Design and Analysis of Data Structures and Algorithms :: Searching

อ.ดร.วรินทร์ วัฒนพรพรหม

การค้นหาข้อมูล (Searching)

- **Searching** คือการค้นหาเพื่อดูว่าข้อมูลที่ต้องการมีอยู่หรือไม่ในโครงสร้างข้อมูล วิธีการค้นหามีหลายวิธี การเลือกใช้จะต้องสัมพันธ์กับรูปแบบของโครงสร้างข้อมูลและพยายามเลือกวิธีที่ทำให้การค้นหานั้นเป็นไปอย่างรวดเร็วที่สุด
- วัตถุประสงค์หลักของการค้นหาข้อมูล
  - ต้องการทราบรายละเอียดของข้อมูลที่ต้องการ **(Retrieve)**
  - ต้องการลบข้อมูลโดยจะต้องค้นหาข้อมูลที่ต้องการลบออกจากโครงสร้าง **(Delete)**
  - เปลี่ยนแปลงแก้ไขรายละเอียดบางอย่างของข้อมูลตัวที่ค้นพบ **(Update)**
  - การเพิ่มข้อมูลใหม่ในโครงสร้างที่มีการเรียงลำดับซึ่งต้องค้นหา2ตำแหน่งที่จะแทรกลงไป **(Insert)**

ประเภทของการค้นหาข้อมูล

- **การค้นหาข้อมูลแบบภายใน (internal searching)** คือการค้นหาข้อมูลในโครงสร้างข้อมูลที่จัดเก็บในหน่วยความจำหลัก เช่น การค้นหาข้อมูลในแถวลำดับ ลิงค์ลิสต์ และทรี เป็นต้น
- **การค้นหาข้อมูลแบบภายนอก (external searching)** คือการค้นหาข้อมูลที่จัดเก็บอยู่ในหน่วยความจำสำรอง เช่น การค้นหาข้อมูลในดิสก์ (secondary storage) ที่มีความจุใหญ่กว่าแต่ช้ากว่า

วิธีการค้นหาข้อมูล

- Sequential Search
- Binary Search
- Indexed Sequential Search
- Hashing Search

Sequential Search

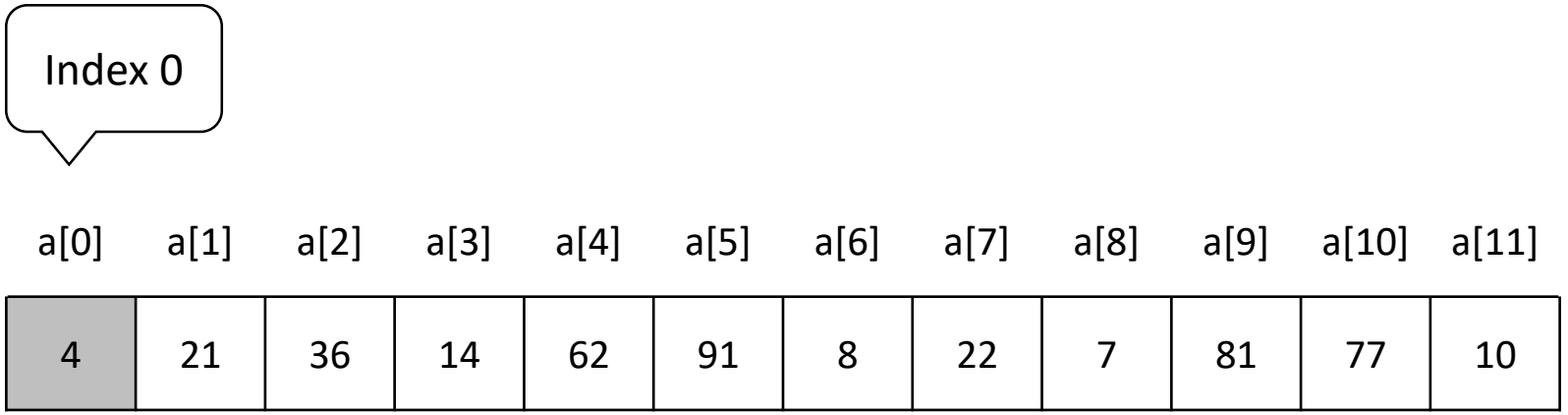
- **การค้นหาแบบลำดับอนุกรม (sequential search)** เป็นวิธีการค้นหาข้อมูลที่ง่ายและตรงไปตรงมาที่สุด การค้นหาทำได้โดยนำค่าหลักไปเปรียบเทียบกับข้อมูลทั้งหมดทีละตัวตั้งแต่ตัวแรกเรียงตามลำดับจนกว่าจะพบข้อมูลที่ต้องการ หรือเปรียบเทียบไปจนถึงตัวสุดท้ายและพบว่าไม่มีข้อมูลนี้อยู่

ขั้นตอนการค้นหาแบบ Sequential Search

1. กำหนดเขตข้อมูลให้เป็น คีย์หลัก หรือดัชนีในการค้นหาข้อมูล
2. อ่านข้อมูล ค่าแรกจากแถวลำดับหรือแฟ้มข้อมูล นำมาเปรียบเทียบกับข้อมูลดัชนี ถ้าตรงกัน แสดงว่าการค้นหาประสบความสำเร็จ แต่ถ้าไม่พบให้อ่านค่าถัดไป นำมาเปรียบเทียบใหม่ ทำเช่น นี้ไปเรื่อย ๆ จนกว่าจะพบ หรือ หมดข้อมูล

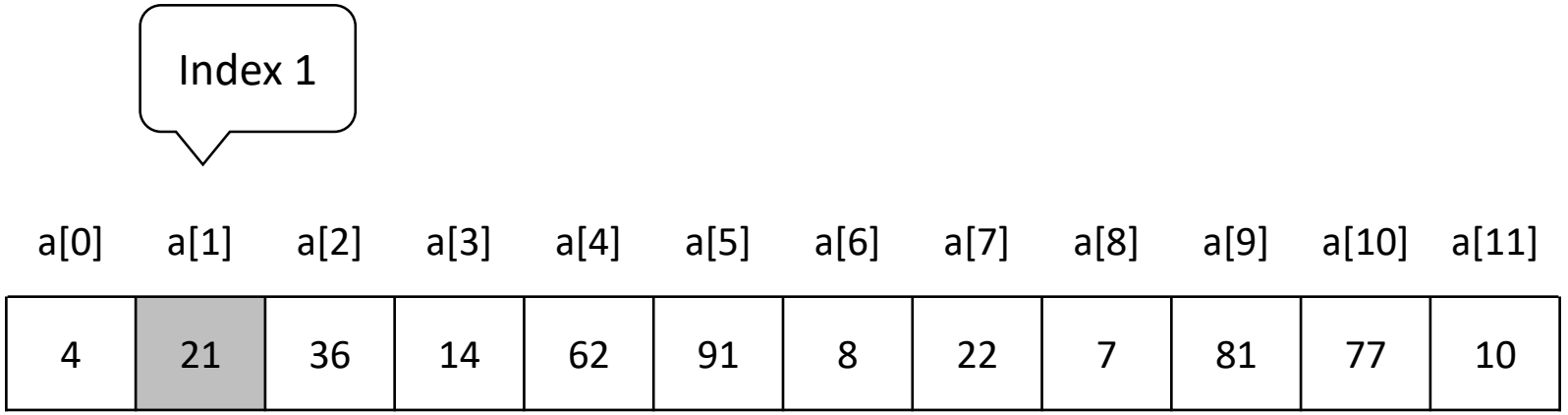
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 4
Location not found



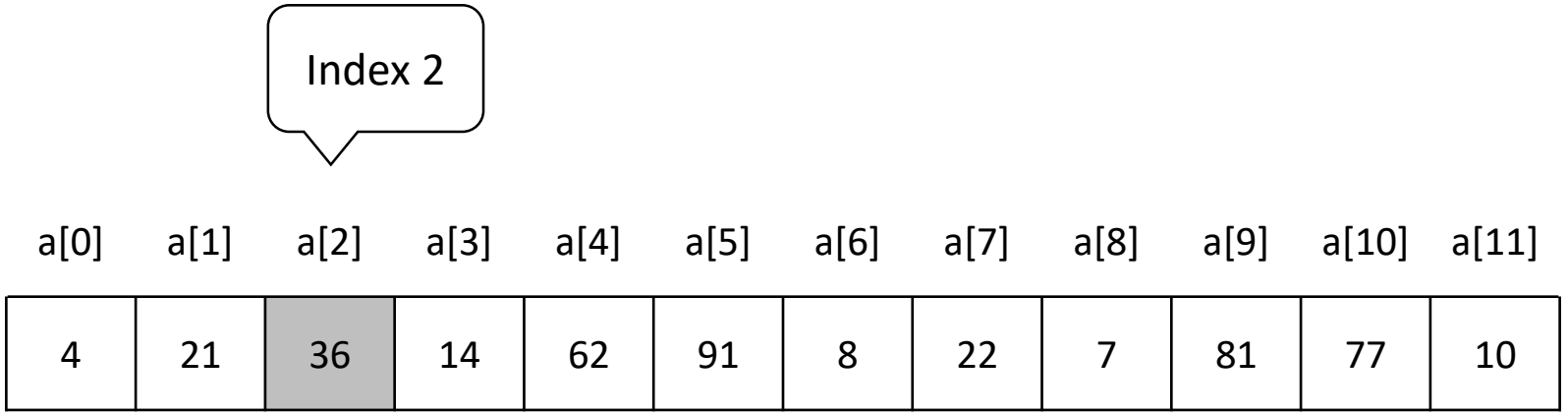
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 21
Location not found



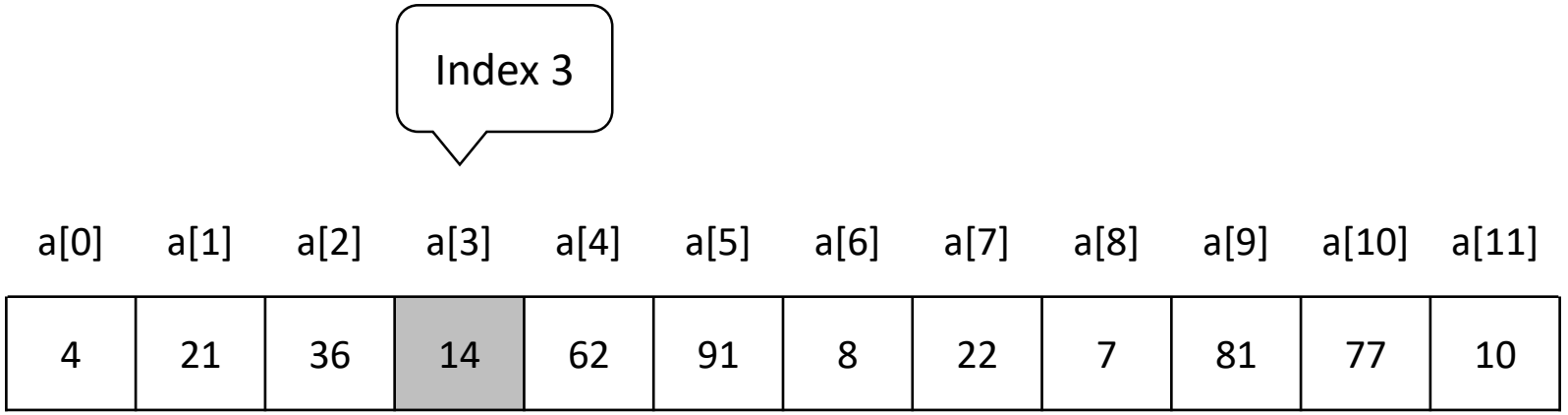
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 36
Location not found



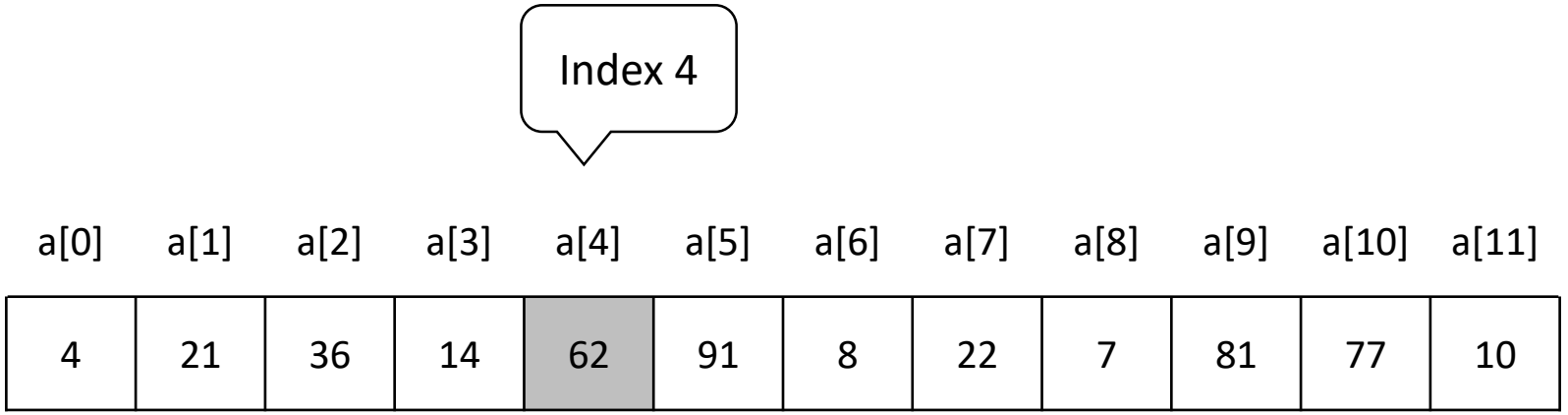
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 14
Location not found



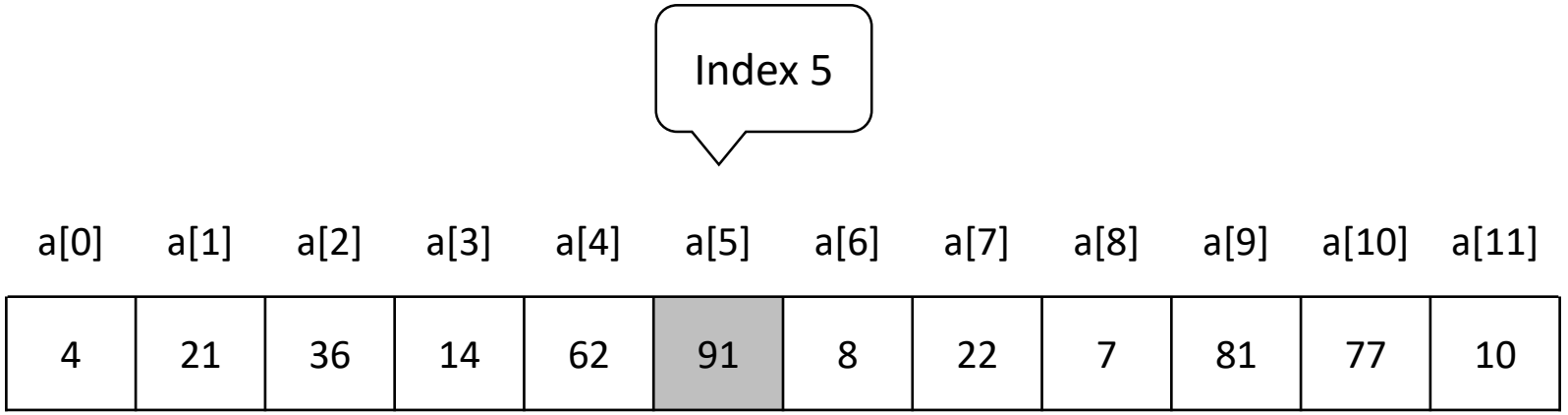
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 62
Location not found



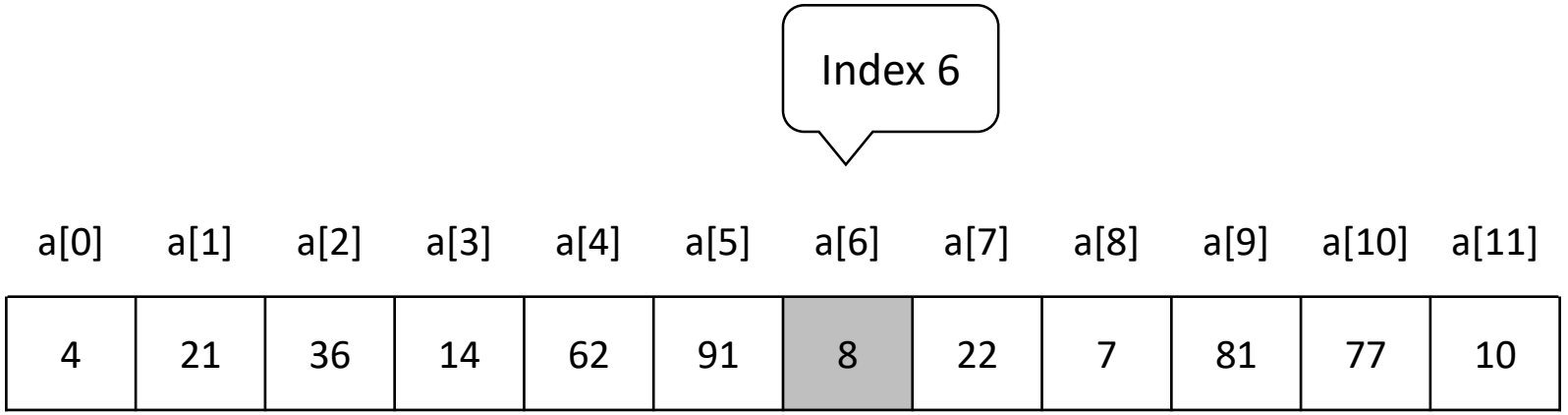
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 91
Location not found



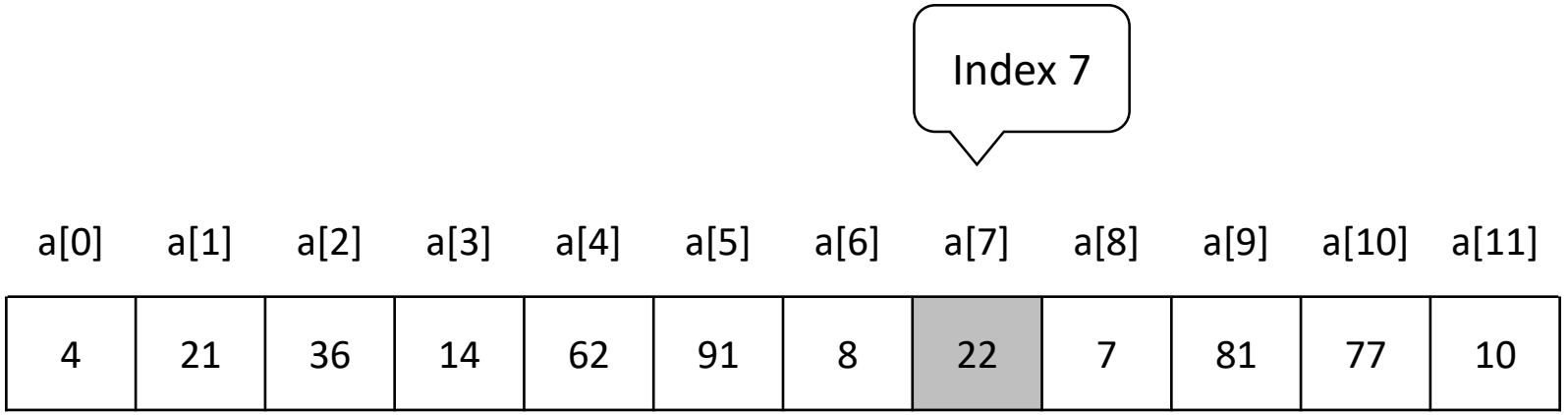
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 8
Location not found



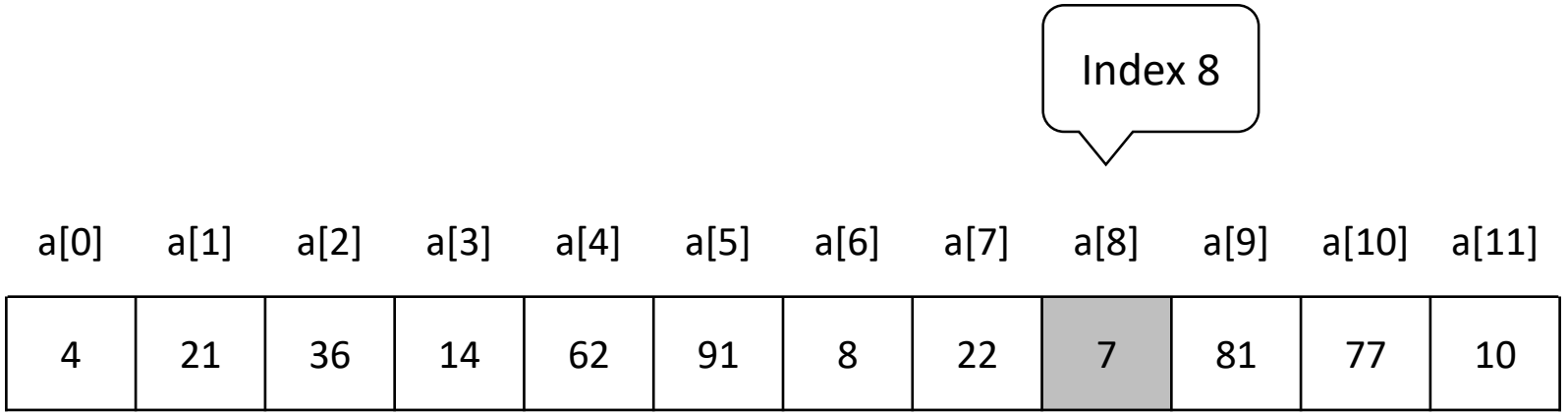
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 22
Location not found



Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

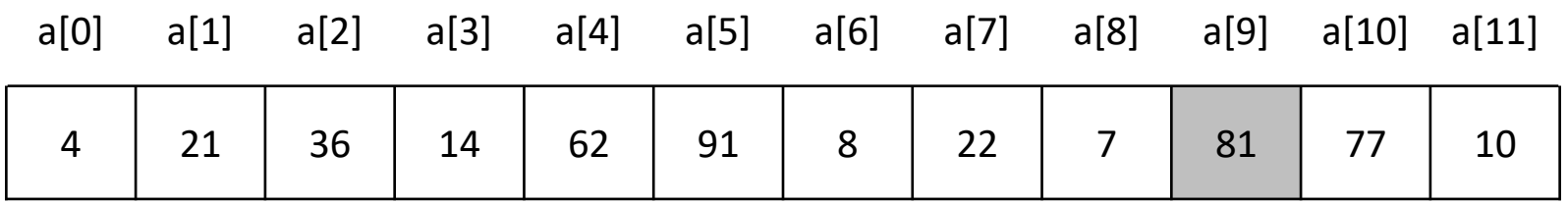
Target: 72
72 ≠ 7
Location not found



Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 81
Location not found

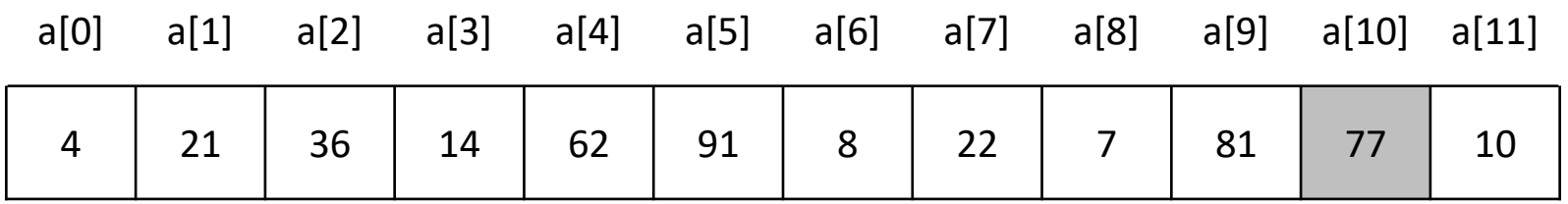
Index 9



Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 77
Location not found

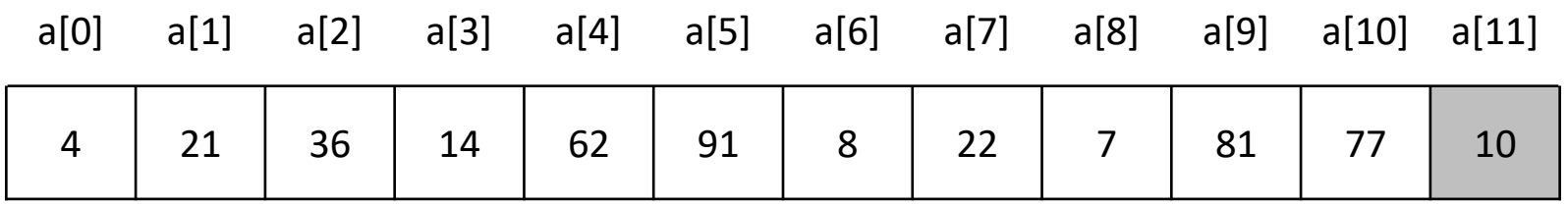
Index 10



Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าไม่เจอ

Target: 72
72 ≠ 10
Location not found

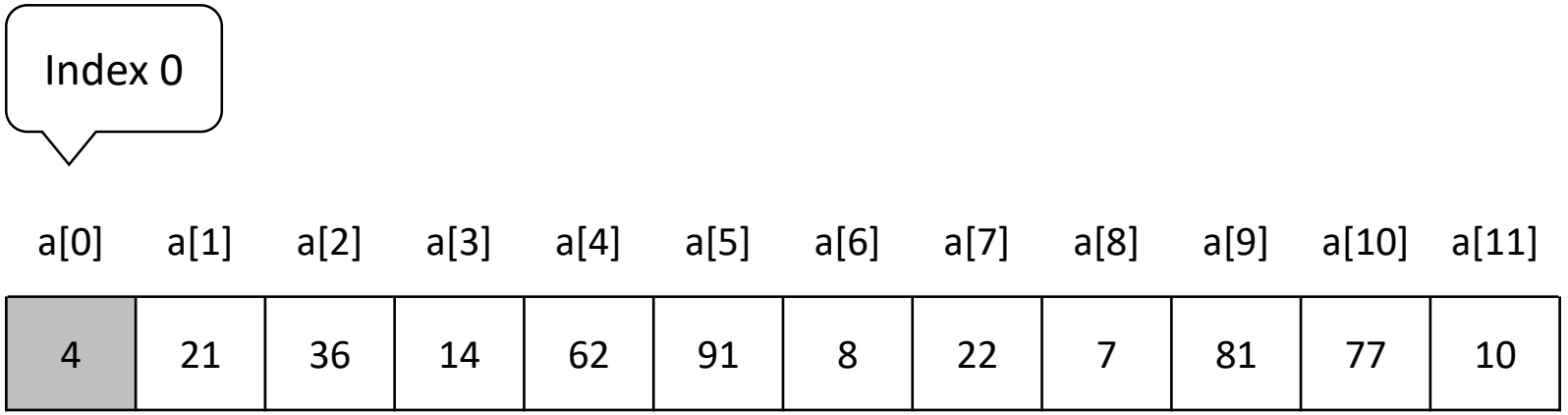
Index 11



$O(n)$

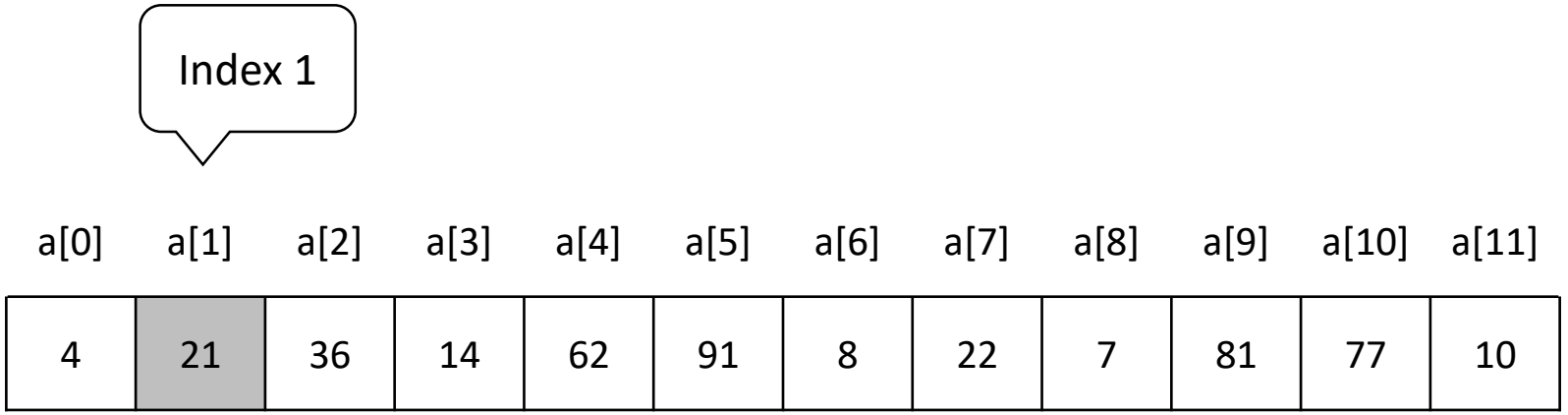
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าเจอ

Target: 62
62 ≠ 4
Location not found



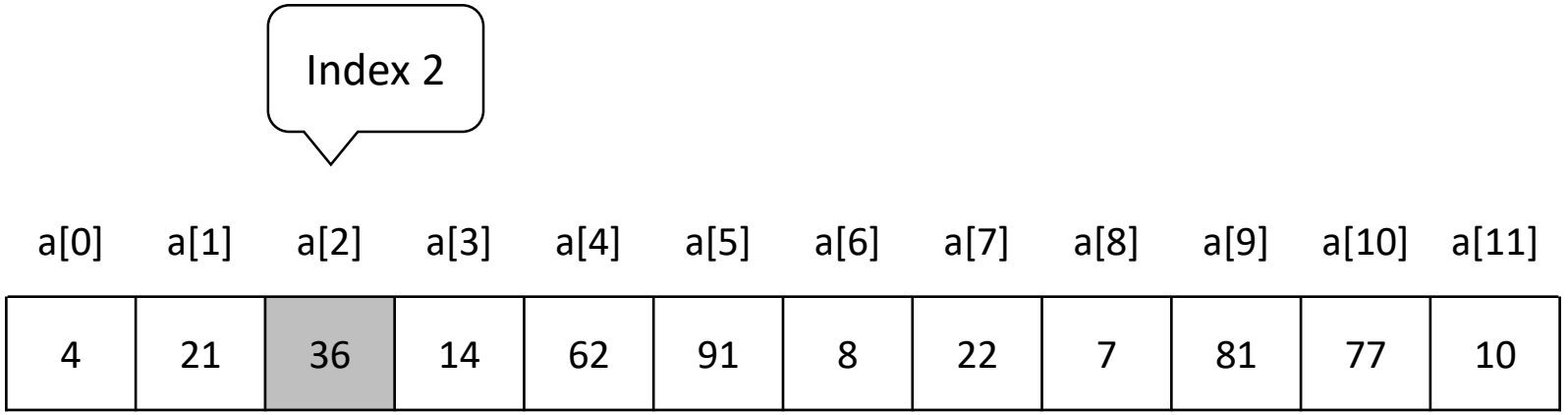
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าเจอ

Target: 76
62 ≠ 21
Location not found



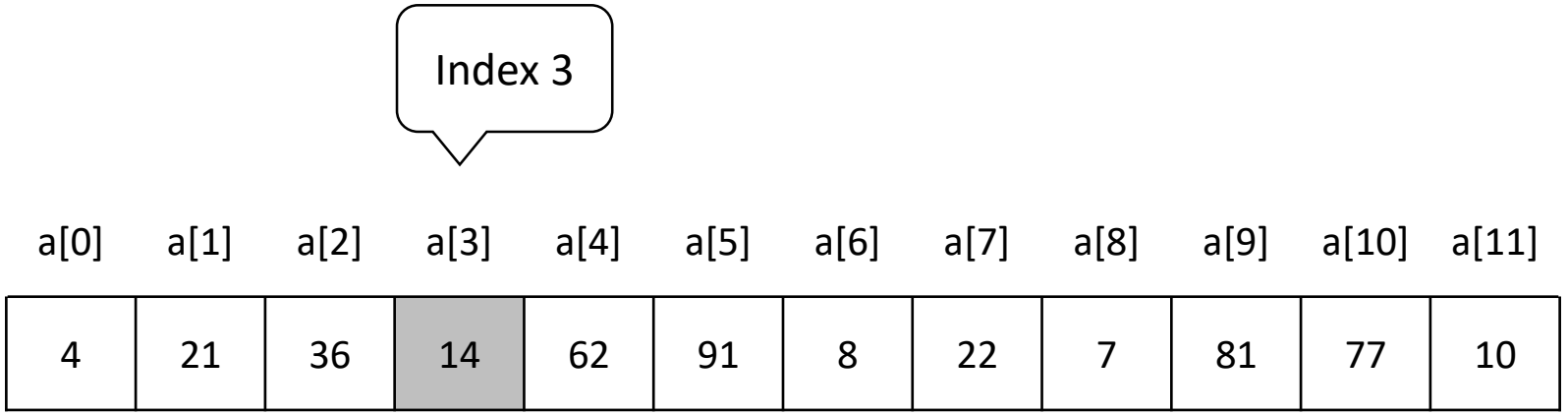
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าเจอ

Target: 62
62 ≠ 36
Location not found



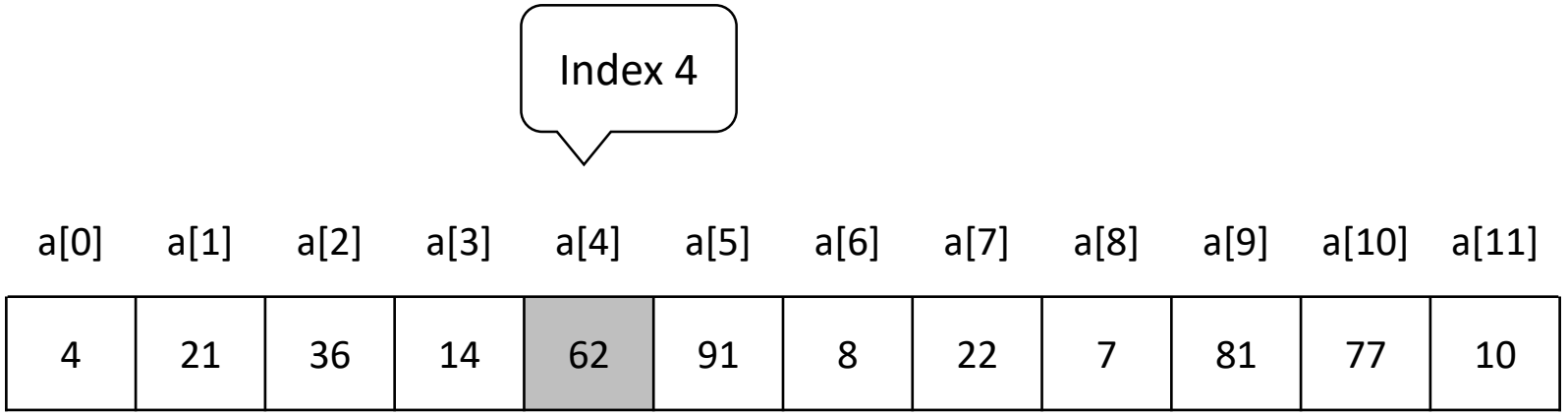
Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าเจอ

Target: 62
62 ≠ 14
Location not found



Sequential Search - กรณีข้อมูลไม่เรียงลำดับแล้วหาค่าเจอ

Target: 62
62 = 62
Location found!!



$\Omega(n)$

การค้นหาแบบ Sequential Search

- ถ้ามีข้อมูลทั้งหมด n จำนวน จำนวนครั้งของการเปรียบเทียบเป็นดังนี้
- **กรณีที่ดีที่สุด** คือกรณีที่ข้อมูลที่ต้องการค้นอยู่ที่ตำแหน่งแรก จำนวนครั้งของการเปรียบเทียบเท่ากับ 1 ครั้ง
- **กรณีที่แย่ที่สุด** คือกรณีที่ข้อมูลที่ต้องการค้นอยู่ที่ตำแหน่งสุดท้าย หรือไม่มีข้อมูลนั้นอยู่เลย จำนวนครั้งของการเปรียบเทียบเท่ากับ n ครั้ง
- ประสิทธิภาพของ Sequential Search คือ $\mathrm{O}(n)$

การค้นหาแบบ Sequential Search

- กรณีที่ข้อมูลที่มีการเรียงลำดับอยู่แล้ว การค้นหาข้อมูลจะรวดเร็วขึ้น
- เนื่องจากสามารถใช้ความสัมพันธ์ของการเรียงลำดับที่มีอยู่มาช่วยในการค้นหาได้
- โดยเฉพาะการค้นหาข้อมูลที่ไม่มีอยู่ในโครงสร้างการค้นหาไม่จำเป็นต้องค้นไปจนถึงข้อมูลตัวสุดท้าย
- เพราะเมื่อเจอข้อมูลที่มีค่ามากกว่า (ข้อมูลเรียงจากน้อยไปมาก) สามารถหยุดค้นได้ทันที

การค้นหาแบบทวิภาค Binary Search

- **การค้นหาแบบทวิภาค (binary search)** เป็นวิธีการค้นหาข้อมูลที่รวดเร็วกว่าการค้นหาแบบเรียงลำดับ
- แต่วิธีนี้ใช้ได้ในกรณีที่ข้อมูลถูกจัดเก็บแบบเรียงลำดับเรียบร้อยแล้ว
- โดยอาจจะเรียงลำดับจากน้อยไปมากหรือจากมากไปน้อย
- การค้นหาข้อมูลด้วยวิธีนี้ ในการเปรียบเทียบแต่ละครั้งสามารถลดจำนวนข้อมูลที่ต้องเปรียบเทียบได้คราวละประมาณครึ่งหนึ่งของที่เหลือ
- ประสิทธิภาพของ Binary Search คือ $O(\log n)$

การค้นหาแบบทวิภาค Binary Search

- การค้นหาข้อมูลแบบทวิภาคเริ่มต้นด้วยการหาว่าตำแหน่งกึ่งกลางของ ข้อมูลทั้งหมดอยู่ที่ตำแหน่งใด
- เนื่องจากข้อมูลทั้งหมดมีการเรียงลำดับค่า ถ้าเรียงจากน้อยไปมาก ข้อมูลที่ตำแหน่งกึ่งกลางจะแบ่งข้อมูลเป็น 2 ส่วน
- ส่วนแรกเป็นข้อมูลที่มีค่าน้อยกว่าค่าที่ตำแหน่งกึ่งกลาง และส่วนที่ 2 เป็นข้อมูลที่มีค่ามากกว่าค่าที่ตำแหน่งกึ่งกลาง

การค้นหาแบบทวิภาค Binary Search

- นำค่าหลักที่ต้องการค้นหาไปเปรียบเทียบกับข้อมูลที่ตำแหน่งกึ่งกลางนั้น
- ถ้าเท่ากันแสดงว่าพบค่าที่ต้องการค้นแล้ว
- แต่ถ้าไม่เท่ากันให้พิจารณาว่าน้อยกว่าหรือมากกว่าข้อมูลที่ตำแหน่งกึ่งกลาง
  - ถ้าค่าหลักมีค่าน้อยกว่าแสดงว่าค่าที่ต้องการค้นอาจจะอยู่ในส่วนที่ 1
  - และถ้ามากกว่าแสดงว่าค่าที่ต้องการค้นอาจจะอยู่ในข้อมูลส่วนที่ 2
  - ให้ดำเนินการเปรียบเทียบในทำนองเดียวกันกับข้อมูลในส่วนที่คาดว่าจะมีข้อมูลที่ต้องการค้นอยู่จนกระทั่งพบข้อมูลที่ต้องการ
  - หรือจนกระทั่งไม่สามารถแบ่งข้อมูลได้อีกแล้วแสดงว่าไม่มีข้อมูลที่ต้องการ

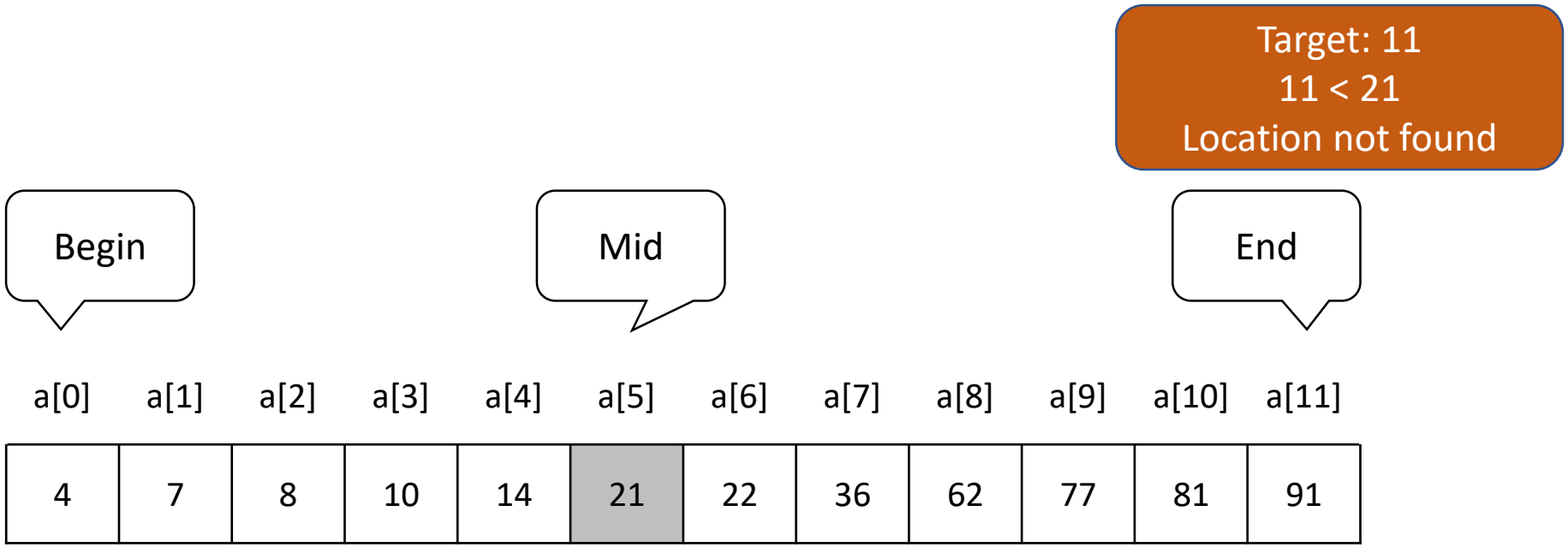
Binary Search (Recursive Style)

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;
        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;
        // If element is smaller than mid, then it can only be
present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}
```

Binary Search (Looping Style)

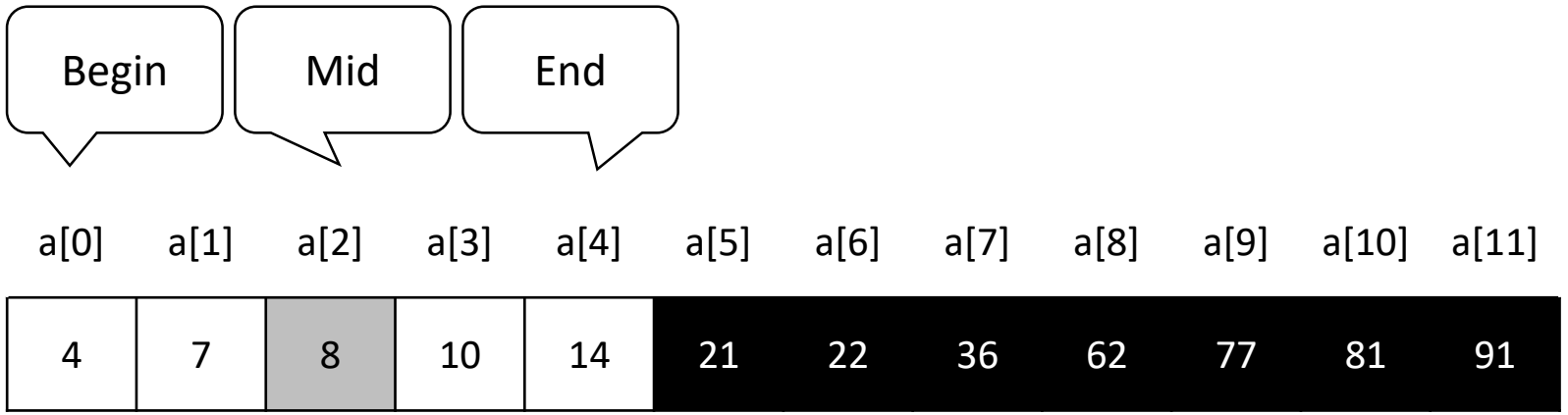
```
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;
        // Check if x is present at mid
        if (arr[m] == x)
            return m;
        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;
        // If x is smaller, ignore right half
        else
            r = m - 1;
    }
    // if we reach here, then element was not present
    return -1;
}
```

Binary Search - กรณีหาค่าไม่เจอ



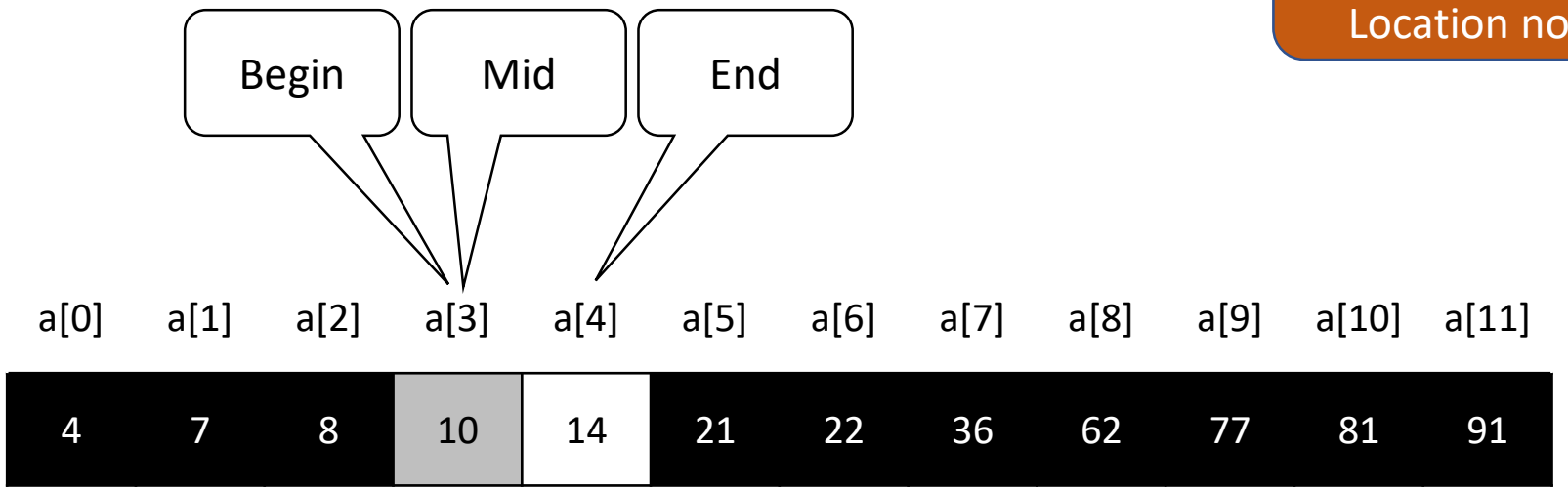
Binary Search - กรณีหาค่าไม่เจอ

Target: 11
11 > 8
Location not found



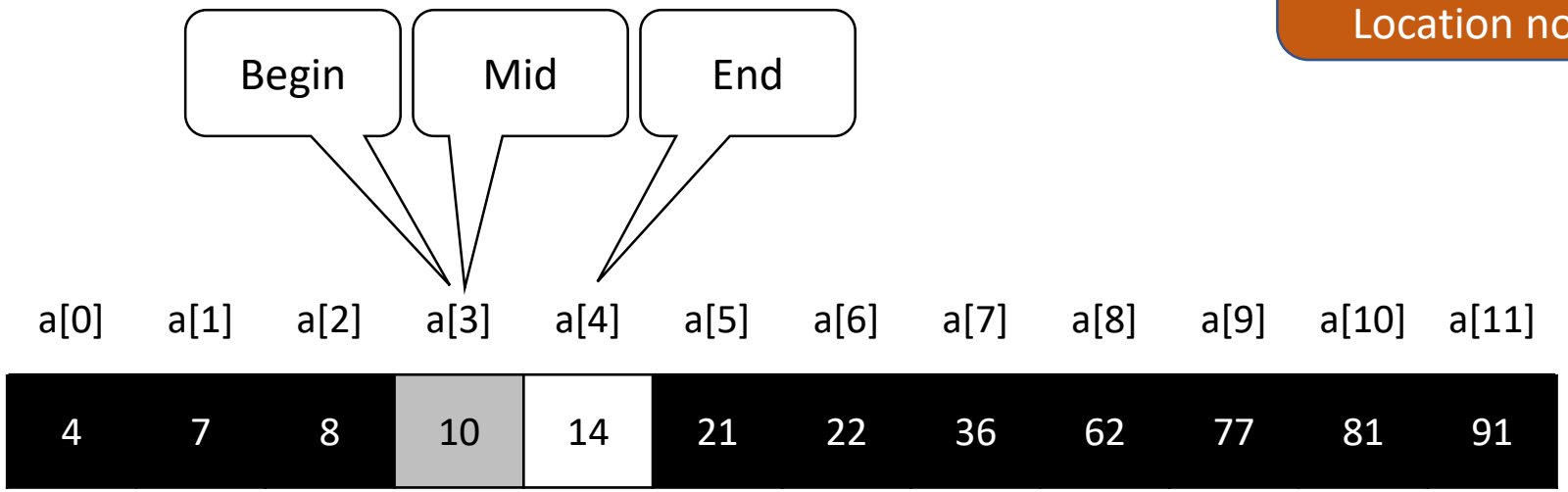
Binary Search - กรณีหาค่าไม่เจอ

Target: 11
11 > 8
Location not found



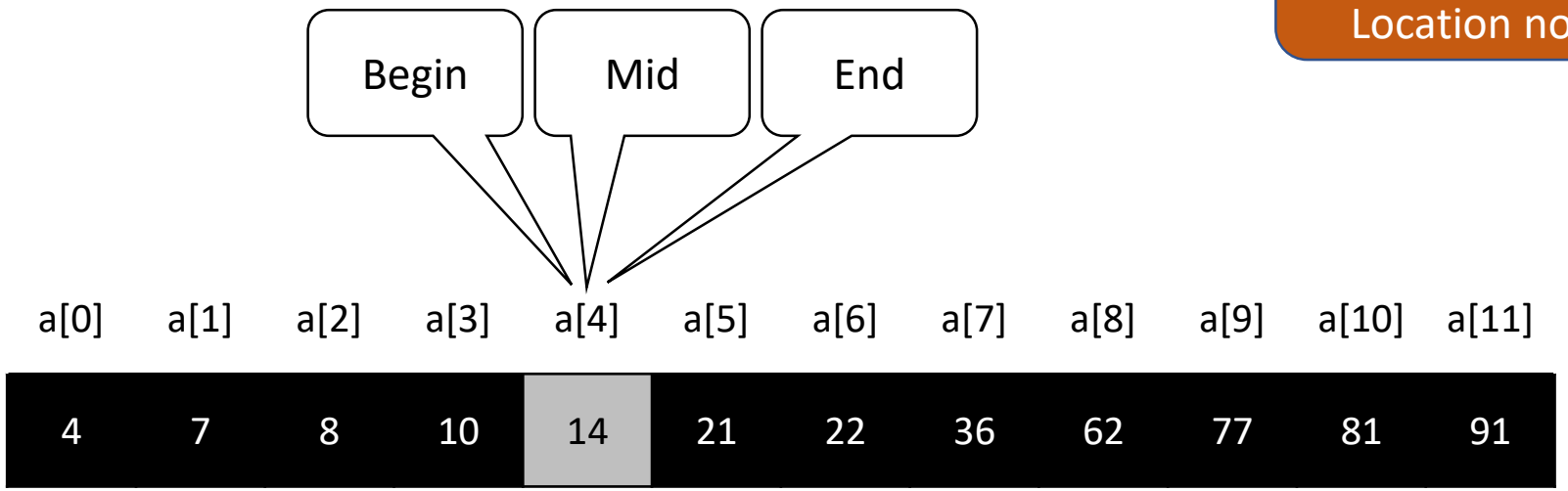
Binary Search - กรณีหาค่าไม่เจอ

Target: 11
11 > 10
Location not found

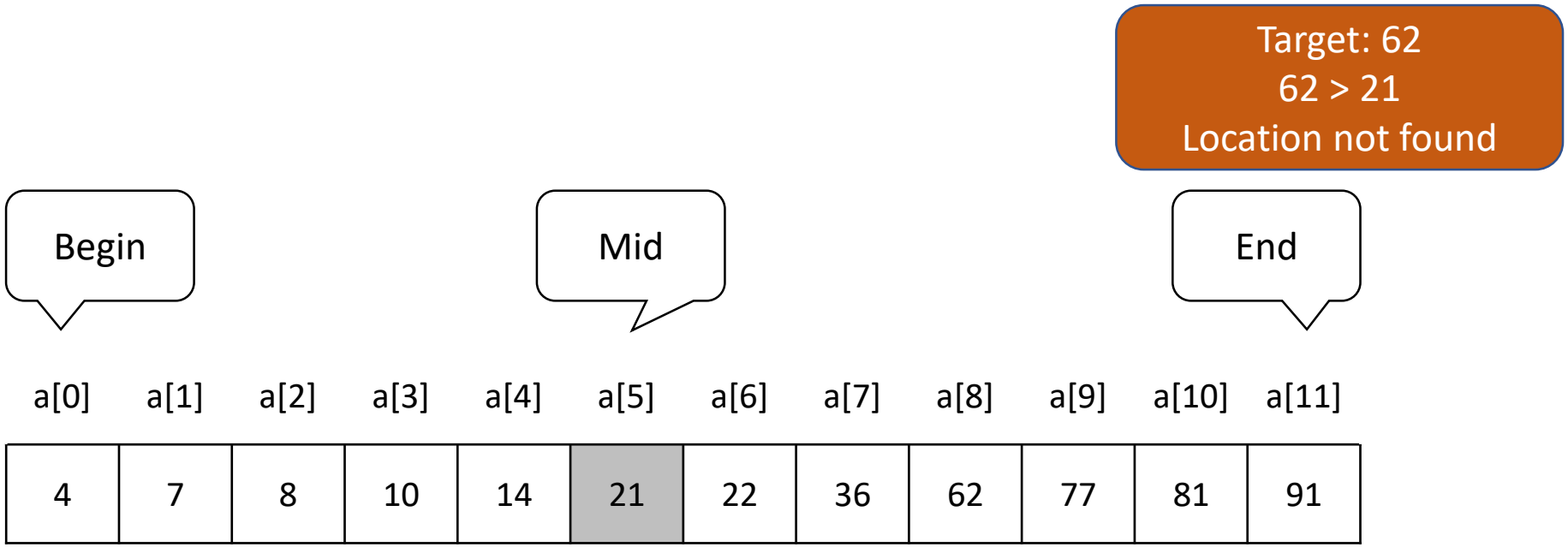


Binary Search - กรณีหาค่าไม่เจอ

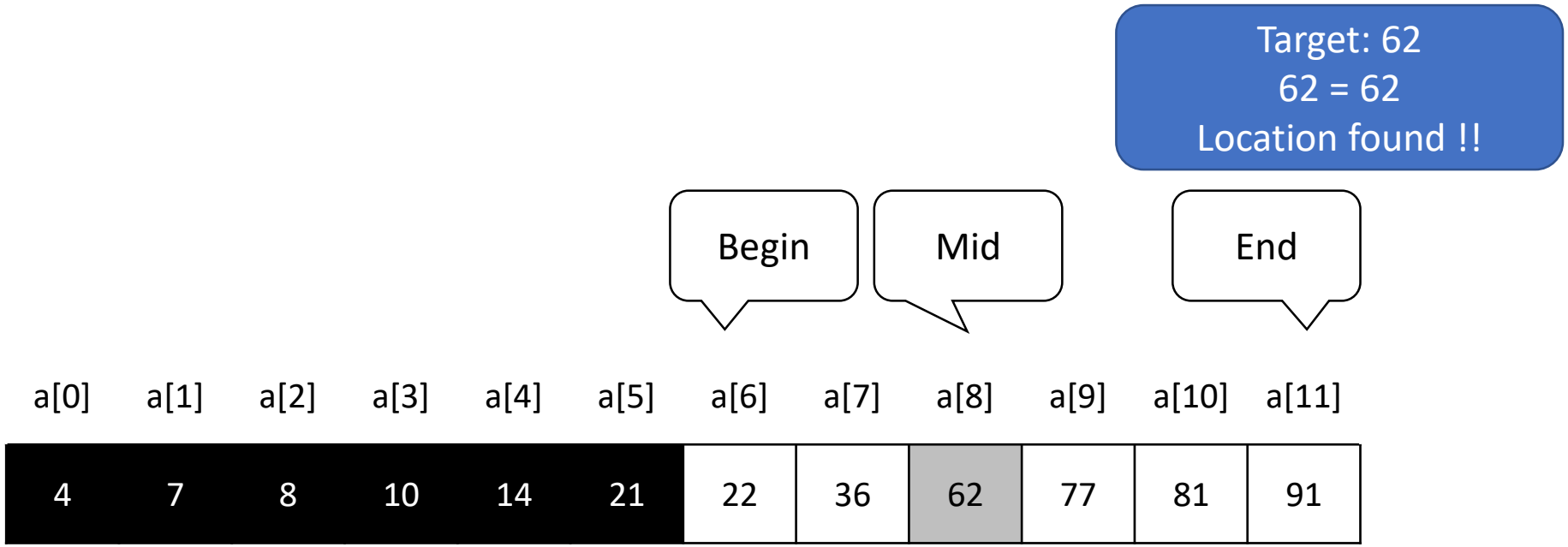
Target: 11
11 < 14
Location not found



Binary Search - กรณีหาค่าไม่เจอ



Binary Search - กรณีหาค่าไม่เจอ



$O(\log n)$

ตารางเปรียบเทียบการค้นหาข้อมูลแบบ Binary Search และ Sequential Search

List size	Iterations	
	Binary Search	Sequential Search
16	4	16
50	6	50
256	8	256
1,000	10	1,000
10,000	14	10,000
100,000	17	100,000
1,000,000	20	1,000,000

การค้นหาแบบลำดับดัชนี (Indexed sequential search)

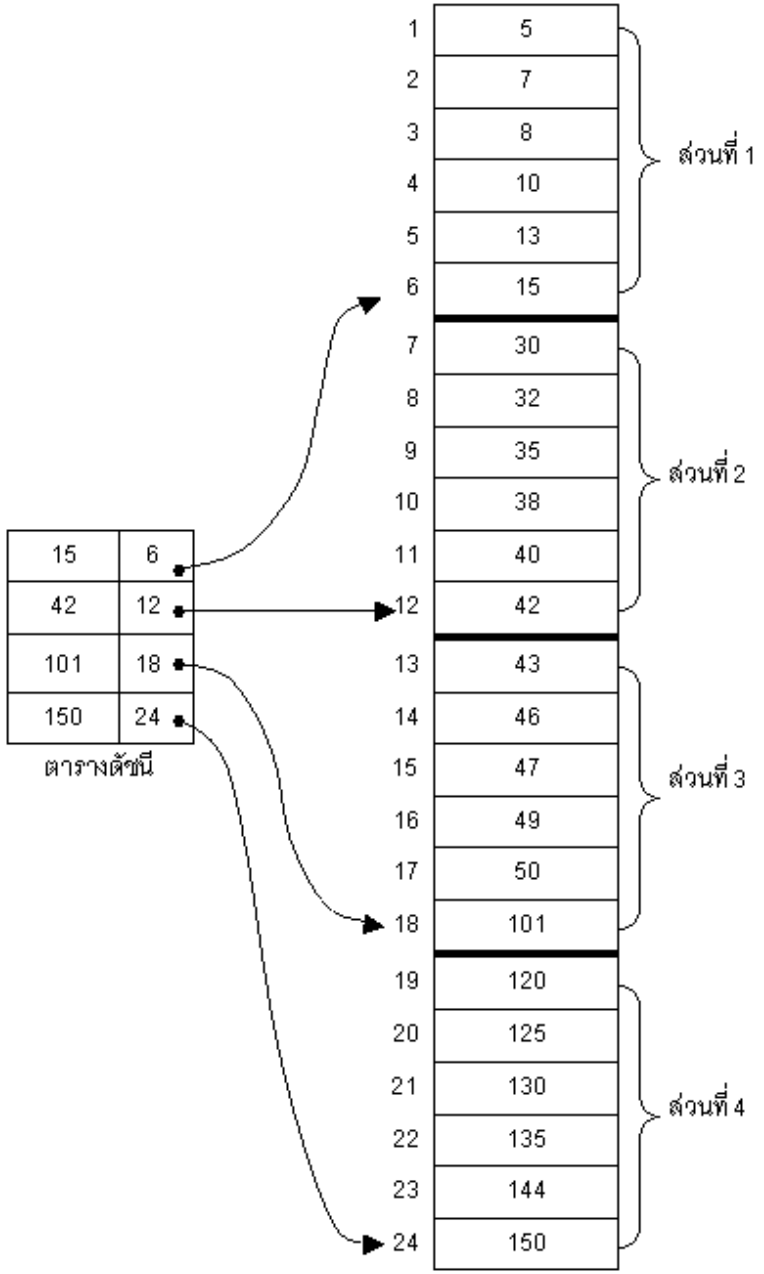
- เป็นวิธีการค้นหาในข้อมูลที่เรียงลำดับแล้ว
- โดยมากใช้กับโครงสร้างที่มีข้อมูลจำนวนมากเพื่อการค้นหาที่รวดเร็ว
- โดยแบ่งข้อมูลออกเป็นส่วน ๆ แล้วสร้าง ตารางดัชนี (index table) เพื่อเก็บค่าสูงสุดหรือค่าต่ำสุดของข้อมูลแต่ละส่วน
- เมื่อต้องการค้นหาค่าใด ๆ จะเริ่มค้นในตารางดัชนีแบบเรียงลำดับก่อน เพื่อหาว่าข้อมูลนั้นควรจะอยู่ในส่วนใดของข้อมูลทั้งหมด
- แล้วจึงไปค้นหาที่ส่วนนั้นแบบเรียงลำดับในข้อมูลจริงต่อไป

ตัวอย่างการค้นหาแบบ Indexed Sequential Search

- แสดงการค้นหาแบบลำดับข้อมูลเลขจำนวนเต็ม ซึ่งมีการเรียงลำดับจากน้อยไปมาก โดยใช้วิธีการค้นหาแบบลำดับดัชนีจากข้อมูล

  5 7 8 10 13 15 30 32 35 38 40 42 43 46 47 49 50 101 120 125 130 135 144 150

- ข้อมูลทั้งหมดเรียงลำดับจากน้อยไปมากมีจำนวนทั้งหมด 24 จำนวน เมื่อแบ่งข้อมูลออกเป็นส่วนละ 6 จำนวนจะได้ข้อมูลทั้งหมด 4 ส่วน และค่าในตารางดัชนีจะเก็บค่าสูงสุดและที่อยู่ของค่าสูงสุดของแต่ละส่วนไว้



ตัวอย่างการค้นหาแบบ Indexed Sequential Search

- *การสร้างตารางดัชนีจะมีเขตข้อมูลดัชนีแต่ละช่วงเป็นไปตาม สมการ ดังนี้*
- *โดย* M *: จำนวนระเบียนข้อมูลทั้งหมด*
- N *: จำนวนช่วงของข้อมูล*
- *จำนวนระเบียนข้อมูลในแต่ละช่วง (ช่วงใด ๆ) =* M/N *(ปัดเศษทิ้ง)*
- จำนวนระเบียนข้อมูลในช่วงสุดท้าย = $M-\{(M/N \times (N-1)\}$

Hashing Search

- การค้นหาแบบแฮชชิง **(Hashing Search)** เป็นวิธีการค้นหาข้อมูลที่พยายามให้มีการเปรียบเทียบค่าหลักให้น้อยที่สุด ในการค้นหาแต่ละครั้งค่าหลักจะเป็นค่าที่บอกตำแหน่งที่อยู่ของค่าหลักนั้นว่าถูกเก็บอยู่ที่ใด โดยการนำค่าหลักมาแปลงให้เป็นที่อยู่หรือแอดเดรสของเรคคอร์ดที่ค่าหลักนั้นอยู่ ซึ่งไม่ต้องเปรียบเทียบกับค่าหลักตัวอื่น ๆ เลย สามารถเข้าถึงข้อมูลได้โดยตรง
- ประสิทธิภาพของอัลกอริทึม คือ $O(1)$
- โดยทั่วไปวิธีที่ทำให้การค้นหาข้อมูลเร็วที่สุดโดยไม่ต้องมีการเปรียบเทียบกับค่าหลักตัวอื่น ๆ เลย เราสามารถทำได้โดยการให้ค่าหลักที่ต้องการค้นเป็นค่าที่บอกว่าข้อมูลนั้นอยู่ที่ตำแหน่งใด เช่น ถ้าเรามีข้อมูลจำนวนเต็มที่มีค่าอยู่ระหว่าง 1 ถึง 10 ดังนี้

1	4	5	8	10

Hashing Search

- ข้อมูลอาจจะเก็บในแถวลำดับซึ่งต้องเตรียมเนื้อที่ไว้ให้เพียงพอ ในตัวอย่างนี้ข้อมูลมีค่าอยู่ระหว่าง 1 ถึง 10 ดังนั้นต้องเตรียมเนื้อที่ไว้ขนาด 10 ที่ใช้ข้อมูลเป็นค่าหลักหรือดัชนีที่บอกตำแหน่งที่อยู่ของข้อมูลนั้นในแถวลำดับได้เลย เมื่อต้องการค้นหาค่าหลักใดๆ ใช้ค่าหลักนั้นไปหาตำแหน่งที่อยู่ของข้อมูลได้โดยตรง ทำให้การค้นหาข้อมูลได้รวดเร็วที่สุด

- วิธีการนี้มีข้อเสีย คือ ถ้าข้อมูลมีช่วงห่างระหว่างค่าสูงสุดและต่ำสุดมาก ต้องใช้เนื้อที่จำนวนมาก ทั้ง ๆ ที่มีข้อมูลไม่มากทำให้ใช้เนื้อที่ไม่คุ้มค่า
- เช่น ถ้าข้อมูลเป็นเลขจำนวนเต็มที่มีค่าอยู่ระหว่าง 1 ถึง 1000 ต้องเตรียมเนื้อที่ทั้งหมด 1000 เรคคอร์ดเพื่อเก็บข้อมูลทั้ง ๆ ที่อาจจะมีข้อมูลอยู่จริงแค่ 50 จำนวนเท่านั้น

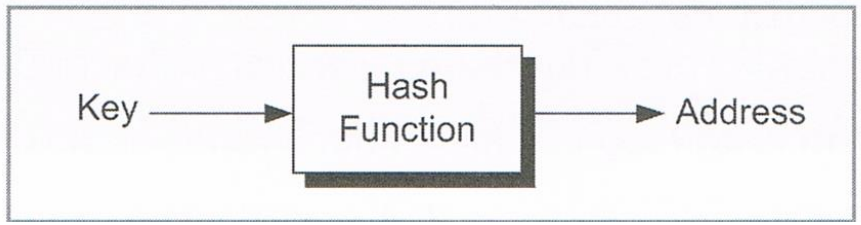
index	value
0	0
1	
2	
3	
4	4
5	5
6	
7	7
8	
9	9

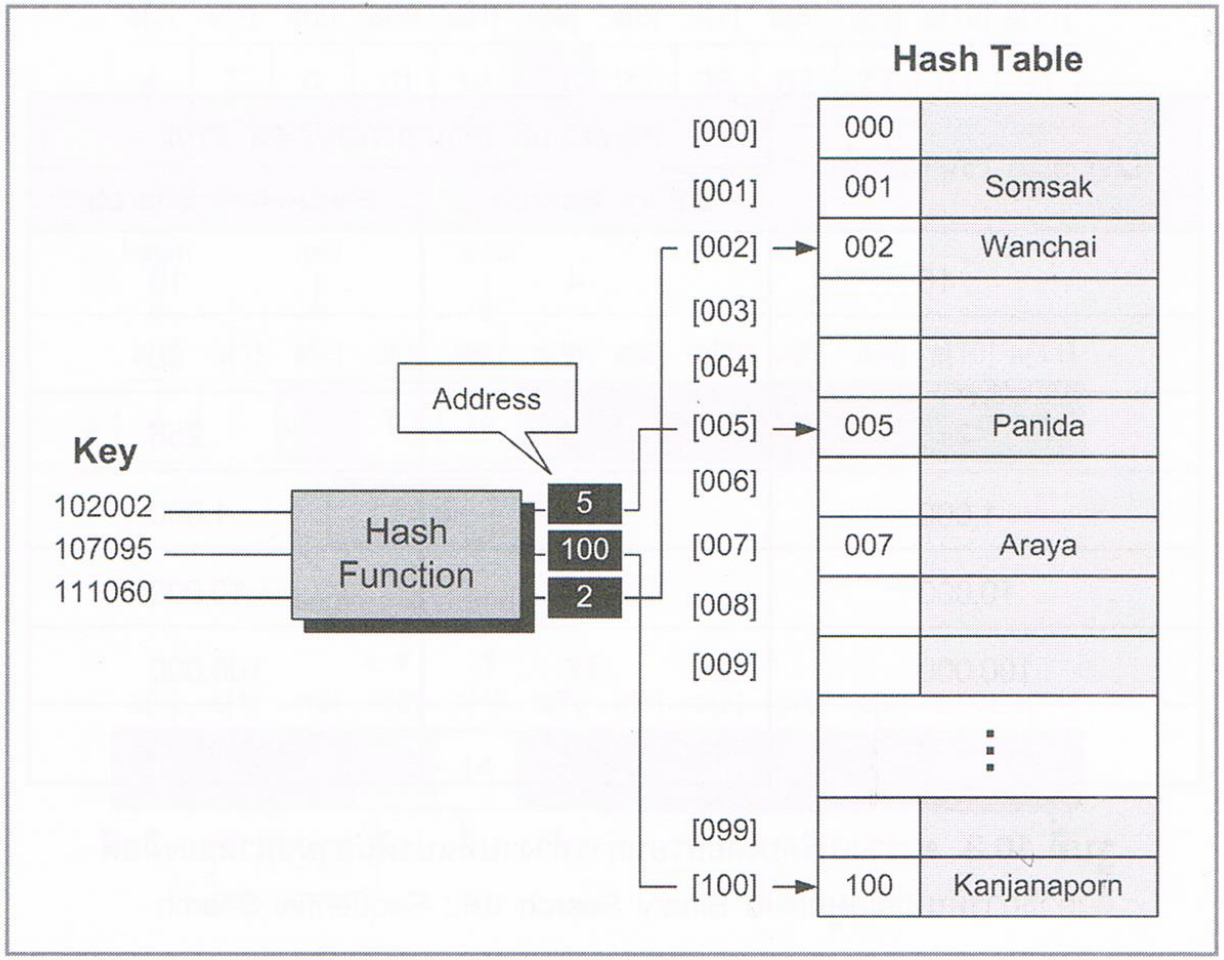
แสดงค่าหลักที่ใช้เป็นตำแหน่งที่อยู่ของเรคคอร์ด

Hashing Search

- ดังนั้นเพื่อให้การเก็บข้อมูลเป็นไปอย่างคุ้มค่าที่สุดเราต้องหาวิธีแปลงค่าหลักให้เป็นค่าที่อยู่ภายในช่วงจำกัดช่วงหนึ่งเท่านั้นโดยพยายามให้มีการซ้ำกันให้น้อยที่สุด
- วิธีการ**แปลงค่าหลักให้เป็นค่าดัชนีบอกตำแหน่งที่เก็บค่าหลักนั้นเรียกว่า แฮชชิง (hashing)** *ฟังก์ชันหรือสูตรที่ใช้ในการแปลงค่าหลักไปเป็นค่าดัชนีบอกตำแหน่งเรียกว่า ฟังก์ชันแฮช (hash function)*
- **และตารางที่ใช้เก็บค่าหลักเรียกว่า ตารางแฮช (hash table)**

Hashing Search

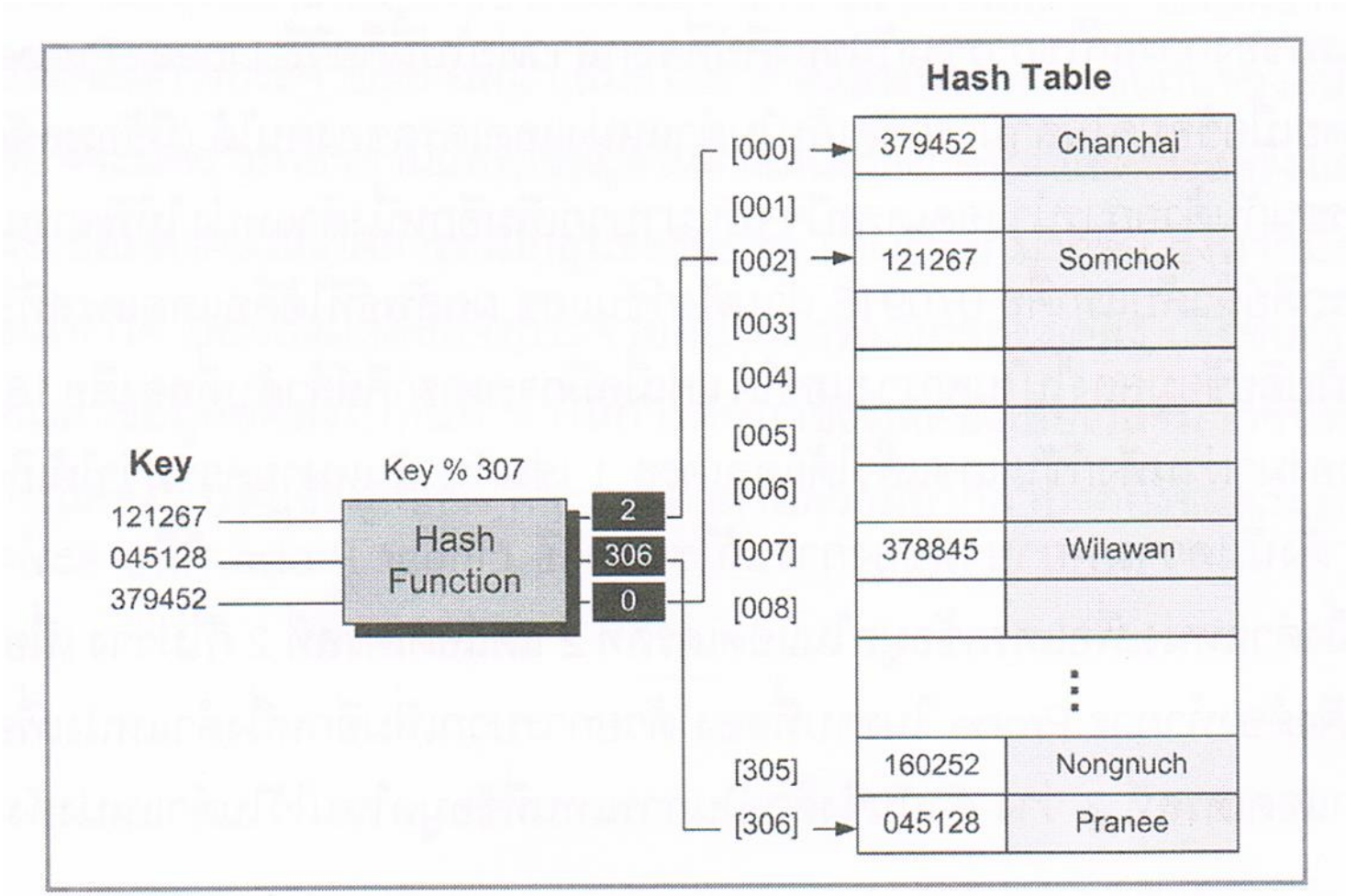




Hashing Search

- ในการแปลงค่าหลักด้วยฟังก์ชันแฮชมีฟังก์ชันแฮชหลายแบบ
- ในแต่ละแบบพบว่าบางครั้งอาจเกิดกรณีค่าหลักที่แตกต่างกันเมื่อผ่านการแปลงด้วยฟังก์ชันแฮชที่เลือกแล้วได้ค่าดัชนีบอกตำแหน่งที่ตรงกันเรียกเหตุการณ์นี้ว่า เกิดการชนกัน (collision)
- เมื่อเกิดการชนกันของค่าหลักเราต้องแก้ไขปัญหานี้โดยหาตำแหน่งที่เก็บค่าหลักใหม่

Hashing Search



การแฮชชิงด้วยวิธีการหาร

Hashing Search

- ฟังก์ชันแฮชที่ดี คือ ฟังก์ชันที่ทำให้เกิดการชนกันน้อยที่สุดเท่าที่จะทำได้ และค่าดัชนีบอกตำแหน่งที่ได้ควรจะกระจายไปทั่วทุกค่าของตารางแฮช
- โดยทั่วไปตารางแฮชต้องมีขนาดใหญ่กว่าจำนวนค่าหลักที่จะถูกเก็บจริง ยิ่งตารางแฮชมีขนาดใหญ่มากเท่าไรยิ่งทำให้การเกิดการชนกันน้อยที่สุด
- แต่ขณะเดียวกันก็ทำให้เปลืองเนื้อที่มากเท่านั้นด้วย แต่อย่างไรก็ตามการเกิดการชนกันน้อยที่สุดเป็นวิธีการที่ให้ประสิทธิภาพในการทำงานสูงสุด

วิธีการหาร (division method)

- เป็นวิธีที่นิยมและรู้จักกันอย่างแพร่หลาย ได้จากการนำค่าหลักที่เป็นเลขจำนวนเต็มหารด้วยขนาดของตารางแฮช และเศษที่ได้จากการหารคือค่าดัชนีของเลขที่ในตารางแฮชนั่นเอง

- ถ้ากำหนดให้ k คือค่าหลักและ n คือขนาดของตารางแฮช สามารถเขียนสัญลักษณ์แทนฟังก์ชันแฮชที่ได้ดังนี้

$$\mathrm{h}(k) \longrightarrow k \bmod n$$

- $\mathrm{h(k)}$ ให้ค่าที่เป็นเศษที่ได้จากการหารค่าหลัก key ด้วย n และถ้า n มีค่าเป็นจำนวนเฉพาะ (prime number) ยิ่งทำให้การเกิดการชนกันน้อยที่สุดด้วย

วิธีการหาร (division method)

- ข้อมูลจำนวนเต็มบวกชุดหนึ่งมีค่าดังนี้

  125 168 289 176 263

สมมติว่าเลือก n มีค่าเท่ากับ 7 จะได้ฟังก์ชันแฮชและค่าในตารางแฮชดังนี้

$$h(k) = k \bmod 7$$

k	h(k)
125	6
168	0
289	2
176	1
263	4

0	1	2	3	4	5	6
168	176	289		263		125

วิธีวิเคราะห์ตัวเลข (digit analysis method)

- เป็นวิธีการวิเคราะห์ตัวเลขของค่าหลักในบางตำแหน่ง โดยพิจารณาจากตัวเลขที่ตำแหน่งต่าง ๆ ที่มีการกระจายของตัวเลขอย่างสม่ำเสมอ และใช้เลขเหล่านั้นมาพิจารณาเป็นค่าดัชนีบอกตำแหน่งในตารางแฮช เนื่องจากค่าดัชนีที่ได้จะเกิดการชนกันน้อยที่สุด

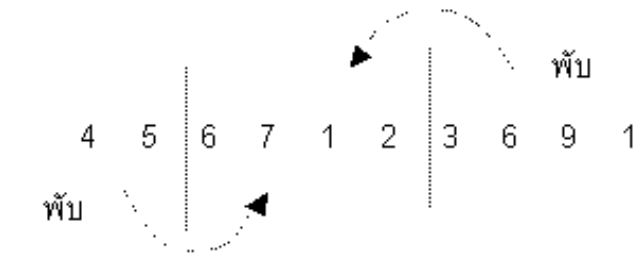
k	h(k)
2 2 8 9 1	29
3 0 4 5 6	05
3 1 4 2 1	12
2 8 4 1 5	81
4 5 2 1 4	51
3 4 1 4 6	44
2 6 9 5 8	65
2 9 4 3 6	93
.	.
.	.
.	.

- สมมติค่าหลักจำนวนหนึ่ง ซึ่งแต่ละค่ามีตัวเลขจำนวน 5 ตำแหน่ง
- สมมติว่ามีจำนวนค่าหลักทั้งหมดเท่ากับ 100 จำนวน
- ในตารางแฮชต้องใช้เนื้อที่สำหรับบรรจุค่าหลักทั้งหมดอย่างน้อย 100 ค่า และค่าดัชนีของเลขที่ตำแหน่งในตารางแฮชเป็น 00 ถึง 99
- ดังนั้นสามารถใช้ตัวเลข 2 หลักได้ และเมื่อวิเคราะห์ตัวเลขในตำแหน่งต่าง ๆ ได้ว่า ในตำแหน่งที่ 2 และ 4 ตัวเลขมีการกระจายสม่ำเสมอ ดังนั้นจะใช้ค่าหลักใน 2 ตำแหน่งนี้แปลงไปเป็นค่าดัชนีบอกตำแหน่งในตารางแฮช

วิธีพับตัวเลข (folding method)

- เป็นวิธีที่แบ่งค่าหลักออกเป็นส่วน ๆ ในแต่ละส่วนเท่ากับจำนวนหลักของค่าดัชนีบอกตำแหน่งในตารางแฮชที่ต้องการ เอาแต่ละส่วนพับเข้าหากันแล้วจึงบวกกัน ผลบวกที่ได้ตัดตัวทดออกจะได้ค่าดัชนีบอกตำแหน่งที่ต้องการ

- ค่าหลัก 4 5 6 7 1 2 3 6 9 1 และต้องการค่าดัชนีบอกตำแหน่งขนาด 4 หลัก แสดงวิธีพับตัวเลขเพื่อหาดัชนีบอกตำแหน่ง



หลังจากพับตัวเลขนำตัวเลขทั้งหมดมาบวกกันได้ดังนี้

```
  6 7 1 2
  1 9 6 3 +
  5 4
-----------
1 4 0 7 5   <-- ดัชนีบอกตำแหน่ง
===========
```

- ผลบวกคือ 14075 ตัดตำแหน่งที่ทดออกได้ค่าดัชนีบอกตำแหน่งในตารางแฮชขนาด 4 หลักเป็น 4075

Hash Function

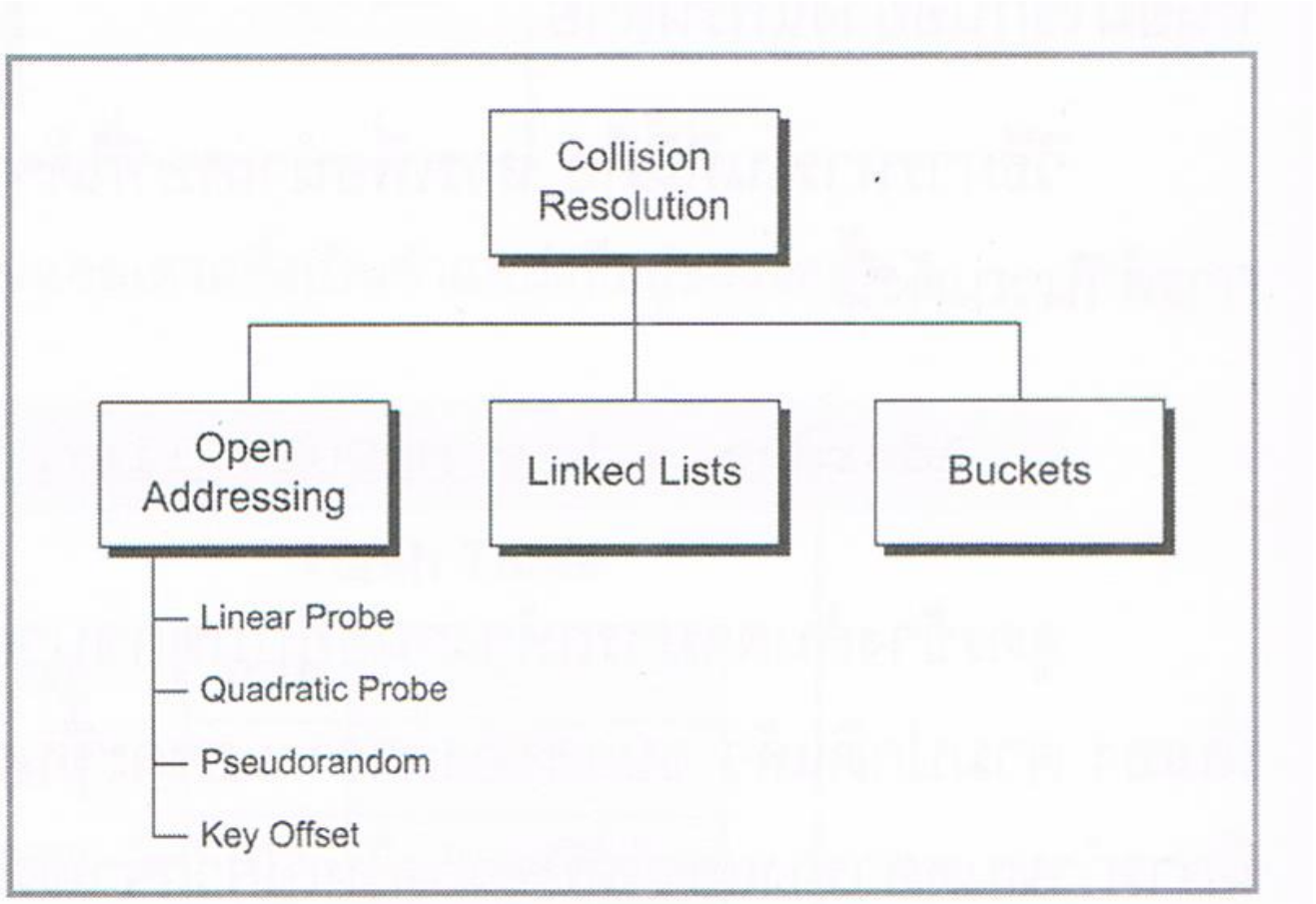
- **วิธีกลางกำลังสอง (mid - square method)** เป็นวิธีที่นำค่าหลักมายกกำลังสองแล้วเลือกตัวเลขในช่วงกลาง ๆ มาเป็นค่าดัชนีบอกตำแหน่งที่ต้องการ โดยเลือกจำนวนตัวเลขให้เท่ากับจำนวนหลักของค่าดัชนีในตารางแฮช

ค่าหลัก	ยกกำลังสอง	ดัชนีบอกตำแหน่ง
834901	6970596779801	596
413618	171079849924	798
756281	571960950961	609
360159	129714505281	145
193482	37435284324	352

เทคนิคการแก้ปัญหาเมื่อเกิดการชนกัน

- Open Addressing
- Linked List หรือ Chaining
- Buckets
- Rehashing

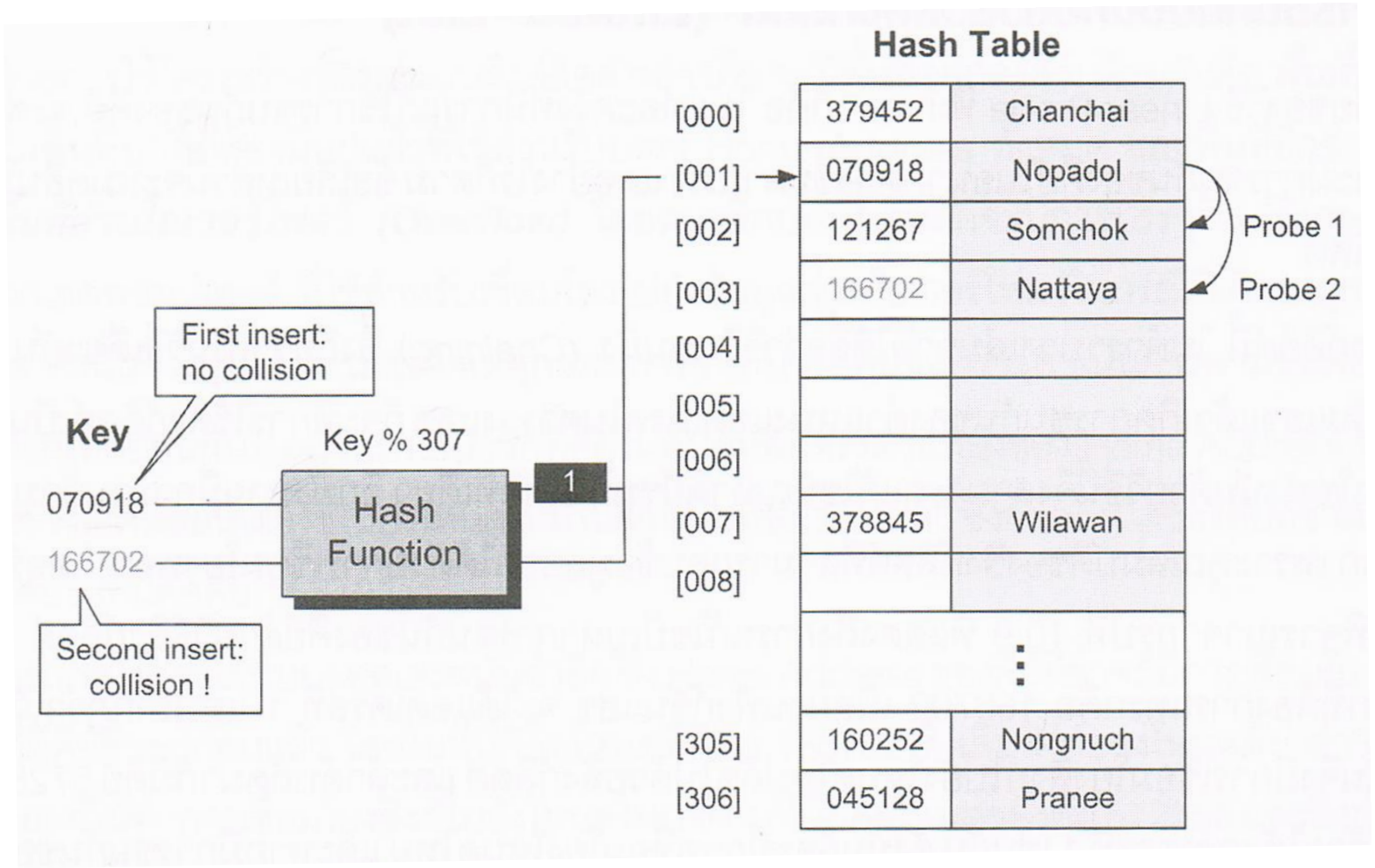
เทคนิคการแก้ปัญหาเมื่อเกิดการชนกัน



Open Addressing ด้วยวิธี Linear Probe

- เป็นวิธีแก้ปัญหาที่ง่ายที่สุดเมื่อเกิดการชนกันให้พิจารณาตำแหน่งที่ว่างถัดไป ถ้ามีที่ว่างก็เก็บค่าหลักนั้นได้เลยแต่ถ้ามีค่าหลักอื่นอยู่ก่อนแล้วให้พิจารณาตำแหน่งที่ว่างถัดไปเรื่อย ๆ จนกว่าจะมีที่ว่างสำหรับเก็บค่าหลักนั้นหรือจนกว่าจะพบตำแหน่งที่เคยพิจารณาเป็นครั้งแรก ซึ่งหมายความว่าตารางแฮชเต็มแล้ว

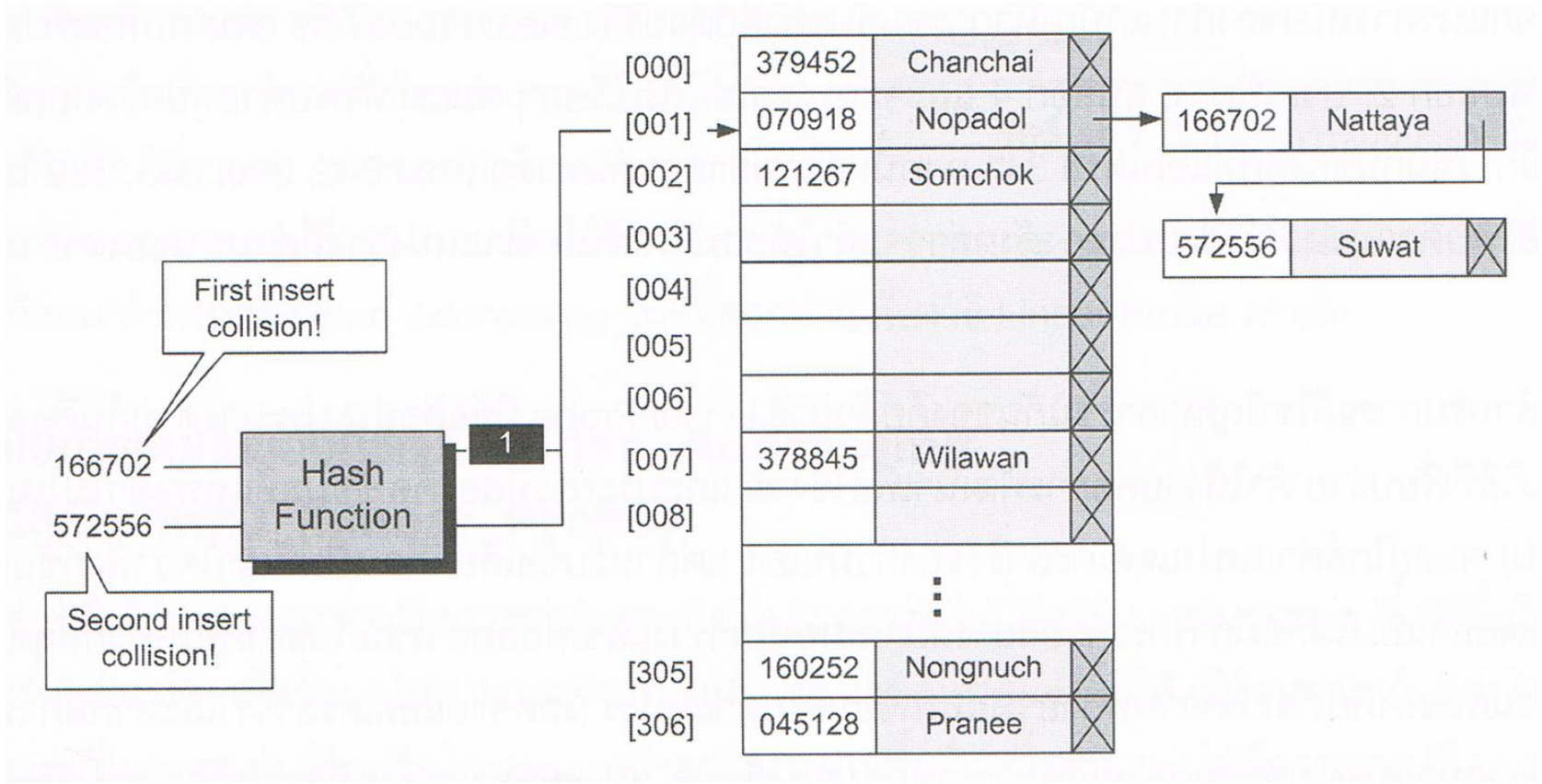
Open Addressing ด้วยวิธี Linear Probe



Linked List หรือ Chaining

- เมื่อคีย์ได้ผ่านฟังก์ชันแฮชแล้วเกิดการชนกันของตำแหน่งแอดเดรสในตารางแฮช ก็จะมีการใช้ลิงก์ลิสต์เป็นตัวเชื่อมโยงถัดไปเป็นลูกโซ่
- สามารถแก้ปัญหาคีย์ชนกันได้เป็นอย่างดี อีกทั้งหากมีการลบข้อมูล ก็สามารถทำได้ง่ายมาก เพราะคุณสมบัติของลิงก์ลิสต์ที่สามารถลบข้อมูลออกได้ด้วยการเชื่อมโยงพอยน์เตอร์ใหม่ได้ทันที

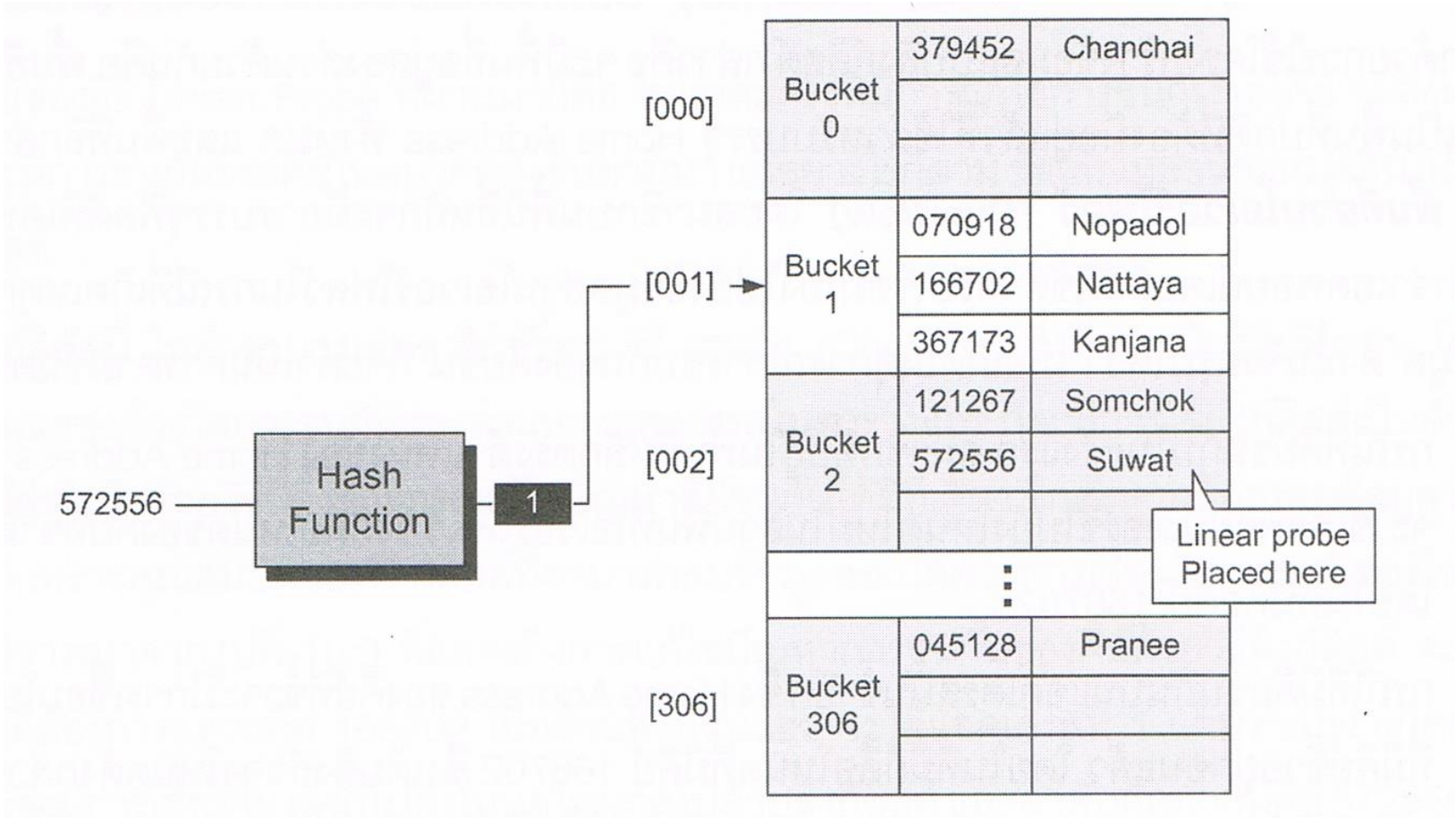
Linked List หรือ Chaining



Bucket Hashing

- เมื่อคีย์ได้ถูกแฮชลงใน Bucket ที่เสมือนกับตะกร้าแล้ว คีย์ที่ชนกันยังสามารถบรรจุลงในตารางแฮชร่วมกันภายในตะกร้าเหล่านั้นได้
- มีการจัดสรรข้อมูลตำแหน่งที่จัดเก็บในรูปแบบของตารางหลายช่อง และหากมีการชนกันของคีย์อีก ก็จะจัดเก็บลงในตารางตำแหน่งถัดไปจนกระทั่งเต็ม

Bucket Hashing



Rehashing

- เป็นวิธีแก้ปัญหาการเกิดการชนกันโดยการนำค่าหลักที่เกิดการชนกันมาทำแฮชชิงอีกด้วยฟังก์ชันแฮชอันใหม่
- เช่น ถ้าให้ $\mathrm{h}(k)$ เป็นฟังก์ชันแฮชโดยวิธีหาร และ $\mathrm{h1}(k)$ เป็นฟังก์ชันแฮชที่ใช้ทำแฮชชิงใหม่จะได้ดังนี้

$$\mathrm{h}(k) = k \mod m$$
$$\mathrm{h1}(k) = (\mathrm{h}(k) + c) \mod m$$

- โดยที่ k คือค่าหลัก c และ m เป็นควรเป็นจำนวนเฉพาะ (prime number) จึงจะทำให้ได้ค่าดัชนีบอกตำแหน่งในตารางแฮชที่มีการกระจายมากที่สุด

Hashing Game

Anagram

- SILENT
- LISTEN

- Sort(SILENT) = Sort (LISTEN) = EILNST

Anagram

- PERFECTIONISM
- IMPERFECTIONS

- Sort(PERFECTIONISM) = Sort (IMPERFECTIONS) = CEEFIIMNOPRST

Anagram with Sorting

```
public static bool AreAnagramsViaArraySort(string value1, string value2)
{
  if (value1 is null) throw new ArgumentNullException(nameof(value1));
  if (value2 is null) throw new ArgumentNullException(nameof(value2));

  if (value1.Length != value2.Length) { return false; }

  var content1 = value1.ToCharArray();
  Array.Sort(content1);
  var content2 = value2.ToCharArray();
  Array.Sort(content2);

  for (var i = 0; i < value1.Length; i++)
  {
    if(content1[i] != content2[i]) { return false; }
  }

  return true;
}
```

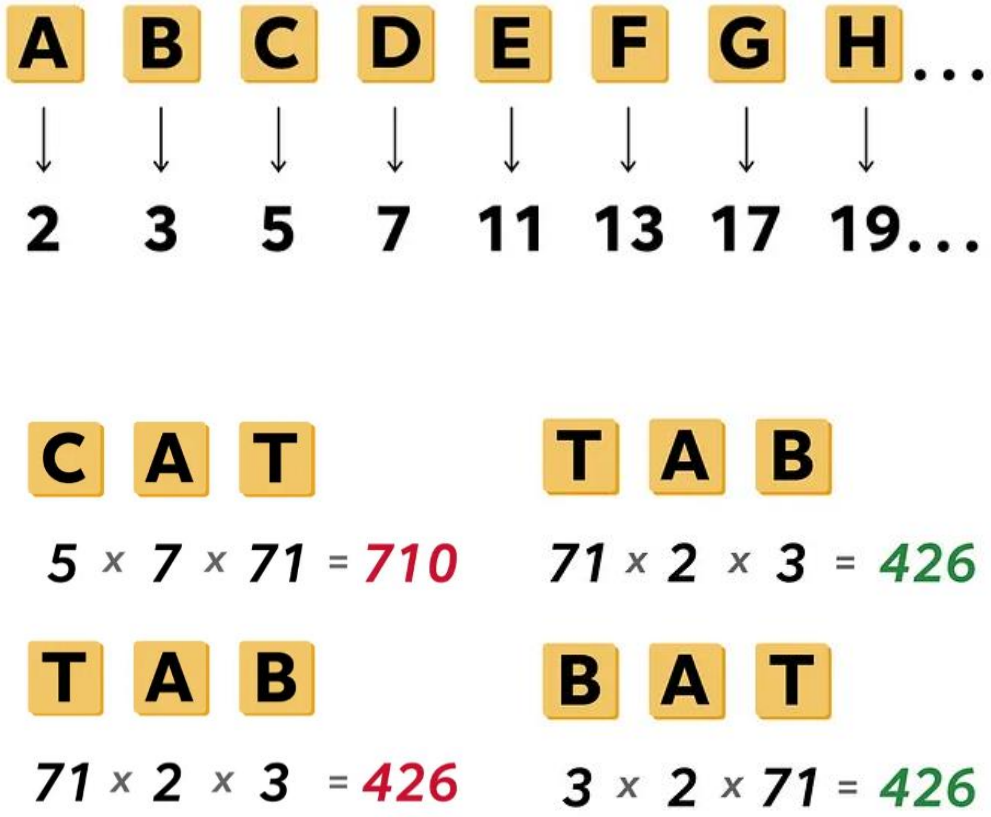
$O(n \log n)$

Anagram with Bucketing

```
int cnt1[26] = {0};
int cnt2[26] = {0};

for(int i = 0; i < word1.length(); i++){
          word1[i] = tolower(word1[i]);
          word2[i] = tolower(word2[i]);
          cnt1[word1[i] - 'a']++;
          cnt2[word2[i] - 'a']++;
}

for(int i = 0; i < 26; i++){
          if(cnt1[i] != cnt2[i]){
          cout << "The words are not anagrams";
          return 0;
          }
}
```

$O(2n)$

Anagram with Hashing




```
public BigInteger GetAnagramNumberUsingSwitchLetterDistribution(this string self)
{
  if (self is null) throw new ArgumentNullException(nameof(self));

  const ulong MaximumValue = 182_641_030_432_767_837;

  var currentValue = 1ul;
  var value = BigInteger.One;

  for (var i = 0; i < self.Length; i++)
  {
    currentValue *= self[i] switch
    {
      'a' => 2,
      'b' => 3,
      'c' => 5,
            // ... Values ommitted for brevity ...
      'x' => 89,
      'y' => 97,
      'z' => 101,
      _ => throw new NotSupportedException()
    };

    if ((i == self.Length - 1) || (currentValue > MaximumValue))
    {
      value *= currentValue;
      currentValue = 1ul;
    }
  }

  return value;
}
```

$O(n)$

Anagram with Hashing

```
private static readonly Dictionary<char, BigInteger> mappings = new()
{
  { 'a', 2 },
  { 'b', 3 },
  { 'c', 5 },
        // ... values deleted for brevity ...
  { 'x', 89 },
  { 'y', 97 },
  { 'z', 101 },
};
```

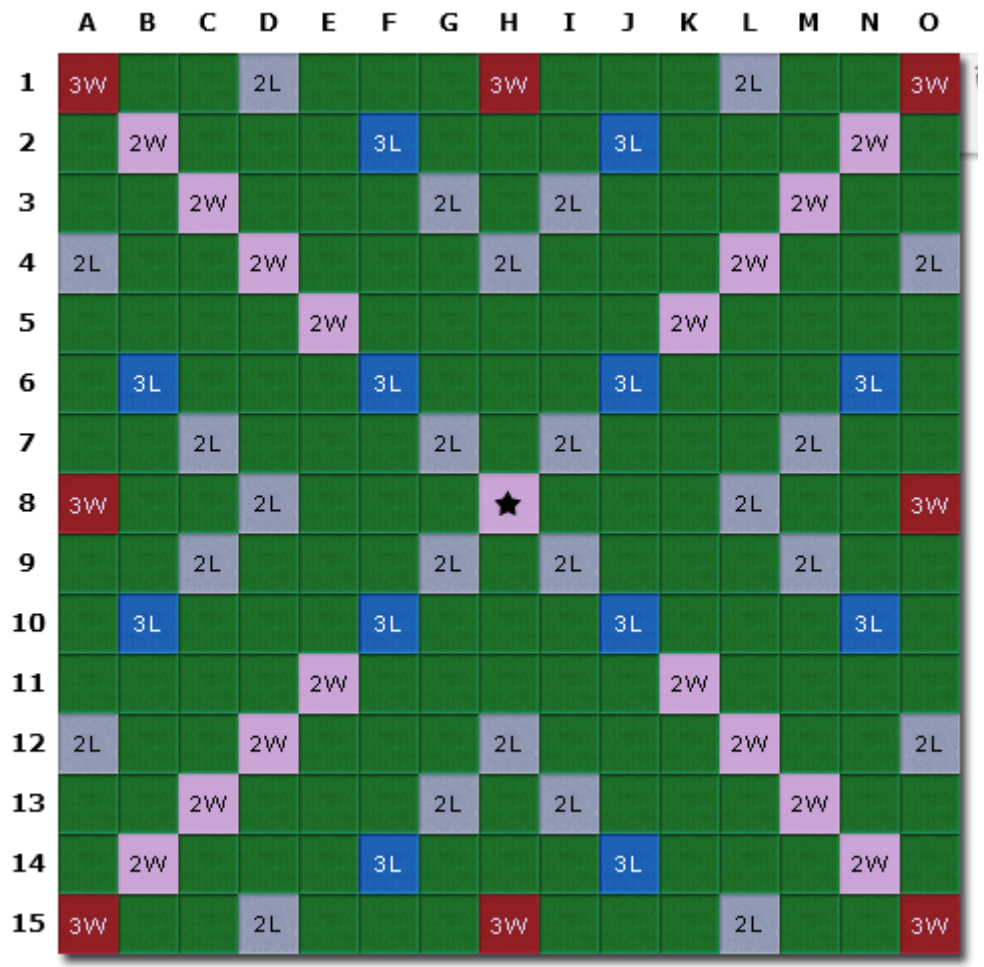
```
public static BigInteger GetAnagramNumber(this string self)
{
  if (self is null) throw new
ArgumentNullException(nameof(self));

  var value = BigInteger.One;

  foreach (var piece in self)
  {
    value *= AnagramComparisons.mappings[piece];
  }

  return value;
}
```

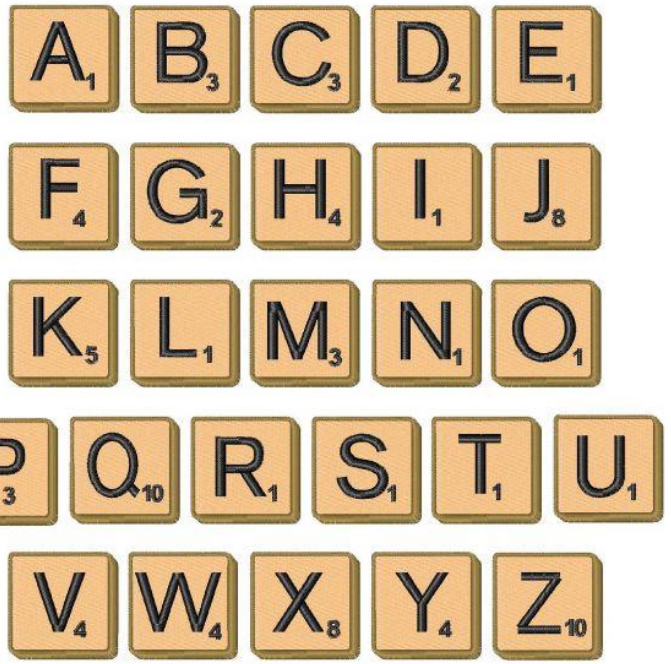
$O(n)$

Scrabble Game

Hash(fried) = Hash(fired)
Hash(gainly) = Hash(laying)
Hash(sadder) = Hash(dreads)
Hash(earth) = Hash(heart)
Hash(triangle) = Hash(integral)
Hash(meat) = Hash(team)
Hash(race) = Hash(care)
Hash(begin) = Hash(being)
Hash(act) = Hash(cat)
Hash(angle) = Hash(angel)
Hash(cider) = Hash(cried)
Hash(night) = Hash(thing)
Hash(elbow) = Hash(below)
Hash(players) = Hash(parsley)
Hash(dusty) = Hash(study)

Scrabble Game





= 3+1+2+1+1=8 pts

= 3+1+1+1+2=8 pts

Scrabble Game : BEGIN with BEING

= 3+1+2+1+(2x1) = 9 pts

= 3+1+1+1+(2x2) = 10 pts

Scrabble Game :: Subset Selection

- Without E and N
  - We will have BIG and GIB
- Without B and E
  - We will have GIN and NIG

- Sort and Remove Recursively
  - Sort(BEGIN) = BEGIN
    - Remove one by one and Hash() them
      - Hash(EGIN), Hash(BGIN), Hash(BEIN), Hash(BEGN) and Hash(BEGI)
        - Hash(GIN), Hash(EIN), Hash(EGN) and Hash(EGI)
          - Hash(IN), Hash(GN) and Hash(GI)