Ve 280

Programming and Introductory Data Structures

ADT Example: IntSet;

Improve ADT Efficiency;

Outline

- A Class Example: a Mutable Set of Integers (IntSet)
- Improve the Efficiency of IntSet

Abstract Data Types

Using Classes

- Next, consider the three final routines:
  - 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`, will this work?
  - Hint: think about `remove`.
- `query` only tells us **whether** the element exists, not **where** – we need one more method…

Abstract Data Types

Using Classes

```
const int MAXELTS = 100;
class IntSet { // OVERVIEW omitted for space
   int       elts[MAXELTS];
   int       numElts;
   int indexOf(int v);
     // EFFECTS: returns the index of
     //          v if it exists in the
     //          array, MAXELTS otherwise.
  public:
    void insert(int v);
    void remove(int v);
    bool query(int v);
    int  size();
};
```

__Note__: 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 user of this class.

Abstract Data Types

Using Classes

```
const int MAXELTS = 100;
class IntSet { // OVERVIEW omitted for space
    int        elts[MAXELTS];
    int        numElts;
    int indexOf(int v); // RME omitted for space
  public:
    void insert(int v);  void remove(int v); // RME omitted
    bool query(int v);   int  size();        // RME omitted
};
```

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

Abstract Data Types

Using Classes

```
const int MAXELTS = 100;
class IntSet { // OVERVIEW omitted for space
    int       elts[MAXELTS];
    int       numElts;
    int indexOf(int v); // RME omitted for space
  public:
    void insert(int v);  void remove(int v); // RME omitted
    bool query(int v);   int  size();        // RME omitted
};
```

With `indexOf`, `query` is trivial…

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

Abstract Data Types

Using Classes

- The code for `insert` is not much more difficult than query:
  - First look for the `indexOf` the element to insert.
  - If it doesn't exist, we need to add this element to the **end** of the array.
  - What is the index of the current "end" ?
  - Place the element in the next slot and **update** `numElts`.
  - The only exception to this is if `numElts` already equals `MAXELTS`.

Abstract Data Types

Using Classes

```
const int MAXELTS = 100;
class IntSet { // OVERVIEW omitted for space
    int        elts[MAXELTS];
    int        numElts;
    int indexOf(int v); // RME omitted for space
  public:
    void insert(int v);  void remove(int v); // RME omitted
    bool query(int v);   int  size();        // RME omitted
};
```

```
void IntSet::insert(int v) {
  if (indexOf(v) == MAXELTS) {
    if (numElts == MAXELTS) throw MAXELTS;
    elts[numElts++] = v;
  }
}
```

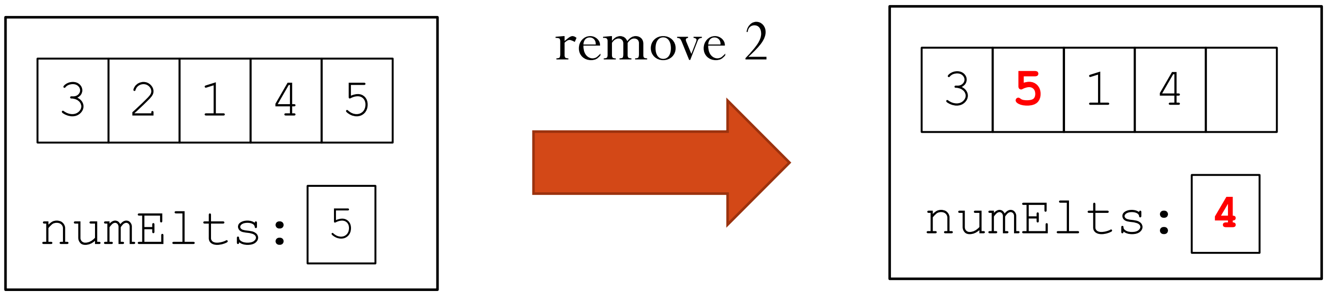
How about Remove?

- If the element (called the `victim`) is in the array, we have to remove it leaving a "hole" in the array.

- What representation invariants are violated?
  - How can we fix them?

How about Remove?

- 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 also breaks the invariant on `numElts`, so we must fix it.



Abstract Data Types

Using Classes

```
void IntSet::remove(int v) {
  int victim = indexOf(v);
  if (victim != MAXELTS) {
    elts[victim] = elts[numElts-1];
    numElts--;
  }
}
```

Abstract Data Types

Using Classes

- Question: There is one problem with our implementation. What is it?

- Hint: Consider the newly-created set:

```
IntSet s;
```

What does the computer actually create when we declare `s`?

Abstract Data Types

Using Classes

- Question: There is one problem with our implementation. What is it?

- Answer: On creation, `s`'s data members are **uninitialized**!

- This means that the value of `numElts` could be a random value, but our representational invariant says it must be zero!

- How can we fix this?

Abstract Data Types

Automatically Initializing Classes

- Using **constructor**!
- The constructor (really, the **default** constructor) has the following type signature:

```
class IntSet { // OVERVIEW omitted for space
     …
  public:
     IntSet();
        // EFFECTS: creates an empty IntSet
     …
};
```

Abstract Data Types

Automatically Initializing Classes

```
IntSet();
  // EFFECTS: creates an empty IntSet
```

- **The name of the function is the same as the name of the class.**
- **This function doesn't have a return type.**
- **It also does not take an argument in this case.**
- It is guaranteed to be the **first** function called immediately after an object is created.
- It builds a "blank" uninitialized `IntSet` and makes it satisfy the rep invariant.

Abstract Data Types

Automatically Initializing Classes

```
IntSet();
  // EFFECTS: creates an empty IntSet
```

- Here's how it's written:

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

Abstract Data Types

Automatically Initializing Classes

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

```
Class_T::Class_T(): anInt(0),
     aDouble(1.2),
     aString("Yes")
{
}
```

- This syntax is called "initialization syntax".
- Each data member is initialized this way.
- **<u>Note</u>**: The order in which elements are initialized is the order they **appear in the definition**, NOT the order in the initialization list. It is a good practice to keep them in the same order to avoid confusion.

Abstract Data Types

Automatically Initializing Classes

- Alternatively, we could write this function as follows, but this is not considered as a good way!

```
IntSet::IntSet()
{
    numElts = 0;
}
```

Abstract Data Types

A Benefit of Classes

- Now, instead of writing this:

```
void add_one (int a[], int elts);
```

and having to worry about the number of elements in the array. All we have to write is this:

```
void add_one (IntSet& set);
```

and we no longer have to worry about the array and its count being separated.

Const Member Functions

- A slight change to the class definition:

```
const int MAXELTS = 100;
class IntSet {
   int        elts[MAXELTS];
   int        numElts;
   int indexOf(int v) const;

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

Const Member Functions

```
int size() const;
```

- Each member function of a class has an extra, implicit parameter named `this`.
  - "`this`" is a pointer to the current instance on which the function is invoked.
- `const` keyword modifies the implicit `this` pointer: `this` is now a pointer to a **const instance**.
  - Means: the member function `size()` cannot change the object on which `size()` is called.
  - By its definition, `size()` shouldn't change the object! Adding `const` keyword prevents any accidental change.
  - It is a good practice to add `const` keyword when possible!

Const Member Functions

- Implement **`size()`**

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

The function body is the same as before.

- A **const** object can only call its **const** member functions!

```
const IntSet is;
cout << is.size(); ✓
is.insert(2); ✗
```

Const Member Functions

- If a const member function calls other **member** functions, they must be **const** too!

```
void A::g() const { f(); }
```

```
void A::f() {...}
```

✘

```
void A::f() const {...}
```

✓

Outline

- A Class Example: a Mutable Set of Integers (IntSet)

- Improve the Efficiency of IntSet

Abstract Data Types

Class Exercise

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

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

Abstract Data Types

Improving 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 is 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 (implementation)** – the abstraction can stay precisely the same.

Abstract Data Types

Improving Efficiency

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

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

Question: What Parts of the Class Should Be Changed?

```
const int MAXELTS = 100;
class IntSet {
    // OVERVIEW: a mutable set of integers
    int elts[MAXELTS];
    int numElts;
    int indexOf(int v) const;
  public:
    IntSet();
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int  size() const;
};
```

Abstract Data Types

Improving Efficiency

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

- `query` also doesn't need to change.

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

- `indexOf` also doesn't need to change.

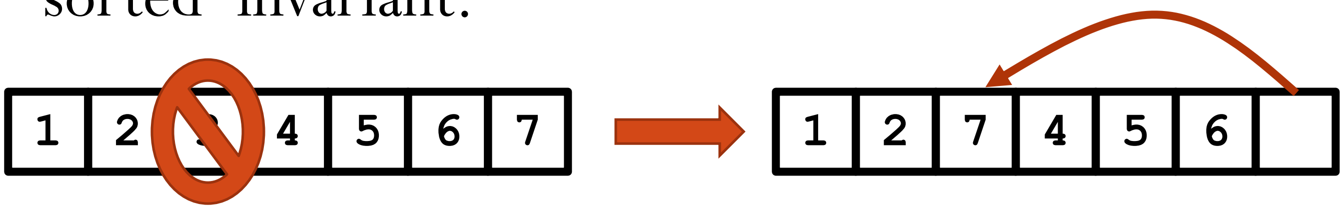
- However, `insert` and `remove` do need to change.

Abstract Data Types

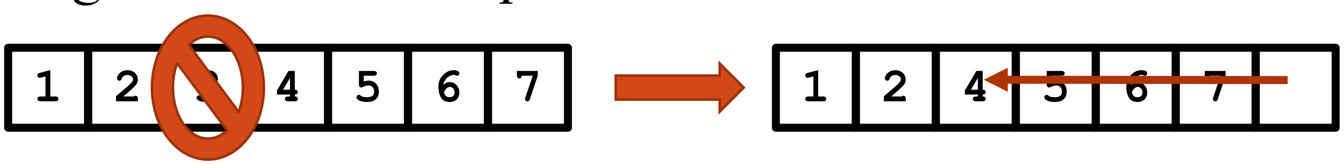
Improving Efficiency

- 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.



Abstract Data Types

Improving 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.

1	2	4	6	7	

remove 4

1	2		6	7	

1	2	6		7	

1	2	6	7		

  - We'll reuse the variable `victim` as a loop variable.
  - `victim`’s invariant is that it always points at the hole in the array.

Abstract Data Types

Improving Efficiency

```
void IntSet::remove(int v) {
  int victim = indexOf(v);
  if (victim != MAXELTS) {
      // victim points at hole
    numElts--; // one less element
    while (victim < numElts) {
      // ..hole still in the array
      elts[victim] = elts[victim+1];
      victim++;
    }
  }
}
```

Abstract Data Types

Improving 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.

Abstract Data Types

Improving Efficiency

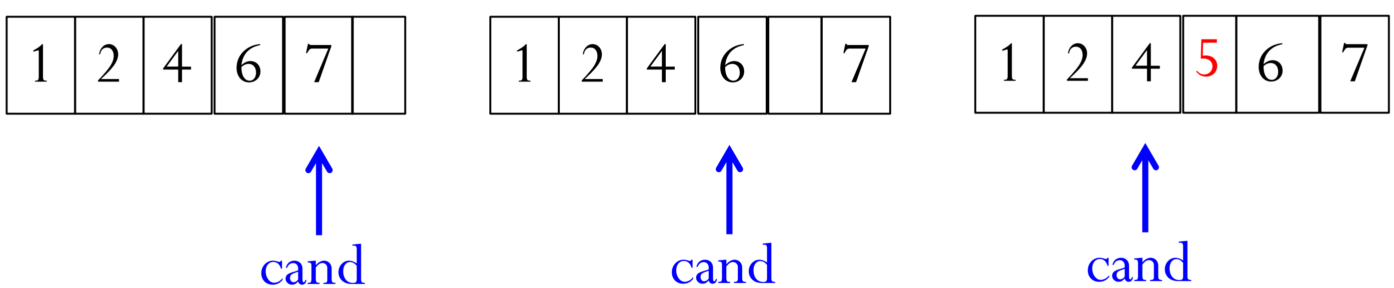
- How are we going to do the insert?
  - Start by moving the last element to the right by one position.
  - Repeat this process until the correct location is found to insert the new element.
  - Stop if the start of the array is reached or the element is sorted.
  - We'll need a new loop variable called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

Abstract Data Types

Improving Efficiency

```
void IntSet::insert(int v) {
  if (indexOf(v) == MAXELTS) { // duplicate not found
    if (numElts == MAXELTS) throw MAXELTS; // no room
    int cand = numElts-1; // last 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;
    numElts++; // repair invariant
  }
}
```

insert 5



Abstract Data Types

Improving Efficiency

```
void IntSet::insert(int v) {
  if (indexOf(v) == MAXELTS) { // duplicate not found
    if (numElts == MAXELTS) throw MAXELTS; // no room
    int cand = numElts-1; // last element
    while ((cand >= 0) && elts[cand] > v) {
      elts[cand+1] = elts[cand];
      cand--;
    }
    // Now, cand points to the
    elts[cand+1] = v;
    numElts++; // repair invar
  }
}
```

Note: 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.

Abstract Data Types

Improving Efficiency

```
void IntSet::insert(int v) {
  if (indexOf(v) == MAXELTS) { // duplicate not found
    if (numElts == MAXELTS) throw MAXELTS; // no room
    int cand = numElts-1; // largest (last) 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;
    numElts++; // repair invariant
  }
}
```

Question: What is the situation when the loop terminates due to `cand < 0`? Is our implementation correct?

Abstract Data Types

Improving Efficiency

- **<u>Question</u>**: Do we have to change `indexOf`?

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

Abstract Data Types

Improving Efficiency

- **<u>Question</u>**: Do we have to change `indexOf`?
- **<u>Answer</u>**: No, but it can be made more efficient with the new representation.
- **<u>How?</u>**

Using **binary search**! (The array is sorted)

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

Abstract Data Types

Complexity

	Unsorted	Sorted
query	O(N)	?
insert	?	?
remove	?	?

Abstract Data Types

Complexity

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

`insert` and `remove` are still **linear**, because they may have to "swap" an element to the beginning/end of the array.

Abstract Data Types

Complexity

	Unsorted	Sorted
`query`	`O(N)`	`O(log N)`
`insert`	`O(N)`	`O(N)`
`remove`	`O(N)`	`O(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!

References

- Constructor
  - Problem Solving with C++, 8th Edition, Chapter 10.2
- const Member Function
  - C++ Primer, 4th Edition, Chapter 7.7.1