Buffer Trees

An I/O Optimized Data Structure

Beatriz Aguiar André Pereira João Marinho
Miguel Rodrigues

Faculty of Enginnering
University of Porto

March 6, 2023

Outline

Outline

Motivation

- I/O efficient algorithms are a key problem in the light of algorithm design.
- There is an increased need for handling huge quantities of data, but the communication between fast internal memory and slower external memory poses a bottleneck!
- Demand for external data structures designs, capable of storing large quantities of data, but that can simultaneously take advantage of the large main memory.

The I/O Bound

Currently, the effects of I/O's slowness are much more visible! The rate of improvement of I/O devices is almost an order of magnitude shorter than the one of microprocessors [1].

Outline

The I/O Model

1. The I/O Model suggests the use of three parameters:
   - N = no. of elements in the problem instance
   - M = no. of elements that can fit into main memory
   - B = no. of elements per block
2. Existing constraints:
   - $M < N$
   - $1 \leq B \leq \frac{M}{2}$
3. Additional important definitions:
   - $n = \frac{N}{B}$ (no. of blocks in the problem)
   - $m = \frac{M}{B}$ (no. of blocks that fit into internal memory)

Definition

An I/O operation (or I/O) is a swap of B elements from internal memory with B consecutive elements from external memory (disk).

The (a, b)-tree

Definition

An (a, b)-tree is a balanced search tree that:

- $2 \leq a \leq \frac{b+1}{2}$
- Each internal node, except the root, has at least a children
- All internal nodes, including the root, have at most b children
- All the leaves are at the same depth

The (a, b)-tree

A generalization for balanced search trees

This type of trees are a generalization for balanced search trees, for instance:

- B-Tree is an (a, b)-tree where:
  - $a = \lceil \frac{b}{2} \rceil$
  - Internal nodes may record data
- Buffer tree
  - $a = \frac{m}{4}$
  - $b = m$
  - Only leaves record data
- And many more...

The Buffer Technique

The crux behind the data structure is to perform operations in a *lazy manner* using main-memory-sized $\Theta(m)$ buffers.

- Buffers of size m are assigned to each internal node
- Operations are not immediately performed
- An auxiliary buffer is used to collect operations, which later on are inserted into the root's buffer

This technique allows for several insertions and deletions of the same elements, and therefore **timestamps** are kept when an element is inserted in the main memory.

(a, b) - tree

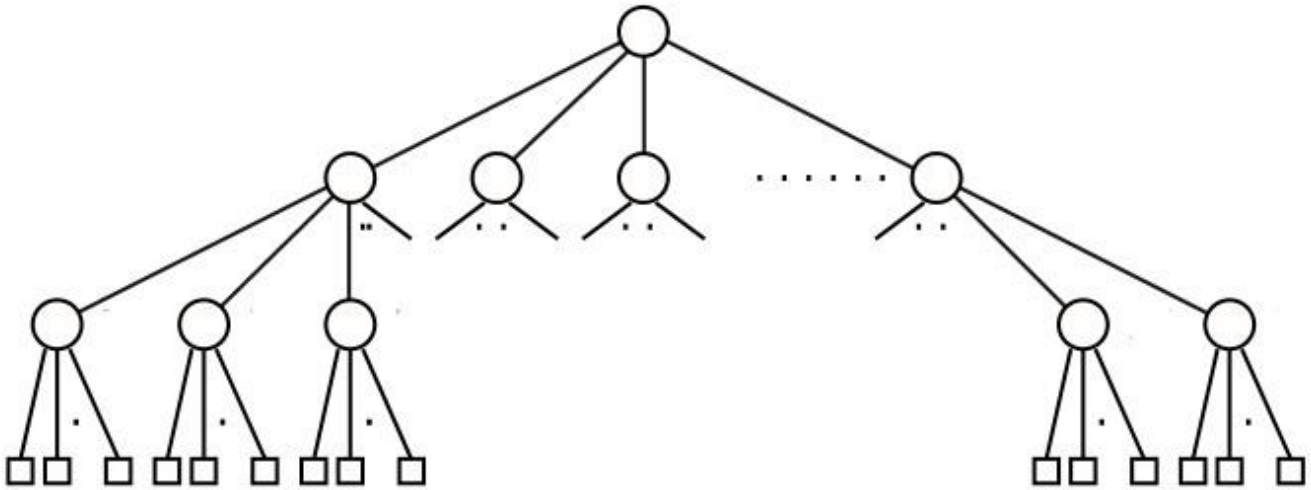

Figure 1: (a, b) - tree

Buffer tree

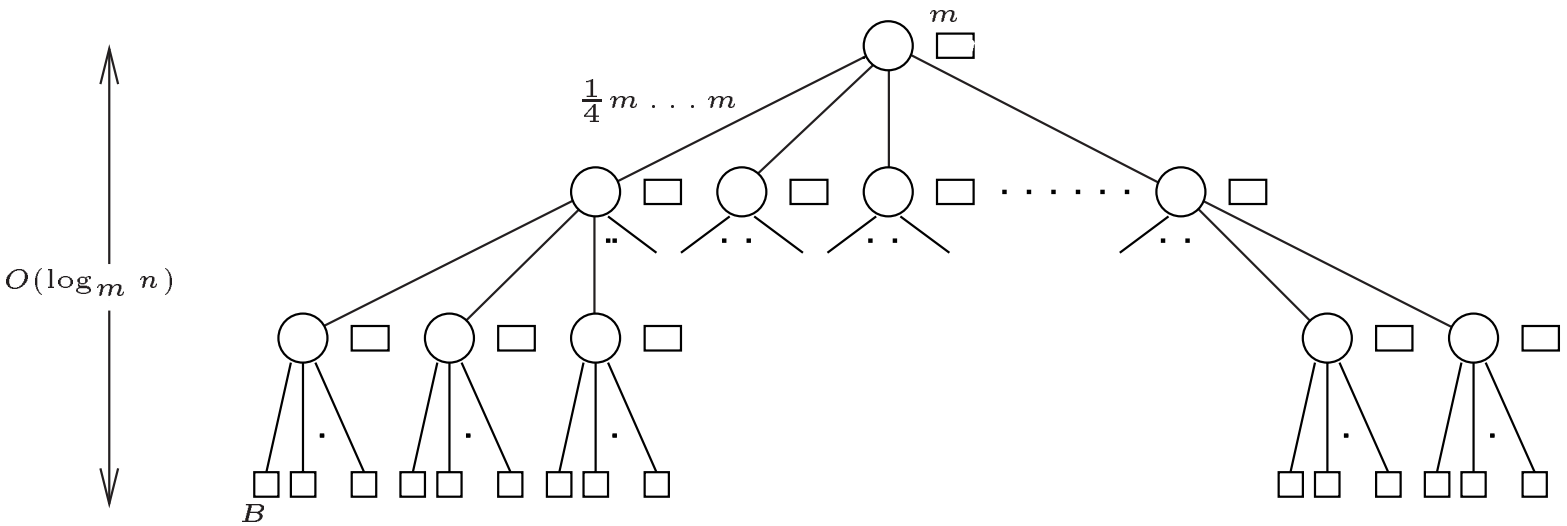

Figure 2: Buffer tree

Outline

Collection

When an operation, either an insertion or a deletion, occurs:

1. Construct an element with:
   - Element's value
   - Timestamp
   - Type of operation (insertion or deletion)
2. Upon collecting B elements in internal memory we insert them in the root's buffer.
3. If root buffer contains more than m blocks we perform `buffer-emptying` process.

Buffer Emptying

- When a buffer reaches the full capacity, all elements are correctly distributed amongst the children nodes, **`buffer-emptying`** process
- Only after finishing all `buffer-emptying` processes on internal nodes do we empty the buffers of the relevant leaf nodes

Buffer Emptying Process

Internal Nodes

1. Load the first M elements in the buffer into internal memory, sort them, and then merge them with the rest of the elements in the buffer. Corresponding insert and delete elements with correct timestamps are deleted
2. Scan through the sorted list while distributing elements to the appropriate buffers one level down
3. If the buffer of any of the children, internal nodes, now contains more than m blocks then recursively apply the buffer emptying process

Buffer Emptying Process

Leaf Nodes

Let v be the leaf node with a full buffer and k the number of leaves of v

1. Sort elements in buffer and remove *matching* operations
2. Merge the sorted list with the elements in the leaves of v while removing matching operations
3. ...

Buffer Emptying Process

Leaf Nodes

- If the number of blocks of elements in the resulting list is **smaller** than k
  1. Place elements in sorted order in the leaves of v
  2. Add dummy-blocks until v has k leaves and update the partition elements of v
- If the number of blocks of elements in the resulting list is **bigger** than k
  1. Place the k smallest blocks in the leaves of v and update the partition elements of v
  2. Repeatedly insert one block of the rest of the elements and rebalance
- Repeatedly delete one dummy block and rebalance

Rebalancing

Remarks

- Before a `rebalance` operation on a node v, a `buffer-emptying` process is performed on the two sibling nodes involved in a `rebalance` operation.
- If the delete results in any leaf buffer becoming full, these buffers are emptied before the next dummy block is deleted.

Insertion on an (a, b)-tree

Rebalancing

Require: **v** has **b + 1** children

1: **if** v is the root **then**
2: let x be a new node and make v its only child
3: **else**
4: let x be the parent of v
5: **end if**
6: Let v' be a new node
7: Let v' be a new child of x immediately after v
8: Split v:
9: Take the rightmost $\lceil \frac{b+1}{2} \rceil$ children from v and make them children of v'
10: Let $v = x$

Deletion from an (a, b)-tree

Rebalancing

Require: **v** hmas **a** - **1** children *AND* **v'** has less than **a** + **t** + **1** children
Fuse **v** and **v'**:
Make all children of v' children of v
Let $v = x$
Let v' be a new sibling of x
if x does not have a sibling (x is the root) *AND* x only has one child **then**
    Delete x
    STOP
end if
Let x be the parent of v
Require: **v** has **a** - **1** children
Share:
Take x children away from v' and make them children of v

Outline

Sorting

Overview

It is possible to sort N elements in $O(N \log N)$ time using a balanced tree.

For a `B-tree`:

- $O(N \log_B N)$ I/Os used

For a Buffer tree:

- $O((\log_m n)/B)$ amortized I/Os on insert and delete
- $O(n)$ I/Os when emptying the buffers at the end

External Priority Queue

Overview

Unlike other search trees, the buffer tree cannot implement a priority queue straightforward.
The smallest element might not be in the leftmost leaf but in the buffer of an internal node of the left-most-leaf path.

External Priority Queue

deletemin

Strategy to perform a `deletemin`:

1. Perform buffer-emptying processes on all nodes from the root to the leftmost leaf.
2. Delete $\frac{mB}{4}$ smallest elements in the tree.
3. Keep these elements in internal memory.

External Priority Queue

deletemin

Strategy to perform a `deletemin`:

1. Perform buffer-emptying processes on all nodes from the root to the leftmost leaf.
2. Delete $\frac{mB}{4}$ smallest elements in the tree.
3. Keep these elements in internal memory.

Amortization

This way we can answer the next $\frac{mB}{4}$ `deletemin` operations without performing any I/Os.

Outline

Example

In he following examples lets assume $m = 8$:

- $a = 2$
- $b = 8$
- $bufferSize = 8$

Insert Example

Figure 3: Insert Example

Insert Example

Figure 3: Insert Example

Insert Example

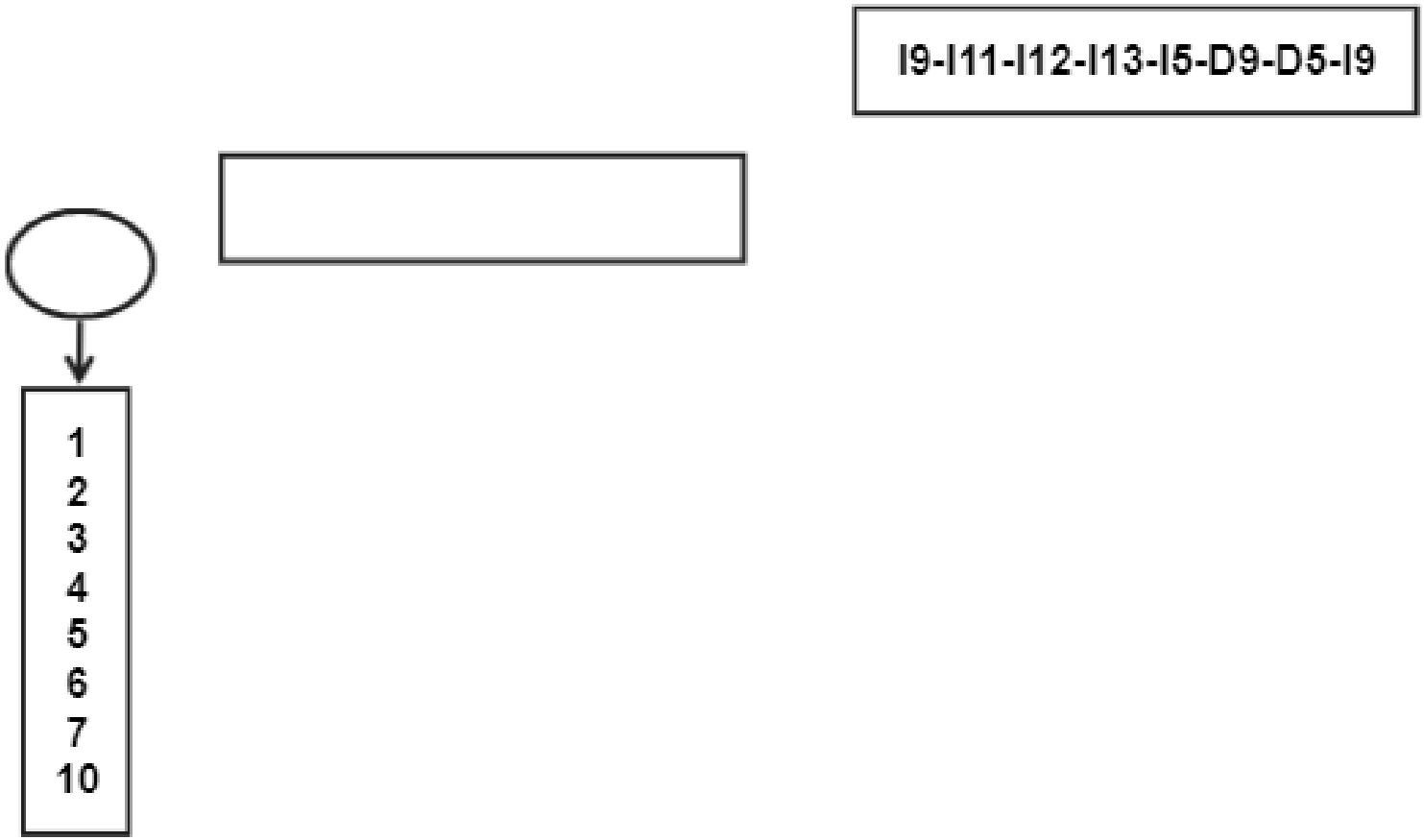

Figure 3: Insert Example

Insert Example

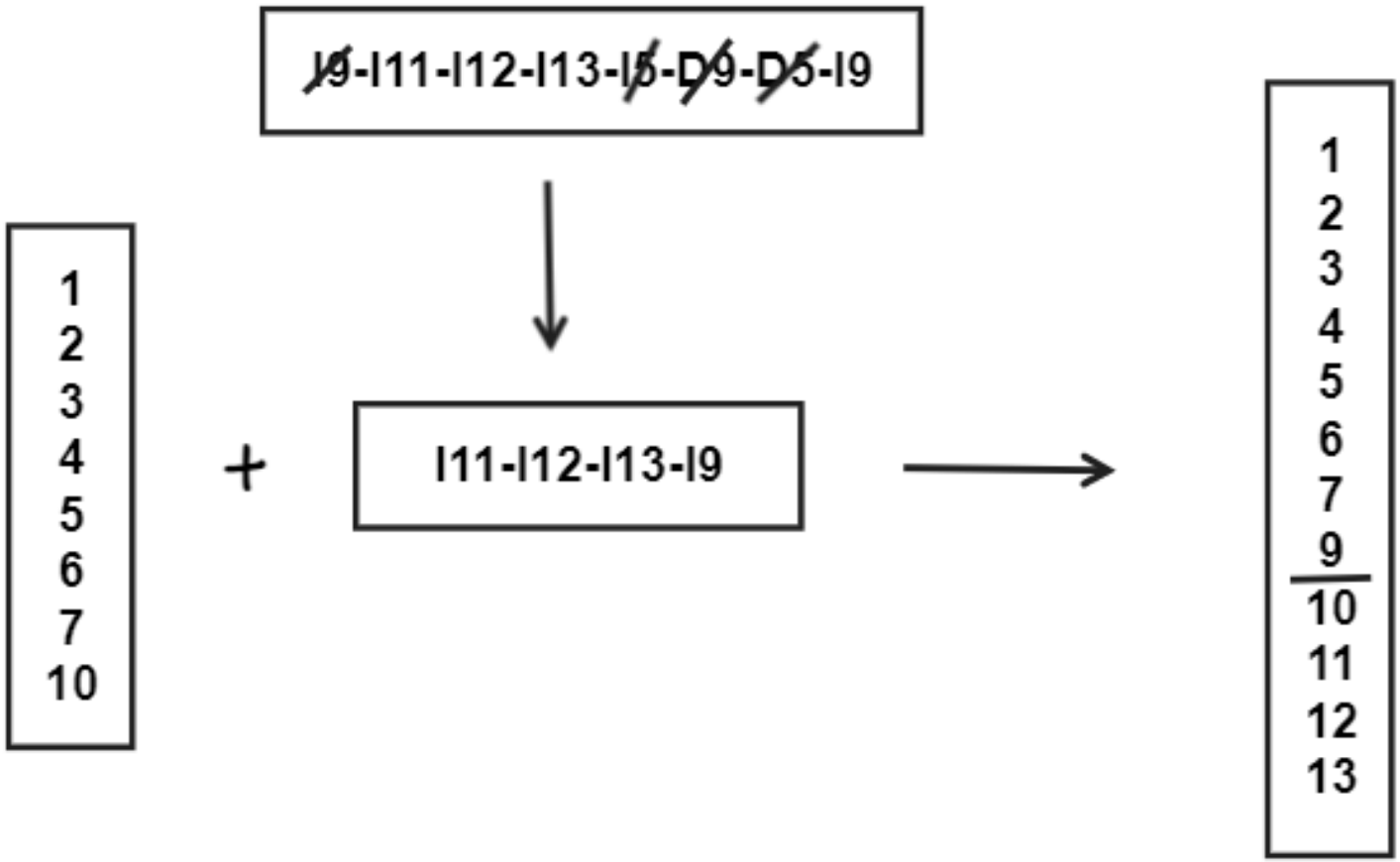

Figure 3: Insert Example

Insert Example

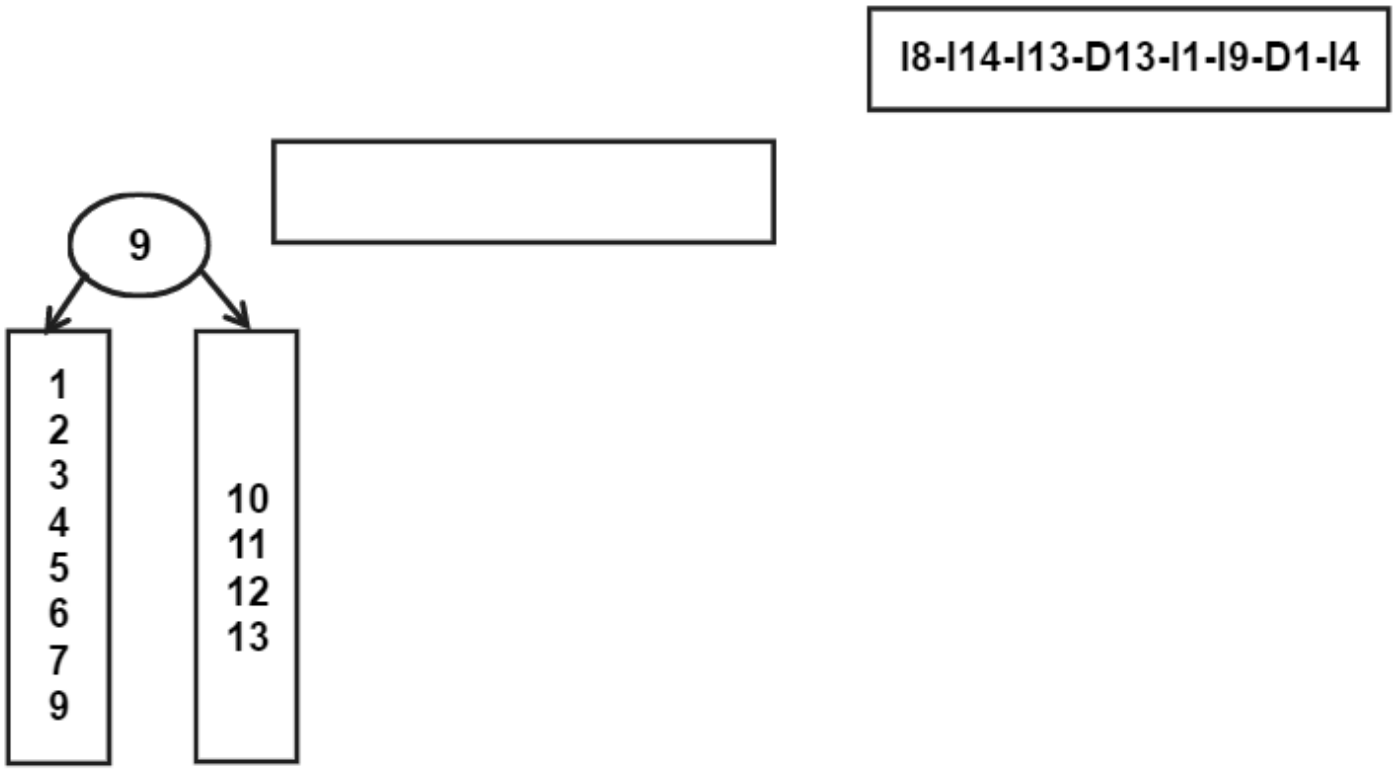

Figure 3: Insert Example

Insert Example

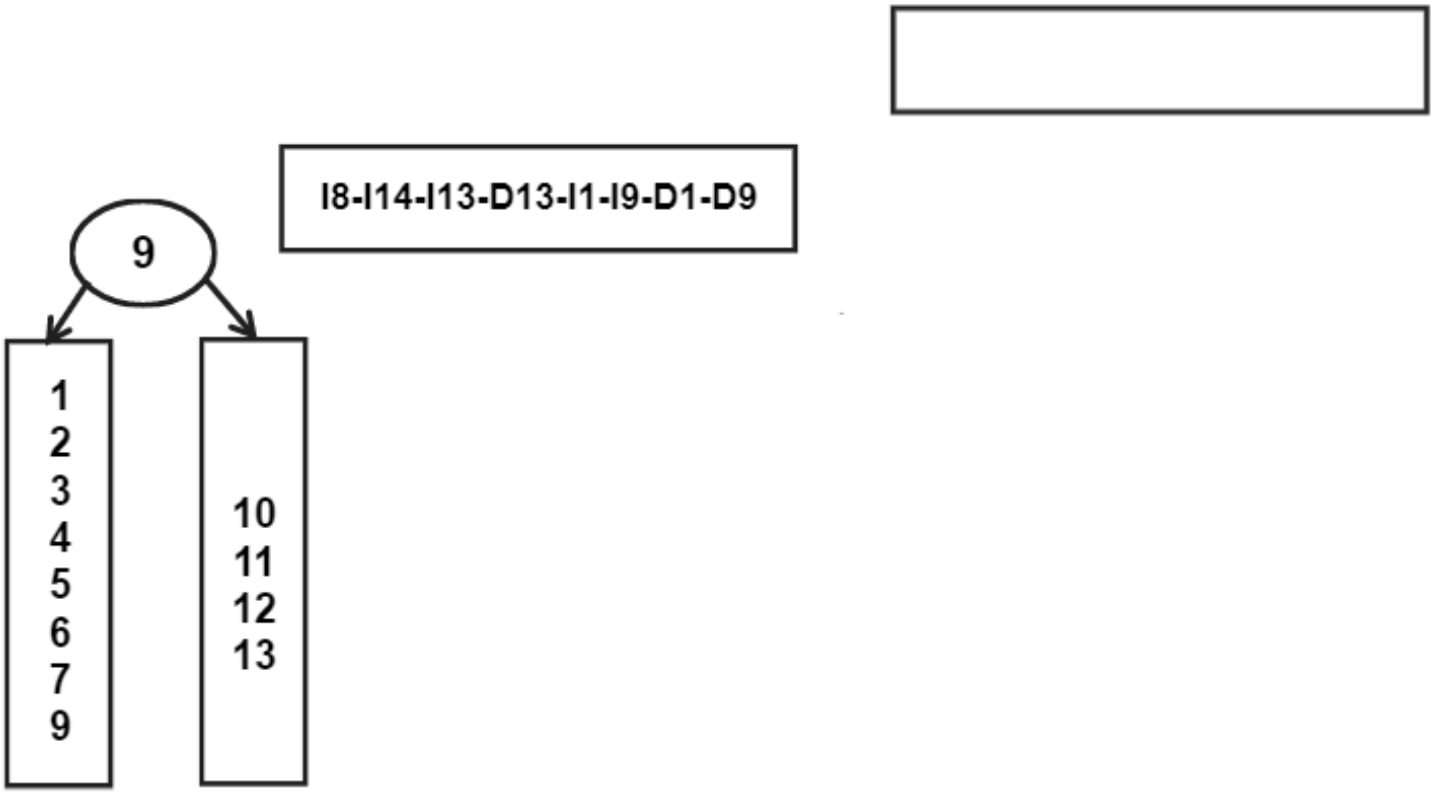

Figure 3: Insert Example

Insert Example

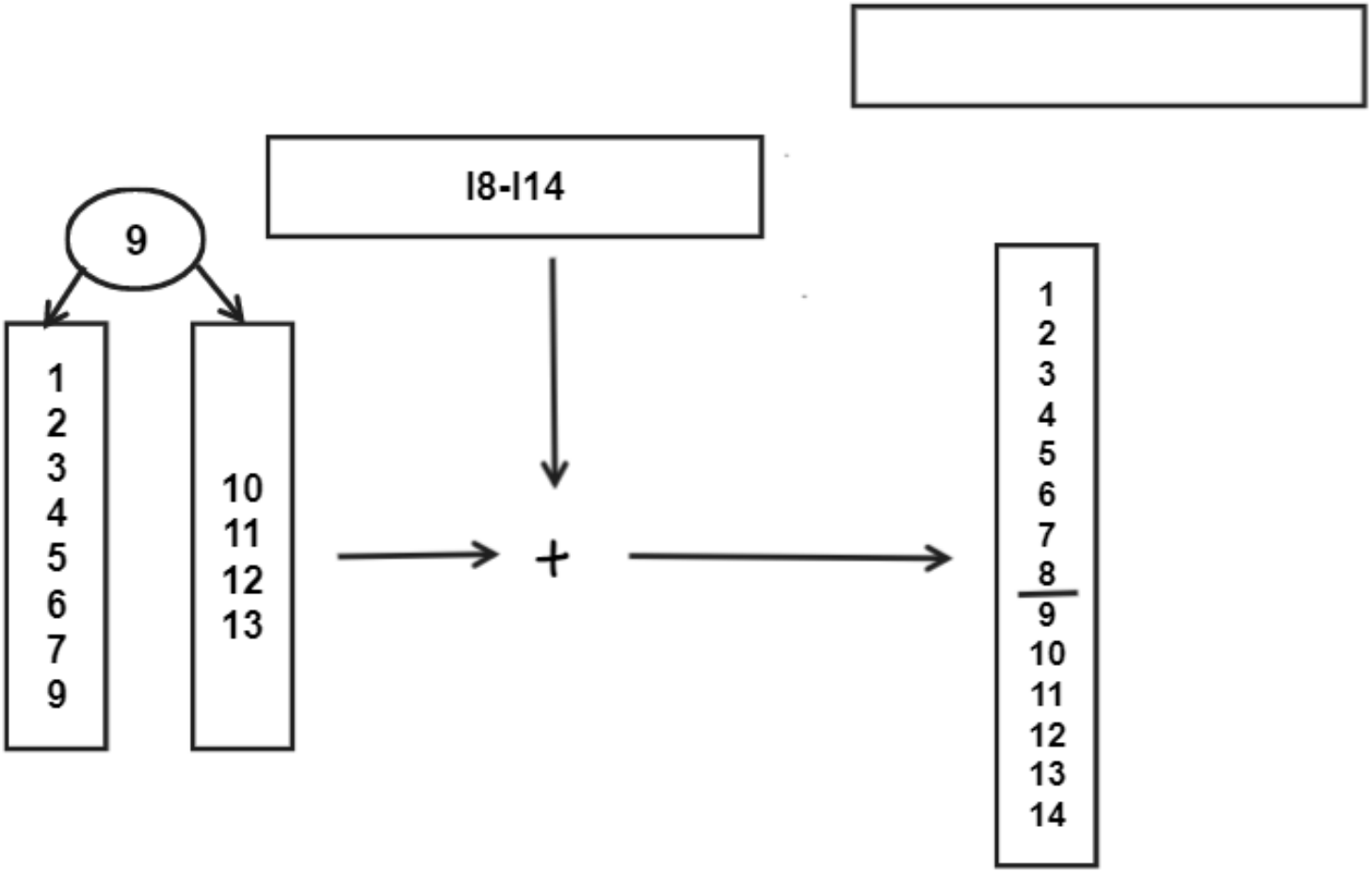

Figure 3: Insert Example

Insert Example

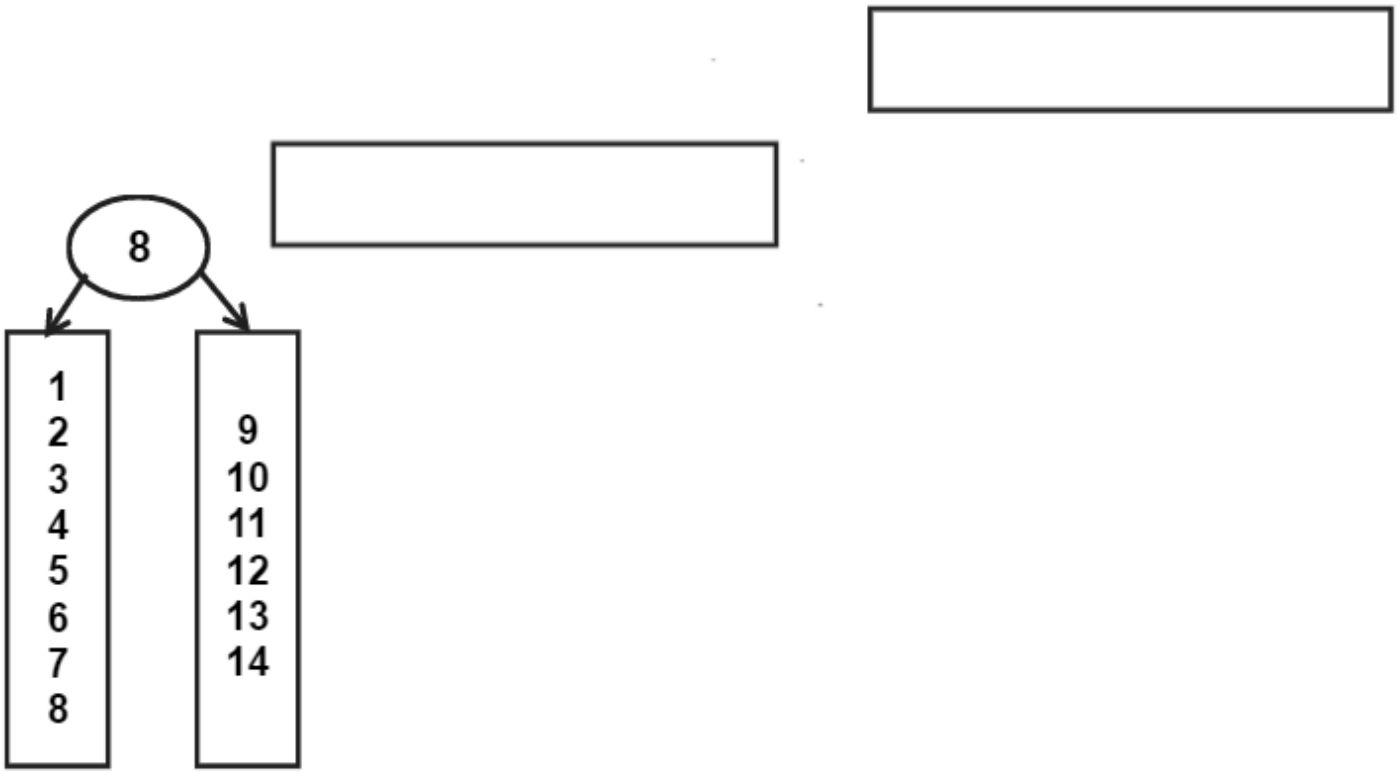

Figure 3: Insert Example

Buffer Distribution Example

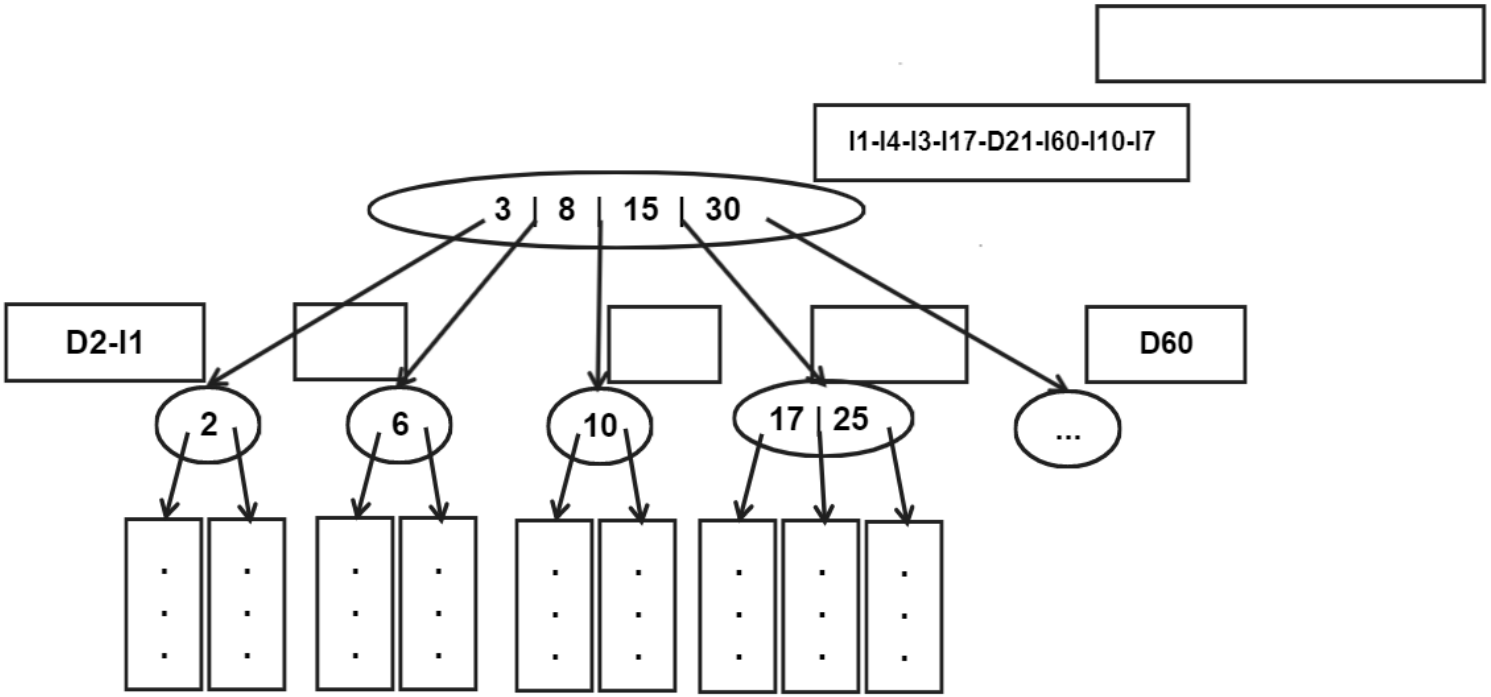

Figure 4: Buffer Distribution Example

Buffer Distribution Example

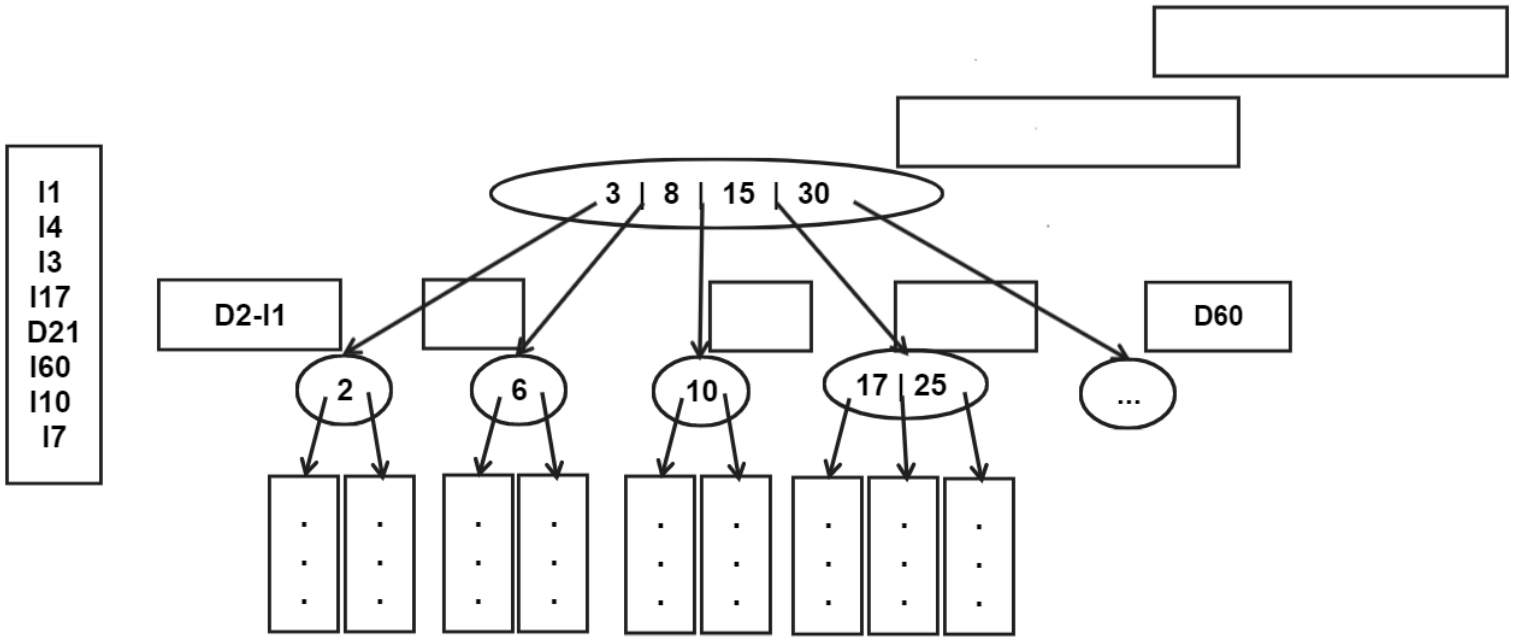

Figure 4: Buffer Distribution Example

Buffer Distribution Example

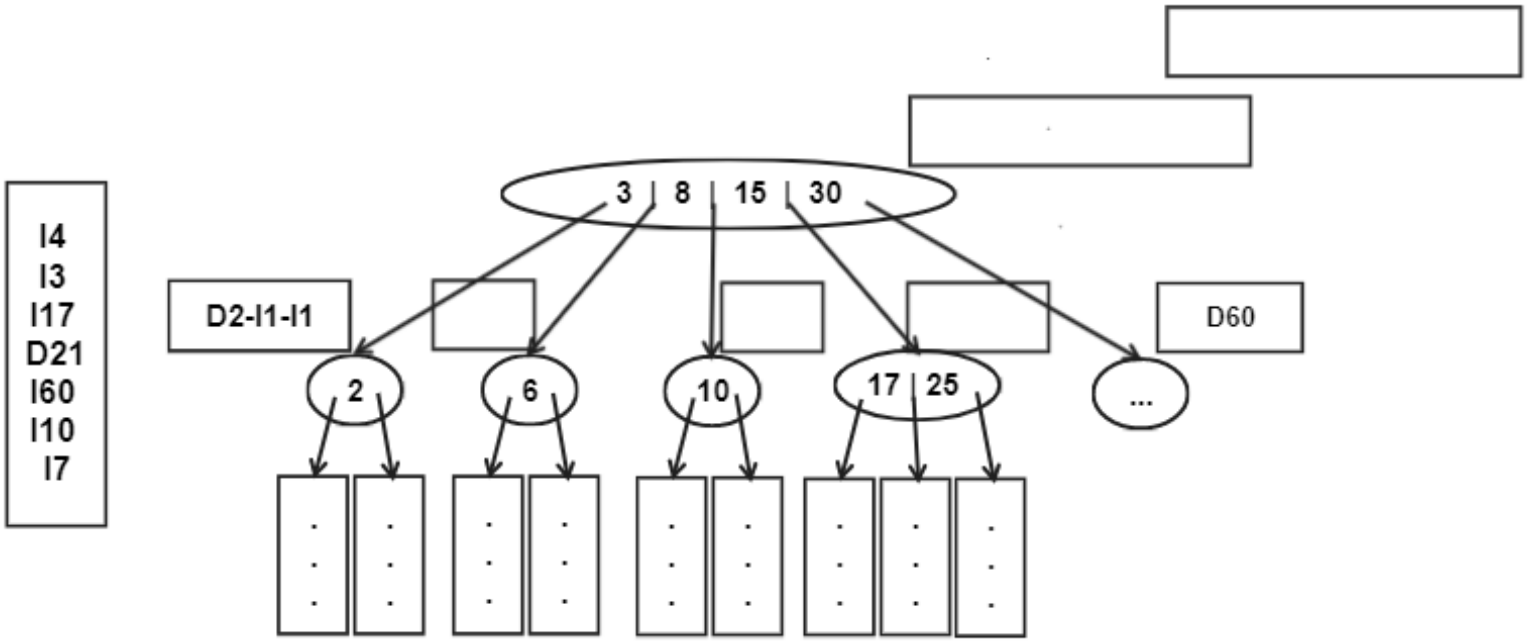

Figure 4: Buffer Distribution Example

Buffer Distribution Example

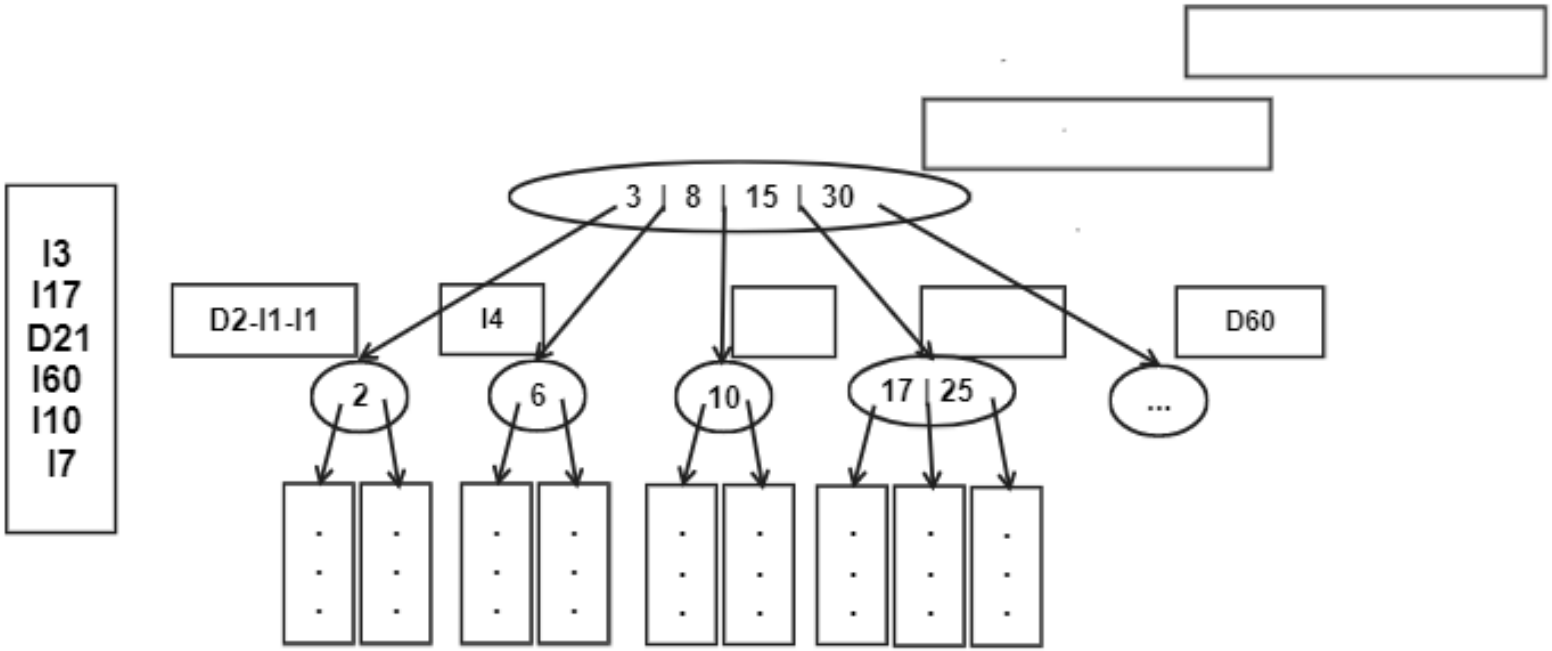

Figure 4: Buffer Distribution Example

Buffer Distribution Example

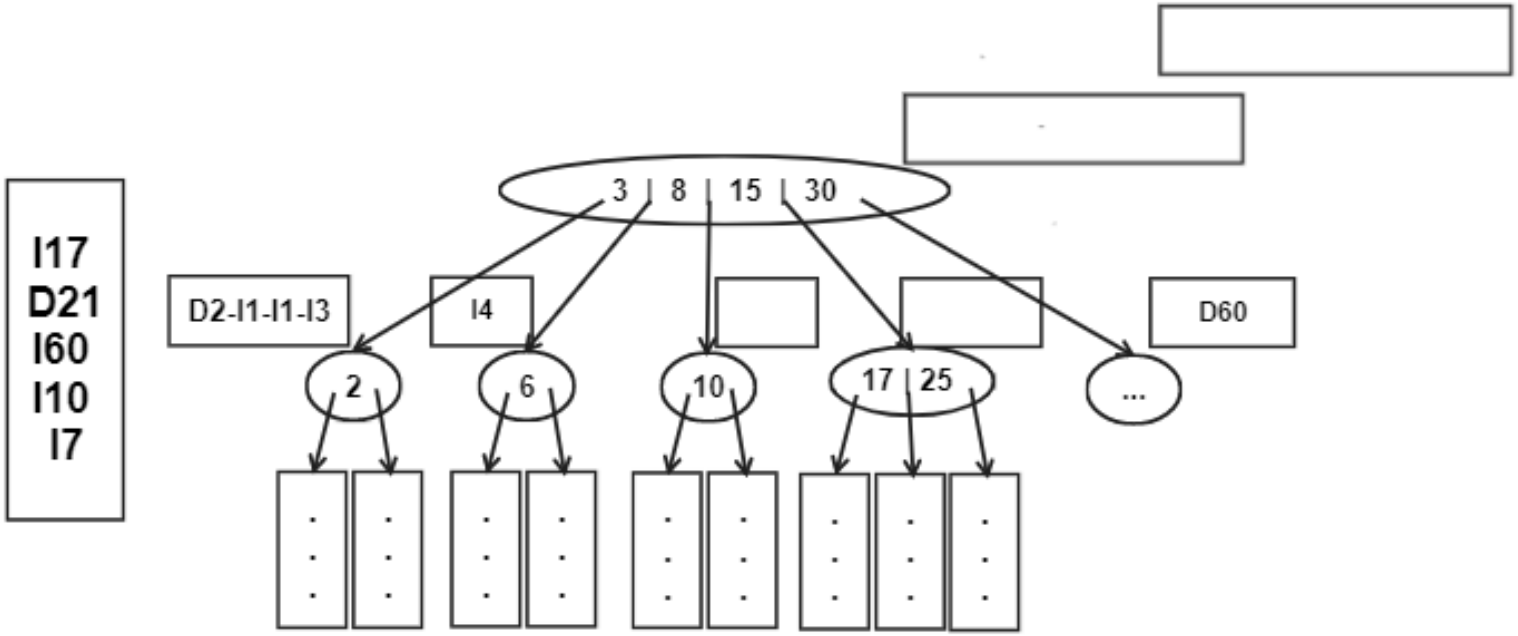

Figure 4: Buffer Distribution Example

Buffer Distribution Example

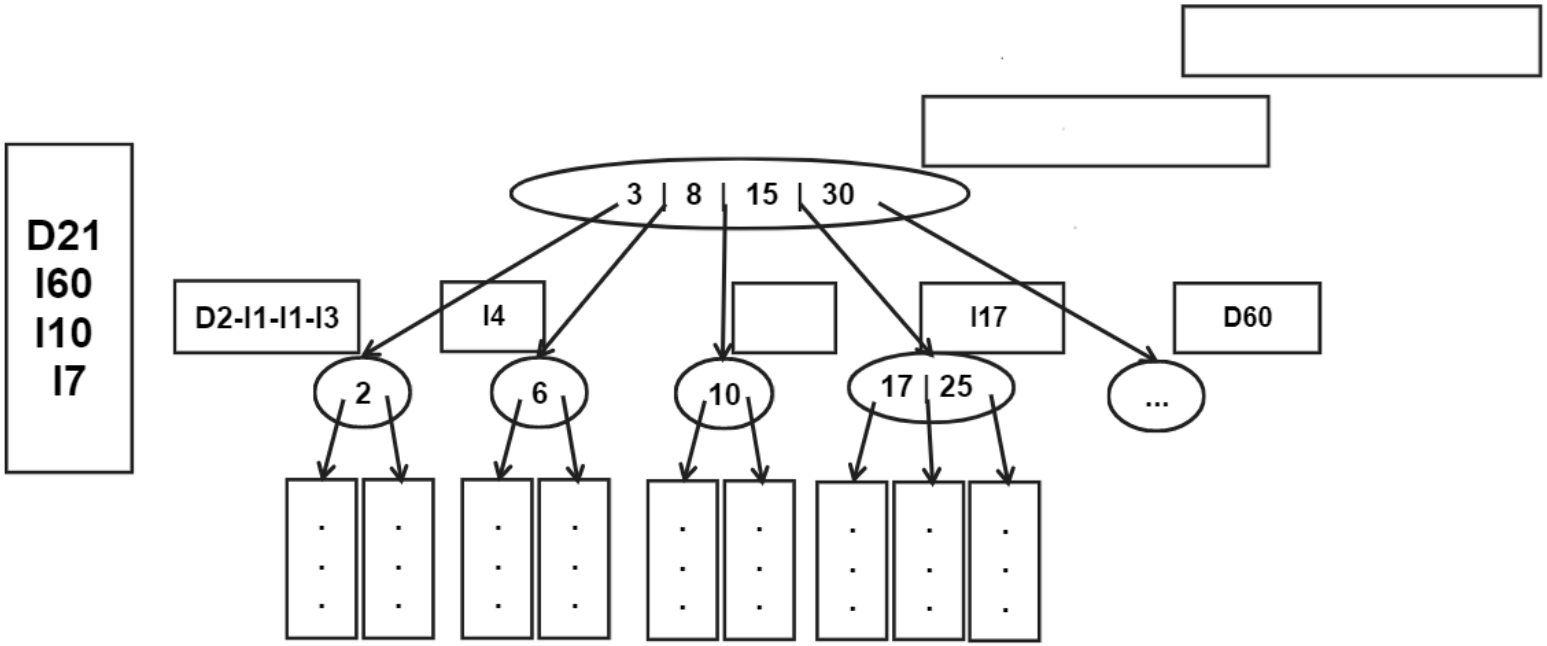

Figure 4: Buffer Distribution Example

Buffer Distribution Example

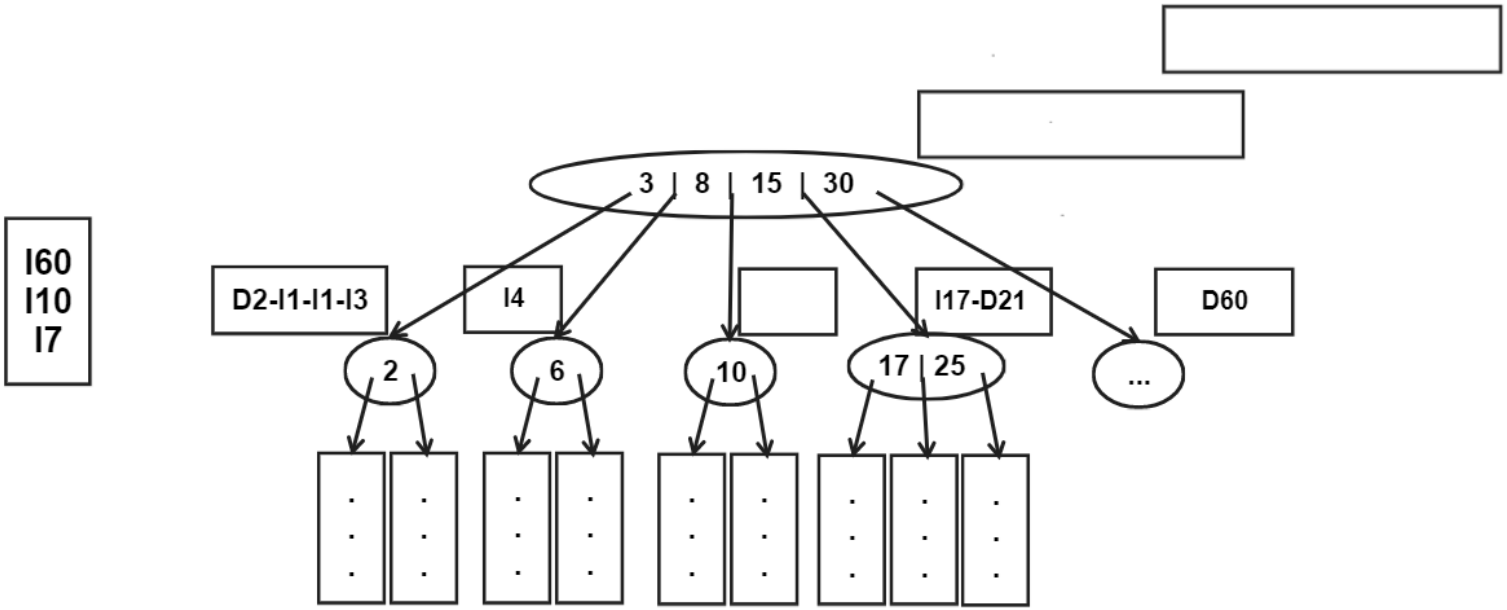

Figure 4: Buffer Distribution Example

Buffer Distribution Example

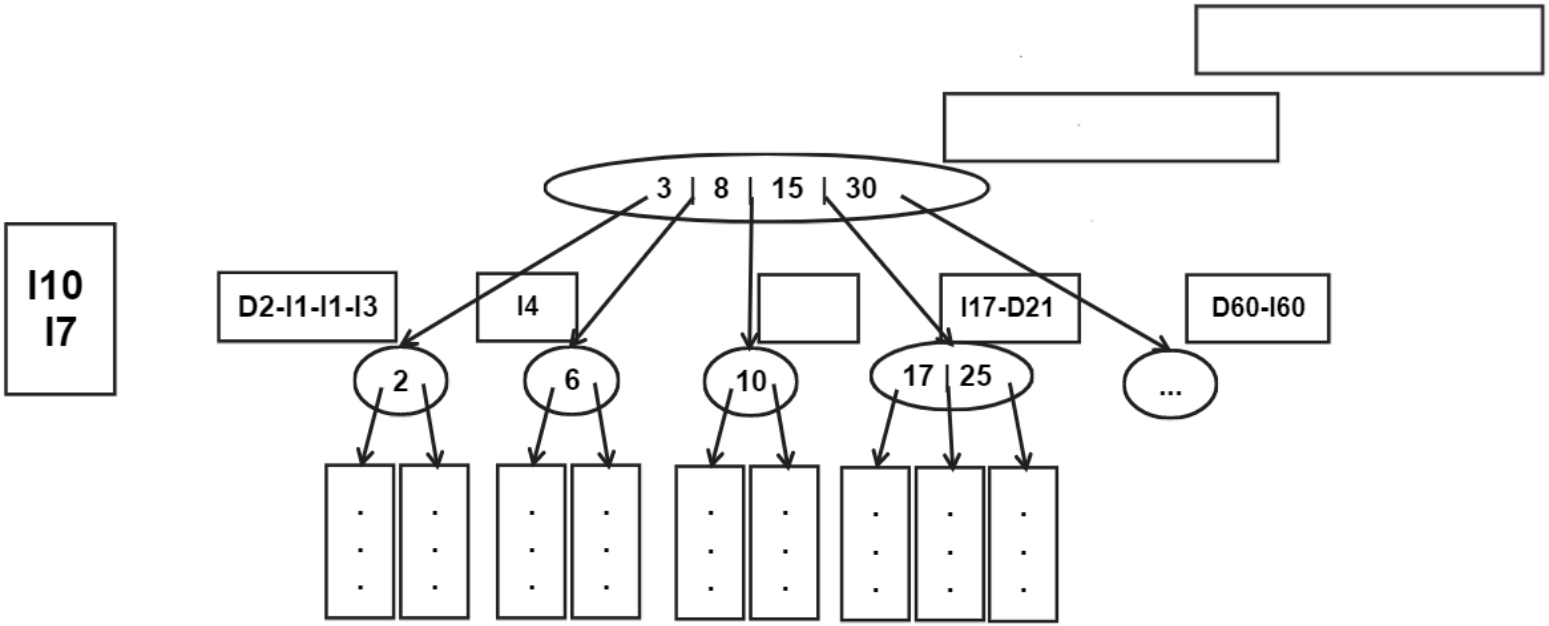

Figure 4: Buffer Distribution Example

Buffer Distribution Example

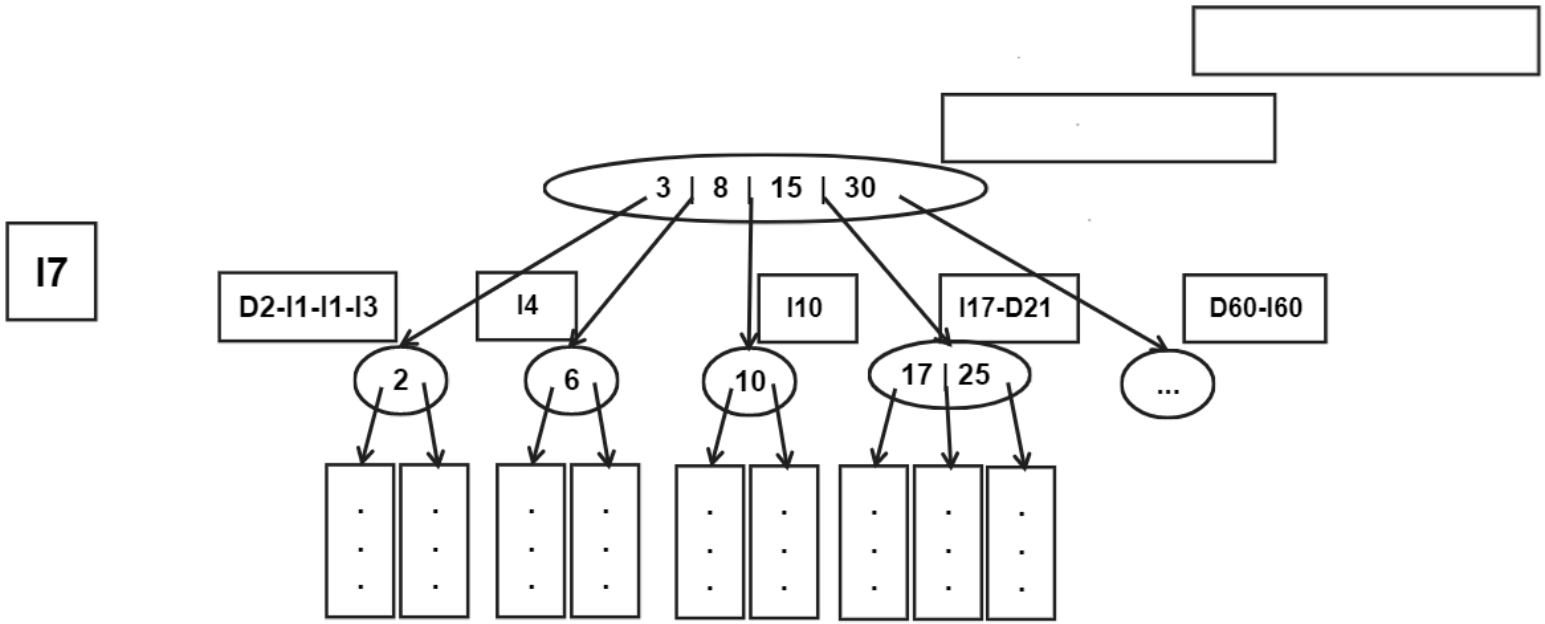

Figure 4: Buffer Distribution Example

Buffer Distribution Example

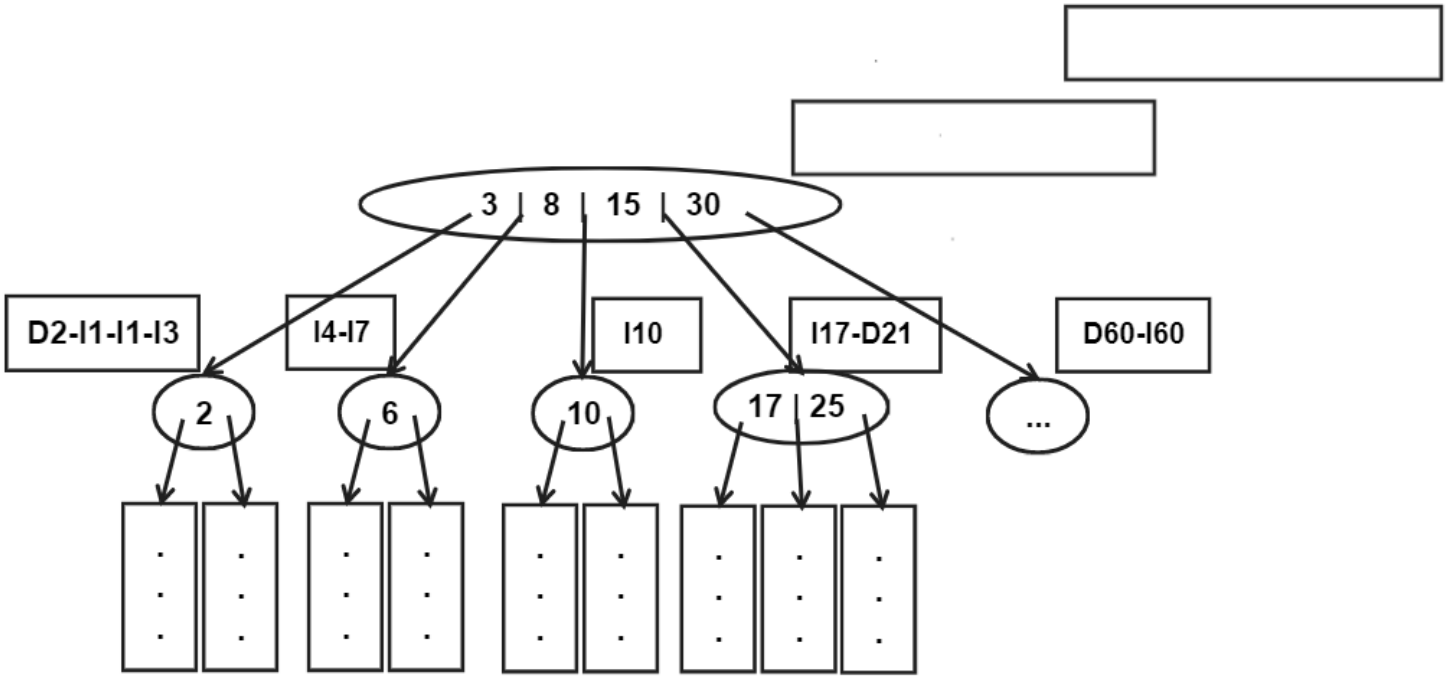

Figure 4: Buffer Distribution Example

Outline

Summary

- Efficient batched external data structure
- Great for parallel computing
- Can deal with massive data
- Amortization is key

Outline

References I

David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick.
A case for intelligent ram.
IEEE micro, 17(2):34–44, 1997.

Lars Arge.
The buffer tree: A technique for designing batched external data structures.
Algorithmica, 37:1–24, 2003.

Alok Aggarwal and S Vitter, Jeffrey.
The input/output complexity of sorting and related problems.
Communications of the ACM, 31(9):1116–1127, 1988.