Algorithms and Data Structures

Text processing and data compression

dr Szymon Murawski

Comarch SA

June 6, 2019

Table of contents I

Course plan

Accessing data in array

0	1	2	3	4	5	6	7	8	9	10
Jan	Tim	Mia	Sam	Leo	Ted	Bea	Lou	Ada	Max	Zoe

Fetching data from array

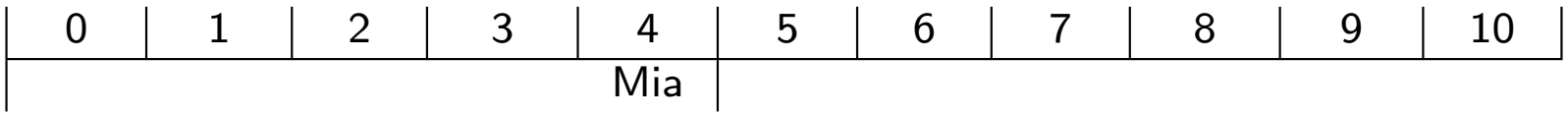
- If we know the index, then fetching data from array is very fast and is independent on array size
- If we don't know the index, then we need to do liner search of the array, which is slow and scales linearly with array size
- Let's create a function $h(k)$, that given a key k would return an index for that key. For example $h(Ada) = 8$
- That function is called hash function, and the data structure is called hash table

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4



Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1

0	1	2	3	4	5	6	7	8	9	10
	Tim			Mia						

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim			Mia						

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim			Mia	Zoe					

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Jan $\rightarrow$ 74 + 97 + 110 = 281 $\rightarrow$ 281 mod 11 = 6

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim			Mia	Zoe	Jan				

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Jan $\rightarrow$ 74 + 97 + 110 = 281 $\rightarrow$ 281 mod 11 = 6
- Ada $\rightarrow$ 65 + 100 + 97 = 262 $\rightarrow$ 262 mod 11 = 9

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim			Mia	Zoe	Jan			Ada	

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Jan $\rightarrow$ 74 + 97 + 110 = 281 $\rightarrow$ 281 mod 11 = 6
- Ada $\rightarrow$ 65 + 100 + 97 = 262 $\rightarrow$ 262 mod 11 = 9
- Leo $\rightarrow$ 76 + 101 + 111 = 288 $\rightarrow$ 288 mod 11 = 2

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Leo		Mia	Zoe	Jan			Ada	

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Jan $\rightarrow$ 74 + 97 + 110 = 281 $\rightarrow$ 281 mod 11 = 6
- Ada $\rightarrow$ 65 + 100 + 97 = 262 $\rightarrow$ 262 mod 11 = 9
- Leo $\rightarrow$ 76 + 101 + 111 = 288 $\rightarrow$ 288 mod 11 = 2
- Sam $\rightarrow$ 83 + 97 + 109 = 289 $\rightarrow$ 289 mod 11 = 3

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Leo	Sam	Mia	Zoe	Jan			Ada	

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Jan $\rightarrow$ 74 + 97 + 110 = 281 $\rightarrow$ 281 mod 11 = 6
- Ada $\rightarrow$ 65 + 100 + 97 = 262 $\rightarrow$ 262 mod 11 = 9
- Leo $\rightarrow$ 76 + 101 + 111 = 288 $\rightarrow$ 288 mod 11 = 2
- Sam $\rightarrow$ 83 + 97 + 109 = 289 $\rightarrow$ 289 mod 11 = 3
- Lou $\rightarrow$ 76 + 111 + 117 = 304 $\rightarrow$ 304 mod 11 = 7

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou		Ada	

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Jan $\rightarrow$ 74 + 97 + 110 = 281 $\rightarrow$ 281 mod 11 = 6
- Ada $\rightarrow$ 65 + 100 + 97 = 262 $\rightarrow$ 262 mod 11 = 9
- Leo $\rightarrow$ 76 + 101 + 111 = 288 $\rightarrow$ 288 mod 11 = 2
- Sam $\rightarrow$ 83 + 97 + 109 = 289 $\rightarrow$ 289 mod 11 = 3
- Lou $\rightarrow$ 76 + 111 + 117 = 304 $\rightarrow$ 304 mod 11 = 7
- Max $\rightarrow$ 77 + 97 + 120 = 294 $\rightarrow$ 394 mod 11 = 8

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	

Hash table example

Hash function

Our hash function h will work in the following way: for every character in key calculate it's ASCII code, add those together and divide by modulo size of the array

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Jan $\rightarrow$ 74 + 97 + 110 = 281 $\rightarrow$ 281 mod 11 = 6
- Ada $\rightarrow$ 65 + 100 + 97 = 262 $\rightarrow$ 262 mod 11 = 9
- Leo $\rightarrow$ 76 + 101 + 111 = 288 $\rightarrow$ 288 mod 11 = 2
- Sam $\rightarrow$ 83 + 97 + 109 = 289 $\rightarrow$ 289 mod 11 = 3
- Lou $\rightarrow$ 76 + 111 + 117 = 304 $\rightarrow$ 304 mod 11 = 7
- Max $\rightarrow$ 77 + 97 + 120 = 294 $\rightarrow$ 394 mod 11 = 8
- Ted $\rightarrow$ 84 + 101 + 100 = 285 $\rightarrow$ 285 mod 11 = 10

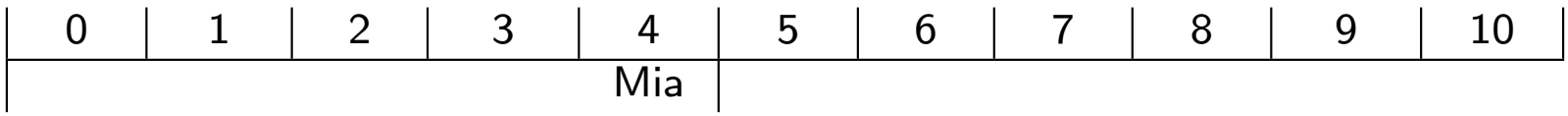
0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted

Hash functions

- Hash function applied to key transforms it into an address
- If key is composed of characters, usually we take the ASCII values of those characters
- Good hash function must have the following properties:
  - **Determinism** - for a given input value it must always generate the same hash value
  - **Uniformity** - output values must be distributed evenly over the output range
  - **Defined range** - output should always be in a range given by user
  - **Collision minimalization** - there should be minimal number of collisions
  - **Speed** - calculating hash should be fast
- Hash functions, apart from being used in hash tables, are also used in cryptographic and checksum algorithms, however for those applications they must have some additional properties

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4



Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1

0	1	2	3	4	5	6	7	8	9	10
	Tim			Mia						

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim			Mia						

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim			Mia	Zoe					

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Sue $\rightarrow$ 83 + 117 + 101 = 301 $\rightarrow$ 301 mod 11 = 4

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim			Mia	Zoe	Sue				

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Sue $\rightarrow$ 83 + 117 + 101 = 301 $\rightarrow$ 301 mod 11 = 4
- Len $\rightarrow$ 76 + 101 + 110 = 287 $\rightarrow$ 287 mod 11 = 1

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Len		Mia	Zoe	Sue				

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Sue $\rightarrow$ 83 + 117 + 101 = 301 $\rightarrow$ 301 mod 11 = 4
- Len $\rightarrow$ 76 + 101 + 110 = 287 $\rightarrow$ 287 mod 11 = 1
- Moe $\rightarrow$ 77 + 111 + 101 = 289 $\rightarrow$ 289 mod 11 = 3

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Len	Moe	Mia	Zoe	Sue				

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Sue $\rightarrow$ 83 + 117 + 101 = 301 $\rightarrow$ 301 mod 11 = 4
- Len $\rightarrow$ 76 + 101 + 110 = 287 $\rightarrow$ 287 mod 11 = 1
- Moe $\rightarrow$ 77 + 111 + 101 = 289 $\rightarrow$ 289 mod 11 = 3
- Lou $\rightarrow$ 76 + 111 + 117 = 304 $\rightarrow$ 304 mod 11 = 7

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Len	Moe	Mia	Zoe	Sue				

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Sue $\rightarrow$ 83 + 117 + 101 = 301 $\rightarrow$ 301 mod 11 = 4
- Len $\rightarrow$ 76 + 101 + 110 = 287 $\rightarrow$ 287 mod 11 = 1
- Moe $\rightarrow$ 77 + 111 + 101 = 289 $\rightarrow$ 289 mod 11 = 3
- Lou $\rightarrow$ 76 + 111 + 117 = 304 $\rightarrow$ 304 mod 11 = 7
- Rae $\rightarrow$ 82 + 97 + 101 = 280 $\rightarrow$ 280 mod 11 = 5

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae		

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Sue $\rightarrow$ 83 + 117 + 101 = 301 $\rightarrow$ 301 mod 11 = 4
- Len $\rightarrow$ 76 + 101 + 110 = 287 $\rightarrow$ 287 mod 11 = 1
- Moe $\rightarrow$ 77 + 111 + 101 = 289 $\rightarrow$ 289 mod 11 = 3
- Lou $\rightarrow$ 76 + 111 + 117 = 304 $\rightarrow$ 304 mod 11 = 7
- Rae $\rightarrow$ 82 + 97 + 101 = 280 $\rightarrow$ 280 mod 11 = 5
- Max $\rightarrow$ 77 + 97 + 120 = 294 $\rightarrow$ 394 mod 11 = 8

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	

Collisions

- Mia $\rightarrow$ 77 (M) + 105 (i) + 97 (a) = 279 $\rightarrow$ 279 mod 11 = 4
- Tim $\rightarrow$ 84 + 105 + 109 = 298 $\rightarrow$ 298 mod 11 = 1
- Bea $\rightarrow$ 66 + 101 + 97 = 264 $\rightarrow$ 264 mod 11 = 0
- Zoe $\rightarrow$ 90 + 11 + 101 = 302 $\rightarrow$ 302 mod 11 = 5
- Sue $\rightarrow$ 83 + 117 + 101 = 301 $\rightarrow$ 301 mod 11 = 4
- Len $\rightarrow$ 76 + 101 + 110 = 287 $\rightarrow$ 287 mod 11 = 1
- Moe $\rightarrow$ 77 + 111 + 101 = 289 $\rightarrow$ 289 mod 11 = 3
- Lou $\rightarrow$ 76 + 111 + 117 = 304 $\rightarrow$ 304 mod 11 = 7
- Rae $\rightarrow$ 82 + 97 + 101 = 280 $\rightarrow$ 280 mod 11 = 5
- Max $\rightarrow$ 77 + 97 + 120 = 294 $\rightarrow$ 394 mod 11 = 8
- Tod $\rightarrow$ 84 + 111 + 100 = 295 $\rightarrow$ 295 mod 11 = 9

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod

Collision resolutions

Open addressing

- In open addressing all elements are directly placed in the table
- When inserting new element, if the index i is already taken, we need to probe the array for a free slot
- There are multiple probing options:
  - Linear probing - we perform linear search for the free slot
  - Quadratic probing - we look at positions $1^2, 2^2, 3^2, \ldots$ away from the original
  - Double hashing - we apply hashing function to the computed hash and repeat the process until we find free slot
- Open addressing - every address (index) is open to every element

Chaining

- In this technique every cell of the array is linked list
- If there are two elements with the same index, they are added to linked list
- While searching for element we fetch corresponding linked list and perform linear search on this list

Chaining example

	Adjacency list
0	→ Bea
1	→ Tim → Len
2	
3	→ Moe
4	→ Mia → Sue
5	→ Zoe → Rae
6	
7	→ Lou
8	→ Max
9	→ Tod
10	

- Mia $\rightarrow 77 + 105 + 97 = 279 \rightarrow 279 \bmod 11 = 4$
- Tim $\rightarrow 84 + 105 + 109 = 298 \rightarrow 298 \bmod 11 = 1$
- Bea $\rightarrow 66 + 101 + 97 = 264 \rightarrow 264 \bmod 11 = 0$
- Zoe $\rightarrow 90 + 11 + 101 = 302 \rightarrow 302 \bmod 11 = 5$
- Sue $\rightarrow 83 + 117 + 101 = 301 \rightarrow 301 \bmod 11 = 4$
- Len $\rightarrow 76 + 101 + 110 = 287 \rightarrow 287 \bmod 11 = 1$
- Moe $\rightarrow 77 + 111 + 101 = 289 \rightarrow 289 \bmod 11 = 3$
- Lou $\rightarrow 76 + 111 + 117 = 304 \rightarrow 304 \bmod 11 = 7$
- Rae $\rightarrow 82 + 97 + 101 = 280 \rightarrow 280 \bmod 11 = 5$
- Max $\rightarrow 77 + 97 + 120 = 294 \rightarrow 394 \bmod 11 = 8$
- Tod $\rightarrow 84 + 111 + 100 = 295 \rightarrow 295 \bmod 11 = 9$

Hash tables analysis

- Performance of the hash tables strongly depends on the hash function - if the collisions are not minimized, then a lot of additional calculations would be required to produce result
- There is no single best hash function, performance depends on the specific data we will be storing in hash table
- If we know the data in advance however, we can devise a perfect hash function for this set of data
- If there are n entries in hash table, and s is the size of it, then on average there will be n/s entries in each index
- n/s is called **load factor** and ideally it should be kept low, so that complexity of all operations will be almost constant
- If load factor increases above some predefined threshold (usually somewhere above 0.75), a **rehashing** is done - size of the array is doubled and all hashes are calculated once again, to keep load factor small

Course plan

1. Hash tables

2. Text processing
   - Introduction
   - Boyer-Moore algorithm
   - Rabin-Karp algorithm
   - Regular expressions

3. Data compression
   - Introduction
   - Huffman coding
   - LZW compression

Course plan

Text processing

- Even though most of the data sent over the internet are multimedia files (video, audio), text processing remains the main function of computers
- Common application of computers is to store, edit, display and sent text files, but in recent years new branches appear with huge dependencies on fast text processing:
  - DNA analysis uses string matching algorithms to identify similarities of DNA strands.
  - Web crawlers process text in web pages to select ones suitable for a given query
  - Plagiarism detection algorithm compare large amounts of text files to detect stolen intellectual property
  - Speech-to-text algorithms and products like Siri/Alexa/OK Google
  - Detecting similarities of audio tracks (Shazam)
- The data set analyzed for this problems are huge, so the algorithm must be fast
- Strings come with no explicit structure, but many algorithms discover implicit structures that they can utilize

Definitions

- For simplicity we will treat strings and patterns as arrays
- `S[]` is the searched string, it has length `n`
- `S[i]` is the `i`-th character of a string `S`
- `S[i..j]` is the substring of a string `S` from `i` to `j`
- `P[]` is the pattern we are looking for in a string `S`, it has length `m`
- We say that pattern `P` was found at index `i`, if `S[i..i+m]` $==$ `P[]`
- There can be multiple occurrences of pattern in string

Brute force solution

- Brute force solution checks every possible option for the pattern to appear
- If pattern is found index of its first occurrence is returned, otherwise -1
- Complexity is $O(mn)$ where n is length of string and m is length of pattern

Pseudocode

```
1 BruteForceSearch(string S, string P)
2   n = S.length
3   m = P.length
4   for i in (0..n-m)
5     for j in (0..m-1)
6       if S[i+j] != P[j]
7         break //exits inner for loop
8     return i
9   return -1
```

Course plan

Boyer-Moore algorithm

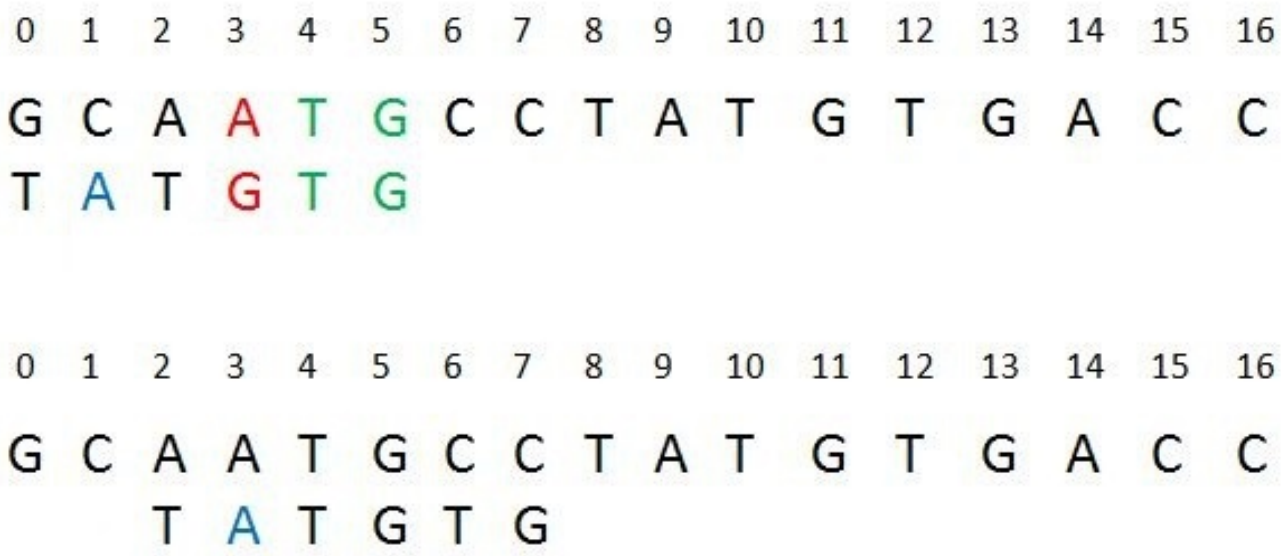
- In brute force solution iterative comparisons are performed - first we compare character in string with first character in pattern and if they match we perform subsequent matchings
- If at any point there is a mismatch, we advance string by one and start the whole procedure once again
- In the end every character in string is compared, which results in large complexity
- The main idea of Boyer-Moore is that when we compare end of pattern to text, we can advance search by more characters than one (as in brute force solution):
  - If the mismatched character is somewhere else in the pattern, we can advance text by that number of characters
  - If the mismatched character is not in the pattern, we can advance text by the whole length of pattern

Pattern preprocessing

- Boyer-Moore algorithm, to work efficiently, must perform pattern preprocessing and store the results in internal tables
- There are two heuristics for which preprocessing is done:
  - **Bad Character Heuristic**
  - **Good Suffix heuristic**
- The algorithm would work with only one of those heuristics implemented, but with two it can choose the maximum number of characters in string to skip

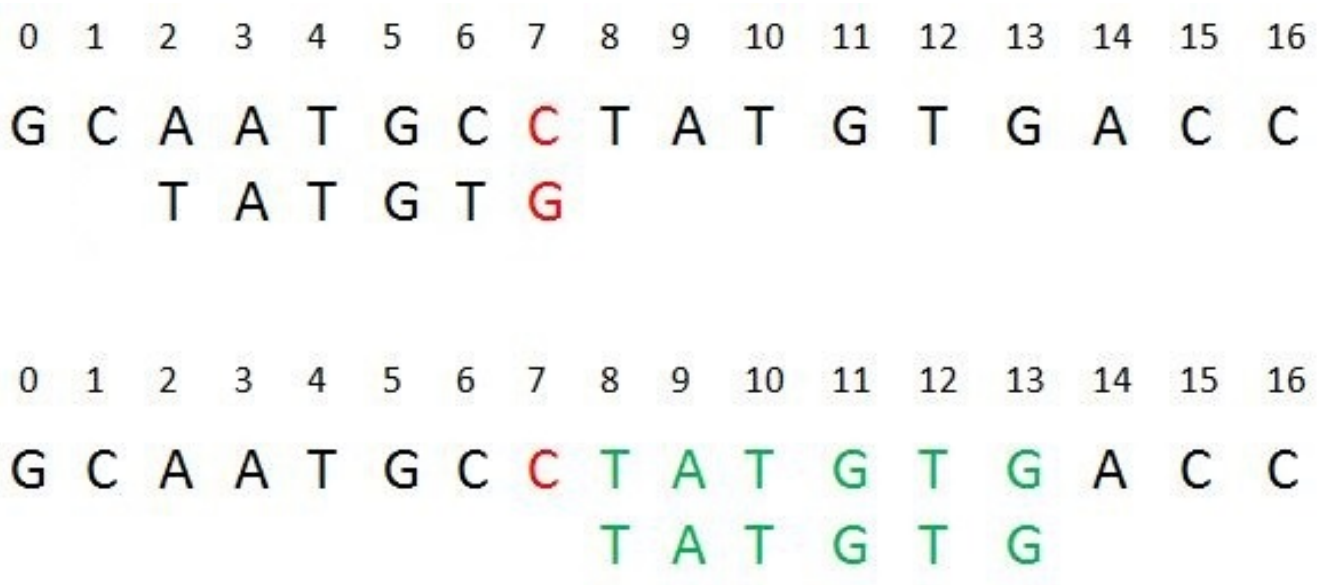
Bad character heuristics

- The character of the text which doesn't match with the current character of the pattern is called the **Bad Character**.
- Upon mismatch, we shift the pattern until:
  1. The mismatch becomes a match



Bad character heuristics

- The character of the text which doesn't match with the current character of the pattern is called the **Bad Character**.
- Upon mismatch, we shift the pattern until:
  1. The mismatch becomes a match
  2. Pattern P move past the mismatched character.



Good suffix heuristics

- Let s be substring of string S which is matched with substring of pattern P. Now we shift pattern until :
  1. Another occurrence of s in P matched with s in S.

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Good suffix heuristics

- Let s be substring of string S which is matched with substring of pattern P. Now we shift pattern until :
  1. Another occurrence of s in P matched with s in S.
  2. A prefix of P, which matches with suffix of s

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P				A	B	B	A	B			

Good suffix heuristics

- Let s be substring of string S which is matched with substring of pattern P. Now we shift pattern until :
  1. Another occurrence of s in P matched with s in S.
  2. A prefix of P, which matches with suffix of s
  3. P moves past s

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

Course plan

Rabin-Karp algorithm

- Brute force solution to string matching problem was to compare every character in string
- Boyer-Moore algorithm reduces the complexity by skipping parts of the string, thanks to preprocessing pattern
- Rabin-Karp algorithm tackles the problem from different angle - it tries to optimize comparison operations

- So far when we compared pattern and string we had to do it one character at a time. Is there a better way?
- Let's apply a hashing function to our pattern: $H(P) = h_P$. If we now apply the same hashing function to part of the string of length m (length of patter) we obtain $H(S[i..i+m]) = h_{Si}$.
- To see if pattern P matches part of the string $S[i..i+m]$ we just need to see if $h_P == h_{Si}$

Hash function

- Right now we don't see any improvement over the brute force solution, as we still are forced to calculate the hash for every substring in S
- That is, until we find a hash function with the following property: hash at the next shift must be efficiently computable from the current hash value and next character in text
- In mathematical terms, we would like to obtain something like this:

$$h(S[i+1..i+m+1]) = h(S[i..i+m]) - h(S[i]) + h(S[i+m+1])$$

- Hash function that has the above property is the rolling hash function developed by Rabin and Karp:
  - Assume that we can treat all characters like numbers (take ASCII code for example) and there are d possible characters in our alphabet
  - Then $S[i]$ will return plain number
  - Let q be some very big prime number and m the length of pattern
  - We obtain the following formula:

$$h(S[i+1..i+m+1]) = ((h(S[i..i+m]) - S[i] * d^{m-1}) * d + S[i+m+1]) \, mod \, q$$

Rabin-Karp example

- Alphabet consist of only four letters ($d = 4$): A, C, G, T with corresponding numbers 0, 1, 2, 3
- Prime $q = 11$
- Pattern $P = ACG$
- Pattern hash $h(P) = (0 * 4^2 + 1 * 4^1 + 2 * d^0) mod\ 11 = 6 mod\ 11 = 6$

String	A	G	G	A	C	G
$h(S[0..2]) = 10$	A	C	G			
$h(S[1..3]) = 7$		A	C	G		
$h(S[2..4]) = 0$			A	C	G	
$h(S[3..5]) = 6$				A	C	G

Algorithms analysis

- Both Boyer-Moore and Rabin-Karp algorithms can be considered when implementing string search algorithm
- Boyer-Moore optimizes pattern shifting, while Rabin-Karp string comparison
- Both algorithms have complexity of $O(m+n)$, while Boyer-Moore can sometimes be run in $O(n)$
- Boyer-Moore is considered the best string matching algorithm for in usual applications and is industry benchmark for comparing other solutions
- Rabin-Karp algorithm requires very good hashing function that will minimize collisions - every collision that is a false hit implies sequential comparison to be run
- String matching is quite common task while processing text, like web page or XML document, surely there must exist a quick and ready to use solution?

Course plan

1. Hash tables
2. Text processing
   - Introduction
   - Boyer-Moore algorithm
   - Rabin-Karp algorithm
   - Regular expressions
3. Data compression
   - Introduction
   - Huffman coding
   - LZW compression


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Regular expressions

- Regular expression (regex, regexp) are a powerful tool to search large amounts of text for specific pattern
- Apart from searching for a pattern they can do substitutions, given a search pattern and replacement value
- There are two standards: POSIX and Perl
- We will stick to Perl standard, as it is more widely used (Java, JavaScript, C#, Postgres all use Perl standard)
- Once you learn the regex syntax, you can use it in almost any language there is!
- At first sight regular expressions look like a mess of random characters, but there is a beautiful structure underneath!

Regular expression examples

- `/^[a-z0-9_-]{3,16}$/` - matches a string that is at least 3 and at most 16 characters long, composed of letters from a to z , numbers from 0 to 9, underscore and hyphen characters
- `/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/` - matches an email
- `/^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$/` - matches an HTML tag
- `/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$/` - matches an url

Regular expression operators

Anchors

- `^` - matches beginning of the string
- `$` - matches end of string
- `\b` - matches word boundary

Quantifiers

- `*` - match zero or more times
- `+` - match one or more times
- `?` - match zero or one time
- `{2}` - match exactly two times
- `{2,4}` - match at least two and at most four times

Regular expression operators

Character classes

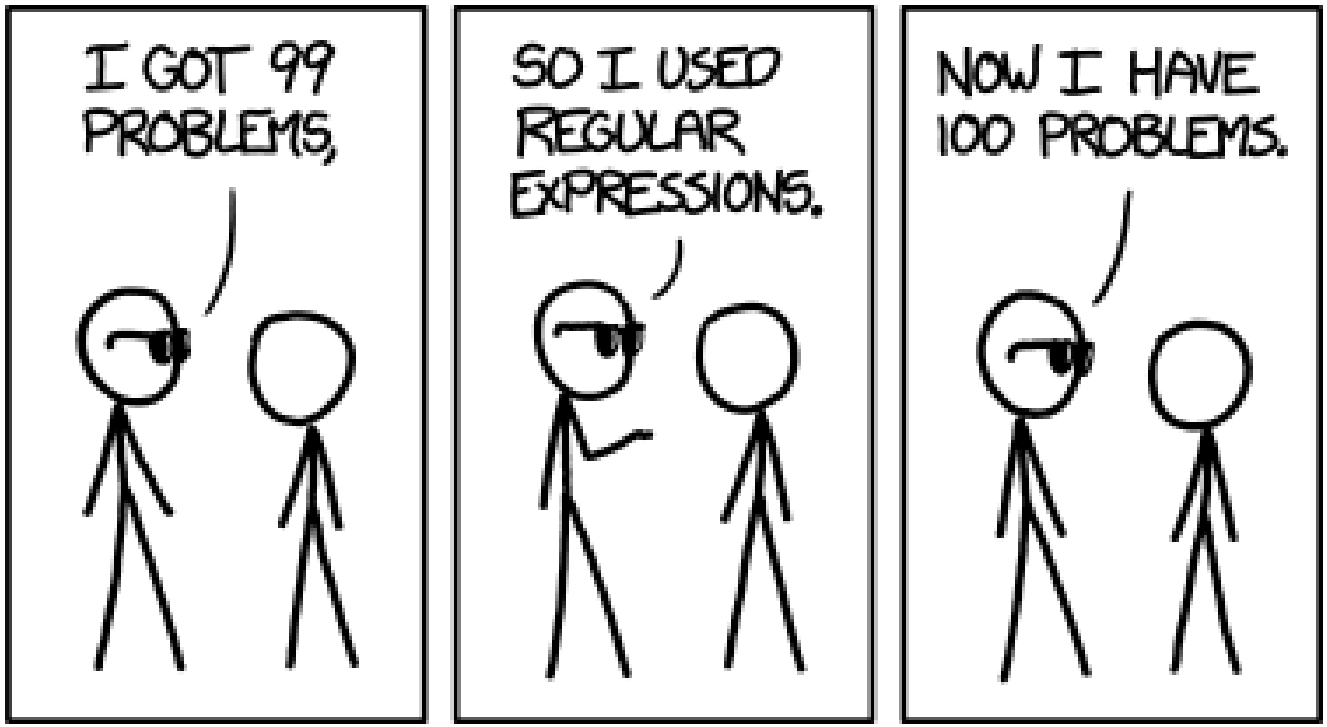
- `.` - match a single character
- `\w` - match a single word character (letter, digit, underscore)
- `\W` - match a single non-word character
- `\d` - match a single digit
- `\D` - match a single non-digit
- `\s` - match a single whitespace character (space, tab, newline)
- `\S` - match a single non-whitespace character

Bracket expressions

- `[abc]` - match a single character that is either a, b or c
- `[^abc]` - match a single character that is neither a, b nor c
- `[a-g]` - match a single character that is either a, b, c, ... g

Regular expression summary

Regular expressions are powerful, but very costly! They are difficult to write, even more difficult to read and not well-written expressions are very inefficient. If you can avoid regular expression do it!



Course plan

1. Hash tables
2. Text processing
   - Introduction
   - Boyer-Moore algorithm
   - Rabin-Karp algorithm
   - Regular expressions
3. Data compression
   - Introduction
   - Huffman coding
   - LZW compression

Course plan

1. Hash tables
2. Text processing
   - Introduction
   - Boyer-Moore algorithm
   - Rabin-Karp algorithm
   - Regular expressions
3. Data compression
   - Introduction
   - Huffman coding
   - LZW compression

Data compression

- More and more data of high quality is being processed and sent through the internet
- We can reduce the size of specific piece of data by compressing it
- Of course there are no free meals - we need to spend more time decoding that file
- DVDs would only hold seconds of video if compression methods were NOT used
- We distinguish between lossless compression, where compression-decompression cycle we get the exact same file, and lossy compression, where the resulting file might differ
- Lossy compression is the backbone for all image, audio and video files - after compression we loose some quality
- Lossless compression is mandatory for text files, binaries, etc.

Simple compression algorithm

- Consider a string of bits: 11000011111001110000
- Let's replace sequences of the same characters with a single digit
- Assume that all strings start with 0
- We get 245234 - much shorter!
- Or is it? In memory string of bits takes 20 bits of memory
- Our result string takes $6 * 8 = 48$ bits of memory!
- Compression ratio is below 1, so this is not a viable compression method

Course plan

1. Hash tables
2. Text processing
   - Introduction
   - Boyer-Moore algorithm
   - Rabin-Karp algorithm
   - Regular expressions
3. Data compression
   - Introduction
   - Huffman coding
   - LZW compression

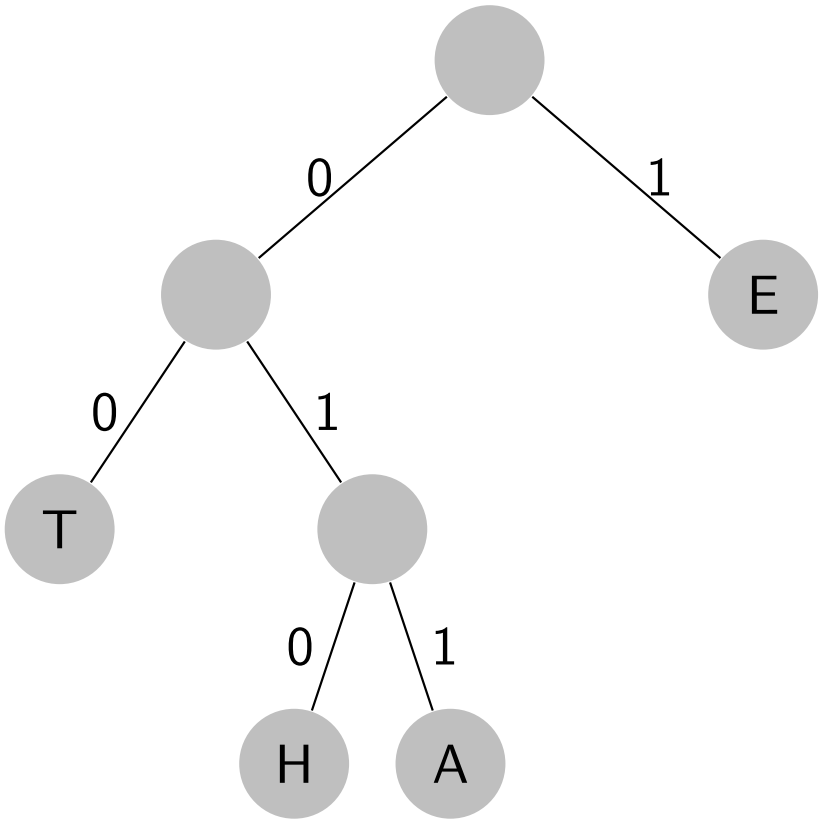
Huffman coding

- Huffman coding algorithm was developed in 1952 and is a lossless, not patented algorithm
- Basic principle - some characters in text are more common than others, so we should use shorter codes for them
- For example 'e' in binary is 01100101, but it's the most common letter in English alphabet, so why not just one or two bits, like 1 or 01?
- Problem now becomes to find an optimal set of codes for a given text
- We need to analyze how frequently different characters occur in text

Variable length encoding

- In Huffman coding characters will have variable length, for example code for 'e' might be only two bits long, but for 'x' it might be 10 bits
- How can we then tell where one character ends and another begins?
- Is 1011001000:
  - 10 followed by 11001000
  - 1011 followed by 001000
  - Something else?
- To avoid this problem we need to ensure, that we never have two codes such that one code starts with another code
- It seems like a hard task, is there a simple solution for this?

Binary tree



Codes

E - 1, T - 00, H - 010, A - 011

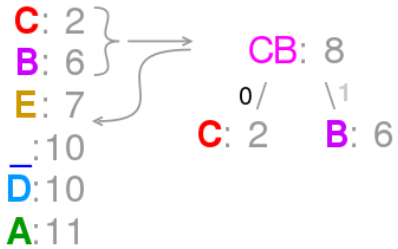
- Each leaf is code for specific letter
- Code is constructed by following paths
- Each left path is 0, right is 1
- Letters that occur frequently are closer to root

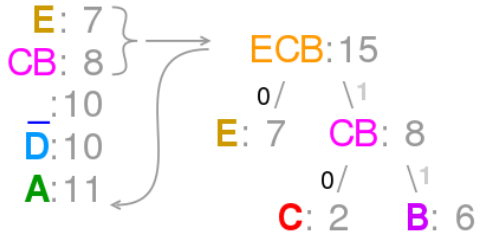
Constructing Huffman tree

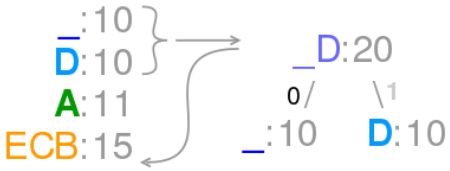
- Make a leaf node for every letter
- Construct a priority queue
- Calculate a probability of occurrence of every letter and add it to priority queue with priority equal to calculated probability
- While there are more than one letter in queue:
  - Remove two nodes from queue with lowest priority (lowest probabilty)
  - Construct an internal node with those nodes as children and priority equal to the sum of child nodes priority
  - Add this new node to priority queue
- The remaining node is the root of the tree

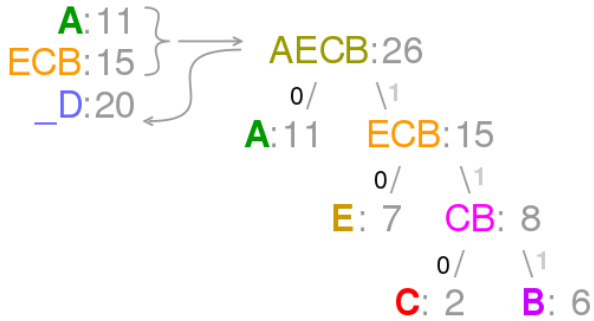
Huffman coding example

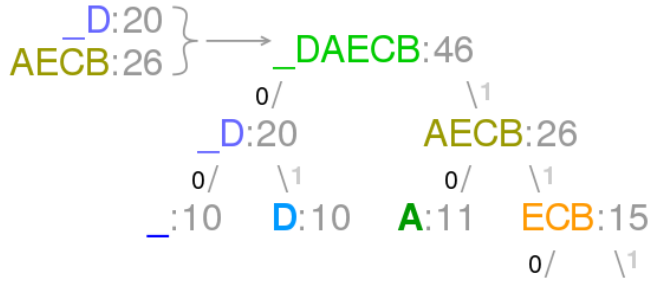
1. "A_DEAD_DAD_CEDED_A_BAD_BABE_A_BEADED_ABACA_BED"

2. 

3. 

4. 

5. 

6. 

7. _: 00
D: 01
A: 10
E: 110
C: 1110
B: 1111

8. "1000011101001000110010011101100111001001000111110010011111011111100010000111111010011100100101111101110100011111100 1"

Huffman coding analysis

- Compression algorithm complexity is $O(n \lg n)$
- We need to send the resulting tree (or ways to construct it) with the compressed message
- Compression ratio is about 2
- Compression is lossless
- We refer to this type algorithms as Run Length Encoding (RLE)
- This is a greedy algorithm
- Coding tree can be constructed in many different ways, so both cipher and decipher must use the same one
- To increase compression ratio we can construct codes for pairs of letters
- This algorithm will not work well if all or most of the characters in alphabet have the same probability

Course plan

LZW compression algorithm

- LZW algorithm, named after it's creators Abraham Lempel, Jacob Ziv, and Terry Welch, was developed in 1984
- It's all purpose compression algorithm, able to compress all kind of data
- GIF format uses this algorithm to compress images, as well as UNIX *compress* utility
- Principle of this algorithm is to construct coding dictionary with sequences of characters that are present in the text, avoiding storing those that aren't

- Initialize a dictionary with codes for all single characters
- Start scanning text character by character
  - At each character find the longest prefix of string, that is already in the dictionary
  - Output the code for that prefix
  - Add the prefix plus the next character to the dictionary as new code
- To decode message we only need to send the initial dictionary of single characters, the rest will be filled during decompression

LZW example

String: TOBEORNOTTOBEORTOBEORNOT#

Symbol	Binary	Decimal
#	00000	0
A	00001	1
B	00010	2
C	00011	3
D	00100	4
E	00101	5
F	00110	6
G	00111	7
H	01000	8
I	01001	9
J	01010	10
K	01011	11
L	01100	12
M	01101	13

Symbol	Binary	Decimal
N	01110	14
O	01111	15
P	10000	16
Q	10001	17
R	10010	18
S	10011	19
T	10100	20
U	10101	21
V	10110	22
W	10111	23
X	11000	24
Y	11001	25
Z	11010	26

Current Sequence	Next Char	Output		Extended Dictionary		Comments
		Code	Bits			
NULL	T					
T	O	20	10100	27:	TO	27 = first available code after 0 through 26
O	B	15	01111	28:	OB	
B	E	2	00010	29:	BE	
E	O	5	00101	30:	EO	
O	R	15	01111	31:	OR	
R	N	18	10010	32:	RN	32 requires 6 bits, so for next output use 6 bits
N	O	14	001110	33:	NO	
O	T	15	001111	34:	OT	
T	T	20	010100	35:	TT	
TO	B	27	011011	36:	TOB	
BE	O	29	011101	37:	BEO	
OR	T	31	011111	38:	ORT	
TOB	E	36	100100	39:	TOBE	
EO	R	30	011110	40:	EOR	
RN	O	32	100000	41:	RNO	
OT	#	34	100010			# stops the algorithm; send the cur seq
		0	000000			and the stop code

LZW analysis

- Coding dictionary usually has 4096 entries
- If dictionary becomes full numerous techniques could be used to increase compression ratio
- A book compressed by this algorithm is roughly half of the original size
- Variations of this algorithm are patented
- It is a lossless compression algorithm
- To decode message we only need to send the initial dictionary of single characters, the rest will be filled during decompression