# Data Structures and Algorithms
# Lecture 1

Overview and Guidelines on

Quality Software Design

# Motivation

- Why do you write programs?
- Why would you write quality software?

- How to write quality software?

- Why data structure is related to quality software design?

# Menu

- Data structures
- Motivation of studying data structures
- Language to study data structures
- Abstraction
- Huffman coding & priority queues
- Information hiding
- Encapsulation
- Efficiency in space & time
- Static vs dynamic data structures

# Data structures cover …

- General issues in data structure design;
- One-dimensional and multi-dimensional arrays;
- Linked lists, doubly-linked lists and operations on these data structures;
- Stacks and operations on stacks;
- Queues and operations on queues;
- Maps and operations on maps;
- Trees & Graphs.

# Data structures

- A data structure is a systematic way of organising a collection of data for efficient access
- Every data structure needs a variety of algorithms for processing the data in it
- Algorithms for insertion, deletion, retrieval, etc.
- So, it is natural to study data structures in terms of
  - **properties**
  - **organisation**
  - **operations**
- This can be done in a programming language independent way. (why?)
- Has been done in Pseudo code, Python, C, C++, Java, etc.

# Why data structures?

Aren't primitive types, like boolean, integers and strings, and simple arrays enough?

- Yes, since the memory model of a computer is simply an array of integers
- But, this model . . .
  - is conceptually inadequate & low level, since information is usually expressed in the form of highly structured data
  - makes it difficult to describe complex algorithms, since the basic operations are too primitive

# Why not just in Java?

Why do we study data structures in a language independent way?

- Java is just one of many languages within the category of object-oriented languages
- The world's most favourite programming language changes about every five to ten years
- However, data structures have been around since the invention of high-level programming languages

Therefore,

- We must be able to realise data structures in other languages
- We also need the abstract context to study algorithms

# Data structures that we will consider

We will study various data structures such as

- Arrays
- Lists
- Stacks
- Queues
- Maps
- Trees
- Graphs

In each case we will define the structure, give examples, and show how the structure is implemented in Java or pseudo code either directly or via other, previously defined, data structures.

# Software quality

- Before digging in the various data structures that will be the main topic of this module it is important to ask the question:
- Why are we doing this?
- Whenever we develop a software system we should strive to create good software.
- To achieve this a number of design principles could be followed.
- Furthermore, the quality of the eventual software can be measured in several ways:
  - correctness,
  - efficiency.
- **A careful design of the data structures used in software system helps in designing good software.**

# Topics

- In the forthcoming slides we first go though the main *principles that help developing good software*: **abstraction, information hiding, encapsulation**.
- We then talk briefly about how data structure design helps improving the quality of the final system.

- Finally we will address another important design issue related to data structures that help producing good software: the choice between **static** and **dynamic structures**.

# Abstraction (1)

- We can talk about abstraction either as a process or as an entity.
- As a process, abstraction denotes the extracting of the essential details about an item, or a group of items, while ignoring the nonessential details.
- As an entity, abstraction denotes a model, a view, or some representation for an actual item which leaves out some of the details of the item.

- Abstraction dictates that some information is more important than other information, but does not provide a specific mechanism for handling the unimportant information.

# Abstraction (2)

- In the context of software development, we can distinguish different kinds of abstractions:

- The aim of ***data* abstraction** is to identify which details of how data is stored and can be manipulated are important and which are not
- The aim of ***procedural* abstraction** is to identify which details of how a task is accomplished are important and which are not

# Key of abstraction…

- Extracting the **commonality** of components and hiding their details

- Abstraction typically focuses on the **outside view** of an object/concept

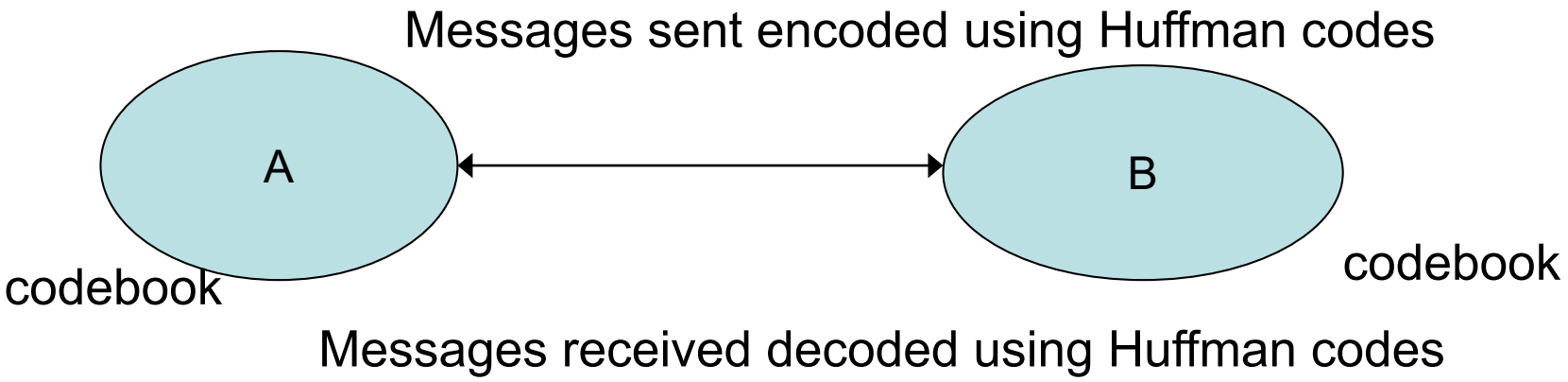
# Abstraction examples

- Looking at a map, we draw roads and highways, forests, not individual trees
- Looking at various bank accounts, what commonality can we extract?
  - Using an O-O approach
    - States
      - *AccountNo*
      - *CustomerName*
      - *Amount*
    - Behaviour
      - *Credit*, *Debit*, and *GetAmount*

# Use of abstraction in design

- Abstraction in design: break things into groups and figure out the details for each separately
- Abstraction leads to a top-down approach..
- Most projects can be improved with abstraction:
  - Think of the high-level. What do you want to accomplish? There should be one goal
  - Refine this goal into parts (components)
  - Think of multiple ways to implement each component

# Communication based on Huffman coding



# Huffman coding - Abstraction example

- Huffman coding is an effective way of encoding (and decoding) textual (or non-textual) data. It has been used in communication, e.g. source coding

- A large information system may need a piece of software that carries out the Huffman encoding of the data stored on a disk or generated from some data source.
- The Huffman encoding software uses priority queue to accomplish its tasks.
- The detailed description of such a module is given later.

- Important points:
  - The description abstracts from all details on how the priority queue and its operations are implemented.
  - Nonetheless the description enables a programmer to focus on the design of the particular module using the priority queue functions given.

# Huffman coding – Key ideas

- Based on the frequency of occurrence of a data item (pixel in images, alphabet in texts).
- The principle is to use a smaller number of bits to encode the data that occurs more frequently (why doing this?)

- Codes are stored in a ***Code Book*** which may be constructed for each image (alphabet) or a set of images
- In all cases the code book plus encoded data must be transmitted to enable decoding.

# More on Huffman coding – code book (code table)

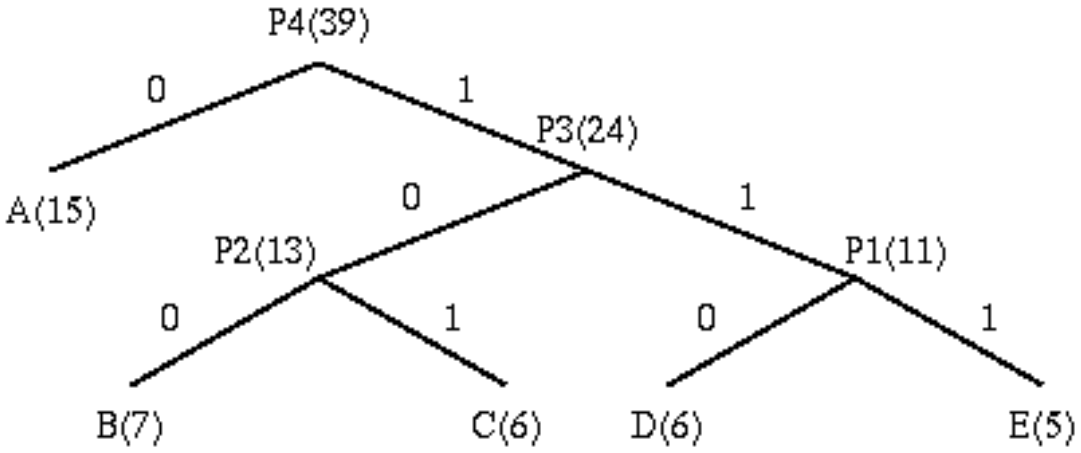
| Symbol | Count |
|---|---|
| A | 15 |
| B | 7 |
| C | 6 |
| D | 6 |
| E | 5 |

| Code | Symbol (Subtotal - # of bits) |
|---|---|
| 0 | A(15) |
| 100 | B(21) |
| 101 | C(18) |
| 110 | D(18) |
| 111 | E(15) |

TOTAL (# of bits): 87

Q: How do we generate Huffman codes?

# More on Huffman coding – en-/de-coding tree

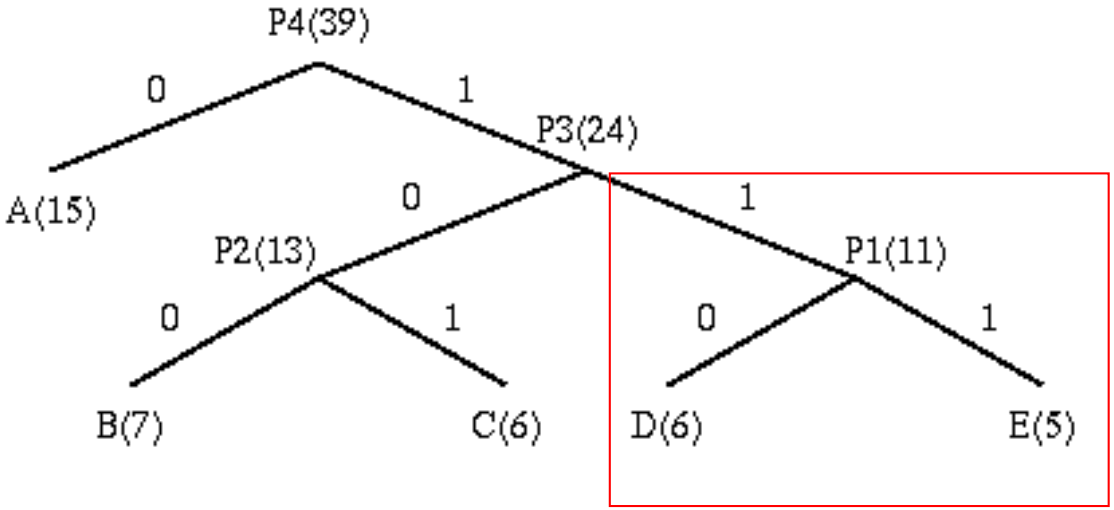



| Symbol | Count | Code | Subtotal (# of bits) |
|---|---|---|---|
| A | 15 | 0 | 15 |
| B | 7 | 100 | 21 |
| C | 6 | 101 | 18 |
| D | 6 | 110 | 18 |
| E | 5 | 111 | 15 |

TOTAL (# of bits): 87

List: A(15),B(7),C(6),D(6),E(5)

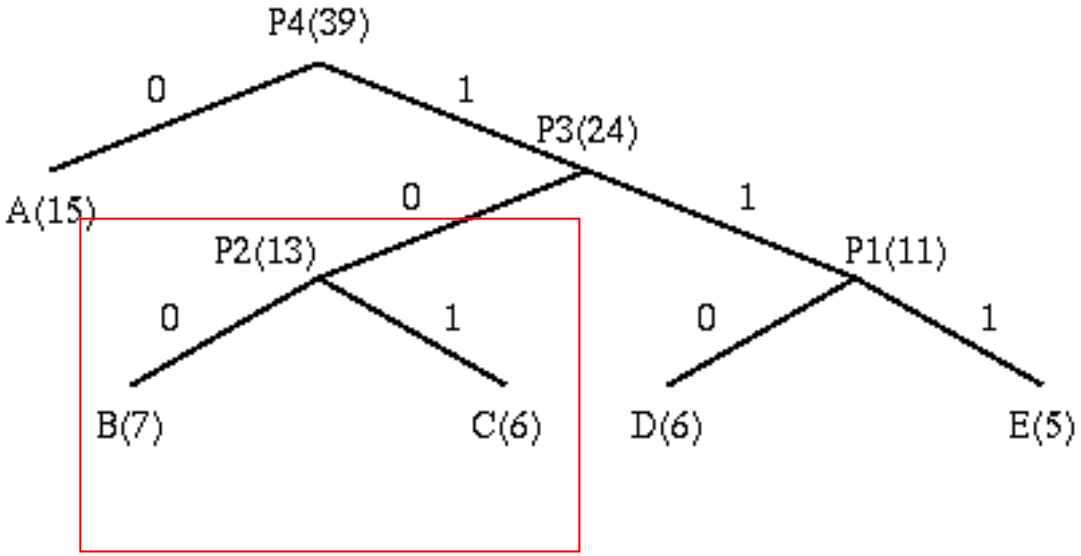
# Constructing the en-/de-coding tree




| Symbol | Count |
|---|---|
| A | 15 |
| B | 7 |
| C | 6 |
| D | 6 |
| E | 5 |

List: A(15),B(7),C(6),**D(6),E(5)** → List: A(15),B(7),C(6),*P1(11)*

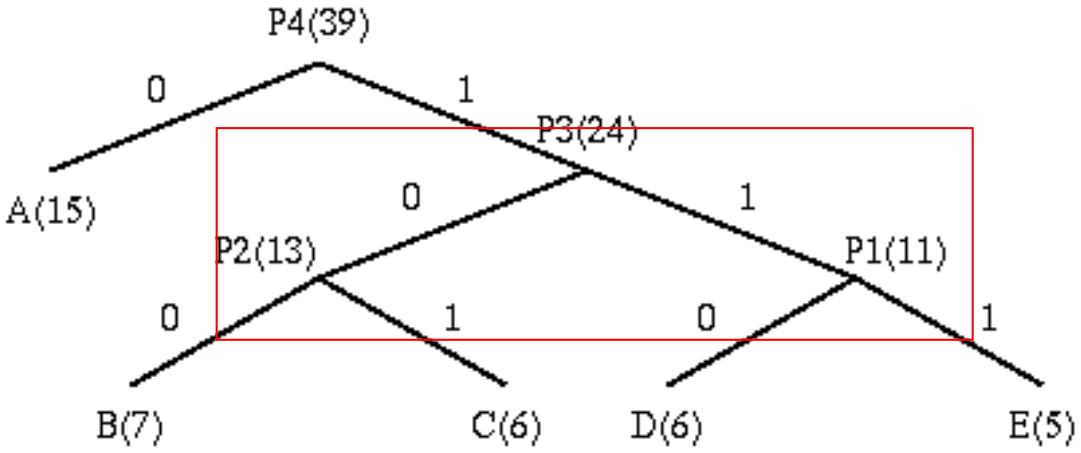
# Constructing the en-/de-coding tree




| Symbol | Count |
|---|---|
| A | 15 |
| B | 7 |
| C | 6 |
| D | 6 |
| E | 5 |

List: A(15),**B(7),C(6),**P1(11) → List: A(15),P1(11),*P2(13)*

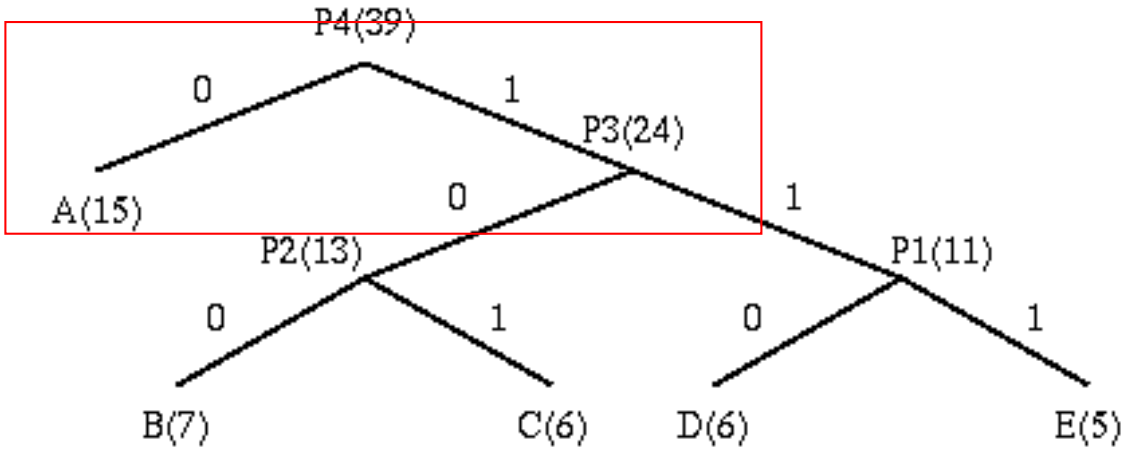
# Constructing the en-/de-coding tree




| Symbol | Count |
|---|---|
| A | 15 |
| B | 7 |
| C | 6 |
| D | 6 |
| E | 5 |

List: A(15),**P1(11),P2(13)** → List: A(15),*P3(24)*

# Constructing the en-/de-coding tree & table




| Symbol | Count |
|---|---|
| A | 15 |
| B | 7 |
| C | 6 |
| D | 6 |
| E | 5 |

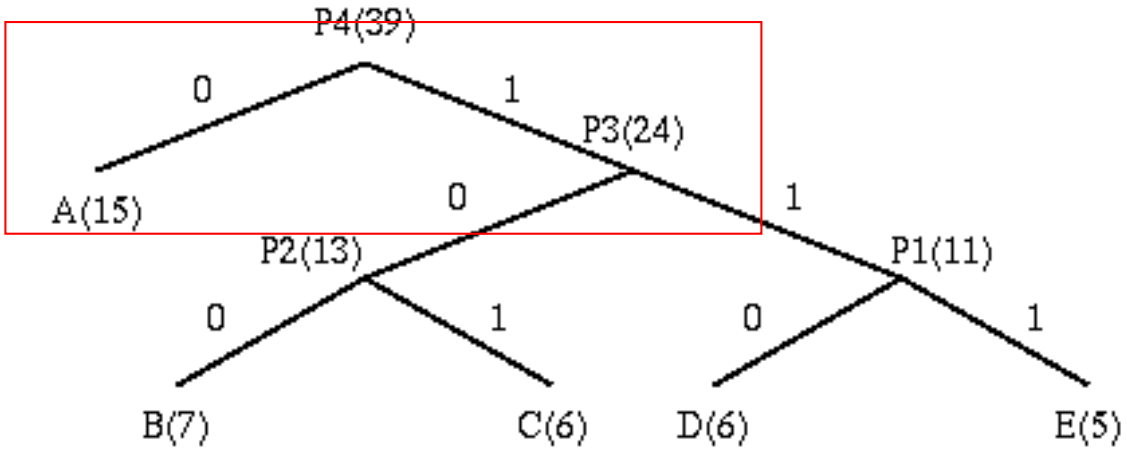
| Code | Subtotal (# of bits) |
|---|---|
| 0 | 15 |
| 100 | 21 |
| 101 | 18 |
| 110 | 18 |
| 111 | 15 |

TOTAL (# of bits): 87

List: **A(15),P3(24)** → List: *P4(39)*

END

# Huffman decoding




**Codebook**

| | |
|---|---|
| A | 0 |
| B | 100 |
| C | 101 |
| D | 110 |
| E | 111 |

**What are the input symbols if the following is received?**

**(1) 011011100**
**(2) 0110111001**
**(3) 1001110110**

(1) 0,110,111,0,0 → ADEAA
(2) 0,110,111,0,0,1 → ADEAA + error
(3) 100,111,0,110 → BEAD

# Exercise

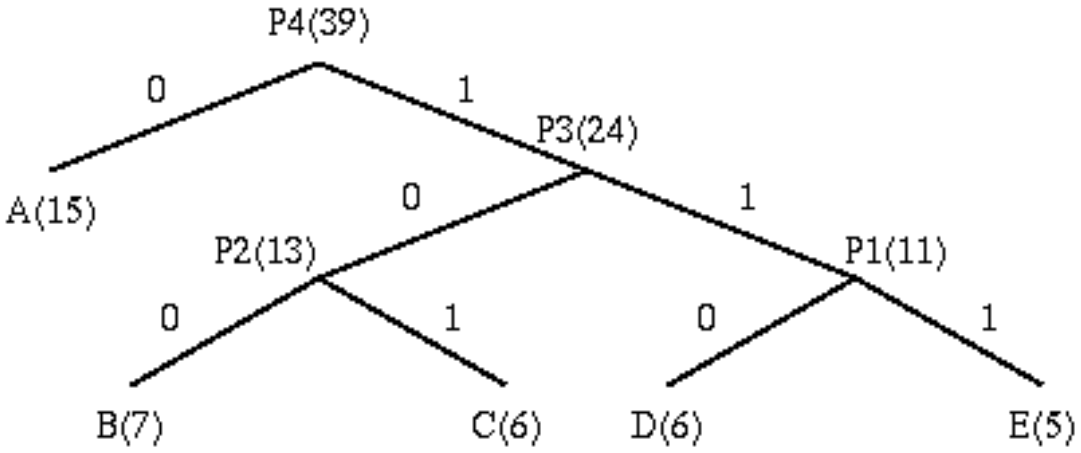
- Given the following symbol frequency table, design the corresponding Huffman coding scheme for it.

| Symbol | Count | Code | Subtotal (# of bits) |
|---|---|---|---|
| A | 1 | | |
| B | 3 | | |
| C | 5 | | |
| D | 7 | | |
| E | 11 | | |
| | | | TOTAL (# of bits): |

- Using the codebook developed, decode:
  - 10011
  - 01101011
  - 0010001

# Why is Huffman coding optimal?

# Optimality of Huffman coding




| Symbol | Count | Code | Subtotal (# of bits) |
|---|---|---|---|
| A | 15 | 0 | 15 |
| B | 7 | 100 | 21 |
| C | 6 | 101 | 18 |
| D | 6 | 110 | 18 |
| E | 5 | 111 | 15 |

TOTAL (# of bits): 87

## Use of abstraction in the design of Huffman coding

- Components
  - Encoding
    - Algorithms?
    - Data structures?
  - Decoding
    - Algorithms?
    - Data structures?
  - Codebook management
    - Algorithms?
    - Data structures?

# Detecting commonality in Huffman coding – en-/decoding tree construction

Data structure used:
List A(15),B(7),C(6),D(6),E(5) →
List A(15),B(7),C(6),P1(11) →
List A(15),P1(11),P2(13) →
List A(15),P3(24) →
List P4(39)

Priority queue
Operations: EXTRACT-MIN, INSERT

# Use of abstraction in Huffman encoding – data structure & algorithm

- A *priority queue* is a data structure for maintaining a set $Q$ of elements each with an associated value (and *key*).
- A priority queue supports the following operations:
  - INSERT(*Q*,*x*) inserts the element *x* into *Q*.
  - MIN(*Q*) returns the element of *Q* with minimal key.
  - EXTRACT-MIN(*Q*) removes and returns the element of *Q* with minimal key.

- Question: difference btn MIN & EXTRACT-MIN

# Use of abstraction in Huffman encoding – data structure & algorithm

- A *binary tree* is a data structure for maintaining a set *Q* of nodes each with an associated value and options of left and right child nodes.

- A *binary tree* supports the following operations:
  - ALLOCATE-NODE creates a new node, returning a reference to the node z created.
  - right(*z*) refers to the right child node of z.
  - left(*z*) refers to the left child node of z.

# Implementation of Huffman coding using priority queues & binary trees

S is a data structure containing pairs ($a$, $f$ [$a$]) where
$a$ is a character in the alphabet and $f$ [$a$] its frequency in the text.
$Q$ is a priority queue, initially empty.

---

Priority Queue functions (methods) we need:

**EXTRACT-MIN($Q$)**
**INSERT($Q$, $z$)**

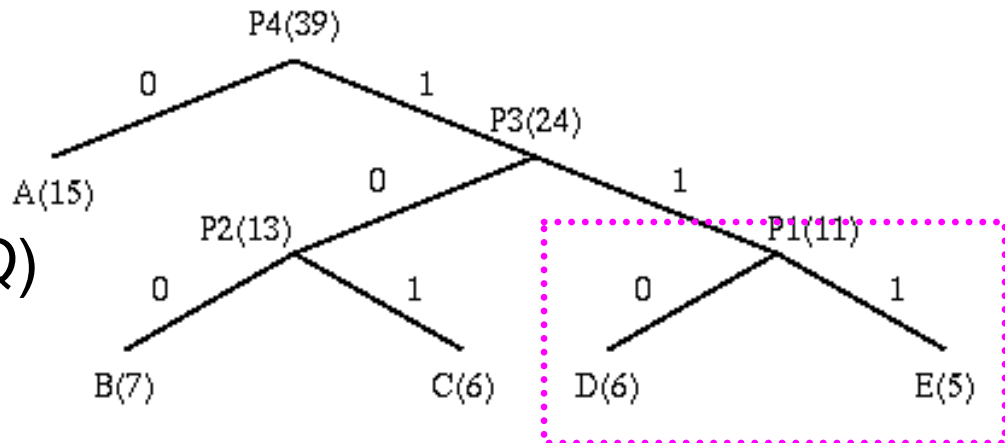
Binary Tree functions (methods) we need?

S is a data structure containing pairs ($a$, $f[a]$) where
$a$ is a character in the alphabet and $f[a]$ its frequency in the text.
$Q$ is a priority queue, initially empty.

---

HUFFMAN ENCODING (building tree from leaves)

$n \leftarrow |S|$; $Q \leftarrow S$;
**for** $i \leftarrow 1$ **to** $n-1$
{
$z \leftarrow$ ALLOCATE-NODE()
right[$z$] $\leftarrow$ EXTRACT-MIN($Q$)
$x \leftarrow$ right[$z$]
left[$z$] $\leftarrow$ EXTRACT-MIN($Q$)
$y \leftarrow$ left[$z$]
$f[z] \leftarrow f[x] + f[y]$
INSERT($Q$, $z$)
}
**return** EXTRACT-MIN($Q$)



## Implementation of Huffman coding using priority queues & binary trees

- Not only priority queue is used in the encoding algorithm shown above
- But also binary tree

- Review again these slides after we finish covering ‘Binary trees’

How does application of ‘abstraction’ principle simplify the task of Huffman coding?

## Exercise:

## implement the algorithm for Huffman decoding

- Input: a pointer to the decoding tree root *z* & received Huffman encoded binary string *i*
- Output: Huffman decoded string

- Do this near the end of this term

# Information hiding

- Information hiding is the principle that users of a module need to know only the essential details of this module (as identified by abstraction)
- So, abstraction leads us to identify details of a module which are important for a user and which are unimportant

- Information hiding tells us that we should actively keep all unimportant details secret from the user and try to prevent him from making use any unimportant details

- The important details of a module that a user needs to know form the specification of a module
- So, information hiding means that modules are used via their specifications, not their implementations.

# Information hiding

- Information hiding hides the internal data or information from direct manipulation.

- Information hiding is related to *privacy, security* (Why?)

# Information hiding example

- Cars provide an example of this in how they interface with drivers.
- They present a ***standard interface***
  - pedals,
  - wheel,
  - shifter,
  - signals,
  - switches, etc.
- on which people are trained and licensed.
- Implications of this??
- Thus, people only have to learn to drive a car; they don't need to learn a completely different way of driving every time they drive a new model.

# Use cases of information hiding

- Hide the physical storage layout for data so that if it is changed, the change is restricted to a small subset of the total program.
- For example, if a three-dimensional point (*x*,*y*,*z*) is represented in a program with three floating point scalar variables and later, the representation is changed to a single array variable of size three…

- A module designed with information hiding in mind would protect the remainder of the program from such a change.

# Information hiding in an O-O world

- In a well-designed object-oriented application, an object **publicizes *what*** it can do—that is, the services it is capable of providing, or its method headers—but
- ***hides*** the internal details both of ***how*** it performs these services and of the data (attributes & structures) that it maintains in order to support these services.

# Java method signature

- The **signature** of a method is the combination of
  - method's name along with
  - number and types of the parameters (and their order).

- *public void setMapReference(int xCoordinate, int yCoordinate)*
  *{*
  *//method code – implementation details*
  *}*
- The method signature is setMapReference(int, int)

# Encapsulation

- Like abstraction, we can consider encapsulation either as a process or as an entity:
- As a process, encapsulation means the act of enclosing one or more items (data/functions) within a (physical or logical) container.
- As an entity, encapsulation, refers to a ***package*** or an enclosure that holds (contains, encloses) one or more items (data/functions).

- The separator between the inside and the outside of this enclosure is sometimes called ***wall*** or *barrier.*

# Encapsulation in an O-O world

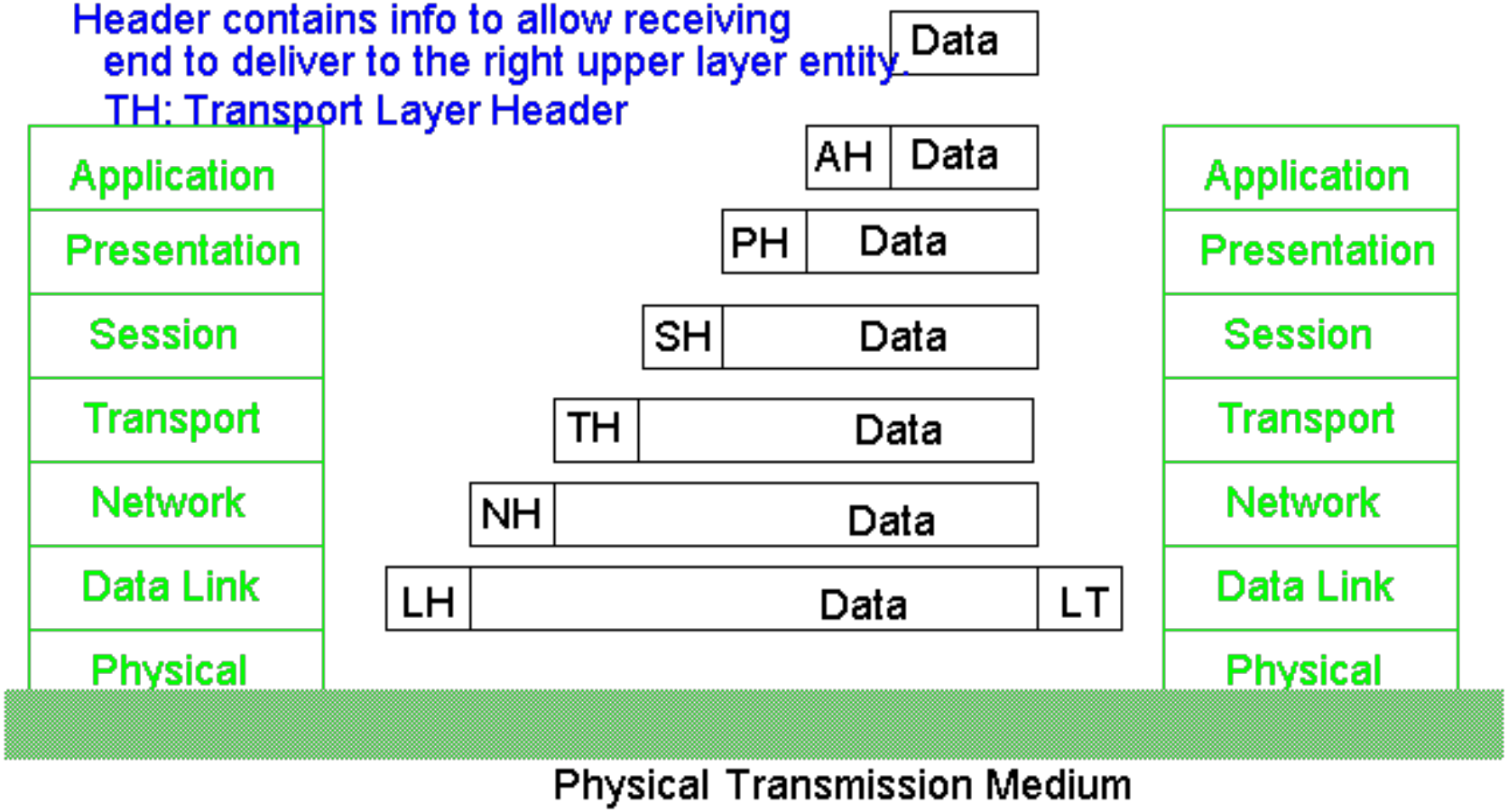
- In object-oriented programming, encapsulation is the inclusion within an object of all the resources needed for the object to function – i.e., the methods and the data.
- The object is said to publish its interfaces.
- Other objects adhere to these interfaces to use the object without having to worry how the object accomplishes it.
- The idea is: don't tell me how you do it; just let me know what you can do!
- An object can be thought of as a self-contained atom. The object interface consists of public methods and instantiated data.

# Encapsulation in communication

- In communication, encapsulation is the inclusion of one data structure within another structure so that the first data structure is hidden.
- For example, a TCP/IP-formatted packet can be encapsulated within an ATM frame.
- Within the context of sending and receiving the ATM frame, the encapsulated packet is simply a bit stream that describes the transfer.

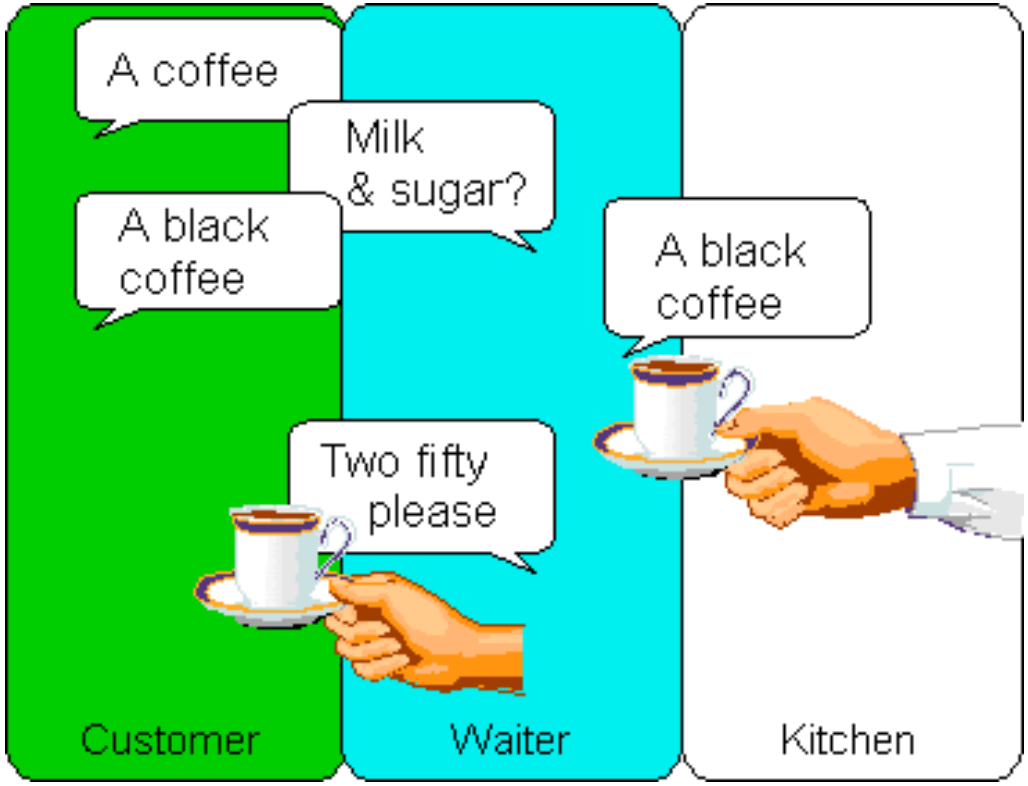
# Packet encapsulation

Message between entities consist of two parts: header and payload.
Data from upper layer are put in the payload.
Header contains info to allow receiving end to deliver to the right upper layer entity.
TH: Transport Layer Header



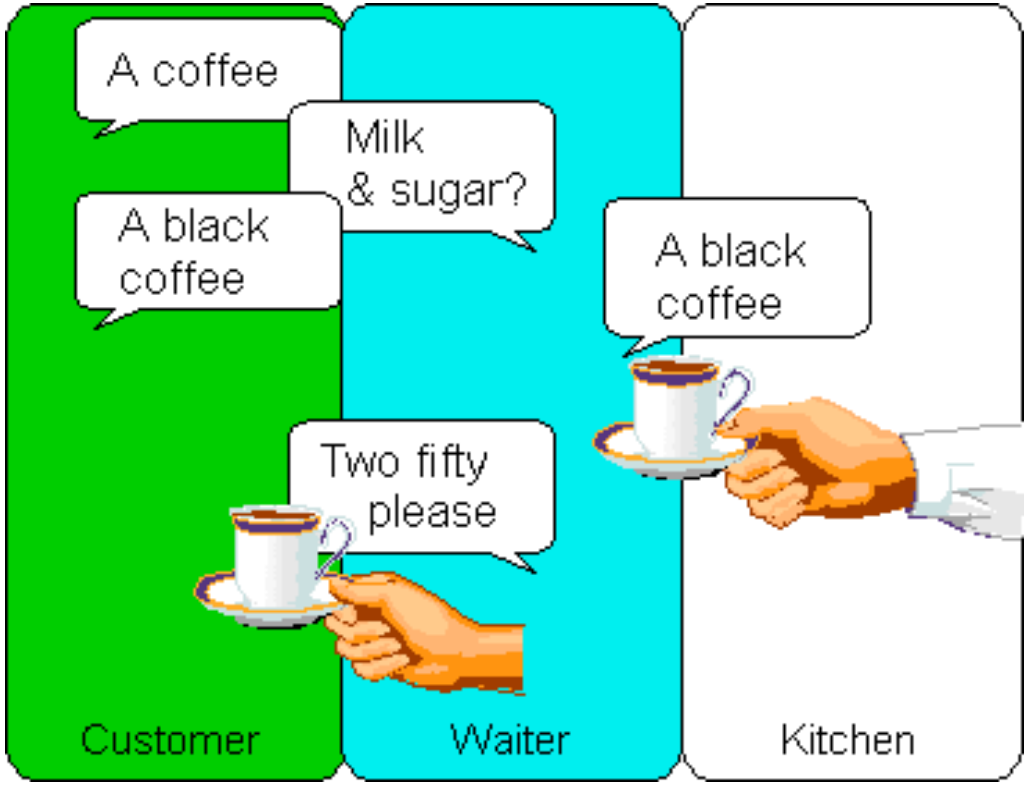
# **Encapsulation example –** 'cup of coffee'

- *Customer*, *waiter* and *kitchen* are three shielded objects in the 'cup of coffee' example.
- Customer does what?
- Waiter does what?
- Kitchen does?
- How do you encapsulate each?
- How do you interface them?



# Encapsulation example

- The customer does not care about the coffee brewing process.
- Even the waiter does not care.
- Encapsulation keeps computer systems flexible. The business process can change easily. (for example?)



# Comparisons

- Abstraction, information hiding, and encapsulation are different, but related, concepts
- **Abstraction** is a technique that helps us identify which specific information is important for the user of a module, and which information is unimportant

- **Information hiding** is the principle that all unimportant information should be hidden from a user
- **Encapsulation** is then the technique for packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible.

# Advantages

Using the processes of abstraction and encapsulation under the guidinglines of information hiding, we enjoy the following advantages:

- Simpler, **modular programs** that are easier to design & understand
- **Side-effects** from direct manipulation of data are **eliminated** or minimised
- **Localisation of errors** (only methods defined on a class can operate on the class data), which allows localised testing
- Program modules are **easier to read, change, and maintain**…

# Data structures & abstraction, info hiding, encapsulation

- Data structures represent one big common factor across programs
- Specification of data structures requires the use of abstraction, info hiding, encapsulation

- Practice of abstraction, info hiding, encapsulation on modular programming leads to further development of data structures
- Further development of data structures leads to better modular programming

# Exercise

- Using the info hiding & encapsulation concepts learnt, design a *courier service package* with the following roles inside.
  - Sender
  - Courier receptionist
  - Shipping agent (or delivery agent)
  - Receiver

- Identify the services/functions of each role.
- Clarify how encapsulation is achieved & highlight the service boundary by using a Java method signature like interface.

# Efficiency: Space

- A well-chosen data structure should try to minimise memory usage (avoid the allocation of unnecessary space)

- Examples:
- Storing a drawing/map via vectors vs. bitmaps vs. compressed bitmaps
- Tradeoffs of space efficiency vs convenience

# Efficiency: Time (1)

- A well-chosen data structure will include operations that are efficient in terms of speed of execution (based on some well-chosen algorithm)
- For our purposes the most important measure for the speed of execution will be the number of accesses to data items stored in the data structure

# Efficiency: Time (2)

**Example:**

- Consider a list of the names of *n* students and the operation we want to perform is searching for a specific name
- Suppose we use a data structure for implementing lists that allows direct access to an arbitrary element of the list.
- If the list is not sorted in any way, then in the worst case we need to look at each of the *n* names on the list (*n* accesses)
- (Q: what is a worst case?)(best case?)(average case?)
- If the list is sorted alphabetically, then we can use **binary search** and in the worst case we need to look at $\log_2(n)$ names on the list ($\log_2(n)$ accesses)

# Static versus dynamic data structures (1)

- Besides time and space efficiency another important criterion for choosing a data structure is whether the number of data items it is able to store can adjust to our needs or is bounded
- This distinction leads to the notions of dynamic data structures vs. static data structures
- **Dynamic data structures** grow or shrink during run-time to fit current requirements e.g. a structure used in modelling traffic flow
- **Static data structures** are fixed at the time of creation
- e.g. a structure used to store a postcode or credit card number (which has a fixed format)

# Static versus dynamic data structures (2)

- Note that it is the *structure* that is static (or dynamic), not the data
- So, in a static data structure the stored data can change over time, only the structure is fixed
- Of course, the stored data could also stay constant in both static and dynamic data structures
- Note that in the definition of a static data structure we have placed no constraints on when it is created, only that ***once it is created it will be fixed***
- This will be important when we determine whether arrays in Java are static data structures or not!

# Example

- An example of a static data structure is an array:

  int [ ] a = new  int [ 50 ] ;

  which allocates memory space to hold 50 integers
- The language provides the construct '**new**' to indicate to the compiler/interpreter how much space of which data type needs to be allocated
- In Java, arrays are always dynamically allocated even when we write:

  inta [ ] = { 10 , 20 , 30 , 40 } ;
- However, the size of the array is fixed
- Q: Is a Java array a static or dynamic data structure?

# Static data structures

**Advantages**

ease of specification

- Programming languages usually provide an easy way to create static data structures of almost arbitrary size

no memory allocation overhead

- Since static data structures are fixed in size,
  - there are no operations that can be used to extend static structures;
  - such operations would need to allocate additional memory for the structure (which takes time)

# Static data structures

**Disadvantages**

- must make sure there is enough capacity

  Since the number of data items we can store in a static data structure is fixed, once it is created, we have to make sure that this number is large enough for all our needs

- more elements? (errors), fewer elements? (waste)
  - However, when our program tries to store more data items in a static data structure than it allows, this will result in an error (e.g. ArrayIndexOutOfBoundsException)
  - On the other hand, if fewer data items are stored, then parts of the static data structure remain empty, but the memory has been allocated and cannot be used to store other data

# Dynamic data structures

- **Advantages**
- There is no requirement to know the exact number of data items since dynamic data structures can shrink and grow to fit exactly the right number of data items, there is no need to know how many data items we will need to store
- Efficient use of memory space
  - extend a dynamic data structure in size whenever we need to add data items which could otherwise not be stored in the structure and
  - shrink a dynamic data structure whenever there is unused space in it, then the structure will always have exactly the right size and no memory space is wasted

# Dynamic data structures

**Disadvantages**

- Memory allocation/de-allocation overhead
- Whenever a dynamic data structure grows or shrinks, then memory space allocated to the data structure has to be added or removed (which requires time)

# Q&A

- Both space efficiency & time efficiency are metrics used to evaluate the performance of an algorithm (and a data structure). (T or F?)
- Dynamic data structures are more space efficient in general. (T or F?)
- Static data structures are more time efficient in general. (T or F?)
- Information hiding is the principle that users of a software component need to know only the essential details of how to *initialize* and *access* the component, and do not need to know the details of the implementation (T or F?)

# Summary

- Data structures
- Motivation of studying data structures
- Language to study data structures
- **Abstraction**
- **Huffman coding & dynamic queues**
- **Information hiding**
- **Encapsulation**
- **Efficiency in space & time**
- **Static vs dynamic data structures**

# Application of ‘abstraction’ ..

- **Data Mining**, the process of extracting patterns from data, has been applied in many fields to obtain good economical results.
- **Big Data Analytics**
- **Knowledge Discovery in Databases** (KDD) is an emerging field under data mining.
- Just as many other forms of knowledge discovery, KDD creates **abstractions** of the input data.

# Readings

- [Mar07] Read 3.1, 3.2, 6.1, 6.4
- [Mar13] Read 3.1, 3.2, 6.1, 6.4