

Data Structures

Introduction

1. Principles
2. Organization of topics
3. Prerequisite skills
4. Course details

1. Principles

- In Data Structures we chiefly study how to manage *collections*
- We choose a data structure because it lends itself to *efficient handling*
- Tools of *algorithm analysis* let us evaluate efficiency of a solution
- We distinguish *implementation* and *interface*

Abstract data type (class):

- ...is a category of data item specified in terms of an instance's data components and the operations that can be performed on it
- An abstract data type or class has an interface and an implementation
- We implement ADTs in Java and C++ with *class* and in C with *struct* and functions that take structures as parameters

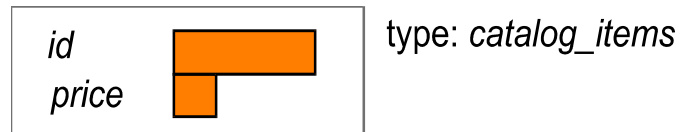
Arrays of objects

- A *collection* may be implemented as a class whose members are an array of objects and an integer denoting the number of structure in the collection
- *Example:*

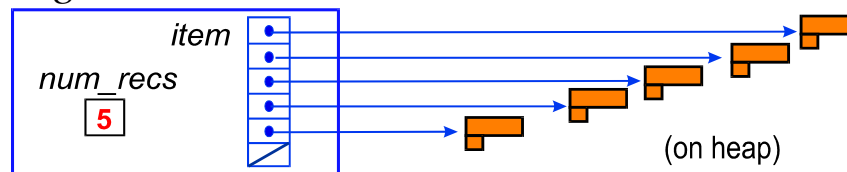
```
public class Employee_roster
{
    Employee emp[100];
    int num_employees;
};
```

Memory layouts for objects and collections

An object to store in a collection:



A generalized collection:



Data structures

Interfaces

- General collection
- Matrix
- Queue
- Dictionary
- Stack
- Priority queue

Implementations

- Array (linear or two-dimensional)
- Dynamically-allocated linked list nodes
- Dyn. alloc. binary search tree nodes

Levels of abstraction

collection, dictionary,
stack, queue

interface

array, linked list,
binary tree

implementation

bits in RAM or on disk

hardware

Comparing implementations

	Sorted Array	List	Bin. search Tree
Speed of access	+	-	+
Speed of modification	-	+	+
Flexibility	-	+	+
Ease of programming	+	-	-

2. Organization of topics

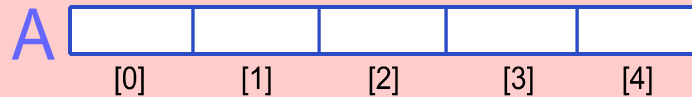
1. Numeric, string, class, and array data
2. Linked lists
3. Stacks and queues
4. Priority queues, heaps
5. Trees
6. Hashing
7. Graphs
8. Multithreading
9. Graphics programming

T1. Numeric, string, class, and array data

- Software engineering relies on *testing and verification* at *design* stage
- *Divide-and-conquer strategy*: break down a complex problem into a simple problem and a smaller version of the original problem
- We want a formula to estimate running time that applies regardless of hardware or quantity of data

Array:

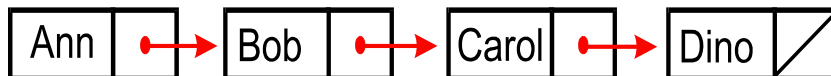
A series of data locations in memory, all of the same type, accessed by the array's name and subscript.



- A *random-access file* has the same cellular-style organization as an array
- Strings are *objects* that contain arrays of characters

T2. Linked lists

- A linked list is a series of *nodes*, each containing a data value and a pointer to the next node.

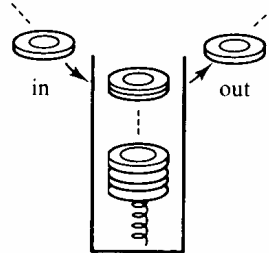


- Unlike an array, a linked list is expandable
- Applications: implementing general collections, stacks, queues, hash tables, graphs

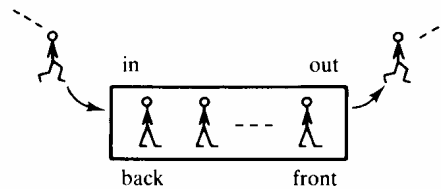
T3. Stacks and queues

- Collections with restricted access

- **Stack:**
last in, first out

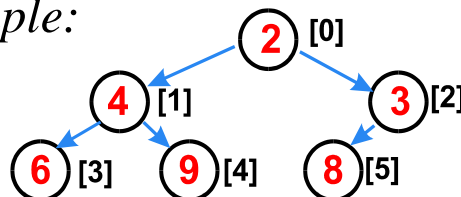


- **Queue:**
first in, first out



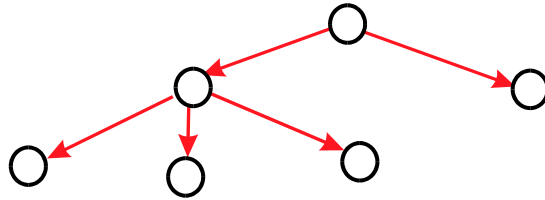
T4. Heaps and priority queues

- *Problem:* Store a collection with fast insertion in order, fast extraction of maximum element
- *One solution:* A structure that we think of as a tree, stored in an array, with special features
- *Example:*



T5. Trees

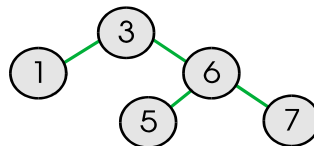
- A tree is a kind of graph
- Each vertex but *root* has one *parent*, has 0 or more *children*



- A vertex may represent a data node
- *Examples:* family tree, outline, module hierarchy, organization chart

Binary search trees

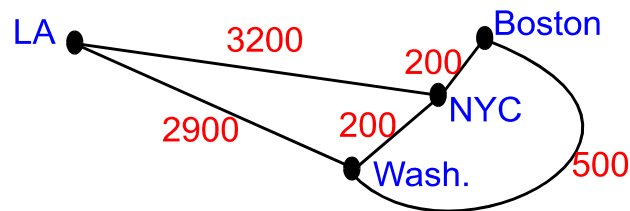
- A BST is a branching structure composed of nodes, each of which contains a data value and references to up to two other nodes.
- Rules of building a BST enable fast insertion, deletion, and retrieval



T6. Hashing

- *Goal:* To beat the speed of phonebook-style (binary) search
- *Ideal:* Constant-time access to an element of a collection, given a key value
- *Tool:* Hash function to derive the location (subscript) from key value
- *Problems:* What if hash function puts two elements of collection in the same location?

T7. Directed and undirected graphs



- Here: a graph with four vertices (cities), 5 edges (connections between cities)
- How do we find the shortest route from Boston to LA?
- How do we represent a graph in memory?

T8. Multithreading

- *Multithreading* is a form of concurrency within a single application
- A thread simulates the exclusive use of a processor
- Java supports multithreading through *Thread* class

T9. Graphics programming

- *Graphics processing* uses two forms: bitmaps, and vector graphics
- The Java language supports graphical user interfaces, including views and controls
- GUI programming is *event-driven*
- The Java *awt* package supports drawing of graphics

3. Prerequisite skills

Use of math in Data Structures

- *Algorithm design*: use of *recursion* (induction)
- *Specification and verification*: use of *logical assertions*
- *Algorithm analysis*: use of asymptotic (going-to- ∞) view of behavior of functions
- *Prerequisites*: CS II, Precalculus

CS II-level basic skills

1. Proficiency with method calls and definitions, including the use of parameters;
2. Strong skills with nested loops such as those found in search and sorting algorithms;
3. String and array manipulation skills in Java, including arrays of objects;
4. Use of dynamic allocation, including linked lists;
5. Strong testing, tracing and debugging skills;

4. Course details

Other basic skills

- Identifying $O(1)$, $O(n)$, and $O(n^2)$ algorithms;
- Identifying linked-list, heap, tree, hash table, and graph structures.

Semester project

- You will write an application that will use all the general collections presented here.
- Submission will be in installments
- The purpose is to test the theory presented here in light of experience
- Some key aspects: documentation, testing, algorithm analysis

- Other objectives
- Group work
- Grading