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

EECS 281: 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?

container
find
searchable

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	Ordered container with pre-defined order. Can NOT arbitrarily insert elements. **Example: Students sorted by ID**

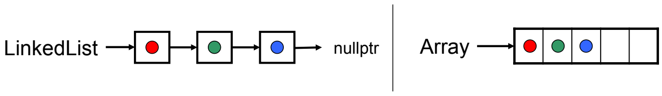
When would sorted containers be preferred over ordered?

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?

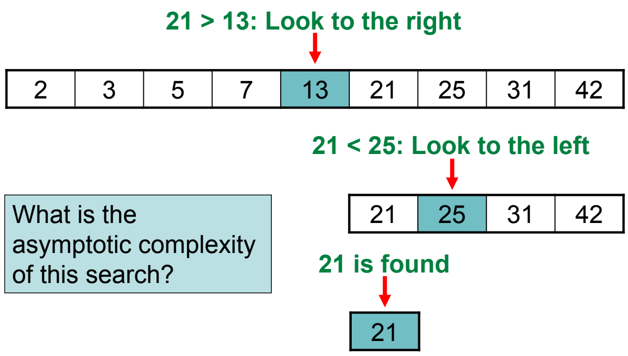
Asymptotic Complexities: **Ordered** Container

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

Binary Search Example: Find 21



What is the asymptotic complexity of this search?

Binary search requires elements to be sorted

Binary Search

```
int bsearch(double a[], double val,
            int left, int right) {
  while (right > left) {
    int mid = left + (right - left) / 2;
    if (val == a[mid])
      return mid;

    if (val < a[mid])
      right = mid;
    else
      left = mid + 1;
  } // while

  return -1;  // val not found
} // 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

```
struct Point {
  int x, y;
  Point() : x(0), y(0) { }
  Point(int x_, int y_) : x(x_), y(y_) { };
}; // Point{}

struct CompareByX {
  bool operator()(const Point &a, const Point &b) const {
    return a.x > b.x;
  } // operator()()
}; // CompareByX{}

struct CompareByY {
  bool operator()(const Point &a, const Point &b) const {
    return a.y < b.y;
  } // operator()()
}; // 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

```
int bsearch(double a[], double val,
            int left, int right) {
  while (right > left) {                    // ONE
    int mid = left + (right - left) / 2;

    if (val == a[mid])                      // TWO
      return mid;
    else if (val < a[mid])                  // THREE
      right = mid;
    else
      left = mid + 1;
  } // while

  return -1;  // val not found
} // bsearch()
```


2 Comparisons Half the Time

```
int bsearch(double a[], double val,
            int left, int right) {
  while (right > left) {                    // ONE
    int mid = left + (right - left) / 2;

    if (a[mid] < val)      // TWO: check < not ==
      left = mid + 1;
    else if (val < a[mid])                  // THREE
      right = mid;
    else   // (val == a[mid])
      return mid;
  } // while

  return -1;  // val not found
} // bsearch()
```


Always 2 Comparisons/Loop

```
int lower_bound(double a[], double val,
                int left, int right) {
  while (right > left) {                    // ONE
    int mid = left + (right - left) / 2;

    if (a[mid] < val)                       // TWO
      left = mid + 1;
    else // (a[mid] >= val)
      right = mid;
  } // while

  return left;
} // 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 use these functions that return iterators

- 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
- https://www.topcoder.com/thrive/articles/Binary%20Search

Search Bounds

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

2	3	5	7	13	21	25	31	42

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

2	3	5	7	13	21	25	31	42

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

2	3	5	7	13	21	25	31	42

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

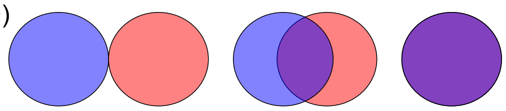
2	3	5	7	13	21	25	31	42

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)

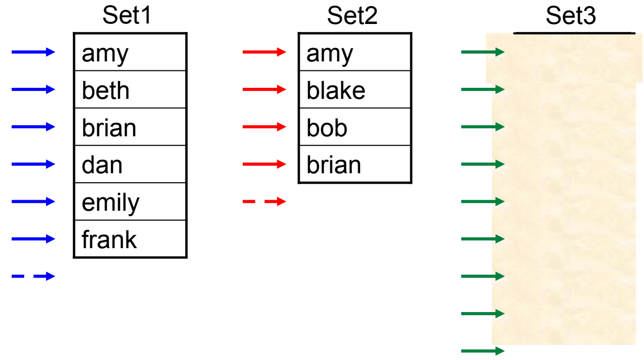
Set Operations (STL implements many of these)

- 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

```
template<class ForIterator1, class ForIterator2, class OutIterator,
              class Compare>
OutIterator set_union(ForIterator1 first1, ForIterator1 last1,
                      ForIterator2 first2, ForIterator2 last2,
                      OutIterator result, Compare compare) {
  while (first1 != last1 && first2 != last2) {
    if (compare(*first1, *first2))
      *result++ = *first1++;  // set1 element less than set2 element
    else if (compare(*first2, *first1))
      *result++ = *first2++;  // set2 element less than set1 element
    else {
      *result++ = *first1++;  // set1 element == set2 element
      ++first2;
    } // else
  } // while
  while (first1 != last1)
    *result++ = *first1++;  // Remaining elements
  while (first2 != last2)
    *result++ = *first2++;  // Remaining elements
  return result;  // returns end of sorted union of set1 and set2
} // set_union()
```


Calling this Function, 2 Ways

```
// Assume we've got vector: v1, linked list: l1
vector<string> out;
// Resize the output vector to hold the union
out.resize(v1.size() + l1.size());
auto it = set_union(v1.begin(), v1.end(),
                    l1.begin(), l1.end(),
                    out.begin(), less<string>());
// Erase from the end of the union to end of out
out.erase(it, out.end());

/////////////////////////////////////////////////
// Do it without resizing!
set_union(v1.begin(), v1.end(),
          l1.begin(), l1.end(),
          back_inserter(out), less<string>());
```


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*)**

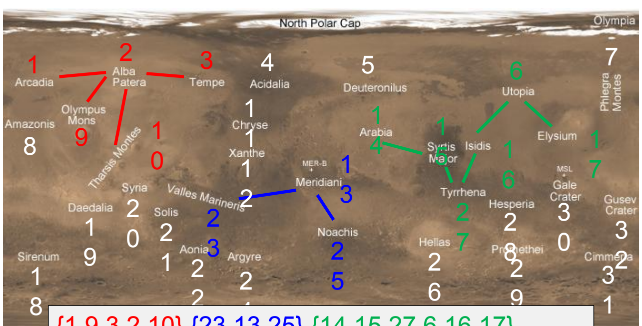
Universe: set of all elements that may be in a set

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 Disjoint Sets

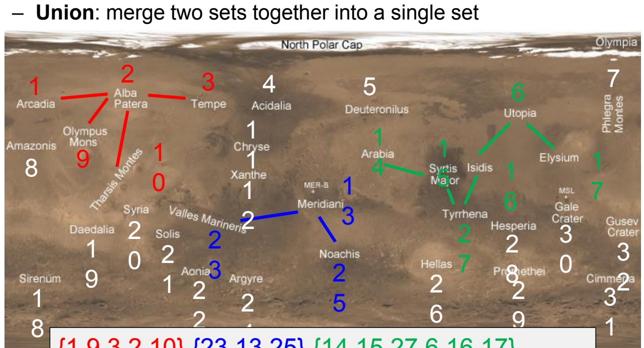
- Sets are **disjoint** if they *do* not share any elements.
  (i.e. an element may only belong to one set)
- Many applications require representing and operating on disjoint sets.
  For example, graph connectivity:



{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} …

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?

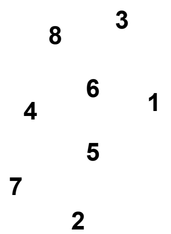
Union(): ***O*(*n*)**
Find(): ***O*(*n*)**

Union-Find Data Structure

- **Idea 1**: every *disjoint* set should have its <u>unique representative</u> (selected element)
  - Every set element *k* must <u>know</u> 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	1	2	3	4	5	6	7	8

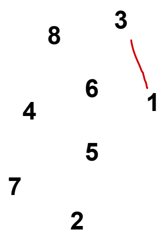


1) Union 1 & 3
2) Union 3 & 8
3) Union 1 & 5
4) Union 7 & 4
5) Union 7 & 2
6) Union 2 & 5

Union(): ***O*(*n*)**
Find(): ***O*(1)**

Union-Find: Example 2
Representatives

Item	1	2	3	4	5	6	7	8
Representative	1	2	1	4	5	6	7	8

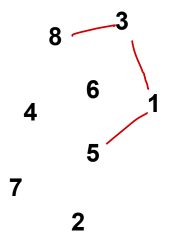


1) Union 1 & 3
2) Union 3 & 8
3) Union 1 & 5
4) Union 7 & 4
5) Union 7 & 2
6) Union 2 & 5

Union(): ***O*(*n*)**
Find(): ***O*(1)**

Union-Find: Example 2
Representatives

Item	1	2	3	4	5	6	7	8
Representative	1	2	1	4	1	6	7	1

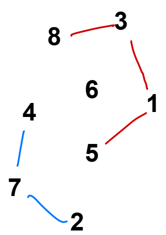


1) Union 1 & 3
2) Union 3 & 8
3) Union 1 & 5
4) Union 7 & 4
5) Union 7 & 2
6) Union 2 & 5

Union(): ***O*(*n*)**
Find(): ***O*(1)**

Union-Find: Example 2
Representatives

Item	1	2	3	4	5	6	7	8
Representative	1	7	1	7	1	6	7	1

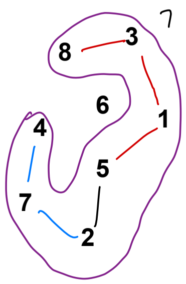


1) Union 1 & 3
2) Union 3 & 8
3) Union 1 & 5
4) Union 7 & 4
5) Union 7 & 2
6) Union 2 & 5

Union(): ***O*(*n*)**
Find(): ***O*(1)**

Union-Find: Example 2
Representatives

Item	1	2	3	4	5	6	7	8
Representative	~~1~~ 7	7	~~1~~ 7	7	~~1~~ 7	6	7	~~1~~ 7



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

Union(): ***O*(*n*)**
Find(): ***O*(1)**

Making Union-Find Faster

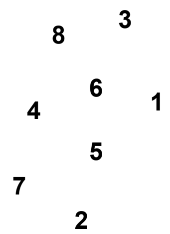
- **Idea 2**: When performing union of two sets, <u>update the smaller set</u> (less work)
- Measure complexity of all unions throughout the lifecycle (together)
  - We call Union() <u>exactly</u> *n - 1* times
  - If we connect to a disjoint element every time, it will take *n* time total (best case)
  - 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								



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

Union(): ***O*(1)**
Find(): ***O*(*n*)**
(worst case – "stick" tree)

Path Compression

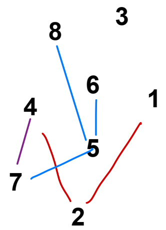
- So far, Find() was read-only
  - For element *j*, finds the representative *k*
  - Traverses other elements on the way (for which *k* is also the representative)
- **Idea 4:** We can tell *j* that its representative is now *k*
  - Same for other elements on path from *j*→*k*
  - Doubles workload of 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**α(*n*))**, where α() grows very slowly
  - α() 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	~~2~~ 1	3	~~4~~ 2	~~5~~ 8	~~6~~ 5	~~7~~ 5	8



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

Union(): ***O*(α(*n*))**
Find(): ***O*(α(*n*))**
(Amortized over the lifetime of Union-Find)

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?