Lecture 4: Array

Rui Wang
Overview of Algorithms

• For most data structures
  – Iterate through all items
  – Insert a new data item
  – Search for a specified item
  – Delete a specified item
  – Sorting

  – The associated cost of these operations
Some Definitions

• **Database**
  – Phone directory, music library, online order database

• **Record**
  – A phone entry, a music piece, an order

• **Field**
  – A phone entry contains name, address, phone numbers ...
  – A music piece contains name, format, length, genre ...
  – An order contains order number, customer number, order amount...
Some Definitions

- Think of an excel spreadsheet:
Some Definitions

• Key
  – Also known as **Search Key**
  – A specific field used to search for a record
  – For example, search for an order using the order number; search for a record of a student using his last name.

• Record can be represented by a class. The class member variables correspond to the fields.
Today

• Basics of Array
• Insertion, Searching, Deletion
• Wrapping Array into Class
• ArrayList
• Ordered Array and Binary Search
Workshop Applet

• Array demo

• Insertion
  – Append new element at the end

• Searching
  – Linearly scan through the array for matching element

• Deletion
  – First search, then delete, and finally shift all elements behind it
  – Can it be made faster?
Workshop Applet

• Deletion
  – Requires filling the hole because data elements must be continuously stored without holes.
Array

• **Insertion**
  – How many steps does it take to insert an element?

• **Searching**
  – How many steps does it take to search an element?
  – Best-case scenario, worst-case scenario, average-case scenario?

• **Deletion**
  – Same questions above
Array

• **Insertion**
  – Takes only 1 step to insert (as long as there are available slots)

• **Searching**
  – Best-case: 1 step
  – Worst-case: \( N \) steps (\( N \): number of elements)
  – Average case: \( N/2 \) steps

• **Deletion**
  – Takes \( k \) steps to search, and \( (N-k) \) steps to move, so \( N \) steps overall.
Array

• Duplication issue
  – In many cases duplicates are not allowed, such as an employee’s SSN
  – In many other cases, duplication is common (such as one’s last name)
  – If duplicates are allowed, searching and deletion can become more complicated.
  – How does enabling duplication change the insertion, searching, and deletion cost?
Array with Duplication Allowed

- **Insertion**
  - Still takes 1 step

- **Searching**
  - Always takes $N$ steps

- **Deletion**
  - Seemingly more costly than before
  - However, still takes $N$ steps, if we perform searching and shifting together
  - Keep track of how many elements have been deleted so far
## Overall

<table>
<thead>
<tr>
<th></th>
<th>No Duplicates</th>
<th>Duplicates Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insertion</strong></td>
<td>0 comparison 1 assignment</td>
<td>0 comparison 1 assignment</td>
</tr>
<tr>
<td><strong>Search</strong></td>
<td>N/2 comparisons</td>
<td>N comparisons</td>
</tr>
<tr>
<td><strong>Deletion</strong></td>
<td>Average N/2 comparisons Average N/2 moves</td>
<td>N comparisons Up to N moves</td>
</tr>
</tbody>
</table>
Java Array

- **Creating an Array**
  ```java
  int[] myarray = new int[100];
  ```
- An array is an object, so here `myarray` is a reference.
Java Array

• The **size** of the array can be found by:
  \[ \text{myarray.length} \]

• Once the array is created, you CANNOT change its size!

• **Indexing**
  \[ \text{myarray[10]} = 66; \]
Java Array

• Initialization

1. For primitive data types, all elements are initialized to be zero by default.
   – But you can explicitly initialize the array by providing the list of elements:
     ```java
     int[] myarray = {2,3,5,7,9,11,13,17};
     ```
   – This will both create the array and initialize the elements.
Java Array

• Initialization

2. For class types, all elements are also initialized to be zero by default, thus they are all null.

Apple[] appArray = new Apple[100];
– At this point, all elements in appArray are null.
Java Array

- **Initialization**
  
  2. For class types, all elements are also initialized to be zero by default, thus they are all `null`.

```
Apple[] appArray = new Apple[100];
- At this point, all elements in appArray are null.
- To create object for each element, you can do:
  for (int i=0; i<appArray.length; i++) {
      appArray = new Apple();
  }
```
Array Example

• array.java
  – Note the difference between capacity and the number of available elements.
  – Simple example, but require explicit indexing, and does not allow for reuse of methods.
Wrapping Array into Class

- highArray.java
  - find()
  - insert()
  - delete()
  - Wrap the common operations into class methods, allowing for reuse
  - Does not require explicit indexing
Wrapping Array into Class

- **ArrayList<DataType>**
  - A useful Java class that implements the basic array operations we discussed
  - Use generics to allow different data types
  - Additionally allows for dynamic expanding of capacity.
  
  - How is this implemented?
  - Check it’s Java Doc page for more information.
class HighArray {
    private long[] a;
    private int nElems;
    public HighArray(int capacity) {...}
    public void insert(long value) {...}
    public boolean find(long key) {...}
}

HighArray arr = new HighArray(100);
arr.insert(999);
Wrapping Array into Class

• **ArrayList<DataType>**
  – A useful Java class that implements the basic array operations we discussed
  – Use generics to allow different data types
  – Additionally allows for dynamic expanding of capacity.
    • How is this implemented?
  – Check it’s Java Doc page for more information.
Wrapping Array into Class

class HighArray {
    private long[] a;
    private int nElems;
    public HighArray(int capacity) {...}
    public void insert(long value) {...}
    public boolean find(long key) {...}
}

HighArray arr = new HighArray(100);
arr.insert(999);
Wrapping Array into Class

HighArray arr;

Public HighArray(int capacity) {
    a = new long[capacity];
    nElems = 0;
}

Constructor

`HighArray arr = new HighArray(100);`

```
Public HighArray(int capacity) {
    a = new long[capacity];
    nElems = 0;
}
```
Public Methods

HighArray arr = new HighArray(100);

long[] a
int nElems

arr.insert(.)
arr.search(.)
arr.delete(.)
Data are Private, Methods are Public

HighArray  arr = new HighArray(100);

long[]  a
int  nElems

arr.insert(.);
arr.search(.);
arr.delete(.);
The Importance of Abstraction

• Wrapping the data storage and associated methods into a class has the advantage that it hides the implementation details from the user.

• So user only needs to concentrate on the functionality provided by the data structures.

• User cannot directly manipulate the data.
The Importance of Abstraction

• For all the data structures we learn, we will encapsulate the data storage and associated methods into a class.

• **Data**
  – Typically private, thus hidden from the user
  – No direct manipulation

• **Methods**
  – Typically public
  – Direct manipulation of data in the methods
Ordered Array

• An array in which data elements are sorted in ascending values.
  – The smallest value at index 0
  – Each element is larger than its previous element

• Insertion
  – How should you insert an element in order to preserve the ordered nature?
Ordered Array

• Searching
  – The main advantage of an ordered array is that it allows faster searching using binary search.

• Think of The Guess-a-Number game:
  – A friend asks you to guess a number that she is thinking of between 1 and 100
  – When you make a guess, she will tell you one of three things: too large, too small, your guess is correct.
Guess-a-Number Game

- Think about a strategy that leads to the fewest number of guesses on average.
Guess-a-Number Game

• Always start by guessing 50
• If she says it’s too small, you know the correct number must be within the range 51 to 100, so take 75 (halfway) as your next guess.
• Similarly if she says it’s too large, you know the correct number must be within the range 1 to 49, so your next guess is 25.
Guess-a-Number Game

- Each guess allows you to divide the range of possible values in half.
- Finally when the range is only one number long, you have the correct answer.
- Let’s take a look at how many guesses do you need to reach the correct answer.
Guess-a-Number Game

• Assuming the correct answer is 33:

<table>
<thead>
<tr>
<th>Step Number</th>
<th>Number Guessed</th>
<th>Result</th>
<th>Range of Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>1–100</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>Too high</td>
<td>1–49</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>Too low</td>
<td>26–49</td>
</tr>
<tr>
<td>3</td>
<td>37</td>
<td>Too high</td>
<td>26–36</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>Too low</td>
<td>32–36</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
<td>Too high</td>
<td>32–33</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>Too low</td>
<td>33–33</td>
</tr>
<tr>
<td>7</td>
<td>33</td>
<td>Correct</td>
<td></td>
</tr>
</tbody>
</table>
Binary Search

• **Goal**: given a specific key, find it’s location in the order array if it exists.

• **Workshop Applet Demo**

• Java Source Code
  – OrderedArray.java
public int find(long searchKey) {
    int lowerBound = 0;
    int upperBound = nElems - 1;

    while (true) {
        int curIn = (lowerBound + upperBound) / 2;
        if (a[curIn] == searchKey) {
            return curIn; // found it
        } else if (lowerBound > upperBound) {
            return nElems; // can't find it
        } else { // divide range
            if (a[curIn] < searchKey) {
                lowerBound = curIn + 1; // it's in upper half
            } else {
                upperBound = curIn - 1; // it's in lower half
            }
        } // end else divide range
    } // end while
} // end find()
```java
public int find(long searchKey) {
    int lowerBound = 0;
    int upperBound = nElems - 1;

    while (true) {
        curIn = (lowerBound + upperBound) / 2;
        if (a[curIn] == searchKey) {
            return curIn; // found it
        } else if (lowerBound > upperBound) {
            return nElems; // can't find it
        } else { // divide range
            if (a[curIn] < searchKey) {
                lowerBound = curIn + 1; // it's in upper half
            } else {
                upperBound = curIn - 1; // it's in lower half
            }
        } // end else divide range
    } // end while
} // end find()
```
public int find(long searchKey)
{
    int lowerBound = 0;
    int upperBound = nElems - 1;

    while(true)
    {
        curIn = (lowerBound + upperBound) / 2;
        if(a[curIn] == searchKey)
        {
            return curIn;  // found it
        }
        else if(lowerBound > upperBound)
        {
            return nElems;  // can't find it
        }
        else
        {
            if(a[curIn] < searchKey)
            {
                lowerBound = curIn + 1;  // it's in upper half
            }
            else
            {
                upperBound = curIn - 1;  // it's in lower half
            }
        }  // end else divide range
    }  // end while
}  // end find()}
public int find(long searchKey)
{
    int lowerBound = 0;
    int upperBound = nElems - 1;

    while(true)
    {
        curIn = (lowerBound + upperBound) / 2;
        if(a[curIn] == searchKey)
            return curIn; // found it
        else if(lowerBound > upperBound)
            return nElems; // can't find it
        else // divide range
        {
            if(a[curIn] < searchKey)
                lowerBound = curIn + 1; // it's in upper half
            else
                upperBound = curIn - 1; // it's in lower half
        } // end else divide range
    } // end while
} // end find()
public int find(long searchKey) {
    int lowerBound = 0;
    int upperBound = nElems - 1;

    while (true) {
        int curIn = (lowerBound + upperBound) / 2;
        if (a[curIn] == searchKey) {
            return curIn; // found it
        } else if (lowerBound > upperBound) {
            return nElems; // can't find it
        } else { // divide range
            if (a[curIn] < searchKey) {
                lowerBound = curIn + 1; // it's in upper half
            } else {
                upperBound = curIn - 1; // it's in lower half
            }
        } // end else divide range
    } // end while
} // end find()
The Ordered Array Class

• We’ve looked at **search**, what about **insertion** and **deletion**?

• orderedArray.java

• We will look at the cost of these operations next time.
Coming Up

• Assignment 1 is due at **4pm** today.
• Office hours today end at **4pm** as scheduled.
• To cope with potential submission issues, you have till **5pm** to send your submission, as described in the submission policy.
• After 5pm, your submission will be counted as late. Late policy will apply.
Coming Up

• You should receive an automatic reply email after you submit.
  – If not, it’s possible that your email is filtered as spam, because you didn’t use the correct title format (a? submission xxxx). If so, just send your submission again with the correct title format.
Coming Up

• If you have significant trouble working out assignment 1, you should seriously consider dropping this class. The upcoming assignments will be a lot more complicated and involved.

• Since this class is the foundation for upper-level classes, I will be very strict with my standards.
Coming Up

• Assignment 2 will be posted by the end of today.

• It has a programming part and a written part.
  – The programming part is a number guessing game, but DIFFERENT from the one we discussed in class!
  – The written part is about program analysis using the Big-O notation.