CMPSC 465: LECTURE VIII

Binary Heaps

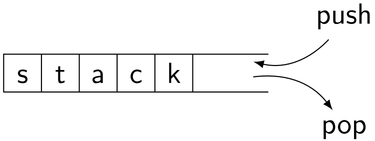
Ke Chen

September 15, 2025

Quick review: stack and queue

Stack

- Last In, First Out (LIFO).
- Constant time insert (push) and delete (pop) at the same end.
- Common implementations: array, linked list.



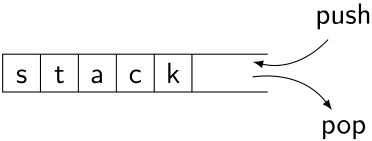
Quick review: stack and queue

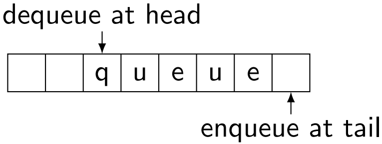
Stack

- Last In, First Out (LIFO).
- Constant time insert (push) and delete (pop) at the same end.
- Common implementations: array, linked list.

Queue

- First In, First Out (FIFO).
- Constant time insert (enqueue) and delete (dequeue) at different ends.
- Common implementations: circular array, linked list with a tail pointer.





Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

- $\diamond$ GetMax
- $\diamond$ Insertion
- $\diamond$ Deletion
- $\diamond$ ChangePriority

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

Sorted list

- ⋄ GetMax
- ⋄ Insertion
- ⋄ Deletion
- ⋄ ChangePriority

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list
◇ GetMax	$O(1)$
◇ Insertion	
◇ Deletion	
◇ ChangePriority	

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list
◇ GetMax	$O(1)$
◇ Insertion	$O(n)$
◇ Deletion	$O(n)$
◇ ChangePriority	$O(n)$

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list	
◇ GetMax	$O(1)$	
◇ Insertion	$O(n)$	array can do binary search but in worst-case requires linear shift; linked list supports constant-time insertion/deletion but cannot binary search
◇ Deletion	$O(n)$	
◇ ChangePriority	$O(n)$	

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list	Array with pointer to max
⋄ GetMax	$O(1)$	
⋄ Insertion	$O(n)$	
⋄ Deletion	$O(n)$	
⋄ ChangePriority	$O(n)$	

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list	Array with pointer to max
◇ GetMax	$O(1)$	$O(1)$
◇ Insertion	$O(n)$	
◇ Deletion	$O(n)$	
◇ ChangePriority	$O(n)$	

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list	Array with pointer to max
◇ GetMax	$O(1)$	$O(1)$
◇ Insertion	$O(n)$	$O(1)$
◇ Deletion	$O(n)$	
◇ ChangePriority	$O(n)$	

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list	Array with pointer to max	
◇ GetMax	$O(1)$	$O(1)$	
◇ Insertion	$O(n)$	$O(1)$	
◇ Deletion	$O(n)$	$O(n)$	worst-case: max is removed
◇ ChangePriority	$O(n)$		

Priority queue

Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list	Array with pointer to max	
◇ GetMax	$O(1)$	$O(1)$	
◇ Insertion	$O(n)$	$O(1)$	
◇ Deletion	$O(n)$	$O(n)$	worst-case: max is removed
◇ ChangePriority	$O(n)$	$O(n)$	may need to find new max

Priority queue

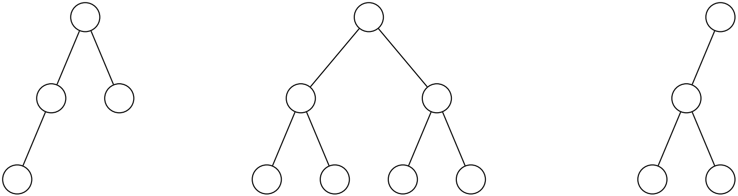
Motivation Can we reorder elements in the container so that those with higher priority are handled first?

	Sorted list	Array with pointer to max	Binary heap
◇ GetMax	$O(1)$	$O(1)$	$O(1)$
◇ Insertion	$O(n)$	$O(1)$	$O(\log n)$
◇ Deletion	$O(n)$	$O(n)$	$O(\log n)$
◇ ChangePriority	$O(n)$	$O(n)$	$O(\log n)$

Complete tree

Definition A binary tree is complete if

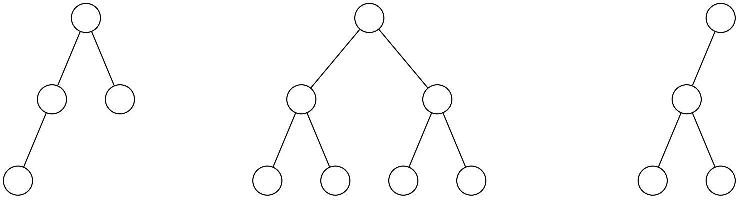
- the tree is completely filled in all levels except possibly in the lowest level; and
- all nodes in the last level are as far left as possible.



Complete tree

Definition A binary tree is complete if

- the tree is completely filled in all levels except possibly in the lowest level; and
- all nodes in the last level are as far left as possible.

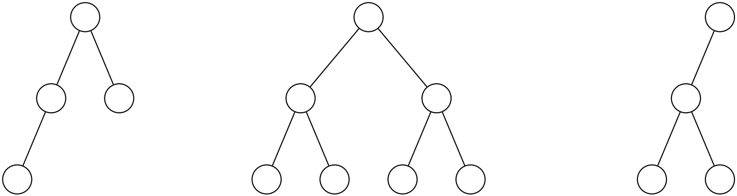


complete

Complete tree

Definition A binary tree is complete if

- the tree is completely filled in all levels except possibly in the lowest level; and
- all nodes in the last level are as far left as possible.



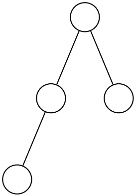
complete

complete

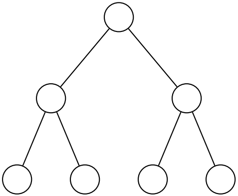
Complete tree

Definition A binary tree is complete if

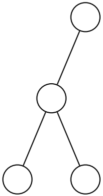
- the tree is completely filled in all levels except possibly in the lowest level; and
- all nodes in the last level are as far left as possible.



complete



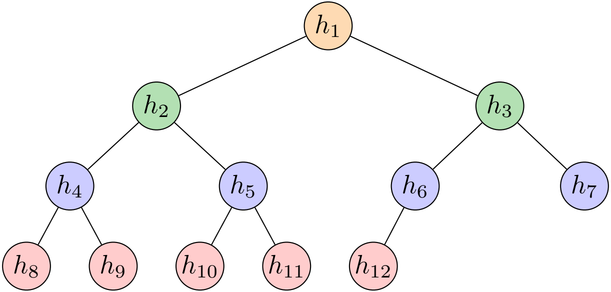
complete



not complete

A complete tree in an array

Observe that a complete tree fits snugly in an array.

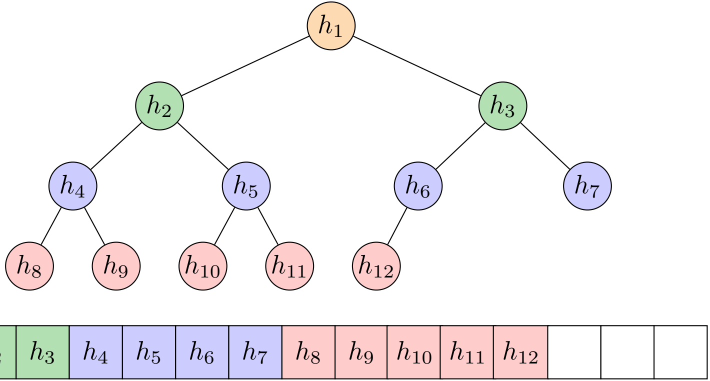


h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}	h_{11}	h_{12}			

A complete tree in an array

Observe that a complete tree fits snugly in an array.

For an index i, $parent(i) = \lfloor i/2 \rfloor$, $left(i) = 2i$, and $right(i) = 2i + 1$.



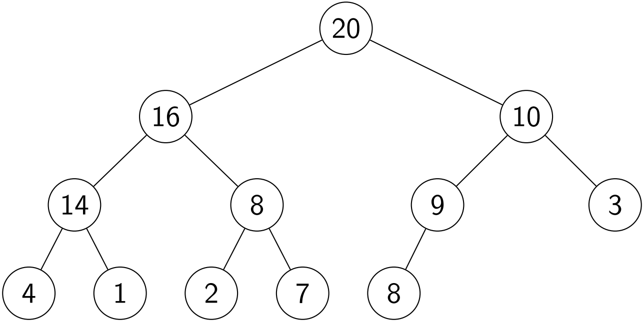
Binary max-heap

A binary max-heap is a complete binary tree that satisfies the Max-Heap Property The key at each node is smaller than or equal to the key of its parent node (except for the root).

Binary max-heap

A binary max-heap is a complete binary tree that satisfies the Max-Heap Property The key at each node is smaller than or equal to the key of its parent node (except for the root).

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8.$



Binary max-heap

A binary max-heap is a complete binary tree that satisfies the

Max-Heap Property The key at each node is smaller than or equal to the key of its parent node (except for the root).

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8.$

- Equivalently, the max-heap property requires $H[i] \leq H[parent(i)]$ for each $i > 1$.

Binary max-heap

A binary max-heap is a complete binary tree that satisfies the

Max-Heap Property The key at each node is smaller than or equal to the key of its parent node (except for the root).

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8.$

- Equivalently, the max-heap property requires $H[i] \leq H[parent(i)]$ for each $i > 1$.
- Duplicate keys are permitted.

Binary max-heap

A binary max-heap is a complete binary tree that satisfies the

Max-Heap Property The key at each node is smaller than or equal to the key of its parent node (except for the root).

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8$.

- Equivalently, the max-heap property requires $H[i] \leq H[parent(i)]$ for each $i > 1$.
- Duplicate keys are permitted.
- Min-heaps are completely analogous.

Binary max-heap

A binary max-heap is a complete binary tree that satisfies the Max-Heap Property The key at each node is smaller than or equal to the key of its parent node (except for the root).

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8.$

- Equivalently, the max-heap property requires $H[i] \leq H[parent(i)]$ for each $i > 1$.
- Duplicate keys are permitted.
- Min-heaps are completely analogous.
- Can be generalized to d-ary heaps.

Binary max-heap

A binary max-heap is a complete binary tree that satisfies the

Max-Heap Property The key at each node is smaller than or equal to the key of its parent node (except for the root).

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8.$

- Equivalently, the max-heap property requires $H[i] \leq H[parent(i)]$ for each $i > 1$.
- Duplicate keys are permitted.
- Min-heaps are completely analogous.
- Can be generalized to d-ary heaps.

Exercise: Does an array sorted in non-decreasing order represent a max-heap? E.g., 10, 8, 7, 7, 3, 2, 2, 1.

Binary max-heap

A binary max-heap is a complete binary tree that satisfies the Max-Heap Property The key at each node is smaller than or equal to the key of its parent node (except for the root).

Example: $H[1..12] = 20, 16, 10, 14, 8, 9, 3, 4, 1, 2, 7, 8$.

- Equivalently, the max-heap property requires $H[i] \leq H[parent(i)]$ for each $i > 1$.
- Duplicate keys are permitted.
- Min-heaps are completely analogous.
- Can be generalized to d-ary heaps.

Exercise: Does an array sorted in non-decreasing order represent a max-heap? E.g., 10, 8, 7, 7, 3, 2, 2, 1.

We will see later that we don't need to sort in order to construct a heap from an array.

Heap operations

Suppose $H[1..n]$ is a binary max-heap.

GetMax Takes $O(1)$, all we have to do is return $H[1]$.

Heap operations

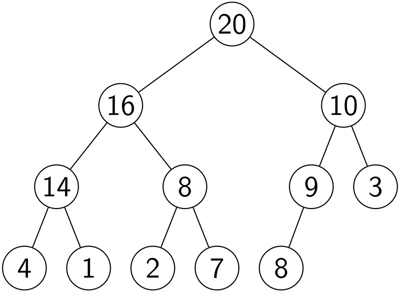
Suppose $H[1..n]$ is a binary max-heap.

GetMax Takes $O(1)$, all we have to do is return $H[1]$.

Insertion

- Insert in the first available position $H[n+1]$.

Example Suppose we want to insert 18 in:



Heap operations

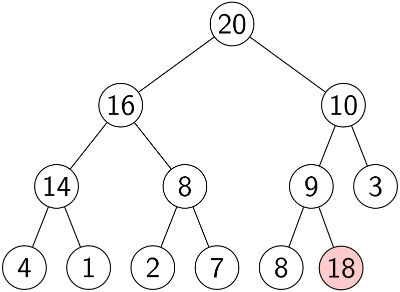
Suppose $H[1..n]$ is a binary max-heap.

GetMax Takes $O(1)$, all we have to do is return $H[1]$.

Insertion

- Insert in the first available position $H[n+1]$.

Example Suppose we want to insert 18 in:



Heap operations

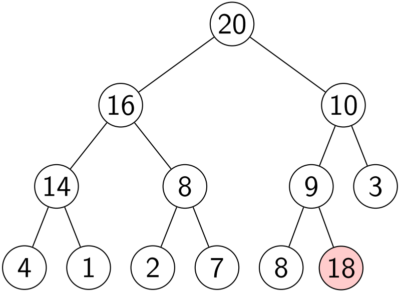
Suppose $H[1..n]$ is a binary max-heap.

GetMax Takes $O(1)$, all we have to do is return $H[1]$.

Insertion

- Insert in the first available position $H[n+1]$.

Example Suppose we want to insert 18 in:

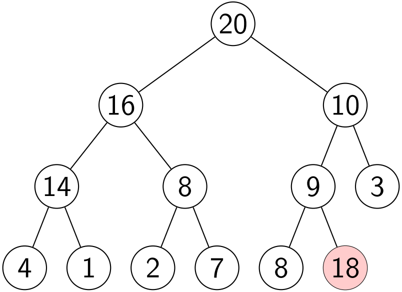


Violate the max-heap property

Heap operations

Insertion

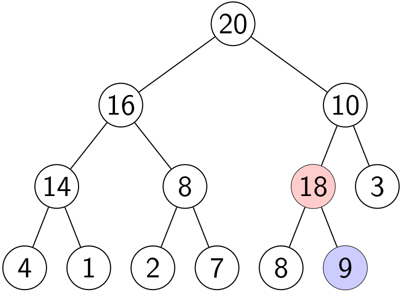
- Insert in the first available position $H[n+1]$.
- **Idea** Let the new element "bubble up" by swapping until it finds the right position.



Heap operations

Insertion

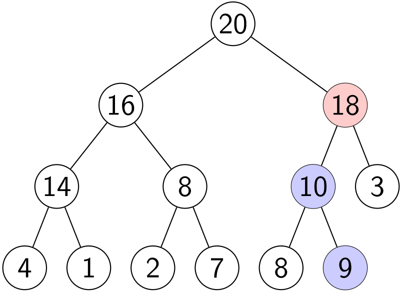
- Insert in the first available position $H[n+1]$.
- Idea Let the new element "bubble up" by swapping until it finds the right position.



Heap operations

Insertion

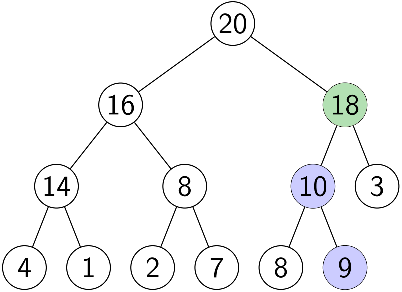
- Insert in the first available position $H[n+1]$.
- Idea Let the new element "bubble up" by swapping until it finds the right position.



Heap operations

Insertion

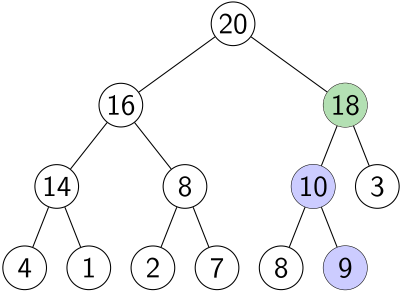
- Insert in the first available position $H[n+1]$.
- **Idea** Let the new element "bubble up" by swapping until it finds the right position.



Heap operations

Insertion

- Insert in the first available position $H[n+1]$.
- **Idea** Let the new element "bubble up" by swapping until it finds the right position.



- This process is called **HeapifyUp**.

Heap operations

HeapifyUp(H, i)

```
while i > 1 and H[i] > H[parent(i)] do
    swap H[i] and H[parent(i)]
    i = parent(i)
```

Insert($H[1..n]$, key)

```
// Assume the array H still has available
   space
H[n+1] = key
HeapifyUp(H, n+1)
n = n+1
```

Correctness?

Heap operations

HeapifyUp(H, i)

```
while i > 1 and H[i] > H[parent(i)] do
    swap H[i] and H[parent(i)]
    i = parent(i)
```

Insert($H[1..n]$, key)

```
// Assume the array H still has available
   space
H[n + 1] = key
HeapifyUp(H, n + 1)
n = n + 1
```

Correctness?

Time complexity?

HeapifyUp takes $O(h)$ time where h is the height of the heap.

Heap operations

HeapifyUp(H, i)

while $i > 1$ **and** $H[i] > H[parent(i)]$ **do**
    swap $H[i]$ and $H[parent(i)]$
    $i = parent(i)$

Insert($H[1..n]$, key)

```
// Assume the array H still has available
   space
```
$H[n+1] = key$
HeapifyUp(H, $n+1$)
$n = n + 1$

Correctness?

Time complexity?

HeapifyUp takes $O(h)$ time where h is the height of the heap.

$h = O(\log n)$.