

CSCI 262 Data Structures

10 – Array List

CS@Mines

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:

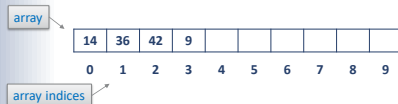
- Add - add an item to the end
- Insert - add an item between two existing elements
- Get - get the value of an item at the specified index
- Erase - remove an item from the list
- Size - obtain the number of elements in the list

*Note: sequential ≠ sorted!

CS@Mines

Array List (aka Vector)

Consider a list data structure built on arrays:

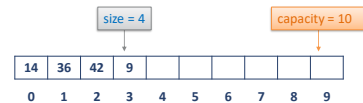


CS@Mines

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*

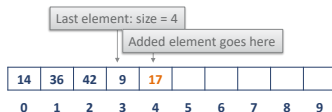


CS@Mines

Array List Operations: add

add:

- Add item to end of array
- Increment size



CS@Mines

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?

CS@Mines

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*.

CS@Mines

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;
};
```

CS@Mines

Array List: constructor

Need to setup initial storage, size, capacity:

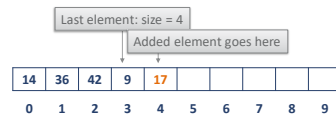
```
array_list::array_list() {
    _capacity = 1;    // or whatever
    _size = 0;
    _arr = new int[_capacity];
}
```

CS@Mines

Array List Operations: add

add:

- Add item to end of array, increment size
- What happens when **size == capacity**?



CS@Mines

Array List Operations: add

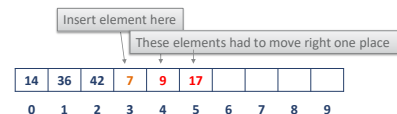
```
void array_list::add(int val) {
    if (_size == _capacity) {
        _capacity = _capacity * 2;
        int* newarr = new int[_capacity];
        for (int j = 0; j < _size; j++)
            newarr[j] = _arr[j];
        delete[] _arr;
        _arr = newarr;
    }
    _arr[_size] = val;
    _size++;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

CS@Mines

Array List Operations: insert

insert:

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



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++;
}
```

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++;
}
```

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++;
}
```

CS@Mines

Array List Operations: refactored (con't)

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

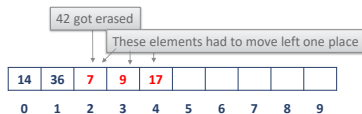
    for (int j = _size; j > index; j--)
        _arr[j] = _arr[j - 1];
    _arr[index] = val;
    _size++;
}
```

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

Up Next

- Wednesday, February 28
 - Operator Overloading
 - Reading: Chapter 8.16
- Friday, March 2
 - Lab 7 – Operator Overloading
 - Project 3 due
 - APT 4 assigned
- Monday, March 5
 - The “Big 3” (and continuing with ArrayList)
 - Reading: Chapters 12.8 – 12.11

CS@Mines