Data Structures

Practical Assignment #2
DHeap

מגישים: עומר ברק (305555070) omerbarak2, חגי בן יהודה (305237000) hagaib1

Documentation

Nested Classes

Modifier and Type	Class and Description
(package private) static class	`DHeap.SortResult` Auxiliary class used to hold the result of a DHeapSortMeasure operation.

Fields

Modifier and Type	Field and Description
private DHeap_Item[]	`array` The actually members of the heap.
(package private) int	`compare_count` Used for testing comparisons inside the heap, for measurements.
private int	`d` The number of children each node in the d-heap structure has.
private int	`max_size` Holds the maximum number of elements possible for the heap to hold.
private int	`size` Holds the number of elements currently in the heap.

Constructor

DHeap

DHeap(int m_d, int m_size)
Constructor for the DHeap class.

Runtime: O(max_size)

Parameters:

m_d - The number of children for each node in the heap

m_size - The maximum size the heap might get

Methods

arrayToHeap

public void arrayToHeap(DHeap_Item[] array1)
The function builds a new heap from the given array. Previous data of the heap will be erased.

Precondition: array1.length <= max_size
Postcondition: isHeap(), size = array.length()
Runtime: O(n)

Parameters:

array1 - An array of all the items to insert into the heap

arrayToHeap

public void arrayToHeap(int[] numbers)
Make a heap from an array of numbers. Erases the current contents of the heap.

Precondition: numbers.length <= max_size
Postcondition: isHeap(), size = array.length()
Runtime: O(n)

Parameters:

numbers - The numbers to insert into the heap

arrayToHeap

public void arrayToHeap(List<Integer> numbers)
Make a heap from a list of Integers. Erases the current contents of the heap.

Precondition: numbers.size() <= max_size
Postcondition: isHeap(), size = array.length()
Runtime: O(n)

Parameters:

numbers - The numbers to insert into the heap

checkHeap

void checkHeap()
An internal method, used to assert the internal array satisfies the properties of a D-ary heap, and a few more internal tests.

Runtime: O(n)

child

public int child(int i, int k)
Computes the index of the k-th child of node i in a complete D-ary tree stored in an array. 1 <= k <= d. Note that indices of arrays in Java start from 0.

Precondition: i >= 0
Runtime: O(1)

Parameters:

i - The node who's child is requested

k - The requested child's index in relations to the parent

Returns:

The index of the requested child

childrenCount

private int childrenCount(int i)
Returns the number of children a specific node has.

Precondition: i >= 0
Runtime: O(1)

Parameters:

i - The node we want to get the number of children it has

Returns:

The number of children the given node has

Decrease_Key

public void Decrease_Key(DHeap_Item item, int delta)
Decrease the key of the given item by the given amount.

Precondition: item.pos < size, item != null, isHeap()
Postcondition: isHeap()
Runtime: O(log(n)/log(d))

Parameters:

item - The item who's key we want to decrease

delta - The value by which we want to decrease the item's key

Delete_Min

public void Delete_Min()
Deletes the smallest element from the heap.

Precondition: size > 0, isHeap()
Postcondition: isHeap()
Runtime: O(d/log(d)*log(n))

Delete

public void Delete(DHeap_Item item)
Deletes the given item from the heap.

Precondition: item.pos < size, item != null, isHeap()
Postcondition: isHeap()
Runtime: O(d/log(d)*log(n))

Parameters:

item - The item we want to delete from the heap

DHeapSort

public static int[] DHeapSort(int[] array)
Return a sorted array containing the same integers in the input array (done using HeapSort with a d-ary heap where d=2).

Runtime: O(n*log(n)), n = array.length

Parameters:

array - The array to sort

Returns:

A new array with the same values as the received one but sorted

DHeapSort

static int[] DHeapSort(int[] array, int d)
Return a sorted array containing the same integers in the input array.

Runtime: O(d/log(d)*n*log(n)), n = array.length

Parameters:

array - The array to sort

d - The arity of the d-ary heap to use

Returns:

A new array with the same values as the received one but sorted

DHeapSortMeasure

static DHeap.SortResult DHeapSortMeasure(int[] array, int d)
Return a sorted array containing the same integers in the input array, and also count the amount of comparisons made to detect the order.

Runtime: O(d/log(d)*n*log(n)), n = array.length

Parameters:

array - The array to sort

d - The arity of the d-ary heap to use

Returns:

A new array with the same values as the received one but sorted

Get_Min

public DHeap_Item Get_Min()
Returns the element with the lowest key value in the heap.

Precondition: heap-size > 0, isHeap(), size > 0
Postcondition: isHeap()
Runtime: O(1)

Returns:

The element with the lowest key value in the heap

getItems

List<DHeap_Item> getItems()
Returns a list of all the items in the heap.

Runtime: O(n)

Returns:

```
All the items in the heap, as a list
```

getSize

public int getSize()
Returns the size of the heap.

Runtime: O(1)

Returns:

```
The current number of elements in the heap
```

hasChildren

private boolean hasChildren(int i)
Says if a certain node has any children.

Precondition: i >= 0
Runtime: O(1)

Parameters:

```
i - The node to check
```

Returns:

```
true iff the node has any children
```

heapify

private void heapify()
Makes sure the internal array is structured like a heap, moving elements where necessary.

Runtime: O(n)

heapifyDown

private void heapifyDown(int i)
Makes sure the elements in the underlying array maintain the heap property, under a specific node in the heap, swapping elements if needed.

Runtime: O(d/log(d)*log(n))

Parameters:

i - The node from which to start the verification

heapifyUp

private void heapifyUp(int i)
Makes sure the elements in the underlying array maintain the heap property, under a specific node in the heap, swapping elements if needed.

Runtime: O(log(n)/log(d))

Parameters:

i - The heap index to start the process from

Insert

public void Insert(DHeap_Item item)
Inserts an item into the heap.

Precondition: item != null, isHeap(), size < max_size
Postcondition: isHeap()
Runtime: O(log(n)/log(d))

Parameters:

item - The heap item to insert

Insert

void Insert(int number)
Inserts a new item from a number into the heap.

Precondition: item != null, isHeap(), size < max_size
Postcondition: isHeap()
Runtime: O(log(n)/log(d))

Parameters:

number - The number to insert to the heap (as a heap item)

isHeap

public boolean isHeap()
public boolean isHeap() The function returns true if and only if the D-ary tree rooted at array[0] satisfies the heap property or size == 0.

Runtime: O(n)

Returns:

```
If the array satisfies the properties of a D-ary tree
```

lastChildIndex

private int lastChildIndex(int i)
Returns the index of the last child of the checked node.

Precondition: i >= 0
Precondition: hasChildren(i) == true
Runtime: O(1)

Parameters:

```
i - The node to check
```

Returns:

```
The index of the last child of the tested node
```

minChildIndex

private int minChildIndex(int i)
Returns the index of the child with the lowest key from the given node's children.

Precondition: hasChildren(i) == true
Runtime: O(d)

Parameters:

```
i - The node who's child we want
```

Returns:

```
The index of the child of the given node with the lowest key
```

newItem

private DHeap_Item newItem(int number)
Creates a new DHeap_Item from a number, to insert it into the heap.

Runtime: O(1)

Parameters:

number - The number to wrap

Returns:

The DHeap_Item wrapping the number

parent

public int parent(int i)
Computes the index of the parent for the heap element at the given index.

Precondition: i > 0
Runtime: O(1)

Parameters:

i - The index of the element who's parent is requested

Returns:

The index of the parent for the given element

popMin

private DHeap_Item popMin()
Returns the element with the lowest key value in the heap, and deletes it from the heap.

Precondition: heap-size > 0, isHeap(), size > 0
Postcondition: isHeap()
Runtime: O(d/log(d)*log(n))

Returns:

The element with the lowest key value in the heap

printTree

void printTree()
Prints the heap as a tree. Used internally for testing.

Runtime: O(n)

printTree

private void printTree(int i, String prefix, boolean isTail)
The internal method that prints a part of the heap as a subtree.

Note: adapted from http://stackoverflow.com/a/8948691
Runtime: O(n), n = sub-heap size.

Parameters:

i - `The index from which to start printing`

prefix - `The prefix used in order to align all all the prints`

isTail - `Says weather the current node is it's father's last child`

setItem

private void setItem(int i, DHeap_Item item)
Set the specified array slot to the given item, and update the item's position.

Runtime: O(1)

Parameters:

i - `The index at which to set the item`

item - `The item to put in the internal array`

swapItems

private void swapItems(int i, int j)
Swaps two items' position in the internal representation if the Heap.

Runtime: O(1)

Parameters:

i - `The index of the first item`

j - `The index of the second item`

Measurements

Part 1

Average comparisons for the arrayToHeap+DHeapSort operations:

m\d	2	3	4
1000	16859.8	16431.4	17608.8
10000	235320.4	226530.5	242759.0
100000	3018566.7	2896628.0	3076698.0

Part 2

Average comparisons for the Decrease-Key tests:

x\d	2	3	4
1	99999.0	99999.0	99999.0
100	153092.6	130732.9	122941.5
1000	303370.5	212932.7	181124.3

Analysis

Part I

Basis

n - size of the heap, d - heap arity, h - height of the heap.
$d \geq 2$ is assumed. For $d = 1$ the heap is degenerated, and the operations will take $\Theta(n^2)$ at the worst case.
The amount of items in depth k is at most d^k. Therefore:

$$\begin{aligned} \sum_{k=1}^{h-1} d^k &\leq n \Rightarrow \frac{d^h - 1}{d-1} \leq n \Rightarrow \frac{d^h - 1}{d} \leq n \Rightarrow d^h \leq nd + 1 \Rightarrow \\ h &\leq \log_d(nd+1) \leq \log_d(2nd) = \log_d(2) + \log_d(n) + 1 \leq \log_d(n) + 2 \end{aligned}$$

Analysis of heapifyDown

It takes exactly $d-1$ comparisons to find the minimum child of a node, and 1 comparison to determine if the current node should be swapped with it's minimum child. Therefore d comparisons are done at each iteration, and there are at most k iterations for a node at height k. Therefore for a call on a node of height k, there are at most dk comparisons.

Analysis of arrayToHeap (implemented with heapify)

Let x be the number of leaves in the heap. Then the number of nodes of height 1 is $\left\lceil \frac{x}{d} \right\rceil \leq \frac{x}{d} + 1$. The number of nodes of height 2 is $\left\lceil \frac{\left\lceil \frac{x}{d} \right\rceil}{d} \right\rceil \leq \frac{\frac{x}{d}+1}{d} + 1 = \frac{x}{d^2} + \frac{1}{d} + 1$. Similarly the number of nodes of height 3 is not greater then $\frac{x}{d^3} + \frac{1}{d^2} + \frac{1}{d} + 1$. Also, obviously $x \leq n$. It's easy to see that the number of nodes of height k is not greater then $\frac{n}{d^k} + 2$. From this, and from the analysis of heapifyDown, , the amount of comparisons done by arrayToHeap is bounded by:

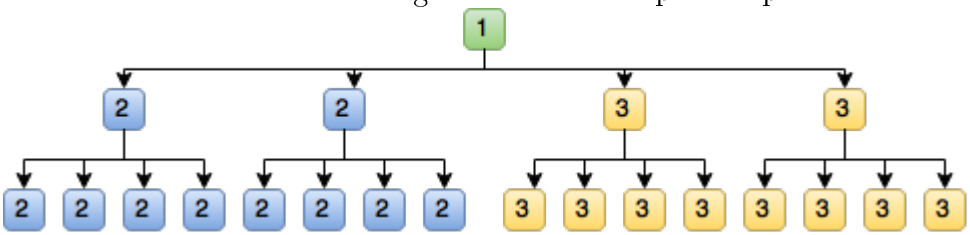
$$\begin{aligned} \mathit{Comparisons} &\leq \sum_{k=1}^{h} dk \left(\frac{n}{d^k} + 2 \right) \overset{*}{\leq} \sum_{k=1}^{h} dk \cdot 2\frac{n}{d^k} = 2nd \sum_{k=1}^{h} \frac{k}{d^k} \leq 2nd \sum_{k=0}^{\infty} \frac{k}{d^k} \\ &= 2nd \frac{d}{(d-1)^2} = 2n \left(\frac{d}{(d-1)} \right)^2 \leq 8n = O(n) \end{aligned}$$

Where the second transition is true for large enough n.
Also, $\Omega(n)$ is an obvious lower bound for the comparisons made by heapify, as at least $n-1$ comparisons are required to find a minimum item in an unordered list.
Therefore arrayToHeap executes $\Theta(n)$ comparisons.

Analysis of DHeapSort (n delete-min calls)

As each delete-min simply calls heapifyDown on the root, then from the analysis of heapifyDown, the number of comparisons is $O(dh) = O(d(\log_d(n)+2)) = O(d\log_d(n)) = O(\frac{d}{\log(d)}\log(n))$. Then the series of n calls to delete-min are executed in $O(\frac{d}{\log(d)}n\log(n))$.
This bound is tight at the following example shows:
For simplicity, we'll assume the heap's last level is a full, and that d is even (the idea can be adapted to other cases).
We are given a heap where the root has key 1, and of it's d children, the left half have key 2, and the right half have the key 3. For every node that has key 2, all it's descendants have key 2, and for every node that has key 3, all it's descendants have key 3. See Figure 1 for example.
Given this scenario, the first half of the n delete-min calls will make most of the 3 nodes travel all the way from the root to the deepest level (to the left side, instead of the 2 nodes). Therefore, for $\Theta(n)$ nodes, $\Theta(d\log_d(n))$ comparisons will be made. Therefore the bound above is tight, $\Theta(\frac{d}{\log(d)}n\log(n))$ comparisons will be made at the worst case. The entire operation will therefore make $\Theta(n)+\ \Theta(\frac{d}{\log(d)}n\log(n)) = \Theta(\frac{d}{\log(d)}n\log(n))$ comparisons at the worst case.

Figure 1: W.C. Heap Example



Measurements Explanation

Assuming the average case has a similar run-time complexity, the measurements can be explained:
As seen in the measurements, the amount of comparisons is linearithmic ($n\log(n)$ relation) in n, as expected.
The amount of comparisons appears to be lowest when $d = 3$. This can be explained by examining the term $\frac{d}{\log(d)}$. Looking at the function $f(x) = \frac{x}{\log(x)}$, we get $f'(x) = \frac{\log(x)-1}{\log^2(x)}$. Therefore $f'(e) = 0$, and it is clear that $x = e$ is a global minimum for $x \geq 1$. So in theory, an e-heap would have the lowest amount of comparisons. In practice 3 is the closest integer to e, so it yields the lowest amount of comparisons achievable using DHeapSort.

Part II

Analysis of heapifyUp

When heapifyUp is called on an item at depth k, then at most k comparisons and swaps are performed, as the item is only compared to it's parent.

Analysis of n Insert, Decrease-Key calls

The maximum depth of an item in the heap is $h \leq \log_d(n) + 2 = O(\log_d(n))$, and each Insert performs 1 heapifyUp, therefore the amount of comparisons in n calls to Insert is bounded by $O(n \log_d(n)) = O(\frac{n \log(n)}{\log(d)})$

By the exact same reasoning, n calls to Decrease-Key are bounded by $O(n \log_d(n)) = O(\frac{n \log(n)}{\log(d)})$

The bounds are tight, as the following input shows:

n items are inserted with the keys: $2n+1, 2n, 2n-1, \ldots, n+1$, and the Decrease-Key calls are done with $delta = n+1$, in the order of insertion.

At the insertion phase, each item inserted is the minimum (at that stage), and therefore must be heapified-up from the bottom of the heap to the root. At the decrease phase, each item items changes from the maximum to the minimum, and must be heapified-up from the bottom to the root.

In a heap of size n, more than $\frac{n}{2} = \Theta(n)$ items are leafs. Therefore $\Theta(n)$ of the operations, in both phases, will indeed cost $\Theta(n \log_d(n))$ comparisons, showing that the bound is tight.

Therefore the amount of comparisons of in the overall operation is at the worst case $\Theta(\frac{n \log(n)}{\log(d)})$.

Note: if $d \geq n$ then the worst case is $\Theta(n)$, because $\Omega(n)$ is an obvious lower bound.

Note: The TA later changed the requirement to analyze the worst case of a single Decrease-Key call. It's obvious that the W.C. for a single call is $\Theta(\log_d(n)) = O(\frac{\log(n)}{\log(d)})$. We left the full analysis as it didn't seem right to delete it.

Measurements Explanation

Assuming the average case has a similar run-time complexity, the measurements show that for $x = 100$, $x = 1000$, the number of comparisons indeed shrinks as d grows, as expected.

The lower-bound for the best case is $n - 1$ comparisons, in the case where no item changes its position (-1 because the root item never needs to be heapified-up). That bound is indeed achieved for the case where $x = 1$, because if a node's key is higher than it's parent, then decreasing it by 1 will not make them swap. Because the heap is built by insertions and not via arrayToHeap, then if a node and it's parent have the

same key, then the parent must have been inserted earlier (because nodes are inserted to the bottom of the heap and are heapified-up). Therefore the parent's key will be decreased earlier, so there isn't a possibility of them swapping.