Slides by Andrew DeOrio,
Jeff Ringenberg
and Brian Noble

# EECS 280

## Programming and Introductory Data Structures

Container ADTs

# Review: Data Abstraction

- **Data abstraction** lets us separate *what* a type is (and what it can do) from *how* the type is implemented

- In C++, we use a `class` to implement data abstraction
  - We can create an Abstract Data Type (ADT) using a `class`

- ADTs let us model complex phenomena
  - More complex than built-in data types like `int`, `double`, etc.

- ADTs make programs easier to maintain and modify
  - You can change the implementation and no users of the type can tell

# Review: Information Hiding and Encapsulation

Information Hiding

- Protect and hide our code from other code that that uses it

Encapsulation

- Keeping data and relevant functions together

# Containers

- The purpose of a *container* is to hold other objects
- For example, an array can hold integers:

```
int array[10];
a[4] = 517;
```

- What if we want more features in our container? Say, a set of integers.
- This is a "set" in the mathematical sense: A collection of zero or more integers, with no duplicates.

# Set abstraction

- Let's create an abstraction that holds a set of integers

- There are four operations on this set that we will define:
  - Insert a value into the set
  - Remove a value from the set
  - Query to see if a value is in the set
  - Count the number of elements in the set

# IntSet Declaration

```
class IntSet {
  public:
    IntSet();
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int  size() const;

  //...
};
```

# IntSet Declaration

```
class IntSet {
  //...

  //EFFECTS: returns true if v is in set,
  //false otherwise
  bool query(int v) const;

  //EFFECTS: returns |set|
  int size() const;

  //...
};
```

# IntSet Representation

```
class IntSet {
  public:
    IntSet();
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int  size() const;
};
```

- The class is incomplete because we haven't chosen a representation for sets
- Choosing a representation involves two things:
  1. Deciding what concrete data elements will be used to represent the values of the set
  2. Providing an implementation for each method

# IntSet Representation

```
class IntSet {
  public:
    IntSet();
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int  size() const;
};
```

- Despite not having a representation for a set, the (incomplete) declaration is all that a **customer** of the `IntSet` abstraction needs to know since it has:
  - The general overview of the ADT.
  - The specification of each method.

# IntSet Representation

- Start with a representation invariant for the set:

- Use an array
- Represent a set of size N as an unordered array of integers with no duplicates, stored in the first N slots of the array.
- `int elts_size`: equal to the number of elements currently in the array.

# IntSet Representation

- Since this is an array, and arrays have maximum sizes, we have to choose a maximum size

```
int ELTS_CAPACITY = 100;
```

  - 100 is arbitrary, but soon we'll get to dynamic memory ☺
- Include the size in the `OVERVIEW`

```
//OVERVIEW: mutable set of ints,
|set|<=ELTS_CAPACITY
```

- Account for full sets in the `REQUIRES` clause for `insert`:

```
//REQUIRES: set is not full
//MODIFIES: this
//EFFECTS: set=set+{v}
void insert(int v);
```

# IntSet::size()

```
class IntSet {
    //OVERVIEW: mutable set of ints,|set|<=ELTS_CAPACITY
  public:
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int  size() const;
    static const int ELTS_CAPACITY = 100;
  private:
    int       elts[ELTS_CAPACITY];
    int       elts_size;
};
```

# static Member Variables

```
class IntSet {
public:
    static const int ELTS_CAPACITY = 100;
};
```

- `static` means "there's only one"
  - All instances of the class share this member variable
  - A lot like a global variable
- `const` means "you can't change this"

# static is Confusing!

```
static int add(int a, int b) {
  return a + b;
}
```

- Static function: visible only inside one file (internal linkage)

```
class IntSet {
public:
    static const int ELTS_CAPACITY = 100;
};
```

- Static member variable: all instances of the `IntSet` class share this member variable

# IntSet::size()

```
class IntSet {
  public:
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int  size() const;
    static const int ELTS_CAPACITY = 100;
  private:
    int       elts[ELTS_CAPACITY];
    int       elts_size;
};
```

```
int IntSet::size() const {
  return elts_size;
}
```

Given this representation, and the representation invariants, we can write the methods.

Because our rep invariant says that `elts_size` is always the size of the set, we can return it directly

# Searching IntSet

- Next, consider the three final member functions:
  - `query`: *search* the array looking for a specific number
  - `remove`: *search* the array for a number; if it exists, remove it
  - `insert`: *search* the array for a number; if it doesn't exist, add it

- All three of these have *search* in common.
- One might be tempted to just write `insert` and `remove` in terms of `query`, but `remove` won't work that way.
- `Query` only tells us **whether** the element exists, not **where** – we need one more method

# IntSet::indexOf()

This member function must be **`private`**. This is because it exposes details about the concrete representation. It is inappropriate to expose these details to a client of this class.

```
class IntSet {
  public:
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int  size() const;
    static const int ELTS_CAPACITY = 100;
  private:
    int       elts[ELTS_CAPACITY];
    int       elts_size;
    int indexOf(int v) const;
    // EFFECTS: returns the index of
    //          v if it exists in the
    //          array, ELTS_CAPACITY otherwise
};
```

# IntSet::indexOf()

```
class IntSet {
  public:
    //...
    static const int ELTS_CAPACITY = 100;
  private:
    int       elts[ELTS_CAPACITY];
    int       elts_size;
    int indexOf(int v) const;
};
int IntSet::indexOf(int v) const {
  for (int i = 0; i < elts_size; ++i) {
    if (elts[i] == v) return i;
  }
  return ELTS_CAPACITY;
}
```

# IntSet::query()

```
class IntSet { // OVERVIEW omitted for space
  public:
    bool query(int v) const;
    //...
    static const int ELTS_CAPACITY = 100;
  private:
    int       elts[ELTS_CAPACITY];
    int       elts_size;
    int indexOf(int v) const;
};
```

With `indexOf`, `query` is easy!

```
bool IntSet::query(int v) const  {
   return indexOf(v) != ELTS_CAPACITY;
}
```

# IntSet::insert()

- The code for `insert` is not much more difficult than query:

1. First look for the `indexOf` the element to insert
2. If it doesn't exist, we need to add this element to the "end" of the array
3. Using `elts_size`, the current "end" is `elts[elts_size-1]`
4. Place the element in the next slot and update `elts_size`

# IntSet::insert()

```
class IntSet {
  public:
    void insert(int v);
    bool query(int v) const;
    //...
    static const int ELTS_CAPACITY = 100;
  private:
    int       elts[ELTS_CAPACITY];
    int       elts_size;
    int indexOf(int v) const;
};
void IntSet::insert(int v) {
  if (query(v)) return;
  assert(elts_size < ELTS_CAPACITY); //REQUIRES!
  elts[elts_size++] = v;
}
```

Take a couple minutes to figure this out.

# IntSet::remove ()

- `remove` works similarly to `insert`.

1. If the element (called the `victim`) is in the array we have to remove it leaving a "hole" in the array
2. Instead of moving each element after the `victim` to the left by one position, pick up the current "last" element and move it to the hole

- This again breaks the invariant on `elts_size`, so we must fix it

# IntSet::remove ()

```
void IntSet::remove(int v) {
  int victim = indexOf(v);
  if (victim == ELTS_CAPACITY) return;//not found
  elts[victim] = elts[--elts_size];
}
```

Take a couple minutes to figure this out

# Initialization Exercise

- Consider the newly-created set

```
int main() {
  IntSet s;
}
```

- **Problem**: On creation, `s`'s data members are uninitialized!
- This means that the value of `elts_size` could be anything, but our representational invariant says it must be zero!

- Exercise: implement the `IntSet` default constructor

# Initialization Exercise

```
IntSet::IntSet()
   : elts_size(0) {}
```

# Exercise

- Add a `print()` member function
- "Promise" not to modify any member variables

```
class IntSet {
  public:
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int  size() const;
    static const int ELTS_CAPACITY = 100;
  private:
    int       elts[ELTS_CAPACITY];
    int       elts_size;
};
```

# Exercise

```
class IntSet {
  //...
  //EFFECTS: prints set
  void print() const;
};

void IntSet::print() const {
  cout << "{ ";
  for (int i=0; i<elts_size; ++i)
    cout << elts[i] << " ";
  cout << "} "<< endl;
}
```

# Using IntSet

```
int main () {
   IntSet is;
   is.insert(7);
   is.insert(4);
   is.insert(7);
   is.print();
   is.remove(7);
   is.print();
}
```

```
./a.out
{ 7 4 }
{ 4 }
```

# IntSet efficiency

```
int IntSet::indexOf(int v) const {
  for (int i = 0; i < elts_size; ++i) {
    if (elts[i] == v) return i;
  }
  return ELTS_CAPACITY;
}
```

- **Question**: How many elements of the array must `indexOf` examine in the worst case if there are 10 elements? If there are 90 elements?

# IntSet efficiency

- We say the time for `indexOf` grows *linearly* with the size of the set.

- If there are N elements in the set, we have to examine all N of them in the worst case. For large sets that perform lots of queries, this might be too expensive.

- Luckily, we can replace this implementation with a different one that can be more efficient. The only change we need to make is to the representation – the abstraction can stay precisely the same.

# Improving IntSet efficiency

- Still use an array (of 100 elements) to store the elements of the set and the values will still occupy the first `elts_size` slots.

- *However, now we'll keep the elements in sorted order*

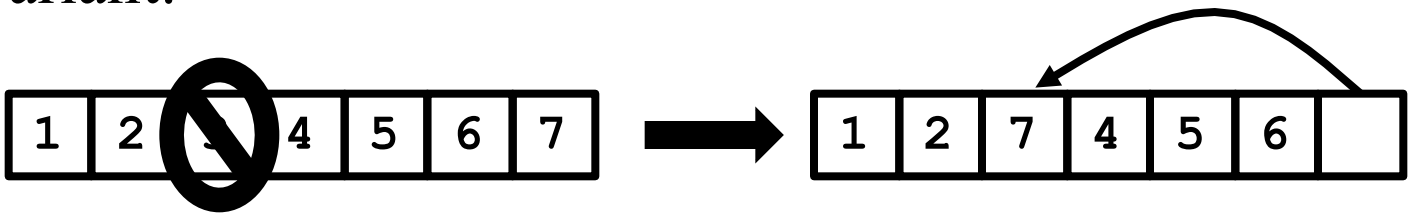
- The constructor and size methods don’t need to change at all since they just use the `elts_size` field.

- `query` also doesn't need to change. If the index exists in the array’s legal bounds, then it’s there.

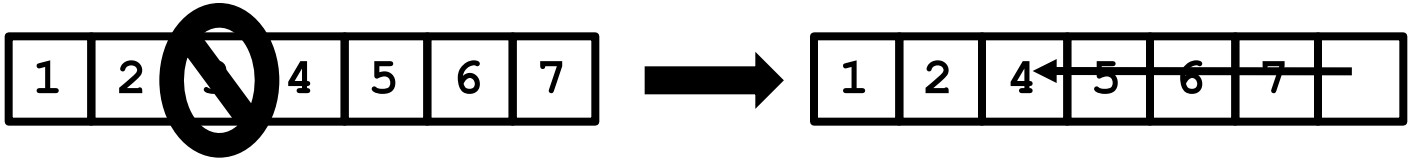
# Improving IntSet efficiency

- However, the others all do need to change. We'll start with the easiest one: `remove()`

- Recall the old version that moved the last element from the end to somewhere in the middle, this will break the new "sorted" invariant.




- Instead of doing a swap, we have to "squish" the array together to cover up the hole.



# Improving IntSet efficiency

- How are we going to do the “squish”?

- Move the element next to the hole to the left leaving a new hole
- Keep moving elements until the hole is “off the end” of the elements

# Improving IntSet efficiency

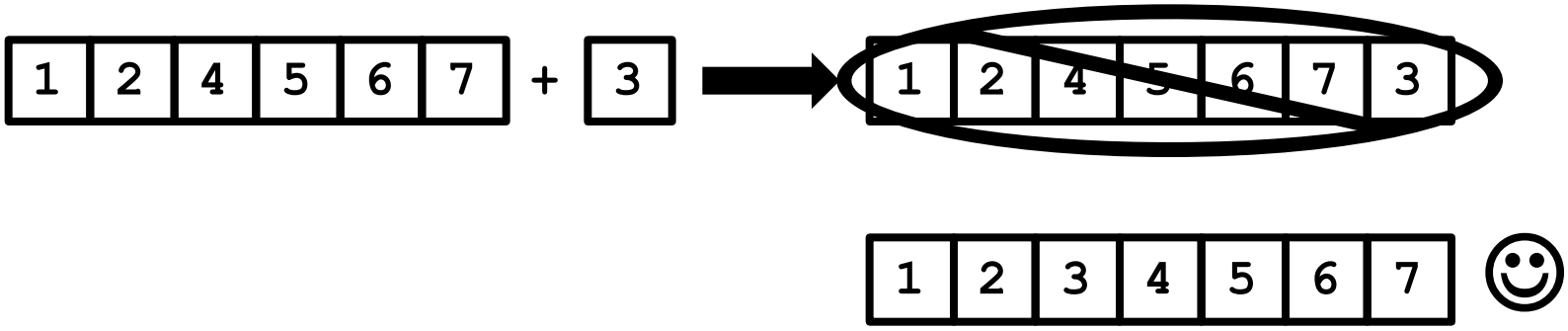
```
void IntSet::remove(int v) {
  int gap = indexOf(v);
  if (gap == ELTS_CAPACITY) return; //not found
  --elts_size; //one less element

  while (gap < elts_size) {
    //there are elts to our right
    elts[gap] = elts[gap+1];
    ++gap;
  }
}
```

Take a couple minutes to figure this out

# Improving IntSet efficiency

- We also have to change `insert` since it currently just places the new element at the end of the array. This also will break the new "sorted" invariant.



# Improving IntSet efficiency

- How are we going to do the insert?

- Start by moving the last element to the right by one position.
- Repeat this process until the correct location is found to insert the new element.
- Stop if the start of the array is reached or the element is sorted.
- We'll need a new loop variable to track this movement called `cand`(idate).
- It's invariant is that it always points to the next element that might have to move to the right.

# Improving IntSet efficiency

```
void IntSet::insert(int v) {
  if (indexOf(v) != ELTS_CAPACITY) return;//present!
  assert(elts_size < ELTS_CAPACITY); //REQUIRES
  int cand = elts_size-1; // largest element
  while ((cand >= 0) && elts[cand] > v) {
    elts[cand+1] = elts[cand];
    --cand;
  }
  // Now, cand points to the left of the "gap"
  elts[cand+1] = v;
  ++elts_size; // repair invariant
}
```

Take a couple minutes to figure this out

# Improving IntSet efficiency

```
void IntSet::insert(int v) {
  if (indexOf(v) != ELTS_CAPACITY) return;//present!
  assert(elts_size <= ELTS_CAPACITY) //enough room?
  int cand = elts_size-1; // largest element
  while ((cand >= 0) && elts[cand] > v) {
    elts[cand+1] = elts[cand];
    --cand;
  }
  // Now, cand points to the le
  elts[cand+1] = v;
  ++elts_size; // repair invari
}
```

We are using the "short-circuit" property of `&&`. If `cand` is not greater than or equal to zero, we never evaluate the right-hand clause.

# Improving IntSet efficiency

- **Question**: Do we have to change `indexOf`?

```
int IntSet::indexOf(int v) const {
  for (int i = 0; i < elts_size; ++i) {
    if (elts[i] == v) return i;
  }
  return ELTS_CAPACITY;
}
```

# Improving IntSet efficiency

- **Question**: Do we have to change `indexOf`?

- **Answer**: No, but it can be made more efficient with the new representation

```
int IntSet::indexOf(int v) const {
  for (int i = 0; i < elts_size; ++i) {
    if (elts[i] == v) return i;
  }
  return ELTS_CAPACITY;
}
```

# Improving IntSet efficiency

- Suppose we are looking for `foo`.
- Compare `foo` against the **middle** element of the array and there are three possibilities:
  1. `foo` is equal to the middle element.
  2. `foo` is less than the element.
  3. `foo` is greater than the element.
- If it's case 1, we're done.
- If it's case 2, then if `foo` is in the array, it must be to the **left** of the middle element
- If it's case 3, then if `foo` is in the array, it must be to the **right** of the middle element.

# Improving IntSet efficiency

- Compare `foo` against the **middle** element of the array and there are three possibilities:
  1. `foo` is equal to the middle element.
  2. `foo` is less than the element.
  3. `foo` is greater than the element.
- The comparison with the middle element eliminates at least half of the array from consideration! Then, we repeat the same thing over again.
- You could write this "repetition" as either a tail-recursive program or an iterative one. Most programmers find the iterative version more natural, so we'll write it iteratively, too.

# Improving IntSet efficiency

- First, we need to find the "bounds" of the array.
- The "leftmost" element is always zero, but the "rightmost" element is `elts_size-1`.

```
int IntSet::indexOf(int v) const {
  int left = 0;
  int right = elts_size-1;
  ...
}
```

# Improving IntSet efficiency

- It's possible that the segment we are examining is empty and we return `ELTS_CAPACITY` since the element is missing.
- A nonempty array has at least one element in it, so `right` is at least as large as `left` (`right >= left`).

```
int IntSet::indexOf(int v) const {
  int left = 0;
  int right = elts_size-1;
  while (right >= left) {
    ...
  }
  return ELTS_CAPACITY;
}
```

# Improving IntSet efficiency

- Next, find the "middle" element. We do this by finding out the size of our segment (`right - left + 1`), then divide it by two, and add it to `left`.

```
int IntSet::indexOf(int v) const {
  int left = 0;
  int right = elts_size-1;
  while (right >= left) {
    int size = right - left + 1;
    int middle = left + size/2;
    ...
  }
  return ELTS_CAPACITY;
}
```

# Improving IntSet efficiency

- Next, find the "middle" element. We do this by finding out the size of our segment (`right - left + 1`), then divide it by two, and add it to `left`.

```
int IntSet::indexOf(int v) const {
  int left = 0;
  int right = elts_size-1;
  while (right >= left) {
    int size = right - left + 1;
    int middle = left + size/2;
    ...
  }
  return ELTS_CAPACITY;
}
```

If there is an odd number of elements, this will be the "true" middle. If there are an even number, it will be the element to the “right" of true middle.

# Improving IntSet efficiency

- Then, we compare against the middle element.
- If that's the one we are looking for, we are done.

```
int IntSet::indexOf(int v) const {
  int left = 0;
  int right = elts_size-1;
  while (right >= left) {
    int size = right - left + 1;
    int middle = left + size/2;
    if (elts[middle] == v) return middle;
    ...
  }
  return ELTS_CAPACITY;
}
```

# Improving IntSet efficiency

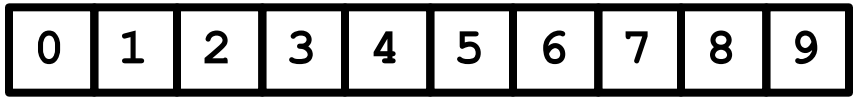
- If we are not looking at the element, the true element (if it exists) must be in either the "smaller" half or the "larger" half.

- If it is in the smaller half, than we can eliminate all elements at index `middle` and higher, so we move "right" to `middle-1`.
- Likewise, if it would be in the larger half, we move "left" to `middle+1`, and we continue looking.

# Improving IntSet efficiency

```
int IntSet::indexOf(int v) const {
    int left = 0;
    int right = elts_size-1;
    while (right >= left) {
      int size = right - left + 1;
      int middle = left + size/2;
      if (elts[middle] == v)
        return middle;
      else if (elts[middle] < v)
        left = middle+1;
      else
        right = middle-1;
    }
    return ELTS_CAPACITY;
  }
```

Take a couple minutes to figure this out. Try using the inputs: `v=3`, `v=8`, and `v=-1` with the array below.



# Improving IntSet efficiency

- Since you eliminate half of the array with each comparison, this is a much more efficient.
- If the array has N elements, you'll need "about" $\log_2(N)$ comparisons to search it.
- This is really cool, because $\log_2(100)$ is less than 7 – so, we need only 7 comparisons in the **worst case**.
- Also, if you double the size of the array, you need only one extra comparison to do the search.

- This is called a *binary search*

# Improving IntSet efficiency

- `insert` and `remove` are still linear, because they may have to "swap" an element to the beginning/end of the array.
- Here is the summary of asymptotic performance of each function:

| | Unsorted | Sorted |
|---|---|---|
| `insert` | `O(N)` | `O(N)` |
| `remove` | `O(N)` | `O(N)` |
| `query` | `O(N)` | `O(log N)` |

All the unsorted versions still require the unsorted `indexOf()` which is `O(N)`

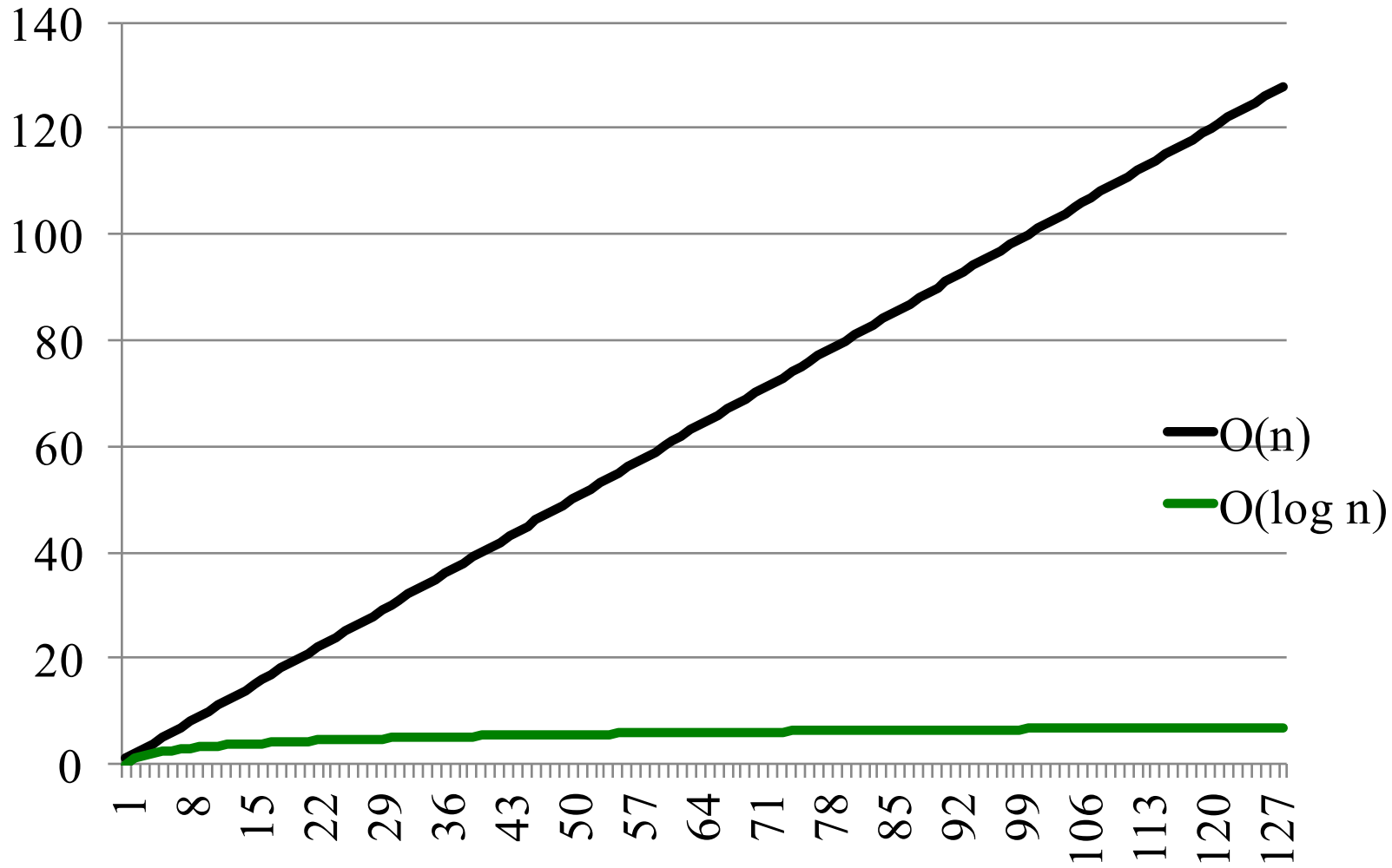
# Improving IntSet efficiency

| | Unsorted | Sorted |
|---|---|---|
| `insert` | `O(N)` | `O(N)` |
| `remove` | `O(N)` | `O(N)` |
| `query` | `O(N)` | `O(log N)` |

- If you are going to do more searching than inserting/removing, you should use the "sorted array" version, because `query` is faster there.

- However, if `query` is relatively rare, you may as well use the "unsorted" version. It's "about the same as" the sorted version for `insert` and `remove`, but it's MUCH simpler!

# Log(n) run time. So what?



# Abstraction

We modified `InSet`. Do we need to change our `main()` program? Why?

```
int main () {
   IntSet is;
   is.insert(7);
   is.insert(4);
   is.insert(7);
   is.print();
   is.remove(7);
   is.print();
}
```

# Abstraction

We modified `InSet`. Do we need to change our `main()` program? Why?

No modification needed, because of data abstraction