Container Classes

The Bag Class - Part 2

The Bag Class—Implementation

The `erase` member function:

```
// removes all copies of @target from the bag
bag::size_type bag::erase(const value_type& target) {
    size_type i = 0, num_erased = 0;
    while (i < used) {
        if (data[i] == target) {
            data[i] = data[--used];
            num_erased++;
        } else {
            i++;
        }
    }
    return num_erased;
}
```

The Bag Class—Implementation

The `operator +=` member function (with error):

```
// inserts a copy of each item in @b into the bag
void operator +=(const bag& b) {
    assert(size() + b.size() <= CAPACITY);

    for (size_type i = 0; i < b.used; i++) {
        data[used] = b.data[i];
        used++;
    }
}
```

What is potentially wrong with this function?

The Bag Class—Implementation

What happens if we say this?

```
bag b1;
b1.insert(1);
b1 += b1; // add b1 to itself
```

The problem lies in this code:

```
for (size_type i = 0; i < b.used; i++) {
    data[used] = b.data[i];
    used++;
}
```

The Bag Class—Implementation

What happens if we say this?

```
bag b1;
b1.insert(1);
b1 += b1; // add b1 to itself
```

If we add an object to itself, then `used` and `b.used` are the same!

```
for (size_type i = 0; i < b.used; i++) {
    data[used] = b.data[i];
    used++;
}
```

The loop will never end because `used` is always increasing!

The Bag Class—Implementation

We can save addend.used in a local variable (s, in the code below):

```
// inserts a copy of each item in @b into the bag
void operator +=(const bag& b) {
    assert(size() + b.size() <= CAPACITY);

    for (size_type i = 0, s = b.used; i < s; i++) {
        data[used] = b.data[i];
        used++;
    }
}
```

The Bag Class—Implementation

Or we could use the copy function from the standard library:

```
// copies items from beginning..end to destination
copy(beginning_location, end_location, destination);
```

The copy function:

- copies items starting at `beginning_location` up to <u>but not including</u> `end_location` to the given `destination`
- locations are specified with their memory addresses

Usage example:

```
// copies all items in b.data to the end of data
copy(b.data, b.data + b.used, data + used);
```

The Bag Class—Implementation

Or we could use the copy function from the standard library:

```
// inserts a copy of each item in @b into the bag
void operator +=(const bag& b) {
    assert(size() + b.size() <= CAPACITY);

    // copies all items in b.data to the end of data
    copy(b.data, b.data + b.used, data + used);

    // update the number of items in the bag
    used += b.used;
}
```

The Bag Class—Implementation

The `operator +` global function:

```
// returns a new bag that is the union of @b1 and @b2
bag operator +(const bag& b1, const bag& b2) {
    assert(b1.size() + b2.size() <= bag::CAPACITY;

    bag union;
    union += b1;
    union += b2;

    return union;
}
```

Complexity Analysis

Complexity Analysis

Here is some code to initialize elements in an array:

```
// create an array of N ints
int array[N];

// initialize each element
for (int i = 0; i < N; i++)
    array[i] = i;
```

The time this algorithm takes to run depends on the value of N

Complexity Analysis

The algorithm has a loop that goes once through each array element:

```
for (int i = 0; i < N; i++)
    array[i] = i;
```

Intuitively:

- if N were half as large, the algorithm would take half as long to run
- if N were twice as large, the algorithm would take twice as long to run

We say this algorithm is *linear* with respect to N

Complexity Analysis

Here is some more code to initialize a different array:

```
// create an array of N x N ints
int array[N][N];

// initialize each element
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        array[i][j] = i;
```

Again, the time this algorithm takes to run depends on the value of N

Complexity Analysis

This algorithm loops through N^2 elements:

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        array[i][j] = i;
```

Intuitively:

- if N were half as large, the algorithm would take one quarter as long to run
- if N were twice as large, the algorithm would take four times as long to run

We say this algorithm is *quadratic* with respect to N

Complexity Analysis

We use a shorthand notation called Big-O to describe complexity

- the first algorithm has a complexity of $O(n)$
- the second has a complexity of $O(n^2)$

Big-O notation:

- for any algorithm that has a function $g(n)$ that describes the time the algorithm takes to execute relative to n, we say that algorithm has complexity $O(g(n))$
- only include the fastest-growing term, ignoring constants and lower-degree terms

Example:

- $5n^4 + n + 1 \quad \Rightarrow O(n^4)$

Complexity Analysis

Frequently encountered functions (in order of increasing complexity):

- constant (1)
- logarithmic (log n)
- linear (n)
- log linear (n log n)
- quadratic (n^2)
- cubic (n^3)
- exponential (2^n)
- factorial (n!)

Complexity Analysis

And some corresponding values:

	n					
	1	**2**	**4**	**8**	**16**	**32**
1	1	1	1	1	1	1
log n	0	1	2	3	4	5
n	1	2	4	8	16	32
n log n	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
n!	1	2	24	40320	2.09×10^{13}	2.63×10^{35}

increasing complexity

Data Structures

We want an algorithm that is as efficient as possible...

- less complex g(n) => faster execution of our program

This entails choosing an efficient data structure to use!

- if an algorithm operates on some data, we want that data to be organized or stored in a way that minimizes the amount of steps we need to take

Example:

- phone companies publish phone books that are in alphabetical (sorted) order to minimize the amount of time it takes to find a person

Data Structures

Assume you are tasked with finding an integer in an array

If the list is unsorted:

- the array must be searched sequentially until the item is found
- on average, half the array will be searched before the value is found
- worst case, the entire array will be searched
- the algorithm using this unordered array (data structure) is linear, or $O(n)$

If the list is sorted:

- the value can be found using binary search
- the complexity in this case is $O(\log_2 n)$!

Data Structures

Assume you are tasked with finding an integer in an array

Comparison:

	n					
	1	10^2	10^4	10^6	10^8	10^{10}
$\log_2$ n	0	6.6	13.3	20	26.6	33.2
n	1	10^2	10^4	10^6	10^8	10^{10}

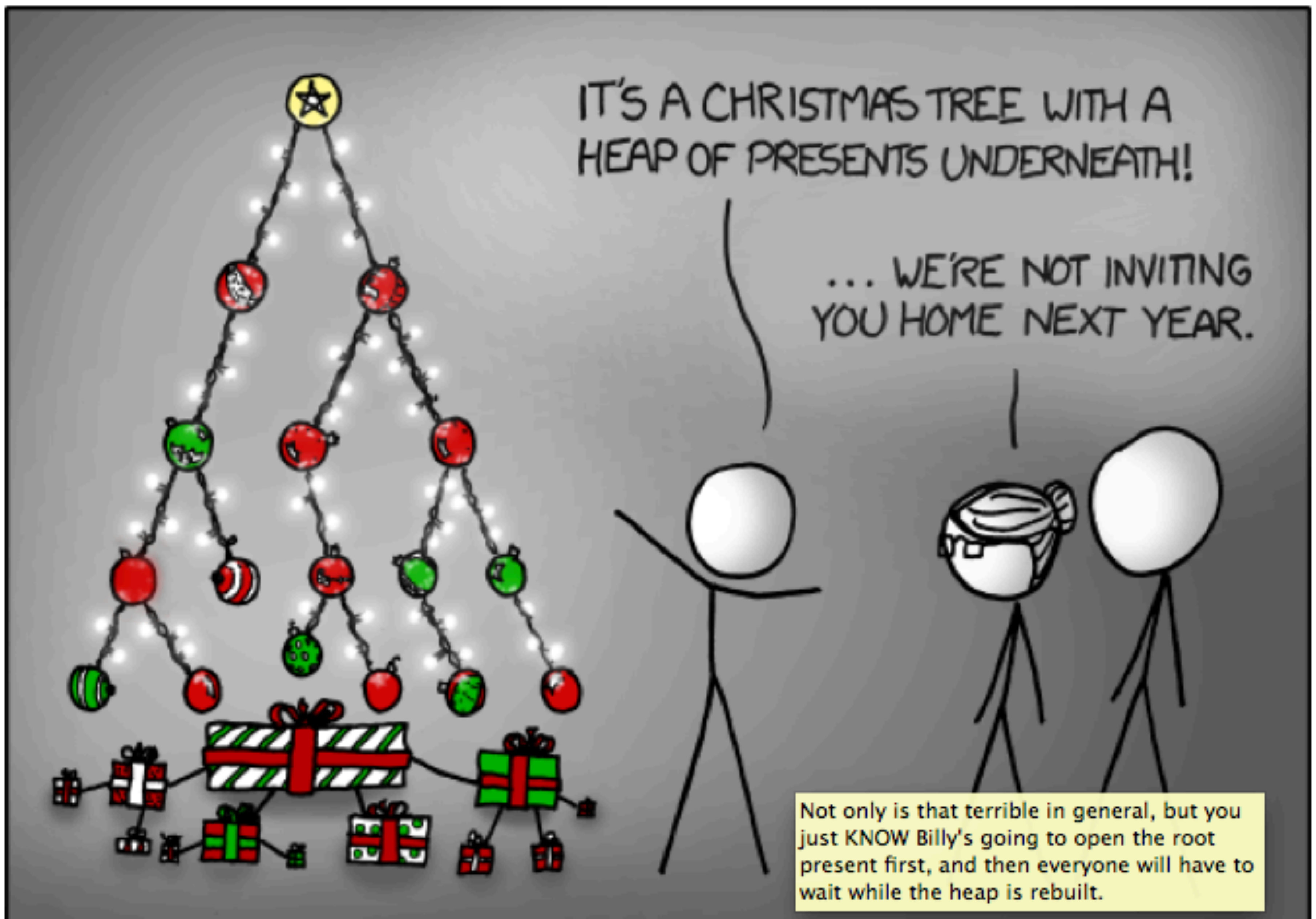
For large values of n, the $\log_2$ n algorithm will win every time!

- it doesn't matter how complicated the innards of the algorithm are...
- the overall complexity will eventually dominate (that's why we only consider the fastest-growing term)

Data Structures

Using the right data structure can have huge benefits for efficiency

- this is the whole reason we study data structures!
- there are countless examples in algorithms where using the right data structure greatly simplifies the overall complexity



Analysis of the Bag Methods

Analysis of the Bag Methods

Default constructor:

```
// default constructor creates an empty bag
bag() : used(0) { }
```

Complexity:

- only a single assignment operation
- worst case: O(1)

Analysis of the Bag Methods

Copy constructor:

```
// copy constructor duplicates an existing bag
// automatic implementation by C++
```

Complexity:

- each element in the existing bag is copied into the new one
- n copies are made, so n operations
- worst case: O(n)

Analysis of the Bag Methods

The count member function:

```
// returns the total number of occurrences of @t
bag::size_type bag::count(const value_type& t) const {
    size_type answer = 0;

    for (size_type i = 0; i < used; i++)
        if (data[i] == t)
            answer++;

    return answer;
}
```

Analysis of the Bag Methods

The `count` member function:

```
for (size_type i = 0; i < used; i++)
    if (data[i] == t)
        answer++;
```

All the work happens in the loop:

- entire array is traversed to find total number of occurrences of target value
- worst case: O(n)

Analysis of the Bag Methods

The `erase_one` member function:

```
// removes a single copy of @target from the bag
bool bag::erase_one(const value_type& target) {
    size_type i = 0;

    while (i < used && data[i] != target) i++;

    if (i == used) return false;

    data[i] = data[--used];
    return true;
}
```

Analysis of the Bag Methods

The `erase_one` member function:

```
// find the first occurrence of target in the array
while (i < used && data[i] != target) i++;
```

Again, the real work happens in the loop:

- the loop stops as soon as the target is found
- the target could be first element in the array and the loop stops immediately
- the target might not be in the array at all, in which case the entire array is scanned
- worst case: O(n)
- best case: O(1)
- average case: O(n)

Analysis of the Bag Methods

The `erase` member function:

```
// removes all copies of @target from the bag
bag::size_type bag::erase(const value_type& target) {
    size_type i = 0, num_erased = 0;
    while (i < used) {
        if (data[i] == target) {
            data[i] = data[--used];
            num_erased++;
        } else {
            i++;
        }
    }
    return num_erased;
}
```

Analysis of the Bag Methods

The `erase` member function:

```
// each item in the array must be checked
while (i < used) {
    // delete current element if equal to target
}
```

The loop dominates again!

- entire array must be traversed to remove all occurrences of target value
- worst case: O(n)

Analysis of the Bag Methods

The `operator +=` member function:

```
// inserts a copy of each item in @b into the bag
void operator +=(const bag& b) {
    assert(size() + b.size() <= CAPACITY);

    // copies all items in b.data to the end of data
    copy(b.data, b.data + b.used, data + used);
    used += b.used;
}
```

Analysis of the Bag Methods

The `operator +=` member function:

```
// copies all items in b.data to the end of data
copy(b.data, b.data + b.used, data + used);
```

Don't be fooled by the absence of a loop...

- this function still must perform a copy operation <u>for each element in the other bag</u>
- what happens in the copy function affects the complexity of this function
- worst case: O(n), where n is the size of the bag being added (RHS of += operator)

Analysis of the Bag Methods

The `operator +` global function:

```
// returns a new bag that is the union of @b1 and @b2
bag operator +(const bag& b1, const bag& b2) {
    assert(b1.size() + b2.size() <= bag::CAPACITY;

    bag union;
    union += b1;
    union += b2;

    return union;
}
```

Analysis of the Bag Methods

The `operator +` global function:

```
bag union;
union += b1;
union += b2;
```

Complexity:

- a call to the default constructor (constant time)
- the two calls to the += operator (linear in the size of the bag being added) dominate
- worst case: $O(n_1 + n_2)$, where n_1 and n_2 are the sizes of the two bags being added.

Analysis of the Bag Methods

The `insert` member function:

```
// inserts a new copy of @entry into the bag
void bag::insert(const value_type& entry) {
    assert(size() < CAPACITY);
    data[used] = entry;
    used++;
}
```

Complexity:

- only a couple of operations, regardless of the size of the bag
- worst case: O(1)

Analysis of the Bag Methods

The `size` member function:

```
// returns the total number of items in the bag
size_type size() const { return used; }
```

Complexity:

- only a single operation (returning a variable's value)
- worst case: O(1)

Analysis of the Bag Methods

Operation	Time Analysis	Comment
default constructor	O(1)	constant time
copy constructor	O(n)	n is the size of the bag being copied
count	O(n)	n is the size of the bag
erase_one	O(n)	linear time in worst case
erase	O(n)	linear time always
operator +=	O(n)	n is the size of the other bag
operator +	$O(n_1 + n_2)$	n_1 and n_2 are the sizes of the bags
insert	O(1)	constant time
size	O(1)	constant time