CS213/293 Data Structure and Algorithms 2025

Lecture 8: Heap

Instructor: Ashutosh Gupta

IITB India

Compile date: 2025-09-13

Topic 8.1

Priority queue

Scheduling problem

On a computational server, users are submitting jobs to run on a single CPU.

- A user also declares the expected run time of the job.
- Jobs can be preempted.

Policy: shortest remaining processing time, which allows interruption of a job if a new job with a smaller run time is submitted.

The policy minimizes average waiting time.

Scheduling problem operations

We need the following operations in the scheduling problem.

- Update the remaining time in every tick
- Delete a job when the remaining time is zero
- Find the next job to run
- Insert a job when it arrives

Definition 8.1

In a priority queue, we dequeue the highest priority element from the enqueue elements with priorities.

Interface of priority queue

https://en.cppreference.com/w/cpp/container/priority_queue

- `priority_queue<T,Container,Compare> q` : allocates new queue q
- `q.push(e)` : adds the given element e to the queue.
- `q.pop()` : removes the highest priority element from the queue.
- `q.top()` : access the highest priority element.

- `Container` class defines the physical data structure where the queue will be stored. The default value is `Vector`.
- `Compare` class defines the method of comparing priorities of two elements.

Topic 8.2

Implementations of priority queue

Implementation using unsorted linked list/array

In case we use a linked list,

- We implement `q.push` by inserting the element at the front of the linked list, which is $O(1)$ operation.
- We need to scan the entire list to find the maximum for implementing `q.pop` and `q.top`

Exercise 8.1

How will we implement a priority queue over unsorted arrays?

Implementation using sorted linked list/array

In case we use a linked list,

- The maximum will be at the end of the list. We can implement `q.pop` and `q.top` in $O(1)$.
- However, `q.push(e)` needs to scan the entire list to find the right place to insert e, which is $O(n)$ operation.

Priority queue

The priority queue is one of the fundamental containers.

Many other algorithms assume access to efficient priority queues.

We will define a data structure heap that provides an efficient implementation for the priority queue.

Commentary: The heap is like the red-black tree, which provides an efficient implementation for ordered maps.

Topic 8.3

Heap - partial sorting!

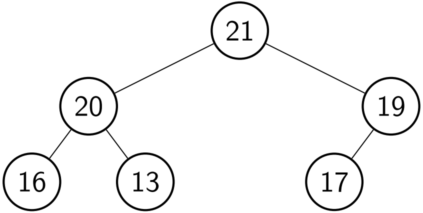
Heap

Definition 8.2

A heap T is a binary tree such that the following holds.

- (structural property) All levels are full except the last one and the last level is left filled.
- (heap property) for each non-root node n, $key(n) \leq key(parent(n))$.

Example 8.1



Exercise 8.2

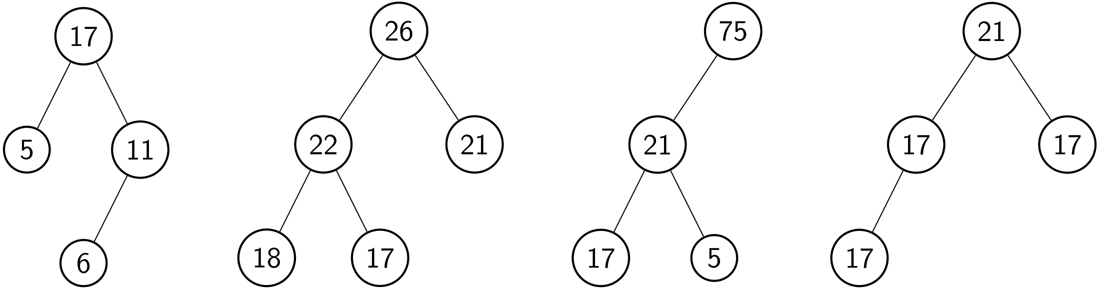
a. Show that nodes on a path from the root to a leaf have keys in non-increasing order.

b. The above definition is called maxheap. Can we symmetrically define minheap?

Exercise: identify heap

Exercise 8.3

Which of the following are Heaps?



Algorithm: maximum

Algorithm 8.1: MAXIMUM(Heap T)

---

return $T[0]$

- Correctness
  - Let us suppose the maximum is not at the root.
  - There is a node n that has maximum key but $parent(n)$ has a smaller key, which violates heap condition.
  - Contradiction.
- Running time is $O(1)$.

Height of heap

Let us suppose a heap has n nodes and height h.

The number of nodes in a complete binary tree of height h is $2^{h+1} - 1$.

Therefore,

$$2^h - 1 < n \leq 2^{h+1} - 1.$$

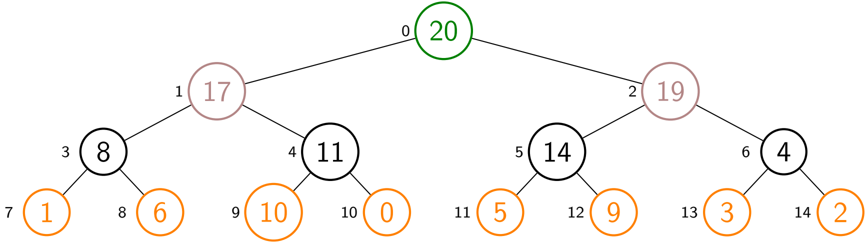
Therefore $h = \lfloor \log_2 n \rfloor$

Exercise 8.4

Give an example of a heap that touches the lower bound.

Storing heap

Let us number the nodes of a heap in the order of level.



parent$(i) = (i-1)/2$, *left*$(i) = 2i+1$, and *right*$(i) = 2i+2$.
We place the nodes on an array and traverse the heap using the above equations.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
20	17	19	8	11	14	4	1	6	10	0	5	9	3	2

Since the last level is left filled, we are guaranteed the nodes are contiguously placed.
Instead of writing *key*(i) for node i in heap T, we will write $T[i]$ to indicate the key.

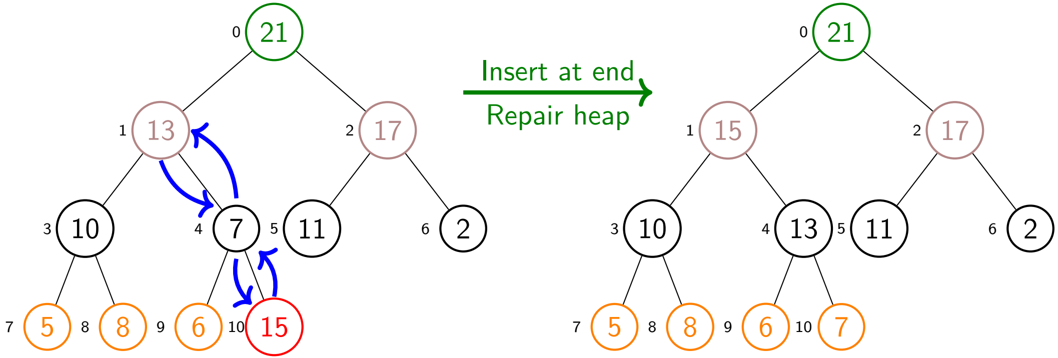
Topic 8.4

Insert in heap – jostling to front

Example: insert in heap

Example 8.2

Where do we insert 15?



- Insert at the first available place, which is easy to spot. (Why?)
- Move up the new key if the heap property is violated.

Algorithm: Insert

Commentary: At the time of insert, we may have not have enough space in T. On insert we may need to apply expand operations like stack and queue.

Algorithm 8.2: INSERT(Heap T, key k)

$i := T.size$;
$T[i] := k$;
while $i > 0$ *and* $T[parent(i)] < T[i]$ **do**
    SWAP(T, parent(i), i);
    $i := parent(i)$
$T.size := T.size + 1$;

- Correctness
  - Structural property holds due to the insertion position.
  - Due to the heap property of input T, the path to i (not including i) the nodes must be in non-increasing order.
  - Let i_0 be the value of i when the loop exits.
  - INSERT replaces the keys of the nodes in the path from i_0 to $T.size$ with the keys of their parents, which implies the keys do not decrease at the internal nodes.
  - Therefore, no introduction of a violation.
  - Therefore, we will have a heap at the end.
- Running time is $O(\log T.size)$.

Exercise 8.5

Why do we need the phrase “not including” and “internal” in the above proof?

Topic 8.5

Heapify: fix the almost heaps

Heapify : a basic operation on a heap

Input to HEAPIFY:

- Let i be a node of a binary tree T with the structural property of heap
- Let us suppose the binary trees rooted at $left(i)$ and $right(i)$ are valid heaps.
- $T[i]$ may be smaller than its children and violates the heap property.

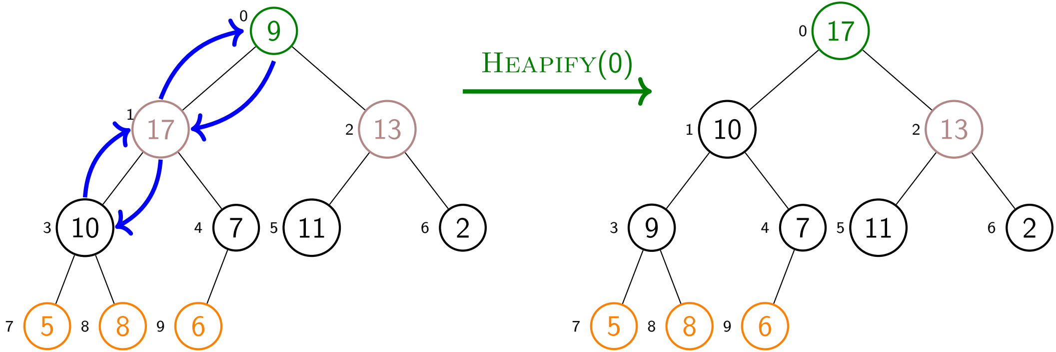
Output of HEAPIFY:

HEAPIFY makes the binary tree rooted at i a heap by pushing down $T[i]$ in the tree.

Example: HEAPIFY

Example 8.3

The trees rooted at positions 1 and 2 are heaps. We have a violation at position 0. Heapify will fix the problem by moving the key down.



- Keep moving down to the child which has the maximum key. (Why?)

Algorithm: Heapify

Algorithm 8.3: HEAPIFY(Heap T, i)

$c := \text{INDEXWITHLARGESTKEY}(T, i, \mathit{left}(i), \mathit{right}(i))$ //assume $T[i] = -\infty$ if $i \geq T.\mathit{size}$.
if $c == i$ **then** **return**;
SWAP(T, c, i);
HEAPIFY(T,c);

- Correctness
  - Same as insert, but we are pushing down.
- Running time is $O(\log T.\mathit{size})$.

Commentary: Assumption $T[i] = -\infty$ if $i \geq T.\mathit{size}$ is a convenience of notation. We may have a situation, where the $T[i]$ exists and has some key. Without loss of correctness, we can interpret them as if the key is $-\infty$. We will need this interpretation later for HEAPSORT.

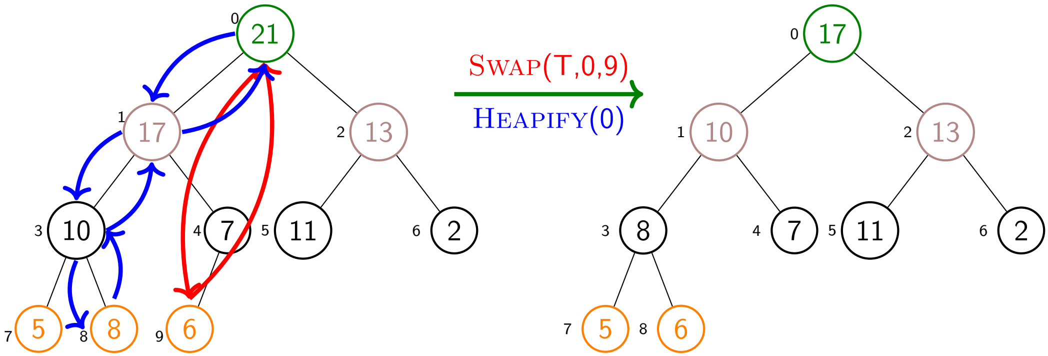
Topic 8.6

Delete maximum in heap

Example: DeleteMax

Example 8.4

Let us delete 21 at position 0.



- Swap with the last position, delete the last position, and run Heapify.

Algorithm: DeleteMax

Algorithm 8.4: DELETEMAX(Heap T)

```
SWAP(T, 0, T.size − 1);
T.size := T.size − 1;
HEAPIFY(T, 0);
return T[T.size];
```

- Correctness
  - The maximum element is removed and heapify returns a heap.
- Running time is $O(\log T.size)$.

Topic 8.7

Build heap

Build heap https://en.cppreference.com/w/cpp/algorithm/make_heap

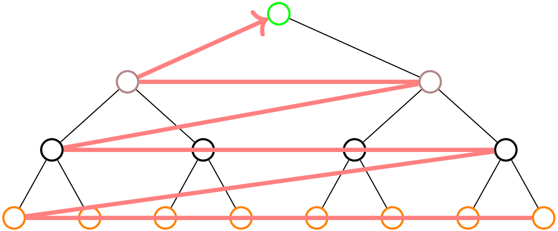
- Input: A binary tree T that has the structural property
  - If the structural property holds, then the T is an array
- Output: A heap over elements of T

Algorithm: BuildHeap

Algorithm 8.5: BuildHeap(Heap T)

for $i := T.size - 1$ **down to** *0* **do**
    Heapify(T, i)

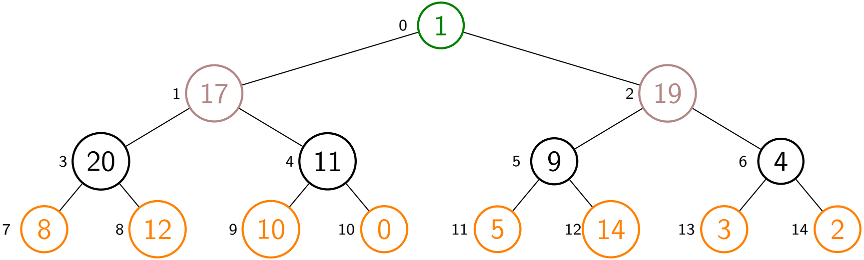
Order of processing in BuildHeap.



Example: BuildHeap

Example 8.5

Consider sequence 1 17 19 20 11 9 4 8 12 10 0 5 14 3 2. Let us fill them in the following tree.



BuildHeap traverses the tree bottom up. Heapify calls execute only the following swaps.

- Heapify(T,5): Swap(T,5,12)
- Heapify(T,1): Swap(T,1,3)
- Heapify(T,0): Swap(T,0,1); Swap(T,1,3); Swap(T,3,8);

The other calls to Heapify will not apply any swaps.

Correctness of BUILDHEAP

- We do not change the structure of T in BUILDHEAP, therefore the tree at any i has the structural property.
- Correctness by induction
  - **Base case:**
    If i does not have children, it is already a heap.
  - **Induction step:**
    We know $left(i) > i$ or $right(i) > i$.
    Due to the induction hypothesis, both the subtrees are heap before processing i.
    The tree at i has structural property. Therefore, HEAPIFY(T, i) will return a heap rooted at i.

Running time of BUILDHEAP

Let us suppose T is a complete tree with n nodes.

Recall: Heapify for a node at height h has $O(h)$ swaps.

At height h the number of nodes is $\lceil n/2^{h+1} \rceil$ and the height of T is $\lfloor \log n \rfloor$.

The total running time of BUILDHEAP is

$$\sum_{h=0}^{\lfloor \log n \rfloor} O(h) \lceil n/2^{h+1} \rceil = O(\frac{n}{2} \sum^{\lfloor \log n \rfloor} \frac{h}{2^h})$$

Commentary: We used identities $O(f)g = O(fg)$ and $O(f) + O(g) = O(f + g)$.

Since $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$, the running time is $O(n)$.

Calculation to show $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

We know

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$$

After differentiating over x,

$$\sum_{h=0}^{\infty} hx^{h-1} = \frac{1}{(1-x)^2}$$

After multiplying with x,

$$\sum_{h=0}^{\infty} hx^{h} = \frac{x}{(1-x)^2}$$

After putting $x = 1/2$,

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

Topic 8.8

Heapsort

Heapsort

Algorithm 8.6: HeapSort(Tree T)

```
T.size = |nodes of T|;
BuildHeap(T);
while T.size > 0 do
    DeleteMax(T)
```

- Since DeleteMax moves maximum to $T.size - 1$ position, the array is sorted in place.
- Running time:
  - BuildHeap is $O(n)$
  - DeleteMax(T) is $O(\log i)$ at size i.
- Total running time: $O(n \log n)$.

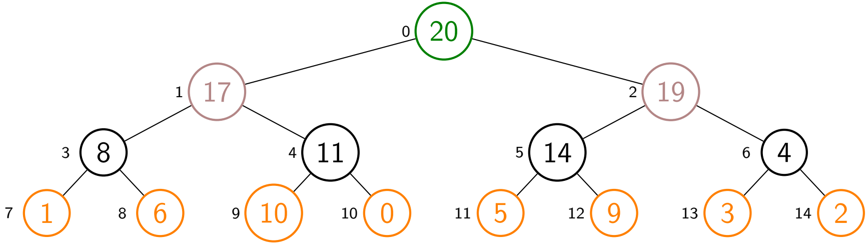
Exercise 8.6

Both BuildHeap and the above loop have iterative runs of Heapify.
Why are their running time complexities different?

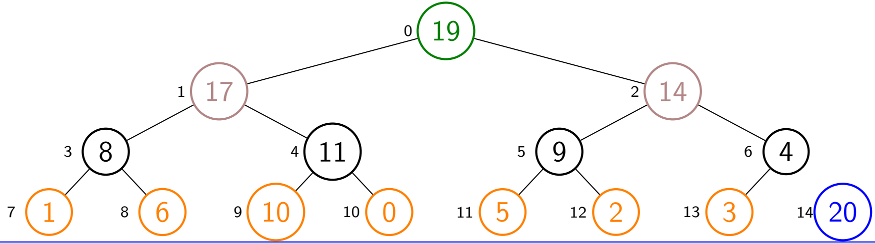
Commentary: Please solve the above exercise to clearly understand the relevant mathematics.

Example: HEAPSORT

Consider the following Heap obtained after running BUILDHEAP.

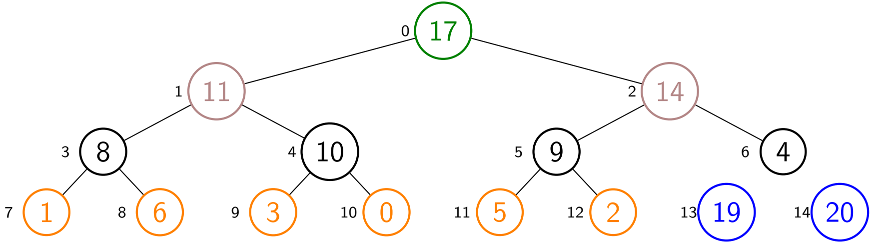


After the first DELETEMAX,



Example: Heapsort(2)

After the second DeleteMax,



DeleteMax has placed 19 and 20 at their sorted position.

Topic 8.9

Tutorial problems

Exercise: implement scheduling problem

Exercise 8.7

Give an implementation for the scheduling problem using Heap.

Exercise: Why heap?

Exercise 8.8

Can a Priority Queue be implemented as a red-black tree? What advantages does a heap implementation have over a red-black tree implementation?

Exercise: BST and Heap (Midterm 2023)

Exercise 8.9

Give a tree, if exists, that is a binary search tree, is a heap, and has more than two nodes. If such a tree does not exist, give a reason.

Exercise: 2D-matrix

Exercise 8.10

Suppose we have a 2D array where we maintain the following conditions: for every (i,j), we have $A(i,j) \leq A(i+1,j)$ and $A(i,j) \leq A(i,j+1)$. Can this be used to implement a priority queue?

Exercise: kth smallest element

Exercise 8.11

Given an unsorted array find the kth smallest element using a priority queue.

Exercise: Merge heaps (Midterm 2023)

Exercise 8.12

Given two heaps give an efficient algorithm to merge the heaps.

Topic 8.10

Problems

True or False

Exercise 8.13

Mark the following statements True / False and also provide justification.

1. The worst-case running time of HeapSort is linear.
2. The worst-case running time of Heapify is linear.

Exercise: Leftist heap (midsem 2024)

Algorithm 8.7: MERGE(LeftistHeap a,LeftistHeap b)

if $a == Null$ **then return** b;
if $b == Null$ **then return** a;
if $(value(b) < value(a))$ **then**
  return MERGE(b, a)

right(a) := MERGE(right(a),b);
if $npl(left(a)) < npl(right(a))$ **then**
  SWAP(left(a),right(a))

npl(a) = min(npl(left(a)),npl(right(a)))+1 **return** a

Algorithm 8.8: INSERT(Node a, LeftistHeap b)

left(a)=right(a):= Null; Return MERGE(a,b)

Algorithm 8.9: DELETEMIN(LeftistHeap a)

if $a == Null$ **then return**;
return merge(left(a),right(a))

Exercise 8.14

A leftist heap is a heap without the structural property. Instead, it satisfies leftist property npl(left(n)) $\geq$ npl(right(n)) for each node n, where

$$npl(n)=\begin{cases} -1 & \text{if n is null} \\ \min(npl(left(n)),npl(right(n)))+1 & \text{otherwise.} \end{cases}$$

In the left, we define operations on the heap. Prove that insert and deleteMin return leftist heap and are O(log m), where m is the size

Commentary: **Solution:** Base case: merge returns inputs. So, the returned object is a leftist heap. Inductive step: Since merge is a recursive program, we assume that the recursive call returns a leftist heap and the root of the returned heap is the root of one of the two inputs. Since the left child of a is never changed, and b and the initial left child of a are smaller than a, the heap property certainly holds on *a* at the last return. The leftist property holds because of the swap.

Merge recursive calls traverse the right paths of input trees. So, the running time depends on the length of the right paths. We prove if the right path has a length of at least r, the tree has at least 2^r -1 nodes. Base case: r = 1. The tree has at least one node. Inductive step: The right subtree has a right path of at least r - 1 nodes, so it has at least 2^{r-1} - 1 nodes. The left subtree must also have a right path of at least r - 1 (otherwise, there is a null path of r - 3, less than the right subtree). Again, the left has 2^{r-1} - 1 nodes. Sum the nodes.

End of Lecture 8