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!

Linked List Review

- We've studied one kind of list: linked lists
- Performance characteristics:
  - Add to end: O(1)
  - Insert: O(1), at head, or tail, or if we have a pointer to the location
  - Erase: O(1), at head or if we have a pointer to the location
  - Get (at some index): O(n)
- Today we examine a list type built on arrays (essentially equivalent to a vector)

Array List (aka Vector)

Consider a list data structure built on arrays:

- These elements are stored in the list
- These indices store "nothing" (there's really always something there, but we don't care about it)

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

Array List Size vs Capacity

- Array size ≠ list size
  - Array size is the capacity of the list
  - Need a separate variable to track size – the number of elements stored in the list

Array List Operations: add

- Add item to end of array
- Increment size

- Last element: size = 4
- Added element goes here

- List element: size = 4
- Added element goes here
Array List Operations: Simple add

array_list
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?

We’ll answer these in a just a bit...

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);
    void erase(int index);
    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

*Doubling results in a nice complexity analysis using amortized analysis, a technique you will learn later.
Array List Operations: add
void array_list::add(int val) {
    if (_size == _capacity) {
        _capacity = _capacity * 2; // Step 1
        int* newarr = new int[_capacity]; // Step 2
        for (int j = 0; j < _size; j++) // Step 3
            newarr[j] = _arr[j]; // Step 4
        _arr = newarr; // Step 5
    }
    _arr[_size] = val; // Step 6
    _size++; // Step 7
}

Array List Operations: insert
void array_list::insert(int index, int val) {
    if (_size == _capacity) {
        _capacity = _capacity * 2; // Again, deal with case when out of space
        int* newarr = new int[_capacity]; // Again, deal with case when out of space
        for (int j = 0; j < _size; j++) // Again, deal with case when out of space
            newarr[j] = _arr[j]; // Again, deal with case when out of space
        _arr = newarr; // Again, deal with case when out of space
    }
    for (int j = _size; j > index; j--) // Here's the basic operation
        _arr[j] = _arr[j - 1];
    _arr[index] = val; // Here's the basic operation
    _size++; // Here's the basic operation
}

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

void array_list::add(int val) {
    _resize();
    _arr[_size] = val;
    _size++;
}

Array List Operations Refactored (con't)
void array_list::insert(int index, int val) {
    _resize();
    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

```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

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();
};

Array List Analysis

- Add to end
  - O(1) in best case (when don’t expand)
  - O(n) in worst case (expand)
- If we double when we expand, get O(1) averaged over many operations (amortized analysis, a topic for 406)
- Insert: O(n) in worst case
- Erase: O(n) in worst case
- Get/set: O(1)

Efficiency: Array vs Linked

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>O(1)*</td>
<td>O(1)*</td>
</tr>
<tr>
<td>Insert</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Erase</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Indexed Get/Set</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Append</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Think about other operations you might use!</td>
</tr>
</tbody>
</table>

*Amortized
†With tail pointer
‡At head or with pointer at location

Stuff We Didn’t Do

- Bounds checking / error handling
  - What if index for insert/erase is out of bounds?
  - Error handling is tricky...
- The “Big 3” – next week
  - Copying, assigning, and destroying array_list objects
- Templates – storing types other than int – after Spring Break
- Iterators
Up Next

- Read Chapter 13.1 for Wednesday
- Wednesday, March 20
  - Operator overloading
  - Reading: Chapter 13.2
- Friday, March 22
  - Lab 10 – Operator overloading
  - APT 4 due
  - Project 4 assigned (will be due April 5)