CSCI 262
Data Structures

15 – Array List

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

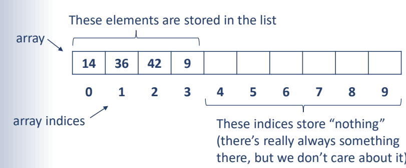
*Note: sequential ≠ sorted!

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)

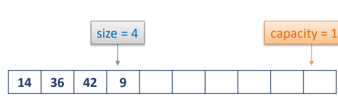
Consider a list data structure built on arrays:



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

Array List Size vs Capacity

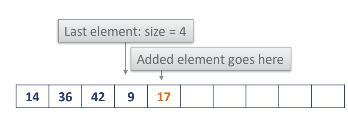
- Array size ≠ list size
  - Array size is the *capacity* of the list
  - Need a separate variable to track *size* – the number of elements *stored* in the list



Array List Operations: add

add:

- Add item to end of array
- Increment size



Array List Operations: Simple add

```
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;
};
```

Declarations of member functions implementing the major operations (and some extra), plus a constructor.

Private variables for our array, and two other bits of state: the array capacity and the list size.

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?

Last element: size = 4

Added element goes here

14	36	42	9	17					
0	1	2	3	4	5	6	7	8	9

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]; ------- 2
                for (int j = 0; j < _size; j++)   --------- 3
                        newarr[j] = _arr[j];      -------------
                delete[] _arr; ------------------------------------- 4
                _arr = newarr; ------------------------------------- 5
        }

        _arr[_size] = val;
        _size++;
}
```

CS@Mines

Array List Operations: insert

insert:

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

Insert element here

These elements had to move right one place

14	36	42	7	9	17				
0	1	2	3	4	5	6	7	8	9

CS@Mines

Array List Operations: insert

```
void array_list::insert(int index, 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;
        }

        for (int j = _size; j > index; j--)
                _arr[j] = _arr[j - 1];
        _arr[index] = val;
        _size++;
}
```

Again, deal with case when out of space

Here's the basic operation

CS@Mines

Array List Operations: insert

```
void array_list::insert(int index, 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;
        }

        for (int j = _size; j > index; j--)
                _arr[j] = _arr[j - 1];
        _arr[index] = val;
        _size++;
}
```

Hm. This looks familiar...

CS@Mines

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];
                delete[] _arr;
                _arr = newarr;
        }
}

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

Private method, capturing common behavior.

CS@Mines

Array List Operations Refactored (con't)

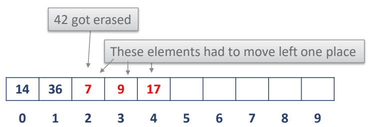
```
void array_list::insert(int index, int val) {
        _resize();

        for (int j = _size; j > index; j--)
                _arr[j] = _arr[j - 1];
        _arr[index] = val;
        _size++;
}
```

CS@Mines

Array List Operations: erase

erase:

- Move elements to left, overwriting erased element



CS@Mines

Array List Operations: erase

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

CS@Mines

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();
};
```

CS@Mines

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)

CS@Mines

Efficiency: Array vs Linked

	Array	Linked
Add	O(1)*	O(1)†
Insert	O(N)	O(1)‡
Erase	O(N)	O(1) ‡
Indexed Get/Set	O(1)	O(N)
Append	O(N)	O(1)
...	Think about other operations you might use!	

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

CS@Mines

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

CS@Mines

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)

CS@Mines