POLYNOMIAL HASH CODE

- Instead of summation, let's use a **polynomial** hash code, which works as follows:
  - **Cut** k into multiple pieces, $x_0, x_1, \ldots, x_{m-1}$ that each takes ≤ 32 bits
  - **Cast** each piece into `int`
  - **Set** a constant a
  - **Compute**: $x_0 a^{m-1} + x_1 a^{m-2} + \cdots + x_{m-3} a^2 + x_{m-2} a + x_{m-1}$
- This is better than summation because **it considers the order of the pieces**.

- Be careful! The final result could become very large, and may **overflow,** i.e., not fit in the 32-bit of the integer, in which case would simply ignore the extra bits!
- To minimize the chance of that, choose a to be a number that takes only few bits.

POLYNOMIAL HASH CODE

- We can choose the value of a based on experiments!
- For example, the authors of the textbook ran some experiments where the keys are taken from a list of 50,000 English words
- They found that taking $a \in \{33, 37, 39, 41\}$ produced $<$ 7 collisions in each case!

CYCLE-SHIFT HASH CODE

- An alternative to **polynomial hash code** is the **cycle-shift** hash code:
  - **Cut** k into multiple pieces, $x_0, x_1, \ldots, x_{m-1}$ that each take ≤ 32 bits
  - **Shift** the hash code *h* by some bits (initially *h* = 0)
  - **Cast** x_0 into `int` and **Add** x_0 to *h*, then **shift** *h* by some bits
  - **Cast** x_1 into `int` and **Add** x_1 to *h*, then **shift** *h* by some bits
  - and so on . . .

- To implement this, **we need to use bitwise operations**. Before we see the implementation, let us remind ourselves of the bitwise operations we've seen earlier this semester.

BITWISE OPERATORS

- Suppose that the bitwise representation of a variable, `a`, is `00000101`
- Suppose that the bitwise representation of a variable, `b`, is `00001001`

Bitwise operator:		Example:
`~exp`	bitwise **complement**	`~a // 11111010`
`exp & exp`	bitwise **and**	`a&b // 00000001`
`exp ^ exp`	bitwise **exclusive-or**	`a^b // 00001100`
`exp | exp`	bitwise **or**	`a|b // 00001101`
`exp1 << exp2`	**shift** exp1 **left** by exp2 bits	`b<<1 // 00010010`
`exp1 >> exp2`	**shift** exp1 **right** by exp2 bits	`b>>1 // 00000100`

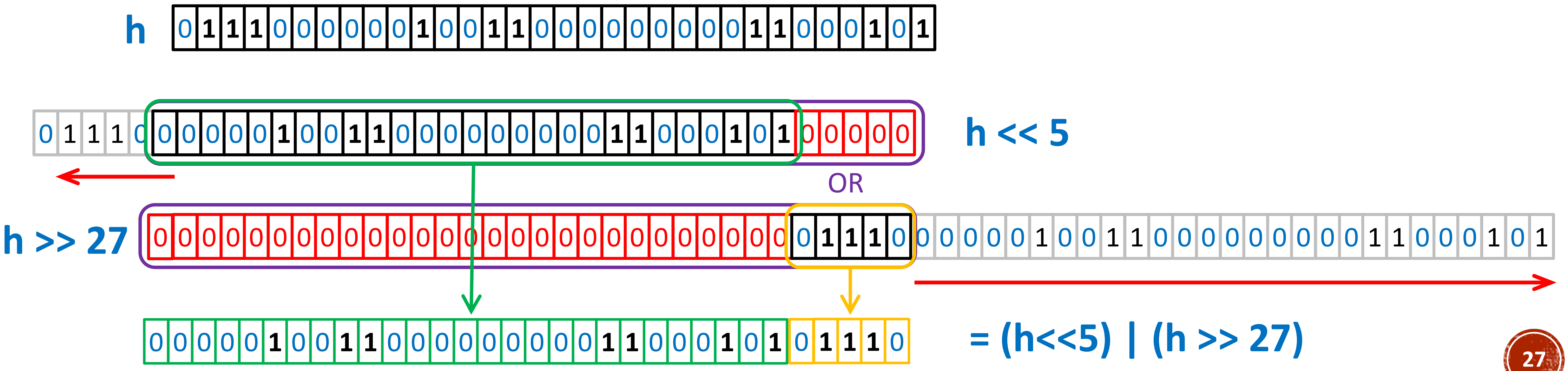
- The **left-shift operator** fills with zeros.
- The **right-shift operator** fills with zeros if `exp1` is an unsigned variable. Otherwise, it fills with 0 if `exp1` is positive, and with 1 if `exp1` is negative.
- In our cycle-shift hash code implementation, we'll use an unsigned int, so the right shift ">>" will always fill with zeros

CYCLE SHIFT HASH CODE

- Here is a C++ implementation of a **cycle shift hash code**: where the key, k, is an array of m elements of type `char`

```
int hashCode(const char* k, int m) {
    unsigned int h = 0;
    for (int i = 0; i < m; i++) {
        h = (h << 5) | (h >> 27);
        h += (unsigned int) k[i];
    }
    return h;
}
```

This line performs a **cyclic 5-bit shift**, i.e., the 5 shifted bits **wrap around** the 32 bits



CYCLE SHIFT HASH CODE

- We can choose **the number of bits to shift** based on experiments!

Example:

- Experimenting with keys taken from list of **25,000 English words**
- For each word, the cyclic shift hash code cuts the word into separate characters, then repeatedly adds a character and performs a cyclic shift.
- As can be seen, given a **5**-bit shift, there were only **4** collisions (i.e., 4 entries that would end up being inserted in an already filled bucket), and no bucket would have more than **2** entries
- Note that, **with a cyclic shift of 0**, this hash code reverts to the summation hash code

Shift	*Collisions*	
	Total	*Max*
0	23739	86
1	10517	21
2	2254	6
3	448	3
4	89	2
5	4	2
6	6	2
7	14	2
8	105	2

COMPRESSION FUNCTIONS

- If you remember, we said a hash function involves **two steps**:
  - Convert the key to an integer $\in [-\infty, \infty]$ called the **hash code**
  - Use something called a "**compression function**" that converts the hash code to an integer in $\in [0, N-1]$, where N is the size of your hash table.
- So far we talked about alternative hash codes:
  - The summation hash code
  - The polynomial hash code
  - The cycle-shift hash code
- Next, we discuss **compression functions** . . .

1) DIVISION FUNCTION

- We need a "**compression function**" that converts the hash code to an integer in $\in [0, N-1]$, where N is the size of your hash table.

- One possible compression function is the **division function**:

$$h(k) = |k| \ mod \ N$$

Example: Given N = 100 and k = −1070, the division function returns 70.

Problem: This **may lead to many collisions**, e.g., given N = 100 the following hash codes will all be converted to the same index:

- $k = -1070$
- $k = -970$
- $k = -870$
- $k = -770$
- ...

Take N to be **a prime number**!

2) THE MAD METHOD

- A better compression function is the **MAD method**, which stands for **Multiply, Add and Divide**:

$$h(k) = |\underline{a}k \underline{+ b}|\ \underline{mod\ \ N}$$

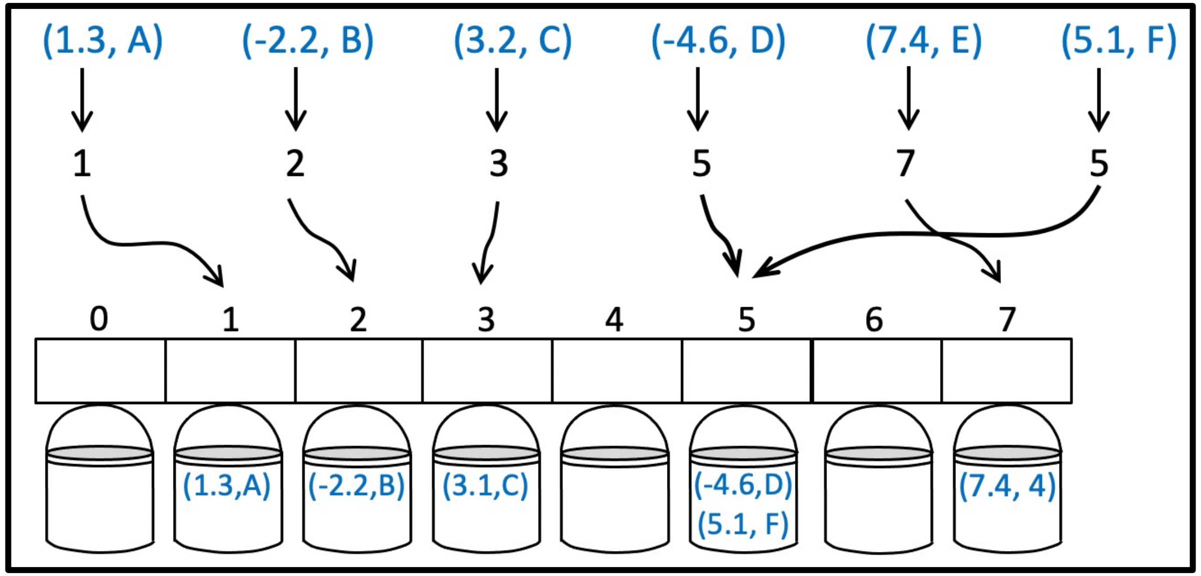
where N is a prime number, and a and b are **randomly chosen** as positive integers that are not multiples of N.

- By "**randomly chosen**", we mean that when you first create the compression function, pick a and b randomly then stick to them
  - E.g., you may randomly pick values of a and b, see how many collisions you end up with; if there are many, pick new values of a and b (which gives a new compression function), and **keep trying till you get a good function, i.e., one that minimizes the chance of collision**.

COLLISION HANDLING SCHEMES

- **Each bucket can be a map implemented as a linked list.** This way, we can readily use the methods that come with the map!

  *This is a method for handling collisions called **Separate Chaining.***



- There is another collision handling method, ***Open Addressing***, which utilizes some simple methods, including:
  - *Linear probing*
  - *Quadratic probing*

- They store the values directly into the hash table cells, in which each cell will store at most one value.

- If A[h(k)] is occupied, keep trying probing different cells until you find a vacancy:
  - Linear: $A[\underline{(h(k)+i) \bmod N}]$ For i=(1,2,3,…)
  - Quadratic: $A[\underline{(h(k)+j^2) \bmod N}]$ For j=(0,1,2,…)

  N is a prime number for both methods

33

ORDERED MAPS

ORDERED MAPS

- So far, entries in a **map** have no particular order. In an **ordered map**, entries are ordered based on their keys.
- In addition to **all the functions in a map ADT**, the **ordered map has the following**:

firstEntry(): Return an iterator to the **entry with smallest key**; if map is empty, return end.

lastEntry(): Return an iterator to the **entry with largest key**; if map is empty, return end.

ceilingEntry(k): Return an iterator to the **entry with the least key** $\geq k$; if there is no such entry, return end.

floorEntry(k): Return an iterator to the **entry with the greatest key** $\leq k$; if there is no such entry, return end.

higherEntry(k): Return an iterator to the **entry with the least key** $> k$; if there is no such entry, return end.

lowerEntry(k): Return an iterator to the **entry with the greatest key** $< k$; if there is no such entry, return end.

ORDERED MAPS

- With ordered maps, the map can be implemented as a **vector**, where entries are sorted in an ascending order based on their keys. We refer to such implementation as an ***ordered search table***.

- What is the running time for `Insert(k,v)` and `erase(k)` in ordered maps?

  - They take O(n) time, since we need to shift entries around.

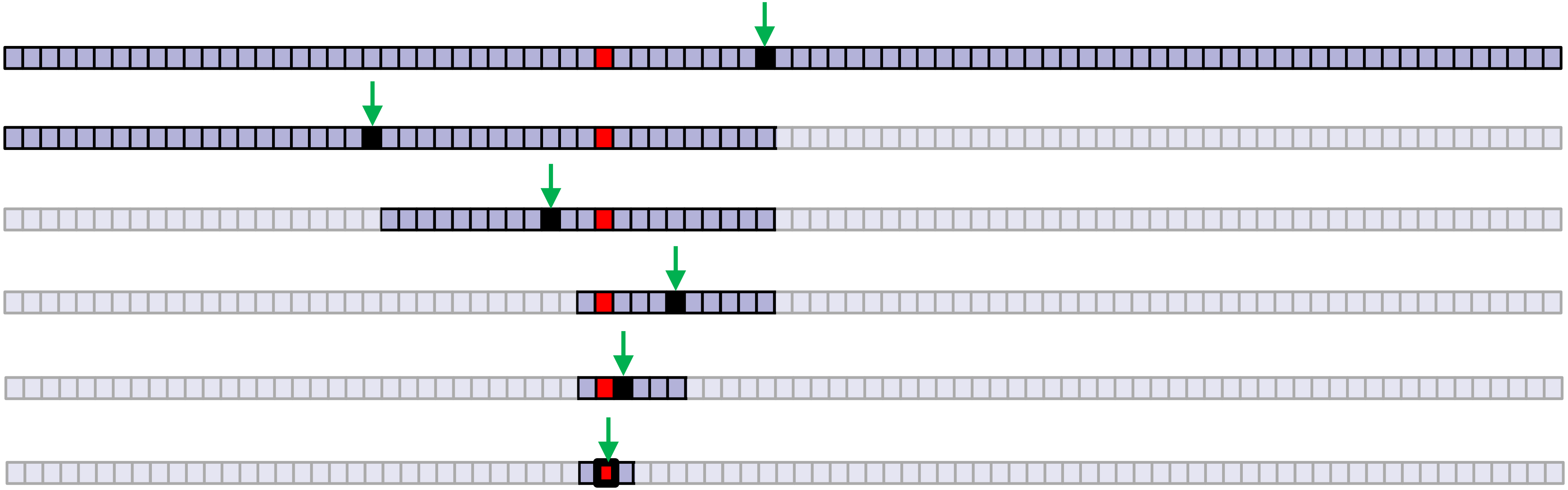
- What about `find(k)`?

  - A naïve implementation would **search through all** n **entries to find one whose key equals** k (or to determine that no such entry exists in the ordered map); this takes O(n) time.

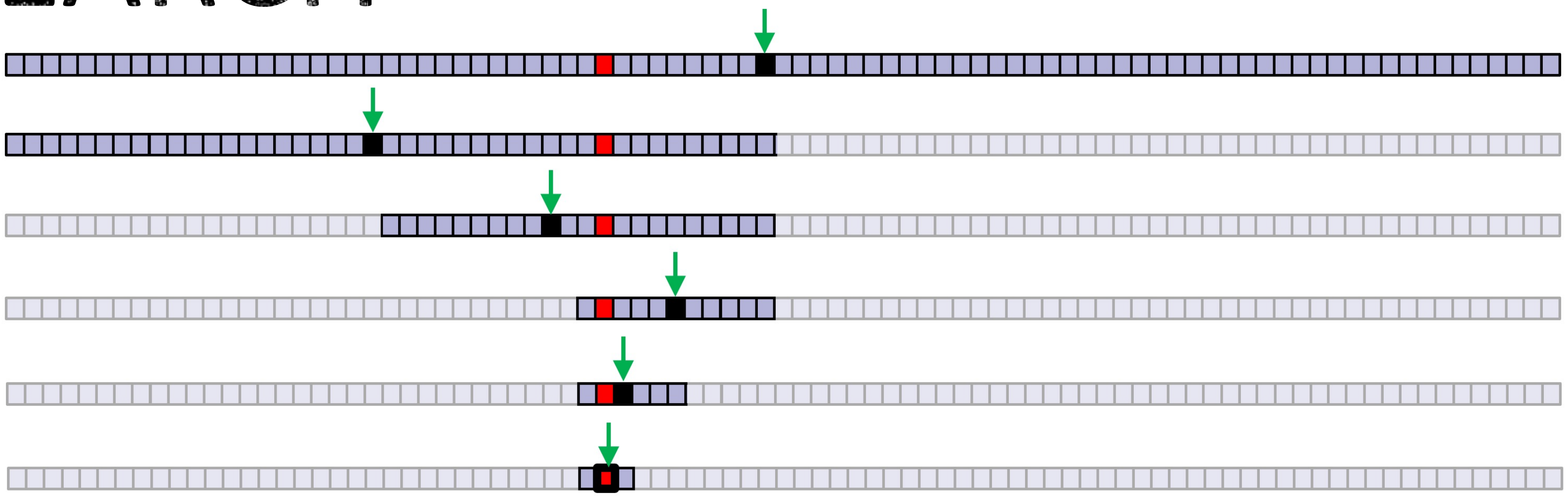
  - But we can do better using "**binary search**"

BINARY SEARCH

- Basic idea: To find an entry with key k, **check the middle of the array**:
  - If you find an entry with key = k, great!
  - If you find an entry with key > k, focus on the **left half**!
  - If you find an entry with key < k, focus on the **right half**!



BINARY SEARCH



- What is the complexity of this algorithm?

O(log n)

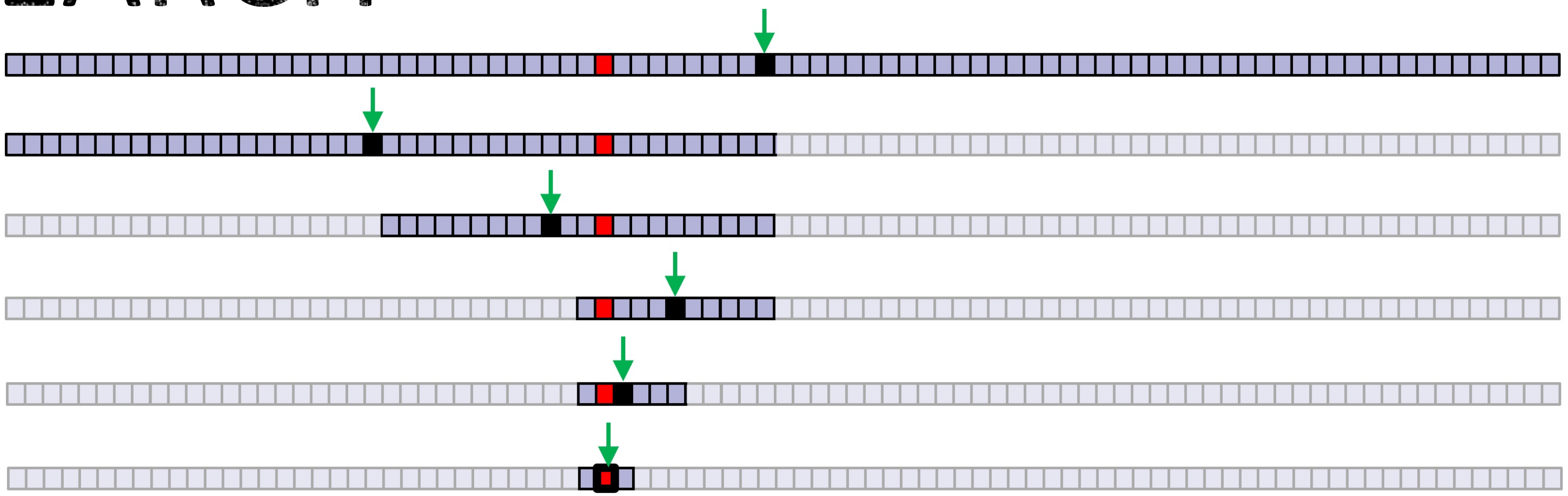
Example:

log(1024) = **10**, and to searching through **1024** entries we need at most **10** comparisons, because:

- After the **1**st comparison, you're left with **512** entries to search
- After the **2**nd comparison, you're left with **256**
- After the **3**rd comparison, you're left with **128**
- After the **4**th comparison, you're left with **64**
- After the **5**th comparison, you're left with **32**
- After the **6**th comparison, you're left with **16**
- After the **7**th comparison, you're left with **8**
- After the **8**th comparison, you're left with **4**
- After the **9**th comparison, you're left with **2**
- After the **10**th comparison, you've found it!

This pattern is the signature of an O(log n) time algorithm!

BINARY SEARCH



- What is the complexity of this algorithm?

O(log n)

In other words:

Grows linearly

Grows exponentially

- With 10 comparisons, you can search 1,024 entries
- With 11 comparisons, you can search 2,048 entries
- With 12 comparisons, you can search 4,096 entries
- With 13 comparisons, you can search 8,192 entries
- With 14 comparisons, you can search 16,384 entries
- With 15 comparisons, you can search 32,768 entries
- With 16 comparisons, you can search 65,536 entries
- With 17 comparisons, you can search 131,072 entries
- With 18 comparisons, you can search 262,144 entries
- With 19 comparisons, you can search 524,288 entries

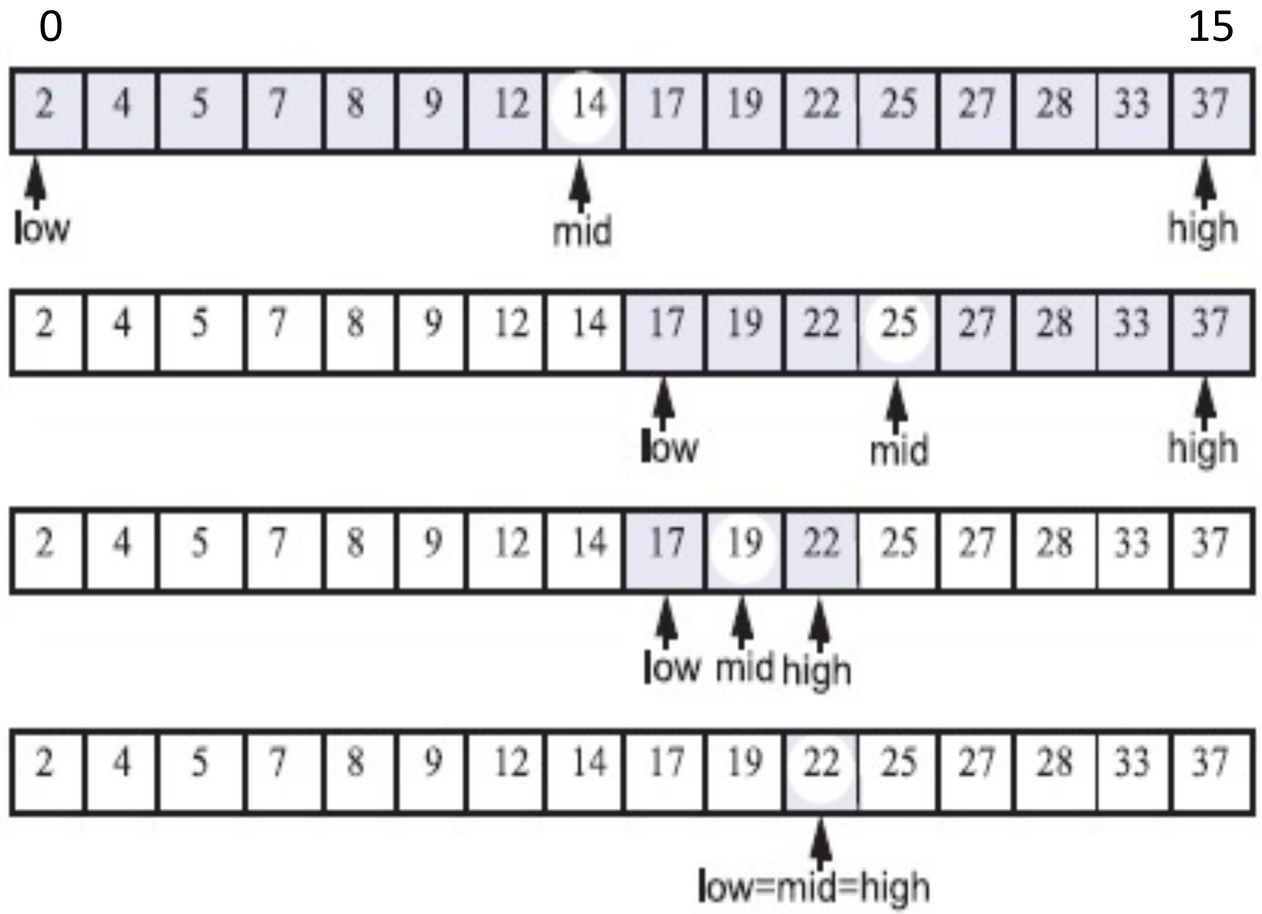
This pattern is the signature of an O(log n) time algorithm!

When n grows exponentially, runtime grows linearly, implying logarithmic runtime!

BINARY SEARCH: PSEUDO CODE

To **search** for an entry with key = k, call **BinarySearch(*L*, *k*, *0*, *n-1*)**, which would run **recursively**

Algorithm BinarySearch(*L*, *k*, *low*, *high*):
  Input: An ordered vector *L* storing *n* entries
    and integers *low* and *high*
  Output: An entry of *L* with key equal to *k* and
    index between *low* and *high*, if such
    an entry exists, and otherwise the
    special sentinel end
  if *low* > *high* **then**
    return end
  else
    $mid \leftarrow \lfloor (low+high)/2 \rfloor$ $e \leftarrow L.\mathrm{at}(mid)$
    if $k = e$.key() **then**
      return *e*
    else if $k < e$.key() **then**
      return BinarySearch(*L*, *k*, *low*, *mid*−1)
    else
      return BinarySearch(*L*, *k*, *mid*+1, *high*)



BINARY SEARCH: PSEUDO CODE

To **search** for an entry with key = k, call **BinarySearch(*L, k, 0, n-1*)**, which would run **recursively**

Algorithm BinarySearch(*L*, *k*, *low*, *high*):
    Input: An ordered vector *L* storing *n* entries and integers *low* and *high*
    Output: An entry of *L* with key equal to *k* and index between *low* and *high*, if such an entry exists, and otherwise the special sentinel end
    if *low* > *high* **then**
        return end
    else
        mid ← ⌊(*low*+*high*)/2⌋   *e* ← *L*.at(*mid*)
        if *k* = *e*.key() **then**
            return *e*
        else if *k* < *e*.key() **then**
            return BinarySearch(*L*, *k*, *low*, *mid*−1)
        else
            return BinarySearch(*L*, *k*, *mid*+1, *high*)

Complexity analysis:

- The runtime is proportional to number of recursive calls.
- **The number of remaining candidates is reduced by half with each recursive call**
  - Initially, the number of candidate entries is n;
  - after the first call, it is at most $n/2$;
  - after the second call, it is at most $n/4$;
  - after the third call, it is at most $n/8$;
  - . . .
  - after the i-th call, it is at most $n/2^i$.
- Hence, the time complexity is **O(log n)**

SEARCH TABLE VS. HASH TABLE

Method	*Hash Table*	*Search Table*
size, empty	$O(1)$	$O(1)$
find	$O(1)$ exp., $O(n)$ worst-case	$O(\log n)$
insert	$O(1)$	$O(n)$
erase	$O(1)$ exp., $O(n)$ worst-case	$O(n)$

- Given a hash function that minimizes collision, finding or erasing an entry from a **hash table** takes:
  - O(1) expected time (since we expect to have a single entry per bucket on average)
  - O(n) time (because in the worst case all elements will end up in the same bucket)
- Given an **ordered search table** :
  - Finding an entry whose key = k takes O(log n) time (since it uses binary search)
  - Erasing an entry takes O(n) time (since we must shift other entries around)

42

SKIP LISTS

SKIP LISTS

- As mentioned earlier, when implementing an ordered map using a **search table** :
  - Finding an entry whose key = k takes O(log n) time (since it uses binary search)
  - Inserting or erasing an entry takes O(n) time (since we must shift other entries)
- However, if we implement an ordered map using a “**skip list**”, then:
  - Finding an entry whose key = k takes O(log n) time **on average**
  - Inserting or erasing an entry takes O(log n) time **on average**
- Here, the notion of **average time complexity** depends on the use of a **random-number generator** in the implementation of the insertions, to help decide where to place the new entry.
- The running time is averaged over **all possible outcomes** of the random numbers used when inserting entries.