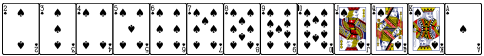
COMP 2011: Data Structures

Lecture 10. Priority Queues and Heapsort

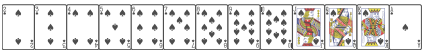
Dr. CAO Yixin

November, 2021

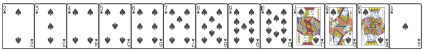


Review of Lecture 9

- Self-balancing binary search trees: AVL trees and red-black trees.
- Binary heaps.
- Operations `insert` and `removeMax` (shape, then bubble up/down).
- Again, all the modifications are restricted to a path
  though you need to check the other child of every node on the path for deletion.
- Also `peek` and `changeKey`.



Priority Queues



Priority queues (🐢)

- A priority queue is an ADT that has `insert` and `removeMax`/`removeMin`.
- Sometimes it supports operations `getMax`/`getMin` and `size` as well.
- Warning: A priority queue is not a queue![1]
- Can be implemented using (un)ordered arrays, (un)ordered linked lists.

- Recall the exercise on Queen Elizabeth Hospital.

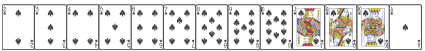
An abstract data type (ADT) defines
- a state of an object and
- operations that act on the object, possibly changing the state.

[1] Indeed, both stacks and queues can be viewed as priority queues.

Using a priority queue to sort

- Insert the elements, one by one, into a priority queue.
- Remove the elements, one by one, from the priority queue.

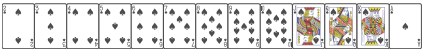
It takes O(_____) time?



Which of the following can be used to implement a priority queue.

- [T] an unsorted array
- [ ] a sorted array
- [ ] a maximum heap
- [ ] a minimum heap
- [ ] a binary search tree
- [ ] a self-balancing binary search tree

If yes, what's the running time for `insert` and `removeMax`?



Implementation	`insert`	`removeMax`	Algorithm	Best	Worst
Unordered Array	$O(1)$	$O(n)$	selection	$O(n^2)$	$O(n^2)$
Ordered Array	$O(n)$	$O(1)$	insertion	$O(n)$	$O(n^2)$
Heap	$O(\log n)$	$O(\log n)$	Heapsort[2]	$O(n \log n)$	$O(n \log n)$
Leonardo Heap	$O(\log n)$	$O(\log n)$	Smoothsort	$O(n \log n)$	$O(n \log n)$
Self-balancing binary search tree	$O(\log n)$	$O(\log n)$		$O(n \log n)$	$O(n \log n)$

[2] Not really

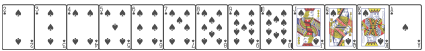
Priority queues and heaps

- In practice, a priority queue is almost always implemented using a heap
- but they're conceptually different!

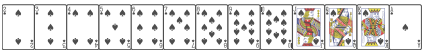
Check java.util.PriorityQueue and its source codes.

Priority queues $\neq$ heaps

Freestyle swimming (🌍) vs. Front crawl (🌍). (Tokyo 2020)

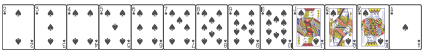


In-place Heapsort

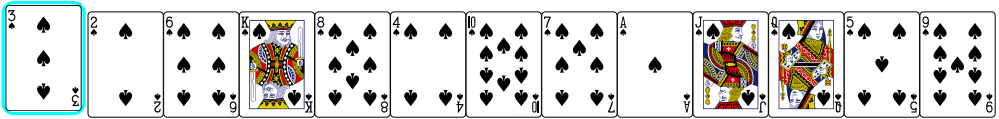


Sorting with a heap

```
1  void heapSort(int[] keys, T[] data) {
2      Heap<T> heap = new Heap<T>(keys.length);
3      for (int i = 0; i < keys.length; i++)
4          heap.insert(keys[i], data[i]);
5      for (int i = keys.length - 1; i >= 0; i--)
6          data[i] = heap.removeMax();
7  }
```

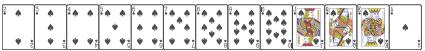


1

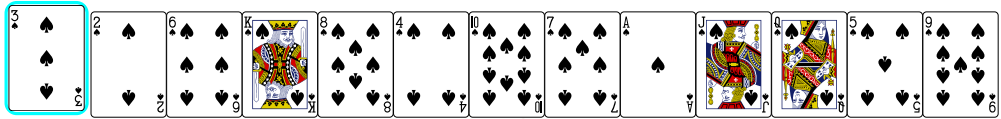


3

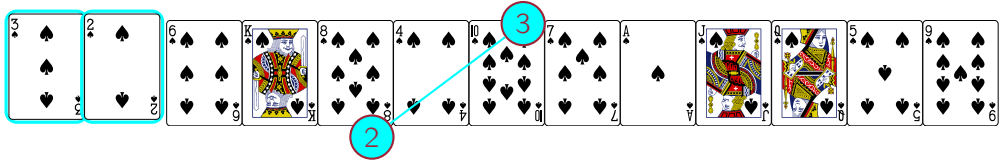
We add the cards into the heap one by one, which initially has only the first card.

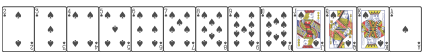


1

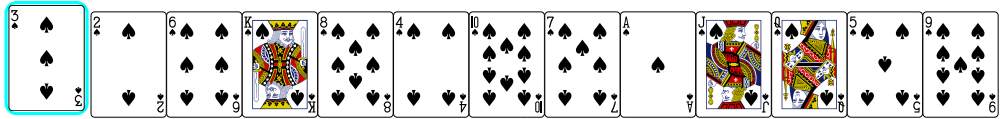


2

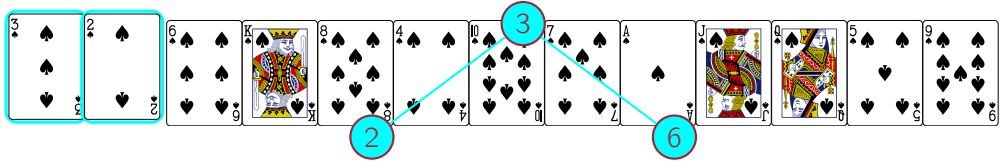


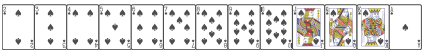


1

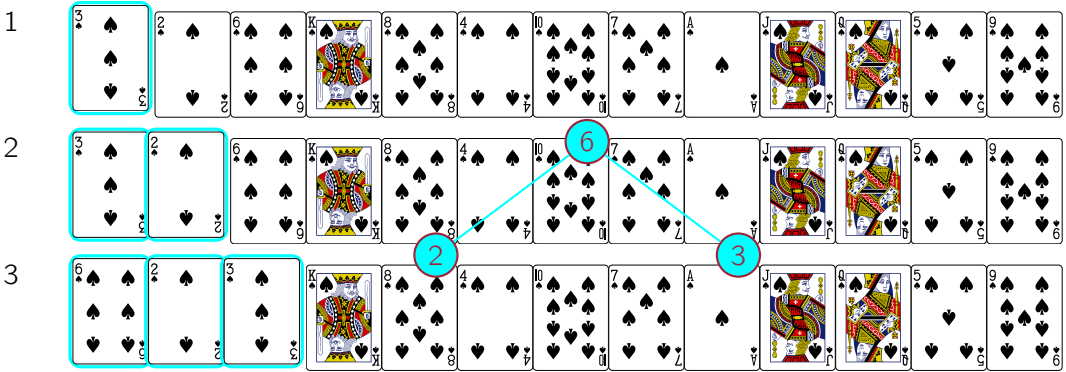


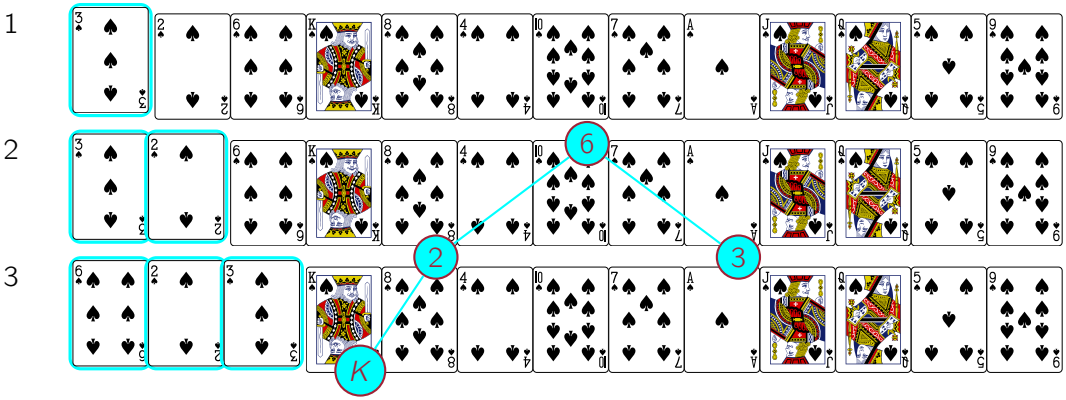
2



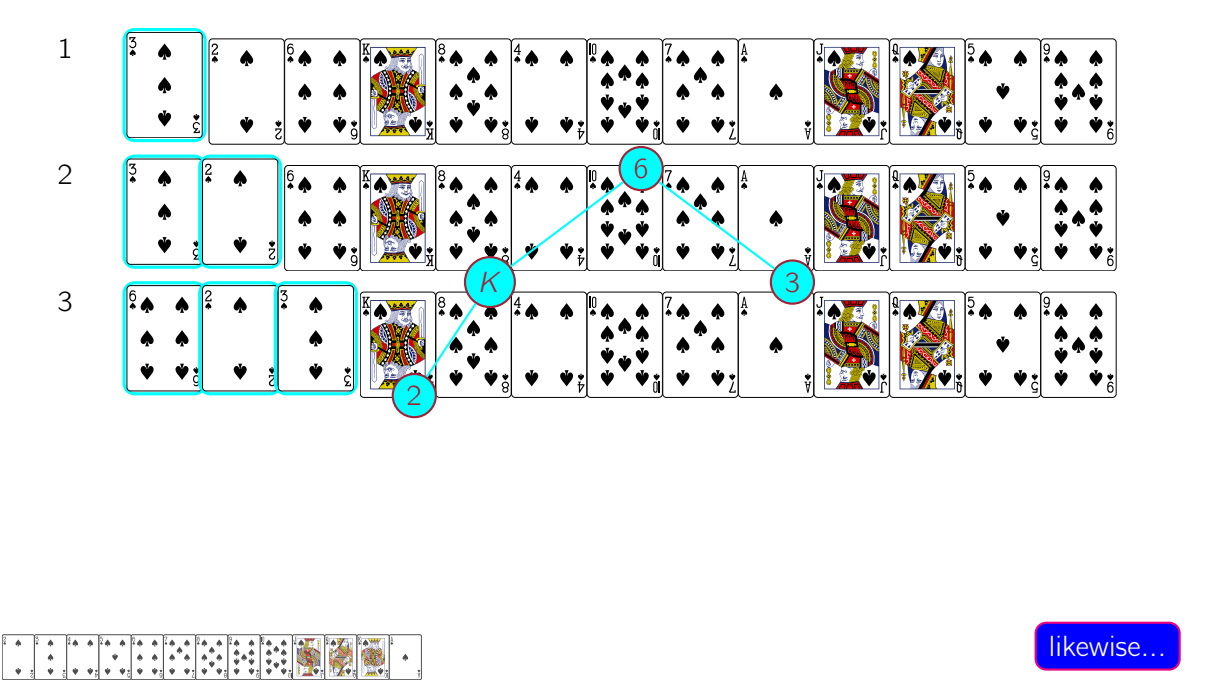


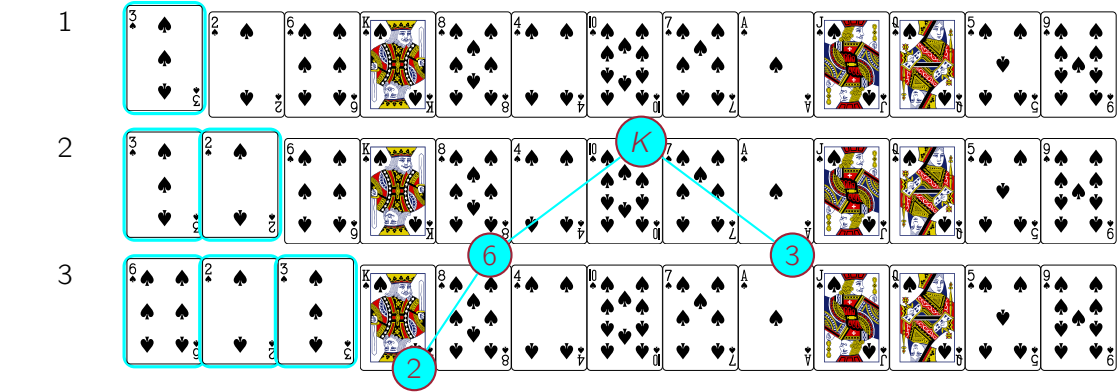
The correct position of the next card (6) is?


1
2
3
6
2
3

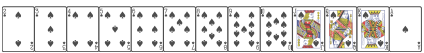


likewise...


1
2
3
6
K
3
2
likewise...



This time we don't swap, so j moves but i stays.

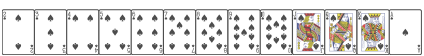


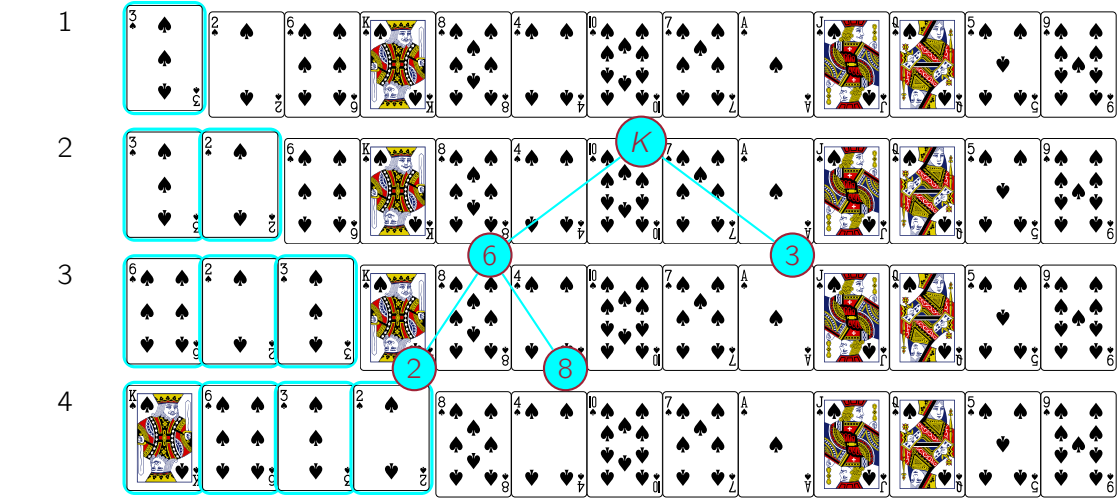
1

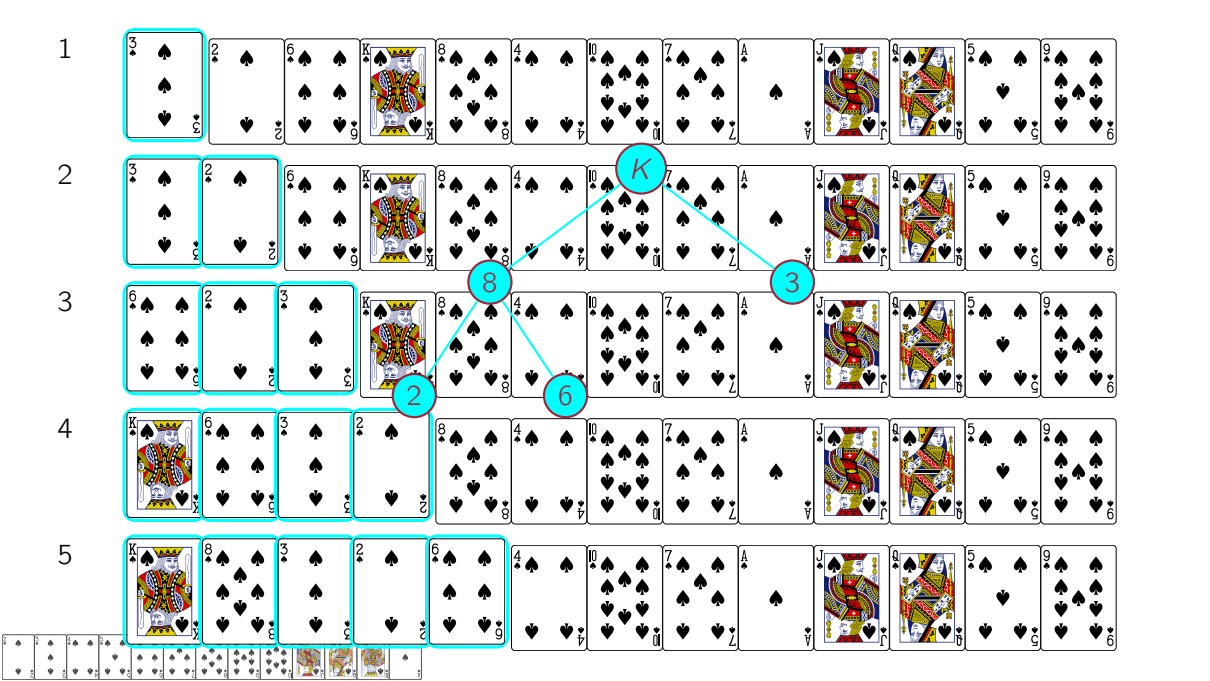
2

3

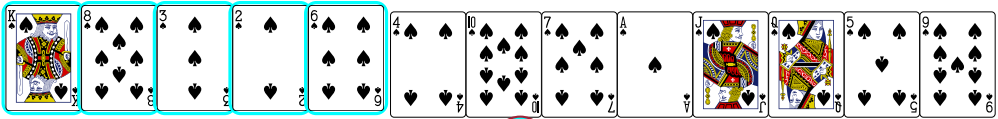
4

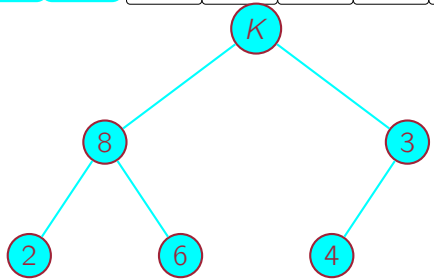


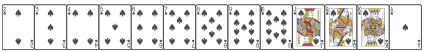

1
2
3
4
K
6
3
2
8


1
2
3
4
5
K
8
3
2
6

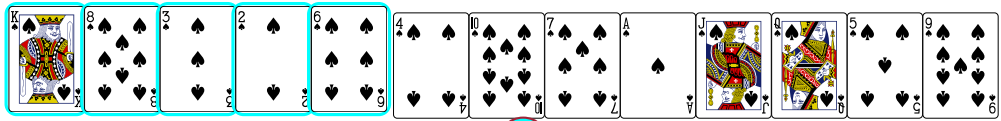
5



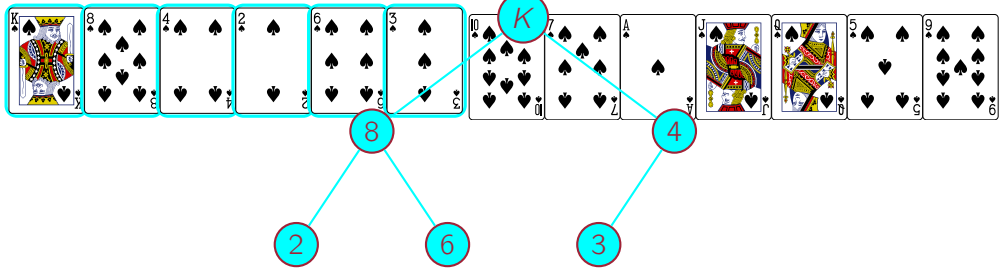




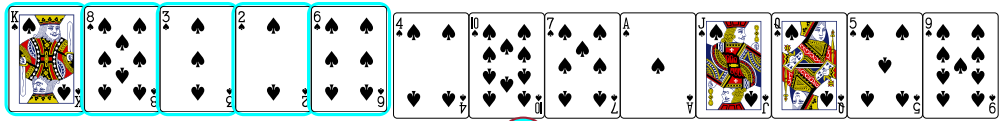
5



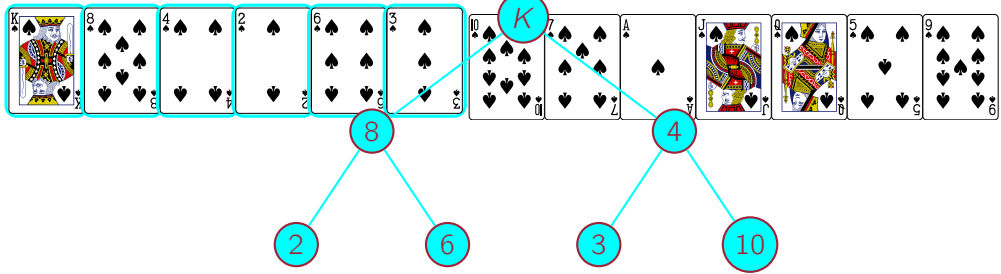
6

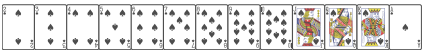


5



6

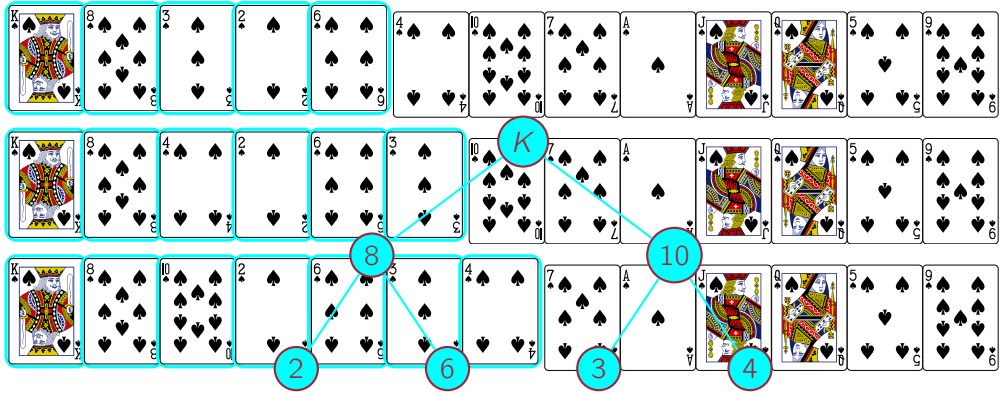


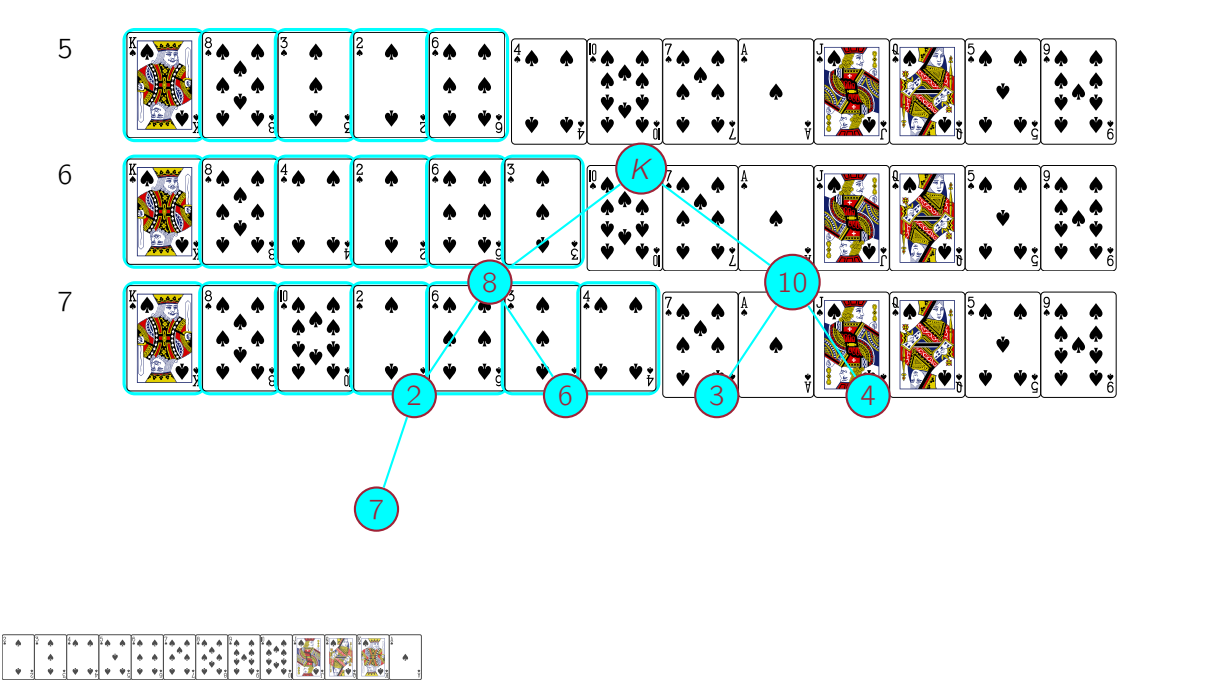


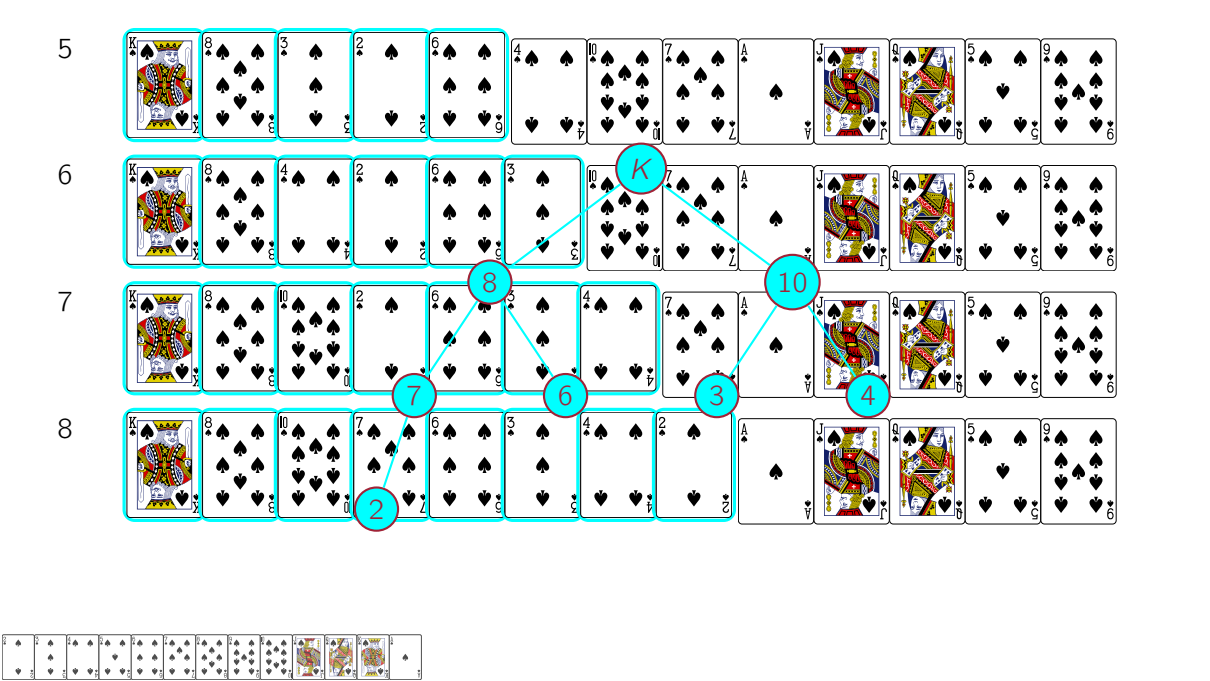
5

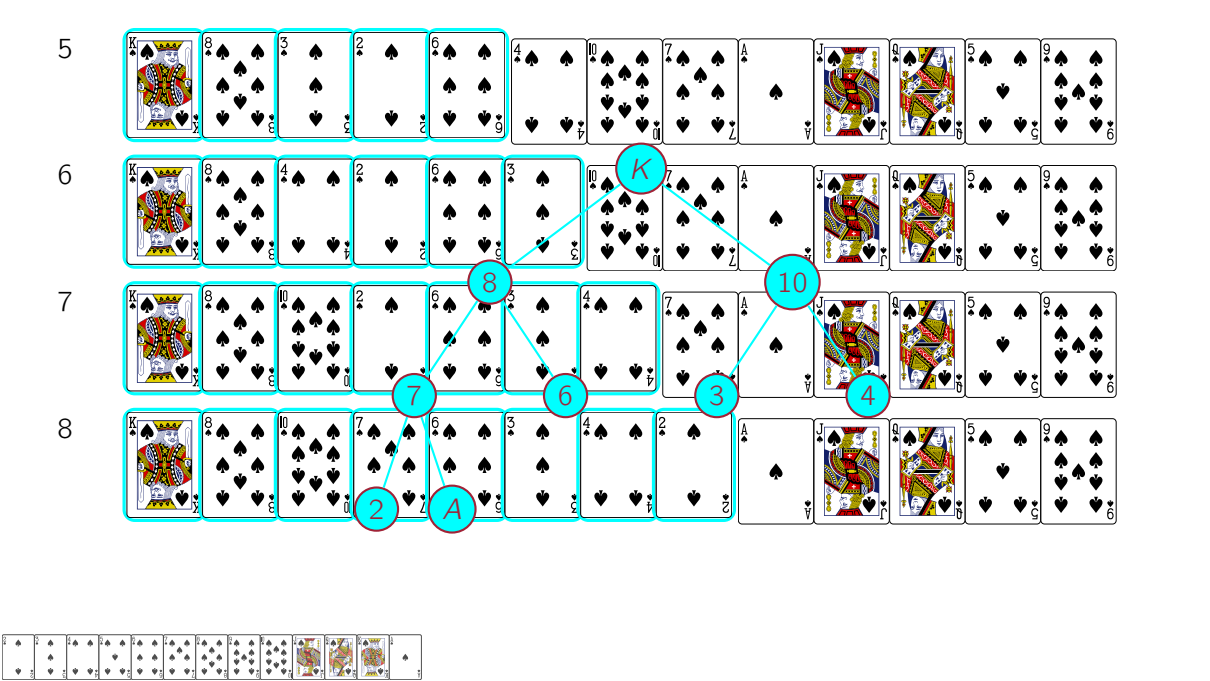
6

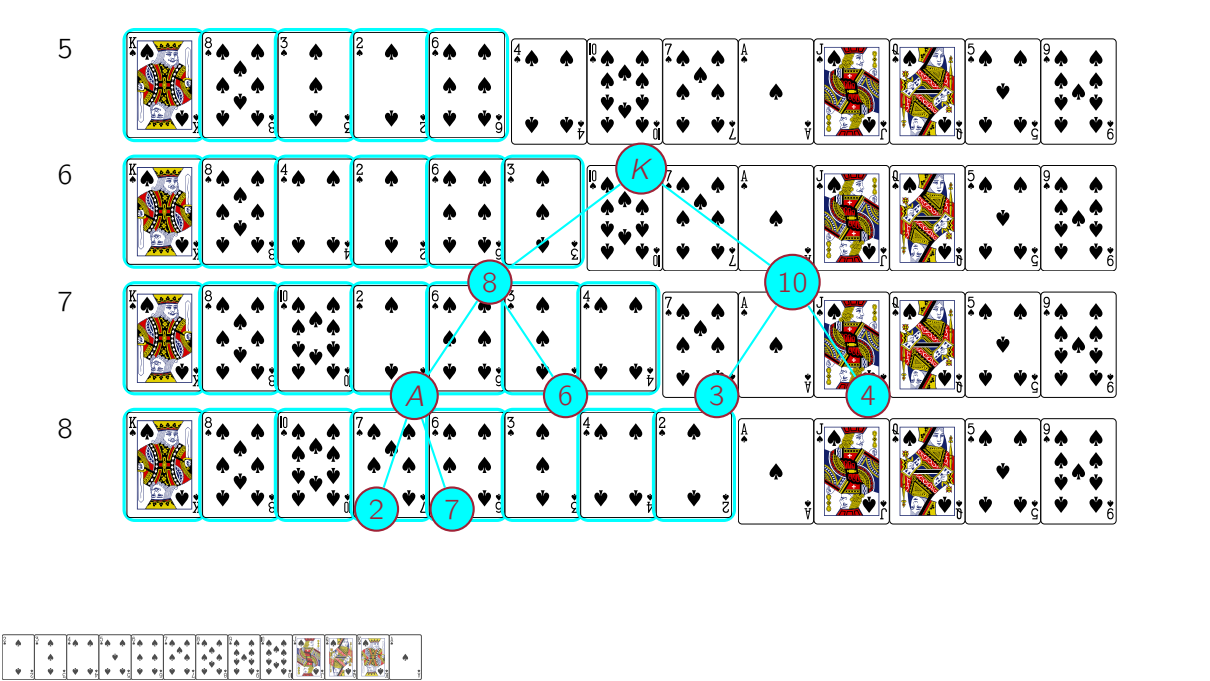
7

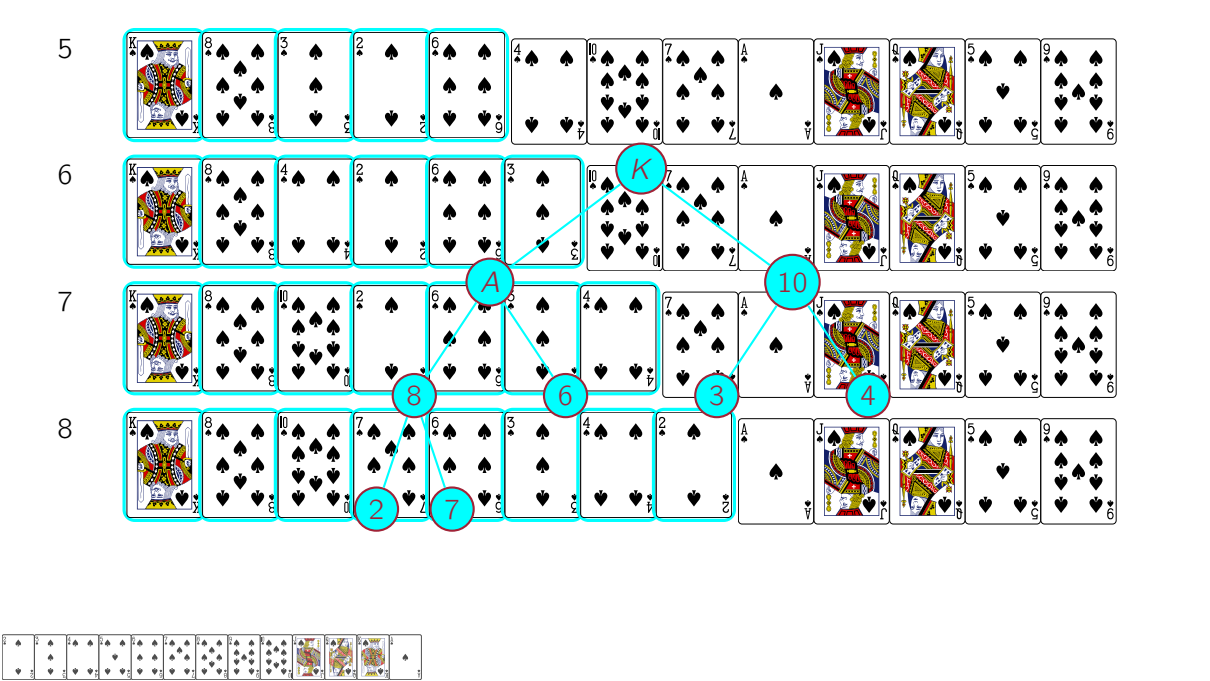


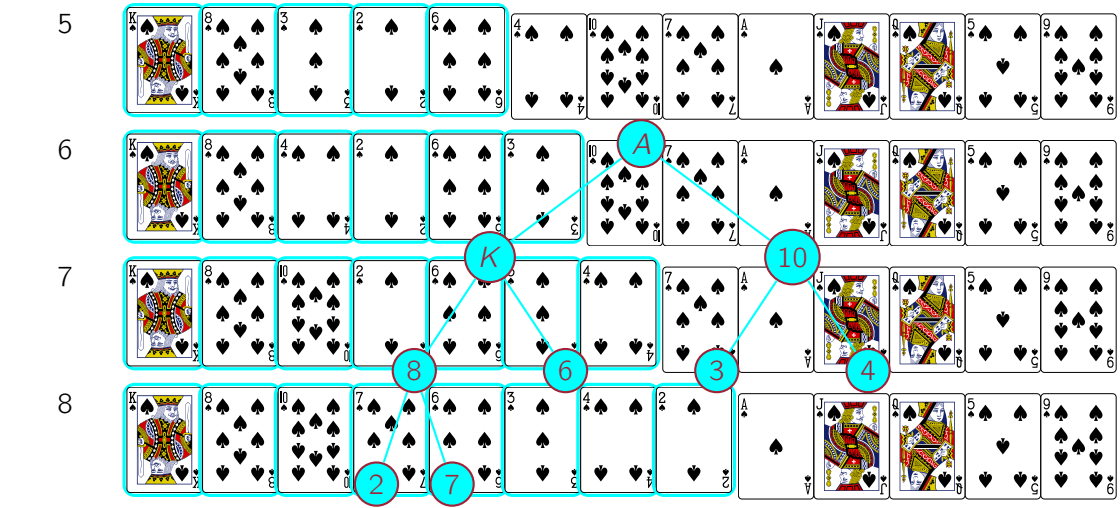

5
6
7
K
8
10
2
6
3
4
7

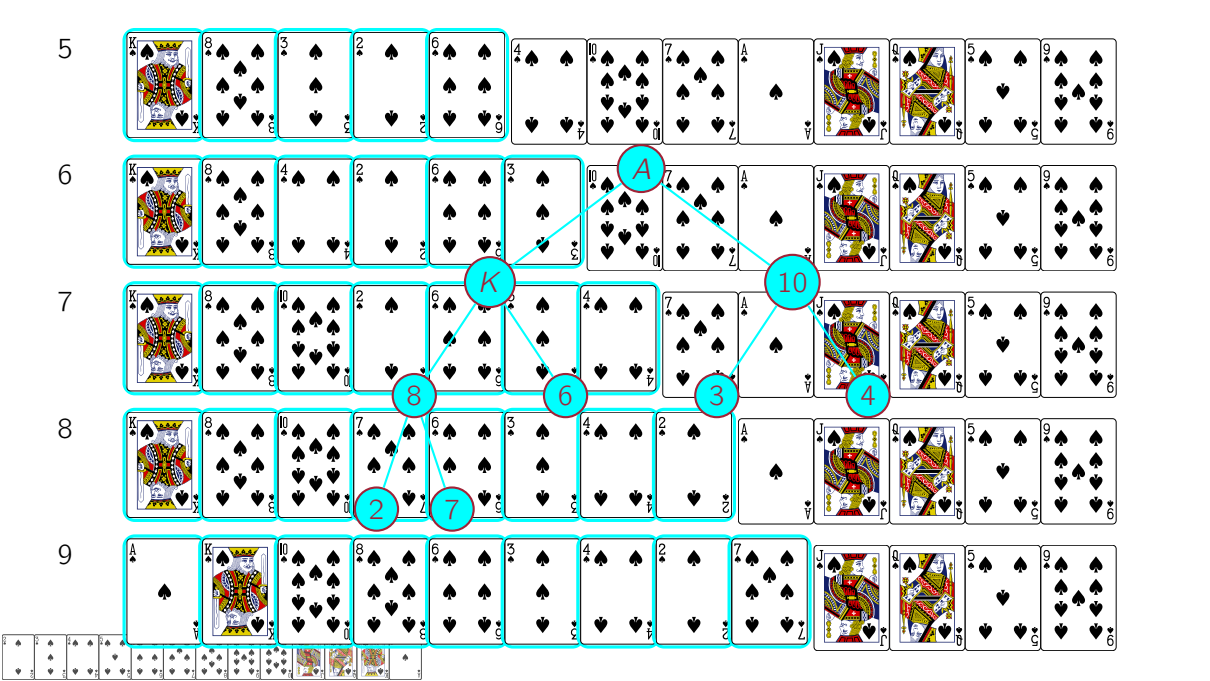

5
6
7
8
K
8
10
7
6
3
4
2


5
6
7
8
K
8
10
7
6
3
4
2
A

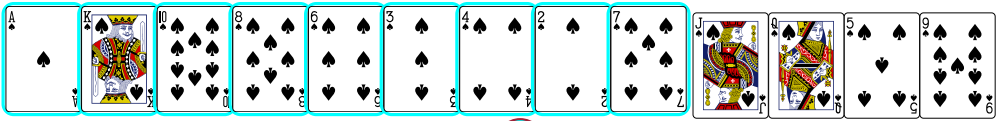

5
6
7
8
K
8
10
A
6
3
4
2
7

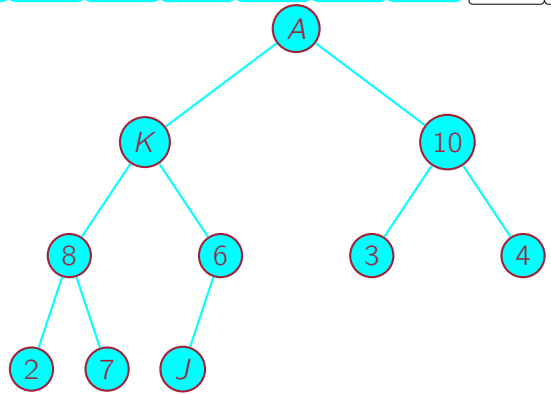

5
6
7
8
K
A
10
8
6
3
4
2
7

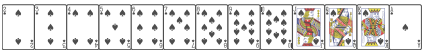

5
6
7
8
A
K
10
8
6
3
4
2
7


5
6
7
8
9
A
K
10
8
6
3
4
2
7

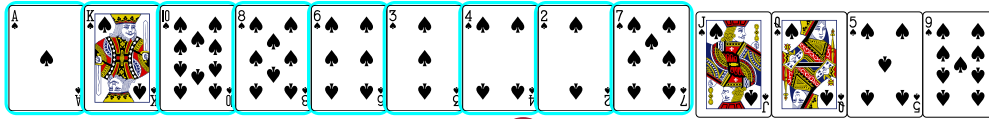
9



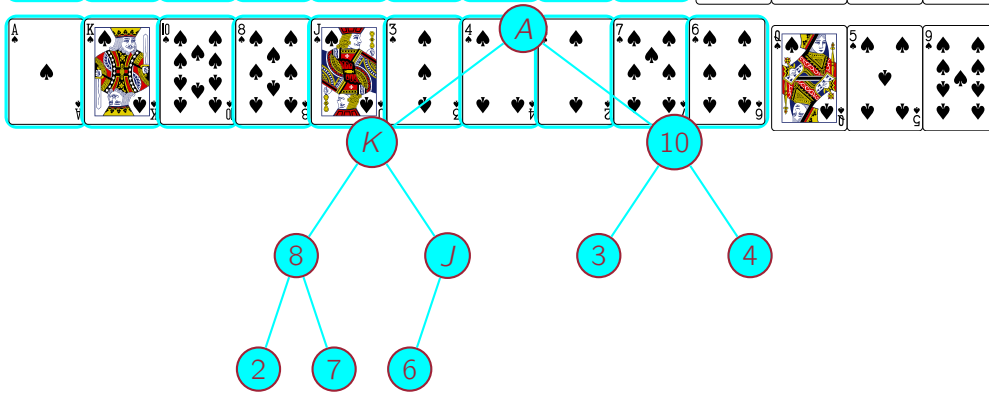


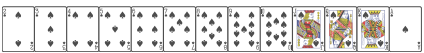


9

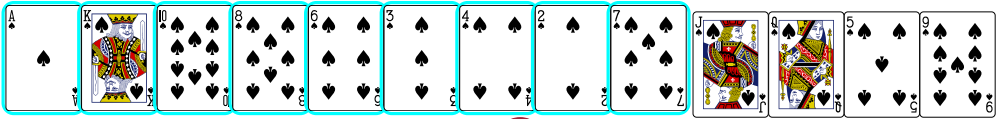


10

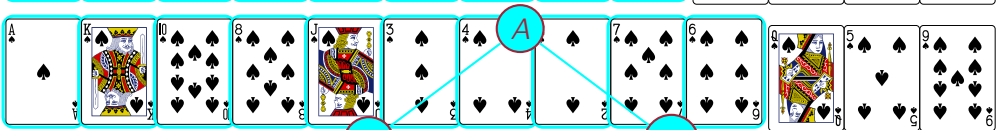


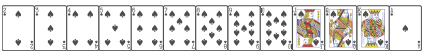


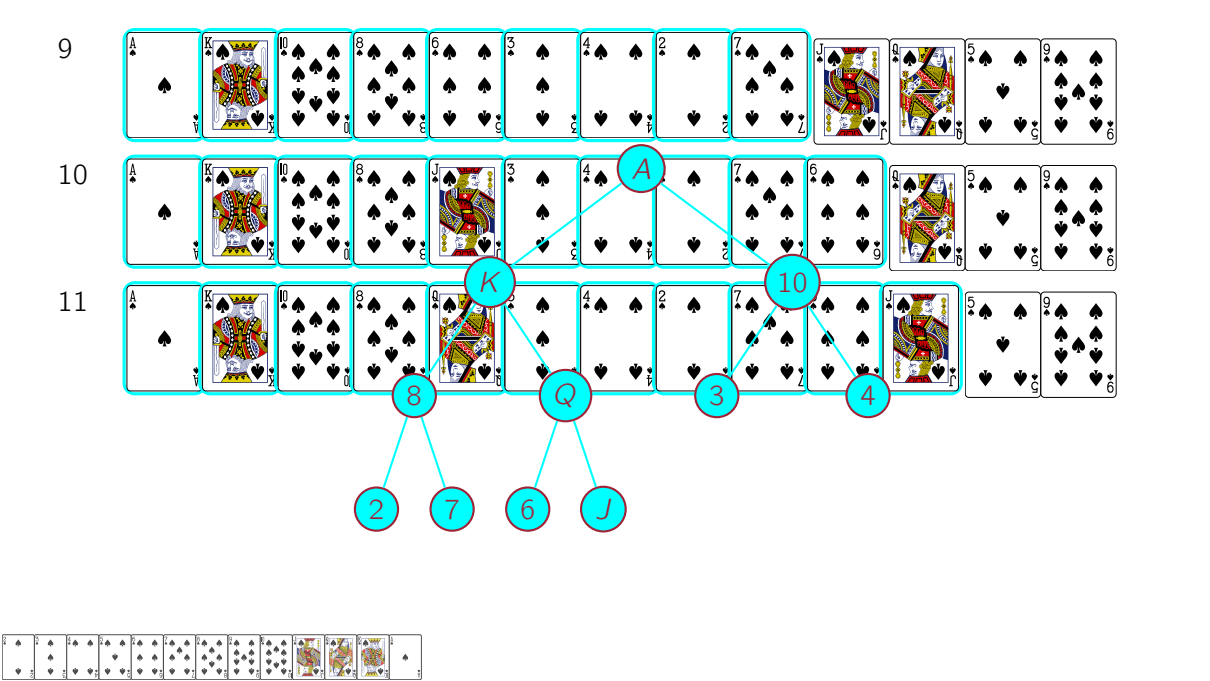
9

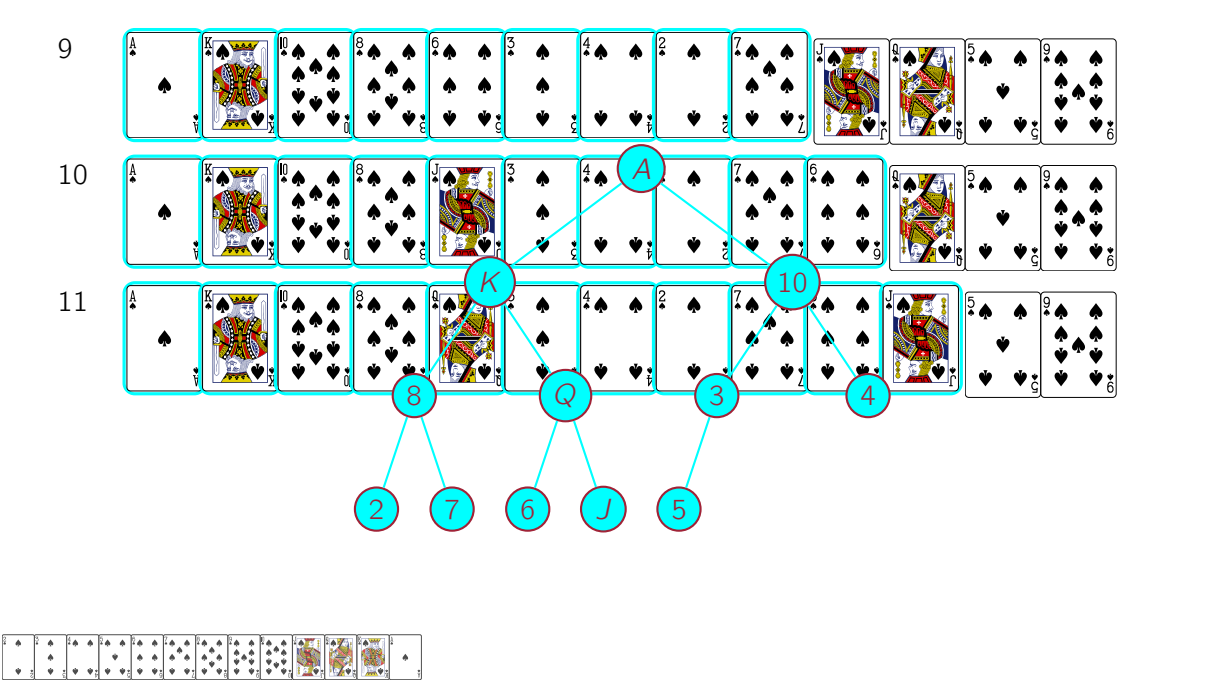


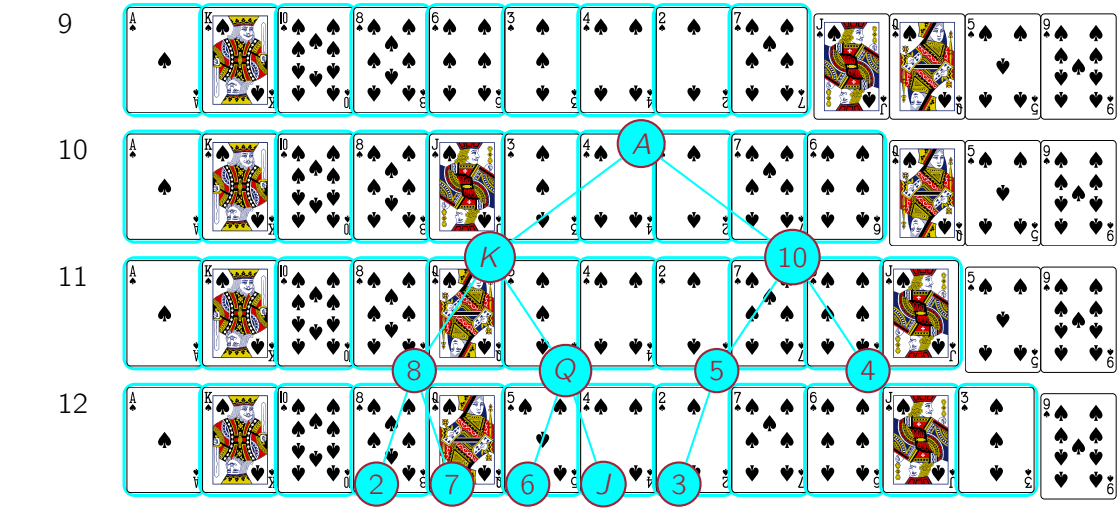
10

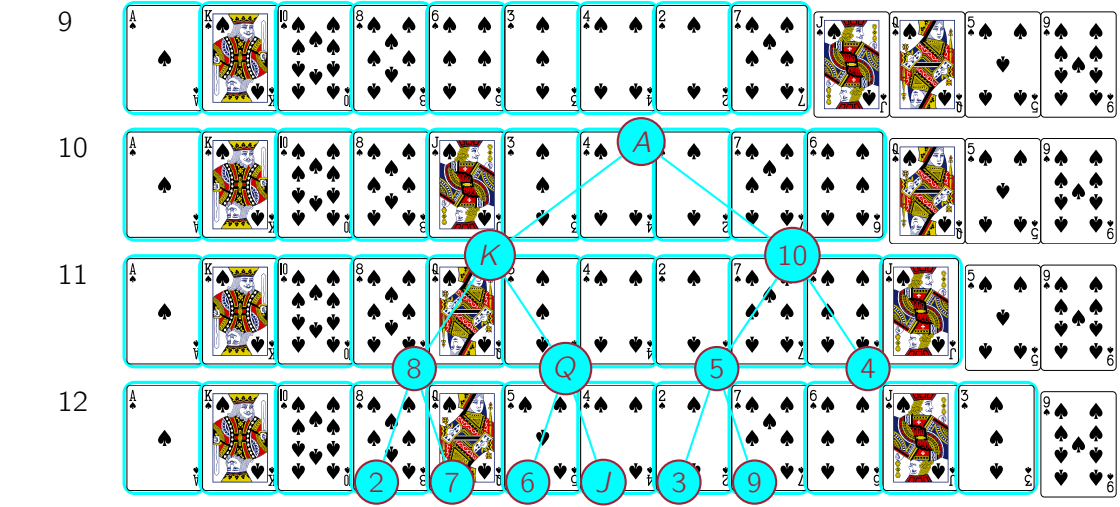


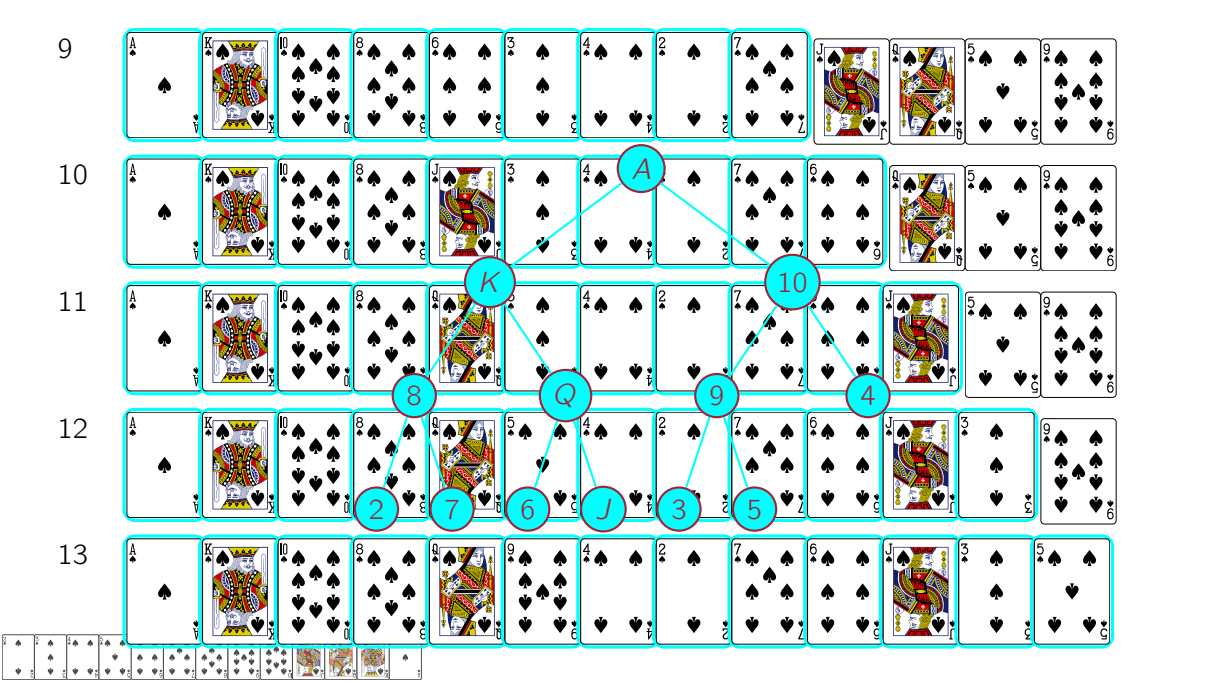


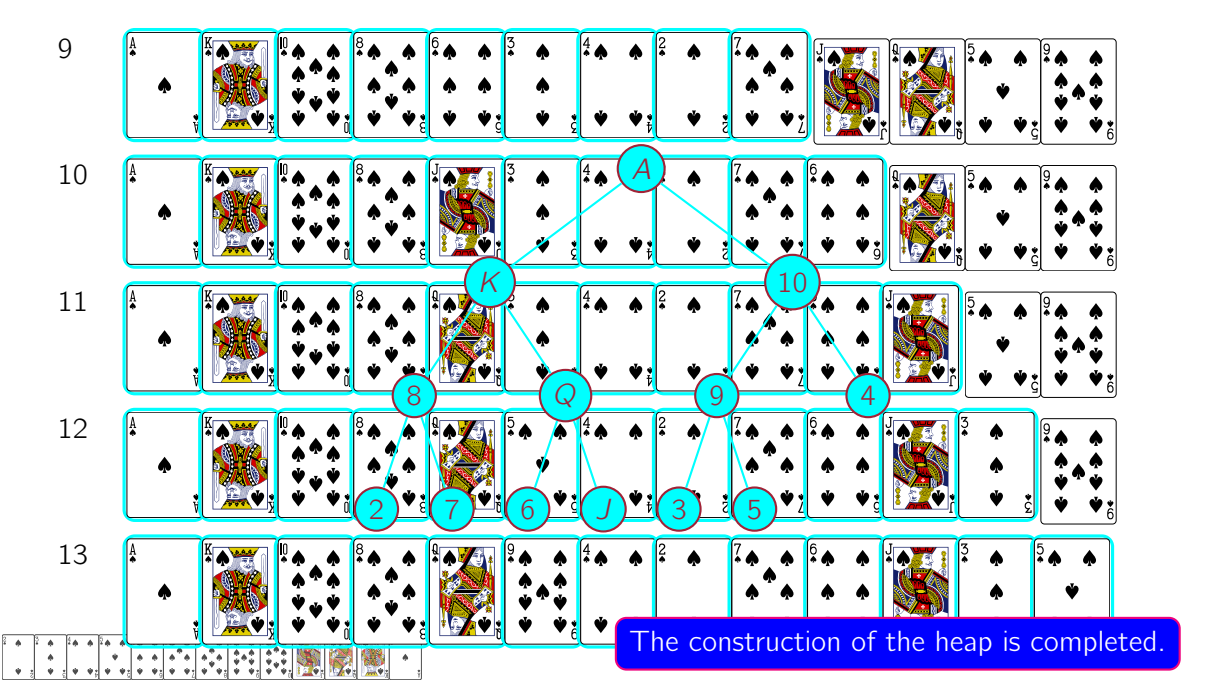

9
10
11
A
K
10
8
Q
3
4
2
7
6
J

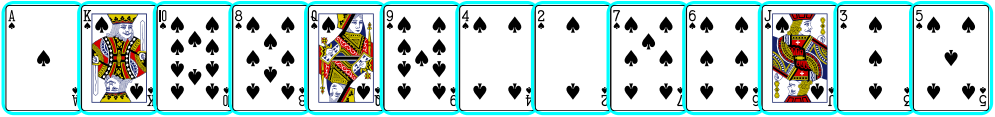

9
10
11
A
K
10
8
Q
3
4
2
7
6
J
5

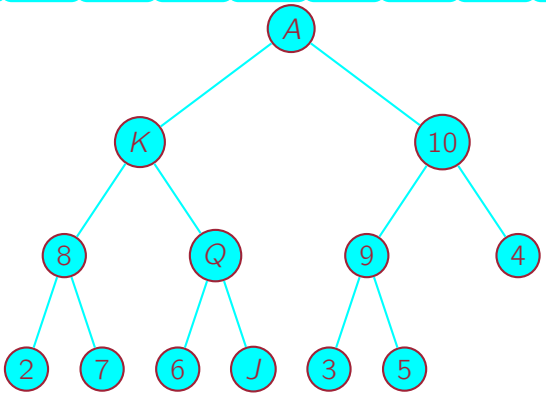

9
10
11
12
A
K
10
8
Q
5
4
2
7
6
J
3

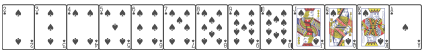

9
10
11
12
A
K
10
8
Q
5
4
2
7
6
J
3
9


9
10
11
12
13
A
K
10
8
Q
9
4
2
7
6
J
3
5


9
10
11
12
13
A
K
10
8
Q
9
4
2
7
6
J
3
5
The construction of the heap is completed.

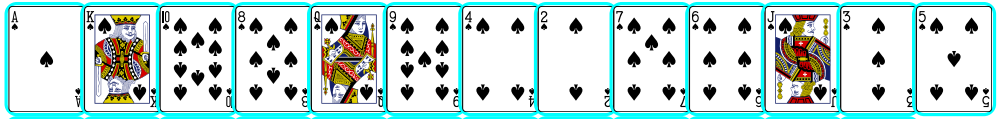




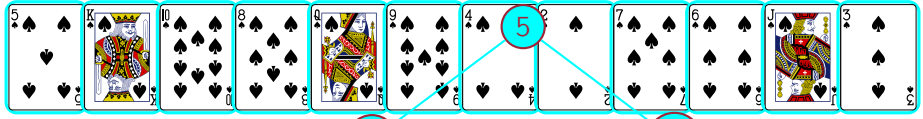


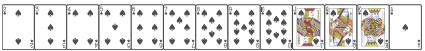
What happens if we delete the root (max)?

13

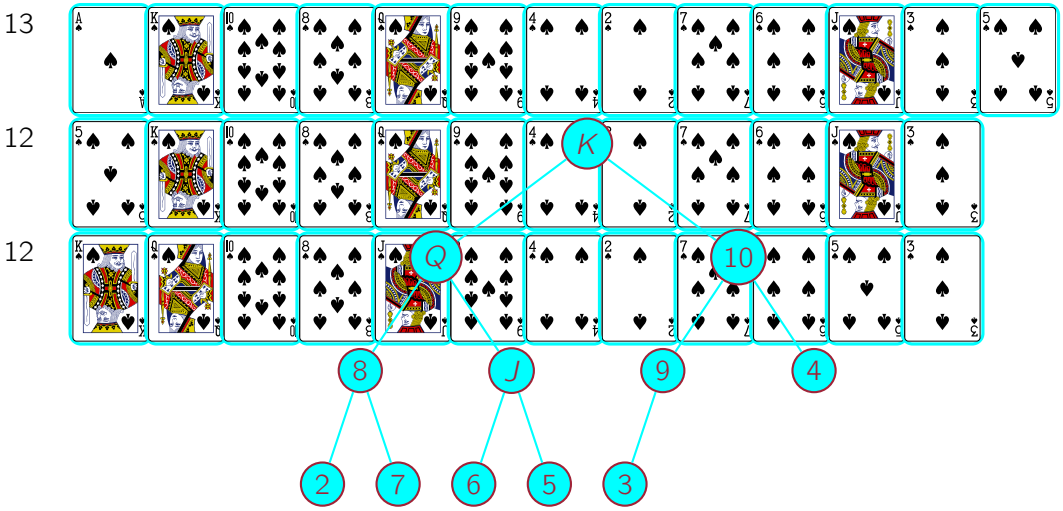


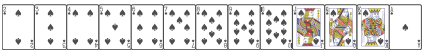
12



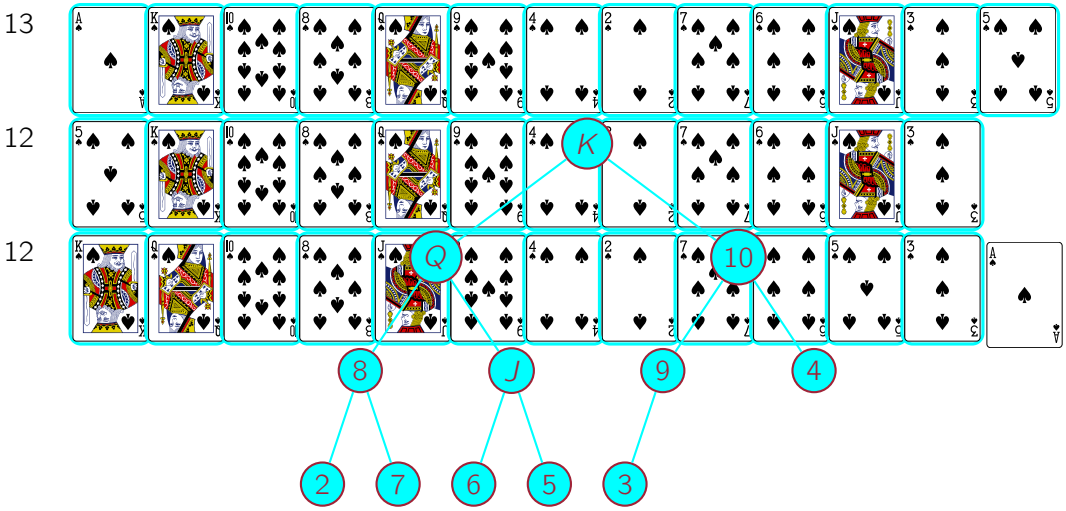


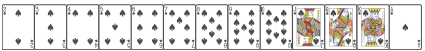
What happens if we delete the root (max)?



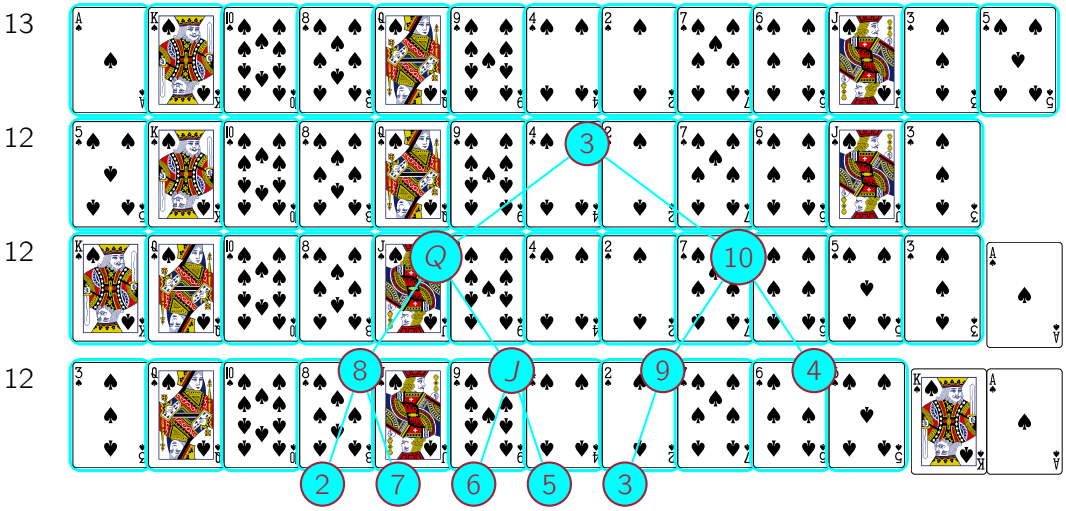


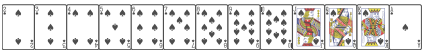
Where do we put A?





The last position is now empty.





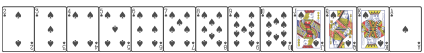
In the next turn, we swap 3 and K, so it's in-place.

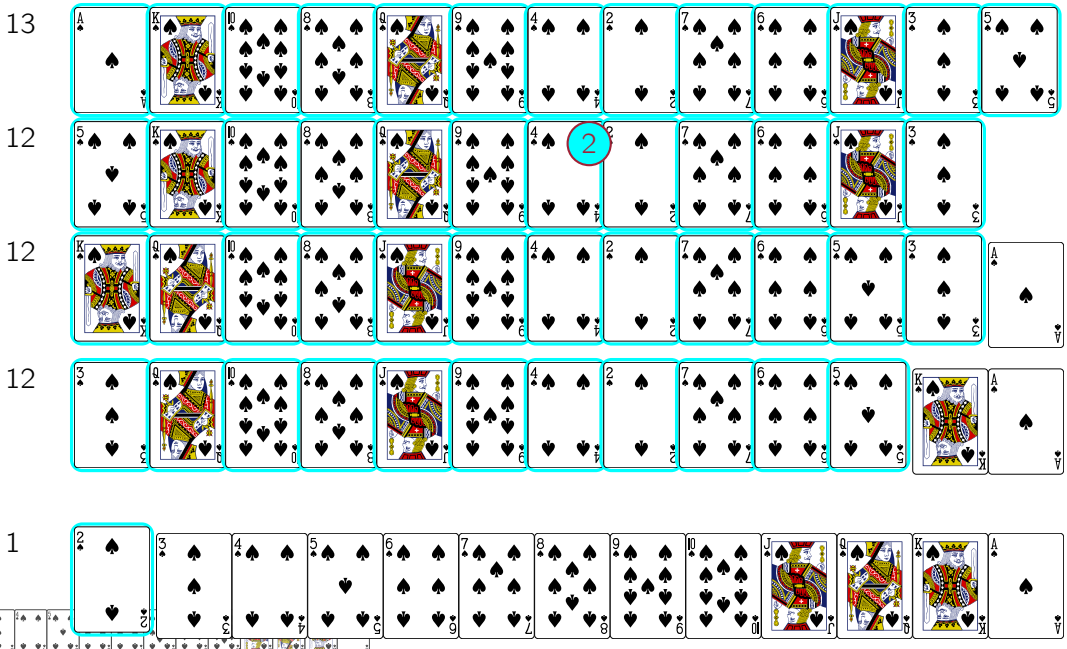
13

12

12

12




13
12
12
12
1

The heapsort method

```
void void heapsort(int[] a) {
    for (int i = 1; i < a.length; i++)
        up(a, i);
    for (int i = a.length - 1; i > 0; i--) {
        swap(a, 0, i);
        down(a, 0, i);
    }
}
```

```
void up(int[] a, int c) {
    if (c == 0) return;
    int p = (c - 1) / 2;
    if (a[c] <= a[p])
        return;
    swap(a, c, p);
    up(a, p);
}
```

```
void down(int[] a, int p, int s) {
    if (p * 2 + 2 > s) return;
    int l = p * 2 + 1;
    if (l+1<s && a[l]<a[l+1]) l++;
    if (a[p] >= a[l]) return;
    swap(a, p, l);
    down(a, l, s);
}
```

Running time of heapsort

`heapify` (turn something into a heap):

$$\sum_{i=1}^{n-1} \lceil \log(i+1) \rceil < \sum_{i=1}^{n} (\log n + 1) = O(n \log n)$$

`removeMax`:

$$\sum_{i=1}^{n-1} \lceil \log(n+1-i) \rceil < \sum_{i=1}^{n} (\log n + 1) = O(n \log n)$$

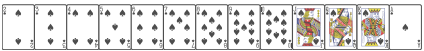
a quick and informal answer

a rigor and detailed analysis

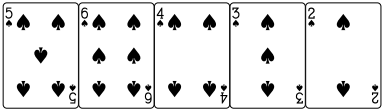
- `heapify` can be done in $O(n)$ time. nontrivial to show
- It's not hard to get an upper bound (O) of the complexity of an algorithm, but it's nontrivial to get a tight one (Θ).
- Analysis of an algorithm is harder than its design.

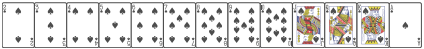
A trivial best-case instance: $[7, 7, 7, \ldots, 7]$.

How about a sorted array, a reversely sorted array (with distinct keys)?

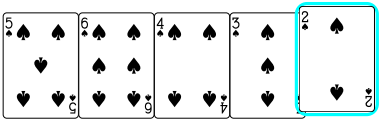


A closer look at selection sort

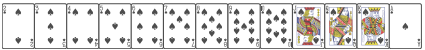




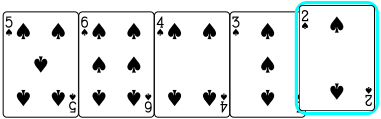
A closer look at selection sort

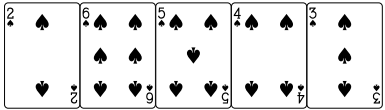


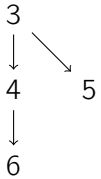
2
↓
3
↓
4
↓
5
↓
6

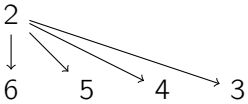


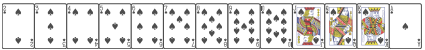
A closer look at selection sort





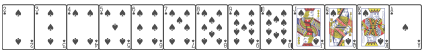






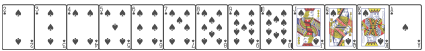
Online resources

- Demo of Sorting Algorithms
- Comparison of Sorting Algorithms
- A slow and interactive version (heapsort others)
- More information about sorting algorithms ()



Stability

- Is heapsort stable?
- If no, can you find an example?

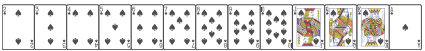


Summary of heapsort

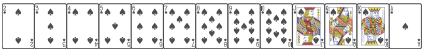
- A heap is usually implemented as an array representing a complete binary tree.
- The root is at index 0 and the last item at index $n - 1$.
- The parent-child relation can be easily decided by the index calculations.

- A heap can be based on a binary tree (not a search tree).
- The last occupied node and the first tree node can be found in $O(\log n)$ time.

- Conceptually, heapsort makes n insertions into a heap, followed by n removals.
- The same array can be used for the initial unordered data, for the heap array, and for the final sorted data.
- Heapsort takes $O(n \log n)$ time and $O(1)$ space (in-place).
- Heapsort can be made stable, then slower. All sorting algorithms can.

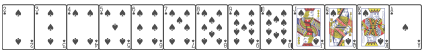


Applications



$[65, 85, 17, 88, 66, 71, 45, 38, 95, 48, 18, 68, 60, 96, 55]$.

The 5th smallest element is ____?

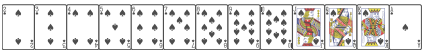


$[65, 85, 17, 88, 66, 71, 45, 38, 95, 48, 18, 68, 60, 96, 55]$.

The 5th smallest element is ____?

Naïve: Sort the array, and return `a[4]`.

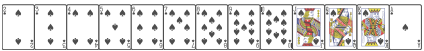
$17, 18, 38, 45, 48, 55, 60, 65, 66, 68, 71, 85, 88, 95, 96$



[65, 85, 17, 88, 66, 71, 45, 38, 95, 48, 18, 68, 60, 96, 55].

The 5th smallest element is ____?

2. Build a min heap on all of them, and do `removeMin` five times.

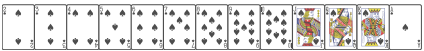


[65, 85, 17, 88, 66, 71, 45, 38, 95, 48, 18, 68, 60, 96, 55].

The 5th smallest element is ____?

3. Keep track of 5 smallest element.

[65, 85, 17, 88, 66]



[65, 85, 17, 88, 66, 71, 45, 38, 95, 48, 18, 68, 60, 96, 55].

The 5th smallest element is ____?

3. Keep track of 5 smallest element.

[65, 85, 17, 88, 66] →[65, 85, 17, 88, 66, 71]

71 replaces 88; there are five elements < 88, so 88 cannot be the answer.

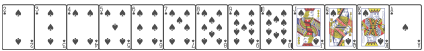
[65, 85, 17, 88, 66, 71, 45, 38, 95, 48, 18, 68, 60, 96, 55].

The 5th smallest element is ____?

3. Keep track of 5 smallest element.

[65, 85, 17, 88, 66] →[65, 85, 17, 88, 66, 71] →[65, 85, 17, 66, 71, 45]

45 replaces 85.

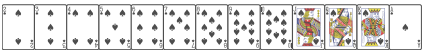


[65, 85, 17, 88, 66, 71, 45, 38, 95, 48, 18, 68, 60, 96, 55].

The 5th smallest element is ____?

3. Keep track of 5 smallest element.

[65, 85, 17, 88, 66] →[65, 85, 17, 88, 66, 71] →[65, 85, 17, 66, 71, 45]
→ ····· →[17, 45, 38, 48, 18, 85]



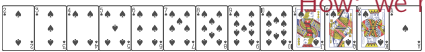
[65, 85, 17, 88, 66, 71, 45, 38, 95, 48, 18, 68, 60, 96, 55].

The 5th smallest element is ____?

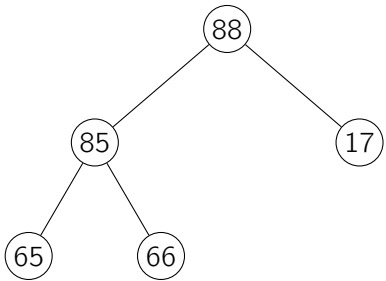
3. Keep track of 5 smallest element.

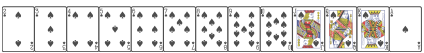
[65, 85, 17, 88, 66] →[65, 85, 17, 88, 66, 71] →[65, 85, 17, 66, 71, 45]
→ ····· →[17, 45, 38, 48, 18, 85]

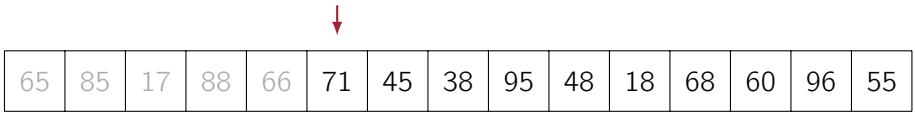
How: we need to `removeMax`, so a maximum heap.

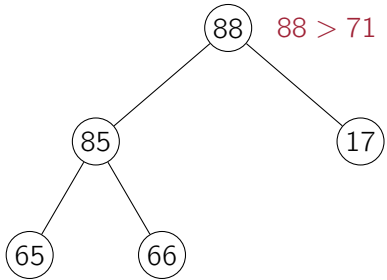


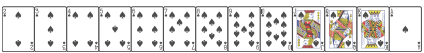
65	85	17	88	66	71	45	38	95	48	18	68	60	96	55

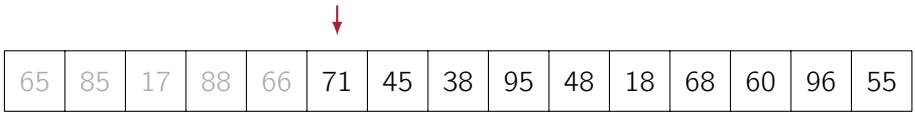


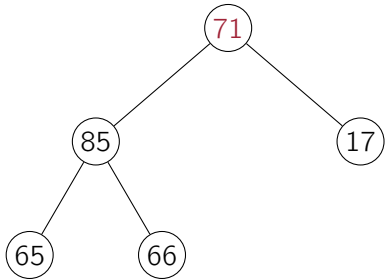


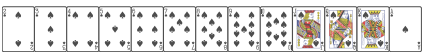

65
85
17
88
66
71
45
38
95
48
18
68
60
96
55

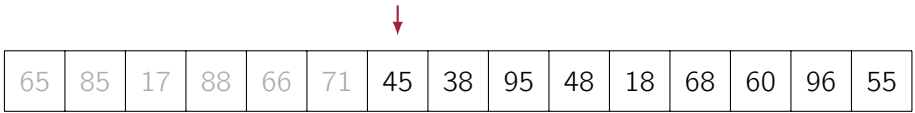

88
88 > 71
85
17
65
66

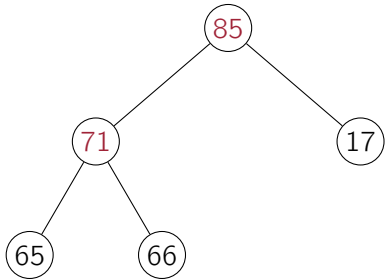


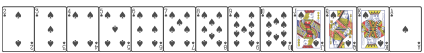

65
85
17
88
66
71
45
38
95
48
18
68
60
96
55


71
85
17
65
66




65
85
17
88
66
71
45
38
95
48
18
68
60
96
55

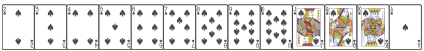

85
71
17
65
66



$O(n \log n)$
sort

$O(n + k \log n)$
a min heap (n)

$O(n \log k)$
a max heap (k)



$O(n \log n)$
sort

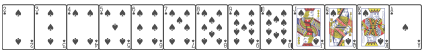
$O(n)$

$O(n + k \log n)$
a min heap (n)

$O(n)$

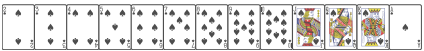
$O(n \log k)$
a max heap (k)

$O(k)$



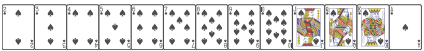
$[10.4m, 35.4b, 66.5k, ..., 120m, 30.5b, ..., 79.8m, 192.7m, 963.5k, ..., 22.1b].$

The wealth gap (HK) between the 1000 richest and the 1000 poorest (🫧)



$[10.4m, 35.4b, 66.5k, \ldots, 120m, 30.5b, \ldots, 79.8m, 192.7m, 963.5k, \ldots\ldots, 22.1b]$.

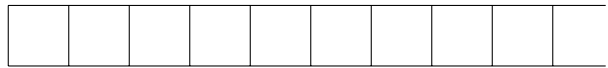
Similar to the previous, we can use a heap to keep track of the 1000 richest.
and another heap to keep track of the 1000 poorest.



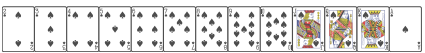
Merging multiple sorted arrays

8	10	14	89
32	38	50	77
16	32	64	128
11	12	13	14

$\Longrightarrow$



First round: 8, 32, 16, 11



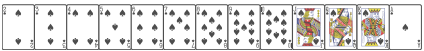
8	10	14	89
32	38	50	77
16	32	64	128
11	12	13	14

$\Longrightarrow$

8									

First round: 8, 32, 16, 11

Next round: 10, 32, 16, 11



Week 12: Hashing