Summary of Data Structures

• General-Purpose
  – Arrays
  – Linked List
  – Trees
  – Hash Tables

• Specialized
  – Stacks
  – Queues, Priority Queues
  – Graphs
Summary of Algorithms

• Associated with each data structure
  – Insert
  – Delete
  – Search (find)

• Sorting
  – Bubble, insertion, selection
  – Merge sort
  – Quick sort
  – Radix sort
Summary of Algorithms

• Recursion
  – Conceptually simple way of solving many problems.
  – Factorial
  – Recursive binary search
  – The Towers of Hanoi
  – Fractals

  – Frequent use in data structures (e.g. trees)
  – How recursion works in a programming language
  – Eliminating Recursion
When to Use What?

• **General-Purpose**
  – For storing real-world data and searching.

• **Speed**
  – Array and Linked Lists < Trees < Hash Tables
  – This is talking about a large amount of data.
  – However, the faster ones also impose limitations
    • Trees are complex
    • Hash tables require prior knowledge about data.
  – Take all aspects into account when making a choice!
Start

Small amount of data?

No

Search and insertion must be very fast?

Yes

Hash Table

No

Key distribution guaranteed random?

Yes

Binary Search Tree

No

Balanced Tree

Yes

Amount of data predictable?

No

Linked list

Yes

Search speed more important than insertion speed?

Yes

Ordered Array

No

Unordered array
Start

Small amount of data? Yes → Amount of data predictable? No → Linked list

Small amount of data? No → Searching and insertion must be very fast? No → Key distribution guaranteed random? Yes → Binary Search Tree

Small amount of data? No → Searching and insertion must be very fast? Yes → Hash Table

Search speed more important than insertion speed? Yes → Ordered Array

Search speed more important than insertion speed? No → Unordered array

Key distribution guaranteed random? No → Balanced Tree
When to Use What?

• **Binary Trees**
  – Defined by nodes and (two) child links.
  – Terminology
  – Traversal (pre-order, in-order, post-order) → **definitely will be tested**.
When to Use What?

• **Binary Search Trees (BST)**
  – Definition.
  – Find minimum / maximum → **code**
  – Search (very similar to binary search) → **code**
  – Insertion (always inserts **as** a leaf node) → **code**
  – Deletion (three cases) → **know how to**

  – Average costs in all three: $O(\log N)$
  – However, may degenerate to $O(N)$ in some cases (**unbalanced** BST).
When to Use What?

- **Self-Balancing Trees**
  - Definition of Balance.
  - Rotation → **code, know how to**
  - AVL tree (adjust balance using one of four cases) → **know how to**
  - Red-Black Tree (enforce 4 rules) → **know how to**
  - Example: Is the following tree a valid Red-Black tree? How to fix the red-red violation?

- Average costs $O(\log N)$, **no degenerate case.**
When to Use What?

• Hash Tables
  – Typically the fastest structure, suitable for sparse data (map a large range to a small range)
  – Key code (e.g. convert a string to a number)
  – Hash function (we usually use %)
  – Solving collisions:
    1. Open Addressing: three probing methods
    2. Separate Chaining.
  – Load factor and Hashtable efficiency.
When to Use What?

- **Hash Tables**
  - Open addressing is suitable when you have a good idea of the amount of data.
  - Three probing methods → **code, calculation**
  - Separate chaining is suitable when the amount of data is hard to predict → **know how to**
  
  - Hashtable Disadvantage: no easy way to do ordered traversal; rehashing (open addressing) is expensive.
When to Use What?

- **Comparison of General Purpose Structures:**

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Search</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>—</td>
</tr>
<tr>
<td>Ordered array</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Linked list</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>—</td>
</tr>
<tr>
<td>Ordered linked list</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary tree (average)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary tree (worst case)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Balanced tree (average and worst case)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Hash table</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>—</td>
</tr>
</tbody>
</table>
When to Use What?

- Comparison of General Purpose Structures:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Search</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>—</td>
</tr>
<tr>
<td>Ordered array</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Linked list</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>—</td>
</tr>
<tr>
<td>Ordered linked list</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary tree (average)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary tree (worst case)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Balanced tree (average and worst case)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Hash table</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>—</td>
</tr>
</tbody>
</table>
When to Use What?

- **Comparison of General Purpose Structures:**

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Search</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$-$</td>
</tr>
<tr>
<td>Ordered array</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Linked list</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$-$</td>
</tr>
<tr>
<td>Ordered linked list</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary tree (average)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary tree (worst case)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Balanced tree (average and worst case)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Hash table</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

Require explicit sorting
When to Use What?

• **Specialized**
  – Stacks, Queues, Priority Queues.
  – Usually used as program aids.
  – Typically implemented using one of the general-purpose structure.
Sorting

• Quadratic Sorting
  – Bubble, insertion, selection
  – Easy to implement, in-place sorting
  – Slow

• Log-Linear Sorting
  – Merge sort (easy to implement, but requires extra memory) → know how to
  – Key step: merge() → what it means, code
Sorting

• **Log-Linear Sorting**
  – **Quick sort** (fastest, sorts in-place) → **know how to**
  – Key step: partition() → **what it means, code**
  – What is a pivot element? Where is it after partition?
  – Selection of pivot element is important for the efficiency of quick sort.
Sorting

• Non-Comparison-Based Sorting
  – Often achieve $O(N)$ time sorting.
  – Radix sort $\rightarrow$ **know how to**
Graphs

• Made of **vertices (nodes)** and **edges**.
• Edges may be directed (directed graph).
• How is a graph different from a tree?
• Terminology
  – Example: is this graph a connected graph?
• Adjacency matrix, adjacency list
  – Given an adjacency matrix, draw the graph → **will be tested**.
Libraries

- **Language Implementations:**
  - Array, Vector, ArrayList
  - Linked Lists
  - Stacks
  - Hashtable, Set, Map
  - Sorting

- **Other Libraries**
  - Binary trees, Self-balancing Trees
  - Graphs
Final Exam

• May 6, 10:30am—12:30pm Marcus Hall 131
• Requests for schedule conflicts must be sent to me by **5pm today**.
• Exam format is similar to mid-term:
  – Multiple choices
  – Short answers
  – Programming
• It is accumulative: 1/3 first semester, 2/3 second semester.
Final Exam

• You **MUST pass the final exam** in order to pass this class.

• The level of difficulty is about the same with mid-term exam.

• Study the quizzes, it is **VERY important**!
  – >50% of questions in the final exam are based on questions appeared in the quizzes.
  – Make sure you understand each question, not just reciting the answer.
Final Exam

• Schedule for the remaining lectures:
  – Wednesday: review of quizzes, assignment 6
  – Thursday: programming exercises
  – Tuesday next week: Graph 2
A Brief Introduction to Physical Computing
What is Physical Computing

• Computation that involves **sensing** and **responding** the physical world.
• Why is this relevant?
What is Physical Computing

- Computation that involves sensing and responding the physical world.
- Why is this relevant?
  Increasingly, computation is not all about designing algorithms, it is also about the means of interacting and communicating with computers.
- Human-computer interaction is where a lot of innovations are happening today.
What is Physical Computing

• What does it involve?
  – Micro-controllers
  – Sensors
  – Servos
  – Electro-mechanical devices
  – Real physical elements
  – Most importantly: **Programming Skills**! (this is what you learn through this class...)
What is Physical Computing

• To give you some flavors:
  – Traveler’s Umbrella Interaction
  – Lego Rubik’s Cube Solver
  – Wooden Mirror
Micro-Controllers

• These interaction systems often use a tiny computer chip called **micro-controllers**.
  – A single circuit tiny computer.
  – Input/output through digital/analog pins
  – Directly talk to electronic component (e.g. LEDs, servos) and other hardware.
  – Serial communication.
  – Low power consumption.
Micro-Controllers

• Micro-controllers are everywhere in our life:
  – Calculators.
  – Cell phones.
  – Printers.
  – Routers.
  – Home appliances.
  – They are the **brains** of modern gadgets.
Micro-Controllers

• They run programs to control physical devices.

• An example (toaster oven):

  \textit{heat at a desired temperature for a desired amount of time}

  \begin{verbatim}
  start_time = get_time();
  while (get_time() - start_time < \texttt{desired_time}) {
    if (get_temperature() < \texttt{desired_temperature} - 15)
      turn on heating element;
    else if (get_temperature() > \texttt{desired_temperature} + 15)
      turn off heating element;
  }
  \end{verbatim}
Micro-Controllers

• What does a micro-controller look like?
  – Many types, and different form factors

• The **Arduino** Micro-controller
  – 16 MHz CPU frequency
  – 32 KB flash memory
  – 2 KB RAM (main memory)
  – 14 input/output pins
  – USB port
  – Easy to program
Micro-Controllers

• How do you program a micro-controller?
• Arduino allows you to program in Java style programming language.
Micro-Controllers

• Some examples:
  – Blink a LED.
  – Control LED brightness from Processing.
  – Control LED using a button
  – Putting a lot of LEDs and Buttons together = Game controller!
  – Big LED display
Micro-Controllers

• Some cool DIY projects:
  – Build a simple robot.
  – Remote controlled door lock.
  – Physical email notifier.
  – Control electronic appliances through the web.
  – Monitor your home power consumption and tweet it on the web.
Micro-Controllers

• **Concluding Remarks:**
  – Physical computing involves sensing and responding to the physical world.
  – Often achieved with micro-controllers – tiny little computers that directly talk to electric devices.
  – Programming is an integral part of physical computing.
  – Some micro-controllers are programmed with Java style language.
Micro-Controllers

• Concluding Remarks:
  – The whole point is to provide you with a broader and refreshing view of what computing and programming mean.
  – How programming can be entertaining, fun.
  – How it is relevant to our life.