CHAPTER 4

STACKS

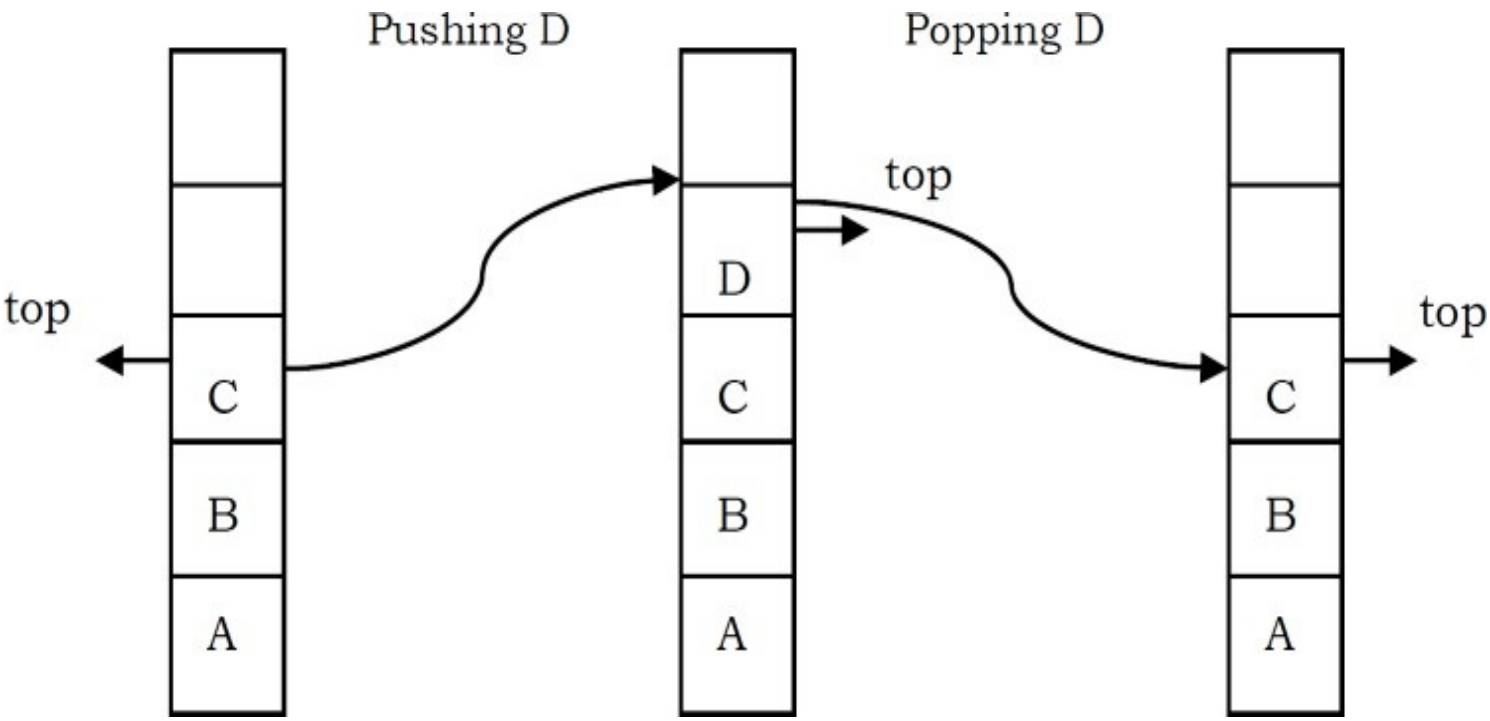
4.1 Apa itu Stack?

Sebuah stack adalah struktur data sederhana yang digunakan untuk menyimpan data (mirip dengan Linked Lists). Dalam tumpukan, urutan data tiba penting. Setumpuk piring di kantin adalah contoh yang baik dari stack. Piring ditambahkan ke tumpukan karena mereka dibersihkan dan mereka ditempatkan di atas. Ketika piring, diperlukan diambil dari bagian atas tumpukan. Piring pertama ditempatkan di stack adalah yang terakhir yang akan digunakan.

Definisi: SEBUAH *tumpukan* adalah daftar memerintahkan di mana penyisipan dan penghapusan dilakukan di salah satu ujung, yang disebut *teratas.* Unsur terakhir dimasukkan adalah yang pertama yang akan dihapus. Oleh karena itu, hal itu disebut terakhir di Pertama keluar (LIFO) atau Pertama di terakhir keluar daftar (Filo).

nama khusus yang diberikan kepada dua perubahan yang dapat dilakukan untuk tumpukan. Ketika elemen dimasukkan dalam tumpukan, konsep ini disebut *Dorong,* dan ketika elemen dihapus dari stack, konsep ini disebut *pop.* Mencoba untuk pop keluar stack kosong disebut *underflo* dan mencoba untuk mendorong elemen dalam tumpukan penuh disebut *meluap.* Umumnya, kita memperlakukan mereka sebagai pengecualian. Sebagai contoh,

mempertimbangkan snapshot dari stack.



4.2 Bagaimana Tumpukan digunakan

Pertimbangkan hari kerja di kantor. Mari kita asumsikan pengembang bekerja pada sebuah proyek jangka panjang. Manajer kemudian memberikan pengembang tugas baru yang lebih penting. Pengembang menempatkan proyek jangka panjang ke samping dan mulai bekerja pada tugas baru. telepon berdering, dan ini adalah prioritas tertinggi karena harus dijawab segera. Pengembang mendorong tugas ini ke dalam baki tertunda dan menjawab telepon.

Bila panggilan selesai tugas yang ditinggalkan untuk menjawab telepon diambil dari tertunda tray dan bekerja kemajuan. Untuk mengambil panggilan lain, itu mungkin harus ditangani dengan cara yang sama, tapi akhirnya tugas baru akan selesai, dan pengembang dapat menarik proyek jangka panjang dari baki tertunda dan melanjutkan dengan itu.

4.3 Stack ADT

Operasi berikut membuat tumpukan sebuah ADT. Untuk mempermudah, asumsikan data adalah tipe integer.

operasi stack utama

- Dorongan (int data): Sisipan *data* ke stack.
- int Pop (): Menghapus dan kembali elemen terakhir dimasukkan dari stack.

operasi stack tambahan

- int Top (): Mengembalikan elemen dimasukkan terakhir tanpa menghapusnya.
- int Ukuran (): Mengembalikan jumlah elemen disimpan dalam stack.
- int IsEmptyStack (): Menunjukkan apakah ada unsur-unsur disimpan dalam tumpukan atau tidak.
- int IsFullStack (): Menunjukkan apakah stack penuh atau tidak.

pengecualian

Mencoba pelaksanaan operasi kadang-kadang dapat menyebabkan kondisi kesalahan, disebut pengecualian. Pengecualian dikatakan “dilemparkan” oleh sebuah operasi yang tidak dapat dijalankan. Dalam Stack ADT, operasi pop dan atas tidak dapat dilakukan jika stack kosong. Mencoba pelaksanaan pop (top) pada stack kosong melempar pengecualian. Mencoba untuk mendorong elemen dalam tumpukan penuh melempar pengecualian.

4.4 Aplikasi

Berikut adalah beberapa aplikasi di mana tumpukan memainkan peran penting.

aplikasi langsung

- Menyeimbangkan simbol
- Infiks-to-postfix konversi
- Evaluasi ekspresi postfix
- Menerapkan fungsi panggilan (termasuk rekursi)
- Menemukan bentang (menemukan bentang di pasar saham, lihat *masalah* bagian)
- Halaman-mengunjungi sejarah dalam browser Web [Kembali Buttons]
- Undo urut dalam editor teks
- Pencocokan Tag di HTML dan XML

aplikasi tidak langsung

- struktur data tambahan untuk algoritma lainnya (Contoh: Pohon algoritma traversal)
- Komponen struktur data lainnya (Contoh: Simulasi antrian, merujuk *antrian* bab)

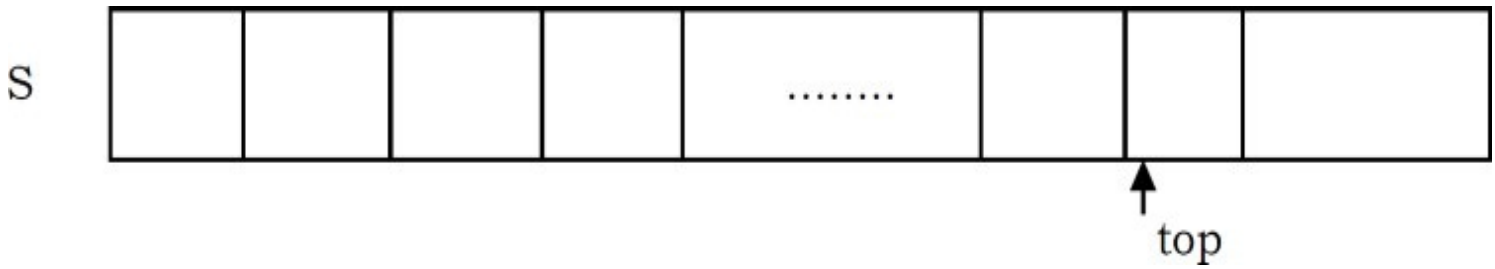
4,5 Implementasi

Ada banyak cara melaksanakan tumpukan ADT; di bawah ini adalah metode yang umum digunakan.

- Sederhana implementasi array berdasarkan
- Dinamis implementasi array berdasarkan
- Pelaksanaan daftar terkait

Implementasi Array sederhana

Implementasi dari ADT tumpukan menggunakan sebuah array. Dalam array, kita menambahkan elemen dari kiri ke kanan dan menggunakan variabel untuk melacak indeks dari elemen atas.



Array menyimpan elemen tumpukan dapat menjadi penuh. Sebuah operasi push maka akan melempar *penuh tumpukan pengecualian.* Demikian pula, jika kita mencoba menghapus sebuah elemen dari stack kosong itu akan membuang *tumpukan pengecualian kosong*.

```
#define MAXSIZE 10
struct ArrayStack {
        int top;
        int capacity;
        int *array;
};
struct ArrayStack *CreateStack() {
        struct ArrayStack *S = malloc(sizeof(struct ArrayStack));
        if(!S)
           return NULL;
        S→capacity = MAXSIZE;
        S→top = -1;
        S→array= malloc(S→capacity * sizeof(int));
        if(!S→array)
           return NULL;
        return S;
}
int IsEmptyStack(struct ArrayStack *S) {
        return (S→top == -1);     // if the condition is true then 1 is returned else 0 is returned
}
int IsFullStack(struct ArrayStack *S){
        //if the condition is true then 1 is returned else 0 is returned
        return (S→top == S→capacity - 1);
}
void Push(struct ArrayStack *S, int data){
        /* S→top == capacity -1 indicates that the stack is full*/
        if(IsFullStack(S))
            printf( "Stack Overflow");
        else        /*Increasing the 'top' by 1 and storing the value at 'top' position*/
            S→ array[++S→top]= data;
}
int Pop(struct ArrayStack *S){
      /* S→top == - 1 indicates empty stack*/
        if(IsEmptyStack(S)){
           printf("Stack is Empty");
           return INT_MIN;;
        }
        else /* Removing element from  'top' of the array and reducing  'top' by 1*/
           return (S→ array[S→top--]);
}
void DeleteStack(struct DynArrayStack *S){
      if(S) {
         if(S→array)
            free(S→array);
        free(S);
      }
}
```

Kinerja & Keterbatasan

prestasi

Membiarkan *n* menjadi jumlah elemen dalam stack. Kompleksitas operasi stack dengan representasi ini dapat diberikan sebagai:

Ruang Kompleksitas (untuk operasi n dorongan)	HAI(*n)*
Waktu Kompleksitas Push ()	O (1)
Waktu Kompleksitas Pop ()	O (1)
Waktu Kompleksitas Ukuran ()	O (1)
Waktu Kompleksitas IsEmptyStack ()	O (1)
Waktu Kompleksitas IsFullStackf)	O (1)
Waktu Kompleksitas DeleteStackQ	O (1)

keterbatasan

Ukuran maksimum tumpukan pertama harus didefinisikan dan tidak dapat diubah. Mencoba untuk mendorong elemen baru ke dalam tumpukan penuh menyebabkan pengecualian implementasi khusus.

Implementasi Array Dinamis

Pertama, mari kita mempertimbangkan bagaimana kita menerapkan tumpukan berbasis array sederhana. Kami mengambil satu variabel indeks *teratas* yang poin ke indeks dari elemen yang paling baru-baru ini dimasukkan ke dalam stack. Untuk insert (atau dorongan) unsur, kami increment *teratas*

Indeks dan kemudian menempatkan elemen baru pada indeks itu. Demikian pula, untuk menghapus (atau pop) unsur kita mengambil elemen di *teratas*

Indeks dan kemudian pengurangan tersebut
teratas indeks. Kami mewakili antrian kosong dengan *teratas* nilai sama dengan -1. Masalah yang masih perlu diselesaikan adalah apa yang kita lakukan ketika semua slot di ukuran array tumpukan tetap ditempati?

Percobaan pertama: Bagaimana jika kita kenaikan ukuran array oleh 1 setiap kali stack penuh?

- Dorong(); meningkatkan ukuran dari S [] oleh 1
- Pop (): Ukuran penurunan S [] oleh 1

Masalah dengan pendekatan ini?

cara ini incrementing ukuran array terlalu mahal. Mari kita lihat alasan untuk ini. Sebagai contoh, di $n = 1$, untuk mendorong elemen membuat array baru ukuran 2 dan menyalin semua elemen array lama ke array baru, dan pada akhirnya menambahkan elemen baru. Di $n = 2$, untuk mendorong elemen membuat array baru ukuran 3 dan menyalin semua elemen array lama ke array baru, dan pada akhirnya menambahkan elemen baru. Demikian pula, di $n = n - 1$, jika kita ingin mendorong elemen membuat array baru ukuran n dan menyalin semua elemen array lama ke array baru dan pada akhirnya menambahkan elemen baru. Setelah n dorongan operasi total waktu $T(n)$ (n Banyaknya operasi copy) sebanding dengan $1 + 2 + ... + n \approx O(n^2)$.

Pendekatan Alternatif: Diulang dua kali lipat

Mari kita meningkatkan kompleksitas dengan menggunakan array *dua kali lipat* teknik. Jika array penuh, membuat array baru dari dua kali ukuran, dan menyalin item. Dengan pendekatan ini, mendorong n item membutuhkan waktu sebanding dengan n (tidak n^2).

Untuk mempermudah, mari kita asumsikan bahwa awalnya kami mulai dengan $n = 1$ dan pindah ke $n = 32$. Itu berarti, kami melakukan penggandaan di 1,2,4,8,16. Cara lain menganalisa pendekatan yang sama adalah: di $n =$ 1, jika kita ingin menambahkan (push) elemen, dua kali lipat ukuran saat array dan menyalin semua elemen dari array lama ke array baru. Di $n = 1$, kita lakukan 1 copy operasi, di $n = 2$, kami melakukan 2 operasi copy, dan pada $n = 4$, kita lakukan 4 operasi copy dan sebagainya. Oleh kami jangkauan waktu $n = 32$, jumlah operasi copy adalah $1 + 2 + 4 + 8 + 16 = 31$ yang kira-kira sama dengan 2 n Nilai (32). Jika kita amati dengan seksama, kita melakukan operasi dua kali lipat $\log n$ waktu. Sekarang, mari kita menggeneralisasi diskusi. Untuk operasi mendorong n kami dua kali lipat ukuran array $\log n$ waktu. Itu berarti, kita akan memiliki $\log n$ istilah dalam ekspresi di bawah ini. Total waktu $T(n)$ dari serangkaian operasi n dorongan sebanding dengan

$$\begin{aligned} 1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 4 + 2 + 1 \\ &= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right) \\ &= n(2) \approx 2n = O(n) \end{aligned}$$

$T(n)$ adalah $O(n)$ dan waktu perolehan diamortisasi dari operasi push adalah O (1).

```
struct DynArrayStack {
    int top;
    int capacity;
    int *array;
};
struct DynArrayStack *CreateStack(){
    struct DynArrayStack *S = malloc(sizeof(struct DynArrayStack));
    if(!S)
        return NULL;
    S→capacity = 1;
    S→top = -1;
    S→array = malloc(S→capacity * sizeof(int));  // allocate an array of size 1 initially
    if(!S→array)
        return NULL;
    return S;
}
int IsFullStack(struct DynArrayStack *S){
    return (S→top == S→capacity-1);
}
void DoubleStack(struct DynArrayStack *S){
    S→capacity *= 2;
    S→array = realloc(S→array, S→capacity * sizeof(int));
}
void Push(struct DynArrayStack *S, int x){
    // No overflow in this implementation
     if(IsFullStack(S))
        DoubleStack(S);
    S→array[++S→top] = x;
}
int IsEmptyStack(struct DynArrayStack *S){
    return S→top == -1;
}
int Top(struct DynArrayStack *S){
    if(IsEmptyStack(S))
        return INT_MIN;
    return S→array[S→top];
}
int Pop(struct DynArrayStack *S){
    if(IsEmptyStack(S))
        return INT_MIN;
    return S→array[S→top--];
}
void DeleteStack(struct DynArrayStack *S){
    if(S) {
        if(S→array)
            free(S→array);
        free(S);
    }
}
```

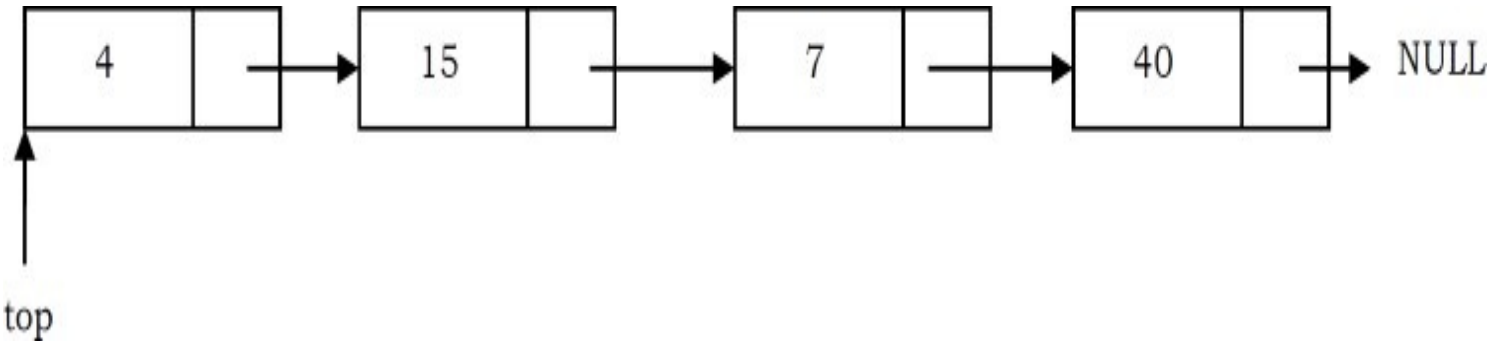
prestasi

Membiarkan *n* menjadi jumlah elemen dalam stack. Kompleksitas untuk operasi dengan representasi ini dapat diberikan sebagai:

Ruang Kompleksitas (untuk *n* mendorong operasi)	HAI(*n)*
Waktu Kompleksitas CreateStack ()	O (1)
Waktu Kompleksitas PushQ	O (1) (rata-rata)
Waktu Kompleksitas PopQ	O (1)
Waktu Kompleksitas Top ()	O (1)
Waktu Kompleksitas IsEmpryStackf)	O (1))
Waktu Kompleksitas IsFullStackf)	O (1)
Waktu Kompleksitas DeleteStackQ	O (1)

catatan: Terlalu banyak doubling dapat menyebabkan overflow memori pengecualian.

Linked Pelaksanaan Daftar



Cara lain menerapkan tumpukan adalah dengan menggunakan daftar Linked. operasi push diimplementasikan dengan memasukkan elemen pada awal daftar. Operasi pop diimplementasikan dengan menghapus node dari awal (header / atas node).

```
struct ListNode{
    int data;
    struct ListNode *next;
};
struct Stack *CreateStack(){
    return NULL;
}
void Push(struct Stack **top, int data){
    struct Stack *temp;
    temp = malloc(sizeof(struct Stack));
    if(!temp)
        return NULL;
    temp→data = data;
    temp→next = *top;
    *top = temp;
}
int IsEmptyStack(struct Stack *top){
    return top == NULL;
}
int Pop(struct Stack **top){
    int data;
    struct Stack *temp;
    if(IsEmptyStack(top))
        return INT_MIN;
    temp = *top;
    *top = *top→next;
    data = temp→data;
    free(temp);
    return data;
}
int Top(struct Stack * top){
    if(IsEmptyStack(top))
        return INT_MIN;
    return top→next→data;
}
void DeleteStack(struct Stack **top){
    struct Stack *temp, *p;
    p = *top;
    while( p→next) {
        temp = p→next;
        p→next = temp→next;
        free(temp);
    }
    free(p);
}
```

prestasi

Membiarkan *n* menjadi jumlah elemen dalam stack. Kompleksitas untuk operasi dengan representasi ini dapat diberikan sebagai:

Ruang Kompleksitas (untuk *n* mendorong operasi)	HAI(*n)*
Waktu Kompleksitas CreateStack ()	O (1)
Waktu Kompleksitas Push ()	O (1) (rata-rata)
Waktu Kompleksitas Pop ()	O (1)
Waktu Kompleksitas Top ()	O (1)
Waktu Kompleksitas IsEmptyStack ()	O (1)
Waktu Kompleksitas DeleteStack ()	HAI(*n)*

4.6 Perbandingan Implementasi

Membandingkan Incremental Strategi dan Strategi Menggandakan

Kami membandingkan strategi tambahan dan strategi dua kali lipat dengan menganalisis total waktu *T (n)* diperlukan untuk melakukan serangkaian operasi n push. Kita mulai dengan stack kosong diwakili oleh sebuah array berukuran 1. Kami menyebutnya *diamortisasi* waktu operasi push adalah waktu rata-rata yang diambil oleh dorongan atas serangkaian operasi, yaitu, *T (n) / n.*

Strategi Incremental

Waktu diamortisasi (waktu rata-rata per operasi) dari operasi push adalah O (*n) [*HAI(*n* 2) / *n].*

menggandakan Strategi

Dalam metode ini, waktu perolehan diamortisasi dari operasi push adalah O (1) [O (*n) / n].*

catatan: Untuk analisis, merujuk pada *Penerapan* bagian.

Membandingkan Pelaksanaan Array dan Linked Pelaksanaan Daftar

Implementasi Array

- Operasi mengambil waktu yang konstan.
- Mahal operasi dua kali lipat setiap sekali-sekali.
- Setiap urutan operasi n (mulai dari tumpukan kosong) - *"Diamortisasi"* terikat membutuhkan waktu sebanding dengan n.

Linked Pelaksanaan Daftar

- Tumbuh dan menyusut anggun.
- Setiap operasi membutuhkan konstan waktu O (1).
- Setiap operasi menggunakan ruang dan waktu untuk berurusan dengan referensi tambahan.

4.7 Stacks: Masalah & Solusi Masalah-1 Diskusikan bagaimana tumpukan dapat digunakan untuk

memeriksa balancing simbol.

Larutan: Tumpukan dapat digunakan untuk memeriksa apakah ekspresi yang diberikan memiliki simbol yang seimbang. Algoritma ini sangat berguna dalam kompiler. Setiap kali parser membaca satu karakter pada satu waktu. Jika karakter adalah pembukaan pembatas seperti (, {, atau [-. Maka ditulis ke stack Ketika pembatas penutupan ditemui seperti),}, atau] -the stack muncul.

Pembukaan dan penutupan pembatas kemudian dibandingkan. Jika mereka cocok, parsing string terus. Jika mereka tidak cocok, parser menunjukkan bahwa ada kesalahan pada baris. Sebuah linear-waktu O (*n)* algoritma berdasarkan pada stack dapat diberikan sebagai:

algoritma:

a) Buat stack.

b) sementara (akhir masukan tidak tercapai) {

1) Jika membaca karakter bukanlah simbol harus seimbang, mengabaikannya.
2) Jika karakter adalah simbol pembuka seperti (, [, {, mendorongnya ke stack
3) Jika itu adalah simbol penutupan seperti),],}, maka jika tumpukan laporan kosong kesalahan. Jika tidak pop stack.
4) Jika simbol Muncul tidak simbol pembukaan yang sesuai, melaporkan kesalahan. }

c) Pada akhir masukan, jika stack tidak kosong laporan kesalahan

contoh: