**Exam 2 Study Guide**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | General |

|  |  |
| --- | --- |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  | Date:  **11/06/2019** (Wednesday, 2nd half of class meeting for Sec. 004). |

|  |  |  |
| --- | --- | --- |
|  |  | Duration: 80 minutes. |

|  |  |
| --- | --- |
|  | Questions will be based on the topics covered since Exam 1, ending with the **10/30/19 lecture** for **Sec. 004**, and the **11/04/19 lecture** for **Sec. 002**): |

|  |  |  |
| --- | --- | --- |
|  |  | Introduction to algorithm analysis (continued). |

|  |  |  |
| --- | --- | --- |
|  |  | *Linked list* |

|  |  |  |
| --- | --- | --- |
|  |  | *Stacks*/*queues* (and applications). |

|  |  |  |
| --- | --- | --- |
|  |  | *Recursion* |

|  |  |  |
| --- | --- | --- |
|  |  | *Trees: fundamentals*,*traversals*,*binary search trees*, . . . (covering only aspects that have been discussed) |

|  |  |
| --- | --- |
|  | Test will be *closed books* and *closed notes*. |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | Relevant Material |

|  |  |
| --- | --- |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  | Most relevant *Lecture Notes*: ***309OrderAnalysisOfAlgorithms01*** through ***310s01SupplementaryNotesOnBigO*** and  ***312LinkedListConceptualIntro*** through ***320BinarySearchTrees***. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | (Focus on topics/material we spent much time on - diligence in attending classes is significant here.) |

|  |  |  |
| --- | --- | --- |
|  |  | *Handouts* (distributed in class) and *examples* (significant number of them posted under **Examples**) covering topics indicated above. |

|  |  |  |
| --- | --- | --- |
|  |  | *Assignments* 4, 5 and 6. |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | Other Resources |

|  |  |
| --- | --- |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  | You may want to check out ***sample past test/exam questions*** already posted on the class homepage. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | You should ***not*** however, expect the questions to be identical in number, kind, topic coverage, *etc.* |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *You should not have to worry about questions being written on topics we have not yet covered; some such questions may appear as sample past questions because the associated topics were appropriate at that time.* |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  | Checklist (exhaustiveness ***not*** guaranteed) of some things you are expected to know and/or know how to do/apply: |

|  |  |
| --- | --- |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Big-O characterization and notation. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Rigorous example characterization/analysis involving O(1), O(n) and O(n2) algorithms for solving an example problem. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Common categories and their growth-rate behavior and relative ordering: O(1), O(log n), O(n), O(n log n), O(n2). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Be able to quickly inspect a code segment (involving the basic flow-of-control constructs - sequence, selection, repetition) and characterize it (as "tightly" as can be determined). |

|  |  |  |
| --- | --- | --- |
|  |  | Linked list. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Array versus linked list: strengths and weaknesses. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *Random access*, *insertion*/*deletion anomaly*, *resizing woe*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | When to use which to take advantage of the strength(s) and minimize/avoid the weakness(es). |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | (There's no one "panacean data structure" -> use the right tool for the right problem.) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Be able to design/implement functions that manipulate linked list(s). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Manipulate: add, delete, modify, search, inspect, ... (may be in combination within a function). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Common coding idioms/"patterns": what they mean and when to use. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **cursor = cursor->link;** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **while (cursor != 0) { ... }** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **while (cursor->link != 0) { ... }** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | ... |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | How head pointer(s) should be passed to a function: *by value* or *by reference*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Be cognizant of and know how to avoid *null-pointer exception*. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | When writing code that dereferences a pointer (at some point), always check that the pointer will never contain the *null address* (at that point). |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Be careful when writing relational expression involving *short-circuit evaluation*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Know when to say no: |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Don't use memory that's not allocated. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Don't access memory that's already deallocated (if that memory access must be done, do it *before* deallocation). |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Don't leak away memory. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Be able to read/understand C++ code that manipulates linked list(s) and *identify any bugs* (and associated problems that can arise from them). |

|  |  |  |
| --- | --- | --- |
|  |  | Stacks/queues and applications. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Containers restricted in specific ways (especially wrt addition and removal of data items) to support commonly required operational characteristics. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | LIFO with stacks and FIFO with queues |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Fundamental operations (besides construction and destruction), STL-style. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Stacks:**push**,**pop**,**top**,**empty**,**size**. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **top + pop**for *traditional pop*. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Queues:**push**,**pop**,**front**,**empty**,**size**. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **push**for *traditional enqueue*. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **front + pop**for *traditional dequeue*. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Error conditions: *underflow* (always possible) and *overflow* (implementation and system resource dependent). |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Keys reasons for their usefulness: LIFO and/or FIFO buffering, reversal and echoing capabilities. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Applications (all related to LIFO and/or FIFO buffering). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Reversal and echoing. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | How to put these effects to use (expressing algorithms involved in *pseudocode*). |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | For examples on how to clearly write pseudocode involving stacks and queues, see "***STL\_Stack\_Queue\_Example***", "***Implementing Queue Using 2 Stacks***" and "***StackQueueAppEg02\_LevelTravOfLLofLL\_Pseudocode***" posted under **Examples**. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Invocation (function-calling) flow of control support/management: system/call/run-time stack. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | (in more depth when covering recursion) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *Level* (*breadth-first*) *traversal* of non-linear data structures. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Concept of *priority queue*. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | STL in perspective. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | How to use STL stack and queue templated containers (**chk\_pal**). |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Implementations using array and linked list. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Using either to implement *stack* is pretty straightforward and mundane - stack has "only 1 door". |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Using either to implement *queue* is more complicated and interesting - queue has "2 doors": |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | *Circular array*. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | **size\_type next\_index(size\_type current\_index, size\_type capacity)** **{ return (current\_index + 1) % capacity; }** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | *Circular linked list*. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | Where should *front* and *rear* be and *why* (especially in regard to*push* and *pop*)? |

|  |  |  |
| --- | --- | --- |
|  |  | Recursion. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Recursive thinking |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Dealing with the seemingly infinite in finite fashion |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | "Divide-and-conquer" + "results of division are identical and smaller versions of the original (what gets divided)" |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | 4 criteria for successful application: recursively decomposible, base case(s), making progress, ultimate reachability of base case(s) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | How function-calling (recursive functions included) is typically implemented with the help of system/call/run-time stack ... |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *Activation records* (*stack frames*) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | and how to use that to trace recursive functions |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Design/implement recursive algorithms for given problems (including problems that involve arrays, linked lists and trees) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Most important first hurdle - express/formulate problem in terms of smaller problems of the same type |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Tail recursion |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Indirect recursion |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Advantages/disadvantages |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Conceptual elegance/clarity and code compactness |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Function-call overhead and stack-overflow risk |

|  |  |  |
| --- | --- | --- |
|  |  | Trees |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | General properties of trees and specific properties of special trees covered |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Tree versus graph |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | *n-ary tree* -> binary tree -> binary search tree and heap |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Most (if not all) "traits" related to a tree must be *recursively applied* (for certain terms such as *balanced*, *binary search tree*, *in-order traversal*, *etc.* to be true) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Representation of binary tree |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Pointer-based representation |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Efficient representation of *complete* binary tree using array |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | *Traversal* (and processing) of trees |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Breadth-first (level): recall the usefulness of queue |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | How algorithm for "breadth-first traversal of linked list of linked lists" is adapted. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Depth-first: *pre-order*, *in-order* and *post-order* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | (Relating to *prefix*, *infix* and *postfix* expressions can be helpful: what replaces "where, relative to the *operands*, a binary *operator* is placed".) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Most tree manipulations would be very difficult (at best) for us to track if we don't do it *recursively* |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Binary search trees (BST) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | What is and why |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Binary tree with storage rule (invariant) to enable binary search and ordered access |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | Which traversal algorithm leads to which kind of ordering |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Insert, search and remove operations |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Focus on the conceptual |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | (Simple algorithms to reach the "low-end" and "high-end" nodes.) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Performance aspects |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Is searching always guaranteed to be logarithmic? If not, how can logarithmic behavior be attained? |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | (What would a BST degenerate into in the worst case?) |