2023

FibonacciHeap

MayarSafadi user name
209826924 id
Mariam khalaila

Mariamk user name
212346076 id
Mariam khalaila

FibonacciHeap:
class fields:
firstRoot: A HeapNode type field points to the first node in the Heap -that is, the first root
lastRoot: A HeapNode type field points to the last node in the Heap -that is, the last root
min: A HeapNode type field indicates the minimum node in the Heap . Which is one from the roots.
size : an int type field, the total number of nodes in the heap.
markNum: an int type field, the number of marked nodes in the heap.
treesNum - an int type field, the number of trees (derivative of the number of roots) in the heap.
links – static variable of type int, counts the total amount of links made so far For all instances of the class, it is initialized to 0.
cuts – static variable of type int, counts the total amount of cuts made so far For all instances of the class,it is initialized to 0.

Department methods:
– public FibonacciHeap()

time complexity O(1)
Defines an empty heap.

– public FibonacciHeap(HeapNode x):

time complexity O(1)
Defines a heap with a single node which is the root.

- public Boolean isEmpty()
time complexity O(1)
A method that returns True if the heap is empty, otherwise it will return False

- public HeapNode insert(int key):

time complexity O(1)
The method receives as input a key and inserts a node with this key into the heap. At first we will create a new node from this key. We will insert this node at the beginning of the heap, insertion in the list is done by updating pointers.
Then we will increase the size and treesNum fields by 1. We will check if the node we inserted is smaller than the minimum, if so, then we will update the min field to point to the node we inserted. At the end we will return as output the node we entered. In total, we did actions in a fixed time.

- public HeapNode findMin()

time complexity O(1)
The method returns the HeapNode node pointed to by the min pointer in the heap.

- public HeapNode size()

time complexity O(1)
The method returns the HeapNode node pointed to by the size pointer in the heap.

- public void meld (FibonacciHeap heap2)

time complexity O(1)
The method connects the two heaps, that is, it updates the last element in the heap
The first to point to the first member in the second heap, and updates the tail field of the heap
the first to point to the last member in the second heap. All in all, it is about updating pointers, therefore O(1)
.

– public delete(HeapNode x)

Worst case Time complexity o(n)
Amortized cost o(log n)
The deletion operation of each node in a heap is a two-step process:

1. First, the key of the node to be deleted is set to minus infinity by calling the decreaseKey operation with a value of 0. This ensures that the node to be deleted will become the smallest value in the heap.
2. Next, the node with the smallest key is identified as the minimum and the deleteMin operation is performed on it. This operation removes the minimum node from the heap and reorganizes the remaining elements to maintain the heap property.

In the worst case, this results in a linear runtime of O(n), where n is the number of nodes in the heap. This is because all nodes in the heap need to be checked to find the minimum node and all nodes need to be rearranged to maintain the heap property after deletion. However, the amortized runtime for the deleteMin operation is O(log n) and the amortized runtime for the decreaseKey operation is also O(log n) because it is performed using a specific algorithm called as Fibonacci Heap. In this algorithm, the deletion process is done by cutting the node from the tree, and then using a series of merge operations to reorganize the remaining elements and maintain the heap property. These merge operations take O(1) time on average, but O(log n) time in the worst case, resulting in the amortized runtime

-Public decreaseKey(HeapNode x, int delta)

The method decreaseKey(HeapNode x, int delta) is used to decrease the key of a given node x in the Fibonacci heap by a certain value delta. When the key of a node is decreased, it may violate the heap property, so the method performs any necessary cuts to restore the heap property. The worst-case time complexity of this method is O(n), as it may require a series of cut operations, each with a linear time complexity of O(1), with a total length of n, depending on the depth of the tree in the Fibonacci heap. This is because when the key of a node is decreased, if the new key is smaller than its parent's key, the cascadingCut operation will be performed. This operation is used to cut the node from its parent and move it to the root list of the heap.

The cascading cut operation may be performed multiple times, each time moving the node up one level in the tree, until the new key is greater than or equal to its parent's key or the node becomes a root.
The cascading cut operation has worst case time complexity of O(n) because the depth of the tree in Fibonacci heap could be linear in n, this means that in some cases cascading cut could be performed n times before the heap property is restored. However, the amortized time complexity of this method is O(1) as in most cases the operation is performed in a fixed time. This is because when the expensive cases occur that have O(n) time complexity, the structure of the heap and the number of marked nodes are changed, which in turn allows for many operations of fixed time to be performed before another expensive operation is needed. This concept is known as Amortized analysis and it's used to analyze the average time complexity of the method over a sequence of operations.

-public DeleteMin()
Worst case time complexity: o(n)
Amortized cost: o(log n)
The DeleteMin() function is used to delete the minimum node from a heap. The complexity of the function is O(n) in the worst case and O(log n) in the amortized case.
The function first checks if the heap is empty. If it is, there is nothing to delete. If there is only one node, it is deleted and the heap becomes empty. If the heap is not empty, the function divides into four cases:

1. If the minimal node has a degree of zero, it is simply deleted.
2. If the minimal node has a degree greater than zero, the function checks if there is a single root or multiple roots. In this case, the node is deleted and the parent field of its children is set to null. This operation is complicated (log n) because it is a Fibonacci heap.
3. The sons of the deleted node are added to the prev and next of the minimum node, while updating the voting.

The function consolidating is activated. This function has a complexity of (n(O) in the worst case and logarithmic n in amortized.
Finally, the size field is decremented by 1. In the worst case, the runtime of the DeleteMin operation is O(n), and in the amortized case, the runtime is O(log n). When looking at a series of actions and not just one action, it can be accepted that the DeleteMin operation runs logarithmically.

– private void consolidating()
Worst case time complexity: o(n)
Amortized cost: o(log n)

The consolidating method is a Public helper method that is used to perform unions between trees of the same degree in the heap. The method begins by calling the heapToArray() method, which returns an array of HeapNode type with a time complexity of O(n).
The heapToArray() method is used to transform the current heap into an array of buckets, where each bucket contains nodes of the same degree. The Consolidate method then saves the output of the heapToArray() method in an array called f.

The arrayToHeap() method is then called with the buckets array built previously by heapToArray(). The arrayToHeap() method takes the array of buckets and transforms it back into the heap structure, ensuring that each rank has at most one representative tree.
The arrayToHeap() method has a time complexity of O(log n).
The consolidating method has a worst-case time complexity of O(n) as it combines both heapToArray() and arrayToHeap() time complexity. the amortized time of consolidating is O(log n), because the amortized time of O(log n) is arrayToHeap() and heapToArray().

– private void heapToArray()
Worst case time complexity: o(n)
Amortized cost: o(log n)
The method, called heapToArray(), starts by generating an array of HeapNode type with a size of $1\varphi\ n$ + log, where each position in the array represents a specific degree that a Fibonacci tree can have. The method then iterates through all the roots in the heap and places them in the corresponding cell in the array, based on the degree of the root. If there are two trees in the same cell, the method links them and moves them to the next cell that represents a degree greater by 1. If this cell is already full, the method will repeat this process. The worst-case scenario would be a heap with n trees that are all roots, in which case the time complexity would be O(n).

$$\sum_{i=0}^{logn} \frac{n}{2^i} \leq n * \sum_{i=0}^{logn} \frac{1}{2^i} \leq\ n * \sum_{i=0}^{\infty} \frac{1}{2^i} = O(n)$$

However, since the number of trees in the heap is not always linear, the amortized time complexity is O(log n). The arrayToHeap() method performs the inverse operation, transforming the array back into a heap. The time complexity of this operation is O(log n). The arrayToHeap() method is also amortized O(log n).

– private void arrayToHeap()
Worst case time complexity: o(log(n))
The method receives as input a list of binomial trees (referred to as the binomial trees) and unites them into a single Fibonacci heap. The list is guaranteed to have no two trees with the same rank, which allows for efficient deletion operations (i.e. deleteMin) that take linear time in n. The final heap has a size bounded by $1\varphi\ n$ + log(n), and the method goes through the list of trees and connects them to form the final heap. The complexity of connecting any two trees is constant, so the total time complexity of this operation is O(log n).\

-private HeapNode Link()
Worst case time complexity: o(1)
 The method accepts as input two trees with the same rank and returns a new tree that contains both of the input trees. The rank of the returned tree is 1 + the rank of the input trees. The method also increases a variable called totalLinks by 1. The method updates the pointers in the fields of the two HeapNodes, which is done in a fixed time. Therefore, the time complexity of this method is O(1).

- public HeapNode findMin()

time complexity: o(1)

The method returns the HeapNode that the min pointer in the heap is pointing to. The time complexity for this method is O(1) as it just returns the value of a pointer.

-private void cascadingCut(HeapNode x, HeapNode parent)

Worst case time complexity: o(n)

Amortized cost: o(log n)

The recursive method described performs cuts on a Fibonacci heap after a decreaseKey operation. It starts by cutting the node x from its parent using the cut method, which has a time complexity of O(1). If the parent is not marked, it is marked. If the parent is marked, the method is called recursively on the parent of the current node. The overall time complexity of the method is O(n) in the worst case, but the amortized time is O(1) because most cases only require a single cut and the expensive cases where a chain of cuts is performed are rare.

- public void cut(HeapNode x, HeapNode parent):

A method whose purpose is to make a cut between a node and its parent, while dividing it into several cases, at the end we add this node to the beginning of our heap. This is done by updating pointers, which is O(1)

– public int[] countersRep()

O(n) complexity

The method returns an array so that the i index stores how many trees are in the heap whose rank is i.
The method initializes an array, in which there will be the number of trees of each rank, of size 1φ n + log because, as we saw in class, this is the maximum size that a binomial tree can have in a heap of size n. Then we will go through all the trees in the heap and add 1 instead of the rank in the array. In total if we have n trees in the worst case then complications o(n).
We did not refer to the amortized time complexity here, but because here it will not necessarily happen as if we take a series of operations that is counterRep each time for a heap that contains n binomial trees each of degree zero then we will always get the same complexity which is o(n).

- public int potential()

O(1) complexity

A method that returns the number of potentials in the heap, as we defined it in the class to be treesnum+ 2*marksnum

- public static int totalLinks()
O(1) complexity, A method that returns the number of links made so far in the heap.
- public static int totalCuts()
O(1) complexity, A method that returns the number of cuts made so far in the heap.
- public int getNumOfTrees()
.O(1) complexity, A method that returns the number of trees in the heap.

– public static int[] kMin(FibonacciHeap H, int k)

Time Complexity :$O(k * \deg(H))$
The given method is for finding the k smallest keys in a Fibonacci heap. It first checks if the heap is empty, and if so, it returns an empty array. The method uses a pointer called "copy" to keep track of the nodes in the original heap and a new empty Fibonacci heap is created. The method starts with the root of the original heap, creates a copy of it, and inserts it into the new heap. The method then repeatedly finds the minimum node in the new heap, inserts its key into the output array, and adds its children to the new heap until k iterations have been completed. The time complexity of this method is O(k*deg(H)), where H is the original heap and deg(H) represents the degree of the heap.

$$1 + degH + \sum_{i=2}^{k-1}(degH + lig(i * degH) = 1 + degH + \sum_{i=2}^{k-1}(degH + log(degH) + \log(i)) \leq \sum_{i=1}^{k}\left(2degH + lig(i)\right) \leq 2k * degH + klogk = O(k(logk + degH) = O(k * degH)$$

The HeapNode class:
Department fields:
• key - an int type field, contains the key value of the node.
• rank - an int type field, contains the rank of the root - the number of its children.
• mark – boolean type field, indicates whether the node is marked or not.
• child – HeapNode type field, points to the leftmost child of the node.
• next – HeapNode type field, points to the next node (that is, connected to it on the right).
• prev – HeapNode type field, points to the previous node (that is, connected to it on the left).
• parent - HeapNode type field, indicates the parent of the node.
• copy- HeapNode type field, this pointer will only be useful in the kmin function, for the rest of the operations you can ignore it, it will always point to null

חלק ניסויי/תיאורטי

Question 1

A.An analysis of the running time of the above series of operations as a function of m is given below:

- Initially, we performed m insert operations, which as previously described, have a constant running time of O(1).
- Next, we performed a single deleteMin operation, which has a running time of O(m log m) due to the complexities of merging the remaining binomial trees into a single tree of size m.
- After that, we performed(log m -1) decreaseKey operations. Each of these operations have a constant time complexity, however the overall time complexity increases as the number of marked nodes in the heap increases. The most expensive operation is the last decreaseKey operation, which is performed on the node in the array at the m-1 place and has a complexity of O(log m) due to the depth of the tree.
- Therefore, the total complexity of the series of operations is $m + m + logm + log$m=O(m + log m + log m) = O(m)

B.

m	Run-Time (ms)	totalLinks	totalCuts	Potential
2^5	0.8731	511	17	18
2^10	1.7458	1023	19	20
2^15	3.3561	2047	21	22
2^20	6.3334	4095	23	24

C. In this series of actions, there is an upper limit on the number of Link (merge) and Cut (decreaseKey) actions that can be carried out. As seen in class, a Link operation is performed only when a deleteMin operation is carried out. In this case, there is only a single deleteMin operation, which determines the number of Links. The number of Links, denoted by m, is a power of 2 at the beginning. After the deleteMin operation, we are left with exactly m trees of degree zero, which means that the Link operations will be carried out until the heap is complete. The number of Link operations can be calculated as

$$totalLinks = \sum_{i=0}^{logm} \frac{m}{2^i} \leq \sum_{i=0}^{inf} \frac{m}{2^i} = O(m)$$

On the other hand, the number of Cut (decreaseKey) operations can be explained as follows: all the decreaseKey operations performed in the beginning, which are log(m)-1, cause only one cut each at a fixed time. These operations prepare the way for the expensive decreaseKey action, which causes a series of cuts from the leaf with the greatest depth to the root +1. The total number of cuts for this series is 2log(m)<=o(log(m)) which again agrees with the results obtained in the table.

The potential of the heap in this case is 3log(m) + 1. This is because the log(m) Cut operations resulted in 1+log(m) trees, and log(m) marked nodes (the parents of those that were cut).

D.

In the case of d, the total number of cuts made during the Decrease key operation is 0. This is due to the fact that the operation is performed by traversing the heap from the smallest node (root) to the largest node (leaf) and decreasing the parent and child nodes by the same amount, thereby maintaining the binomial tree structure.

This ensures that no node violates the class reserve. The potential of the heap in this case is 2log(m)+1, as the number of marked nodes is log(m) and the number of trees remains 1. The total number of links in the heap is m-1, which is related to the Delete-min/Delete operation and remains unchanged.

E.

In the case where row 2 is deleted, the resulting structure is similar to a linked list, consisting of M trees of degree 0(**totalCuts = 0**) . As a result, during each Decrease key operation, no cuts will be made. The potential of the heap in this case is m+1, as the number of trees is equal to the number of members, with each node inserted as a root. The total number of links in the heap is 0, as no Delete-min/Delete operation was performed.

F.

In this case, after the original operations, a chain of leaves is marked as a result of cutting their children. When the node m-2 is decreased, it is cut and a chain of cuts is made towards the root, resulting in a total of log(m) - 2 cuts. Along with the log(m) cuts made before, the total number of cuts made is 2log(m)-1.

The potential of the heap in this case is 2log(m), as the number of trees is 2log(m) and there are no marked nodes as all previously marked nodes have become roots. The total number of links in the heap is m-1, which is related to the Delete-min operation and remains unchanged.

The most expensive Decrease key operation will be the last one, performed on the node m-2, as it traverses the entire route to the root, which has a length of log(m)-1.

Question 2

m	Run-Time (ms)	totalLinks	totalCuts	Potential
728	2721200	720	0	6
6,560	3350300	7270	0	6
59048	30155000	66307	0	9
531440	190415000	597734	0	10
4782968	1761313200	5380686	0	14

B. An asymptotic analysis of the running time of the above series of operations is as follows: Initially, m Insert operations are performed, each with a constant time complexity of O(1), resulting in a total time complexity of O(m). Following this, a series of Delete-min operations are performed. The first Delete-min operation has the greatest complexity, with a time complexity of O(m). After the first Delete-min operation, the size of the tree is reduced to 1-m, which corresponds to the binary representation of the number, with each place of unity representing a binomial tree of that degree. The tree after this operation has a time complexity of O(log m). Subsequent Delete-min operations are less complex, as they are executed on a smaller tree, resulting in a time complexity of O(log m) for each operation. Therefore, the overall time complexity of all Delete-min operations is O(m log m) due to:

$$runTimeForSequence2 = m + m + \sum_{i=2}^{\frac{m}{2}} \log(m-i) \leq 2m + \sum_{i=1}^{\frac{m}{2}} \log(m) = O(mlogm)$$

C. There is an upper limit on the amount of linking and cutting operations that are performed during this series. The number of cutting operations that are performed during the series is zero, as no decreaseKey or deleteMin operations are performed. The number of links that are performed during the series is upper bounded by O(mlog(m)).

$$LinksNumber \leq m + \sum_{i=2}^{\frac{m}{2}} 2\log(m-i) \leq m + \sum_{i=2}^{\frac{m}{2}} \log(m) = O(m * logm)$$

This is because the first deleteMin operation results in at most m links, and subsequent deleteMin operations result in at most (1-m)(2log(m)) links in the worst case, where the binomial trees are combined into one big tree.

However, it should be noted that by tests and larger runs on m, it was found that the number of links is upper bounded by 2m. A specific reason for this discrepancy could not be determined, and therefore the upper bound of O(mlog(m)) is provided.

A bound is placed on the potential, and it appears to align with the results obtained in the table. The potential function is defined as the number of trees plus 2 times the number of marked nodes. As no decreaseKey operations are performed in this series, the number of marked nodes remains zero throughout the run.

The number of trees is upper bounded by O(2/m*log(m)), as after m/2 deleteMin operations, a Fibonacci tree of size m/2 is obtained. The number of trees is equal to the number of units in the binary representation of the number, which requires O(log(m)) bits to display. This aligns with the results obtained in the table, where in 3 cases the potential is upper bounded by O(log(m)).