Lecture 9
Ordered and Sorted Ranges
Algorithms and D.S. to Represent Sets

EECS 281: Data Structures & Algorithms

Ordered and Sorted Containers

Data Structures & Algorithms

Container Review

- Objects storing a variable number of other objects
- Allow for control/protection of data
- Can copy/edit/sort/move many objects at once
- Examples: vector, deque, stack, map, list, array

Unordered Container

Ordered Container

Nested Container?

Types of Containers

Type	Distinctive interfaces (not all methods listed)
Container	Supports `add()` and `remove()` operations
Searchable Container	Adds `find()` operation
Sequential Container	Allows iteration over elements in some order
Ordered Container	Sequential container which maintains current order. Can arbitrarily insert new elements anywhere. **Example: Books on a shelf**
Sorted Container	Sequential container with pre-defined order. Can NOT arbitrarily insert elements. **Example: Students sorted by ID**

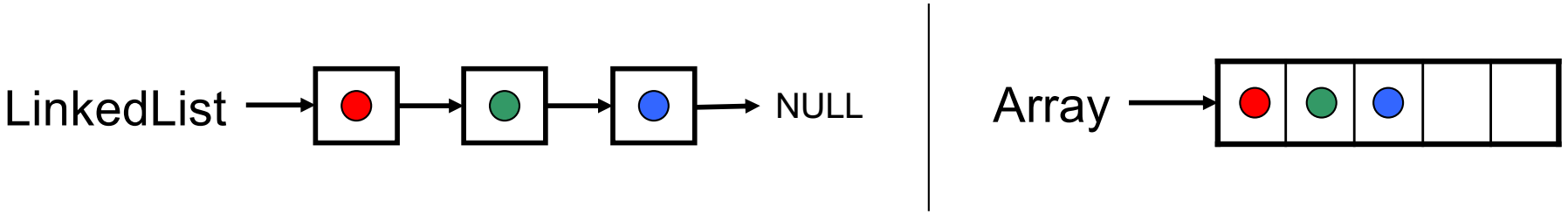
When would sorted containers be preferred over ordered?

Ordered Clarification

- Ordered container elements maintain their “relative” position unless they are removed
- Example: If A comes before Q, and then Z is inserted between them, their relative ordering has not changed
  - A still comes before Q
- If Q is then removed, the container is still ordered
  - Before and after delete, A comes before Z

Implementing Sorted and Ordered Containers

- Two implementation styles: connected, contiguous



- Preferred implementation dependent upon requirements of application
  - Know which operations will be called often
- Study multiple implementations
  - Know asymptotic complexity of each operation

When would a linked list be preferred over an array?

Asymptotic Complexities: **Ordered** Container

Operation	**Array**	**Linked List**
`addElement(val)`	$O(1)$	$O(1)$
`remove(val)`	$O(n)$	$O(n)$
`remove(iterator)`	**$O(n)$**	**$O(n)$ or $O(1)$**
`find(val)`	$O(n)$	$O(n)$
`Iterator::operator*()`	$O(1)$	$O(1)$
`operator[](unsigned)`	**$O(1)$**	**$O(n)$**
`insertAfter(iterator, val)`	**$O(n)$**	**$O(1)$**
`insertBefore(iterator, val)`	**$O(n)$**	**$O(n)$ or $O(1)$**

Asymptotic Complexities: **Sorted** Container

Operation	**Array**	**Linked List**
`addElement(val)`	$O(n)$	$O(n)$
`remove(val)`	$O(n)$	$O(n)$
`remove(iterator)`	**$O(n)$**	**$O(n)$ or $O(1)$**
`find(val)`	**$O(\log n)$**	**$O(n)$**
`Iterator::operator*()`	$O(1)$	$O(1)$
`operator[](unsigned)`	**$O(1)$**	**$O(n)$**
`insertAfter(iterator, val)`	**N/A**	**N/A**
`insertBefore(iterator, val)`	**N/A**	**N/A**

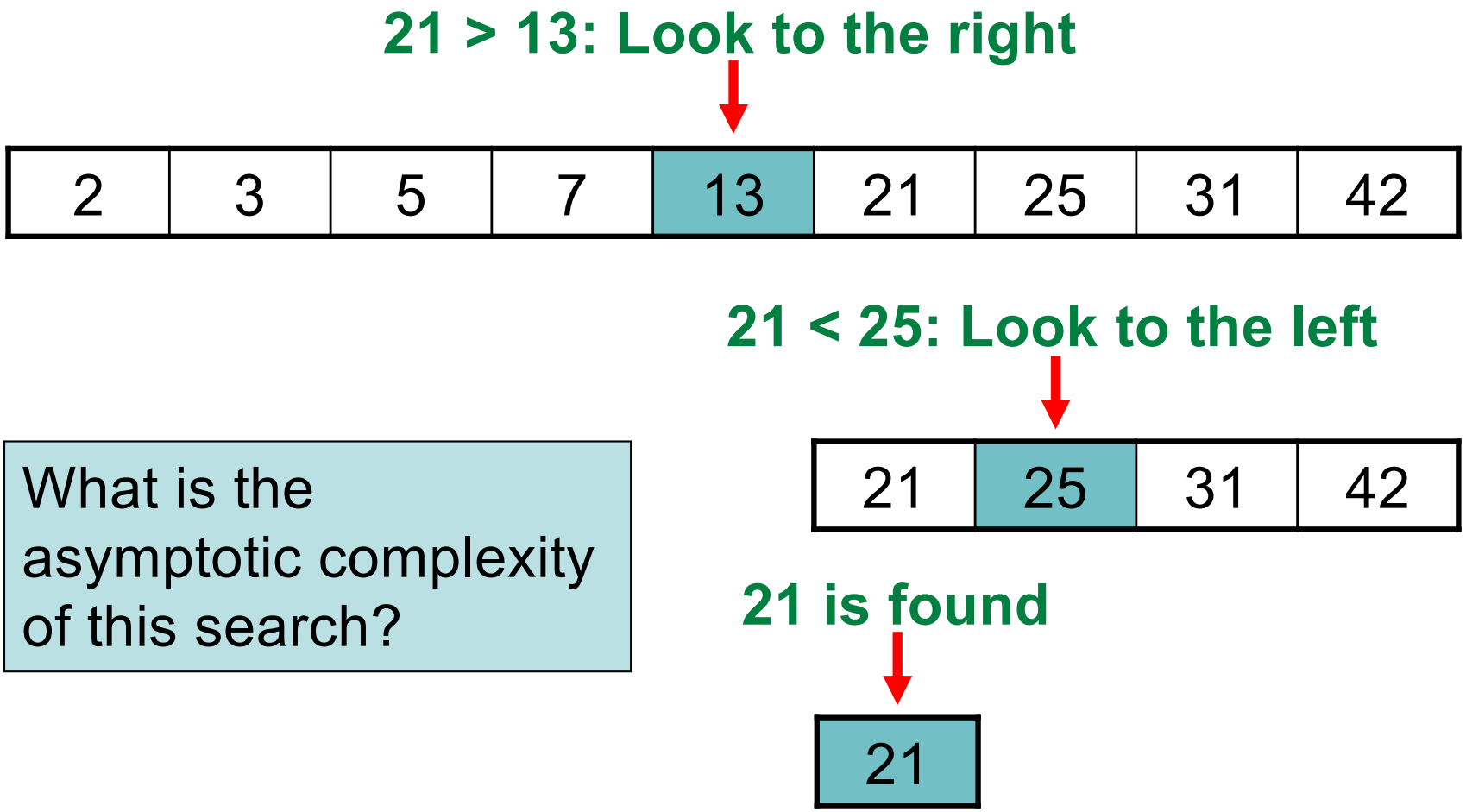
Ordered and Sorted Containers

Data Structures & Algorithms

Binary Search and Bounds

Data Structures & Algorithms

Binary Search Example: Find 21



What is the asymptotic complexity of this search?

Binary search requires elements to be sorted

Binary Search

```
1   int bsearch(double a[], double val,
2               int left, int right) {
3     while (right > left) {
4       int mid = left + (right - left) / 2;
5       if (val == a[mid])
6         return mid;
7
8       if (val < a[mid])
9         right = mid;
10      else
11        left = mid + 1;
12    }  // while
13
14    return -1;  // val not found
15  }  // bsearch()
```

n is size of a[]
n = right – left
loop at most *k* times
1 step
1 step
1 step
1 step
1 step

n is split in half each loop
n = *n* / 2
$2^k = n$
Total: 5k steps = O(k)
But what is k? **log(*n*)**

Asymptotic Complexity = *O*(log *n*)

How do we compare elements that are objects?

Comparators (Function Objects)

Given elements a and b, tell if a "<" b

```
1   struct Point {
2     int x, y;
3     Point() : x(0), y(0) { }
4     Point(int xx, int yy) : x(xx), y(yy) { };
5   };  // Point{}
```

```
6   struct CompareByX {
7     bool operator()(const Point &a, const Point &b) const {
8       return a.x > b.x;
9     }  // operator()()
10  };  // CompareByX{}
```

```
11  struct CompareByY {
12    bool operator()(const Point &a, const Point &b) const {
13      return a.y < b.y;
14    }  // operator()()
15  };  // CompareByY{}
```

Speeding up Binary Search

- **Speed-up idea: == rarely triggers,**
  - **check if < first**
  - **else if >**
  - **else must be ==**

- More radical idea: move the == check out of the loop
  - Find a sharp *lower bound* for the sought element first
  - *First item >= what I'm looking for*
  - Check for the value == after the loop

Almost Always 3 Comparisons/Loop

```
1   int bsearch(double a[], double val,
2               int left, int right) {
3     while (right > left) {                    // ONE
4       int mid = left + (right - left) / 2;
5
6       if (val == a[mid])                      // TWO
7         return mid;
8       else if (val < a[mid])                  // THREE
9         right = mid;
10      else
11        left = mid + 1;
12    }  // while
13
14    return -1;  // val not found
15  }  // bsearch()
```

2 Comparisons Half the Time

```
1  int bsearch(double a[], double val,
2              int left, int right) {
3    while (right > left) {                    // ONE
4      int mid = left + (right - left) / 2;
5
6      if (a[mid] < val)      // TWO: check < not ==
7        left = mid + 1;
8      else if (val < a[mid])                  // THREE
9        right = mid;
10     else  // (val == a[mid])
11       return mid;
12   }  // while
13
14   return -1;  // val not found
15 }  // bsearch()
```

Always 2 Comparisons/Loop

```
1   int lower_bound(double a[], double val,
2                   int left, int right) {
3     while (right > left) {                    // ONE
4       int mid = left + (right - left) / 2;
5
6       if (a[mid] < val)                       // TWO
7         left = mid + 1;
8       else // (a[mid] >= val)
9         right = mid;
10    }  // while
11
12    return left;
13  }  // lower_bound()
```

Must check if `val` is present in `a[]` – do this before returning or (as in STL) require the caller to do this.

Binary Search in STL

`binary_search()` returns a `bool`, not the location

To find locations (iterators), use

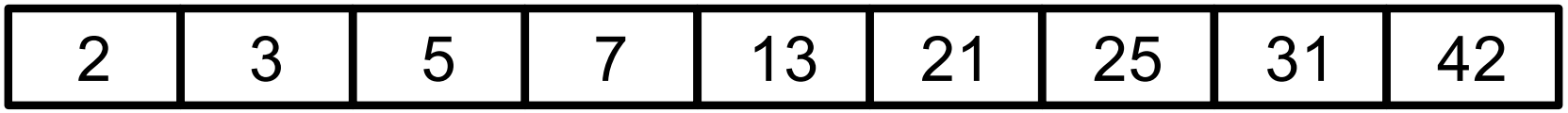
- `lower_bound()` First item not less than target
- `upper_bound()` First item greater than target
- `equal_range()` Pair(lb, ub), all items equal to target

References

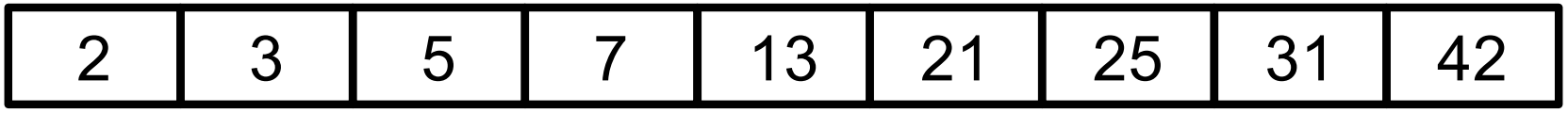
- http://en.wikipedia.org/wiki/Binary_search_algorithm
- http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binarySearch

Search Bounds

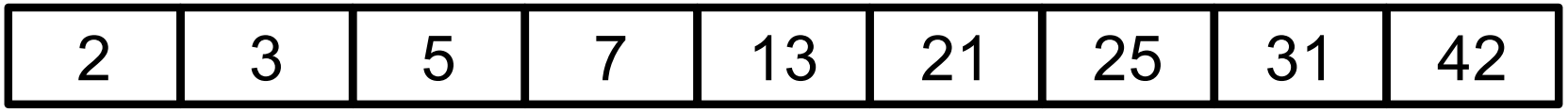
- `lower_bound(begin(v), end(v), 7)`



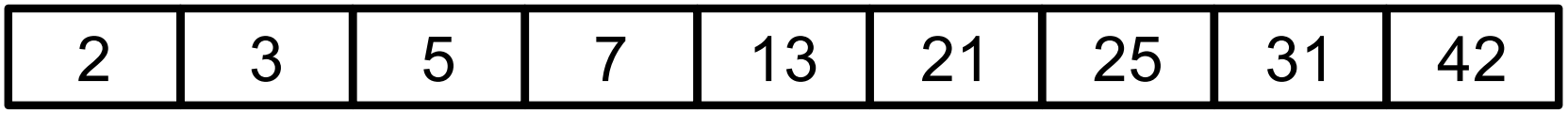
- `lower_bound(begin(v), end(v), 26)`



- `upper_bound(begin(v), end(v), 21)`



- `upper_bound(begin(v), end(v), 4)`



Binary Search and Bounds

Data Structures & Algorithms

Representing Sets

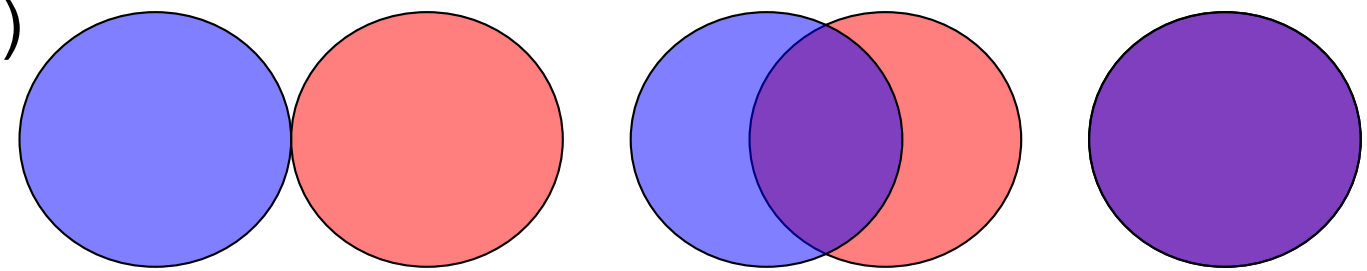
Data Structures & Algorithms

Searchable Containers as Sets

A set is well-defined if you can tell
if any given element is in the set
(Searchable containers well suited to finding elements for sets)

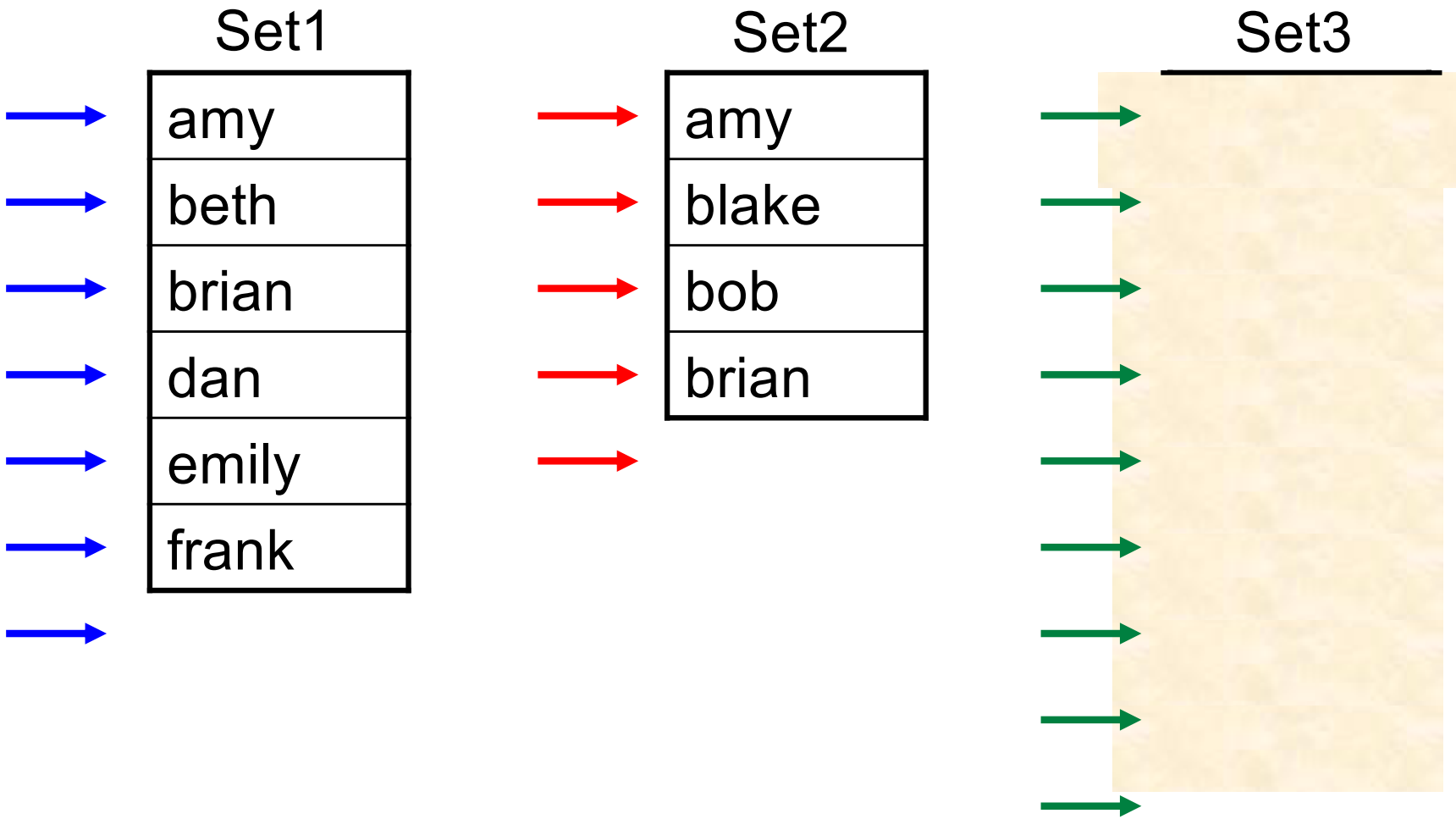
Set Operations (STL implements many of these)

- Union (∪): in one set or the other (OR)
- Intersection (∩): in both sets (AND)
- Set Difference (\): in one and not the other (AND-NOT)
- Symmetric Difference (÷): in only one (XOR)
- addElement (+)
- isElement (∈)
- isEmpty



`set_union()` Example

Set1 and Set2 are sorted ranges
Set3 is a union of Set1 and Set2



set_union() Example Code

```
1  template<class ForIterator1, class ForIterator2, class OutIterator,
2                  class Compare>
3  OutIterator set_union(ForIterator1 first1, ForIterator1 last1,
4                        ForIterator2 first2, ForIterator2 last2,
5                        OutIterator result, Compare compare) {
6    while (first1 != last1 && first2 != last2) {
7      if (compare(*first1, *first2))
8        *result++ = *first1++;  // set1 element less than set2 element
9      else if (compare(*first2, *first1))
10       *result++ = *first2++;  // set2 element less than set1 element
11     else {
12       *result++ = *first1++;  // set1 element == set2 element
13       ++first2;
14     }  // else
15   }  // while
16   while (first1 != last1)
17     *result++ = *first1++;  // Remaining elements
18   while (first2 != last2)
19     *result++ = *first2++;  // Remaining elements
20   return result;  // returns end of sorted union of set1 and set2
21 }  // set_union()
```

Implementing [sub]sets with ranges

Method	**Asymptotic Complexity**
`initialize()`	$O(1)$ or $O(n \log n)$
`clear()`	$O(1)$ or $O(n)$
`isMember()`	$O(\log n)$
`copy()`	$O(n)$
`set_union()`	$O(n)$
`set_intersection()`	$O(n)$

Universe: set of all elements that may be in a set

Job Interview Problems

- Given a sorted array with n elements and a number z
- Do the following in $O(n)$ time
  - Find pairs (x, y) such that $x - y = z$
  - Find pairs (x, y) such that $|x - y|$ is closest to z
  - Count all pairs (x, y) such that $x + y < z$
- What if the array was not sorted ?

Representing Sets

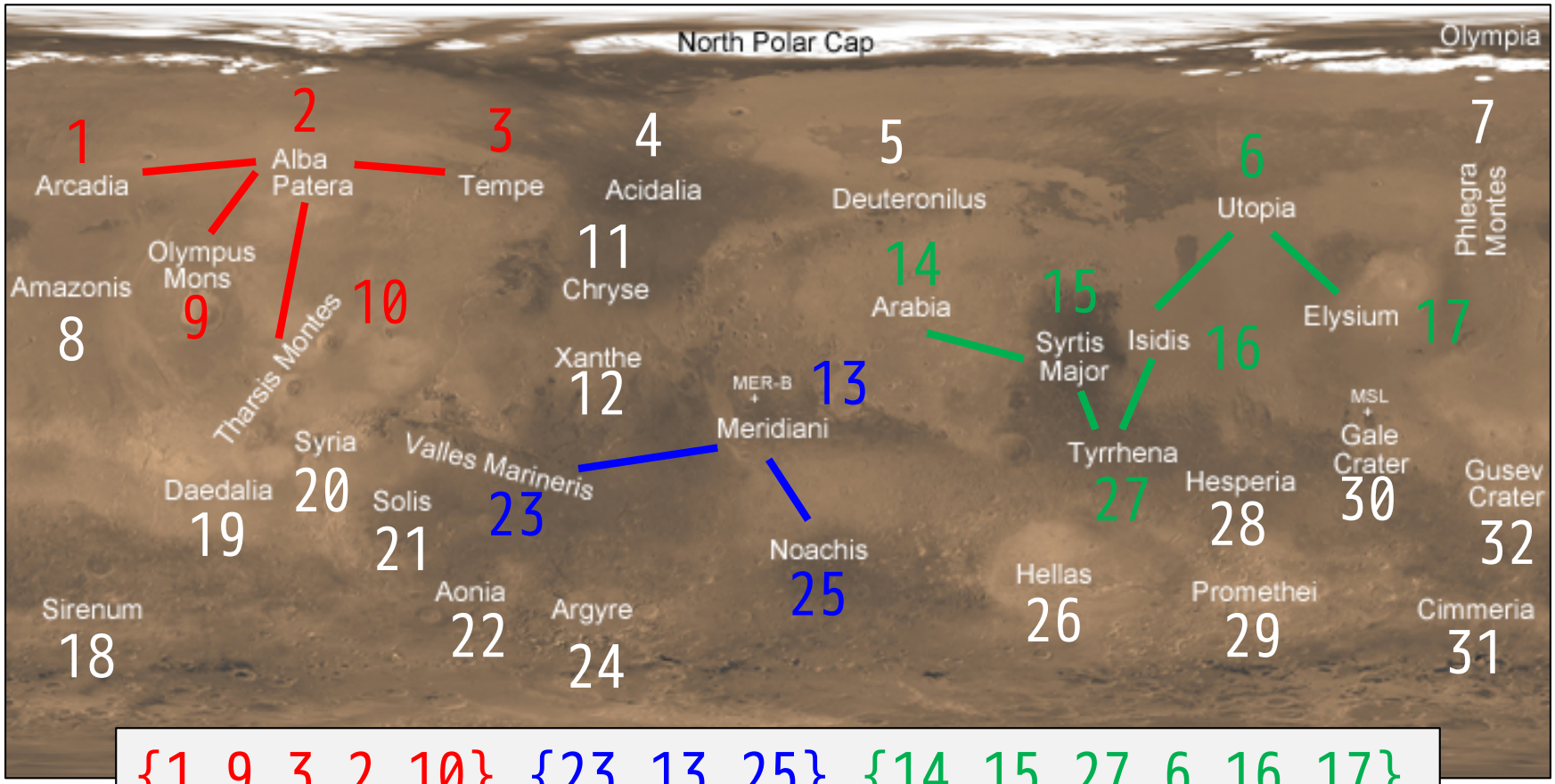
Data Structures & Algorithms

Union-Find

Data Structures & Algorithms

Representing Disjoint Sets

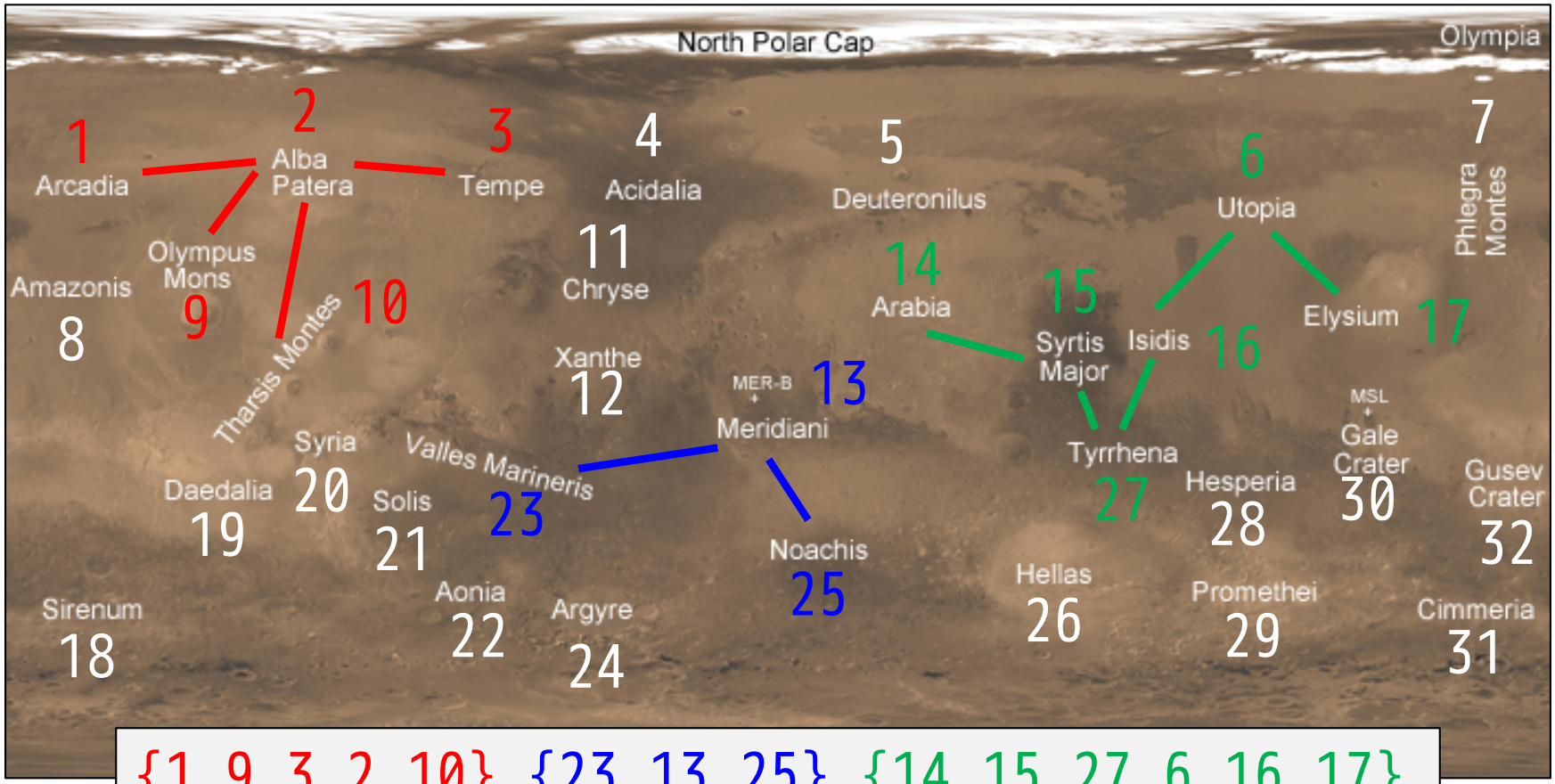
- Sets are **disjoint** if they *do* not share any elements.
  (i.e. an element may only belong to one set)
- Many applications require representing and operating on disjoint sets.
  For example, graph connectivity:



{1,9,3,2,10} {23,13,25} {14,15,27,6,16,17}
Also, single element sets: {8} {4} {5} {11} …

Representing Disjoint Sets

- In this context, only two set operations make sense:
  - **Find**: check if an element belongs to a particular set
    (or if two belong to the same set)
  - **Union**: merge two sets together into a single set



{1,9,3,2,10} {23,13,25} {14,15,27,6,16,17}
Also, single element sets: {8} {4} {5} {11} …

Representing Disjoint Sets

- In this context, only two set operations make sense:
  - **Find**: check if an element belongs to a particular set (or if two belong to the same set)
  - **Union**: merge two sets together into a single set

- Consequently, data structures for representing disjoint sets are often referred to as "**union-find**".
  - How can we implement this so that the two operations are as efficient as possible?
  - We also consider the amortized complexity over the entire lifetime of operations. (i.e. start with *n* single element sets, end up with one big set)

Union-Find: Example 1
Separate Containers

Items: 1 2 3 4 5 6 7 8 9 10

Group 1: 1 4 5 8 10	Group 2: 2 6 7

Are 1 and 8 connected? (i.e. in the same set?)

If we add a connection between 1 and 6,
how long does it take to merge the two sets?

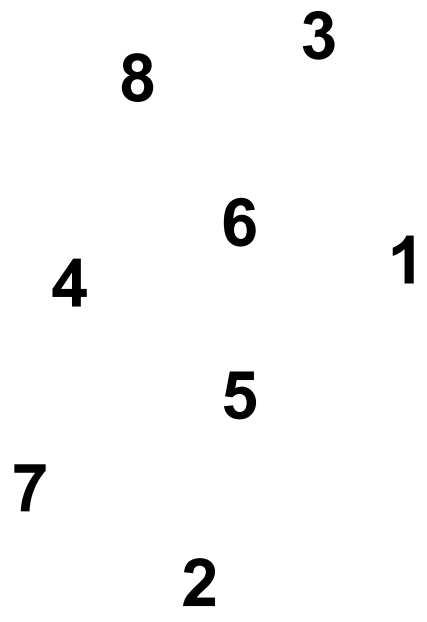
Union(): $O(n)$
Find(): $O(n)$

Union-Find Data Structure

- **Idea 1**: every *disjoint* set should have its unique representative (selected element)
  - Every set element *k* must know its representative *j*
- **Therefore**: to tell if *k* and *m* are in the same set, *compare their representatives*
  - `Find()` operation becomes quite fast!

Union-Find: Example 2 Representatives

Item	1	2	3	4	5	6	7	8
Representative								



`Union()`: $O(n)$
`Find()`: $O(1)$

1) Join 1 & 3
2) Join 3 & 8
3) Join 1 & 5
4) Join 7 & 4
5) Join 7 & 2
6) Join 2 & 5

Making Union-Find Faster

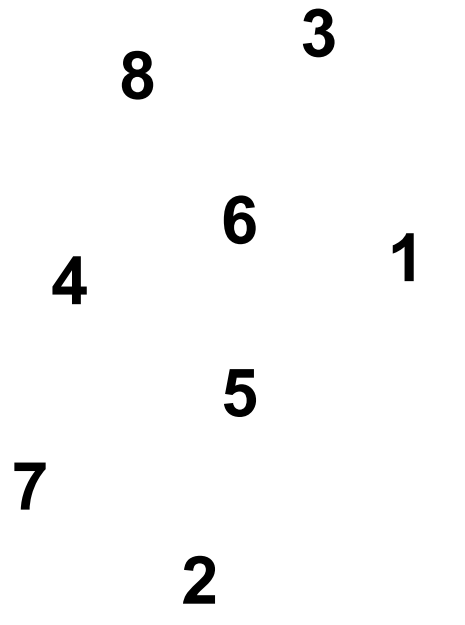
- **Idea 2**: When performing union of two sets, update the smaller set (less work)
- Measure complexity of all unions throughout the lifecycle (together)
  - We call `Union()` exactly *n - 1* times
  - If we connect to a disjoint element every time, it will take *n* time total (best case)
  - Merging large sets, say *n/2* and *n/2* elements, will take *O(n)* time for one `Union()` – too slow!

Smarter Union-Find

- **Idea 3**: No need to store the actual representative for each element, as long as you can find it quickly
  - Each element knows someone who knows the representative (may need more steps)
  - `Union()` becomes very fast: one of representatives will need to know the other
  - `Find()` becomes slower

Union-Find: Example 3
Hierarchical Representatives

Item	1	2	3	4	5	6	7	8
Representative								



`Union()`: $O(1)$

`Find()`: $O(n)$

(worst case – "stick" tree)

1) Join 1 & 6
2) Join 1 & 8
3) Join 1 & 3
4) Join 5 & 6
5) Join 4 & 7
6) Join 4 & 2
7) Join 5 & 4

Path Compression

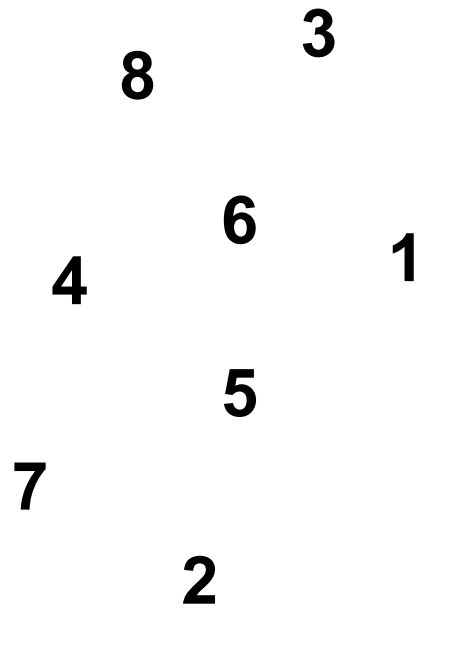
- So far, `Find()` was read-only
  - For element *j*, finds the representative *k*
  - Traverses other elements on the way (for which *k* is also the representative)
- **Idea 4:** We can tell *j* that its representative is now *k*
  - Same for other elements on path from $j \rightarrow k$
  - Doubles workload of `Find()`, but same Big-O

Complexity with Path Compression

- Must use amortized analysis over the life cycle of union-find (starting with *n* disjoint sets, and merging until there is one set containing all elements)
- Result is surprising
  - $O(n^*\alpha(n))$, where $\alpha()$ grows very slowly
  - $\alpha()$ is the inverse-Ackerman function
  - In practice, almost-linear-time performance
- Details taught in more advanced courses

Union-Find: Example 4
Path Compression

Item	1	2	3	4	5	6	7	8
Representative								



1) Join 2 & 4
2) Join 1 & 2
3) Join 5 & 7
4) Join 5 & 6
5) Join 8 & 5
6) Join 4 & 7

`Union()`: $O(\alpha(n))$
`Find()`: $O(\alpha(n))$

(Amortized over the lifetime of Union-Find)

Union-Find

Data Structures & Algorithms

Study Questions

- What is the difference between a sorted and an ordered container?
- When should you implement a sorted container with an array instead of a linked list?
- When should you implement an ordered container with an array instead of a linked list?
- What is binary search? Study STL's interface to it.
- What are comparison operators and comparator objects?
- How are searchable containers and sets related?
- What is a universe set?
- Give an example of a universe set and a subset of it.
- Implement `set_intersection()`.
- How would you implement a Union-Find data structure?