Strings and Pattern Matching

- Brute Force, Rabin-Karp, Knuth-Morris-Pratt

What’s up?

I’m looking for some string.

That’s quite a trick considering that you have no eyes.

Oh yeah? Have you seen your writing? It looks like an EKG!
String Searching

• The previous slide is not a great example of what is meant by “String Searching.” Nor is it meant to ridicule people without eyes....

• The object of **string searching** is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).

• As with most algorithms, the main considerations for string searching are speed and efficiency.

• There are a number of string searching algorithms in existence today, but the two we shall review are **Brute Force** and **Rabin-Karp**.
Brute Force

- The Brute Force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found:

<table>
<thead>
<tr>
<th>TWO ROADS DIVERGED IN A YELLOW WOOD ROADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD ROADS</td>
</tr>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD ROADS</td>
</tr>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD ROADS</td>
</tr>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD ROADS</td>
</tr>
</tbody>
</table>

- Compared characters are italicized.
- Correct matches are in boldface type.

- The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.
Brute Force Pseudo-Code

• Here’s the pseudo-code
  
  do
      if (text letter == pattern letter)
          compare next letter of pattern to next letter of text
      else
          move pattern down text by one letter
  while (entire pattern found or end of text)
**Brute Force-Complexity**

- Given a pattern M characters in length, and a text N characters in length...

- **Worst case:** compares pattern to each substring of text of length M. For example, M=5.

1) `AAAAA`AAAAAAAAAAAAAAAAAAAAAAAAN
   `AAAAH` 5 comparisons made

2) `AAAAA`AAAAAAAAAAAAAAAAAAAAAAAAN
   `AAAAH` 5 comparisons made

3) `AAAAA`AAAAAAAAAAAAAAAAAAAAAAAAN
   `AAAAH` 5 comparisons made

4) `AAAAA`AAAAAAAAAAAAAAAAAAAAAAAAN
   `AAAAH` 5 comparisons made

5) `AAAAA`AAAAAAAAAAAAAAAAAAAAAAAAN
   `AAAAH` 5 comparisons made

....

N) `AAAAAAAAAAAAAAAAAAAAAAAAAAAAHHH`
   5 comparisons made  `AAAAAH`

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: $O(MN)$
Brute Force-Complexity (cont.)

• Given a pattern M characters in length, and a text N characters in length...

• **Best case if pattern found**: Finds pattern in first M positions of text. For example, M=5.

1) AAAAAAAAAAAAAAABCD
AAAAA 5 comparisons made

• Total number of comparisons: M
• Best case time complexity: O(M)
Brute Force-Complexity (cont.)

- Given a pattern M characters in length, and a text N characters in length...

- **Best case if pattern not found**: Always mismatch on first character. For example, M=5.

1) \[ AAAAAAAAAAAAAAAAAAAAAAAAAAH \]
   \[ OOOOH \quad 1 \text{ comparison made} \]

2) \[ AAAAAAAAAAAAAAAAAAAAAAAAAAAAH \]
   \[ OOOOH \quad 1 \text{ comparison made} \]

3) \[ AAABBBBBBBBBBBBBBBBBBBBBBBBAAAAAH \]
   \[ OOOOH \quad 1 \text{ comparison made} \]

4) \[ AAAAABBBBBBBBBBBBBBBBBBBBBBBBBAAAAAH \]
   \[ OOOOH \quad 1 \text{ comparison made} \]

5) \[ AAAAAABBBBBBBBBBBBBBBBBBBBBBBBBAAAAAH \]
   \[ OOOOH \quad 1 \text{ comparison made} \]

...  

N) \[ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHH \]
   \[ 1 \text{ comparison made} \quad OOOOH \]

- Total number of comparisons: N

- Best case time complexity: \( O(N) \)
Rabin-Karp

- The Rabin-Karp string searching algorithm uses a hash function to speed up the search.

Rabin & Karp’s

Heavenly Homemade Hashish

Fresh from Syria
Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence of text to be compared.

- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.

- If the hash values are equal, the algorithm will do a **Brute Force comparison** between the pattern and the M-character sequence.

- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

- Perhaps a figure will clarify some things...
Rabin-Karp Example

Hash value of “AAAAA” is 37
Hash value of “AAAAAH” is 100

1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100   1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100   1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100   1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100   1 comparison made
   ...
N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHH
   AAAAH
   6 comparisons made
   100=100
Rabin-Karp Pseudo-Code

A pattern is $M$ characters long.

- $\text{hash}_p =$ hash value of pattern
- $\text{hash}_t =$ hash value of first $M$ letters in body of text

```
do
    if (hash_p == hash_t)
        brute force comparison of pattern and selected section of text
        hash_t = hash value of next section of text, one character over
    while (end of text or brute force comparison == true)
```
Rabin-Karp

• Common Rabin-Karp questions:

  “What is the hash function used to calculate values for character sequences?”

  “Isn’t it time consuming to hash every one of the M-character sequences in the text body?”

  “Is this going to be on the final?”

• To answer some of these questions, we’ll have to get mathematical.
Rabin-Karp Math

• Consider an M-character sequence as an M-digit number in base $b$, where $b$ is the number of letters in the alphabet. The text subsequence $t[i .. i+M-1]$ is mapped to the number

$$x(i) = t[i] \cdot b^{M-1} + t[i+1] \cdot b^{M-2} + ... + t[i+M-1]$$

• Furthermore, given $x(i)$ we can compute $x(i+1)$ for the next subsequence $t[i+1 .. i+M]$ in constant time, as follows:

$$x(i+1) = t[i+1] \cdot b^{M-1} + t[i+2] \cdot b^{M-2} + ... + t[i+M]$$

$$x(i+1) = x(i) \cdot b$$  \hspace{1cm} \text{Shift left one digit}

$$- t[i] \cdot b^M$$  \hspace{1cm} \text{Subtract leftmost digit}

$$+ t[i+M]$$  \hspace{1cm} \text{Add new rightmost digit}

• In this way, we never explicitly compute a new value. We simply adjust the existing value as we move over one character.
Rabin-Karp Mods

- If M is large, then the resulting value (~bM) will be enormous. For this reason, we hash the value by taking it \text{mod} a prime number q.

- The \text{mod} function (% in Java) is particularly useful in this case due to several of its inherent properties:
  - \((x \mod q) + (y \mod q)) \mod q = (x+y) \mod q\)
  - \((x \mod q) \mod q = x \mod q\)

- For these reasons:

  \[
  h(i) = ((t[i] \cdot b^{M-1} \mod q) + (t[i+1] \cdot b^{M-2} \mod q) + \ldots + (t[i+M-1] \mod q)) \mod q
  \]

  \[
  h(i+1) = (h(i) \cdot b \mod q)
  \]
  Shift left one digit
  \[
  -t[i] \cdot b^{M} \mod q
  \]
  Subtract leftmost digit
  \[
  +t[i+M] \mod q \)
  Add new rightmost digit
  \[
  \mod q
  \]
Strings and Pattern Matching

Rabin-Karp Pseudo-Code

*pattern is M characters long*

hash\_p = hash value of pattern

hash\_t = hash value of first M letters in body of text

**do**

**if** (hash\_p == hash\_t)
    brute force comparison of pattern and selected section of text
    hash\_t = hash value of next section of text, one character over

**while** (end of text or brute force comparison == true)
Rabin-Karp Complexity

• If a sufficiently large prime number is used for the hash function, the hashed values of two different patterns will usually be distinct.

• If this is the case, searching takes $O(N)$ time, where $N$ is the number of characters in the larger body of text.

• It is always possible to construct a scenario with a worst case complexity of $O(MN)$. This, however, is likely to happen only if the prime number used for hashing is small.
The Knuth-Morris-Pratt Algorithm

• The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.

• A failure function \( f \) is computed that indicates how much of the last comparison can be reused if it fails.

• Specifically, \( f \) is defined to be the longest prefix of the pattern \( P[0,\ldots,j] \) that is also a suffix of \( P[1,\ldots,j] \)
  - Note: not a suffix of \( P[0,\ldots,j] \)

• Example:
  - value of the KMP failure function:

    | j | 0 | 1 | 2 | 3 | 4 | 5 |
    |---|---|---|---|---|---|---|
    | \( P[j] \) | a | b | a | b | a | c |
    | \( f(j) \) | 0 | 0 | 1 | 2 | 3 | 0 |

• This shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison.
  - if the comparison fails at (4), we know the a,b in positions 2,3 is identical to positions 0,1
The KMP Algorithm (contd.)

• Time Complexity Analysis

• define \( k = i - j \)

• In every iteration through the while loop, one of three things happens.
  - 1) if \( T[i] = P[j] \), then \( i \) increases by 1, as does \( j \) \( k \) remains the same.
  - 2) if \( T[i] \neq P[j] \) and \( j > 0 \), then \( i \) does not change and \( k \) increases by at least 1, since \( k \) changes from \( i - j \) to \( i - f(j-1) \)
  - 3) if \( T[i] \neq P[j] \) and \( j = 0 \), then \( i \) increases by 1 and \( k \) increases by 1 since \( j \) remains the same.

• Thus, each time through the loop, either \( i \) or \( k \) increases by at least 1, so the greatest possible number of loops is \( 2n \)

• This of course assumes that \( f \) has already been computed.

• However, \( f \) is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is \( O(m) \)

• Total Time Complexity: \( O(n + m) \)
The KMP Algorithm (contd.)

- the KMP string matching algorithm: Pseudo-Code

Algorithm KMPMatch\((T,P)\)

Input: Strings \(T\) (text) with \(n\) characters and \(P\) (pattern) with \(m\) characters.
Output: Starting index of the first substring of \(T\) matching \(P\), or an indication that \(P\) is not a substring of \(T\).

\[
f \leftarrow \text{KMPFailureFunction}(P) \text{ \{build failure function\}} \\
i \leftarrow 0 \\
j \leftarrow 0 \\
\text{while } i < n \text{ do} \\
\hspace{1em} \text{if } P[j] = T[i] \text{ then} \\
\hspace{2em} \text{if } j = m - 1 \text{ then} \\
\hspace{3em} \text{return } i - m - 1 \text{ \{a match\}} \\
\hspace{2em} i \leftarrow i + 1 \\
\hspace{2em} j \leftarrow j + 1 \\
\hspace{1em} \text{else if } j > 0 \text{ then} \text{ \{no match, but we have advanced\}} \\
\hspace{2em} j \leftarrow f(j-1) \text{ \{j indexes just after matching prefix in P\}} \\
\hspace{1em} \text{else} \\
\hspace{2em} i \leftarrow i + 1 \\
\text{return "There is no substring of } T \text{ matching } P" \]
The KMP Algorithm (contd.)

- The KMP failure function: Pseudo-Code

Algorithm KMPFailureFunction($P$);

Input: String $P$ (pattern) with $m$ characters

Output: The failure function $f$ for $P$, which maps $j$ to the length of the longest prefix of $P$ that is a suffix of $P[1,..,j]$


t ← 1

j ← 0

while $i \leq m-1$ do

    if $P[j] = T[j]$ then
        \{we have matched $j + 1$ characters\}
        \[f(i) \leftarrow j + 1\]
        i ← i + 1
        j ← j + 1
    
    else if $j > 0$ then
        \{j indexes just after a prefix of $P$ that matches\}
        \[j \leftarrow f(j-1)\]
    
    else
        \{there is no match\}
        \[f(i) \leftarrow 0\]
        i ← i + 1


The KMP Algorithm (contd.)

- A graphical representation of the KMP string searching algorithm

```
| a | b | a | c | a | a | b | a | c | c | a | b | a | c | a | b | a | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
| a | b | a | c | a | b |
| 7 |
| a | b | a | c | a | b |
| 8 | 9 | 10 | 11 | 12 |
| a | b | a | c | a | b |
| 13 |
| a | b | a | c | a | b |
| 14 | 15 | 16 | 17 | 18 | 19 |
| a | b | a | c | a | b |
```

no comparison needed here
Regular Expressions

• notation for describing a set of strings, possibly of infinite size

• \( \varepsilon \) denotes the empty string

• \( ab + c \) denotes the set \{ab, c\}

• \( a^* \) denotes the set \{\( \varepsilon \), a, aa, aaa, ...\}

• Examples
  - \( (a+b)^* \) all the strings from the alphabet \{a,b\}
  - \( b^*(ab^a)^*b^* \) strings with an even number of a’s
  - \( (a+b)^*\text{sun}(a+b)^* \) strings containing the pattern “sun”
  - \( (a+b)(a+b)(a+b)a \) 4-letter strings ending in a
Finite State Automaton

• “machine” for processing strings
Composition of FSA’s

Strings and Pattern Matching
Tries

• A trie is a tree-based data structure for storing strings in order to make pattern matching faster.

• Tries can be used to perform **prefix queries** for information retrieval. Prefix queries search for the longest prefix of a given string X that matches a prefix of some string in the trie.

• A trie supports the following operations on a set S of strings:

  - **insert(X)**: Insert the string X into S
    
    **Input**: String  
    **Output**: None

  - **remove(X)**: Remove string X from S
    
    **Input**: String  
    **Output**: None

  - **prefixes(X)**: Return all the strings in S that have a longest prefix of X
    
    **Input**: String  
    **Output**: Enumeration of strings
Tries (cont.)

- Let $S$ be a set of strings from the alphabet $\Sigma$ such that no string in $S$ is a prefix to another string. A **standard trie** for $S$ is an ordered tree $T$ that:
  - Each edge of $T$ is labeled with a character from $\Sigma$
  - The ordering of edges out of an internal node is determined by the alphabet $\Sigma$
  - The path from the root of $T$ to any node represents a prefix in $\Sigma$ that is equal to the concatenation of the characters encountered while traversing the path.

- For example, the standard trie over the alphabet $\Sigma = \{a, b\}$ for the set $\{aabab, abaab, babbb, bbbaa, bbab\}$

![Trie Diagram](attachment:image.png)
Tries (cont.)

• An internal node can have 1 to $d$ children when $d$ is the size of the alphabet. Our example is essentially a binary tree.

• A path from the root of $T$ to an internal node $v$ at depth $i$ corresponds to an $i$-character prefix of a string of $S$.

• We can implement a trie with an ordered tree by storing the character associated with an edge at the child node below it.
Compressed Tries

- A **compressed trie** is like a standard trie but makes sure that each trie had a degree of at least 2. Single child nodes are compressed into an single edge.

- A **critical node** is a node \( v \) such that \( v \) is labeled with a string from \( S \), \( v \) has at least 2 children, or \( v \) is the root.

- To convert a standard trie to a compressed trie we replace an edge \((v_0, v_1)\) each chain on nodes \((v_0, v_1 \ldots v_k)\) for \( k \geq 2 \) such that
  - \( v_0 \) and \( v_1 \) are critical but \( v_1 \) is critical for \( 0 < i < k \)
  - each \( v_1 \) has only one child

- Each internal node in a compressed tire has at least two children and each external is associated with a string. The compression reduces the total space for the trie from \( O(m) \) where \( m \) is the sum of the the lengths of strings in \( S \) to \( O(n) \) where \( n \) is the number of strings in \( S \).
Compressed Tries (cont.)

• An example:

```
ab
ab
a
b
bb
bb
b
bb
a
aa
aa
a
b
12 3 45
abab baab babbb
aaa bab12 3 45
```
Prefix Queries on a Trie

**Algorithm** prefixQuery\((T, X)\):

**Input:** Trie \(T\) for a set \(S\) of strings and a query string \(X\)

**Output:** The node \(v\) of \(T\) such that the labeled nodes of
the subtree of \(T\) rooted at \(v\) store the strings
of \(S\) with a longest prefix in common with \(X\)

\(v \leftarrow T\).root()
\(i \leftarrow 0\) \{ \(i\) is an index into the string \(X\) \}

**repeat**

for each child \(w\) of \(v\) do

let \(e\) be the edge \((v, w)\)

\(Y \leftarrow \text{string}(e)\) \{ \(Y\) is the substring associated with \(e\) \}

\(l \leftarrow Y\).length() \{ \(l=1\) if \(T\) is a standard trie \}

\(Z \leftarrow X\).substring\((i, i+l-1)\) \{ \(Z\) holds the next \(l\) characters of \(X\) \}

if \(Z = Y\) then

\(v \leftarrow w\)

\(i \leftarrow i+1\) \{ move to \(W\), incrementing \(i\) past \(Z\) \}

break out of the **for** loop

else if a proper prefix of \(Z\) matched a proper prefix
of \(Y\) then

\(v \leftarrow w\)

break out of the **repeat** loop

until \(v\) is external or \(v \neq w\)

return \(v\)
Insertion and Deletion

- Insertion: We first perform a prefix query for string X. Let us examine the ways a prefix query may end in terms of insertion.
  - The query terminates at node v. Let X₁ be the prefix of X that matched in the trie up to node v and X₂ be the rest of X. If X₂ is an empty string we label v with X and the end. Otherwise we create a new external node w and label it with X.
  - The query terminates at an edge e = (v, w) because a prefix of X match prefix(v) and a proper prefix of string Y associated with e. Let Y₁ be the part of Y that X matched to and Y₂ the rest of Y. Likewise for X₁ and X₂. Then X = X₁ + X₂ = prefix(v) + Y₁ + X₂. We create a new node u and split the edges (v, u) and (u, w). If X₂ is empty then w label u with X. Otherwise we create a node z which is external and label it X.

- Insertion is O(dn) when d is the size of the alphabet and n is the length of the string t insert.
Strings and Pattern Matching

Insertion and Deletion (cont.)

```
abab  baab  abbb

search stops here

insert(bbaaabb)
```

```
abab  baab  abbb

abbb

bab

aaa

search stops here

insert(bbaaabb)
```
Lempel Ziv Encoding

• Constructing the trie:
  - Let phrase 0 be the null string.
  - Scan through the text
  - If you come across a letter you haven’t seen before, add it to the top level of the trie.
  - If you come across a letter you’ve already seen, scan down the trie until you can’t match any more characters, add a node to the trie representing the new string.
  - Insert the pair (nodeIndex, lastChar) into the compressed string.

• Reconstructing the string:
  - Every time you see a ‘0’ in the compressed string add the next character in the compressed string directly to the new string.
  - For each non-zero nodeIndex, put the substring corresponding to that node into the new string, followed by the next character in the compressed string.
Lempel Ziv Encoding (contd.)

- A graphical example:

Uncompressed text: \textbf{how now brown cow in town.}

phrases: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Compressed text: 0h0o0w0_0n2w4b0r6n4c6_0i5_0t9.

Trie:
File Compression

- text files are usually stored by representing each character with an 8-bit ASCII code (type `man ascii` in a Unix shell to see the ASCII encoding)

- the ASCII encoding is an example of **fixed-length encoding**, where each character is represented with the same number of bits

- in order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others

- **variable-length encoding** uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters.

- Example:
  - text: `java`
  - encoding: `a = “0”, j = “11”, v = “10”`
  - encoded text: `110100` (6 bits)

- How to decode?
  - `a = “0”, j = “01”, v = “00”`
  - encoded text: `010000` (6 bits)
  - is this `java, jvv, jaaaa ...`
Encoding Trie

• to prevent ambiguities in decoding, we require that the encoding satisfies the **prefix rule**, that is, no code is a prefix of another code
  - \( a = \text{“0”}, j = \text{“11”}, v = \text{“10”} \) satisfies the prefix rule
  - \( a = \text{“0”}, j = \text{“01”}, v = \text{“00”} \) does **not** satisfy the prefix rule (the code of \( a \) is a prefix of the codes of \( j \) and \( v \))

• we use an **encoding trie** to define an encoding that satisfies the prefix rule
  - the characters stored at the external nodes
  - a left edge means 0
  - a right edge means 1

\[
\begin{align*}
A &= 010 \\
B &= 11 \\
C &= 00 \\
D &= 10 \\
R &= 011
\end{align*}
\]
Example of Decoding

- trie:

- encoded text:
  01011011010000101001011011010

- text:
  A = 010
  B = 11
  C = 00
  D = 10
  R = 011

- text:

\[
\text{encoded text:} \\
01011011010000101001011011010 \\
\text{text:}
\]
Trie this!

Strings and Pattern Matching
**Optimal Compression**

- An issue with encoding tries is to insure that the encoded text is as short as possible:

  ABRACADABRA
  010110110100001010010110110101
  **29 bits**

  ABRACADABRA
  0010110001000011001100
  **24 bits**
Huffman Encoding Trie

ABRACADABRA

<table>
<thead>
<tr>
<th>character</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A

B

R

C

D

5

2

2

1

1

6

4

2

2

1

1

B

R

C

D
Huffman Encoding Trie (contd.)

```plaintext
5
A

5
A

4
2
B

2
2
R

6
1
C

2
1
D

11
0
B

0
R

1
C

1
D

0
1
0
1

0
1
0
1
```
Final Huffman Encoding Trie

A  B  R  A  C  A  D  A  B  R  A
0  100 101 0  110 0  111 0  100 101 0

23 bits
Another Huffman Encoding Trie

ABRACADABRA

<table>
<thead>
<tr>
<th>character</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A  
5  
B  
2  
R  
2  
D  
1  
C  
1  

Strings and Pattern Matching
Another Huffman Encoding Trie

Strings and Pattern Matching
Another Huffman Encoding Trie

Strings and Pattern Matching
Another Huffman Encoding Trie

A B R A C A D A B R A
0 10 110 0 1100 0 1111 0 10 110 0
23 bits
Construction Algorithm

- with a Huffman encoding trie, the encoded text has minimal length

**Algorithm** Huffman($X$):

**Input**: String $X$ of length $n$

**Output**: Encoding trie for $X$

Compute the frequency $f(c)$ of each character $c$ of $X$. Initialize a priority queue $Q$.

**for** each character $c$ in $X$ **do**

Create a single-node tree $T$ storing $c$

$Q$.insertItem($f(c)$, $T$)

**while** $Q$.size() > 1 **do**

$f_1$ $\leftarrow$ $Q$.minKey()

$T_1$ $\leftarrow$ $Q$.removeMinElement()

$f_2$ $\leftarrow$ $Q$.minKey()

$T_2$ $\leftarrow$ $Q$.removeMinElement()

Create a new tree $T$ with left subtree $T_1$ and right subtree $T_2$.

$Q$.insertItem($f_1 + f_2$)

**return** tree $Q$.removeMinElement()

- running time for a text of length $n$ with $k$ distinct characters: $O(n + k \log k)$
Image Compression

- we can use Huffman encoding also for binary files (bitmaps, executables, etc.)
- common groups of bits are stored at the leaves
- Example of an encoding suitable for b/w bitmaps