Priority Queues and Heaps

Dr. Anirban Ghosh

School of Computing
University of North Florida

What is a Priority Queue?

Definition

It is an **abstract data type** that can store a collection of **comparable** items supporting the following operations:

1. `insert`(w): creates a new item e in the priority queue
2. `min`(): returns (but does not remove) the minimal item among the ones present in the priority queue; returns `null` if empty
3. `removeMin`(): removes and returns the minimal item among the ones present in the priority queue; returns `null` if empty
4. `size`(): returns the number of records in the priority queue
5. `isEmpty`(): returns a boolean indicating whether the priority queue is empty

Operation	Return value	PQ contents
insert(5)		$\{5\}$
insert(9)		$\{5,\ 9\}$
insert(3)		$\{3,\ 5,\ 9\}$
min$()$	3	$\{3,\ 5,\ 9\}$
removeMin$()$	3	$\{5,\ 9\}$
insert(7)		$\{5,\ 7,\ 9\}$
removeMin$()$	5	$\{7,\ 9\}$
removeMin$()$	7	$\{9\}$
removeMin$()$	9	$\{\ \}$
removeMin$()$	null	$\{\ \}$
isEmpty$()$	true	$\{\ \}$

```java
public interface PriorityQueue<E extends Comparable<E>> {
    int size(); // returns the number elements currently stored
    boolean isEmpty(); // checks if the priority queue is empty
    E min(); // returns the minimum element present in the priority queue
    E removeMin(); // returns and removes the minimum element present in the priority queue
    void insert(E e); // inserts a new element into the priority queue
}
```

- Applications in daily life: servicing customers having higher priorities, etc.
- Job scheduling in operating systems and computer networks
- Use as an auxiliary data structure for other algorithms: graph algorithms, heap-sort, etc.
- …

So how to implement PQs?

- Arrays and lists will result in linear runtimes
- One can use a RB-tree to implement a priority queue (logarithmic runtimes can be guaranteed)
- But a simpler and fast array-based (not space-wasting) data structure exists
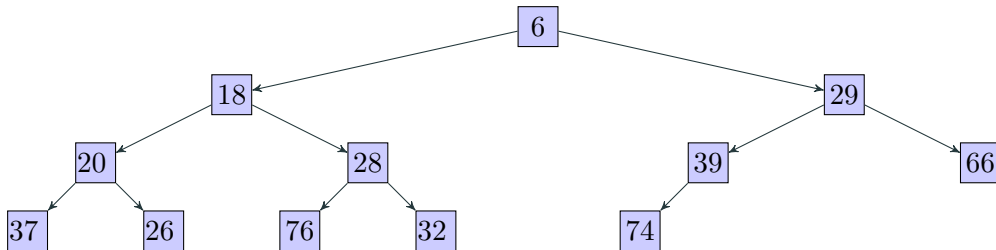- That data structure is **Binary Heap**
- Let us define it …
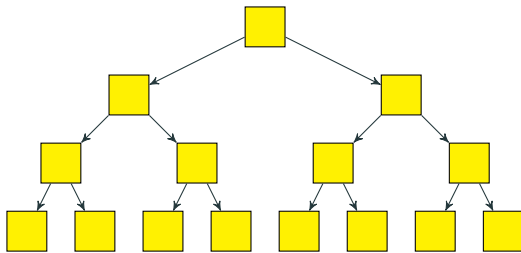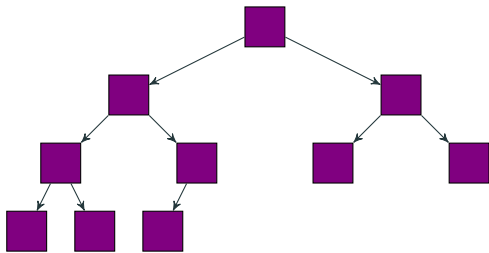
> **Definition**
>
> A binary tree T is **complete** if except the last level every other level is full (the last level may also be full but is not a requirement) and the nodes in the last level reside in the leftmost possible positions.

Definition

A binary tree T is a **heap** if,

1. the root of every subtree T' of T contains the smallest item in T'; this means the root of T always contains the smallest item in T

2. T is complete

A heap storing n items has height
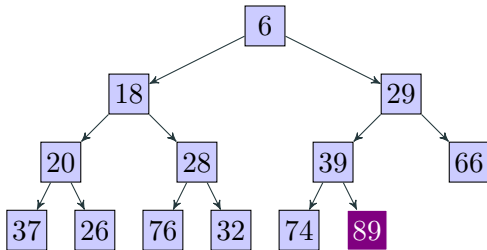$$h = \lfloor \log_2 n \rfloor = O(\log n)$$

*Binary heaps are different from binary search trees!*

Algorithm

1. Insert the new item x in the next position at the last level of the heap; if the last level is full **create a new level** with x
2. **while** x is not at the root and x is smaller than its parent
   Swap x with its parent, moving the item up the heap
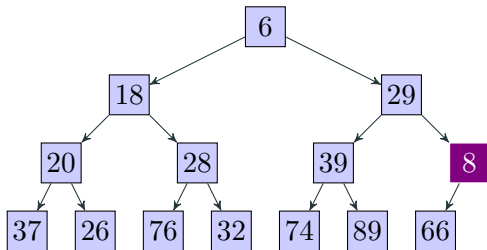
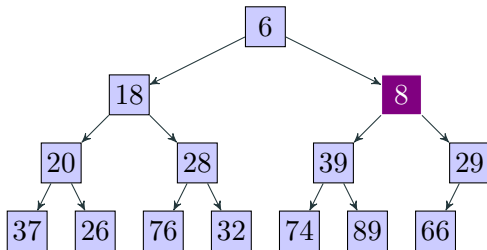Place 89 at the first available spot at the last level; 89 is smaller than its parent 39; insertion process terminates
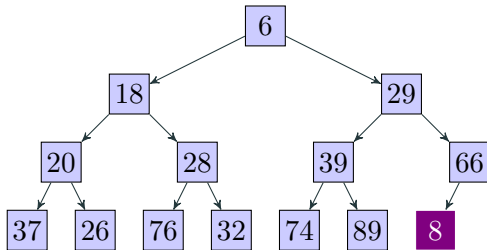
Place 8 at the first available spot at the last level; 8 is smaller than its parent 66; swap them

8 is smaller than its current parent 29; swap them

8 is greater than its current parent 6; stop

- Insert 25
- Insert -2

- One insertion takes $O(h)$ time, where h is the height of a heap, since, for fixing a heap, we need to climb up its whole height in the worst-case
- Since height of a heap is $O(\log n)$; one insertion takes $O(h) = O(\log n)$ time

Time taken to create the whole heap on n elements

Since a single insertion takes $O(\log n)$ time, creation of the whole heap on n elements takes $n \times O(\log n) = O(n \log n)$

Algorithm

1. Save the root element in a variable
2. Delete and bring the rightmost element in the last level of the heap to the root
3. **while** there exists a child of the root element that is smaller than it
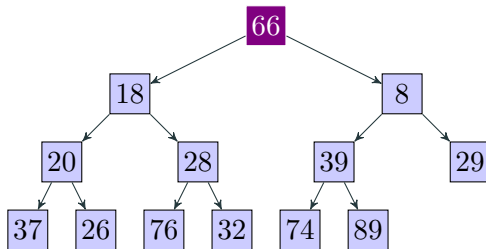   Swap the root element with its smaller child

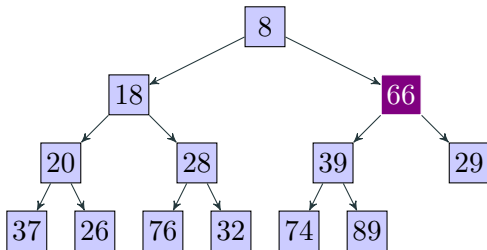☞ In min-heaps, only the current minimum element can be deleted/removed
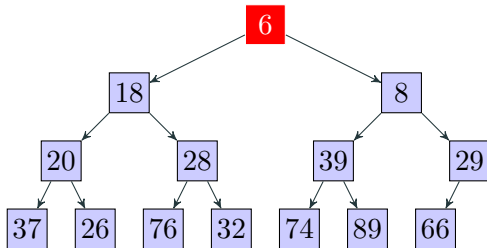
About to remove 6 (the minimum element)

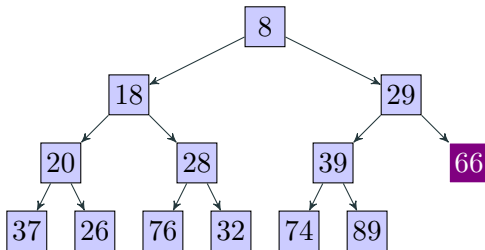Delete and bring the rightmost element 66 from the last level to the root

Choose the smaller child and swap 66 with it

Removal process terminates
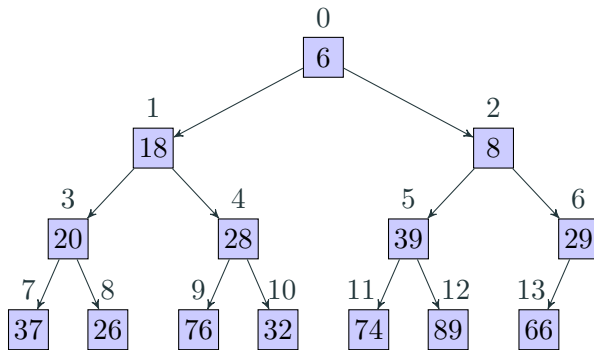
- One deletion takes $O(h)$ time, where h is the height of a heap, since, for fixing a heap, we need to climb down its whole height in the worst-case
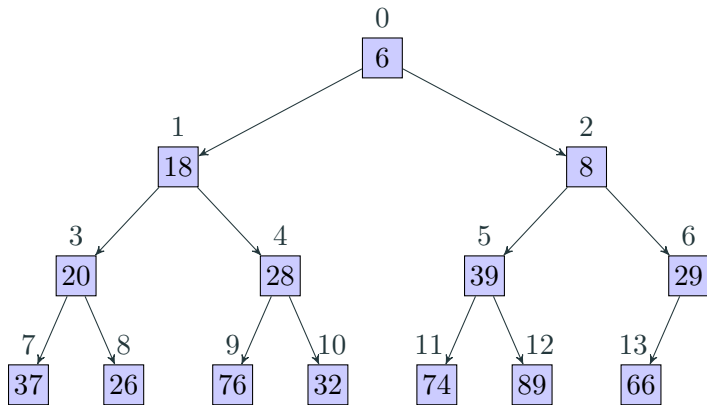- Since height of a heap is $O(\log n)$; one deletion takes $O(h) = O(\log n)$ time

How to represent heaps using an array?

Rules

- The root is at array index index 0
- If a node has index i, then its left child (if any) is at array index $2i + 1$ and its right child (if any) is at array index $2i + 2$
- Parent of the node at index i can be found at array index $\lfloor (i - 1)/2 \rfloor$

How to represent heaps?

Element	6	18	8	20	28	39	29	37	26	76	32	74	89	66
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

- The rightmost location of the last level (needed for the insert and delete operations) is precisely the last index of the array
- Since a heap can grow in size over time, an `ArrayList` can be used to maintain the heap
- Deleting the last element (required for heap removal operations) is easy when an `ArrayList` is used since no left shifting is needed for array compaction

See the class `MinHeap`

Algorithm

1. Insert the n records to be sorted into an empty heap H
2. Run `removeMin()` n times on H to get a sorted sequence of the input records

Time complexity

Step 1 takes $n \times O(\log n) = O(n \log n)$ time since every insertion takes $O(\log n)$ time

Step 2 takes $n \times O(\log n) = O(n \log n)$ time since every `removeMin()` takes $O(\log n)$ time

Total time taken: $O(n \log n) + O(n \log n) = 2 \times O(n \log n) = O(n \log n)$

See the class `HeapSort`

Further information

☞ The type of heaps we have considered so far are known as **min-heap**s
☞ Creation of a min-heap can be executed faster in $O(n)$ time using a different approach
☞ If the maximal element is always at the top, we call it a **max-heap**