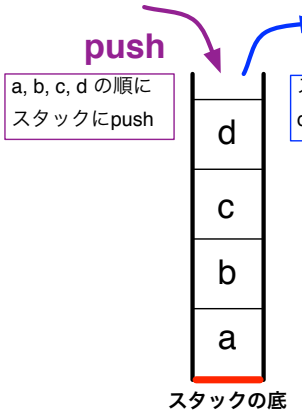
データ構造 スタック

大東文化大学
水谷正大

スタック（stack, push-down stack）

1次元的なデータの並びで、データの追加（挿入）や取り出し（削除）を一方の『端』からだけに制限したデータ構造

スタックデータの読み出しはスタックの『端』からデータを取り出した上でのみ可能。端以外の場所にある要素は、端から順にデータを取り出していって、それが端になるまでは参照できない（スタック中のデータを直接読み出したり書き込んだりすることは許されない）



スタックに可能な操作

clearstack
初期化：空スタックを用意

push
スタックにデータ追加する

pop
スタックからデータを取り出す

LIFO(Last In First Out)
データアクセスは後入れ先出し

待ち行列（queue）

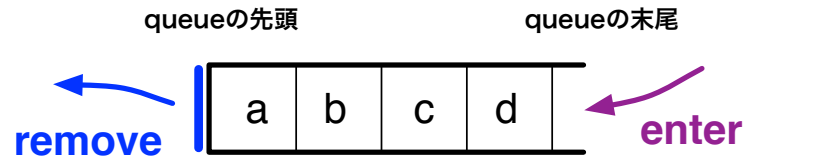
1次元的なデータの並びで、データの追加（挿入）をある一方、取り出し（削除）をもう一方からだけに制限したデータ構造

待ち行列に可能な操作

enter
待ち行列ににデータを追加

clearqueue
初期化：空の待ち行列を用意

premove
待ち行列からデータを取り出す



FIFO(First In First Out)
データアクセスは先入れ先出し

スタックのプログラム動作記述

```
clearstack s;
```

空スタック s = {}; を準備する

```
s.push(a);
```

スタック s に a をpushする　s <= {a};

s = {a, b, c, d}; スタックの底は　{.....}; の左端

s = {a, b, c}; に push すると　s <= {a, b, c, d}; 右端に追加

```
p = s.pop;
```

端のd が取り出されて　p <= d, s <= {a, b, c};

プログラム例

```
clearstack s;       // スタックを用意 s = {};
s.push(1);          // s = {1};
s.push(2);          // s = {1, 2};
s.push(3);          // s = {1, 2, 3};
p = s.pop;          // s = {1, 2}
    print p;        //  3
p = s.pop;          // s = {1};
     print p;       //  2
p = s.pop;          // s = {};
    print p;         //  1

s.empty;        //  true
```

プログラム例 括弧 ( ) の対応を調べる

括弧の初め（ が来たらpush、括弧の終わり ） が来たらpopする
文字列 strを調べ終わった後にスタック s が空でないときは対応がとれていない

```
clearstack s;
bool flag = true;

for(i=0 ; i < str.size() ; i++ ){
    char c = str[i];
    if( c == '(' ){ // 括弧の初めが来たら
        s.push( c );
    } else if( c == ')' ){ // 括弧の終わりが来たら
        if( s.empty() )
            flag = false;
        else
            s.pop();
    }
}
if( flag && st.empty() )
     cout << "Yes";
else
     cout << "No";
```

```
(1+1)                   Yes
3*(2+1)+(6-(5*4))       Yes
(1+1)*(2+(2*3)          No
4(*(2-3)                No
((1+2))/(4*(3+8)-5)     Yes
```

プログラム例 括弧 ( ),[ ] の対応

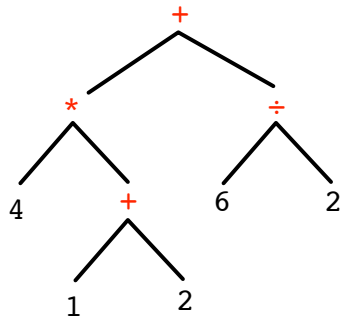
丸括弧（ ）と 角括弧［ ］がバランスしているか
丸括弧)が来たら (に、角括弧 ］が来たら［ に必ず対応すべし

```
clearstack s;
bool flag = true;
for(int i=0 ; i < str.size() ; i++ ){
        char c = str[i];
    if( c == '(' || c == '['){ // 括弧の初めが来た
        s.push( c );
    } else if( c == ')' ){ // 丸括弧の終わり ')' が来た
        if( s.empty() || s.top() == '[' ){
            flag = false;
            break;
        } else if( s.top() == '(' ){
            s.pop();
        }
    }else if( c == ']' ){ // 角括弧の終わり ']' が来た
        if( s.empty() || s.top() == '(' ){
            flag = false;
            break;
        } else if( s.top() == '[' ){
            s.pop();
        }
    }
}
if( flag && s.empty() )
    cout << "yes";
else
    cout << "no";
```

```
(the [first] I will see at (school) is you).   Yes
[ First In ] ( First Out ).                    Yes
This night, (walk with [you until] dawn).      Yes
( [ ( ] ) ) [ ( ) ]                            No
```

プログラム例 四則演算の逆ポーランド記法

2項演算子の後置記法　　3+4 ⇒ 34+　　(3+4)*2 ⇒ 34+2*



1 2 + 4 * 6 2 ÷ +

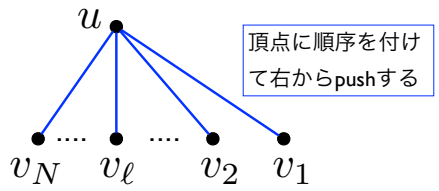
```
clearstack s;               s ={};

s.push( 1 );                s = {1};
s.push( 2 );                s = {1, 2};
s.push( s.pop + s.pop );    s = {3};
s.psh( 4 );                 s = {3, 4}
s.push( s.pop * s.pop );    s = {12};
s.push( 6 );                s = {12, 6};
s.push( 2 );                s = {12, 6,
2};
s.push( s.pop / s.pop );    s = {12, 3};
s.push( s.pop + s.pop );    s = {15};

cout << s.pop;              15
```

プログラム例

深さ優先探索

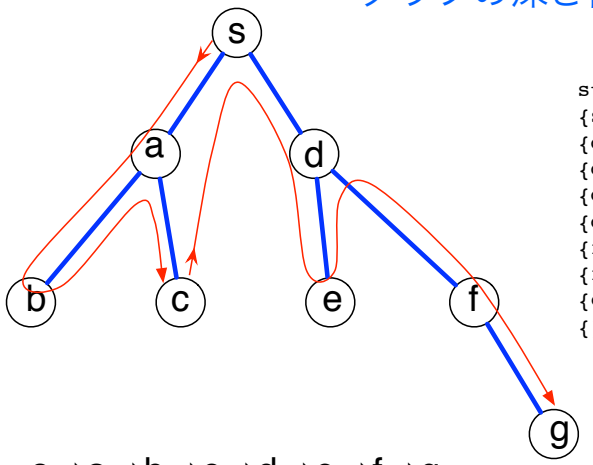
DFS(s)　　頂点 s からの深さ優先探索

```
clearstack s;
s.push( s );
while ( not s.empty ) {
    u = s.pop;
    if( visited( u ) == false ) { //まだ u を未訪問なら
        visited( u ) <= true;
        something for 頂点 u;
        for (each u に接続する頂点 v) {
            s.push( v );
        }
    }
}
```



頂点に順序を付けて右からpushする

グラフの深さ優先探索




```
stack={};
{s};
{d, a};        visit s
{d, c, b};     visit a
{d, c};        visit b
{d};           visit c
{f, e}         visit d
{f};           visit e
{g};           visit f
{};            visit g
```

s→a→b→c→d→e→f→g