List is an Abstract Data Type

A list contains a sequential* collection of values.

We denote the contents of a list as items, entries, or elements.

There are many different kinds of lists, but in general, we may expect a list to support operations such as:

- **Add** - add an item to the end
- **Insert** - add an item between two existing elements
- **Get** - get the value of an item at the specified index
- **Erase** - remove an item from the list
- **Size** - obtain the number of elements in the list

*Note: sequential ≠ sorted!

Array List (aka Vector)

Consider a list data structure built on arrays:

We will be using dynamically allocated arrays (for reasons which will become clear later)

Array List Operations: size

**size:**

- Returns # of elements in list
- Array size ≠ list size
  - Array size is the capacity of the list
  - Need a separate variable to track size

Array List Operations: Simple add

```Java
arr[size] = val;
size++;
```

Questions:

- What happens if we forget to increment size?
- How are size and capacity related?
- What happens when we run out of room?
Encapsulating Array List

We need to:
• Keep array, size, capacity all together
• Maintain consistent state

Encapsulation helps us by:
• Keeping data together with functions on data
• Hiding implementation details from user

The primary enabler of encapsulation is the class.

A Simple Array List Class

class array_list {
public:
array_list();
int size();
int get(int index);
void set(int index, int val);
void add(int val);
void insert(int index, int val);
int& operator[](int index);
private:
int* _arr;
int _size;
int _capacity;
};

Array List: constructor

Need to setup initial storage, size, capacity:

array_list::array_list() {
    _capacity = 1;   // or whatever
    _size = 0;
    _arr = new int[_capacity];
}

Array List Operations: add

add:
• Add item to end of array, increment size
• What happens when size == capacity?

Expanding Capacity

Steps:
1. Double* our capacity variable
2. Create a new array using the new capacity
3. Copy everything from old array to new array
4. Delete old array
5. Update the array pointer to point to the new array

Array List Operations: add

void array_list::add(int val) {
if (_size == _capacity) {
    _capacity = _capacity * 2;
    int* newarr = new int[_capacity];
    for (int j = 0; j < _size; j++)
        newarr[j] = _arr[j];
    delete[] _arr;
    _arr = newarr;
}
    _arr[_size] = val;
    _size++;
}
Array List Operations: insert

insert:
- Move elements to right
- Put element in place in array

```
void array_list::insert(int index, int val) {
    if (_size == _capacity) {
        _capacity = _capacity * 2;
        int* newarr = new int[_capacity];
        if (int j = 0; j < _size; j++)
            newarr[j] = _arr[j];
        delete[] _arr;
        _arr = newarr;
    }
    for (int j = _size; j > index; j--)
        _arr[j] = _arr[j - 1];
    _arr[index] = val;
    _size++;
}
```

Array List Operations: erase

erase:
- Move elements to left, overwriting erased element

```
void array_list::erase() {
    for (int j = _size; j > index; j--)
        _arr[j] = _arr[j - 1];
    _arr[index] = val;
    _size++;
}
```
Array List Operations: erase

```cpp
void array_list::erase(int index) {
    for (int j = index; j < _size - 1; j++)
        _arr[j] = _arr[j + 1];
    _size--;
}
```

Array List Operations: inlines

```cpp
class array_list {
public:
    array_list();
    int size() { return _size; }
    int get(int index) { return _arr[index]; }
    int set(int index, int val) { _arr[index] = val; }
    void add(int val);
    void insert(int index, int val);
    void erase(int index);
    int& operator[](int index) { return _arr[index]; }

private:
    int* _arr;
    int _size;
    int _capacity;
    void _resize();
};
```

Up Next

- **Friday, October 19**
  - Lab 8 – Ancient Algorithms
  - Project 2 - Mazes Due
  - APT 3 Assigned
- **Monday, October 22**
  - Operator Overloading
  - Reading: Chapter 13
- **Wednesday, October 24**
  - The “Big 3” (and continuing with ArrayList)