Main Topics

- Basics of Java
- Mathematical formalism, induction
- Asymptotic notation
- Container class ADT: Stacks and Queues, Lists,
- Container class implementations: Linked List, Double linked lists, Arrays, growing buffer, trees
- ADTS: Graphs, Trees Polynomials
- Graph traversals: Inorder, Preorder, Postorder, BFS, DFS
- Sorting
- Special Topics: Computability, Complexity, Crypto, Compiling (only tokenizing and parsing)
Basics of Java

You have had several opportunities to hone your skills as a java programmer in this course. You have gained experience in implementing classes for abstract data types (objects).

You should be more comfortable in dealing with linked structures in java now (But in the exam, pseudocode would be sufficient). This is a very important skill; Not only is useful for making efficient container classes, it is fundamental to the efficient design of large systems. It is important that you continue to develop this!

Mathematical Formalism

It is imperative that one is able to communicate effectively in this business. You must be able to follow a design specification, and likewise you should be able to write one. You should be able develop efficient and logically sound algorithms for nontrivial problems, and you should be able to argue about the correctness of your algorithms.

Often it is necessary to prove certain mathematical identities, or about mathematical structures such as graphs and trees. Induction is often helpful here. You should be comfortable with proving theorems by induction.
Asymptotic notation (Big O, Big Theta, Big Omega)

This is a robust method to quantify:
- The running time of algorithms
- The inherent difficulty of problems

You should know the definitions of these three. You should know how to use these definitions to prove $f \in O(g)$ or $f \notin O(g)$.

You should be able to analyze a algorithm and give a tight upper bound on the running time. This may require more than counting the number of nested loops.

You should be familiar with the master method and induction proofs that prove the running time.
Abstract Data Types (ADTs)

(ADTs) are a method of dealing with the complexity of large systems. The basic premise is this: recall that primitive data types

- can be stored in variables, and

- there are operations (i.e. *, +, -) that can be performed on them

ADTs are a generalization of this idea. We define

- the type of information to be stored in a class, and

- the operations that can be performed on this class.

For example, a polynomial ADT keeps a list of terms, as (power, coefficient) pairs, as the data. The operations on a polynomial are the natural ones: derivative, scalar multiplication, etc.

The running time of the individual operations depends upon the way that the data is structured. We looked at several examples of this:

<table>
<thead>
<tr>
<th></th>
<th>AdjMatrix</th>
<th>AdjList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree -</td>
<td>Linked</td>
<td>Doublylinked</td>
</tr>
<tr>
<td>Poly -</td>
<td>Array</td>
<td>Poly</td>
</tr>
<tr>
<td>Stack -</td>
<td>Array</td>
<td>Linked</td>
</tr>
<tr>
<td>Queue -</td>
<td>Array</td>
<td>Linked</td>
</tr>
</tbody>
</table>

And others. You should have an idea of how long each operation takes in each implementation (and you should know how this is derived, so that you can reason about implementations that you haven’t seen).
There is rarely a case where one implementation performs outright better than another one (otherwise, of course, the bad implementation would never be considered). So each has its share of advantages and disadvantages.

Often a good implementation that is well-suited to the problem that you are solving can make a big difference in the efficiency of algorithms that operate on these data structures.

Stacks and Queues

We saw that sometimes ADTs are used to restrict the kind of operations that you can perform. This is the case of Stacks and Queues. These behave much like a linked list, but they are restricted to be first-in, first-out (in the case of stacks) or first-in, last-out (in the case of Queues).

This is a measure for handling the complexity of writing large programs. For example, if you know that your data is going to be handled first-in first-out, you don’t have to think about the other cases.

It turns out that stack-type and queue-type access to a data structure occurs frequently. (e.g. Queues are used in on-line scheduling and BFS traversals, Stacks are used for implementing recursive procedures, compiling).
Container classes

An important class of ADTs are the container ADTs. These are ADTs whose sole purpose is to store other classes. Their data is a set of objects (sometimes of a specific type). They have operations such as insert(X), delete(X), contains(X), and maybe some others. We looked at several implementations:

- Arrays, Growing arrays
- Lists
- Binary search trees, Heaps, Hashes

For each, you should know how the data is structured, the algorithms for the operations on these structures, and the running time for each.

Here are some additional (but not complete!) notes on some of these containers:

Binary Search trees

Binary search trees organize data by putting each data element on a node in the Binary tree.

Binary search trees are used to store a set of data, where each set element conforms to a total ordering. Binary search trees have the BST property: For every node \( n \), the value at left child (if the left child exists) occurs before the value at \( n \), likewise the right child (if it exists) occurs after the value at \( n \).

Heaps

Heaps are used to implement priority queues. Heaps organize the data by putting each data element on a node in a complete binary tree. Heap-ordered trees have the heap-order property: For ever node \( n \) the value at each child occurs before the value at \( n \).
Hashes

Hash tables are used to facilitate efficient insertion/retrieval of data from the set. The strategy is to write a function that maps inputs uniformly into one of \( n \) buckets. Ideally, if each element is mapped to a unique bucket, then insert/contains/delete take \( O(1) \). But even if this is not satisfied, we divide the size of the search space by \( n \) on the average, using only a constant increase in the running time for insert/contains/delete.

Trees and Graphs

Trees and graphs are a natural way of modeling many different concepts:

- Trees e.g., model syntax trees, org charts, file directories
- Graphs, e.g., model physical locations, relations, automata,

When solving problems involving graphs and trees, you are often required to traverse the nodes. You should be familiar with the common strategy for doing this: inorder traversal, preorder traversal, postorder traversal, depth first search, breadth first search.

Special topics

Computability

An algorithm, by definition, has a finite number of instructions. So it is useful to ask what sort of problem can be solved by following an algorithm. It turns out, surprisingly, that not all problems can be solved! For example, there is no algorithm for the halting problem.

Most things that we consider to be full-fledged computers are equivalent in terms of computing power. This is because they can all simulate each other.
Complexity

There are many problems that have algorithms, but not efficient algorithms. An important class of these are the searching problems (the class NP). Informally, these are decision problems where, for any true instance one can check efficiently a polynomial length ‘proof’ of a solution.

Searching problems include Satisfiability (of Boolean formulas), Circuit satisfiability, certain scheduling problems, and the clique problem. All of these problems are in fact equivalent to each other in terms of difficulty: they are all the hardest of the searching problems (we call a problem with this property is NP-complete).

An efficient algorithm for one of these implies an efficient algorithm all NP problems. No efficient algorithm has been found for these, and it is suspected that no such algorithm exists.

Cryptography

Cryptography is the science of obfuscating messages in such a way that any person to which the message is not intended will not be able to decipher the meaning of the message, but the intended receiver will be able to determine the meaning of the message. We surveyed a number of different cryptographic schemes from the past and present. You should be familiar with the principles of modern cryptography, such as public key encryption.
Compiling strategies

Automata

Finite automata is a natural model for simple computing machines. The languages (i.e. problems) recognized by finite automata are called regular languages. Regular langages are exactly those recognized by regular expressions. You should be able to construct a finite automata for tokenizing a given language.

Context free languages

Context free languages are those languages that can be expressed by context free grammars. You should be able to construct a context free grammar to recognize a given context free language. Not all context free grammar can be parsed deterministically, but there are a number of classes of context free grammars that can. One such class is the LL(1) grammars, which are those that can be recognized by a recursive-descent parsing strategy.