Data Structures – Practical Assignment 2

Oren Gal – [redacted]

Chen Barnoy – [redacted]

Class documentation:

DHeap

Fields:

Type	Name	Description
int	**size**	The current heap size.
int	**max_size**	The maximum possible size of the heap.
int	**d**	The number of children of each node.
DHeap_item[]	**array**	Array that represents the heap. Each item in the array is a **DHeap_item.**

Skeleton methods:

Type	Signature	Description	Time complexity
int	getSize()	Returns the heap's size.	$O(1)$ Because the field is updated during the other methods without changing their optimal time complexity.
int	arrayToHeap(**DHeap_item[]** array1)	The function builds a new heap from the given array. Previous data of the heap should be erased. Returns the number of comparisons made between items.	$O(n)$

Boolean	isHeap()	The function returns true if and only if the D-ary tree rooted at array[0] satisfies the heap property or has size == 0.	$O(n \cdot d)$
static int	parent(**int** i, **int** d)	Compute the index of the parent of node i, in a complete d-ary tree stored in an array.	$O(1)$
static int	child(**int** i, **int** k, **int** d)	Compute the index of the k-th child of node i, in a complete d-ary tree stored in an array.	$O(1)$
int	Insert(**DHeap_item** item)	Inserts the given item to the heap. Returns the number of comparisons made during the insertion.	$O(logn)$
int	Delete_Min()	Deletes the minimum item in the heap. Returns the number of comparisons made during the deletion.	$O(d \cdot logn)$
DHeap_item	Get_Min()	Returns the minimum item in the heap.	$O(1)$
int	Decrease_Key(**DHeap_item** item, **int** delta)	Decerases the key of the given item by delta. Returns number of comparisons made as a result of the decrease.	$O(logn)$
int	Delete(**DHeap_item** item)	Deletes the given item from the heap. Returns number of comparisons during the deletion.	$O(d \cdot logn)$
static int	DHeapSort(**int[]** array1, **int** d)	Sort the input array using heap-sort (build a heap, and perform n times: get-min, del-min). Sorting should be done using the DHeap, name of the items is irrelevant. Returns the number of comparisons performed.	$O(d \cdot n \cdot logn)$

Helper methods:

Type	Signature	Description	Time complexity
int	heapifyDown(**DHeap_item** item)	Compares iteratively between the item and its children, and switches its position with the minimal child, if they break the heap rule. Returns the number of comparisons performed.	$O(d \cdot logn)$
int	heapifyUp(**DHeap_item** item)	Compares iteratively between the item and its parent, and switches its position with the parent, if they break the heap rule.	$O(logn)$

		Returns the number of comparisons performed.	
DHeap_item	minChild(**DHeap_item** item)	Returns the item's child with the minimal key.	$O(d)$
int	numOfChildren(**DHeap_item** item)	Returns the number of item's children.	$O(1)$
void	switchItems(**DHeap_Item** item, **DHeap_Item** parent)	Switches between item and its parent.	$O(1)$
Boolean	parentIsLegal (**DHeap_item** item)	Compares the item to its parent and returns true if and only if the heap property is maintained.	$O(1)$
int	hasLegalChildren(**DHeap_item** item)	Returns -1 if and only if all children of item follow the heap rule, else, returns the position of the child that breaks the heap rule.	$O(d)$
int	getLeavesStartIndex()	Calculates the index i for which the indices {i ,..., size-1} represent the set of all leaves in the heap.	$O(1)$

DHeap_Item

Fields:

Type	Name	Description
String	**name**	The item's name.
int	**key**	The item's key.
int	**pos**	The item's position in the array that represents the heap.

Methods: Time complexity of all methods is $O(1)$

Type	Signature	Description
int	getKey()	Returns the item's key.
int	getPos()	Returns the item's pos.
String	getName()	Returns the item's name.
void	setPos(**int** pos1)	Sets pos to position.
void	setKey(**int** key1)	Sets key to key1.

Benchmark measurements – part 1 (arrayToHeap+DHeapSort)

No. elements	d	Avg. no. comparisons	$d*n*\log_d n$
1,000	2	16,137	19,932
1,000	3	15,800	18,863
1,000	4	17,028	19,932
10,000	2	228,331	265,754
10,000	3	220,364	251,508
10,000	4	236,843	265,754
100,000	2	2,947,778	3,321,928
100,000	3	2,834,527	3,143,855
100,000	4	3,017,290	3,321,928

הכנסה של מערך לערימה עולה $O(n)$, heap-sort עולה $O(d \cdot n \log_d n)$, ולכן הסיבוכיות האסימפטומטית של שתי הפעולות האלו תהיה הגדולה מבניהן - $O(d \cdot n \log_d n)$. נשים לב שהחסם $d \cdot n \log_d n$ מתאר בצורה טובה את מספר ההשוואות.

נראה דוגמה שבה יש $d \cdot n \log_d n$ השוואות ובכך נוכיח שהחסם הדוק (חסם תטא): DHeap-sort מבצע n איטרציות של delete-min . נתבונן בערימה שבה כל המפתחות שונים ומסודרים בכל רמה לפי הגודל משמאל – הכי קטן, לימין – הכי גדול, ונניח בה"כ ש-heapify-down משווה כל צומת לבניו לפי- הבן הימני

מושווה ראשון ואז הולכים שמאלה אחד אחד לפי הסדר. במקרה זה, בכל איטרציה נבצע heapify-down בעלות מקסימלית של $d \cdot \log_d n$ השוואות, ועבור n פעולות כאלה נקבל $d \cdot n \log_d n$ השוואות, כנדרש.

Benchmark measurements – part 2 (DecreaseKey)

delta	d	Avg. no. comparisons
1	2	99,999
1	3	99,999
1	4	99,999
100	2	152,953
100	3	130,858
100	4	122,891
1000	2	303,453
1000	3	213,246
1000	4	18,1145

הערה 1: לפי הניסוח של השאלה ולפי מה שנכתב בפורום אין קשר ישיר בין המדידות שביצענו בסעיף לבין הניתוח התאורטי – במדידות יש מקרה ספציפי של Decrease-key לפי סדר הכנסה עם delta קבוע, ובשאלה התבקשנו לעשות זאת בלי תלות ב delta, ולפי התשובות בפורום, עבור סדרה **כלשהי** של פעולות ולא בהכרח לפי סדר ההכנסה.

הערה 2: המקרה delta=1 הוא מקרה קצה, בו לא ייתכן שיבוצעו השוואות כלל, למעט השוואה אחת של כל ילד לאביו – סה"כ 99,999 השוואות כי המקרה שבו עושים decrease-key לשורש לא מוסיפה השוואה. הסיבה לזה היא המחיקה לפי סדר ההכנסה: עבור delta=1 המקרה היחיד בו תיתכן השוואה היא המקרה בו אבא של הצומת שהפעלנו עליו Decrease-key הוא בעל מפתח זהה לצומת. זה לא ייתכן, כי אם נתבונן בצומת, וניצור שרשרת ממנו למעלה באופן הבא: נשרשר אותו אל אביו אם יש להם את אותו מפתח ואם לא, נעצור. נקבל שרשרת שבה האיבר הגבוה ביותר בשרשרת הוכנס ראשון, אחריו הוכנס זה שמתחתיו וכן הלאה, כאשר האחרון שהוכנס הוא זה שבתחתית השרשרת. הסיבה לזה היא שheapify-up לא מחליף בין צמתים עם אותו מפתח. לכן, כשנפעיל decrease-key על כל איברי הערימה, הוא יופעל על השרשרת החל מהגבוה ביותר עד לנמוך ביותר, ובאופן זה כלל הערימה לא יופר בשום שלב. זה נכון לכל צומת בעץ ולכן לא יבוצעו השוואות כלל בכל התהליך.

תשובה לשאלה:

פעולת decrease-key יחידה עולה $O(\log_d n)$ במקרה הגרוע, לכן סדרה של n פעולות במקרה הגרוע היא $O(n \log_d n)$. נראה דוגמה של n פעולות Decrease-key עבורן נבצע $n \log_d n$ השוואות ובכך נוכיח שהחסם הוא הדוק (חסם תטא):

נתבונן בעץ מלא שבו כל המפתחות זהים, נניח שכולם 0. בשלב ראשון נפעיל decrease-key על עלה כלשהו עם delta=1 ובכך נשווה אותו לכל הרמות מעליו, כלומר $\log_d n$ השוואות והוא יהפוך לשורש. בשלב השני נפעיל decrease-key על עלה אחר עם מפתח 0 עם delta=2. כעת הוא גם יעלה כל הדרך למעלה ויהפוך לשורש, כלומר $\log_d n$ השוואות. נמשיך כך, ובכל שלב מגדיל את delta ב-1 על מנת שיפועפע עד השורש. נשים לב שהאלגוריתם מוגדר היטב, כי 0 הוא המפתח המינימלי בעץ, לכן כל עוד לא שינינו את כל האיברים יש צומת עם מפתח 0 ומהמינימליות שלו – הוא עלה. ואנחנו מבצעים את האלגוריתם עד שאין צמתים עם מפתח 0 כלומר בדיוק n פעמים, פעם אחת על כל צומת. סה"כ $n \log_d n$ השוואות, כנדרש.