Data Structures n Algorithms

Hai Hoang Minh

18 Oct
2004

Dynamic n Static Arrays

what is a static Arrays?

A static arrays is a fixed length container containing n elements indexable from the range [0,n-1]

Indexable This mean that each slot/ index in the array can be referred with a number

When and where is a static Array used?

1. Storing n accessing sequential data
2. Temporarily storing objects
3. Used by IO routines as buffers - bộ đệm
4. Look up tables n inverse look up tables
5. Can be used to return multiple values from a function.
6. Used in dynamic programming to cache answers O(1) to subproblems.

Complexity

	Static	Dynamic
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	N/A	O(n)
Appending	N/A	O(1)
Deleting	N/A	O(n)

Static arrays

A =	44	12	-5	17	6	0	3	9
	0	1	2	3	4	5	6	7

- elements in A are referenced by their index
- no other way to access elements in an array
- Array indexing is 0-based, meaning the first element is found in position zero

A[0] = 44

A[1] = 12

A[4] = 6

A[7] = 9

A[9] => index out of bounds!

giới hạn

Dynamic arrays

- the dynamic arrays can grow n shrink in size

thu nhỏ

A = 34 | 4

A.add(-7) | A = 34 | 4 | -7

A.add(34) | A = 34 | 4 | -7 | 34

A.remove(4) | A = 34 | -7 | 34

- suppose we create a dynamic array with an initial capacity of two n then begin adding elements to it.

Ø | Ø 7 | Ø 7 | -9

7 | -9 | 3 | Ø 7 | -9 | 3 | 12

7 | -9 | 3 | 12 | 5 | Ø | Ø | Ø | Ø

7 | -9 | 3 | 12 | 5 | 4 | Ø | Ø | Ø

- Create an integer array with the number of elements entered by user

```
n = int(input("Nhập số phần tử của mảng: "))
arr = []

for i in range(n):
    element = int(input("Nhập phần tử thứ {}: ".format(i + 1)))
    arr.append(element)

print("Mảng đã tạo:", arr)
```

- Inserts an element with a specified value n position into array.

```
value = int(input("Nhập giá trị phần tử cần chèn: "))
index = int(input("Nhập vị trí cần chèn: "))

arr.insert(index, value)

print("Mảng sau khi chèn:", arr)
```

- Remove an element at position

```
index = int(input("Nhập vị trí cần xóa: "))

del arr[index]

print("Mảng sau khi xóa:", arr)
```

- Remove an elements with value

```
value = int(input("Nhập giá trị phần tử cần xóa: "))

arr.remove(value)

print("Mảng sau khi xóa:", arr)
```

- Specifies the position of the elements with the specified in the array.

```
value = int(input("Nhập giá trị phần tử cần tìm: "))

if value in arr:
    index = arr.index(value)
    print("Vị trí của phần tử {} trong mảng: {}".format(value, index))
else:
    print("Phần tử {} không tồn tại trong mảng.".format(value))
```

Linked List

Linked list : is a sequential list of nodes that hold data which point to other nodes also containing data

tuần tự?

Where are linked list used?

- used in many List, Queue & Stack implementations.
- Correct for creating circular lists.
- Used in separate chaining, which is present certain Hash tables implementations to deal with hashing collisions.

Sự liên kết

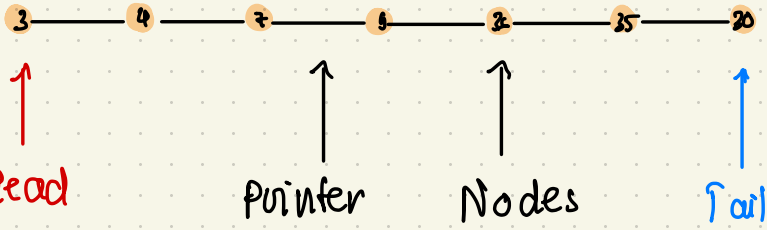
Terminology

Head : the first node in a linked list

Tail : the last node in a linked list

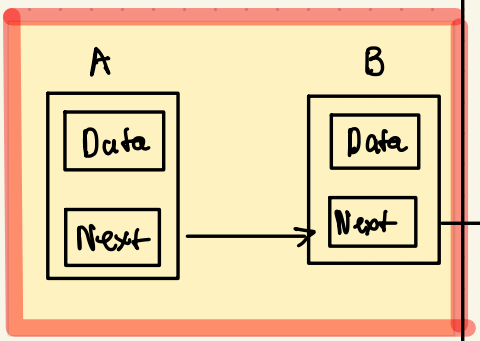
Pointer : Reference to another node

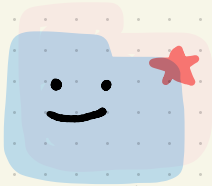
Node : An objects containing data n pointer(s)



Singly linked list

- each node has data and a pointer to the next node.

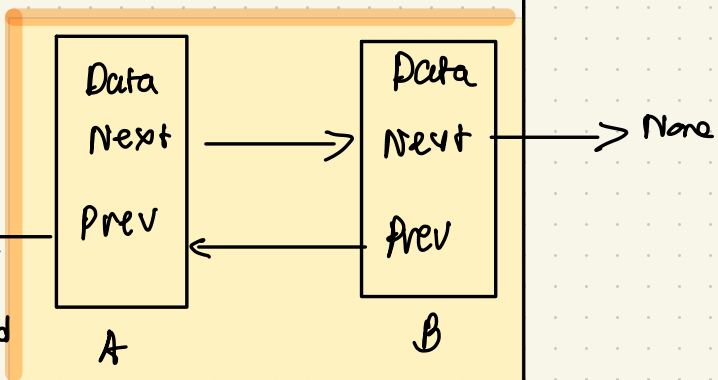




Doubly linked list

add a pointer to the prev node in a doubly-linked list

→ forward or backward

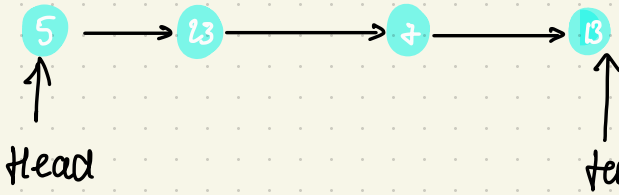


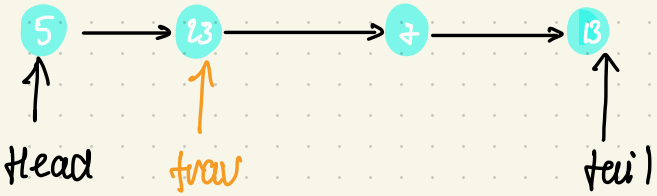
Singly n doubly linked lists

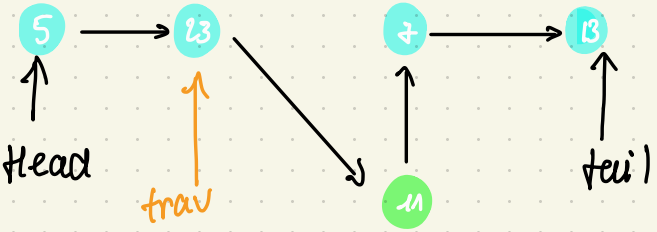
	Pros	Cons
Singly Linked	Uses less memory Simpler implementation	Cannot easily access previous elements
Doubly Linked	Can be traversed backwards	Takes 2x memory

Insert singly linked lists

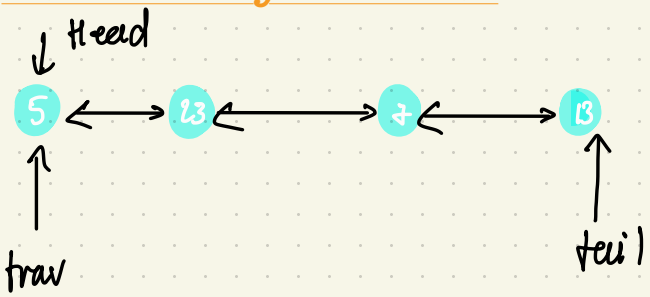
insert 11 where the third node is

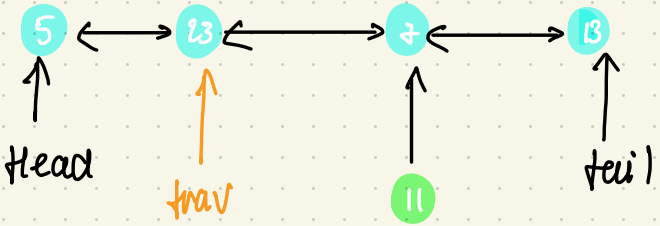


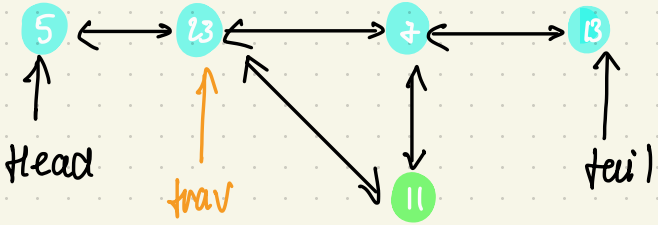




Insert doubly linked list

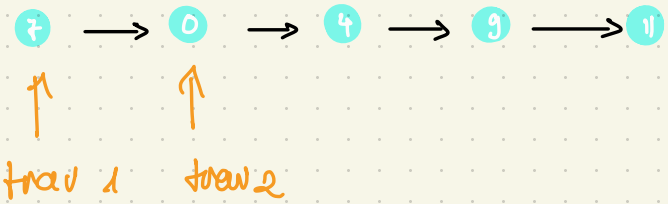


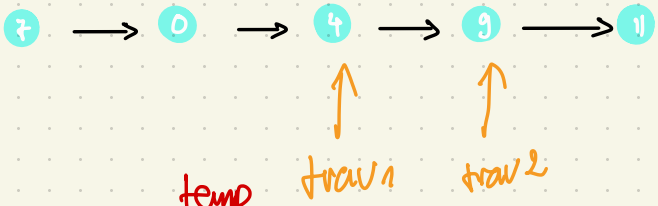


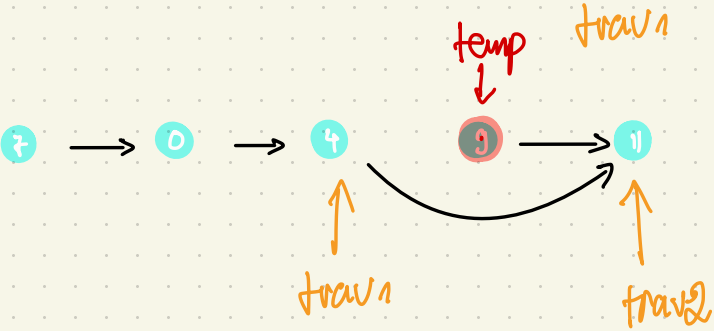


Removing from singly linked list

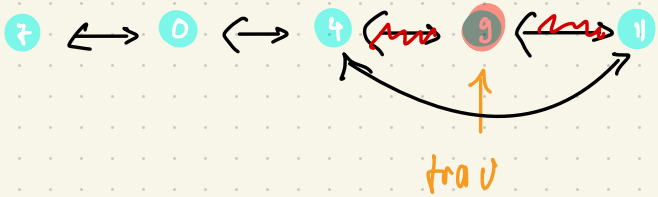
remove 9 from the following SLL







Removing from doubly linked list



Complexity

	Singly Linked	Doubly Linked
Search	O(n)	O(n)
Insert at head	O(1)	O(1)
Insert at tail	O(1)	O(1)

Complexity

	Singly Linked	Doubly Linked
Remove at head	O(1)	O(1)
Remove at tail	O(n)	O(1)
Remove in middle	O(n)	O(n)

insertion :

```
def append_at_a_location(self, data):
    current = self.head
    prev = self.head
    node = Node(data)
    while current:
        if current.data == data:
            node.next = current
            prev.next = node
        prev = current
        current = current.next
```

We can now use ' the preceding code to insert a new node at an intermediate position:

```
words.append_at_a_location('ham')
current = words.head
while current:
    print(current.data)
    current = current.next
```

Search

```
def search(self, data):
    for node in self.iter():
        if data == node:
            return True
    return False
```

```
print(words.search('sspam'))
print(words.search('spam'))
```

Output:
False
True

```
def delete(self, data):
    current = self.head
    prev = self.head
    while current:
        if current.data == data:
            if current == self.head:
                self.head = current.next
            else:
                prev.next = current.next
            self.size -= 1
            return
        prev = current
        current = current.next
```

The worst-case time complexity of the delete operation is O(n)

Assuming that we already have a linked list of three items – *eggs*, *ham*, *spam*, the following code is for deleting a data element with the value "*ham*" from the given linked list:

```
words.delete("ham")
current = words.head
while current:
    print(current.data)
    current = current.next
```

Output:

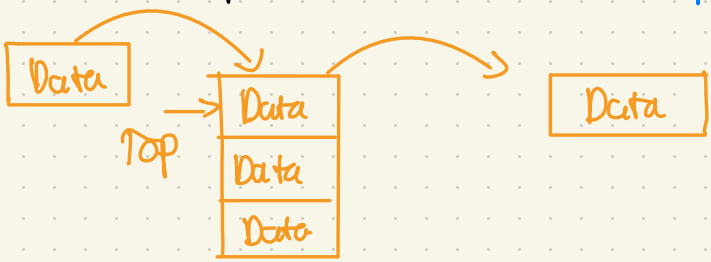
eggs
spam

Stack

What is a stack

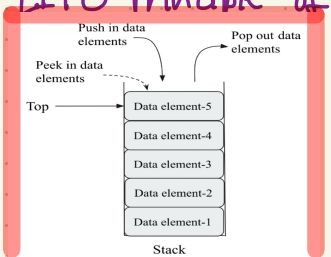
- a stack is a one-ended linear data structure which models a real world stack by having two primary operations, namely push n pop

cấu trúc dữ liệu 1 đầu



Instructions

- Using LIFO Principle of Stack



When & where is a stack used?

- Used by undo mechanisms in text editors.
- Used in compiler syntax check for matching brackets and braces.
- Can be used to model a pile of books or plates.

Complexity	
Pushing	O(1)
Popping	O(1)
Peeking	O(1)
Searching	O(n)
Size	O(1)

Example: Brackets:

[C{}](){}

Current Bracket: [C {

Reversed Bracket:] C }

x loại vì = reversed loại tiếp

Current Bracket: (

Reversed Bracket:)

— loại

— loại tiếp

```
Let S be a stack
For bracket in bracket_string:

    rev = getReversedBracket(bracket)

    If isLeftBracket(bracket):
        S.push(bracket)

    Else If S.isEmpty() or S.pop() != rev:
        return false // Invalid

return S.isEmpty() // Valid if S is empty
```

Applications of stacks

- **Arithmetic expressions and Polish notation**
  - the way to write arithmetic expression is known as a notation
  - an arithmetic expression can be written in 3 different : — Infix, Prefix, Postfix

tiền tố
hậu tố

HASHTAG

Expression No	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

AIR MAIL 21/10/2021 AIR MAIL

Eg :

infix	prefix	postfix
2 + 3	+ 2 3	2 3 +
p - q	- p q	p q -

$(A+B^{**}C)*D/E-(F+G)$

=) $-/*+A^{**}BC\ DE+FG$ Prefix

=) $ABC^{**}+D*E/FG+-$ postfix

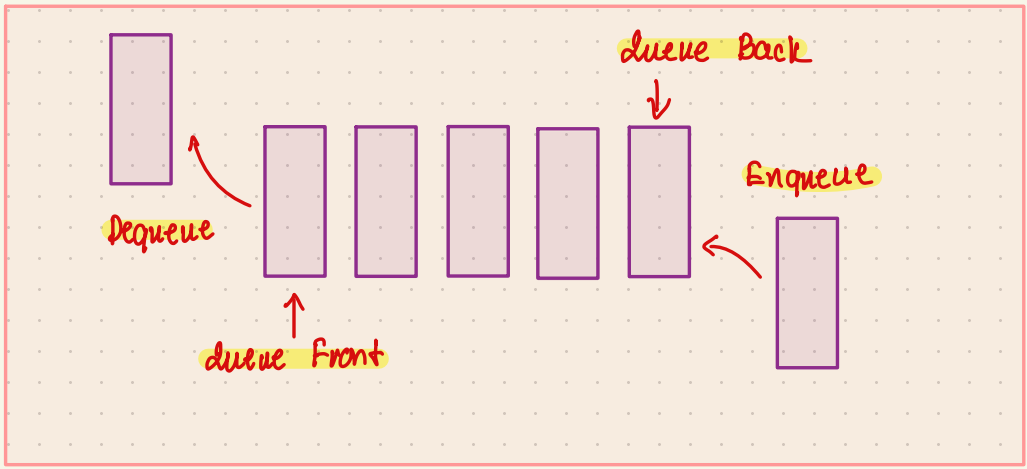
$((A+B)*D)^{**}(E-F)$

=) $**-+AB\ D-EF$ prev

=) $AB+D*EF-**$

Queue

- a queue is a linear data structure which model real world queus by having 2 primary operations namely enqueue and dequeue.



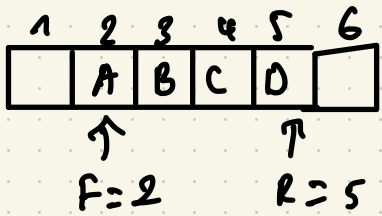
when and where is a Queue used?

- any waiting line models a queue, for example a lineup at movie threatre.
- can be used to efficently keep track of the X most recently added elements.
- web sever request management where u want first come first serve.
- Breath first search (BFS) graph traversal

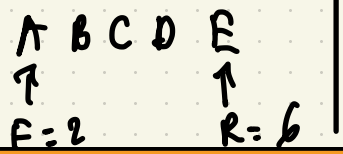
Complexity

Enqueue	O(1)
Dequeue	O(1)
Peeking	O(1)
Contains	O(n)
Removal	O(n)
Is Empty	O(1)

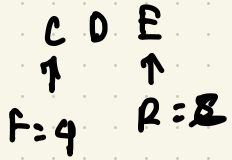
Example:



Add E:



Remove 2 elements




```python
class Queue:

    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.queue:
            raise IndexError("Queue is empty")
        return self.queue.pop(0)

    def is_empty(self):
        return not self.queue

    def size(self):
        return len(self.queue)


if __name__ == "__main__":
    queue = Queue()
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print(queue.queue)
    print(queue.dequeue())
    print(queue.queue)
```

```python
# Tạo một hàng đợi mới
queue = Queue()

# Thêm phần tử vào hàng đợi
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

# Xem phần tử đầu tiên của hàng đợi
print(queue.peek())  # Output: 1

# Xóa và trả về phần tử đầu tiên của hàng đợi
print(queue.dequeue())  # Output: 1

# Kiểm tra xem hàng đợi có trống hay không
print(queue.is_empty())  # Output: False

# Xem số lượng phần tử trong hàng đợi
print(queue.size())  # Output: 2
```

25 July

Chapter 3: Sorting

Outline

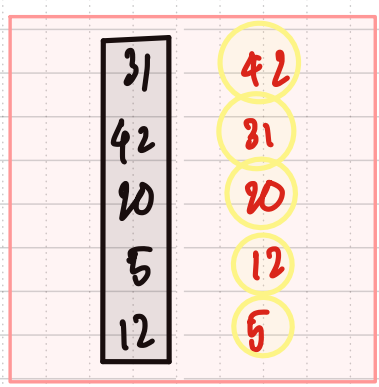
- bumble sort
- insertion sort
- slection sort
- Quick sort
- Tim sort

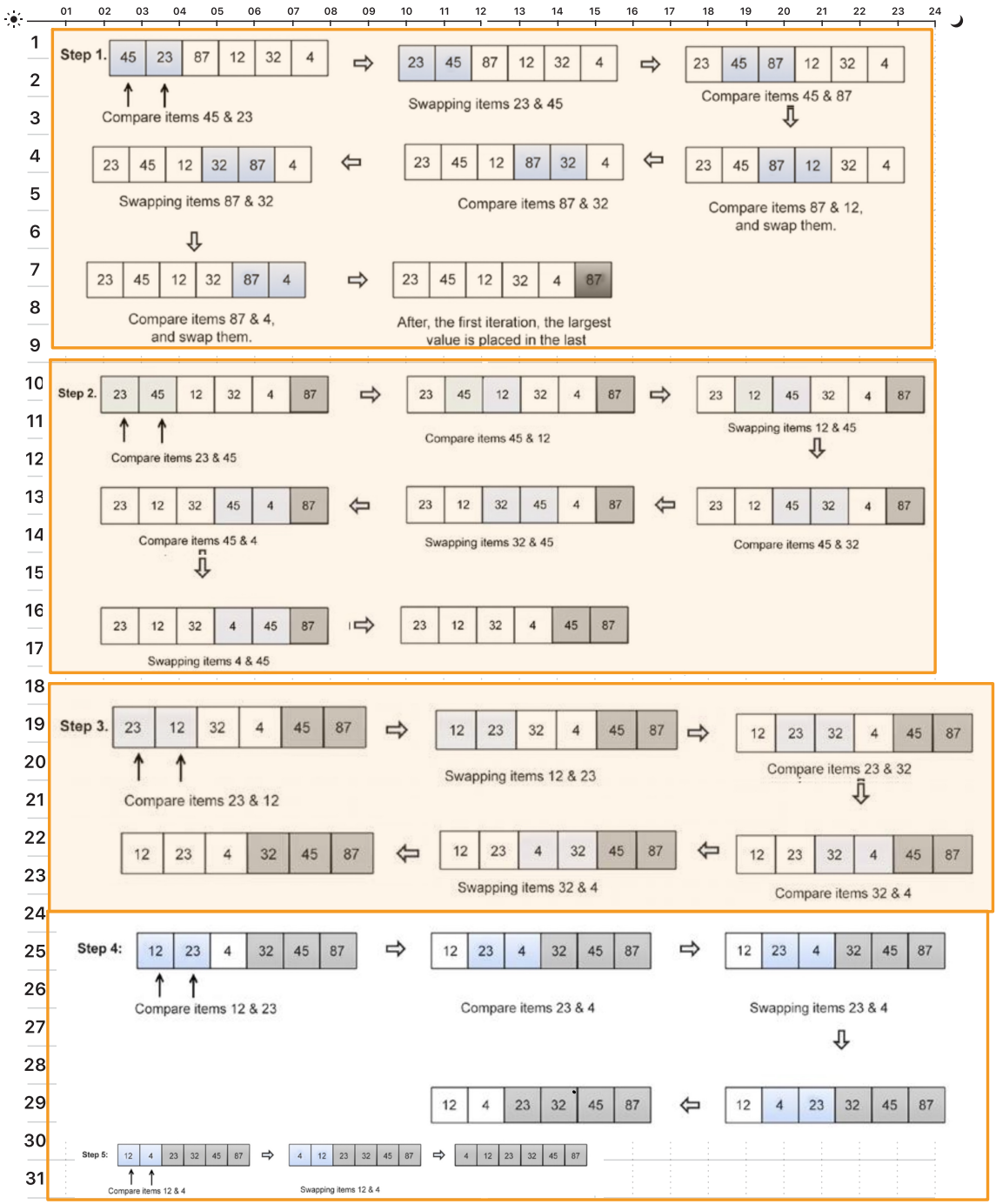
Sorting will be done based on the value of this key

Bumble Sort

- Given an unordered list, we compared adjacent elements, after that, place them in right order according to their values

- This process is repeated $n-1$ times for a list of n




Step 1.
Compare items 45 & 23
Swapping items 23 & 45
Compare items 45 & 87
Compare items 87 & 12, and swap them.
Compare items 87 & 32
Swapping items 87 & 32
Compare items 87 & 4, and swap them.
After, the first iteration, the largest value is placed in the last
Step 2.
Compare items 23 & 45
Compare items 45 & 12
Swapping items 12 & 45
Compare items 45 & 32
Swapping items 32 & 45
Compare items 45 & 4
Swapping items 4 & 45
Step 3.
Compare items 23 & 12
Swapping items 12 & 23
Compare items 23 & 32
Compare items 32 & 4
Swapping items 32 & 4
Step 4:
Compare items 12 & 23
Compare items 23 & 4
Swapping items 23 & 4
Step 5:
Compare items 12 & 4
Swapping items 12 & 4

```python
def bubble_sort(a):
    swapped = True
    while swapped:
        swapped = False
        for i in range(len(a) - 1):
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True

    return a

a=[23,43,2,7,9,3]
sorted_array=bubble_sort(a)

print("sorted arrays: ", sorted_array)
```

// while <dk>:
end while

insert_value = unsorted list[ ]

for i in range (1, len (a))

i = [ ]

j = i

while j > 0 and a[j] < a[j] - 1

swap

j - = 1 ;

b < a ak.

=> b, a = a, b

b > 0

```
for (int i=len(a); i > 0 ,i--)
if ( j >0 && a[j]<a[j-1]
```

Insertion sort

The idea of insertion sort is that maintain 2 sublists, one that is sorted n 1 that is not

We take elements from the unsorted sublist

-> insert them in the correct position in sorted

The insertion sort algorithms:

- Start with 1 element, taking -> sorted
- Take elements 1 by 1 from unsorted n place them at the correct position in sorted list.

Eg : Sort a list {45, 23, 87, 12, 32, 4}

45 -> 23, 45, -> 23, 45, 87

-> 12, 23, 45, 87

-> 4, 12, 23, 45, 87

```python
def insertion_sort(a):
    for i in range(1, len(a)):
        j = i
        while j > 0 and a[j] < a[j - 1]:
            a[j], a[j - 1] = a[j - 1], a[j]
            j -= 1
    return a
if __name__ == "__main__":
    a = [10, 8, 7, 6, 5, 4, 3, 2, 1]
    print("Unsorted array:", a)
    sorted_array = insertion_sort(a)
    print("Sorted array:", sorted_array)
```

mỗi lần j giảm thì xét từng phần tử j-1

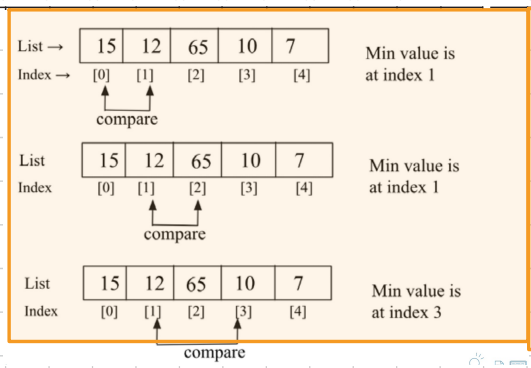
j-1 > j

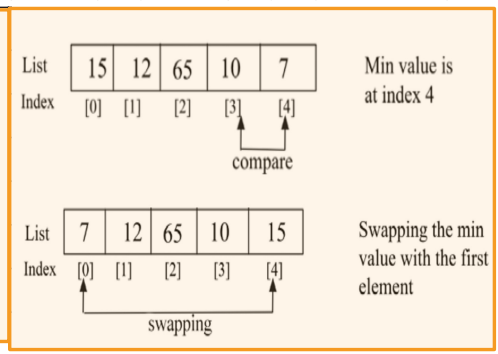
Selection sort

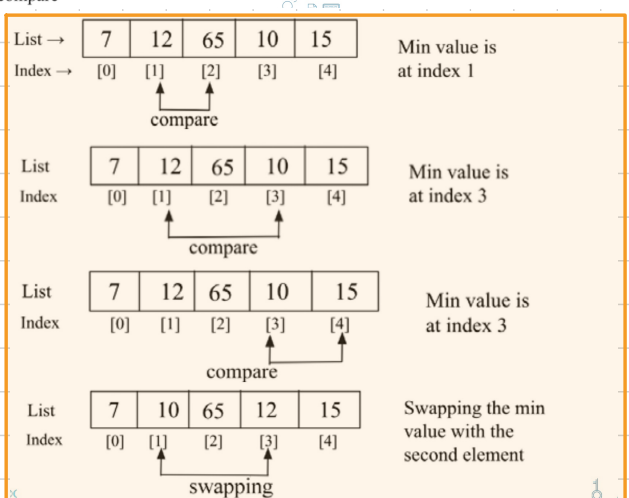
- The selection sort begins by finding the smallest list and interchanges it with data stored at the first position in the list.
  Thus, sorted up to the first element.
  → repeated for (n - 1) times.

- Next, the second smallest element
  => identified n interchanges with second
  => 2 elements sorted.

- The process is repeated, swapped with element in the third indx => the first 3 elements now sorted.








```python
def selection_sort(array):

  for i in range(len(array)):
    min_index = i
    for j in range(i + 1, len(array)):
      if array[j] < array[min_index]:
        min_index = j

    array[i], array[min_index] = array[min_index], array[i]

  return array

if __name__ == "__main__":
  array = [10, 5, 2, 1, 8, 7, 6, 3, 4]
  sorted_array = selection_sort(array)
  print(sorted_array)
```

i chạy hết (a)

} min = j

Example :

13, 21, 15, 3, (12), 9, 14, 7, 6

pivot

Step 1: 13, 21, 15 > 12 dem vào

3 < 12 → [3]

Step 2: [3], [9], [7], [6], [12], [13], [21], [15], [14]

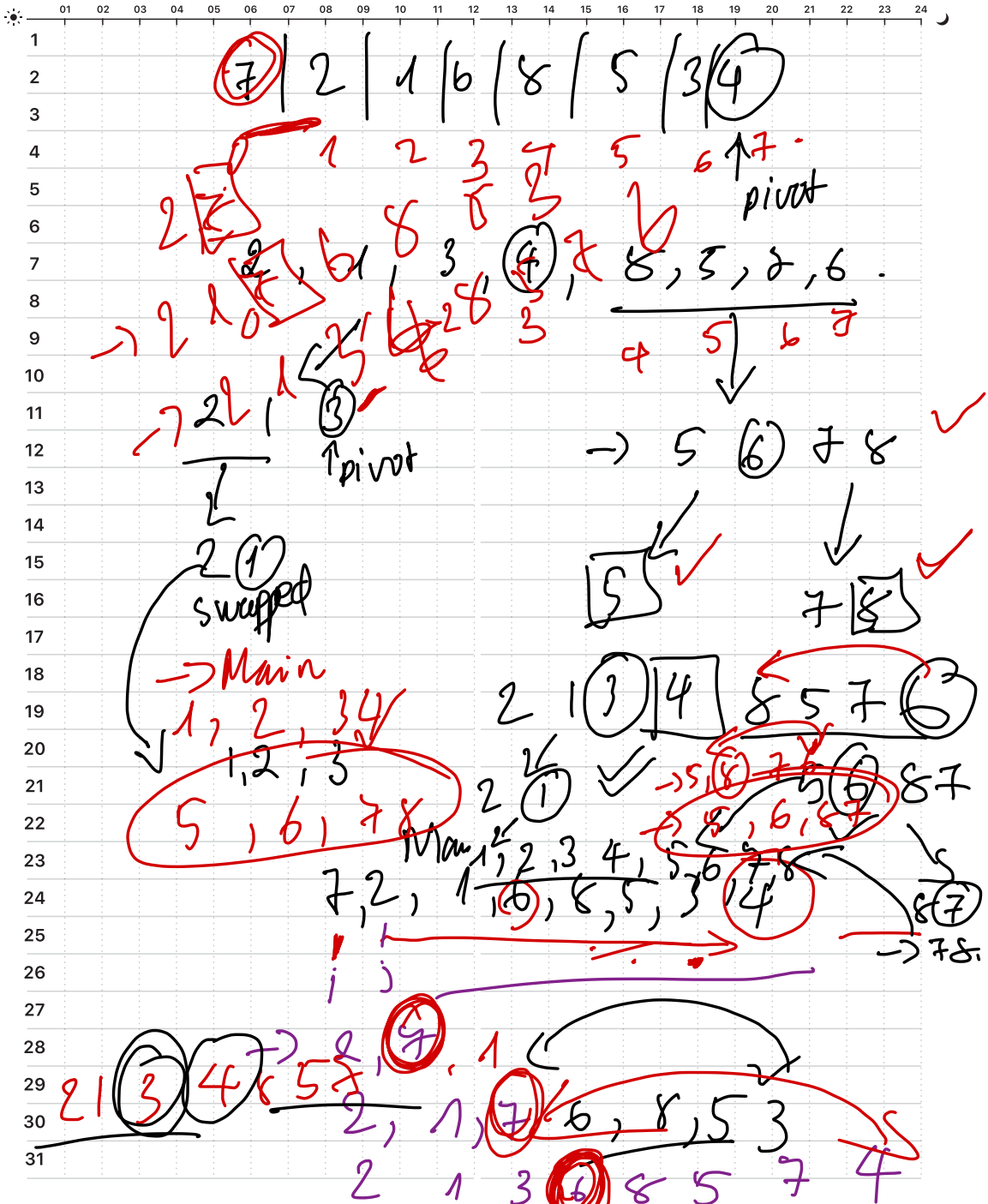
Sắp xếp: 3, 9, 7, 6, (12), 13, 21, 15, 14

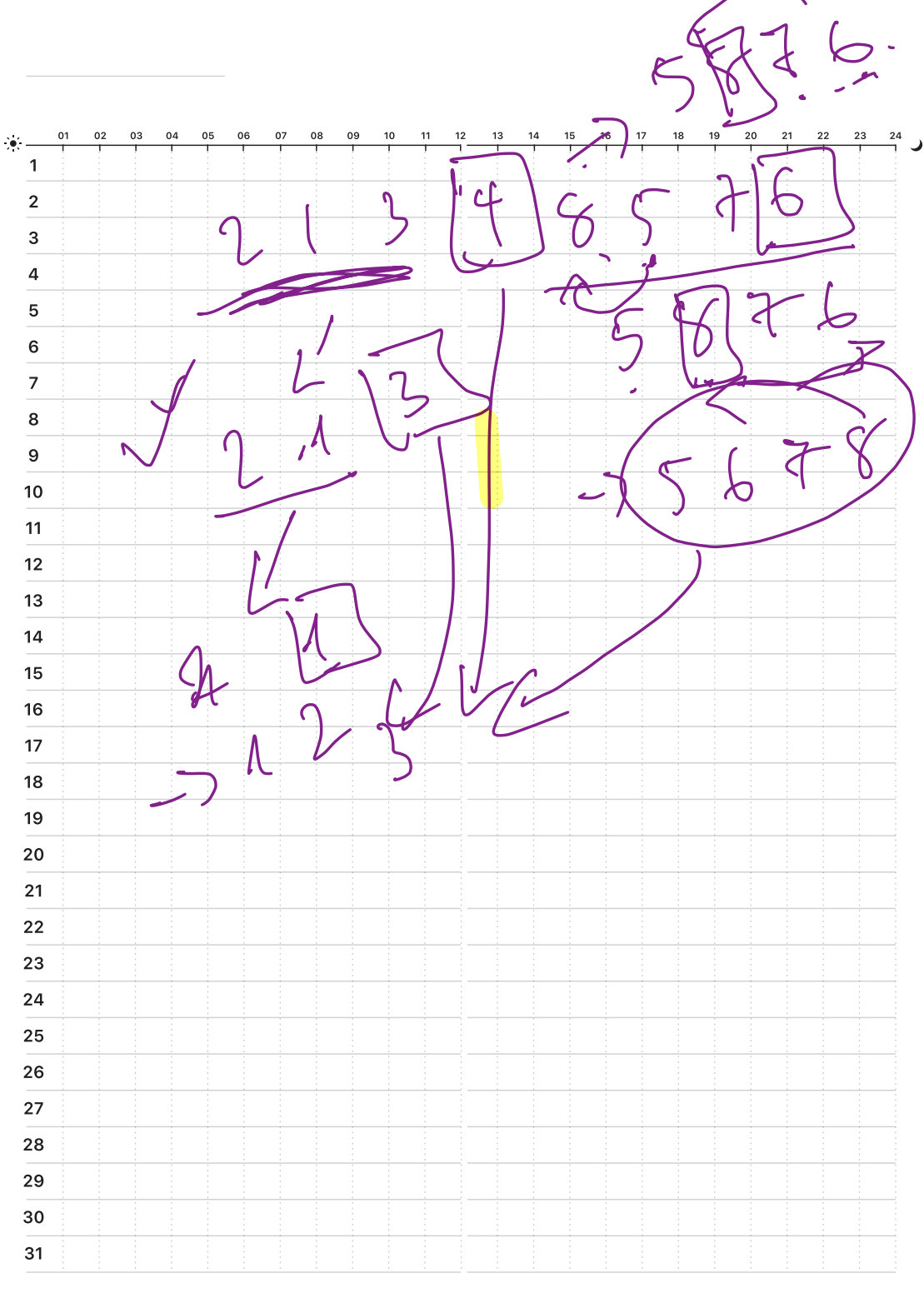
Step 2: 3 9 (7) 6, 12, 13, 21 (15) 14

3, 14, 15, 21

Step 3:

```python
def swap(a, b):
  temp = a
  a = b
  b = temp

def partition(arr, low, high):
  pivot = arr[high]
  i = (low - 1)

  for j in range(low, high):
    if arr[j] <= pivot:
      i += 1
      swap(arr[i], arr[j])

  swap(arr[i + 1], arr[high])
  return (i + 1)

def quickSort(arr, low, high):
  if low < high:
    pi = partition(arr, low, high)

    quickSort(arr, low, pi - 1)
    quickSort(arr, pi + 1, high)

def main():
  arr = [10, 8, 7, 6, 5, 4, 3, 2, 1]
  n = len(arr)

  quickSort(arr, 0, n - 1)

  for i in range(n):
    print(arr[i])

if __name__ == "__main__":
  main()
```


pivot
swapped
Main
pivot



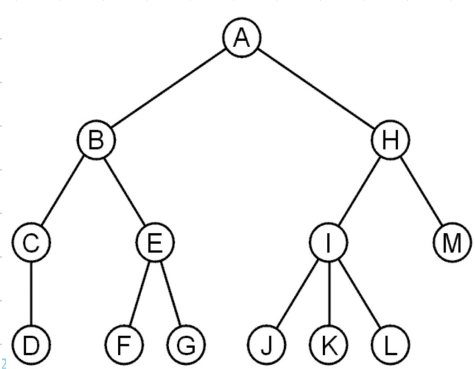
Chapter 4: Trees

Out Lines:

- Basic concepts
- Tree traversal
- Binary trees
- Binary search trees
- Balanced binary trees

1. Basic concepts.

- A tree is a non-linear data structure as there is a parent-child relationship betweens the items.
- The top of the tree is known as **root node**.
- There should not be any cycle among the nodes...

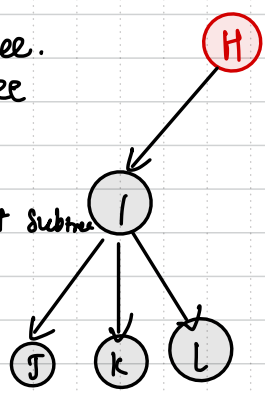


- **Node:** Each circled letter in the preceding diagram represents a node

- **Root node:**
  - is the first node from which all orther nodes
  - doesn't have a parent node
  - always one unique root node.

- **Edge:** the connection among 2 nodes
  the total number of edges will be maximum of 1 less than the total nodes in the trees.

- **Subtree:** is a tree whose nodes descend from other tree.
  Eg: H, I, L form a subtree



- **Parent:** the parent node of that subtree
  Eg: H is parent I
  I parent J, K, L

Child:
- descendant from parent node
- for all node orther them the root node 1 parent node.

- **Degree** — the total number of children
  - a tree consisting of only 1 node has degree of 0.
  - deg (A) = 2
  - deg (I) = 3
  - deg (J) = 0

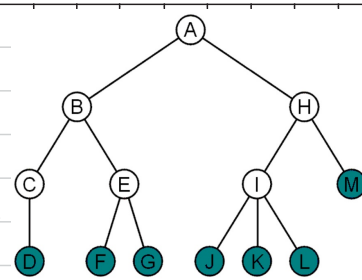
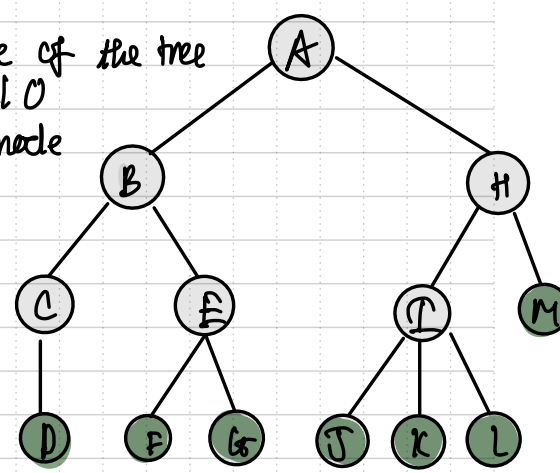


Figure 4: Example tree with leaf nodes

- **Leaf node**
  - does not have any children and is the terminal node
  - all other nodes are said to be internal nodes, that is, they are internal to tree
  - the degree of the leaf node is always 0
- **Sibling** — same parent node & siblings
  - Eg: J, K, L
- **Level**
  - the root node of the tree is to be at level 0
  - the children node at level 1
  - when at 1 → 2

Eg: A is level 0
B, H level 1
C E I n M level 2



- **Path** is a sequence of nodes $(a_0, a_1, \ldots a_n)$ where a_{k+1} is a child of a_k
  - the length of this path is n
  Eg: the path (B, E, G) has length 2.

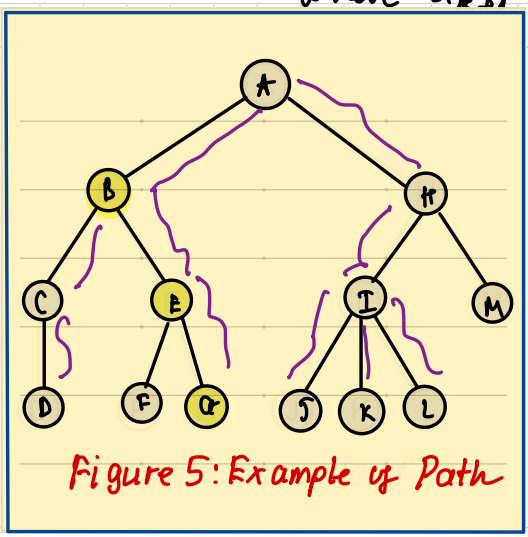


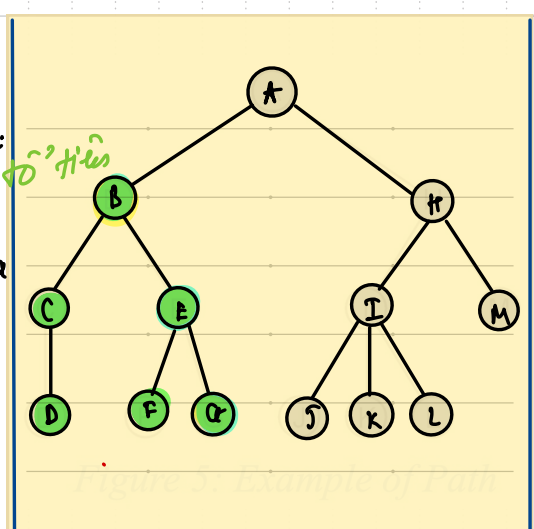
Figure 5: Example of Path

- **Height**: of tree is the total nodes in longest path of the tree
  Eg: the height of the tree is 4.

- **Depth**: is number of edges from the root
  Eg: E has 2
  L has 3.

if a path exists from node a to node b:

- a is an ancestor of b
- b is an descendant of a

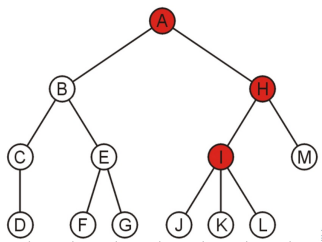


- We can add the adjective strict to exclude equality: a is a strict descendent of b if a is a descendant of b but a ≠ b

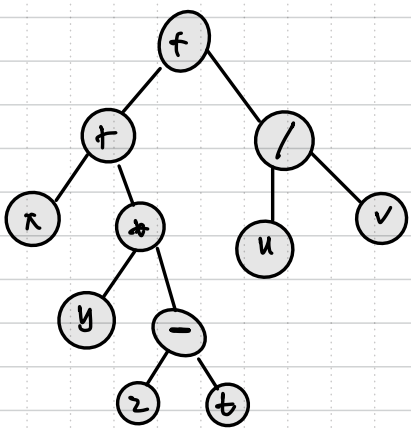
The root node is an ancestor of all nodes

Example: The descendants of node B: B, C, D, E, F, G

The ancestors of node I: I, H, A



Example:



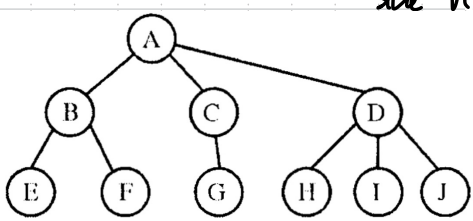
representing arithmetic expression $x+y*(z-t)+v/u$

Figure 7. Tree showing the table of contents of a book chapter

Tree representation

for every node in general tree, it has:
- a leftmost child node (first child)
- a sibling next to the right

assume from left to right
Eg: For node B, the leftmost child is E
the right sibling is C

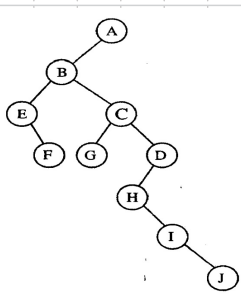


if each node has a specification:

CHILD	DATA	SIBLING

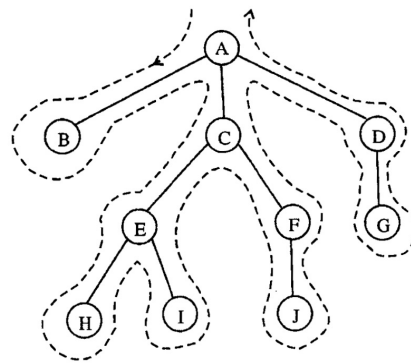
CHILD : pointer to the leftmost node

SIBLING : Pointer to the right next sibling node



Example

- Pre-order: A B C E H I F J D G
- In-order: B A H E I C J F G D
- Post-order: B H I E J F C G D A



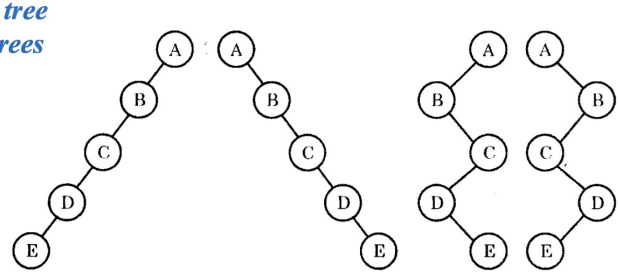
Binary trees

Là 1 dạng quan trọng của cấu trúc cây

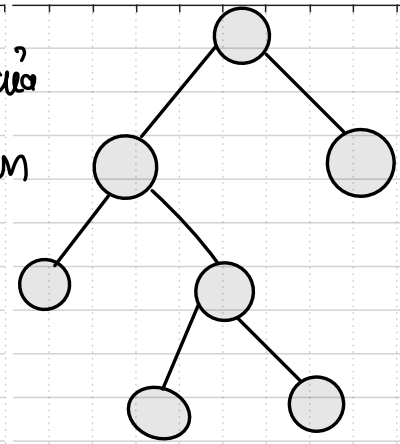
Mỗi nút trên cây có tối đa là 2 cây con

mỗi cây con : < cây con trái
cây con phải

- Tree a) is called a ***left-skewed tree***
- Tree b) is called ***right-skewed tree***
- Trees c), d) are called ***zigzag trees***

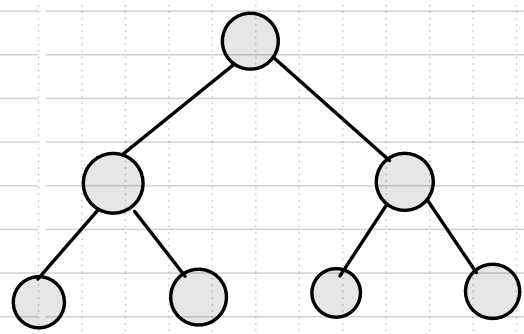


- Nếu tất cả các nút của cây có 0 hoặc 2 con và lá nút nào có 1 con



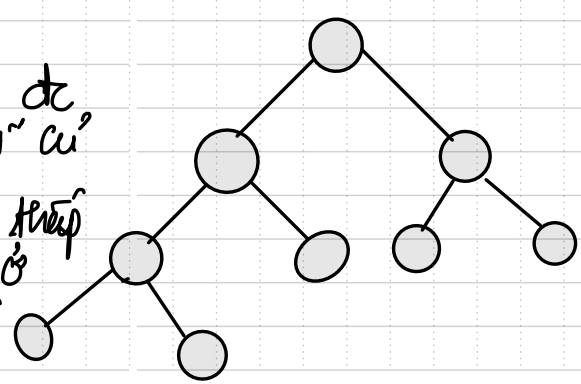
A full binary tree

- Là các cây mà phần dc lấp đầy ở có lá gian trong ở bất kỳ nút nào



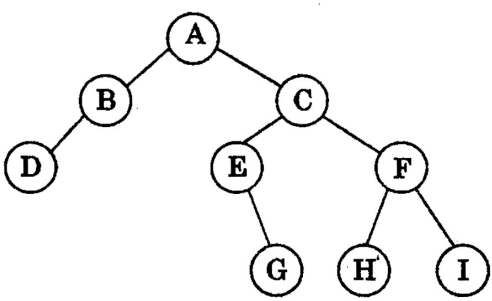
A perfect binary tree

- Cây hoàn chỉnh dc lấp đầy và tất cả các nút ở cấp thấp nhất. Lấp đầy ở phía bên trái

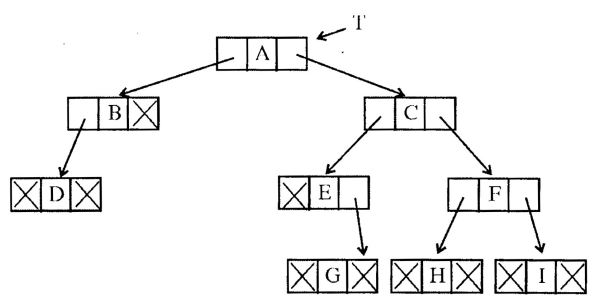


Representing Binary Trees with pointers

LEFT	DATA	RIGHT



Dạng mới mới



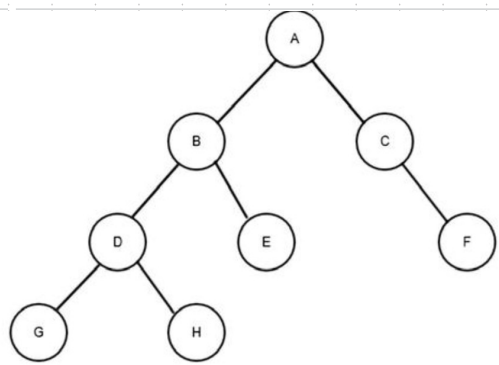
Hình 6.10

Truy cập theo thứ tự

Truy cập theo thứ tự hoạt động như sau: chúng tôi bắt đầu truy cập cây con bên trái một cách đệ quy và một khi cây con bên trái được truy cập, nút gốc sẽ được truy cập và cuối cùng là cây con bên phải được truy cập đệ quy. Nó có ba bước sau:

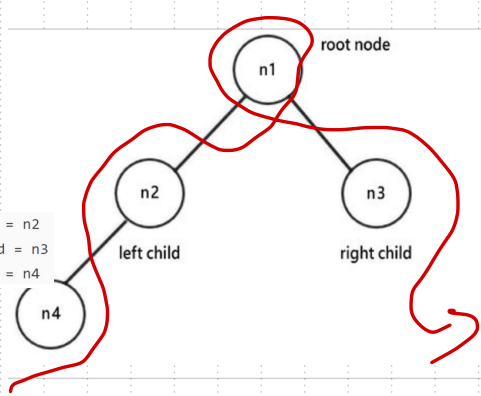
- Chúng tôi bắt đầu đi qua cây con bên trái và gọi một hàm thứ tự đệ quy
- Tiếp theo, chúng tôi truy cập nút gốc
- Cuối cùng, chúng tôi đi qua cây con bên phải và gọi một hàm thứ tự đệ quy

G D H B E A C F




```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None

n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")

def inorder(root_node):
    current = root_node
    if current is None:
        return
    inorder(current.left_child)
    print(current.data)
    inorder(current.right_child)

inorder(n1)
```

```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```



Pre-order traversal

```
def preorder(root_node):
    current = root_node
    if current is None:
        return
    print(current.data)
    preorder(current.left_child)
    preorder(current.right_child)

preorder(n1)
```

First, check if the current node is null or empty. If it is empty, it means the tree is an empty tree, and if the current node is not empty, then we traverse the tree using the pre-order algorithm.

The pre-order traversal algorithm traverses the tree in the order of root, left subtree, and right subtree recursively, as shown in the above code

Output: root node
left child node
left grandchild node
right child node

Post-order :

G H D E B F C A

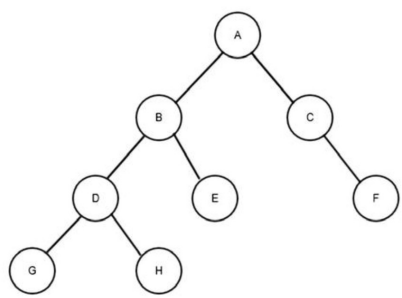


Fig.11. An example binary tree for post-order tree traversal

Post-order traversal

```
def postorder( root_node):
    current = root_node
    if current is None:
        return
    postorder(current.left_child)
    postorder(current.right_child)
    print(current.data)

postorder(n1)
```

First, check if the current node is null or empty. If it is not empty, we traverse the tree using the post-order algorithm as discussed, and finally, when we apply the above post-order traversal algorithm on the above sample tree of four nodes with n1 as the root node.

Output: left grandchild node
left child node
right child node
root node

Level-order traversal

Level 0: 4

level 1: 2 8

level 2: 1 3 5 10

thus, the level-order for this tree is: 4, 2, 8, 1, 3, 5, 10

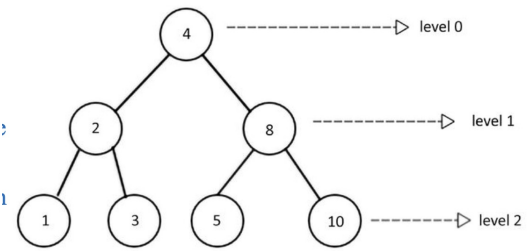
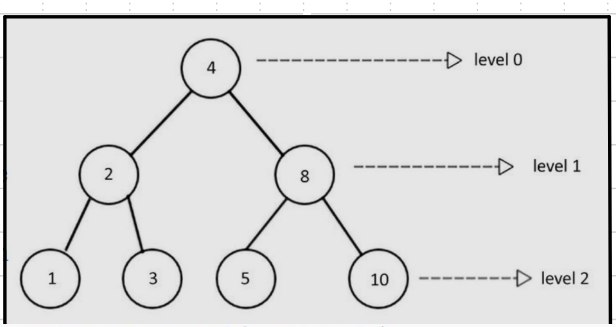


Fig.12. An example binary tree for level-order tree traversal

Using queue data structure.




```
def level_order_traversal(root_node):
    list_of_nodes = []
    traversal_queue = deque([root_node])
    while len(traversal_queue) > 0:
        node = traversal_queue.popleft()
        list_of_nodes.append(node.data)
        if node.left_child:
            traversal_queue.append(node.left_child)
            if node.right_child:
                traversal_queue.append(node.right_child)
    return list_of_nodes

print(level_order_traversal(n1))
```

```
class TreeNode:
    def __init__(self, data=None):
        self.data = data
        self.right = None
        self.left = None
class Stack:
    def __init__(self):
        self.elements = []

    def push(self, item):
        self.elements.append(item)

    def pop(self):
        return self.elements.pop()
```

We will look at building a tree for an expression written in postfix notation. For this, we define a tree node.

Define the stack class

CMC UNIVERSITY Parsing a reverse Polish expression

```
expr = "4 5 + 5 3 - *".split()
stack = Stack()
```

an example of an arithmetic expression and set up the stack

CMC UNIVERSITY Parsing a reverse Polish expression

```
for term in expr:
    if term in "+-*/":
        node = TreeNode(term)
        node.right = stack.pop()
        node.left = stack.pop()
    else:
        node = TreeNode(int(term))
    stack.push(node)
```

To evaluate the expression, we can use the following function

```
def calc(node):
    if node.data == "+":
        return calc(node.left) + calc(node.right)
    elif node.data == "-":
        return calc(node.left) - calc(node.right)
    elif node.data == "*":
        return calc(node.left) * calc(node.right)
    elif node.data == "/":
        return calc(node.left) / calc(node.right)
    else:
        return node.data
```

```
root = stack.pop()
result = calc(root)
print(result)
```

Binary Search Trees

Searching : - find the location of the desired data item from a collection data items.

- thuật toán tìm kiếm trả về giá trị của giá trị đó trên kiểm
if the data item is not present it returns None

different ways in which data can be organzied, affected algorithm search:

sorted { 1, 3, 5, 7, 9 ... }

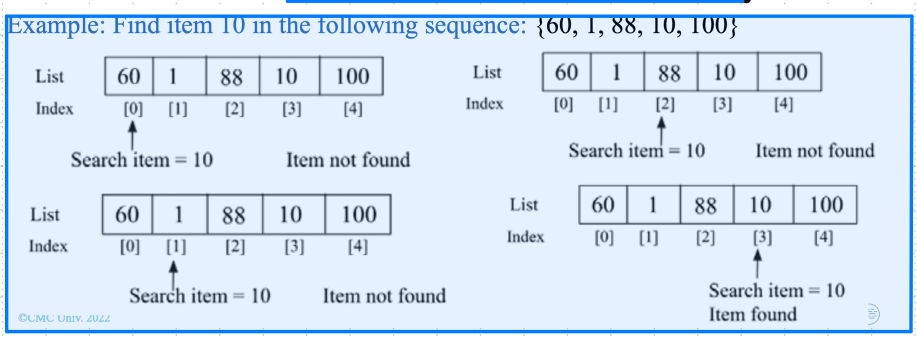
unsorted { 3, 77, 8, 2, 88, 70 }

Linear search:

- use in which the given list of data items is **not sorted**.
- we linearly match the desired of the list **one - by - one** till the end.

Example: Find item 10 in the following sequence

{60, 1, 88, 10, 100}



Program:

```
def search(unordered_list, term):
    for i, item in enumerate(unordered_list):
        if term == unordered_list[i]:
            return i
    return None
list1 = [60, 1, 88, 10, 11, 600]

search_term = 10
index_position = search(list1, search_term)
print(index_position)

list2 = ['packt', 'publish', 'data']
search_term2 = 'data'
Index_position2 = search(list2, search_term2)
print(Index_position2)
```

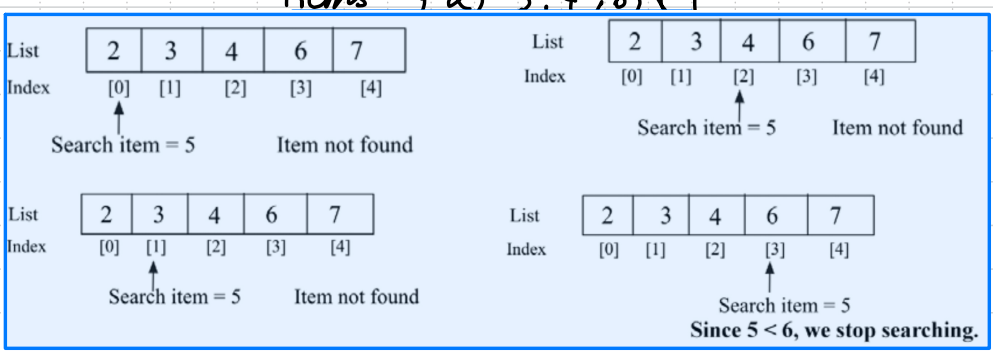
the worst-case time complexity of an unordered linear search is $O(n)$

Ordered linear search

Step : ① Move through the list sequentially

② if the value greater than object then quit and return None.

Example : search for item 5 in the list of items {1, 2, 3, 4, 6, 7}



Program :

```
def search_ordered(ordered_list, term):
    ordered_list_size = len(ordered_list)
    for i in range(ordered_list_size):
        if term == ordered_list[i]:
            return i
        elif ordered_list[i] > term:
            return None
    return None
list1 = [2, 3, 4, 6, 7]

search_term = 5
index_position1 = search_ordered(list1, search_term)

if index_position1 is None:
    print("{} not found".format(search_term))
else:
    print("{} found at position {}".format(search_term, index_position1))
```

the worst-case time complexity of an ordered is $O(n)$

Jump search

- is searching for a given element from an ordered (or sorted) list of elements.
-
  this method devide-and-conquer
  → compare the search value

Algorithm:

- Firstly, divide the sorted list of data into subsets of data elements called blocks. Within each block, the highest value will lie within the last element (the array is sorted)
- Next, start comparing the search value with the last element of each block.

There can be three conditions:

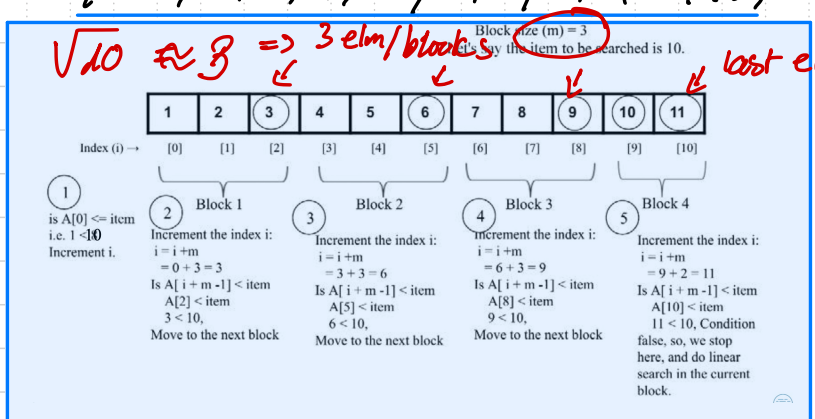
1. If the search value is greater than the last element of the block, compare it with the next block.
2. If the search value is less than the last element of the block, it means the desired search value must be present in the current block, apply linear search in this block and return the index position.
3. If the search value is the same as the compared element of the block, we return the index position of the element and we return the candidate.

the size is taken as $\sqrt{n}$, length n

Example: search for item 10 in the list

{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 }

$\sqrt{10} \approx 3$ => 3 elm/blocks



Program :

```
def linear_search(subA, term):
    print("Entering Linear Search")
    size = len(subA)
    for i in range(size):
        if term == subA[i]:
            return i
        elif subA[i] > term:
            return -1
    return -1
def jump_search(A, item):
    import math
    print("Entering Jump Search")
    sizeA = len(A)
    block_size = int(math.sqrt(sizeA))
    i = 0
    while i != len(A)-1 and A[i] <= item:
        print("Block under consideration - {}".format(A[i:i
            +block_size]))
        if i+ block_size > len(A):
            block_size = len(A) - i
            block_list = A[i: i+block_size]
            j = linear_search(block_list, item)
            if j == -1:
                print("Element not found")
                return
            return i + j
        if A[i + block_size -1] == item:
            return i+block_size-1
        elif A[i + block_size - 1] > item:
            block_array = A[i: i + block_size - 1]
            j = linear_search(block_array, item)
            if j == -1:
                print("Element not found")
                return
            return i + j
        i += block_size

print(jump_search([1,2,3,4,5,6,7,8,9, 10, 11], 10))
```

Output

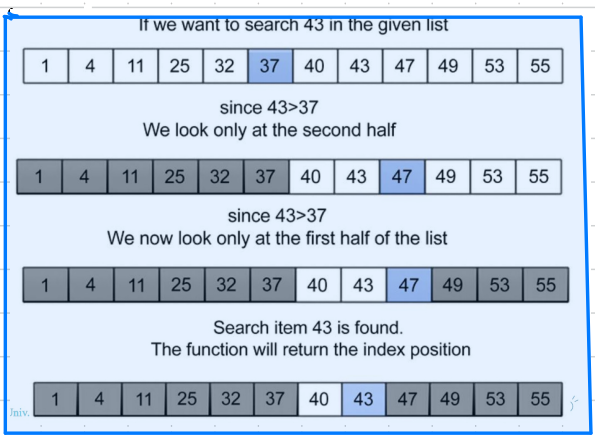
```
Entering Jump Search
Block under consideration - [1, 2, 3]
Block under consideration - [4, 5, 6]
Block under consideration - [7, 8, 9]
Block under consideration - [10, 11]
Entering Linear Search
9
```


the worst-case time complexity will be $O(\sqrt{n})$

Binary Search

- finds a given item from the given sorted list
- it starts searching by dividing the given list in half
  - if the search item is smaller than the middle value then it look for the searched item only in the first half
  - if the search item is greater than the middle value it will only look at the second half the list
- repeat the same process every time until we find the search item or we have checked the whole list.

Example: search for item 43 from a list of 12 items



Program =

```
def binary_search_iterative(ordered_list, term):
    size_of_list = len(ordered_list) - 1
    index_of_first_element = 0
    index_of_last_element = size_of_list
    while index_of_first_element <= index_of_last_element:
        mid_point = (index_of_first_element + index_of_last_element)/2
        if ordered_list[mid_point] == term:
            return mid_point
        if term > ordered_list[mid_point]:
            index_of_first_element = mid_point + 1
        else:
            index_of_last_element = mid_point - 1
    if index_of_first_element > index_of_last_element:
        return None
list1 = [10, 30, 100, 120, 500]

search_term = 10
index_position1 = binary_search_iterative(list1, search_term)
if index_position1 is None:
    print("The data item {} is not found".format(search_term))
else:
    print("The data item {} is found at position {}".format(search_term,
index_position1))
```

$n/2^k = 1$

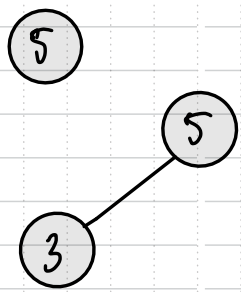
$\Rightarrow k = \log_2(n)$

the binary search algorithm has the worst-case time complexity of $O(\log n)$

Binary search tree operations

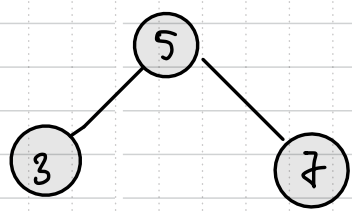
Inserting nodes :

Example : create a binary search tree by inserting data items 5, 3, 7, 1.



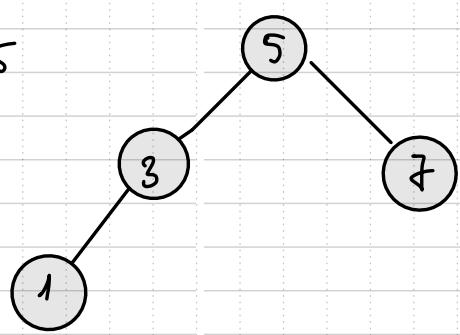
Step 2: 3 < 5

Step 3: 7 > 5



Step 4 : 1 < 5

1 < 3



Program

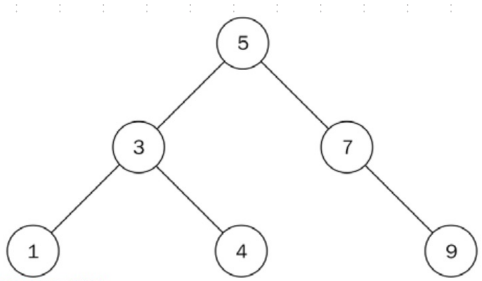
```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None

class Tree:
    def __init__(self):
        self.root_node = None
```

```
def insert(self, data):
    node = Node(data)
    if self.root_node is None:
        self.root_node = node
        return self.root_node
    else:
        current = self.root_node
        parent = None
        while True:
            parent = current
            if node.data < parent.data:
                current = current.left_child
                if current is None:
                    parent.left_child = node
                    return self.root_node
            else:
                current = current.right_child
                if current is None:
                    parent.right_child = node
                    return self.root_node
```

```
def inorder(self, root_node):
    current = root_node
    if current is None:
        return
    self.inorder(current.left_child)
    print(current.data)
    self.inorder(current.right_child)
```

tree = Tree()
r = tree.insert(5)

3
7
9
1
4



=> output :

1
3
4
5
7
9

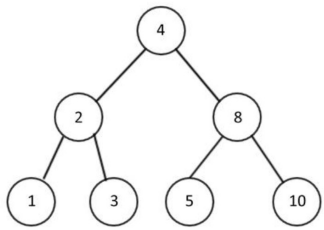
Searching the tree

Searching the tree

Example: binary search tree that has nodes 1, 2, 3, 4, 8, 5,10, as shown in the Figure.

Search for a node with a value of 5:

- start from the root node 4, 4 < 5
- move to the right subtree 8, 8 > 5
- move to the left subtree 5, 5 = 5, return "item found".



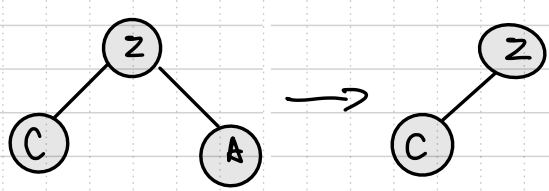
Program :

```
def search(self, data):
    current = self.root_node
    while True:
        if current is None:
            print("Item not found")
            return None
        elif current.data is data:
            print("Item found", data)
            return data
        elif current.data > data:
            current = current.left_child
        else:
            current = current.right_child
```

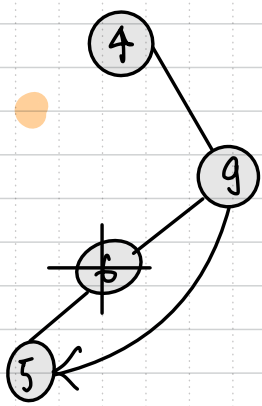
```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)

tree.search(9)
```

Deleting nodes

(1) No children : directly remove the node



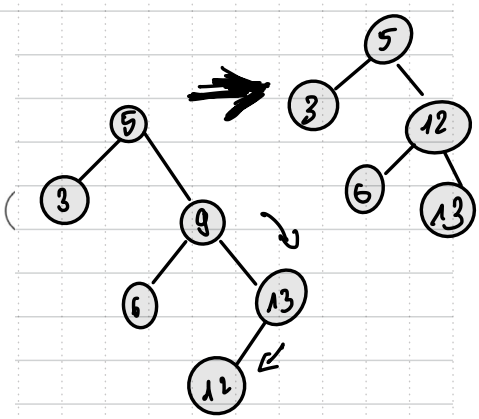
(2) One child : swap the value of that node with its child then delete node



(3) two children : first find the in-order successor or predecessor swap their values then delete node

Example : delete node 9 in the following figure.

- Find a successor node of node 9: the node that has the minimum value in the right subtree, node 12 (the first element when apply the in-order traversal on the right subtree of the node 9)
- Move the content of the successor node 12 into the node 9 to be deleted.
- Delete the successor node 12



Program

```
def get_node_with_parent(self, data):
    parent = None
    current = self.root_node
    if current is None:
        return (parent, None)
    while True:
        if current.data == data:
            return (parent, current)
        elif current.data > data:
            parent = current
            current = current.left_child
        else:
            parent = current
            current = current.right_child

    return (parent, current)
```

Code to get from parent

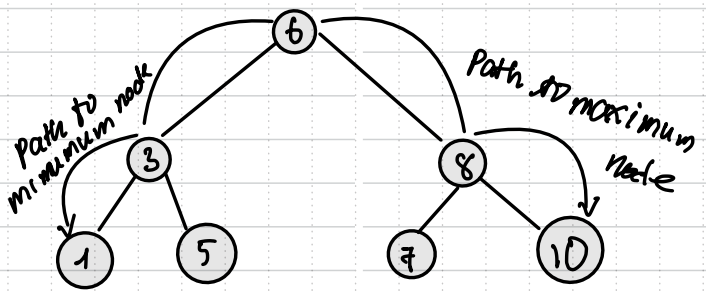
```
def remove(self, data):
    parent, node = self.get_node_with_parent(data)

    if parent is None and node is None:
        return False

    # Get children count
    children_count = 0

    if node.left_child and node.right_child:
        children_count = 2
    elif (node.left_child is None) and (node.right_child is None):
        children_count = 0
    else:
        children_count = 1
```

```
if children_count == 0:
    if parent:
        if parent.right_child is node:
            parent.right_child = None
        else:
            parent.left_child = None
    else:
        self.root_node = None
```

```
else:
    parent_of_leftmost_node = node
    leftmost_node = node.right_child
    while leftmost_node.left_child:
        parent_of_leftmost_node = leftmost_node
        leftmost_node = leftmost_node.left_child

    node.data = leftmost_node.data

    if parent_of_leftmost_node.left_child == leftmost_node:
        parent_of_leftmost_node.left_child = leftmost_node.right_

    else:
        parent_of_leftmost_node.right_child = leftmost_node.right
```

in the worst case scenario its take $O(h)$ where h is the height of the tree

Finding the minimum and maximum nodes.



Program:

```
def find_min(self):
    current = self.root_node
    while current.left_child:
        current = current.left_child

    return current.data

def find_max(self):
    current = self.root_node
    while current.right_child:
        current = current.right_child

    return current.data
```

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)
print(tree.find_min())
print(tree.find_max())
```

Chapter 5: Graphs

Outline :- Basic concepts
- graph representations
- graph traversals
- shortest paths
- minimum spanning tree

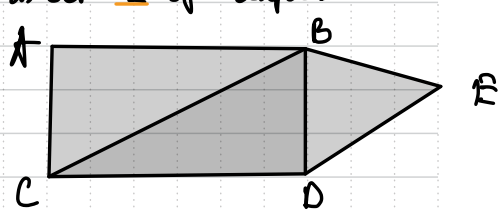
1. Basic concepts

- **A graph** is set of a finite number of vertices n edges, in which the edges are the links between vertices, n each edge in a graph joins two distinct nodes

- **A graph** is a formal mathematical representation of a network

a graph $G = (V, E)$
- in set V of vertices
- in set E of edges.

Example: Graph $G = (V, E)$



- $V = \{A, B, C, D, E\}$.
- $E = \{\{A,B\}, \{A,C\}, \ldots \{D,E\}\}$

1. Example of a graph

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								
21																								
22																								
23																								
24																								
25																								
26																								
27																								
28																								
29																								
30																								
31																								