University of Zurich

Department of Informatics
Database Technology Group
Prof. Dr. Anton Dignös

Informatics II
Exercise 5

Published date: March 19, 2021
Labs date: Week 5

Goal:

- Implement `heapify(A, i)` and `BuildHeap(A,n)` in C.
- Study priority queue using heap.
- Practise how partition works in quicksort.

Heap and Heapsort

Task 1. This is a follow-up question to the algorithms `Heapify(A, n, i)` and `BuildHeap(A,n)` taught in the lecture. Recall that `A` is an array of n integers. In this task, we are using max-heapify and building a max-heap.

1. Implement `Heapify(A, n, i)` and `BuildHeap(A,n)` in C.

```c
void heapify(int A[], int i, int n) {
  int m = i;
  int l = 2 * i;
  int r = 2 * i + 1;

  if (l < n && A[l] > A[m])
    m = l;

  if (r < n && A[r] > A[m])
    m = r;

  if (m != i) {
    swap(&A[i], &A[m]); // exchange A[i] and A[m];
    heapify(A, m, n);
  }
}
```

```c
void buildHeap(int A[], int n) {
  for (int i = n / 2 − 1; i ≥0; i−−)
    heapify(A, i, n);
}
```

2. In `BuildHeap(A,n)`, why is the loop that starts from $\lfloor n/2 \rfloor$ instead of n sufficient? Explain.

```
Algorithm: BuildHeap(A, n)
for $i = \lfloor n/2 \rfloor$ to 1 do
    Heapify($A$, $i$, $n$)
```

All elements from the position $\lfloor n/2 \rfloor + 1$ to n, both included, are examined by heapify function in the loop. For example, assume i = n / 2, then in heapify(A, n / 2, n), l = i * 2 = n; hence element at position n is examined.

Task 2. Heap can be used for designing advanced data structures. In this task, we study one of the most popular applications of a heap – *priority queue*. We use the max-heap to construct a (max) prioprity queue.

A ***priority queue*** is a data structure that maintains an array of elements, and each element has a "key" and "priority". This array of elements is a heap in terms of the "priority" values, and both values of key and priority are integers. As the struct has not been taught, we store values of key and priority of the same element in the same position of two arrays: `int priority[]` and `int key[]`, and we always keep positions for key and prioprity of an element the same. You can think of an element as a bundle of two values of the same position from `priority[]` and `key[]`. Differences in using "index" and "position" in C and pseudocode, see arrays of the cheat sheet.

`priority`	15	13	9	5	12
`key`	5	2	1	4	6

Table 1: The priority queue with 5 elements. The first element has the priority = `priority[1]` = 15 and key = `key[1]` = 5. This array of elements is a prioprity queue because `priority[]` is a heap.

A prioprity queue supports the MAXIMUM, EXTRACT-MAX, INCREASE-PRIORITY and INSERT functions. Next, we study how to implement these functions.

1. MAXIMUM(`int priority[]`, `int key[]`) returns the key of the element with the largest priority.

```
Algorithm: MAXIMUM(int priority[], int key[])
return ?
```

   (a) Take the priority queue in Table 1 as input, what does MAXIMUM function return?
   5

   (b) Complete MAXIMUM function by filling in the "?".

```
Algorithm: MAXIMUM(int priority[], int key[])
return $key[1]$
```

   (c) What's the time commplexity? Explain.
   $O(1)$ since it simply returns the first element in array *key*.

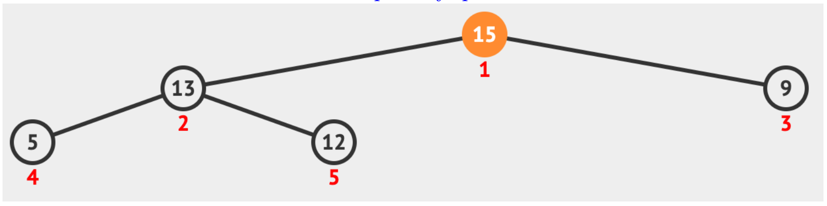
2. EXTRACT-MAX(`int priority[]`, `int key[]`, `int n`) removes and returns the element with the largest priority.

```
Algorithm: EXTRACT-MAX(int priority[], int key[], int n)
if $n < 1$ then
    error "No element to be removed."
$max$ = ?
priority[?] = priority[n]
key[?] = key[n]
$n = n - 1$
heapify(priority, key, ?, ?) // max-heapify
/* key and priority of the same element have the same
   position after heapify.                              */
return $max$
```

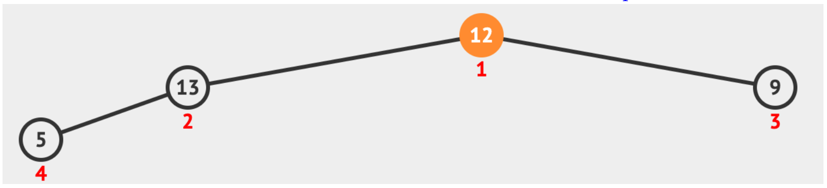
(a) Take the priority queue in Table 1 as input where n = 5, illustrate how values of `priority[]` and `key[]` change in the function EXTRACT-MAX.

First, we remove the first element and move the last element to the first position, then we do the normal heapify process. The key of each element is not shown in the figure, and the numbers in red are the positions of the elements.

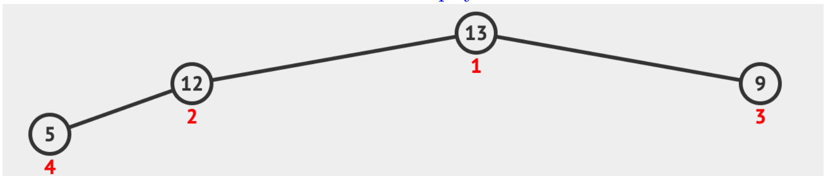
Initial priority queue.



After remove first element and move last element to position 1.



After heapify.



Finally, we have $priority = [13, 12, 9, 5]$, and accordingly $key = [2, 6, 1, 4]$.

(b) Complete EXTRACT-MAX function by filling in the "?".

```
Algorithm: EXTRACT-MAX(int priority[], int key[], int n)
if $n < 1$ then
    error "No element to be removed."
$max$ = key[1]
priority[1] = priority[n]
key[1] = key[n]
$n = n - 1$
heapify(priority, key, n, 1) // max-heapify
/* key and priority of the same element have the same position
   after heapify.                                            */
return $max$
```

(c) What's the time commplexity? Explain.

The running time of EXTRACT-MAX is $O(\lg n)$, since the most time consuming part is the heapify that is in $O(\lg n)$.

3. INCREASE-PRIORITY(`int priority[]`, `int key[]`, `int i`, `int p`) increases the priority of element at position i to p.

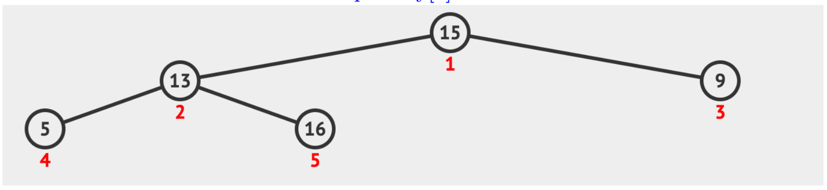
We first update the priority of the element at position i. Increasing the priority of the element at position i might violate that its priority value is larger than its parent's priopity, if so, we swap the element at position i and its parent. To guarantee the correctness of the heap, we might need to change the positions several elements.

```
Algorithm: INCREASE-PRIORITY(int priority[], int key[], int i, int p)
priority[i] = p
parent = Parent(i)
while $i \neq 0$ & $priority[i] > priority[parent]$ do
    swap(priority[i], priority[parent])
    swap(key[i], key[parent])
    i = ?
    parent = ?
```

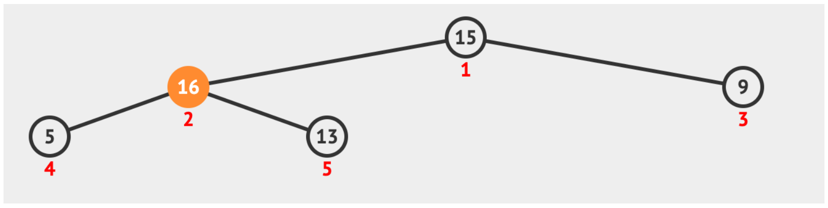
(a) Set the priority queue in Table 1 as input; i = 5; p = 16. We increase priority of element at position i = 5 to p = 16. Illustrate how values of `priority[]` and `key[]` change in the function INCREASE-PRIORITY.

First, we set the value at position 5 to be 16, then we do a itertive check with parents until a correct position. The key of each element is not shown in the figure, and the numbers in red are the positions of the elements.

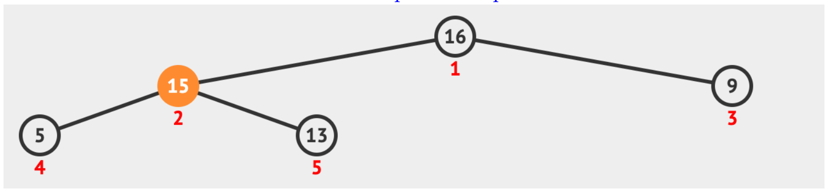
set priority[5] = 16



Switch with its parent at position 2



Switch with its parent at position 1



Finally, we have *priority* $= [16, 15, 9, 5, 13]$, and accordingly *key* $= [6, 5, 1, 4, 2]$.

(b) Complete INCREASE-PRIORITY function by filling in the ”?”.

```
Algorithm: INCREASE-PRIORITY(int priority[], int key[], int
  i, int p)
priority[i] = p
parent = Parent(i)
while i ≠ 0 & priority[i] > priority[parent] do
    swap(priority[i], priority[parent])
    swap(key[i], key[parent])
    i = parent
    parent = Parent(i)
```

(c) What's the time commplexity? Explain.

The running time of INCREASE-PRIORITY on an n-element heap is $O(\lg n)$, since the path traced from the current postion to the first position by calling the Parent() function has length $O(\lg n)$.

4. INSERT(`int priority[]`, `int key[]`, `int n`, `int k`, `int p`) inserts an element with key = k and priority = p into the priopity queue with n elements.

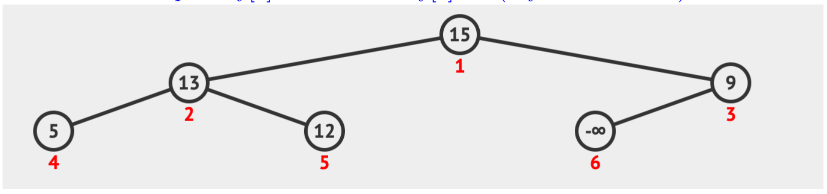
We first put the inserted element in the end of the priority queue and temporially set the priority of this element to $-\infty$. Then it calls INCREASE-PRIORITY to set the priority of inserted element to p and maintain the max-heap property.

```
Algorithm: INSERT(int priority[], int key[], int n, int k,
  int p)
n = n + 1
priority[?] = −∞
key[?] = k
INCREASE-PRIORITY(priority, key, n, p)
```

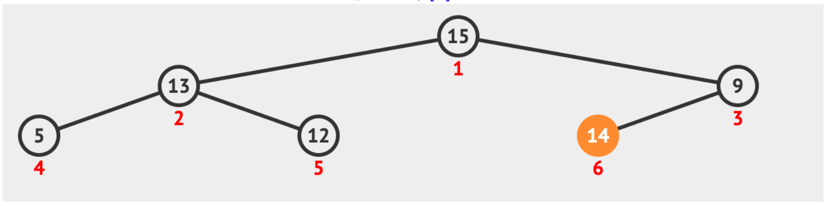
(a) Set the priority queue in Table 1 as input; insert an element with key = 3 and priority = 14. Illustrate how values of `priority[]` and `key[]` change in the function INSERT

First, we set the value at position 6 to be 14, then we do a itertive check with parents until the node arrives at a proper position. The key of each element is not shown in the figure, and the numbers in red are the positions of the elements.

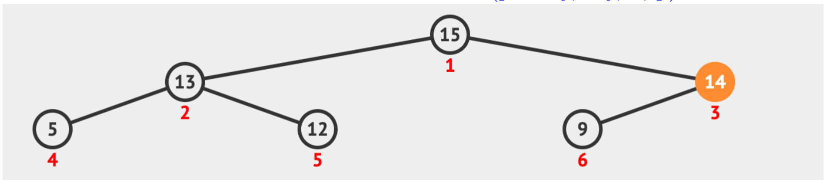
set priority[6] $= -\infty$ and key[6] $= 3$(key is not shown.)



set priority[6] $= 14$



After call INCREASE-PRIORITY(priority, key, n, p)



Finally, we have $priority = [15, 13, 14, 5, 12, 9]$, and accordingly $key = [5, 2, 3, 4, 6, 1]$.

(b) Complete INSERT function by filling in the "?".

```
Algorithm: INSERT(int priority[], int key[], int n, int k, int p)
n = n + 1
priority[n] = $-\infty$
key[n] = k
INCREASE-PRIORITY(priority, key, n, p)
```

(c) What's the time commplexity? Explain.

The running time of INSERT on an n-element heap is $O(\lg n)$, since the most time consuming part is the INCREASE-PRIORITY that is in $O(\lg n)$.

Assume that we are maintaining a min priority queue using a min-heap. Consider the following functions.

a) Write pseudocode for the MINIMUM function.

```
Algorithm: MINIMUM(int priority[], int key[])
return $key[1]$
```

b) Write pseudocode for EXTRACT-MIN function.

```
Algorithm: EXTRACT-MIN(int priority[], int key[], int n)
if $n < 1$ then
    error "No element to be removed."
$min$ = key[1]
priority[1] = priority[n]
key[1] = key[n]
$n = n - 1$
heapify(priority, key, n, 1) // min-heapify
/* key and priority of the same element have the same position
   after heapify.                                             */
return $min$
```

c) Write pseudocode for DECREASE-PRIORITY function.

```
Algorithm: DECREASE-PRIORITY(int priority[], int key[], int
 i, int p)
priority[i] = p
parent = Parent(i)
while $i \neq 0$ & $priority[i] < priority[parent]$ do
    swap(priority[i], priority[parent])
    swap(key[i], key[parent])
    i = parent
    parent = Parent(i)
```

d) Write pseudocode for INSERT function.

```
Algorithm: INSERT(int priority[], int key[], int n, int k, int p)
n = n + 1
priority[n] = $+\infty$
key[n] = k
DECREASE-PRIORITY(priority, key, n, p)
```

Quicksort

Task 3. The key element of the quicksort algorithm is its partitioning procedure.

1. Complete the boxes below to complete algorithm Partition(A, l, r) that partitions array $A[l..r]$.

```
Algorithm: Partition(A,l,r)
$x$ = A[l];
i = l;
for $j$ = [ $l + 1$ ] to [ $r$ ] do
    if $A[j]$ [ $\leq$ ] $x$ then
        i = [ i + 1 ] ;
        exchange A[i] and A[j];
exchange [ A[i] ] and [ A[l] ] ;
return [ i ] ;
```

2. Implement `Partition(A,l,r)` in C.

```c
int partition(int A[], int l, int r) {
    int x = A[l], i = l;
    for (int j = l + 1; j ≤r; j++) {
        if (A[j] ≤x) {
            i += 1;
            int t = A[i];
            A[i] = A[j];
            A[j] = t;
        }
    }
    int t = A[i];
    A[i] = A[l];
    A[l] = t;
    return i;
}
```

Task 4. Given an array `A[]` of n integers, find the k-th biggest number of `A`.
Example 1:

Input: $[3, 2, 1, 5, 6, 4]$ and $k = 2$
Output: 5

Example 2:

Input: $[3, 2, 3, 1, 2, 4, 5, 5, 6]$ and $k = 4$
Output: 4

Provide the solution using Partition function in Exercise 3 and implement the solution in C.
Hint: think of how we do partition in quick sort.

We can solve it by first quick sort the array in an ascending order, then return the kth element at position n - k + 1, which requires $O(n \lg n)$ time. Actually we can do it even faster.

Recall in quick sort, after each partition operation, assume that the pivot is at position i. so all elements from $A[l...i-1]$ are no bigger than $A[i]$, all elements from $A[i+1...r]$ are no less than to $A[i]$. In an array of n integer, when i = n - k + 1 that is the k-th integer from the tail, we find the k-th largest number.

We can modify the quick sort algorithm to solve this problem. For an array of n integers, in the process partition, when i = n - k + 1 obtained from the partition is exactly the index we need, we will return $A[i]$ directly. If the i is smaller than n - k + 1, we will search the right subarray, otherwise we will search the left subarray. This turns the original recursion of two subarrays into only one subarray, which improves time efficiency. This is the "quick selection" algorithm.

```c
int partition(int A[], int l, int r) {
    int x = A[l], i = l;
    for (int j = l + 1; j <= r; j++) {
        if (A[j] <= x) {
            i += 1;
            int t = A[i];
            A[i] = A[j];
            A[j] = t;
        }
    }
    int t = A[i];
    A[i] = A[l];
    A[l] = t;
    return i;
}


int quickSelect(int A[], int l, int r, int index) {
    int q = partition(A, l, r);
    if (q == index) {
        return A[q];
    } else {
        return q < index ? quickSelect(A, q + 1, r, index)
                         : quickSelect(A, l, q - 1, index);
    }
}

int findKthLargest(int A[], int n, int k) {
    return quickSelect(A, 0, n - 1, n - k); // k-largest element are at index n - k
}
```

We can also use heap sort to solve this problem: create a max-heap and delete first number $k - 1$ times, then after the deletions, the first number of the heap is k-largest number. This method also has a $O(k \log n)$ running time.