

Today's Material

- List
 - Definition & Operations
- List Abstract Data Type (ADT)
 - Definition & Operations
- List Implementation: ArrayList

List - Definition

- What is a list?
 - An **ordered** sequence of elements A_1, A_2, \dots, A_N
 - Elements may be of arbitrary, but the same type (i.e., all ints, all doubles etc.)
- Example list with 4 elements indexed 0..4
 - 2, 6, 1, 2, 3
 - Index of 2 is 0
 - Index of 6 is 1
 - Index of 1 is 2
 - Index of 2 is 3
 - Index of 3 is 4

List Operations: add

- Add a new element to the end of the list
 - 4, 6, 1, 4, 5 → **add(8)** → 4, 6, 1, 4, 5, **8**
- Add a new element at an arbitrary position
 - 4, 6, 1, 4, 5 → **add(2, 9)** → 4, 6, **9**, 1, 4, 5

List Operations: remove

- Remove an existing element from the list
 - 4, 6, 1, 4, 5 → **remove(2)** → 4, 6, 4, 5
 - 4, 6, 1, 4, 5 → **remove(0)** → 6, 1, 4, 5
 - 4, 6, 1, 4, 5 → **remove(1)** → 4, 1, 4, 5
 - 4, 6, 1, 4, 5 → **remove(4)** → 4, 6, 1, 4

List Operations:

indexOf & lastIndexOf

- **indexOf** returns the index of the **first** matching element
 - 4, 6, 1, 4, 5 → **indexOf(6)** → 1
 - 4, 6, 1, 4, 5 → **indexOf(5)** → 4
 - 4, 6, 1, 4, 5 → **indexOf(4)** → 0
- **lastIndexOf** returns the index of the **last** matching element
 - 4, 6, 1, 4, 5 → **lastIndexOf(1)** → 2
 - 4, 6, 1, 4, 5 → **lastIndexOf(4)** → 3

List Operations: *get* & *set*

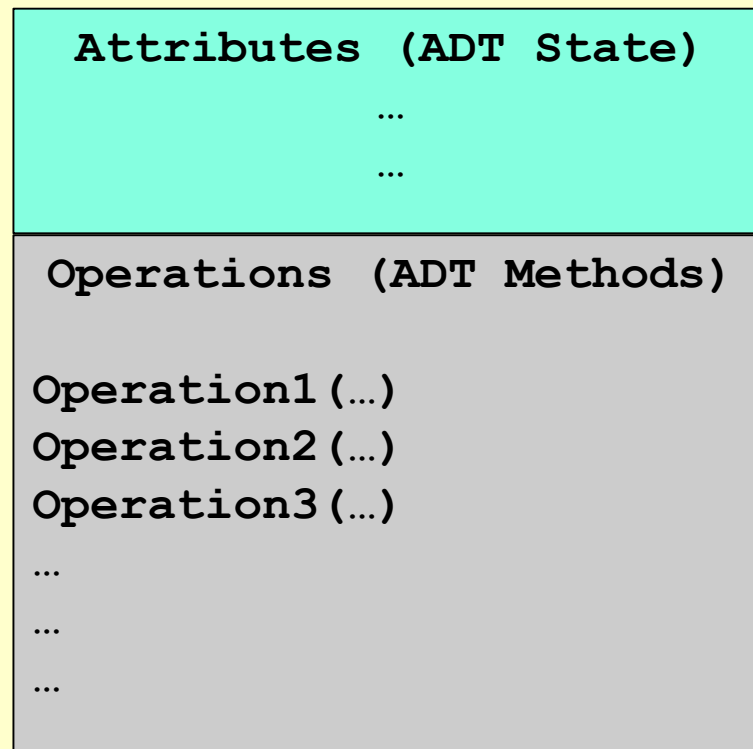
- *get* returns the element at a particular index
 - 4, 6, 1, 4, 5 → *get*(1) → 6
 - 4, 6, 1, 4, 5 → *get*(3) → 4
 - 4, 6, 1, 4, 5 → *get*(0) → 4
- *set* changes the value of the element at a particular index
 - 4, 6, 1, 4, 5 → *set*(1, 9) → 4, 9, 1, 4, 5
 - 4, 6, 1, 4, 5 → *set*(3, 5) → 4, 9, 1, 5, 5
 - 4, 6, 1, 4, 5 → *set*(4, 7) → 4, 9, 1, 4, 7

Abstract Data Type (ADT)

- So far we have defined an Abstract Data Type (ADT) with a set of operations
 - Operations specify how the ADT behaves, but does not reveal how they are implemented
 - In many cases, there are more than one way to implement an ADT
- In this course we will cover lots of ADTs
 - Lists
 - Stacks
 - Queues
 - Trees & Search Trees
 - Hash Tables & Tries

Abstract Data Type (ADT) - more

- An Abstract Data Type (ADT) is a **data structure** with a set of **operations**



List ADT

```
public class List {
```

```
...
```

```
public:
```

```
    void add(int e);           // Add to the end (append)
    void add(int pos, int e);  // Add at a specific position
    void remove(int pos);      // Remove
    int indexOf(int e);        // Forward Search
    int lastIndexOf(int e);    // Backward Search
    bool clear();              // Remove all elements
    bool isEmpty();            // Is the list empty?
    int first();                // First element
    int last();                 // Last element
    int get(int pos);          // Get at a specific position
    int size();                 // # of elements in the list
};
```

Using List ADT

```
public static void main(String args[]){
    // Create an empty list object
    List list = new List();

    list.add(10);      // 10
    list.add(5);       // 10, 5
    list.add(1, 7);    // 10, 7, 5
    list.add(2, 9);    // 10, 7, 9, 5

    list.indexOf(7);   // Returns 1
    list.get(3);       // Return 5
    list.remove(1);    // 10, 9, 5
    list.size();       // Returns 3
    list.isEmpty();    // Returns false

    list.remove(0);    // 9, 5
    list.clear();      // empty list

}/* end-main */
```

Lists: Implementation

- Two types of implementation:
 - Array-Based - Today
 - Linked List - Next Week
- We will compare worst case running time of ADT operations with different implementations

Lists: Array-Based Implementation

- Basic Idea:
 - Pre-allocate a big array of size MAX_SIZE
 - Keep track of first free slot using a variable N
 - Empty list has $N = 0$
 - **Shift elements** when you have to **insert or delete**

0	1	2	3	N-1			MAX_SIZE
A_1	A_2	A_3	A_4	A_N-1			

Array List ADT - Java Declarations

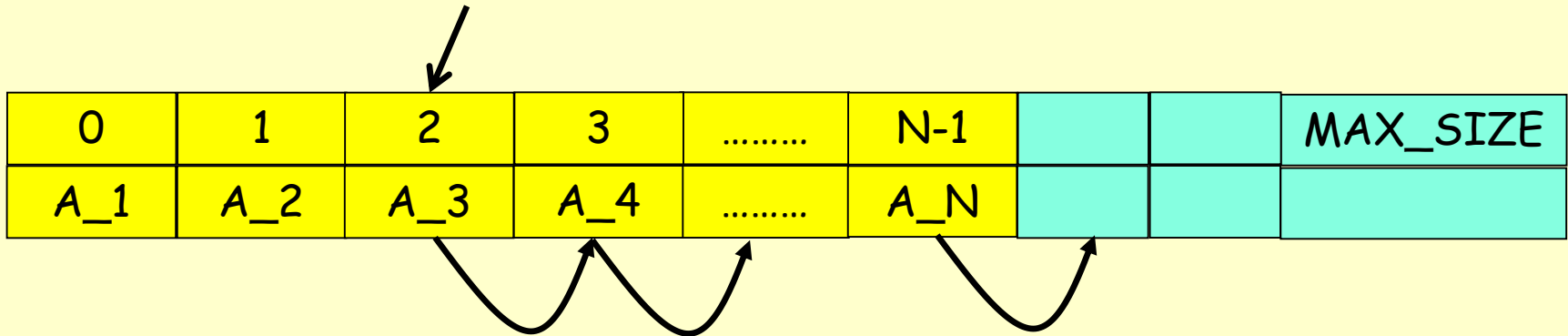
ArrayList ADT

```
public class ArrayList {  
private:  
    int capacity;  
    int noOfItems;  
    int items[];  
  
public:  
    ArrayList (); // Constructor  
  
    void add(int pos, int e);  
    void remove(int pos);  
    int indexOf(int e);  
    bool isEmpty();  
    int first();  
    int last();  
    int get(int pos);  
    int size();  
};
```

```
// Constructor:  
// Make empty list  
ArrayList() {  
    items = new int[10];  
    capacity = 10;  
    noOfItems = 0;  
} // end-ArrayList
```

Lists Operations: add

- **add**(Position P, ElementType E)
 - Example: **add(2, X)**: Insert X at position 2
- **Basic Idea**: Shift existing elements to the right by one slot and insert new element



- Here is the final list. **Running time: $O(N)$**

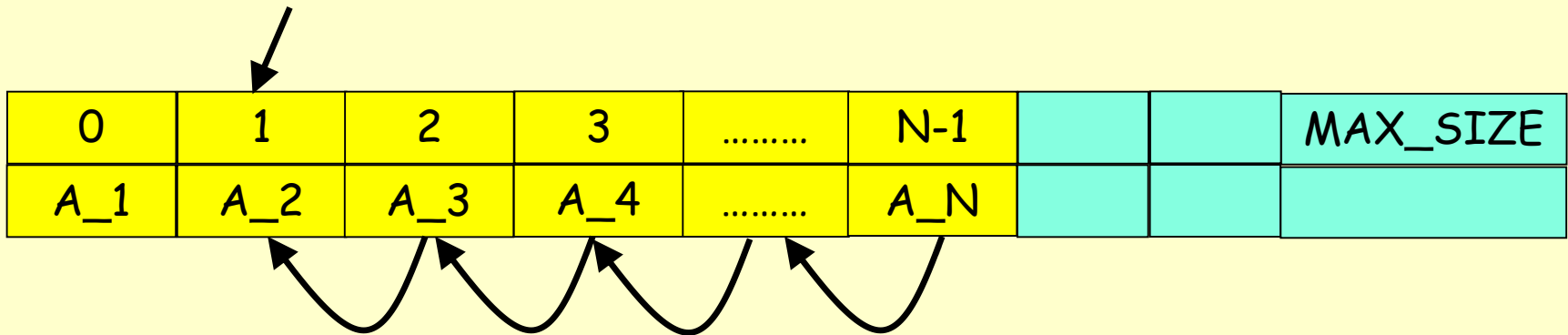
0	1	2	3	N-1	N		MAX_SIZE
A_1	A_2	X	A_3	A_N-1	A_N		

Lists Insert - Full Array

- What if the array is already full?
 - Can return an error
 - Not preferred
 - Typically though, you would do the following:
 - (1) Allocate a bigger array (usually double the capacity)
 - (2) Copy all elements from the old array to the new one
 - (3) Free up the space used by the old array
 - (4) Start using the new array
 - With this dynamic array implementation, you would also need to keep track of the capacity of the array

Lists Operations: remove

- **remove**(Position P)
 - Example: **remove(1)**: Delete element at position 1
- **Basic Idea**: Remove element and shift existing elements to the left by one



- Here is the final list. **Running time: $O(N)$**

0	1	2	3	N-2	N-1		MAX_SIZE
A_1	A_3	A_4	A_5	A_N			

Lists Operations: indexOf

- **indexOf**(ElementType E)
 - Example: **indexOf(X)**: Search X in the list

0	1	2	3	N-1			MAX_SIZE
A_1	A_2	A_3	A_4	A_N			

- Must do a linear search
 - Running time: $O(N)$

Lists Operations: isEmpty

- **isEmpty()**
 - Returns true if the list is empty

0	1	2	3	N-1			MAX_SIZE
A_1	A_2	A_3	A_4	A_N			

- Trivial - Return true if $N == 0$
 - Running time: $O(1)$

Lists Operations: first, last, get

- **first()**: Return $A[0]$
- **last()**: Return $A[N-1]$
- **get(Position K)**: Return $A[K]$

0	1	2	3	N-1			MAX_SIZE
A_1	A_2	A_3	A_4	A_N			

- **first** - Running time: $O(1)$
- **last** - Running time: $O(1)$
- **get** - Running time: $O(1)$

Application of Lists

- Polynomial ADT: store and manipulate single variable polynomials with non-negative exponents
 - $10X^3 + 4X^2 + 7 = 10X^3 + 4X^2 + 0X^1 + 7X^0$
 - Data structure: stores coefficients C_i and exponents i
- Array Implementation: $C[i] = C_i$
 - E.g. $C[3] = 10, C[2] = 4, C[1] = 0, C[0] = 7$
- ADT operations: Input polynomials in arrays A and B , result in C
 - Addition: $C[i] = ?$
 - Multiplication: ?

Application of Lists: Polynomials

- Add(Poly A, Poly B, Poly C):
 - $C[i] = A[i] + B[i]$ for $0 \leq i \leq N$
- Multiply(Poly A, Poly B, Poly C):
 - Set $C[i] = 0$ for $0 \leq i \leq N$
 - For each pair (i, j)
 - $C[i+j] = C[i+j] + A[i] * B[j]$
- Divide(Poly A, Poly B, Poly C):
 - You do it...

Application of Lists: Polynomials

- Problem with Array Implementation: **Sparse Polynomials**
 - E.g. $10X^{3000} + 4X^2 + 7$
 - Waste of space and time (C_i are mostly 0s)
- Solution?
 - Use singly linked list, sorted in decreasing order of exponents

