Binomial Heap

Tel-Aviv University
Course: Data Structures semester B/2024

Student 1 Name: Antonella Campania

Student 1 ID: 324823095

Student 2 Name: Yonatan Rosh

Student 2 ID: 209135763

Class BinomialHeap

Constructor

Initializes an empty binomial heap. Sets the size to 0, and pointers `last` and `min` to `null`. Prepares the heap to accept new elements with the correct initial state.

Time Complexity: O(1) - Simple initialization of variables.

Properties

size: Maintains the count of elements within the heap.

last: Points to the last node in the circular linked list of tree roots.

min: Keeps track of the node with the minimum key value.

insert(int key, String info)

The `insert` function adds a new element to the heap. It creates a new node (HeapNode) and a new item (HeapItem) with the provided key and info. If the heap is empty, it sets this new node as both the `min` and `last`. If the heap is not empty, the new node may either be added directly if there is no conflict with existing nodes of the same rank, or it may trigger a meld operation to maintain the heap properties according to the binomial heap rules.

A new node is initially added as a separate single-node tree. If the heap contains another tree of the same rank (here, rank 0), the two trees are merged through a meld operation, combining them into a single tree of higher rank.

Time Complexity: O(log n) - The operation may require a traversal and melding of trees as the heap grows, which can be performed in logarithmic time relative to the number of elements in the heap.

deleteMin()

This function removes the smallest element from the heap, which is typically found at the root of one of the trees. The process involves removing the root node of the tree containing the minimum element and then reorganizing the heap to maintain its properties. This reorganization may involve merging the children of the removed node back into the main heap, often requiring multiple meld operations.

The children of the removed node, if any, are each added back to the heap. These children are themselves roots of smaller binomial trees, which are then merged into the existing

heap structure using the meld operation to maintain the heap order.

Time Complexity: O(log n) - Removing the minimum element and restructuring the heap typically requires operations proportional to the logarithm of the heap's size.

findMin()

The `findMin` function retrieves the smallest element in the heap without removing it. This function directly returns the item contained within the node pointed to by `min`, allowing access to the minimum value efficiently.

Time Complexity: O(1) - Directly accessing the `min` pointer requires constant time.

searchMin()

The `searchMin` function iterates over all tree roots to find the minimum key. It updates the `min` pointer to point to the root with the smallest key, ensuring that subsequent calls to `findMin` return the correct value.

Time Complexity: O(log n) - As the number of roots is logarithmic relative to the heap size, the iteration is log(n) in time complexity.

decreaseKey(HeapItem item, int diff)

Decreases the key of a specified item by a given difference. The function adjusts the key value and then checks if the heap properties are violated. If so, it uses the `swapItem` helper function to move the node upwards, swapping it with its parent until the heap property is restored.

Time Complexity: O(log n) - Potentially requires traversing up to the root, adjusting node positions.

swapItem(HeapNode child, HeapNode parent)

A helper function that swaps the items of two nodes, typically used in the `decreaseKey` function. It exchanges the items of the child and parent nodes and updates their `HeapItem` pointers accordingly to reflect the swap.

Time Complexity: O(1) - A fixed number of pointer swaps and updates.

delete(HeapItem item)

Deletes a specific item from the heap. It first decreases the item's key to the minimum possible value using `decreaseKey`, and then removes it using `deleteMin`, effectively

removing the node while maintaining heap order.

Time Complexity: O(log n) - Involves both `decreaseKey` and `deleteMin`, each logarithmic in complexity.

meld(BinomialHeap other)

The `meld` function merges the current binomial heap (`this`) with another binomial heap (`other`). This operation involves combining the root lists of the two heaps and linking trees of the same rank. The `meld` function is crucial for operations where multiple heaps need to be unified into a single heap structure.

Steps and Function Calls:
1. **Check for Empty Heaps**: If either heap is empty, the operation is trivial, and the non-empty heap remains as is.
2. **Merge Root Chains**: The function first calls `mergeRootsChains`, which takes the root nodes of both heaps and creates a unified list of root nodes, ordered by rank.
   - This function iterates through the root lists of both heaps and merges them based on rank, preparing the structure for potential linking.
3. **Initialize Pointers for Traversal**: Sets up pointers to traverse the merged root list. The traversal aims to link trees of the same rank into larger trees.
4. **Traverse and Link Same Rank Trees**: As it traverses the root list, when two trees of the same rank are found, `joinTrees` is called to combine them.
   - **joinTrees**: Ensures the tree with the smaller key becomes the parent, incrementing its rank, and attaches the other as its child.
   - During traversal, if three consecutive trees of the same rank are found, `orderTriplet` is called to arrange them correctly by key before linking.
5. **Update Heap Properties**: After linking, the heap's properties (`min`, `last`, `size`) are updated to reflect the newly formed heap.
   - The `min` pointer is updated to point to the smallest key in the new heap, ensuring that future operations like `findMin` function correctly.

The meld function carefully ensures that after merging, no two trees in the heap have the same rank, maintaining the binomial heap properties. By systematically linking trees and resolving rank conflicts, it allows for efficient heap operations across multiple combined structures.

Time Complexity: O(log(n) + log(k)) - The complexity is based on the number of root nodes of the two heaps before the merge, where `n` and `k` are the sizes of the two heaps.

mergeRootsChains(HeapNode node1, HeapNode node2, int numTrees1, int numTrees2)

A helper function that merges two chains of root nodes, ordering them by rank. It traverses both chains, linking nodes into a single ordered list. This function is essential for the meld operation, ensuring that trees are correctly ordered by rank before further operations are performed.

Time Complexity: O(n + m) - Where `n` and `m` are the lengths of the two root chains being merged.

joinTrees(HeapNode minNode, HeapNode maxNode)

Combines two binomial trees of the same rank. The node with the smaller key becomes the parent, and the other becomes its child. This operation increments the parent's rank, maintaining the structural properties of the binomial heap.

Time Complexity: O(1) - Involves a constant number of pointer adjustments to link the nodes.

orderTriplet(HeapNode a, HeapNode b, HeapNode c)

Orders three nodes by their keys and rearranges their links to reflect the correct sequence. This function ensures that node comparisons and link changes maintain the required order for binomial tree properties.

Time Complexity: O(1) - A fixed number of comparisons and pointer rearrangements.

findMaxRank(HeapNode start, int numTrees)

Identifies the node with the maximum rank within a set of trees by iterating over the nodes and comparing their ranks. This function is crucial for operations that need to determine the largest tree in terms of degree.

Time Complexity: O(log n) - Logarithmic relative to the number of trees in the heap.

size()

Returns the total number of elements present in the heap. This function provides an easy way to access the current size property.

Time Complexity: O(1) - Simple access to the size property.

empty()

Checks whether the heap is empty, returning true if there are no elements. This is a basic

check performed before certain operations to ensure validity.

Time Complexity: O(1) - Simple boolean evaluation.

numTrees()

Calculates the number of trees currently present in the heap based on the size. It uses the binary representation of the size to determine the count.

Time Complexity: O(1) - Efficient bit counting operation.

HeapNode Class

Represents a node within the heap. Each `HeapNode` can have children, a sibling, and a parent, forming the structural elements of the binomial trees within the heap. Nodes hold references to their associated `HeapItem`, which contains the key and additional data.

HeapItem Class

Encapsulates the data stored in each node of the heap, including the key and additional information associated with that key. Each `HeapItem` has a back-reference to the `HeapNode` it belongs to, allowing efficient key-based operations.

Section b :

- **ניסוי 1:** נכניס לערמה בינומית ריקה את האיברים בסדר עולה.

מספר סידורי i	זמן ריצה (מילישניות)	מספר החיבורים הכולל	מספר העצים בסיום	סכום דרגות הצמתים שמחקנו
1	17	6555	5	0
2	13	19675	7	0
3	22	59040	8	0
4	44	177134	12	0
5	97	531431	9	0

- **ניסוי 2:** נכניס לערמה בינומית ריקה את האיברים בסדר אקראי, ולאחר מכן נמחק את המינימום פעמים.

מספר סידורי i	זמן ריצה (מילישניות)	מספר החיבורים הכולל	מספר העצים בסיום	סכום דרגות הצמתים שמחקנו
1	23	6698	6	3423
2	12	25148	7	15313
3	23	63976	9	34460
4	32	243421	12	154859
5	125	564322	10	298611

- **ניסוי3:** נכניס לערמה בינומית ריקה את האיברים בסדר יורד, ולאחר מכן נמחק את המינימום עד שנישאר עם איברים.

מספר סידורי i	זמן ריצה (מילישניות)	מספר החיבורים הכולל	מספר העצים בסיום	סכום דרגות הצמתים שמחקנו
1	25	6555	5	6529
2	18	19675	5	19649
3	29	59040	5	59014
4	47	177134	5	177108
5	139	531431	5	531405

Analysis of Experiment 1: ascending Order Insertion

Each element is inserted one after another, starting from 1 and going up to n.

Each new element initially starts as a separate tree of rank 0.

As each element is inserted, it is typically added as a separate tree. However, if a tree of the same rank already exists, the two trees will merge.

In ascending order, elements are added such that each new element fills the next available rank.

The process is similar to incrementing a binary number. Each insertion may

trigger a series of merges that act like binary carries.

For example, adding 111 to a binary number like 011011011 results in 100100100, which involves a series of carries. Similarly, inserting elements from 1 to n involves structured merging without complex reordering.

The time complexity is O(n) because:

Because the elements are added in a strictly increasing order, merges are predictable and efficient.

There are no swaps needed to maintain heap order because the elements are inserted in increasing order. Each merge involves only a simple linking operation therefore as we saw in class, similar to the increment method, the time complexity is o(n).

Analysis of Results for Experiment 2:

we first insert n elements and then perform n/2 `deleteMin()` operations.

1. Inserting n Elements:
   - We add each element into the binomial heap.
   - The `insert` function has a time complexity of O(log n) in the worst case because it might cause several tree merges.

2. Performing n/2 `deleteMin` Operations:
   - Each `deleteMin` removes the root with the smallest key and merges its children back into the heap.
   - This operation also takes O(log n) time because of the need to re-merge trees.

- Each insertion takes about (O(log n)) time. So, for all (n) insertions, the total time is:

-Deleting (n/2) Elements with `deleteMin`:**
- Similarly, each deletion also takes

If we add up the times for all the insertions and deletions, we get:
This is basically saying that our operations together take Θ(nlogn) time.

Analysis of Results for Experiment 3:

When inserting elements in descending order into a binomial heap, each element is initially added as a separate tree. Since the elements are inserted from largest to smallest, the heap grows with minimal need for merging during insertion. Thus, the overall time complexity for inserting all n elements is amortized to O(n).

For the `deleteMin` operations, each deletion removes the root of the smallest tree. In this case, there is no need for merging subtrees, but a new minimum must be identified among the remaining roots. This requires scanning the list of roots, leading to a time complexity of O(logn) for each deletion.

Given that we perform (n-31) deletions, and each involves finding the new minimum, the total time complexity is O(nlogn).

The relationship between the number of links, the sum of deleted ranks, and the number of trees at the end can be expressed as follows:

- **Num of links –num of nodes + num of trees (at the end)= sum of deleted ranks**

  In the 3rd experiment table we can clearly see that, after all deletions 31 nodes remain.