Exam study sheet for CS2711

Here is the list of topics you need to know for the midterm. For each data structure listed below, make sure you can do the following:

1. Give an example of this data structure (e.g., draw a binary search tree on 5 nodes).
2. Know the basic properties of data structures (e.g., that a heap has height $\log n$) and be able to prove them by induction.
3. Know which basic operations the data structure supports and be able to show them on an example (e.g., insertion into an AVL tree).
4. Know what is the time complexity of its basic operations and how they compare between different structures (e.g., searching in an AVL tree vs. searching in a linked list).
5. For each algorithm, make sure you know its time complexity, can write pseudocode for it and can show its execution on an example.

For the analysis of algorithms (chapter 4), you need to know time complexity and induction, and be able to solve problems similar to ones in labs and assignments. Also, have a basic understanding of amortized analysis (for splay trees and doubling array size) and probabilistic analysis (hash tables, skip lists, quicksort).

List of topics

1. **Basic data structures (chapters 3 and 6)**: Linked list, doubly linked list, array, extendable array. Know why doubling the size is a better way to grow an array.
2. **Stacks and queues (chapter 5)**: Know stack and queues data types (push/pop, enqueue/dequeue). Be able to do parentheses/tag matching using stacks.
3. **Trees (chapter 7)**: Know both general trees and binary trees. Terminology like root/parent/child, height (remember that height of the root is 0), depth/size, leaf/internal node, proper binary tree, complete tree. Traversals (preorder, inorder, postorder). Evaluating arithmetic expressions using postorder traversal, drawing trees using inorder traversal. Euler tour of a tree. Linked-list and array representations of trees.

   Know the relationships among tree height, the number of nodes, number of edges, number of leaves, etc and be able to prove them (e.g, by induction).
4. **Heaps (chapter 8)**: Know the definition of a heap and a priority queue, both bottom-up and top-down heap construction. Using arrays to store heaps. Heapsort.
5. **Hash tables (chapter 9)**: Know the definition of a hash table. Collision-handling schemes: chaining, linear probing, quadratic probing, double hashing. Be able to give an example of a hash function and a hash table, and state which properties a good hash function should have (output "looks random", not likely to get a collision). Know polynomial hash codes as well as $h(x) = ax + b \mod n$ function class.
6. **Skip lists (chapter 9)**: Know the definition and performance of a skip list. Have an intuition why skip list is efficient on average, why it has on average $\log n$ layers. Know which additional operations (successor, minimum) a skip list implements efficiently.
7. **Search trees (chapter 10)**: Know the definitions and properties of a generic binary search tree, as well as AVL tree, splay tree, (2,4) tree and red-black tree. Know the relationship between (2,4) trees and red-black trees (but I will not ask much more about (2,4) and red-black trees, so concentrate on AVL and splay trees). Be able to do AVL tree rotations (insert/delete) and the "splaying" operation.
8. **Sorting (chapter 11)**: For all sorting algorithms, know how they work (i.e., be able to show on a given input how they work), and their running time (whether average or worst-case and examples for worst/best cases). Know when you would use each of them (i.e., conditions when you would choose radix-sort, the fact that quicksort is in-place, etc). Know the lower bound for sorting. Know how to adapt quicksort to do selection.

Post-midterm topics

9. **Disjoint sets (chapter 11)**: Know that it has Makeset, Union and Find operations, that it can be implemented using linked lists and that sets are used in Kruskal's algorithm, clustering, etc.

10. **Text processing (chapter 12)**: Know Boyer-Moore and Knuth-Morris-Pratt algorithms, as described in the textbook. In particular, know which precomputation each of them does. For the suffix tries, know what they are and how they are used for predicting words. Know the running times. Huffman codes: know what they are, be able to read and write them, know that that's a greedy algorithm.

11. **Algorithm design (chapter 12, notes)**: Greedy algorithms and dynamic programming were the main design paradigms we looked at (we also talked about backtracking, but just as a case when dynamic programming does not work). Know what properties of a problem you need to apply these paradigms (locally optimal choice is globally optimal for greedy, optimal subproblems property for dynamic). Know examples of algorithms for both of them (Kruskal, Huffman, Dijkstra are greedy, Floyd-Warshall, LCS are dynamic programming). Be able to design simple greedy algorithms, like the first one in the assignment and ones in the lab.

    Know scheduling algorithms (notes) and be able to say in which case which algorithm works (e.g., that dynamic programming is not efficient when deadlines are very large, and greedy works when there are no durations), and what you'd do in that case (backtracking). Know Knapsack problem and its variants (Fractional Knapsack can be solved by greedy, but Simple Knapsack is already NP-complete and can be reduced to scheduling, but can be approximated to within the factor of 2 by a greedy algorithm).

12. **Graphs (chapter 13)**: Adjacency list vs. adjacency matrix representation, directed graphs, weighted graphs. Graph terminology (path, cycle, DAG, spanning tree...)

    For all graph algorithms, know their name (in particular, know what Floyd-Warshall,Dijkstra, Kruskal, Prim-Jarnik and Bellman-Ford algorithms do), the problem they solve (e.g., all-pair shortest path), running time and restrictions (e.g., no negative-weight cycles).

    - Graph traversals: Depth First Search (DFS) / Breadth First Search (BFS). Running time, which problems they solve (e.g., BFS solves single-source shortest path on unweighted graphs, both can detect cycles, etc). Be able to produce a BFS/DFS tree for a given (possibly directed) graph, showing back/cross edges. Know topological sort and detecting strongly connected components. Time O(n+m).
    - Single-source shortest paths: BFS, Dijkstra's algorithm, Bellmann-Ford (see slides for the latter). Know that Dijkstra's algorithm is greedy, requires positive weights on edges and uses a priority queue as an auxiliary data structure. Dijkstra: $O((n+m)\log n)$ or $O(n^2)$, Bellman-Ford $O(nm)$. Know that Bellman-Ford can work with negative edges present and can detect negative cycles. Know how on DAGs there is a simple algorithm using topological sort that works even with negative edges, and that it can be used to find the longest path in a DAG (but in general finding longest path is not known to be efficiently solvable).
    - All-pairs shortest paths: definition of transitive closure, computing transitive closure using matrix multiplication. Floyd-Warshall algorithm for all-pairs shortest paths $O(n^3)$ – think of just the weighted case.
    - Minimum spanning trees: Kruskal's algorithm and Prim-Jarnik's algorithm. Both are greedy; Kruskal's uses sets, Prim-Jarnik' uses a priority queue. Both run in $O((m+n)\log n)$. Borůvka's algorithm is in-between: it keeps clusters, but then for every cluster finds a best edge out of it. Kruskal's and Borůvka's algorithm can be converted to work on many-processor architecture. A variation of Kruskal's can be used for clustering.