Struktur Data Non-Linear: Heap

Tim Olimpiade Komputer Indonesia

Pendahuluan

Melalui dokumen ini, kalian akan:

- Mengenal dan mengimplementasikan struktur data *heap*.
- Mengetahui mengapa diperlukan *heap*.

Motivasi

Anda diberikan sejumlah operasi. Setiap operasi dapat berbentuk salah satu dari:

- add(x), artinya simpan bilangan x.
- getMax(), artinya dapatkan bilangan terbesar yang saat ini masih disimpan.
- deleteMax(), artinya hapus bilangan terbesar dari penyimpanan.

Motivasi

Berikut contoh operasinya dan perilaku yang diharapkan:

- add(5), bilangan yang disimpan: [5].
- add(7), bilangan yang disimpan: [5, 7].
- add(3), bilangan yang disimpan: [5, 7, 3].
- getMax(), laporkan bahwa 7 merupakan bilangan terbesar.
- deleteMax(), bilangan yang disimpan: [5, 3].
- getMax(), laporkan bahwa 5 merupakan bilangan terbesar.

Solusi Sederhana

- Solusi paling mudah adalah membuat sebuah *array* besar dan variabel yang menunjukkan posisi terakhir elemen pada *array*.
- Untuk setiap operasi add(x), tambahkan elemen *array*, geser variabel penunjuk, lalu urutkan data.
- Operasi getMax() dapat dilayani dengan mengembalikan elemen terbesar.
- Operasi deleteMax() dapat dilayani dengan menggeser variabel penunjuk.

Analisis Solusi Sederhana

- Misalkan N menyatakan banyaknya elemen pada *array*.
- Dengan cara ini, operasi add(x) berlangsung dalam $O(N \log N)$, apabila pengurutannya menggunakan *quicksort*.
- Operasi getMax() dan deleteMax() berlangsung dalam $O(1)$.

- Perhatikan bahwa pengurutan akan lebih efisien jika digunakan *insertion sort*, sehingga kompleksitas add(x) menjadi $O(N)$.

Analisis Solusi Sederhana

Operasi	Dengan sorting
add(x)	$O(N \log N)$
getMax()	$O(1)$
deleteMax()	$O(1)$

Masalah Solusi Sederhana

- Solusi sederhana ini tidak efisien ketika banyak dilakukan operasi add(x).
- Kita akan mempelajari bagaimana *heap* mengatasi masalah ini secara efisien.

Bagian 1

Pengenalan Heap

Heap

- *Heap* merupakan struktur data yang umum dikenal pada ilmu komputer.
- Nama *heap* sendiri berasal dari Bahasa Inggris, yang berarti "gundukan".

Operasi Heap

Heap mendukung operasi:

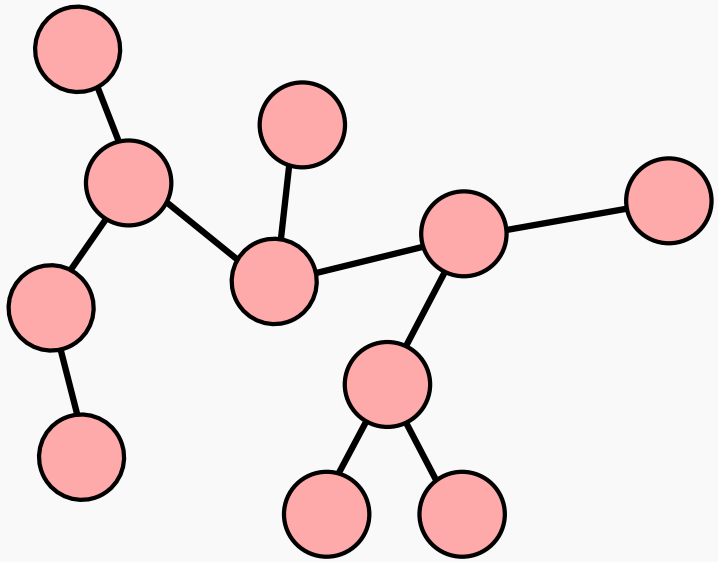
- *push*, yaitu memasukan elemen baru ke penyimpanan.
- *pop*, yaitu membuang elemen **terbesar** dari penyimpanan.
- *top*, yaitu mengakses elemen **terbesar** dari penyimpanan.

Cara Kerja Heap

- *Heap* dapat diimplementasikan dengan berbagai cara.
- Kita akan mempelajari salah satunya, yaitu *binary heap*.
- Sebelum itu, diperlukan pengetahuan mengenai *tree*.

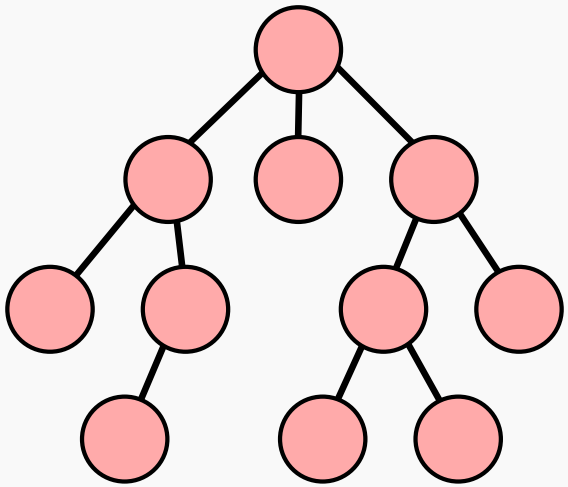
Tree

- Seperti yang telah dipelajari, *tree* merupakan suatu graf yang setiap *node*-nya saling terhubung dan tidak memiliki *cycle*.



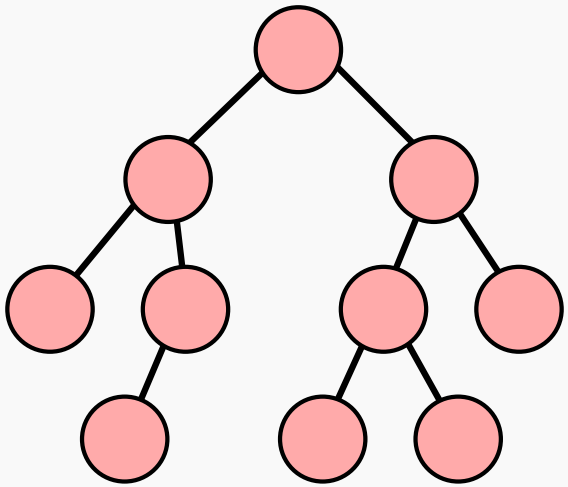
Rooted Tree

- Suatu *tree* yang memiliki hierarki dan memiliki sebuah akar disebut sebagai *rooted tree*.



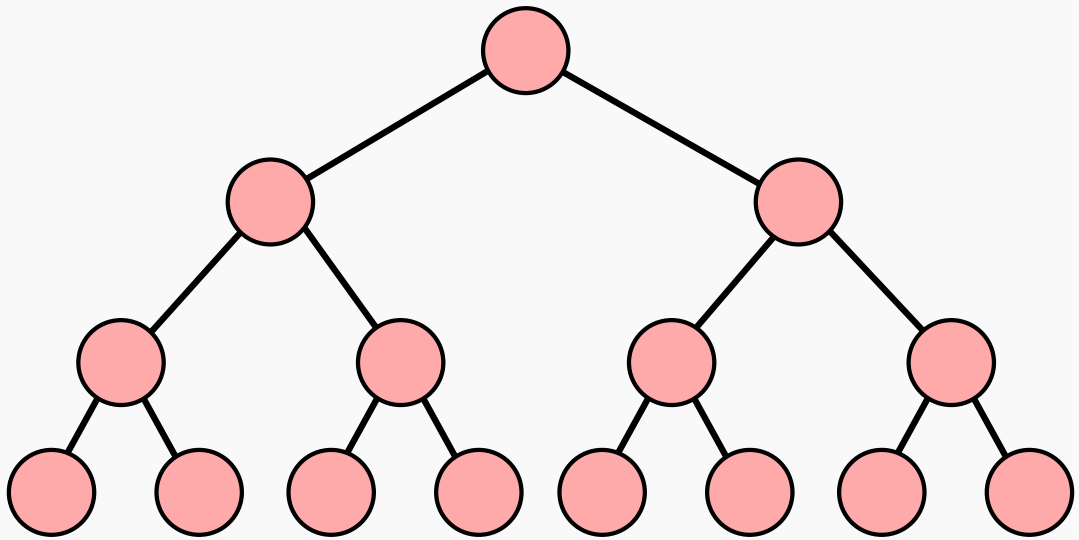
Binary Tree

- Suatu *rooted tree* yang setiap *node*-nya memiliki 0, 1, atau 2 anak disebut dengan *binary tree*.



Full Binary Tree

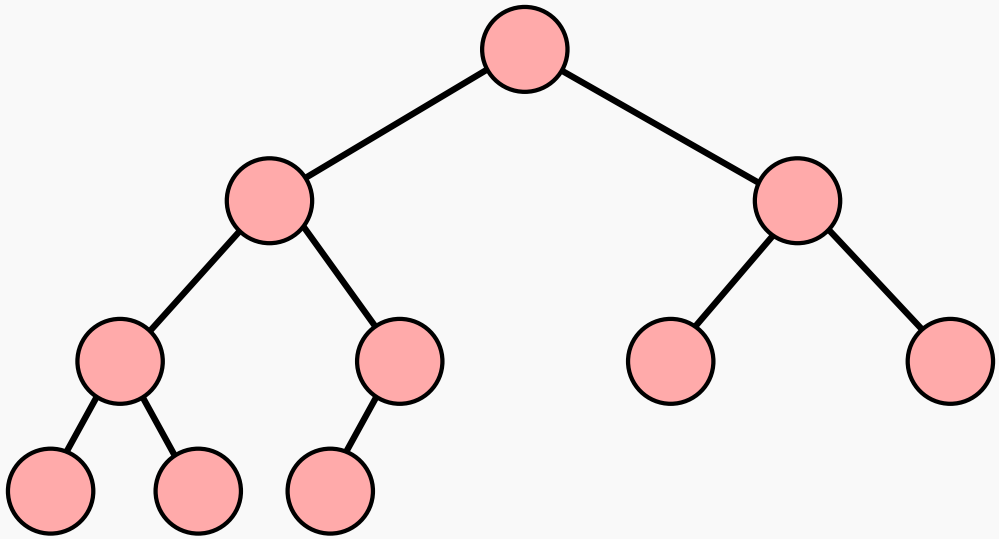
- Suatu *binary tree* yang seluruh *node*-nya memiliki 2 anak, kecuali tingkat paling bawah yang tidak memiliki anak, disebut dengan *full binary Tree*
- Bila banyaknya *node* adalah N, maka ketinggiannya adalah $O(\log N)$.



Complete Binary Tree

Complete binary tree adalah *binary tree* yang:

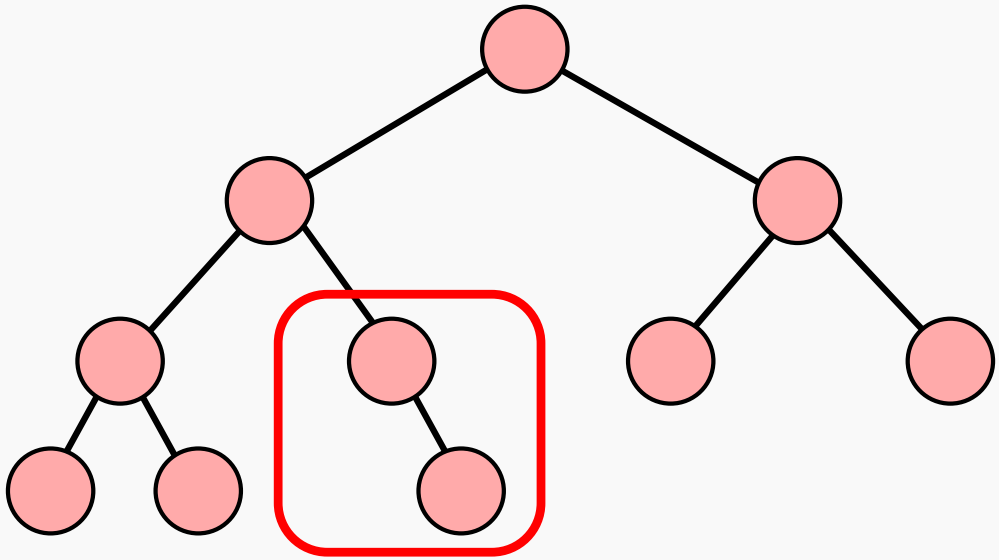
- Seluruh *node*-nya memiliki 2 anak, kecuali tingkat paling bawah.
- Tingkat paling bawahnya dapat terisi sebagian, tetapi harus terisi dari kiri ke kanan.



- Bila banyaknya *node* adalah N, maka ketinggiannya adalah $O(\log N)$.

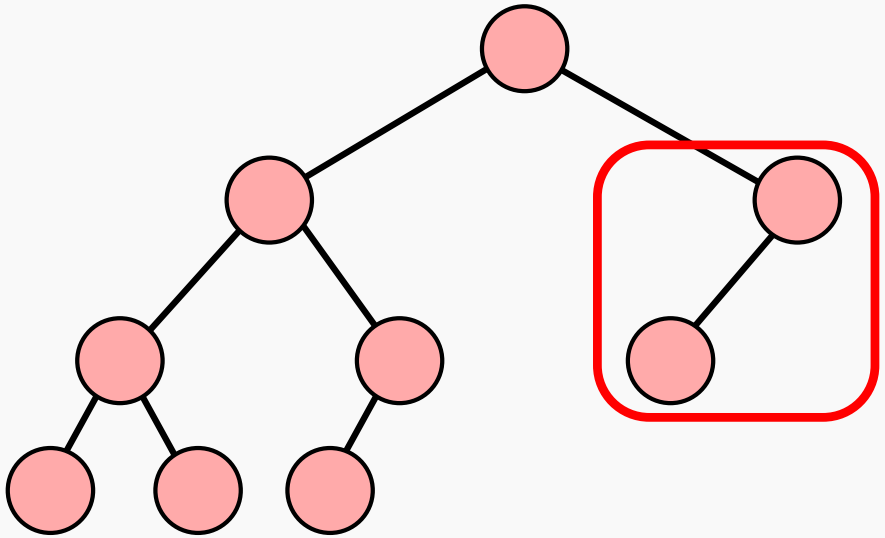
Bukan Complete Binary Tree

Berikut bukan *complete binary tree*, sebab elemen di tingkat paling bawah tidak berisi dari kiri ke kanan (terdapat lubang).



Bukan Complete Binary Tree

Berikut juga bukan *complete binary tree*, sebab terdapat *node* tanpa 2 anak pada tingkat bukan paling bawah.

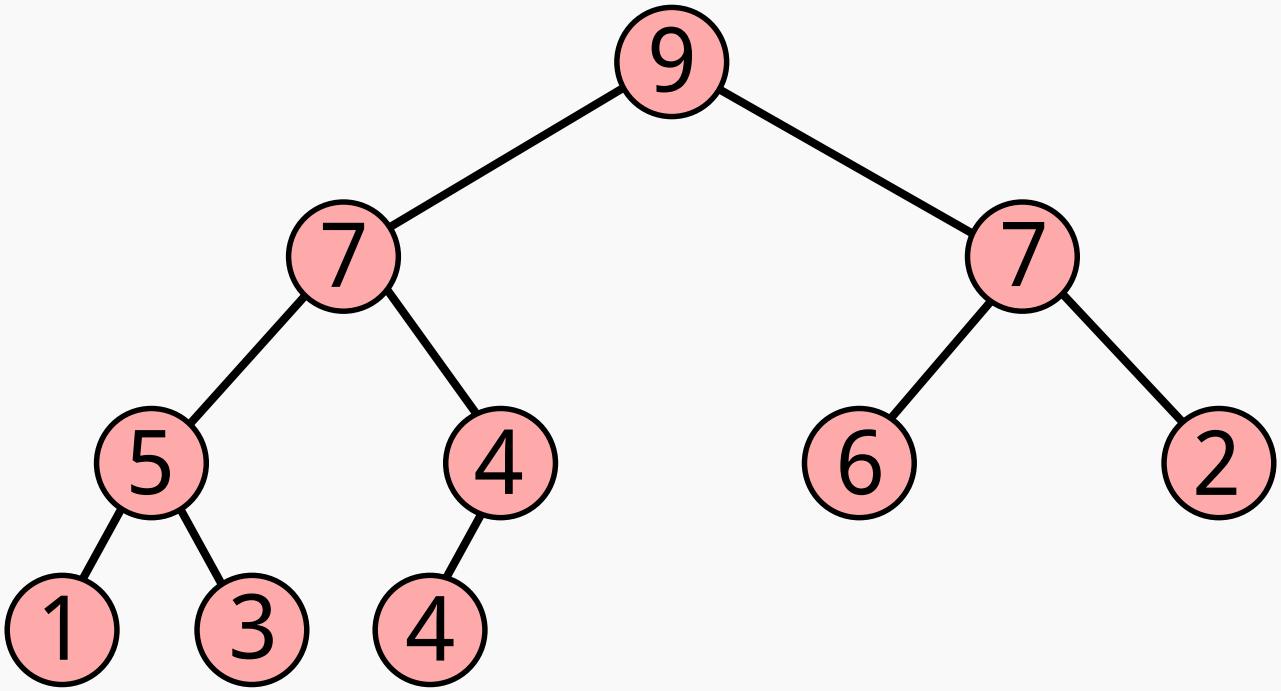


Struktur Binary Heap

Struktur data *binary heap* memiliki sifat:

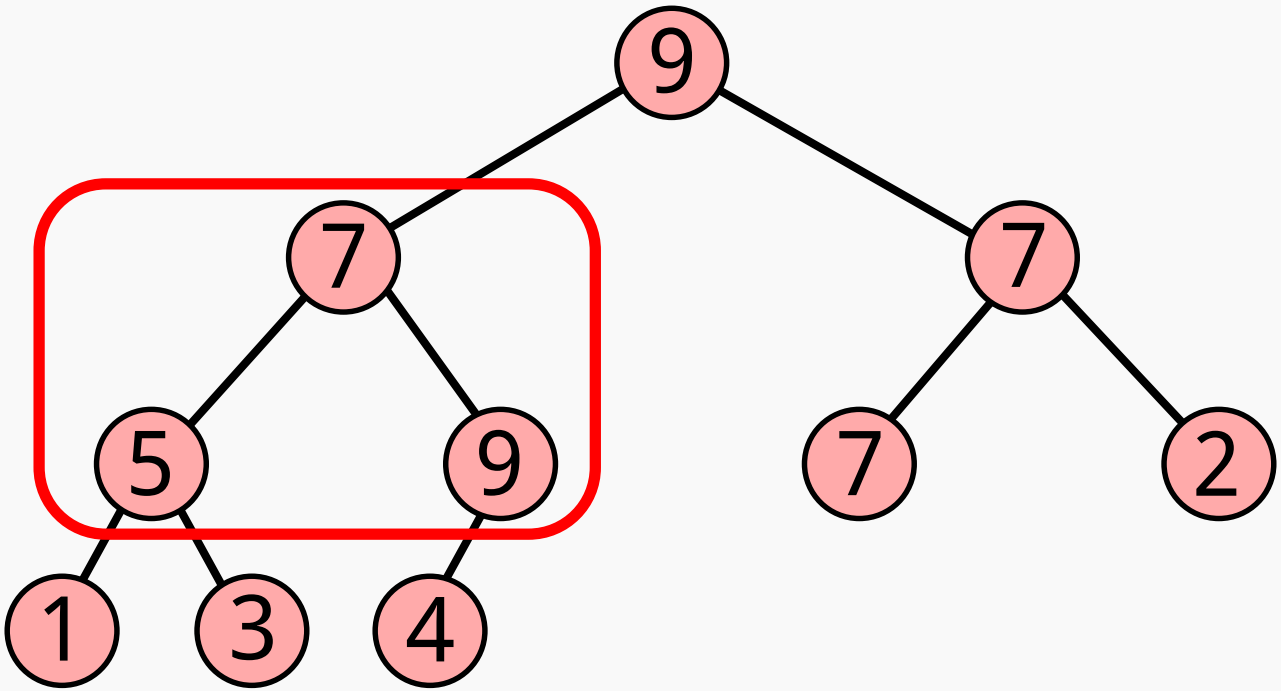
- Berstruktur *complete binary tree*.
- Setiap *node* merepresentasikan elemen yang disimpan pada *heap*.
- Setiap *node* memiliki nilai yang **lebih besar** daripada *node* anak-anaknya.

Contoh Binary Heap



Contoh Bukan Binary Heap

Bukan binary heap.



Mengapa Harus Demikian?

- Struktur seperti ini menjamin operasi-operasi yang dilayani *heap* dapat dilakukan secara efisien.
- Misalkan N adalah banyaknya elemen yang sedang disimpan.
- Operasi *push* dan *pop* bekerja dalam $O(\log N)$, sementara *top* bekerja dalam $O(1)$.
- Kita akan melihat satu persatu bagaimana operasi tersebut dilaksanakan.

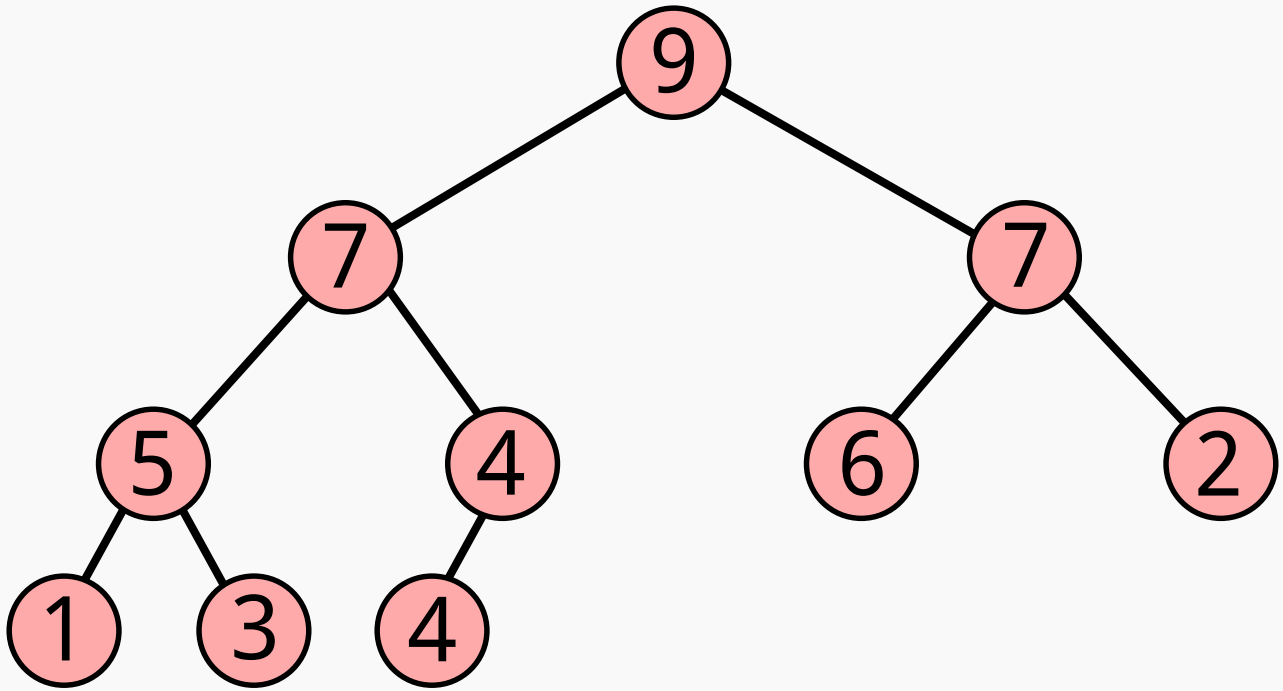
Operasi Push

Melakukan *push* pada *binary heap* dilakukan dengan 2 tahap:

- Tambahkan *node* baru di posisi yang memenuhi aturan *complete binary tree*.
- Selama elemen *node* yang merupakan orang tua langsung dari elemen ini memiliki nilai yang lebih kecil, tukar nilai elemen kedua *node* tersebut.

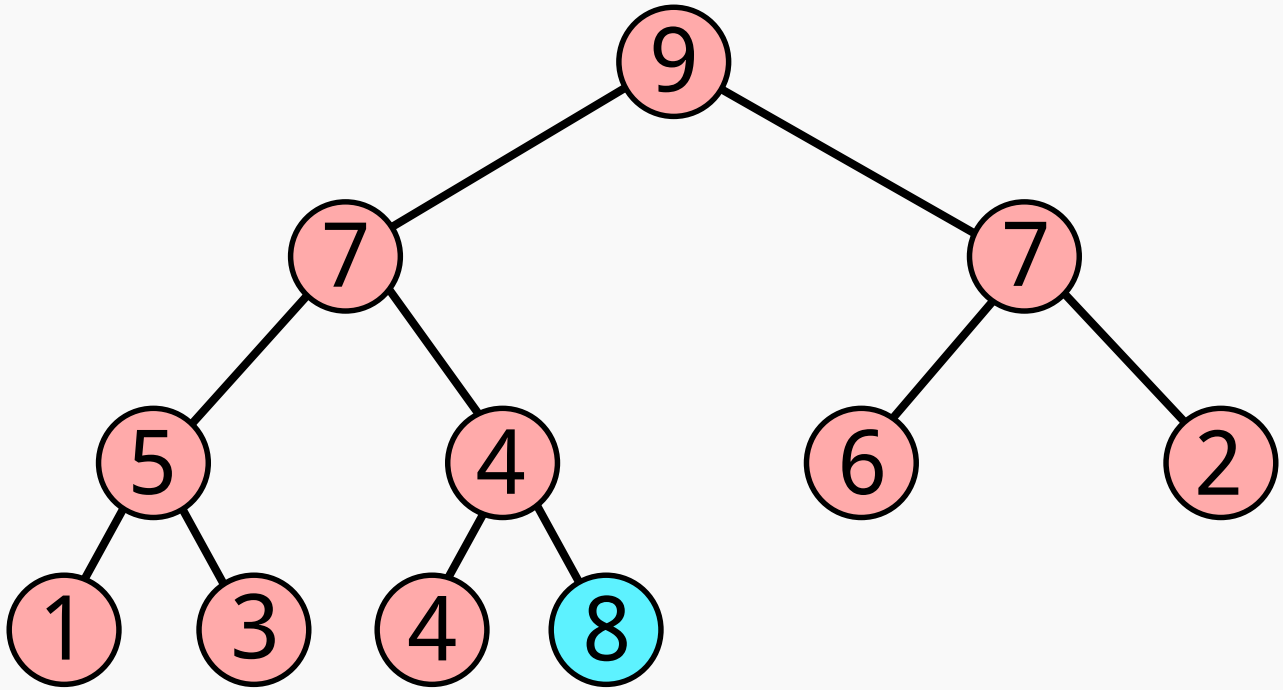
Operasi Push

Sebagai contoh, misalkan hendak ditambahkan elemen bernilai 8 ke suatu *binary heap* berikut:



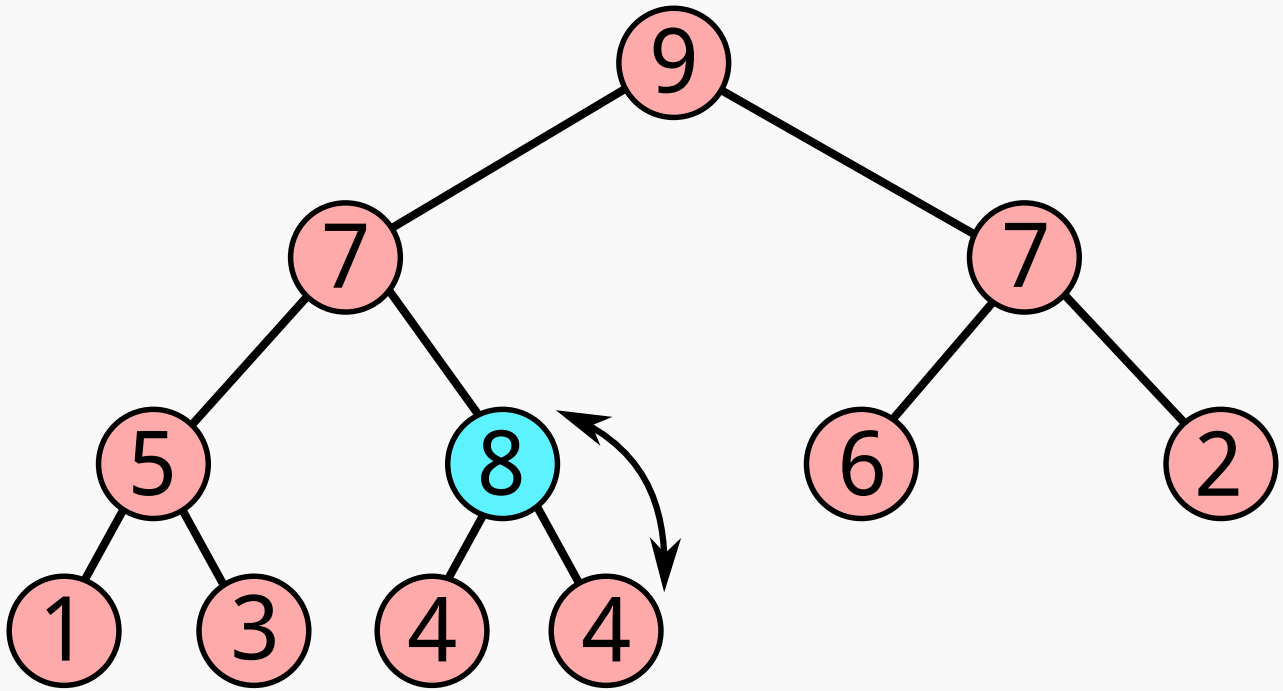
Operasi Push (lanj.)

Tambahkan *node*.



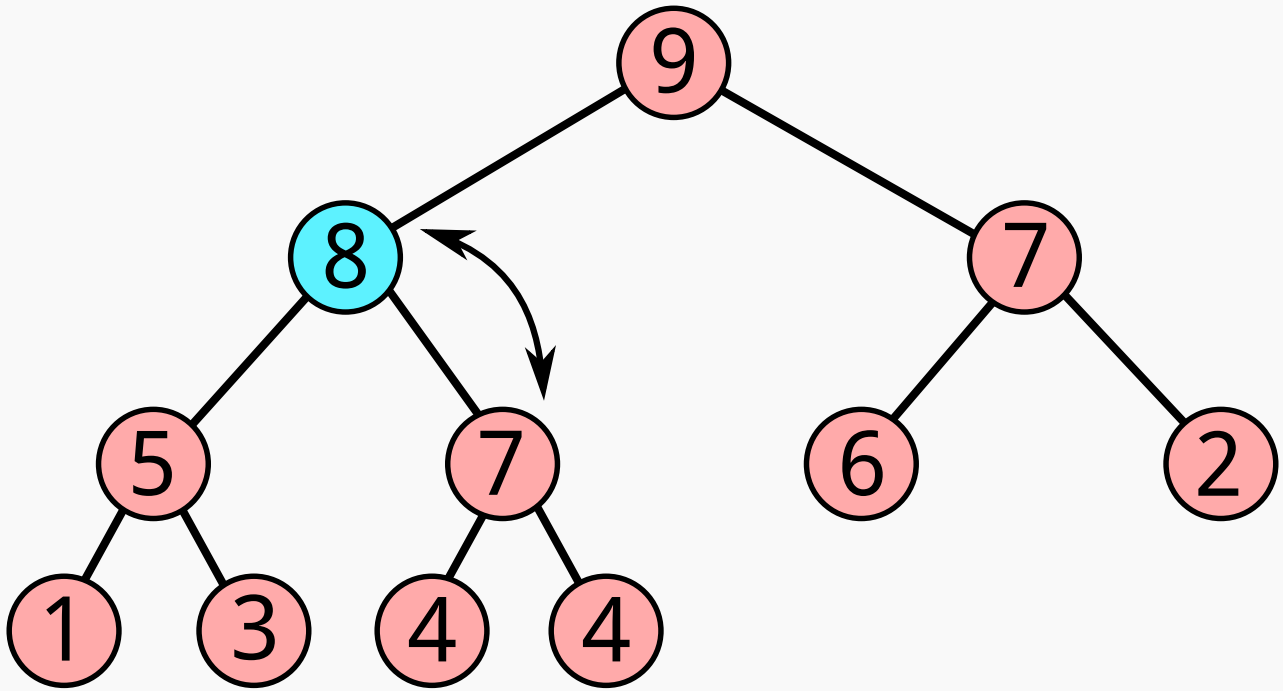
Operasi Push (lanj.)

Karena *parent*-nya memiliki nilai lebih kecil, tukar nilainya.



Operasi Push (lanj.)

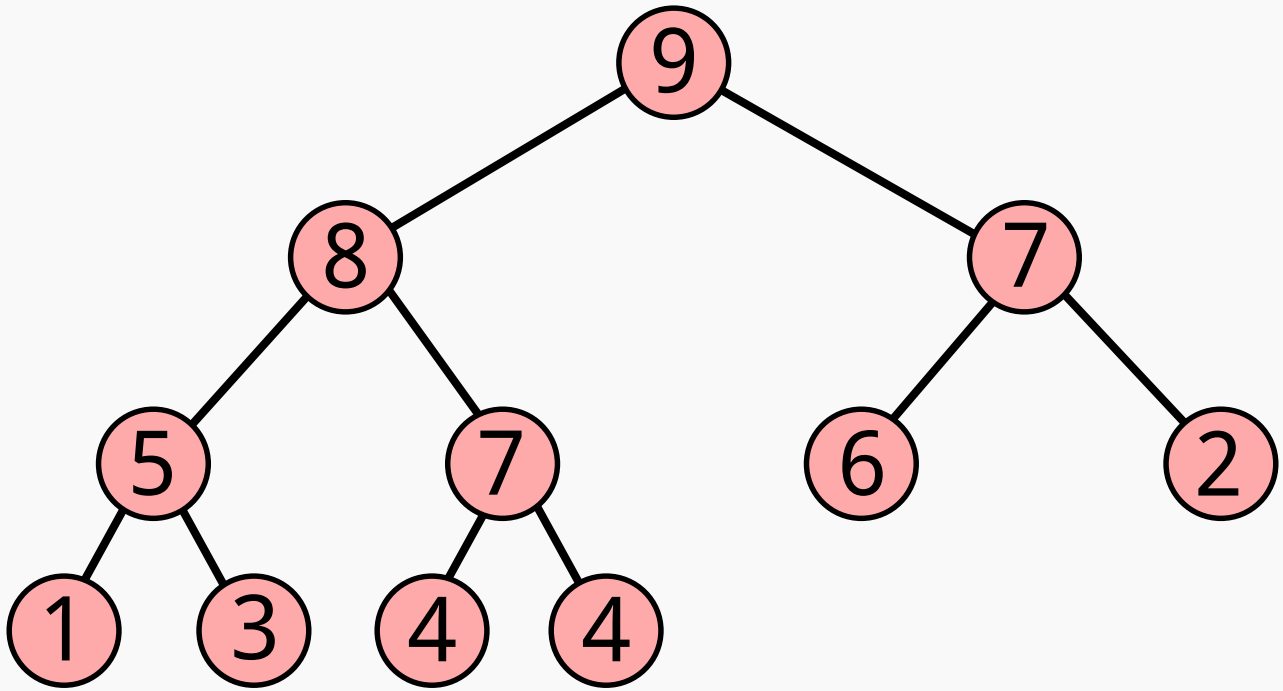
Karena *parent*-nya masih memiliki nilai lebih kecil, tukar lagi.



Operasi Push (lanj.)

Parent-nya sudah memiliki nilai yang lebih besar.

Operasi *push* selesai.



Kompleksitas Push

- Kasus terburuk terjadi ketika pertukaran yang terjadi paling banyak.
- Hal ini terjadi ketika elemen yang dimasukkan merupakan nilai yang paling besar pada *heap*.
- Banyaknya pertukaran yang terjadi sebanding dengan kedalaman dari *complete binary tree*.
- Kompleksitas untuk operasi *push* adalah $O(\log N)$.

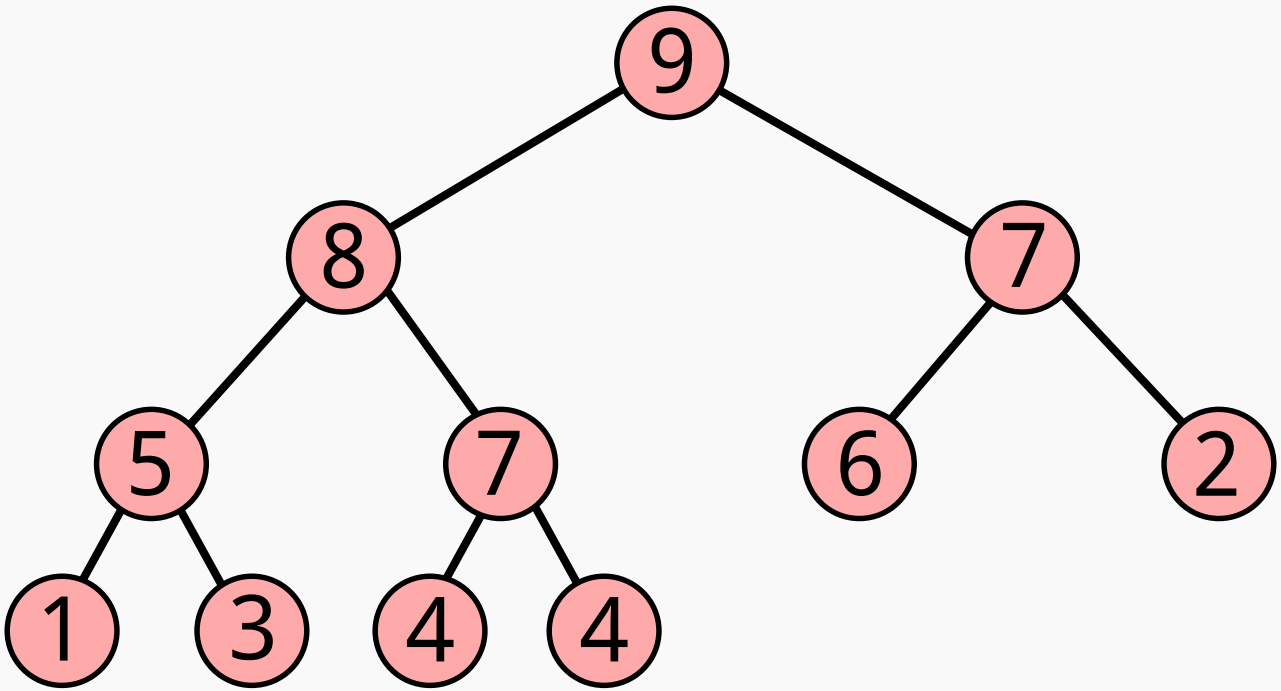
Operasi Pop

Melakukan *pop* pada *binary heap* dilakukan dengan 3 tahap:

- Tukar posisi elemen pada *root* dengan elemen terakhir mengikuti aturan *complete binary tree*.
- Buang elemen terakhir *binary heap*, yang telah berisi elemen dari *root*.
- Selama elemen yang ditukar ke posisi *root* memiliki anak langsung yang berelemen lebih besar, tukar elemen tersebut dengan salah anaknya yang memiliki elemen **terbesar**.

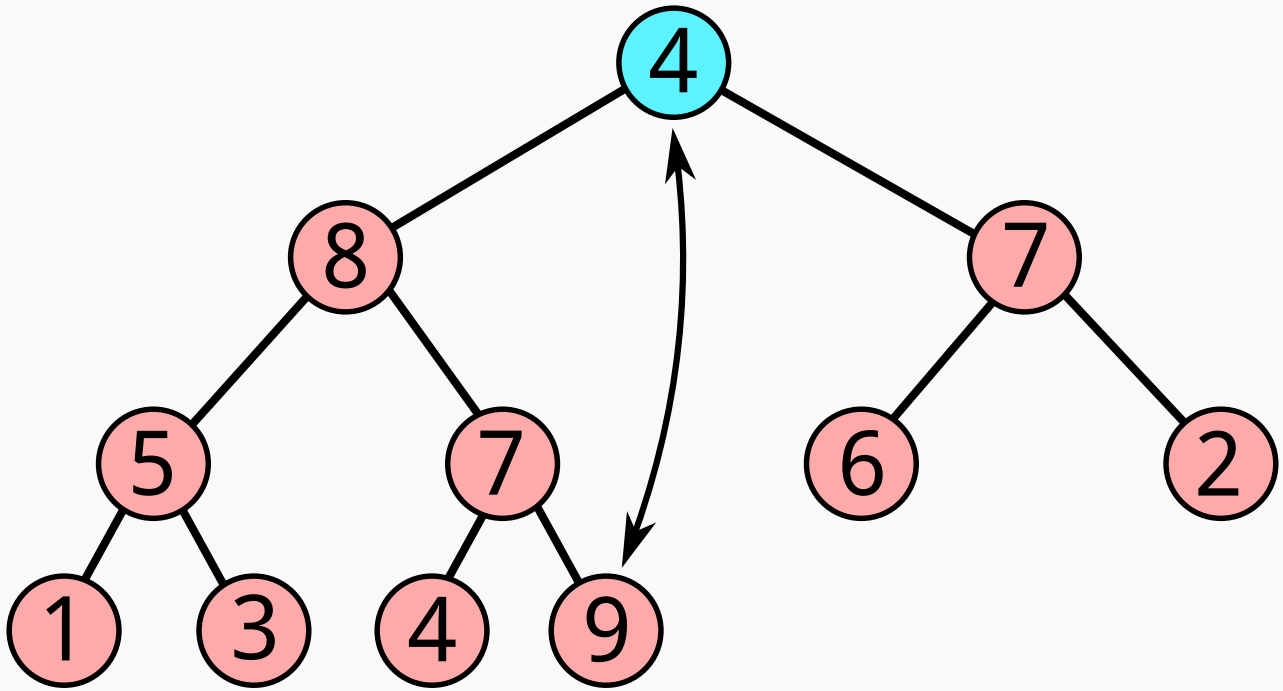
Operasi Pop (lanj.)

Misalkan akan dilakukan pop pada *heap* berikut:



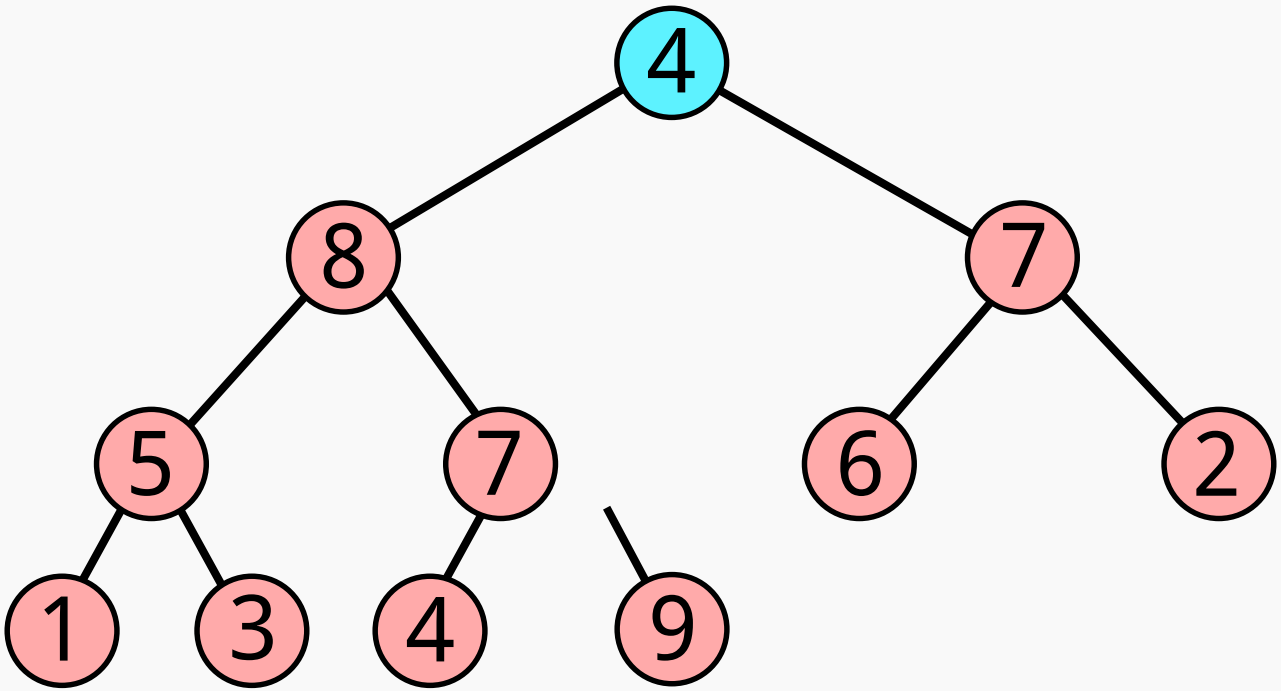
Operasi Pop (lanj.)

Tukar elemen pada *root* dengan elemen terakhir pada *complete binary tree*.



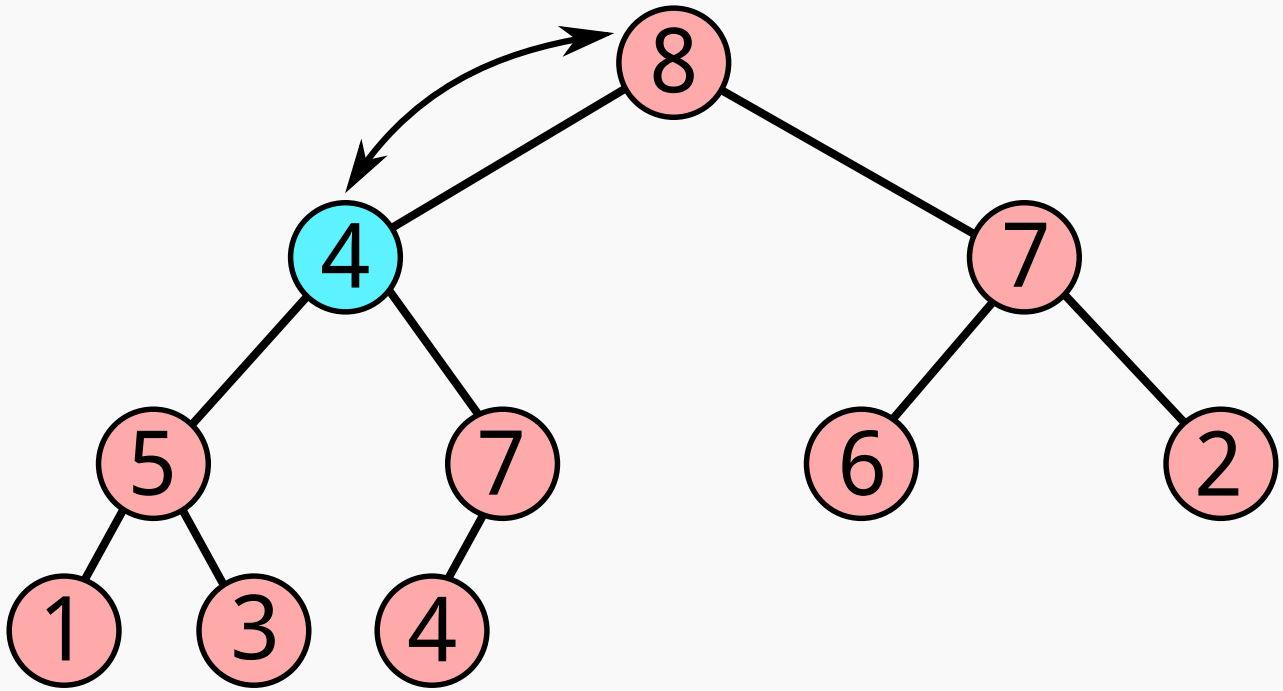
Operasi Pop (lanj.)

Buang elemen terakhir.



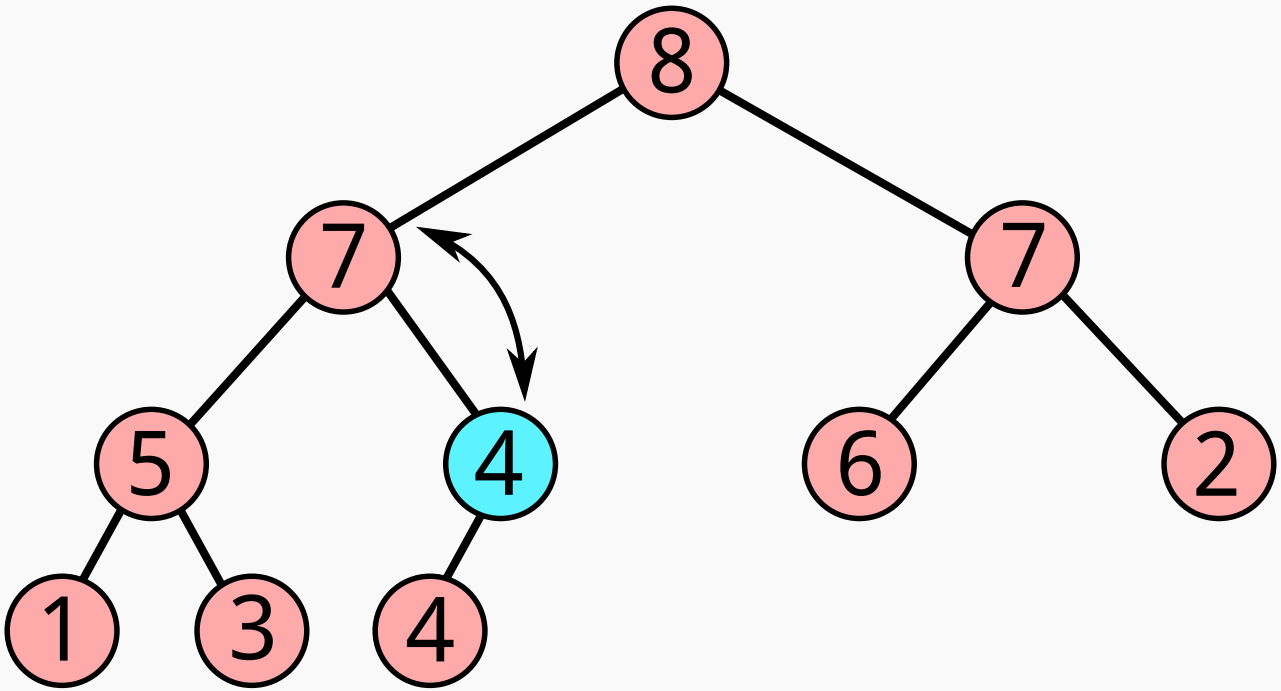
Operasi Pop (lanj.)

Perbaiki struktur *heap* dengan menukar elemen pada *root* dengan anaknya yang bernilai **terbesar**.



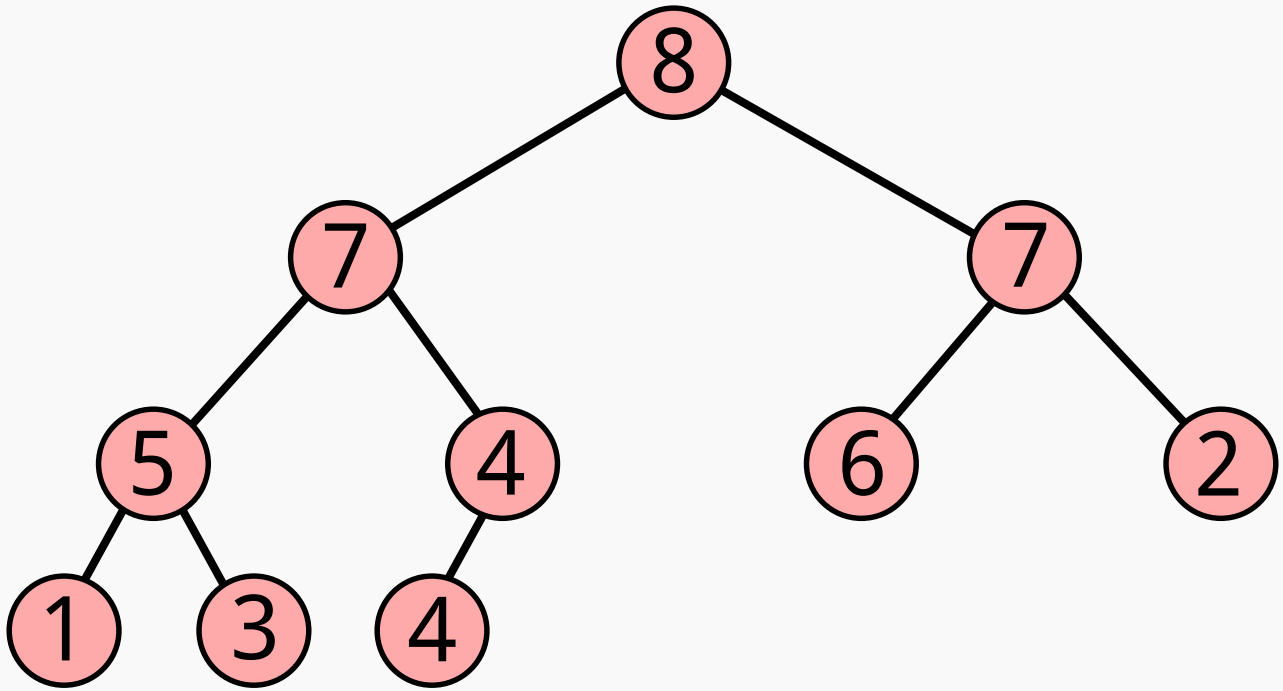
Operasi Pop (lanj.)

Karena masih terdapat anaknya yang lebih besar, tukar lagi.



Operasi Pop (lanj.)

Kini sudah tidak ada anak yang bernilai lebih besar, operasi pop selesai.



Kompleksitas Pop

- Kasus terburuk juga terjadi ketika pertukaran yang terjadi paling banyak.
- Hal ini terjadi ketika elemen yang ditempatkan di *root* cukup kecil, sehingga perlu ditukar sampai ke tingkat paling bawah.
- Banyaknya pertukaran yang terjadi sebanding dengan kedalaman dari *complete binary tree*.
- Kompleksitas untuk operasi *pop* adalah $O(\log N)$.

Operasi Top

- Operasi ini sebenarnya sesederhana mengembalikan elemen pada *root binary heap*.
- Kompleksitas operasi ini adalah $O(1)$.

Analisis Solusi dengan Heap

Penerapan *heap* pada persoalan motivasi:

Operasi	Dengan sorting	Dengan Heap
add(x)	$O(N \log N)$	$O(\log N)$
getMax()	$O(1)$	$O(1)$
deleteMax()	$O(1)$	$O(\log N)$

Kini seluruh operasi dapat dilakukan dengan efisien.

Bagian 2

Implementasi Binary Heap

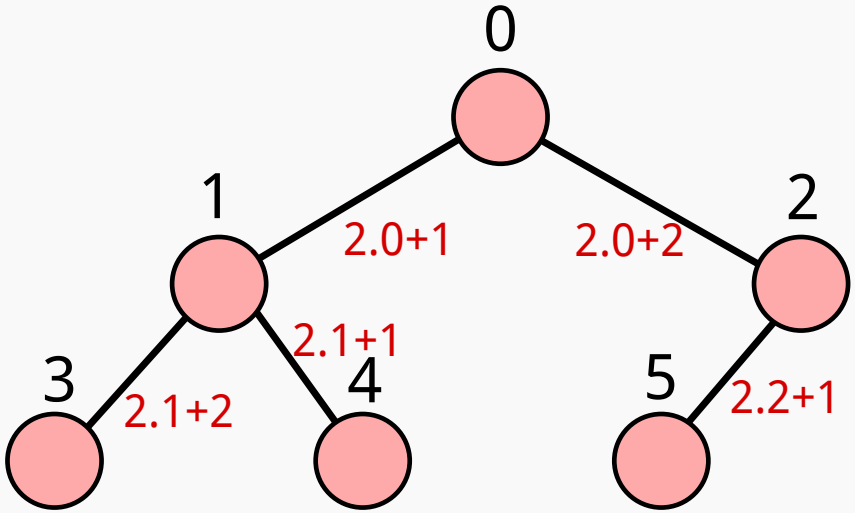
Membuat Tree

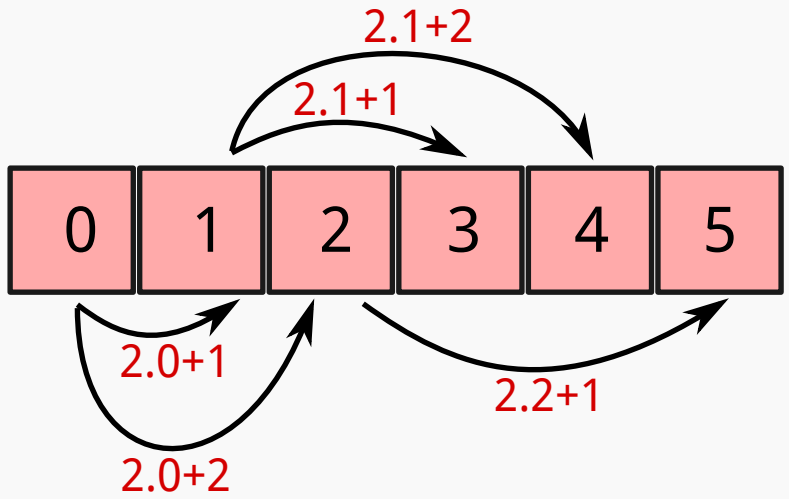
- Representasi *tree* pada implementasi dapat menggunakan teknik representasi graf yang telah dipelajari sebelumnya.
- Namun, untuk *tree* dengan kondisi tertentu, kita dapat menggunakan representasi yang lebih sederhana.
- Terutama pada kasus ini, yang mana *tree* yang diperlukan adalah *complete binary tree*.

Representasi Complete Binary Tree

- Kedengarannya kurang masuk akal, tetapi *complete binary tree* dapat direpresentasikan dengan sebuah *array*.
- Misalkan *array* ini bersifat *zero-based*, yaitu dimulai dari indeks 0.
- Elemen pada indeks ke-i menyatakan elemen pada *node* ke-i.
- Anak kiri dari *node* ke-i adalah *node* ke-$(2i + 1)$.
- Anak kanan dari *node* ke-i adalah *node* ke-$(2i + 2)$.

Representasi Complete Binary Tree (lanj.)





Representasi Complete Binary Tree (lanj.)

- Dengan logika yang serupa, orang tua dari *node* ke-i adalah *node* ke-$\lfloor \frac{i-1}{2} \rfloor$.
- Apabila Anda memutuskan untuk menggunakan *one-based*, berarti rumusnya menjadi:
  - Anak kiri: $2i$.
  - Anak kanan: $2i + 1$.
  - Orang tua: $\lfloor \frac{i}{2} \rfloor$
- Representasi ini sangat mempermudah implementasi *binary heap*.

Representasi Array

- Karena panjang *array* dapat bertambah atau berkurang, diperlukan *array* yang ukurannya dinamis.
- Pada contoh ini, kita akan menggunakan *array* berukuran statis dan sebuah variabel yang menyatakan ukuran *array* saat ini.
- Berikut prosedur untuk inisialisasi, dengan asumsi *maxSize* menyatakan ukuran terbesar pada *heap* yang mungkin.

```
INITIALIZEHEAP(maxSize)
1  // Buat array arr berukuran maxSize
2  size = 0
```

arr dan *size* merupakan variabel global, yaitu *array* dan variabel yang menyatakan ukurannya saat ini.

Implementasi Fungsi Pembantu

Buat juga beberapa fungsi yang akan membantu mempermudah penulisan kode.

GETPARENT(x)

1 **return** FLOOR($(x-1)/2$)

GETLEFT(x)

1 **return** $2x+1$

GETRIGHT(x)

1 **return** $2x+2$

Implementasi Push

```
PUSH(val)
1   i = size
2   arr[i] = val
3   while (i > 0) ∧ (arr[i] > arr[GETPARENT(i)])
4         SWAP(arr[i], arr[GETPARENT(i)])
5         i = GETPARENT(i)
6   size = size + 1
```

Implementasi Pop

```
POP()
 1  SWAP(arr[0], arr[size − 1])
 2  size = size − 1
 3  i = 0
 4  swapped = true
 5  while swapped
 6        maxIdx = i
 7        if (GETLEFT(i) < size) ∧ (arr[maxIdx] < arr[GETLEFT(i)])
 8              maxIdx = GETLEFT(i)
 9        if (GETRIGHT(i) < size) ∧ (arr[maxIdx] < arr[GETRIGHT(i)])
10              maxIdx = GETRIGHT(i)
11        SWAP(arr[i], arr[maxIdx])
12        swapped = (maxIdx ≠ i) // true bila terjadi pertukaran
13        i = maxIdx
```

Implementasi Top

```
TOP()
    return arr[size − 1]
```

... sangat sederhana.

Pembuatan Heap

Ketika Anda memiliki data N elemen, dan hendak dimasukkan ke dalam *heap*, Anda dapat:

1. Membuat *heap* kosong, lalu melakukan *push* satu per satu hingga seluruh data dimuat *heap*. Kompleksitasnya $O(N \log N)$.
2. Membuat *array* dengan N elemen, lalu *array* ini dibuat menjadi *heap* dalam $O(N)$. Caranya akan dijelaskan pada bagian berikutnya.

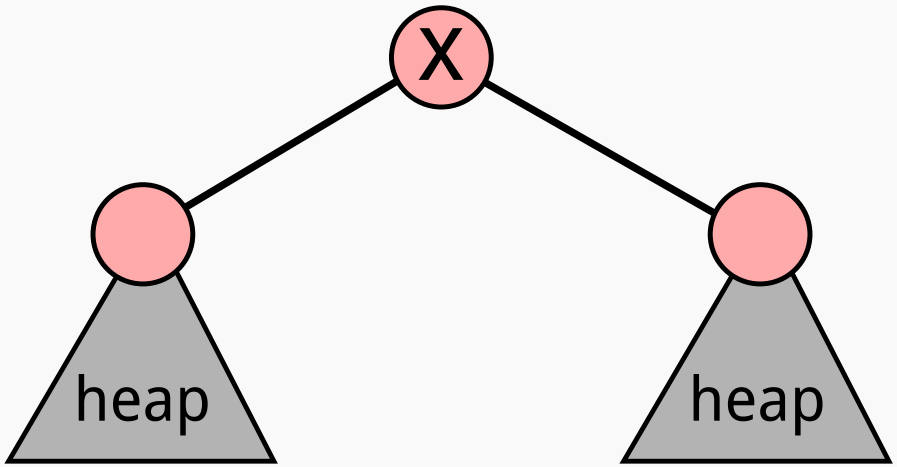
Pembuatan Heap Secara Efisien

Untuk membuat *heap* dari N elemen, caranya adalah:

1. Buat *array* dengan N elemen, lalu isi *array* ini dengan elemen-elemen yang akan dimasukkan pada *heap*.
2. Untuk semua *node*, mulai dari tingkat kedua dari paling bawah:
   - Misalkan *node* ini adalah *node* x.
   - Pastikan *subtree* yang bermula pada *node* x **membentuk heap** dengan operasi baru, yaitu **heapify**.

Operasi Heapify

- **Heapify** adalah operasi untuk membentuk sebuah *heap* yang bermula pada suatu *node*, dengan asumsi anak kiri dan anak kanan *node* tersebut sudah membentuk *heap*.



Operasi Heapify (lanj.)

- Berhubung kedua anak telah membentuk *heap*, kita cukup memindahkan elemen *root* ke posisi yang tepat.
- Jika elemen *root* sudah lebih besar daripada elemen anaknya, tidak ada yang perlu dilakukan.
- Sementara bila elemen *root* lebih kecil daripada salah satu elemen anaknya, tukar elemennya dengan elemen salah satu anaknya yang **paling besar**.
- Kegiatan ini sebenarnya sangat mirip dengan yang kita lakukan pada operasi *pop*.

Operasi Heapify (lanj.)

```
HEAPIFY(rootIdx)
 1   i = rootIdx
 2   swapped = true
 3   while swapped
 4         maxIdx = i
 5         if (GETLEFT(i) < size) ∧ (arr[maxIdx] < arr[GETLEFT(i)])
 6               maxIdx = GETLEFT(i)
 7         if (GETRIGHT(i) < size) ∧ (arr[maxIdx] < arr[GETRIGHT(i)])
 8               maxIdx = GETRIGHT(i)
 9         SWAP(arr[i], arr[maxIdx])
10         swapped = (maxIdx ≠ i) // true bila terjadi pertukaran
11         i = maxIdx
```

Operasi Heapify (lanj.)

Operasi *pop* sendiri dapat kita sederhanakan menjadi:

POP()

1 SWAP($arr[0], arr[size - 1]$)
2 $size = size - 1$
3 HEAPIFY(0)

Kompleksitas Heapify

- Sekali melakukan *heapify*, kasus terburuknya adalah elemen *root* dipindahkan hingga ke paling bawah *tree*.
- Jadi kompleksitasnya adalah $O(H)$, dengan H adalah ketinggian dari *complete binary tree*.
- Berhubung $H = O(\log N)$, dengan N adalah banyaknya elemen pada *heap*, kompleksitas *heapify* adalah $O(\log N)$.

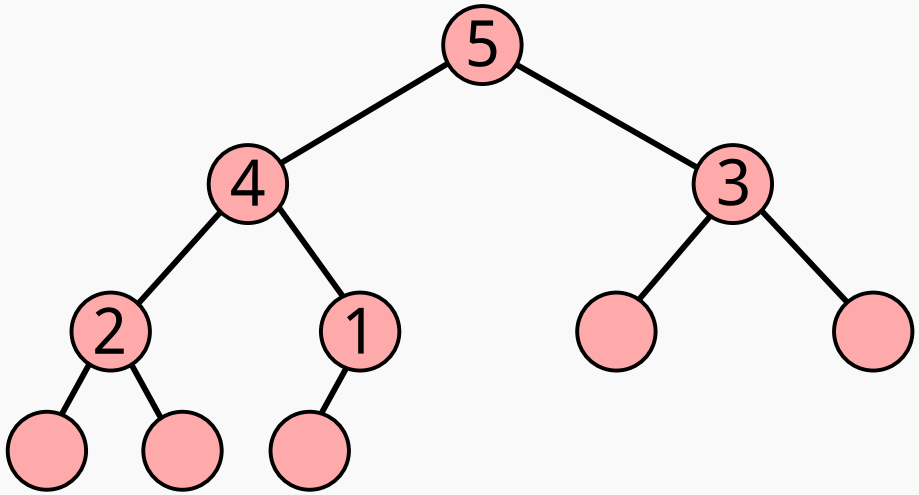
Implementasi Pembangunan Heap Secara Efisien

- Setelah memahami *heapify*, kita dapat menulis prosedur pembangunan *heap* dari *array* A berisi N elemen secara efisien.
- Elemen terakhir yang berada pada tingkat kedua paling bawah dapat ditemukan dengan mudah, yaitu elemen dengan indeks $N/2$ dibulatkan ke bawah dan dikurangi 1.

```
makeHeap(N)
1  initializeHeap(N)
2  for i = 0 to N − 1
3        arr[size] = A[i]
4        size = size + 1
5  for i = ⌊N/2⌋ − 1 to 0
6        heapify(i)
```

Ilustrasi Pembangunan Heap Secara Efisien

Angka pada *node* menyatakan urutan pelaksanaan *heapify*.



Analisis Pembangunan Heap Secara Efisien

- Dilakukan $N/2$ kali operasi *heapify*, yang masing-masing $O(\log N)$.
- Kompleksitasnya terkesan $O(N \log N)$.
- Namun pada kasus ini, sebenarnya setiap *heapify* tidak benar-benar $O(\log N)$.
- Kenyataannya, banyak operasi *heapify* yang dilakukan pada tingkat bawah, yang relatif lebih cepat dari *heapify* pada tingkat di atas.
- Perhitungan secara matematis membuktikan bahwa kompleksitas keseluruhan MAKEHEAP adalah $O(N)$.

Catatan Implementasi

- Tentu saja, Anda dapat membuat *heap* dengan urutan yang terbalik, yaitu elemen terkecilnya di atas.
- Dengan demikian, operasi yang didukung adalah mencari atau menghapus elemen terkecil.
- Biasanya *heap* dengan sifat ini disebut dengan **min-heap**, sementara *heap* dengan elemen terbesar di atas disebut dengan **max-heap**.
- Agar lebih rapi, Anda dapat menggunakan **struct** (C) atau **class** (C++) pada implementasi *heap*.

Manfaat Heap

- Pada ilmu komputer, *heap* dapat digunakan sebagai *priority queue*, yaitu antrean yang terurut menurut suatu kriteria.
- Sifat *heap* juga dapat digunakan untuk optimisasi suatu algoritma. Contoh paling nyatanya adalah untuk mempercepat algoritma Dijkstra.
- Berbagai solusi persoalan *greedy* juga dapat diimplementasikan secara efisien dengan *heap*.

Library Heap

- Bagi pengguna C++, struktur data **priority_queue** dari *header* **queue** merupakan struktur data *heap*.

Penutup

- Dengan mempelajari *heap*, Anda memperdalam pemahaman tentang bagaimana penggunaan struktur data yang tepat dapat membantu menyelesaikan persoalan tertentu.
- *Heap* dapat digunakan untuk pengurutan yang efisien, dan akan dibahas pada bagian berikutnya.