Efficient Data Structures

Data Structures and Algorithms (094224)

Tutorial 6

Winter 2022/23

Question 1

A *labeled binary tree* is a binary tree with a unique label in each node

Prove/Disprove:
Let T be a binary tree with n nodes and K a set of n distinct keys. There exists a single mapping of keys from K to nodes in T such that the resulting labeled binary tree maintain the binary search tree property.

The binary search tree property for labeled binary tree:
if y is a node in the left (resp., right) subtree of node x, then $label(y) < label(x)$ (resp., $label(y) > label(x)$)

Solution

Definition:

- The ith *order statistic* ⟨סטטיסטי הסדר⟩ of a set of n elements is the ith smallest element
  - The minimum is the first order statistic ($i = 1$)
  - The maximum is the nth order statistic ($i = n$)
- Example:
  - $S = \{10, 1, 5, 18, 7, 3, 4\}$. The linear order is $(1, 3, 4, 5, 7, 10, 18)$
    - The second order statistic is 3
    - The fourth order statistic is 5

Solution

Lemma

Let T be a binary tree with root r and n nodes, K a set of n distinct keys, ℓ the left child of r and s_ℓ the number of nodes in the subtree rooted at ℓ. If an assignment of labels from K to the nodes of T satisfies the binary search tree property, then $label(r)$ is the $s_\ell + 1$ order statistic of K.

Proof.

- For every node v of T in the left subtree of r it holds that $label(v) < label(r)$
  - T maintain the binary search tree property
- For every node v of T in the right subtree of r it holds $label(v) > label(r)$
  - T maintain the binary search tree property
- In the right subtree of r there are $n - s_\ell - 1$ nodes
- $label(r)$ is larger then s_ℓ keys in K and smaller then $n - s_\ell - 1$
- $label(r)$ is the $s_\ell + 1$ order statistic of K

Solution — cont.

Claim: There exists a single mapping of keys from K to nodes of T such that the resulting labeled binary tree maintain the binary search tree property

Proof. By induction on h, the height of T

- Base: holds for $h = 0$ since there is only one node and key
- Hypothesis: The claim holds for binary tree T with height $< h$
- Step:
- Let r be the root of T, ℓ its left child and i its right child
- Let s_ℓ be the number of nodes in the subtree rooted at ℓ
- Let x be the $s_\ell + 1$ order statistic of K
- By Lemma $label(r) = x$
- Let $L = \{k \in K \mid k < x\}$ and $I = \{k \in K \mid k > x\}$
  - $|L| = s_\ell$, $|I| = n - s_\ell - 1$

Solution — cont.

- The keys in the left subtree must be from L
- The keys in the right subtree must be from I
- The subtree rooted at ℓ has height $< h$. By induction hypothesis there exists a single mapping of keys from L to the nodes of the subtree rooted at ℓ such that the resulting labeled tree satisfies the binary search tree property
- The subtree rooted at i has height $< h$. By induction hypothesis there exists a single mapping of keys from I to the nodes of the subtree rooted at i such that the resulting labeled tree satisfies the binary search tree property

1 Binary search trees

2 Augmenting Data Structures

3 Non Distinct Keys

Rank and Select Operations

- Given a set S of n elements
- The $rank(x)$ operation on an element $x \in S$ returns its position in the linear order of the elements
- The $select(i)$ operation returns the ith order statistic of S

Example:

- $S = \{10, 1, 5, 18, 7, 3, 4\}$. The linear order is $(1, 3, 4, 5, 7, 10, 18)$
- $rank(1) = 1$, $rank(3) = 2$, $rank(18) = 7$
- $select(1) = 1$, $select(2) = 3$, $select(7) = 18$

Question 2

Design a dynamic data structure D with distinct keys that supports operations

- $\texttt{Init}(D)$ – initialize a new and empty data structure
- $\texttt{Insert}(D, x)$ – insert a new object x into D
- $\texttt{Delete}(D, x)$ – delete object x from D
- $\texttt{Rank}(x)$ – returns the position of $x.key$ in the linear order of keys in D

such that the run time of $\texttt{Init}(D)$ is $O(1)$ and the run time of $\texttt{Insert}(D, x)$, $\texttt{Delete}(D, x)$, and $\texttt{Rank}(x)$ is $O(\log n)$ where n is the number of elements currently stored in the data structure.

Augmenting Data Structures

- Some new required operations on a data set may be implemented using a "textbook" data structure
  - Doubly linked list, queue, binary search tree, 2-3 tree,...
- Usually no need to create an entirely new type of data structure
- Often it will suffice to augment a textbook data structure by storing additional information in it
- Augmenting a data structure is not always straightforward
  - Added information must be updated and maintained
- Suggested process:
  1. Choose an underlying data structure
  2. Determine additional information to maintain in the underlying data structure
  3. Verify that we can maintain the additional information for the basic modifying operations on the underlying data structure
  4. Develop new operations

2-3 tree

2-3 tree ⟨2-3 עץ⟩

- Rooted tree T satisfying:
  - Each internal node has degree 2 or 3
  - All leaves are at the same level
- Objects are stored only at the leaves
  - Internal nodes are there for organization purposes
- Data structure's attribute:
  - *root* = pointer to root of T
- Node attributes:
  - *left* = pointer to left child in T
  - *middle* = pointer to middle child in T
  - *right* = pointer to right child in T
  - p = pointer to parent in T
  - *key* = maximum key in its subtree
  - Leaves have additional satellite attributes (storing the actual data)
- When degree is 2, left and middle children are in use
- The 2-3 tree property:

  keys in left subtree $<$ keys in middle subtree $<$ keys in right subtree

Solution

- We will show an implementation based on a 2-3 tree
  - D will be an augmented 2-3 tree
- Each node v of the 2-3 tree will have (in addition) an attribute *size*
  - $v.size = \#$ of leaves (other than the sentinels) in the subtree rooted at v (including v)
  - Updating the value of *size* attribute will occur during `Insert`(D, x) and `Delete`(D, x) operations in $O(\log n)$ time each

`Init(D)`: (high-level)

- Similar to `2_3_Init` with the addition that the *size* attribute of sentinels and root is set to 0

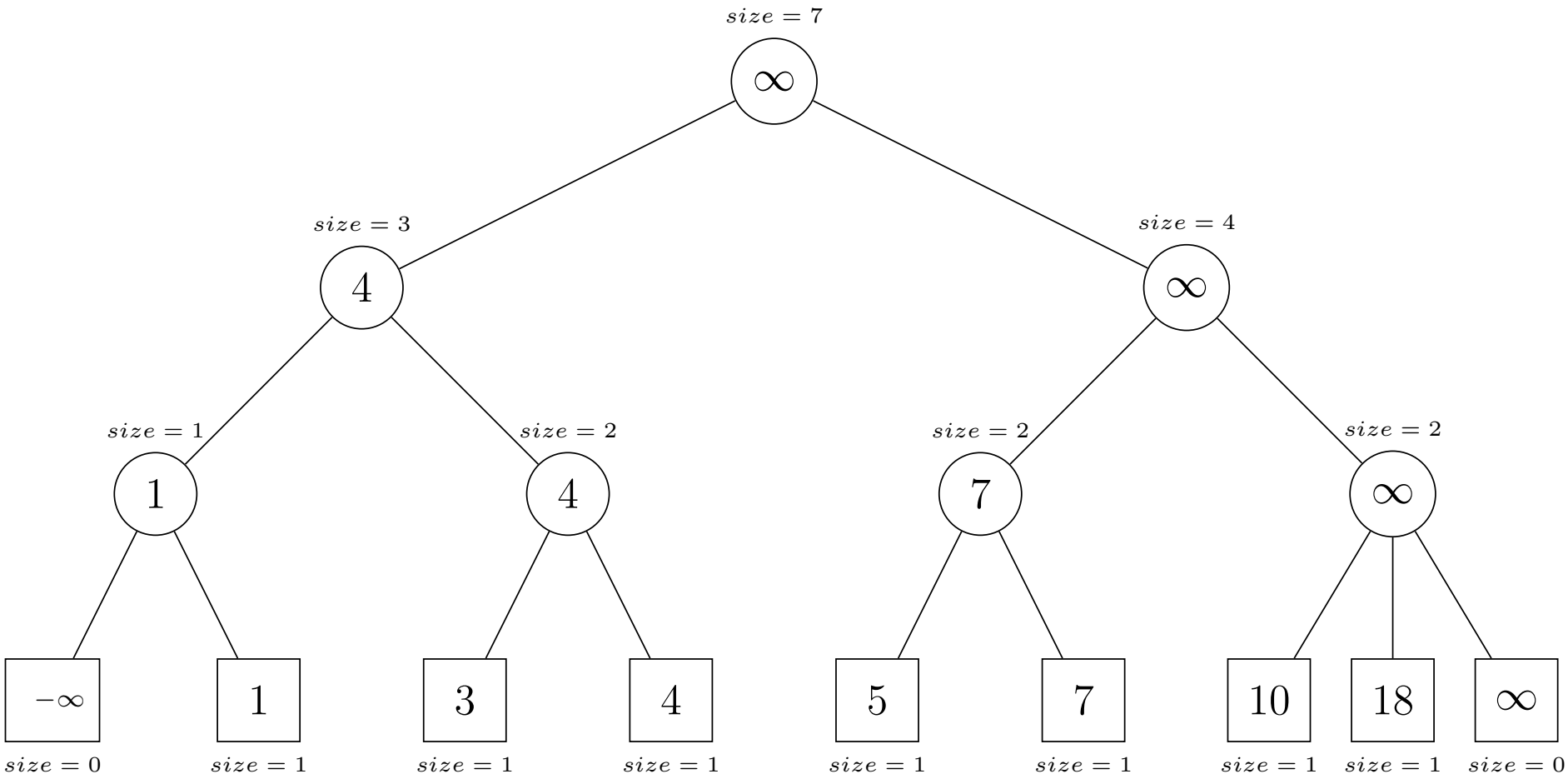
Solution — cont.

Returns the position of $x.key$ in the linear order of elements in D

`Rank`(x)

```
1:  rank = 1
2:  y = x.p
3:  while y ≠ NIL do
4:      if x == y.middle then
5:          rank = rank + y.left.size
6:      else if x == y.right then
7:          rank = rank + y.left.size + y.middle.size
8:      x = y
9:      y=y.p
10: return rank
```

Solution — example



Solution — cont.

Complexity Analysis:

- Each iteration of the while loop – one level up in the 2-3 tree
- In total – $O(height(D))$
- Recall, the height of a 2-3 tree is $O(\log n)$
- Time complexity of `Rank`(x) is $O(\log n)$

Solution — cont.

`Insert`(D, x): (high-level)

- We will follow `2_3_Insert(T, z)` to describe the insertion of a new node to D and the process of updating the *size* attributes
  - Recall, z is the node to be inserted
- Set $z.size = 1$
- `2_3_Insert`(T, z) searches for the position to insert node z. Let y be its parent
- If $deg(y) = 2$, insert z and update $y.size = y.size + 1$ (in `Insert_And_Split`(x, z))
  - Update *size* values of all ancestors of y (add one to their previous value)
- Otherwise, update *size* attributes of y and the new node created during `Insert_And_Split`(x, z) (sum of children's *size* attribute)
  - Same problem in higher level
- If a new root needs to be added update its *size* value
  - Sum of children's *size* attribute
- We only added $O(1)$ time at each level in total $O(height(D))$

Solution — cont.

`Delete`(D, x): (high-level)

- We will follow `2_3_Delete`(T, z) to describe the deletion of a node from D and the process of updating the *size* attributes
  - Recall, z is the node to be deleted
- Let $y = z.p$ and delete z
- Update $y = y.size - 1$
- If $deg(y) = 2$ after deletion of z
  - Update *size* values of all ancestors of y (subtract one from their previous value)
- Otherwise, update *size* attribute of y and y sibling during `Borrow_Or_Merge`(y) (sum of children's *size* attribute)
  - Same problem at higher level
- We only added $O(1)$ time at each level In total $O(height(D))$

Question 3

Design a dynamic data structure D with distinct keys that supports operations

- $\texttt{Init}(D)$ – initialize a new and empty data structure
- $\texttt{Insert}(D, x)$ – insert a new object x into D
- $\texttt{Delete}(D, x)$ – delete object x from D
- $\texttt{Select}(D, i)$ – returns a pointer to the element with the ith smallest key in D if exists, otherwise returns NIL

such that the run time of $\texttt{Init}(D)$ is $O(1)$ and the run time of $\texttt{Insert}(D, x)$, $\texttt{Delete}(D, x)$, and $\texttt{Select}(D, i)$ is $O(\log n)$ where n is the number of elements currently stored in the data structure.

Solution

- We will use 2-3 tree to implement the desired dynamic data structure with additional attribute *size* as defined in question 2
- Since we maintain the same attribute *size* we will use the same `Insert`(D, x), `Delete`(D, x) and, `Init`(D) as in question 2

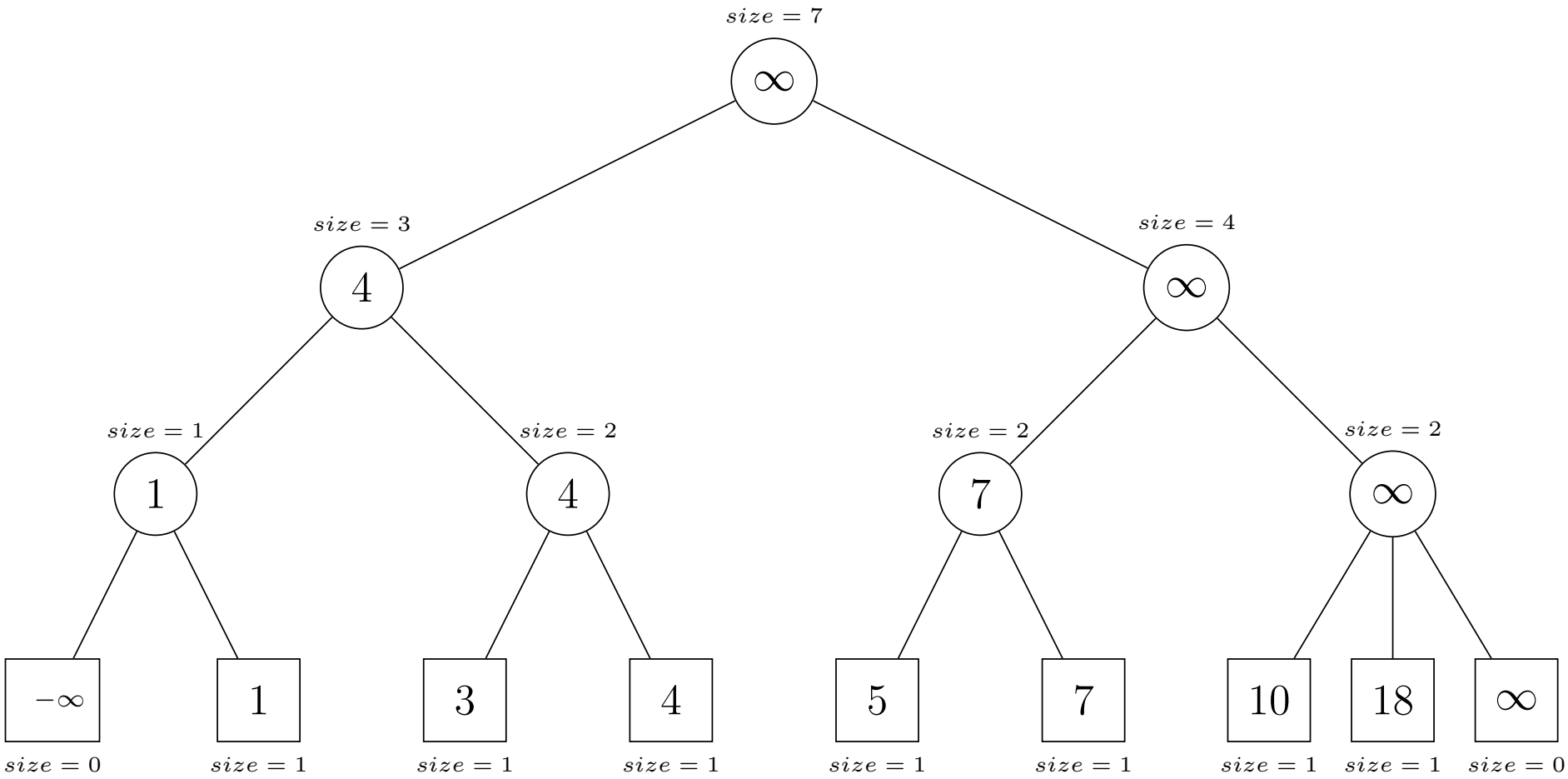
Solution — cont.

Returns a pointer to the element with the ith smallest key in the 2-3 tree rooted at x if such key exists, otherwise returns *NIL*

`Select_Rec`(x, i)

1: **if** $x.size < i$ **then**
2: return *NIL*
3: **if** x is a leaf **then**
4: return x
5: $s\_left = x.left.size$
6: $s\_left\_middle = x.middle.size + x.left.size$
7: **if** $i \leq s\_left$ **then**
8: return `Select_Rec`$(x.left, i)$
9: **else if** $i \leq s\_left\_middle$ **then**
10: return `Select_Rec`$(x.middle, i - s\_left)$
11: **else** return `Select_Rec`$(x.right, i - s\_left\_middle)$

Solution — example



Solution — cont.

Notice:

- In order to implement `Select`(D, i) we need to run `Select_Rec`$(D.root, i)$

Complexity Analysis:

- Each recursion call – one level down in the 2-3 tree rooted at x
- In total – $O(height(x))$
- Recall, the height of 2-3 tree is $O(\log n)$
- Time complexity of `Select`(D, i) is $O(\log n)$

Question 4

Design a dynamic data structure D with distinct keys that supports operations

- `Init`(D) – initialize a new and empty data structure
- `Insert`(D, x) – insert the element x into D
- `Delete`(D, x) – delete element x from D
- `Sum_of_smaller`(D, k) – returns the sum of keys in D that are less or equal to k

such that the run time of `Init`(D) is $O(1)$ and the run time of `Insert`(D, x), `Delete`(D, x), and `Sum_of_smaller`(D, k) is $O(\log n)$ where n is the number of elements currently stored in the data structure.

Solution

- We will show an implementation based on a 2-3 tree
  - D will be an augmented 2-3 tree
- Each node v of the 2-3 tree will have (in addition) an attribute *sum*
  - $v.sum$ = sum of keys in the leaves (other than the sentinels) in the subtree rooted at v (including v)
  - Updating the attribute *sum* will occur during `Insert`(D, x) and `Delete`(D, x) operations in $O(\log n)$ time each

`Init(D)`: (high-level)

- similar to `2_3_Init` with the addition that the *sum* attribute of sentinels and root is set to 0

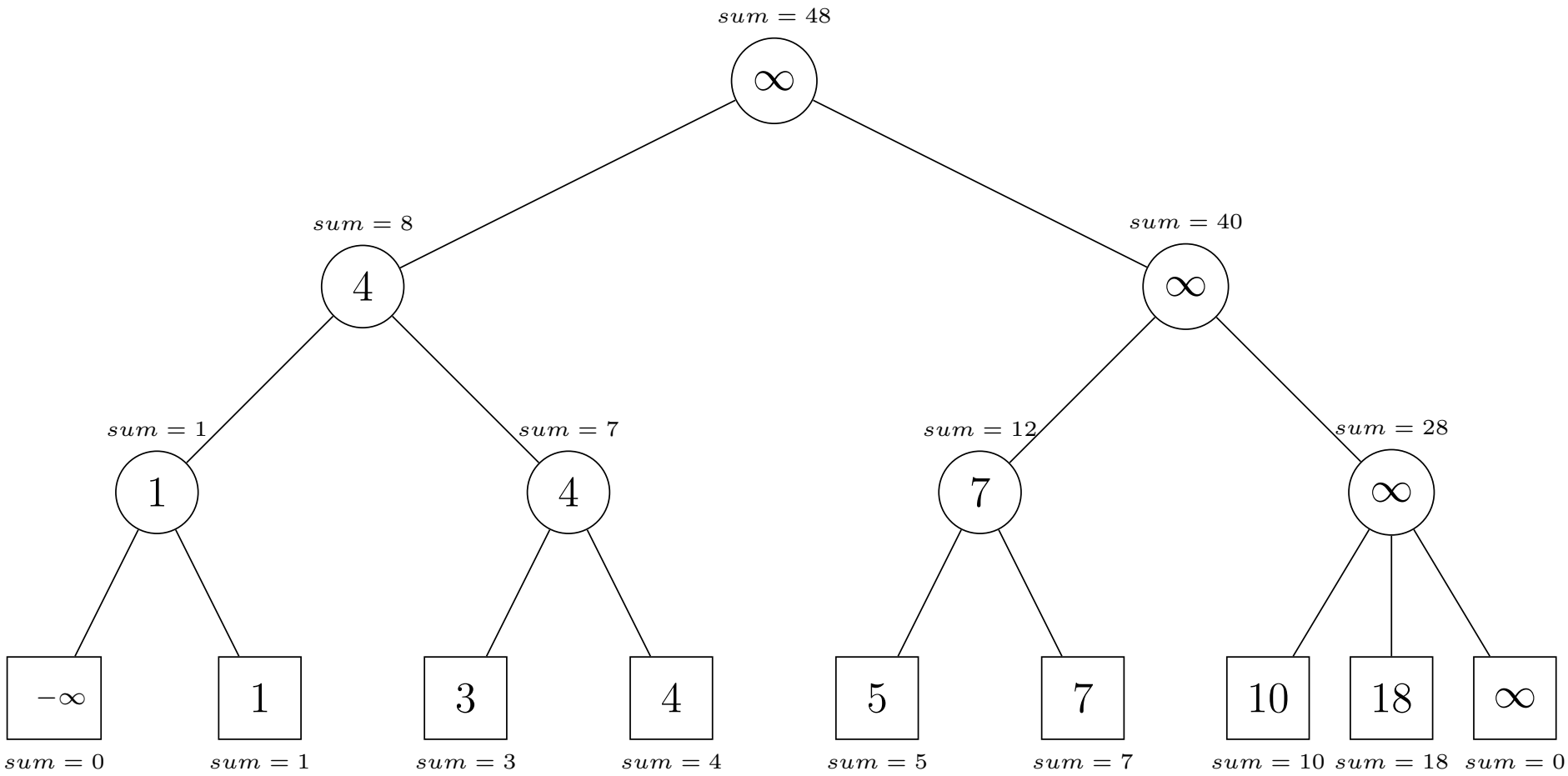
Solution — cont.

Returns the sum of keys that are $\leq k$ in the subtree rooted at x

`Sum_Of_Smaller_Rec`(x, k)

1: **if** x is a leaf **then**
2: **if** $x.key \neq -\infty$ **then**
3: **if** $x.key \leq k$ **then**
4: return $x.key$
5: **else**
6: return 0
7: **else**
8: return: Error no keys smaller then k
9: **if** $k \leq x.left.key$ **then**
10: return `Sum_Of_Smaller_Rec`$(x.left, k)$
11: **else if** $k \leq x.middle.key$ **then**
12: return $x.left.sum + Sum\_Of\_Smaller\_Rec(x.middle, k)$
13: **else** return
$x.left.sum + x.middle.sum + Sum\_Of\_Smaller\_Rec(x.right, k)$

Solution — example



Solution — cont.

Notice:

- In order to implement `Sum_Of_Smaller`(D, k) we need to run `Sum_Of_Smaller_Rec`$(D.root, k)$

Complexity Analysis:

- Each recursion call – one level down in the 2-3 tree rooted at x
- In total – $O(height(x))$
- Recall, the height of 2-3 tree is $O(\log n)$
- Time complexity of `2_3_Select`(x, i) is $O(\log n)$

Solution — cont.

`Insert`(D, x): (high-level)

- We will follow `2_3_Insert(T, z)` to describe the insertion of a new node to D and the process of updating the *sum* attributes
  - Recall, z is the node to be inserted
- Set $z.sum = z.key$
- `2_3_Insert`(T, z) searches for the position to insert node z. Let y be its parent
- If $deg(y) = 2$, insert z and update $y.sum = y.sum + z.key$ (in `Insert_And_Split`(x, z))
  - Update *sum* values of all ancestors of y (add $z.key$ to their previous value)
- Otherwise, update *sum* attributes of y and the new node created during `Insert_And_Split`(x, z) (sum of children's *sum* attribute)
  - Same problem in higher level
- If a new root needs to be added update its *sum* value
  - Sum of children's *sum* attribute
- We only added $O(1)$ time at each level In total $O(height(D))$

Solution — cont.

`Delete`(D, x): (high-level)

- We will follow `2_3_Delete`(T, z) to describe the deletion of a node from D and the process of updating the *sum* attributes
  - Recall, z is the node to be deleted
- Let $y = z.p$ and delete z
- Update $y = y.sum - z.key$
- If $deg(y) = 2$ after deletion of z
  - Update *sum* values of all ancestors of y (subtract $z.key$ from their previous value)
- Otherwise, update *sum* attribute of y and y's sibling during `Borrow_Or_Merge`(y) (sum of children's *sum* attribute)
  - Same problem at higher level
- We only added $O(1)$ time at each level In total $O(height(D))$

Question 5

Design a dynamic data structure D with distinct keys where each element has a numerical attribute *value* that supports operations

- `Init`(D) – initialize a new and empty data structure
- `Insert`(D, x) – insert the element x into D
- `Delete`(D, x) – delete element x from D
- `Largest_Value`(D) – returns the largest value of elements in D

such that the run time of `Init`(D) and `Largest_Value`(D) is $O(1)$ and the run time of `Insert`(D, x) and `Delete`(D, x) is $O(\log n)$ where n is the number of elements currently stored in the data structure.

Solution

- We will show an implementation based on a 2-3 tree
  - D will be an augmented 2-3 tree
- Each node v of the 2-3 tree will hold the attribute *value*
  - $v.value$ = largest value in the subtree rooted at v (including v)
  - Updating the attribute *value* will occur during `Insert`(D, x) and `Delete`(D, x) operations in $O(\log n)$ time each

`Init(D)`: (high-level)

- Similar to `2_3_Init` with the addition that the *value* attribute of sentinels and root is set to $-\infty$

Solution — cont.

Returns the largest value in D

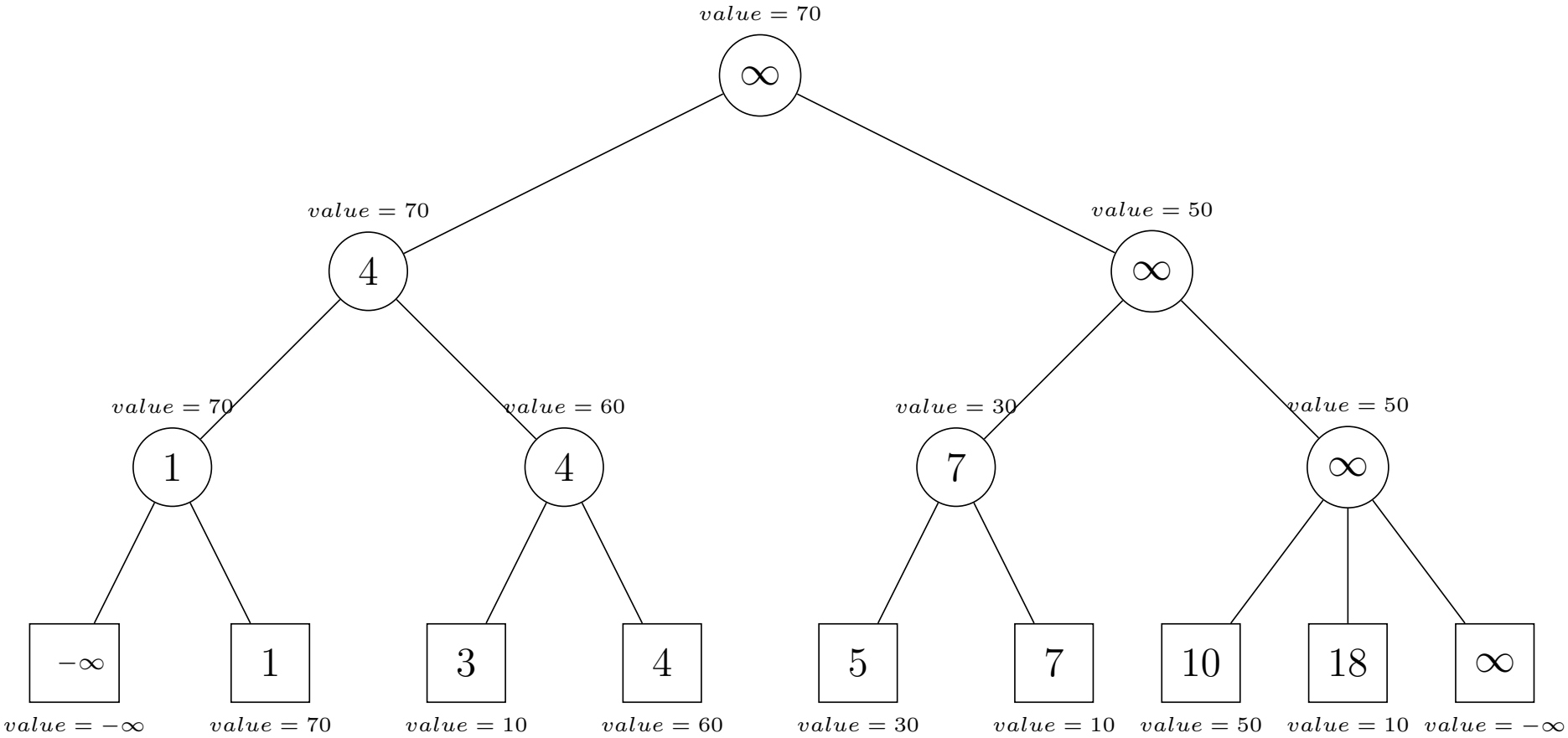
`Largest_Value`(D)

1: **if** $D.root.value \neq -\infty$ **then** ▷ check if D is empty
2: return $D.root.value$
3: **else**
4: return error: D is empty

Complexity Analysis:

- Only constant time operations – $O(1)$

Solution — example



Solution — cont.

`Insert`(D, x): (high-level)

- We will follow `2_3_Insert`$(\mathtt{T}, \mathtt{z})$ to describe the insertion of a new node to D and the process of updating the *value* attributes
  - Recall, z is the node to be inserted
- `2_3_Insert`(T, z) searches for the position to insert node z. Let y be its parent
- If $deg(y) = 2$, insert z and update $y.value = \max_{w \in Child(y)}\{w.value\}$ (in `Insert_And_Split`(x, z))
  - Update *value* of all ancestors of y (max of children's *value* attribute)
- Otherwise, update *value* attributes of y and the new node created during `Insert_And_Split`(x, z) (max of children's *value* attribute)
  - Same problem in higher level
- If a new root needs to be added update its *value* attribute
  - Max of children's *value* attribute
- We only added $O(1)$ time at each level In total $O(height(D))$

Solution — cont.

`Delete`(D, x): (high-level)

- We will follow `2_3_Delete`(T, z) to describe the deletion of a node from D and the process of updating the *value* attributes
  - Recall, z is the node to be deleted
- Let $y = z.p$ and delete z
- Update $y.value = \max_{w \in Child(y)}\{w.value\}$
- If $deg(y) = 2$ after z delete
  - Update *value* of all ancestors of y (max of children's *value*)
- Otherwise, update *value* attribute of y and y's sibling during `Borrow_Or_Merge`(y) (max of children's *value* attribute)
  - Same problem at higher level
- We only added $O(1)$ time at each level In total $O(height(D))$

Non distinct keys

- All data structures we saw assumed uniqueness of keys
- This assumption can be easily lifted with slight modifications in the data structure operations
- This assumption can also be lifted without changing the data structure operations
  - For example:
  - We maintain an additional data structure attribute *count* initialized to 0 once the data structure is initialized
  - After each insertion of an object the count attribute is incremented by 1
  - Let k be some object's key to be inserted
  - We define the ordered pair $(k, count)$ as the new key
  - Let (k_1, c_1) and (k_2, c_2) be two new keys
  - We define an order on keys by $(k_1, c_1) < (k_2, c_2)$ iff $k_1 < k_2 \vee (k_1 = k_2 \wedge c_1 < c_2)$