Universitas Esa Unggul

Smart, Creative and Entrepreneurial

www.esaunggul.ac.id

CMC 101 TOPIK DALAM PEMROGRAMAN

PERTEMUAN 9

PROGRAM STUDI MAGISTER ILMU KOMPUTER

FAKULTAS ILMU KOMPUTER

TOPIK DALAM PEMROGRAMAN
Struktur Data Dasar

Pertemuan 9

TUJUAN PERKULIAHAN

- Mahasiswa memahami beberapa tipe persoalan yang penting.
- Framework Analisis
  - Pengukuran Input
  - Unit untuk mengukur running time
  - Orde pertumbuhan
  - Efisiensi kasus terburuk, kasus terbaik, dan kasus rata-rata
- Notasi Asimptotik
  - Notasi O
  - Notasi Ω
  - Notasi Θ
- Kelas efisiensi dasar

Algorithm Efficiency, Big O Notation, ADT's, and Role of data Structures

- Algorithm Efficiency
- Big O Notation
- Role of Data Structures
- Abstract Data Types (ADTs)
- Data Structures
- The Java Collections API

Algorithm Efficiency

- Let’s look at the following algorithm for initializing the values in an array:

```
final int N = 500;
int [] counts = new int[N];
for (int i=0; i<counts.length; i++)
     counts[i] = 0;
```

- The length of time the algorithm takes to execute depends on the value of N

Algorithm Efficiency

- In that algorithm, we have one loop that processes all of the elements in the array
- Intuitively:
  - If N was half of its value, we would expect the algorithm to take half the time
  - If N was twice its value, we would expect the algorithm to take twice the time
- That is true and we say that the algorithm efficiency relative to N is linear

Algorithm Efficiency

- Let's look at another algorithm for initializing the values in a different array:

```
final int N = 500;
int [] [] counts = new int[N][N];
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
      counts[i][j] = 0;
```

- The length of time the algorithm takes to execute still depends on the value of N

Algorithm Efficiency

- However, in the second algorithm, we have two nested loops to process the elements in the two dimensional array
- Intuitively:
  - If N is half its value, we would expect the algorithm to take one quarter the time
  - If N is twice its value, we would expect the algorithm to take quadruple the time
- That is true and we say that the algorithm efficiency relative to N is quadratic

Big-O Notation

- We use a shorthand mathematical notation to describe the efficiency of an algorithm relative to any parameter n as its “Order” or Big-O
  - We can say that the first algorithm is O(n)
  - We can say that the second algorithm is $O(n^2)$
- Let T(n) be a function that formulates the time an algorithm needs to be completed, where n is the parameter that specifies the size of the problem, we say that the algorithm is O(T(n)) [or the algorithm has the time-complexity of O(T(n))].

Big-O Notation

- **Big-O notation measures how fast the the running time of the algorithm grows with increase in the size of the problem , not how long will it take for our algorithm to run as a function of the size of the problem. Therefore,**
  - We only include the fastest growing term and ignore any multiplying by or adding of constants. Since they are not dependent on the size of the problem.
  - If our time growth function has multiple terms dependent on the problem size n, we only take the dominating term as the Big-O measure.
  - Example

Seven Growth Functions

- Eight functions O(n) that occur frequently in the analysis of algorithms (in order of increasing rate of growth relative to n):
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - Log Linear ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
  - Factorial ≈ $n!$

Growth Rates Compared

	n=1	n=2	n=4	n=8	n=16	n=32
1	1	1	1	1	1	1
logn	0	1	2	3	4	5
n	1	2	4	8	16	32
nlogn	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
n!	1	2	24	40320	2.09e+13	2.63e+35

Big-O for a Problem

- O(T(n)) for *a problem* means there is some O(T(n)) algorithm that solves the problem
- Don't assume that the specific algorithm that you are currently using is the best solution for the problem
- There may be other correct algorithms that grow at a smaller rate with increasing n
- Many times, the goal is to find an algorithm with the smallest possible growth rate

Data Structures

- That brings up the topic of the Data structure on which the algorithm operates.
- Data Structure is a particular way of organizing the data in computer memory so that it can be used efficiently.

Role of Data Structures

- If we are using an algorithm manually on some amount of data, we intuitively try to organize the data in a way that minimizes the number of steps that we need to take. **As an example,** publishers offer dictionaries with the words listed in alphabetical order to minimize the length of time it takes us to look up a word.

Role of Data Structures

- We can do the same thing for algorithms in our computer programs
- Example: Finding a numeric value in a list
  - If we assume that the list is unordered, we must search from the beginning to the end
  - On average, we will search half the list
  - Worst case, we will search the entire list
  - Algorithm is O(n), where n is size of array or list.

Role of Data Structures

- Find a match with `value` in an unordered list

```
int [] list = {7, 2, 9, 5, 6, 4};

for (int i=0; i<list.length, i++)
   if (value == list[i])
       return true; // found it
return false; //did not find it.
```

Role of Data Structures

- If we assume that the list is ordered, we can still search the entire list from the beginning to the end to determine if we have a match
- But, <u>we do not need to search that way</u>
- Because the values are in numerical order, we can use a binary search algorithm
- Like the old parlor game “Twenty Questions”
- Algorithm is $O(\log_2 n)$, where n is size of array

Role of Data Structures

- Find a match with `value` in an ordered list

```
int [] list = {2, 4, 5, 6, 7, 9};
int min = 0, max = list.length-1;
while (min <= max) {
    if (value == list[(min+max)/2])
        return true;  // found it
    else
        if (value < list[(min+max)/2])
            max = (min+max)/2 - 1;
        else
            min = (min+max)/2 + 1;
}
return false;           // didn't find it
```

Role of Data Structures

- The difference in the structure of the data between an unordered list and an ordered list can be used to reduce algorithm Big-O
- This is the role of data structures and why we study them
- We need to be as clever in organizing our data efficiently as we are in designing an algorithm for processing it efficiently. In fact we can not separate one task from another.

Abstract Data Types (ADT's)

- A data type is a set of values and operations that can be performed on those values.
- The Java primitive data types (e.g. int) have values and operations defined in Java itself.
- An Abstract Data Type (ADT) is a (usually more sophisticated) data type that has values and operations that are not defined in the language itself. Instead, in Java, an ADT is implemented using a class or an interface.

Abstract Data Types (ADT's)

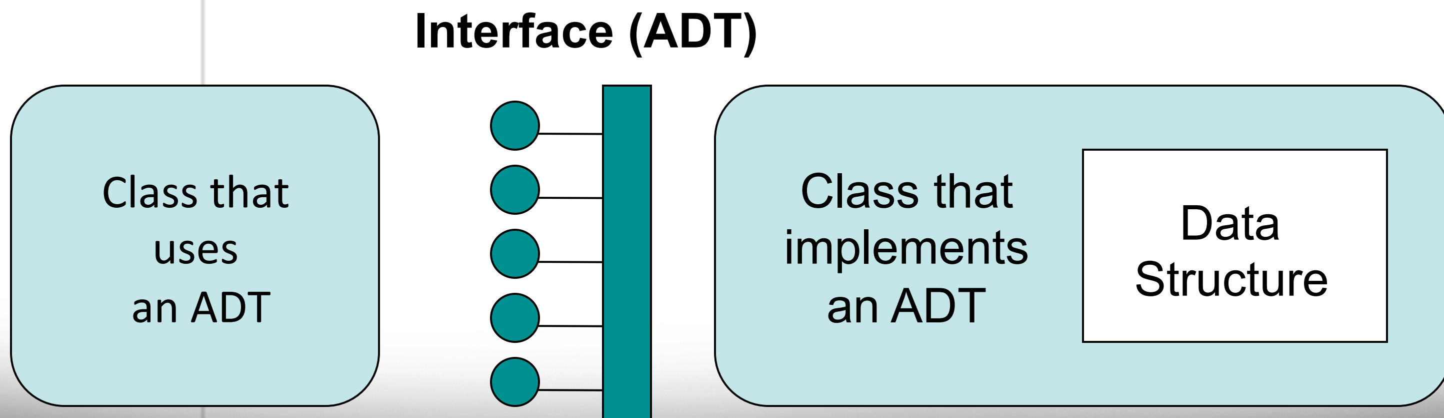
- The code for Arrays.sort is designed to sort an array of Comparable objects:

```
public static void sort (Comparable [ ] data)
```

- The Comparable interface defines an ADT
- There are no objects of Comparable "class"
- There are objects of classes that implement the Comparable interface.
- Arrays.sort only uses methods defined in the Comparable interface, i.e. compareTo().

ADT's and Data Structures

- Data structures are used to implement an Abstract Data Type. A data structure is used to:
  - to organize the data that the ADT is encapsulating.
- The type of data structure should be hidden by the API (the methods) of the ADT.



Collections

- A collection is a typical example of Abstract Data Type.
- A collection is a data type that contains and allows access to a group of objects.
- The Collection ADT is the most general form of ADTs designed for containing/accessing a group of objects.
- We have more specific forms of Collection ADTs which describe the access "strategy" that models that collection:
  - A *Set* is a group of things without any duplicates
  - A *Stack* is the abstract idea of a pile of things, LIFO
  - A *Queue* is the abstract idea of a waiting line, FIFO
  - A *List* is an indexed group of things

The Java Collections API

- The classes and interfaces in the Java Collections Library are named to indicate the underlying data structure and the abstract Data type.
- For example, the `ArrayList` we studied in CS110 uses an underlying *array* as the data structure for storing its objects and implements its access model as a *list*
- However, from the user's code point of view, the data structure is hidden by the API.

java.util.* Collection

Methods declared in Interfaces are hidden in subtypes

See also: Legacy Collection Diagram

www.falkhausen.de Version 0.9 Copyright 2002-04 by Markus Falkhausen. All rights reserved.

Collection

Accessors + Collectors
boolean isEmpty ()
boolean add / remove (Object o)
boolean add / removeAll (Collection c)
Object
boolean equals (Object o)
int hashCode ()
Other Public Methods
void clear ()
boolean contains (Object o)
boolean containsAll (Collection c)
Iterator iterator ()
boolean retainAll (Collection c)
int size ()
Object[] toArray ()
Object[] toArray (Object a[])

List

Accessors
Object get / set (int index)
Object set (int index, Object element)
Collectors
void add (int index, Object element)
boolean addAll (int index, Collection c)
Object remove (int index)
Other Public Methods
int indexOf (Object o)
int lastIndexOf (Object o)
ListIterator listIterator ()
ListIterator listIterator (int index)
List subList (int fromIndex, int toIndex)

Set

AbstractCollection

AbstractCollection ()
String toString ()

SortedSet

Comparator comparator ()
Object first ()
SortedSet headSet (Object toElement)
Object last ()
SortedSet subSet (Object fromElement, Object toElement)
SortedSet tailSet (Object fromElement)

AbstractSet

AbstractSet ()

AbstractList

AbstractList ()
void removeRange (int fromIndex, int toIndex)

Cloneable

Serializable

RandomAccess

TreeSet

TreeSet ()
TreeSet (Comparator c)
TreeSet (Collection c)
TreeSet (SortedSet s)
Object clone ()

HashSet

HashSet ()
HashSet (Collection c)
HashSet (int initialCapacity)
HashSet (int initialCapacity, float loadFactor)
Object clone ()

AbstractSequentialList

AbstractSequentialList ()

ArrayList

ArrayList ()
ArrayList (int initialCapacity)
ArrayList (Collection c)
Collectors
void removeRange (int fromIndex, int toIndex)
Object
Object clone ()
Other Public Methods
void ensureCapacity (int minCapacity)
void trimToSize ()

LinkedList

LinkedList ()
LinkedList (Collection c)
Accessors
Object getFirst ()
Object getLast ()
Collectors
void addFirst (Object o)
void addLast (Object o)
Object removeFirst ()
Object removeLast ()
Object
Object clone ()

LinkedHashSet

LinkedHashSet ()
LinkedHashSet (int initialCapacity)
LinkedHashSet (Collection c)
LinkedHashSet (int initialCapacity, float loadFactor)

Sumber : https://www.cs.umb.edu