# IKI10400 • Struktur Data & Algoritma: Binary Heap & Huffman Code

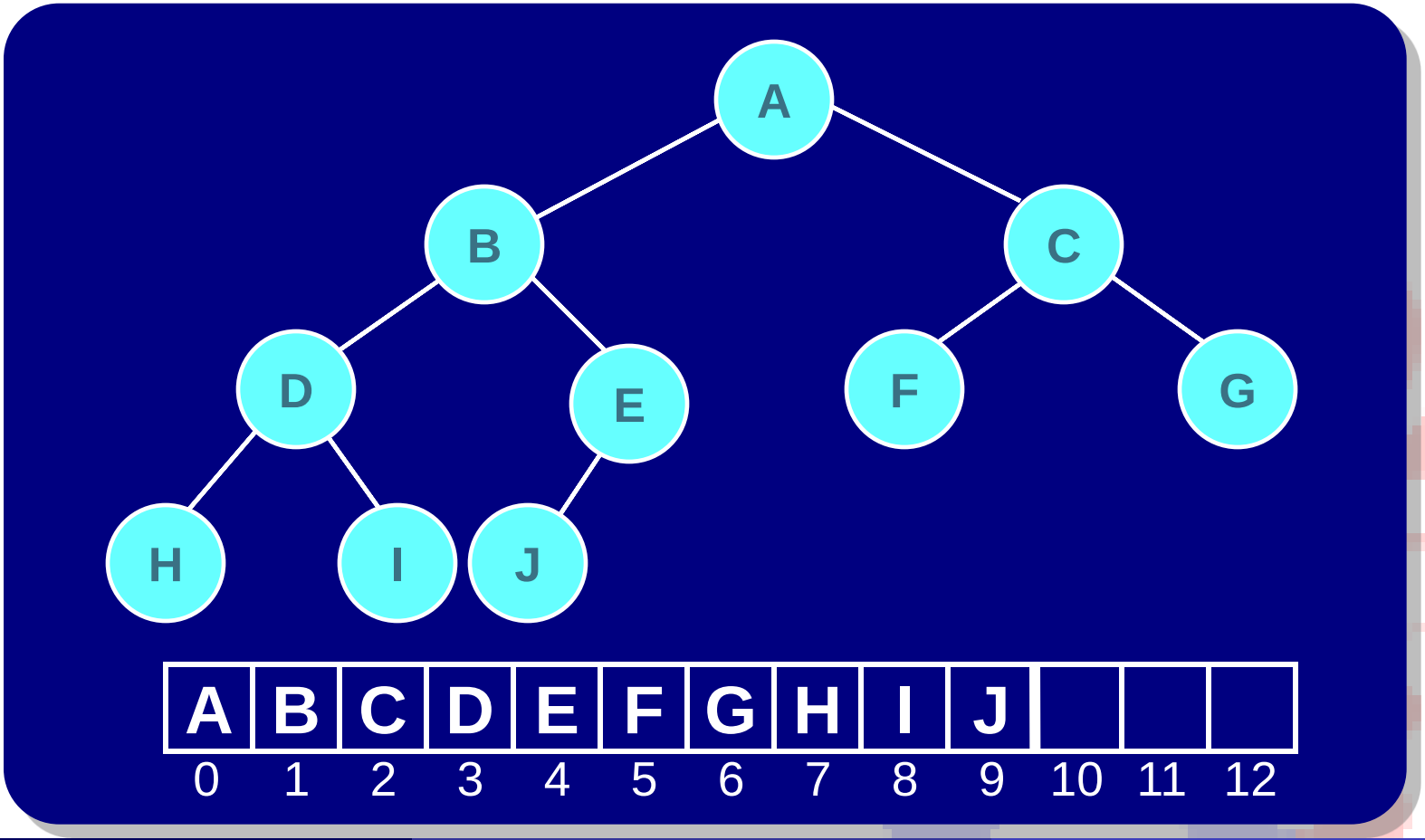
**Fakultas Ilmu Komputer • Universitas Indonesia**

*Slide acknowledgments*:
Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung

# BINARY HEAP

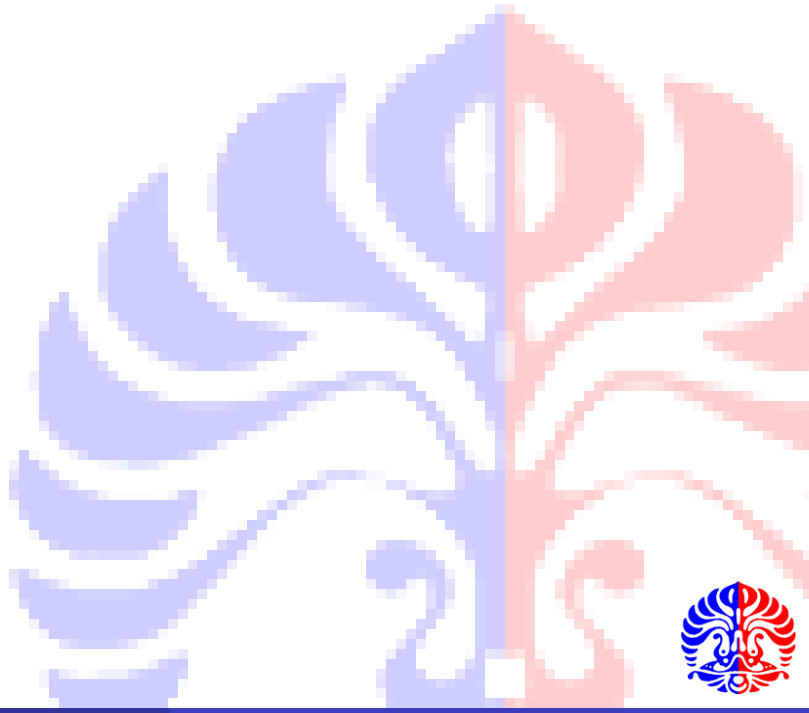
# Review

- Complete binary tree:
  - sebuah tree adalah terisi penuh (complete), kecuali pada level terbawah yang terisi dari kiri ke kanan.



# Priority Queue

- Sebuah queue dengan perbedaan aturan sebagai berikut:
  - operasi *enqueue* tidak selalu menambahkan elemen pada akhir *queue,* namun, meletakkannya sesuai urutan prioritas.



# Binary Heap

- Hanya diperbolehkan mengakses (membaca) item yang minimum.
- operasi dasar:
  - **menambahkan** item baru dengan kompleksitas waktu worst-case yang logaritmik.
  - **menghapus** item yang minimum dengan kompleksitas waktu worst-case yang logaritmik.
  - **mencari** item yang minimum dengan kompleksitas waktu konstan.

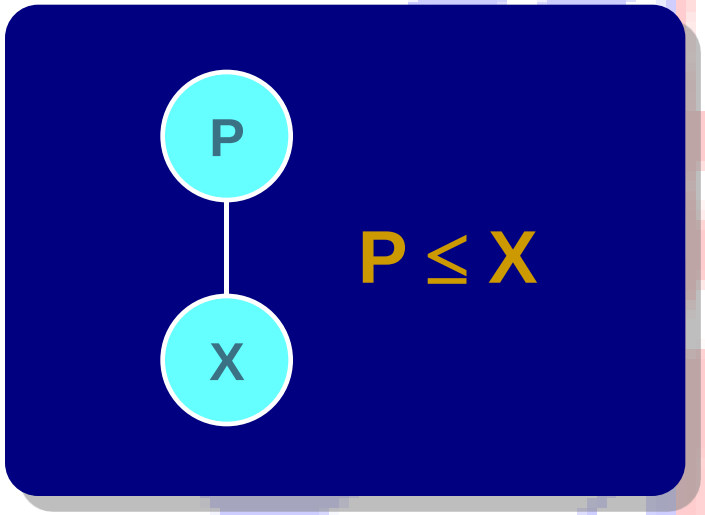
# Properties (Aturan)

- Structure Property
  - Data disimpan pada  complete binary tree

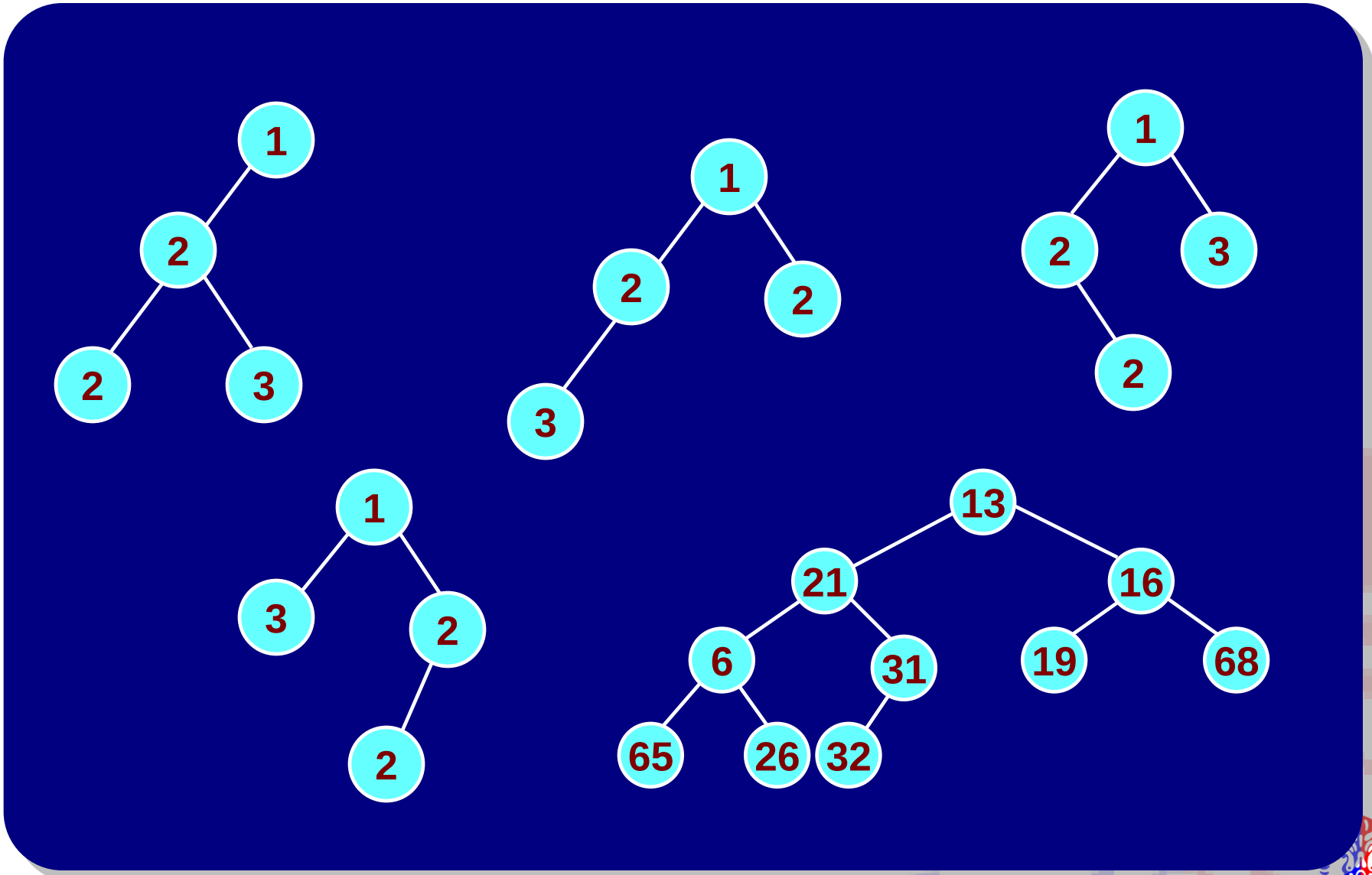
    $\Rightarrow$ tree selalu balance.

    $\Rightarrow$ seluruh operasi dijamin  O(log n) pada worst case

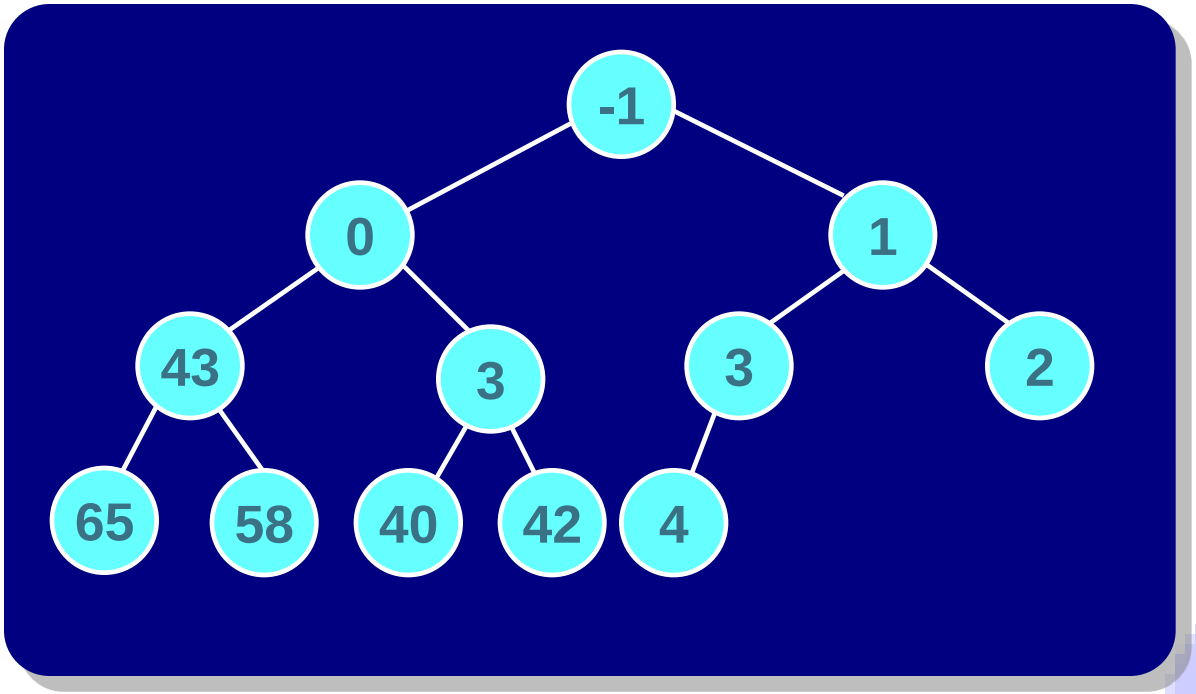
    $\Rightarrow$ data disimpan menggunakan  array atau java.util.Vector
- Ordering Property
  - Heap Order:
    - Parent $\leq$ Child



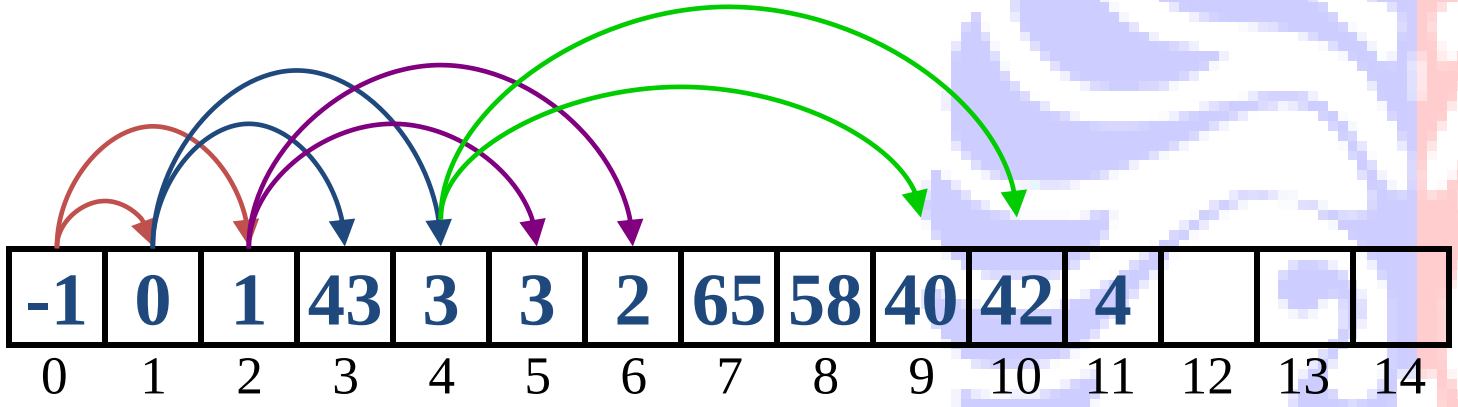
# Mana yang Binary Heap?



# Representasi Heap

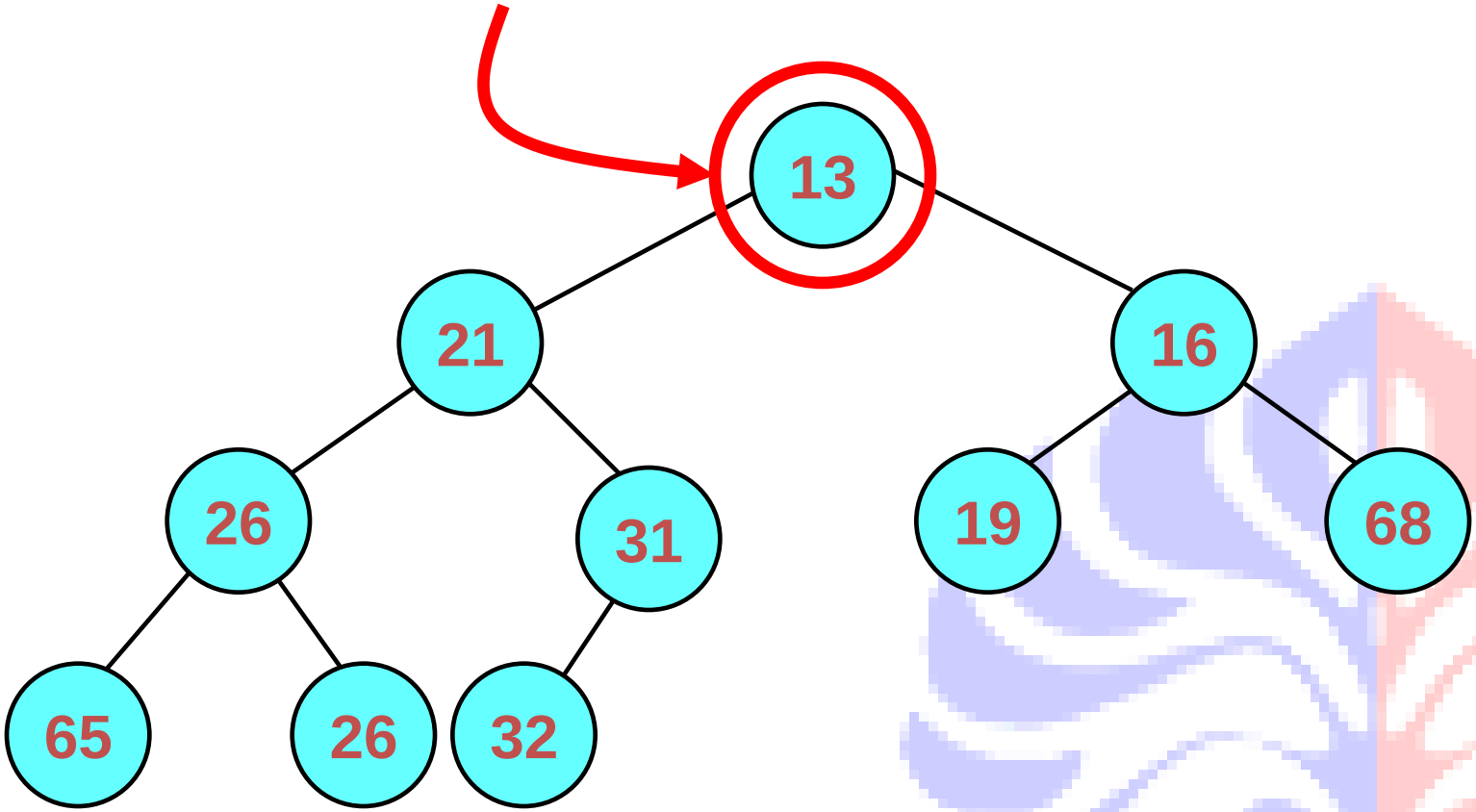



- Root pada indeks 0
- anak kiri dari i pada indeks 2i + 1
- anak kanan dari i pada indeks 2i + 2 = 2(i + 1)
- *Parent* dari i pada indeks floor((i - 1) / 2)



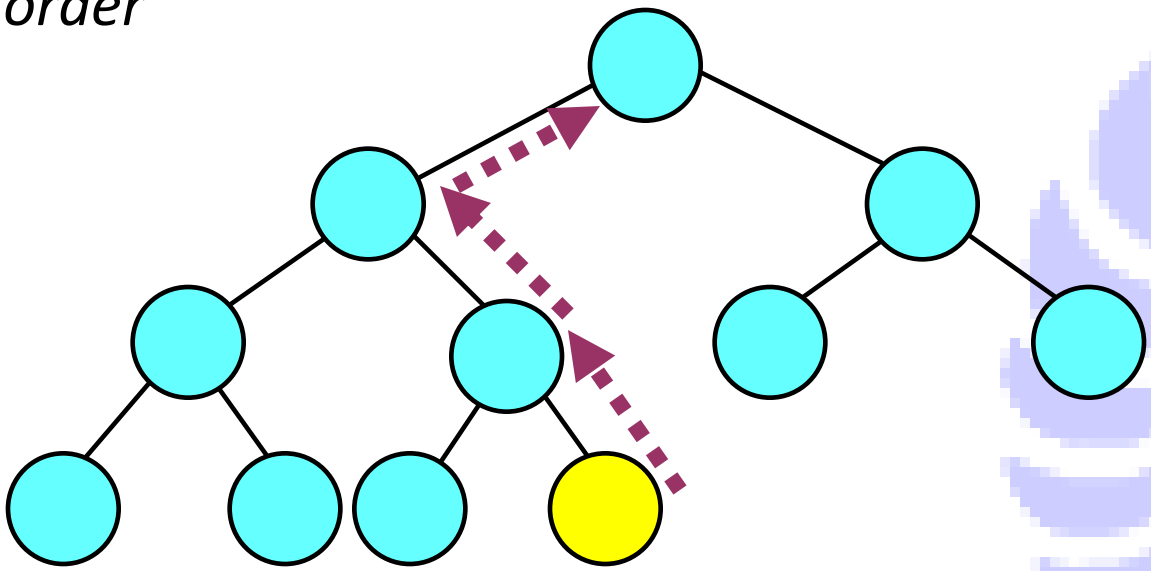
# Find Minimum

- Bagaimana cara mengakses elemen terkecil?
- Return root -> konstan



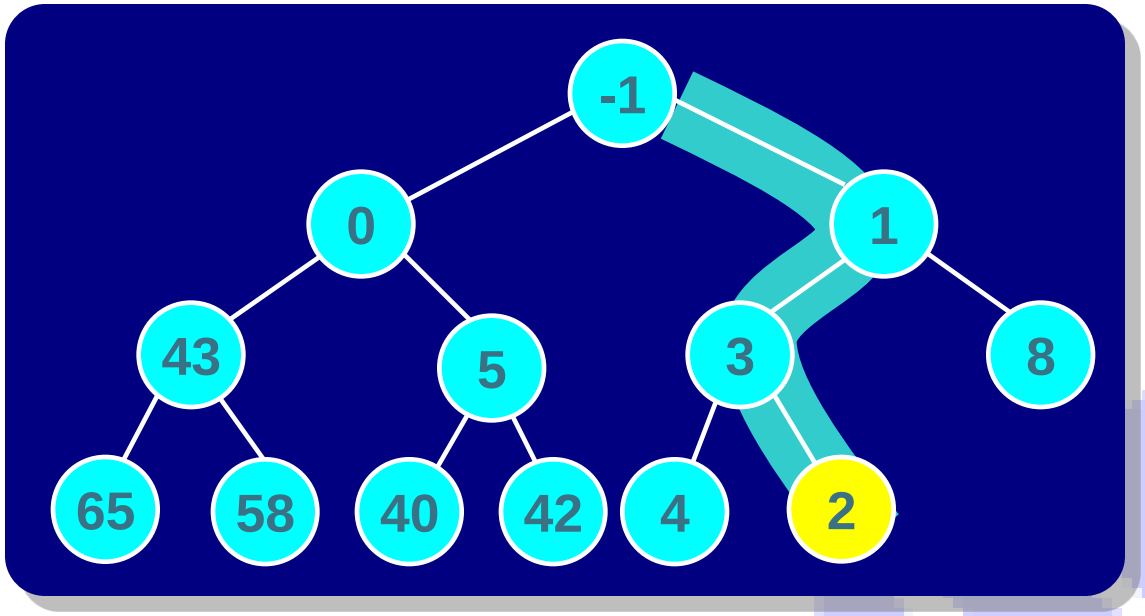
# Insertion

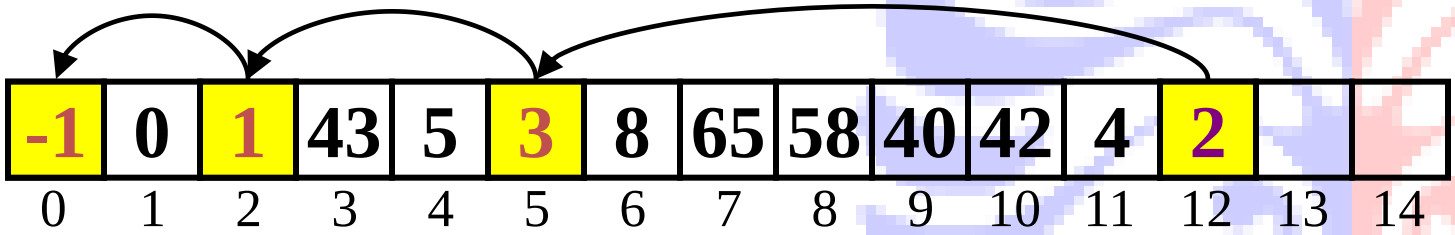
- Masukkan elemen baru ke:
  - Level paling bawah yang masih ada slot kosongnya, bila level sudah penuh, buat level baru lagi.
  - Letakkan elemen baru tersebut di slot kosong paling kiri. Tetap menjaga *complete binary tree*.
- Bila *node parent* lebih besar, tukar elemen dengan *parent*-nya. Lakukan hal tersebut sampai *root* (*percolate up*) -> menjaga *heap order*



# Insertion

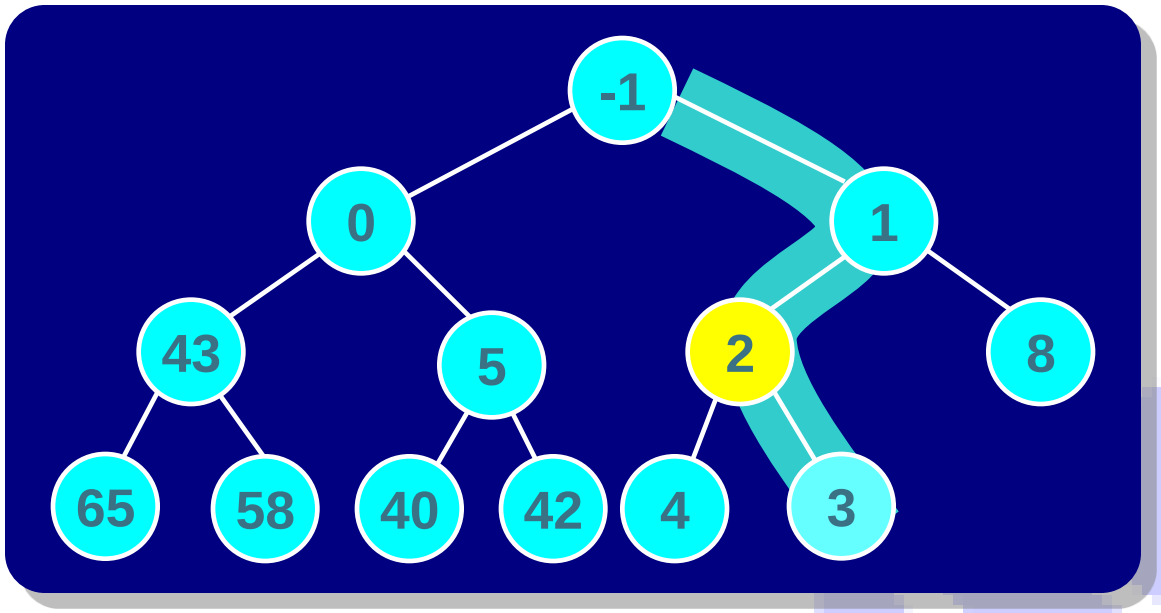
- Insert 2 (Percolate Up)

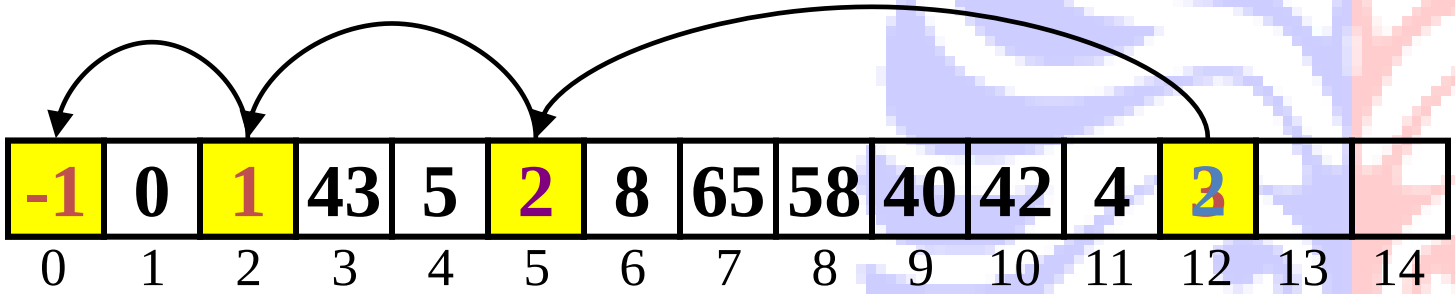


# Insertion

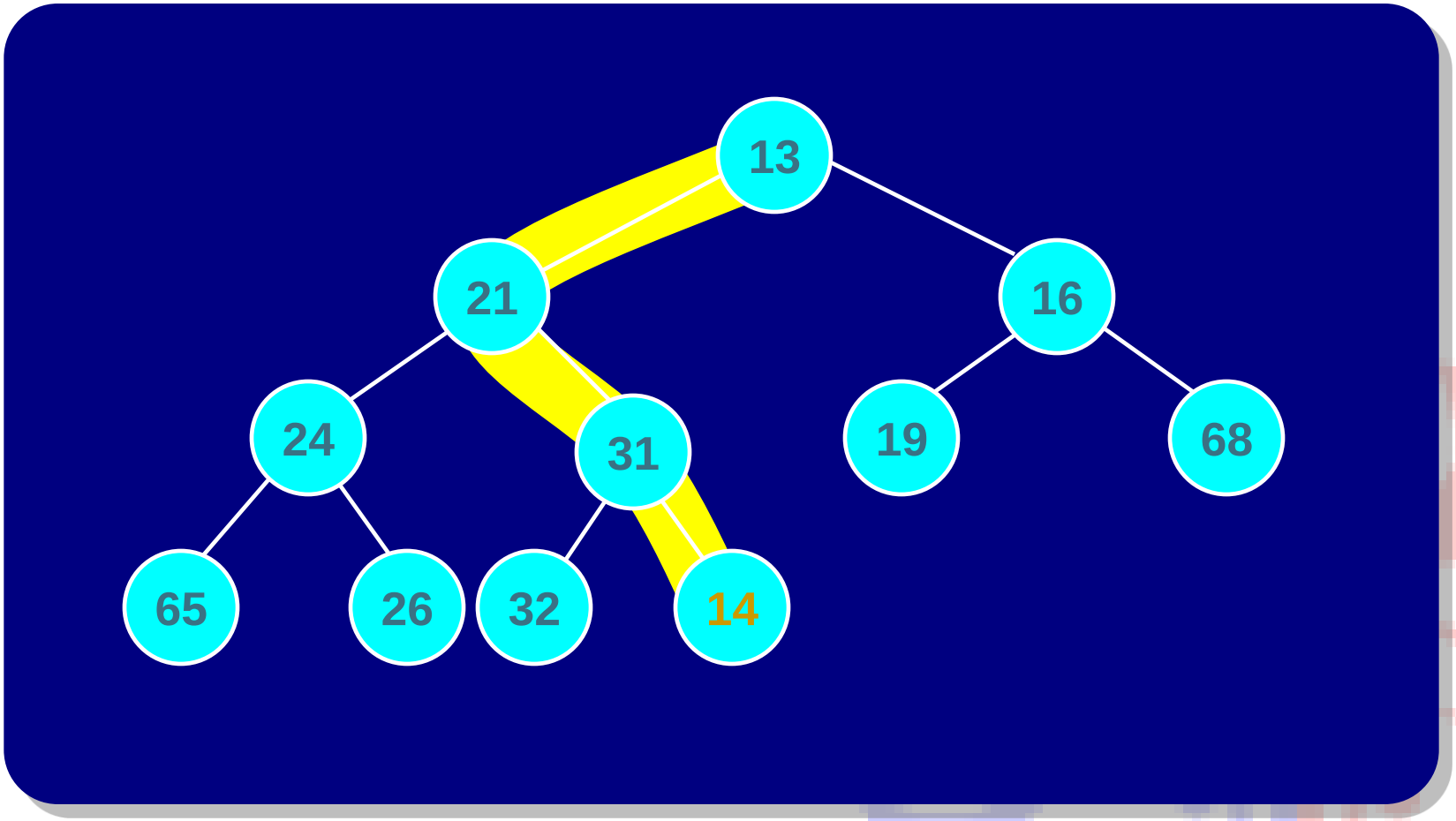
- Insert 2 (Percolate Up)



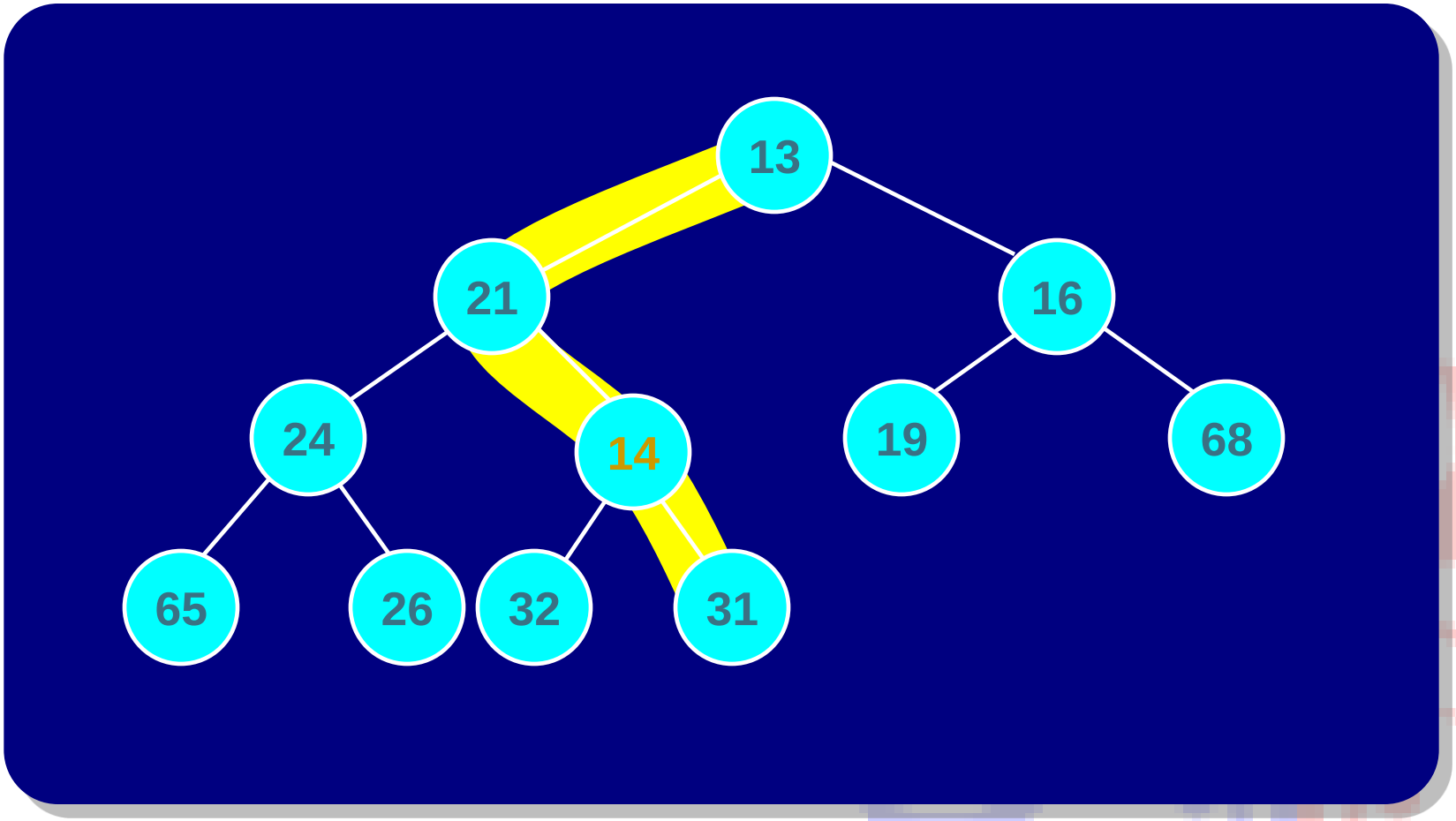
# Insertion

- Insert 14



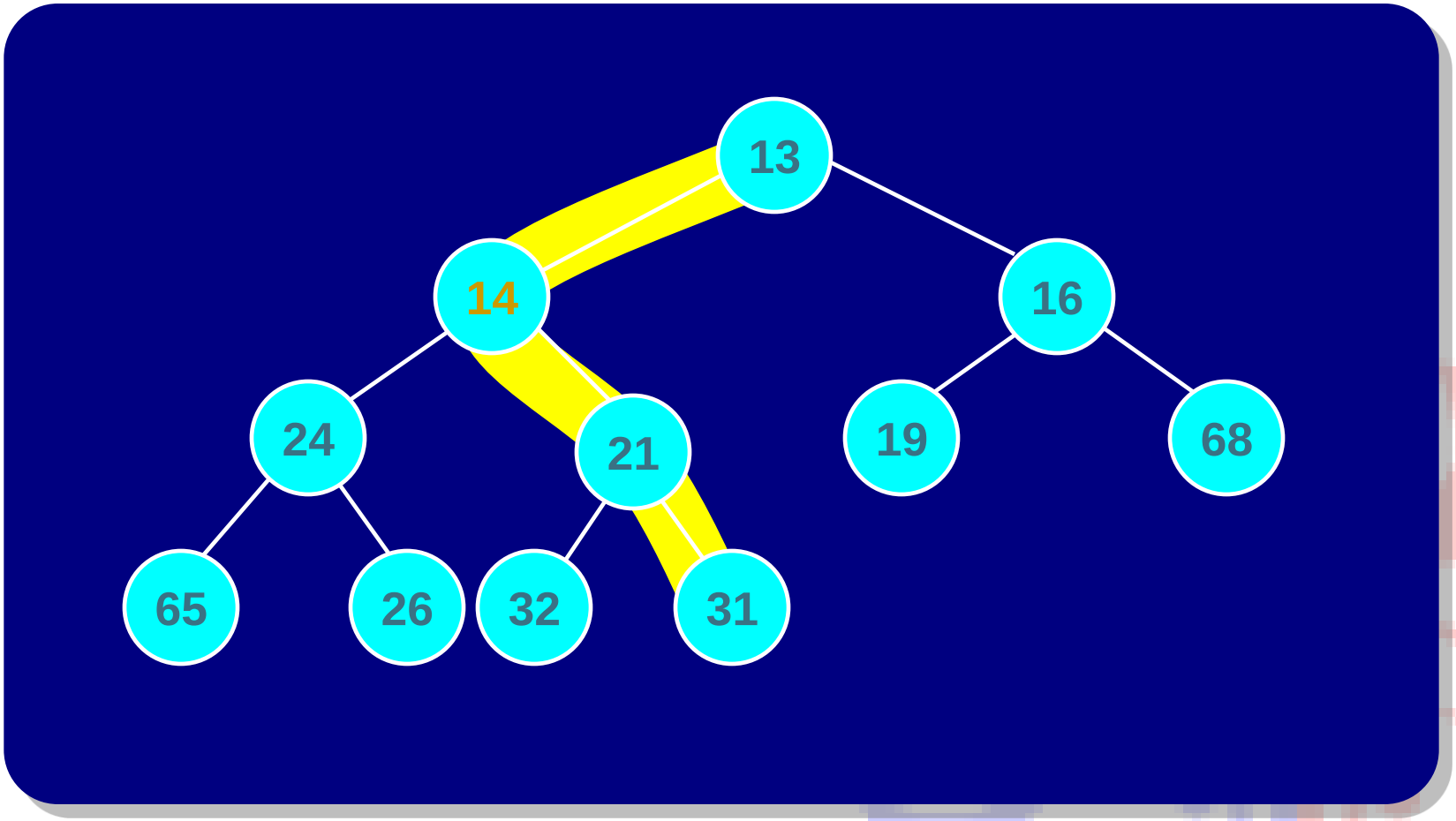
# Insertion

- Insert 14



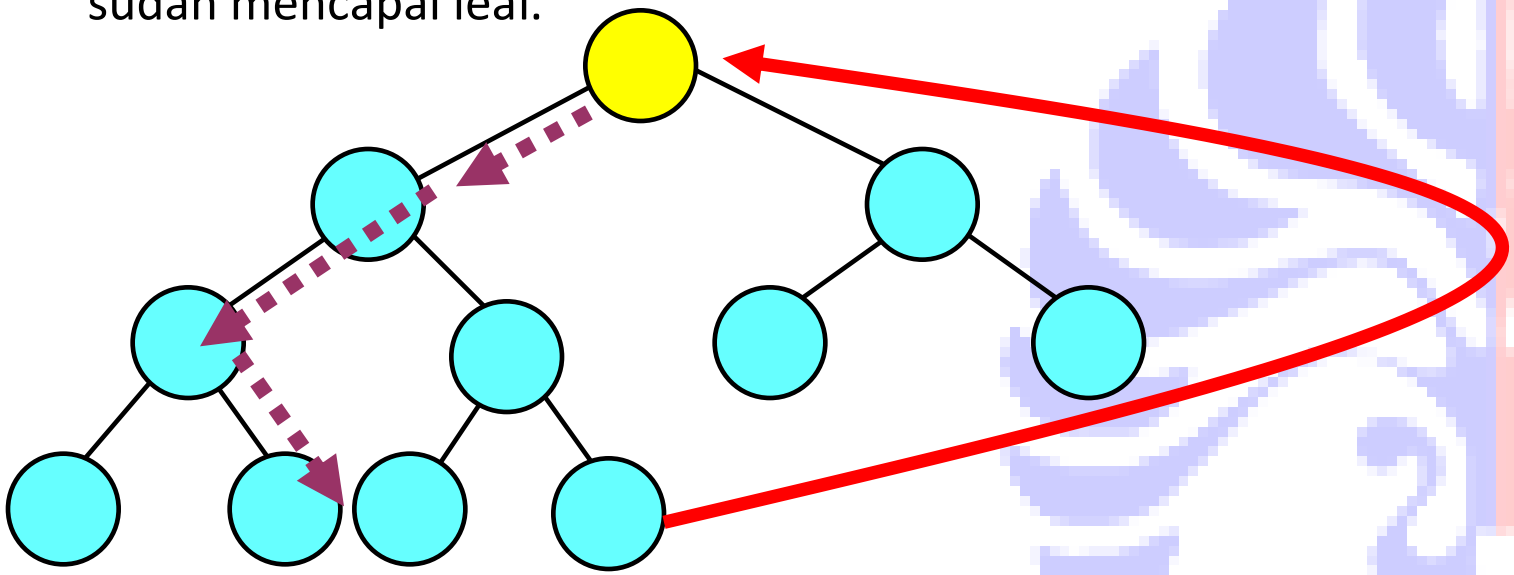
# Insertion

- Insert 14



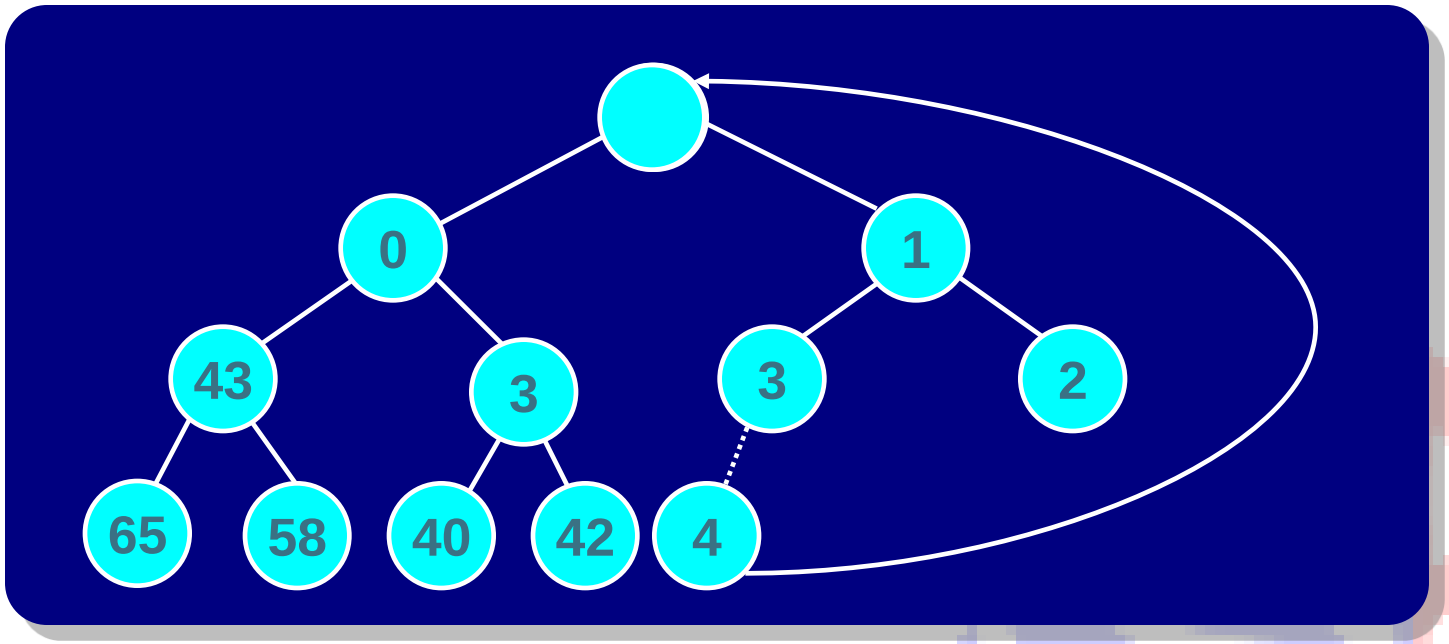
# Delete Minimum

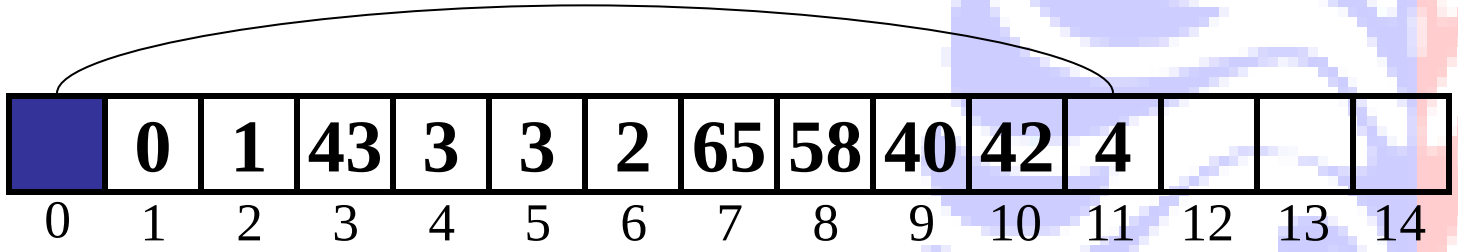
- Hanya *root* yang dapat dihapus. Mengapa?
  - Kita hanya dapat mengakses elemen terkecil saja.
- Tukar *root* dengan elemen paling kanan di level paling bawah.
- Lakukan *percolate down*:
  - Dimulai dari root, cari anak **terkecil** dari node tersebut, bila anak terkecilnya < node yang sedang dikunjungi, tukar node tsb. Lakukan sampai anak terkecil dari node yang dikunjungi tidak < dari node yang dikunjungi atau sudah mencapai leaf.



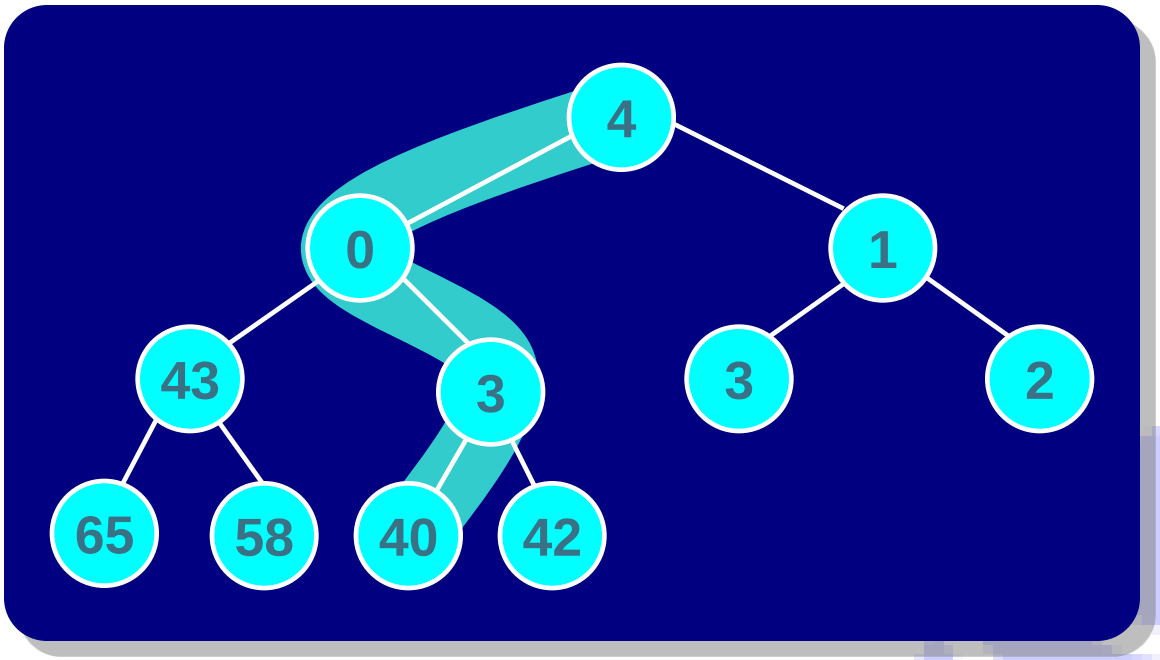
# Delete Minimum

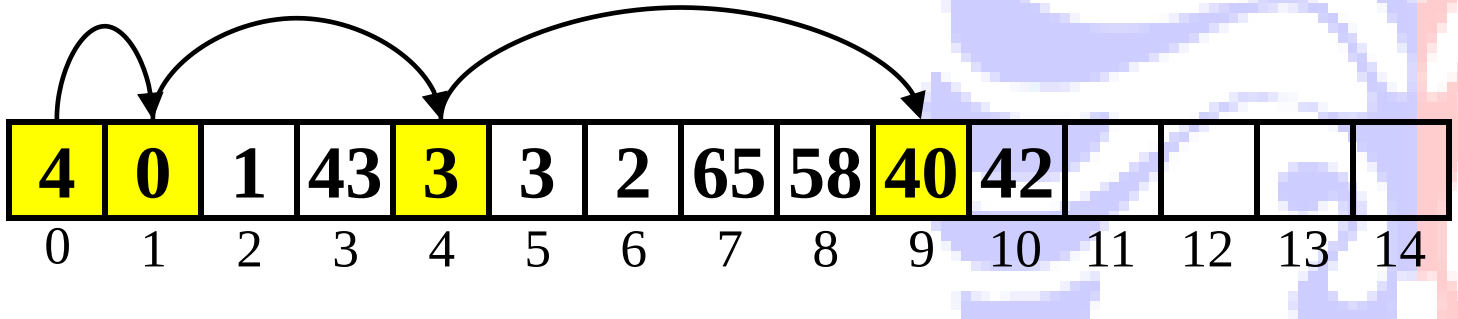
- Percolate Down



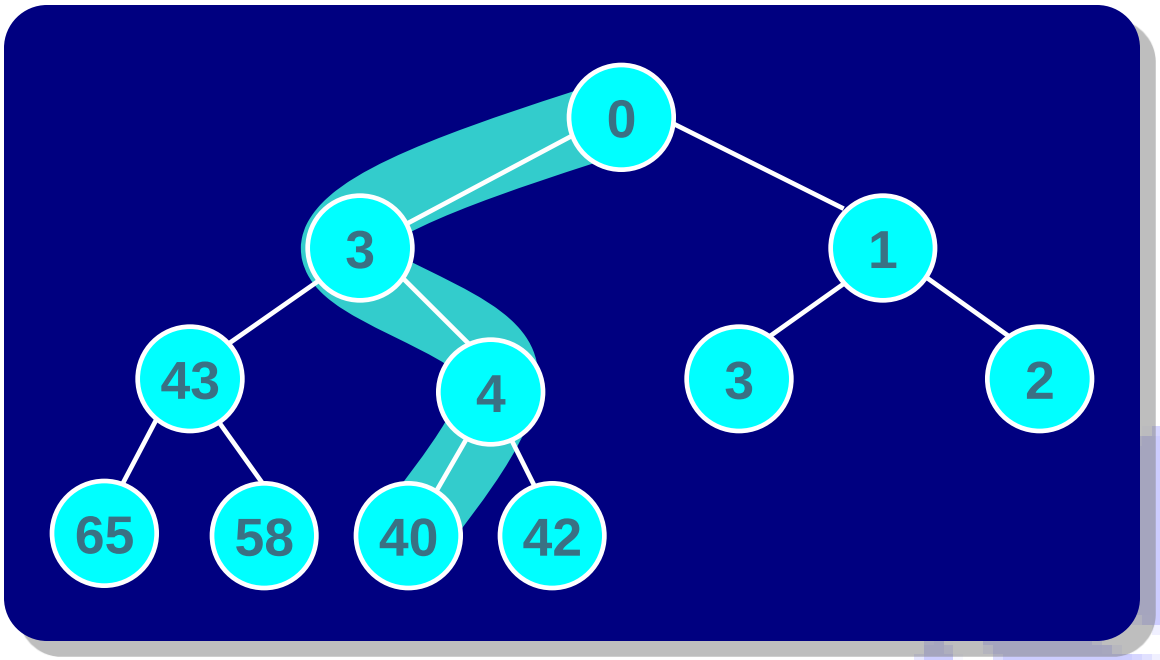



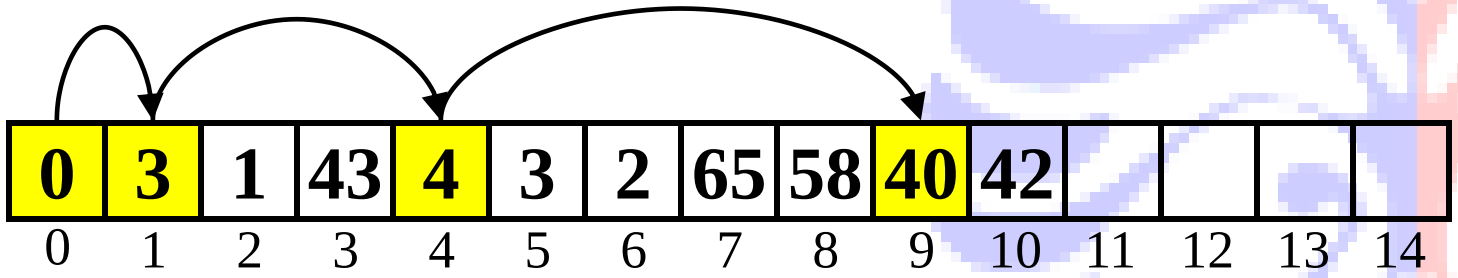
# Delete Minimum



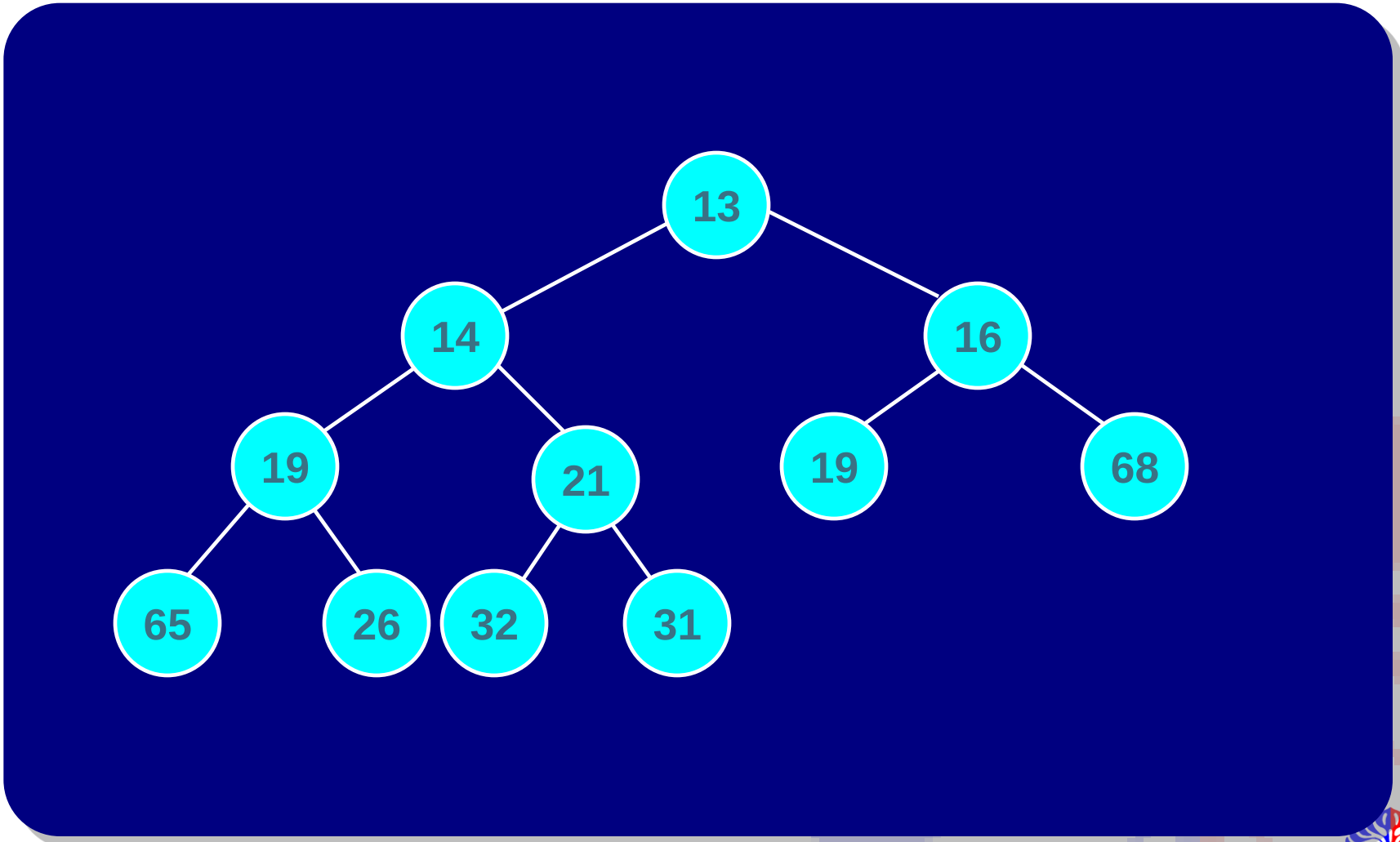



# Delete Minimum: Completed



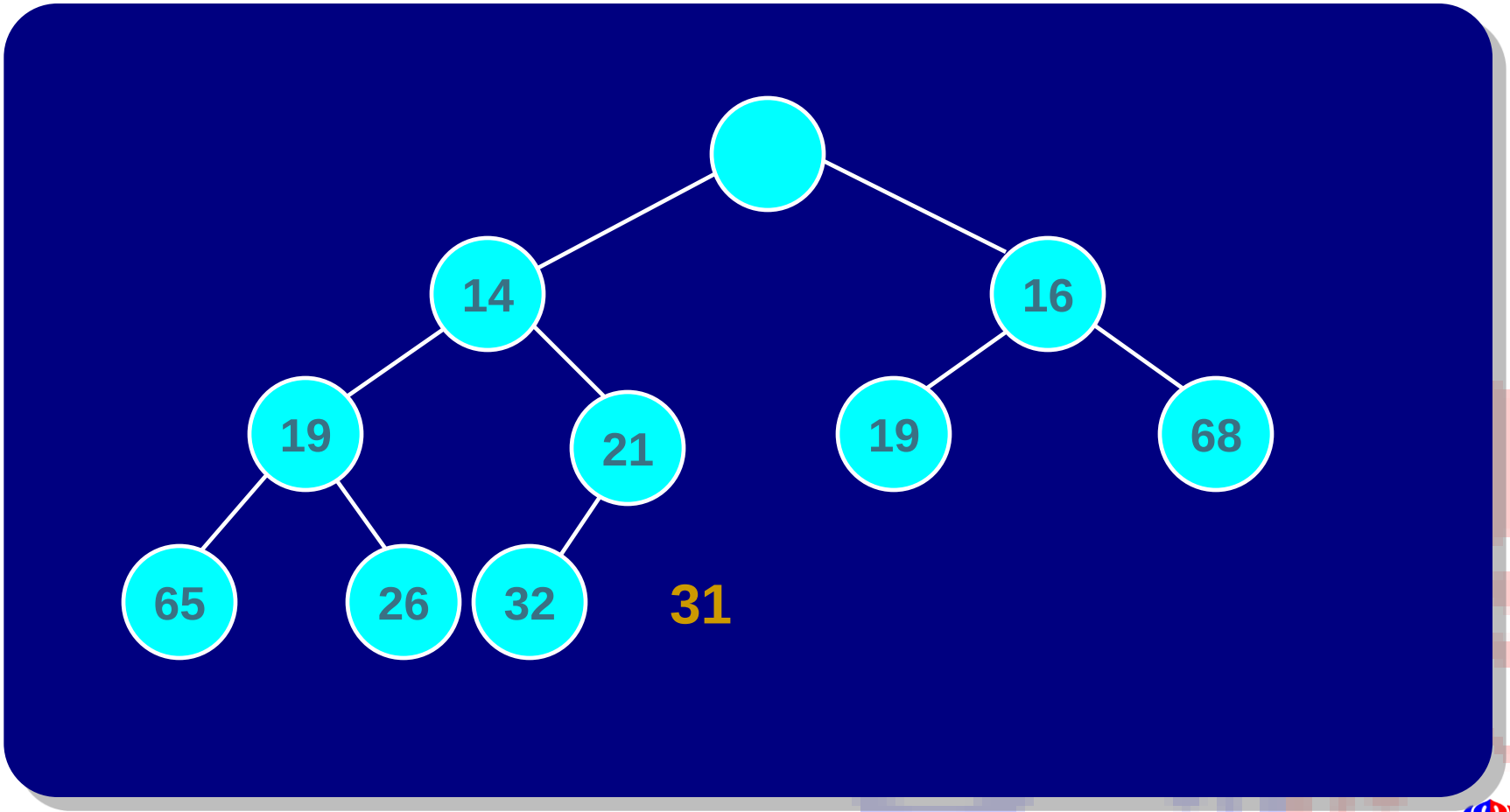



# Delete Min (2)

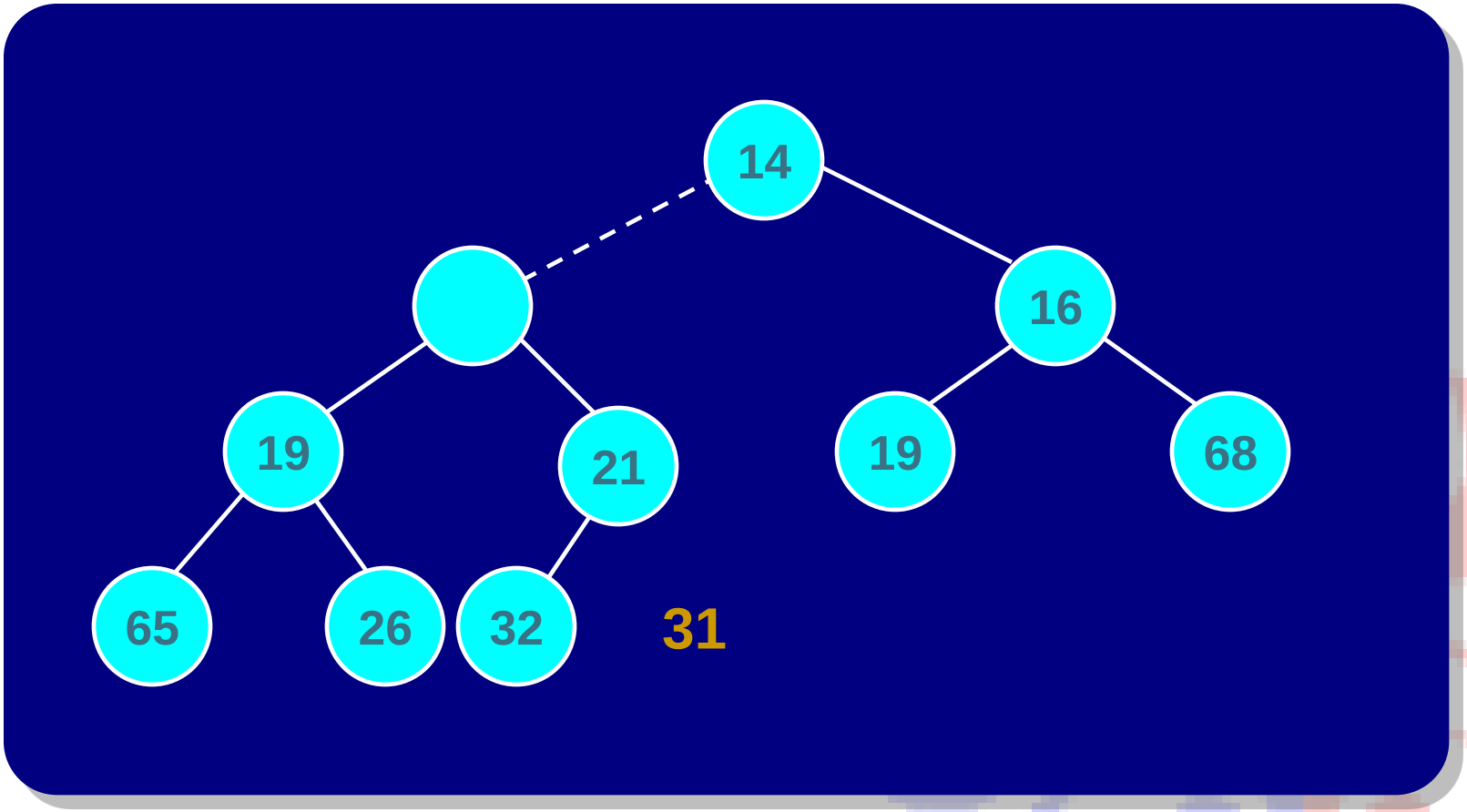


# Delete Min (2)

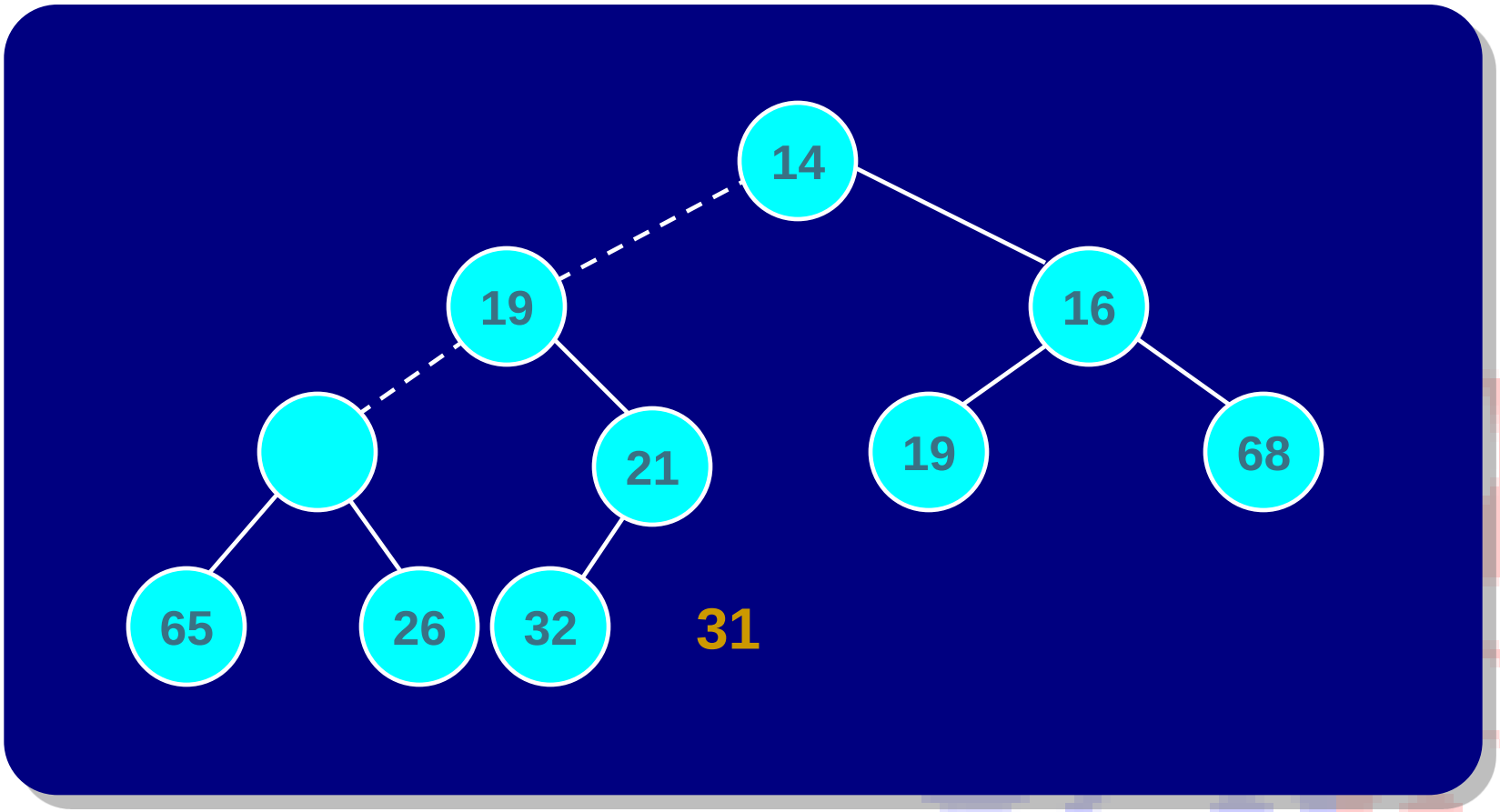
- Percolate Down



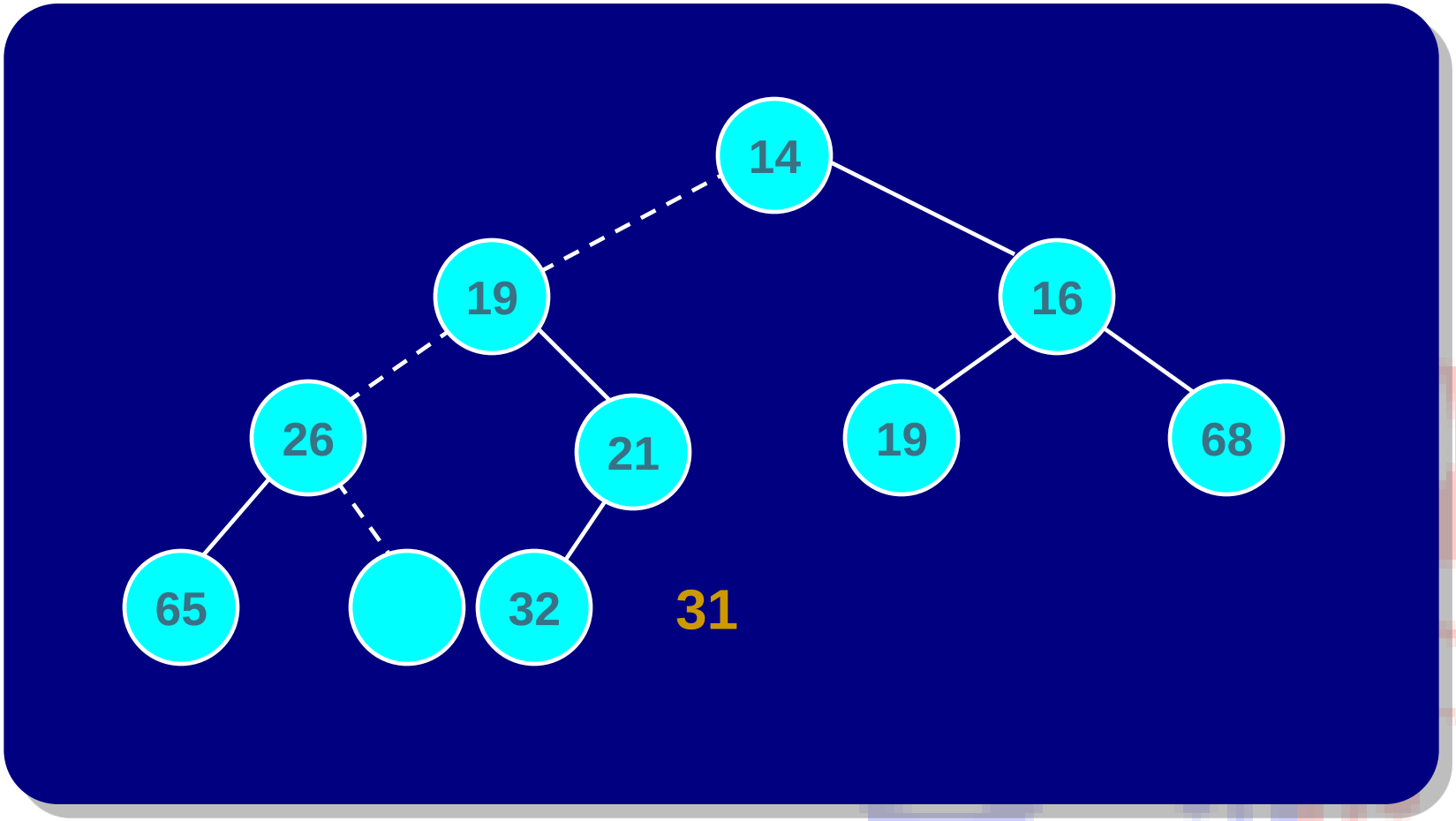
# Delete Min (2)



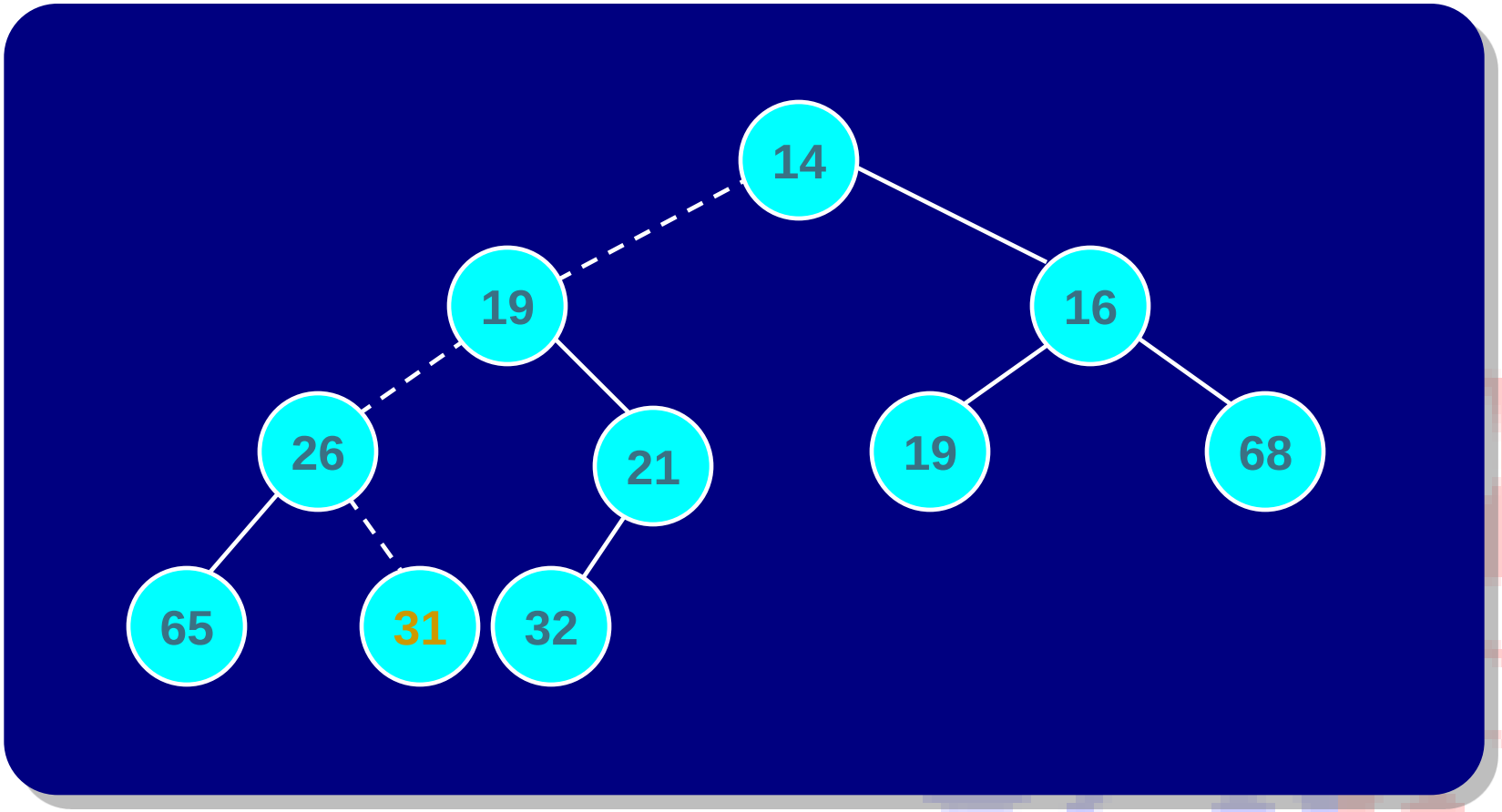
# Delete Min (2)



# Delete Min (2)



# Delete Min (2)



# Latihan

- Simulasi operasi-operasi berikut pada *Min Binary Heap*:
  - Insert: 40, 20, 5, 55, 76, 31, 3
  - Delete Min
  - Delete Min
  - Insert: 10, 22
  - Delete Min
  - Delete Min
- Gambarkan isi Binary Heap, setelah operasi terakhir.

# Heap Constructor

```
public class VectorHeap implements PriorityQueue
{
    protected Vector data;

    public VectorHeap() { data = new Vector(); }

    public VectorHeap(Vector v) {
        int i;
        data = new Vector(v.size());
        // we know ultimate size
        for (i = 0; i < v.size(); i++) {
            // add elements to heap
            add((Comparable) v.elementAt(i));
        }
    }
```

# Heap's Methods

```
protected static int parentOf(int i)   {
    return (i - 1) / 2;
}

protected static int leftChildOf(int i) {
    return 2 * i + 1;
}

protected static int rightChildOf(int i) {
    return 2 * (i + 1);
}

public Comparable peek() {
    // findMin
    return (Comparable) data.elementAt(0);
}
```

# Removal & Insertion

```
public Comparable remove()
 {
      Comparable minVal = peek();
      data.setElementAt (
          data.elementAt (data.size() - 1), 0);
      data.setSize (data.size() - 1);
      if (data.size() > 1) pushDownRoot(0);
      return minVal;
  }

  public void add(Comparable value) {
      data.addElement (value);
      percolateUp (data.size() - 1);
  }
```

# Percolate Down

```
protected void pushDownRoot (int root)
{
    int heapSize = data.size();
    Comparable value = (Comparable)
        data.elementAt (root);
    while (root < heapSize) {
        int childpos = leftChildOf(root);
        if (childpos < heapSize) {
            // choose the smallest child
            if ((rightChildOf(root) < heapSize) &&
                (((Comparable) (data.elementAt(
                childpos+1))).compareTo
                ((Comparable) (data.elementAt(
                childpos))) < 0))
            {
                childpos++;
            }
```

# Percolate Down

```
                if (((Comparable)(data.elementAt(
                    childpos))).compareTo (value) < 0)
                {
                    data.setElementAt (
                        data.elementAt(childpos), root);
                    root = childpos; // keep moving down
                } else { // found right location
                    data.setElementAt(value, root);
                    return;
                }
            } else { // at a leaf! insert and halt
                data.setElementAt (value, root);
                return;
            }
        }
    }
```

# Percolate Up

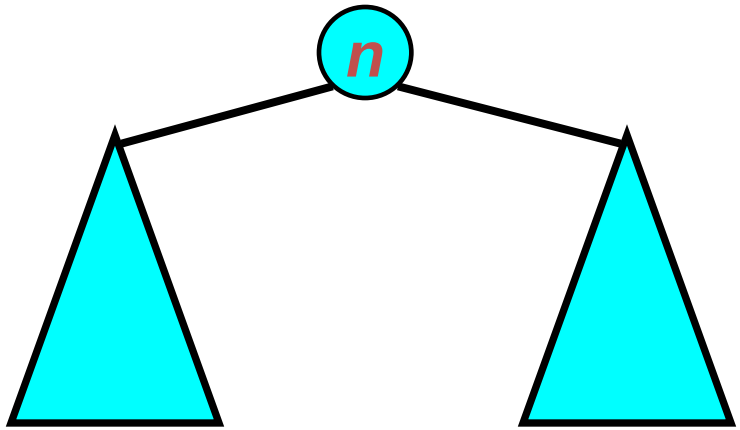
```
protected void percolateUp (int leaf)
{
    int parent = parentOf(leaf);
    Comparable value =
        (Comparable)(data.elementAt(leaf));
    while (leaf > 0 && (value.compareTo
        ((Comparable)(data.elementAt(parent)))
        < 0))
    {
        data.setElementAt (data.elementAt
            (parent), leaf);
        leaf = parent;
        parent = parentOf(leaf);
    }
    data.setElementAt(value,leaf);
}
```

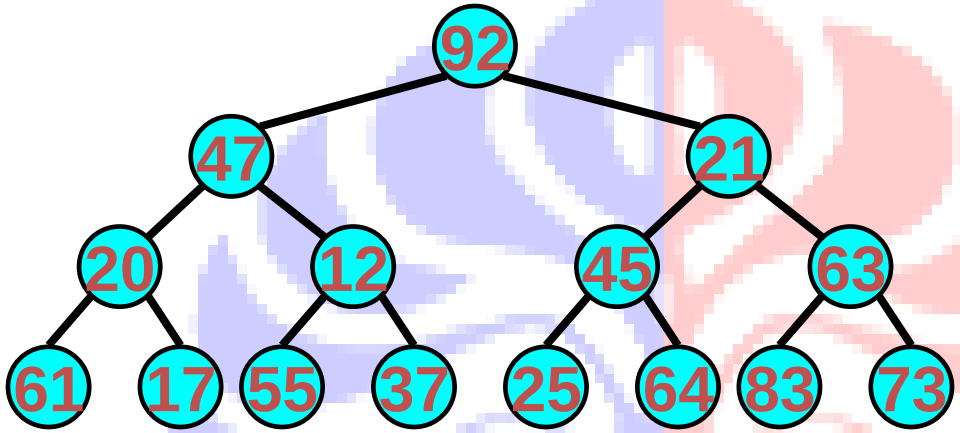
# Heap's Methods

```
public boolean isEmpty()
{
   return data.size() == 0;
}
public int size()
{
    return data.size();
}
public void clear()
{
    data.clear();
}
public String toString()
{
    return "<VectorHeap: " + data + ">";
}
```

# `fixHeap` Algorithm

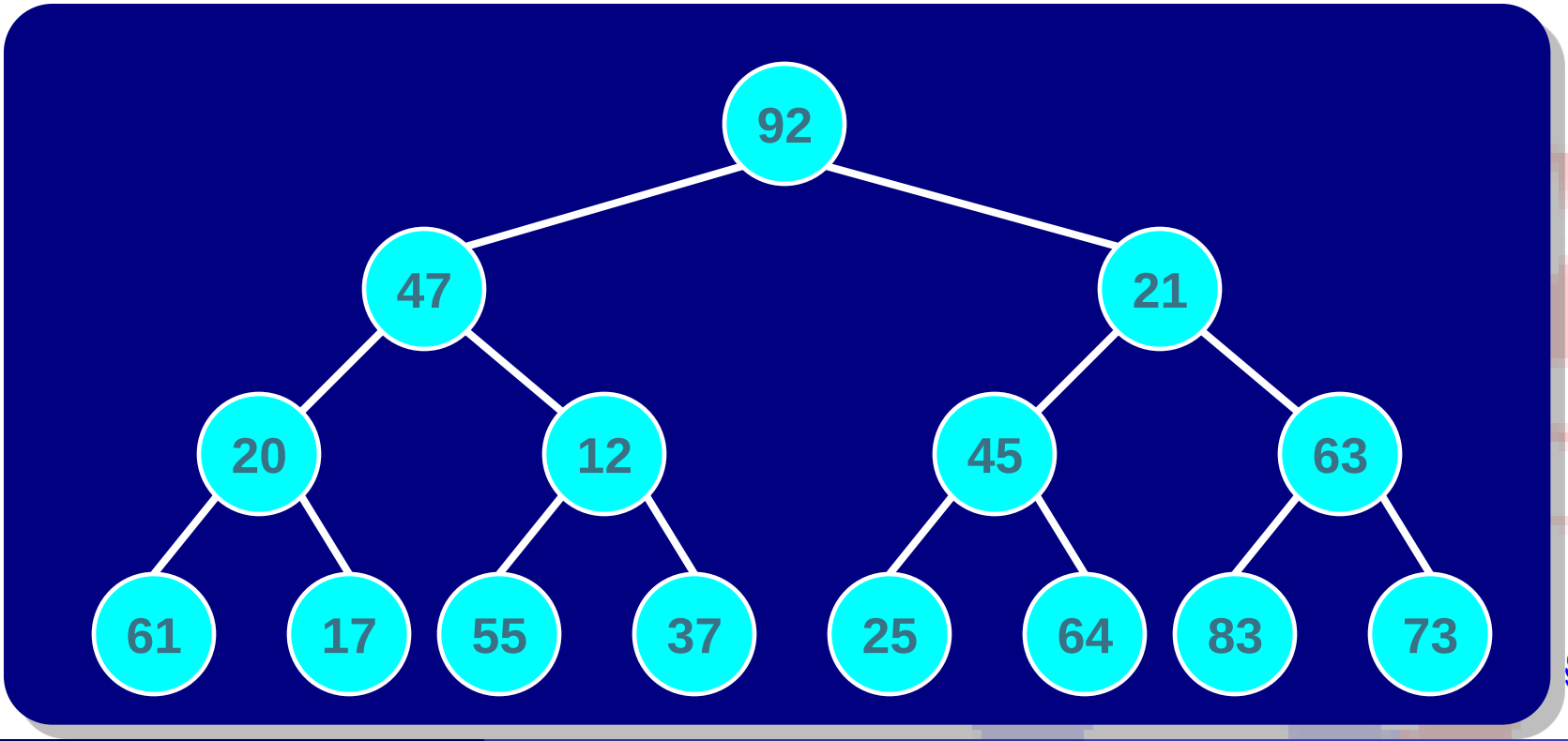
- Versi rekursif:
  - **`fixHeap`** left subtree
  - **`fixHeap`** right subtree
  - **`percolateDown`** root
- Tidak efektif! Ada cara lain yang lebih efisien:
- Panggil `percolateDown` menggunakan *reverse level order* pada *non-leaves node*
- Saat pemanggilan `percolateDown` pada *node n*, sub tree dibawahnya akan dijamin sudah memenuhi *heap order*



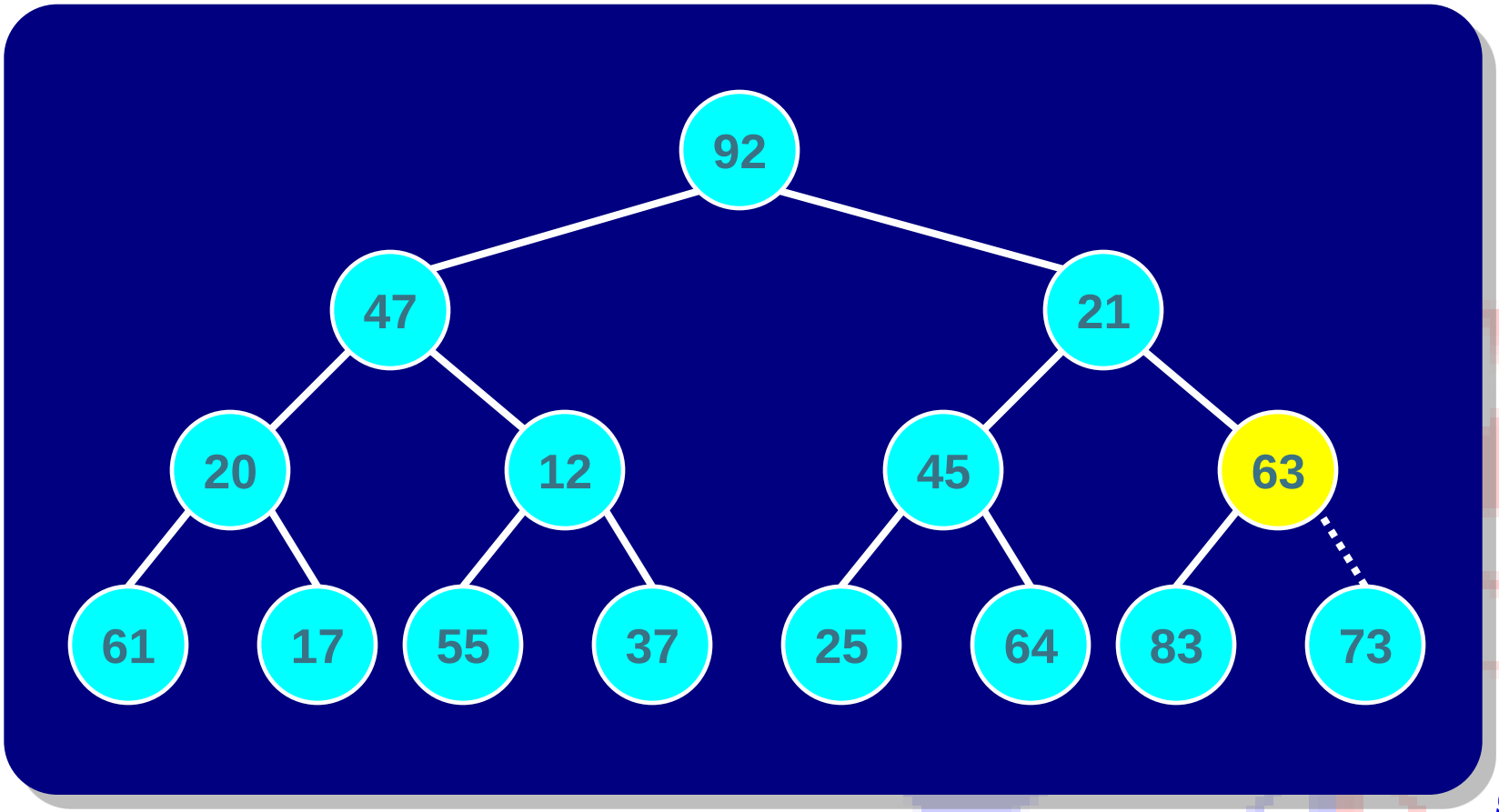
# Fix Heap / Heapify

- Operasi fixHeap menerima *complete tree* yang tidak memenuhi *heap order* dan memperbaikinya.
- penambahan sebanyak N dapat dilakukan dengan O(n log n)
- Operasi fix heap dapat dilakukan dengan O(n) !



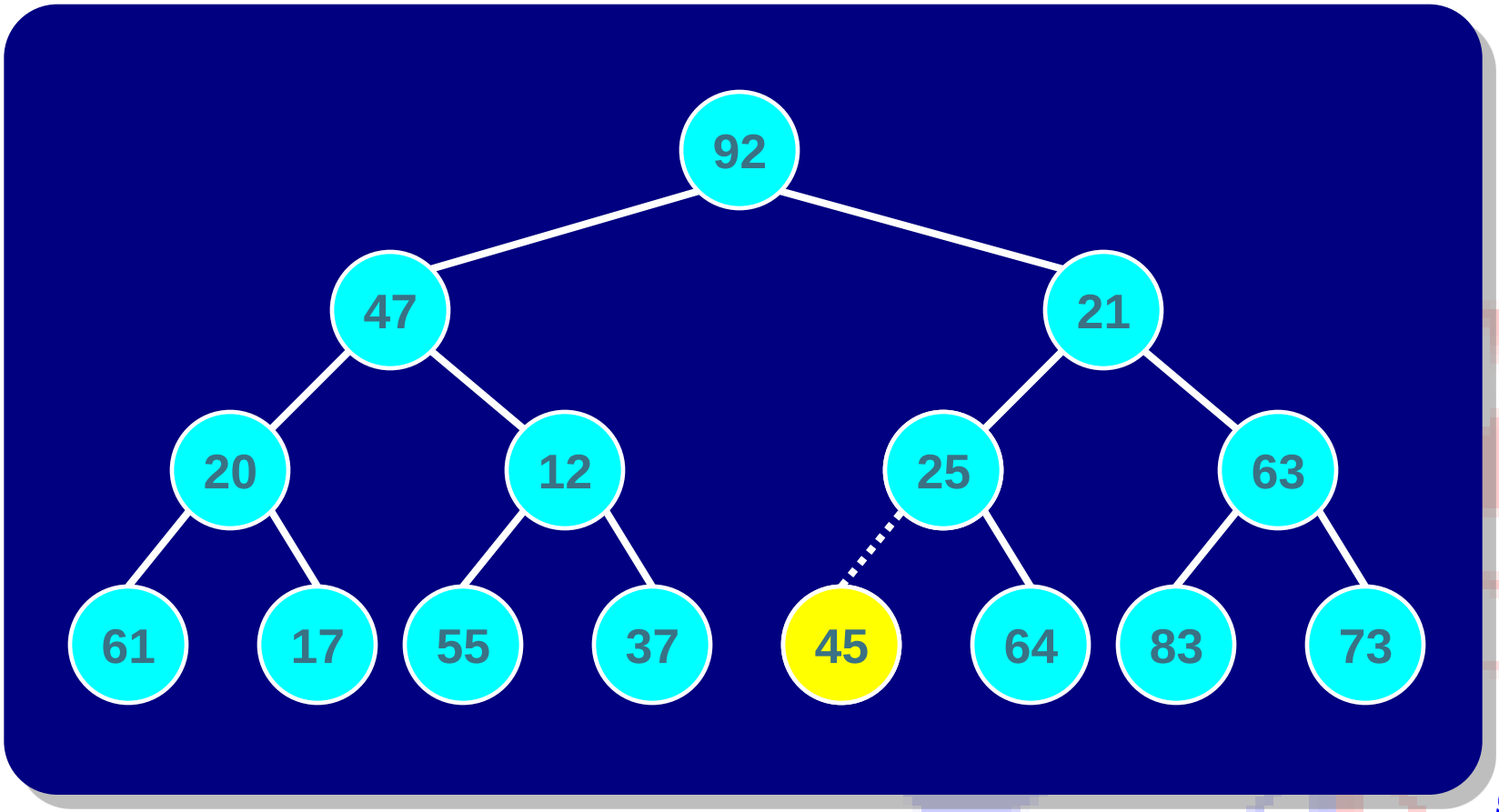
# Fix Heap / Heapify

- Percolate down 6



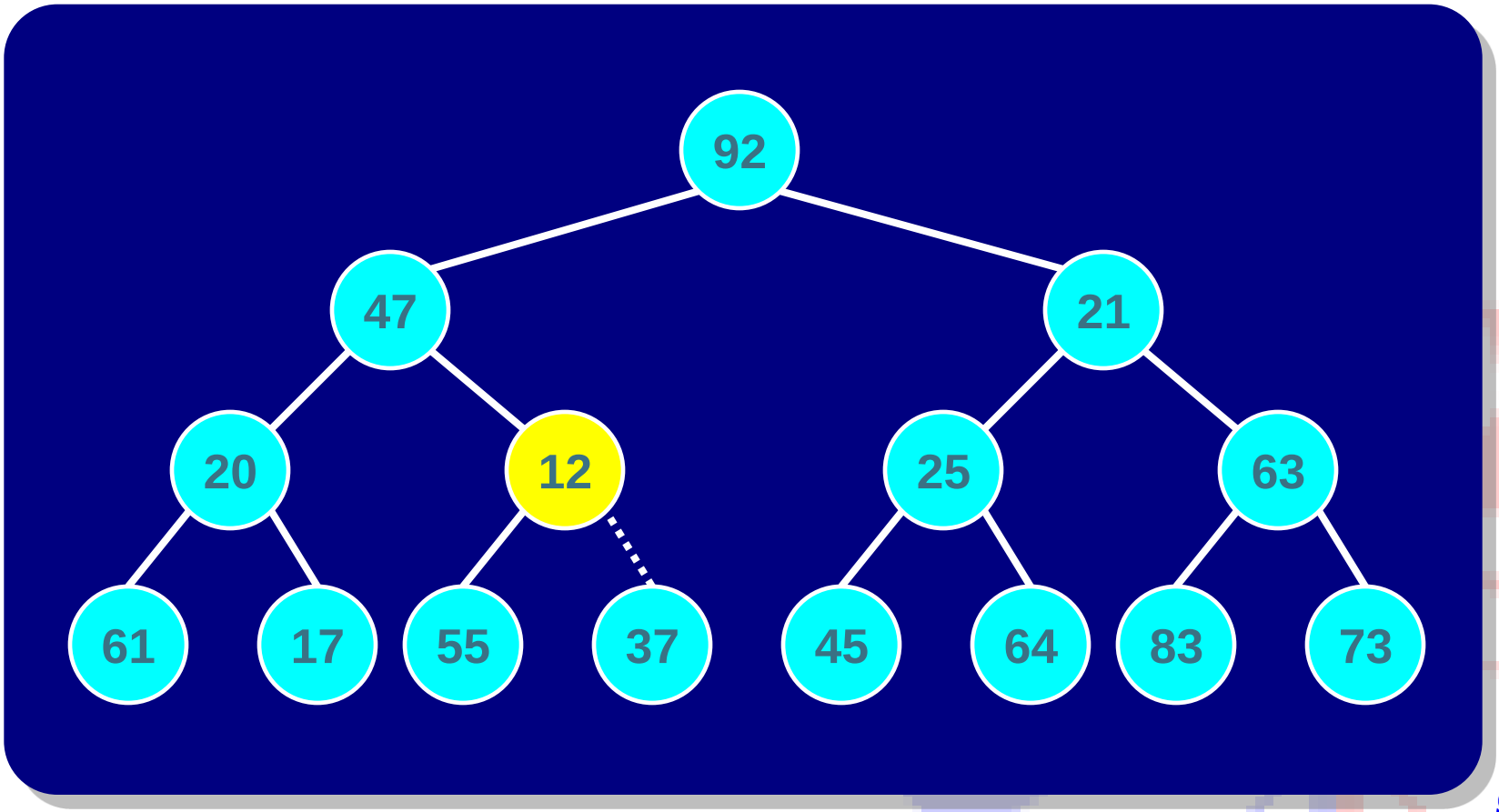
# Fix Heap / Heapify

- Percolate down 5



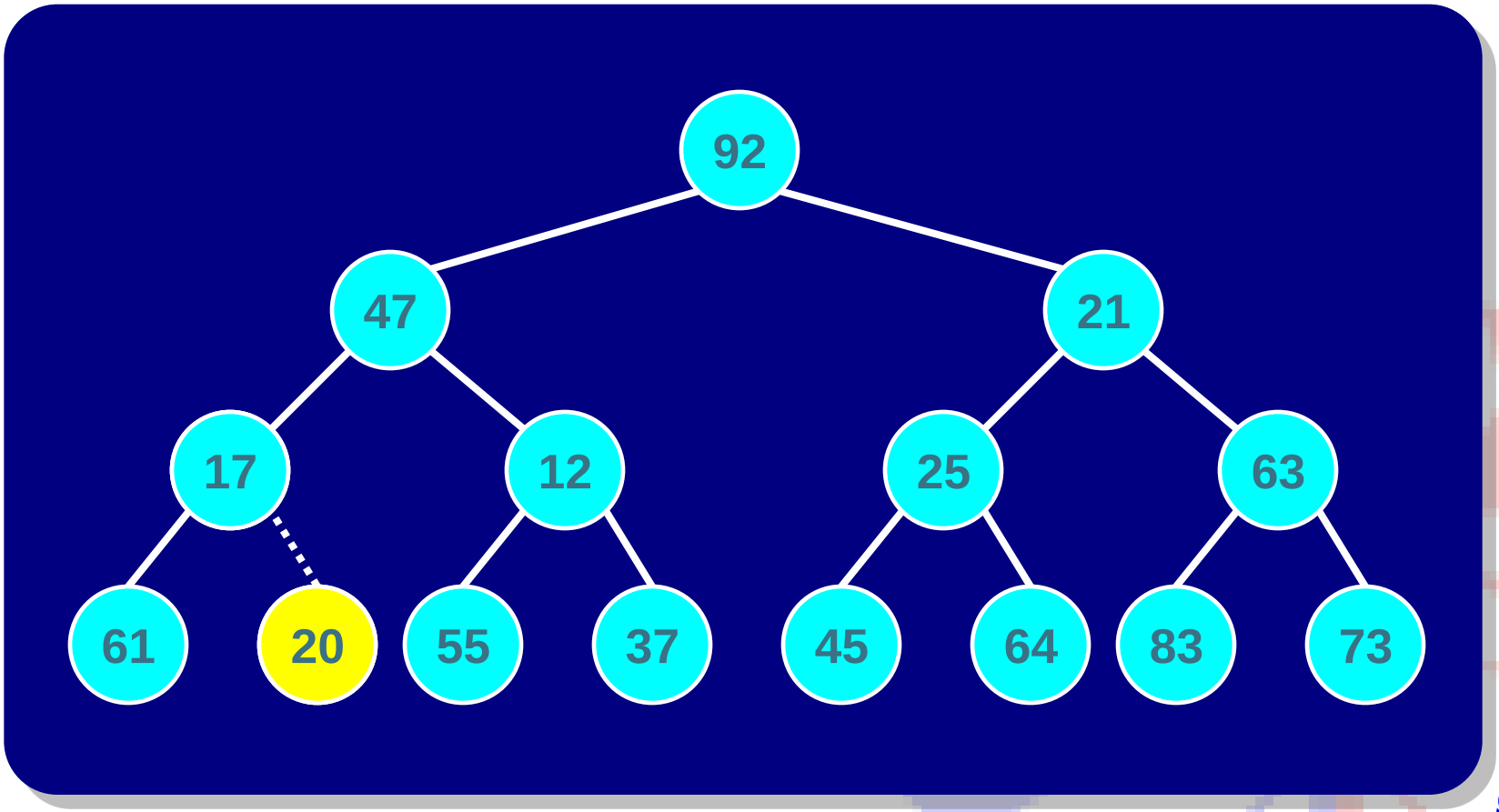
# Fix Heap / Heapify

- Percolate down 4



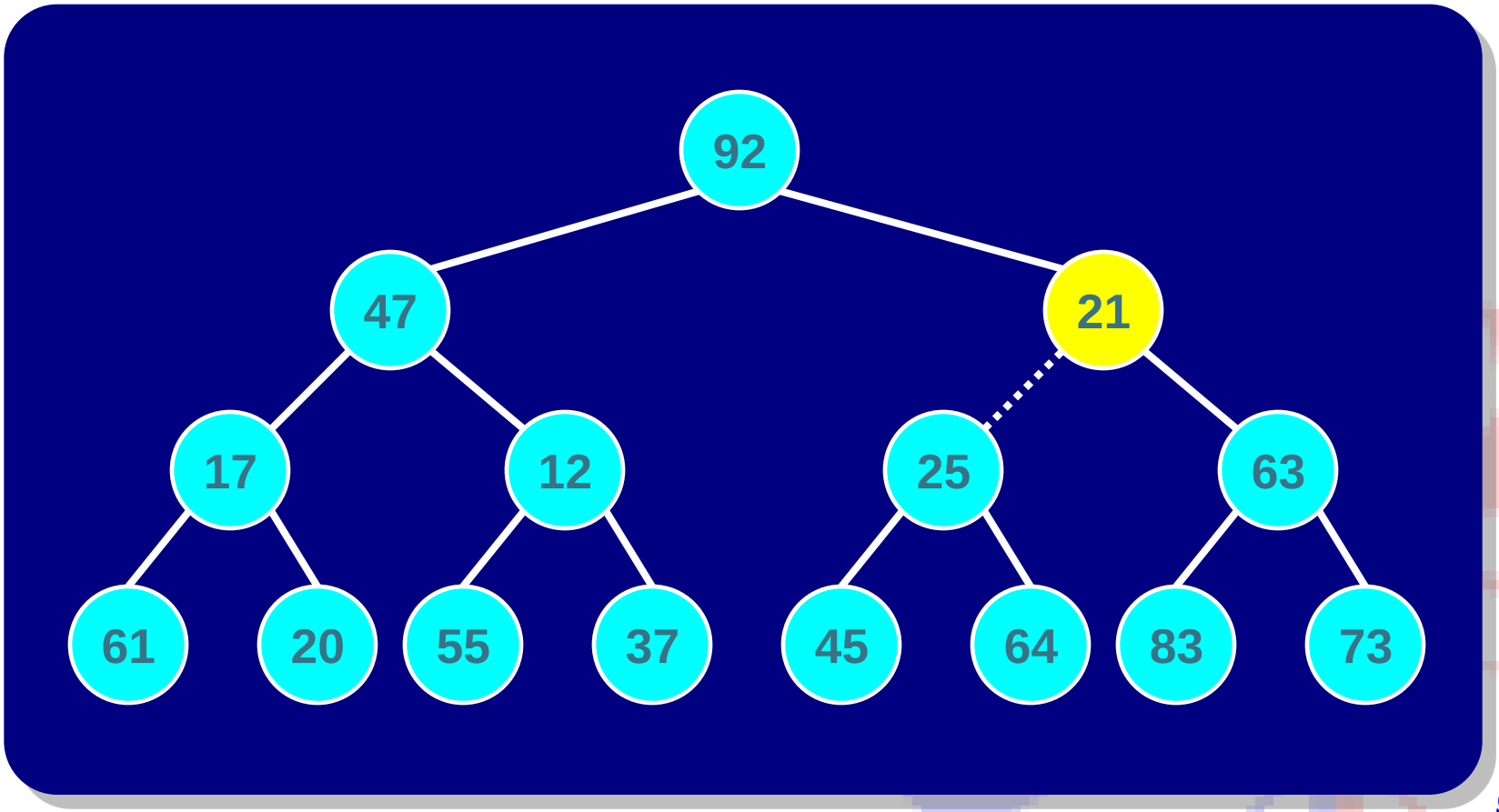
# Fix Heap / Heapify

- Percolate down 3



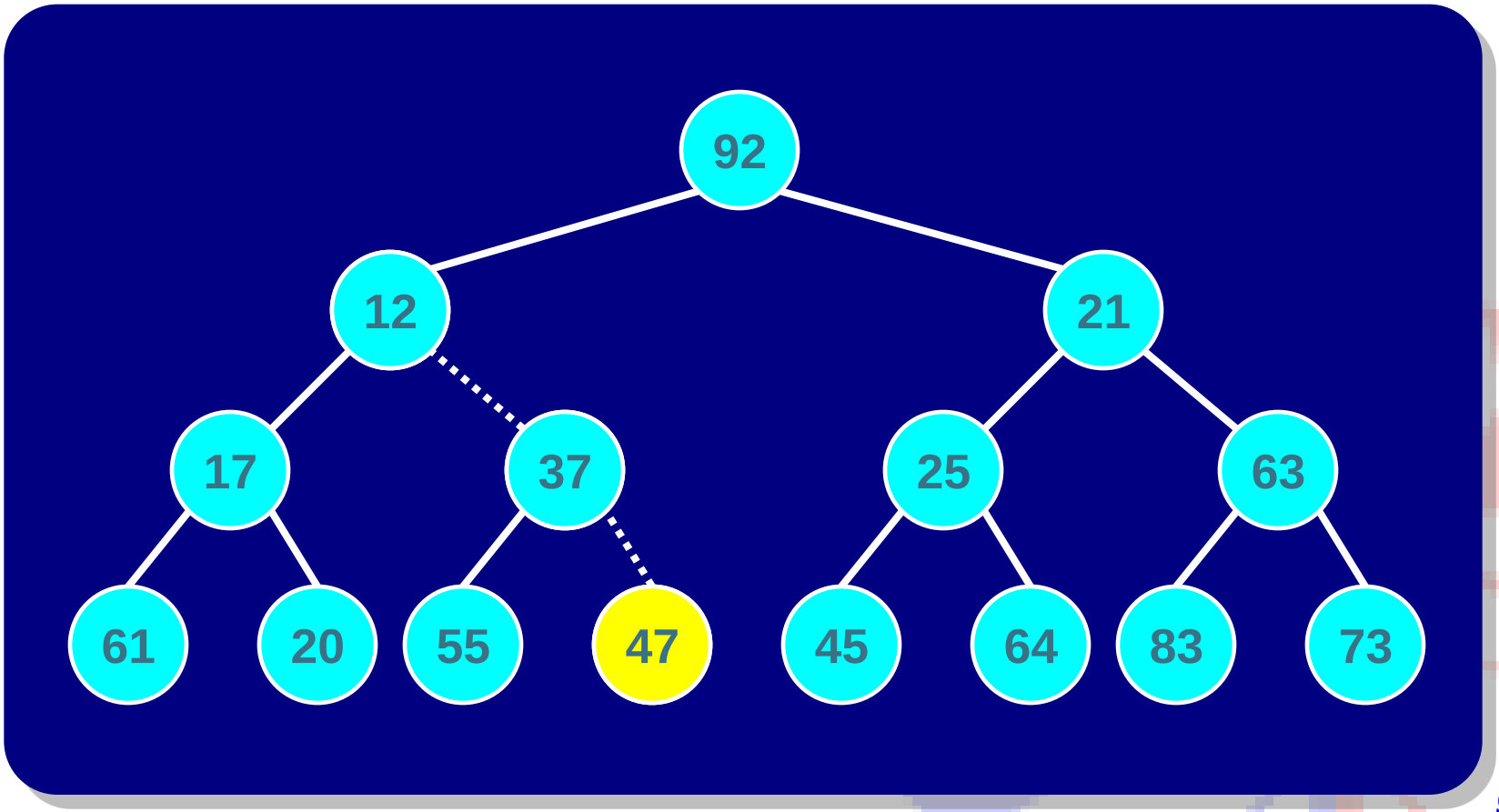
# Fix Heap / Heapify

- Percolate down 2



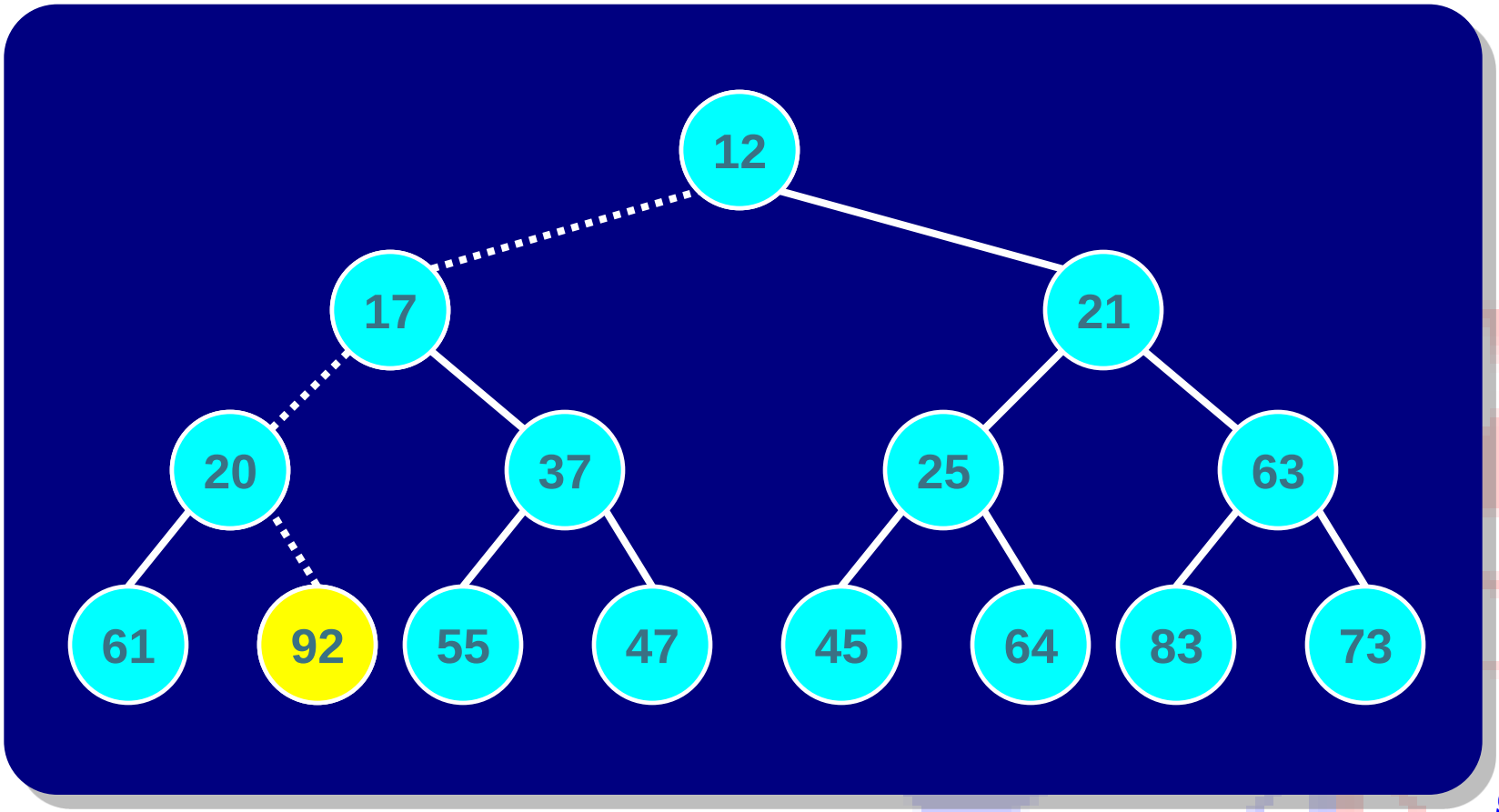
# Fix Heap / Heapify

- Percolate down 1

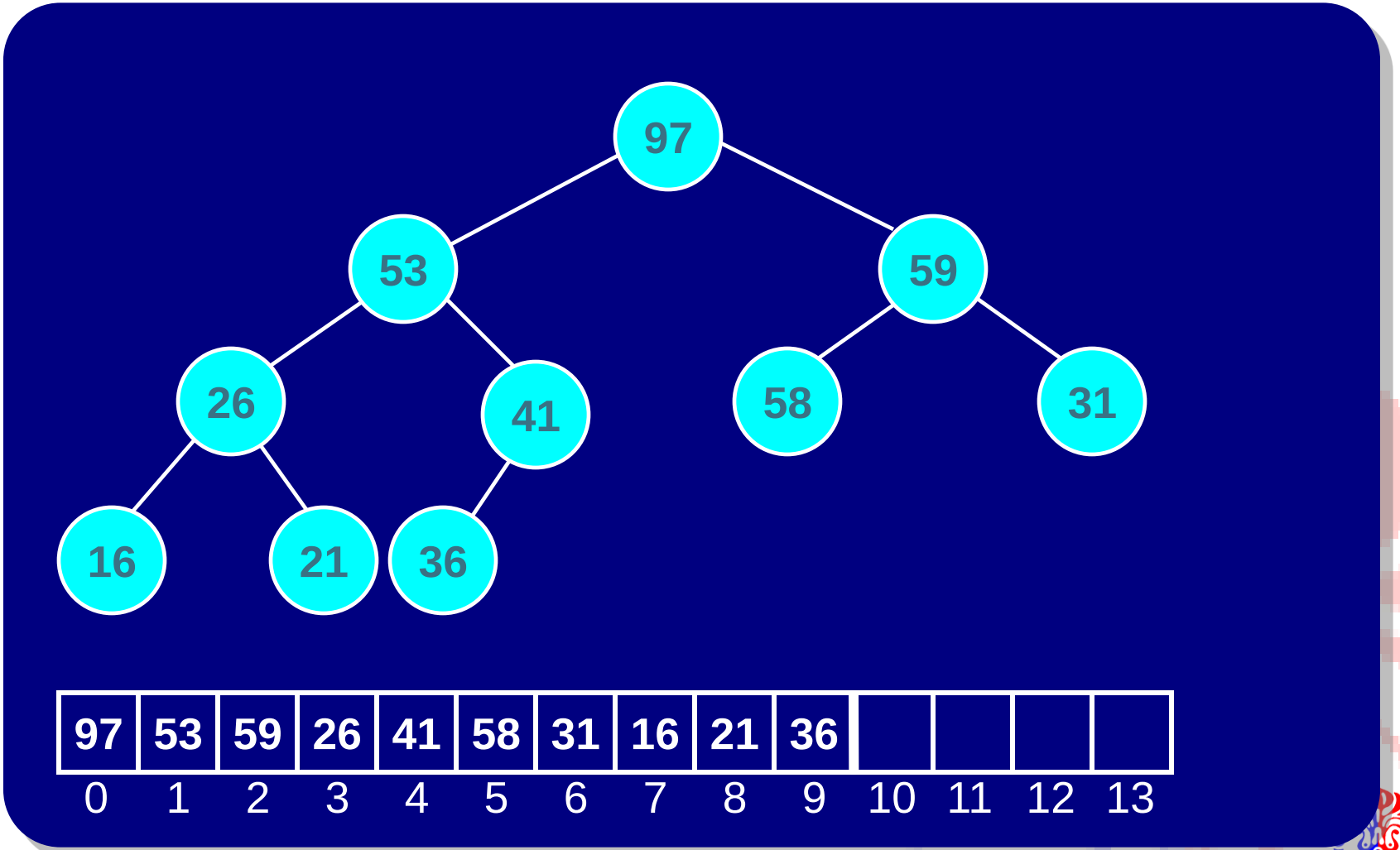


# Fix Heap / Heapify

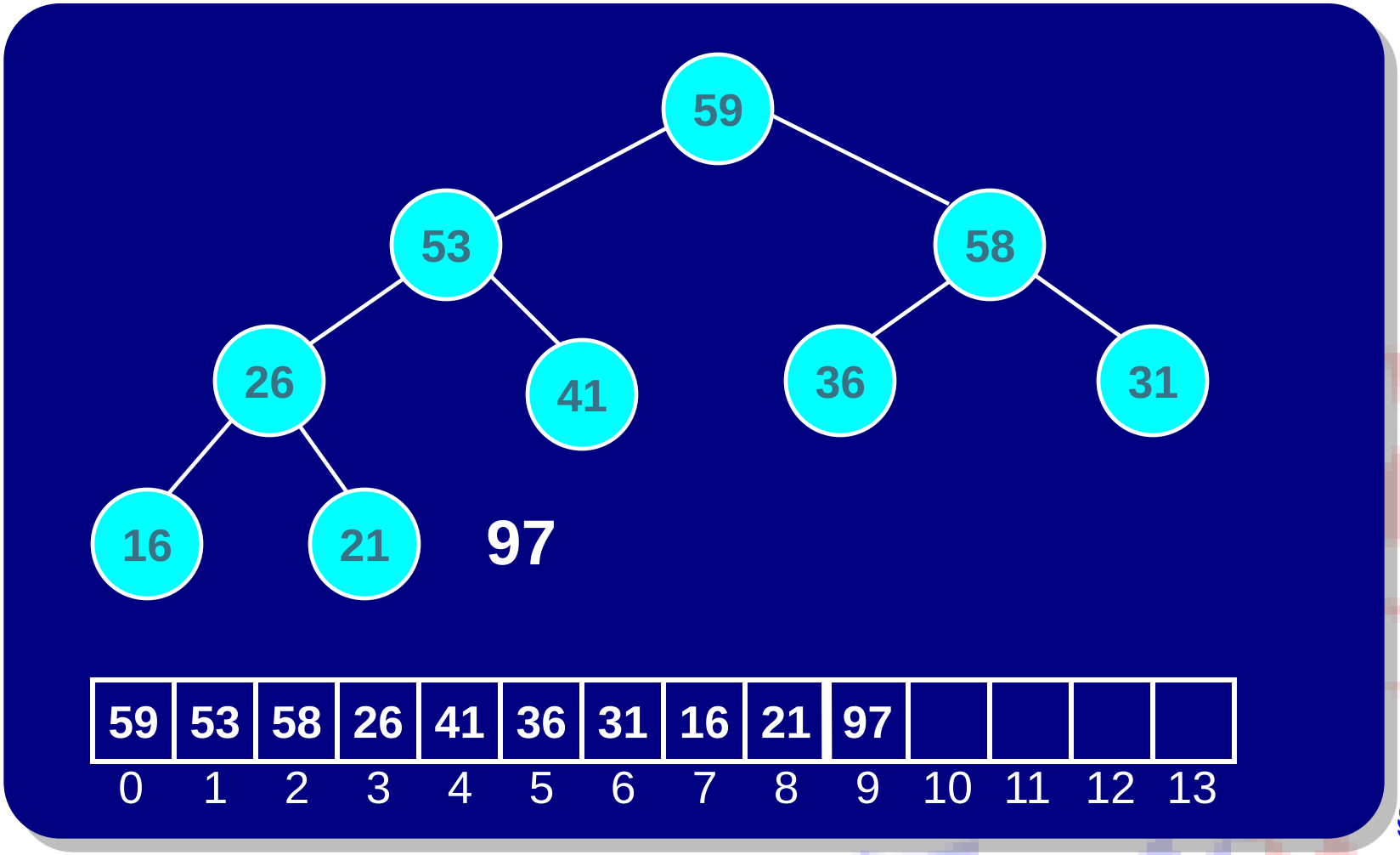
- Percolate down 0



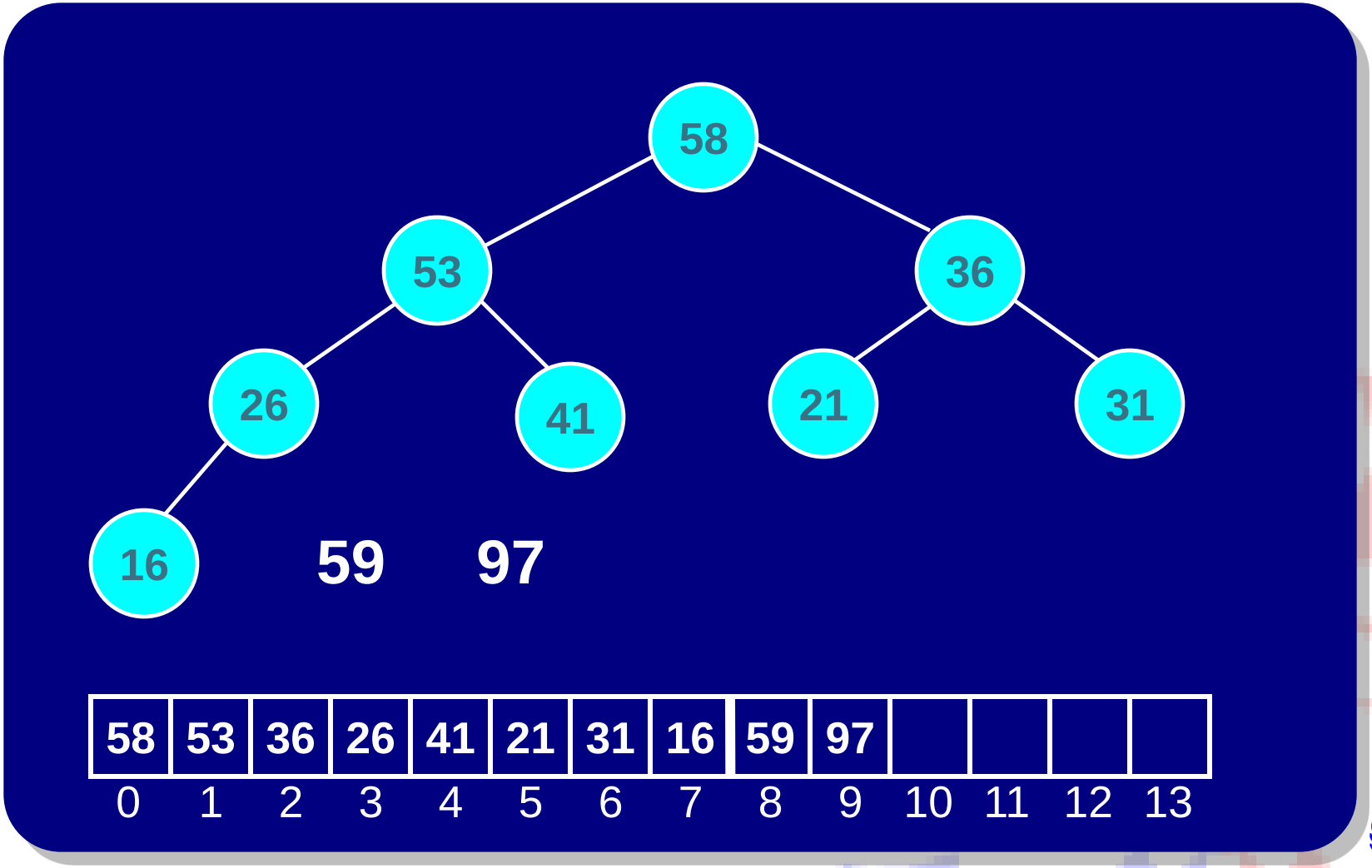
# Max Heap


97
53
59
26
41
58
31
16
21
36
97 53 59 26 41 58 31 16 21 36
0 1 2 3 4 5 6 7 8 9 10 11 12 13

# *Heap* setelah *deleteMax* pertama

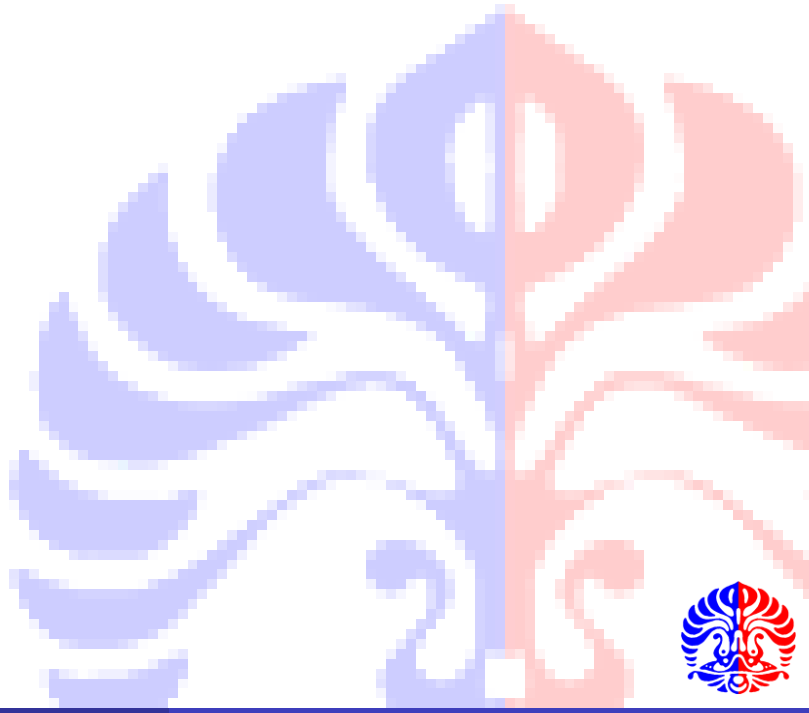


# *Heap* setelah *deleteMax* kedua



# Heap Sort

1. Buat sebuah  heap tree
2. ambil elemen pada posisi root dari heap setiap pengambilan elemen, dan lakukan heapify.



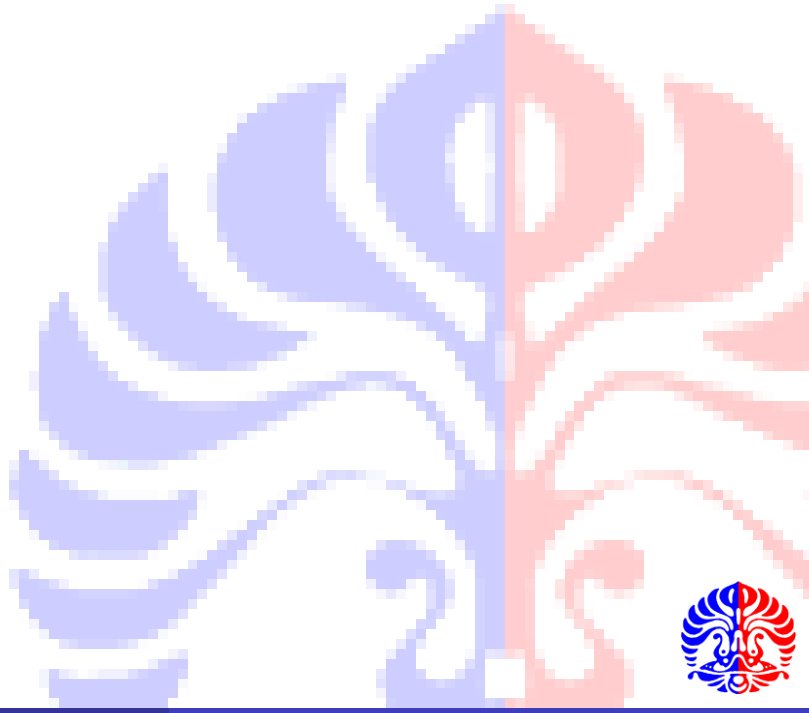
# Rangkuman

- Priority queue dapat diimplementasi kan menggunakan binary heap
- Aturan-aturan pada binary heap
  - structure property
    - complete binary tree
  - ordering property
    - Heap order: Parent $\leq$ Child
- Operasi pada binary heap
  - insertion: kompleksitas waktu *O(log n)* pada worst case
  - find min: kompleksitas waktu *O(1)*
  - delete min: kompleksitas waktu *O(log n)* pada worst case
- Binary heap dapat digunakan untuk *sorting*

# HUFFMAN CODE

# Outline

- Compression
- Huffman compression



# Kompresi (*Compression*)

- Proses:
  - *Encoding*: raw → compressed
  - *Decoding*: compressed → raw
- Jenis Kompresi
  - *Lossy*: data yang dikompres kemungkinan tidak memiliki kualitas yang sama persis dengan data asli.
    - Contoh: MPEG, JPEG, MP3
  - *Lossless:* data yang dikompresi dapat dikembalikan menjadi data asli tanpa kehilangan kualitas data.
    - Contoh: zip, GIF, wav
- Beberapa algoritma kompresi:
  - RLE: Run Length Encoding
  - Lempel-Zif
  - Huffman Encoding
- Tingkat kompresi bergantung pada jenis file.

# Huffman Compression: Motivation

- Perhatikan teks berikut:

  **`If a woodchuck could chuck wood!`**

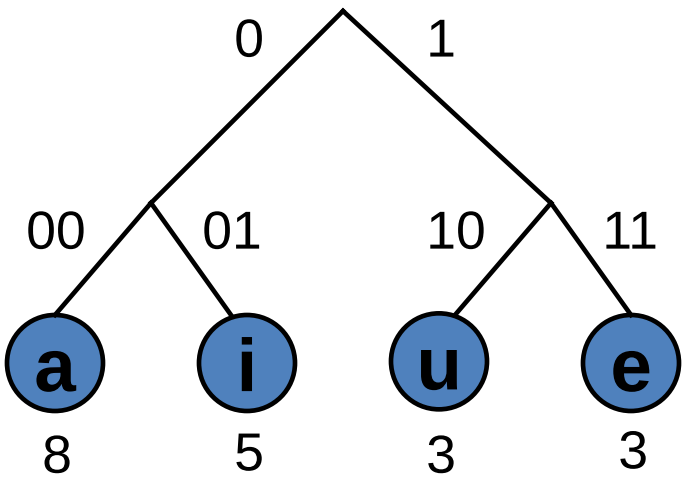
- Terdiri dari 32 karakter, masing-masing karakter diencode dengan 8 bit ASCII code. Untuk menyimpan kalimat diatas membutuhkan 256 bit.
  - 32 char $\times$ 8 bit = 256 bits
- Teori:
  - Untuk meng-encode ***N*** karakter yang berbeda dibutuhkan $\boldsymbol{log_2 N}$ bit
- Observasi:
  - Ada 13 karakter berbeda $\rightarrow$ 4 bit
  - Kalimat diatas dapat di kompres sehingga hanya membutuhkan 128 bits
- Apakah menggunakan encoding dengan panjang bit berbeda dapat memperbaiki tingkat kompresi ?

# Huffman Compression: Ide

- Untuk karakter yang sering muncul, gunakan representasi encoding yang pendek.
- Untuk karakter yang jarang muncul, gunakan representasi encoding yang panjang.
- Bila panjang bit encoding tiap karakter beda-beda, bagaimana memisahkan sekumpulan bit encoding dengan bit encoding karakter lain?
- Prefix code: Tidak ada code encoding sebuah karakter yang merupakan prefix dari code encoding karakter lain.
- Prefix code dimodelkan dengan tree, data hanya pada leaves !

# Huffman Encoding: perbandingan

- Menggunakan encoding dengan panjang bits sama.
- Ada 4 karakter, masing-masing di encode dengan 2 bits.
- Karakter ***a*** muncul 8 kali, ***i*** 5 kali, ***u*** dan ***e*** masing-masing 3 kali.
- Panjang bit yang dibutuhkan: ***42 bit***




a = 00 → 16 bits

i = 01 → 10 bits

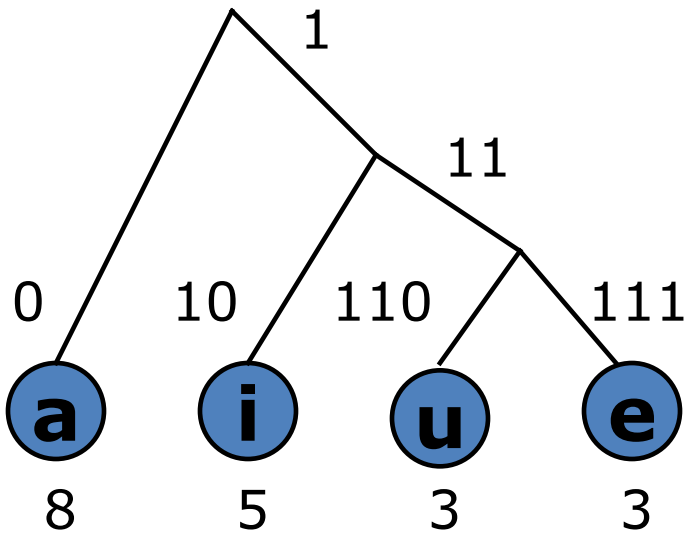
u = 10 → 6 bits

e = 11 → 6 bits

**Total : 38 bits**

# Huffman Encoding: perbandingan

- Menggunakan encoding dengan panjang bits berbeda-beda.
- Karakter ***a*** di encode dengan binary code: **0**,
  ***i*** dengan **10**, ***u*** dengan **110**, dan ***e*** dengan **111**.
- Panjang bit yang dibutuhkan: ***36 bit***
- ***Kesimpulan: lebih baik dari pada encoding yang fixed length.***

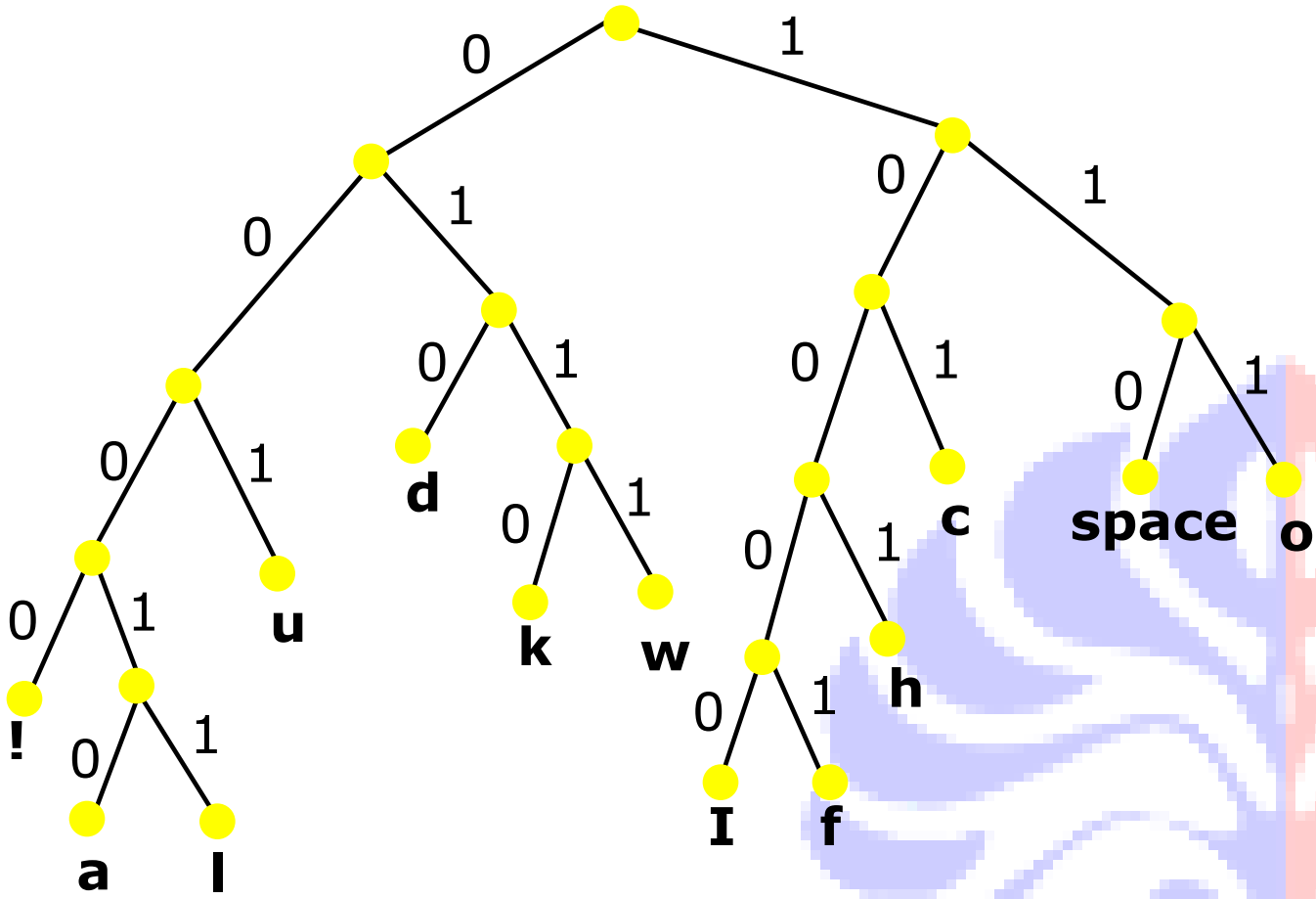



a = 0 → 8 bits
i = 10 → 10 bits
u = 110 → 9 bits
e = 111 → 9 bits
**Total: 36 bits**

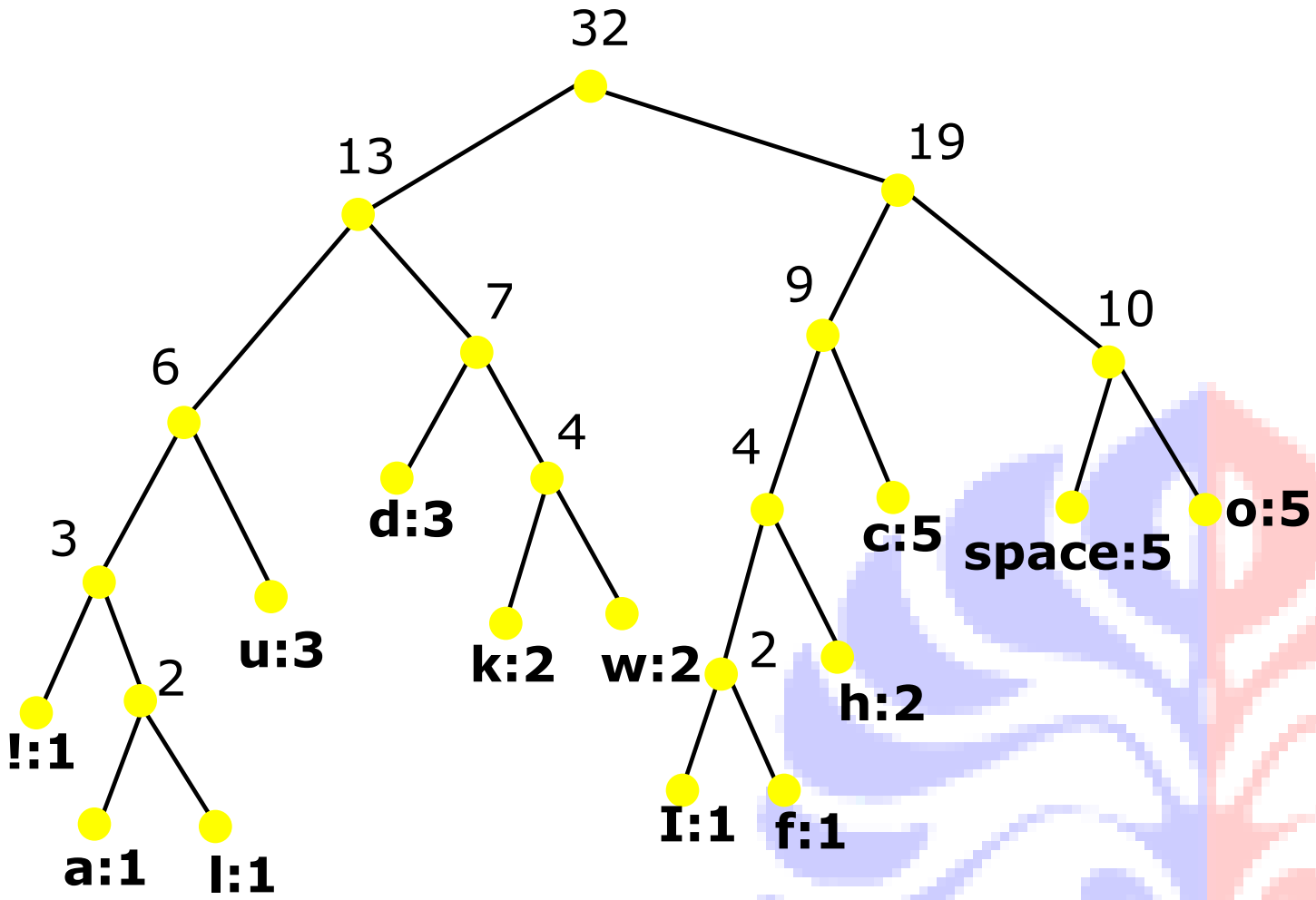
# Huffman Code

- Algoritma Huffman digunakan untuk menggenerate prefix code.
- Prefix code: Tidak ada code encoding sebuah karakter yang merupakan prefix dari code encoding karakter lain.
- Prefix code dimodelkan dengan tree, data hanya pada leaves !
- Walau algoritma Huffman hanya salah satu algoritma untuk menghasilkan prefix code, istilah *huffman code* lebih umum dikenal dari pada prefix code.

# Huffman Encoding



# Huffman Encoding

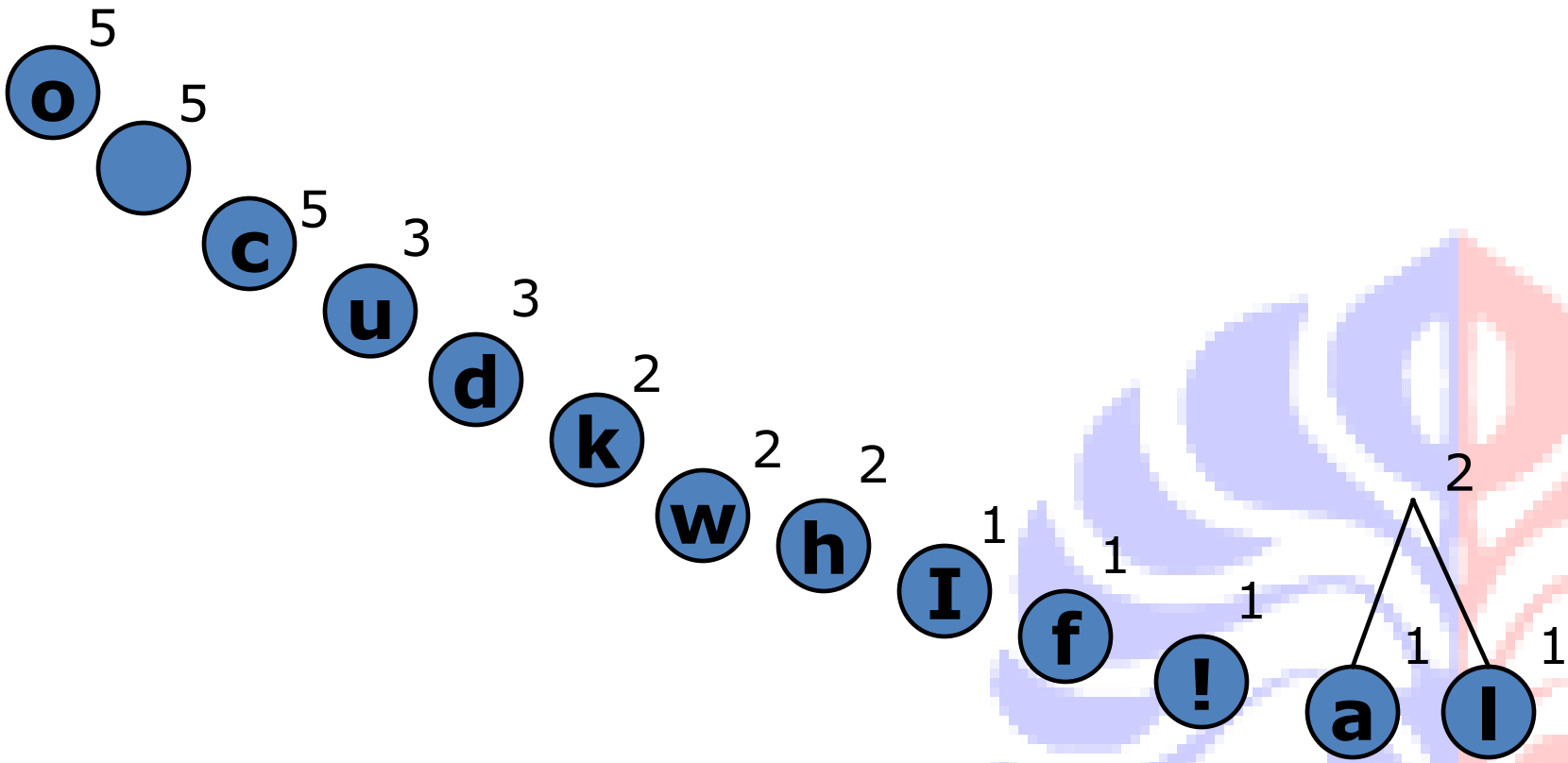


# Huffman Encoding (freq)

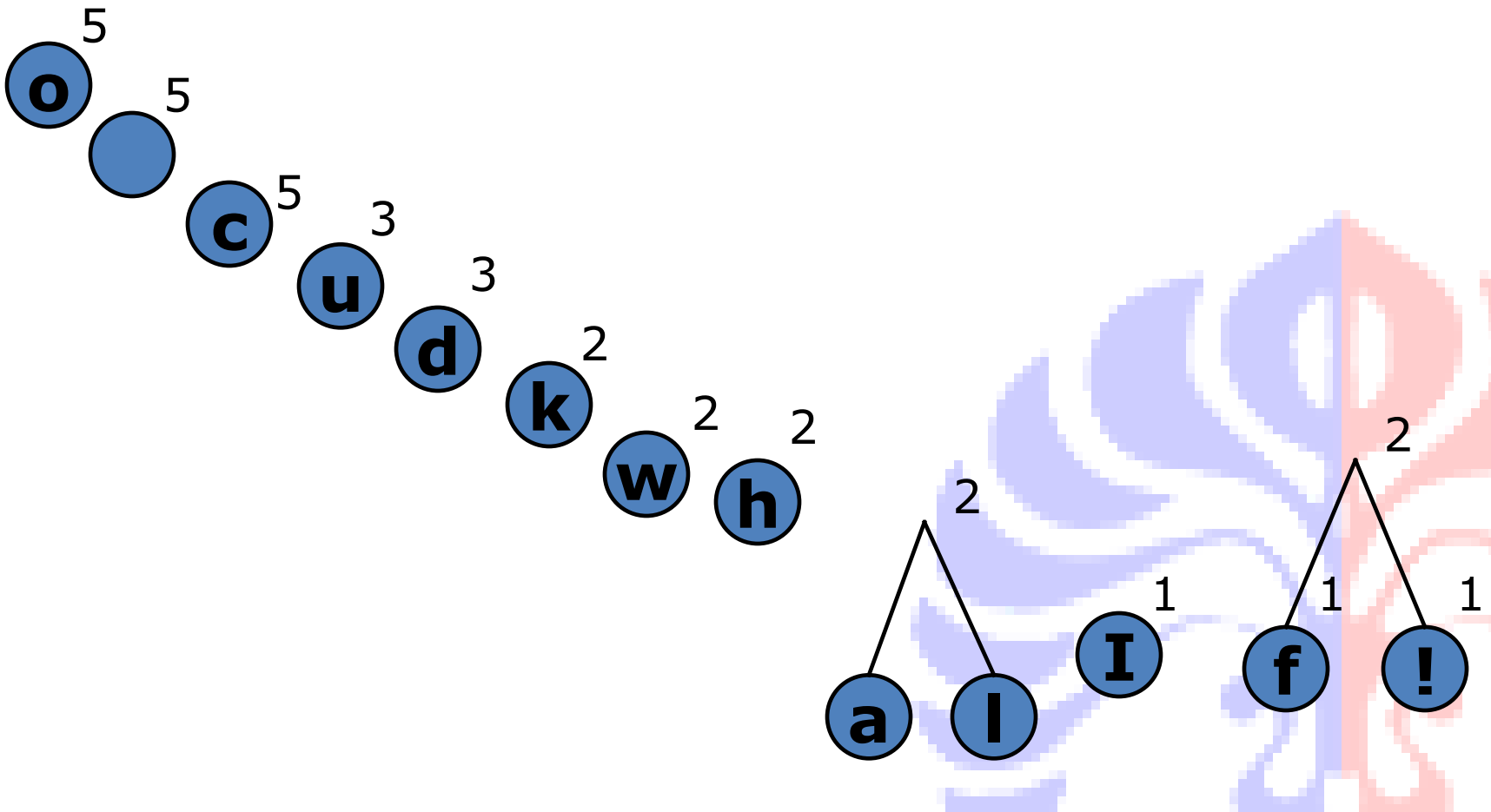
| | | | |
|---|---|---|---|
| ! | = 0000 (1) | I | = 10000 (1) |
| a | = 00010 (1) | f | = 10001 (1) |
| l | = 00011 (1) | h | = 1001 (2) |
| u | = 001 (3) | c | = 101 (5) |
| d | = 010 (3) | space | = 110 (5) |
| k | = 0110 (2) | o | = 111 (5) |
| w | = 0111 (2) | | |

Cost: $\sum d_i * f_i$ = 111 bits = 44% $\times$ 256 bits

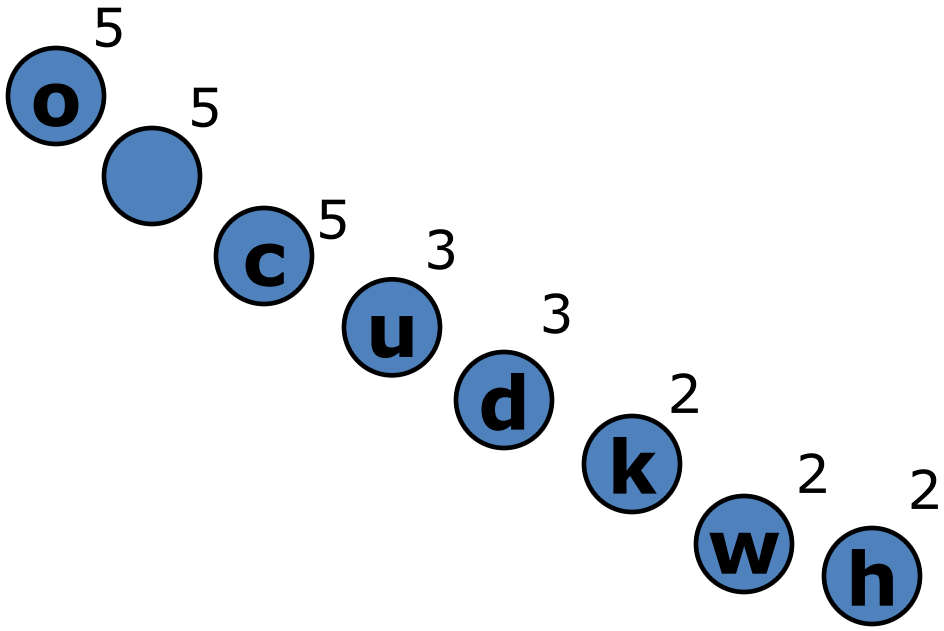
# Huffman Encoding: langkah-langkah

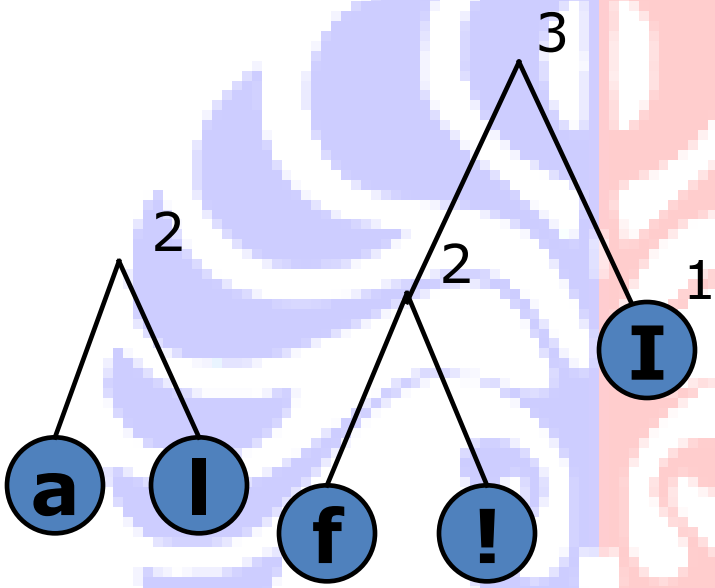


# Huffman Encoding: langkah-langkah

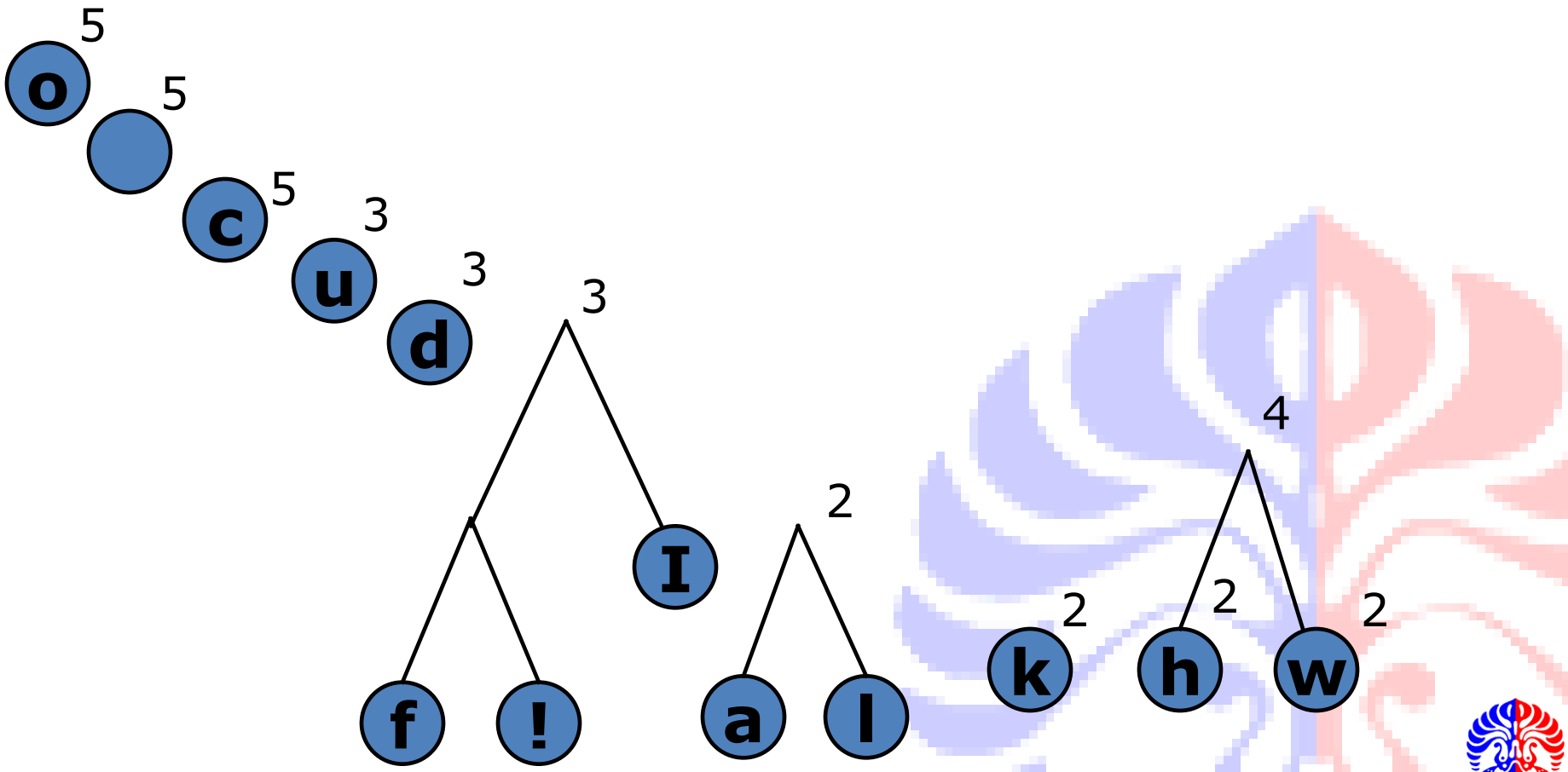


# Huffman Encoding: langkah-langkah

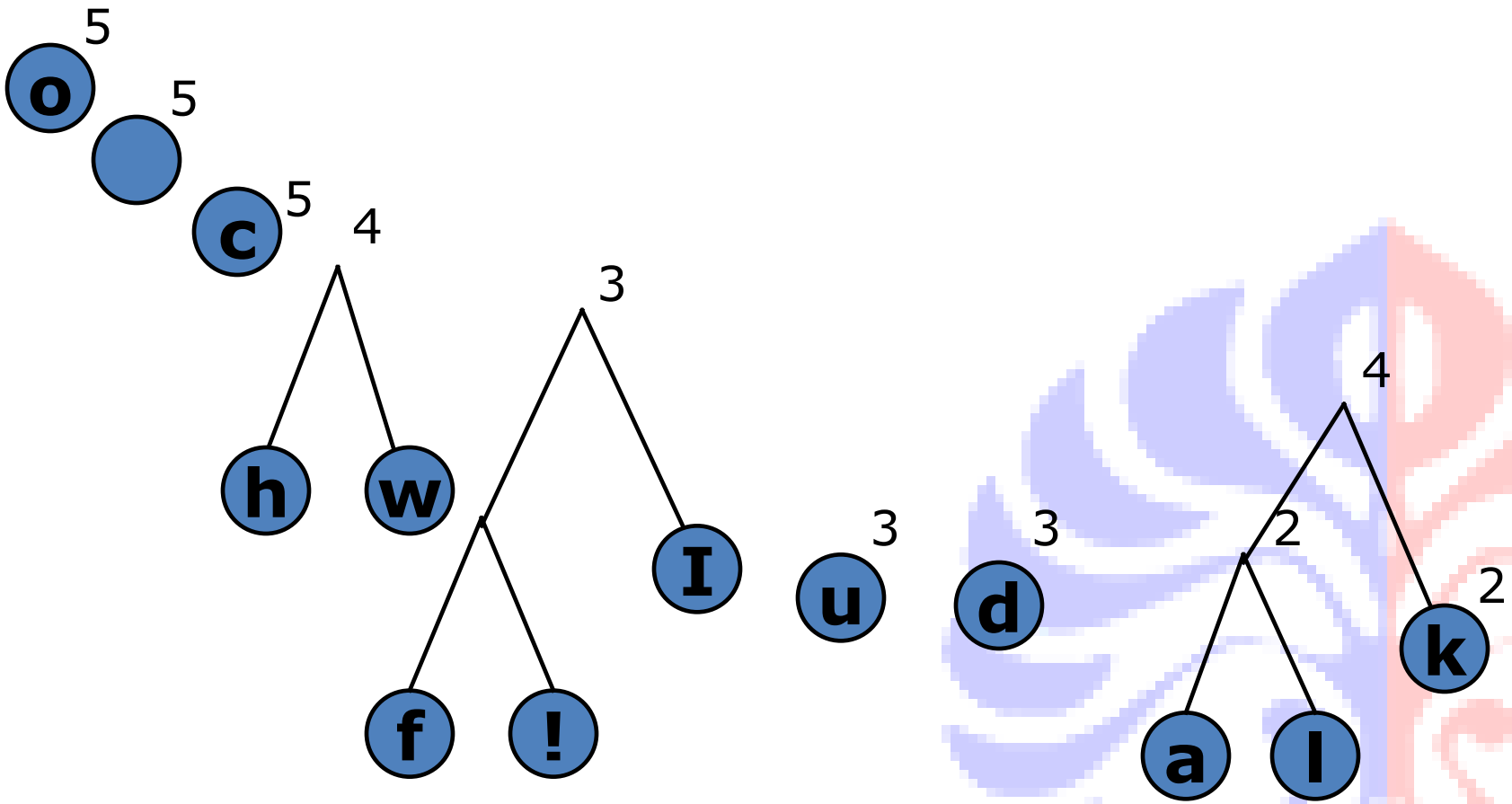


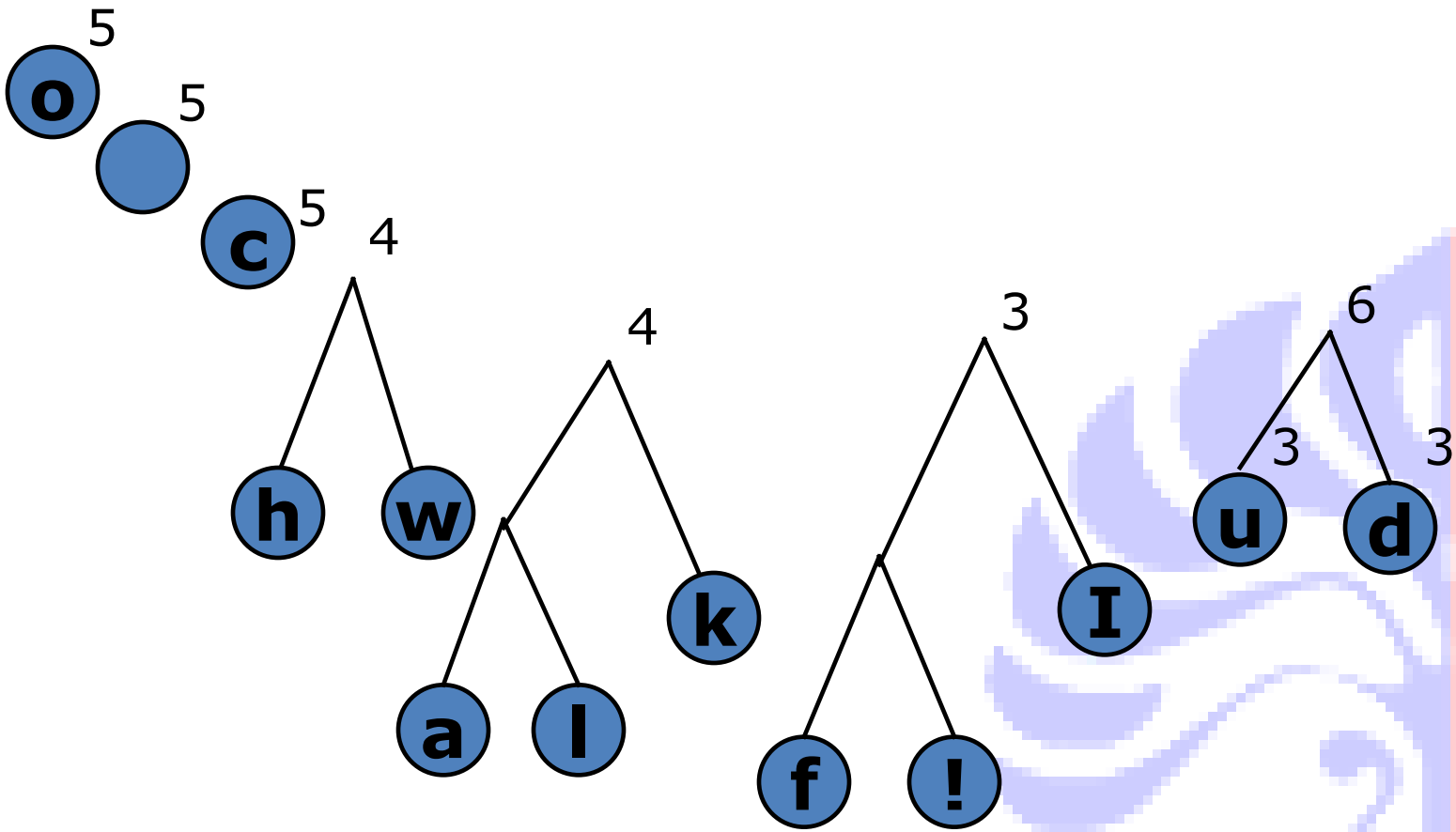
# Huffman Encoding: langkah-langkah



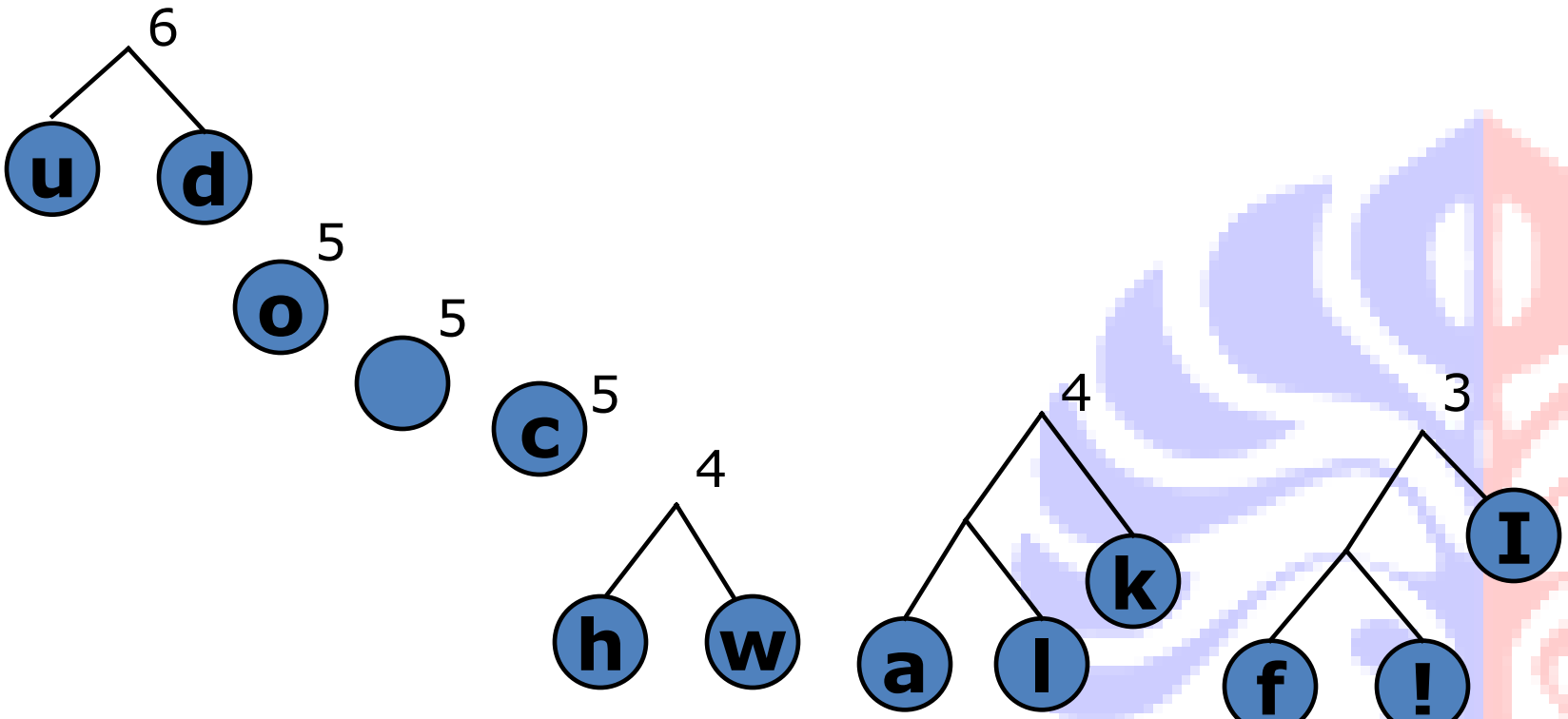
# Huffman Encoding: langkah-langkah



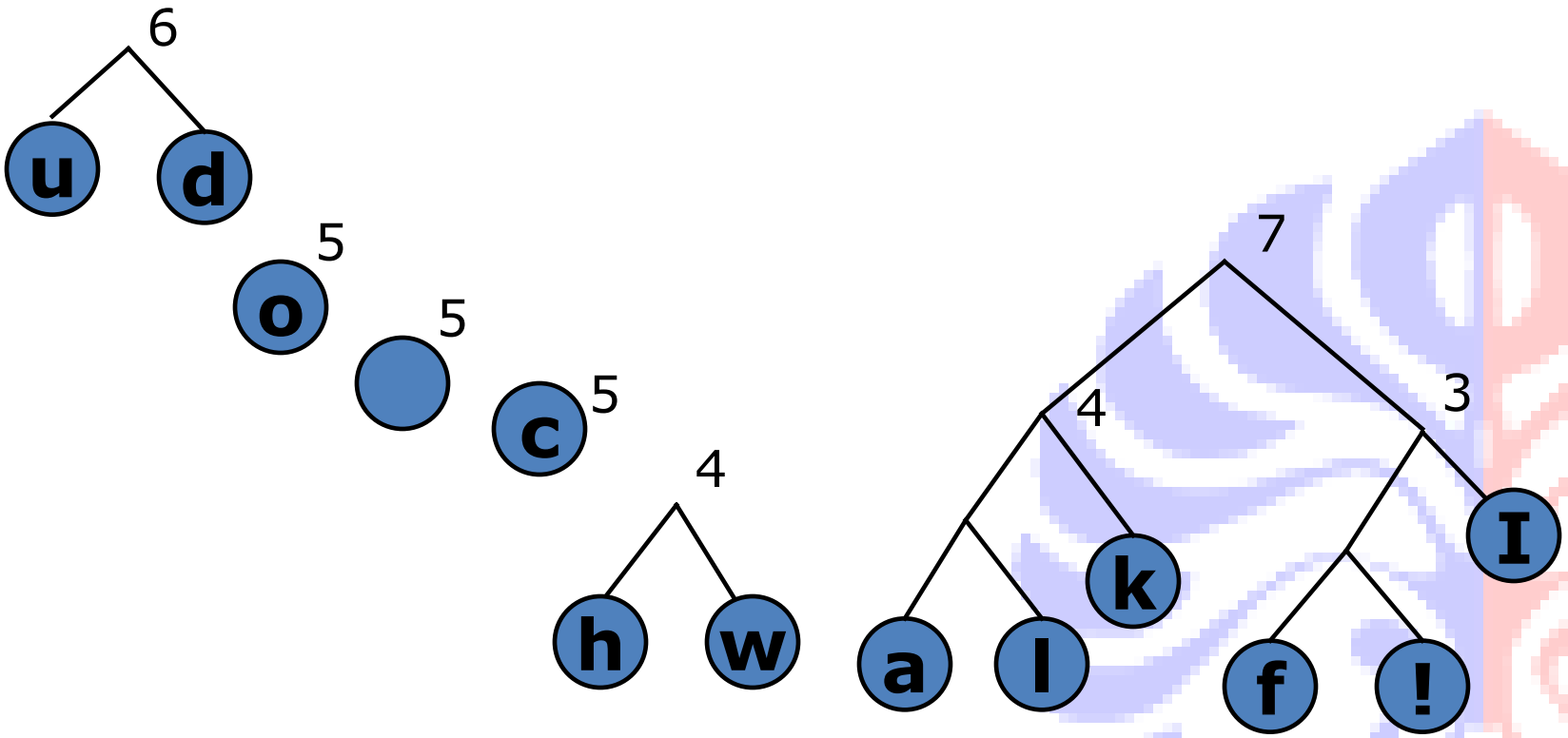
# Huffman Encoding: langkah-langkah



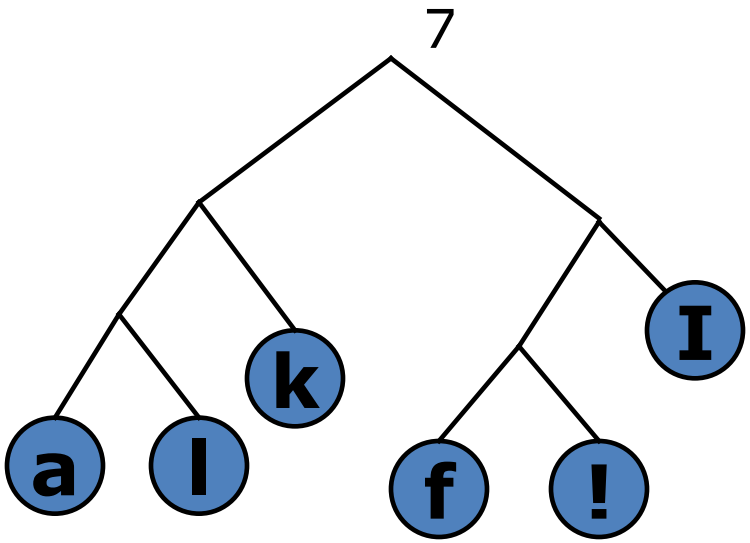
# Huffman Encoding: langkah-langkah

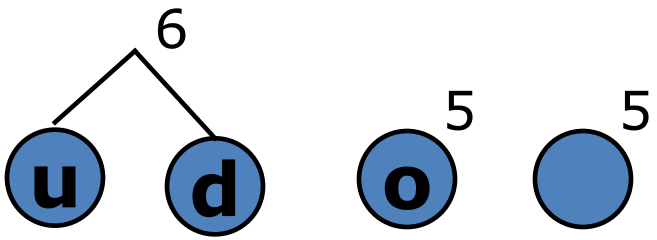


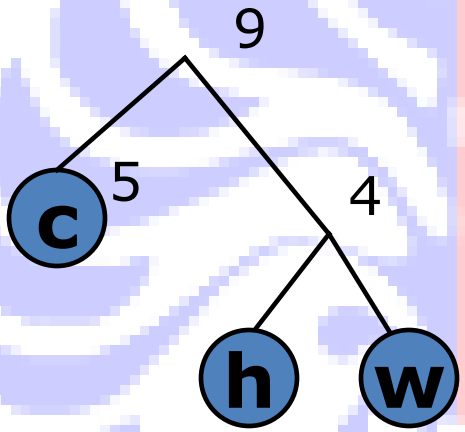
# Huffman Encoding: steps



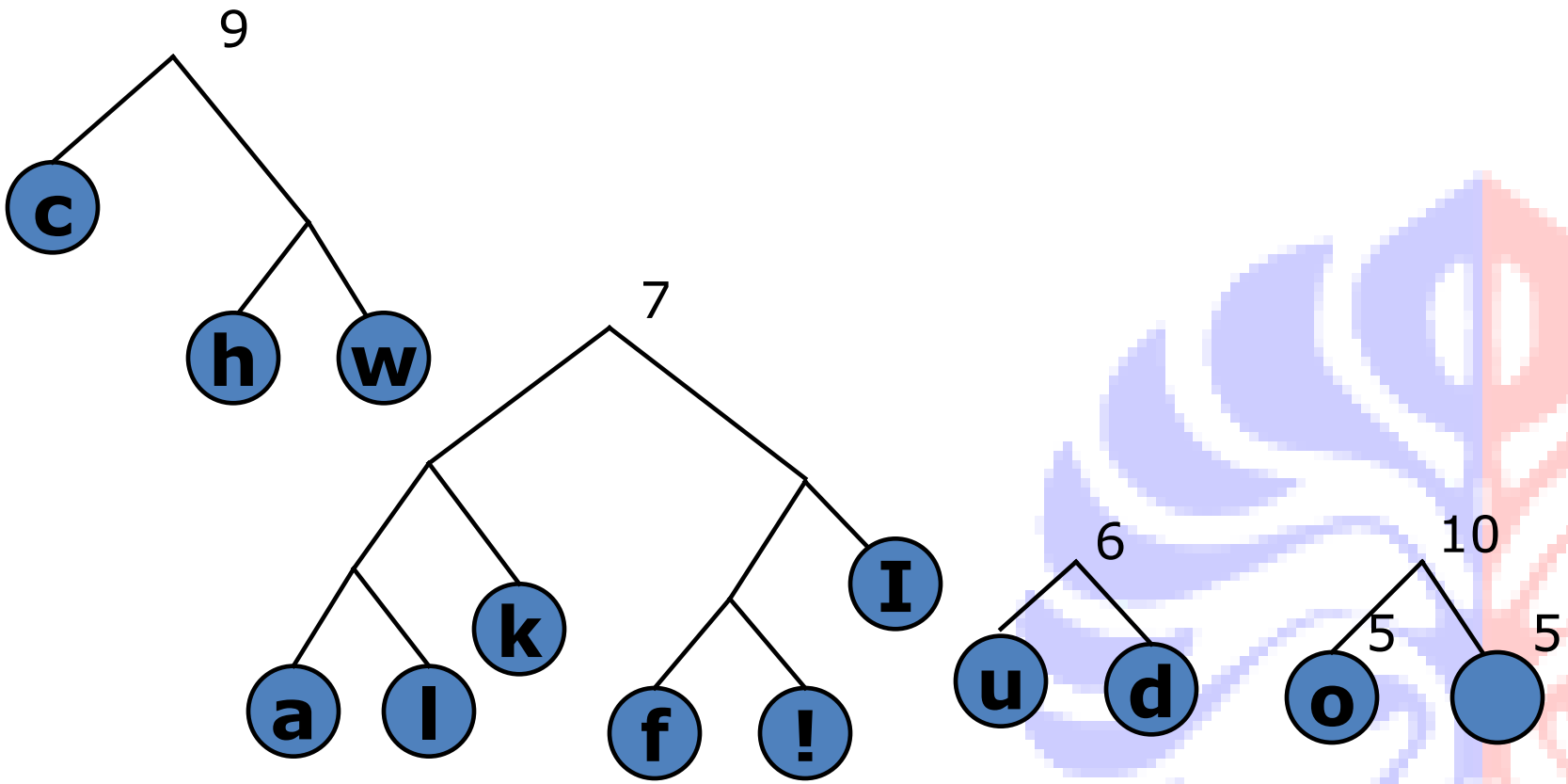
# Huffman Encoding: langkah-langkah



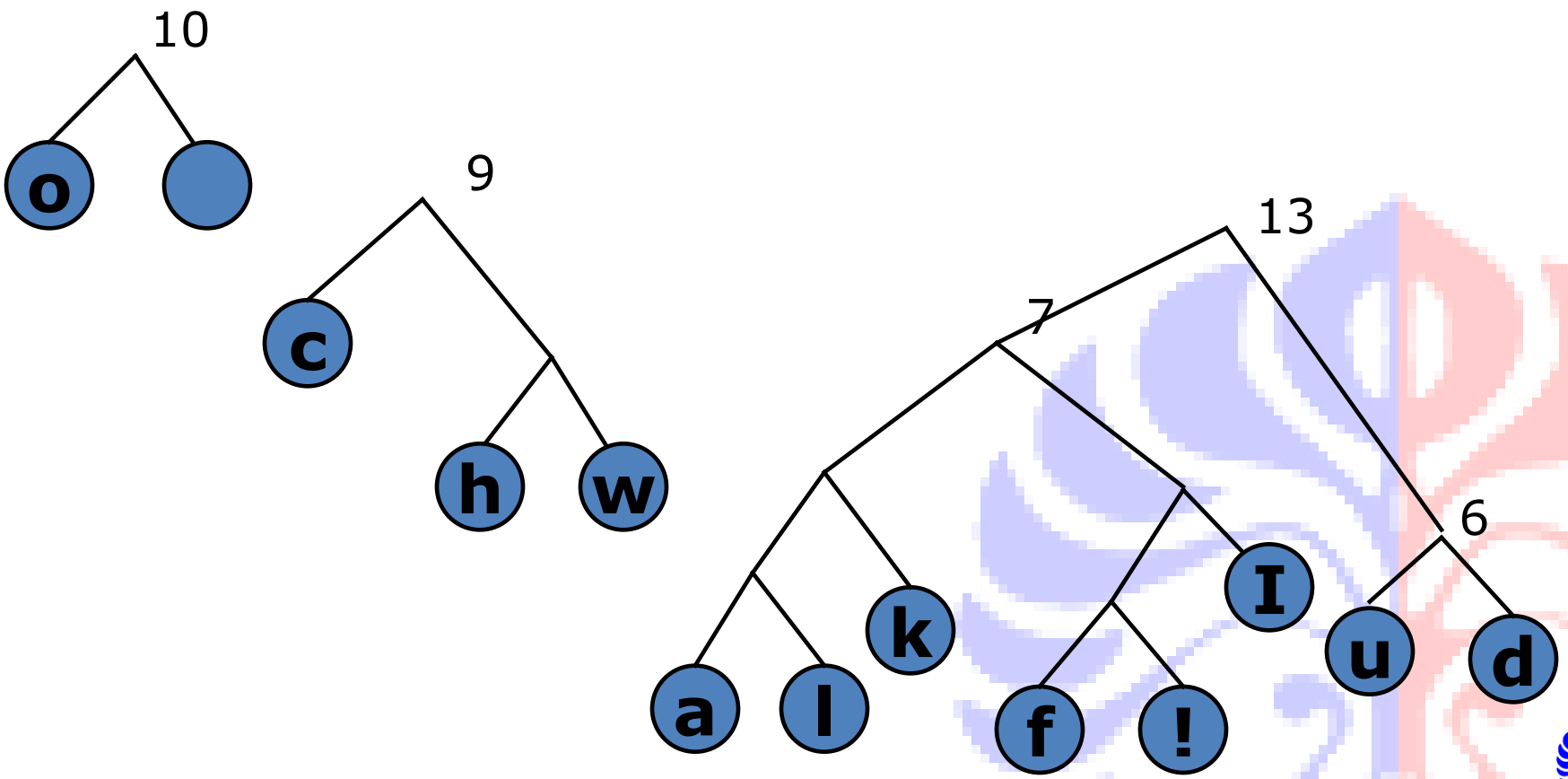



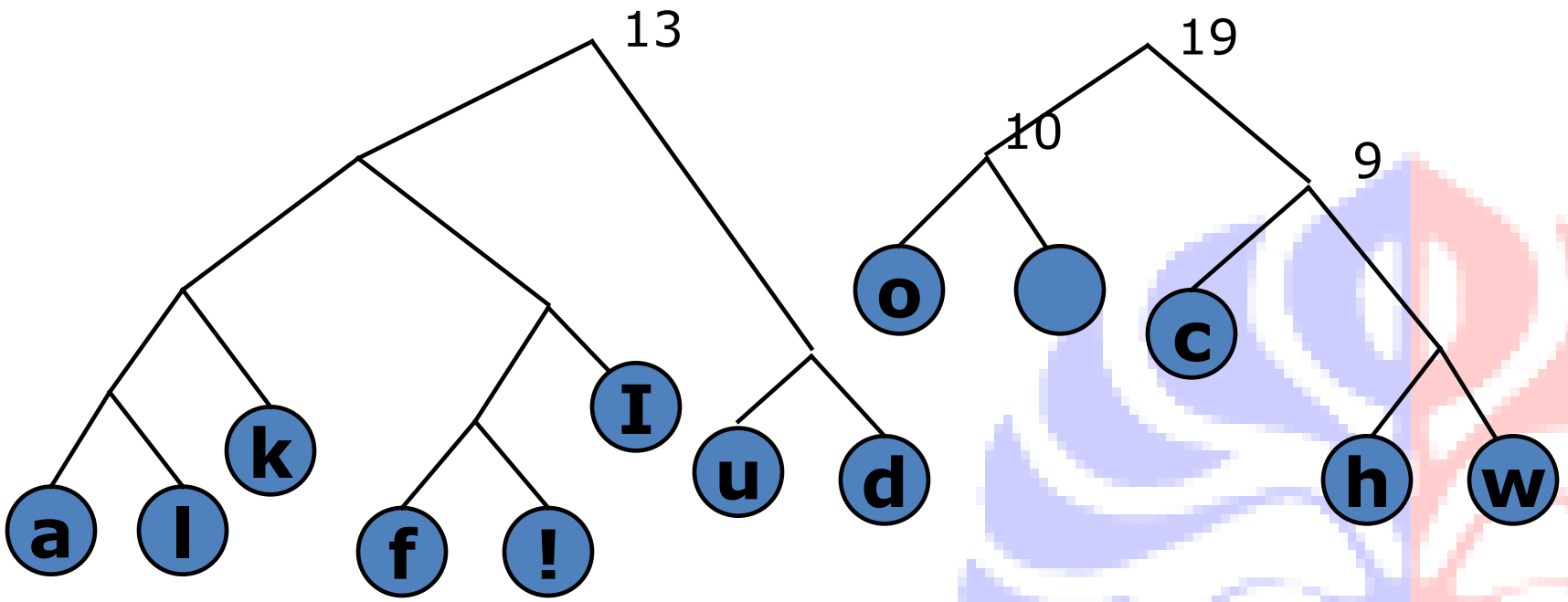
# Huffman Encoding: langkah-langkah



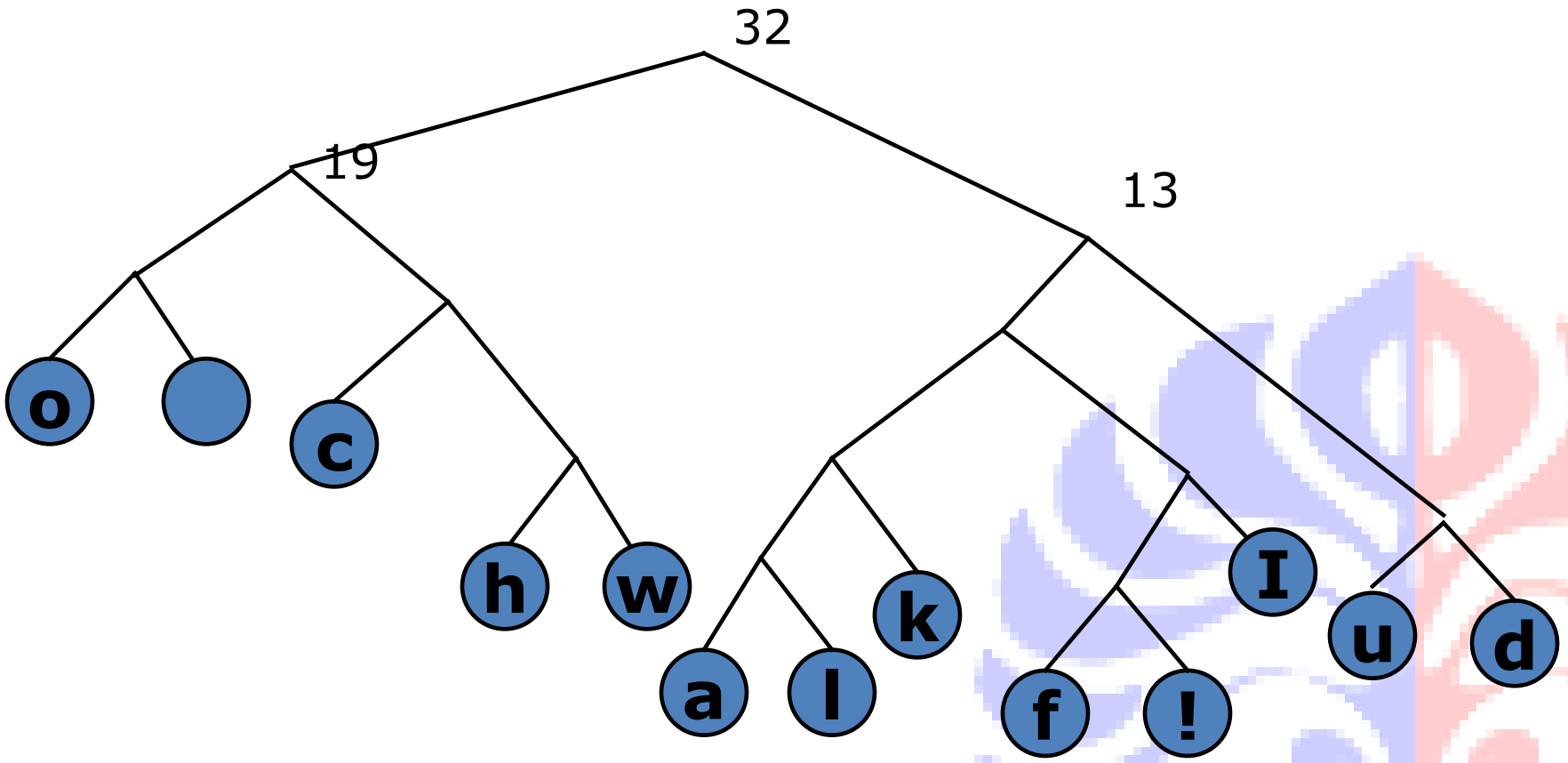
# Huffman Encoding: langkah-langkah



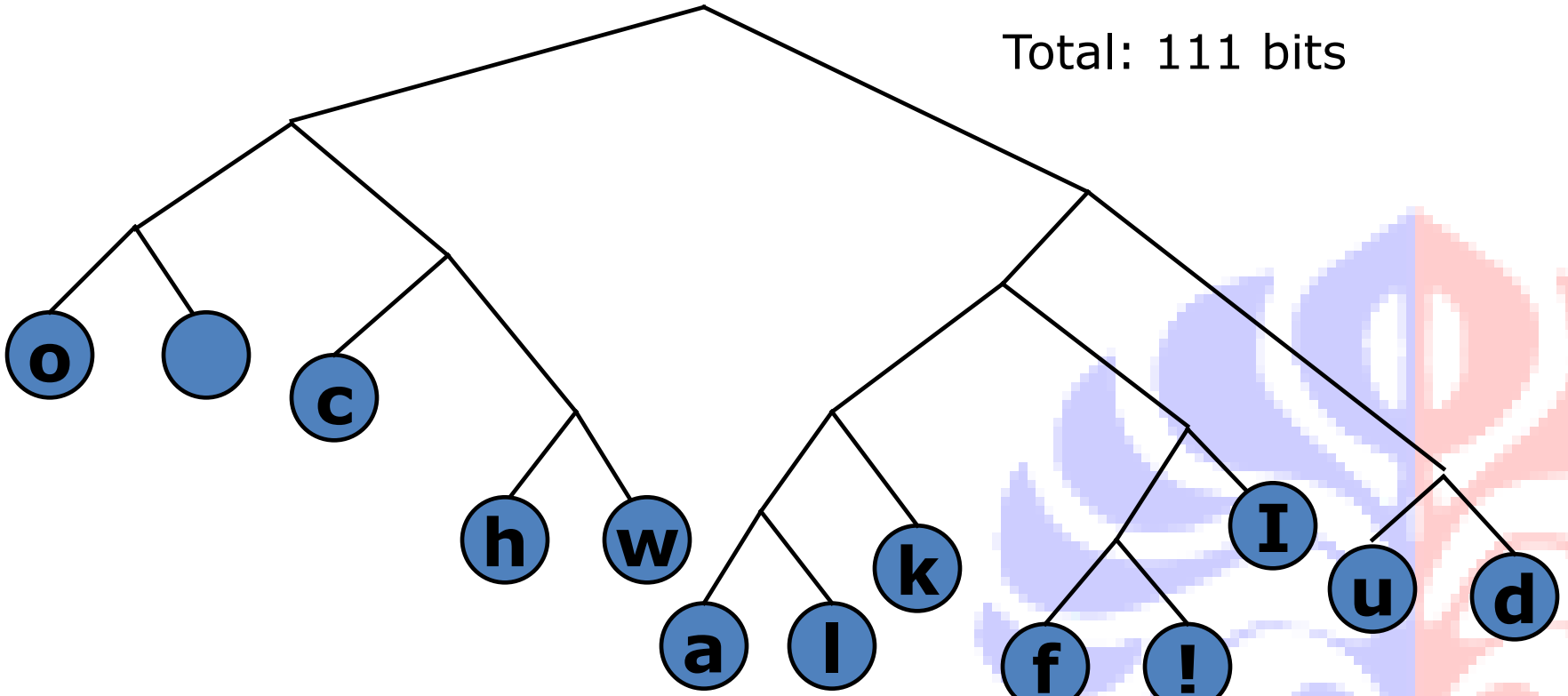
# Huffman Encoding: langkah-langkah



# Huffman Encoding: langkah-langkah



# Huffman Encoding: langkah-langkah



# Rangkuman

- Huffman encoding menggunakan informasi frekuensi kemunculan untuk meng-kompres file.
- Karakter dengan kemunculan paling tinggi di berikan encoding yang paling pendek dan sebaliknya.
- Penentuan code encoding menggunakan representasi binary tree.