Ve 280

Programming and Elementary Data Structures

Dynamic Resizing

Learning Objectives:

Learn a basic strategy for managing a dynamic storage using arrays

Learn some basics of complexity analysis

Dynamic Resizing

Modifying `Insert()`

- We have modified `IntSet` to allow a client to specify the **capacity** of an `IntSet`.
- However, this doesn't really get around the "big instance" problem, since the caller itself might not know how big the set will grow.
- So, what we **really** want to do is to create an `IntSet` that can **grow** as big as it needs to.
- To do this, we only need to modify the `insert()` method.

Dynamic Resizing

Modifying `Insert()`

- We will use the unsorted representation. We will focus on the action of **resizing**, not the action of inserting.

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

We want to modify `throw sizeElts`

Dynamic Resizing

Modifying `Insert()`

- Rather than throw an exception if the array is at maximum capacity, we will instead **grow** the array.

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

Dynamic Resizing

Modifying `Insert()`

- The `grow` method won't take any arguments or return any values.
- It should **never** be called from outside of the class, so add it as a **private** method taking no arguments and returning void.

```
class IntSet {
  // data members ...
  void grow();
  // EFFECTS: enlarge the elts array,
  //          preserving current contents
public:
  // ...
};
```

Dynamic Resizing

Modifying `Insert()`

- `grow` will look like the assignment operator.

- It must perform the following steps:
  1. Allocate a bigger array.
  2. Copy the smaller array to the bigger one.
  3. Destroy the smaller array.
  4. Modify elts/sizeElts to reflect the new array.

Note the order of allocation and destroy. Can we switch this order?

Dynamic Resizing

Modifying `Insert()`

```
void grow() {
   int *tmp = new int[sizeElts + 1];
   for (int i = 0; i < numElts; i++) {
      tmp[i] = elts[i];
   }
   delete [] elts;
   elts = tmp;
   sizeElts += 1;
}
```

1. Allocate a bigger array.
2. Copy the smaller array to the bigger one.
3. Destroy the smaller array.
4. Modify elts/sizeElts to reflect the new array.

Dynamic Resizing

Group Exercise – Modifying `Insert()`

- Unfortunately, we might end up doing a lot of copying.
- Suppose a client creates an `IntSet` of capacity 1, and then inserts N elements into it.
- **<u>Question</u>**: What's the number of integer copies performed by the function `grow` in the worst case?

```
void grow() {
  int *tmp = new int[sizeElts + 1];
  for (int i = 0; i < numElts; i++) {
    tmp[i] = elts[i];
  }
  delete [] elts;
  elts = tmp;
  sizeElts += 1;
}
```

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

Dynamic Resizing

Group Exercise – Modifying `Insert()`

- Suppose a client creates an `IntSet` of capacity 1, and then inserts N elements into it. What's the number of integer copies in the worst case?

<u>Answer</u>:

- The worst case happens when all the elements inserted are different!
- Before each new insertion, `numElts == sizeElts`.
- We need to call grow each time we insert a new element.
- When we grow an array of size k to one of size k+1, we copied k items.
- We did this for k from 1 to N-1.
- So the total number of copies is:

$$1 + 2 + \ldots + (N-2) + (N-1) = N(N-1)/2$$

Dynamic Resizing

Group Exercise – Modifying `Insert()`

- Suppose a client creates an `IntSet` of capacity 1, and then inserts N elements into it. What's the number of integer copies in the worst case?

<u>Answer</u>:

- $N(N-1)/2$
- This is a quadratic function in N.
- This means that as the `IntSet` grows, the cost to build the `IntSet` grows much faster.

- **How can we make this better?**

Dynamic Resizing

Optimizing `grow()`

- **How can we make `grow()` better?**

- The intuition is that we aren't buying enough room each time we copy the array:
  - We copy N things, but only buy room for one more slot.

- Instead, we'd like to buy more slots for each N things we copy.
- The new version is only **slightly** different from the old version.
- However, it has **very** different performance characteristics.

Dynamic Resizing

Optimizing `grow()`

```
void grow() {
  int *tmp = new int[sizeElts * 2];
  for (int i = 0; i < numElts; i++) {
    tmp[i] = elts[i];
  }
  delete [] elts;
  elts = tmp;
  sizeElts *= 2;
}
```

Instead of growing the array by one, we double it.

Dynamic Resizing

Group Exercise – Optimizing `grow()`

- Suppose a client creates an `IntSet` of capacity 1, and then inserts N elements into it using the new version of `grow()`.
- **<u>Question</u>**: What's the number of integer copies performed by the function `grow` in the worst case?

```
void grow() {
  int *tmp = new int[sizeElts * 2];
  for (int i = 0; i < numElts; i++) {
    tmp[i] = elts[i];
  }
  delete [] elts;
  elts = tmp;
  sizeElts *= 2;
}
```

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

Dynamic Resizing

Group Exercise – Optimizing `grow()`

Answer:

- After the first `grow`, the capacity is 2. We copy 1 item.
- After the second `grow`, the capacity is 4. We copy 2 items.
- After the k-th `grow`, the capacity is 2^k. We copy 2^{k-1} items.

- Suppose $2^m < N \leq 2^{m+1}$
- How many times we need to call grow?
  - m+1 times
- How many copies we perform?
  - $T = 1+2+4+ \ldots +2^m = 2^{m+1} - 1 < 2N$

Dynamic Resizing

Group Exercise – Optimizing `grow()`

<u>**Answer**</u>:

- `T` (the number of copies) $<$ `2N`
- So, instead of copying almost (N-1)N/2 elements, we copy fewer than 2N of them.

Dynamic Resizing

Group Exercise – Optimizing `grow()`

Answer:

- Here's a little table showing what this means:

# elements	(N-1)N/2	2N
1	0	2
8	28	16
64	2016	128
512	130816	1024
2048	2096128	4096

- The "double" implementation is **much** better than the "by-one" implementation.

Reference

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 11.4 Classes and Dynamic Arrays