CS 304 Final Review

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

Northeastern Illinois University

Chicago, Illinois, 60625

1 December 2016

Exam format

- 15 True/False (15%)
- 20 multiple choices (50%)
- 3 short answers (16%)
- 2 coding questions (19%)

Topics

- Topics from the midterm (Chapters 1-5, about 30%)
  - Focusing on recursion
- The `List` ADT (Chapters 6 and 7)
- Binary search trees (Chapter 8)
- Priority queues, heaps, and graphs (Chapter 9)
- Sorting and searching algorithms (Chapter 10)

Chapter 1

- Classes and objects
  - An object is an instantiation of a class.
  - A class defines the structure of its objects.
  - Class inheritance - classes are organized in an "is-a" hierarchy.
  - Classes implement interfaces - all methods must be completed.
- Time complexity
  - How does the time taken depend on the input size?
  - Order of complexity: big O notation.
    - Calculate the number of operations performed.
    - Take only the fastest-growing term (highest exponent).
    - Remove any constant factors.
    - $3n^2 + 6n + 100$ becomes O(n^2).
    - O(1) < O(log$n$) < O(n) < O($n$log$n$) < O(n^2) < O(n^3) < O(2^n)

Chapter 2

- Abstract data type (ADT)
  - A data type whose properties (domain and operations) are specified independently of any particular implementation.
- The `StringLog` ADT
  - `StringLog` methods - constructors, `insert`, `clear`, `isFull`, `size`, `toString`, `contains`
  - Array-based vs linked-based implementations
  - Bounded and unbounded

Chapter 3

- Stack
  - Definition, LIFO, can only access from the top of the stack.
  - Stack methods: **`push`**, **`pop`**, **`top`**, **`isEmpty`**, **`size`**, (**`isFull`**)
  - Time complexities
  - Array-based vs linked-based implementations
  - Bounded vs unbounded
- Exceptional cases in the **`Stack`** ADT
  - **`pop`**, **`top`** an empty stack: throw a **`StackUnderflowException`**
  - **`push`** onto a full stack: throw a **`StackOverflowException`**
- Postfix expressions: `1 2 * 4 + 3 /`

Chapter 4

- Recursion: a method that calls itself.
- Two components of a recursive method:
  - There must be special cases to handle the simplest tasks directly so that the function will stop calling itself. (Base case)
  - Every recursive call must simplify the task in some way. (Recursive call)
- Avoiding infinite recursion:
  - Must check the base case first.
  - No recursion in the base case!
  - And the recursive call(s) must bring you closer to the base case.
- Mutual recursion: `A` calls `B`, `B` calls `C`, `C` calls `A`.
- Dynamic storage allocation is used for supporting recursive calls.

Chapter 5

- Queue - A structure in which elements are added to the rear and removed from the front; a "first in, first out" (FIFO) structure.
  - Queue methods: **`enqueue`**, **`dequeue`**, **`isEmpty`**, **`size`**, (**`isFull`**, **`enlarge`**)
  - Array-based vs linked-based implementations
  - Bounded vs unbounded
- Exceptional cases in the **`Queue`** ADT
  - **`dequeue`** an empty queue: throw a **`QueueUnderflowException`**
  - **`enqueue`** onto a full queue: throw a **`QueueOverflowException`**

Chapter 6

- Object comparisons
  - What does the "==" operator compare? Reference or value?
- Linked lists operations
  - Creating a linked list
  - Creating a node
  - Inserting and deleting a node
  - Traversing a linked list
  - Time complexities
- By copy (by value) vs by reference
  - Storing the copy of the object or the reference to the object.
  - When to choose "by copy" and "by reference"?
  - Memory and time considerations, whether the changes need to be left on the objects, etc.

Chapter 6

- Linked lists implementations
  - Array-based implementation
    - Insertion for sorted lists: need to shift elements after the insertion position one slot toward the end.
    - Deletion for sorted lists: need to shift elements after the deletion position one slot toward the front.
    - Insertion for unsorted lists: can simply insert the element to the first empty slot in the list. No shift is needed.
    - Deletion for unsorted lists: use the last element to replace the deleted element. No shift is needed.

Chapter 6

- Linked lists implementation
  - Reference-based implementation
    - Insertion for unsorted lists: can simply insert the element to the front of the list.
    - Insertion and deletion for sorted lists: need to update references of the related nodes.
    - Each node in a singly linked list has two fields: **`data`** and **`link`** (a reference to the next node).
- Adding an element to a reference-based sorted list requires three steps:
  1. Find the location where the new element belongs;
  2. Create a node for the new element;
  3. Correctly link the new node into the identified location.

Chapter 7

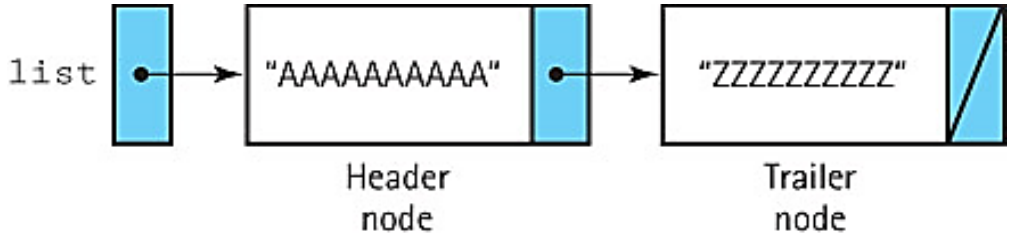
- Circular linked lists
  - A list in which every node has a successor; the "last" element is succeeded by the "first" element.
  - Insertion and deletion - don't forget the last node in the list!
- Doubly linked lists
  - A linked list in which each node is linked to both its successor (`link`) and its predecessor (`back`).
  - Has more references to update when inserting or deleting an element.
- Linked lists with headers and trailers
  - Header node - A placeholder node at the beginning of a list; used to simplify list processing.
  - Trailer node - A placeholder node at the end of a list; used to simplify list processing.

Chapter 7

Singly linked list

Doubly linked list

Linked list with header and trailer



Chapter 8

- Binary search trees (BSTs)
  - A binary tree is built out of nodes; each has a value and two links (references): left child and right child.
  - The tree itself has a reference to the root node.
  - Order property
    - For every node: all left descendants have a smaller value,
    - and all right descendants have a larger value.
  - To search for a value, start from the root.
    - If the current node's value is correct, return **`true`**.
    - Otherwise recursively search the left or right subtree (just one!)
    - Base case: if the tree is empty (**`null`**), return **`false`**.
    - Kind of like binary search.

Chapter 8

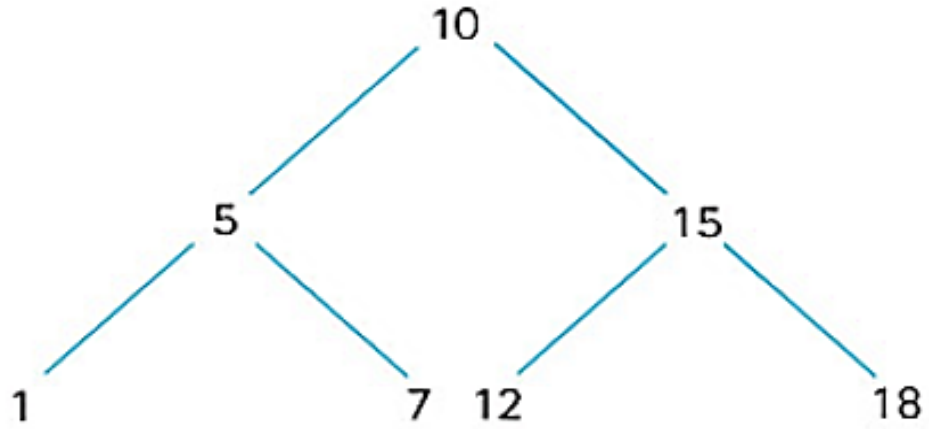
- BST operations
  - Insertion is similar to search.
    - When we reach a `null` reference, replace it with the new node.
  - For deletion, we first do a search to find the node. Then, three cases:
    - No children: replace parent's `link` with `null`
    - One child: replace parent's `link` with the child.
    - Two children: Find the previous node (rightmost descendant of left child) or the next node (leftmost descendant of right child). Swap its value with this node's, then recursively remove that previous or next node.

Chapter 8

- BST operations
  - Complexity: Searching, insertion, and deletion are O(h) where h is the height.
    - Balanced if each node has about as many left descendants as right.
    - In a balanced tree, O(h) is O($\log n$).
    - In an unbalanced tree, it could be as bad as O(n).
    - Building a balanced BST is O($n \log n$).
    - Building an unbalanced BST could be as bad as O(n^2).

Chapter 8

- BST implementation
  - Iterative vs recursive
  - Methods - **`size`**, **`contains`**, **`add`**, **`remove`**
- BST traversal



  - In-order: 1, 5, 7, 10, 12, 15, 18
  - Pre-order: 10, 5, 1, 7, 15, 12, 18
  - Post-order: 1, 7, 5, 12, 18, 15, 10

Chapter 8

- Balancing BSTs
  - Why is it needed?
  - How to balance a binary search tree?
- Full binary trees and complete binary trees
  - A full binary tree is a binary tree in which all of the leaves are on the same level and every nonleaf node has two children.
  - A complete binary tree is a binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible.

Chapter 9

- Priority queues
  - A priority queue is an abstract data type in which only the highest-priority element can be accessed.
- Priority queue implementations
  - An unsorted list - dequeuing would require searching through the entire list.
  - An array-based sorted list - Enqueuing is expensive.
  - A reference-based sorted list - Enqueuing again is O(n).
  - A binary search tree - On average, O($\log n$) steps for both **`enqueue`** and **`dequeue`**.
  - A heap - guarantees O($\log n$) steps, even in the worst case.

Chapter 9

- Heaps
  - An implementation of a priority queue based on a complete binary tree which satisfies two properties:
    - The shape property: the tree must be a complete binary tree.
    - The order property: for every node in the tree, the value stored in that node is greater than or equal to the value in each of its children.
  - Heap methods - **`enqueue`**, **`dequeue`**, **`reheapDown`**, **`reheapUp`**
  - Time complexities

Chapter 9

- Storing a heap in an array
  - In a heap, the following relationships hold for an element at position **`index`**:
    - If the element is not the root, its parent is at position **`(index - 1) / 2`**.
    - If the element has a left child, the child is at position **`(index * 2) + 1`**.
    - If the element has a right child, the child is at position **`(index * 2) + 2`**.
- Examining if the tree stored in an array is a heap.

Chapter 9

- Graphs
  - A data structure that consists of a set of nodes and a set of edges that relate the nodes to each other.
  - $G = (V, E)$ where $V(G)$ is a finite, nonempty set of vertices and $E(G)$ is a set of edges (written as pairs of vertices).
  - Vertex, edge, adjacent vertices, vertex degree
  - Undirected graph and directed graph (digraph)
  - Complete graph and weighted graph
  - A tree is a graph, but a graph is not necessarily a tree.

Chapter 9

- Graphs
  - With the array-based implementation, a graph consists of
    - an integer variable **`numVertices`**,
    - a one-dimensional array **`vertices`**,
    - a two-dimensional array **`edges`** (the adjacency matrix).
  - With the link-based implementation, a graph is represented by adjacency lists.
    - An adjacency list is a linked list that identifies all the vertices to which a particular vertex is connected; each vertex has its own adjacency list.
    - Two alternate approaches:
      - Use an array of vertices that each contains a reference to a linked list of nodes.
      - Use a linked list of vertices that each contains a reference to a linked list of nodes.

Chapter 9

- Graph traversal
  - Depth-first strategy (DFS) - The traversal goes down a branch to its deepest point and moving up. Uses stack as the fundamental data structure.
  - Breadth-first strategy (BFS) - The traversal visits each vertex on level 0 (the root), then each vertex on level 1, then each vertex on level 2, and so on. Uses queue as the fundamental data structure.

Chapter 10

- Selection sort
  - Find the smallest element, put it in the right place.
  - Find the smallest remaining element, put it in the right place.
  - Repeat until everything is in place.
  - Time complexity: O(n^2)
- Bubble sort
  - **`BubbleUp`** the smallest element to the first slot in the array.
  - **`BubbleUp`** the smallest remaining element to the second slot in the array.
  - Repeat until everything is in place.
  - **`BubbleUp`** operation changes the locations of other elements in the array.
  - Time complexity: O(n^2)

Chapter 10

- Insertion sort
  - An array is divided into a sorted part and an unsorted part.
  - The insertion sort algorithm moves elements one at a time from the unsorted part into the correct position of the sorted part.
  - This process continues until all the elements have been sorted.
  - Time complexity: O(n^2) for the general case, O(n) for the best case where all the elements are already sorted in the correct order. The worst case scenario is when the elements in the array are in reverse order.

Chapter 10

- Merge sort
  - Divide the unsorted array into n subarrays, each containing 1 element (remember that an array of 1 element is considered sorted).
  - Repeatedly merge subarrays to produce new subarrays until there is only 1 subarray remaining. This will be the sorted array.
  - Merge sort is a recursive algorithm:
    - The base case is an array of only one element.
    - At the recursive step, the merge sort is applied to each half problem, recursively sorting each half and putting them back together.
  - Requires an auxiliary array.
  - Time complexity: O($n\log n$)

Chapter 10

- Quick sort
  - A divide-and-conquer algorithm and inherently recursive.
  - At each stage the part of the array being sorted is divided into two subarrays, with everything in the left subarray less than or equal to the split value (pivot) and everything in the right subarray greater than the split value.
    - The same approach is used to sort each of the smaller subarrays (a smaller case).
    - This process goes on until the small subarrays do not need to be further divided (the base case).
  - Time complexity: O($n\log n$) if the splits divide the segment of the array approximately in half. It could be as bad as O(n^2) if the splits are very lopsided.

Chapter 10

- Heap sort
  - Building a heap for the unsorted sequence.
  - Take the root (maximum) element off the heap, and put it into its place.
  - **`reheap`** the remaining elements. (This puts the next-largest element into the root position.)
  - Repeat until there are no more elements.
  - Time complexity: O($n\log n$)
- Stability of a sorting algorithm
  - Stable sort: A sorting algorithm that preserves the order of duplicates.
  - **`quickSort`** and **`heapSort`** are inherently unstable.

Chapter 10

- Linear search
  - A linear search examines all values in a sequence until it finds a match or reaches the end.
  - Time complexity: O(n)
- Binary search
  - Binary search cuts the search in half each time. We do not visit every element.
  - This is only possible when the values in the array are already sorted.
  - Time complexity: O($\log n$)

Chapter 10

- Hashing
  - The technique for ordering and accessing elements in a list in a relatively constant amount of time by manipulating the key to identify its location in the list.
    - Hash function, hash table
    - Collision, linear probing
    - Bucket and chaining strategies