Review

Jasleen Kaur

Spring 2014

Abstractions

- One of the 3 major concepts in the official syllabus:
  - Abstraction
    - Abstractions simplify by hiding the details
    - CS is too complex to know everything
    - Allow us to know about more things
    - e.g., circuits, bits, bytes, Java, data structures, ...
    - As a user, abstractions are helpful
      - But someone has to know the details – you, for data structures
  - Correctness
    - Always a requirement
      - We’ll deal with only through your implementations (pre-conditions, post-conditions, testing)
  - Efficiency (algorithmic complexity)
    - Which choice will be more efficient for large amount of data?
- We’ll focus mostly on abstraction and efficiency
Abstract Data Types (ADTs)

- Abstract Data Types:
  - A set of objects together with a set of operations
  - Mathematical abstraction
    - Does not describe how the set of operations is implemented

- Algorithm:
  - A high level, language-independent description of a step-by-step process

- Data Structure:
  - A specific organization of data and family of algorithms for implementing an ADT (data + operations)

- Implementation of a data structure or ADT:
  - A specific implementation in a specific language

STACKS & QUEUES

ADT, Applications
An ADT Example – Stacks

- Stacks are (last-in-first-out) LIFO lists
  - Only the top element is accessible
  - Usually drawn vertically

- Supports operations:
  - push – add an element at the top
  - pop – remove an element from the top
  - top – return (not remove) the element at the top
  - isEmpty – return true if the Stack is empty
  - ...

- Example Stack:
  - push(6)
  - push(3)
  - push(1)
  - push(4)

Stack Application: Balanced Parenthesis

- How to check if parenthesis are balanced in a mathematical expression?
  - Algorithm:
    - Read expression symbol-by-symbol
    - For each opening parenthesis encountered, push it onto the stack
    - For each closing parenthesis encountered, pop the top element from the stack
    - If parenthesis matched, stack should be empty at the end

- Examples:
  - (12 * 3 + (42 / 4) + (3 – 5 * (4 / 2))) + 42
  - (12 * 3 + (42 / 4) + (3 – 5 * (4 / 2))) + 42

- This is exactly how compilers check your code and return errors!
Another ADT Example – Queues

- Queues are (first-in-first-out) FIFO lists
  - Usually drawn horizontally

- Supports operations:
  - enqueue – add an element at the rear
  - dequeue – remove an element from the front
  - peek – return (not remove) the element at the front
  - isFull – return true if the queue is full
  - ...

- Example Queue:
  - enqueue (6)
  - enqueue (3)
  - enqueue (1)
  - enqueue (4)

---

Queue Applications

- Virtually, every real-life line supposed to be a queue
  - First-come-first-served

- Common data structure for arbitrating resource access
  - Printer queue
  - File server queue
  - Network access queue

- Whole branch of mathematics devoted to studying queues!
  - Queuing Theory
  - Probabilistically computes average wait times, queue lengths, etc
Array or Linked List: Which one to use?

**Array**
- May waste unneeded space or run out of space
- Uses just-enough space per element
- Operations (so far) very simple/fast
  - Access to $k^{th}$ element
    - Simple access
  - insertAtPosition:
    - Must shift all later elements

**Linked List**
- Always just enough space
- But needs more space per element
- Operations (so far) very simple/fast
  - Access to $k^{th}$ element
    - Must traverse all earlier elements
  - insertAtPosition:
    - After traversal, simple insert

Which operations would run faster?

RUNNING TIME ESTIMATIONS

Big-O Notation, Code Analysis
Big-O vs $\Theta$

- **$O$: upper bound**
  - $T(N) = O(f(N))$, if there are positive constants $c$ and $n_0$ such that $T(N) \leq cf(N)$ when $N \geq n_0$

- **$\Omega$: lower bound**
  - $T(N) = \Omega(f(N))$, if there are positive constants $c$ and $n_0$ such that $T(N) \geq cf(N)$ when $N \geq n_0$

- **$\Theta$: tight bound**
  - $T(N) = \Theta(f(N))$, if and only if $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$

- **Common practice:**
  - Use a tight (smallest) bound in specifying $O(.)$
  - We’ll also try to do this in this course

Summary: Run-time vs. $O(.)$

- **$O(.)$:**
  - Helps reason about performance as a function of $n$

- **Run-time:**
  - Important for small $n$
    - Or not-so-small $n$, depending on the constant factors
  - Cross-over point $n_0$ changes as technology advances
    - CPU and memory speed
    - New efficient compilers
    - Clever programming

- **Demo: Tester.java**
How to Sort?

- Applications of sorting:
  - Clearly, searching in a sorted array can be more efficient
    - O(log N) vs O(N)
  - Many databases list elements in a sorted order
    - e.g., alphabetical

- How to sort?
  - Insertion sort
  - Merge sort
  - Quick sort
  - Bucket sort
  - External sorting
Criteria

- Run-time:
  - What is the run-time complexity of the algorithm?

- Space:
  - How much space does the sorting algorithm require for sorting?
    - Is copying needed? – O(N) additional space
    - In-place sorting – O(1) additional space

- Stability:
  - Does sorting retain input order for duplicates?
    - e.g., if names originally in alphabetical order, then sorted according to county, would names still be sorted within a county?
  - Important property for databases

Insertion Sort: Properties

- Running time?
  - Worst case is O(N^2) – reversed input
  - Best case is O(N) – for almost sorted data
    - What if all elements are equal?
  - Average case is Ω(N^2)
    - True for any sorting algorithm that exchanges only adjacent elements

- Space?
  - Is an in-place sorting algorithm – O(1) extra space

- Sorting algorithm of choice when:
  - Data is nearly sorted
  - Data is small (due to low overhead)
    - When input has shrunk in divide-and-conquer algorithms
A Lower Bound for Sorting

- Average case run-time for insertion sort is $\Omega(N^2)$
  - True for any sorting algorithm that exchanges only adjacent elements

- Why?
  - Inversion: any pair $(i,j)$, $i < j$, such that $a[i] > a[j]$
  - Average number of inversions in an array: $N(N-1)/4$
    - Any list and its reverse have a total of $N(N-1)/2$ inversions
  - In algorithms that swap only adjacent elements:
    - Each swap removes only one inversion
    - So, $\Omega(N^2)$ swaps are required

- To do better, must:
  - Compare and swap elements that are far apart
    - Try to eliminate more than just one inversion per swap

Divide and Conquer: Sorting Examples

- Mergesort
  - Divide array into two halves
  - Recursively sort left and right halves
  - Merge the two halves

- Quicksort
  - Partition array into small and large items
  - Recursively sort the two sets
Divide and Conquer: Run-time Speedup

- Assume: each step itself takes constant time
  - If each step:
    - Reduces the problem size by at least a constant factor
    - Complexity – $O(\log N)$: binary search
  - If each step:
    - Divides the problem into a constant number of parts, and conquers these
    - Then combines the solutions in linear time
    - Complexity – $O(N \log N)$: mergesort
  - If each step:
    - Divides the problem into a constant number of parts, and conquers these
    - Then combines the solutions in constant time
    - Complexity – $O(N)$: findMin

Mergesort: Properties

- Run-time – $O(N \log N)$ for all input
  - Merging step always takes $\Theta(N)$ time $\Rightarrow$ sorting is $\Theta(N \log N)$
- Space complexity:
  - Uses linear extra memory – $O(N)$
- Additional work of copying to the temporary array (and back) throughout the algorithm slows the sort considerably
  - This cost is language dependent
    - Java (generic sort)
      - Comparisons may be expensive
      - Moving elements is cheap
    - C++ (generic sort)
      - Copying elements can be expensive
      - Comparing can be made cheap by compiler
- Quicksort – uses fewer moves, but some more comparisons
**Quicksort: Properties**

- **Demo:** [http://www.sorting-algorithms.com/quick-sort](http://www.sorting-algorithms.com/quick-sort)
- **Run-time**
  - Worst-case – $O(N^2)$
  - Best-case – $O(N \log N)$
  - Average-case – $O(N \log N)$ – uses very few comparisons
- **Space:**
  - $O(\log N)$ – $O(N)$: uses less space than mergesort
- **Stability?**
  - Not stable – e.g., pivot is reordered with respect to same-value elements
- **Not good for small arrays**
- **Has good in-memory performance**
  - Small footprint and good locality

**Bucket Sort: Properties & Limitation**

- **Run-time:**
  - $O(M+N)$
  - If $M$ is $O(N)$, then run-time is $O(N)$
- **How does it improve the comparison-sort lower bound?**
  - Uses a more powerful operation than simple comparisons
  - Performs M-way comparisons in unit time!
- **Limited usability if values can be large (< M)**
  - Too many buckets needed!
  - e.g., just 10 numbers in the range 0 to 999!
Radix Sort: Properties

- Each pass is stable
  - Items that agree in the current digit retain the ordering determined in prior passes
  - This is why the algorithm works in the first place!

- Run-time
  - $O(p(N+b))$
    - $p$: number of passes
    - $N$: number of elements to sort
    - $b$: number of buckets
The Priority Queue ADT

- Operations?
  - `insert`
  - `deleteMin`
  - `min`
  - `create, isFull, isEmpty, ...`

- Implementation options:

<table>
<thead>
<tr>
<th></th>
<th>create</th>
<th>insert</th>
<th>min</th>
<th>deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>O(1), O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted array</td>
<td>O(1), O(N)</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Linked list</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Binary tree</td>
<td>O(1), O(N)</td>
<td>O(log N)</td>
<td>O(1)</td>
<td>O(log N)</td>
</tr>
</tbody>
</table>

Binary Heaps

- Are binary trees satisfying two additional properties:
  - “Structure” property:
    - Levels are “filled” in order, left to right
    - Also known as complete binary trees
      - All levels (except possibly bottom) completely filled
      - Bottom level filled from left to right
  - “Heap order” property:
    - Key in parent of X ≤ key in X
      - Only defined for Comparable objects
    - Where would the minimum be?
    - Where would the maximum be?
Binary Heaps: Structural Properties

- How many nodes does a binary heap with height \( h \) have?
  - \( \geq 2^h \)
  - \( \leq 2^{h+1} - 1 \)

- What is the height of a binary heap with \( N \) nodes?
  - \( \lfloor \log_2 N \rfloor \)

Binary Heaps: Array Implementation

- Structure:
  - Root at \( A[0] \)
  - \( A[x] \) has left child at \( A[2x+1] \)
  - \( A[x] \) has right child at \( A[2x+2] \)
  - \( A[x] \) has parent at \( A[(x-1)/2] \)

- Optimization – move everything right by 1
  - Root at \( A[1] \)
  - \( A[x] \) has left child at \( A[2x] \)
  - \( A[x] \) has right child at \( A[2x+1] \)
  - \( A[x] \) has parent at \( A[x/2] \)
Binary Heaps: Other Operations

- **Heap operations**
  - min: $O(1)$
  - insert: $O(\log N)$
  - deleteMin: $O(\log N)$

- **Additional operations:**
  - decreaseKey($p$, $\Delta$): $O(\log N)$
  - increaseKey($p$, $\Delta$): $O(\log N)$
  - delete($p$): $O(\log N)$
  - buildHeap: $O(N)$
  - find($x$): find element $x$
  - findMax: find the maximum element
  - merge($H_1$, $H_2$): merge two heaps, each of size $O(N)$
    - $O(N)$ inserts => $O(N \log N)$ time
    - Copy $H_2$ at end of $H_1$ and use buildHeap => $O(N)$ time!

All operations we have seen so far take just $O(1)$ extra space!!

Dynamic Dictionaries

Jasleen Kaur

Spring 2014
The Dynamic Dictionary ADT

- Operations?
  - insert, remove
  - find, max, min
  - create, isFull, isEmpty, ...

- Implementation options:
  - insert         find        remove
  - Array          O(1)        O(N)        O(N)
  - Sorted array   O(N)        O(log N)   O(N)
  - Linked list    O(1)        O(N)        O(N)
  - Sorted linked list O(N)      O(N)        O(1)
  - Binary search tree (BST)
    - average O(N)      O(N)        O(N)
    - Balanced BST O(log N)   O(log N)   O(log N)
  - Hash tables    O(N)        O(N)        O(N)
    - average O(1)        O(1)        O(1)

Binary Trees: Postorder Traversal

- Postorder: process left subtree, right subtree, then node
  - Inorder: 1, 2, 3, 4, 7, 6, 8
  - Preorder: 6, 2, 1, 4, 3, 7, 8
  - Postorder: 1, 3, 7, 4, 2, 8, 6

```java
public static <T> void postOrder(bNode<T> root){
    if (root == null) return;
    postOrder(root.left);
    postOrder(root.right);
    process(root.data);
}
```

```java
       6
      / \
     2   8
    /   /
   1    4
   / \   / \
  3   7 16
```
Binary Search Trees (BSTs)

- Important application of binary trees – searching
- Binary search tree – for every node with value X,
  - Values of all items in left subtree are ≤ X
  - Values of all items in right subtree are > X
- Implication: all tree elements can be ordered consistently
  - Only defined for Comparable class

“Balance” in a BST

- Issue: tree may be imbalanced
  - Deletions – favor making left subtree deeper
  - Insertions – if data is presorted, tree has only nodes with no left children
- Add an extra structural condition – balance
  - No node is allowed to get too deep
  - Updates take longer on average

- How to balance?
  - Left and right subtrees should have same height
  - Every node must have left and right subtree of same height
AVL Trees: Operations

- All lookup-based operations exactly the same as BST
  - Upper bound on height: $1.44 \log(N+2) - 1.328$
  - Typical height ~ $\log N$
- Updates must preserve the balance property, though
  - Insert, remove
    - $O(\log N)$ to do simple BST insert, remove
    - 2-4 link manipulations to correct balance
The Dynamic Dictionary ADT

- Operations?
  - insert, remove
  - find, max, min
  - create, isFull, isEmpty, ...

- Implementation options:

<table>
<thead>
<tr>
<th>Implementation options</th>
<th>insert</th>
<th>find</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted array</td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Linked list</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Binary search tree (BST)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Hash tables</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Balanced BST</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>Hash tables</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Hashing – Implementing Hash Tables

- Ingredients of a hash table:
  - An array of some fixed size, containing the items
    - A[0, ..., TableSize-1]
  - A hash function that maps a key to an array location
    - Hash: Keys \( \rightarrow \) \( \{0, ..., \text{TableSize}-1\} \)
    - Location also referred as “cell”/“bucket”
  - A collision resolution strategy
    - What to do when two different keys map to the same cell
    - Alternatively, where to insert if cell is already filled up?

Hashing not suitable for: min, max, sorting!
Hash tables – collision resolution: **probing**

1. Chaining (closed addressing)
2. **Probing (open addressing)**
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

In case of collision, try alternative locations until an empty cell is found.

Probe sequence: $h_0(x), h_1(x), h_2(x), \ldots$, with $h_i(x) = \left[\text{hash}(x) + f(i)\right] \% \text{tableSize}$

The function $f(i)$ is different for the different probing methods.

---

Hash tables – collision resolution: **Cuckoo hashing**

**Goal:** constant-time $O(1)$ find in the worst case

- Example application: network routing tables
  - [remove also takes $O(1)$ time]

**Insert** has worst-case $O(N)$ run-time

Keep two hash tables, and use two different hash functions.
Graphs ADT

Jasleen Kaur

Spring 2014
Graph Representation: \( G = (V, E) \)

- **Adjacency Matrix**
  - A \(|V| \times |V|\) matrix – \([i,j]^{th}\) entry represents edge from \(i^{th}\) to \(j^{th}\) vertex

- **Adjacency List**
  - An array of linked lists of max length \(|V|\), with the \(i^{th}\) entry storing edges from the \(i^{th}\) vertex

Topological Ordering: Properties

- **Definition**: an ordering of vertices such that
  - if there is a path from \(i\) to \(j\), then \(i\) appears before \(j\) in the ordering

- **Not possible if graph has a cycle**
  - For any two vertices \(v\) and \(w\) on the cycle, \(v\) precedes \(w\) and \(w\) precedes \(v\)

- **Ordering need not be unique**
  - Any legal ordering will do
  - e.g.: \(v1, v2, v5, v4, v3, v7, v6\)
  - e.g.: \(v1, v2, v5, v4, v7, v3, v6\)
Improvement

- If graph is sparse, only a few vertices have indegrees updated in each iteration
  - However, search for 0-degree vertex scans all vertices
- How to do better with adjacency list representation?
  - “Box” a node when its indegree is updated to zero
  - 0-degree vertices found in box in subsequent iterations
  - If “box” is a stack, insertion/deletion takes constant time!

Improvement: Analysis

```
for (int k=0; k<|V|; k++) indegree[k] = 0;
for each vertex i
    for each edge (i,j), indegree[j] += 1;
Stack S = new Stack();
for each vertex k
    if (indegree[k] == 0) S.push(k);
for (int k=0; k<|V|; k++)
    i = S.pop();
    for each edge (i,j),
        indegree[j] -= 1;
        if (indegree[j] == 0) S.push(j); }
```

- Adjacency Lists:
  - $O(|V|+|E|)$ to compute the indegrees
  - $O(|V|)$ to initially place vertices in S
  - In each iteration, $O(1)$ to find node $i$ – total: $O(|V|)$
  - Across all iterations, $O(|E|)$ to update the indegrees (and push to S)
  - Total run-time: $O(|V| + |E| + |V| + |V| + |E|) = O(|V| + |E|)$
- This is the best that can be done (visit each node and edge once)!
Shortest-path problems on graphs

Input: a weighted graph $G = (V,E)$, with cost function $c$

**Single-pair SP**
For specified vertices $s$ and $t$, determine the shortest path – the path of minimum total cost – from $s$ to $t$

**Single-source SP**
For a specified vertex $s$, determine shortest paths from $s$ to all vertices

**All Pairs SP**
Determine shortest paths between each pair of vertices

Example application: what the airlines/navigation services pre-compute

We'll study the Warshall-Floyd Algorithm for the All-Pairs SP (APSP) problem

---

Do negative-cost edges matter?

If there is a cycle with negative cost, shortest path may not be defined

Otherwise, the problem is well-defined
**Input:** the graph in adjacency-matrix form: \( \text{adj}[^{|V|}x|^{|V|}] \)

**Output:** Two matrices

\( \text{dist}[^{|V|}x|^{|V|}] \): \( \text{dist}[i][j] \) is the length of the SP from \( v_i \) to \( v_j \)

\( \text{path}[^{|V|}x|^{|V|}] \): \( \text{path}[i][j] \) is some vertex on the SP from \( v_i \) to \( v_j \)
APSP: The Warshall-Floyd algorithm

Input: the graph in adjacency-matrix form: \( \text{adj}[|V|\times|V|] \)

Output: Two matrices
- \( \text{dist}[|V|\times|V|] \): \( \text{dist}[i][j] \) is the length of the SP from \( v_i \) to \( v_j \)
- \( \text{path}[|V|\times|V|] \): \( \text{path}[i][j] \) is some vertex on the SP from \( v_i \) to \( v_j \)

Big-idea definition: \( \text{dist}^{(k)}[i][j] \): length of SP from \( v_i \) to \( v_j \) that only uses \( \{v_1, v_2, ..., v_k\} \) as intermediate vertices

\[
\begin{align*}
\text{dist}^{(0)}[i][j] &= \text{adj}[i][j], \quad \text{we seek dist}^{(|V|)}[i][j] \\
\text{dist}^{(k)}[i][j] &= \\
&\min \left\{ \text{dist}^{(k-1)}[i][j], \text{dist}^{(k-1)}[i][k] + \text{dist}^{(k-1)}[k][j] \right\}
\end{align*}
\]

APSP: The Warshall-Floyd algorithm

```
public static void apsp(int[][] adj, int[][] dist, int[][] path){
    //Initialization
    int n = adj.length; //Notation for |V|
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++){
            dist[i][j] = adj[i][j];
            path[i][j] = i; //A "default" value
        }
    for (int k=0; k<n; k++) //Considering the k’th vertex:
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    path[i][j] = k;
                }
}
```