Today

- Resizable Arrays (cont)
- Algorithm Analysis (intro)

Assignments

- HW 2 due
- HW 3 out
The ArraySeq class

For HW-3, you will implement a “resizable array” (called ArraySeq)
  • Similar to Java’s ArrayList and C++’s Vector classes

Resizable arrays address fixed-sized issue of normal arrays
  • Arrays in C++ are a fixed size at creation
  • Once we reach the capacity of the array, can’t add more elements

Basic idea of a resizable array:
  • start with an initial capacity (in our case, 0 slots)
  • once all slots full, next add dynamically “resizes” the array

Resizing the array involves four steps:
  1. create a new array with twice the current array’s capacity (e.g., 1, 2, 4, 8, ...)
  2. copies all current elements to the new array
  3. deletes the current array
  4. sets the current array to the new array

Resizable arrays introduce a “trade off” in terms of efficiency
  • Solves the fixed-size problem
  • Maintains fast access to elements
  • (Slightly) increases the cost of adding (because of resizing)
The basic structure of ArraySeq for HW-3

```cpp
template<typename T>
class ArraySeq : public Sequence<T>
{

  // constructor, copy, move, destructor
  ArraySeq();
  ArraySeq(const ArraySeq& rhs);
  ArraySeq(ArraySeq&& rhs);
  ArraySeq& operator=(const ArraySeq& rhs);
  ArraySeq& operator=(ArraySeq&& rhs);
  ~ArraySeq();

  // sequence operations
  int size() const;
  bool empty() const;
  T& operator[](int index);
  const T& operator[](int index) const;
  void insert(const T& elem, int index);
  void erase(int index);
  bool contains(const T& elem) const;

  private:

  // the underlying (resizable) array
  T* array = nullptr;

  // size of list
  int count = 0;

  // max capacity of the array
  int capacity = 0;

  // helper to double capacity of the array
  void resize();

  // helper to delete the array list
  void make_empty();

};
```
Q: Why does the array have the following type?

```cpp
T* array = nullptr;
```

Q: How do we dynamically allocate an array of size $n$?

```cpp
array = new T[n];
```

Q: How do we deallocate an array (as opposed to a single item)?

```cpp
delete [] array; // reclaims all $n$ allocated items
```

Q: What does it mean to do a “deep copy” of an array?

- assuming we have an “old” array of size $m$ and a “new” array of capacity $n$ ...
- copy each item one at a time
- copy all $m$ elements from old array to same positions in new array

Note we never initialize the elements of the resizable array ...

- we track the amount filled via the `count`
- `count ≤ capacity` is always true

Q: How do insert and erase work?

- by shifting array values to the left or right

We are never decreasing the capacity of an array ... only growing it

- could lead to wasted space
**Basic Algorithm Analysis Ideas**

**Input size** $n$

- We analyze algorithms in terms of the size of their input (e.g., $n$ elements)
- The idea is that algorithms often taken longer the more input they have
- E.g., sorting 5 elements is usually faster than sorting 5 million

**“Worst Case” Analysis**

- Certain input patterns (for all $n$) that take the most amount of time
- E.g., some sorting algorithms take longer if input is in reverse order
- Can be many worst cases for an algorithm
- Can be algorithms where all the cases are the same (no worst case)
- Big-O notation is usually related to the worst cases (depending on context)

**“Best Case” Analysis**

- Certain input patterns (for all $n$) that take the least amount of time
- E.g., some sorting algorithms take less time if input is already sorted
- Can be many best cases, or none (all cases the same)
“Average Case” Analysis

- The average amount of time over all inputs (for all \( n \))
- Often based on “typical” input frequency distributions

One notion of efficiency analysis: **performance testing**

- run an implementation and time it (for different input sizes)
- depends heavily on specific implementation, language, machine, etc

Another notion of efficiency analysis: **count primitive operations**

- Primitive steps (e.g., assembly instructions) as a surrogate for time
- We assume each primitive step takes roughly same amount of (unit) time
- We want to know how many primitive steps needed relative to input size