Lecture 8
Ordered and Sorted Ranges
Algorithms and D.S. to Represent Sets

EECS 281: Data Structures & Algorithms

Container Review

- Objects storing a variable number of data items
- Allow for control/protection of data
- Can copy/edit/sort/order many objects at once
- Used in creating more complex data structures
  - Containers within containers
  - Useful in searching through data
  - Databases are simply fancy containers
- Examples: array, list, stack, queue, map
- STL (Standard Template Library)

Types of Containers

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

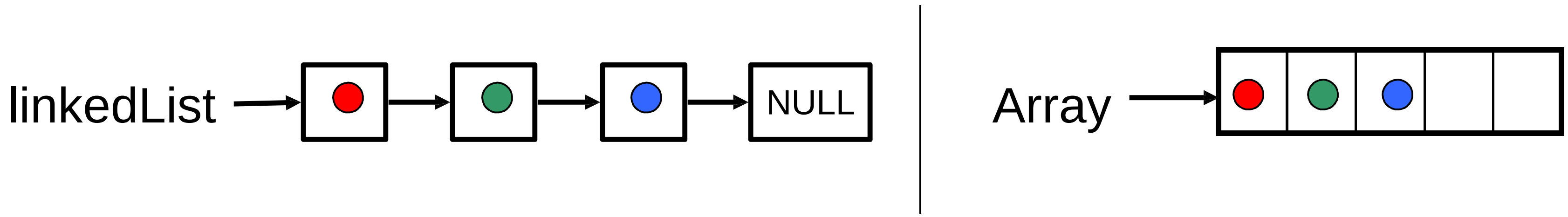
When would sorted containers be more useful than ordered?

Interfaces for Sorted and Ordered Containers

Interface	Sorted	Ordered
addElement(val)	Override searchable container version	Inherited from searchable container
remove(val)	Inherited from searchable container	Inherited from searchable container
isMember(val)	Inherited from searchable container	Inherited from searchable container
find(val)	Inherited from searchable container	Inherited from searchable container
findPosition(val)	Yes	Yes
operator[]()	Yes	Yes
withdraw(iterator)	Yes	Yes
insertAfter()	No	Yes
insertBefore()	No	Yes

Implementing Sorted and Ordered Containers

- More than one implementation



- 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 $O(1)$
isMember(val)	$O(n)$	$O(n)$
find(val)	$O(n)$	$O(n)$
findPosition(val)	$O(n)$	$O(n)$
iterator operator*()	$O(1)$	$O(1)$
operator[](unsigned)	$O(1)$	$O(n)$
insertAfter()	$O(n)$	$O(1)$
insertBefore()	$O(n)$	$O(n)$ 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 $O(1)$
isMember(val)	$O(\log n)$	$O(n)$
find(val)	$O(\log n)$	$O(n)$
findPosition(val)	$\mathrm{O}(\log n)$	$O(n)$
iterator operator*()	$O(1)$	$O(1)$
operator[](unsigned)	$O(1)$	$O(n)$

Binary Search Example
(Searching for 21)

21 > 13: Look to the right

2	3	5	7	13	21	25	31	42

21 < 25: Look to the left

21	25	31	42

21 is found

21

What is the asymptotic complexity of this search?

Note that elements must be sorted

Binary Search

```
int search(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 - 1;
        else
            left = mid + 1;
    }
    return -1;   // val  not  found
}
```

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? k = log(n)

Asymptotic Complexity = $O(\log n)$

How do we compare elements that are objects?

Speeding up Binary Search

- **The slowest instructions: conditionals**
- Speed-up idea: == rarely triggers, so check for < first
- More radical idea: move the == check out of the loop
  - Find a sharp *lower bound* for the sought element first
  - Check for the value == after the loop
- Experiment
  - Linear search
  - Linear inverse
  - Linear vectorized
  - Binary

Binary Search in STL

binary_search() returns a bool, not the location

To find locations (iterators), use

- lower_bound()
- upper_bound()
- equal_range()

References

- http://www.sgi.com/tech/stl/table_of_contents.html
- http://en.wikipedia.org/wiki/Binary_search_algorithm
- http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binarySearch

Comparators (Function Objects)

Given elements a and b, tell if a > b

```
struct Point {
   int x, y;
   Point() : x(-1), y(-1) { }
   Point(int xx, int yy) : x(xx), y(yy) { };
};
```

```
struct CompareByX {
   bool operator()(const Point &p1, const Point &p2) const {
      return p1.x > p2.x;
   }
};
```

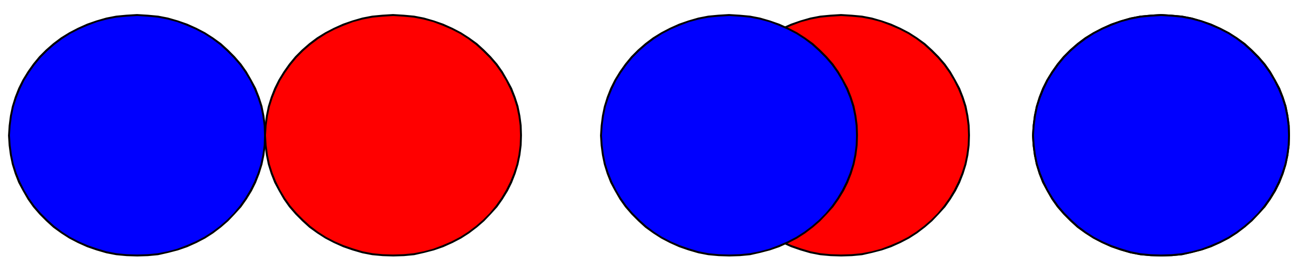
```
struct CompareByY {
   bool operator()(const Point &p1, const Point &p2) const {
      return p1.y > p2.y;
   }
};
```

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)

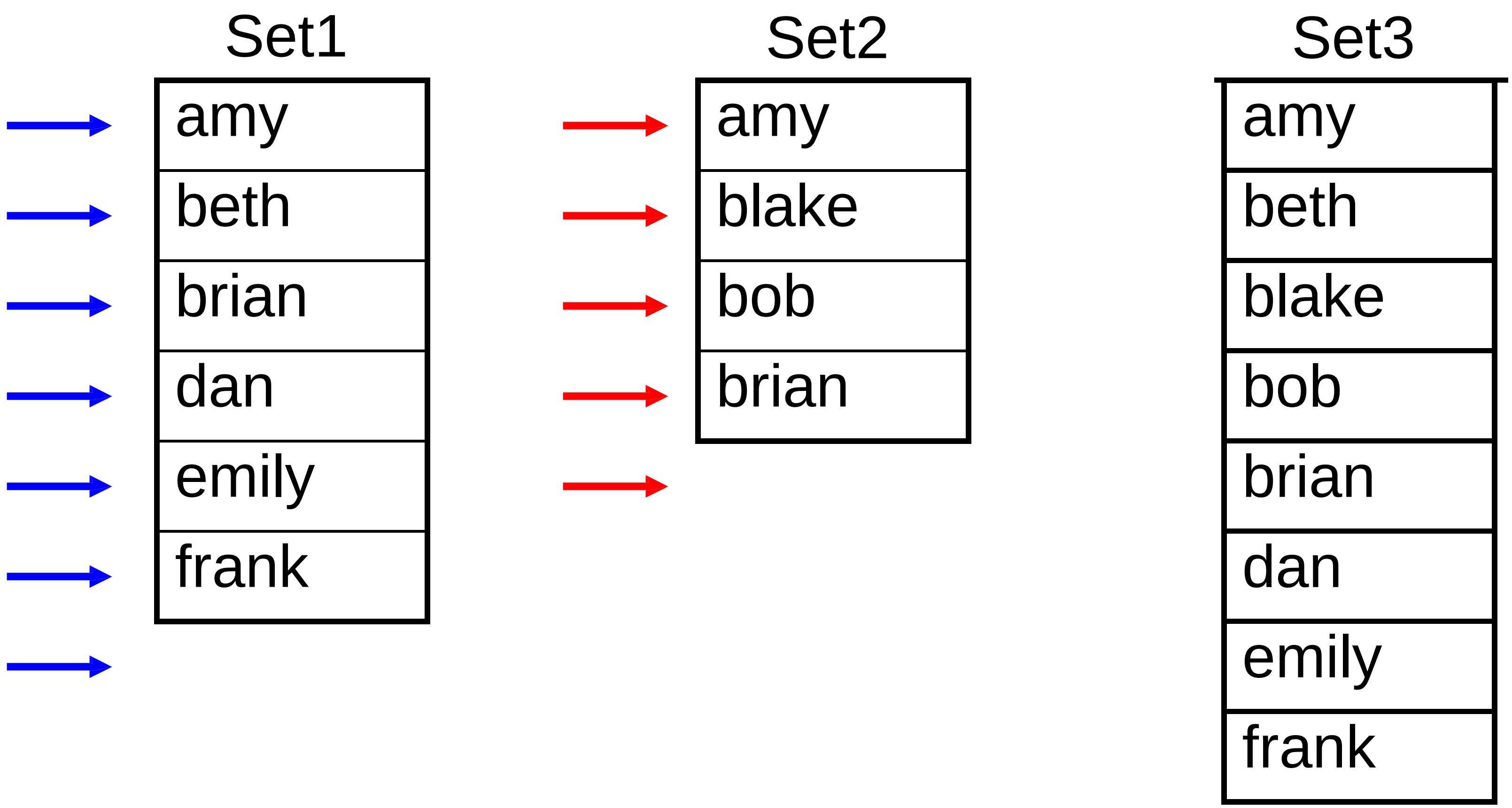
Set Operations (STL implements many of these)

- union ($\cup$) : in one set or the other (OR)
- intersection ($\cap$) : in both sets (AND)
- set-difference (***/***) : in one and not the other (AND-NOT)
- symmetric difference ($\div$) : in only one (XOR)
- isElement ($\in$)
- isEmpty
- addElement



set_union() Example

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



set_union() Example Code

```
template<class InIterator1, class InIterator2, class OutIterator, class Pred>
OutIterator set_union(InIterator1 first1, InIterator1 last1,
                      InIterator2 first2, InIterator2 last2,
                      OutIterator result, Pred pred) {
  while (first1 != last1 && first2 != last2) {
    if (pred(*first1, *first2))
      *result++ = *first1++;    // set1 element less than set2 element
    else if (pred(*first2, *first1))
      *result++ = *first2++;    // set2 element less than set1 element
    else {
      *result++ = *first1++;    // set1 element == set2 element
      first2++;
    }
  }
  while (first1 != last1) *result++ = *first1++;   // Remaining elements
  while (first2 != last2) *result++ = *first2++;
  return result;   // returns sorted union of set1 and set2
}
```

How would you implement set_intersection()?

Implementing [sub]sets with ranges

Method	Asymptotic Complexity
initialize()	$O(n)$
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 the subset

Example:

Universe:	Integers 1 to 366
Subset1:	Birthdays of people in class
Subset2:	Dates of U of M football games
Subset3:	Dates of exams

Bitvectors and Bitsets

- vector<bool> (template specialization)
- Uses 1 bit per element, not 1 byte per element
- Can be used to implement sets of integers
  - *Member testing takes constant time*
- Can use int with bitwise operators to represent subsets
  - Efficient for small universes

```
vector<bool> V(5);
V[0] = true;
V[1] = true;
V[2] = false;
V[3] = true;
V[4] = true;
```

What number can represent V?

V = 11011
1 + 2 + 0 + 8 + 16 = 27

- Or use uint64_t to get 64 bits (long long int before C++11)
- Use vector<bool> for arbitrary size that can be changed
- Use bitset<99> if size is known at compile time

Bitwise Operators

```
unsigned char c  = 127;   // 0111 1111
unsigned char c1  = 63;   // 0011 1111
unsigned char c2  = 67;   // 0100 0011

//---- NOT
unsigned char flipped = ~c;   // 1000 0000 (128)

//---- AND
unsigned char and = c1 & c2; // 0000 0011 (3)

//---- OR
unsigned char or  = c1 |c2;   // 0111 1111 (127)

//---- XOR
unsigned char xor  = c1 ^ c2; // 0111 1100 (124)
```

Op	Desc
~	NOT
&	AND
|	OR
^	XOR

Named Bitfields in C/C++

```
struct packedData {
    char c;       // 8 bits
    bool enqueued :1;
    bool visited  :1;
    unsigned other :20;
};

cout << sizeof(packedData) << " bytes" << endl;
// prints "4 bytes", i.e., 32 bits
```

Job Interview Question: Birthday Parties

Our company organizes a monthly birthday party whenever an employee has a birthday in a given month. Given the birthdays of all employees, find how many parties will occur per year.

Constraints

- $O(1)$ additional space
- Read every birthday at most once

Job Interview Problems (solve at home)

- 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 ?

Maintaining Disjoint Sets

- Consider how study groups are formed
  - Assuming no student is in two study groups
- Initially, every student is a one-person study group
- Two operations
  - Check if two students are in the same group
  - If not, merge the two groups
- Groups cannot be split or disbanded
- How can this be done efficiently?

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*
- **Idea 2**: to tell if *k* and *m* are in the same set, *compare their representatives*
  - Redundancy check becomes fast
- Two main operations: union() and find()
- Lifecycle of a union-find data structure
  - Starts with *N* entirely disjoint elements
  - Ends up with all of them in one set

Union-Find Example

Everything is stored in an array

- A[j] is the representative of j

1 2 3 4 5 6 7 8 9 10

1. Connect 2 and 6

1 <u>2</u> 3 4 5 <u>2</u> 7 8 9 10

2. Connect 8 and 6

1 <u>2</u> 3 4 5 <u>2</u> 7 <u>2</u> 9 10

3. Connect 9 and 4

1 <u>2</u> 3 <u>4</u> 5 <u>2</u> 7 <u>2</u> <u>4</u> 10

Making Union-Find Faster

- **Idea 3**: 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)
  - But merging large sets, say *N/2* and *N/2* elements, will take *O(N)* time for one union() – too slow!

Smarter Union-Find

- **Idea 4**: No need to store actual representative for each element, as long as 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*() cannot be faster than *Find*()

Another Optimization: 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 5:**

We can tell j that it's representative is k

  - Same for other elements on path from $j \rightarrow k$
  - Doubles runtime of *Find*(), but same O()

Asymptotic Complexity?

- Must use amortized analysis over the life cycle of union-find
- Result is surprising
  - $O(N\alpha(N))$, where $\alpha()$ grows <u>very slowly</u>
  - $\alpha()$ is the reverse-Ackerman function
  - In practice, almost-linear-time performance
- Details taught in more advanced courses

Study Questions

1. What is the difference between
   a sorted and an ordered container?
2. When should you implement a sorted container
   with an array instead of a linked list?
3. When should you implement an ordered container
   with an array instead of a linked list?
4. What is binary search? Study STL's interface to it.
5. What are comparison operators and comparator objects?
6. How are searchable containers and sets related?
7. What is a universe set?
8. Give an example of a universe set and a subset of it.
9. Implement set_intersection().
10. What is the difference between STL bitvectors & bitsets?
11. How would you implement a Union-Find data structure?