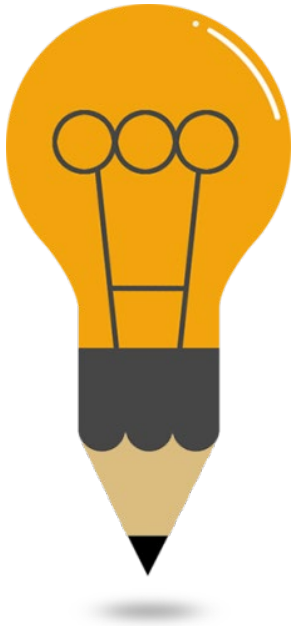


# STRUKTUR DATA

## Pertemuan 11



# Agenda Pertemuan



**1**

**Konsep Searching (Pencarian)**

**2**

**Sequential /Linear Search**

**3**

**Binary Search**

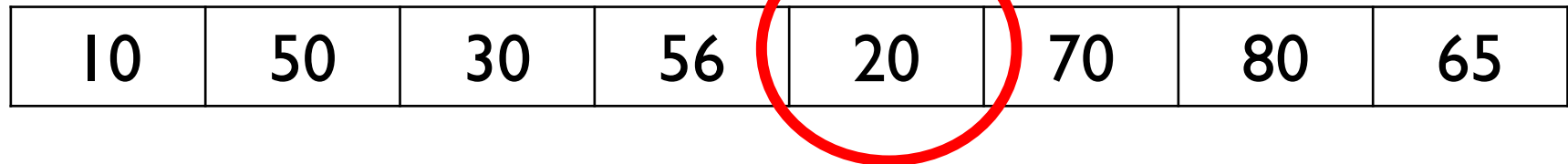
**4**

**Jump Search**

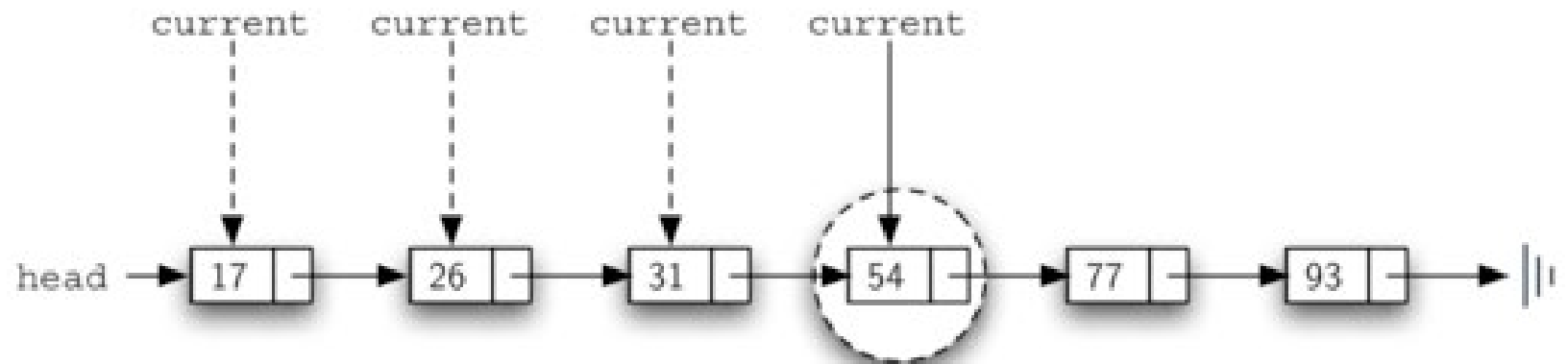
# Konsep *Searching*

- Proses pencarian adalah menemukan data tertentu di dalam sekumpulan data yang **bertipe sama**
- Pada materi ini akan diilustrasikan dengan data yang berstruktur array

**Cari '20'**



**Cari '54'**



## Contoh Kasus Pencarian

- Misal terdapat suatu **Array A** yang sudah terdefinisi elemen-elemennya. **X** adalah suatu elemen yang bertipe sama dengan elemen **Array A**. Tentukan apakah **X** terdapat di dalam **Array A**. Jika ditemukan, tulis pesan "X ditemukan" atau return index array atau return Boolean TRUE, sebaliknya jika tidak ditemukan, tulis pesan "X tidak ditemukan" atau return Boolean FALSE.
- Contoh:

21	36	8	7	10	36	68	32	12	10	36
0	1	2	3	4	5	6	7	8	9	10

- Misal  $X = 68$ , maka output yang dihasilkan adalah "68 ditemukan" atau " $Y = 6$ " atau TRUE
- Misal  $X = 100$ , maka output yang dihasilkan adalah "100 tidak ditemukan" atau " $Y = -1$ " atau FALSE

## Bagaimana jika ada duplikasi elemen?

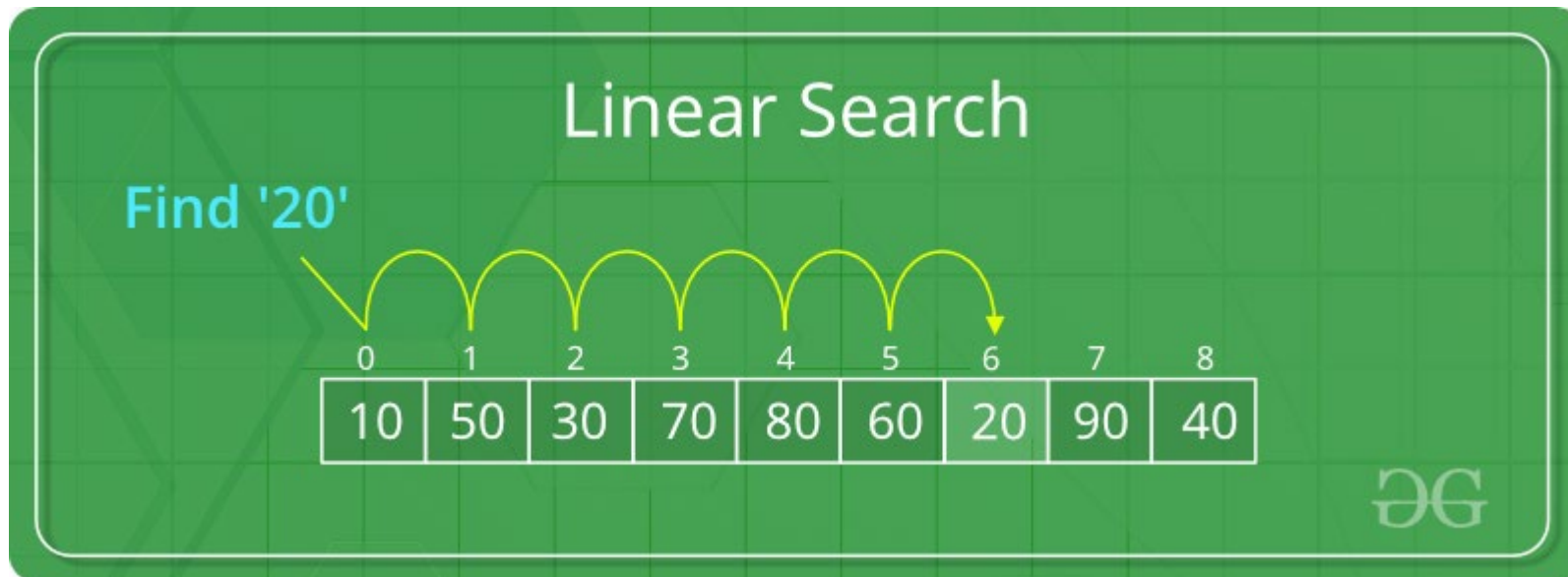
- Apabila **X** yang dicari jumlahnya lebih dari satu di dalam **Array A**, maka hanya **X** yang pertama kali ditemukan yang diambil. Proses pencarian dihentikan setelah X pertama ditemukan atau **X** yang dicari tidak ada.
- Contoh:

21	36	8	7	10	36	68	32	12	10	36
0	1	2	3	4	5	6	7	8	9	10

- Terdapat tiga buah nilai 36
- Bila  $X = 36$ , maka:
  - Hasilnya "36 ditemukan"
  - Hasilnya  $Y = 2$
  - Hasilnya `ketemu = true`

# Sequential/Linear Search

- Disebut juga Pencarian Beruntun
- *Sequential/Linear Search* membandingkan setiap elemen array satu per satu secara beruntun, mulai dari elemen pertama, sampai elemen yang dicari ditemukan atau sampai seluruh elemen sudah diperiksa.



# Ilustrasi Sequential Search

Mencari elemen “33”  
dengan memeriksa  
elemen satu per satu.

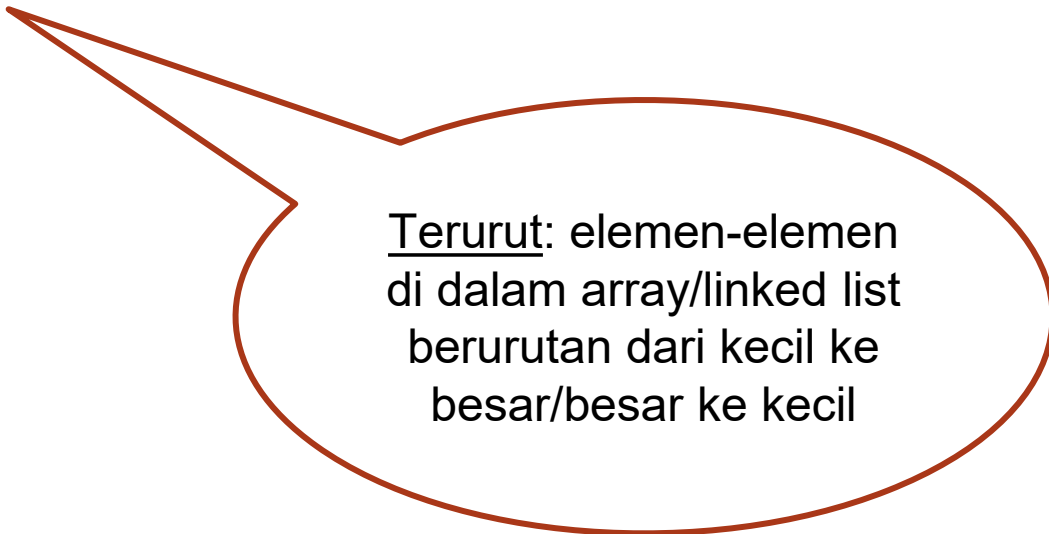
Sehingga ditemukan  
bahwa elemen “33”  
berada pada indeks  
ke-6.

Linear Search



# Jenis Sequential Search

- Sequential search pada array tidak terurut
- Sequential search pada array terurut



Terurut: elemen-elemen  
di dalam array/linked list  
berurutan dari kecil ke  
besar/besar ke kecil



# Sequential Search pada Array Tidak Terurut

- Pencarian dilakukan dengan memeriksa setiap elemen Array mulai dari elemen pertama sampai elemen yang dicari ketemu atau sampai seluruh elemen telah diperiksa.

13	16	14	21	76	21
0	1	2	3	4	5

- Misal nilai yang dicari adalah:  $X = 21$ . maka elemen yang diperiksa: 13, 16, 14, 21 (**ditemukan!**). Indeks Array yang dikembalikan:  $y = 3$
- Misal nilai yang dicari adalah:  $X = 13$ . maka elemen yang diperiksa: 13 (**ditemukan!**). Indeks Array yang dikembalikan:  $y = 0$
- Misal nilai yang dicari adalah:  $X = 15$ . maka elemen yang diperiksa: 13, 16, 14, 21, 76, 21 (**tidak ditemukan!**). Indeks Array yang dikembalikan:  $y = -1$

# Implementasi Program (Array)

```
int seq_search(int data[], int size, int x) {  
    for (int i=0; i<size; i++) {  
        if (data[i]==x) return i;  
    }  
    return -1;  
}
```

*Memeriksa elemen array satu per satu hingga data x ditemukan*

*Jika ditemukan maka return index array  
Jika tidak ditemukan return -1*

# Implementasi Program (Linked List)

```
int search(int x) { //x adalah nilai yang dicari
    int j = 1;
    ptrnode tmp = head;
    while(tmp != NULL) {
        if(x==tmp->value) {
            return j;
        }
        else {
            tmp = tmp->next; j++;
        }
    }
    return -1; //jika tidak ada yang dicari return -1
}
```

*Memeriksa node mulai dari head hingga data x ditemukan.*

*Jika ditemukan maka return di node berapa (j)  
Jika tidak ditemukan return -1*

# Sequential Search pada Array Terurut

- Apabila array sudah terurut, maka proses pencarian dapat dibuat **lebih efisien**.
  - Terurut dari nilai terkecil ke nilai terbesar, yaitu untuk setiap
$$i = 0..N, \text{ Nilai } [i-1] < \text{ Nilai } [i]$$
  - Terurut dari nilai terbesar ke nilai terkecil, yaitu untuk setiap
$$i = 0..N, \text{ Nilai } [i-1] > \text{ Nilai}[i]$$
- Caranya yaitu dengan menghilangkan langkah pencarian yang tidak perlu yaitu **jika nilai elemen array yang diperiksa sudah melewati nilai X yang dicari** maka bisa di-stop

# Implementasi Program (Array)

```
int seq_search_sorted(int data[], int size, int x) {  
    for (int i = 0; i < size; i++) {  
        if (data[i] > x) {  
            return -1;  
        }  
  
        if (data[i] == x) {  
            return i;  
        }  
    }  
}
```

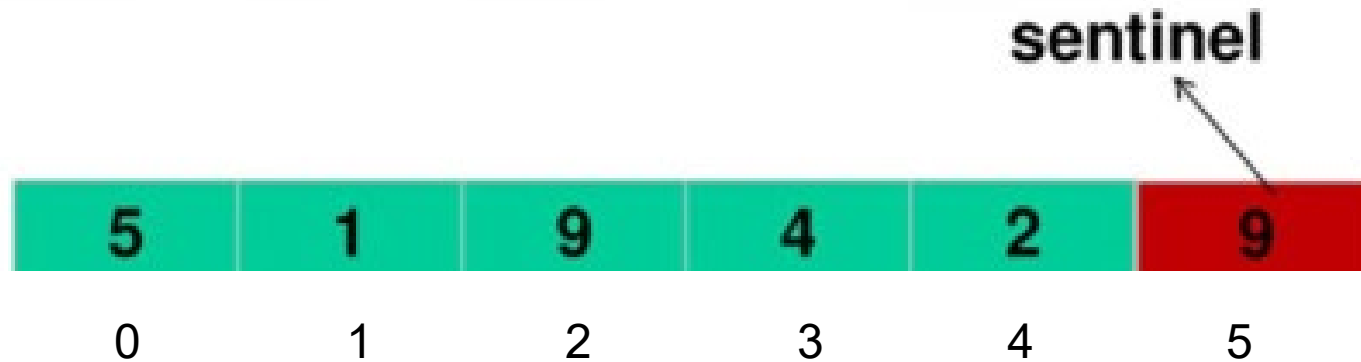
*STOP ketika sudah melewati data yang dicari.*

***Bagaimana untuk Linked List?***

# Sequential Search Menggunakan Sentinel

- Merupakan pengembangan dari sequential search
- Sentinel adalah elemen fiktif yang ditambahkan sesudah elemen terakhir dari array
  - Jika elemen terakhir array adalah  $A[N]$ , maka sentinel dipasang pada elemen  $A[N+1]$
- Sentinel **nilainya sama dengan nilai data yang dicari**
  - Sehingga proses pencarian selalu menemukan data yang dicari
  - Periksa kembali letak data tersebut ditemukan, apakah:
    - Di antara elemen array sesungguhnya ( dari  $A[0]$  sampai  $A[N]$  )
    - Pada elemen fiktif  $A[N+1]$  yang berarti sesungguhnya **X tidak ada di dalam array A**

# Sequential Search Menggunakan Sentinel



Programmer harus hati-hati dengan pendefinisian batas indeks array, tidak boleh menambahkan data melebihi rentang indeks.

Misal untuk array ini kita harus mendeklarasikan panjangnya lebih dari 5 karena kita harus punya tempat untuk sentinel.

- Data yang dicari = 9
- Tempatkan data yang dicari pada sentinel
- Telusuri array seperti sequential search tanpa sentinel, jika data ditemukan pada **sentinel** maka data yang dicari **tidak ada/tidak ditemukan**. Tetapi, jika data yang dicari ditemukan **bukan pada sentinel** maka **data ditemukan**.

# Ide Penggunaan Sentinel

```
int LinearSearch(int arr[], int len, int target) {  
    for (int i = 0; i < len; i++) {  
        if (arr[i] == target) {  
            return i;  
        }  
    }  
    return -1;  
}
```

$N+1$

$N$

Total Comparisons  
 $= 2N+1$

```
int LinearSearch(int arr[], int len, int target) {  
    arr[len] = target;  
    int i = 0;  
    while (arr[i] != target) {  
        i++;  
    }  
    if (i != len) return i;  
    return -1;  
}
```

Sentinel value

Signals the end of the search

And yes, assignment to  
arr[len] is a side-effect that can have bad  
consequences

$N+1$

1

Total Comparisons  
 $= N+2$



# Sequential Search Secara Umum

- Secara umum, Sequential search lambat. Waktu pencarian sebanding dengan jumlah elemen array. Pada kasus X tidak terdapat dalam Array, kita harus memeriksa seluruh elemen array.
  - Bayangkan bila array berukuran 100.000 elemen
  - Bila satu pemeriksaan elemen array membutuhkan waktu 0,01 detik, maka untuk 100.000 kali pemeriksaan membutuhkan waktu 1000 detik atau 16,7 menit!
- Algoritma sequential search tidak praktis untuk data berukuran besar
- Algoritma yang lebih cepat dari sequential search adalah **algoritma binary search**

# Binary Search

- Binary Search (pencarian biner) hanya bisa diterapkan pada sekumpulan **data yang sudah terurut** (terurut menaik atau menurun)
- Contoh data yang sudah terurut banyak ditemukan pada kehidupan sehari-hari:
  - Data kontak telepon di HP terurut dari nama A sampai Z
  - Data pegawai diurut berdasarkan nomor induk pegawai dari kecil ke besar
  - Data mahasiswa diurutkan berdasarkan NIM
  - Kata-kata di dalam kamus diurut dari A sampai Z
- Keuntungan data yang terurut adalah **memudahkan pencarian**.

# Contoh Binary Search

- Misal untuk mencari arti kata tertentu dalam kamus Bahasa Inggris, kita tidak perlu membuka kamus itu dari halaman awal sampai akhir satu per satu:
  - Pertama Kamus tersebut kita bagi dua di tengah-tengah.
  - Jika yang dicari tidak ada di pertengahan, kita cari lagi di belahan kiri atau kanan dengan membagi dua lagi belahan tersebut.
  - Begitu seterusnya sampai kata yang dicari ketemu.

# Ilustrasi Binary Search

## Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2 <sup>nd</sup> half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 > 56 take 1 <sup>st</sup> half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91



# Algoritma Binary Search

- Kita memerlukan dua buah indeks pada **Array A** yaitu indeks terkecil dan indeks terbesar (Misal **L** dan **H**). Umumnya  $L=0$  dan  $H=N$
- Langkah 1: bagi dua elemen array pada elemen tengah. Elemen tengah adalah elemen dengan indeks  **$M = (L+H) / 2$** 
  - Elemen tengah array  $A[M]$  membagi array menjadi dua bagian, yaitu bagian kiri Array  $A[L..M-1]$  dan bagian kanan Array  $A[M+1..H]$
- Langkah 2: periksa apakah  $A[M] = X$ . Jika Ya, maka pencarian dihentikan (data ditemukan).
  - Jika tidak, jika  $A[M] < X$  maka pencarian dilakukan pada array bagian kiri.
  - Jika  $A[M] > X$  maka pencarian dilakukan pada array bagian kanan.
- Langkah 3: Ulangi langkah 1-2 sampai  $X$  ditemukan atau  $L > H$  (ukuran array sudah nol!)

# Ilustrasi Algoritma Binary Search 1

- Misal ada array A dengan 8 elemen yang berurutan menurun. Data yang dicari adalah  $X=18$

81	76	21	18	16	13	10	7
L=0	1	2	3	4	5	6	H=7

- Langkah 1:**  $L=0$  dan  $H=7$ , maka indeks elemen tengah  $M = (0+7) / 2 = 3$

kiri	81	76	21	18	16	13	10	7	kanan
	L=0	1	2	3	4	5	6	H=7	

- Langkah 2:**  $A[3] = 18$ ? Ya! ( $X$  ditemukan, pencarian dihentikan)

# Ilustrasi Algoritma Binary Search 2

- Data yang dicari adalah  $X=16$

81	76	21	18	16	13	10	7
L=0	1	2	3	4	5	6	H=7

- **Langkah 1:**  $L=0$  dan  $H=7$ , maka indeks elemen tengah  $M = (0+7) / 2 = 3$

kiri	81	76	21	18	16	13	10	7	kanan
	L=0	1	2	3	4	5	6	H=7	

- **Langkah 2:**  $A[3] = 16$ ? Tidak! Tentukan pencarian akan dilakukan di bagian kiri atau kanan dengan pemeriksaan:
  - $A[3] > 16$  ? Ya! Lakukan pencarian di array bagian kanan

## Ilustrasi Algoritma Binary Search 2 (cont.)

*Ulangi langkah 1 dan 2*

- $L = M+1 = 4$  dan  $H = 7$

16	13	10	7
L=4	5	6	H=7

- **Langkah 1:** Indeks elemen tengah  $M = (4+7) / 2 = 5$

kiri	16	13	10	7	kanan
	L=4	5	6	H=7	

- **Langkah 2:**  $A[5] = 16$ ? Tidak! Tentukan pencarian akan dilakukan di bagian kiri atau kanan dengan pemeriksaan:
  - $A[5] > 16$  ? Tidak! Lakukan pencarian di array bagian kiri



## Ilustrasi Algoritma Binary Search 2 (cont.)

*Ulangi langkah 1 dan 2*

- $L = 4$  (tetap) dan  $H = 4$

16
$L = H = 4$

- **Langkah 1:** Indeks elemen tengah  $M = (4+4) / 2 = 4$

16
$M = L = H = 4$

- **Langkah 2:**  $A[4] = 16$ ? Ya! (X ditemukan, pencarian dihentikan)

# Implementasi Program (Array)

```
int binary_search(int data[], int size, int x){  
    int L = 0;  
    int H = size-1;  
    int M = -1;  
    int index = -1;  
  
    while(L <= H){  
        M = (L+H)/2;  
        if(data[M] == x)  
            return M;  
        else{  
            if(data[M] < x) L = M + 1;  
            else H = M - 1;  
        }  
    }  
    return -1;  
}
```

*while: iterasi langkah 1 dan 2 sampai habis atau ketemu*

*Langkah 1*

*Langkah 2*

*Cari di bagian kanan*

*Cari di bagian kiri*

# Implementasi Program (Linked List)

```
ptrnode binarySearch(int x){
    ptrnode start = head;
    ptrnode last = NULL;
    do{
        // temukan node tengah
        ptrnode mid = middle(start, last); Langkah 1
        // jika node tengah NULL
        if (mid == NULL) return NULL;
        // Jika x ditemukan di node tengah
        if (mid -> value == x) return mid; Langkah 2
        // Jika nilai x lebih dari node tengah
        else if (mid -> value < x) start = mid -> next; Cari di bagian kanan
        // Jika nilai x kurang dari node tengah
        else last = mid; Cari di bagian kiri
    } while (last == NULL || last != start);

    // jika tidak ditemukan
    return NULL;
}
```

# Binary Search Secara Umum

- Pada setiap kali pencarian, array dibagi dua menjadi dua bagian yang berukuran sama (atau beda selisih 1 elemen)
- Pada setiap pembagian, elemen tengah  $M$  diperiksa apakah sama dengan  $X$ .
  - Pada *worst case scenario*, yaitu pada kasus  $X$  tidak terdapat di dalam array, array dibagi sejumlah  $2^{\log(N)}$  kali, sehingga jumlah pemeriksaan yang dilakukan juga sebanyak  $2^{\log(N)}$  kali.
  - Jika jumlah elemen 1000, pembagian array yang dilakukan adalah  $2^{\log(1000)} \approx 3$  kali. Jumlah pemeriksaan elemen array juga 3 kali.
- Untuk data terurut, algoritma *binary search* lebih cepat dibandingkan *sequential search*

# Sequential Search VS Binary Search

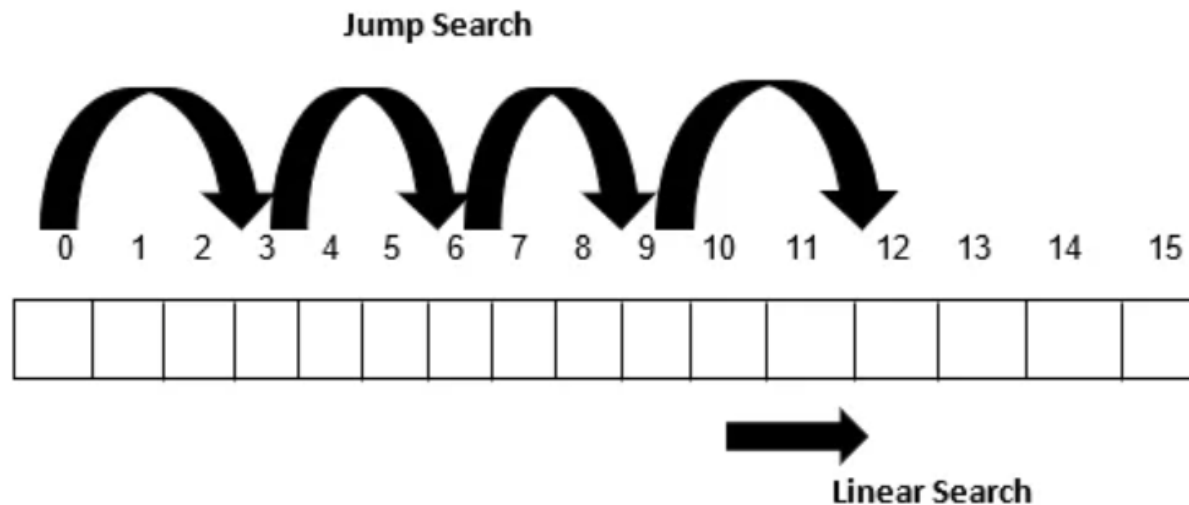
- Misal untuk kasus nilai X tidak ditemukan dalam array:
  - Untuk array berukuran 256 elemen:
    - *sequential search* melakukan perbandingan elemen array sebanyak 256 kali,
    - *binary search* melakukan perbandingan elemen array sebanyak  $2^{\log(256)} = 8$  kali.
  - Untuk array berukuran 1024 elemen:
    - *sequential search* melakukan perbandingan elemen array sebanyak 1024 kali,
    - *binary search* melakukan perbandingan elemen array sebanyak  $2^{\log(1024)} = 10$  kali.
  - Untuk array berukuran N elemen:
    - *sequential search* melakukan perbandingan elemen array sebanyak N kali,
    - *binary search* melakukan perbandingan elemen array sebanyak  $2^{\log(N)}$  kali.
- Karena  $2^{\log(N)} < N$  untuk N yang besar, maka algoritma *binary search* lebih cepat daripada algoritma *sequential search*
  - Sehingga Algoritma *binary search* lebih disukai untuk mencari data pada array terurut
- Namun untuk data yang tidak terurut, hanya dapat menggunakan algoritma *sequential search*

# Jump Search

- Jump Search merupakan algoritma yang relatif baru untuk mencari elemen dalam **data yang terurut** (seperti Binary Search)
- Ide pencarian ini adalah untuk **mencari jumlah elemen yang lebih sedikit** dibandingkan dengan Linear Search
- Dilakukan dengan melewati beberapa elemen array dalam jumlah tetap atau **melompat ke depan** dengan jumlah langkah yang tetap dalam setiap iterasi.

Misal pada array dengan panjang  $n$ , maka lakukan **jump** setiap blok dimana panjang bloknnya adalah  $\sqrt{n}$ .

Setelah menemukan blok yang benar, elemen akan dicari menggunakan Linear Search.



# Ilustrasi Algoritma Jump Search

- Misal dilakukan pencarian elemen **55** pada array **A** = (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610) dengan ukuran 16
- Langkah 1: Hitung ukuran block  $m = \sqrt{n} = 4$
- Langkah 2: Bandingkan nilai ke-m yaitu pada **A[m-1]** dengan elemen yang dicari 55. Karena **A[3] ≤ 55**, **jump** ke blok berikutnya
- Langkah 3: Hitung  $m = m + \sqrt{n} = 8$
- Langkah 4: Bandingkan nilai A[m-1] dengan elemen yang dicari 55. Karena **A[7] ≤ 55**, **jump** ke blok berikutnya
- Langkah 5: Hitung  $m = m + \sqrt{n} = 12$
- Langkah 6: Bandingkan nilai A[m-1] dengan elemen yang dicari 55. Karena **A[11] = 89**, jadi **A[11] != 55** dan **A[11] > 55**, maka lakukan **Linear Search** dari m sebelumnya (7) sampe m sekarang (11)

0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610
---	---	---	---	---	---	---	----	----	----	----	----	-----	-----	-----	-----

**Linear Search dari index 7**

# Implementasi Program (Array)

```
int jump_search(int data[], int n, int x) {
    int prev = 0;
    int m = sqrt(n); //hitung block size= akar(n)

    while(data[m-1] <= x) {
        if(data[m-1] == x)
            return m-1; //return posisi index elemen yang dicari

        prev = m; //prev untuk menyimpan index m sebelumnya
        m += sqrt(n);
        if(m > n) // if m melebihi ukuran array, maka tidak ditemukan
            return -1;
    }

    for(int j = prev; j<m; j++) { // lakukan linear search pada block
        if(data[j] == x)
            return j; //return posisi index elemen yang dicari
    }
    return -1;
}
```



# Time Complexity

**linear search < jump search < binary search**

	Time Complexity
Linear Search	$O(n)$
Binary Search	$O(2\log(n))$ $O(\log(n))$
Jump Search	$O(\sqrt{n})$



TERIMA KASIH