heaps
priority queues

fall 2024 - data structures

Priority Queue

- queue
  - FIFO
  - Call center
  - A new call is added to back
  - Calls are taken from the front
- Sometimes FIFO may not be enough
  - Air-traffic control center
  - Need to add some factors now:
    - plane distance, time spent waiting, amount of fuel

PQ

- Priority queue
  - Collection of prioritized elements that allows arbitrary element insertion
    - Düz elemanlar harici artık öncelik var
  - Allows the removal of the element that has first priority
    - İlk önceliğe sahip olan eleman queue dan çıkarılacak – artık tam fifo değil.
- When an element is added, user gives a **key**
  - This key is the *importance*
  - Usually the minimal key is the most important

PQ ADT

- insert(k,v)
  - Creates an entry with key **k** and value **v**
- min()
  - Returns the minimum key.
- removeMin()
  - Removes and returns the entry with minimum key.
- size()
  - Number of entries in the queue
- isEmpty()
  - Returns a boolean

PQ

- Can have multiple entries with same key.
  - In that case, *min* and *removeMin* chooses arbitrarily

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Heaps

- A PQ is an ADT.
  - When we implement it, it becomes a data structure.
  - Can be implemented by using a List.
  - Can also be implemented by using a **binary heap**
    - binary heap implementation is faster
- Uses a binary tree structure.

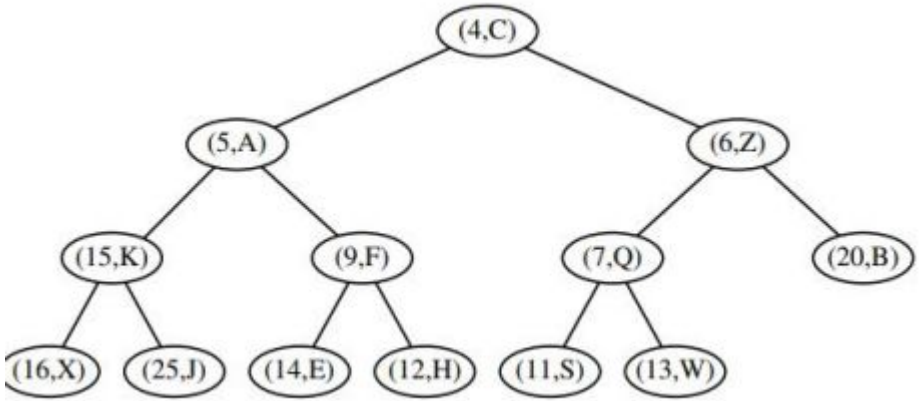
Relational property

- Heap-order property
  - In a heap, for every position *p* other than the root, the key stored at *p* is greater than or equal to the key stored at *p*'s parent.
  - Bir heap içinde, root haricindeki herhangi bir *p* noktasında saklanan veri, *p* nin parent'ında saklanandan ya büyük ya da eşittir.
- So, keys encountered on a path from root to a leaf are in **nondecreasing** order.
  - Minimal key is always stored at the root.

Structural property

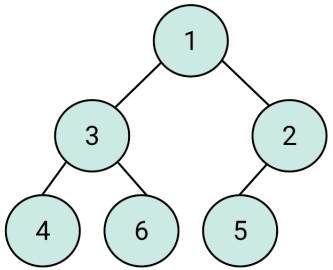
- We want the heap to have minimum height.
- Enforced by:
  - Heap T must be **complete** binary tree.

Completeness

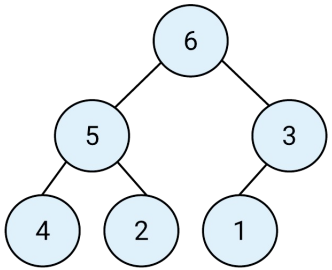


heaps

- Two types:
  - Min-heap
    - Value is smaller than or equal to its children
      - Same for all subtrees
    - Good for *priority queues*
  - Max-heap
    - Value is larger than or equal to its children
      - Same for all subtrees
    - Good for *heapsort*



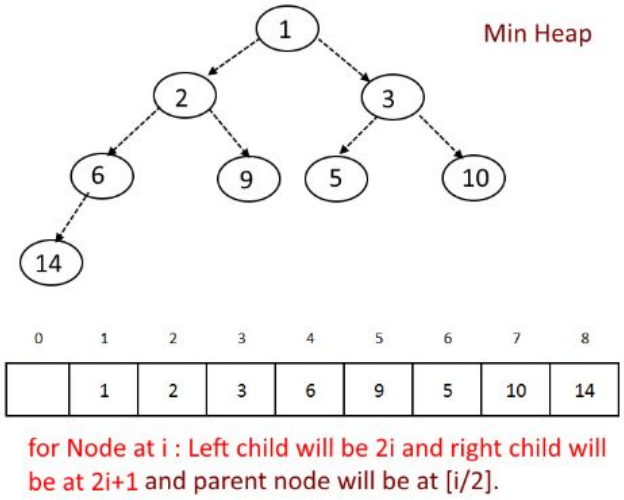
Min heap



Max Heap

representation

- A heap can be represented as an array.
- We can easily get the children and parent of a node by using simple arithmetics on the indices of the array.
  - We can start from 1 to make the math work.
  - We can also start from 0. Children will become 2i+1 and 2i+2
  - Parent is floor(i-1 / 2)
- Find the left and right child and parent of (2).



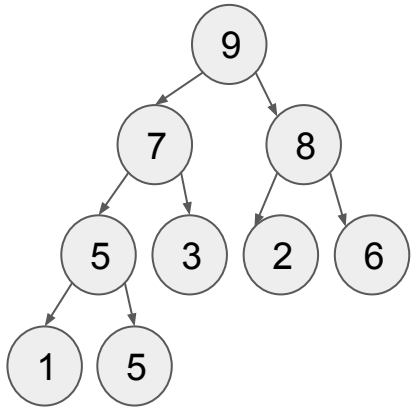
- In heaps, children are not ordered. Can be however.
- a node must be larger than or equal to its children.
  - for max heap
- a node must be smaller than or equal to its children
  - for min heap
- max heap
  - root is the largest
- min heap
  - root is the smallest

Creating a heap

- We will create a heap from a given array.
- Heap is not **unique**
  - You can have different heaps but still satisfying the **complete binary tree** and **min max heap** property.
- 10,20,15,30,40
  - Show step by step

Creating a new heap

- Since it is a complete binary tree, we know where to add.
  - We add it there, but then we check whether the tree is a max or min heap.
  - If not, we turn it into one by replacing, swapping elements.
- Let’s do: 5,1,2,6,3,8,9,7,5



insertion

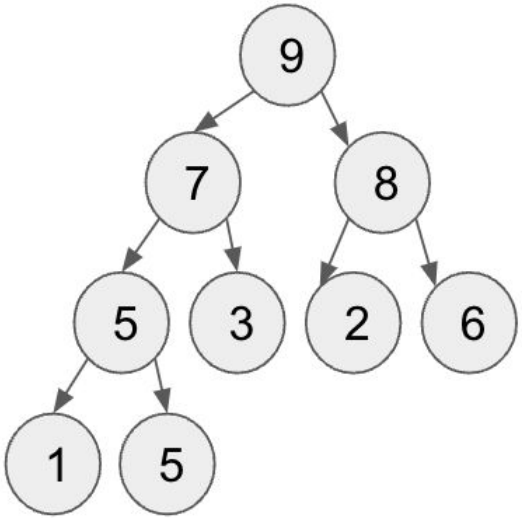
- Let’s say we have:
  - 50,30,20,15,10,8,16
    - This is already a max-heap
    - Just draw the tree.
- Later, we want to add 60 to it.
  - At first glance, we understand it should be at the root!
  - But we don’t add it like that.
    - Every operation we do, the result must be a complete binary tree + satisfy max/min property.
  - So we add it as the *next* element.
    - Starting from the leftmost.

complexity

- how many operations we did?
  - we did swaps.
  - Number of swaps are dependent on the **height** of the tree.
  - Height of the tree is **log(n)**.
- Therefore insertion is log(n).
  - Worst case.
  - Best case is O(1) -> You don’t swap anything.
- We added a leaf and moved upwards.
  - Bottom up heapify

deletion

- in heap, only logical deletion is done to the root.
  - If you want to delete some other elements, maybe you should not use a heap.
  - We get the maximum or minimum key (max heap, min heap)
  - So, we will delete the root!
- When we delete the root, we replace it with the last element and remove the last element.
- O(logn)

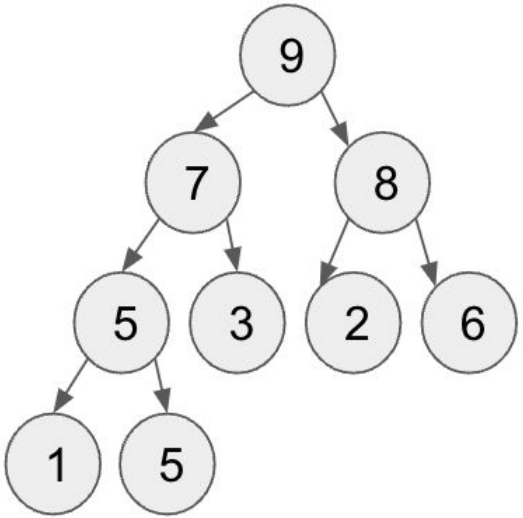


Deletion

- Delete the root
- 50,30,20,15,10,8,16

deletion

- We will delete 9.
  - Swap 9 with last element -> 5.
  - 5 becomes the root.
    - Complete binary tree but not max heap
  - We check the children of 5.
    - 7 and 8
    - 8 is larger, so we swap 8 with 5.
  - We do this until the max heap is satisfied.



Heapsort

- We use an array to implement heap.
  - Array is static. Size doesnt change.
  - But the heap size changes when deleted.
- We also saw that when we deleted items, it was sorted (reversely)
  - And each time we delete something from the heap, the array is becoming free.
  - What if we put the deleted element back into the array but in the back.
- So:
  - 50,30,20,15,10,8,16
  - We delete 50. Remember, we put 16 into its place.
  - 16,30,20,15,10,8,_ → Last place is free.
    - Why not put the deleted item there?
    - After satisfying the max heap array will be: 30,16,20,15,10,8,50.
    - Let's delete the max again. 30 will be replaced by 8. (Not 50, remember the size!)
    - Becomes 8,16,20,15,10,_,50. We heapify: 20,16,8,15,10,_, 50 → Put the 30 there:
      - 20 16 8 15 10 30 50
      - …

Heapsort

- So, we have some numbers.
  - We can sort them by creating a heap and then deleting the heap!
  - In-place sorting.
    - Space complexity O(1)

downward heapify

- How to make a heap out of a given tree.
  - We Convert a binary tree into a heap.

Imagine that this tree is given to us: 10, 20, 15, 12, 40, 25, 18

- We start from the last element.
  - if it satisfies max heap, we continue
  - 18, 25, 40, 12 are leaves → max heaps -> we continue
  - We get to 15. We look at the children. Is it a max heap? No.
    - We compare 25,18 -> we swap the larger with 15. swap(15,25)
  - We get to 20. We look at children. 12, 40 -> swap(40,20)
  - We get to 10. We look at children. 40,15 -> swap(40,10)
  - We check 10 again, because it still has children -> 12,20 → swap(20,10)

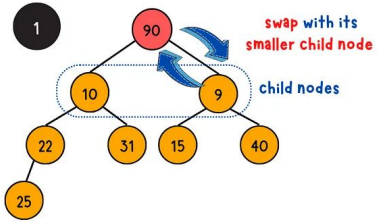
build_heap vs heapify

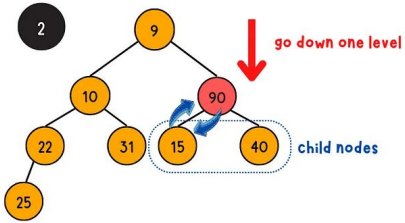
- When you have a tree and you try to make it into a heap, we call it heapify.
- When you build your heap from ground up, it is build_heap.
- Imagine:
  - We have a heap.
  - Deleted the root.
  - Not a heap anymore.
  - Making it into a heap : heapify.

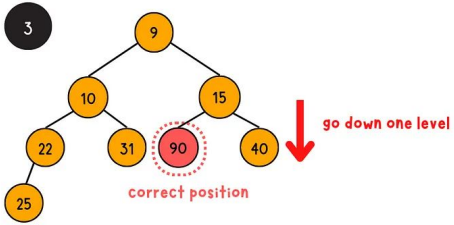
Sift Down vs. Sift Up

Sift Down

- Compare itself to both of its child nodes and move down in the heap until it's in the correct position.

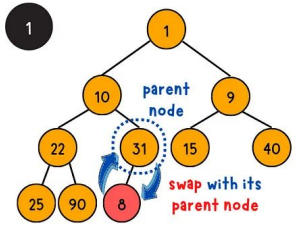


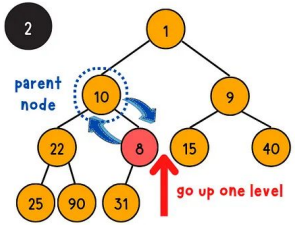


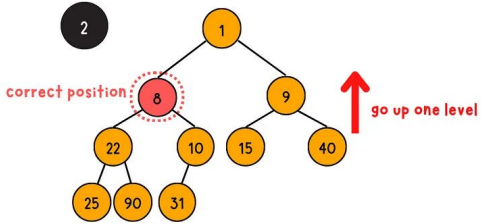


Sift Up

- Compare itself to its parent node and move up in the heap until it's in the correct position.







PQ

- Here, we have some items → we care about the key.
- 8, 6, 3, 10, 5, 4, 9
  - We can say the smallest number is the most important
  - We can say the largest number is the most important.
- In the first case,
  - We want to remove the minimum -> 3.
- In the second case;
  - We want to remove the maximum -> 10.
- So, we will use a **min heap** or **max heap**
- Again, we can just write functions and methods and store numbers in an array.
  - complexity will be o(n)
  - we need to find the element, we need to shift elements
  - But if we implement PQ as HEAP;
    - Inserting and removing is O(logn) -> faster.

PQ

- So,
- For a given array:
  - First you need to create a heap (min or max depending on what is the most important)
  - And you may need to get the most important element and give the new heap.
- All done by using the things we learned

good to read / watch

- https://yuminlee2.medium.com/golang-heap-data-structure-45760f9562dc
- https://www.youtube.com/watch?v=HqPJF2L5h9U
-