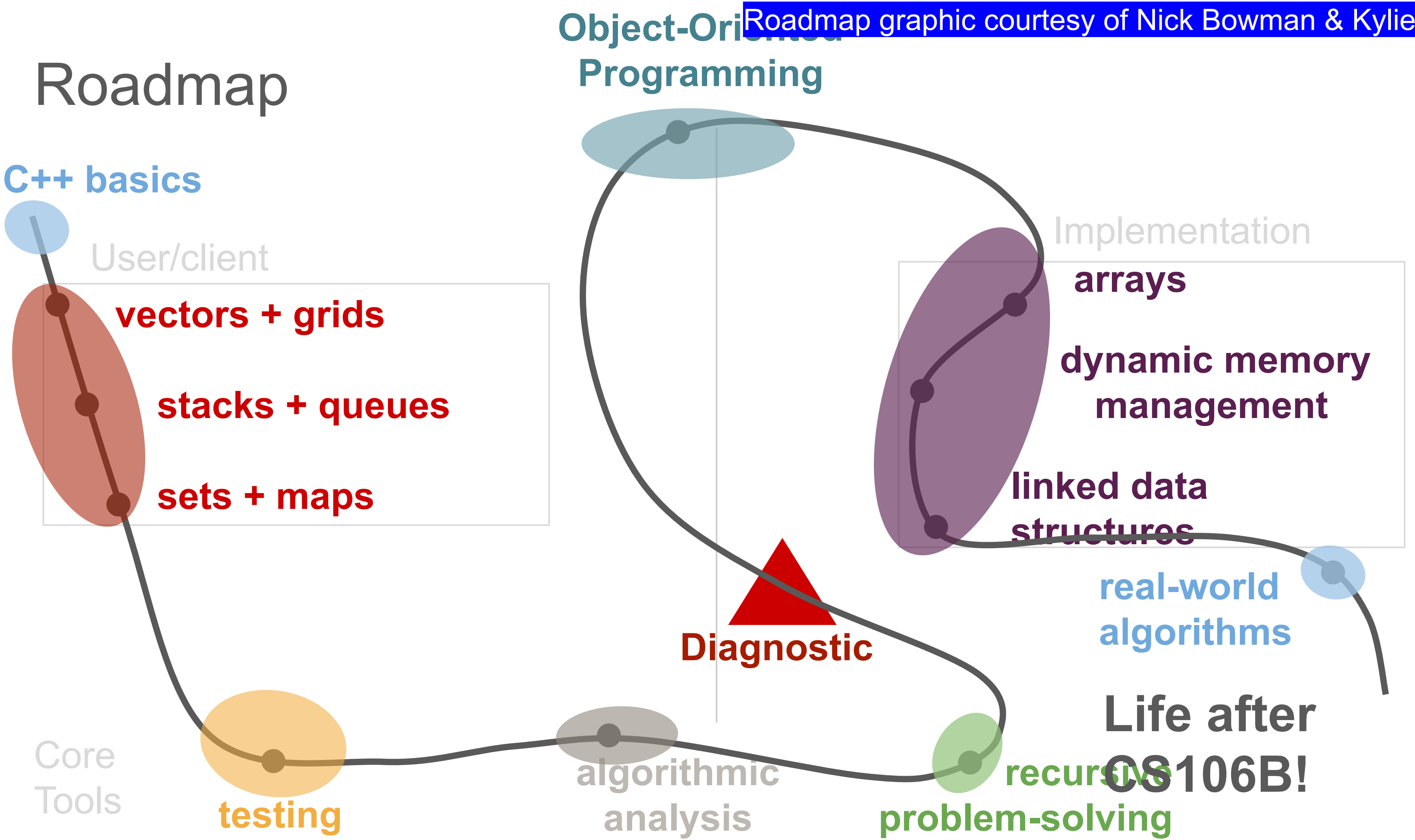
Linked List Wrapup + Introduction to Sorting

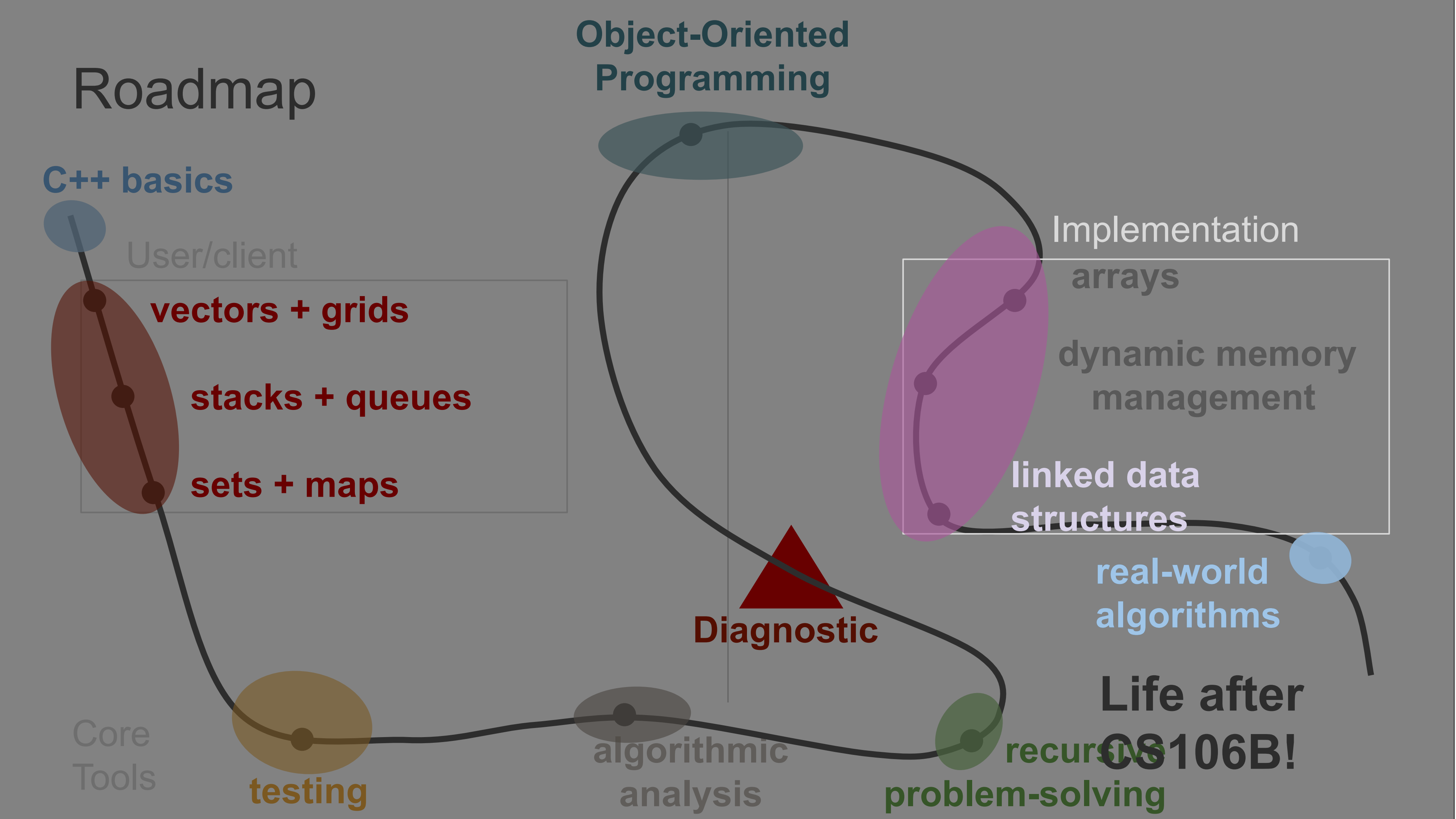
What is a problem/challenge in your everyday life that you feel empowered to solve with CS?

(put your answers the chat)

Roadmap

Roadmap graphic courtesy of Nick Bowman & Kylie Jue





Roadmap
Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today's questions

How can we implement linked list operations that require rewiring multiple elements?

What are real-world algorithms that can be used to organize data?

Today’s topics

1. Review
2. Advanced Linked List Operations
3. Introduction to Sorting Algorithms

Review

[linked list operations]

Yesterday

- Linked lists can be used outside classes - you'll do this on Assignment 5!

- Think about when you want to pass pointers by reference in order to edit the original pointer and to avoid leaking memory.

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Three applications of traversal

- Printing a linked list
- Measuring a linked list
- Freeing a linked list

```
    /* Note: This looks a little different for freeing */
while (list != nullptr) {
    // DO SOMETHING
    list = list->next;
}
```

Linked List Traversal Takeaways

- Temporary pointers into lists are very helpful!
  - When processing linked lists iteratively, it's common to introduce pointers that point to cells in multiple spots in the list.
  - This is particularly useful if we're destroying or rewiring existing lists.

- Using a **while** loop with a condition that checks to see if the current pointer is **nullptr** is the prevailing way to traverse a linked list.

- Iterative traversal offers the most flexible, scalable way to write utility functions that are able to handle all different sizes of linked lists.

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Two ways to add (so far)

- Insertion at the front: **prependTo()**
  - Prepending is faster but results in a reversed order of items (things added earlier are at the back of the list)

- Insertion at the back: **appendTo()**
  - Appending requires traversing all items but maintains order (things added earlier are at the front of the list)

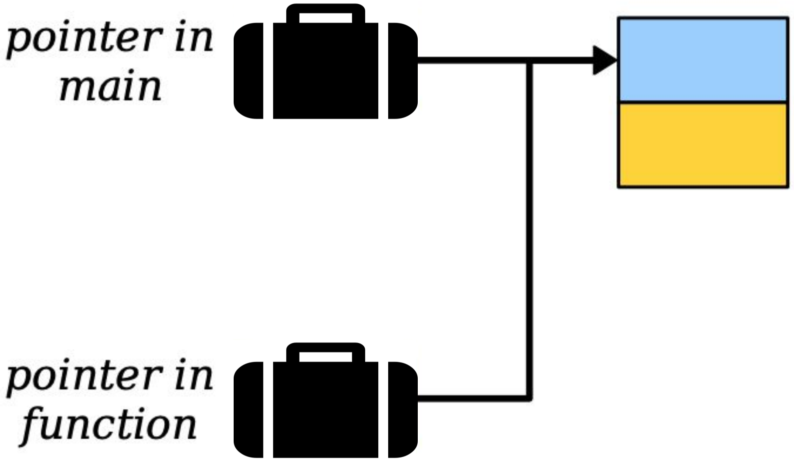
```
void nameOfAddFunction(Node*& list, string data) {
    ...
}
```

Pointers by Value

- Unless specified otherwise, function arguments in C++ are passed by value – this includes pointers!

- A function that takes a pointer as an argument gets a copy of the pointer.

- We can change where the copy points, but not where the original pointer points.



Unresolved Issue

- What is the big-O complexity of appending to the back of a linked list using our algorithm?

Unresolved Issue

- What is the big-O complexity of appending to the back of a linked list using our algorithm?

- **Answer:** `O(n)`, where n is the number of elements in the list, since we have to find the last position each time.

Unresolved Issue

- What is the big-O complexity of appending to the back of a linked list using our algorithm?

- **Answer:** `O(n)`, where n is the number of elements in the list, since we have to find the last position each time.

- This seems suspect – `O(n)` for a single insertion is pretty bad! Can we do better?

More Linked Lists!

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring (by inserting and deleting in the middle of a list!)**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

A more efficient append

A more efficient **appendTo()**

- Yesterday, we saw an O(n) **appendTo()** that added to the back of a linked list. We can do better!

- What if we know we're going to add many things in some maintained order?

- Specifically, we'll use the example of adding items from a vector into linked list.

Attempt #1

```
Node* createListWithAppend(Vector<string> values) {
   if (values.isEmpty()) {
       return nullptr;
   }
   Node* head = new Node(values[0], nullptr);

   for (int i = 1; i < values.size(); i++) {
       appendTo(head, values[i]);
   }
   return head;
}
```

Attempt #1: **What's the runtime? (poll)**

```
Node* createListWithAppend(Vector<string> values) {
  if (values.isEmpty()) {
     return nullptr;
  }
  Node* head = new Node(values[0], nullptr);

  for (int i = 1; i < values.size(); i++) {
     appendTo(head, values[i]);
  }
  return head;
}
```

A. O(N)
B. O(N^2)
C. O(N^3)
D. O(log N)

Attempt #1: **What's the runtime? (poll)**

```
Node* createListWithAppend(Vector<string> values) {
  if (values.isEmpty()) {
      return nullptr;
  }
  Node* head = new Node(values[0], nullptr);

  for (int i = 1; i < values.size(); i++) {
      appendTo(head, values[i]);
  }
  return head;
}
```

A. O(N)
B. $O(N^2)$
C. $O(N^3)$
D. O(log N)

Attempt #2: **append()** – Let's code it!

How does it work?

```
int main() {
    Vector<string> values = {"Nick", "Kylie", "Trip"};
    Node* list = createListWithTailPtr(values);

    /* Do other list-y things here, like printing/freeing the list. */
    return 0;
}
```

```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Nick", "Kylie", "Trip"}

```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

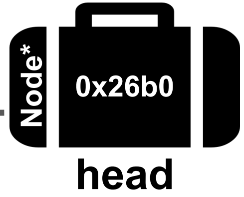
{"Nick", "Kylie", "Trip"}

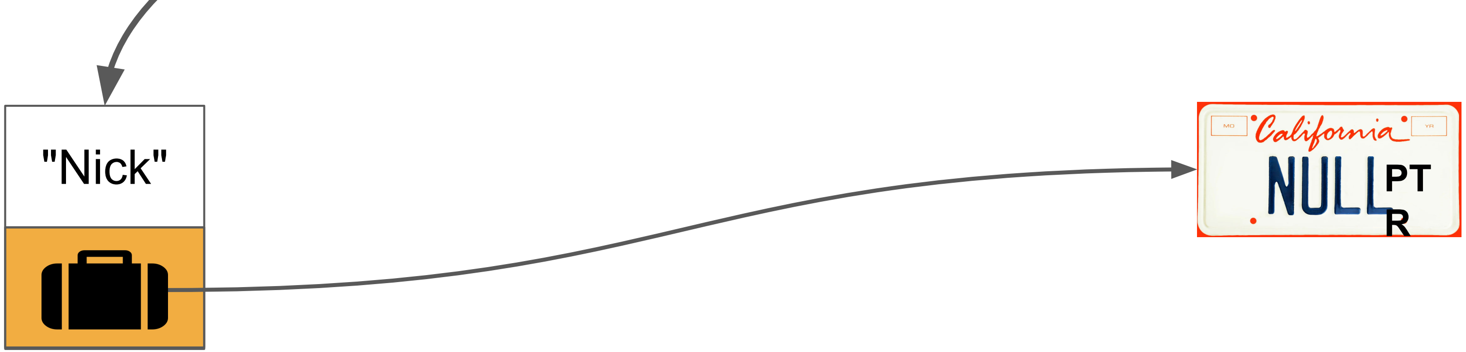
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Nick",
"Kylie",
"Trip"}

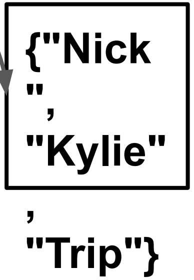
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

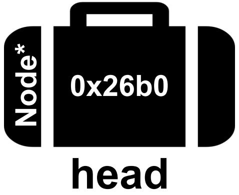
{"Nick", "Kylie", "Trip"}

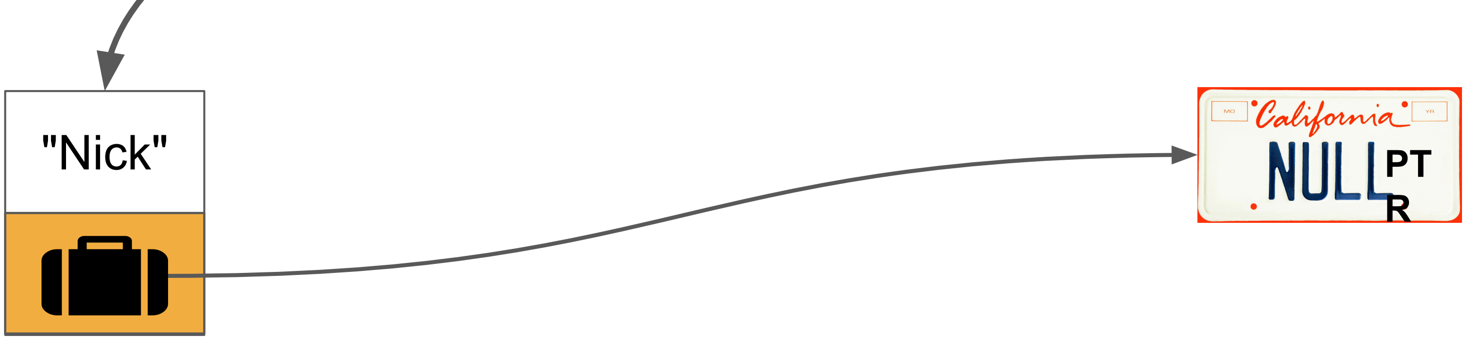




```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

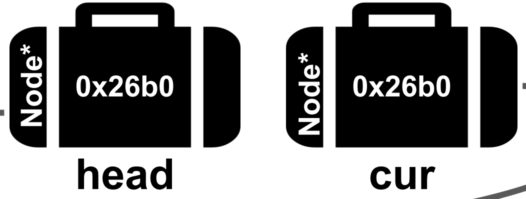






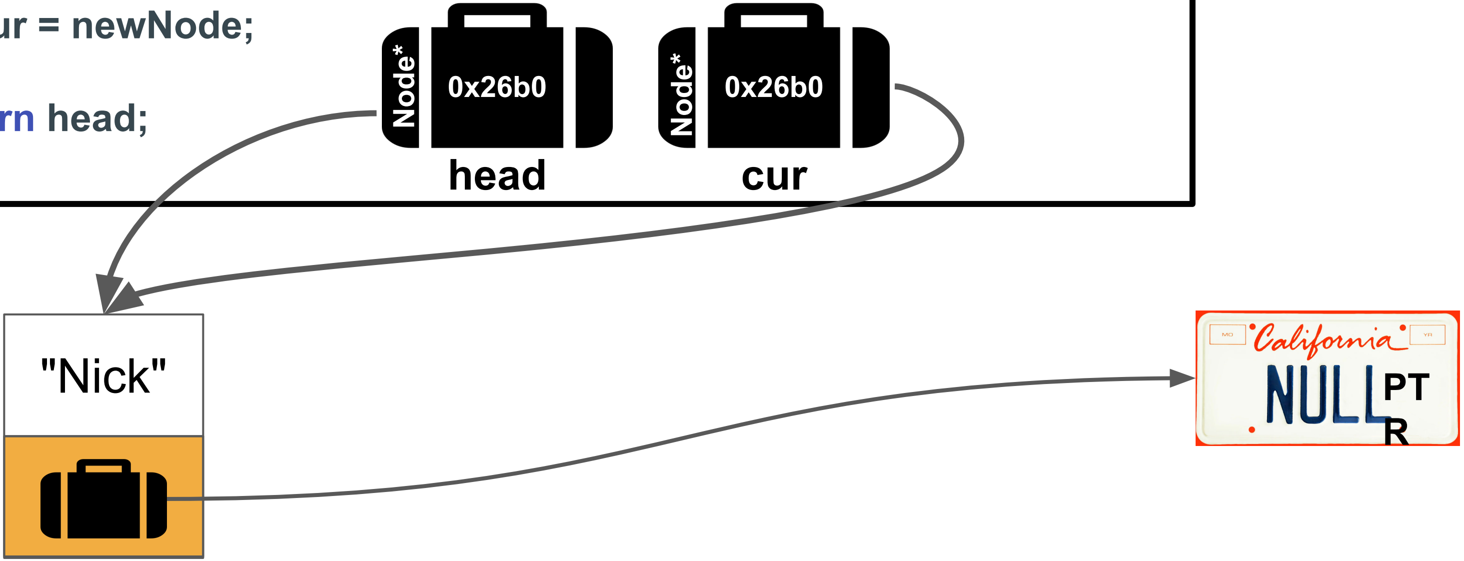
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Nick", "Kylie", "Trip"}



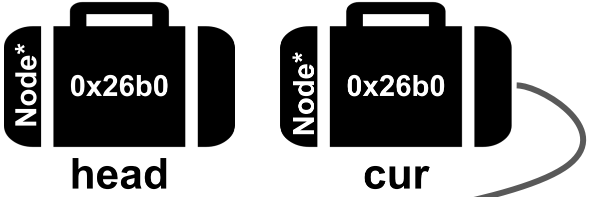
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Nick", "Kylie", "Trip"}



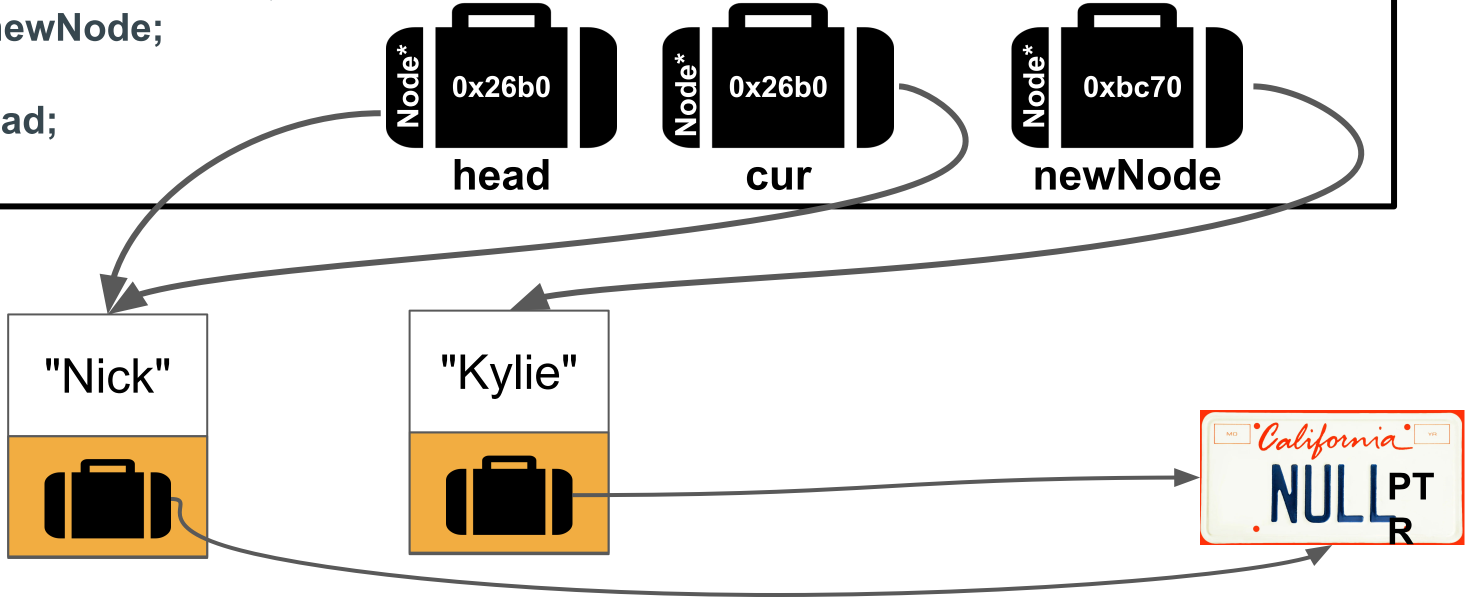
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Nick", "Kylie", "Trip"}

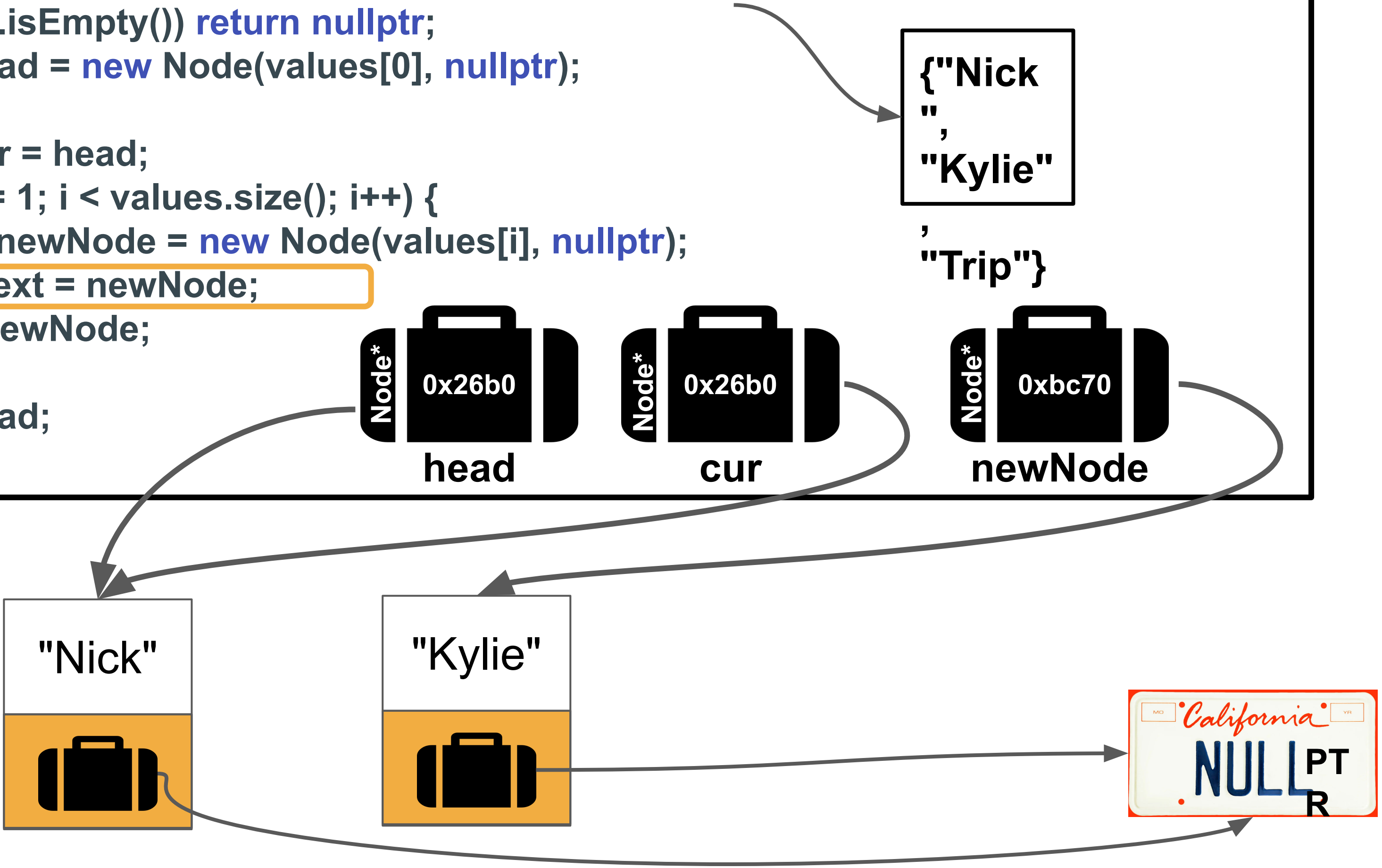


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

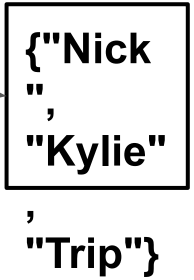
{"Nick", "Kylie", "Trip"}

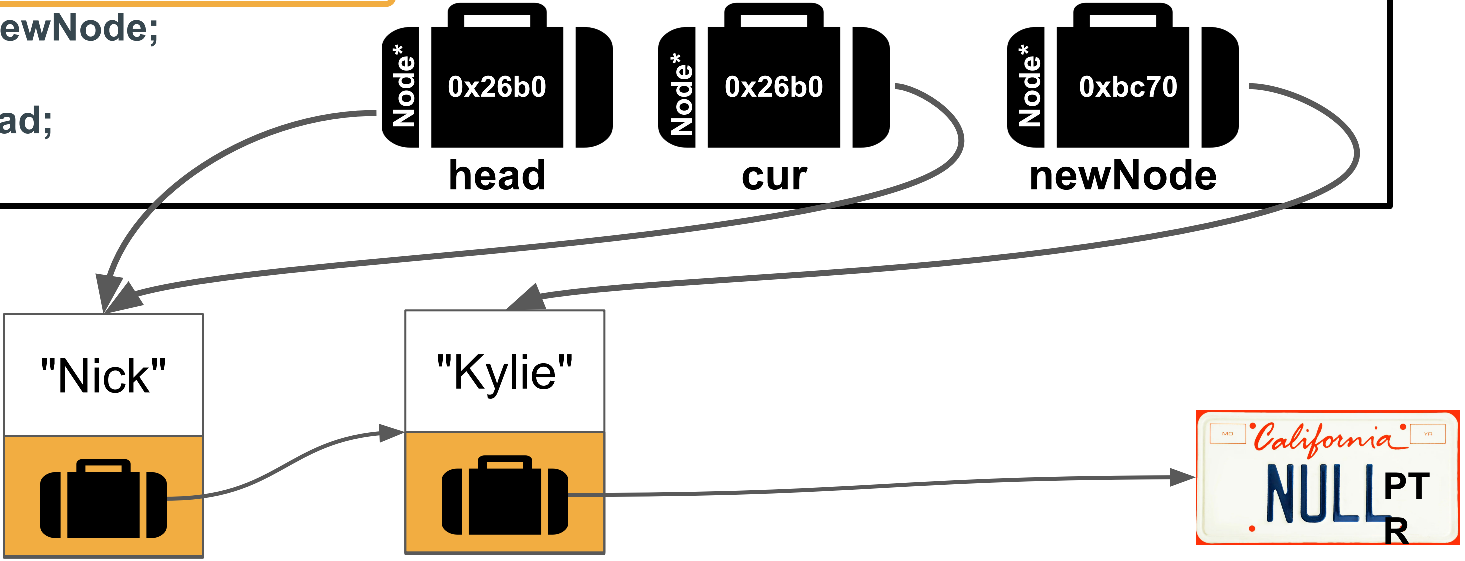


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

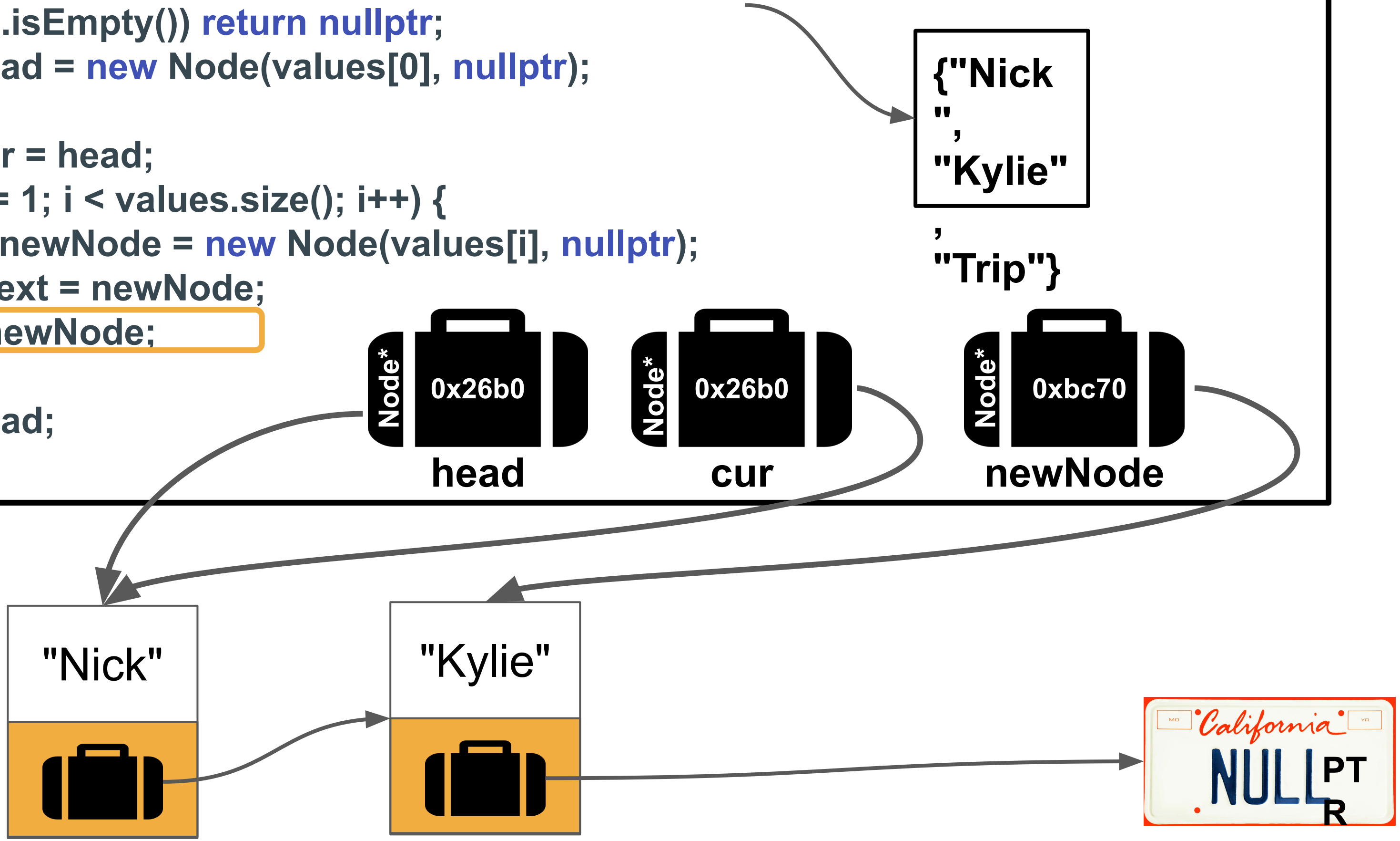


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

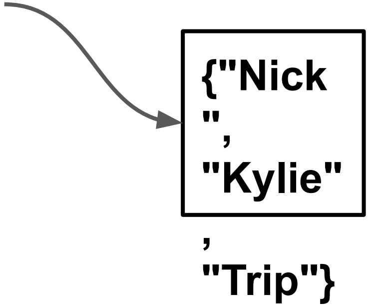


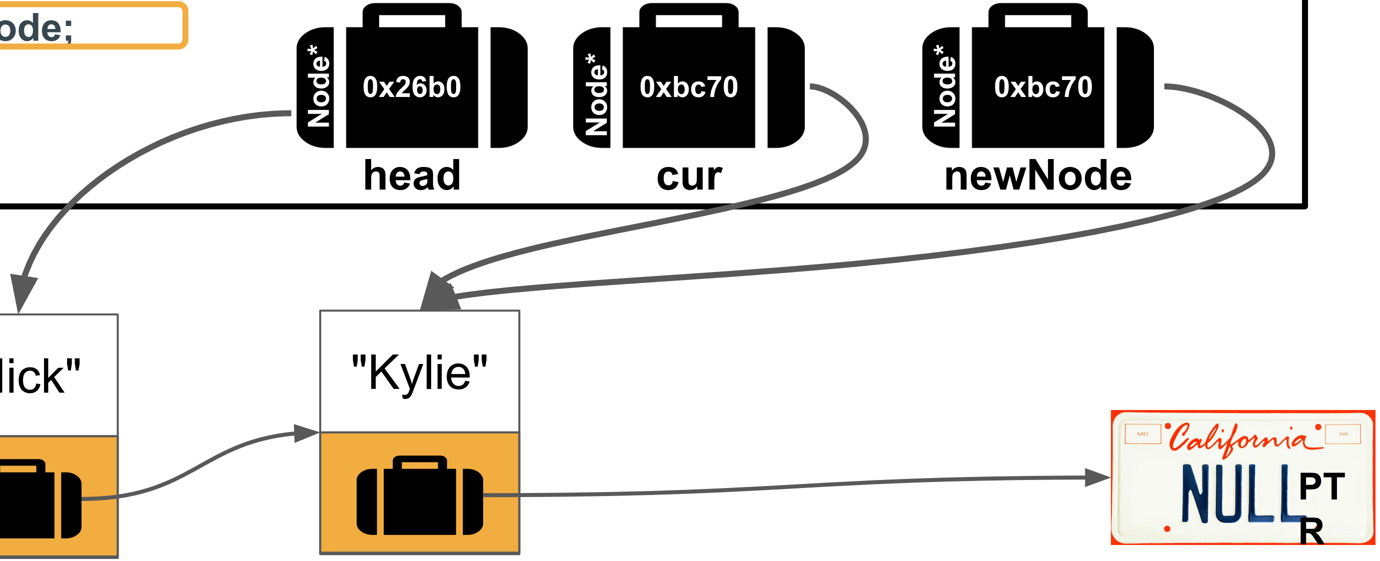


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

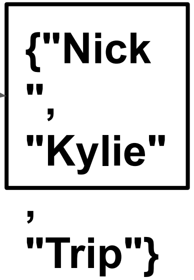


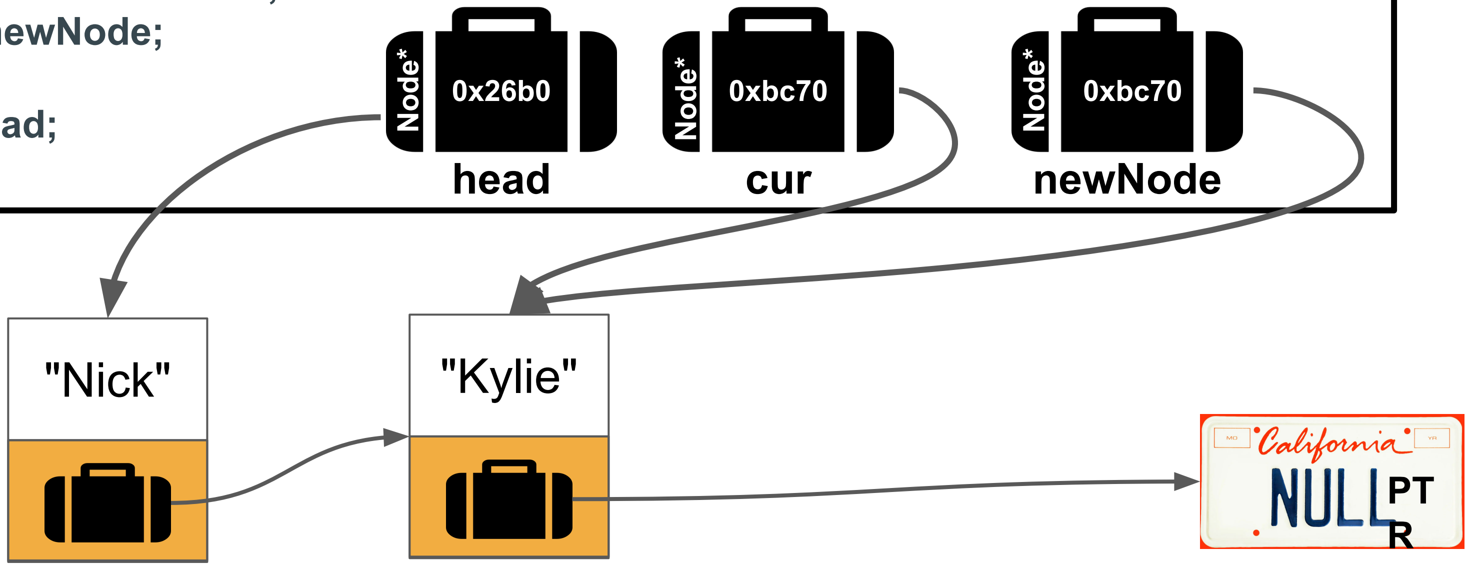
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



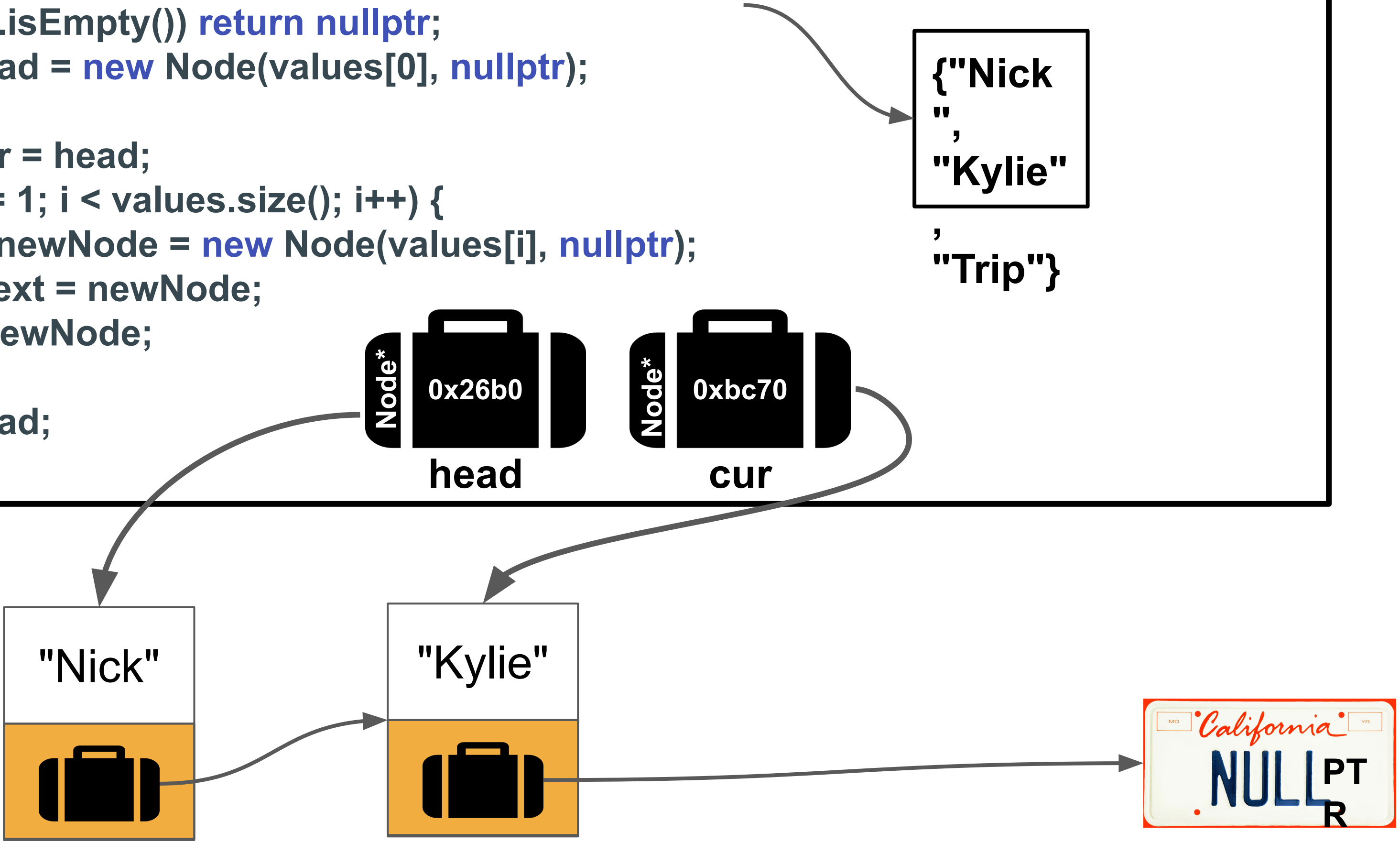


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

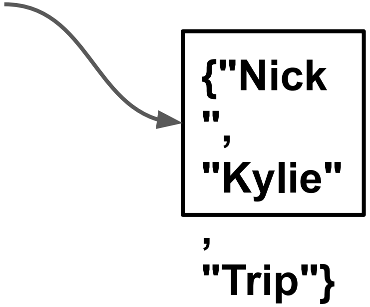


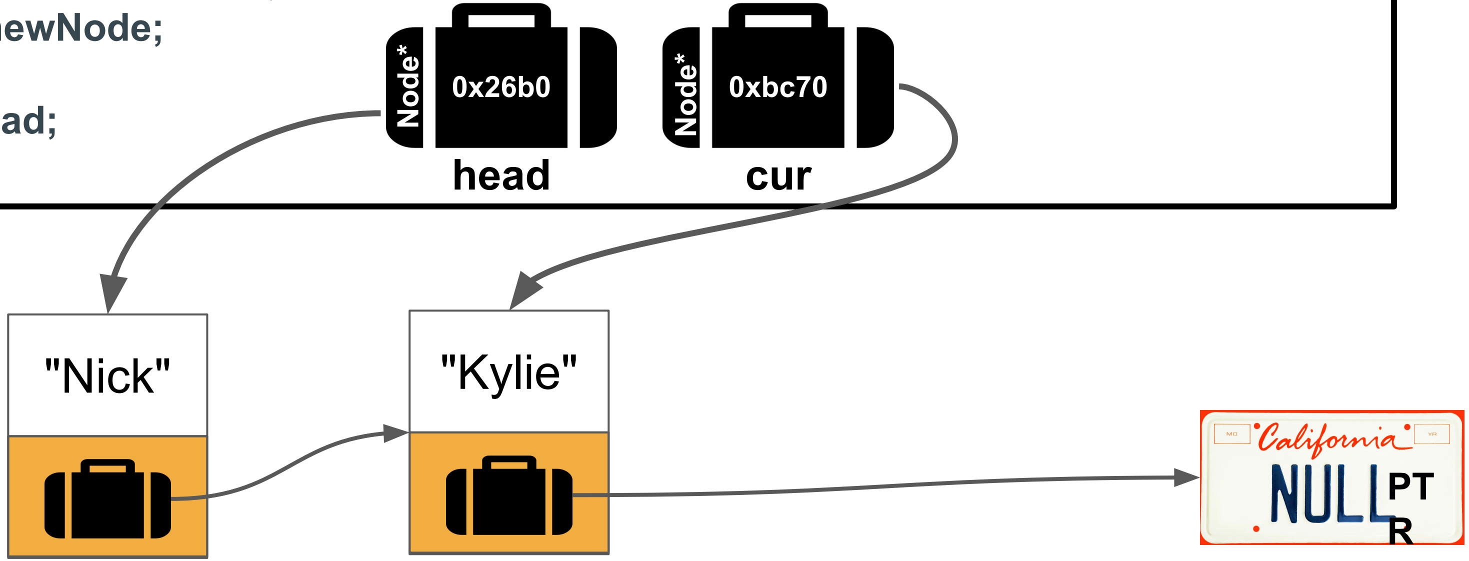


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



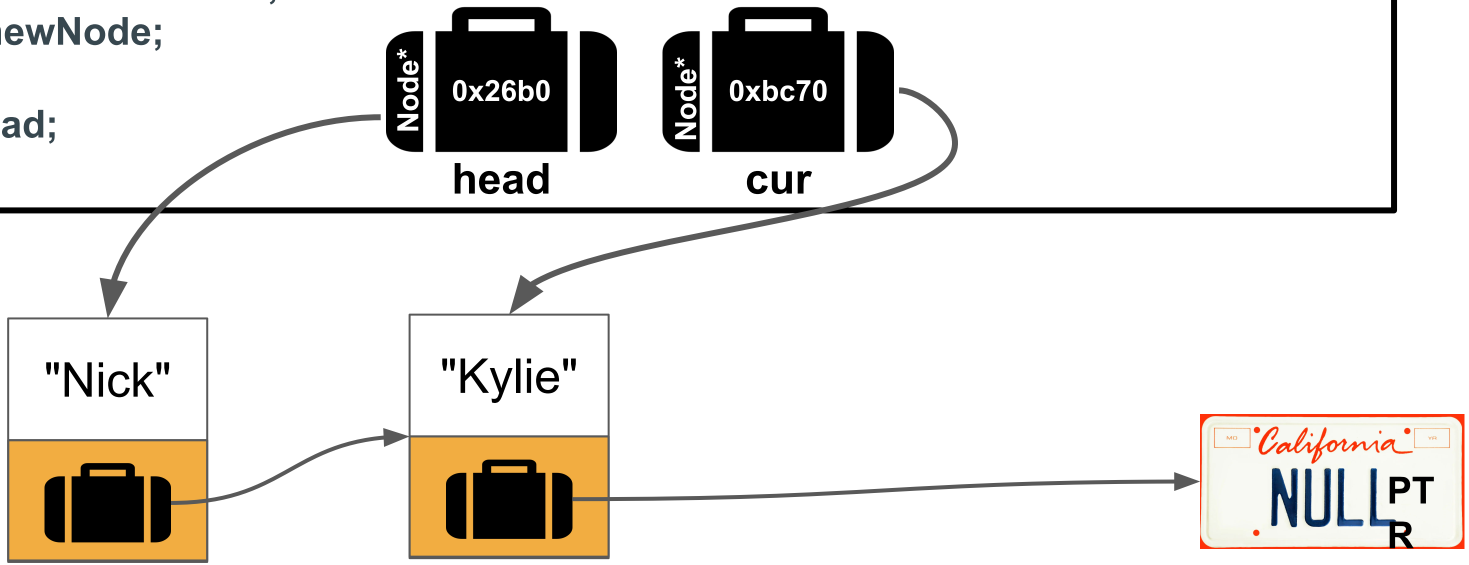
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



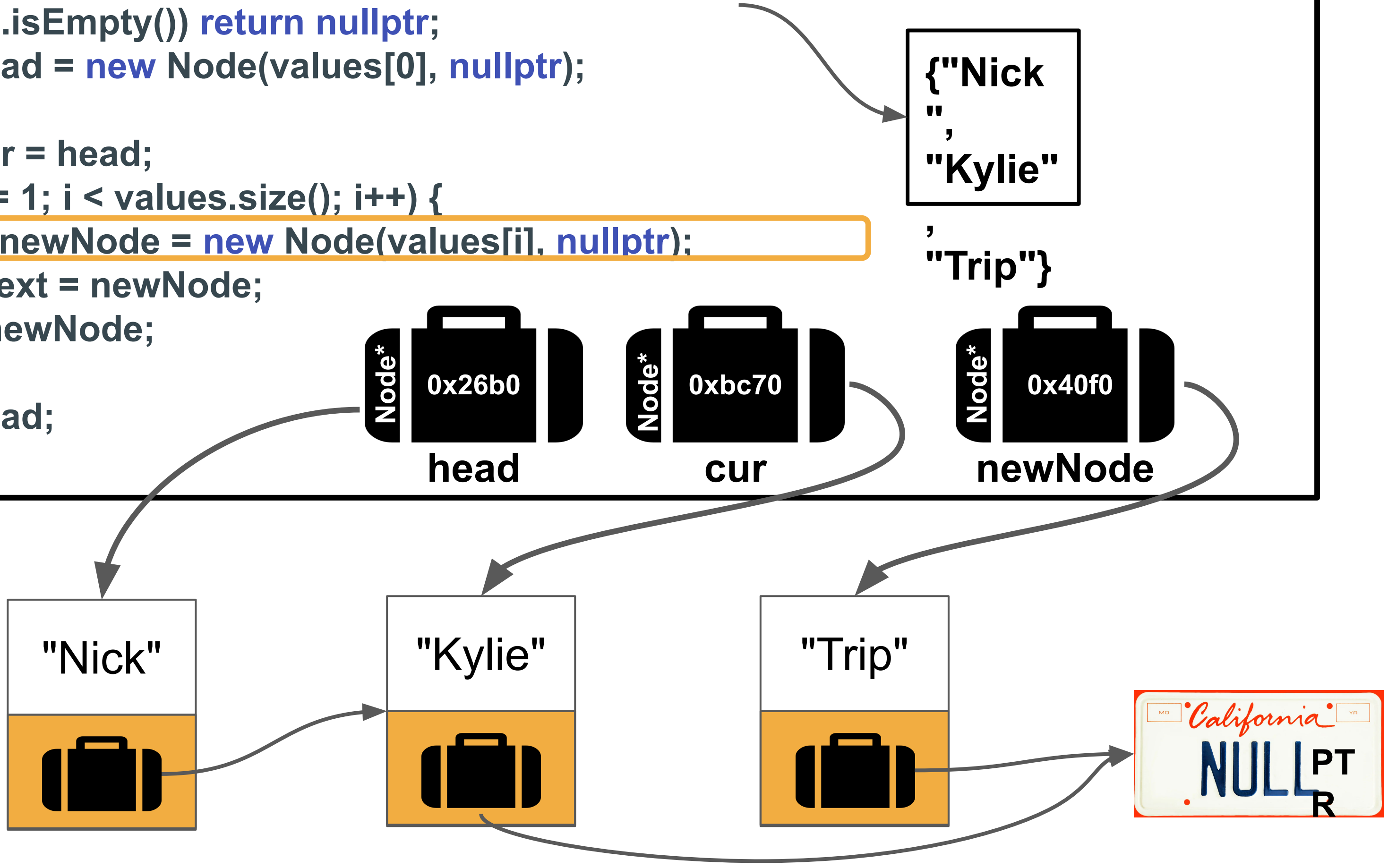


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

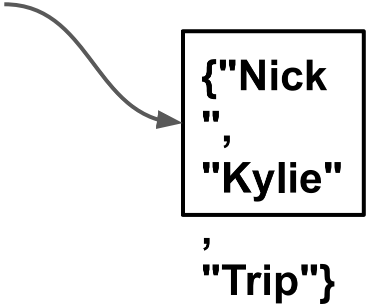
{"Nick", "Kylie", "Trip"}

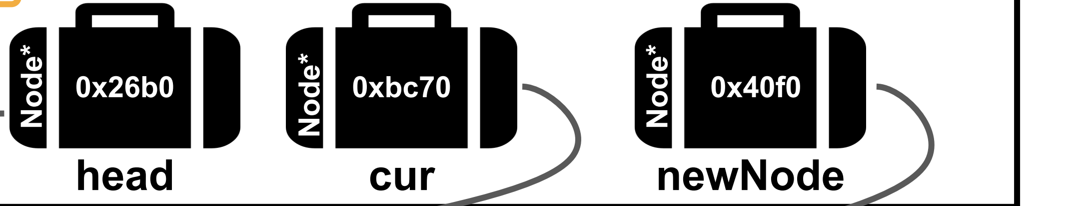


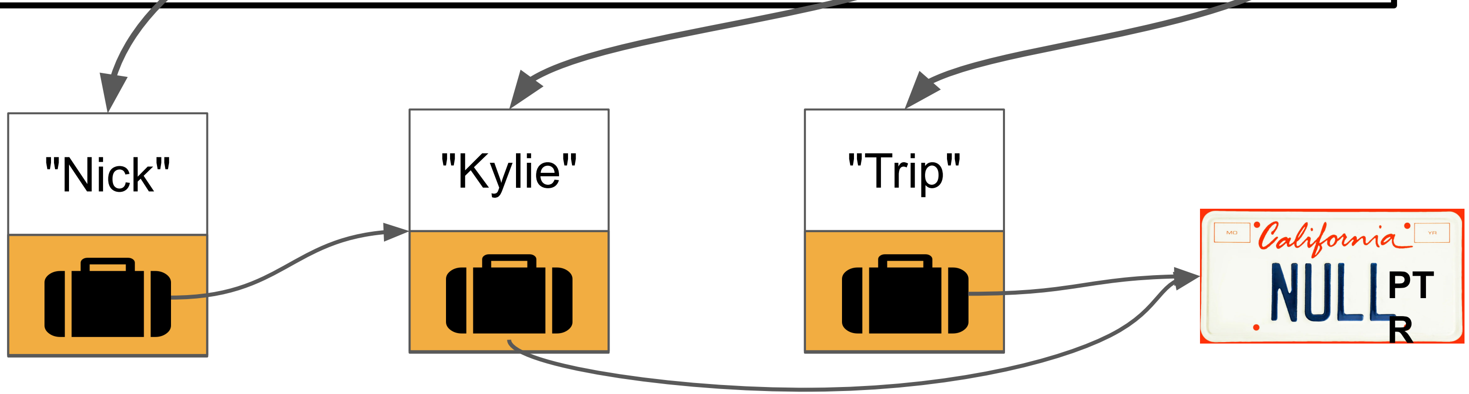
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

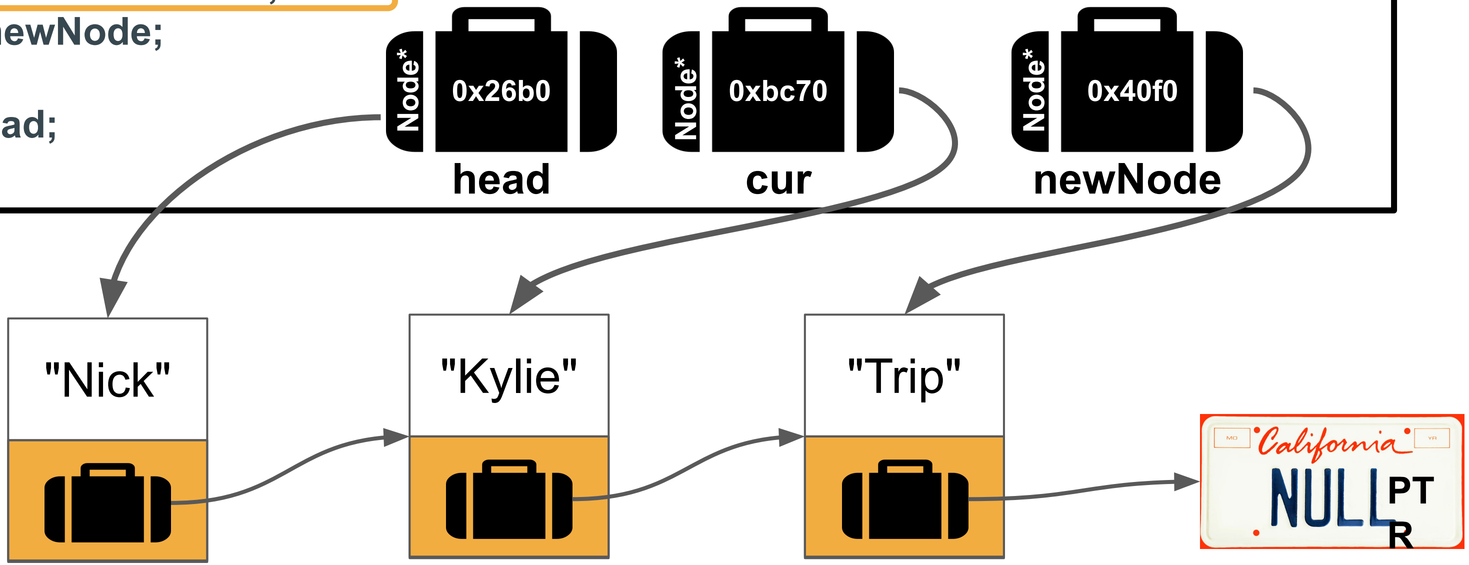




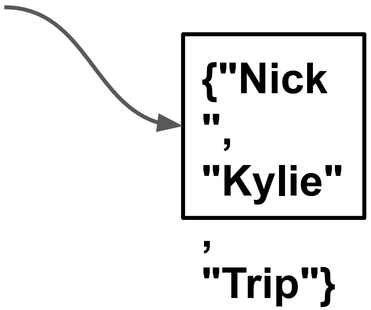


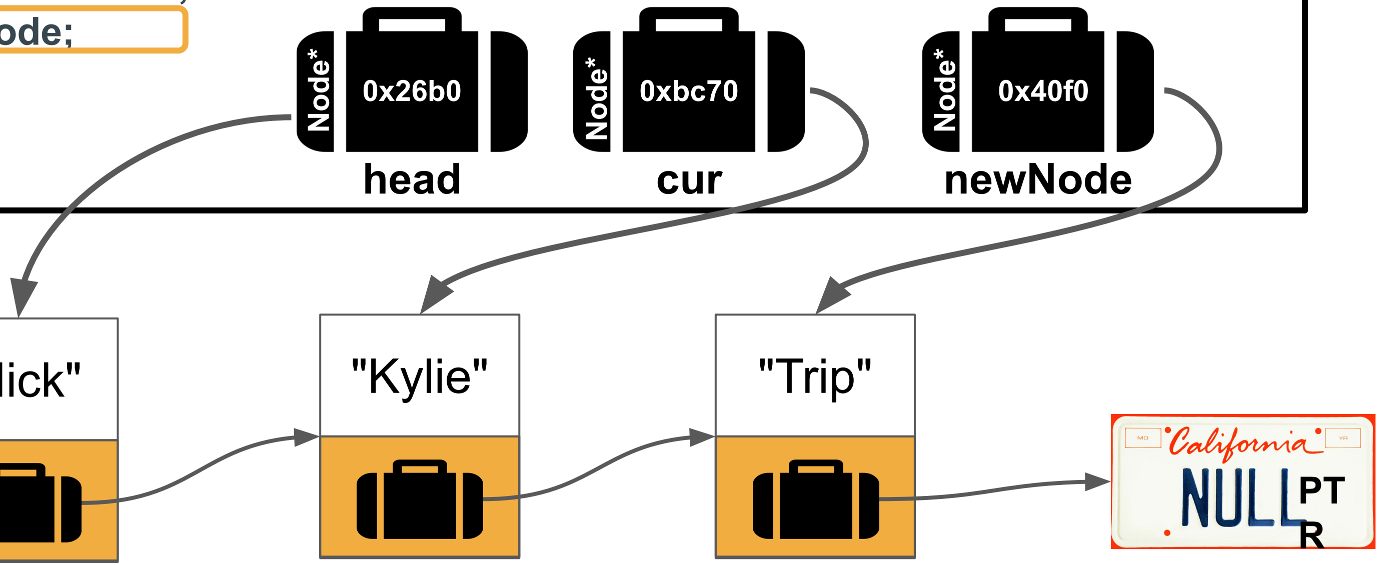
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Nick", "Kylie", "Trip"}

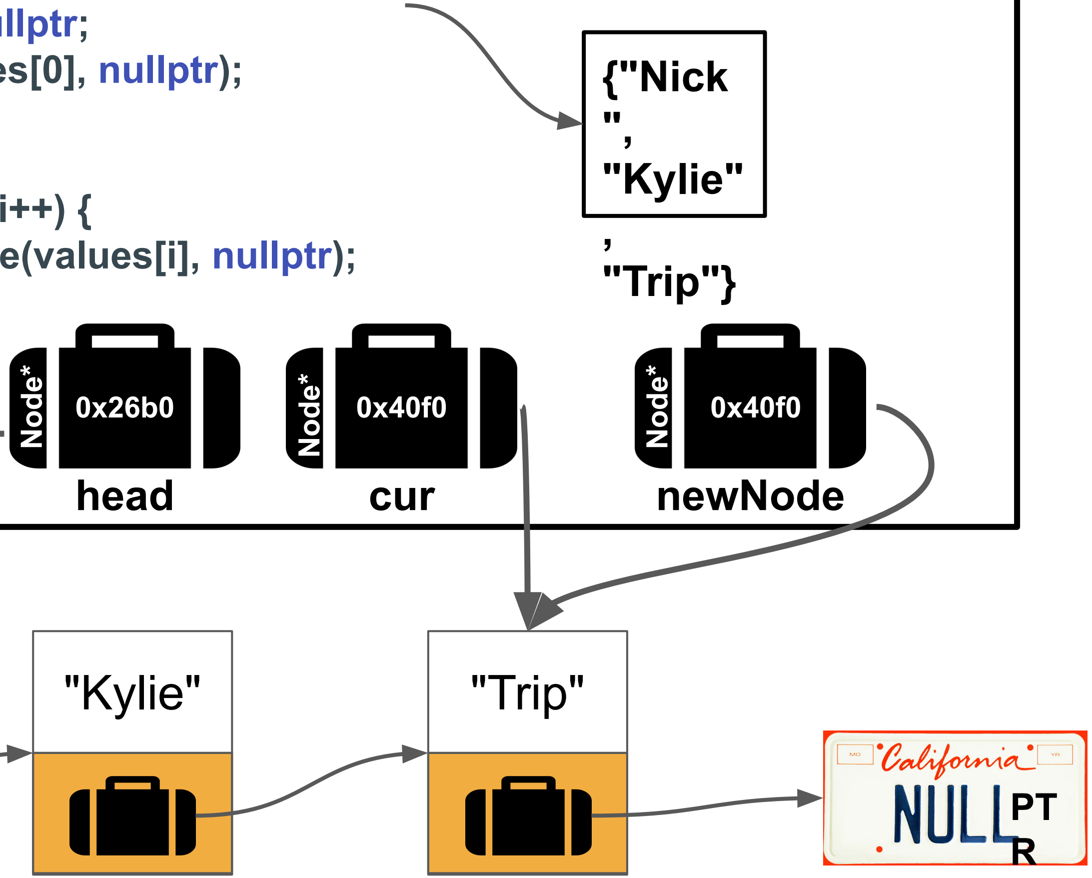


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

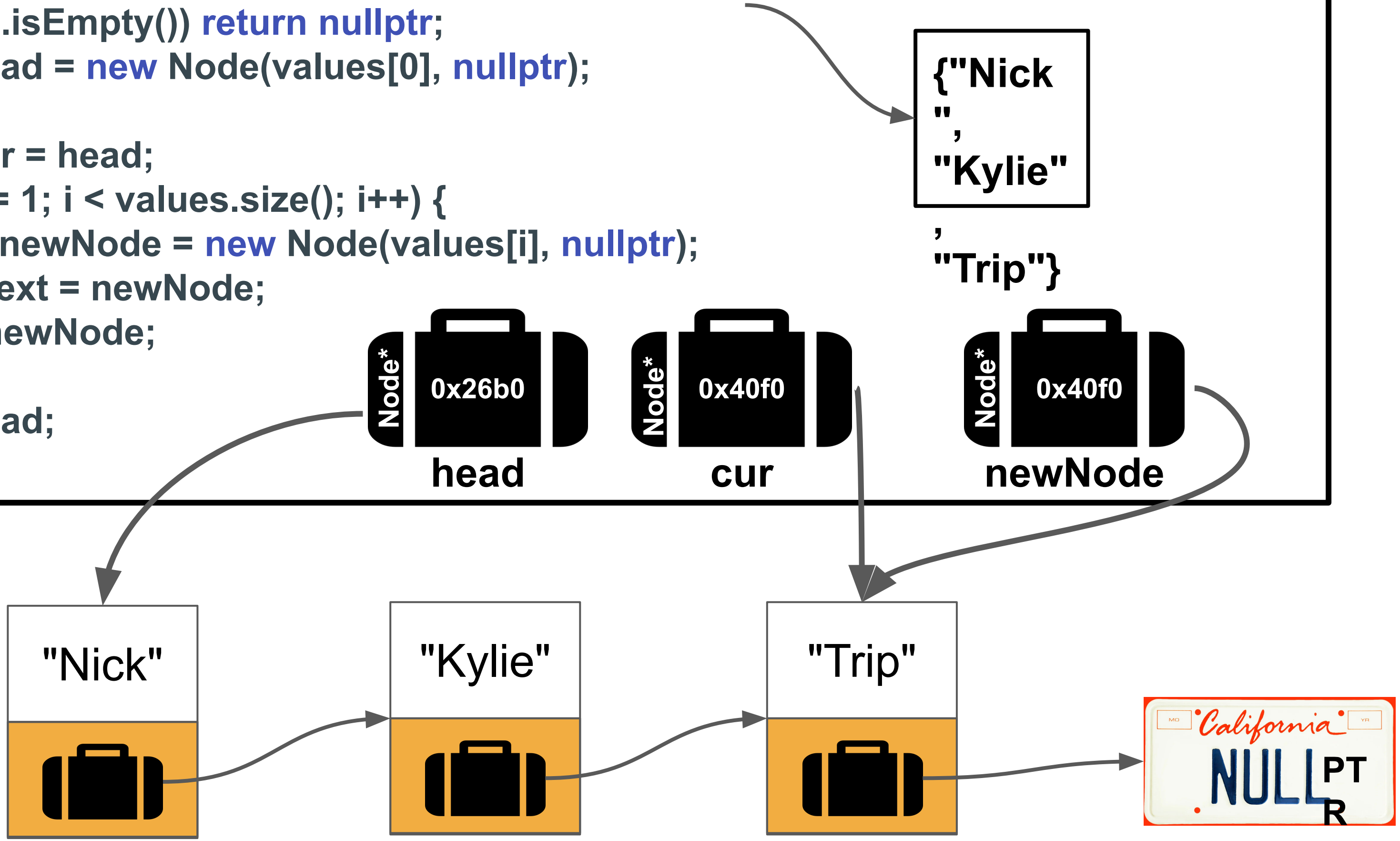




```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

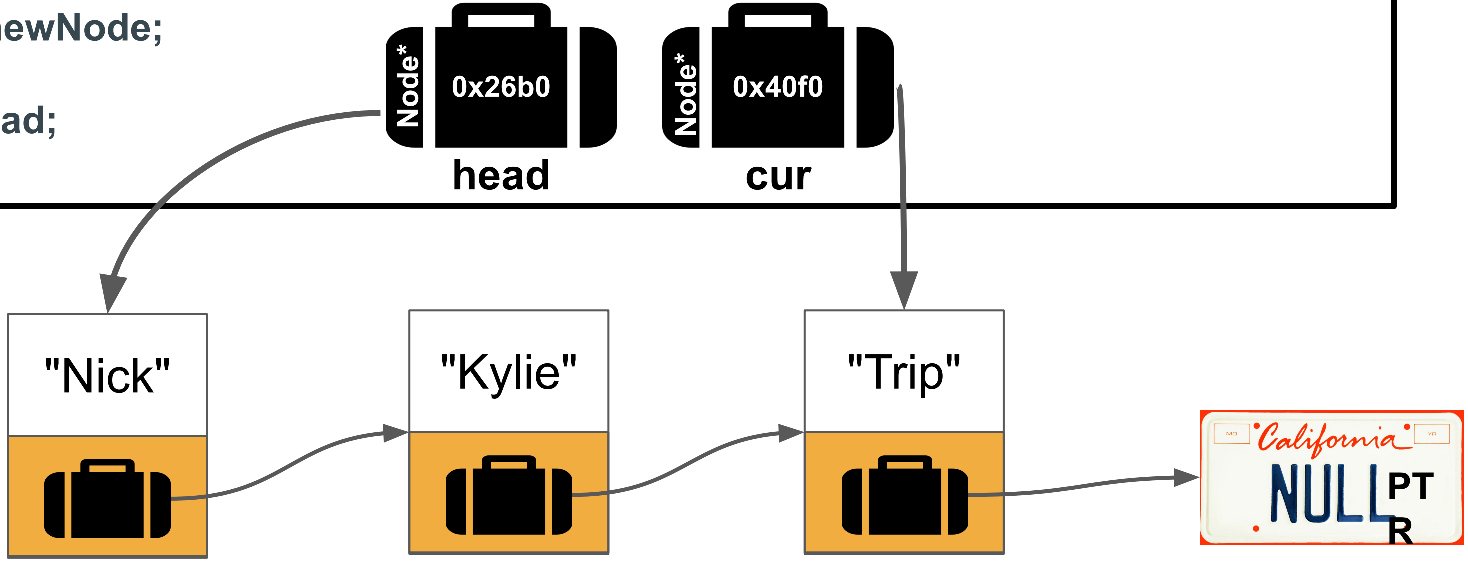


```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```



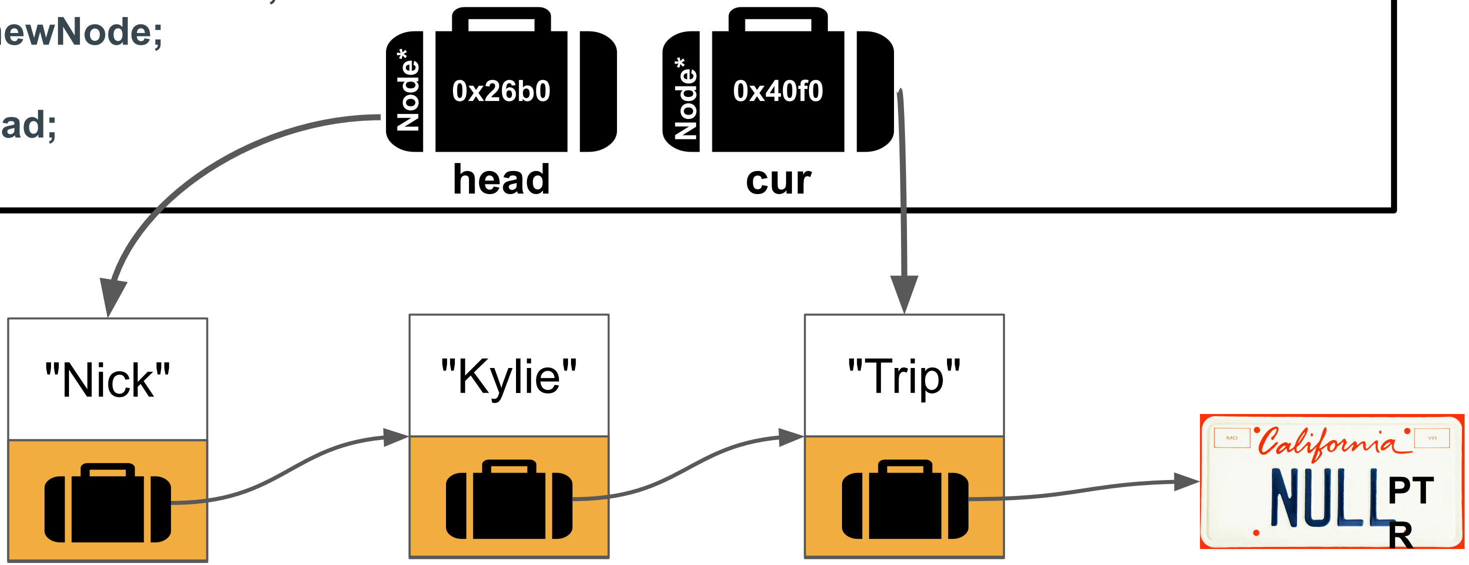
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Nick", "Kylie", "Trip"}



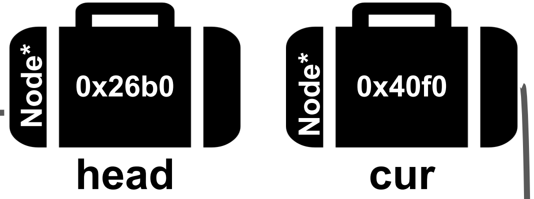
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

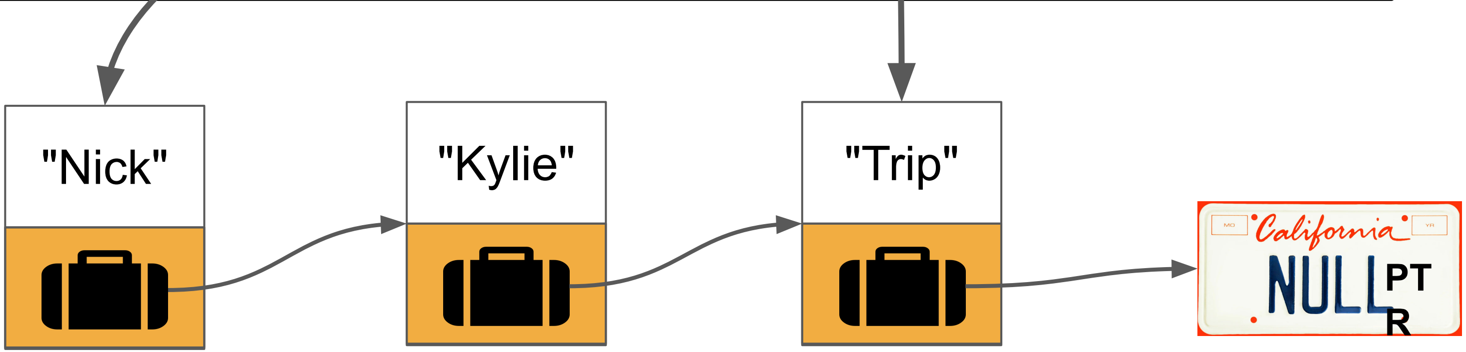
{"Nick", "Kylie", "Trip"}



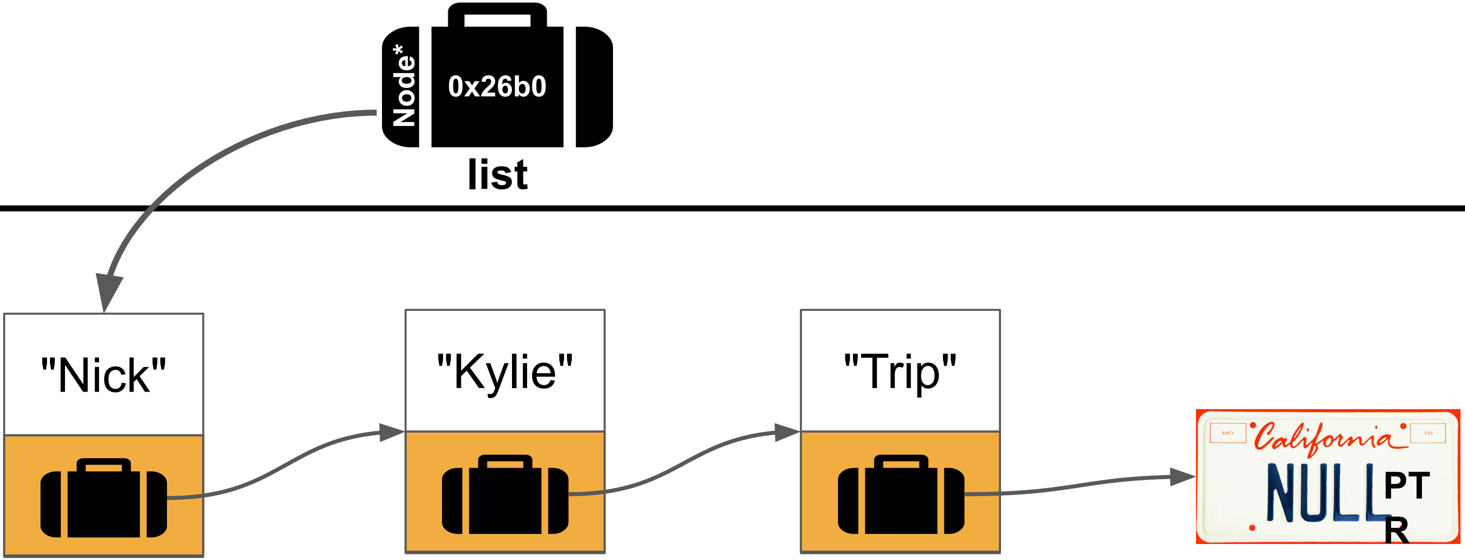
```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

{"Nick", "Kylie", "Trip"}





```
int main() {
    Vector<string> values = {"Nick", "Kylie", "Trip"};
    Node* list = createListWithTailPtr(values);

    /* Do other list-y things here, like printing/freeing the list. */
    return 0;
}
```



We just built a linked list with the **desired order of elements maintained** in **O(n)** time. Awesome!

Manipulating the middle of a list

Insertion/deletion in the middle of a list

- Why might we want this?
  - To maintain a particular sorted order of the list
  - To find and remove a particular element in the list

Insertion/deletion in the middle of a list

- Why might we want this?
  - To maintain a particular sorted order of the list
  - To find and remove a particular element in the list

- We're going to write two functions:
  - `alphabeticalAdd(Node*& list, string data)`
  - `remove(Node*& list, string dataToRemove)`

Insertion/deletion in the middle of a list

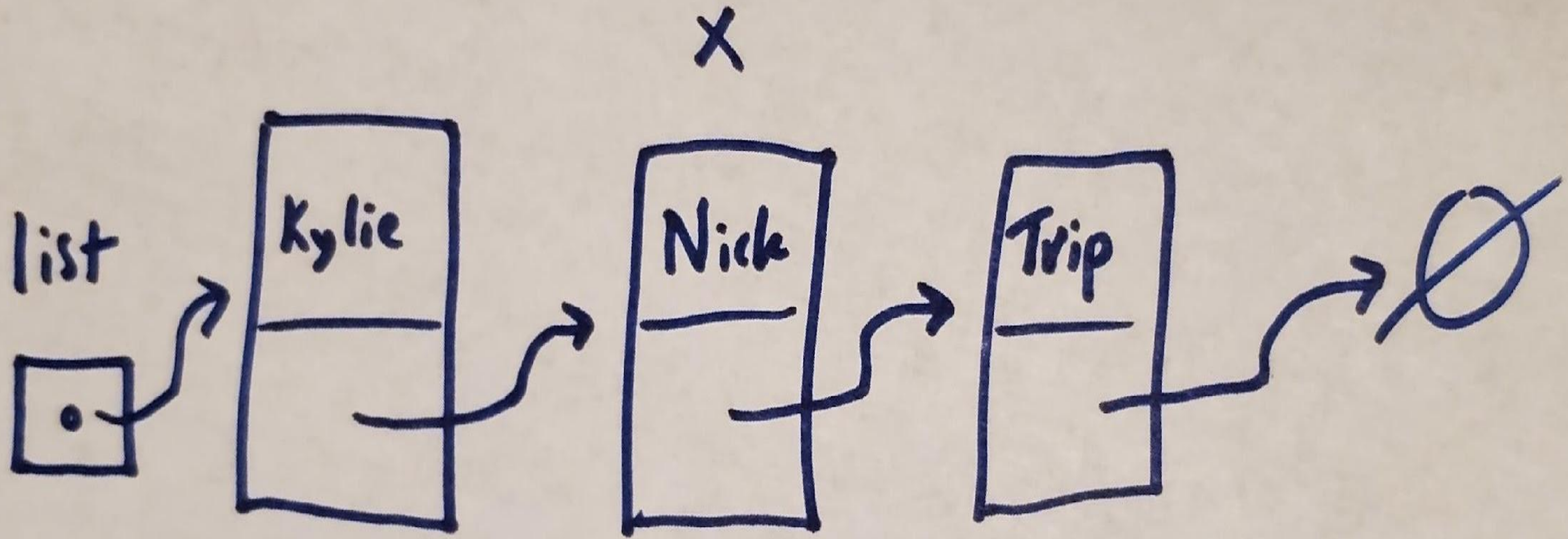
- Why might we want this?
    - To maintain a particular sorted order of the list
    - To find and remove a particular element in the list

- We're going to write two functions:
    - alphabeticalAdd(Node*& list, string data)
    - remove(Node*& list, string dataToRemove)

Note that we'll need to pass our list by reference!

alphabeticalAdd() –
Let’s code it!

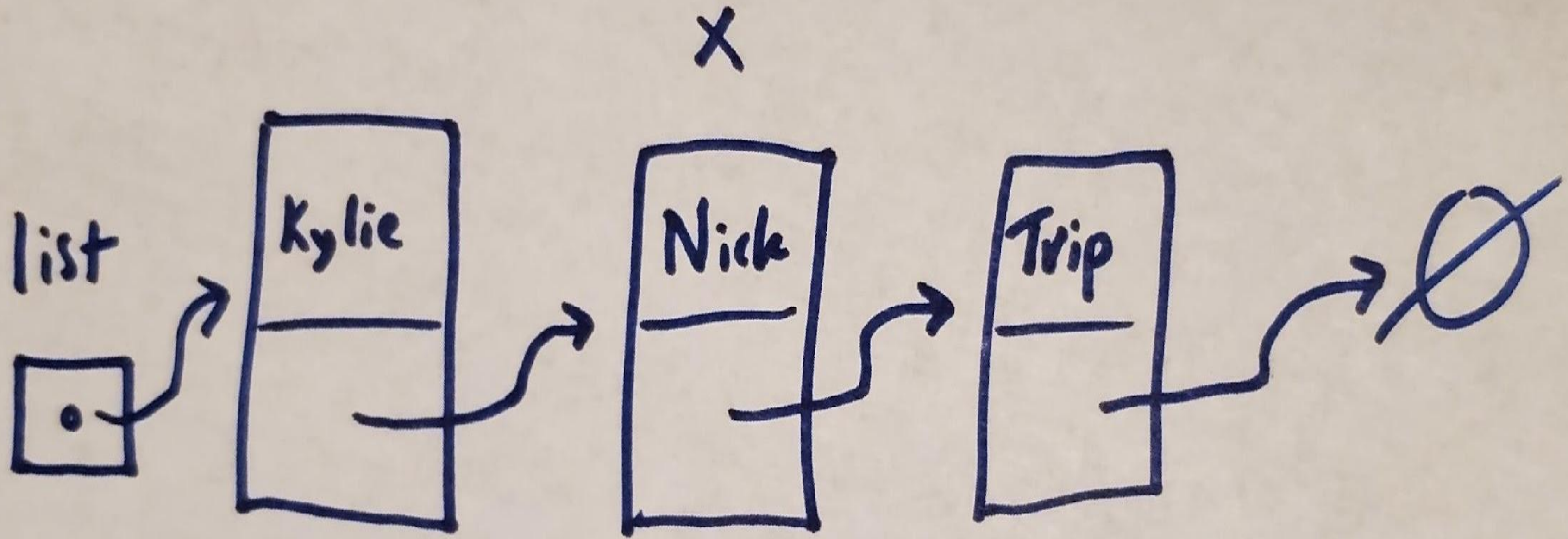
Implementing remove()


x
list
Kylie
Nick
Trip
Ø

```
void remove(Node*& list, string dataToRemove) {
    Node* cur = list;
    Node* prev = nullptr;
    while (cur != nullptr) {
        if (cur->data == dataToRemove) {
            // Rewire
                prev->next = cur->next;
                delete cur;
                prev = cur;
        }
    }
}
```

What’s Wrong?

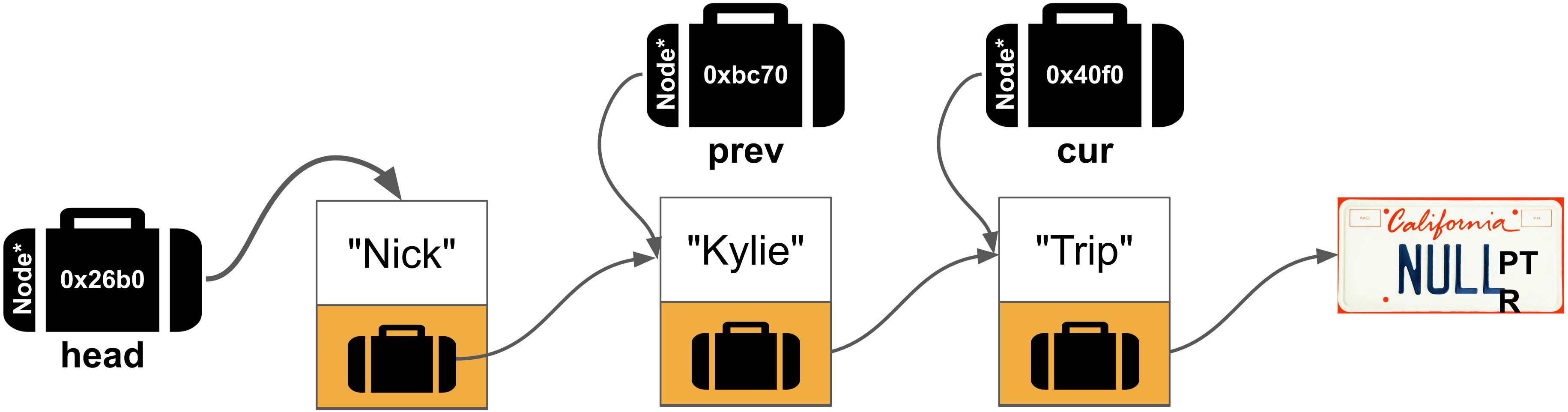
```
void remove(Node*& list, string dataToRemove) {
    Node* cur = list;
    Node* prev = nullptr;
    while (cur != nullptr) {
        if (cur->data == dataToRemove) {
            // Rewire
                prev->next = cur->next;
                delete cur;
                prev = cur;
        }
    }
}
```


x
list
Kylie
Nick
Trip

```
void remove(Node*& list, string dataToRemove) {
    Node* cur = list;
    Node* prev = nullptr;
    while (cur != nullptr) {
        if (cur->data == dataToRemove) {
            // Rewire
            if (prev != nullptr) {
                prev->next = cur->next;
            } else {
                list = cur->next;
            }
                delete cur;
        }
        prev = cur;
        cur = cur->next;
    }
}
```

Takeaways for manipulating the middle of a list

- While traversing to where you want to add/remove a node, you'll often want to keep track of both a current pointer and a previous pointer.
  - This makes rewiring easier between the two!
  - This also means you have to check that neither is nullptr before dereferencing.



Linked list summary

- You've now learned lots of ways to manipulate linked lists!
  - Traversal
  - Rewiring
  - Insertion (front/back/middle)
  - Deletion (front/back/middle)

- You've seen linked lists in classes and outside classes, and pointers passed by value and passed by reference.

- Assignment 5 will really test your understanding of linked lists.
  - Draw lots of pictures!
  - Test small parts of your code at a time to make sure individual operations are working correctly.

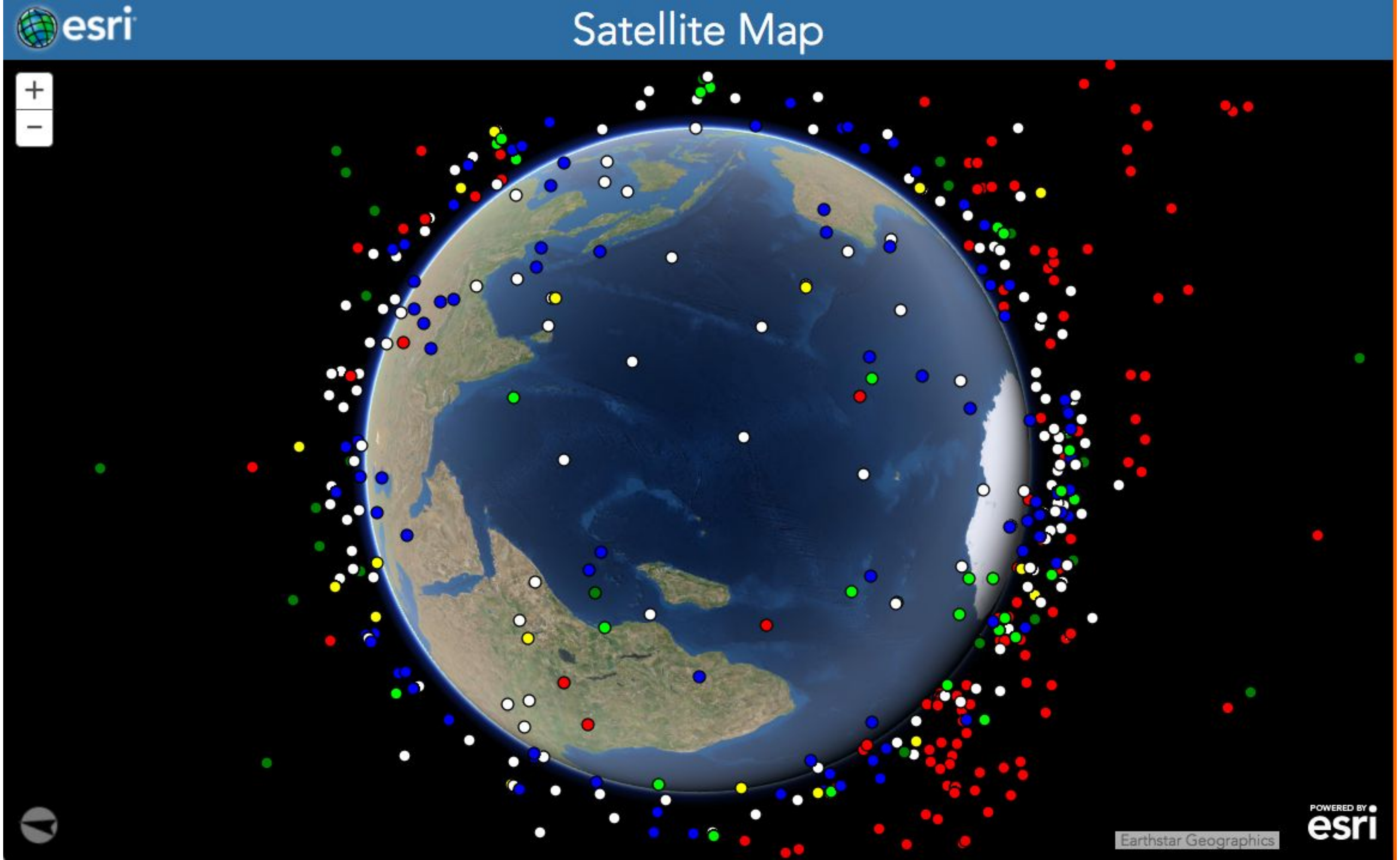
Announcements

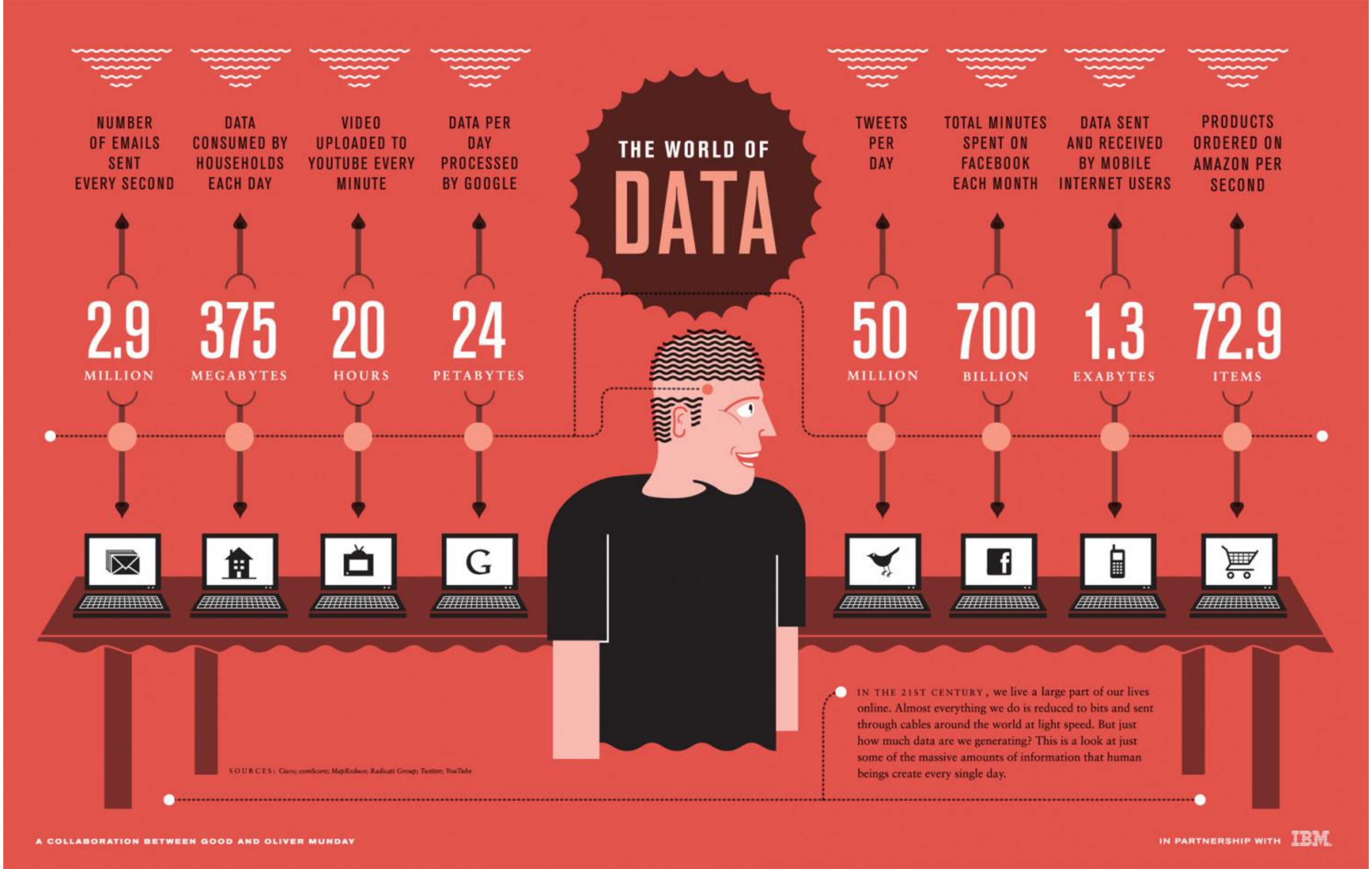
Announcements

- Assignment 5 will be released by the end of the day today and will be due on **Tuesday, August 3 at 11:59pm PDT**.
    - YEAH for A5 will be **Friday 7/30** at 11:30am PDT.

- The **End of Quarter Assessment** will take place over a 3-day period, like our Mid-Quarter Diagnostic, and that will be **August 13-15**. More on it later.

Real-World Algorithms


esri
Satellite Map
+
−
Earthstar Geographics
POWERED BY
esri


THE WORLD OF DATA
NUMBER OF EMAILS SENT EVERY SECOND
2.9 MILLION
DATA CONSUMED BY HOUSEHOLDS EACH DAY
375 MEGABYTES
VIDEO UPLOADED TO YOUTUBE EVERY MINUTE
20 HOURS
DATA PER DAY PROCESSED BY GOOGLE
24 PETABYTES
G
TWEETS PER DAY
50 MILLION
TOTAL MINUTES SPENT ON FACEBOOK EACH MONTH
700 BILLION
f
DATA SENT AND RECEIVED BY MOBILE INTERNET USERS
1.3 EXABYTES
PRODUCTS ORDERED ON AMAZON PER SECOND
72.9 ITEMS
IN THE 21ST CENTURY, we live a large part of our lives online. Almost everything we do is reduced to bits and sent through cables around the world at light speed. But just how much data are we generating? This is a look at just some of the massive amounts of information that human beings create every single day.
SOURCES: Cisco; comScore; MapReduce; Radicati Group; Twitter; YouTube
A COLLABORATION BETWEEN GOOD AND OLIVER MUNDAY
IN PARTNERSHIP WITH IBM

What are real-world algorithms that can be used to organize data?

Sorting

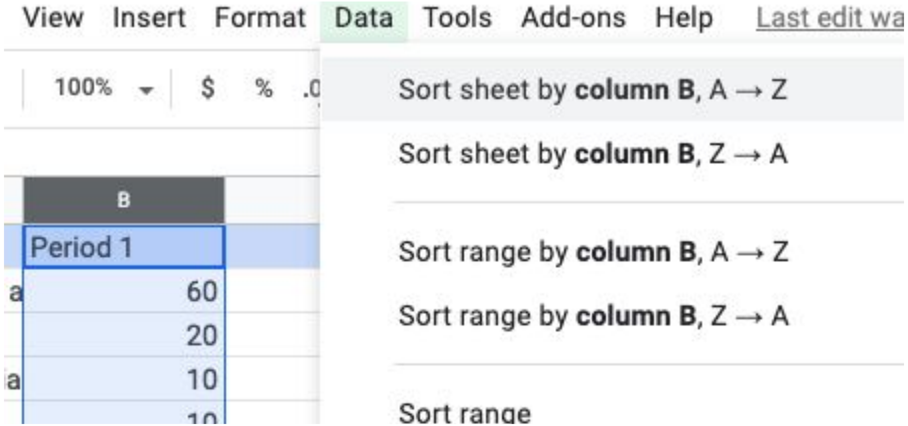
Definition

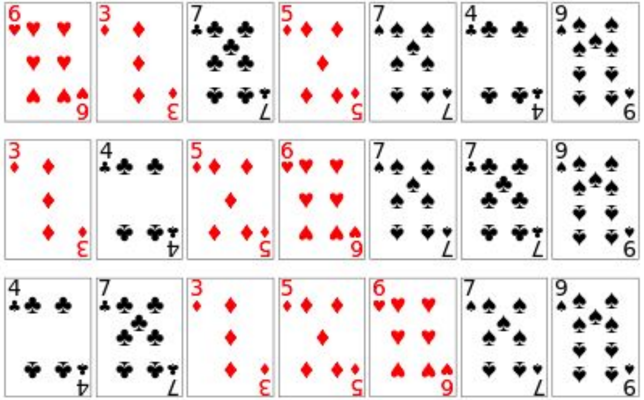
sorting

Given a list of data points, sort those data points into ascending / descending order by some quantity.

Why is sorting useful?

Time	Auto	Athlete	Nationality	Date	Venue
4:37.0		Anne Smith	United Kingdom	3 June 1967[8]	London
4:36.8		Maria Gommers	Netherlands	14 June 1969[8]	Leicester
4:35.3		Ellen Tittel	West Germany	20 August 1971[8]	Sittard
4:29.5		Paola Pigni	Italy	8 August 1973[8]	Viareggio
4:23.8		Natalia Mărășescu	Romania	21 May 1977[8]	Bucharest
4:22.1	4:22.09	Natalia Mărășescu	Romania	27 January 1979[8]	Auckland
4:21.7	4:21.68	Mary Decker	United States	26 January 1980[8]	Auckland
	4:20.89	Lyudmila Veselkova	Soviet Union	12 September 1981[8]	Bologna
	4:18.08	Mary Decker-Tabb	United States	9 July 1982[8]	Paris
	4:17.44	Maricica Puică	Romania	9 September 1982[8]	Rieti
	4:16.71	Mary Decker-Slaney	United States	21 August 1985[8]	Zürich
	4:15.61	Paula Ivan	Romania	10 July 1989[8]	Nice
	4:12.56	Svetlana Masterkova	Russia	14 August 1996[8]	Zürich
	4:12.33	Sifan Hassan	Netherlands	12 July 2019	Monaco





Approaches to sorting

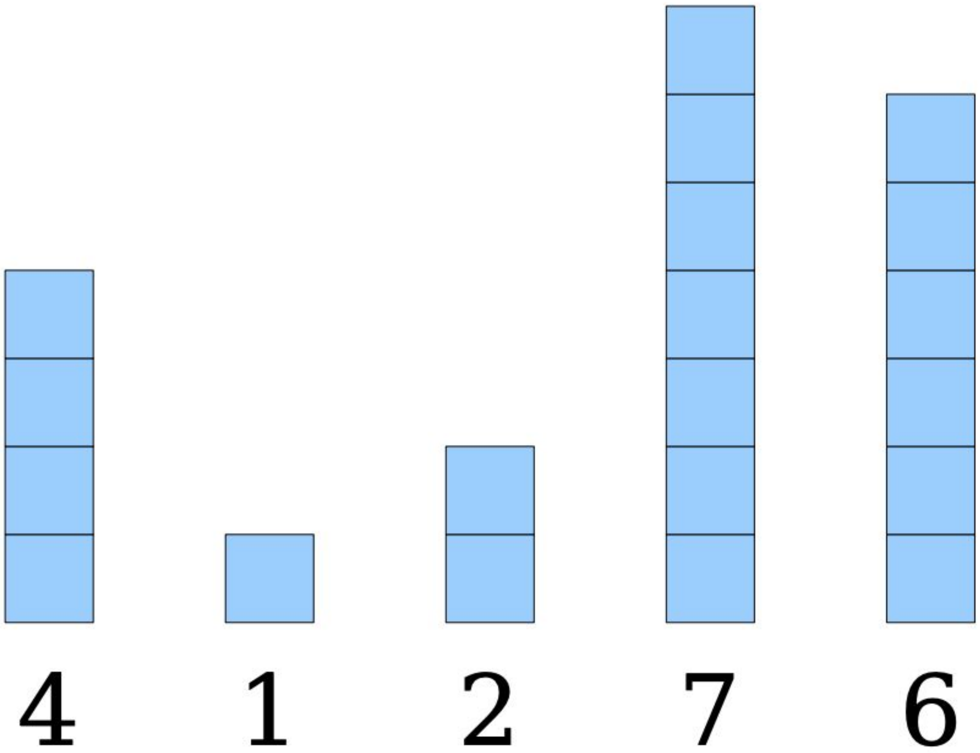
- Suppose we want to rearrange a sequence to put elements into ascending order (each element is less than or equal to the element that follows it).

- Over the next 2 days, we're going to answer the following questions:
  - What are some strategies we could use?
  - How do those strategies compare?
  - Is there a “best” strategy?

Sorting algorithms

Animations courtesy of Keith Schwarz!

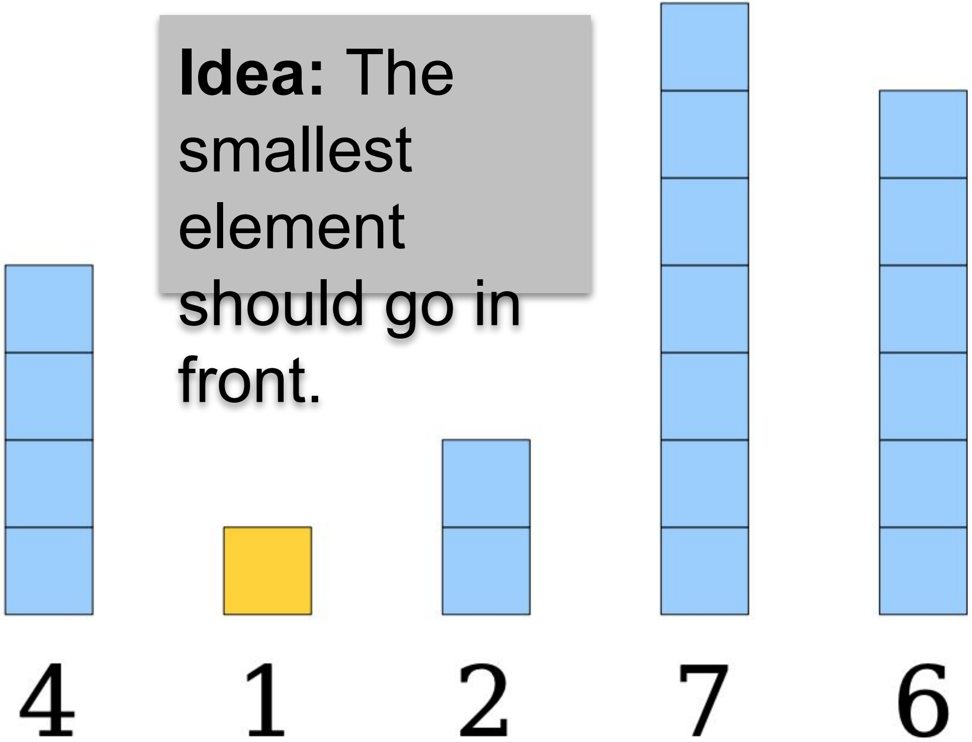
Our first sort: Selection sort



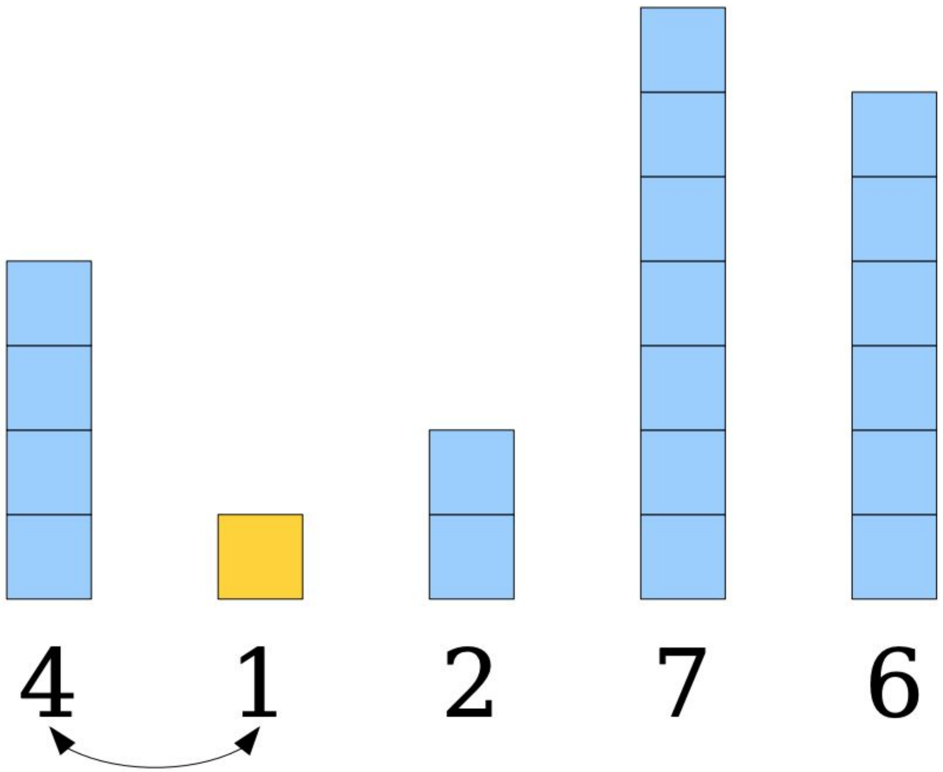
Visualizations created by

Our first sort: Selection sort

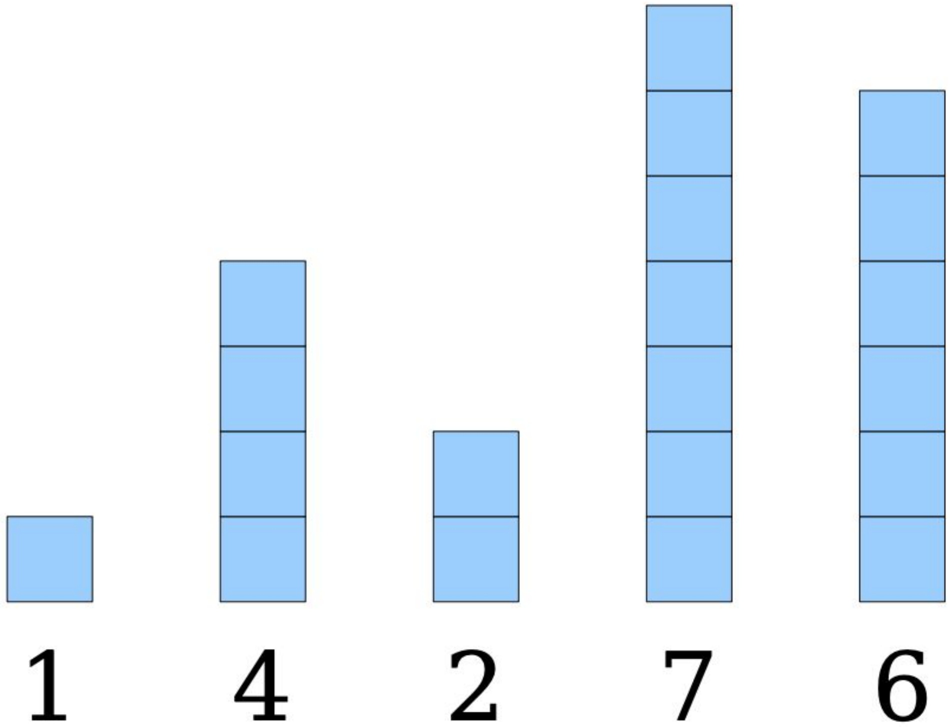
Idea: The smallest element should go in front.



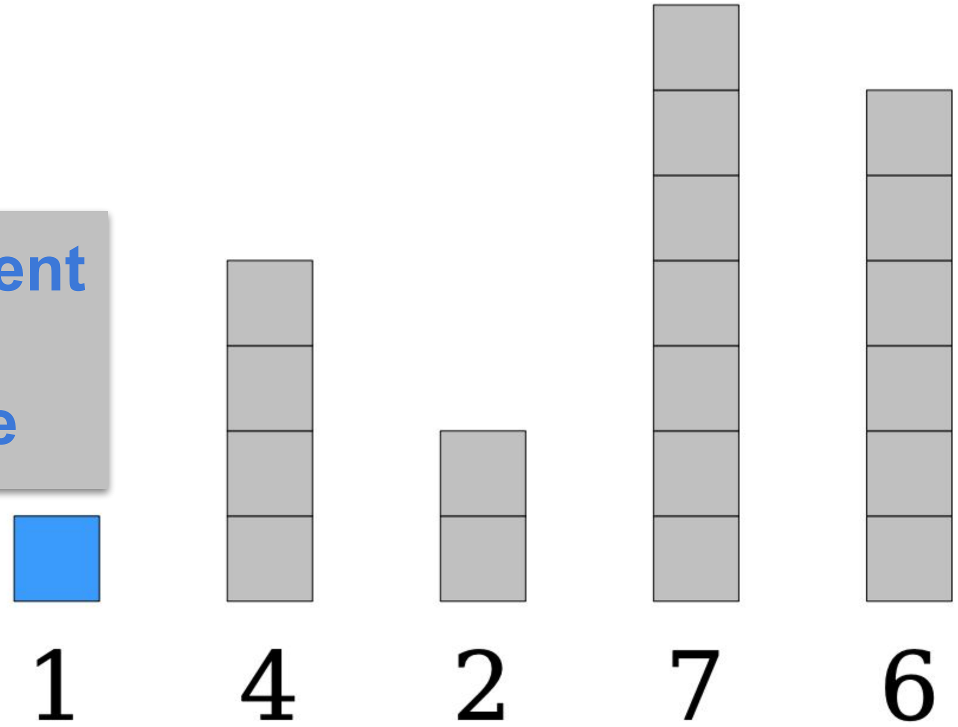
Our first sort: Selection sort



Our first sort: Selection sort



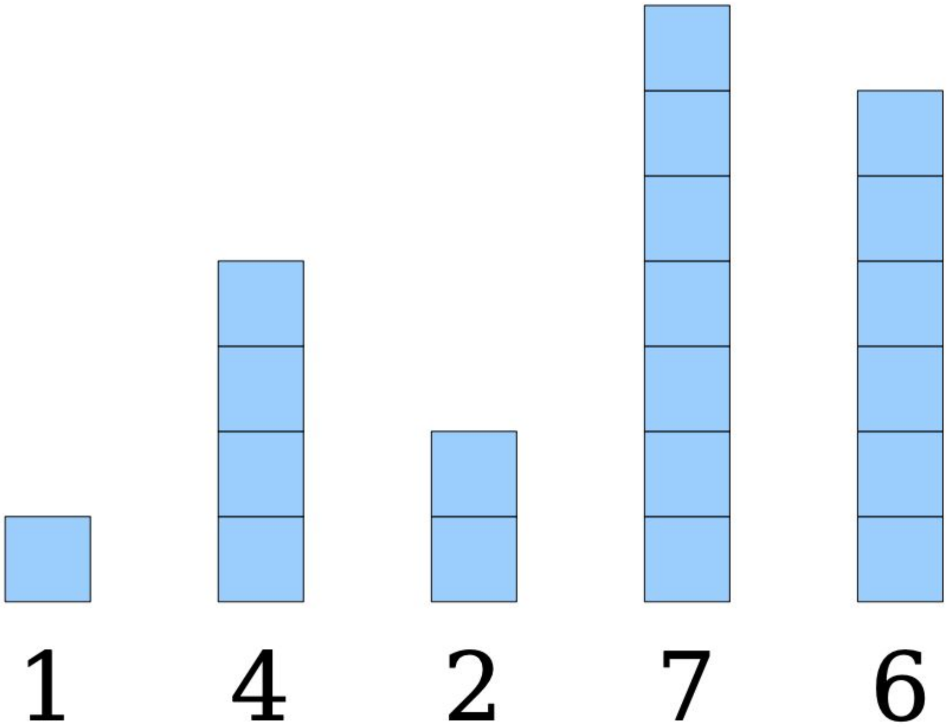
Our first sort: Selection sort



This element is in the right place now.

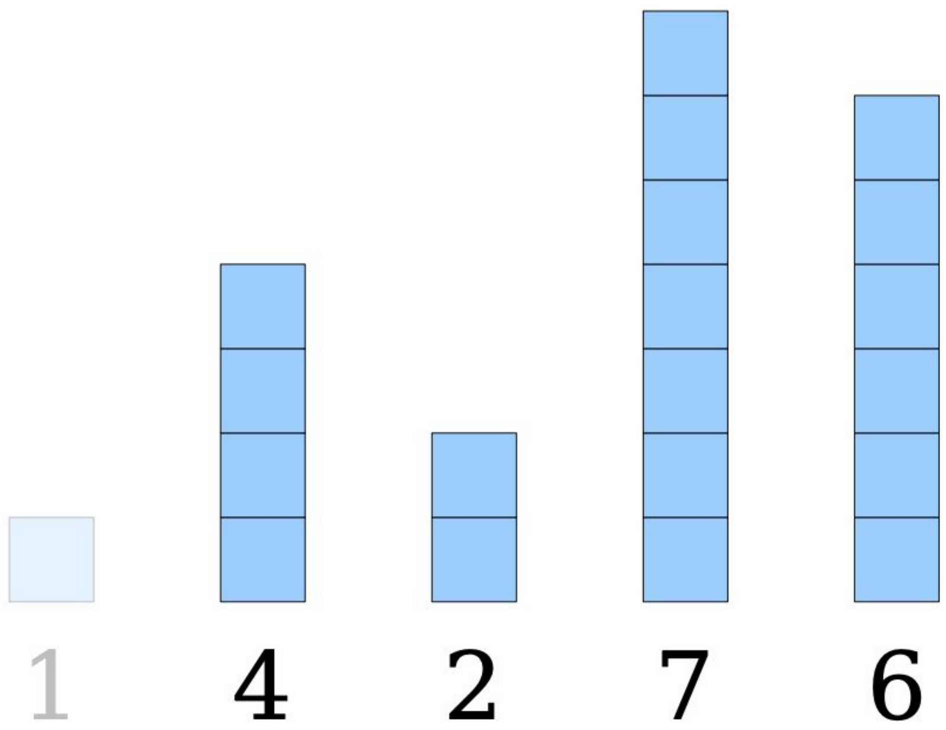
The remaining elements are in no particular order.

Our first sort: Selection sort



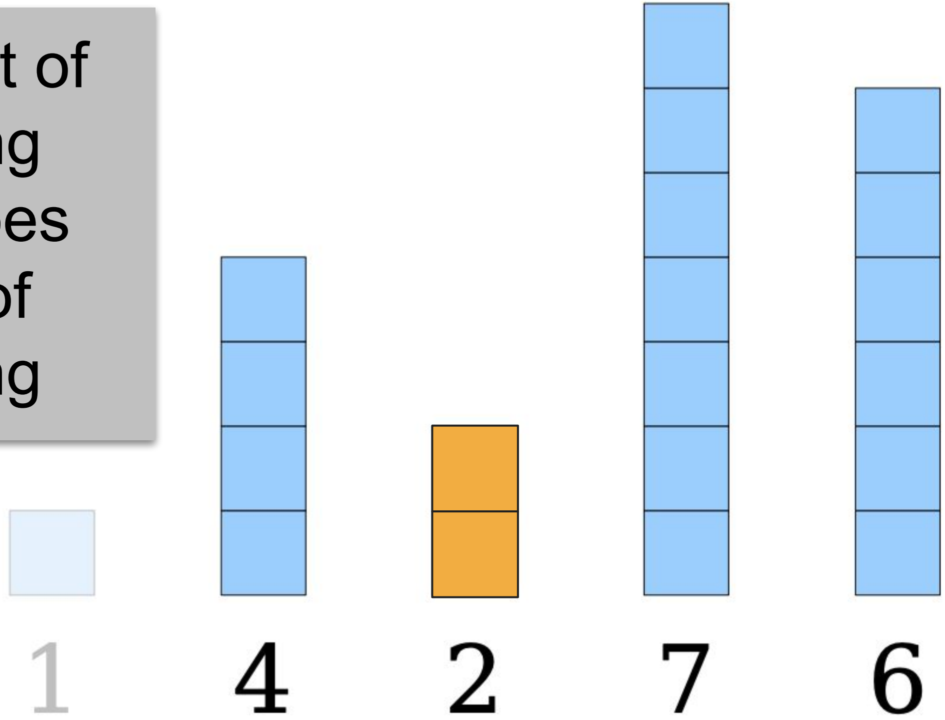
Visualizations created by

Our first sort: Selection sort

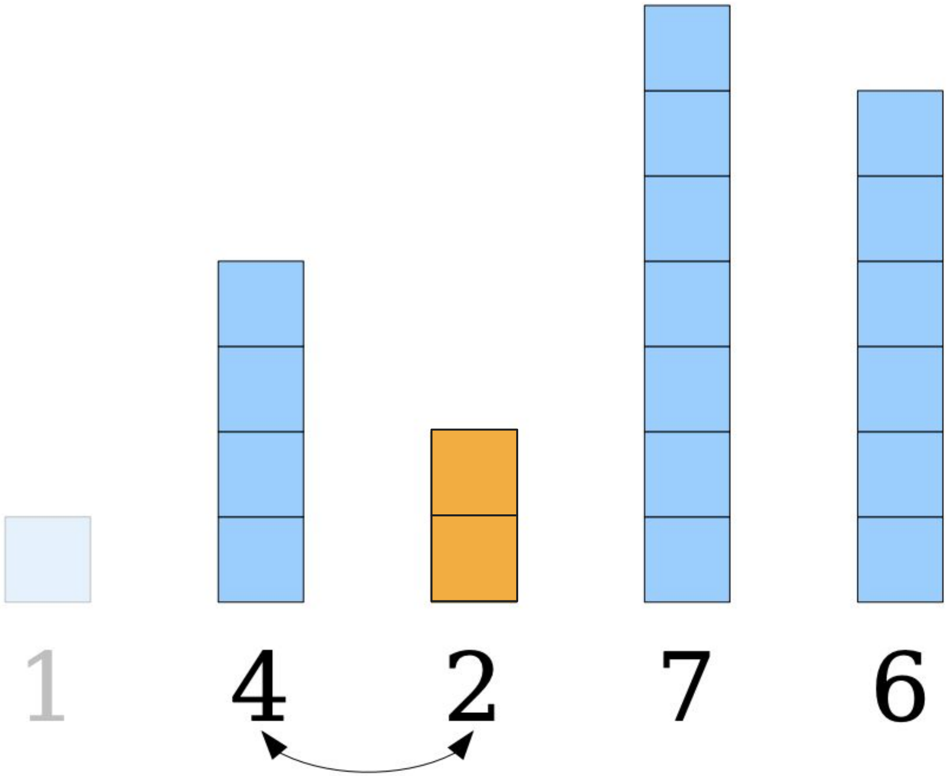


Our first sort: Selection sort

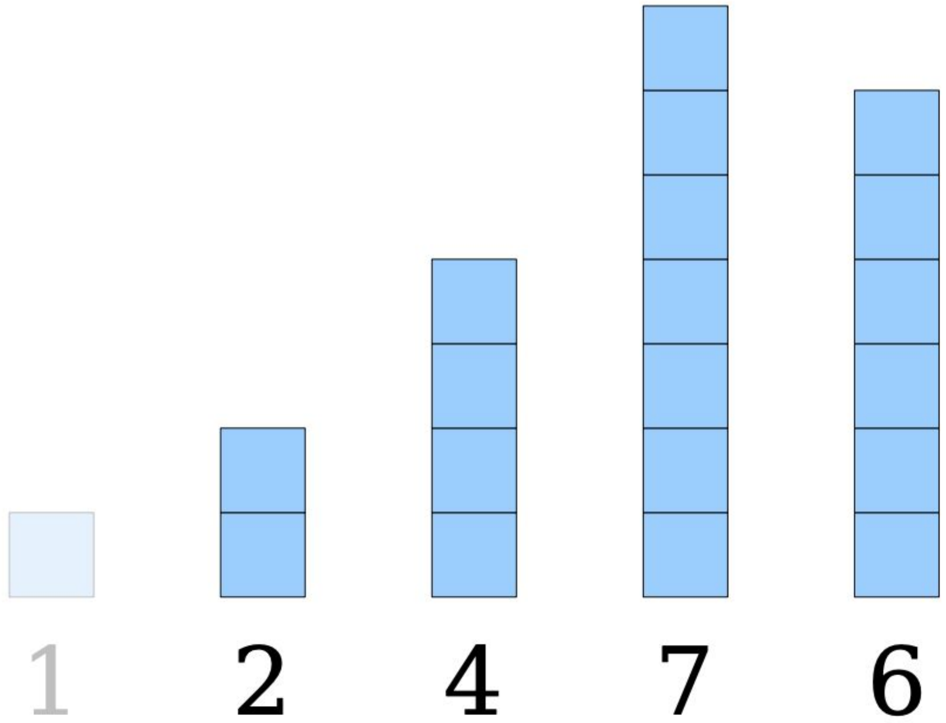
The smallest of the remaining elements goes at the front of the remaining elements.



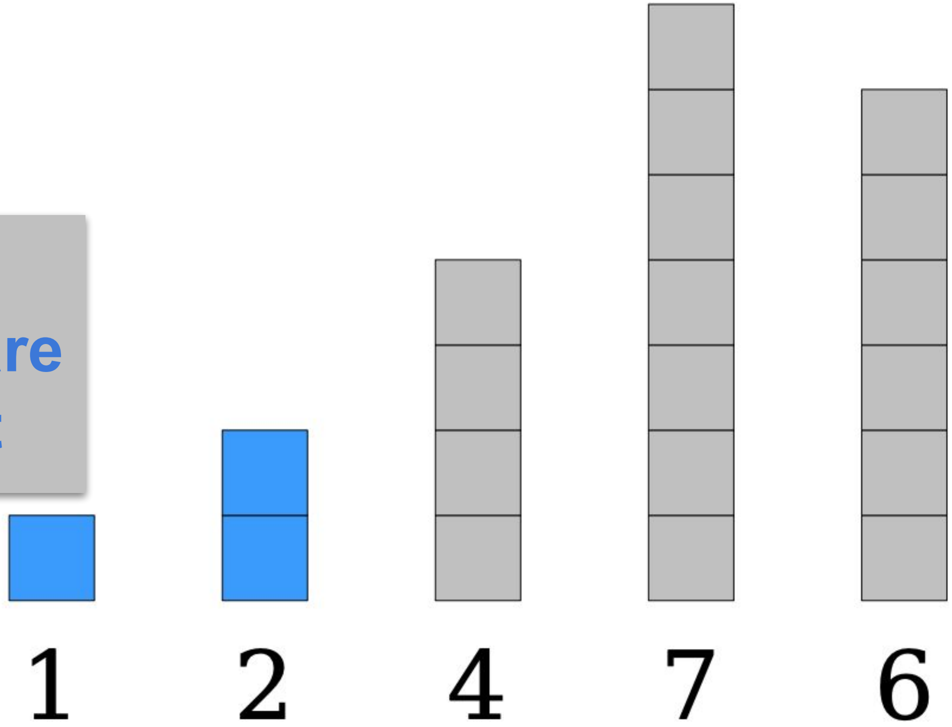
Our first sort: Selection sort



Our first sort: Selection sort



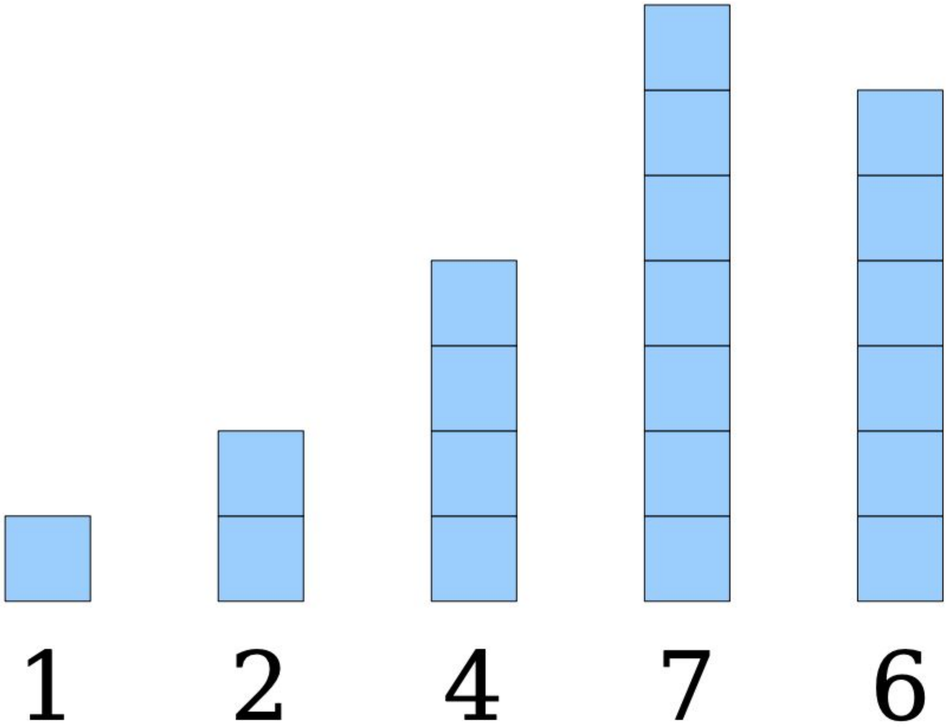
Our first sort: Selection sort



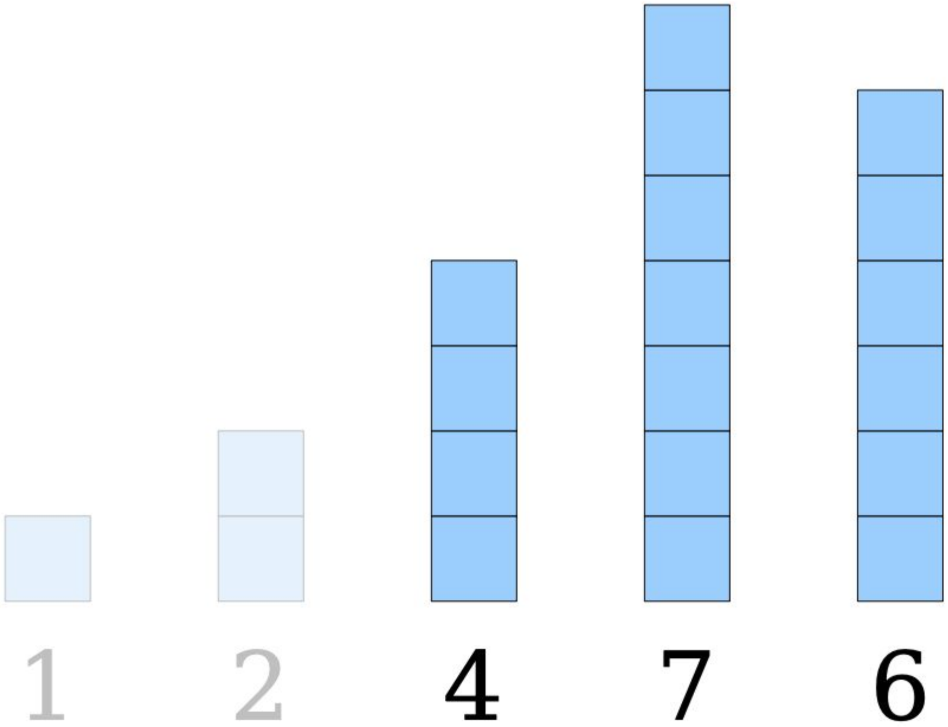
These elements are in the right place now.

The remaining elements are in no particular order.

Our first sort: Selection sort



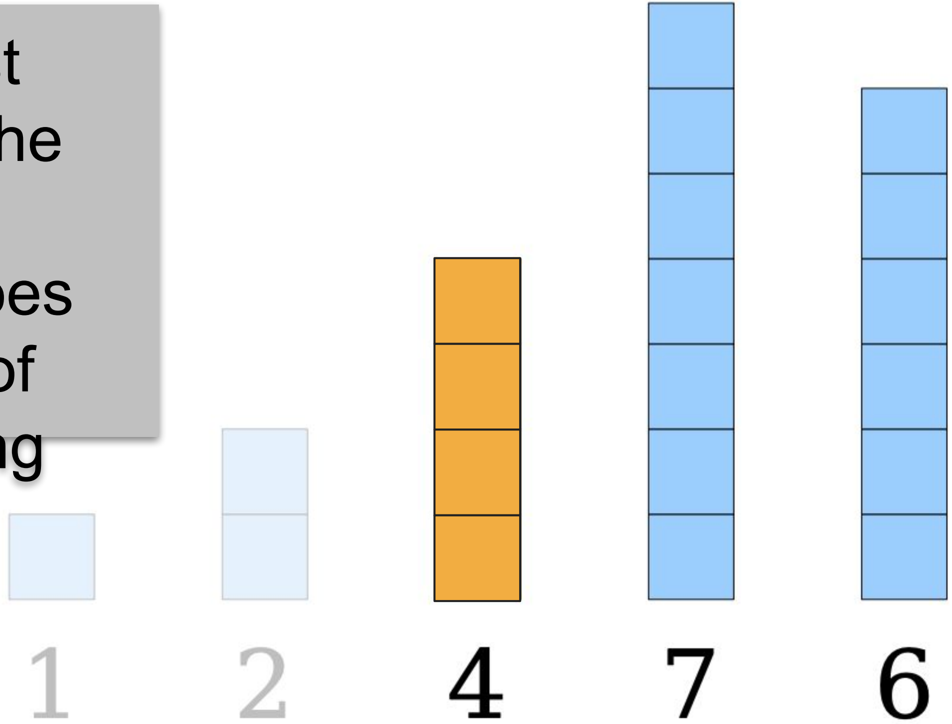
Our first sort: Selection sort



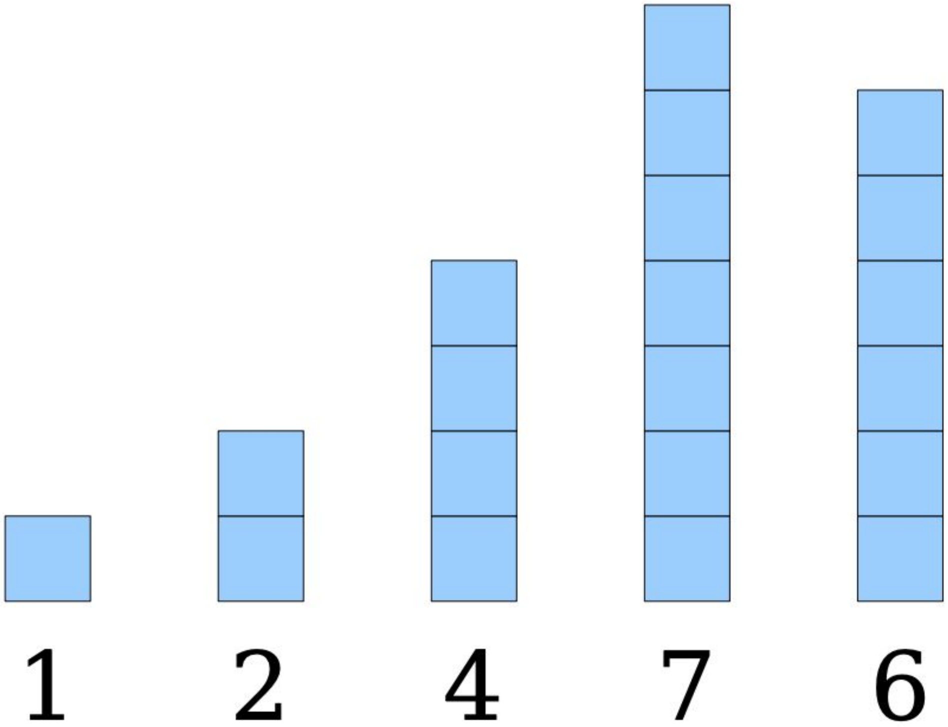
Visualizations created by

Our first sort: Selection sort

The smallest element of the remaining elements goes at the front of the remaining elements.

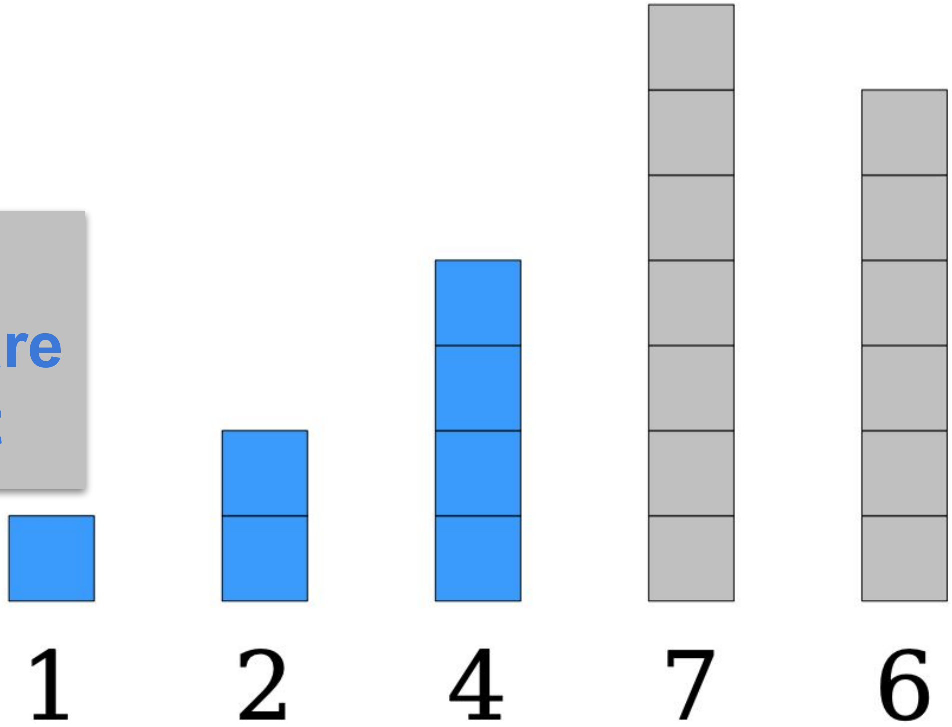


Our first sort: Selection sort



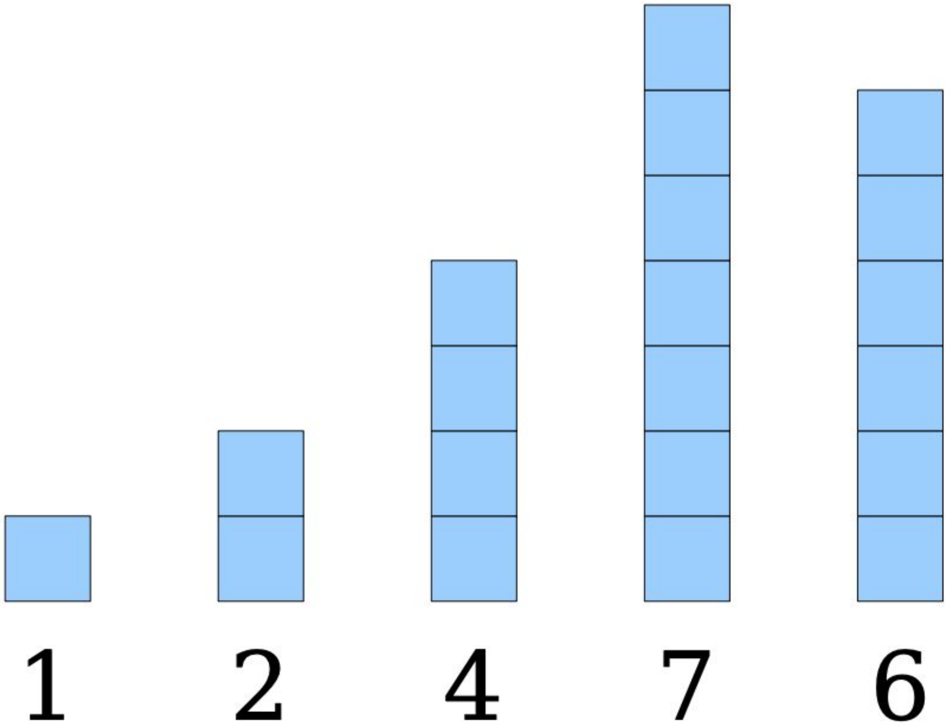
Our first sort: Selection sort

These elements are in the right place now.

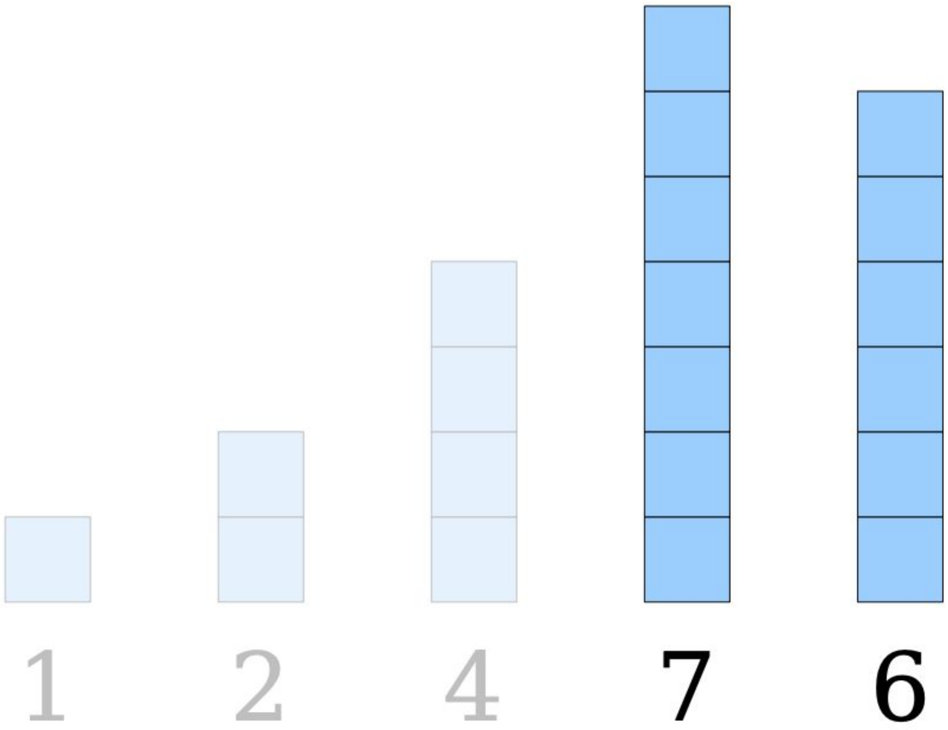


The remaining elements are in no particular order.

Our first sort: Selection sort

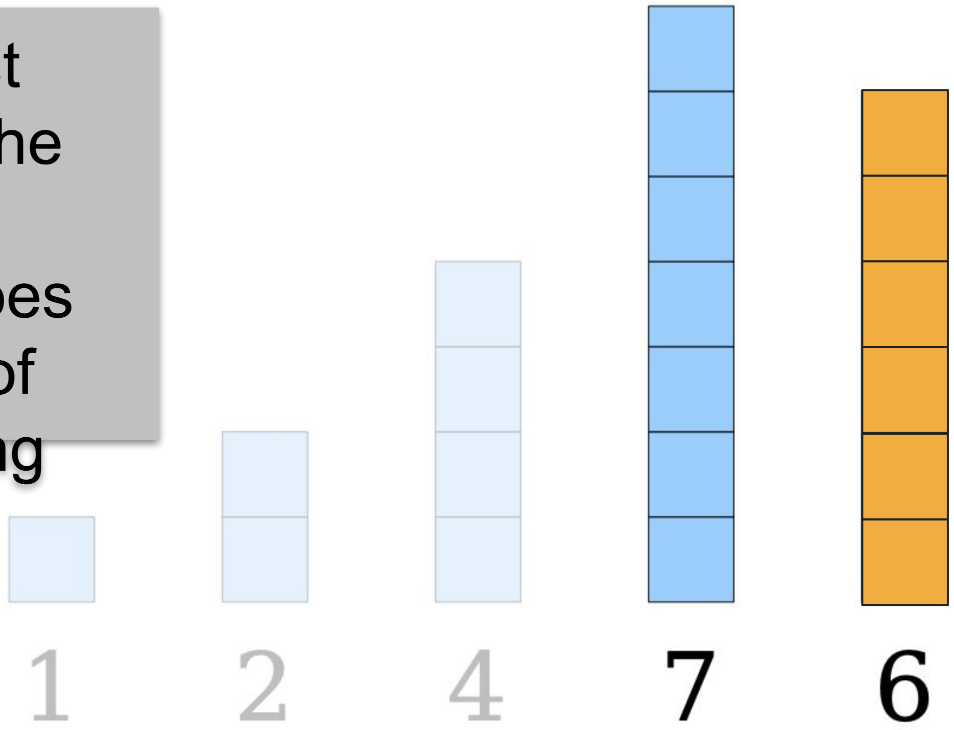


Our first sort: Selection sort

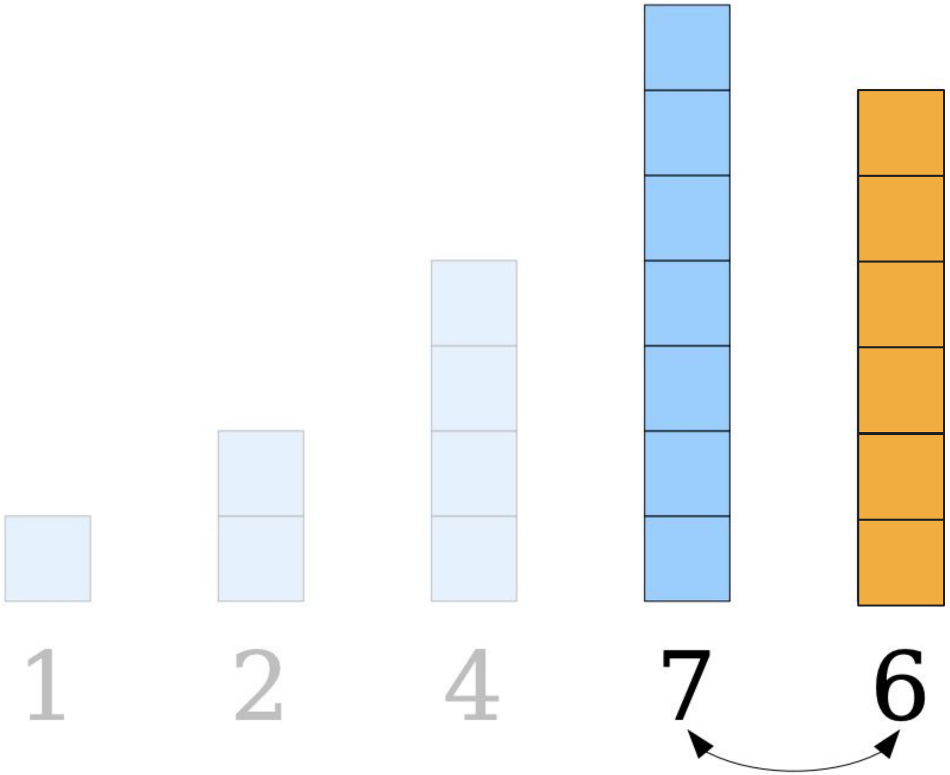


Our first sort: Selection sort

The smallest element of the remaining elements goes at the front of the remaining elements.

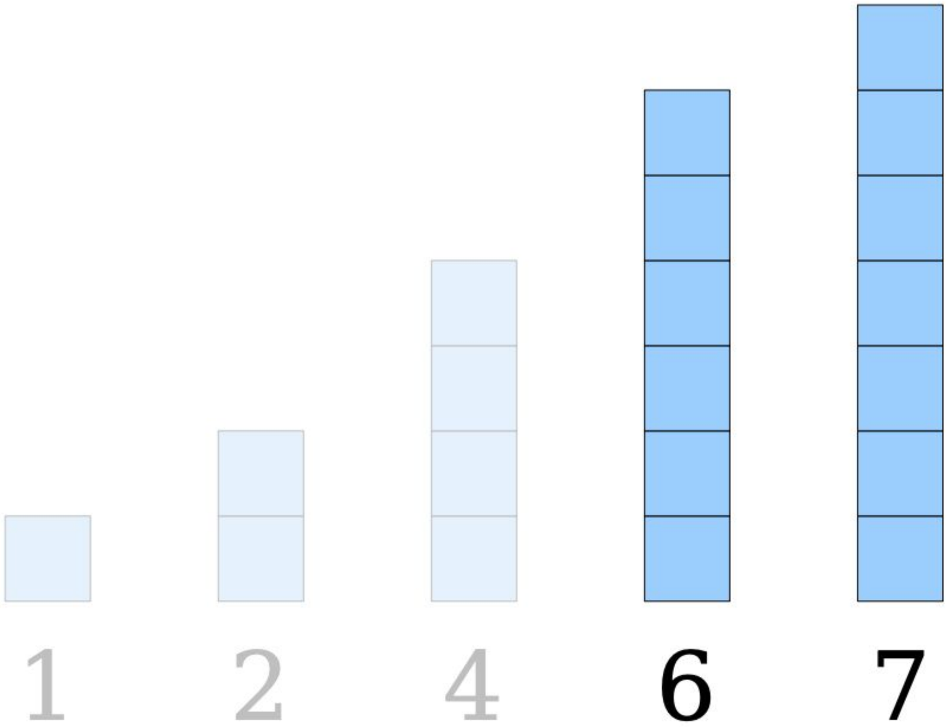


Our first sort: Selection sort



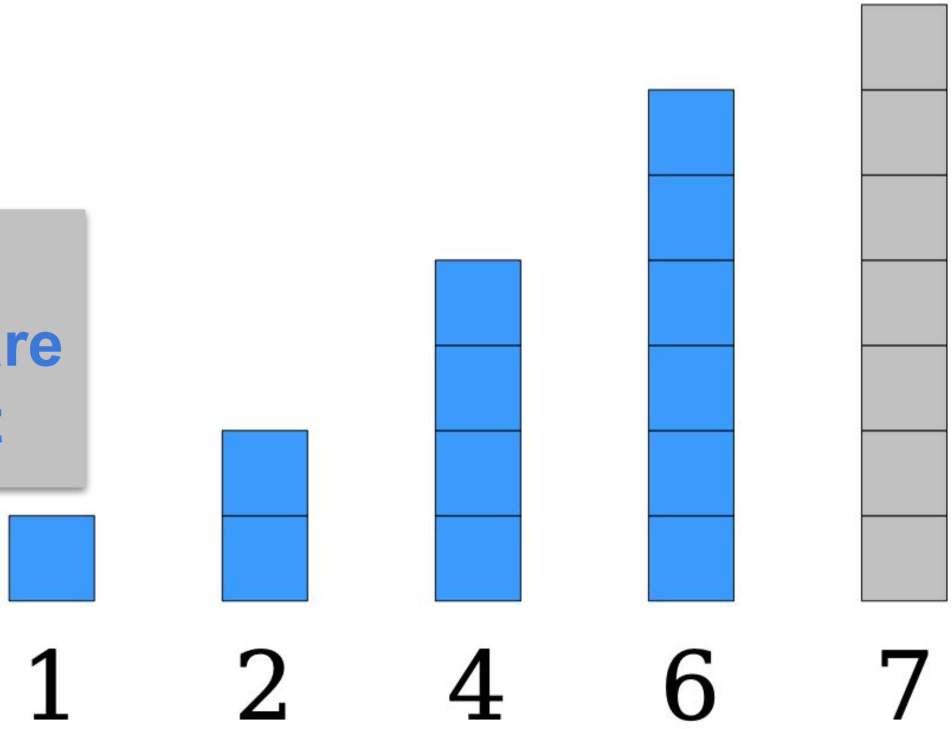
Visualizations created by

Our first sort: Selection sort



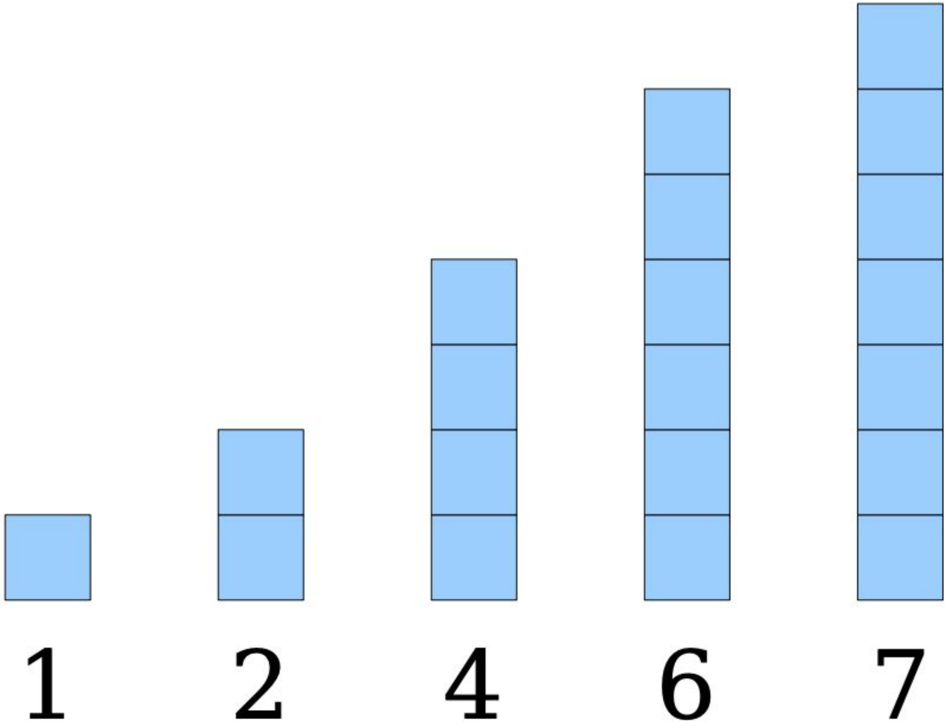
Our first sort: Selection sort

These elements are in the right place now.

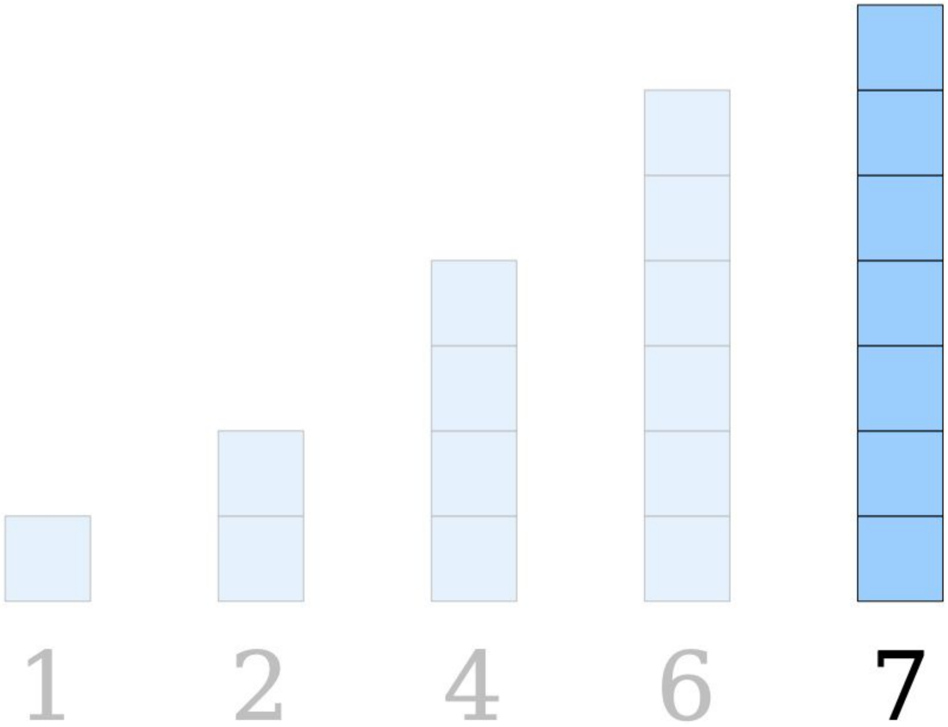


The remaining elements are in no particular order.

Our first sort: Selection sort

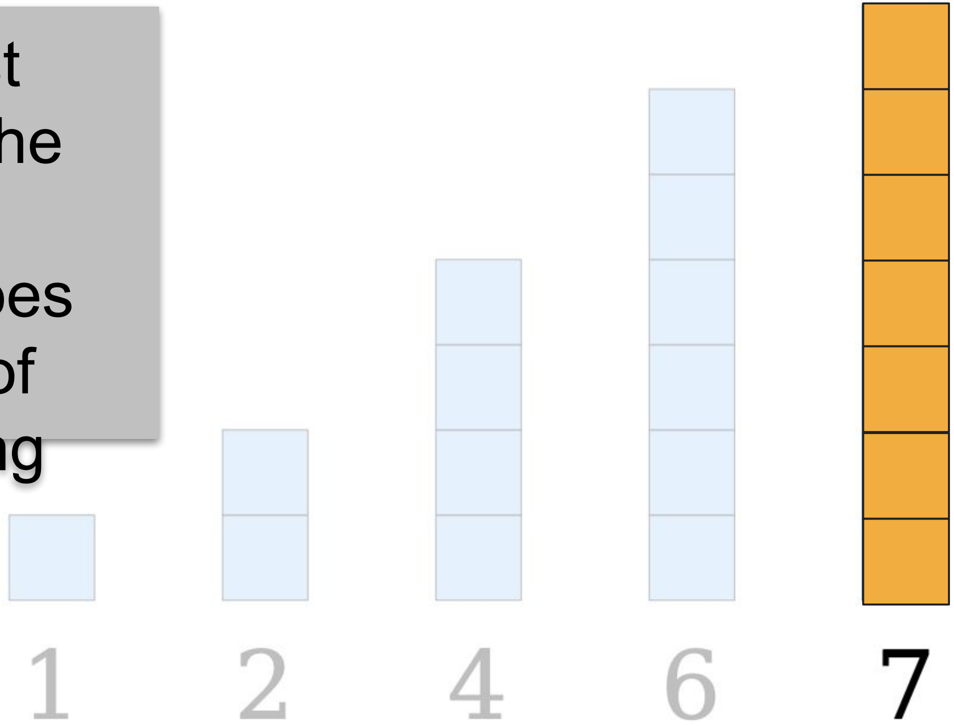


Our first sort: Selection sort

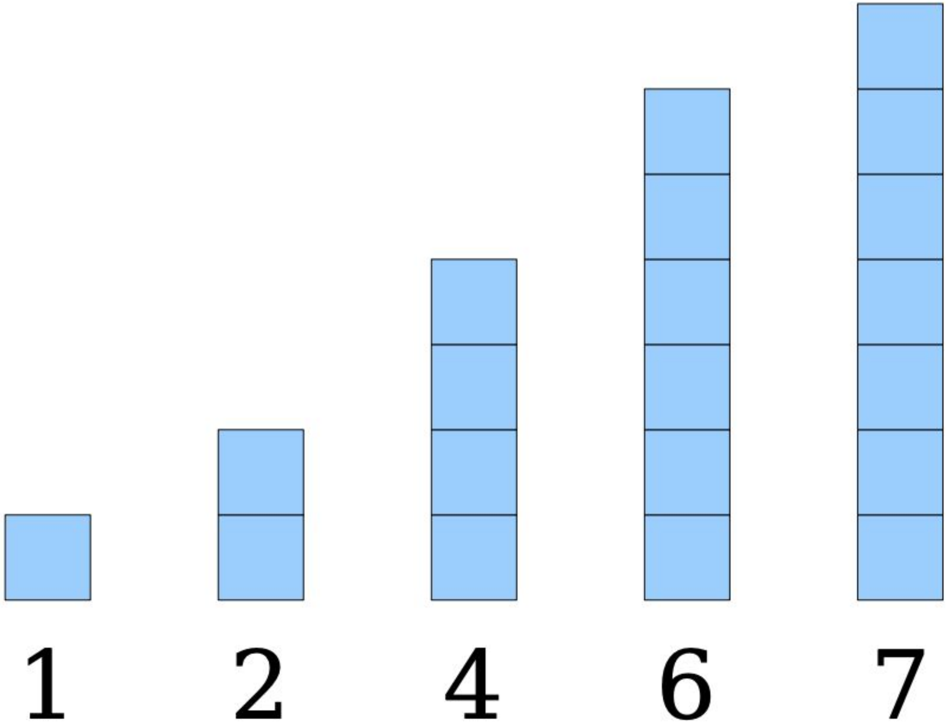


Our first sort: Selection sort

The smallest element of the remaining elements goes at the front of the remaining elements.

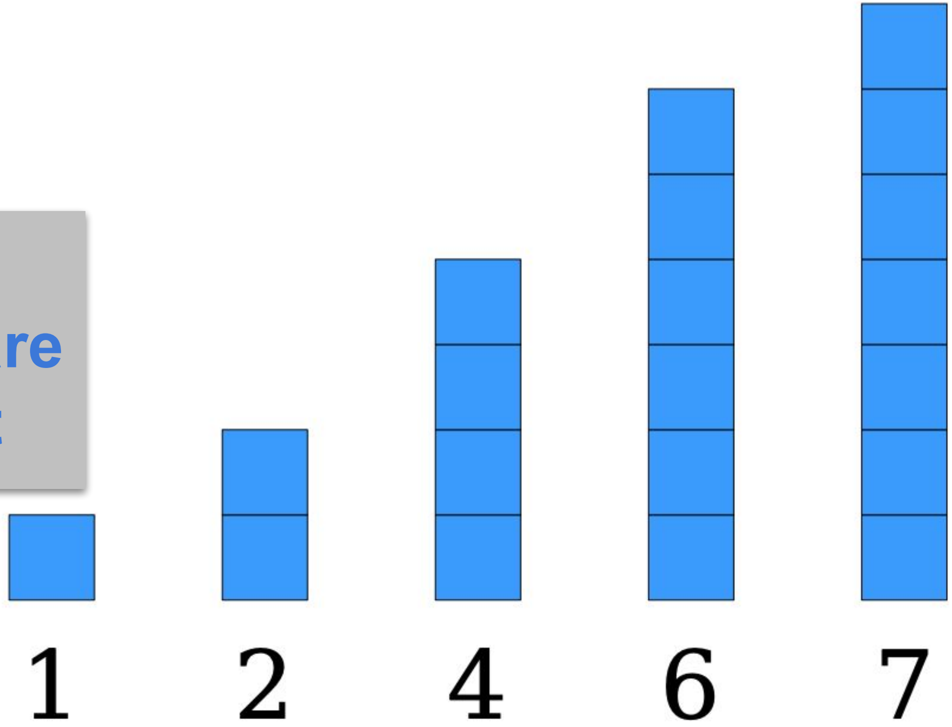


Our first sort: Selection sort



Our first sort: Selection sort

These elements are in the right place now.



Selection sort algorithm

- Find the smallest element and move it to the first position.
- Find the smallest element of what’s left and move it to the second position.
- Find the smallest element of what’s left and move it to the third position.
- Find the smallest element of what’s left and move it to the fourth position.
- (etc.)

```
void selectionSort(Vector<int>& elems) {
    for (int index = 0; index < elems.size(); index++) {
        int smallestIndex = indexOfSmallest(elems, index);
        swap(elems[index], elems[smallestIndex]);
    }
}
/**
* Given a vector and a starting point, returns the index of the smallest
* element in that vector at or after the starting point
*/
int indexOfSmallest(const Vector<int>& elems, int startPoint) {
    int smallestIndex = startPoint;
    for (int i = startPoint + 1; i < elems.size(); i++) {
        if (elems[i] < elems[smallestIndex]) {
            smallestIndex = i;
        }
    }
    return smallestIndex;
}
```

Analyzing selection sort

- How much work do we do for selection sort?

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all **`n`** elements.

Analyzing selection sort

- How much work do we do for selection sort?
    - To find the smallest value, we need to look at all **`n`** elements.
    - To find the second-smallest value, we need to look at **`n - 1`** elements.

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all **`n`** elements.
  - To find the second-smallest value, we need to look at **`n - 1`** elements.
  - To find the third-smallest value, we need to look at **`n - 2`** elements.

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all **`n`** elements.
  - To find the second-smallest value, we need to look at **`n - 1`** elements.
  - To find the third-smallest value, we need to look at **`n - 2`** elements.
  - This process continues until we have found every last "smallest element" from the original collection.

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all **`n`** elements.
  - To find the second-smallest value, we need to look at **`n - 1`** elements.
  - To find the third-smallest value, we need to look at **`n - 2`** elements.
  - This process continues until we have found every last "smallest element" from the original collection.

- This, the total amount of work we have to do is is **`n + (n - 1) + (n - 2) + … + 1`**

The complexity of selection sort

- There is a mathematical formula that tells us

  `n + (n-1) + ... + 2 + 1 = (n * (n+1)) / 2`

- Thus, the overall complexity of selection sort can be simplified as follows:
  - Total work = `O((n * (n+1)) / 2)`

The complexity of selection sort

- There is a mathematical formula that tells us

```
n + (n-1) + ... + 2 + 1 = (n * (n+1)) / 2
```

The complexity of selection sort

- There is a mathematical formula that tells us

  `n + (n-1) + ... + 2 + 1 = (n * (n+1)) / 2`

- Thus, the overall complexity of selection sort can be simplified as follows:
  - Total work = `O((n * (n+1)) / 2)`

    `= O(n * (n+1))` ← Big-O ignores constant factors

The complexity of selection sort

- There is a mathematical formula that tells us

$$n + (n-1) + ... + 2 + 1 = (n * (n+1)) / 2$$

- Thus, the overall complexity of selection sort can be simplified as follows:
  - Total work = $O((n * (n+1)) / 2)$

$$= O(n * (n+1))$$

Big-O ignores constant factors

$$= O(n^2 + n)$$

The complexity of selection sort

- There is a mathematical formula that tells us

$$n + (n-1) + ... + 2 + 1 = (n * (n+1)) / 2$$

- Thus, the overall complexity of selection sort can be simplified as follows:
  - Total work = $O((n * (n+1)) / 2)$

$= O(n * (n+1))$ ← Big-O ignores constant factors

$= O(n^2 + n)$

$= O(n^2)$ ← Big-O ignores low-order terms

Selection sort takeaways

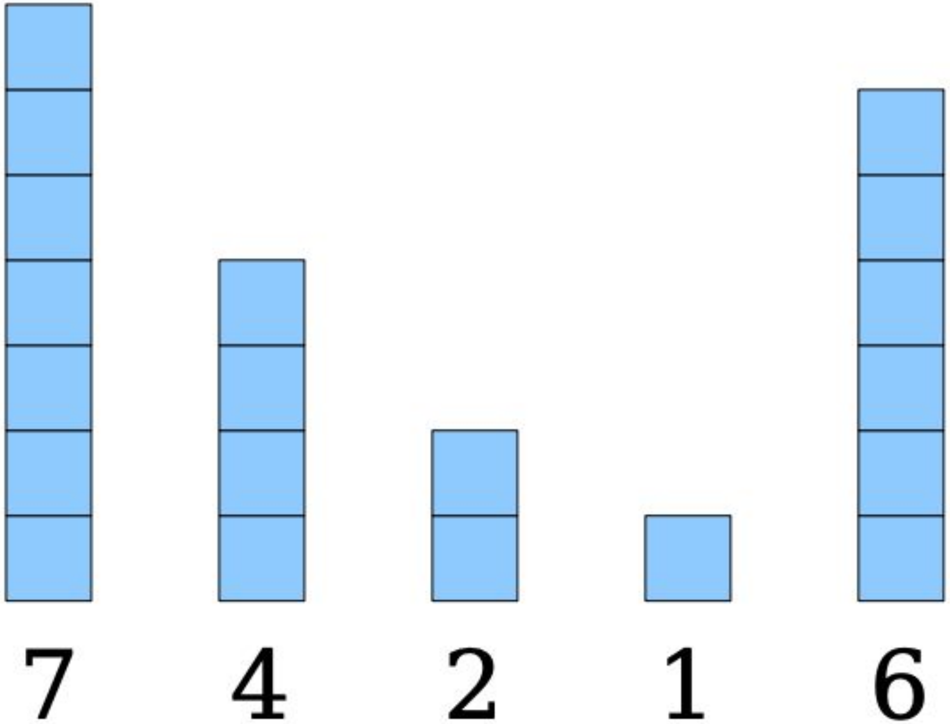
- Selection sort works by "selecting" the smallest remaining element in the list and putting it in the front of all remaining elements.

- Selection sort is an $O(n^2)$ algorithm.

- Can we do better?
  - Yes!

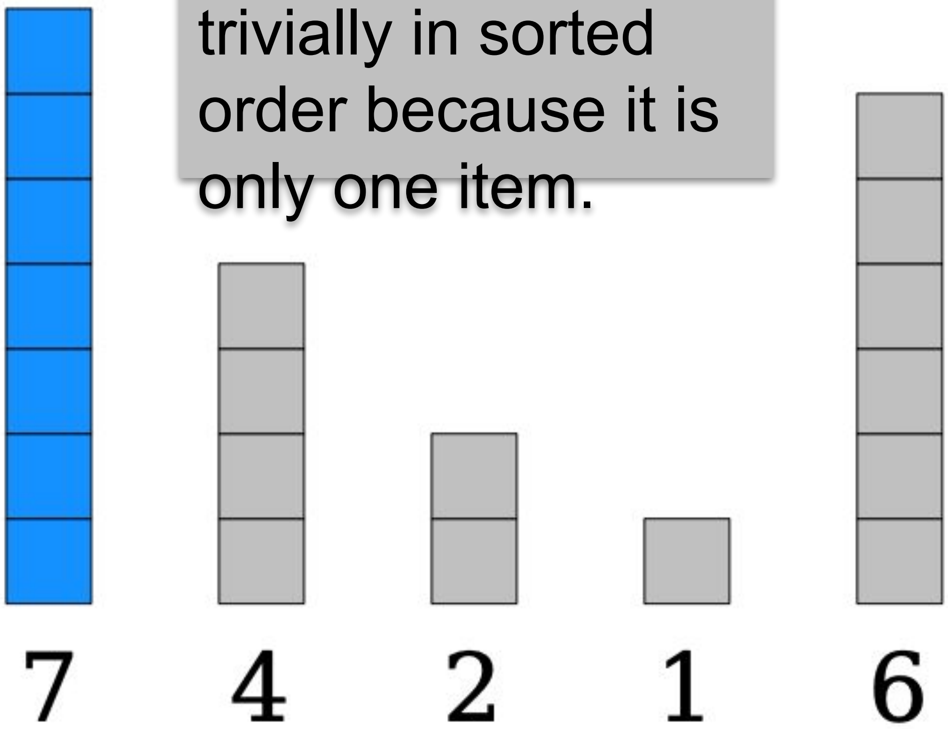
Another take on sorting

Insertion sort



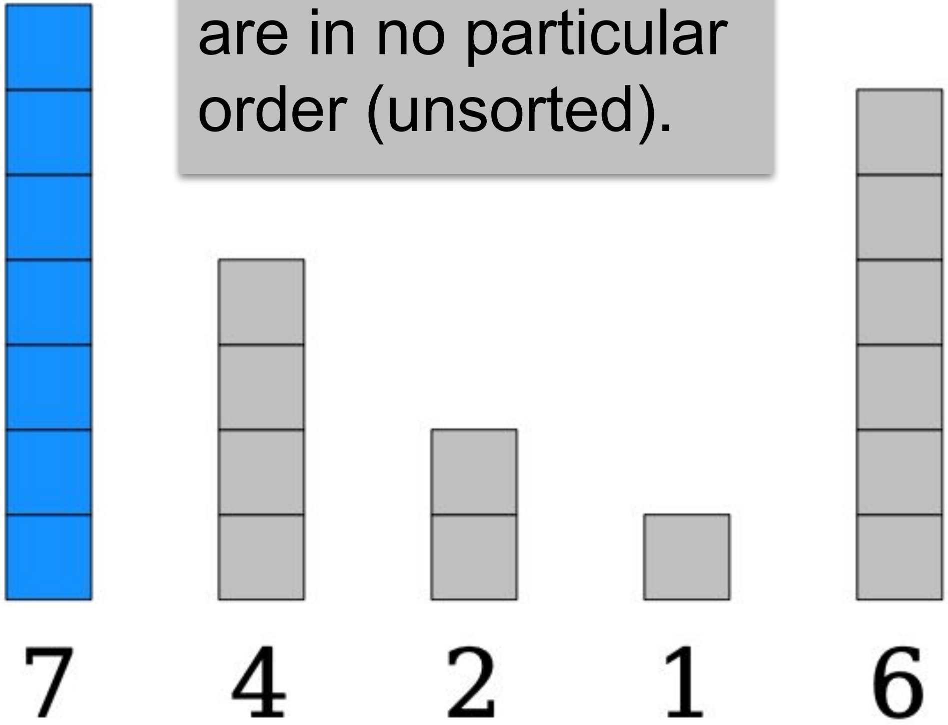
Insertion sort

Considered alone, the blue item is trivially in sorted order because it is only one item.



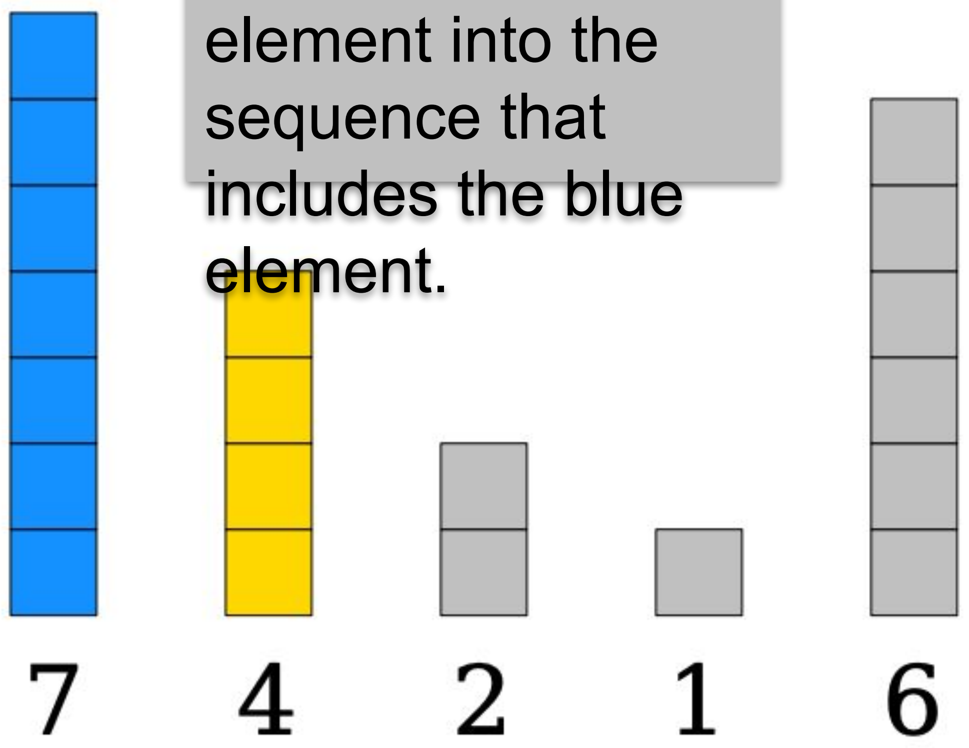
Insertion sort

The items in gray are in no particular order (unsorted).

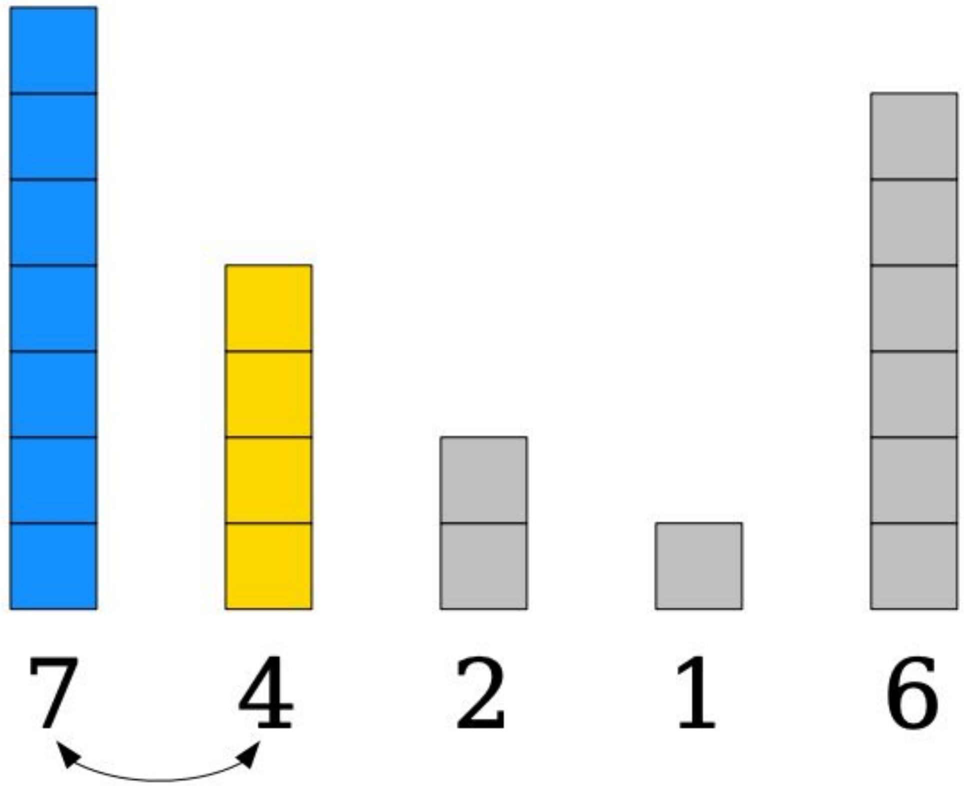


Insertion sort

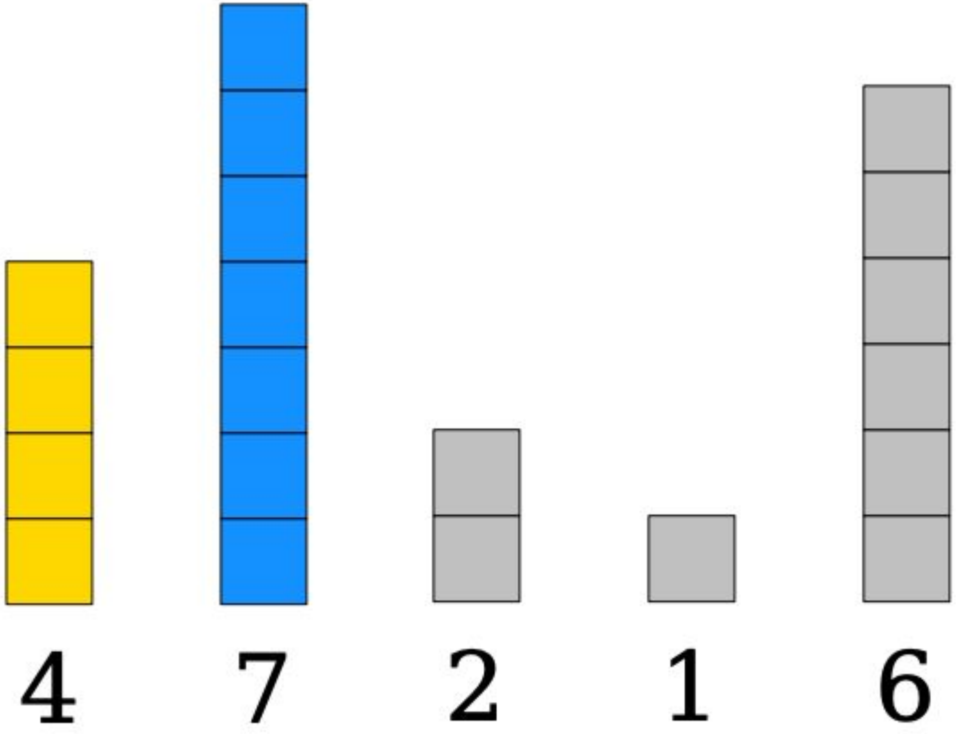
Insert the yellow element into the sequence that includes the blue element.



Insertion sort

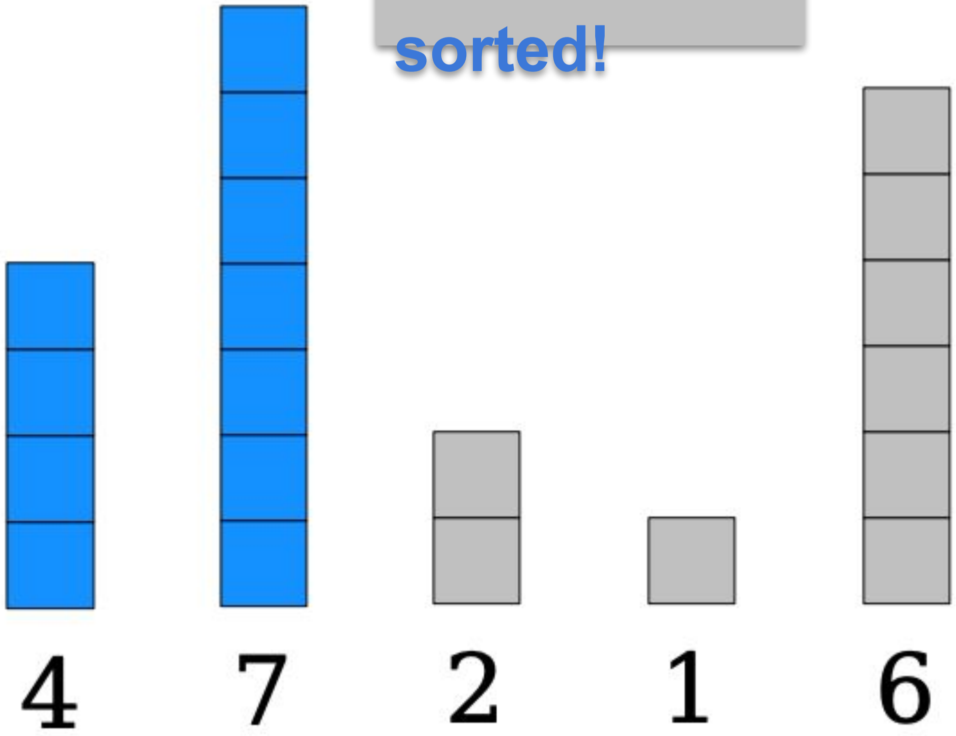


Insertion sort



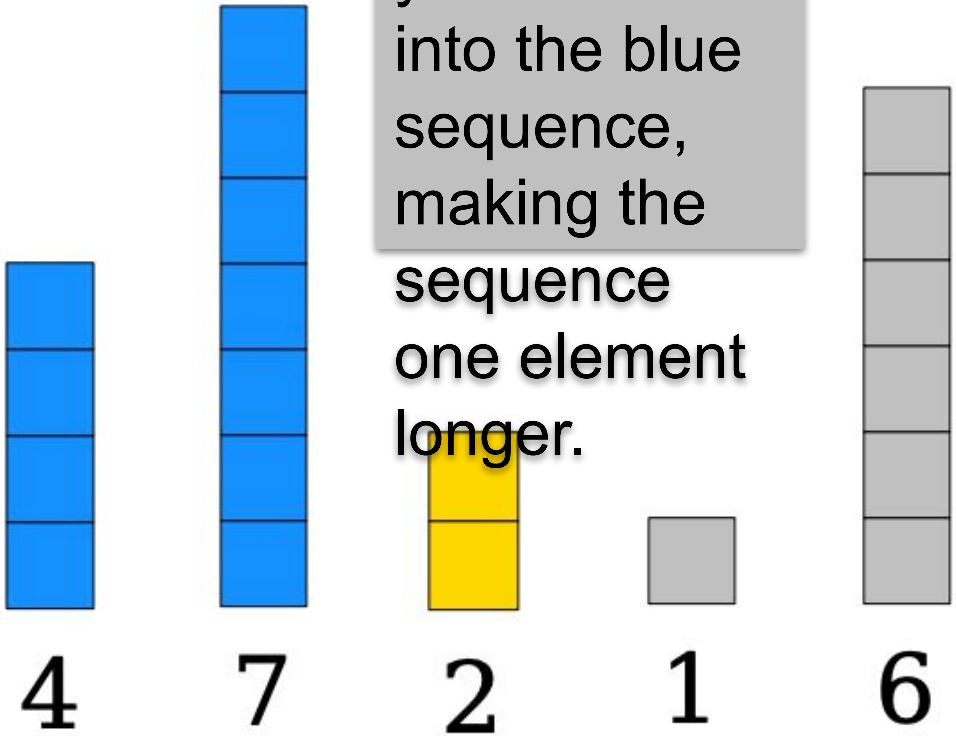
Insertion sort

The blue elements are sorted!

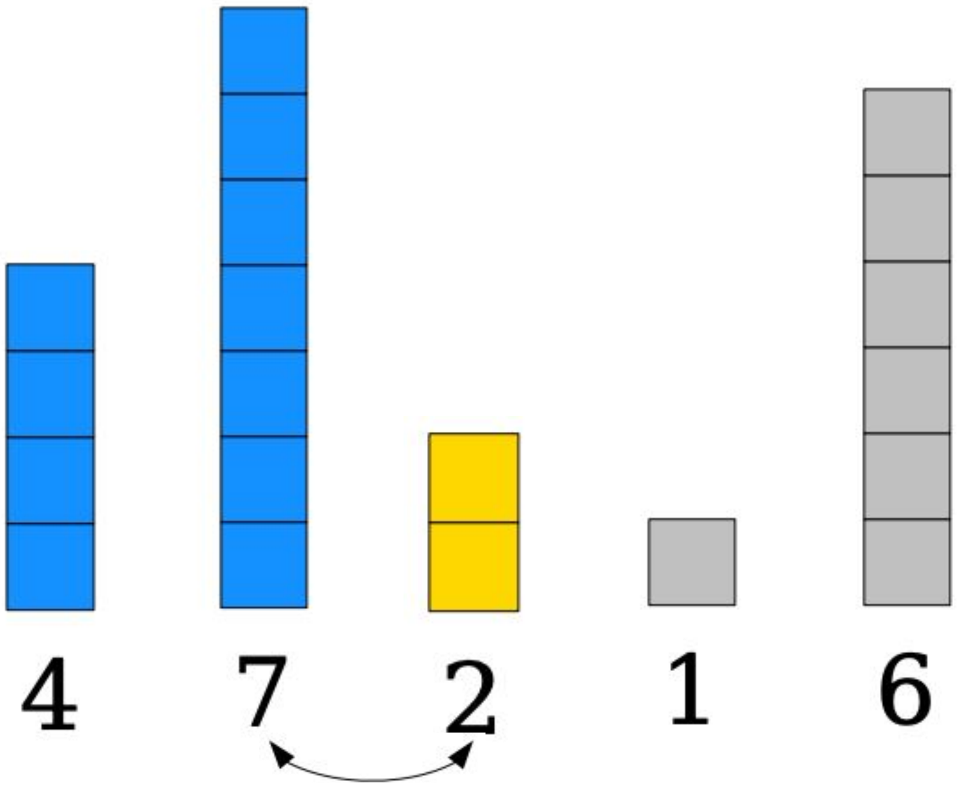


Insertion sort

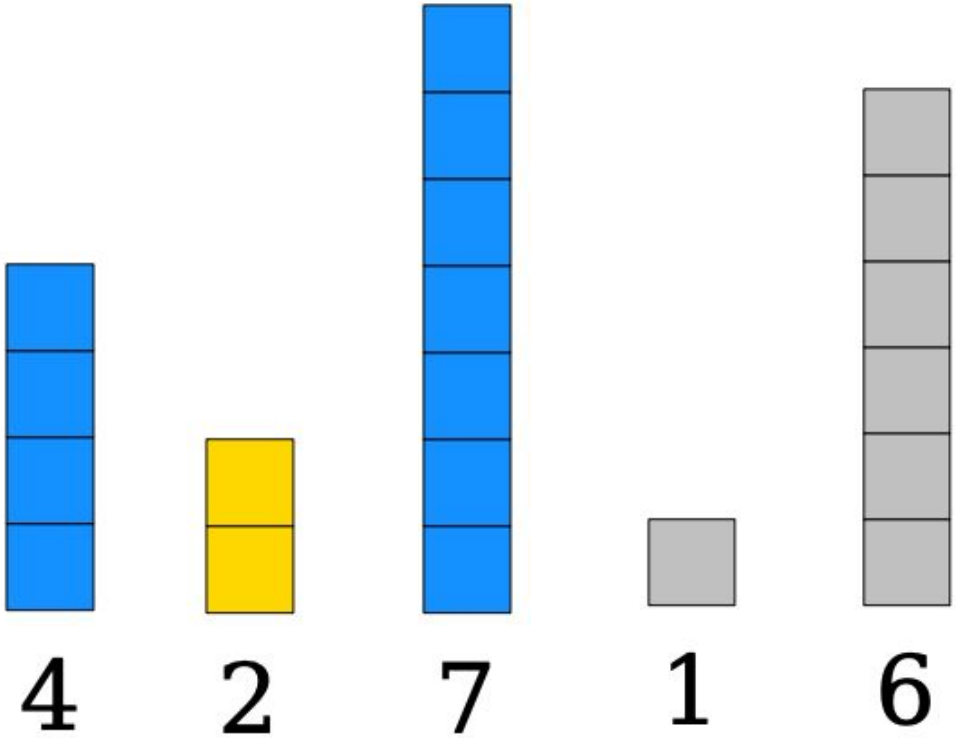
Insert the yellow item into the blue sequence, making the sequence one element longer.



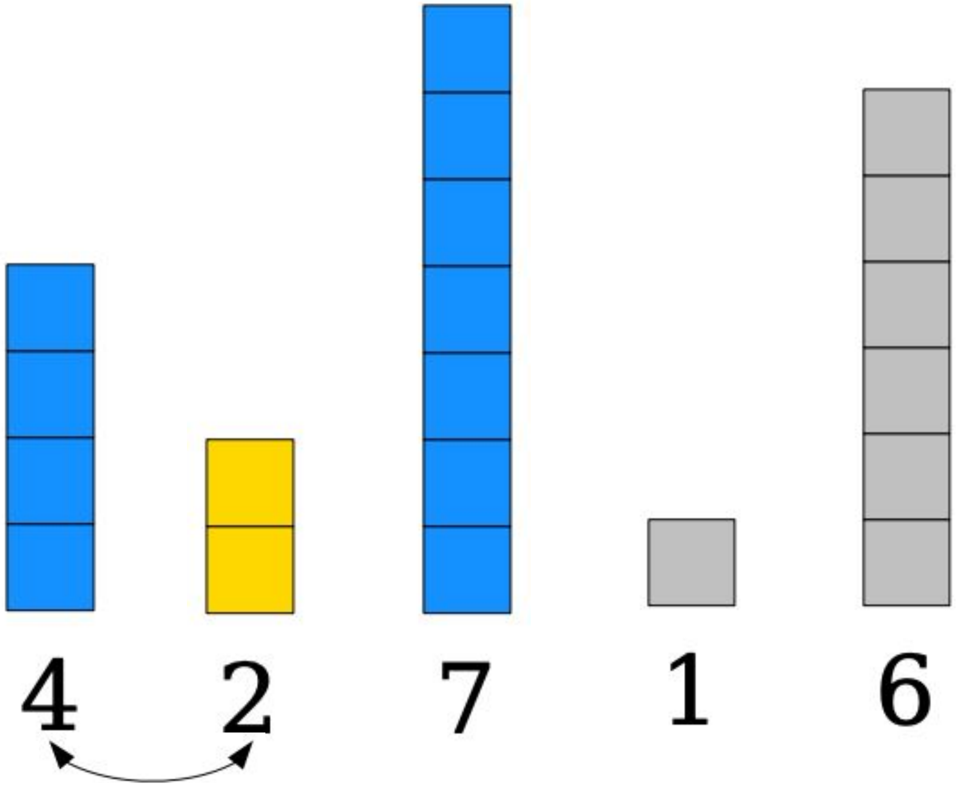
Insertion sort



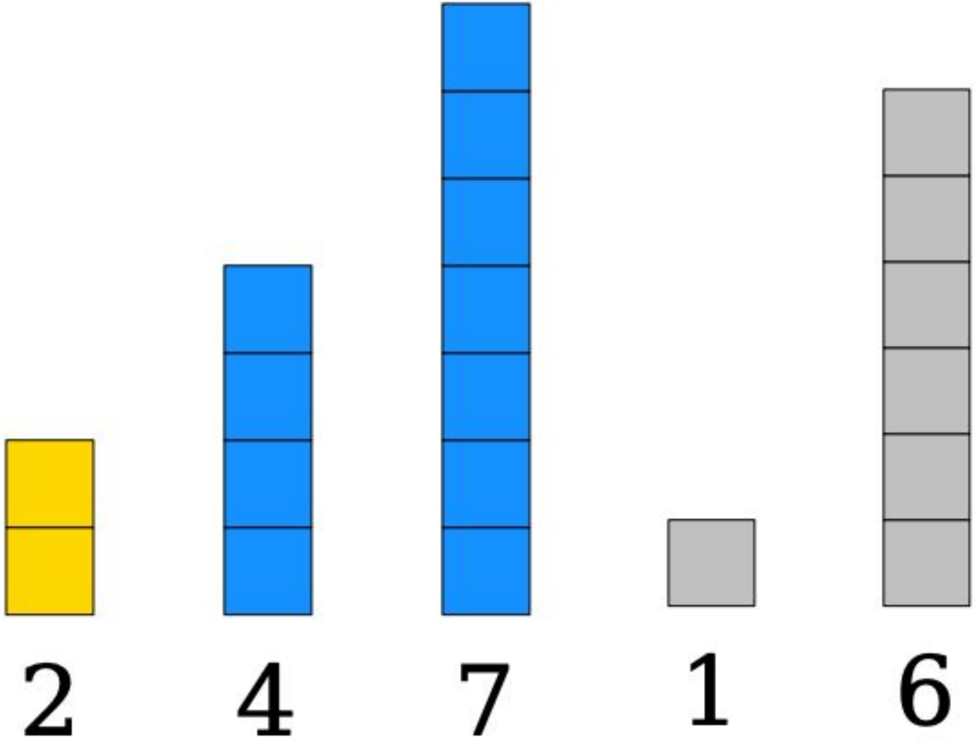
Insertion sort



Insertion sort

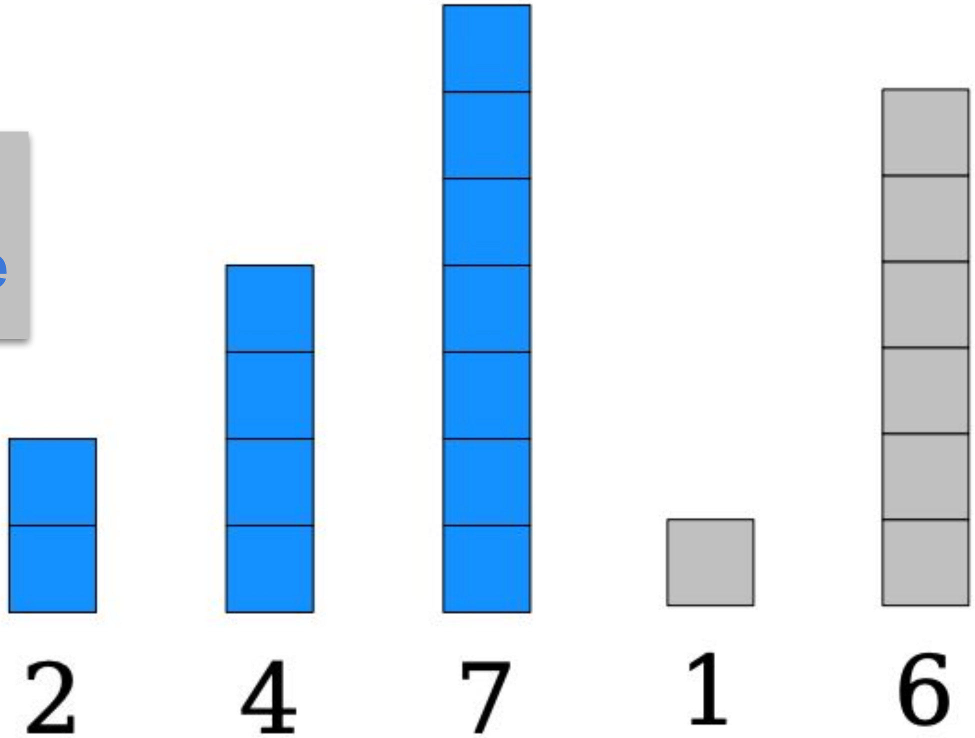


Insertion sort



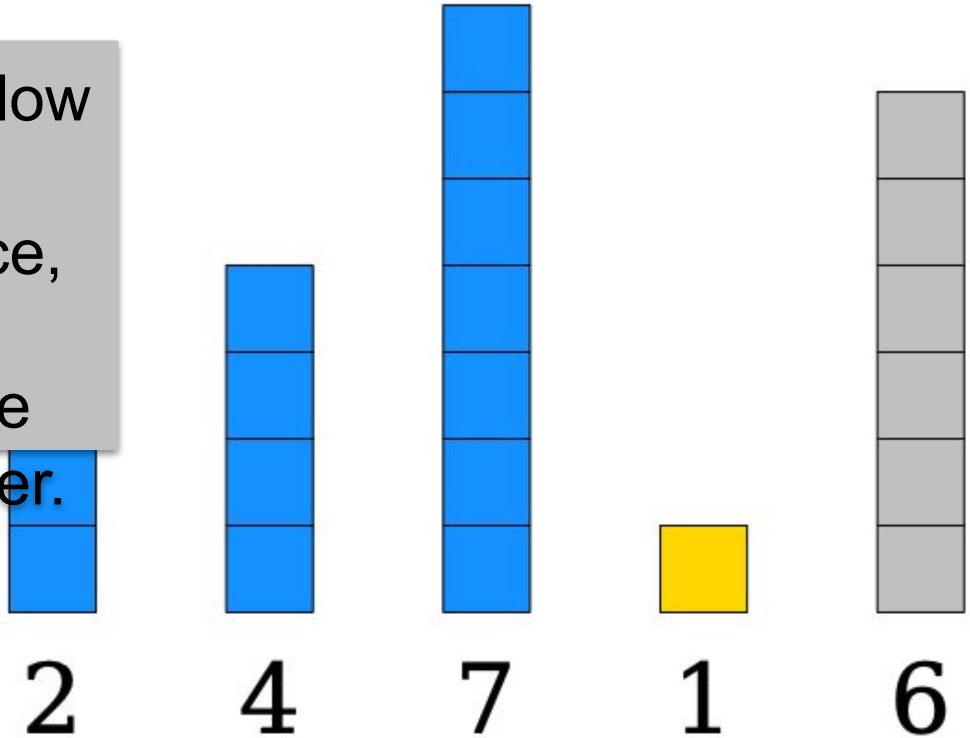
Insertion sort

The blue elements are sorted!

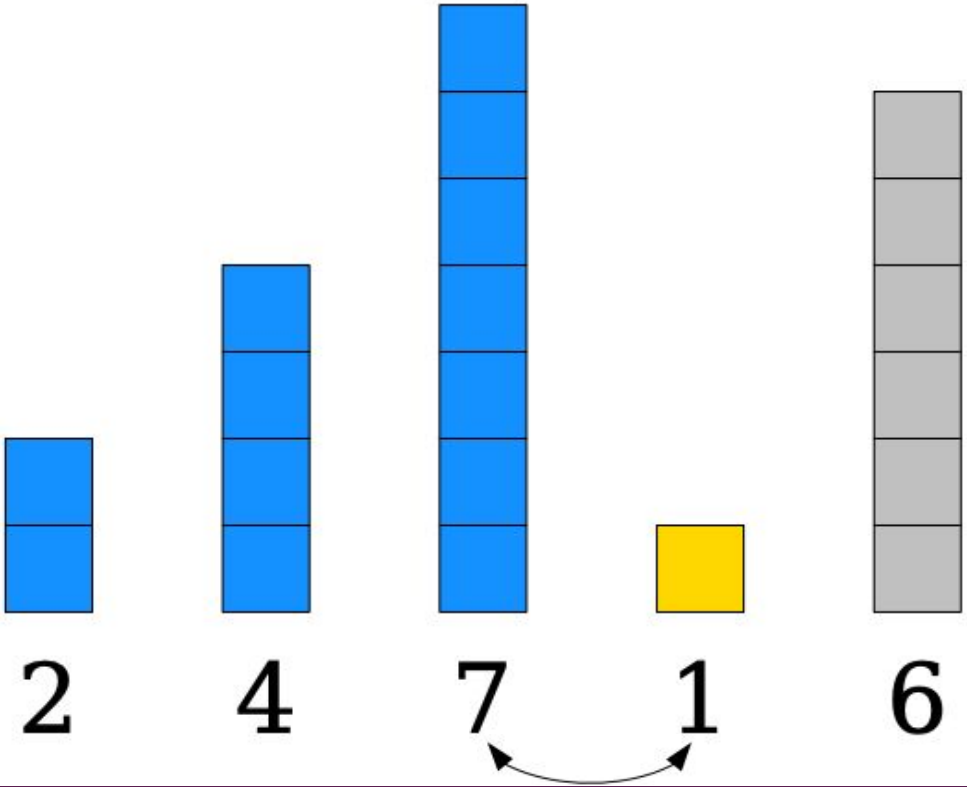


Insertion sort

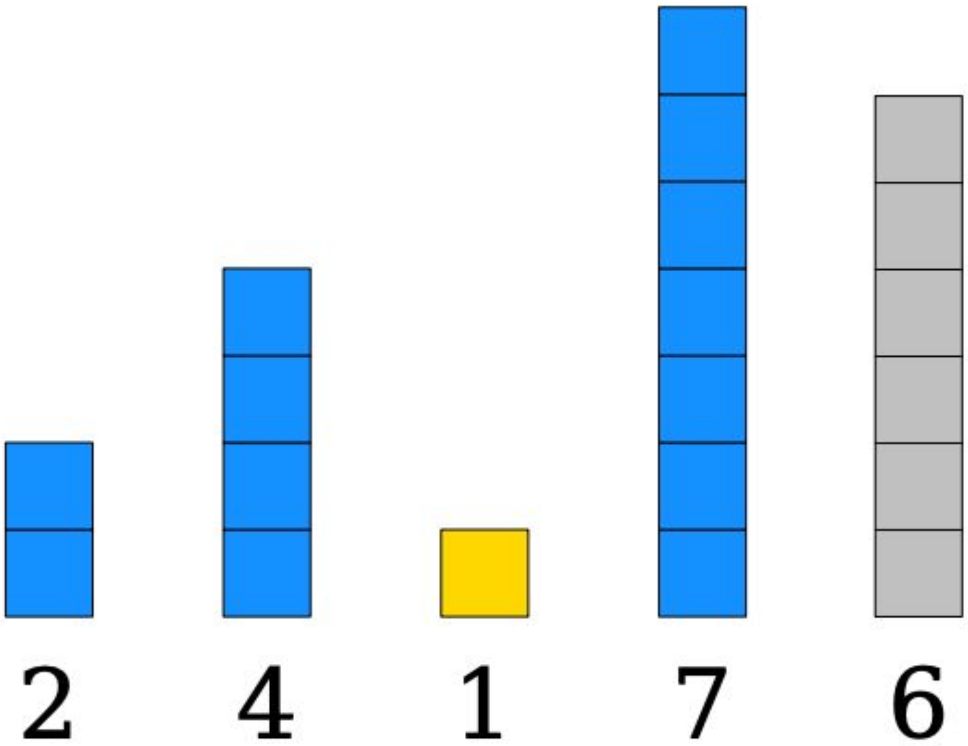
Insert the yellow item into the blue sequence, making the sequence one element longer.



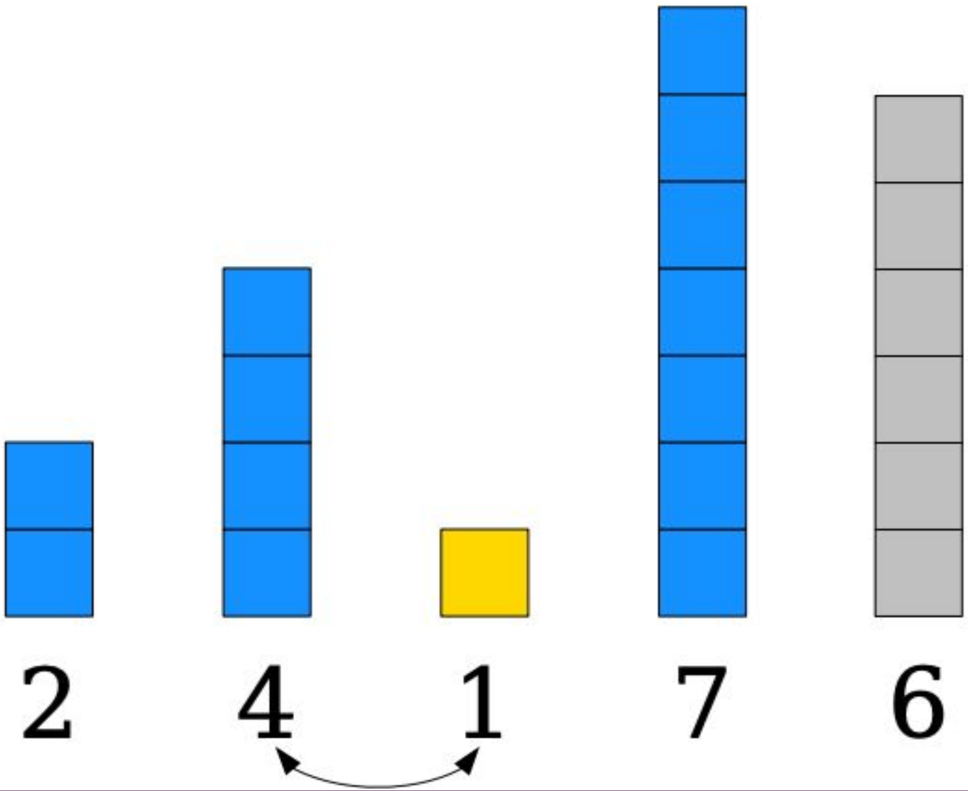
Insertion sort



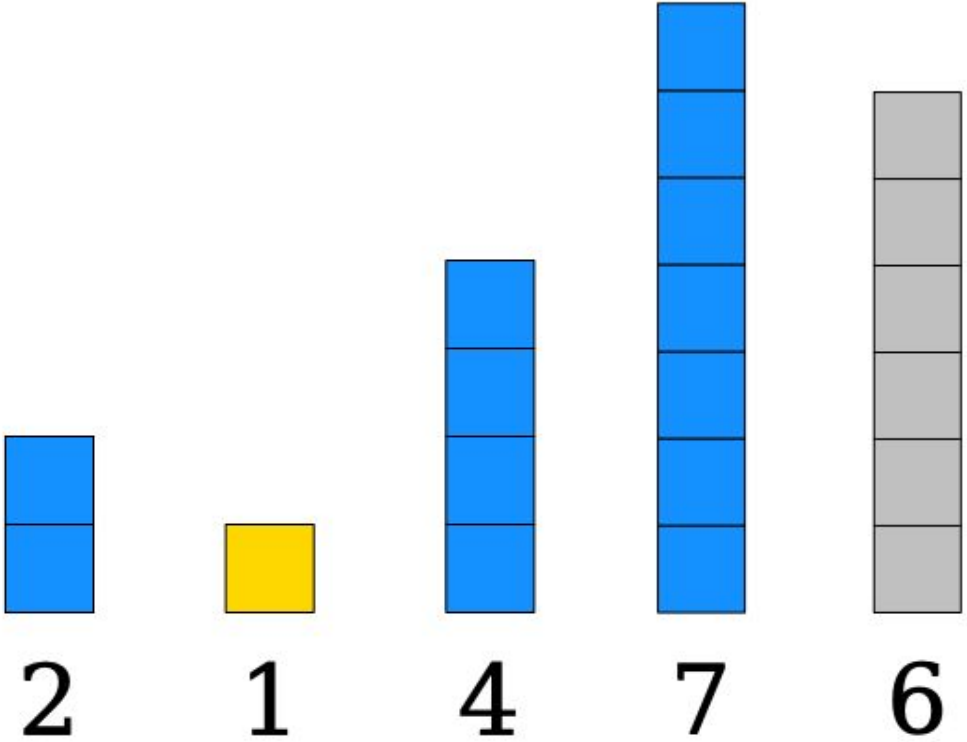
Insertion sort



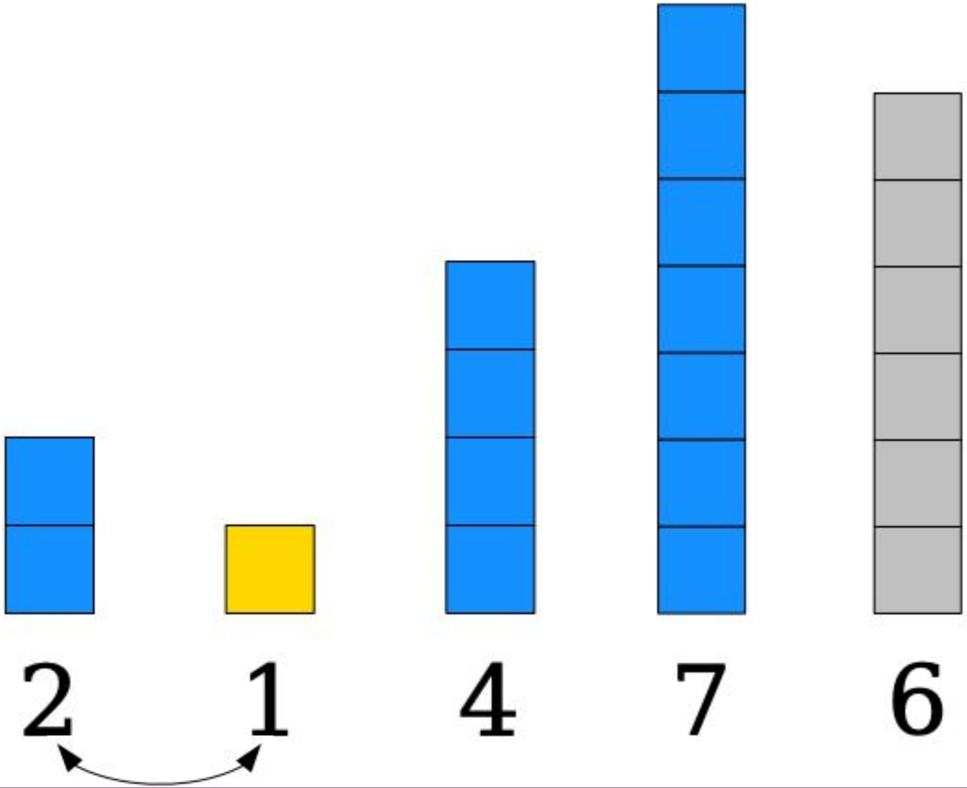
Insertion sort



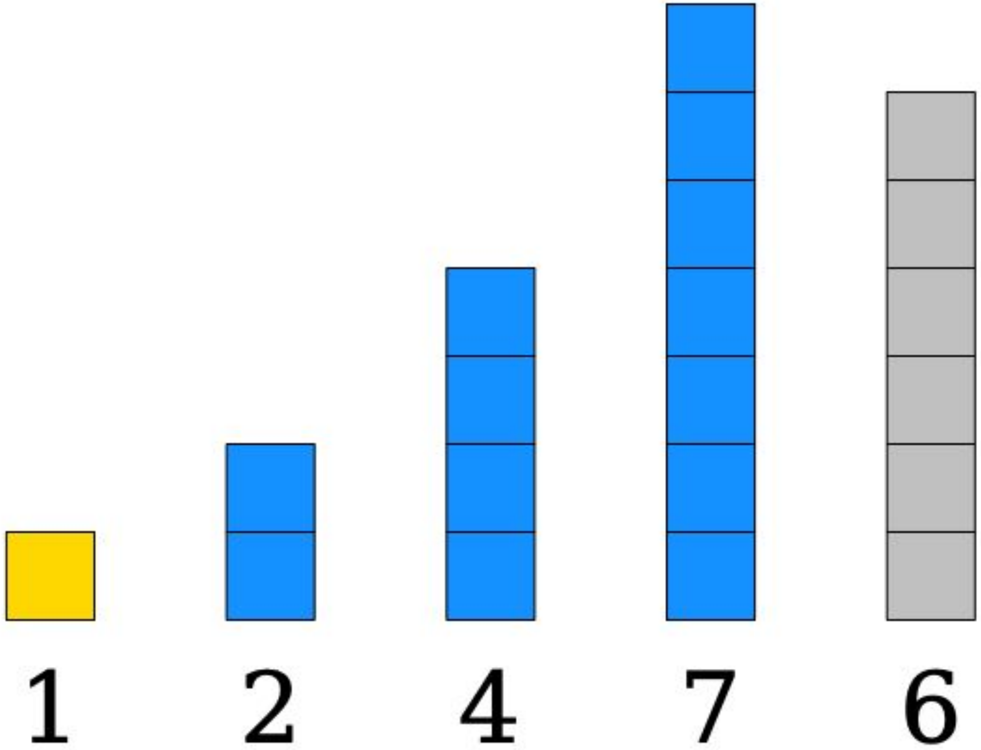
Insertion sort



Insertion sort

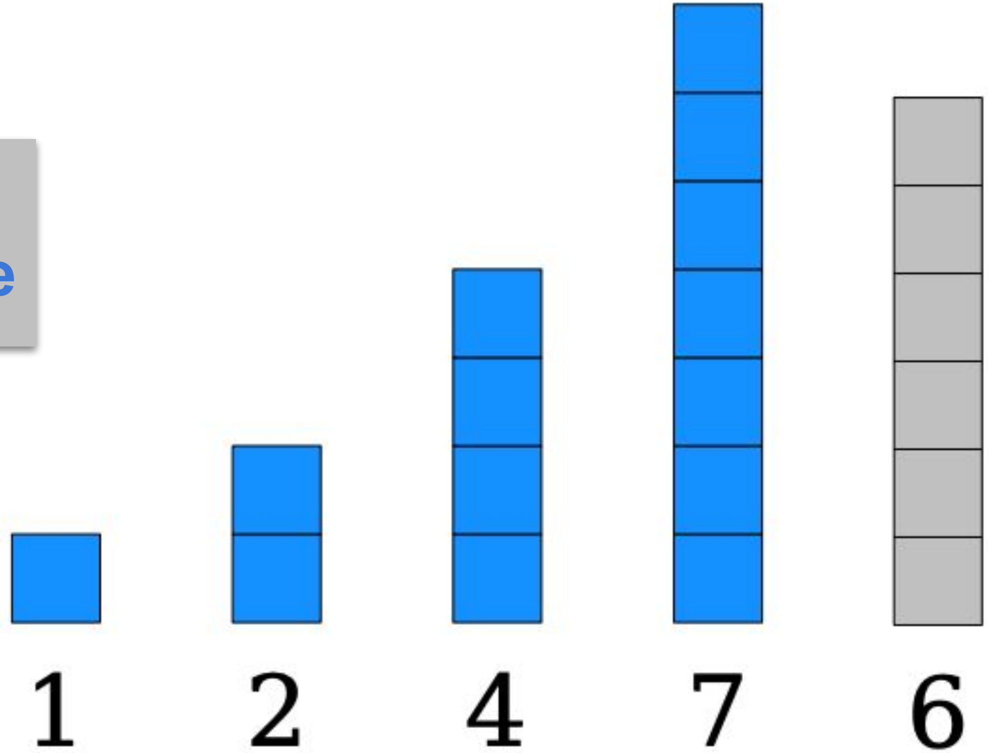


Insertion sort



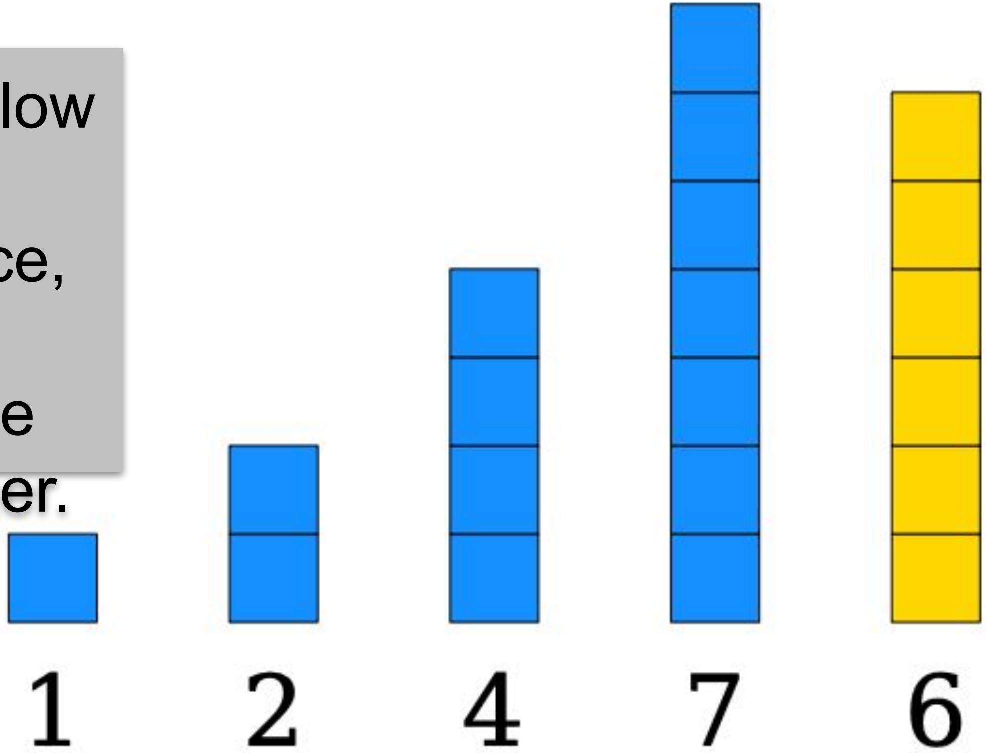
Insertion sort

The blue elements are sorted!

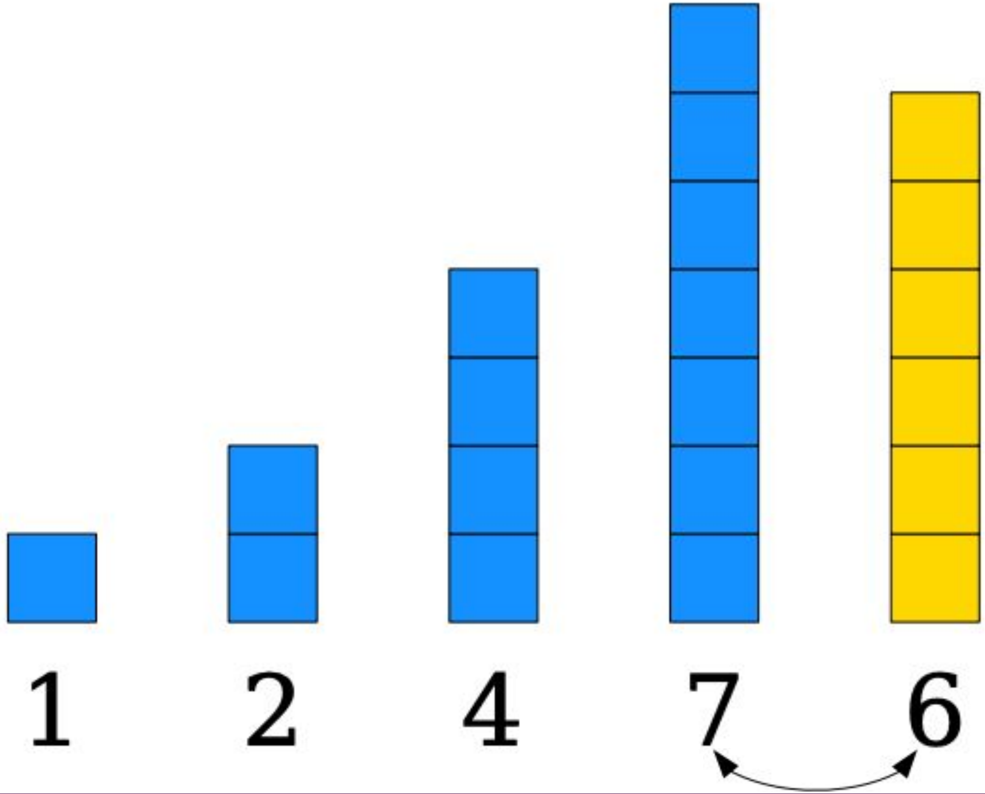


Insertion sort

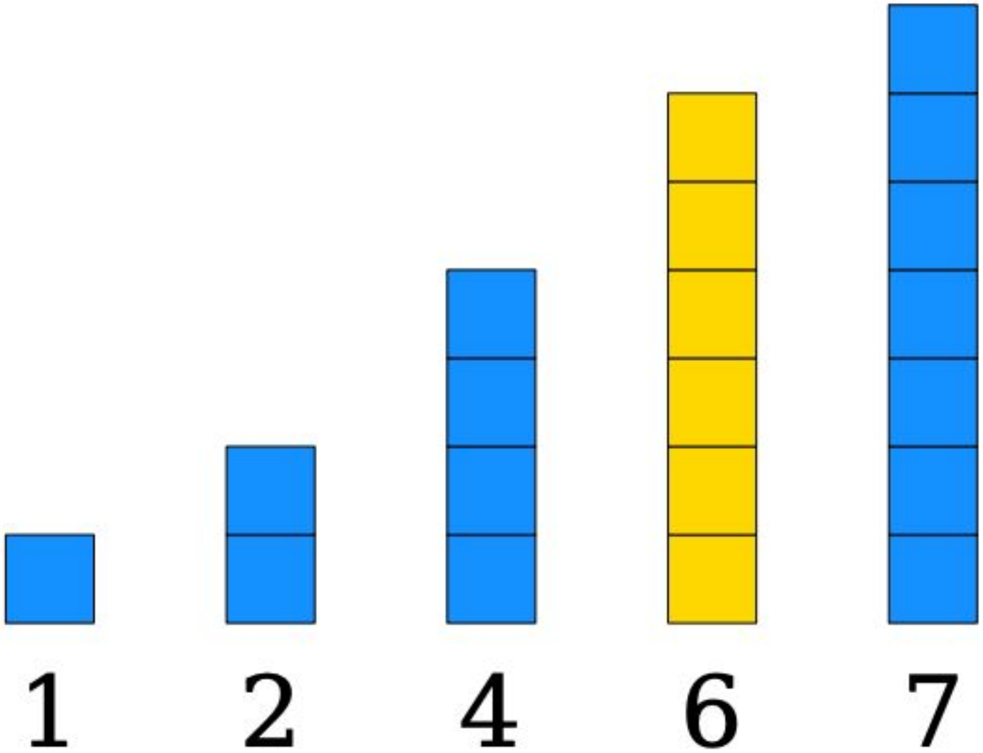
Insert the yellow item into the blue sequence, making the sequence one element longer.



Insertion sort

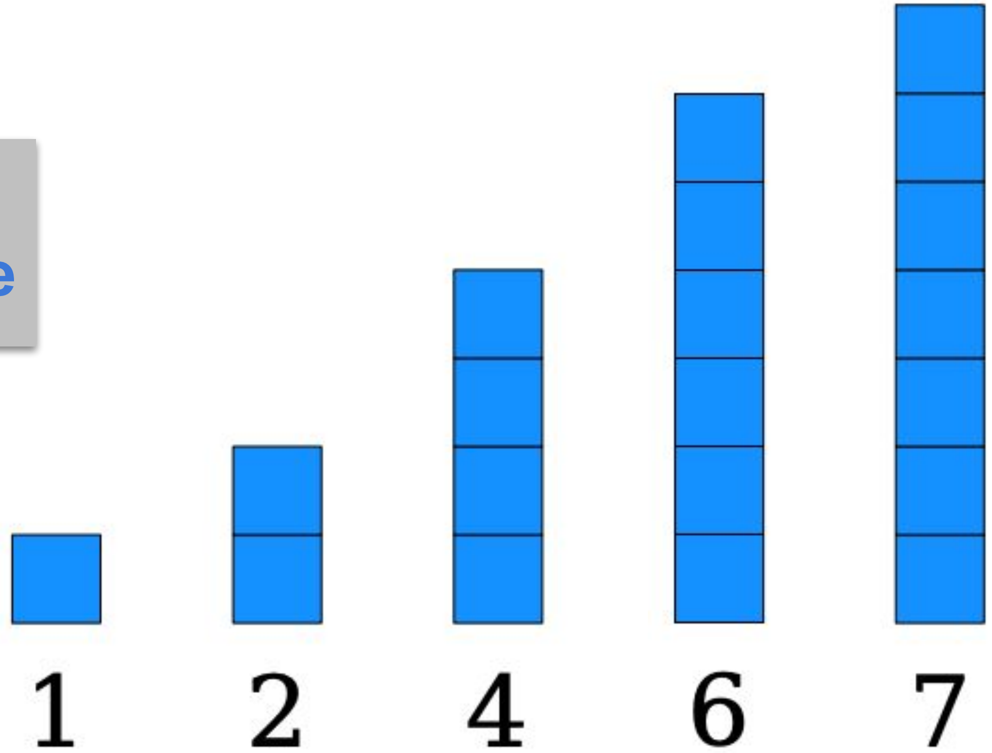


Insertion sort



Insertion sort

The blue elements are sorted!



Insertion sort algorithm

- Repeatedly insert an element into a sorted sequence at the front of the array.

- To insert an element, swap it backwards until either:
  - (1) it's bigger than the element before it, or
  - (2) it's at the front of the array.

Insertion sort code

```
void insertionSort(Vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        /* Scan backwards until either (1) the preceding
         * element is no bigger than us or (2) there is
         * no preceding element. */
        for (int j = i - 1; j >= 0; j--) {
            if (v[j] <= v[j + 1]) break;
            /* Swap this element back one step. */
            swap(v[j], v[j + 1]);
        }
    }
}
```

The complexity of insertion sort

- In the worst case (the array is in reverse sorted order), insertion sort takes time $O(n^2)$.
  - The analysis for this is similar to selection sort!

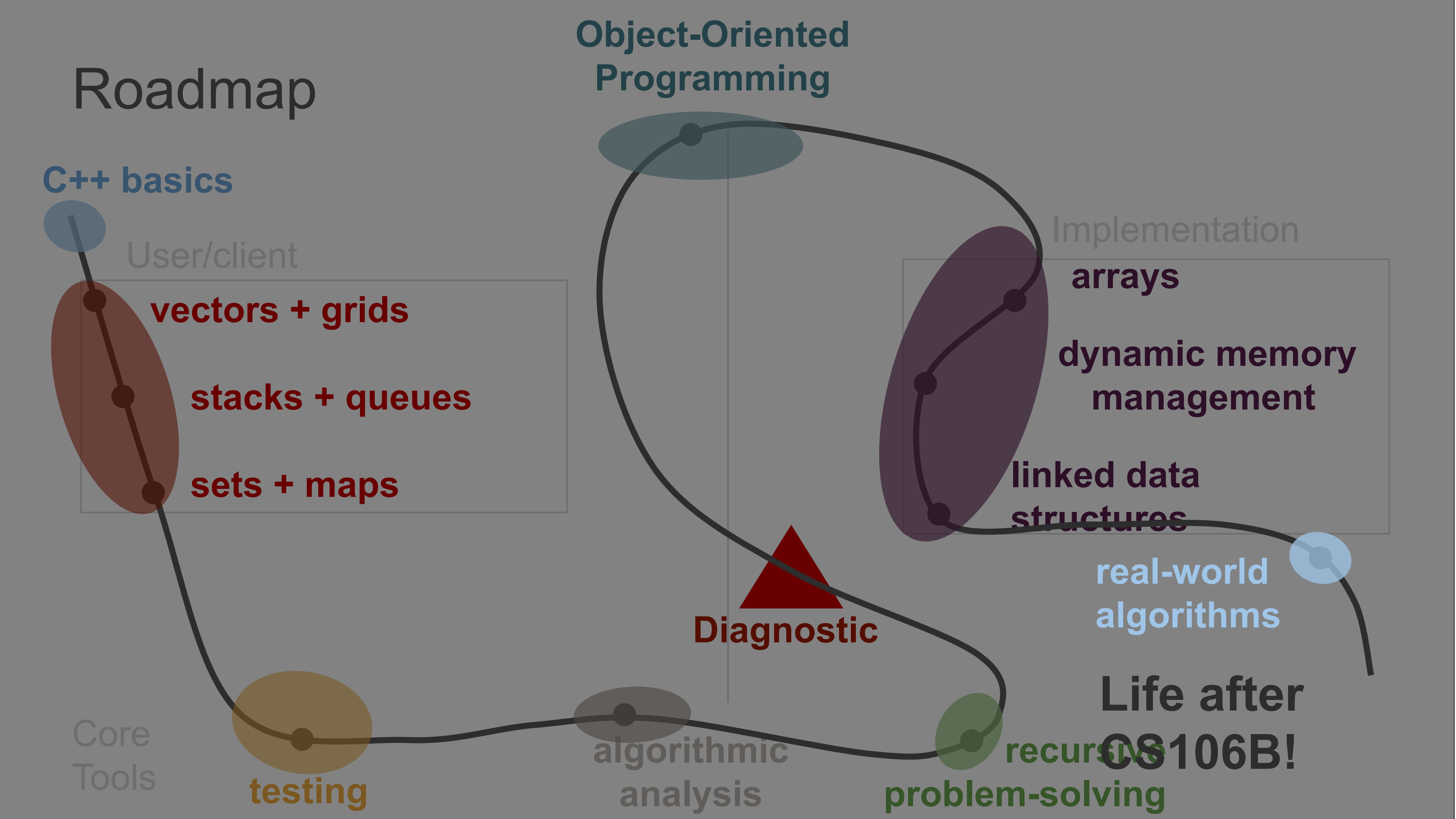
- In the best case (the array is already sorted), insertion takes time O(n) because you only iterate through once to check each element.
  - Selection sort, however, is always $O(n^2)$ because you always have to search the remainder of the list to guarantee that you're finding the minimum at each step.

- **Fun fact**: Insertion sorting an array of random values takes, *on average*, $O(n^2)$ time.
  - This is beyond the scope of the class – take CS109 if you're interested in learning more!

Tomorrow, we’ll see more efficient sorting algorithms that use “**divide-and-conquer**”!


std::stable_sort (gcc) - 8950 comparisons, 20268 array accesses, 1.00 ms delay
http://panthema.net/2013/sound-of-sorting

What’s next?


Roadmap
Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Mergesort and Quicksort