Algorithms and Data Structures Review

Angelos-Christos Anadiotis
Data structures

• Give structure to our data
• Separate between unstructured and structured data
• Data can be structured in several different ways
• Decision of the optimal data structure depends on the use case

How do we measure the data access cost in order to decide which is the optimal data structure for our data?
Example: Binary search tree
Example: Binary search tree

Query: Find 5
Example: Binary search tree

Query: Find 5
Example: Binary search tree

Query: Find 5
Example: Binary search tree

Query: Find 5
Example: Binary search tree

Query: Find 5

4 steps
Example: Hash table

Bucket 0 → 8
Bucket 1 → 1, 9
Bucket 2 → 2, 10
Bucket 3 → 3, 11
Bucket 4 → 4, 12
Bucket 5 → 5, 13
Bucket 6 → 6, 14
Bucket 7 → 7, 15

\( h(x) = x \mod 8 \)
Example: Hash table

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

Query: Find 5

h(x) = x mod 8
Example: Hash table

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

Query: Find 5

$h(x) = x \mod 8$
Example: Hash table

Query: Find 5

1 step

h(x) = x mod 8
Computational complexity

• Operations on data structures (as well as every algorithm) require resources in terms of memory and time
  • The amount of resources required corresponds to the complexity of each operation

• Time complexity is the number of elementary operations executed on the input data, where each elementary operation is expected to require constant time

• Space complexity is the amount of memory required to execute an algorithm on the input data
Big O, Omega, Theta

• Big O (O) describes an upper bound on the algorithm run time
• Big Omega (Ω) describes a lower bound on the algorithm run time
• Big Theta (Θ) describes a tight bound on the algorithm run time

• Big O is typically used for giving the time complexity of an algorithm
• The upper bound that we give is expected to be the tightest one (even though this is not required by theory)
Compare complexity

\begin{align*}
O(1) & \quad \text{constant time} \\
O(\log N) & \quad \text{logarithmic time} \\
O(N) & \quad \text{linear time}
\end{align*}

\[ \text{Input data size} \quad \text{Time} \]
Compare complexity

- **O(1)**: Constant time
- **O(N)**: Linear time
- **O(logN)**: Logarithmic time

- **Linear search**
- **Binary search**
- **Hash table search**

Time vs. Input data size graph
Array

• Collection of elements, each identified by a position in the data structure

• An array may have multiple dimensions, however it is always stored as single-dimensional in the main memory

• Typically, an array has pre-defined size, and when we need to exceed it, we allocate a new, bigger, array where we copy the smaller one

• Insertion and data access cost is constant: O(1)
Stack

• Collection of elements that follow a Last-In-First-Out (LIFO) order
• Supports two fundamental operations: push and pop
• Insertion and removal of elements takes place from the same position of the data structure (front or rear)
• No assumptions on the data structure used for storage – stack is about the order of elements
Queue

- Collection of elements that follow a First-In-First-Out (FIFO) order
- Supports two fundamental operations: enqueue and dequeue
- Insertion and removal of elements takes place at opposite positions of the data structure
- Like the Stack, the Queue does not make any assumptions on the data structure used for the storage of the elements

enqueue (abc) → dequeue 18ns
Priority queue

- A queue where each element is associated with a priority
- Priority is used for dequeuing
  - We dequeue the highest priority element of the collection
- Same operations as for queue, but further require the priority when enqueuing an element

### enqueue (abc, 10)

<table>
<thead>
<tr>
<th>elements</th>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>6ah4</td>
</tr>
<tr>
<td></td>
<td>skja</td>
</tr>
<tr>
<td></td>
<td>8372</td>
</tr>
<tr>
<td></td>
<td>8rn3</td>
</tr>
<tr>
<td></td>
<td>889sz</td>
</tr>
<tr>
<td></td>
<td>jer82</td>
</tr>
<tr>
<td></td>
<td>dwn</td>
</tr>
<tr>
<td></td>
<td>nn</td>
</tr>
<tr>
<td></td>
<td>18ns</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

### dequeue abc

<table>
<thead>
<tr>
<th>elements</th>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>6ah4</td>
<td>1</td>
</tr>
<tr>
<td>skja</td>
<td>2</td>
</tr>
<tr>
<td>8372</td>
<td>3</td>
</tr>
<tr>
<td>8rn3</td>
<td>4</td>
</tr>
<tr>
<td>889sz</td>
<td>5</td>
</tr>
<tr>
<td>jer82</td>
<td>6</td>
</tr>
<tr>
<td>dwn</td>
<td>7</td>
</tr>
<tr>
<td>nn</td>
<td>8</td>
</tr>
<tr>
<td>18ns</td>
<td>9</td>
</tr>
<tr>
<td>abc</td>
<td>10</td>
</tr>
</tbody>
</table>
Circular queue

• A queue where the last position is connected with the first one
• Often referred to as ring buffer
• Provide enqueue and dequeue operations with the difference that, when the circular queue becomes full, we start all over again
Set

• Collection of unique elements
• Order is not important
• Sets are useful when we want to guarantee uniqueness of an element
• Usually implemented with the help of another data structure like a hash table or a tree
Map

• The map is a collection of <key, value> pairs
• Keys are distinct
• Fundamental operations:
  • add
  • remove
  • lookup
• Maps are typically used to implement dictionaries that translate keys into values
Hash table

• The hash table, or hash map, stores mappings of keys to values in the form of <key, value> pairs
• Specialization of a map that uses a hash function to implement an index for efficient lookup
• Keys are hashed and stored in the index
• Values are stored to buckets that the index points to
• In case there are collisions, buckets may have to be chained to store multiple values that correspond to different keys
Hash table design

Index

Buckets

[DataAI] – Introduction to Algorithms and Data Structures
Angelos-Christos Anadiotis
Linked list

• Collection of elements with no particular order
• Like an array but the size is dynamic
• Usually implemented by linking together individual memory segments

Singly linked list

Doubly linked list
Tree traversal

- Visit each node in a tree exactly one time
- Assume binary trees for simplicity, but can be generalized
- Depth First Search
  - We go as deep as possible on each subtree, before we move on to its sibling
- Breadth First Search
  - We go level by level
In-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Visit root node
  • Recursively traverse the right subtree
In-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Visit root node
  • Recursively traverse the right subtree
In-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Visit root node
  • Recursively traverse the right subtree
In-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Visit root node
  • Recursively traverse the right subtree
In-order tree traversal

- Until all nodes have been visited:
  - Recursively traverse the left subtree
  - Visit root node
  - Recursively traverse the right subtree
In-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Visit root node
  • Recursively traverse the right subtree
In-order tree traversal

- Until all nodes have been visited:
  - Recursively traverse the left subtree
  - Visit root node
  - Recursively traverse the right subtree
In-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Visit root node
  • Recursively traverse the right subtree
Pre-order tree traversal

• Until all nodes have been visited:
  • Visit root node
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
Pre-order tree traversal

- Until all nodes have been visited:
  - Visit root node
  - Recursively traverse the left subtree
  - Recursively traverse the right subtree
Pre-order tree traversal

- Until all nodes have been visited:
  - Visit root node
  - Recursively traverse the left subtree
  - Recursively traverse the right subtree
Pre-order tree traversal

• Until all nodes have been visited:
  • Visit root node
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
Pre-order tree traversal

• Until all nodes have been visited:
  • Visit root node
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
Pre-order tree traversal

• Until all nodes have been visited:
  • Visit root node
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
Pre-order tree traversal

• Until all nodes have been visited:
  • Visit root node
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
Pre-order tree traversal

• Until all nodes have been visited:
  • Visit root node
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree

Angelos-Christos Anadiotis

[DataAI] – Introduction to Algorithms and Data Structures
Post-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
  • Visit root node
Post-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
  • Visit root node
Post-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
  • Visit root node
Post-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
  • Visit root node
Post-order tree traversal

- Until all nodes have been visited:
  - Recursively traverse the left subtree
  - Recursively traverse the right subtree
  - Visit root node

D E B F
Post-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
  • Visit root node
Post-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
  • Visit root node
Post-order tree traversal

• Until all nodes have been visited:
  • Recursively traverse the left subtree
  • Recursively traverse the right subtree
  • Visit root node
Binary tree – Insertion

• Generally, every entry has a key and a value
  • The key might be equal to the value

• New entries are inserted at the leaf level

• Search the key starting from the root until a leaf node is found

• Create a new node with the new entry as a child of the leaf
  • If the key of the new node is less than the leaf, then the new node becomes a left child
  • If the key of the new node is greater than the leaf, then the new node becomes a right child
Binary tree – Insertion
Binary tree – Insertion

Insert: 14
Binary tree – Insertion

Insert: 14, 15
Binary tree – Search

Insert: 14, 15
Lookup: 15
Binary tree – Search

Insert: 14, 15
Lookup: 15
Binary tree – Search

Insert: 14, 15

Lookup: 15

Search complexity: $O(h)^*$

$h$: height of the binary tree
Tree rotation

• Change the structure of the tree without changing the order of the elements
• Move one node up (pivot) and one node down (root)
• Right rotation:
  • Make the pivot the new root
  • Make the old root the right child of the pivot
  • Make the right child of the pivot, the left child of the old root
• Left rotation:
  • Make the pivot the new root
  • Make the old root the left child of the pivot
  • Make the left child of the pivot, the right child of the old root
Tree rotation

• Change the structure of the tree without changing the order of the elements
• Move one node up (pivot) and one node down (root)
• Right rotation:
  • Make the pivot the new root
  • Make the old root the right child of the pivot
  • Make the right child of the pivot, the left child of the old root
• Left rotation:
  • Make the pivot the new root
  • Make the old root the left child of the pivot
  • Make the left child of the pivot, the right child of the old root
Left rotation example
Left rotation example

Root node: 10
Pivot node: 12

Nodes: 9, 11, 14
Left rotation example
Right rotation example
Right rotation example

```
12
  10
   9
   11
pivot
14

root
```

[DataAI] – Introduction to Algorithms and Data Structures  Angelos-Christos Anadiotis
AVL tree

• The problem with Binary Search Trees (BSTs) is that search takes $O(h)$
• In severely skewed trees $\rightarrow O(n)$, $n$: number of nodes
• Need to increase the tree height only when it is absolutely necessary, i.e. there is no more branch left to insert a new node
• AVL trees guarantee that, for every node, the balance factor is within the set {-1, 0, +1}
• The Balance factor of a given node is defined as the height difference of the two subtrees of that node
Rebalancing

• When the balance factor is violated due to insertion or deletion of a node, the AVL needs to rebalance

• Possible rebalancing:
  • Right
  • Left
  • Right-Left
  • Left-Right

• After we apply the rebalancing operations the tree will be considered AVL again
Rebalancing cases on insertion

• Suppose that a node X has balance factor outside the accepted set (e.g., let balance factor be +2)
• Let Z be the highest child of node X
  • insertion was made on Z and that’s why the balance factor of X increased

<table>
<thead>
<tr>
<th>Violations and rebalancing operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Position of Z wrt X</strong></td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>Z is right child of X</td>
</tr>
<tr>
<td>Z is left child of X</td>
</tr>
<tr>
<td>Z is right child of X</td>
</tr>
<tr>
<td>Z is left child of X</td>
</tr>
</tbody>
</table>
AVL insertion example

- Diagram of a binary search tree with nodes 9, 10, 11, 13, and 14.
AVL insertion example

BF = 0
BF = 0
BF = 0
BF = -1

Angelos-Christos Anadiotis
AVL insertion example

**Insert 12**

13

10

9

11

14

12
AVL insertion example

Insert 12
Rebalancing cases on insertion

• Suppose that a node X has balance factor outside the accepted set (e.g., let balance factor be +2)

• Let Z be the highest child of node X
  • insertion was made on Z and that’s why the balance factor of X increased

<table>
<thead>
<tr>
<th>Violations and rebalancing operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Position of Z wrt X</strong></td>
</tr>
<tr>
<td>Z is right child of X</td>
</tr>
<tr>
<td>Z is left child of X</td>
</tr>
<tr>
<td>Z is right child of X</td>
</tr>
<tr>
<td>Z is left child of X</td>
</tr>
</tbody>
</table>
AVL insertion example

[Diagram showing the process of AVL insertion and balance factors]

Left Rotation

[Diagram showing the result of the left rotation]

BF = +1
BF = -1
BF = 0
BF = -2
BF = -1
BF = 0
BF = 0
BF = +1
BF = 0
BF = +1
BF = -2
BF = -1
BF = 0
BF = 0

[DataAl] – Introduction to Algorithms and Data Structures
Angelos-Christos Anadiotis
AVL insertion example

Right Rotation
Divide and Conquer

• Divide
  • Split the problem into smaller instances of the original problem

• Conquer
  • Recursively solve the subproblems

• Combine
  • Put together all solutions to compose the final solution to the original problem
Merge Sort

• Suppose an array A of N elements: A[1..N]
• Merge sort relies on divide-and-conquer method
  • Divide: split the array into the smallest possible unit
  • Conquer: compare each element with its adjacent list and merge

• Worst case computational complexity: O(NlogN)
## Merge sort – Example

| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

Source: [https://en.wikipedia.org/wiki/Merge_sort](https://en.wikipedia.org/wiki/Merge_sort)
## Merge sort – Example

<table>
<thead>
<tr>
<th>38</th>
<th>27</th>
<th>43</th>
<th>3</th>
<th>9</th>
<th>82</th>
<th>10</th>
</tr>
</thead>
</table>

38 27 43 3

9 82 10

Source: https://en.wikipedia.org/wiki/Merge_sort
Merge sort – Example

Source: https://en.wikipedia.org/wiki/Merge_sort
Merge sort – Example

Source: https://en.wikipedia.org/wiki/Merge_sort
Merge sort – Example

Source: https://en.wikipedia.org/wiki/Merge_sort
# Merge sort – Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>38</th>
<th>27</th>
<th>43</th>
<th>3</th>
<th>9</th>
<th>82</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>38</td>
<td>27</td>
<td>43</td>
<td>3</td>
<td>9</td>
<td>82</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>38</td>
<td>27</td>
<td>43</td>
<td>3</td>
<td>9</td>
<td>82</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>38</td>
<td>27</td>
<td>43</td>
<td>3</td>
<td>9</td>
<td>82</td>
<td>10</td>
</tr>
</tbody>
</table>

Source: https://en.wikipedia.org/wiki/Merge_sort
## Merge sort – Example

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>27</td>
<td>43</td>
<td>3</td>
<td>9</td>
<td>82</td>
</tr>
</tbody>
</table>

Source: [https://en.wikipedia.org/wiki/Merge_sort](https://en.wikipedia.org/wiki/Merge_sort)
Merge sort – Example

Source: https://en.wikipedia.org/wiki/Merge_sort
Merge sort – Example

Source: https://en.wikipedia.org/wiki/Merge_sort
### Merge sort – Example

<table>
<thead>
<tr>
<th>38</th>
<th>27</th>
<th>43</th>
<th>3</th>
<th>9</th>
<th>82</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>27</td>
<td>43</td>
<td>3</td>
<td>9</td>
<td>82</td>
<td>10</td>
</tr>
<tr>
<td>38</td>
<td>27</td>
<td>43</td>
<td>3</td>
<td>9</td>
<td>82</td>
<td>10</td>
</tr>
<tr>
<td>38</td>
<td>27</td>
<td>43</td>
<td>3</td>
<td>9</td>
<td>82</td>
<td>10</td>
</tr>
<tr>
<td>27</td>
<td>38</td>
<td>3</td>
<td>43</td>
<td>9</td>
<td>82</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>38</td>
<td>43</td>
<td>9</td>
<td>10</td>
<td>82</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>10</td>
<td>27</td>
<td>38</td>
<td>43</td>
<td>82</td>
</tr>
</tbody>
</table>

Source: https://en.wikipedia.org/wiki/Merge_sort
Merge sort – Example

Source: https://en.wikipedia.org/wiki/Merge_sort
Quicksort

• Divide-and-conquer algorithm
• Similar to Merge Sort, but partitioning is decided on a pivot, which is not given by the algorithm

• Pivot points may be:
  • the first element
  • the last element
  • a random element
  • anything else that makes sense for every use case

• Computational complexity
  • Worst case: $O(N^2)$
  • Best case: $O(N\log N)$ or even $O(N)$ based on the implementation
Quicksort steps

• Pick a pivot element (e.g., the last one) and compare it with all elements from the beginning of the partition

• For every element which is greater than the pivot, move the greater element after the pivot by moving the pivot from position k to position k-1 and exchanging the element at position k-1 with the element to be placed after the pivot

• Repeat recursively

➢ The goal is to have all the elements greater than the pivot, being placed after it, and all the elements less than the pivot, placed before it
Quicksort – Example

Pivot
Current
Final

3  7  8  5  2  1  9  5  4

Source: https://en.wikipedia.org/wiki/Quicksort
# Quicksort – Example

<table>
<thead>
<tr>
<th>Pivot</th>
<th>Current</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quicksort – Example

3 7 8 5 2 1 9 5 4
Quicksort – Example

3 7 8 5 2 1 9 5 4

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

[Diagram showing a quicksort example with numbers 3 7 8 5 2 1 9 5 4, highlighting the pivot and current elements.]

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

[Image: Quicksort algorithm example with a pivot, current, and final states of the array]

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

3 7 8 5 2 1 9 5 4

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

3 5 8 5 2 1 9 4 7

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

3 9 8 5 2 1 4 5 7

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Pivot
Current
Final

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Pivot: 3
Current: 1 8 5 2 4 9
Final: 5 7

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

3 1 2 5 4 8 9 5 7
Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

3 1 2 4 5 8 9 5 7
Quicksort – Example

3 1 2 4 5 8 9 5 7

3 1 2 5 8 9 5 7

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

```
3 1 2 4 5 8 9 5 7
1 2 3 5 5 7 9 8
```

Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
Quicksort – Example

Source: https://en.wikipedia.org/wiki/Quicksort
String matching

- Given a text of \( n \) characters \( T[1..n] \) and a pattern of \( m \) characters \( P[1..m] \), we want to find whether \( P \) exists within \( T \) with \( m \leq n \)
- The naïve implementation will take every character of \( T \) and start matching it with every character of \( P \)
  - If a pair of characters do not match, then we shift the check in \( T \) by one character and we repeat the process
  - Computational complexity is thus \( O((n - m + 1) \times m) \)
- Knuth-Morris-Pratt algorithm executes the search in \( O(n) \)
Knuth-Morris-Pratt algorithm

• Intuition: maintain information about a matching prefix while searching
Knuth-Morris-Pratt algorithm

- Intuition: maintain information about a matching prefix while searching
Knuth-Morris-Pratt algorithm

• Intuition: maintain information about a matching prefix while searching
Knuth-Morris-Pratt algorithm

• Intuition: maintain information about a matching prefix while searching
Knuth-Morris-Pratt algorithm

- Intuition: maintain information about a matching prefix while searching

Find the longest prefix $P_k$ of $P_q$ which is also a suffix of $T_{s+q}$
Knuth-Morris-Pratt algorithm

- Intuition: maintain information about a matching prefix while searching

Find the longest prefix $P_k$ of $P_q$ which is also a suffix of $T_{s+q}$
Knuth-Morris-Pratt algorithm

• Intuition: maintain information about a matching prefix while searching

```
b a c b a b a b a a a b c b a a b
```

\[ s' = s + 2 \]

```
T
```

```
P
```

```
a b a b a c a
```
Knuth-Morris-Pratt algorithm

- Intuition: maintain information about a matching prefix while searching
Knuth-Morris-Pratt algorithm

• Use the pattern itself to deduct how many positions we need to shift forward

• Suppose that q characters of the pattern have been matched and that k out of the q characters appear in both the prefix and the suffix of the pattern
  • Note that the prefix and the suffix characters may overlap

➢ Then, we know that we can shift the pattern P by the number of characters required to make its prefix match the suffix of the text T
Knuth-Morris-Pratt algorithm

- Maintain an array $p[1..m]$, with its every position $p[i]$ giving the length of the longest prefix of $P$ that is a suffix of $P_i$
- The array $p$ is precomputed and used throughout the search
- Every time that we need to shift on the text $T$ after having matched the first $q$ characters of the pattern $P$, we shift by $(q – p[q])$
Minimum spanning tree

• Suppose an undirected, weighted graph \( G = (V, E) \)
• A spanning tree is a tree that connects all the vertices of the graph
• The minimum spanning tree is the spanning tree with the minimum weight, where the weight of the tree is defined as the sum of the weights of all edges
Minimum spanning tree example
Minimum spanning tree example
Minimum spanning tree example
Kruskal MST algorithm

• Greedy algorithm
• Consider a forest of trees
  • Initially the forest includes just the graph vertices
• Consider the set of all edges with their weights
• While the set of edges is not empty and while we do not have a single tree
  • Get the edge with the smallest weight
  • If the edge connects two trees, then add it to the forest
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example

[Diagram of a graph with edges and weights, illustrating the Kruskal MST algorithm example.]
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example
Kruskal MST algorithm – Example

[Diagram of a graph with labeled vertices and edges, illustrating the Kruskal algorithm to find the Minimum Spanning Tree.]
Readings


- The web is full of information about data structures and algorithms – pick the source that works best for you