Computer Science 331
Basic Data Structures

Mike Jacobson

Department of Computer Science
University of Calgary

Lectures #9–10

Outline

Objectives for Today

Objectives for Today:

- Review of several basic data structures, including types of *arrays* and *linked lists*
- *Reference:* Text, Chapter 3

Assumption: You have seen most of this already! Some implementation and analysis details may be new.

Suggested Exercises for Later:

- Write specifications of requirements for the various operations being discussed
- Write a few of the algorithms sketched here in more detail
- Sketch proofs of correctness, and analyses of worst-case running times, using techniques from class

Static Array

A data structure providing access to a *fixed* number of data cells of some type

- Attributes:
  - *length* : number of data cells for which access is provided
  - base type: the type of data to be stored in each cell
- Data cells have unique integer *indices* between 0 and $length - 1$
- A data cell can be accessed *at unit cost* by specifying its index
- Many programming languages, including `Java`, directly support this data structure

Example

Suppose A is the following array of `String`'s:

0	1	2	3	4	5
a	c	x	g	h	`null`

- Length of A:
- Base Type of A:
- Current value of $A[3]$:
- Charge to access or store an entry of A at a given index:

Automatic Initialization of an Array

An operation like

```
String[] sArray = new String[25];
```

declares the type of a variable (in this case, `sArray` — setting this to be an array that stores `String`'s) and sets the *length* of the array (in this case, 25)

Initial Value in Each Cell: The *default value* for the base type

- Default Value for Numeric Types:
- Default Value for `char` Type:
- Default Value for `boolean` Type:
- Default Value for Class Types:

Initialization of an Array with Values

Initial values can be enclosed in braces, separated by commas

- `A.length` automatically set to the number of initial values listed

Example: The statement

```
int[] age = { 2, 4, 7, 3, 6, 5 }
```

creates the following array

	0	1	2	3	4	5
age:	2	4	7	3	6	5

Cost To Initialize an Array:

-
-

Traversal of an Array

Visiting some or all of the cells in an array...

- Beginning at some index (usually 0)
- Going in either direction (usually by increasing index)

Since arrays allow direct access, implementing *traversals* is straightforward:

```
for (int i=0; i < A.length; i++) {
  // process array entry A[i]
}
```

Worst-Case Cost for a Traversal:

Application: Finding a Given Value

Strategy (linear search):

- Traverse array from index 0
- Compare each array element with the given value until it is found or all entries have been checked
- Return index if the value is found; throw an exception or return an exceptional value (eg, -1) otherwise

Since at most a constant number of steps are used at each array index, the worst-case cost is:

Replacing an Element of an Array (by position)

Problem: Given an index i and value v, replace contents at position i with v

How To Do This:

Error Conditions:

-
-

Worst-Case Cost:

-
-

Replacing an Element of an Array (by value)

Problem: Given values v and w, replace w with v in the array, or report that v was not found

How To Do This:

- Find index i such that $A[i] = w$ or report that w is not in the array. Cost:
- Set $A[i] = v$. Cost:

Error Conditions: none

Worst-Case Cost:

-

Additional Operations for Storage of Sets

Suppose now that an array is used to store a **set**:

- Elements of a set — and the values in the currently used part of the array — are distinct
- New attribute: *numElements* — size of the set currently stored
- *Requirements:*
  - $numElements \leq length$ and the set's elements are stored at positions $0, 1, \ldots, numElements - 1$
  - Default values for base type are stored at positions $numElements, numElements + 1, \ldots, length - 1$

Insertion of an Element into a Set

Problem: Given a value v, add v to the represented set

Error Conditions:

- $numElements = length$ (array is already full)
- v is already in the set

Situations of Interest:

- Storage order of elements in the array is unimportant *and* the new element is guaranteed *not* to be in the set already
- Storage order of elements in the array is unimportant *but* it is possible that the “new” element is already in the set
- Storage order of elements in the array is important

Insertion of an Element into a Set (Case 1)

Case 1: Storage Order is Unimportant and the New Element is Guaranteed Not To Be in the Set

How To Do This:

- If $numElements = A.length$, report that A is full.
- Otherwise, set $A[numElements] = v$ and increment $numElements$.

Worst-Case Cost:

Insertion of an Element into a Set (Case 2)

Case 2: Storage Order is Unimportant But the Element Might Be in the Set Already

How To Do This:

- If $numElements = A.length$, report that A is full. Cost:
- If there exists an index i such that $A[i] = v$, report that v is already in A. Cost:
- Otherwise, set $A[numElements] = v$ and increment $numElements$. Cost:

Worst-Case Cost:

Insertion of an Element into a Set (Case 3)

Case 3: Insertion if Storage Order *is* Important:

How To Do This:

- If $numElements = A.length$, report that A is full.
- Otherwise, locate the index i where the element should be placed
- “shift” all elements from the insertion location “up” one position in the array and copy the new element into its correct spot.

Worst-Case Cost:

Deletion of an Element from an Set

Problem: Given a value v, remove v from the represented set

Error Conditions: v is not in the array

Deletion if Storage Order is Unimportant:

- Find index i such that $A[i] = v$ or report that v is not in the array.
- Set $A[i] = A[numElements - 1]$; decrement $numElements - 1$.

Worst-Case Cost:

Deletion if Storage Order is Important

- Find index i such that $A[i] = v$ or report that v is not in the array.
- "Shift" all elements at index $i + 1$ to $numElements - 1$ one position "down"; decrement $numElements$.

Worst-Case Cost:

Dynamic Arrays

Lengths of *dynamic arrays* can be changed as needed

Java (and a few other languages) support dynamic arrays

- In Java, a dynamic array is called an `ArrayList`
- Older versions provided a `Vector` instead (still supported, but not recommended).

Reasons To Use a Dynamic Array:

- it may be difficult to derive a rigorous upper bound on the number of elements that will be stored in the array,
- extra memory is not available (or expensive), so allocating a large static array with an excessive number of unused entries is not feasible.

Initialization of a Dynamic Array

An operation like

```
ArrayList<String> SDArray =
                    new ArrayList<String>();
```

declares the name of a dynamic array (in this case, `SDArray`), and sets the base type of the dynamic array (in this case, `String`). Similarly,

```
ArrayList<Integer> SDIntegers =
                new ArrayList<Integer>();
```

creates a new dynamic array with base type `Integer`.

Note: `ArrayLists` must store `Object`'s instead of primitive data types, so we must "wrapper classes" for primitive types to define dynamic arrays in Java that contain them.

Accessing and Modifying a Dynamic Array

Every dynamic array in Java is an `Object`, and you must access or modify it by calling one of its methods — using the usual syntax for method calls.

Example: To find the current *size* of a dynamic array `SDIntegers` (that is, its current number of entries), you should call its `size` method: The statement

```
s = SDIntegers.size();
```

would set the value of the variable `s` to be the size of this dynamic array.

Adding an Entry to a Dynamic Array

Java uses static arrays to implement a dynamic array.

Consider an operation that increases the size of a dynamic array; if this corresponds to the use of a single `ArrayList` method then the size increases by one.

Case: The size of the `ArrayList` is less than or equal to the the `length` of the underlying static array, after the operation.

This is case is easy! Carry out the operation in (pretty much) exactly the same way as you would if you were working with a static array.

Adding an Entry to a Dynamic Array

Case: The static array was "full" (that is, completely used) before this operation, so that the size of our `ArrayList` should now be *one more than* the length of the static array that currently represents it.

In this case we must replace the static array currently being used with another static array, with the same base type, and with length `newLength` — where this value is strictly greater than the length of the static array that is being replaced.

The method `System.arraycopy` can be used to do this quickly.

Once this is done (and references to the old static array are replaced with references to the new one) the `ArrayList` operation can proceed as in the first case.

Adding an Entry to a Dynamic Array

Q: What is a good choice for the length, `newLength`, of the new array?

Bad Idea: Set `newLength` to be one more than the length of the old static array (or greater by a small constant)

A Much Better Idea: Increase the length by a constant *factor:* in particular, it is a good idea to set `newLength` to be *twice* as large as the old static array.

Why?

-

Linked Data Structures

Consist of zero or more *nodes* that are allocated as-needed and that are connected via references or pointers

- *Advantage:* Structures can grow as needed, unlike static arrays — and at low cost, unlike dynamic arrays
- *Disadvantage:* Constant-time direct access (by index or position) is not supported
- *Reference:* Sections 3.3 and 3.4 of the text includes an extensive discussion including Java implementations

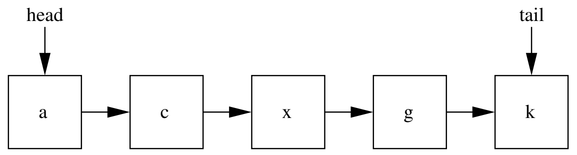
Singly Linked Lists

Brief Description: Nodes are Linearly Connected — each has a *value* and a reference to its *successor* node

Attributes:

- *head*: Reference to the first node in the list

Example:



Optional Attributes:

- *tail*: Reference to the last node in the list (optional)
- *length*: Number of nodes in the list

Initialization of a Linked List

How To Do This:

- Set the head (and tail) to be null
- Set length to be 0.

Worst-Case Cost:

Traversal of a Linked List

How To Do This:

- Initialize a “cursor” to the head node
- While the cursor is not null
  - “Visit” or “process” the node pointed to by the cursor.
  - Set cursor to its successor.

Worst-Case Cost:

Application: Finding a Given Element

Searching by Value:

- *How To Do This:*
  - Traverse the list from the beginning; halt once the value being searched for is found.
- *Worst-Case Cost:*

Searching by Position:

- *How To Do This:*
  - Traverse the list from the beginning; halt once the desired position is reached.
- *Worst-Case Cost:*

Replacing an Element of a Singly Linked List

How To Do This:

- Traverse the list from the beginning; halt once the value to be replaced is found.
- Overwrite the value stored in the current node with the new value.

Worst-Case Cost:

Insertion of an Element into a Set (Case 1)

Case 1: Storage Order is Unimportant and the New Element is Guaranteed Not To Be in Set

How To Do This:

- Create a new node whose value is the element to insert, and whose successor is set to the head node.
- Set the head to the new node.

Worst-Case Cost:

Insertion of an Element (Case 2)

Case 2: Storage Order is Unimportant But the Element Might Be in the Set Already

How To Do This:

- Traverse the entire list to check whether the element is already in the list. Cost: $\Theta(n)$
- If the element is not in the list, insert it at the head. Cost: $\Theta(1)$

Worst-Case Cost:

Insertion of an Element (Case 3)

Case 3: If Storage Order is Important

How To Do This:

- Traverse the list from the beginning to find node (cursor) that should come *before* the new node.
- Set the new node's successor field to the successor field of the cursor.
- Set the cursor's successor field to the new node.

A Complication:

- If the new node goes at the beginning of the list, its successor is the current head, after which head must be changed to the new node

Worst-Case Cost:

Deletion of an Element

How To Do This:

- Traverse the list from the beginning to locate the node to delete (target) *and* its predecessor.
- Set the predecessor's successor node to the target's successor node (thus "unlinking" the node pointed to by target from the list).
- Need the tail's predecessor in addition to the tail itself in this case.

A Complication:

- Deleting the head must be handled separately.

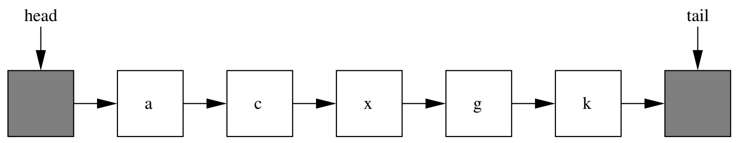
Worst-Case Cost:

Singly Linked Lists with Dummy Nodes

Singly Linked Lists with Dummy Nodes:

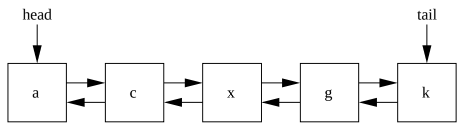
- *Variation:* Nodes at head (and tail) do not store values — they are placeholders
- *Motivation:* Simplifies implementation of some operations

Example:



Doubly Linked Lists

Variation: Nodes now have references to their *predecessors* as well as their *successors*



Advantages:

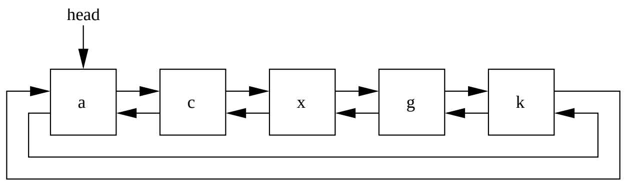
- Coding simplified (node's predecessor easily found)
- Some operations are now much more efficient

Disadvantage:

- extra storage overhead for the additional predecessor references

Circular Lists

Variation over Doubly-Linked List: Replace pair of dummy nodes with a single one



Advantage over Doubly Linked List:

- slightly less extra storage (only one dummy node)

Linked Lists in Java

A class

`LinkedList<E>`

is included as part of the package `java.util`.

This class is implemented as a doubly linked list, and it implements the

`List<E>`

interface.

Arrays versus Linked Lists

Reasons to Use an Array Instead of a Linked List

-
-

Reasons to Use a Linked List Instead of an Array

-
-

Additional Programming Support in Java

An `Iterator` is an object associated with any `Collection` — including any `LinkedList` .

This provides a kind of “marker” or “placeholder” that can be used, along with an enhanced `for` statement, to examine the elements of the associated `Collection`, one by one.

Additional information about this can be found in Section 6.3 of the textbook.

Additional Programming Support in Java

Q: Why Would You Need This?

A: Remember “information hiding” — and “programming by contract!”

This allows us to implement many of the algorithms that we could, if we had access to things like “pointers” to nodes in a linked list — without assuming anything about the internal representation of the objects we work with — and without needing direct access to `private` methods or data.