Priority Queues and Heaps

Bùi Tiến Lên

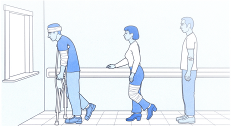
2021

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Priority queue applications

Event-driven simulation.	(customers in a line, colliding particles)
Numerical computation.	(reducing roundoff error)
Data compression.	(Huffman codes)
Graph searching.	(Dijkstra's algorithm, Prim's algorithm)
Number theory.	(sum of powers)
Artificial intelligence.	(A* search)
Statistics.	(maintain largest M values in a sequence)
Operating systems.	(load balancing, interrupt handling)
Discrete optimization.	(bin packing, scheduling)
Spam filtering.	(Bayesian spam filter)



Priority queue

Concept 1

A **priority queue** is a data structure for maintaining a set S of elements, each with an associated value called a **key**. There are two kinds of priority queues: **max-priority queues** and **min-priority queues**.

Max-priority queue supports the following operations.

- $\textsc{Insert}(S, x)$: insert the element x into the set S or $S \leftarrow S \cup \{x\}$.
- $\textsc{Max}(S)$: return the element of S with the largest key.
- $\textsc{Remove-Max}(S)$: remove and return the element of S with the largest key.
- $\textsc{Increase-Key}(S, x, \Delta k)$: increase the value of element x's key by the Δk.

Unordered and ordered array implementation

- A sequence of operations on a max-priority queue

operation	*argument*	*return value*	*size*	*contents (unordered)*							*contents (ordered)*						
insert	P		1	P							P						
insert	Q		2	P	Q						P	Q					
insert	E		3	P	Q	E					E	P	Q				
remove max		Q	2	P	E						E	P					
insert	X		3	P	E	X					E	P	X				
insert	A		4	P	E	X	A				A	E	P	X			
insert	M		5	P	E	X	A	M			A	E	M	P	X		
remove max		X	4	P	E	M	A				A	E	M	P			
insert	P		5	P	E	M	A	P			A	E	M	P	P		
insert	L		6	P	E	M	A	P	L		A	E	L	M	P	P	
insert	E		7	P	E	M	A	P	L	E	A	E	E	L	M	P	P
remove max		P	6	E	M	A	P	L	E		A	E	E	L	M	P	

Analysis

Challenge Implement **all** operations efficiently.

implementation	**insert**	**remove max**	**max**
unordered array	1	N	N
ordered array	N	1	1
goal			

Binary Heaps

- Insert Key
- Remove Max
- Increase Key

Introduction

Concept 2

- **Max heap**: A tree is heap-ordered if the key in each node is larger than or equal to the keys in all of that node's children (if any)
- **Min heap**: A tree is heap-ordered if the key in each node is smaller than or equal to the keys in all of that node's children (if any)

Theorem 1

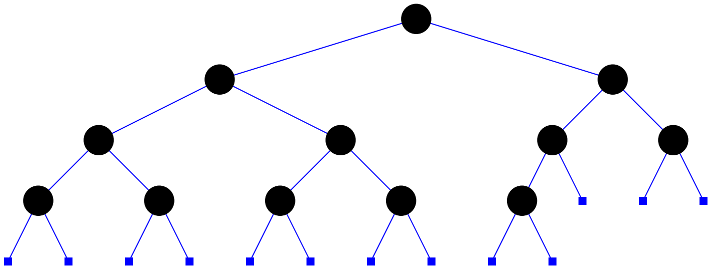
- *Max heap: No node in a heap-ordered tree has a key larger than the key at the root*
- *Min heap: No node in a heap-ordered tree has a key smaller than the key at the root*

Binary heap representations

Concept 3

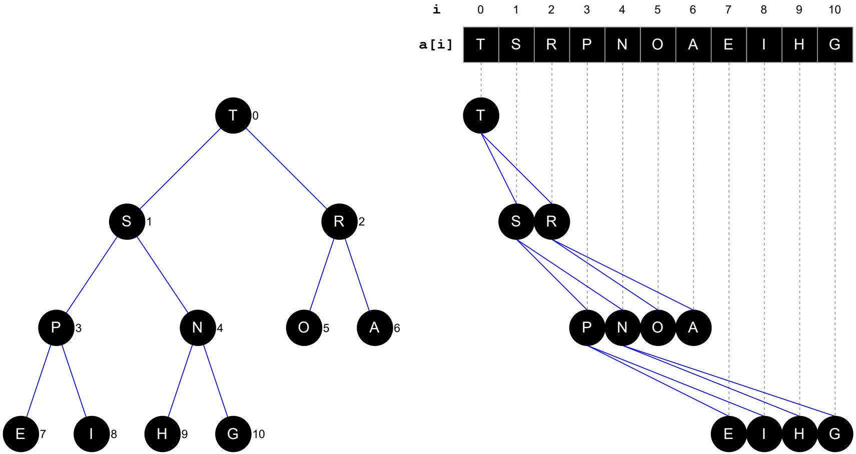
A **binary heap** data structure is a complete binary tree that can be represented by **an array object**.

- Complete binary tree is perfectly balanced, except for bottom level.
- Height of complete binary tree with N nodes is $\log_2(N+1)$.



Binary heap representations (cont.)

- Array representation of a max binary heap.



Binary heap properties

- Largest key is $a[0]$, which is root of max binary heap.
- Can use array indices to move through tree.

PARENT(i)
    return $\lfloor \frac{i-1}{2} \rfloor$

LEFTCHILD(i)
    return $2i + 1$

RIGHTCHILD(i)
    return $2i + 2$

Promotion in a heap

- **Scenario**. Child's key becomes larger key than its parent's key. (violation)
- To eliminate the violation:
  - Exchange key in child with key in parent.
  - Repeat until heap order restored.

```
UP-HEAP(a, i)
    while i > 0
        p ← PARENT(i)
        if a[i] > a[p]
            SWAP(a, i, p)
            i ← p
        else
            return
```

Insertion in a heap

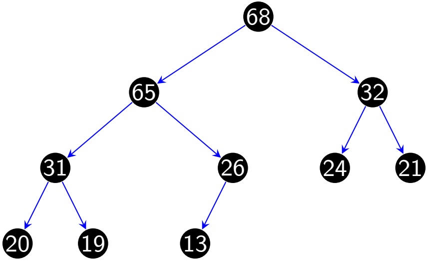
Insert. Add node at end, then percolate it up.

Cost. At most $\log_2 N + 1$ compares.

$\textsc{Insert}(a,\ k)$
    $n \leftarrow a.size$
    $a[n] \leftarrow k$
    $\textsc{Up-Heap}(a,\ n)$
    $a.size \leftarrow a.size + 1$

Example

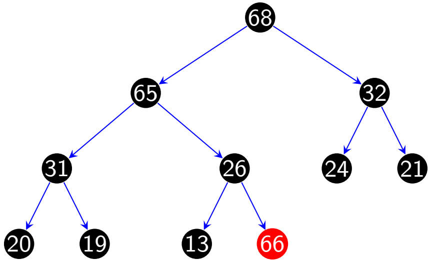
- Given a max binary heap



0	1	2	3	4	5	6	7	8	9	10	11
68	65	32	31	26	24	21	20	19	13		

Example (cont.)

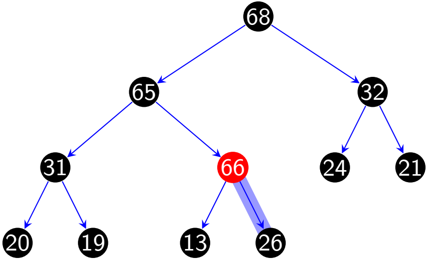
- Insert key 66 into the heap



0	1	2	3	4	5	6	7	8	9	10	11
68	65	32	31	26	24	21	20	19	13	66	

Example (cont.)

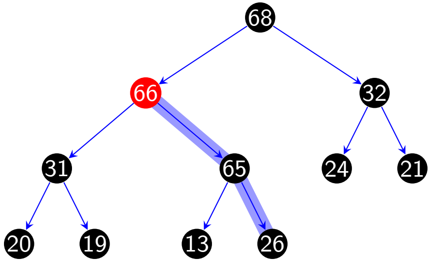
- Swap 66 and 26



0	1	2	3	4	5	6	7	8	9	10	11
68	65	32	31	66	24	21	20	19	13	26	

Example (cont.)

- Swap 66 and 65



0	1	2	3	4	5	6	7	8	9	10	11
68	66	32	31	65	24	21	20	19	13	26	

Demotion in a heap

- **Scenario**. Parent's key becomes smaller than one (or both) of its children's. (vilolation)
- To eliminate the violation:
  - Exchange key in parent with key in larger child.
  - Repeat until heap order restored.

```
Down-Heap(a, i)
    l ← LeftChild(i)
    r ← RightChild(i)
    largest ← i
    if l < a.size and a[l] > a[largest] then largest ← l
    if r < a.size and a[r] > a[largest] then largest ← r
    if largest ≠ i then
        Swap(a, i, largest)
        Down-Heap(a, largest)
```

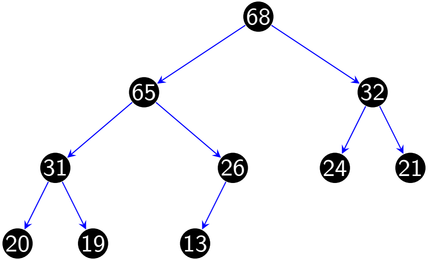
Delete the maximum in a heap

- **Delete max**. Exchange root with node at end, then sink it down.
- **Cost**. At most $2\log_2 N$ compares.

$\textsc{Remove-Max}(a)$
$\quad n \leftarrow a.size$
$\quad \textsc{Swap}(a,\ 0,\ n-1)$
$\quad \textsc{Down-Heap}(a,\ 0)$
$\quad a.size \leftarrow a.size - 1$
$\quad$ **return** $a[n-1]$ and delete $a[n-1]$

Example

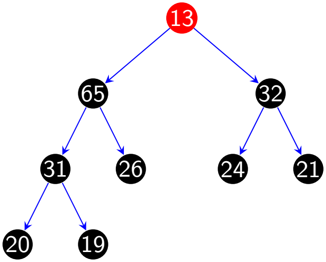
- Given a max binary heap



0	1	2	3	4	5	6	7	8	9	10	11
68	65	32	31	26	24	21	20	19	13		

Example (cont.)

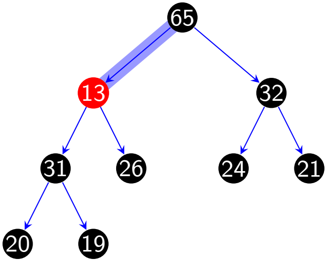
- Delete max : swap 68 and 13, delete 68



0	1	2	3	4	5	6	7	8	9	10	11
13	65	32	31	26	24	21	20	19	68		

Example (cont.)

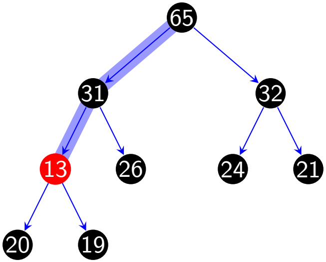
- Swap 13 and 65



0	1	2	3	4	5	6	7	8	9	10	11
65	13	32	31	26	24	21	20	19	68		

Example (cont.)

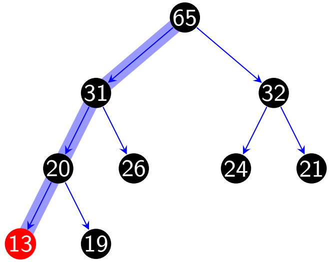
- Swap 13 and 31



0	1	2	3	4	5	6	7	8	9	10	11
65	31	32	13	26	24	21	20	19	68		

Example (cont.)

- Swap 13 and 20



0	1	2	3	4	5	6	7	8	9	10	11
65	31	32	20	26	24	21	13	19	68		

Analysis

implementation	insert	remove max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	$\log_2 N$	$\log_2 N$	1

Increase Key

- To increase the value of a certain key inside the max-heap, we need to reach this key first. In ordinary heaps, we can't search for a specific key inside the heap.
- Therefore, we'll keep a **map** (**hash table**) beside the original array. This map will store the index of each key inside the heap.

$\textsc{Increase-Key}(a, k, \Delta k, map)$
  $i \leftarrow map[k]$
  $map.\textsc{Remove}(k)$
  $a[i] \leftarrow a[i] + \Delta k$
  $map[a[i]] \leftarrow i$
  $\textsc{Up-Heap}(a, 0, map)$

d-Heaps

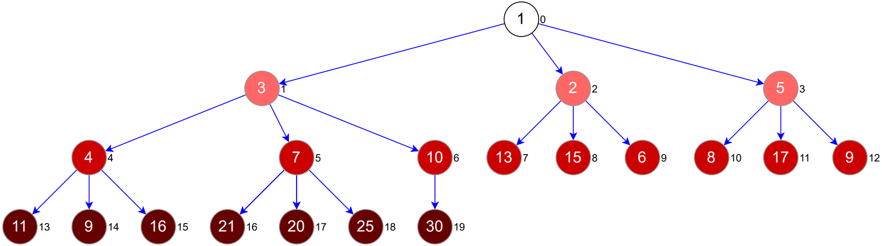
Introduction

Concept 4

a *d*-**heap** is a heap, each node of which has *d* children.

- A min 3-heap

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	2	5	4	7	10	13	15	6	8	17	9	11	9	16	21	20	25	30



Analysis

- A d-heap is much shallower than a binary heap

implementation	**insert**	**remove max**	**max**
unordered array	1	N	N
ordered array	N	1	1
binary heap	$\log_2 N$	$\log_2 N$	1
d-heap	$\log_d N$	$\log_d N$	1

Application

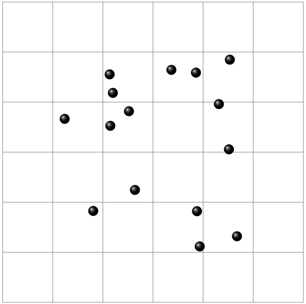
- Simulation
- Time-driven Simulation
- Event-driven Simulation

Molecular dynamics simulation of hard discs

Goal. Simulate the motion of N moving particles that behave according to the laws of elastic collision.

Problem. N bouncing balls in the unit square.

- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.



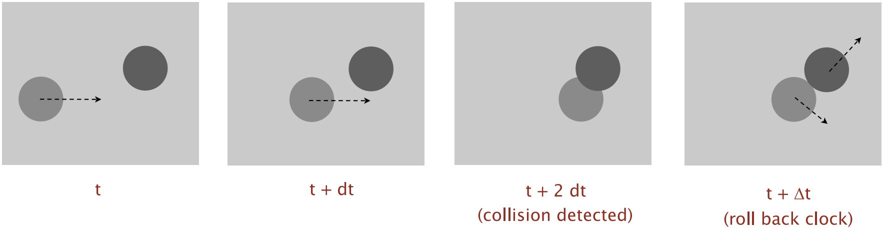
Bouncing balls

- Check for balls colliding with each other.
  - Physics problems: when? what effect?
  - CS problems: which object does the check? too many checks?

Time-driven simulation

Discretize time in quanta of size dt.

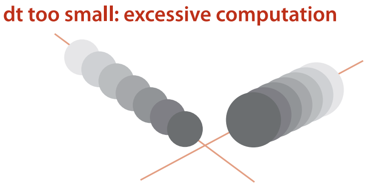
- Update the position of each particle after every dt units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.

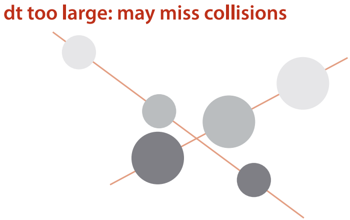


Time-driven simulation (cont.)

Main drawbacks

- $\sim N^2/2$ overlap checks per time quantum.
- Simulation is too slow if dt is very small.
- May miss collisions if dt is too large. (if colliding particles fail to overlap when we are looking)





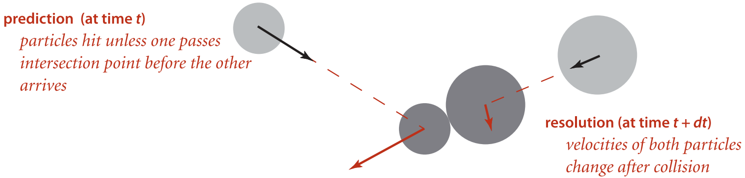
Event-driven simulation

Change **state** only when something happens.

- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain PQ of collision events, prioritized by time.
- Remove the min = get next collision.

Collision prediction. Given position, velocity, and radius of a particle. When will it collide next with a wall or another particle?

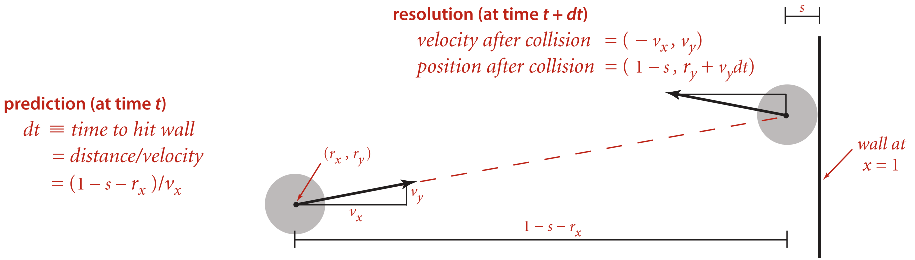
Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.



Particle-wall collision

Collision prediction and resolution.

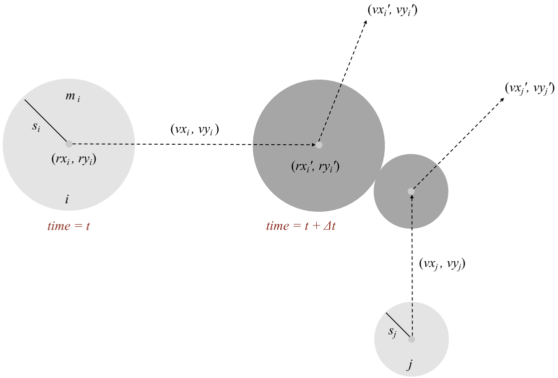
- Particle of radius s at position (r_x, r_y).
- Particle moving in unit box with velocity (v_x, v_y).
- Will it collide with a vertical wall? If so, when?



Particle-particle collision prediction

Collision prediction

- Particle i: radius s_i, position (rx_i, ry_i), velocity (vx_i, vy_i).
- Particle j: radius s_j, position (rx_j, ry_j), velocity (vx_j, vy_j).



Particle-particle collision prediction (cont.)

- Will particles i and j collide? If so, when?

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \Delta r \geq 0 \\ \infty & \text{if } d < 0 \\ -\dfrac{\Delta v \Delta r}{\Delta v \Delta v} & \text{otherwise} \end{cases} \tag{1}$$

$$\sigma = \sigma_i + \sigma_j \tag{2}$$

$$d = (\Delta v \Delta r)^2 - (\Delta v \Delta v)(\Delta r \Delta r - \sigma^2) \tag{3}$$

where

$$\Delta v = (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j) \tag{4}$$

$$\Delta r = (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j) \tag{5}$$

Particle-particle collision resolution

Collision resolution

- When two particles collide, how does velocity change?

$$\begin{aligned} vx_i' &= vx_i + \frac{Jx}{m_i} \\ vy_i' &= vy_i + \frac{Jy}{m_i} \\ vx_j' &= vx_j - \frac{Jx}{m_j} \\ vy_j' &= vy_j - \frac{Jy}{m_j} \end{aligned} \tag{6}$$

where

$$\begin{aligned} J &= \frac{2m_i m_j(\Delta v \Delta r)}{\sigma(m_i + m_j)} \\ Jx &= \frac{J\Delta rx}{\sigma} \\ Jy &= \frac{J\Delta ry}{\sigma} \end{aligned} \tag{7}$$

Collision system: event-driven simulation main loop

Initialization

- Fill PQ with all *potential* particle-wall collisions. ("potential" since collision may not happen if some other collision intervenes)
- Fill PQ with all *potential* particle-particle collisions.

Main loop

- Delete the impending event from PQ (min priority $= t$).
- If the event has been invalidated, ignore it.
- Advance all particles to time t, on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.

Workshop

Quiz

1. What is a priority queue?

   ....

   ....

   ....

2. What is a binary heap?

   ....

   ....

   ....

3. What is a *d*-heap?

   ....

   ....

   ....

Projects

- Programming exercises in [Cormen, 2009, Sedgewick, 2002]

References

Cormen, T. H. (2009).
Introduction to algorithms.
MIT press.

Sedgewick, R. (2002).
Algorithms in Java, Parts 1-4, volume 1.
Addison-Wesley Professional.

Walls and Mirrors (2014).
Data Abstraction And Problem Solving with C++.
Pearson.