

Intro to Data Structures & Algorithms

Wagner Truppel
Lecturer, Dept. of Computer Science & Engineering
UC Riverside

wagner@cs.ucr.edu
<http://www.cs.ucr.edu/~wagner>

<http://www.cs.ucr.edu/cs14>

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 1

Today's Topics

- ADT Counter review
- ADT Array
- ADT List
- Implementing the ADT List
 - ◆ Array implementation
 - ◆ Pointer implementation

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 2

ADT Counter review

```
adt CounterADT // "filename"
uses Integer // other adt's used
defines Counter // name of type defined
operations
  get: Counter ---> Int // "observer"
  inc: Counter ---> Counter // "modifier"
  new: ---> Counter // "constructor"
preconditions // none for this adt
axioms // "contract"
  get( new() ) = 0 // new starts at 0
  get( inc(c) ) = get(c) + 1 // inc's by 1
```

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 3

ADT Array

- What's an array really like ? What do we need to represent one as an ADT?
- It's a *linear* collection of elements of some type, so we need a **type**. Call it **T**.
- We need to be able to refer to each element, so we need some kind of index... an **integer** will do just fine.
- We need to be able to *retrieve* and *set* the values stored in the array, so we need **get** and **put** functions
- And if we're going to search through the array to find some element, we need to know when to stop... we need an array **length**

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 4

ADT Array

```

adt Array
uses Integer, Comparable
defines Array<T: Comparable>
    // Comparable supports ==
operations
new: Integer x T -/-> Array<T>
get: Array<T> x Integer -/-> T
put: Array<T> x Integer x T -/-> Array<T>
length: Array<T> ---> Integer

"-/->" indicates a function with
preconditions
failure of preconditions should
generate exceptions
    
```

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 5

ADT Array

```

preconditions
new(l, t): l > 0
get(a, i): 0 <= i < length(a)
put(a, i, t): 0 <= i < length(a)

axioms
length( new(l, t) ) = l
length( put(a, i, t) ) = length(a)
get(new(l,t), i) = t
get(put(a, i, t), j)
    = t           if i == j,
    = get(a, j)  if i != j
    
```

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 6

ADT Array

- With a complete ADT for arrays we can now
 - ◆ implement arrays any way we like
 - ◆ make sure that our implementations do satisfy the ADT Array interface
 - ◆ make sure that our implementations do satisfy the array axioms

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 7

ADT List

- What's a list? What do we need to represent one as an ADT?
- It's a *linear* collection of elements of some type, so we need a **type**. Call it **T**.
- We need to be able to refer to each element, *but it need not be an integer*.
 - ◆ We call it an **iterator**. (more later)
- We need to be able to *retrieve* and *set* the values stored in the list, so we need **get** and **put** functions
- And if we're going to search through the list to find some element, we need to know when to stop.
 - ◆ The **iterator** will also take care of that.

■ Looks a lot like the ADT Array, doesn't it?

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 8

ADT List

- The ADT List is similar to the ADT Array, but is more general
 - ◆ It does not impose *integer* indices
 - ◆ It defines more operations
- ADT List operations
 - ◆ Create a list
 - ◆ Destroy a list
 - ◆ Is the list empty
 - ◆ How many items?
 - ◆ Insert an item at a given position
 - ◆ Delete an item from the list
 - ◆ Retrieve an item from the list

■ Its formal definition is rather complex so we'll skip it!

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 9

ADT List: Array implementation

- Array characteristics
 - ◆ Very convenient - an *almost* built-in DS
 - ◆ Easy and fast **retrieval** of elements
 - ◆ But cannot be resized once defined or, if we want to resize them, we need to create a new array and copy all elements from the old to the new array
 - ★ This creates additional problems.
 - Ex: could run out of memory in the process of duplicating the old array
 - ◆ Slow **insertion** and **deletion** (more next)

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 10

ADT List: Array implementation

```

ArrayBasedList
typedef char itemType;
- const int MAX_SIZE = 100
- itemType items[MAX_SIZE];
- int size;
// functions
            
```

- Let's use **char** as an example
- We can create a class **ArrayBasedList**, where the array holds **chars**
- After we're done, we can template it
- The class would have these (private) member variables:
 - ◆ typedef char itemType;
 - ◆ const int MAX_SIZE = 100;
 - ◆ itemType items[MAX_SIZE];
 - ◆ int size;

size

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|--------|-------|---|---|---|-----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | size-1 | max-1 | | | | | |
| T | C | Z | L | S | U | Z | C | V | A | ... | B | ? | ? | ? | ... | ? |

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 11

ADT List: Array implementation

- Operations (member functions of the class):
 - ◆ **Creation**: in the constructor for the class, allocate the array and set size to 0. Note that every item gets "zeroed out". **O(1)** or **O(n)** depending on item type.
 - ◆ **Destruction**: in the destructor of the class, de-allocate the array (using delete); may need to de-allocate items too; no need to set size to 0, but good practice anyway. **O(1)** or **O(n)** depending on item type.
 - ◆ bool isEmpty() { return size == 0; } **O(1)**
 - ◆ int getLength() { return size; } **O(1)**

size

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|--------|-------|---|---|---|-----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | size-1 | max-1 | | | | | |
| T | C | Z | L | S | U | Z | C | V | A | ... | B | ? | ? | ? | ... | ? |

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 12

ADT List: Node class/struct

- Singly-linked list

| Node |
|-----------------------------|
| - int stuff |
| - Node * next |
| + Node(int data, Node * nn) |
| + int getStuff() |
| + void setStuff(int data) |
| + Node * getNext() |
| + void setNext(Node * nn) |

```

class Node
{
private:
    int stuff; // for now, let's make it an int
    Node * next; // why a pointer and
                // not the obj itself?

public:
    Node(int data, Node * nn);
    // other member functions
};

typedef Node * NodePtr; // pointer to a Node
                    
```

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 16

ADT List: Node class/struct

- Doubly-linked list

| Node |
|--|
| - int stuff |
| - Node * prev |
| - Node * next |
| + Node(int data, Node * nn, Node * pn) |
| + int getStuff() |
| + void setStuff(int data) |
| + Node * getNext() |
| + void setNext(Node * nn) |
| + Node * getPrevious() |
| + void setPrevious(Node * pn) |

```

class Node
{
private:
    int stuff; // for now, let's make it an int
    Node * prev;
    Node * next;

public:
    Node(int data, Node * nn, Node * pn);
    // other member functions
};

typedef Node * NodePtr; // pointer to a Node
                    
```

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 17

ADT List: Node class/struct

- Singly-linked or doubly-linked, either way the **header** and the **tail** are *pointers* to the first and last **Nodes** - they're **not** **Nodes** themselves

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 18

ADT List: Pointer implementation

```

SinglyLinkedList
class Node
typedef * Node NodePtr;
- NodePtr head;
- NodePtr tail;
- int size;
// functions
        
```

- Let's stick to singly-linked lists for now
- We can create a class **SinglyLinkedList** to hold, say, integers. We can template it later.
- The class would have these (private) member variables:
 - ◆ Node * head;
 - ◆ Node * tail; // if desired or needed
 - ◆ int size; // if desired or needed
- The class **Node** might also be internal to the **SinglyLinkedList** class

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 19

ADT List: Pointer implementation

```

SinglyLinkedList
class Node
typedef * Node NodePtr;
- NodePtr head;
- NodePtr tail;
- int size;
// functions
        
```

- Operations (member functions of the class):
 - ◆ **Creation**: in the constructor for the class, allocate the head and tail pointers, set them to NULL, and set size to 0. **O(1)**
 - ◆ **Destruction**: in the destructor of the class, must first de-allocate each Node (using delete); no need to set size to 0, but good practice anyway. **O(n)**
 - ◆ bool isEmpty() { return head == NULL; } **O(1)**
or
 - ◆ bool isEmpty() { return size == 0; } **O(1)**
 - ◆ int getLength() { return size; } **O(1)**
 - ◆ getLength() would be **O(n)** if we didn't maintain explicit size information using the integer *size*

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 20

ADT List: Pointer implementation

```

SinglyLinkedList
class Node
typedef * Node NodePtr;
- NodePtr head;
- NodePtr tail;
- int size;
// functions
        
```

- Operations (member functions of the class):
 - ◆ **3 kinds of insertions**: (need to test preconditions!)
 - * insertFirst(int k) **O(1)**
 - * insertLast(int k) **O(1)** with tail, **O(n)**, with no tail
 - * insertAt(int k, int index) **O(n)**
 - ◆ **3 kinds of deletions**: (need to test preconditions!)
 - * deleteFirst() **O(1)**
 - * deleteLast() **O(n)** for singly with or without tail, **O(n)** for doubly with no tail, **O(1)** for doubly with tail
 - * deleteAt(int index) **O(n)**
 - ◆ **3 kinds of retrievals**: (need to test preconditions!)
 - * char getFirst() **O(1)**
 - * char getLast() **O(n)** without tail, **O(1)** with tail
 - * char get(int index) **O(n)**

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 21

ADT List: Comparing Implementations

| | Array | Singly-L | Doubly-L |
|-----------------|---------------------------------------|---------------------------------------|---------------------------------------|
| Creation | O(1) if built-in types O(n) if not | O(1) | O(1) |
| Destruction | Same as above | O(n) | O(n) |
| isEmpty() | O(1) | O(1) | O(1) |
| getLength() | O(1) | O(1) with <i>size</i> O(n) without | O(1) with <i>size</i> O(n) without |
| insertFirst() | O(n) | O(1) | O(1) |
| insertLast() | O(1) | O(1) with <i>tail</i> O(n) without | O(1) with <i>tail</i> O(n) without |
| insertAtIndex() | O(n) | O(n) | O(n) |

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 22

ADT List: Comparing Implementations

| | Array | Singly-L | Doubly-L |
|-----------------|-------|---------------------------------------|---------------------------------------|
| deleteFirst() | O(n) | O(1) | O(1) |
| deleteLast() | O(1) | O(n) even with <i>tail</i> | O(1) with <i>tail</i> O(n) without |
| deleteAtIndex() | O(n) | O(n) | O(n) |
| getFirst() | O(1) | O(1) | O(1) |
| getLast() | O(1) | O(1) with <i>tail</i> O(n) without | O(1) with <i>tail</i> O(n) without |
| getAtIndex() | O(1) | O(n) | O(n) |

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 23

ADT List: Comparing Implementations

- Occasionally, it's useful to define a few more operation
- These are useful when you need to do lots of insertions/deletions in sequence, in which case you should use a doubly-linked list

| | Array | Singly-L | Doubly-L |
|----------------|-------|----------|----------|
| insertBefore() | O(n) | O(n) | O(1) |
| insertAfter() | O(n) | O(1) | O(1) |
| deleteBefore() | O(n) | O(n) | O(1) |
| deleteAfter() | O(n) | O(1) | O(1) |
| next() | O(1) | O(1) | O(1) |
| previous() | O(1) | O(n) | O(1) |

©2003 W L Truppel CS 14: Intro. Data Structures & Algs. • Lecture 3 24
