Practical Applications & Projects

Building Real-World Systems with Data Structures

Minseok Jeon
DGIST

November 2, 2025

Outline

Introduction

Course Overview

Learning by Building

Apply data structures concepts by implementing complete, working systems

Projects Covered:

- Text editor with undo/redo (Stack)
- Simple database index (B-Trees)
- Social network graph analysis (Graphs)
- Autocomplete engine (Tries)
- Memory allocator simulator (Free lists)
- Project structuring and testing

Why Practical Projects?

Benefits:

- **Deeper Understanding**: See how data structures solve real problems
- **Design Skills**: Learn to choose appropriate structures
- **Integration**: Combine multiple data structures effectively
- **Testing**: Develop comprehensive testing strategies
- **Portfolio**: Build projects for interviews and resumes

Key Principle

Every project demonstrates a core data structure in a practical context

Text Editor with Undo/Redo

Project 1: Text Editor with Undo/Redo

Project Requirements:

- Basic text operations: insert, delete, replace
- Undo last operation
- Redo previously undone operation
- Multiple undo/redo levels
- Clear redo stack on new operations

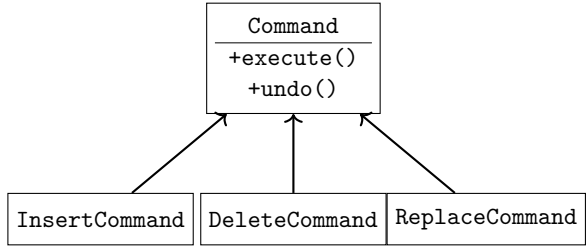
Core Data Structure:

- **Stack** for undo/redo
- Two stacks: undo stack and redo stack
- Command pattern for operations

Complexity:

- Undo/Redo: $O(1)$
- Operations: $O(n)$ worst case

Design: Command Pattern



Executed operations

Undo Stack

Command 3
Command 2
Command 1

Undone operations

Redo Stack

Command 4
Command 5

Implementation: Command Interface

```python
from abc import ABC, abstractmethod

class Command(ABC):
    """Abstract base class for text editor commands."""

    @abstractmethod
    def execute(self, text: List[str]) -> None:
        """Execute the command."""
        pass

    @abstractmethod
    def undo(self, text: List[str]) -> None:
        """Undo the command."""
        pass

class InsertCommand(Command):
    def __init__(self, position: int, text: str):
        self.position = position
        self.text = text

    def execute(self, text: List[str]) -> None:
        for i, char in enumerate(self.text):
            text.insert(self.position + i, char)

    def undo(self, text: List[str]) -> None:
        for _ in range(len(self.text)):
            text.pop(self.position)
```

Implementation: TextEditor Class

```python
class TextEditor:
    def __init__(self):
        self.text: List[str] = []
        self.undo_stack: List[Command] = []
        self.redo_stack: List[Command] = []

    def execute(self, command: Command) -> None:
        command.execute(self.text)
        self.undo_stack.append(command)
        self.redo_stack.clear()  # Clear redo on new operation

    def undo(self) -> bool:
        if not self.undo_stack:
            return False
        command = self.undo_stack.pop()
        command.undo(self.text)
        self.redo_stack.append(command)
        return True

    def redo(self) -> bool:
        if not self.redo_stack:
            return False
        command = self.redo_stack.pop()
        command.execute(self.text)
        self.undo_stack.append(command)
        return True
```

Text Editor: Key Design Decisions

Stack Operations:

- Push to undo stack on execute
- Pop from undo, push to redo on undo
- Pop from redo, push to undo on redo
- Clear redo stack on new operation

Extensions:

- Compound commands (macro recording)
- History size limit
- Save state detection
- Text statistics
- Find and replace all

Testing Strategy

- Test each command type independently
- Test undo/redo sequences
- Test redo clearing on new operation
- Test edge cases (empty stack, invalid positions)

Database Index with B-Trees

Project 2: Database Index using B-Trees

Project Requirements:

- Store key-value pairs with sorted keys
- Insert, search, delete operations
- Handle large datasets efficiently
- Maintain balance automatically
- Support range queries

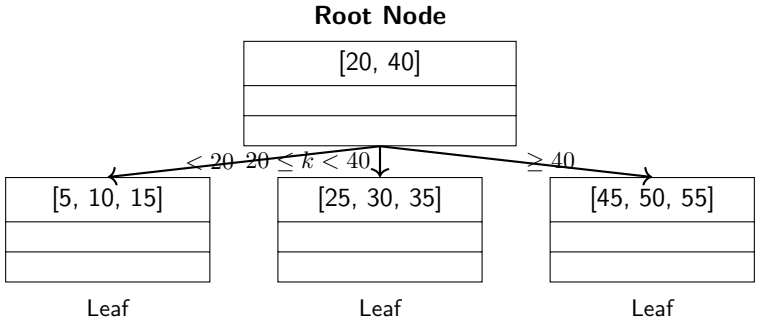
Core Data Structure:

- **B-Tree** (order t)
- Each node: $t - 1$ to $2t - 1$ keys
- Self-balancing
- Disk-friendly (minimize I/O)

Complexity:

- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$

B-Tree Structure



B-Tree Properties (order $t = 3$)

- Each node has $2 \leq$ keys ≤ 5 (except root)
- Keys are sorted within each node
- All leaves at the same level

Implementation: B-Tree Node

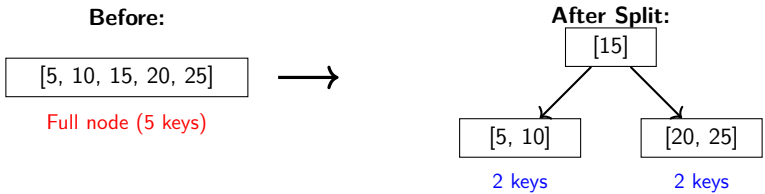
```python
class BTreeNode:
    def __init__(self, leaf=True):
        self.keys = []          # List of keys
        self.values = []        # List of values (for leaf nodes)
        self.children = []      # List of child nodes
        self.leaf = leaf        # Is this a leaf node?

class BTree:
    def __init__(self, t=3):
        """Initialize B-Tree with minimum degree t."""
        self.root = BTreeNode()
        self.t = t  # Each node: t-1 to 2t-1 keys

    def search(self, key, node=None):
        """Search for a key in O(log n) time."""
        if node is None:
            node = self.root

        i = 0
        while i < len(node.keys) and key > node.keys[i]:
            i += 1

        if i < len(node.keys) and key == node.keys[i]:
            return node.values[i] if node.leaf else self.search(key, node.children[i])

        return None if node.leaf else self.search(key, node.children[i])
```

Implementation: B-Tree Insert

```python
def insert(self, key, value):
    root = self.root

    # If root is full, split it
    if len(root.keys) == (2 * self.t) - 1:
        new_root = BTreeNode(leaf=False)
        new_root.children.append(self.root)
        self._split_child(new_root, 0)
        self.root = new_root

    self._insert_non_full(self.root, key, value)

def _split_child(self, parent, index):
    """Split a full child node."""
    t = self.t
    full_child = parent.children[index]
    new_child = BTreeNode(leaf=full_child.leaf)

    mid_index = t - 1
    # Move middle key up to parent
    parent.keys.insert(index, full_child.keys[mid_index])

    # Split keys and values
    new_child.keys = full_child.keys[mid_index + 1:]
    full_child.keys = full_child.keys[:mid_index]
    # ... (similar for values and children)

    parent.children.insert(index + 1, new_child)
```

B-Tree Operations: Visual Example

Inserting 17 into a full node causes split:



- Middle key (15) promoted to parent
- Left child contains smaller keys
- Right child contains larger keys
- Both children have valid number of keys

Database Integration

Simple Database System:

- Primary key index (B-Tree)
- Secondary indexes (multiple B-Trees)
- Insert records with auto-increment ID
- Search by ID or indexed field
- Range queries
- Delete records

Use Cases:

- Database management systems
- File systems (e.g., ext4, NTFS)
- Any sorted data on disk

Why B-Trees?

- Minimize disk I/O
- Nodes = disk blocks
- Shallow tree (high branching factor)
- All leaves at same level

Social Network Graph Analysis

Project 3: Social Network Graph Analysis

Project Requirements:

- Represent users and friendships
- Find degrees of separation (shortest path)
- Suggest friends (mutual connections)
- Detect communities
- Calculate influence metrics
- Clustering coefficient

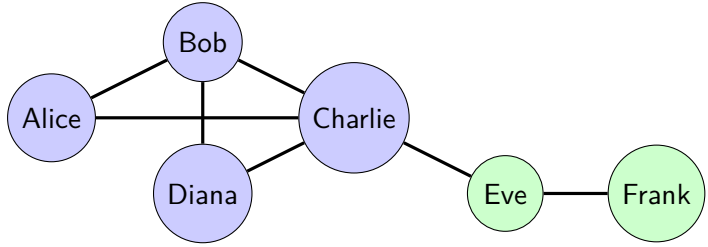
Core Data Structure:

- **Graph** (adjacency list)
- Undirected or directed
- Efficient for sparse graphs

Algorithms:

- BFS for shortest path
- DFS for communities
- PageRank for influence

Social Network: Graph Representation



Community 1

Community 2

Graph Metrics:

- Alice to Frank: 4 degrees of separation (Alice $\rightarrow$ Charlie $\rightarrow$ Eve $\rightarrow$ Frank)
- Bob has highest clustering coefficient (friends are connected)
- Two communities visible

Implementation: Social Network

```python
from collections import defaultdict, deque

class SocialNetwork:
    def __init__(self, directed=False):
        self.users = {}  # user_id -> User object
        self.graph = defaultdict(set)  # adjacency list
        self.directed = directed

    def add_connection(self, user1_id, user2_id):
        """Add friendship/follow relationship."""
        self.graph[user1_id].add(user2_id)
        if not self.directed:
            self.graph[user2_id].add(user1_id)

    def degrees_of_separation(self, user1_id, user2_id):
        """Find shortest path using BFS."""
        if user1_id == user2_id:
            return (0, [user1_id])

        visited = {user1_id}
        queue = deque([(user1_id, [user1_id])])

        while queue:
            current_id, path = queue.popleft()
            for friend_id in self.graph[current_id]:
                if friend_id == user2_id:
                    return (len(path), path + [friend_id])
                if friend_id not in visited:
                    visited.add(friend_id)
                    queue.append((friend_id, path + [friend_id]))
```

Friend Suggestions Algorithm

```python
def suggest_friends(self, user_id, max_suggestions=5):
    """Suggest friends based on mutual connections."""
    current_friends = self.graph[user_id]
    mutual_counts = defaultdict(int)

    # Count mutual friends for non-friends
    for friend_id in current_friends:
        for friend_of_friend_id in self.graph[friend_id]:
            if (friend_of_friend_id != user_id and
                friend_of_friend_id not in current_friends):
                mutual_counts[friend_of_friend_id] += 1

    # Sort by mutual friend count
    suggestions = sorted(
        mutual_counts.items(),
        key=lambda x: x[1],
        reverse=True
    )[:max_suggestions]

    return [(self.users[uid], count) for uid, count in suggestions]
```

Example: Alice is friends with Bob and Charlie. Bob and Charlie are both friends with Diana. $\rightarrow$ Suggest Diana to Alice (2 mutual friends).

Influence Metrics: PageRank

PageRank Algorithm:

- Iterative computation
- User's influence = sum of friends' influence / their friend count
- Damping factor (0.85)
- Converges after iterations

Formula:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in \text{in}(u)} \frac{PR(v)}{|\text{out}(v)|}$$

where $d = 0.85$, N = number of users

Clustering Coefficient:

Measures how connected a user's friends are to each other.

$$C(u) = \frac{\text{actual connections}}{\text{possible connections}}$$

Interpretation:

- $C = 1$: All friends know each other (tight community)
- $C = 0$: No friends know each other (bridge user)
- High clustering $\rightarrow$ local community

Community Detection

Algorithm: Connected Components

1. Start DFS from unvisited user
2. Mark all reachable users as one community
3. Repeat until all users visited

Complexity:

- Time: $O(V + E)$
- Space: $O(V)$

Advanced Methods:

- Girvan-Newman (edge betweenness)
- Louvain method (modularity optimization)
- Label propagation

Applications:

- Recommend groups
- Targeted advertising
- Influence propagation
- Network analysis

Autocomplete Engine

Project 4: Autocomplete Engine

Project Requirements:

- Insert words with frequencies
- Search for words by prefix
- Suggest top-k completions
- Dynamic updates
- Handle large dictionaries

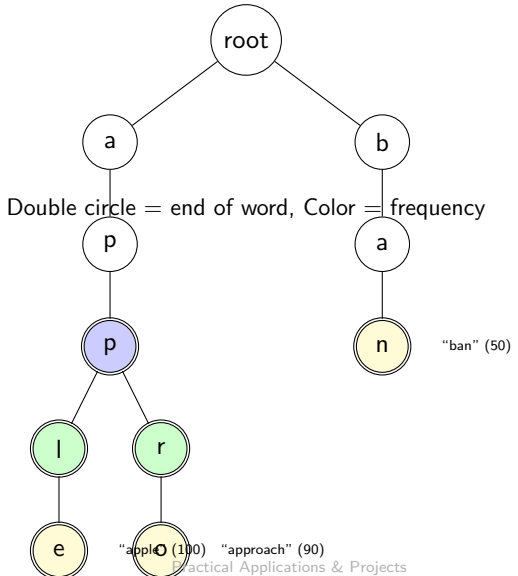
Core Data Structure:

- **Trie** (prefix tree)
- Frequency tracking at nodes
- Priority queue for top-k

Complexity:

- Insert: $O(m)$ where $m =$ word length
- Search: $O(m)$
- Autocomplete: $O(m + n)$ where $n =$ results

Trie Structure for Autocomplete



Implementation: Trie Node and Insert

```python
class TrieNode:
    def __init__(self):
        self.children = {}  # char -> TrieNode
        self.is_end_of_word = False
        self.frequency = 0
        self.word = None

class AutocompleteEngine:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str, frequency: int = 1):
        """Insert word with frequency."""
        node = self.root
        for char in word.lower():
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]

        node.is_end_of_word = True
        node.word = word
        node.frequency += frequency

    def _find_node(self, prefix: str):
        """Find node corresponding to prefix."""
        node = self.root
        for char in prefix:
            if char not in node.children:
                return None
            node = node.children[char]
        return node
```

Autocomplete Algorithm

```python
def autocomplete(self, prefix: str, max_suggestions: int = 10):
    """Get autocomplete suggestions for prefix."""
    prefix = prefix.lower()
    node = self._find_node(prefix)

    if node is None:
        return []

    # Collect all words with this prefix
    suggestions = []
    self._collect_words(node, suggestions)

    # Sort by frequency and return top suggestions
    suggestions.sort(key=lambda x: x[1], reverse=True)
    return suggestions[:max_suggestions]

def _collect_words(self, node, suggestions):
    """Recursively collect all words from node."""
    if node.is_end_of_word:
        suggestions.append((node.word, node.frequency))

    for child in node.children.values():
        self._collect_words(child, suggestions)
```

Example: For prefix "app", collect all descendants: "apple" (100), "application" (80), "approach" (90). Sort by frequency and return top 10.

Autocomplete: Advanced Features

Optimizations:

- Min-heap for top-k (avoid sorting all)
- Store top-k at each node (cache)
- Compressed tries (radix tree)
- Limit recursion depth

Personalization:

- User-specific frequency
- Recent searches
- Context-aware suggestions
- Location-based

Fuzzy Matching:

- Allow typos (edit distance)
- Suggest corrections
- Phonetic matching

Real-World Applications:

- Search engines (Google, Bing)
- Code editors (IDEs)
- Mobile keyboards
- E-commerce search
- Command-line interfaces

Memory Allocator Simulator

Project 5: Memory Allocator Simulator

Project Requirements:

- Allocate memory blocks
- Free allocated blocks
- Coalesce adjacent free blocks
- Handle fragmentation
- Track usage statistics
- Multiple allocation strategies

Core Data Structure:

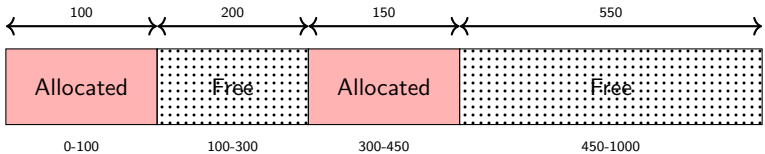
- **Free list** (linked list)
- Hash table for allocated blocks
- Doubly-linked for coalescing

Strategies:

- First-fit: $O(n)$
- Best-fit: $O(n)$
- Worst-fit: $O(n)$

Memory Layout Visualization

Memory Map (1000 bytes total):



Allocation Request (50 bytes):

- **First-fit**: Use free block at 100-300 (first available)
- **Best-fit**: Use free block at 100-300 (smallest that fits)
- **Worst-fit**: Use free block at 450-1000 (largest)

Implementation: Memory Block

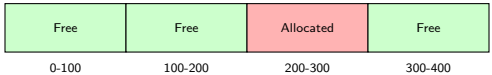
```python
class MemoryBlock:
    def __init__(self, start: int, size: int, is_free: bool = True):
        self.start = start
        self.size = size
        self.is_free = is_free
        self.next = None  # Next block in list
        self.prev = None  # Previous block in list

    @property
    def end(self):
        return self.start + self.size

class MemoryAllocator:
    def __init__(self, total_size: int, strategy):
        self.total_size = total_size
        self.strategy = strategy
        self.head = MemoryBlock(0, total_size, is_free=True)
        self.allocated_blocks = {}  # address -> block

    def malloc(self, size: int):
        """Allocate memory block."""
        block = self._find_free_block(size)
        if block is None:
            return None  # Allocation failed
        if block.size > size:
            self._split_block(block, size)
        block.is_free = False
        self.allocated_blocks[block.start] = block
        return block.start
```

Coalescing Adjacent Free Blocks

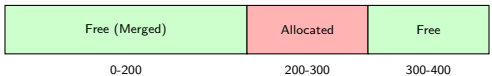
```python
def free(self, address: int):
    """Free allocated block."""
    if address not in self.allocated_blocks:
        return False

    block = self.allocated_blocks[address]
    del self.allocated_blocks[address]
    block.is_free = True

    # Coalesce with adjacent free blocks
    self._coalesce(block)
    return True

def _coalesce(self, block):
    """Merge adjacent free blocks."""
    # Coalesce with next block
    if block.next and block.next.is_free:
        block.size += block.next.size
        block.next = block.next.next
        if block.next:
            block.next.prev = block

    # Coalesce with previous block
    if block.prev and block.prev.is_free:
        block.prev.size += block.size
        block.prev.next = block.next
        if block.next:
            block.next.prev = block.prev
```

Coalescing Example

Before Coalescing:



After Coalescing:



Benefits:

- Reduces fragmentation
- Creates larger free blocks
- Improves allocation success rate
- Essential for long-running systems

Allocation Strategy Comparison

Strategy	Speed	Fragmentation	Use Case
First-Fit	Fast	Moderate	General purpose
Best-Fit	Slow	Low	Memory constrained
Worst-Fit	Slow	High	Large allocations

Trade-offs:

- **First-Fit**: Fast but may create small unusable fragments at start
- **Best-Fit**: Minimizes wasted space but creates tiny fragments
- **Worst-Fit**: Keeps large blocks available but wastes space

Real-World: Most allocators use variants of first-fit with segregated free lists for different size classes (e.g., jemalloc, tcmalloc).

Project Structuring and Testing

Project Structure Best Practices

Directory Layout:

- `src/` - Source code
  - `data_structures/`
  - `algorithms/`
  - `utils/`
- `tests/` - Test files
- `docs/` - Documentation
- `requirements.txt`
- `README.md`
- `setup.py`

Principles:

- Separate concerns
- One class per file (large projects)
- Clear naming conventions
- Package initialization files
- Version control (git)

Documentation:

- Docstrings for all public APIs
- README with usage examples
- API reference
- Architecture diagrams

Testing Strategies

Unit Tests:

- Test individual methods
- Isolate dependencies
- Fast execution
- High coverage

Integration Tests:

- Test component interaction
- End-to-end scenarios
- Realistic workloads

Edge Cases:

- Empty inputs
- Maximum sizes
- Invalid inputs

Performance Tests:

- Benchmark operations
- Verify complexity
- Regression testing
- Memory usage

Property-Based Testing:

- Test invariants
- Generate random inputs
- Find edge cases automatically
- Use hypothesis library

Tools:

- `unittest`, `pytest`
- `coverage.py`

Testing Example: Text Editor

```python
import unittest

class TestTextEditor(unittest.TestCase):
    def setUp(self):
        self.editor = TextEditor()

    def test_insert_at_beginning(self):
        self.editor.insert(0, "Hello")
        self.assertEqual(self.editor.get_text(), "Hello")

    def test_undo_redo_sequence(self):
        self.editor.insert(0, "A")
        self.editor.insert(1, "B")
        self.editor.undo()
        self.assertEqual(self.editor.get_text(), "A")
        self.editor.redo()
        self.assertEqual(self.editor.get_text(), "AB")

    def test_redo_cleared_on_new_operation(self):
        self.editor.insert(0, "Test")
        self.editor.undo()
        self.editor.insert(0, "New")
        self.assertFalse(self.editor.can_redo())

    def test_large_text_operations(self):
        large_text = "x" * 10000
        self.editor.insert(0, large_text)
        self.assertEqual(len(self.editor.get_text()), 10000)
```

Code Quality and Best Practices

Code Style:

- Follow PEP 8 (Python)
- Consistent naming
- Clear variable names
- Avoid magic numbers
- Type hints

Error Handling:

- Validate inputs
- Raise appropriate exceptions
- Document error conditions
- Fail fast

Performance:

- Profile before optimizing
- Document complexity
- Avoid premature optimization
- Test performance

Maintenance:

- Regular refactoring
- Keep functions small
- Single responsibility principle
- Version control commits
- Code reviews

Summary

Projects Summary

Project	Core Structure	Key Algorithm
Text Editor	Stack	Command pattern
DB Index	B-Tree	Split/merge
Social Network	Graph	BFS, PageRank
Autocomplete	Trie	Prefix traversal
Memory Allocator	Free List	Coalescing

Common Themes:

- Choose data structure based on operations
- Combine multiple structures when needed
- Test thoroughly (unit, integration, edge cases)
- Measure and document performance
- Maintain clean, readable code

Key Takeaways

1. **Data Structures Enable Solutions**
   - Stacks enable undo/redo naturally
   - B-Trees excel at disk-based sorted data
   - Graphs model relationships and networks
   - Tries optimize prefix-based search
   - Free lists manage dynamic memory

2. **Design Matters**
   - Understand requirements before choosing structures
   - Consider time/space trade-offs
   - Plan for scalability and edge cases

3. **Testing is Essential**
   - Comprehensive tests catch bugs early
   - Property-based tests find unexpected issues
   - Performance tests verify complexity

Building Your Portfolio

Next Steps:

1. **Implement These Projects**
   - Start with text editor (simplest)
   - Work up to memory allocator (most complex)
   - Add your own features and extensions
2. **Extend and Experiment**
   - Add GUI to text editor
   - Implement concurrent B-Tree
   - Add recommendation system to social network
   - Build fuzzy matching for autocomplete
   - Compare allocation strategies empirically
3. **Document and Share**
   - Write clear READMEs
   - Create demonstrations
   - Share on GitHub
   - Discuss in interviews

Additional Project Ideas

More Projects to Try:

- **Task Scheduler**: Priority queue + heap for job scheduling
- **File System Simulator**: Tree + hash table for directories
- **LRU Cache**: Hash table + doubly-linked list
- **Spell Checker**: Trie + edit distance algorithm
- **Git-like Version Control**: DAG + hash table
- **JSON Parser**: Stack for nested structures
- **URL Shortener**: Hash table + base conversion
- **Rate Limiter**: Queue + sliding window
- **Expression Evaluator**: Stack + parsing
- **Prefix Sum Range Queries**: Segment tree or Fenwick tree

Remember

The best way to learn data structures is to use them to solve real problems!

Thank You!

Questions?

"The only way to learn a new programming language is by writing programs in it." – Dennis Ritchie

The same applies to data structures:
learn by building projects!