CSI 2103: Data Structures

Arrays (Ch 5)

Yonsei University
Spring 2022

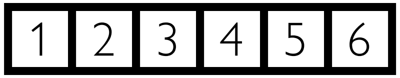
Seong Jae Hwang

Aims

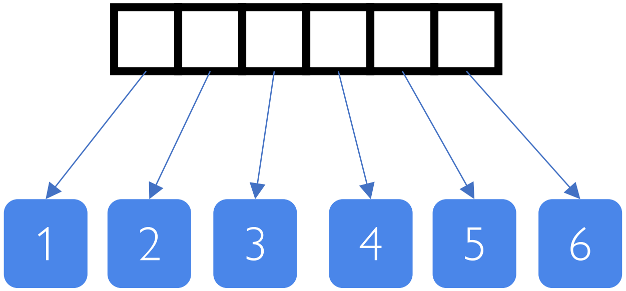
- Array as a data structure
- Basic operations
  - Add (Insert)
  - Remove (Delete)
  - Find (Search)
- Sorted vs. Unsorted
- Drawbacks

Array

- Sequenced collection of variables indexed by consecutive integers
- The definition is…a little confusing when using Python for this topic
- In Python, there's "array" class and "list" class
  - array: similar to Java and C/C++ array; elements are of same types

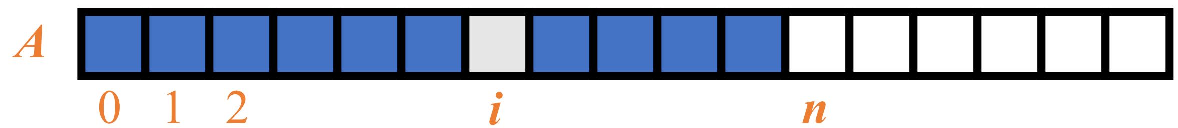


  - list: sequence of references; elements may be of different types
    - used more commonly; dynamically resizes when full



Array

- For this class, we will follow the general notion of "array" (sequence of elements) by just using the "list" class with the same type
  - Sometimes, we will use a Python package "numpy" and its array (in HW 1) which is closer the "array" we want
- Don't think too hard; we will generally refer to a sequence of elements that can be accessed with integer indices as an "array"
- For an array `A` we can
  - Access elements by index: `A[i]`
  - Access the length of array `A` by `len(A)`



Array: Operations

- Three array operations (not primitive operations):
  - Add (Insert)
  - Delete (Remove)
  - Search (Find)
- Not as easy as it sounds
- See the time complexity

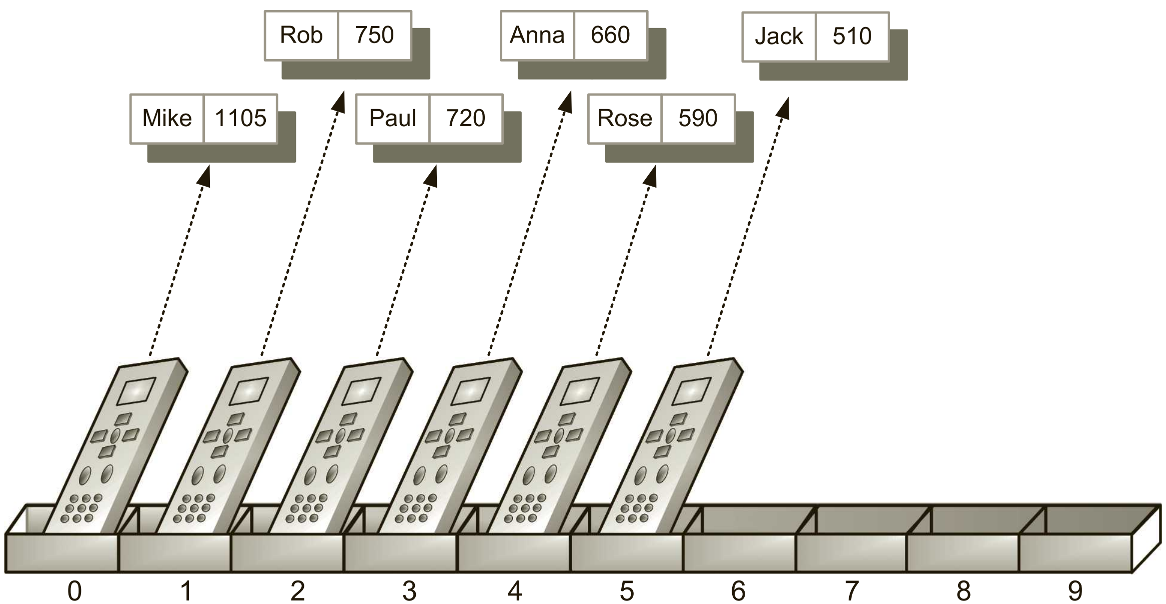
Array

- Store primitive elements

High scores	940	880	830	790	750	660	650	590	510	440
	0	1	2	3	4	5	6	7	8	9

indices

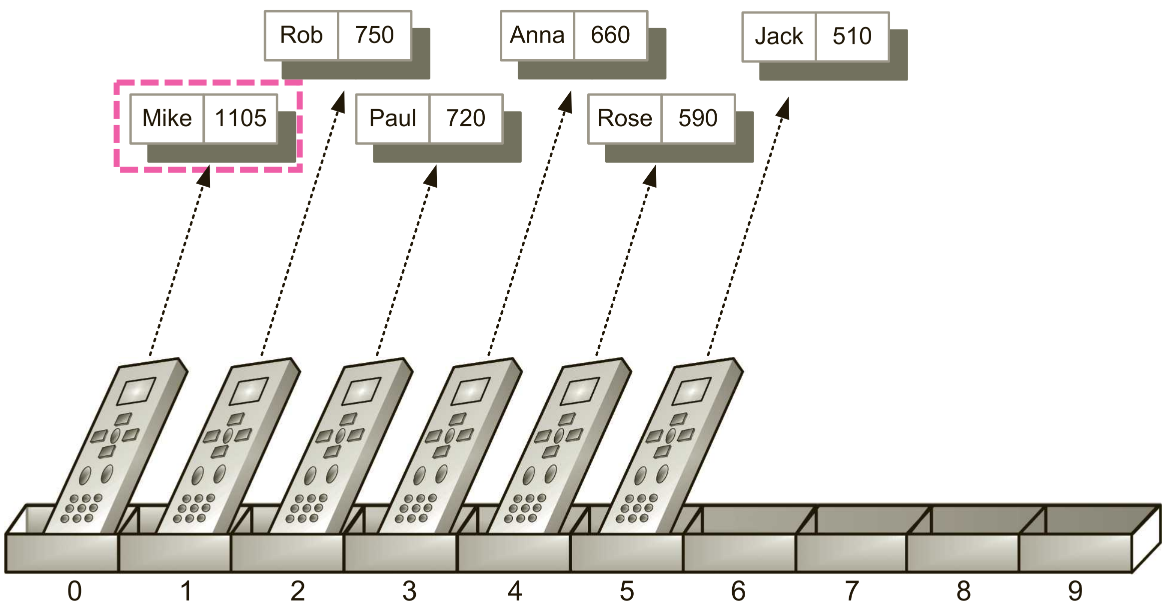
- Store references to objects
  - We will make a `Scoreboard` array storing [Player, Score] information from high to low scores



Python Example: Game Entries

- A game entry stores the player's name and the game score

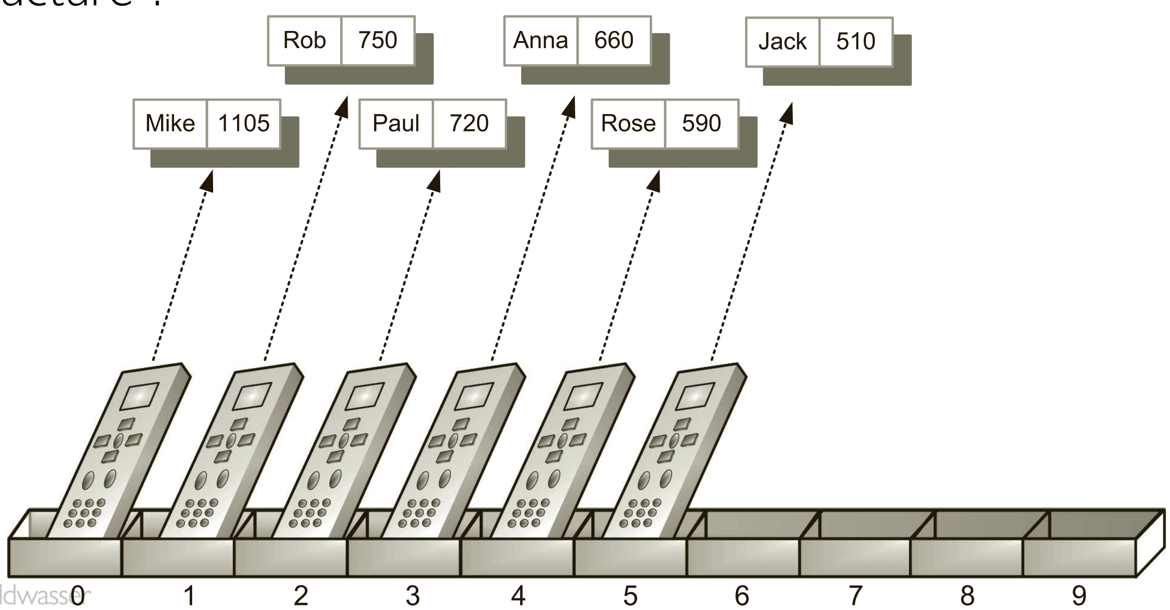
```
class GameEntry:
  """Represents one entry of a list of high scores."""

  def __init__(self, name, score):
    self._name = name
    self._score = score

  def get_name(self):
    return self._name

  def get_score(self):
    return self._score

  def __str__(self):
    return '({0}, {1})'.format(self._name, self._score) # e.g., '(Bob, 98)'
```



Python Example: Scoreboard

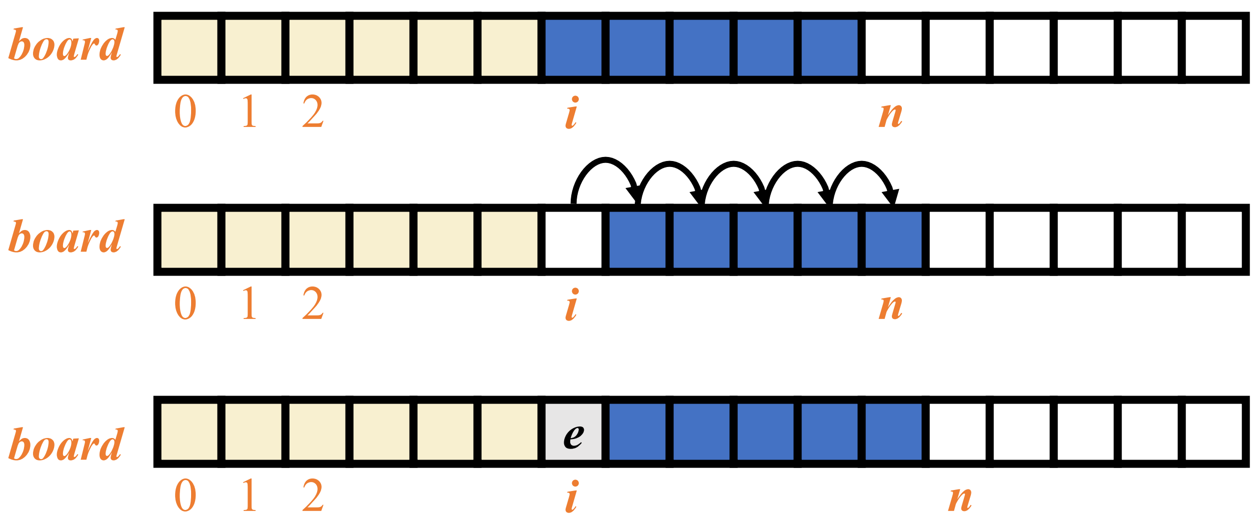
- Keep track of players and their best scores in a `GameEntry` class array named `board`
- `board` is sorted by score (high to low)
- How would we use this "data structure"?

```
class Scoreboard:
  """Fixed-length sequence of high scores in nondecreasing order."""

  def __init__(self, capacity=10):
    """Initialize scoreboard with given maximum capacity.

    All entries are initially None.
    """
    self._board = [None] * capacity        # reserve space for future scores
    self._n = 0                            # number of actual entries
```



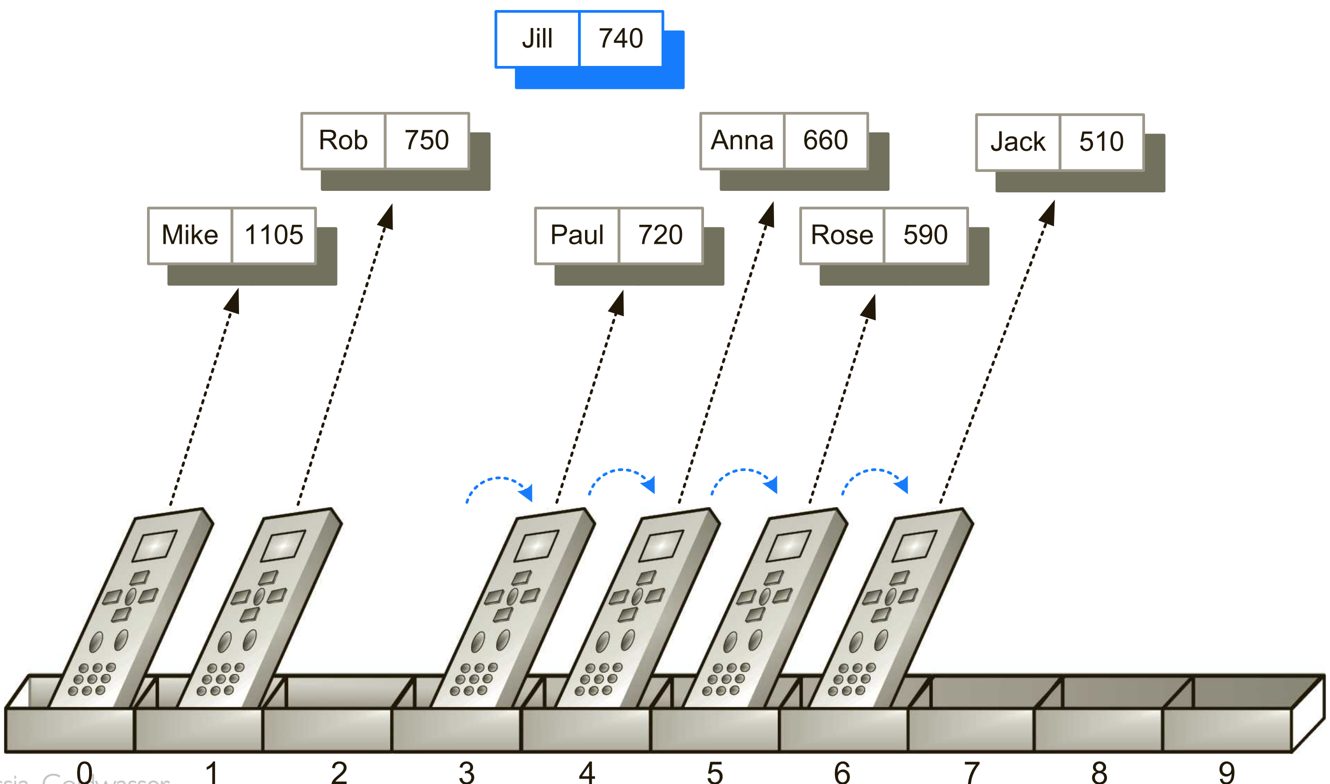
Adding an Entry

- To add an entry `e` in a sorted array `board`:
  - Find the place `i` to add that maintains the sorted order
  - Make room to add by shifting `n-i` entries towards the right
- Time Complexity? $O(n)$



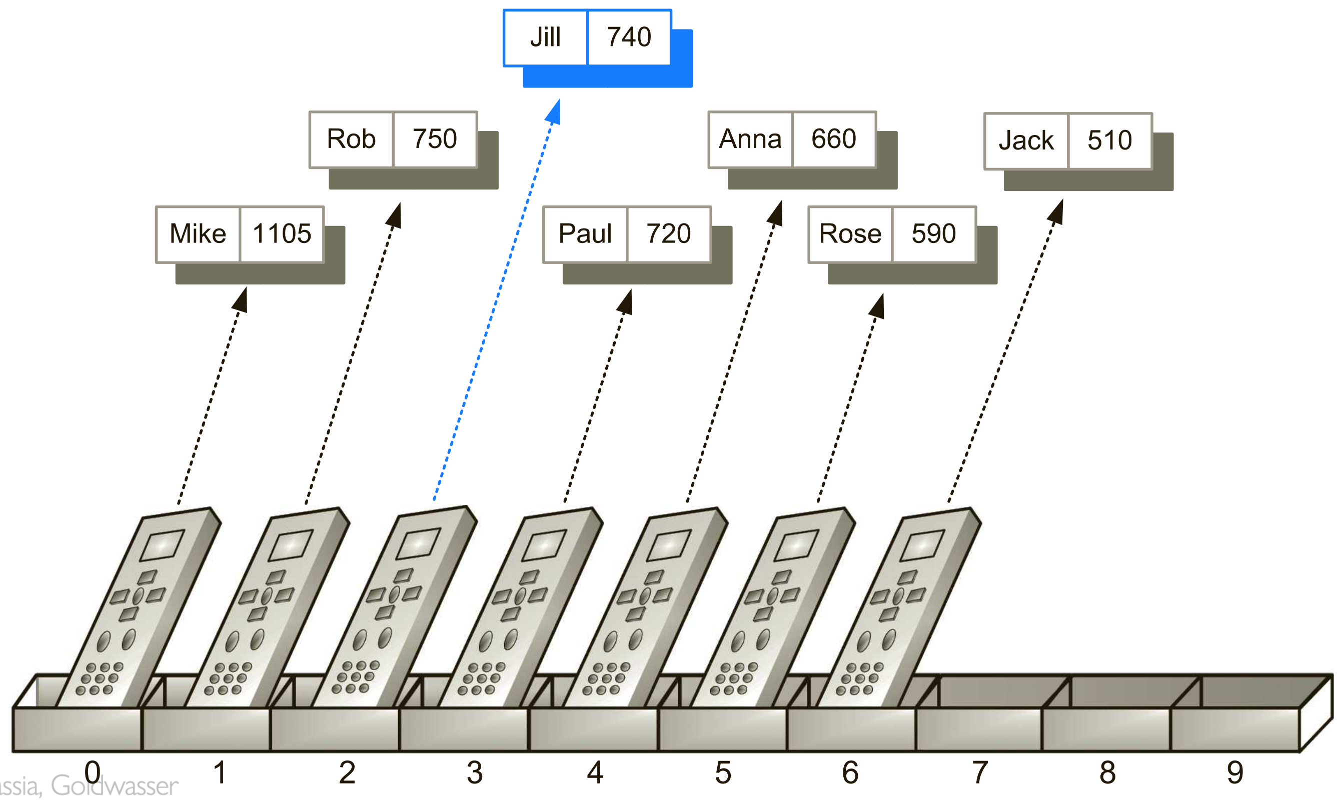
Adding an Entry

- To add an entry `e` in a sorted array `board`:
  - Find the place `i` to add that maintains the sorted order
  - Make room to add by shifting `n-i` entries towards the right
- Time Complexity? $O(n)$



Adding an Entry

- To add an entry `e` in a sorted array `board`:
  - Find the place `i` to add that maintains the sorted order
  - Make room to add by shifting `n-i` entries towards the right
- Time Complexity? $O(n)$



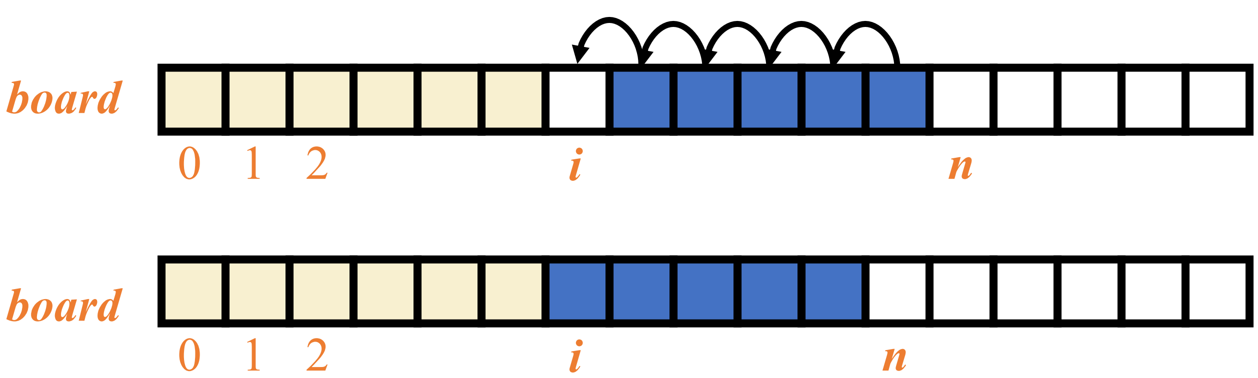
Adding an Entry

- To add an entry `e` in a sorted array `board`:
  - Find the place `i` to add that maintains the sorted order
  - Make room to add by shifting `n-i` entries towards the right
- Time Complexity? $O(n)$ Whenever you see a loop, check!

```python
def add(self, entry):
  """Consider adding entry to high scores."""
  score = entry.get_score()

  # Does new entry qualify as a high score?
  # answer is yes if board not full or score is higher than last entry
  good = self._n < len(self._board) or score > self._board[−1].get_score()

  if good:
    if self._n < len(self._board):          # no score drops from list
      self._n += 1                          # so overall number increases

    # shift lower scores rightward to make room for new entry
    j = self._n − 1
    while j > 0 and self._board[j−1].get_score() < score:
      self._board[j] = self._board[j−1]     # shift entry from j-1 to j
      j −= 1                                # and decrement j
    self._board[j] = entry                  # when done, add new entry
```

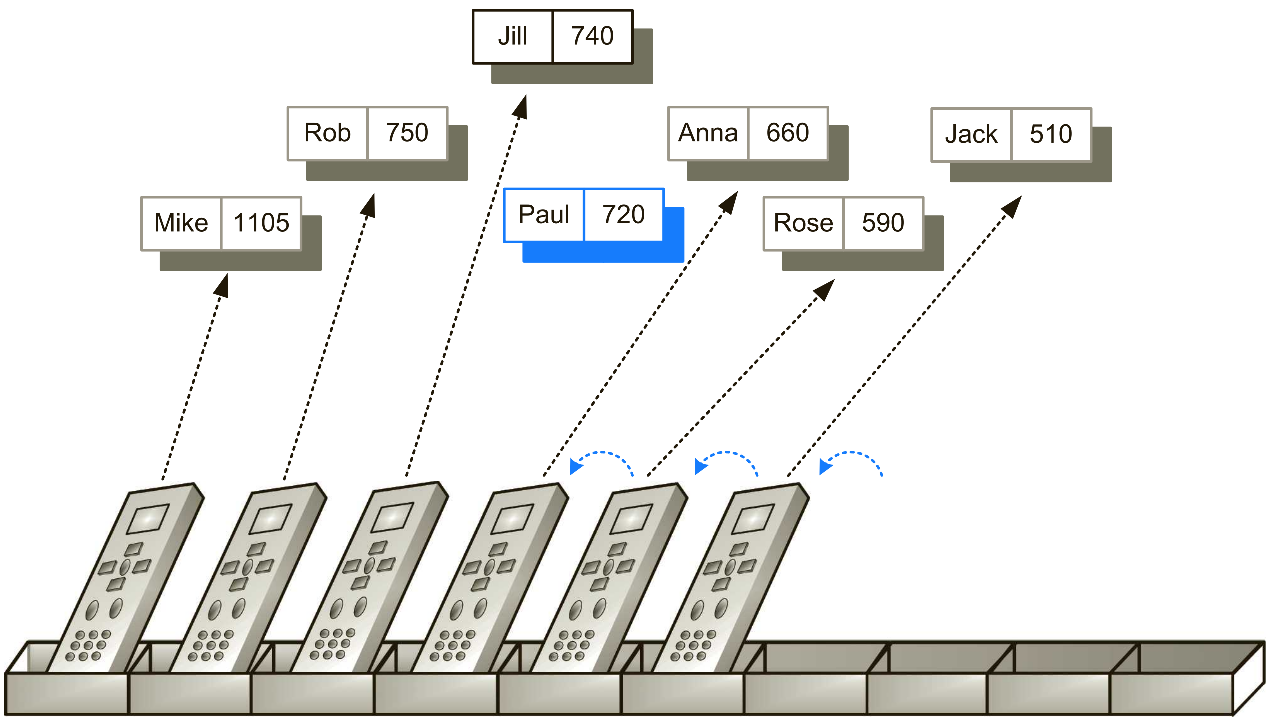
Removing an Entry

- To remove an entry e at index `i`:
  - Remove the entry at `board[i]`
  - Fill the hole by shifting `n-i-1` entries backwards



Removing an Entry

- To remove an entry e at index i:
  - Remove the entry at board[i]
  - Fill the hole by shifting n-i-1 entries backwards
- Time Complexity? $O(n)$



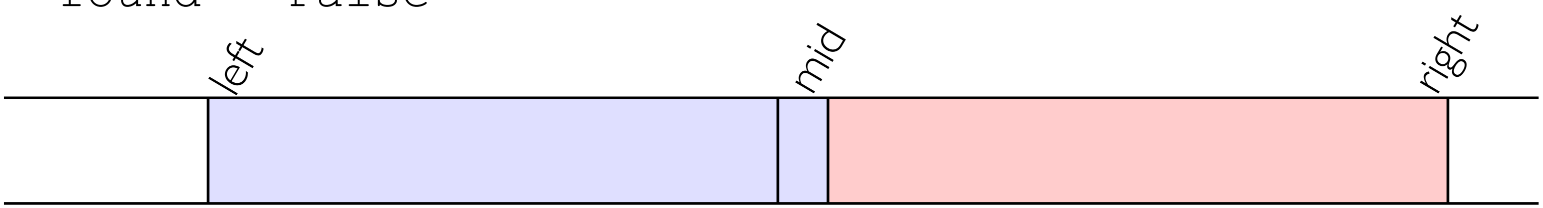
Removing an Entry

- To remove an entry `e` at index `i`:
  - Remove the entry at `board[i]`
  - Fill the hole by shifting `n-i-1` entries backwards
- Time Complexity? $O(n)$
  - There's no comparison since we know the exact index to move
  - But the shifting still has the linear *worst-case* time complexity

Binary Search

- What about finding an entry with a specific value (score)?

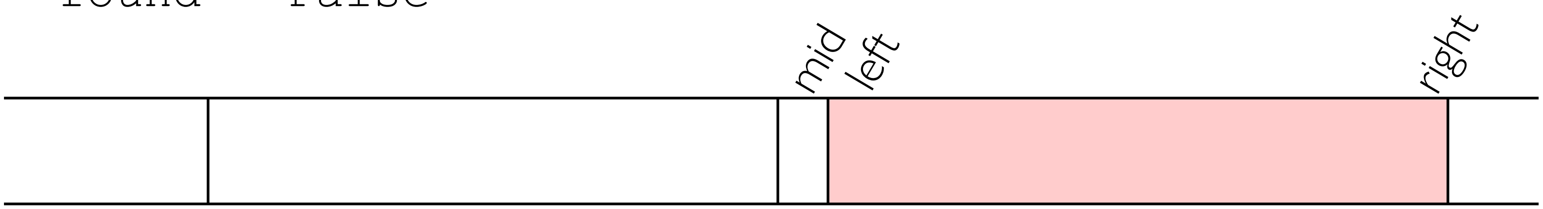
```
left = 0
right = n - 1
while left < right:
  mid = (left + right) / 2
  if a[mid] < x:
    left = mid + 1
  else:
    right = mid
if (left == right) and (a[left] == x):
  found = True
  foundpos = left
else:
  found = False
```



Binary Search

- `x` has to be in the right half

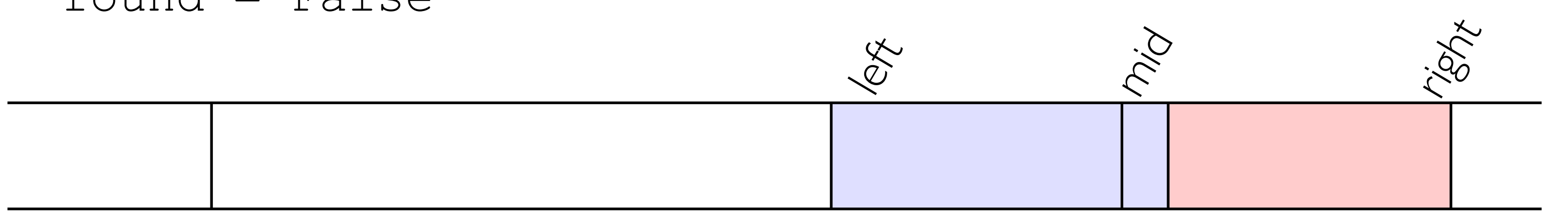
```
left = 0
right = n - 1
while left < right:
  mid = (left + right) / 2
  if a[mid] < x:
    left = mid + 1
  else:
    right = mid
if (left == right) and (a[left] == x):
  found = True
  foundpos = left
else:
  found = False
```



Binary Search

- Find the new `mid`

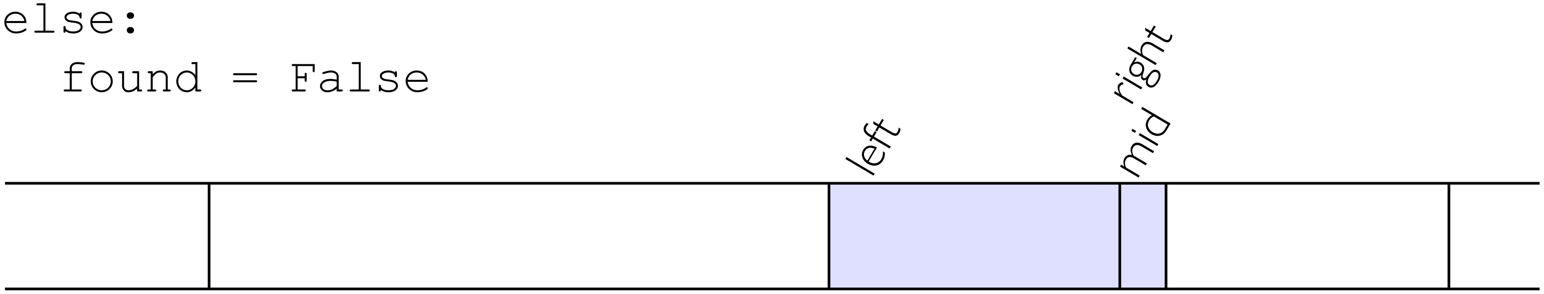
```
left = 0
right = n - 1
while left < right:
  mid = (left + right) / 2
  if a[mid] < x:
    left = mid + 1
  else:
    right = mid
if (left == right) and (a[left] == x):
  found = True
  foundpos = left
else:
  found = False
```



Binary Search

- `x` has to be in the left half

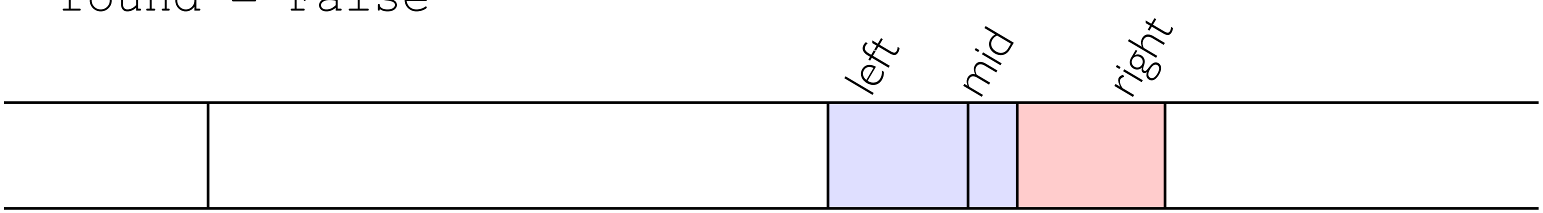
```
left = 0
right = n - 1
while left < right:
  mid = (left + right) / 2
  if a[mid] < x:
    left = mid + 1
  else:
    right = mid
if (left == right) and (a[left] == x):
  found = True
  foundpos = left
else:
  found = False
```



Binary Search

- Find the new `mid`, and repeat until `left` and `right` "meet"

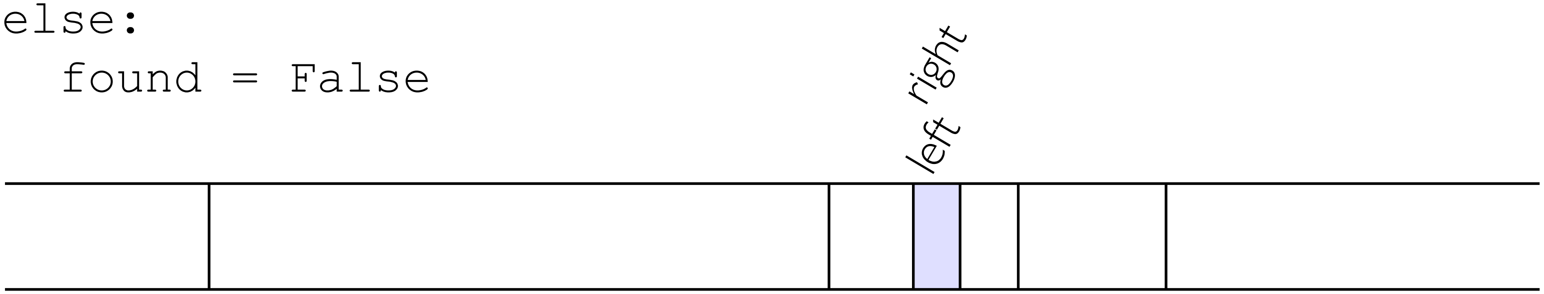
```
left = 0
right = n - 1
while left < right:
  mid = (left + right) / 2
  if a[mid] < x:
    left = mid + 1
  else:
    right = mid
if (left == right) and (a[left] == x):
  found = True
  foundpos = left
else:
  found = False
```



Binary Search

- What is the time complexity? $O(\log_2 n) = O(\log n)$

```
left = 0
right = n - 1
while left < right:
  mid = (left + right) / 2
  if a[mid] < x:
    left = mid + 1
  else:
    right = mid
if (left == right) and (a[left] == x):
  found = True
  foundpos = left
else:
  found = False
```



Time Complexity

- Sorted Array
  - Find (Search): $O(\log n)$ with binary search
  - Add (Insertion): $O(n)$
  - Remove (Deletion): $O(n)$
  - *Array needs to be sorted at first: best sorting is $O(n \log n)$

- Unsorted Array
  - Find (Search): $O(n)$ with linear search
  - Add (Insertion): $O(1)$ by just adding at an empty index
  - Remove (Deletion): $O(1)$ by just find with index and remove
    - *Avoid shifting by tracking indices?

Trade-offs: Sorted array vs. Unsorted array

- If your application needs to search entries fast: Sorted Array
  - Sorting does not need to happen all the time
  - Will cover various sorting algorithms later
- If your application just needs to add/remove fast: Unsorted Array

- This is why we analyze the time complexity of various algorithms
- Not one data structure is the best at everything!
- Understand the pros/cons and pick the best data structure for your goal

Operation	Sorted Array	Unsorted Array
Search	$O(\log n)$	$O(n)$
Insertion	$O(n)$	$O(1)$
Deletion	$O(n)$	$O(1)$
Sorting	$O(n \log n)$	X

Extra: Multidimensional Array

- When an element of an array is also an array: 2D-array
- Also called a matrix
- Use two indices: typically, i and j
  - row: vertical index
  - column: horizontal index

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

row 4, column 1: data[4][1] is 65

- Ex: This two-dimensional data is essentially a "list of lists"

22	18	709	5	33
45	32	830	120	750
4	880	45	66	61

data = [[22, 18, 709, 5, 33], [45, 32, 830, 120, 750], [4, 880, 45, 66, 61]]

Summary

- Array as a data structure
  - Sorted vs. Unsorted
- Time Complexity
  - Find
  - Add
  - Remove
- One is not always better than the other one: Trade-offs
- Next: Slightly more flexible and general data structure
  - Again, different pros and cons of operations