CS 3345 Exam Study Guide

Based on your uploaded lecture notes • Excluding B-trees

Purpose: This file explains the main concepts from the covered lectures in plain language so you can study from it directly or paste it into a quiz maker. The emphasis is on definitions, intuition, major operations, complexity, and the differences between similar topics.

1. Algorithm Analysis and Big O

Algorithm analysis is the study of how an algorithm behaves as the input size gets larger. In this course, the main idea is not how many seconds a program took on one specific computer. Instead, the goal is to understand how the running time grows when the problem gets very large.

There are several ways to think about efficiency. You can measure actual running time, count the number of operations, or study the order of growth. The most important one here is order of growth, because it tells you how the algorithm scales in a machine-independent way.

Big O notation is used to describe an asymptotic upper bound. In practice, that usually means it describes the worst-case growth rate of the algorithm. Big O does not care about tiny details such as exact processor speed or minor constants. It focuses on the dominant part of the running time.

When simplifying an expression to Big O, keep the largest-growing term and ignore additive constants and multiplicative constants. For example:
- 4 + 6n becomes O(n)
- n^2 + 2n + 2 becomes O(n^2)
- log n + n becomes O(n)
- n log n + 100n becomes O(n log n)

Two basic rules are used constantly:
- Addition rule: if two pieces of code run one after the other, the total complexity is dominated by the larger term.
- Multiplication rule: if one loop is nested inside another, the complexities multiply.

That is why two separate loops of O(n) and O(n^2) give O(n^2), but a loop of O(n) inside a loop of O(n) gives O(n^2).

You should know the major complexity classes and what they mean:
- O(1): constant time, does not depend on input size
- O(log n): logarithmic time, grows very slowly
- O(n): linear time, grows directly with input size
- O(n log n): common in efficient sorting and divide-and-conquer algorithms
- O(n^c): polynomial time
- O(c^n): exponential time
- O(n!): factorial time

As you move down that list, scalability gets worse. Constant and logarithmic algorithms handle large inputs very well. Exponential and factorial algorithms become impractical quickly.

You should also understand best case, average case, and worst case. For example, in linear search:
- Best case: the value is the first element
- Worst case: the value is not present or is the last element
- Average case: the value is somewhere in the middle

The course mostly emphasizes worst-case behavior, because that gives a reliable upper bound.

2. Maximum Subsequence Sum

The maximum subsequence sum problem asks for the contiguous block of values in an array whose sum is as large as possible. The word contiguous is important. You are not choosing random elements from different positions. You are choosing one uninterrupted segment.

If the array is:
-2, 11, -4, 13, -5, -2

then the maximum contiguous subsequence is:
11, -4, 13

and its sum is 20.

This problem is used to compare multiple algorithmic strategies.

The brute-force idea is to try every possible starting index and every possible ending index, then compute the sum of that interval. That leads to O(n^3), because:
- there are many choices for the start
- many choices for the end
- and then another pass to add the elements in that range

A better version avoids recomputing sums from scratch each time and improves the complexity to O(n^2).

The divide-and-conquer version is more interesting conceptually. It says the maximum subsequence must be in one of exactly three places:
- completely in the left half
- completely in the right half
- crossing the middle

So the algorithm recursively solves the left half, recursively solves the right half, computes the best crossing sum, and then returns the maximum of those three answers.

This leads to the recurrence:
$T(n) = 2T(n/2) + O(n)$

The two recursive calls handle the left and right halves, and the linear part comes from finding the best sum that crosses the middle. The resulting complexity is O(n log n).

This topic is also tied to recursion itself. A correct recursive algorithm must have:
- a base case
- progress toward the base case

Without a base case, recursion never stops. Without progress, it keeps repeating the same problem forever.

3. The List ADT

A list ADT represents an ordered sequence of elements. ADT stands for abstract data type. That means the focus is on what operations the structure supports, not yet how those operations are implemented internally.

Typical list operations include:
- making the list empty
- finding an item
- inserting an item
- removing an item
- printing the list
- finding the element at a certain position

The two main implementations discussed in this course are array-based lists and linked lists.

Array-based lists:
An array stores elements in consecutive memory locations. This makes direct indexing fast. If you want the k-th element, you can jump to it immediately, so findKth is O(1).

The weakness of arrays is insertion and deletion in the middle or front. Since the elements must stay packed together, inserting one value may require shifting many others. That makes insert and remove potentially O(n).

Linked lists:
A linked list stores each element in a separate node, and each node points to the next one. The nodes do not need to be stored next to each other in memory.

This makes insertion and deletion easier, because you can change links instead of shifting many elements. If you are already at the correct location, insertion and deletion can be O(1).

The weakness is access by position. To reach the k-th element, you must move through the nodes one at a time, so findKth is O(n).

The main tradeoff is:
- arrays are better for fast indexed access
- linked lists are better for local insertion and deletion

That tradeoff shows up again and again in data structures.

4. Iterators

An iterator is a standard mechanism for traversing a collection. Instead of writing completely different traversal code for every structure, the iterator provides a common interface.

The main iterator methods are:
- hasNext(): checks whether another element exists
- next(): returns the next element
- remove(): removes the current element in a supported implementation

The advantage of iterators is consistency. Whether the underlying structure is an ArrayList, LinkedList, HashSet, or TreeSet, the traversal idea stays similar. This is why iterators are closely tied to Java collections and the enhanced for loop.

Conceptually, an iterator separates two concerns:
- the collection decides how elements are stored
- the iterator decides how to walk through them

That makes code cleaner and more general.

5. Stack ADT

A stack is a restricted form of list where insertion and deletion happen at only one end, called the top. Because the most recently inserted item is removed first, a stack follows LIFO order: last in, first out.

The two basic stack operations are:
- push: insert an element onto the top
- pop: remove the element from the top

Trying to pop from an empty stack is an error.

Stacks can be implemented efficiently using either arrays or linked lists. In both cases, push and pop can be O(1).

Why stacks matter:
Stacks are useful whenever you need to reverse the natural order of processing or remember the most recent unfinished item.

Balancing symbols:
One major application is checking whether parentheses, brackets, and braces are balanced in a program.

The algorithm is:
- start with an empty stack
- when you see an opening symbol, push it
- when you see a closing symbol, the stack must not be empty
- pop the top opening symbol and make sure it matches
- at the end, the stack must be empty

This works because the most recently opened symbol must be the first one closed.

Postfix expressions:
Another major use is converting infix expressions to postfix. Infix is the normal form people write, like:
a + b * c

Postfix writes operators after their operands, like:
a b c * +

Postfix is useful because it preserves operator precedence without requiring extra parentheses during evaluation.

The conversion idea is:
- if you read an operand, output it
- if you read an operator, pop and output any operators on the stack with higher or equal precedence, then push the new operator
- if you read an opening parenthesis, push it
- if you read a closing parenthesis, pop until the matching opening parenthesis is found
- when input ends, pop all remaining operators

Stacks also naturally evaluate postfix expressions, because each operator acts on the most recent unfinished operands.

6. Queue ADT

A queue is another restricted form of list, but now insertion and deletion happen at opposite ends. A queue follows FIFO order: first in, first out.

The two basic queue operations are:
- enqueue: insert at the rear
- dequeue: remove from the front

Queues model arrival order. The earliest element inserted is the earliest one removed.

Queues can be implemented with arrays or linked lists. In an array implementation, the structure usually tracks:
- front
- back
- current size

One issue with a simple array queue is wasted space. As elements are dequeued, the front moves forward. Even if the queue becomes small, the back may reach the end of the array. The usual solution is a circular array, where the queue wraps around and reuses freed positions at the front.

The main conceptual difference from stacks is:
- stacks reverse recent order
- queues preserve arrival order

7. Trees

A tree is a hierarchical data structure made of nodes. Each node can have at most one parent and zero or more children. Trees are useful because they model hierarchical relationships and, when kept balanced, can support much faster operations than linear structures.

Important terminology:
- root: the top node
- parent: a node directly above another node
- child: a node directly below another node
- sibling: two nodes with the same parent
- leaf: a node with no children
- edge: a connection between parent and child
- path: a sequence of connected nodes
- depth: the number of edges from the root to a node
- height of a node: the length of the longest path from that node to a leaf
- height of a tree: the height of the root

A tree with n nodes has n - 1 edges.

Trees are often defined recursively:
A tree is either empty, or it consists of a root plus zero or more subtrees.

This recursive viewpoint is important because many tree algorithms are naturally recursive.

8. Binary Trees and Tree Shape

A binary tree is a tree where each node has at most two children, usually called the left child and the right child.

Because each node has at most two children, binary trees are easier to reason about than general trees. Many important structures, including BSTs and AVL trees, are special kinds of binary trees.

A binary tree of height h has at most 2^(h+1) - 1 nodes. This formula explains why small height matters so much. If the height stays logarithmic, then the number of levels you need to search through stays small.

Special binary tree shapes:
- Full binary tree: every node has either 0 or 2 children
- Complete binary tree: all levels are full except possibly the last, and the last level is filled from left to right
- Perfect binary tree: every internal node has exactly 2 children and all leaves are at the same depth
- Skewed tree: every node has only one child, so the tree looks like a chain

Balanced trees try to avoid becoming skewed, because skewed trees lose the performance advantages of tree structure.

9. Tree Traversals

A traversal is the order in which you visit nodes.

Depth-first traversals:
These explore downward before moving across.

Inorder:
- visit left subtree
- visit node
- visit right subtree

For a BST, inorder traversal outputs the keys in sorted order.

Preorder:
- visit node
- visit left subtree
- visit right subtree

This is useful when you want the root processed before its descendants.

Postorder:
- visit left subtree
- visit right subtree
- visit node

This is useful when children must be processed before the parent, such as deleting a tree or evaluating certain expression trees.

Breadth-first traversal:
Also called level-order traversal. This visits the tree level by level from top to bottom and usually uses a queue.

Traversal matters because the same tree can produce very different outputs depending on the order of visitation.

10. Binary Search Trees (BSTs)

A binary search tree is a binary tree with an ordering rule:
- every key in the left subtree is smaller than the node's key
- every key in the right subtree is larger than the node's key

This rule is called the BST invariant. It is what makes searching efficient.

Searching in a BST:
Start at the root.
- if the target equals the current node, you found it
- if the target is smaller, go left
- if the target is larger, go right

Instead of scanning every node, the search eliminates half the possibilities at each step when the tree is well shaped.

Basic BST operations:
- contains: determine whether a value is in the tree
- findMin: move left until no more left child exists
- findMax: move right until no more right child exists
- insert: follow the BST rule until reaching a null position, then create the new node
- delete: remove a node while preserving the BST property

BST deletion has three cases:
Case 1: the node is a leaf
Just remove it.

Case 2: the node has one child
Remove the node and connect its parent directly to its child.

Case 3: the node has two children
Replace the node with either:
- its inorder predecessor, which is the largest node in the left subtree

or

- its inorder successor, which is the smallest node in the right subtree

Then delete that predecessor or successor from its original position.

The major weakness of a BST is shape. If insertions happen in an unlucky order, the BST can become skewed. In that case, searching, inserting, and deleting degrade toward O(n) instead of O(log n).

So the BST idea is powerful, but only if the tree stays reasonably balanced.

11. AVL Trees

An AVL tree is a self-balancing binary search tree. It keeps the BST ordering property, but adds another rule:
for every node, the heights of the left and right subtrees can differ by at most 1.

This rule prevents the tree from becoming too unbalanced.

Why AVL trees matter:
The cost of BST operations depends on the tree height. If height stays small, operations stay fast. AVL trees enforce enough balance to keep the height at O(log n), which means search, insertion, and deletion also stay O(log n).

The course notes connect AVL height to a recurrence for the minimum number of nodes in an AVL tree of height h:
$N_h = 1 + N_{(h-1)} + N_{(h-2)}$

This recurrence grows similarly to Fibonacci numbers, which implies the height grows logarithmically with the number of nodes.

AVL insertion:
The insertion begins exactly like a normal BST insertion. After the new value is inserted, you move back up the tree checking whether any node became unbalanced.

If the balance condition is violated, one of four cases occurred:

Case 1: left-left
The new node was inserted into the left subtree of the left child.
Fix: single rotation.

Case 2: left-right
The new node was inserted into the right subtree of the left child.
Fix: double rotation.

Case 3: right-left
The new node was inserted into the left subtree of the right child.
Fix: double rotation.

Case 4: right-right
The new node was inserted into the right subtree of the right child.
Fix: single rotation.

The reason for single versus double rotation:
- outside insertions (left-left and right-right) can be fixed with one rotation
- inside insertions (left-right and right-left) need two rotations because one rotation alone would not restore the correct structure

AVL deletion:
Deletion starts like BST deletion. After removing the node, you may need to rebalance on the way back up, just as with insertion.

The key idea behind AVL trees is strict balance. They do extra maintenance work during updates so that future searches remain reliably fast.

12. Splay Trees

A splay tree is another kind of self-adjusting BST, but it works very differently from an AVL tree.

A splay tree does not store extra balancing information like node heights. It keeps the BST ordering rule, but it rebalances lazily based on access patterns.

The main rule is:
whenever a search successfully finds a node, that node is moved to the root by rotations. This process is called splaying.

Why this is useful:
If a node was expensive to reach because it was deep, the tree uses that expensive search as an opportunity to improve itself. Once the node is moved upward, future searches for that node, or nearby nodes, may become much faster.

So splay trees are adaptive. They naturally bring recently used or frequently used items toward the top.

Insertion and deletion:
The notes describe insertion and deletion as being the same as in a basic BST, but successful searches trigger the major self-adjustment.

Performance:
A single search in a splay tree can range from O(log n) to O(n), depending on the current shape of the tree. However, over many operations, the amortized time becomes O(log n).

That means one operation might be expensive, but the average cost over a long sequence of operations is still efficient.

Main splay rotation patterns:
Zig-zig:
The node, its parent, and its grandparent are aligned in the same direction.
This uses a double rotation in the same direction pattern.

Zig-zag:
The node is aligned opposite its parent and grandparent.
This also uses a double rotation, but with a different structure.

A final single rotation may also be needed if the node ends up one level below the root.

Splay trees preserve the BST invariant during rotations. That means all keys in the left subtree remain smaller than the root key, and all keys in the right subtree remain larger.

The best way to compare AVL and splay trees is this:
- AVL trees maintain strict balance all the time using stored height information
- splay trees do not maintain strict balance state, but they improve themselves through access

So AVL trees are more rigid and predictable, while splay trees are more flexible and adaptive.

13. How These Concepts Connect

These topics are not isolated. They build on each other.

Big O explains how we measure efficiency.
The maximum subsequence sum problem shows how different algorithmic strategies affect complexity.
Lists, stacks, and queues show how changing access rules changes both behavior and performance.
Trees introduce hierarchical structure.
Binary search trees add ordering to make search efficient.
AVL trees add strict balancing to protect worst-case performance.
Splay trees use access patterns to improve performance over time without strict balance metadata.

That progression is one of the biggest ideas in the course: the shape and rules of a data structure directly control the performance of the operations you perform on it.

14. Key Distinctions to Keep Straight

List vs stack vs queue:
- a list is general-purpose
- a stack inserts and deletes from one end
- a queue inserts and deletes from opposite ends

Array list vs linked list:
- array list has fast direct access, slow middle insertion/deletion
- linked list has slow positional access, but easier local insertion/deletion

General binary tree vs BST:
- a binary tree only limits the number of children
- a BST also imposes an ordering rule

BST vs AVL:
- both are ordered binary trees
- AVL adds strict balancing using subtree height differences

AVL vs splay:
- AVL maintains balance proactively during updates
- splay adjusts reactively when nodes are accessed

DFS vs BFS:
- DFS explores down a branch before moving across
- BFS explores level by level

Inorder vs preorder vs postorder:
- inorder: left, node, right
- preorder: node, left, right
- postorder: left, right, node

15. Final Concept Summary

If you understand these core ideas, you understand the heart of the exam material:
- Big O describes growth, not exact time
- recursion must have a base case and progress
- arrays trade cheap access for expensive shifting
- linked lists trade cheap splicing for expensive positional access
- stacks are LIFO
- queues are FIFO
- trees represent hierarchy
- traversal order changes what output you get
- BSTs use ordering to speed search
- AVL trees enforce strict balance for O(log n) operations
- splay trees adapt to access patterns and achieve good amortized performance

This study guide is designed to be readable as one continuous file, but it also breaks naturally into quiz-ready chunks. Each section can become definition questions, comparison questions, tracing questions, or concept explanation questions.