Vol. 11, No. 4, 2026, pp. 251-262
DOI: https://doi.org/10.29210/027278jpgi0005

Contents lists available at Journal IICET

JPGI (Jurnal Penelitian Guru Indonesia)
ISSN: 2541-3163(Print) ISSN: 2541-3317 (Electronic)

Journal homepage: https://jurnal.iicet.org/index.php/jpgi



Functional validation of numeric and string key processing in a B-Tree visualization simulator

Edison Pardengganan Siahaan
Teknik Informaika, Fakultas Teknik, Univesitas Mpu Tantular, Jakarta, Indonesia

Article Info

Article history:

Received Oct 01th, 2026
Revised Nov 02th, 2026
Accepted Nov 19th, 2026

Keyword:

B-Tree visualization,
Software validation,
Differential testing,
Invariant checking,
Algorithm visualization

ABSTRACT

This study evaluates the functional correctness and core-operation response of a Java-based B-Tree visualization simulator supporting numeric and string keys. The iterative prototype uses a Model–View–Controller architecture, Observer-based animation, and Memento-based undo/redo. Numeric inputs from 1 to 999 are normalized to three-digit zero-padded strings; alphabetic keys of up to five characters are processed separately. Evaluation comprises seven validator unit tests, four black-box functional scenarios, B-Tree invariant checks, and randomized differential comparison against java.util.TreeSet. Across orders 3–10 and 200 random seeds per mode, 300 operations per scenario yielded 960,000 operations; the reported runs showed no invariant violations or content mismatches. Reported core-operation response times were below 100 ms, excluding interface rendering. Testing also identified defects that were subsequently corrected. These results support software correctness within the tested domain, but do not demonstrate usability or learning gains because no user-based evaluation was conducted.

© 2026 The Authors. Published by IICET.
This is an open access article under the CC BY-NC-SA license
(https://creativecommons.org/licenses/by-nc-sa/4.0)

Corresponding Author:

Edison Pardengganan Siahaan,
Teknik Informatika, Fakultas Teknik, Universitas Mpu Tantular,
Email: edisonsiahaan@mputantular.ac.id

Introduction

The index data structure is a core component of database systems and file systems (Bayer & McCreight, 1972; Comer, 1979). B-Tree, introduced by Bayer and McCreight (1972), maintains a high balance of the tree so that the amount of access to secondary storage media remains small. The search, insertion, and deletion operations in the B-Tree have an O (log n) complexity (Comer, 1979; Cormen et al., 2022). Therefore, B-Tree is the main material in the data structure and database course.

However, the dynamic B-Tree process includes many special cases, namely splitting, borrowing keys from sister nodes, *merging*, and *underflow* (Cormen et al., 2022), making them difficult to trace manually. Algorithmic visualization is widely used to help understand this kind of process (Hundhausen et al., 2002; Shaffer et al., 2010). A recent systematic review of 90 papers showed that visualization tools were associated with better engagement and learning outcomes, but rigorous evaluations were still rare and visualization models designed specifically for specific challenges were still needed (Sibia et al., 2025). Other reviews of algorithmic design teaching reported mixed results: students liked visualizations, few studies measured learning outcomes experimentally, and learning improved when visualization demanded students take an active role (Liu et al., 2025). These findings prompted the design of interactive simulators, with user input, step-by-step animations, and *undo/redo* features.

The comparison of visualization tools provides a requirements-level justification for a new prototype rather than evidence that the prototype improves learning. Existing systems differ in the data structure shown, the type of input they accept, animation controls, and whether past operations can be revisited. The present development combines a true B-Tree, separate numeric and alphabetic input modes, controllable animation, and multi-step undo/redo in a single Java application. The feature comparison in Table 1 is a descriptive review of the cited tools, not a controlled performance benchmark. It therefore establishes a design gap that can be examined through reproducible software checks while leaving comparative usability and teaching effectiveness for a separate study.

The available B-Tree simulators have a wide range of features. The USFCA Data Structure Visualizations (Wales, n.d.) provides B-Tree animations with maximum degrees of 3 to 7 degrees and step-by-step playback control. The STX B+ Tree prop (Bingmann, 2007) depicts the B+ tree with an integer or string key. VisuAlgo (Halim, n.d.) is a common visualization tool that is widely used, but does not have a B-Tree module as of the author's access date. Based on a search of four B-Tree or B+ tree simulators, namely USFCA (Wales, n.d.), STX B+ Tree (Bingmann, 2007), OpenDSA (OpenDSA Project, n.d.), and B+ Tree Viewer on the Minibase package (B+ Tree Viewer, n.d.), which were selected with the criteria of having a tree structure visualization, supporting B-Tree or B+ tree operations, and being freely accessible on September 14, 2026 through the keyword searches "B-Tree visualization", "B+ tree simulator", and "interactive data structure structure visualizer" on Google and Google Scholar (Table 1), only STX B+ Tree explicitly supports string keys, while the B+ Tree Viewer documentation does not mention the supported key type. No simulator provides a history of multi-step *undo/redo* operations, and step-by-step browsing is available on USFCA, OpenDSA, and B+ Tree Viewer. In fact, database systems commonly index text columns such as CHAR and VARCHAR with B-Tree (PostgreSQL Global Development Group, n.d.-a), so simulators that only accept numbers are less representative of real practice.

Supporting numeric keys and strings on a single B-Tree engine requires consistent comparison rules. If an integer is stored as a String, lexicographic comparison results in a different order than a numerical sequence; for example "10" is considered smaller than "2". The B-Tree remains structurally valid against any consistent sequence, but the numeric key visualization will deviate from the taught sequence. Normalization with *zero-padding*, which is documented in the Amazon SimpleDB developer guide (Amazon Web Services, n.d.) and can be implemented on relational databases with functions such as lpad in PostgreSQL (PostgreSQL Global Development Group, n.d.-b), aligns the two sequences for fixed-width non-negative integers, but carries limitations of maximum width and the absence of negative numbers. More common encoding schemes, such as ELEN (Seymour, n.d.), handle negative numbers and free lengths, but are more complex. The *zero-padding* approach is chosen over generic type-based comparators (*Comparable*) because it maintains a single benchmark path (*String.compareTo*) for B-Tree logic, simplifies the *TreeSnapshot* structure for animations, and simplifies invariant testing, with the consequence that such limitations are explicitly tested and reported.

This study aims to design and test the support of two key modes, numeric and string, on the Java-based B-Tree visualization simulator. The contributions are: (1) the design of the key normalization with *zero-padding* and its input validation rules; (2) the Model-View-Controller-based simulator architecture with *the Observer* pattern for step-by-step animation and *the Memento* pattern for *undo/redo*; and (3) systematic verification through unit testing, functional testing (*black-box*), and B-Tree invariant checking. The scope of the study was limited to positive integers 1–999 and basic alphabet strings processed in separate modes, orders 3 through 10, as well as insertion, deletion (with predecessor), and search operations.

The study addresses three operational questions. First, can each input mode enforce its documented domain and retain the intended ordering when keys are inserted, searched, or deleted? Second, can the implemented structure preserve essential B-Tree invariants and match an independent set implementation under repeated randomized operations? Third, do the reported functional scenarios demonstrate that interface actions and core operations satisfy the specified acceptance conditions? The evaluation unit is a program operation or defined functional scenario, not a human participant. Accordingly, validation concerns functional software behavior; it does not estimate learning gains, user satisfaction, or the educational impact of visual representations.

A successful outcome is operationalized as an accepted or correctly rejected input, a correctly maintained tree state, agreement with the TreeSet oracle, or a passed functional scenario. The design also records core processing time for specified operations, while animation and rendering times are excluded. These distinctions make the evidence inspectable and prevent architectural features from being mistaken for empirically validated pedagogical benefits. As prior visualization research emphasizes engagement and active learning (Hundhausen et al., 2002; Shaffer et al., 2010; Liu et al., 2025), the implemented interaction controls are pedagogical design opportunities rather than measured learning outcomes.

Literature Review

B-Tree

The B-Tree is a balanced search tree introduced by Bayer and McCreight (1972). In contrast to a binary tree, a single node can store many keys and have many children. This study uses definitions based on the order m (Comer, 1979; Knuth, 1998), which is the maximum number of children per node. The properties of the B-Tree of the order m: (a) each node has the most m children, or $m-1$ key; (b) each non-root internal node has at least $\lceil m/2 \rceil$ children, so each non-root node has at least $\lceil m/2 \rceil - 1$ key; (c) non-leafy roots have at least two children; (d) nodes with k-keys have k+1 children; (e) keys in ascendingly ordered nodes; and (f) all leaves are at the same depth. This definition was chosen because the definition of minimum degree t in Cormen et al. (2022) only includes an even order. In this paper, tree height is defined as the number of sides from root to leave.

Insertion is done at the leaf node. When the number of keys exceeds $m-1$, the node is split at the median key, the median is promoted to the parent, and the break can spread to the roots so that the height of the tree increases (Comer, 1979; Knuth, 1998). On removal, the key on the internal node is replaced by its predecessor (predecessor, the largest key on the left subtree), and then the replacement key is removed from the leaves. If the node lacks a key (*underflow*), the node borrows the key from the surplus sibling through rotation through the parent (*borrow*), or if there is none, merges with the sibling along with the key separating the parent (*merge*); this process can creep upwards (Cormen et al., 2022; Knuth, 1998).

Comparison of Numerical and Lexicographic Keys

Numeric keys are compared by magnitude, while string keys are compared lexicographically, character by character from the left. In Java, the *String.compareTo method compares the* value of a UTF-16 code unit (Oracle, n.d.-b). Therefore "10" is smaller than "2" (character '1', code 49, precedes '2', code 50). Character-code-based order also means that capital letters precede lowercase letters ("Z" < "a"), as opposed to dictionary sequences that consider collation of a particular language (Oracle, n.d.-b). When relying only on this comparison for numbers, the tree order does not correspond to the numerical order, although the tree remains consistent with the lexicographic order.

Zero-Padding Normalization

This technique adds a zero character in front of the number until it reaches a *fixed-width*. The Amazon SimpleDB developer guide, which stores all values as strings and compares them lexicographically, suggests *zero-padding* (along with a shift or offset for negative numbers) so that the string sequence is the same as the numerical sequence (Amazon Web Services, n.d.). In relational databases, something similar can be done with character filler functions such as lpad (PostgreSQL Global Development Group, n.d.-b). With a width of 3 digits, 2 becomes "002" and 10 becomes "010", so "002" < "010" correspond to 2 < 10. In Java, adding a zero can be done with a 0 marker in the *Formatter*, e.g. *String.format("%03d", n)* (Oracle, n.d.-a).

This sequence equivalence occurs only if all keys are non-negative integers and do not exceed the specified width, so the maximum width must be specified at the beginning. A more common alternative, which deals with both negative numbers and free lengths, is an encoding scheme that includes length markers and signs, such as ELEN (Seymour, n.d.), but the implementation is more complex than *zero-padding*.

Previous Visualization Simulator

Algorithmic visualization has long been researched in computational education (Hundhausen et al., 2002; Shaffer et al., 2010). Some relevant tools: USFCA Data Structure Visualizations (Wales, n.d.) (B-Tree with maximum degrees 3 to 7 and animation controls), STX B+ Tree props (Bingmann, 2007) (B+Tree structures with integer or string keys, desktop applications), VisuAlgo (Halim, n.d.) (common tools without B-Tree modules), OpenDSA (OpenDSA Project, n.d.) (B+ tree module in interactive textbooks), and B+ Tree Viewer (B+ Tree Viewer, n.d.) (Minibase package component for database teaching run via btview commands, with a step-by-step operation trace tracing). The explored OpenDSA module focuses on visualizations embedded in eTextbooks (e.g. insertion exercises on the B+ tree), so that the user follows a predefined flow and does not enter keys freely as in a standalone simulator (e.g. USFCA). The B+ Tree Viewer documentation does not mention the supported key types. Table 1 summarizes the comparison of the features of the tools.

Table 1. Comparison of features of previous B-Tree/B+tree simulators and proposed systems

Features	USFCA (Galles, n.d.)	STX B+ Tree (Bingmann, 2007)	OpenDSA (OpenDSA Project, n.d.)	B+ Tree Viewer (B+ Tree Viewer, n.d.)	Proposed system
Structure	B-Tree	B+ tree	B+ tree	B+ tree	B-Tree
Adjustable order	3 to 7	Can be selected from multiple fixed values	Uncontrollable (fixed, 4th degree)	Unmanageable	3 to 10
Numeric key	Yes	Yes	Yes	Yes	Yes
String key	No	Yes	No	Unknown	Yes (separate mode)
Step animation controls	Yes	No	Yes	Yes (Step-by-step trace tracing)	Yes
History of *undo/redo* operation	No	No	No	No	Yes
Multilingual	No	No	No	No	Yes (ID/EN)

Architecture and Design Patterns

The Model-View-Controller pattern separates data and logic, views, and interaction controllers. The *Observer* pattern allows objects to subscribe to events from other objects without direct dependencies, and the *Memento pattern* stores the state of the object so that it can be restored without revealing its internal details (Gamma et al., 1994). All three are relevant for simulators that require incremental animation and operation cancellation.

Software Testing

Unit testing verifies functions or classes separately, *black-box* testing assesses user-side behavior without looking at the code, and *white-box testing* assesses the scope of the code structure (Ammann & Offutt, 2016). For data structures, invariant checks after a series of random operations help find violations that are not caught in manual test cases. This approach can be strengthened with *property-based testing* using random data generators (MacIver et al., 2019). Comparisons with established reference implementations (*oracles*) are also commonly used to assess the correctness of the content of the data structure.

Method

Software Development Model

This research uses *a Prototyping* (iterative) model. This model was chosen because the development of visualization applications requires repeated feedback, especially related to interface and animation accuracy. The development stages in general include needs analysis, architectural design, implementation, testing, and iterative evaluation.

The development process is carried out in four main iterations. In the first iteration, the focus is on building the foundation of logic by implementing the B-Tree core algorithm (*BTree* and *BTreeNode*) using the String data type, which at this stage does not have an animation feature. In the second iteration, development is focused on implementing the Model-View-Controller architecture and *the Observer* pattern. Added *StepRecorder* and *AnimationManager* classes so that any node changes (such as *splits* and *merges*) can be recorded as a *snapshot* and visualized *step-by-step*.

The third iteration expands functionality by adding numeric key and string mode support. *A zero-padding* technique is implemented on *the InputValidator* to align numerical sequences, accompanied by interface updates to accept *batch* inputs. The fourth iteration focuses on refining and fixing errors. The *undo/redo* feature is added by utilizing *TreeSnapshot* as a memento (*Memento* pattern), along with multilingual (ID/EN) support. This stage also fixed a critical error found during testing, namely a regular expression that erroneously deletes the zero prefix in the string key ("007") when the string mode is still receiving numeric input, as well as *underflow* handling on the deletion of internal nodes.

The four iterations can be read as an engineering traceability sequence: algorithmic structure and correctness are established as the target of Iteration 1; event recording and stepwise presentation form the target of Iteration 2; input-domain validation and mode separation form the target of Iteration 3; and regression repair, undo/redo,

and language handling form the target of Iteration 4. The checkpoints are defined by the functions and constraints described in Tables 2–5, followed by the final verification evidence in Tables 6–8. This mapping clarifies what each development stage sought to deliver; it must not be read as evidence that independent user acceptance tests were administered after every iteration.

The acceptance criteria separate observable software properties from educational constructs. Structural correctness requires preservation of node bounds, ordering, search ranges, and equal leaf depth. Functional correctness requires the expected response to valid and invalid inputs and to selected insertion, deletion, search, and undo operations. The specified performance threshold applies to the core-operation response test, not the complete animation pipeline. Usability, ease of learning, satisfaction, and learning improvement would require defined participant tasks and outcome instruments, which were not part of the reported prototype tests.

Needs Analysis

Table 2. Functional needs

ID	Functional needs	Description
F-01	Insert operation	Add a key to the B-Tree and handle *splits*.
F-02	Delete operation (delete)	Remove keys, exchange them with predecessors, handle *underflow*.
F-03	Search operation (search))	Search for the key and display the *path tracking*.
F-04	Pemrosesan *batch*	Insert or delete multiple keys at once.
F-05	*Undo/redo*	Cancel or repeat operations based on *snapshot history*.
F-06	Parameter settings	Change the order of the B-Tree (3–10) and key type (numeric or string).
F-07	Validate feedback	Rejects empty, out-of-bound, or incorrectly formatted input.
F-08	Multilingual	Supports Indonesian and English interfaces.
F-09	Animation	Display the movement of the nodes with Pause, Step, and Slider controls.

Tabel 3. Non-functional Needs

ID	Non-functional needs	Description and metrics
NF-01	Sequence accuracy	The sequence of keys is lexicographically or numemericall, verified with *the TreeSet oracle* (Table 7).
NF-02	Responsiveness of operations	The response time of the B-Tree core operation (without depiction) < 100 ms on 100 keys (Table 8).
NF-03	Maintainability	Modular code with MVC separation, shown by the number of classes per layer (Table 4).

System Architecture Design

The application implements a Model-View-Controller (MVC) architecture that separates the logic of the data structure from the interface (Figure 1). The model consists of *BTree*, *BTreeNode*, *TreeSnapshot*, and *NodeSnapshot*. The model package contains only the logic and data structure without importing the graphical interface library (*javax.swing*), so the algorithm can be tested independently. The View consists of *BTreeVisualization*, *LeftPanel*, *CenterPanel*, *RightPanel*, and *BTreePanel* (*Graphics2D canvas*) along with layout helper classes (*TreeLayout*) and custom interface components (*UIComponents*). The Controller and Manager consist of *BTreeController*, *AnimationManager*, *InputValidator*, and *UndoRedoManager*. The event and listener mechanisms involve *StepRecorder*, *BTreeEvent*, *AnimationStep*, as well as *the BTreeListener* and *AnimationListener* interfaces. The division of classes into these architectural layers is shown in Table 4. Figure 2 shows a diagram of the core class, and Figure 3 shows the sequence of interactions on insert operations.

Table 4. Number of classes per architecture layer>

Coating	Quantity	Classes
Model	4	*BTree*, *BTreeNode*, *TreeSnapshot*, *NodeSnapshot*
Controller dan Manager	6	*BTreeController*, *AnimationManager*, *InputValidator*, *UndoRedoManager*, *ValidationResult*, *LanguageManager*
Events and listeners	7	*StepRecorder*, *BTreeEvent*, *AnimationStep*, *BTreeListener*, *AnimationListener*, *BTreeEventType*, *StepType*
View	8	*BTreeVisualization*, *LeftPanel*, *CenterPanel*, *RightPanel*, *BTreePanel*, *UIComponents*, *CustomDialogs*, *TreeLayout*
Quantity	**25**	Includes interfaces and enumerations; *inner classes* are not counted.

The implementation of the design pattern is as follows: Observer Pattern (two levels). The flow of events does not occur directly from *BTree* to the interface, but rather through two levels. Level 1: *BTree* emits an event (*BTreeEvent*) to *BTreeListener* implemented by *StepRecorder*, which records it into an *AnimationStep* list. Level 2: *AnimationManager* plays the recording and emits the event to *the AnimationListener* implemented by *BTreeVisualization* (interface).

Memento patterns. This pattern supports *the undo/redo* feature. *Originator*: *BTree*, which is an object whose state is stored. *Memento*: *TreeSnapshot* and *NodeSnapshot*, which are immutable objects that record the state of the tree. *Caretaker*: *UndoRedoManager*, which is an object that stores and manages the memento stack. *TreeSnapshot* is also brought by events to draw each step of the animation.

Key Normalization and Benchmarking Design

One of the design decisions in this study was the use of a single comparison path (String) for two key modes. The entire key is stored as a String within *the BTreeNode* and compared to *the String.compareTo*. The validation and normalization flows are shown in Figure 4, and a summary of the rules is shown in Table 5.

Numeric mode. The input is parsed with *Integer.parseInt*. To simplify the visual representation, the number 0 is rejected so that the accepted range is a positive integer 1–999. The accepted value is normalized to a fixed width with *String.format("%03d", value)* (Oracle, n.d.-a), so that the number 10 is saved as "010". Negative numbers, values above 999, and non-numeric inputs are rejected.

String mode. Text (e.g. "AX") is accepted only if it consists of a basic alphabet (A–Z, a–z) without spaces or special characters, with a maximum length of five characters. The sorting is done lexicographically purely with *String.compareTo*. This comparison is case-sensitive based on the value of the UTF-16 code unit (Oracle, n.d.-b), so that the letter "Z" precedes the letter "a".

Interface view. The regular *expression ^0\d+$* is applied only in numeric mode to remove the front zero when displayed, so that "010" appears as "10". In string mode, this expression is not applied. Because string mode only accepts alphabets, numeric input such as "007" is rejected before it reaches the appearance stage.

Duplicate keys and mode switching. Key inputs already in the tree are rejected by the validator and the user receives an error message. If the user switches the key mode (from numeric to string or vice versa), the system empties the tree automatically to prevent structural inconsistencies and mixing of key types within a single tree.

Table 5. Key Validation and Normalization Rules

Rules	Numeric mode	String mode
Data type	Integers	Text alphabet (A-Z, a-z)
Value limitation	1 to 999 (0 number rejected)	Maximum length of 5 characters
Special characters	Rejected (numbers only)	Rejected (spaces and symbols are not allowed)
Duplicate	Rejected	Rejected
Internal normalization	Zero-padding 3 digit ("010")	Without normalization ("AX")
Interface display	No zero ahead ("10")	As per the input ("AX")
When the mode is changed	Trees are empited	Trees are emptied

Testing Plans

Tests were performed by mode (numeric and string) in a JDK 17 environment with NetBeans IDE, using JUnit 4.13.2 and the hamcrest-core 1.3 library. Tests were conducted on the Windows 11 operating system, Intel Core i3 processor, and 8 GB of RAM. There are four types of tests. The submission describes source code and automated scripts, but no stable repository URL or commit identifier is supplied here; these should be confirmed by the authors for reproducibility.

Research design and unit of analysis. This is an iterative software-development and technical-validation study, not a randomized comparison involving students or teachers. The units tested are input-validation cases, defined interface scenarios, and machine-generated B-Tree operations. Its intended application is educational visualization of B-Tree operations, but the reported evaluation does not recruit a target-user sample. Consequently, respondent sampling frames, expert rating scales, Cronbach's alpha, inter-rater coefficients, and statistical power calculations for human outcomes are not applicable to the present data; they cannot be reported without a separately conducted study.

Measurement and traceability. The InputValidator tests map to domain restrictions and normalization rules in Table 5. Repeated operations are assessed through an independent reference container and a structural invariant checker, so a discrepancy can arise from either incorrect contents or an invalid tree layout. The functional scenarios correspond to selected behaviors listed in Table 2, and the response-time criterion corresponds to NF-02 in Table 3. Other listed capabilities, including multilingual operation and batch entry, are described as implemented functions, but the manuscript should not treat them as exhaustively verified by the four reported interface scenarios. Source-code revision, test logs, and a stable repository link would strengthen independent replication and should be supplied by the authors when available.

Testing the InputValidator unit. Test cases include value limits (0, 1, 999, 1000), incorrect input formatting, empty inputs, and string mode testing (e.g. "AX" is accepted, "A1" is rejected, and "007" is rejected because string modes are restricted to the alphabet). The results are presented in Table 6.

B-Tree invariant verification. After each operation, an independent checker verifies: (1) strictly ordered keys without duplicates within nodes; (2) at most $m-1$ keys per node and, except for the root, at least $\lceil m/2 \rceil - 1$ keys; (3) exactly $k+1$ children for an internal node containing k keys, with at least two children for an internal root; (4) all leaves at the same depth; (5) every child subtree bounded by its parent separator keys; and (6) an in-order traversal matching java.util.TreeSet. These are structural and content checks, not tests of educational effectiveness.

Random differential testing. *java.util.TreeSet* acts as an independent *oracle*. For each mode, orders 3 to 10 (8 values) and *seeds* 0 to 199 (200 values) are formed 1,600 scenarios. Each scenario executes 300 random insert and delete operations, with a random numeric key of 1–999 normalized to 3 digits. Duplicate keys are prevented (not inserted into either *TreeSet* or B-Tree) to keep the two structures in sync. After each operation, the contents of the B-Tree are compared to the *TreeSet* and all invariants are checked, bringing the total to 480,000 operations per mode. The results are presented in Table 7.

Rationale for test volume. For each of the two input modes, eight permitted tree orders (3 through 10) are crossed with 200 random seeds and 300 insert/delete operations per seed. This produces $8 \times 200 \times 300 = 480{,}000$ operations per mode and 960,000 overall. The repetition provides broad computational stress coverage across configurations and reproducible random sequences, not a probability sample of real-world users or a statistically powered test of learning effectiveness. The source does not provide a formal optimality analysis of 200 seeds or 300 operations; these settings should therefore be described as explicit coverage parameters rather than as numbers justified by inferential sampling theory. Variation in operation mix, key distribution, and boundary frequency may still expose defects not encountered in the reported runs.

Replicability and independent checks. The reported operating system, Java runtime, testing libraries, hardware, orders, seeds, and number of operations establish several important conditions for reproduction. To permit exact replay, a future technical appendix should also disclose the pseudo-random generator and seed initialization, test-case implementation, complete test logs, source-code commit identifier, and the exact rules used for applying accepted or rejected duplicate keys. These items are a recommended reporting extension, not additional experiments claimed to have been performed here. The same distinction applies to the reported sub-100 ms measurements: the available evidence supports response times for the stated tested operations on the specified setup, not a hardware-independent upper bound.

Functional testing (black-box) via the interface. Insert scenario (form a 3-tall tree), remove the internal key, search for missing keys, and repeat *undo* is executed through the interface (Table 8). For NF-02 requirements, the response time of the core operation is measured with *System.nanoTime() after the* JVM warm-up *phase* of 1,000 operations so as not to be biased by the initial compilation. The target is a response time < 100 ms.

Results and Discussions

Implementation Results and Discussion

The application was successfully built with the Java Swing graphical interface (Figure 1). The left panel contains order settings, manual and *batch* inputs, and animation controls; the middle panel displays tree visualizations; and the right panel contains operation logs, tree information, searches, and *undo/redo* buttons. The division of the 25 application classes into four layers is shown in Table 4.

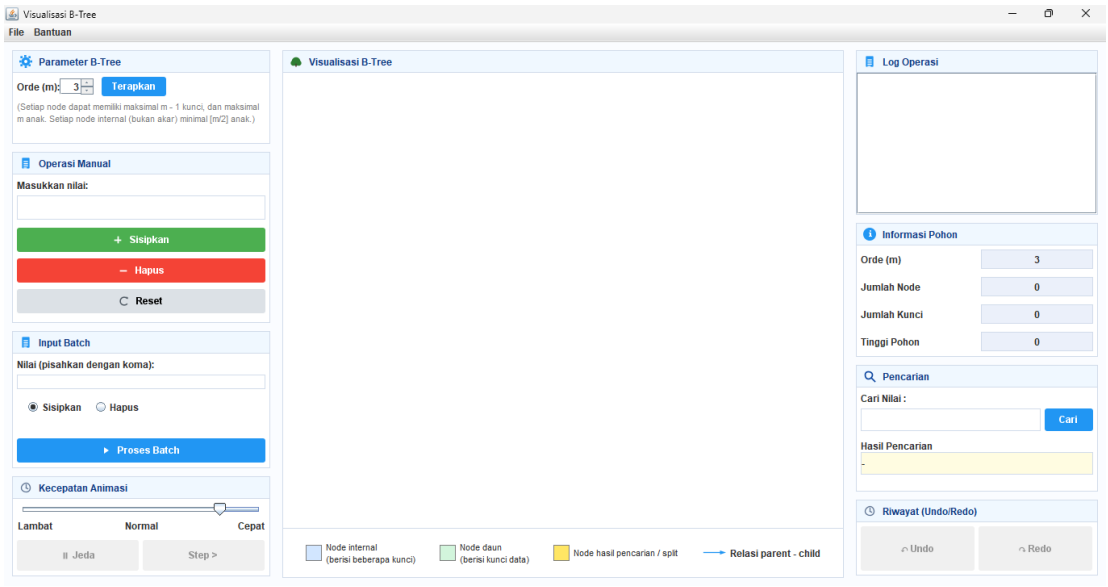


Figure 1. Screenshot of Main Interface

Figure 2 compares the visualization in numerical mode, which displays a key with no zero in front, and string mode. Figure 3 shows a series of split proces animations in string mode.

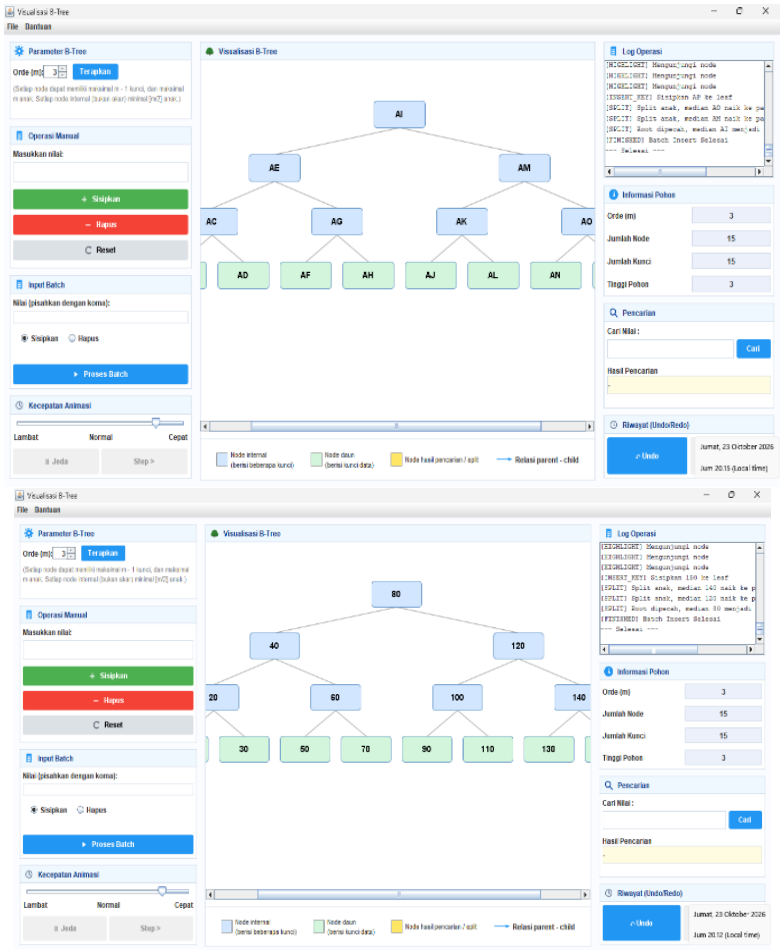


Figure 2. Numerical Mode (left) and String Mode (right) visualization

Consequences of the zero-padding approach. A single comparison path simplifies B-Tree logic and testing, but limits numerical keys to positive integers 1–999. Range expansion requires only adding padding widths, while negative and decimal numbers require more general coding schemes such as ELEN (Seymour, n.d.) or type-based comparators (*Comparable*).

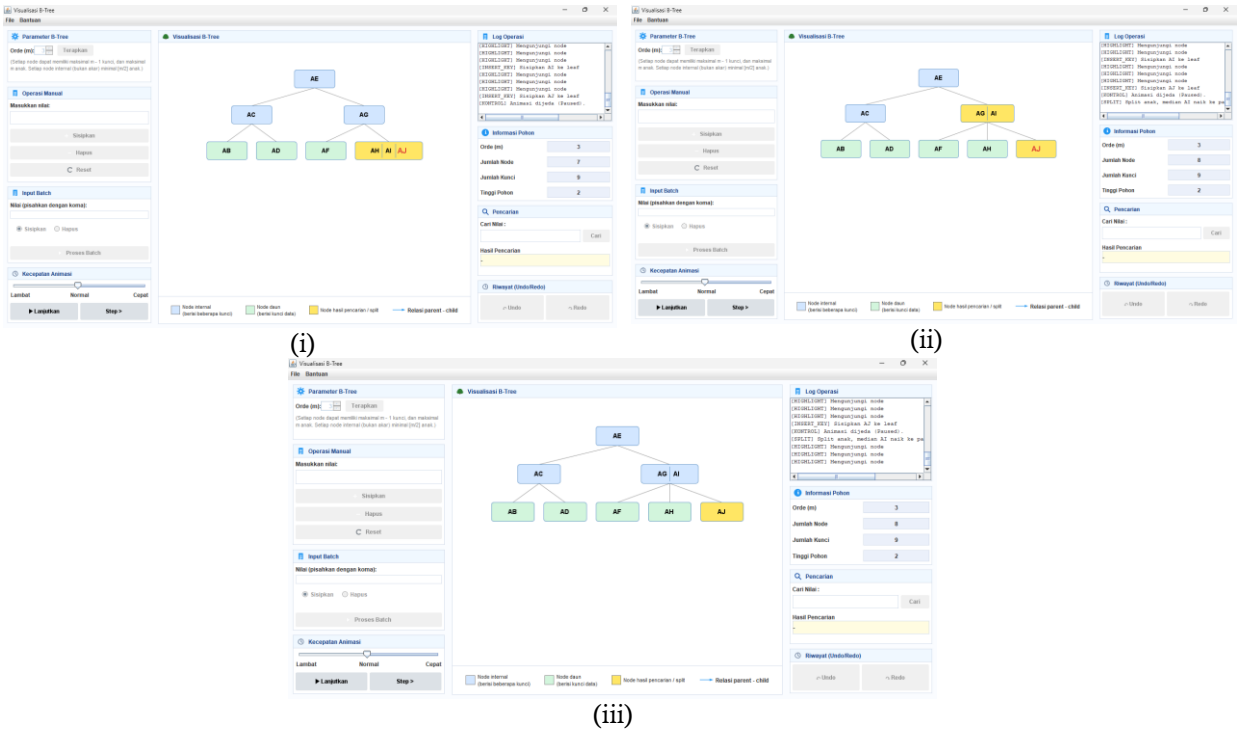


Figure 3. Animation sequence of split process, steps i, ii, and iii

Position against the previous simulator (Table 1). The simulator differs in order ranges 3–10 (USFCA: 3–7), string key support on B-Tree structures (STX B+ Tree supports strings but is B+ tree), and a history of multi-step *undo/redo* operations not found in the comparison simulator. The simulator does not yet support mixed keys in a single tree and has not implemented a B+ tree.

Testing Results and Discussion

Automated testing is run with JUnit 4.13.2. Table 6 summarizes the results of unit testing on *the InputValidator*. All test cases are pass-statused, indicating that the validation rules in Table 5, including the rejection of "007" in string mode and *zero-padding* conversion in numerical mode, are running as planned.

Table 6. Validator Input Test Results (numeric and string mode)

ID	Feedback	Mode	Hope	Actual	Status
V-01	10	Numerical	Valid, to "010"	"010"	Passed
V-02	0	Numerical	Rejected	Rejected	Passed
V-03	1000	Numerical	Rejected	Rejected	Passed
V-04	007	String	Rejected (not alphabetical)	Rejected	Passed
V-05	AX	String	Valid	"AX"	Passed
V-06	A1	String	Rejected	Rejected	Passed
V-07	(blank)	Both	Rejected	Rejected	Passed

Table 7. Invariant Verification and Test Results with TreeSet

Mode	Orde	Number of scenarios	Number of operations	Invariant violation	TreeSet *Compatibility*
Numerical	3–10	1.600	480.000	0	100%
String	3–10	1.600	480.000	0	100%

Table 8 summarizes functional testing via interface (*black-box*) and core operation response time measurement (NF-02). Measurements with *System.nanoTime()* after JVM heating showed that the B-Tree core operation was completed well below the 100 ms threshold. Interface depiction time was not included in these measurements.

Table 8. Functional Test Results (black-box) and performance (NF-02)

ID	Scenario	Hope	Response Time	Status
F-01	Insert 15 ascending numbers (10, 20, ..., 150) on order 3	Tree height = 3	2 ms	Passed
F-02	Remove the internal key (predecessor)	Key replaced, no *underflow*	< 1 ms	Passed
F-03	Find a key that doesn't exist	Displays a "Not Found" message"	< 1 ms	Passed
F-04	*Undo* 3 times after insertion	The tree returns to its original state	< 1 ms	Passed

Fit with the objective. Objective (1), support for two key modes, is achieved: the validator accepts and normalizes numeric keys 1–999 and constricts string keys in the alphabet (Table 6), and both modes generate the same tree content as *the TreeSet* on 960,000 random operations (Table 7). Objective (2), an event-based architecture, is indicated by the division of 25 classes into four layers (Table 4) as well as *the* two-level Observer flow (Figures 1 and 3). Objective (3), systematic verification, is indicated by unit, functional, and invariant test results, noting that these results apply to the test cases being run.

The results provide complementary forms of evidence rather than a single undifferentiated success claim. Table 6 concerns domain validation, Table 7 concerns agreement with a set oracle and B-Tree invariants, and Table 8 concerns a limited selection of end-to-end interface actions and core-operation timing. Agreement with TreeSet checks which keys are present, but the oracle does not itself verify whether the visual animation accurately depicts every intermediate split, borrowing step, or merge. Conversely, an invariant checker can detect an invalid internal structure even when the final key set is correct. The combined approach is therefore stronger than using either check alone, while still leaving animation fidelity, usability, and instructional effectiveness outside the validated outcomes.

Analysis of Findings and Limitations

Zero display error in front (string mode). In the initial version, the regular expression *^0\d+$* was applied globally to the interface, so string keys such as "007" were incorrectly displayed as "7". This error was detected by the *InputValidator unit test* in the early version of case V-04, before the string mode was restricted to the alphabet. Fix: the regular expression was moved into a conditional block to run only in numeric mode.

Duplicate and underflow errors. In the initial version, the algorithm allowed the insertion of double keys, and *the fixUnderflow* call used the wrong index when deleting an internal node. This error was detected by *BTreeDifferentialTest* because *TreeSet* rejected duplicates and kept the sequence. Fix: added the *contains* function and *childToVisit tracking* to the deletion process.

Test power. *TreeSets* that are independent of the B-Tree implementation catch errors that can pass structural checks, such as missing or growing keys even if the tree structure remains valid. In contrast, invariant checkers catch structural violations (such as empty nodes or unlevel leaves) that are not visible from the tree content alone. The two checks complement each other.

Validity and generalizability. The construct measured here is implementation correctness over the stated domain. Repeated randomized tests broaden the collection of operation sequences but cannot mathematically prove that all possible B-Tree states or execution paths are correct. The restricted numeric range, five-character alphabetic strings, eight tree orders, and selected functional scenarios limit generalization to other encodings, larger keys, B+ trees, or concurrent settings. Response-time figures may change with hardware, Java virtual machine behavior, memory pressure, tree size, and graphical rendering. Likewise, the separation into 25 classes describes modular organization but is not in itself a measured maintainability score. These limitations should accompany any claim that the prototype is reliable or easily transferable.

Educational validity is a separate question. Animation controls, operation logs, undo/redo, and a consistent mapping between numeric and lexical ordering may support active exploration, but no student performance, task-completion, accessibility, or usability data were collected. A future controlled or mixed-methods study could define learning objectives, compare equivalent exercises across instructional conditions, collect completion and error measures, use a validated usability scale where appropriate, and report participant selection and ethical procedures. Such research would test pedagogical effectiveness without retrospectively attributing unobserved user outcomes to this technical study.

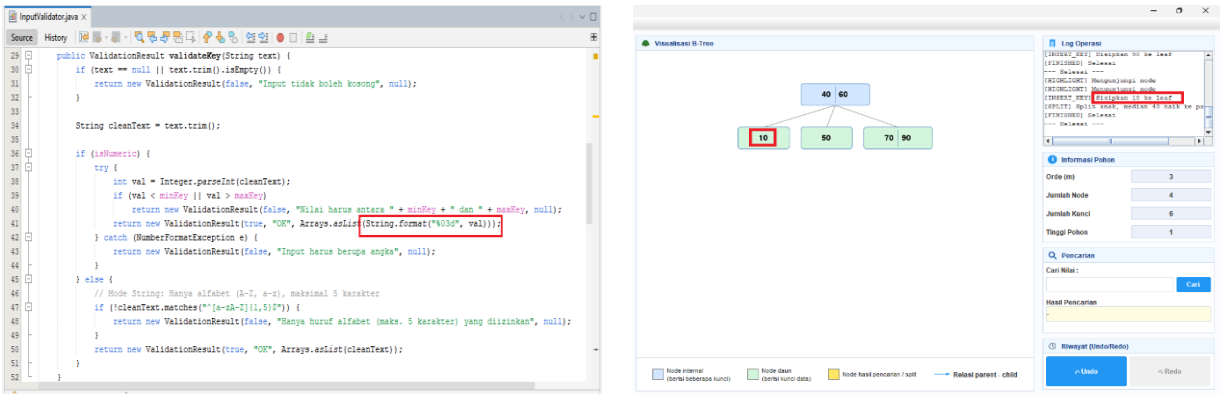


Figure 4. Illustration of Internal Padding ("010") and Interface display ("10")

Limitations. The numerical key range is limited to 1–999; no negative or decimal numbers; the sequence of strings is based on UTF-16 code so that uppercase letters precede lowercase letters; the depiction time and scalability of Swing rendering on more than 10,000 keys have not been tested; and there has been no controlled learning evaluation in users so the educational impact of the simulator is not empirically known. Threats to validity. Internal: developer and tester. Construct: *seed* representation and random keys. External: performance and visual test results apply to Java Swing, JDK 17, and the test tools used.

Conclusions

The Java-based B-Tree visualization simulator was successfully designed and implemented with the support of two key modes, namely numeric and string. Numeric mode uses three-digit *zero-padding* normalization for integers 1–999 so that a *single String.compareTo* comparison path is consistent for both modes. The Model-View-Controller architecture with *Observer* and *Memento* patterns supports step-by-step animation and *undo/redo features*. In the test cases executed, seven validator test cases (JUnit 4) and four functional scenarios were passed, and the response time of the core operation was well below the 100 ms threshold. Differential testing with *TreeSet* on 960,000 random operations (480,000 per mode) found no structural invariant violations or tree content mismatches. The testing process also revealed two critical errors (related to regular expressions and duplicate and *underflow handling*) that were fixed in the final version of the application. The principal contribution is therefore a bounded, reproducible engineering approach: explicit key-domain rules, a single comparator path for distinct modes, event-driven visualization architecture, and complementary invariant and oracle-based verification. Within the documented test configurations, the simulator met the reported correctness and core-response criteria. The findings do not establish superior learning outcomes or usability compared with existing visualization tools; those remain priorities for subsequent empirical evaluation.

References

Ammann, P., & Offutt, J. (2016). *Introduction to software testing* (2nd ed.). Cambridge University Press.

Amazon Web Services. (n.d.). *Zero padding*. In *Amazon SimpleDB developer guide*. https://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/ZeroPadding.html

B+ tree viewer (btview). (n.d.). Minibase documentation. https://itlab.uta.edu/courses/cse5331And4331/mini_doc-2.0/vistools/btview.html

Bayer, R., & McCreight, E. (1972). Organization and maintenance of large ordered indexes. *Acta Informatica, 1*(3), 173–189.

Bingmann, T. (2007). *STX B+ tree*. https://panthema.net/2007/stx-btree/

Comer, D. (1979). The ubiquitous B-tree. *ACM Computing Surveys, 11*(2), 121–137.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

Galles, D. (n.d.). *USFCA data structure visualizations: B-tree*. https://www.cs.usfca.edu/~galles/visualization/BTree.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Halim, S. (n.d.). *VisuAlgo*. https://visualgo.net

Hundhausen, C., Douglas, S., & Stasko, J. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing, 13*(3), 259–290.

Knuth, D. E. (1998). *The art of computer programming: Vol. 3. Sorting and searching* (2nd ed.). Addison-Wesley.

Liu, J., Poulsen, S., Goodwin, E., Chen, H., Williams, G., Gertner, Y., & Franklin, D. (2025). Teaching algorithm design: A literature review. *ACM Transactions on Computing Education, 25*(2). https://doi.org/10.1145/3727987

MacIver, D. R., Hatfield-Dodds, Z., & Many Other Contributors. (2019). Hypothesis: A new approach to property-based testing. *Journal of Open Source Software, 4*(43), Article 1891. https://doi.org/10.21105/joss.01891

OpenDSA Project. (n.d.). *OpenDSA: Interactive data structures and algorithms eTextbook*. Virginia Tech. https://opendsa.org/OpenDSA/Books/Everything/html/BTree.html

Oracle. (n.d.-a). *Formatter (Java SE 17 API documentation)*. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Formatter.html

Oracle. (n.d.-b). *String (Java SE 17 API documentation)*. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html

PostgreSQL Global Development Group. (n.d.-a). *Operator classes and operator families*. In *PostgreSQL 18 documentation*. https://www.postgresql.org/docs/18/indexes-opclass.html

PostgreSQL Global Development Group. (n.d.-b). *String functions and operators*. In *PostgreSQL 18 documentation*. https://www.postgresql.org/docs/18/functions-string.html

Seymour, P. (n.d.). *Efficient lexicographic encoding of numbers (ELEN)* [Unpublished manuscript]. https://zanopha.com/docs/elen.pdf

Shaffer, C. A., Cooper, M. L., Alon, A. J. D., Akbar, M., Stewart, M., Ponce, S., & Edwards, S. H. (2010). Algorithm visualization: The state of the field. *ACM Transactions on Computing Education, 10*(3), Article 9. https://doi.org/10.1145/1821996.1821997

Sibia, N., Liut, M., & Nobre, C. (2025). Exploring the role of visualization tools in enhancing computing education: A systematic literature review. In *EuroVis 2025—Education papers*. The Eurographics Association. https://doi.org/10.2312/eved.20251027