1, g)
from top(g)=1 loop
   s1
   push(0, g)
until !e1
```

loop 文の assertion は最初,真でなければならない.push(1, g)することで assertion:top(g)=1 が必ず成り立つようにしている.最初以外 assertion は偽でなければならない.loop 節の最後で push(0, g)することで assertion が偽になるようにしている.

繰り返し文２
C 言語

```
for (i=0;   e1   i++){
      s1
}
```

Janus

```
push(1, g)
from top(g)=1 loop
   s1
   i+=1
   push(0, g)
until !e1
```

C 言語の while 文の条件 e1 を Janus では until に e1 の否定を条件として記述する.Janus での繰り返し文の assertion は最初真でなければならない.from の前で push(1, g)することで assertion : top(g)=1 が必ず成り立つようにしている.最初以外 assertion は偽でなければ

ならない.loop 節の最後で push(0, g)することで assertion が必ず偽になるようにしている.

可逆シミュレーション

すべての関数$f: X \to Y$について,ある G が存在し,すべての$x \in X$について,$fst\big(f'(x)\big) = f(x)$を満たす,単射関数$f': X \to Y \times G$が存在する.$f'$は$f$を単射化したものである.出力 G は$f'$を単射にするもので,ゴミ出力と言われる.

IRを非可逆言語,Rを可逆言語とする.可逆言語 Rで書かれたプログラム qが非可逆言語 IRで書かれたプログラム p の可逆シミュレーションであるというのは,全ての x について,$fst\big([\![q]\!]^R(x)\big) = [\![p]\!]^{IR}(x)$であることである.これは$[\![q]\!]^R$が$[\![p]\!]^{IR}$から単射化された関数であることを表す.

埋め込み

ランダウアー法は埋め込みと呼ばれる方法である.可逆性を保証するため,出力にごみデータを追加し,非可逆な計算により失われるデータ,制御情報を保存する.例えばc言語でのx=3という代入は直前の x の値が分からなくなってしまうので非可逆である.Janus では x^=3をする前に push(x,g)によって代入の直前の値を保存することで可逆になる.埋め込みによる可逆シミュレーションの方法では時間的,空間的な面でトレードオフを持つ方法が広く研究されている.

Bennett 法

Janus における Bennett 法は call,結果のコピー,uncall によって行われる.uncall は関数を逆呼び出しすることが出来る.Bennett 法は出力に入力したものを追加する.call による順方向の計算が呼び出されると,可逆性を保証するために不要なゴミ情報が出てくる. このとき uncall により関数を逆呼び出しすることで call により生じたゴミ情報を浄化することが出来る.

参考文献

Holger Bock Axelsen, Tetsuo Yokoyama : Programming Techniques for Reversible Comparison Sorts.
Tetsuo Yokoyama : Reversible Computation and Reversible Programming Languages.
C.H.Bennett : Logical Reversibility of Computation.
DONALD E.KNUTH : The Art of Computer Programming Volume 3 Sorting and Searching Second Edition 日本語版.