# MODULE-5

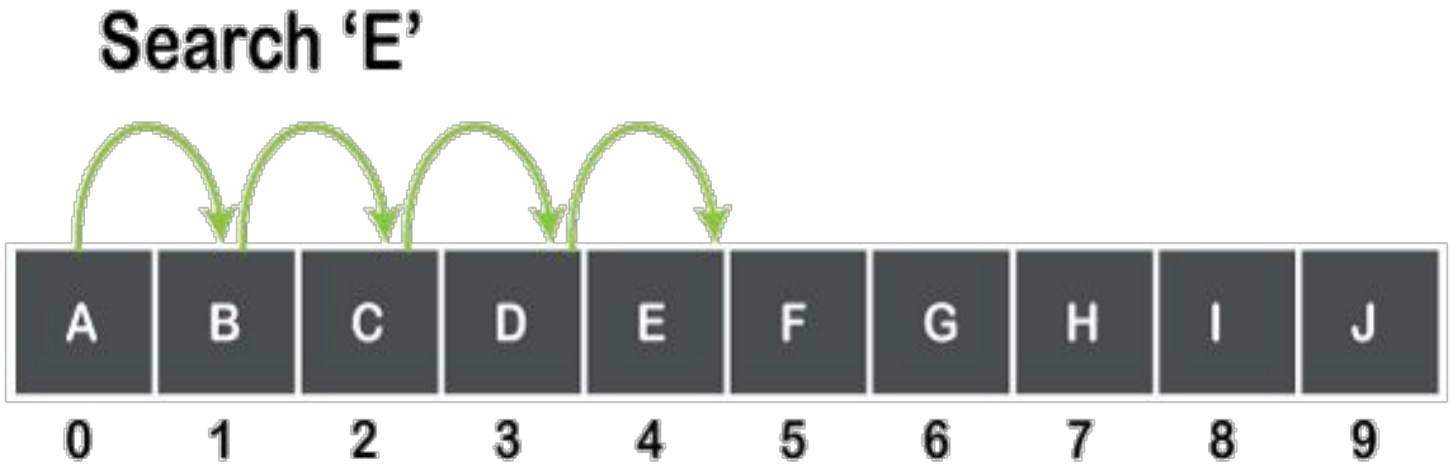
# Searching

# Lecture Objectives:

- Learn how to implement the sequential search algorithm
- Learn how to implement the binary search algorithm
- To learn how to estimate and compare the performance of algorithms
- To learn how to measure the running time of a program.

# What is a linear search?

**A linear search is also known as a sequential search that simply scans each element at a time.**



- Necessary components to search a list of fdata
  - Array containing the list
  - Length of the list
  - Item for which you are searching

After search completed

  - If item found, report “success,” return location in array
  - If item not found, report “not found” or “failure”

- Suppose that you want to determine whether 27 is in the list or not.

- First compare 27 with list[0]; that is, compare 27 with 35
- Because list[0] ≠ 27, you then compare 27 with list[1]
- Because list[1] ≠ 27, you compare 27 with the next element in the list
- Because list[2] = 27, the search stop

- This search is successful!

- List

| [1] | [2] | [3] | [4] | [5] | [6] | [7] | |
|---|---|---|---|---|---|---|---|
| 35 | 12 | 27 | 18 | 45 | 16 | 38 | ..... |

- Let’s now search for 10
- The search starts at the first element in the list; that is, at list[0]
- Proceeding as before, we see that this time the search item, which is 10, is compared with every item in the list
- Eventually, no more data is left in the list to compare with the search item; this is an unsuccessful search

# Complexity of Linear search

**As linear search scans each element one by one until the element is not found. If the number of elements increases, the number of elements to be scanned is also increased.**

**We can say that the *time taken to search the elements is proportional to the number of elements*.**

**Therefore, the worst-case complexity is O(n).**

# Complexity of Linear search

| Case | Time Complexity |
|---|---|
| **Best Case** | O(1) |
| **Average Case** | O(n) |
| **Worst Case** | O(n) |

**Best Case Complexity - In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is O(1).**
**Average Case Complexity - The average case time complexity of linear search is O(n).**
**Worst Case Complexity - In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array.**

- Problem: Given a sorted array of integers and an integer *i*, find the index of any occurrence of *i* if it appears in the array. If not, return -1.
  - We could solve this problem using a standard iterative search; starting at the beginning, and looking at each element until we find *i*
  - What is the runtime of an iterative search?

- Since the array is sorted, we can do better.

# BINARY SEARCH ALGORITHM

- Algorithm idea: Start in the middle, and only search the portions of the array that might contain the element i.  Eliminate half of the array from consideration at each step.
  - Can be written iteratively, but is harder to get right

- Called binary search because it chops the area to examine in half each time
  - Implemented in Java as Arrays.binarySearch in java.util package

# DIVIDE-AND-CONQUER

- **divide-and-conquer** algorithm: a means for solving a problem that first separates the main problem into 1 or more smaller problems, then solves each of the smaller problems, then uses those sub-solutions to solve the original problem
  - 1: "divide" the problem up into pieces
  - 2: "conquer" each smaller piece
  - 3: (if necessary) combine the pieces at the end to produce the overall solution

  - binary search is one such algorithm

# BINARY SEARCH PSEUDOCODE

binary search array *a* for value *i*:
    if all elements have been searched,
        result is -1.
    examine middle element *a*[*mid*].
    if a[*mid*] equals *i*,
        result is *mid*.
    if *a*[*mid*] is greater than *i*,
        binary search left half of *a* for *i*.
    if *a*[*mid*] is less than *i*,
        binary search right half of *a* for *i*.

# BINARY SEARCH REQUIREMENTS

- The one *pre-requisite of binary search* is that an array should be in sorted order, whereas the linear search works on both sorted and unsorted array.
- The binary search algorithm is based on the divide and conquer technique, which means that it will divide the array recursively.

**There are three cases used in the binary search:**

**Case 1: data<a[mid] then right=mid-1.**
**Case 2: data>a[mid] then left = mid+1.**
**Case 3: data = a[mid] // element is found**

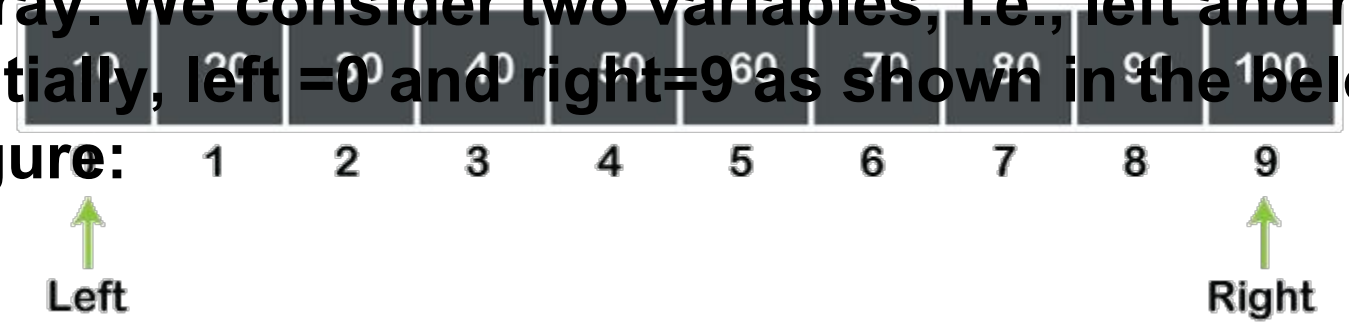
**In the above case, 'a' is the name of the array, mid is the index of the element calculated recursively, data is the element that is to be searched, left denotes the left element of the array and right denotes the element that**

# BINARY SEARCH EXAMPLE

**Suppose we have an array of 10 size which is indexed from 0 to 9 as shown in the below figure:**

**We want to search for 70 element from the above array.**

**Step 1: First, we calculate the middle element of an array. We consider two variables, i.e., left and right. Initially, left =0 and right=9 as shown in the below figure:**

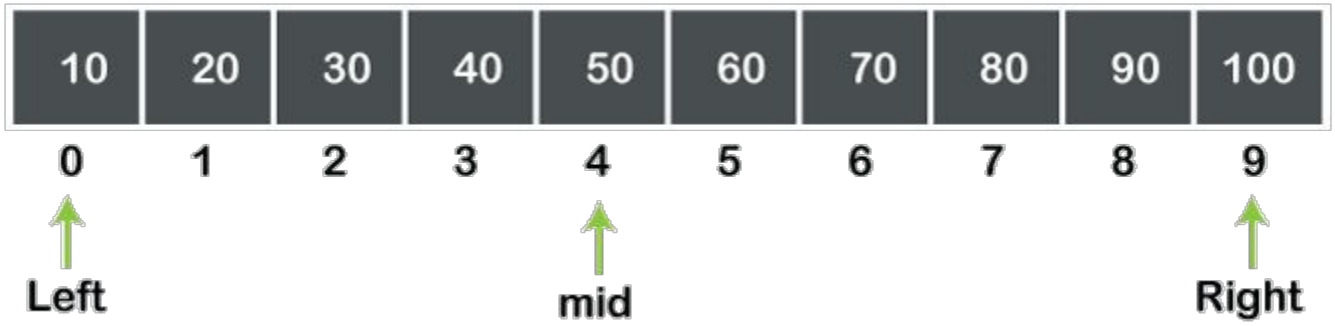


# BINARY SEARCH EXAMPLE

**The middle element value can be calculated as:**

$$mid = \frac{left + right}{2}$$

**Therefore, mid = 4 and a[mid] = 50. The element to be searched is 70, so a[mid] is not equal to data. The case 2 is satisfied, i.e., data>a[mid].**

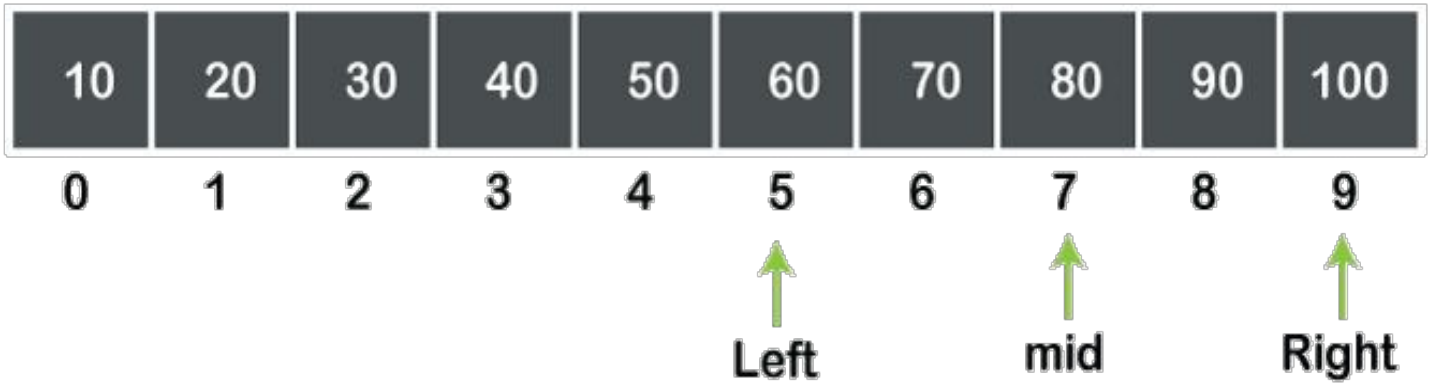


# BINARY SEARCH EXAMPLE

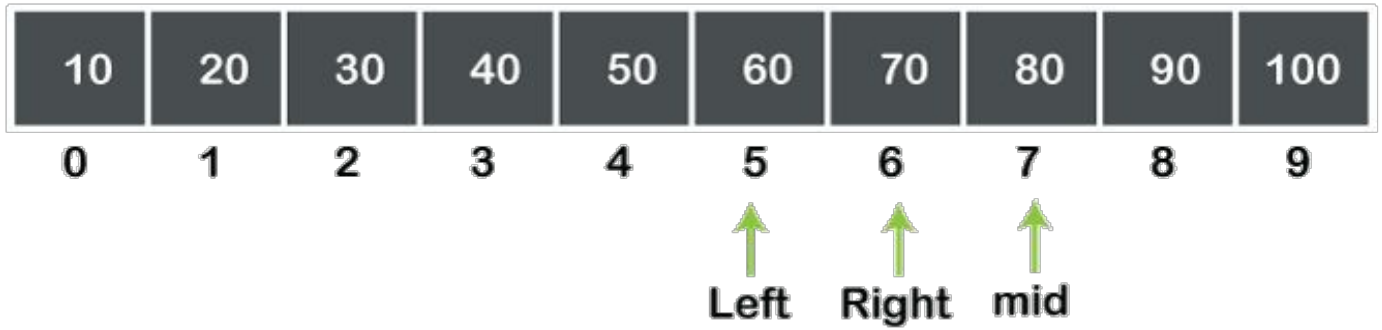
**Step 2: As data>a[mid], so the value of left is incremented by mid+1, i.e., left=mid+1.**

**The value of mid is 4, so the value of left becomes 5. Now, we have got a subarray as shown in the below figure:**

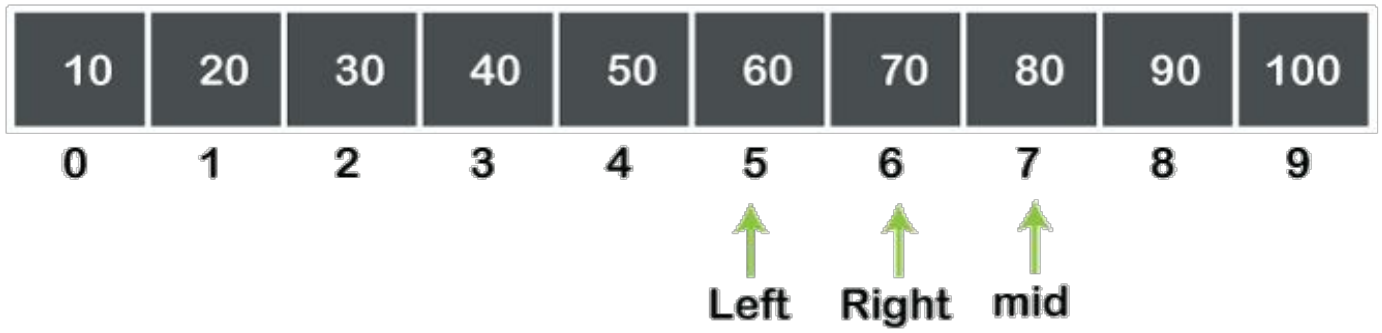
# BINARY SEARCH EXAMPLE

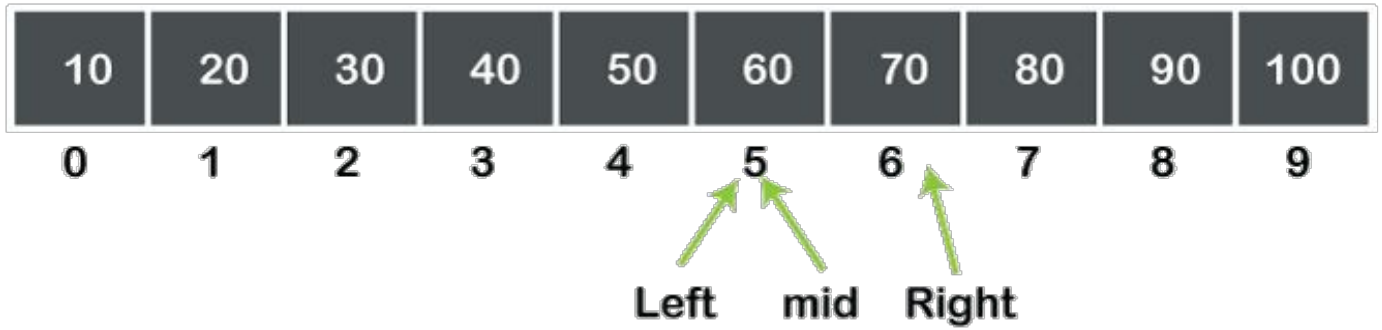


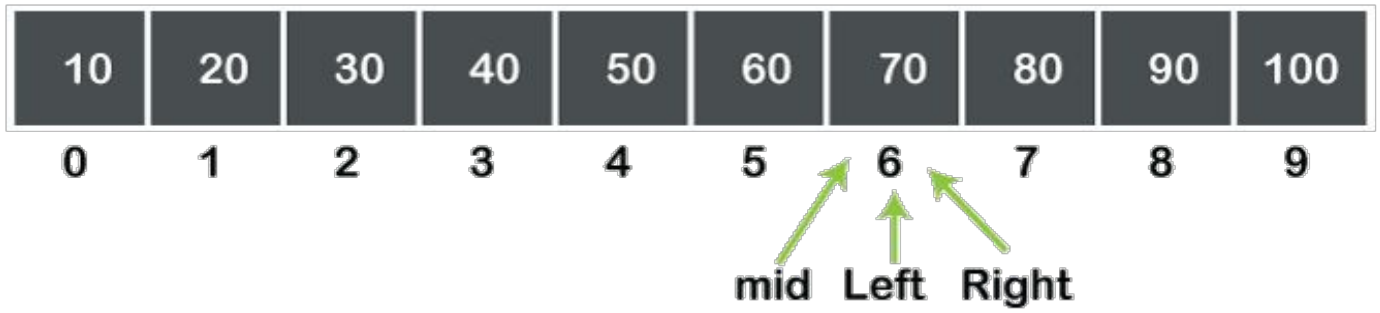

**As a[mid]>data, the value of right is decremented by mid-1. The value of mid is 7, so the value of right becomes 6. The array can be represented as:**



# BINARY SEARCH EXAMPLE

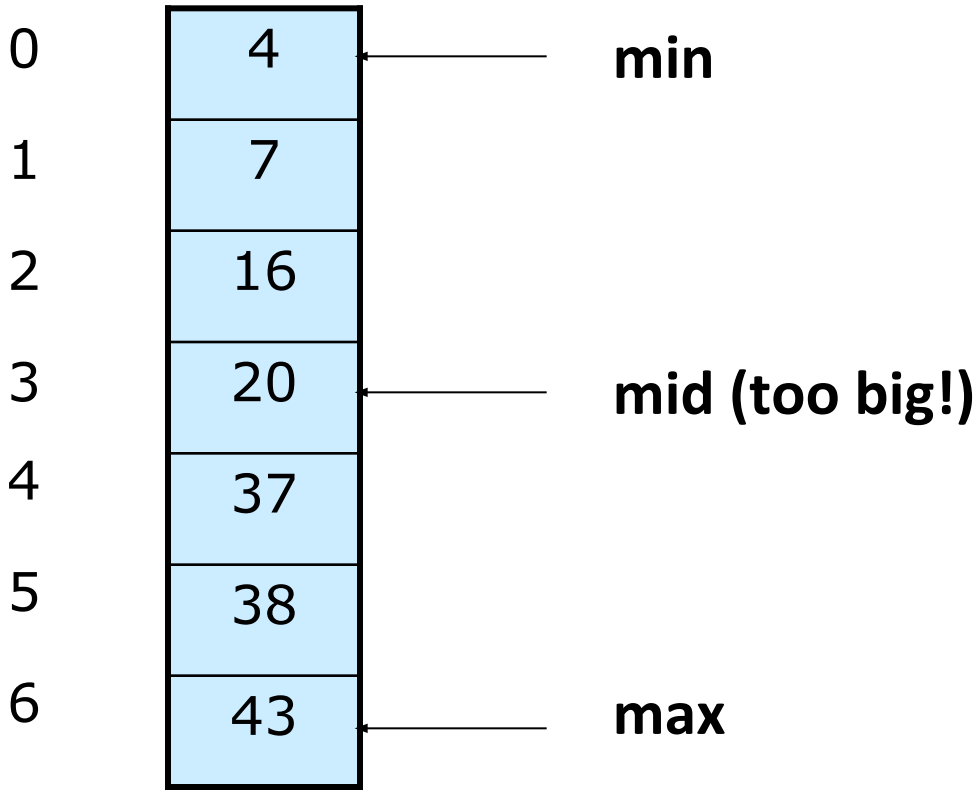


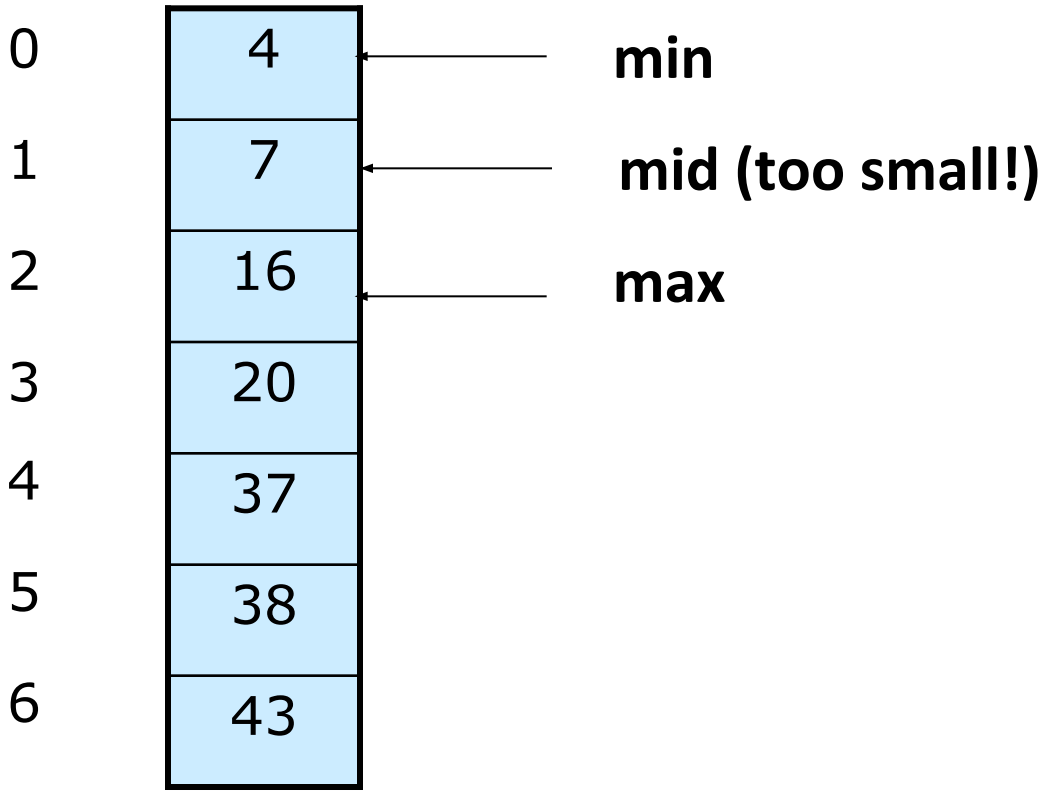
# BINARY SEARCH EXAMPLE

**i = 16**



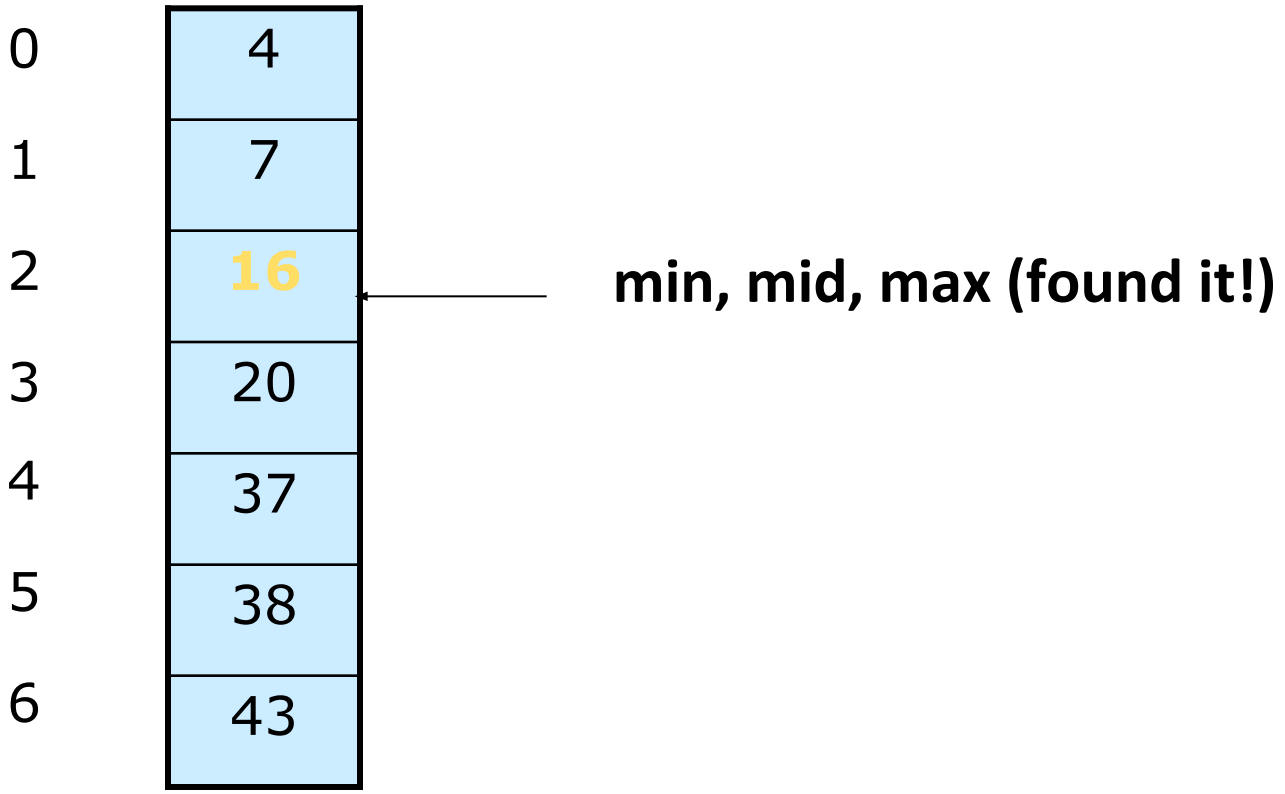
# BINARY SEARCH EXAMPLE

**i = 16**



# BINARY SEARCH EXAMPLE

**i = 16**



# SELECTION SORT

# SELECTION SORT: IDEA

1. We have two group of items:
   - sorted group, and
   - unsorted group
2. Initially, all items are in the unsorted group. The sorted group is empty.
   - We assume that items in the unsorted group unsorted.
   - We have to keep items in the sorted group sorted.

# SELECTION SORT: CONT'D

1. Select the "best" (eg. smallest) item from the unsorted group, then put the "best" item at the end of the sorted group.
2. Repeat the process until the unsorted group becomes empty.

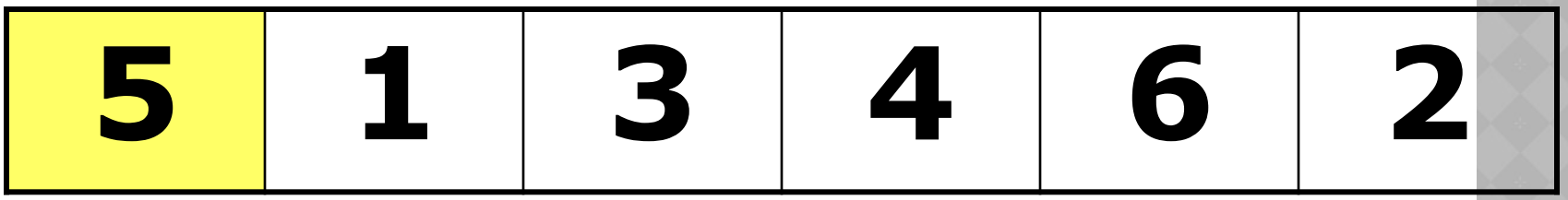
# SELECTION SORT

| 5 | 1 | 3 | 4 | 6 | 2 |
|---|---|---|---|---|---|

Comparison

Data Movement

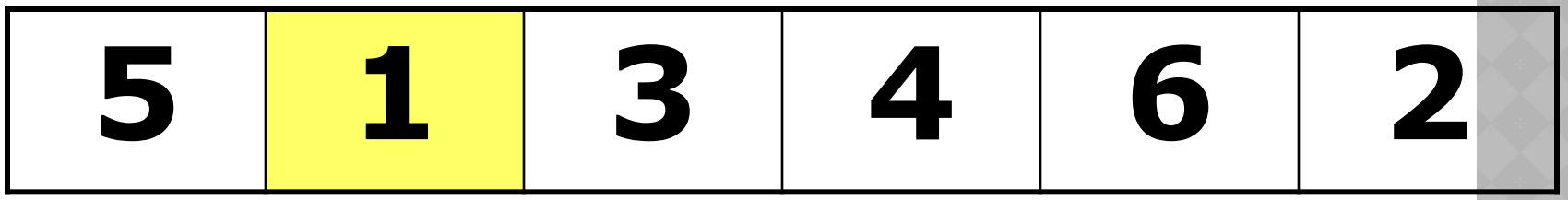
# SELECTION SORT




Compariso
n

Data
Movement

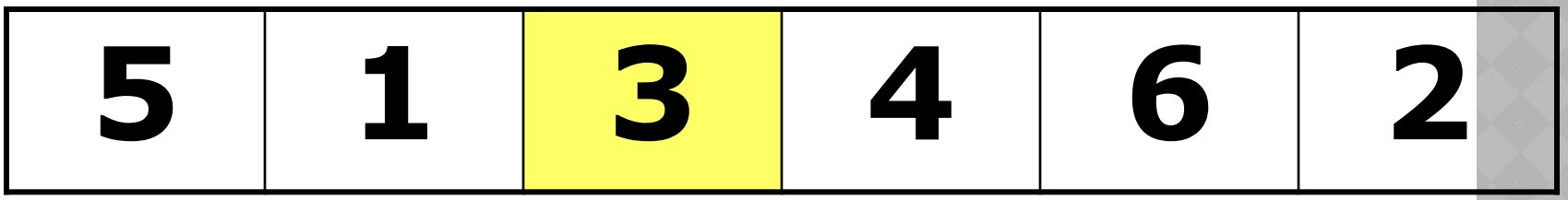
# SELECTION SORT




Comparison

Data Movement

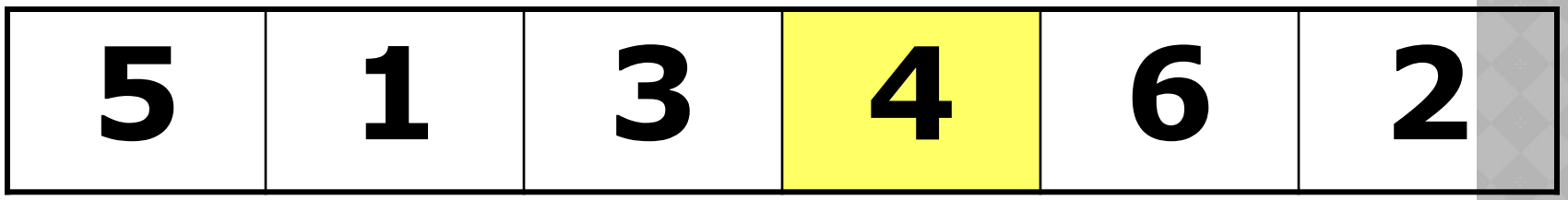
# SELECTION SORT




Compariso
n

Data
Movement

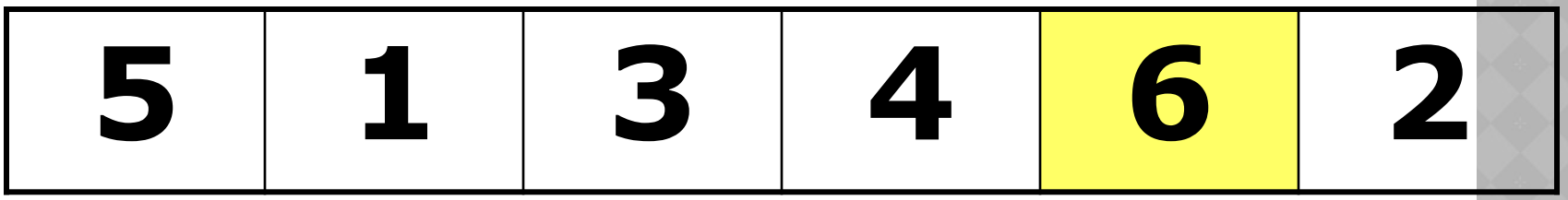
# SELECTION SORT




Comparison

Data Movement

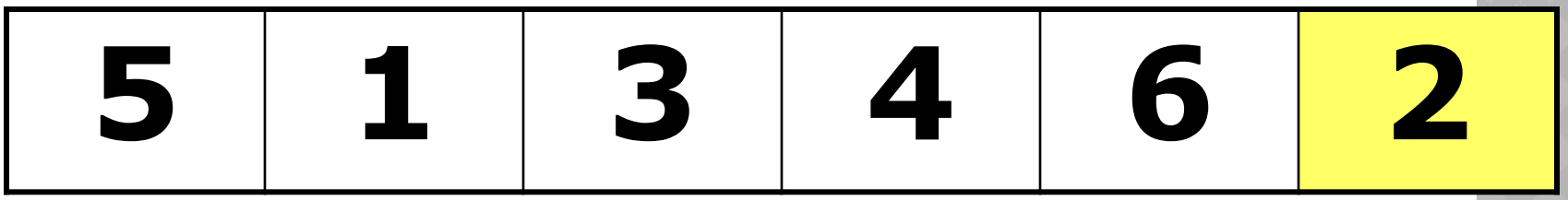
# SELECTION SORT




Compariso
n

Data
Movement

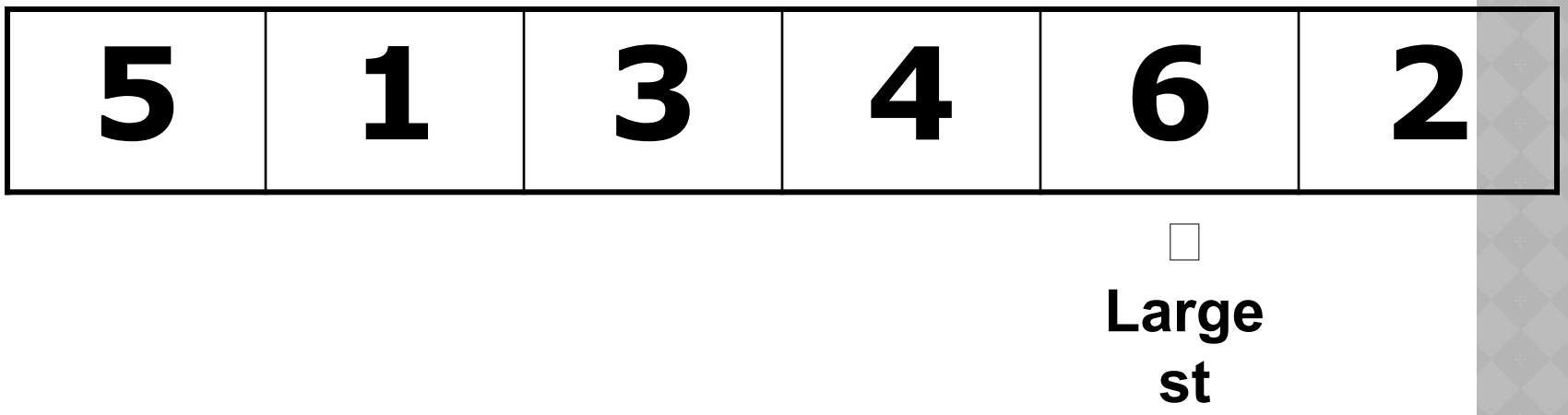
# SELECTION SORT



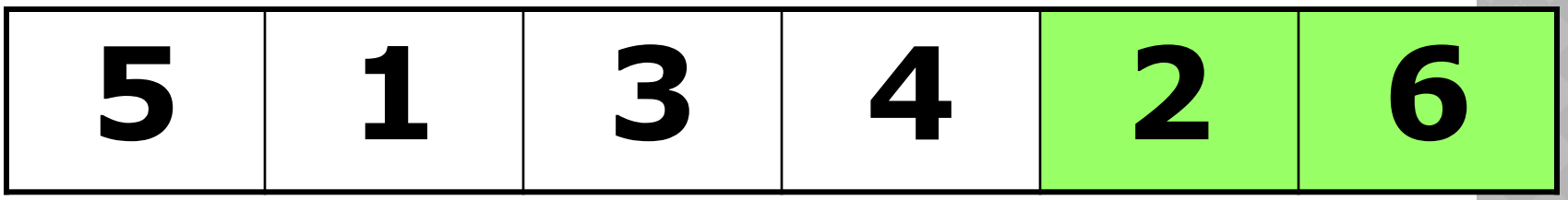

Compariso
n

Data
Movement

# SELECTION SORT



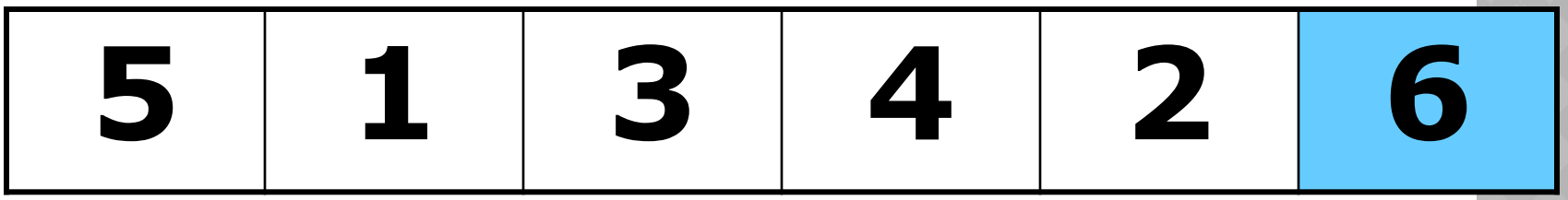
# SELECTION SORT




Comparison

Data Movement

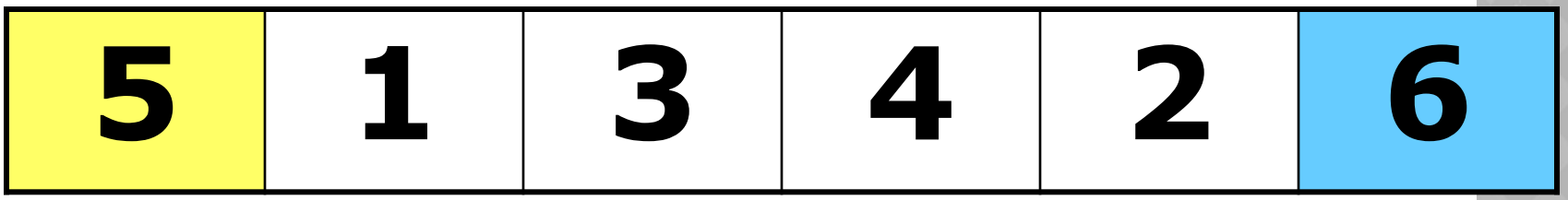
# SELECTION SORT




Comparison

Data Movement

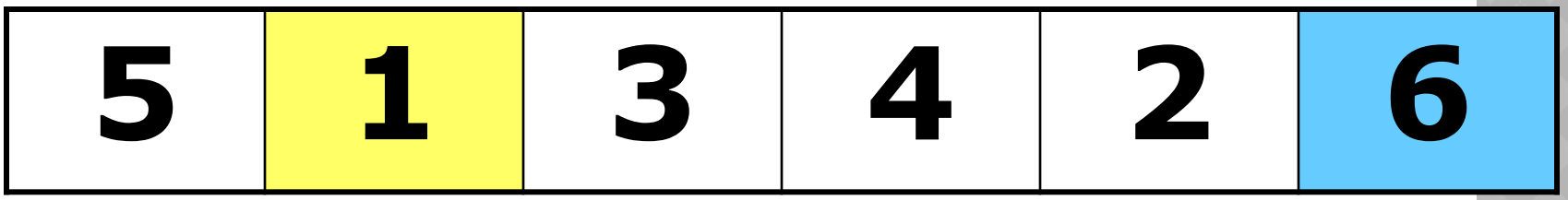
# SELECTION SORT




Compariso
n

Data
Movement

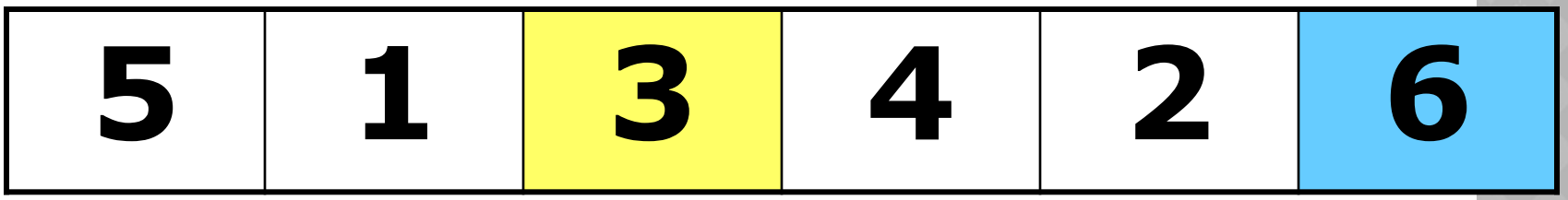
# SELECTION SORT




Compariso
n

Data
Movement

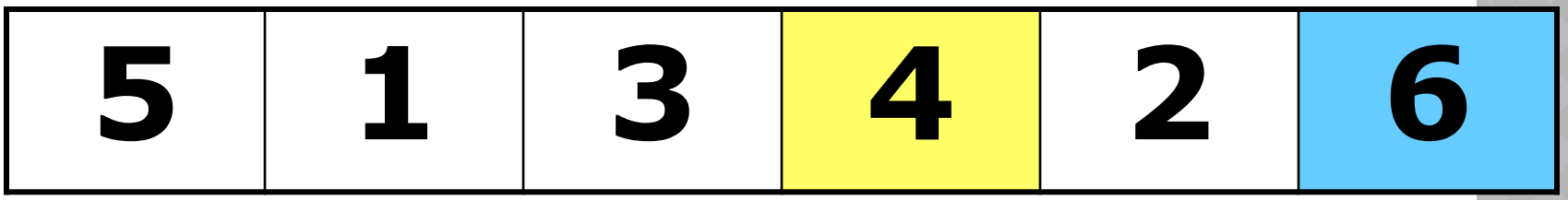
# SELECTION SORT




Compariso
n

Data
Movement

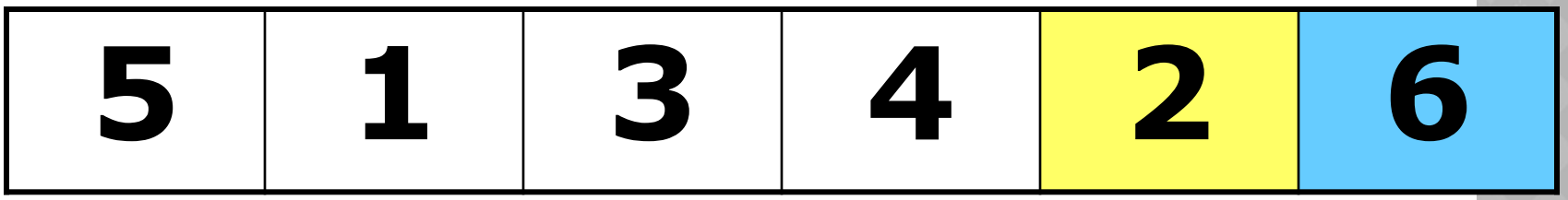
# SELECTION SORT




Comparison

Data Movement

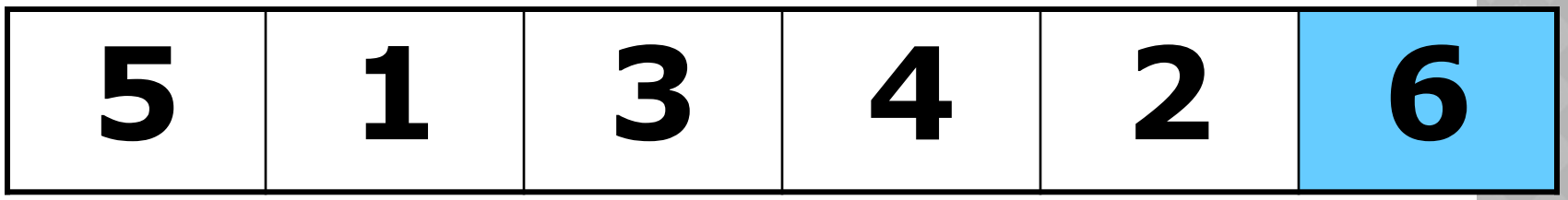
# SELECTION SORT




Compariso
n

Data
Movement

# SELECTION SORT

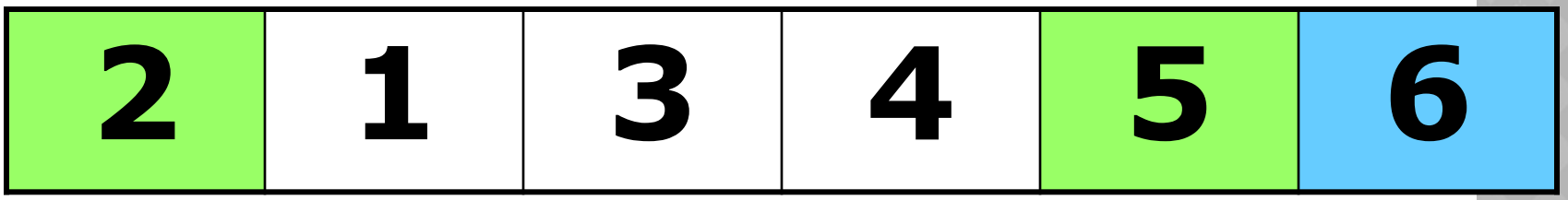



Largest

Comparison

Data Movement

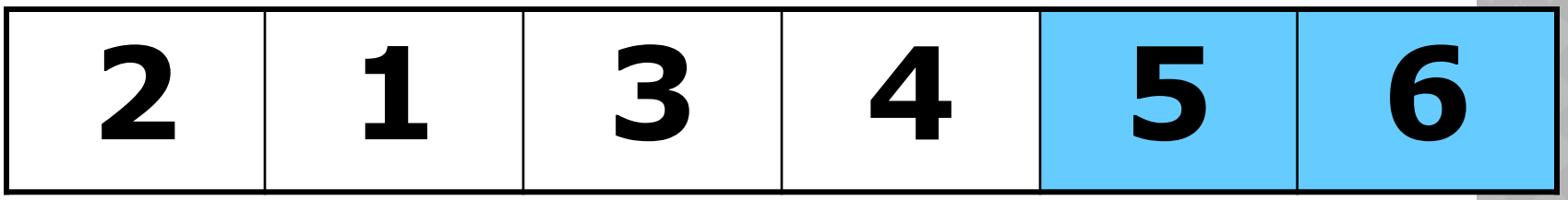
# SELECTION SORT




Compariso
n

Data
Movement

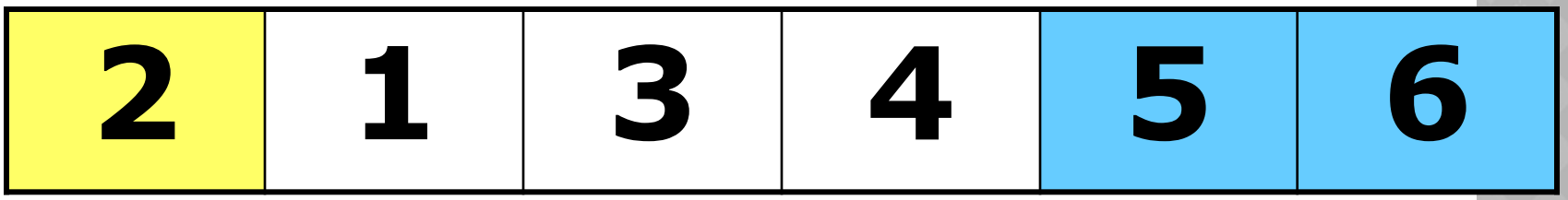
# SELECTION SORT




Comparison

Data Movement

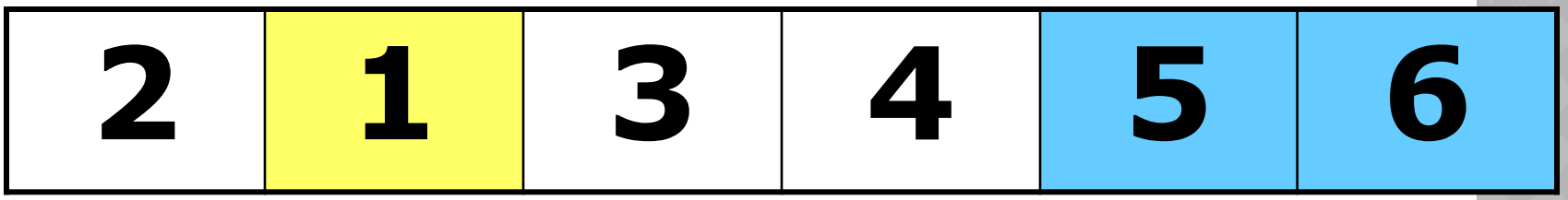
# SELECTION SORT




Compariso
n

Data
Movement

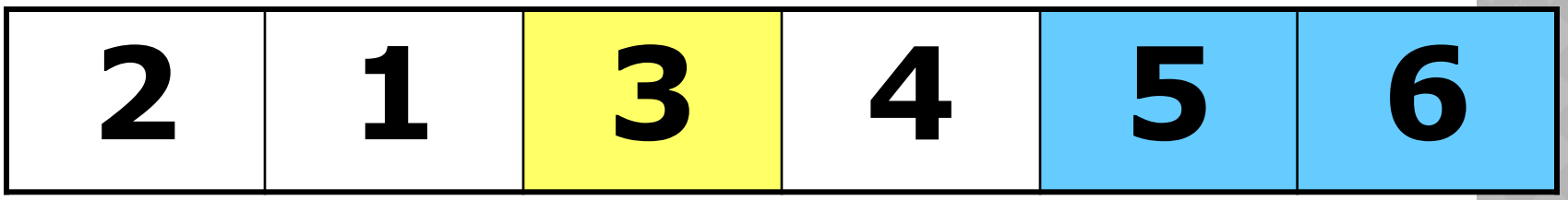
# SELECTION SORT




Comparison

Data Movement

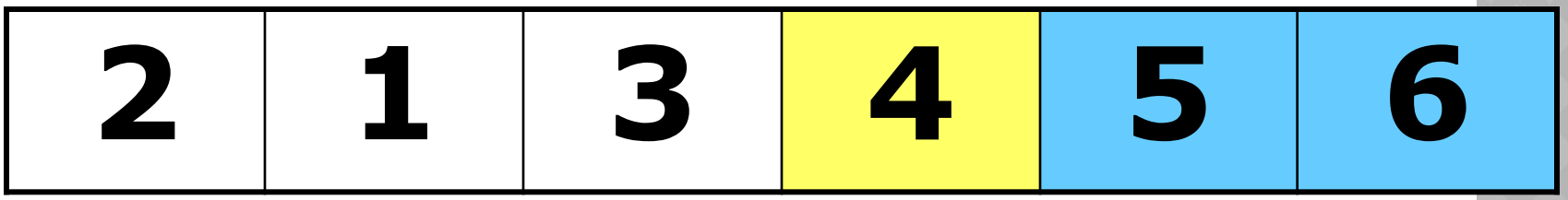
# SELECTION SORT




Compariso
n

Data
Movement

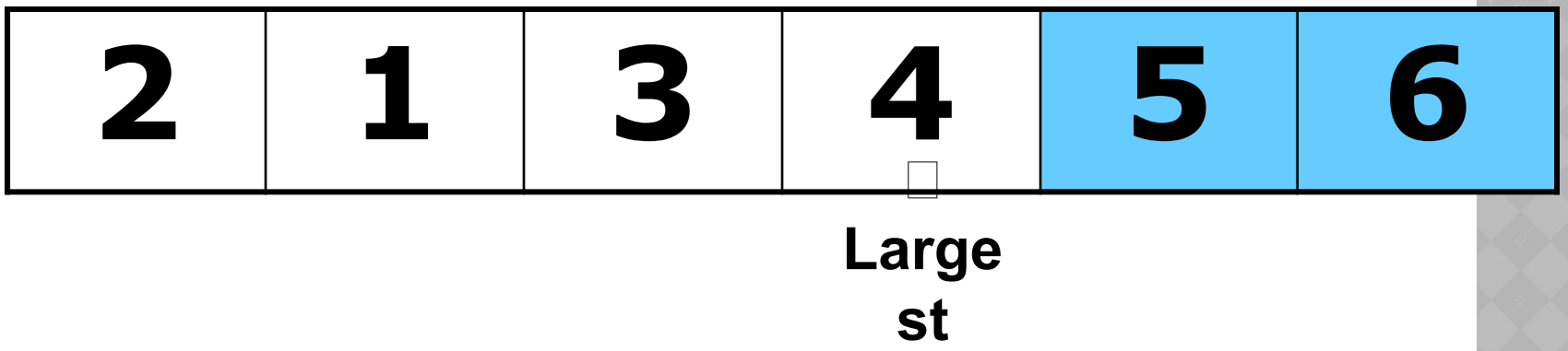
# SELECTION SORT




Compariso
n

Data
Movement

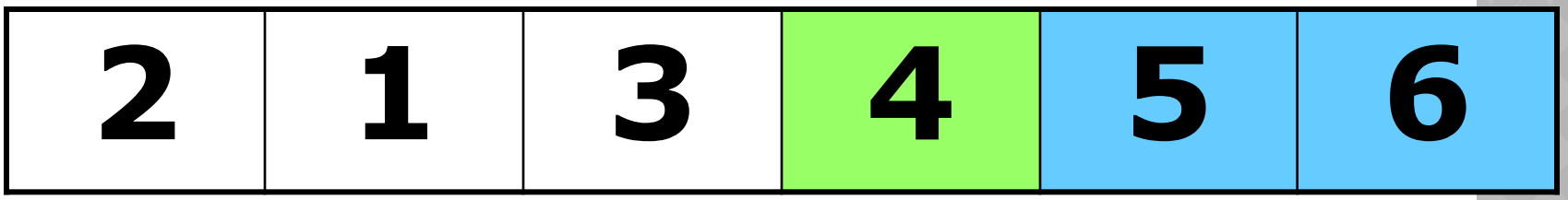
# SELECTION SORT




Compariso
n

Data
Movement

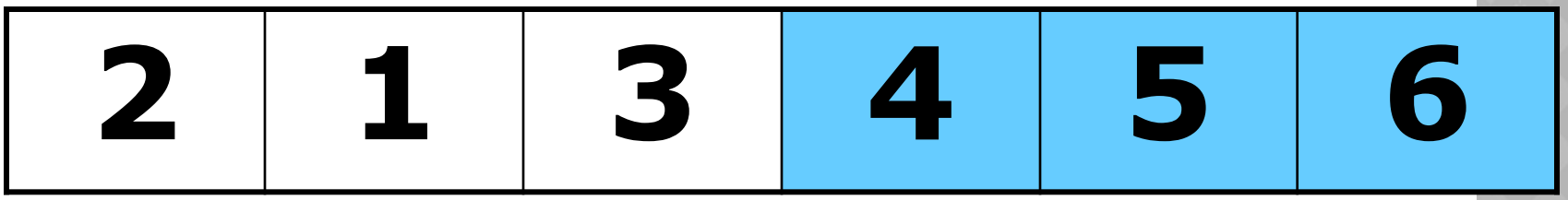
# SELECTION SORT




Comparison

Data Movement

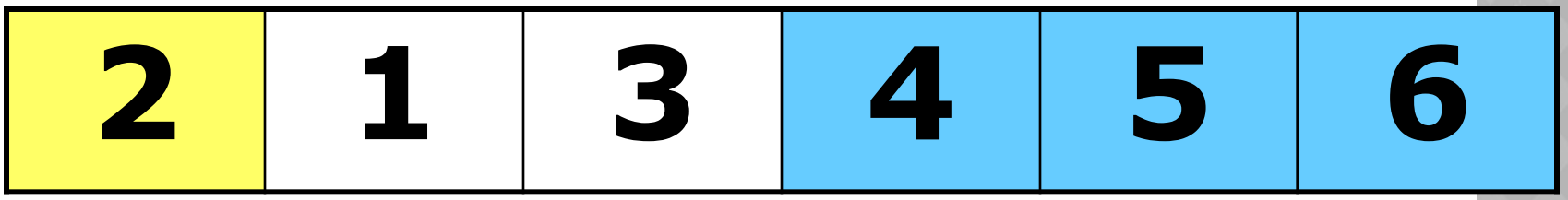
# SELECTION SORT




Comparison

Data Movement

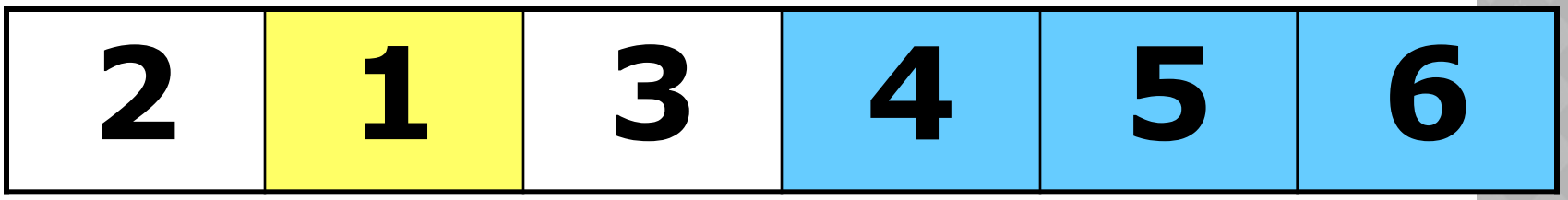
# SELECTION SORT




Comparison

Data Movement

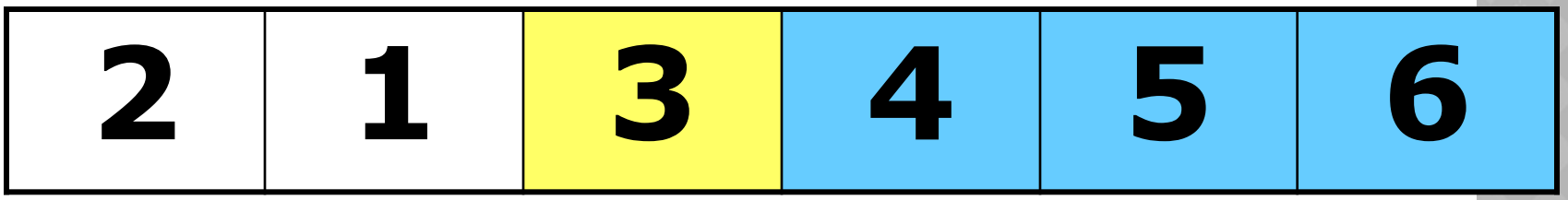
# SELECTION SORT




Comparison

Data Movement

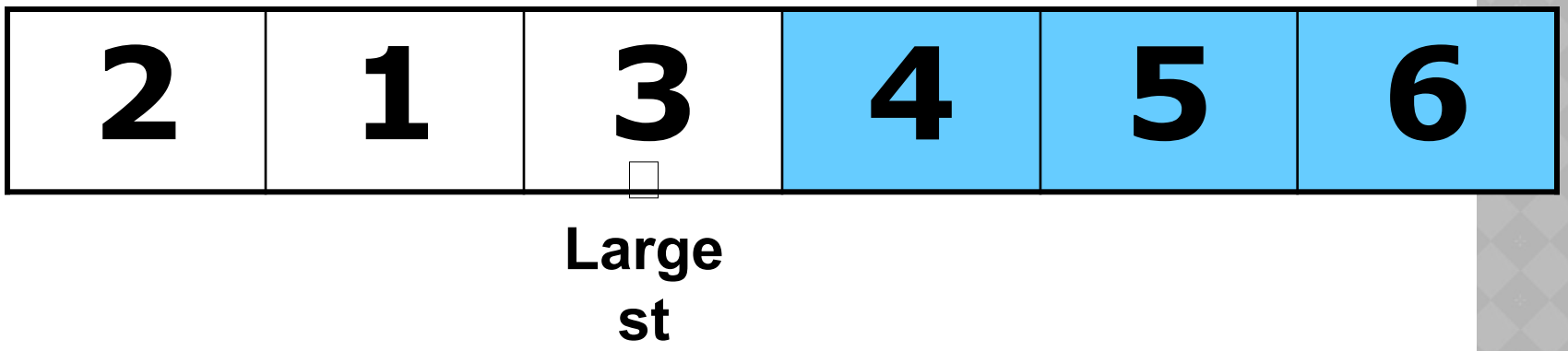
# SELECTION SORT




Compariso
n

Data
Movement

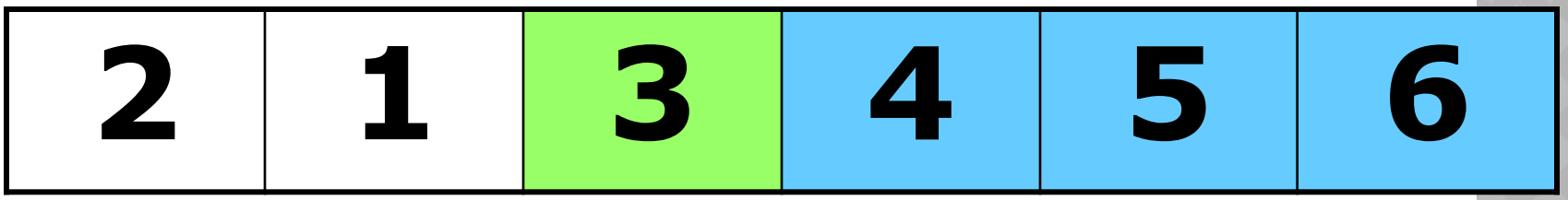
# SELECTION SORT




Comparison

Data Movement

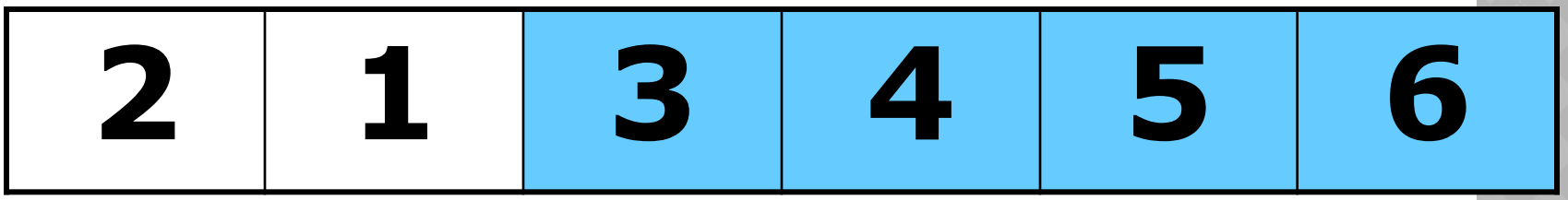
# SELECTION SORT




Comparison

Data Movement

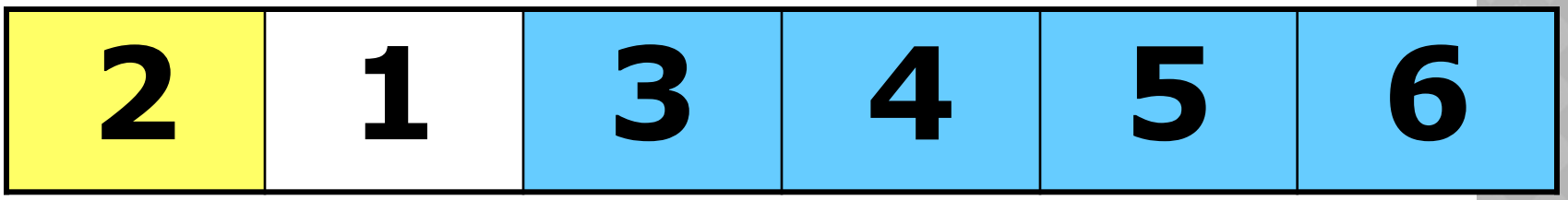
# SELECTION SORT




Comparison

Data Movement

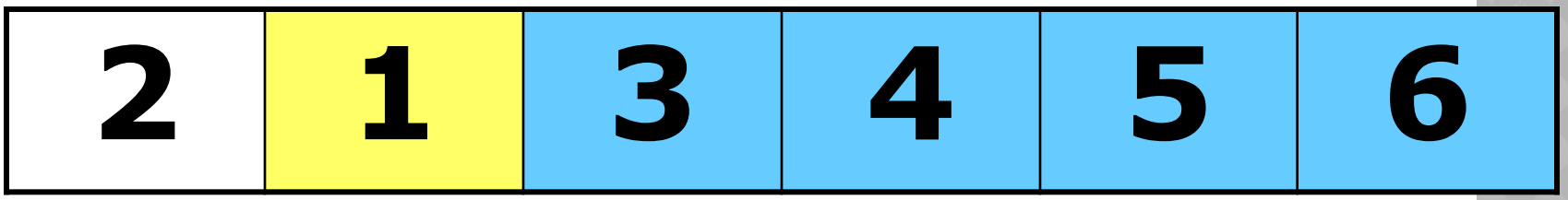
# SELECTION SORT




Comparison

Data Movement

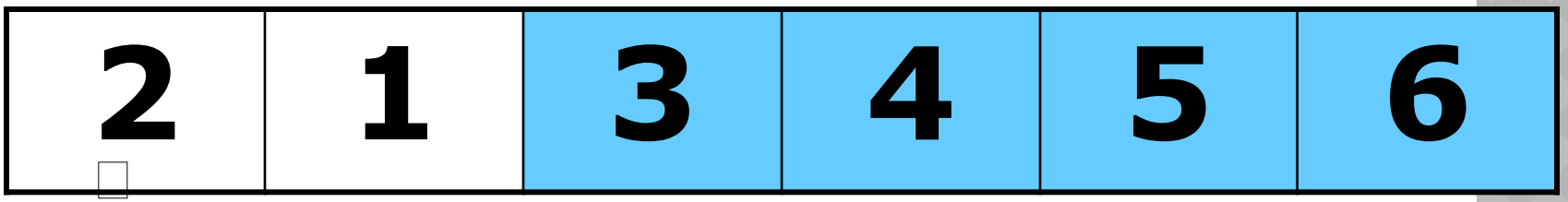
# SELECTION SORT



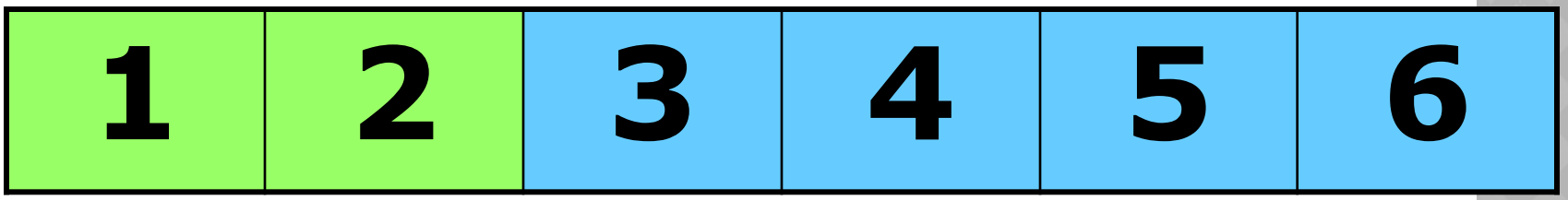

Comparison

Data Movement

# SELECTION SORT



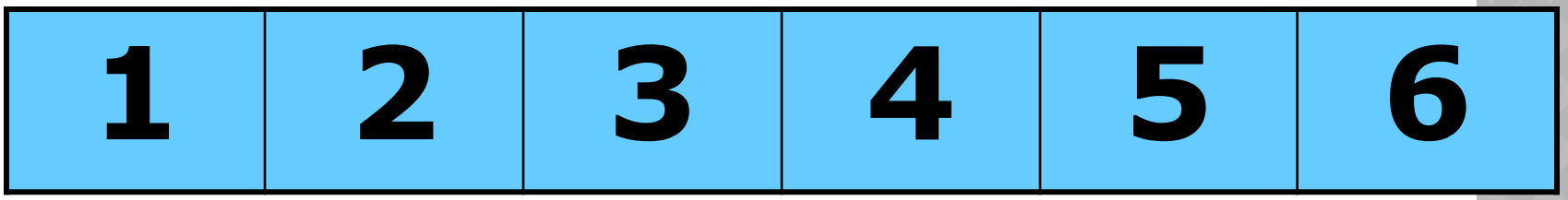
# SELECTION SORT




Compariso
n

Data
Movement

# SELECTION SORT

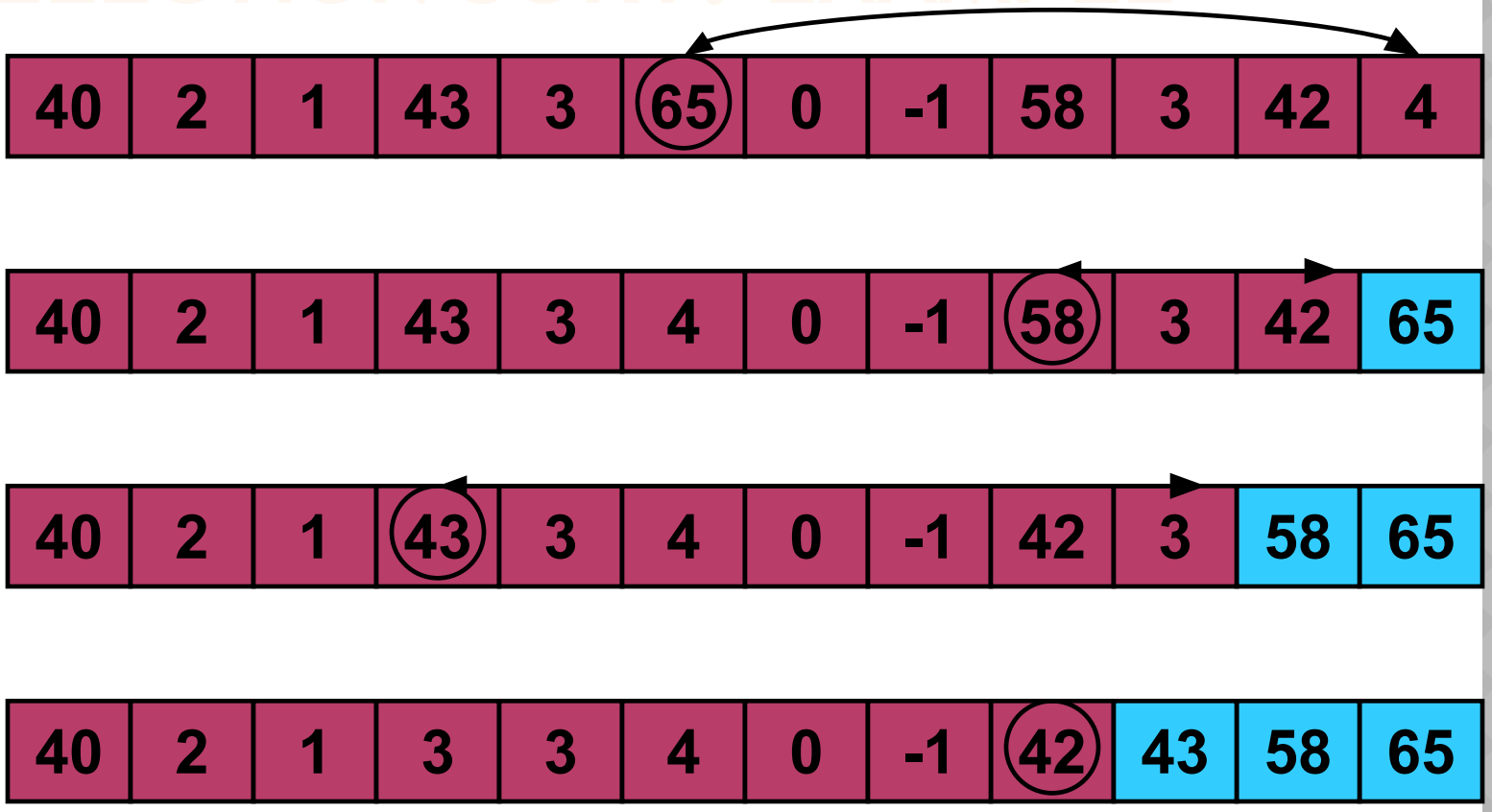



**DONE!**

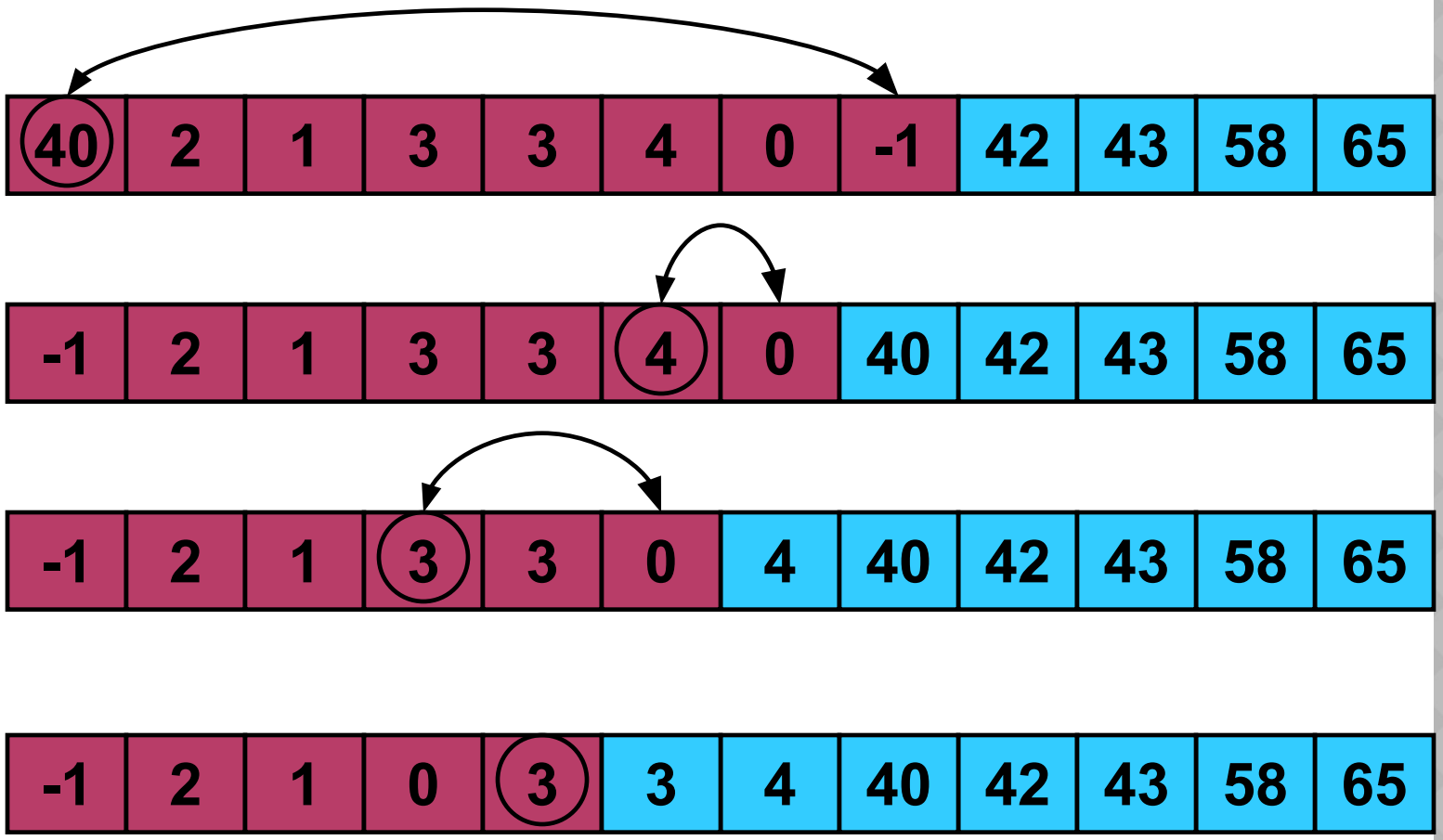
Comparison

Data Movement

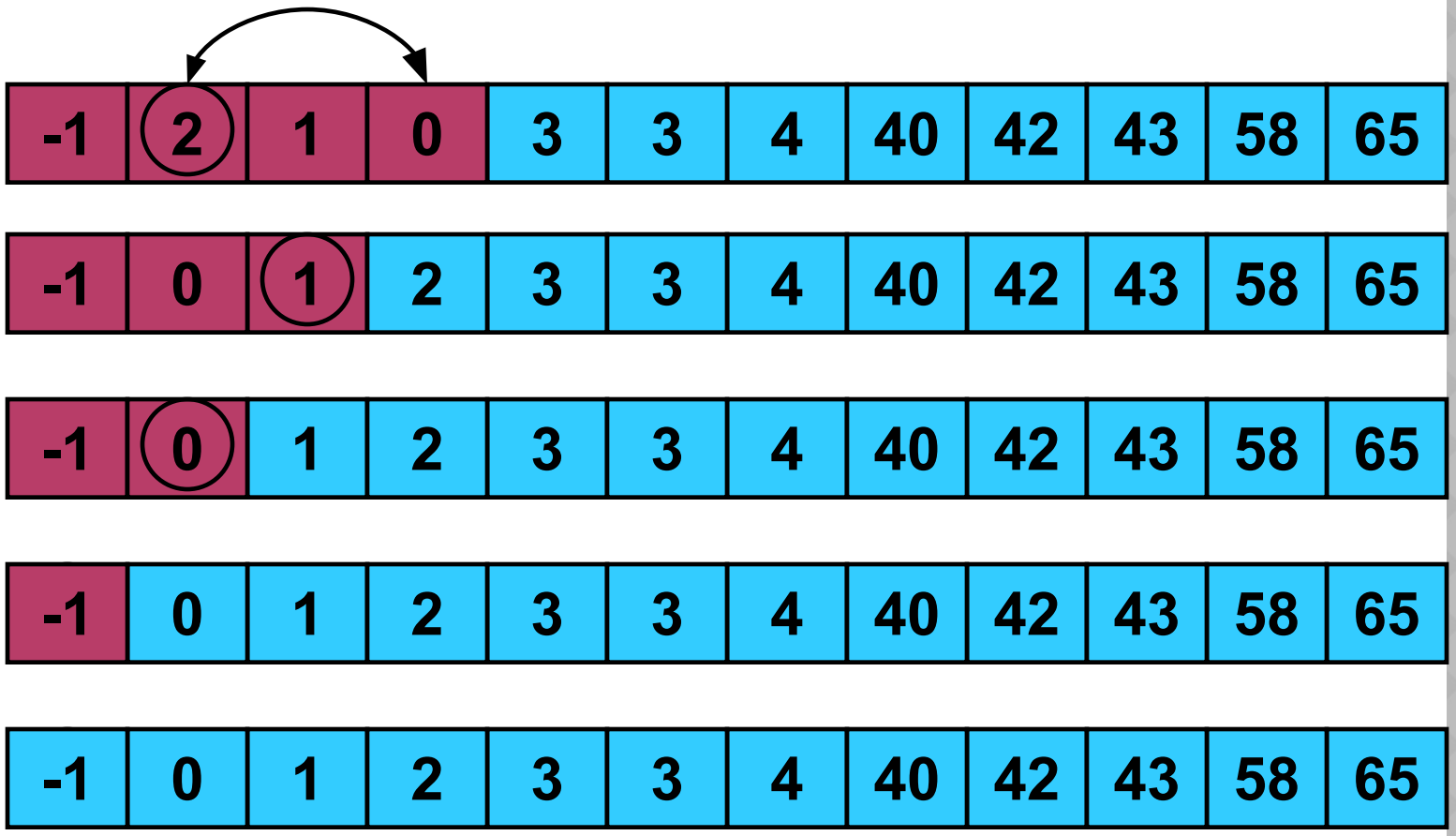
# SELECTION SORT: EXAMPLE



# SELECTION SORT: EXAMPLE



# SELECTION SORT: EXAMPLE



# SELECTION SORT: ANALYSIS

- Running time:
  - Worst case: $O(N^2)$
  - Best case: $O(N^2)$

# HEAP SORT

- Heap sort is one of the sorting algorithms used to arrange a list of elements in order.
- Heap sort algorithm uses one of the tree concepts called **Heap Tree**.
- In this sorting algorithm, we use **Max Heap** to arrange list of elements in Ascending order and **Min Heap** to arrange list elements in Descending order.

# STEP BY STEP PROCESS

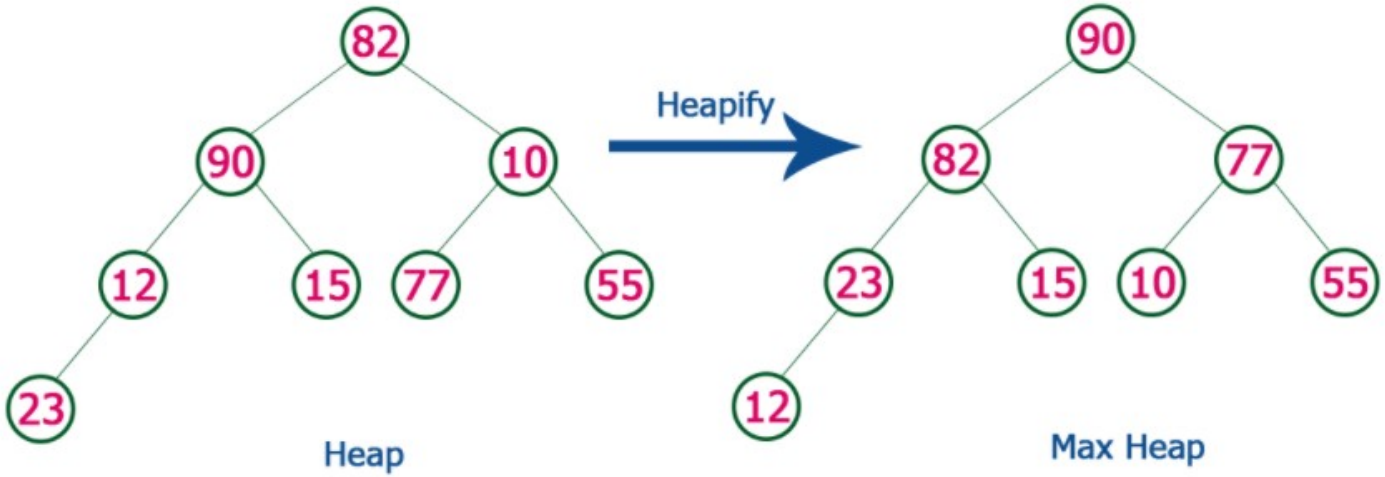
- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Max Heap.**
- **Step 3 -** Delete the root element from Max Heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Max Heap becomes empty.
- **Step 6 -** Display the sorted list

# EXAMPLE:

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap

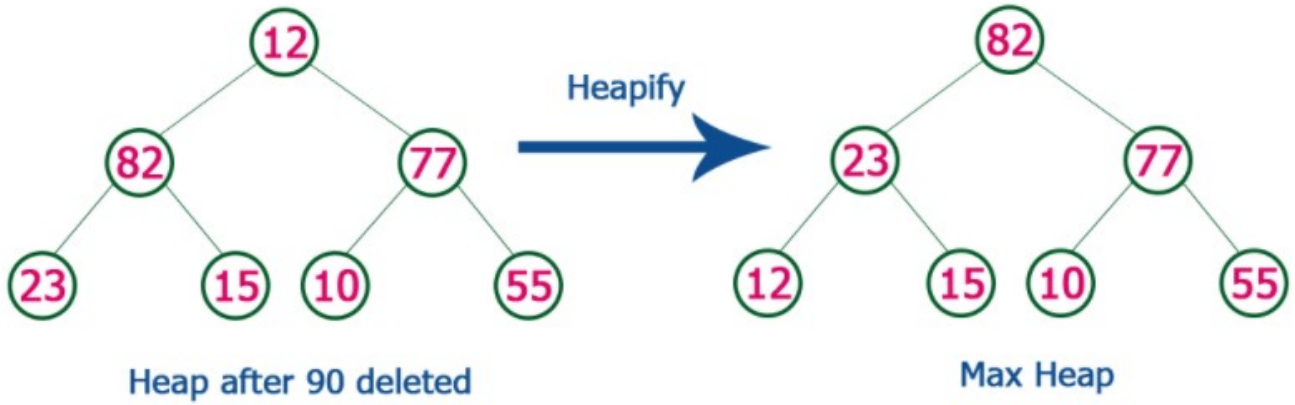



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

# EXAMPLE: CUNTD....

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

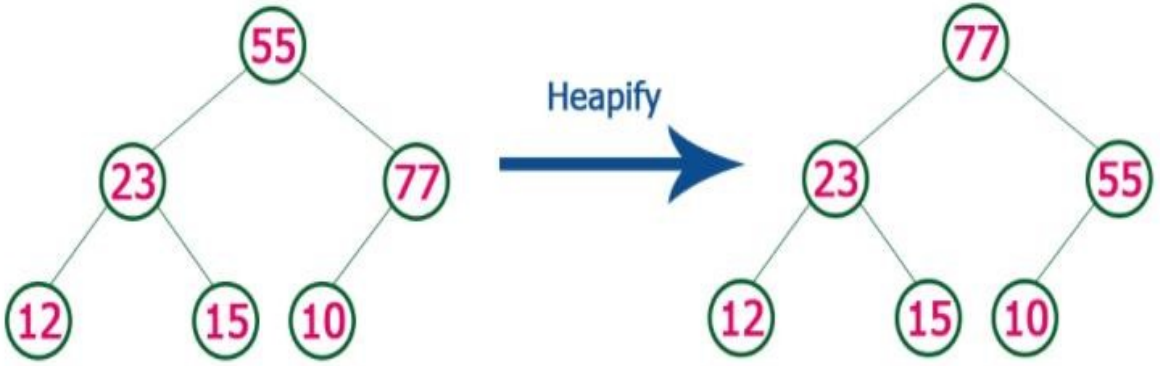



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

# EXAMPLE: CUNTD....

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.




Heap after 82 deleted

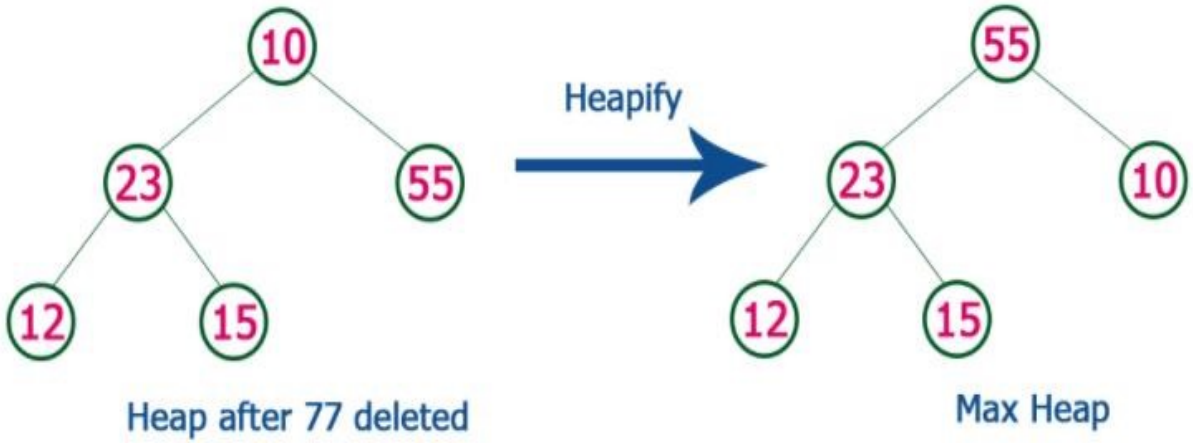
Max Heap

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82**, **90**

# EXAMPLE: CUNTD....

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

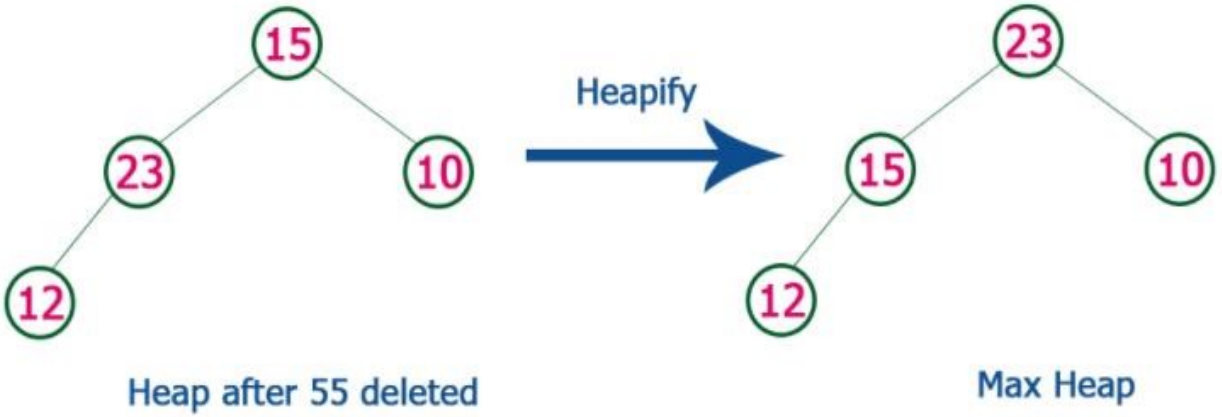



list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77**, **82**, **90**

# EXAMPLE: CUNTD....

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.




list of numbers after swapping 55 with 15.

12, 15, 10, 23, **55, 77, 82, 90**

# EXAMPLE: CUNTD....

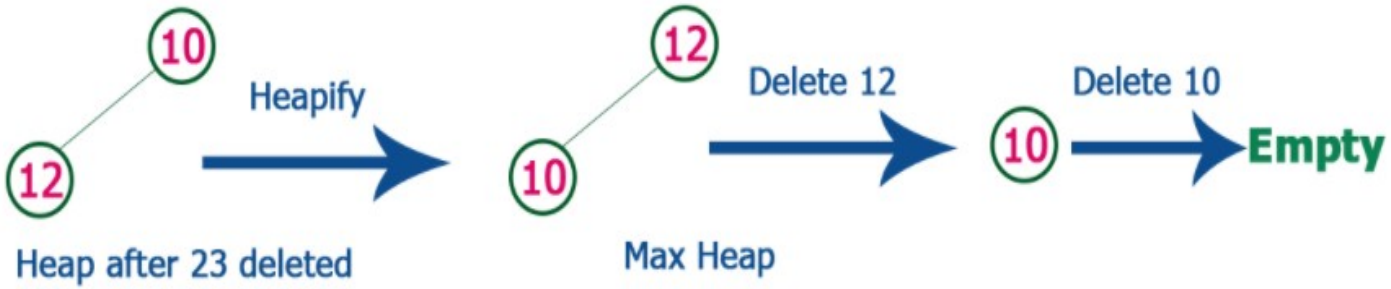
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.


list of numbers after swapping 23 with 12.

12, 15, 10, **23, 55, 77, 82, 90**

# EXAMPLE: CUNTD....

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.




list of numbers after Deleting 15, 12 & 10 from the Max Heap.

**10, 12, 15, 23, 55, 77, 82, 90**

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

# COMPLEXITY OF THE HEAP SORT ALGORITHM

To sort an unsorted list with 'n' number of elements, following are the complexities.

- Worst Case : O(n log n)
- Best Case : O(n log n)
- Average Case : O(n log n)

# Quick Sort

Adyasha Rath

# "DIVIDE AND CONQUER"

- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves  Merge sort
- **Idea 2 :** Partition array into items that are "small" and items that are "large", then recursively sort the two sets  Quick sort

# QUICKSORT

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that Merge sort does

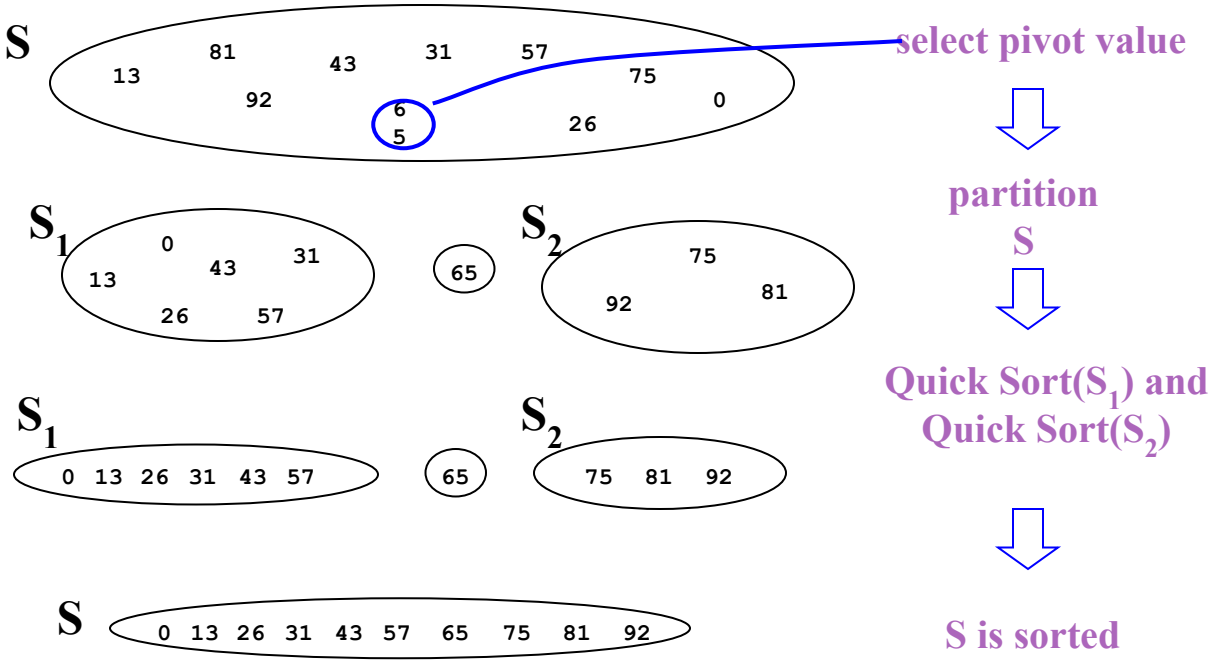
  - Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot

  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

# "FOUR EASY STEPS"

- To sort an array **S**
  1. If the number of elements in **S** is 0 or 1, then return. The array is sorted.
  2. Pick an element $v$ in **S**. This is the ***pivot*** value.
  3. Partition **S**-$\{v\}$ into two disjoint subsets, $\mathbf{S}_1$ = {all values $x \leq v$}, and $\mathbf{S}_2$ = {all values $x \geq v$}.
  4. Return Quick sort ($\mathbf{S}_1$), $v$, Quick sort ($\mathbf{S}_2$)

# THE STEPS OF QUICK SORT



# PARTITION ALGORITHM

```
Partition (A, lb, ub)
   {
        Pivot=a[lb];
        Start=lb;
end
    while (start<end)
     {
           while(a[start]<=pivot)
```

```
    {
        start++;
    }
while (a[end]>pivot)
    {
end--;
    }
if(start<end)
    {
swap(a[start], a[end]);
    }
}
```

```
swap(a[lb], a[end])
return end;
}
```

**QUICK SORT**

```
Quick sort (A, lb, ub)
{
      if(lb<ub)
       {
            Loc=Partition(A, lb, ub);
            Quick sort(A, lb, loc-1);
           Quick sort(A, loc+1, ub);
        }
}
```

## Principles of Quick Sort

- Quick Sort means by sorting the array in **ascending order**.
- We will choose one element as **pivot element or key element**.
- Either we will choose a pivot element in first or last or any random element as pivot element or medium element as pivot element
- There are many techniques to use pivot element.
- But I will choose the starting element as pivot element.
- Quick sort is based on **divide and conquer** techniques
- Complete array will be divided into **sub arrays**.
- Dividing arrays into sub arrays is known as **PARTITIONING**
- Partitioning of that array is known as **backbone** of that array.

## Principles of Quick Sort

- We are going to partition the array in a way such that all the elements less than this pivot element would be to the left side of the pivot element and all the elements greater than this pivot element would be to the right side of the pivot element.
- If any element is there which is equal to this pivot element could go either of the way, it depends on the how to implement that logic.
- Partition means to find out the proper place for the pivot element and partitioning the array into two parts.

## Principles of Quick Sort

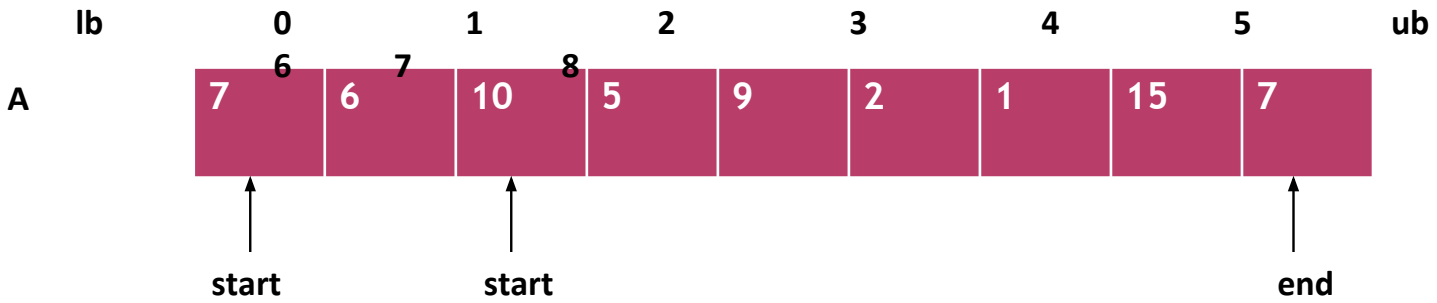
Conditions

If you find any element which is less than the or equal to this pivot element then we are going to increment the this until we found greater than this.

Similarly from ‘end’ if any element is found less than the pivot element

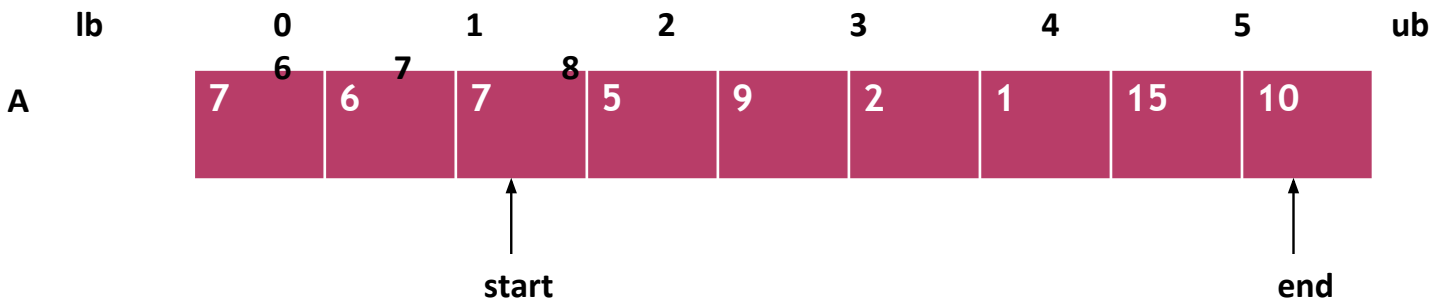
then decrement stops

# EXAMPLE




**Pivot=7 a[0]**
**7<=7, yes increment the start**
**6<=7, yes, increment the start**
**10<=7, no greater than the pivot element, stop**
**Location of start will be at 10**

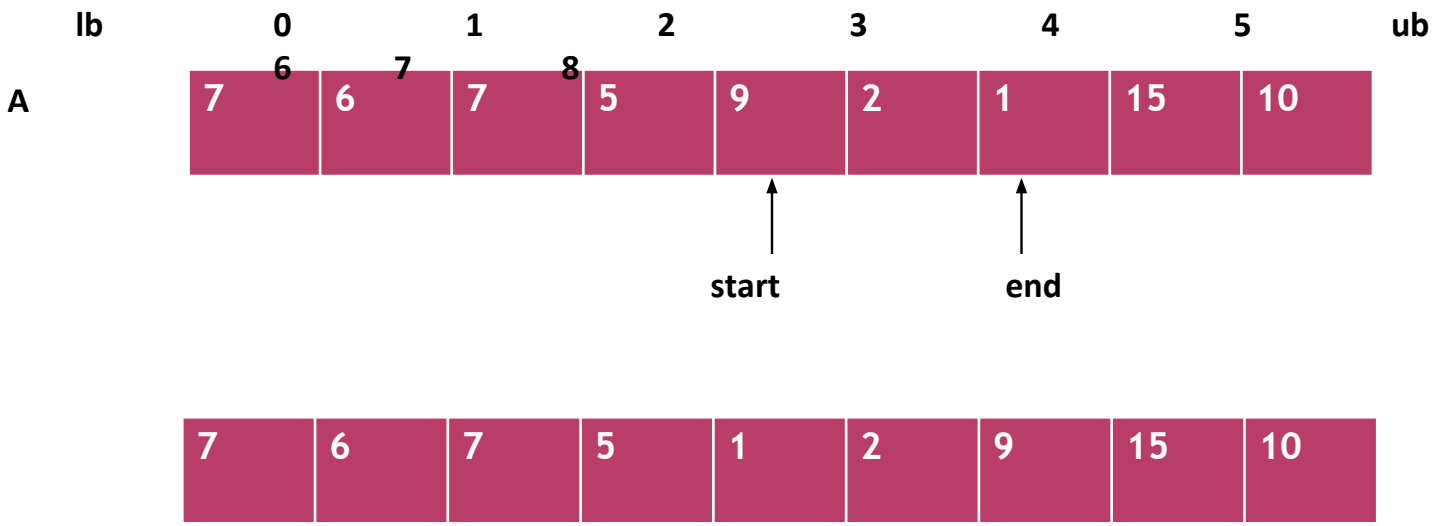
**At end**
**7>7, 7=7, No, we will not decrement this end**
**Now we will swap a[start] with a[end]**

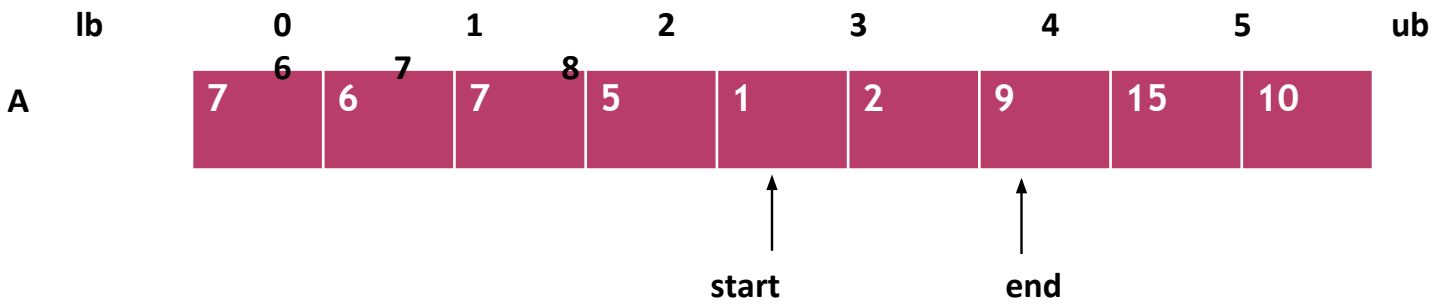



7<=7, yes increment
5<=7, yes increment
9<=7, no stop

At end

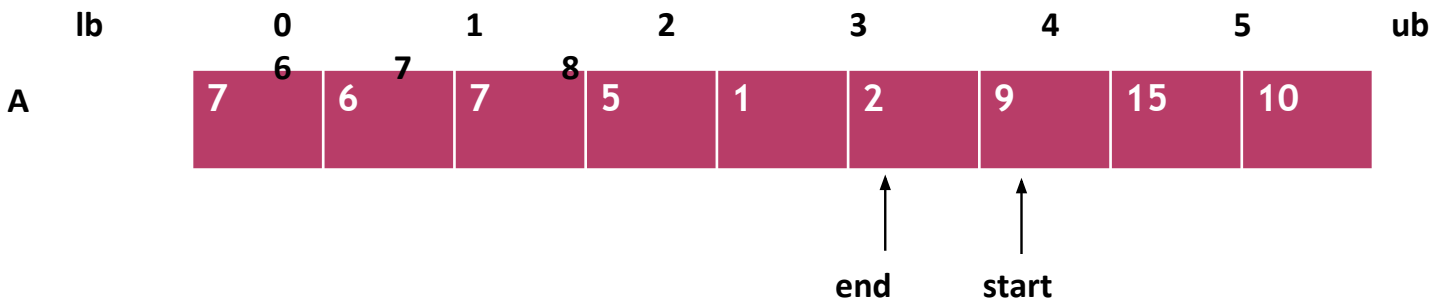
10>7, yes decrement
15>7, yes decrement
1>7, No, stop decrement


lb
0
1
2
3
4
5
ub
6
7
8
A
7
6
7
5
9
2
1
15
10
start
end
7
6
7
5
1
2
9
15
10



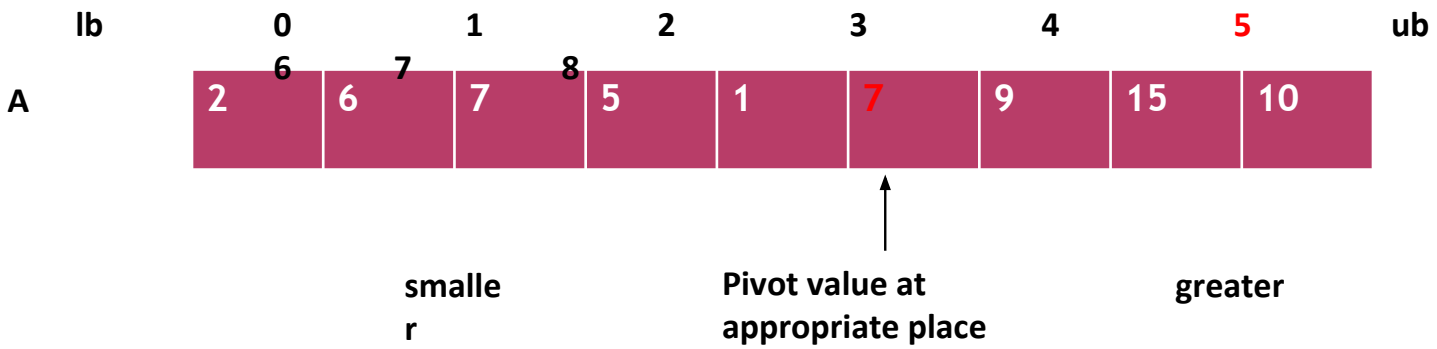

1<=7, yes increment
2<=7, yes increment
9<=7, no stop increment

At end
A[end]=9
9>7, yes decrement
2>=7, no stop




**We can not swap here because start variable has crossed the end variable. Start is now greater than end.**
**We can swap the pivot element with end element**

# AFTER SWAPPING

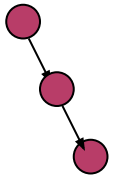



**We have considered the pivot element as 7**
**Whatever element is present at left side of pivot element these are smaller than the pivot element and whatever present at the right side of the pivot element these are greater than the pivot**

# PERFORMANCE OF QUICK SORT

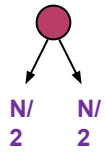
- **Running time of Quick sort depends on whether the partitioning is balanced or un-balanced**
- **The balanced partitioning gives best case time complexity**

- **The unbalanced partitioning gives worst case time complexity**

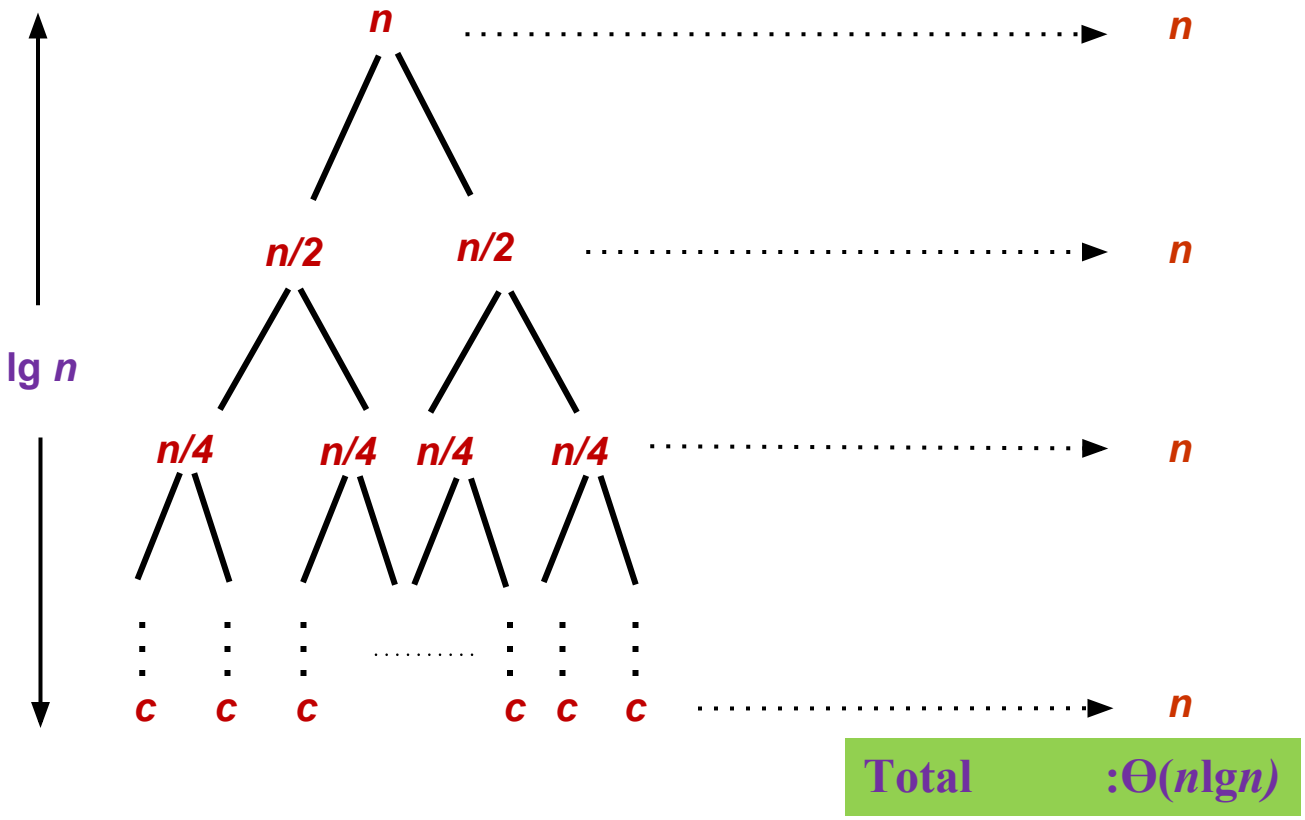


# QUICK SORT BEST CASE PERFORMANCE

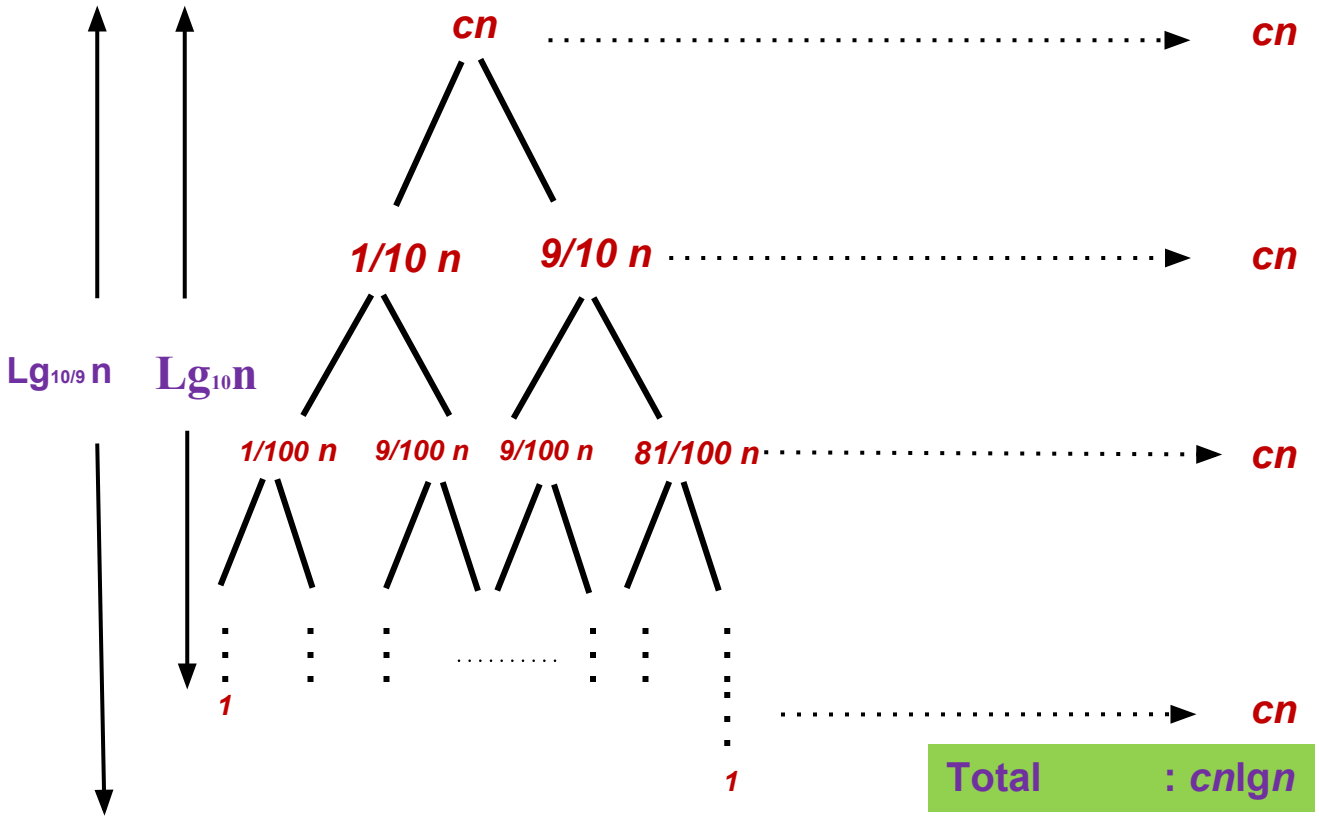
- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - T(0) = T(1) = O(1)
    - constant time if 0 or 1 element
  - For N > 1, 2 recursive calls plus linear time for partitioning
  - **T(N) = 2T(N/2) + O(N)**
    - Same recurrence relation as Merge sort
  - T(N) = O(N log N)



# RECURSION TREE FOR MERGE SORT



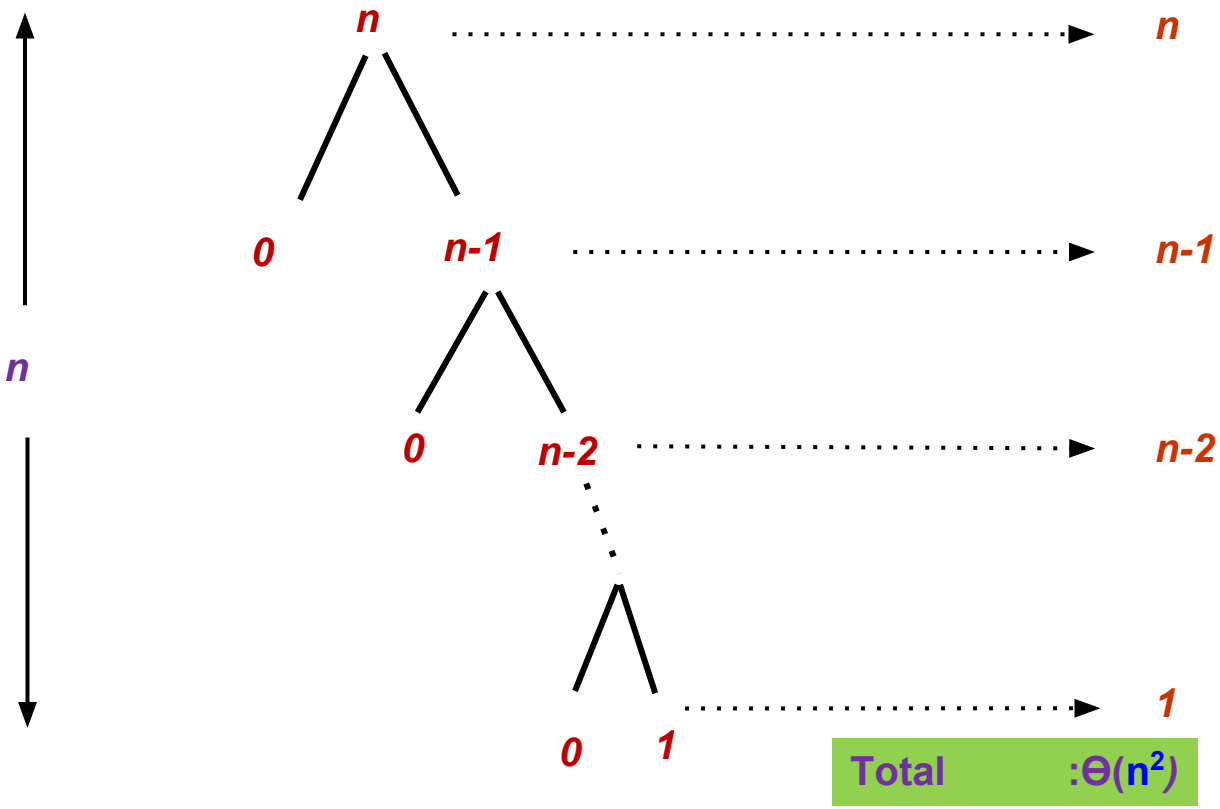
# BALANCED PARTITIONING



# QUICKSORT WORST CASE PERFORMANCE

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - $T(N) \leq a$ for $N \leq C$
  - $T(N) = T(N-1) + \Theta(N)$
  - $= T(N-2) + T(N-2) + \Theta(N)$
  - $= T(0) + \Theta(1) + \ldots + \Theta(N)$
  - $T(N) = \Theta(N^2)$
- Fortunately, *average case performance* is O(N log N)

# WORST CASE



# AVERAGE CASE ANALYSIS

- In average case, PARTITION produces a mix of "GOOD" and "BAD" at the alternate levels in the tree.
- In Fig(a) the partition at the root costs n and produces a "BAD" split:
  - Two sub-arrays **0** and **n-1**
- The partitioning of the sub-arrays of size n-1 costs is n-1 and produces a "GOOD" split:
  - Two sub-arrays of size **(n-1)/2-1** and **(n-1)/2**
- In Fig(b), a single level of recursion tree that is very well balanced
- The combination of bad split followed by good split produces 3-sub arrays of size **0, (n-1)/2-1** and **(n-1)/2** at the combined partition cost of **Θ(n)+ Θ(n-1)= Θ(n)**
- The **Θ(n-1)** cost of bad split can be absorbed into the **Θ(n)** of good split.

# AVERAGE CASE

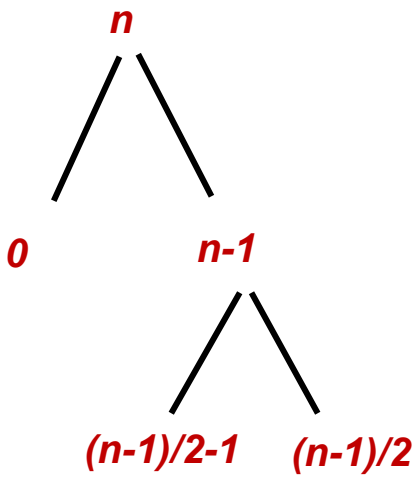



Fig (a): bad split

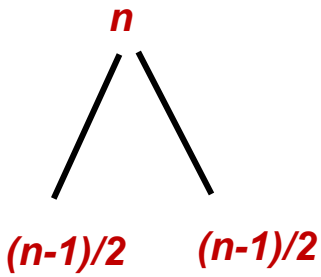
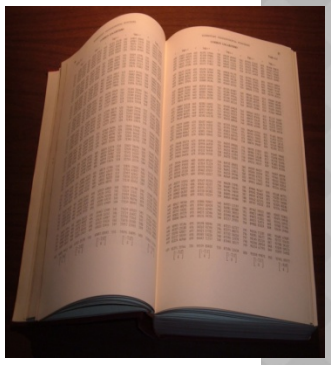



Fig (b): good split

# HASHING

# CONTENTS



# Introduction to Hashing

- Hashing is the technique used for performing almost constant time search in case of insertion, deletion and find operation.
- The essence of hashing is to facilitate the next level searching method when compared with the linear or binary search.
- It is the process of mapping large amount of data item to a smaller table with the help of a hashing function.
- The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection (i.e. collection size).
- Due to this hashing process, the result is a Hash data structure that can store or retrieve data items in an average time disregard to the collection size.

CONTINUED…

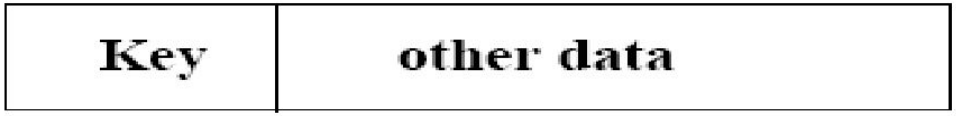
- Let ‘n’ distinct record Table is one of the important data structure used for information retrieval.
- Records with keys K1,K2-----Kn are stored in a file and we want to retrieve a record with a given key value K.
- One way is to start from the first record by comparing K with its key value, if found then stop ,otherwise proceed to the next record. The searching time is directly proportional to the number of records in the file.
- Another way is to use an **access table, where** the searching time is significantly reduced.

- Let us assume a function f and apply this function to K, then it returns ‘i’ that is f(k)=i. The ith entry of the access table gives the location of record with key value K.
- One type of such access table is known as Hash table.
- The hash table is an array which contains key values with pointers to the corresponding records.
- The basic idea here is to place a key inside the hash table, and the location/index of that key will be calculated from the given key value itself.
- The one to one correspondence between a key value and index in the hash table is known as **hashing**.

# Example: The Search Problem

- Find items with keys matching a given search key
  - Given an array A, containing n keys, and a search key x, find the index i such as x=A[i]
  - As in the case of sorting, a key could be part of a large record.

example of a record



# Applications

- Keeping track of customer account information at a bank
  - Search through records to check balances and perform transactions
- Keep track of reservations on flights
  - Search to find empty seats, cancel/modify reservations
- Search engine
  - Looks for all documents containing a given word

# Hash Table

- There are some type of tables which helps to retrieve information in efficient manner.
- The hash table is one of them.
- This table is an array of constant size which depends on applications.
- The hash table contains key values with pointers to the corresponding records.
- The basic idea is to place key value into location in the hash table and this location will be calculated from the key value itself.
- The one to one correspondence between the key value and index in the hash table is known as hashing.

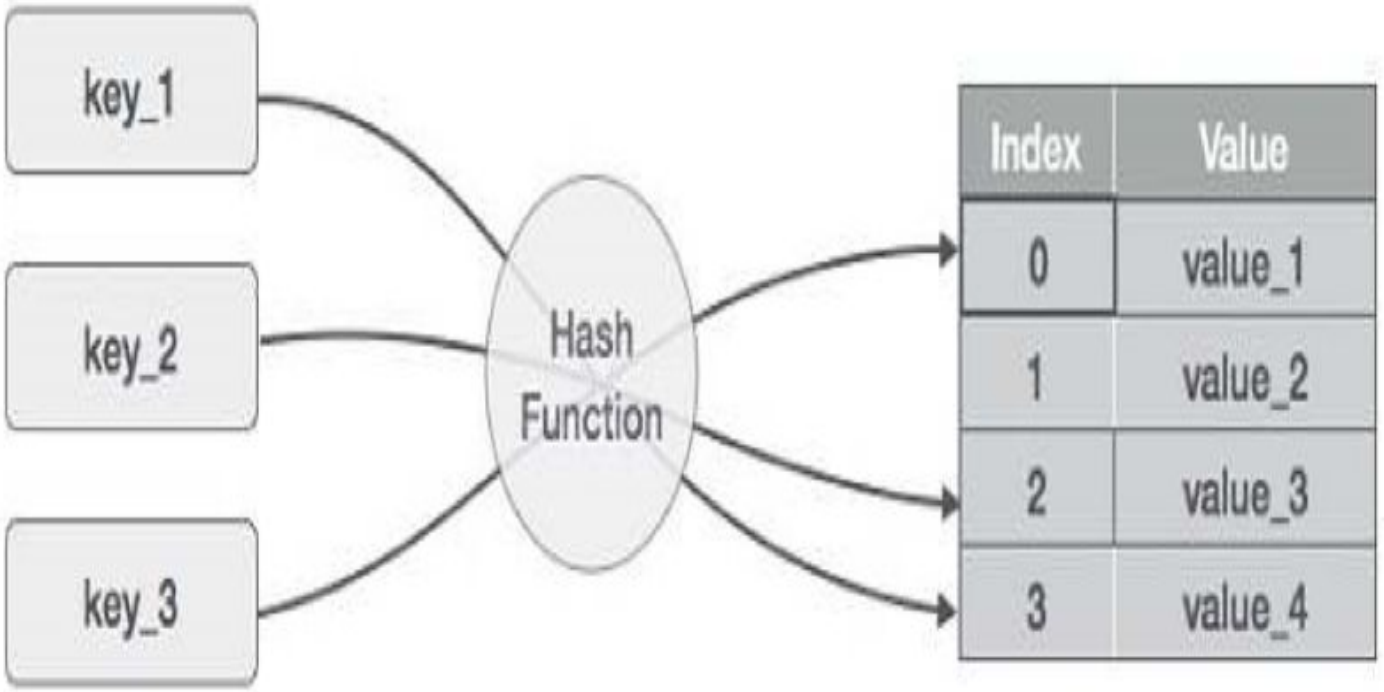
- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called *key*, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc
  - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1.*
- The mapping is called a *hash function*.

A hash table, or a hash map, is a data structure that associates keys (names) with values (attributes).

- Look-Up Table
- Dictionary
- Cache
- Extended Array

Continued…

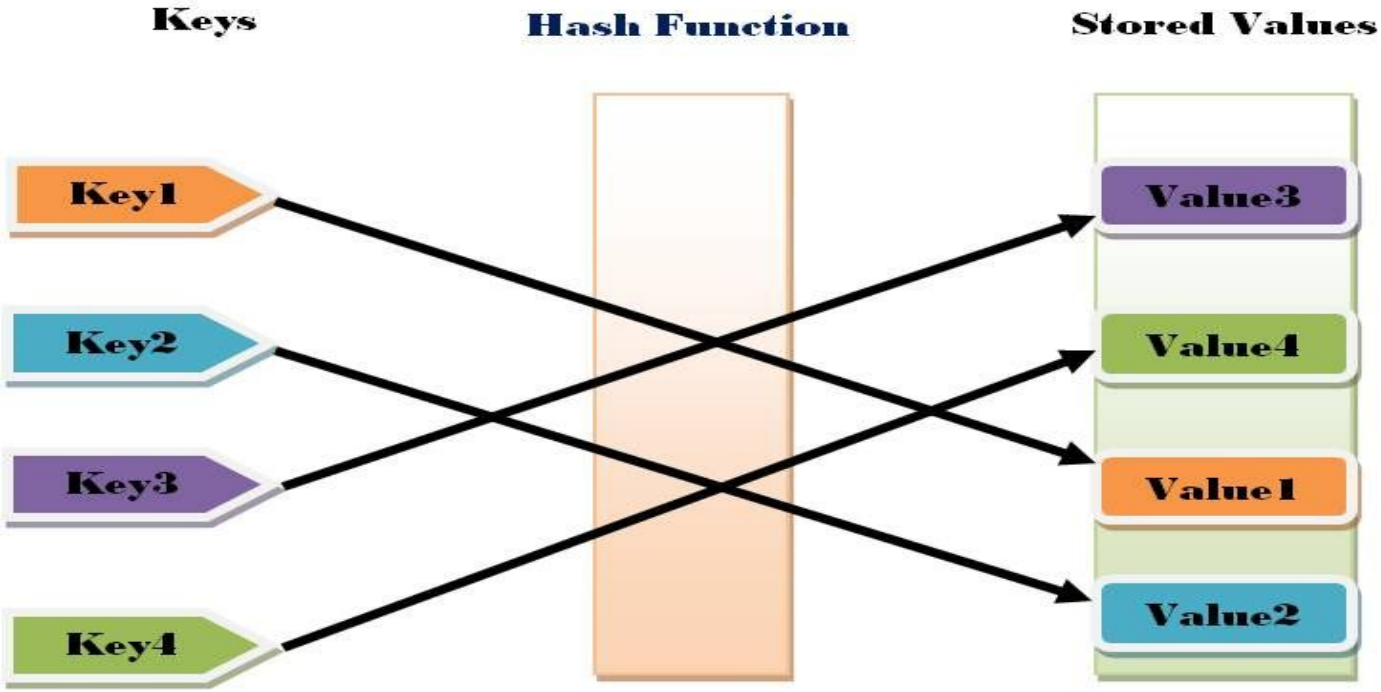
# Example: HASH TABLE



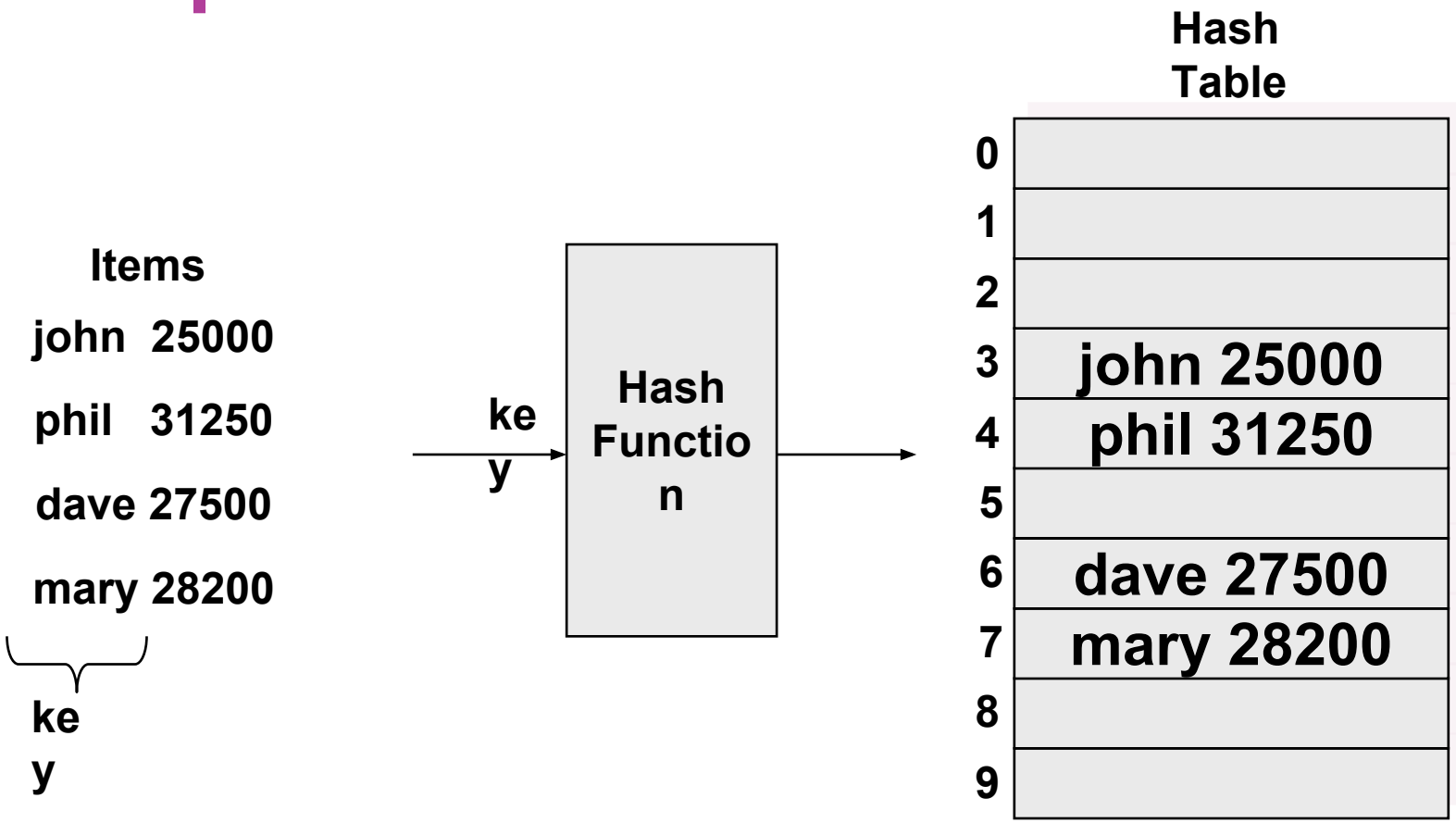
# Choosing a Hash Function

- A hash function maps keys to small integers (buckets). An ideal hash function maps the keys to the integers in a random-like manner, so that bucket values are evenly distributed even if there are regularities in the input data.

- This process can be divided into two steps:
  - Map the key to an integer.
  - Map the integer to a bucket.



# Example


Items
john 25000
phil 31250
dave 27500
mary 28200
key
key
Hash Function
Hash Table
0
1
2
3
john 25000
4
phil 31250
5
6
dave 27500
7
mary 28200
8
9

- **The hash function:**
  - ✔ **must be simple to compute.**
  - ✔ **must distribute the keys evenly among the cells.**

**<u>Different Approaches to generate hash function</u>**

1. **Truncation Method**
2. **Mid Square Method**
3. **Folding Method**
4. **Modular Method**

# 1. Truncation Method

Ignore a part of the key and use the remaining part directly as the index

Ex:1 If a hash table contains 999 entries at the most or 999 different key indexes may be kept, then a hash function may be defined such that from an eight digit integer 12345678, first, second and fifth digits from the right may be used to define a key index i.e. 478, which is the key position in the hash lookup table where this element will be inserted. Any other key combination may be used.

Ex:2 If students have an 9-digit identification number, take the last 3 digits as the table position

e.g. 925371622

becomes

622

# 2. Mid Square Method

- In Mid-Square method, the key element is multiplied by itself to yield the square of the given key.
- If the hash table contains maximum 999 entries, then three digits from the result of square may be chosen as a hash key for mapping the key element in the lookup table.
- It generates random sequences of hash keys, which are generally key dependent.
- Mid Square method is a flexible method of selecting the hash key value.

Example: If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57. All digits of the original key value (equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Thus, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

# 3. Folding Method

- Partition the key into several parts and combine the parts in a convenient way (often addition or multiplication) to obtain the index.
- Ex-1:

  An eight digit integer can be divided into groups of three, three and two digits (or any other combination) the groups added together and truncated if necessary to be in the proper range of indices.

  Hence 12345678 maps to 123+456+78 = 657, since any digit can affect the index, it offers a wide distribution of key values.
- Ex-2: Split a 9-digit number into three 3-digit numbers, and add them

  e.g. 925371622 becomes 925 + 376 + 622 = 1923

# 4. Modular Method

- For mapping a given key element in the hash table, mod operation of individual key is calculated. The remainder denotes particular address position of each element.
- The result so obtained is divided by an integer, usually taken to be the size of the hash table to obtain the remainder as the hash key to place that element in the lookup table.

# 5. The Division Method

- Idea:
  - Map a key k into one of the m slots by taking the remainder of k divided by m

    $h(k) = k \bmod m$
- Advantage:
  - fast, requires only one operation
- Disadvantage:
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

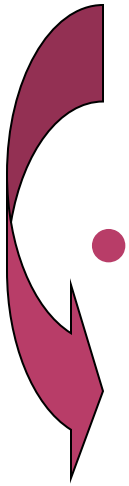
# Example - The Division Method

- If $m = 2^p$, then h(k) is just the least significant p bits of k
  - p = 1 ⇒ m = 2
    ⇒ h(k) = {0, 1}, least significant 1 bit of k
  - p = 2 ⇒ m = 4
    ⇒ h(k) = {0, 1, 2, 3}, least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

| | m 97 | m 100 |
|---|---|---|
| 16838 | 57 | 38 |
| 5758 | 35 | 58 |
| 10113 | 25 | 13 |
| 17515 | 55 | 15 |
| 31051 | 11 | 51 |
| 5627 | 1 | 27 |
| 23010 | 21 | 10 |
| 7419 | 47 | 19 |
| 16212 | 13 | 12 |
| 4086 | 12 | 86 |
| 2749 | 33 | 49 |
| 12767 | 60 | 67 |
| 9084 | 63 | 84 |
| 12060 | 32 | 60 |
| 32225 | 21 | 25 |
| 17543 | 83 | 43 |
| 25089 | 63 | 89 |
| 21183 | 37 | 83 |
| 25137 | 14 | 37 |
| 25566 | 55 | 66 |
| 26966 | 0 | 66 |
| 4978 | 31 | 78 |
| 20495 | 28 | 95 |
| 10311 | 29 | 11 |
| 11367 | 18 | 67 |

# Collision

- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If |K| ≤ m, collisions may or may not happen, depending on the hash function
  - If |K| > m, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function



# Collision Resolution(Open Hashing)

- If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it.
- There are several methods for dealing with this:
  - Separate chaining
  - Open addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

# SEPARATE CHAINING

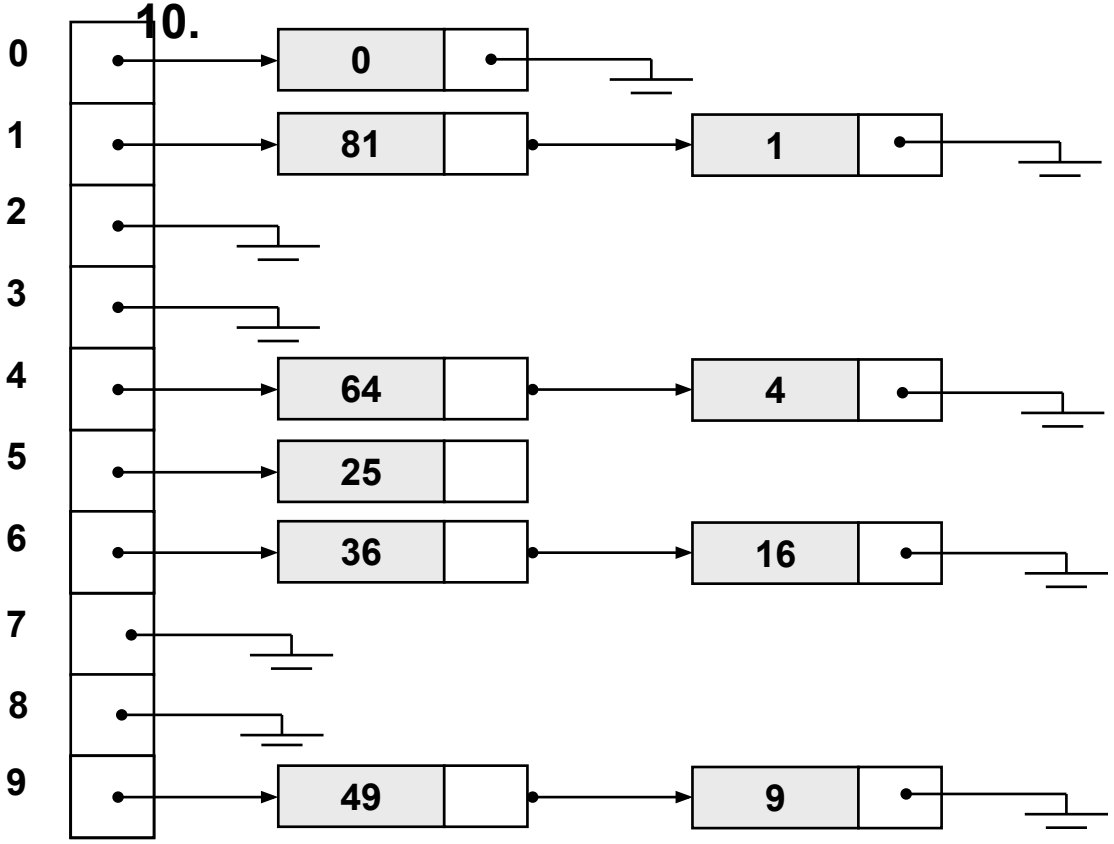
- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.

- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.

## EXAMPLE

**Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81**

**hash(key) = key % 10.**

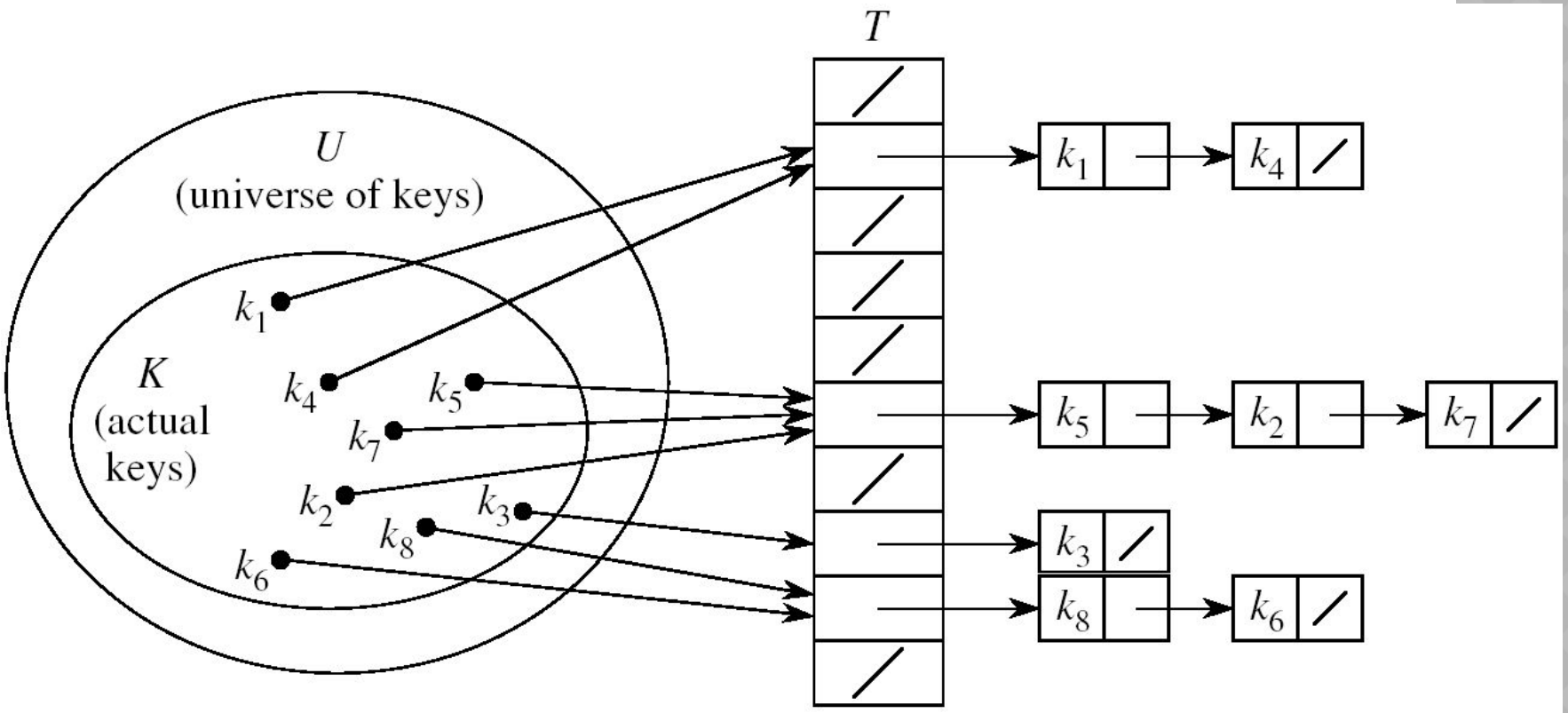


# OPERATIONS

- **Initialization**: all entries are set to NULL
- **Find**:
  - locate the cell using hash function.
  - sequential search on the linked list in that cell.
- **Insertion**:
  - Locate the cell using hash function.
  - (If the item does not exist) insert it as the first item in the list.
- **Deletion**:
  - Locate the cell using hash function.
  - Delete the item from the linked list.

# Handling Collisions Using Separate Chaining

- Idea:
  - Put all elements that hash to the same slot into a linked list

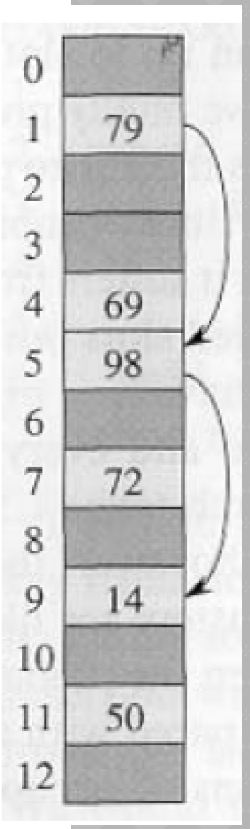



  - Slot j contains a pointer to the head of the list of all elements that hash to j

# Closed Hashing(Open Addressing)

- If we have enough contiguous memory to store all the keys (m > N) ⇒ store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - Insertion: if a slot is full, try another one, until you find an empty one
  - Search: follow the same sequence of probes
  - Deletion: more difficult ... (we'll see why)
- Search time depends on the length of the probe sequence!

e.g., insert 14



# Generalize hash function notation:

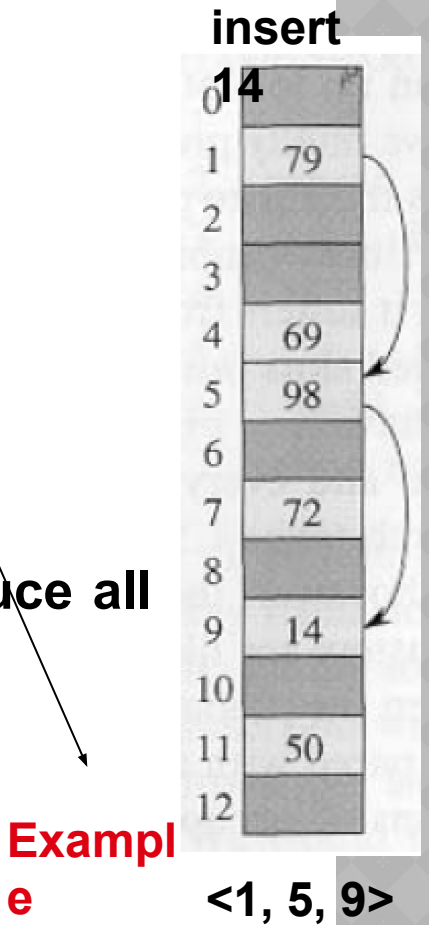
- A hash function contains two arguments now:
  (i) Key value, and (ii) Probe number

  $h(k,p), \quad p=0,1,...,m-1$

- Probe sequences

  $\langle h(k,0), h(k,1), ..., h(k,m-1) \rangle$

  - Must be a permutation of $\langle 0,1,...,m-1 \rangle$
  - There are $m!$ possible permutations
  - Good hash functions should be able to produce all sequences



# Common Open Addressing Methods

- **Linear probing**
- **Quadratic probing**
- **Double hashing**

- **Note: None of these methods can generate more than $m^2$ different probing sequences!**

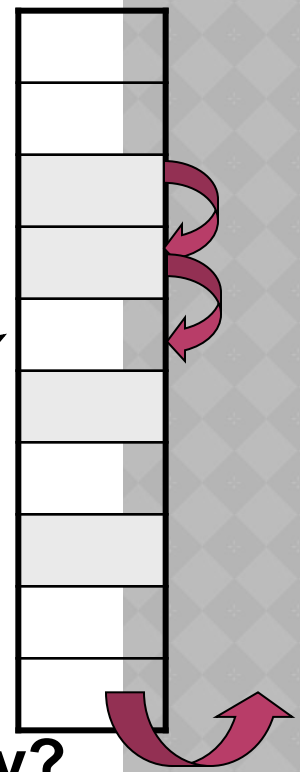
# Linear probing: Inserting a key

- Idea: when there is a collision, check the next available position in the table (i.e., probing)

$h(k,i) = (h_1(k) + i) \bmod m$

$i=0,1,2,\ldots$

- First slot probed: $h_1(k)$
- Second slot probed: $h_1(k) + 1$
- Third slot probed: $h_1(k)+2$, and so on



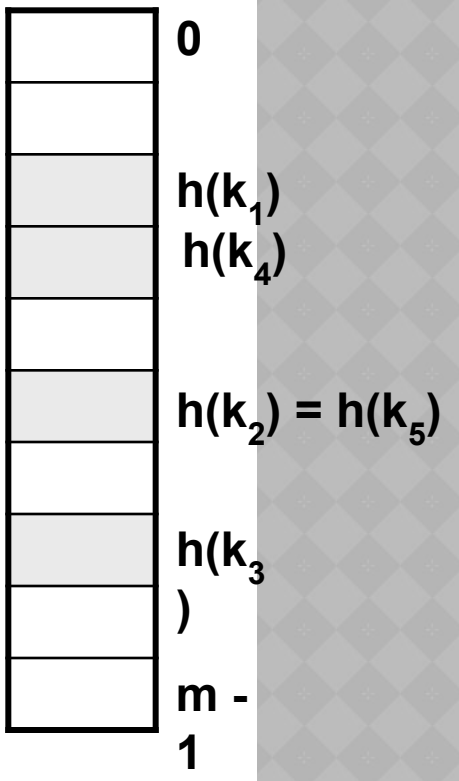
wrap around

probe sequence: < h1(k), h1(k)+1 , h1(k)+2 , …>

- Can generate m probe sequences maximum, why?

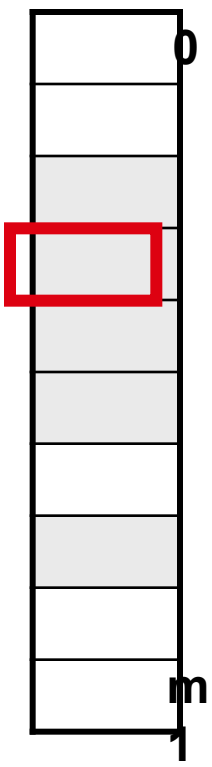
# Linear probing: Searching for a key

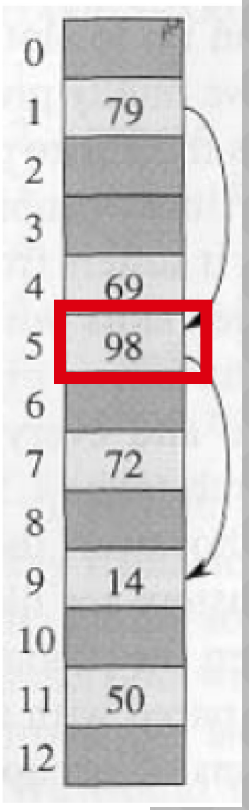
- Three cases:
  (1) Position in table is occupied with an element of equal key
  (2) Position in table is empty
  (3) Position in table occupied with a different element
- Case 2: probe the next higher index until the element is found or an empty position is found
- The process wraps around to the beginning of the table



# Linear probing: Deleting a key

- **Problems**
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys



# Linear probing: Example

**If we have a list of size 20 (m = 20). We want to put some elements in linear probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}**

| x | h(x, i) = (h'(x) + i) mod 20 |
|---|---|
| 96 | i = 0, h(x, 0) = 16 |
| 48 | i = 0, h(x, 0) = 8 |
| 63 | i = 0, h(x, 0) = 3 |
| 29 | i = 0, h(x, 0) = 9 |
| 87 | i = 0, h(x, 0) = 7 |
| 77 | i = 0, h(x, 0) = 17 |
| 48 | i = 0, h(x, 0) = 8<br>i = 1, h(x, 1) = 9<br>i = 2, h(x, 2) = 10 |
| 65 | i = 0, h(x, 0) = 5 |
| 69 | i = 0, h(x, 0) = 9<br>i = 1, h(x, 1) = 10<br>i = 2, h(x, 2) = 11 |
| 94 | i = 0, h(x, 0) = 14 |
| 61 | i = 0, h(x, 0) = 1 |

**Hash Table: -**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 61 | | 63 | | 65 | | 87 | 48 | 29 | 48 | 69 | | | 94 | | 96 | 77 | | |

# Quadratic Probing

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where $h' : U --> (0, 1, . . , m - 1)$

- Clustering problem is less serious but still an issue (*secondary clustering*)

i=0,1,2,..

- How many probe sequences quadratic probing generate ? $m$

(the initial probe position determines the probe sequence)

# Quadratic Probing: Example

**If we have a list of size 20 (m = 20). We want to put some elements in linear probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}**

| x | $h(x, i) = (h'(x) + i^2) \bmod 20$ |
|---|---|
| 96 | i = 0, h(x, 0) = 16 |
| 48 | i = 0, h(x, 0) = 8 |
| 63 | i = 0, h(x, 0) = 3 |
| 29 | i = 0, h(x, 0) = 9 |
| 87 | i = 0, h(x, 0) = 7 |
| 77 | i = 0, h(x, 0) = 17 |
| 48 | i = 0, h(x, 0) = 8<br>i = 1, h(x, 1) = 9<br>i = 2, h(x, 2) = 12 |
| 65 | i = 0, h(x, 0) = 5 |
| 69 | i = 0, h(x, 0) = 9<br>i = 1, h(x, 1) = 10 |
| 94 | i = 0, h(x, 0) = 14 |
| 61 | i = 0, h(x, 0) = 1 |

**Hash Table**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 61 | | 63 | | 65 | | 87 | 48 | 29 | 69 | | 48 | | 94 | | 96 | 77 | | |

# Double Hashing

(1) Use one hash function to determine the first slot

(2) Use a second hash function to determine the increment for the probe sequence

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,...$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on ...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element
- Can generate $m^2$ probe sequences maximum

# Double Hashing: Example

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- **Insert key 14:**

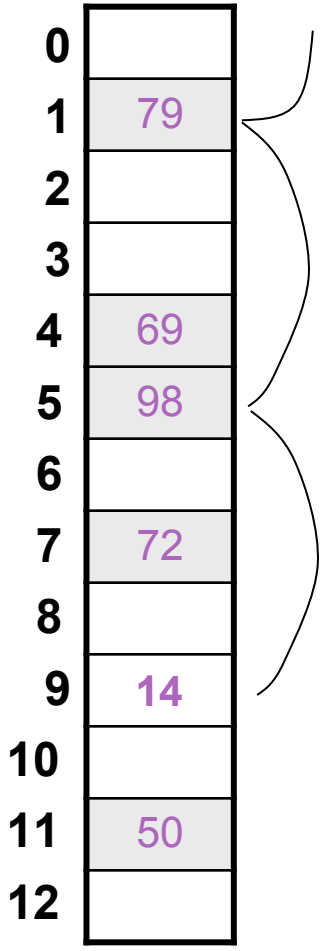
$h_1(14,0) = 14 \bmod 13 = 1$

$h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$



# Applications of Hashing

- Compilers use hash tables to keep track of declared variables.
- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time.
- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again.
- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different.
- Storing sparse data

# THANK YOU