Design and Analysis of Computer Algorithms

11059005 蕭耕宏

Assignment 4

Due: May 26, 2025 (before class)

Please use A4-sized papers to write your answers for the homework sets.

1. Recall the unordered binomial tree, U_k: U_0 is a tree with one node. For $k > 0$, a U_k tree is obtained from two disjoint U_{k-1} trees by attaching the root of one to the root of the other.

   (a) Prove that, given a U_k tree for some k, there is a unique node at depth k. We let the root be at depth 0 and call the node at depth k the handle of the U_k tree. Please refer to Figure 1.

   (b) Show the following characterization of an U_k tree. Suppose a given U_k tree for some k is rooted at r and has handle v. Then,

   i. there are disjoint trees $U'_0, U'_1, ...U'_{k-1}$ not containing r, with roots $r'_0, r'_1, ...r'_{k-1}$ respectively;

   ii. U_k is obtained by attaching each r'_i to r for $0 \leq i \leq k-1$;

   iii. v is the handle of U'_{k-1}

   This decomposition of a U_k is illustrated in Figure 1.

   Solution:

   a.

   Basic Idea:

   The unordered binomial tree U_k is defined recursively: U_0 is a single node, and for $k > 0$, a U_k tree is formed by attaching the root of one U_{k-1} tree to the root of another U_{k-1} tree. We use induction to prove that this construction ensures exactly one node exists at depth k, which we call the handle.

   Detailed Approach:

   we can prove this by induction on k.

   For k = 0, the tree has only one node, which is the handle.

   For k = a > 0, we assume there is a unique code at depth i for U_a.

   For k = a + 1, U_{a+1} is constructed by linking two U_a trees.

   Let the U_a whose root becomes the root of U_{a+1} be T_1 and the other be T_2.

   For T_1, the depth of all nodes remains the same, so the handle of T_1 is still at the depth a.

   The depth of all T_2's nodes, however, is increased by 1.

   That is, the handle of T_2 is at depth a + 1 while being the only node at that depth.

   Thus, there is a unique node at depth a + 1 in U_{a+1}.

   Correctness

   The recursive construction ensures that only the handle of the attached U_a tree(i.e., T_2) reaches depth a + 1, as the depth increment applies uniformly to T_2. The induction hypothesis guarantees uniquesness within each U_a tree, and the attachment process does not introduce any additional nodes at depth a + 1. Therefore, the handle of U_k is unique.

Example

Consider U_2:

- U_0 is a single node.
- U_1 is formed by attaching the root of U_0 to another U_0, resulting in a tree with depth 1.
- U_2 is formed by attaching the root of two U_1 trees, resulting in a tree with depth 2, where the handle is the unique node at depth 2.

b.

Basic Idea:

The charactorization states that a U_k tree can be decomposed into a root r, disjoint trees $U'_0, U'_1, ...U'_{k-1}$ with roots $r'_0, r'_1, ...r'_{k-1}$, where each U'_i is attached to the root r, and the handle v is the handle of U'_{k-1}. We will prove this by induction on k.

Detailed Approach:

we can prove this by induction on k. For k = 0, the tree has only one node.

- the disjoint trees are empty.
- U_0 is obtained by attaching the empty trees to r.
- the handle of U_0 is the root itself.

For k = a > 0, we can assume the statements is true for U_a.

- the disjoint trees are $U'_0, U'_1, ...U'_{a-1}$ with roots $r'_0, r'_1, ...r'_{a-1}$.
- U_a is obtained by attaching each r'_i to r for $0 \leq i \leq a-1$.
- the handle of U'_a is the root of $U'(a-1)$

For k = a + 1, U_{a+1} is constructed by linking two U_a trees. Let the U_a whose root becomes the root of U_{a+1} be T_1 and the other be T_2.

- Since the root of T_1 is the root of U_{a+1}, U_{a+1} have disjoint trees $U'_0, U'_1, ...U'_{a-1}$ with roots $r'_0, r'_1, ...r'_{a-1}$. T_2 is the missing piece for U_{a+1}'s disjoint trees since since its root is the child of the root of T_1, a.k.a. the root of U_{a+1}.
- together with the root of $T_2(U_a)$, U_{a+1} is obtained by attaching each r'_i to the root of U_{a+1} for $0 \leq i \leq a$.
- To check whether the handle of U_{a+1} is the handle of U'_a, we can see that the root of T_2 is the child of root of U_{a+1}. This operation increases the depth of all nodes in T_2 by 1. Thus, the handle of U'_a is at depth a + 1, which becomes the handle of U_{a+1}.

Correctness

The recursive construction ensures that the disjoint trees are correctly formed, and the handle of U_k is indeed the handle of U'_{k-1}. The induction hypothesis guarantees that the decomposition holds for each U_a tree, and the attachment process does not introduce any additional nodes at depth k. Therefore, the characterization holds.

Example

Consider U_2:

- U_0 is a single node.
- U_1 is formed by attaching the root of U_0 to another U_0, resulting in a tree with depth 1.
- U_2 is formed by attaching the root of two U_1 trees, resulting in a tree with depth 2, where the handle is the unique node at depth 2.
- The disjoint trees are the two U_1 trees, and the handle of U'_1 is the root of one of them.

2. A research project needs a data structure D that supports the operations min, extract-min, max, extract-max, and insert(x). You know n, the estimated maximum number of elements in D, but

you do not know anything about the values of the elements. The implementation can use only one array A of size n + 1, plus a constant number of additional variables. One array location should contain one data element. A student who has taken Algorithm course last year proposes the following heap-like structure D:

(I) The root of the tree contains no element.

(II) The left subtree is a heap with min value at the root (min-heap), and the right subtree is a heap with max value at the root (max-heap).

(III) Each element of the min-heap is less than or equal to the element in the same position in the max-heap.

(IV) If there are k, k ≤ n, elements in D, they are in array locations 2, . . . , k + 1.

(V) If an element in the min-heap does not have a corresponding element in the maxheap, then the parent of the missing element is larger than the element in the minheap. This can happen only for leaves in the min-heap.

Please answer the followings.

(a) How fast can the minimum and maximum element be reported? How is the index of the "corresponding element" determined? Give a time bound.

(b) The student claims that operations extract-min, extract-max and insert(x) can each be implemented in O(log n) time. Either describe how each can be implemented, analyze the time complexity, and address the correctness, or prove that the structure does not support the operations as claimed.

Solution:
(a)

Basic Idea:
The proposed data structure D consists of a min-heap and a max-heap, where the min-heap contains the minimum elements and the max-heap contains the maximum elements. The minimum element can be found at the root of the min-heap, and the maximum element can be found at the root of the max-heap. The index of the corresponding element is determined by calculating the index based on the properties of heaps.

Detailed Approach:
the minimum and maximum element can be reported in O(1) time. The index of the corresponding element can be determined by calculating $s = \lfloor \log_2(i) \rfloor$ where i is the index of the element. Let the index of a max-heap's element be j and the index of a min-heap's element be k. Then $j = k + 2^s$. The time bound for this operation is O(1) since it only requires a few arithmetic operations.

Correctness:
The correctness of the proposed structure is based on the properties of heaps. The min-heap ensures that the minimum element is at the root, and the max-heap ensures that the maximum element is at the root. The index calculation ensures that the corresponding elements in the min-heap and max-heap are correctly identified.

Example:
Consider a min-heap and max-heap with the following elements:

```
Min-Heap: [2, 4, 6, 8]
Max-Heap: [10, 12, 14, 16]
```

The minimum element is 2 (at index 2), and the maximum element is 10 (at index 3). The corresponding index for the minimum element is calculated as follows:

```
s = floor(log_2(2)) = 1
j = 2 + 2^1 = 4
```

The corresponding index for the maximum element is 4, which matches the index in the max-heap.

Time Complexity:
The time complexity for reporting the minimum and maximum elements is O(1) since it only requires accessing the root of the min-heap and max-heap, respectively. As for determining the index of the corresponding element, it requires a few arithmetic operations, which also takes O(1) time.

(b)

Basic Idea:
The operations manipulate the min-heap and max-heap while maintaining their properties and the correspondence between elements (Property III). Extract-Min and Extract-Max remove the root of the respective heap and restore the heap property, while Insert adds a new element and adjusts both heaps to maintain all invariants.

Detailed Approach:
extract-max, extract-min, and insert(x) can be implemented in O(log n) time. The implementation is as follows:

- extract-min: we can implement this by removing the root of the min-heap and replacing it with the last element in the min-heap. Then we can call Sift-Down-Min to restore the heap property. The time complexity is O(log n) since Sift-Down-Min takes O(log n) time.

```
EXTRACT-MIN(D, x):
1  x = A[2]
2  A[2] = A[k+1]
3  k = k - 1
4  Sift-Down-Min(A, 2, k)
```

- extract-max: the same idea as extract-min but for the max-heap.

```
EXTRACT-MAX(D, x):
1  x = A[3]
2  A[3] = A[k+1]
3  k = k - 1
4  Sift-Down-Max(A, 3, k)
```

)
- insert(x): we can implement this by inserting the new element at the end of the min-heap and then calling Sift-Up to restore the heap property. If the new element is larger than its

corresponding max-heap element, we swap them and call Sift-Down on the max-heap to restore its property.

```
INSERT(D, x):
 1  k = k + 1
 2  A[k+1] = x
 3  i = k+1
 4  if i in min-heap (e.g., i = 2, 4, 5, 8, ...):
 5        Sift-Up-Min(A, i)
 6        j = Corresponding-Index(i)
 7        if j > k+1 and A[i] ≥ Parent(j):
 8              Swap(A[i], Parent(j))
 9              Sift-Down-Max(A, j, k)
10  else:
11        Sift-Up-Max(A, i)
12        j = Corresponding-Index(i)
13        if A[i] < A[j]:
14              Swap(A[i], A[j])
15              Sift-Down-Min(A, j, k)
```

Correctness:
The correctness of the operations is based on the properties of heaps. Extract-Min and Extract-Max maintain the min-heap and max-heap properties, respectively, while Insert ensures that both heaps remain valid after adding a new element. The correspondence between elements in the min-heap and max-heap is preserved throughout these operations.

Example:
initial:

```
A = [None, 5, 10, 7, 8, 12, 11], k = 6
min-heap: [5, 7, 8]
max-heap: [10, 12, 11]
```

Extract-Min:

```
A = [None, 5, 10, 7, 8, 12, 11], k = 6
x = 5
A[2] = A[7]  # Replace root with last element
A = [None, 11, 10, 7, 8, 12], k = 5
Sift-Down-Min(A, 2, 5)  # Restore min-heap property
```

A = [None, 7, 10, 11, 8, 12], k = 5

Extract-Max:

```
A = [None, 7, 10, 11, 8, 12], k = 5
x = 10
A[3] = A[6]  # Replace root with last element
A = [None, 7, 12, 11, 8], k = 4
Sift-Down-Max(A, 3, 4)  # Restore max-heap property
```

A = [None, 7, 12, 11, 8], k = 4 Insert(x):

```
A = [None, 7, 12, 11, 8], k = 4
x = 6
A[5] = 6  # Insert new element
k = 5
Sift-Up-Min(A, 5)  # Restore min-heap property
if 5 in min-heap:
  j = Corresponding-Index(5)  # j = 7
  if j > k+1 and A[5] ≥ Parent(j):  # Check if swap needed
    Swap(A[5], Parent(j))  # Swap with parent in max-heap
    Sift-Down-Max(A, j, k)  # Restore max-heap property
else:
  Sift-Up-Max(A, 5)  # Restore max-heap property
  j = Corresponding-Index(5)  # j = 7
  if A[5] < A[j]:  # Check if swap needed
    Swap(A[5], A[j])  # Swap with corresponding max-heap element
    Sift-Down-Min(A, j, k)  # Restore min-heap property
```

x is inserted to max-heap, so Sift-Up-Max is called. However, nothing is swapped since the new element is smaller than its corresponding max-heap element. When compared to the corresponding value in min-heap, the new element is smaller, so it is swapped with the corresponding min-heap element, and Sift-Down-Min is called to restore the min-heap property.

final result:

```
A = [None, 6, 12, 7, 8, 11], k = 5
min-heap: [6, 7, 8]
max-heap: [12, 11]
```

Time Complexity:
The time complexity for each operation is O(log n) because the Sift-Down and Sift-Up operations take O(log n) time in a heap structure, and the number of elements in the heaps is at most n.

3. (Problem 19.4-2 and 19.4-4 in textbook)

Recall the rank we introduced in class for the union by rank strategy used in disjointsets data structure. Prove that every node in an n-node tree has rank at most $\lfloor \log n \rfloor$. Then, use this result to show that m operations of Make-set, Union, and Find-set on a disjoint-set forest with union by rank but without path compression run in O(m log n) time. (Please refer to the consequence of Lemma 19.7)

Solution:

Basic Idea:
In a disjoint-set forest with union by rank, a node's rank is the height of its subtree(for roots) or 0 (for non-roots after union). We show that a tree with rank r has at least 2^r nodes, so for n nodes, the rank is at most $\lfloor \log_2 n \rfloor$.

Detailed Approach:
Part 1: Prove every node's rank is at most $\lfloor \log n \rfloor$

- Definition:
  - MAKE-SET: creates a single-node tree with rank 0.
  - UNION: Equal ranks ⟶ new rank = old rank + 1; otherwise, higher rank prevails.
- Proof:

- A tree with rank r has at least 2^r nodes (two rank $r-1$ trees union to rank r, each with $\geq 2^{r-1}$ nodes).
- For n nodes, $n \geq 2^r$, so $r \leq \lfloor \log n \rfloor$.
- Holds for all roots; non-roots have rank 0 or irrelevant after union.

Part 2: m Operations in $O(m \log n)$ Time

- Operations:
  - MAKE-SET: $O(1)$.
  - UNION: $O(1)$ (plus FIND-SET calls).
  - FIND-SET: $O(\log n)$ (height $\leq \lfloor \log n \rfloor$).
- Total Time:
  - m operations, each $O(\log n)$ worst-case, yields $O(m \log n)$.

Correctness:
The correctness follows from the properties of union by rank. The rank ensures that the height of any tree in the disjoint-set forest is logarithmic in the number of nodes, which guarantees efficient operations. The union operation maintains the rank properties, and the find operation traverses up to the root, which is bounded by the logarithm of the number of nodes.

Example:
For $n = 8$, max rank is $\lfloor \log_2 8 \rfloor = 3$.

Operations like UNION involve at most 2 FIND-SET calls, each traversing at most 3 edges

For $m = 100$ operations, total time is $O(100 \log_2 8) = O(300)$, or $O(m \log n)$ in general.

Time Complexity:
The time complexity for m operations of Make-set, Union, and Find-set on a disjoint-set forest with union by rank but without path compression is $O(m \log n)$. Each operation is efficient due to the logarithmic height of the trees, ensuring that the overall complexity remains manageable even for large values of m and n.