

EECS3101 Algorithms (Summer'26)

Digest of Tutorial 3, May 25

Doubling Dynamic Array and Generic BSTs in Java

Jackie Wang

HOW TO USE THIS DIGEST

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under “*What you should be able to do*”.

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie’s actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

TABLE OF CONTENTS

Main Ideas	2
Roadmap	2
Key Terms, Notation, and Structures	5
Key Examples and Demonstrations	6
Common Pitfalls and Misconceptions	7
What You Should Be Able to Do	9
Suggested Practice	9
Where to Review	10
Closing Reminder	11

Main Ideas

- Tutorial 3 finished the dynamic-array story by comparing fixed-increment resizing with the doubling strategy, with the main payoff being the amortized cost of push.
- The key technical result is that doubling does not improve the worst-case cost of one resizing push; that worst case remains $O(n)$, because existing elements still must be copied.
- The improvement appears only over a sequence of pushes: the resize costs form a geometric sequence $i, 2 \cdot i, 4 \cdot i, \dots, n - 1$, whose total is $O(n)$, so the amortized cost per push is $O(1)$.
- The second half shifted from analysis to implementation, introducing the generic `BSTNode<E>` structure used for binary search trees in Java.
- Jackie used in-order traversal and a small manually constructed BST to preview the Java/JUnit work that supports Assignment 1 and Programming Test 1.
- Jackie also pointed students to optional EECS2101 pre-study assignments on recursion, singly linked lists, and trees; these are practice resources, not submissions.

Roadmap

Tutorial logistics and preparation resources

Jackie began by situating the session as Tutorial 3. The tutorial had two main parts: dynamic-array amortized analysis with doubling, and a first look at generic BST programming in Java. Jackie also announced optional pre-study materials on e-class: three past EECS2101 programming assignments covering recursion, singly linked lists, and trees. These are intended as extra Java practice before Assignment 1 and Programming Test 1, not as required submissions. Jackie emphasized that reading solution code is not enough preparation; students need to practise writing and testing code.

Assignment 1 was planned for release the next Wednesday, and Programming Test 1 was still a few weeks away. Jackie described Assignment 1 as the most relevant study material for the programming test, while the optional EECS2101 materials can help students rebuild prerequisite Java fluency.

From constant increments to doubling

The first technical segment revisited the dynamic-array stack from Tutorial 2. In the fixed-increment strategy, the array capacity grew by adding a constant C . In the doubling strategy, the key code change is that the new capacity is based on `capacity * 2`; after copying, the stored capacity is also doubled. Jackie highlighted that this small implementation change leads to a major amortized-time difference.

The code was split into two conceptual regions. The unconditional push part increments the top index and stores the new element. The conditional resizing part runs only when `size() == capacity`. That resizing block creates a larger array, copies the existing elements, redirects data, and updates capacity.

Worst-case cost of one push

Jackie first separated worst-case analysis from amortized analysis. If a push triggers resizing, the copy loop runs once for each currently stored element. Therefore, the worst-case running time for a single push is still $O(n)$, even under doubling. The benefit of doubling is not that copying becomes cheaper; the benefit is that copying happens less often.

This distinction was important enough that Jackie returned to it later when answering a student question: $O(1)$ amortized does not mean every individual push takes constant time.

Resize costs as a geometric sequence

The iPad notes then visualized the resizing events. Starting with initial capacity i , the first resize copies i elements, the second copies $2 \cdot i$, the third copies $4 \cdot i$, and so on. Jackie rewrote these as

$$2^0 \cdot i, \quad 2^1 \cdot i, \quad 2^2 \cdot i, \quad \dots$$

to make the geometric pattern visible. The common factor is 2, unlike the constant-increment case, where the costs formed an arithmetic sequence.

A key counting point was reused from Tutorial 2: assume, without loss of generality for the analysis pattern, that there are n pushes and the last push triggers the last resize. Immediately before that n -th push, only $n - 1$ elements already exist, so the last resize copies $n - 1$ elements, not n .

Solving for the number of resizes

To apply the geometric-series formula, Jackie derived the number of resizing events. The k -th resizing cost follows the pattern

$$2^{k-1} \cdot i.$$

Since the last resizing cost is $n - 1$, the analysis sets

$$2^{k-1} \cdot i = n - 1.$$

After dividing by i and applying logarithms, Jackie obtained

$$k = \log_2 \left(\frac{n-1}{i} \right) + 1.$$

This step is where the logarithm facts matter: logarithms let us move an exponent down, and $2^{\log_2 x} = x$.

Total resizing cost and amortized push cost

With i as the initial term, common factor 2, and the derived value of k , Jackie applied the geometric sum formula. The resizing total simplifies to

$$2n - 2 - i,$$

so the aggregate resizing cost over n pushes is $O(n)$. Dividing by the number of push operations gives

$$\frac{O(n)}{n} = O(1).$$

Thus, with doubling, push has $O(1)$ amortized running time, even though a resizing push still has $O(n)$ worst-case running time.

Tutorial materials and Eclipse setup

After the break, Jackie pointed students to the Tutorial 3 materials and the short solution video for the comparison task. Jackie then demonstrated the familiar Eclipse import workflow: import an existing project into the workspace and select the archive file. The starter project was expected not to compile immediately, because students must implement missing methods and satisfy the JUnit tests.

The starter project focuses on simple BST utilities, especially in-order traversal and searching. Jackie positioned these exercises as preparation for later assignment work and programming-test practice.

Generic `BSTNode<E>` and internal/external nodes

The BST programming segment introduced the generic `BSTNode<E>` class. Each node stores a key, a value of generic type E , and references to its parent, left child, and right child. Jackie connected this to earlier linked-node structures: a tree node has one parent reference but multiple child references, so it naturally represents a nonlinear structure.

Jackie also distinguished internal and external nodes. A constructor with no arguments creates an external node, which acts as a placeholder and does not contribute meaningful key-value data to search or traversal results. A constructor with key and value creates an internal node. The notes use circles for internal nodes and squares for external nodes.

Recursive in-order traversal and call-by-value

Jackie then showed a recursive helper-method pattern for in-order traversal. The public traversal method creates an `ArrayList` called `order`, passes it to a private recursive helper, and returns it after the helper finishes. Inside the helper, internal nodes contribute to the traversal, while external nodes do not.

The implementation relies on Java call-by-value for object references: each recursive call receives a copy of the reference to the same `ArrayList` object. Mutating the list through any recursive frame accumulates results in the same list. Jackie pointed students back to EECS2030 call-by-value materials and the optional EECS2101 recursion assignment for extra practice with this idea.

Constructing and traversing a small BST

The final trace manually constructed a BST with internal nodes (28, “alan”), (21, “mark”), and (35, “tom”), plus four external nodes. Jackie emphasized the need to set both child and parent links: for example, setting n28’s left child to n21 should be paired with setting n21’s parent to n28.

The purpose of the trace was not just to memorize this tree, but to understand object references at run time. Once the links are correct, in-order traversal starting from n28 should return the internal nodes in sorted-key order: 21, 28, 35.

Key Terms, Notation, and Structures

- **Dynamic array:** An array-backed structure that can allocate a larger array and copy existing elements when more capacity is needed.
- **Fixed-increment strategy:** A resizing strategy that grows capacity by adding a fixed constant C ; from Tutorial 2, its amortized push cost remains linear.
- **Doubling strategy:** A resizing strategy that replaces capacity by $2 \cdot \text{capacity}$; this tutorial showed that it gives $O(1)$ amortized push.
- **Worst-case running time:** The cost of one operation in its most expensive case; for a resizing push, this is $O(n)$ because all current elements are copied.
- **Amortized running time:** The total cost of a sequence of operations divided by the number of operations; for n pushes under doubling, this is $O(1)$ per push.
- **Initial capacity i :** The starting size of the dynamic array in the analysis.
- **capacity:** The current length of the underlying array; this is doubled when resizing occurs.
- **Resize cost sequence:** Under doubling, resize costs follow $i, 2 \cdot i, 4 \cdot i, \dots$, a geometric sequence with common factor 2.
- **Last resize cost $n - 1$:** If the n -th push triggers the last resize, the array contains $n - 1$ existing elements before that push is stored.
- **Number of resizes k :** The analysis solves $2^{k-1} \cdot i = n - 1$, giving $k = \log_2\left(\frac{n-1}{i}\right) + 1$.
- **Geometric sum:** The summation tool used for doubling costs; here it leads to an aggregate resizing cost of $2n - 2 - i = O(n)$.
- **BSTNode<E>:** A generic binary-tree node class with a key, a value of type E , and parent/left/right references.
- **Generic parameter E :** A type parameter allowing the node value to be a String, Integer, Person, or another object type.

- **Recursive type:** A class design where fields have the same class type, such as `BSTNode<E>` fields inside `BSTNode<E>`.
- **Internal node:** A BST node with meaningful key-value data and child references.
- **External node:** A placeholder node with no meaningful key-value data; external nodes do not contribute to the in-order traversal sequence.
- **In-order traversal:** For a BST, the recursive traversal of left subtree, node, then right subtree; it produces keys in sorted order when the tree is a valid BST.
- **Recursive helper method:** A private method used to carry extra state, such as an accumulating `ArrayList`, through recursive calls.
- **Call by value for references:** Java copies the reference value when passing an object parameter; recursive calls can still mutate the same referenced object.
- **JUnit test:** A test case used to check whether the implemented traversal or search method returns the expected result.

Key Examples and Demonstrations

Worst-case push under doubling

Setup: The dynamic-array stack is full, so `size() == capacity` and the next push triggers the resizing block.

Main point: Even with doubling, resizing requires copying all current elements into the new array. The larger future capacity does not remove the current copy cost, so this single operation has worst-case cost $O(n)$.

What to practise: Trace a small array with capacity 4: fill it, trigger a resize, and identify exactly which lines correspond to allocation, copying, pointer update, capacity update, and the actual push.

Amortized derivation for the doubling strategy

Setup: Start with initial capacity i , assume n pushes, and assume the last push triggers the last resizing routine. The resize costs are $2^0 \cdot i, 2^1 \cdot i, 2^2 \cdot i, \dots, n-1$.

Main point: The costs form a geometric sequence. Solving $2^{k-1} \cdot i = n-1$ gives the number of resize terms, and the geometric sum simplifies to $2n - 2 - i$, which is $O(n)$. Therefore the amortized push cost is $O(1)$.

What to practise: Re-derive the formula without looking at the notes. In particular, justify why the last copy cost is $n-1$, and explain why $2^{\log_2 x} = x$ is used in the simplification.

Importing and using the Tutorial 3 starter project

Setup: Jackie demonstrated downloading the Tutorial 3 starter code and importing it into Eclipse as an existing project from an archive file.

Main point: The starter project may have compilation errors until the required classes or methods are implemented. The goal is to use the JUnit tests to drive the implementation of BST operations such as traversal and search.

What to practise: Repeat the import process yourself, run the tests, inspect the failing methods, and implement the missing code until the JUnit results match the expected behaviour.

Recursive in-order traversal with a shared ArrayList

Setup: The traversal method creates an empty `ArrayList<BSTNode<E>>` and passes it to a recursive helper. The helper recursively visits left subtree, current node, and right subtree for internal nodes.

Main point: The same list object accumulates results across recursive calls because Java passes a copy of the reference value, not a new list object. External nodes are base cases and do not add anything to the traversal sequence.

What to practise: Trace the helper on a three-node BST and mark when each recursive call receives the same list reference. Then modify the example mentally: what would go wrong if each recursive call created a fresh list and did not combine the results correctly?

Manually constructing the 21, 28, 35 BST

Setup: The JUnit example creates internal nodes for (28, “alan”), (21, “mark”), and (35, “tom”), then creates four external nodes and links everything into a small BST.

Main point: Correct tree construction requires setting references in both directions: parent to child and child to parent. In-order traversal then verifies the structure by returning keys in sorted order 21, 28, 35.

What to practise: Redraw the object diagram and label each parent, left, and right reference. Then write the corresponding setup code without looking at the solution.

Common Pitfalls and Misconceptions

Confusing worst-case and amortized claims

Pitfall: Saying that doubling makes every push operation $O(1)$.

Why it causes problems: A push that triggers resizing still copies all current elements, so one operation can still be linear.

How to avoid it: Always state the type of analysis: worst-case single push is $O(n)$, while amortized push over a long sequence is $O(1)$.

Counting every push as a resize

Pitfall: Summing costs as if every push operation executes the resizing block.

Why it causes problems: Amortized analysis works because resizing happens only occasionally. Treating every push as a resize destroys the intended analysis.

How to avoid it: Separate ordinary push cost from resizing cost. In this tutorial's derivation, focus on the sequence of resize events: first resize, second resize, third resize, and so on.

Using n instead of $n - 1$ for the last resize

Pitfall: Assuming the last resizing routine copies n elements when the n -th push triggers it.

Why it causes problems: The copy happens before the new element is stored. At that moment, there are only $n - 1$ existing elements.

How to avoid it: Draw the array state immediately before the triggering push: $n - 1$ occupied slots are copied, then the n -th element is added afterward.

Losing the geometric-sequence pattern

Pitfall: Treating the doubling costs as if they formed an arithmetic sequence.

Why it causes problems: Fixed increments lead to an arithmetic sequence; doubling leads to a geometric sequence. Using the wrong summation formula gives the wrong conclusion.

How to avoid it: Rewrite the resize costs as $2^0 \cdot i, 2^1 \cdot i, 2^2 \cdot i, \dots$. The exponent pattern is the clue that the common factor is 2.

Reading code solutions without practising

Pitfall: Looking at posted Java solutions and assuming recognition is enough preparation.

Why it causes problems: Programming tests require writing, tracing, debugging, and adapting code, not just understanding a finished solution when it is already in front of you.

How to avoid it: Use the optional EECS2101 materials and Tutorial 3 starter project actively: import, code, run tests, fix errors, and repeat.

Forgetting parent links in tree construction

Pitfall: Setting left or right links but forgetting the corresponding parent link.

Why it causes problems: The object structure no longer matches the intended tree, and later operations that depend on upward navigation can fail.

How to avoid it: Treat each connection as a pair of updates: after `parent.setLeft(child)`, also check `child.setParent(parent)`.

Treating external nodes as meaningful data nodes

Pitfall: Adding external nodes to the in-order traversal sequence or treating their placeholder values as searchable keys.

Why it causes problems: External nodes are placeholders for missing children or future insertions; including them corrupts the traversal result.

How to avoid it: In traversal and search code, explicitly distinguish `isInternal()` from `isExternal()` and only collect meaningful internal nodes.

What You Should Be Able to Do

After this lecture, you should be able to:

- identify the code-level difference between fixed-increment resizing and doubling in a dynamic-array stack;
- explain why a single resizing push is still $O(n)$ in the worst case under doubling;
- derive the resize-cost sequence $i, 2 \cdot i, 4 \cdot i, \dots, n - 1$ and explain why it is geometric;
- justify why the last resize copies $n - 1$ elements when the n -th push triggers it;
- solve $2^{k-1} \cdot i = n - 1$ for k using logarithms;
- apply the geometric-sum formula to show that total resizing cost under doubling is $O(n)$;
- explain why $O(n)$ total resizing cost over n pushes gives $O(1)$ amortized cost per push;
- compare the amortized behaviour of fixed increments and doubling without confusing amortized cost with worst-case cost;
- import the Tutorial 3 starter project into Eclipse and use JUnit tests to guide implementation;
- describe the fields of `BSTNode<E>` and explain why it is a recursive type;
- distinguish internal nodes from external nodes in the BST representation used in the tutorial;
- implement or trace a recursive in-order traversal helper using a shared `ArrayList`;
- explain how Java call-by-value for object references allows recursive calls to accumulate results in the same list;
- manually construct a small BST by correctly setting `left`, `right`, and `parent` links.

Suggested Practice

- Redo the full doubling derivation from scratch: draw the first few resizing events, write the geometric sequence, solve for k , sum the series, and divide by n .
- Trace a dynamic-array stack with a small initial capacity such as $i = 2$ or $i = 4$; record exactly which pushes trigger resizing and how many elements are copied.

- Compare Tutorial 2 and Tutorial 3 side by side: explain why fixed increments give linear amortized cost while doubling gives constant amortized cost.
- Watch the posted short solution video for Tutorial 3 Task 2, then write your own comparison of the two resizing strategies in words.
- Import the Tutorial 3 starter project into Eclipse using `Select archive file`, run the JUnit tests, and implement the missing traversal and search methods.
- Recreate the small BST with keys 21, 28, 35 in Java, including all external nodes and parent links; then verify the in-order traversal result.
- Review Java generics and call-by-value using the optional EECS2030/EECS2101 review materials Jackie pointed to, especially if recursive helper methods still feel unclear.
- Work through at least one optional EECS2101 pre-study assignment actively rather than only reading its solution.

Where to Review

- **Tutorial 3 PDF/iPad notes, page 1:** overall structure of the tutorial: Dynamic Arrays Part 2 and testing generic BSTs using JUnit.
- **Tutorial 3 PDF/iPad notes, page 2:** main derivation for amortized analysis of dynamic arrays with doubling, including the resize sequence, the k -th resize, and the final $O(1)$ amortized result.
- **Tutorial 3 PDF/iPad notes, page 3:** visual reminder that worst-case cost applies to a particular resizing push, while amortized cost averages over n push operations.
- **Tutorial 3 PDF/iPad notes, page 4:** structure of a generic binary tree node, including parent, key, value, left, and right.
- **Tutorial 3 PDF/iPad notes, page 5:** internal versus external nodes in the BST representation.
- **Tutorial 3 PDF/iPad notes, page 6:** recursive in-order traversal helper and the shared `ArrayList` accumulation pattern.
- **Tutorial 3 PDF/iPad notes, page 7:** JUnit-style trace for constructing and traversing the small BST with keys 21, 28, 35.
- **Tutorial 3 instructions and starter code:** use these for the actual Eclipse/JUnit practice; the digest is not enough to prepare for programming work.
- **Posted Task 2 solution video:** review it after attempting the strategy-comparison task yourself.

- **Optional e-class pre-study materials:** past EECS2101 assignments on recursion, singly linked lists, and trees, plus starter projects and solutions.
- **EECS2030 call-by-value refresher:** revisit this if the recursive traversal helper with a shared list is confusing.

Closing Reminder

Tutorial 3 connects a precise mathematical analysis with concrete Java implementation practice. Use this digest to find the main ideas, but do the real work in the official notes, the recording, the tutorial starter code, the JUnit tests, and the optional practice materials. In particular, do not stop at recognizing the amortized derivation or reading the BST solution code: redraw the resize diagrams, re-derive the formulas, import the project, write the methods, trace the object references, and debug your own mistakes.