AGH
Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Introduction to Computer Science
Lecture 03

Version: 2023

Marek Wilkus Ph. D. http://home.agh.edu.pl/~mwilkus
Faculty of Metallurgy and Industrial Computer Science
AGH UST Kraków

Algorithms

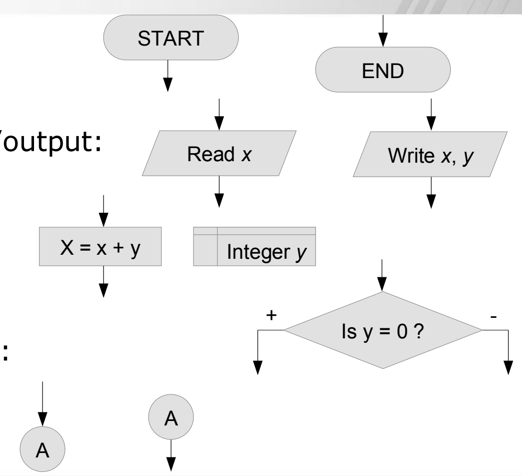
- An **Algorithm** – A finite set of definitive instructions used to solve a class of problems or to perform specific computation.
  - Finite – must have an end resulting in an output. (compare with its implementation!)
  - Definitive – each step is precisely stated.
  - Computable – each step can be carried out by a machine executing it.
- **Algorithm correctness** means that a specific algorithm:
  - Will finish with a correct result,
  - With correct input data, will always finish.

Algorithm description

- Block diagram,
- Pseudocode,
- Programming language code,
- Formulation,
- (precise!) description,
- ...

Block diagrams

- Start, End:
- Data input/output:
- Operation:
- Conditional:
- Label:

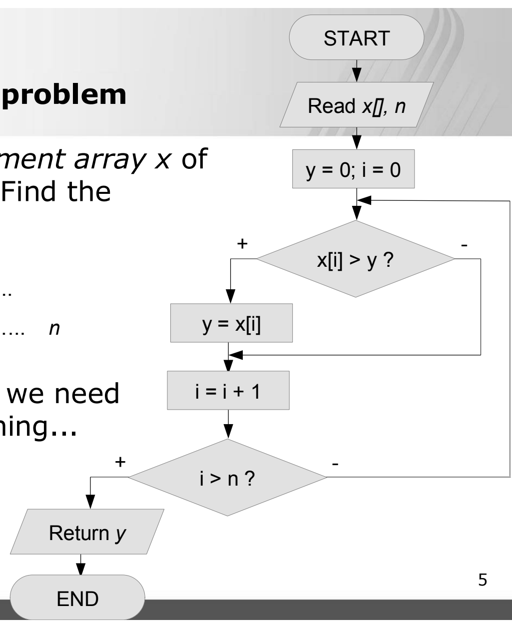


A simple problem

- We have an *n-element array x* of positive integers. Find the largest one.

17	2	26	0	10	...	
0	1	2	3	4	.....	*n*

- Looks simple, but we need to assume something...

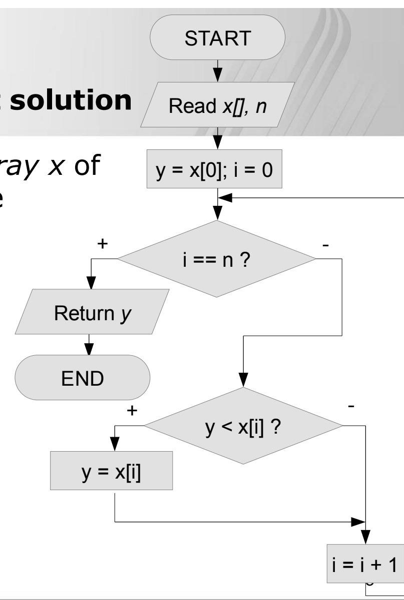


A simple problem
But more elegant solution

- We have an *n-element array x* of positive integers. Find the largest one.

17	2	26	0	10	...	
0	1	2	3	4	.....	*n*

- Notice this will:
  - Work on empty arrays,
  - Use less assumptions.



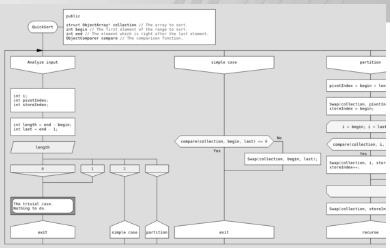
Flowcharting

- It has still some ambiguity related to blocks contents.
- Some pseudocode (or maths) may "leak" to blocks.
- Allows to define program calls.
- If the flowchart is too big to print on page, we can use labels to split it.
- ...or we can use labels to logically split complex algorithm to smaller "ways".

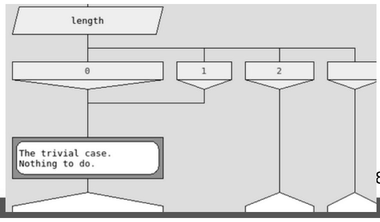
- Loops look quite poor on them.
- Case-type conditionals look even worse
  - Usually, some kind of conditional is improvised for it.

A brief fixes for flowcharting

- If a lot of loop is needed, some description systems (e.g. DRAKON) supply "loop blocks".



- If a switch is needed, some description systems supply just branching of the line.



Pseudocode

- Allows to quickly describe actions.
- Properly written, can be easier to implement.
- Allows to skip unneeded implementation details.
- ...but leaves significant ambiguity.

- So it is needed to maintain an unified, programming-language bias.

Algorithms complexity

- The algoritam working on a specific data executes the specific operations on them.
- But what happens if the amount of data changes?

- The amount of resources (computational: CPU, or CPU's time, or memory) changes.
  - There is a **computational** complexity,
  - And **memory** complexity.

- The change depends on a specific algorithm.

Which is the "specific operation"?

- The operation directly interfacing with the data, key to the algorithm operation.

- In sorting algorithm, it's comparing of sorted values and changing their positions.

- In integration or optimization algorithms, it can be calculating the function's values at a specific point.

- In parameters identification, it's comparing the verified function to the source.

Big O notation

- As the input data grows (n), the computational requirement (O) grow accordingly.
- The function describing dependency between input data vs computational requirements can be written in the Big O notation:

  - $O(1)$, $O(n^2)$, $O(n)$, etc.

A formal definition of Big O notation:

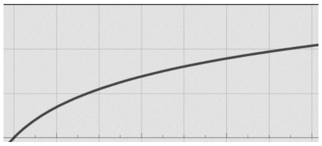
- The function *f* is at most O(*g*(*n*)) if and only if there exist $n_0>0$ and $c>0$ constants so that for every $n \geq n_0$: $f(n) \leq c*g(n)$
- Domain of these nonnegative-valued functions are nonnegative integers.

A typical O functions

- O(1) – the algorithm has a constant requirement for resources, nevertheless of the input.
- Example:
  - Obtaining the *n*-th value in the array.
  - Finding is a number even or odd.
  - Determine the number of diagonal lines of the convex polygon having *n* vertices (which is ½(n*(n-3))).

A typical O functions

- O(log n) – the value goes with the log (by default $\log_2$) curve.
- Such algorithms are definitely very efficient.



- Example:
  - Find the item of the value *x* in a sorted array by binary search.
    - Details: Divide the array by 2, then look into which half to perform this again until we get the element we're looking for.

A typical O functions

- O(n) – the complexity grows linearly with number of input data.
- Examples:
  - Find maximum element of an n-number array.
  - Calculating statistics of a text string.
  - Any algorithm that iterates in an entire input data set.

A typical O functions

- O(n log n) – linear-logarithmic complexity.
- For smaller data sets, behaves like linear.
- Then, it grows significantly.
- Example:
  - Many algorithms which divide problem and perform operations on the divided parts,
  - Quick sort, Merge sort.

A typical O functions

- $O(n^x)$ where x is 2, 3, etc. - Quadratic (n^2), Cubic (n^3).
- Examples:
  - Nested loops iterating over entire data sets.
  - Bubble sort,
  - Find duplicate items in an array (the most simple version).

A typical O functions

- $O(2^n)$ – Exponential curve
- As can be imagined, this is not an effective approach.

- Examples:
  - Obtain all subsets of n-element set.

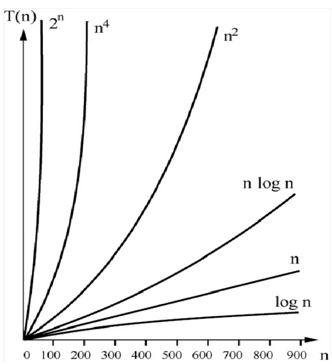
A typical O functions

- O(n!) - Factorial – Each additional data record causes the number to increase significantly.
- Definitely not an effective approach.

- Examples:
  - Finding all permutations of a given n-element data.

Summing up

- Solving a problem, it is important to develop an algorithm that has the lowest growth of computational/time complexity.

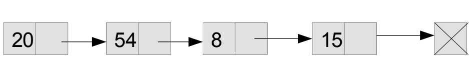


Simple data structures

- The **array** is the set of data which has a **pre-defined size** ans is accessed by an **index**. → random access.
- The **stack** and **queue** are already known structures which have a specific access methods.
- The **list's elements**, contrary to the array, can be freely arranged and moved, but the access is **linear** – there is no way to access *n*-th element without accessing *n-1*-th. There is **no size limit** except the memory limit of the program/machine.

The linked lists

- A list's **item** consists of its value and the **pointer to the next item**.
- The last item has its pointer pointing to nullptr (or other predefined value) – then we know that we have reached the end of list.
- We cannot access it by the index – only sequentially. (not without cheating)



Adding an item

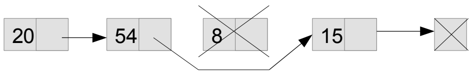
- Just point the item's pointer to the first item of the list, now point the first item's pointer to the item recently added.

- Pseudocode (**next** is the pointer, **ListFirstItem** is the starting item pointer):

  New_item.next=ListFirstItem
  ListFirstItem = *New_item

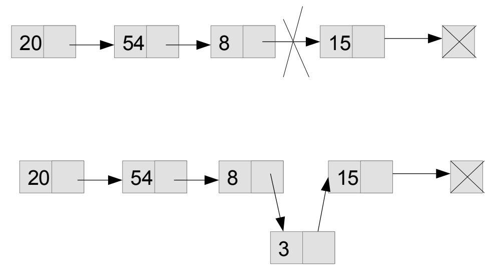
Deleting an item

- To delete the item:
  - To delete the **first** item, just point the list pointer to the first item's **next** pointer and dispose the memory from the item omitted.
  - To delete the **n-th** item, go to its **previous** item and put the pointer to its **next** item to the next item of the deleted item. Now, dispose the deleted item.
  - To delete the **last** item, put the null value to the pointer of the one but last item. Then dispose the last item.



Inserting an item

- To insert the item at the n-th place, go to the n-1 item and save its **next** pointer in the new item.
- Now, set its **next** pointer to the new item.



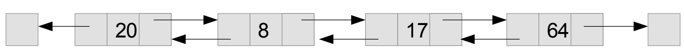
Doubly-linked list

- It can be seen that it is impossible to move back in the single-linked list.
- Applications (like FAT filesystem) use buffering and look-up tables to support easier navigation in these lists.
- Is it possible to modify the structure so it's possible to **go back**?

- It is possible to **add a "previous"** pointer pointing at the previous item.

Doubly-linked list

- Each item is made of its data, the **next** pointer and the **previous** pointer.



- Notice that it is possible to interpret this as **two singly-linked lists**, but one is processed in one direction, while the other in the reverse.
  - One list uses **next** pointers only, while the second uses only **previous**.
  - ...but it is possible to switch dynamically between directions!

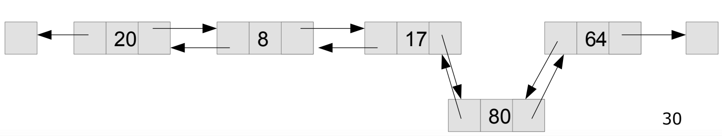
Doubly-linked lists – typical operations

- Traversal: Can be done forwards and backwards.
  - ...so the starting point can be the first or the last item.
  - It is always possible to go item→previous.

- Appending an item:
  - Create a new node with two nullptrs as pointers,
  - Append the node's pointer to the first/last nullptr of the list,
  - Modify the according pointers of the structure to point into the former first/last item.

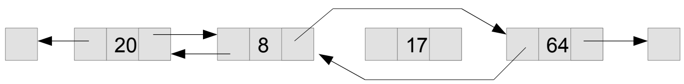
Doubly-linked lists – typical operations

- Inserting a node to the specific point (after the insertion item)
  - Determine the insertion item,
  - Create a new item, its pointers are nullptr.
  - NewItem→next = insertionItem→next
  - NewItem→previous = insertionItem
  - InsertionItem→next→previous = NewItem
  - InsertionItem→next = newItem



Doubly linked lists – typical operations

- Delete the item:
  - deletetItem→previous→next = deletedItem→next
  - deletedItem→next→previous = deletedItem→previous



Circularity

- Both singly and doubly linked lists can be implemented **circular** way.
- Then, the starting pointer should be present in the code.
- It can be implemented as a ring buffer (with a singly-linked lists) or as a both FIFO and LIFO buffer (with doubly linked list).
- Remember that **removing an item from a circular list** should maintain the circularity **and** a pointer which holds it in the memory.

Structures for easy searching of the data

- Imagine we have an **array** of numbers, but it is sorted increasing way:

0	1	2	4	5	7	16	18	22	23	28	29	30	32	33	36	38	40	45	47	50
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

- Is it possible to quickly find where is the specific number?
- If the array is not sorted, we would have to compare each value until the specific one is found.
- If it is sorted, we can use bisection.

- At which index is the number 16?

0	1	2	4	5	7	16	18	22	23	28	29	30	32	33	36	38	40	45	47	50
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

- We will divide the array to halves:

0	1	2	4	5	7	16	18	22	23
0	1	2	3	4	5	6	7	8	9

28	29	30	32	33	36	38	40	45	47	50
10	11	12	13	14	15	16	17	18	19	20

- The middle (element 10, counting from 0) is **28**. So to find 16, we have to process the **left** part holding **smaller** values.
- Now we will do it again.

- In which place is the number 16?

0	1	2	4	5	7	16	18	22	23
0	1	2	3	4	5	6	7	8	9

- We will divide the array to halves:

0	1	2	4	5
0	1	2	3	4

7	16	18	22	23
5	6	7	8	9

- The middle (element 5, counting from 0) is **7**. So to find 16, we have to process the **right** part holding **larger** values.
- Now we will do it again.

- In which place is the number 16?

7	16	18	22	23
5	6	7	8	9

- We will divide the array to halves:

7	16
5	6

18	22	23
7	8	9

- The middle (element 2, counting from 0) is **18**. So to find 16, we have to process the **left** part holding smaller values.
- Now we will do it again.

- In which place is the number 16?

7	16
5	6

- We will divide the array to halves:

7	16
5	6

- The middle (element 1, counting from 0) is **16**. If we're out of luck, we would have to go the left part.
- We get the index of number 16: 6 counting from 0

Recursion

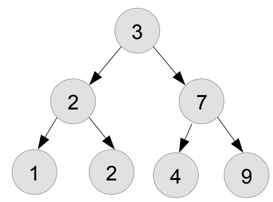
- Notice that we can describe the algorithm with 4 steps:
  0. If the given part of the array is a single number, it is the result (equal or closest), so we end.
  1. Divide the array in half.
  2. Determine the half in which the item to be found may be present.
  3. Do the same thing with this half.
- Which means we can **call the same function** by itself.
- The algorithm that calls itself is a **recursive** algorithm.
- And these algorithms must have a clear stop condition.

Conclusions, search trees

- So instead of 7 comparisons (pessimistic 21), it was possible to do it in 4 (pessimistic 5).
- For faster searching, it is possible to use a specific structures for storing ordered data.
- One of such structures can be a **search tree**.
- Its items, or **nodes**, have data and two pointers.

Binary search tree

- Every node has two pointers: **left** and **right**.
- Such node can be presented as a hierarchical tree.
- The items are ordered the way that **all items on the left side** of the node have **smaller** values than this node's value.
  For the larger values, it's the right node.



Pointers not shown are nullptr.

Binary search tree

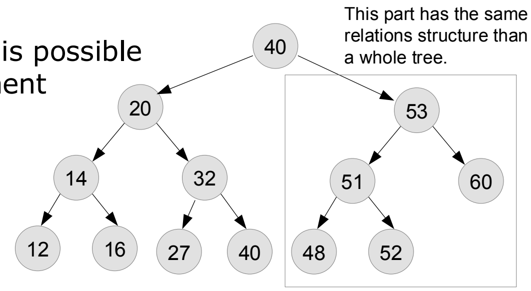
- In the implementation, BST nodes may also have an **up** pointer which points to the parent node.
- ...and they store more data in additional fields.
- Now, to find a node of specific number, we start from the **root**.
  - If the root's value is smaller than the value we're looking for, we choose the **right** node to find larger values. Else - the left node.
  - And now we are doing the same thing with the new node.
  - We end this when the value is found…
  - ...or we land with a **leaf** – a node with both child nullptrs.

Shortcuts

- To get a minimum value, go left until we cannot go anymore.
- To get a maximum – go right.
- ...and we don't have to compare anything!
- To get the minimum/maximum value from a range, compare until we reach the root of the specific range and then go left/right accordingly without comparing.

A sub-tree property

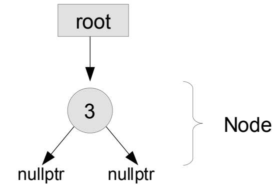
- In a properly made BST, a sub-tree has the same relations between numbers as the whole tree. It means that a sub-tree can be considered a separate tree.
- It means that it is possible to easily implement BST operations using recursion.



Implementation: data structures

```
struct Node {
    int data;
    Node* left;
    Node* right;
};

Node * root=nullptr;
```



A tree of 1 node.

Adding the data to BST

- The **add** function is **recursive**.
  1. If the parent is nullptr, we create a new node of a specific value to it and return it.
  2. If the parent's data is larger than value, we have to insert the smaller item to the **left** side. So we **add** the node to the parent→left.
  3. Else, we **add** the node to the parent→right.
- We always return the pointer to the parent node.

Finding the minimum value

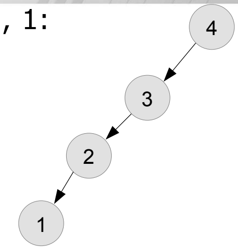
- A recursive way:
  1. If current node is nullptr, we return nullptr (empty tree).
  2. If current node's left branch is nullptr, we cannot go left anymore → we return current node as it is minimum.
  3. Else, we find the minimum of the node to the left.

Traversal

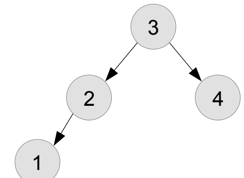
- To traverse the BST in ascending order:
  1. If the given parent is null, abandon traversal.
  2. Traverse the left pointer as a parent.
  3. Raturn the parent's data.
  4. Traverse the right pointer as parent.

A specific case: Is it still a tree?

- Let's add these items in order: 4, 3, 2, 1:
  – The root will be 4,
  – Left of it, 3 will get attached.
  – To the left, 2 will be attached.
  – And finally 1, to the left of 2.
  – So it starts to look like a singly linked list, not a tree.



- We have to **balance** the BST:



Balanced BST

- In the balanced BST, the height of left and right sub-tree of any node is the same or different by 1.
- When we insert, delete or search for a node, in unbalanced BST we may run into a O(n) complexity (like a linked list). Balanced BST guarantees O(log n) complexity.
- In some implementations, balancing BST may impact efficiency during insertion, but make things faster during acquiring of information.

Is the tree balanced?

- We measure the height of both sub-trees. The absolute difference between heights of left and right sub-trees must be less than 1.
- The node without leaves is balanced.
- For each node, its left sub-tree must be balanced.
- ...and the right sub-tree must be balanced too.

- This can be implemented recursively:
  - If we got a null root → height = 0, balanced.
  - Check is the left subtree balanced. Not → return not-balanced.
  - The same with the right one.
  - If the |leftHeight-rightHeight|>1 → return not-balanced.
  - Return larger of heights+1.

Self-balancing binary trees

- There are a few algorithms that guarantee that the item added to the will mke the tree balanced.
- Usually, some attribute like position of the node or its depth should be stored with node, as computing it all time causes uncontrollable increase of complexity.
- Example: AVL Trees.

AVL Tree

- Every node has its balance factor BF stored in its contents:

  $BF = h_L - h_R$

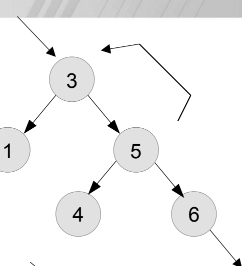
- If BF==0, both sub-trees have equal height. BF==1 - left one is higher, BF==-1 - the right one is higher.
- There must not be any other values of BF than these 3. This assures the balance of the tree.

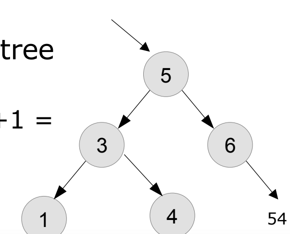
Inserting into AVL

- We start with inserting the value as usual.
- Then, we check are the newly calculated BF values correct (in [-1, 0, 1] set). If not - we need to **rotate** the tree elements.
- There are 4 types of rotation:
  - Right-Right
  - Left-Left
  - Right-Left
  - Left-Right

Example: RR rotation

- We will rotate 5 with 3 in a part of the tree presented in the picture.
- BF of 5 is -1.
- But BF of 3 is **-2**. ←WRONG!
- The height of 5’s left subtree is 1 bigger than right one.
- The height of the 5’s tree is $\max(h_L,h_R)+1$, which is right subtree height+1.
  - ...so it’s left subtree’s height +1+1 = h_L+2.
  - Let this hL be now called **h.**

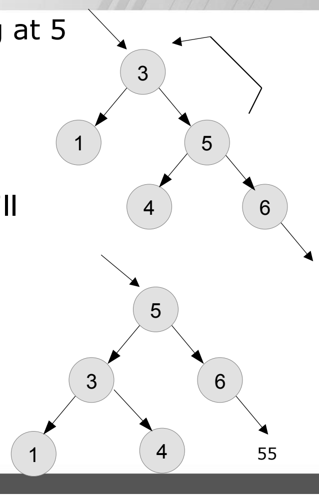




Example: RR rotation (2)

- The height of the subtree starting at 5 it its **h**+2.
- The BF of the 3’s tree is -2 (right subtree is 2 levels higher).

- Now it we replace 5 with 3, we will get:
  - 3’s BF will be 0 (balanced)
  - 5’s BF will be 0 too (balanced)



LL Rotation

- A mirror of the RR rotation.
- Performed for the situations of the opposite unbalancing.

RL Rotation

- If we rotate LL, we can get the leaf in the position in which it will be easier to torare RR.
- It is possible to join these two rotations in a single algorithm.

- Executed when there is an imbalance of 3 consequent (descending) nodes.
- Mirrored version: LR rotation.

So when which rotation?

- Notice that LR and RL rotztions work on **3 consecutive nodes**.

- If there is an imbalance in left subtree’s left subtree, we rotate RR.
- If there’s in right sub-tree’s right subtree, LL.
- If there is an imbalance in the left subtree’s right subtree, LR.
- If in right sub-tree’s left sub-tree, RL.

Summing up

- By increasing complexity of an insertion/deletion operations, we are able to get certain that the search time will be O(log n).
- This way, an efficient data storage can be designed.

- Applications: Operating system queues, indexing large records in databases.

One more application of trees

- Most data is encoded in form of n-bit words, or **bytes**.
- Modern computer systems use 8-bit bytes, rarely 7-bit in data transfer.
- So as it is already known from programming, with 8-bit byte we can encode a number 0..255.
- But if we store **text** in Latin alphabet, we almost never use more than half of this set!
- Is it possible to **store the text more efficiently?**

Selecting the proper encoding

- So, as the Latin alphabet has 26 characters, x2 - upper case, +space, + special characters, we can maybe fit in 7 bits?
- But what if we **dynamically** change the bit width of the character?
  - See the Morse code: The letter **e**, the most frequent in English, is encoded with the shortest signal - a single dot.
- It is possible to develop such encoding for **any** data we have and store the key to decode it in a binary tree.

Using a binary tree

- We construct the binary tree from the incoming data.
- The most frequent characters get the shortest bit lengths.
- The encoding must have a **character uniqueness** because...

Let A=0 and B=01
The bit stream:

```
0101000...
```

Does it start with A or B?

Huffman algorithm

- Let’s simplify it to a known text. We have a text made of non-unique characters:

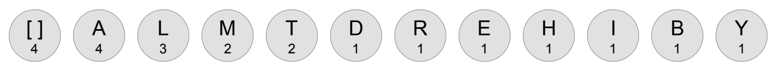
```
MARY HAD A LITTLE LAMB
```

- We will write down the unique characters and their count in this 22-character (176 bit) line.
- Then, by dividing the count by the total count, we will get the probability of encountering the specific character.

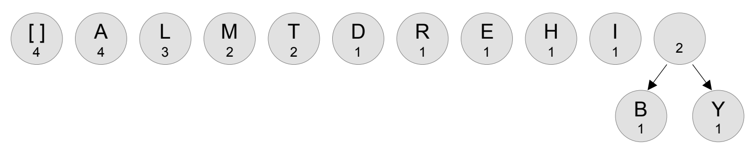
Letter	Count	Probability
M	2	0.09
A	4	0.18
R	1	0.04
Y	1	0.04
[space]	4	0.18
H	1	0.04
D	1	0.04
L	3	0.14
I	1	0.04
T	2	0.09
E	1	0.04
B	1	0.04

Huffman algorithm (2)

- Now, let’s build a tree items using their count/probability:

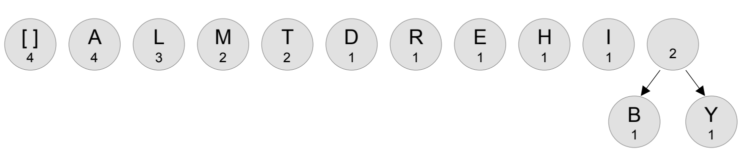


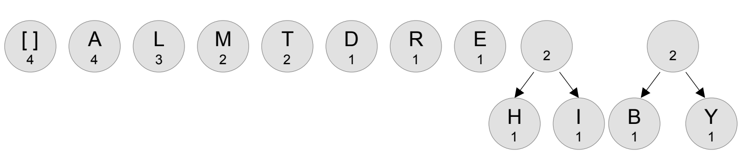
- We will start with the two most rare characters and bind them with a value-less tree node of frequency 1+1=2:



Huffman algorithm (3)

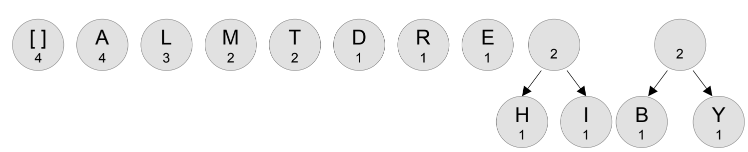
- Let’s now do this again: Link two most rare characters with a value-less tree item

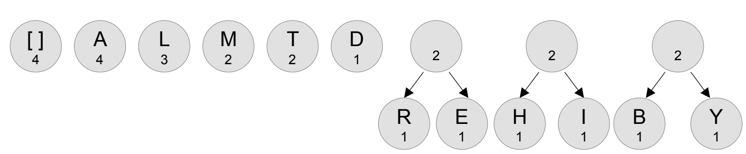




Huffman algorithm (4)

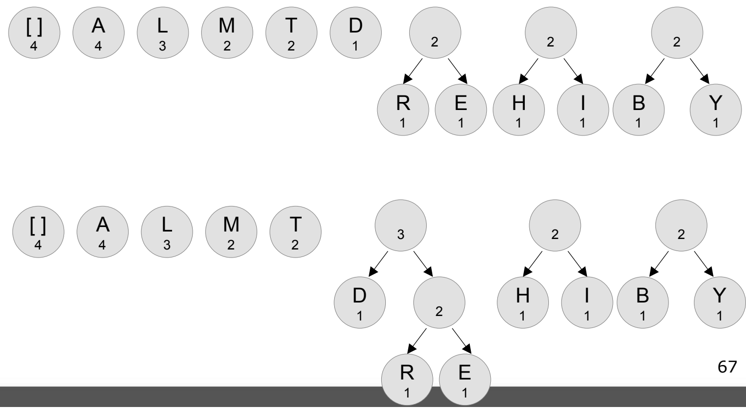
- Let’s now do this again: Link two most rare characters with a value-less tree item





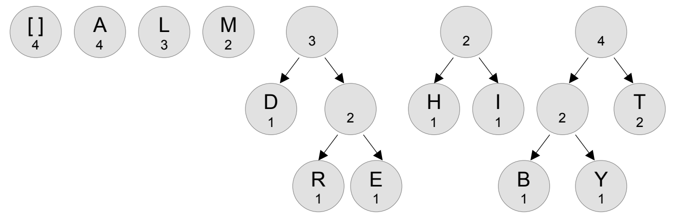
Huffman algorithm (5)

- ...and again...



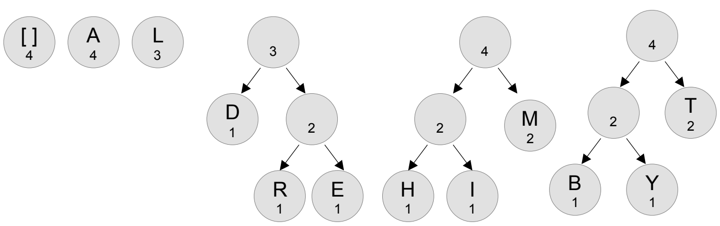
Huffman algorithm (6)

- I took a T now to have a nicely balanced tree...



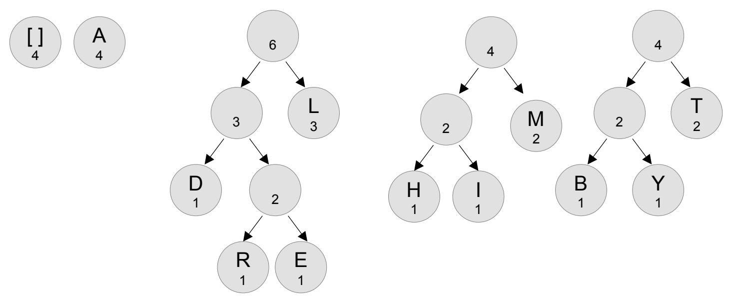
Huffman algorithm (7)

- Now the most rare are M and H+I...



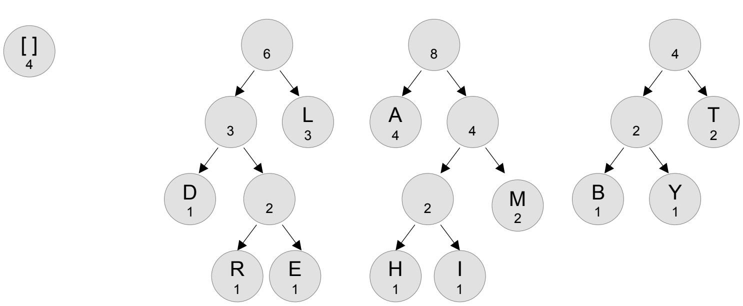
Huffman algorithm (8)

- Now the most rare are L and D+E+R...



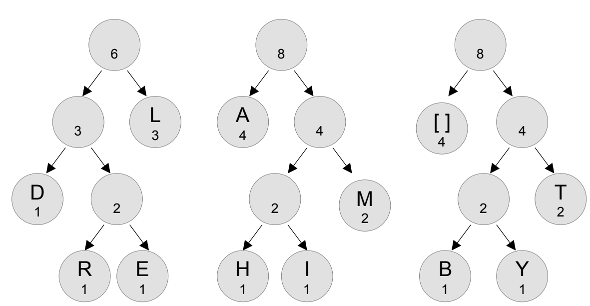
Huffman algorithm (9)

- Now the most rare are A and M+H+I...



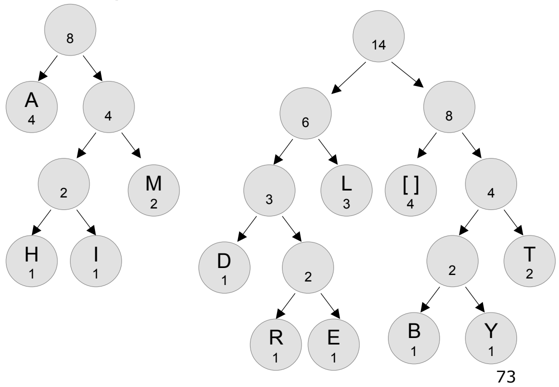
Huffman algorithm (10)

- Now the most rare are space and T+B+Y...



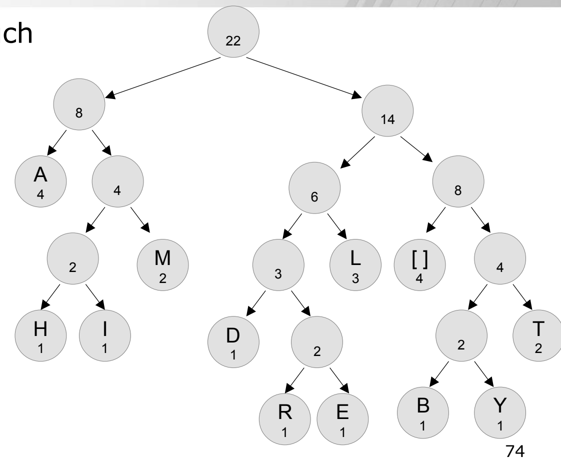
Huffman algorithm (11)

- Now the most rare are L+D+R+E and, equally to the other, this one with a space...



Huffman algorithm (12)

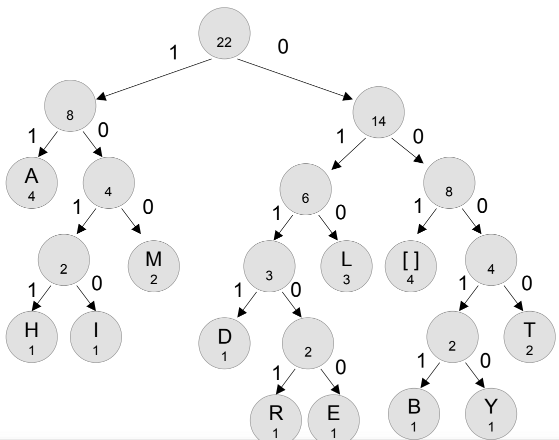
- There isn't much choice now...



The result and how to use it

- Now: Going left = 1, going right = 0.

- M=100
- A=11
- R=01101
- Y=00010
- [ ]=001
- H=1011
- D=0111
- L=010
- I=1010
- T=0000
- E=01100
- B=00011



The result:

- 76 bits:
  1001101101000100011011110111001110010101010000000000010011000010101110000011

- We decode the first encountered valid sequence (we cannot go anymore in the tree).
- Then we go again from the root.

- This is a fully operating compression method used in practical applications - as one of the algorithms in Deflate (ZIP) or in one of JPEG encoding stage.

Thank you for attention