LINMA2111 - CM2 - 2015

Mathieu Dath & Damien Scieur

May 4, 2015

In the last course, we have seen the QuickSort algorithm (Hoare, 1962). This algorithm is often used in practice due to the low constant in its complexity. We have also seen that there is some randomness in this algorithm. One may now wonder if there is a deterministic algorithm in $\mathcal{O}(n \log n)$ (worst case complexity) which sorts "in place" (meaning that it uses no extra memory.

Fortunately for the people who really hate randomness, the answer is yes, but we need to be smart about the way we keep track of the information. In other words, we need a good data structure.

Stack v.s. Queue

Let us make a little digression about two well studied data structures, stacks and queues. Both are ordered lists of entries with the same static structure but different ways to access/modify it. The difference is graphically explained on Figure 1.

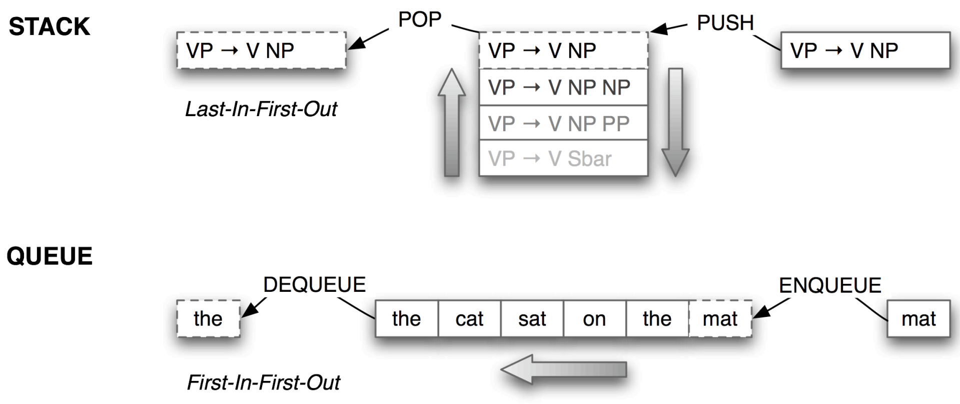

Figure 1: Graphic representations of the stack and the queue structures. Elements are inserted using the Push/Enqueue action and removed using the Pop/Dequeue action.

We define a data structure as a static structured set (like a list, a tree, etc.) with a list of operations (like read, insert, etc.). It can thus be seen as an abstract mathematical concept.

A data structure for a deterministic QuickSort

If we get back to our problem, the structure we need is a priority queue: a set of numbers with the following operations:

- Insert
- Extract Max

An example of the use of such a structure is a todo list where the elements are tasks and where each task is assigned a priority (which can be interpreted as a number).

If we have an implementation for a priority queue, then we have a sorting algorithm for [T(1) T(2) $\cdots$ T(n)], we just need to:

1. Insert T(1), T(2), ..., T(n) to an empty priority queue.
2. Extract Max n times

The complexity of such an algorithm is $n\,(\text{cost(Insert)} + \text{cost(Extract Max)})$.

Potential structures for this sorting algorithm

We need a data structure to implement the sorting algorithm algorithm we just described.

Unordered table Using an unordered table, the operations complexity are

- Insert: $\mathcal{O}(1)$
- Extract Max: $\Theta(n)$

The sorting algorithm associated to this implementation has thus the complexity $\Theta(n^2)$.
$\Rightarrow$ This is SelectionSort.

Ordered table Using an ordered table, the operations complexity are

- Insert: $\mathcal{O}(n)$ [*cost of shifting the entries*]
- Extract Max: $\mathcal{O}(1)$

The sorting algorithm associated to this implementation has thus the complexity $\mathcal{O}(n^2)$.
$\Rightarrow$ This is InsertionSort.

"Heap" Using a heap, the operations complexity are

- Insert: $\mathcal{O}(\log n)$
- Extract Max: $\mathcal{O}(\log n)$

The sorting algorithm associated to this implementation has thus the complexity $\mathcal{O}(n \log n)$.

Heap structure

A heap is a tree with the following properties:

- essentially binary complete (only the last level may be incomplete (as it is filled from left to right))
- entry(x) $\leq$ entry(father(x)) $\forall$ node $x \neq$ root

An example of a heap is shown in Figure 2.

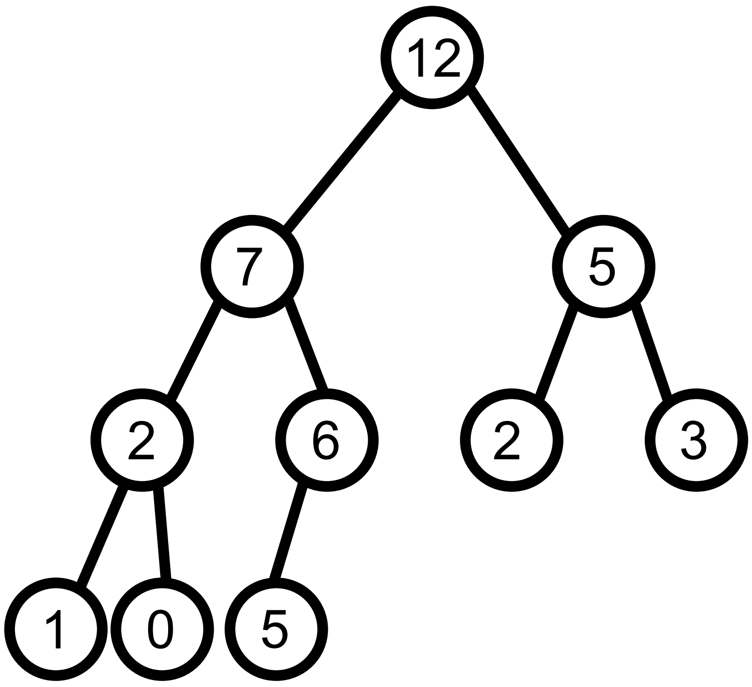

Figure 2: Example of a heap

Insert To insert a new node into the heap, add it to the last level in the first free position and swap it with its father until the heap is properly re-established.
$\Rightarrow \mathcal{O}(\log n)$

Extract Max To extract the maximal node from the heap, one needs to:

1. Swap root (the max entry) with last entry
2. Remove the max entry
3. Swap new root with its *largest* child until the heap properly is restored.

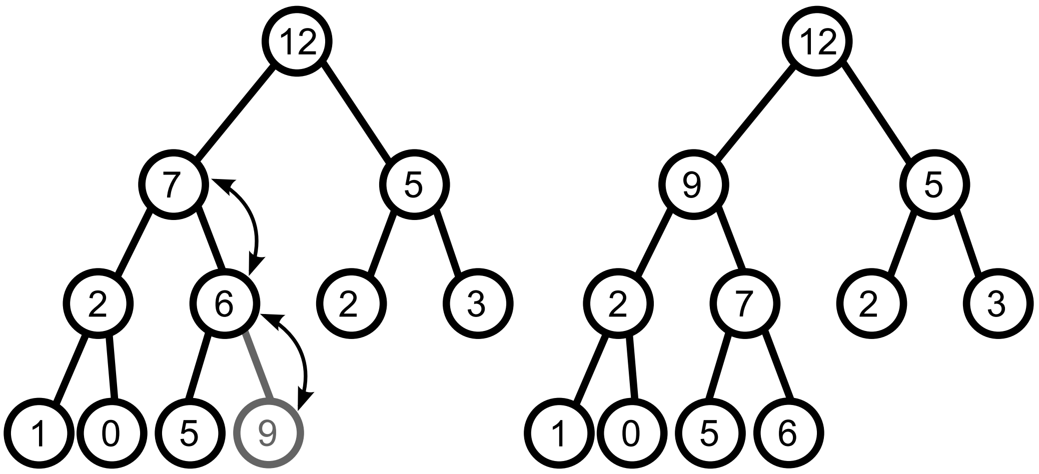

Figure 3: A heap before/after adding one element

$\Rightarrow \mathcal{O}(\log n)$

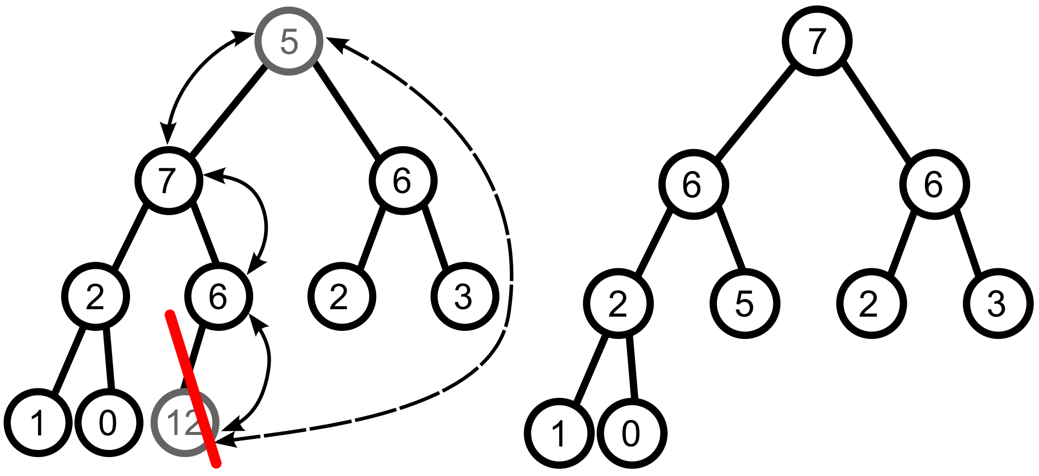

Figure 4: A heap before/after removing the largest entry

The resulting sorting algorithm is called the HeapSort (Williams, 1964) and has a complexity of $\mathcal{O}(n \log n)$.

Remarks

- The heap may be stored as an array:

Heap (T)	12	7	6	2	6	2	3	1	0	5
Index	1	2	3	4	5	6	7	8	9	10

with the following properties:

$$\begin{aligned}
&\text{LeftChild}(T(i)) = T(2i)\\
&\text{RightChild}(T(i)) = T(2i+1)\\
&\text{Father}(T(i)) = T\left(\left\lfloor \frac{i}{2} \right\rfloor\right)
\end{aligned}$$

$\Rightarrow$ We can thus sort in place!

- Create a heap from an input can be done in $\mathcal{O}(n)$.
- There exist a correspondence between a heap and a priority queue.
- The implementation of a data structure often offer trade-offs: one operation can be cheap if another is costly.

Can we do better?

Now that we have found a way to deterministically sort an array in $\mathcal{O}(n \log n)$, we can ask ourselves: can we sort in $o(n \log n)$ (i.e. in a complexity strictly better than $n \log n$) for either the worst case or the average case complexity, if we assume nothing on the entries (i.e. all we can test is "compare two entries", "$T(i) \leq T(j)$")?

Eg: 20-questions game With 20 Yes-No questions, you can guess a number between 1 and 1000000, but not an arbitrary number between 1 and 2000000 ($\geq 2^{20}$), no matter the questioning strategy.
In order to answer our question, let us introduce the following theorems.

Theorem 1. *A binary tree (thus with at most 2 children for each node) with N leaves has at least $\lfloor \log N \rfloor$ levels.*

Proof. Label each edge 0 or 1 (see Figure 5). Every leaf is uniquely identified by a binary word encoding the path root $\rightarrow$ leaf.
As we cannot uniquely identify N nodes with binary words all of length $< \lfloor \log N \rfloor$, there is at least one path of length $> \log N$. □

Theorem 2. *No deterministic sorting algorithm can sort n entries (with just comparisons) in $o(n \log n)$ worst case scenario.*

Proof. Build the decision tree (see Figure 6) from the sequence of questions to be asked to the data (the comparisons). The number of leaves of this tree is $n! = n(n-1)(n-2)...1$. This is the number of possible orders of the input array, each of which will require a different sequence of decisions (leading to different answers to the questions, as the algorithm is

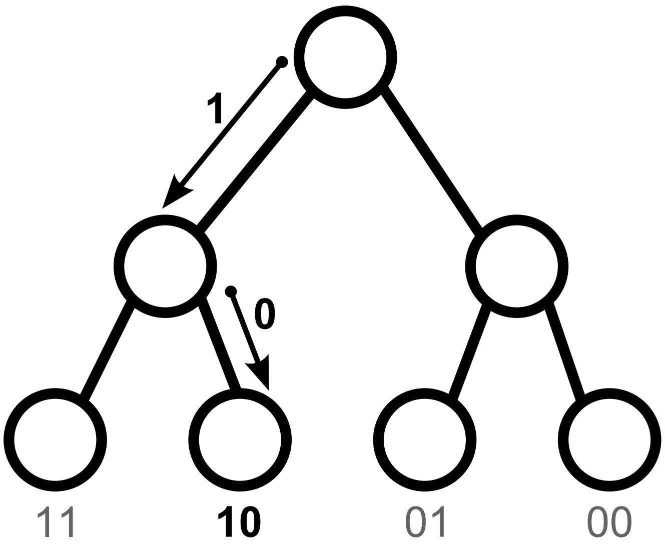

Figure 5: Example of a binary tree

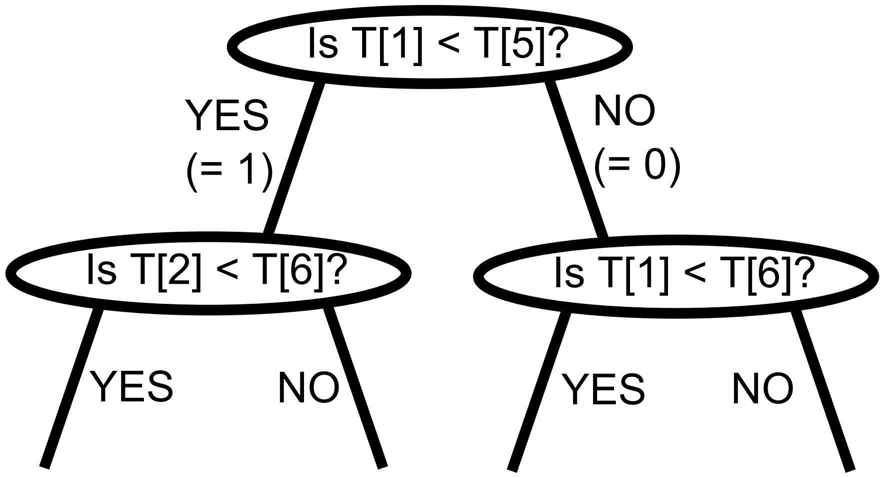

Figure 6: Decision tree

deterministic: the same answers leads to the same actions on the data).

$\Rightarrow \exists$ one path of length $\log(n!)$ and thus $\log(n!)$ questions.

Using the Stirling approximation $n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$, we obtain $\log(n!) = n \log n$. We have thus $\Omega(n \log n)$ as worst case complexity. $\square$

This last theorem answers our question for the worst case scenario, but can the average-case complexity be better? The following theorems will answer that question.

Theorem 3 (Shannon 1948)**.** *Let an N-set be endowed with the probabilistic distribution $\rho_1, \rho_2, ..., \rho_N$. Label every entry with a binary word (in a prefix way: no word is a prefix to another). Then the expected length (weighted by (ρ_i)) is $\geq \sum_{i=1}^{N} -\rho_i \log \rho_i$.*

Theorem 4. *The average length of a path root $\rightarrow$ leaf of a N-leaf binary tree, with uniform distribution on leaves, is at least $\sum_{i=1}^{N} -\frac{1}{N} \log \frac{1}{N} = \log N$.*

Theorem 5. *The average-case complexity of* any *deterministic sorting algorithm using comparisons is $\Omega(n \log n)$.*

Proof. Apply previous theorems on decision tree. $\square$

NB: Randomized algorithms cannot be better either.
As we now know, with our current hypothesis, we cannot do better than $\Omega(n \log n)$. But with supplementary knowledge, we can beat the $\Omega(n \log n)$ bound.

If we know that the entries $\in \{1, 2, ..., k\}$, we can define the algorithm CountingSort on [T(1) T(2) $\cdots$ T(n)].

CountingSort

1. - Count how many entries $= 1 : U(1)$
   - Count how many entries $= 2 : U(2)$
   - $\vdots$
   - Count how many entries $= k : U(k)$
2. Create $V = [U(1), U(1) + U(2), U(1) + U(2) + U(3), ..., U(1) + U(2) + \cdots + U(k)]$.
3. - Copy entries $= 1$ to final array $W(1), ..., W(U(1))$.
   - Copy entries $= 2$ to final array $W(U(1) + 1), ..., W(U(1) + U(2))$.
   - $\vdots$
   - Copy entries $= k$ to final array $W(U(1) + \cdots + U(k-1) + 1), ..., W(U(1) + \cdots + U(k))$.

We have thus the following relations (for $i = n, ..., 1$):

$$\begin{aligned} W(V(T(i))) &= T(i) \\ V(T(i)) &= V(T(i)) - 1 \end{aligned}$$

Complexity The worst case complexity of this algorithm is $\mathcal{O}(k+n)$, computed as follow:

$$\left.\begin{aligned} \text{2 passes on T: } \mathcal{O}(n) \\ \text{Creation of U: } \mathcal{O}(k) \end{aligned}\right\} \mathcal{O}(k+n)$$

Remark Often an entry is just a key to satellite data (a file, a picture, a book, etc.). In those cases, CountingSort is stable: the order of two entries with the same key values is preserved. They are not swaped in the final array W.