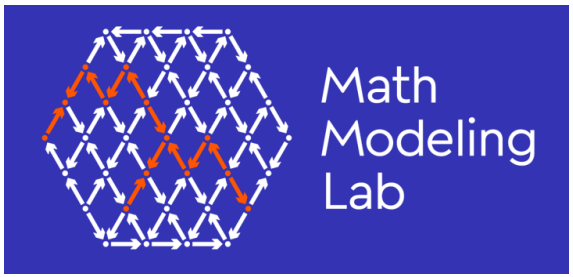
Lecture 3. Data structures with a direct access.

Dmitri Shmelkin

Math Modeling Competence Center at Moscow Research Center of Huawei
Shmelkin.Dmitri@huawei.com

February 10, 2023



Overview

Data Structures: what and why?

When designing an algorithm, an Algorithm Designer needs to think about, how to store data efficiently?

Example Element query in Lecture 1 is based on a sorted array. Binary Search works efficiently thanks to the ability to find the median element in $O(1)$ time, which is ensured by (1) the ascending order and (2) the direct access to elements by index.

An algorithm designer applies the following **selection criteria** for data structures:

A) Efficiency to facilitate the main query. Designer needs to think about the priorities, none structure is good for all purposes

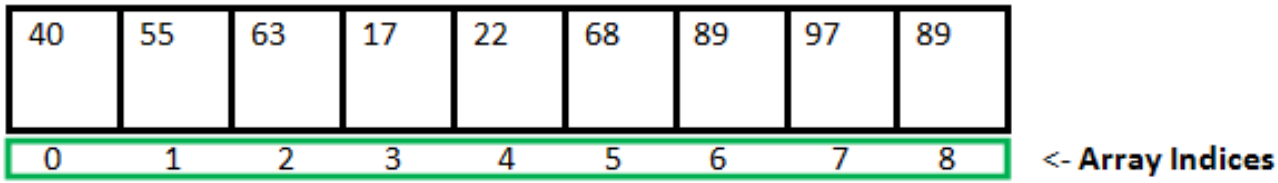
B) The size of required memory as function in n, the size of input. If $n >> 0$, $\Theta(n^2)$ memory is rarely accepted (Range query with $O(n^3)$ memory in Lecture 1).

C) If the data is not very stable, the complexity of Insert and Delete functions, and moreover, if A) is guaranteed by some property or invariant, then the complexity to restore this invariant after Insert/Delete

D) in many cases, to reach A), a precomputation is needed, it may need a considerable effort, for example, $O(n \log(n))$ to sort numbers in the array.

Static Array

The simplest of all structures is Array, which is also the basic type of C++ and other type aware languages



• In computer memory, an array occupies a continuos sequence of entries of the size needed to store an element of required type
• The memory is allocated, then the array entries are available to assign a value of necessary type (don't forget **initialization**!).
• Advantage: direct access with $O(1)$ complexity.
• Drawback: flexible extension is impossible, so the size of array is to be estimated or forcasted. If in different scenarios the size is different, it is usually handled by allocating the maximal size always, which may be not efficient.

Conclusion Static arrays are good for small/average size data with the size either a priori known or de facto having small variance.

Dynamic Array or Vector

• To overcome the drawbacks of static array, a dynamic array is often proposed and supported in different languages and libraries, such as std::vector in C++.
• Vector is a container that allows for a direct access via an operator [ ], similar to a static array and also enocurages using iterator for sequential access.
• Vector allows for adding a new element to the end of the array with **pushback** function. How does it work?
• When an arrray of a type is claimed, a minimal size $S = s_{min}$ is allocated
• When the size is (almost) fully utilized, set $S := 2S$ and a new continuos segment of size equal to S is allocated, array elements relocated to new positions, and the old memory is released.

Theorem 1

Consider an utilization of vector such that n elements are added to it, sequentially. Then average complexity of adding one element is $O(1)$.

Proof of Theorem 1

Proof: For simplicity, first assume that $n = 2^k s_{min}$, hence, while adding n elements relocation is done k times with sizes $s_{min}, 2s_{min}, \ldots, 2^{k-1}s_{min}$. So totally, $s_{min}(2^k - 1) = n - s_{min}$ elements are relocated, or < 1 time per each of n elements. In general, we find k such that $2^{k-1}s_{min} \leq n \leq 2^k s_{min}$ and have < 2 relocations per 1 element. □

Remark: The approach of Theorem, when we account for an average complexity of operations, which are periodically done, is called **amortized analysis**. We will see more examples of this sort below.

• Besides pushback, Vector allows for adding and deleting to a middle with the following obvious drawback:

Theorem 2

Deleting the m-th element in a vector with n elements and insertion to m-th position have both complexity $O(n - m)$.

Stack

• Stack is a container that allows for 2 operations, **Push** to the top and **Pop** from the top, and Pop is not applicable if the stack is empty, so a Boolean function **IsEmpty** is needed to avoid **underflow**.

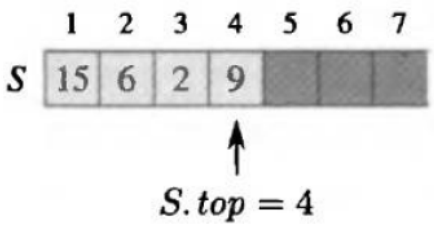
• Historically, stack appeared after Alan Turing in theories of computers, as a math abstraction of function calls. Now the concept of Call Stack is essential when you debug your program.
• Time Machine of interactive SW with Undo, Redo buttons uses Stack:

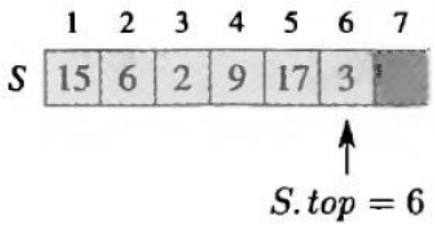
• In Queueing theory stack is known under the name of LIFO, Last-In-First-Out.

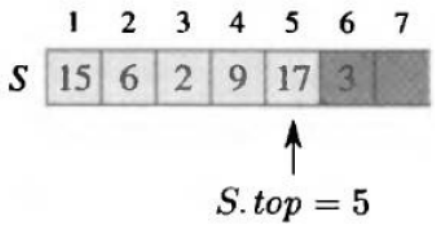
Stack based on Dynamic Array

• Stack can be implemented based on several other basic structures. For example, based on Dynamic array.
• Stack can be thought of as a Dynamic Array S together with an integer value, *top*, of position of the top, initialized as -1.

IsEmpty : *return* $top < 0$
$Push(x)$: $top := top + 1; S[top] := x;$
Pop : *if* (*IsEmpty*) *return* 0;
$top := top - 1;$ *return* $S[top + 1];$

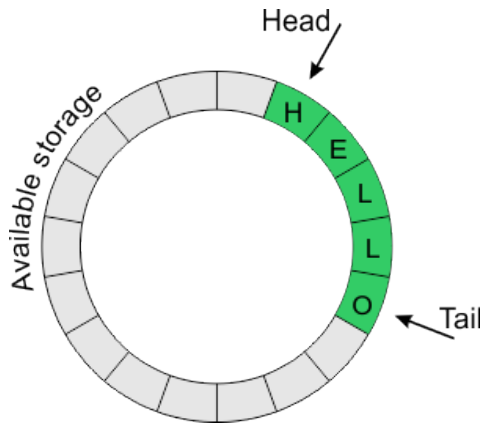






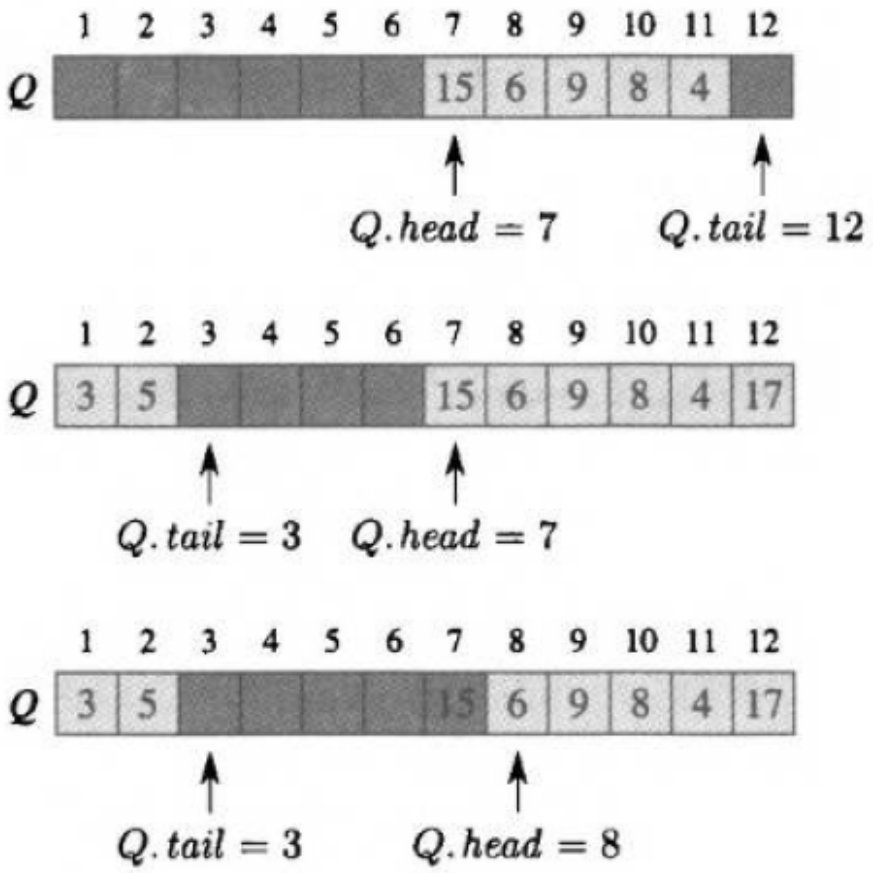
Queue

• What is a Queue is intuitively clear and explained as FIFO, First-In-First-Out.
• Queue allows for **Enqueue** that adds an element to the tail of the queue and **Dequeue** that extracts the element from the head. **IsEmpty** is also useful.
• Analogously to Stack, Queue can be implemented based on a Array.
• Since Queue is often **used in Hardware for packet buffers**, a limited memory size (n items) version is useful. The idea is to think of the array with positions $[1, 2, \ldots, n]$ as a ring or as $\mathbb{Z}/n\mathbb{Z}$, the set of (unusual) residuals $\mathrm{mod}(n)$.
• We may think of the current queue members as an **arc on a ring**; Enqueue and Dequeue operations make this arc rotate clockwise and eventually override the elements, which have been previously Enqueue then Dequeue.
• But the total number of items may not get more than n. Therefore, if there are n of them, then all Enqueue are to be rejected until first Dequeue call.



Queue based on Array

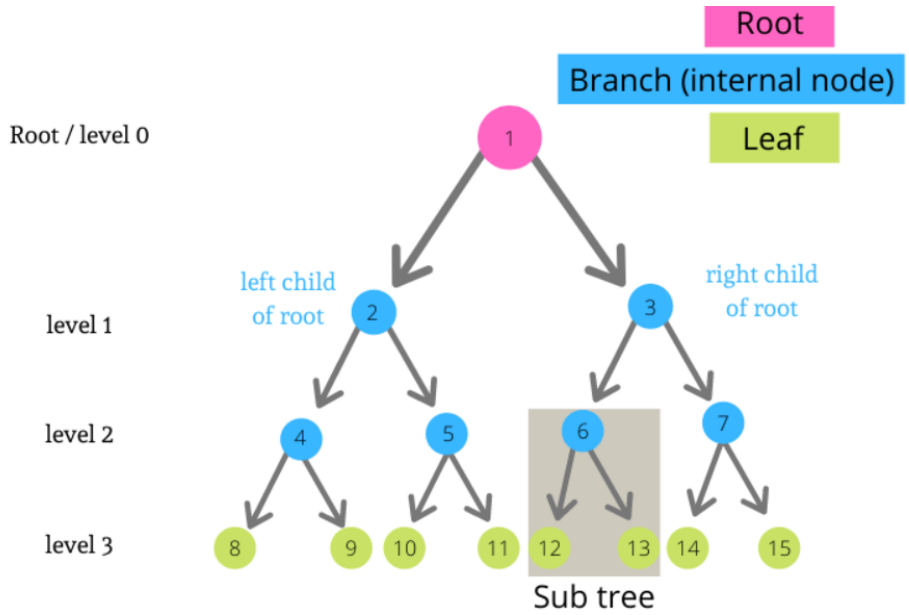
- We allocate an array with n items, and use *head* and *size* initiated both as 0.
- *Tail* is a function: *Tail* : *return* $(head + size) \bmod (n)$.
- *Enqueue*(x) : *if* $(size = n)$ *return*; $D[Tail] := x$; $size := size + 1$;
- *Dequeue* : *if* $(size = 0)$ *return ERROR*; $retval := D[head]$;
  $head := (head + 1) \bmod (n)$; $size := size - 1$; *return retval*;



Binary Tree: definitions

Definition 3

A *binary tree* with *root* is an oriented graph such that the root has no incoming edges and all other nodes have exactly 1 incoming edge (from the *parent*) and at most 2 outgoing edges (to the *children*). The node's *depth or level* is the length of the oriented path from root. Every node gives rise to a subtree with the node as root. A tree is called *balanced* if in every subtree the maximal and minimal depth of leaves difference is 0 or 1.



Tree Data Structure

Heap

• As we saw, a sorted array is an efficient data structure for binary search. But insertion to such an array with the goal to preserve the order takes linear time.
• This contradiction was probably the starting point to invent another internal order in the array, to support the search of Maximum with $O(1)$ time.
• The idea of heap is to enrich the array with the structure of Binary Tree, provided by arithmetic functions:

$$parent(i) = \lfloor i/2 \rfloor, leftson(j) = 2j, rightson(j) = 2j + 1 \quad (1)$$

Remark 1. $parent(leftson(j)) = parent(rightson(j)) = j$
Remark 2. *parent* is the right bit shift $>>$, *leftson* is left bit shift $<<$ in C++, *rightson* is changing last bit after shift. So these operations are bitwise efficient.

Definition 4

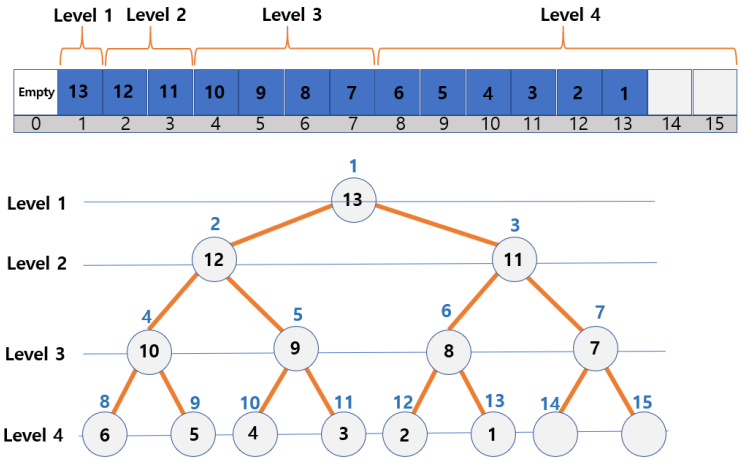
An array A of numbers is a Max Heap if for any i holds $A[parent(i)] \geq A[i]$.

Heap: main properties

• The definition of Heap is clear and natural but it is not clear whether a given set of numbers can be rearranged to meet that property.
• An array D with the functions *parent*, *leftson*, *rightson* comes equipped with the structure of binary tree with $D[1]$ as the root.

Lemma 5

1. *The tree has depth* $\lceil log_2(n) \rceil$, *where* n *is the size of* D.
2. *The tree is full in levels* $1, 2, \ldots, \lfloor log_2(n) \rfloor$, *hence, by* **1**, *it is balanced.*
3. *Let* $D_i \subseteq D$ *be the subtree with* i *as the root.* $|D_i| > 1$ *if and only if* $2i \leq n$.
4. D *is a heap if and only if* $D[i]$ *is the maximum in* D_i, *for every* i.



Heapify

Lemma 5.4 yields the idea to get heap property recursively:
Heapify: Input is D, i such that $D_{leftson(i)}$ and $D_{rightson(i)}$ meet heap property
1. $l := leftson(i), r := rightson(i), m := i$//check whether the sons contain
2. *if* $(l \leq n$ *and* $D[l] > D[m])$ $m := l$//larger element than i
3. *if* $(r \leq n$ *and* $D[r] > D[m])$ $m := r$//if yes, swap the maximum with i
4. *if* $(m \neq i)$
5. $\quad tmp := D[i]; D[i] := D[m]; D[m] := tmp;$
5. $Heapify(m)$//and call *Heapify* recursively
Output: D_i meets heap property

Analysis of Heapify

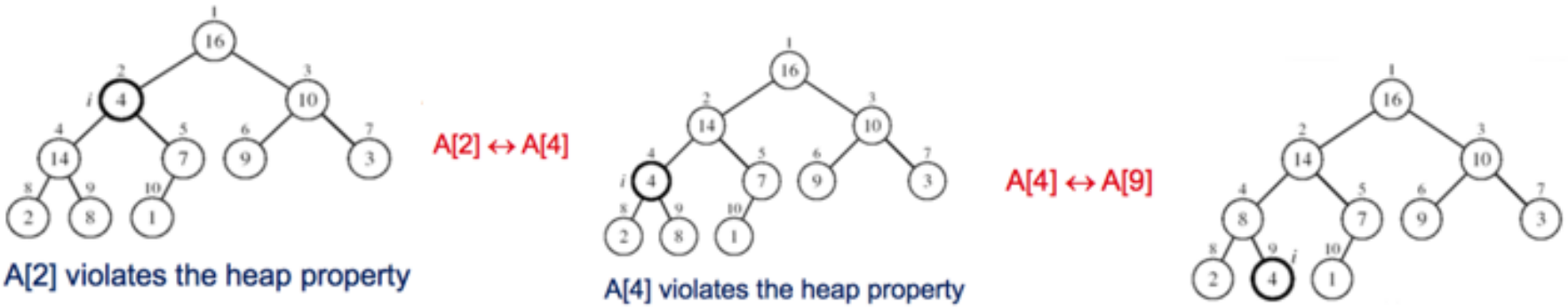
Theorem 6

Heapify(i) *is of complexity* $O(h)$, *where* h *is the height of* D_i, *equal to* $\lceil \log_2(n/i) \rceil$

Proof: *Heapify* is a recursion and the recursive step reduces the problem for the tree D_i of size $m = |D_i|$ to $p = 1$ similar problem of size at most $m/q, q = 1.5$. The local computations take $O(1) = O(m^d), d = 0$ complexity. So appplying Theorem 9 from Lecture 2, we have:

$$\log_q(p) = \log_{1.5}(1) = 0 = d \Rightarrow \ \textit{Heapify} \in O(m^0 \log_2(m)) = O(\log_2(m)) \quad (2)$$

It remains to note that D_i is balanced because D is balanced, hence, $h = \lceil \log_2(m) \rceil$. The formula for h is also clear. □



Building Heap

• *Heapify* makes it possible to rearrange an array to a Heap, in an efficient way: **BuildHeap** Input: array D of size n. Output: D is rearranged to meet Heap property. *for* $i := \lfloor n/2 \rfloor : 1$ $Heapify(D, i)$.

Theorem 7

BuildHeap output meets Heap property and it is of $O(n)$ complexity.

Proof: By 5.3 above, the nodes i with $i > n/2$ are leafs, so corresponding D_i meet Heap property. Then the procedure traverses all nodes and *Heapify* for some node i is called after that for its children, so makes D_i a heap.
Claim: There are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in D_1 tree.
Indeed, we have $\lceil n/2 \rceil$ bound for leafs or nodes of height 0. Then, induction by h.
By Theorem 6, complexity of *BuildHeap* is

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) \in O(n \frac{1}{2} \sum_{h=0}^{\infty} \frac{h}{2^h}) \tag{3}$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h x^h |_{x=0.5} = \frac{x}{(1-x)^2} |_{x=0.5} = 2. \quad \square$$

Using Heap as a Priority Queue: Insert

• We discussed a queue that is organized as FIFO. In many cases, however, we need to decouple the time of arrival and the priority.
• Sorted array requires linear complexity for insert. Heap performs better.
• Priority queue needs to support the operations: *Insert*, *Maximum*, *ExtractMax*.
• By Heap property, *Maximum* just returns $D[1]$.

Insert: Input: array D of size n, which is now a dynamic parameter, and a key k.

1. $n := n + 1, i := n$//initially, place at the last position
2. *while* $(i > 1$ *and* $D[parent(i)] < k)$
3. $\quad D[i] := D[parent(i)]$, $i := parent(i)$.//move $parent(i)$ down and k up
4. *endwhile*;
5. $D[i] = k$;

Output: k moved up to the correct position. D meets heap property.

Lemma 8

Insert has complexity $O(\log_2(n))$.

Using Heap as a Priority Queue: ExtractMax

• *ExtractMax* is also important because after we remove $D[1]$ we need to restore heap property, and this is done with *Heapify*:

ExtractMax Input: a heap D with n elements. Output: $D \setminus D[1]$ is a heap.

1. *if* $(n < 1)$ *return*

2. $D[1] := D[n]$ //swap the last element with the maximum

3. $n := n - 1$ //forget the maximum

4. *Heapify*(1)//restore heap property

Lemma 9

ExtractMax has complexity $O(\log_2(n))$.

Heap Sort

• The procedure *BuildHeap* is an efficient, linear complexity **in-place** algorithm.
• So if we have the array already arranged as a heap, we may efficiently use the heap structure to sort the array in place, as follows:
HeapSort Input: array D of size n. Output: D is sorted in ascending order.
1. *BuildHeap*(D)
2. *for* $i := n : 1$
3. $tmp = D[1]; D[1] = D[n]; D[n] = tmp;$//maximum $\leftrightarrow$ the last element
4. $n := n - 1;$ //exclude the maximum out of the heap
5. *Heapify*(1);//*Heapify* moves the new maximum up to $D[1]$

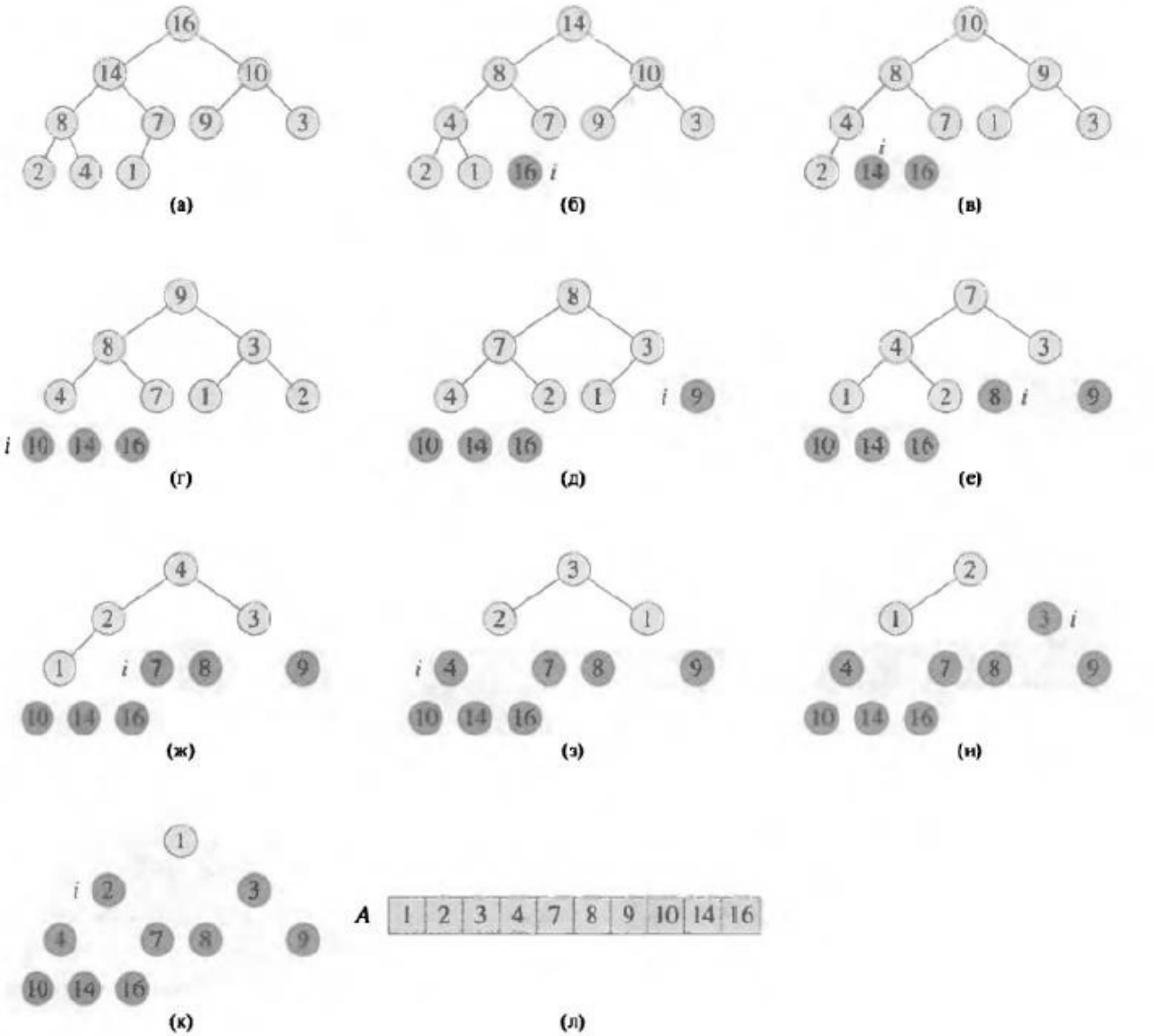
Theorem 10

HeapSort is a correct in-place sorting algorithm of $O(n \log_2(n))$ *complexity.*

Proof *HeapSort* has n steps and by Theorem 6, each step is of $O(\log_2(n))$ complexity. □

Remark: Heap Sort is in-place and this is an advantage vs. Merge Sort.

Heap Sort Example

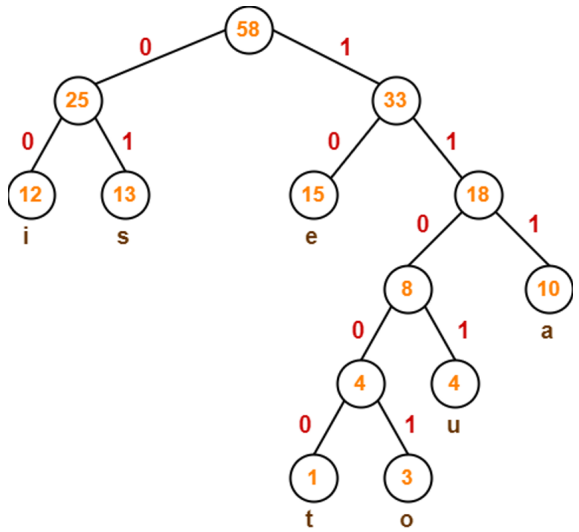


Source Coding

• Data compression or source coding refer to a map taking information source symbols to bits sequences, such that the source symbols can be exactly recovered from the binary bits, with the aim to minimize the size of file.
• Assume that the source is DNA, recorded as a sequence of amino acides, cytosine [C], guanine [G], adenine [A] or thymine [T], so we need to map 4 letters, C,G,A, and T to binary sequences.
• A naïve solution is to use $\{(0,0),(0,1),(1,0),(1,1)\}$.
• If the sequence of symbols is large (DNA contains $> 3 \cdot 10^9$ of them) and the frequencies of different symbols are not equal (at least in some long continuous segments), then it is possible to use 1 bit for the most frequent symbol, and more bits for less frequent ones.
• **Constraint:** the chosen sequences shall not be prefixes of each other (why?)
Example: Assume that the frequencies are 0.5 for C, 0.05 for G, 0.15 for A, and 0.3 for T. Then naïve solution yields $2N$ bits for N symbols. Consider the coding: $C = (0), G = (100), A = (101), T = (11)$. This coding meets the above constraint and uses $(0.5 \cdot 1 + 0.05 \cdot 3 + 0.15 \cdot 3 + 0.3 \cdot 2)N = 1.4N$ bits.

Huffman codes

The above code optimization probem, referred to as Huffman codes, can be formulated as follows. The code comes equipped with the corresponding binary tree (Desicion Tree of decoding algorithm!). Going down the tree, turn left or right means appending 0 or 1, respectively. When coming to a leaf, we print out the corresponding symbol and restart.



Every leaf, $i = 1, \ldots, q$ is assigned with frequency $f(i)$ and depths, $d(i)$, equal to the number of bits used for the leaf. The best tree is to minimize the function:

$$\sum_{i=1}^{q} f(i)d(i). \tag{4}$$

Properties of optimal Huffman trees

Assign the cost $f(n)$ to every node $n \in T$ of the tree as the sum of frequencies of all leaves below the node, as above. Then the cost function (4) is equal to:

$$\sum_{n \in T} f(n). \tag{5}$$

Lemma 11

1. *Except for the leaves every node of optimal Huffman trees has 2 sons.*
2. *The optimal Huffman tree with q leaves has $2q - 1$ nodes.*
3. *There is an optimal tree such that the nodes with 2 minimal frequency values are on the lower level and are sons of the same node.*

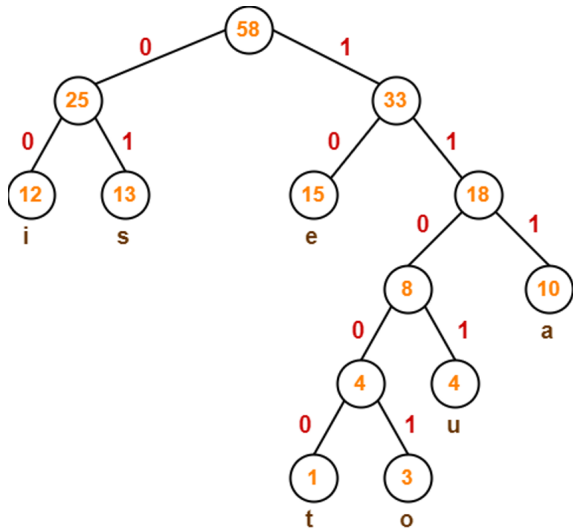
Proof: **1.** if some node at level k has just 1 son, say, left son, then the whole subtree has codes with 0 at $k+1$-th position, so this position can be removed in all leaves of this subtree, leading to decreasing the objective function (4). **2** follows from **1.** **3.** Take an optimal tree and swap 2 symbols with the smallest frequency values with the ones as in the statement **3**, then we get another Huffman tree with the same objective function. □

Reduction of optimal Huffman trees

Lemma 12

Let T_q be an optimal Huffman tree for q symbols that fits the property 11.3. Replace the two leaves like in 11.3 with one leaf, and replace corresponding two symbols with new one, s, with frequency equal to the sum of those for removed two symbols, to get a tree T_{q-1}. Then T_{q-1} is an optimal Huffman tree.

Proof Assume that symbols from T_{q-1} have a Huffman tree with smaller value of objective (5). Take the leaf corresponding to s and add 2 children of it, with obvious changes in symbols. Then we get a Huffman tree better than T_q. □



Building Huffman Tree

Lemma 11 and 12 give rise for the following elegant algorithm for Huffman tree.
BuildHuffmanTree Input: An array f of size n of pairs, symbol and its frequency, future Huffman tree leaves. Output: a Huffman tree with these n leaves.

1. Rearrange f to a Minimum PriorityQueue with *BuildHeap* algorithm
2. *for* $k := n+1 : 2n-1$//each loop adds 1 internal node to the tree
3. $(s_1, f_1) = Min(f)$; $ExtractMin(f)$;//s_1, s_2 are either symbols or empty
4. $(s_2, f_2) = Min(f)$; $ExtractMin(f)$;//for previously inserted internal nodes
5. Make a node $n_k = (\ , f_1 + f_2)$ of Huffman tree with sons (s_1, f_1) and (s_2, f_2)
6. $PriorityQueue.Insert(n_k)$.
7. *endfor*

Theorem 13

BuildHuffmanTree builds an optimal Huffman tree and is of complexity $O(n \log_2(n))$

Exercise: Watch the example on the next page and prove Theorem 13.

Building Huffman Tree Example