Apa Itu Struktur Data?

Struktur data adalah **cara menyimpan dan mengelola data di dalam komputer**. Hal ini tentu penting, karena kamu akan banyak berkutat dengan data saat menggunakan atau membangun suatu program.
Pada dasarnya, ada beberapa struktur data yang umum digunakan, yaitu:

- **Array**
- **Linked List**
- **Queue**
- **Stack**
- **Binary Tree**
- **Binary Search Tree**
- **Heap**
- **Graph**

Masing-masing memiliki fungsi dan cara penggunaan yang berbeda-beda. Untuk lebih jelasnya, yuk kita lihat pembahasan yang lebih detail di bagian selanjutnya.

Baca juga: Belajar Bahasa Pemrograman Dasar Untuk Pemula

7+ Jenis Struktur Data yang Umum Digunakan

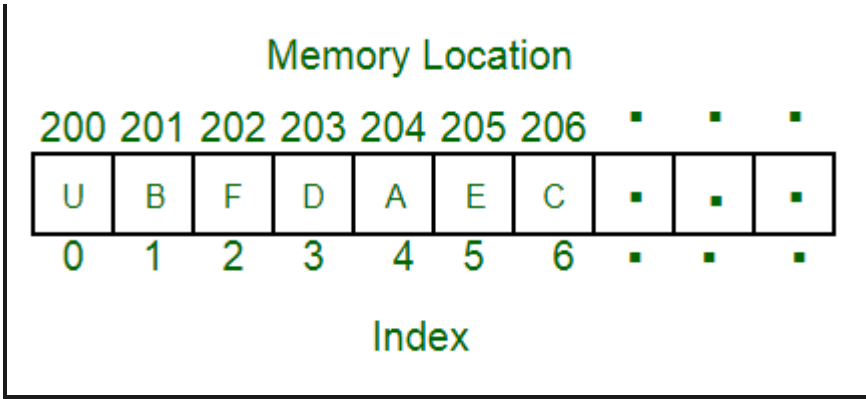
Berikut adalah 7+ struktur data yang perlu kamu pahami saat belajar pemrograman:

1. Array

Array berarti susunan. Sesuai dengan namanya, array adalah struktur data yang disusun secara linear dan berdekatan.

Nah, data yang disusun biasanya memiliki jenis yang sama. Dengan begitu, pengguna data dapat dengan mudah menyortir data berdasarkan tipenya.

Jika digambarkan, berikut adalah ilustrasi array:

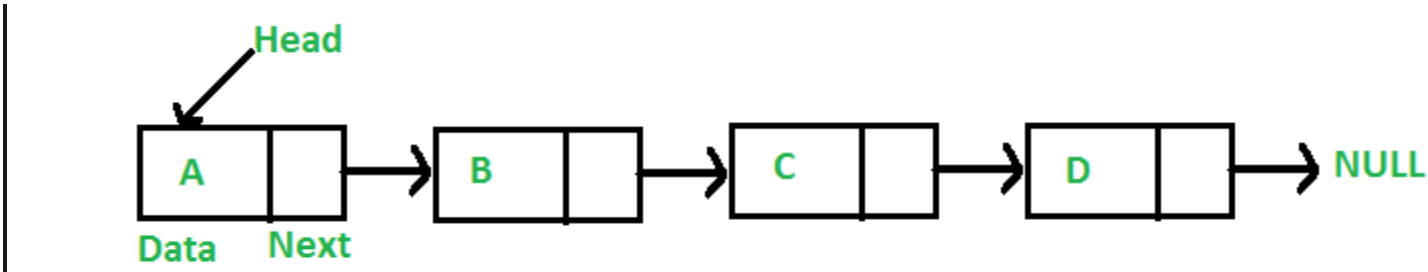


sumber: geeksforgeeks

2. Linked Lists

Sama seperti array, linked lists adalah struktur data yang bersifat linear. Bedanya, linked lists tidak disusun secara berdekatan.

Lalu, bagaimana cara agar setiap datanya terhubung? Jadi, setiap **data**–alias *node*–terhubung melalui *pointer*.
Untuk mempermudah pemahaman, berikut adalah ilustrasi linked lists:



sumber: geeksforgeeks

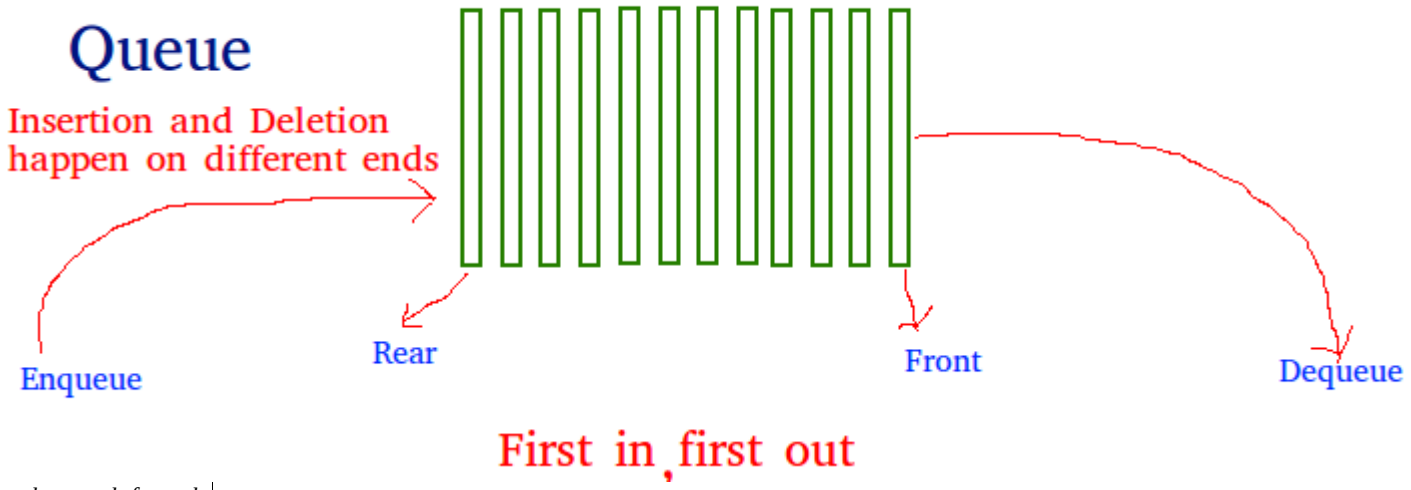
Seperti yang kamu lihat, masing-masing linked list terdiri dari **data** dan **pointer** yang mengarah ke data selanjutnya.

3. Queue

Secara harfiah, arti dari queue adalah antrian. Mengapa disebut begitu? Karena queue adalah struktur data linear yang cara kerjanya sama seperti antrian.

Jadi, data di queue tersusun dalam suatu urutan. Dan data yang diproses duluan adalah data yang pertama kali masuk ke dalam urutannya.

Dengan kata lain,queue menggunakan sistem FIFO (*First In First Out* alias data pertama yang masuk adalah data yang pertama keluar). Ilustrasinya seperti ini:

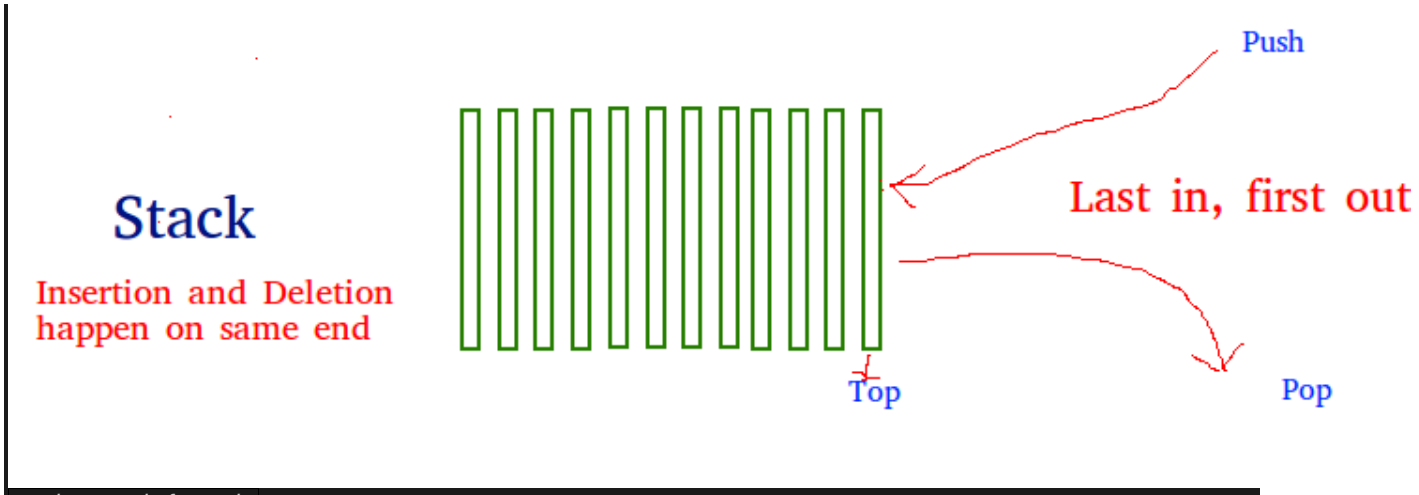


sumber: geeksforgeeks

4. Stack

Stack adalah kebalikan dari queue. Jadi, struktur data ini menggunakan sistem LIFO (*Last In First Out* alias data yang terakhir masuk adalah data yang pertama dikeluarkan).
Selain FILO, stack juga kadang disebut menggunakan sistem FILO (*First In Last Out*, jadi data yang pertama masuk adalah data yang terakhir keluar).
Meskipun pada dasarnya sama saja sih seperti LIFO.
Walau begitu, stack masih punya kesamaan dengan queue. Karena keduanya sama-sama bersifat linear.

Berikut adalah contoh ilustrasi stack. Seperti yang kamu lihat, data yang pertama masuk adalah data yang paling lama menunggu untuk diproses:

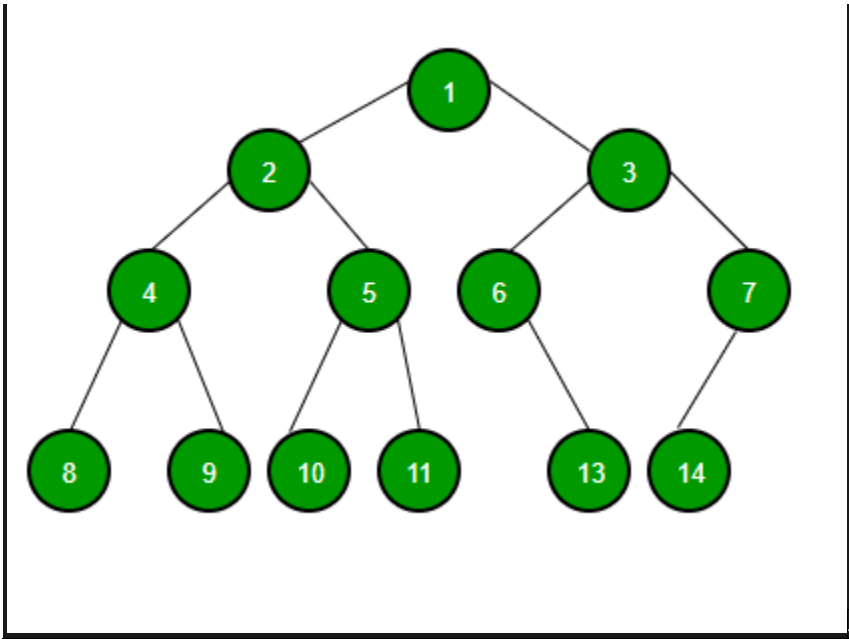


sumber: geeksforgeeks

5. Binary Tree

Binary tree adalah struktur data yang disusun dalam bentuk hierarki. Jadi, setiap titik data akan mengalami percabangan. Dan setiap titik maksimal hanya punya dua cabang.

Hubungan antara titik data dengan kedua cabangnya disambungkan dengan pointer. Ilustrasinya seperti ini:



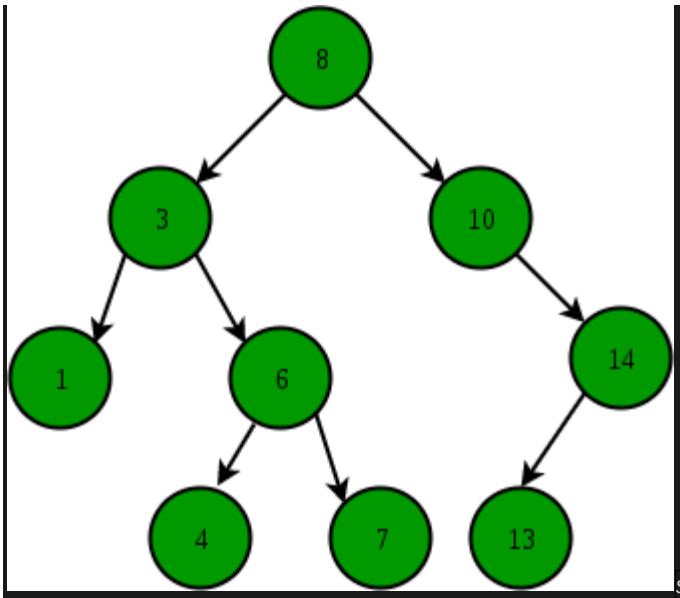
sumber: geeksforgeeks

6. Binary Search Tree

Binary search tree adalah salah satu jenis binary tree. Bentuknya pun kurang lebih sama. Bedanya, binary search tree **menentukan nilai dari setiap percabangan data**.

Mari kita ambil contoh, misalkan data utama memiliki cabang di **sisi kiri** dan **kanan**. Maka, data di **cabang kiri harus punya nilai yang lebih kecil** dari data utama. Sebaliknya, data di **cabang kanan mesti punya nilai yang lebih besar** dibanding data utama.

Aturan ini berlaku juga untuk cabang-cabang di bawahnya. Jadi, makin ke bawah cabang di sebelah kiri nilainya akan semakin berkurang, sedangkan cabang di sebelah kanan makin ke bawah akan semakin meningkat nilainya.



sumber: geeksforgeeks

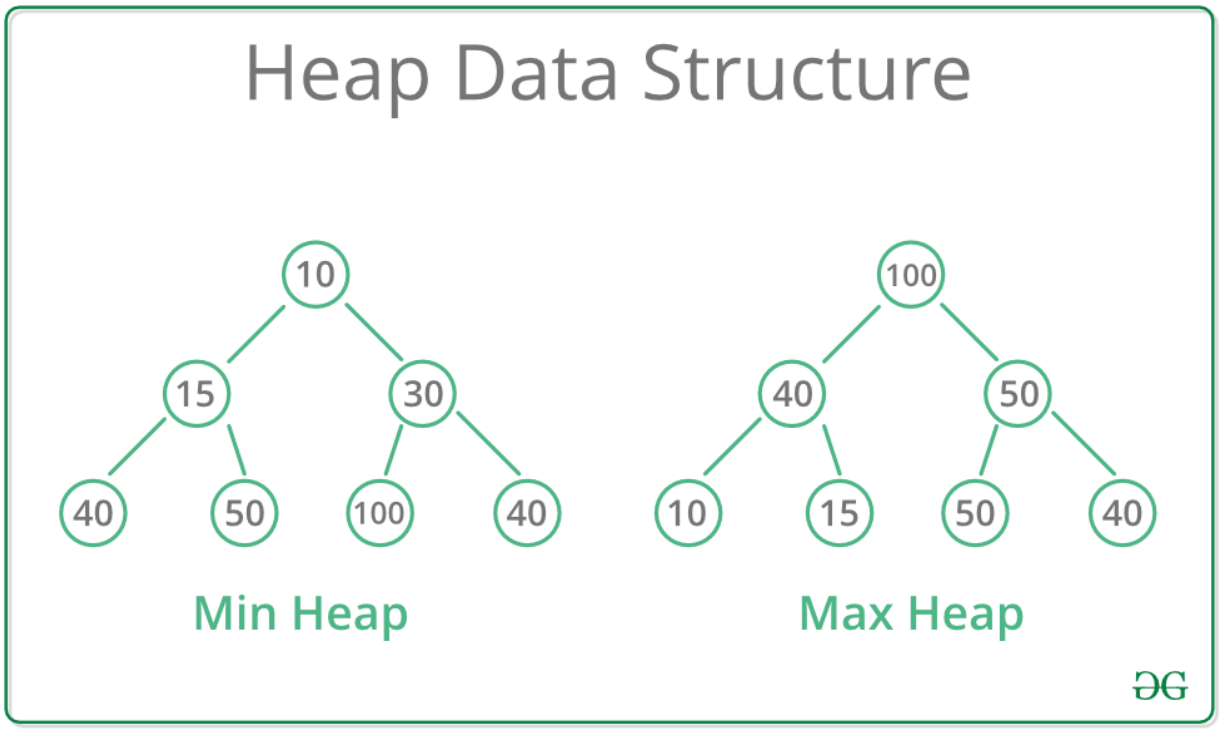
7. Heap

Heap merupakan struktur data yang bentuknya sama juga seperti binary tree. Perbedaannya terletak dari aturan nilai datanya.

Jadi, ada dua jenis struktur heap, yaitu **max heap** dan **min heap**.

- **Max heap** merupakan struktur data di mana nilai data utama harus lebih tinggi dibanding cabang-cabangnya.
- Sebaliknya, **Min heap** adalah menempatkan nilai terendah pada data utamanya. Jadi, makin ke bawah, nilai data cabangnya akan semakin tinggi.

Berikut adalah ilustrasi perbedaan max heap dengan min heap:

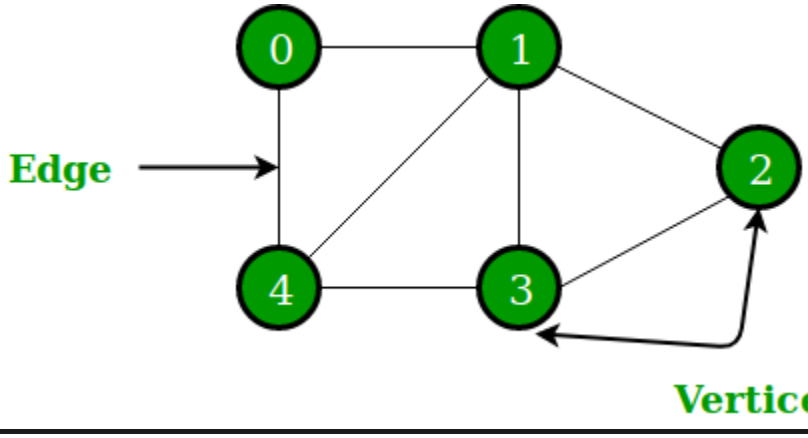


sumber: geeksforgeeks

8. Graph

Graph adalah struktur data yang bersifat non-linear. Jadi, setiap data bisa terhubung secara bebas.

Di linked lists, data biasanya disebut *nodes*, sedangkan di graph biasanya disebut *vertices*. Sedangkan *pointer* di linked lists biasanya disebut *edges* di graph.
Berikut adalah penggambaran hubungan non-linear di dalam graph:



sumber: geeksforgeeks

Dan berikut versi sederhana dari representasi data yang akan kita gunakan. Kita akan menggunakan JavaScript sebagai contoh. Dapat juga diaplikasikan ke bahasa pemrograman lain

```
const hotels = [
{ price: 180, brand: "Hotel Tugu Lombok" },
{ price: 78, brand: "Sheraton Senggigi Beach Resort" },
...
...
{ price: 317, brand: "The Oberio" }
]
```

Pengulangan di Dalam Pengulangan: $O_{(N^2)}$

Sekarang, bagaimana caranya mencari harga terendah dan tertinggi? Cara yang paling naif adalah dengan membandingkan harga satu-per-satu. Berikut potongan kode sebagai ilustrasi.

```
for (let i = 0; i < hotels.length; i++) {
  for (let j = 0; j < hotels.length; j++) {
    // kode untuk membandingkan satu harga dengan harga lainnya...
  }
}
```

Mari kita simulasikan ketika jumlah data semakin banyak.

N	**3**	**5**	**10**	**100**
Jumlah Operasi	9	25	100	10000

Lihat tabel diatas, antara 3 dan 5 cukup dekat perbedaannya hanya dua. Namun jumlah operasi yang dijalankan sangat signifikan bedanya. Semakin banyak datanya semakin signifikan jumlah operasi yang dijalankan.

Dengan kata lain kita membutuhkan n^2 dan untuk mencari harga terendah (min) dan harga tertinggi (max). Itulah sebabnya kita menyebutnya dengan notasi $\mathbf{O_{(n^2)}}$. Algoritma seperti ini sangatlah lambat dan tidak optimal.

Pengulangan Dari Sebuah Set: $O_{(N)}$

Jika solusi pertama tidak optimal, adakah solusi lain? Tentu. Misalnya kita bisa mencari apakah harga adalah yang terendah atau tertinggi dalam satu kali pengulangan saja seperti ilustrasi kode berikut.

```
for (let i = 0; i < hotels.length; i++) {
  // cari harga paling kecil...
  // cari harga paling besar...
}
```

Ketika kita membutuhkan proses "pengulangan untuk setiap item", merupakan notasi **$O_{(N)}$**.

Hanya Satu Operasi: $O_{(1)}$

Jika dengan asumsi bahwa data `hotels` sudah diurutkan berdasarkan harga terendah ke harga tertinggi, maka pencarian harga terendah dan tertinggi menjadi semakin efisien.

```
const hotels = [
  { price: 78, brand: "Sheraton Senggigi Beach Resort" },
  { price: 180, brand: "Hotel Tugu Lombok" },
  ...
  ...
  { price: 317, brand: "The Oberio" }
]
```

Untuk mendapatkan harga terendah, kita tinggal memanggil `hotels[0].price`. Sementara untuk harga tertinggi kita bisa menggunakan `hotels[hotels.length-1].price`. Selain lebih mudah, eksekusi baris kode juga menjadi lebih cepat dan efisien.

Ketika kode menjalankan sebuah operasi, misalkan: *baris kode untuk mencari item barang teratas* atau *mendapatkan nilai sebuah array dengan indeks ke-5*. Operasi seperti ini kita bisa sebut sebagai notasi $O_{(1)}$.

Tidak masalah sebanyak apapun isi dari sebuah array atau sebanyak apapun jumlah baris di basis data, untuk mengambil nilai array dengan mengakses indeks maka algoritma berjalan secara konstan atau *constant time*.

Kita sudah melihat beberapa contoh algoritma untuk menampilkan kisaran harga dari yang terendah hingga tertinggi dan hasilnya sebagai berikut.

Notasi O	Istilah Lain	Jumlah Operasi	Algoritma
$O_{(n^2)}$	Quadratic	n^2	Komparasi seluruh harga. Pengulangan dalam pengulangan
$O_{(n)}$	Linear	2n	Mencari harga terendah dan tertinggi. 1 kali pengulangan
$O_{(1)}$	Constant	2	Asumsi sudah diurut berdasarkan harga, tinggal mencari elemen pertama dan elemen terakhir

Dan secara umum berikut urutan dari notasi O besar diurutkan dari yang tercepat hingga yang terpelan.

<--- Super Cepat ---------- Super Lambat --------->

Nama	**Constant**	**Logaritmic**	**Linear**	**Quadratic**	**Exponential**
Notasi	$O_{(1)}$	$O_{(\log n)}$	$O_{(n)}$	$O_{(n^2)}$	$O_{(n^n)}$

Beberapa Contoh Notasi O Besar

Mari kita melihat contoh notasi O besar dari fungsi, ekspresi dan operasi JavaScript yang sederhana.

1. Array.push()

push() merupakan sebuah metode untuk menambahkan item baru kedalam sebuah array. Item yang ditambahkan akan berada diakhir array tersebut. Contoh penggunaan dapat dilihat sebagai berikut.

```
const animals = ['ants', 'goats', 'cows'];
animals.push('fish');
console.log(animals); // ['ants', 'goats', 'cows', 'fish']
```

Apakah notasi yang tepat untuk baris kode animals.push('fish');? Karena metode push() tidak peduli dengan seberapa banyak atau sedikit jumlah item yang ada, artinya operasi yang berjalan tetap sama, maka metode push() ini dapat diwakilkan dengan notasi $\mathbf{O_{(1)}}$ atau **konstan**.

2. Array.pop()

pop() merupakan sebuah metode yang mengambil item terakhir dari array sehingga jumlah item yang ada di array akan berkurang satu. Berikut contoh penggunaannya.

```
const plants = ['broccoli', 'cauliflower', 'cabbage', 'tomato'];
plants.pop();
console.log(plants); // ["broccoli", "cauliflower", "cabbage"]
```

Apakah notasi yang tepat untuk baris kode plants.pop();? Mirip seperti metode push() diatas, metode pop() juga tidak mempermasalahkan jumlah item yang ada, artinya operasi yang berjalan tetap sama, maka metode pop() ini juga dapat diwakilkan dengan notasi $\mathbf{O_{(1)}}$ atau **konstan**.

3. Array.unshift()

unshift() adalah sebuah metode untuk menambahkan satu atau beberapa item ke bagian awal dari sebuah array. Contoh penggunaannya sebagai berikut.

```
const array1 = [1, 2, 3];
array1.unshift(4, 5);
console.log(array1); // [4, 5, 1, 2, 3]
```

Sekilas operasi unshift() ini terlihat seperti operasi yang konstan seperti push() dan pop() namun jika kita melakukan implementasi ulang metode ini, maka akan terlihat notasi yang sebenarnya. Berikut kira-kira implementasi dari unshift(), implementasi naif tentunya sekedar gambaran.

```
function unshift(arr, newItem) {
  let newArr = [];
  newArr[0] = newItem;
  for (let i = 1; i < arr.length + 1; i++) {
    newArr[i] = arr[i - 1];
  }
  return newArr;
}
```

Hal yang menambah kompleksitas adalah ketika kita harus mengubah indeks dari array karena kita akan menempatkan item baru di indeks ke-0. Secara otomatis indeks akan bergeser sebanyak satu langkah. Dan karena itu kita menggunakan pengulangan for hingga menjadikan operasi unshift() dapat diwakilkan oleh notasi **linear** atau $\mathbf{O_{(n)}}$. Kita harus menyadari apa yang dilakukan oleh sebuah fungsi, operasi ataupun pustaka sehingga kita dapat memprediksi kira-kira seberapa tingkat kompleksitasnya.

Menghitung Total Kompleksitas Kode

Setelah sebelumnya kita sudah menghitung kompleksitas satu baris kode, sekarang kita akan menghitung total keseluruhan untuk beberapa baris kode. Kita akan mulai dari contoh sederhana terlebih dahulu.

```
for (let i = 0; i < 10; i++) {
  // O(n)
  for (let j = 0; j < 10; j++) {
    // O(n)
    console.log(i); // O(1)
    console.log(j); // O(1)
  }
}
```

Kompleksitas potongan kode diatas dapat dihitung dengan mengalikan notasi-notasi setiap baris. Karena terdapat pengulangan bersarang, maka operasi yang digunakan adalah perkalian. Artinya hasil dari pengulangan bersarang tersebut adalah: $\mathbf{O(n) * O(n) = O_{(n^2)}}$.

Jika perintah yang berada di level yang sama seperti:

```
console.log(i); // O(1)
console.log(j); // O(1)
```

Maka operasi yang akan kita lakukan adalah menjumlahkan. Sehingga untuk kedua perintah diatas hasilnya adalah $\mathbf{O_{(1)} + O_{(1)} = O_{(2)}}$. Jika digabungkan hasilnya menjadi $\mathbf{O_{(n^2)}}$ $\mathbf{O_{(2)}}$. Namun biasanya untuk jenis kode diatas cukup dilambangkan dengan $\mathbf{O_{(n^2)}}$ _ $\mathbf{O_{(2)}}$ karena $\mathbf{O_{(2)}}$ tidak signifikan perbedaannya.

Mari kita lihat contoh kode berikutnya.

```
for (let i = 0; i < 10; i++) {
  // O(n)
  console.log(i);
}

for (let j = 0; j < 10; j++) {
  // O(n)
  console.log(j);
}
```

Potongan kode deiatas dapat dilambangkan sebagai $\mathbf{O_{(n)}}$ ditambah dengan $\mathbf{O_{(n)}}$ sehingga menjadi $\mathbf{O_{(2n)}}$.

Kesimpulan

Sebagai kesimpulan, notasi O besar atau *Big-O Notation* merupakan metode untuk menghitung kompleksitas dari potongan kode yang kita buat. Sehingga dapat menumbuhkan kesadaran kita untuk mencari alternatif yang lebih baik sebelum data semakin besar dan berdampak kepada performa aplikasi yang kita buat.

Notasi O besar inipun tidak hanya semata berlaku di bagian kode yang kita tulis, namun di *database* pun berlaku. Jadi proses pengambilan data di database dengan sintaksis SQL yang dapat dianggap sebagai proses perulangan akan sangat tidak efektif jika di bagian algoritma kode kita kembali menggunakan perulangan. Maka proses tersebut akan menjadi $\mathbf{O_{(n^2)}}$ karena akan terdapat perulangan didalam perulangan.

Begitu juga halnya jika kita mengambil data dengan tabel yang sudah diindeks. Secara otomatis notasinya akan berubah dari $\mathbf{O_{(n)}}$ menjadi $\mathbf{O_{(\log n)}}$.

```
dvdrental=# EXPLAIN ANALYZE SELECT * FROM film WHERE title='Academy Dinosaur';
EXPLAIN ANALYZE SELECT * FROM film WHERE title='Academy Dinosaur';
                                             QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Seq Scan on film  (cost=0.00..66.50 rows=1 width=384) (actual time=0.031..0.505 rows=1 loops=1)
   Filter: ((title)::text = 'Academy Dinosaur'::text)
   Rows Removed by Filter: 999
 Planning time: 1.558 ms
 Execution time: 0.605 ms
(5 rows)
```

Tanpa Index

Tanpa menggunakan indeks database melakukan yang disebut dengan "Sequential Scan". Beberapa yang lain menyebutnya dengan istilah "Full Table Scan" yang kurang-lebih melakukan perulangan setiap barisnya dan membandingkan dengan argumen query yang kita tentukan.

Dengan kata lain operasi *sequential* seperti contoh diatas dapat kita beri notasi $\mathbf{O_{(n)}}$. Seiring bertambahnya jumlah data, efisiensi akan semakin menurun.

Hal termudah untuk mengangkat performa untuk kasus ini adalah dengan menambahkan indeks di tabel terkait dan kita dapat melihat perbedaan yang cukup signifikan terutama ketika data sudah semakin banyak.

```
dvdrental=# CREATE INDEX idx_title on film(title);
CREATE INDEX idx_title on film(title);
CREATE INDEX
dvdrental=# EXPLAIN ANALYZE SELECT * FROM film WHERE title='Academy Dinosaur';
EXPLAIN ANALYZE SELECT * FROM film WHERE title='Academy Dinosaur';
WARNING: terminal is not fully functional
-  (press RETURN)
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Index Scan using idx_title on film  (cost=0.28..8.29 rows=1 width=384) (actual time=0.068..0.069 rows=1
loop   Index Cond: ((title)::text = 'Academy Dinosaur'::text)
 Planning time: 0.502 ms
 Execution time: 0.109 ms
(4 rows)
```

Dengan Index

Sehingga operasi diatas berubah notasinya menjadi $\mathbf{O_{(\log n)}}$.

Hal yang menarik lainnya, Redis sebuah *in-memory database* menyertakan notasi kompleksitas waktu di setiap perintah yang ada di dokumentasinya. Contohnya dapat dilihat seperti perintah append, del, lpush, dan masih banyak lagi yang lainnya.

Sebagai penutup, berikut daftar kompleksitas kode dari operasi-operasi yang umum kita jumpai.

Kompleksitas	**Operasi**
$O_{(1)}$	Menjalankan sebuah perintah
$O_{(1)}$	Mendapatkan sebuah item dari array, objek atau variabel
$O_{(\log n)}$	Pengulangan yang berkurang setengahnya setiap iterasi
$O_{(n^2)}$	Pengulangan dalam pengulangan
$O_{(n^3)}$	Pengulangan dalam pengulangan dalam pengulangan