

# Organizing Data: Arrays, Linked Lists



# Announcements

- Exam information
  - Thursday
  - Exam coverage on Piazza
  - Review Session on Wednesday
  
- Questions?

From Yesterday

# Example 2: Merge

list a

0 1 2 3

58 67 74 90

0 1 2 3

58 67 74 90

0 1 2 3

58 67 74 90

0 1 2 3

58 67 74 90

0 1 2 3

58 67 74 90

list b

0 1 2 3

19 26 31 44

0 1 2 3

19 26 31 44

0 1 2 3

19 26 31 44

0 1 2 3

19 26 31 44

0 1 2 3

19 26 31 44

list c

0 1 2 3 4 5 6 7

19

0 1 2 3 4 5 6 7

19 26

0 1 2 3 4 5 6 7

19 26 31

0 1 2 3 4 5 6 7

19 26 31 44

0 1 2 3 4 5 6 7

19 26 31 44 58 67 74 90

# Analyzing Efficiency

- **Constant time** operations: comparing values and appending elements to the output.
- If you merge two lists of size  $i/2$  into one new list of size  $i$ , what is the **maximum number of appends** that you must do? **maximum number of comparisons?**
- Example: say we are merging two pairs of 2-element lists:




8 appends for 8 elements

- If you have a group of lists to be merged pairwise, and the total number of elements in the whole group is  $n$ , the total number of appends will be  $n$ .
- **Worse case number comparisons?**  $n/2$  or less, but still  $O(n)$

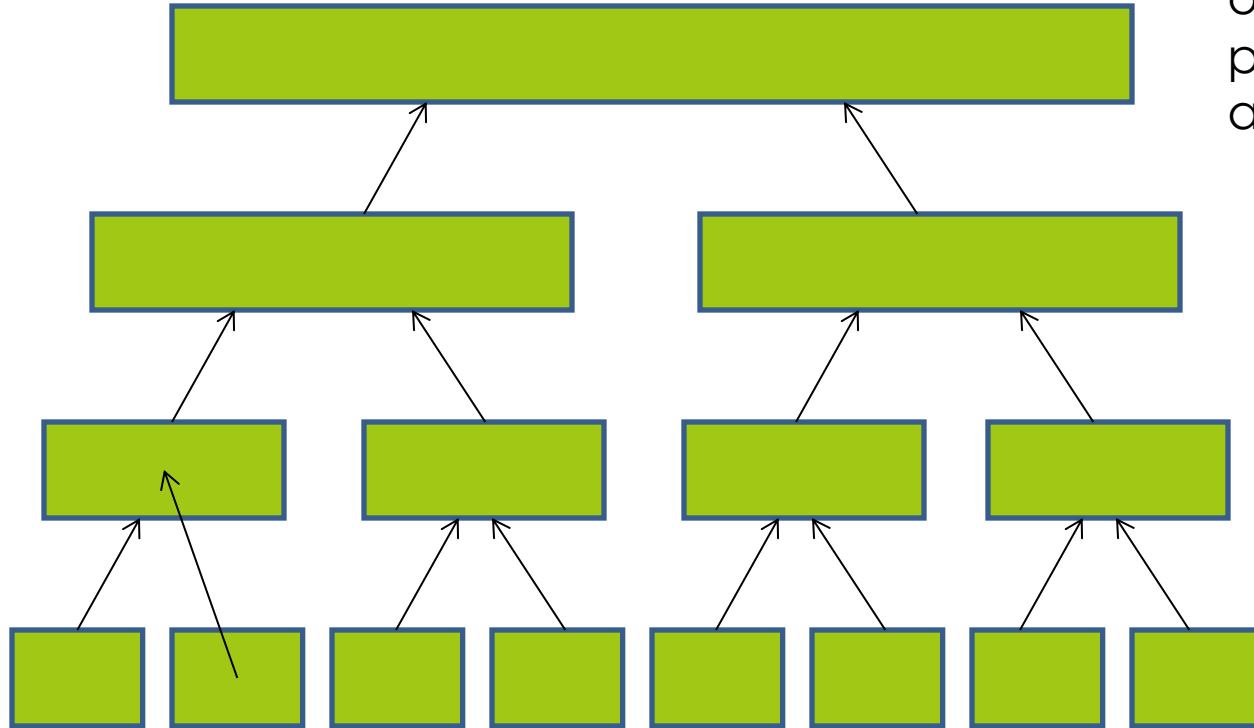
# How many merges?

- We saw that each group of merges of  $n$  elements takes  $O(n)$  operations.
- How many times do we have to merge  $n$  elements to go from  $n$  groups of size 1 to 1 group of size  $n$ ?
- Example: Merge sort on 32 elements.
  - Break down to groups of size 1 (base case).
  - Merge 32 lists of size 1 into 16 lists of size 2.
  - Merge 16 lists of size 2 into 8 lists of size 4.
  - Merge 8 lists of size 4 into 4 lists of size 8.
  - Merge 4 lists of size 8 into 2 lists of size 16.
  - Merge 2 lists of size 16 into 1 list of size 32.
- **In general:  $\log_2 n$  merges of  $n$  elements.**


$$5 = \log_2 32$$

# Putting it all together

It takes  $\log_2 n$  merges to go from  $n$  groups of size 1 to a single group of size  $n$ .



Total number of elements per level is always  $n$ .

It takes  $n$  appends to merge all pairs to the next higher level.

**Multiply the number of levels by the number of appends per level.**

# Big O

- In the worst case, merge sort requires  **$O(n \log_2 n)$**  time to sort an array with  $n$  elements.

Number of operations

$$n \log_2 n$$

$$(n + n/2) \log_2 n$$

$$4n \log_{10} n$$

$$n \log_2 n + 2n$$

Order of Complexity

$$O(n \log n)$$

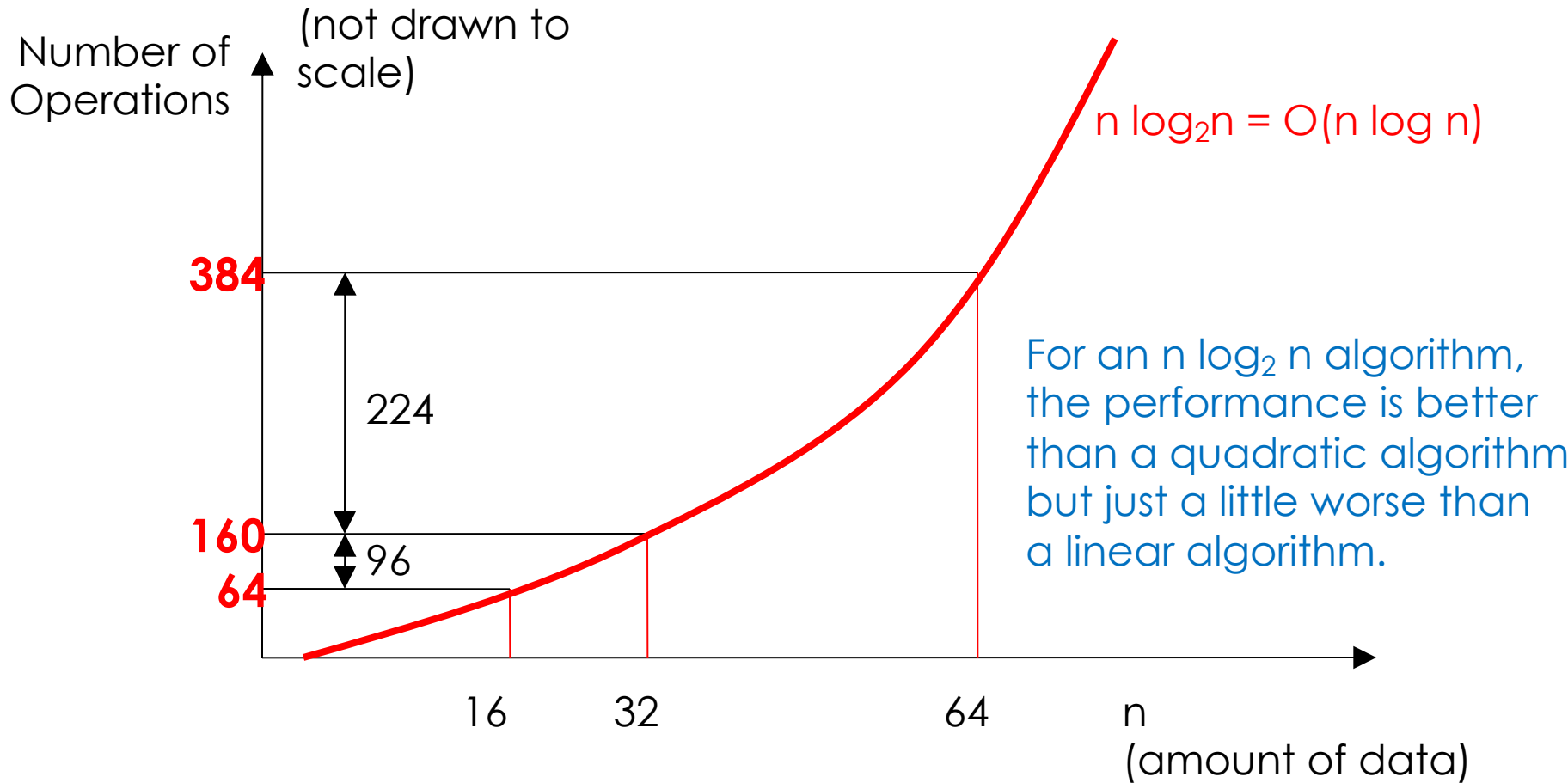
$$O(n \log n)$$

$$O(n \log n)$$

$$O(n \log n)$$



# $O(N \log N)$



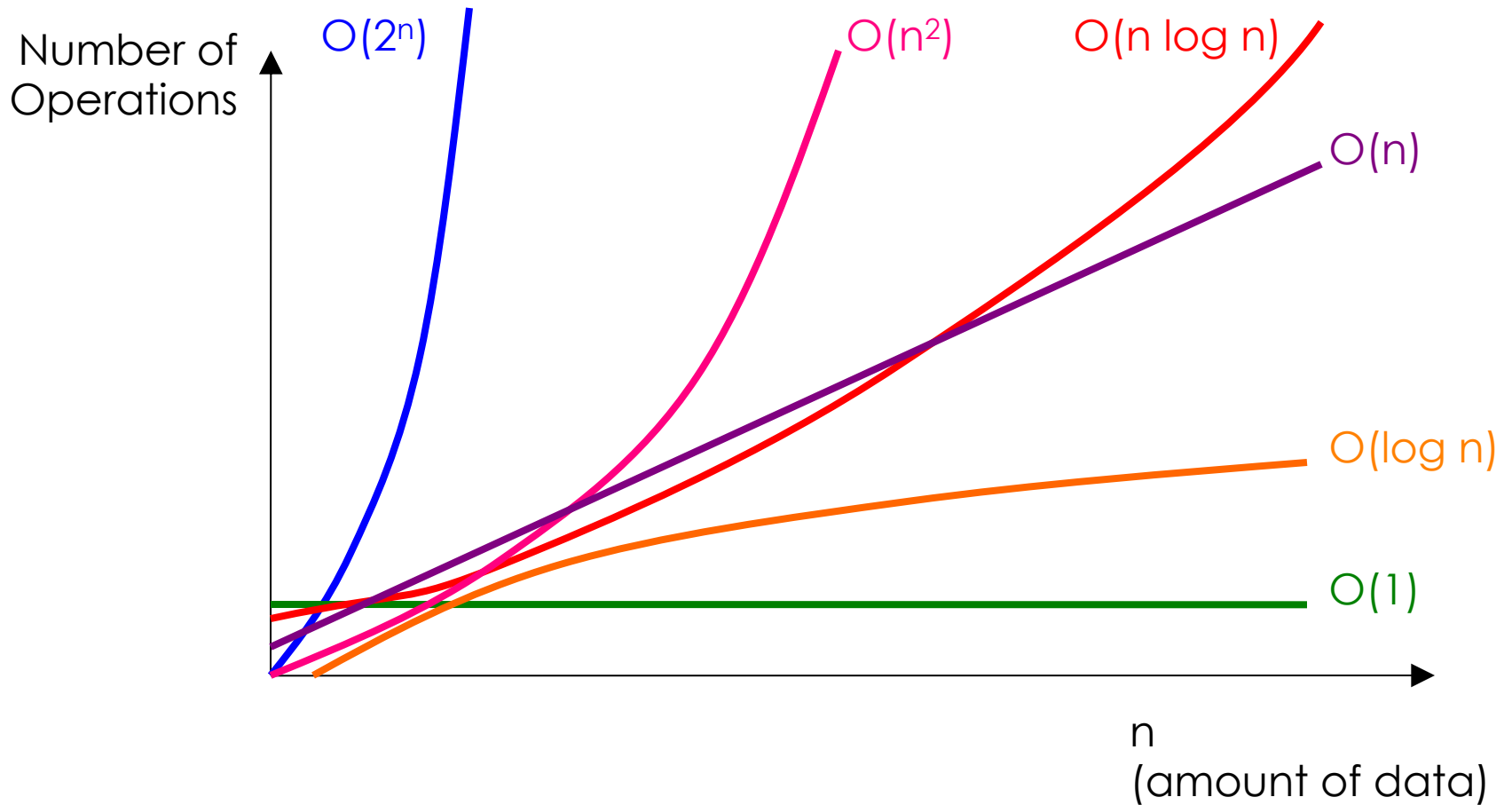
# Merge vs. Insertion Sort

n	isort $(n(n+1)/2)$	msort $(n \log_2 n)$	Ratio
8	36	24	0.67
16	136	64	0.47
32	528	160	0.3
$2^{10}$	524,800	10,240	0.02
$2^{20}$	549,756,338,176	20,971,520	0.00004

# Sorting and Searching

- Recall that if we wanted to use binary search, the list must be sorted.
- ▣ What if we sort the array first using merge sort?
  - ▣ Merge sort  $O(n \log n)$  (worst case)
  - ▣ Binary search  $O(\log n)$  (worst case)
  - ▣ Total time:  
(worst case)  $O(n \log n) + O(\log n) = \mathbf{O(n \log n)}$

# Comparing Big O Functions



# From PS5

- 3b
  - Budget of 5 comparisons
  - How many items can we search?

# Analyzing Binary Search

- ▣ Suppose we search for a key larger than anything in the list.
- ▣ Example sequences of range sizes:
  - 8, 4, 2, 1            (4 key comparisons)
  - 16, 8, 4, 2, 1        (5 key comparisons)
  - 17, 8, 4, 2, 1        (5 key comparisons)
  - 18, 9, 4, 2, 1        (5 key comparisons)
  - ...
  - 31, 15, 7, 3, 1        (still 5 key comparisons)
  - 32, 16, 8, 4, 2, 1    (at last, 6 key comparisons)
- ▣ Notice:  $8 = 2^3$ ,  $16 = 2^4$ ,  $32 = 2^5$
- ▣ Therefore:  $\log 8 = 3$ ,  $\log 16 = 4$ ,  $\log 32 = 5$

# Generalizing the Analysis

“floor”

- *Some notation:*  $\lfloor x \rfloor$  means round  $x$  down, so  $\lfloor 2.5 \rfloor = 2$
- Binary search of  $n$  elements will do at most  $1 + \lfloor \log_2 n \rfloor$  comparisons  
 $1 + \lfloor \log_2 8 \rfloor = 1 + \lfloor \log_2 9 \rfloor = \dots 1 + \lfloor \log_2 15 \rfloor = 4$   
 $1 + \lfloor \log_2 16 \rfloor = 1 + \lfloor \log_2 17 \rfloor = \dots 1 + \lfloor \log_2 31 \rfloor = 5$
- Why? We can split search region in half  $1 + \lfloor \log_2 n \rfloor$  times before it becomes empty.
- "Big O" notation: we ignore the "1 +" and the floor function. **We say Binary Search has complexity  $O(\log n)$ .**

# PS5

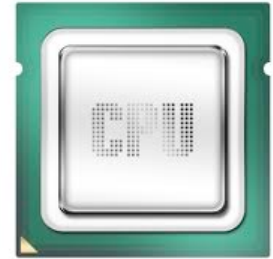
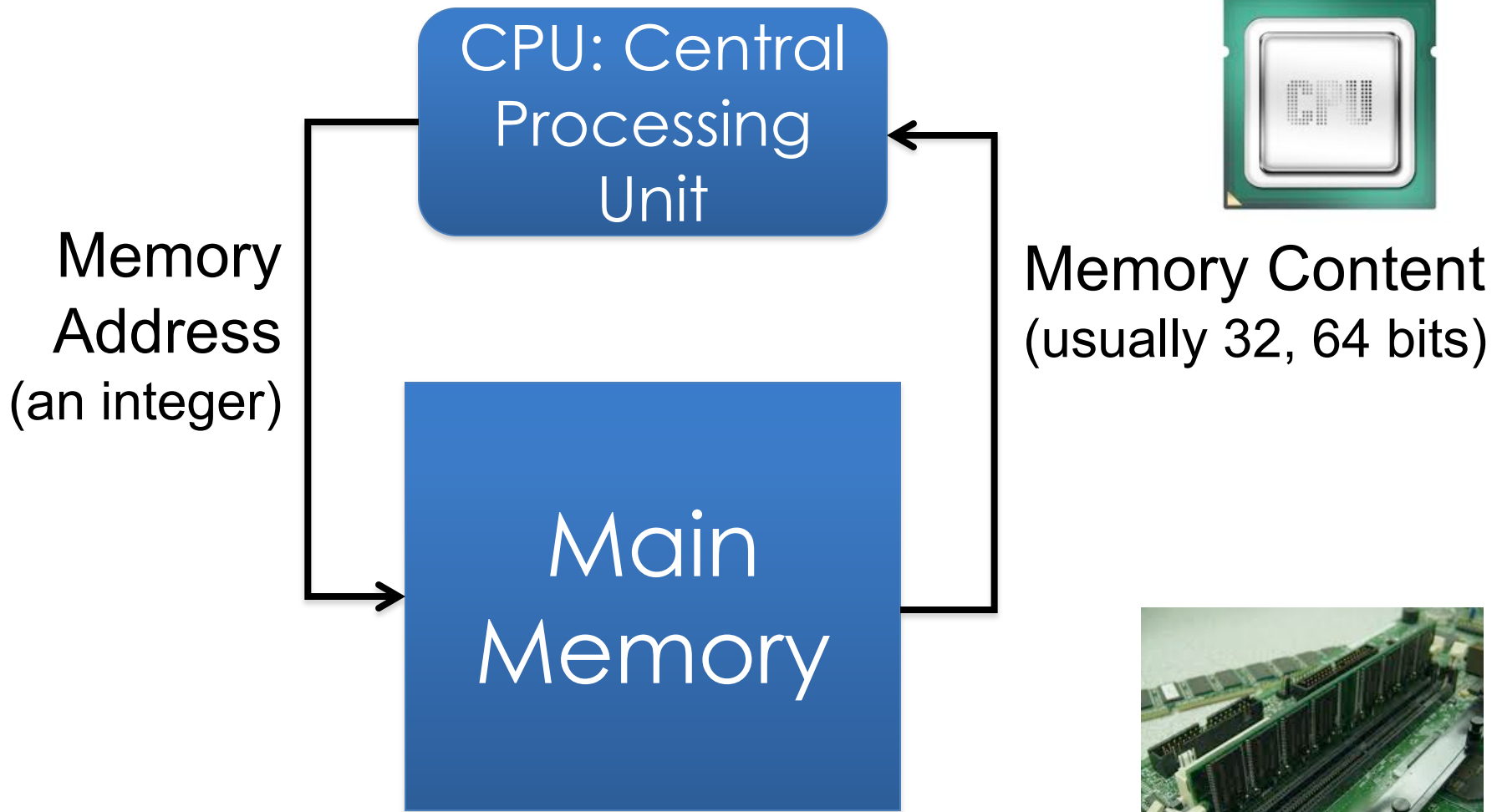
- Q5b
- For the best performance of Quicksort, would we rather have the two sublists of equal length, or would we rather have one be very short and the other very long? Explain briefly.



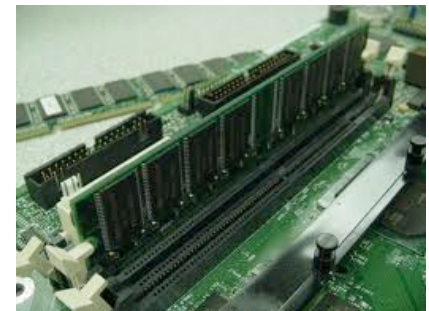


# Organizing Data

# Computer Memory

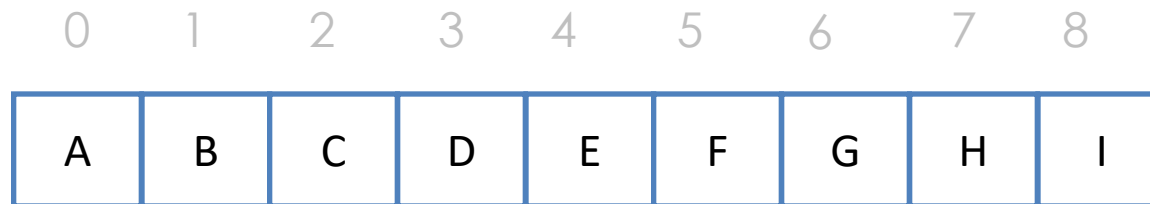


Memory Content  
(usually 32, 64 bits)



# Recall Lists

- Ordered **collection** of data
- Our mental model is based on indexed data slots



- But how are lists actually stored in computer's memory?

# Organizing Data in Memory

- We are going to see in future lectures how data types such as integers, strings are represented in computer memory as sequence of bits (0s, 1s).
- Today we will work at a higher-level of abstraction, and discuss organizing collections of data in memory.
- For example, how are Python lists organized in memory?
- How could we organize our data to capture hierarchical relationships between data?

# Data Structure

- The organization of data is a very important issue for computation.
- A **data structure** is a way of storing data in a computer so that it can be used efficiently.
  - Choosing the right data structure will allow us to develop certain algorithms for that data that are more efficient.

# Today's Lecture

- Two basic structures for ordered sequences:
  - arrays
  - linked lists

# Arrays in Memory

- An **array** is a very simple data structure for holding a sequence of data. They have a direct correspondence with memory system in most computers.
- Typically, array elements are stored in adjacent memory cells. The subscript (or index) is used to calculate an offset to find the desired element.

Address	Content
100:	50
104:	42
108:	85
112:	71
116:	99

Example: data = [50, 42, 85, 71, 99]  
Assume we have a byte-addressable computer, integers are stored using 4 bytes (32 bits) and the first element is stored at address 100. Nothing special about 100, just an example. The array could start at any address.

# Arrays in Memory

- Example: `data = [50, 42, 85, 71, 99]`  
Assume we have a byte-addressable computer, integers are stored using 4 bytes (32 bits) and our array starts at address 100.
- If we want `data[3]`, the computer takes the address of the start of the array (100 in our example) and adds **the index \* the size** of an array element (4 bytes in our example) to find the element we want.

Location of `data[3]` is  $100 + 3 * 4 = 112$

- Do you see why it makes sense for the first index of an array to be 0?

	Content
100:	50
104:	42
108:	85
112:	71
116:	99



# Arrays: Pros and Cons

- Pros:

- Access to an array element is fast since we can compute its location quickly (constant time).

- Cons:

- If we want to insert or delete an element, we have to shift subsequent elements which slows our computation down.
- We need a large enough block of memory to hold our array.

# Arrays in Python

- Array module
- Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained.
- We only use Python lists in 15-110. Python lists are akin to structures called dynamic arrays.

# Linked Lists

- Another data structure that stores a sequence of data values is the **linked list**.
- Data values in a linked list do not have to be stored in adjacent memory cells.
- To accommodate this feature, each data value has an additional “pointer” that indicates where the next data value is in computer memory.
- In order to use the linked list, we only need to know where the first data value is stored.

# Linked List Example

- Linked list to store the sequence: data = [50, 42, 85, 71, 99]

Assume each integer and each pointer requires 4 bytes.

Starting Location of List (head)
124

	<b>data</b>	<b>next</b>
100:	42	148
108:	99	0 (null)
116:		
124:	50	100
132:	71	108
140:		
148:	85	132
156:		

# Linked List Example

- To insert a new element, we only need to change a few pointers.
- Example: Insert 20 between 42 and 85

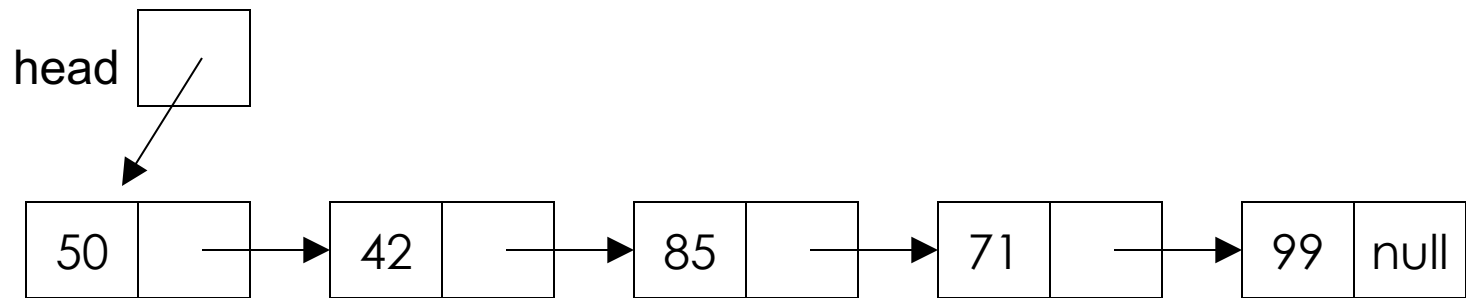
Starting Location of List (head)
124

Assume each integer and pointer requires 4 bytes.

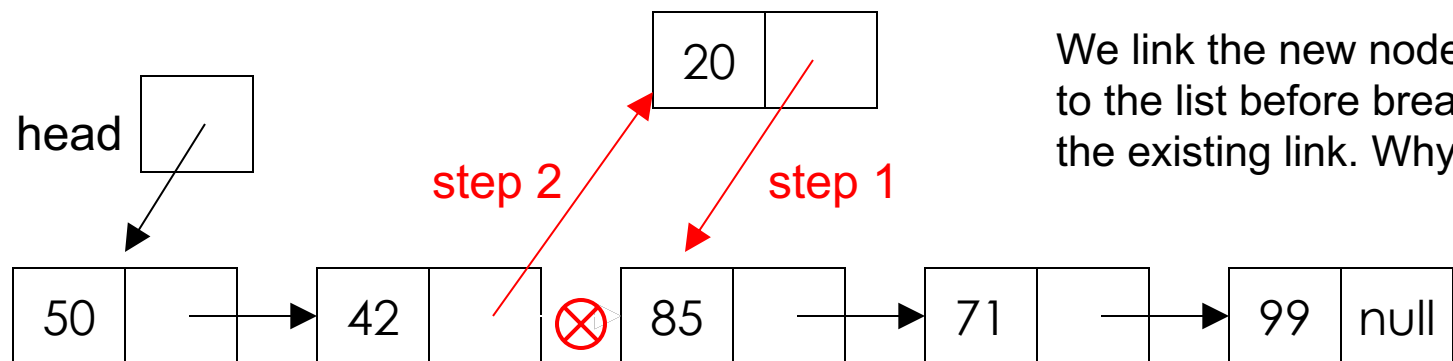
	data	next
100:	42	156
108:	99	0 (null)
116:		
124:	50	100
132:	71	108
140:		
148:	85	132
156:	20	148

# Drawing Linked Lists Abstractly

- [50, 42, 85, 71, 99]



- Inserting 20 after 42:



# Linked Lists: Pros and Cons

## □ Pros:

- Inserting and deleting data does not require us to move/shift subsequent data elements.

## □ Cons:

- If we want to access a specific element, we need to traverse the list from the head of the list to find it, which can take longer than an array access.
- Linked lists require more memory. (Why?)

# Two-dimensional arrays

- Some data can be organized efficiently in a **table** (also called a **matrix** or **2-dimensional array**)

- Each cell is denoted with two subscripts, a row and column indicator

$$B[2][3] = 50$$

B	0	1	2	3	4
0	3	18	43	49	65
1	14	30	32	53	75
2	9	28	38	50	73
3	10	24	37	58	62
4	7	19	40	46	66



# 2D Lists in Python

```
data = [ [1, 2, 3, 4],  
         [5, 6, 7, 8],  
         [9, 10, 11, 12]  
        ]
```

```
>>> data[0]
```

```
[1, 2, 3, 4]
```

```
>>> data[1][2]
```

```
7
```

```
>>> data[2][5] index error
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

# 2D List Example in Python

- Find the sum of all elements in a 2D array

```
def matrix_summation(table):
```

```
    sum_so_far = 0
```

```
    for row in range(0, len(table)):
```

```
        for col in range(0, len(table[row])):
```

```
            sum_so_far = sum_so_far + table[row][col]
```

```
    return sum_so_far
```

number of rows in the table

Number of columns in the given row of the table

In a rectangular matrix, this number will be fixed so we could use a fixed number for row such as `len(table[0])`

# Trace the Nested Loop

```
def matrix_summation(table):  
    sum_so_far = 0  
    for row in range(0, len(table)):  
        for col in range(0, len(table[row])):  
            sum_so_far += table[row][col]  
    return sum_so_far
```

row          col          sum

0            0            1

...

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

```
data = [ [1, 2, 3, 4],  
         [5, 6, 7, 8],  
         [9, 10, 11, 12]  
       ]  
matrix_summation(data)
```

**len(table) = 3**

**len(table[row]) = 4 for every row**

# Tracing the Nested Loop

```
def matrix_summation(table):  
    sum_so_far = 0  
    for row in range(0, len(table)):  
        for col in range(0, len(table[row])):  
            sum_so_far += table[row][col]  
    return sum_so_far
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

`len(table) = 3`

`len(table[row]) = 4 for every row`

row	col	sum
0	0	1
0	1	3
0	2	6
0	3	10
1	0	15
1	1	21
1	2	28
1	3	36
2	0	45
2	1	55
2	2	66
2	3	78

# Recall Arrays and Linked Lists

	<b>Advantages</b>	<b>Disadvantages</b>
<b>Arrays</b>	Constant-time lookup (search) if you know the index	Requires a contiguous block of memory
<b>Linked Lists</b>	Flexible memory usage	Linear-time lookup (search)

How can we exploit the advantages of arrays and linked lists to improve search time in dynamic data sets?

# Hash Tables

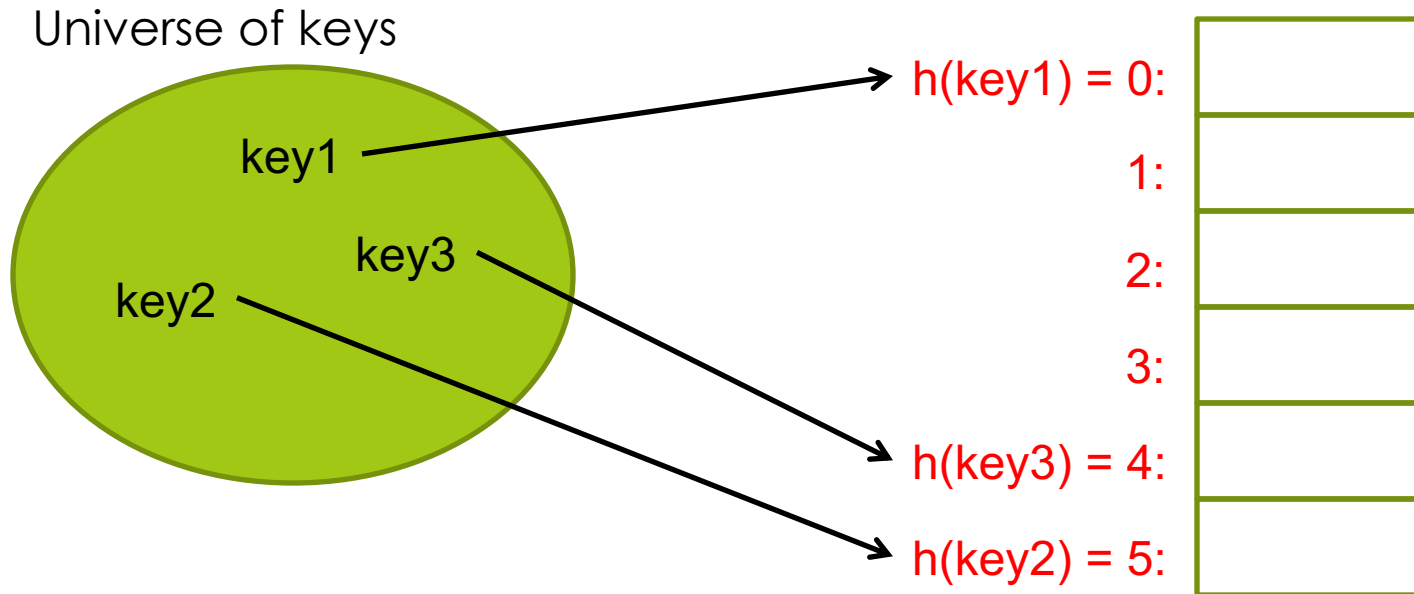
# Key-Value Pairs

- Many applications require a flexible way to look up information. For example, we may want to retrieve information (value) about employees based on their SSN (key).
- Associative arrays: collection of (key, value pairs)
  - Key-value pair examples: name-phone number, username-password pairs, zipcode-shipping costs
- If we could represent the key as an integer and store all data in an array we would have constant look up time. Can we always do that?

No, memory is a bounded resource.

# Hashing

- A “hash function”  $h(\text{key})$  that maps a key to an array index in  $0..k-1$ .
- To search the array `table` for that key, look in `table[h(key)]`



A hash function  $h$  is used to map keys to hash-table (array) slots. Table is an array bounded in size. The size of the universe for keys may be larger than the array size. We call the table slots buckets.



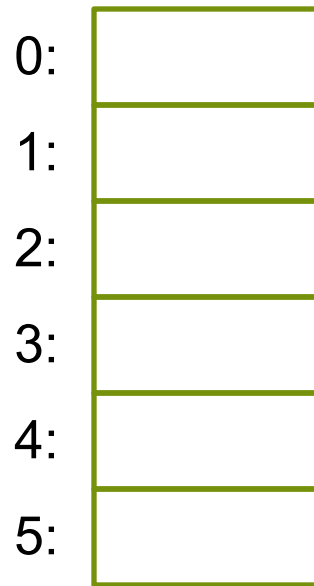
# Example: Hash function

- Suppose we have (key,value) pairs where the key is a string such as (name, phone number) pairs and we want to store these key value pairs in an array.
- We could pick the array position where each string is stored based on the first letter of the string using this hash function:

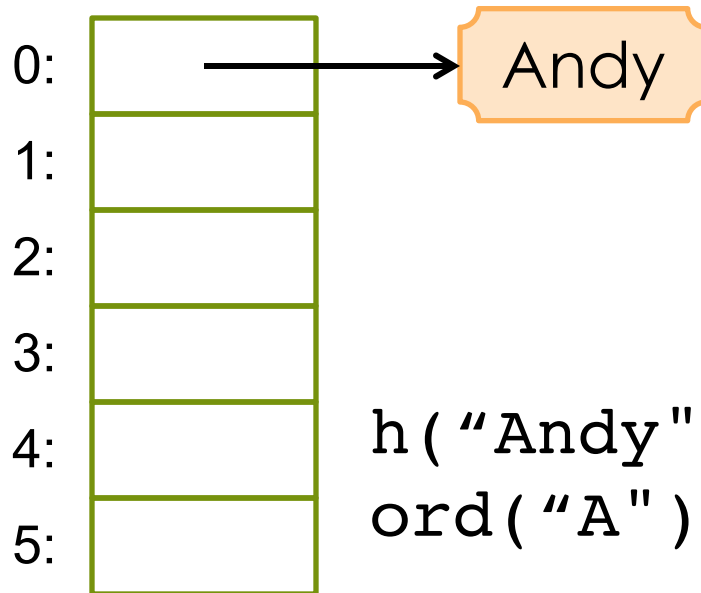
```
def h(str):  
    return (ord(str[0]) - 65) % 6
```

Note  $\text{ord}('A') = 65$

# An Empty Hash Table



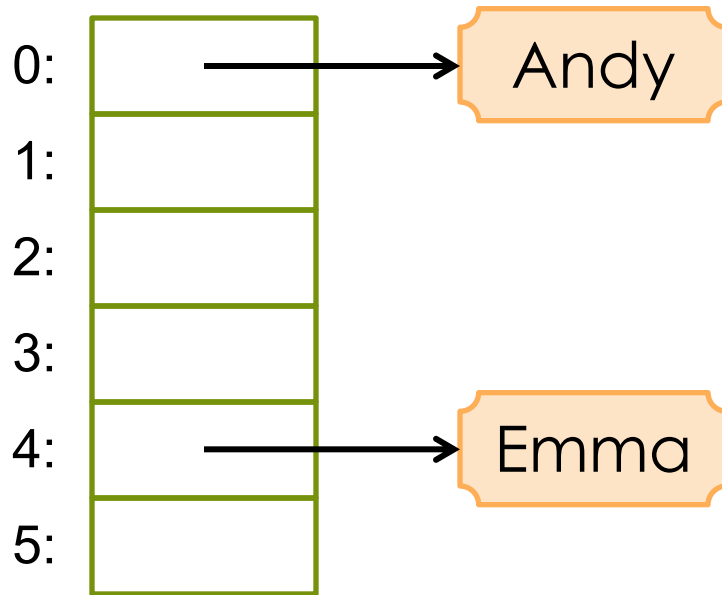
# Add Element "Andy"



$h("Andy")$  is 0 because  
 $\text{ord}("A") = 65$  and  $(65-65) \% 6 = 0$ .

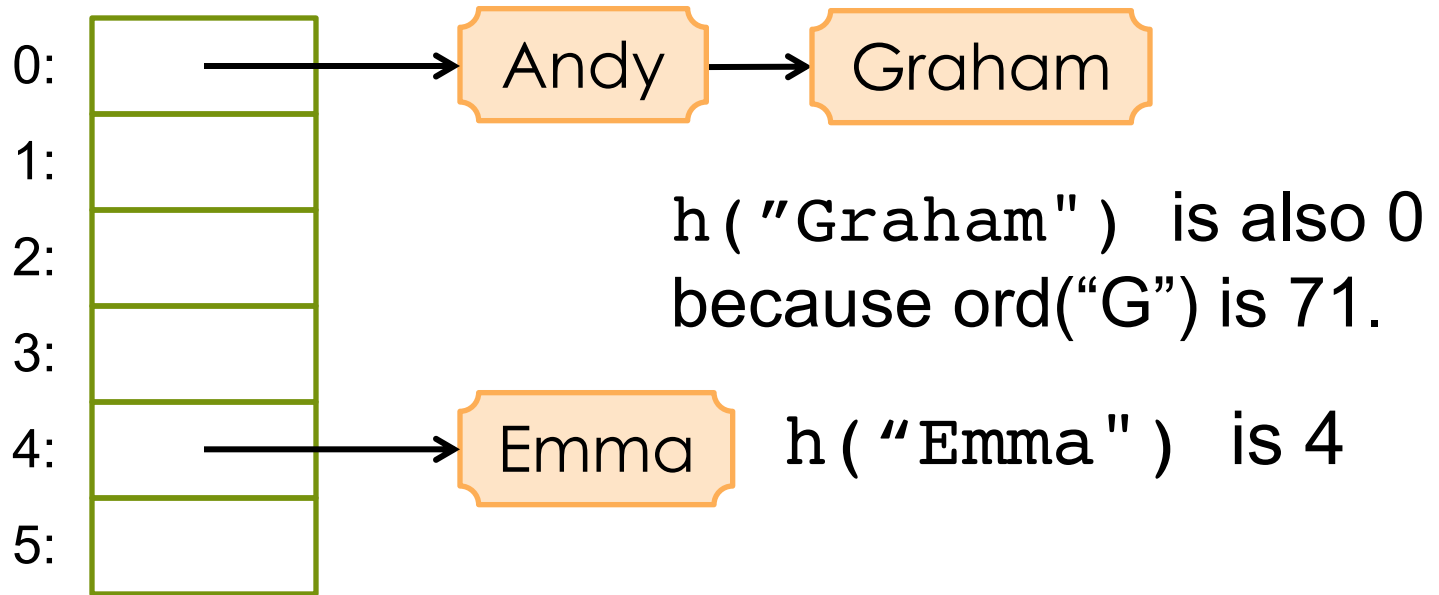
Suppose we use the function  $h$  from the previous slide.

# Add Element "Emma"



$h("Emma")$  is 4 because  $\text{ord}("E") = 69$  and  $(69 - 65) \% 6 = 4$ .

# Add Element "Graham"



In order to add Graham's information to the table we had to form a link list for bucket 0.

# Collisions

- “Andy” and “Graham” end up in the same bucket.
- These are collisions in a hash table.
- Why do we want to minimize collisions?

# Minimizing Collisions

- The more collisions you have in a bucket, the more you have to search in the bucket to find the desired element.
- We want to try to minimize the collisions by creating a hash function that distribute the keys (strings) into different buckets as evenly as possible.

# Requirements for the Hash Function $h(x)$

- Must be fast:  $O(1)$
- Must distribute items roughly uniformly throughout the array, so everything doesn't end up in the same bucket.



# What's A Good Hash Function?

- For strings:
  - Treat the characters in the string like digits in a base-256 number.
  - Divide this quantity by the number of buckets,  $k$ .
  - Take the remainder, which will be an integer in the range  $0..k-1$ .

# Fancier Hash Functions

- How would you hash an integer  $i$ ?
  - Perhaps  $i \% k$  would work well.
- How would you hash a list?
  - Sum the hashes of the list elements.
- How would you hash a floating point number?
  - Maybe look at its binary representation and treat that as an integer?

# Efficiency

- If the keys (strings) are distributed well throughout the table, then each bucket will only have a few keys and the search should take  $O(1)$  time.
- Example:  
If we have a table of size 1000 and we hash 4000 keys into the table and each bucket has approximately the same number of keys (approx. 4), then a search will only require us to look at approx. 4 keys  $\Rightarrow O(1)$
- But, the distribution of keys is dependent on the keys and the hash function we use!

# Summary of Search Techniques

Technique	Setup Cost	Search Cost
Linear search	0, since we're given the list	$O(n)$
Binary search	$O(n \log n)$ to sort the list	$O(\log n)$
Hash table	$O(n)$ to fill the buckets	$O(1)$

# Associative Arrays

- Hashing is a method for implementing associative arrays. Some languages such as Python have associative arrays (**mapping** between keys and values) as a built-in data type.
- Examples:
  - Name in contacts list => Phone number
  - User name => Password
  - Product => Price

# Dictionary Type in Python

This example maps car brands (*keys*) to prices (*values*).

```
>>> cars = {"Mercedes": 55000,  
            "Bentley": 120000,  
            "BMW": 90000}
```

```
>>> cars["Mercedes"]
```

```
55000
```

Keys can be of any **immutable** data type.

Dictionaries are implemented using hashing.

# Iteration over a Dictionary

```
>>> for i in cars:  
    print(i)
```

```
BMW  
Mercedes  
Bentley
```

Think what the loop variables are bound to in each case.

```
>>> for i in cars.items():  
    print(i)
```

```
("BMW", 90000)  
("Mercedes", 55000)  
  
("Bentley", 120000)
```

Note also that there is no notion of ordering in dictionaries. There is no such thing as the first element, second element of a dictionary.

```
>>> for k,v in cars.items():  
    print(k, ":", v )
```

```
BMW : 90000  
Mercedes 55000  
Bentley : 120000
```

# Some Dictionary Operations

- ▣ `d[key] = value` -- Set `d[key]` to `value`.
- ▣ `del d[key]` -- Remove `d[key]` from `d`. Raises a an error if `key` is not in the map.
- ▣ `key in d` -- Return `True` if `d` has a key `key`, else `False`.
- ▣ `items()` -- Return a new view of the dictionary's items ((`key`, `value`) pairs).
- ▣ `keys()` -- Return a new view of the dictionary's keys.
- ▣ `pop(key[, default])` If `key` is in the dictionary, remove it and return its value, else return `default`. If `default` is not given and `key` is not in the dictionary, an error is raised.

Source: <https://docs.python.org/>