Midterm 2 Review Topics

Data Structures

You must justify all your answers to recieve full credit

Midterm 2 has 4 theory questions and 1 short C++ program. Total time is 90 minutes.

1. **Write algorithms with List, Stack or Queue ADTs**
   - 1.A. Given a list/stack/queue algorithm pseudocode, find its time complexity.
   - 1.B. Given an algorithm pseudocode, draw the list state at a certain moment.
   - 1.C. Given a problem description, implement the algorithm at ADT Level to implement it.
   - 1.D. Write algorithms and estimate the time complexity of algorithms processing expressions.
2. **Use properties of rooted, ordered trees, traverse their nodes**
   - 2.A. Given some tree properties and element counts, calculate or estimate other counts.
   - 2.B. Use Tree ADT to implement some algorithms to manipulate trees.
   - 2.C. Use tree traversal to solve related algorithmic problems.
   - 2.D. Estimate the time complexity of a tree operation given input data distribution.
3. **Manipulate binary search trees (BST)**
   - 3.A. Perform insert and delete operations in arbitrary binary search tree.
   - 3.B. Verify some properties of binary search trees assuming their element counts.
   - 3.C. Use binary trees to encode another structure such as a multiway tree.
   - 3.D. Build AVL trees, perform rotations, run insert and delete operations.
4. **Use and analyze priority queues and heaps**
   - 4.A. Use priority queue ADT to implement and analyze simple algorithms.
   - 4.B. Store binary trees into arrays.
   - 4.C. Perform and analyze heap operations for insert and delete.
   - 4.D. Use and analyze Heapsort.
5. **Use and analyze sorting algorithms**
   - 5.A. Use and analyze Selection sort, Insertion sort, Bubble sort algorithms.
   - 5.B. Use and analyze Merge sort.
   - 5.C. Use and analyze Quicksort algorithms.
   - 5.D. Use and analyze Radix sort and Counting sort.
6. **(C++ code) Use STL structures, implement custom inheritance and polymorphism**
   - 6.A. (C++ code) Use STL classes for lists, stacks, queues with iterators.
   - 6.B. (C++ code) Use STL classes for tree-related operations.
   - 6.C. (C++ code) Use STL classes for priority queue operations.
   - 6.D. (C++ code) Use inheritance and virtual functions.
   - 6.E. (C++ code) Use polymorphism and template classes or functions.

1 Write algorithms with List, Stack or Queue ADTs

1.A. **Given a list/stack/queue algorithm pseudocode, find its time complexity.** A small C++ program or a pseudocode is given to solve some specific task (filter list elements by predicate, reverse a list, return a new list containing elements with even numbers only, etc.) Verify, if this code works in all cases and find its time complexity in terms of input length n, if the list/vector/stack/queue are implemented in a certain way.

1.B. **Given an algorithm pseudocode, draw the list state at a certain moment.** A small C++ program or a pseudocode is given to solve some specific task. At some point in the algorithm (when a condition succeeds or during the k-th iteration of a loop etc.) show the state of the data structures involved.

1.C. **Given a problem description, implement the algorithm at ADT Level to implement it.** There is one or more lists/stacks/queues and an algorithmic task to process them. Implement this task as a small C++ program or as a pseudocode.

1.D. **Write algorithms and estimate the time complexity of algorithms processing expressions.** An expression with numbers (or Boolean values) and some operators is given. Transform it into a prefix (or infix, or postfix) notation or perform some calculations in this expression.

2 Use properties of rooted, ordered trees, traverse their nodes

2.A. **Given some tree properties and element counts, calculate or estimate other counts.** Given some tree properties or assumptions (is it a full n-ary tree, a complete tree, a perfect tree) and a number of elements (total nodes, internal nodes, leaves, height, min depth of a leave, average number of children), estimate some other counts.

2.B. **Use Tree ADT to implement some algorithms to manipulate trees.** Use Tree ADT functions to perform some editing algorithms on trees; implement them as small C++ programs or pseudocode.

2.C. **Use tree traversal to solve related algorithmic problems.** Do some variant of tree traversal (BFS or DFS - in particular, preorder, inorder, postorder), to do some aggregation or mass edit.

2.D. **Estimate the time complexity of a tree operation given input data distribution.** Calculate the expected value of running time (or number of certain operations), given the tree structure and probabilistic distribution of inputs.

3 Manipulate binary search trees (BST)

3.A. **Perform insert and delete operations in arbitrary binary search tree.** Given a binary search tree (possibly not complete or not balanced), perform operations to preserve the order relation of the tree.

3.B. **Verify some properties of binary search trees assuming their element counts.** Given assumptions about the BST element counts, check some statement about the BST.

3.C. **Use binary trees to encode another structure such as a multiway tree.** Encode or decode multiway trees, parenthesized expressions and similar structures into BSTs.

3.D. **Build AVL trees, perform rotations, run insert and delete operations.** Verify AVL tree properties, check what happens to the properties as we rotate, insert or delete nodes.

4 Use and analyze priority queues and heaps

4.A. **Use priority queue ADT to implement sorting and similar tasks.** Write an algorithm to find the second largest element in array, to find the median, etc. Use the ADT of Priority queues – and estimate the time complexity of your algorithm.

4.B. **Store binary trees into arrays.** Store complete (or close to complete) binary trees into arrays – either 0-based or 1-based. Convert trees into arrays and back. Write expressions on array indices to locate elements in the binary tree (parents, grandparents, children, siblings, etc.)

4.C. **Perform and analyze heap operations for insert and delete.** Run several heap operations and estimate how many comparisons and swaps you may need to run these operations.

4.D. **Use and analyze Heapsort.** Analyze an optimal `buildHeap()` operation which proceeds from the bottom up. For example, estimate how many swaps are necessary to add a single element at a certain level of the heap. Estimate the total number of comparisons to run heapsort.

5 Use and analyze sorting algorithms

5.A. **Use and analyze Selection sort, Insertion sort, Bubble sort algorithms.** Run Selection (Insertion and Bubble) sorting algorithms on certain initial arrays, for example those that are nearly sorted. Also trace the swaps of individual numbers in sortable arrays.

5.B. **Use and analyze Merge sort.** Estimate the runtime of divide and conquer algorithms (Merge sort or variants of it). Count the full number of comparisons as you run mergesort.

5.C. **Use and analyze Quicksort algorithms.** Run one or more iterations of Quicksort – and draw the intermediate states of an array.

5.D. **Use and analyze Radix sort and Counting sort.** Run Radix sort or Counting sort as pseudocode, draw some intermediate results created by these algorithms.

6 (C++ code) Use STL structures, implement custom inheritance and polymorphism

6.A. **(C++ code) Use STL classes for lists, stacks, queues with iterators.** Implement a simple algorithm using `std::list`, `std::stack`, `std::queue`, `std::vector`.

6.B. **(C++ code) Use STL classes for tree-related operations.** Implement a simple algorithm using `std::set` or `std::map` data structures – possibly with a custom comparator.

6.C. **(C++ code) Use STL classes for priority queue operations.** Implement a simple algorithm using `std::priority_queue` data structure – possibly with a custom comparator.

6.D. **(C++ code) Use inheritance and virtual functions.** Implement a class with one or more subclasses and class method overriding. Understand constructor chaining and initialization. Declare some of them virtual to ensure that the right behavior is invoked.

6.E. **(C++ code) Use polymorphism and template classes or functions.** Use STL classes parametrized with different template class types; ensure that polymorphic algorithms can run – the same algorithm and the same data structure works with different underlying types.