Dynamic Arrays

A data sequence that contains similar values

Motivation

- Suppose you have a variable data set from that ranges anywhere from 10,000 - 10, 000, 000 numbers and requires random access
  - Using a Standard array
    - *pros?*
    - *cons?*

Dynamic Array

- A dynamic array grows in size as needed using a predefined algorithm
  - Functions just like a standard array, but handles growth automatically
- Pros:
  - Allocates memory as needed
  - Allows dynamically sized arrays
- Cons:
  - Memory intensive
  - Still wastes space

Dynamic Arrays in C

- Called ‘Vectors’ in C/C++
  - ArrayList in Java
- An element can be accessed, inserted, or removed by specifying its index
- Operations
  - insert - insert at an element index
  - read - read at an element index
  - delete - delete from an element index
  - size - return the current number of values in the list

Vector Implementation

```
typedef struct Vector{
    int max_size; //initialize to 0
    int current_size; //initialize to 0
    Data * list; //initialize to NULL
    void (*insert)(struct * Vector, Data d, int index);
    void (*remove)(struct * Vector, int index);
    Data * d (*read)(struct * Vector, int index);
} Vector;
```

How to grow

- How should we grow our dynamic array?
  - 2 dynamic methods
    - *incremental expansion - add 1 for each*
      - Only add exactly what is needed to fit all elements
    - *geometric expansion - double size when max*
      - Expand by some multiplier leaving free, empty space

Vector insert

```
insert(vector, index, value){
        if(index>=vector.max_size)
                vector.max_size = index * 2; //geometric expansion
                new_list = array[vector.max_size]
                copy vector.list -> new_list
                delete vector.list
                vector.list = new list
        if(index >= vector.current_size)
                        vector.current_size = index
        vector.list[index] = value
}
```

Classwork:
Deleting From a Vector

Vector Delete

- Two ways to handle deletion
  - deleteVector(vector, index){
    if(index < vector.max_size)
    vector.list[index] = EMPTY
    }
    - *Problems?*
      - Iteration
- Actual Deletion
  - Expensive and must reallocate space
    - *Solves iteration problem, or does it?*

Vector Read

- Vector read is a simple function that just requires you to check boundary conditions
  - readVector(vector, index){
    if(index >= vector.max_size)
    return FAIL
    else
    return vector.list[index]
    }
- How does your deletion strategy affect your read function?

When to use a Vector

- You know a range of elements needed
- You need random access
- memory isn’t a **primary** concern