Algorithms and Data Structures

Heaps and Priority Queues

dr Szymon Murawski

Comarch SA

April 12, 2019

Table of contents I

1. Heaps
    - Introduction
    - Maintaining heap property
    - Building a heap
2. Heapsort
3. Priority queues
4. Molecular dynamics simulation of hard spheres

Course plan

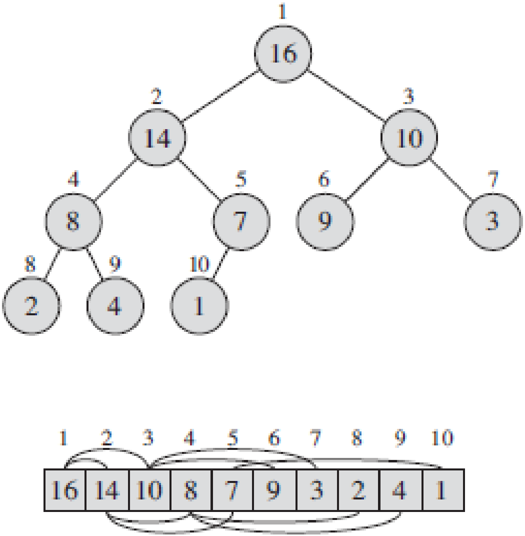
1. Heaps
    - Introduction
    - Maintaining heap property
    - Building a heap
2. Heapsort
3. Priority queues
4. Molecular dynamics simulation of hard spheres

Course plan

1. Heaps
    - Introduction
    - Maintaining heap property
    - Building a heap
2. Heapsort
3. Priority queues
4. Molecular dynamics simulation of hard spheres

Heaps

- Heap is a data structure that is a special kind of binary tree, that satisfies **heap property**
- Binary tree used in heap is a complete tree, meaning that every level, except possibly the last is completely filled and in the last level all nodes are as far to the right as they can be
- Heap property can be one of the following:
    - **Max-heap** - If P is a parent node of C, then P.key $\geq$ C.key
    - **Min-heap** - If P is a parent node of C, then P.key $\leq$ C.key
- Heap is stored in memory as simple array - we can do that, because there are no "holes" in the complete binary tree
- It is the basis for heapsort algorithm
- Priority queues are often represented as heaps



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Array representation of binary heap

- Binary heap is typically represented as array
- Traversal method used to achieve array representation is Level Order - we go through every left from left to right
- Assume that array is $A[]$ and heap is H
- Properties:
  - `H.size <= A.length` - length of underlying array must be at least equal to size of the heap
  - `H.root = A[0]` Root element of heap is stored at the front of array
  - (H.getNode(i) = A[i]) ith node in heap is stored at ith index in array and has value of $A[i]$
  - `H.getParent(i) = A[(i-1)/2]` - parent of ith node is stored at array index $(i-1)/2$
  - `H.getLeftChild(i) = A[2i+1]` - left child of ith node is stored at array index $2i+1$
  - `H.getRightChild(i) = A[2i+2]` - right child of ith node is stored at array index $2i+2$

Course plan

1. Heaps
   - Introduction
   - Maintaining heap property
   - Building a heap
2. Heapsort
3. Priority queues
4. Molecular dynamics simulation of hard spheres

Maintaining heap property

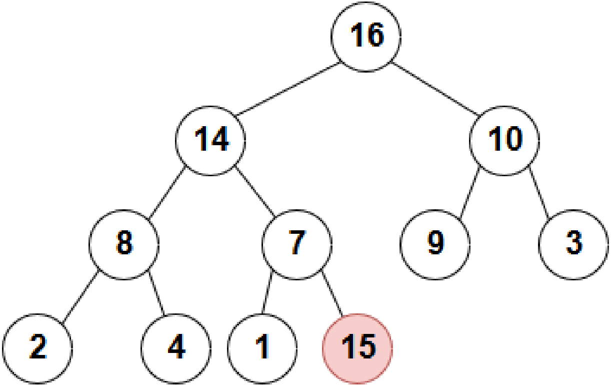
- Heap property says, that value stored in every node is no less than the value in the children of that node (or greater, depending on type of heap)
- Adding or removing an element from the heap can violate the heap property, so it must be immediately restored!
- Operations for adding and removing an element from the heap are usually called `push` and `pop`
- Heap data structure allows only the removal of top (root) element
- While inserting new element into the heap we cannot specify exact position of it - heap automatically places it in proper position

Heap push

- We insert new element at the end of the array
- Since heap property could be violated we restore it by doing the following:
    1. Compare value of new element to that of its parent. Terminate if the value is lesser (for max heap)
    2. If not then swap new element with its parent and repeat the process

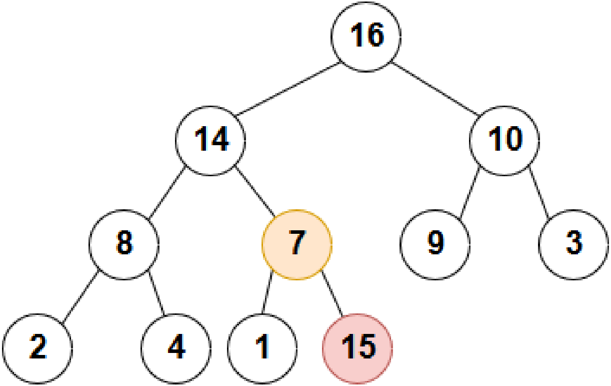
```
Push(elem)
  heap.size++
  A[heap.lastIndex] = elem
  SiftUp(heap.lastIndex)

SiftUp(elemIndex)
  if elemIndex == 0 // We are at the root
    return
  parentIndex = floor(elemIndex/2)
  if A[elemIndex] > A[parentIndex]
    swap(A[elemIndex], A[parentIndex])
    SiftUp(parentIndex)
```

Heap push example



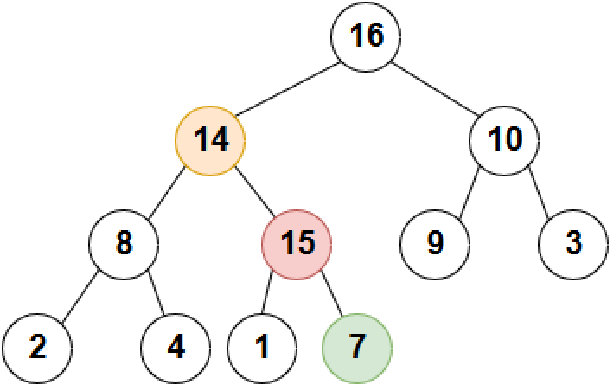
16	14	10	8	7	9	3	2	4	1	15

Heap push example



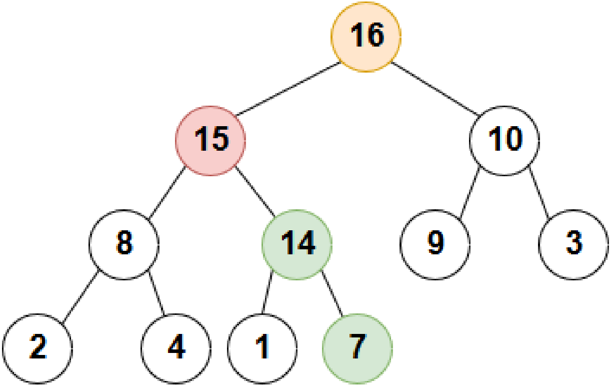
16	14	10	8	7	9	3	2	4	1	15

Heap push example



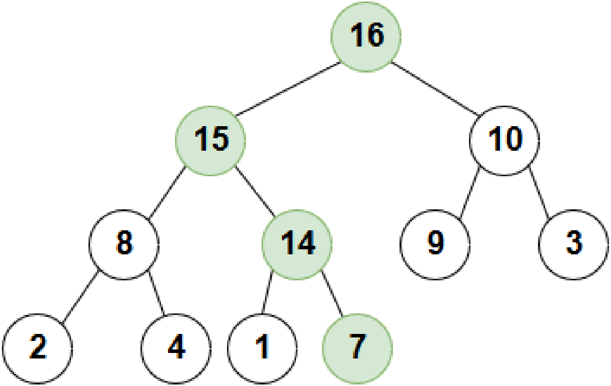
16	14	10	8	15	9	3	2	4	1	7

Heap push example



16	15	10	8	14	9	3	2	4	1	7

Heap push example



16	15	10	8	14	9	3	2	4	1	7

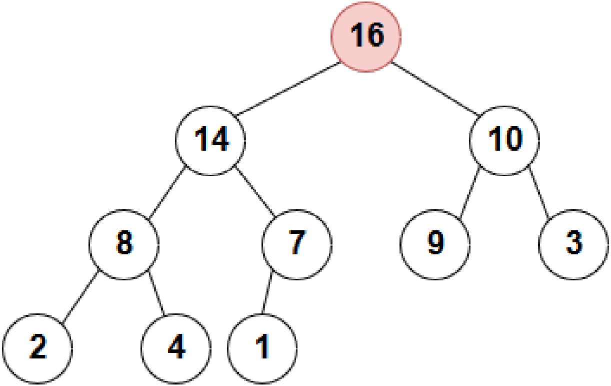
Heap pop

- We pop the root element
- We restore the heap structure and heap property by doing the following:
    - Place element that was at the end of the array at the root place
    - Terminate if the element has no children
    - Else check if heap property is violated, if so then swap the element with child of maximum value (for max heap)
    - Repeat until heap property is restored

Heap pop pseudocode

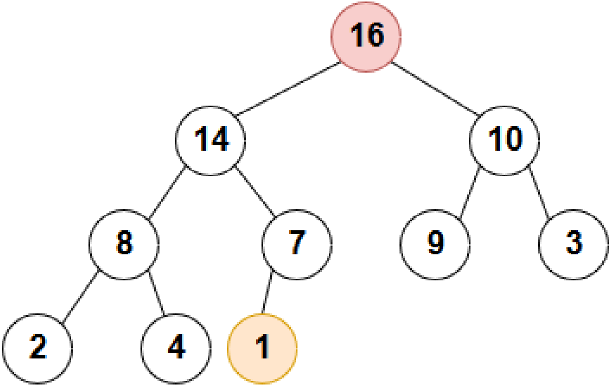
```
Pop()
  output = A[0]
  heap.size--
  A[0] = A[heap.lastIndex]
  SiftDown(0) // To restore heap property of new root
  return output

SiftDown(elemIndex)
  leftChildIndex = heap.getLeftChildIndex(elemIndex)
  rightChildIndex = heap.getRightChildIndex(elemIndex)
  return if leftChildIndex = NULL and rightChildIndex = NULL
  maxIndex = A[leftChildIndex] >= A[rightChildIndex] ?
    leftChildIndex : rightChildIndex
  if (A[elemIndex] < A[maxIndex])
    swap(A[elemIndex], A[maxIndex])
    siftDown(maxIndex)
```

Heap pop example



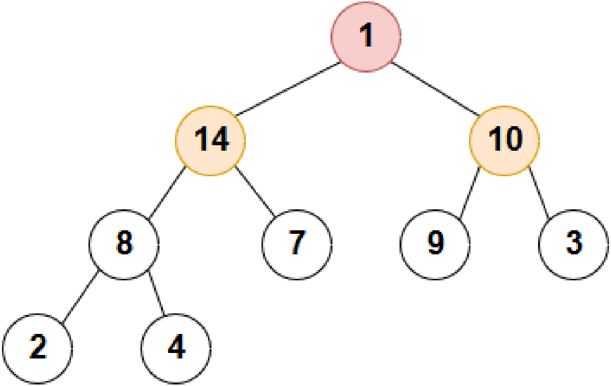
16	14	10	8	7	9	3	2	4	1

Heap pop example



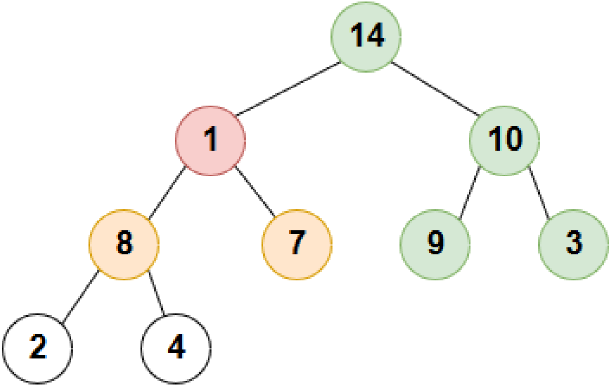
16	14	10	8	7	9	3	2	4	1

Heap pop example



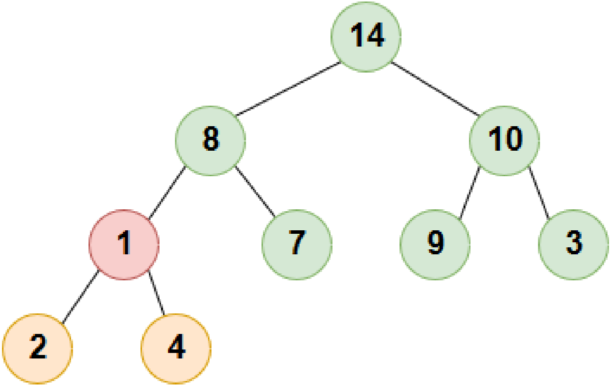
1	14	10	8	7	9	3	2	4

Heap pop example



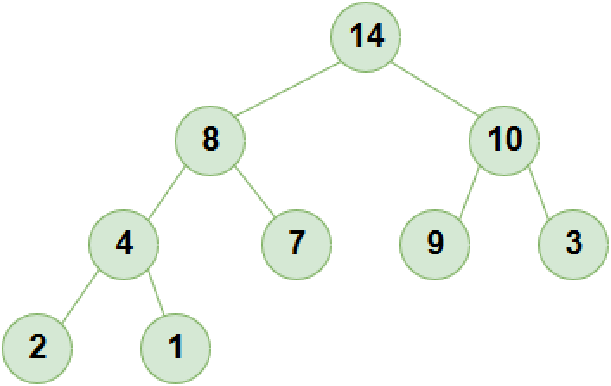
14	1	10	8	7	9	3	2	4

Heap pop example



14	8	10	1	7	9	3	2	4

Heap pop example



14	8	10	4	7	9	3	2	1

Heap pop and push analysis

- In pop and push the time consuming operation is siftUp or siftDown
- In pessimistic case both siftUp and siftDown methods would require to move an element to the other side of heap
- In other terms element would visit each level of heap
- If as a dominant operation we take comparison between elements, then siftUp does one comparison per level and siftDown two comparisons (comparing value of two children and of one children and original element)
- Complete tree with n nodes has $lg\ n$ levels
- Thus complexity of both pop and push is $O(lg\ n)$ (we ommit constant 2 for pop operation)
- Even though adding and removing an element to/from heap is not as easy as in lists, both operations are fairly fast!

Course plan

1. Heaps
   - Introduction
   - Maintaining heap property
   - Building a heap
2. Heapsort
3. Priority queues
4. Molecular dynamics simulation of hard spheres

Building a heap

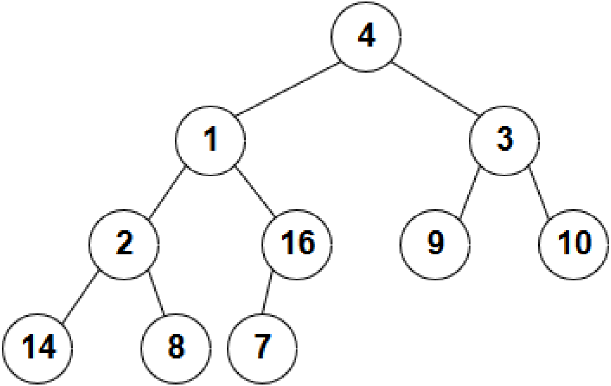
- Simple method of building a heap is to start from an empty array and sequentially adding new elements to at the end and calling siftUp procedure
- Complexity of this method is $O(nlg\ n)$, since we need to call siftUp method with complexity $O(lg\ n)$ once for each of n elements
- It is however possible to build a heap in linear time, using Floyd's algorithm
- In this approach we start with all the elements inserted into array in any order, that array represents complete binary tree
- Next step is to transform this binary tree into heap, by restoring (or rather introducing) heap property.
- To do that we could call siftUp for each element, but that would arrive at previous $O(nlg\ n)$ result
- Instead we call siftDown, which let's us obtain $O(n)$ algorithm

Floyd's algorithm

- Start with all elements in array in arbitrary order - they represent a complete binary tree
- Restore heap property by calling SiftDown on all internal nodes, skipping leaves

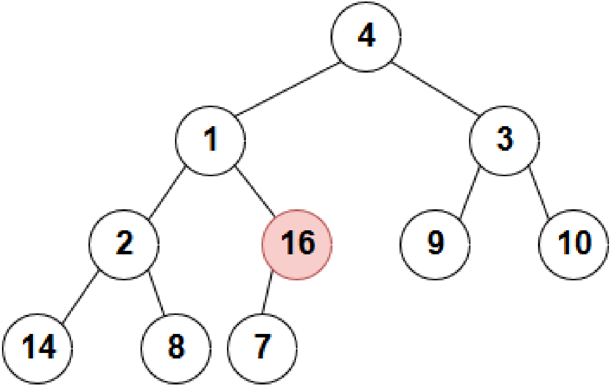
```
BuildHeap(A)
  heap.size = A.length
  foreach index in (heap.size/2..0)
    siftDown(index)
```

Building a max heap



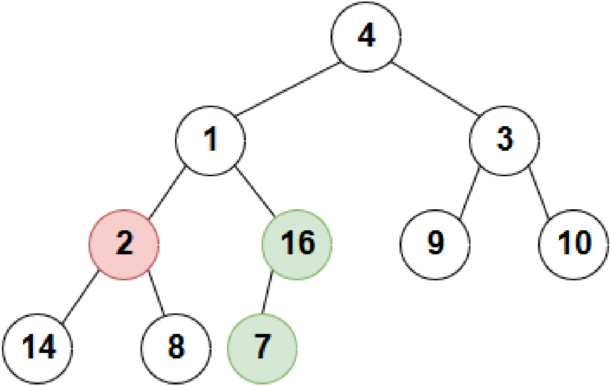
4	1	3	2	16	9	10	14	8	7

Building a max heap



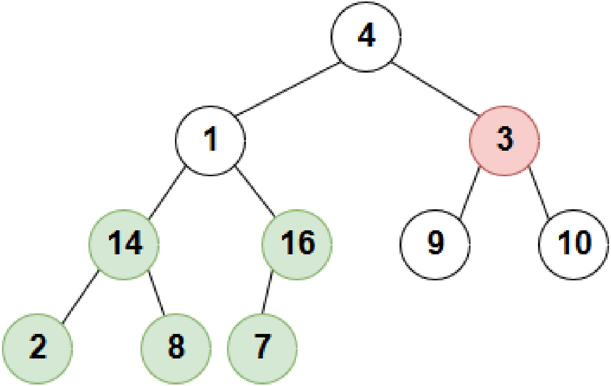
4	1	3	2	16	9	10	14	8	7

Building a max heap



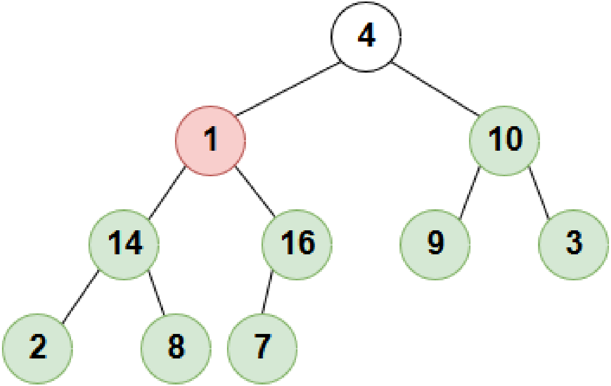
4	1	3	2	16	9	10	14	8	7

Building a max heap



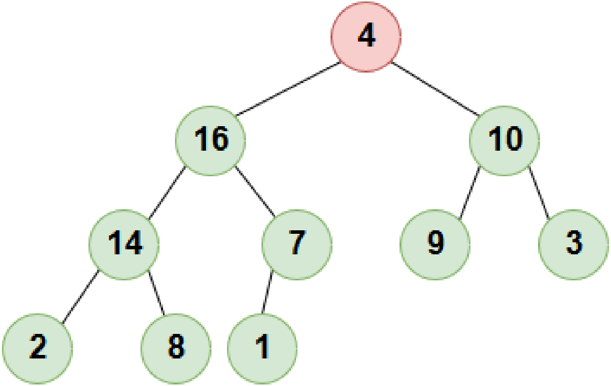
4	1	3	14	16	9	10	2	8	7

Building a max heap



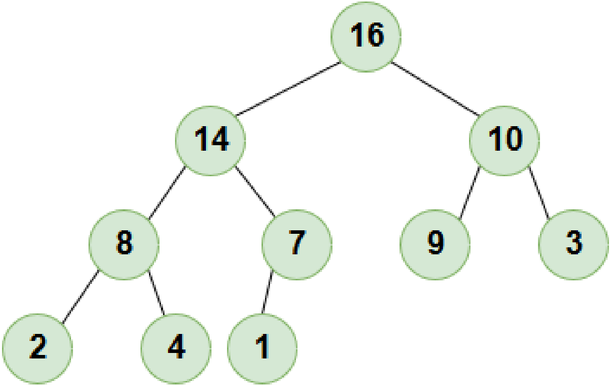
4	1	10	14	16	9	3	2	8	7

Building a max heap



4	16	10	14	7	9	3	2	8	1

Building a max heap



16	14	10	8	7	9	3	2	4	1

Floyd's algorithm analysis

- With unheaped array we can either start at the first element and for all elements call SiftUp, or start at the end of the array and call SiftDown on each element
- As we learned previously both SiftUp and SiftDown methods have the same complexity of $O(lg\ n)$
- But let's calculate how many maximum moves we would need to do for both approaches:
  - SiftUp: $(h * n/2) + ((h-1) * n/4) + ((h-2) * n/8) + \cdots + (0 * 1)$
  - SiftDown $(0 * n/2) + (1 * n/4) + (2 * n/8) + \cdots + (h * 1)$
- With SiftUp approach in worst case we need to move all leaves ($n/2$ elements) to the top of the heap
- Contrary with SiftDown all leaves are already at the bottom of the heap, so we only need to move root (one element) all the way down!
- Sum of the first series is $O(nlg\ n)$, but sum of the second (using Taylor series approach) is $O(n)$

Course plan

1. Heaps
   - Introduction
   - Maintaining heap property
   - Building a heap
2. Heapsort
3. Priority queues
4. Molecular dynamics simulation of hard spheres

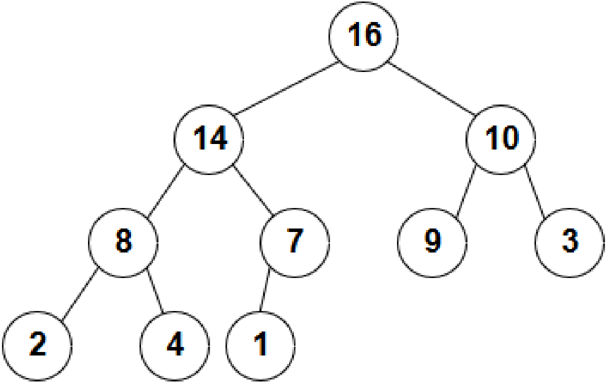
Heapsort algorithm

- Heapsort algorithm uses heap as underlying data structure for sorting problem.
- Heapsort (and heap data structure) was invented in 1964 by J.W.J. Williams, in the same year R.W. Floyd published an improved version of the algorithm.
- Algorithm consists of two steps: first we create a new max-heap for given array, then we construct sorted array by repeatedly calling heap.pop() and placing the output at the end of the array.
- It can be though of as an improved selection sort, since we are also dividing the array into sorted and unsorted part. Instead of finding minimum value in linear time we use heap to find it in $O(lg\ n)$ time however.
- There are many variations of this algorithm, depending on problem and computer architecture it can even beat quicksort in average case!

Heapsort pseudocode

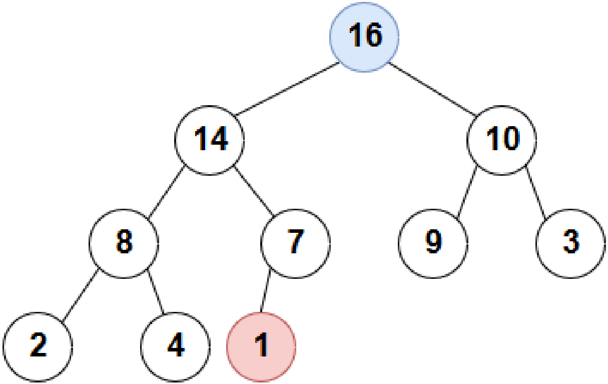
```
Heapsort(A)
  heap = createMaxHeap(A)
  while(heap.Size > 0)
    currentMax = heap.pop()
    A[heapSize] = currentMax
```

Heapsort example



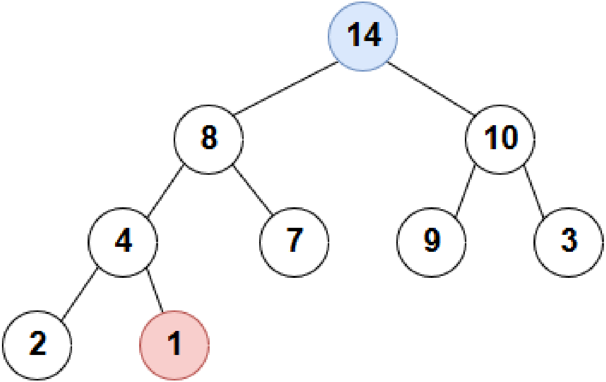
16	14	10	8	7	9	3	2	4	1

Heapsort example



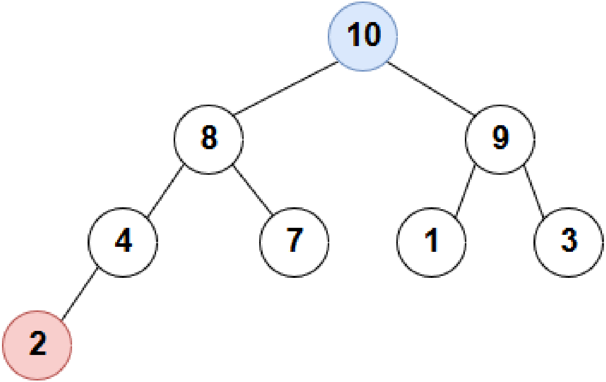
16	14	10	8	7	9	3	2	4	1

Heapsort example



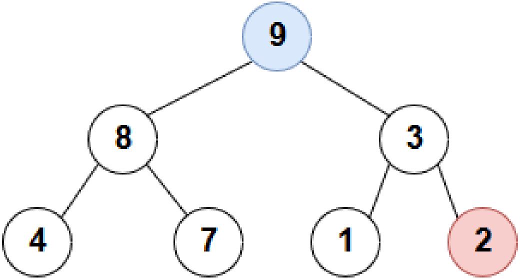
14	8	10	4	7	9	3	2	1	16

Heapsort example



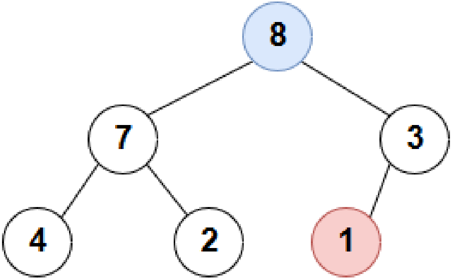
10	8	9	4	7	1	3	2	14	16

Heapsort example



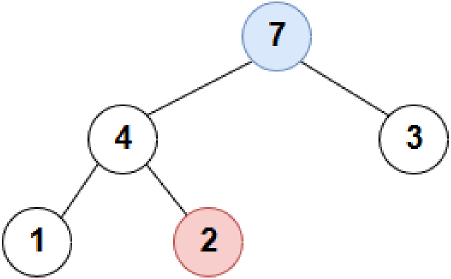
9	8	3	4	7	1	2	10	14	16

Heapsort example



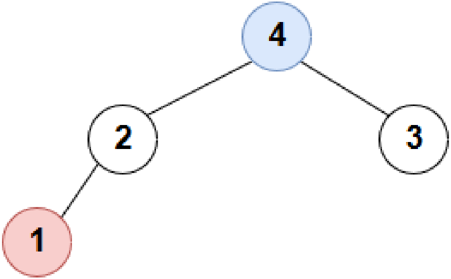
8	7	3	4	2	1	9	10	14	16

Heapsort example



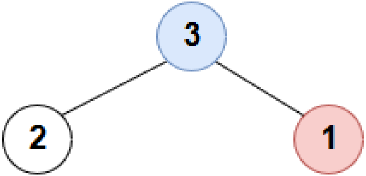
7	4	3	1	2	8	9	10	14	16

Heapsort example



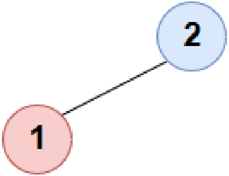
4	2	3	1	7	8	9	10	14	16

Heapsort example



3	2	1	4	7	8	9	10	14	16

Heapsort example

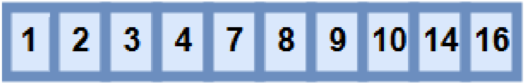


2	1	3	4	7	8	9	10	14	16

Heapsort example

1	2	3	4	7	8	9	10	14	16

Heapsort example



1	2	3	4	7	8	9	10	14	16

Heapsort analysis

- Creation of a heap is $O(n)$ complexity operation
- `heap.pop()` on the other hand is $O(lg\ n)$ operation, due to underlying siftUp procedure
- Since we are calling `heap.pop()` for every element in the array, total complexity is $O(nlg\ n + n) = O(nlg\ n)$ in the worst case!
- We are also not using any addition memory - heapsort is in-place sorting algorithm
- It is although unstable
- Heapsort is the best algorithm for sorting if worst-case scenario is our main concern. This is a case for security and embedded systems, so for example linux kernel uses heapsort.
- In average case it is placed second, just behind quicksort, due to the fact that quicksort makes greater use of cache and branch prediction than
- Still, some variations of heapsort achieve complexity very close to the theoretical lower bound for sorting!

Choosing the right sorting algorithm once again

- Choose insertion sort if the input is small, or elements are roughly sorted, or you need very simple code
- Choose heapsort if worst-case scenario is your main concern
- Choose quicksort in general case

Course plan

1. Heaps
   - Introduction
   - Maintaining heap property
   - Building a heap
2. Heapsort
3. Priority queues
4. Molecular dynamics simulation of hard spheres

Priority queues

- A priority queue is a data structure, that maintains set of elements in such a way, that always the element with largest (or smallest) key is at presented at the front
- It can be thought of as an extensions of queue data structure
- There are two kind of priority queues, max and min-priority queues
- Priority queue can be implemented with array as underlying data structure, that would result in $O(1)$ complexity for inserting new element, but $O(n)$ for retrieving maximum
- Optimal implementation uses heap as data structure, which results in $O(lg\ n)$ complexity for both adding and removing an element, and $O(1)$ complexity for just peeking at maximum element

Priority queue applications

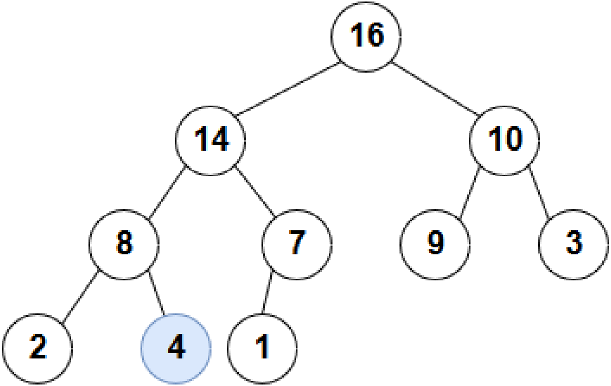
- Bandwidth management - routers use priority queues to determine which traffic should be handled first. VoIP and IPTV for example should be prioritized, while simple html requests can wait a bit
- Event based simulations - instead of using very discreet time partition in simulation, we can introduce event based handling of processes using priority queues (more on that later)
- Several graph algorithms use priority queues (Dijkstra shortes path algorithm, Prim's minimum spanning tree algorithm)
- $A*$ pathfinding algorithm (used widely in video games) use priority queue to store information about nodes to visit
- Huffman coding algorithm can be implemented using priority queue
- Operating systems use priority queues for load balancing and interrupt handling
- Priority queues are also used in spam filtering algorithms (Bayesian spam filter)

Increase key

- In addition to standard heap interface, effective priority queues implement two additional methods
- `IncreaseKey(i, newValue)` is used to increase key (priority) of a given element
- `DecreaseKey(i, newValue)` is used to decrease key (priority) of a given element

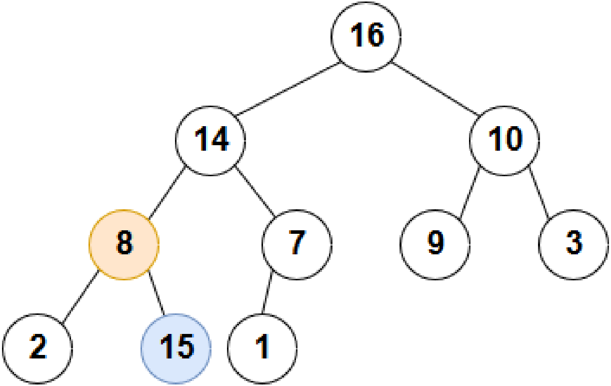
```
IncreaseKey(i, newValue)
  if key < A[i]
    return error ”New key is smaller than current key”
  A[i] = key
  while i > 1 and A[Parent(i)] < A[i]
    swap(A[i], A[Parent(i)])
    i = Parent(i)
```

Increase key example



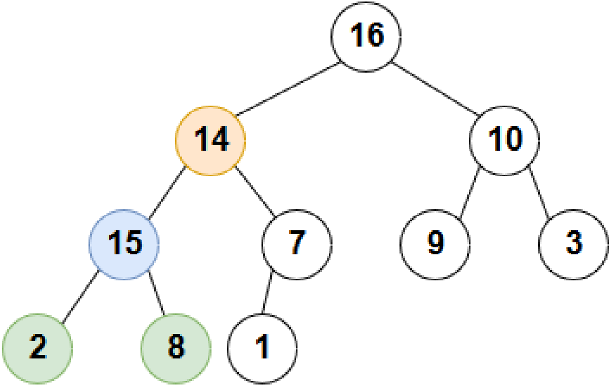
16	14	10	8	7	9	3	2	4	1

Increase key example



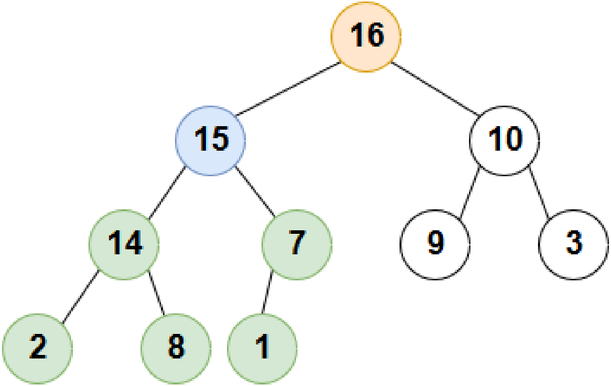
16	14	10	8	7	9	3	2	15	1

Increase key example



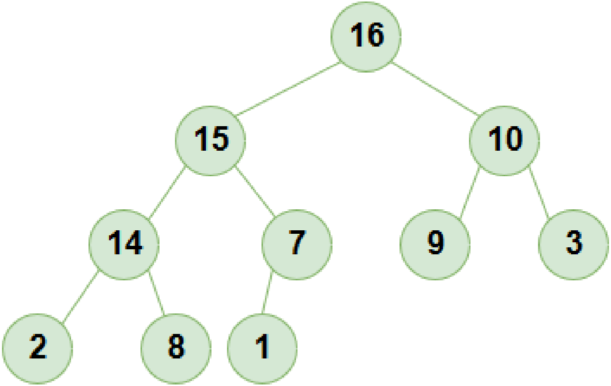
16	14	10	15	7	9	3	2	8	1

Increase key example



16	15	10	14	7	9	3	2	8	1

Increase key example



16	15	10	14	7	9	3	2	8	1

Course plan

1. Heaps
   - Introduction
   - Maintaining heap property
   - Building a heap
2. Heapsort
3. Priority queues
4. Molecular dynamics simulation of hard spheres

Molecular dynamics simulation of hard spheres

Goal

Simulate the motion of N moving particles in a box that behave according to the laws of elastic collision.

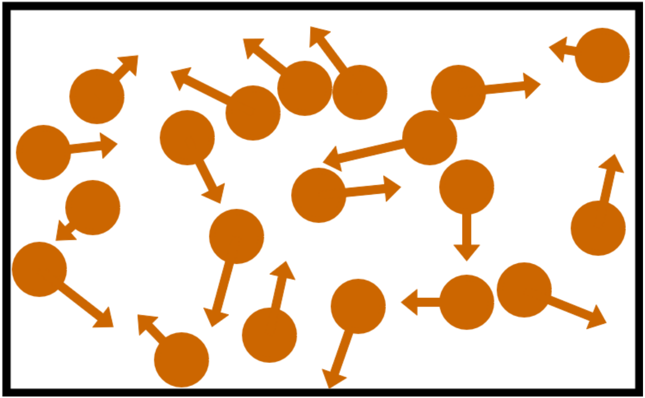
- `https://www.youtube.com/watch?v=X_BUr_cnAn4`
- `https://www.youtube.com/watch?v=Pj-1Y4iOmiw`

Hard sphere model

- Each particle is a sphere with known position, velocity, mass and radius
- Particles are infinitely hard - they cannot be squeezed (hardcore!)
- Moving particles interact via perfect elastic collisions with each other and with fixed walls
- There are no other forces in the system
- Between collisions particles move with constant velocity in a straight line
- We will be using a simplification of this model in which we operate only on 2D plane - hard disks model.

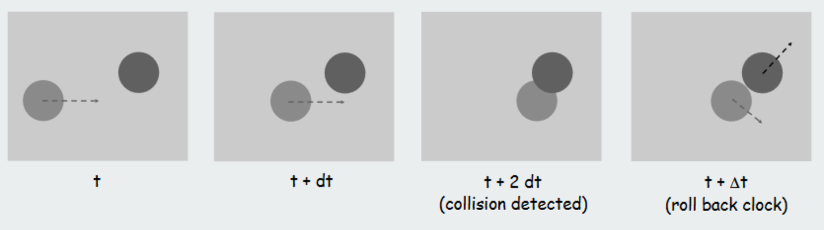
- Hard sphere model is of great significance in physics
- It can be used to show relation between macroscopic observables (temperature, pressure) to microscopic dynamics (motion of individual particles)
- Maxwell and Boltzmann used the model to derive the distribution of speeds of interacting molecules as a function of temperature
- Einstein used it to explain the Brownian motion of pollen grains immersed in

Hard sphere model illustration



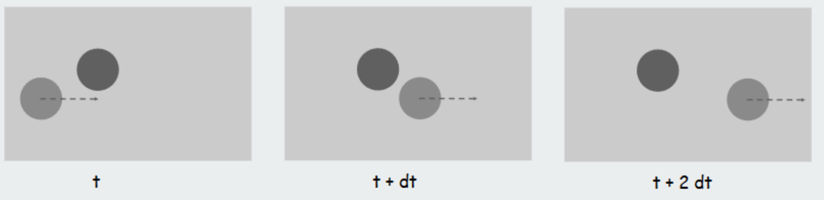
Time driven simulation

- We can easily simulate a system like this by means of time-driven simulation
- It boils down to choosing an arbitrary small Δt, and advancing time in discreet manner
- After each step check if any two particles overlap. If they do reverse time to the time of collision, update velocities according to laws of elastic motion and continue with simulation



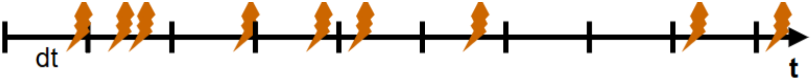
Problems with time driven simulation

- There are two main problems with time-driven simulation like this
  1. At each step we need to perform $O(n^2)$ collision checks
  2. If the particles move too fast, or Δt is too big, we can easily miss collision
- We can try to overcome second problem by decreasing Δt, but it would mean that simulation would advance very little and would take a lot of (real) time to finish!



Event-driven simulation

- According to the model particles travel with constant speed in a straight line between collisions. That part is boring
- The fun begins when two particles collide - we need to make some calculations what the new velocities will be for each one
- We can then approach the problem of simulating the model by only keeping track of collisions that will occur in the future and at each step advance time to the moment of the next collision



Basic physics review

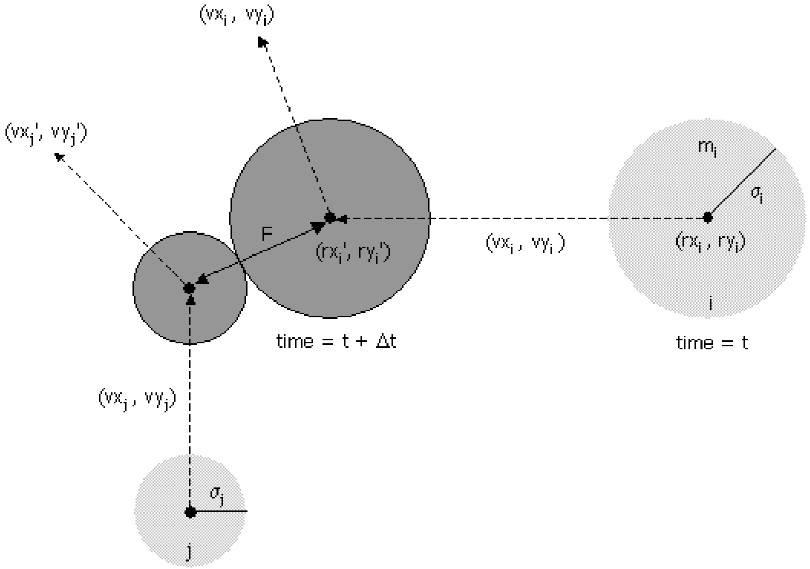
- Particles travel in straight line with constant speed $\overrightarrow{v}$, meaning, that if they start at position $\overrightarrow{r}$, after Δt time they will be at position:

$$\overrightarrow{r}(t+\Delta t) = \overrightarrow{r}(t) + \overrightarrow{v}\Delta t$$

- Assume that we have identical particles, each being a disk with radius σ
- Two particles are colliding if the distance between them is equal to $d = \sigma_1 + \sigma_2 = 2\sigma$
- We can recall the equation for distance between two points on a plane: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y1)^2}$
- Combining all of the above we arrive at the following equation for Δt until next collision occurs:

$$\overrightarrow{v}^2_{12}(\Delta t)^2 + 2(\overrightarrow{v}_{12}\overrightarrow{r}_{12})\Delta t + (\overrightarrow{r}^2_{12} - \sigma^2) = 0$$

Collision between particles



Cases for collision

$$\vec{v}_{12}^2(\Delta t)^2 + 2(\vec{v}_{12}\vec{r}_{12})\Delta t + (\vec{r}_{12}^2 - \sigma^2) = 0$$

- We can solve the quadratic equation for Δt, obtaining the following time until next collision:

$$\Delta t = \begin{cases} \infty, & \text{if } (\vec{v}_{12}\vec{r}_{12}) \geq 0 \\ \infty, & \text{if } \Delta < 0 \\ -\frac{(\vec{v}_{12}\vec{r}_{12})+\sqrt{\Delta}}{\vec{v}_{12}^2}, & \text{otherwise} \end{cases}$$

- Δ is our quadratic equation discriminant ($\Delta = b^2 - 4ac$)
- First case is for particles that are traveling away from each other
- Second case is for particles that are traveling towards each other, but are not near enough
- Third case is the one we are interested in!

Event-driven simulation

- We store records of collision times in priority queue, with minimum time in the front of the queue
- We initialize the program by inserting into the queue all potential particle and wall collisions
- Simulation loop consists of the following steps:
    1. Delete the impending event, i.e., the one with the minimum priority t.
    2. If the event corresponds to an invalidated collision, discard it. The event is invalid if one of the particles has participated in a collision since the event was inserted onto the priority queue.
    3. If the event corresponds to a physical collision between particles i and j:
        1. Advance all particles to time t along a straight line trajectory.
        2. Update the velocities of the two colliding particles i and j according to the laws of elastic collision.
        3. Determine all future collisions that would occur involving either i or j, assuming all particles move in straight line trajectories from time t onwards. Insert these events onto the priority queue.
    4. If the event corresponds to a physical collision between particles i and a wall, do the analogous thing for particle i.

Implementation details

Class Particle

- `double x, y` - position of particle on x,y plane
- `v_x, v_y` - x and y velocities of particle
- `int collisionCount` - total number of times the particle was involved in a collision
- `double timeUntilWallCollision()` - returns time at which next collision with a wall will occur (in reality we should distinguish between vertical and horizontal walls)
- `double timeUntilParticleCollision(Particle b)` - returns time at which collision with another particle `b` will occur
- `void bounceOffWall()` - updates velocity after bouncing off the wall
- `void bounceOffParticle(Particle b)` - updates velocity after bouncing off another particle

Implementation details 2

Class Event

- `Event(double t, Particle a, Particle b)` - constructor of new event object, that will happen at time `t` and will involve particles `a` and `b`
- `double time` - time when the event will happen
- `int collision_a, collisions_b` - total number of collisions of both particles at the time of creating this event instance
- `Particle a, b` - particles taking part in this event
- `bool isValid()` - returns true if the event is still valid, meaning that since creating this event both particles did not take part in another collision. We can easily check that by comparing collisions counts stored in event and in particle object
- We can make use of interfaces or inheritance to distinguish particle and wall collision events

Implementation details 3

class PriorityQueue

- Should store elements of type `Event`
- In front of queue should be events with smallest time of collision

class Simulation

- `PriorityQueue<Event> MinPQ` - priority queue holding collision events
- `List<Particle> Particles` - list of all particles in our system
- `void Predict(Particle a)` - predicts all possible collisions between this particle and every other and adds them to the queue
- `void Initialize()` - predicts all possible collisions between all particles and adds them to priority queue
- `void Advance(double t)` - advances time by `t`

Analysis

- It is more complicated algorithm, but it produces best results. We will never miss any collision that will occur
- Initialization takes n^2 calculations - we need to calculate potential collisions between each particles
- At every step of simulation we are doing $lg\ n$ calculations to fetch the next event and $2n$ calculations to predict new potential collisions. Inserting them into priority queue takes $2nlg\ n$