Mahidol University

Faculty of Information and Communication Technology

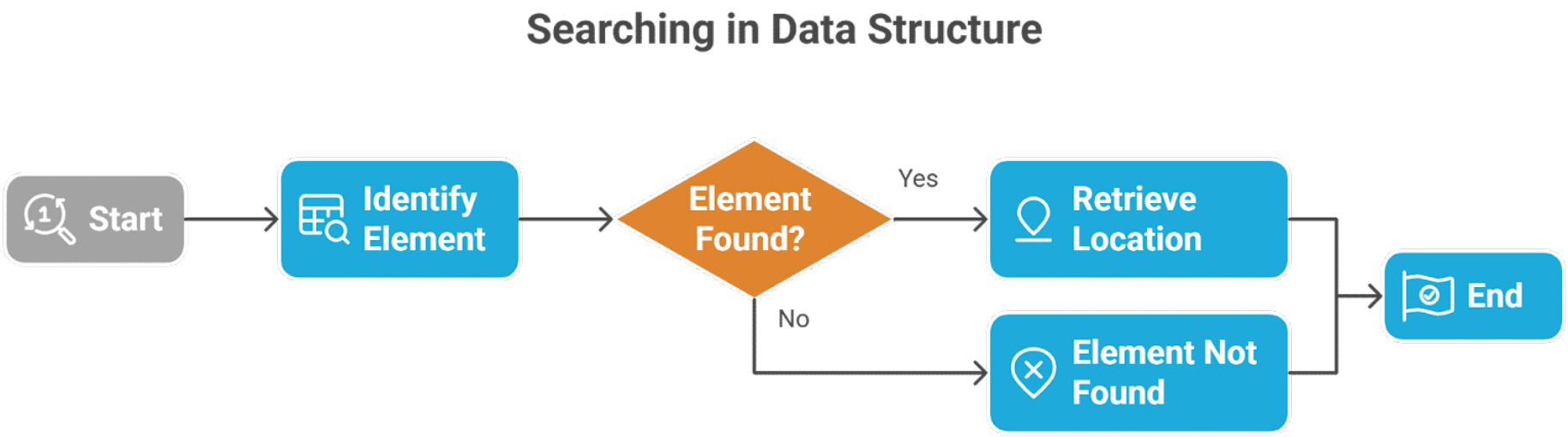
Searching

ITCS 503 – Design and Analysis of Algorithms

ปัญญาของแผ่นดิน Wisdom of the Land

Searching

- Searching in data structure and algorithms refer to the process of identifying the location of the item or element that we are looking for among a set of items or in a repository.



- Searching in data structures is crucial for efficient data retrieval, thereby enhancing performance in databases, search engines, and AI systems. These algorithms enable the rapid identification of specific data points, enhancing performance in databases and search engines.

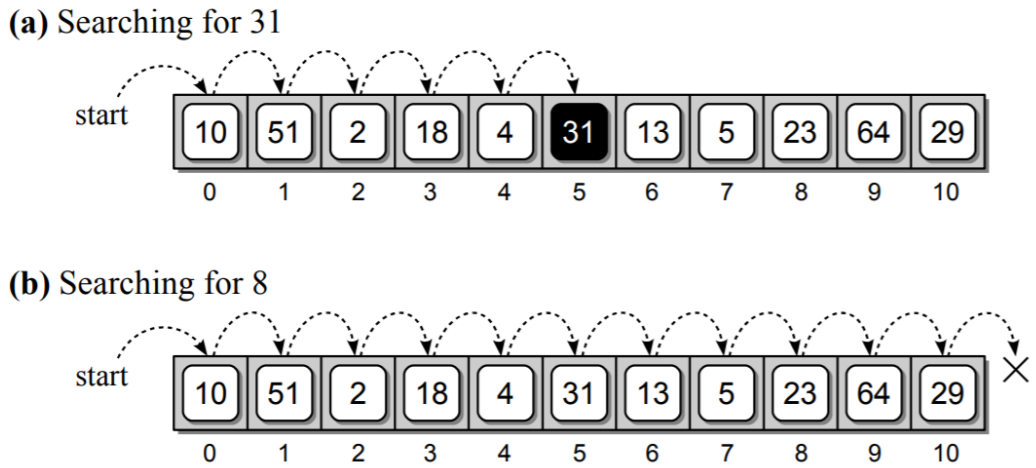
Searching

- **Numerical search**
  - Search for a number in a list or an array
    - Problem: Given an array `arr[]` of `n` elements, write a function to search a given element `x` in `arr[]`.
  - Reference: https://www.geeksforgeeks.org/searching-algorithms/ (codes can be found here)
    - Linear search
    - Binary search
    - Jump search

- **Text search**
  - Search for a pattern from a text
  - Reference: https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/ (codes can be found here)
    - Naïve search
    - KMP algorithm
    - Boyer Moore Algorithm
    - Z algorithm
    - Regular expression

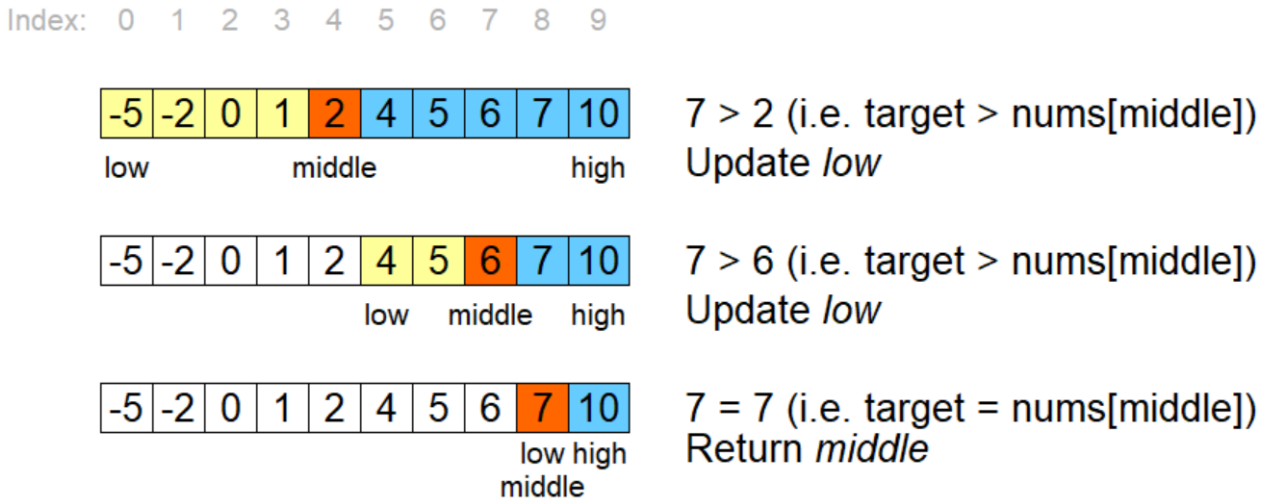
Numerical search: Linear Search

- **Principle**
  - Keep finding the key value `x` in `arr[]` one element at a time.
- **Approach (Heuristic)**
  - Start from the leftmost element of `arr[]` and one by one compare `x` with each element of `arr[]`
  - If x matches with an element, return the index.
  - If x doesn't match with any of elements, return -1.
- **Time Complexity Performance**: $O(n)$ where n is the size of the array.



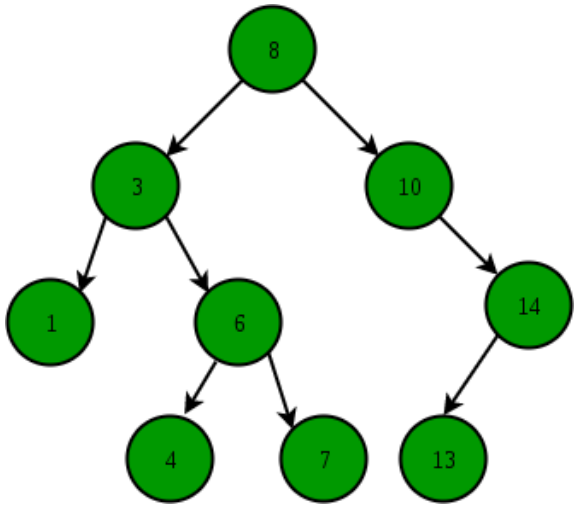
Numerical search: Binary Search

- **Principle**
  - If `arr[]` is already **sorted**, we should be able to find the key value `x` faster by cutting down the search space.
- **Approach (Heuristic)**
  - Begin with an interval covering the whole array.
  - If the key value `x` is less than the item in the middle of the interval, narrow the interval to the lower half.
  - Otherwise, the key value `x` must be in the upper half.
  - Repeatedly check until the value is found or the interval is empty.
- **Time Complexity Performance**: $O(\log n)$ where n is the size of the array.



Binary Search Tree

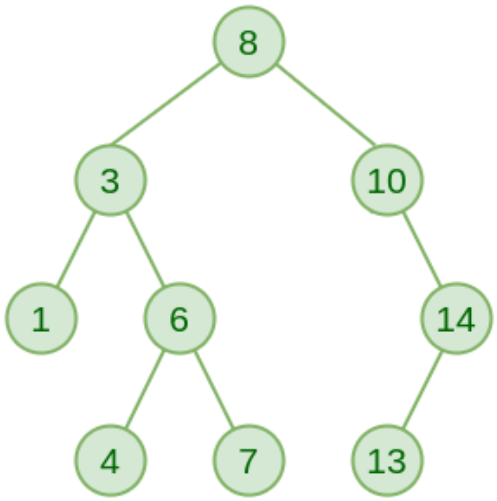
- Binary Search Tree is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys lesser than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree each must also be a binary search tree.
  - There must be no duplicate nodes.



Pay attention to how a new node is inserted to the tree.

Numerical Search: Binary Search Tree

- **Principle**
  - Use binary tree structure to help with the search.
  - The tree must follow the following rules:
    - The value of the left child is always less than of equal to the value of the parent
    - The value of the right child is always greater than the value of the parent
- **Approach (Heuristic)**
  - Construct the binary search tree
  - Search for the key value x by traversing the tree (divide and conquer)
    - In-order traversal
    - Pre-order traversal
    - Post-order traversal
- **Performance**: $O(\log n)$ where n is the size of the array.



Binary Search Tree

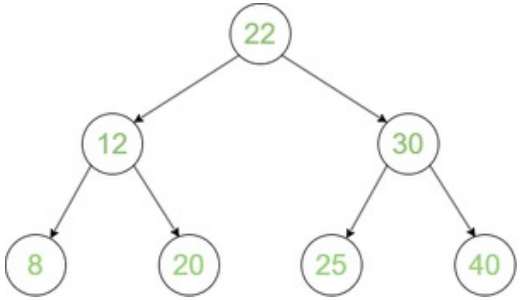
Numerical Search: Binary Search Tree

- **Traveling down the tree (search path): Ordered traversal**

```
def printInorder(root):
    if root:
        # Traverse left subtree
        printInorder(root.left)

        # Visit node
        print(root.data,end=" ")

        # Traverse right subtree
        printInorder(root.right)
```

```
def preOrder(node):
    if not node:
        return

    print node.data,
    preOrder(node.left)
    preOrder(node.right)
```

```
def printPostOrder(node):
    if node is None:
        return

    # Traverse left subtree
    printPostOrder(node.left)

    # Traverse right subtree
    printPostOrder(node.right)

    # Visit Node
    print(node.data, end = " ")
```



Output:

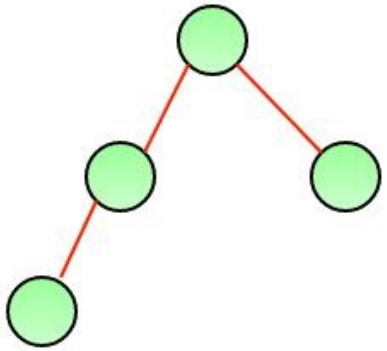
Inorder Traversal: 8 12 20 22 25 30 40

Preorder Traversal: 22 12 8 20 30 25 40

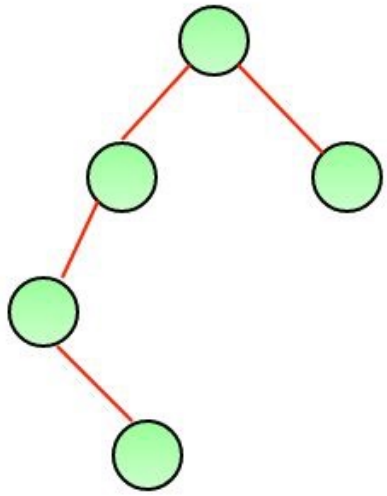
Postorder Traversal: 8 20 12 25 40 30 22

Numerical Search: Binary Search Tree

- **Issue**:
  - A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes.
  - A height-unbalanced tree introduces unbalanced search performance. The time performance of the binary tree search could be greater than O(Log n).
  - Balanced Binary Search trees are performance-wise good as they provide O(log n) time complexity for search, insert and delete.



A height balanced tree



Not a height balanced tree

Numerical Search: Binary Search Tree

- **Issue**:
  - Height-unbalanced tree: check if the tree is height-balanced

```
class Node:
    # Constructor to create a new Node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# function to find height of binary tree


def height(root):

    # base condition when binary tree is empty
    if root is None:
        return 0
    return max(height(root.left), height(root.right)) + 1
```

```
def isBalanced(root):

    # Base condition
    if root is None:
        return True

    # for left and right subtree height
    lh = height(root.left)
    rh = height(root.right)

    # allowed values for (lh - rh) are 1, -1, 0
    if (abs(lh - rh) <= 1) and isBalanced(
            root.left) is True and isBalanced(root.right) is True:
        return True

    # if we reach here means tree is not
    # height-balanced tree
    return False


# Driver function to test the above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.left.left = Node(8)
if isBalanced(root):
    print("Tree is balanced")
else:
    print("Tree is not balanced")
```

Numerical Search: Binary Search Tree

- **Issue**:
  - Height-unbalanced tree: solve by balancing the height of the tree.
  - **Method 1**: Complete rearrangement of the tree.
    - Traverse the tree to get the array of numbers
    - Use the midpoint of the array as the root of the BST
    - Construct the tree

```
# binary tree node
class Node:
    def __init__(self, d):
        self.data = d
        self.left = None
        self.right = None
```

```
def sortedArrayToBST(arr):

    if not arr:
        return None

    # find middle index
    mid = (len(arr)) // 2

    # make the middle element the root
    root = Node(arr[mid])

    # left subtree of root has all
    # values <arr[mid]
    root.left = sortedArrayToBST(arr[:mid])

    # right subtree of root has all
    # values >arr[mid]
    root.right = sortedArrayToBST(arr[mid+1:])
    return root
```

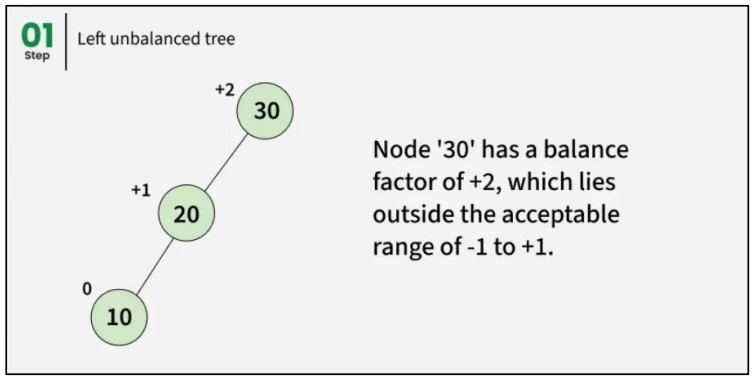
The initial array must already be sorted.

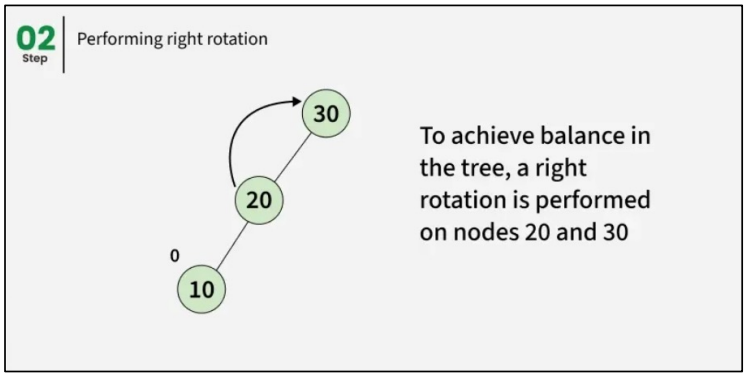
Numerical Search: Binary Search Tree

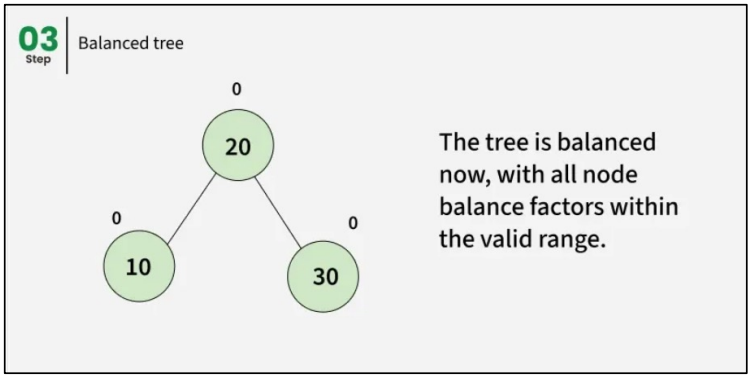
- **Issue**:
  - Height-unbalanced tree: solve by balancing the height of the tree.
  - **Method 2**: Rotate the tree for every insertion (AVL Tree - Adelson-Velsky and Landis Tree)
    - Check the heights of the root's left tree and the root's right tree.
    - If the difference between the heights are greater than 1, then perform appropriate rotation.
    - Rotations: rotations are designed to restore balance in O(1) time while ensuring the overall time complexity remains O(log n). AVL Trees use four types of rotations to rebalance themselves after insertions and deletions:
      - Left-Left (LL) Rotation
      - Right-Right (RR) Rotation
      - Left-Right (LR) Rotation
      - Right-Left (RL) Rotation

AVL Tree Rotation

- **Left-Left Rotation**:
  - **Occurs** when a node is inserted into the left subtree of the left child, causing the height difference to be > 1.
  - **Fix**: Perform a single right rotation.

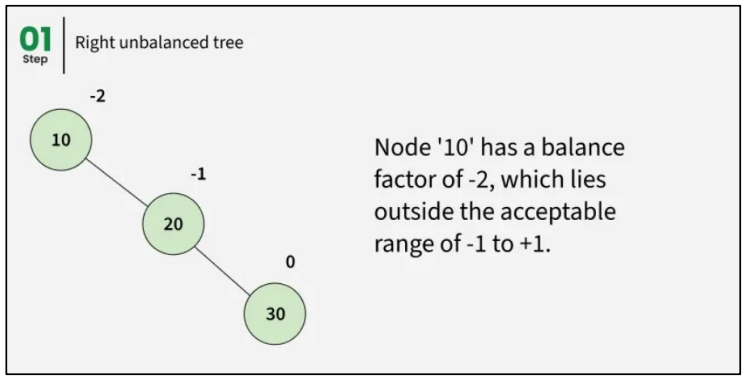


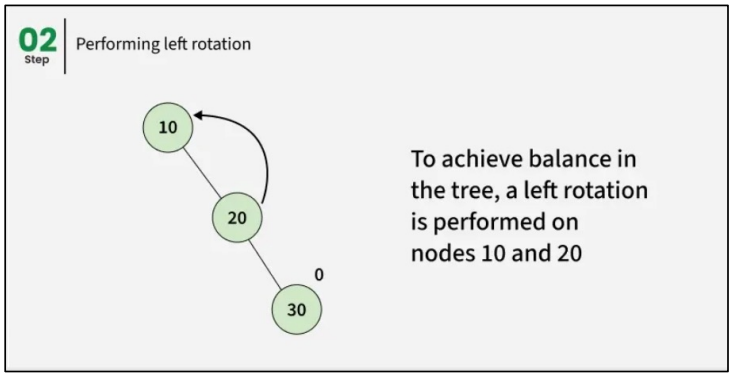


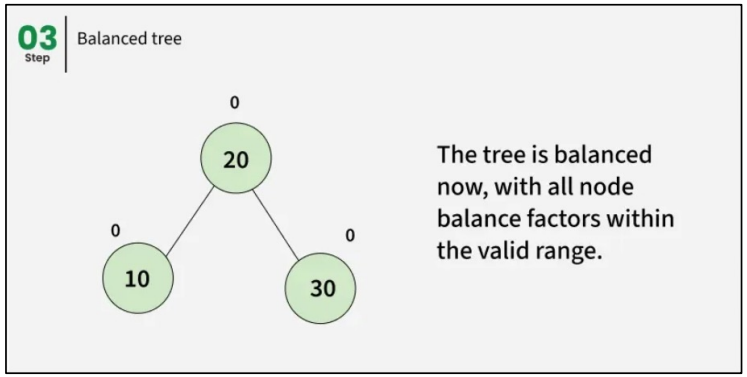


AVL Tree Rotation

- **Right-Right Rotation**:
  - **Occurs** when a node is inserted into the right subtree of the right child, making the balance factor less than -1.
  - **Fix**: Perform a single left rotation.

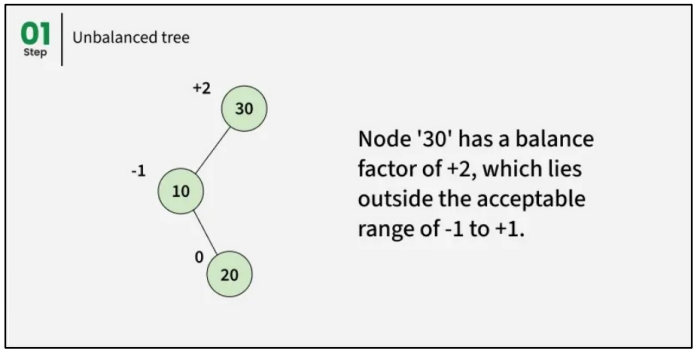


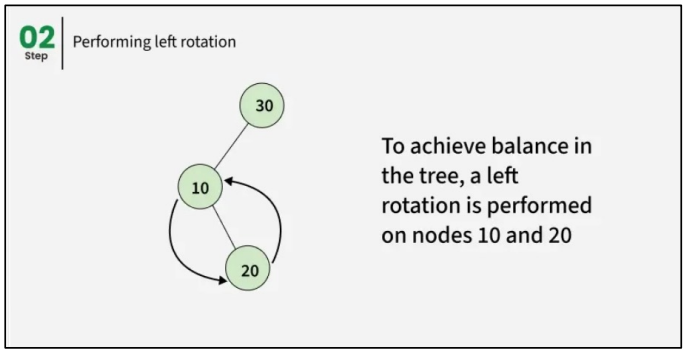


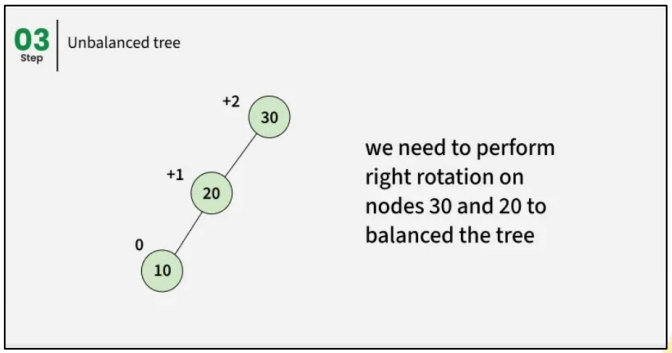


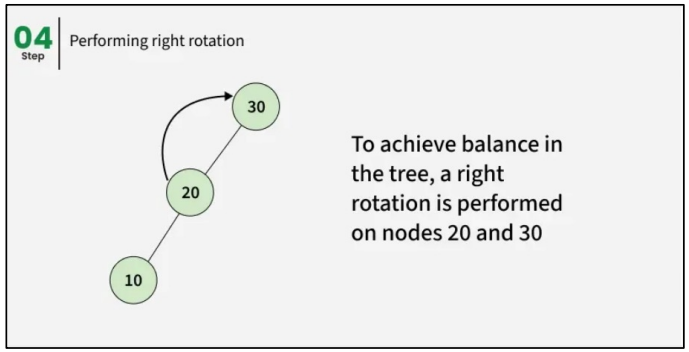
AVL Tree Rotation

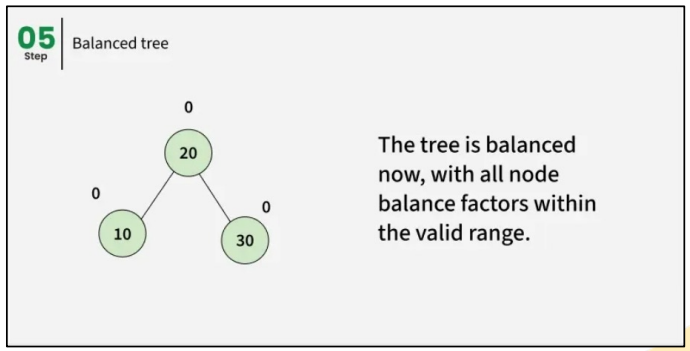
- **Left-Right Rotation**:
  - **Occurs** when a node is inserted into the right subtree of the left child, which disturbs the balance factor of an ancestor node, making it left-heavy.
  - **Fix**: Perform a left rotation on the left child, followed by a right rotation on the node.





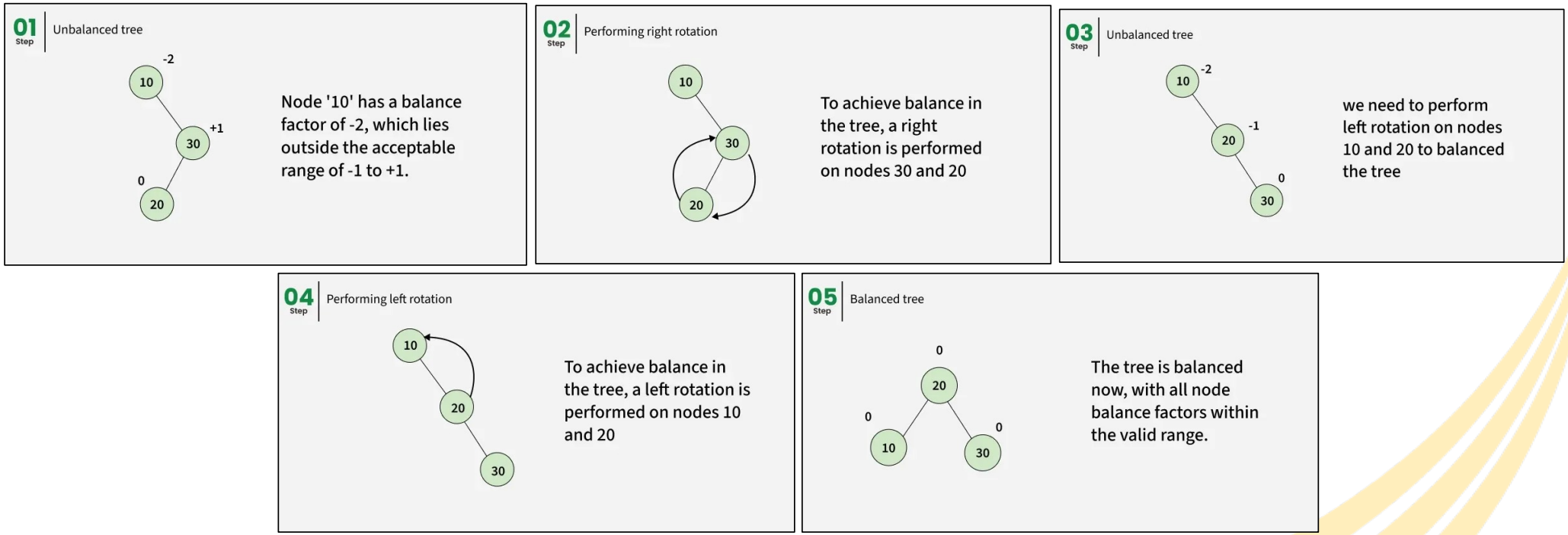






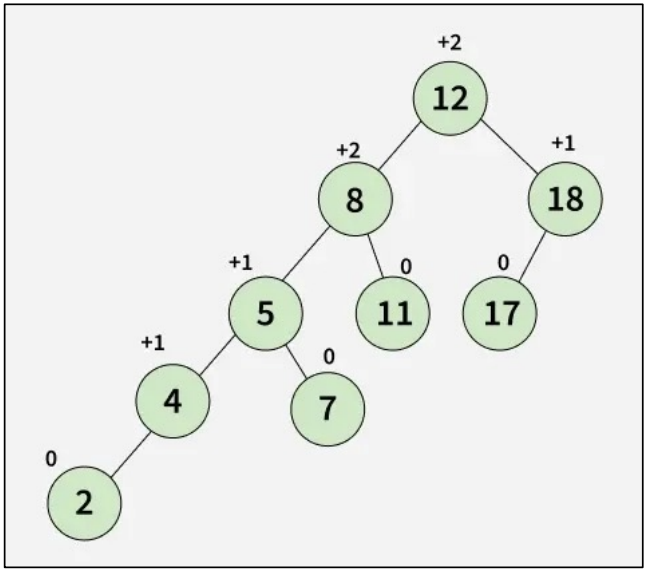
AVL Tree Rotation

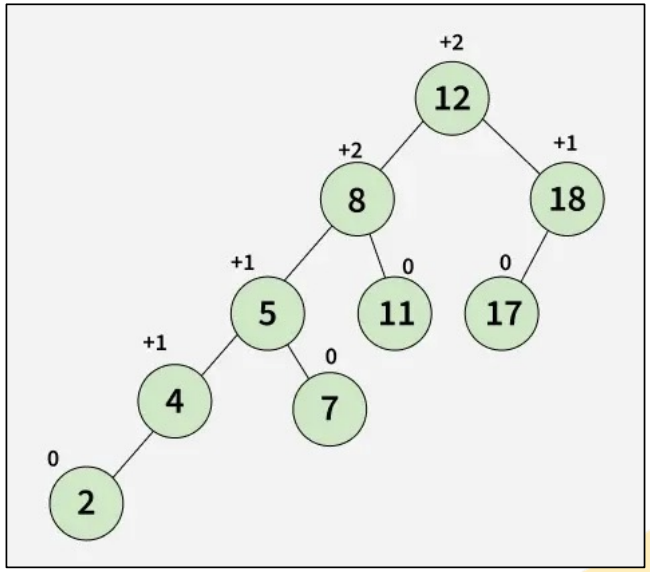
- **Right-Left Rotation**:
  - **Occurs** when a node is inserted into the left subtree of the right child, which disturbs the balance factor of an ancestor node, making it right-heavy.
  - **Fix**: Perform a right rotation on the right child, followed by a left rotation on the node.



Homework 1

- Given a list of elements: [12, 8, 18, 5, 11, 17, 4, 7, 2]
  1. Construct a binary tree by adding one element at a time by using the first element as the root of the tree and remain the root of the tree. Show the tree image and determine whether the tree is a balanced tree by showing the height of the root's left tree and the root's right tree.
  2. Generating 5 random lists of numbers of size n = 20 and repeat step 1.
  3. Choosing all unbalanced trees from 2. Construct balanced binary tree using method 1 balancing process.
  4. Choosing all unbalanced trees from 2. Construct balanced binary tree using method 2 balancing process.

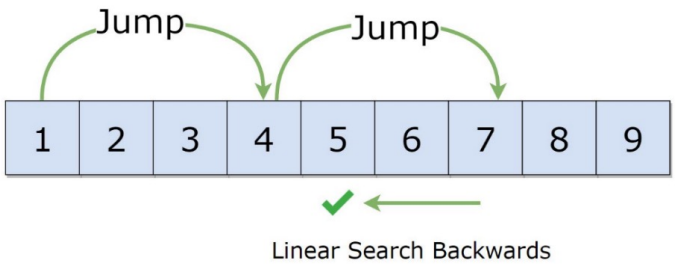




Numerical search: Jump Search

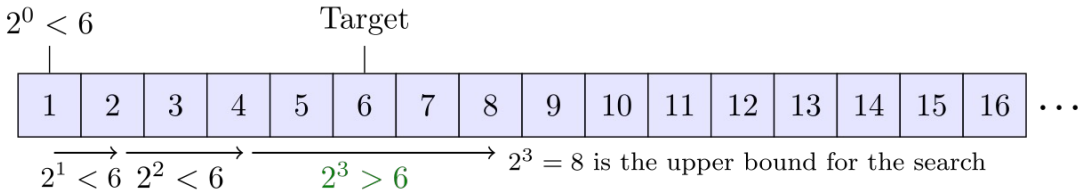
- **Principle**
  - If `arr[]` is already sorted, we should be able to find the key value `x` faster by jumping to the next element.
- **Approach (Heuristic)**
  - Start from the leftmost element of `arr[]` and jump by `m`
  - Compare the current element in `arr[]` and `x`
    - If the current element in `arr[]` matches `x`, return the index.
    - If `x` is less than the element in `arr[]`, perform backward linear search for `x` until x is found.
      - If x is not found, return -1
    - If `x` is greater than the element, jump to the next element by equal-size `m`. (this can be cleverly adjusted)

- **Performance**: between $O(n)$ and $O(\log n)$ where n is the size of the array. Worst case $O((n/m) + m - 1)$
- Note: you can jump according to Fibonacci number or 2^k numbers (similar to exponential search)



Numerical search: Exponential Search

- **Principle**
  - If `arr[]` is already sorted, we should be able to find the key value `x` faster by increasing the search space.
- **Approach (Heuristic)**
  - Begin with an interval of size 1 (left most element).
  - If the key value `x` is greater than the last item in the interval, increase the interval size by 2.
  - Otherwise, the key value `x` must be in the upper half of the interval.
  - Repeatedly check until the value is found or the interval is empty.
- **Performance**: $O(\log n)$ where n is the size of the array.



Homework 2

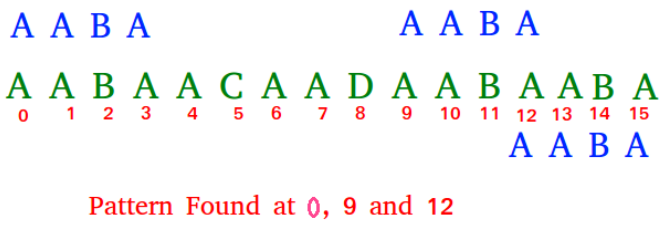
1. Generate 5 random lists for each n = 50, 100, 300, 500, 800, 1000.
2. Implement search algorithms
   - Implement linear search
   - Implement balanced binary tree search using method 1 and method 2
   - Implement jump search using equal jump and fibonacci jump
   - Implement exponential search
3. Plot average time steps for all searching algorithms in steps 2. Analyze your result and see if the results follow the theoretical time complexity provided in class.

Text Search: Naïve Search

- Text search or string search is an algorithm that finds a specific pattern in a text.

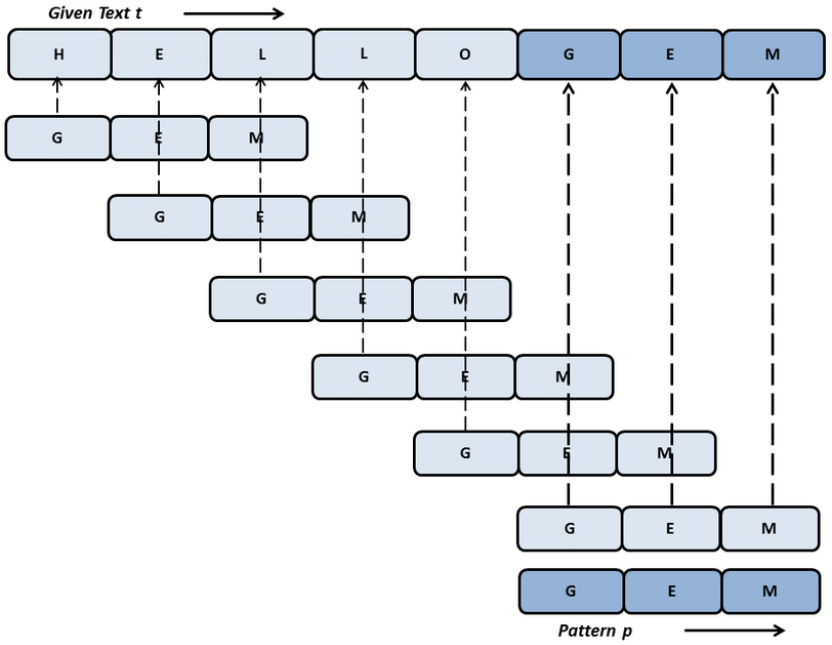
Text : A A B A A C A A D A A B A A B A size **n**

Pattern : A A B A size **m**



- **Principle**:
  - Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
- **Performance**: $O((n - m + 1) * m)$ in the worst case.

Given a text `t[0..n-1]` and a pattern `p[0..m-1]`, write a function search for `p[]` in `t[]`) and prints all occurrences of `p[]` in `t[]`. Assume that `n > m`.



Text Search: KMP (Knuth Morris Pratt) Algorithm

- **Principle**
  - If there is a mismatch, we must shift the pattern intelligently.

- **Approach (Heuristic)**
  - The proper suffixes of the matched part of the pattern, will be compared with the proper prefixes of the matched part of the pattern. The longest match between the two will be used for shifting.

- **Definitions**
  - A proper prefix and proper suffix are prefix and suffix except the whole string.

  - For example,
    - Prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB".
    - Suffixes of "ABC" are "", "C", "BC" and "ABC". Proper suffixes are "", "C", and "BC"

Text Search: KMP (Knuth Morris Pratt) Algorithm

- Example

text

a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	b	c	y

pattern

a	b	c	d	a	b	c	y

First: place the pattern at position 0 of the text. Determine:

a	b	c	d	a	b	c	y

mismatched

proper suffixes: y, cy, bcy, abcy **Matched proper prefix**: abc

Is there **a proper suffix that is also a prefix** of **this matched part** of the pattern?

No. So the pattern is jumped to here.

a	b	c	d	a	b	c	y

No match at the beginning, so we shift here

a	b	c	d	a	b	c	y

Is there **a suffix that is also a prefix** of **this matched part** of the pattern?

Yes – abc and abc
So we jump here.

a	b	c	d	a	b	c	y

And so on

Text Search: KMP (Knuth Morris Pratt) Algorithm

- Example

a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	b	c	y
a	b	c	d	a	b	c	y															
			a	b	c	d	a	b	c	y												
				a	b	c	d	a	b	c	y											
							a	b	c	d	a	b	c	y								
								a	b	c	d	a	b	c	y							
										a	b	c	d	a	b	c	y					
											a	b	c	d	a	b	c	y				
															a	b	c	d	a	b	c	y

The key is to compute the number of matches between the prefix and suffix for different part of the pattern. This can be pre-computed (usually called lps[] array), and this gives the KMP time of $O(n+m)$. Each element in the text is being compared once on average!

Text Search: KMP (Knuth Morris Pratt) Algorithm

- Reference:
  - https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
  - https://www.youtube.com/watch?v=GTJr8OvyEVQ
  - https://www.ics.uci.edu/~eppstein/161/960227.html

Text Search: Boyer Moore Algorithm

- **Principle**
  - Performs the comparisons from right to left and then jumps based on mismatch similar to KMP
- **Approach (Heuristic)**
  - **Bad Character Heuristic**
    - Case 1 – Mismatch become match
    - Case 2 – Pattern move past the mismatched character
  - **Good Suffix Heuristic**
    - Case 1 - Another occurrence of t in P matched with t in T
    - Case 2 - A prefix of P, which matches with suffix of t in T
    - Case 3 - P moves past t

Text Search: Boyer Moore Algorithm: Bad Character

- **Bad Character**: Case 1 – Mismatch become match
  - We will lookup the position of the last occurrence of the mismatched character in the pattern,
    - if the mismatched character exists in the pattern, then we'll shift the pattern such that the corresponding mismatched character in the text T is aligned with the last occurrence of the character in the prefix of the pattern.

G	C	A	A	T	G	C	C	A	A	T	G	T	G	A	C	C
A	**A**	T	G	T	G											
		A	**A**	T	G	T	G									

The 2ND A is **the last occurrence of the mismatched character in the pattern**!
So, we shift the pattern to this location

Text Search: Boyer Moore Algorithm: Bad Character

- **Bad Character**: Case 2 – Pattern move past the mismatched character
  - If the mismatched character in the text does not appear in the pattern, just shift pass it.

G	C	A	A	T	G	C	C	A	A	T	G	T	G	A	C	C
A	**A**	T	G	T	G											
		A	**A**	T	G	T	G									
								A	A	T	G	T	G			

C is not in the pattern.

So, we shift the entire pattern to this location

Text Search: Boyer Moore Algorithm: Good Suffix

- **Good Suffix**: Case 1 – Another occurrence of t in P matched with t in T (t is a substring of P)
  - Pattern P might contain few more occurrences of t (a matched part of the text). In such case, we will try to shift the pattern to align the first occurrence with the first backward-matched t in text T (matched suffix).

A	B	A	A	B	A	B	A	C	B	A						
C	A	B	A	B												
		C	A	B	A	B										

AB in the pattern matched AB in t

So, we shift the pattern to this location

Text Search: Boyer Moore Algorithm: Good Suffix

- **Good Suffix**: Case 2 – A prefix of P, which matches with suffix of t in T
  - It is not always likely that we will find the occurrence of t in P. Sometimes there is no occurrence at all, in such cases sometimes we can search for some matched suffix of t matching with some prefix of P and try to align them by shifting P.

A	A	B	A	B	A	B	A	C	B	A						
A	**B**	B	A	B												
			A	**B**	B	A	B									

AB in the pattern matched AB in subset of t

So, we shift the pattern to this location

Text Search: Boyer Moore Algorithm: Good Suffix

- **Good Suffix**: Case 3 – P moves past t
  - If the case 1 and 2 are not satisfied, we will shift the pattern past the t.

A	A	C	A	B	A	B	A	C	B	A						
C	B	A	A	B												
					C	B	A	A	B							

AB in the pattern does not matched prefix of P

So, we shift the pattern to this location

Text Search: Boyer Moore Algorithm: Test Your Concept

- **Which rule?**:
  - Determine where to move.

A	B	A	A	B	A	B	A	C	B	A						
A	A	B	A	B												
		A	A	B	A	B										

Text Search: Boyer Moore Algorithm: Good Suffix

- **Performance**: Boyer Moore algorithm is considered the fastest string search algorithm (its variants may perform a bit faster). Just like KMP, each character in the text is compared once. On average, it takes time $\boldsymbol{O(n/m)}$.

- **Preprocessing**: Boyer Moore algorithm requires the preprocessing of characters, prefix and suffix.
  - There is a table containing shift values for each character (bad character) and subsets of pattern prefixes (good suffix). The time complexity is $\boldsymbol{O(m)}$ – to learn the location of all characters in the pattern. The characters in the text that is not in the pattern will give maximum shift.
  - Finding the subsets of the pattern prefixes and suffixes can be considered trivial. (See: https://www.geeksforgeeks.org/number-substrings-string/ and https://www.geeksforgeeks.org/program-print-substrings-given-string/) The time complexity is $\boldsymbol{O(m^2)}$

- **Implementation:**
  - https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/?ref=lbp
  - https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/
  - https://www-igm.univ-mlv.fr/~lecroq/string/node14.html

Text Search: Z Algorithm

- **Principle**:
  - Similar to KMP, the pattern comparison is shifted when a mismatch is found.
  - The shift amount is pre-computed using the Z array.

- **Definition of Z array**:
  - For a string str[0..n-1], Z array is of same length as the text string.

  - An element Z[i] = length of the longest prefix substring starting from str[i] which is also a prefix of P (str[0..n-1]).
  - The first entry of Z array is not computed since complete string is always prefix of itself.

i	0	1	2	3	4	5	6						
text	a	a	b	a	a	c	d						
	X	1											
	X	1	0										
	X	1	0	2									
	X	1	0	2	1								
	X	1	0	2	1	0							
	X	1	0	2	1	0	0						

- i = 1: "a" is the longest prefix substring of "abaacd" which is also a prefix of "aabaacd", matched_len = 1.
- i = 2 "" is the longest prefix substring of "baacd" which is also a prefix of "aabaacd", matched_len = 0
- i = 3 "aa" is the longest prefix substring of "aacd" which is also a prefix of "aabaacd", matched_len = 2
- i = 4 "a" is the longest prefix substring of "acd" which is also a prefix of "aabaacd", matched_len = 1
- i = 5 "" is the longest prefix substring of "cd" which is also a prefix of "aabaacd", matched_len = 1
- i = 6 "" is the longest prefix substring of "d" which is also a prefix of "aabaacd", matched_len = 1

Text Search: Z Algorithm

- Z Array examples

pattern	a	a	a	a	a	a							
Z array	X	5	4	3	2	1							
pattern	a	b	a	b	a	b	a	b					
Z array	X	0	6	0	4	0	2	0					
pattern	a	a	b	c	a	a	b	x	a	a	a	z	
Z array	X	1	0	0	3	1	0	0	2	2	1	0	

Text Search: Z Algorithm

- How is the Z Array used to help with the search?

	0	1	2	3	4	5	6	7	8	9			
text	b	a	a	b	a	a							
pattern	a	a	b										
pattern $ text	a	a	b	**$**	b	a	a	b	a	a			
Z array (p$t)	X	1	0	0	0	3	1	0	2	1			

Since **the length** of the pattern is 3 and it is equal to the z array of p$t[**5**] == 3.
Then the pattern is found at location **i == 5**.

- **Performance**: The construction of the z array use $\boldsymbol{O(m^2)}$ and the search for "m" in the Z-array is $\boldsymbol{O(n)}$ (linear search).

Regular Expression

- A regular expression (sometimes called RegEx a rational expression) is a sequence of characters that define a search pattern. It is mainly used for string matching, searching, and manipulation in text processing. Regular expressions are widely used in programming, data validation, and text parsing.

- Regular expressions are a generalized way to match patterns with sequences of characters. It is used in every major programming language like C++, Java and Python.

- How to write RegEx string matching rules:
  - https://www.geeksforgeeks.org/write-regular-expressions/
  - https://www.geeksforgeeks.org/python-regex-cheat-sheet/
  - https://regexr.com/ (RegEx Reference menu)
  - https://www.programiz.com/python-programming/regex
  - https://www.dataquest.io/wp-content/uploads/2019/03/python-regular-expressions-cheat-sheet.pdf
  - https://docs.python.org/3/howto/regex.html
  - https://www.tutorialspoint.com/python/python_reg_expressions.htm

- RegEx tester:
  - https://regexr.com/

Regular Expression

- Basic Components of Regular Expressions
  - Literals are the normal characters that match exactly as they appear.
    - Example: "cat" matches the word cat in a text.
  - Metacharacters are special characters used to assist sophisticated matching in regex.
  - Quantifiers define the number of repetitions that the match should appear.

Metacharacter	Meaning	Example
`.`	Matches any single character except newline	`c.t` matches **cat**, **cut**, **cot**
`^`	Matches the start of a string	`^Hello` matches **Hello world**, but not **Hi Hello**
`$`	Matches the end of a string	`world$` matches **Hello world**, but not **worldly**
`[]`	Matches any one character inside the brackets	`[abc]` matches **a**, **b**, or **c**
`|`	`|`	OR operator
`\`	Escape character to use special characters literally	`\.` matches a **dot (.)** instead of any character

Quantifier	Meaning	Example
`*`	Matches **0 or more** occurrences	`ba*` matches **b**, **ba**, **baa**, **baaa**
`+`	Matches **1 or more** occurrences	`ba+` matches **ba**, **baa**, **baaa**, but not **b**
`?`	Matches **0 or 1** occurrence	`ba?` matches **b** or **ba**
`{n}`	Matches exactly **n** occurrences	`a{3}` matches **aaa**
`{n,}`	Matches **n or more** occurrences	`a{2,}` matches **aa**, **aaa**, **aaaa**, etc.
`{n,m}`	Matches **between n and m** occurrences	`a{2,4}` matches **aa**, **aaa**, or **aaaa**

Regular Expression

- Examples

a) Matching a Phone Number

Regex: `\d{3}-\d{3}-\d{4}`

Matches: **123-456-7890**

b) Validating an Email

Regex: `[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}`

Matches: user@example.com

c) Extracting Dates (dd/mm/yyyy or dd-mm-yyyy)

Regex: `\b\d{2}[-/]\d{2}[-/]\d{4}\b`

Matches: **12/05/2024** or **12-05-2024**

```python
import re

text = "Contact me at john.doe@example.com or visit example.com."
pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"

match = re.search(pattern, text)
if match:
    print("Email found:", match.group())
```

Output:

```
Email found: john.doe@example.com
```

Homework 3

- **Password Validation:** Write a regular expression program (python) to validate a password based on the following rules:
  - Must be at least 8 characters long.
  - Must contain at least one uppercase letter.
  - Must contain at least one lowercase letter.
  - Must contain at least one digit.
  - Must contain at least one special character (!@#$%^&*()-+=).

- **Date Extraction**: Write a regular expression program (python) to extract date from a text. Date should be in the following formats:
  - dd/mm/yyyy
  - dd-mm-yyyy
  - dd mmm yy (e.g., 14 July 25)
  - dd mmm yyyy (e.g., 4 July 2025)