Nguyen-Wong B-Trees

A B-tree is an *n*-way tree with the following characteristics

- It is balanced.
- It is ordered.
- It has a node size restriction.

A balanced *n*-way tree

An empty *n*-way tree is balanced.

A non-empty *n*-way tree is balanced iff

- all its child trees have the same height,
- all its child trees are balanced.

An ordered *n*-way tree

An empty *n*-way tree is ordered.

A non-empty *n*-way tree is ordered iff

- the data elements at root node x are in strict ascending order,
- all the data elements in x.child[i] are less than the value at x.data[i],
- all the data elements in x.child[i+1] are greater than the value at x.data[i],
- all the children are ordered.

A B-tree

A B-tree is a balanced, ordered *n*-way tree with a lower and upper limit on the number of data values in a node.

The order of a Nguyen-Wong B-tree is the maximum number of data values in a node.

Except for the root, the minimum number of data values in a node is the floor of order/2.

The Nguyen-Wong B-tree

- The Nguyen-Wong B-tree uses the composite pattern. Child trees are B-trees.
- It uses the primitive operations of the n-tree.
- It uses the visitor pattern for insertions and deletions to maintain the B-tree properties.
- It uses functional programming to pass a function as a parameter.

Examples

Every node in a Nguyen-Wong B-tree of order 4 (except the root) has 2 to 4 data values.

Every node in a Nguyen-Wong B-tree of order 5 (except the root) has 2 to 5 data values.

Every node in a Nguyen-Wong B-tree of order 6 (except the root) has 3 to 6 data values.

Exercises for the student

`isEmpty()`	`NTreeIsEmptyVis.hpp`
`contains()`	`NWBTreeContainsVis.hpp`
`height()`	`NWBTreeHeightVis.hpp`
`maxVis()`	`NWBTreeMaxVis.hpp`
`minVis()`	`NWBTreeMinVis.hpp`
`numNodes()`	`NWBTreeNumNodesVis.hpp`
`numValues()`	`NWBTreeNumValuesVis.hpp`

Demo Nguyen-Wong B-tree contains unit test

```
// ========== NWBTreeContainsVis ==========
template<class T>
class NWBTreeContainsVis : public ANTreeVis<T> {
private:
    T _val; // Input parameter.
    int _result; // Output result.

public:
    // ========== Constructor ==========
    NWBTreeContainsVis(T val) :
        _val(val) {
    }
```

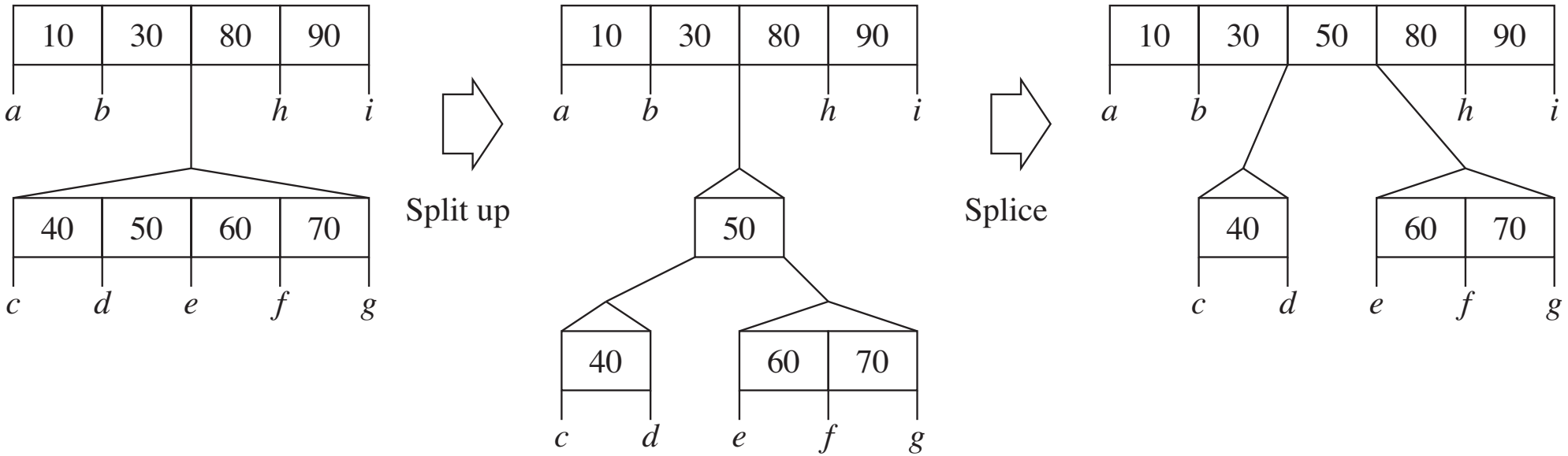
```
// ========= visit =========
void caseAt(int size, NTree<T> &host) override {
    switch (size) {
    case 0:
    {
        cerr << "NWBTreeContainsVis: Exercise for the student."
             << endl;
        throw -1;
    }
    default:
    {
        int k = 0;
        while (k < size && host.getData(k) < _val) {
            k++;
        }
        cerr << "NWBTreeContainsVis: Exercise for the student."
             << endl;
        throw -1;
    }
    }
}
```

```
    // ========= result =========
    // Pre: This visitor has been accepted by a host tree.
    // Post: The position of val in its node is returned if val
    // is contained in this tree; otherwise, -1 is returned.

    int result() const {
        cerr << "BiTCSVcontainsVis: Exercise for the student."
             << endl;
        throw -1;
    }
};

// Global function for convenience
template<class T>
int contains(T key, NTree<T> const &tree) {
    cerr << "contains: Exercise for the student."
         << endl;
    throw -1;
}
```

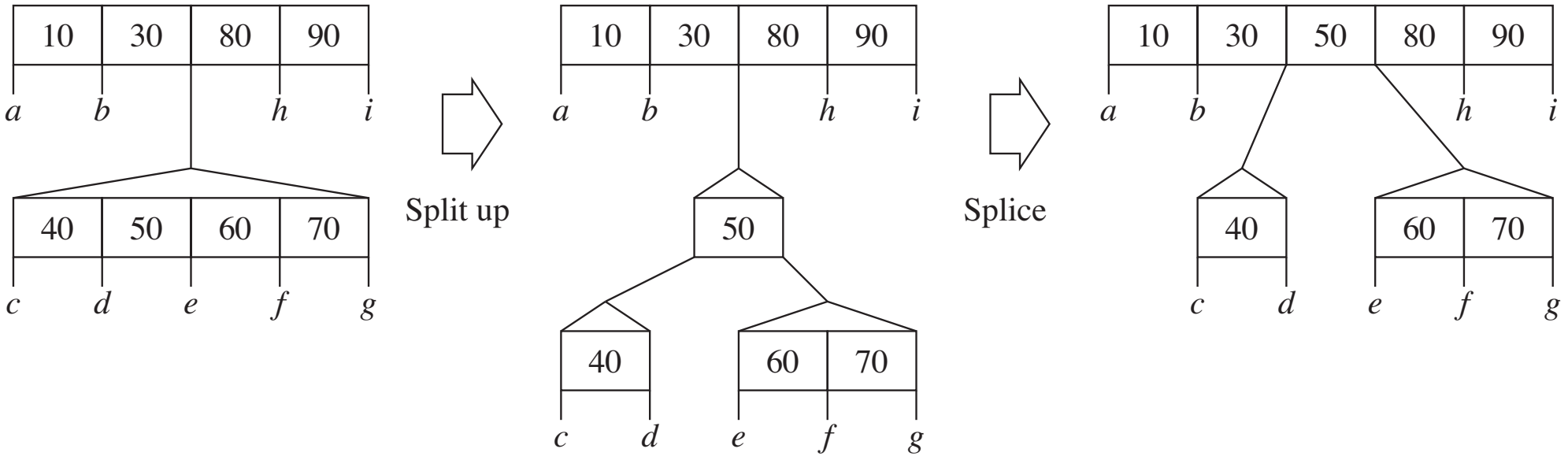
Insertion and deletion

Based on moving data vertically in a tree while preserving

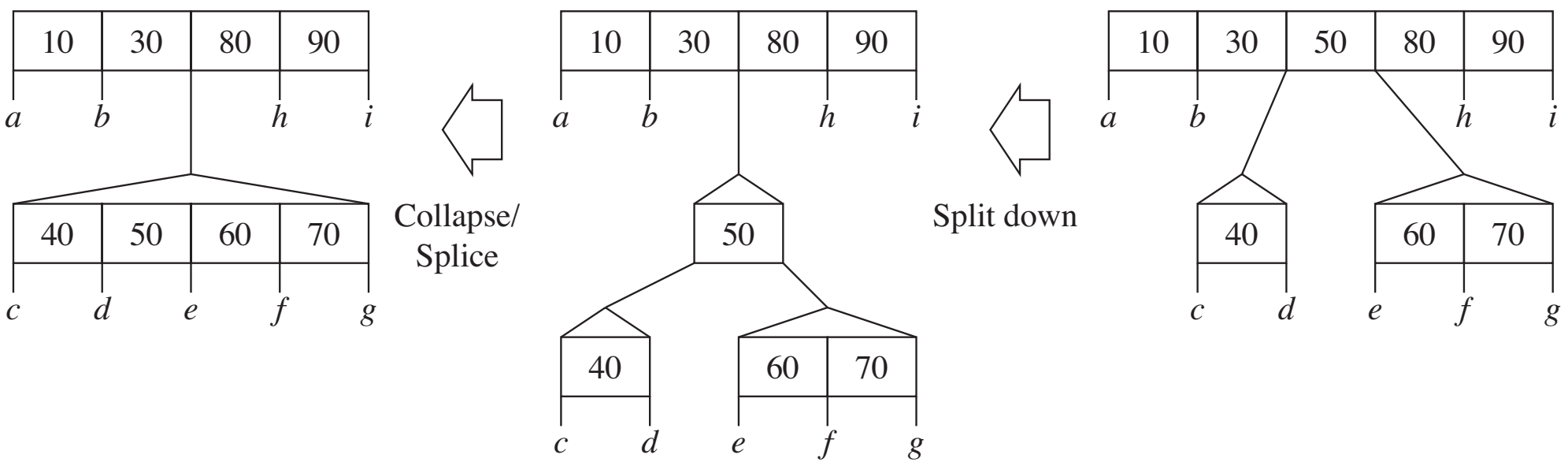
- the height balance,
- the order,
- the node size restriction.



(a) Lifting a value up the tree.



(a) Lifting a value up the tree.



(b) Pushing a value down the tree.

Inserting into a Nguyen-Wong B-tree

- Start at the root.
- Push down to a leaf and insert.
- SplitUpAndApply back to the root.

The `splitUpAndApply()` utility function

- Only splits if the tree is too wide.
- Applies `cmd`, which is a no-op or a splice.

```
splitUpAndApply(order, cmd)
    if (size > order)
        splitUpAt(size / 2)
        Execute cmd(host)
```

Inserting into a Nguyen-Wong B-tree

```
insert(key, order)
    if (tree is empty) // Inserting into empty root.
        Create new single-node temp tree temp
        spliceAt(0, temp)
        return
    else
        Call inHelper(key, order, no-op lambda command)
```

Inserting into a Nguyen-Wong B-tree

```
inHelper(key, order, cmd)
    if (tree is empty) // Inserting at a leaf.
        Create new single-node temp tree temp with key key
        Execute cmd(temp) // Splice temp into parent tree.
        return
    else
        Determine k, the index for which _data[k] == key or, if key
            is not in this tree, the index of the child tree in which key
            should be inserted.
        if (k < size && _data(k) == _key)
            return; // Duplicate keys not allowed.
        oldCmd = cmd
        cmd = splice-at-k lambda command
        Call _child[k].inHelper(key, order, cmd)
        cmd = oldCmd
        Call splitUpAndApply(order, cmd)
```

Inserting into an order-4 Nguyen-Wong B-tree

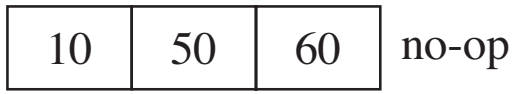
The tree has one non-full node at the root.

(a) Initial tree.
Insert 30.

10	50	60

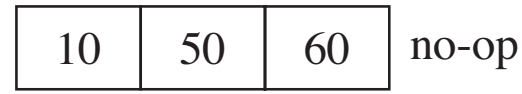
(a) Initial tree.
Insert 30.

(b) Call helper with no-op command.

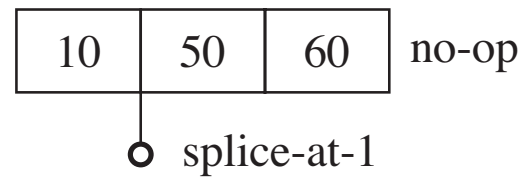


(a) Initial tree.
Insert 30.

(b) Call helper with no-op command.

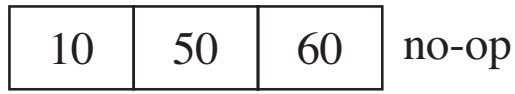


(c) `k` = 1.
Call helper with splice-at-1 command.

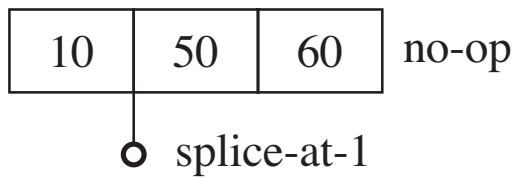


(a) Initial tree.
Insert 30.

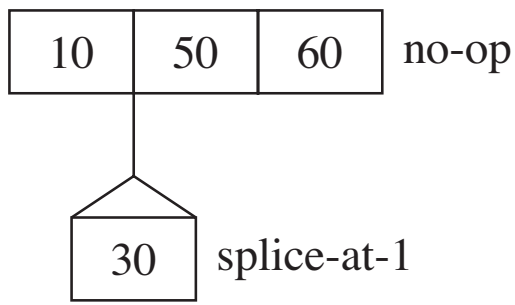
(b) Call helper with no-op command.



(c) `k = 1`.
Call helper with splice-at-1 command.

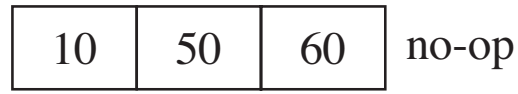


(d) Create single-value 30 temp tree.
Splice into parent tree.

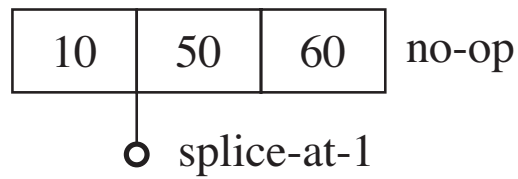


(a) Initial tree.
Insert 30.

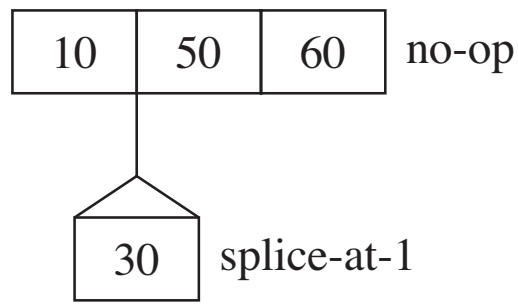
(b) Call helper with no-op command.



(c) `k` = 1.
Call helper with splice-at-1 command.



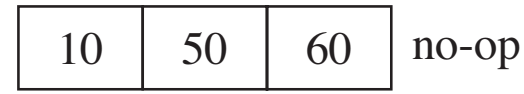
(d) Create single-value 30 temp tree.
Splice into parent tree.



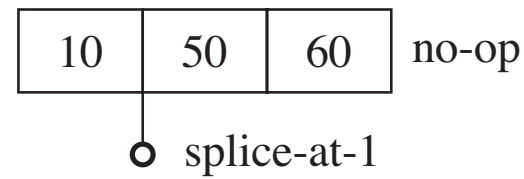
(e) Call `splitUpAndApply`
with splice-at-1 command.

(a) Initial tree.
Insert 30.

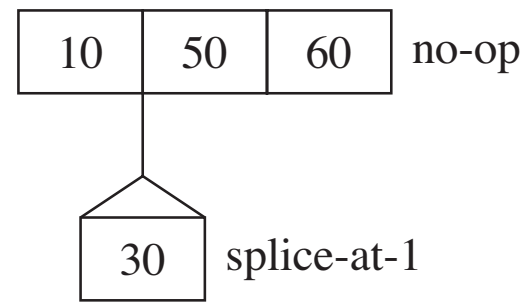
(b) Call helper with no-op command.



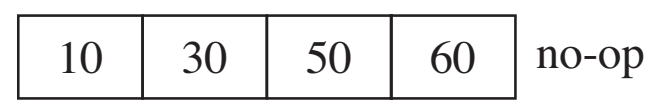
(c) `k` = 1.
Call helper with splice-at-1 command.



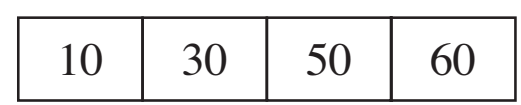
(d) Create single-value 30 temp tree.
Splice into parent tree.



(e) Call `splitUpAndApply`
with splice-at-1 command.



(f) Call `splitUpAndApply`
with no-op command.



Inserting into an order-4 Nguyen-Wong B-tree

The tree has one full node at the root.

(a) Initial tree.
Insert 70.

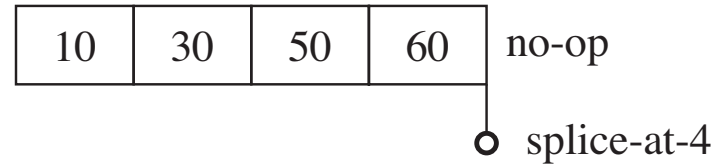
(a) Initial tree.
Insert 70.

(b) Call helper with no-op command.

(a) Initial tree.
Insert 70.

(b) Call helper with no-op command.

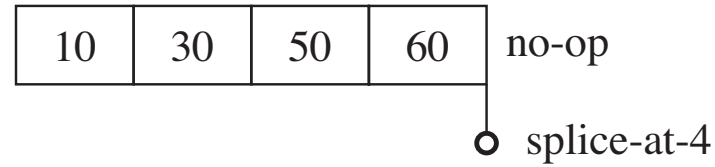
(c) `k` = 4.
Call helper with splice-at-4 command.



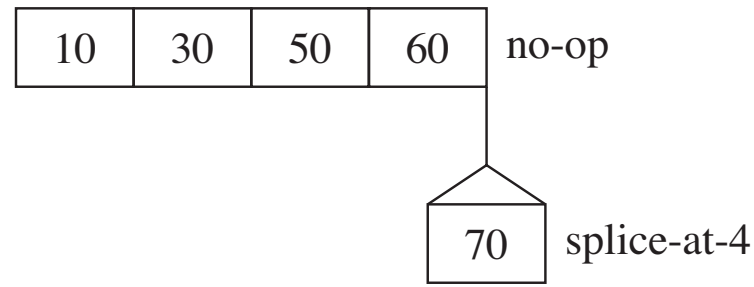
(a) Initial tree.
Insert 70.

(b) Call helper with no-op command.

(c) `k` = 4.
Call helper with splice-at-4 command.



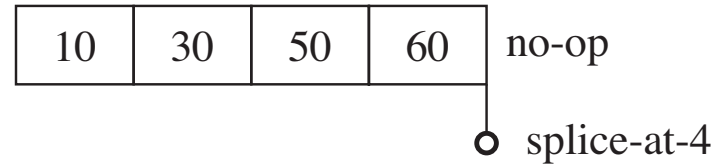
(d) Create single-value 70 temp tree.
Splice into parent tree.



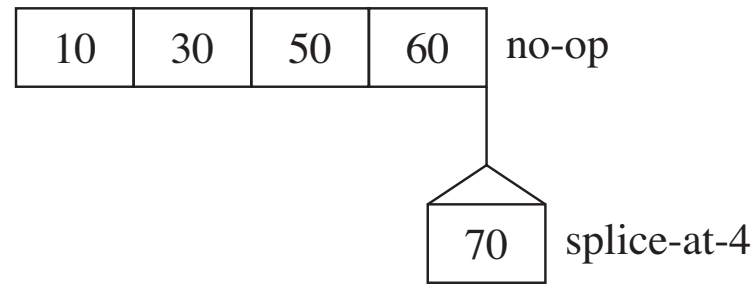
(a) Initial tree.
Insert 70.

(b) Call helper with no-op command.

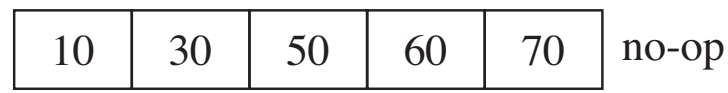
(c) `k` = 4.
Call helper with splice-at-4 command.



(d) Create single-value 70 temp tree.
Splice into parent tree.



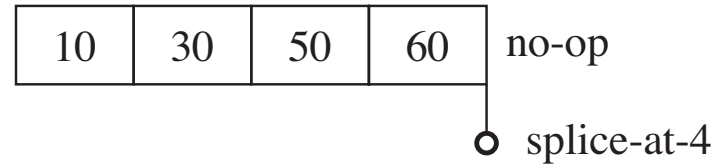
(e) Call `splitUpAndApply`
with splice-at-4 command.



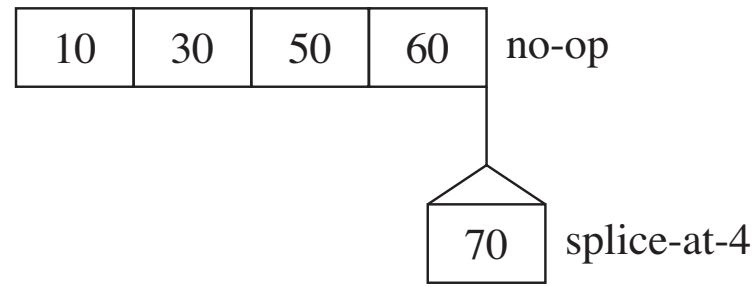
(a) Initial tree.
Insert 70.

(b) Call helper with no-op command.

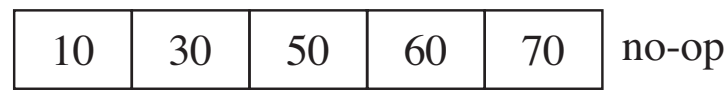
(c) `k` = 4.
Call helper with splice-at-4 command.



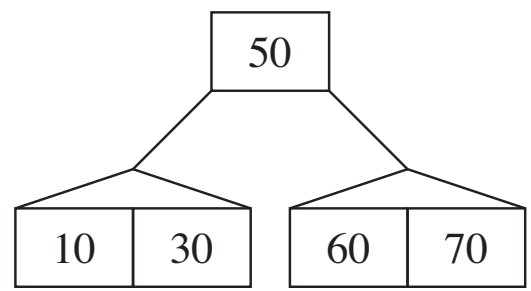
(d) Create single-value 70 temp tree.
Splice into parent tree.



(e) Call `splitUpAndApply`
with splice-at-4 command.

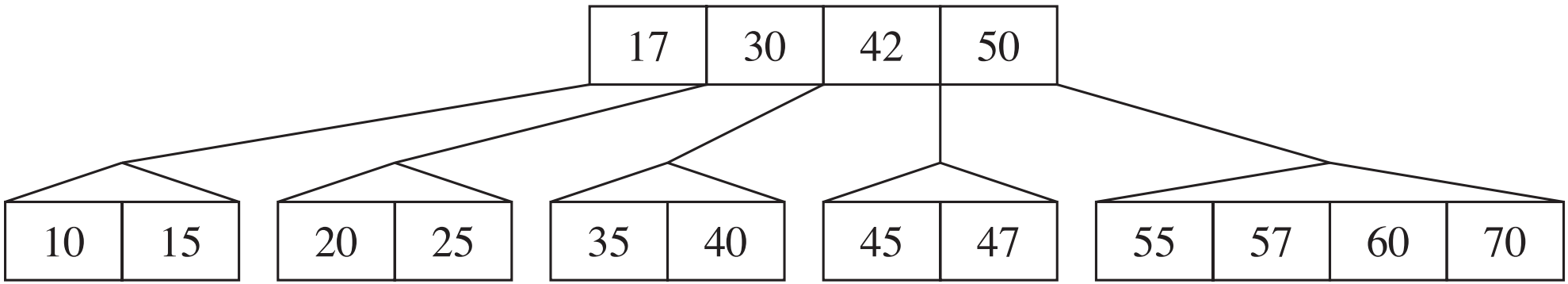


(f) Call `splitUpAndApply`
with no-op command.

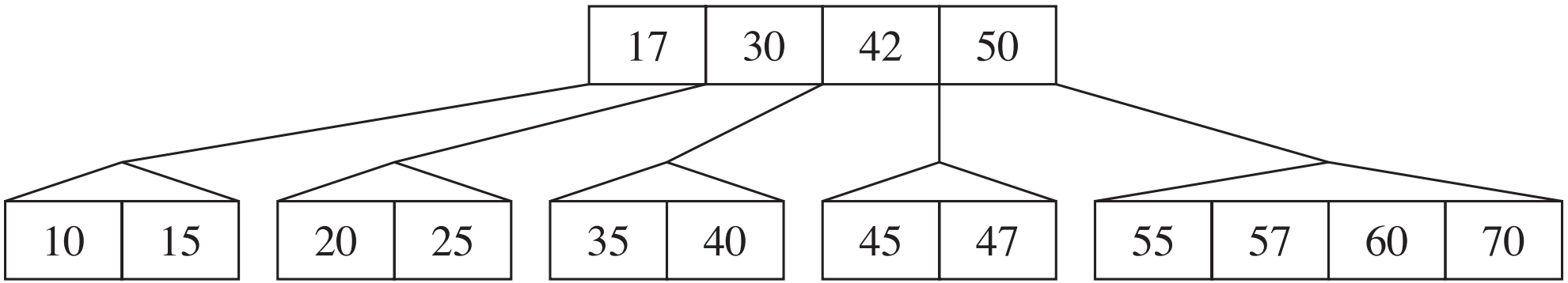


Inserting into an order-4 Nguyen-Wong B-tree

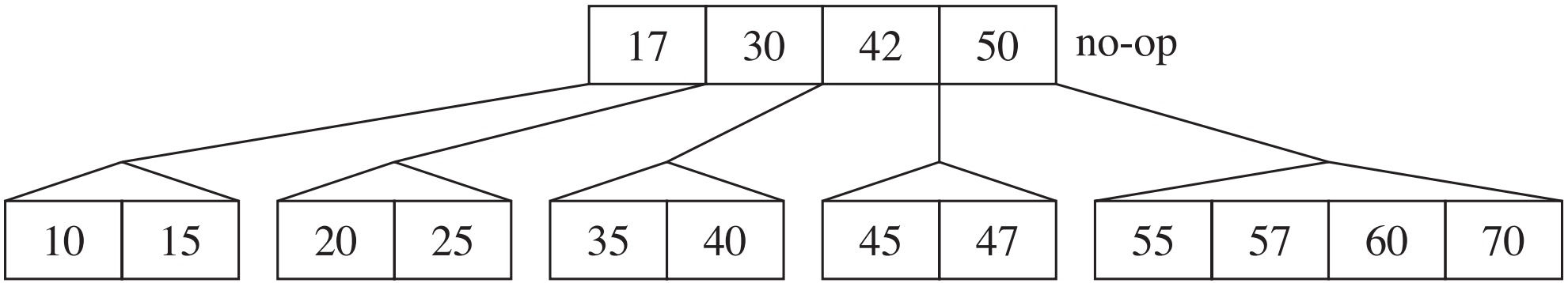
The tree has children and a full root.



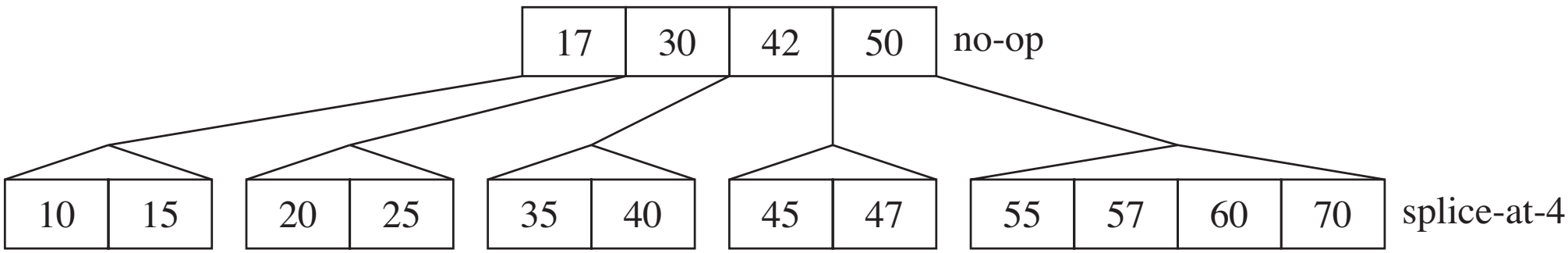
(a) Initial tree. Insert 65.



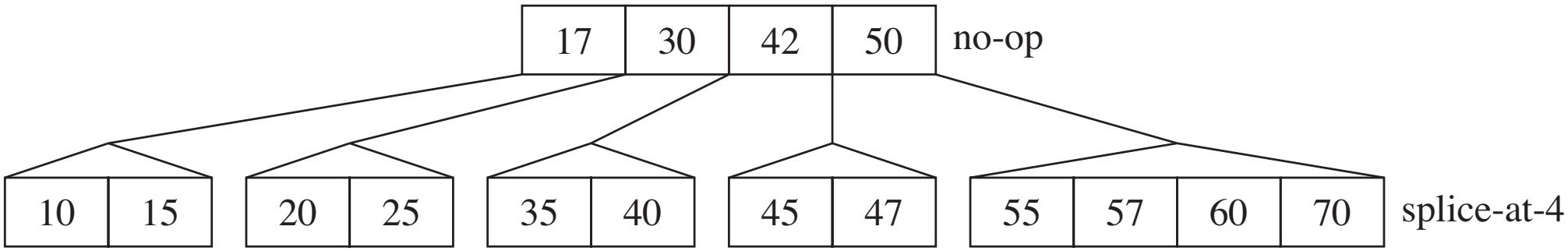
(a) Initial tree. Insert 65.



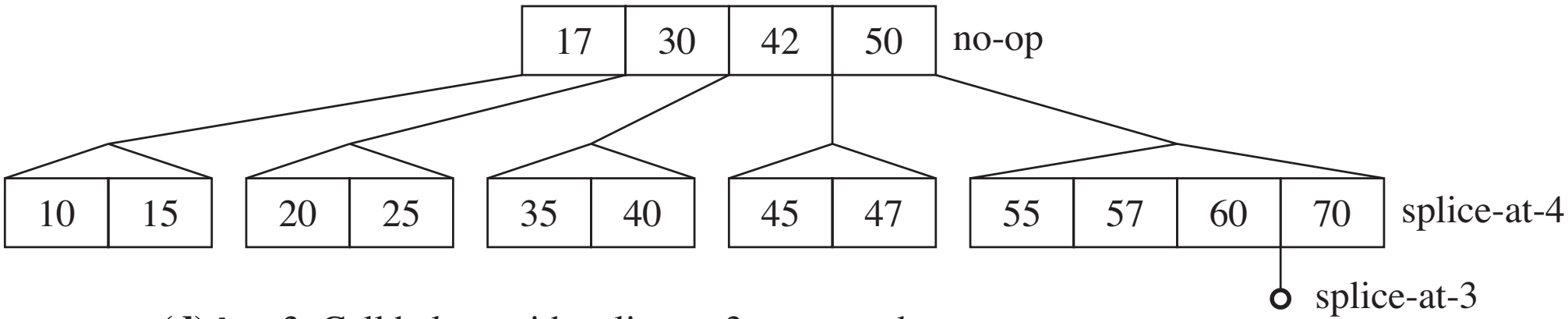
(b) Call helper with no-op command.



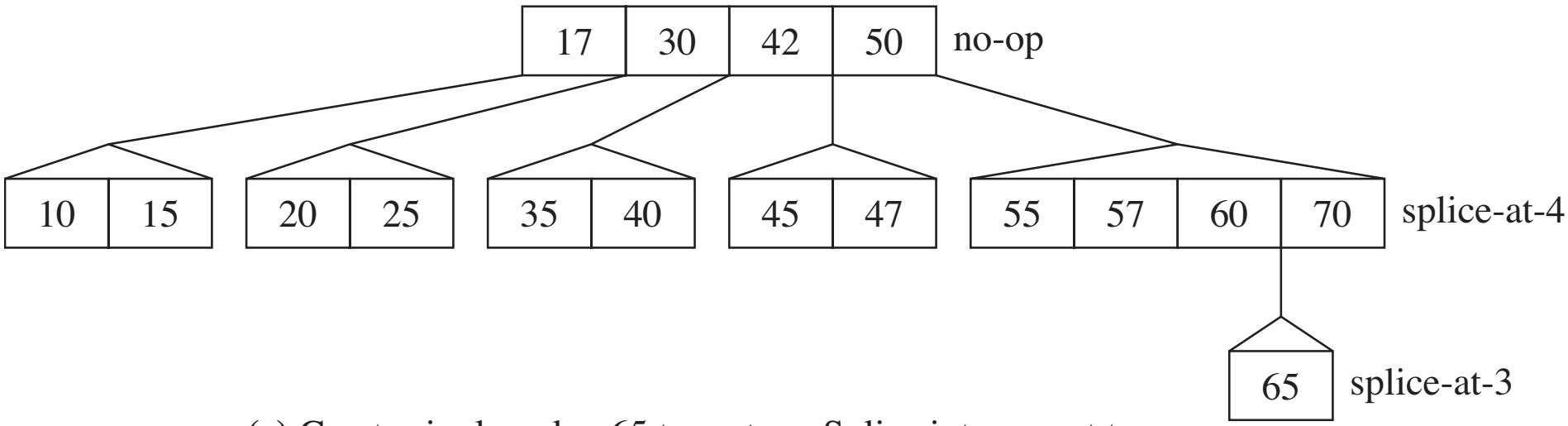
(c) `k` = 4. Call helper with splice-at-4 command.



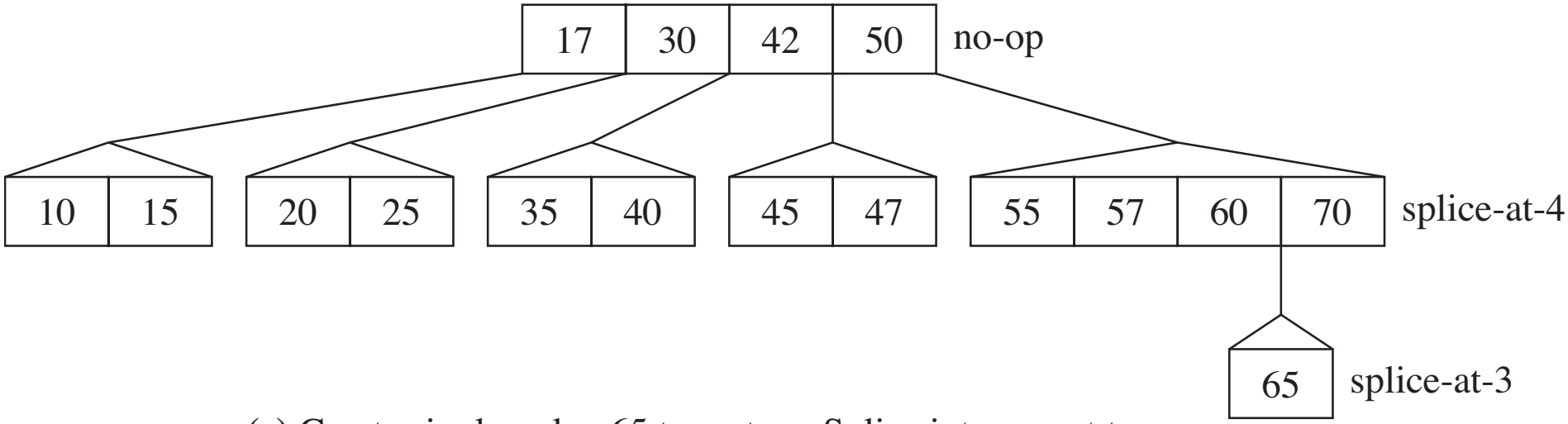
(c) `k` = 4. Call helper with splice-at-4 command.



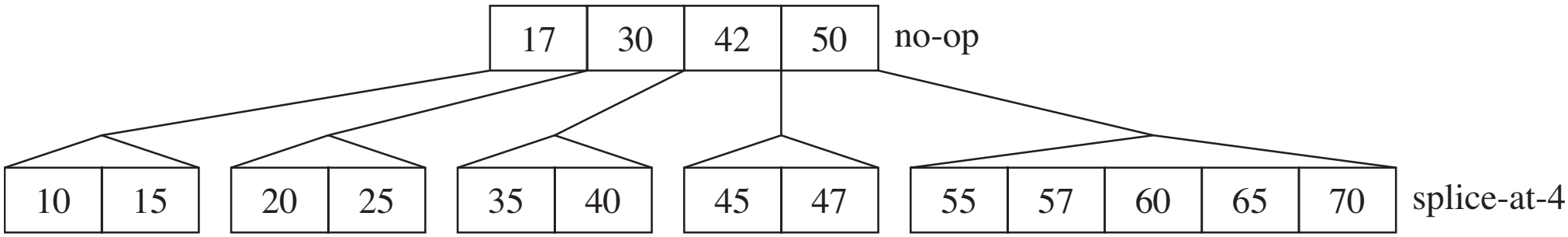
(d) `k` = 3. Call helper with splice-at-3 command.



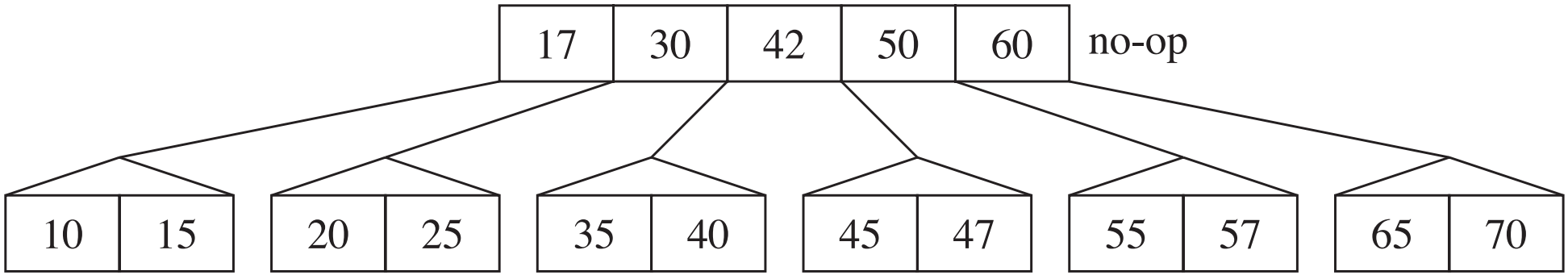
(e) Create single-value 65 temp tree. Splice into parent tree.



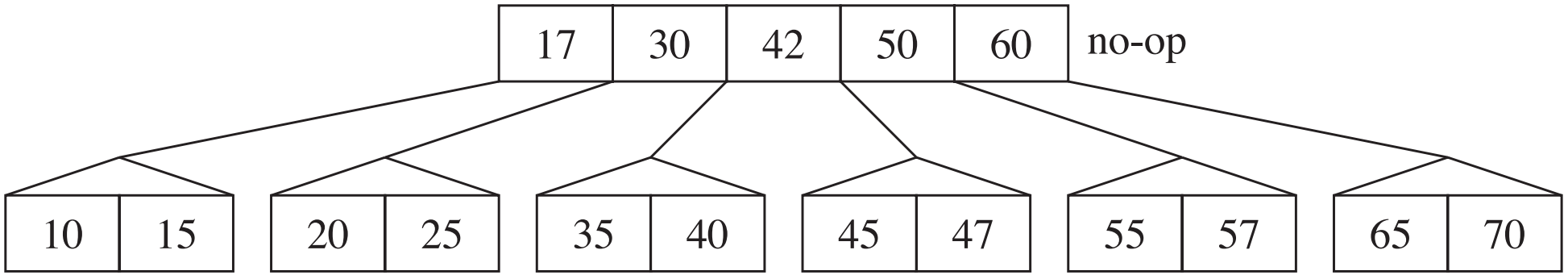
(e) Create single-value 65 temp tree. Splice into parent tree.



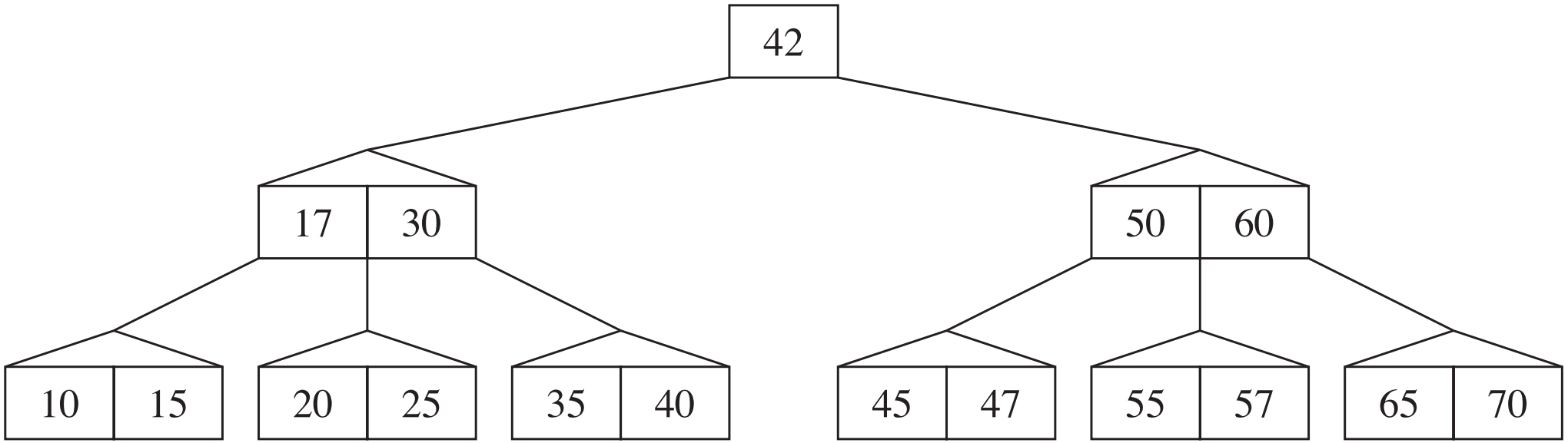
(f) Call `splitUpAndApply` with splice-at-3 command.



(g) Call `splitUpAndApply` with splice-at-4 command.



(g) Call `splitUpAndApply` with splice-at-4 command.



(h) Call `splitUpAndApply` with no-op command.

Removing from a Nguyen-Wong B-tree

```
remove(key, order)
    if (tree is empty)
        return // There is nothing to remove
    else
        if (tree has one element)
            Collapse tree with children
        Call remHelper(key, order, no-op lambda command)
```

Removing from a Nguyen-Wong B-tree

```
remHelper(key, order, cmd)
    if (tree is empty)
        return // There is nothing to remove
    else if (tree has one element)
        if (_data[0] == key)
            splitDownAt(0) // Remove the key. This tree becomes empty.
        else // key is not in the tree
            Execute cmd(host) // Splice back to the parent tree.
    else
        Determine k, the index for which _data[k] == key or, if key
            is not in this tree, the index for which key is in the left child of
            _data[k] (except when key > _data[size-1], in which case
            k gets size-1).
        Push _data[k] down the tree with split down then collapse/splice.
        oldCmd = cmd
        cmd = splice-at-k lambda command
        Call _child[k].remHelper(key, order, cmd)
        cmd = oldCmd
        Call splitUpAndApply(order, cmd)
```

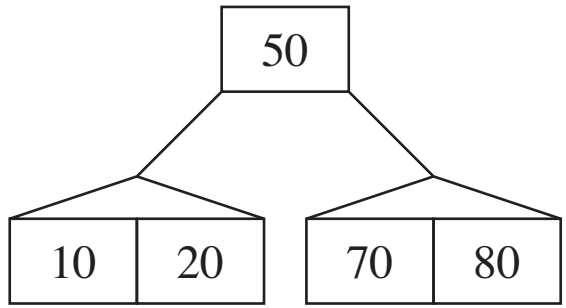
Specification of `splitDownAt(int i)`

```
void splitDownAt(int i);
// Post: If _data->size() == 0 nothing is done.
// Otherwise,
// Assert: 0 <= i < _data->size().
// If _data->size() == 1 the single value
// is deleted and this tree is empty.
// Otherwise, the element at position i
// is split down.
```

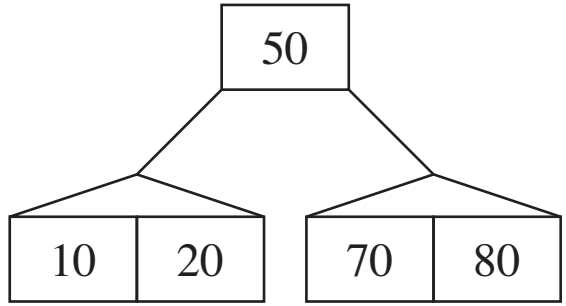
Removing from an order-4 Nguyen-Wong B-tree

The root has only one element, so first collapse the root.

(a) Initial tree.
Remove 20.



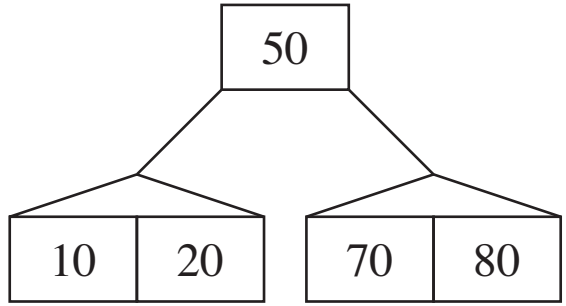
(a) Initial tree.
Remove 20.



(b) Collapse tree with children.
Call helper with no-op command.

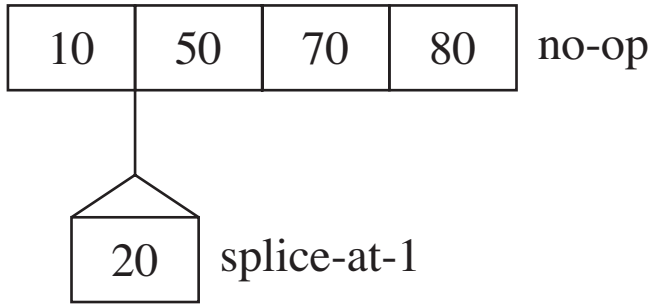
no-op

(a) Initial tree.
Remove 20.

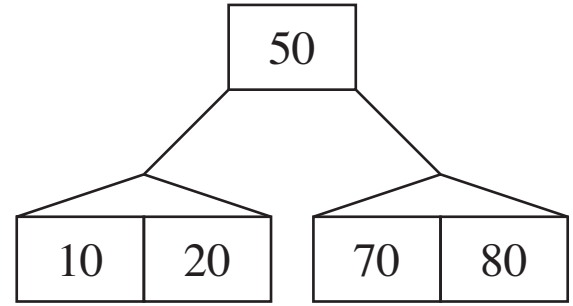


(b) Collapse tree with children.
Call helper with no-op command.

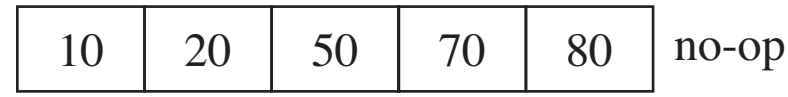
(c) `k` = 1.
Push 20 down.
Call helper with splice-at-1 command.



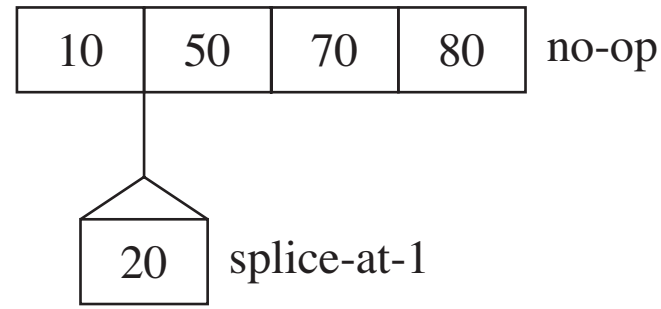
(a) Initial tree.
Remove 20.



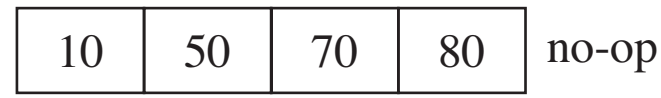
(b) Collapse tree with children.
Call helper with no-op command.



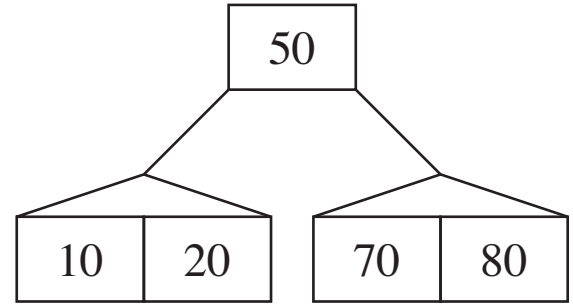
(c) `k` = 1.
Push 20 down.
Call helper with splice-at-1 command.



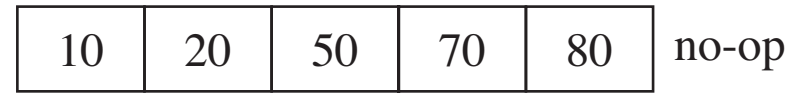
(d) Split down at 0.
This tree becomes empty.
Do not execute splice-at-1 command.



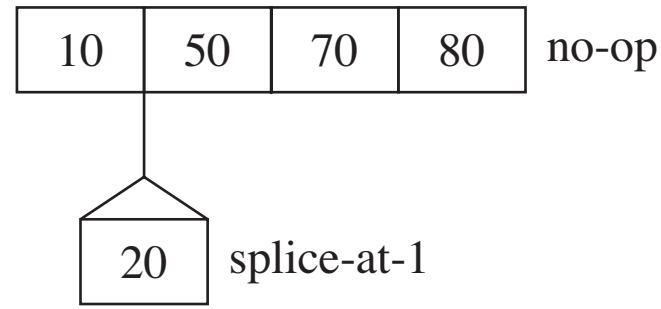
(a) Initial tree.
Remove 20.



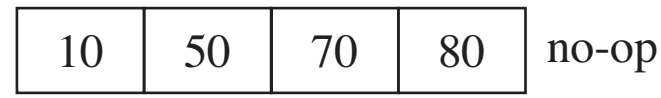
(b) Collapse tree with children.
Call helper with no-op command.



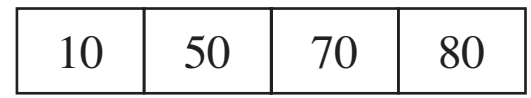
(c) `k` = 1.
Push 20 down.
Call helper with splice-at-1 command.



(d) Split down at 0.
This tree becomes empty.
Do not execute splice-at-1 command.



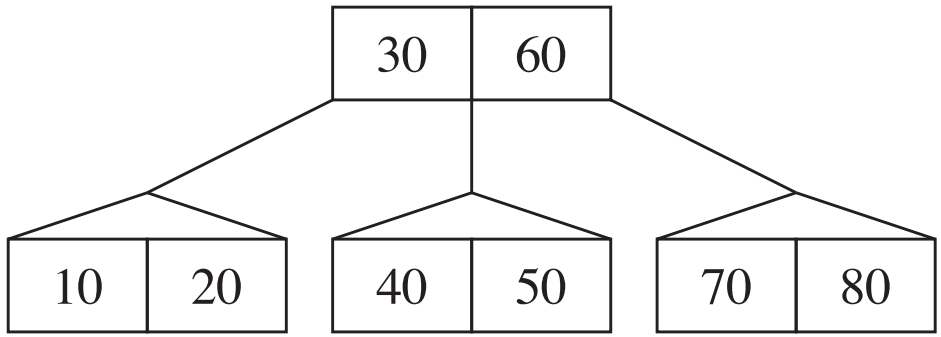
(e) Call `splitUpAndApply`
with no-op command.



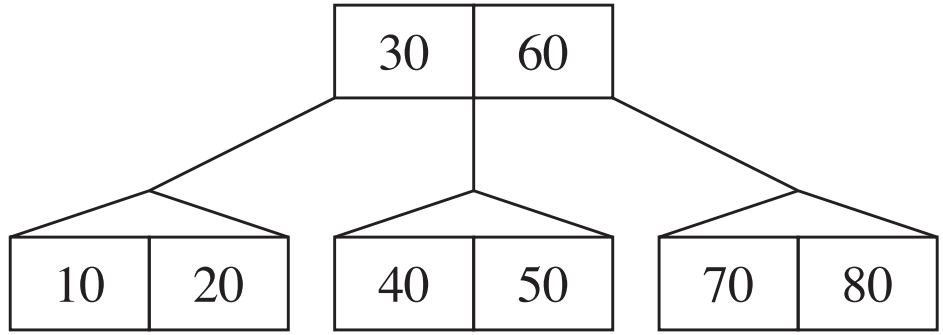
Removing from an order-4 Nguyen-Wong B-tree

Remove an element from a child, so push down an element from the root in anticipation of the deletion.

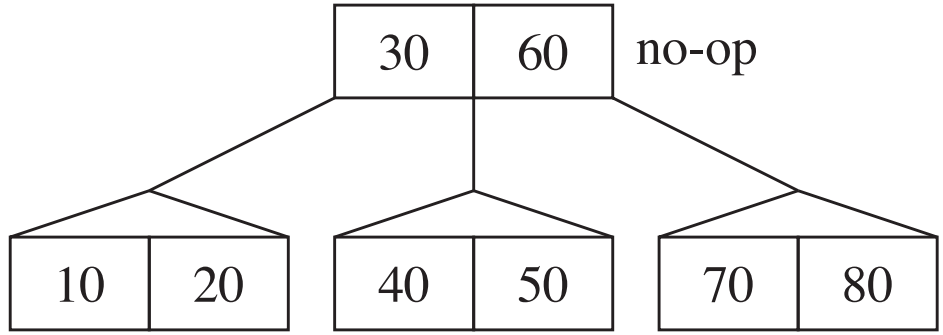
(a) Initial tree.
Remove 20.



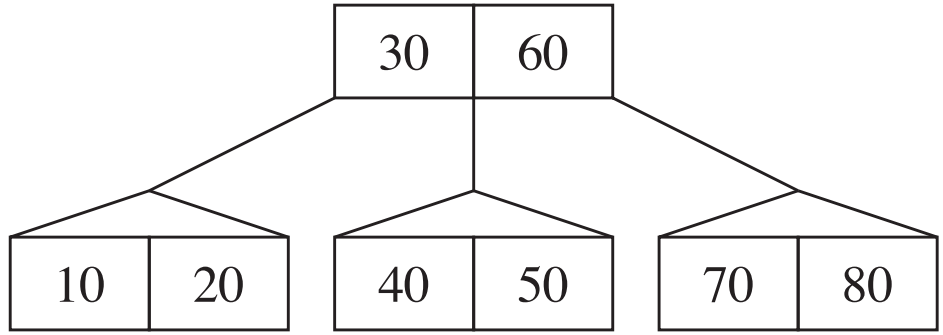
(a) Initial tree.
Remove 20.



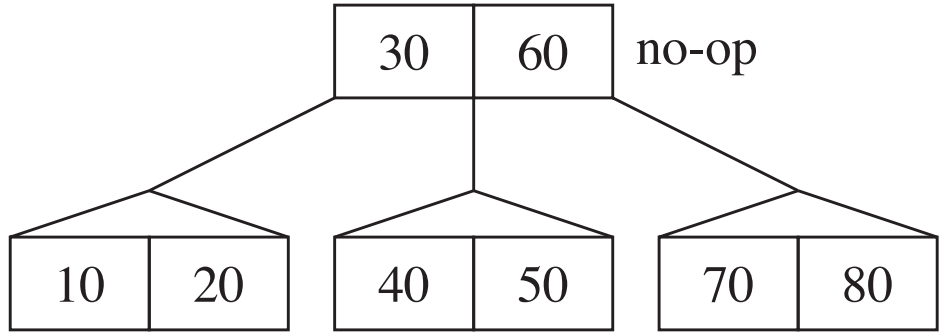
(b) Call helper with no-op command.



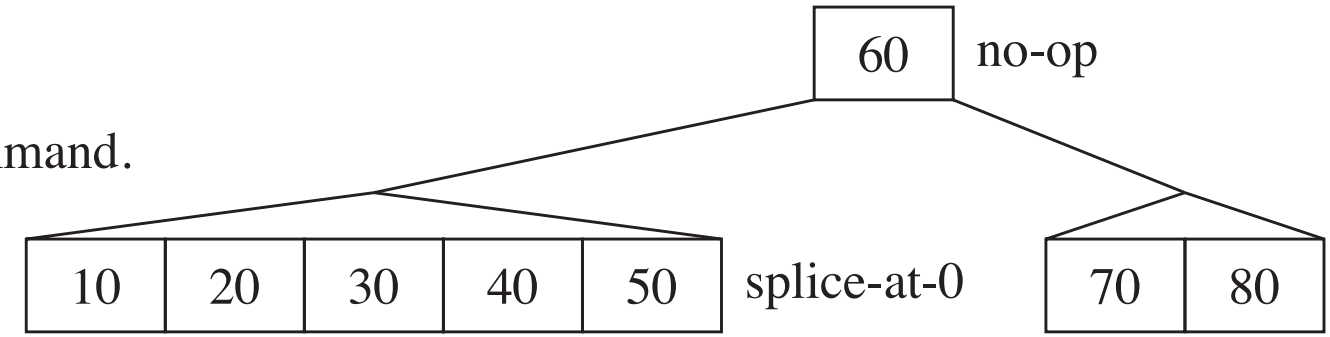
(a) Initial tree.
Remove 20.

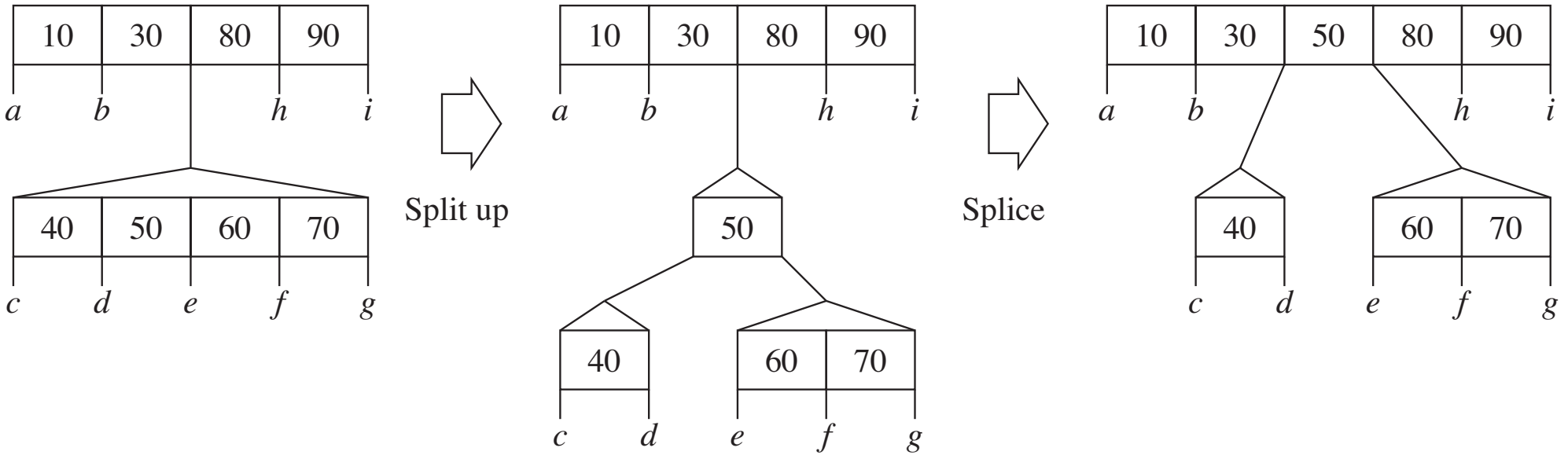


(b) Call helper with no-op command.

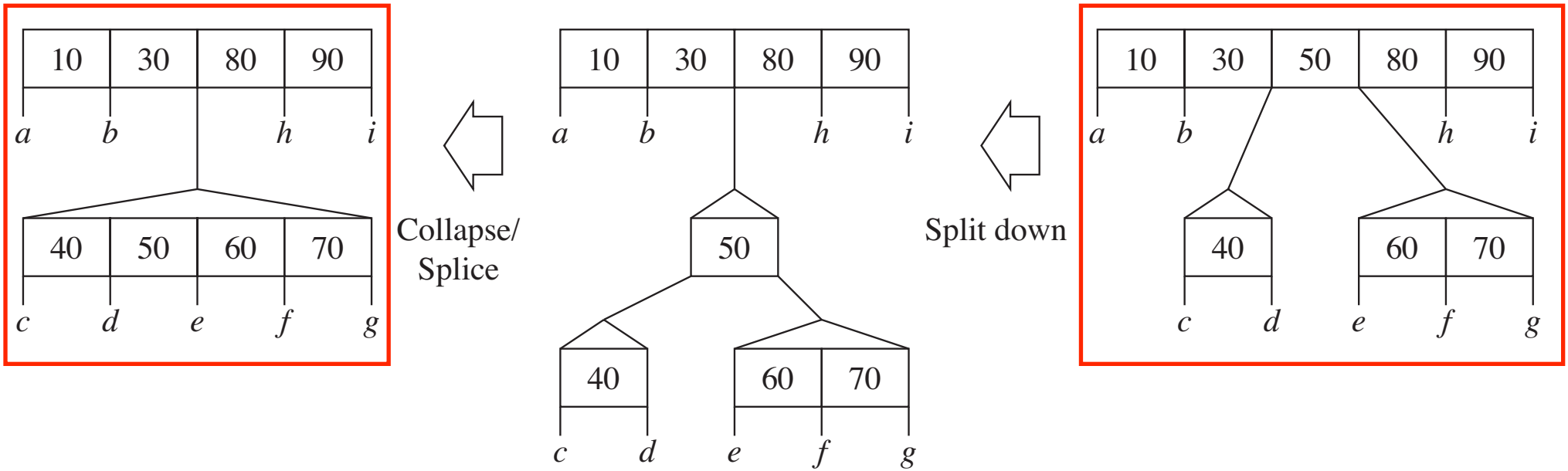


(c) `k` = 0.
Push 30 down.
Call helper with splice-at-0 command.



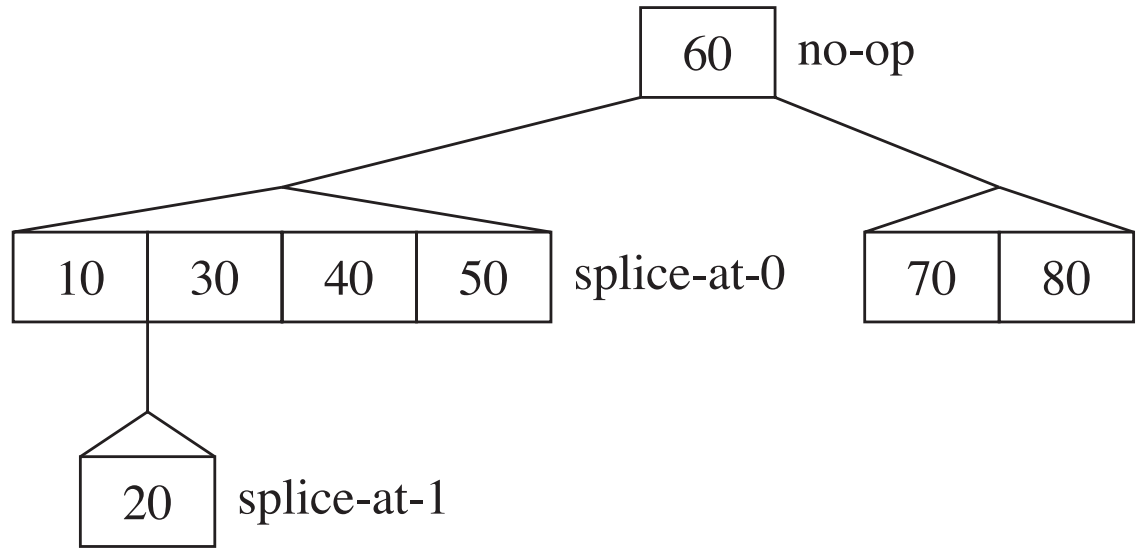


(a) Lifting a value up the tree.

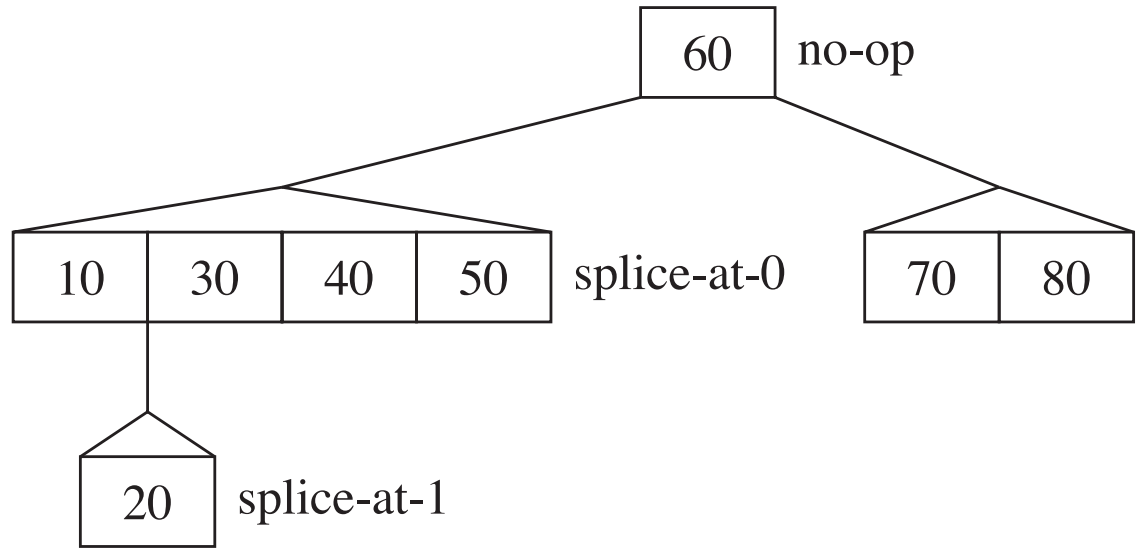


(b) Pushing a value down the tree.

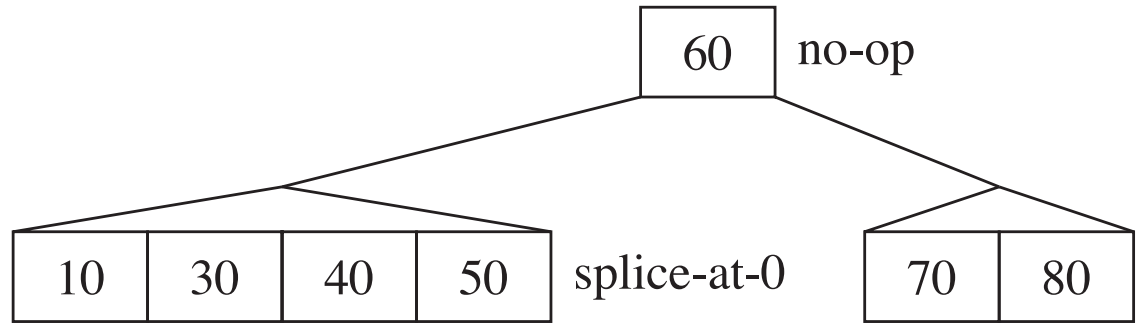
(d) `k` = 1.
Push 20 down.
Call helper with splice-at-1 command.



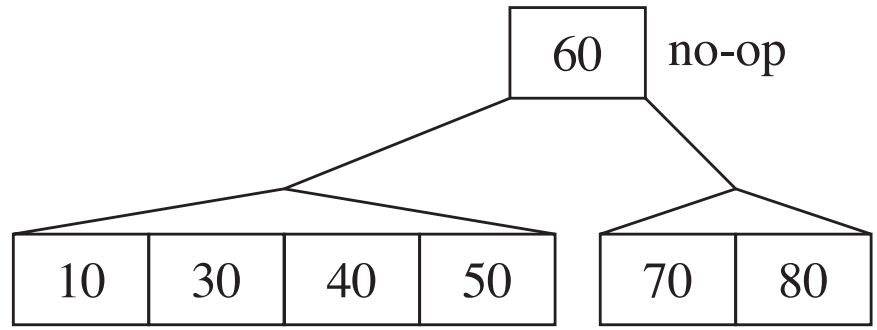
(d) `k` = 1.
Push 20 down.
Call helper with splice-at-1 command.



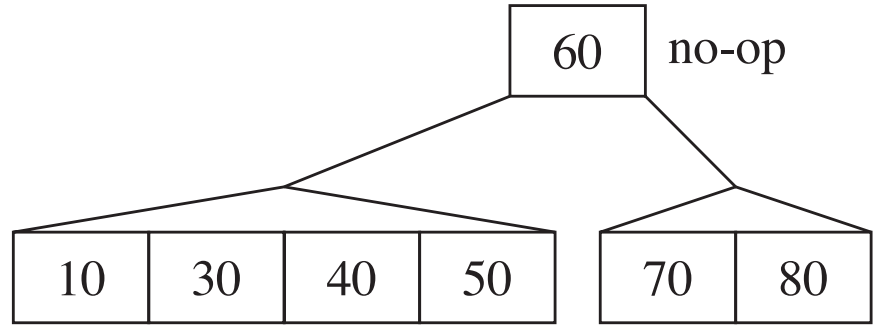
(e) Split down at 0.
This tree becomes empty.
Do not execute splice-at-1 command.



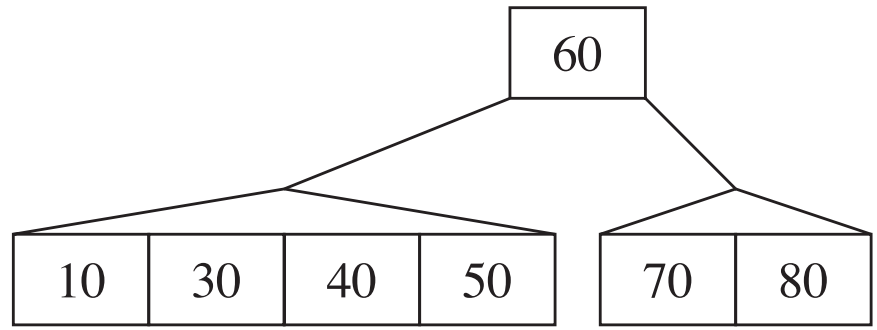
(f) Call `splitUpAndApply` with splice-at-0 command.



(f) Call `splitUpAndApply` with splice-at-0 command.



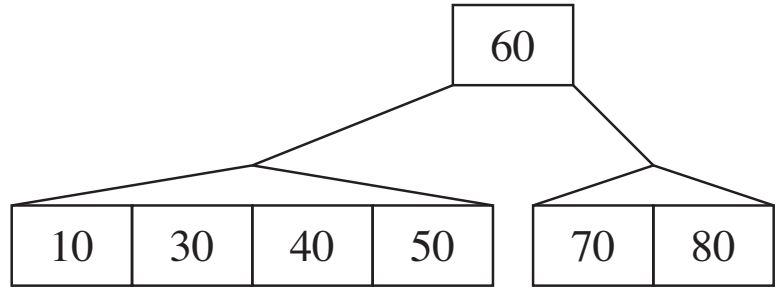
(g) Call `splitUpAndApply` with no-op command.



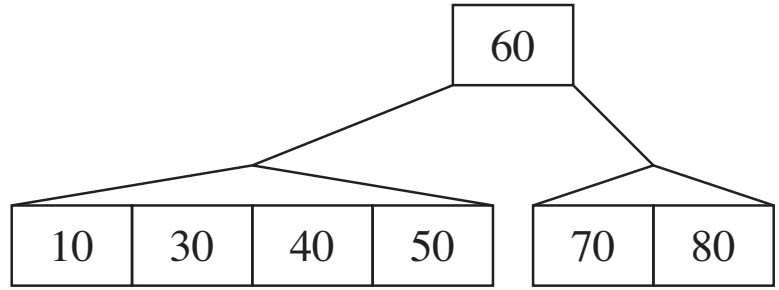
Removing from an order-4 Nguyen-Wong B-tree

Remove an element not in the tree, with a root having only one element. So, first collapse the root. The tree is modified to be "better" balanced in the process.

(a) Initial tree.
Remove 35.

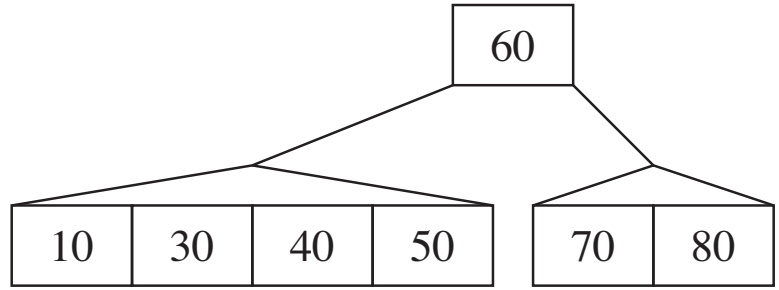


(a) Initial tree.
Remove 35.

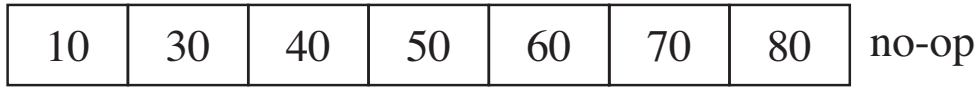


(b) Collapse tree with children.
Call helper with no-op command.

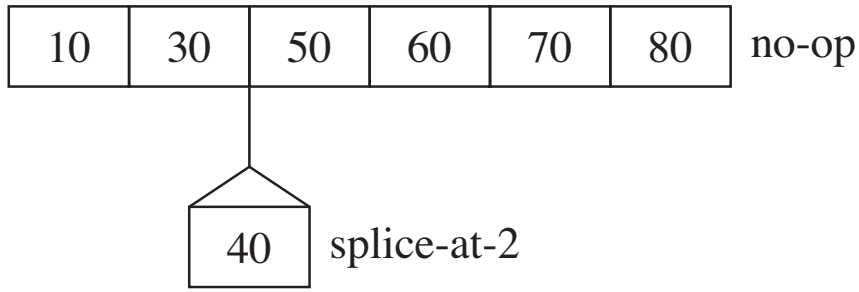
(a) Initial tree.
Remove 35.



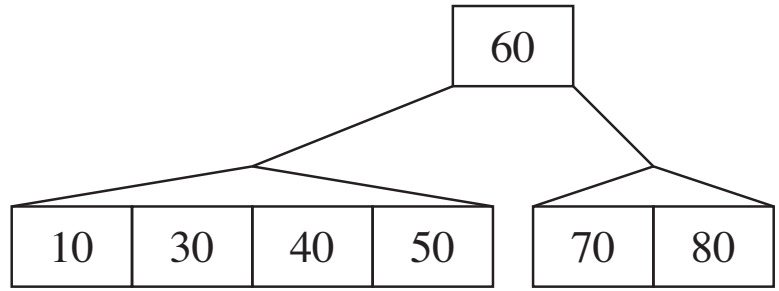
(b) Collapse tree with children.
Call helper with no-op command.



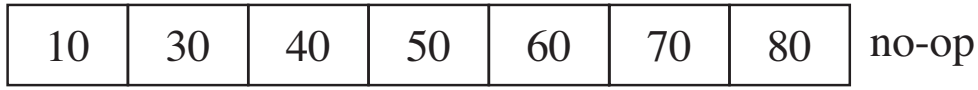
(c) k = 2.
Push 40 down.
Call helper with splice-at-2 command.



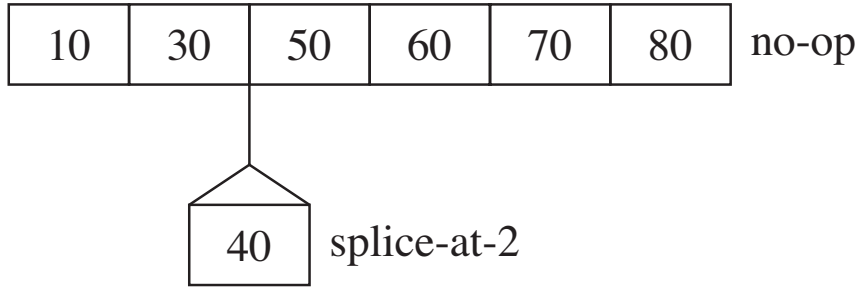
(a) Initial tree.
Remove 35.



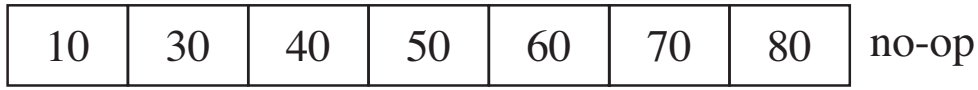
(b) Collapse tree with children.
Call helper with no-op command.



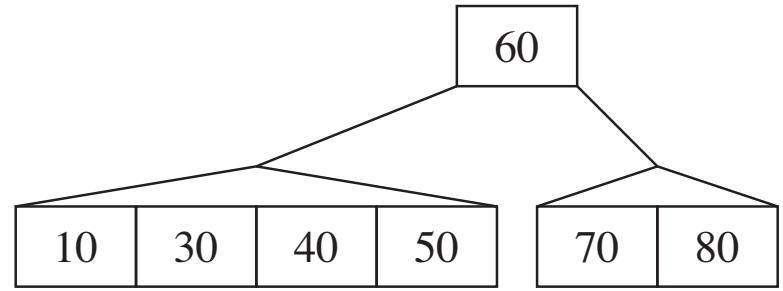
(c) `k` = 2.
Push 40 down.
Call helper with splice-at-2 command.



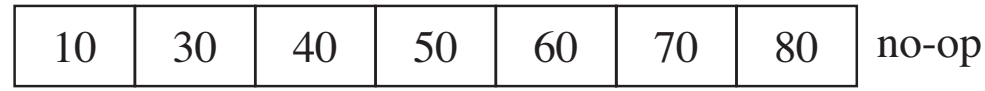
(d) Call `splitUpAndApply` with splice-at-2 command.



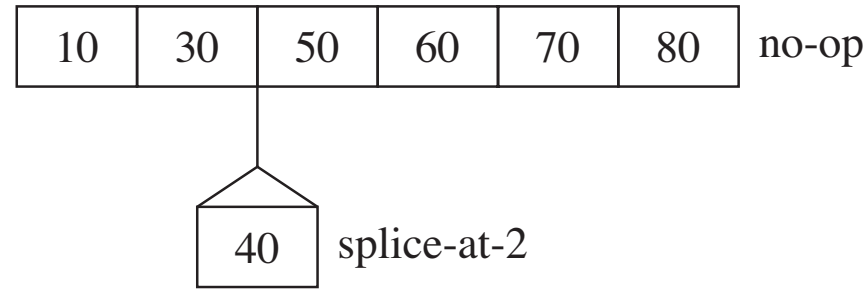
(a) Initial tree.
Remove 35.



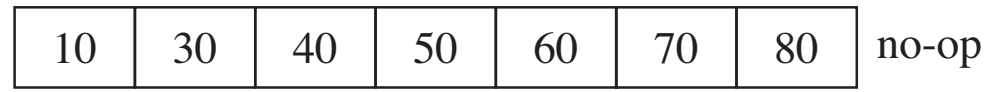
(b) Collapse tree with children.
Call helper with no-op command.



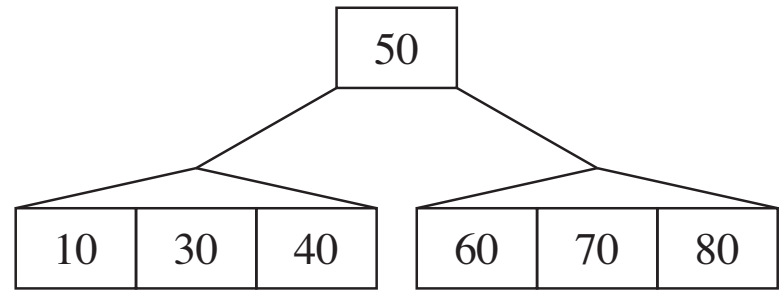
(c) `k` = 2.
Push 40 down.
Call helper with splice-at-2 command.



(d) Call `splitUpAndApply` with splice-at-2 command.



(e) Call `splitUpAndApply` with no-op command.

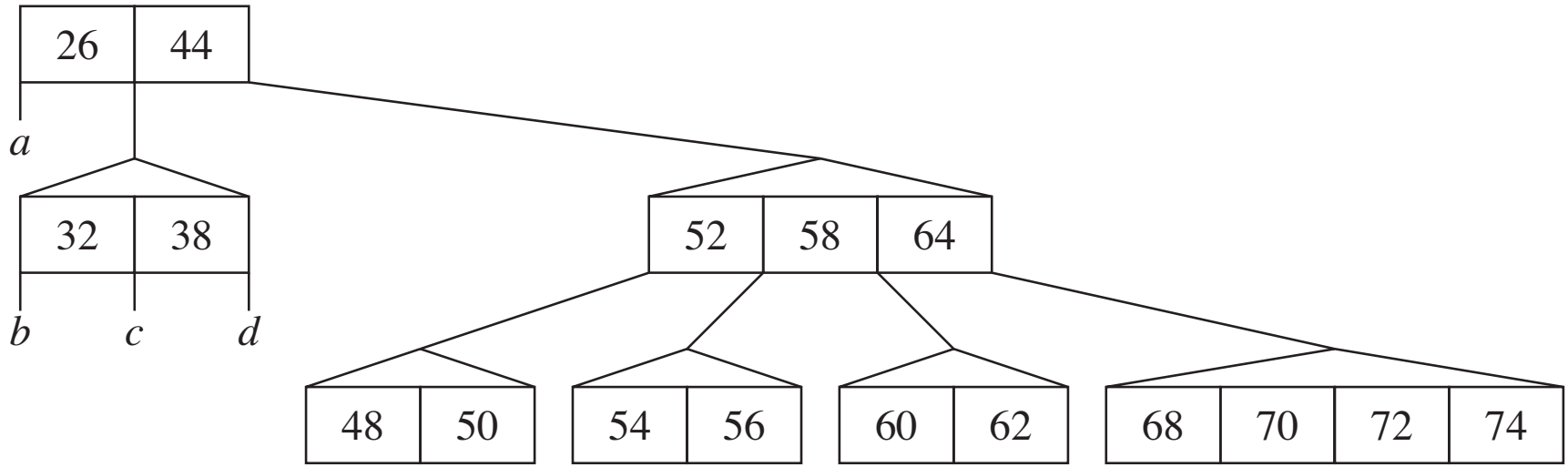


Removing from an order-4 Nguyen-Wong B-tree

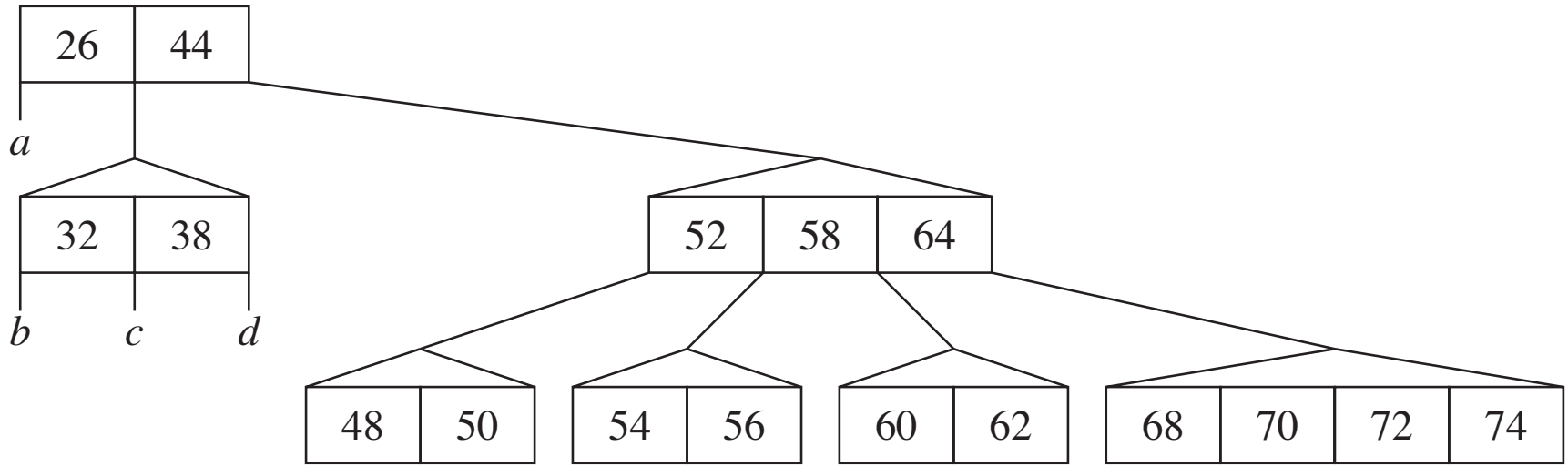
Remove an element not in the root or a leaf.

Initially,

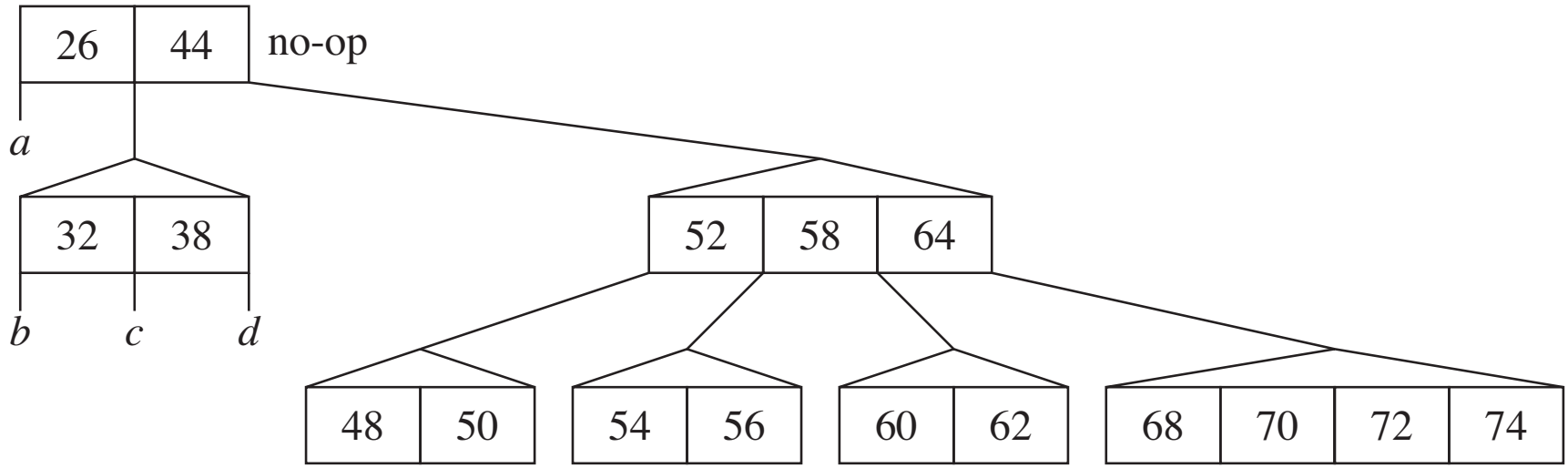
- Height of tree a (not shown) = 2
- Height of tree b (not shown) = 1
- Height of tree c (not shown) = 1
- Height of tree d (not shown) = 1



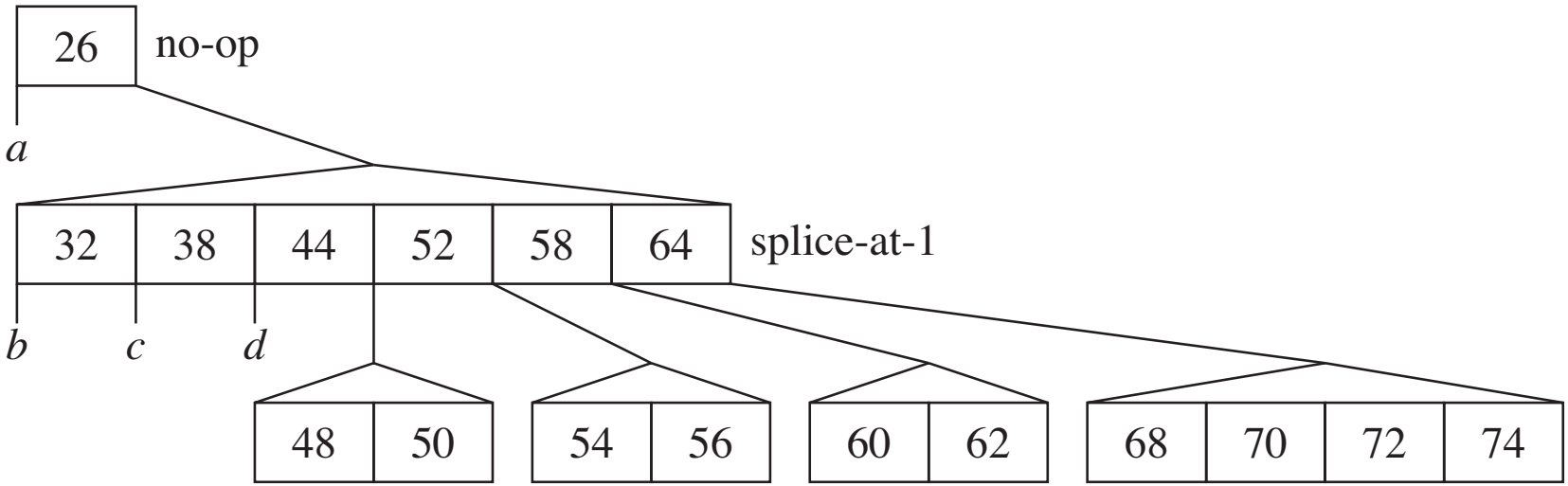
(a) Initial tree. Remove 64.



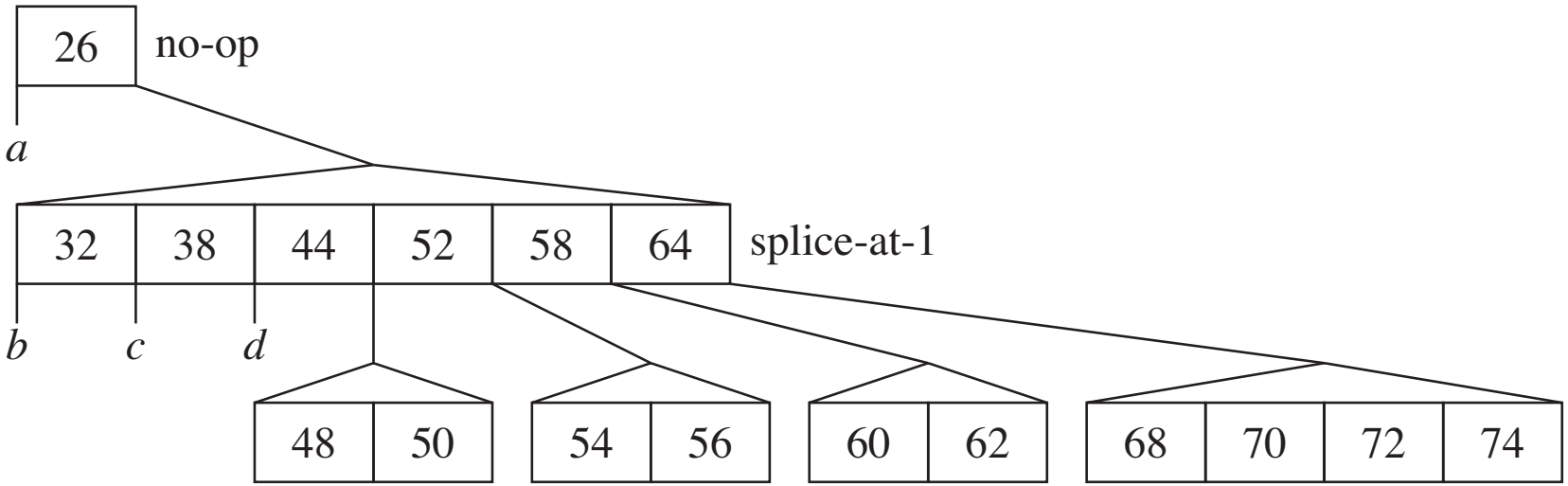
(a) Initial tree. Remove 64.



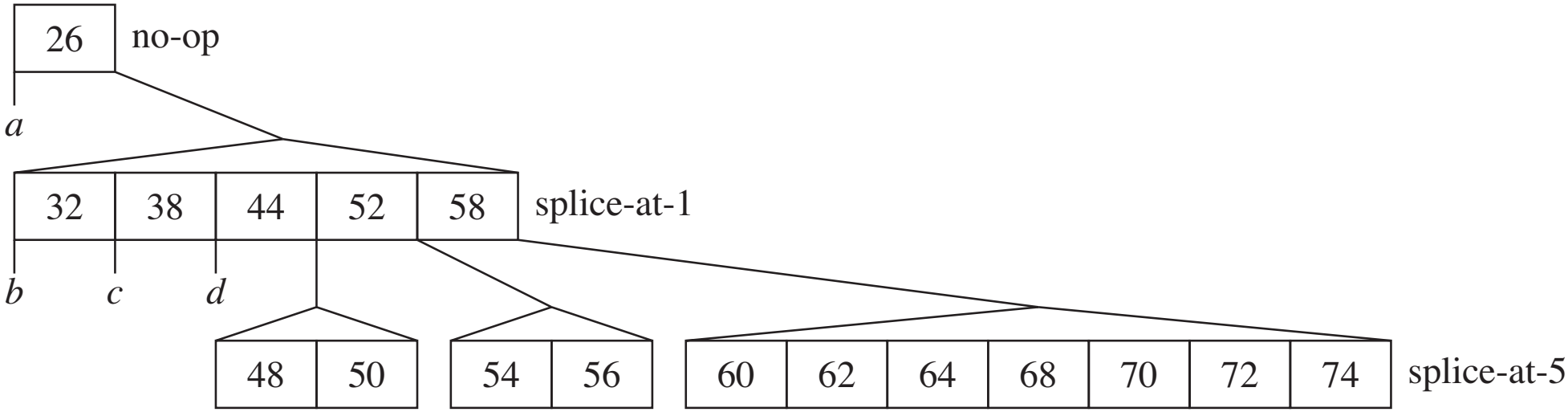
(b) Call helper with no-op command.



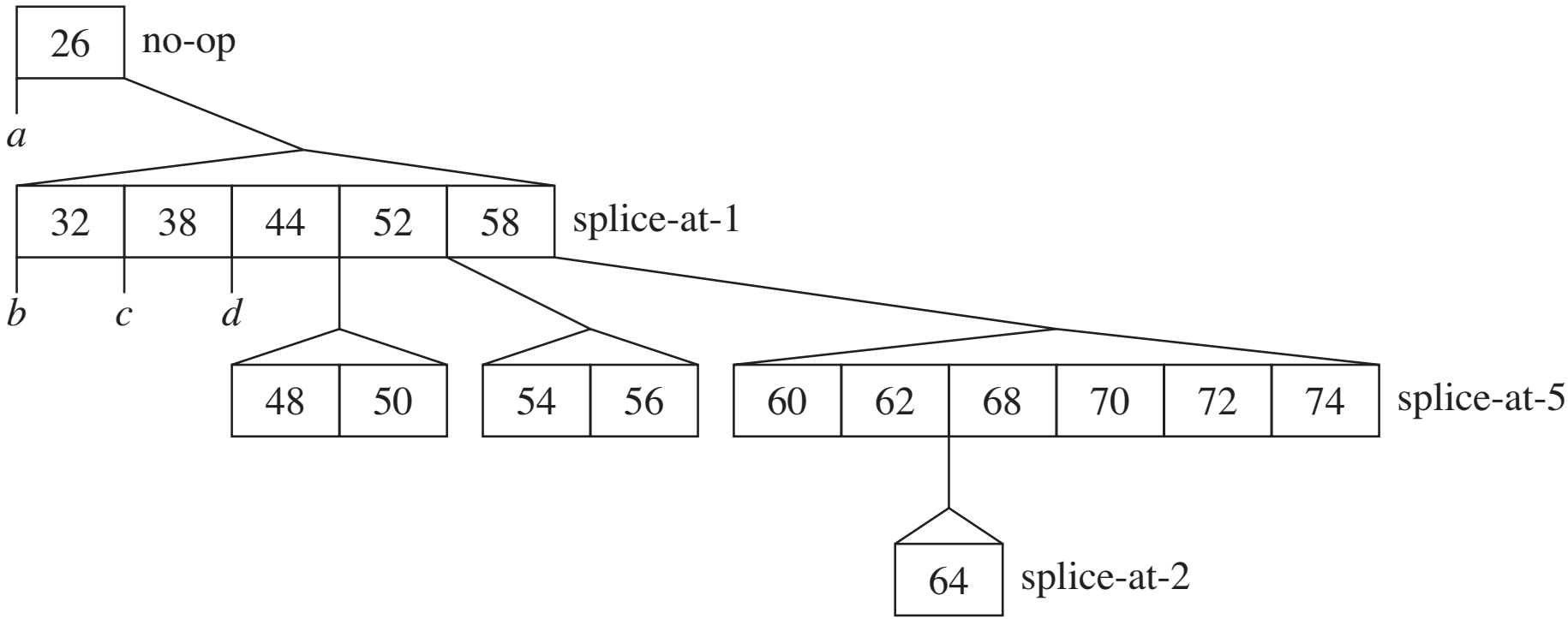
(c) k = 1. Push 44 down. Call helper with splice-at-1 command.



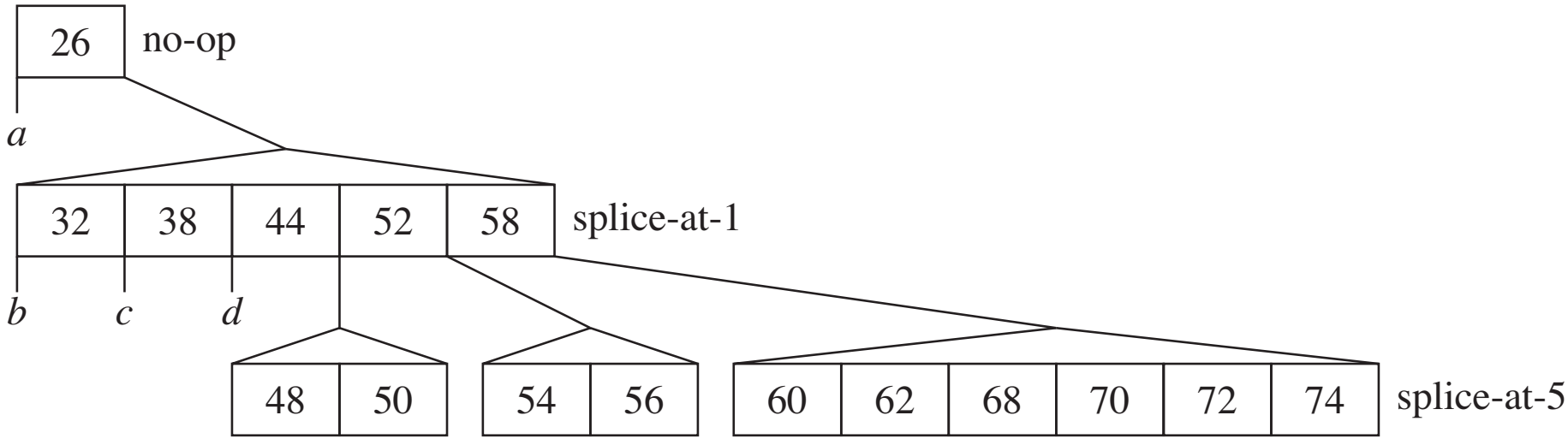
(c) `k` = 1. Push 44 down. Call helper with splice-at-1 command.



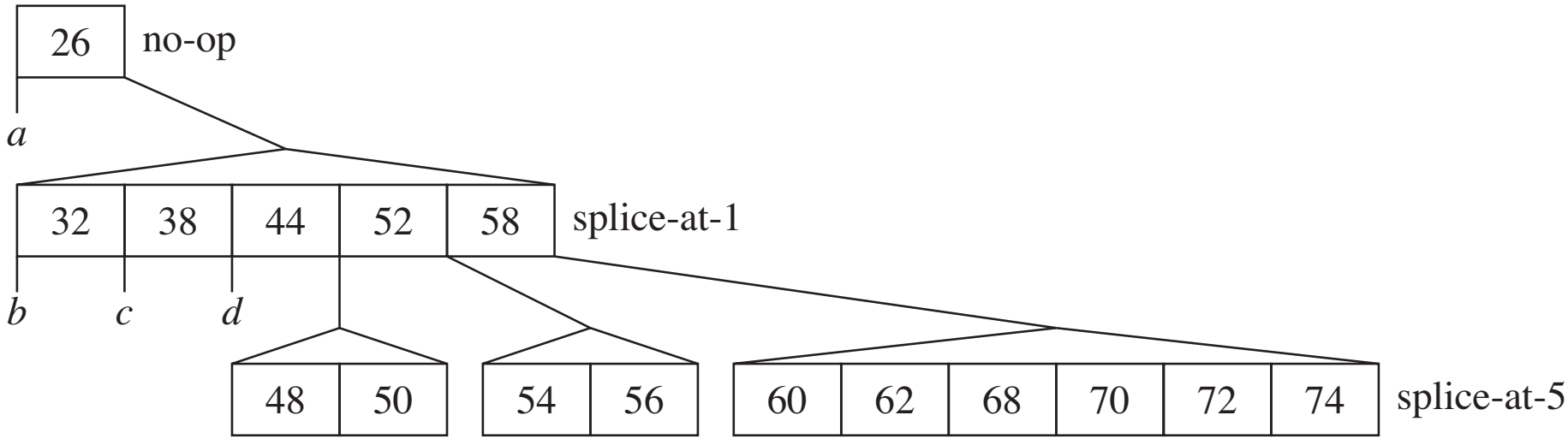
(d) `k` = 5. Push 64 down. Call helper with splice-at-5 command.



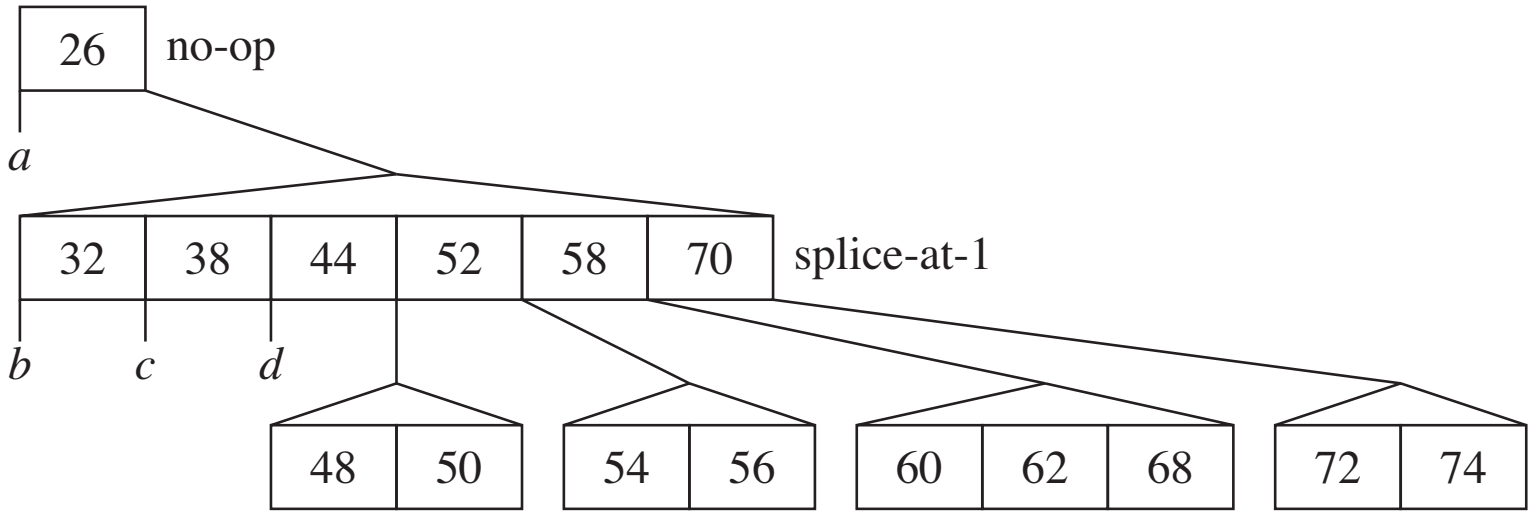
(e) `k` = 2. Push 64 down. Call helper with splice-at-2 command.



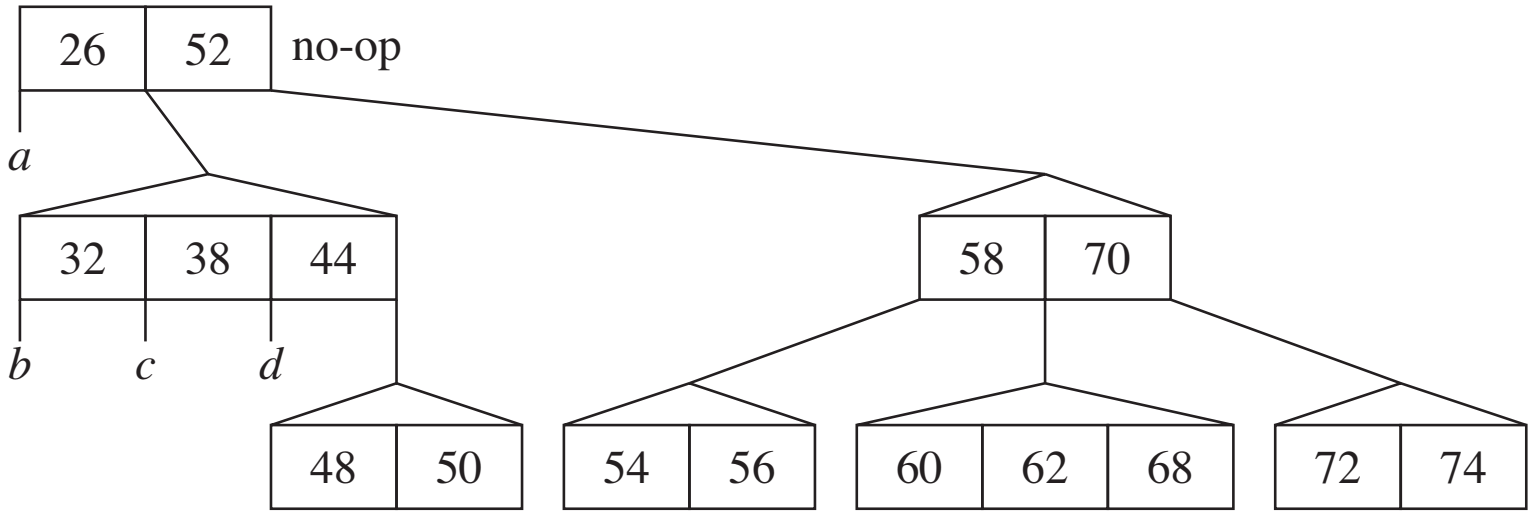
(f) Split down at 0. This tree becomes empty. Do not execute splice-at-2 command.



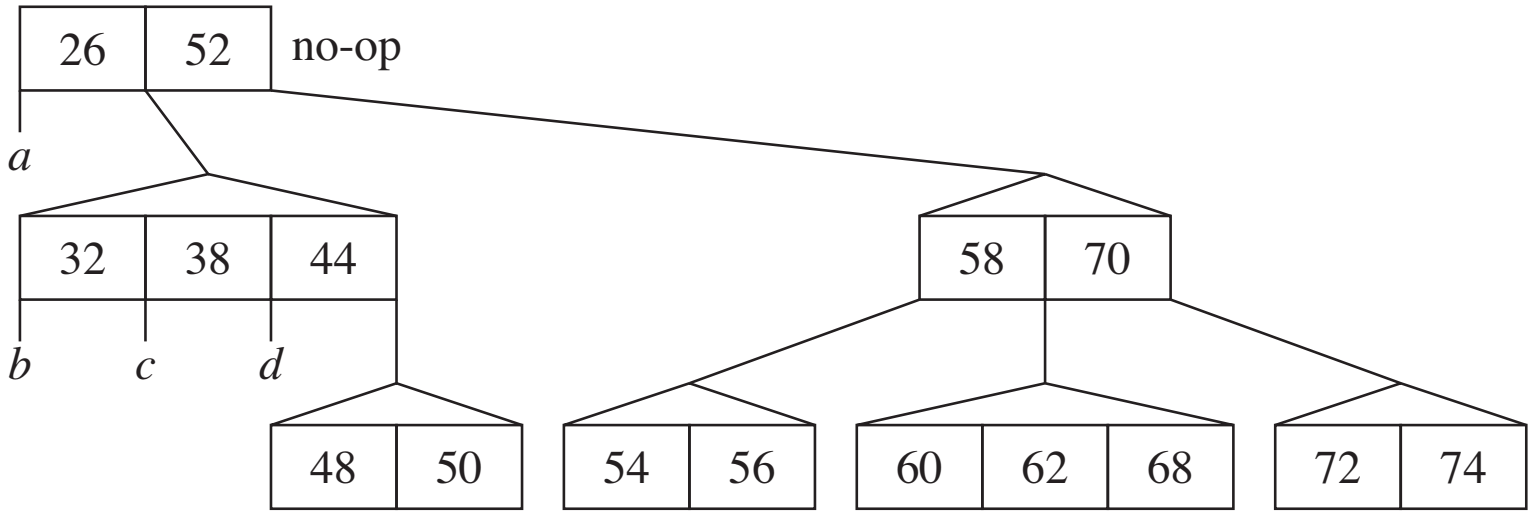
(f) Split down at 0. This tree becomes empty. Do not execute splice-at-2 command.



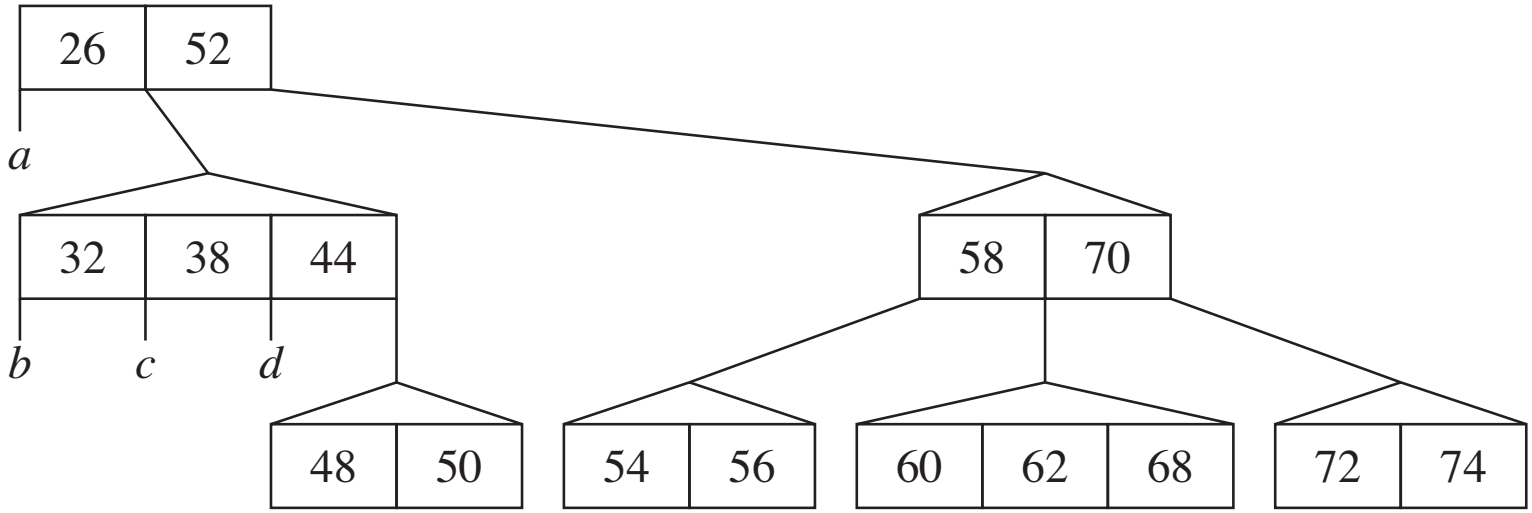
(g) Call `splitUpAndApply` with splice-at-5 command.



(h) Call `splitUpAndApply` with splice-at-1 command.



(h) Call `splitUpAndApply` with splice-at-1 command.



(i) Call `splitUpAndApply` with no-op command.

Design Patterns for Self-Balancing Trees

Dung ("Zung") Nguyen and Stephen B. Wong
Dept. of Computer Science
Rice University
Houston, TX 77005
dxnguyen@rice.edu, swong@rice.edu

Object-Oriented Programming, Systems, Languages and Applications
(OOPSLA) 2002 Educator Symposium
November 2002

Abstract

We describe how we lead students through the process of specifying and implementing a design of mutable tree data structures as an object-oriented framework. Our design entails generalizing the visitor pattern in which the tree structure serves as host with a varying number of states and the algorithms operating on the tree act as visitors.

We demonstrate the capabilities of our tree framework with an object-oriented insertion algorithm and its matching deletion algorithm, which maintain the host tree's height balance while constrained to a maximum number of elements per node. We implement our algorithms in Java and make extensive use of anonymous inner classes. The key design elements are commands manufactured on the fly as anonymous inner objects. Their closures provide the appropriate context for them to operate with little parameter passing and thus promote a declarative style of programming with minimal flow control, reducing code complexity.