Struktur Data Dasar

Tim Olimpiade Komputer Indonesia

Pendahuluan

Melalui dokumen ini, kalian akan:

- Mengenal beberapa macam struktur data dasar.
- Mengetahui pentingnya penggunaan struktur data.
- Mengetahui operasi-operasi yang dapat dilakukan pada struktur data dasar.

Tentang Struktur Data

Struktur Data

Merupakan tata cara untuk merepresentasikan dan menyimpan data, sehingga mendukung operasi terhadap data tersebut secara efisien.

Kilas Balik: Array

Array merupakan contoh struktur data dasar yang mendukung operasi berikut:

- Membaca nilai yang disimpan pada suatu indeks sembarang.
- Mengubah nilai yang disimpan pada suatu indeks sembarang.

Akses indeks secara sembarang ini biasa disebut sebagai *random access*.

Kilas Balik: Array (lanj.)

- Bagaimana kalau kita hendak menyisipkan suatu elemen sebagai indeks pertama dari *array*?
- Kita harus menggeser seluruh isi *array*, barulah memasukkan elemen yang hendak dimasukkan di indeks pertama.
- Operasi ini dilaksanakan dalam $O(N)$, dengan N adalah ukuran *array*.

Kilas Balik: Array (lanj.)

- Bagaimana jika operasi ini sering dilakukan?
- Melaksanakannya dalam $O(N)$ kurang efisien.
- Adakah cara yang lebih baik?

Bagian 1

Linked List

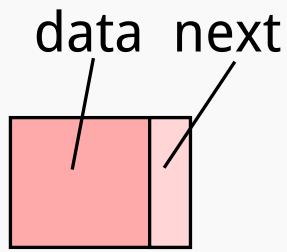
Mengenal Linked List

Linked list terdiri dari kumpulan **node**.

Node dapat diartikan sebagai sebuah titik yang nantinya dapat dihubungkan dengan *node* lainnya.

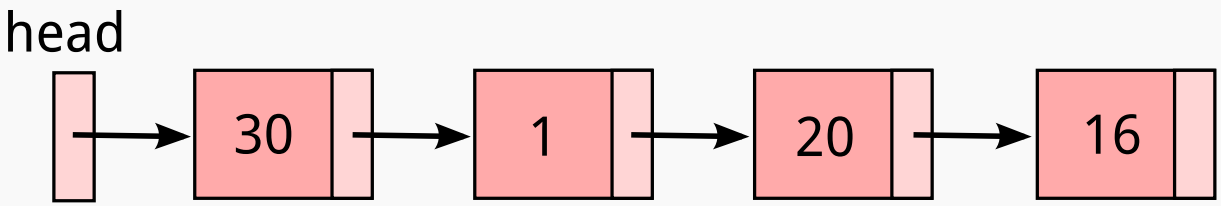
Sebuah *node* menyimpan dua informasi:

1. data: informasi yang disimpan.
2. next: *pointer* ke *node* berikutnya.



Mengenal Linked List (lanj.)

- *Pointer-pointer* ini menghubungkan antar *node* dalam *linked list*.
- *Node* paling depan biasa disebut *head*.



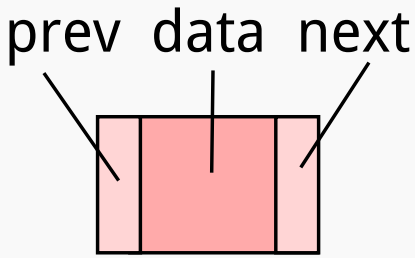
Jenis Linked List

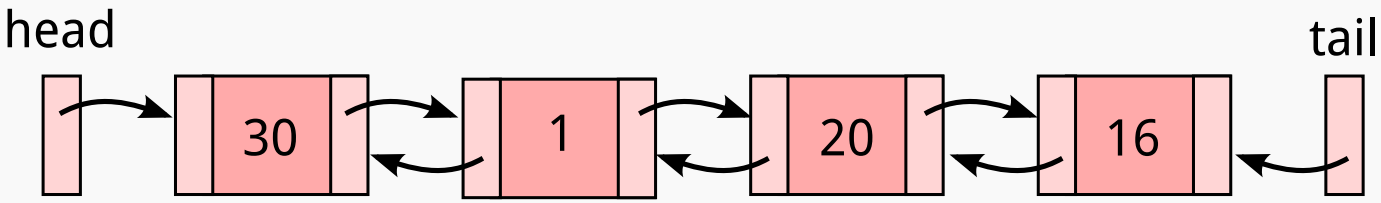
Berdasarkan hubungannya dengan *node* lain, *linked list* terbagi menjadi 2 macam, yaitu:

- *singly linked list*: tiap *node* hanya memiliki *pointer* ke *node* selanjutnya saja (next).
- *doubly linked list*: tiap *node* memiliki *pointer* ke *node* selanjutnya (next) dan *node* sebelumnya (prev).

Pada pembahasan ini, kita akan menggunakan *doubly linked list*.

Struktur Doubly Linked List





Linked List dan Array

- Pada *doubly linked list*, biasanya kita hanya memiliki referensi ke *head* dan *tail*.
- Untuk mengakses elemen ke-x dari *linked list*, kita dapat melakukannya dengan:

```
GET(head, x)
1  current = head
2  for i = 2 to x
3        current = current.next
4  return current
```

- Terlihat tidak efisien?

Linked List dan Array (lanj.)

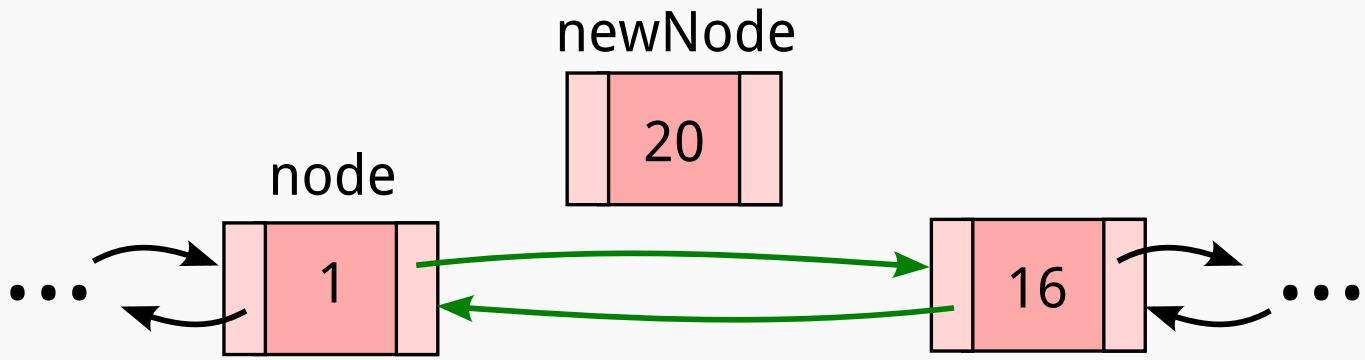
Doubly linked list memiliki keuntungan dalam:

- Menyisipkan elemen baru.
- Menghapus suatu elemen.

Kedua operasi tersebut dapat dilakukan secara efisien.

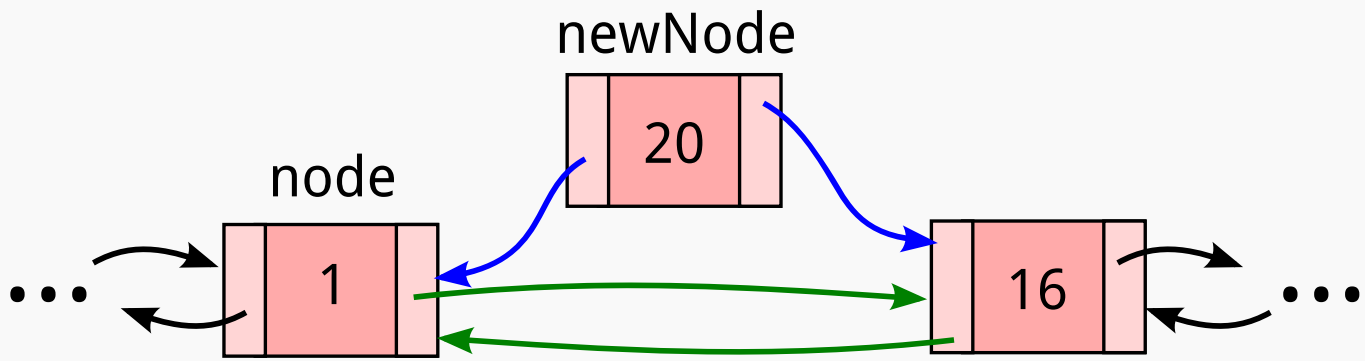
Menyisipkan Elemen Linked List

Diberikan *node* yang bukan *tail*, sisipkan *newNode* sesudahnya.



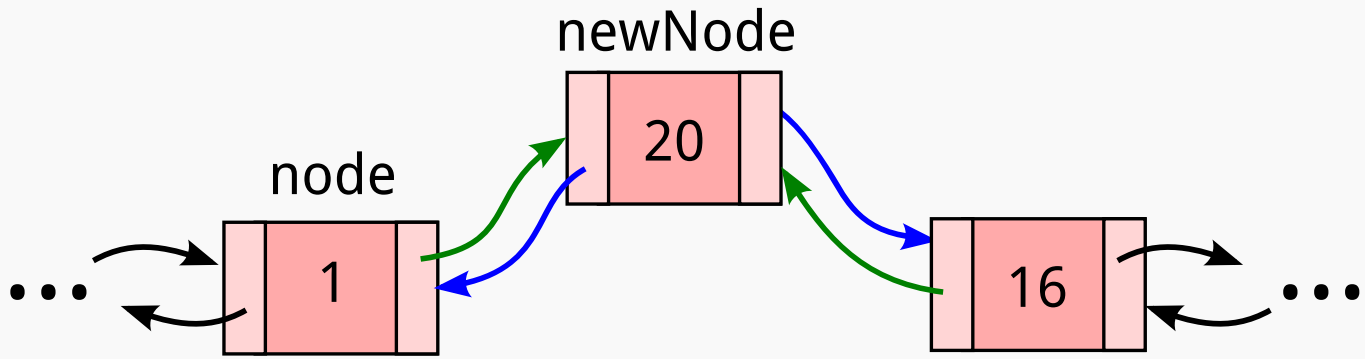
Menyisipkan Elemen Linked List (lanj.)

1. Isi *pointer newNode.prev* untuk mengarah ke *node*.
2. Isi *pointer newNode.next* untuk mengarah ke *node.next*.



Menyisipkan Elemen Linked List (lanj.)

3. Perbaiki *pointer* *newNode.prev.next* untuk mengarah ke *newNode*.
4. Perbaiki *pointer* *newNode.next.prev* untuk mengarah ke *newNode*.

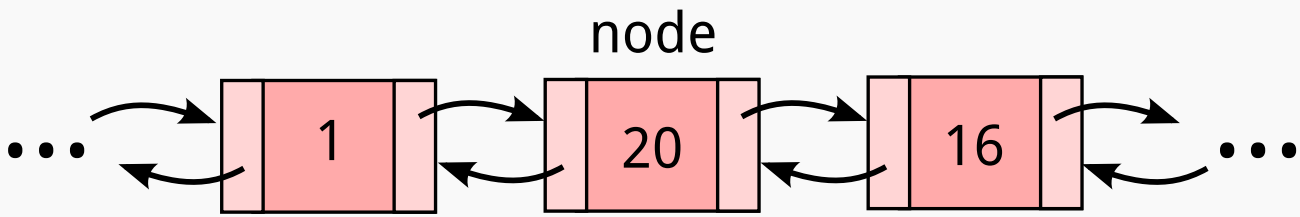


Menyisipkan Elemen Linked List (lanj.)

- Untuk menyisipkan elemen di paling awal atau paling akhir, gunakan cara serupa.
- Tidak ada pergeseran, hanya "cabut" dan "pasang" *pointer*.
- Kompleksitasnya adalah $O(1)$.

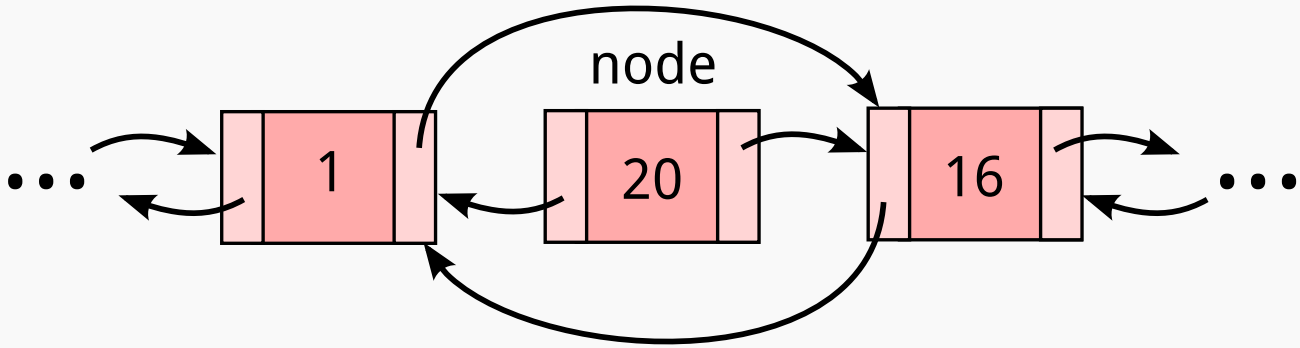
Menghapus Elemen Linked List

Diberikan *node* yang bukan *head* maupun *tail*, hapus dari *linked list*.



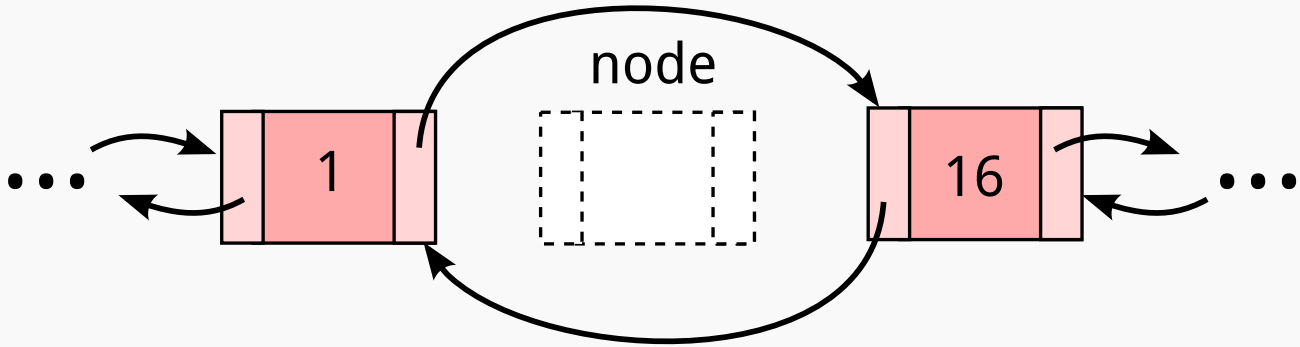
Menghapus Elemen Linked List (lanj.)

1. Ubah *node.prev.next* menjadi *node.next*.
2. Ubah *node.next.prev* menjadi *node.prev*.



Menghapus Elemen Linked List (lanj.)

3. Hapus *node*.



Menghapus Elemen Linked List (lanj.)

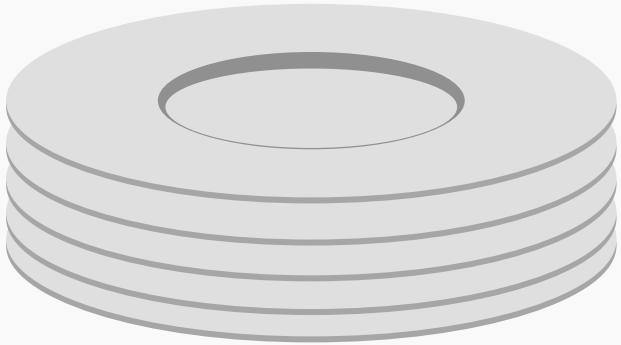
- Anda juga dapat menghapus *head* atau *tail* dengan cara serupa.
- Sama seperti menyisipkan elemen, tidak ada operasi pergeseran di sini.
- Kompleksitasnya adalah $O(1)$.

Rangkuman

- *Linked list* memang tidak mendukung *random access* secara efisien.
- Namun *linked list* mendukung operasi menyisipkan dan menghapus jika diketahui *node* tempat penyisipan/penghapusan dilakukan.

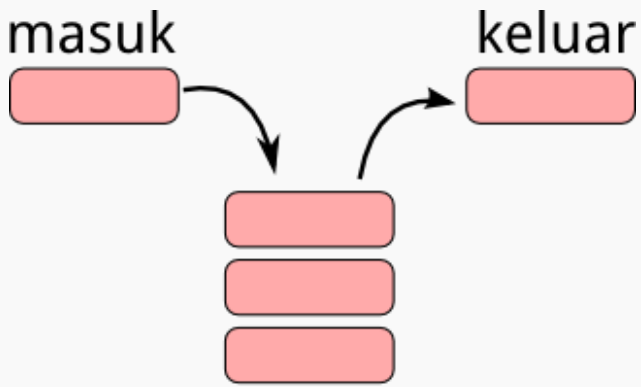
Mengenal Stack

- *Stack* dapat dimisalkan seperti tumpukan piring pada umumnya.
- Jika terdapat piring baru yang ingin dimasukkan, maka piring tersebut masuk dari paling atas.
- Jika sebuah piring akan diambil dari tumpukan, maka yang diambil juga piring yang paling atas.



Mengenal Stack (lanj.)

- Struktur data *stack* menyimpan informasi dalam bentuk tumpukan.
- Informasi yang baru dimasukkan ke paling atas tumpukan.
- Hanya informasi paling atas yang bisa diakses/dihapus pada setiap waktunya.
- Oleh karena itu struktur data *stack* disebut memiliki sifat LIFO (Last In First Out).



Operasi pada Stack

Stack memiliki operasi sebagai berikut:

- *push*, yaitu memasukkan elemen baru ke bagian atas tumpukan.
- *pop*, yaitu membuang elemen paling atas tumpukan.
- *top*, yaitu mengakses elemen paling atas tumpukan.

Aplikasi Stack

- Eksekusi fungsi pada sejumlah bahasa pemrograman biasanya menggunakan struktur data *stack*, tertama untuk fungsi rekursif.
- Pemanggilan fungsi rekursif yang menambah kedalaman berarti melakukan "*push*" pada *stack* eksekusi fungsi.
- Fungsi yang dieksekusi adalah fungsi di paling atas *stack*.
- Setelah fungsi di paling atas selesai, dilakukan "*pop*" dan eksekusi fungsi dilanjutkan ke fungsi yang ada di paling atas *stack* berikutnya.
- Oleh sebab itu, ketika pemanggilan fungsi terlalu dalam, terjadi stack overflow.

Aplikasi Stack Lainnya

- *Stack* juga digunakan pada kalkulator ekspresi matematika dalam notasi *postfix*.
- Notasi *postfix* adalah notasi penulisan ekspresi matematika dengan urutan operand, operand, dan operator.
- Contoh:
  - "1 2 +" bermakna "$1 + 2$"
  - "1 2 3 + −" bermakna "$1 - (2 + 3)$"
  - "1 2 3 + − 4 ×" bermakna "$(1 - (2 + 3)) \times 4$"
- Notasi yang biasa kita gunakan adalah notasi *infix*, yaitu dengan urutan operand, operator, dan operand.

Aplikasi Stack Lainnya (lanj.)

- Diberikan sebuah ekspresi dalam notasi *postfix*, nilai akhirnya dapat dicari dengan skema kerja *stack*.
- Pada awalnya, inisialisasi sebuah *stack* kosong.
- Proses ekspresi dari kiri ke kanan:
  1. Jika ditemukan operand, *push* ke dalam *stack*.
  2. Jika ditemukan operator, *pop* dua kali untuk mendapat dua operand teratas *stack*, hitung, lalu *push* kembali ke dalam *stack*.
- Satu-satunya nilai terakhir di dalam *stack* adalah hasil ekspresinya.

Eksekusi Ekspresi Postfix

Ekspresi: $1\ 2\ 3\ +\ -\ 4\ \times$

1. *Push* angka 1, *stack*: [1].
2. *Push* angka 2, *stack*: [1, 2].
3. *Push* angka 3, *stack*: [1, 2, 3].
4. Ditemukan $+$:
   - *Pop* dua kali, didapat nilai 2 dan 3, *stack*: [1].
   - Operasikan $2 + 3$, dan *push*, *stack*: [1, 5]
5. ...

Eksekusi Ekspresi Postfix (lanj.)

Ekspresi: 1 2 3 + − 4 ×

5. Ditemukan -:
   - *Pop* dua kali, didapat nilai 1 dan 5, *stack*: [].
   - Operasikan 1 - 5, dan *push*, *stack*: [-4]
6. *Push* angka 4, *stack*: [-4, 4].
7. Ditemukan ×:
   - *Pop* dua kali, didapat nilai -4 dan 4, *stack*: [].
   - Operasikan -4 × 4, dan *push*, *stack*: [-16]

Jadi 1 2 3 + − 4 × = −16

Implementasi Stack

- Anda dapat mengimplementasikan *stack* menggunakan *singly linked list*.
- *Node* yang perlu Anda simpan cukup *head* saja (atau *tail* saja), berhubung penyisipan/penghapusan selalu dilakukan di bagian tersebut.

Alternatif Implementasi Stack

- Alternatif yang seringkali lebih mudah adalah menggunakan sebuah *array* dan variabel penunjuk.
- Variabel penunjuk ini menyatakan indeks *array* yang menjadi elemen paling atas *stack*, dan bergerak maju/mundur sesuai dengan perintah *push*/*pop*.
- Seluruh operasi dapat dilakukan dalam $O(1)$.

Alternatif Implementasi Stack (lanj.)

INITIALIZESTACK(*maxSize*)

```
1  // Buat array stack berukuran maxSize
2  topOfStack = 0
```

PUSH(*item*)

```
1  topOfStack = topOfStack + 1
2  stack[topOfStack] = item
```

POP()

```
1  topOfStack = topOfStack − 1
```

TOP()

```
1  return stack[topOfStack]
```

Alternatif Implementasi Stack (lanj.)

- Pastikan nilai *maxSize* sama dengan maksimal operasi *push* yang mungkin dilakukan.
- Pada operasi *pop*, kita tidak benar-benar menghapus elemennya, melainkan hanya variabel penunjuknya yang "dimundurkan".

Bagian 2

Stack

Contoh Soal

- Anda diberikan sebuah string, misalnya `acaabcbcd`.
- Cari string `abc` dalam string tersebut. Jika ditemukan maka hapus string `abc` tersebut, lalu ulangi pencarian.
- Pencarian berakhir ketika tidak terdapat string `abc` lagi.
- Tentukan total penghapusan yang berhasil dilakukan
- Contoh, pada string `acaabcbcd` terdapat sebuah string `abc`, dan hapus string tersebut menjadi `acabcd`. Lalu, ditemukan lagi string `abc` dan hapus menjadi `acd`. Karena tidak ditemukan lagi string `abc`, maka jawabannya adalah 2.

Pembahasan Soal

- Lakukan iterasi setiap karakter pada string tersebut.
- Untuk setiap karakter, *push* ke dalam *stack*.
- Cek 3 karakter teratas pada *stack*.
- Jika 3 karakter teratas merupakan `abc`, artinya terdapat 1 penghapusan. Lalu *pop* ketiga huruf tersebut dari *stack*.

Pada soal ini, Anda harus dapat memodifikasi struktur data *stack* agar Anda dapat melakukan operasi *top* pada 3 elemen teratas. Kompleksitas total adalah $O(N)$, dengan N merupakan panjang string.

Bagian 3

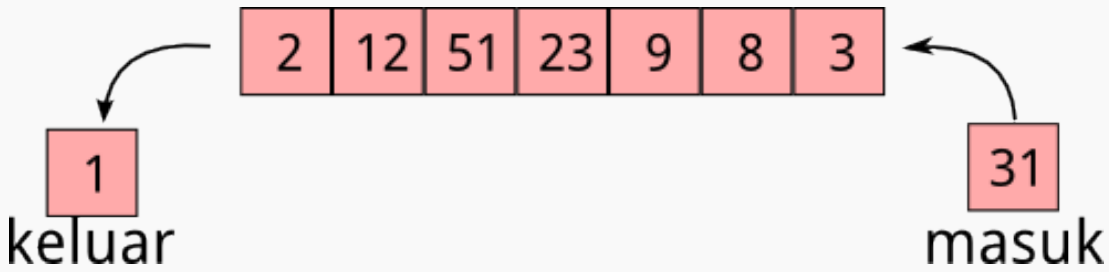
Queue

Mengenal Queue

- Apakah anda pernah melihat antrean pembelian?
- Struktur data *queue* mirip dengan analogi antrean tersebut.
- Saat seorang ingin masuk ke antrean, maka orang tersebut harus mengantri dari belakang.
- Sementara itu, orang yang dilayani terlebih dahulu adalah orang yang paling depan.

Mengenal Queue (lanj.)

- Struktur data *queue* menyimpan informasi dalam bentuk antrean.
- Informasi yang baru dimasukkan ke paling belakang antrean.
- Hanya informasi paling depan yang bisa diakses/dihapus pada setiap waktunya.
- Oleh karena itu struktur data *queue* disebut memiliki sifat FIFO (First In First Out).



Operasi Queue

Queue memiliki beberapa operasi yang dapat dilakukan:

- *push*, yaitu memasukkan elemen baru ke bagian akhir antrean.
- *pop*, yaitu mengeluarkan elemen paling depan antrean.
- *front*, yaitu mengakses elemen yang paling depan antrean.

Aplikasi Queue

- Sesuai namanya, pada komputer *queue* digunakan untuk berbagai hal yang memerlukan antrean.
- Misalnya antrean berkas-berkas yang akan diunduh dari internet untuk ditampilkan pada *browser* Anda.
- *Queue* akan sering kita gunakan ketika sudah memasuki materi graf.
- Tepatnya ketika melakukan *breadth-first search*.

Implementasi Queue

- Anda dapat mengimplementasikan *queue* menggunakan *singly linked list*.
- Anda dapat menyimpan *node head* dan *tail*.
- Setiap *push*, sisipkan elemen sesudah *tail*.
- Setiap *pop*, hapus *node head*.

Alternatif Implementasi Queue

- Lagi-lagi, alternatif yang seringkali lebih mudah adalah menggunakan sebuah *array* dan dua variabel penunjuk.
- Variabel penunjuk ini menyatakan indeks *array* yang menjadi elemen paling depan dan belakang *queue*.
- Kedua variabel penunjuk ini selalu bergerak maju.
- Seluruh operasi dapat dilakukan dalam $O(1)$.

Alternatif Implementasi Queue (lanj.)

INITIALIZEQUEUE(*maxSize*)

```
1  // Buat array queue berukuran maxSize
2  head = 1
3  tail = 0
```

PUSH(*item*)

```
1  tail = tail + 1
2  queue[tail] = item
```

POP()

```
1  head = head + 1
```

FRONT()

```
1  return queue[head]
```

Alternatif Implementasi Queue (lanj.)

- Kelemahan dari implementasi ini adalah beberapa elemen di bagian awal *array* tidak digunakan kembali.
- Misalnya telah dilakukan 15 kali *push*, dan 11 kali *pop*.
- Sebanyak 11 elemen pertama pada *array* tidak akan digunakan kembali.
- Ini adalah pemborosan, karena aslinya hanya terdapat 4 elemen di dalam *queue*.
- Meskipun demikian, dalam dunia kompetisi hal ini masih dapat diterima.
- Pastikan nilai *maxSize* sama dengan maksimal operasi *push* yang mungkin dilakukan.

Penutup

- Struktur data yang baru kita pelajari ini merupakan struktur data dasar.
- Pada lain kesempatan, kita akan mempelajari struktur data yang lebih kompleks dan manfaatnya lebih "berasa", seperti *heap* untuk *priority queue*, *binary search tree* untuk kamus, dan *segment tree* untuk *dynamic range query*.